Wydawnictwa
Naukowo-
Techniczne

KLASYKA INFORMATYKI

Niklaus Wirth

Algorytmy + struktury danych = programy

Wydawnictwa
Naukowo-
Techniczne
Warszawa

KLASYKA INFORMATYKI

W skład serii „Klasyka Informatyki" wchodzą dzieła najwybitniejszych uczonych świata w dziedzinie informatyki – książki o nieprzemijającej wartości, stanowiące bazę solidnego, klasycznego wykształcenia każdego profesjonalnego informatyka.

Wydawnictwa Naukowo-Techniczne przygotowały tę serię ze szczególną pieczołowitością, powierzając tłumaczenie poszczególnych tomów znakomitym specjalistom. Wyboru książek dokonano w ścisłej współpracy z polskim środowiskiem akademickim, dedykując serię głównie studentom informatyki i młodym pracownikom naukowym.

Niklaus Wirth

Algorytmy + struktury danych = programy

Wydanie szóste

Z angielskiego przełożyli:

Michał Iglewski
Marek Missala
Jerzy Czyżowicz
Jerzy Dąbrowski

Scaned by Dante

O Autorze:

Profesor **Niklaus Wirth** urodził się (1934 r.) w Winterthur, w Szwajcarii. Jest absolwentem (1958 r.) Wydziału Elektrotechniki na Politechnice (ETH) w Zurychu. Ma tytuł magistra, który otrzymał (1960 r.) na Laval University w Kanadzie, i tytuł doktora, który otrzymał (1963 r.) na University of California. Jest współzałożycielem Wydziału Informatyki na Stanford University. W latach 1963-1967 pełnił tam funkcję wicedyrektora. W latach 1968-1981 piastował stanowisko profesora na ETH, gdzie zajmował się pracą dydaktyczną. Obecnie kieruje działem informatyki na tej uczelni.

Z komputerami i programowaniem profesor Wirth zetknął się na wykładach analizy numerycznej w Laval University. Używano tam wówczas komputera Alvac IIIE. Ponieważ był on notorycznie niesprawny, większość ćwiczeń z programowania robiono na papierze. Wirth szybko zrozumiał, że komputery, do których należy przyszłość, muszą być niezawodne i jak najefektywniej oprogramowane. Postanowił poświęcić się programowaniu. W 1960 roku opracował gramatykę precedencyjną, po czym zajął się językami programowania.

Pierwszym stworzonym przez Wirtha językiem był **Euler** – przedstawiony nie tylko na papierze, ale też wdrożony na komputerach IBM 704 i Boroughs B500, stanowiący punkt wyjścia do opracowania języków programowania strukturalnego.

W 1967 roku Wirth wrócił do Szwajcarii i od 1968 roku zajął się **Pascalem**. Kompilator Pascala został przedstawiony w 1970 roku, a języka Pascal zaczęto uczyć na ETH już w 1972 roku. Dziś jest on uważany za najważniejszy współczesny język programowania.

Odniósłszy wielkie sukcesy z Pascalem, Wirth zainteresował się przetwarzaniem wieloprogramowym. Wynikiem jego prac był język **Modula-2**. Pierwszy kompilator tego języka powstał w 1979 roku do pracy na PDP-11.

W 1980 roku Wirth opracował **Lilith** – kombinację Pascala i Moduli-2, a później Oberona, którego struktura była rozszerzeniem podstawowej struktury Moduli-2.

Dziś profesor Niklaus Wirth jest uważany za jednego z największych uczonych świata w dziedzinie informatyki, a jego dzieło – Pascal – za najważniejszy język programowania. Jego marzeniem jest stworzenie bardziej strukturalnego, bardziej przyjaznego i mocniejszego języka niż ten ostatni. Miejmy nadzieję, że Mu się to uda!

Dane o oryginale

NIKLAUS WIRTH
Eidgenossische Technische Hochschule
Zurich, Switzerland

Algorithms + Data Structures = Programs

Authorized translation from the English language edition published by Prentice Hall, Inc.
Copyright © 1976

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Polish language edition published by Wydawnictwa Naukowo-Techniczne
Copyright © 2002

Prowadzenie serii *Elżbieta Beuermann*

Redaktor pierwszego wydania *Jowita Koncewicz-Krzemień*
Redaktor szóstego wydania *Ewa Zdanowicz*

Okładkę i strony tytułowe projektował *Paweł G. Rubaszewski*
Redaktor techniczny *Grażyna Miazek*
Korekta *Zespół*
Skład i łamanie *ANGO*

© Copyright for the Polish edition by Wydawnictwa Naukowo-Techniczne
Warszawa 1980, 1999, 2000, 2001, 2002

All Rights Reserved
Printed in Poland

Utwór w całości ani we fragmentach nie może być powielany ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych, kopiujących, nagrywających i innych, w tym również nie może być umieszczany ani rozpowszechniany w postaci cyfrowej zarówno w Internecie, jak i w sieciach lokalnych bez pisemnej zgody posiadacza praw autorskich.

Adres poczty elektronicznej: wnt@pol.pl
Strona WWW: www.wnt.com.pl

ISBN 83-204-2740-1

Spis treści

Przedmowa 11

1

Podstawowe struktury danych 17

1.1. Wprowadzenie 17
1.2. Pojęcie typu danych 20
1.3. Proste typy danych 23
1.4. Standardowe typy proste 24
1.5. Typy okrojone 27
1.6. Tablice 28
1.7. Rekordy 32
1.8. Rekordy z wariantami 37
1.9. Zbiory 39
1.10. Reprezentacja tablic, rekordów i zbiorów 45
1.10.1. Reprezentacja tablic 46
1.10.2. Reprezentacja rekordów 48
1.10.3. Reprezentacja zbiorów 49
1.11. Plik sekwencyjny 50
1.11.1 Elementarne operatory plikowe 53
1.11.2. Pliki z podstrukturami 56
1.11.3. Teksty 58
1.11.4. Program redagowania pliku 66

2

Sortowanie 73

2.1. Wprowadzenie 73
2.2. Sortowanie tablic 76
2.2.1. Sortowanie przez proste wstawianie 77
2.2.2. Sortowanie przez proste wybieranie 80
2.2.3. Sortowanie przez prostą zamianę 82
2.2.4. Sortowanie za pomocą malejących przyrostów 86
2.2.5. Sortowanie drzewiaste 88

2.2.6. Sortowanie przez podział 94
2.2.7. Znajdowanie mediany 101
2.2.8. Porównanie metod sortowania tablic 103
2.3. Sortowanie plików sekwencyjnych 105
2.3.1. Łączenie proste 105
2.3.2. Łączenie naturalne 111
2.3.3. Wielokierunkowe łączenie wyważone 118
2.3.4. Sortowanie polifazowe 124
2.3.5. Rozdzielanie serii początkowych 136

3

Algorytmy rekurencyjne 145

3.1. Wprowadzenie 145
3.2. Kiedy nie stosować rekursji 148
3.3. Dwa przykłady programów rekurencyjnych 151
3.4. Algorytmy z powrotami 158
3.5. Problem ośmiu hetmanów 163
3.6. Problem trwałego małżeństwa 168
3.7. Problem optymalnego wyboru 175

4

Dynamiczne struktury informacyjne 183

4.1. Rekurencyjne typy danych 183
4.2. Wskaźniki lub odniesienia 186
4.3. Listy liniowe 192
4.3.1. Operacje podstawowe 192
4.3.2. Listy uporządkowane i listy reorganizowane 195
4.3.3. Pewne zastosowanie: sortowanie topologiczne 203
4.4. Struktury drzewiaste 211
4.4.1. Pojęcia podstawowe i definicje 211
4.4.2. Podstawowe operacje na drzewach binarnych 220
4.4.3. Przeszukiwanie drzewa z wstawianiem 223
4.4.4. Usuwanie z drzewa 232
4.4.5. Analiza przeszukiwania drzewa z wstawianiem 234
4.4.6. Drzewa zrównoważone 237
4.4.7. Wstawianie w drzewach zrównoważonych 239
4.4.8. Usuwanie węzłów z drzew zrównoważonych 244
4.4.9. Optymalne drzewa poszukiwań 248
4.4.10. Drukowanie struktury drzewa 254
4.5. Drzewa wielokierunkowe 263
4.5.1. B-drzewa 265
4.5.2. Binarne B-drzewa 277
4.6. Transformacje kluczy (kodowanie mieszające) 284
4.6.1. Wybór funkcji transformującej 286
4.6.2. Usuwanie kolizji 286
4.6.3. Analiza transformacji kluczy 292

5

Struktury językowe i kompilatory . . . 301

5.1. Definicja i struktura języka . . . 301
5.2. Analiza zdań . . . 304
5.3. Konstrukcja diagramu składni . . . 309
5.4. Konstrukcja analizatora składniowego dla zadanej gramatyki . . . 313
5.5. Konstrukcja programu analizatora sterowanego składnią . . . 317
5.6. Translator z notacji BNF na struktury danych sterujące analizatorem . . . 320
5.7. Język programowania PL/0 . . . 328
5.8. Analizator składniowy dla języka PL/0 . . . 332
5.9. Reagowanie na błędy składniowe . . . 341
5.10. Maszyna PL/0 . . . 352
5.11. Generowanie kodu wynikowego . . . 355

Dodatek A

Zbiór znaków ASCII . . . 372

Dodatek B

Diagramy składni dla Pascala . . . 373

Skorowidz przedmiotowy . . . 379

Skorowidz programów . . . 383

Przedmowa

NANI

W dzisiejszych czasach programowanie stało się dziedziną wiedzy, której opanowanie ma zasadnicze znaczenie przy rozwiązywaniu wielu problemów inżynierskich, a którą przy tym można badać i prezentować w sposób naukowy. Programowanie awansowało – przestało być rzemiosłem, a stało się dyscypliną akademicką. Pierwsze prace o niezwykłej doniosłości , które zapoczątkowały ten rozwój, były udziałem E.W. Dijkstry i C.A.R. Hoare'a. Praca Dijkstry „Notes on structured programming"* spowodowała „rewolucję" w programowaniu, ukazując je w nowym świetle – jako przedmiot nauki i współzawodnictwa intelektualnego. Hoare w artykule „Axiomatic basis of computer programming"** pokazuje w sposób klarowny, że programy można analizować, stosując ścisłe rozumowanie matematyczne. W obu tych pracach spotykamy przekonywającą argumentację stwierdzenia, że programiści mogliby uniknąć wielu błędów, gdyby zdawali sobie sprawę z istoty metod i technik, które dotychczas stosowali intuicyjnie i nierzadko nieświadomie. Prace te koncentrowały się wokół aspektów budowy i analizy programów lub, dokładniej, wokół problemu struktury algorytmów reprezentowanych przez teksty programów. Jest oczywiste, że systematyczne i naukowe podejście do konstrukcji programów ma szczególne znaczenie w przypadku dużych, złożonych programów, w których korzysta się ze skomplikowanych zbiorów danych. A zatem metodyka programowania wiąże się również z koniecznością uwzględnienia wszystkich aspektów struktury danych. *Programy* stanowią w końcu skonkretyzowane sformułowania abstrakcyjnych *algorytmów* na podstawie określonej reprezentacji i struktury danych. Niezwykle ważną pracą, wprowadzającą porządek do kłopotliwie różnorodnej terminologii i pojęć dotyczących struktur *danych*, była publikacja Hoare'a „Notes on data structuring"***. Wynika z niej, że jakiekolwiek decyzje dotyczące struktury danych

---

* W *Structured programming* Dahla, Dijkstry i Hoare'a, New York, Academic Press 1972, s. 1–82.

** W *Comm. ACM*. 1969. **12**. No. 10, s. 576–583.

*** W *Structured programming*. s. 83–174.

mogą być podjęte jedynie na podstawie znajomości algorytmów zastosowanych do danych i, vice versa, struktura i wybór algorytmów zależą często ściśle od struktury danych. Krótko mówiąc, problemy tworzenia programów i struktur danych są ze sobą ściśle powiązane.

Z dwóch przyczyn książkę tę rozpoczynam rozdziałem dotyczącym struktur danych. Po pierwsze, intuicyjnie wyczuwa się, że dane poprzedzają algorytm: trzeba dysponować pewnymi obiektami, zanim zacznie się wykonywać na nich operacje. Po drugie (i to jest bardziej bezpośrednia przyczyna), przyjmuję założenie, że czytelnik zna podstawowe pojęcia z dziedziny programowania. Wstępne kursy programowania tradycyjnie koncentrują się wokół algorytmów działających na stosunkowo prostych strukturach danych; dlatego też wydaje się właściwe zamieszczenie wprowadzającego rozdziału o strukturach danych.

W książce tej, zwłaszcza zaś w rozdz. 1, będę się opierać na teorii i terminologii rozwiniętej przez Hoare'a* i zrealizowanej w języku programowania Pascal**. Głównym założeniem tej teorii jest, że dane reprezentują przede wszystkim pewne abstrakcje obiektów rzeczywistych, zdefiniowane jako struktury abstrakcyjne niekoniecznie realizowane w językach programowania. Podczas tworzenia programu reprezentacja danych jest stopniowo precyzowana – jednocześnie z udoskonalaniem algorytmu – w coraz większym stopniu dostosowywana do ograniczeń narzuconych przez dany system programowania***. W związku z tym zaproponuję pewne zasady konstruowania struktur danych, zwanych strukturami podstawowymi. Niezwykle ważne jest, że te struktury są bez trudu realizowalne w pamięciach istniejących maszyn cyfrowych, ponieważ tylko wtedy można je uważać za elementy rzeczywistej reprezentacji danych, jako molekuły utworzone na końcowym etapie precyzowania opisu danych. Do struktur tych należy *rekord*, *tablica* (o stałych wymiarach) oraz *zbiór*. Nie przypadkiem te podstawowe struktury odpowiadają fundamentalnym pojęciom matematycznym.

Kamieniem węgielnym tej teorii jest rozróżnienie między strukturami podstawowymi i złożonymi. Struktury podstawowe są molekułami (zbudowanymi z atomów) stanowiącymi składowe struktur złożonych. Zmienne o strukturze podstawowej zmieniają jedynie swoją wartość, nigdy natomiast nie zmienia się ich struktura ani zbiór wartości, które mogą przyjmować. W konsekwencji nie zmienia się wielkość zajmowanego przez nie obszaru pamięci. Charakterystyczną cechą struktur „złożonych" jest możliwość zmiany zarówno wartości, jak i struktury podczas wykonania programu. Potrzebne są więc bardziej wyrafinowane sposoby ich realizacji w maszynie cyfrowej.

Strukturą pośrednią w sensie tej klasyfikacji jest plik sekwencyjny lub mówiąc krótko – ciąg. W sposób oczywisty zmienia on swoją długość; jednak ta

* W Notes on data structuring.

** N. Wirth: The programming language Pascal. *Acta Informatica*, 1971, **1**, No. 1, s. 35–63.

*** N. Wirth: Program development by stepwiserefinement. *Comm. ACM*, **14**, No. 4, 1971, s. 221–227.

zmiana struktury ma charakter trywialny. Ponieważ plik sekwencyjny odgrywa podstawową rolę w nieomal wszystkich systemach komputerowych, w rozdz. 1 zaliczam go w poczet struktur podstawowych.

Drugi rozdział dotyczy *algorytmów sortowania*. Zawiera on opisy wielu różnorodnych metod służących temu samemu celowi. Zalety i wady tych metod są przedstawione na podstawie analizy matematycznej niektórych algorytmów. Analiza taka jest konieczna do wyboru optymalnego algorytmu rozwiązującego zadany problem. Podział na metody sortowania tablic i metody sortowania plików (zwane często sortowaniem wewnętrznym i zewnętrznym) świadczy o decydującym wpływie reprezentacji danych na wybór stosowanych algorytmów oraz ich złożoność. Algorytmom sortowania nie poświęciłbym tak dużo miejsca w książce, gdyby nie fakt, że sortowanie stanowi doskonały przykład pozwalający zobrazować wiele charakterystycznych zasad programowania i sytuacji występujących w większości innych zastosowań. Sortowanie idealnie nadaje się do tego celu. Wydaje się wręcz, że podstawą całego kursu programowania mogłyby być wyłącznie przykłady programów z zakresu sortowania.

Inny temat, pomijany zwykle we wstępnych kursach programowania, ale odgrywający ważną rolę w koncepcji rozwiązań wielu algorytmów, stanowi pojęcie rekursji. W związku z tym trzeci rozdział jest poświęcony algorytmom rekurencyjnym. Rekursję przedstawiam jako uogólnienie iteracji, będące ważnym i mocnym narzędziem programowania. W wielu ćwiczeniach z programowania daje się, niestety, takie przykłady na zastosowanie rekursji, w których wystarczałoby skorzystać z iteracji. Natomiast w rozdz. 3 koncentruję się na przykładach problemów, w których rekursja pozwala na najbardziej naturalne sformułowanie rozwiązania, podczas gdy użycie iteracji prowadziłoby do nieporęcznych i nieprzejrzystych rozwiązań. Idealne zastosowanie dla rekursji stanowi klasa algorytmów z powrotami, ale najbardziej oczywistymi przykładami są algorytmy działające na danych o rekurencyjnie definiowanej strukturze. Tego rodzaju przypadki są przedstawione w ostatnich dwóch rozdziałach, do których rozdział trzeci stanowi podstawowe wprowadzenie.

Rozdział 4 dotyczy *dynamicznych struktur danych*, tzn. danych, których struktura zmienia się podczas wykonania programu. Pokazuję tam, że rekurencyjne struktury danych stanowią ważną podklasę często używanych struktur dynamicznych. Chociaż rekurencyjny sposób definiowania struktur w takich przypadkach jest nie tylko możliwy, ale również naturalny, w praktyce się go na ogół nie stosuje. Zamiast tego ukazuje się programiście mechanizm zastosowany do realizacji maszynowej tych struktur danych, zmuszając go do korzystania z bezpośrednich odwołań bądź zmiennych *wskaźnikowych*. W książce tej przedstawiam powyższą metodę oraz stan wiedzy w dniu dzisiejszym, kiedy piszę te słowa. Rozdział 4 poświęcam programowaniu za pomocą wskaźników zastosowanemu do list, drzew oraz przykładów dotyczących bardziej nawet złożonych danych. Przedstawiam metodę programowania zwaną często (niezbyt właściwie) „przetwarzaniem list". Wiele miejsca przeznaczam na omówienie organizacji struktur drzewiastych, zwłaszcza zaś przeglądanie drzew. Rozdział

kończy prezentacja metody tablic rozproszonych, zwanej także kodowaniem mieszającym, zalecanej zwykle przy przeglądaniu drzew. Pozwala to na porównanie dwóch zupełnie odmiennych metod dla często spotykanych zastosowań.

Ostatni rozdział zawiera zwarte wprowadzenie do definiowania języków formalnych i do problemu analizy syntaktycznej oraz omówienie konstrukcji *kompilatora* dla pewnego prostego języka i prostej maszyny cyfrowej. Są trzy powody włączenia tego rozdziału do niniejszej książki. Po pierwsze, programista powinien mieć przynajmniej ogólną wiedzę o podstawowych zagadnieniach i metodach kompilacji języków programowania. Po drugie, wzrasta liczba zastosowań wymagających prostych języków wejściowych i języków sterujących. Po trzecie, języki formalne definiują pewną strukturę rekurencyjną ciągów symboli: translatory tych języków stanowią więc doskonałe przykłady zastosowania metod rekurencyjnych, niezbędnych do uzyskania przejrzystej struktury programów, obecnie na ogół coraz większych i bardziej złożonych. Wybór jako przykładu języka PL/0 był podyktowany koniecznością dokonania kompromisu między językiem za prostym na to, aby był wartościowym przykładem, a językiem, którego kompilator znacznie przekraczałby wielkość programów możliwych do zamieszczenia w książce, przeznaczonej nie tylko dla twórców translatorów.

Programowanie jest sztuką konstruktywną. W jaki sposób należy uczyć konstruktywnej i twórczej działalności? Jedną z metod jest wyabstrahowanie na podstawie wielu przykładów zespołu elementarnych zasad tworzenia i przekazanie ich w systematyczny sposób. Jednakże programowanie jest dziedziną wielce zróżnicowaną, często wymagającą złożonej działalności intelektualnej. Przypuszczenie, że kiedykolwiek można by jego naukę skondensować w postaci ścisłych recept, wydaje się niesłuszne. W arsenale środków nauczyciela programowania pozostaje zatem staranny wybór i prezentacja typowych przykładów. Oczywiście nie należy oczekiwać, że każda osoba odniesie takie same korzyści ze studiowania przykładów. Wiele zależy od samego uczącego się, od jego pracowitości oraz intuicji, zwłaszcza w przypadku długich programów. Włączenie ich do książki nie jest więc przypadkowe. Długie programy w praktyce zdarzają się dosyć często i są o wiele bardziej przydatne przy prezentacji owego nieuchwytnego, ale ważnego składnika zwanego stylem i przejrzystą, uporządkowaną strukturą. Służą również jako przykłady ćwiczeń ze sztuki *czytania* programów, zbyt często zaniedbywanej na rzecz umiejętności układania programów. Jest to pierwszorzędna motywacja faktu zamieszczenia przykładów dużych programów w sposób całościowy. Czytelnik ogląda stopniową ewolucję programu: ukazują mu się kolejne „obrazy migawkowe" tworzonego programu, świadczące o *stopniowym precyzowaniu* szczegółów. Prezentując ostateczną postać programów, celowo przywiązywałem dużą wagę do szczegółów stanowiących źródło najczęstszych błędów programowania. Przedstawienie jedynie zasady samego algorytmu i jego analizy matematycznej może być wystarczające dla umysłów akademickich, jednakże wydaje się mało przydatne dla inżynierów i praktyków. W związku z tym w książce tej ściśle przestrzegałem zasady zamieszczania programów napisanych

w takim języku, w jakim mogłyby być bezpośrednio wykonane przez maszynę cyfrową.

Powstaje oczywiście problem znalezienia takiej postaci programu, która będzie mogła być wykonywana przez komputer, ale będzie jednocześnie niezależna od niego na tyle, żeby można ją było prezentować w tej książce. Pod tym względem ani szeroko stosowane języki, ani notacje abstrakcyjne nie stanowią adekwatnego rozwiązania. Kompromisowym wyjściem jest użycie języka Pascal, dokładnie pod tym kątem opracowanego i konsekwentnie stosowanego w tej książce. Programy są zrozumiałe dla programistów obeznanych z innymi językami wysokiego poziomu, takimi jak Algol 60 czy PL/I; łatwo jest bowiem zrozumieć notację Pascala na podstawie czytanego tekstu. Pewne przygotowanie byłoby jednak bardzo przydatne. Idealną podstawę stanowi książka *Systematic Programming**, w której również przyjęto notację Pascala. Moją intencją nie było jednakże stworzenie podręcznika języka Pascal; istnieją bardziej odpowiednie prace służące temu celowi**.

Niniejsza książka jest skondensowanym i jednocześnie nieco zmodyfikowanym zbiorem wykładów z programowania prowadzonych w Eidgenossische Technische Hochschule w Zurychu (ETH). Wiele idei i poglądów przedstawionych w tej książce zawdzięczam dyskusjom z moimi kolegami z ETH. W szczególności chciałbym podziękować Panu H. Sandmayrowi za staranne przeczytanie rękopisu i Pani Heidi Theiler za uwagę i cierpliwość przy przepisywaniu tekstu. Chciałbym też wspomnieć o stymulującym wpływie spotkań Grup Roboczych 2.1 i 2.3 Międzynarodowej Federacji Przetwarzania Informacji (IFIP); szczególnie wartościowe wnioski wyciągnąłem z rozmów z E.W. Dijkstrą i C.A.R. Hoare'em. Książka ta nie powstałaby też, gdyby nie było zaplecza komputerowego zapewnionego przez ETH.

N. Wirth

---

* N. Wirth, Englewood Cliffs, N.J., Prentice-Hall 1973. [Polski przekład: *Wstęp do programowania systematycznego*. Wyd. 2, Warszawa, WNT 1987. – Przyp. red. pol.]

** K. Jensen i N. Wirth: *Pascal – User manual and report*. Berlin, Springer 1974.

Generał porucznik L. Euler za naszym pośrednictwem składa poniższą Deklarację. Wyznaje otwarcie:

...

III. Że nawet będąc królem matematyków, będzie się wstydził swego błędu, pozostającego w niezgodzie ze zdrowym rozsądkiem i podstawową wiedzą, a popełnionego przy wnioskowaniu na podstawie wzorów, że ciało pod wpływem sił przyciągania zlokalizowanych w środku sfery zmieni nagle kierunek poruszania się w stronę środka;

IV. Że zrobi wszystko, co możliwe, aby nie być ponownie oszukanym przez zły wzór. Przeprasza gorąco za to, iż pewnego razu, wprowadzony w zakłopotanie paradoksalnym wynikiem, oświadczył: „chociaż wydaje się to być w niezgodzie z rzeczywistością, jednak musimy ufać naszym obliczeniom bardziej niż naszym zmysłom";

V. Że w przyszłości nie będzie więcej zapisywał sześćdziesięciu stron obliczeniami po to, aby osiągnąć wynik, który można uzyskać w dziesięciu wierszach po pewnych ważnych przemyśleniach; i jeśli kiedykolwiek będzie miał zakasać rękawy i spędzić trzy dni i noce z rzędu na obliczeniach, to wcześniej poświęci kwadrans, aby stwierdzić, które reguły obliczeń będą najbardziej przydatne.

Fragment z *Diatribe du docteur Akakia* Voltaire'a
(listopad 1752)

1 Podstawowe struktury danych

1.1. Wprowadzenie

Nowoczesną maszynę cyfrową wynaleziono w celu ułatwienia i przyspieszenia wykonywania skomplikowanych i czasochłonnych obliczeń. W większości jej zastosowań najważniejsze znaczenie ma możliwość przechowywania i dostępu do dużych partii informacji, natomiast zdolność wykonywania obliczeń w wielu przypadkach okazuje się prawie nieistotna.

We wszystkich tych przypadkach duże partie informacji, które mają być przetwarzane, reprezentują – w pewnym sensie – **abstrakcyjny model** części świata rzeczywistego. Informacja dostępna komputerowi stanowi pewien wybrany zbiór danych o świecie rzeczywistym, mianowicie zbiór danych uznanych za istotne do rozwiązania rozważanego problemu, tj. taki, co do którego ma się przekonanie, że można z niego uzyskać żądane wyniki. Dane te reprezentują model abstrakcyjny w tym sensie, że pewne właściwości obiektów rzeczywistych są ignorowane, ponieważ nie są związane z badanym problemem. Ten model abstrakcyjny jest więc również pewnym uproszczeniem zespołu faktów.

Jako przykład rozważmy kartotekę osobową pracowników. Każdy pracownik jest (abstrakcyjnie) reprezentowany w kartotece przez zbiór danych potrzebnych bądź pracodawcy, bądź do prowadzenia jego ksiąg rachunkowych. Zbiór ten może zawierać pewne informacje identyfikujące zatrudnionego, np. nazwisko czy też wysokość uposażenia. Jednakże nie będzie on zwykle zawierał nieistotnych tu informacji, takich jak kolor włosów, waga czy wzrost.

Przy rozwiązywaniu problemu – czy to za pomocą komputera, czy też bez niego – trzeba dokonać wyboru abstrakcyjnego modelu rzeczywistości, czyli zdefiniować zbiór danych mających reprezentować rzeczywistą sytuację. Wybór ten powinien być podporządkowany problemowi, który ma być rozwiązany. Potem następuje wybór reprezentacji tych informacji. Ten wybór jest z kolei podyktowany narzędziem, które służy do rozwiązania problemu, czyli możliwościami komputera. W większości przypadków oba te kroki nie są całkowicie niezależne.

Wybór reprezentacji danych jest często dosyć trudny i zdeterminowany nie tylko możliwościami komputera. Trzeba również brać pod uwagę operacje, które mają być wykonywane na danych. Dobrym przykładem jest reprezentacja liczb stanowiących abstrakcyjny obraz pewnych właściwości charakteryzowanych obiektów. Jeśli jedyną (bądź przynajmniej dominującą) operacją na liczbach ma być dodawanie, to dobrą metodą reprezentacji jest zapisanie liczby n za pomocą n kresek. Przy takiej reprezentacji reguła dodawania jest oczywista i bardzo prosta. Liczby rzymskie są reprezentowane zgodnie z tą samą prostą zasadą i reguły dodawania są dla nich podobnie oczywiste w przypadku małych liczb. Natomiast reprezentacja za pomocą liczb arabskich wymaga reguł dalekich od oczywistości (dla małych liczb), muszą więc być one zapamiętane. Odmienna sytuacja występuje wówczas, gdy mamy do czynienia z dodawaniem dużych liczb lub mnożeniem i dzieleniem. Dzięki zastosowaniu zasady systematycznego układu opartego na przydzielaniu wag pozycjom cyfr rozłożenie tych operacji na operacje elementarne jest dużo prostsze w przypadku reprezentacji za pomocą liczb arabskich.

Wewnętrzna reprezentacja liczb w maszynach cyfrowych jest oparta na cyfrach dwójkowych (bitach). Reprezentacja ta jest niewygodna dla ludzi, zawiera bowiem dużą zazwyczaj liczbę cyfr dwójkowych, ale jest najbardziej odpowiednia dla urządzeń elektronicznych, ponieważ dwie wymagane wartości 0 i 1 mogą być wygodnie i w sposób niezawodny reprezentowane przez występowanie lub niewystępowanie prądów elektrycznych, ładunku elektrycznego bądź pól magnetycznych.

Z powyższego przykładu wynika, że problem reprezentacji trzeba analizować na wielu poziomach szczegółowości. Na przykład przy problemie reprezentacji pozycji jakiegoś obiektu pierwsza decyzja może prowadzić do wyboru pary liczb rzeczywistych stanowiących współrzędne kartezjańskie bądź biegunowe. Druga decyzja może prowadzić do reprezentacji zmiennopozycyjnej, gdzie każda liczba rzeczywista x składa się z pary liczb całkowitych oznaczających ułamek f i wykładnik e przy pewnej podstawie (np. $x = f \cdot 2^e$). Kolejna decyzja – wynikająca ze świadomości, że dane trzeba przechowywać w pamięci maszyny cyfrowej – może prowadzić do dwójkowej, pozycyjnej reprezentacji liczb całkowitych. Ostatnią decyzją może być reprezentacja cyfr dwójkowych przez kierunek strumienia magnetycznego w pamięci magnetycznej. Oczywiście pierwsza z tych decyzji wynika przede wszystkim z samego problemu, następne zaś zależą coraz bardziej od wyboru narzędzia, którym się posługujemy, i jego technologii. Nie można więc żądać od programisty decydowania o reprezentacji liczb czy wręcz o charakterystyce urządzenia przechowującego informacje. Takie „decyzje na niskim szczeblu" zostawia się projektantom komputera, którzy mają najwięcej potrzebnych informacji na temat stosowanych technologii i mogą podjąć najbardziej sensowne decyzje odpowiednie dla wszystkich (lub prawie wszystkich) zastosowań, w których odgrywa rolę pojęcie liczby.

W tym kontekście widać wyraźnie znaczenie **języków programowania**. Język programowania reprezentuje pewną abstrakcyjną maszynę, która rozumie

występujące w nim terminy, stanowiące pewien wyższy poziom abstrakcji w stosunku do obiektów używanych przez rzeczywisty komputer. Programista używający takiego języka „wyższego poziomu" jest więc uwolniony od problemów reprezentacji liczb, jeśli liczba jest elementarnym obiektem w języku (ale również w pewnym sensie skrępowany).

Znaczenie używania języka oferującego odpowiedni zbiór podstawowych pojęć abstrakcyjnych wspólnych dla większości problemów przetwarzania danych polega głównie na niezawodności programów wynikowych. Łatwiej napisać program przy użyciu znanych pojęć liczb, zbiorów, ciągów czy powtórzeń (iteracji) zamiast bitów, „słów" czy skoków. Oczywiście komputer będzie reprezentował wszystkie dane – zarówno liczby, zbiory, jak i ciągi – jako wielką masę bitów. Jednakże fakt ten dopóty jest nieistotny dla programisty, dopóki nie musi się on interesować problemem reprezentacji wybranych przez siebie pojęć i dopóki może być pewnym, że odpowiednie reprezentacje w komputerze (czy w translatorze) są wystarczające do jego celów.

Dokonanie przez inżyniera (czy przez twórcę translatora) odpowiedniego wyboru reprezentacji jest tym łatwiejsze, im bliższe danemu komputerowi są odpowiednie pojęcia abstrakcyjne. Zwiększa się też wówczas prawdopodobieństwo, że pojedynczy wybór będzie odpowiedni dla wszystkich (lub prawie wszystkich) możliwych do pomyślenia zastosowań. Fakt ten określa pewne ograniczenia na stopień oddalenia poziomu abstrakcji używanych pojęć od danego rzeczywistego komputera. Nie miałoby na przykład sensu wprowadzenie obiektów geometrycznych jako podstawowych jednostek danych dla języka o uniwersalnym zastosowaniu, właściwa bowiem reprezentacja owych obiektów, z uwagi na ich złożoność, zależy w dużej mierze od wyboru operacji, które będą wykonywane na tych obiektach. Jednakże twórca uniwersalnego języka i jego translatora nie będzie znał natury tych operacji ani częstości ich występowania i każdy wybór, jakiego dokona, może nie odpowiadać pewnym potencjalnym zastosowaniom.

Powyższe rozważania określają dokonany w tej książce wybór notacji dla opisu algorytmów i danych. Oczywiście chcielibyśmy używać znanych pojęć z dziedziny matematyki, takich jak liczby, zbiory, ciągi itd., zamiast pojęć zależnych od komputera, takich jak np. ciągi bitów. Jednakże równie oczywiste jest, że powinniśmy stosować taką notację, dla której istnieją efektywne translatory. Jednocześnie wiadomo, że niemądre byłoby używanie języka ściśle uzależnionego od komputera. Język taki byłby bowiem nieprzydatny do zapisu programów w pewnej notacji abstrakcyjnej, zostawiającej zazwyczaj otwarte problemy reprezentacji.

Jako kompromis między tymi dwoma przypadkami krańcowymi stworzono język programowania Pascal i jego właśnie używa się w tej książce [1.3 i 1.5]. Istnieje wiele efektywnych translatorów tego języka dla różnych maszyn cyfrowych. Dowiedziono też, że wybrana w Pascalu notacja jest dostatecznie bliska rzeczywistym maszynom, tak że charakterystyczne dla Pascala pojęcia oraz ich reprezentacje można łatwo objaśnić. Ponadto język ten jest dosyć bliski

innym językom (szczególnie Algolowi 60), toteż uzyskane tu doświadczenia mogą być również z powodzeniem wykorzystane przy okazji używania tych języków.

1.2. Pojęcie typu danych

W matematyce zmienne klasyfikuje się zgodnie z ich pewnymi istotnymi właściwościami. Rozróżnia się zmienne rzeczywiste, zespolone i logiczne lub też zmienne reprezentujące pojedyncze wartości, zbiory wartości, zbiory zbiorów, a także funkcje, funkcjonały, zbiory funkcji itd. W przetwarzaniu danych klasyfikacja zmiennych jest pojęciem równie ważnym, jeśli nie ważniejszym. Przyjmiemy jako zasadę, że **każda stała, zmienna, wyrażenie czy też funkcja jest pewnego typu**. Typ ten dokładnie charakteryzuje zbiór wartości, do którego należy stała, bądź jakie może przyjmować zmienna czy wyrażenie, bądź jakie mogą być generowane przez funkcję.

W tekstach matematycznych można zazwyczaj, abstrahując od kontekstu, określić typ zmiennej na podstawie kroju czcionki, jaką jest oznaczany. Nie jest to możliwe w programach dla komputera, ponieważ ma się do dyspozycji jeden, ogólnie dostępny krój czcionek (tzn. litery łacińskie). Szeroko przyjęto zatem stosowanie zasady, że typ podaje się explicite w **deklaracji** stałej, zmiennej czy funkcji oraz że deklaracja ta poprzedza w tekście programu użycie owej stałej, zmiennej czy też funkcji. Zasada ta staje się szczególnie istotna, jeśli się zwróci uwagę na fakt, że kompilator musi dokonać wyboru reprezentacji obiektów wewnątrz pamięci maszyny. Oczywiście wielkość obszaru pamięci przydzielonego zmiennej należy wybierać odpowiednio do zakresu wartości, które ta zmienna może przyjmować. Jeśli informacje te są znane kompilatorowi, można uniknąć tzw. dynamicznej alokacji pamięci. Jest to często klucz do efektywnej realizacji algorytmu.

Poniżej podano podstawowe cechy charakterystyczne pojęcia typu używane w niniejszej książce i zrealizowane w języku Pascal [1.2].

(1) Typ danych określa zbiór wartości, do którego należy stała bądź które może przyjmować zmienna lub wyrażenie albo które mogą być generowane przez operator lub funkcję.

(2) Typ wartości oznaczonej przez stałą, zmienną lub wyrażenie można określić na podstawie ich postaci bądź deklaracji, bez konieczności wykonywania procesu obliczeniowego.

(3) Każdy operator lub funkcja ma argumenty ustalonego typu, jak również daje wynik ustalonego typu. Jeśli operator dopuszcza argumenty wielu typów (np. znak „+" oznacza dodawanie zarówno liczb całkowitych, jak i rzeczywistych), to typ wyniku określony jest pewnymi specyficznymi dla języka regułami.

W konsekwencji kompilator może korzystać z informacji o typach w celu sprawdzenia poprawności wielu różnych konstrukcji. Na przykład przypisanie wartości logicznej (Boolean) zmiennej arytmetycznej (real) może zostać wykryte bez potrzeby wykonywania programu. Ten rodzaj redundancji w tekście programu jest niezwykle pomocny przy tworzeniu programów i należy go uważać za podstawową zaletę dobrych języków programowania wysokiego poziomu w stosunku do kodu maszynowego (czy też kodu w języku adresów symbolicznych). Oczywiście dane będą ostatecznie reprezentowane przez długie ciągi cyfr dwójkowych bez względu na to, czy program został napisany w języku wysokiego poziomu, w którym występuje pojęcie typu, czy też w kodzie, w którym to pojęcie nie istnieje. Dla komputera pamięć stanowi jednolitą masę bitów bez jakiejkolwiek struktury. Jednakże to właśnie ta struktura abstrakcyjna umożliwia programistom rozpoznawać znaczenie w monotonnym krajobrazie pamięci maszyny.

W teorii przedstawionej w tej książce oraz w języku programowania Pascal istnieją określone metody definiowania typów danych. W większości przypadków nowe typy danych definiuje się za pomocą typów danych zdefiniowanych wcześniej. Wartości takiego typu są zwykle konglomeratami **wartości składowych** (ang. *component values*) o pierwotnie zdefiniowanych **typach składowych** (ang. *constituent types*) i mówi się, że są one ustrukturowane. Jeśli występuje tylko jeden typ składowy, tzn. wszystkie składowe są tego samego typu, to typ ów nazywa się **typem podstawowym** (ang. *base type*).

Liczbę różnych wartości należących do typu T nazywa się **mocą** (ang. *cardinality*) T. Moc typu pozwala określić wielkość pamięci potrzebnej do reprezentowania zmiennej x typu T (oznaczonej $x : T$).

Ponieważ typy składowe również mogą być ustrukturowane itd., tworzy się więc w rezultacie całe hierarchie struktur; oczywiście jednak ostatnie składowe muszą być atomowe. Potrzebna jest więc pewna notacja służąca do określania takich typów prostych nieustrukturowanych. Narzuca się tu metoda polegająca na **wyliczeniu** wartości tworzących typ. Na przykład w programie dotyczącym figur geometrycznych na płaszczyźnie można określić typ prosty o nazwie *kształt*, którego wartości mogą być oznaczone identyfikatorami *prostokąt*, *kwadrat*, *elipsa*, *okrąg*. Oprócz takich typów definiowanych przez programistę muszą istnieć pewne **typy standardowe** (ang. *standard types*), które są zdefiniowane wcześniej. Na ogół typy standardowe zawierają **liczby** i **wartości logiczne**. Jeśli wartości typu są uporządkowane, to typ będziemy nazywali **uporządkowanym** (ang. *ordered*) lub **skalarnym** (ang. *scalar*). W języku Pascal wszystkie typy nieustrukturowane są uważane za uporządkowane; w przypadku jawnego wyliczenia wartości przyjmuje się, że wartości te są uporządkowane zgodnie z kolejnością ich wyliczenia.

Przyjmując powyższe zasady, możemy definiować typy proste, następnie zaś budować konglomeraty – typy złożone, o dowolnym stopniu zagnieżdżenia. W rzeczywistości nie wystarcza dysponować tylko jedną, ogólną metodą tworzenia typów złożonych. W związku z różnymi praktycznymi problemami

reprezentacji i zastosowań język programowania ogólnego przeznaczenia powinien oferować wiele **metod strukturalizacji** typów danych. W sensie matematycznym wszystkie te metody mogą być równoważne; różnice polegają na wyborze operatorów stosowanych do konstruowania wartości i wybierania składowych tych wartości. Podstawowymi strukturami typów złożonych, przedstawionymi w tej książce są: **tablica**, **rekord**, **zbiór** i **ciąg (plik)**. Bardziej skomplikowane struktury nie są zwykle definiowane jako typy „statyczne", lecz są „dynamicznie" generowane podczas wykonywania programu i mogą w tymże procesie zmieniać swój kształt i rozmiary. Struktury takie, omawiane w rozdz. 4, zawierają listy, pierścienie, drzewa i ogólne grafy skończone.

Zmienne i typy wprowadza się do programu w celu użycia ich w obliczeniach. W związku z tym musi istnieć pewien zbiór **operatorów**. Dla standardowych typów danych w językach programowania proponuje się pewną liczbę prostych, standardowych (atomowych) operacji wraz z pewną liczbą metod ich składania (tworzenia operacji złożonych) za pomocą proponowanych operatorów prostych. Problem składania operacji uważa się często za esencję sztuki programowania. Jednakże okaże się, że właściwe, trafne składanie danych jest problemem równie podstawowym, jak i ważnym.

Najważniejszymi operacjami prostymi są **porównanie** i **przypisanie**, tzn. test na równość (i uporządkowanie, w przypadku typów skalarnych) oraz instrukcja narzucająca równość. Fundamentalna różnica między tymi dwiema operacjami wyraża się w tej pracy zastosowaniem dwóch różnych oznaczeń (chociaż, niestety, w szeroko używanych językach programowania, takich jak Fortran czy PL/I, stosuje się to samo oznaczenie).

Test na równość: $x = y$
Przypisanie: $x := y$

Te podstawowe operacje definiuje się dla większości typów danych, ale należy zaznaczyć, że ich wykonywanie może się wiązać ze znaczną stratą czasu wykonania, jeśli dane są duże i mają złożoną strukturę.

Oprócz testu na równość (czy na uporządkowanie) i przypisania wprowadza się klasę fundamentalnych operacji zwanych **operacjami konwersji typów**. Są one odwzorowaniami typów danych na inne typy (są szczególnie istotne w przypadku typów złożonych). Wartości złożone generuje się z ich wartości składowych za pomocą tzw. **konstruktorów**, a wartości składowe wybiera się za pomocą tzw. **selektorów**. Konstruktory i selektory stanowią więc operacje konwersji typów będące funkcjami z typów składowych w typy złożone i vice versa. Każda metoda strukturalizacji wiąże się z konkretną parą złożoną z konstruktora i selektora, o wyraźnie odrębnych oznaczeniach.

Standardowe proste typy danych wymagają również określonego zestawu standardowych prostych operatorów. Dlatego też równolegle z wprowadzeniem standardowych typów dla wartości liczbowych i wartości logicznych wprowadzimy powszechnie przyjęte operacje arytmetyczne i logiczne.

1.3. Proste typy danych

W wielu programach stosuje się liczby całkowite wtedy, gdy własności numeryczne nie są potrzebne, a liczba całkowita reprezentuje tylko wybór spośród pewnej małej liczby możliwości. W tych przypadkach wprowadzamy nowy, prosty typ danych *T*, wyliczając zbiór wszystkich możliwych wartości $c_1, c_2, ..., c_n$:

---

type $T = (c_1, c_2, ..., c_n)$ (1.1)

Mocą typu *T* jest wartość funkcji $moc(T) = n$.

PRZYKŁADY

type *kształt* = (*prostokąt, kwadrat, elipsa, okrąg*)
type *kolor* = (*czerwony, żółty, zielony*)
type *płeć* = (*mężczyzna, kobieta*)
type *Boolean* = (*false, true*)
type *dzieńtygodnia* = (*poniedziałek, wtorek, środa, czwartek, piątek, sobota, niedziela*)
type *waluta* = (*frank, marka, funt, dolar, szyling, lir, gulden, korona, rubel, cruzeiro, jen*)
type *przeznaczenie* = (*piekło, czyściec, niebo*)
type *środeklokomocji* = (*pociąg, autobus, samochód, statek, samolot*)
type *ranga* = (*szeregowiec, kapral, sierżant, porucznik, kapitan, major, pułkownik, generał*)
type *obiekt* = (*stała, typ, zmienna, procedura, funkcja*)
type *struktura* = (*plik, tablica, rekord, zbiór*)
type *warunek* = (*ręczna, niezaładowana, parzystość, skos*)

Definicja takiego typu wprowadza nie tylko nowy identyfikator typu, ale jednocześnie zbiór identyfikatorów oznaczających poszczególne wartości definiowanego typu. Identyfikatorów tych można używać w programie jako stałych, a ich wystąpienia czynią program znacznie bardziej zrozumiałym. Wprowadźmy na przykład zmienne *p*, *d*, *r* oraz *b*:

var *p*: *płeć*
var *d*: *dzieńtygodnia*
var *r*: *ranga*
var *b*: *Boolean*

Możliwe są wtedy następujące instrukcje przypisania:

p := *mężczyzna*
d := *niedziela*

$r := major$
$b := true$

Instrukcje te są oczywiście bardziej obrazowe aniżeli ich numeryczne odpowiedniki

$p := 1 \qquad d := 7 \qquad r := 6 \qquad b := 2$

oparte na założeniu, że p, d, r oraz b są typu *integer*, stałe zaś są odwzorowane na liczby naturalne według kolejności ich wyliczenia. Ponadto translator może wykrywać nierozważne stosowanie operatorów arytmetycznych do typów nienumerycznych, jak np.

$p := p + 1$

Jeśli typ traktuje się jako uporządkowany, to sensowne jest jednak wprowadzenie funkcji generujących następnik bądź poprzednik argumentu. Funkcje te oznacza się odpowiednio przez $succ(x)$ i $pred(x)$. Uporządkowanie wartości typu T określa następująca reguła:

$$(c_i < c_j) \equiv (i < j) \tag{1.2}$$

1.4. Standardowe typy proste

Standardowe typy proste są to takie typy, które w większości maszyn cyfrowych występują jako „możliwości wbudowane". Należą do nich: zbiór liczb całkowitych, zbiór wartości logicznych i zbiór znaków drukarki. W większych komputerach można również korzystać z liczb ułamkowych oraz z odpowiednich prostych operatorów. Do oznaczania tych typów będziemy odpowiednio używali identyfikatorów

integer, *Boolean*, *char*, *real*

Typ *integer* stanowi podzbiór wszystkich liczb całkowitych, którego zakres jest uzależniony od indywidualnych właściwości danej maszyny. Jednakże zakłada się, że wszystkie operacje wykonywane na danych tego typu są dokładne i odpowiadają podstawowym prawom arytmetyki oraz że wykonanie programu zostanie przerwane w przypadku, gdy wynik obliczenia znajdzie się poza reprezentowanym podzbiorem. Standardowe operatory dotyczą czterech podstawowych operacji arytmetycznych: dodawania (+), odejmowania (−), mnożenia (*) i dzielenia (**div**). Ostatnia operacja rozumiana jest jako dzielenie całkowite, w którym ignoruje się ewentualną resztę, tzn. dla dodatnich m i n zachodzi

$$m - n < (m \,\mathbf{div}\, n) * n \leqslant m \tag{1.3}$$

Korzystając z operacji dzielenia, wprowadza się operator „modulo" spełniający równość

$$(m \ \mathbf{div}\ n) * n + (m \ \mathbf{mod}\ n) = m \tag{1.4}$$

Wobec powyższego m **div** n stanowi część całkowitą ilorazu m i n, natomiast m **mod** n jest resztą tego ilorazu.

Typ *real* oznacza pewien podzbiór liczb rzeczywistych. Podczas gdy zakłada się, że arytmetyka wartości typu *integer* daje wyniki dokładne, dla arytmetyki wartości typu *real* dopuszcza się wyniki niedokładne w ramach pewnych błędów zaokrąglenia spowodowanych wykonywaniem obliczeń na skończonej liczbie cyfr. Jest to zasadniczy powód tego, że w większości języków programowania wyraźnie rozróżnia się typy *integer* i *real*.

Do oznaczania dzielenia liczb rzeczywistych, dającego w wyniku liczbę rzeczywistą jako wartość ilorazu, będziemy stosowali znak /, w przeciwieństwie do znaku **div**, oznaczającego dzielenie całkowite.

Dwie wartości standardowego typu *Boolean* są oznaczane identyfikatorami *true* oraz *false*. Do operatorów boolowskich zaliczamy koniunkcję, alternatywę i negację. Wartości wyników tych operacji w zależności od argumentów podano w tabl. 1.1. Koniunkcję oznacza się symbolem $\wedge$ (lub **and**), alternatywę – symbolem $\vee$ (lub **or**), negację zaś – symbolem $\neg$ (lub **not**). Zauważmy, że

TABLICA 1.1
Operatory boolowskie

p	q	$p \vee q$	$p \wedge q$	$\neg p$
true	*true*	*true*	*true*	*false*
true	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*
false	*false*	*false*	*false*	*true*

zastosowanie operatora porównania daje wynik typu *Boolean*. W związku z tym wynik porównania może być przypisany zmiennej bądź użyty jako argument operatora boolowskiego w wyrażeniu typu *Boolean*. Na przykład dla zmiennych boolowskich p i q oraz zmiennych całkowitych $x = 5$, $y = 8$, $z = 10$ dwie instrukcje przypisania:

$$p := x = y$$
$$q := (x < y) \wedge (y \leqslant z)$$

dają w wyniku $p = false$ oraz $q = true$.

Typ standardowy *char* obejmuje zbiór znaków drukarki. Niestety, nie istnieje ogólnie przyjęty standardowy zbiór znaków stosowany we wszystkich systemach komputerowych. Użycie określenia „typ standardowy" może zatem prowadzić do nieporozumień; należy je więc rozumieć jako „typ standardowy dla systemu komputerowego, w którym dany program ma być wykonywany".

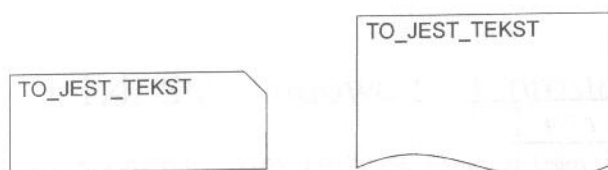
Zbiór znaków zdefiniowany przez ISO (International Standards Organization), zwłaszcza zaś jego amerykańska wersja ASCII (American Standard Code for Information Interchange) jest chyba zbiorem najszerzej uznanym i stosowanym. Zbiór znaków ASCII podano w tabl. A.1 zamieszczonej w dodatku A. Zawiera on 95 znaków drukowanych (**graficznych**) i 33 znaki sterujące, używane zazwyczaj przy przesyłaniu danych i do sterowania urządzeniami drukującymi. Podzbiór zawierający 64 znaki drukarki (bez małych liter) jest szeroko stosowany pod nazwą **ograniczonego** zbioru znaków ASCII.

Aby konstruować niezależne od komputera algorytmy, w których korzysta się ze znaków (wartości typu *char*), musimy przyjąć pewne właściwości zbioru znaków jako wiążące, a mianowicie:

(1) Typ *char* zawiera 26 *liter* łacińskich, 10 *cyfr* arabskich i pewną liczbę innych znaków graficznych, jak np. znaki przestankowe.
(2) Podzbiory liter i cyfr są *uporządkowane* i *spójne*, tzn.

$$\begin{aligned} &('\mathrm{A}' \leqslant x) \wedge (x \leqslant '\mathrm{Z}') \equiv x \text{ jest literą} \\ &('0' \leqslant x) \wedge (x \leqslant '9') \equiv x \text{ jest cyfrą} \end{aligned} \tag{1.5}$$

(3) Typ *char* zawiera znak pusty, którego można używać jako separatora (spacji).
Znak ten jest oznaczany symbolem _ na rys. 1.1.



RYSUNEK 1.1
Reprezentacja tekstu

Do pisania programów w postaci niezależnej od maszyny szczególnie ważna jest możliwość stosowania dwóch funkcji konwersji między typami *char* i *integer*. Będziemy je zapisywali *ord*(*c*), co oznacza liczbę porządkową znaku *c* w zbiorze *char* oraz *chr*(*i*), co oznacza *i*-ty znak ze zbioru *char*. Jak widać, *chr* jest funkcją odwrotną do *ord* i vice versa, tzn.

$$\begin{aligned} &ord(chr(i)) = \mathrm{i} \quad (\text{jeśli } chr(i) \text{ jest zdefiniowana}) \\ &chr(ord(c)) = c \end{aligned} \tag{1.6}$$

Szczególnie warte uwagi są funkcje

$$\begin{aligned} &f(c) = ord(c) - ord('0') = \text{pozycja } c \text{ wśród cyfr} \\ &g(i) = chr(i) + ord('0') = i\text{-ta cyfra} \end{aligned} \tag{1.7}$$

Na przykład $f('3') = 3$, $g(5) = '5'$. Funkcje *f* i *g* stanowią funkcje wzajemnie odwrotne, tzn.

$$\begin{aligned} &f(g(i)) = i \quad (0 \leqslant i \leqslant 9) \\ &g(f(c)) = c \quad ('0' \leqslant c \leqslant '9') \end{aligned} \tag{1.8}$$

Funkcji konwersji używa się w celu dokonania przekształcenia wewnętrznych reprezentacji liczb na ciągi cyfr i odwrotnie. W rzeczywistości reprezentują one owe przekształcenia na najbardziej elementarnym poziomie, mianowicie na poziomie pojedynczej cyfry.

1.5. Typy okrojone

Często się zdarza, że zmienna przyjmuje wartości jedynie z pewnego przedziału wartości określonego typu. Wyraża się to, definiując typ tej zmiennej jako pewien **typ okrojony**, zgodnie z formatem

---

type *T* = *min*. .*max* (1.9)

---

gdzie *min* oraz *max* stanowią granice przedziału.

PRZYKŁADY
type *rok* = 1900. .1999
type *litera* = 'A'. .'Z'
type *cyfra* = '0'. .'9'
type *oficer* = *porucznik*. .*generał*
□

Dla danych zmiennych

var *y*: *rok*
var *L*: *litera*

dozwolone są przypisania *y* := 1973 i *L* := 'W', niedozwolone natomiast *y* := 1291 i *L* := '9'. Poprawności tego rodzaju przypisań nie można sprawdzić podczas translacji, chyba że wartość przypisywana jest oznaczona stałą lub zmienną tego samego typu. Jednakże dopuszczalność przypisań w rodzaju

y := *i*

oraz

L := *c*

gdzie *i* jest typu *integer*, *c* zaś typu *char*, można sprawdzić jedynie podczas wykonania programu. W praktyce systemy dokonujące tego rodzaju sprawdzeń okazały się niezwykle użyteczne przy tworzeniu programów. Korzystanie w nich z nadmiarowości informacji do wykrywania ewentualnych błędów stanowi pierwszorzędną motywację stosowania języków wysokiego poziomu.

1.6. Tablice

Tablica (ang. *array*) jest prawdopodobnie najbardziej znaną strukturą danych, ponieważ w wielu językach, zwłaszcza w Fortranie i Algolu 60, jest jedyną bezpośrednio dostępną strukturą. Tablica jest strukturą jednorodną; jest złożona ze składowych tego samego typu zwanego **typem podstawowym**. Tablicę zwie się również strukturą o **dostępie swobodnym** (ang. *random access*); wszystkie składowe mogą być wybrane w dowolnej kolejności i są jednakowo dostępne. W celu wybrania pojedynczej składowej nazwę całej struktury uzupełnia się tzw. **indeksem** wybierającym składową. Indeks ten powinien być wartością pewnego typu zwanego **typem indeksującym** (ang. *index type*) tablicy. Definicja typu tablicowego T zawiera więc zarówno specyfikację typu podstawowego T_0, jak i typu indeksującego I.

---

type T = **array**[I] **of** T_0 (1.10)

PRZYKŁADY

type *Wiersz* = **array**[1..5] **of** *real*
type *Karta* = **array**[1..80] **of** *char*
type *alfa* = **array**[1..10] **of** *char*
□

Konkretną wartość zmiennej

var *x*: *Wiersz*

której każda składowa spełnia równanie $x_i = 2^{-i}$, można zobrazować tak, jak pokazano na rys. 1.2.

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

RYSUNEK 1.2
Tablica typu *Wiersz*

Złożoną wartość x typu T o wartościach składowych $c_1, ..., c_n$ można oznaczyć za pomocą **konstruktora tablicowego** i instrukcji przypisania:

$$x := T(c_1, ..., c_n) \tag{1.11}$$

Operatorem odwrotnym do konstruktora jest **selektor**. Służy on do wybrania konkretnej składowej tablicy. Dla danej zmiennej tablicowej x selektor jest

oznaczony nazwą tablicy, po której występuje indeks i wybierający odpowiednią składową:

$$x[i] \tag{1.12}$$

Działając na tablicach (zwłaszcza zaś na dużych), stosuje się technikę selektywnego aktualizowania poszczególnych składowych zamiast konstruowania całkowicie nowych wartości złożonych. Wyraża się to traktowaniem zmiennej tablicowej jako tablicy zmiennych składowych i umożliwieniem wykonywania przypisań wybranym składowym.

PRZYKŁAD
$x[i] := 0.125$
□

Aczkolwiek aktualizacja selektywna powoduje zmianę jedynie pojedynczej składowej wartości, jednak z koncepcyjnego punktu widzenia musimy przyjąć, że zmienia się również całą wartość złożoną.

Niezwykle ważne konsekwencje pociąga za sobą fakt, że indeksy tablicy, tzn. „nazwy" składowych muszą być elementami zdefiniowanego typu skalarnego. Indeksy mogą być obliczane, tzn. stała indeksowa może być zastąpiona **wyrażeniem indeksowym**. Wyrażenie to ma być obliczone, a jego wynik decyduje o tym, która składowa zostanie wybrana. Wspomniana powyżej możliwość stanowi niezwykle mocne narzędzie programowania, lecz jednocześnie umożliwia popełnianie jednego z najczęściej spotykanych błędów: wartość wynikowa może znaleźć się poza przedziałem określonym jako możliwy zakres indeksu tablicy. Będziemy zakładali, że komputer sygnalizuje odpowiednie ostrzeżenie w przypadku niepoprawnego dostępu do nie istniejącej składowej tablicy.

Typ indeksujący powinien być typem skalarnym, tzn. typem niezłożonym, na którym określona jest pewna relacja porządkująca. Jeśli typ podstawowy tablicy jest również typem uporządkowanym, to dostajemy naturalne uporządkowanie typu tablicowego. Naturalne uporządkowanie dwóch tablic jest zdeterminowane dwoma odpowiadającymi sobie nierównymi składowymi o najniższych możliwych indeksach. Formalnie wyraża się to następująco:
Dla danych dwóch tablic x i y relacja $x < y$ zachodzi wtedy i tylko wtedy,

gdy istnieje indeks k taki, że $x[k] < y[k]$ oraz $x[i] = y[i]$ (1.13)
dla wszystkich $i < k$.

Na przykład

(2, 3, 5, 7, 9) < (2, 3, 5, 7, 11)
'LABEL' < 'LIBEL'

W większości zastosowań nie zakłada się istnienia relacji porządkującej na typach tablicowych.

Moc typu złożonego otrzymuje się jako iloczyn mocy jego składowych. Ponieważ wszystkie składowe typu tablicowego A są tego samego typu podstawowego B, otrzymujemy

$$\text{moc}(A) = (\text{moc}(B))^n \tag{1.14}$$

gdzie $n = \text{moc}(I)$, I zaś jest typem indeksującym tablicy.

Poniższy krótki program ilustruje użycie selektora tablicowego. Zadaniem programu jest znalezienie najmniejszego indeksu i składowej o wartości x. Poszukiwania dokonuje się, przeglądając sekwencyjnie tablicę a, zadeklarowaną jako

```
var a: array[1..N] of T; {N > 0}
i := 0;
repeat i := i + 1 until (a[i] = x) ∨ (i = N);                    (1.15)
if a[i] ≠ x then „nie ma takiego elementu w a"
```

Inny wariant tego programu ilustruje znaną metodę z **wartownikiem** (ang. *sentinel*), ustawionym na końcu tablicy. Zastosowanie wartownika znacznie upraszcza warunek zakończenia iteracji.

```
var a: array[1..N + 1] of T;
i := 0; a[N + 1] := x;
repeat i := i + 1 until a[i] = x;                                (1.16)
if i > N then „nie ma takiego elementu w a"
```

Przypisanie $a[N+1] := x$ jest przykładem **aktualizacji selektywnej** (ang. *selective updating*), tzn. zmiany wybranej składowej zmiennej złożonej. Niezależnie od tego, jak często powtórzyło się wykonanie instrukcji $i := i + 1$, spełniony jest warunek

$$a\,[j] \neq x, \qquad \text{dla} \qquad j = 1 \ldots i-1$$

Warunek ten jest spełniony w obu wersjach (1.15) i (1.16); nazywany jest **niezmiennikiem pętli** (ang. *loop invariant*).

Jeżeli elementy tablicy zostały wcześniej uporządkowane (posortowane), przeszukiwanie można znacznie przyspieszyć. W takim przypadku najczęściej stosuje się metodę połowienia przedziału, w którym ma się znajdować poszukiwany element. Metodę tę, zwaną **przeszukiwaniem połówkowym** (przeszukiwaniem binarnym; ang. *binary search*), ilustruje (1.17). Przy każdym powtórzeniu dzieli się na połowę przedział między indeksami i oraz j. Górną granicą wymaganej liczby porównań jest $\lceil \log_2(N) \rceil$.

```
i := 1;  j := N;
repeat k := (i + j) div 2;
   if x > a[k] then i := k + 1 else j := k - 1                   (1.17)
until (a[k] = x) ∨ (i > j)
```

(Odpowiednim warunkiem niezmienniczym na wejściu do instrukcji iteracyjnej **repeat** jest

$a[h] < x$, dla $h = 1 \ldots i-1$
$a[h] > x$, dla $h = j+1 \ldots N$

W związku z tym, jeśli wykonanie programu kończy się przy $i > j$, oznacza to, że nie istnieje $a[h] = x$ takie, że $1 \leqslant h \leqslant N$).

Składowe typów tablicowych mogą być również złożone. Tablica, której składowe są również tablicami, jest zwana **macierzą** (ang. *matrix*). Na przykład

M: **array** [1..10] **of** *Wiersz*

jest tablicą o dziesięciu składowych (wierszach), z których każda składa się z pięciu składowych typu *real*, i nazywa się macierzą 10×5 o składowych rzeczywistych. Selektory mogą być konkatenowane tak, że

$M[i]\,[j]$

oznacza j-tą składową wiersza $M(i)$, który jest i-tą składową macierzy M. Oznaczenie to skraca się zwykle do postaci

$M[i, j]$

Podobnie deklaracja

M: **array** [1..10] **of array** [1..5] **of** *real*

może być napisana w sposób bardziej zwarty w postaci

M: **array** [1..10, 1..5] **of** *real*

Gdy pewna operacja ma być wykonana dla wszystkich składowych tablicy (lub kilku kolejnych składowych), wygodnie jest stosować instrukcję „dla", co ilustruje poniższy przykład.

Załóżmy, że ułamek f jest reprezentowany za pomocą tablicy d w ten sposób, że

$$f = \sum_{i=1}^{k-1} d_i * 10^{-i}$$

tzn. przez swoje $(k-1)$-cyfrowe rozwinięcie dziesiętne. Następnie f ma zostać podzielone przez 2. Dokonuje się tego, powtarzając operację dzielenia dla wszystkich $k-1$ cyfr, począwszy od $i = 1$. Operacja ta polega na podzieleniu cyfry przez 2, z uwzględnieniem ewentualnego przeniesienia z poprzedniej pozycji i zachowaniem ewentualnej reszty r do następnego kroku (por. (1.18)).

$r := 10 * r + d[i];$
$d[i] := r$ **div** $2;$ (1.18)
$r := r - 2 * d[i]$

PROGRAM 1.1
Obliczanie potęg liczby 2

```
program potęga (output);
{reprezentacja dziesiętna ujemnych potęg liczby 2}
const n = 10;
type cyfra = 0..9;
var i, k, r: integer;
    d: array [1..n] of cyfra;
begin for k := 1 to n do
    begin write ('.'); r := 0;
      for i := 1 to k - 1 do
          begin r := 10 * r + d[i]; d[i] := r div 2;
                r := r - 2 * d[i]; write(chr(d[i] + ord('0')))
          end;
      d [k] := 5; writeln('5')
    end
end.
```

Postępowanie to zastosowano w programie 1.1 w celu otrzymania tablicy ujemnych potęg dwójki. Powtarzanie operacji połowienia służących do obliczenia wartości 2^{-1}, 2^{-2}, ..., 2^{-n} jest ponownie wyrażone za pomocą instrukcji „dla", prowadząc do zagnieżdżenia dwóch takich instrukcji. Wyniki programu dla $n = 10$ są następujące:

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.7. Rekordy

Najbardziej ogólną metodą tworzenia typów złożonych jest łączenie w typ złożony elementów o dowolnych, być może złożonych typach. Przykładami z matematyki są liczby zespolone złożone z dwóch liczb rzeczywistych bądź współrzędne punktów złożone z dwóch lub więcej liczb rzeczywistych w zależności od wymiaru przestrzeni. Przykładem z przetwarzania danych jest opis osoby za pomocą kilku istotnych cech charakterystycznych, takich jak imię, nazwisko, data urodzenia, płeć i stan cywilny.

W matematyce taki typ złożony zwie się **iloczynem kartezjańskim** typów składowych. Wypływa to z faktu, że zbiór wartości definiujący ten typ składa się ze wszystkich możliwych kombinacji wartości wziętych odpowiednio ze wszystkich typów składowych. Liczba takich kombinacji jest równa iloczynowi liczb elementów wszystkich typów składowych, czyli moc typu złożonego równa jest iloczynowi mocy typów składowych.

W przetwarzaniu danych typy złożone, takie jak opisy osób czy innych obiektów, służą do zapisywania i rejestracji ukierunkowanych problemowo charakterystyk tych osób czy obiektów; są one zazwyczaj przechowywane w plikach i „bankach danych". Fakt ten tłumaczy stosowanie angielskiego słowa **record** (zapis, rejestr, nagranie, przechowywany zestaw informacji) do oznaczania tego rodzaju danych zamiast terminu „iloczyn kartezjański".

Ogólnie, **typ rekordowy** T jest zdefiniowany w następujący sposób:

---

type T = **record** $s_1 : T_1$;
$s_2 : T_2$;
...
$s_n : T_n$
end (1.19)

$\text{moc}(T) = \text{moc}(T_1) * \ldots * \text{moc}(T_n)$

PRZYKŁADY

type *Zespolona* = **record** *re*: *real*;
im: *real*
end

type *Data* = **record** *dzień*: 1..31;
miesiąc: 1..12;
rok: 1..2000
end

type *Osoba* = **record** *nazwisko*: *alfa*;
imię: *alfa*;
dataurodzenia: *Data*;
płeć: (*mężczyzna*, *kobieta*);
stancywilny: (*wolny*, *żonaty*, *owdowiały*, *rozwiedziony*)
end

□

Za pomocą **konstruktora rekordowego** można utworzyć wartość typu T, a następnie przypisać ją jakiejś zmiennej tego typu:

$$x := T(x_1, x_2, \ldots, x_n) \tag{1.20}$$

gdzie x_i są odpowiednio wartościami z typów składowych T_i.

Dla danych zmiennych rekordowych

z: *Zespolona*
d: *Data*
p: *Osoba*

konkretne wartości mogą być przypisane np. tak, jak to zilustrowano na rys. 1.3:

z := *Zespolona* (1.0, −1.0)
d := *Data* (1,4,1973)
p := *Osoba* ('WIRTH', 'CHRIS', *Data* (18,1,1966), *mężczyzna*, *wolny*)

Zespolona z
1.0
− 1.0

Data d
1
4
1973

Osoba p		
WIRTH		
CHRIS		
18	1	1966
mężczyzna		
wolny		

RYSUNEK 1.3
Rekordy typów *Zespolona*, *Data* i *Osoba*

Identyfikatory $s_1, ..., s_n$ wprowadzone w definicji typu rekordowego stanowią nazwy przydzielone poszczególnym składowym zmiennych tego typu; stosuje się je w **selektorach rekordowych** odnoszących się do zmiennych rekordowych. Dla danej zmiennej x: T, jej i-ta składowa jest oznaczona przez

$$x \,.\, s_i \tag{1.21}$$

Aktualizację selektywną x prowadzi się przy użyciu tego samego oznacznika selektora stojącego po lewej stronie instrukcji przypisania:

$x \,.\, s_i := x_i$

gdzie x_i jest wartością (wyrażeniem) typu T_i.

Dla danych zmiennych rekordowych

z: *Zespolona*
d: *Data*
p: *Osoba*

selektory niektórych składowych są następujące:

z.im (typu *real*)
d.miesiąc (typu 1..12)
p.nazwisko (typu *alfa*)
p.dataurodzenia (typu *Data*)
p.dataurodzenia.dzień (typu 1..31)

Przykład typu *Osoba* dowodzi, że typy składowe rekordu mogą również mieć pewną złożoną strukturę. Selektory można zatem składać. Oczywiście operację składania typów o różnych składowych można wykonywać wielokrotnie (wielopoziomowo). Na przykład *i*-ta składowa tablicy *a*, będącej składową zmiennej rekordowej *r*, jest oznaczana przez

$r.a[i]$

a składowa o selektorze *s* *i*-tej składowej rekordowej tablicy *a* jest oznaczana przez

$a[i].s$

Charakterystyczną cechą iloczynu kartezjańskiego jest to, że zawiera on *wszystkie* kombinacje elementów typów składowych. Trzeba jednak pamiętać, że w praktyce nie wszystkie muszą być „prawidłowe", tzn. sensowne. Na przykład zdefiniowany powyżej typ *Data* zawiera wartości

(31,4,1973) i (29,2,1815)

które odpowiadają datom nie istniejących nigdy dni. Jak widać, definicja tego typu nie odzwierciedla sytuacji rzeczywistej. Niemniej jednak praktycznie jest dostatecznie bliska rzeczywistości; od programisty zależy, aby bezsensowne wartości nigdy nie wystąpiły podczas wykonania programu.

Przytoczony poniżej krótki fragment programu ilustruje użycie zmiennych rekordowych. Jego zadaniem jest zliczenie „Osób" reprezentowanych w tablicy *a* jako kobiety stanu wolnego.

```
var a: array [1..N] of Osoba
    licznik: integer;
licznik := 0;
for i := 1 to N do                                          (1.22)
    if (a[i].płeć = kobieta) ∧ (a[i].stancywilny = wolny)
    then licznik := licznik + 1
```

Niezmiennikiem pętli jest tu

$licznik = C(i)$

gdzie $C(i)$ jest liczbą elementów podzbioru $a_1 \dots a_i$, będących kobietami stanu wolnego.

W innym wariancie powyższego fragmentu programu korzysta się z konstrukcji zwanej **instrukcją wiążącą with**:

```
for i := 1 to N do
    with a[i] do                                            (1.23)
if  (płeć = kobieta) ∧ (stancywilny = wolny) then
    licznik := licznik + 1
```

Konstrukcja **with** r **do** s oznacza, że można używać nazw selektorów z typu zmiennej r bez poprzedzenia ich nazwą zmiennej, ponieważ wiadomo, że będą się do niej odnosiły. Użycie instrukcji **with** skraca więc tekst programu, jak również zapobiega wielokrotnemu obliczaniu od nowa składowej indeksowanej $a[i]$.

W następnym przykładzie założymy, że (być może w celu szybszego wyszukiwania) pewne grupy osób w tablicy a są ze sobą w pewien sposób powiązane. Informacja wiążąca jest reprezentowana przez dodatkową składową rekordu *Osoba*, nazwaną **łączem** (wiązaniem; ang. *link*). Łącza wiążą ze sobą rekordy w liniowy łańcuch, tak że można łatwo odszukać poprzednika i następnika każdej osoby. Interesującą własnością tej metody wiązania jest to, że łańcuch może być przeglądany w obu kierunkach na podstawie tylko jednej liczby umieszczonej w każdym rekordzie. Poniżej przedstawiono funkcjonowanie tej metody.

Załóżmy, że indeksy trzech kolejnych elementów łańcucha są równe i_{k-1}, i_k, i_{k+1}. Wartość łącza dla k-tego elementu wybiera się jako $i_{k+1} - i_{k-1}$. Przeglądając „w przód" łańcuch elementów, i_{k+1} określa się na podstawie dwóch aktualnych zmiennych indeksowanych $x = i_{k-1}$ i $y = i_k$ jako

$$i_{k+1} = x + a[y].łącze$$

a podczas przeglądania „wstecz" i_{k-1} określone jest na podstawie $x = i_{k+1}$ i $y = i_k$ jako

$$i_{k-1} = x - a[y].łącze$$

Przykładem może być powiązanie ze sobą w łańcuchy wszystkich osób tej samej płci (zob. tabl. 1.2).

TABLICA 1.2
Tablica o elementach typu *Osoba*

Imię	Płeć	Łącze
1 Karolina	K	2
2 Krzysztof	M	2
3 Paulina	K	5
4 Robert	M	3
5 Janusz	M	3
6 Jadwiga	K	5
7 Ryszard	M	5
8 Maria	K	3
9 Anna	K	1
10 Mateusz	M	3

Strukturę rekordu i strukturę tablicy charakteryzuje wspólnie cecha „dostępu swobodnego". Rekord jest strukturą bardziej ogólną w tym sensie, że nie ma tu wymagania, aby wszystkie typy składowe były identyczne. Jednakże stosowanie tablicy zapewnia większą elastyczność, ponieważ selektory składowych tablicy mogą być wartościami obliczanymi w programie (reprezentowanymi przez wyrażenia), podczas gdy selektory składowych rekordu są sztywno ustalonymi identyfikatorami wprowadzonymi w definicji odpowiedniego typu.

1.8. Rekordy z wariantami

W praktyce okazuje się zazwyczaj wygodne i naturalne traktowanie dwóch typów po prostu jako wariantów tego samego typu. Na przykład typ *Współrzędne* można traktować jako sumę dwóch wariantów, mianowicie współrzędnych kartezjańskich i biegunowych, których składowymi są odpowiednio (a) dwie wielkości liniowe i (b) wielkość liniowa i kątowa. W celu rozpoznania wariantu, który aktualnie jest wartością zmiennej, wprowadza się trzecią składową zwaną **wyróżnikiem typu** albo **polem znacznikowym**.

```
type Współrzędne =
    record case rodzaj: (kartezjańskie, biegunowe) of
                kartezjańskie: (x, y: real);
                biegunowe: (r: real; φ: real)
    end
```

W tym przypadku nazwą pola znacznikowego jest *rodzaj*, a nazwy współrzędnych stanowią albo x i y – wartości współrzędnych kartezjańskich, albo r i φ – wartości współrzędnych biegunowych.

Zbiór wartości typu *Współrzędne* jest **sumą** dwóch typów:

$T_1 = (x, y\colon real)$
$T_2 = (r\colon real; \varphi\colon real)$

a jego moc jest sumą mocy T_1 i T_2.

Bardzo często jednak nie mamy do czynienia z dwoma całkowicie różnymi typami, lecz raczej z typami o częściowo identycznych składowych. Fakt ten zadecydował o nazwie takiego typu – **rekord z wariantami** (ang. *variant record*). Przykładem jest typ *Osoba* zdefiniowany w poprzednim punkcie, w którym informacje zapisywane w pliku zależą od płci osoby. Na przykład dla mężczyzny charakterystycznymi informacjami mogą być jego waga oraz to, czy jest brodaty, podczas gdy dla kobiety informacjami tymi mogą być trzy charakterystyczne wymiary figury (natomiast waga może pozostać nieujawniona). Definicja takiego typu wyglądałaby więc następująco:

```
type Osoba =
    record nazwisko, imię: alfa;
           dataurodzenia: data;
           stancywilny: (wolny, żonaty, owdowiały, rozwiedziony);
       case płeć:    (mężczyzna, kobieta) of
       mężczyzna:    (waga: real;
                      brodaty: Boolean);
       kobieta: (wymiary: array [1..3] of integer)
    end
```

A oto jak wygląda ogólna postać definicji rekordu z wariantami:

$$
\begin{array}{l}
\textbf{type } T = \\
\quad \textbf{record } s_1 : T_1; \ldots; s_{n-1} : T_{n-1}; \\
\qquad \textbf{case } s_n : T_n \textbf{ of} \\
\qquad\quad c_1 : (s_{1,1} : T_{1,1}; \ldots; s_{1,n_1} : T_{1,n_1}); \\
\qquad\quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\qquad\quad c_m : (s_{m,1} : T_{m,1}; \ldots; s_{m,n_m} : T_{m,n_m}) \\
\quad \textbf{end}
\end{array}
\tag{1.24}
$$

Identyfikatory s_i i s_{ij} są nazwami składowych dla typów T_i i T_{ij}, a s_n jest nazwą wyróżniającego pola znacznikowego o typie T_n. Stałe $c_1 \ldots c_m$ oznaczają wartości typu (skalarnego) T_n. Zmienna x typu T zawiera składowe

$x.s_1, x.s_2, \ldots, x.s_n, x.s_{k,1}, \ldots, x.s_{k,n_k}$

wtedy i tylko wtedy, gdy (bieżąca) wartość $x.s_n$ jest równa c_k. Składowe $x.s_1, \ldots, x.s_n$ tworzą **część wspólną** m wariantów.

Wynika stąd, że użycie selektora składowej $x.s_{k,h}$ $(1 \leqslant h \leqslant n_k)$, gdy $x.s_n \neq c_k$, należy traktować jako poważny błąd programu, gdyż może to prowadzić do wielu paradoksalnych konsekwencji. Dla zdefiniowanego powyżej typu *Osoba* może to bowiem spowodować np. zapytanie, czy pani jest brodata, lub (w przypadku aktualizacji selektywnej) żądanie, aby taką była.

Jak widać, rekordy z wariantami wymagają niezwykle uważnego programowania. Odpowiednie operacje na poszczególnych wariantach najlepiej grupować w instrukcję wybiórczą, tzw. **instrukcję wyboru**, której struktura odzwierciedla strukturę definicji rekordu z wariantami.

$$
\begin{array}{l}
\textbf{case } x.s_n \textbf{ of} \\
\quad c_1 : S_1; \\
\quad c_2 : S_2; \\
\quad \ldots \\
\quad c_m : S_m \\
\textbf{end}
\end{array}
\tag{1.25}
$$

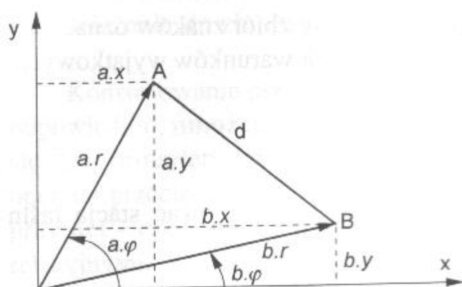
S_k zastępuje instrukcję obowiązującą dla przypadku, kiedy x przyjmuje postać wariantu k, tzn. jest ono wybierane do wykonania wtedy i tylko wtedy, gdy pole znacznikowe $x.s_n$ ma wartość c_k. W konsekwencji można dosyć łatwo nie dopuścić do niepoprawnego użycia nazw selektorów, sprawdzając, czy każde S_k zawiera co najwyżej selektory

$x.s_1, \ldots, x.s_{n-1}$

oraz

$x.s_{k,1}, \ldots, x.s_{k,n_k}$

Przytoczony poniżej fragment programu ma za zadanie obliczenie odległości między punktami A i B zadanymi w postaci zmiennych a i b typu *Współrzędne*, będącego rekordem z wariantami. Procedura obliczeń jest odpowiednio odmienna dla każdej z czterech możliwych kombinacji określania zmiennych za pomocą współrzędnych kartezjańskich lub biegunowych (zob. rys. 1.4).



RYSUNEK 1.4
Współrzędne kartezjańskie i biegunowe

```
case a.rodzaj of
    kartezjańskie: case b.rodzaj of
                   kartezjańskie: d := sqrt(sqr(a.x − b.x) + sqr(a.y − b.y));
                   biegunowe:     d := sqrt(sqr(a.x − b.r * cos(b.φ))
                                            + sqr(a.y − b.r * sin(b.φ)))
                   end;
    biegunowe:     case b.rodzaj of
                   kartezjańskie: d := sqrt(sqr(a.r * sin(a.φ) − b.x)
                                            + sqr(a.r * cos(a.φ) − b.y));
                   biegunowe:     d := sqrt(sqr(a.r) + sqr(b.r)
                                            − 2 * a.r * b.r * cos(a.φ − b.φ))
                   end
end
```

1.9. Zbiory

Trzecią podstawową strukturą danych – oprócz tablicy i rekordu – jest **struktura zbioru** (ang. *set structure*). Definiuje się ją za pomocą deklaracji o podanej poniżej postaci:

type T = **set of** T_0 (1.26)

Wartościami zmiennej x typu T są zbiory elementów typu T_0. Zbiór wszystkich możliwych podzbiorów elementów T_0 nazywa się **zbiorem potęgowym** T_0. W związku z tym typ T stanowi zbiór potęgowy swego **typu podstawowego** T_0.

PRZYKŁADY

type *zbiórcałk* = **set of** 0..30
type *zbiórznaków* = **set of** *char*
type *stantaśmy* = **set of** *wyjątek*
□

Podstawę dla drugiego przykładu stanowi standardowy zbiór znaków oznaczony typem *char*; podstawą dla trzeciego przykładu jest zbiór warunków wyjątkowych, który można zdefiniować jako typ skalarny

type *wyjątek* = (*niezaładowana*, *ręczna*, *parzystość*, *skos*)

opisujący różne stany wyjątkowe, w jakich może się znajdować stacja taśm magnetycznych. Dla danych zmiennych

zc : *zbiórcałk*
zz : *zbiórznaków*
t : **array** [1..6] **of** *stantaśmy*

można wykonać przypisania dla poszczególnych wartości typu zbiorowego, np.*

zc := [1, 4, 9, 16, 25]
zz := ['+', '−', '*', '/']
t[3] := [*ręczna*]
t[5] := [ ]
t[6] := [*niezaładowana*..*skos*]

Wartość przypisania zmiennej *t*[3] stanowi zbiór jednoelementowy zawierający pojedynczy element *ręczna*; zmiennej *t*[5] jest przypisany zbiór pusty, co oznacza, że piąta stacja taśm została przywrócona do stanu normalnej pracy (nie wyjątkowego), natomiast zmiennej *t*[6] przypisano zbiór złożony z wszystkich czterech stanów wyjątkowych.

Moc typu zbiorowego T jest określona wzorem

$$\mathrm{moc}(T) = 2^{\mathrm{moc}\,(T_0)} \tag{1.27}$$

Wzór ten wynika bezpośrednio z faktu, że każdy z elementów T_0 musi być reprezentowany przez jedną z dwóch wartości: „jest" lub „nie ma" oraz stąd, że wszystkie elementy są wzajemnie niezależne. Dla efektywnej i ekonomicznej realizacji jest istotne, aby typ podstawowy był nie tylko skończony, ale aby jego moc była stosunkowo niewielka.

* W przeciwieństwie do ogólnie przyjętej notacji do oznaczania zbiorów stosujemy nawiasy kwadratowe zamiast klamrowych. Nawiasy klamrowe rezerwujemy do oznaczania komentarzy w programach.

Na wszystkich typach zbiorowych są określone następujące operatory elementarne:

$*$	przecięcie zbiorów
$+$	suma zbiorów
$-$	różnica zbiorów
in	należenie do zbioru

Konstruowanie przecięcia (iloczynu) i sumy zbiorów nazywa się też często, odpowiednio, **mnożeniem** i **dodawaniem** zbiorów; zgodnie z tą analogią określa się też priorytety operatorów zbiorowych: największy priorytet otrzymuje operator przecięcia zbiorów, następnie – operator sumy i różnicy, najmniejszy zaś priorytet – operator należenia do zbioru, klasyfikowany na równi z operatorami relacyjnymi. Poniżej podano przykłady wyrażeń zbiorowych i ich odpowiedników z nawiasami:

$r*s+t = (r*s)+t$
$r-s*t = r-(s*t)$
$r-s+t = (r-s)+t$
$x \textbf{ in } s+t = x \textbf{ in } (s+t)$

Naszym pierwszym przykładem na zastosowanie struktury zbioru jest program prostego analizatora leksykalnego dla pewnego translatora. Będziemy zakładali, że zadaniem analizatora leksykalnego jest przetłumaczenie ciągu znaków na ciąg jednostek tekstowych tłumaczonego języka, tj. **atomów leksykalnych** (leksemów), zwanych tu **symbolami**. Analizator leksykalny powinien być procedurą, która przy każdym wywołaniu wczytuje pewną liczbę znaków wejściowych, potrzebnych do wygenerowania następnego symbolu wyjściowego. Poniżej podajemy szczegółowo zasady tłumaczenia.

(1) Zbiór symboli wyjściowych zawiera takie elementy, jak *identyfikator*, *liczba*, *mniejszyrówny*, *większyrówny*, *stajesię* i inne, odpowiadające różnym pojedynczym znakom, takim jak $+$, $-$, $*$ itd.
(2) Symbol *identyfikator* generuje się po przeczytaniu ciągu liter i cyfr, zaczynającego się od litery.
(3) Symbol *liczba* generuje się po przeczytaniu ciągu cyfr.
(4) Symbole *mniejszyrówny*, *większyrówny* i *stajesię* generuje się odpowiednio po przeczytaniu następujących par znaków: $<=$, $>=$, $:=$.
(5) Odstępy i symbole końca wiersza są opuszczane.

Mamy do dyspozycji prostą procedurę *czytaj*(x), która pobiera kolejny znak z ciągu wejściowego i przypisuje go zmiennej x. Wynikowy symbol wyjściowy jest przypisany zmiennej globalnej o nazwie *sym*. Ponadto korzysta się ze

zmiennych globalnych *id* i *licz*, których przeznaczenie wyniknie z analizy programu 1.2, oraz ze zmiennej *zn*, reprezentującej aktualnie analizowany znak ciągu wejściowego. *S* oznacza odwzorowanie ze zbioru znaków w zbiór symboli, tzn. jednowymiarową tablicę symboli, dla której dziedziną indeksu są wszystkie znaki różne od liter i cyfr. Użycie zbiorów znaków ilustruje, w jaki sposób można zaprogramować analizator leksykalny niezależnie od uporządkowania znaków w zadanym zbiorze znaków.

PROGRAM 1.2
Analizator leksykalny

```
var zn: char;
    sym: symbol;
    licz: integer;
    id: record
          k: 0..maxk;
          a: array [1..maxk] of char
        end;
procedure analeks;
   var zn1 : char;
begin {pomiń spacje}
   while zn = '␣' do read(zn);
   if zn in ('A'..'Z') then
      with id do
      begin sym := identyfikator; k := 0;
         repeat if k < maxk then;
                begin k := k + 1; a[k] := zn
                end;
            read(zn)
         until ¬ (zn in ['A'..'Z', '0'..'9'])
      end else
   if zn in ('0'..'9') then
      begin sym = liczba; licz := 0;
         repeat licz := 10 * licz + ord(zn) − ord('0');
                read(zn)
         until ¬ (zn in ['0'..'9'])
      end else
   if zn in ['<', ':', '>') then
      begin zn1 := zn; read(zn);
         if zn = '=' then
         begin
            if zn1 = '<' then sym := mr else
            if zn1 = '>' then sym := wr else sym := stajesię;
            read(zn)
```

```
            end
            else sym := S[zn1]
        end else
    begin {inne symbole}
        sym := S[zn]; read(zn)
    end
end {analeks}
```

Drugi przykład ilustruje problem właściwego zaplanowania rozkładu zajęć. Załóżmy, że mamy M studentów, z których każdy wybrał pewną liczbę spośród N wykładów. Należy stworzyć taki rozkład zajęć, aby nie dopuścić do powstania konfliktów w przypadku zaplanowania różnych wykładów w tym samym czasie [1.1].

Ogólnie, konstrukcja rozkładu zajęć jest jednym z najtrudniejszych problemów kombinatorycznych, w których decyzja jest uwarunkowana wieloma czynnikami ograniczającymi. W naszym przykładzie znacznie uprościmy to zagadnienie, nie roszcząc sobie pretensji do rozwiązania rzeczywistego rozkładu zajęć.

Przede wszystkim musimy sobie zdawać sprawę z tego, że w celu odpowiedniego wybrania „równoległych" sesji możemy korzystać ze zbioru danych utworzonego na podstawie rejestracji poszczególnych studentów, a mianowicie z wyliczenia wykładów, które nie mogą być prowadzone jednocześnie. W związku z tym programujemy najpierw proces redukcji danych, korzystając z poniższych deklaracji oraz przyjmując, że studenci są ponumerowani liczbami całkowitymi od 1 do M, wykłady zaś – od 1 do N.

```
type wykład = 1..N;
     student = 1..M;
     wybór = set of wykład;
var  s: wykład;                                              (1.28)
     i: student;
     rejestracja: array [student] of wybór;
     konflikt: array [wykład] of wybór;
```

{Na podstawie rejestracji poszczególnych studentów określ zbiór wykładów powodujących konflikt}

```
for s := 1 to N do konflikt[s] := [ ];
for i := 1 to M do
    for s := 1 to N do
    if s in rejestracja[i] then
        konflikt[s] := konflikt[s] + rejestracja[i]
```

(Zwróćmy uwagę, że *s* **in** *konflikt* [*s*] jest konsekwencją tego algorytmu).

Główne zadanie polega teraz na stworzeniu rozkładu zajęć, tzn. listy sesji, z których każda jest zbiorem wykładów nie dającym sytuacji konfliktowej. Z całego zbioru wykładów wybieramy odpowiednie, nie powodujące konfliktu podzbiory wykładów, dopóty odejmując je od zmiennej zbiorowej o nazwie *reszta*, dopóki ten zbiór reszty wykładów nie stanie się pusty.

```
var k: integer;
    reszta, sesja: wybór;
    rozkładzajęć: array [1..N] of wybór;
k:= 0; reszta:= [1..N];
while reszta ≠ [ ] do                                          (1.29)
  begin sesja:= kolejny dobry rozkład;
        reszta:= reszta − sesja;
        k:= k + 1; rozkładzajęć [k]:= sesja
  end
```

W jaki sposób tworzymy „kolejny dobry rozkład"? Na początek możemy wybrać dowolny wykład ze zbioru reszty wykładów. Następnie wybór kolejnych kandydatów może być ograniczony do tych wykładów z reszty, które nie dają konfliktu z wykładami wybranymi początkowo. Zbiór ten oznaczono w programie identyfikatorem *zbiórpróbny*. Rozważając kandydaturę konkretnego wykładu ze zbioru próbnego, widać, że jego wybór zależy od tego, czy przecięcie zbioru wykładów już wybranych ze zbiorem konfliktowym dla tego kandydata jest puste. Rozważania te prowadzą do następującego rozwinięcia instrukcji „*sesja*:= *kolejny dobry rozkład*":

```
var s, t: wykład;
    zbiórpróbny: wybór;
begin s:= 1;
    while ¬ (s in reszta) do s:= s + 1;
    sesja:= [s]; zbiórpróbny:= reszta − konflikt[s];           (1.30)
    for t:= 1 to N do
        if t in zbiórpróbny then
        begin if konflikt[t] * sesja = [ ] then
                  sesja:= sesja + [t]
        end
end
```

Oczywiście przedstawione rozwiązanie znajdowania „dobrych rozkładów" sesji nie wygeneruje rozkładu zajęć, który byłby optymalny w jakimkolwiek sensie. W krańcowych przypadkach może dojść do tego, że liczba sesji będzie równa liczbie wykładów, nawet jeśli byłoby możliwe znalezienie bardziej optymalnych rozwiązań.

TABLICA 1.3
Podstawowe struktury danych

Struktura	Deklaracja	Selektor	Dostęp do składowych	Typ składowych	Moc
Tablica	a: **array** $[I]$ **of** T_0	$a[i]$ $(i \in I)$	Selektor z obliczanym indeksem i	Wszystkie jednakowe (T_0)	$moc\,(T_0)^{moc(I)}$
Rekord	r: **record** $s_1 : T_1$; $s_2 : T_2$; ... $s_n : T_n$ **end**	$r.s$ $(s \in [s_1, \ldots, s_n])$	Selektor z deklarowaną nazwą składowej s	Mogą być różne	$\prod_{i=1}^{n} moc\,(T_i)$
Zbiór	s: **set of** T_0	nie istnieje	Test przynależności za pomocą operatora relacyjnego **in**	Wszystkie jednakowe (typu skalarnego T_0)	$2^{moc(T_0)}$

1.10. Reprezentacja tablic, rekordów i zbiorów

Istota stosowania pojęć abstrakcyjnych w programowaniu polega na tym, że programy można układać, rozumieć i sprawdzać na podstawie praw rządzących tymi pojęciami, bez konieczności odwoływania się do wiedzy o tym, jak te pojęcia są realizowane i reprezentowane w konkretnej maszynie cyfrowej. Niemniej jednak znajomość szeroko stosowanych metod reprezentowania podstawowych pojęć programowania, takich jak struktury danych, może być bardzo cenna dla programisty. Umożliwi mu ona powzięcie właściwych decyzji przy układaniu programu i wyborze danych, uwarunkowanych nie tylko abstrakcyjnymi właściwościami tych struktur, lecz również ich realizacją w dostępnych maszynach cyfrowych przy uwzględnieniu ich szczególnych możliwości i ograniczeń.

Problem reprezentacji danych polega na znalezieniu odwzorowania struktur abstrakcyjnych na pamięć maszyny. Pamięć komputera jest – w pierwszym przybliżeniu – wektorem pojedynczych komórek pamięci, zwanych **słowami**. Indeksy słów zwane są **adresami**.

var *pamięć*: **array** [*adres*] **of** *słowo* (1.31)

Moce typów *adres* i *słowo* są rozmaite dla różnych komputerów. Specjalnym zagadnieniem jest zmienność mocy typu *słowo*. Jego logarytm nazywa się **długością słowa** – jest to liczba bitów, z których składa się jedna komórka pamięci.

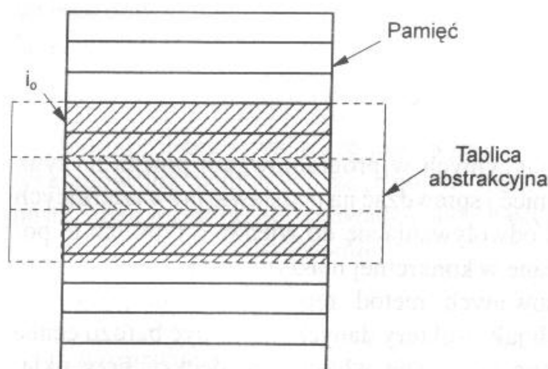
1.10.1. Reprezentacja tablic

Reprezentacja tablicy jest pewnym odwzorowaniem tablicy (abstrakcyjnej) o składowych typu T na pamięć będącą wektorem o składowych typu *słowo*.

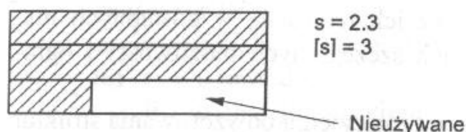
Tablica powinna być odwzorowana w ten sposób, aby obliczanie adresów składowych było możliwie najprostsze, a więc najbardziej ekonomiczne. Adres, czyli indeks i pamięci dla j-tej składowej, oblicza się za pomocą funkcji liniowej

$$i = i_0 + j * s \tag{1.32}$$

gdzie i_0 jest adresem pierwszej składowej, s zaś jest liczbą słów „zajmowanych" przez każdą składową. Ponieważ słowo jest z definicji najmniejszą, bezpośrednio dostępną jednostką pamięci, wysoce pożądane jest, aby s było liczbą całkowitą (w najprostszym przypadku $s = 1$). Jeśli s nie jest liczbą całkowitą (a jest to normalna sytuacja), to s zaokrągla się zwykle w górę do najbliższej liczby całkowitej $\lceil s \rceil$. Każda składowa tablicy zajmuje wtedy $\lceil s \rceil$ słów, gdzie $\lceil s \rceil - s$ pozostaje nieużywane (zob. rys. 1.5 i 1.6). Zaokrąglanie w górę liczby słów jednej składowej nazywa



RYSUNEK 1.5
Odwzorowanie tablicy na pamięć



RYSUNEK 1.6
Reprezentacja dopełnieniowa elementu tablicy

się **dopełnianiem** (ang. *padding*). Współczynnik wykorzystania pamięci u jest to stosunek minimalnej liczby słów pamięci potrzebnych do reprezentacji struktury do liczby słów pamięci rzeczywiście użytych:

$$u = \frac{s}{s'} = \frac{s}{\lceil s \rceil} \tag{1.33}$$

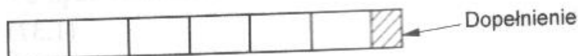
Z jednej strony, twórca translatora będzie dążył do tego, aby współczynnik wykorzystania pamięci był możliwie najbliższy jedności; z drugiej strony

jednakże, uzyskanie dostępu do fragmentów słowa jest kłopotliwe i stosunkowo nieefektywne. W związku z tym niezbędne jest tu przyjęcie rozwiązania kompromisowego. Rozważyć trzeba następujące aspekty tego zagadnienia.

(1) Dopełnianie zmniejsza wykorzystanie pamięci.
(2) Rezygnacja z dopełniania stwarza konieczność przyjęcia nieefektywnej metody dostępu do części słowa.
(3) Zastosowanie dostępu do części słowa zwiększa długość programu wynikowego (przetłumaczonego programu), co niweczy oszczędności uzyskane przez rezygnację z dopełniania.

W rzeczywistości aspekty 2 i 3 przeważają na korzyść automatycznego stosowania dopełniania. Trzeba pamiętać, że współczynnik wykorzystania pamięci będzie zawsze $u > 0{,}5$, jeśli $s > 0{,}5$. Jednakże, jeżeli $s \leqslant 0{,}5$, to współczynnik wykorzystania pamięci można znacznie zwiększyć dzięki umieszczaniu w każdym słowie więcej niż jednej składowej tablicy. Metodę tę nazywa się **upakowywaniem** (ang. *packing*). Jeżeli w jednym słowie upakuje się n składowych tablicy, współczynnik wykorzystania pamięci (zob. rys. 1.7) wynosi

$$u = \frac{n \cdot s}{\lceil n \cdot s \rceil} \tag{1.34}$$



RYSUNEK 1.7
Upakowanie sześciu składowych w jednym słowie

Dostęp do i-tej składowej tablicy upakowanej wiąże się z obliczeniem adresu słowa j, pod którym jest umieszczona żądana składowa, oraz pozycji składowej k w słowie.

$$\begin{aligned} j &= i \ \mathbf{div} \ n \\ k &= i \ \mathbf{mod} \ n = i - j * n \end{aligned} \tag{1.35}$$

W większości języków programowania nie pozostawia się programiście możliwości sterowania reprezentacją abstrakcyjnych struktur danych. Jednakże powinna istnieć możliwość przekazania translatorowi informacji, że upakowanie byłoby pożądane przynajmniej w tych przypadkach, w których więcej niż jedna składowa zmieści się w całości w pojedynczym słowie, tzn. wtedy, gdy możliwy do uzyskania stopień oszczędności pamięci przekracza liczbę 2. Wprowadzamy więc konwencję, że żądanie upakowania będzie oznaczane poprzedzeniem symbolu **array** (bądź **rekord**) w deklaracji symbolem **packed**.

PRZYKŁAD
type *alfa* = **packed array** [1..*n*] **of** *char*
□

Możliwość ta jest szczególnie przydatna w przypadku maszyn o długich słowach i stosunkowo wygodnej metodzie dostępu do fragmentów słowa. Istotną cechą przedrostka **packed** jest to, że w żadnej mierze nie zmienia on znaczenia (i poprawności) programu. Oznacza to, że można łatwo wskazać wybór reprezentacji, bez jakiegokolwiek wpływu na zmianę znaczenia programu.

Koszt dostępu do poszczególnych składowych tablicy można częstokroć w dużym stopniu zredukować, jeśli cała tablica jest rozpakowana (lub spakowana) od razu. W tych przypadkach można efektywnie przeglądać sekwencyjnie całą tablicę bez potrzeby obliczania wartości funkcji rozmieszczenia dla każdej składowej. Zakładać więc będziemy istnienie dwóch zdefiniowanych poniżej standardowych procedur *pack* i *unpack*. Załóżmy, że dane są dwie zmienne:

u: **array**[$a..d$] **of** T
p: **packed array** [$b..c$] **of** T

gdzie $a \leqslant b \leqslant c \leqslant d$ są tego samego typu skalarnego. Wówczas

$$pack(u, i, p), \qquad (a \leqslant i \leqslant b - c + d) \tag{1.36}$$

jest równoważne z

$$p[j] := u[j + i - b], \qquad j = b \dots c$$

natomiast

$$unpack(p, u, i), \qquad (a \leqslant i \leqslant b - c + d) \tag{1.37}$$

jest równoważne z

$$u[j + i - b] := p[j], \qquad j = b \dots c$$

1.10.2. Reprezentacja rekordów

Odwzorowanie rekordów na pamięć maszyny (przydzielenie pamięci) dokonuje się przez proste rozmieszczenie jedna za drugą kolejnych składowych rekordów. Adres składowej (pola) r_i względem adresu początku rekordu r zwany jest **adresem względnym** (ang. *offset address*) oznaczanym tu przez k_i. Oblicza się go jako

$$k_i = s_1 + s_2 + \dots + s_{i-1} \tag{1.38}$$

gdzie s_j jest długością (w słowach) j-tej składowej. W przypadku tablicy (wszystkie składowe tego samego typu) otrzymujemy

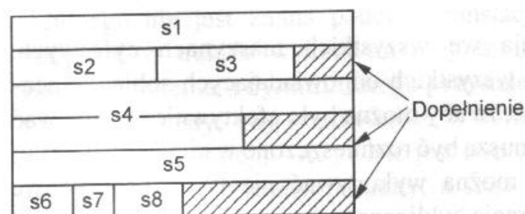
$$s_1 = s_2 = \dots = s_n$$

a zatem

$$k_i = s_1 + \dots + s_{i-1} = (i - 1) \cdot s$$

Ogólność struktury rekordu uniemożliwia zastosowanie tak prostej liniowej funkcji obliczania adresu względnego. To właśnie było przyczyną żądania, aby składowe rekordu były wybierane tylko przez ustalone identyfikatory (selektory). Odpowiednie adresy względne są wtedy znane podczas translacji. Powszechnie znana jest wynikająca stąd większa efektywność dostępu do pól rekordu.

Jeśli więcej niż jedna składowa mieści się całkowicie w jednym słowie pamięci, to znów mamy do czynienia z zagadnieniem upakowywania (zob. rys. 1.8). Żądanie zastosowania upakowywania zaznacza się w programie, poprze-



RYSUNEK 1.8
Reprezentacja upakowanego rekordu

dzając w deklaracji symbol **record** symbolem **packed**. Ponieważ adresy względne oblicza translator, może on również określić adres względny składowej w ramach słowa. Oznacza to, że w wielu maszynach cyfrowych upakowywanie rekordów wiąże się z dużo mniejszą stratą efektywności dostępu do pamięci niż w przypadku upakowywania tablic.

1.10.3. Reprezentacja zbiorów

Zbiór s wygodnie jest reprezentować w pamięci komputera za pośrednictwem jego **funkcji charakterystycznej** $C(s)$. Jest to wektor wartości logicznych, którego i-ta składowa określa występowanie bądź brak wartości i w zbiorze. Rozmiar tego wektora jest określony mocą typu.

$$C(s_i) = (i \textbf{ in } s) \tag{1.39}$$

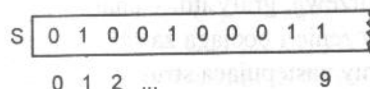
Na przykład zbiór małych liczb całkowitych

$s = [1, 4, 8, 9]$

jest reprezentowany przez ciąg wartości logicznych F (fałsz) i T (prawda):

$C(s) = (\text{F T F F T F F F T T})$

jeśli typem podstawowym zbioru s jest 0..9. W pamięci komputera sekwencja wartości logicznych jest reprezentowana przez tzw. **ciąg bitów** (zob. rys. 1.9).



RYSUNEK 1.9
Reprezentacja zbioru jako ciągu bitów

Reprezentacja zbioru przez jego funkcję charakterystyczną ma tę zaletę, że operacje obliczania sumy, przecięcia i różnicy dwóch zbiorów można zrealizować w maszynie cyfrowej jako elementarne operacje logiczne. Poniższe tożsamości, obowiązujące dla wszystkich elementów i typu podstawowego zbiorów x i y, odzwierciedlają odpowiedniości między operacjami logicznymi a operacjami na zbiorach:

$$\begin{aligned} i\ \mathbf{in}(x + y) &\equiv (i\ \mathbf{in}\ x) \vee (i\ \mathbf{in}\ y) \\ i\ \mathbf{in}(x * y) &\equiv (i\ \mathbf{in}\ x) \wedge (i\ \mathbf{in}\ y) \\ i\ \mathbf{in}(x - y) &\equiv (i\ \mathbf{in}\ x) \wedge \neg (i\ \mathbf{in}\ y) \end{aligned} \tag{1.40}$$

Takie operacje logiczne występują we wszystkich maszynach cyfrowych, a ponadto działają *jednocześnie* na wszystkich odpowiadających sobie elementach (bitach) słowa. Okazuje się więc, że aby można było efektywnie zrealizować podstawowy zbiór operacji, zbiory muszą być rozmieszczone w niewielkiej, stałej liczbie słów pamięci, na których można wykonywać nie tylko podstawowe operacje logiczne, lecz również operacje cyklicznego przesuwania bitów. Sprawdzenie, czy element należy do zbioru, realizuje się za pomocą jednej operacji przesunięcia, a następnie operacji badania bitu (bitu znaku). W rezultacie sprawdzenie, czy jest spełnione

$$x\ \mathbf{in}\ [c_1, c_2, \ldots, c_n]$$

można zrealizować znacznie oszczędniej, aniżeli obliczając wartość przytoczonego poniżej wyrażenia boolowskiego

$$(x = c_1) \vee (x = c_2) \vee \ldots \vee (x = c_n)$$

Z powyższych rozważań wynika wniosek, że strukturę zbioru powinno się stosować jedynie dla *niewielkich typów podstawowych*. Górna granica mocy typu podstawowego, gwarantująca stosunkowo oszczędną realizację tej struktury, zależy od wielkości słowa w dostępnej maszynie; tak więc komputery o dużej długości słowa są pod tym względem wygodniejsze. Jeżeli długość słowa jest niewielka, można przyjąć reprezentację, w której stosuje się kilka słów dla jednego zbioru.

1.11. Plik sekwencyjny

Omawiane dotychczas struktury danych, a mianowicie tablice, rekordy i zbiory – charakteryzuje wspólna cecha, jaką jest skończoność mocy (zakładając, że moce typów składowych są skończone). Reprezentacja takich danych w pamięci maszyny cyfrowej nie sprawia specjalnych kłopotów.

Większość tzw. struktur złożonych – ciągi, drzewa, grafy itd. – charakteryzuje moc nieskończona. Fakt ten ma głębokie znaczenie i pociąga za sobą ważne konsekwencje praktyczne. Jako przykład rozważmy następującą strukturę *ciągu*.

Ciąg o typie podstawowym T_0 jest to bądź ciąg pusty, bądź konkatenacja ciągu (o typie podstawowym T_0) z wartością typu T_0.

Typ o strukturze ciągu T składa się więc z nieskończonej liczby wartości. Każda wartość zawiera skończoną liczbę składowych typu T_0, lecz liczba ta jest nieograniczona, tzn. dla każdego takiego ciągu można skonstruować ciąg od niego dłuższy.

Podobne rozważania stosuje się do wszystkich innych złożonych struktur danych. Pierwszym wnioskiem wynikającym z tych rozważań jest fakt, że wielkość pamięci niezbędna do pomieszczenia wartości złożonego typu strukturalnego nie jest znana podczas translacji; w rzeczywistości może się ona zmieniać w trakcie wykonania programu. Powstaje kwestia zorganizowania pewnego schematu **dynamicznego przydzielania pamięci**, w którym pamięć jest pobierana wtedy, gdy odpowiednie wartości wzrastają, i zwalniana do innych potrzeb – gdy maleją. Oczywiste jest więc, że problem rozmieszczenia struktur złożonych jest delikatny i trudny, a jego rozwiązanie będzie miało ogromny wpływ na efektywność wykorzystania pamięci. Wyboru odpowiedniego rozwiązania tego zagadnienia należy dokonać na podstawie znajomości podstawowych operacji stosowanych na strukturze oraz częstości ich wykonywania. Ponieważ żadnej z takich informacji nie zna twórca translatora języka, rozsądne jest zrezygnowanie z wprowadzenia w języku (do ogólnego przeznaczenia) struktur złożonych. Podobnie programista powinien unikać ich używania, jeśli dane zadanie można rozwiązać przy zastosowaniu jedynie struktur podstawowych.

W większości języków omija się problem konieczności opracowania mechanizmu tworzenia struktur złożonych (przy braku informacji o ich potencjalnym zastosowaniu), ponieważ wszystkie takie struktury składają się bądź z elementów nieustrukturowanych, bądź ze struktur podstawowych. Dowolne struktury mogą więc być generowane jawnie za pomocą operacji zdefiniowanych przez programistę, jeśli tylko istnieją możliwości dynamicznego przydzielania pamięci dla elementów tych struktur, ich dynamicznego łączenia oraz odwoływania się do tych elementów. Metody generowania i przetwarzania takich struktur są omówione w rozdz. 4.

Istnieje jednak pewna struktura, złożona w tym sensie, że jej moc jest nieskończona, lecz używana tak szeroko i tak często, że konieczne wydaje się włączenie jej do zbioru struktur podstawowych. Strukturą taką jest **ciąg** (ang. *sequence*). W celu przedstawienia abstrakcyjnego pojęcia ciągu wprowadzimy następującą terminologię i notację:

(1) $\langle\ \rangle$ oznacza ciąg pusty.
(2) $\langle x_0 \rangle$ oznacza ciąg składający się z jednej składowej x_0 (ciąg jednoelementowy).
(3) Jeśli $x = \langle x_1, ..., x_n \rangle$, $y = \langle y_1, ..., y_n \rangle$ są ciągami, to
$$x \mathbin{\&} y = \langle x_1, ..., x_m, y_1, ..., y_n \rangle \tag{1.41}$$
jest **konkatenacją** ciągów x i y.

(4) Jeśli $x = \langle x_1, ..., x_n \rangle$ jest ciągiem niepustym, to $pierwszy(x) = x_1$ (1.42)
oznacza pierwszy element ciągu x.

(5) Jeśli $x = \langle x_1, ..., x_n \rangle$ jest ciągiem niepustym, to
$reszta(x) = \langle x_2, ..., x_n \rangle$ (1.43)
jest ciągiem x bez jego pierwszej składowej. W rezultacie dostajemy tożsamość
$$\langle pierwszy(x) \rangle \,\&\, reszta(x) \equiv x \tag{1.44}$$

Wprowadzenie powyższej notacji nie oznacza, że będzie ona stosowana w programach działających w rzeczywistych komputerach. W gruncie rzeczy jest pożądane, aby operacja konkatenacji *nie* była stosowana w całej swej ogólności oraz aby posługiwanie się strukturą ciągu ograniczało się do stosowania pewnego, starannie wybranego zbioru operatorów, dobranych do pewnej dziedziny zastosowań, aczkolwiek zdefiniowanego na podstawie abstrakcyjnych pojęć ciągu i konkatenacji. Staranny wybór owego zbioru operatorów umożliwia opracowanie odpowiedniej, efektywnej metody rozmieszczenia ciągów w pamięci; mechanizm dynamicznego przydziału pamięci będzie wtedy wystarczająco prosty, aby programista nie musiał troszczyć się o jego szczegóły.

Aby zaznaczyć, że ciąg wprowadzony jako podstawowa struktura danych pozwala na stosowanie jedynie ograniczonego zbioru operatorów, pozwalającego w istocie tylko na ściśle sekwencyjny dostęp do składowych, strukturę tę zwie się **plikiem sekwencyjnym** (ang. *sequential file*) lub krótko **plikiem** (ang. *file*). Analogicznie do definicji tablicy i zbioru, typ plikowy jest zdefiniowany formułą

---

type T = **file of** T_0 (1.45)

---

wyrażającą, że każdy plik typu T składa się z 0 lub więcej składowych typu T_0.

PRZYKŁADY
type *text* = **file of** *char*
type *pakiet* = **file of** *card*
□

Istota **dostępu sekwencyjnego** polega na tym, że w każdej chwili bezpośrednio dostępna jest tylko jedna konkretna składowa ciągu. Składowa ta jest określona przez **bieżącą (aktualną) pozycję** (ang. *current position*) mechanizmu dostępu. Operatory plikowe mogą zmienić składową na następną składową bądź na pierwszą składową całego pliku. Formalnie przedstawiamy pozycję pliku, zakładając, że plik składa się z dwóch części: części x_L – na lewo od pozycji bieżącej oraz części x_P – na prawo od niej. Oczywiste jest, że równość

$$x \equiv x_L \,\&\, x_P \tag{1.46}$$

wyraża relację niezmienniczą.

Następną, niezwykle ważną konsekwencją stosowania sekwencyjnej metody dostępu jest to, że procesy konstruowania i przeglądania ciągu są istotnie odmienne; nie mogą zatem występować na przemian w dowolnej kolejności. Jak widać, plik konstruuje się, dołączając kolejne składowe (na końcu), następnie zaś można go analizować za pomocą sekwencyjnego przeglądania. Plik będzie więc znajdował się w jednym z dwóch **stanów**: w stanie konstruowania (pisania) bądź w stanie przeglądania (czytania).

Zaleta stosowania ściśle sekwencyjnej metody dostępu jest szczególnie wyraźna wtedy, gdy pliki mają być rozmieszczone w tzw. wtórnych ośrodkach przechowywania danych, tzn. jeśli występują przesłania między różnymi ośrodkami. Metoda sekwencyjnego dostępu jest jedyną, która pozwala ukryć przed programistą zawiłości mechanizmu stosowanego przy takich przesłaniach. W szczególności dozwala ona korzystać z prostych metod **buforowania** gwarantujących optymalne użytkowanie zasobów złożonego systemu komputerowego.

Wśród wielu rodzajów ośrodków przechowywania danych istnieją takie, dla których metoda dostępu sekwencyjnego jest w gruncie rzeczy jedyną możliwą. Z pewnością należą do nich wszystkie rodzaje taśm. Jednakże nawet na bębnach magnetycznych i dyskach każda ścieżka zapisu jest ośrodkiem zapewniającym jedynie dostęp sekwencyjny. Ściśle sekwencyjna metoda dostępu jest charakterystyczna dla wszystkich urządzeń o napędzie mechanicznym, a także dla wielu innych.

1.11.1. Elementarne operatory plikowe

Przystąpimy teraz do sformułowania abstrakcyjnego pojęcia dostępu sekwencyjnego za pośrednictwem zbioru konkretnych **elementarnych operatorów plikowych**, które może stosować programista. Są one zdefiniowane za pomocą pojęć ciągu i konkatenacji. Są to operatory: inicjowania procesu generowania pliku, inicjowania procesu przeglądania, dołączania składowej na końcu ciągu oraz przejścia do przeglądania następnej składowej ciągu. Dla dwóch ostatnich operatorów przyjmuje się, że implicite dotyczą pewnej pomocniczej zmiennej reprezentującej bufor. Zakładamy, że bufor taki jest automatycznie związany z każdą zmienną plikową x i będziemy go oznaczać przez $x\uparrow$. Oczywiście, jeśli x jest typu T, to $x\uparrow$ jest typu podstawowego T_0.

(1) Konstruowanie ciągu pustego. Operacja

$$rewrite(x) \tag{1.47}$$

zastępuje przypisanie

$$x := \langle\ \rangle$$

Operacja ta używana jest do powtórnego zapisania bieżącego x i zapoczątkowania procesu konstruowania nowego ciągu; operacji tej odpowiada przewinięcie taśmy.

(2) Wydłużenie ciągu. Operacja

$$put(x) \tag{1.48}$$

zastępuje przypisanie

$$x := x \mathbin{\&} \langle x\uparrow \rangle$$

którego efektem jest dołączenie wartości $x\uparrow$ na końcu ciągu x.

(3) Zapoczątkowanie przeglądania. Operacja

$$reset(x) \tag{1.49}$$

odpowiada ciągowi przypisań

$$x_L := \langle\ \rangle$$
$$x_P := x$$
$$x\uparrow := pierwszy(x)$$

Operację tę stosuje się w celu zapoczątkowania procesu czytania ciągu.

(4) Przejście do następnej składowej. Operacja

$$get(x) \tag{1.50}$$

zastępuje ciąg przypisań

$$x_L := x_L \mathbin{\&} \langle pierwszy\ (x_P)\rangle$$
$$x_P := reszta(x_P)$$
$$x\uparrow := pierwszy(reszta(x_P))$$

Należy pamiętać, że funkcja $pierwszy(s)$ jest zdefiniowana tylko wtedy, gdy $s \neq \langle\ \rangle$.

Operatory *rewrite* i *reset* nie zależą od pozycji pliku przed ich wykonaniem. W każdym przypadku wynikiem działania każdego z tych operatorów jest przesunięcie do początku pliku.

Podczas przeglądania ciągu trzeba mieć możliwość rozpoznania końca ciągu, w przeciwnym bowiem razie przypisanie

$$x\uparrow := pierwszy(x_P)$$

reprezentuje operację niezdefiniowaną. Osiągnięcie końca pliku jest równoznaczne ze spełnieniem warunku, że część prawa x_P ciągu jest pusta. Wprowadzimy więc predykat

$$eof(x) \equiv x_P = \langle\ \rangle \tag{1.51}$$

oznaczający, że osiągnięto koniec pliku. Warunkiem wykonania operacji $get(x)$ jest fałszywość predykatu $eof(x)$.

Wszystkie operacje na plikach można w zasadzie zdefiniować za pomocą tych czterech podstawowych operatorów plikowych. W praktyce jednakże często

łączy się operację przejścia do następnej pozycji pliku (*get* lub *put*) z dostępem do bufora. Podamy więc dwie dalsze procedury; dają się one wyrazić za pomocą operatorów podstawowych. Niech *v* będzie zmienną, *e* zaś wyrażeniem o typie równym typowi T_0 składowej pliku. Wówczas

read(*x*, *v*) będzie równoznaczne z
$v := x\uparrow$; *get*(*x*)

podczas gdy

write(*x*, *e*) będzie równoznaczne z
$x\uparrow := e$; *put*(*x*)

Zaleta wynikająca ze stosowania *read* i *write* zamiast *get* i *put* polega nie tylko na skróceniu zapisu, lecz również na prostocie koncepcyjnej. Możliwe jest bowiem teraz ignorowanie istnienia zmiennej buforowej $x\uparrow$, której wartość jest czasami niezdefiniowana. Zmienna buforowa może być jednak przydatna do celów „przeglądania z wyprzedzeniem".

Wstępnymi warunkami wykonania obu tych procedur są:

$\neg$ *eof*(*x*) dla *read*(*x*, *v*)
eof(*x*) dla *write*(*x*, *e*)

Podczas czytania predykat *eof*(*x*) przyjmuje wartość *true* natychmiast po przeczytaniu ostatniego elementu pliku *x*. Przytoczone poniżej dwa schematy programów dla sekwencyjnej konstrukcji i przetwarzania pliku *x* ilustrują powyższe rozważania. Zdania *R* i *S* oraz predykat *p* stanowią dodatkowe parametry schematów.

Zapisywanie do pliku *x*:

```
rewrite(x);                                        (1.52)
while p do
    begin R(v); write(x, v)
    end
```

Odczytywanie z pliku *x*:

```
reset(x);                                          (1.53)
while ¬ eof(x) do
    begin read(x, v); S(v)
    end
```

1.11.2. Pliki z podstrukturami

W większości zastosowań duże pliki zawierają pewne podstruktury wewnętrzne. Na przykład, chociaż książkę można traktować jako jeden ciąg znaków, dzieli się ją na rozdziały lub części. Celem takiego podziału jest stworzenie pewnych punktów orientacyjnych, pewnych współrzędnych, umożliwiających orientację w długim ciągu informacji. Istniejące ośrodki przechowywania danych często dają pewne możliwości używania takich punktów orientacyjnych (np. znaczników na taśmach) i możliwości znajdowania ich z większą szybkością niż przy przeglądaniu sekwencyjnym wszystkich informacji między tymi punktami.

W naszym podejściu naturalnym sposobem wprowadzenia podstruktur pierwszego poziomu jest traktowanie takiego pliku jako ciągu składowych, które są również ciągami, czyli jako pliku plików. Zakładając, że najniższe (w sensie poziomu) składowe są typu U, podstruktury są typu

T' = **file of** U

a cały plik jest typu

T = **file of** T'

Jest oczywiste, że można w ten sposób skonstruować plik o dowolnie głębokiej hierarchii podstruktur. Typ T_n można ogólnie zdefiniować rekurencyjnie:

$T_0 = U$
T_i = **file of** T_{i-1} $\qquad i = 1, \ldots, n$

Tego rodzaju pliki często nazywa się **plikami wielopoziomowymi**, a składową typu T_i – **segmentem** i-tego poziomu. Przykład pliku wielopoziomowego stanowi książka, w której kolejnym poziomom segmentacji odpowiadają rozdziały, podrozdziały, punkty i wiersze. Jednakże najczęściej spotykany jest przypadek pliku z tylko jednym poziomem segmentacji.

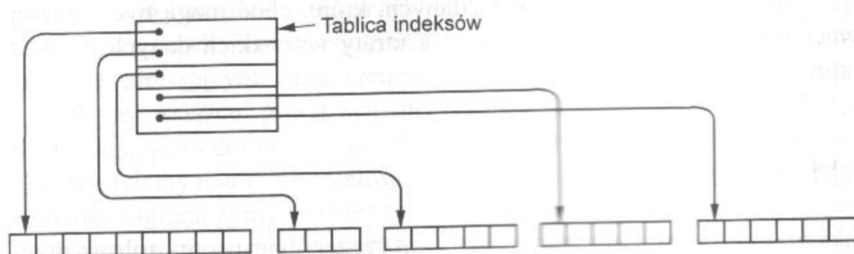
Plik jednopoziomowy w żadnym sensie nie jest identyczny z tablicą plików. Przede wszystkim liczba segmentów jest zmienna i plik można rozszerzać tylko na jego końcu. Stosując wprowadzoną notację i przyjmując następującą definicję pliku:

x: **file of file of** U

powiemy, że $x\uparrow$ oznacza aktualnie dostępny segment, $x\uparrow\uparrow$ zaś aktualnie dostępną składową jednostkową. Odpowiednio *put*($x\uparrow$) i *get*($x\uparrow$) odwołują się do składowej jednostkowej, a *put*(x) i *get*(x) oznaczają operacje dołączenia segmentu i przejścia do następnego segmentu.

Pliki segmentowane można łatwo rozmieszczać we wszystkich sekwencyjnych ośrodkach przechowywania danych, w tym również na taśmach. Segmenta-

cja nie zmieniła pierwotnych cech pliku pozwalających jedynie na dostęp sekwencyjny zarówno do pojedynczych składowych, jak i – prawdopodobnie znacznie szybciej – do segmentów. Inne ośrodki przechowywania danych, np. bębny i dyski, składają się zazwyczaj z pewnej liczby ścieżek, stanowiących ośrodki sekwencyjne, jednak na ogół zbyt małych na to, aby pomieścić cały plik. Pliki na dyskach są w konsekwencji rozrzucone na kilku ścieżkach i zawierają dodatkowe informacje sterujące łączące ścieżki w jeden ciąg. Punkt startowy każdej ścieżki zawiera naturalny łatwo dostępny znacznik segmentu. Dostęp do takiego znacznika jest nawet bardziej bezpośredni niż do znaczników w ośrodkach w pełni sekwencyjnych. W pamięci wewnętrznej można np. przechowywać wektor adresów początków segmentów oraz faktyczne długości segmentów (zob. rys. 1.10).



RYSUNEK 1.10
Plik indeksowany o pięciu segmentach

Opisana powyżej struktura nosi nazwę **pliku indeksowanego** (zwanego również **plikiem o dostępie bezpośrednim**). Organizacja bębnów i dysków dopuszcza zwykle wiele fizycznych znaczników na ścieżce, od których można zacząć czytanie lub pisanie. W związku z tym nie jest konieczne, aby każdy segment zajmował pełną ścieżkę, co prowadziłoby do nieefektywnego wykorzystania pamięci w przypadku, gdy segmenty będą krótsze niż długość ścieżki. Obszar pamięci między dwoma kolejnymi fizycznymi znacznikami zwany jest **segmentem fizycznym** (lub **sektorem**) w odróżnieniu od **segmentu logicznego**, będącego znaczącą jednostką struktur danych programu. W każdym segmencie fizycznym znajduje się co najwyżej jeden segment logiczny, a każdy segment logiczny zajmuje co najmniej jeden segment fizyczny (nawet jeśli jest pusty). Należy pamiętać, że pomimo nazywania takich plików plikami o „dostępie bezpośrednim" średni czas potrzebny na umiejscowienie segmentu (tzw. **czas oczekiwania**) równy jest połowie czasu pełnego obrotu dysku.

Operacja pisania dla plików indeksowanych jest również wykonywana sekwencyjnie na końcu pliku. W związku z tym pliki indeksowane są szczególnie użyteczne w tych przypadkach, w których względnie rzadko są modyfikowane. Zmian dokonuje się, rozszerzając plik bądź przepisując i jednocześnie aktualizując cały plik. Przeglądanie takiego pliku może się odbywać dużo bardziej selektywnie i szybciej dzięki zastosowaniu indeksowania. Jest to typowa sytuacja dla tzw. **banków danych**.

Systemy pozwalające na selektywne zmienianie fragmentów ze środka pliku są zwykle skomplikowane, a posługiwanie się nimi ryzykowne, ponieważ nowe porcje informacji muszą być tej samej wielkości co – zastępowane nimi – stare porcje. Ponadto w zastosowaniach wymagających dużych ilości danych stosowanie aktualizacji selektywnej nie jest wskazane, gdyż w przypadku jakiegokolwiek błędu – spowodowanego niepoprawnym programem czy też awarią sprzętu – dane powinny być w takim stanie, aby można było powrócić i powtórzyć nie wykonany proces. W związku z powyższym aktualizacja jest zwykle zorganizowana w ten sposób, że stary plik zastępuje się nowym, zaktualizowanym, dopiero po uprzednim sprawdzeniu, czy nowy plik jest poprawny. Z uwagi na potrzebę aktualizacji organizacja sekwencyjna jest najlepsza z punktu widzenia niezawodności. Taka struktura powinna być preferowana w stosunku do bardziej wyszukanych organizacji dużych zbiorów danych, które, choć mogą być bardziej efektywne, jednak wiążą się z możliwością utraty wszystkich danych w razie awarii sprzętu.

1.11.3. Teksty

Pliki o składowych typu *char* odgrywają szczególnie istotną rolę w przetwarzaniu danych: stanowią pomost między sprzętem liczącym a jego użytkownikami. Ciągi znaków stanowią zarówno **czytelny ciąg wejściowy** zadany przez programistę, jak i **czytelny ciąg wyjściowy** reprezentujący wyniki obliczone przez komputer. Tego rodzaju danym należy więc nadać standardową nazwę:

type *text* = **file of** *char*

Komunikację między procesem obliczeniowym a jego twórcą można ostatecznie reprezentować jako dwa pliki tekstowe. Jeden z nich stanowi **dane wejściowe** (ang. *input*) do procesu obliczeniowego, drugi zaś **dane wyjściowe** (ang. *output*) – obliczone wyniki. Będziemy przeto zakładać istnienie we wszystkich programach tych dwóch plików, zadeklarowanych jako

var *input*, *output*: *text*

Zgodnie z założeniem, że pliki te reprezentują standardowe ośrodki wejścia i wyjścia systemu komputerowego (takie jak czytnik kart i drukarka), przyjmiemy, że plik wejściowy (*input*) może być jedynie wczytywany, a plik wyjściowy (*output*) – wypisywany.

Ponieważ w większości przypadków używa się wspomnianych plików standardowych, założymy, że jeśli pierwszy parametr procedur *read* i *write* nie jest zmienną plikową, to domyślnie przyjmuje się, że jest nią odpowiednio *input* bądź *output*. Ponadto wprowadzimy konwencję, że obie te standardowe procedury można zapisywać z dowolną liczbą parametrów. Sumę wymienionych ustaleń notacyjnych reprezentują poniższe związki:

read($x1$, ..., xn) zastępuje *read*(*input*, $x1$, ..., xn)
write($x1$, ..., xn) zastępuje *write*(*output*, $x1$, ..., xn)
read(f, $x1$, ..., xn) zastępuje
  begin *read*(f, $x1$); ... ; *read*(f, xn) **end**
write(f, $x1$, ..., xn) zastępuje
  begin *write*(f, $x1$); ... ; *write*(f, xn) **end**

Teksty są typowymi przykładami ciągów mających podstruktury. Jednostkami podstruktury są zwykle rozdziały, podrozdziały i wiersze. Często stosowaną metodą reprezentowania tych podstruktur w plikach tekstowych są specjalne znaki rozdzielające. Najbardziej znanym przykładem takiego znaku jest odstęp (spacja); podobne symbole można stosować do zaznaczenia końca wiersza, podrozdziału i rozdziału. Na przykład szeroko stosowany zbiór znaków ISO (a także jego amerykańska wersja ASCII) zawiera wiele takich elementów, zwanych **znakami sterującymi** (ang. *control characters*); zob. dodatek A.

W tej książce powstrzymamy się od stosowania specjalnych znaków rozdzielających i definitywnego określenia metody reprezentowania podstruktur. Tekst będziemy natomiast traktowali jako ciąg wierszy (każdy wiersz jest ciągiem znaków). Ograniczymy również rozważania do jednego tylko poziomu podstruktur, a mianowicie *wierszy*. Jednakże zamiast traktowania tekstu jako pliku plików znaków drukarki będziemy go traktować po prostu jako plik znaków, ale wprowadzimy dodatkowe operatory i predykaty służące do oznaczania i rozpoznawania wierszy. Ich działanie można najłatwiej zrozumieć, jeśli założy się, że wiersze są oddzielane (hipotetycznymi) separatorami (nie należącymi do typu *char*), zadaniem zaś owych procedur i predykatów będzie umieszczanie i rozpoznawanie tych separatorów. Dodatkowe operatory są następujące:

writeln(f) Dołącz znacznik końca wiersza do pliku f.
readln(f) Przeskocz ciąg znaków pliku f aż do znaku występującego bezpośrednio po najbliższym znaczniku wiersza.
eoln(f) Funkcja boolowska; ma wartość *true*, jeśli aktualna pozycja pliku wskazuje na seperator wiersza, w przeciwnym razie – *false*. (Będziemy uważali, że jeśli *eoln*(f) = *true*, to $f\uparrow$ = *spacja*).

Obecnie sformułujemy schematy dwóch programów pisania i czytania tekstów, analogicznie do schematów „pisania" i „czytania" dowolnych plików [zob. (1.52) i (1.53)] W schematach tych zakłada się istnienie pliku tekstowego f i przywiązuje należytą uwagę do tworzenia i rozpoznawania struktury wierszowej. Niech $R(x)$ oznacza instrukcję przypisania wartości zmiennej x (typu *char*), która jednocześnie określa warunki p i q oznaczające, odpowiednio, „jest to ostatni znak wiersza" i „jest to ostatni znak pliku". Niech U będzie instrukcją wykonywaną przy wczytywaniu początku każdego wiersza, $S(x)$ instrukcją wykonywaną dla każdego znaku x pliku f, V zaś instrukcją wykonywaną przy końcu każdego wiersza.

Pisanie tekstu *f*:

```
rewrite(f);
while ¬ q do
   begin
      while ¬ p do
         begin R(x); write(f, x)
         end;
      writeln(f)
   end                                              (1.54)
```

Czytanie tekstu *f*:

```
reset(f);
while ¬ eof(f) do
   begin U;
      while ¬ eoln(f) do
         begin read(f, x); S(x)
         end;
      V; readln(f)
   end                                              (1.55)
```

W pewnych przypadkach struktura wierszowa nie niesie żadnej szczególnie istotnej informacji. Nasze założenie o wartości zmiennej buforowej przy napotkaniu znacznika wiersza [zob. definicję funkcji *eoln*(*f*)] umożliwia zastosowanie w tych przypadkach prostego schematu programu. Należy pamiętać, że zgodnie z definicją *eoln* koniec wiersza jest reprezentowany przez dodatkowy odstęp.

```
while ¬ eof(f) do                                   (1.56)
   begin read(f, x); S(x)
   end
```

W większości języków programowania dopuszcza się używanie argumentów typu *integer* lub *real* w procedurach czytania i pisania. Rozszerzenie to staje się oczywiste, jeśli wartości typów *integer* i *real* zostaną zdefiniowane jako tablice znaków o elementach oznaczających poszczególne cyfry liczb. Tego rodzaju definicje występują zwłaszcza w językach ściśle przeznaczonych do zastosowań ekonomicznych. W językach tych jest wymagana dziesiętna reprezentacja wewnętrzna liczb. Ważną zaletą wprowadzenia typów *integer* i *real* jako podstawowych jest to, że szczegółowe określenia reprezentacji mogą być opuszczone, a system liczący może stosować różne (być może bardziej przydatne dla swych celów) reprezentacje liczb. W rzeczywistości w systemach przeznaczonych do obliczeń naukowych wybiera się często dwójkową reprezentację liczb, z wielu względów lepszą od reprezentacji dziesiętnej.

Przy takim rozwiązaniu programista nie może zakładać, że liczby będą czytane lub wypisywane w plikach tekstowych bez odpowiednich operacji konwersji. Zazwyczaj operacje konwersji są ukryte w instrukcjach czytania i pisania z argumentami o typach liczbowych. Doświadczony programista jest jednak świadom faktu, że takie instrukcje (zwane **instrukcjami wejścia-wyjścia**) realizują dwa różne zadania: przekazanie danych między dwoma ośrodkami przechowywania danych oraz transformację reprezentacji danych. Druga z tych funkcji może być stosunkowo złożona i czasochłonna.

W dalszej treści tej książki instrukcje czytania i pisania o argumentach liczbowych będą stosowane zgodnie z regułami obowiązującymi w języku Pascal. Reguły te pozwalają na pewien rodzaj definicji formatu dla sterowania procesu przesyłania. Definicja formatu określa liczbę żądanych cyfr w przypadku instrukcji pisania. Owa liczba znaków, zwana też „wielkością pola", jest podana bezpośrednio za argumentem, mianowicie:

write(*f*, *x* : *n*)

Argument *x* ma być zapisany w pliku *f*; jego wartość jest przekształcana na ciąg (co najmniej) *n* znaków: jeżeli trzeba, cyfry są poprzedzone znakiem oraz odpowiednią liczbą spacji.

Dalsze szczegóły nie są potrzebne do zrozumienia przykładów programów zamieszczonych w tej książce. Jednakże przedstawimy przykłady dwóch procedur (programy 1.3 i 1.4) konwersji reprezentacji liczbowych, w celu zobrazowania kosztownej **złożoności** takich operacji, wbudowanych zazwyczaj w standardowe instrukcje pisania. Procedury te opisują konwersję liczb rzeczywistych z postaci dziesiętnej na wybraną reprezentację wewnętrzną i na odwrót. (Stałe w nagłówku są określone zgodnie z właściwościami zmiennopozycyjnego wzorca liczb dla maszyny cyfrowej CDC 6000: 11-bitowy wykładnik dwójkowy i 48-bitowa mantysa. Funkcja *expo*(*x*) oznacza wykładnik liczby *x*).

PROGRAM 1.3*
Wczytaj liczbę rzeczywistą

```
procedure readreal(var f: text; var x: real);
    {czyta liczbę rzeczywistą x z pliku f}
    {poniższe stałe są zależne od systemu}
    const t48 = 281474976710656; { = 2**48}
        granica = 56294995342131; { = t48 div 5}
        z = 27; { = ord('0')}
        lim1 = 322; {największy wykładnik}
        lim2 = −292; {najmniejszy wykładnik}
```

* Występująca w programach 1.3 i 1.4 funkcja boolowska *odd* przyjmuje wartość *true* dla nieparzystych wartości swojego argumentu, a wartość *false* w przeciwnym przypadku. – *Przyp. tłum.*

```
type całkdod = 0..323;
var zn: char; y: real; a, i, e: integer;
    s, ss: boolean; {znaki liczby}
function dzies(e: całkdod): real; { = 10**e, 0 < e < 322}
    var i: integer; t: real;
begin i := 0; t := 1.0;
    repeat if odd(e) then
      case i of
        0: t := t * 1.0E1;
        1: t := t * 1.0E2;
        2: t := t * 1.0E4;
        3: t := t * 1.0E8;
        4: t := t * 1.0E16;
        5: t := t * 1.0E32;
        6: t := t * 1.0E64;
        7: t := t * 1.0E128;
        8: t := t * 1.0E256
      end;
      e := e div 2; i := i + 1
    until e = 0;
    dzies := t
end;
begin
  {pomiń spacje na początku}
  while f↑ = ' ' do get(f);
  zn := f↑;
  if zn := '−' then
     begin s := true; get(f); zn := f↑
     end else
     begin s := false;
        if zn = '+' then
        begin get(f); zn := f↑
        end
     end;
  if ¬ (zn in ['0'..'9']) then
  begin sygnał('POWINNA BYĆ CYFRA'); stop;
  end;
  a := 0; e := 0;
  repeat if a < granica then a := 10 * a + ord(zn) − z else e := e + 1;
     get(f); zn := f↑
  until ¬ (zn in ['0'..'9']);
  if zn = '.' then
  begin {wczytaj część ułamkową} get(f); zn := f↑;
     while zn in ['0'..'9'] do
```

```
            begin if a < granica then
                begin a := 10 * a + ord(zn) - z; e := e - 1
                end;
                get(f); zn := f↑
            end;
        end;
        if zn = 'E' then
        begin {wczytaj czynnik skalujący} get(f); zn := f↑;
            i := 0;
            if zn = '-' then
                begin ss := true;
                    get(f); zn := f↑
                end else
            begin ss := false; if zn = '+' then
                begin get(f); zn := f↑
                end
            end;
            while zn in ['0'..'9'] do
            begin if i < granica then begin i := 10 * i + ord(zn) - z end;
                get(f); zn := f↑
            end;
            if ss then e := e - i else e := e + i
        end;
        if e < lim2 then
            begin a := 0; e := 0
            end else
        if e > lim1 then
        begin sygnał('ZA DUŻA LICZBA'); stop end;
        {0<a<2**49}
        if a ≥ t48 then y := ((a+1) div 2) * 2.0 else y := a;
        if s then y := -y;
        if e<0 then x := y/dzies(-e) else
        if e ≠ 0 then x := y * dzies(e) else x := y;
        while (f↑ = ' ') ∧ (¬ eof(f)) do get(f);
end {readreal}
```

PROGRAM 1.4

Wypisz liczbę rzeczywistą

```
procedure writereal(var f: text; x: real; n: integer);
    {wypisz n-znakową liczbę rzeczywistą x wg dziesiętnego wzorca zmienno-
    pozycyjnego}
    {poniższe stałe zależą od przyjętej reprezentacji zmiennopozycyjnej liczb
    rzeczywistych}
```

```
const t48 = 281474976710656;      {= 2 ** 48; 48 = wielkość mantysy}
      z = 27;    {ord('0')}
type całkdod = 0..323;    {dopuszczalny zakres wykładnika dziesiętnego}
var c, d, e, e0, e1, e2, i: integer;
function dzies(e : całkdod): real;      {10 ** e, 0 < e < 322}
   var i: integer; t: real;
begin i := 0; t := 1.0;
   repeat if odd(e) then
      case i of
         0: t := t * 1.0E1;
         1: t := t * 1.0E2;
         2: t := t * 1.0E4;
         3: t := t * 1.0E8;
         4: t := t * 1.0E16;
         5: t := t * 1.0E32;
         6: t := t * 1.0E64;
         7: t := t * 1.0E128;
         8: t := t * 1.0E256
      end;
      e := e div 2; i := i+1
   until e = 0;
   dzies := t
end {dzies};
begin {potrzeba przynajmniej 10 znaków: b+9.9E+999}
   if x=0 then
   begin repeat write(f, ' '); n := n-1
         until n ≤ 1;
      write(f, '0')
   end else
   begin
      if n ≤ 10 then n := 3 else n := n-7;
      repeat write(f, ' '); n := n-1
      until n ≤ 15;
      {1 < n ≤ 15, liczba drukowanych cyfr}
      begin {test znaku, następnie obliczenie wykładnika}
         if x<0 then
            begin write(f, '-'); x := -x
            end else write(f, ' ');
         e := expo(x);      {e=entier(log2(abs(x)))}
         if e ≥ 0 then
            begin e := e * 77 div 256+1; x := x/dzies(e);
               if x ≥ 1.0 then
                  begin x := x/10.0; e := e+1
                  end
```

```
            end else
            begin e := (e+1) * 77 div 256; x := dzies(-e) * x;
                if x<0.1 then
                    begin x := 10.0 * x; e := e-1
                    end
            end ;
        {0.1 ≤ x < 1.0}
        case n of {zaokraglanie}
              2: x := x+0.5E-2;
              3: x := x+0.5E-3;
              4: x := x+0.5E-4;
              5: x := x+0.5E-5;
              6: x := x+0.5E-6;
              7: x := x+0.5E-7;
              8: x := x+0.5E-8;
              9: x := x+0.5E-9;
             10: x := x+0.5E-10;
             11: x := x+0.5E-11;
             12: x := x+0.5E-12;
             13: x := x+0.5E-13;
             14: x := x+0.5E-14;
             15: x := x+0.5E-15;
        end ;
    if x ≥ 1.0 then
        begin x := x * 0.1 ; e := e+1 ;
        end;
    c := trunc(x, 48) ;   {= trunc(x * (2 ** 48))}
    c := 10 * c; d := c div t48;
    write(f, chr(d+z), '.');
    for i := 2 to n do
    begin c := (c-d * t48) * 10; d := c div t48;
        write(f, chr(d+z))
    end;
    write(f, 'E') ; e := e-1;
    if e<0 then
        begin write(f, '-'); e := -e
        end else write(f, '+');
    e1 := e * 205 div 2048; e2 := e-10 * e1;
    e0 := e1 * 205 div 2048; e1 := e1-10 * e0;
    write(f, chr(e0+z), chr(e1+z), chr(e2+z))
  end
 end
end {writereal}
```

1.11.4. Program redagowania pliku

Przedstawione poniżej zadanie jest przykładem zastosowania struktur sekwencyjnych. Przy okazji zaprezentowano metodę budowania i objaśniania programu i jego ewolucji. Metodę tę zwaną **stopniowym precyzowaniem** (ang. *stepwise refinement*) [1.4], [1.6] będziemy również stosować do objaśniania wielu algorytmów w dalszej treści książki.

Zadanie polega na zbudowaniu programu redagującego dany tekst x i dającego w wyniku tekst y. Redagowanie oznacza usuwanie pewnych wierszy z tekstu źródłowego, wstawianie nowych lub zastępowanie starych wierszy nowymi. Redagowaniem zarządza pewien ciąg **rozkazów redagujących** reprezentowany przez standardowy tekst *input*. Rozkazy te przyjmują następującą postać:

W, m Wstawienie tekstu po m-tym wierszu.
U, m, n Usunięcie wierszy od m-tego do n-tego.
Z, m, n Zamiana wierszy od m-tego do n-tego.
K Koniec procesu redagowania.

Każdy rozkaz jest wierszem standardowego pliku *input*, który będziemy nazywali plikiem rozkazów, m i n są numerami wierszy, a wstawiany tekst powinien występować bezpośrednio za rozkazem W bądź Z.

Rozkazy redagujące będą uszeregowane według wzrastających numerów wierszy. Zasada ta sugeruje **ściśle sekwencyjne przetwarzanie** tekstu wejściowego x. Oczywiste jest, że stan procesu jest określony bieżącą pozycją x, czyli numerem wiersza aktualnie analizowanego.

Załóżmy teraz, że z programu redagującego będzie się korzystać w trybie interakcyjnym i że plik rozkazów reprezentuje np. dane podawane z urządzenia końcowego. Przy takich założeniach jest wysoce pożądane, aby operator urządzenia końcowego otrzymywał pewne wtórne informacje zwrotne. Właściwą i użyteczną formą takiej informacji zwrotnej jest zawartość tego wiersza, do którego doszedł proces w wyniku wykonanego rozkazu. Wiersz ten będziemy dalej nazywali **wierszem bieżącym (aktualnym**; ang. *current line*). Ważną konsekwencją zasady, że po każdym rozkazie drukuje się wiersz bieżący, jest konieczność reprezentowania go przez bezpośrednio dostępną zmienną. Wiersz ów wpisywany jest do tej zmiennej po wczytaniu go z pliku x, ale przed wypisaniem do pliku y. Sposób ten nazywa się „czytaniem z wyprzedzeniem" (ang. *lookahead*). Program redagujący można teraz sformułować następująco:

```
program redagujący (x, y, input, output);
var nwb: integer; {numer wiersza bieżącego}
    wb: wiersz; {wiersz bieżący}
    x, y: text;
```

```
begin wczytaj rozkaz;                                          (1.57)
    repeat interpretuj rozkaz;
           wypisz wiersz;
           wczytaj rozkaz
    until rozkaz = 'K'
end.
```

Obecnie przystąpimy do dokładnego określenia poszczególnych zdań programu. Precyzując instrukcje *czytaj rozkaz* i *interpretuj rozkaz*, zauważmy, że rozkaz składa się zazwyczaj z trzech części: kodu rozkazu i dwóch parametrów. Wobec tego wprowadzimy trzy zmienne: *kod*, *m* oraz *n* zapewniające komunikację między odpowiednimi dwoma podprogramami realizującymi te instrukcje.

```
var kod, zn: char;
    m, n: integer
Czytaj rozkaz:
              read (kod, zn);
              if zn = ',' then read(m, zn) else m := nwb;      (1.58)
              if zn = ',' then read(n) else n := m;
```

Powyższe sformułowanie obejmuje przypadki rozkazów o 0, 1 bądź 2 parametrach, ponieważ na brakujące parametry podstawia się odpowiednie wartości domyślne.

```
Interpretuj rozkaz:
              przepisz;
              if kod = 'W' then
              begin schowajwiersz;
                    wstaw;
              end else
              if kod = 'U' then pomiń else                      (1.59)
              if kod = 'Z' then
              begin wstaw;
                    pomiń;
              end else
              if kod = 'K' then przepiszresztę else Błąd
```

W drugim kroku precyzowania opiszemy instrukcje *przepisz*, *wstaw* i *pomiń* użyte w (1.59), stosując operacje działające na pojedynczych wierszach, tzn. operacje *pobierzwiersz* i *schowajwiersz*. Wszystkie te instrukcje mają strukturę pętli. *Przepisz* ma za zadanie przepisanie ciągu wierszy z pliku x do pliku y, poczynając od wiersza bieżącego, a kończąc na m-tym. *Pomiń* czyta wiersze z pliku x aż do wiersza n-tego bez przepisywania ich do pliku y.

```
Przepisz:        while nwb<m do
                 begin schowajwiersz;
                       pobierzwiersz
                 end
Pomiń:           while nwb<n do pobierzwiersz;
Wstaw:           wczytajwiersz;
                 while niekoniec do                                (1.60)
                    begin schowajwiersz; wczytajwiersz
                    end;
                 pobierzwiersz;
Przepiszwiersz:  while ¬ eof(x) do
                    begin schowajwiersz; pobierzwiersz
                    end;
                 schowajwiersz;
```

W trzecim i ostatnim kroku precyzowania opiszemy operacje *pobierzwiersz*, *schowajwiersz*, *wczytajwiersz* i *wypiszwiersz*, korzystając z operacji działających na pojedynczych znakach. Wprowadzone do tej pory operacje działają na całych wierszach, przy czym nie czyniono żadnych założeń odnośnie do szczegółowej struktury pojedynczego wiersza. Wiemy, że wiersze są ciągami znaków. Wydaje się potrzebne do dalszych rozważań zadeklarowanie zmiennej *wb* (zawierającej wiersz bieżący) w postaci

var *wb*: **file of** *char*

Należy jednak pamiętać o zasadzie, że nie powinno się stosować struktury o nieskończonej mocy, jeśli wystarczy zastosować inną strukturę podstawową, np. tablicę. I rzeczywiście, w naszym przypadku struktura tablicy jest zupełnie wystarczająca, jeśli ograniczymy długość wiersza do, powiedzmy, 80 znaków. Określimy zatem

var *wb*: **array** [1..80] **of** *char*

Wszystkie cztery zdefiniowane poniżej podprogramy korzystają ze zmiennej i jako indeksu tablicy *wb*, która w rzeczywistości jest stosowana lokalnie i z powodzeniem mogłaby być zadeklarowana lokalnie w każdym podprogramie; ponadto należy wprowadzić zmienną globalną L, oznaczającą długość wiersza bieżącego.

```
Pobierzwiersz:  i := 0; nwb := nwb+1;
                while ¬ eoln(x) do
                   begin i := i+1; read(x, wb[i])
                   end;
                L := i; readln(x)
```

```
Schowajwiersz: i := 0;                                        (1.61)
               while i<L do
                   begin i := i+1; write(y, wb[i])
                   end;
               writeln(y)
Wczytajwiersz: i := 0;
               while ¬ eoln(input) do
                   begin i := i+1; read(wb[i])
                   end;
               readln
Wypiszwiersz:  i := 0; write(nwb);
               while i<L do
                   begin i := i+1; write(wb[i])
                   end;
               writeln
```

Warunek *niekoniec* w podprogramie *wstaw* łatwo teraz wyrazić jako

$L \neq 0$

Na tym zakończyliśmy opis konstruowania programu redagowania pliku.

Ćwiczenia

1.1

Załóżmy, że moce typów standardowych *integer*, *real* i *char* są odpowiednio równe c_I, c_R i c_C. Jakie są moce następujących typów danych zdefiniowanych w przykładach tego rozdziału: *płeć*, *Boolean*, *dzieńtygodnia*, *litera*, *cyfra*, *oficer*, *wiersz*, *alfa*, *zespolona*, *data*, *osoba*, *współrzędne*, *zbiórznaków*, *stantaśmy*?

1.2

Jak mógłbyś reprezentować zmienne typów wymienionych w ćwiczeniu 1.1:
(a) w pamięci Twojego komputera?
(b) w języku Fortran?
(c) w Twoim ulubionym języku programowania?

1.3

Jakie są ciągi instrukcji (Twojego komputera) realizujące:
(a) operacje zapisywania w pamięci i pobierania z pamięci elementów spakowanych rekordów i tablic?
(b) operacje na zbiorach wraz z testem należenia do zbioru?

1.4

Czy jest możliwe sprawdzenie podczas wykonania poprawności stosowania rekordów z wariantami? Czy jest to również możliwe podczas translacji?

1.5

Jakie są przyczyny definiowania niektórych zbiorów danych jako plików sekwencyjnych, a nie tablic?

1.6

Należy zrealizować pliki sekwencyjne zgodnie z definicją w p. 1.11, mając do dyspozycji komputer o wielkiej pamięci wewnętrznej. Możesz wprowadzić ograniczenie, że pliki nie przekroczą nigdy ustalonej długości L. W związku z tym możesz reprezentować pliki w postaci tablic.
Podaj sposób realizacji tych plików, zawierający wybraną reprezentację danych i procedury dla elementarnych operatorów plikowych *get*, *put*, *reset* i *rewrite*, korzystając z definicji aksjomatycznej w p. 1.11.

1.7

Rozwiąż ćwiczenie 1.6 dla przypadku plików segmentowanych.

1.8

Dany jest rozkład jazdy pociągów na wielu liniach kolejowych. Znajdź reprezentację danych (w postaci tablic, rekordów lub plików) pozwalającą na określenie czasów przyjazdu i odjazdu dla danej stacji i danego pociągu.

1.9

Dany jest tekst T w postaci pliku i dwie krótkie listy słów w postaci dwóch tablic A i B. Przyjmijmy, że słowa są tablicami znaków o niewielkiej, stałej długości maksymalnej. Napisz program przekształcający tekst T na tekst S przez zastąpienie każdego wystąpienia słowa A_i odpowiadającym mu słowem B_i.

1.10

Jakie zmiany – inne definicje stałych itp. – są konieczne w przypadku adaptacji programów 1.3 i 1.4 do Twojego komputera?

1.11

Napisz procedurę analogiczną do programu 1.4 o nagłówku

procedure *writereal*(**var** *f*: *text*; *x*: *real*; *n*, *m*: *integer*);

Zadaniem procedury jest transformacja wartości x na ciąg przynajmniej n znaków (zapisywanych w pliku f) reprezentujących wartość w dziesiętnej postaci stałopozycyjnej z m-cyfrową częścią ułamkową. Jeśli to konieczne, liczba ma być poprzedzona odpowiednią liczbą odstępów i (lub) znakiem.

1.12

Zapisz algorytm redagowania pliku z p. 1.11.4 w postaci pełnego programu.

1.13

Porównaj trzy następujące wersje przeszukiwania połówkowego z programem 1.17. Które z tych trzech programów są poprawne? Które są bardziej optymalne? Zakładamy poniższe deklaracje i stałą $N>0$:

```
var i, j, k: integer;
    a: array[1..N] of T;
    x: T
```

PROGRAM A:

```
i := 1; j := N;
repeat k := (i+j) div 2;
    if a[k]<x then i := k else j := k
until (a[k]=x) ∨ (i≥j)
```

PROGRAM B:

```
i := 1; j := N;
repeat k := (i+j) div 2;
if x≤a[k] then j := k-1;
    if a[k]≤x then i := k+1
until i>j
```

PROGRAM C:

```
i := 1; j := N;
repeat k := (i+j) div 2;
    if x<a[k] then j := k else i := k+1
until i≥j
```

Wskazówka: Po zakończeniu każdego programu musi być $a[k]=x$, jeśli taki element istnieje, lub $a[k] \neq x$, jeśli nie istnieje żaden element o wartości x.

1.14

Wytwórnia produkująca płyty i taśmy z nagraniami organizuje ankietę w celu określenia powodzenia swoich wyrobów i zorganizowania radiowej parady przebojów. Osoby biorące udział w ankiecie mają być podzielone na cztery grupy w zależności od płci i wieku (powiedzmy, do 20 lat włącznie i powyżej 20 lat). Każda osoba podaje pięć przebojów. Przeboje są oznaczone liczbami od 1 do N (powiedzmy, $N=30$). Wyniki ankiety są zapisane w pliku.

```
type przebój = 1..N;
    płeć = (mężczyzna, kobieta);
    odpowiedź =
        record nazwisko, imię: alfa;
                s: płeć;
                wiek: integer;
                wybór: array[1..5] of przebój
        end;
var ankieta: file of odpowiedź
```

Każdy element pliku reprezentuje zatem respondenta i zawiera jego *nazwisko*, *imię*, *płeć*, *wiek* i pięć wybranych przebojów zgodnie z priorytetem. Plik ten stanowi dane wejściowe programu, który ma utworzyć następujące wyniki:

(1) Listę przebojów w kolejności liczb zdobytych głosów. Każdy element listy zawiera numer przeboju oraz liczbę określającą, ile razy był wymieniony u poszczególnych respondentów. Na liście nie występują te przeboje, na które nie głosowano.

(2) Cztery oddzielne listy z nazwiskami i imionami wszystkich respondentów, którzy na pierwszym miejscu wymienili jeden z trzech najpopularniejszych przebojów w swojej kategorii. Każda z pięciu list ma być poprzedzona odpowiednim nagłówkiem.

Literatura

1.1. Dahl O.J., Dijkstra E.W., Hoare C.A.R.: *Structured Programming*. New York, Academic Press 1972, s. 155–165.

1.2. Hoare C.A.R.: Notes on Data Structuring. **In**: Dahl O.J., Dijkstra E.W., Hoare C.A.R. *Structured Programming*, s. 83–174.

1.3. Jensen K., Wirth N.: PASCAL, User Manual and Report. *Lecture Notes in Computer Science*. Vol. 18, Berlin, Springer-Verlag 1974.

1.4. Wirth N.: Program Development by Stepwise Refinement. *Comm. ACM*, **14**, No. 4, 1971, s. 221–227.

1.5. Wirth N.: The Programming Language PASCAL. *Acta Informatica*, **1**, No. 1, 1971, s. 35–63.

1.6. Wirth N.: On the Composition of Well-Structured Programs. *Computing Surveys*, **6**, No. 4, 1974, s. 247–259.

2 Sortowanie

2.1. Wprowadzenie

Podstawowyn celem tego rozdziału jest zilustrowanie za pomocą dużej liczby przykładów sposobów użycia wprowadzonych w poprzednim rozdziale struktur danych oraz pokazanie, jaki wpływ ma wybór struktury dla przetwarzanych danych na algorytmy wykonania określonego zadania. Sortowanie jest dobrym przykładem tego, że określone zadanie może być wykonane według wielu różnych algorytmów. Każdy z algorytmów ma pewne zalety i wady, które trzeba przeanalizować dla konkretnego zastosowania.

Sortowaniem (ang. *sorting*) nazywamy proces ustawiania zbioru obiektów w określonym **porządku**. Sortowanie stosuje się w celu ułatwienia późniejszego wyszukiwania elementów sortowanego zbioru. Tak zdefiniowane sortowanie jest w wielu dziedzinach podstawowym, niemal powszechnie spotykanym działaniem. Obiekty są posortowane na listach płac, w książkach telefonicznych, w spisach treści, w bibliotekach, słownikach, magazynach i niemal wszędzie tam, gdzie potrzebne jest przeszukiwanie i wyszukiwanie składowanych obiektów. Już małe dzieci uczy się kłaść swoje rzeczy „w porządku"; spotykają się one z pewnym rodzajem sortowania dużo wcześniej niż z arytmetyką.

Wynika stąd, że sortowanie jest istotnym i niezbędnym działaniem, zwłaszcza w przetwarzaniu danych. Cóż bowiem innego może być łatwiejsze do sortowania niż „dane"! Niemniej jednak zainteresujemy się głównie metodami jeszcze bardziej podstawowymi, stosowanymi w konstrukcji algorytmów. Niewiele istnieje metod nie mających żadnego związku z algorytmami sortowania. W szczególności sortowanie jest idealnym przykładem pozwalającym ukazać wielką różnorodność algorytmów rozwiązywania tego samego problemu, z których wiele jest w pewnym sensie optymalnych, a większość z nich ma pewne zalety w stosunku do innych. Jest to więc doskonały przykład wskazujący na konieczność dokonywania analizy algorytmów. Co więcej, sortowanie nadaje się dobrze do wykazania, jak znaczne korzyści można osiągnąć dzięki tworzeniu skomplikowanych algorytmów, chociaż proste i naturalne metody są łatwo dostępne.

Zależność wyboru algorytmu od struktury przetwarzanych danych – będąca zjawiskiem powszechnym – w przypadku sortowania jest tak głęboka, że metody sortowania podzielono ogólnie na dwie klasy, a mianowicie: na metody **sortowania tablic** i metody **sortowania plików** (sekwencyjnych). Te dwie klasy są często nazywane sortowaniem **wewnętrznym** i **zewnętrznym**, ponieważ tablice są przechowywane w szybkiej i o dostępie swobodnym, „wewnętrznej" pamięci komputerów, pliki zaś są zazwyczaj przechowywane w powolniejszych, lecz pojemniejszych „zewnętrznych" pamięciach w urządzeniach poruszanych mechanicznie (dyski, taśmy). Znaczenie tego rozróżnienia jest widoczne w przykładzie sortowania ponumerowanych kart. Nadanie kartom struktury tablicy odpowiada położeniu ich na stole przed sortującym tak, że każda karta z osobna jest widoczna i dostępna (zob. rys. 2.1).

RYSUNEK 2.1
Sortowanie tablicy

Nadanie kartom struktury pliku pociąga za sobą to, że widoczne są tylko karty leżące na wierzchu każdej sterty (zob. rys. 2.2). Tego rodzaju ograniczenie będzie oczywiście miało poważny wpływ na zastosowaną metodę sortowania, lecz jest nie do uniknięcia, jeżeli karty, które trzeba ułożyć, nie mieszczą się na powierzchni stołu.

Zanim przejdziemy do dalszych rozważań, wprowadzimy terminologię i oznaczenia, których będziemy używać w tym rozdziale. Dane są obiekty

$a_1, a_2, ..., a_n$

Sortowanie polega na permutowaniu tych obiektów aż do chwili osiągnięcia uporządkowania

$a_{k_1}, a_{k_2}, ..., a_{k_n}$

takiego, że dla zadanej **funkcji porządkującej** f zachodzi

$$f(a_{k_1}) \leqslant f(a_{k_2}) \leqslant ... \leqslant f(a_{k_n}) \qquad (2.1)$$

RYSUNEK 2.2
Sortowanie pliku

Na ogół nie oblicza się wartości funkcji porządkującej według określonego wzoru, ale przechowuje się je w jawnej postaci jako składowe (pola) każdego obiektu. Wartość tej funkcji nazywa się **kluczem** obiektu. Wynika stąd, że struktura rekordu jest szczególnie odpowiednia do reprezentowania obiektów a_i. Dlatego też we wszystkich przytoczonych poniżej algorytmach sortowania będziemy używać typu *obiekt* zdefiniowanego jako

```
type obiekt = record klucz: integer;
                {deklaracje innych składowych}
              end
```
(2.2)

„Inne składowe" reprezentują właściwe dane dotyczące obiektu; klucz służy tylko do identyfikacji obiektów. Jednakże dopóki będziemy zajmować się algorytmami sortowania, dopóty klucz jest *jedyną* istotną składową, nie ma zatem żadnej potrzeby definiowania pozostałych składowych. Wybór typu *integer* jako typu klucza jest cokolwiek arbitralny. Oczywiście można równie dobrze użyć dowolnego typu, w którym jest zdefiniowana relacja liniowego porządku.

Metodę sortowania nazywamy **stabilną** (ang. *stable*), jeżeli podczas procesu sortowania pozostaje nie zmieniony względny porządek obiektów o identycznych kluczach. Stabilność sortowania jest często pożądana wtedy, gdy obiekty są już uporządkowane (posortowane) według pewnych drugorzędnych kluczy, tzn. według właściwości, których nie odzwierciedla klucz podstawowy.

Rozdziału tego *nie* należy traktować jako wyczerpującego przeglądu metod sortowania. Przedstawiono w nim bardzo szczegółowo tylko pewne wybrane, specyficzne metody. Czytelnik zainteresowany pełnym potraktowaniem tematyki sortowania może je znaleźć w znakomitym i wyczerpującym kompendium D.E. Knutha [2.7] (zob. także Lorin [2.10]).

2.2. Sortowanie tablic

Najważniejszym wymaganiem stawianym metodom sortowania tablic jest oszczędne korzystanie z dostępnej pamięci. Wynika stąd, że permutowanie obiektów powodujące ich uporządkowanie powinno być wykonywane w miejscu oraz że metody przenoszenia obiektów z tablicy *a* do wynikowej tablicy *b* są właściwie o wiele mniej interesujące. Ograniczając zatem nasz wybór metod do tych, które spełniają kryterium oszczędnego wykorzystania pamięci, przechodzimy do pierwszej klasyfikacji według ich efektywności, tj. oszczędnego zużycia czasu. Dobrą **miarą efektywności** jest liczba *Po* koniecznych **porównań** kluczy i liczba *Pr* koniecznych **przesunięć** (przestawień) **obiektów**. Wielkości te są funkcjami liczby n sortowanych obiektów. Dobre algorytmy sortowania wymagają liczby porównań rzędu $n \log n$, jednakże na początku omówimy kilka łatwych i naturalnych metod sortowania, zwanych **metodami prostymi**. Wymagają one rzędu n^2 porównań kluczy. Następujące trzy ważne powody przemawiają za prezentacją prostych metod, zanim przejdziemy do omówienia szybszych algorytmów.

(1) Proste metody szczególnie dobrze nadają się do wyjaśnienia własności głównych zasad sortowania.
(2) Ich programy są łatwe do zrozumienia i krótkie. Nie należy zapominać, że programy także zajmują pamięć!
(3) Chociaż skomplikowane metody wymagają mniejszej liczby operacji, jednakże operacje te są zazwyczaj bardziej skomplikowane. Wynika stąd, że aczkolwiek prostych metod nie powinno się stosować dla dużych n, to dla dostatecznie małych n są one szybsze.

Metody sortowania przeznaczone do sortowania obiektów w miejscu można podzielić na trzy zasadnicze grupy:

(1) Sortowanie przez wstawianie (ang. *insertion sort*).
(2) Sortowanie przez wybieranie (selekcję; ang. *selection sort*).
(3) Sortowanie przez zamianę (ang. *exchange sort*).

Zajmiemy się teraz analizą i porównaniem tych trzech grup. Programy będą działać na zmiennej *a*, której składowe mają być posortowane w miejscu, i będą się odwoływać do typu danych *obiekt* (2.2) oraz typu *indeks*, zdefiniowanego jako

type *indeks* = 0..*n*; (2.3)
var *a*: **array** [1..*n*] **of** *obiekt*

2.2.1. Sortowanie przez proste wstawianie

Metoda ta jest powszechnie stosowana przez grających w karty. Obiekty (karty) są podzielone umownie na ciąg wynikowy $a_1 \dots a_{i-1}$ i ciąg źródłowy $a_i \dots a_n$. W każdym kroku, począwszy od $i = 2$ i zwiększając i o jeden, i-ty element ciągu źródłowego przenosi się do ciągu wynikowego, *wstawiając* go w odpowiednim miejscu.

TABLICA 2.1
Przykładowy proces sortowania przez proste wstawianie

Klucze początkowe	44	55	12	42	94	18	06	67
i = 2	44	55	12	42	94	18	06	67
i = 3	12	44	55	42	94	18	06	67
i = 4	12	42	44	55	94	18	06	67
i = 5	12	42	44	55	94	18	06	67
i = 6	12	18	42	44	55	94	06	67
i = 7	06	12	18	42	44	55	94	67
i = 8	06	12	18	42	44	55	67	94

Proces **sortowania przez wstawianie** pokazano na przykładzie ośmiu losowo wybranych liczb (zob. tabl. 2.1). A oto algorytm prostego wstawiania:

```
for i := 2 to n do
    begin x := a[i];
        „wstaw x w odpowiednim miejscu w a1 ... ai”
    end
```

Podczas znajdowania odpowiedniego miejsca wygodnie jest stosować na przemian porównania i przesunięcia, tzn. „przesiewać” x przez porównanie x z następnym obiektem a_j; następnie albo wstawić x, albo przesunąć a_j na prawo, po czym postąpić analogicznie z kolejnym obiektem na lewo. Zauważmy, że zakończenie procesu „przesiewania” może nastąpić z dwóch powodów:

(1) Znaleziono obiekt a_j z kluczem mniejszym niż klucz x.
(2) Osiągnięto lewy koniec ciągu wynikowego.

Ten typowy przypadek iteracji z dwoma warunkami kończącymi przywodzi nas do dobrze znanej metody z wartownikiem. Łatwo ją w tym przypadku zastosować przez utworzenie obiektu wartownika $a_0 = x$. (Zauważmy, że z tego powodu należy rozszerzyć zakres indeksu w deklaracji a do $0..n$). Pełny algorytm jest sformułowany w programie 2.1.

PROGRAM 2.1
Sortowanie przez proste wstawianie

```
procedure prostewstawianie;
   var i, j: indeks; x: obiekt;
begin
   for i := 2 to n do
   begin x := a[i]; a[0] := x; j := i-1;
      while x.klucz < a[j].klucz do
         begin a[j+1] := a[j]; j := j-1
         end;
      a[j+1] := x
   end
end
```

Analiza metody prostego wstawiania. Liczba Po_i porównań kluczy w i-tym przesiewaniu wynosi co najwyżej $i-1$, co najmniej 1 oraz – przy założeniu, że wszystkie permutacje n kluczy są równie prawdopodobne – średnio $i/2$. Liczba Pr_i przesunięć (podstawień obiektów) wynosi Po_i+2 (razem z ustawieniem wartownika). Wynika stąd, że liczby porównań i przesunięć przedstawiają się następująco:

$$Po_{min} = n - 1 \qquad Pr_{min} = 2(n-1)$$

$$Po_{śr} = \frac{1}{4}(n^2+n-2) \qquad Pr_{śr} = \frac{1}{4}(n^2+9n-10) \tag{2.4}$$

$$Po_{max} = \frac{1}{2}(n^2+n)-1 \qquad Pr_{max} = \frac{1}{2}(n^2+3n-4)$$

Liczby te są najmniejsze wtedy, gdy obiekty są uporządkowane już w ciągu źródłowym, największe zaś, gdy są one początkowo ustawione w odwrotnej kolejności. W tym sensie sortowanie przez wstawianie wykazuje rzeczywiście *zachowanie naturalne*. Jest oczywiste, że podany algorytm opisuje *stabilny* proces sortowania: pozostawia nie zmieniony porządek obiektów z identycznymi kluczami.

Algorytm prostego wstawiania można łatwo poprawić, jeśli zauważymy, że ciąg wynikowy $a_1 \ldots a_{i-1}$, w którym należy umieścić obiekt, jest już uporządkowany. W związku z tym można zastosować szybszą metodę ustalenia miejsca wstawienia nowego obiektu. Najprostszym sposobem jest zastosowanie metody przeszukiwania połówkowego, w którym próbkuje się ciąg wynikowy w środku i dzieli go dalej na połowę, aż znajdzie się miejsce wstawienia nowego obiektu. Zmodyfikowany algorytm wstawiania, zwany **wstawianiem połówkowym**, pokazano w programie 2.2.

PROGRAM 2.2
Sortowanie przez wstawianie połówkowe

```
procedure wstawianiepołówkowe;
    var i, j, l, p, m: indeks; x: obiekt;
begin
    for i := 2 to n do
    begin x := a[i]; l := 1; p := i-1;
        while l ⩽ p do
            begin m := (l+p) div 2;
                if x.klucz < a[m].klucz then p := m-1 else l := m+1
            end;
        for j := i-1 downto l do a[j+1] := a[j];
        a[l] := x;
    end
end
```

Analiza metody wstawiania połówkowego. Miejsce wstawienia nowego obiektu jest znalezione, jeśli $a_j.klucz \leqslant x.klucz < a_{j+1}.klucz$. Tak więc przeszukiwany przedział musi mieć końcową długość 1, a stąd wynika, że połowienie przedziału złożonego z i kluczy wykonuje się $\lceil \log_2 i \rceil$ razy. Stąd

$$Po = \sum_{i=1}^{n} \lceil \log_2 i \rceil$$

Aproksymując tę sumę całką, otrzymamy

$$\int_1^n \log x \, dx = x(\log x - c) \Big|_1^n = n(\log n - c) + c \tag{2.5}$$

gdzie $c = \log e = 1/\ln 2 = 1{,}44269...$ Liczba porównań nie zależy od początkowego ustawienia obiektów. Jednakże stosowanie dzielenia całkowitego do połowienia długości przeszukiwanego przedziału powoduje, że rzeczywista liczba porównań potrzebnych dla i obiektów może być o 1 większa od spodziewanej. Wynika to stąd, że miejsce wstawienia obiektu w „niższym" końcu jest przeciętnie znajdowane nieco szybciej od miejsca wstawienia w „wyższym" końcu; faworyzowane są więc takie przypadki, w których obiekty są początkowo bardzo nieuporządkowane. Jest to zatem przypadek *nienaturalnego zachowania się* algorytmu sortowania.

$$Po \doteq n(\log n - \log e \pm 0{,}5)$$

Niestety, wskutek zastosowania metody przeszukiwania połówkowego zmniejszyła się tylko liczba porównań, a nie liczba potrzebnych przesunięć. W rzeczywistości przesuwanie obiektów, tzn. kluczy i związanych z nimi informacji, jest przeważnie o wiele bardziej czasochłonne niż porównywanie

kluczy; ulepszenie nie może być duże: najważniejszy składnik Pr jest wciąż rzędu n^2. A w dodatku powtórne sortowanie już posortowanej tablicy zabiera więcej czasu niż proste wstawienie z liniowym przeszukiwaniem! Przykład ten pokazuje, że „oczywiste ulepszenie" ma często o wiele mniejsze znaczenie, niż się tego można było spodziewać, a zdarzają się przypadki, w których „ulepszenie" może mieć w rzeczywistości skutek odwrotny do zamierzonego. Podsumowując, nie wydaje się, aby sortowanie przez wstawianie było metodą odpowiednią dla maszyn cyfrowych: wstawianie obiektu i następujące po tym przesuwanie całego wiersza obiektów o jedną pozycję jest nieekonomiczne. Lepszych wyników można się spodziewać po metodzie, w której przemieszcza się tylko pojedyncze obiekty na większą odległość. Jest to idea metody sortowania przez wybieranie (selekcję).

2.2.2. Sortowanie przez proste wybieranie

W metodzie tej przyjęto następującą zasadę:

(1) Wybieramy obiekt o najmniejszym kluczu.
(2) Wymieniamy go z pierwszym obiektem a_1.

Powtarzamy te operacje z pozostałymi $n-1$ obiektami, następnie z $n-2$ obiektami, aż pozostanie tylko jeden obiekt – największy. Metodę tę pokazano w tabl. 2.2 na przykładzie tych samych ośmiu kluczy co w tabl. 2.1.

TABLICA 2.2
Przykładowy proces sortowania przez proste wybieranie

Klucze początkowe	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

Program jest sformułowany następująco:

```
for i := 1 to n - 1 do
  begin „przypisz zmiennej k indeks najmniejszego obiektu spośród
        a[i] ... a[n]"; „wymień a_i z a_k"
  end
```

Metoda ta, zwana **prostym wybieraniem**, jest w pewnym sensie odwrotna do prostego wstawiania; w metodzie prostego wstawiania w każdym kroku rozważa się tylko *jeden*, kolejny obiekt *ciągu źródłowego* i *wszystkie* obiekty *tablicy wynikowej*, aby znaleźć miejsce wstawienia nowego obiektu; w metodzie prostego wybierania rozważa się *wszystkie* obiekty *tablicy źródłowej*, aby znaleźć obiekt z najmniejszym kluczem i ustawić go jako kolejny element *ciągu wynikowego*. Pełny algorytm prostego wybierania znajduje się w programie 2.3.

PROGRAM 2.3
Sortowanie przez proste wybieranie

```
procedure prostewybieranie;
    var i, j, k: indeks; x: obiekt;
begin for i := 1 to n-1 do
        begin k := i; x := a[i];
            for j := j+1 to n do
                if a[j].klucz < x.klucz then
                begin k := j; x := a[j]
                end;
            a[k] := a[i]; a[i] := x;
            end
end
```

Analiza metody prostego wybierania. Widzimy, że liczba Po porównań kluczy nie zależy od początkowego ustawienia kluczy. Z tego względu możemy tę metodę uważać za zachowującą się mniej naturalnie niż proste wstawianie. Otrzymujemy

$$Po = \frac{1}{2}(n^2 - n)$$

Liczba Pr przesunięć równa się co najmniej

$$Pr_{\min} = 3(n-1) \tag{2.6}$$

w przypadku początkowo uporządkowanych kluczy, co najwyżej zaś

$$Pr_{\max} = \mathrm{trunc}\left(\frac{n^2}{4}\right) + 3(n-1)$$

jeżeli klucze były ustawione początkowo w odwrotnej kolejności. Średnią $Pr_{śr}$ trudno obliczyć, pomimo prostoty algorytmu. Zależy ona od tego, ile razy k_j jest mniejsze od wszystkich poprzedzających je liczb $k_1 \dots k_{j-1}$ wtedy, gdy przegląda się ciąg liczb $k_1 \dots k_n$. Wartość ta uśredniona dla wszystkich $n!$ permutacji n kluczy wynosi

$H_n - 1$

gdzie H_n jest n-tą **liczbą harmoniczną**

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} \tag{2.7}$$

(zob. Knuth, Vol. 1, s. 95–99).

H_n można wyrazić jako

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \ldots \tag{2.8}$$

gdzie $\gamma = 0{,}577216\ldots$ jest stałą Eulera. Dla dostatecznie dużych n możemy pominąć składniki ułamkowe i przybliżyć średnią liczbę przypisań w i-tym kroku przez

$$F_i = \ln i + \gamma + 1$$

Średnia liczba przesunięć $Pr_{śr}$ w sortowaniu przez wybieranie jest więc sumą F_i dla i zmieniającego się od 1 do n.

$$Pr_{śr} = \sum_{i=1}^{n} F_i = n(\gamma + 1) + \sum_{i=1}^{n} \ln i$$

Dzięki dalszemu przybliżeniu sumy dyskretnych składników przez całkę

$$\int_1^n \ln x \, dx = x(\ln x - 1) \Big|_1^n = n \ln n - n + 1$$

otrzymujemy wartość przybliżoną

$$Pr_{śr} \doteq n(\ln n + \gamma) \tag{2.9}$$

Podsumowując, algorytm prostego wybierania jest przeważnie lepszy od prostego wstawiania, chociaż w przypadkach, w których klucze są już posortowane lub prawie posortowane, proste wstawianie jest jednak nieco szybsze.

2.2.3. Sortowanie przez prostą zamianę

Metod sortowania nie można poklasyfikować w sposób całkowicie jednoznaczny. Obie omówione poprzednio metody mogą być także uważane za sortowanie przez zamianę. W tym punkcie przedstawiamy jednakże metodę, w której zamiana dwóch obiektów jest jej cechą najbardziej charakterystyczną. Podany poniżej algorytm prostej zmiany opiera się na zasadzie porównywania

i zamiany par sąsiadujących ze sobą obiektów dopóty, dopóki wszystkie obiekty nie będą posortowane.

Podobnie jak w poprzednich metodach prostego wybierania, powtarzamy przejścia przez tablicę, przesuwając za każdym razem najmniejszy z pozostałych obiektów na lewy koniec tablicy. Jeżeli dla odmiany ustawimy tablicę pionowo zamiast poziomo – przy pewnej dozie wyobraźni – będziemy uważać jej elementy za znajdujące się w naczyniu z wodą bąbelki o „wagach" proporcjonalnych do ich

TABLICA 2.3
Przykład sortowania bąbelkowego

Klucze początkowe	i=2	i=3	i=4	i=5	i=6	i=7	i=8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

kluczy, to każde przejście przez tablicę da w wyniku wypchnięcie bąbelka na poziom odpowiadający jego wadze (zob. tabl. 2.3). Metoda ta znana jest pod nazwą **sortowania bąbelkowego** (ang. *bubble sort*). Jej najprostszą postać opisano w programie 2.4.

PROGRAM 2.4
Sortowanie bąbelkowe

```
procedure sortowaniebąbelkowe;
   var i, j: indeks; x: obiekt;
begin for i := 2 to n do
      begin for j := n downto i do
            if a[j-1].klucz > a[j].klucz then
            begin x := a[j-1]; a[j-1] := a[j]; a[j] := x
            end
      end
end {sortowaniebąbelkowe}
```

Można łatwo wprowadzić pewne ulepszenia do tego algorytmu. W tablicy 2.3 pokazano, że ostatnie trzy przejścia nie mają żadnego wpływu na kolejność obiektów, ponieważ są one już posortowane. Oczywistym sposobem usprawniającym algorytm będzie zapamiętanie, czy dokonywano w trakcie przejścia jakichś zamian. Przejście, w którym nie można już wprowadzić żadnej zamiany,

wskazuje, że można zakończyć wykonanie algorytmu. Dodatkowym ulepszeniem tego usprawnienia będzie zapamiętanie pozycji (indeks k) ostatniej zamiany, a nie samego faktu jej dokonania. Na przykład jasne jest, że wszystkie pary obiektów sąsiadujących poniżej tego indeksu k są ustawione w pożądanej kolejności. Następne przejścia można więc zakończyć na tym indeksie, zamiast iść aż do ustalonej uprzednio dolnej granicy i. Uważny programista dostrzeże jednakże osobliwą asymetrię. Mianowicie pojedynczy, źle ustawiony bąbelek z „ciężkiego" końca tablicy, która oprócz tego jednego obiektu jest już posortowana, zostanie przeniesiony na miejsce w pojedynczym przejściu, a źle ustawiony obiekt z „lekkiego" końca będzie zmierzał w stronę właściwego miejsca tylko o jeden krok w każdym przejściu. Na przykład tablica

12 18 42 44 55 67 94 06

będzie posortowana ulepszoną metodą sortowania bąbelkowego w pojedynczym przejściu, ale tablica

94 06 12 18 42 44 55 67

wymaga do posortowania siedmiu przejść. Ta nienaturalna asymetria sugeruje trzecie ulepszenie: zmianę kierunku kolejnych przejść. Stosowną nazwą dla otrzymanego algorytmu jest **sortowanie mieszane** (ang. *shake sort*). W tablicy 2.4 zilustrowano jego działanie w przypadku zastosowania go do tych samych ośmiu kluczy co w tabl. 2.3.

TABLICA 2.4
Przykład sortowania mieszanego

i=2	3	3	4	4
p=8	8	7	7	4
↑	↓	↑	↓	↑
44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	42	44
18	94	18	55	55
06	18	67	67	67
67	67	94	94	94

Analiza sortowania bąbelkowego i sortowania mieszanego. Liczba porównań w algorytmie prostej zamiany wynosi

$$Po = \frac{1}{2}(n^2 - n) \tag{2.10}$$

a najmniejsza, średnia i największa liczba przesunięć (przypisań) obiektów wynosi odpowiednio:

$$Pr_{\min}=0, \quad Po_{śr}=\frac{3}{4}(n^2-n), \quad Po_{\max}=\frac{3}{2}(n^2-n) \tag{2.11}$$

PROGRAM 2.5
Sortowanie mieszane

```
procedure sortowaniemieszane;
   var j, k, l, p: indeks; x: obiekt;
begin l := 2; p := n; k := n;
   repeat
      for j := p downto l do
         if a[j-1].klucz>a[j].klucz then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end;
      l := k+1;
      for j := l to p do
         if a[j-1].klucz>a[j].klucz then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end;
      p := k-1;
   until l>p
end {sortowaniemieszane}
```

Analiza metod ulepszonych, zwłaszcza zaś sortowania mieszanego jest skomplikowana. Najmniejsza liczba porównań wynosi $Po_{\min}=n-1$. Knuth wykazał, że dla poprawionej metody sortowania bąbelkowego średnia liczba przejść jest proporcjonalna do $n-k_1\sqrt{n}$, a średnia liczba porównań jest proporcjonalna do $\frac{1}{2}[n^2-n(k_2+\ln n)]$. Zauważmy jednak, że wszystkie podane powyżej usprawnienia w żaden sposób nie wpływają na liczbę przesunięć obiektów; redukują tylko liczbę zbędnych, podwójnych sprawdzeń. Niestety, zamiana dwóch obiektów jest przeważnie o wiele kosztowniejszą operacją niż porównywanie kluczy; dlatego też nasze pomysłowe ulepszenia mają wbrew intuicji o wiele mniejszy wpływ na wynik końcowy.

Analiza wykazuje, że metoda sortowania przez zamianę i jej niewielkie ulepszenia są gorsze zarówno od metody sortowania przez wstawianie, jak i od metody sortowania przez wybieranie; w rzeczywistości sortowanie bąbelkowe nie może nam nic zaoferować prócz zabawnej nazwy. Algorytm sortowania miesza-

nego daje korzyści w przypadkach, w których obiekty są już prawie uporządkowane – w praktyce są to rzadkie przypadki.

Można wykazać, że średnia odległość, jaką każdy z n obiektów musi przebyć podczas sortowania, wynosi $n/3$ miejsc. Ta uwaga podsuwa nam pewien kierunek poszukiwań ulepszonych, tzn. efektywniejszych metod sortowania. We wszystkich bowiem prostych metodach sortowania, tak naprawdę, przesuwa się każdy obiekt w jednym elementarnym kroku o jedno miejsce. Dlatego też ich wykonanie musi wymagać rzędu n^2 takich kroków. Każde ulepszenie musi się opierać na założeniu przesuwania obiektów w pojedynczym skoku na większą odległość.

Poniżej omówimy trzy ulepszone metody, po jednej dla każdej podstawowej metody sortowania: wstawiania, wybierania i zamiany.

2.2.4. Sortowanie za pomocą malejących przyrostów

Poprawienie metody prostego sortowania przez wstawianie zaproponował D.L. Shell w 1959 r. Metodę tę wyjaśnimy na naszym wzorcowym przykładzie ośmiu obiektów (zob. tabl. 2.5). Początkowo grupuje się i sortuje oddzielnie wszystkie obiekty oddalone od siebie o cztery miejsca (przyrost jest równy 4). Proces ten nazywamy sortowaniem „co cztery". W przykładzie ośmiu obiektów każda grupa zawiera dokładnie po dwa obiekty. Po tej pierwszej fazie tworzy się grupy składające się z obiektów oddalonych o dwa miejsca i od nowa sortuje. Ten proces nazywamy sortowaniem „co dwa". Na koniec – w trzeciej fazie – wszyskie obiekty sortuje się normalnie, czyli „co jeden".

W pierwszej chwili nasuwa się pytanie: czy konieczność wykonania kilku faz sortowania, z których każda obejmuje wszystkie obiekty, nie zwiększy jeszcze

TABLICA 2.5
Sortowanie za pomocą malejących przyrostów

	44	55	12	42	94	18	06	67
Sortowanie „co 4"								
	44	18	06	42	94	55	12	67
Sortowanie „co 2"								
	06	18	12	42	44	55	94	67
Sortowanie „co 1"								
	06	12	18	42	44	55	67	94

bardziej nakładu pracy. Jednakże każdy krok sortowania łańcucha albo obejmuje stosunkowo mało obiektów, albo dotyczy obiektów już dość dobrze uporządkowanych i wymagających wykonania niewielu przestawień.

Jak można łatwo zauważyć, metoda ta prowadzi do tablicy uporządkowanej; jest też dość oczywiste, że w każdym kolejnym kroku zyskuje się w stosunku do kroków poprzednich (ponieważ każde sortowanie „co i" łączy dwie grupy posortowane w poprzednim sortowaniu „co $2i$"). Jest także oczywiste, że można zaakceptować każdy ciąg przyrostów, jeżeli tylko ostatni przyrost będzie jednością, ponieważ – w najgorszym razie – w ostatniej fazie wykonana zostanie cała praca. Jednakże o wiele mniej oczywiste jest to, że metoda malejących przyrostów daje nawet lepsze wyniki wtedy, gdy stosuje się przyrosty różne od potęg liczby 2.

Z tego powodu program budujemy bez założenia o określonym ciągu przyrostów. Przyrosty te są oznaczone przez

$$h_1, h_2, ..., h_t$$

przy czym spełnione jest

$$h_t = 1, \; h_{i+1} < h_i \tag{2.12}$$

Każde z h-sortowań programujemy metodą sortowania przez proste wstawianie; aby uprościć warunek kończący szukanie miejsca wstawienia obiektu, stosujemy metodę z wartownikiem.

Jest oczywiste, że każde sortowanie musi ustawić własnego wartownika oraz że program musi jak najprościej określać jego miejsce. Nie wystarczy więc rozszerzyć tablicę a o pojedynczy obiekt $a[0]$, lecz należy ją rozszerzyć o h_1 obiektów. Prowadzi to do następującej deklaracji a:

a: **array** $[-h_1. .n]$ **of** *obiekt*

Algorytm ten jest opisany przez procedurę o nazwie *sortowanieShella* [2.11] w programie 2.6 dla $t=4$.

PROGRAM 2.6
Sortowanie metodą Shella

```
procedure sortowanieShella;
   const t=4;
   var i, j, k, s: indeks; x: obiekt; m: 1..t;
      h: array [1..t] of integer;
begin h[1]:= 9; h[2]:= 5; h[3]:= 3; h[4]:= 1;
   for m:=1 to t do
   begin k:= h[m]; s:= -k;   {miejsce wartownika}
      for i:= k+1 to n do
      begin x:= a[i]; j:= i-k;
         if s=0 then s:= -k; s:= s+1; a[s]:= x;
```

```
            while x.klucz<a[j].klucz do
            begin a[j+k]:=a[j]; j:=j-k
            end;
            a[j+k]:=x
        end
    end
end
```

Analiza sortownia metodą Shella. Analiza tego algorytmu prowadzi do pewnych bardzo trudnych problemów matematycznych, z których wiele nie doczekało się jeszcze rozwiązania. W szczególności nie wiadomo, jaki wybór przyrostów pozwala uzyskać najlepsze rezultaty. Zaskakujące jest jednakże, że przyrosty nie powinny być swoimi dzielnikami. Uniknie się wtedy zjawiska występującego w powyższym przykładzie, w którym każda faza sortowania łączy dwa łańcuchy nie mające przedtem ze sobą żadnego związku. Pożądane jest, aby różne łańcuchy oddziaływały na siebie jak najczęściej oraz by prawdziwe było następujące twierdzenie:

Jeżeli ciąg posortowany „co k" posortujemy „co i", to ciąg ten pozostaje posortowany „co k".

Knuth [2.8] wykazuje, że trafnym wyborem przyrostów jest ciąg (wypisany w odwrotnej kolejności):

1, 4, 13, 40, 121, ...

gdzie $h_{k-1}=3h_k+1$, $h_t=1$ i $t=\lfloor \log_3 n \rfloor-1$. Zaleca także ciąg

1, 3, 7, 15, 31, ...

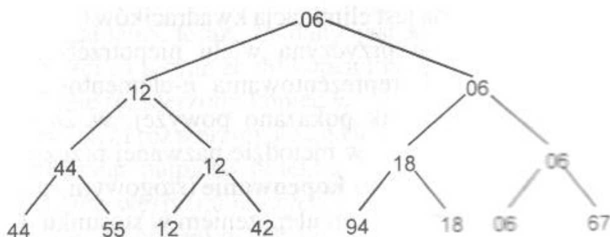
gdzie $h_{k-1}=2h_k+1$, $h_t=1$ i $t=\lfloor \log_2 n \rfloor-1$.

Analiza matematyczna algorytmu sortowania metodą Shella wykonana dla drugiego wyboru wykazuje, że do posortowania n obiektów konieczny jest nakład pracy proporcjonalny do $n^{1.2}$. Chociaż jest to znaczny postęp w stosunku do n^2, nie będziemy się dłużej zajmować tą metodą, ponieważ znane są jeszcze lepsze algorytmy.

2.2.5. Sortowanie drzewiaste

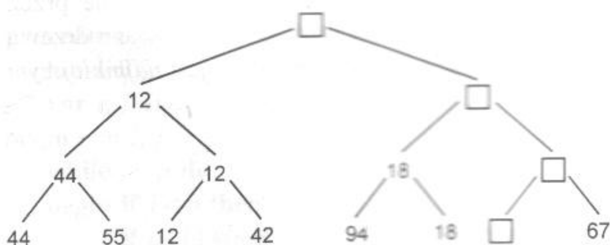
Metoda sortowania przez proste wybieranie polega na wielokrotnym dokonaniu wyboru najmniejszego klucza spośród n obiektów, spośród $n-1$ obiektów itd. Jasne, że znalezienie najmniejszego klucza spośród n obiektów wymaga $n-1$ porównań, a na znalezienie go wśród $n-1$ obiektów potrzeba $n-2$ porównań. A więc jak można poprawić sortowanie przez wybieranie? Może się to udać tylko przez zapamiętanie większej ilości informacji, oprócz zwykłej identyfikacji pojedynczego najmniejszego obiektu. Na przykład za pomocą $n/2$

porównań można wyznaczyć mniejszy klucz każdej pary obiektów, za pomocą następnych $n/4$ porównań można wybrać mniejszy z każdej pary tych mniejszych kluczy itd. W końcowym efekcie za pomocą tylko $n-1$ porównań możemy zbudować drzewo wyboru takie jak na rys. 2.3 i jako korzeń wyznaczyć najmniejszy klucz [2.2].



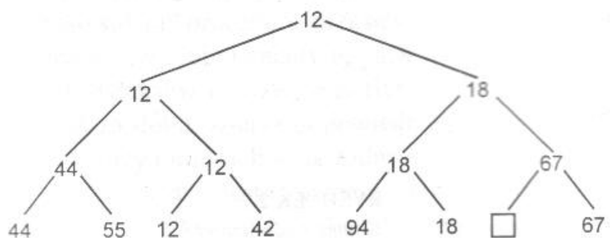
RYSUNEK 2.3
Wielokrotny wybór między dwoma kluczami

Na drugi krok składa się teraz zejście po ścieżce wyznaczonej przez najmniejszy klucz i wyeliminowanie go przez zastąpienie kolejno albo przez pusty kwadracik (lub klucz $-\infty$) na samym dole, albo obiekt z alternatywnej gałęzi w węzłach pośrednich (zob. rys. 2.4 i 2.5). Znowu obiekt, który się pojawił w korzeniu drzewa, ma najmniejszy (drugi z kolei) klucz i można go w ten sam sposób usunąć. Po n takich krokach wybierania drzewo jest puste (tzn. pełne kwadracików) i proces sortowania jest zakończony. Należy zauważyć, że każdy



RYSUNEK 2.4
Wybór najmniejszego klucza

z n kroków wybierania wymaga tylko $\log_2 n$ porównań. Stąd cały proces sortowania wymaga tylko rzędu $n \cdot \log n$ podstawowych operacji oraz dodatkowo n operacji potrzebnych do zbudowania drzewa. Jest to bardzo poważna poprawa w stosunku do prostych metod wymagających rzędu n^2 kroków, a nawet w stosunku do sortowania metodą Shella wymagającej rzędu $n^{1.2}$ kroków.



RYSUNEK 2.5
Wypełnianie kwadracików

Oczywiście wzrosła liczba operacji pomocniczych związanych z zadaniem i dlatego w metodzie sortowania drzewiastego pojedyncze kroki są bardziej skomplikowane. Przede wszystkim, w celu zapamiętania zwiększonej porcji informacji zebranej w pierwszym przejściu, trzeba stworzyć pewien rodzaj struktury drzewiastej. Naszym następnym zadaniem będzie znalezienie metod efektywnego zorganizowania tej informacji.

Jest oczywiste, że szczególnie pożądana jest eliminacja kwadracików ($-\infty$), które w końcu zajmują całe drzewo i są przyczyną wielu niepotrzebnych porównań. Ponadto trzeba znaleźć sposób reprezentowania n-elementowego drzewa w n jednostkach pamięci zamiast, jak pokazano powyżej, w $2n-1$ jednostkach. Cele te są faktycznie zrealizowane w metodzie nazwanej przez jej wynalazcę J. Williamsa [2.14] **sortowaniem przez kopcowanie** (stogowym; ang. *heap sort*). Widoczne jest, że metoda ta jest wielkim ulepszeniem w stosunku do bardziej znanych rozwiązań sortowania drzewiastego.

Kopiec (stóg; ang. *heap*) definiujemy jako ciąg kluczy

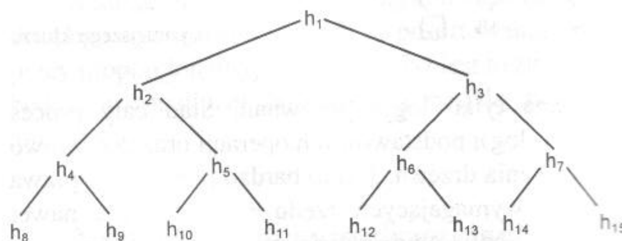
$h_l, h_{l+1}, \ldots, h_p$

takich, że

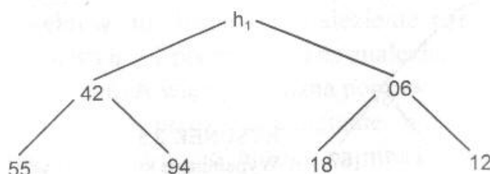
$$h_i \leqslant h_{2i}$$
$$h_i \leqslant h_{2i+1} \qquad (2.13)$$

dla wszystkich $i = l \ldots p/2$. Jeżeli drzewo binarne jest reprezentowane przez tablicę taką, jaką pokazano na rys. 2.6, to wynika stąd, że posortowane drzewa z rys. 2.7 i rys. 2.8 są kopcami, a w szczególności, że obiekt h_1 jest *najmniejszym* elementem kopca.

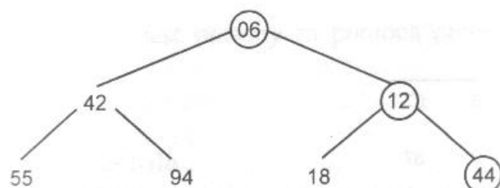
$h_1 = min(h_1 \ldots h_n)$



RYSUNEK 2.6
Tablica h przedstawiona jako drzewo binarne



RYSUNEK 2.7
Kopiec siedmioelementowy



RYSUNEK 2.8
Klucz 44 przesiewany przez kopiec

Załóżmy teraz, że dany jest kopiec o elementach $h_{l+1} \dots h_p$ dla pewnych wartości l i p oraz że do kopca ma być dodany nowy obiekt x tak, że utworzony zostanie rozszerzony kopiec $h_l \dots h_p$. Weźmy na przykład kopiec $h_1 \dots h_7$ pokazany na rys. 2.7 i rozszerzmy go „na lewo" o obiekt $h_1 = 44$. Nowy kopiec otrzymujemy, wstawiając najpierw obiekt x na wierzchołek drzewa, a później „przesiewając" go przez węzły mniejszych elementów kopca, które podnoszą się dzięki temu do góry. W podanym przykładzie liczba 44 jest przestawiona najpierw z liczbą 06, a następnie z liczbą 12, tworząc w ten sposób drzewo pokazane na rys. 2.8. Sformułujemy teraz ten algorytm przesiewania następująco: i, j są parą indeksów wskazujących na elementy, które mają być zamienione podczas każdego kroku przesiewania. Czytelnika zachęcamy do upewnienia się, czy zaproponowana metoda przesiewania rzeczywiście zachowuje warunki (2.13) definiujące kopiec.

PROGRAM 2.7
Przesiewanie

```
procedure przesiewanie (l, p: indeks);
    label 13;
    var i, j: indeks; x: obiekt;
begin i := l; j := 2*i; x := a[i];
    while j≤p do
    begin if j<p then
            if a[j].klucz>a[j+1].klucz then j := j+1;
            if x.klucz≤a[j].klucz then goto 13;
            a[i] := a[j]; i := j; j := 2*i   {przesiewanie}
    end;
13: a[i] := x
end;
```

Pewien zręczny sposób budowy kopca w miejscu zaproponował R.W. Floyd. Zastosował on procedurę przesiewania pokazaną w programie 2.7. Dana jest tablica $h_1 \dots h_n$; jej elementy $h_{n/2} \dots h_n$ tworzą już kopiec, ponieważ nie ma takich dwóch indeksów i, j, że $j = 2i$ (lub $j = 2i+1$). Obiekty te tworzą to, co można uważać za dolny wiersz odpowiedniego drzewa binarnego (zob. rys. 2.6) i nie musi między nimi zachodzić żadna relacja. Kopiec rozszerzamy obecnie na lewo. W każdym kroku jest dołączany nowy obiekt i ustawiany prawidłowo dzięki przesiewaniu. Proces ten, zilustrowny w tabl. 2.6, daje w wyniku kopiec poka-

TABLICA 2.6
Budowa kopca

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
44	55	06	42	94	18	12	67
44	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

zany na rys. 2.6. W rezultacie proces tworzenia kopca n-elementowego $h_1 \dots h_n$ w miejscu jest opisany następująco:

```
l := (n div 2) + 1;
while l > 1 do
   begin l := l - 1; przesiewanie (l, n)
   end
```

W celu posortowania elementów kopca trzeba teraz wykonać n kroków przesiewania; po każdym kroku kolejny element może być zdjęty z wierzchołka kopca. Jeszcze raz powstają pytania: gdzie trzymać zmieniające się elementy wierzchołkowe i czy będzie możliwe sortowanie w miejscu. Oczywiście, jest takie rozwiązanie! W każdym kroku należy zdjąć z kopca jego ostatni element, powiedzmy x, przechować wierzchołkowy element w zwolnionym przez x miejscu i przesiać x na właściwe miejsce. Potrzebne do posortowania elementów $n-1$ kroków pokazano na przykładzie kopca w tabl. 2.7.

TABLICA 2.7
Przykład procesu sortowania przez kopcowanie

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

Proces ten jest opisany za pomocą procedury *przesiewanie* (program 2.7) następująco:

```
p := n;
while p > 1 do
    begin x := a[1]; a[1] := a[p]; a[p] := x;
        p := p - 1; przesiewanie(l, p)
    end
```

Przykład z tabl. 2.7 pokazuje, że końcowy porządek jest w rzeczywistości odwrócony. Można to jednakże łatwo usunąć, zmieniając kierunek relacji porządku w procedurze przesiewania. W wyniku otrzymujemy procedurę *sortowanieprzezkopcowanie* przedstawioną w programie 2.8.

PROGRAM 2.8
Sortowanie przez kopcowanie

```
procedure sortowanieprzezkopcowanie;
    var l, p: indeks; x: obiekt;
    procedure przesiewanie;
        label 13;
        var i, j: indeks;
    begin i := l; j := 2*i; x := a[i];
        while j ≤ p do
        begin if j < p then
                  if a[j].klucz < a[j+1].klucz then j := j+1;
            if x.klucz ≥ a[j].klucz then goto 13;
            a[i] := a[j]; i := j; j := 2*i
        end;
    13: a[i] := x
    end;
begin l := (n div 2)+1; p := n;
    while l > 1 do
        begin l := l-1; przesiewanie
        end;
    while p > 1 do
        begin x := a[1]; a[1] := a[p]; a[p] := x;
            p := p-1; przesiewanie
        end
end {sortowanieprzezkopcowanie}
```

Analiza metody sortowania przez kopcowanie. Na pierwszy rzut oka nie widać, że ta metoda daje dobre wyniki. Przede wszystkim duże obiekty są najpierw przesiewane w lewo, a dopiero później przenoszone daleko w prawo. Rzeczywiście, nie zaleca się stosowania tej procedury dla małej liczby obiektów,

takiej jak przedstawiona w przykładzie. Jednakże dla dużych n sortowanie przez kopcowanie jest bardzo efektywne i im większe n, tym lepsze daje wyniki – nawet w porównaniu z sortowaniem metodą Shella. W najgorszym przypadku potrzeba $n/2$ kroków przesiewania, w których obiekty są przesiewane przez $\log(n/2)$, $\log(n/2-1)$, ..., $\log(n-1)$ miejsc, przy czym logarytm jest brany przy podstawie 2 i obcinany do najbliższej, mniejszej wartości całkowitej. Następnie faza sortowania wymaga $n-1$ przesiewań z co najwyżej $\log(n-1)$, $\log(n-2)$, ..., 1 przestawieniami obiektów. Ponadto potrzeba $n-1$ przestawień, aby zapamiętać obiekt przesiany z prawej strony. Rozumowanie to wykazuje, że metoda sortowania przez kopcowanie wymaga, *nawet w najgorszym przypadku*, liczby kroków rzędu $n \cdot \log n$. Doskonała efektywność dla najgorszego przypadku jest jedną z najważniejszych zalet metody sortowania przez kopcowanie.

Nie jest w zupełności jasne, w jakim przypadku można się spodziewać najgorszej (lub najlepszej) efektywności. Ogólnie wydaje się, że sortowanie przez kopcowanie faworyzuje takie ciągi wejściowe, których elementy są w mniejszym lub większym stopniu uporządkowane w odwrotnej kolejności i dlatego metoda ta wykazuje zachowanie nienaturalne. Oczywiście, jeżeli obiekty były uporządkowane odwrotnie, to faza budowy kopca nie wymaga żadnego przesuwania. Dla ośmiu obiektów z naszego przykładu następujące ciągi początkowe wymagają odpowiednio najmniejszej i największej liczby przesunięć:

$Pr_{\min} = 13$ dla ciągu

94 67 44 55 12 42 18 6

$Pr_{\max} = 24$ dla ciągu

18 42 12 44 6 55 67 94

Średnia liczba przesunięć elementów ciągu wynosi w zaokrągleniu $\frac{1}{2} \cdot n \cdot \log n$, a odchylenia od tej wartości są stosunkowo niewielkie.

2.2.6. Sortowanie przez podział

Po omówieniu dwóch nowoczesnych metod sortowania, działających na zasadach wstawiania i wybierania, przedstawimy trzecią, ulepszoną metodę, w której stosuje się zasadę zamiany. Pamiętając o tym, że sortowanie bąbelkowe było na ogół najmniej efektywne z trzech prostych metod sortowania, można przewidywać możliwość poważnych ulepszeń. Nadspodziewanie ulepszenia wprowadzone do metody sortowania przez zamianę, które będą omówione w tym punkcie, dają najlepszą ze znanych dotychczas metod sortowania tablic. Jej efektywność tak się rzucała w oczy, że wynalazca C.A.R. Hoare nazwał ją **sortowaniem szybkim** (ang. *quick sort*) [2.5] i [2.6].

W metodzie sortowania szybkiego korzysta się z faktu, że w celu zapewnienia efektywności powinno się wymieniać obiekty położone daleko od siebie. Załóżmy, że danych jest n obiektów ustawionych w odwrotnym porządku kluczy. Można posortować je, wykonując tylko $n/2$ wymian, biorąc najpierw obiekty – skrajny z lewej strony i skrajny z prawej strony, a następnie posuwać się stopniowo do środka z obu stron. Oczywiście takie postępowanie możliwe jest tylko dlatego, że uporządkowanie było dokładnie odwrotne. Lecz mimo wszystko czegoś można się z tego przykładu nauczyć.

Spróbujmy użyć następującego algorytmu. Wybierzmy losowo jakiś obiekt i nazwijmy go x; przeglądajmy tablicę od lewej strony, aż znajdziemy obiekt $a_i > x$, a następnie przeglądajmy tablicę od prawej, aż znajdziemy $a_j < x$. Wymieńmy teraz te dwa obiekty i kontynuujmy proces „przeglądania i zamiany", aż nastąpi spotkanie gdzieś w środku tablicy. W rezultacie otrzymamy tablicę podzieloną na lewą część z kluczami mniejszymi od x oraz prawą część z kluczami większymi od x. Ten proces dzielenia jest sformułowany w postaci procedury w programie 2.9. Zauważmy, że relacje $>$ i $<$ są zastąpione przez relacje $\geqslant$ i $\leqslant$, których zaprzeczeniami w instrukcji **while** są relacje $<$ i $>$. Po tej zmianie x pełni funkcję wartownika zarówno przy posuwaniu się od lewej, jak i od prawej strony.

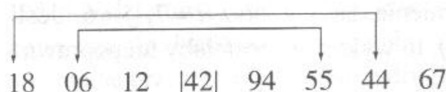
PROGRAM 2.9
Podział

```
procedure podział;
    var i, j: indeks; w, x: obiekt;
begin i := 1; j := n;
    wybierz losowo obiekt x;
    repeat
        while a[i].klucz < x.klucz do i := i+1;
        while x.klucz < a[j].klucz do j := j-1;
        if i ≤ j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
                  i := i+1; j := j-1
            end
    until i > j
end
```

Jeżeli na przykład za x wybierzemy środkowy klucz 42, to w tablicy kluczy

44 55 12 42 94 06 18 67

trzeba dokonać dwóch wymian, aby otrzymać tablicę podzieloną



18 06 12 |42| 94 55 44 67

Ostanie wartości indeksów wynoszą $i=5$ oraz $j=3$. Klucze $a_1 \dots a_{i-1}$ są mniejsze bądź równe kluczowi $x=42$, klucze $a_{j-1} \dots a_n$ są większe bądź równe temu kluczowi. Wynika stąd, że otrzymaliśmy podział na dwie części, a mianowicie

$$a_k.klucz \leqslant x.klucz \quad \text{dla} \quad k=1 \dots i-1 \qquad (2.14)$$
$$a_k.klucz \geqslant x.klucz \quad \text{dla} \quad k=j+1 \dots n$$

oraz

$$a_k.klucz = x.klucz \quad \text{dla} \quad k=j+1 \dots i-1$$

Algorytm ten jest bardzo prosty i efektywny, ponieważ zmienne i, j oraz x mogą być w czasie przeglądania tablicy trzymane w szybkich rejestrach maszyny cyfrowej. Jednakże algorytm może być też nieporęczny, jak świadczy o tym przypadek n identycznych kluczy, w którym dokonuje się $n/2$ wymian elementów. Te niepotrzebne wymiany można łatwo wyeliminować, zmieniając instrukcje przeglądania na

```
while a[i].klucz ≤ x.klucz do i := i+1;
while x.klucz ≤ a[j].klucz do j := j-1;
```

Jednakże w tej sytuacji wybrany element x, który należy przecież do tablicy, nie może być już wartownikiem podczas przeglądania tablicy. Sytuacja, w której wszystkie klucze będą identyczne, spowoduje wyjście poza granice tablicy, chyba że zastosujemy bardziej skomplikowany warunek kończący przeglądanie. Prostota warunków użytych w progamie 2.9 równoważy dodatkowe wymiany, które zdarzają się stosunkowo rzadko w przeciętnym, „losowym" przypadku. Drobną oszczędność można jednak osiągnąć, zmieniając warunek sterujący krokiem wymiany na

$i<j$

zamiast $i \leqslant j$. Zmianą tą nie można jednakże objąć dwóch instrukcji:

$i := i+1; \; j := j-1$

które wobec tego wymagają osobnej instrukcji warunkowej. Potrzebę tej instrukcji wykazuje następujący przykład z $x=2$:

1 1 1 2 1 1 1

Pierwsze przejście i wymiana daje w wyniku

1 1 1 1 1 1 2

oraz $i=5$, $j=6$. Drugie przejście nie zmienia tablicy oraz $i=7$, $j=6$. Jeśli wymiana nie byłaby objęta warunkiem $i \leqslant j$, to wykonana zostałaby niepoprawna wymiana a_6 z a_7.

Można nabrać zaufania do poprawności algorytmu podziału, sprawdziwszy, że dwa stwierdzenia (2.14) są niezmiennikami instrukcji **repeat**. Początkowo, gdy $i=1$ i $j=n$, są one oczywiście prawdziwe, a przy wyjściu z wartościami $i>j$ zapewniają pożądany wynik.

Przypomnijmy, że naszym celem było nie tylko znalezienie podziałów elementów tablicy, lecz także posortowanie tablicy. Jednakże od podziału tablicy do jej posortowania jest tylko jeden mały krok: po podzieleniu tablicy przeprowadzamy ten sam proces dla jej obu części, potem dla części tych części itd. – dopóty, dopóki każda z części nie będzie się składać tylko z jednego obiektu. Taki sposób postępowania jest opisany w programie 2.10.

PROGRAM 2.10
Sortowanie szybkie

```
procedure sortowanieszybkie;
   procedure sortuj(l, p: indeks);
      var i, j: indeks; x, w: obiekt;
   begin i := l; j := p;
      x := a[(l+p) div 2];
      repeat
         while a[i].klucz < x.klucz do i := i+1;
         while x.klucz < a[j].klucz do j := j-1;
         if i <= j then
         begin w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1
         end
      until i > j;
      if l < j then sortuj(l, j);
      if i < p then sortuj(i, p)
   end;
begin sortuj (1, n)
end {sortowanieszybkie}
```

Procedura *sortuj* wywołuje siebie rekurencyjnie. Użycie rekursji w algorytmach jest bardzo skutecznym narzędziem i będzie dokładnie omówione w rozdz. 3. W niektórych starszych językach programowania rekursja z pewnych technicznych względów jest zabroniona. Pokażemy teraz, jak ten sam algorytm można napisać w postaci procedury nierekurencyjnej. Narzucającym się rozwiązaniem jest zastąpienie rekursji przez iterację; będą jednak wtedy potrzebne dodatkowe operacje pomocnicze.

Sprawą kluczową dla rozwiązania iteracyjnego jest przechowywanie listy żądań podziałów części tablicy, których jeszcze nie dokonano. Po każdym kroku pojawiają się dwa żądania podziału. Tylko jedno z nich może być bezpośrednio wykonane przez kolejną iterację; drugie przechowuje się na tej liście. Oczywiście

ważne jest, aby lista żądań była ustawiona w określonej kolejności, a mianowicie – w odwrotnej kolejności. Wynika stąd, że pierwsze zapamiętane żądanie musi być wykonane na końcu i vice versa; lista zachowuje się podobnie do pulsującego stosu. W poniższej, nierekurencyjnej wersji sortowania szybkiego, każde żądanie jest reprezentowane po prostu przez lewy i prawy indeks podziału, który ma być jeszcze dalej dzielony. Tak więc wprowadzimy zmienną tablicową o nazwie *stos* oraz indeks *s* zaznaczający ostatnie zajęte miejsce stosu (zob. progr. 2.11). Wybór odpowiedniej wielkości stosu *m* będzie przedyskutowany podczas dokonywania analizy metody sortowania szybkiego.

PROGRAM 2.11
Nierekurencyjna wersja sortowania szybkiego

```
procedure sortowanieszybkie1;
    const m = 12;
    var i, j, l, p: indeks;
        x, w: obiekt;
        s: 0..m;
        stos: array [1..m] of
                record l, p: indeks end;
begin s := 1; stos[1].l := 1; stos[1].p := n;
    repeat {weź żądanie z wierzchołka stosu}
        l := stos[s].l; p := stos[s].p; s := s-1;
        repeat {dziel a[l]...a[p]}
            i := l; j := p; x := a[(l+p) div 2];
            repeat
                while a[i].klucz < x.klucz do i := i+1;
                while x.klucz < a[j].klucz do j := j-1;
                if i ≤ j then
                begin w := a[i]; a[i] := a[j]; a[j] := w;
                    i=i+1; j:=j-1
                end
            until i > j;
            if i < p then
            begin {żądanie posortowania prawej części}
                s := s+1; stos[s].l := i; stos[s].p := p
            end;
            p := j
        until l ≥ p
    until s = 0
end {sortowanieszybkie1}
```

Analiza metody sortowania szybkiego. W celu zanalizowania efektywności sortowania szybkiego musimy najpierw zbadać przebieg procesu podziału. Po wybraniu ograniczenia *x* posuwamy się przez całą tablicę. Dokonuje się więc

dokładnie n porównań. Liczbę wymian można wyznaczyć, stosując następujące rozumowanie probabilistyczne.

Załóżmy, że zbiór danych do podziału składa się z n kluczy $1 \ldots n$ oraz że wybraliśmy ograniczenie x. Po zakończeniu podziału obiekt x będzie zajmował x-te miejsce tablicy. Potrzebna liczba wymian jest równa liczbie elementów lewego podziału $(x-1)$ razy prawdopodobieństwo zamiany klucza. Klucz jest zamieniany, jeżeli jest nie mniejszy niż ograniczenie x. Prawdopodobieńswo to wynosi $(n-x+1)/n$. Spodziewaną liczbę wymian otrzymujemy, sumując po wszystkich możliwych wyborach ograniczenia i dzieląc tak otrzymaną sumę przez n.

$$Pr = \frac{1}{n}\sum_{x=1}^{n}\frac{n-x}{n}(n-x+1) = \frac{n}{6} - \frac{1}{6n} \tag{2.15}$$

Stąd spodziewana liczba wymian wynosi około $n/6$.

Jeżeli przyjmiemy, że będziemy mieli dużo szczęścia i zawsze jako ograniczenie uda nam się wybrać medianę, to każdy proces podziału rozbije tablicę na dwie jednakowe części, a wówczas potrzebna do posortowania liczba przejść będzie równa $\log n$. Wtedy całkowita liczba porównań będzie wynosić $n \cdot \log n$, całkowita zaś liczba wymian elementów $n/6 \cdot \log n$. Oczywiście nie można liczyć na to, że zawsze wybierzemy medianę. W rzeczywistości szansa natrafienia na nią wynosi tylko $1/n$. Zadziwiające jest jednakże, że w przypadku losowego wyboru ograniczenia średnia efektywność sortowania szybkiego jest gorsza od optymalnej tylko $2 \cdot \ln 2$ razy.

Jednakże i metoda sortowania szybkiego ma swoje słabe strony. Przede wszystkim, tak jak i wszystkie nowoczesne metody, niezbyt dobrze działa dla małych wartości n. Jej przewaga nad pozostałymi metodami polega na możliwości łatwego wstawiania prostej metody sortowania działającej na małych podziałach. Jest to szczególnie korzystne dla rekurencyjnej wersji programu.

Pozostała nam jeszcze do omówienia kwestia najgorszego przypadku. Jak wtedy działa metoda sortowania szybkiego? Odpowiedź, niestety, rozczarowuje i odkrywa jedną ze słabości sortowania szybkiego, które w tych przypadkach staje się sortowaniem powolnym. Rozważmy na przykład nieszczęśliwy przypadek wyboru za każdym razem największej wartości z podziału jako ograniczenia x. Wtedy każdy krok dzieli segment n obiektów na lewą część składającą się z $n-1$ obiektów oraz prawą część składającą się z pojedynczego obiektu. W rezultacie potrzeba dokonać n, a nie tylko $\log n$ podziałów i stąd efektywność w przypadku najgorszym jest rzędu n^2.

Jak widać, decydującym krokiem jest wybór ograniczenia x. W naszym przykładowym programie wybierany był obiekt środkowy. Zauważmy, że równie dobrze można wybierać albo pierwszy, albo ostatni element $a[l]$ lub $a[p]$. Przypadkiem najgorszym jest wówczas tablica początkowo uporządkowana; sortowanie szybkie wykazuje zatem zdecydowaną „niechęć” do zadań trywialnych i preferuje tablice nieuporządkowane. Przy wyborze obiektu środkowego ta

dziwna właściwość sortowania szybkiego jest mniej widoczna, ponieważ początkowo posortowana tablica będzie przypadkiem optymalnym! W rzeczywistości przeciętna efektywność jest nieco lepsza przy wyborze obiektu środkowego. Hoare proponuje, aby wyboru x dokonywać „losowo" lub wybierać medianę z małej próbki, powiedzmy trzech kluczy [2.12] i [2.13]. Tego typu racjonalny wybór prawie zupełnie nie wpływa na średnią wydajność metody sortowania szybkiego, lecz poprawia znacząco jej wydajność w najgorszych przypadkach. Z powyższych rozważań wynika, że sortowanie oparte na metodzie sortowania szybkiego jest pewnego rodzaju loterią, na której można dużo stracić, jeżeli nie będzie się miało szczęścia.

Bardzo ważnych rzeczy można się nauczyć na podstawie opisanego doświadczenia; dotyczą one bezpośrednio programisty. Jakie będą skutki opisanego powyżej wykonania programu 2.11 w najgorszym przypadku? Doszliśmy do wniosku, że każdy podział da w wyniku prawy segment składający się z jednego tylko obiektu; żądanie posortowania tego segmentu jest odkładane na stos w celu późniejszego wykonania. Dlatego największa liczba żądań, a więc całkowita potrzebna długość stosu wynosi n. Taka sytuacja jest nie do przyjęcia. (Zauważmy, że w nie lepszej sytuacji – a tak naprawdę to nawet w gorszej – jest wersja rekurencyjna, ponieważ system pozwalający na rekurencyjne wywołania procedur będzie musiał automatycznie przechowywać wartości zmiennych lokalnych i parametrów wszystkich wywołanych procedur oraz używać do tego celu niejawnego stosu). Można temu zaradzić, przechowując na stosie żądanie posortowania dłuższego segmentu i kontynuując dalej dzielenie krótszych części. W takim przypadku wielkość stosu m można ograniczyć do $m = \log_2 n$.

Zmiana, której trzeba dokonać w programie 2.11, ogranicza się do części zapamiętującej nowe żądanie. Teraz ten fragment wygląda następująco:

```
if j-l<p-i then
begin if i<p then
    begin {zapamiętaj na stosie żądanie podziału prawej części}
        s:= s+1; stos[s].l:= i; stos[s].p:= p
    end;
    p:= j {sortuj lewą część}
end else
begin if l<j then
    begin {zapamiętaj na stosie żądanie podziału lewej części}
        s:= s+1; stos[s].l:= l; stos[s].p:= j
    end;
    l:= i {sortuj prawą część}
end
```

2.2.7. Znajdowanie mediany

Mediana n obiektów jest zdefiniowana jako obiekt mniejszy od połowy (lub równy połowie) n obiektów oraz większy (lub równy) od drugiej połowy n obiektów. Na przykład medianą elementów

16 12 99 95 18 87 10

jest 18.

Zagadnienie znajdowania mediany bywa tradycyjnie łączone z sortowaniem, ponieważ nieomylnym sposobem określania mediany jest posortowanie zbioru n obiektów, a następnie wybranie obiektu środkowego. Jednakże potencjalnie o wiele szybszą metodę znajdowania mediany dają nam podziały robione przez program 2.9. Opiszemy metodę, którą łatwo daje się uogólnić do zagadnienia znajdowania k-tego najmniejszego z n elementów zbioru. Znajdowanie mediany będzie przypadkiem specjalnym dla $k=n/2$.

Algorytm wynaleziony przez C.A.R. Hoare'a [2.4] działa następująco. Najpierw stosuje się operację podziału z metody sortowania szybkiego dla $l=1$ i $p=n$ oraz z $a[k]$ wybranym jako wartość dzieląca (granica) x. Otrzymane w wyniku wartości indeksów i oraz j są takie, że

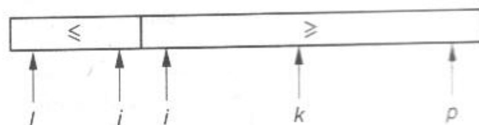
(1) $a[h] \leqslant x$ dla wszystkich $h<i$

(2) $a[h] \geqslant x$ dla wszystkich $h>j$ (2.17)

(3) $i>j$

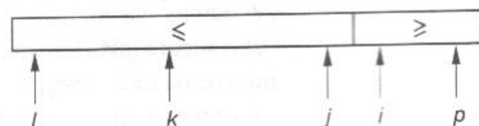
Mogą wystąpić trzy przypadki:

(1) Wartość dzieląca x okazała się za mała; w wyniku tego granica między dwoma częściami przebiega poniżej pożądanej wartości k. Proces podziału musi być powtórzony dla elementów $a[i] \dots a[p]$ (zob. rys. 2.9).



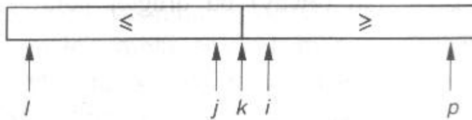
RYSUNEK 2.9
Za mała granica

(2) Wybrana granica x okazała się za duża; operacja dzielenia musi być powtórzona dla przedziału $a[l] \dots a[j]$ (zob. rys. 2.10).



RYSUNEK 2.10
Za duża granica

(3) $j<k<i$: element $a[k]$ dzieli tablicę na dwa przedziały w zadanej proporcji i dlatego jest już szukaną wielkością (zob. rys. 2.11).



RYSUNEK 2.11
Granica właściwa

Proces dzielenia musi być powtarzany dopóty, dopóki nie wystąpi przypadek 3. Iterację tę opisano w następującym fragmencie programu:

```
l := 1; p := n;
while l<p do
   begin x := a[k];
      podział (a[l] ... a[p]);                                  (2.18)
      if j<k then l := i;
      if k<i then p := j
   end
```

Polecamy czytelnikowi oryginalny artykuł Hoare'a, w którym znajduje się formalny dowód poprawności tego algorytmu. Cały program *znajdź* można w prosty sposób z niego wyprowadzić.

PROGRAM 2.12
Znajdź *k*-ty element

```
procedure znajdź(k: integer);
   var l, p, i, j, w, x: integer;
begin l := 1; p := n;
   while l<p do
   begin x := a[k]; i := l; j := p;
      repeat {dziel}
         while a[i] < x do i := i+1;
         while x < a[j] do j := j-1;
         if i≤j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
               i := i+1; j := j-1
            end
      until i>j;
      if j<k then l := i;
      if k<i then p := j
   end
end {znajdź}
```

Jeżeli założymy, że średnio każde dzielenie przepoławia długość podziału, w którym znajduje się pożądana wielkość, to liczba potrzebnych porównań wynosi

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \doteq 2n \tag{2.19}$$

tzn. jest ona rzędu n. Pokazaliśmy w ten sposób, jak wydajny jest program *znajdź* w przypadku szukania mediany i analogicznych wielkości, oraz wykazaliśmy jego przewagę nad bezpośrednimi metodami sortowania całego zbioru kandydatów w celu wybrania k-tego (najlepsze z nich są rzędu $n \cdot \log n$). Jednakże w najgorszym przypadku każdy krok dzielenia zmniejsza wielkość zbioru kandydatów o 1, co prowadzi do liczby potrzebnych porównań rzędu n^2. I tym razem stosowanie tego algorytmu nie przynosi żadnych korzyści, jeżeli liczba elementów jest mała, powiedzmy mniejsza niż 10.

2.2.8. Porównanie metod sortowania tablic

Aby podsumować powyższy przegląd metod sortowania, spróbujemy porównać ich efektywność. Jeżeli n będzie oznaczać liczbę obiektów, które należy posortować, to Po i Pr będą nadal oznaczać, odpowiednio, liczbę potrzebnych porównań kluczy i przesunięć obiektów. Ścisłe wzory analityczne można podać dla wszystkich trzech prostych metod sortowania. Są one zestawione w tabl. 2.8. Nazwy kolumn: min, max, śr określają, odpowiednio, wartości najmniejsze, największe oraz spodziewane wartości uśrednione dla wszystkich $n!$ permutacji n obiektów.

TABLICA 2.8
Porównanie prostych metod sortowania

	Min	Śr	Max
Proste wstawianie	$Po = n-1$ $Pr = 2(n-1)$	$(n^2+n-2)/4$ $(n^2-9n-10)/4$	$(n^2-n)/2-1$ $(n^2+3n-4)/2$
Proste wybieranie	$Po = (n^2-n)/2$ $Pr = 3(n-1)$	$(n^2-n)/2$ $n(\ln n + 0{,}57)$	$(n^2-n)/2$ $n^2/4 + 3(n-1)$
Prosta zamiana (sortowanie bąbelkowe)	$Po = (n^2-n)/2$ $Pr = 0$	$(n^2-n)/2$ $(n^2-n)*0{,}75$	$(n^2-n)/2$ $(n^2-n)*1{,}5$

Nie są znane wystarczająco proste, dokładne wzory dla nowoczesnych metod. Najważniejsze jest to, że potrzebna praca obliczeniowa jest równa $c_1 n^{1,2}$ w przypadku sortowania metodą Shella oraz $c_i n \cdot \log n$ w przypadkach sortowania przez kopcowanie i sortowania szybkiego.

TABLICA 2.9
Czasy wykonania programów sortowania

	Uporządkowany		Losowy		Odwrotnie uporządkowany	
Wstawianie proste	12	23	366	1444	704	2836
Wstawianie połówkowe	56	125	373	1327	662	2490
Wybieranie proste	489	1907	509	1956	695	2675
Sortowanie bąbelkowe	540	2165	1026	4054	1492	5931
Sortowanie bąbelkowe ze znacznikiem	5	8	1104	4270	1645	6542
Sortowanie mieszane	5	9	961	3642	1619	6520
Sortowanie metodą Shella	58	116	127	349	157	492
Sortowanie przez kopcowanie	116	253	110	241	104	226
Sortowanie szybkie	31	69	60	146	37	79
Sortowanie przez łączenie*	99	234	102	242	99	232

* Zob. p. 2.3.1

Wzory te dają zgrubną ocenę efektywności jako funkcji n oraz umożliwiają podział algorytmów sortowania na metody prymitywne lub proste (n^2) oraz nowoczesne lub „logarytmiczne" ($n \cdot \log n$). Jednakże ze względów praktycznych dobrze jest móc dysponować danymi eksperymentalnymi, które umożliwiłyby ocenę wielkości współczynnika c_i, pozwalającą na dalsze rozróżnienie metod sortowania. Co więcej, w powyższych wzorach nie uwzględniono pracy obliczeniowej poświęconej na wykonywanie operacji innych niż porównania kluczy i przesunięcia elementów, takich jak sterowanie wykonaniem pętli itp. Jasne, że takie czynniki w pewnym stopniu zależą od konkretnego systemu, ale i tak wyniki otrzymane eksperymentalnie są bardzo pouczające. W tablicy 2.9 przedstawiono czasy (w milisekundach) działania poprzednio omówionych metod zaprogramowanych w języku Pascal i wykonanych przez maszynę cyfrową CDC 6400. Trzy kolumny zawierają czasy sortowania już uporządkowanego ciągu, losowo wybranej permutacji oraz tablicy uporządkowanej odwrotnie. Lewa strona każdej kolumny dotyczy 256 obiektów, prawa 512 obiektów. Przedstawione wyniki pozwalają wyraźnie odróżnić metody rzędu n^2 od metod $n \cdot \log n$. Godne uwagi są następujące wnioski.

(1) Usprawnienie wstawiania połówkowego w stosunku do prostego wstawiania nie ma faktycznie żadnego znaczenia, a nawet wpływa ujemnie w przypadku istniejącego już porządku.

(2) Sortowanie bąbelkowe jest zdecydowanie najgorszą ze wszystkich porównywanych metod. Jej ulepszona wersja – sortowanie mieszane – jest nadal gorsza od metod prostego wstawiania i prostego wyboru oprócz patologicznego przypadku sortowania już posortowanej tablicy.

(3) Sortowanie szybkie jest lepsze od sortowania przez kopcowanie 2 do 3 razy. Tablicę uporządkowaną odwrotnie sortuje się tą metodą z szybkością praktycznie taką samą jak tablicę już posortowaną.

TABLICA 2.10
Czasy wykonania programów sortowania (klucze wraz ze związanymi z nimi danymi)

	Uporządkowany		Losowy		Odwrotnie uporządkowany	
Wstawianie proste	12	46	366	1129	704	2150
Wstawianie połówkowe	46	76	373	1105	662	2070
Wybieranie proste	489	547	509	607	695	1430
Sortowanie bąbelkowe	540	610	1026	3212	1492	5599
Sortowanie bąbelkowe ze znacznikiem	5	5	1104	3237	1645	5762
Sortowanie mieszane	5	5	961	3071	1619	5757
Sortowanie metodą Shella	58	186	127	373	157	435
Sortowanie przez kopcowanie	116	264	110	246	104	227
Sortowanie szybkie	31	55	60	137	37	75
Sortowanie przez łączenie*	99	196	102	195	99	187

* Zob. p. 2.3.1

Trzeba nadmienić, że wyniki uzyskano, sortując obiekty składające się tylko z klucza, bez towarzyszących danych. Nie jest to założenie zbyt realistyczne; w tablicy 2.10 pokazano, jaki wpływ na wyniki ma zwiększenie wielkości obiektów. W badanym przykładzie dane zajmowały 7 razy tyle pamięci co klucz. W tablicy 2.10 lewa strona każdej kolumny zawiera czas potrzebny do posortowania kluczy bez towarzyszących danych; prawa strona kolumny odpowiada sortowaniu z towarzyszącymi danymi; $n = 256$. Należy zwrócić uwagę na następujące wnioski:

(1) Metoda prostego wyboru znacznie zyskała i wyszła na prowadzenie wśród prostych metod.
(2) Metoda sortowania bąbelkowego jest dalej zdecydowanie najgorsza (straciła całkowicie znaczenie!) i tylko jej „ulepszenie" zwane sortowaniem mieszanym jest od niej nieco gorsze w przypadku uporządkowania odwrotnego.
(3) Metoda sortowania szybkiego umocniła nawet swoją pozycję najszybszej metody i jest najlepszą ze znanych obecnie metod sortowania tablic.

2.3. Sortowanie plików sekwencyjnych

2.3.1. Łączenie proste

Przedstawionych w poprzednim punkcie algorytmów sortowania nie można, niestety, stosować wtedy, gdy dane przeznaczone do posortowania nie mieszczą się w głównej pamięci maszyny cyfrowej i są przechowywane np. w peryferyjnych pamięciach sekwencyjnych, takich jak taśmy. W tym przypadku dane

opiszemy jako plik (sekwencyjny), którego zasadniczą cechą jest to, że w każdej chwili jest dostępny bezpośrednio jeden i tylko jeden jego składnik. Jest to poważne ograniczenie w porównaniu z możliwościami oferowanymi przez strukturę tablicy i dlatego też trzeba stosować zupełnie odmienne metody sortowania. Jedną z najważniejszych metod jest **sortowanie przez łączenie** (scalanie; ang. *merge sort*). Łączenie (lub scalanie) oznacza wiązanie dwóch (lub więcej) ciągów uporządkowanych w jeden ciąg uporządkowany przez wybieranie kolejnych obiektów spośród aktualnie dostępnych elementów tych ciągów. Łączenie jest o wiele prostsze od sortowania i jest stosowane jako operacja pomocnicza w bardziej złożonym procesie sortowania sekwencyjnego. Jednym ze sposobów sortowania opartym na łączeniu jest tzw. **łączenie proste**, które przebiega według następującego schematu:

(1) Dzielimy ciąg *a* na połowy, nazwane *b* i *c*.
(2) Łączymy ciągi *b* i *c*, składając pojedyncze ich elementy w pary uporządkowane.
(3) Nazywamy połączony ciąg *a* i powtarzamy kroki 1 i 2, łącząc tym razem pary uporządkowane w czwórki uporządkowane.
(4) Powtarzamy poprzednie kroki, łącząc czwórki w ósemki, i dalej postępujemy w ten sam sposób, podwajając za każdym razem długość łączonych podciągów dopóty, dopóki cały ciąg nie zostanie uporządkowany.

Jako przykład rozważmy ciąg

44 55 12 42 94 18 06 67

W kroku 1 otrzymamy po podzieleniu ciągi

44 55 12 42
94 18 06 67

Po połączeniu pojedynczych elementów (które są ciągami uporządkowanymi o długości 1) w pary uporządkowane otrzymujemy

44 94 ' 18 55 ' 06 12 ' 42 67

Po ponownym podzieleniu w środku i połączeniu par uporządkowanych otrzymujemy

06 12 44 94 ' 18 42 55 67

Po wykonaniu trzeciej operacji dzielenia i łączenia otrzymujemy w końcu pożądany wynik

06 12 18 42 44 55 67 94

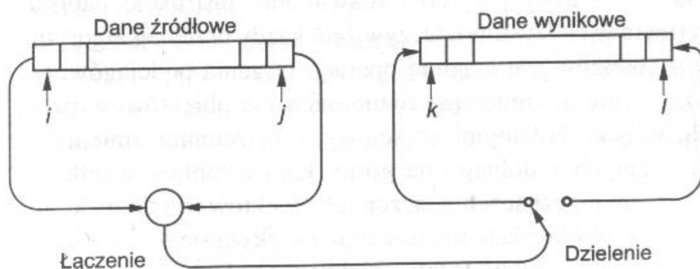
Każda operacja dotycząca całego zbioru danych nazywa się **fazą**, a najmniejszy podproces, którego kilkakrotne powtórzenie tworzy proces sortowania, nazywa się **przebiegiem** lub **etapem**. W powyższym przykładzie sortowanie

składa się z trzech przebiegów, a każdy przebieg składa się z fazy dzielenia i fazy łączenia. Aby wykonać sortowanie, są niezbędne trzy taśmy, dlatego ten proces nazywany jest **łączeniem trójtaśmowym** (ang. *three-tape merge*).

W obecnej postaci fazy dzielenia nie przyczyniają się do sortowania, ponieważ nie permutują w żaden sposób elementów; w tym sensie są bezproduktywne, chociaż stanowią połowę wszystkich operacji kopiowania danych. Fazy dzielenia można całkowicie usunąć, wiążąc je z fazami łączenia. Zamiast łączenia w jeden ciąg dane wyjściowe procesu łączenia są natychmiast rozdzielane na dwie taśmy, które będą wejściem dla następnego przebiegu. W przeciwieństwie do poprzedniego dwufazowego sortowania przez łączenie metoda ta nazywa się **łączeniem jednofazowym** lub **łączeniem wyważonym** (zrównoważonym; ang. *balanced merge*). Jest ona oczywiście lepsza, ponieważ wymaga o połowę mniej operacji kopiowania; ceną płaconą za tę zaletę jest czwarta taśma.

Omówimy teraz dokładnie, jak się tworzy program łączenia. Początkowo niech dane znajdują się w tablicy, która będzie jednakże przeglądana w sposób *całkowicie sekwencyjny*. Druga wersja programu sortowania przez łączenie będzie oparta na strukturze plikowej danych; pozwala to na porównanie dwóch programów i wykazanie, jak bardzo postać programu zależy od przyjętej reprezentacji danych.

Można łatwo użyć jednej tablicy zamiast dwóch plików, jeśli tylko będziemy rozważać tablicę jako ciąg z dwoma końcami. Zamiast łączenia dwóch plików źródłowych możemy pobierać obiekty z dwóch końców tablicy. Tak więc ogólną postać połączonej fazy dzielenia i łączenia można pokazać tak jak na rys. 2.12. Miejsce przeznaczenia łączonych obiektów zmieniamy po każdej parze uporządkowanej w pierwszym przebiegu, po każdej czwórce uporządkowanej w drugim przebiegu itd.; wypełniamy w ten sposób równomiernie oba ciągi wynikowe reprezentowane przez dwa końce jednej tablicy. Po każdym przebiegu tablice zamieniają się rolami: źródłowa staje się wynikową i odwrotnie.



RYSUNEK 2.12
Sortowanie przez łączenie proste z dwiema tablicami

Dalsze uproszczenie programu można osiągnąć, łącząc dwie pojęciowo różne tablice w jedną tablicę o podwójnej długości. Tak więc dane będą reprezentowane przez tablicę

a: **array** [1..2*n] **of** *obiekt* (2.20)

Indeksy i oraz j będą określać dwa źródłowe obiekty, podczas gdy k oraz l będą wyznaczać miejsca przeznaczenia danych wynikowych (zob. rys. 2.12). Danymi początkowymi będą oczywiście obiekty $a_1 \dots a_n$. Potrzebna jest zmienna boolowska *góra* określająca kierunek przepływu danych; *góra* = *true* będzie oznaczać, że przetwarzane aktualnie dane $a_1 \dots a_n$ będą przesuwane „w górę" na miejsce danych $a_{n+1} \dots a_{2n}$, natomiast *góra* = *false* będzie oznaczać, że dane $a_{n+1} \dots a_{2n}$ będą przenoszone „na dół" – na $a_1 \dots a_n$. Zmienna *góra* zmienia wartość po każdym kolejnym przebiegu. I wreszcie wprowadza się zmienną p, określającą długość łączonych podciągów. Jej wartość wynosi początkowo 1 i jest podwajana przed każdym następnym przebiegiem. Aby zagadnienie w pewnym stopniu uprościć, założymy, że n jest zawsze potęgą liczby 2. Tak więc pierwsza wersja programu łączenia prostego przybiera następującą postać:

```
procedure sortowanieprzezłączenie;
   var i, j, k, l: indeks;
       góra: Boolean; p: integer;
begin góra := true; p := 1;
   repeat {inicjuj indeksy}
      if góra then
         begin i := 1; j := n; k := n+1; l := 2*n
         end else
         begin k := 1; l := n; i := n+1; j := 2*n                (2.21)
         end;
      „łącz p-tki z i-ciągu oraz j-ciągu do
      k-ciągu oraz l-ciągu";
      góra := ¬ góra; p := 2*p
   until p=n
end
```

W następnym kroku budowy programu rozwiniemy instrukcję napisaną w języku naturalnym (ujętą w cudzysłów). Oczywiście każdy przebieg algorytmu łączenia obejmujący n obiektów jest ciągiem operacji łączenia podciągów, tzn. łączenia p-tek. Aby zapewnić równomierne rozmieszczenie obiektów w dwóch ciągach wynikowych, między kolejnymi częściowymi łączeniami zmienia się miejsce przeznaczenia danych z dolnego na górny koniec tablicy wynikowej i odwrotnie. Jeżeli miejscem przeznaczenia łączonych obiektów jest dolny koniec tablicy wynikowej, to indeks k określa to miejsce oraz zwiększa się o 1 po każdym przesunięciu jednego elementu ciągu. Jeżeli obiekty mają być przesunięte na górny koniec tablicy wynikowej, to indeks l określa miejsce ich przeznaczenia oraz zmniejsza się o 1 po każdym przesunięciu. W celu uproszczenia opisu aktualnego stanu łączenia będziemy zawsze określać miejsce przeznaczenia indeksem k, zamieniając wartości zmiennych k i l po połączeniu każdej p-tki, a używany za każdym razem przyrost będziemy określać zmienną h, przy czym h będzie równe 1 albo -1. Powyższe rozważania doprowadziły nas do utworzenia następującego fragmentu programu:

```
h := 1; m := n; {m=liczba obiektów do połączenia}
repeat q := p; r := p; m := m−2*p;
        „połącz q obiektów z i oraz r obiektów z j,
        k jest indeksem ciągu wynikowego z przyrostem h"      (2.22)
    h := −h;
    „zamień k z l"
until m=0
```

W kolejnym kroku precyzowania programu należy sformułować instrukcje opisujące właściwe łączenie. Musimy w tym miejscu pamiętać, że koniec podciągu, który po łączeniu pozostał niepusty, ma być dołączony do ciągu wyjściowego za pomocą zwykłej operacji kopiowania.

```
while (q≠0) ∧ (r≠0) do
begin {wybierz obiekt z i lub z j}
    if a[i].klucz < a[j].klucz then
        begin „przesuń obiekt z i do k, zwiększ i oraz k"; q := q−1      (2.23)
        end else
        begin „przesuń obiekt z j do k, zwiększ j oraz k"; r := r−1
        end
end;
„skopiuj koniec ciągu i";
„skopiuj koniec ciągu j"
```

Po zaprogramowaniu operacji kopiowania końców ciągów program będzie całkowicie ukończony. Przed napisaniem go w całości postaramy się wyeliminować ograniczenie, że n ma być potęgą liczby 2. Które fragmenty algorytmu ulegną modyfikacji po usunięciu tego ograniczenia? Łatwo można się przekonać, że najprościej poradzić sobie z sytuacją bardziej ogólną, postępując tak długo, jak tylko to jest możliwe, według pierwotnego schematu. W tym przypadku znaczy to tyle, że będziemy dopóty łączyć p-tki, dopóki resztki pozostałe na ciągach źródłowych nie będą krótsze niż p. Jedynym fragmentem programu, który ulegnie zmianie, będą instrukcje wyznaczające wartości q oraz r, określające długości ciągów do połączenia. Podane poniżej cztery instrukcje zastępują następujące trzy instrukcje:

```
q := p; r := p; m := m−2*p
```

oraz, o czym czytelnik powinien się sam przekonać, stanowią efektywną realizację opisanej powyżej strategii; zauważmy, że m określa ogólną liczbę obiektów dwóch ciągów źródłowych, które pozostały jeszcze do połączenia:

```
if m≥p then q := p else q := m; m := m−q;
if m≥p then r := p else r := m; m := m−r;
```

Ponadto, w celu zapewnienia zakończenia wykonania programu, warunek $p=n$ sterujący „zewnętrzną" iteracją musi być zmieniony na $p \geq n$. Po tych mody-

fikacjach możemy przejść do opisu całego algorytmu w postaci pełnego programu (zob. program 2.13).

PROGRAM 2.13
Sortowanie przez łączenie proste

```
procedure sortowanieprzezłączenie;
   var i, j, k, l, t: indeks;
      h, m, p, q, r: integer; góra: Boolean;
      {zauważ, że a ma indeksy 1 ... 2*n}
begin góra := true; p := 1;
   repeat h := 1; m := n;
      if góra then
      begin i := 1; j := n; k := n+1; l := 2*n
      end else
      begin k := 1; l := n; i := n+1; j := 2*n
      end;
      repeat {połącz ciągi z i oraz z j do k}
         {q=długość ciągu i, r=długość ciągu j}
         if m≥p then q := p else q := m; m := m−q;
         if m≥p then r := p else r := m; m := m−r;
         while (q≠0) ∧ (r≠0) do
         begin {połącz}
            if a[i].klucz < a[j].klucz then
            begin a[k] := a[i]; k := k+h; i := i+1; q := q−1
            end else
            begin a[k] := a[j]; k := k+h; j := j−1; r := r−1
            end
         end;
         {kopiuj koniec ciągu j}
         while r≠0 do
            begin a[k] := a[j]; k := k+h; j := j−1; r := r−1
            end;
         {kopiuj koniec ciągu i}
         while q≠0 do
            begin a[k] := a[i]; k := k+h; i := i+1; q := q−1
            end;
         h := −h; t := k; k := l; l := t
      until m=0;
      góra := ¬ góra; p := 2*p
   until p≥n;
   if ¬ góra then
   for i := 1 to n do a[i] := a[i+n]
end {sortowanieprzezłączenie}
```

Analiza sortowania przez łączenie. Ponieważ każdy przebieg podwaja wartość p oraz ponieważ sortowanie kończy się wtedy, gdy $p \geqslant n$, wymaga ono $\lceil \log_2 n \rceil$ przebiegów. Z definicji, każdy przebieg kopiuje dokładnie raz cały zbiór obiektów, stąd całkowita liczba przesunięć wynosi dokładnie

$$Pr = n\lceil \log n \rceil \tag{2.24}$$

Liczba porównań kluczy Po jest nawet mniejsza niż Pr, ponieważ porównania nie występują w operacjach kopiowania końców ciągów. Jednakże, ponieważ metoda sortowania przez łączenie bywa zazwyczaj stosowana przy użyciu pamięci zewnętrznych, pracochłonność opercji przesunięcia jednego elementu ciągu jest o kilka rzędów wielkości większa od pracochłonności operacji porównania. Dlatego też szczegółowa analiza liczby porównań ma małe znaczenie praktyczne.

Algorytm sortowania przez łączenie wytrzymuje pomyślne próby porównania nawet z nowoczesnymi metodami sortowania, omawianymi w poprzednim rozdziale. Jednakże stosunkowo wysoki jest koszt manipulowania indeksami oraz, co jest wadą decydującą, potrzebna jest pamięć dla $2n$ obiektów. To jest właśnie przyczyną, że sortowanie przez łączenie bardzo rzadko stosuje się do tablic, tj. do danych umieszczonych w pamięci głównej. Wyniki porównania rzeczywistych czasów wykonania algorytmu sortowania przez łączenie zamieszczono w ostatnich wierszach tabl. 2.9 i 2.10. Są one lepsze niż wyniki sortowania przez kopcowanie, ale gorsze od sortowania szybkiego.

2.3.2. Łączenie naturalne

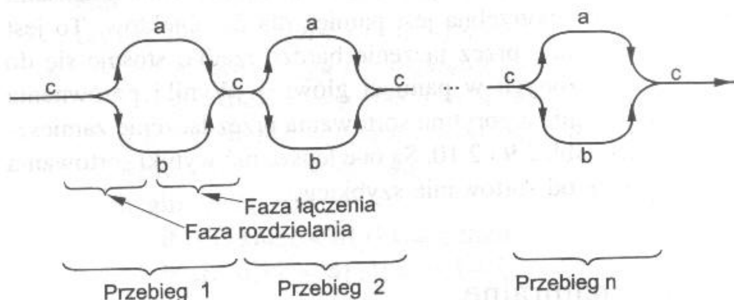
W metodzie łączenia prostego nie korzysta się z ewentualnego, częściowego uporządkowania danych źródłowych. Długość wszystkich podciągów łączonych w k-tym przebiegu jest równa lub mniejsza od 2^k; tzn. nie korzysta z faktu, że w k-tym przebiegu mogą występować uporządkowane podciągi o długości większej od 2^k oraz że mogą być łączone te dłuższe podciągi. W rzeczywistości każde dwa uporządkowane podciągi o długościach m i n mogą być bezpośrednio połączone w jeden podciąg $(m+n)$-elementowy. Sortowanie przez łączenie, które zawsze scala dwa możliwie najdłuższe podciągi, nazywa się **sortowaniem przez łączenie naturalne.**

Uporządkowany podciąg często nazywany jest **napisem**. Jednakże, ponieważ słowa napis o wiele częściej używa się jako nazwy ciągu znaków, wprowadzimy (za Knuthem) do naszej terminologii słowo *seria* zamiast *napis* i będziemy go używać wtedy, gdy mówimy o uporządkowanych podciągach. Ciąg $a_i \ldots a_j$ taki, że

$$\begin{aligned} &a_k \leqslant a_{k+1} \qquad \text{dla } k = i \ldots j-1 \\ &a_{i-1} > a_i \\ &a_j > a_{j+1} \end{aligned} \tag{2.25}$$

będziemy nazywać **największą serią** lub krótko **serią**. Metoda sortowania przez łączenie naturalne łączy więc (największe) serie, a nie ciągi o stałej, wcześniej określonej długości. Serie mają tę własność, że jeżeli łączy się dwa ciągi n serii, to powstaje jeden ciąg dokładnie n serii. Stąd całkowita liczba serii zmniejsza się po każdym przejściu o połowę, a więc potrzebna liczba przesunięć obiektów wynosi w najgorszym przypadku $n \cdot \lceil \log_2 n \rceil$, chociaż przeciętnie jest mniejsza. Oczekiwana liczba porównań będzie jednakże dużo większa, ponieważ oprócz porównań potrzebnych do wyboru obiektów konieczne są porównania sąsiadujących obiektów każdego pliku w celu określenia końców serii.

W naszym następnym ćwiczeniu z programowania skonstruujemy algorytm łączenia naturalnego zgodnie z tą samą metodą stopniowego precyzowania, z której już korzystaliśmy przy tworzeniu algorytmu łączenia prostego. Tym razem zastosujemy strukturę pliku sekwencyjnego zamiast tablicy; w wyniku otrzymamy niewyważone (niezrównoważone), dwufazowe, trójtaśmowe sortowanie przez łączenie. Zakładamy, że ciąg źródłowy jest zadany w pliku c, w którym znajdzie się też później posortowany ciąg wynikowy. (Naturalnie,



RYSUNEK 2.13
Fazy sortowania i przebiegi sortowania

w konkretnych zastosowaniach z dziedziny przetwarzania danych, dane źródłowe muszą być ze względów bezpieczeństwa najpierw skopiowane z oryginalnej taśmy do pliku c). Dwoma taśmami pomocniczymi są a i b. Każdy przebieg składa się z fazy rozdzielania, w której serie z pliku c są równomiernie rozdzielane na a i b, oraz z fazy łączenia, w której serie z a i b są łączone w c. Proces ten pokazano na rys. 2.13.

Jako przykład w tabl. 2.11 przedstawiono plik c w pierwotnym stanie (wiersz 1) oraz po kolejnych przebiegach (wiersze 2–4) sortowania przez łączenie naturalne 20 liczb. Zauważmy, że potrzebne były tylko trzy przebiegi. Sorto-

TABLICA 2.11
Przykład sortowania przez łączenie naturalne

17	31′	5	59′	13	41	43	67′	11	23	29	47′	3	7	71′	2	19	57′	37	61
5	17	31	59′	11	13	23	29	41	43	47	67′	2	3	7	19	57	71′	37	61
5	11	13	17	23	29	31	41	43	47	59	67′	2	3	7	19	37	57	61	71
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

wanie kończy się wtedy, gdy w pliku *c* będzie już tylko jedna seria. (Zakładamy, że początkowo dana jest przynajmniej jedna niepusta seria w pliku).

Niech zmienna *l* służy do zliczania serii łączonych w plik *c*. Jeżeli obiekty globalne zdefiniujemy jako

```
type taśma = file of obiekt;                                   (2.26)
var c: taśma
```

to program nasz można sformułować następująco:

```
procedure łączenienaturalne;
   var l: integer;
      a, b: taśma;
begin                                                          (2.27)
   repeat rewrite(a); rewrite(b); reset(c);
      rozdzielanie;
      reset(a); reset(b); rewrite(c);
      l:= 0; łączenie
   until l = 1
end
```

Dwie fazy są dwiema oddzielnymi instrukcjami. Trzeba je teraz sprecyzować, tzn. dokładniej opisać. Sprecyzowany opis może być albo wstawiony bezpośrednio, albo zapisany w postaci procedur i wtedy te skrócone instrukcje będą wywołaniami procedur. Tym razem wybierzemy drugą metodę i zdefiniujemy

```
procedure rozdzielanie; {z c do a i b}
begin
   repeat kopiujserię(c, a);                                   (2.28)
      if ¬ eof(c) then kopiujserię(c, b)
   until eof(c)
end
```

oraz

```
procedure łączenie; {z a i b do c}
begin
   repeat łączserie; l:= l+1
   until eof(b);                                               (2.29)
   if ¬ eof(a) then
      begin kopiujserię(a, c); l:= l+1
      end
end
```

Powyższa metoda rozdzielania daje przypuszczalnie w wyniku albo równą liczbę serii w plikach *a* i *b*, albo o jedną serię więcej w pliku *a*. Ponieważ odpowiadające sobie pary serii są scalane, może jeszcze pozostać seria w pliku *a*, którą trzeba po prostu skopiować. W sformułowaniach procedur *łączenie* i *rozdzielanie* występują procedury podporządkowane *łączserie* i *kopiujserię*, których funkcji łatwo się można domyślić. Procedury te będą teraz sprecyzowane; wymagają one wprowadzenia globalnej zmiennej logicznej *ks* określającej, czy został osiągnięty koniec serii.

```
procedure kopiujserię(var x, y: taśma);
begin {kopiuj jedną serię z x do y}                              (2.30)
   repeat kopiuj(x, y) until ks
end

procedure łączserie;
begin {łącz serie z a i b w c}
   repeat if a↑.klucz < b↑.klucz then
      begin kopiuj (a, c);
         if ks then kopiujserię(b, c)                            (2.31)
      end else
      begin kopiuj(b, c);
         if ks then kopiujserię(a, c)
      end
   until ks
end
```

Proces porównywania i wyboru kluczy w trakcie łączenia serii kończy się wtedy, gdy tylko zostanie wyczerpana któraś z dwóch serii. Następnie reszta tej serii, która nie została jeszcze wyczerpana, musi być po prostu skopiowana do pliku wynikowego. Służy do tego wywołanie procedury *kopiujserię*.

Powyższe dwie procedury są zdefiniowane za pomocą podporządkowanej procedury *kopiuj*, przenoszącej jeden element pliku źródłowego *x* do pliku wynikowego *y* i sprawdzającej, czy został osiągnięty koniec serii. Procedura *kopiuj* jest już opisana za pomocą instrukcji *read* i *write*. W celu określenia końca serii trzeba pamiętać klucz ostatniego wczytanego (skopiowanego) obiektu, aby można go było porównać z kluczem następnika. To „patrzenie w przód" jest zrealizowane przez sprawdzanie zmiennej buforowej pliku *x*↑.

```
procedure kopiuj(var x, y: taśma);
   var buf: obiekt;                                              (2.32)
begin read(x, buf); write(y, buf);
   if eof(x) then ks := true else ks := buf.klucz > x↑.klucz
end
```

W ten sposób zakończyliśmy konstrukcję procedury sortującej przez łączenie naturalne. Godny ubolewania jest tylko fakt, że – jak to mógł zauważyć uważny czytelnik – program jest niepoprawny. Jest on niepoprawny w tym znaczeniu, że w pewnych przypadkach źle sortuje. Rozważmy na przykład następujący ciąg danych wejściowych:

3 2 5 11 7 13 19 17 23 31 29 37 43 41 47 59 57 61 71 67

Rozdzielając sąsiadujące serie, na przemian do plików *a* i *b*, otrzymujemy

$a =$ 3 ′ 7 13 19 ′ 29 37 43 ′ 57 61 71
$b =$ 2 5 11 ′ 17 23 31 ′ 41 47 59 ′ 67

Ciągi te będą połączone w jedną serię i sortowanie zakończy się pomyślnie. Przykład ten, chociaż nie powoduje niepoprawnego wykonania programu, uświadamia nam, że zwyczajna dystrybucja serii do kilku plików może dać w wyniku mniejszą liczbę serii, niż ich było początkowo. Dzieje się tak dlatego, że pierwszy obiekt $i+2$ serii może być większy niż ostatni obiekt i-tej serii, co powoduje automatyczne połączenie ich w jedną serię.

Pomimo że procedura *rozdzielanie* według założenia rozkłada serię równomiernie do dwóch plików, z powyższych uwag wypływa ważny wniosek, że rzeczywiste liczby serii w plikach *a* i *b* mogą się znacznie różnić. Nasza procedura łączenia będzie jednakże kończyć działanie wtedy, gdy tylko osiągnie koniec pliku *b*, tracąc być może koniec drugiego pliku. Rozważmy następujące dane wejściowe, sortowane (i obcinane) w dwóch kolejnych przebiegach:

TABLICA 2.12
Złe wyniki programu sortowania przez łączenie

17	19	13	57	23	29	11	59	31	37	7	61	41	43	5	67	47	71	2	3
13	17	19	23	29	31	37	41	43	47	57	71	11	59						
11	13	17	19	23	29	31	37	41	43	47	57	59	71						

Przykład tego błędu jest typowy dla wielu sytuacji w programowaniu. Błąd jest skutkiem przeoczenia jednej z możliwych konsekwencji operacji uważanej za nieskomplikowaną. Błąd ten jest typowy także dlatego, że istnieje kilka sposobów jego usunięcia i trzeba jeden z nich wybrać. Często istnieją dwie takie możliwości, różniące się w sposób zasadniczy.

(1) Dochodzimy do wniosku, że operacja rozdzielania jest źle zaprogramowana, ponieważ nie spełnia wymagania, aby liczby serii były równe (lub różniły się co najwyżej o 1). Przerabiamy pierwotny schemat operacji i odpowiednio poprawiamy źle zaprogramowaną procedurę.
(2) Dochodzimy do wniosku, że poprawa wymaga daleko idących modyfikacji i staramy się znaleźć sposób, w jaki można zmienić inne części algorytmu tak, aby je przystosować do niepoprawnych w tej chwili części programu.

Ogólnie, pierwszy sposób wydaje się bezpieczniejszy, jaśniejszy, prostszy i dający większą gwarancję przed późniejszymi konsekwencjami spowodowanymi niezauważonymi, powikłanymi efektami ubocznymi. Z tych przyczyn jest to sposób rozwiązania najczęściej (i słusznie) zalecany.

Trzeba jednakże podkreślić, że nie zawsze można całkowicie odrzucać drugi sposób. Z tego też powodu zbadamy go dokładnie w naszym przykładzie i rozważymy możliwość modyfikacji procedury łączenia, a nie procedury rozdzielania, która pierwotnie była przyczyną błędu.

Działanie takie spowoduje, że schemat rozdzielania pozostanie nietknięty, lecz zrezygnujemy z warunku, aby serie były równomiernie rozdzielone. Może to być przyczyną efektywności mniejszej niż optymalna. Jednakże efektywność w najgorszym przypadku nie zmienia się, a co więcej, przypadek bardzo nierównomiernego rozdzielenia jest statystycznie *mało prawdopodobny*. Względy efektywności nie są więc poważnym argumentem przeciwko temu rozwiązaniu.

Jeżeli nie jest spełniony już dłużej warunek równomiernego rozdzielenia serii, to procedura łączenia musi być tak zmieniona, aby po osiągnięciu końca jednego pliku kopiowała *cały* koniec drugiego pliku, a nie tylko co najwyżej jedną serię.

Zaproponowane zmiany są proste i bardzo łatwe do wprowadzenia w porównaniu z jakąkolwiek zmianą w schemacie rozdzielania. (Czytelnik powinien sam przekonać się o prawdzie tego stwierdzenia). Poprawiona wersja algorytmu łączenia zawarta jest w całości w programie 2.14.

PROGRAM 2.14
Sortowanie przez łączenie naturalne

```
program sortowanieprzezłączenie(input, output);
{3-taśmowe, 2-fazowe sortowanie przez łączenie naturalne}
type obiekt = record klucz: integer
                {inne pola są zdefiniowane w tym miejscu}
              end;
   taśma = file of obiekt;
var c: taśma; n: integer; buf: obiekt;
procedure wydruk(var f: taśma);
var x: obiekt;
begin reset(f);
   while ¬ eof(f) do
      begin read(f, x); write(output, x.klucz)
      end;
   writeln
end {wydruk};
procedure łączenienaturalne;
var l: integer; {liczba połączonych serii}
   ks: boolean; {znacznik końca serii}
   a, b: taśma;
```

```
procedure kopiuj(var x, y: taśma);
   var buf: obiekt;
begin read(x, buf); write(y, buf);
   if eof(x) then ks := true else ks := buf.klucz > x↑.klucz
end;

procedure kopiujserię(var x, y: taśma);
begin {kopiuj jedną serię z x do y}
   repeat kopiuj(x, y) until ks
end;

procedure rozdzielanie;
begin {z c do a i b}
   repeat kopiujserię(c, a);
   if ¬ eof(c) then kopiujserię(c, b)
   until eof(c)
end;

procedure łączserie;
begin {z a i b do c}
   repeat
      if a↑.klucz < b↑.klucz then
      begin kopiuj(a, c);
         if ks then kopiujserię(b, c)
      end else
      begin kopiuj(b, c);
         if ks then kopiujserię(a, c)
      end
   until ks
end;

procedure łączenie;
begin {z a i b do c}
   repeat łączserie; l := l+1
   until eof(a) ∧ eof(b);
   while ¬ eof(a) do
   begin kopiujserię(a, c); l := l+1
   end;
   while ¬ eof(b) do
   begin kopiujserię(b, c); l := l + 1
   end;
   wydruk(c)
end;

begin
   repeat rewrite(a); rewrite(b); reset(c);
      rozdzielanie;
```

```
        reset(a); reset(b); rewrite(c);
        l := 0; łączenie;
    until l=1
end;

begin {program główny; wczytaj ciąg wejściowy zakończony zerem}
    rewrite(c); read(buf.klucz);
    repeat write(c, buf); read(buf.klucz)
    until buf.klucz=0;
    wydruk(c);
    łączenienaturalne;
    wydruk(c)
end.
```

2.3.3. Wielokierunkowe łączenie wyważone

Praca wkładana w sortowanie sekwencyjne jest proporcjonalna do liczby potrzebnych przebiegów, ponieważ – z definicji – w każdym przebiegu kopiuje się cały zbiór danych. Sposobem prowadzącym do zmniejszenia liczby przebiegów jest rozkładanie serii na więcej niż dwie taśmy. Z łączenia r serii równomiernie rozłożonych na N taśmach otrzymuje się r/N serii. Drugi przebieg redukuje liczbę serii do r/N^2, trzeci do r/N^3, a po k przebiegach pozostanie r/N^k serii. Ogólna liczba przebiegów potrzebnych do posortowania n elementów przez **N-kierunkowe łączenie** jest wobec tego równa $k = \lceil \log_N n \rceil$. Ponieważ każdy przebieg wymaga n operacji kopiowania, więc w najgorszym przypadku ogólna liczba operacji kopiowania wynosi

$$Pr = n \cdot \lceil \log_N n \rceil$$

Jako następne ćwiczenie z programowania zbudujemy program sortujący oparty na łączeniu wielokierunkowym. Aby bardziej odróżnić go od poprzedniego dwufazowego łączenia naturalnego, określimy nasz program jako jednofazowe sortowanie przez łączenie wyważone. Wynika stąd, że w każdym przebiegu korzysta się z takiej samej liczby plików wejściowych oraz plików wyjściowych, na które są rozkładane kolejne serie. Dlatego też przy użyciu N plików algorytm będzie oparty na $N/2$-kierunkowym łączeniu przy założeniu, że N jest liczbą parzystą. Postępując według przyjętej poprzednio strategii, nie będziemy się kłopotać wykrywaniem samoczynnego łączenia się dwóch kolejnych serii rozłożonych na tę samą taśmę. W konsekwencji jesteśmy zmuszeni zaprojektować program bez zakładania równej liczby serii na taśmach wejściowych.

W programie tym po raz pierwszy spotykamy się z naturalnym zastosowaniem struktury danych o postaci tablicy plików. W istocie rzeczy jest zadziwiające, jak bardzo ten program różni się od poprzedniego z powodu zmiany

łączenia dwukierunkowego na wielokierunkowe. Po pierwsze, różnice wynikają z sytuacji, że proces łączenia nie może być tak po prostu zakończony po wyczerpaniu się jednego z plików wejściowych. Zamiast tego trzeba przechowywać listę plików wejściowych ciągle jeszcze aktywnych, tzn. jeszcze niewyczerpanych. Druga komplikacja bierze się z potrzeby zamiany po każdym przejściu grup plików wejściowych i wyjściowych.

Rozpoczniemy od zdefiniowania – oprócz wprowadzonych uprzednio typów *obiekt* i *taśma* – typu

$$nrtaśmy = 1..N \tag{2.33}$$

Oczywiście numery taśm służą do indeksowania tablicy plików składających się z obiektów. Załóżmy, że początkowy ciąg obiektów dany jest jako zmienna

$$f0\colon taśma \tag{2.34}$$

oraz że w procesie sortowania używa się N taśm, gdzie N jest parzyste:

$$f\colon \mathbf{array}[nrtaśmy]\ \mathbf{of}\ taśma \tag{2.35}$$

Zalecaną metodą podejścia do zagadnienia zamiany taśm jest wprowadzenie mapy indeksów taśm. Zamiast odwoływać się do taśmy przez indeks i, będziemy się odwoływać przez mapę t, tzn. za każdym razem zamiast

$f[i]$ będziemy pisać $f[t[i]]$

gdzie mapa zdefiniowana jest jako

$$t\colon \mathbf{array}\ [nrtaśmy]\ \mathbf{of}\ nrtaśmy \tag{2.36}$$

Jeżeli początkowo $t[i] = i$ dla wszystkich i, to zamiana taśm będzie polegać na zwykłej zamianie par składników mapy

$t[1] \leftrightarrow t[nh+1]$
$t[2] \leftrightarrow t[nh+2]$
...
$t[nh] \leftrightarrow t[n]$

gdzie $nh = n/2$. Wynika stąd, że zawsze możemy uważać

$f[t[1]]$ ... $f[t[nh]]$

za taśmy wejściowe oraz

$f[t[nh+1]]$ ... $f[t[n]]$

za taśmy wyjściowe. (W następstwie tego w komentarzach $f[t[j]]$ będziemy nazywać po prostu „taśmą j"). Algorytm można teraz sformułować w następującej początkowej postaci:

```
procedure taśmowesortowanieprzezłączenie;
  var i, j: nrtaśmy;
      l: integer; {liczba rozłożonych serii}
      t: array [nrtaśmy] of nrtaśmy;
begin {rozłóż początkowe serie na t[1] ... t[nh]}
  j := nh; l := 0;
  repeat if j < nh then j := j+1 else j := 1;
     „skopiuj jedną serię z f0 na taśmę j";
     l := l+1
  until eof(f0);                                                (2.37)
  for i := 1 to n do t[i] := i;
  repeat {łącz z t[1] ... t[nh] na t[nh+1] ... t[n]}
     „ustaw taśmy wejściowe";
     l := 0;
     j := nh+1; {j=indeks taśmy wyjściowej}
     repeat l := l+1;
           „połącz serie z taśm wejściowych na t[j]";
        if j < n then j := j+1 else j := nh+1
     until „wyczerpane są wszystkie taśmy wejściowe";
     „zamień taśmy"
  until l=1
  {taśmą posortowaną jest t[1]}
end
```

Najpierw sprecyzujemy operację kopiowania używaną w początkowym rozdzieleniu serii; również tym razem wprowadzimy pomocniczą zmienną, aby ostatnio wczytany obiekt przechowywać w buforze:

buf: *obiekt*

oraz zastąpimy zdanie „skopiuj jedną serię z $f0$ na taśmę j" instrukcją

```
repeat read(f0, buf);
   write (f[j], buf)                                            (2.38)
until (buf.klucz > f0↑.klucz) ∨ eof(f0)
```

Kopiowanie serii kończy się wtedy, gdy natkniemy się na pierwszy obiekt następnej serii ($buf.klucz > f0\uparrow.klucz$) lub gdy dojdziemy do końca całego pliku wejściowego ($eof(f0)$).

W rozważanym algorytmie sortowania pozostały do sprecyzowania następujące instrukcje:

(1) ustaw taśmy wejściowe
(2) połącz serie z taśm wejściowych na $t[j]$
(3) zamień taśmy

oraz predykat

(4) wszystkie taśmy wejściowe są wyczerpane.

Najpierw musimy dokładnie zidentyfikować bieżące pliki wejściowe. Zauważmy, że liczba „aktywnych" plików wejściowych może być mniejsza niż $n/2$. W rzeczywistości plików źródłowych może być co najwyżej tyle, ile jest serii; sortowanie kończy się wtedy, gdy pozostanie już tylko jeden plik. W ten sposób jest możliwe, aby podczas inicjowania ostatniego przebiegu sortowania było już mniej niż nh serii. Wprowadzamy z tego powodu zmienną, powiedzmy $k1$, określającą rzeczywistą liczbę używanych plików wejściowych. Zapoczątkowanie zmiennej $k1$ wstawiamy do instrukcji „ustaw taśmy wejściowe" w następujący sposób:

if $l < nh$ **then** $k1 := l$ **else** $k1 := nh$;
for $i := 1$ **to** $k1$ **do** *reset*$(f[t[i]])$;

Naturalnie instrukcja (2) będzie zmniejszać $k1$ wtedy, gdy skończy się któryś z plików źródłowych. Dlatego predykat (4) można łatwo wyrazić relacją

$k1 = 0$

Instrukcję (2) trudniej sprecyzować; składa się na nią wielokrotny wybór najmniejszego klucza spomiędzy dostępnych obiektów źródłowych, a następnie przeniesienie wybranego elementu do pliku wynikowego, tzn. bieżącego pliku wyjściowego. Konieczność określania końca każdej serii znowu komplikuje proces. Koniec serii może wystąpić wtedy, gdy (1) następny klucz jest mniejszy niż klucz bieżący lub (2) został osiągnięty koniec pliku źródłowego. W drugim przypadku zmniejszenie $k1$ powoduje usunięcie taśmy; w pierwszym przypadku serię zamyka się przez wyłączenie pliku z dalszego wyboru elementów, lecz tylko do chwili, w której zostanie zakończone tworzenie bieżącej serii wyjściowej. Jasne, że potrzebna jest w takim razie druga zmienna, powiedzmy $k2$, określająca liczbę taśm źródłowych używanych w rzeczywistości do wybierania następnego elementu. Jej wartość początkowo określa się jako równą $k1$ i zmniejsza wówczas, gdy tylko zakończy się seria z powodu warunku (1).

Niestety, wprowadzenie $k2$ nie wystarcza; znajomość liczby taśm to za mało. Musimy wiedzieć dokładnie, *które* taśmy są jeszcze używane. Oczywistym rozwiązaniem jest zastosowanie tablicy zmiennych boolowskich, określających użycie taśm. Wybierzemy jednakże inną metodę, która doprowadzi nas do procedury dokonującej wyboru efektywniej, ponieważ pomimo wszystko jest to najczęściej wykonywana część całego algorytmu. Zamiast tablicy boolowskiej wprowadzamy drugą mapę taśm, powiedzmy ta. Mapa ta będzie używana zamiast mapy t, a jej wartości $ta[1]$ ... $ta[k2]$ będą indeksami używanych taśm. Dlatego instrukcję (2) można sformułować następująco:

```
k2 := k1;
repeat „wybierz najmniejszy klucz, niech ta[mx] będzie numerem jego taśmy";
   read(f[ta[mx]], buf);
   write(f[t[j]], buf);                                              (2.39)
   if eof(f[ta[mx]]) then „usuń taśmę" else
   if buf.klucz > f[ta[mx]]↑.klucz then „zamknij serię"
until k2 = 0
```

Ponieważ liczba jednostek taśmowych dostępnych w dowolnej maszynie cyfrowej jest zazwyczaj dość mała, więc algorytm wyboru określony szczegółowo w następnym kroku precyzowania programu może być prostym przeszukiwaniem liniowym. Instrukcja „usuń taśmę" obejmuje zmniejszenie *k*1, a także *k*2 oraz przepisanie indeksów w mapie *ta*. Instrukcja „zamknij serię" zmniejsza jedynie wartość *k*2 i odpowiednio przestawia indeksy w mapie *ta*. Szczegóły pokazano w programie 2.15, będącym ostatnim po (2.39) krokiem precyzowania programu (2.37). Zauważmy, że procedura *rewrite* przewija taśmy, skoro tylko zostanie przeczytana z nich ostatnia seria. Instrukcja „zamień taśmy" jest opracowana zgodnie z podanymi wcześniej wyjaśnieniami.

PROGRAM 2.15
Sortowanie przez łączenie wyważone

```
program łączeniewyważone(output);
{sortowanie n-kierunkowe przez łączenie wyważone}
const n=6; nh=3;  {liczba taśm}
type obiekt=record
               klucz: integer
            end;
   taśma=file of obiekt;
   nrtaśmy=1..n;
var dł, los: integer;  {używane przy generacji pliku}
   kt: boolean;        {koniec taśmy}
   buf: obiekt;
   f0: taśma; {f0 jest taśmą wejściową z liczbami losowymi}
   f: array [1..n] of taśma;

procedure wydruk(var f: taśma; n: nrtaśmy);
   var z: integer;
begin writeln('TAŚMA', n: 2); z := 0;
   while ¬ eof(f) do
   begin read(f, buf); write(output, buf.klucz: 5); z := z+1;
      if z=25 then
         begin writeln(output); z := 0
         end
```

```
    end;
    if z≠0 then writeln(output); reset(f)
end {wydruk};

procedure taśmowesortowanieprzezłączenie;
    var i, j, mx, tx: nrtaśmy;
        k1, k2, l: integer;
        x, min: integer;
        t, ta: array [nrtaśmy] of nrtaśmy;

begin {rozdziel początkowe serie na t[1] ... t[nh]}
    for i := 1 to nh do rewrite(f[i]);
    j := nh; l := 0;
    repeat if j < nh then j := j+1 else j := 1;
        {skopiuj jedną serię z f0 na taśmę j}
        l := l+1;
        repeat read(f0, buf); write(f[j], buf)
        until (buf.klucz > f0↑.klucz) ∨ eof(f0)
    until eof(f0);
    for i := 1 to n do t[i] := i;
    repeat {łącz z t[1] ... t[nh] na t[nh+1] ... t[n]}
        if l < nh then k1 := l else k1 := nh;
        {k1 = liczba taśm wejściowych w tej fazie}
        for i := 1 to k1 do
            begin reset(f[t[i]]); wydruk(f[t[i]], t[i]); ta[i] := t[i]
            end;
        l := 0; {l=liczba połączonych serii}
        j := nh+1; {j = indeks taśmy wyjściowej}
        repeat {połącz serie z t[1] ... t[k1] na t[j]}
            k2 := k1; {k2=liczba aktywnych taśm wejściowych}
            l := l+1;
            repeat {wybierz obiekt najmniejszy}
                i := 1; mx := 1; min := f[ta[1]]↑.klucz;
                while i < k2 do
                begin i := i+1; x := f[ta[i]]↑.klucz;
                      if x < min then
                      begin min := x; mx := i
                      end
                end;
                {ta[mx] ma obiekt najmniejszy, przesuń go na t[j]}
                read(f[ta[mx]], buf); kt := eof(f[ta[mx]]);
                write (f[t[j]], buf);
                if kt then
                begin rewrite(f[ta[mx]]); {usuń taśmę}
```

```
                ta[mx] := ta[k2]; ta[k2] := ta[k1];
                k1 := k1-1; k2 := k2-1
             end else
             if buf.klucz > f[ta[mx]]↑.klucz then
             begin tx := [ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx;
                k2 := k2-1
             end
          until k2=0;
          if j<n then j := j+1 else j := nh+1
       until k1=0;
       for i := 1 to nh do
          begin tx := t[i]; t[i] := t[i+nh]; t[i+nh] := tx
          end
    until l=1;
    reset(f[t[1]]); wydruk(f[t[1]], t[1]);  {posortowane wyjście jest na t[1]}
end {taśmowesortowanieprzełączenie};

begin {generuj losowo plik f0);
   dł := 200; los := 7789; rewrite(f0);
   repeat los := (131071*los) mod 2147483647;
      buf.klucz := los div 2147484; write(f0, buf); dł := dł-1
   until dł=0;
   reset(f0); wydruk(f0, 1);
   taśmowesortowanieprzełączenie
end.
```

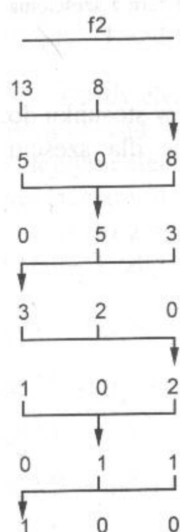
2.3.4. Sortowanie polifazowe

Omówiliśmy już podstawowe metody oraz zdobyliśmy właściwe przygotowanie, możemy więc teraz zbadać i zaprogramować jeszcze jeden algorytm sortowania, efektywniejszy od sortowania wyważonego. Widzieliśmy, że łączenie wyważone eliminuje czyste operacje kopiowania, ponieważ operacje dystrybucji i łączenia są zespolone w jednej fazie. Pojawia się pytanie, czy taśmy mogą być jeszcze lepiej wykorzystane. Jest to naprawdę trudny problem: kluczem do następnego usprawnienia jest rezygnacja ze sztywnego pojęcia, jakim jest prosty przebieg. Oznacza to, że będziemy używać taśm w sposób bardziej skomplikowany, niż przeznaczając $N/2$ taśm na źródłowe i drugie tyle na wynikowe oraz zamieniając taśmy źródłowe z wynikowymi na końcu każdego oddzielnego przebiegu. W zamian rozszerzamy pojęcie przebiegu. Metodę tę wynalazł R.L. Gilstad [2.3] i nazwał ją **sortowaniem polifazowym** (wielofazowym; ang. *polyphase sort*).

Metodę sortowania polifazowego zilustrujemy na przykładzie trzech taśm. Za każdym razem obiekty łączone z dwóch taśm odsyła się na trzecią taśmę. Skoro

tylko jedna z taśm źródłowych się wyczerpie, staje się natychmiast taśmą wynikową dla operacji łączenia z jeszcze niewyczerpanej taśmy źródłowej oraz z taśmy, która uprzednio była taśmą wynikową.

Ponieważ wiemy, że n serii na każdej taśmie wejściowej jest transformowanych w n serii na taśmie wyjściowej, wystarczy wypisać liczby serii znajdujących się na taśmach (zamiast określać ustawienie kluczy). Na rysunku 2.14 przyjęto, że początkowe dwie taśmy wejściowe $f1$ i $f2$ zawierają odpowiednio 13 i 8 serii. Tak więc w pierwszym „przebiegu" dokona się połączenia 8 serii z $f1$ i $f2$ i przesłania ich na $f3$, w drugim „przebiegu" – pozostałych 5 serii z $f1$ i $f3$ będzie po połączeniu przesłanych na $f2$ itd. W końcu na taśmie $f1$ znajdzie się posortowany plik.

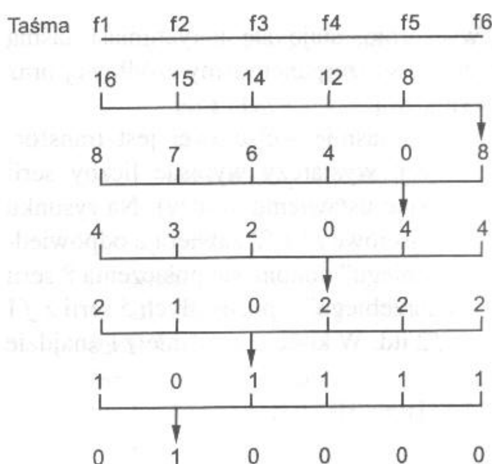


RYSUNEK 2.14
Sortowanie polifazowe 21 serii z trzema taśmami

Drugi przykład pokazuje metodę polifazową z sześcioma taśmami. Niech początkowo na taśmie $f1$ znajduje się 16 serii, 15 serii – na $f2$, 14 na $f3$, 12 na $f4$ oraz 8 na $f5$; w pierwszym częściowym przebiegu 8 serii zostanie połączonych na taśmie $f6$; ostatecznie taśma $f2$ będzie zawierać posortowany zbiór (zob. rys. 2.15).

Sortowanie polifazowe jest efektywniejsze niż łączenie wyważone, ponieważ – przy użyciu N taśm – działa zawsze jako łączenie $(N-1)$-kierunkowe, a nie jako łączenie $N/2$-kierunkowe. Ponieważ liczba potrzebnych przebiegów wynosi w przybliżeniu $\log_N n$, gdzie n jest liczbą elementów do posortowania, N zaś stopniem operacji łączenia, metoda polifazowa zapowiada istotny postęp w stosunku do łączenia wyważonego.

Oczywiście w powyższych przykładach początkowe rozdzielenie serii zostało starannie dobrane. Aby znaleźć początkowe rozdzielenie serii prowadzące do właściwego działania metody, zaczniemy analizę od końca, począwszy od rozdzielenia końcowego (ostatni wiersz na rys. 2.15). Wpisując w tablice dwa



RYSUNEK 2.15
Sortowanie polifazowe 65 serii z sześcioma taśmami

powyższe przykłady i przesuwając każdy wiersz o jedną pozycję w stosunku do wiersza poprzedniego, otrzymujemy w tabl. 2.13 i 2.14 dane dla sześciu przebiegów oraz, odpowiednio, trzech i sześciu taśm.

TABLICA 2.13
Idealne rozdzielenie serii na dwie taśmy

l	$a_1^{(l)}$	$a_2^{(l)}$	$\Sigma a_i^{(l)}$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

Z tablicy 2.13 możemy wydedukować relacje

$$\left.\begin{aligned} a_2^{(l+1)} &= a_1^{(l)} \\ a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} \end{aligned}\right\} \quad \text{dla } l > 0 \tag{2.40}$$

oraz $a_1^{(0)} = 1$, $a_2^{(0)} = 0$. Kładąc $a_1^{(i)} = f_i$, otrzymujemy

$$f_{i+1} = f_i + f_{i-1} \qquad \text{dla } i \geqslant 1$$

$$f_1 = 1 \tag{2.41}$$

$$f_0 = 0$$

TABLICA 2.14
Idealne rozdzielenie serii na pięć taśm

l	$a_1^{(l)}$	$a_2^{(l)}$	$a_3^{(l)}$	$a_4^{(l)}$	$a_5^{(l)}$	$\Sigma a_i^{(l)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Są to wzory rekurencyjne (inaczej: relacje rekurencyjne) definiujące tzw. **ciąg Fibonacciego**:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Każdy element ciągu Fibonacciego jest sumą swoich dwóch poprzedników. Wynika stąd, że początkowe liczby serii na dwóch taśmach muszą być dwoma kolejnymi elementami ciągu Fibonacciego; wtedy sortowanie polifazowe z trzema taśmami działa właściwie.

A co z drugim przykładem (tabl. 2.14) z sześcioma taśmami? Wzory na kolejne liczby można także łatwo wyprowadzić:

$$
\begin{aligned}
a_5^{(l+1)} &= a_1^{(l)} \\
a_4^{(l+1)} &= a_1^{(l)} + a_5^{(l)} = a_1^{(l)} + a_1^{(l-1)} \\
a_3^{(l+1)} &= a_1^{(l)} + a_4^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} \\
a_2^{(l+1)} &= a_1^{(l)} + a_3^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} \\
a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} + a_1^{(l-4)}
\end{aligned}
\tag{2.42}
$$

Podstawiając f_i na $a_1^{(i)}$, otrzymujemy

$$
\begin{aligned}
& f_{i+1} = f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4} \qquad \text{dla } i \geqslant 4 \\
& f_4 = 1 \\
& f_i = 0 \qquad \text{dla } i < 4
\end{aligned}
\tag{2.43}
$$

Są to tzw. liczby Fibonacciego rzędu 4. Ogólnie, **liczby Fibonacciego rzędu** p są zdefiniowane następująco:

$$
\begin{aligned}
& f_{i+1}^{(p)} = f_i^{(p)} + f_{i-1}^{(p)} + \ldots + f_{i-p}^{(p)} \qquad \text{dla } i \geqslant p \\
& f_p^{(p)} = 1 \\
& f_i^{(p)} = 0 \qquad \text{dla } 0 \leqslant i < p
\end{aligned}
\tag{2.44}
$$

Zauważmy, że zwykłe liczby Fibonacciego są rzędu 1.

TABLICA 2.15
Liczby serii dające się idealnie rozdzielić

l \ n	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001
15	1597	13745	34513	56417	76806	95617
16	2584	25281	66526	110913	152351	190465
17	4181	46499	128233	218049	302201	379399
18	6765	85525	247177	428673	599441	755749
19	10946	157305	476449	842749	1189041	1505425
20	17711	289329	918385	1656801	2358561	2998753

W ten sposób pokazaliśmy, że początkowe liczby serii dla idealnego sortowania polifazowego z n taśmami są sumami dowolnych $n-1$, $n-2$, ..., 1 (zob. tabl. 2.15) kolejnych liczb Fibonacciego rzędu $n-2$. Wynika z tego, że metodę można stosować tylko wtedy, gdy liczba serii jest sumą $n-1$ takich sum Fibonacciego. Powstaje istotne pytanie: co robić wtedy, gdy początkowo liczba serii nie jest taką idealną sumą? Odpowiedź jest prosta i typowa w takich sytuacjach: zasymulujemy istnienie hipotetycznych, pustych serii w ten sposób, aby suma serii rzeczywistych i hipotetycznych była sumą idealną. Puste serie zwane są **seriami fikcyjnymi**. Nie jest to jednak w pełni zadowalająca odpowiedź, ponieważ natychmiast powstaje dalsze i trudniejsze pytanie: jak podczas łączenia rozpoznawać fikcyjne serie? Zanim odpowiemy na to pytanie, musimy najpierw zbadać ważniejszy problem rozdzielania początkowego serii i zdecydować się na sposób rozdzielenia rzeczywistych i fikcyjnych serii na $n-1$ taśm.

Aby znaleźć właściwy sposób dokonywania rozdzielania, musimy jednakże wiedzieć, jak są łączone rzeczywiste i fikcyjne serie. Jasne, że wybór fikcyjnej serii z taśmy i znaczy tyle, że taśma i nie bierze udziału w tym łączeniu, co powoduje, że łączenie odbywa się z mniej niż $n-1$ źródeł. Łączenie serii fikcyjnych ze wszystkich $n-1$ taśm źródłowych powoduje, że w rzeczywistości nie wykonuje się operacji łączenia, tylko na taśmie wynikowej zapisuje serię fikcyjną. Wnioskujemy z tego, że serie fikcyjne powinny być rozłożone możliwie

równomiernie, ponieważ zainteresowani jesteśmy aktywnym łączeniem serii z możliwie największej liczby taśm.

Zapomnijmy na chwilę o seriach fikcyjnych i zastanówmy się nad zagadnieniem rozdzielenia *nieznanej* liczby serii na $n-1$ taśm. Jasne, że liczby Fibonacciego rzędu $n-2$, określające pożądane liczby serii na każdej taśmie, mogą być generowane podczas trwania operacji rozdzielania. Zakładając na przykład $n = 6$ oraz odwołując się do tabl. 2.14, zaczniemy od rozłożenia serii tak, jak podano w wierszu z indeksem $l = 1$ (1, 1, 1, 1, 1); jeżeli pozostały jeszcze jakieś serie, przechodzimy do drugiego wiersza (2, 2, 2, 2, 1); jeżeli ciągle są jeszcze dostępne serie, to rozdzielanie przebiega według trzeciego wiersza (4, 4, 4, 3, 2) itd. Indeks wiersza nazwiemy *poziomem*. Oczywiście im większa liczba serii, tym wyższy będzie poziom liczb Fibonacciego, który notabene jest równy liczbie przebiegów lub przestawień taśm potrzebnych do późniejszego sortowania.

Można teraz sformułować pierwszą wersję algorytmu rozdzielania.

(1) Celem rozdzielania niech będą liczby Fibonacciego rzędu $n-2$, poziomu 1.
(2) Dokonaj rozdzielenia według zadanego celu.
(3) Jeżeli cel został osiągnięty, oblicz następny poziom liczb Fibonacciego; różnica między nimi a liczbami z poprzedniego poziomu tworzy nowy cel rozdzielania. Wróć do kroku 2. Jeżeli cel nie może być osiągnięty z powodu wyczerpania danych wejściowych, to zakończ proces rozdzielania.

Wzory na obliczanie następnego poziomu liczb Fibonacciego są zawarte w definicji (2.44). Możemy więc skoncentrować uwagę na kroku 2, w którym przy zadanym celu kolejne serie mają być jedna po drugiej rozdzielone na $n-1$ taśm. W tym miejscu muszą powtórnie pojawić się w naszych rozważaniach serie fikcyjne.

Załóżmy, że przy zwiększaniu poziomu zapisujemy następny cel za pomocą różnic d_i dla $i = 1 \dots n-1$, gdzie d_i oznacza liczbę serii, które mają być w tym kroku przesłane na taśmę i. Możemy teraz założyć, że natychmiast odsyłamy d_i fikcyjnych serii na taśmę i; dalsze rozdzielanie będziemy wtedy uważać za *zastępowanie* serii fikcyjnych seriami rzeczywistymi, zaznaczając każde zastąpienie przez odjęcie 1 od d_i. Tak więc d_i będzie wskazywać liczbę fikcyjnych serii na taśmie i w chwili, gdy wejście będzie puste.

Nie wiadomo, który algorytm daje optymalne rozdzielenie serii. Bardzo dobrą metodą okazało się tzw. **rozdzielanie poziome** (zob. Knuth, vol. 3. s. 270). Określenie to można zrozumieć, jeżeli serie będzie się uważać za sterty w postaci silosów, jak pokazano to na rys. 2.16 dla $n = 6$ i poziomu 5 (zob. tabl. 2.14).

Aby uzyskać możliwie najszybciej równe rozdzielenie pozostałych fikcyjnych serii, zastępujemy je seriami rzeczywistymi, redukując wymiar stert przez zdejmowanie fikcyjnych serii poziomami, postępując z lewa na prawo. Tym sposobem serie są rozdzielone na taśmy tak, jak wskazują ich kolejne numery na rys. 2.16.

8					
7	1				
6	2	3	4		
5	5	6	7	8	
4	9	10	11	12	
3	13	14	15	16	17
2	18	19	20	21	22
1	23	24	25	26	27
	28	29	30	31	32

RYSUNEK 2.16
„Poziomy rozkład" serii

Możemy teraz opisać algorytm w postaci procedury o nazwie *wybierztaśmę*, która będzie wywoływana za każdym razem, gdy zostanie skopiowana seria i należy wybrać nową taśmę dla następnej serii. Zakładamy istnienie zmiennej i oznaczającej indeks bieżącej taśmy wynikowej. Wielkości a_i oraz d_i określają idealną liczbę serii i liczbę fikcyjnych serii dla taśmy i.

j: *nrtaśmy*;
a, *d*: **array** [*nrtaśmy*] **of** *indeks*; (2.45)
poziom: *integer*

Zmienne te są inicjowane następującymi wartościami:

$a_i = 1, \quad d_i = 1 \quad$ dla $i = 1, ..., n-1$
$a_n = 0, \quad d_n = 0 \quad$ (fikcyjne)
$j = 1$
$poziom = 1$

Zauważmy, że zawsze wtedy, gdy zwiększany jest poziom, procedura *wybierztaśmę* musi obliczyć wartości następnego wiersza tabl. 2.14, tzn. wartości $a_1^{(l)} ... a_{n-1}^{(l)}$. „Następny cel", tzn. różnice $d_i = a_i^{(l)} - a_i^{(l-1)}$ muszą także być policzone w tym czasie. W opisywanym algorytmie korzysta się z faktu, że otrzymane d_i zmniejszają się wraz ze wzrostem indeksu (obniżające się schodki na rys. 2.16). (Zauważmy, że wyjątkiem jest przejście z poziomu 0 do poziomu 1; dlatego algorytm można stosować, począwszy od poziomu 1). Procedura *wybierztaśmę* kończy się zmniejszeniem d_j o 1; operacja ta występuje zamiast operacji zastąpienia serii fikcyjnej na taśmie j serią rzeczywistą.

```
procedure wybierztaśmę;
  var i: nrtaśmy; z: integer;
begin
  if d[j] < d[j+1] then j := j+1 else
  begin if d[j]=0 then
    begin poziom := poziom+1; z := a[1];
      for i := 1 to n-1 do                                    (2.46)
      begin d[i] := z+a[i+1]-a[i]; a[i] := z+a[i+1]
      end
```

```
        end;
        j := 1
    end;
    d[j] := d[j]-1
end
```

Przy założeniu, że dostępna jest procedura kopiująca serię z taśmy źródłowej $f0$ na $f[j]$, możemy opisać początkową fazę rozdzielania następująco (zakładamy zawsze, że na taśmie źródłowej znajduje się przynajmniej jedna seria):

```
repeat wybierztaśmę; kopiujserię                                    (2.47)
until eof(f0)
```

W tym miejscu musimy się jednakże zatrzymać na moment. Przypomnijmy sobie zjawisko, jakie występowało przy rozdzielaniu serii w omawianym poprzednio algorytmie łączenia naturalnego; chodzi mianowicie o fakt, że dwie serie zapisane kolejno na taśmie wynikowej mogą utworzyć jedną serię, powodując, że nieprawdziwa staje się zakładana liczba serii. Wtedy łatwo poradziliśmy sobie z tym efektem ubocznym, zmieniając algorytm sortowania tak, aby jego poprawność nie zależała od liczby serii. Jednakże w metodzie sortowania polifazowego jesteśmy szczególnie zainteresowani śledzeniem dokładnej liczby serii na wszystkich taśmach. Wynika stąd, że nie możemy przeoczyć efektów takiego przypadkowego połączenia.

Nie można więc ominąć dodatkowej komplikacji algorytmu rozdzielania. Staje się konieczne pamiętanie kluczy ostatnich obiektów z ostatniej serii z każdej taśmy. Do tego celu wprowadzamy zmienną

```
ostatni: array [nrtaśmy] of integer
```

Następną próbą opisu algorytmu dystrybucji może być fragment programu

```
repeat wybierztaśmę;
   if ostatni[j] ≤ f0↑.klucz then
      „ciąg dalszy starej serii";                                   (2.48)
   kopiujserię; ostatni[j] := f0↑.klucz
until eof(f0)
```

Popełniliśmy oczywisty błąd: zapomnieliśmy, że *ostatni*$[j]$ otrzymuje zdefiniowaną wartość dopiero po skopiowaniu pierwszej serii! W poprawnym rozwiązaniu najpierw rozdziela się po jednej serii na każdą z $n-1$ taśm bez sprawdzania zmiennej *ostatni*$[j]$. Pozostałe serie są rozdzielane według programu (2.49).

```
while ¬ eof(f0) do
begin wybierztaśmę;
   if ostatni[j] ≤ f0↑.klucz then
   begin {ciąg dalszy starej serii}                                    (2.49)
      kopiujserię;
      if eof(f0) then d[j] := d[j]+1 else kopiujserię
   end
   else kopiujserię
end
```

Zakładamy tutaj, że przypisanie wartości zmiennej *ostatni*[*j*] jest zawarte w procedurze *kopiujserię*.

Znajdujemy się w końcu w sytuacji umożliwiającej zmierzenie się z zasadniczym algorytmem sortowania przez łączenie polifazowe. Jego podstawowa struktura podobna jest do głównych części programu łączenia n-kierunkowego: zewnętrznej pętli łączącej serie do chwili, w której wyczerpią się taśmy źródłowe; wewnętrznej pętli łączącej po jednej serii ze wszystkich taśm źródłowych; wreszcie, najbardziej wewnętrznej pętli wybierającej początkowy klucz i przepisującej odpowiedni element na taśmę wynikową. Zasadnicze różnice są następujące:

(1) Zamiast $n/2$ jest teraz tylko jedna taśma wynikowa w każdym przebiegu.
(2) Zamiast zamiany $n/2$ taśm wejściowych i $n/2$ wyjściowych taśmy są *zmieniane kolejno*. Uzyskuje się to dzięki zastosowaniu mapy indeksów taśm t.
(3) Liczba taśm wejściowych zmienia się z serii na serię; na początku każdej serii określa się ją na podstawie liczników serii fikcyjnych d_i. Jeżeli $d_i > 0$ dla wszystkich i, to $n-1$ serii fikcyjnych „łączy się" w jedną serię fikcyjną przez zwiększenie licznika d_n dla taśmy wyjściowej. W przeciwnym przypadku łączy się jedną serię ze wszystkich taśm, których $d_i = 0$, oraz dla wszystkich innych taśm zmniejsza się wartość d_i, co oznacza, że została zabrana jedna fikcyjna seria. Liczbę taśm wejściowych biorących udział w łączeniu oznaczamy zmienną k.
(4) Nie można zdefiniować zakończenia fazy przez sprawdzanie warunku końca pliku na $n-1$ taśmie, ponieważ mogą być jeszcze potrzebne operacje łączenia, dla których z tej taśmy będą pobierane serie fikcyjne. Można za to określić potrzebną teoretycznie liczbę serii na podstawie współczynników a_i. Współczynniki $a_i^{(l)}$ były obliczone podczas fazy rozdzielania; teraz można je powtórnie policzyć „od tyłu".

Można teraz na podstawie tych reguł sformułować główną część algorytmu sortowania polifazowego przy założeniu, że wszystkich $n-1$ taśm jest już na nowo ustawionych, a mapa taśm ma początkowe wartości $t_i = i$.

```
repeat {łącz z t[1] ... t[n-1] na t[n]}
   z := a[n-1]; d[n] := 0; rewrite(f[t[n]]);
   repeat k := 0; {łącz jedną serię}
      {określ liczbę k aktywnych taśm wejściowych}
      for i := 1 to n-1 do
      if d[i] > 0 then d[i] := d[i]-1 else
         begin k := k+1; ta[k] := t[i]                    (2.50)
         end;
      if k=0 then d[n] := d[n]+1 else
         „łącz jedną serię rzeczywistą z t[1] ... t[k]";
      z := z-1
   until z := 0;
   reset(f[t[n]]);
   „zamień kolejne taśmy w mapie t; oblicz a[i] dla następnego poziomu";
   rewrite(f[t[n]]); poziom := poziom - 1
until poziom=0;
{posortowane dane wyjściowe znajdują się na t[1]}
```

Rzeczywista operacja łączenia jest niemal identyczna z operacją łączenia w programie sortowania przez łączenie n-kierunkowe; jedyna różnica polega na tym, że algorytm usuwania taśm jest trochę prostszy. Kolejną zamianę w mapie indeksów taśm i odpowiednich liczników d_i (oraz powtórne obliczenie niższego poziomu współczynników a_i) wykonuje się w prosty sposób i można to szczegółowo obejrzeć w programie 2.16, który reprezentuje całość algorytmu sortowania polifazowego.

PROGRAM 2.16
Sortowanie polifazowe

```
program sortowaniepolifazowe(output);
{sortowanie polifazowe z n taśmami}
const n=6;      {liczba taśm}
type obiekt = record
                 klucz: integer
              end;
   taśma = file of obiekt;
   nrtaśmy = 1..n;
var dł, los: integer;       {używane przy generacji pliku}
   kt: boolean;
   buf: obiekt;
   f0: taśma; {f0 jest taśmą wejściową z losowymi liczbami}
   f: array [1..n] of taśma;

procedure wydruk(var f: taśma; n: nrtaśmy);
   var z: integer;
```

```
begin z := 0;
   writeln('TAŚMA', n: 2);
   while ¬ eof(f) do
   begin read(f, buf); write(output, buf.klucz: 5);
      z := z+1;
      if z=25 then
         begin writeln(output); z := 0
         end
   end;
   if z≠0 then writeln(output); reset(f)
end {wydruk};

procedure sortpolifazowe;
   var i, j, mx, tn: nrtaśmy;
      k, poziom: integer;
      a, d: array [nrtaśmy] of integer;
         {a[j]=idealna liczba serii na taśmie j}
         {d[j]=liczba fikcyjnych serii na taśmie j}
      dn, x, min, z: integer;
      ostatni: array [nrtaśmy] of integer;
         {ostatni[j]=klucz ostatniego obiektu na taśmie j}
      t, ta: array [nrtaśmy] of nrtaśmy;
         {mapy numerów taśm}
   procedure wybierztaśmę;
      var i: nrtaśmy; z: integer;
   begin
      if d[j] < d[j+1] then j := j+1 else
      begin if d[j]=0 then
            begin poziom := poziom+1; z := a[1];
               for i := 1 to n-1 do
               begin d[i] := z+a[i+1]-a[i];
                  a[i] := z+a[i+1]
               end
            end;
         j := 1
      end;
      d[j] := d[j]-1
   end;

   procedure kopiujserię;
   begin {skopiuj jedną serię z f0 na taśmę j}
      repeat read(f0, buf); write(f[j], buf);
      until eof(f0) ∨ (buf.klucz > f0↑.klucz);
      ostatni[j] := buf.klucz
   end;
```

```
begin {rozdziel początkowe serie}
  for i := 1 to n-1 do
    begin a[i] := 1; d[i] := 1; rewrite(f[i])
    end;
  poziom := 1; j := 1; a[n] := 0; d[n] := 0;
  repeat wybierztaśmę; kopiujserię
  until eof(f0) ∨ (j = n-1);
  while ¬ eof(f) do
  begin wybierztaśmę;
    if ostatni[j] ≤ f0↑.klucz then
    begin {dalszy ciąg starej serii}
      kopiujserię;
      if eof(f0) then d[j] := d[j]+1 else kopiujserię
    end
    else kopiujserię
  end;
  for i := 1 to n-1 do reset(f[i]);
  for i := 1 to n do t[i] := i;
  repeat {łącz z t[1] ... t[n-1] na t[n]}
    z := a[n-1]; d[n] := 0; rewrite(f[t[n]]);
    repeat k := 0; {łącz jedną serię}
      for i := 1 to n-1 do
      if d[i] > 0 then d[i] := d[i]-1 else
        begin k := k+1; ta[k] := t[i]
        end;
      if k = 0 then d[n] := d[n]+1 else
      begin {łącz jedną rzeczywistą serię z t[1] ... t[k]}
        repeat i := 1; mx := 1;
          min := f[ta[1]]↑.klucz;
          while i < k do
          begin i := i+1; x := f[ta[i]]↑.klucz;
            if x < min then
            begin min := x; mx := i
            end
          end;
          {ta[mx] zawiera obiekt najmniejszy;
                    przesuń go na t[n]}
          read(f[ta[mx]], buf); kt := eof(f[ta[mx]]);
          write(f[t[n]], buf);
          if (buf.klucz > f[ta[mx]]↑.klucz) ∨ kt then
          begin {pomiń tę taśmę}
            ta[mx] := ta[k]; k := k-1
          end
```

```
            until k=0
        end;
        z := z-1
    until z=0;
    reset(f[t[n]]); wydruk(f[t[n]], t[n]); {zamień kolejno taśmy}
    tn := t[n]; dn := d[n]; z := a[n-1];
    for i := n downto 2 do
        begin t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1]-z
        end;
    t[1] := tn; d[1] := dn; a[1] := z;
    {posortowane wyjście jest na t[1]}
    wydruk(f[t[1]], t[1]); poziom := poziom - 1
  until poziom=0;
end {sortpolifazowe};

begin {generuj losowo plik}
  dł := 200; los := 7789;
  repeat los := (131071*los) mod 2147483647;
    buf.klucz := los div 2147484; write(f0, buf);
    dł := dł-1
  until dł=0;
  reset(f0); wydruk(f0, 1);
  sortpolifazowe
end.
```

2.3.5. Rozdzielanie serii początkowych

Przedstawione powyżej rozwiązania doprowadziły nas do skomplikowanych programów sortujących, ponieważ prostsze metody działające na tablicach opierały się na założeniu, że pamięć o dostępie swobodnym jest wystarczająco duża na to, aby zmieścić cały zbiór danych do sortowania. Bardzo często tak duża pamięć jest nieosiągalna i trzeba stosować pamięci sekwencyjne, takie jak taśmy. Zauważmy, że w skonstruowanych do tej pory metodach sortowania sekwencyjnego nie potrzeba praktycznie żadnej pamięci operacyjnej, z wyjątkiem pamięci na bufory plików i oczywiście dla samego programu. Jednakże faktem jest, że nawet małe maszyny cyfrowe mają pewną pamięć operacyjną o dostępie swobodnym, która prawie zawsze jest większa niż ta, która jest potrzebna dla skonstruowanych tutaj programów. Nie można usprawiedliwić rezygnacji z optymalnego wykorzystania tej pamięci.

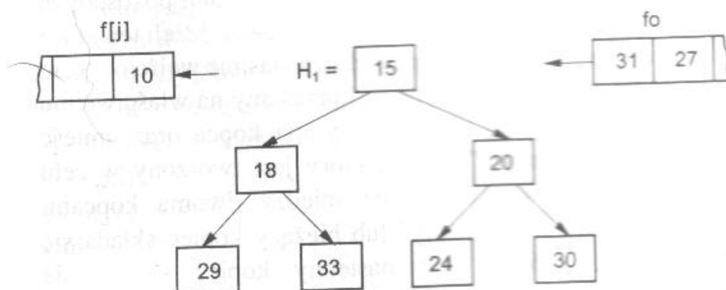
Rozwiązanie polega na połączeniu metod sortowania tablic i plików. W szczególności można zastosować w fazie rozdzielania serii początkowych odpowiednio przystosowaną metodę sortowania tablic, tak aby te serie miały już długość l równą, w przybliżeniu, wielkości dostępnej pamięci operacyjnej. Jasne

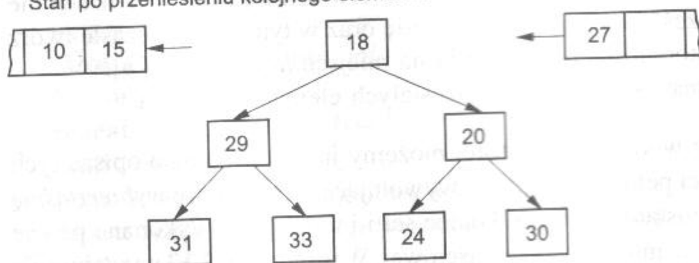
jest, że dodatkowe sortowanie tablic w kolejnych przebiegach łączenia nie może zwiększyć efektywności, ponieważ przetwarzane serie będą coraz dłuższe, a więc, będą zawsze większe od dostępnej pamięci głównej. W rezultacie możemy z powodzeniem skoncentrować uwagę na usprawnieniu algorytmu generującego serie początkowe.

Rzecz jasna poszukiwania skoncentrujemy na logarytmicznych metodach sortowania tablic. Najodpowiedniejsza z nich to metoda sortowania drzewiastego, czyli przez kopcowanie (zob. p. 2.2.5). Kopiec można traktować jako tunel, przez który muszą przejść wszystkie składniki pliku – jedne szybciej, drugie zaś wolniej. Najmniejszy klucz pobiera się z wierzchołka kopca, a zastępowanie go jest procesem bardzo efektywnym. Proces przechodzenia składnika z taśmy wejściowej $f0$ przez cały „tunel kopcowy" h na taśmę wyjściową $f[j]$ można prosto opisać w następujący sposób:

write($f[j]$, $h[1]$);
read($f0$, $h[1]$); (2.51)
przesiewanie(1, n)

„Przesiewanie" jest to opisany w p. 2.2.5 proces przesiewania w dół kopca na odpowiednie miejsce ostatnio wstawionego składnika $h[1]$. Zauważmy, że $h[1]$ jest obiektem *najmniejszym* w kopcu. Przykład pokazano na rys. 2.17.





RYSUNEK 2.17
Przesiewanie klucza przez kopiec

Ostatecznie program staje się bardziej skomplikowany z następujących powodów:

(1) Kopiec h jest początkowo pusty i trzeba go wypełnić.
(2) Pod koniec kopiec jest wypełniony tylko częściowo i ostatecznie staje się pusty.
(3) Musimy śledzić początki nowych serii, aby we właściwym czasie zmienić indeks taśmy wyjściowej j.

Zanim przejdziemy dalej, zadeklarujemy formalnie zmienne, które na pewno będą dalej używane:

var $f0$: *taśma*;
  f: **array** [*nrtaśmy*] **of** *taśma*;
  h: **arry** [1 ..m] **of** *obiekt*;
  l, p: *integer*

m jest wielkością kopca h. Będziemy używać stałej mh do oznaczenia $m/2$; l i p są indeksami tablicy h. Proces przechodzenia składników można podzielić na pięć rozłącznych części.

(1) Wczytaj pierwszych mh obiektów z $f0$ i umieść je w górnej połowie kopca, gdzie nie jest potrzebne żadne uporządkowanie kluczy.
(2) Wczytaj następnych mh obiektów i umieść je w dolnej połowie kopca, przesiewając każdy obiekt na właściwe mu miejsce (budowanie kopca).
(3) Ustaw l równe m i powtarzaj następujący krok dla wszystkich pozostałych obiektów z $f0$. Umieść $h[1]$ na właściwej taśmie wynikowej. Jeżeli ten klucz jest mniejszy lub równy kluczowi kolejnego obiektu na taśmie wejściowej, to ten element należy do tej samej serii i może być przesiany na właściwe mu miejsce. W przeciwnym przypadku zmniejsz wymiar kopca oraz umieść nowy obiekt na drugim, „wyższym" kopcu, który jest tworzony w celu gromadzenia następnej serii. Rozgraniczenie między dwoma kopcami zaznaczamy indeksem l. Tak więc „dolny" lub bieżący kopiec składa się z obiektów $h[1] \dots h[l]$, „górny" lub następny kopiec składa się z $h[l+1] \dots h[m]$. Jeżeli $l=0$, to zmień taśmę wyjściową i ustaw l równe m.
(4) Teraz plik źródłowy jest wyczerpany. Najpierw ustaw p równe m; następnie wypisz dolną część kończącą bieżącą serię oraz w tym samym czasie twórz górną część i stopniowo przesuwaj ją na miejsce $h[l+1] \dots h[p]$.
(5) Ostatnia seria jest generowana z pozostałych elementów kopca.

Znaleźliśmy się w sytuacji, w której możemy już szczegółowo opisać tych pięć etapów w postaci pełnego programu wywołującego procedurę *wybierztaśmę* zawsze wtedy, gdy zostanie wykryty koniec serii i muszą być wykonane pewne czynności, aby zmienić indeks taśmy wyjściowej. W programie 2.17 wywołuje się w tym miejscu pusty podprogram, który tylko zlicza generowane serie. Wszystkie elementy odsyła się na taśmę $f1$.

PROGRAM 2.17
Rozdzielanie serii początkowych przez kopiec

```
program rodziel(f0, f1, output);
{rozdzielanie początkowe serii za pomocą sortowania przez kopcowanie}
const m=30; mh=15; {rozmiar kopca}
type obiekt=record
                 klucz: integer
           end;
   taśma=file of obiekt;
   indeks=0..m;
var l, p: indeks;
   f0, f1: taśma;
   licznik: integer; {licznik serii}
   h: array [1..m] of obiekt; {kopiec}

procedure wybierztaśmę;
begin licznik:= licznik+1;
   {pusta; zlicza rozdzielane serie}
end {wybierztaśmę};

procedure przesiewanie(l, p: indeks);
   label 13;
   var i, j: integer; x: obiekt;
begin i:= l; j:= 2*i; x:= h[i];
   while j≤p do
   begin if j<p then
      if h[j].klucz>h[j+1].klucz then j:= j+1;
      if x.klucz≤h[j].klucz then goto 13;
      h[i]:= h[j]; i:= j; j:= 2*i
   end;
13: h[i]:= x
end;

begin {utwórz serie początkowe za pomocą sortowania przez kopcowanie}
   licznik:= 0; reset(f0); rewrite(f1);
   wybierztaśmę;
   {krok 1: wypełnij górną połowę kopca h}
   l:= m;
   repeat read(f0, h[l]); l:= l-1
   until l=mh;
   {krok 2: wypełnij dolną połowę kopca h}
   repeat read(f0, h[l]); przesiewanie(l, m); l:= l - 1
   until l = 0;
   {krok 3: przechodzenie serii przez zapełniony kopiec}
   l:= m;
```

```
    while ¬ eof(f0) do
    begin write (f1, h[1]];
        if h[1].klucz≤f0↑.klucz then
        begin {nowy obiekt należy do tej samej serii}
            read(f0, h[1]); przesiewanie(1, l);
        end else
        begin {nowy obiekt należy do następnej serii}
            h[1]:= h[l]; przesiewanie(1, l−1);
            read(f0, h[l]); if l≤mh then przesiewanie(l, m);
            l:= l−1;
            if l=0 then
            begin {kopiec jest pełen; zacznij nową serię}
                l:= m; wybierztaśmę
            end
        end
    end;
    {krok 4: wypisz dolną część kopca}
    p:= m;
    repeat write (f1, h[1]);
        h[1]:= h[l]; przesiewanie(1, l−1);
        h[l]:= h[p]; p:= p−1;
        if l≤mh then przesiewanie(l, p); l:= l−1
    until l=0;
    {krok 5: wypisz górną część kopca; generuj ostatnią serię}
    wybierztaśmę;
    while p>0 do
    begin write(f1, h[1]);
        h[1]:= h[p]; przesiewanie (1, p); p:= p−1
    end;
    writeln(licznik)
end.
```

Jeżeli teraz spróbujemy połączyć ten program na przykład z sortowaniem polifazowym, to napotkamy poważne trudności. Pojawią się one z następujących powodów. Program sortujący w części początkowej składa się z dość skomplikowanej procedury zamiany taśm i jest napisany przy założeniu, że dostępna jest procedura *kopiujserię*, dostarczająca dokładnie jedną serię na wybraną taśmę. Z drugiej strony, program sortowania przez kopcowanie jest złożoną procedurą działającą przy założeniu, że jest dostępna zamknięta procedura *wybierztaśmę*, która po prostu wybiera nową taśmę. Nie byłoby żadnego kłopotu, gdyby w jednym lub w dwóch programach omawiane procedury były wywoływane w jednym miejscu; w rzeczywistości wywołuje się je w kilku miejscach w obu programach.

Tego rodzaju sytuację najlepiej opisują tzw. **procedury współbieżne**; mają one zastosowanie w tych przypadkach, w których współistnieje kilka procesów. Najbardziej typowym reprezentantem będzie połączenie procesu produkującego ciąg informacji w oddzielnych porcjach oraz procesu konsumującego te informacje. Relację między producentem i konsumentem można wyrazić w postaci dwóch procedur współbieżnych. Jedna z nich może być równie dobrze programem głównym.

Procedurę współbieżną można uważać za procedurę zawierającą jeden lub kilka punktów przerwania. Po napotkaniu takiego punktu przerwania sterowanie wraca do programu, który wywołał procedurę współbieżną. Gdy procedura współbieżna zostanie ponownie wywołana, wtedy wykonanie jej będzie podjęte w punkcie przerwania. W naszym przykładzie sortowanie polifazowe możemy uważać za program główny wołający *kopiujserię* sformułowaną w postaci procedury współbieżnej. Składa się ona z głównego tekstu programu 2.17, w którym każde wołanie procedury *wybierztaśmę* będzie reprezentować punkt przerwania. Sprawdzenie, czy został osiągnięty koniec pliku, trzeba wszędzie zastąpić sprawdzeniem, czy procedura współbieżna skończyła się wykonywać. Logicznie brzmiącym sformułowaniem będzie *kks* (*kopiujserię*) zamiast $eof(f0)$.

Analiza i wnioski. Jakiej efektywności można się spodziewać po sortowaniu polifazowym z początkowym rozdzielaniem serii za pomocą sortowania przez kopcowanie? Omówimy najpierw spodziewane zyski z wprowadzenia kopca.

W ciągu losowo rozłożonych kluczy spodziewana średnia długość serii równa się 2. Ile wynosi ta długość po przejściu ciągu przez kopiec o wymiarze m? Chciałoby się odpowiedzieć, że m, lecz – na szczęście – rzeczywisty wynik analizy probabilistycznej jest o wiele lepszy, a mianowicie wynosi $2m$ (zob. Knuth, vol. 3, s. 254). Stąd współczynnik usprawnienia jest równy m. Przybliżoną efektywność sortowania polifazowego można odczytać z tabl. 2.15, gdzie są zaznaczone największe liczby serii początkowych, które mogą być posortowane w zadanej liczbie częściowych przebiegów (poziomów) z zadaną liczbą n taśm. W przykładzie z 6 taśmami i wymiarem kopca $m=100$ plik zawierający do 165 680 100 serii początkowych może być posortowany w 20 częściowych przebiegach. Jest to już godna podkreślenia efektywność.

Przeglądając ponownie połączenie sortowania polifazowego z sortowaniem przez kopcowanie, można się tylko zdumiewać złożonością tego programu. A przecież wykonuje on to samo, w prosty sposób zdefiniowane, zadanie uporządkowania zbioru obiektów co każdy z krótkich programików opartych na prostych zasadach sortowania tablic. Morał całego tego rozdziału można traktować jako ilustrację:

(1) ścisłego związku między algorytmem i założonymi strukturami danych, a w szczególności wpływu struktur danych na algorytm;
(2) udoskonalenia zwiększającego efektywność programu nawet wtedy, gdy dostępne dla tego programu struktury danych (ciągi zamiast tablic) nie są dostosowane do zadania.

Ćwiczenia

2.1

Które z algorytmów podanych w programach 2.1 do 2.6, 2.8, 2.10 oraz 2.13 są stabilnymi metodami sortowania?

2.2

Czy program 2.2 będzie wciąż dobrze działał, jeżeli w instrukcji **while** $l \leqslant p$ zastąpi się przez $l < p$? Czy program będzie dalej poprawny, jeżeli instrukcję $p := m-1$ oraz instrukcję $l := m+1$ uprości się do $p := m$ oraz $l := m$? Jeżeli nie, to znajdź zbiory wartości $a_1 \dots a_n$, dla których zmieniony program się nie wykona.

2.3

Zaprogramuj dla dostępnej Ci maszyny cyfrowej i zmierz czasy wykonania trzech prostych metod sortowania. Znajdź wagi, przez które należy mnożyć współczyniki *Po* i *Pr*, aby otrzymać oszacowania czasu wykonania.

2.4

Sprawdź działanie programu 2.8 sortowania przez kopcowanie na różnych losowych ciągach oraz wyznacz, ile razy jest średnio wykonywana instrukcja **goto** 13. Ponieważ jest to stosunkowo mała liczba, interesujące staje się pytanie: czy istnieje sposób usunięcia sprawdzenia

$x.klucz \geqslant a[j].klucz$

poza pętlę **while**?

2.5

Rozwiąż następującą „oczywistą" wersję procedury *podział* (program 2.9):

```
i := 1; j := n;
x := a[(n+1) div 2].klucz;
repeat
   while a[i].klucz < x do i := i+1;
   while x < a[j].klucz do j := j-1;
   w := a[i]; a[i] := a[j]; a[j] := w
until i > j
```

Znajdź zbiór wartości $a_1 \dots a_n$, dla których ta wersja wykona się źle.

2.6

Napisz program łączący ze sobą algorytmy sortowania szybkiego i sortowania bąbelkowego w sposób następujący. Użyj metody sortowania szybkiego, aby otrzymać (nieposortowane) podziały o długości $m\,(1 \leqslant m \leqslant n)$; następnie zakończ zadanie, stosując metodę sortowania bąbelkowego. Zauważ, że drugą metodę można zastosować do całej tablicy n elementów, minimalizując w ten sposób konieczną „buchalterię". Znajdź wartość m minimalizującą całkowity czas sortowania.

Uwaga. Jasne, że optymalna wartość m będzie nieduża. Może się więc opłacać, aby sortowanie bąbelkowe przeszło dokładnie $m-1$ razy przez tablicę, zamiast aby było wykonywane do momentu, gdy w ostatnim przebiegu nie wystąpiło żadne przestawienie.

2.7

Wykonaj takie samo doświadczenie co w ćwiczeniu 6, stosując metodę sortowania przez proste wybieranie zamiast metody sortowania bąbelkowego. Naturalnie sortowanie przez wybieranie nie może być rozciągnięte na całą tablicę; dlatego najprawdopodobniej zwiększy się liczba operacji wykonywanych na indeksach.

2.8

Napisz rekurencyjny algorytm sortowania szybkiego w ten sposób, aby sortowanie krótszego podziału wykonało się przed sortowaniem dłuższego podziału. Pierwszą czynność wykonaj instrukcją iteracyjną, drugą wywołaniem rekurencyjnym. (Tak napisany program będzie więc zawierał jedno wywołanie rekurencyjne zamiast dwóch wywołań w programie 2.10 i żadnego wywołania w programie 2.11).

2.9

Znajdź permutację kluczy 1, 2, ..., n, dla której sortowanie szybkie działa najgorzej (najlepiej); ($n=5, 6, 8$).

2.10

Zaprogramuj metodę łączenia naturalnego analogicznie do metody łączenia prostego z programu 2.13, tzn. tak, aby program działał na tablicy o podwójnej długości, poczynając od jej obu końców i posuwając się do środka. Porównaj efektywność działania z programem 2.13.

2.11

Zauważ, że w dwukierunkowym łączeniu naturalnym najmniejszy spośród dostępnych kluczy nie jest wybierany bez zastanowienia. Natomiast, po napotkaniu końca serii, pozostałe elementy drugiej serii są już po prostu kopiowane do ciągu wyjściowego. Na przykład łączenie

2, 4, 5, 1, 2, ...
3, 6, 8, 9, 7, ...

daje w wyniku ciąg

2, 3, 4, 5, 6, 8, 9, 1, 2, ...

zamiast

2, 3, 4, 5, 1, 2 6, 8, 9, ...

który, jak się może wydawać, jest lepiej uporządkowany. Jaki jest powód przyjęcia takiej strategii?

2.12

Czemu służy zmienna *ta* w programie 2.15? W jakiej sytuacji jest wykonywana instrukcja

begin *rewrite*($f[ta[mx]]$); ...

a kiedy instrukcja

begin $tx := ta[mx]$; ...?

2.13

Dlaczego potrzebna jest zmienna *ostatni* w programie 2.16 sortowania polifazowego, a nie jest ona potrzebna w programie 2.15?

2.15

Metodą sortowania podobną do polifazowej jest tzw. **sortowanie przez łączenie kaskadowe** [2.1] i [2.9]. Zasosowano w niej odmienny wzór łączenia. Mając na przykład danych sześć taśm *T1*, ... *T6*, łączenie kaskadowe także rozpoczyna się od „idealnego rozdzielenia" serii na *T1* ... *T5*, następnie wykonuje się pięciokierunkowe łączenie z *T1* ... *T5* na *T6* do chwili, gdy *T5* stanie się pusta, następnie (nie korzystając z *T6*) czterokierunkowe łączenie na *T5*, trzykierunkowe łączenie na *T4*, dwukierunkowe łączenie na *T3* i w końcu kopiuje się *T1* na *T2*. Następny przebieg odbywa się w ten sam sposób, rozpoczynając się od pięciokierunkowego łączenia na *T1* itd. Chociaż schemat ten wydaje się gorszy od sortowania polifazowego, ponieważ w pewnych momentach nie korzysta się z wszystkich taśm oraz wykonuje się zwykłe operacje kopiowania, jest on jednak niespodziewanie lepszy od sortowania polifazowego dla (bardzo) dużych plików i dla sześciu lub więcej taśm. Napisz dobrze zbudowany program dla metody łączenia kaskadowego.

Literatura

2.1. Betz B.K., Carter: *ACM National Conf.*, **14**, 1959, Paper 14.

2.2. Floyd R.W.: Treesort (algorytmy 113 i 243). *Comm. ACM*, **5**, No. 8, 1962, s. 434 i *Comm. ACM*, **7**, No. 12, 1964, s. 701.

2.3. Gilstad R.L.: Polyphase Merge Sorting – An Advanced Technique. *Proc. AFIPS Eastern Jt. Comp. Conf.*, **18**, 1960, s. 143–148.

2.4. Hoare C.A.R.: Proof of a Program: FIND. *Comm. ACM*, **13**, No. 1, 1970, s. 39–45.

2.5. Hoare C.A.R.: Proof of a Recursive Program: Quicksort. *Comp. J.*, **14**, No. 4, 1971, s. 391–395.

2.6. Hoare C.A.R.: Quicksort. *Comp. J.*, **5**, No. 1, 1962, s. 10–15.

2.7. Knuth D.E.: *The Art of Computer Programming*. Vol. 3, Reading, Mass.: Addison-Wesley 1973.

2.8. Knuth D.E.: *The Art of Computer Programming*. Vol. 3, s. 86–95.

2.9. Knuth D.E.: *The Art of Computer Programming*. Vol. 3, s. 289.

2.10. Lorin H.: A Guided Bibliography to Sorting. *IBM Syst. J.*, **10**, No. 3, 1971, s. 244–254.

2.11. Shell D.L.: A Highspeed Sorting Procedure. *Comm. ACM*, **2**, No. 7, 1959, s. 30–32.

2.12. Singleton R.C.: An Efficient Algorithm for Sorting with Minimal Storage (algorytm 347). *Comm. ACM*, **12**, No. 3, 1969, s. 185.

2.13. Van Emden M.H.: Increasing the Efficiency of Quicksort (algorytm 402). *Comm. ACM*, **13**, No. 9, 1970, s. 563–566, 693.

2.14. Williams J.W.J.: Heapsort (algorytm 232). *Comm. ACM*, **7**, No. 6, 1964, s. 347–348.

3 Algorytmy rekurencyjne

3.1. Wprowadzenie

Obiekt zwany jest **rekurencyjnym** (ang. *recursive*), jeżeli częściowo składa się z siebie samego lub jego definicja odwołuje się do jego samego. Rekursję (ang. *recursion*) spotykamy nie tylko w matematyce, ale także w życiu codziennym. Kto z nas nie widział obrazka reklamowego, którego częścią jest ten sam obrazek?

RYSUNEK 3.1
Obrazek rekurencyjny

Rekursja jest szczególnie silnym narzędziem w definicjach matematycznych. A oto kilka przykładów:

(1) Liczby naturalne:
   (a) 1 jest liczbą naturalną;
   (b) następnik liczby naturalnej jest liczbą naturalną.

(2) Struktury drzewiaste:
(a) ○ jest drzewem (zwanym drzewem pustym);
(b) jeśli t_1 i t_2 są drzewami, to

jest drzewem (narysowanym „do góry nogami").
(3) Funkcja silni $n!$ (dla argumentów całkowitych, nieujemnych):
(a) $0! = 1$;
(b) jeśli $n > 0$, to $n! = n \cdot (n-1)!$

Siła rekursji wyraża się w możliwości definiowania nieskończonego zbioru obiektów za pomocą skończonego wyrażenia. W ten sam sposób nieskończoną liczbę obliczeń można opisać za pomocą skończonego programu rekurencyjnego, nawet jeśli program nie zawiera jawnych iteracji. Algorytmy rekurencyjne są jednakowoż szczególnie przydatne, jeśli problem, który rozwiązujemy, funkcja, której wartości obliczamy, czy też struktury danych, na których działamy, są zdefiniowane w sposób rekurencyjny. Mówiąc ogólnie, program rekurencyjny P może być wyrażony jako złożenie $\mathscr{P}$ instrukcji podstawowych S_i (nie zawierających P) i samego programu P.

$$P \equiv \mathscr{P}[S_i, P] \tag{3.1}$$

Narzędziem umożliwiającym tworzenie programów rekurencyjnych jest pojęcie **procedury** (podprogramu). Pozwala ono nadać instrukcji nazwę, za pomocą której można uaktywnić tę instrukcję. Jeśli procedura P zawiera bezpośrednie odwołanie do samej siebie, to P nazywa się procedurą **bezpośrednio rekurencyjną**. Jeśli P zawiera odwołanie do innej procedury Q, która zawiera bezpośrednie lub pośrednie odwołanie do P, to P nazywa się procedurą **pośrednio rekurencyjną**. Użycie rekursji nie musi więc być natychmiast dostrzeżone w tekście programu.

Zazwyczaj wiąże się z procedurą pewien zbiór obiektów lokalnych, tj. zmiennych, stałych, typów i procedur zdefiniowanych lokalnie dla tej procedury i nie istniejących lub nie mających poza nią znaczenia. Za każdym razem, gdy taka procedura jest uaktywniana rekurencyjnie, tworzy się nowy zbiór zmiennych lokalnych. Chociaż te zmienne mają takie same nazwy jak odpowiadające im elementy w zbiorze utworzonym przy poprzednim uaktywnieniu tej samej procedury, jednakże mają one inne wartości, a konfliktów związanych z ich nazywaniem unika się dzięki regule zakresu identyfikatorów: identyfikatory zawsze odnoszą się do ostatnio utworzonego zbioru zmiennych. Ta sama reguła obowiązuje w stosunku do parametrów procedury, które – z definicji – znane są tylko w tej procedurze.

Podobnie jak instrukcje iteracyjne, procedury rekurencyjne dopuszczają możliwość wykonywania nieskończonych obliczeń. Wiąże się z tym konieczność

rozważenia **problemu stopu** (zakończenia). Zasadniczym wymaganiem w takiej sytuacji jest uzależnienie rekurencyjnego wywołania procedury P od warunku B, który w pewnym momencie przestaje być spełniony. Dlatego też schemat algorytmów rekurencyjnych można dokładniej wyrazić jako

$$P \equiv \textbf{if } B \textbf{ then } \mathscr{P}[S_i, P] \tag{3.2}$$

lub jako

$$P \equiv \mathscr{P}[S_i, \textbf{if } B \textbf{ then } P] \tag{3.3}$$

W celu wykazania, że proces iteracji się skończy, definiuje się zazwyczaj pewną funkcję $f(x)$ (x jest zbiorem zmiennych programu) taką, że $f(x) \leqslant 0$ implikuje warunek zakończenia powtarzania (instrukcji **while** – dopóki lub **repeat** – powtarzaj), a następnie dowodzi się, że wartość $f(x)$ zmniejsza się przy każdym powtórzeniu. W taki sam sposób – wykazując, że każde wykonanie P zmniejsza $f(x)$ – dowodzi się zakończenia wykonywania programu rekurencyjnego. Szczególnie oczywistym sposobem zapewnienia zakończenia programu jest związanie z procedurą P parametru n (przekazywanego przez wartość), a następnie wywołanie procedury P z wartością tego parametru równą $n-1$. Zastąpienie warunku B warunkiem $n>0$ gwarantuje wtedy zakończenie obliczeń. Można to wyrazić za pomocą następujących schematów programów:

$$P(n) \equiv \textbf{if } n>0 \textbf{ then } \mathscr{P}[S_i, P(n-1)] \tag{3.4}$$

$$P(n) \equiv \mathscr{P}[S_i, \textbf{if } n>0 \textbf{ then } P(n-1)] \tag{3.5}$$

W praktycznych zastosowaniach nie wystarczy pokazać, że głębokość rekursji jest skończona, ale również, że jest względnie mała. Przyczyną tego jest fakt, że każde rekurencyjne uaktywnienie procedury P wymaga pewnego obszaru pamięci służącego do ulokowania zmiennych procedury. Oprócz tych zmiennych lokalnych musi być zanotowany bieżący stan obliczenia po to, aby można go było odzyskać z chwilą zakończenia wykonywania kolejnego uaktywnienia procedury P. Mieliśmy już do czynienia z taką sytuacją przy omawianiu procedury sortowania szybkiego w rozdz. 2. Stwierdziliśmy wtedy, że dla „naiwnej" wersji programu sortującego, składającego się z instrukcji dokonującej podziału n obiektów na dwie grupy oraz z dwóch rekurencyjnych wywołań procedur sortujących te grupy, głębokość rekursji w najgorszym przypadku była bliska n. Dzięki zręczniejszemu podejściu do tego zagadnienia można było określić ograniczenie tej głębokości przez $\log n$. Różnica między n i $\log n$ jest wystarczająco duża, aby sytuację wysoce nieodpowiednią dla rekursji zastąpić taką, w której rekursja jest idealnym rozwiązaniem praktycznym.

3.2. Kiedy nie stosować rekursji

Algorytmy rekurencyjne są szczególnie odpowiednie wtedy, gdy rozważany problem lub przetwarzane dane są zdefiniowane w sposób rekurencyjny. Nie oznacza to jednak, że algorytm rekurencyjny musi być najlepszym sposobem rozwiązania danego zagadnienia. W rzeczywistości objaśnianie pojęcia algorytmów rekurencyjnych na takich nieodpowiednich przykładach było główną przyczyną rozprzestrzeniania się obaw i niechęci do używania rekursji w programowaniu oraz utożsamiania rekursji z nieefektywnością. Działo się tak również i z tego powodu, że szeroko rozpowszechniony język programowania Fortran zakazuje rekurencyjnego użycia podprogramów, zmuszając do szukania innych rozwiązań nawet wówczas, gdy sformułowanie rekurencyjne jest najodpowiedniejsze.

Programy, w których możemy uniknąć rekursji, można scharakteryzować za pomocą następującego schematu, uwidaczniającego zasadę ich budowy:

$$P \equiv \textbf{if } B \textbf{ then } (S\,;\,P) \tag{3.6}$$

i równoważnego mu

$$P \equiv (S;\ \textbf{if } B \textbf{ then } P) \tag{3.7}$$

Schematy te są naturalne w sytuacjach, gdy obliczane wartości zdefiniowano za pomocą prostych relacji rekurencyjnych. Spójrzmy na dobrze znany przykład obliczania kolejnych wartości funkcji silnia $f_i = i!$

$$\begin{aligned} i &= 0, \quad 1, \quad 2, \quad 3, \quad 4, \quad 5, \ \ldots \\ f_i &= 1, \quad 1, \quad 2, \quad 6, \quad 24, \quad 120, \ \ldots \end{aligned} \tag{3.8}$$

Wartość „zerowa" zdefiniowana jest explicite jako $f_0 = 1$, podczas gdy kolejne liczby są zazwyczaj definiowane rekurencyjnie za pomocą ich poprzedników:

$$f_{i+1} = (i+1) \cdot f_i \tag{3.9}$$

Formuła ta sugeruje zastosowanie algorytmu rekurencyjnego przy przejściu do obliczania n-tej wartości funkcji. Wprowadzając dwie zmienne I i F oznaczające wartości i i f_i dla i-tego poziomu rekursji, poniższe obliczenie (3.10) pozwoli znaleźć kolejną liczbę w ciągu (3.8):

$$I := I+1\,; \qquad F := I * F \tag{3.10}$$

Zastępując S w schemacie (3.6) formułą (3.10), uzyskujemy następujący program rekurencyjny:

$$P \equiv \textbf{if } I < n \textbf{ then } (I := I+1\,;\ F := I * F;\ P) \tag{3.11}$$
$$I := 0;\ F := 1;\ P$$

Pierwszy wiersz programu możemy wyrazić w przyjętym przez nas języku programowania jako

```
procedure P;
begin if I < n then
   begin I := I+1; F := I*F; P
   end
end                                        (3.12)
```

Częściej używa się w istocie równoważnej postaci (3.13). Procedura *P* została zastąpiona tzw. procedurą funkcyjną

```
function F(I: integer): integer;
begin if I > 0 then F := I*F(I-1)          (3.13)
            else F := 1
end
```

(zwaną też funkcją) tj. procedurą, z którą jest związana explicite wartość wynikowa określonego typu; procedura funkcyjna może dzięki temu występować bezpośrednio jako składnik wyrażeń. Zmienna *F* staje się wtedy zbyteczna, a rolę *I* odgrywa parametr procedury. Jest oczywiste, że w tym przypadku można rekursję zastąpić iteracją, co prowadzi do programu

```
I := 0; F := 1;                            (3.14)
while I < n do
   begin I := I+1; F := I*F
   end
```

Na ogół programy odpowiadające schematom (3.6) lub (3.7) powinno się przekształcać do powyższej postaci, korzystając ze schematu (3.15):

$$P \equiv (x := x_0;\ \textbf{while}\ B\ \textbf{do}\ S) \tag{3.15}$$

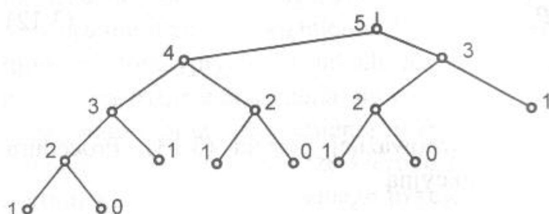
Istnieją bardziej skomplikowane schematy rekurencyjne, które mogą i powinny być przekształcone do postaci iteracyjnej. Jako przykład można podać obliczanie kolejnych liczb Fibonacciego, zdefiniowanych za pomocą relacji rekurencyjnej

$$fib_{n+1} = fib_n + fib_{n-1} \qquad \text{dla } n > 1 \tag{3.16}$$

gdzie $fib_1 = 1$, $fib_0 = 0$. Bezpośrednie, „naiwne” podejście do tego zadania prowadzi do programu

```
function Fib(n: integer): integer;
begin if n = 0 then Fib := 0 else
      if n = 1 then Fib := 1 else          (3.17)
      Fib := Fib(n-1) + Fib(n-2)
end
```

Obliczanie wartości fib_n za pomocą wywołania procedury funkcyjnej $Fib(n)$ powoduje jej rekurencyjne uaktywnienie. Jak często? Zauważmy, że każde wywołanie z $n > 1$ powoduje dwa dalsze wywołania, tj. całkowita liczba wywołań rośnie wykładniczo (zob. rys. 3.2). Taki program jest więc niepraktyczny.



RYSUNEK 3.2
15 wywołań funkcji *Fib* spowodowanych przez wywołanie $Fib(5)$

Liczby Fibonacciego można jednak obliczać, posługując się schematem iteracyjnym. Unika się wtedy ponownego obliczania tych samych wartości dzięki wprowadzeniu zmiennych pomocniczych x i y takich, że $x = fib_i$ i $y = fib_{i-1}$.

```
{oblicz x = fib_n dla n > 0}
i := 1; x := 1; y := 0;
while i < n do                                    (3.18)
   begin z := x; i := i+1;
         x := x+y; y := z
   end
```

(Zauważmy, że trzy przypisania wartości zmiennym x, y i z można wyrazić za pomocą tylko dwóch przypisań, bez potrzeby wprowadzania zmiennej pomocniczej z: $x := x+y$; $y := x-y$).

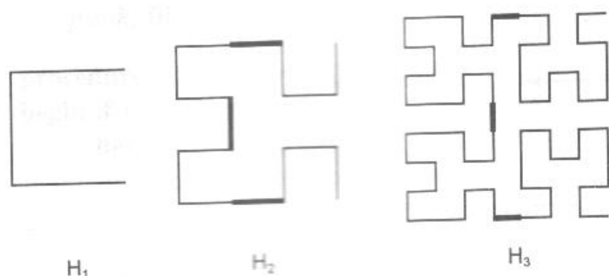
Wniosek jest następujący: unikajmy rekursji, jeśli istnieje *oczywiste* rozwiązanie iteracyjne.

Nie powinno to jednak być utożsamiane z unikaniem rekursji za wszelką cenę. Istnieje wiele sensownych zastosowań rekursji, z których kilka przedstawimy w następnych punktach. Fakt, że procedury rekurencyjne realizuje się w maszynach w istocie nierekurencyjnych dowodzi, że w praktyce każdy program rekurencyjny można przekształcić na program czysto iteracyjny. Wymaga to jednak jawnego posługiwania się stosem rekursji. Operacje dotyczące obsługi stosu zaciemniają zasadniczą ideę programu do tego stopnia, że na ogół staje się ona bardzo trudna do zrozumienia. Wniosek z tego jest następujący: algorytmy, które w swej istocie są rekurencyjne, a nie iteracyjne, powinny być formułowane jako procedury rekurencyjne. Aby właściwie ocenić ten wniosek, czytelnik powinien porównać programy 2.10 i 2.11.

Pozostałą część tego rozdziału poświęcimy opracowaniu pewnych programów rekurencyjnych w sytuacjach, gdy rekursja jest w pełni uzasadniona. Liczne przykłady zastosowania rekursji w przypadkach, w których podstawowe struktury danych sugerują oczywiste i naturalne rozwiązanie rekurencyjne, można znaleźć również w rozdz. 4 i 5.

3.3. Dwa przykłady programów rekurencyjnych

Atrakcyjny układ graficzny pokazany na rys. 3.5. powstał ze złożenia pięciu krzywych. Regularny kształt tych krzywych sugeruje możliwość kreślenia ich za pomocą pisaka sterowanego przez komputer. Celem naszym jest odkrycie schematu rekursji, zgodnie z którym mógłby być skonstruowany program kreślący. Trzy spośród pięciu nakładających się krzywych mają kształt przedstawiony na rys. 3.3; oznaczmy je przez H_1, H_2 i H_3. Po bliższym przyjrzeniu okazuje się, że krzywą H_{i+1} otrzymano ze złożenia 4 krzywych H_i dwukrotnie zmniejszonych, odpowiednio obróconych i powiązanych razem trzema liniami. Zauważmy, że H_1 może być traktowana jako złożenie 4 pustych krzywych H_0, powiązanych ze sobą trzema liniami prostymi. H_i jest zwana **krzywą Hilberta rzędu *i***, od nazwiska wynalazcy krzywej D. Hilberta (1891).



RYSUNEK 3.3
Krzywe Hilberta rzędu 1, 2 i 3

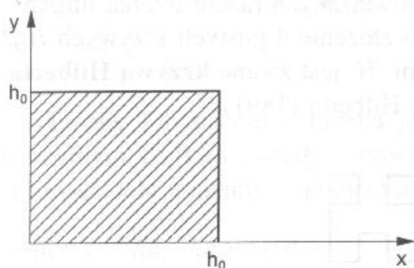
Załóżmy, że nasze podstawowe narzędzia do kreślenia stanowią: dwie zmienne współrzędne x i y, procedura *ustawpióro* (ustawiająca pióro pisaka w punkcie o współrzędnych x i y) i procedura *kreśl* (przesuwająca pióro z jego pozycji bieżącej do pozycji wskazanej przez x i y).

Ponieważ każda krzywa H_i składa się z 4 dwukrotnie zmniejszonych kopii krzywej H_{i-1}, wydaje się naturalne podzielenie procedury rysującej H_i na cztery części, z których każda rysuje krzywą H_{i-1} odpowiednio zmniejszoną i obróconą. Jeśli te cztery części oznaczymy przez A, B, C i D, a procedury kreślące linie łączące będziemy oznaczać strzałkami odpowiednio zorientowanymi, to uzyskamy następujący **schemat rekursji** (zob. rys. 3.3):

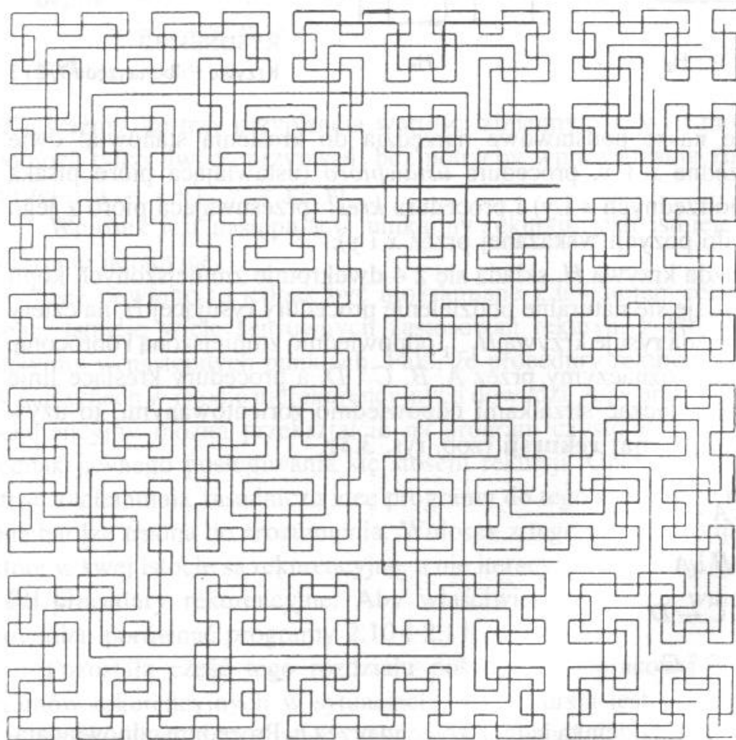
$$
\begin{aligned}
&A: \quad D \leftarrow A \downarrow A \rightarrow B \\
&B: \quad C \uparrow B \rightarrow B \downarrow A \\
&C: \quad B \rightarrow C \uparrow C \leftarrow D \\
&D: \quad A \downarrow D \leftarrow D \uparrow C
\end{aligned}
\qquad (3.19)
$$

Oznaczmy długość odcinka jednostkowego przez h. Procedurę odpowiadającą części A można łatwo zdefiniować za pomocą wywołań rekurencyjnych analogicznie zbudowanych procedur B, D i jej samej.

```
procedure A(i: integer);
begin if i > 0 then
      begin D(i-1); x := x-h; kreśl;
            A(i-1); y := y-h; kreśl;
            A(i-1); x := x+h; kreśl;
            B(i-1)                                    (3.20)
      end
end
```



RYSUNEK 3.4
Powierzchnia kreślenia



RYSUNEK 3.5
Krzywe Hilberta H_1 ... H_5

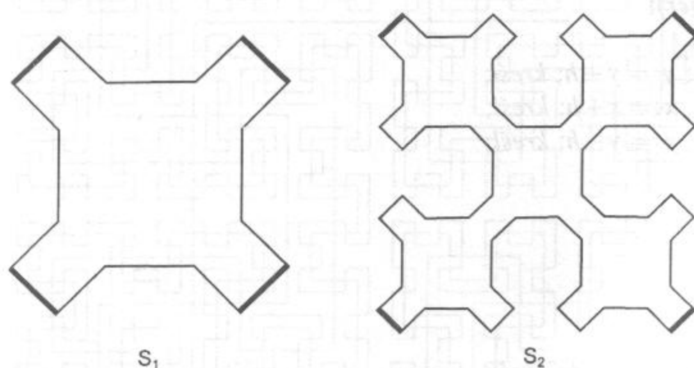
Procedura ta będzie wywoływana przez program główny po jednym razie dla każdej z nakładanych na siebie krzywych Hilberta. Program główny określa również punkt początkowy krzywej, tj. początkowe wartości zmiennych x i y oraz przyrost jednostkowy h. Przez $h0$ jest oznaczona pełna szerokość strony, przy czym dla pewnego $k \geqslant n$ musi zachodzić $h0 = 2^k$ (zob. rys. 3.4). Cały program kreśli n krzywych Hilberta $H_1 \dots H_n$ (zob. program 3.1 i rys. 3.5).

PROGRAM 3.1
Krzywe Hilberta

```
program Hilbert(pisak, output);
{wykreśl krzywe Hilberta rzędu od 1 do n}
const n = 4; h0 = 512;
var i, h, x, y, x0, y0: integer;
    pisak: file of integer;

procedure A(i: integer);
begin if i > 0 then
        begin D(i-1); x := x-h; kreśl;
              A(i-1); y := y-h; kreśl;
              A(i-1); x := x+h; kreśl;
              B(i-1);
        end
end;

procedure B(i: integer);
begin if i > 0 then
        begin C(i-1); y := y+h; kreśl;
              B(i-1); x := x+h; kreśl;
              B(i-1); y := y-h; kreśl;
              A(i-1)
        end
end;

procedure C(i: integer);
begin if i > 0 then
        begin B(i-1); x := x+h; kreśl;
              C(i-1); y := y+h; kreśl;
              C(i-1); x := x-h; kreśl;
              D(i-1)
        end
end;

procedure D(i: integer);
begin if i > 0 then
```

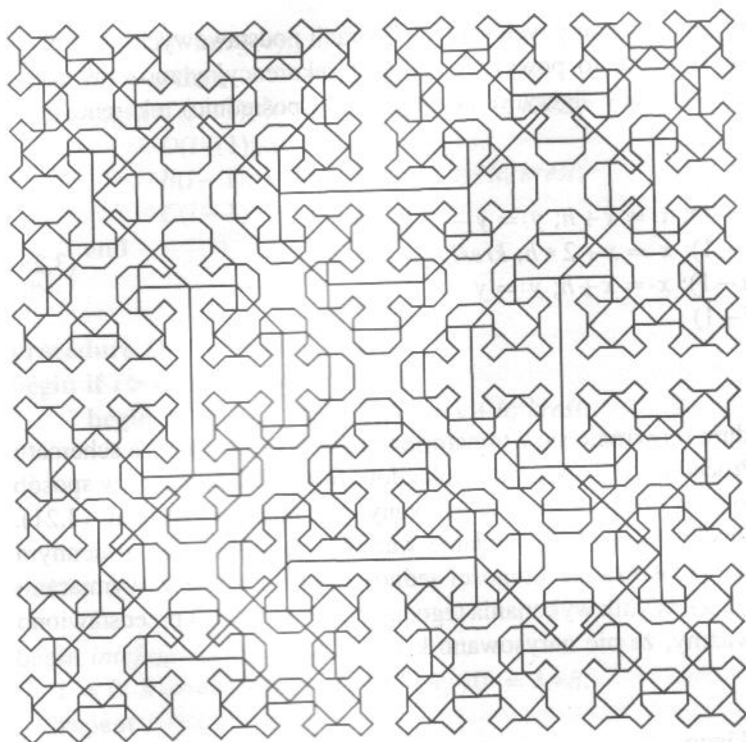
```
        begin A(i−1); y := y−h; kreśl;
              D(i−1); x := x−h; kreśl;
              D(i−1); y := y+h; kreśl;
              C(i−1)
        end
end;

begin inicjujpisanie;
    i := 0; h := h0; x0 := h div 2; y0 := x0;
    repeat {kreśl krzywą Hilberta rzędu i}
        i := i+1; h := h div 2;
        x0 := x0+(h div 2); y0 := y0+(h div 2);
        x := x0; y := y0; ustawpióro;
        A(i)
    until i = n;
    zakończpisanie
end.
```

Podobny, ale nieco bardziej złożony i bardziej wymyślny estetycznie przykład przedstawiono na rys. 3.7. Jest to układ otrzymany z nałożenia na siebie kilku krzywych, z których dwie pokazano na rys. 3.6. Krzywa S_i jest zwana **krzywą Sierpińskiego rzędu *i***. Jaki jest schemat rekursji? Na pierwszy rzut oka wydaje się, że krzywa S_1 z odciętą ewentualnie jedną krawędzią stanowi podstawowy klocek budowlany. Nie prowadzi to jednak do rozwiązania.



RYSUNEK 3.6
Krzywe Sierpińskiego rzędu 1 i 2

W odróżnieniu od krzywych Hilberta krzywe Sierpińskiego są zamknięte (bez punktów przecięć). Wynika stąd, że podstawowy schemat rekursji sprowadza się do kreślenia krzywych otwartych, natomiast powiązanie czterech części składowych za pomocą połączeń nie należy już do schematu rekursji. W rzeczywistości połączenia te składają się z czterech odcinków, leżących w skrajnych rogach,



RYSUNEK 3.7
Krzywe Sierpińskiego $S_1 \dots S_4$

wyróżnionych na rys. 3.6 tłustym drukiem. Traktować je możemy jako krzywe tworzące *niepustą* krzywą S_0 będącą kwadratem ustawionym na jednym wierzchołku.

Zbudowanie schematu rekursji jest teraz proste. Cztery układy składowe oznaczamy ponownie przez A, B, C i D, a odcinki łączące będą, jak poprzednio, kreślone explicite. Zauważmy, że układy składowe różnią się między sobą jedynie obrotem o 90°.

Podstawowy wzorzec dla krzywych Sierpińskiego jest następujący:

$$S: A \searrow B \swarrow C \nwarrow D \nearrow \tag{3.21}$$

a schematy rekursji są postaci:

$$\begin{aligned} &A: A \searrow B \Rightarrow D \nearrow A \\ &B: B \swarrow C \Downarrow A \searrow B \\ &C: C \nwarrow D \Leftarrow B \swarrow C \\ &D: D \nearrow A \Uparrow C \nwarrow D \end{aligned} \tag{3.22}$$

(Strzałki podwójne oznaczają linie o podwójnej długości jednostkowej).

Jeżeli do kreślenia użyjemy tych samych operacji podstawowych co w przykładzie z krzywą Hilberta, to powyższy schemat rekurencyjny możemy łatwo przekształcić na następujący algorytm (bezpośrednio i pośrednio) rekurencyjny:

```
procedure A(i: integer);
begin if i > 0 then
      begin A(i-1); x := x+h; y := y-h; kreśl;
            B(i-1); x := x+2*h; kreśl;                    (3.23)
            D(i-1); x := x+h; y := y+h; kreśl;
            A(i-1)
      end
end
```

Powyższą procedurę otrzymuje się z transformacji pierwszego wiersza schematu rekursji (3.22). Procedury odpowiadające układom *B*, *C* i *D* tworzy się w sposób analogiczny. Program główny jest zbudowany zgodnie z wzorcem (3.21). Zadaniem programu głównego jest nadanie wartości początkowych zmiennym (współrzędnym) oraz określenie długości jednostkowej zgodnie z wymiarami papieru (program 3.2). Wynik wykonania tego programu dla $n = 4$ przedstawiono na rys. 3.7. Zauważmy, że nie narysowano krzywej S_0.

PROGRAM 3.2
Krzywe Sierpińskiego

```
program Sierpiński(pisak, output);
{wykreśl krzywe Sierpińskiego rzędu od 1 do n}
const n = 4; h0 = 512;
var i, h, x, y, x0, y0: integer;
    pisak: file of integer;
procedure A(i: integer);
begin if i > 0 then
      begin A(i-1); x := x+h; y := y-h; kreśl;
            B(i-1); x := x+2*h; kreśl;
            D(i-1); x := x+h; y := y+h; kreśl;
            A(i-1)
      end
end;
procedure B(i: integer);
begin if i > 0 then
      begin B(i-1); x := x-h; y := y-h; kreśl;
            C(i-1); y := y-2*h; kreśl;
            A(i-1); x := x+h; y := y-h; kreśl;
            B(i-1)
      end
end;
```

```
procedure C(i: integer);
begin if i > 0 then
      begin C(i−1); x := x−h; y := y+h; kreśl;
            D(i−1); x := x−2 * h; kreśl;
            B(i−1); x := x−h; y := y−h; kreśl;
            C(i−1)
      end
end;

procedure D(i: integer);
begin if i > 0 then
      begin D(i−1); x := x+h; y := y+h; kreśl;
            A(i−1); y := y+2 * h; kreśl;
            C(i−1); x := x−h; y := y+h; kreśl;
            D(i−1)
      end
end;

begin inicjujpisanie;
   i := 0; h := h0 div 4; x0 := 2 * h; y0 := 3 * h;
   repeat i := i+1; x0 := x0−h;
      h := h div 2; y0 := y0+h;
      x := x0; y := y0; ustawpióro;
      A(i); x := x+h; y := y−h; kreśl;
      B(i); x := x−h; y := y−h; kreśl;
      C(i); x := x−h; y := y+h; kreśl;
      D(i); x := x+h; y := y+h; kreśl;
   until i = n;
   zakończpisanie
end.
```

Rozwiązanie rekurencyjne użyte w obu przykładach jest eleganckie, a przy tym oczywiste i przekonywające. Poprawność programów można łatwo wykazać na podstawie ich struktury i użytego schematu rekursji. Ponadto, dzięki wprowadzeniu w sposób jawny parametru i zgodnie ze schematem 3.5, uzyskaliśmy gwarancję zakończenia działania programu, jako że głębokość rekursji nie może być większa niż n. W przeciwieństwie do tego sformułowania rekurencyjnego programy równoważne, w których uniknięto rekursji, są wyjątkowo niezgrabne i nieczytelne. Czytelnikowi nie przekonanemu o słuszności tego stwierdzenia radzimy spróbować zrozumieć programy przedstawione w artykule [3.3].

3.4. Algorytmy z powrotami

Tematem szczególnie intrygującym w programowaniu jest „automatyczne rozwiązywanie problemów". Zadanie polega na znalezieniu algorytmów szukających rozwiązań dla specyficznych problemów nie za pomocą określonej reguły obliczania, ale metodą prób i błędów. Zwykle stosowanym tu sposobem jest podział rozwiązywanego zadania na „podzadania". Dla zadań tych często najbardziej naturalnym opisem jest opis rekurencyjny. Cały proces rozwiązywania problemu możemy więc traktować jako proces prób i poszukiwań, w którym jest stopniowo rozbudowywane i przeglądane drzewo podzadań. W wielu zastosowaniach drzewo to rośnie bardzo szybko, zazwyczaj wykładniczo, w zależności od zadanego parametru. Równie szybko wzrasta ilość pracy związanej z przeglądaniem drzewa. Najczęściej drzewo można stopniowo obcinać, stosując reguły heurystyczne, co pozwala zredukować obliczenia do rozmiarów możliwych do przyjęcia.

Celem naszym nie jest analizowanie ogólnych reguł heurystycznych. Zamierzamy natomiast omówić ogólne zasady podziału zadań, związanych z rozwiązywaniem pewnej klasy problemów, na podzadania oraz możliwości stosowania rekursji. Zaczniemy od przedstawienia omawianej metody na przykładzie dobrze znanym – znalezienia **drogi skoczka szachowego**.

Załóżmy, że dana jest szachownica $n \times n$ o n^2 polach. Skoczek, który może poruszać się po niej zgodnie z regułami szachowymi, jest umieszczony na polu o współrzędnych x_0, y_0. Problem polega na obiegnięciu całej szachownicy, tzn. na znalezieniu takiej drogi o długości n^2-1 ruchów, że każde pole będzie dokładnie raz „odwiedzone".

Zagadnienie obejścia n^2 pól szachownicy w sposób oczywisty redukuje się do zadania wykonania następnego ruchu lub ewentualnego wykazania, że taki ruch nie jest możliwy. Zajmijmy się więc algorytmem próbującym wykonać następny ruch. Pierwszą wersję algorytmu przedstawia procedura (3.24).

```
procedure próbuj następny ruch;
begin zapoczątkuj wybieranie ruchów;
  repeat wybierz następnego kandydata z listy następnych ruchów;
    if dopuszczalny then
    begin zapisz ruch;                                        (3.24)
      if na szachownicy są wolne pola then
      begin próbuj następny ruch;
      if nieudany then
        wykreśl ostatni zapis ruchu
      end
    end
  until (ruch był udany) ∨ (nie ma następnego kandydata)
end
```

Chcąc uściślić pojęcia występujące w opisie tego algorytmu, jesteśmy zmuszeni podjąć pewne decyzje dotyczące reprezentowania danych. Nie budzi wątpliwości reprezentowanie szachownicy za pomocą macierzy, powiedzmy h. Do oznaczania wartości indeksowych wprowadzamy typ *indeks*:

```
type indeks = 1..n;
var h: array [indeks, indeks] of integer;                    (3.25)
```

Reprezentowanie każdego pola przez liczbę całkowitą zamiast wartości logicznej (mówiącej o tym, czy pole jest zajęte) pozwala nam zapisać historię kolejnych posunięć na szachownicy. Wprowadzamy następujące ustalenie:

$h[x, y] = 0$: pole $\langle x, y\rangle$ nie zostało jeszcze odwiedzone
$h[x, y] = i$: pole $\langle x, y\rangle$ zostało odwiedzone w i-tym ruchu $(1 \leqslant i \leqslant n^2)$ (3.26)

Kolejna decyzja dotyczy wyboru parametrów. Powinny one określać warunki początkowe dla następnego ruchu, a także przekazywać informacje o tym, czy ruch był poprawny. W celu realizacji pierwszego zadania wprowadzamy współrzędne x, y określające bieżącą pozycję skoczka oraz zmienną i będącą numerem kolejnego ruchu (w celu jego zapisania). W związku z drugim zadaniem wprowadzamy parametr logiczny q przekazujący wynik działania procedury: $q = true$ oznacza sukces; $q = false$ oznacza porażkę.

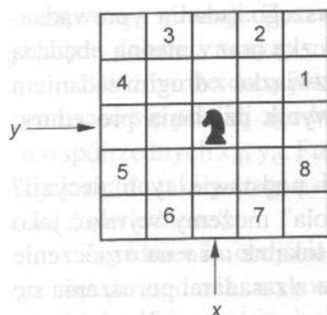
Które instrukcje mogą być sprecyzowane na podstawie tych decyzji? Z pewnością warunek „na szachownicy są wolne pola" możemy wyrazić jako „$i < n^2$". Ponadto, jeśli wprowadzimy dwie zmienne lokalne u i v na oznaczenie współrzędnych następnego pola (określonego zgodnie z zasadami poruszania się skoczka), to predykat „dopuszczalny" może być wyrażony jako logiczne połączenie warunków: (1) to nowe pole leży na szachownicy, tzn. $1 \leqslant u \leqslant n$ oraz $1 \leqslant v \leqslant n$; (2) to pole jeszcze nie zostało odwiedzone, tzn. $h[u, v] = 0$. Operację zapisania ruchu poprawnego możemy wyrazić za pomocą instrukcji przypisania $h[u, v] := i$, usunięcie zaś tego zapisu jako $h[u, v] := 0$. Jeżeli wprowadzimy zmienną lokalną $q1$ i użyjemy jej jako parametru wynikowego w rekurencyjnym wywołaniu procedury realizującej ten algorytm, to warunek „ruch był udany" możemy zastąpić zmienną $q1$. Uzyskujemy w ten sposób następujące sformułowanie algorytmu:

```
procedure próbuj(i: integer; x, y: indeks; var q: boolean);
var u, v: integer; q1: boolean;
begin zapoczątkuj wybieranie ruchów;
    repeat niech u, v będą współrzędnymi następnego ruchu określonego
        regułami szachowymi;
        if (1≤u≤n) ∧ (1≤v≤n) ∧ (h[u, v] = 0) then
        begin h[u, v] := i;                                  (3.27)
```

```
            if i<sqr(n) then
              begin próbuj(i+1, u, v, q1);
                 if ¬ q1 then h[u, v] := 0
              end else q1 := true
        end
    until q1 ∨ (nie ma następnego kandydata);
    q := q1
end
```

Kolejny krok precyzowania algorytmu doprowadzi nas do programu wyrażonego całkowicie w terminach przyjętego przez nas języka programowania. Zauważmy, że do tej pory program był konstruowany w sposób niezależny od reguł przesuwania skoczka po szachownicy. Odłożenie na później rozważań takich szczegółów było rozmyślne; teraz jednak należy je już uwzględnić.

Dla pary $\langle x, y\rangle$, określającej bieżącą pozycję skoczka, istnieje potencjalnie osiem par $\langle u, v\rangle$ opisujących następną pozycję. Na rysunku 3.8 zostały one ponumerowane liczbami od 1 do 8.



RYSUNEK 3.8
Osiem możliwych ruchów skoczka

Prostym sposobem obliczenia u i v w zależności od x i y jest dodanie do x i y różnic współrzędnych, zapamiętanych uprzednio parami w jednej tablicy lub pojedynczo w dwóch tablicach. Oznaczmy przez a i b te dwie odpowiednio zapoczątkowane tablice różnic. Licznika k użyjemy do zliczania „kolejnych kandydatów". Program 3.3 stanowi końcową wersję programu „droga skoczka szachowego".

PROGRAM 3.3
Droga skoczka szachowego

```
program drogaskoczkaszachowego(output);
const n=5; nsq=25;
type indeks=1..n;
var i, j: indeks;
    q: boolean;
    s: set of indeks;
```

```
    a, b: array [1..8] of integer;
    h: array [indeks, indeks] of integer;

procedure próbuj(i: integer; x, y: indeks; var q: boolean);
   var k, u, v: integer; q1: boolean;
begin k := 0;
   repeat k := k+1; q1 := false;
      u := x+a[k]; v := y+b[k];
      if (u in s) ∧ (v in s) then
      if h[u, v] = 0 then
      begin h[u, v] := i;
         if i < nsq then
           begin próbuj(i+1, u, v, q1);
              if ¬ q1 then h[u, v] := 0
           end else q1 := true
      end
   until q1 ∨ (k=8);
   q := q1
end {próbuj};

begin s := [1, 2, 3, 4, 5];
   a[1] :=  2; b[1] :=  1;
   a[2] :=  1; b[2] :=  2;
   a[3] := -1; b[3] :=  2;
   a[4] := -2; b[4] :=  1;
   a[5] := -2; b[5] := -1;
   a[6] := -1; b[6] := -2;
   a[7] :=  1; b[7] := -2;
   a[8] :=  2; b[8] := -1;
   for i := 1 to n do
      for j := 1 to n do h[i, j] := 0;
   h[1, 1] := 1; próbuj(2, 1, 1, q);
   if q then
      for i := 1 to n do
      begin for j := 1 to n do write(h[i, j]: 5);
         writeln
      end
   else writeln('NIE ISTNIEJE ROZWIĄZANIE')
end.
```

Procedurę rekurencyjną wywołuje się po raz pierwszy z parametrami x_0, y_0 określającymi współrzędne pola początkowego. Polu temu musi być przypisana wartość 1; pozostałe pola muszą być oznaczone jako wolne.

$h[x_0, y_0] := 1$; *próbuj* $(2, x_0, y_0, q)$

TABLICA 3.1
Trzy drogi skoczka szachowego

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

Nie możemy przeoczyć jeszcze jednego szczegółu: zmienna $h[u, v]$ jest określona tylko wtedy, gdy u i v mają wartości w przedziale $1 \dots n$. W konsekwencji wyrażenie z formuły (3.27), wstawione na miejsce warunku „dopuszczalny", jest określone, jeśli dwa pierwsze czynniki koniunkcji są prawdziwe. W programie 3.3 uwzględniono już poprawne sformułowanie tego warunku. Co więcej, podwójna relacja $1 \leqslant u \leqslant n$ została zastąpiona przez wyrażenie u **in** $[1, 2, \dots, n]$, które dla dostatecznie małej wartości n może być efektywniejsze (zob. p. 1.10.3). W tablicy 3.1 przedstawiono rozwiązania dla $n=5$ i pozycji wyjściowych $\langle 1, 1\rangle$ i $\langle 3, 3\rangle$ oraz dla $n=6$ i pozycji wyjściowej $\langle 1, 1\rangle$.

Parametr wynikowy q i zmienną lokalną $q1$ można zastąpić zmienną globalną, co uprościłoby nieco program.

Jakiego rodzaju ogólne wnioski można wyciągnąć z tego przykładu? Jakie jego cechy sprawiają, że jest on typowy dla omawianej klasy algorytmów? Czego ten przykład nas nauczył? Jego cechą charakterystyczną jest to, że kolejne kroki, które mogłyby doprowadzić do rozwiązania końcowego, są zapisywane, później zaś, w chwili ujawnienia faktu, że konkretny krok nie zmierza do rozwiązania końcowego (prowadzi w „ślepą uliczkę"), odpowiednie zapisy są usuwane. Algorytmy działające wg powyższej zasady są nazywane **algorytmami z powrotami** (ang. *backtracking algorithms*). Wzorzec ogólny algorytmu (3.28) uzyskuje się z procedury (3.24) przy założeniu, że liczba potencjalnych kandydatów w każdym kroku jest skończona.

```
procedure próbuj;
begin zapoczątkuj wybieranie kandydatów;
   repeat wybierz następnego kandydata;
      if dopuszczalny then
      begin zapisz go;
         if rozwiązanie niepełne then
         begin próbuj wykonać następny krok;                    (3.28)
```

```
            if próba nieudana then usuń zapis
        end
    end
  until próba udana ∨ nie ma następnego kandydata
end
```

Programy rzeczywiste mogą oczywiście przybierać postaci pochodne od schematu (3.28). W jednym z często spotykanych schematów pochodnych używa się parametru jawnie określającego głębokość rekursji, co pozwala na proste sformułowanie warunku zakończenia.

Jeśli ponadto w każdym kroku liczba kandydatów badanych jest ustalona i wynosi, powiedzmy, m, to uzyskujemy schemat działania określony przez algorytm (3.29); jest on uaktywniany za pomocą instrukcji „*próbuj*(1)".

```
procedure próbuj(i: integer);
  var k: integer;
begin k := 0;
  repeat k := k+1; wybierz k-tego kandydata;
    if dopuszczalny then
    begin zapisz go;
      if i < n then
      begin próbuj(i+1);                                    (3.29)
        if próba nieudana then usuń zapis
      end
    end
  until próba udana ∨ (k=m)
end
```

W dalszej części tego rozdziału omówimy trzy inne przykłady. Ujawniają one różne wcielenia schematu abstrakcyjnego (3.29) i mogą służyć jako ilustracje właściwego użycia rekursji.

3.5. Problem ośmiu hetmanów

Problem ośmiu hetmanów jest dobrze znanym przykładem zastosowania metody prób i błędów oraz algorytmów z powrotami. Zajmował się nim C.F. Gauss w 1850 r., lecz nie znalazł pełnego rozwiązania. Nie powinno to jednak nikogo dziwić. Charakterystyczną właściwością omawianych problemów jest to, że nie dają się rozwiązać analitycznie. Wymagają one natomiast dużo mozolnego wysiłku, cierpliwości i dokładności. Algorytmy z powrotami uzyskały znaczenie prawie wyłącznie dzięki maszynom cyfrowym, które z wymienionymi właściwościami radzą sobie znacznie lepiej niż ludzie (nawet genialni).

Problem ośmiu hetmanów można sformułować następująco (zob. także [3.4]): ustawić na szachownicy osiem hetmanów w taki sposób, aby żaden z nich nie szachował żadnego innego.

Używając jako szablonu schematu (3.29), łatwo otrzymujemy następujące, „surowe" jeszcze rozwiązanie programowe:

```
procedure próbuj(i: integer);
begin
   zapoczątkuj wybieranie pozycji i-tego hetmana;
   repeat dokonaj następnego wyboru pozycji;
      if bezpieczna then
      begin ustaw hetmana;
         if i < 8 then                                        (3.30)
         begin próbuj(i+1);
            if próba nieudana then usuń hetmana
         end
      end
   until próba udana ∨ nie ma więcej pozycji
end
```

Przed dalszym precyzowaniem programu konieczne jest podjęcie pewnych decyzji dotyczących reprezentowania danych. Ponieważ zgodnie z regułami szachowymi hetman szachuje wszystkie inne figury ustawione w tej samej kolumnie, wierszu lub na przekątnej, można wnioskować, że każda kolumna może zawierać dokładnie jednego hetmana oraz że wybór pozycji i-tego hetmana jest ograniczony do i-tej kolumny. Parametr i jest więc indeksem kolumny, a wybraną (spośród możliwych ośmiu wartości) pozycję w tej kolumnie oznaczamy za pomocą indeksu wierszy j. Pozostał jeszcze problem reprezentowania ośmiu hetmanów na szachownicy. Narzucającym się rozwiązaniem jest znowu macierz kwadratowa, ale po bliższej analizie okazuje się, że prowadziłoby to do dość nieporęcznego testowania możliwości ustawienia hetmana na szachownicy. Byłoby to wielce niepożądane, jeśli weźmiemy pod uwagę fakt, że jest to najczęściej wykonywana operacja. Dlatego też powinniśmy wybrać taką reprezentację danych, która maksymalnie uprościłaby operację testowania. Najlepszą receptą na to jest reprezentowanie całej informacji istotnej i często używanej w sposób pozwalający na bezpośredni dostęp. W naszym przypadku informacją tą nie są pozycje hetmanów, ale fakt, czy w odpowiednim wierszu lub na przekątnej został już ustawiony hetman. (Wiemy już, że dokładnie jeden hetman będzie umieszczony w każdej kolumnie k, gdzie $1 \leqslant k \leqslant 1$). Rozważania te prowadzą do deklaracji następujących zmiennych:

```
var x: array [1..8] of integer;
    a: array [1..8] of boolean;                               (3.31)
    b: array [b1..b2] of boolean;
    c: array [c1..c2] of boolean;
```

gdzie

$x[i]$ oznacza pozycję hetmana w i-tej kolumnie;
$a[j]$ oznacza brak hetmana w j-tym wierszu;
$b[k]$ oznacza brak hetmana na k-tej przekątnej o kierunku ↙;
$c[k]$ oznacza brak hetmana na k-tej przekątnej o kierunku ↘.

Wybór granic indeksów $b1$, $b2$, $c1$, $c2$ jest podyktowany sposobem obliczania indeksów tablic b i c; zauważmy, że wszystkie pola leżące na przekątnej o kierunku mają stałą sumę współrzędnych $i+j$, podczas gdy pola leżące na przekątnej o kierunku mają stałą różnicę współrzędnych $i-j$. Odpowiednie rozwiązanie można znaleźć w programie 3.4.

Przy powyższych założeniach instrukcja „ustaw hetmana” jest sprecyzowana jako

$$x[i] := j;\ a[j] := false;\ b[i+j] := false;\ c[i-j] := false \tag{3.32}$$

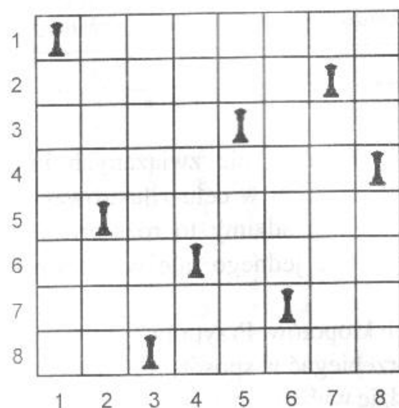
Instrukcja „usuń hetmana” przybiera postać

$$a[j] := true;\ b[i+j] := true;\ c[i-j] := true \tag{3.33}$$

a warunek „bezpieczna” jest spełniony, jeśli pole $\langle i, j\rangle$ leży w wierszu i na przekątnych, które są wolne (odpowiada to wartości *true*). Dlatego warunek ten może być zapisany w postaci wyrażenia

$$a[j] \wedge b[i+j] \wedge c[i-j] \tag{3.34}$$

Kończy to proces tworzenia algorytmu, którego wersję końcową stanowi program 3.4. Znalezione przez ten program rozwiązanie $x = (1, 5, 8, 6, 3, 7, 2, 4)$ przedstawiono na rys 3.9.



RYSUNEK 3.9
Jedno z rozwiązań problemu ośmiu hetmanów

PROGRAM 3.4
Osiem hetmanów

```
program osiemhetmanów(output);
{znajdź jedno rozwiązanie problemu ośmiu hetmanów}
var i: integer; q: boolean;
   a: array [  1..8] of boolean;
   b: array [ 2..16] of boolean;
   c: array [–7..7] of boolean;
   x: array [  1..8] of integer;

procedure próbuj(i: integer; var q: boolean);
   var j: integer;
begin j := 0;
   repeat j := j+1; q := false;
      if a[j] ∧ b[i+j] ∧ c[i–j] then
      begin x[i] := j;
         a[j] := false; b[i+j] := false; c[i–j] := false;
         if i<8 then
         begin próbuj(i+1, q);
            if ¬ q then
            begin a[j] := true; b[i+j] := true; c[i–j] := true
            end
         end else q := true
      end
   until q ∨ (j = 8)
end {próbuj};

begin
   for i :=    1 to   8 do a[i] := true;
   for i :=    2 to 16 do b[i] := true;
   for i := –7 to   7 do c[i] := true;
   próbuj(1, q);
   if q then
      for i := 1 to 8 do write(x[i]: 4);
   writeln
end.
```

Przed przejściem do omawiania innych zagadnień, nie związanych już z szachownicą, użyjmy przykładu z ośmioma hetmanami w celu zilustrowania ważnego rozszerzenia metody prób i błędów. Wprowadzimy to rozszerzenie w celu znajdowania przez odpowiedni algorytm nie jednego, ale wszystkich rozwiązań.

Rozszerzenie metody nie sprawia większych kłopotów. Przypomnijmy sobie fakt, że wybieranie nowych kandydatów musi przebiegać w sposób systematyczny, gwarantujący, że żaden z kandydatów nie będzie wybrany więcej niż jeden raz.

Ta właściwość algorytmu odpowiada takiemu przeglądaniu drzewa kandydatów, w którym każdy wierzchołek jest odwiedzany dokładnie raz. Pozwala ona – po znalezieniu kolejnego rozwiązania i zanotowaniu go – na bezpośrednie kontynuowanie procesu wybierania. Schemat ogólny (3.35) wyprowadzono z algorytmu (3.29).

```
procedure próbuj(i: integer);
   var k: integer;
begin
   for k := 1 to m do                                          (3.35)
   begin wybierz k-tego kandydata;
      if dopuszczalny then
      begin zapisz go;
         if i<n then próbuj(i+1) else wydrukuj rozwiązanie;
         usuń zapis
      end
   end
end
```

Zauważmy, że uproszczenie warunku zakończenia procesu wybierania kolejnego kandydata do pojedynczego członu $k=m$ pozwoliło zastąpić instrukcję **repeat** odpowiednią instrukcją **for**. Swojego rodzaju niespodziankę stanowi fakt, że znalezienie wszystkich rozwiązań jest realizowane przez program prostszy od programu znajdującego tylko jedno rozwiązanie.

Rozszerzony algorytm znajdujący wszystkie 92 rozwiązania problemu ośmiu hetmanów przedstawiono w postaci programu 3.5. W rzeczywistości istnieje tylko 12 istotnie różnych rozwiązań; program 3.5 nie rozpoznaje rozwiązań symetrycznych. Pierwszych 12 rozwiązań znalezionych przez program zamieszczono w tabl. 3.2. Liczby N znajdujące się w kolumnie po prawej stronie tablicy oznaczają częstość wykonywania warunku testującego bezpieczeństwo ustawienia hetmana. Średnia wartość N dla wszystkich 92 rozwiązań wynosi 161.

TABLICA 3.2
Dwanaście rozwiązań problemu ośmiu hetmanów

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

PROGRAM 3.5
Osiem hetmanów

```
program osiemhetmanów(output);
var i: integer;
    a: array [  1..  8] of boolean;
    b: array [  2..16] of boolean;
    c: array [-7..  7] of boolean;
    x: array [  1..  8] of integer;

procedure drukuj;
    var k: integer;
begin for k := 1 to 8 do write(x[k]: 4);
    writeln
end {drukuj};

procedure próbuj(i: integer);
    var j: integer;
begin
    for j := 1 to 8 do
        if a[j] ∧ b[i+j] ∧ c[i-j] then
        begin x[i] := j;
            a[j] := false; b[i+j] := false; c[i-j] := false;
            if i<8 then próbuj(i+1) else drukuj;
            a[j] := true; b[i+j] := true; c[i-j]true;
        end
end {próbuj};

begin
    for i :=   1 to   8 do a[i] := true;
    for i :=   2 to 16 do b[i] := true;
    for i := -7 to   7 do c[i] := true;
    próbuj(1)
end.
```

3.6. Problem trwałego małżeństwa

Załóżmy, że dane są dwa zbiory A i B o tej samej mocy n. Problem: znaleźć zbiór n par $\langle a, b\rangle$ takich, że a **in** A i b **in** B oraz a i b spełniają pewne ograniczenia. Istnieje wiele różnych kryteriów ograniczających; jednym z nich jest „reguła trwałego małżeństwa".

Załóżmy, że A jest zbiorem mężczyzn, B zaś zbiorem kobiet. Każdy mężczyzna i każda kobieta mają ustalone preferencje dotyczące swoich partnerów. Jeśli n par zostało dobranych tak, że istnieje mężczyzna i kobieta, którzy nie stanowią małżeństwa, ale mężczyzna przedkłada tę kobietę nad aktualną żonę,

a kobieta przedkłada tego mężczyznę nad aktualnego męża, to dobór małżeństw będziemy nazywać nietrwałym. Jeżeli żadna taka para nie istnieje, to dobór będziemy nazywać **trwałym**.

Sytuacja taka jest charakterystyczna dla wielu podobnych problemów, w których trzeba przyjąć pewne ustalenia (dobór) na podstawie istniejących preferencji, np. wybór szkoły przez studentów, dobór rekrutów do różnych jednostek wojska itd. Przykład z małżeństwami jest szczególnie intuicyjny; zauważmy jednak, że określona lista preferencji nie zmienia się po dokonaniu kolejnego ustalenia (przypisania). Założenie to wprawdzie upraszcza problem, ale z drugiej strony przedstawia wypaczony obraz rzeczywistości.

Jednym ze sposobów znalezienia rozwiązania jest próba łączenia w pary elementów obu zbiorów dopóty, dopóki się ich nie wyczerpie. Przystąpmy do znalezienia wszystkich trwałych małżeństw. Korzystając ze schematu programu (3.35), z łatwością możemy naszkicować rozwiązanie. Przez *próbuj*(*m*) oznaczmy algorytm znajdujący partnerkę dla mężczyzny *m* i niech szukanie to przebiega wg listy ustalonych wcześniej preferencji mężczyzny. Pierwsza wersja programu oparta na tych założeniach ma postać następującą:

```
procedure próbuj(m: mężczyzna);
   var p: pozycja;
begin
   for p := 1 to n do
   begin wybierz p-tą preferencję mężczyzny m;
      if dopuszczalna then
      begin zanotuj małżeństwo;                                    (3.36)
         if m nie jest ostatnim mężczyzną then próbuj(succ(m))
         else zanotuj zbiór trwałych małżeństw;
         wykreśl małżeństwo
      end
   end
end
```

Ponownie znaleźliśmy się w sytuacji, w której dalszy proces precyzowania jest niemożliwy bez podjęcia pewnych decyzji dotyczących reprezentowania danych. Wprowadzimy trzy typy skalarne i, w celu uproszczenia, wartości ich będziemy oznaczać za pomocą liczb całkowitych od 1 do *n*. Chociaż te trzy typy są formalnie identyczne, dla uzyskania większej czytelności będziemy je oznaczać różnymi nazwami. W szczególności będzie można dzięki temu łatwiej zauważyć, co dana zmienna oznacza.

```
type mężczyzna = 1..n;
     kobieta   = 1..n;                                             (3.37)
     pozycja   = 1..n;
```

Dane początkowe są reprezentowane za pomocą dwóch macierzy określających preferencje mężczyzn i kobiet.

var *kmp*: **array** [*mężczyzna, pozycja*] **of** *kobieta*; (3.38)
mkp: **array** [*kobieta, pozycja*] **of** *mężczyzna*;

I tak, *kmp*[*m*] stanowi listę preferencji mężczyzn *m*, tzn. $kmp[m]\,[p]=kmp[m, p]$ jest kobietą o pozycji *p* na liście preferencji mężczyzny *m*. Podobnie, *mkp*[*k*] jest listą preferencji kobiety *k*, a *mkp*[*k, p*] określa jej *p*-ty wybór.

Wyniki będą reprezentowane za pomocą tablicy kobiet – *x*, gdzie *x*[*m*] oznacza partnerkę mężczyzny *m*. W celu zachowania symetrii – zwanej także „równouprawnieniem" – pomiędzy mężczyznami i kobietami wprowadzono dodatkową tablicę *y*, przy czym *y*[*k*] oznacza partnera kobiety *k*.

var *x*: **array** [*mężczyzna*] **of** *kobieta*; (3.39)
y: **array** [*kobieta*] **of** *mężczyzna*;

Jest oczywiste, że tablica *y* nie jest koniecznie potrzebna, skoro służy do przechowywania informacji dostępnej już pośrednio z tablicy *x*. W rzeczywistości dla wszystkich żonatych mężczyzn *m* i zamężnych kobiet *k* spełnione są następujące relacje:

$$x[y[k]]=k, \qquad y[x[m]]=m \tag{3.40}$$

Wartość *y*[*k*] można więc obliczyć przez proste przejrzenie tablicy *x*; wprowadzenie tablicy *y* zwiększa jednak istotnie efektywność algorytmu. Informacja reprezentowana przez *x* i *y* jest potrzebna do określenia trwałości proponowanego zbioru małżeństw. Ponieważ zbiór ten jest konstruowany stopniowo przez wybranie pary dwojga ludzi, a następnie testowanie trwałości ewentualnego małżeństwa, więc tablice *x* i *y* są potrzebne nawet wtedy, gdy nie są jeszcze w pełni zdefiniowane. W celu zaznaczenia, które elementy tych tablic zostały określone, wprowadzimy dwie tablice logiczne:

kawaler: **array** [*mężczyzna*] **of** *boolean* (3.41)
panna: **array** [*kobieta*] **of** *boolean*

przy czym

$\neg$ *kawaler* [*m*] oznacza, że *x*[*m*] jest określone,
$\neg$ *panna* [*k*] oznacza, że *y*[*k*] jest określone.

Bliższa analiza proponowanego algorytmu ujawnia jednak, że stan cywilny mężczyzny jest określony przez wartość *m* w prosty sposób, mianowicie

$$\neg\, kawaler[i] \equiv i<m \tag{3.42}$$

Pozwala to nam uniknąć wprowadzenia tablicy *kawaler*.

Dochodzimy zatem do sprecyzowania programu (3.43). Predykat *dopuszczalna* możemy zastąpić koniunkcją wartości logicznych *panna* i *trwałe*; dokładną definicję funkcji *trwałe* podamy później.

```
procedure próbuj(m: mężczyzna);
  var p: pozycja; k: kobieta;
begin for p := 1 to n do;
  begin k := kmp[m, p];
    if panna [k] ∧ trwałe then
    begin x[m] := k; y[k] := m; panna[k] := false;
      if m<n then próbuj(succ(m))                      (3.43)
      else zanotuj zbiór trwałych małżeństw;
      panna[k] := true
    end
  end
end
```

Warto tutaj zwrócić uwagę na duże podobieństwo powyższego programu do programu 3.5.

Kluczowym zadaniem jest teraz sprecyzowanie pojęcia trwałości. Pojęcia tego nie można jednak wyrazić za pomocą tak prostego wyrażenia jak w przypadku definiowania bezpieczeństwa pozycji hetmana w programie 3.5. Pierwsza rzecz, jaką powinniśmy zauważyć, to fakt, że trwałość wynika na podstawie definicji z porównania preferencji lub pozycji. Pozycje mężczyzn lub kobiet nie są jednak dostępne w sposób jawny za pośrednictwem żadnej z dotychczas wprowadzonych struktur danych. Oczywiście można obliczyć pozycję kobiety k w świadomości mężczyzny m, ale wymaga to kosztownego odszukania k w tablicy *kmp*[m].

Ponieważ obliczanie trwałości jest operacją wykonywaną bardzo często, wydaje się celowe przechowywanie tej informacji w sposób bardziej bezpośredni. Z tego powodu wprowadzimy dwie tablice:

```
pmk: array [mężczyzna, kobieta] of pozycja;            (3.44)
pkm: array [kobieta, mężczyzna] of pozycja
```

takie, że *pmk*[m, k] oznacza pozycję kobiety k na liście preferencji mężczyzny m, a *pkm*[k, m] oznacza pozycję mężczyzny m na liście kobiety k. Jasne jest, że wartości tych pomocniczych tablic są stałe i można je obliczyć na początku na podstawie wartości *kmp* i *mkp*.

Określając predykat *trwałe*, skorzystamy z jego początkowej definicji. Przypominamy, że badamy możliwość powiązania mężczyzny m i kobiety k, gdzie $k = kmp[m, p]$, tj. k ma pozycję p na liście mężczyzny m. Załóżmy optymistycznie, że trwałość całego zbioru zostaje zachowana, a następnie spróbujmy znaleźć możliwe przyczyny kłopotów. Gdzie one mogą być ukryte? Istnieją dwie symetryczne możliwości:

(1) Może istnieć kobieta *pk*, która przedkłada mężczyznę *m* nad swojego męża i którą *m* przedkłada nad kobietę *k*.
(2) Może istnieć mężczyzna *pm*, który przedkłada kobietę *k* nad swoją żonę i którego *k* przedkłada nad mężczyznę *m*.

W przypadku (1) porównujemy pozycje *pkm*[*pk*, *m*] i *pkm*[*pk*, *y*[*pk*]] dla wszystkich kobiet przedkładanych nad *k* przez *m*, tj. dla wszystkich $pk = kmp[m, i]$ takich, że $i < p$. Zauważmy, że wszystkie te kobiety-kandydatki są już zamężne (jeśliby któraś z nich była jeszcze panną, to *m* by ją wybrał). Testowanie trwałości możemy w tym przypadku sformułować w postaci prostego przeglądania liniowego; *t* oznacza trwałość.

```
t := true; i := 1;
while (i<p) ∧ t do
   begin pk := kmp[m, i]; i := i+1;                              (3.45)
      if ¬ panna[pk] then t := pkm[pk, m]<pkm[pk, y[pk]]
   end
```

W przypadku (2) musimy zbadać wszystkich kandydatów *pm*, których *k* przedkłada nad aktualnego męża *m*, tj. wszystkich mężczyzn $pm = mkp[k, i]$ takich, że $i < pkm[k, m]$. Przez analogię do przypadku (1), należy porównać pozycje *pmk*[*pm*, *k*] i *pmk*[*pm*, *x*[*pm*]]. Musimy jednak zachować ostrożność i omijać porównania pociągające za sobą odwołania do *x*[*pm*], gdzie *pm* jest jeszcze kawalerem. Trzeba w tym celu sprawdzić, czy $pm < m$ (wszyscy mężczyźni poprzedzający *m* są już żonaci).

Pełny algorytm przedstawiono w programie 3.6. Tablica 3.3 zawiera przykładowe dane wejściowe reprezentujące tablice *kmp* i *mkp*. Dziewięć trwałych rozwiązań obliczonych przez ten program pokazano w tabl. 3.4.

PROGRAM 3.6
Trwałe małżeństwa

```
program małżeństwo(input, output);
{problem trwałych małżeństw}
const n=8;
type mężczyzna=1..n; kobieta=1..n; pozycja=1..n;
var m: mężczyzna; k: kobieta; p: pozycja;
  kmp: array [mężczyzna, pozycja] of kobieta;
  mkp: array [kobieta, pozycja] of mężczyzna;
  pmk: array [mężczyzna, kobieta] of pozycja;
  pkm: array [kobieta, mężczyzna] of pozycja;
  x: array [mężczyzna] of kobieta;
  y: array [kobieta] of mężczyzna;
  panna: array [kobieta] of boolean;
procedure drukuj;
  var m: mężczyzna; pm, pk: integer;
```

```
begin pm := 0; pk := 0;
  for m := 1 to n do
  begin write(x[m]: 4);
    pm := pm + pmk[m, x[m]]; pk := pk + pkm[x[m], m]
  end;
writeln(pm: 8, pk: 4);
end {drukuj};

procedure próbuj(m: mężczyzna);
  var p: pozycja; k: kobieta;
  function trwałe: boolean;
    var t: boolean; i, lim: pozycja;
      pm: mężczyzna; pk: kobieta;

  begin t := true; i := 1;
    while (i<p) ∧ t do
    begin pk := kmp[m, i]; i := i+1;
      if ¬ panna[pk] then t := pkm[pk, m] < pkm[pk, y[pk]]
    end;
    i := 1; lim := pkm[k, m];
    while (i<lim) ∧ t do
    begin pm := mkp[k, i]; i := i+1;
      if pm<m then t := pmk[pm, k] > pmk[pm, x[pm]]
    end;
    trwałe := t
  end {trwałe};
begin {próbuj}
  for p := 1 to n do
  begin k := kmp[m, p];
    if panna[k] then
      if trwałe then
      begin x[m] := k; y[k] := m; panna[k] := false;
        if m<n then próbuj(succ(m)) else drukuj;
        panna[k] := true
      end
  end
end {próbuj};
begin {program główny}
  for m := 1 to n do
    for p := 1 to n do
    begin read(kmp[m, p]); pmk[m, kmp[m, p]] := p
    end;
  for k := 1 to n do
    for p := 1 to n do
```

```
        begin read(mkp[k, p]); pkm[k, mkp[k, p]] := p
        end;
    for k := 1 to n do panna[k] := true;
    próbuj(1)
end.
```

TABLICA 3.3
Przykładowe dane wejściowe dla programu trwałych małżeństw

Pozycja		1	2	3	4	5	6	7	8
Mężczyzna	1 wybiera kobietę	7	2	6	5	1	3	8	4
	2	4	3	2	6	8	1	7	5
	3	3	2	4	1	8	5	7	6
	4	3	8	4	2	5	6	7	1
	5	8	3	4	5	6	1	7	2
	6	8	7	5	2	4	3	1	6
	7	2	4	6	3	1	7	5	8
	8	6	1	4	2	7	5	3	8
Kobieta	1 wybiera mężczyznę	4	6	2	5	8	1	3	7
	2	8	5	3	1	6	7	4	2
	3	6	8	1	2	3	4	7	5
	4	3	2	4	7	6	8	5	1
	5	6	3	1	4	5	7	2	8
	6	2	1	3	8	7	4	6	5
	7	3	5	7	2	4	1	8	6
	8	7	2	8	4	5	6	3	1

TABLICA 3.4
Rozwiązania znalezione przez program trwałych małżeństw

		x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	*pm*	*pk*	*c**
Rozwiązanie	1	7	4	3	8	1	5	2	6	16	32	21
	2	2	4	3	8	1	5	7	6	22	27	449
	3	2	4	3	1	7	5	8	6	31	20	59
	4	6	4	3	8	1	5	7	2	26	22	62
	5	6	4	3	1	7	5	8	2	35	15	47
	6	6	3	4	8	1	5	7	2	29	20	143
	7	6	3	4	1	7	5	8	2	38	13	47
	8	3	6	4	8	1	5	7	2	34	18	758
	9	3	6	4	1	7	5	8	2	43	11	34

**c* = liczba testowań warunku trwałości
Rozwiązanie 1 = optymalne męskie rozwiązanie
Rozwiązanie 9 = optymalne kobiece rozwiązanie

Przedstawiony algorytm zbudowano na podstawie prostego schematu algorytmu z powrotami. Jego efektywność zależy w sposób zasadniczy od „zmyślności" schematu ograniczającego rozrost drzewa rozwiązań. Nieco szybszy, ale bardziej złożony i mniej przejrzysty algorytm został przedstawiony przez McVitiego i Wilsona [3.1] i [3.2], którzy także rozszerzyli ten algorytm na przypadek zbiorów (kobiet i mężczyzn) nierównolicznych.

Dwa ostatnie przykłady były związane ze znalezieniem wszystkich rozwiązań problemów (przy podanych pewnych ograniczeniach). W praktyce często stosuje się algorytmy wybierające jedno lub kilka rozwiązań, w pewnym sensie optymalnych. Na przykład byłoby interesujące znalezienie rozwiązania, które przeciętnie najlepiej zadowala mężczyzn lub kobiety, lub wszystkie osoby.

Zauważmy, że w tabl. 3.4 pokazano sumy pozycji wszystkich kobiet na listach (preferencji) ich mężów oraz sumy pozycji wszystkich mężczyzn na listach (preferencji) ich żon. Są to

$$pm = \sum_{m=1}^{n} pmk[m, x[m]], \qquad pk = \sum_{m=1}^{n} pkm[x[m], m] \tag{3.46}$$

Rozwiązanie o najmniejszej wartości *pm* zwane jest optymalnym męskim rozwiązaniem trwałym; rozwiązanie o najmniejszej wartości *pk* zwane jest optymalnym kobiecym rozwiązaniem trwałym. Charakterystyczną cechą obranej strategii szukania rozwiązań jest to, że rozwiązania dobre z męskiego punktu widzenia znajdują się na początku, a rozwiązania dobre z kobiecego punktu widzenia pojawiają się na końcu. Przyjęty algorytm jest stronniczy w stosunku do męskiej populacji. Można to łatwo zmienić, stosując systematyczną zamianę roli mężczyzny i kobiety, tj. wzajemną zamianę jedynie *mkp* i *kmp* oraz *pmk* i *pkm*.

Powstrzymamy się od dalszego rozszerzania tego programu, a poszukiwanie optymalnego rozwiązania zostanie zrealizowane dla kolejnego, ostatniego już przykładu algorytmu z powrotami.

3.7. Problem optymalnego wyboru

Ostatni przykład algorytmu z powrotami jest logicznym rozszerzeniem dwóch poprzednich przykładów o ogólnym schemacie (3.35). Początkowo stosowaliśmy regułę powracania w celu znalezienia *pojedynczego* rozwiązania danego problemu. Odpowiednie przykłady to określenie drogi skoczka szachowego i rozmieszczenie ośmiu hetmanów. Następnie postanowiliśmy znaleźć *wszystkie* rozwiązania danego problemu; przykłady – osiem hetmanów i trwałe małżeństwa. Teraz zależeć nam będzie na znalezieniu **rozwiązania optymalnego.**

W celu znalezienia tego rozwiązania należy generować wszystkie możliwe rozwiązania i w trakcie tej czynności wybrać jedno, w pewnym sensie optymalne. Zakładając, że optymalność jest zdefiniowana za pomocą pewnej funkcji $f(x)$ o wartościach dodatnich, można uzyskać odpowiedni algorytm, zastępując w schemacie (3.35) instrukcję *drukuj rozwiązanie* instrukcją

$$\textbf{if } f(\textit{rozwiązanie}) > f(\textit{optimum}) \textbf{ then } \textit{optimum} := \textit{rozwiązanie} \tag{3.47}$$

Zmienna *optimum* służy do zapamiętania najlepszego z dotychczas znalezionych rozwiązań. Rzecz jasna należy jej nadać poprawną wartość początkową; ponadto, aby uniknąć wielokrotnego obliczania wartości *f(optimum)*, wartość tę pamięta się zazwyczaj za pomocą innej zmiennej.

Przykładem ogólnego problemu znajdowania rozwiązania optymalnego jest następujące zagadnienie – dość często spotykane i mające duże znaczenie praktyczne: dla określonego zbioru obiektów znaleźć podzbiór (wybór) optymalny, uwzględniając pewne ograniczenia. Podzbiory stanowiące rozwiązania tworzy się stopniowo, rozważając pojedyncze obiekty ze zbioru podstawowego. Procedura *próbuj* opisuje proces badania możliwości dołączenia do podzbioru pojedynczego obiektu; wywołuje się ją rekurencyjnie (w celu zbadania następnego obiektu) dopóty, dopóki nie przejrzy się wszystkich obiektów.

Zauważmy, że przeglądanie każdego obiektu (nazywanego kandydatem w poprzednich przykładach) może dać dwa wyniki: *włączenie* badanego obiektu do aktualnego podzbioru lub jego *wykluczenie*. Ponieważ te dwa przypadki należy rozpisać explicite, użycie instrukcji **repeat** lub instrukcji **for** jest niewłaściwe. Odpowiedni schemat rekursji pokazano w procesie (3.48) (zakładamy, że obiekty są ponumerowane liczbami 1, 2, ..., *n*)

```
procedure próbuj(i: integer);
begin
1: if włączenie obiektu jest akceptowalne then
   begin dołącz i-ty obiekt;
      if i<n then próbuj(i+1) else sprawdź optymalność;        (3.48)
      usuń i-ty obiekt
   end;
2: if wykluczenie obiektu jest akceptowalne then
      if i<n then próbuj(i+1) else sprawdź optymalność
end
```

Ze schematu tego wyraźnie widać, że istnieje 2^n możliwych podzbiorów; w celu radykalnego zmniejszenia liczby badanych elementów musimy zastosować właściwe kryterium akceptowalności. Chcąc wyjaśnić ten proces, zajmiemy się konkretnym przykładem problemu optymalnego wyboru. Niech każdy z *n* obiektów $a_1, ..., a_n$ będzie scharakteryzowany za pomocą optymalnej wagi *w* i wartości *v*. Przez **podzbiór optymalny** będziemy rozumieć podzbiór o największej sumie wartości jego składowych, spełniający jednocześnie ograniczenie nałożone na łączną wagę tych składowych. Jest to problem dobrze znany wszystkim podróżnym, którzy pakując swoje walizki, dokonują wyboru spośród *n* przedmiotów w taki sposób, aby ogólna wartość przewożonych rzeczy była optymalna, a łączna ich waga nie przekraczała dopuszczalnego limitu.

Nadszedł moment podjęcia decyzji dotyczącej sposobu reprezentowania opisanych faktów za pośrednictwem danych. Na podstawie poprzednich rozważań wprowadzamy następujące deklaracje:

```
type indeks = 1..n;
     obiekt = record w, v: integer end;
var  a: array [indeks] of obiekt;                    (3.49)
     limw, całkv, maxv: integer;
     s, opts: set of indeks;
```

Zmienne *limw* i *całkv* oznaczają limit wagowy i wartość całkowitą wszystkich *n* obiektów. Wartości tych zmiennych nie ulegają zmianie podczas całego procesu wybierania. Zmienna *s* reprezentuje aktualnie zbudowany podzbiór obiektów, w którym każdy obiekt jest reprezentowany przez jego nazwę (indeks). Zmienna *opts* służy do pamiętania optymalnego z dotychczas zbadanych wyborów, *maxv* zaś oznacza wartość tego wyboru.

Co będzie stanowić kryterium akceptowalności kolejnego obiektu? Jeśli rozważamy *włączenie*, to obiekt będzie wybrany wtedy, gdy nie wystąpi przekroczenie limitu wagowego. W przypadku przekroczenia możemy przerwać dołączanie dalszych obiektów. Jeśli jednak rozważamy *wykluczenie*, to kryterium akceptowalności, decydujące o tym, czy warto rozszerzyć aktualny podzbiór, możemy określić następująco: łączna wartość obiektów, którą można uzyskać po wykluczeniu, powinna być nie mniejsza niż do tej pory znaleziona wartość optymalna. Jeśli jest mniejsza, to kontynuowanie rozszerzania aktualnego podzbioru, aczkolwiek może doprowadzić do znalezienia jakichś rozwiązań, nie da na pewno rozwiązania optymalnego. Dlatego też dalsze szukanie na tej drodze jest bezowocne. Na podstawie tych dwóch warunków akceptacji możemy określić wielkości, które należy obliczać dla każdego kolejnego kroku procesu wyboru; tutaj są to

(1) całkowita waga *cw* aktualnie skonstruowanego podzbioru *s*;
(2) wartość *mv*, którą można jeszcze uzyskać przez dalsze rozszerzanie podzbioru *s*.

Te dwie wielkości będą reprezentowane przez parametry procedury *próbuj*.

Warunek *włączenie obiektu jest akceptowalne* można teraz sformułować następująco:

$$cw + a[i].w \leqslant limw \tag{3.50}$$

a odpowiednie sprawdzenie optymalności wyboru przybierze postać

```
if mv > maxv then
  begin {nowe optimum, zapamiętaj je}                (3.51)
    opts := s; maxv := mv
  end
```

Ostatnie przypisanie wynika z następującego rozumowania: w chwili, gdy rozważono wszystkich *n* obiektów, wartość, którą można uzyskać, jest wartością skonstruowanego podzbioru.

Warunek *wykluczenie obiektu jest akceptowalne* możemy wyrazić tak:

$$mv - a[i].v > maxv \tag{3.52}$$

Ponieważ część tego wyrażenia występuje powtórnie w programie, chcąc uniknąć jej ponownego obliczania, wartość $mv - a[i].v$ przypisano zmiennej $mv1$.

Kolejne etapy precyzowania programu: (3.48)–(3.52) oraz odpowiednie instrukcje nadające wartości początkowe zmiennym globalnym prowadzą do ostatecznej wersji programu. Zwraca uwagę łatwość wyrażenia – za pomocą operatorów teoriomnogościowych – operacji włączenia i wykluczenia obiektu. Wyniki wykonania programu 3.7 dla limitów wagowych od 10 do 120 można znaleźć w tabl. 3.5.

PROGRAM 3.7
Optymalny wybór

```
program wybór(input, output);
{znajdź optymalny wybór obiektów przy założeniu pewnych ograniczeń}
const n = 10;
type indeks = 1..n;
   obiekt = record v, w: integer end;
var i: indeks;
   a: array [indeks] of obiekt;
   limw, całkv, maxv: integer;
   w1, w2, w3: integer;
   s, opts: set of indeks;
   z: array [boolean] of char;
procedure próbuj(i: indeks; cw, mv: integer);
   var mv1: integer;
begin {spróbuj włączyć obiekt i}
   if cw + a[i].w ≤ limw then
   begin s := s + [i];
      if i < n then próbuj(i+1, cw+a[i].w, mv) else
         if mv > maxv then
         begin maxv := mv; opts := s
         end;
      s := s - [i]
   end;
   {spróbuj wykluczyć obiekt i} mv1 := mv - a[i].v;
   if mv1 > maxv then
   begin if i < n then próbuj(i+1, cw, mv1) else
         begin maxv := mv1; opts := s
         end
   end
end {próbuj};
```

```
begin całkv := 0;
  for i := 1 to n do
    with a[i] do
    begin read(w, v); całkv := całkv + v
    end;
  read(w1, w2, w3);
  z[true] := '*'; z[false] := ' ';
  write(' WAGA ');
  for i := 1 to n do write(a[i].w: 4);
  writeln; write(' WARTOŚĆ ');
  for i := 1 to n do write(a[i].v: 4);
  writeln;
  repeat limw := w1; maxv := 0; s := [ ]; opts := [ ];
    próbuj(1, 0, całkv);
    write(limw);
    for i := 1 to n do write('   ', z[i in opts]);
    writeln; w1 := w1 + w2
  until w1 > w3
end.
```

TABLICA 3.5
Przykładowe wyniki z programu optymalnego wyboru

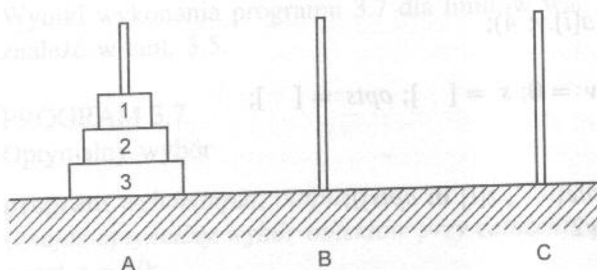
Waga Wartość	10 18	11 20	12 17	13 19	14 25	15 21	16 27	17 23	18 25	19 24
10	*									
20							*			
30					*		*			
40	*				*		*			
50	*	*		*			*			
60	*	*	*	*	*					
70	*	*			*		*		*	
80	*	*	*		*		*	*		
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*

Taki rodzaj algorytmu z powrotami, z czynnikiem ograniczającym rozrost drzewa przeglądania, zwany jest także **algorytmem podziału i ograniczeń** (ang. *branch and bound algorithm*).

Ćwiczenia

3.1

(Wieże Hanoi). Dane są trzy pręty i n krążków o różnych średnicach. Krążki mogą być nakładane na pręty, tworząc w ten sposób „wieże". Załóżmy, że n krążków zostało umieszczonych początkowo na pręcie A w kolejności malejących średnic. Na rysunku 3.10 zilustrowano tę sytuację dla $n=3$. Naszym zadaniem jest przesunięcie n krążków z pręta A na pręt C z zachowaniem pierwotnego porządku. Zasady przesuwania są następujące:



RYSUNEK 3.10
Wieże Hanoi

(1) w każdym kroku przesuwa się dokładnie jeden krążek z jednego pręta na inny pręt;
(2) krążek nie może być nigdy nałożony na krążek o mniejszej średnicy;
(3) pręt B może być użyty jako magazyn pomocniczy.

Znajdź algorytm realizujący to zadanie. Zauważ, że jest wygodnie potraktować wieżę jako całość składającą się z pojedynczego górnego krążka i z wieży utworzonej z pozostałych krążków. Algorytm przedstaw w postaci programu rekurencyjnego.

3.2

Napisz procedurę, która znajduje wszystkie $n!$ permutacji n elementów $a_1, ..., a_n$ w miejscu, tj. bez użycia tablicy dodatkowej. Po znalezieniu kolejnej permutacji wywołuje się procedurę parametryczną Q, która np. drukuje znalezioną permutację.

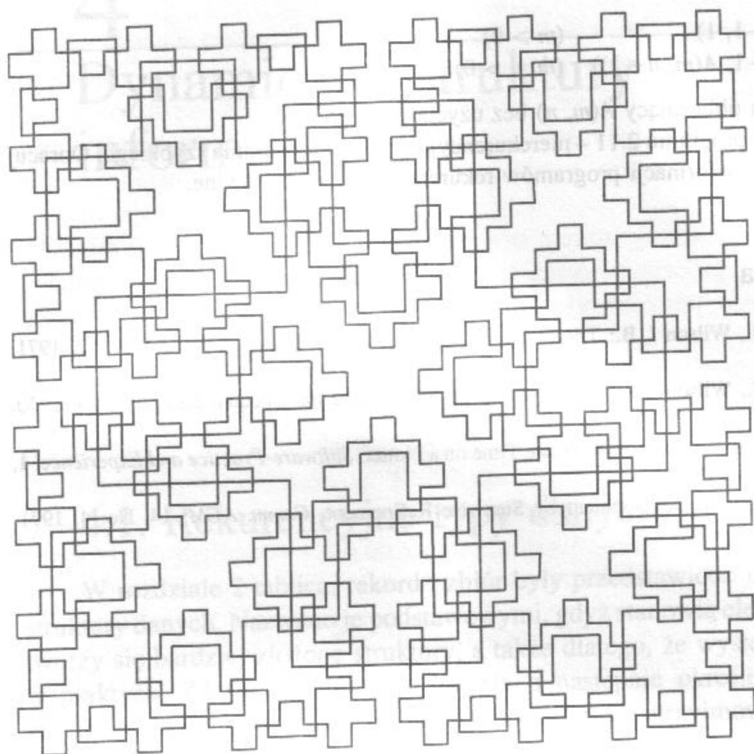
Wskazówka: zadanie znalezienia wszystkich permutacji elementów $a_1, ..., a_n$ potraktuj jako m zadań częściowych znajdujących wszystkie permutacje pierwszych $m-1$ elementów z ciągu $a_1, ..., a_m$, przy czym w i-tym zadaniu elementy a_1 i a_m zostają na początku zamienione.

3.3

Wyprowadź schemat rekursji dla układu krzywych przedstawionego na rys. 3.11, będącego superpozycją czterech krzywych W_1, W_2, W_3, W_4. Struktura ich jest podobna do struktury krzywych Sierpińskiego (3.21) i (3.22). Na podstawie schematu rekursji przygotuj program rekurencyjny rysujący te krzywe.

3.4

Tylko 12 z 92 rozwiązań znalezionych przez program ośmiu hetmanów (3.5) jest istotnie różnych. Pozostałe można otrzymać za pomocą symetrii względem osi lub punktu środkowego szachownicy. Opracuj program, który znajduje 12 głównych rozwiązań. Zauważ np., że ustawianie w kolumnie 1 można ograniczyć do pozycji 1–4.



RYSUNEK 3.11
Krzywe W rzędu od 1 do 4

3.5

Zmień program *Trwałe małżeństwo* tak, aby znajdował optymalne (męskie lub kobiece) rozwiązanie. Uzyskasz w ten sposób program „podziału i ograniczeń" tego samego typu co program 3.7.

3.6

Pewne przedsiębiorstwo kolejowe obsługuje n stacji $S_1, ..., S_n$. W celu poprawienia działania służby informacyjnej dla podróżnych planuje się wprowadzenie komputerowych urządzeń końcowych (terminali). Po dostarczeniu takiemu terminalowi nazw stacji wyjazdowej S_W i stacji docelowej S_D podróżny uzyskałby natychmiast listę połączeń kolejowych z minimalnym łącznym czasem podróży.
Opracuj program tworzący tego rodzaju informację. Załóż, że rozkład jazdy (Twój bank danych) jest reprezentowany za pomocą odpowiedniej struktury danych zawierającej czasy odjazdów (przyjazdów) wszystkich pociągów. Naturalnie nie wszystkie stacje mają połączenie bezpośrednie (zob. także ćwicz. 1.8).

3.7

Funkcja Ackermanna A jest zdefiniowana dla wszystkich nieujemnych argumentów całkowitych m i n w sposób następujący:

$A(0, n) = n+1$

$A(m, 0) = A(m-1, 1) \qquad (m > 0)$
$A(m, n) = A(m-1, A(m, n-1)) \qquad (m, n > 0)$

Opracuj program obliczający $A(m, n)$ bez użycia rekursji.
Jako wzorca użyj programu 2.11 – nierekurencyjnej wersji sortowania szybkiego. Opracuj zbiór reguł dla transformacji programów rekurencyjnych na iteracyjne.

Literatura

3.1. McVitie D.G., Wilson L.B.: The Stable Marriage Problem. *Comm. ACM*, **14**, No. 7, 1971, s. 486–492.
3.2. McVitie D.G., Wilson L.B.: Stable Marriage Assignment for Unequal Sets. *BIT*, **10**, 1970, s. 295–309.
3.3. Space Filling Curves, or How to Waste Time on a Plotter. *Software-Practice and Experience*, **1**, No. 4, 1971, s. 403–440.
3.4. Wirth N.: Program Development by Stepwise Refinement. *Comm. ACM*, **14**, No. 4, 1971, s. 221–227.

4 Dynamiczne struktury informacyjne

4.1. Rekurencyjne typy danych

W rozdziale 2 tablica, rekord i zbiór były przedstawione jako podstawowe struktury danych. Nazwano je podstawowymi, gdyż stanowią elementy, z których tworzy się bardziej złożone struktury, a także dlatego, że występują najczęściej w praktyce. Zdefiniowanie typu danych, a następnie określenie specyfikacji zmiennych tego typu oznacza, że zakres wartości przyjmowanych przez te zmienne i jednocześnie ich wzorzec pamięciowy są ustalone raz na zawsze. Zmienne zadeklarowane w ten sposób są nazywane **statycznymi**. Jednakże jest wiele zagadnień wymagających daleko bardziej skomplikowanych struktur danych. Zagadnienia te charakteryzują się zmiennością struktur danych podczas procesu obliczeniowego. Dlatego też struktury te nazywamy strukturami **dynamicznymi**. Naturalnie składowe takich struktur są – na pewnym poziomie dokładności – statyczne, czyli są jednego z podstawowych typów danych. Niniejszy rozdział jest poświęcony konstrukcji, analizie i zarządzaniu dynamicznymi strukturami informacyjnymi.

Godne uwagi jest to, że istnieją bliskie analogie między metodami używanymi do strukturalizacji algorytmów oraz metodami używanymi do strukturalizacji danych. Oprócz tych analogii istnieją również różnice (inaczej metody byłyby identyczne), ale porównanie metod strukturalizacji programów i danych jest bez wątpienia pouczające.

Elementarną instrukcją niezłożoną jest przypisanie. Jej odpowiednikiem w rodzinie struktur danych jest typ skalarny, niezłożony. Stanowią one podstawowe cegiełki dla złożonych instrukcji i typów danych. Najprostszymi strukturami, otrzymanymi przez wyliczenie lub ułożenie w kolejności, są instrukcja złożona i rekord. Obie struktury składają się ze skończonej (zwykle niewielkiej) liczby wyliczonych składowych, które mogą się między sobą różnić. Jeżeli składowe są jednakowe, to nie trzeba ich indywidualnie wypisywać; używamy instrukcji **for** (dla) i struktury **array** (tablica) w celu określenia znanej, skończonej liczby powtórzeń. Wybór jednego z dwóch lub więcej wariantów

wyraża się przez instrukcję warunkową **if** (jeśli) lub instrukcję wyboru **case** i, odpowiednio, przez rekord z wariantami. I wreszcie, powtarzanie nieznaną początkowo (i potencjalnie nieskończoną) liczbę razy jest wyrażane przez instrukcję **while** (dopóki) lub **repeat** (powtórz). Odpowiadającą strukturą danych jest ciąg (plik), najprostsze pojęcie pozwalające na konstruowanie typów o nieskończonej mocy.

Powstaje pytanie, czy istnieje struktura danych odpowiadająca w podobny sposób instrukcji procedury. Naturalnie najbardziej interesującą i oryginalną własnością procedur jest **rekursja**. Wartości takiego rekurencyjnego typu danych zawierałyby jedną lub więcej składowych, należących do tego samego typu danych, analogicznie do procedury zawierającej jedno lub więcej odwołań do samej siebie. Podobnie jak procedury, definicje takich typów danych mogą być bezpośrednio lub pośrednio rekurencyjne.

Prostym przykładem obiektu, który byłby najwłaściwiej reprezentowany przez typ zdefiniowany rekurencyjnie, jest wyrażenie arytmetyczne spotykane w językach programowania. Rekursji używa się do odzwierciedlenia możliwości zagnieżdżania, tj. użycia nawiasowanych podwyrażeń jako argumentów w wyrażeniu. Załóżmy, że wyrażenie jest zdefiniowane nieformalnie jak następuje:

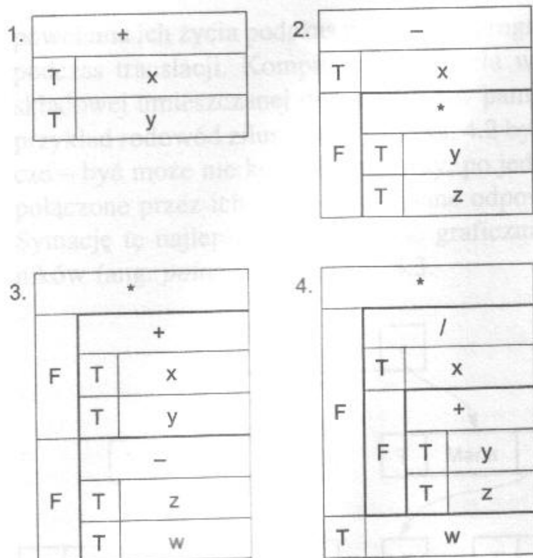
> **Wyrażenie** składa się z kolejno następujących po sobie: składnika, operatora i składnika. (Składniki te stanowią argumenty operatora).
> **Składnik** jest albo zmienną – reprezentowaną przez identyfikator – albo wyrażeniem ujętym w nawiasy.

Stosując dostępne już narzędzia oraz rekursję, można łatwo opisać typ danych, którego wartości reprezentują takie wyrażenia:

```
type wyrażenie = record op: operator;
                   arg1, arg2: składnik
                 end;                                   (4.1)
type składnik  = record
                   if t then (id: alfa)
                        else (podwyr: wyrażenie)
                 end
```

Tak więc każda zmienna typu składnik jest złożona z dwóch składowych, mianowicie pola znacznikowego *t* oraz – jeśli *t* jest prawdą – pola *id* lub w przeciwnym przypadku pola *podwyr*. Rozważmy np. następujące cztery wyrażenia:

(1) $x+y$
(2) $x-y(y*z)$
(3) $(x+y)*(z-w)$ (4.2)
(4) $(x/(y+z))*w$



RYSUNEK 4.1
Wzorzec pamięciowy dla rekurencyjnych struktur rekordowych

Wyrażenia te mogą być zobrazowane przez wzorce z rys. 4.1, które przedstawiają ich zagnieżdżoną, rekurencyjną strukturę, a także wyznaczają sposób odwzorowania tych wyrażeń w pamięci.

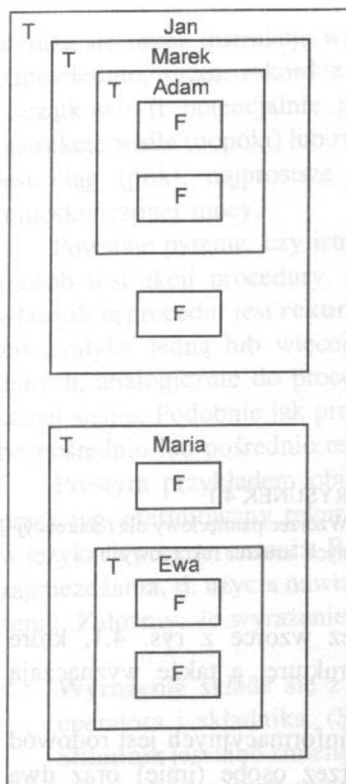
Drugim przykładem rekurencyjnych struktur informacyjnych jest rodowód rodziny: niech *rodowód* będzie zdefiniowany przez osobę (imię) oraz dwa rodowody jej rodziców. Definicja prowadzi nieuchronnie do struktury nieskończonej. Prawdziwe rodowody są ograniczone ze względu na brak informacji na pewnym poziomie drzewa genealogicznego. Można to uwzględnić, stosując ponownie strukturę z wariantami, jak pokazano w definicji (4.3).

type *rod* = **record**
  if *znany* **then**
    (*imię*: *alfa*;
    ojciec, *matka*: *rod*)   (4.3)
end

(Zauważmy, że każda zmienna typu *rod* ma co najmniej jedną składową, mianowicie pole znacznikowe o nazwie *znany*. Jeżeli jego wartością jest prawda (*true*), to istnieją trzy inne pola; w przypadku przeciwnym żadne). Wartość wyznaczoną przez (rekurencyjny) konstruktor rekordów

x = (*T*, *Jan*, (*T*, *Marek*, (*T*, *Adam*, (*F*), (*F*)), (*F*)),
  (*T*, *Maria*, (*F*), (*T*, *Ewa*, (*F*), (*F*))))

przedstawiono na rys. 4.2 w sposób sugerujący pewien wzorzec pamięciowy. (Identyfikator typu *rod* poprzedzający każdy konstruktor został pominięty ze względu na użycie rekordów jednego typu).



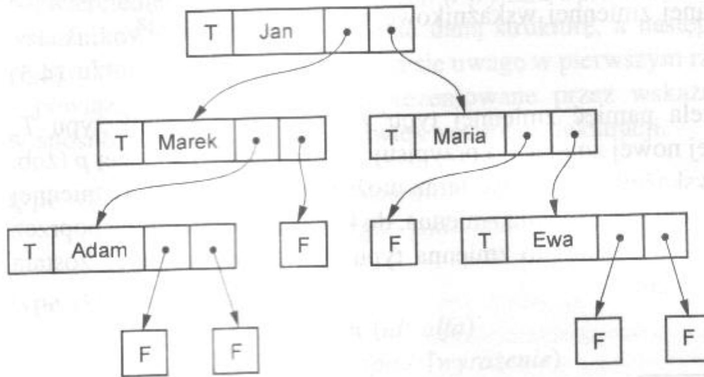
RYSUNEK 4.2
Struktura rodowodu

Jasna staje się ważna rola wariantów; tylko dzięki ich użyciu można ograniczyć rekurencyjną strukturę danych i dlatego też są nieodłączną częścią każdej definicji rekurencyjnej. W tym przypadku analogie między pojęciami strukturalizacji programów i danych stają się szczególnie widoczne. Instrukcja warunkowa musi być częścią każdej procedury rekurencyjnej, aby wykonanie procedury mogło się zakończyć. Zakończenie wykonania w sposób oczywisty odpowiada skończonej mocy danych.

4.2. Wskaźniki lub odniesienia

Charakterystyczną właściwością struktur rekurencyjnych wyróżniającą je spośród struktur podstawowych (tablice, rekordy, zbiory) jest możliwość zmieniania ich rozmiarów. Tak więc nie można przydzielać stałej ilości pamięci dla struktury zdefiniowanej rekurencyjnie, w konsekwencji czego kompilator nie może określać konkretnych adresów składowych takich zmiennych. W celu rozwiązania tego problemu stosuje się najczęściej metodę **dynamicznego przydzielania pamięci**, tj. przydzielania pamięci każdej ze składowych w chwili

powołania ich życia podczas wykonania programu zamiast przydzielania pamięci podczas translacji. Kompilator przydziela wtedy stałą ilość pamięci na **adres** składowej umieszczanej dynamicznie w pamięci zamiast na samą składową. Na przykład rodowód zilustrowany na rys. 4.2 byłby reprezentowany przez pojedyncze – być może nie kolejne – rekordy, po jednym dla każdej osoby. Osoby te są połączone przez ich adresy przypisane odpowiednio polom „ojciec" i „matka". Sytuację tę najlepiej można opisać graficznie przy użyciu strzałek lub wskaźników (ang. *pointers*); zob. rys. 4.3.



RYSUNEK 4.3
Struktura połączona wskaźnikami

Należy podkreślić, że użycie wskaźników w realizacji struktur rekurencyjnych jest jedynie pewną metodą. Programista nie musi być świadom ich istnienia. Pamięć może być przydzielana automatycznie przy pierwszym odwołaniu do nowej składowej. Jeśli jednak użycie odniesień (referencji; ang. *references*) lub wskaźników jest realizowane w sposób jawny, to mogą być konstruowane struktury danych bardziej ogólne od tworzonych przez czysto rekurencyjne definicje danych. W szczególności jest możliwe zdefiniowanie struktur „nieskończonych" lub cyklicznych i określenie pewnych struktur jako **współdzielonych** (ang. *shared*). Dlatego też powszechne stało się we współczesnych językach programowania umożliwienie bezpośredniej manipulacji odniesieniami do danych w uzupełnieniu do manipulacji na samych danych. Powoduje to konieczność wyraźnego notacyjnego rozróżnienia między danymi a odniesieniami do danych, a także to, że w konsekwencji muszą być wyróżnione typy danych, których wartości są wskaźnikami (odniesieniami) do innych danych. Będziemy w tym celu używać następującej notacji:

type $T_p = \uparrow T$ (4.4)

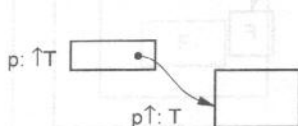
Deklaracja typu (4.4) stwierdza, że wartości typu T_p są wskaźnikami do danych typu T. Tak więc strzałka w (4.4) oznacza „wskazuje na". Podstawowe znaczenie ma fakt, że typ wskazywanych elementów wynika jednoznacznie z deklaracji T_p.

Mówimy, że typ T_p jest *związany z typem T*. Takie związanie pozwala odróżnić wskaźniki w językach wysokiego poziomu od adresów w kodzie wewnętrznym, jest też najważniejszym środkiem zwiększenia ochrony tych danych w programowaniu dzięki redundancji notacji.

Wartości typów wskaźnikowych są generowane wtedy, gdy jednostce danych przydziela się pamięć dynamicznie. Przyjmijmy, że za każdym razem sytuacja taka będzie wyraźnie zaznaczana. Stanowi to przeciwieństwo sytuacji, w której pierwsze wystąpienie jakiegoś obiektu danych wiąże się z automatycznym przydzieleniem pamięci. W tym celu wprowadzamy procedurę wewnętrzną *new* (nowy). Dla danej zmiennej wskaźnikowej p typu T_p instrukcja

$$new(p) \tag{4.5}$$

efektywnie przydziela pamięć zmiennej typu T, generuje wskaźnik typu T_p odwołujący się do tej nowej zmiennej i przypisuje ten wskaźnik zmiennej p (zob. rys. 4.4). Wartość wskaźnika może być teraz utożsamiana z wartością zmiennej wskaźnikowej p. W celu rozróżnienia zmienna, do której odwołujemy się poprzez p, jest oznaczana przez $p\uparrow$. Jest to zmienna typu T, dla której pamięć została przydzielona dynamicznie.



RYSUNEK 4.4
Dynamiczne przydzielanie pamięci zmiennej $p\uparrow$

Stwierdziliśmy powyżej, że warianty są istotne w każdym typie rekurencyjnym dla zapewnienia skończoności. Przykład rodowodu rodziny stanowi wzorzec ukazujący najczęściej występujący układ (zob. (4.3)), a mianowicie przypadek, w którym pole znacznikowe jest dwuwartościowe (boolowskie) i w którym jego wartość równa fałsz (*false*) implikuje brak dalszych składowych. Wyrażone jest to przez **schemat deklaracji** (4.6).

type T = **record if** p **then** $S(T)$ **end** (4.6)

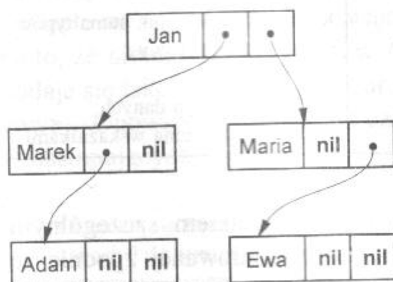
$S(T)$ oznacza sekwencję definicji pól, która zawiera jedno lub więcej pól typu T, co zapewnia rekurencyjność. Wszystkie struktury typu określonego według (4.6) będą wyznaczać drzewiastą (lub listową) strukturę, podobną do pokazanej na rys. 4.3. Jej szczególną własnością jest to, że zawiera ona wskaźniki do składowych zawierających tylko pole znacznikowe, tj. nie zawierających dalszych istotnych informacji. Metoda realizacji w maszynie struktur rekurencyjnych z użyciem wskaźników sugeruje łatwy sposób zaoszczędzania pamięci przez włączenie informacji pola znacznikowego do wartości samego wskaźnika. Powszechnym rozwiązaniem jest **rozszerzenie** zakresu wartości typu T_p o pojedynczą wartość nie wskazującą na żaden element. Oznaczamy tę wartość symbolem specjalnym **nil** (nic, pusty) i rozumiemy, że **nil** jest automatycznie elementem wszystkich deklarowanych typów wskaźnikowych. To rozszerzenie zakresu wartości wskaź-

ników wyjaśnia, dlaczego struktury skończone mogą być generowane *bez* jawnego wystąpienia wariantów (warunków) w ich (rekurencyjnych) deklaracjach.

Nowe sformułowania typów danych zadeklarowanych w (4.1) i (4.3) – oparte na jawnych wskaźnikach – przedstawione są odpowiednio w (4.7) i (4.8). Zauważmy, że w drugim z wymienionych przykładów (który odpowiada schematowi (4.6)) składowa wariantowa rekordu zniknęła, ponieważ $p\uparrow.znany = false$ wyraża się teraz jako $p =$ **nil**. Przemianowanie typu *rod* na *osoba* jest odzwierciedleniem różnic powstałych w wyniku wprowadzenia jawnych wartości wskaźnikowych. Zamiast rozważać daną strukturę, a następnie analizować jej podstrukturę i jej składowe, zwraca się uwagę w pierwszym rzędzie na składowe, a powiązania między nimi (reprezentowane przez wskaźniki) nie wynikają w sposób oczywisty z jakiejkolwiek ustalonej deklaracji.

```
type wyrażenie = record op: operator;
                        arg1, arg2: ↑składnik
                 end;
type składnik  = record                                        (4.7)
                        if t then (id: alfa)
                        else (pod:↑wyrażenie)
                 end

type osoba     = record imię: alfa;
                        ojciec, matka:↑osoba                   (4.8)
                 end
```

Struktura danych reprezentująca rodowód, pokazana na rys. 4.2 i 4.3, jest ponownie przedstawiona na rys. 4.5, na którym wskaźniki do osób nieznanych są oznaczone przez **nil**. Wynikające z tego lepsze wykorzystanie pamięci jest oczywiste.



RYSUNEK 4.5
Struktura ze wskaźnikami o wartościach **nil**

Odwołując się znowu do rys. 4.5, załóżmy, że Marek i Maria są rodzeństwem, tj. mają tego samego ojca i matkę. Daje się to łatwo wyrazić przez zastąpienie dwóch wartości **nil** w odpowiednich polach dwóch rekordów. Realizacja, w której ukrywa się pojęcie wskaźnika lub używa innej metody

obsługi pamięci, zmusiłaby programistę do dwukrotnego reprezentowania każdego z rekordów Adam i Ewa. Jakkolwiek przy dostępie do danych w celu ich zbadania nie jest istotne, czy dwaj ojcowie (i dwie matki) występują w dwóch rekordach czy też w jednym, jednak różnica jest *istotna wtedy, gdy* jest dopuszczalna **selektywna aktualizacja**. Traktowanie wskaźników jako jawne jednostki danych, a nie jako ukryte środki realizacyjne, pozwala programiście jasno wyrażać się tam, gdzie zamierza stosować **współdzielenie pamięci**.

Dalszą konsekwencją jawności wskaźników jest możliwość definiowania i manipulowania cyklicznymi strukturami danych. Ta dodatkowa elastyczność daje, oczywiście, nie tylko większe możliwości, ale również wymaga zwiększenia uwagi programisty, ponieważ manipulacja cyklicznymi strukturami danych może łatwo doprowadzić do nie kończących się procesów.

Tego rodzaju sytuacja, że zwiększenie możliwości i elastyczności idzie w parze z niebezpieczeństwem popełniania błędów, jest dobrze znana w programowaniu i w szczególności przypomina problemy z instrukcją skoku. I rzeczywiście, jeżeli analogia między strukturami programowymi i strukturami danych ma być rozszerzona, to czysto rekurencyjna struktura danych może być umieszczona na poziomie odpowiadającym procedurze, podczas gdy wprowadzenie wskaźników jest porównywalne do korzystania z instrukcji skoku. I tak, jeżeli instrukcja skoku pozwala na konstruowanie dowolnego schematu programu (włączając pętle), to wskaźniki pozwalają na składanie dowolnych struktur danych (włączając cykle). Porównanie odpowiadających sobie struktur programowych i danych pokazano w skróconej formie w tabl. 4.1.

TABLICA 4.1
Odpowiedniości między strukturami programowymi i strukturami danych

Wzorzec konstrukcji	Instrukcja programu	Typ danych
Element podstawowy	Przypisanie	Typ skalarny
Wyliczenie	Instrukcja złożona	Typ rekordowy
Powtórzenie stałą liczbę razy	Instrukcja „dla"	Typ rekordowy
Wybór	Instrukcja warunkowa	Rekord z wariantami, suma typów
Powtórzenie nieznaną z góry liczbę razy	Instrukcje „dopóki" lub „powtarzaj"	Ciąg lub typ plikowy
Rekursja	Instrukcja procedury	Rekurencyjny typ danych
Ogólny „graf"	Instrukcja skoku	Struktura połączona wskaźnikami

W rozdziale 3 widzieliśmy, że iteracja jest przypadkiem szczególnym rekursji i że wywołanie rekurencyjnej procedury P zdefiniowanej zgodnie ze schematem (4.9)

procedure P;
begin
  if B **then begin** P_0; P **end** (4.9)
end

gdzie P_0 jest instrukcją nie wywołującą P, jest równoważne instrukcji iteracyjnej

while B **do** P_0

którą można zastąpić to wywołanie.

Analogie przedstawione w ogólnym zarysie w tabl. 4.1 ujawniają, że podobny związek zachodzi między rekurencyjnymi typami danych a ciągiem. W rzeczywistości typ rekurencyjny zdefiniowany zgodnie ze schematem

```
type T = record
        if B then (t0: T0; t: T)          (4.10)
    end
```

gdzie T_0 jest typem nie odwołującym się do typu T, jest równoważny i wymienialny na sekwencyjny typ danych

file of T_0

Widać z tego, że rekursję można zastąpić iteracją w definicjach programów *i* danych wtedy (i tylko wtedy), gdy nazwa procedury lub typu występuje rekurencyjnie tylko raz, na końcu (lub na początku) takiej definicji.

Pozostała część tego rozdziału jest poświęcona generowaniu i manipulowaniu na strukturach danych, których składowe są połączone jawnymi wskaźnikami. Specjalny nacisk położono na struktury o szczególnie prostych wzorcach; sposoby manipulowania bardziej złożonymi strukturami mogą być wyprowadzone ze sposobów manipulowania tworami podstawowymi. Przykładami tworów podstawowych są listy liniowe lub ciągi łańcuchowe (najprostszy przypadek) i drzewa. Fakt, że przede wszystkim zajmiemy się tymi „elementami budulcowymi" przy strukturalizacji danych, nie świadczy o tym, że bardziej złożone struktury nie występują w praktyce. W istocie, poniższe opowiadanie, które znalazło się w jednej z gazet szwajcarskich (Zurich, czerwiec 1922) jest dowodem na to, że nieregularność może występować nawet w przypadkach, które zwykle podaje się jako przykłady struktur regularnych, jak na przykład drzewo (rodowodowe). Opowiadanie mówi o człowieku, który opisuje niedole swego życia w następujących słowach:

> Ożeniłem się z wdową, która miała dorosłą córkę. Mój ojciec, który odwiedzał nas często, zakochał się w mojej przybranej córce i ożenił się z nią. Tak więc mój ojciec stał się moim przybranym synem, a moja przybrana córka stała się moją matką. Kilka miesięcy później moja żona urodziła syna, który stał się przybranym bratem mojego ojca i jednocześnie moim wujem. Żona mojego ojca, czyli moja przybrana córka miała także syna. W ten sposób uzyskałem brata i jednocześnie wnuka. Moja żona jest moją babką, ponieważ jest matką mojej matki. Tak więc jestem mężem mojej żony i jednocześnie jej przybranym wnukiem; innymi słowy, jestem swoim dziadkiem.

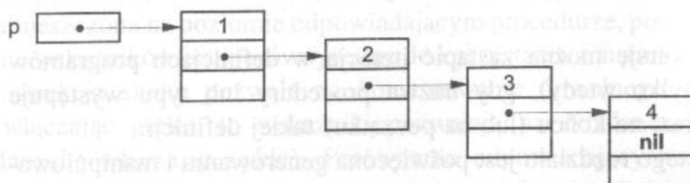
4.3. Listy liniowe

4.3.1. Operacje podstawowe

Najprostszym sposobem wiązania lub łączenia zbioru elementów jest ułożenie ich w jedną **listę** lub **kolejkę**. W tym przypadku tylko pojedyncze połączenie jest niezbędne do odwołania się od dowolnego elementu do jego następnika.

Przyjmijmy, że typ T jest zdefiniowany w sposób pokazany w (4.11). Każda zmienna tego typu składa się z trzech składowych, a mianowicie: klucza identyfikującego, wskaźnika do jej następnika i prawdopodobnie dalszych związanych z nią informacji, opuszczonych w deklaracji (4.11).

```
type T = record klucz: integer;
                nast: ↑T;                    (4.11)
                .........
         end
```



RYSUNEK 4.6
Przykład listy

Listę elementów typu T ze wskaźnikiem do jej pierwszej składowej, przypisanym zmiennej p, przedstawiono na rys. 4.6. Prawdopodobnie najprostszą operacją, jakiej można dokonać na tej liście, jest **wstawienie elementu na jej początek**. Najpierw, gdy elementowi typu T przydziela się pamięć, odniesienie do niego (wskaźnik) zostaje przypisane tymczasowej zmiennej wskaźnikowej q. Następnie należy dokonać prostego uaktualnienia dwóch zmiennych wskaźnikowych, jak to pokazano w instrukcjach (4.12).

```
new(q); q↑.nast := p; p := q                 (4.12)
```

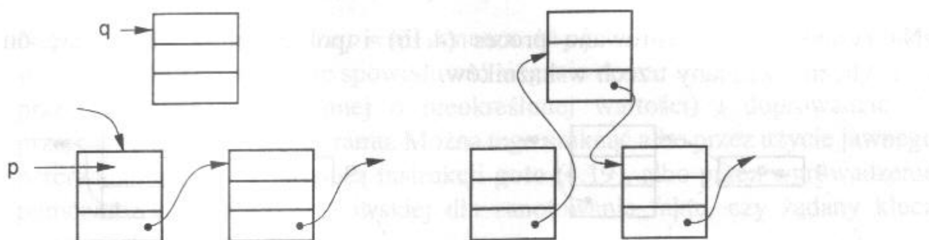
Zauważmy, że istotna jest kolejność tych trzech instrukcji.

Operacja wstawiania elementu na początek listy pokazuje od razu, jak taka lista może być generowana: poczynając od pustej listy, przez wielokrotne dodawanie elementu początkowego listy. Taki proces **generowania listy** jest realizowany w algorytmie (4.13); dołącza się tu n elementów.

```
p := nil; {zacznij od pustej listy}
while n > 0 do
  begin new(q); q↑.nast := p; p := q;        (4.13)
    q↑.klucz := n; n := n-1
  end
```

Jest to najprostszy sposób tworzenia listy. Jednakże wynikowy porządek elementów jest odwrotnością porządku ich „pojawiania się”. W niektórych zastosowaniach jest to niewskazane; w konsekwencji nowe elementy muszą być dołączane na *końcu* listy. Koniec listy można łatwo rozpoznać, przeglądając ją, ale takie nieco naiwne podejście zwiększa nakład pracy, którą można zaoszczędzić, używając drugiego wskaźnika, np. q, wskazującego zawsze na ostatni element. Metodę tę zastosowano w programie 4.4, który generuje odsyłacze do danego tekstu. Wadą jej jest to, że pierwszy wstawiany element trzeba traktować inaczej niż pozostałe.

Korzystanie ze wskaźników w sposób jawny upraszcza znacznie pewne operacje, które bez wskaźników są nieporęczne w opisie. Wśród elementarnych operacji na listach znajdują się operacje wstawiania i usuwania elementów (selektywne aktualizowanie listy), a także przeglądanie listy. Zajmijmy się najpierw **wstawianiem elementu do listy**.

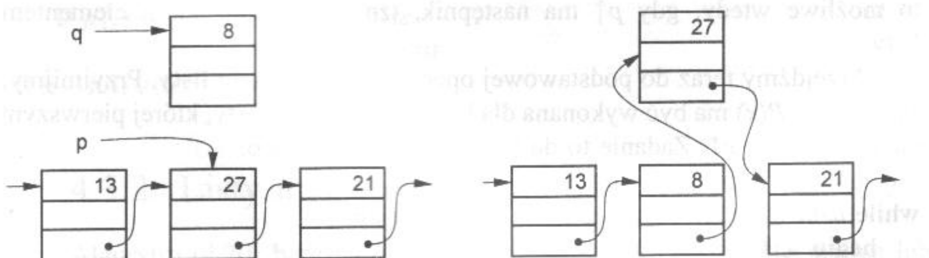


RYSUNEK 4.7
Wstawienie do listy po $p\uparrow$

Przyjmijmy, że element określony przez wskaźnik (zmienną) q ma być wstawiony do listy po elemencie wyznaczonym przez wskaźnik p. Konieczne uaktualnienia zmiennych wskaźnikowych przedstawiono w (4.14), a ich efekt widzimy na rys. 4.7.

$$q\uparrow.nast := p\uparrow.nast;\ p\uparrow.nast := q \tag{4.14}$$

Jeżeli interesuje nas wstawianie *przed* wyznaczonym elementem $p\uparrow$, to wydaje się, że ze względu na brak powiązań z elementami poprzedzającymi, jednokierunkowy łańcuch nie pozwala na to. Jednakże prosty „chwyt” rozwiązuje



RYSUNEK 4.8
Wstawienie do listy przed $p\uparrow$

nasz problem – spójrzmy na fragment (4.15) i na rys. 4.8. Przyjmijmy, że kluczem nowego elementu jest $k = 8$.

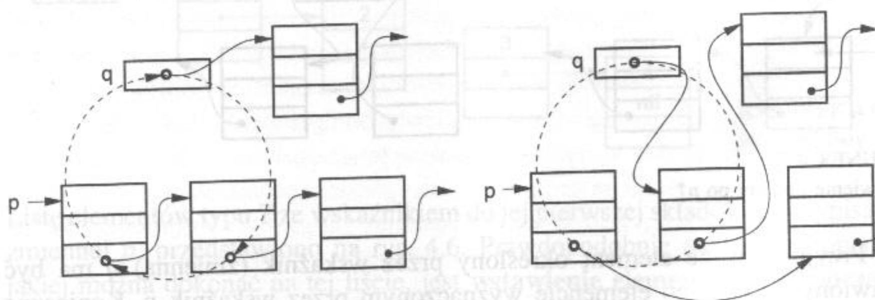
$$new(q);\ q\uparrow := p\uparrow; \qquad (4.15)$$
$$p\uparrow.klucz := k;\ p\uparrow.nast := q$$

„Chwyt” polega na wstawieniu nowej składowej *po* $p\uparrow$, a następnie zamianie wartości nowego elementu i elementu $p\uparrow$.

Rozważmy następnie proces **usuwania elementu z listy**. Usunięcie następnika $p\uparrow$ jest oczywiste. W instrukcjach (4.16) jest to pokazane łącznie z ponownym wstawieniem usuniętego elementu na początku innej listy (wyznaczonej przez q). Zmienna r jest pomocniczą zmienną typu $\uparrow T$.

$$r := p\uparrow.nast;\ p\uparrow.nast := r\uparrow.nast; \qquad (4.16)$$
$$r\uparrow.nast := q;\ q := r$$

Na rysunku 4.9 zilustrowano proces (4.16) i pokazano, że składa się on z cyklicznej zamiany trzech wskaźników.



RYSUNEK 4.9
Usunięcie z listy i ponowne wstawienie

Trudniejsze jest usunięcie określonego elemetu (zamiast jego następnika), podobnie jak wstawienie przed $p\uparrow$: cofnięcie się do poprzednika wskazanego elementu jest niemożliwe. Natomiast skreślenie następnika po przesunięciu jego wartości do przodu jest rozwiązaniem względnie oczywistym i prostym. Jest to możliwe wtedy, gdy $p\uparrow$ ma następnik, tzn. nie jest ostatnim elementem listy.

Przejdźmy teraz do podstawowej operacji **przeglądania listy**. Przyjmijmy, że operacja $P(x)$ ma być wykonana dla każdego elementu listy, której pierwszym elementem jest $p\uparrow$. Zadanie to da się wyrazić następująco:

while *lista wyznaczona przez p nie jest pusta* **do**
    begin *wykonaj operację P*;
        przejdź do następnika
    end

Dokładnie operacja ta jest opisana przez instrukcję (4.17).

while $p \neq$ **nil do**
    begin $P(p\uparrow)$; $p := p\uparrow.nast$ (4.17)
    end

Z definicji instrukcji **while** i z definicji struktury listowej wynika, że operacja P jest wykonywana dla wszystkich elementów listy i dla żadnych innych.

Operacją wykonywaną bardzo często na listach jest **wyszukiwanie w liście** elementu o danym kluczu x. Tak jak w strukturach plikowych, przeszukiwanie jest tu czysto sekwencyjne. Przeszukiwanie kończy się po znalezieniu szukanego elementu lub osiągnięciu końca listy. Przyjmiemy ponownie, że początek listy jest wyznaczony przez wskaźnik P. A oto pierwsza próba sformułowania tego prostego przeszukiwania:

while $(p \neq$ **nil**$) \wedge (p\uparrow.klucz \neq x)$ **do** $p := p\uparrow.nast$ (4.18)

Należy jednak zauważyć, że $p =$ **nil** oznacza, że $p\uparrow$ nie istnieje. Tak więc badanie warunku zakończenia może spowodować sięganie do nie istniejącej zmiennej (w przeciwieństwie do zmiennej o nieokreślonej wartości) i doprowadzić do przerwania wykonania programu. Można tego uniknąć albo przez użycie jawnego przerwania iteracji za pomocą instrukcji **goto** (4.19), albo przez wprowadzenie pomocniczej zmiennej boolowskiej dla zanotowania faktu, czy żądany klucz został znaleziony (4.20).

while $p \neq$ **nil do**
    if $p\uparrow.klucz = x$ **then goto** *Znaleziony* (4.19)
                else $p := p\uparrow.nast$

Użycie instrukcji **goto** wymaga istnienia docelowej etykiety w pewnym miejscu programu. Zauważmy, że instrukcja ta w pewien sposób kłóci się z instrukcją **while**. Ujawnia się to w tym, że warunek po **while** jest mylący; instrukcja kontrolowana nie zawsze musi być wykonywana tak długo, jak długo $p \neq$ **nil**.

$b := true$;
while $(p \neq$ **nil**$) \wedge b$ **do**
    if $p\uparrow.klucz = x$ **then** $b := false$ (4.20)
                else $p := p\uparrow.nast$
$\{(p =$ **nil**$) \vee \neg b\}$

4.3.2. Listy uporządkowane i listy reorganizowane

Algorytm (4.20) bardzo przypomina procedury przeszukujące tablicę lub plik. W rzeczywistości plik nie jest niczym innym jak listą, dla której sposób łączenia kolejnych elementów jest nieokreślony lub ukryty. Ponieważ prymityw-

ne operatory plikowe nie pozwalają na wstawianie nowych elementów (z wyjątkiem wstawienia na koniec) lub na ich usuwanie (z wyjątkiem usunięcia *wszystkich* elementów), wybór reprezentacji pozostawiony jest całkowicie twórcy realizacji, który może zastosować sekwencyjne przydzielanie pamięci – umieszczając kolejne składowe w spójnym obszarze pamięci. Listy liniowe ze wskaźnikami używanymi w sposób jawny zapewniają *większą elastyczność* i z tego powodu powinny być stosowane wszędzie tam, gdzie dodatkowa elastyczność jest potrzebna.

Jako przykład rozważmy problem, który będzie wielokrotnie występował w tym rozdziale, służąc do ilustracji alternatywnych rozwiązań i metod. Jest to problem czytania tekstu, zbierania jego wszystkich słów i obliczania częstości ich występowania. Nazywa się go konstrukcją **skorowidza**.

Narzucającym się rozwiązaniem tego problemu jest skonstruowanie *listy* słów rozpoznanych w tekście. Listę przegląda się dla każdego słowa. Po znalezieniu danego słowa licznik częstości zwiększa się o 1; w przeciwnym przypadku słowo dołącza się do listy. Upraszczając, nazwiemy ten proces *przeszukiwaniem*, chociaż może oczywiście zawierać *wstawianie*.

Aby móc skoncentrować naszą uwagę na sprawie zasadniczej – obsłudze list – przyjmiemy, że słowa zostały już wybrane z badanego tekstu i zakodowane jako liczby całkowite oraz są dostępne w postaci pliku wejściowego.

Sformułowanie procedury o nazwie *szukaj* wynika bezpośrednio z algorytmu (4.20). Zmienna *korzeń* odpowiada początkowi listy, do której są wstawiane nowe słowa zgodnie z instrukcjami (4.12). Pełny algorytm przedstawiono jako program 4.1. Zawiera on procedurę drukowania skonstruowanej listy skorowidza w postaci tabeli. Proces drukowania jest przykładem, w którym pewna czynność jest wykonywana jednorazowo dla każdego elementu z listy, tak jak to pokazano w sposób schematyczny w (4.17).

PROGRAM 4.1
Proste wstawianie do listy

```
program lista(input, output);
  {proste wstawianie do listy}
  type ref = ↑ słowo;
     słowo = record klucz: integer;
                    licznik: integer;
                    nast: ref
             end;
  var k: integer; korzeń: ref;

  procedure szukaj(x: integer; var korzeń: ref);
    var w: ref; b: boolean;
  begin w := korzeń; b := true;
    while (w ≠ nil) ∧ b do
      if w↑.klucz = x then b := false else w := w↑.nast;
```

```
        if b then
        begin {nowa składowa} w := korzeń; new(korzeń);
           with korzeń↑ do
           begin klucz := x; licznik := 1; nast := w
           end
        end else
        w↑.licznik := w↑. licznik+1
     end {szukaj};
     procedure drukujlistę (w: ref);
     begin while w≠nil do
        begin writeln(w↑. klucz, w↑. licznik);
              w := w↑. nast
        end
     end {drukujlistę};
begin korzeń := nil; read(k);
     while k≠0 do
        begin szukaj(k, korzeń); read(k)
        end;
     drukujlistę(korzeń)
end.
```

Algorytm liniowego przeglądania użyty w programie 4.1 przypomina procedurę przeszukiwania tablicy lub pliku, a w szczególności w podobny sposób można uprościć warunek zakończenia pętli – stosując metodę z *wartownikiem*. Wartownik może równie dobrze posłużyć przy przeszukiwaniu listy; jest on reprezentowany przez fikcyjny element na końcu listy. Procedurę *szukaj* z programu 4.1 możemy zastąpić nową procedurą (4.21), jeśli dodamy zmienną globalną *wartownik*, a inicjowanie zmiennej *korzeń* zastąpimy instrukcjami:

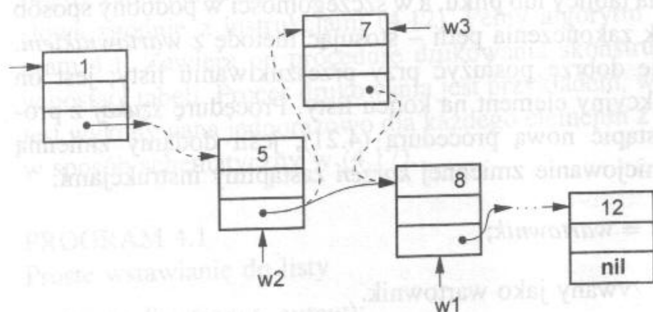
```
new(wartownik); korzeń := wartownik;
```

które generują element używany jako wartownik.

```
procedure szukaj(x: integer; var korzeń: ref);
   var w: ref;
begin w := korzeń; wartownik↑. klucz := x;
   while w↑. klucz≠x do w := w↑. nast;
   if w≠wartownik then w↑. licznik := w↑. licznik+1 else          (4.21)
      begin {nowa składowa} w := korzeń; new(korzeń);
         with korzeń↑ do
         begin klucz := x; licznik := 1; nast := w
         end
      end
end {szukaj}
```

Oczywiście moc i elastyczność listy łączonej (listy z dowiązaniami; ang. *linked list*) są w tym przykładzie niewłaściwie wykorzystane i liniowe przeglądanie całej listy warto stosować tylko w przypadku niewielkiej liczby elementów. Niemniej jednak można wprowadzić proste usprawnienie: **przeszukiwanie listy uporządkowanej**. Jeśli lista jest uporządkowana (powiedzmy według rosnących wartości kluczy), to przeglądanie można najpóźniej zakończyć wtedy, gdy rozpozna się pierwszy klucz większy od nowego klucza. Uporządkowanie listy otrzymuje się przez wstawianie nowych elementów w odpowiednie miejsce na liście zamiast na jej początek. W efekcie uporządkowanie uzyskuje się praktycznie za darmo. Wynika to z prostoty operacji wstawiania do listy łączonej, tj. z pełnego wykorzystania jej elastyczności. Możliwości takich nie dają struktury tablicowe i plikowe. (Zauważmy jednakowoż, że nawet w uporządkowanych listach nie ma odpowiednika przeszukiwania połówkowego tablic).

Wyszukiwanie w listach uporządkowanych jest typowym przykładem sytuacji opisanej w (4.15), w której pewien element musi być wstawiony przed danym obiektem – konkretnie przed pierwszym obiektem, którego klucz jest większy. Opisana tu metoda różni się jednak od metody użytej w (4.15). Zamiast kopiowania wartości dwa wskaźniki przebiegają listę podczas jej przeglądania; $w1$ poprzedza $w2$ i dlatego $w2$ wyznacza właściwe miejsce wstawienia wtedy, gdy okaże się, że $w1$ wskazuje na za duży klucz. Proces wstawiania pokazano na rys. 4.10. Zanim przejdziemy dalej, musimy rozważyć dwie okoliczności:



RYSUNEK 4.10
Wstawianie do listy uporządkowanej

(1) Wskaźnik do nowego elementu ($w3$) powinien być przypisany zmiennej $w2\uparrow.nast$ z wyjątkiem sytuacji, w której lista jest jeszcze pusta. Ze względu na prostotę i efektywność wolimy nie odróżniać tej sytuacji za pomocą instrukcji warunkowej. Jedynym sposobem uniknięcia tego jest umieszczenie fikcyjnego elementu na początku listy.

(2) Przeglądanie z dwoma wskaźnikami przebiegającymi wzdłuż listy wymaga, aby lista zawierała co najmniej jeden element (oprócz fikcyjnego). Wynika stąd, że inaczej traktuje się wstawianie pierwszego elementu niż pozostałych.

Propozycję spełniającą powyższe zalecenia i uwagi przedstawiono w procedurze (4.23). Zastosowano w niej pomocniczą procedurę *wstaw* (4.22), zadeklarowaną jako lokalna dla procedury *szukaj*. Generuje ona i inicjuje nowy element *w*.

```
procedure wstaw(w: ref);
   var w3: ref;
begin new(w3);
   with w3↑ do
        begin klucz := x; licznik := 1; nast := w                    (4.22)
        end;
   w2↑.nast := w3
end {wstaw}
```

Instrukcja inicjująca „*korzeń* := **nil**" w programie 4.1 jest zgodnie z tym zastąpiona przez

new(*korzeń*); *korzeń*↑.*nast* := **nil**

Odwołując się do rys. 4.10, określimy warunek, przy którym podczas przeglądania przechodzimy do następnego elementu; składa się on z dwóch czynników:

$(w1\uparrow.klucz < x) \wedge (w1\uparrow.nast \neq \mathbf{nil})$

W wyniku otrzymujemy procedurę przeszukiwania przedstawioną w programie (4.23).

```
procedure szukaj(x: integer; var korzeń: ref);
   var w1, w2: ref;
begin w2 := korzeń; w1 := w2↑.nast;
   if w1 = nil then wstaw(nil) else
   begin
      while (w1↑.klucz < x) ∧ (w1↑.nast ≠ nil) do                (4.23)
         begin w2 := w1; w1 := w2↑.nast
         end;
      if w1↑.klucz = x then w1↑.licznik := w1↑.licznik + 1 else
         wstaw(w1)
   end
end {szukaj};
```

Niestety, propozycja ta zawiera błąd logiczny. Pomimo naszych starań wkradł się „chochlik"! Zachęcamy czytelnika, aby spróbował wykryć nasze przeoczenie przed dalszym czytaniem. Tym, którzy zrezygnują z tego detektywistycznego zajęcia, powinno wystarczyć stwierdzenie, że w procedurze (4.23) element umieszczony na liście jako pierwszy będzie zawsze przepychany na koniec listy. Błąd ten da się naprawić, jeżeli uwzględnimy fakt, że gdy

przeglądanie kończy się ze względu na drugi czynnik, wtedy nowy element musi być wstawiony *po* $w1\uparrow$ zamiast *przed*. Tak więc instrukcję „*wstaw*(*w*1)” zastępujemy przez

```
begin if w1↑.nast = nil then
        begin w2 := w1; w1 := nil
        end                                                    (4.24)
    wstaw(w1)
end
```

Łatwowierny czytelnik dał się znowu oszukać, gdyż instrukcja (4.24) jest ciągle niepoprawna. W celu znalezienia pomyłki przyjmijmy, że nowy klucz leży między ostatnim i przedostatnim kluczem. Obydwa czynniki warunku kontynuacji stają się fałszywe wtedy, gdy przeglądanie osiąga koniec listy i – w konsekwencji – wstawienie nowego klucza następuje poza elementem końcowym. Jeżeli ten sam klucz wystąpi ponownie, zostanie on wstawiony poprawnie i będzie dwukrotnie występował na liście. Lekarstwem jest zastąpienie warunku

$w1\uparrow.nast = \textbf{nil}$

w programie (4.24) przez

$w1\uparrow.klucz < x$

W celu przyspieszenia przeszukiwania warunek kontynuacji instrukcji **while** może być ponownie uproszczony dzięki użyciu wartownika. Wymaga to obecności zarówno *fikcyjnego początku*, jak i wartownika na końcu listy. Tak więc lista powinna być zainicjowana za pomocą następujących instrukcji:

new(*korzeń*); *new*(*wartownik*); *korzeń*↑. *nast* := *wartownik*;

i postać procedury przeszukiwania istotnie się upraszcza (zob. (4.25)).

```
procedure szukaj (x: integer; var korzeń: ref);
    var w1, w2, w3: ref;
begin w2 := korzeń; w1 := w2↑. nast; wartownik↑. klucz := x;
    while w1↑. klucz < x do
        begin w2 := w1; w1 := w2↑. nast
        end;
    if (w1↑. klucz = x) ∧ (w1 ≠ wartownik) then
        w1↑. licznik := w1↑. licznik + 1 else                  (4.25)
    begin new (w3); {wstaw w3 pomiędzy w1 i w2}
        with w3↑ do
            begin klucz := x; licznik := 1; nast := w1
            end;
        w2↑. nast := w3
    end
end {szukaj}
```

Najwyższy czas zapytać, jakie korzyści można osiągnąć z przeszukiwania uporządkowanej listy. Pamiętając, że dodatkowy stopień komplikacji jest niewielki, nie powinno się oczekiwać znacznego postępu.

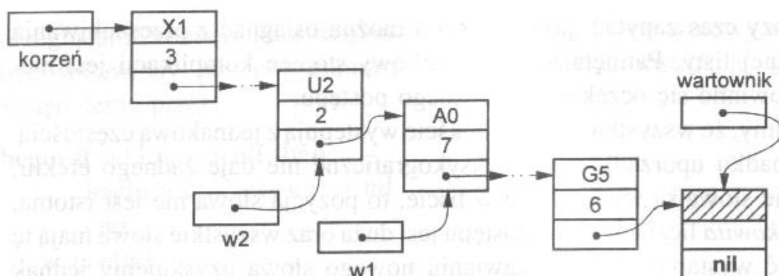
Przyjmijmy, że wszystkie słowa w tekście występują z jednakową częstością. W tym przypadku uporządkowanie leksykograficzne nie daje żadnego efektu; jeśli wszystkie słowa są wymienione w liście, to pozycja słowa nie jest istotna, jeśli tylko *całkowita* liczba kroków dostępu jest duża oraz wszystkie słowa mają tę samą częstość wystąpień. Przy wstawianiu nowego słowa uzyskujemy jednak pewne korzyści. Zamiast pierwotnego przeglądania całej listy średnio należy przejrzeć jedynie jej połowę. Tak więc wstawianie do listy uporządkowanej opłaca się tylko wtedy, gdy ma być generowany skorowidz wielu różnych słów o relatywnie małych częstościach występowania. Z tego też względu poprzednie przykłady są przede wszystkim odpowiednie jako ćwiczenia w programowaniu, nie zaś jako przykłady zastosowań praktycznych.

Reprezentacja danych w postaci listy łączonej jest zalecana wtedy, gdy liczba elementów jest relatywnie niewielka (np. < 100), zmienia się, a – ponadto – gdy brak informacji o ich częstościach wystąpień. Typowym przykładem jest tablica symboli w translatorach języków programowania. Każda deklaracja powoduje dołączenie nowego symbolu, który przy napotkaniu końca zakresu jego ważności jest usuwany z listy. Użycie prostych list łączonych jest odpowiednie dla zastosowań o względnie krótkich programach. Nawet w tym przypadku można osiągnąć znaczne usprawnienie metody dostępu przez zastosowanie bardzo prostego sposobu, który tu ponownie omówimy, ponieważ stanowi ładny przykład demonstrujący elastyczność struktury listy łączonej.

Cechą charakterystyczną programów jest to, że wystąpienia tych samych identyfikatorów są bardzo często *zgrupowane*, tzn. za jednym wystąpieniem następuje często jedno lub kilka kolejnych wystąpień tego samego słowa. Informacja ta sugeruje reorganizowanie listy po każdym dostępie przez przeniesienie ostatnio znalezionego słowa na początek listy, co minimalizuje długość drogi przeszukiwania listy przy kolejnym wyszukiwaniu tego elementu. Ta metoda dostępu jest nazywana **przeszukiwaniem listy z przestawianiem** lub – nieco pompatycznie – **samoorganizującym się przeszukiwaniem listy**. Prezentując odpowiedni algorytm w postaci procedury, którą można wstawić do programu 4.1, korzystamy z naszych dotychczasowych doświadczeń i wprowadzamy wartownika już na wstępie. Rzeczywiście, wartownik nie tylko przyspiesza przeglądanie, ale w tym przypadku również upraszcza program. Lista na początku nie jest pusta, ponieważ zawiera już wartownika. Instrukcjami inicjującymi są

new(*wartownik*); *korzeń* := *wartownik*;

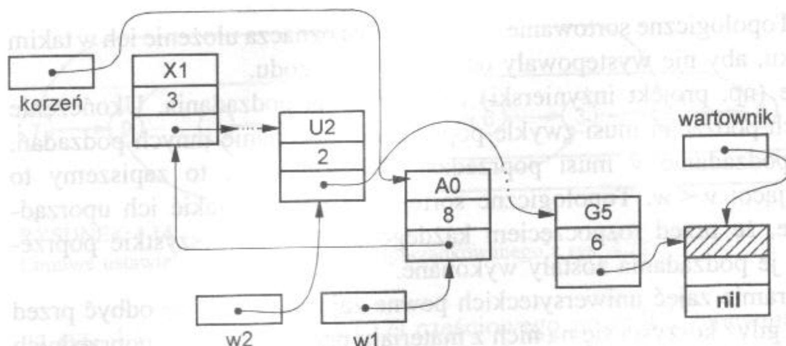
Zauważmy, że zasadniczą różnicą między nowym algorytmem a prostym przeszukiwaniem listy (4.21) jest przestawianie podejmowane po znalezieniu elementu. Jest on wtedy odłączany lub usuwany ze swej starej pozycji na liście i wstawiany na jej początek. Usuwanie to ponownie wymaga dwóch „polujących”



RYSUNEK 4.11
Lista przed przestawieniem

wskaźników pozwalających na dostęp do poprzednika $w2\uparrow$ identyfikowanego elementu $w1\uparrow$. Jednocześnie wymaga to specjalnego traktowania pierwszego elementu (pusta lista). Aby wyobrazić sobie proces zmiany połączeń, odwołamy się do rys. 4.11. Pokazane są tu dwa wskaźniki, przy czym $w1\uparrow$ identyfikuje żądany element. Konfigurację otrzymaną po poprawnym przestawieniu elementu przedstawiono na rys. 4.12, nową zaś, pełną procedurę przeszukiwania – w programie (4.26).

```
procedure szukaj(x: integer; var korzeń: ref);
  var w1, w2: ref;
begin w1 := korzeń; wartownik↑.klucz := x;
  if w1 = wartownik then
  begin {pierwszy element} new(korzeń);
    with korzeń↑ do
      begin klucz := x; licznik := 1; nast := wartownik
      end
  end else
  if w1↑.klucz = x then w1↑.licznik := w1↑.licznik + 1 else
  begin {szukaj}
    repeat w2 := w1; w1 := w2↑.nast
    until w1↑.klucz = x;                                          (4.26)
    if w1 = wartownik then
    begin {wstaw}
      w2 := korzeń; new(korzeń);
      with korzeń↑ do
        begin klucz := x; licznik := 1; nast := w2
        end
    end else
    begin {znaleziony, teraz przestaw}
      w1↑.licznik := w1↑.licznik + 1;
      w2↑.nast := w1↑.nast; w1↑.nast := korzeń; korzeń := w1
    end
  end
end {szukaj}
```



RYSUNEK 4.12
Lista po przestawieniu

Usprawnienie tej metody przeszukiwania silnie zależy od stopnia zgrupowania danych wejściowych. Dla danego współczynnika zgrupowania stopień usprawnienia będzie korzystniejszy w przypadku dużych list. Aby przekonać się, jakiego stopnia usprawnienia można oczekiwać, przeprowadzono eksperyment, stosując powyższy program tworzenia skorowidza do tekstu krótkiego i do tekstu względnie długiego, porównując metodę liniowego uporządkowania listy (4.21) z metodą reorganizowania listy (4.26). Wyniki pomiarów zawarto w tabl. 4.2. Niezbyt szczęśliwie, usprawnienie jest największe w sytuacji, w której należy i tak zastosować inną organizację danych. Powrócimy do tego przykładu w p. 4.4.

TABLICA 4.2
Porównanie metod przeglądania listy

	Test 1	Test 2
Liczba różnych kluczy	53	582
Liczba wystąpień kluczy	315	14341
Czas przeszukiwania z porządkowaniem	6207	3200622
Czas przeszukiwania z przestawianiem	4529	681584
Współczynnik usprawnienia	1,37	4,70

4.3.3. Pewne zastosowanie: sortowanie topologiczne

Odpowiednim przykładem użycia elastycznej, dynamicznej struktury danych jest proces **sortowania topologicznego**. Jest to proces sortowania danych, dla których zdefiniowano **częściowy porządek**, tzn. porządek jest określony dla *niektórych* par danych, ale nie dla wszystkich. Jest to sytuacja dosyć powszechna.

A oto kilka przykładów częściowego uporządkowania.

(1) W słowniku lub glosarium słowa są zdefiniowane za pomocą innych słów. Jeżeli słowo v jest zdefiniowane za pomocą słowa w, to oznaczymy to przez

$v \prec w$. Topologiczne sortowanie słów słownika oznacza ułożenie ich w takim porządku, aby nie występowały odsyłacze do przodu.

(2) Zadanie (np. projekt inżynierski) rozbija się na podzadania. Ukończenie pewnych podzadań musi zwykle poprzedzać wykonanie innych podzadań. Jeżeli podzadanie v musi poprzedzać podzadanie w, to zapiszemy to następująco: $v \prec w$. Topologiczne sortowanie oznacza takie ich uporządkowanie, że przed rozpoczęciem każdego podzadania wszystkie poprzedzające je podzadania zostały wykonane.

(3) W programie zajęć uniwersyteckich pewne zajęcia muszą się odbyć przed innymi, gdyż korzysta się na nich z materiału przekazanego na poprzednich wykładach. Jeżeli zajęcia v stanowią warunek wstępny do odbycia zajęć w, to $v \prec w$. Topologiczne sortowanie oznacza ułożenie zajęć w takim porządku, aby żadne zajęcia nie wskazywały na zajęcia późniejsze jako swój warunek wstępny.

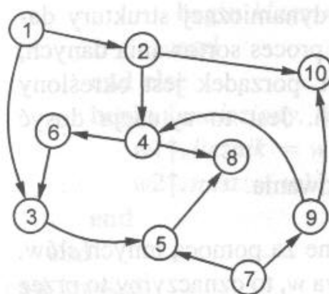
(4) W programie niektóre procedury mogą zawierać odwołania do innych procedur. Jeżeli procedura v jest wywoływana przez procedurę w, to napiszemy $v \prec w$. Topologiczne sortowanie powoduje ułożenie deklaracji procedur w taki sposób, aby nie występowały odwołania do przodu.

Ogólnie, częściowe uporządkowanie zbioru S jest relacją między elementami zbioru S. Oznaczana jest ona symbolem $\prec$ (czytanym jako „poprzedza") i spełnia trzy następujące własności (aksjomaty) dla dowolnych różnych x, y, z ze zbioru S:

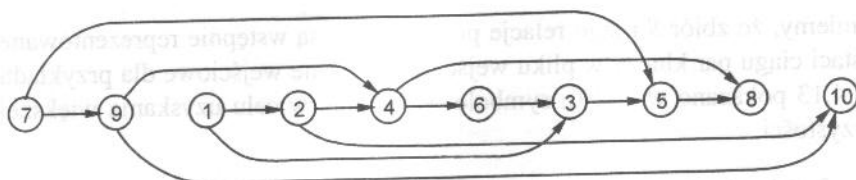
(1) jeśli $x \prec y$ i $y \prec z$, to $x \prec z$ (przechodniość);
(2) jeśli $x \prec y$, to nieprawda, że $y \prec x$ (antysymetria); (4.27)
(3) nieprawda, że $x \prec x$ (przeciwzwrotność).

Z oczywistych powodów przyjmiemy, że zbiory S, które mają być sortowane topologicznie, są skończone. Tak więc częściowy porządek może być zilustrowany za pomocą diagramu lub grafu, w którym węzły oznaczają elementy zbioru S, strzałki zaś reprezentują powiązania porządkujące. Przykład pokazano na rys. 4.13.

Problem sortowania topologicznego polega na osadzeniu częściowego porządku w porządku liniowym. Graficznie oznacza to ustawienie węzłów grafu w rząd, tak aby wszystkie strzałki były skierowane na prawo, jak to pokazano



RYSUNEK 4.13
Zbiór częściowo uporządkowany



RYSUNEK 4.14
Liniowe ustawienie zbioru częściowo uporządkowanego z rys. 4.13

na rys. 4.14. Własności (1) i (2) częściowego porządku zapewniają brak pętli w grafie. Jest to właśnie warunek wstępny, przy którym takie osadzenie w porządku liniowym jest możliwe.

Jak postępować, aby znaleźć jeden z możliwych liniowych porządków? Recepta jest całkiem prosta. Zaczniemy od wyboru obiektu, który nie jest poprzedzany przez żaden inny (musi taki istnieć, gdyż w przeciwnym razie musiałaby istnieć pętla). Jest on umieszczony na początku listy wynikowej i usunięty ze zbioru S. Pozostał nam w dalszym ciągu zbiór częściowo uporządkowany, do którego stosujemy ponownie ten sam algorytm dopóty, dopóki zbiór nie będzie pusty.

W celu ściślejszego opisania tego algorytmu musimy ustalić strukturę danych oraz reprezentację zbioru S i jego uporządkowanie. Wybór reprezentacji jest określony przez wykonywane operacje, w szczególności operację wybierania elementów, które nie mają poprzedników. Dlatego też każdy obiekt powinien być reprezentowany przez trzy cechy: identyfikujący go klucz, zbiór jego następników i liczbę jego poprzedników. Ponieważ n, czyli liczba elementów w S nie jest podana *a priori*, więc wygodnie jest reprezentować zbiór w postaci listy łączonej (listy z dowiązaniami). W konsekwencji dodatkowa pozycja w opisie każdego obiektu zawiera łącze (dowiązanie) do następnego obiektu w liście. Załóżmy, że klucze są liczbami całkowitymi (niekoniecznie od 1 do n). Analogicznie, zbiór następników każdego obiektu jest reprezentowany przez listę łączoną. Każdy element listy następników jest opisany przez jego identyfikację (odsyłacz do głównej listy) i łącze do następnego elementu tej listy. Jeżeli elementy głównej listy, w której każdy obiekt ze zbioru S występuje tylko raz, nazwiemy *przewodnikami*, a elementy łańcuchów następników nazwiemy *maruderami*, to otrzymamy następujące deklaracje typów danych:

```
type pref = ↑przewodnik;
     mref = ↑maruder;
     przewodnik = record klucz, licznik: integer;
                         ślad: mref;
                         nast: pref
                  end;                                  (4.28)
     maruder = record id: pref;
                      nast: mref
               end
```

Przyjmiemy, że zbiór S i jego relacje porządkujące są wstępnie reprezentowane w postaci ciągu par kluczy w pliku wejściowym. Dane wejściowe dla przykładu z rys. 4.13 pokazano w (4.29); symbole $<$ dodano w celu uzyskania większej przejrzystości.

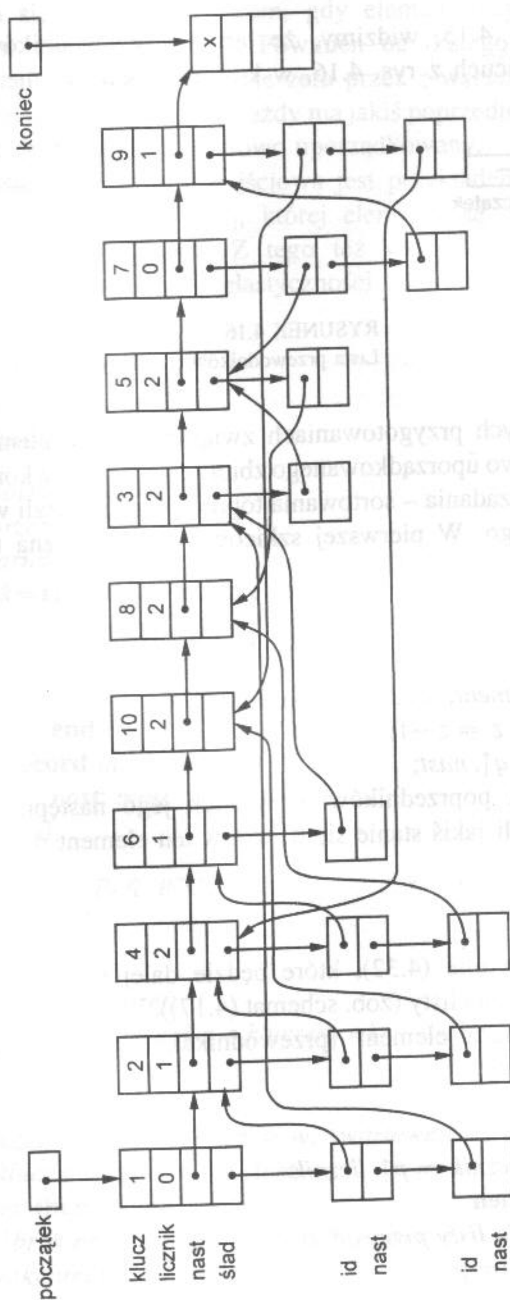
$$\begin{array}{ccccccc} 1<2 & 2<4 & 4<6 & 2<10 & 4<8 & 6<3 & 1<3 \\ 3<5 & 5<8 & 7<5 & 7<9 & 9<4 & 9<10 & \end{array} \tag{4.29}$$

Pierwsza część programu sortowania topologicznego musi wczytać dane z pliku wejściowego i przetworzyć je na strukturę listową. Odbywa się to za pomocą sukcesywnego czytania par kluczy x i y ($x < y$). Oznaczmy wskaźniki do ich reprezentacji w liście przewodników przez p i q. Odpowiednie rekordy trzeba zlokalizować przez przeszukanie listy, a jeżeli nie znajdują się na liście, to należy je na niej umieścić. Zadanie to wykonuje procedura funkcyjna L. Następnie do listy maruderów dla x jest dodana nowa pozycja z identyfikacją y. Licznik poprzedników y zwiększa się o 1. Algorytm ten jest nazwany *fazą wejściową* (4.30). Na rysunku 4.15 zilustrowano strukturę danych powstałą z przetworzenia danych wejściowych (4.29) przez algorytm (4.30). Omawiany fragment programu odwołuje się do funkcji $L(w)$ wyznaczającej odniesienie (referencję) do składowej listy z kluczem w (zob. także program 4.2). Zakładamy, że ciąg wejściowych par kluczy jest ograniczony przez dodatkowe zero.

```
{faza wejściowa} read(x);
new(początek); koniec := początek; z := 0;
while x≠0 do
begin read(y); p := L(x); q := L(y);                              (4.30)
   new(t); t↑. id := q; t↑. nast := p↑. ślad;
   p↑. ślad := t; q↑. licznik := q↑. licznik + 1;
   read(x)
end
```

Po skonstruowaniu w fazie wejściowej struktury danych z rys. 4.15 proces sortowania topologicznego może być wykonany zgodnie z uprzednim opisem. Ze względu na to, że zawiera on powtarzalne wybieranie elementów z zerowym licznikiem poprzedników, słuszne wydaje się wstępne zebranie tych elementów w łańcuch. Ponieważ można zauważyć, że pierwotny łańcuch przewodników nie będzie dalej potrzebny, więc to samo pole *nast* może być użyte ponownie w celu połączenia przewodników z zerową liczbą poprzedników. Taka operacja zastąpienia jednego łańcucha przez inny występuje często w przetwarzaniu list. Szczegółowo jest ona opisana w algorytmie (4.31). Dla wygody buduje ona łańcuch w *przeciwnym kierunku*.

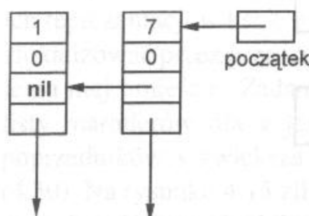
```
{szukaj przewodników z zerem poprzedników}
   p := początek; początek := nil;
   while p≠koniec do
```



RYSUNEK 4.15
Struktura listowa wygenerowana przez program sortowania topologicznego

```
    begin q := p; p := q↑. nast;
        if q↑. licznik=0 then
        begin {wstaw q↑ do nowego łańcucha}                     (4.31)
            q↑. nast := początek; początek := q
        end
    end
```

Patrząc na rys. 4.15, widzimy, że łańcuch przewodników *nast* będzie zastąpiony przez łańcuch z rys. 4.16, w którym wskaźniki nie opisane pozostawiono bez zmian.



RYSUNEK 4.16
Lista przewodników o zerowych licznikach

Po tych wstępnych przygotowaniach związanych z ustaleniem wygodnej reprezentacji częściowo uporządkowanego zbioru *S* możemy na koniec przejść do naszego faktycznego zadania – sortowania topologicznego, czyli wyprodukowania ciągu wyjściowego. W pierwszej szkicowej wersji można to opisać, jak następuje:

```
q := początek;
while q≠nil do
begin {wypisz ten element, a następnie usuń go}
    writeln(q↑. klucz); z := z−1;                                (4.32)
    t := q↑. ślad; q := q↑. nast;
    „Zmniejsz licznik poprzedników wszystkich jego następników na liście
    maruderów t; jeżeli jakiś stanie się 0, wstaw ten element do listy przewod-
    ników q”
end
```

Zadanie w algorytmie (4.32), które będzie dalej precyzowane, zawiera jeszcze jedno przeglądanie listy (zob. schemat (4.17)). W każdym kroku zmienna pomocnicza *p* wyznacza element (przewodnika), którego licznik ma być zmniejszony i testowany.

```
while t≠nil do
begin p := t↑. id; p↑. licznik := p↑. licznik−1;
    if p↑. licznik = 0 then                                      (4.33)
    begin {wstaw p↑ do listy przewodników}
        p↑. nast := q; q := p
```

```
      end;
      t:= t↑.nast
   end
```

To kończy opracowanie programu sortowania topologicznego. Zauważmy, że licznik *z* zlicza przewodników wygenerowanych w fazie wejściowej. Licznik ten zmniejsza się za każdym razem, gdy element listy przewodników jest wypisywany w fazie wyjściowej. Powinien on dlatego mieć wartość 0 po wykonaniu programu. Nieosiągnięcie zera przez *z* wskazuje na pozostawienie w strukturze elementów, z których każdy ma jakiś poprzednik. W tym przypadku, oczywiście, zbiór *S* nie jest częściowo uporządkowany.

Zaprogramowana tu faza wyjściowa jest przykładem procesu działającego na pulsującej liście, tj. takiej, której elementy są wstawiane i usuwane w nieprzewidzianej kolejności. Z tego też względu jest to przykład procesu korzystającego w pełni z elastyczności zapewnianej przez listę jawnie łączoną.

PROGRAM 4.2
Sortowanie topologiczne

```
program topsort(input, output);
type pref=↑przewodnik;
   mref=↑maruder;
   przewodnik=record klucz: integer;
                     licznik: integer;
                     ślad: mref;
                     nast: pref
                  end;
   maruder=record id: pref;
                  nast: mref
               end;

var początek, koniec, p, q: pref;
   t: mref; z: integer;
   x, y: integer;

function L(w: integer): pref;
   {odniesienie do przewodnika z kluczem w}
   var h: pref;

begin h:= początek; koniec↑.klucz:= w; {wartownik}
   while h↑.klucz≠w do h:= h↑.nast;
   if h=koniec then
      begin {brak na liście elementów z kluczem w}
         new(koniec); z:= z+1;
```

```
        h↑. licznik := 0; h↑. ślad := nil; h↑. nast := koniec
      end;
    L := h
end {L};

begin {zainicjuj listę przewodników}
    new(początek); koniec := początek; z=0;

{faza wejściowa} read(x);
    while x≠0 do
    begin read(y); writeln(x, y);
        p := L(x); q := L(y);
        new(t); t↑. id := q; t↑. nast := p↑. ślad;
        p↑. ślad := t; q↑. licznik := q↑. licznik+1;
        read(x)
    end;

{szukaj przewodników z licznikiem=0}
    p := początek; początek := nil;
    while p≠koniec do
    begin q := p; p := p↑. nast;
        if q↑. licznik=0 then
            begin q↑. nast := początek; początek := q
            end
    end;

{faza wyjściowa} q := początek;
    while q≠nil do
    begin writeln(q↑. klucz); z := z-1;
        t := q↑. ślad; q := q↑. nast;
        while t≠nil do
        begin p := t↑. id; p↑. licznik := p↑. licznik-1;
            if p↑. licznik=0 then
            begin {wstaw p↑ do listy q}
                p↑. nast := q; q := p
            end;
            t := t↑. nast
        end
    end;
    if z≠0 then
    writeln ('TEN ZBIÓR NIE JEST CZĘŚCIOWO UPORZĄDKOWANY')
end.
```

4.4. Struktury drzewiaste

4.4.1. Pojęcia podstawowe i definicje

Wiemy już, że ciąg i listę można wygodnie zdefiniować w następujący sposób: ciąg (lista) o typie podstawowym T jest albo

(1) pustym ciągiem (listą), albo
(2) konkatenacją (połączeniem) elementu typu T i ciągu (listy) o typie podstawowym T.

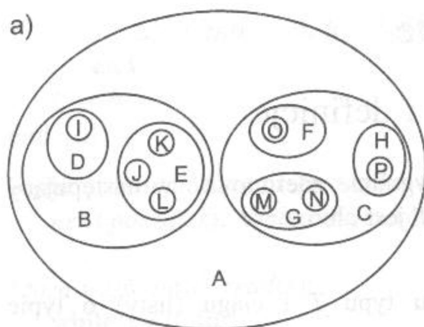
W ten sposób za pomocą rekursji zdefiniujemy zasadę strukturalizacji, a mianowicie tworzenie ciągu lub iterację. Ciągi i iteracje są tak powszechne, że traktowane są zwykle jako podstawowe formy struktur i postępowania. Powinniśmy pamiętać, że *mogą* one być zdefiniowane za pomocą rekursji, podczas gdy odwrotność tego nie jest prawdą; rekursja może być w efektywny i elegancki sposób użyta do definiowania znacznie bardziej złożonych struktur. Znanym przykładem tego są drzewa. Zdefiniujemy strukturę drzewiastą następująco: **struktura drzewiasta** o typie podstawowym T jest albo

(1) strukturą pustą, albo
(2) węzłem typu T ze skończoną liczbą dowiązanych rozłącznych struktur drzewiastych o typie podstawowym T, nazywanych **poddrzewami**.

Z podobieństwa rekurencyjnych definicji ciągów i struktur drzewiastych wynika, że ciąg (lista) jest strukturą drzewiastą, w której każdy węzeł ma co najwyżej jedno „poddrzewo". Dlatego ciąg (lista) nazywany jest także **drzewem zdegenerowanym**. Jest wiele sposobów reprezentacji struktur drzewiastych. Na przykład niech typ podstawowy T obejmuje litery; odpowiednia struktura jest przedstawiona na kilka sposobów na rys. 4.17. Wszystkie reprezentacje mają taką samą strukturę i dlatego są równoważne. Spośród nich graf najwyraźniej ilustruje relacje powiązań. Charakter tych powiązań uzasadnia wybór nazwy struktury – „drzewo". Rzecz dziwna, zwykle rysuje się drzewo odwrócone lub – wyrażając ten fakt inaczej – uwidacznia się korzenie drzewa. Ostatnie sformułowanie jest nieco mylące, gdyż **korzeniem** nazywamy zwykle najwyższy węzeł (A). Chociaż wiemy, że drzewa w swej naturze są tworami bardziej skomplikowanymi od naszych abstrakcji, nasze struktury drzewiaste będziemy w dalszej treści książki nazywać po prostu **drzewami**.

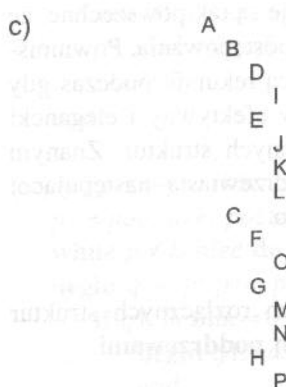
Drzewem uporządkowanym jest drzewo, w którym gałęzie każdego węzła są uporządkowane. Tak więc dwa uporządkowane drzewa na rys. 4.18 są różnymi obiektami. Węzeł y, który znajduje się bezpośrednio poniżej węzła x, nazywamy (bezpośrednim) **potomkiem** x. Jeżeli x znajduje się na **poziomie** i, to mówimy, że y znajduje się na poziomie $i+1$. I odwrotnie, węzeł x nazywany (bezpośrednim) **przodkiem** y. Poziom korzenia drzewa definiujemy jako 1. Maksymalny poziom wszystkich elementów drzewa nazywamy jego **głębokością** lub **wysokością**.

a)

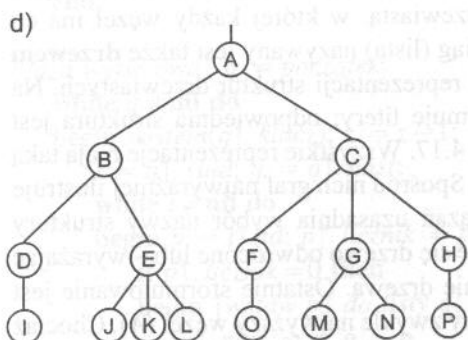


b) (A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))

c)

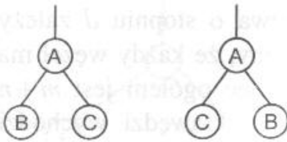


d)



RYSUNEK 4.17
Reprezentacja struktury drzewiastej:
(a) zbiory zagnieżdżone; (b) nawiasy zagnieżdżone; (c) wcięcia; (d) graf

Element nie mający potomków nazywamy **elementem końcowym** lub **liściem**. Element nie będący liściem nazywamy **węzłem wewnętrznym**. Liczbę (bezpośrednich) potomków wewnętrznego węzła nazywamy jego **stopniem**. Maksymalny stopień wszystkich węzłów jest stopniem drzewa. Liczbę gałęzi lub krawędzi, przez które należy przejść od korzenia do węzła x, nazywamy **długością**

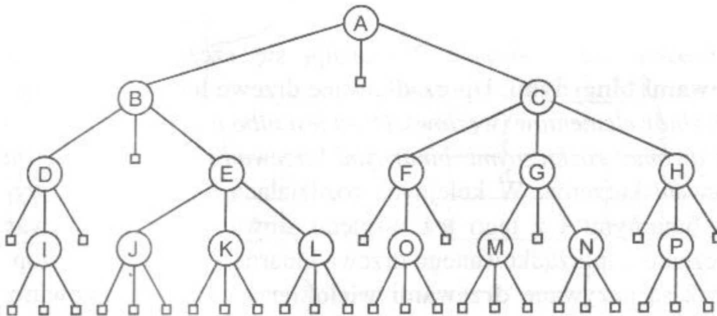


RYSUNEK 4.18
Dwa różne drzewa binarne

ścieżki (lub drogi) x. Długość ścieżki korzenia równa jest 1, jego bezpośredni potomek ma długość ścieżki równą 2 itd. Ogólnie, długość ścieżki węzła na poziomie i jest równa i. Długość ścieżki drzewa definiujemy jako sumę długości ścieżek wszystkich jego składowych. Nazywamy ją także **długością ścieżki wewnętrznej**. Na przykład długość ścieżki wewnętrznej drzewa z rys. 4.17 równa jest 52. Oczywiście średnia długość ścieżki P_I jest dana wzorem

$$P_I = \frac{1}{n} \sum_i n_i \cdot i \tag{4.34}$$

gdzie n_i jest liczbą węzłów na poziomie i. W celu zdefiniowania pojęcia długości ścieżki zewnętrznej rozszerzymy drzewo o specjalny węzeł wszędzie tam, gdzie w oryginalnym drzewie występowało puste poddrzewo. Czyniąc to, przyjmiemy, że wszystkie węzły powinny mieć ten sam stopień, mianowicie stopień drzewa. Rozszerzenie drzewa w ten sposób oznacza uzupełnienie go pustymi gałęziami, przy czym węzły specjalne nie mają – oczywiście – potomków. Drzewo z rys. 4.17 rozszerzone o węzły specjalne pokazano na rys. 4.19; węzły specjalne są reprezentowane przez kwadraciki.



RYSUNEK 4.19
Drzewo trójkowe rozszerzone o specjalne węzły

Długość ścieżki zewnętrznej definiujemy teraz jako sumę długości ścieżek wszystkich specjalnych węzłów. Jeżeli liczba specjalnych węzłów na poziomie i wynosi m_i, to średnia długość zewnętrznej ścieżki P_E jest dana wzorem

$$P_E = \frac{1}{m} \sum_i m_i \cdot i \tag{4.35}$$

Dla drzewa z rys. 4.19 długość ścieżki zewnętrznej wynosi 153.

Liczba specjalnych węzłów m dodanych do drzewa o stopniu d zależy bezpośrednio od liczby pierwotnych węzłów n. Zauważmy, że każdy węzeł ma dokładnie jedną krawędź prowadzącą do niego. Tak więc ogółem jest $m+n$ krawędzi w rozszerzonym drzewie. Z drugiej strony, d krawędzi wychodzi z każdego węzła i żadna z węzła specjalnego. Czyli otrzymujemy $dn+1$ krawędzi (jedna krawędź wejściowa do korzenia). Te dwa wyniki prowadzą do następującej zależności między liczbą specjalnych węzłów m a liczbą pierwotnych węzłów n: $dn+1=m+n$, czyli

$$m=(d-1)n+1 \tag{4.36}$$

Maksymalna liczba węzłów w drzewie o danej wysokości h jest osiągana wtedy, gdy wszystkie węzły mają d poddrzew, z wyjątkiem tych na poziomie h, które nie mają żadnego. W drzewie o stopniu d poziom 1 zawiera 1 węzeł (korzeń), poziom 2 zawiera jego d potomków, poziom 3 zawiera d^2 potomków d węzłów z poziomu 2 itd. Daje to

$$N_d(h)=1+d+d^2+\dots+d^{h-1}=\sum_{i=0}^{h-1} d^i \tag{4.37}$$

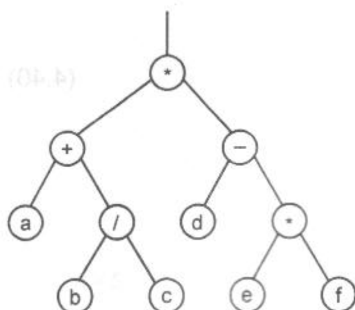
jako wzór na maksymalną liczbę węzłów drzewa o wysokości h i stopniu d. Dla $d=2$ otrzymujemy

$$N_2(h)=\sum_{i=0}^{h-1} 2^i = 2^h - 1 \tag{4.38}$$

Drzewa uporządkowane o stopniu 2 okazują się szczególnie ważne. Nazywane są **drzewami binarnymi**. Uporządkowane drzewo binarne definiujemy jako *skończony zbiór elementów (węzłów), który jest albo pusty, albo zawiera korzeń (węzeł) z dwoma rozłącznymi binarnymi drzewami zwanymi lewym i prawym poddrzewem* korzenia. W kolejnych rozdziałach zajmiemy się wyłącznie drzewami binarnymi i z tego też względu słowa „drzewo" używać będziemy na oznaczenie „uporządkowanego drzewa binarnego". Drzewa o stopniu większym niż 2 są nazywane **drzewami wielokierunkowymi**; omówimy je w p. 4.5.

Znane przykłady drzew *binarnych* to drzewo genealogiczne, gdzie matka i ojciec każdej osoby są traktowani jako potomkowie (!); historia turnieju tenisowego, gdzie każda gra jest węzłem wyznaczonym przez jej zwycięzcę, a dwie poprzednie gry zawodników oznaczają jej potomków; wyrażenie arytmetyczne z dwuargumentowymi operatorami, gdzie każdy operator wyznacza węzeł z jego argumentami jako poddrzewami (zob. rys. 4.20)

Wrócimy teraz do problemu reprezentacji drzew. Jest oczywiste, że ilustrowanie takich rekurencyjnych struktur za pomocą struktur rozgałęziających się sugeruje użycie znanego już nam aparatu wskaźników. Zamiast deklarować zmienne o stałej strukturze drzewiastej, co oczywiście nie dałoby żadnych

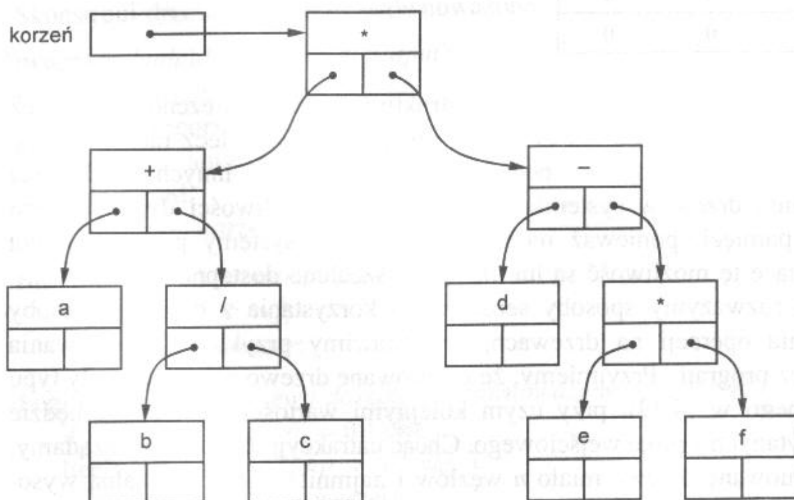


RYSUNEK 4.20
Reprezentacja w postaci drzewa wyrażenia $(a+b/c)*$ $*(d-e*f)$

korzyści, zdefiniujemy *węzły* jako zmienne o stałej strukturze, tj. ustalonego typu, w których stopień drzewa określi liczbę wskaźników odwołujących się do poddrzew węzła. Oczywiście odwołanie do pustego drzewa jest oznaczone przez **nil**. Tak więc składowe drzewa z rys. 4.20 są typu zdefiniowanego następująco:

type *węzeł* = **record** *op*: *char*;
lewe, *prawe*: ↑ *węzeł* (4.39)
end

i drzewo może być wtedy skonstruowane tak, jak to pokazano na rys. 4.21.



RYSUNEK 4.21
Drzewo reprezentowane jako struktura danych

Istnieją sposoby reprezentacji abstrakcyjnej idei struktury drzewiastej za pomocą innych dostępnych typów danych, jak np. tablic. Zwykle postępuje się w ten sposób we wszystkich językach, które nie zapewniają możliwości dynamicznego przydziału pamięci dla składowych i odwoływania się do nich poprzez wskaźniki. W tym przypadku drzewo z rys. 4.20 może być reprezentowane za pomocą zmiennej tablicowej zadeklarowanej następująco:

```
t: array [1. .11] of
      record op: char;                                    (4.40)
          lewe, prawe: integer
      end
```

Wartości kolejnych składowych pokazano w tabl. 4.3.

TABLICA 4.3
Drzewo reprezentowane przez tablicę

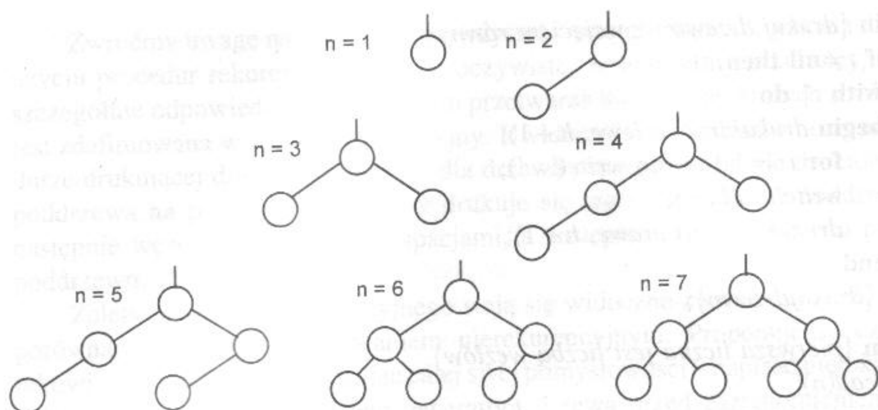
1	*	2	3
2	+	6	4
3	−	9	5
4	/	7	8
5	*	10	11
6	*a*	0	0
7	*b*	0	0
8	*c*	0	0
9	*d*	0	0
10	*e*	0	0
11	*f*	0	0

Chociaż podstawowa, abstrakcyjna struktura danych reprezentowana przez tablicę *t* jest drzewem, nie nazwiemy jej jednak drzewem, lecz raczej tablicą, zgodnie z jej deklaracją. Nie będziemy dalej rozważać innych możliwości przedstawiania drzew w systemach nie mających możliwości dynamicznego przydziału pamięci, ponieważ możemy przyjąć, że systemy programowania i języki mające tę możliwość są lub będą powszechnie dostępne.

Zanim rozważymy sposoby sensownego korzystania z drzew i sposoby wykonywania operacji na drzewach, przedstawimy przykład konstruowania drzewa przez program. Przyjmiemy, że generowane drzewo ma mieć węzły typu zdefiniowanego w (4.39), przy czym kolejnymi wartościami węzłów będzie n liczb wczytanych z pliku wejściowego. Chcąc uatrakcyjnić problem, zażądamy, aby skonstruowane drzewo miało n węzłów i najmniejszą dopuszczalną wysokość.

W celu uzyskania najmniejszej wysokości przy danej liczbie węzłów należy przydzielić możliwie największą liczbę węzłów na wszystkich poziomach z wyjątkiem ostatniego. Można to łatwo osiągnąć przez rozdzielenie tej samej liczby wprowadzanych węzłów na prawo i lewo dla każdego węzła w drzewie. Otrzymane w ten sposób struktury drzew dla $n = 1, 2, ..., 7$ przedstawiono na rys. 4.22.

Zasadę równego rozkładu dla znanej liczby n węzłów można najlepiej sformułować za pomocą pojęć rekurencyjnych.



RYSUNEK 4.22
Drzewa doskonale zrównoważone

(1) Wykorzystaj jeden węzeł na korzeń.
(2) Zbuduj lewe poddrzewo z $nl = n$ **div** 2 węzłami w niniejszy sposób.
(3) Zbuduj prawe poddrzewo z $np = n - nl - 1$ węzłami w niniejszy sposób.

PROGRAM 4.3
Skonstruuj drzewo doskonale zrównoważone

```
program budujdrzewo(input, output);
type ref=↑węzeł;
   węzeł=record klucz: integer;
            lewe, prawe: ref
         end;
var n: integer; korzeń: ref;

function drzewo(n: integer): ref;
   var nowywęzeł: ref;
      x, nl, np: integer;

begin {skonstruuj drzewo doskonale zrównoważone o n węzłach}
   if n=0 then drzewo:= nil else
   begin nl:=n div 2; np:=n-nl-1;
      read(x); new(nowywęzeł);
      with nowywęzeł↑ do
         begin klucz:=x; lewe:=drzewo(nl); prawe:=drzewo(np)
         end;
      drzewo:=nowywęzeł
   end
end {drzewo};

procedure drukujdrzewo(t: ref; h: integer);
   var i: integer;
```

```
begin {drukuj drzewo t z wcięciem równym h}
   if t≠nil then
   with t↑ do
   begin drukujdrzewo(lewe, h+1);
      for i:= 1 to h do write('   ');
      writeln (klucz);
      drukujdrzewo (prawe, h+1);
   end
end {drukujdrzewo};

begin {pierwsza liczba jest liczbą węzłów}
   read(n);
   korzeń:= drzewo (n);
   drukujdrzewo (korzeń, 0)
end.
```

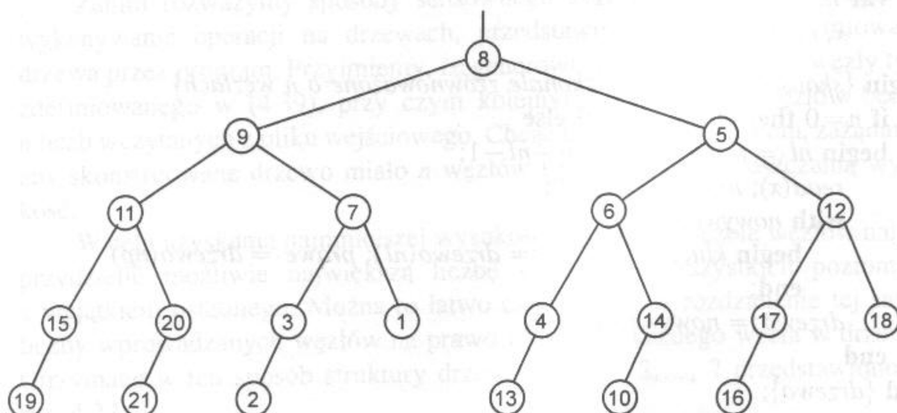
Zasada ta jest wyrażona za pomocą procedury rekurencyjnej będącej częścią programu 4.3, który wczytuje plik wejściowy i konstruuje drzewo doskonale zrównoważone. Wprowadzamy następującą definicję:

> Drzewo jest **doskonale zrównoważone** (dokładnie wyważone; ang. *perfectly balanced*), jeżeli dla każdego węzła liczby węzłów w jego lewym i prawym poddrzewie różnią się co najwyżej o 1.

Przyjmijmy np. następujące dane wejściowe dla drzewa o 21 węzłach:

21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

Program 4.3 skonstruuje wtedy drzewo doskonale zrównoważone pokazane na rys. 4.23.



RYSUNEK 4.23
Drzewo wygenerowane przez program 4.3

Zwróćmy uwagę na prostotę i przejrzystość tego programu, uzyskaną dzięki użyciu procedur rekurencyjnych. Jest oczywiste, że algorytmy rekurencyjne są szczególnie odpowiednie, jeśli program przetwarza informacje, których struktura jest zdefiniowana w sposób rekurencyjny. Uwidacznia się to ponownie w procedurze drukującej drzewo wynikowe: dla drzewa pustego nic się nie drukuje, dla poddrzewa na poziomie L najpierw drukuje się jego własne lewe poddrzewo, następnie węzeł, poprzedzony L spacjami, a następnie drukuje się jego prawe poddrzewo.

Zalety algorytmu rekurencyjnego stają się widoczne zwłaszcza wtedy, gdy porównamy go ze sformułowaniem nierekurencyjnym. Proponujemy czytelnikowi, aby spróbował – używając całej swej pomysłowości – napisać nierekurencyjny odpowiednik powyższego generatora drzewa przed zaznajomieniem się z programem (4.41). Program ten jest przedstawiony bez dalszych komentarzy i może stanowić dla czytelnika próbę sił przy odkrywaniu, w jaki sposób i dlaczego działa.

```
program budujdrzewo(input, output);
type ref=↑węzeł;
  węzeł=record klucz: integer;
              lewe, prawe: ref
        end;
var i, n, nl, np, x: integer;
  korzeń, p, q, r, fik: ref;
  s: array [1..30] of {stos}
      record n: integer; rf: ref
      end;
begin {pierwsza liczba jest liczbą węzłów}                    (4.41)
  read(n); new(korzeń); new(fik); {pusty}
  i:=1; s[1].n:=n; s[1].rf:=korzeń;
  repeat n:=s[i].n; p:=s[i].rf; i:=i-1; {„pop" – zmniejsz}
    if n=0 then p↑.prawe:=nil else
    begin p:=fik;
      repeat nl:=n div 2; np:=n-nl-1;
        read(x); new(q); q↑.klucz:=x;
        i:=i+1; s[i].n:=np; s[i].rf:=q; {„push" – zwiększ}
        n:=nl; p↑.lewe:=q; p:=q
      until n=0;
      q↑.lewe:=nil; p↑.prawe:=fik↑.lewe
    end
  until i=0;
  drukujdrzewo(korzeń↑.prawe, 0)
end.
```

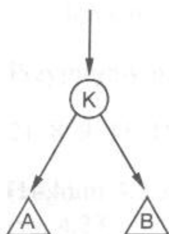
4.4.2. Podstawowe operacje na drzewach binarnych

Istnieje wiele zadań, które należy wykonywać na strukturze drzewiastej; czynnością powszechną jest wykonywanie danej operacji P na każdym z elementów drzewa. P stanowi wtedy parametr ogólniejszego zadania – odwiedzenia wszystkich węzłów – nazywanego zwykle **przeglądaniem drzewa** (przechodzeniem drzewa; ang. *tree traversal*).

Jeżeli rozważymy nasze zadanie jako jeden proces sekwencyjny, to odwiedzanie pojedynczych węzłów odbywa się w pewnym porządku i można uważać, że węzły zostały ułożone wzdłuż linii. W rzeczywistości opis wielu algorytmów znacznie się uprości, jeśli będziemy mówić o przetwarzaniu *następnego* elementu drzewa wyznaczonego zgodnie z tym porządkiem.

Istnieją trzy podstawowe uporządkowania wynikające w sposób naturalny ze struktury drzew. Podobnie jak samą strukturę drzewiastą, można je zręcznie wyrazić za pomocą pojęć rekurencyjnych. Powołując się na drzewo binarne z rys. 4.24, w którym K oznacza korzeń, A i B zaś – lewe i prawe poddrzewa, wyróżniamy trzy porządki:

(1) *preorder* (*wzdłużny*): K, A, B (odwiedzamy korzeń *przed* poddrzewami);
(2) *inorder* (*poprzeczny*): A, K, B;
(3) *postorder* (*wsteczny*): A, B, K (odwiedzamy korzeń *po* poddrzewach).



Rysunek 4.24
Drzewo binarne

Przeglądając drzewo z rys. 4.20 i zapisując występujące w węzłach symbole w kolejności ich napotkania, otrzymujemy następujące porządki:

(1) *preorder* (*wzdłużny*): $* + a \; / \; b \; c \; - d \; * \; e \; f$;
(2) *inorder* (*poprzeczny*): $a + b \; / \; c \; * \; d - e \; * \; f$;
(3) *postorder* (*wsteczny*): $a \; b \; c \; / + d \; e \; f \; * - *$.

Rozpoznajemy tu trzy rodzaje zapisu wyrażeń: wzdłużne przeglądanie drzewa daje **notację przedrostkową**, wsteczne – **przyrostkową**, a przeglądanie poprzeczne pokrywa się z powszechnie stosowaną notacją **wrostkową**, chociaż brakuje nawiasów służących do określenia pierwszeństwa operatorów.

Sformułujemy teraz trzy metody przeglądania drzewa przez trzy konkretne programy z jawnym parametrem t, oznaczającym drzewo, na którym będziemy działać, i domyślnym parametrem p, wyznaczającym operację wykonywaną na każdym węźle. Przyjmijmy następujące definicje:

```
type ref = ↑ węzeł;
   węzeł = record...
             lewe, prawe: ref                                    (4.42)
         end
```

Wspomniane już trzy metody dają się teraz bez trudu sformułować jako procedury *rekurencyjne*. Dowodzi to ponownie faktu, że operacje na rekurencyjnie zdefiniowanych strukturach danych najwygodniej jest definiować w postaci algorytmów rekurencyjnych.

```
procedure wzdłużny(t: ref);
begin if t ≠ nil then
      begin P(t);
            wzdłużny(t↑. lewe);                                  (4.43)
            wzdłużny(t↑. prawe)
      end
end

procedure poprzeczny(t: ref);
begin if t ≠ nil then
      begin poprzeczny(t↑. lewe);
            P(t);                                                (4.44)
            poprzeczny(t↑. prawe);
      end
end

procedure wsteczny(t: ref);
begin if t ≠ nil then
      begin wsteczny(t↑. lewe);                                  (4.45)
            wsteczny(t↑. prawe);
            P(t)
      end
end
```

Zauważmy, że wskaźnik *t* jest przekazany procedurze przez wartość. Wyraża to fakt, że odpowiednia wielkość jest *odniesieniem* (*referencją*) do rozważanego poddrzewa, a *nie* zmienną, której wartością jest wskaźnik; przekazanie *t* przez zmienną mogłoby spowodować zmianę tej wartości.

Przykładem procedury przeglądania drzewa jest procedura, która drukuje drzewo, z odpowiednim wcięciem wskazującym poziom każdego węzła (zob. program 4.3).

Drzewa binarne są często stosowane do reprezentowania zbioru danych, którego elementy mają być wybierane za pomocą identyfikującego je klucza. Jeżeli drzewo jest tak zorganizowane, że dla każdego węzła t_i wszystkie klucze z lewego poddrzewa węzła t_i są mniejsze od klucza węzła t_i, a klucze z prawego

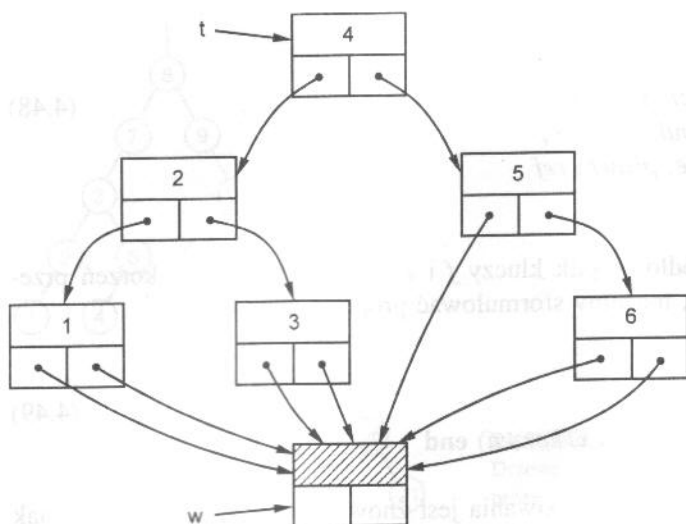
poddrzewa są od niego większe, to takie drzewo jest nazywane **drzewem poszukiwań**. W drzewie takim można znaleźć określony klucz, posuwając się wzdłuż ścieżki poszukiwania – począwszy od korzenia – i przechodząc do lewego lub prawego poddrzewa danego węzła w zależności tylko od wartości klucza w tym węźle. Jak już wiemy, z n elementów można zbudować drzewo binarnę o wysokości $\log n$. Dlatego też poszukiwanie pośród n elementów może wymagać co najwyżej $\log n$ porównań, jeżeli drzewo jest doskonale zrównoważone. Oczywiście drzewo jest bardziej odpowiednią strukturą do organizacji takiego zbioru danych niż lista liniowa omówiona w poprzednim punkcie.

Ponieważ takie poszukiwanie przebiega wzdłuż pojedynczej ścieżki od korzenia do wybranego węzła, to można je łatwo zaprogramować za pomocą iteracji (4.46).

```
function lok(x: integer; t: ref): ref;
  var znaleziony: boolean;
begin znaleziony := false;
  while (t ≠ nil) ∧ ¬ znaleziony do
  begin
    if t↑.klucz = x then znaleziony := true else                (4.46)
    if t↑.klucz > x then t := t↑.lewe else t := t↑.prawe
  end;
  lok := t
end
```

Funkcja *lok*(x, t) przyjmuje wartość **nil** wtedy, gdy w drzewie o korzeniu t nie znaleziono klucza x. Podobnie jak w przypadku przeszukiwania list, ze względu na złożoność warunku zakończenia spróbujemy znaleźć lepsze rozwiązanie. Polega ono na użyciu *wartownika* na końcu listy. Metoda ta daje się również zastosować w przypadku drzewa. Użycie wskaźników pozwala na kończenie wszystkich gałęzi tym samym wartownikiem. Uzyskana w ten sposób struktura nie jest już takim drzewem, jakie rozważaliśmy dotychczas, ale drzewem o liściach przywiązanych do punktu zakotwiczenia (rys. 4.25). Wartownik może być traktowany jako wspólny reprezentant wszystkich zewnętrznych węzłów, o które drzewo pierwotne zostało rozszerzone (zob. rys. 4.19). Uzyskana w ten sposób uproszczona procedura przeszukiwania jest pokazana w (4.47).

```
function lok(x: integer; t: ref): ref;
begin w↑.klucz := x; {wartownik}
  while t↑.klucz ≠ x do                                        (4.47)
    if x < t↑.klucz then t := t↑.lewe else t := t↑.prawe;
  lok := t
end
```



RYSUNEK 4.25
Drzewo poszukiwań z wartownikiem

Zauważmy, że w tym przypadku *lok*(x, t) przyjmuje wartość w, tj. wartość wskaźnika do wartownika wtedy, gdy w drzewie o korzeniu t nie znaleziono klucza x; w przejmuje więc rolę wskaźnika **nil**.

4.4.3. Przeszukiwanie drzewa z wstawianiem

Trudno pokazać moc metody dynamicznego przydziału pamięci z dostępem za pośrednictwem wskaźników w przykładach, w których konkretny zbiór danych jest konstruowany, a następnie przechowywany bez zmian. Bardziej odpowiednimi przykładami są te zastosowania, w których struktura drzewa zmienia się, tzn. rośnie lub/i kurczy się podczas wykonywania programu. Właśnie z tego powodu niewłaściwe są inne reprezentacje danych, takie jak tablice, a *tym* właściwym rozwiązaniem jest drzewo z elementami łączonymi przez wskaźniki.

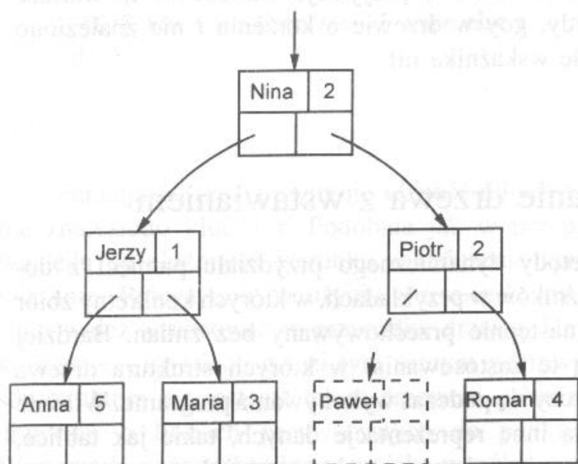
Najpierw omówimy przypadek silnie rosnącego, ale nigdy nie kurczącego się drzewa. Odpowiednim przykładem jest problem skorowidza, o którym już wspominaliśmy w związku z listami łączonymi. Zajmiemy się nim ponownie. W problemie tym dany jest ciąg słów i należy określić liczbę wystąpień każdego z nich. Oznacza to, że – poczynając od pustego drzewa – szukamy w drzewie każdego słowa. Po jego znalezieniu zwiększa się licznik wystąpień; w przeciwnym razie jest ono wstawiane jako nowe słowo (z licznikiem równym 1). Proces ten nazwiemy **przeszukiwaniem drzewa z wstawianiem**. Przyjęto następujące definicje typów danych:

type *ref* = ↑*słowo*;
 słowo = **record**
  klucz: *integer*;   (4.48)
  licznik: *integer*;
  lewe, *prawe*: *ref*
 end

Przyjmując dalej źródłowy plik kluczy *f* i zmienną oznaczającą korzeń przeszukiwanego drzewa, możemy sformułować program jako

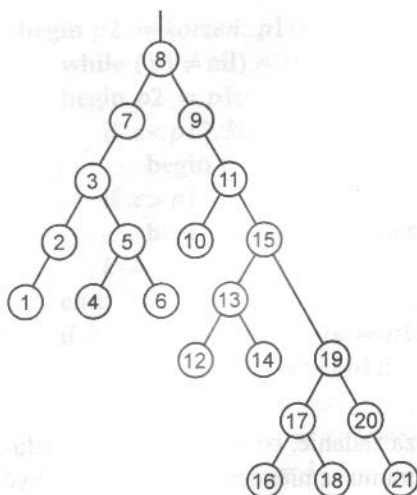
reset(*f*);
while ¬ *eof*(*f*) **do**   (4.49)
 begin *read*(*f*, *x*); *szukaj*(*x*, *korzeń*) **end**

Znajdowanie ścieżki poszukiwania jest znowu bardzo proste. Jeśli jednak prowadzi ona do „martwego końca" (tj. pustego poddrzewa oznaczonego przez wartość wskaźnika **nil**), to dane słowo musi być wstawione do drzewa na miejsce pustego poddrzewa. Rozważmy np. drzewo binarne z rys. 4.26 i wstawienie słowa „Paweł". Wynik oznaczono liniami przerywanymi na tym samym rysunku.



RYSUNEK 4.26
Wstawianie w drzewie binarnym uporządkowanym

Cała operacja jest przedstawiona w programie 4.4. Proces przeszukiwania jest sformułowany w postaci procedury rekurencyjnej. Zwróćmy uwagę, że parametr *p* jest przekazywany przez zmienną, a *nie* przez wartość. Jest to istotne, gdyż przy wstawianiu *zmiennej* mającej uprzednio wartość **nil** musi być przypisana nowa wartość wskaźnika. Korzystając z ciągu wejściowego 21 liczb, użytych przez program 4.3 do konstrukcji drzewa z rys. 4.23, program 4.4 daje w wyniku drzewo binarne pokazane na rys. 4.27.



RYSUNEK 4.27
Drzewo poszukiwań wygenerowane przez program 4.4

PROGRAM 4.4
Przeszukiwanie drzewa z wstawieniem

```
program dszuk(input, output);
{przeszukiwanie drzewa binarnego z wstawianiem}
type ref = ↑słowo;
    słowo = record klucz: integer;
                licznik: integer;
                lewe, prawe: ref;
            end;
var korzeń: ref; k: integer;
procedure drukujdrzewo(w: ref; l: integer);
    var i: integer;
begin if w ≠ nil then
    with w↑ do
    begin drukujdrzewo(lewe, l+1);
        for i := 1 to l do write('   ');
        writeln(klucz);
        drukujdrzewo(prawe, l+1)
    end
end;
procedure szukaj(x: integer; var p: ref);
begin
    if p = nil then
    begin {brak słowa w drzewie; wstaw je}
        new(p);
        with p↑ do
            begin klucz := x; licznik := 1; lewe := nil; prawe := nil
            end
```

```
    end else
    if x<p↑.klucz then szukaj(x, p↑.lewe) else
    if x>p↑.klucz then szukaj(x, p↑.prawe) else
        p↑.licznik := p↑.licznik+1
end {szukaj};
begin korzeń := nil;
    while ¬ eof (input) do
        begin read(k); szukaj(k, korzeń)
        end;
    drukujdrzewo(korzeń, 0)
end.
```

Użycie wartownika ponownie nieco upraszcza zadanie, co pokazano w procedurze (4.50). Oczywiście na początku programu zmiennej *korzeń* musi być przypisana wartość wskaźnika do wartownika, a nie wartość **nil**, a przed każdym przeszukiwaniem wartość poszukiwana x musi być podstawiona na pole klucza wartownika.

```
procedure szukaj(x: integer; var p: ref);
begin
    if x<p↑. klucz then szukaj(x, p↑. lewe) else
    if x>p↑. klucz then szukaj(x, p↑. prawe) else
    if p ≠ w then p↑. licznik := p↑. licznik+1 else            (4.50)
        begin {wstaw} new (p);
            with p↑ do
            begin klucz := x; lewe := w; prawe := w; licznik := 1
            end
        end
end
```

Ponownie, ale po raz ostatni, opracujemy alternatywną wersję tego programu, rezygnując ze stosowania rekursji. Uniknięcie rekursji nie jest już tak proste, jak wtedy, gdy nie uwzględnialiśmy wstawiania, ponieważ należy teraz pamiętać co najmniej jeden krok wstecz przeglądanej ścieżki. Zapamiętywanie zostało w programie 4.4 zrealizowane przez użycie parametru przekazywanego przez zmienną.

Aby poprawnie dołączyć wstawianą składową, musimy znać odniesienie do jej poprzednika (przodka) i wiedzieć, czy składowa ma być wstawiona jako lewe czy też prawe poddrzewo przodka. W tym celu wprowadzono dwie zmienne: $p2$ i k (kierunek).

```
procedure szukaj (x: integer; korzeń: ref);
    var p1, p2: ref; k: integer;
```

```
begin p2 := korzeń; p1 := p2↑.prawe; k := 1;
  while (p1 ≠ nil) ∧ (k ≠ 0) do
  begin p2 := p1;
    if x < p1↑.klucz then
      begin p1 := p1↑.lewe; k := -1 end else                    (4.51)
    if x > p1↑.klucz then
      begin p1 := p1↑.prawe; k := 1 end else
    k := 0
  end;
  if k = 0 then p1↑.licznik := p1↑.licznik + 1 else
    begin {wstaw} new(p1);
      with p1↑ do
      begin klucz := x; lewe := nil; prawe := nil;
        licznik := 1
      end;
      if k < 0 then p2↑.lewe := p1 else p2↑.prawe := p1
    end
end
```

Podobnie jak przy przeszukiwaniu listy z wstawieniem, wprowadzamy dwa wskaźniki *p*1 i *p*2, które przebiegają drogę przeszukiwania tak, że *p*2 zawsze wyznacza przodka *p*1↑. Poprawne zainicjowanie procesu przeszukiwania wymaga wtedy użycia pomocniczego, fikcyjnego elementu, wyznaczanego przez wskaźnik zwany *korzeniem*. Początek faktycznego drzewa poszukiwań jest wyznaczony przez wskaźnik *korzeń*↑.*prawe*. W związku z tym program musi rozpocząć się od następujących instrukcji:

new(*korzeń*); *korzeń*↑.*prawe* := **nil**

zamiast pierwotnej instrukcji przypisania

korzeń := **nil**

Jakkolwiek celem tego algorytmu jest przeszukiwanie, można go użyć równie dobrze do sortowania. W rzeczywistości przypomina on bardzo metodę sortowania przez wstawianie, a ze względu na użycie drzewa, zamiast tablicy, znika problem przestawiania składowych w momencie wstawiania. Sortowanie za pomocą drzewa może być zaprogramowane prawie tak efektywnie, jak najlepsze znane metody sortowania tablic. Należy jednak zachować pewne środki ostrożności. Oczywiście należy teraz inaczej traktować przypadek napotkania pasującego klucza. Jeżeli warunek $x = p\uparrow.klucz$ potraktujemy tak jak warunek $x > p\uparrow.klucz$, to algorytm będzie realizował stabilną metodę sortowania, tj. dane o identycznych kluczach pojawią się podczas przeglądania drzewa w kolejności ich wstawiania.

Istnieją na ogół lepsze sposoby sortowania, ale w zastosowaniach wymagających jednocześnie przeszukiwania i sortowania jest gorąco polecany algorytm przeszukiwania drzewa z wstawianiem. W rzeczywistości jest on często stosowa-

ny w kompilatorach i bankach danych do organizowania danych w celu ich przechowywania i odszukiwania. Odpowiednim przykładem jest konstrukcja **indeksu odsyłaczy** dla danego tekstu. Prześledźmy ten problem dokładniej.

Naszym zadaniem jest skonstruowanie programu, który (podczas czytania tekstu f i drukowania go wraz z numeracją wierszy) przechowuje wszystkie słowa tego tekstu, zapamiętując również numery wierszy, w których każde słowo wystąpiło. Po zakończeniu przeglądania tekstu ma być wygenerowana tablica zawierająca wszystkie słowa w porządku alfabetycznym z listami ich wystąpień.

Oczywiście drzewo poszukiwań (zwane także **drzewem leksykograficznym**) jest najodpowiedniejsze do reprezentowania słów rozpoznanych w tekście. Każdy węzeł zawiera teraz nie tylko słowo jako wartość klucza, ale jest także początkiem listy numerów wierszy. Każdy zapis wystąpienia nazwiemy *pozycją*. W omawianym przykładzie uwzględniamy więc zarówno drzewa, jak i listy liniowe. Program składa się z dwóch zasadniczych części (zob. program 4.5), a mianowicie: z fazy przeglądania i z fazy drukowania tablicy. Druga faza jest bezpośrednim zastosowaniem procedury przeglądania drzewa, w której przejście przez każdy węzeł powoduje drukowanie wartości klucza (słowa) i przebiegnięcie związanej z nim listy numerów wierszy (pozycji).

PROGRAM 4.5
Generator odsyłaczy

```
program odsyłacz(f, output);
{generator odsyłaczy wykorzystujący drzewo binarne}
const c1 =   10; {długość słów}
      c2 =    8; {ilość liczb w wierszu}
      c3 =    6; {ilość cyfr w liczbie}
      c4 = 9999; {maksymalny numer wiersza}
type alfa = packed array [1. .c1] of char;
     sref = ↑słowo;
     pref = ↑pozycja;
     słowo = record klucz: alfa;
                    pierwszy, ostatni: pref;
                    lewe, prawe: sref
             end;
     pozycja = packed record
                    lno : 0. .c4;
                    nast : pref
             end;
var korzeń: sref;
    k, k1 : integer;
    n: integer;     {bieżący numer wiersza}
    id: alfa;
```

```
        f: text;
        a: array [1. .c1] of char;

procedure szukaj(var s1: sref);
    var s: sref; x: pref;
begin s := s1;
    if s = nil then
    begin new(s); new(x);
        with s↑ do
        begin klucz := id; lewe := nil; prawe := nil;
            pierwszy := x; ostatni := x
        end;
        x↑. lno := n; x↑.nast := nil; s1 := s
    end else
    if id<s↑. klucz then szukaj(s↑. lewe) else
    if id>s↑. klucz then szukaj(s↑. prawe) else
    begin new(x); x↑. lno := n; x↑.nast := nil;
        s↑. ostatni↑. nast := x; s↑. ostatni := x
    end
end {szukaj};
procedure drukujdrzewo(s: sref);
    procedure drukujsłowo(s: słowo);
        var l: integer; x: pref;
    begin write('  ', s. klucz);
        x := s. pierwszy; l := 0;
        repeat if l = c2 then
                begin writeln;
                    l := 0; write ('  ': c1+1)
                end;
            l := l+1; write (x↑. lno: c3); x := x↑. nast
        until x = nil;
        writeln
    end {drukujsłowo};
begin if s≠nil then
    begin drukujdrzewo (s↑. lewe);
        drukujsłowo (s↑); drukujdrzewo (s↑. prawe)
    end
end {drukujdrzewo};

begin korzeń := nil; n := 0; k1 := c1;
    page (output); reset (f);
    while ¬ eof (f) do
    begin if n = c4 then n := 0;
        n := n+1; write (n: c3); {następny wiersz}
        write ('  ');
```

```
while ¬ eoln(f) do
begin {przejrzyj niepusty wiersz}
   if f↑ in['A'..'Z'] then
   begin k := 0;
      repeat if k<c1 then
         begin k := k+1; a[k] := f↑;
         end;
         write(f↑); get(f)
      until ¬ (f↑ in ['A'..'Z', '0'..'9']);
      if k≥k1 then k1 := k else
      repeat a[k1] := ' '; k1 := k1 − 1
      until k1 = k;
      pack(a, 1, id); szukaj(korzeń)
   end else
   begin {sprawdź, czy apostrof lub komentarz}
      if f↑ = '''' then
         repeat write(f↑); get(f)
         until f↑ = '''' else
      if f↑ = '{' then
         repeat write(f↑); get(f)
         until f↑ = '}';
      write(f↑); get(f)
   end
end;
writeln; get(f)
end;
page(output); drukujdrzewo(korzeń);
end.
```

Niżej podano dalsze wyjaśnienia dotyczące *generatora odsyłaczy* z programu 4.5.

(1) Słowo jest dowolnym ciągiem liter i cyfr, rozpoczynającym się od litery.
(2) Tylko $c1$ pierwszych znaków pamięta się jako wartość klucza. Wynika stąd, że dwa słowa nie różniące się na pierwszych $c1$ pozycjach są traktowane jako identyczne.
(3) $c1$ znaków upakowuje się do tablicy *id* (typu *alfa*). Jeżeli $c1$ jest dostatecznie małe, to wiele komputerów może porównywać takie upakowane tablice za pomocą pojedynczej instrukcji.
(4) Zmiennej $k1$ używa się jako indeksu przy opisie następującego warunku niezmienniczego dotyczącego znakowego buforu a:

   $a[i] = ' '$ dla $i = k1 + 1, ..., c1$

   Słowa zawierające mniej niż $c1$ znaków są rozszerzone o odpowiednią liczbę spacji.

(5) Wskazane jest drukowanie numerów wierszy w porządku rosnącym w indeksie odsyłaczy. Dlatego też listy pozycji muszą być generowane zgodnie z kolejnością ich przeglądania przy drukowaniu. Wymaganie to sugeruje użycie dwóch wskaźników w każdym węźle: jednego wskazującego na pierwszą, drugiego – na ostatnią pozycję na liście.
(6) Program przeglądania jest tak skonstruowany, że słowa w apostrofach i w komentarzach są pomijane; zakładamy, że cytaty i komentarze nie wykraczają poza koniec wiersza tekstu.

W tablicy 4.4 pokazano wyniki przetwarzania tekstu krótkiego programu.

TABLICA 4.4
Przykładowe wyniki programu 4.5

```
 1  PROGRAM PERMUTE(OUTPUT);
 2     CONSTN = 4;
 3     VAR I: INTEGER;
 4         A: ARRAY [1..N] OF INTEGER;
 5
 6     PROCEDURE PRINT;
 7        VAR I: INTEGER;
 8     BEGIN FOR I := 1 TO N DO WRITE(A[I]:3);
 9        WRITELN
10     END {PRINT};
11
12     PROCEDURE PERM(K: INTEGER);
13        VAR I, X: INTEGER;
14     BEGIN
15        IF K = 1 THEN PRINT ELSE
16        BEGIN PERM(K-1);
17           FOR I := 1 TO K-1 DO
18           BEGIN X := A[I]; A[I] := A[K]; A[K] := X;
19              PERM(K-1);
20                 X := A[I]; A[I] := A[K]; A[K] := X;
21           END
22        END
23     END {PERM};
24
25  BEGIN
26     FOR I := 1 TO N DO A[I] := I;
27     PERM(N)
28  END.

ARRAY     4
A         4    8   18   18   18   18   20   20
         20   20   26
BEGIN     8   14   16   18   25
CONST     2
```

TABLICA 4.4 (cd.)

DO	8	17	26					
ELSE	15							
END	10	21	22	23	28			
FOR	8	17	26					
IF	15							
INTEGER	3	4	7	12	13			
I	3	7	8	8	13	17	18	18
	20	20	26	26	26			
K	12	15	16	17	18	18	19	20
	20							
N	2	4	8	26	27			
OF	4							
OUTPUT	1							
PERMUTE	1							
PERM	12	16	19	27				
PRINT	6	15						
PROCEDURE	6	12						
PROGRAM	1							
THEN	15							
TO	8	17	26					
VAR	3	7	13					
WRITELN	9							
WRITE	8							
X	13	18	18	20	20			

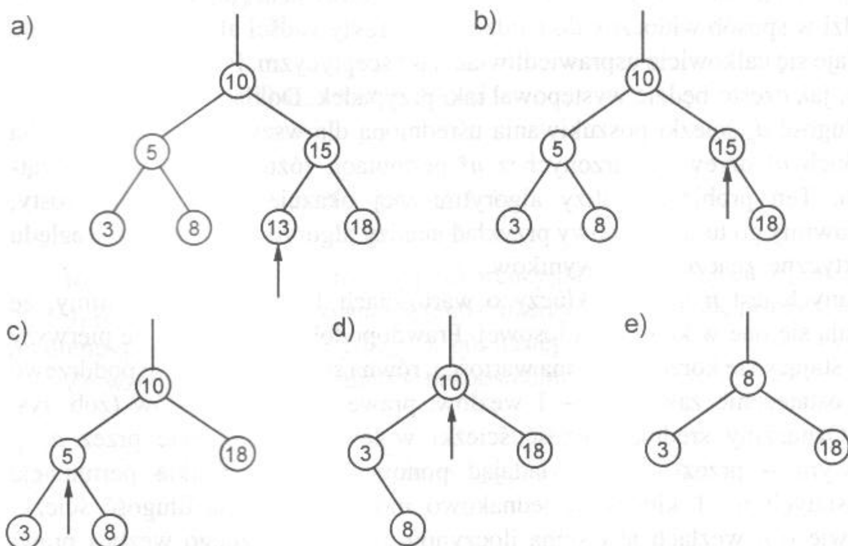
4.4.4. Usuwanie z drzewa

Przejdziemy teraz do problemu odwrotnego względem wstawiania, tj. do usuwania. Zadaniem naszym jest zdefiniowanie algorytmu usuwania węzła o kluczu x z drzewa z uporządkowanymi kluczami. Niestety, usunięcie elementu nie jest na ogół tak proste jak wstawienie. Jest to łatwa operacja wtedy, gdy usuwany element jest węzłem końcowym lub ma tylko jednego potomka. Trudności pojawiają się przy usuwaniu elementu mającego *dwóch* potomków, ponieważ jednym wskaźnikiem nie możemy wskazywać dwóch kierunków. W tej sytuacji usuwany element musi być zastąpiony przez skrajny prawy element lewego poddrzewa lub skrajny lewy węzeł prawego poddrzewa, z których każdy ma co najwyżej jednego potomka. Szczegóły algorytmu są zawarte w rekurencyjnej procedurze *usuń* (4.52). Rozróżnia się w niej trzy przypadki:

(1) Brak składowej o kluczu równym x.
(2) Składowa o kluczu x ma co najwyżej jednego potomka.
(3) Składowa o kluczu x ma dwóch potomków.

```
procedure usuń(x: integer; var p: ref);
   var q: ref;
   procedure us(var r: ref);
   begin if r↑.prawe ≠ nil then us(r↑.prawe) else
      begin q↑.klucz := r↑.klucz; q↑.licznik := r↑.licznik;
         q := r; r := r↑.lewe
      end
   end;
begin {usuń}                                                        (4.52)
   if p = nil then writeln ('BRAK SŁOWA W DRZEWIE') else
   if x<p↑.klucz then usuń(x, p↑.lewe) else
   if x>p↑.klucz then usuń(x, p↑.prawe) else
   begin {usuń p↑} q := p;
      if q↑.prawe = nil then p := q↑.lewe else
      if q↑.lewe = nil then p := q↑.prawe else us(q↑.lewe);
      {dispose(q)}
   end
end {usuń}
```

Pomocnicza procedura rekurencyjna *us* jest wywoływana tylko w trzecim przypadku. „Schodzi" ona wzdłuż skrajnej prawej gałęzi lewego poddrzewa elementu $q\uparrow$, który należy usunąć, a następnie wymienia związaną z $q\uparrow$ informację (klucz i licznik) na odpowiadające wartości skrajnie prawego składnika $r\uparrow$ lewego poddrzewa, po czym $r\uparrow$ może być zwolniony. Nieokreśloną dotąd procedurę *dispose*(q) można uważać za odwrotną lub przeciwną do procedury *new*(q). Ta



RYSUNEK 4.28
Usuwanie z drzewa

druga przydziela pamięć na nowy składnik, podczas gdy poprzednia może być użyta do wskazania systemowi komputerowemu, że pamięć zajęta przez $q\uparrow$ jest ponownie w dyspozycji systemu (rodzaj rotacji pamięci).

Aby zilustrować funkcjonowanie procedury (4.52), odwołamy się do rys. 4.28. Rozważmy drzewo (a); następnie usuwajmy kolejno węzły o kluczach 13, 15, 5, 10. Drzewo powstałe w wyniku tej operacji pokazano na rys. 4.28(b–e).

4.4.5. Analiza przeszukiwania drzewa z wstawianiem

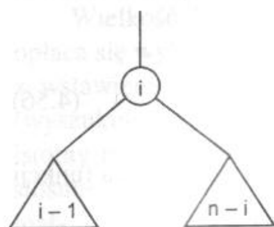
Naturalną – i zdrową – reakcją jest podejrzliwość w stosunku do algorytmu przeszukiwania drzewa z wstawianiem. Należy zachować co najmniej nieco sceptycyzmu do czasu otrzymania bardziej szczegółowej charakterystki zachowania algorytmu. Wielu programistów niepokoi początkowo osobliwy fakt, że na ogół nie wiadomo, jak drzewo będzie rosło ani jaki przyjmie kształt. Możemy tylko zgadywać, że prawdopodobnie nie będzie ono drzewem doskonale zrównoważonym. O ile średnia liczba porównań koniecznych do zlokalizowania klucza w drzewie doskonale zrównoważonym o n węzłach wynosi w przybliżeniu $h = \log n$, to liczba porównań w drzewie powstałym w wyniku działania tego algorytmu będzie większa od h. Ale o ile?

Przede wszystkim łatwo znaleźć przypadek najgorszy. Przyjmijmy, że wszystkie klucze pojawiają się w ściśle rosnącym (lub malejącym) porządku. Wtedy każdy klucz dołączany jest na prawo (lewo) od swego poprzednika i otrzymujemy całkowicie zdegenerowane drzewo, które jest po prostu listą liniową. Średni czas szukania wynosi wtedy $n/2$ porównań. Najgorszy przypadek prowadzi w sposób widoczny do bardziej złej efektywności algorytmu szukania, co wydaje się całkowicie usprawiedliwiać nasz sceptycyzm. Powstaje oczywiście pytanie, jak często będzie występował taki przypadek. Dokładniej, chcielibyśmy znać długość a_n ścieżki poszukiwania uśrednioną dla wszystkich n kluczy i dla wszystkich $n!$ drzew utworzonych z $n!$ permutacji różnych n kluczy początkowych. Ten problem analizy algorytmicznej okazuje się zupełnie prosty; przedstawimy go tu jako typowy przykład analizy algorytmu, a także ze względu na praktyczne znaczenie jej wyników.

Danych jest n różnych kluczy o wartościach $1, 2, \ldots, n$. Przyjmijmy, że pojawiają się one w kolejności losowej. Prawdopodobieństwo tego, że pierwszy klucz – stający się korzeniem – ma wartość i, równa się $1/n$. Jego lewe poddrzewo będzie ostatecznie zawierać $i-1$ węzłów, prawe zaś $n-i$ węzłów (zob. rys. 4.29). Oznaczmy średnią długość ścieżki w lewym poddrzewie przez a_{i-1}, w prawym – przez a_{n-i}, zakładając ponownie, że wszystkie permutacje z pozostałych $n-1$ kluczy są jednakowo możliwe. Średnia długość ścieżki w drzewie o n węzłach jest sumą iloczynów poziomu każdego węzła i prawdopodobieństwa dostępu do niego. Jeżeli przyjmiemy, że wszystkie węzły będą szukane z jednakowym prawdopodobieństwem, to

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i \tag{4.53}$$

gdzie p_i jest długością ścieżki węzła i.



RYSUNEK 4.29
Rozkład wag na gałęziach

W drzewie z rys. 4.29 węzły możemy podzielić na trzy klasy:

(1) $i-1$ węzłów w lewym poddrzewie ma średnią długość ścieżki $a_{i-1}+1$;
(2) korzeń ma długość ścieżki równą 1;
(3) $n-i$ węzłów w prawym poddrzewie ma średnią długość ścieżki $a_{n-i}+1$;

Tak więc wzór (4.53) można wyrazić jako sumę trzech składników:

$$a_n^{(i)} = (a_{i-1}+1)\frac{i-1}{n} + 1\cdot\frac{1}{n} + (a_{n-i}+1)\frac{n-i}{n} \tag{4.54}$$

Szukana wielkość a_n daje się teraz wyprowadzić jako średnia z $a_n^{(i)}$ dla $i = 1, ..., n$, czyli ze wszystkich drzew o kluczu 1, 2, ..., n w korzeniu.

$$a_n = \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n} + \frac{1}{n} + (a_{n-1}+1)\frac{n-i}{n}\right]$$

$$= 1 + \frac{1}{n^2}\sum_{i=1}^{n}[(i-1)a_{i-1} + (n-i)a_{n-i}] \tag{4.55}$$

$$= 1 + \frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\cdot a_i$$

Równanie (4.55) jest formułą rekurencyjną wyznaczającą a_n, o postaci $a_n = f_1(a_1, a_2, \dots, a_{n-1})$. Z równania tego możemy wyprowadzić prostszą formułę rekurencyjną o postaci $a_n = f_2(a_{n-1})$, jak następuje:

Ze wzoru (4.55) otrzymujemy bezpośrednio

(1) $a_n = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i\cdot a_i = 1 + \frac{2}{n^2}(n-1)a_{n-1} + \frac{2}{n^2}\sum_{i=1}^{n-2} i\cdot a_i$

(2) $a_{n-1} = 1 + \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i\cdot a_i$

Mnożąc (2) przez $(n-1/n)^2$, otrzymujemy

$$(3) \quad \frac{2}{n^2}\sum_{i=1}^{n-2} i \cdot a_i = \frac{(n-1)^2}{n^2}(a_{n-1}-1)$$

i podstawiając (3) do (1), dostajemy

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1} + 2n - 1) \tag{4.56}$$

Okazuje się, że a_n można wyrazić w postaci nierekurencyjnej za pomocą funkcji harmonicznej

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

$$a_n = 2\frac{n+1}{n}H_n - 3 \tag{4.57}$$

(Sceptyczny czytelnik powinien sprawdzić, czy wzór (4.57) spełnia zależność rekurencyjną (4.56)).

Z formuły Eulera

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \ldots$$

(gdzie stała Eulera $\gamma \approx 0{,}577$) wyprowadzamy dla dużych n zależność

$$a_n \approx 2\,[\ln(n) + \gamma] - 3 = 2\ln(n) - c$$

Ponieważ średnia długość ścieżki w drzewie doskonale zrównoważonym wynosi w przybliżeniu

$$a'_n = \log(n) - 1 \tag{4.58}$$

więc pomijając stałe wyrażenia znikające dla dużych n, otrzymujemy

$$\lim_{n\to\infty} \frac{a_n}{a'_n} = \frac{2\ln(n)}{\log(n)} = 2 \cdot \ln 2 \approx 1{,}386 \tag{4.59}$$

Co daje nam wynik (4.59) tej analizy? Mówi on nam o tym, że dzięki pracy włożonej w skonstruowanie doskonale zrównoważonego drzewa (zamiast „losowego” drzewa otrzymanego z programu 4.4) można – wciąż zakładając, że szuka się kluczy z jednakowym prawdopodobieństwem – spodziewać się średniego skrócenia ścieżki poszukiwania co najwyżej o 39%. Należy podkreślić tutaj słowo „średnio”, gdyż zysk może oczywiście być niewspółmiernie większy w niekorzystnym przypadku, w którym utworzone drzewo całkowicie zdegenerowało się do listy. Przypadek ten jest jednak bardzo mało prawdopodobny (jeżeli wszystkie

permutacje n wstawianych kluczy są jednakowo prawdopodobne). W związku z tym warto zauważyć, że średnia długość ścieżki w drzewie „losowym" rośnie ściśle logarytmicznie wraz z liczbą jego węzłów, chociaż w najgorszym przypadku długość ścieżki rośnie liniowo.

Wielkość 39% narzuca ograniczenie na wielkość dodatkowej pracy, którą opłaca się wykonać przy jakiejkolwiek reorganizacji struktury drzewa związanej z wstawieniem kluczy. Naturalnie stosunek r między częstością dostępu do (wyszukiwania) węzłów (informacji) a częstością wstawiania wpływa w sposób istotny na opłacalność nakładów w takich przedsięwzięciach. Im większy ten stosunek, tym większa opłacalność reorganizacji. Wielkość 39% jest dostatecznie mała, aby w większości zastosowań usprawnienia algorytmu prostego wstawiania w drzewo nie opłacały się – chyba że liczba węzłów *oraz* stosunek liczby dostępów do liczby wstawień są duże (lub obawiamy się najgorszego przypadku).

4.4.6. Drzewa zrównoważone

Z poprzednich rozważań wynika, że procedura wstawiania, która zawsze przywraca doskonale zrównoważoną strukturę drzewa, rzadko kiedy jest opłacalna, ponieważ zrównoważenie drzewa po losowym wstawianiu jest operacją dość skomplikowaną. Możliwość usprawnień leży w sformułowaniu mniej ścisłej definicji „zrównoważenia". Kryteria takiego „niedokładnego" zrównoważenia powinny prowadzić do uproszczenia reorganizacji drzewa kosztem tylko niewielkiego pogorszenia średniej efektywności szukania. Jedną z takich definicji zrównoważenia zaproponowali Adelson-Velskii i Landis [4.1]. Kryterium zrównoważenia jest następujące:

> Drzewo jest **zrównoważone** wtedy i tylko wtedy, gdy dla każdego węzła wysokości dwóch jego poddrzew różnią się co najwyżej o 1.

Drzewa spełniające ten warunek są często nazywane **drzewami AVL** (od nazwisk wynalazców). Będziemy je po prostu nazywać **drzewami zrównoważonymi**, gdyż wydaje się, że to kryterium zrównoważenia jest najodpowiedniejsze. (Zauważmy, że wszystkie drzewa doskonale zrównoważone są także drzewami AVL-zrównoważonymi).

Definicja ta jest nie tylko prosta, ale także prowadzi do łatwej w obsłudze procedury równoważącej drzewo, a średnia długość ścieżki poszukiwania jest praktycznie identyczna z odpowiednią długością w drzewie doskonale zrównoważonym.

Nawet w najgorszym przypadku na drzewie zrównoważonym można wykonywać w $O(\log n)$ jednostkach czasu następujące operacje:

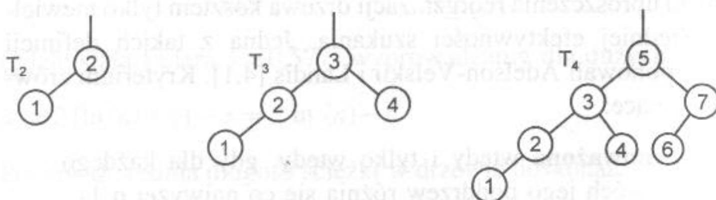
(1) lokalizację węzła o danym kluczu;
(2) wstawienie węzła o danym kluczu;
(3) usunięcie węzła o danym kluczu.

Stwierdzenia te są bezpośrednią konsekwencją twierdzenia udowodnionego przez Adelsona-Velskiiego i Landisa, które gwarantuje, że niezależnie od liczby węzłów drzewo zrównoważone nie będzie nigdy wyższe o więcej niż 45% od swego doskonale zrównoważonego odpowiednika. Jeżeli oznaczymy wysokość zrównoważonego drzewa o n węzłach przez $h_z(n)$, to

$$\log(n+1) \leqslant h_z(n) \leqslant 1{,}4404 \cdot \log(n+2) - 0{,}328 \tag{4.60}$$

Oczywiście optimum jest osiągane wtedy, gdy drzwo jest doskonale zrównoważone dla $n = 2^k - 1$. Ale jaka jest struktura najgorszego AVL-zrównoważonego drzewa?

Aby znaleźć maksymalną wysokość h drzewa zrównoważonego o n węzłach, przyjmijmy ustalone h i spróbujmy skonstruować drzewo zrównoważone o minimalnej liczbie węzłów. Zalecamy tę strategię, gdyż tak jak w przypadku minimalnego h, wartość tę można osiągnąć tylko dla pewnych, szczególnych wartości n. Oznaczmy drzewo o wysokości h przez T_h. Oczywiście więc T_0 jest drzewem pustym, a T_1 drzewem o jednym węźle. Aby skonstruować drzewo T_h dla $h > 1$, łączymy korzeń z dwoma poddrzewami znowu o minimalnej liczbie węzłów. Tak więc poddrzewa są również typu T. Łatwo zauważyć, że jedno poddrzewo *musi* mieć wysokość $h-1$, a drugie może mieć wysokość o jeden mniejszą, tj. $h-2$. Na rysunku 4.30 pokazano drzewa o wysokościach 2, 3 i 4.



RYSUNEK 4.30
Drzewa Fibonacciego o wysokościach 2, 3 i 4

Ponieważ zasada budowy tych drzew bardzo przypomina zasadę tworzenia liczb Fibonacciego, zwane są one **drzewami Fibonacciego**. Definiuje się je następująco:

(1) Drzewo puste jest drzewem Fibonacciego o wysokości 0.
(2) Pojedynczy węzeł jest drzewem Fibonacciego o wysokości 1.
(3) Jeżeli T_{h-1} i T_{h-2} są drzewami Fibonacciego o wysokościach $h-1$ oraz $h-2$, to $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$ jest drzewem Fibonacciego o wysokości h.
(4) Tylko takie drzewa są drzewami Fibonacciego.

Liczba węzłów drzewa T_h daje się określić za pomocą prostej zależności rekurencyjnej:

$$N_0 = 0, \quad N_1 = 1, \quad N_h = N_{h-1} + 1 + N_{h-2} \tag{4.61}$$

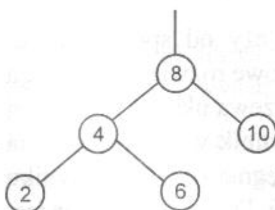
N_i są tymi liczbami węzłów, dla których można uzyskać najgorszy przypadek (górna granica h) warunku (4.60).

4.4.7. Wstawianie w drzewach zrównoważonych

Zastanówmy się, co może się dziać przy wstawianiu nowego węzła do drzewa zrównoważonego. Dany jest korzeń k oraz lewe i prawe poddrzewa L i P. Należy rozróżnić trzy przypadki (zakładamy, że nowy węzeł jest wstawiany do lewego poddrzewa, zwiększając jego wysokość o 1):

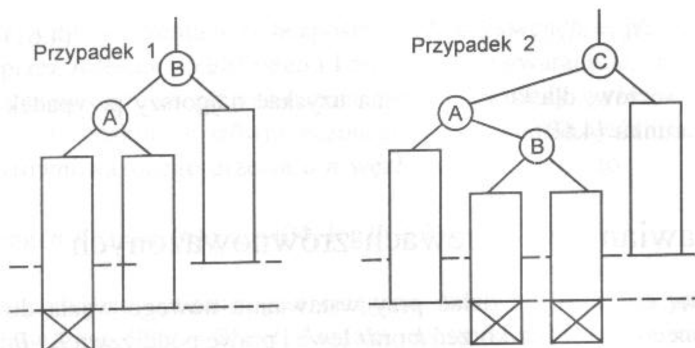
(1) $h_L = h_P$ L i P stają się drzewami o różnej wysokości, ale kryterium zrównoważenia jest w dalszym ciągu spełnione;
(2) $h_L < h_P$ L i P uzyskują jednakową wysokość, zrównoważenie drzewa nawet się poprawia;
(3) $h_L > h_P$ kryterium zrównoważenia nie jest spełnione i drzewo musi być przebudowane.

Rozważmy drzewo z rys. 4.31. Węzły o kluczach 9 lub 11 mogą być wstawione bez ponownego równoważenia. Drzewo o korzeniu 10 stanie się jednostronne (przyp. 1), a drzewo o korzeniu 8 poprawi swoje zrównoważenie (przyp. 2). Wstawianie węzłów 1, 3, 5 lub 7 pociąga za sobą jednak konieczność ponownego równoważenia drzewa.

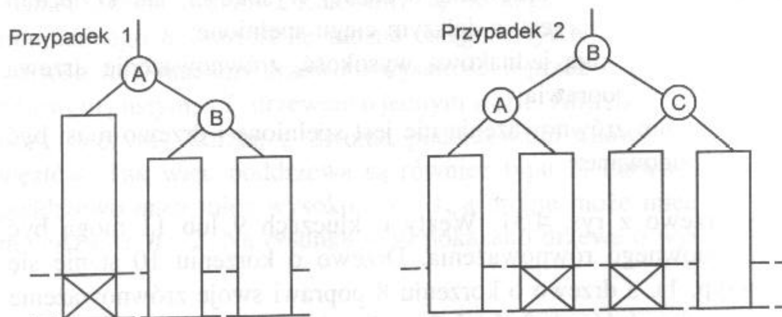


RYSUNEK 4.31
Drzewo zrównoważone

Nieco dokładniejsze przeanalizowanie sytuacji wykazuje, że istnieją tylko dwa istotnie różne układy wymagające odrębnego traktowania. Pozostałe można z nich wyprowadzić przez symetrię. Przypadek 1 jest scharakteryzowany przez wstawianie kluczy 1 lub 3 do drzewa z rys. 4.31, a przypadek 2 – przez wstawienie węzłów 5 lub 7. Te dwa przypadki są uogólnione na rys. 4.32, na którym prostokąty oznaczają poddrzewa, a przyrost wysokości powstały przy wstawianiu węzłów jest oznaczony krzyżykiem. Pożądane zrównoważenie tych dwóch struktur przywracają proste transformacje. Ich wynik pokazano na rys. 4.33; zauważmy, że dopuszczalne są tylko przesunięcia w pionie, natomiast nie może zmienić się ustawienie pokazanych węzłów i poddrzew względem siebie w poziomie.



RYSUNEK 4.32
Brak zrównoważenia spowodowany wstawianiem



RYSUNEK 4.33
Przywrócenie zrównoważenia

Algorytm wstawiania i równoważenia istotnie zależy od sposobu przechowywania informacji o zrównoważeniu drzewa. Krańcowe rozwiązanie polega na trzymaniu wszystkich informacji o zrównoważeniu drzewa ukrytych w samej strukturze drzewa. W tym przypadku jednakże współczynnik wyważenia węzła musi być wyznaczany za każdym razem od nowa, jeżeli ulegnie zmianie w wyniku wstawiania, co prowadzi do znacznie gorszej efektywności. Drugą skrajnością jest przypisanie współczynnika wyważenia każdemu węzłowi i pamiętanie go wraz z węzłem. Definicję (4.48) typu węzła rozszerzamy wtedy następująco:

```
type węzeł = record klucz: integer;
                    licznik: integer;
                    lewe, prawe: ref;                    (4.62)
                    wyważ: −1..+1
             end
```

W dalszych rozważaniach za współczynnik wyważenia węzła będziemy uważać różnicę wysokości prawego i lewego poddrzewa i oprzemy wynikający z tego algorytm na typie węzła (4.62).

Proces wstawiania węzła składa się w zasadzie z następujących trzech kolejnych etapów:

(1) Idź po ścieżce przeszukiwania drzewa, aż stwierdzisz, że klucza nie ma w drzewie.
(2) Wstaw nowy węzeł i wyznacz wynikający z tego współczynnik wyważenia.
(3) Cofaj się po ścieżce przeszukiwania i sprawdzaj w każdym węźle współczynnik wyważenia.

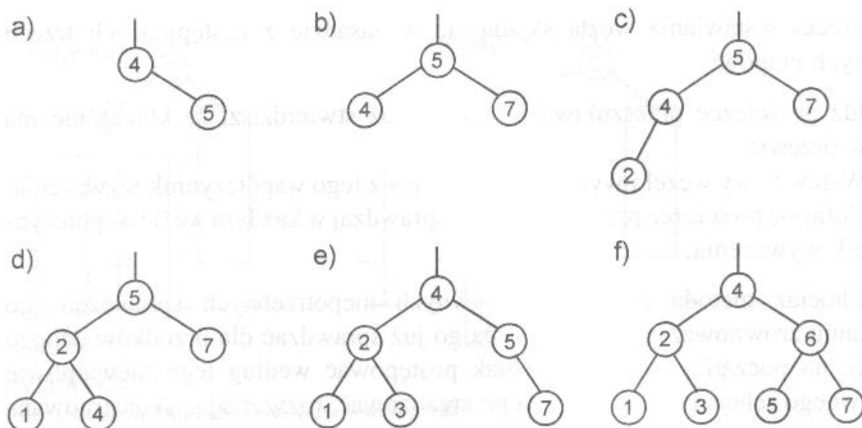
Chociaż metoda ta wymaga pewnych niepotrzebnych sprawdzeń (po uzyskaniu zrównoważenia nie potrzeba go już sprawdzać dla przodków danego węzła), na początku będziemy jednak postępować według tego niewątpliwie poprawnego schematu, gdyż można go zrealizować, rozszerzając skonstruowaną wcześniej procedurę przeszukiwania z wstawianiem z programu 4.4. Procedura ta opisuje operację szukania potrzebną w każdym pojedynczym węźle; dzięki jej rekurencyjnej postaci można w niej łatwo umieścić dodatkową operację „w drodze powrotnej po ścieżce przeszukiwania". W każdym kroku musi być przekazywana informacja, czy zwiększyła się wysokość poddrzewa, w którym dokonano wstawienia. Rozszerzymy więc listę parametrów procedury o zmienną boolowską h, o znaczeniu: „*wysokość poddrzewa zwiększyła się*". Oczywiście h musi być parametrem przekazywanym przez zmienną, gdyż używa się go do przenoszenia wyniku.

Załóżmy teraz, że proces wraca do węzła $p\uparrow$ z lewej gałęzi (zob. rys. 4.32) z informacją, że zwiększyła się jej wysokość. Musimy teraz rozróżnić trzy sytuacje, w zależności od wysokości poddrzew przed wstawieniem:

(1)	$h_L < h_P$, $p\uparrow.wywa\dot{z} = +1$	uprzednie niezrównoważenie w p zostało usunięte;
(2)	$h_L = h_P$, $p\uparrow.wywa\dot{z} = 0$	ciężar przesunięty jest teraz w lewo;
(3)	$h_L > h_P$, $p\uparrow.wywa\dot{z} = -1$	konieczne jest ponowne równoważenie.

W trzecim przypadku zbadanie współczynnika wyważenia korzenia lewego poddrzewa (powiedzmy, $p1\uparrow.wywa\dot{z}$) pozwala określić, czy wystąpił przypadek 1 czy też przypadek 2 z rys. 4.32. Jeżeli ten węzeł ma także wyższe lewe poddrzewo, to mamy do czynienia z przypadkiem 1; jeżeli wyższe jest prawe poddrzewo, to z przypadkiem 2. (Przekonaj się, że w tym przypadku nie może wystąpić lewe poddrzewo ze współczynnikiem wyważenia równym 0 w swoim korzeniu). Potrzebne operacje równoważenia są całkowicie wyrażone przez zmianę wartości wskaźników. W rzeczywistości wskaźniki są zamieniane cyklicznie, przy czym wykonuje się jedną lub dwie zamiany (rotacje), w których biorą udział dwa lub trzy węzły. Oprócz zamiany wskaźników muszą być uaktualnione współczynniki wyważenia odpowiednich węzłów. Szczegóły można znaleźć w procedurze (4.63) szukania ze wstawianiem i równoważeniem (wyważaniem).

Zasadę działania pokazano na rys. 4.34. Rozważmy drzewo binarne (a) składające się tylko z dwóch węzłów. Wstawienie najpierw klucza 7 daje



RYSUNEK 4.34
Wstawianie węzłów w drzewie zrównoważonym

w wyniku drzewo niezrównoważone (tj. listę liniową). Zrównoważenie wymaga pojedynczej zamiany *PP*, dającej drzewo (b). Dalsze wstawienie węzłów 2 oraz 1 daje niezrównoważone poddrzewo o korzeniu 4. Poddrzewo to równoważy się przez pojedynczą zamianę *LL* (d). Wstawienie teraz klucza 3 spowoduje, że kryterium zrównoważenia nie będzie spełnione w węźle 5. Aby ponownie zrównoważyć drzewo, należy wykonać bardziej skomplikowaną podwójną zamianę *LP*; w wyniku otrzymujemy drzewo (e). Przy następnym wstawianiu jedynym kandydatem do utraty zrównoważenia jest węzeł 5. I rzeczywiście, po wstawieniu węzła 6 trzeba odwołać się do czwartego przypadku niezrównoważenia przedstawionego w (4.63) – wykonać podwójną zamianę *PL*. Drzewo końcowe widzimy na rys. 4.34 (f).

```
procedure szukaj(x: integer; var p: ref; var h: boolean);
  var p1, p2: ref; {h=false}
begin
  if p=nil then
  begin {brak klucza w drzewie; wstaw go}
     new(p); h:= true;
     with p↑ do                                                    (4.63)
     begin klucz:= x; licznik:= 1; lewe:= nil; prawe:= nil; wyważ:= 0
     end
  end else
  if x<p↑.klucz then
  begin szukaj (x, p↑.lewe, h);
     if h then {urosła lewa gałąź}
     case p↑.wyważ of
     1: begin p↑.wyważ:= 0; h:= false end;
     0: p↑.wyważ:= −1;
```

```
  −1: begin {wyważ drzewo} p1 := p↑.lewe;
          if p1↑.wyważ = −1 then
          begin {pojedyncza zamiana LL}
              p↑.lewe := p1↑.prawe; p1↑.prawe := p;
              p↑.wyważ := 0; p := p1
          end else
          begin {podwójna zamiana LP} p2 := p1↑.prawe;
              p↑.prawe := p2↑.lewe; p2↑.lewe := p1;
              p↑.lewe := p2↑.prawe; p2↑.prawe := p;
              if p2↑.wyważ = −1 then p↑.wyważ := +1 else p↑.wyważ := 0;
              if p2↑.wyważ = +1 then p1↑.wyważ := −1 else p1↑.wyważ := 0;
              p := p2
          end;
          p↑.wyważ := 0; h := false
      end
    end
end else
if x > p↑.klucz then                                                  (4.63)
begin szukaj (x, p↑.prawe, h);
    if h then {urosła prawa gałąź}
    case p↑.wyważ of
  −1: begin p↑.wyważ := 0; h := false end;
   0: p↑.wyważ := +1;
   1: begin {wyważ drzewo} p1 := p↑.prawe;
          if p1↑.wyważ = +1 then
          begin {pojedyncza zamiana PP}
              p↑.prawe := p1↑.lewe; p1↑.lewe := p;
              p↑.wyważ := 0; p := p1
          end else
          begin {podwójna zamiana PL} p2 := p1↑.lewe;
              p1↑.lewe := p2↑.prawe; p2↑.prawe := p1;
              p↑.prawe := p2↑.lewe; p2↑.lewe := p;
              if p2↑.wyważ = +1 then p↑.wyważ := −1 else p↑.wyważ := 0;
              if p2↑.wyważ = −1 then p1↑.wyważ := +1 else
              p1↑.wyważ := 0; p := p2
          end;
          p↑.wyważ := 0; h := false
      end
    end
end
else
begin p↑.licznik := p↑.licznik + 1; h := false end
end {szukaj}
```

A oto dwa szczególnie intersujące pytania dotyczące efektywności algorytmu wstawiania do drzewa zrównoważonego:

(1) Jaka jest spodziewana wysokość konstruowanego drzewa zrównoważonego, jeśli każda z $n!$ permutacji n kluczy występuje z jednakowym prawdopodobieństwem?
(2) Jakie jest prawdopodobieństwo konieczności zrównoważenia drzewa po wstawieniu węzła?

Matematyczna analiza tego skomplikowanego algorytmu jest ciągle problemem otwartym. Empiryczne testy potwierdzają przypuszczenie, że oczekiwana wysokość drzewa zrównoważonego wygenerowanego przez procedurę (4.63) wynosi $h = \log(n) + c$, gdzie c jest niewielką stałą ($c \approx 0{,}25$). Oznacza to, że w praktyce drzewo AVL-zrównoważone zachowuje się równie dobrze jak drzewo doskonale zrównoważone, pomimo że jest znacznie łatwiejsze w obsłudze. Dane empiryczne mówią również, że średnio na dwa wstawienia jest konieczne jedno zrównoważenie drzewa. Pojedyncze i podwójne zamiany są jednakowo prawdopodobne. Oczywiście przykład z rys. 4.34 został celowo tak dobrany, aby zademonstrować możliwie największą liczbę zamian przy minimalnej liczbie wstawień!

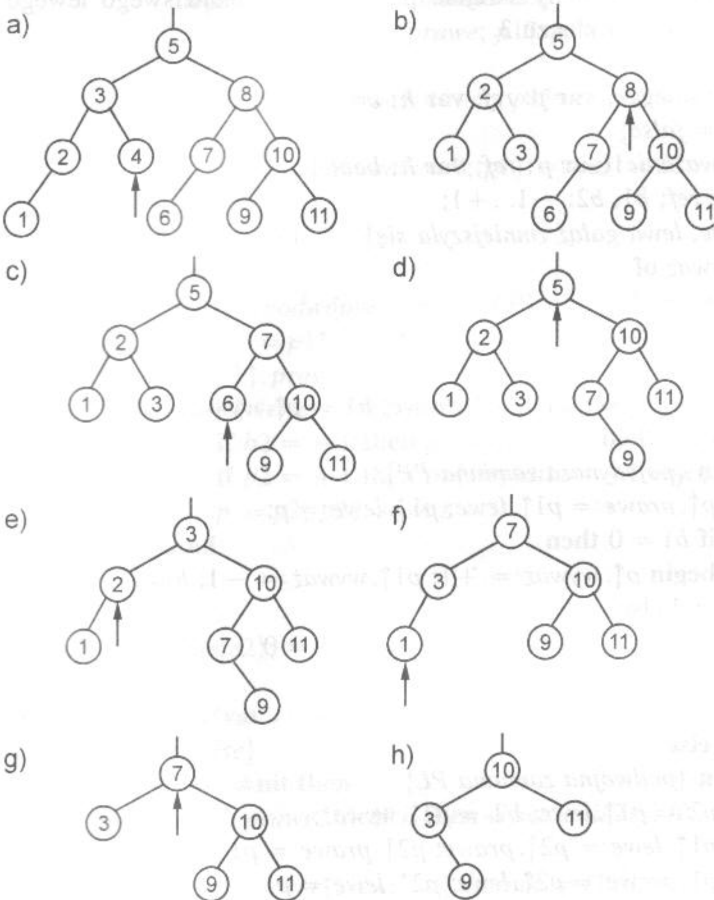
Stopień komplikacji operacji równoważenia (wyważania) sugeruje, że drzew zrównoważonych należy używać wtedy, gdy pobieranie informacji jest znacznie częstsze od wstawiania. Prawdziwe jest to zwłaszcza dlatego, że w celu ekonomicznego wykorzystania pamięci węzły takich drzew są zazwyczaj przechowywane w postaci gęsto upakowanych rekordów. Szybkość dostępu i aktualizacji współczynników wyważenia – wymagających tylko dwóch bitów pamięci – jest często decydującym czynnikiem wpływającym na efektywność operacji równoważenia (wyważania). Obliczenia doświadczalne wykazują, że drzewa zrównoważone tracą wiele ze swego uroku, jeżeli stosuje się gęste upakowanie rekordów. Tak naprawdę trudno wygrać z łatwym i prostym algorytmem wstawiania do drzewa!

4.4.8. Usuwanie węzłów z drzew zrównoważonych

Nasze doświadczenia z usuwaniem węzłów z drzew sugerują, że w przypadku drzew zrównoważonych usuwanie będzie także bardziej skomplikowane niż wstawianie. Jest to rzeczywiście prawdą, pomimo że operacja wyważania (równoważenia) jest w zasadzie taka sama jak przy wstawianiu. W szczególności wyważanie polega na pojedynczej lub podwójnej zamianie (rotacji) węzłów.

Usuwanie węzła z drzewa zrównoważonego przebiega zgodnie z algorytmem (4.52). Najprostsze przypadki to węzły końcowe i węzły o jednym potomku. Jeżeli usuwany węzeł ma dwa poddrzewa, to ponownie zastąpimy go skrajnie prawym węzłem lewego poddrzewa. Podobnie jak przy wstawianiu węzłów

(4.63), dodamy zmienną boolowską *h* oznaczającą, że „wysokość poddrzewa zmniejszyła się". Zrównoważenie może być potrzebne tylko wtedy, gdy *h* = *true*. Zmiennej *h* przypisujemy wartość *true* po znalezieniu i usunięciu węzła lub jeżeli wyważanie samo redukuje wysokość poddrzewa.



RYSUNEK 4.35
Usuwanie węzłów z drzewa zrównoważonego

W (4.64) przedstawimy dwie (symetryczne) operacje wyważania w postaci procedur, ponieważ są one wywoływane z więcej niż jednego miejsca algorytmu usuwania. Zauważmy, że *wyważanie*1 jest stosowane wtedy, gdy zmniejszyła się wysokość lewej gałęzi, natomiast *wyważanie*2 – gdy prawej.

Działanie procedury pokazano na rys. 4.35. W drzewie zrównoważonym (a) są usuwane kolejno węzły o kluczach 4, 8, 6, 5, 2, 1 i 7; wyniki usuwania to drzewa (b)...(h).

Usunięcie klucza 4 jest samo w sobie proste, gdyż jest to węzeł końcowy. Jednakże powoduje to niezrównoważenie węzła 3. Jego zrównoważenie wymaga pojedynczej zmiany *LL*. Wyważanie okazuje się konieczne także po usunięciu

węzła 6. Tym razem wyważane jest prawe poddrzewo korzenia (7) przez pojedynczą zamianę *PP*. Usunięcie węzła 2, choć samo w sobie proste, ponieważ ma on tylko jednego potomka, wymaga skomplikowanej podwójnej zamiany *PL*. Czwarty przypadek, podwójna zamiana *LP* jest wywoływana po usunięciu węzła 7, który jest najpierw zastąpiony skrajnie prawym elementem swego lewego poddrzewa, czyli węzłem o kluczu 3.

```
procedure usuń(x: integer; var p: ref; var h: boolean);
   var q: ref; {h = false}
   procedure wyważanie1(var p: ref; var h: boolean);
      var p1, p2: ref; b1, b2: -1..+1;
   begin {h = true, lewa gałąź zmniejszyła się}
      case p↑.wyważ of
    -1: p↑.wyważ := 0;
     0: begin p↑.wyważ := +1; h := false
        end;
     1: begin {wyważanie} p1 := p↑.prawe; b1 := p↑.wyważ;
           if b1 ≥ 0 then
           begin {pojedyncza zamiana PP}
              p↑.prawe := p1↑.lewe; p1↑.lewe := p;
              if b1 = 0 then
              begin p↑.wyważ := +1; p1↑.wyważ := -1; h := false
              end else
              begin p↑.wyważ := 0; p1↑.wyważ := 0
              end;
              p := p1                                              (4.64)
           end else
           begin {podwójna zamiana PL}
              p2 := p1↑.lewe; b2 := p2↑.wyważ;
              p1↑.lewe := p2↑.prawe; p2↑.prawe := p1;
              p↑.prawe := p2↑.lewe; p2↑.lewe := p;
              if b2 = +1 then p↑.wyważ := -1 else p↑.wyważ := 0;
              if b2 = -1 then p1↑.wyważ := +1 else p1↑.wyważ := 0;
              p := p2; p2↑.wyważ := 0
           end
        end
      end
   end {wyważanie1};

   procedure wyważanie2 (var p: ref; var h: boolean);
      var p1, p2: ref; b1, b2: -1..+1;
   begin {h = true, prawa gałąź zmniejszyła się}
      case p↑.wyważ of
     1: p↑.wyważ := 0;
```

```
    0:  begin p↑.wyważ := −1; h := false
        end;
   −1:  begin {wyważanie} p1 := p↑.lewe; b1 := p1↑.wyważ;
          if b1 ≤ 0 then
          begin {pojedyncza zamiana LL}
            p↑.lewe := p1↑.prawe; p1↑.prawe := p;
            if b1 = 0 then
            begin p↑.wyważ := −1; p1↑.wyważ := +1; h := false
            end else
            begin p↑.wyważ := 0; p1↑.wyważ := 0
            end;
            p := p1
          end else
          begin {podwójna zamiana LP}
            p2 := p1↑.prawe; b2 := p2↑.wyważ;
            p1↑.prawe := p2↑.lewe; p2↑.lewe := p1;
            p↑.lewe := p2↑.prawe; p2↑.prawe := p;
            if b2 = −1 then p↑.wyważ := +1 else p↑.wyważ := 0;
            if b2 = +1 then p1↑.wyważ := −1 else p1↑.wyważ := 0;
            p := p2; p2↑.wyważ := 0
          end
        end                                                      (4.64)
      end
    end {wyważanie2};

    procedure us(var r: ref; var h: boolean);
    begin {h = false}
      if r↑.prawe ≠ nil then
      begin us(r↑.prawe, h); if h then wyważanie2(r, h)
      end else
      begin q↑.klucz := r↑.klucz; q↑.licznik := r↑.licznik;
        r := r↑.lewe; h := true
      end
    end;
begin {usuń}
    if p = nil then
      begin writeln('BRAK KLUCZA W DRZEWIE'); h := false
      end else
    if x < p↑.klucz then
      begin usuń(x, p↑.lewe, h); if h then wyważanie1(p, h)
      end else
    if x > p↑.klucz then
      begin usuń(x, p↑.prawe, h); if h then wyważanie2(p, h)
      end else
```

```
  begin {usuń p↑} q := p;
    if q↑.prawe = nil then
      begin p := q↑.lewe; h := true
      end else
    if q↑.lewe = nil then
      begin p := q↑.prawe; h := true
      end else                                          (4.64)
    begin us(q↑.lewe, h);
      if h then wyważanie1(p, h)
    end;
    {dispose(q)}
  end
end {usuń}
```

Oczywiście do usunięcia elementu z drzewa zrównoważonego może być także potrzebne – w najgorszym przypadku – wykonanie $O(\log n)$ operacji. Jednakże nie wolno przeoczyć istotnej różnicy między zachowaniem się procedur wstawiania i usuwania. Podczas gdy wstawianie pojedynczego klucza może wymagać co najwyżej jednej zamiany (dwóch lub trzech węzłów), usuwanie może wymagać zamiany w *każdym* węźle na ścieżce poszukiwania. Rozważmy np. usunięcie skrajnie prawego węzła drzewa Fibonacciego. W tym przypadku usunięcie dowolnego węzła prowadzi do zmniejszenia wysokości drzewa; ponadto usunięcie skrajnie prawego węzła wymaga maksymalnej liczby zamian. Dlatego jest to najgorszy wybór węzła w najgorszym przypadku drzewa zrównoważonego – raczej nieszczęśliwy zbieg okoliczności! Jakie jest jednak w ogóle prawdopodobieństwo wystąpienia zamiany? Wyniki doświadczalne są zadziwiające. Podczas gdy jedna zamiana jest wywoływana na około dwa wstawiania, to tylko jedna jest konieczna na każde pięć usunięć węzłów. Z tego też względu usuwanie z drzew zrównoważonych jest prawie tak łatwe – lub tak skomplikowane – jak wstawianie.

4.4.9. Optymalne drzewa poszukiwań

Nasze dotychczasowe rozważania nad organizowaniem drzew poszukiwań oparte były na założeniu, że częstość dostępu jest jednakowa dla wszystkich węzłów, czyli wszystkie klucze są jednakowo prawdopodobne jako argumenty wyszukiwania. Jest to prawdopodobnie najlepsze założenie wtedy, gdy brak danych o rozkładzie dostępu do węzłów. *Zdarzają się* jednak przypadki (stanowią one raczej wyjątki niż regułę), w których znane jest prawdopodobieństwo dostępu do pojedynczych kluczy. Cechą charakterystyczną takich przypadków jest zazwyczaj to, że klucze są ustalone, tj. drzewo nie podlega ani wstawianiu, ani usuwaniu i zachowuje stałą strukturę. Typowym przykładem jest analizator leksykalny translatora, w którym dla każdego słowa (identyfikatora) bada się, czy

jest ono słowem kluczowym (zastrzeżonym). Statystyczne pomiary dokonane na setkach tłumaczonych programów dają dokładne dane o względnej częstości wystąpień, a tym samym o częstości dostępu do poszczególnych kluczy.

Przyjmijmy, że prawdopodobieństwo dostępu do i-tego węzła wynosi p_i.

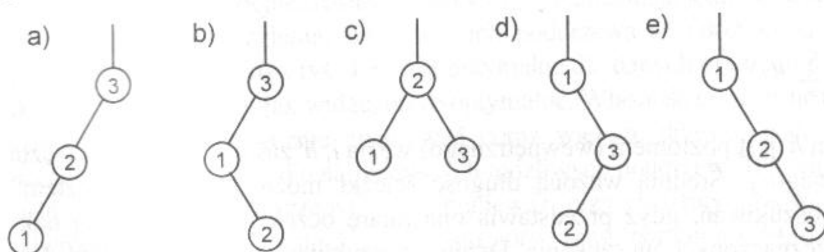
$$Pr\ \{x=k_i\} = p_i \qquad \sum_{i=1}^{n} p_i = 1 \tag{4.65}$$

Chcielibyśmy teraz zorganizować drzewo tak, aby łączna liczba kroków poszukiwania – obliczona dla dostatecznie wielu prób – była najmniejsza. Z tego powodu zmodyfikujemy definicję długości ścieżki (4.34), dołączając do każdego węzła pewną wagę. Węzły o częstym dostępie staną się „ciężkimi" węzłami, a węzły rzadko odwiedzane będą „lekkimi" węzłami. **Ważona długość ścieżki** (wewnętrznej) **poszukiwania** jest więc sumą wszystkich ścieżek z korzenia do każdego węzła z uwzględnieniem wag określających prawdopodobieństwo dostępu do węzłów.

$$P_I = \sum_{i=1}^{n} p_i\ h_i \tag{4.66}$$

h_i jest poziomem węzła i (lub jego odległością od korzenia $+1$). Celem naszym jest *zminimalizowanie ważonej długości ścieżki poszukiwań* dla danego rozkładu prawdopodobieństwa.

Jako przykład rozważmy zbiór kluczy 1, 2, 3 o prawdopodobieństwach dostępu $p_1 = 1/7$, $p_2 = 2/7$ i $p_3 = 4/7$. Z kluczy tych można zbudować drzewo poszukiwań na pięć różnych sposobów (zob. rys. 4.36).



RYSUNEK 4.36
Drzewa poszukiwań o trzech węzłach

Ważone długości ścieżek wyznaczono zgodnie z (4.66).

$$P_I^{(a)} = \frac{1}{7}(1 \cdot 3 + 2 \cdot 2 + 4 \cdot 1) = \frac{11}{7}$$

$$P_I^{(b)} = \frac{1}{7}(1 \cdot 2 + 2 \cdot 3 + 4 \cdot 1) = \frac{12}{7}$$

$$P_I^{(c)} = \frac{1}{7}(1 \cdot 2 + 2 \cdot 1 + 4 \cdot 2) = \frac{12}{7}$$

$$P_I^{(d)} = \frac{1}{7}(1 \cdot 1 + 2 \cdot 3 + 4 \cdot 2) = \frac{15}{7}$$

$$P_I^{(e)} = \frac{1}{7}(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3) = \frac{17}{7}$$

Tak więc w tym przykładzie drzewem optymalnym okazuje się drzewo zdegenerowane (a), a nie drzewo doskonale zrównoważone.

Przykład analizatora leksykalnego translatora sugeruje natychmiast, że problem ten powinno się analizować przy nieco ogólniejszym warunku: słowa występujące w tekście źródłowym nie zawsze są słowami kluczowymi; tak naprawdę jest to raczej sytuacja wyjątkowa. Stwierdzenie faktu, że dane słowo *k nie jest* kluczem w drzewie poszukiwań, można interpretować jako dostęp do hipotetycznego „specjalnego węzła" wstawianego między następnym niższym i następnym wyższym kluczem (zob. rys. 4.19) z przyporządkowaną mu długością ścieżki zewnętrznej. Jeżeli znane są także prawdopodobieństwa q_i wystąpienia argumentu szukania x między dwoma kluczami k_i oraz k_{i+1}, to informacje te mogą znacznie zmienić strukturę drzewa optymalnego. Uogólnimy więc problem, rozważając również poszukiwania niepomyślne.

Całkowita średnia ważona długość ścieżki wynosi teraz

$$P = \sum_{i=1}^{n} p_i \, h_i + \sum_{j=0}^{n} q_j \, h'_j \tag{4.67}$$

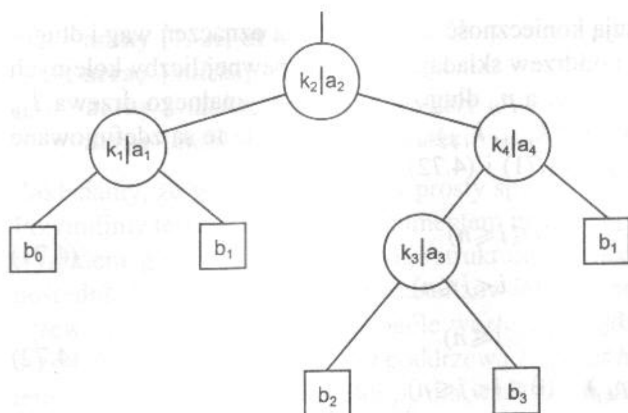
gdzie

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$$

przy czym h_i jest poziomem (wewnętrznego) węzła i, h' zaś jest poziomem węzła zewnętrznego j. Średnią ważoną długość ścieżki można nazwać „kosztem" drzewa poszukiwań, gdyż przedstawia ona miarę oczekiwanej wielkości nakładów przeznaczonych na szukanie. Drzewo poszukiwań, którego struktura daje najmniejszy koszt spośród drzew o danym zbiorze kluczy k_i i prawdopodobieństwach p_i i q_j, jest nazywane **drzewem optymalnym**.

Do znalezienia drzewa optymalnego nie jest konieczne, aby suma współczynników p i q wynosiła 1. W rzeczywistości prawdopodobieństwa te są zwykle wyznaczane w eksperymentach, w których zliczane są dostępy do węzłów. Zamiast używać prawdopodobieństw p_i i q_j, będziemy dalej używać liczników częstości i oznaczymy je przez

a_i = ile razy argument poszukiwania x równy był k_i
b_j = ile razy argument poszukiwania x znajdował się między k_j i k_{j+1}



RYSUNEK 4.37
Drzewo poszukiwań wraz z częstościami dostępów

Dodatkowo b_0 oznacza, ile razy x było mniejsze od k_1, b_n zaś – ile razy x było większe od k_n (zob. rys. 4.37). Będziemy dalej używać P do oznaczenia skumulowanej ważonej długości ścieżki zamiast średniej długości ścieżki:

$$P = \sum_{i=1}^{n} a_i h_i + \sum_{j=0}^{n} b_j h_j' \tag{4.68}$$

Tak więc oprócz uniknięcia obliczania prawdopodobieństw na podstawie zmierzonych częstości uzyskaliśmy dalszą korzyść w możliwości stosowania wyłącznie liczb całkowitych w naszych poszukiwaniach drzewa optymalnego.

Uwzględniając fakt, że liczba możliwych konfiguracji z n węzłów rośnie wykładniczo wraz ze wzrostem n, zadanie znalezienia optimum dla dużych n wydaje się beznadziejne. Jednakże drzewa optymalne mają jedną ważną własność ułatwiającą ich znalezienie: wszystkie ich poddrzewa są także optymalne. Na przykład, jeżeli drzewo z rys. 4.37 jest optymalne dla danych a_i i b_j, to poddrzewo o kluczach k_3 i k_4 jest, jak widać, także optymalne. Własność ta sugeruje algorytm, który może systematycznie znajdować coraz większe drzewa, poczynając od pojedynczych węzłów jako najmniejszych możliwych poddrzew. Tak więc drzewo rośnie „od liści do korzenia", co (ponieważ przywykliśmy rysować drzewa odwrócone) oznacza kierunek „z dołu do góry" (ang. *bottom-up*) [4.6].

Równanie (4.69) jest kluczem do tego algorytmu. Niech P będzie ważoną długością ścieżki drzewa oraz niech P_L i P_P będą ważonymi długościami ścieżek dla lewego i prawego poddrzewa. P jest więc sumą P_L *i* P_P oraz liczby przejść przez korzeń podczas poszukiwań, która jest po prostu łączną liczbą prób poszukiwań W.

$$P = P_L + W + P_P \tag{4.69}$$

$$W = \sum_{i=1}^{n} a_i + \sum_{j=0}^{n} b_j \tag{4.70}$$

W nazwiemy **wagą** drzewa. *Średnia* długość ścieżki jest więc równa P/W.

Rozważania te wskazują konieczność wprowadzenia oznaczeń wag i długości ścieżek dla dowolnych poddrzew składających się z pewnej liczby kolejnych kluczy. Niech w_{ij} oznacza wagę, a p_{ij} długość ścieżki optymalnego drzewa T_{ij}, zawierającego węzły o kluczach $k_{i+1}, k_{i+2}, \dots, k_j$. Wielkości te są zdefiniowane przez zależności rekurencyjne (4.71) i (4.72).

$$\begin{aligned} w_{ii} &= b_i && (0 \leqslant i \leqslant n) \\ w_{ij} &= w_{i,j-1} + \mathrm{a_j} + \mathrm{b_j} && (0 \leqslant i < j \leqslant n) \end{aligned} \tag{4.71}$$

$$\begin{aligned} p_{ii} &= w_{ii} && (0 \leqslant i \leqslant n) \\ p_{ij} &= w_{ij} + \min_{i < k \leqslant j} (p_{i,k-1} + p_{kj}) && (0 \leqslant i < j \leqslant n) \end{aligned} \tag{4.72}$$

Ostatnie równanie wynika bezpośrednio z (4.69) i z definicji optymalności.

Ponieważ jest około $(1/2)n^2$ wartości p_{ij} oraz ponieważ wzory (4.72) wymagają wyboru spośród $0 < j-i < n$ przypadków, operacja minimalizacji pociągnie za sobą około $(1/6)n^3$ operacji. Knuth wykazał, że dzięki poniższym rozważaniom można ograniczyć czynnik n. Pozwoli to na stosowanie algorytmu w praktyce.

Niech r_{ij} będzie wartością k, która osiąga minimum we wzorze (4.72). Można ograniczyć szukanie r_{ij} do znacznie mniejszego przedziału, tj. zredukować liczbę kroków obliczania $j-i$. Kluczem do rozważań jest obserwacja, że jeżeli znaleźliśmy korzeń r_{ij} optymalnego poddrzewa T_{ij}, to ani rozszerzenie drzewa przez dodatnie węzła z prawej strony, ani usunięcie węzła skrajnego z lewej strony nigdy nie spowoduje przesunięcia korzenia na lewo. Wyraża się to zależnością

$$r_{i,j-1} \leqslant r_{ij} \leqslant r_{i+1,j} \tag{4.73}$$

co ogranicza poszukiwania możliwych rozwiązań dla r_{ij} do przedziału $r_{i,j-1} \dots r_{i+1,j}$ i sprowadza ogólną liczbę elementarnych kroków do $O(n^2)$. Jesteśmy teraz gotowi do szczegółowego skonstruowania algorytmu optymalizacji. Odwołamy się do następujących definicji, opartych na optymalnych drzewach T_{ij}, składających się z kluczy $k_{i+1} \dots k_j$:

(1) a_i: częstość poszukiwania k_i;
(2) b_j: częstość występowania argumentu poszukiwania x między k_j oraz k_{j+1};
(3) w_{ij}: waga drzewa T_{ij};
(4) p_{ij}: ważona długość ścieżki drzewa T_{ij};
(5) r_{ij}: indeks korzenia drzewa T_{ij}.

Mając dany typ

type *indeks* = 0. .*n*

deklarujemy następujące tablice:

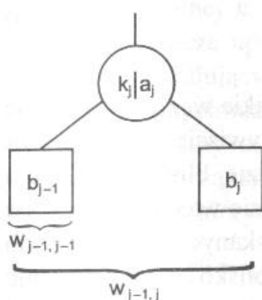
$$a: \textbf{array } [1..n] \textbf{ of } integer;$$
$$b: \textbf{array } [indeks] \textbf{ of } integer; \qquad (4.74)$$
$$p,\ w: \textbf{array } [indeks,\ indeks] \textbf{ of } integer;$$
$$r: \textbf{array } [indeks,\ indeks] \textbf{ of } indeks$$

Zakładamy, że waga w_{ij} została w prosty sposób obliczona z a i b (zob. (4.71)). Przyjmijmy teraz, że w będzie argumentem tworzonej procedury oraz r będzie jej wynikiem, gdyż r całkowicie opisuje strukturę. Tablicę p można uważać za wynik pośredni. Rozpoczynając od przeanalizowania najmniejszych możliwych poddrzew, tj. nie zawierających w ogóle węzłów, przejdziemy do coraz większych drzew. Oznaczmy szerokość $j-i$ poddrzewa T_{ij} przez h. Zgodnie z (4.72) możemy teraz łatwo wyznaczyć wartości p_{ii} dla wszystkich drzew, dla których $h = 0$.

for $i := 0$ **to** n **do** $p[i, i] := w[i, i]$ (4.75)

W przypadku $h = 1$ będziemy mieli do czynienia z drzewami składającymi się z jednego węzła, będącego oczywiście korzeniem (rys. 4.38).

for $i := 0$ **to** $n-1$ **do**
  begin $j := i+1$; $p[i, j] := p[i, i] + p[j, j]$; $r[i, j] := j$ (4.76)
  end



RYSUNEK 4.38
Drzewo optymalne o jednym węźle

Zauważmy, że i oznacza lewą, j zaś – prawą granicę indeksów w rozważanym drzewie T_{ij}. W przypadkach $h>1$ użyjemy instrukcji iteracyjnej z h przebiegającymi od 2 do n. Przypadek $h = n$ obejmuje całe drzewo $T_{0,n}$. W każdym przypadku minimalna długość ścieżki p_{ij} i związany z nią indeks korzenia r_{ij} są wyznaczane przez prostą instrukcję iteracyjną o indeksie k zmieniającym się w przedziale określonym przez (4.73).

for $h := 2$ **to** n **do**
for $i := 0$ **to** $n-h$ **do**
begin $j := i + h$;
  „znajdź m oraz min = minimum $(p[i, m-1] + p[m, j])$ dla wszystkich m takich, że $r[i, j-1] \leqslant m \leqslant r[i+1, j]$"; (4.77)
  $p[i, j] := min + w[i, j]$; $r[i, j] := m$
end

Szczegóły sprecyzowania instrukcji ujętej w cudzysłów można znaleźć w programie 4.6. Średnią długość ścieżki drzewa $T_{0,n}$ podaje teraz iloraz $p_{0,n}/w_{0,n}$, a jego korzeniem jest węzeł o indeksie $r_{0,n}$.

Z algorytmu (4.77) widać, że liczba kroków potrzebnych do wyznaczenia optymalnej struktury jest rzędu $O(n^2)$; także wielkość wymaganej pamięci jest rzędu $O(n^2)$. Jest to nie do przyjęcia w przypadku bardzo dużych n. Pożądane są więc algorytmy bardziej efektywne. Jednym z nich jest algorytm opracowny przez Hu i Tuckera [4.5], który wymaga tylko $O(n)$ pamięci i $O(n \cdot \log n)$ obliczeń. Dotyczy on jednak tylko przypadku, w którym częstości kluczy są równe zeru ($a_i = 0$), czyli rejestrowane są tylko nieudane próby poszukiwania. Walker i Gotlieb opisali [4.11] inny algorytm, wymagający także $O(n)$ elementów pamięci i $O(n \cdot \log n)$ operacji. Zamiast próby znalezienia optimum algorytm ten jedynie obiecuje otrzymanie prawie optymalnego drzewa. Dlatego też może on być oparty na zasadach **heurystycznych**. Ogólna idea tego algorytmu jest następująca.

Rozważmy węzły (rzeczywiste i specjalne) rozmieszczone na liniowej skali, obciążone swoimi częstościami (lub prawdopodobieństwami) dostępów. Znajdźmy następnie węzeł najbliższy „środkowi ciężkości". Węzeł ten jest nazywany **centroidem**, a jego indeks wynosi

$$\frac{1}{w}\left(\sum_{i=1}^{n} i \cdot a_i + \sum_{j=0}^{n} j \cdot b_j\right) \tag{4.78}$$

zaokrąglone do najbliższej liczby całkowitej. Jeżeli wszystkie węzły mają równe wagi, to korzeń drzewa optymalnego pokrywa się oczywiście z centroidem i – wnioskując dalej – w większości przypadków będzie bliskim sąsiadem centroidu. W celu znalezienia lokalnego optimum stosuje się wtedy ograniczone szukanie, następnie zaś procedurę tę stosuje się do uzyskanych w ten sposób dwóch poddrzew. Prawdopodopobieństwo, że korzeń leży blisko centroidu, rośnie wraz z wielkością drzewa n. Po osiągnięciu przez poddrzewa „odpowiedniej" wielkości ich optimum można wyznaczyć przez powyższy, dokładny algorytm.

4.4.10. Drukowanie struktury drzewa

Przejdziemy teraz do związanego z poprzednimi problemu, jak otrzymać wydruk, który *ukaże* strukturę drzewa w odpowiednio jasnej postaci graficznej, ograniczając się tylko do możliwości drukarki wierszowej. To znaczy, chcielibyśmy nakreślić obraz drzewa, drukując klucze jako węzły i łącząc je odpowiednio pionowymi lub poziomymi kreskami.

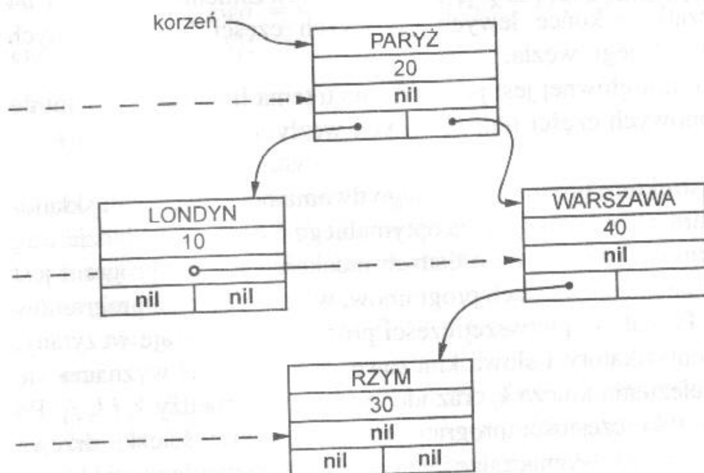
Drukarka wierszowa reprezentowana przez plik tekstowy, tj. ciąg znaków, pozwala tylko na pisanie od lewej strony ku prawej i od góry ku dołowi. Dlatego też wydaje się celowe zbudowanie najpierw reprezentacji *drzewa* oddającej jego strukturę topologiczną. Kolejnym krokiem jest *rozplanowanie* tego obrazu

w sposób uporządkowany na stronie wydruku i obliczenie dokładnych współrzędnych węzłów i linii łączących.

Przy realizacji pierwszego etapu możemy oprzeć się na naszym doświadczeniu z algorytmami generowania drzew i bez wahania zastosować rekurencyjne rozwiązanie zdefiniowanego rekurencyjnie problemu. Sformułujemy procedurę funkcyjną *drzewo*, analogiczną do użytej w programie 4.3. Parametry i oraz j ograniczają indeksy węzłów należących do drzewa. Jego korzeń jest więc zdefiniowany jako węzeł o indeksie r_{ij}. Powinniśmy teraz także zdefiniować typ zmiennych mających reprezentować węzły. Muszą one zawierać dwa wskaźniki do swych poddrzew i klucz węzła. Z powodów, które wyjaśnimy przy omawianiu drugiego etapu, włączymy dwa dodatkowe pola *poz* i *łącze*. Wybrane definicje są pokazane w (4.79), a utworzoną procedurę funkcyjną znajdziemy w programie 4.6.

```
type ref = ↑węzeł;
    węzeł = record klucz: alfa;
                   poz: pozycja;
                   lewe, prawe, łącze: ref                    (4.79)
            end
```

Zauważmy, że procedura ta zlicza liczbę generowanych węzłów za pomocą zmiennej globalnej k. Węzłowi k-temu jest przypisana wartość k-tego klucza, a ponieważ klucze uporządkowane są alfabetycznie, więc k pomnożone przez stały czynnik skalujący wyznacza poziomą współrzędną każdego klucza, która to wartość jest natychmiast zapamiętywana wraz z innymi informacjami. Zauważmy, że odeszliśmy od konwencji stosowania liczb całkowitych jako kluczy, a przyjęliśmy, że są teraz typu *alfa* będącego tablicą znaków o danej (maksymalnej) długości, w którym określono porządek alfabetyczny.



RYSUNEK 4.39
Drzewo zbudowane przez program 4.6

Aby unaocznić to, do czego dotychczas doszliśmy, odwołajmy się do rys. 4.39. Mając dany zbiór n kluczy i obliczoną macierz r_{ij}, instrukcje

k := 0; *korzeń* := *drzewo* (0, *n*)

zbudują wstępnie połączoną strukturę drzewiastą, z zapisanymi poziomymi pozycjami węzłów oraz ich pionowymi pozycjami, określonymi niejawnie przez poziom węzłów w drzewie.

Możemy teraz przejść do drugiego etapu: rozplanowania drzewa na papierze. W tym przypadku musimy zacząć działać od korzenia, idąc w dół, w każdym kroku przetwarzając jeden wiersz węzłów. Ale jak dostaniemy się do węzłów leżących w jednym wierszu? Właśnie w celu połączenia węzłów z tego samego wiersza zadeklarowaliśmy uprzednio pole rekordu – *łącze*. Połączenia, które zbuduje algorytm, są zaznaczone liniami przerywanymi na rys. 4.39. Na każdym etapie przetwarzania zakładamy istnienie połączeń między węzłami, które mają być wydrukowane – nazwiemy te połączenia łańcuchem *bieżącym* (*aktualnym*). Podczas przetwarzania węzła jego następniki (o ile istnieją) połączymy w drugi łańcuch – który nazwiemy łańcuchem *kolejnym*. Po przejściu o jeden poziom niżej łańcuch kolejny staje się łańcuchem bieżącym, a łańcuch kolejny zostaje oznaczony jako pusty.

Szczegółowy zapis algorytmu znajdziemy w programie 4.6. Poniższe uwagi ułatwią jego zrozumienie:

(1) Łańcuchy węzłów w wierszach są generowane od lewej strony ku prawej tak, że skrajnie lewy węzeł staje się ostatnim. Lista musi być odwrócona, aby można było przeglądać węzły w tej samej kolejności. Odwrócenia dokonuje się wtedy, gdy lista *kolejna* staje się listą *bieżącą*.
(2) Drukowana linia zawierająca klucze – zwana linią główną – zawiera także poziome części linii łączących węzły (zob. rys. 4.40). Zmienne $u1$, $u2$, $u3$, $u4$ oznaczają początki i końce lewych i prawych części linii poziomych wychodzących z danego węzła.
(3) Budowa każdej linii głównej jest poprzedzona trzema liniami służącymi do oznaczenia pionowych części linii łączących węzły.

Opiszmy teraz strukturę programu 4.6. Jego dwoma podstawowymi składowymi są: (1) procedura znalezienia drzewa optymalnego o danym rozkładzie wag w i (2) procedura drukowania drzewa o danych indeksach r. Cały program jest przystosowany do działania na tekstach programów, w szczególności programów napisanych w języku Pascal. W pierwszej części program taki zostaje wczytany, rozpoznawane są identyfikatory i słowa kluczowe, następnie zaś wyznacza się liczniki a_i oraz b_j znalezienia klucza k_i oraz identyfikatorów między k_j i k_{j+1}. Po wydrukowaniu statystyki częstości program oblicza długość ścieżki drzewa doskonale zrównoważonego, wyznaczając jednocześnie korzenie jego poddrzew. Następnie jest drukowana średnia ważona długość ścieżki i postać graficzna całego drzewa.

W części trzeciej wywołuje się procedurę *optdrzewo* wyznaczającą drzewo optymalne, a następnie to drzewo jest także drukowane. Na zakończenie te same procedury użyte są do wyznaczenia i wydrukowania drzewa optymalnego wtedy, gdy są uwzględnione jedynie częstości wystąpień kluczy.

PROGRAM 4.6
Znajdowanie optymalnego drzewa poszukiwań

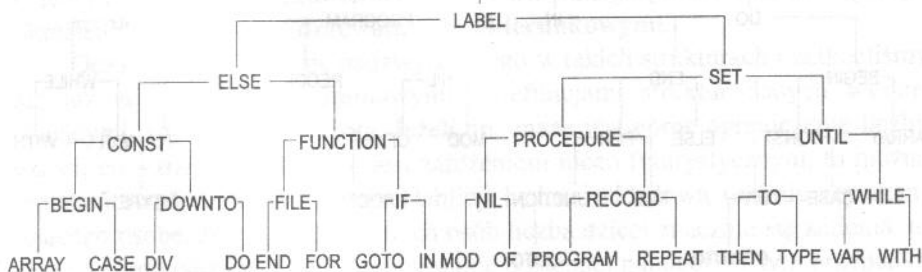
```
program drzewooptymalne(input, output);
const n = 31; {liczba kluczy}
   kln = 10; {max długość klucza}
type indeks = 0. .n;
   alfa = packed array [1. .kln] of char;
var ch: char;
   k1, k2: integer;
   id: alfa; {identyfikator lub klucz}
   buf: array [1. .kln] of char; {bufor znaków}
   klucz: array [1. .n] of alfa;
   i, j, k: integer;
   a: array [1. .n] of integer;
   b: array [indeks] of integer;
   p, w: array [indeks, indeks] of integer;
   r: array [indeks, indeks] of indeks;
   suma, sumb: integer;
function wyważdrzewo(i, j: indeks): integer;
   var k: integer;
begin k := (i+j+1) div 2; r[i, j] := k;
   if i≥j then wyważdrzewo := b[k] else
      wyważdrzewo := wyważdrzewo (i, k−1)+wyważdrzewo(k, j)+w[i, j]
end {wyważdrzewo};

procedure optdrzewo;
   var x, min: integer;
      i, j, k, h, m: indeks;
begin {argument: w, wynik: p, r}
   for i := 0 to n do p[i, i] := w[i, i]; {szerokość drzewa h=0}
   for i := 0 to n−1 do {szerokość drzewa h=1}
   begin j := i+1;
      p[i, j] := p[i, i] + p[j, j];      r[i, j] := j
   end;
   for h := 2 to n do              {h=szerokość rozważanego drzewa}
   for i := 0 to n−h do            {i=lewy indeks rozważanego drzewa}
   begin j := i+h;                 {j=prawy indeks rozważanego drzewa}
      m := r[i, j−1]; min := p[i, m−1]+p[m, j];
      for k := m+1 to r[i+1, j] do
```

```
        begin x := p[i, k-1]+p[k, j];
            if x<min then
                begin m := k; min := x
                end
        end;
        p[i, j] := min+w[i, j]; r[i, j] := m
    end
end {optdrzewo};

procedure drukujdrzewo;
    const lw = 120; {szerokość linii drukarki}
    type ref = ↑węzeł;
        pozycja = 0. .lw;
        węzeł = record klucz: alfa;
                    poz: pozycja;
                    lewe, prawe, łącze: ref
                end;
    var korzeń, bieżący, kolejny: ref;
        q, q1, q2: ref;
        i, k: integer;
        u, u1, u2, u3, u4: pozycja;
    function drzewo(i, j: indeks): ref;
        var p: ref;
    begin if i=j then p := nil else
        begin new(p);
            p↑.lewe := drzewo(i, r[i, j]-1);
            p↑.poz := trunc((lw-kln)*k/(n-1))+(kln div 2);
            k := k+1;
            p↑.klucz := klucz[r[i, j]];
            p↑.prawe := drzewo(r[i, j], j)
        end;
        drzewo := p
    end;

begin k := 0; korzeń := drzewo(0, n);
    bieżący := korzeń; korzeń↑.łącze := nil;
    kolejny := nil;
    while bieżący≠nil do
    begin {schodź w dół; najpierw pisz linie pionowe}
        for i := 1 to 3 do
        begin u := 0; q := bieżący;
            repeat u1 := q↑.poz;
                repeat write(' '); u := u+1
                until u = u1;
```

```
            write('|'); u := u+1; q := q↑.łącze
         until q = nil;
         writeln
      end;
      {teraz drukuj główną linię; przechodząc po węzłach listy bieżącej, zbieraj
      ich następniki i utwórz kolejną listę}
      q := bieżący; u := 0;
      repeat unpack(q↑.klucz, buf, 1);
         {ustaw klucz na poz} i := kln;
         while buf[i] = ' ' do i := i-1;
         u2 := q↑.poz-((i-1) div 2); u3 := u2+i;
         q1 := q↑.lewe; q2 := q↑.prawe;
         if q1 = nil then u1 := u2 else
            begin u1 := q1↑.poz; q1↑.łącze := kolejny;
               kolejny := q1
            end;
         if q2 = nil then u4 := u3 else
            begin u4 := q2↑.poz+1; q2↑.łącze := kolejny;
               kolejny := q2
            end;
         i := 0;
         while u<u1 do begin write(' '); u := u+1 end;
         while u<u2 do begin write('-'); u := u+1 end;
         while u<u3 do begin i := i+1; write(buf[i]); u := u+1 end;
         while u<u4 do begin write('-'); u := u+1 end;
         q := q↑.łącze
      until q = nil;
      writeln;
      {odwróć kolejną listę i uczyń ją listą bieżącą}
      bieżący := nil;
      while kolejny ≠ nil do
         begin q := kolejny; kolejny := q↑.łącze;
            q↑.łącze := bieżący; bieżący := q
         end
   end
end {drukujdrzewo};

begin {inicjowanie tablicy kluczy i liczników}
   klucz[ 1] := 'ARRAY      '; klucz[ 2] := 'BEGIN      ';
   klucz[ 3] := 'CASE       '; klucz[ 4] := 'CONST      ';
   klucz[ 5] := 'DIV        '; klucz[ 6] := 'DOWNTO     ';
   klucz[ 7] := 'DO         '; klucz[ 8] := 'ELSE       ';
   klucz[ 9] := 'END        '; klucz[10] := 'FILE       ';
   klucz[11] := 'FOR        '; klucz[12] := 'FUNCTION   ';
```

```
  klucz[13] := 'GOTO      '; klucz[14] := 'IF        ';
  klucz[15] := 'IN        '; klucz[16] := 'LABEL     ';
  klucz[17] := 'MOD       '; klucz[18] := 'NIL       ';
  klucz[19] := 'OF        '; klucz[20] := 'PROCEDURE ';
  klucz[21] := 'PROGRAM   '; klucz[22] := 'RECORD    ';
  klucz[23] := 'REPEAT    '; klucz[24] := 'SET       ';
  klucz[25] := 'THEN      '; klucz[26] := 'TO        ';
  klucz[27] := 'TYPE      '; klucz[28] := 'UNTIL     ';
  klucz[29] := 'VAR       '; klucz[30] := 'WHILE     ';
  klucz[31] := 'WITH      ';
  for i := 1 to n do
    begin a[i] := 0; b[i] := 0
    end;
  b[0] := 0; k2 := kln;
  {przejrzyj tekst wejściowy i wyznacz a i b}
  while ¬ eof(input) do
  begin read(ch);
    if ch in ['A'..'Z'] then
    begin {identyfikator lub klucz} k1 := 0;
      repeat if k1 < kln then
               begin k1 := k1 + 1; buf[k1] := ch
               end;
          read(ch)
      until ¬ (ch in ['A'..'Z', '0'..'9']);
      if k1 ⩾ k2 then k2 := k1 else
      repeat buf[k2] := ' '; k2 := k2 - 1
      until k2 = k1;
      pack(buf, 1, id);
      i := 1; j := n;
      repeat k := (i+j) div 2;
        if klucz[k] ⩽ id then i := k+1;
        if klucz[k] ⩾ id then j := k-1;
      until i > j;
      if klucz[k] = id then a[k] := a[k] + 1 else
        begin k := (i+j) div 2; b[k] := b[k] + 1
        end
    end else
    if ch = '''' then
      repeat read(ch) until ch = '''' else
    if ch = '{' then
      repeat read(ch) until ch = '}'
  end;
  writeln('KLUCZE i CZĘSTOŚCI WYSTĄPIEŃ:');
  suma := 0; sumb := b[0];
```

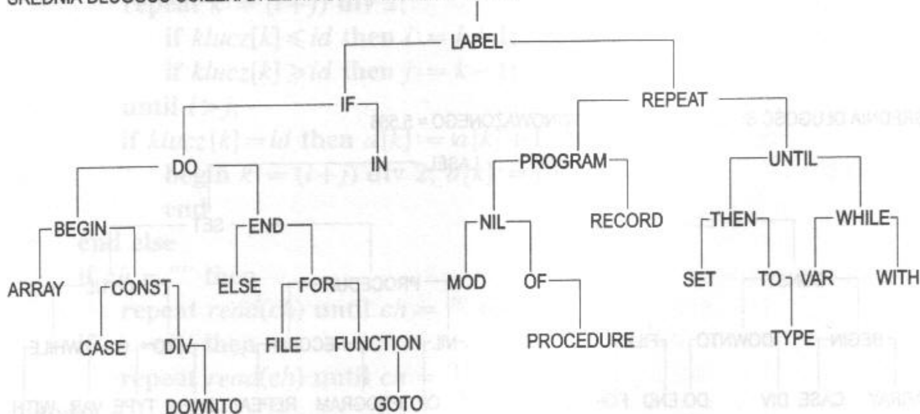
```
    for i := 1 to n do
    begin suma := suma + a[i]; sumb := sumb + b[i];
       writeln(b[i - 1], a[i], '   ', klucz[i])
    end;
    writeln(b[n]);
    writeln('        ________        ________');
    writeln(sumb, suma);
    {wyznacz w na podstawie a i b}
    for i := 0 to n do
    begin w[i, i] := b[i];
       for j := i + 1 to n do w[i, j] := w[i, j - 1] + a[j] + b[j]
    end;
    write('ŚREDNIA DŁUGOŚĆ ŚCIEŻKI DRZEWA
      ZRÓWNOWAŻONEGO = ')
    writeln(wyważdrzewo(0, n)/w[0, n]:6:3); drukujdrzewo;
    optdrzewo;
    write('ŚREDNIA DŁUGOŚĆ ŚCIEŻKI DRZEWA OPTYMALNEGO = ');
    writeln(p[0, n]/w[0, n]:6:3); drukujdrzewo;
    {uwzględnij teraz tylko klucze, ustaw b = 0}
    for i := 0 to n do
    begin w[i, i] := 0;
       for j := i + 1 to n do w[i, j] := w[i, j - 1] + a[j]
    end;
    optdrzewo;
    writeln('DRZEWO OPTYMALNE GDY UWZGLĘDNIA SIĘ TYLKO
        KLUCZE');
    drukujdrzewo
end.
```



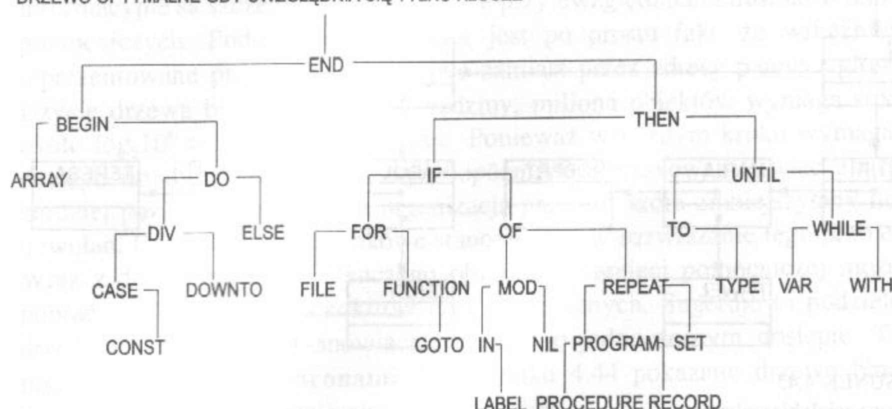
RYSUNEK 4.40
Drzewo doskonale zrównoważone

TABLICA 4.5
Klucze i częstości wystąpień

4	7	ARRAY
14	27	BEGIN
19	0	CASE
15	2	CONST
8	5	DIV
0	0	DOWNTO
0	20	DO
0	8	ELSE
0	28	END
1	0	FILE
0	12	FOR
0	2	FUNCTION
0	0	GOTO
9	13	IF
23	2	IN
208	0	LABEL
22	0	MOD
17	10	NIL
24	7	OF
17	2	PROCEDURE
0	1	PROGRAM
53	1	RECORD
6	8	REPEAT
16	0	SET
10	13	THEN
0	12	TO
6	2	TYPE
1	8	UNTIL
39	5	VAR
0	8	WHILE
0	0	WITH
37		
549	203	



RYSUNEK 4.41
Optymalne drzewo poszukiwań



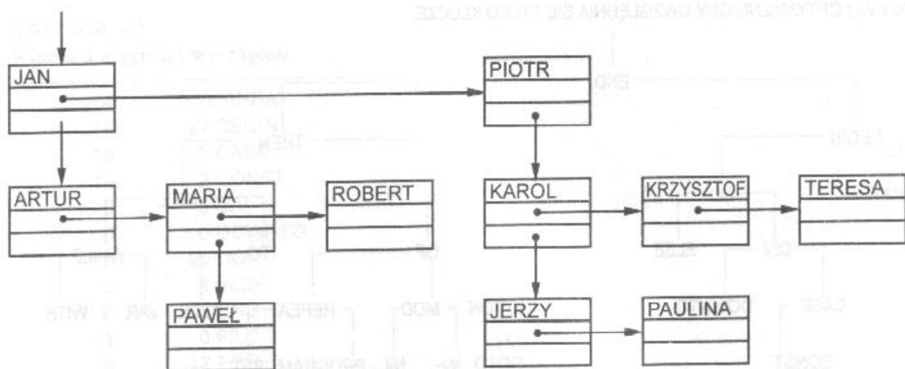
RYSUNEK 4.42
Drzewo optymalne, gdy uwzględnia się tylko klucze

W tablicy 4.5 i na rysunkach 4.40, 4.41, 4.42 przedstawiono wyniki programu 4.6 działającego na własnym tekście. Różnice między trzema rysunkami wykazują, że drzewo zrównoważone nie może być nawet traktowane jako drzewo prawie optymalne oraz że częstości wystąpień elementów nie będących kluczami wpływają w decydujący sposób na postać struktury optymalnej.

4.5. Drzewa wielokierunkowe

Dotychczas ograniczaliśmy nasze rozważania do drzew, w których każdy węzeł miał co najwyżej dwóch potomków, tj. do drzew binarnych. Wystarczają one całkowicie wtedy, gdy chcemy np. przedstawić powiązania rodzinne, zwracając szczególną uwagę na pochodzenie, tzn. na fakt, że każda osoba związana jest ze swymi rodzicami. Mimo wszystko nie miewa się więcej niż dwoje rodziców! Ale co będzie, jeżeli ktoś spojrzy na problem z punktu widzenia potomstwa? Trzeba liczyć się z faktem, że niektórzy ludzie mają więcej niż dwoje dzieci i ich drzewa będą zawierały węzły z wieloma gałęziami. Z braku lepszego określenia nazwiemy je **drzewami wielokierunkowymi**.

Oczywiście nie ma nic nadzwyczajnego w takich strukturach i zetknęliśmy się już ze środkami programowymi i definicjami struktur danych wystarczającymi, aby stawić im czoła. Jeżeli np. znane jest górne ograniczenie liczby dzieci, co – trzeba przyznać – jest założeniem nieco futurystycznym, to można reprezentować dzieci za pomocą tablicy będącej składową rekordu reprezentującego osobę. Jeżeli wśród różnych osób liczba dzieci znacznie się zmienia, to może to spowodować słabe wykorzystanie dostępnej pamięci. W tym przypadku odpowiedniejszym rozwiązaniem będzie ustawienie potomstwa w listę liniową i przypisanie rodzicowi wskaźnika do najmłodszego lub najstarszego potomka.



RYSUNEK 4.43
Drzewo wielokierunkowe

Definicja typu, którą w tym przypadku można przyjąć, jest opisana przez (4.80), a odpowiadającą strukturę danych pokazano na rys. 4.43.

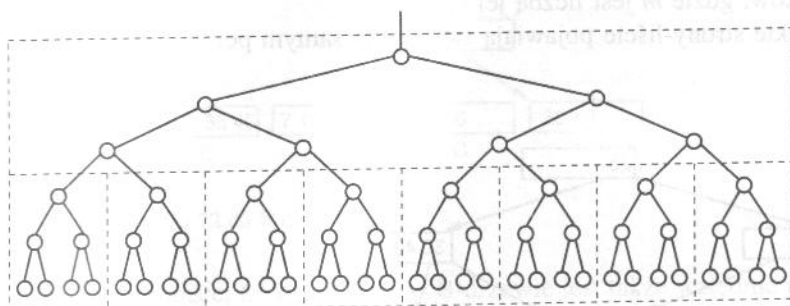
type *osoba* = **record** *imię* : *alfa* ;
rodzeństwo : ↑ *osoba*;
potomstwo : ↑ *osoba*
end (4.80)

Zauważmy teraz, że po obróceniu rys. 4.43 o 45° będzie on wyglądać zupełnie jak drzewo binarne. Ale ten obraz jest mylący, gdyż funkcjonalnie dwa odniesienia (referencje) mają całkiem odmienne znaczenie. Nie można bezkarnie traktować jednakowo rodzeństwa i potomstwa i nie powinno się tego robić nawet przy konstruowaniu definicji danych. Przykład ten można także łatwo rozszerzyć do bardziej skomplikowanej struktury, wprowadzając do rekordu każdej osoby więcej składowych, które mogłyby reprezentować dalsze powiązania rodzinne. Powiązaniem nie dającym się w zasadzie uzyskać z odniesienia między rodzeństwem i potomstwem jest odniesienie do męża lub żony, a nawet odwrotne powiązanie: z ojcem lub matką. Struktura taka szybko rozrasta się w złożony „relacyjny bank danych", który może zawierać w sobie wiele drzew. Algorytmy operowania na takich strukturach są blisko związane z ich definicjami danych i nie ma sensu określenie jakichkolwiek ogólnych zasad lub szerzej stosowalnych metod.

Jednakże istnieje bardzo praktyczna dziedzina zastosowań drzew wielokierunkowych, interesująca z ogólnego punktu widzenia. Jest to konstrukcja i zarządzanie dużymi drzewami, w których trzeba wykonywać operacje wstawiania i usuwania, ale dla których pamięć podstawowa komputera nie jest wystarczająco duża lub jest zbyt kosztowna, aby używać jej do magazynowania danych.

Przyjmijmy więc, że węzły drzewa są przechowywane w pamięci pomocniczej, np. dyskowej. Przedstawione w tym rozdziale dynamiczne struktury

informacyjne są szczególnie odpowiednie przy uwzględnieniu nośników pamięci pomocniczych. Podstawową nowością jest po prostu fakt, że wskaźniki są reprezentowane przez adresy dyskowe zamiast przez adresy pamięci głównej. Użycie drzewa binarnego dla, powiedzmy, miliona obiektów wymaga średnio około $\log_2 10^6 \approx 20$ kroków szukania. Ponieważ w każdym kroku wymaga się dostępu do dysku (z nieodłącznym opóźnieniem czasowym), więc znacznie bardziej pożądana byłaby taka organizacja pamięci, która zmniejszyłaby liczbę odwołań. Drzewo wielokierunkowe stanowi idealne rozwiązanie tego problemu. Wraz z dostępem do pojedynczego obiektu w pamięci pomocniczej możemy pobrać bez dodatkowych kosztów całą grupę danych. Sugeruje to podzielenie drzewa na poddrzewa stanowiące jednostki o jednoczesnym dostępie. Takie poddrzewa nazwiemy **stronami**. Na rysunku 4.44 pokazano drzewo binarne podzielone na strony zawierające po 7 węzłów.



RYSUNEK 4.44
Drzewo binarne podzielone na „strony"

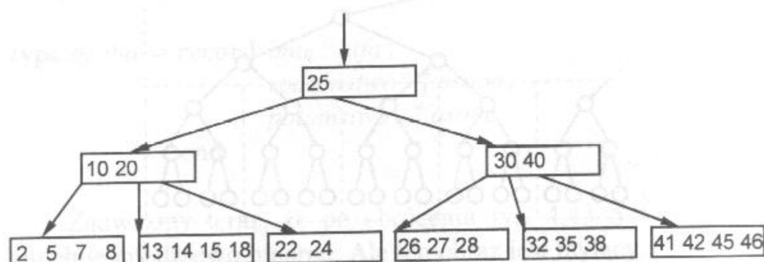
Oszczędne korzystanie z dostępu do dysku – dostęp do każdej strony pociąga za sobą dostęp do dysku – może dać wyraźne efekty. Jeżeli założymy, że na stronie umieścimy 100 węzłów (jest to liczba możliwa do przyjęcia), to dla drzewa o milionie pozycji wystąpi średnio około $\log_{100} 10^6 = 3$ żądań dostępu do stron zamiast 20. Ale oczywiście, jeżeli drzewo rozrastałoby się w sposób „losowy", to w najgorszym przypadku może pojawić się nawet 10^4 żądań dostępu do stron! Jasne jest więc, że schemat kontrolowanego wzrostu jest niemal obowiązkowy w przypadku drzew wielokierunkowych.

4.5.1. B-drzewa

Szukając kryterium kontrolowanego wzrostu drzewa, odrzucimy takie, w których wymaga się doskonałego zrównoważenia – ze względu na zbyt duże dodatkowe koszty równoważenia (wyważania). Reguły te muszą być oczywiście nieco osłabione. Bardzo praktyczne kryterium zaproponował R. Bayer [4.2] w 1970 r.: każda strona (z wyjątkiem jednej) zawiera od n do $2n$ węzłów przy ustalonym n. Tak więc w drzewie o N danych i maksymalnej wielkości strony – $2n$

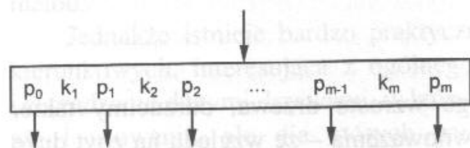
węzłów na stronie – w najgorszym przypadku wystąpi $\log_n N$ żądań dostępów do stron – a realizacja dostępu do stron dominuje przecież w całości nakładów ponoszonych na szukanie. Ponadto istotny współczynnik wykorzystania pamięci wynosi co najmniej 50%, ponieważ strony są co najmniej w połowie pełne. Oprócz tych wszystkich korzyści schemat wymaga względnie prostych algorytmów poszukiwania, wstawiania i usuwania. Omówimy je szczegółowo po kolei. Podstawowe struktury danych są nazywane **B-drzewami** i mają następujące własności (n jest **rzędem** B-drzewa):

(1) Każda strona zawiera co najwyżej $2n$ obiektów (kluczy).
(2) Każda strona, z wyjątkiem strony zawierającej korzeń, zawiera co najmniej n obiektów.
(3) Każda strona jest albo liściem, tj. nie ma potomków, albo ma $m+1$ potomków, gdzie m jest liczbą jej kluczy.
(4) Wszystkie strony-liście pojawiają się na tym samym poziomie.



RYSUNEK 4.45
B-drzewo rzędu 2

Na rysunku 4.45 przedstawiono B-drzewo rzędu 2 o 3 poziomach. Wszystkie strony zawierają 2, 3 lub 4 obiekty. Wyjątek stanowi korzeń, który może zawierać tylko jeden obiekt. Wszystkie liście pojawiają się na poziomie 3. Jeżeli B-drzewo spłaszczymy do jednego poziomu przez wciśnięcie potomków między klucze ich przodków, to klucze utworzą ciąg rosnący od lewej strony ku prawej. Uporządkowanie to reprezentuje naturalne rozszerzenie organizacji drzew binarych i determinuje metodę poszukiwania obiektów o danym kluczu. Rozważmy stronę o postaci pokazanej na rys. 4.46 i niech x będzie argumentem poszukiwania.



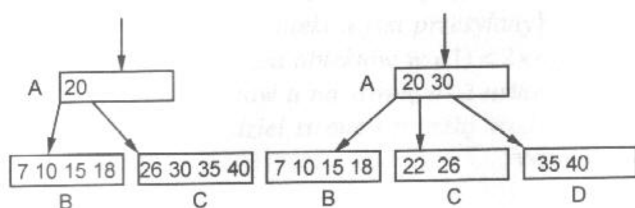
RYSUNEK 4.46
Strona B-drzewa o m kluczach

Zakładając, że strona znajduje się już w pamięci głównej, możemy użyć zwykłych metod poszukiwania pośród kluczy $k_1 \dots k_m$. Jeżeli m jest dostatecznie duże, można zastosować tu przeszukiwanie połówkowe. W przeciwnym przypadku wystarcza

zwyczajne sekwencyjne przeszukiwanie strony. (Zauważmy, że czas zużyty na przeszukanie strony jest najprawdopodobniej niezauważalny w stosunku do czasu potrzebnego na ściągnięcie strony z pamięci pomocniczej do głównej). W przypadku nieudanego poszukiwania znajdziemy się w jednej z poniższych sytuacji:

(1) $k_i < x < k_{i+1}$, dla $1 \leqslant i \leqslant m$. Kontynuujemy poszukiwanie na stronie $p_i\uparrow$.
(2) $k_m < x$. Kontynuujemy poszukiwanie na stronie $p_m\uparrow$.
(3) $x < k_1$. Kontynuujemy poszukiwanie na stronie $p_0\uparrow$.

Jeżeli w jakimś przypadku wyznaczony wskaźnik jest równy **nil**, tj. brak strony będącej potomkiem, to wtedy w całym drzewie nie ma obiektu o kluczu x i poszukiwanie jest zakończone.



RYSUNEK 4.47
Wstawienie klucza 22 do B-drzewa

Zadziwiające, że wstawianie w B-drzewie jest także względnie proste. Jeżeli należy wstawić obiekt na stronie zawierającej $m < 2n$ obiektów, to proces wstawiania ogranicza się do tej strony. Tylko wstawianie do pełnej już strony wpływa na strukturę drzewa i może spowodować przydzielenie nowych stron. Aby zrozumieć, co dzieje się w tym przypadku, odwołajmy się do rys. 4.47, który ilustruje wstawienie klucza 22 do B-drzewa rzędu 2. Oto kolejność postępowania:

(1) Stwierdzamy brak klucza 22 w drzewie. Wstawienie na stronę C jest niemożliwe, gdyż jest ona pełna.
(2) Stronę C dzielimy na dwie strony (tj. przydzielamy pamięć nowej stronie D).
(3) Rozdzielamy $m+1$ kluczy równo na strony C i D, klucz środkowy przesuwamy w górę do strony A, będącej przodkiem stron C i D.

Ten bardzo elegancki schemat zachowuje wszystkie charakterystyczne własności B-drzew. W szczególności, otrzymane po dzieleniu strony zawierają dokładnie n obiektów. Oczywiście wstawienie obiektu do strony przodka może spowodować jej przepełnienie i – w konsekwencji – *dalsze dzielenie*. W krańcowym przypadku taka sytuacja może powtarzać się dopóty, dopóki nie dojdziemy do korzenia. Jest to jedyna możliwość zwiększenia się wysokości B-drzewa. B-drzewo ma więc dziwny zwyczaj: rośnie od liści w kierunku korzenia.

Na podstawie tych skróconych opisów utworzymy teraz program. Jest oczywiste, że sformułowanie rekurencyjne będzie najodpowiedniejsze ze względu na przenoszenie się procesu dzielenia stron wstecz, wzdłuż ścieżki po-

szukiwania. Ogólna struktura programu będzie więc zbliżona do wstawiania do drzewa zrównoważonego, chociaż szczegóły są odmienne.

Przede wszystkim należy sformułować definicję struktury stronicowej. Zdecydowaliśmy, że obiekty będą reprezentowane w postaci tablicy.

```
type strona = record m: indeks;
                     p0: ref;                                    (4.81)
                     e: array [1..nn] of obiekt
              end
```

gdzie

```
const nn = 2*n;
type ref = ↑strona;
     indeks = 0..nn
```

oraz

```
type obiekt = record klucz: integer;
                     p: ref;                                     (4.82)
                     licznik: integer
              end
```

Ponownie, składowa obiektu – *licznik* – występuje zamiast wszystkich rodzajów innych informacji, które mogą być związane z każdym obiektem, ale nie odgrywa żadnej roli w procesie poszukiwania. Zauważmy, że każda strona ma miejsce na $2n$ obiektów. Pole m wskazuje liczbę aktualnie wykorzystanych miejsc na obiekty. Gwarantowane jest wykorzystanie pamięci w co najmniej 50%, gdyż $m \geqslant n$ (z wyjątkiem strony-korzenia).

Algorytm poszukiwania z wstawianiem w B-drzewie jest częścią programu 4.7, sformułowaną w postaci procedury *szukaj*. Jej podstawowa struktura jest nieskomplikowana, przypomina procedurę prostego poszukiwania w drzewie binarnym, z wyjątkiem tego, że decyzja nie jest wyborem jednego z dwóch rozgałęzień. Zamiast tego „poszukiwanie w stronie" jest reprezentowane w postaci przeszukiwania tablicy *e*.

Algorytm wstawiania jest sformułowany w postaci oddzielnej procedury jedynie w celu uzyskania przejrzystości. Procedura ta jest wywoływana po stwierdzeniu przez procedurę *szukaj*, że obiekt powinien być przesłany w górę drzewa (w kierunku korzenia). Fakt ten odnotowuje się za pomocą wynikowego parametru boolowskiego *h*. Pełni on taką funkcję jak w algorytmie wstawiania do drzewa zrównoważonego, gdzie *h* wskazuje, że drzewo urosło. Jeżeli *h* ma wartość *true*, to drugi parametr *u* reprezentuje obiekt przesłany w górę. Zauważmy, że wstawianie rozpoczyna się od hipotetycznych stron, a mianowicie „specjalnych węzłów" z rys. 4.19; nowy obiekt jest przenoszony w górę poprzez parametr *u* do strony-liścia, gdzie dokonuje się rzeczywistego wstawienia. Schemat ten naszkicowano w (4.83).

```
procedure szukaj(x: integer; a: ref; var h: boolean; var u: obiekt);
begin if a = nil then
   begin {brak x w drzewie}
      Podstaw x na u; ustaw jako wartość h true, co wskaże, że obiekt u zostanie
      przesłany w górę drzewa
   end else
   with a↑ do
   begin {poszukaj x na stronie a↑}
      połówkowe przeszukiwanie tablicy;
      if znaleziony then                                                    (4.83)
         zwiększ licznik wystąpień odpowiedniego obiektu else
      begin szukaj (x, potomek, h, u);
         if h then {obiekt u jest przesyłany}
            if (liczba obiektów w a↑)<2*n then
               wstaw u na stronę a↑ i ustaw jako wartość h false
            else dziel stronę i prześlij środkowy obiekt w górę
      end
   end
end
```

Wartość *true* parametru h po wywołaniu procedury *szukaj* w programie głównym wskazuje na rozdzielenie strony-korzenia. Proces ten wymaga oddzielnego zaprogramownia ze względu na wyjątkową funkcję pełnioną przez tę stronę. Składa się on jedynie z przydziału pamięci nowej stronie (korzeniowi) i wstawienia pojedynczego obiektu przekazanego przez parametr u. W konsekwencji nowa strona-korzeń zawiera jedynie pojedynczy obiekt. Szczegóły znajdziemy w programie 4.7.

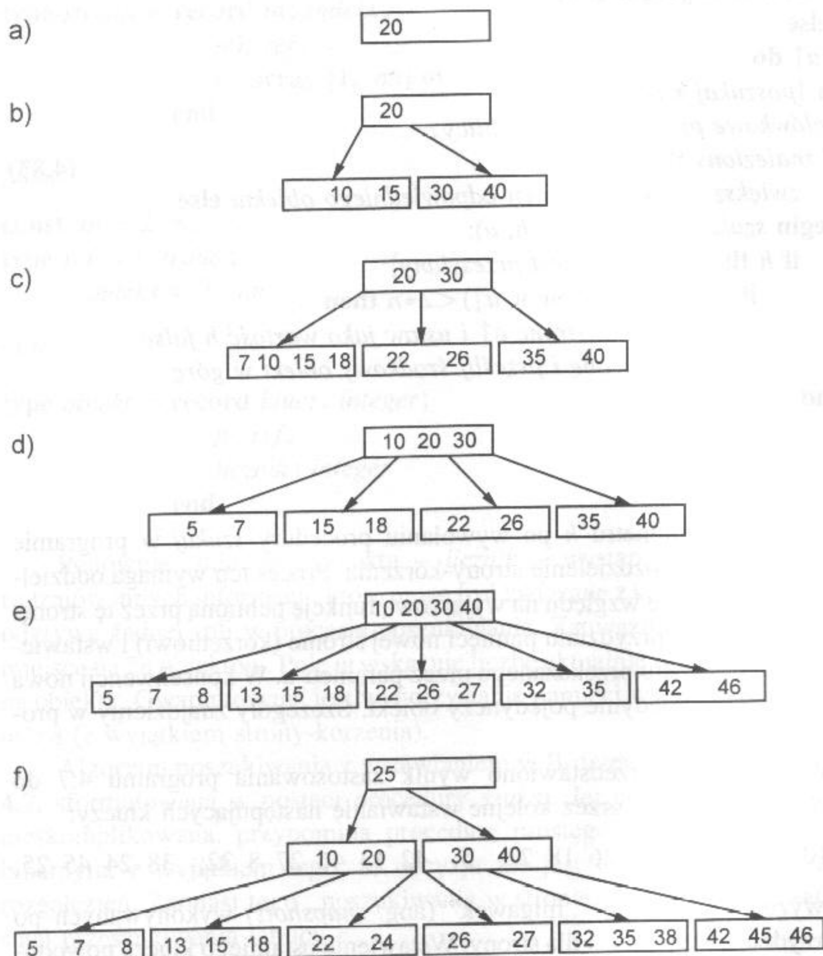
Na rysunku 4.48 przedstawiono wynik zastosowania programu 4.7 do skonstruowania B-drzewa przez kolejne wstawianie następujących kluczy:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

Średniki wyznaczają pozycje „migawek" (ang. *snapshots*) wykonywanych po każdym przydzieleniu pamięci dla strony. Wstawienie ostatniego klucza powoduje dwa dzielenia i przydzielenie trzech nowych stron.

Zwróćmy uwagę na specjalne znaczenie instrukcji **with** w tym programie. Widoczne jest to już w szkicu (4.83). Po pierwsze wskazuje ona, że identyfikatory składowych strony automatycznie odwołują się do strony $a\uparrow$ w instrukcjach objętych instrukcją **with**. W razie przechowywania stron w pamięci pomocniczej – co jest przecież konieczne w dużym systemie banku danych – instrukcję **with** można dodatkowo interpretować jako zalecenie przesłania danej strony do pamięci głównej. Ponieważ każde wywołanie *szukaj* wiąże się z przydzieleniem pamięci dla jednej strony w pamięci głównej, niezbędnych jest co najwyżej $k = \log_n N$ odwołań rekurencyjnych. Tak więc, jeśli drzewo zawiera N pozycji, to musimy zapewnić miejsce w *pamięci głównej dla k* stron. Jest to czynnik

ograniczający wielkość strony $2n$. W rzeczywistości musimy przechować więcej niż k stron, ponieważ wstawianie może spowodować dzielenie stron. Wynika stąd, że najkorzystniej umiejscowić stronę-korzeń w pamięci podstawowej, gdyż każde pytanie musi rozpocząć się od strony-korzenia.



RYSUNEK 4.48
Rozrastanie się B-drzewa rzędu 2

Inną pozytywną cechą organizacji B-drzewa jest to, że jest ona odpowiednia do czysto sekwencyjnego aktualizowania całego banku danych. Każdą stronę pobiera się do pamięci głównej dokładnie jeden raz.

Zasada *usuwania* obiektów z B-drzewa jest prosta, ale komplikuje się przy rozważaniu szczegółów. Możemy wyróżnić tu dwa przypadki:

(1) Usuwany obiekt znajduje się na liściu; wtedy algorytm usuwania jest jasny i prosty.

(2) Obiekt nie znajduje się na liściu. Musi być zastąpiony przez jeden z dwu najbliższych leksykograficznie obiektów, które znajdują się na liściu i mogą być łatwo usunięte.

Znajdowanie odpowiedniego klucza w drugim przypadku jest analogiczne do znajdowania go przy usuwaniu w drzewie binarnym. Schodzimy wzdłuż skrajnie prawych wskaźników do strony-liścia P, wymieniamy usuwany obiekt na skrajnie prawy obiekt strony P i zmniejszamy wielkość strony P o 1.

W dowolnym przypadku po zredukowaniu wielkości należy sprawdzić liczbę obiektów m na zredukowanej stronie. Jeśli $m<n$, to zostałaby naruszona podstawowa własność B-drzew. Należy podjąć wtedy dodatkowe kroki. Stan *niedomiaru* jest wskazywany przez parametr boolowski h przekazywany przez zmienną.

Przeciwdziałamy niedomiarowi, pożyczając lub „anektując" obiekt z jednej z sąsiednich stron. W związku z tym, że wymaga to ściągnięcia strony Q do pamięci głównej – co jest operacją stosunkowo kosztowną – zaleca się wykonanie wszystkiego, co można w tej niekorzystnej sytuacji, i zaanektowanie jednorazowo więcej niż jednego obiektu. Zazwyczaj rozkłada się obiekty ze stron P i Q równomiernie na obie strony. Nazywamy to *równoważeniem* (*wyważaniem*).

Oczywiście zdarza się, że nie można zaanektować żadnego obiektu, ponieważ strona Q osiągnęła już minimalną wielkość n. W tym przypadku ogólna liczba obiektów na stronach P i Q wynosi $2n-1$. Możemy wtedy *połączyć* dwie strony w jedną, dodając środkowy obiekt ze strony przodka stron P i Q, następnie zaś usunąć stronę Q. Proces ten stanowi dokładną odwrotność rozdzielania stron. Możemy go prześledzić na rys. 4.47, analizując usuwanie klucza 22.

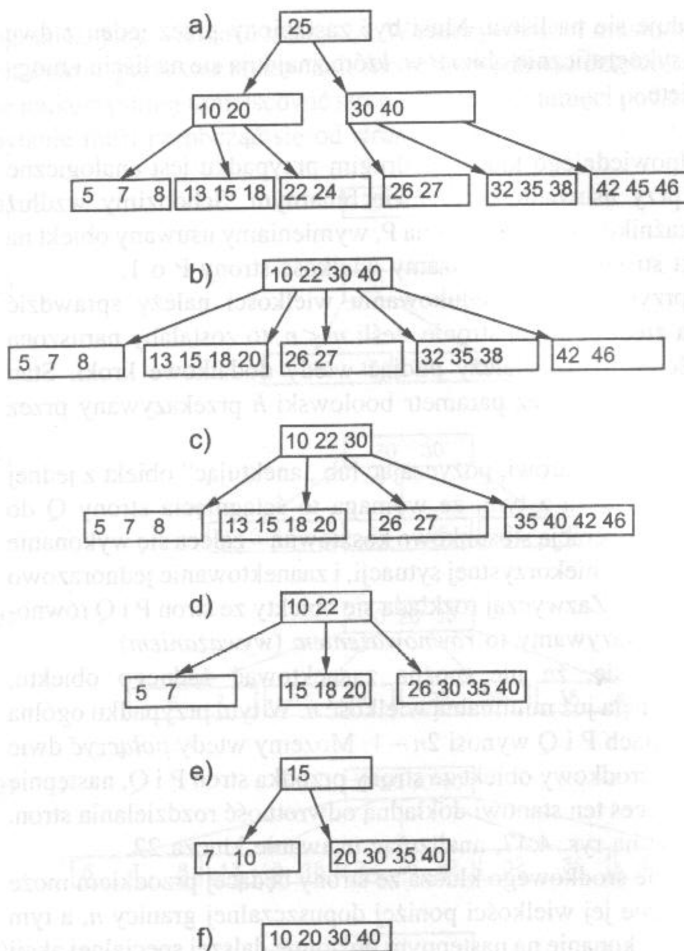
Ponownie, usunięcie środkowego klucza ze strony będącej przodkiem może spowodować zmniejszenie jej wielkości poniżej dopuszczalnej granicy n, a tym samym konieczne jest wykonanie na następnym poziomie dalszej specjalnej akcji (wyważania lub łączenia). W krańcowym przypadku łączenie stron może *przenieść się* aż do korzenia. Zmniejszenie się wielkości korzenia do 0 powoduje jego usunięcie, a tym samym zmniejszenie wysokości B-drzewa. Jest to faktycznie jedyny sposób, aby B-drzewo zmniejszyło wysokość.

Na rysunku 4.49 przedstawiono stopniowe zanikanie B-drzewa z rys. 4.48 przez kolejne usuwanie kluczy:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;

Średniki ponownie wyznaczają miejsca wykonania „migawek", tzn. usuwania stron. Algorytm usuwania został włączony do programu 4.7 w postaci procedury. Warto szczególnie zwrócić uwagę na podobieństwo jego struktury do algorytmu usuwania w drzewie zrównoważonym.

Obszerną analizę zachowania się B-drzew podjęto i przedstawiono we wspomnianym uprzednio artykule Bayera, McCreighta. Zawiera on, w szczególności, rozważania na temat optymalnej wielkości strony n, która zależy istotnie od rodzaju pamięci i systemu komputerowego.



RYSUNEK 4.49
Zanikanie B-drzewa rzędu 2

PROGRAM 4.7
Poszukiwanie z wstawianiem i usuwaniem w B-drzewie

```
program Bdrzewo(input, output);
{poszukiwanie, wstawianie i usuwanie w B-drzewie}
const n = 2; nn = 4;      {wielkość strony}
type ref = ↑strona;
   obiekt = record klucz: integer;
                   p: ref;
                   licznik: integer;
            end;
   strona = record m: 0..nn; {liczba obiektów}
                   p0: ref;
```

```
            e: array[1..nn] of obiekt;
        end;
var korzeń, q: ref; x: integer;
    h: boolean; u: obiekt;

procedure szukaj(x: integer; a: ref; var h: boolean; var v: obiekt);
{szukaj klucza x w B-drzewie o korzeniu a; po znalezieniu zwiększ licznik; gdy
brak, wstaw obiekt o kluczu x i liczniku 1 do drzewa; przypisz obiekt parametrowi
v, jeżeli trzeba go przesłać na niższy poziom; h := „drzewo a zwiększyło
wysokość”}
var k, l, r: integer; q: ref; u: obiekt;

    procedure wstaw;
        var i: integer; b: ref;
    begin {wstaw u na prawo od a↑.e[r]}
        with a↑ do
        begin if m < nn then
            begin m := m+1; h := false;
                for i := m downto r+2 do e[i] := e[i-1];
                e[r+1] := u
            end else
            begin {strona a↑ jest pełna; rozdziel ją i przypisz obiekt, który się
                    pojawił, parametrowi v} new(b);
                if r ≤ n then
                begin if r = n then v := u else
                        begin v := e[n];
                            for i := n downto r+2 do e[i] := e[i-1];
                            e[r+1] := u
                        end;
                    for i := 1 to n do b↑.e[i] := a↑.e[i+n]
                end else
                begin {wstaw u na odpowiedniej stronie}
                    r := r-n; v := e[n+1];
                    for i := 1 to r-1 do b↑.e[i] := a↑.e[i+n+1];
                    b↑.e[r] := u;
                    for i := r+1 to n do b↑.e[i] := a↑.e[i+n]
                end;
                m := n; b↑.m := n; b↑.p0 := v.p; v.p := b
            end
        end {with}
    end {wstaw};

begin {szukaj klucza x na stronie a↑; h = false}
    if a = nil then
    begin {brak obiektu o kluczu x w drzewie} h := true;
        with v do
```

```
          begin klucz := x; licznik := 1; p := nil
          end
      end else
      with a↑ do
      begin l := 1; r := m; {przeszukiwanie połówkowe tablicy}
          repeat k := (l+r) div 2;
              if x ≤ e[k].klucz then r := k-1;
              if x ≥ e[k].klucz then l := k+1;
          until r < l;
          if l-r > 1 then
          begin {znaleziony} e[k].licznik := e[k].licznik+1;
              h := false
          end else
          begin {brak obiektu na tej stronie}
              if r = 0 then q := p0 else q := e[r].p;
              szukaj(x, q, h, u); if h then wstaw
          end
      end
end {szukaj};

procedure usuń(x: integer; a: ref; var h: boolean);
{poszukaj i usuń klucz x w B-drzewie a; jeżeli powstanie niedomiar, to jeżeli to
możliwe, wyważ z sąsiednią stroną, w przeciwnym przypadku połącz strony;
h := „strona a jest za mała"}
    var i, k, l, r: integer; q: ref;
    procedure niedomiar(c, a: ref; s: integer; var h: boolean);
        {a = strona z niedomiarem, c = strona-przodek}
        var b: ref; i, k, mb, mc: integer;
    begin mc := c↑.m; {h = true, a↑.m = n-1}
        if s < mc then
        begin {b := strona na prawo od a} s := s+1;
            b := c↑.e[s].p; mb := b↑.m; k := (mb-n+1) div 2;
            {k = liczba dostępnych obiektów na sąsiedniej stronie b}
            a↑.e[n] := c↑.e[s]; a↑.e[n].p := b↑.p0;
            if k > 0 then
            begin {przesuń k obiektów z b do a}
                for i := 1 to k-1 do a↑.e[i+n] := b↑.e[i];
                c↑.e[s] := b↑.e[k]; c↑.e[s].p := b;
                b↑.p0 := b↑.e[k].p; mb := mb-k;
                for i := 1 to mb do b↑.e[i] := b↑.e[i+k];
                b↑.m := mb; a↑.m := n-1+k; h := false
            end else
            begin {połącz strony a i b}
                for i := 1 to n do a↑.e[i+n] := b↑.e[i];
```

```
            for i := s to mc−1 do c↑.e[i] := c↑.e[i+1];
            a↑.m := nn; c↑.m := mc−1; {dispose(b)}
        end
    end else
    begin {b := strona na lewo od a}
        if s = 1 then b := c↑. p0 else b := c↑.e[s−1].p;
        mb := b↑.m+1; k := (mb−n) div 2;
        if k > 0 then
        begin {przesuń k obiektów ze strony b do a}
            for i := n−1 downto 1 do a↑.e[i+k] := a↑.e[i];
            a↑.e[k] := c↑.e[s]; a↑.e[k].p := a↑.p0; mb := mb−k;
            for i := k−1 downto 1 do a↑.e[i] := b↑.e[i+mb];
            a↑.p0 := b↑.e[mb].p;
            c↑.e[s] := b↑.e[mb]; c↑.e[s].p := a;
            b↑.m := mb−1; a↑.m := n−1+k; h := false
        end else
        begin {połącz strony a i b}
            b↑.e[mb] := c↑.e[s]; b↑.e[mb].p := a↑.p0;
            for i := 1 to n−1 do b↑.e[i+mb] := a↑.e[i];
            b↑.m := nn; c↑.m := mc−1; {dispose(a)}
        end
    end
  end {niedomiar};

  procedure us(p: ref; var h: boolean);
    var q: ref; {globalne a, k}
  begin
    with p↑ do
    begin q := e[m].p;
        if q ≠ nil then
        begin us(q, h); if h then niedomiar(p, q, m, h)
        end else
        begin p↑.e[m].p := a↑.e[k].p; a↑.e[k] := p↑.e[m];
            m := m−1; h := m < n
        end
    end
  end {us}

begin {usuń}
  if a = nil then
  begin writeln('BRAK KLUCZA W DRZEWIE'); h := false
  end else
  with a↑ do
```

```
    begin l := 1; r := m; {przeszukiwanie połówkowe tablicy}
      repeat k := (l+r) div 2;
        if x ≤ e[k].klucz then r := k-1;
        if x ≥ e[k].klucz then l := k+1;
      until l > r;
      if r = 0 then q := p0 else q := e[r].p;
      if l-r > 1 then
      begin {znaleziony, usuń teraz e[k]}
        if q = nil then
        begin {a jest stroną końcową} m := m-1; h := m < n;
          for i := k to m do e[i] := e[i+1];
        end else
        begin us(q, h); if h then niedomiar(a, q, r, h)
        end
      end else
      begin usuń(x, q, h); if h then niedomiar(a, q, r, h)
      end
    end
  end {usuń};

procedure drukujdrzewo(p: ref; l: integer);
  var i: integer;
begin if p ≠ nil then
  with p↑ do
  begin for i := 1 to l do write('      ');
      for i := 1 to m do write(e[i].klucz: 4);
      writeln;
      drukujdrzewo(p0, l+1);
      for i := 1 to m do drukujdrzewo(e[i].p, l+1)
  end
end;
begin korzeń := nil; read(x);
  while x ≠ 0 do
  begin writeln('SZUKANY KLUCZ', x);
     szukaj(x, korzeń, h, u);
     if h then
     begin {wstaw nową podstawową stronę} q := korzeń; new(korzeń);
        with korzeń↑ do
           begin m := 1; p0 := q; e[1] := u
           end
     end;
     drukujdrzewo(korzeń, 1); read(x)
  end;
```

```
    read(x);
    while x≠0 do
    begin writeln('USUŃ KLUCZ', x);
        usuń(x, korzeń, h);
        if h then
        begin {wymiar strony podstawowej został zmniejszony}
            if korzeń↑.m = 0 then
            begin q := korzeń; korzeń := q↑.p0; {dispose(q)}
            end
        end;
        drukujdrzewo(korzeń, 1); read(x)
    end
end.
```

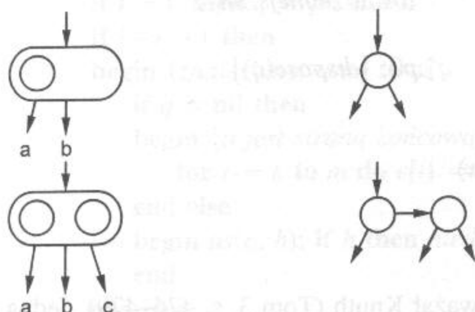
Odmiany schematu B-drzewa rozważał Knuth (Tom 3, s. 476–479). Jedną z istotnych obserwacji jest fakt, że rozdzielanie stron należy opóźnić tak samo, jak opóźnia się łączenie stron, i najpierw należy spróbować zrównoważyć sąsiadujące ze sobą strony. Inne sugerowane usprawnienia wydają się przynosić niewielkie korzyści.

4.5.2. Binarne B-drzewa

Rodzajem B-drzew, który wydaje się najmniej interesujący, są B-drzewa pierwszego rzędu ($n=1$). Ale czasem warto zwrócić uwagę nawet na taki przypadek. Jest oczywiste, że B-drzewa pierwszego rzędu są bezużyteczne do reprezentowania dużych, uporządkowanych, indeksowanych zbiorów danych wymagających korzystania z pamięci pomocniczej; około 50% wszystkich stron zawierałoby tylko po jednej pozycji. Z tego względu zapomnijmy o przechowywaniu w pamięciach pomocniczych i ponownie rozważmy problem przeszukiwania drzew przy użyciu tylko *pamięci jednopoziomowej*.

Binarne B-drzewo (BB-drzewo) składa się z węzłów (stron) zawierających po jednym lub dwa obiekty. Tak więc strona zawiera dwa lub trzy wskaźniki do potomków; sugeruje to termin **2–3 drzewo**. Zgodnie z definicją B-drzew wszystkie liście znajdują się na tym samym poziomie, a pozostałe strony mają dwóch lub trzech potomków (także korzeń). Ponieważ mamy teraz do czynienia jedynie z pamięcią podstawową, konieczne jest oszczędne wykorzystanie obszaru pamięci i dlatego reprezentacja obiektów w węźle w postaci tablicy jest nieodpowiednia. Inną możliwość stanowi dynamiczne, łączone przydzielanie pamięci; oznacza to, że w każdym węźle istnieje lista łączona obiektów o długości 1 lub 2. Ponieważ każdy węzeł ma co najwyżej trzech potomków i w związku z tym wymaga przechowania tylko do trzech wskaźników, chciałoby się połączyć wskaźniki do potomków i wskaźniki w liście obiektów tak, jak to pokazano na rys. 4.50. W ten sposób węzeł B-drzewa traci swą bieżącą postać i obiekty

przejmują na siebie funkcję węzłów w regularnym drzewie binarnym. Jednakże pozostaje konieczność rozróżnienia między wskaźnikami (pionowymi) do potomków i wskaźnikami (poziomymi) do „rodzeństwa" na tej samej stronie. Rozróżnienie to można uzyskać za pomocą tylko jednego bitu, ponieważ jedynie



RYSUNEK 4.50
Reprezentacja węzłów BB-drzewa

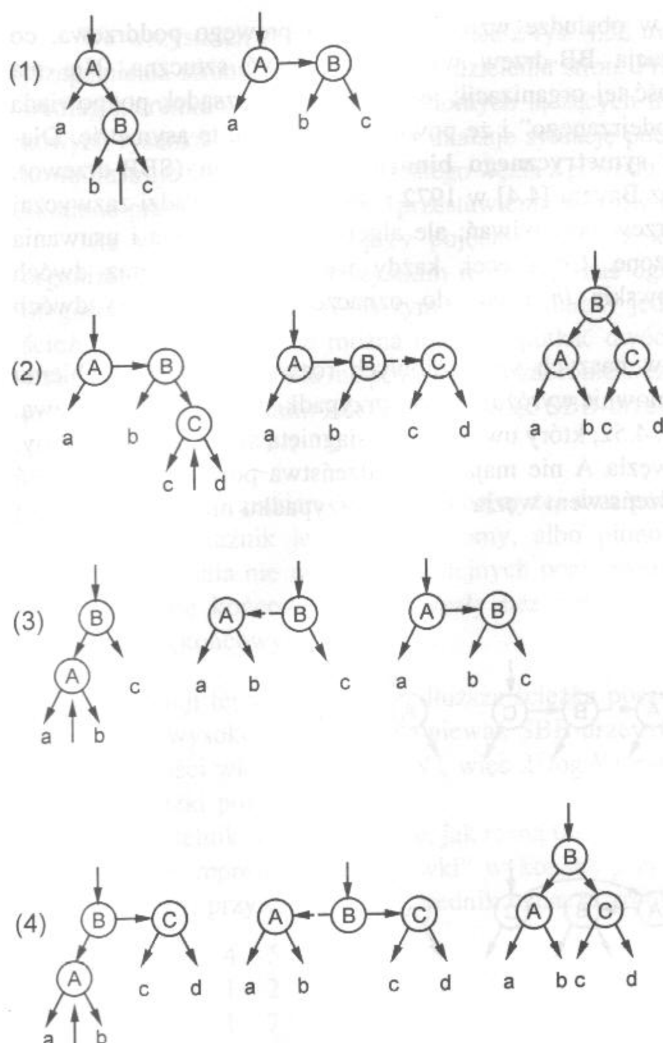
wskaźniki pokazujące na prawo mogą być poziome. Zastosujemy tu zmienną boolowską h o znaczeniu „poziomy". W (4.84) znajdziemy definicję węzła przy użyciu tej reprezentacji. Zaproponował i zbadał ją R. Bayer [4.3] w 1971 r., stwierdzając, że gwarantuje ona maksymalną długość ścieżki $p = 2 \cdot \lceil \log N \rceil$.

```
type węzeł = record klucz: integer;
                    ......................
                    lewe, prawe: ref;                    (4.84)
                    h: boolean
             end
```

Rozważając problem wstawiania kluczy, należy wyróżnić cztery możliwe sytuacje wynikające z rozrastania się lewego lub prawego poddrzewa. Te cztery przypadki pokazano na rys. 4.51. Pamiętajmy, że B-drzewa charakteryzują się własnością rozrastania się od dołu ku korzeniowi oraz spełnieniem warunku, iż wszystkie liście znajdują się na tym samym poziomie.

Najprostszy jest przypadek (1), w którym *prawe* poddrzewo węzła A rośnie oraz A jest jedynym kluczem na swojej (hipotetycznej) stronie. Wtedy potomek B staje się po prostu rodzeństwem A, tzn. wskaźnik pionowy staje się wskaźnikiem poziomym. To proste „podnoszenie się" prawego ramienia nie jest możliwe wtedy, gdy A ma już rodzeństwo. Otrzymujemy wówczas stronę z trzema węzłami i musimy ją podzielić (przypadek 2). Środkowy węzeł B przesyła się w górę na następny wyższy poziom.

Przypomnijmy teraz, że *lewe* poddrzewo węzła B zwiększyło swą wysokość. Jeżeli B jest ponownie sam na stronie (przypadek 3), tj. jego prawy wskaźnik odnosi się do potomka, to lewe poddrzewo (A) może stać się rodzeństwem B.



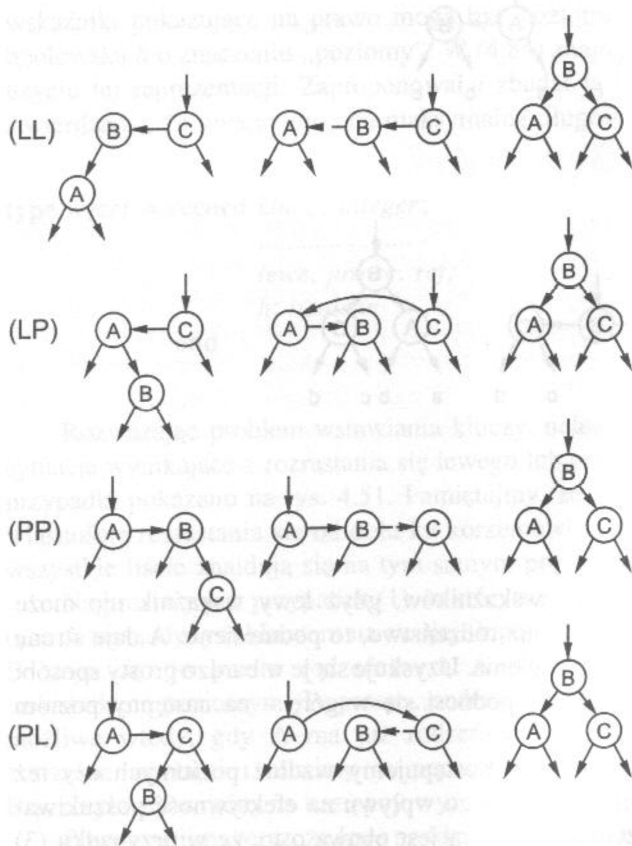
RYSUNEK 4.51
Wstawianie węzła w BB-drzewie

(Konieczna jest prosta zamiana wskaźników, gdyż lewy wskaźnik nie może być poziomy). Jeśli jednak B ma już rodzeństwo, to podniesienie A daje stronę o trzech pozycjach, co wymaga dzielenia. Uzyskuje się je w bardzo prosty sposób: C staje się potomkiem B, którego podnosi się w górę – na następny poziom (przypadek 4).

Należy zauważyć, że fakt, czy postępujemy wzdłuż poziomych czy też pionowych wskaźników, nie ma istotnego wpływu na efektywność poszukiwania klucza. Dlatego też czymś sztucznym jest obawa o to, że w przypadku (3) lewy wskaźnik staje się poziomym, chociaż jego strona w dalszym ciągu nie zawiera więcej niż dwa obiekty. W rzeczywistości algorytm wstawiania ujaw-

nia dziwną asymetrię w obsłudze wzrostu lewego i prawego poddrzewa, co powoduje, że organizacja BB-drzew wydaje się dosyć sztuczna. Nie ma „dowodu" na osobliwość tej organizacji; jednak zdrowy rozsądek podpowiada nam, że jest tu coś „podejrzanego" i że powinniśmy usunąć tę asymetrię. Dlatego powstało pojęcie **symetrycznego binarnego B-drzewa** (SBB-drzewo), badanego również przez Bayera [4.4] w 1972 r. Pojęcie to prowadzi zazwyczaj do efektywniejszych drzew poszukiwań, ale algorytmy wstawiania i usuwania są nieco bardziej złożone. Co więcej, każdy węzeł wymaga teraz dwóch bitów (zmienne boolowskie *lh* i *ph*) do oznaczenia rodzaju jego dwóch wskaźników.

Ograniczając się w naszych szczegółowych rozważaniach do problemu wstawiania, musimy ponownie wyróżnić cztery przypadki rozrastania się drzewa. Są one pokazane na rys. 4.52, który uwidacznia osiągniętą symetrię. Zauważmy, że wzrost poddrzewa węzła A nie mającego rodzeństwa powoduje, że korzeń poddrzewa staje się rodzeństwem węzła A. Tego przypadku nie trzeba już dalej rozważać.



RYSUNEK 4.52
Wstawianie w SBB-drzewach

We wszystkich czterech przypadkach z rys. 4.52 uwzględniono wystąpienie przepełnienia strony, następnie zaś – dzielenia stron. Przypadki te są oznaczone według kierunków wskaźników poziomych łączących troje rodzeństwa na środkowych rysunkach. Lewa kolumna ukazuje sytuację początkową; kolumna środkowa ilustruje fakt podniesienia dolnego węzła z powodu wzrostu jego poddrzewa; kolumna prawa pokazuje wynik przestawienia węzłów (dzielenia stron).

Nie warto upierać się przy pojęciu strony, z którego wywodzi się ta organizacja, gdyż przede wszystkim interesuje nas ograniczenie maksymalnej długości ścieżki do $2 \cdot \log N$. W tym celu wystarczy jedynie upewnić się, że na ścieżce poszukiwania nie można nigdzie spotkać dwóch kolejnych poziomych wskaźników. Jednakże nie ma powodu, aby zabronić węzłów z prawymi i lewymi poziomymi wskaźnikami. Zdefiniujemy więc SBB-drzewo jako drzewo o poniższych własnościach:

(1) Każdy węzeł zawiera klucz i co najwyżej dwa poddrzewa (wskaźniki).
(2) Każdy wskaźnik jest albo poziomy, albo pionowy. Na żadnej ścieżce poszukiwania nie ma dwóch kolejnych poziomych wskaźników.
(3) Wszystkie końcowe węzły (węzły bez potomków) znajdują się na tym samym (końcowym) poziomie.

Z definicji tej wynika, że najdłuższa ścieżka poszukiwania nie przekracza podwojonej wysokości drzewa. Ponieważ SBB-drzewo o N węzłach nie może mieć wysokości większej od $\lceil \log N \rceil$, więc $2\lceil \log N \rceil$ jest górnym ograniczeniem długości ścieżki poszukiwania.

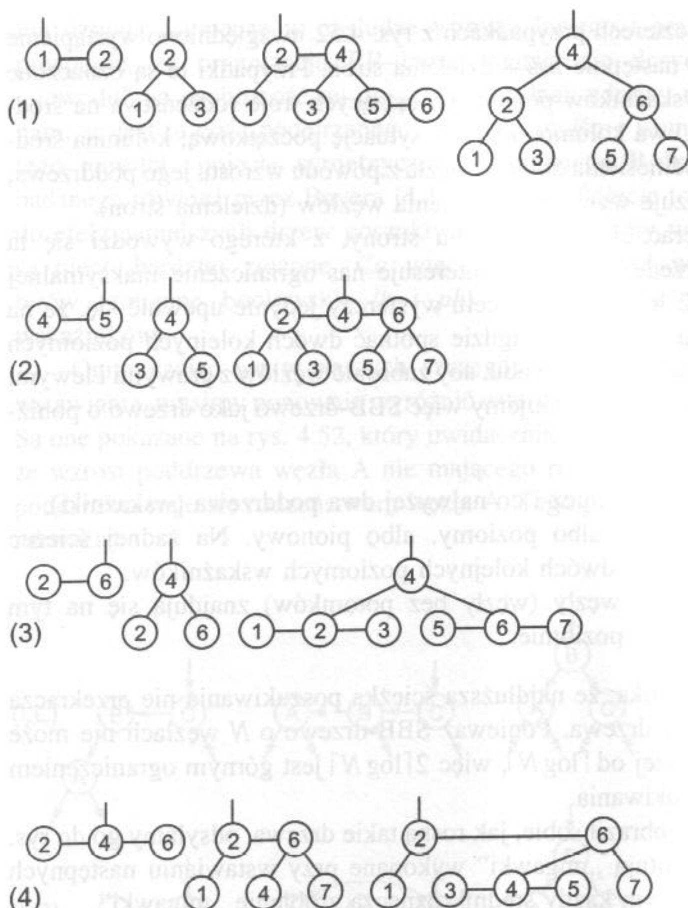
Aby czytelnik wyobraził sobie, jak rosną takie drzewa, odsyłamy go do rys. 4.53. Wiersze reprezentują „migawki" wykonane przy wstawianiu następnych ciągów kluczy, przy czym każdy średnik oznacza zrobienie „migawki".

```
(1) 1 2; 3; 4 5 6; 7;
(2) 5 4; 3; 1 2 7 6;                    (4.85)
(3) 6 2; 4; 1 7 3 5;
(4) 4 2 6; 1 7; 3 5;
```

Rysunki te powodują, że trzecia własność B-drzew staje się niezwykle oczywista: wszystkie końcowe węzły znajdują się na tym samym poziomie. Można dostrzec podobieństwo tych struktur do niedawno przystrzyżonego żywopłotu. Nazywamy takie struktury **żywopłotami.**

Algorytm konstrukcji żywopłotów sformułowano w (4.87), zgodnie z definicją typu węzła (4.86) z dwiema składowymi *lh* i *ph* określającymi, czy lewy i prawy wskaźnik są wskaźnikami poziomymi.

```
type węzeł = record klucz: integer;
                    licznik: integer;
                    lewe, prawe: ref;          (4.86)
                    lh, ph: boolean
             end
```



RYSUNEK 4.53
Rozrastanie się drzew-żywopłotów przez wstawianie elementów ciągów (4.85)

Rekurencyjną procedurę *szukaj* ponownie wzorowano na podstawowym algorytmie wstawiania do drzewa binarnego (zob. 4.87). Dodano trzeci parametr h, który wskazuje, czy drzewo o korzeniu p zmieniło się; odpowiada on bezpośrednio parametrowi h z programu poszukiwania w B-drzewie. Konsekwencją przedstawienia „stron" w postaci list łączonych jest fakt przeglądania strony przez jedno lub dwa odwołania do procedury szukania. Musimy odróżnić przypadek poddrzewa (wskazanego pionowym wskaźnikiem), które rozrosło się, od przypadku węzła (wskazanego poziomym wskaźnikiem), będącego rodzeństwem i otrzymującego nowe rodzeństwo, co wymaga dzielenia stron. Problem ten daje się łatwo rozwiązać przez wprowadzenie zmiennej h o trzech wartościach, oznaczających odpowiednio:

(1) $h = 0$: poddrzewo p nie wymaga zmiany struktury drzewa;
(2) $h = 1$: węzeł p otrzymał rodzeństwo;
(3) $h = 2$: poddrzewo p zwiększyło swą wysokość.

```
procedure szukaj(x: integer; var p: ref; var h: integer);
  var p1, p2: ref;
begin
  if p = nil then
  begin {brak słowa w drzewie; wstaw je}
    new(p); h := 2
    with p↑ do
    begin klucz := x; licznik := 1; lewe := nil;
      prawe := nil; lh := false; ph := false
    end
  end else
  if x < p↑.klucz then
  begin szukaj(x, p↑.lewe, h);
    if h ≠ 0 then
    if p↑.lh then
    begin p1 := p↑.lewe; h := 2; p↑.lh := false;
      if p1↑.lh then
      begin {LL} p↑. lewe := p1↑.prawe;
        p1↑.prawe := p; p1↑.lh := false; p := p1
      end else                                              (4.87)
      if p1↑.ph then
      begin {LP} p2 := p1↑.prawe; p1↑.ph := false;
        p1↑.prawe := p2↑.lewe; p2↑.lewe := p1;
        p1↑.lewe := p2↑.prawe; p2↑.prawe := p; p := p2
      end
    end else
    begin h := h-1; if h ≠ 0 then p↑.lh := true
    end
  end else
  if x > p↑.klucz then
  begin szukaj(x, p↑.prawe, h);
    if h ≠ 0 then
    if p↑.ph then
    begin p1 := p↑.prawe; h := 2; p↑.ph := false;
      if p1↑.ph then
      begin {PP} p↑.prawe := p1↑.lewe;
        p1↑.lewe := p; p1↑.ph := false; p := p1
      end else
      if p1↑.lh then
      begin {PL} p2 := p1↑.lewe; p1↑.lh := false;
        p1↑.lewe := p2↑.prawe; p2↑.prawe := p1;
        p↑.prawe := p2↑.lewe; p2↑.lewe := p; p := p2
      end
    end else
```

```
        begin h := h-1; if h≠0 then p↑.ph := true
        end
    end else                                                    (4.87)
    begin p↑.licznik := p↑.licznik+1; h := 0
    end
end {szukaj}
```

Zauważmy, że działania podjęte w celu zreorganizowania węzłów bardzo przypominają działania z algorytmu przeszukiwania drzewa zrównoważonego (4.63). Na podstawie procedury (4.87) widzimy, że wszystkie cztery przypadki można zrealizować za pomocą prostych zamian wskaźników: pojedynczych zamian, w przypadkach *LL* i *PP*; podwójnych zamian, w przypadkach *LP* i *PL*. W rzeczywistości procedura (4.87) okazuje się trochę prostsza od procedury (4.63). Schemat drzew-żywopłotów okazuje się alternatywny dla kryterium zrównoważenia AVL. Porównanie ich wydajności jest więc możliwe i pożądane.

Powstrzymamy się od analizy matematycznej i skoncentrujemy się na niektórych podstawowych różnicach. Można udowodnić, że *drzewa AVL-zrównoważone są podzbiorem drzew-żywopłotów*. Tak więc klasa drzew-żywopłotów jest większa. Wynika z tego, że długość ich ścieżki jest średnio większa od długości ścieżki w drzewach AVL. Zwróćmy w związku z tym uwagę na „najgorszy przypadek" drzewa (4) z rys. 4.53. Z drugiej strony, reorganizowanie węzłów będzie występowało rzadziej. Dlatego też drzewa zrównoważone będą uprzywilejowane w tych zastosowaniach, w których wyszukiwanie jest znacznie częstsze niż wstawianie (lub usuwanie); jeżeli ten stosunek jest nieduży, to odpowiedniejsze będą drzewa-żywopłoty.

Bardzo trudno określić linię graniczną. Zależy ona nie tylko od stosunku częstości wyszukiwań do zmian struktury, ale również od właściwości realizacji. Dotyczy to zwłaszcza przypadku, w którym rekordy węzłów mają gęsto upakowaną reprezentację i – w konsekwencji – dostęp do pól wymaga wyboru części słowa. Działania na polach boolowskich (*lh*, *ph* w przypadku żywopłotów) mogą być w wielu realizacjach efektywniejsze od działań na polach trójwartościowych (pole *wyważ* w przypadku drzew zrównoważonych).

4.6. Transformacje kluczy (kodowanie mieszające)

W tym punkcie przedstawimy następujący problem ogólny, który posłuży nam do tworzenia rozwiązań demonstrujących metody dynamicznego przydzielania pamięci dla danych:

> Dany jest zbiór *S* obiektów scharakteryzowanych przez wartość klucza. Na zbiorze kluczy jest określona relacja porządkująca. Jak należy zorganizować zbiór *S*, aby wyszukanie obiektu o kluczu *k* wymagało możliwie najmniejszych nakładów.

Oczywiście w pamięci komputera ostateczny dostęp do każdego obiektu uzyskuje się przez określenie adresu a pamięci. Powyższy problem sprowadza się więc w zasadzie do znalezienia odpowiedniej funkcji H odwzorowującej klucze (K) w adresy (A):

$H: K \to A$

W punkcie 4.5 taka funkcja była realizowana w maszynie cyfrowej w postaci różnych algorytmów przeszukiwania list i drzew przy zastosowaniu różnych podstawowych organizacji danych. Zaprezentujemy tu inne – proste w swojej istocie i bardzo efektywne w wielu przypadkach – ujęcie tego problemu. To, że ma ono także kilka wad, przedyskutujemy w następnej kolejności.

Odpowiednią dla tej metody organizacją danych jest struktura tablicy. H jest więc funkcją transformującą klucze w indeksy tablicy, co jest powodem powszechnie stosowanego terminu na oznaczenie tej metody: **transformacja kluczy**. Należy tu zaznaczyć, że nie będziemy musieli odwoływać się do procedur dynamicznego przydzielania pamięci, ponieważ tablica jest jedną z podstawowych struktur statycznych. W zasadzie punkt ten nie powinien znajdować się w rozdziale, w którym omawia się problemy dynamicznych struktur informacyjnych, ale ponieważ z opisywanej tu metody często korzysta się w takich dziedzinach, w których konkuruje ze strukturami drzewiastymi, wydaje się, że można go tu zamieścić.

Podstawową trudność w użyciu transformacji kluczy stanowi fakt, że zbiór możliwych wartości kluczy jest znacznie większy od zbioru adresów (indeksów tablicy). Typowym przykładem jest użycie słów złożonych z co najwyżej 10 liter, jako kluczy identyfikujących poszczególne osoby ze zbioru, powiedzmy, 1000 osób. Mamy więc 26^{10} możliwych kluczy, które musimy odwzorować w 10^3 możliwych indeksów. H nie jest więc funkcją różnowartościową. Mając zadany klucz k, w pierwszym kroku operacji wyszukiwania należy obliczyć związany z nim indeks $h = H(k)$, w drugim zaś kroku – co jest oczywiście konieczne – sprawdzić, czy obiekt o kluczu k jest rzeczywiście identyfikowany przez h w tablicy T, tj. sprawdzić, czy $T[H(k)].klucz = k$. Nasuwają się natychmiast dwa pytania:

(1) Jakiego rodzaju funkcji H należy użyć?
(2) Jak postąpić w sytuacji, gdy H nie wyznaczy adresu pożądanego obiektu?

Odpowiedź na pytanie 2 brzmi, że należy zastosować metodę wyznaczającą adres alternatywny, powiedzmy indeks h', a jeżeli i on nie jest jeszcze adresem potrzebnego obiektu, to trzeci indeks h'' itd. Przypadek znalezienia pod danym adresem innego klucza niż żądany nazwiemy **kolizją**. Algorytm wyznaczający alternatywne indeksy nazwiemy **usuwaniem kolizji**. Poniżej omówimy zasady dokonywania wyboru funkcji transformującej i metody usuwania kolizji.

4.6.1. Wybór funkcji transformującej

Wstępnym kryterium wyboru funkcji transformującej jest wymaganie możliwie równomiernego rozkładu kluczy w obszarze wartości indeksów. Oprócz tego wymagania rozkład nie jest ograniczony żadnym schematem i jest pożądane, aby sprawiał wrażenie całkowicie losowego. Własność ta przyczyniła się do nadania tej metodzie niezbyt naukowej nazwy: **kodowanie mieszające** (haszowanie; ang. *hashing*, tj. siekanie lub mieszanie argumentu); funkcja H jest nazywana **funkcją mieszającą** (haszującą; ang. *hash function*). Oczywiście powinna ona być efektywnie obliczalna, tj. powinna składać się z niewielu podstawowych operacji arytmetycznych.

Przyjmijmy, że jest określona funkcja $ord(k)$ wyznaczająca liczbę porządkową klucza k w zbiorze wszystkich kluczy. Przypomnijmy dalej, że indeksy i tablicy należą do przedziału $0 \ldots N-1$, gdzie N jest wielkością tablicy. Narzucającą się wtedy funkcją jest

$$H(k) = ord(k) \ \mathbf{mod} \ N \qquad (4.88)$$

Funkcja ta ma własność równomiernego rozkładu wartości kluczy w zbiorze indeksów i z tego względu stanowi podstawę większości funkcji transformujących. Ponadto można wyjątkowo łatwo obliczać jej wartości, jeżeli N jest potęgą liczby 2. Lecz właśnie takie rozwiązanie jest nie do przyjęcia w przypadku kluczy będących ciągami liter. Założenie, że wszystkie klucze są jednakowo prawdopodobne, jest w tym przypadku z gruntu fałszywe. Słowa różniące się kilkoma literami najprawdopodobniej zostaną odwzorowane w te same indeksy, powodując bardzo nierównomierny rozkład. Zalecane jest więc, aby N występujące we wzorze (4.88) było **liczbą pierwszą** [4.7]. W konsekwencji zamiast prostego obcięcia cyfr dwójkowych musimy użyć pełnej operacji dzielenia; nie stanowi to jednak istotnej przeszkody, ponieważ większość nowoczesnych komputerów ma wbudowaną operację dzielenia.

Często używa się funkcji mieszających opartych na stosowaniu operacji logicznych, takich jak „różnica symetryczna", do niektórych części kluczy reprezentowanych jako ciąg cyfr dwójkowych. W niektórych maszynach cyfrowych operacje te mogą być szybsze od dzielenia, ale czasem dają zupełnie nierównomierny rozkład kluczy w zbiorze indeksów. Z tego względu nie będziemy dalej rozważać bardziej szczegółowo tych metod.

4.6.2. Usuwanie kolizji

Jeżeli pozycja w tablicy odpowiadająca danemu kluczowi okazuje się niewłaściwym obiektem, to występuje kolizja, tj. dwa obiekty mają klucze odwzorowywane w ten sam indeks. Konieczne jest drugie „sondowanie", oparte na indeksie uzyskanym w sposób deterministyczny z danego klucza. Istnieje kilka metod generowania wtórnych indeksów. Jedną z oczywistych i efektywnych

metod jest łączenie wszystkich pozycji o jednakowym pierwotnym indeksie $H(k)$ w listę łączoną. Nazywane jest to **wiązaniem bezpośrednim**. Elementy tej listy mogą znajdować się w tablicy pierwotnej lub poza nią. W tym drugim przypadku przeznaczony na nie obszar jest nazywany **obszarem nadmiarowym**. Metoda ta jest całkiem efektywna, chociaż jej wadą jest to, że trzeba przechowywać wtórne listy i każda pozycja musi mieć miejsce na wskaźnik (lub indeks) do odpowiadającej jej listy kolidujących obiektów.

Inną metodą usuwania kolizji jest całkowite zrezygnowanie z połączeń i w zamian, po prostu, przeglądanie innych pozycji w tej samej tablicy aż do znalezienia poszukiwanego obiektu lub wolnego miejsca, co pozwala przyjąć, że określonego klucza nie ma w tablicy. Metoda ta jest nazywana **adresowaniem swobodnym** [4.9]. Naturalnie ciąg wyznaczonych indeksów wtórnych musi być zawsze taki sam dla danego klucza. Schemat algorytmu przeglądania tablicy można przedstawić następująco:

$h := H(k)$; $i := 0$
repeat
  if $T[h]$.*klucz* = k **then** *obiekt znaleziony* **else**
  if $T[h]$.*klucz* = *wolny* **then** *brak obiektu* **else** (4.89)
  begin {*kolizja*}
    $i := i+1$; $h := H(k) + G(i)$
  end
until *znaleziony lub brak obiektu w tablicy* (*lub tablica pełna*)

W literaturze można znaleźć propozycje różnych funkcji usuwających kolizję. Artykuł przeglądowy Morrisa w 1968 r. [4.8] spowodował znaczne ożywienie na tym polu. Najprostszą metodą jest sprawdzanie kolejnych pozycji tablicy – interpretując ją jako tablicę cykliczną – aż do chwili znalezienia klucza lub dotarcia do pustego miejsca. Tak więc $G(i) = i$, a indeksy h_i przyjmują w tym przypadku wartości

$$h_0 = H(k)$$
$$h_i = (h_0 + i) \textbf{ mod } N, \quad i = 1 \dots N-1 \tag{4.90}$$

Metoda ta jest nazywana **sondowaniem liniowym** (ang. *linear probing*). Jej wadą jest tendencja do grupowania się obiektów wokół pierwotnych kluczy (kluczy, które nie kolidowały przy wstawianiu). Idealnym rozwiązaniem byłoby równomierne rozproszenie przez funkcję G kluczy na pozostałe adresy. Praktycznie jest to zbyt kosztowne i preferowane są metody kompromisowe, wymagające prostych obliczeń, niemniej jednak lepsze od funkcji liniowej (4.90). W jednej z nich korzysta się z funkcji kwadratowej, tak że ciąg badanych indeksów jest następujący:

$$h_0 = H(k)$$
$$h_i = (h_0 + i^2) \textbf{ mod } N \quad (i > 0) \tag{4.91}$$

Zauważmy, że obliczenie kolejnego indeksu nie musi wymagać wykonania operacji podnoszenia do kwadratu, jeżeli skorzystamy z zależności rekurencyjnej (4.92) dla $h_i = i^2$ oraz $d_i = 2i + 1$:

$$\begin{aligned} h_{i+1} &= h_i + d_i \\ d_{i+1} &= d_i + 2 \end{aligned} \quad (i > 0) \qquad (4.92)$$

gdzie $h_0 = 0$ oraz $d_0 = 1$. Metodę tę nazywamy **sondowaniem kwadratowym** (ang. *quadratic probing*); nie powoduje ona zasadniczo pierwotnego grupowania, chociaż nie wymaga praktycznie dodatkowych obliczeń. Jej niewielką wadą jest to, że podczas sondowania nie są przeszukiwane wszystkie pozycje tablicy, tj. może się zdarzyć, że przy wstawianiu nie znajdzie się wolnego miejsca, chociaż takie istnieją. Gdy N jest *liczbą pierwszą*, to przegląda się co najmniej połowę tablicy. Twierdzenie to można wywieść z następujących rozważań. Jeżeli sondowania i-te oraz j-te zbiegają się w tym samym miejscu tablicy, to możemy wyrazić to przez równanie

$$i^2 \ \mathbf{mod}\ N = j^2 \ \mathbf{mod}\ N$$

lub

$$(i^2 - j^2) \equiv 0 \ (\mathrm{mod}\ N)$$

Rozkładając różnicę na dwa czynniki, otrzymujemy

$$(i+j)\ (i-j) \equiv 0 \ (\mathrm{mod}\ N)$$

Ponieważ $i \neq j$, to wnioskujemy, że i lub j musi być równe *co najmniej* $N/2$, aby zachodziło $(i+j) = cN$, gdzie c jest liczbą całkowitą.

W praktyce wada ta nie ma wielkiego znaczenia, gdyż konieczność wykonania $N/2$ wtórnych sondowań dla uniknięcia kolizji występuje wyjątkowo rzadko, tylko w przypadku prawie pełnej tablicy.

Program 4.8 powstał z programu generatora odsyłaczy (4.5) przez zastosowanie metody rozpraszania. Podstawowe różnice występują w procedurze *szukaj* i w zastąpieniu wskaźnika *sref* przez tablicę słów T. Funkcją mieszającą H jest modulo wielkości tablicy, a kolizje są usuwane metodą sondowania kwadratowego. Zauważmy, że do uzyskania dobrych wyników zdecydowanie przyczynia się to, że rozmiar tablicy jest liczbą pierwszą.

Chociaż metoda transformacji kluczy jest w tym przypadku najefektywniejsza – rzeczywiście skuteczniejsza od organizacji drzewiastych – ma ona również wadę. Po przejrzeniu tekstu i zebraniu słów, chcielibyśmy wypisać słowa w porządku alfabetycznym. Łatwo to zrobić, jeżeli stosujemy organizację drzewiastą, gdyż jej istota polega na przeszukiwaniu uporządkowanego drzewa. Nie jest to jednak proste wtedy, gdy używamy transformacji kluczy. Uwidacznia się teraz pełne znaczenie słowa *mieszanie*. Nie dość, że wydrukowanie tablicy należy poprzedzić sortowaniem (dla prostoty, w programie 4.8 zastosowano sortowanie przez prosty wybór), ale nawet okazuje się korzystne zapamiętanie wstawianych kluczy przez połączenie ich w specjalną listę. Tak więc zwiększona

efektywność metody mieszania dotycząca tylko procesu wyszukiwania jest częściowo niwelowana przez dodatkowe operacje uzupełniające pełny proces generowania uporządkowanego indeksu odsyłaczy.

PROGRAM 4.8
Generator odsyłaczy przy zastosowaniu tablicy kodów mieszających

```
program odsyłacz(f, output);
{generator odsyłaczy przy zastosowaniu tablicy kodów mieszających}
label 13;
const c1=10;        {długość słów}
      c2=8;         {ilość liczb w wierszu}
      c3=6;         {ilość cyfr w liczbie}
      c4=9999;      {maksymalny numer wiersza}
       p=997;       {liczba pierwsza}
       wolne='          ';
type indeks=0..p;
     obref=↑obiekt;
     słowo=record klucz: alfa;
                  pierwszy, ostatni: obref;
                  kolejny: indeks
           end;
     obiekt=packed record
                  lno: 0..c4;
                  nast: obref
           end;
var i, top: indeks;
      k, k1: integer;
      n: integer;   {bieżący numer wiersza}
      id: alfa;
      f: text;
      a: array [1..c1] of char;
      t: array [0..p] of słowo;    {tablica kodów mieszających}

procedure szukaj;
   var h, d, i: indeks;
      x: obref; z: boolean;
   {zmienne globalne: t, id, top}

begin h := ord(id) mod p;
   z := false; d := 1;
   new(x); x↑.lno := n; x↑.nast := nil;
   repeat
      if t[h].klucz=id then
```

```
            begin {znaleziona} z := true;
                t[h].ostatni↑.nast := x; t[h].ostatni := x
            end else
        if t[h].klucz = wolne then
            begin {nowa pozycja} z := true;
                with t[h] do
                begin klucz := id; pierwszy := x; ostatni := x;
                    kolejny := top
                end;
                top := h
            end else
            begin {kolizja} h := h+d; d := d+2;
                if h ≥ p then h := h-p;
                if d = p then
                    begin writeln('PRZEPEŁNIENIE TABLICY'); goto 13
                    end
            end
    until z
end {szukaj};

procedure drukujtablicę;
    var i, j, m: indeks;
    procedure drukujsłowo(s: słowo);
        var l: integer; x: obref;
    begin write('   ', s.klucz);
        x := s.pierwszy; l := 0;
        repeat if l = c2 then
                begin writeln;
                    l := 0; write(' ': c1+1)
                end;
            l := l+1; write(x↑.lno: c3); x := x↑.nast
        until x = nil;
        writeln
    end {drukujsłowo};
begin i := top;
    while i ≠ p do
    begin {przeglądaj listę łączoną i znajdź najmniejszy klucz}
        m := i; j := t[i].kolejny;
        while j ≠ p do
            begin if t[j].klucz < t[m].klucz then m := j;
                j := t[j].kolejny
            end;
        drukujsłowo(t[m]);
        if m ≠ i then
```

```
            begin t[m].klucz := t[i].klucz;
                t[m].pierwszy := t[i].pierwszy;
                t[m].ostatni := t[i].ostatni
            end;
        i := t[i].kolejny
    end
end {drukujtablicę};

begin n := 0; k1 := c1; top := p; reset(f);
    for i := 0 to p do t[i].klucz := wolne;
    while ¬ eof(f) do
    begin if n = c4 then n := 0;
        n := n+1; write(n: c3);    {następny wiersz}
        write(' ');
        while ¬ eoln(f) do
        begin {przeglądaj niepusty wiersz}
            if f↑ in ['A'..'Z'] then
            begin k := 0;
                repeat if k < c1 then
                    begin k := k+1; a [k] := f↑;
                    end;
                    write(f↑); get(f)
                until ¬ (f↑ in ['A'..'Z', '0'..'9']);
                if k ≥ k1 then k1 := k else
                repeat a[k1] := ' '; k1 := k1 - 1
                until k1 = k;
                pack(a, 1, id); szukaj;
            end else
            begin {sprawdź, czy apostrof lub komentarz}
                if f↑ = '''' then
                    repeat write(f↑); get(f)
                    until f↑ = '''' else
                if f↑ = '{' then
                    repeat write(f↑); get(f)
                    until f↑ = '}';
                write(f); get(f)
            end
        end;
        writeln; get(f)
    end;
13: page; drukujtablicę
end.
```

4.6.3. Analiza transformacji kluczy

Wstawianie i wyszukiwanie danych przez transformację kluczy działa oczywiście kiepsko w najgorszym przypadku. Przede wszystkim jest całkiem możliwe, że przy pewnym argumencie wyszukiwania kolejne sondowania trafią we wszystkie zajęte pozycje tablicy, konsekwentnie omijając pozycję pożądaną lub pozycje wolne. Należy jednak mieć zaufanie do poprawności zasad teorii prawdopodobieństwa przy stosowaniu metody mieszającej. Chcielibyśmy sobie zapewnić, aby *średnia* liczba sondowań była mała. Poniższe rozważania probabilistyczne dowodzą, że liczba sondowań jest nawet *bardzo mała*.

Przyjmijmy ponownie, że wszystkie możliwe klucze są jednakowo prawdopodobne, a funkcja mieszająca H rozkłada je równomiernie w zbiorze indeksów tablicy. Załóżmy też, że klucz ma być wstawiony do tablicy o wymiarze n zawierającej już k obiektów. Prawdopodobieństwo trafienia za pierwszym razem w wolne miejsce wynosi $1-k/n$. Jest to również prawdopodobieństwo p_1, że niezbędne jest tylko jedno porównanie. Prawdopodobieństwo, że niezbędne jest tylko jedno powtórne sondowanie, równe jest iloczynowi prawdopodobieństwa kolizji za pierwszym razem i prawdopodobieństwa trafienia w wolne miejsce za następnym razem. Ogólnie, prawdopodobieństwo p_1 wstawiania wymagającego i sondowań wynosi:

$$\begin{aligned}
p_1 &= \frac{n-k}{n} \\
p_2 &= \frac{k}{n}\cdot\frac{n-k}{n-1} \\
p_3 &= \frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{n-k}{n-2} \\
&\dots \\
p_i &= \frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{k-2}{n-2}\cdot\ldots\cdot\frac{k-i+2}{n-i+2}\cdot\frac{n-k}{n-i+1}
\end{aligned} \tag{4.93}$$

Wartość oczekiwana liczby sondowań wymaganych przy wstawianiu klucza $k+1$ wynosi więc

$$E_{k+1} = \sum_{i=1}^{k+1} i\cdot p_i = 1\cdot\frac{n-k}{n} + 2\frac{k}{n}\cdot\frac{n-k}{n-1} + \ldots +$$

$$+ (k+1)\cdot\left(\frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{k-2}{n-2}\ldots\frac{1}{n-k+1}\right) = \frac{n+1}{n-k+1} \tag{4.94}$$

Ponieważ liczba sondowań przy wstawianiu danej jest taka sama jak liczba sondowań przy jej poszukiwaniu, więc wynik (4.94) można stosować do

obliczania średniej liczby sondowań E potrzebnych do znalezienia losowego klucza w tablicy. Przyjmijmy ponownie, że n oznacza wymiar tablicy, a m faktyczną liczbę kluczy w tablicy. Wtedy

$$E = \frac{1}{m}\sum_{k=1}^{m} E_k = \frac{n+1}{m}\sum_{k=1}^{m}\frac{1}{n-k+2} = \frac{n+1}{m}(H_{n+1} - H_{n-m+1}) \qquad (4.95)$$

gdzie $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$ jest funkcją harmoniczną. H_n można przybliżyć przez $H_n \approx \ln(n) + \gamma$, gdzie γ jest stałą Eulera. Podstawiając $\alpha = m/(n+1)$, otrzymamy

$$E = \frac{1}{\alpha}(\ln(n+1) - \ln(n-m+1)) = \frac{1}{\alpha}\ln\frac{n+1}{n+1-m} = \frac{-1}{\alpha}\ln(1-\alpha) \qquad (4.96)$$

α jest w przybliżeniu ilorazem liczby zajętych i dostępnych adresów, nazywanym **współczynnikiem wypełnienia**: $\alpha = 0$ oznacza, że tablica jest pusta; $\alpha = n/(n+1)$ oznacza, że tablica jest pełna. Wartość oczekiwaną E sondowań potrzebnych do wyszukania lub wstawienia losowo wybranego klucza podano w tabl. 4.6 jako funkcję współczynnika wypełnienia α. Wyniki te są zaiste zdumiewające i tłumaczą wyjątkową efektywność metody transformacji kluczy. Nawet przy wypełnieniu tablicy w 90% potrzeba średnio tylko 2,56 sondowań dla umiejscowienia klucza lub znalezienia wolnego miejsca! Zauważmy w szczególności, że wielkości te nie zależą od całkowitej liczby kluczy w tablicy, a jedynie od współczynnika wypełnienia.

TABLICA 4.6
Wartość oczekiwana liczby sondowań jako funkcja współczynnika wypełnienia

α	E
0,1	1,05
0,25	1,15
0,5	1,39
0,75	1,85
0,9	2,56
0,95	3,15
0,99	4,66

W powyższej analizie zakładało się użycie metody usuwania kolizji równomiernie rozpraszającej klucze na pozostałe adresy. Metody stosowane praktycznie dają nieco gorsze rezultaty. Szczegółowa analiza *sondowania liniowego* pozwala określić wartość oczekiwaną liczby sondowań wzorem (4.97) [4.10]:

$$E = \frac{1-\alpha/2}{1-\alpha} \qquad (4.97)$$

W tablicy 4.7 znajdziemy niektóre wartości $E(\alpha)$. Wyniki otrzymane nawet dla najgorszej metody usuwania kolizji są tak dobre, że istnieje pokusa, aby uważać transformację kluczy (kodowanie mieszające) za panaceum na wszystko. Można tak myśleć chociażby dlatego, że jej efektywność znacznie przewyższa nawet najbardziej wymyślne struktury drzewiaste, które poprzednio omawialiśmy, przynajmniej pod względem liczby porównań potrzebnych do wyszukiwania lub wstawiania. Dlatego ważne jest uwypuklenie niektórych wad metody kodowania mieszającego, nawet jeżeli są one oczywiste przy obiektywnej analizie.

TABLICA 4.7
Wartość oczekiwana liczby sondowań dla sondowania liniowego

α	E
0,1	1,06
0,25	1,17
0,5	1,50
0,75	2,50
0,9	5,50
0,95	10,50

Na pewno główną jej wadą, w stosunku do metod dynamicznego przydzielania pamięci, jest *stała wielkość tablicy*, co uniemożliwia dopasowanie jej do aktualnych potrzeb. Dlatego jest tu niezbędne możliwie dokładne określenie *a priori* liczby obiektów w celu uniknięcia złego wykorzystania pamięci lub słabej efektywności (a czasami – przepełnienia tablicy). Nawet wtedy, gdy znana jest dokładna liczba obiektów – przypadek bardzo rzadki – troska o dużą efektywność metody dyktuje niewielkie zwiększenie tablicy (powiedzmy o 10%).

Druga ważna niedogodność metod rozpraszania ujawnia się wtedy, gdy klucze mają być nie tylko wstawiane lub wyszukiwane, ale gdy mają być także usuwane. *Usuwanie* pozycji z tablicy mieszającej jest wyjątkowo *niewygodne*, chyba że stosujemy wiązanie bezpośrednie w oddzielnym obszarze nadmiarowym. Należy więc sprawiedliwie przyznać, że organizacje drzewiaste są wciąż atrakcyjne i zalecane wtedy, gdy ilość danych jest trudna do określenia, znacznie się zmienia, a czasem nawet się zmniejsza.

Ćwiczenia

4.1

Wprowadźmy pojęcie typu rekurencyjnego

rectype $T = T_0$

oznaczające sumę zbioru wartości zdefiniowanych przez typ T_0 i pojedynczej wartości **nic**, tj.

$T = T_0 \cup \{\textbf{nic}\}$

Można wtedy np. uprościć definicję typu *rod* [zob. (4.3)] następująco:

rectype *rod* = **record** *imię*: *alfa*;
ojciec, *matka*: *rod*
end

Jaki jest wzorzec pamięciowy struktury rekurencyjnej odpowiadającej rys. 4.2? Prawdopodobnie realizacja takiej struktury powinna opierać się na schemacie dynamicznego przydzielania pamięci, a pola *ojciec* oraz *matka* z powyższego przykładu powinny zawierać wskaźniki wygenerowane automatycznie, ale ukryte przed programistą. Jakie trudności powstają przy realizacji takiej struktury?

4.2

Zdefiniuj strukturę danych opisaną w ostatnim ustępie p. 4.2 za pomocą pojęć rekordu i wskaźnika. Czy można przedstawić powiązania w tej rodzinie za pomocą pojęcia typu rekurencyjnego zaproponowanego w poprzednim ćwiczeniu?

4.3

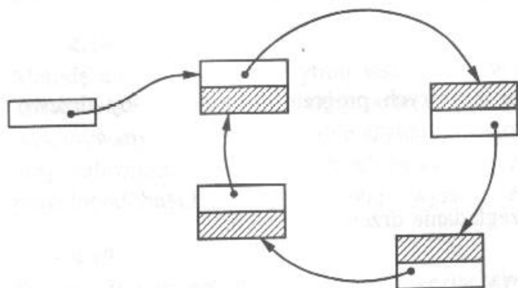
Przyjmijmy, że kolejka Q typu *pierwszy wchodzi – pierwszy wychodzi* (ang. *first-in-first-out*) o elementach typu T_0 jest zrealizowana jako lista łączona. Zdefiniuj odpowiednią strukturę danych, procedury wstawiania i wybierania elementu z Q oraz funkcję sprawdzającą, czy kolejka jest pusta. Procedury powinny zawierać własny mechanizm ekonomicznego wykorzystania pamięci.

4.4

Przyjmijmy, że rekordy listy łączonej zawierają klucz typu *integer*. Napisz program sortowania tej listy według rosnących wartości kluczy. Następnie zbuduj procedurę odwracającą listę.

4.5

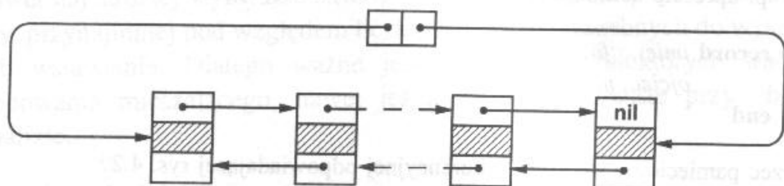
Listy cykliczne (zob. rys. 4.54) są zazwyczaj tworzone z tzw. **nagłówkiem listy**. W jakim celu się go wprowadza? Napisz procedury wstawiania, usuwania i wyszukiwania elementu o zadanym kluczu. Wykonaj to dla listy z nagłówkiem i powtórnie dla listy bez niego.



RYSUNEK 4.54
Lista cykliczna

4.6

Lista dwukierunkowa jest listą elementów połączonych w obydwóch kierunkach. (Zob. rys. 4.55). Obydwa połączenia rozpoczynają się od nagłówka. Analogicznie do poprzedniego ćwiczenia, nalęży zbudować zestaw procedur szukania, wstawiania i usuwania elementów.



RYSUNEK 4.55
Lista dwukierunkowa

4.7

Czy program 4.2 wykona się poprawnie, jeżeli w jego danych wejściowych wystąpi wielokrotnie ta sama para $\langle x, y\rangle$?

4.8

W wielu przypadkach komunikat „ZBIÓR NIE JEST CZĘŚCIOWO UPORZĄDKOWANY" w programie 4.2 nie jest zbyt pomocny. Rozszerz ten program tak, aby drukował ciąg elementów tworzących pętlę, jeżeli ona istnieje.

4.9

Napisz program wczytujący tekst programu, rozpoznający wszystkie definicje i wywoływania procedur (podprogramów) oraz próbujący ustanowić topologiczny porządek wśród podprogramów. Niech zachodzi $P<Q$ zawsze, gdy P jest wywołane w Q.

4.10

Narysuj drzewo konstruowane przez program 4.3, jeżeli dane wejściowe zawierają $n+1$ liczb:

$n, 1, 2, 3, \ldots, n$

4.11

Jakie ciągi węzłów otrzymamy, przeglądając drzewo z rys. 4.23 w porządku preorder, inorder i postorder?

4.12

Znajdź zasadę tworzenia ciągu n liczb, z których program 4.4 utworzyłby drzewo doskonale zrównoważone.

4.13

Rozważmy następujące dwa sposoby przeglądania drzew binarnych:

(a) (1) przeglądanie prawego poddrzewa;
(2) odwiedzanie korzenia;
(3) przeglądanie lewego poddrzewa;

(b) (1) odwiedzanie korzenia;
(2) przeglądanie prawego poddrzewa;
(3) przeglądanie lewego poddrzewa.

Czy istnieją jakieś proste zależności między ciągami węzłów uzyskanymi przez przeglądanie drzewa tymi dwoma sposobami a sposobami określonymi w tekście (p. 4.4.2)?

4.14

Zdefiniuj strukturę reprezentującą n-arne drzewa. Następnie napisz procedurę przeglądającą n-arne drzewo i tworzącą z tych samych elementów drzewo binarne. Przyjmując, że klucz jednego elementu zajmuje k słów, a każdy wskaźnik zajmuje 1 słowo pamięci, określ, ile zaoszczędzi się słów pamięci dzięki zastosowaniu drzewa binarnego zamiast n-arnego.

4.15

Przyjmijmy, że drzewo jest zbudowane na podstawie poniższej rekurencyjnej definicji struktury danych (zob. ćwiczenie 4.1):

rectype *drzewo* = **record** *x*: *integer*;
lewe, *prawe*: *drzewo*
end

Sformułuj procedurę znajdującą element o danym kluczu x i wykonującą na tym elemencie operację P.

4.16

W systemie plikowym katalog wszystkich plików zorganizowano w postaci uporządkowanego drzewa binarnego. Każdy węzeł oznacza plik i określa nazwę pliku oraz między innymi datę ostatniego dostępu do tego pliku, zakodowaną w postaci liczby całkowitej. Napisz program przeglądający drzewo i usuwający wszystkie pliki, do których ostatni dostęp nastąpił przed ustaloną datą.

4.17

W strukturze drzewiastej częstość dostępu do każdego elementu mierzy się empirycznie, przypisując każdemu węzłowi licznik dostępów. Co jakiś czas aktualizuje się organizację drzewa, przeglądając je i generując nowe drzewo za pomocą programu 4.4 oraz wstawiając klucze w porządku malejącym wartości liczników. Napisz program wykonujący tę reorganizację. Czy średnia długość ścieżki poszukiwania dla tego drzewa jest równa, gorsza czy też znacznie gorsza od długości ścieżki poszukiwania dla drzewa optymalnego?

4.18

Metodę analizowania algorytmu wstawiania w drzewie opisaną w p. 4.5 można także zastosować do obliczenia liczby oczekiwanej porównań Po i przesunięć (wymian) Pr, wykonywanych przez sortowanie szybkie (program 2.10) sortujące N-elementową tablicę przy założeniu, że wszystkich $n!$ permutacji n kluczy $\{1, 2, ..., n\}$ jest jednakowo prawdopodobnych. Znajdź analogię i wyznacz Po_n oraz Pr_n.

4.19

Narysuj drzewo zrównoważone o 12 węzłach i o największej wysokości ze wszystkich drzew zrównoważonych o 12 węzłach. W jakiej kolejności należy wstawiać węzły, aby procedura (4.63) zbudowała to drzewo?

4.20

Znajdź ciąg n wstawianych kluczy tak, aby procedura (4.63) wykonała każdą z czterech operacji wyważenia (*LL*, *LP*, *PL*, *PP*) co najmniej raz. Jaka jest najmniejsza długość n takiego ciągu kluczy?

4.21

Znajdź drzewo zrównoważone o kluczach 1 ... n i taką permutację tych kluczy, że jeżeli zadamy je procedurze usuwania (4.64), to wykona ona co najmniej raz każdą z czterech operacji wyważania (*LL*, *LP*, *PL*, *PP*). Jaka jest najmniejsza długość n takiego ciągu kluczy?

4.22

Jaka jest średnia długość ścieżki drzewa Fibonacciego T_n?

4.23

Napisz program generujący drzewo prawie optymalne na podstawie algorytmu wyboru centroidu jako korzenia (4.78).

4.24

Przyjmijmy, że do pustego B-drzewa rzędu 2 wstawiamy (program 4.7) klucze 1, 2, 3, ... Które klucze spowodują dzielenie stron? Które klucze spowodują zwiększanie się wysokości drzewa?
Jeżeli klucze są usuwane z drzewa w tej samej kolejności, to które klucze spowodują łączenie (i usuwanie) stron, które zaś spowodują zmniejszenie wysokości? Podaj odpowiedź dla (a) schematu usuwania stosującego wyważanie (tak jak w programie 4.7) i (b) schematu bez wyważania (w przypadku niedomiaru pobiera się jedną pozycję z sąsiedniej strony).

4.25

Napisz program poszukiwania, wstawiania i usuwania kluczy w binarnym B-drzewie. Użyj definicji węzła (4.84). Schemat wstawiania pokazano na rys. 4.51.

4.26

Znajdź ciąg wstawianych kluczy, który – zaczynając od pustego symetrycznego binarnego B-drzewa – spowoduje co najmniej jednokrotne wykonanie przez procedurę (4.87) każdej z czterech operacji wyważania (*LL*, *LP*, *PP*, *PL*). Jaki ciąg jest najkrótszy?

4.27

Napisz procedurę usuwania elementów w symetrycznym binarnym B-drzewie. Znajdź następnie drzewo i krótki ciąg usuwanych kluczy powodujący co najmniej jednokrotne wystąpienie każdej z czterech sytuacji wymagającej wyważenia.

4.28

Korzystając z dostępnej Ci maszyny cyfrowej, porównaj efektywność algorytmów wstawiania i usuwania w drzewach binarnych, drzewach AVL-zrównoważonych i symetrycznych binarnych B-drzewach. W szczególności zbadaj efekt upakowania danych, tj. wyboru oszczędnej reprezentacji danych, w której przeznacza się tylko 2 bity na informację o wyważeniu w każdym węźle.

4.29

Zmodyfikuj algorytm drukowania z programu 4.6 tak, aby można go było użyć do wypisywania symetrycznych binarnych B-drzew z poziomymi i pionowymi łukami.

4.30

Jeżeli ilość informacji związanej z każdym kluczem jest względnie duża (w porównaniu do samego klucza), to nie należy jej przechowywać w tablicy kodów mieszających. Wyjaśnij dlaczego i zaproponuj schemat reprezentowania takiego zbioru danych.

4.31

Rozważ propozycję rozwiązania problemu grupowania przez budowanie drzew nadmiarowych zamiast list nadmiarowych, tj. przez organizowanie kolidujących kluczy w drzewa. Tak więc każdą pozycję tablicy kodów mieszających (rozproszonych) można uważać za korzeń (być może pustego) drzewa,

4.32

Opracuj schemat wstawiania i *usuwania* z tablicy kodów mieszających, w którym używa się przyrostów kwadratowych do usuwania kolizji. Porównaj ten schemat eksperymentalnie z prostą organizacją drzew binarnych, stosując losowe ciągi kluczy do wstawiania i usuwania.

4.33

Podstawową wadą tablicy kodów mieszających jest fakt, że wielkość tablicy musi być ustalona wtedy, gdy nie jest znana faktyczna liczba pozycji. Załóżmy, że Twój system komputerowy ma mechanizm dynamicznego przydzielania pamięci, który umożliwia otrzymanie pamięci w dowolnym czasie. Tak więc, gdy tablica kodów mieszających H jest pełna (lub prawie pełna), wtedy jest generowana większa tablica H', do której przesyła się wszystkie klucze z tablicy H, a pamięć dla tablicy H może być już przekazana do dyspozycji tego mechanizmu. Nazywa się to *przemieszaniem*. Napisz program realizujący przemieszanie tablicy H o wymiarze n.

4.34

Bardzo często klucze nie są liczbami całkowitymi, ale ciągami liter. Długości tych słów mogą się znacznie różnić i dlatego nie można wygodnie i oszczędnie przechowywać tych słów w polach kluczy o stałej wielkości. Napisz program działający na tablicy kodów mieszających i *kluczach o zmiennej długości*.

Literatura

4.1. Adelson-Velskii G.M., Landis E.M.: *Doklady Akademii Nauk SSSR*, **146**, 1962, s. 263–266; tłumaczenie angielskie w *Soviet Math*, **3**, s. 1259–1263.

4.2. Bayer R., McCreight E.: Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, **1**, No. 3, 1972, s. 173–189.

4.3. Bayer R., McCreight E.: Binary B-trees for Virtual Memory. *Proc. 1971 ACM SIGFIDET Workshop*, San Diego, Nov. 1971, s. 219–235.

4.4. Bayer R., McCreight E.: Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, **1**, No. 4, 1972, s. 290–306.

4.5. Hu T.C., Tucker A.C.: *SIAM J. Applied Math*, **21**, No. 4, 1971, s. 514–432.

4.6. Knuth D.E.: Optimum Binary Search Trees. *Acta Informatica*, **1**, No. 1, 1971, s. 14–25.
4.7. Maurer W.D.: An Improved Hash Code for Scatter Storage. *Comm. ACM*, **11**, No. 1, 1968, s. 35–38.
4.8. Morris R.: Scatter Storage Techniques. *Comm. ACM*, **11**, No. 1, 1968, s. 38–43.
4.9. Peterson W.W.: Addresing for Random-acces Storage. *IBM J. Res & Dev.*, **1**, 1957, s. 130–146.
4.10. Schay G., Spruth W.: Analysis of a File Addresing Method. *Comm. ACM*, **5**, No. 8, 1962, s. 459–462.
4.11. Walker W.A., Gotlieb C.C.: A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees. **In**: *Graph Theory and Computing*, New York, Academic Press 1972, s. 303–323.

5
Struktury językowe i kompilatory

W rozdziale tym postaramy się opracować kompilator (translator) prostego, elementarnego języka programowania. Program kompilatora może posłużyć jako przykład programu o niemałej złożoności i rozmiarze, powstałego w wyniku systematycznego, dobrze zorganizowanego postępowania. Ilustruje on zastosowanie zasad strukturalizacji programu i danych przedstawionych w poprzednich rozdziałach. Dodatkowym celem jest ogólne wprowadzenie czytelnika w zasady budowy i działania kompilatorów. Wiedza i intuicja związane z tym tematem zarówno spotęgują ogólne rozumienie sztuki programowania w języku wysokiego poziomu, jak i ułatwią programiście opracowywanie własnych systemów przydatnych do określonych celów i zastosowań. Ponieważ wiadomo, że inżynieria budowy kompilatorów jest dziedziną skomplikowaną i rozległą, rozdział niniejszy musi mieć z konieczności charakter wprowadzający i opisowy. Być może najważniejszym tutaj faktem jest to, że struktura języka znajduje swoje odbicie w strukturze kompilatora oraz że złożoność lub prostota języka w znacznym stopniu decyduje o złożoności kompilatora. Dlatego też zaczniemy od opisania budowy języka, a następnie zajmiemy się wyłącznie prostymi strukturami, które prowadzą do prostych, modularnych translatorów. Konstrukcje językowe charakteryzujące się tego rodzaju prostotą strukturalną odpowiadają, jak się okazuje, niemal wszystkim prawdziwym potrzebom pojawiającym się w stosowanych w praktyce językach programowania.

5.1. Definicja i struktura języka

Podstawą każdego języka jest **słownik**. Elementy słownika są zazwyczaj nazywane słowami; w świecie języków formalnych nazywa się je **symbolami** (podstawowymi). Cechą charakterystyczną języków jest to, ża pewne ciągi słów są rozpoznawane jako poprawne, dobrze zbudowane **zdania** języka. O innych ciągach słów mówi się, że są niepoprawne lub źle zbudowane. Co decyduje o tym,

że ciąg słów jest zdaniem poprawnym lub nie? Decyduje o tym gramatyka, mówiąc inaczej – składnia lub struktura języka. **Składnię** definiujemy jako zbiór reguł lub formuł, określający zbiór (formalnie poprawnych) zdań. Ważniejsze jest jednak to, że taki zbiór reguł nie tylko pozwala nam decydować, czy dany ciąg słów jest zdaniem, ale także dla konkretnego zdania określa jego strukturę, pomocną w rozpoznaniu znaczenia zdania. Jasne jest więc, że składnia i **semantyka** (znaczenie) są blisko ze sobą powiązane. Dlatego definicje strukturalne będziemy traktować jako definicje pomocnicze dla ważniejszych celów. Nie powinno to nas jednak powstrzymać od początkowego przebadania samych aspektów strukturalnych języka z pominięciem problemów jego znaczenia i interpretacji.

Rozważmy na przykład zdanie „koty śpią". Słowo „koty" jest podmiotem, a „śpią" – orzeczeniem. Zdanie to należy do języka, który np. może być zdefiniowany za pomocą następującej składni:

⟨zdanie⟩ ::= ⟨podmiot⟩⟨orzeczenie⟩
⟨podmiot⟩ ::= *koty* | *psy*
⟨orzeczenie⟩ ::= *śpią* | *jedzą*

Trzy powyższe wiersze pozwalają stwierdzić, że

(1) Zdanie składa się z podmiotu i następującego po nim orzeczenia.
(2) Podmiot jest albo pojedynczym słowem „koty", albo słowem „psy".
(3) Orzeczenie jest albo słowem „śpią", albo słowem „jedzą".

Podstawowa koncepcja jest więc następująca: zdanie języka można wyprowadzić z **początkowego symbolu** ⟨zdanie⟩, stosując kolejno **reguły zastępowania**.

Formalizm, za pomocą którego reguły te są zapisywane, zwany jest **notacją BNF** (Backus-Naur-Form). Był on po raz pierwszy użyty do zdefiniowania Algolu 60 [5.7]. Konstrukcje zdaniowe ⟨zdanie⟩, ⟨podmiot⟩ i ⟨orzeczenie⟩ zwane są **symbolami pomocniczymi** (nieterminalnymi); słowa *koty*, *psy*, *śpią* i *jedzą* nazywają się **symbolami końcowymi** (**terminalnymi**), reguły zaś są nazywane **produkcjami**. Symbole ::= i | nazywa się **metasymbolami** notacji BNF. Jeśli w celu skrócenia zapisu użyjemy pojedynczych, wielkich liter do oznaczania symboli pomocniczych, małych zaś liter do oznaczania symboli końcowych, to poprzedni przykład może mieć następującą postać:

PRZYKŁAD 1

$$\begin{aligned} S &::= AB \\ A &::= x \mid y \\ B &::= z \mid w \end{aligned} \qquad (5.1)$$

Język zdefiniowany przez tę składnię zawiera cztery zdania *xz*, *yz*, *xw*, *yw*.
□

W celu uściślenia omawianych pojęć wprowadzimy następujące definicje matematyczne:

(1) Język $L = L(T, N, P, S)$ jest określony przez
 (a) słownik symboli końcowych – T;
 (b) zbiór symboli pomocniczych (kategorii gramatycznych) – N;
 (c) zbiór produkcji (reguł syntaktycznych) – P;
 (d) symbol S (należący do N), zwany symbolem początkowym.

(2) Język $L(T, N, P, S)$ jest zbiorem ciągów symboli końcowych ξ, które mogą być wyprowadzone z S zgodnie z podaną niżej regułą 3.

$$L = \{\xi \mid S \stackrel{*}{\rightarrow} \xi \quad \text{i} \quad \xi \in T^*\} \tag{5.2}$$

 (liter greckich używamy do oznaczania ciągów symboli).
 T^* oznacza zbiór wszystkich ciągów symboli z T.

(3) Ciąg δ_n może być *wyprowadzony* z ciągu δ_0 wtedy i tylko wtedy, gdy istnieją ciągi $\delta_1, \delta_2, \ldots, \delta_{n-1}$ takie, że każdy ciąg δ_i może być bezpośrednio wyprowadzony z δ_{i-1} zgodnie z podaną poniżej regułą 4.

$$(\delta_0 \stackrel{*}{\rightarrow} \delta_n) \leftrightarrow ((\delta_{i-1} \rightarrow \delta_i) \quad \text{dla} \quad i = 1, \ldots, n) \tag{5.3}$$

(4) Ciąg η może być *bezpośrednio wyprowadzony* z ciągu ξ wtedy i tylko wtedy, gdy istnieją ciągi $\alpha, \beta, \xi', \eta'$ takie, że
 (a) $\xi = \alpha \xi' \beta$;
 (b) $\eta = \alpha \eta' \beta$;
 (c) P zawiera produkcję $\xi' ::= \eta'$.

Uwaga: notacji $\alpha ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ używamy jako skróconego zapisu zbioru produkcji $\alpha ::= \beta_1, \alpha ::= \beta_2, \ldots, \alpha ::= \beta_n$.

Ciąg xz z przykładu 1 może być wyprowadzony za pomocą następującego ciągu wyprowadzeń bezpośrednich: $S \rightarrow AB \rightarrow xB \rightarrow xz$; stąd $S \stackrel{*}{\rightarrow} xz$ i ponieważ $xz \in T^*$, więc xz jest zdaniem języka, tzn. $xz \in L$. Zauważmy, że w ostatnim kroku wyprowadzenia otrzymujemy ciąg, który zawiera tylko symbole *końcowe*, oraz że symbole pomocnicze A i B mogły się pojawić tylko we wcześniejszych krokach wyprowadzenia. Reguły gramatyczne zwane są produkcjami, ponieważ określają one, jak można wyprowadzać lub produkować nowe formy (zdaniowe).

Język nazywamy **bezkontekstowym** wtedy i tylko wtedy, gdy można go zdefiniować za pomocą zbioru produkcji bezkontekstowych. Produkcja jest bezkontekstowa wtedy i tylko wtedy, gdy jest postaci

$$A ::= \xi \qquad (A \in N, \xi \in (N \cup T)^*)$$

tj. jeśli jej lewa strona składa się z pojedynczego symbolu pomocniczego i może być zastąpiona przez ξ niezależnie od kontekstu, w jakim pojawia się A. Jeśli produkcja jest postaci

$$\alpha A \beta ::= \alpha \xi \beta$$

to zwana jest **kontekstową**, ponieważ zastąpienie A przez ξ może się zdarzyć tylko w kontekście α i β. W następnych punktach tego rozdziału ograniczymy się do rozważania języków bezkontekstowych.

W przykładzie 2 pokazano, jak dzięki zastosowaniu rekursji można wyprowadzić nieskończenie wiele zadań za pomocą skończonego zbioru produkcji.

PRZYKŁAD 2

$$S ::= xA$$
$$A ::= z \mid yA \tag{5.4}$$

Następujące zdania mogą być wyprowadzone z symbolu początkowego S:

xz

xyz

xyyz

xyyyz

.....

□

5.2. Analiza zdań

Podstawowym zadaniem translatora języka nie jest generowanie, lecz **rozbiór** zdań i struktur zdaniowych. Wynika stąd, że kolejne kroki generujące zdanie muszą być odtworzone w odwrotnej kolejności (po wczytaniu zdania). Jest to na ogół bardzo skomplikowane, a czasami nawet niewykonalne zadanie. Jego złożoność zależy bardzo silnie od rodzaju reguł produkcji użytych przy definiowaniu języka. Opracowanie algorytmów rozbioru dla języków o dość skomplikowanych regułach strukturalnych jest zadaniem teorii **analizy składniowej**. Naszym celem jest tylko naszkicowanie metody konstrukcji algorytmów rozbioru, tak prostych i efektywnych, aby można było stosować je w praktyce. Wynika stąd po prostu, że koszt liczenia związany z analizą zdania musi być funkcją liniową długości zdania; w najgorszym zaś razie funkcja zależności może mieć postać $n \cdot \log n$, gdzie n jest długością zdania. Oczywiście nie będziemy się zajmować problemem znalezienia algorytmu rozbioru dla zadanego języka, ale będziemy działać pragmatycznie w kierunku odwrotnym: zdefiniujemy algorytm efektywny i wtedy określimy klasę języków, które mogą być poddawane jego działaniu [5.3].

Pierwszą konsekwencją podstawowego założenia o efektywności algorytmu jest to, że określenie kolejnego kroku analizy musi zależeć tylko od obecnego stanu obliczenia i od pojedynczego, aktualnie wczytywanego symbolu. Drugim najważniejszym żądaniem jest to, że żadnego z kolejnych kroków analizy nie można cofnąć. Te dwa założenia określają technikę rozbioru zwaną zazwyczaj **analizą bez powrotów z wyprzedzeniem o jeden symbol** (ang. *one-symbol--lookahead without backtracking*).

Opiszemy teraz metodę zwaną rozbiorem **generacyjnym** lub **zstępującym** (ang. *topdown*). Polega ona na próbie odtworzenia kroków generacyjnych (tworzących na ogół drzewo) od symbolu początkowego do zdania końcowego, „z góry na dół" [5.5] i [5.6]. Wróćmy do przykładu 1. Mamy dane zdanie *psy jedzą* i musimy określić, czy należy ono do języka. Z definicji, może tak być tylko wtedy, jeśli można je wyprowadzić z symbolu początkowego ⟨zdanie⟩. Patrząc na reguły składniowe, widzimy, że może ono tylko wtedy być zdaniem, jeśli jest podmiotem z następującym po nim orzeczeniem. Teraz nasze zadanie podzielimy na dwie części; po pierwsze określimy, czy początkową część zdania można wyprowadzić z symbolu ⟨podmiot⟩. Jest to prawda, bo słowo *psy* może być bezpośrednio wyprowadzone; symbol *psy* zakreślamy w zdaniu wejściowym (miejsce, z którego czytamy, przesuwamy o jedną pozycję w prawo) i przechodzimy do drugiej części naszego zadania – zbadania, czy pozostałą część zdania można wyprowadzić z symbolu ⟨orzeczenie⟩. Ponieważ znowu tak jest, więc proces rozbioru kończy się wynikiem pomyślnym. Graficznie możemy ten proces przedstawić za pomocą poniższego diagramu, w którym po lewej stronie są zaznaczone symbolicznie kolejne podzadania (cele), a po prawej stronie występuje pozostała do wczytania część analizowanego ciągu słów.

⟨zdanie⟩	*psy jedzą*
⟨podmiot⟩ ⟨orzeczenie⟩	*psy jedzą*
psy ⟨orzeczenie⟩	*psy jedzą*
⟨*orzeczenie*⟩	*jedzą*
jedzą	*jedzą*
- - -	- - -

Kolejny przykład ilustruje proces analizy zdania *xyyz* zgodnie z produkcjami z przykładu 2.

S	*xyyz*
xA	*xyyz*
A	*yyz*
yA	*yyz*
A	*yz*
yA	*yz*
A	*z*
z	*z*
- - -	- - -

Ponieważ proces odtwarzania kroków generujących zdanie zwany jest **rozbiorem**, więc opisany powyżej algorytm nazywa się **algorytmem rozbioru**. W omawianych dwóch przykładach decyzja o zastąpieniu symbolu pomocniczego mogła być powzięta po zbadaniu kolejnego, pojedynczego symbolu ciągu wejściowego. Nie zawsze jest to jednak możliwe. Rozważmy następujący przykład:

PRZYKŁAD 3

$S ::= A \mid B$

$A ::= xA \mid y$ (5.5)

$B ::= xB \mid z$

□

Spróbujemy dokonać rozbioru zdania *xxxz*

S	$xxxz$
A	$xxxz$
xA	$xxxz$
A	xxz
xA	xxz
A	xz
xA	xz
A	z

i ... utknęliśmy. Źródłem trudności jest pierwszy krok. Decyzja o tym, czy S ma być zastąpione przez A czy też przez B, nie może być podjęta jedynie na podstawie sprawdzenia kolejnego symbolu ciągu wejściowego. Istnieje co prawda rozwiązanie polegające na posuwaniu się zgodnie z jedną z wybranych możliwości dopóty, dopóki można, a następnie na powrocie wzdłuż tej samej drogi do ostatniego punktu powodującego trudności. Postępowanie takie nazywa się **analizą z powrotami**. Dla języka z przykładu 3 nie istnieje żadne ograniczenie liczby kroków, które należy unieważnić. Sytuacja taka jest oczywiście wysoce niepożądana; dlatego też w praktycznych zastosowaniach powinno się unikać struktur językowych powodujących powroty w procesie rozbioru. Dopuszczać więc będziemy tylko systemy gramatyczne z następującym ograniczeniem: symbole początkowe alternatywnych prawych stron produkcji muszą być różne.

REGUŁA 1

Dla zadanej produkcji

$A ::= \xi_1 \mid \xi_2 \mid \dots \mid \xi_n$

zbiory pierwszych symboli w zdaniach, które mogą być wyprowadzone z ξ_i muszą być rozłączne, tzn.

$pierw(\xi_i) \cap pierw(\xi_j) = \varnothing$ dla wszystkich $i \neq j$

Zbiór $pierw(\xi)$ jest zbiorem wszystkich symboli końcowych, które mogą wystąpić na pierwszej pozycji w zdaniach wyprowadzanych z ξ. Zbiór ten może być wyznaczony wg następujących zasad:

(1) jeśli pierwszy symbol argumentu jest symbolem końcowym, to $pierw(a\xi) = \{a\}$

(2) jeśli pierwszy symbol jest symbolem pomocniczym i istnieje produkcja

$A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

to

$pierw(A\xi) = pierw(\alpha_1) \cup pierw(\alpha_2) \cup \ldots \cup pierw(\alpha_n)$

Zauważmy, że w przykładzie 3: $x \in pierw(A)$ i $x \in pierw(B)$. Dlatego też pierwsza produkcja nie spełnia reguły 1. W rzeczywistości nietrudno znaleźć taką składnię dla języka z przykładu 3, która spełniałaby regułę 1. Rozwiązanie polega na wyłączeniu wspólnej części „przed nawias". Podane poniżej produkcje są *równoważne* produkcjom (5.5) w tym sensie, że generują ten sam zbiór zadań:

$$S ::= C \mid xS \qquad (5.5a)$$
$$C ::= y \mid z$$

Niestety, reguła 1 nie jest dostatecznie silna na to, aby osłonić nas przed innymi kłopotami.

PRZYKŁAD 4

$$S ::= Ax$$
$$A ::= x \mid \varepsilon \qquad (5.6)$$

ε oznacza tutaj pusty ciąg symboli. Jeśli spróbujemy dokonać rozbioru zdania x, to możemy dostać się w następujący „ślepy zaułek":

S	x
Ax	x
xx	x
x	- -

□

Kłopot wyniknął z zastosowania produkcji $A ::= x$ zamiast $A ::= \varepsilon$. Sytuacja taka, zwana **problemem pustego słowa**, powstaje tylko wtedy, gdy z symbolu pomocniczego można wyprowadzić pusty ciąg symboli. W celu jej uniknięcia wprowadzimy kolejną regułę.

REGUŁA 2

Dla każdego symbolu $A \in N$, z którego można wyprowadzić pusty ciąg symboli ($A \xrightarrow{*} \varepsilon$), zbiór jego pierwszych symboli musi być rozłączny ze zbiorem symboli, które mogą następować po dowolnym ciągu wyprowadzonym z A, tzn.

$pierw(A) \cap nast(A) = \varnothing$

Zbiór $nast(A)$ wyznacza się następująco: dla każdej produkcji P_i postaci

$X ::= \xi A \eta$

przez S_i oznaczamy $pierw(\eta_i)$; suma wszystkich takich zbiorów S_i tworzy zbiór $nast(A)$. Jeśli z co najmniej jednego η_i możemy wyprowadzić pusty ciąg symboli,

to zbiór $nast(X)$ musi być również włączony do $nast(A)$. W przykładzie 4 reguła 2 nie jest spełniona dla symbolu A, ponieważ

$pierw(A) = nast(A) = \{x\}$

Wielokrotne powtarzanie się pewnego wzorca symboli wyrażamy zazwyczaj za pomocą rekurencyjnej definicji konstrukcji zdaniowej. Na przykład produkcja

$A ::= B \mid AB$

opisuje zbiór ciągów B, BB, BBB, ... Zgodnie z regułą 1 użycie jej jest teraz niedozwolone, ponieważ

$pierw(B) \cap pierw(AB) = pierw(B) \neq \varnothing$

Jeśli zastąpimy tę produkcję jej nieznacznie zmodyfikowaną wersją

$A ::= \varepsilon \mid AB$

prowadzącą do zdań ε, B, BB, BBB, ..., to naruszymy regułę 2, ponieważ

$pierw(A) = pierw(B)$

a zatem

$pierw(A) \cap nast(A) \neq \varnothing$

Te dwie reguły ograniczające nie pozwalają oczywiście na stosowanie definicji lewostronnie rekurencyjnych. Problem ten można prosto rozwiązać albo stosując prawostronną rekursję

$A ::= \varepsilon \mid BA$

albo rozszerzając notację BNF o operator iteracji. Wybierzemy to drugie rozwiązanie i przez $\{B\}$ będziemy oznaczać zbiór ciągów

ε, B, BB, BBB, ...

Oczywiście trzeba zdawać sobie sprawę z faktu, że każde użycie operatora iteracji jest związane z generowaniem ciągu pustego. (Nawiasy $\{$ oraz $\}$ są metasymbolami rozszerzonej notacji BNF).

Z poprzedniego rozumowania i z transformacji produkcji (5.5) na (5.5a) może się wydawać, że „sztuczka" z transformacją gramatyki stanowi środek uniwersalny, zapobiegający wszystkim problemom analizy składniowej. Musimy jednak pamiętać, że struktura zdania jest narzędziem pomocnym przy określaniu znaczenia zdania, a przy wyjaśnianiu znaczenia konstrukcji zdaniowej korzystamy zazwyczaj ze znaczenia składowych tej konstrukcji. Weźmy np. język wyrażeń budowanych z argumentów a, b, c i znaku minus, oznaczającego odejmowanie:

$S ::= A \mid S - A$
$A ::= a \mid b \mid c$

Zgodnie z gramatyką struktura zdania $a-b-c$ może być wyrażona przy użyciu nawiasów w sposób następujący: $((a-b)-c)$. Gdy jednak gramatyka zostanie przekształcona do postaci składniowo równoważnej z uniknięciem lewostronnej rekursji:

$S ::= A \mid A-S$
$A ::= a \mid b \mid c$

wówczas to samo zdanie uzyska inną strukturę, którą możemy opisać jako $(a-(b-c))$. Biorąc pod uwagę ogólnie przyjęte znaczenie odejmowania, widzimy, że te dwie postacie nie są wcale semantycznie równoważne.

Wniosek końcowy jest więc następujący: definiując struktury składniowe dla języka o określonym znaczeniu, należy być świadomym jego **struktur semantycznych**, ponieważ te pierwsze muszą stanowić odbicie tych drugich.

5.3. Konstrukcja diagramu składni

W poprzednim punkcie przedstawiliśmy generacyjny algorytm rozbioru. Stosuje się go dla gramatyk spełniających reguły ograniczające 1 i 2. Zajmiemy się teraz problemem przekształcenia tego algorytmu na konkretny program. Możemy zastosować jedną z dwóch istniejących, zasadniczo różnych metod. Pierwsza z nich to zaprojektowanie ogólnego programu rozbioru, odpowiedniego dla każdej gramatyki (spełniającej reguły 1 i 2). W tym przypadku konkretne gramatyki są zadawane w postaci pewnej struktury danych, na której działa program. Taki ogólny program jest w pewnym sensie sterowany strukturą danych; stąd jego nazwa **analizator sterowany składnią**. Druga metoda polega na opracowaniu programu rozbioru generacyjnego dla zadanego języka. Program taki konstruuje się, przekształcając systematycznie kolejne produkcje gramatyki na ciągi instrukcji, zgodnie z określonymi regułami. Obydwie metody mają swoje zalety i wady; obydwie będą dalej omówione. Przy opracowywaniu kompilatora dla zadanego języka programowania stosunkowo niewiele korzysta się z dużej elastyczności i zdolności do parametryzacji, właściwych analizatorowi ogólnemu, podczas gdy analizator dla konkretnego języka pozwala na ogół uzyskać bardziej efektywny i łatwiejszy w obsłudze system, a zatem jest lepszy. W obu przypadkach korzystne jest przedstawienie składni w postaci tzw. **diagramu składni**. Diagram taki odzwierciedla przebieg sterowania podczas rozbioru zdania.

Charakterystyczną cechą podejścia generacyjnego jest to, że podstawowy *cel* procesu rozbioru jest znany na początku. Celem tym jest rozpoznanie zdania, tj. wyprowadzenie ciągu symboli z symbolu początkowego. Stosowanie reguł produkcji, tzn. zastępowanie pojedynczego symbolu przez ciąg symboli, odpowiada rozbiciu pojedynczego celu na pewną liczbę celów częściowych (podcelów), które powinny być osiągnięte w określonym porządku. Stąd też

bierze się inna nazwa rozbioru generacyjnego – **rozbiór ukierunkowany celem** (ang. *goal-oriented parsing*). Przy konstrukcji analizatora możemy, korzystając z wzajemnej odpowiedniości symboli pomocniczych i celów, opracować jeden podanalizator dla każdego symbolu pomocniczego. Zadaniem takiego podanalizatora będzie rozpoznanie frazy zdaniowej dającej się wyprowadzić z odpowiadającego mu symbolu pomocniczego. Proces konstrukcji diagramu składni reprezentującego cały analizator rozbijemy także na kolejne podprocesy konstruowania poddiagramów dla poszczególnych symboli pomocniczych według reguł opisanych poniżej.

REGUŁY KONSTRUKCJI DIAGRAMU

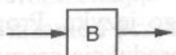
A1. Każdy symbol pomocniczy A, dla którego istnieje produkcja

$A ::= \xi_1 \mid \xi_2 \mid ... \mid \xi_n$

zostaje przekształcony na diagram symbolu A o strukturze określonej przez prawą stronę produkcji, zgodnie z regułami A2–A6.

A2. Każde wystąpienie symbolu *końcowego* x w ξ_i odpowiada instrukcji rozpoznającej ten symbol i pobierającej kolejny symbol z ciągu wejściowego. W diagramie jest ono reprezentowane przez łuk z etykietą x w kółeczku.

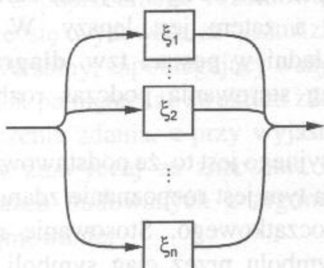
A3. Każde wystąpienie symbolu *pomocniczego* B w ξ_i odpowiada uaktywnieniu podprogramu opisanego diagramem dla symbolu B. W diagramie dla symbolu ξ_i jest ono reprezentowane przez łuk z etykietą B (w kwadracie).



A4. Produkcja postaci

$A ::= \xi_1 \mid ... \mid \xi_n$

jest przekształcana na diagram



gdzie każde $\boxed{\xi_i}$ jest utworzone przez stosowanie reguł A2–A6 dla ξ_i.

A5. Ciąg ξ postaci

$$\xi = \alpha_1\alpha_2 \dots \alpha_m$$

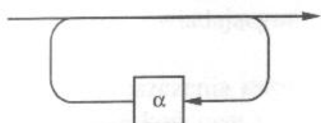
jest przekształcany na diagram

gdzie każde $\boxed{\alpha_i}$ powstaje z α_i przez stosowanie reguł A2–A6.

A6. Jeśli ξ jest postaci

$\xi = \{\alpha\}$

to jest ono przekształcane na diagram



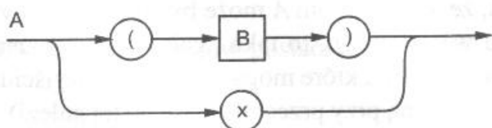
gdzie $\boxed{\alpha}$ powstaje z α przez stosowanie reguł A2–A6.

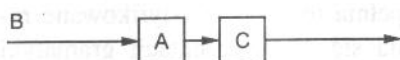
PRZYKŁAD 5

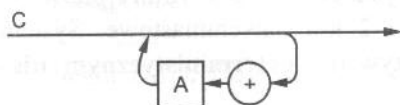
$$A ::= x \mid (B)$$
$$B ::= AC \tag{5.7}$$
$$C ::= \{+A\}$$

W gramatyce tej symbole $+$, x, (,) są symbolami końcowymi, podczas gdy nawiasy { oraz }, należące do rozszerzonej notacji BNF, są metasymbolami. Język gramatyki o symbolu początkowym A składa się z wyrażeń zbudowanych z argumentów x, operatora $+$ i nawiasów. Przykładami zdań są:

x
(x)
$(x+x)$
$((x))$
....



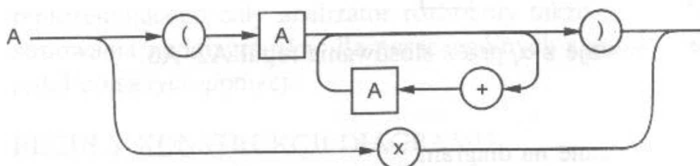




RYSUNEK 5.1
Diagramy składni dla przykładu 5

W wyniku zastosowania sześciu reguł konstrukcyjnych otrzymamy 3 diagramy przedstawione na rys. 5.1. Zauważmy, że można dokonać redukcji tego systemu diagramów do pojedynczego diagramu, stosując odpowiednie podstawienia na *C* w *B* i na *B* w *A* (zob. rys. 5.2).

□



RYSUNEK 5.2
Zredukowany diagram składni dla przykładu 5

Diagram rozbioru jest równoważną reprezentacją gramatyki języka i może być stosowany zamiast zbioru produkcji zapisanych w notacji BNF. W wielu (jeśli wręcz nie we wszystkich) zastosowaniach stanowi on wygodniejszą formę zapisu niż BNF. Daje mianowicie bardziej przejrzysty i zwięzły obraz struktury języka, a także pozwala bardziej bezpośrednio zrozumieć proces rozbioru. *Jest to postać opisu właściwa przy projektowaniu języka, zwłaszcza w fazie wstępnej*. Przykłady specyfikacji składni dla całych języków przedstawiono w p. 5.7 dla języka PL/0 i w dodatku B dla Pascala.

Reguły ograniczające 1 i 2 zostały wprowadzone w celu umożliwienia rozbioru deterministycznego na podstawie analizy z wyprzedzeniem tylko o jeden symbol. Jak te reguły wyglądają przy reprezentacji gramatyki w postaci diagramu? W tym przypadku przejrzystość diagramu staje się bardziej oczywista. I tak:

(1) Reguła 1 odpowiada wymaganiu, że w każdym punkcie rozgałęzienia o wyborze dalej śledzonej gałęzi decyduje pierwszy symbol tej gałęzi. Wynika z tego, że żadne dwie gałęzie nie mogą zaczynać się od tego samego symbolu.
(2) Reguła 2 odpowiada wymaganiu, że jeśli diagram *A* może być prześledzony bez wczytania żadnego symbolu wejściowego, to taka „gałąź pusta" musi być zaetykietowana wszystkimi symbolami, które mogą wystąpić po wyjściu z *A*. (Wpłynie to na decyzję podejmowaną przy przechodzeniu do tej gałęzi).

Sprawdzenie, czy system diagramów spełnia te dwie zmodyfikowane reguły, jest proste i nie wymaga odwoływania się do reprezentacji gramatyki w notacji BNF. Dla każdego diagramu *A* wystarczy określić zbiory *pierw* (*A*) i *nast* (*A*), po czym zastosowanie reguł 1 i 2 jest natychmiastowe. System diagramów spełniający te dwie reguły nazywamy **deterministycznym diagramem składni**.

5.4. Konstrukcja analizatora składniowego dla zadanej gramatyki

Na podstawie deterministycznego diagramu składni konkretnego języka (jeśli taki diagram w ogóle istnieje) możemy łatwo opracować program akceptujący zdania i dokonujący ich rozbioru. Diagram stanowi bowiem w istocie schemat blokowy programu. W trakcie tworzenia programu wskazane jest postępowanie dokładnie wg reguł przejścia podobnych do reguł, które prowadzą od notacji BNF do reprezentacji składni w postaci diagramu. Reguły przejścia od diagramu do programu są podane poniżej. Stosuje się je w pewnym środowisku programowym, składającym się z programu głównego z zanurzonymi w nim procedurami odpowiadającymi różnym podcelom i procedury pobierającej kolejny symbol.

W celu uproszczenia założymy, że analizowane zdanie znajduje się w pliku *input*, a symbole końcowe są pojedynczymi znakami. Przyjmiemy również, że wewnątrz omawianego środowiska istnieje zmienna *ch* reprezentująca kolejny, wczytany symbol. Przejście do następnego symbolu wyrazimy za pomocą instrukcji

read(*ch*)

Program główny składać się więc będzie z instrukcji początkowej, wczytującej pierwszy znak, oraz z instrukcji uaktywniającej proces analizy celu głównego. Poszczególne procedury odpowiadające analizie innych celów czy diagramów otrzymamy, stosując następujące reguły. (Instrukcję otrzymaną z diagramu *S* będziemy oznaczać przez *T*(*S*)).

REGUŁY PRZEJŚCIA OD DIAGRAMU DO PROGRAMU

B1. Zredukuj, na ile to możliwe, liczbę diagramów, stosując odpowiednie podstawienia.

B2. Każdy diagram zastąp deklaracją procedury zgodnie z regułami B3–B7.

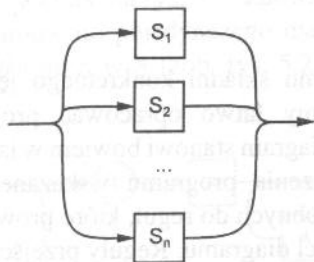
B3. *Ciąg* elementów postaci

zastąp instrukcją złożoną

---

begin $T(S_1)$; $T(S_2)$; ...; $T(S_n)$ **end**

B4. *Diagram alternatywny*



zastąp instrukcją wyboru

---

case *ch* **of**
L_1: $T(S_1)$;
L_2: $T(S_2)$;
....
L_n: $T(S_n)$
end

---

lub instrukcją warunkową

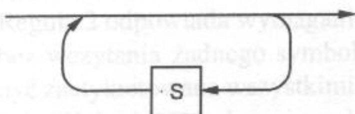
---

if *ch* **in** L_1 **then** $T(S_1)$ **else**
if *ch* **in** L_2 **then** $T(S_2)$ **else**
....
if *ch* **in** L_n **then** $T(S_n)$ **else**
błąd

---

gdzie L_i to zbiór symboli początkowych konstrukcji S_i ($L_i = pierw(S_i)$).
Uwaga: jeśli L_i składa się z pojedynczego symbolu *a*, to test „*ch* **in** L_i" powinien być zastąpiony przez „$ch = a$".

B5. Pętlę postaci



zastąp instrukcją

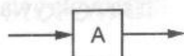
---

while *ch* **in** *L* **do** *T*(*S*)

---

gdzie *T*(*S*) otrzymano z *S* zgodnie z regułami B3–B7, a zbiór *L* jest równy *pierw*(*S*) (zob. uwagę do reguły B4).

B6. Element diagramu oznaczający inny diagram *A*



zastąp instrukcją wywołania procedury *A*.

B7. Element diagramu oznaczający symbol końcowy *x*

zastąp instrukcją

---

if *ch*=*x* **then** *read*(*ch*) **else** *błąd*

---

gdzie *błąd* jest procedurą wywołaną przy napotkaniu niepoprawnej konstrukcji.

Zastosowanie tych reguł zilustrujemy za pomocą programu analizatora (program 5.1) otrzymanego ze zredukowanego diagramu z przykładu 5 (rys. 5.2).

PROGRAM 5.1
Program analizatora dla gramatyki z przykładu 5

```
program analizator(input, output);
   var ch: char;
   procedure A;
   begin if ch='x' then read(ch) else
      if ch='(' then
      begin read(ch); A;
         while ch='+' do
            begin read(ch); A
            end;
         if ch=')' then read(ch) else błąd
      end else błąd
   end;
begin read(ch); A
end.
```

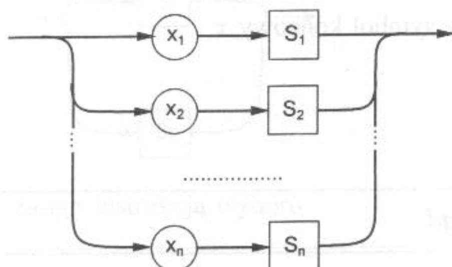
Dokonując przejścia od diagramu do powyższego programu, stosowaliśmy swobodnie pewne oczywiste reguły programowania pozwalające uprościć program. I tak czwarty wiersz programu w początkowej postaci wyglądał, jak następuje:

```
if ch='x' then
   if ch='x' then read(ch) else błąd
else ...
```

Podobne uproszczenia zastosowano dla wiersza piątego i siódmego.

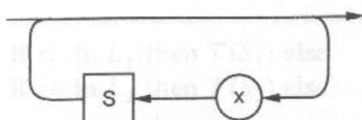
Zastanówmy się, kiedy takie uproszczenia są możliwe i jak je wyrazić bezpośrednio za pomocą diagramów. Dwa szczególne przypadki są wykrywane za pomocą następujących, dodatkowych reguł:

B4a



if $ch = 'x_1'$ **then begin** *read*(*ch*); $T(S_1)$ **end else**
if $ch = 'x_2'$ **then begin** *read*(*ch*); $T(S_2)$ **end else**
....
if $ch = 'x_n'$ **then begin** *read*(*ch*); $T(S_n)$ **end else** *błąd*

B5a



while $ch = 'x'$ **do**
begin *read*(*ch*); $T(S)$ **end**

Oprócz tego często występującą konstrukcję

read(*ch*); $T(S)$;
while B **do**
begin *read*(*ch*); $T(S)$ **end**

możemy wyrazić w krótszej postaci:

repeat *read*(*ch*); $T(S)$ **until** $\neg B$ (5.8)

Procedurę *błąd* rozmyślnie pozostawiliśmy do tej pory niewyspecyfikowaną. Ponieważ w tej chwili istotny jest dla nas fakt, czy zdanie wejściowe jest czy też nie jest poprawnie zbudowane, wywołanie tej procedury możemy sobie wyobrazić jako zakończenie działania programu. W praktyce, oczywiście, stosuje się bardziej wyszukane metody traktowania zdań niepoprawnych. Omówimy je w p. 5.9.

5.5. Konstrukcja programu analizatora sterowanego składnią

W poprzednim punkcie opisaliśmy reguły konstrukcji programu analizatora dla konkretnego języka. Inne rozwiązanie polega na opracowaniu ogólnego programu rozbioru. Dane dla tego programu będą składały się z opisu gramatyki języka i zbioru zdań do analizy. Działanie tego programu będzie polegało na stosowaniu reguł rozbioru generacyjnego. Jeżeli odpowiedni diagram składni jest deterministyczny, tzn. jeśli stosujemy analizę bez powrotów z wyprzedzeniem o jeden symbol, to konstrukcja analizatora ogólnego jest bardzo prosta.

Pierwsza część konstrukcji analizatora polega na przejściu od gramatyki reprezentowanej w postaci diagramów składni do odpowiedniej struktury danych [5.2]. Naturalna metoda reprezentacji diagramu polega na wprowadzeniu dla każdego symbolu jednego węzła struktury (rekordu) i powiązaniu tych węzłów za pomocą wskaźników. Reguły przejścia od diagramu do struktury danych podano poniżej. Są one dość oczywiste. Węzły struktury danych są rekordami z dwoma wariantami: jednym dla symbolu końcowego i jednym dla symbolu pomocniczego. Pierwszy wariant opisuje sam symbol końcowy, drugi zawiera wskaźnik do struktury danych reprezentującej odpowiedni symbol pomocniczy. Obydwa warianty zawierają ponadto po 2 wskaźniki: jeden określający *następnik* danego symbolu i drugi prowadzący do listy *członów alternatywy*.

Jeden element struktury można przedstawić graficznie w sposób następujący:

<table>
<tr><td colspan="2">sym</td></tr>
<tr><td>człalt</td><td>nast</td></tr>
</table>

dokładna zaś definicja odpowiedniego typu danych ma postać

```
type wskaźnik = ↑węzeł;
  węzeł =
  record nast, człalt: wskaźnik;
     case końcowy: boolean of                    (5.9)
        true: (ksym: char);
        false: (psym: poczwsk)
  end
```

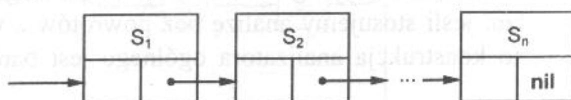
Potrzebny jest nam również jeden element do reprezentacji pustego ciągu symboli. Ciąg taki będziemy traktować jako element końcowy, a na jego oznaczenie wprowadzimy stałą *pusty*. Reguły przejścia od diagramu do struktury danych są analogiczne do reguł B1–B7.

REGUŁY PRZEJŚCIA OD DIAGRAMU DO STRUKTURY DANYCH

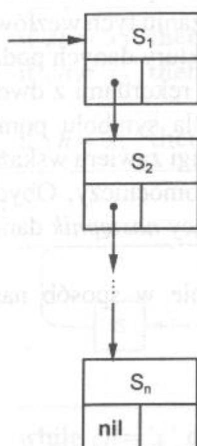
C1. Zredukuj, na ile to możliwe, liczbę diagramów, stosując odpowiednie podstawienia.

C2. Każdy diagram zastąp odpowiednią strukturą danych zgodnie z regułami C3–C7.

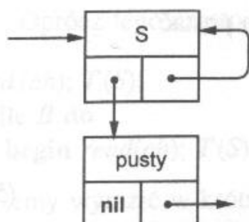
C3. Ciąg elementów (opisany rysunkiem dla reguły B3) zastąp następującą listą węzłów danych:



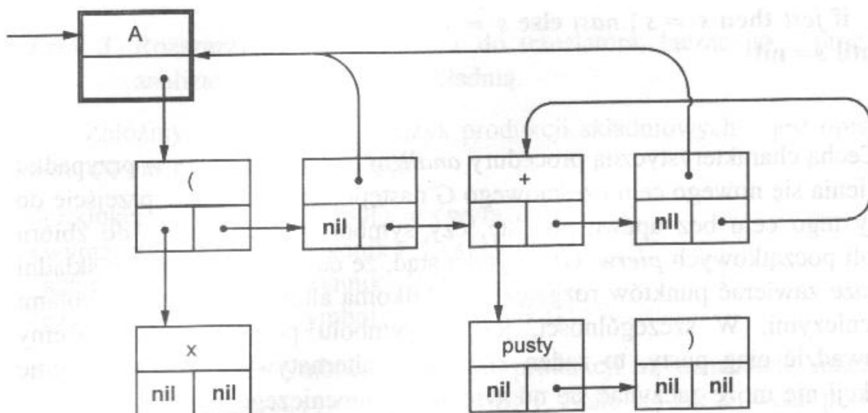
C4. Listę członów alternatywy (zob. rys. dla reguły B4) zastąp następującą strukturą danych:



C5. Pętlę (zob. rys. dla reguły B5) zastąp następującą strukturą:



Stosując powyższe reguły np. dla diagramu składni z przykładu 5 (rys. 5.2), otrzymamy strukturę pokazaną na rys. 5.3. Strukturę danych identyfikuje się za pomocą węzła *początkowego* zawierającego nazwę odpowiedniego symbolu pomocniczego (celu). Węzeł ten nie jest jednak niezbędny. Odwołania do niego w innych strukturach można zastąpić odwołaniami do elementu następnego. Węzłem początkowym posługujemy się głównie przy drukowaniu nazwy struktury.



RYSUNEK 5.3
Struktura danych reprezentująca diagram z rys. 5.2

$$\textbf{type}\ poczwsk = \uparrow początek;\quad początek = \textbf{record}\ wejście\colon wskaźnik;\ sym\colon char\ \textbf{end} \tag{5.10}$$

```
type poczwsk = ↑początek;
   początek =
      record wejście: wskaźnik                                  (5.10)
         sym: char
      end
```

Zdanie analizowane jest reprezentowane jako ciąg znaków w pliku *input*. Podstawową część programu analizującego zdanie stanowić będzie instrukcja iteracyjna opisująca przejście od jednego węzła struktury do następnego. Sam program będzie miał postać procedury opisującej interpretację diagramu; jeśli podczas interpretacji napotkamy węzeł reprezentujący symbol pomocniczy, to interpretacja diagramu dla tego symbolu musi poprzedzić zakończenie interpretacji aktualnego diagramu. Procedura interpretująca będzie więc wywoływana *rekurencyjnie*. Jeżeli bieżący symbol (*sym*) w pliku *input* jest taki sam jak symbol bieżącego węzła struktury, to pole *nast* określi nam kolejny węzeł diagramu; w przeciwnym przypadku przejdziemy do węzła wskazanego przez pole *człalt*.

```
procedure analizuj(cel: poczwsk; var jest: boolean);
  var s: wskaźnik;
begin s := cel↑.wejście;
  repeat
    if s↑.końcowy then
      begin if s↑.ksym = sym then
            begin jest := true; pobsym
            end                                                 (5.11)
        else jest := (s↑.ksym = pusty)
      end
    else analizuj(s↑.psym, jest);
```

```
        if jest then s := s↑.nast else s := s↑.czlalt
    until s = nil
end
```

Cechą charakterystyczną procedury *analizuj* (5.11) jest to, że w przypadku pojawienia się nowego celu częściowego *G* następuje bezpośrednie przejście do analizy tego celu bez upewnienia się, czy symbol bieżący należy do zbioru symboli początkowych *pierw* (*G*). Wynika stąd, że odpowiedni diagram składni nie może zawierać punktów rozgałęzień z kilkoma alternatywnymi symbolami pomocniczymi. W szczególności, jeśli z symbolu pomocniczego możemy wyprowadzić ciąg pusty, to żaden z członów alternatywy po prawej stronie produkcji nie może zaczynać się od symbolu pomocniczego.

Z programu (5.11) można łatwo wyprowadzić bardziej wymyślne analizatory sterowane składnią, działające na szerszej klasie gramatyk. Również nieznaczne tylko modyfikacje pozwoliłyby na wykonywanie przez ten algorytm analizy z powrotami. Prowadziłoby to jednak do znacznego zmniejszenia efektywności algorytmu.

Reprezentacja składni w postaci diagramu ma jedną zasadniczą wadę – komputer nie może czytać bezpośrednio diagramów. Tymczasem przed rozpoczęciem rozbioru należy w jakiś sposób zbudować strukturę danych sterującą analizatorem. Spośród różnych reprezentacji gramatyki najbardziej odpowiednią postać danych dla ogólnego programu rozbioru stanowi reprezentacja gramatyki w notacji BNF. Dlatego w następnym punkcie zajmiemy się problemem opracowania programu, który czyta produkcje w notacji BNF i przekształca je zgodnie z regułami B1–B7 na wewnętrzną strukturę danych, odpowiednią dla programu analizatora (5.11) (zob. [5.8]).

5.6. Translator z notacji BNF na struktury danych sterujące analizatorem

Translator akceptujący produkcje w notacji BNF i przekształcający ich reprezentacje jest autentycznym przykładem programu, dla którego dane wejściowe możemy traktować jako zdania w pewnym języku. W rzeczywistości bowiem notację BNF można uważać za język, którego składnię możemy oczywiście znów opisać za pomocą produkcji w notacji BNF. W konsekwencji translator ten może służyć jako kolejny przykład konstrukcji analizatora, rozszerzonego przez nas później do translatora, czyli – mówiąc ogólnie – programu przetwarzającego swoje dane wejściowe. Translator skonstruujemy w trzech kolejnych krokach:

Krok 1. Zdefiniujemy składnię metajęzyka zwanego RBNF (Rozszerzony BNF).
Krok 2. Skonstruujemy analizator dla RBNF zgodnie z regułami z p. 5.4.

Krok 3. Rozszerzymy ten analizator do translatora, łącząc go z programem analizatora sterowanego składnią.

Załóżmy, że metajęzyk – język produkcji składniowych – jest opisany za pomocą następujących produkcji:

⟨produkcja⟩ ::= ⟨symbol⟩ = ⟨wyrażenie⟩
⟨wyrażenie⟩ ::= ⟨składnik⟩ {,⟨składnik⟩}
⟨składnik⟩ ::= ⟨czynnik⟩ {⟨czynnik⟩}
⟨czynnik⟩ ::= ⟨symbol⟩ | [⟨składnik⟩]
(5.12)

Zauważmy, że metasymbole w języku produkcji są oznaczone inaczej niż odpowiadające im metasymbole notacji BNF. Stało się tak z dwóch powodów:

(1) aby rozróżnić metasymbole i symbole języka produkcji w (5.12);
(2) aby wprowadzić znaki powszechnie spotykane w sprzęcie komputerowym; w szczególności, aby zamiast symbolu ::= stosować pojedynczy znak =.

Wzajemną odpowiedniość między metasymbolami w notacji BNF i symbolami w RBNF pokazano w tabl. 5.1. Zakładamy dodatkowo, że każde zdanie języka produkcji jest zakończone kropką. Składnię języka z przykładu 5 możemy teraz zapisać w postaci trzech zdań języka RBNF:

$A = x, (B).$
$B = AC.$
$C = [+A].$
(5.13)

W celu uproszczenia konstruowanego translatora zakładamy ponadto, że symbole końcowe są *pojedynczymi literami* oraz że każda produkcja jest napisana w oddzielnym wierszu. Pozwala to na zwiększenie czytelności danych wejściowych przez odpowiednie stosowanie odstępów, ignorowanych przez translator. Instrukcję *read*(*ch*) w regule B7 zastępujemy wywołaniem procedury pobierającej z wejścia kolejny, znaczący symbol. Procedura ta jest bardzo prostym odpowiednikiem *analizatora leksykalnego*. Zadaniem analizatora leksykalnego jest określenie i pobranie kolejnego *symbolu*, reprezentowanego w ciągu wejściowym przez grupę *znaków*, zgodnie z regułami reprezentacji języka. U nas symbole pokrywają się ze znakami, jest to jednak specjalny przypadek, rzadko spotykany w praktyce.

TABLICA 5.1
Odpowiedniość między metasymbolami i symbolami

Metasymbole BNF	Symble RBNF
::=	=
|	,
{	[
}	]

Jako ostatnie ustalenie wprowadzimy zasadę, że symbole pomocnicze będą reprezentowane za pomocą liter od A do H, symbole końcowe zaś – za pomocą liter od I do Z. Jest to ustalenie przyjęte wyłącznie dla wygody, bez głębszego znaczenia. Pozwoli ono uniknąć tworzenia słownika symboli końcowych i słownika symboli pomocniczych.

Postępując teraz dokładnie wg reguł konstrukcji B1–B7, po upewnieniu się, że składnia (5.12) spełnia reguły ograniczające 1 i 2, otrzymamy w wyniku program 5.2 będący analizatorem języka opisanego przez produkcje (5.12). Zauważmy, że analizator leksykalny nosi w programie nazwę *pobsym*.

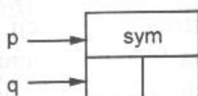
PROGRAM 5.2
Analizator języka (5.12)

```
program analizator (input, output);
label 99;
const pusty='*';
var sym: char;

procedure pobsym;
begin
   repeat read (sym); write (sym) until sym ≠ ' '
end {pobsym};

procedure błąd;
begin writeln;
   writeln('NIEPOPRAWNY INPUT'); goto 99
end {błąd};

procedure składnik;
   procedure czynnik;
   begin
      if sym in ['A'..'Z', pusty] then pobsym else
      if sym='[' then
      begin pobsym; składnik;
         if sym=']' then pobsym else błąd
      end else błąd
   end {czynnik};
begin czynnik;
   while sym in ['A'..'Z', '[', pusty] do czynnik
end {składnik};

procedure wyrażenie;
begin składnik;
   while sym=',' do
```

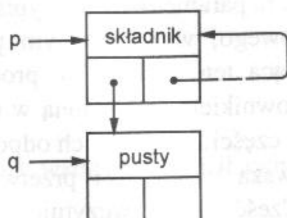
```
   begin pobsym; składnik
   end
end {wyrażenie};

begin {program główny}
   while ¬ eof(input) do
   begin pobsym;
      if sym in ['A'..'Z'] then pobsym else błąd;
      if sym = '=' then pobsym else błąd;
      wyrażenie;
      if sym ≠ '.' then błąd;
      writeln; readln;
   end;
99: end.
```

Czynniki:

1. <symbol>



2. [<składnik>]



Składniki:

<czynnik–1> . . . <czynnik–n>

Wyrażenia:

<składnik–1> <składnik–2> . . . <składnik–n>

Ostatni krok przy opracowywaniu translatora jest związany z konstrukcją struktury danych reprezentującej wczytane produkcje w notacji BNF i przetwarzanej przez procedurę analizującą składnię (5.11). Niestety, krok ten nie daje się sformalizować tak łatwo jak poprzedni. Dlatego zrezygnujemy z podejścia formalnego i odpowiednie struktury opiszemy za pomocą rysunków. Struktury te są przekazywane jako parametry wynikowe odpowiednich procedur rozbioru, uprzednio „rozszerzonych" do procedur translacji. Wartościami parametrów aktualnych będą wskaźniki *p*, *q*, *r*, identyfikujące różne fragmenty utworzonych struktur, tak jak to pokazano na rysunku.

Zadaniem procedury *czynnik* jest tworzenie nowych elementów struktury danych. Procedury *składnik* i *wyrażenie* łączą te elementy w listę, przy czym *składnik* używa pola *nast*, a *wyrażenie* pola *człalt*. Pozostałe szczegóły związane z tworzeniem struktury danych można znaleźć w programie 5.3.

Sposób przetwarzania symboli pomocniczych wymaga dodatkowego wyjaśnienia. Może się zdarzyć, że symbol pomocniczy pojawi się jako czynnik, zanim wystąpi jako symbol pomocniczy po lewej stronie produkcji. Struktury opisujące symbole pomocnicze są powiązane w jedną listę utworzoną z węzłów początkowych struktur. Procedura *znajdź* (*sym*, *h*) służy do umiejscowienia symbolu *sym* w tej liście. Jeżeli się go umiejscowi, to parametrowi *h* przypisuje się wskaźnik do odpowiedniego elementu początkowego; w przeciwnym przypadku do listy zostaje dołączona struktura opisująca ten symbol. W procedurze *znajdź* zastosowano metodę szukania z wartownikiem, omówioną w rozdz. 4.

Program 5.3 składa się z trzech części; każda z nich odpowiada jednej sekcji danych wejściowych. Część pierwsza wczytuje i przetwarza *produkcje* na odpowiednie struktury danych. Część druga wczytuje pojedynczy symbol oznaczający *symbol początkowy* gramatyki (dane dla części pierwszej i drugiej są rozdzielone znakiem $). Część trzecią stanowi procedura *analizuj* (5.11). Wczytuje ona kolejne *zdania* i analizuje je na podstawie struktur danych, utworzonych w pierwszej części programu 5.3.

Warto zauważyć, że program 5.3 utworzono, wstawiając jedynie dodatkowe instrukcje do *niezmienionego* programu 5.2. Otrzymany program analizuje tylko zdania poprawne i może być użyty jako szkielet do rozszerzonego programu, który by nie tylko analizował, ale również przetwarzał zaakceptowane zdania lub tłumaczył je na inną postać. Tworzenie analizatorów i translatorów poprzez takie *stopniowe precyzowanie* czy raczej *stopniowe wzbogacanie* programu jest metodą szczególnie zalecaną. Pozwala ona projektantowi programu na skupienie uwagi na kolejno wybranych aspektach języka. Ułatwia więc sprawdzenie poprawności programu, a co najmniej pozwala uzyskać w trakcie opracowywania programu stosunkowo duże zaufanie co do poprawności jego działania. Omawiany program jest stosunkowo prosty i tworząc go, wyróżniliśmy tylko dwa kroki wzbogacenia. W przypadku bardziej złożonych języków czy też bardziej skomplikowanego schematu translacji liczba pojedynczych kroków wzbogacania jest znacznie większa. Bardzo podobny sposób tworzenia programu w trzech kolejnych krokach przedstawimy w p. 5.8–5.11.

PROGRAM 5.3
Translator języka (5.12)

```
program analizatorogólny (input, output);
label 99;
const pusty='*';
type wskaźnik=↑węzeł;
    poczwsk=↑początek;
    węzeł=record nast, czlalt: wskaźnik;
                case końcowy: boolean of
                    true: (ksym: char);
                    false: (psym: poczwsk)
            end;
    początek=record sym: char;
                    wejście: wskaźnik;
                    nast: poczwsk
                end;
var lista, wartownik, h: poczwsk;
    p: wskaźnik;
    sym: char;
    ok: boolean;

procedure pobsym;
begin
    repeat read (sym); write (sym) until sym ≠' '
end {pobsym};

procedure znajdź (s: char; var h: poczwsk);
{zlokalizuj na liście symbol pomocniczy s; jeśli nie występuje, to go dodaj}
    var h1: poczwsk;
begin h1 := lista; wartownik↑.sym := s;
    while h1↑.sym≠s do h1 := h1↑.nast;
    if h1=wartownik then
        begin {wstaw} new (wartownik);
            h1↑.nast := wartownik; h1↑.wejście := nil
        end;
    h := h1
end {znajdź};

procedure błąd;
begin writeln;
    writeln('NIEPOPRAWNA SKŁADNIA'); goto 99
end {błąd};

procedure składnik (var p, q, r: wskaźnik);
    var a, b, c: wskaźnik;
```

```
   procedure czynnik(var p, q: wskaźnik);
      var a, b: wskaźnik; h: poczwsk;

   begin if sym in ['A'..'Z', pusty] then
      begin {symbol} new (a);
         if sym in ['A'..'H'] then
         begin {pomocniczy} znajdź (sym, h);
            a↑.końcowy := false; a↑.psym := h;
         end else
         begin {końcowy}
            a↑.końcowy := true; a↑.ksym := sym
         end;
         p := a; q := a; pobsym
      end else
      if sym='[' then
      begin pobsym; składnik(p, a, b); b↑.nast := p;
         new (b); b↑.końcowy := true; b↑.ksym := pusty;
         a↑.człalt := b; q := b;
         if sym=']' then pobsym else błąd
      end else błąd
   end {czynnik};
begin czynnik(p, a); q := a;
   while sym in ['A'..'Z'], '[', pusty] do
   begin czynnik (a↑.nast, b); b↑.człalt := nil; a := b
   end;
   r := a
end {składnik};

procedure wyrażenie (var p, q: wskaźnik);
   var a, b, c: wskaźnik;
begin składnik(p, a, c); c↑.nast := nil;
   while sym=',' do
   begin pobsym;
      składnik(a↑.człalt, b, c); c↑.nast := nil; a := b
   end;
   q := a
end {wyrażenie};

procedure analizuj(cel: poczwsk; var jest: boolean);
   var s: wskaźnik;
begin s := cel↑.wejście;
   repeat
      if s↑.końcowy then
```

```
    begin if s↑.ksym=sym then
                begin jest := true; pobsym
                end
            else jest := (s↑.ksym=pusty)
    end
    else analizuj(s↑.psym, jest);
    if jest then s := s↑.nast else s := s↑.członalt
  until s=nil
end {analizuj};

begin {produkcje}
  pobsym; new(wartownik); lista := wartownik;
  while sym ≠ '$' do
  begin znajdź(sym, h);
    pobsym; if sym='=' then pobsym else błąd;
    wyrażenie(h↑.wejście, p); p↑.członalt := nil;
    if sym='.' then błąd;
    writeln; readln; pobsym
  end;
  {sprawdź, czy wszystkie symbole są zdefiniowane}
  h := lista; ok := true;
  while h≠wartownik do
  begin if h↑.wejście=nil then
        begin writeln('SYMBOL NIEZDEFINIOWANY', h↑.sym);
              ok := false
        end;
    h := h↑.nast
  end;
  if ¬ ok then goto 99;
  {symbol początkowy gramatyki}
  pobsym; znajdź(sym, h); readln; writeln;
  {zdania}
  while ¬ eof(input) do
    begin write(' '); pobsym; analizuj(h, ok);
      if ok ∧ (sym = '.') then writeln ('POPRAWNE')
                          else writeln('NIEPOPRAWNE');
      readln
    end;
99: end.
```

Jak dowodzi opracowanie programu 5.3, **analizę składniową sterowaną strukturą danych** cechuje swoboda i elastyczność nie spotykana w przypadku analizatora opracowywanego dla konkretnego języka. Na ogół nie wymaga się tak dużego stopnia elastyczności; jest on jednak niezbędny przy opracowywaniu

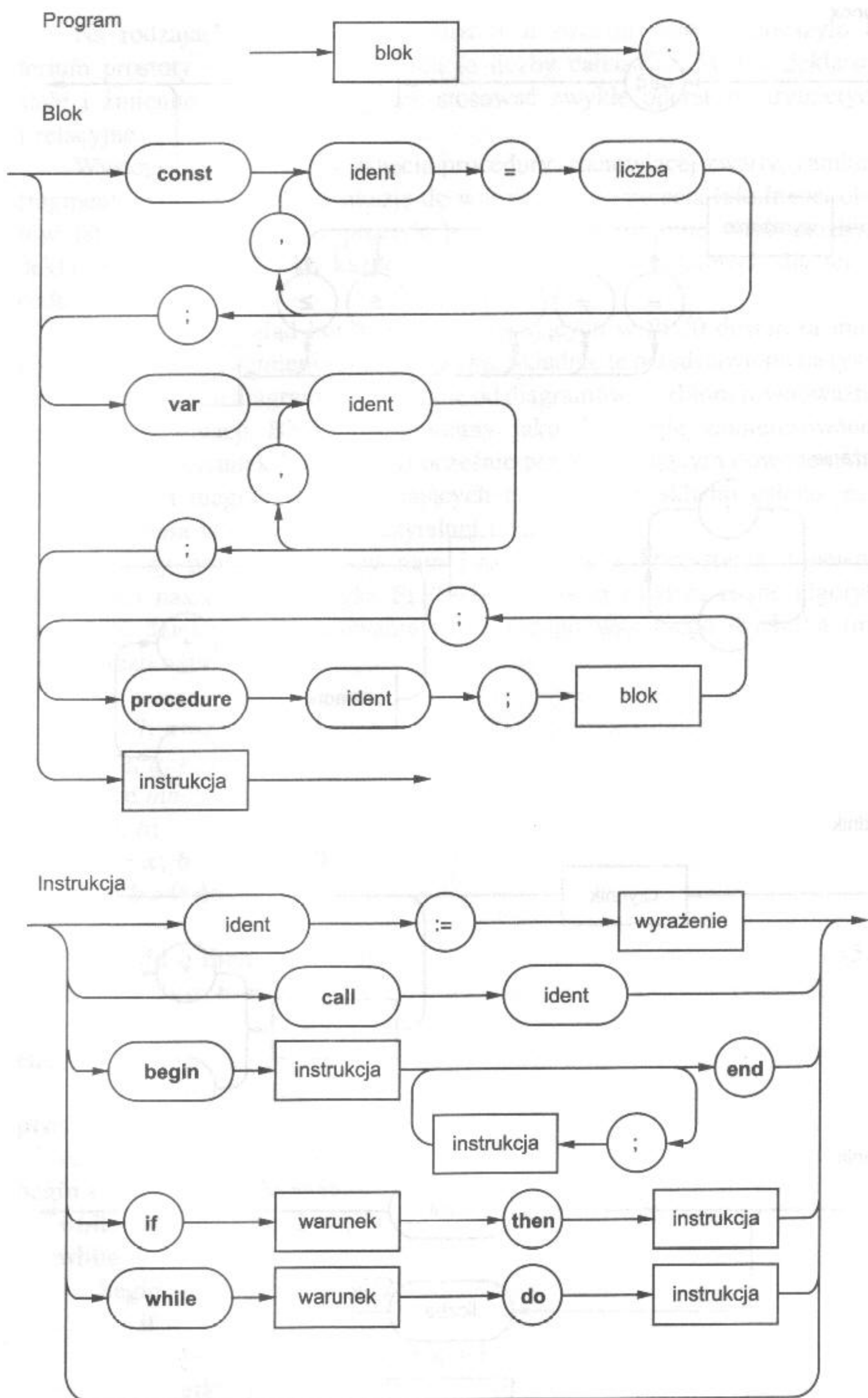
kompilatorów dla tzw. **języków rozszerzalnych**. Podstawową cechą języków rozszerzalnych jest możliwość dodawania do języka nowych konstrukcji składniowych wg uznania programisty. Podobnie jak dla programu 5.3, dane dla kompilatora języka rozszerzalnego składają się ze specyfikacji rozszerzeń języka i z samego programu, w którym użyto tych rozszerzeń. Kompilatory o bardziej ambitnym schemacie działania pozwalają nawet na zmienianie języka podczas tłumaczenia programu – przez przeplatanie części programu z sekcjami nowych specyfikacji.

Idea języka rozszerzalnego wywołała w swoim czasie spore zainteresowanie. Wysiłkom związanym z realizacją kompilatorów takich języków nie towarzyszyły jednak wyraźne sukcesy. Podstawowym powodem tego był fakt, że zagadnienia związane ze składnią i analizą zdań stanowią jedynie część problemów translacji i to, w gruncie rzeczy, część stosunkowo niewielką. Jest to również część dająca się najłatwiej sformalizować, co m.in. pozwala na reprezentację składni języka w postaci dość regularnej struktury danych. Znacznie większe trudności napotykamy przy próbie formalizacji *semantyki* języka, przy czym – na nasze potrzeby – semantykę rozumiemy jako wynik translacji. Problem formalizacji znaczenia nie został do tej pory rozwiązany w stopniu choćby zbliżonym do zadawalającego; może to tłumaczyć, dlaczego projektanci kompilatorów ze znacznie mniejszym entuzjazmem odnoszą się obecnie do takich języków. Końcowe punkty tego rozdziału poświęcimy opracowaniu prostego kompilatora dla niedużego, specyficznego języka programowania.

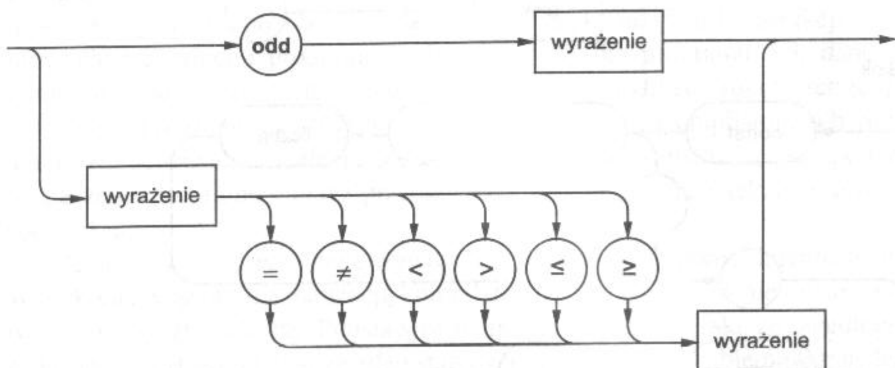
5.7. Język programowania PL/0

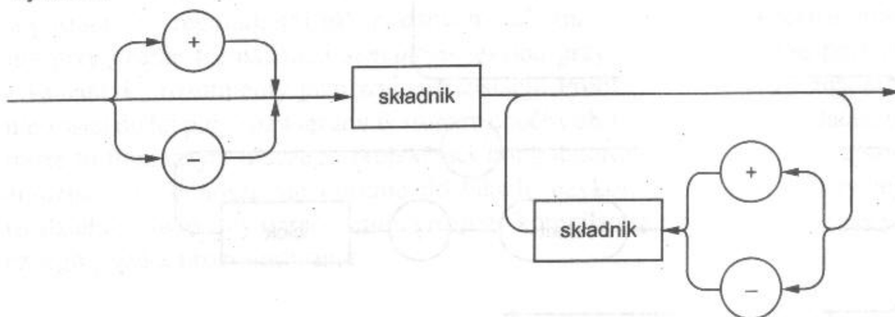
W pozostałych punktach tego rozdziału zajmiemy się opracowaniem kompilatora dla języka o nazwie PL/0. Język PL/0 został zaprojektowany, aby z jednej strony, można było na jego przykładzie zademonstrować najbardziej podstawowe cechy kompilacji języków wysokiego poziomu, z drugiej zaś strony, aby jego kompilator był niewielki. Pozwoliłoby to umieścić w tej książce pełny program kompilatora. Bez wątpienia można by wybrać język prostszy lub też bardziej złożony; PL/0 stanowi jeden z możliwych kompromisów między prostotą – pozwalającą poprowadzić wykład w przejrzysty sposób a dostateczną złożonością – dzięki której uzyskujemy wartościowy projekt kompilatora. Język programowania Pascal jest znacznie bardziej skomplikowany, chociaż do opracowania jego kompilatora użyto tych samych metod. Składnię Pascala opisano w dodatku B.

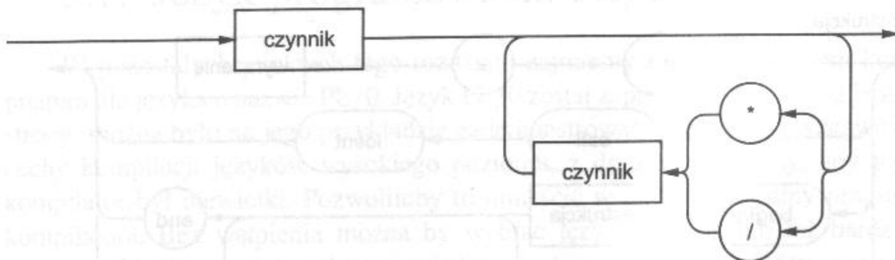
Pod względem struktur programowych PL/0 stanowi pewną całość. Podstawową strukturą jest oczywiście instrukcja przypisania. Schematy strukturalizacji takie, jak ciąg, wykonanie warunkowe oraz iteracja, prowadzą do znanych nam instrukcji: złożonej **begin/end**, jeśli **if** i dopóki **while**. W języku PL/0 istnieje również pojęcie podprogramu i w związku z tym występują w nim: deklaracja procedury oraz instrukcja wywołania procedury.

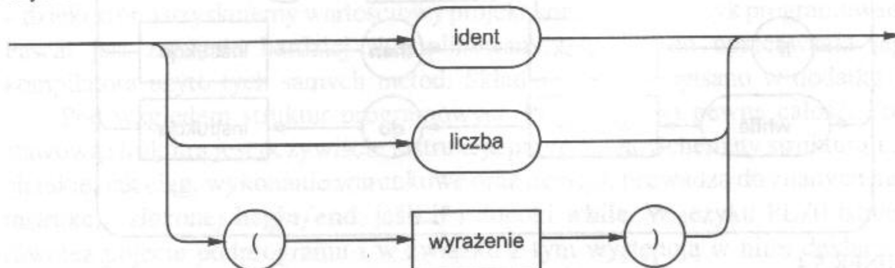


RYSUNEK 5.4
Składnia języka PL/0









RYSUNEK 5.4 (cd.)

Na rodzajach i liczbie wprowadzonych struktur danych zaważyło kryterium prostoty – jedyny typ danych to liczby całkowite. Można deklarować stałe i zmienne tego typu, a także stosować zwykłe operatory arytmetyczne i relacyjne.

Występowanie w języku pojęcia procedury, stanowiącej zwarty, zamknięty fragment programu, stwarza okazję do wprowadzenia pojęcia **lokalności** obiektów (stałych, zmiennych i procedur). Język PL/0 cechuje więc możliwość deklarowania w nagłówku każdej procedury obiektów lokalnych dla tej procedury.

Ten krótki przegląd konstrukcji występujących w PL/0 dostarcza intuicji koniecznych do zrozumienia składni języka. Składnię tę przedstawiono na rys. 5.4 za pomocą siedmiu diagramów. Przejście od diagramów do zbioru równoważnych produkcji w notacji BNF pozostawiamy jako ćwiczenie zainteresowanemu czytelnikowi. Rysunek 5.4 jest równocześnie przekonywającym dowodem mocy opisowej tych diagramów pozwalających na zapisanie składni całego języka programowania w tak zwartej i czytelnej postaci.

Poniższy program posłuży nam jako ilustracja korzystania z pewnych możliwości naszego minijęzyka PL/0. Program ten zawiera znane algorytmy mnożenia, dzielenia i znajdowania największego wspólnego dzielnika (nwd) dwóch liczb naturalnych.

```
const m=7, n=85;
var x, y, z, q, r;
procedure mnóż;
   var a, b;
begin a := x; b := y; z := 0;
   while b>0 do
   begin
      if odd b then z := z+a;                          (5.14)
      a := 2*a; b := b/2;
   end
end;

procedure dziel;
   var w;
begin r := x; q := 0; w := y;
   while w<=r do w := 2*w;
   while w>y do
      begin q := 2*q; w := w/2;                        (5.15)
         if w<=r then
            begin r := r-w; q := q+1
            end
      end
end;
```

```
procedure nwd;
   var f, g;
begin f := x; g := y;
   while f≠g do
      begin if f<g then g := g−f;
            if g<f then f := f−g;                    (5.16)
      end;
   z := f
end;

begin
   x := m; y := n; call mnóż;
   x := 25; y := 3; call dziel;
   x := 84; y := 36; call nwd;
end.
```

5.8. Analizator składniowy dla języka PL/0

Konstruowanie kompilatora języka PL/0 rozpoczniemy od opracowania analizatora składniowego. Zadanie to nie sprawi większych kłopotów, jeśli zastosujemy reguły konstrukcji B1–B7 omówione w p. 5.4. Przed ich użyciem należy jedynie sprawdzić, czy składnia języka spełnia reguły ograniczające 1 i 2. Przy sprawdzaniu skorzystamy ze sformułowania tych reguł dla diagramów składni.

Reguła 1 mówi, że każda gałąź wychodząca z punktu rozgałęzienia musi się zaczynać od innego symbolu. Weryfikacja tej reguły dla diagramów składni z rys. 5.4 nie stwarza większych kłopotów. Reguła 2 stosuje się do wszystkich diagramów, przez które można przejść bez konieczności wczytania jakiegokolwiek symbolu. Jedynym takim diagramem ze składni PL/0 jest diagram opisujący instrukcje. Reguła 2 żąda, aby wszystkie symbole, które mogą następować po instrukcji, były różne od symboli rozpoczynających instrukcję. Ponieważ w przyszłości będziemy korzystać z wartości zbiorów *pierw* i *nast* dla wszystkich diagramów, określimy te zbiory dla wszystkich siedmiu symboli pomocniczych (diagramów) w gramatyce PL/0 (z wyjątkiem symbolu „program"). Z tablicy 5.2 wynika, że zbiory *pierw* i *nast* dla symbolu „instrukcja" są rozłączne, możemy więc stosować reguły konstrukcyjne B1–B7.

Uważny czytelnik spostrzeże, że w odróżnieniu od poprzednich przykładów symbole podstawowe języka PL/0 nie są pojedynczymi znakami. Symbole te są ciągami znaków, np. BEGIN czy := . Podobnie jak w programie 5.3, zadaniem tzw. analizatora leksykalnego będzie podział wejściowego ciągu znaków na symbole podstawowe (leksykalne). Analizator leksykalny zaprogramowano jako procedurę *pobsym* dostarczającą następny symbol. Czynności wykonywane przez analizator leksykalny możemy podzielić na:

TABLICA 5.2
Zbiory *pier* i *nast* dla symboli pomocniczych gramatyki PL/0

Symbol pomocniczy *S*	*pierw* (*S*)	*nast* (*S*)
Blok	**const var procedure** *ident* **if call begin while**	. ;
Instrukcja	*ident* **call begin if while**	. ; **end**
Warunek	**odd** + − (*ident liczba*	**then do**
Wyrażenie	+ − (*ident liczba*	. ;) R **end then do**
Składnik	*ident liczba* (	. ;) R + − **end then do**
Czynnik	*ident liczba* (	. ;) R + − * / **end then do**

(1) opuszczanie separatorów (odstępów);
(2) rozpoznawanie słów zastrzeżonych, takich jak BEGIN, END itd.;
(3) rozpoznawanie identyfikatorów nie będących słowami zastrzeżonymi; faktyczny identyfikator jest przypisany zmiennej *id*;
(4) rozpoznawanie ciągów cyfr jako liczb; aktualna wartość jest przypisana zmiennej *num*;
(5) rozpoznawanie par znaków specjalnych, takich jak np. :=.

Procedura *pobsym* używa lokalnej procedury *pobznak*, której zadaniem jest pobranie następnego znaku. Oprócz tego procedura *pobznak*:

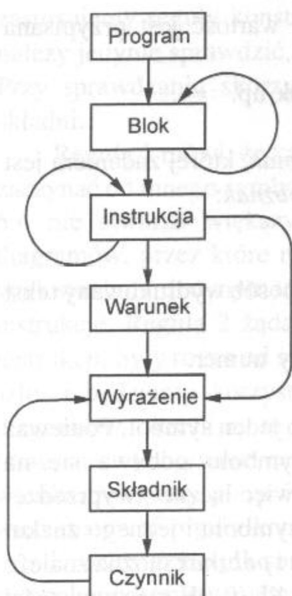
(1) rozpoznaje i przetwarza koniec wiersza znaków;
(2) kopiuje plik *input* do pliku *output*; powstaje w ten sposób wydrukowany tekst programu;
(3) na początku każdego wiersza drukuje jego kolejny numer.

Analizator składniowy „czyta z wyprzedzeniem" o jeden symbol. Ponieważ określenie przez analizator leksykalny kolejnego symbolu odbywa się na podstawie „czytania z wyprzedzeniem" o jeden znak, więc łączne „wyprzedzenie" stosowane przez kompilator składa się z jednego symbolu i jednego znaku.

Szczegóły związane z działaniem procedur *pobsym* i *pobznak* można znaleźć w programie 5.4 stanowiącym pełny analizator języka PL/0. W rzeczywistości program analizatora został już rozszerzony o pewne dodatkowe czynności, mianowicie umieszcza on wszystkie identyfikatory oznaczające stałe, zmienne i procedury we wspólnej *tablicy*. Wystąpienie identyfikatora wewnątrz instrukcji powoduje przejrzenie tej tablicy w celu określenia, czy identyfikator został poprawnie zadeklarowany. Brak odpowiedniej deklaracji możemy traktować jako

błąd składniowy, ponieważ formalnie jest to błąd popełniony podczas tworzenia tekstu programu i spowodowany użyciem „nielegalnego" symbolu. Fakt, że błąd ten może być wykryty tylko na podstawie informacji zawartej w tablicy, jest konsekwencją istotnych *zależności kontekstowych* tkwiących wewnątrz języka. Chodzi tu mianowicie o regułę wiążącą deklarację identyfikatora ze sposobem jego użycia. Ten rodzaj zależności kontekstowej występuje praktycznie we wszystkich językach programowania. Pomimo tego faktu składnia bezkontekstowa jest bardzo pomocnym modelem do opisu takich języków i znacznie ułatwia systematyczną konstrukcję analizatora. Analizator tworzy szkielet programu, który łatwo możemy rozszerzyć, uwzględniając zależności kontekstowe języka. Przykładem takiego rozszerzenia jest tablica identyfikatorów wprowadzona do omawianego przez nas analizatora.

Zanim przejdziemy do konstrukcji poszczególnych procedur analizatora odpowiadających kolejnym diagramom składni, warto określić, w jaki sposób te diagramy są od siebie zależne. Zależności te możemy przedstawić za pomocą tzw. **diagramu zależności**. W diagramie zależności dla każdego diagramu składni G są zaznaczone wszystkie diagramy składni $G_1, ..., G_n$, na podstawie których zdefiniowano G. Innymi słowy, diagram zależności określa, które procedury mogą być wywołane przez daną procedurę. Diagram zależności dla języka PL/0 pokazano na rys. 5.5.



RYSUNEK 5.5
Diagram zależności dla języka PL/0

Pętle na rysunku 5.5 wskazują na wystąpienie rekursji. Istotne jest więc, aby język, w którym zapiszemy kompilator PL/0, dopuszczał rekurencyjne wywołania procedur. Ponadto diagram zależności pozwala nam wyciągnąć wnioski o hierarchicznej organizacji programu analizatora. Na przykład wszystkie

procedury mogą być lokalne dla procedury analizującej konstrukcję zdaniową ⟨program⟩ (dlatego procedura ta stanowi główną część programu analizatora). Poza tym wszystkie procedury występujące w diagramie poniżej procedury *blok* mogą być lokalne dla procedury analizującej konstrukcję ⟨blok⟩. Oczywiście wszystkie te procedury wywołują procedurę *pobsym*, a ta z kolei wywołuje procedurę *pobznak*.

PROGRAM 5.4
Analizator języka PL/0

```
program PL0 (input, output);
{kompilator języka PL/0, tylko analiza składniowa}
label 99;
const lsz=11;     {liczba słów zastrzeżonych}
   tmax=100;      {długość tablicy identyfikatorów}
   cmax=14;       {maksymalna liczba cyfr w liczbach}
   al=10;         {długość identyfikatorów}
type symbol=
   (żaden, ident, liczba, plus, minus, razy, dziel, oddsym, równe, różne, mniejsze,
   mnrówne, większe, wrówne, lnaw, pnaw, przec, średnik, kropka, przypis,
   beginsym, endsym, ifsym, thensym, whilesym, dosym, callsym, constsym,
   varsym, procsym);
   alfa=packed array [1..al] of char;
   obiekt=(stała, zmienna, procedura);
var ch: char;          {ostatni wczytany znak}
   sym: symbol;        {ostatni wczytany symbol}
   id: alfa;           {ostatni wczytany identyfikator}
   num: integer;       {ostatnia wczytana liczba}
   lz: integer;        {licznik znaków}
   dw: integer;        {długość wiersza}
   kk: integer;
   wiersz: array [1..81] of char;
   a: alfa;
   słowo: array [1..lsz] of alfa;
   zsym: array [1..lsz] of symbol;
   ssym: array [char] of symbol;
   tablica: array [0..tmax] of
         record nazwa: alfa;
            rodzaj: obiekt
         end;

procedure błąd(n: integer);
begin writeln('  ': lz, '↑', n: 2); goto 99
end {błąd};
```

```
procedure pobsym;
  var i, j, k: integer;
  procedure pobznak;
  begin if lz=dl then
    begin if eof(input) then
        begin write('PROGRAM NIEDOKOŃCZONY'); goto 99
        end;
      dw := 0; lz := 0; write(' ');
      while ¬ eoln(input) do
        begin dw := dw+1; read(ch); write(ch); wiersz[dw] := ch
        end;
      writeln; dw := dw+1; read(wiersz[dw])
    end;
    lz := lz+1; ch := wiersz[lz]
  end {pobznak};
begin {pobsym}
  while ch=' ' do pobznak;
  if ch in ['A'..'Z'] then
  begin {identyfikator lub słowo zastrzeżone} k := 0;
    repeat if k<al then
      begin k := k+1; a[k] := ch
      end;
      pobznak
    until ¬ (ch in ['A'..'Z', '0'..'9']);
    if k≥kk=  then kk := k else
      repeat a[kk] := ' '; kk := kk-1
      until kk=k;
    id := a; i := 1; j := lsz;
    repeat k := (i+j) div 2;
      if id≤słowo[k] then j := k-1;
      if id≥słowo[k] then i := k+1
    until i>j;
    if i-1>j then sym := zsym[k] else sym := ident
  end else
  if ch in ['0'..'9'] then
  begin {liczba} k := 0; num := 0; sym := liczba;
    repeat num := 10*num+(ord(ch)-ord('0'));
      k := k+1; pobznak
    until ¬ (ch in ['0'..'9']);
    if k>cmax then błąd(30)
  end else
  if ch=':' then
  begin pobznak;
    if ch='=' then
```

```
        begin sym := przypis; pobznak
        end else sym := żaden;
    end else
    begin sym := ssym[ch]; pobznak
    end
end {pobsym};

procedure blok(tx: integer);
    procedure wprowadź(k: obiekt);
    begin {wprowadź obiekt do tablicy}
        tx := tx+1;
        with tablica[tx] do
        begin nazwa := id; rodzaj := k;
        end
    end {wprowadź};

    function pozycja(id: alfa): integer;
        var i: integer;
    begin {znajdź identyfikator id w tablicy}
        tablica[0].nazwa := id; i := tx;
        while tablica[i].nazwa ≠ id do i := i-1;
        pozycja := i
    end {pozycja};

    procedure deklstałej;
    begin if sym=ident then
        begin pobsym;
            if sym=równe then
            begin pobsym;
                if sym=liczba then
                    begin wprowadź(stała); pobsym
                    end
                else błąd(2)
            end else błąd(3)
        end else błąd(4)
    end {deklstałej};

    procedure deklzmiennej;
    begin if sym = ident then
        begin wprowadź(zmienna); pobsym
        end else błąd(4)
    end {deklzmiennej};

    procedura instrukcja;
        var i: integer;
```

```
procedure wyrażenie;
    procedure składnik;
        procedure czynnik;
            var i: integer;
        begin
            if sym=ident then
            begin i := pozycja(id);
                if i = 0 then błąd(11) else
                if tablica[i].rodzaj=procedura then błąd(21);
                pobsym
            end else
            if sym=liczba then
            begin pobsym
            end else
            if sym=lnaw then
            begin pobsym; wyrażenie;
                if sym=pnaw then pobsym else błąd(22)
            end
            else błąd(23)
        end {czynnik};
    begin {składnik} czynnik;
        while sym in [razy, dziel] do
            begin pobsym; składnik
            end
    end {składnik};
begin {wyrażenie}
    if sym in [plus, minus] then
        begin pobsym; składnik
        end else składnik;
    while sym in [plus, minus] do
        begin pobsym; składnik
        end
end {wyrażenie};

procedure warunek;
begin
    if sym=oddsym then
    begin pobsym; wyrażenie
    end else
    begin wyrażenie;
        if ¬ (sym in [równe, różne, mniejsze, mnrówne, większe,
            wrówne]) then błąd(20) else
        begin pobsym; wyrażenie
        end
```

```
        end
    end {warunek};
  begin {instrukcja}
    if sym=ident then
    begin i := pozycja(id);
        if i=0 then błąd(11) else
        if tablica[i].rodzaj≠zmienna then błąd(12);
        pobsym; if sym=przypis then pobsym else błąd(13);
        wyrażenie
    end else
    if sym=callsym then
    begin pobsym;
        if sym≠ident then błąd(14) else
            begin i := pozycja(id);
                if i=0 then błąd(11) else
                if tablica[i].rodzaj≠procedura then błąd(15);
                pobsym
            end
    end else
    if sym=ifsym then
    begin pobsym; warunek;
        if sym=thensym then pobsym else błąd(16);
        instrukcja;
    end else
    if sym=beginsym then
    begin pobsym; instrukcja;
        while sym=średnik do
            begin pobsym; instrukcja
            end;
        if sym=endsym then pobsym else błąd (17)
    end else
    if sym=whilesym then
    begin pobsym; warunek;
        if sym=dosym then pobsym else błąd(18);
        instrukcja
    end
  end {instrukcja};

begin {blok}
  if sym=constsym then
  begin pobsym; deklstałej;
    while sym=przec do
        begin pobsym; deklstałej
        end;
```

```
        if sym = średnik then pobsym else błąd(5)
    end;
    if sym = varsym then
    begin pobsym; deklzmiennej;
        while sym = przec do
            begin pobsym; deklzmiennej
            end;
        if sym = średnik then pobsym else błąd(5)
    end;
    while sym = procsym do
    begin pobsym;
        if sym = ident then
            begin wprowadź(procedura); pobsym
            end
        else błąd(4);
        if sym = średnik then pobsym else błąd(5);
        blok(tx);
        if sym = średnik then pobsym else błąd(5);
    end;
    instrukcja
end {blok};

begin {program główny}
    for ch := 'A' to ';' do ssym[ch] := żaden;
    słowo[ 1] := 'BEGIN     '; słowo[ 2] := 'CALL      ';
    słowo[ 3] := 'CONST     '; słowo[ 4] := 'DO        ';
    słowo[ 5] := 'END       '; słowo[ 6] := 'IF        ';
    słowo[ 7] := 'ODD       '; słowo[ 8] := 'PROCEDURE ';
    słowo[ 9] := 'THEN      '; słowo[10] := 'VAR       ';
    słowo[11] := 'WHILE     ';
    zsym[1]     := beginsym;    zsym[2]    := callsym;
    zsym[3]     := constsym;    zsym[4]    := dosym;
    zsym[5]     := endsym;      zsym[6]    := ifsym;
    zsym[7]     := oddsym;      zsym[8]    := procsym;
    zsym[9]     := thensym;     zsym[10]   := varsym;
    zsym[11]    := whilesym;
    ssym['+']   := plus;        ssym['−']  := minus;
    ssym['*']   := razy;        ssym['/']  := dziel;
    ssym['(']   := lnaw;        ssym[')']  := pnaw;
    ssym['=']   := równe;       ssym[',']  := przec;
    ssym['.']   := kropka;      ssym['≠']  := różne;
    ssym['<']   := mniejsze;    ssym['>']  := większe;
    ssym['≤']   := mnrówne;     ssym['≥']  := wrówne;
    ssym[';']   := średnik;
```

```
    page(output);
    lz := 0; dw := 0; ch := '  '; kk := al; pobsym;
    blok(0);
    if sym≠kropka then błąd(9);
99: writeln
end.
```

5.9. Reagowanie na błędy składniowe

Do tej pory jedynym zadaniem analizatora syntaktycznego było określenie, czy wejściowy ciąg symboli należy do języka czy też nie. Ujawnienie wewnętrznej struktury zdania stanowiło wynik uboczny. W momencie rozpoznania niepoprawnej konstrukcji cel pracy analizatora był osiągnięty i program mógł zakończyć działanie. Nie jest to jednak rozwiązanie, które można przyjąć w kompilatorach stosowanych w praktyce. Rzeczywisty kompilator powinien postawić odpowiednią diagnozę o błędzie i dalej kontynuować proces analizy – chociażby w celu znalezienia innych błędów. Kontynuacja jest możliwa jedynie wówczas, gdy przyjmie się pewne założenie dotyczące natury błędu i intencji autora niepoprawnego programu lub też ominie się w procesie analizy pewną część tekstu wejściowego (można też zastosować i jedno, i drugie podejście jednocześnie). Sztuka wybrania pewnego założenia o wysokim prawdopodobieństwie poprawności jest dość skomplikowana. Jak dotąd wymykała się ona wszelkim próbom udanej formalizacji. Formalizacje nie obejmują bowiem wielu czynników wywierających wpływ na umysł ludzki. Na przykład pospolitym błędem jest opuszczanie symboli interpunkcyjnych, takich jak średnik (nie tylko w programowaniu), natomiast jest wielce nieprawdopodobne, że ktoś zapomni o napisaniu operatora + w wyrażeniu arytmetycznym. Zarówno średnik, jak i plus są dla analizatora jedynie symbolami końcowymi. Dla człowieka-programisty średnik ma z trudem określone znaczenie i wydaje się zbyteczny na końcu wiersza, podczas gdy znaczenie operatora arytmetycznego nie pozostawia żadnych wątpliwości. Istnieje wiele innych względów, które należy rozważyć przy projektowaniu adekwatnego systemu reagowania na błędy składniowe; wszystkie one zależą od konkretnego języka i nie mogą być uogólnione na całość języków bezkontekstowych.

Niemniej jednak istnieją pewne reguły, które można postulować i które są słuszne nie tylko dla tak prostych języków jak PL/0. Charakterystyczne, że dotyczą one zarówno początkowej fazy tworzenia języka, jak i projektowania mechanizmu reagowania na błędy w analizatorze języka. Przede wszystkim jest oczywiste, że sensowne reagowanie na błędy jest znacznie ułatwione czy nawet w ogóle możliwe w przypadku **języków o prostej strukturze**. W szczególności, jeśli po postawieniu diagnozy dla jakiegoś błędu część ciągu wejściowego należy ominąć (zignorować), to język powinien zawierać takie słowa kluczowe, których

trudno użyć w sposób niewłaściwy, dzięki czemu umożliwiłyby one analizatorowi podjęcie dalszej analizy. W języku PL/0 reguła ta jest wyraźnie stosowana: każda instrukcja strukturalna zaczyna się trudnym do pomylenia słowem kluczowym, takim jak **begin**, **if**, **while**; podobnie, każda deklaracja zaczyna się od słowa **var**, **const** lub **procedure**. Regułę tę będziemy dlatego nazywać **regułą słów kluczowych**.

Druga reguła w sposób bezpośredni dotyczy konstrukcji analizatora. Dla rozbioru generacyjnego jest charakterystyczne, że cele rozbija się na podcele, analizator zaś wywołuje wtedy odpowiednie podanalizatory do analizowania tych podcelów. Reguła druga mówi, że jeśli analizator wykryje błąd, to (zamiast przerwania analizy i powiadomienia o tym analizatora głównego) powinien kontynuować czytanie ciągu wejściowego do punktu, od którego można by podjąć dalszą wiarogodną analizę. Regułę tę będziemy nazywać **regułą postępowania bez paniki**. Pragmatyczną konsekwencją tej reguły jest fakt, że analizator może skończyć swoje działanie tylko wtedy, gdy dojdzie do końcowego punktu programu.

Dokładna interpretacja reguły „postępowania bez paniki" jest następująca: analizator po wykryciu nielegalnego ciągu symboli szuka pierwszego symbolu, który może być poprawnym następnikiem aktualnie analizowanej konstrukcji. Implikuje to, że każdy analizator zna zbiór możliwych następników *nast* (w chwili jego uaktywnienia).

W pierwszym kroku precyzowania programu do listy parametrów każdej procedury analizującej dodamy dodatkowy parametr *fpocz*, który będzie określał zbiór możliwych następników. Na końcu każdej procedury umieścimy również test weryfikujący, czy kolejny symbol ciągu wejściowego należy do zbioru *fpocz* (z wyjątkiem przypadków, w których zapewnia to logika programu).

Krótkowzrocznością byłoby jednak omijać fragment tekstu wejściowego aż do miejsca wystąpienia pierwszego możliwego następnika, bez względu na wszystkie okoliczności. Ostatecznie programista mógł początkowo opuścić tylko jeden symbol (np. średnik); zignorowanie całego tekstu do miejsca wystąpienia symbolu ze zbioru *nast* byłoby niezbyt szczęśliwe. Dlatego też zbiory *fpocz* poszerzymy o dodatkowe symbole – słowa kluczowe, określające początki konstrukcji, których nie powinniśmy przeoczyć. Symbole przekazywane za pomocą parametrów *fpocz* zręczniej więc będzie nazywać **symbolami odgradzającymi** niż możliwymi następnikami. Zbiory symboli odgradzających będą tworzone z różnych słów kluczowych, a potem stopniowo rozszerzane o symbole-następniki w miarę posuwania się po drzewie podcelów. Dla wygody wprowadzimy dodatkową procedurę o nazwie *test*, której zadaniem będzie wykonywanie opisanego sprawdzania. Procedura ta (5.17) ma trzy parametry:

(1) zbiór dopuszczalnych następników – *s*1; jeżeli symbol bieżący nie należy do tego zbioru, to został wykryty błąd;
(2) zbiór dodatkowych symboli odgradzających – *s*2; wystąpienie symbolu z *s*2 jest wyraźnym błędem, ale w żadnym przypadku symbol ten nie powinien być zignorowany lub ominięty;
(3) liczbę *n* – określającą rozpoznany błąd.

```
procedure test(s1, s2: zbiórsym; n: integer);
begin if ¬ (sym in s1) then
   begin błąd(n); s1 := s1 + s2;                    (5.17)
      while ¬ (sym in s1) do pobsym
   end
end
```

Procedurę (5.17) można także zastosować na początku procedur analizujących w celu sprawdzenia, czy symbol bieżący jest dopuszczalnym symbolem początkowym. Jest to zalecane we wszystkich przypadkach, w których procedura analizująca X jest wywoływana w sposób bezwarunkowy. Przykładem może tu być instrukcja

if $sym = a_1$ **then** S_1 **else**

....

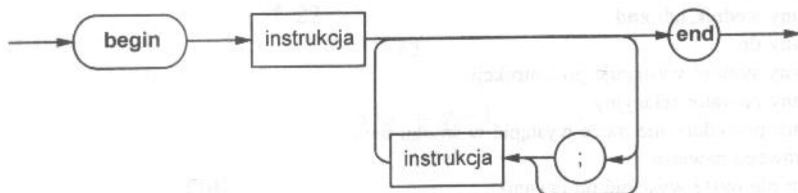
if $sym = a_n$ **then** S_n **else** X

stanowiąca wynik translacji produkcji

$$A ::= a_1 S_1 | \ldots | a_n S_n | X \tag{5.18}$$

W tym przypadku parametr $s1$ powinien być równy zbiorowi symboli początkowych dla X, podczas gdy $s2$ stanowi zbiór możliwych następników dla A (zob. tabl. 5.2). Szczegóły korzystania z tej procedury można znaleźć w programie 5.5, stanowiącym rozszerzoną wersję programu 5.4. Dla wygody czytelnika powtórzono tekst całego programu z wyjątkiem inicjowania zmiennych globalnych i procedury *pobsym*, które pozostają niezmienione.

W omawianym dotychczas schemacie próbowało się reagować na błędy przez zignorowanie jednego lub kilku symboli tekstu wejściowego. Nie jest to szczęśliwa strategia we wszystkich tych przypadkach, w których błąd został spowodowany *opuszczeniem* symbolu. Praktyka wykazuje, że tego rodzaju błędy sprowadzają się głównie do opuszczania symboli mających jedynie znaczenie syntaktyczne i nie powodujących żadnych akcji. Przykładem jest tu znak średnika w PL/0. Dzięki temu, że zbiory następników zostały rozszerzone o pewne słowa kluczowe, analizator nie pomija zbyt dużego fragmentu tekstu i zachowuje się tak, jak gdyby opuszczony symbol został wstawiony. Tego typu działanie można



RYSUNEK 5.6
Zmodyfikowana składnia instrukcji złożonej

zaobserwować w części programu (5.19) analizującej instrukcję złożoną. Opuszczone średniki są „wstawiane" przed słowami kluczowymi. Zbiór *poczinstr* jest zbiorem symboli początkowych dla konstrukcji „instrukcja".

```
if sym=beginsym then
begin pobsym;
   instrukcja ([średnik, endsym]+fpocz);
   while sym in [średnik]+poczinstr do
      begin                                                    (5.19)
         if sym=średnik then pobsym else błąd;
         instrukcja ([średnik, endsym]+fpocz)
      end;
   if sym=endsym then pobsym else błąd
end
```

Poprawność przyjętej diagnostyki błędów syntaktycznych i zdolność reagowania na niezwykłe sytuacje można ocenić na przykładzie programu PL/0. Przedstawiony tekst programu (5.20) stanowi zawartość pliku *output* otrzymanego z programu 5.5. Tablica 5.3. zawiera teksty komunikatów o błędach odpowiadające numerom błędów z programu 5.5.

TABLICA 5.3
Komunikaty o błędach z kompilatora PL/0

1. Użyj = zamiast :=
2. Po = powinna wystąpić liczba
3. Po identyfikatorze powinno wystąpić =
4. Po **const**, **var**, **procedure** powinien wystąpić identyfikator
5. Brakuje przecinka lub średnika
6. Niepoprawny symbol po deklaracji procedury
7. Spodziewana instrukcja
8. Niepoprawny symbol po części instrukcyjnej bloku
9. Spodziewana kropka
10. Brakuje średnika między instrukcjami
11. Niezadeklarowany identyfikator
12. Przypisanie wartości procedurze lub stałej jest niedozwolone
13. Spodziewany operator przypisania :=
14. Po **call** powinien wystąpić identyfikator
15. Wywołanie stałej lub zmiennej nie ma sensu
16. Spodziewany **then**
17. Spodziewany średnik lub **end**
18. Spodziewany **do**
19. Niepoprawny symbol występuje po instrukcji
20. Spodziewany operator relacyjny
21. Identyfikator procedury nie może wystąpić w środku wyrażenia
22. Brakuje prawego nawiasu
23. Symbol ten nie może wystąpić po czynniku
24. Wyrażenie nie może zacząć się od tego symbolu
25. Za duża liczba

Podany poniżej program (5.20) otrzymano z programów (5.14)–(5.16) przez wprowadzenie do nich błędów syntaktycznych.

```
const m=7, n=85
var x, y, z, q, r;
  ↑ 5
        ↑ 5

procedure mnóż;
   var a, b
begin a:=u; b:=y; z:=0
↑ 5
      ↑ 11
   while b>0 do
      ↑ 10
   begin
      if odd b do z:=z+a;
                 ↑ 16
                 ↑ 19
      a:=2a; b:=b/2;
            ↑ 23
   end
end;                                                    (5.20)
procedure dziel
   var w;
      ↑ 5
   const dwa=2, trzy:=3;
      ↑ 7
                      ↑ 1
begin r=x; q:=0; w:=y;
      ↑ 13
      ↑ 24
   while w ≤ r do w:=dwa*w;
   while w > y
      begin q:=(2*q; w:=w/2);
         ↑ 18
                      ↑ 22
                             ↑ 23
         if w ≤ r then
            begin r:=r-w q:=q+1
                              ↑ 23
            end
      end
end;
```

```
procedure nwd;
  var f, g;
begin f := x; g := y
  while f≠g do
      ↑ 17
    begin if f < g then g := g-f;
          if g < f then f := f-g;
  z := f
end;
begin
  x := m; y := n; call mnóż;
  x := 25; y := 3; call dziel;                    (5.20)
  x := 84; y := 36; call nwd;
  call x; x := nwd; nwd=x
      ↑ 15
            ↑ 21
                ↑ 12
                   ↑ 13
                   ↑ 24
end.
  ↑17
  ↑ 5
  ↑ 7
PROGRAM NIEDOKOŃCZONY
```

Jasne jest, że żaden schemat translacji, który z rozsądną efektywnością analizuje zdania poprawne, nie będzie jednocześnie zdolny do przetwarzania w sensowny sposób wszystkich możliwych konstrukcji niepoprawnych. Każdy schemat o rozsądnych kosztach realizacji będzie działał w sposób niezadowalający dla pewnych konstrukcji niepoprawnych. Dobry kompilator powinien mieć jednak następujące własności:

(1) żaden ciąg wejściowy nie powinien doprowadzić do przerwania działania programu kompilatora;
(2) wszystkie konstrukcje, które w myśl definicji języka są nielegalne, powinny być wykryte i zaznaczone na wydrukowanym tekście programu;
(3) błędy pojawiające się względnie często, będące autentycznymi pomyłkami programisty (przeoczenia i nieporozumienia) są właściwie diagnozowane i nie powodują żadnych dalszych potknięć kompilatora (tzw. **fałszywych** komunikatów o błędach).

Omawiany schemat działa w stopniu zadowalającym, aczkolwiek w wielu miejscach można by próbować usprawnić jego działanie. Zaletą jego jest to, że został zbudowany w systematyczny sposób na podstawie kilku podstawowych reguł. Reguły te zostały jedynie uzupełnione pewnymi ustaleniami dotyczącymi parametrów, wynikającymi z doświadczeń zdobytych przy praktycznym używaniu języka.

PROGRAM 5.5
Analizator PL/0 z reagowaniem na błędy składniowe

```
program PL0(input, output);
{kompilator PL/0, reagowanie na błędy składniowe}
label 99;
const lsz = 11;         {liczba słów zastrzeżonych}
   tmax = 100;          {długość tablicy identyfikatorów}
   cmax = 14;           {maksymalna liczba cyfr w liczbach}
   al = 10;             {długość identyfikatorów}
type symbol =
   (żaden, ident, liczba, plus, minus, razy, dziel, oddsym, równe, różne, mniejsze,
   mnrówne, większe, wrówne, lnaw, pnaw, przec, średnik, kropka, przypis,
   beginsym, endsym, ifsym, thensym, whilesym, dosym, callsym, constsym,
   varsym, procsym);
   alfa = packed array [1..al] of char;
   obiekt = (stała, zmienna, procedura);
   zbiórsym = set of symbol;
var ch: char;           {ostatni wczytany znak}
   sym: symbol;         {ostatni wczytany symbol}
   id: alfa;            {ostatni wczytany identyfikator}
   num: integer;        {ostatnia wczytana liczba}
   lz: integer;         {licznik znaków}
   dw: integer;         {długość wiersza}
   kk: integer;
   wiersz: array [1..81] of char;
   a: alfa;
   słowo: array [1..lsz] of alfa;
   zsym: array [1..lsz] of symbol;
   ssym: array [char] of symbol;
   poczdekl, poczinstr, poczczyn: zbiórsym;
   tablica: array [0..tmax] of
            record nazwa: alfa;
                   rodzaj: obiekt
            end;
procedure błąd(n: integer);
begin writeln('    ': lz, '↑', n: 2);
end {błąd};

procedure test(s1, s2: zbiórsym; n: integer);
begin if ¬ (sym in s1) then
   begin błąd(n); s1 := s1 + s2;
      while ¬ (sym in s1) do pobsym
   end
end {test};
```

```
procedure blok(tx: integer; fpocz: zbiórsym);
   procedure wprowadź(k: obiekt);
   begin {wprowadź obiekt do tablicy}
      tx := tx + 1;
      with tablica[tx] do
      begin nazwa := id; rodzaj := k;
      end
   end {wprowadź};

   function pozycja(id: alfa): integer;
      var i: integer;
   begin {znajdź identyfikator id w tablicy}
      tablica[0].nazwa := id; i := tx;
      while tablica[i].nazwa ≠ id do i := i−1;
      pozycja := i
   end {pozycja};

   procedure deklstałej;
   begin if sym = ident then
      begin pobsym;
         if sym in [równe, przypis] then
         begin if sym = przypis then błąd(1);
            pobsym;
            if sym = liczba then
               begin wprowadź(stała); pobsym
               end
            else błąd(2)
         end else błąd(3)
      end else błąd(4)
   end {delstałej};

   procedure deklzmiennej;
   begin if sym = ident then
      begin wprowadź(zmienna); pobsym
      end else błąd(4)
   end {deklzmienej};

   procedure instrukcja(fpocz: zbiórsym);
      var i: integer;
      procedure wyrażenie (fpocz: zbiórsym);
         procedure składnik (fpocz: zbiórsym);
            procedure czynnik (fpocz: zbiórsym);
               var i: integer;
            begin test(poczczyn, fpocz, 24);
               while sym in poczczyn do
```

```
          begin
             if sym = ident then
             begin i := pozcyja(id);
                if i = 0 then błąd(11) else
                if tablica[i].rodzaj = procedura then błąd(21);
                pobsym
             end else
             if sym = liczba then
             begin pobsym;
             end else
             if sym = lnaw then
             begin pobsym; wyrażenie([pnaw]+fpocz);
                if sym = pnaw then pobsym else błąd(22)
             end;
             test(fpocz, [lnaw], 23)
          end
       end {czynnik};
    begin {składnik} czynnik(fpocz+[razy, dziel]);
       while sym in [razy, dziel] do
          begin pobsym; czynnik (fpocz+[razy, dziel])
          end
    end {składnik};
 begin {wyrażenie}
    if sym in [plus, minus] then
          begin pobsym; składnik(fpocz+[plus, minus])
          end else składnik(fpocz+[plus, minus]);
    while sym in [plus, minus] do
       begin pobsym; składnik(fpocz+[plus, minus])
       end
 end {wyrażenie};

 procedure warunek(fpocz: zbiórsym);
 begin
    if sym = oddsym then
    begin pobsym; wyrażenie(fpocz);
    end else
    begin wyrażenie([równe, różne, mniejsze, mnrówne, większe, wrówne]
                   +fpocz);
       if ¬ (sym in [równe, różne, mniejsze, mnrówne, większe, wrówne])
           then błąd(20) else
       begin pobsym; wyrażenie(fpocz)
       end
    end
 end {warunek};
```

```
  begin {instrukcja}
    if sym = ident then
    begin i := pozycja(id);
      if i = 0 then błąd(11) else
      if tablica[i].rodzaj ≠ zmienna then błąd(12);
      pobsym; if sym = przypis then pobsym else błąd(13);
      wyrażenie(fpocz);
    end else
    if sym = callsym then
    begin pobsym;
      if sym ≠ ident then błąd(14) else
        begin i := pozycja(id);
          if i = 0 then błąd(11) else
          if tablica[i].rodzaj ≠ procedura then błąd(15);
          pobsym
        end
    end else
    if sym = ifsym then
    begin pobsym; warunek([thensym, dosym] + fpocz);
      if sym = thensym then pobsym else błąd(16);
      instrukcja(fpocz)
    end else
    if sym = beginsym then
    begin pobsym; instrukcja([średnik, endsym] + fpocz);
      while sym in [średnik] + poczinstr do
      begin
        if sym = średnik then pobsym else błąd(10);
        instrukcja([średnik, endsym] + fpocz)
      end;
      if sym = endsym then pobsym else błąd(17)
    end else
    if sym = whilesym then
    begin pobsym; warunek([dosym] + fpocz);
      if sym = dosym then pobsym else błąd(18);
      instrukcja(fpocz);
    end;
    test(fpocz, [ ], 19)
  end {instrukcja};

begin {blok}
  repeat
    if sym = constsym then
    begin pobsym;
      repeat deklstałej;
```

```
                while sym = przec do
                      begin pobsym; deklstałej
                      end;
                if sym = średnik then pobsym else błąd(5)
          until sym ≠ ident
      end;
      if sym = varsym then
      begin pobsym;
          repeat deklzmiennej;
                while sym = przec do
                      begin pobsym; deklzmiennej
                      end;
                if sym = średnik then pobsym else błąd(5)
          until sym ≠ ident;
      end;
      while sym = procsym do
      begin pobsym;
          if sym = ident then
                begin wprowadź(procedura); pobsym
                end
          else błąd(4);
          if sym = średnik then pobsym else błąd(5);
          blok(tx, [średnik] + fpocz);
          if sym = średnik then
                begin pobsym; test(poczinstr + [ident, procsym], fpocz, 6)
                end
          else błąd(5)
      end;
      test(poczinstr + [ident], poczdekl, 7)
  until ¬(sym in poczdekl);
  instrukcja([średnik, endsym] + fpocz);
  test(fpocz, [ ], 8);
end {blok};

begin {program główny}
  ... nadanie zmiennym wartości początkowych (zob. program 5.4) ...
  lz := 0; dw := 0; ch := ' '; kk := al; pobsym;
  blok(0, [kropka] + poczdekl + poczinstr);
  if sym ≠ kropka then błąd(9);
99: writeln
end.
```

5.10. Maszyna PL/0

Jest istotnie godne uwagi, że przy opracowywaniu kompilatora PL/0 do tej pory nic nie wiemy o maszynie, dla której ten kompilator ma produkować kod wynikowy. Ale dlaczego struktura maszyny docelowej miałaby mieć wpływ na schemat analizy syntaktycznej i na schemat reagowania na błędy? W rzeczywistości *nie powinna* mieć żadnego wpływu. Natomiast odpowiedni schemat generowania kodu dla dowolnego koputera powinien być nałożony, metodą stopniowego precyzowania, na program analizatora syntaktycznego. Skoro mamy zamiar to właśnie robić, to powinniśmy precyzyjnie określić maszynę docelową.

W celu uproszczenia opisu kompilatora oraz uniknięcia ubocznych rozważań związanych ze szczególnymi właściwościami rzeczywistej maszyny przyjmiemy jako maszynę docelową komputer specjalnie „przykrojony" do potrzeb języka PL/0. Maszyna taka w rzeczywistości nie istnieje; jest to maszyna hipotetyczna. Nazywać ją będziemy **maszyną PL/0**.

Nie zamierzamy w tym punkcie wyjaśniać dokładnie powodów przyjęcia takiej właśnie, a nie innej organizacji maszyny PL/0. Punkt ten należy traktować jako podręcznik składający się z intuicyjnego wprowadzenia i z następującej po nim, szczegółowej definicji komputera w postaci algorytmu. Formalizacja ta może służyć jako przykład dokładnego, algorytmicznego opisu współczesnych maszyn. Algorytm interpretuje sekwencyjnie instrukcje maszyny PL/0 i dlatego zwany jest **interpretatorem**.

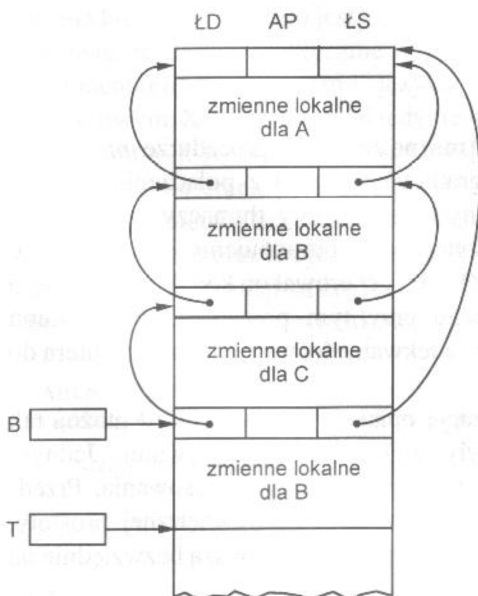
Maszyna PL/0 składa się z dwóch rodzajów pamięci, rejestru instrukcji i trzech rejestrów adresowych. **Pamięć programu**, zwana **kodem**, jest wypełniona przez kompilator i pozostaje niezmieniona podczas interpretacji tego kodu. Możemy ją więc traktować jako pamięć, z której można tylko pobierać informacje. **Pamięć danych** S jest zorganizowana jako **stos**; interpretacja każdego operatora arytmetycznego polega na zastąpieniu dwóch elementów na wierzchołku stosu (argumentów) wynikiem operacji. Indeks (adres) elementu znajdującego się na wierzchołku stosu jest przechowywany w **rejestrze wierzchołka stosu** T. **Rejestr instrukcji** I zawiera instrukcję aktualnie interpretowaną. Rejestr adresowy P (**licznik instrukcji**) określa następną instrukcję przeznaczoną do interpretacji.

Każda procedura w języku PL/0 może mieć zmienne lokalne. Ponieważ procedury mogą być uaktywniane rekurencyjnie, nie można przydzielać pamięci dla zmiennych lokalnych przed wywołaniem procedury. Z tego powodu segmenty danych poszczególnych procedur są umieszczane po kolei w pamięci stosowej S podczas uaktywniania tych procedur. Zakończenie realizacji danego wywołania procedury jest związane ze zwolnieniem pamięci przeznaczonej na jej segment danych, tzn. elementów na wierzchołku stosu. Widać stąd, że stos jest właściwą strukturą dla opisanej strategii przydzielania pamięci. Z każdą procedurą jest związana pewna informacja o niej samej, mianowicie adres wywołania jej w programie (tzw. **adres powrotu**) i adres segmentu danych procedury, z której została ona wywołana. Te dwa adresy umożliwiają właściwą kontynuację wykonywania programu po zakończeniu działania procedury. Adresy te możemy

traktować jako wewnętrzne lub niejawne zmienne lokalne, umieszczone w segmencie danych wywołanej procedury. Nazywać je będziemy **adresem powrotu** *AP* i **łączem (wiązaniem) dynamicznym** *ŁD*. Początek łańcucha łączy dynamicznych, tj. adres ostatnio umieszczonego na stosie segmentu danych, jest przechowywany w **bazowym rejestrze adresowym** *B*.

Skoro przydzielenie pamięci odbywa się podczas wykonania programu (interpretacji), to kompilator nie może umieszczać adresów absolutnych w generowanym kodzie. Znając jednak adresy zmiennych wewnątrz segmentu danych, kompilator może generować **adresy względne**. Interpretator, wyliczając adres zmiennej, do adresu bazowego odpowiedniego segmentu danych musi dodać adres zmiennej w segmencie, tzw. **przesunięcie**. Jeżeli zmienna jest lokalna dla procedury aktualnie interpretowanej, to adres bazowy znajduje się w rejestrze *B*. W przeciwnym razie w celu wyznaczenia adresu bazowego należy zejść po łańcuchu segmentów danych. Kompilator może znać jednak tylko statyczną długość ścieżki prowadzącej do segmentu potrzebnej procedury, podczas gdy łańcuch dynamiczny reprezentuje dynamiczną historię uaktywnień procedur. Niestety jednak, ścieżka statyczna i ścieżka dynamiczna nie muszą się pokrywać.

Załóżmy na przykład, że procedura *A* wywołuje procedurę *B* zadeklarowaną jako lokalna dla *A*, *B* wywołuje *C* (lokalną dla *B*) oraz procedura *C* wywołuje *B* (rekurencyjnie). Będziemy mówić, że *A* jest zadeklarowana na poziomie 1, *B* na poziomie 2, *C* na poziomie 3 (zob. rys. 5.7). Jeżeli w procedurze *B* odwołujemy się do zmiennej *a* zadeklarowanej w *A*, to kompilator wie, że **różnica poziomów** między *B* i *A* wynosi 1. Schodząc jednak po łańcuchu dynamicznym o jeden poziom niżej, uzyskaliśmy dostęp do zmiennej lokalnej dla procedury *C*!



RYSUNEK 5.7
Stos maszyny PL/0

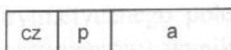
Zrozumiałe jest więc, że potrzebny jest drugi łańcuch, umożliwiający w takich sytuacjach właściwe łączenie segmentów danych. Nazywać go będziemy **łańcuchem statycznym**, a pojedyncze wiązanie łańcucha – **łączem statycznym** *ŁS*.

Adresy będą więc generowane jako pary liczb wskazujących statyczną różnicę poziomów i względne przesunięcie wewnątrz segmentu danych. Zakładamy, że w każdym słowie pamięci danych możemy przechowywać jeden adres lub jedną liczbę całkowitą.

Zbiór rozkazów maszyny PL/0 jest dopasowany do wymagań języka PL/0. Zawiera on następujące rozkazy:

(1) rozkaz umieszczania liczby (stałej) na wierzchołku stosu (STA);
(2) rozkaz pobierania wartości zmiennej i ładowania jej na wierzchołek stosu (ŁAD);
(3) rozkaz pamiętania (PAM) odpowiadający instrukcji przypisania;
(4) rozkaz uaktywniania podprogramu (PRO), odpowiadający wywołaniu procedury;
(5) rozkaz przydzielania pamięci na stosie (PRZ) przez zwiększenie wartości rejestru *T*;
(6) rozkazy bezwarunkowego (SKB) i warunkowego skoku (SKW), używanego w instrukcjach **if** i **while**;
(7) rozkazy realizujące operatory arytmetyczne i relacyjne (OPR).

Wzorzec rozkazu jest określony przez jego składowe: kod realizowanej czynności *cz* i parametr składający się z dwóch części (zob. rys. 5.8). W zależności od kodu czynności *cz* parametr określa: tożsamość operatora (OPR), liczbę (STA, PRZ), adres w programie (SKB, SKW, PRO) lub adres w danych (ŁAD, PAM).



RYSUNEK 5.8
Wzorzec rozkazu

Szczegóły działania maszyny PL/0 można znaleźć w procedurze *interpretuj*, stanowiącej część programu 5.6. Program ten powstał z połączenia pełnego kompilatora i interpretatora we wspólny system, który tłumaczy, a następnie wykonuje programy w języku PL/0. Czytelnikowi proponujemy jako ćwiczenie modyfikację tego programu w taki sposób, aby generował on kod dla istniejącego w rzeczywistości komputera. Otrzymane przy tym powiększenie programu kompilatora może stanowić pewną miarę adekwatności wybranego komputera do realizacji naszego zadania.

Nie ulega wątpliwości, że organizację opisanej maszyny PL/0 można tak rozbudować, aby pewne operacje były wykonywane efektywniej. Jednym z możliwych przykładów jest wybór innego mechanizmu adresowania. Przedstawione rozwiązanie zostało wybrane z powodu jego wewnętrznej prostoty, a także dlatego, że wszelkie jego ewentualne usprawnienia muszą bezwzględnie na nim opierać się i z niego wychodzić.

5.11. Generowanie kodu wynikowego

Kompilator tłumaczy pojedyncze rozkazy na podstawie ich kodów czynności i parametrów będących liczbami lub adresami. Podczas przetwarzania deklaracji stałych, zmiennych i procedur kompilator kojarzy wartości parametrów rozkazów z odpowiednimi identyfikatorami. Z tego też względu tablicę zawierającą identyfikatory rozszerza się o związane z każdym z nich atrybuty. Jeśli identyfikator oznacza stałą, to jego atrybutem jest wartość stałej; jeśli identyfikator oznacza zmienną, to jego atrybutem jest adres zmiennej składający się z wartości przesunięcia i poziomu; jeżeli identyfikator oznacza procedurę, to jego atrybutami są: adres wejścia do procedury i poziom procedury. Odpowiednie rozszerzenie deklaracji zmiennej *tablica* pokazano w programie 5.6. Jest to godny uwagi przykład stopniowego precyzowania (czy wzbogacania) deklaracji danych jednocześnie z precyzowaniem części instrukcyjnej programu.

Wartości stałych występują bezpośrednio w tekście programu, określenie adresów jest natomiast jednym z zadań kompilatora. Język PL/0 jest dostatecznie prosty na to, aby sekwencyjne przydzielanie pamięci dla zmiennych i rozkazów programu przebiegało w sposób oczywisty. Przetworzenie każdej deklaracji sprowadza się zatem do zwiększenia licznika adresów danych o 1 (każda zmienna, zgodnie z definicją maszyny PL/0, zajmuje jedno słowo pamięci). Z chwilą rozpoczęcia kompilacji nowej procedury licznikowi adresów danych *dx* nadaje się wartość początkową. Ponieważ segment danych składa się wtedy tylko z trzech zmiennych wewnętrznych *AP*, *ŁD* i *ŁS* (zob. p. 5.10), więc wartość początkowa *dx* jest równa 3. Wyznaczenie atrybutów identyfikatorów odbywa się w procedurze *wprowadź*, służącej do wprowadzania nowych identyfikatorów do tablicy.

Po zapewnieniu bezpośredniego dostępu do atrybutów argumentów generowanie kodu wynikowego jest już dość proste. Stosowa organizacja maszyny PL/0 sprawia, że istnieje praktycznie wzajemnie jednoznaczna odpowiedniość między argumentami i operatorami języka źródłowego a rozkazami w programie wynikowym. Kompilator musi jedynie wykonać odpowiednie przejście do *notacji przyrostkowej*. W notacji przyrostkowej operatory występują zawsze po argumentach – w odróżnieniu od popularnej notacji wrostkowej, w której operatory występują między argumentami. Notacja przyrostkowa jest także nazywana notacją polską (autorem jej był Polak J. Łukasiewicz) lub *notacją beznawiasową* (nawiasy w tej notacji są zbyteczne). Przykłady odpowiedniości między notacją przyrostkową i notacją wrostkową pokazano w tabl. 5.4 (zob. również p. 4.4.2).

TABLICA 5.4
Przykłady wyrażeń w notacji wrostkowej i przyrostkowej

Notacja wrostkowa	Notacja przyrostkowa
$x+y$	$xy+$
$(x-y)+z$	$xy-z+$
$x-(y+z)$	$xyz+-$
$x*(y+z)*w$	$xyz+*w*$

Prostą metodę przejścia do notacji przyrostkowej pokazano w procedurach *wyrażenie* i *składnik* w programie 5.6. Istota tej metody polega jedynie na opóźnieniu transmisji operatora arytmetycznego. Czytelnik powinien sprawdzić, że w prezentowanym układzie procedur dokonujących rozbioru uwzględniono właściwą interpretację przyjętych zwyczajowo reguł pierwszeństwa różnych operatorów.

Nieco bardziej skomplikowana jest translacja instrukcji warunkowych i iteracyjnych. W tym przypadku trzeba generować rozkaz skoków, dla których adres skoku nie jest jeszcze znany. Jeżeli zależy nam na ściśle sekwencyjnym generowaniu rozkazów (do pliku wyjściowego), to jest konieczne stosowanie kompilatora *dwuprzebiegowego*. Zadaniem drugiego przebiegu translacji będzie uzupełnienie rozkazów skoku obliczonymi adresami. Alternatywne rozwiązanie zastosowane w naszym kompilatorze polega na umieszczaniu rozkazów w tablicy przechowywanej w pamięci operacyjnej. Metoda ta pozwala uzupełnić brakujące adresy, skoro tylko są znane.

Jedyną dodatkową operacją, jaką należy wykonać dla rozkazu skoku do przodu, jest zapamiętanie miejsca wystąpienia tego rozkazu, tj. jego indeksu w pamięci programu. Indeksu tego używa się później w celu lokalizacji niepełnego rozkazu. Szczegóły można znaleźć w programie 5.6 (zob. procedury przetwarzające instrukcje **if** i **while**). Układ kodu wynikowego produkowanego dla instrukcji **if** i **while** jest następujący (*L*1 i *L*2 oznaczają adresy w kodzie):

if *W* **then** *I*	**while** *W* **do** *I*
kod wynikowy dla warunku *W*	L1: kod wynikowy dla warunku *W*
SKW L1	SKW L2
kod wynikowy dla instrukcji *I*	kod wynikowy dla instrukcji *I*
L1: ...	SKB L1
	L2: ...

Dla wygody wprowadzimy pomocniczą procedurę *gen*. Jej zadaniem jest przetłumaczenie i wyprowadzenie rozkazu zgodnie z jego trzema parametrami. Zwiększa ona również licznik rozkazów *lr* wyznaczający adres następnego wyprowadzanego rozkazu.

Poniższy przykład przedstawia program wynikowy otrzymany w wyniku kompilacji procedury *mnóż* (5.14). Komentarze po prawej stronie mają jedynie charakter objaśniający.

Program wynikowy odpowiadający procedurze 5.14

```
2  PRZ   0, 5   przydzielenie pamięci dla łączy i zmiennych lokalnych
3  ŁAD   1, 3   x
4  PAM   0, 3   a
```

```
 5  ŁAD  1, 4   y
 6  PAM  0, 4   b
 7  STA  0, 0   0
 8  PAM  1, 5   z
 9  ŁAD  0, 4   b
10  STA  0, 0   0
11  OPR  0, 12  >
12  SKW  0, 29
13  ŁAD  0, 4   b
14  OPR  0, 7   testowanie nieparzystości
15  SKW  0, 20
16  ŁAD  1, 5   z
17  ŁAD  0, 3   a
18  OPR  0, 2   +
19  PAM  1, 5   z
20  STA  0, 2   2
21  ŁAD  0, 3   a
22  OPR  0, 4   *
23  PAM  0, 3   a
24  ŁAD  0, 4   b
25  STA  0, 2   2
26  OPR  0, 5   /
27  PAM  0, 4   b
28  SKB  0, 9
29  OPR  0, 0   powrót
```

Wiele zagadnień związanych z kompilacją języków programowania ma charakter bardziej złożony niż te, które przedstawiliśmy podczas omawiania kompilatora języka PL/0 dla maszyny PL/0 [5.4]. O większości z nich trudno jest mówić w równie gładki sposób, jak w przypadku omawianego kompilatora. Czytelnik próbujący wzbogacić ten kompilator albo przez rozszerzenie języka programowania, albo przez wprowadzenie bardziej „życiowego" komputera, przekona się szybko o słuszności tego stwierdzenia. Niemniej jednak przedstawione w niniejszym rozdziale podstawowe podejście do projektowania złożonych programów okazuje się słuszne, a przy rozważaniu problemów bardziej wymyślnych i złożonych metoda ta okaże się nawet bardziej wartościowa. W rzeczywistości metodę tę zastosowano z powodzeniem przy realizacji kompilatorów o dużych rozmiarach [5.1] i [5.9].

PROGRAM 5.6
Kompilator języka PL/0

```
program PL0 (input, output);
{kompilator PL/0 łącznie z generowaniem kodu wynikowego}
label 99;
const lsz = 11;        {liczba słów zastrzeżonych}
   tmax = 100;   {długość tablicy identyfikatorów}
   cmax = 14;    {maksymalna liczba cyfr w liczbach}
   al = 10;      {długość identyfikatorów}
   amax = 2047;  {maksymalny adres}
   pozmax = 3;   {maksymalna głębokość zagnieżdżenia bloków}
   rmax = 200;   {długość tablicy z rozkazami kodu wynikowego}
type symbol =
   (żaden, ident, liczba, plus, minus, razy, dziel, oddsym, równe, różne,
   mniejsze, mnrówne, większe, wrówne, lnaw, pnaw, przec, średnik, kropka,
   przypis, beginsym, endsym, ifsym, thensym, whilesym, dosym, callsym,
   constsym, varsym, procsym);
   alfa = packed array [1..al] of char;
   obiekt = (stała, zmienna, procedura);
   zbiórsym = set of symbol;
   czynności = (sta, opr, ład, pam, pro, prz, skb, skw);
   rozkaz = packed record
            cz: czynności; {kod czynności}
            p: 0..pozmax; {poziom}
            a: 0..amax;   {przesunięcie}
         end;
{STA 0, a : ładuj stałą a
 OPR 0, a : wykonaj operację a
 ŁAD p, a : ładuj zmienną p, a
 PAM p, a : pamiętaj zmienną p, a
 PRO p, a : wywołaj procedurę a na poziomie p
 PRZ 0, a : zwiększ t-rejestr o a
 SKB 0, a : skocz bezwarunkowo do a
 SKW 0, a : skocz warunkowo do a}
var ch: char;          {ostatni wczytany znak}
   sym: symbol;        {ostatni wczytany symbol}
   id: alfa;           {ostatni wczytany identyfikator}
   num: integer;       {ostatnia wczytana liczba}
   lz: integer;        {licznik znaków}
   dw: integer;        {długość wiersza}
   kk: integer;
   lr: integer;        {licznik rozkazów}
   wiersz: array [1..81] of char;
```

```
    a: alfa;
    kod: array [0..rmax] of rozkaz;
    słowo: array [1..lsz] of alfa;
    zsym: array [1..lsz] of symbol;
    ssym: array [char] of symbol;
    mnem: array [czynności] of
        packed array [1..5] of char;
    poczdekl, poczinstr, poczczyn: zbiórsym;
    tablica: array [0..tmax] of
            record nazwa: alfa;
                case rodzaj: obiekt of
                stała: (wartość: integer);
                zmienna, procedura: (poziom, adr: integer)
            end;
    błędy: boolean;

procedure błąd(n: integer);
begin writeln(' **** ',' ': lz-1, '↑', n: 2); błędy := true
end {błąd};

procedure pobsym;
    var i, j, k: integer;
    procedure pobznak;
    begin if lz=dw then
        begin if eof(input) then
                begin write('PROGRAM NIEDOKOŃCZONY'); goto 99
                end;
            dw := 0; lz := 0; write(lr: 5, ' ');
            while ¬ eol(input) do
                begin dw := dw+1; read(ch); write(ch); wiersz[dw] := ch
                end;
            writeln; dw := dw+1; read(wiersz[dw])
        end;
        lz := lz+1; ch := wiersz[lz]
    end {pobznak};

begin {pobsym}
    while ch=' ' do pobznak;
    if ch in ['A'..'Z'] then
    begin {identyfikator lub słowo zastrzeżone} k := 0;
        repeat if k<al then
            begin k := k+1; a[k] := ch
            end;
            pobznak
        until ¬ (ch in ['A'..'Z', '0'..'9']);
```

```
        if k≥kk then kk := k else
            repeat a[kk] := ' '; kk := kk−1
            until kk=k;
        id := a; i := 1; j := lsz;
        repeat k := (i+j) div 2;
            if id≤słowo[k] then j := k−1;
            if id≥słowo[k] then i := k+1
        until i>j;
        if i−1>j then sym := zsym[k] else sym := ident
    end else
    if ch in ['0'..'9'] then
    begin {liczba} k=0; num := 0; sym := liczba;
        repeat num := 10*num+(ord(ch)−ord('0'));
            k := k+1; pobznak
        until ¬ (ch in ['0'..'9']);
        if k>cmax then błąd(30)
    end else
    if ch=':' then
    begin pobznak;
        if ch='=' then
        begin sym := przypis; pobznak
        end else sym := żaden;
    end else
    begin sym := ssym[ch]; pobznak
    end
end {pobsym};

procedure gen(x: czynności; y, z: integer);
begin if lr>rmax then
        begin write('ZA DŁUGI PROGRAM'); goto 99
        end;
    with kod[lr] do
        begin cz := x; p := y; a := z
        end;
    lr := lr+1
end {gen};

procedure test(s1, s2: zbiórsym; n: integer);
begin if ¬ (sym in s1) then
        begin błąd(n); s1 := s1+s2;
            while ¬ (sym in s1) do pobsym
        end
end {test};
```

```
procedure blok(poz, tx: integer; fpocz: zbiórsym);
  var dx: integer;        {indeks segmentu danych}
    tx0: integer;         {początkowa wartość indeksu tx}
    lr0: integer;         {początkowa wartość licznika lr}

  procedure wprowadź(k: obiekt);
  begin {wprowadź obiekt do tablicy}
    tx := tx+1;
    with tablica[tx] do
    begin nazwa := id; rodzaj := k;
      case k of
        stała: begin if num > amax then
                       begin błąd(31); num := 0 end;
                   wartość := num
               end;
        zmienna: begin poziom := poz; adr := dx; dx := dx+1;
                 end;
        procedura: poziom := poz
      end
    end
  end {wprowadź};

  function pozycja(id: alfa): integer;
    var i: integer;
  begin {znajdź identyfikator id w tablicy}
    tablica [0].nazwa := id; i := tx;
    while tablica [i].nazwa ≠ id do i := i-1;
    pozycja := i
  end {pozycja};

  procedure deklstałej;
  begin if sym=ident then
    begin pobsym;
      if sym in [równe, przypis] then
      begin if sym=przypis then błąd(1);
        pobsym;
        if sym=liczba then
          begin wprowadź(stała); pobsym
          end
        else błąd(2)
      end else błąd(3)
    end else błąd(4)
  end {deklstałej};
```

```
procedure deklzmiennej;
begin if sym=ident then
      begin wprowadź(zmienna); pobsym
      end else błąd(4)
end {deklzmiennej};

procedure drukujkod;
   var i: integer;
begin {drukuj program wyprodukowany dla aktualnego bloku}
  for i:= lr0 to lr-1 do
      with kod[i] do
         writeln(i, mnem [cz]: 5, p: 3, a: 5)
end {drukujkod};

procedure instrukcja(fpocz: zbiórsym);
   var i, lr1, lr2: integer;

   procedure wyrażenie(fpocz: zbiórsym);
      var opaddyt: symbol;

      procedure składnik(fpocz: zbiórsym);
         var opmult: symbol;

         procedure czynnik(fpocz: zbiórsym);
            var i: integer;
         begin test(poczczyn, fpocz, 24);
            while sym in poczczyn do
            begin
               if sym=ident then
               begin i:= pozycja(id);
                  if i=0 then błąd(11) else
                  with tablica [i] do
                  case rodzaj of
                        stała: gen(sta, 0, wartość);
                        zmienna: gen(ład, poz-poziom, adr);
                        procedura: błąd(21)
                  end;
                  pobsym
               end else
               if sym=liczba then
               begin if num > amax then
                     begin błąd(31); num:= 0
                     end;
                  gen(sta, 0, num); pobsym
               end else
```

```
                if sym=lnaw then
                begin pobsym; wyrażenie([pnaw]+fpocz);
                    if sym=pnaw then pobsym else błąd(22)
                end;
                test(fpocz, [lnaw], 23)
            end
        end {czynnik};
    begin {składnik} czynnik(fpocz + [razy, dziel]);
        while sym in [razy, dziel] do
            begin opmult := sym; pobsym;
                czynnik(fpocz + [razy, dziel]);
                if opmult=razy then gen(opr, 0, 4)
                else gen(opr, 0, 5)
            end
    end {składnik};
begin {wyrażenie}
    if sym in [plus, minus] then
        begin opaddyt := sym; pobsym;
            składnik(fpocz + [plus, minus]);
            if opaddyt=minus then gen(opr, 0, 1)
        end else składnik(fpocz + [plus, minus]);
    while sym in [plus, minus] do
        begin opaddyt := sym; pobsym;
            składnik(fpocz + [plus, minus]);
            if opaddyt=plus then gen(opr, 0, 2) else gen(opr, 0, 3)
        end
end {wyrażenie};

procedure warunek(fpocz: zbiórsym);
    var oprel: symbol;
begin
    if sym=oddsym then
    begin pobsym; wyrażenie(fpocz); gen(opr, 0, 6)
    end else
    begin wyrażenie([równe, różne, mniejsze, większe, mnrówne, wrówne]
                    +fpocz);
        if ¬ (sym in [równe, różne, mniejsze, większe, mnrówne, wrówne])
                then błąd(20) else
        begin oprel := sym; pobsym; wyrażenie(fpocz);
            case oprel of
                równe:      gen(opr, 0, 8);
                różne:      gen(opr, 0, 9);
                mniejsze:   gen(opr, 0, 10);
                wrówne:     gen(opr, 0, 11);
```

```
                większe:    gen(opr, 0, 12);
                mnrówne:  gen(opr, 0, 13);
            end
        end
    end
  end {warunek};


begin {instrukcja}
  if sym = ident then
  begin i := pozycja(id);
    if i = 0 then błąd(11) else
    if tablica [i].rodzaj ≠ zmienna then
      begin {przypisanie wartości nie zmiennej} błąd(12); i := 0
      end;
    pobsym; if sym = przypis then pobsym else błąd(13);
    wyrażenie(fpocz);
    if i ≠ 0 then
      with tablica[i] do gen(pam, poz-poziom, adr)
  end else
  if sym = callsym then
  begin pobsym;
    if sym ≠ ident then błąd(14) else
      begin i := pozycja(id);
        if i=0 then błąd(11) else
        with tablica[i] do
          if rodzaj = procedura then gen(pro, poz-poziom, adr)
          else błąd(15);
        pobsym
      end
  end else
  if sym = ifsym then
  begin pobsym; warunek([thensym, dosym] + fpocz);
    if sym = thensym then pobsym else błąd(16);
    lr1 := lr; gen(skw, 0, 0);
    instrukcja(fpocz); kod[lr1].a := lr
  end else
  if sym = beginsym then
  begin pobsym; instrukcja([średnik, endsym] + fpocz);
    while sym in [średnik] + poczinstr do
    begin
      if sym = średnik then pobsym else błąd(10);
      instrukcja([średnik, endsym] + fpocz)
    end;
```

```
        if sym = endsym then pobsym else błąd(17)
    end else
    if sym = whilesym then
    begin lr1 := lr; pobsym; warunek([dosym] + fpocz);
        lr2 := lr; gen(skw, 0, 0);
        if sym = dosym then pobsym else błąd(18);
        instrukcja(fpocz); gen(skb,0, lr1); kod [lr2].a := lr
    end;
    test(fpocz, [   ], 19)
  end [instrukcja};

begin [blok] dx := 3; tx0 := tx;
  tablica [tx].adr := lr; gen(skb, 0, 0);
  if poz > pozmax then błąd(32);
  repeat
    if sym = constsym then
    begin pobsym;
        repeat deklstałej;
            while sym = przec do
                begin pobsym; deklstałej
                end;
            if sym = średnik then pobsym else błąd(5)
        until sym ≠ ident
    end;
    if sym = varsym then
    begin pobsym;
        repeat deklzmiennej;
            while sym = przec do
                begin pobsym; deklzmiennej
                end;
            if sym = średnik then pobsym else błąd(5)
        until sym ≠ ident;
    end;
    while sym = procsym do
    begin pobsym;
        if sym = ident then
            begin wprowadź(procedura); pobsym
            end
        else błąd(4);
        if sym = średnik then pobsym else błąd(5);
        blok(poz + 1, tx, [średnik] + fpocz);
        if sym = średnik then
            begin pobsym; test(poczinstr + [ident, procsym], fpocz,6)
            end
```

```
          else błąd(5)
      end;
      test(poczinstr + [ident], poczdekl, 7)
   until ¬ (sym in poczdekl);
   kod [tablica [tx0].adr].a := lr;
   with tablica [tx0] do
      begin adr := lr; {adres początkowy kodu}
      end;
   lr0 := lr; gen(alo, 0, dx);
   instrukcja([średnik, endsym] + fpocz);
   gen(opr, 0, 0); {powrót}
   test(fpocz, [ ], 8);
   drukujkod;
end {blok};

procedure interpretuj;
   const długstosu = 500;
   var p, b, t: integer; {rejestry: programowy, bazowy, stosowy}
      i: rozkaz; {rejestr rozkazu}
      s: array [1 ..długstosu] of integer; {stos – pamięć z danymi}

   function baza(p: integer): integer;
      var b1: integer;
   begin b1 := integer; {znajdź bazę segmentu leżącego p poziomów niżej}
      while p > 0 do
         begin b1 := s [b1]; p := p−1
         end;
      baza := b1
   end {baza};

begin writeln('POCZĄTEK INTERPRETACJI');
   t := 0; b := 1; p := 0;
   s[1] := 0; s[2] := 0; s[3] := 0;
   repeat i := kod[p]; p := p+1;
      case i.cz of
    sta: begin t := t+1; s[t] := i.a
         end;
    opr: case i.a of {operator}
         0: begin {powrót}
               t := b−1; p := s[t+3]; b := s[t+2];
            end;
         1: s[t] := −s[t];
         2: begin t := t−1; s[t] := s[t] + [t+1];
            end;
```

```
          3: begin t := t-1; s[t] := s[t] - s[t + 1]
             end;
          4: begin t := t-1; s[t] := s[t] * s[t + 1]
             end;
          5: begin t := t-1; s[t] := s[t] div s[t + 1]
             end;
          6: s[t] := ord(odd(s[t]));
          8: begin t := t-1; s[t] := ord(s[t] = s[t + 1])
             end;
          9: begin t := t-1; s[t] := ord(s[t] ≠ s[t + 1])
             end;
         10: begin t := t-1; s[t] := ord(s[t] < s[t + 1])
             end;
         11: begin t := t-1; s[t] := ord(s[t] ≥ s[t + 1])
             end;
         12: begin t := t-1; s[t] := ord(s[t] > s[t + 1])
             end;
         13: begin t := t-1; s[t] := ord(s[t] ≤ s[t + 1])
             end;
        end;
     ład:    begin t := t+1; s[t] := s[baza(i.p) + i.a]
             end;
     pam:    begin s[baza(i.p) + i.a] := s[t]; writeln(s[t]); t := t-1
             end;
     pro:    begin {nowy blok}
                s[t+1] := baza(i.p); s[t+2] := b; s[t+3] := p;
                b := t+1; p := i.a
             end;
     prz:    t := t+i.a;
     skb:    p := i.a;
     skw:    begin if s[t] = 0 then p := i.a; t := t-1
             end
     end {case}
   until p=0
   write('KONIEC INTERPRETACJI');
end {interpretuj};
begin {program główny}
   for ch := 'A' to ';' do ssym[ch] := żaden;
   słowo[1] := 'BEGIN    ';   słowo[2] := 'CALL       ';
   słowo[3] := 'CONST    ';   słowo[4] := 'DO         ';
   słowo[5] := 'END      ';   słowo[6] := 'IF         ';
   słowo[7] := 'ODD      ';   słowo[8] := 'PROCEDURE  ';
   słowo[9] := 'THEN     ';   słowo[10] := 'VAR       ';
   słowo[11] := 'WHILE   ';
```

```
   zsym[1] := beginsym;        zsym[2] := callsym;
   zsym[3] := constsym;        zsym[4] := dosym;
   zsym[5] := endsym;          zsym[6] := ifsym;
   zsym[7] := oddsym;          zsym[8] := procsym;
   zsym[9] := thensym;         zsym[10] := varsym;
   zsym[11] := whilesym;
   ssym['+'] := plus;          ssym['-'] := minus;
   ssym['*'] := razy;          ssym['/'] := dziel;
   ssym['('] := lnaw;          ssym[')'] := pnaw;
   ssym['='] := równe;         ssym[','] := przec;
   ssym['.'] := kropka;        ssym['≠'] := różne;
   ssym['<'] := mniejsze;      ssym['>'] := większe;
   ssym['⩽'] := mnrówne;       ssym['⩾'] := wrówne;
   ssym[';'] := średnik;
   mnem[sta] := 'STA';         mnem[opr] := 'OPR';
   mnem[ład] := 'ŁAD';         mnem[pam] := 'PAM';
   mnem[pro] := 'PRO';         mnem[alo] := 'PRZ';
   mnem[skb] := 'SKB';         mnem[skw] := 'SKW';
   poczdekl := [constsym, varsym, procsym];
   poczinstr := [beginsym, callsym, ifsym, whilesym];
   poczczyn := [ident, liczba, lnaw];
   page(output); błędy := false;
   lz := 0; lr := 0; dl := 0; ch := ' '; kk := al; pobsym;
   blok(0, 0, [kropka]+poczdekl+poczinstr);
   if sym≠kropka then błąd(9);
   if błędy then write('BŁĘDY W PROGRAMIE PL/0') else interpretuj;
99: writeln;
end.
```

Ćwiczenia

5.1

Rozważmy następującą składnię:

$S ::= A$
$A ::= B \mid \textbf{if}\ A\ \textbf{then}\ A\ \textbf{else}\ A$
$B ::= C \mid B+C \mid +C$
$C ::= D \mid C*D \mid *D$
$D ::= x \mid (A) \quad \mid -D$

Które symbole są końcowe, a które pomocnicze? Określ zbiory *pierw* (X) i *nast* (X) dla każdego symbolu pomocniczego X. Skonstruuj ciąg bezpośrednich wyprowadzeń dla następujących zdań:

$x+x$
$(x+x)*(+-x)$
$(x*-+x)$
if $x+x$ **then** $x*x$ **else** $-x$
if x **then if** $-x$ **then** x **else** $x+x$ **else** $x*x$
if $-x$ **then** x **else if** x **then** $x+x$ **else** x

5.2

Czy gramatyka z ćwiczenia 5.1 spełnia reguły ograniczające 1 i 2 dla rozbioru generacyjnego przy czytaniu z wyprzedzeniem o jeden symbol? Jeśli nie, to znajdź składnię równoważną, spełniającą te reguły. Przedstaw tę składnię w postaci diagramu składni i w postaci struktury danych zastosowanej w programie 5.3.

5.3

Powtórz ćwiczenie 5.2 dla następującej składni:

$S ::= A$
$A ::= B$ | **if** C **then** A | **if** C **then** A **else** A
$B ::= D = C$
$C ::=$ **if** C **then** C **else** C | D

Wskazówka: w celu zastosowania rozbioru generacyjnego z wyprzedzeniem o jeden symbol może okazać się konieczne usunięcie lub zastąpienie pewnych konstrukcji gramatyki innymi.

5.4

Rozważmy problem rozbioru generacyjnego zdań języka o następującej składni:

$S ::= A$
$A ::= B + A \mid DC$
$B ::= D \mid D * B$
$D ::= x \mid (C)$
$C ::= +x \mid -x$

Na podstawie wyprzedzenia o co najwyżej ile symboli będzie można dokonywać rozbioru zdań, zgodnie z powyższą składnią?

5.5

Przekształć opis języka PL/0 (rys. 5.4) na równoważny mu zbiór produkcji w notacji BNF.

5.6

Napisz program określający zbiory symboli *pierw*(*S*) i *nast*(*S*) dla każdego symbolu pomocniczego *S* z danego zbioru produkcji.
Wskazówka: wykorzystaj część programu 5.3 do konstrukcji wewnętrznej reprezentacji składni w postaci struktury danych. Następnie działaj na otrzymanej strukturze danych.

5.7

Rozszerz język PL/0 i jego kompilator o następujące konstrukcje:

(a) instrukcję warunkową postaci
⟨*instrukcja*⟩ ::= **if** ⟨*warunek*⟩ **then** ⟨*instrukcja*⟩
else ⟨*instrukcja*⟩

(b) instrukcję iteracyjną postaci
⟨*instrukcja*⟩ ::= **repeat** ⟨*instrukcja*⟩ {; ⟨*instrukcja*⟩}
until ⟨*warunek*⟩

Czy istnieją jakieś trudności, które mogłyby spowodować konieczność zmiany postaci lub interpretacji pewnych cech języka PL/0? Przy rozważaniu tego problemu nie należy rozszerzać zbioru instrukcji maszyny PL/0.

5.8

Rozszerz język PL/0 i jego kompilator o procedury z parametrami. Rozważ dwa możliwe rozwiązania i wybierz jedno z nich do realizacji.

(1) *Parametry przekazywane przez wartość.* Parametry aktualne procedury są wyrażeniami. Wartości wyrażeń zostają przypisane zmiennym lokalnym reprezentowanym przez parametry formalne wymienione w nagłówku procedury.

(2) *Parametry przekazywane przez zmienną.* Parametry aktualne są zmiennymi. Wywołanie procedury powoduje podstawienie tych parametrów na miejsce parametrów formalnych. W realizacji oznacza to przekazanie adresu parametru aktualnego do wywołanej procedury i zapamiętanie tego adresu w słowie odpowiadającym parametrowi formalnemu. Dostęp do parametrów aktualnych jest pośredni (przez przekazane adresy). Parametry przekazywane przez zmienną pozwalają odwoływać się do zmiennych zadeklarowanych na zewnątrz procedury. Reguły zakresu deklaracji mogą być więc zmienione następująco: w każdej procedurze, w sposób bezpośredni możemy odwoływać się tylko do zmiennych lokalnych; do zmiennych nielokalnych odwołujemy się wyłącznie przez parametry.

5.9

Rozszerz język PL/0 i jego kompilator o pojęcie tablicy. Załóż, że zakres indeksów zmiennej tablicowej *a* jest wskazany w jej deklaracji w sposób następujący:
var *a* (*dolneograniczenie*: *górneograniczenie*)

5.10

Zmodyfikuj kompilator PL/0 tak, aby generował kod wynikowy dla komputera, z którego możesz korzystać.
Wskazówka: aby uniknąć kłopotów związanych z dobraniem odpowiedniej reprezentacji programu wynikowego dla programu ładującego, kompilator powinien generować program w jęzku symbolicznym. W pierwszej wersji programu kompilatora nie próbuj optymalizować programu wynikowego, np. ze wzlgędu na użycie rejestrów. Ewentualne optymalizacje powinny być uwzględnione w czwartym kroku stopniowego precyzowania programu kompilatora.

5.11

Rozszerz program 5.5 do programu o nazwie „*ładnydruk*". Zadaniem programu „*ładnydruk*" będzie czytanie tekstu w języku PL/0 i drukowanie go w postaci odzwierciedlającej strukturę programu za pomocą odpowiedniego rozdzielania wierszy i stosowania wcięć. Na początku, na podstawie struktury syntaktycznej języka PL/0 określ dokładne reguły rozdzielania i wcinania; następnie wprowadź w życie te reguły przez odpowiednie nałożenie instrukcji pisania na program 5.5 (usuwając oczywiście instrukcje pisania z analizatora leksykalnego).

Literatura

5.1. Ammann U.: The Method of Structured Programming Applied to the Development of a Compiler. In: *International Computing Symposium 1973*, A. Günther et al. eds., Amsterdam, North-Holland Publishing Co. 1974, s. 93–99.

5.2. Cohen D.J., Gotlieb C.C.: A List Structure Form of Grammars for Syntactic Analysis. *Comp. Surveys*, **2**, No. 1, 1970, s. 65–82.

5.3. Floyd R.W.: The Syntax of Programming Languages-A Survey. *IEEE Trans.*, EC–13, 1964, s. 346–353.

5.4. Gries D.: *Compiler Construction for Digital Computers*. New York, Wiley 1971.

5.5. Knuth D.E.: Top-down Syntax Analysis. *Acta Informatica*, **1**, No. 2, 1971, s. 79–110.

5.6. Lewis P.M., Stearns R.E.: Syntax-directed Transduction. *J. ACM*, **15**. No. 3, 1968, s. 465–488.

5.7. Naur P. (ed.).: Report on the Algorithmic Language ALGOL 60, *ACM*, **6**, No. 1, 1963, s. 1–17.

5.8. Schorre D.V.: META II, A Syntax-oriented Compiler Writing Language. *Proc. ACM Natl. Conf.*, **19**, 1964, D 1.3.1–D 1.3.11.

5.9. Wirth N.: The Design of a PASCAL Compiler. *Software-Practice and Experience*, **1**, No. 4, 1971, s. 309–333.

Dodatek A
Zbiór znaków ASCII

y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	″	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	′	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	gs	−	=	M	]	m	}
14	so	rs	.	>	N	↑	n	~
15	si	us	/	?	O	—	o	del

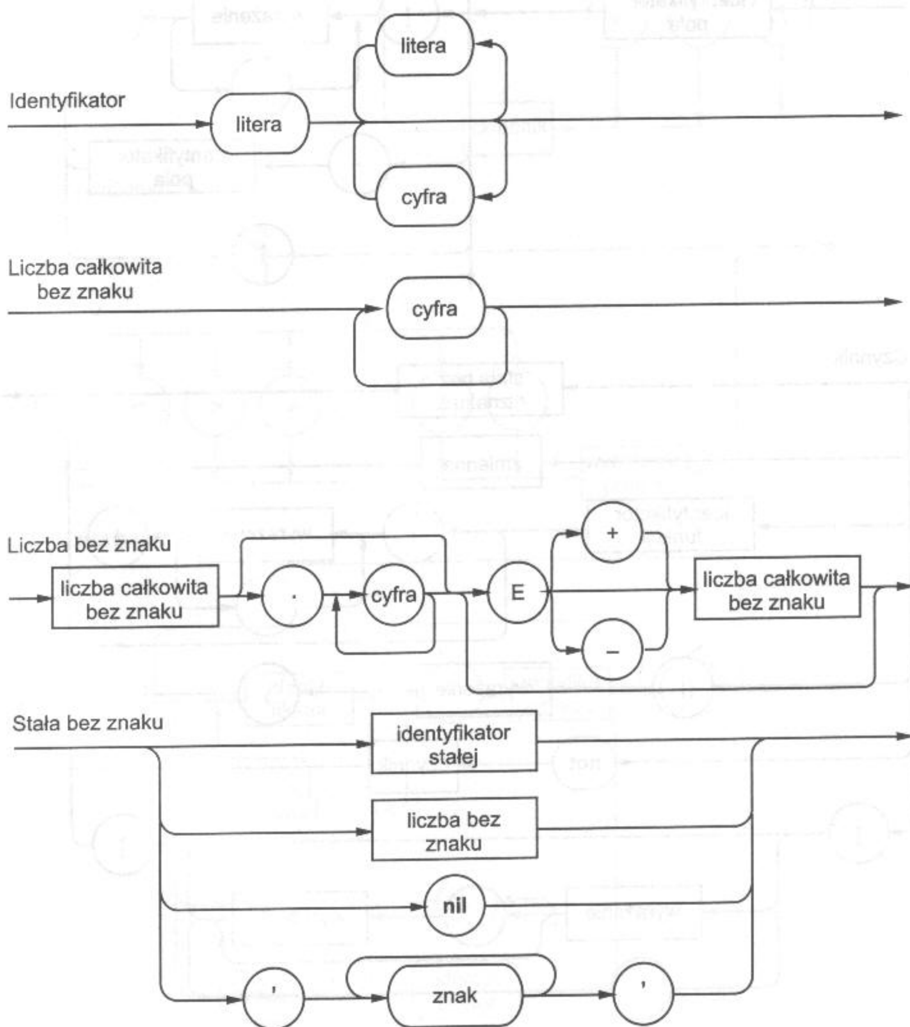
Liczbę porządkową znaku *ch* oblicza się na podstawie wartości jej współrzędnych w tablicy jako

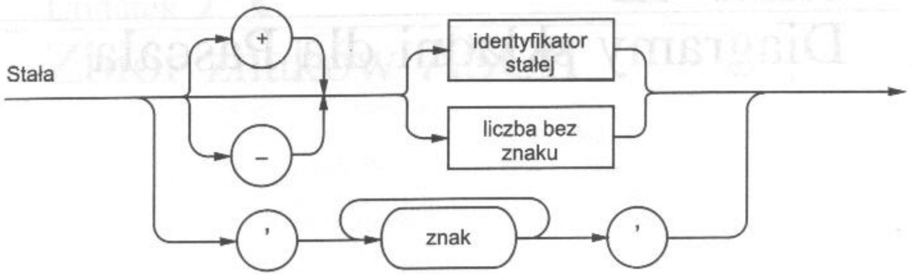
$ord(ch) = 16 * x + y$

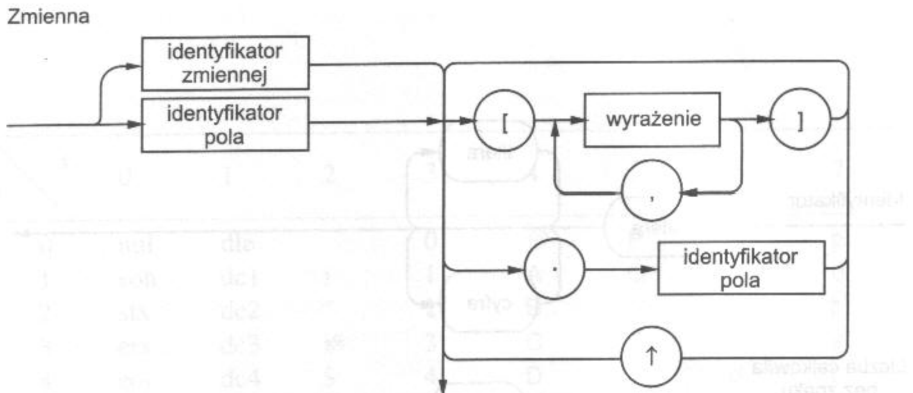
Znaki o numerach od 0 do 31 oraz 127 są tzw. **znakami sterującymi**, używanymi przy transmisji danych i sterowaniu urządzeniami. Znak o numerze 32 jest odstępem (spacją).

Dodatek B

Diagramy składni dla Pascala







Czynnik

stała bez znaku

zmienna

identyfikator funkcji

(

wyrażenie

)

,

(

wyrażenie

)

not

czynnik

[

wyrażenie

..

wyrażenie

,

]

Składnik

czynnik
*
/
div
mod
and
czynnik

Wyrażenie proste

+
−
składnik
+
−
or
składnik

Wyrażenie

wyrażenie proste
=
<
>
≠
≤
≥
in
wyrażenie proste

Lista parametrów

(
;
identy-fikator
:
identyfika-tor typu
)
function
,
var
,
procedure
identy-fikator

Typ prosty

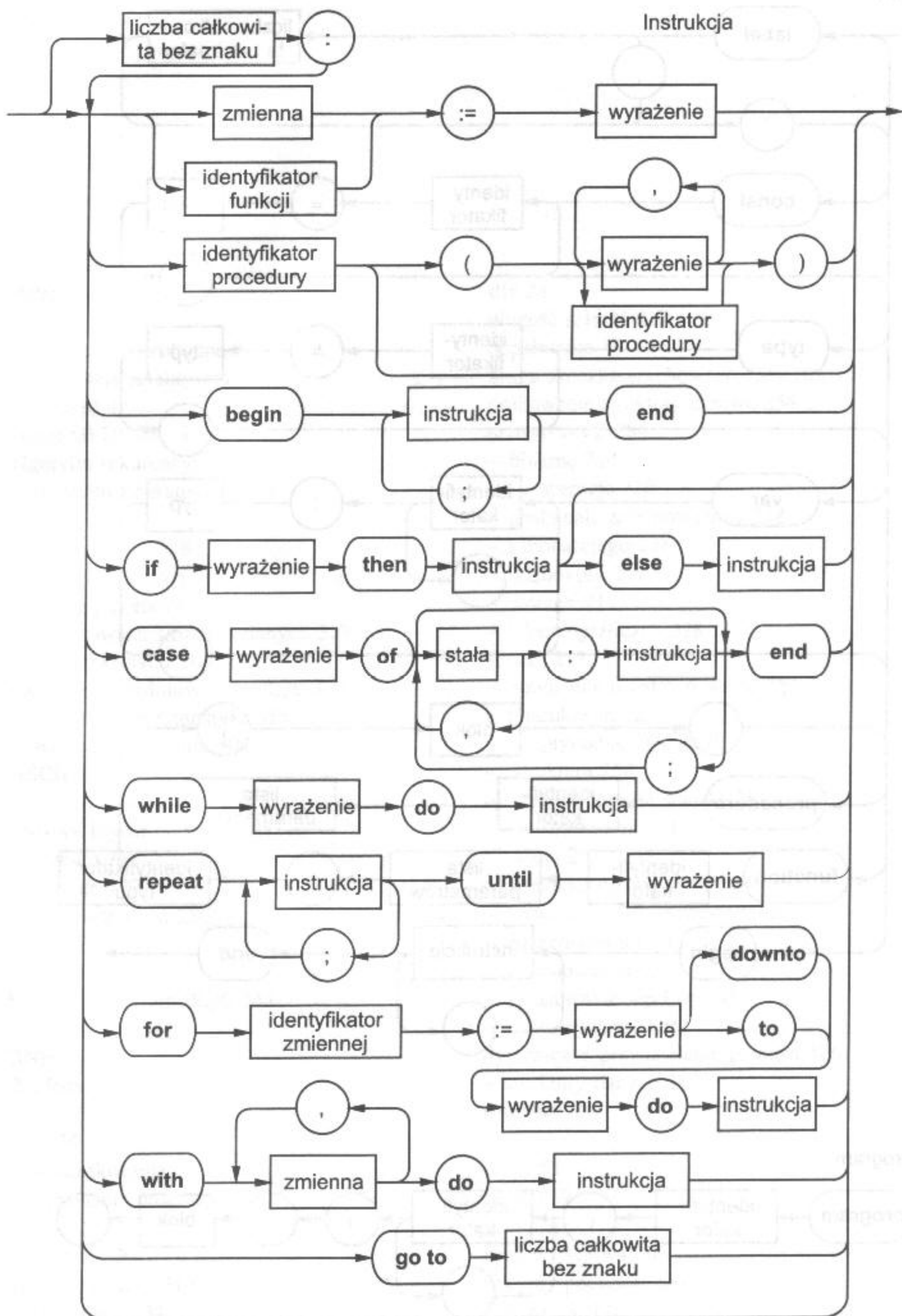
identyfikator typu
(
identy- fikator
,
)
stała
..
stała

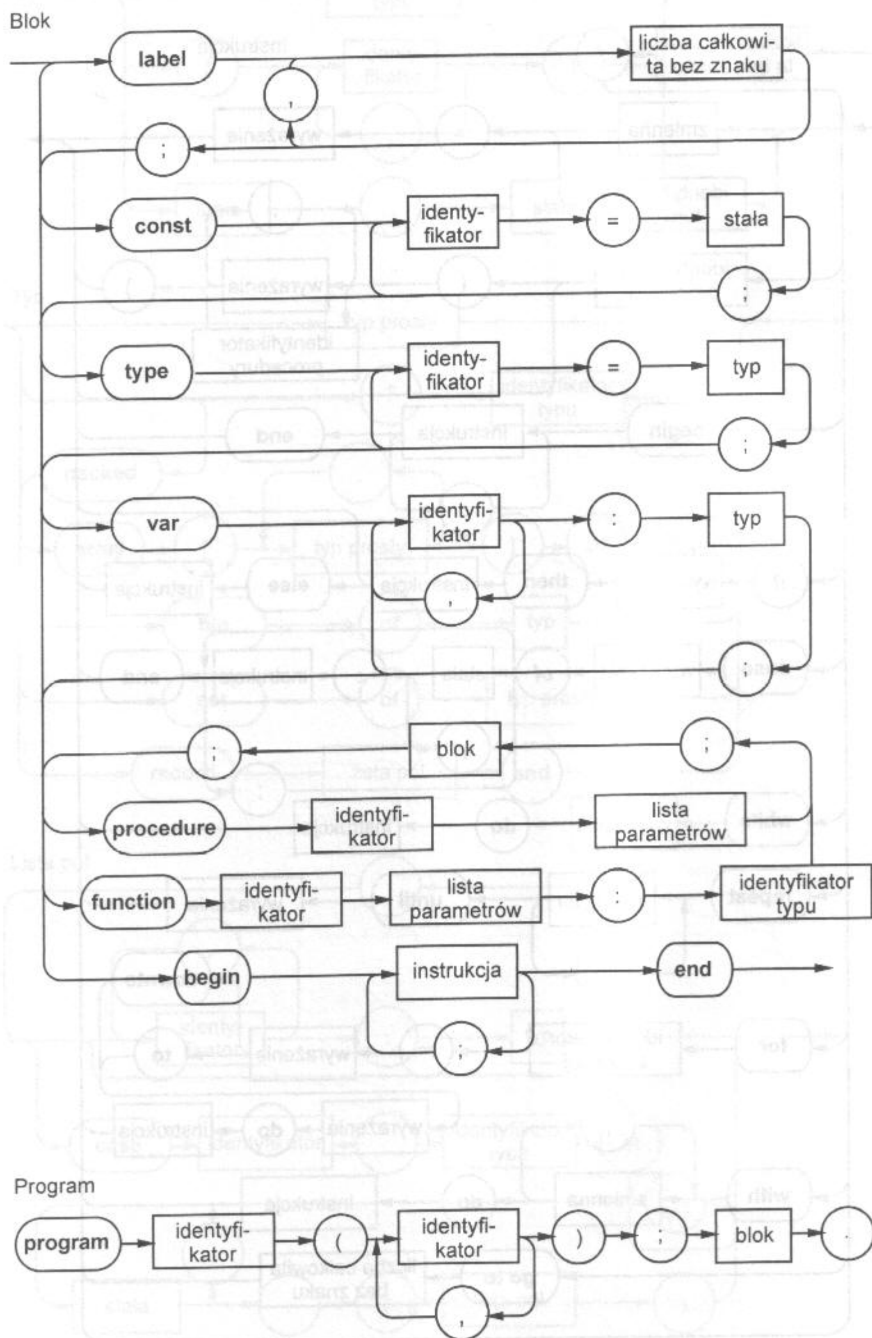
Typ

typ prosty
↑
identyfikator typu
packed
array
[
typ prosty
,
]
of
typ
file
of
typ
set
of
typ prosty
record
lista pól
end

Lista pól

;
,
identy- fikator
:
typ
case
identyfikator
:
identyfikator typu
of
;
stała
,
:
(
lista pól
)


Instrukcja
liczba całkowita bez znaku
:
zmienna
:=
wyrażenie
identyfikator funkcji
identyfikator procedury
(
,
wyrażenie
)
identyfikator procedury
begin
instrukcja
end
;
if
wyrażenie
then
instrukcja
else
instrukcja
case
wyrażenie
of
stała
:
instrukcja
end
,
;
while
wyrażenie
do
instrukcja
repeat
instrukcja
until
wyrażenie
;
downto
for
identyfikator zmiennej
:=
wyrażenie
to
wyrażenie
do
instrukcja
,
with
zmienna
do
instrukcja
go to
liczba całkowita bez znaku


Blok
label
liczba całkowi-
ta bez znaku
,
;
const
identy-
fikator
=
stała
;
type
identy-
fikator
=
typ
;
var
identyfi-
kator
:
typ
,
;
blok
;
;
procedure
identyfi-
kator
lista
parametrów
function
identyfi-
kator
lista
parametrów
:
identyfikator
typu
begin
instrukcja
end
;
Program
program
identyfi-
kator
(
identyfi-
kator
)
;
blok
.
,

Skorowidz przedmiotowy

Adres powrotu 352
– względny 353
adresowanie swobodne 287
aktualizacja selektywna 190
alfa (typ) 47
Algol 60 20, 28
algorytm rekurencyjny 145
– podziału i ograniczeń 179
– rozbioru 305
– z powrotami 158
analiza składniowa 304
– – bez powrotów 304
– – sterowana strukturą danych 327
– – z powrotami 306
analizator składniowy dla PL/0 332
– – – zadanej gramatyki 313
– sterowany składnią 309
ASCII 25, 59

Bazowy rejestr adresowy 353
B-drzewo 265
– binarne 277
– – symetryczne 280
BB-drzewo 277
blok 335
błędy kompilatora PL/0 344
– składniowe 341
BNF 302, 313
Boolean 24

char 24
czas oczekiwania 57
– wykonania programu 104

Deklaracja 20
diagram składni 313
– zależności 334
div 24
długość ścieżki 213
dopełnianie 46
droga skoczka szachowego 158, 162
drukowanie struktury drzewa 254
drzewo AVL 284
– binarne 214
– –, operacje 220
– doskonale zrównoważone 218
– Fibonacciego 238
–, głębokość 211
–, korzeń 211
– leksykograficzne 228
–, liść 212
– optymalne o jednym węźle 253
– poszukiwań 222
– – optymalne 249, 257
–, struktura 254
–, średnia długość ścieżki 251
– uporządkowane 211
–, waga 251
– wielokierunkowe 214, 263
– wyważone *zob.* zrównoważone
– zdegenerowane 211
– zrównoważone 237
– –, usuwanie węzłów 245
– –, wstawianie 239
dynamiczne przydzielanie pamięci 186
– struktury danych 183
dzielenie 24

eof(f) 55
eoln(f) 59

Faza wejściowa 206
fikcyjny początek 200
Fortran 28, 148

funkcja charakterystyczna zbioru 49
– haszująca *zob.* mieszająca
– mieszająca 286
– porządkująca 74
– transformująca 286

Generator odsyłaczy 228, 230
generowanie listy 192
get 56
głębokość drzewa 211

Identyfikator 41
indeks odsyłaczy 228
input 58
instrukcja „dla" 32
integer 24
interpretator 352
ISO 26, 59

Język bezkontekstowy 303

Klucz obiektu 75
kod 352
– wynikowy 355
kodowanie mieszające 284
kolejka 192
kolizja 285
kompilator 301
konstruktor 22
kopiec 90, 137
korzeń 211
krzywa Hilberta 151
– Sierpińskiego 154

Liczba 41
licznik instrukcji 352
lista 192
– bieżąca 256
–, generowanie 192
– kolejna 256
– liniowa 192, 196
–, przeglądanie 194
– reorganizowana 195
– uporządkowana 195
–, usuwanie 194
–, wstawianie 193, 196
–, wyszukiwanie 195
liść 212

Łańcuch bieżący 256
– kolejny 256
– statyczny 354
łącze dynamiczne 353
łączenie jednofazowe 107
– naturalne 111
– *N*-kierunkowe 118
– proste 105, 110
– wyważone 107
– – wielokierunkowe 118

Malejące przyrosty 86
maszyna PL/0 352
mediana 101
metasymbole 302, 321
metoda prosta 76
– prostego wstawiania 78, 103
– – wybierania 81, 103
– prostej zamiany 103
– przeglądania listy 203
– Shella 87
– strukturalizacji typów 22
miara efektywności 76
mod 25
model abstrakcyjny 17

Nadmiar 271, 287
napis 111
new(*p*) 188
nil 188
notacja beznawiasowa 355
– BNF 302, 313
– przedrostkowa 220
– przyrostkowa 220, 355
– RBNF 320
– wrostkowa 220, 355

Obiekt rekurencyjny 145
obszar nadmiarowy 287
odniesienie 186, 221
operacja konkatenacji 51
operacje na zbiorach 41
– podstawowe 192
operatory 22
– boolowskie 25
– konwersji typów 22
– plikowe 53
– porównania 22
– przypisania 22
optymalne drzewo poszukiwań 249, 257
optymalny wybór 175
ord(*x*) 26, 32
output 58

pack 48
pamięć jednopoziomowa 277
– programu 352

Pascal 19
pierwszy(*x*) 54
plik indeksowany 57
– o bezpośrednim dostępie 57
– sekwencyjny 52
– z podstrukturami 56
PL/0 328
–, lokalność obiektów 331
–, maszyna 352
poddrzewo 211
podstruktury 56
podział 94
pole znacznikowe 37
porządek poprzeczny 220
– wsteczny 220
– wzdłużny 220
powrót 158
pozycja 228
problem ośmiu hetmanów 163
– trwałego małżeństwa 168
procedura bezpośrednio rekurencyjna 146
– pośrednio rekurencyjna 146
– rekurencyjna 146, 221
procedury współbieżne 141
produkcje 302
program redagowania pliku 66
– rekurencyjny 151
prosta zamiana 82
proste wstawianie 77
– wybieranie 80
przeglądanie drzewa 220
– listy 203
przesiewanie 91
przestawianie 201
przesunięcie 353
przeszukiwanie drzewa z wstawianiem 223, 234
– listy 196
– – uporządkowanej 198
– – z przestawianiem 201
przewodnik 205
przodek 211
put(*x*) 54, 56

read 55
readln 59
real 24
referencja *zob.* odniesienie
reguła postępowania bez paniki 342
– słów kluczowych 342
– zastępowania 302
rekord 32
rekordy z wariantami 37
rekurencja *zob.* rekursja
rekurencyjne typy danych 183
rekursja 145, 184
rejestr wierzchołka stosu 352
reprezentacja danych 18
– rekordów 48
– tablic 45
– zbiorów 49
reset(*x*) 54
reszta(*x*) 52
rewrite(*x*) 53
rozbiór 304, 305
– ukierunkowany celem 310
rozdzielanie serii 126, 127, 128
– – początkowych 136
rozszerzenie zakresu 188
równoważenie 272

Samoorganizujące się przeszukiwanie listy 201
SBB-drzewo 280
segment fizyczny 57
– logiczny 57
sekwencyjne przetwarzanie tekstu 66
selektor 22, 28
semantyka języka 328
seria 112
– fikcyjna 128
– początkowa 136
silnia 146
składnia 302
–, konstrukcja diagramu 309
składnik 184
skorowidz 196
sondowanie 288
– kwadratowe 288
– liniowe 287, 293
sortowanie 73, 104
– przez kopcowanie 90, 93, 105
– – łączenie 106, 111
– – – naturalne 111
– – podział 94
– – prostą zamianę 82
– – proste wybieranie 80, 105
– – – wstawianie 77, 104
– Shella 87, 105
– szybkie 94, 105
– tablic 74, 76, 103
– topologiczne 203
– za pomocą malejących przyrostów 86
stan 53
stopień 212
stopniowe precyzowanie 66
stos 98

stóg *zob.* kopiec
struktura danych 183
– drzewa 254
– drzewiasta 211
– językowa 301
– semantyczna 309
– zbioru 39
struktury współdzielone 187
symbole 41, 301
– końcowe 302
– odgradzające 342
– pomocnicze 302

Tablica 28
tekst 58
transformacje kluczy 285
typy danych 20
–, indeksujący 28
– –, okrojony 27
– –, podstawowy 21, 28, 50
– –, prosty 24
– –, skalarny 21
– – –, standardowy 21, 24
– –, złożony 32
– – związane 188

unpack 48
upakowywanie 47
usuwanie elementu z listy 194
– kolizji 285
– węzłów z drzewa zrównoważonego 245
– z drzewa 232

Ważona długość ścieżki poszukiwania 249
węzeł wewnętrzny 212
wiązanie bezpośrednie 287
wiersz bieżący 66
write 55
writeln(f) 59
wskaźnik 186
współdzielenie pamięci 190
wstawianie elementu do listy 192, 193
– – – –, proste 196
– – na początek listy 192
– połówkowe 78, 105
– w drzewach zrównoważonych 239
wybór reprezentacji 18
wyrażenie 184
– indeksowe 29
wyróżnik typu 37
wysokość 211
wyszukiwanie w liście 195
wyważanie *zob.* równoważenie
wzbogacanie programu 327

Zależności kontekstowe 334
zbiór 39
– znaków ASCII 372
zmienna statyczna 183

Żywopłot 281

Skorowidz programów

PROGRAM 1.1. Obliczanie potęg liczby 2 32
PROGRAM 1.2. Analizator leksykalny 42
PROGRAM 1.3. Wyczytaj liczbę rzeczywistą 61
PROGRAM 1.4. Wpisz liczbę rzeczywistą 63
PROGRAM 2.1. Sortowanie przez proste wstawianie 78
PROGRAM 2.2. Sortowanie przez wstawianie połówkowe 79
PROGRAM 2.3. Sortowanie przez proste wybieranie 81
PROGRAM 2.4. Sortowanie bąbelkowe 83
PROGRAM 2.5. Sortowanie mieszane 85
PROGRAM 2.6. Sortowanie metodą Shella 87
PROGRAM 2.7. Przesiewanie 91
PROGRAM 2.8. Sortowanie przez kopcowanie 93
PROGRAM 2.9. Podział 95
PROGRAM 2.10. Sortowanie szybkie 97
PROGRAM 2.11. Nierekurencyjna wersja sortowania szybkiego 98
PROGRAM 2.12. Znajdź *k*-ty element 102
PROGRAM 2.13. Sortowanie przez łączenie proste 110
PROGRAM 2.14. Sortowanie przez łączenie naturalne 114
PROGRAM 2.15. Sortowanie przez łączenie wyważone 122
PROGRAM 2.16. Sortowanie polifazowe 133
PROGRAM 2.17. Rozdzielanie serii początkowych przez kopiec 139
PROGRAM 3.1. Krzywe Hilberta 153
PROGRAM 3.2. Krzywe Sierpińskiego 156
PROGRAM 3.3. Droga skoczka szachowego 160
PROGRAM 3.4. Osiem hetmanów 166
PROGRAM 3.5. Osiem hetmanów 168
PROGRAM 3.6. Trwałe małżeństwa 173
PROGRAM 3.7. Optymalny wybór 178
PROGRAM 4.1. Proste wstawianie do listy 196
PROGRAM 4.2. Sortowanie topologiczne 209
PROGRAM 4.3. Skonstruuj drzewo doskonale zrównoważone 217
PROGRAM 4.4. Przeszukiwanie drzewa z wstawianiem 225
PROGRAM 4.5. Generator odsyłaczy 228
PROGRAM 4.6. Znajdowanie optymalnego drzewa poszukiwań 257
PROGRAM 4.7. Poszukiwanie z wstawianiem i usuwaniem w B-drzewie 272
PROGRAM 4.8. Generator odsyłaczy przy zastosowaniu tablicy kodów mieszających 289

PROGRAM 5.1. Program analizatora dla gramatyki z przykładu 315
PROGRAM 5.2. Analizator języka 322
PROGRAM 5.3. Translator języka 325
PROGRAM 5.4. Analizator języka PL/0 335
PROGRAM 5.5. Analizator PL/0 z reagowaniem na błędy składniowe 347
PROGRAM 5.6. Kompilator języka PL/0 358

WNT. Warszawa 2002. Wyd. VI
Ark. wyd. 27,4. Ark. druk. 24,0
Symbol Et/83741/WNT
Zakład Poligraficzno-Wydawniczy
„POZKAL"

„W dzisiejszych czasach programowanie stało się dziedziną wiedzy, której opanowanie ma zasadnicze znaczenie przy rozwiązywaniu wielu problemów inżynierskich, a którą przy tym można badać i prezentować w sposób naukowy. Programowanie awansowało – przestało być rzemiosłem, a stało się dyscypliną akademicką".

NIKLAUS WIRTH

Programowanie to konstruktywna i twórcza działalność. W książce przedstawiono systematyczne podejście do rozwiązywania problemów programistycznych o charakterze zarówno naukowym, jak i praktycznym. W prezentowanych metodach szczególną uwagę zwrócono na zasady strukturalizacji danych w procesie tworzenia programu z uwzględnieniem różnych dziedzin zastosowań. Przedstawiono także algorytmy sortowania, algorytmy rekurencyjne, dynamiczne struktury danych oraz struktury językowe i kompilatory.

Cena 68,- zł
ISBN 83-204-2740-1