ALGORYTMY I STRUKTURY DANYCH
WYKŁAD IV (materiały pomocnicze)

Struktury danych, listy i słowniki

Polsko Japońska Wyższa Szkoła Technik Komputerowych

Warszawa, 23 listopada 2008

Plan wykładu:

- lista:
  - operacje na listach,
  - implementacje list,
- stos:
  - algorytm DFS przechodzenia grafu,
  - algorytm obliczania wartości wyrażeń arytmetycznych,
- kolejka:
  - algorytm BFS przechodzenia grafu,

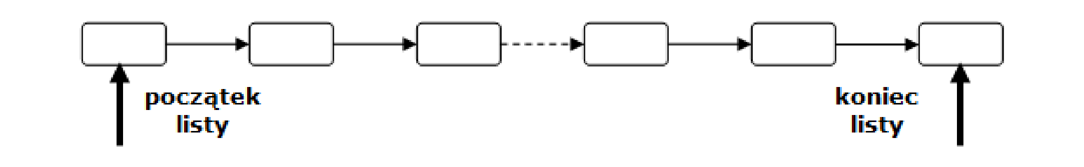
Plan wykładu c.d.:

- słownik:
  - implementacja w tablicy haszującej,
  - drzewa poszukiwań binarnych BST,
  - drzew poszukiwań binarnych AVL,
- sortowanie przy użyciu drzew poszukiwań binarnych.

Lista

Lista – operacje na listach

Idea. *Listą* nazywamy liniową strukturę danych, w której potrafimy wyróżnić element pierwszy (początek listy) oraz element ostatni (koniec listy).

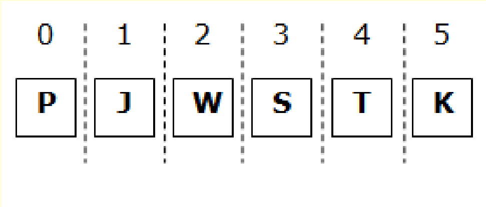


Często używane operacje:

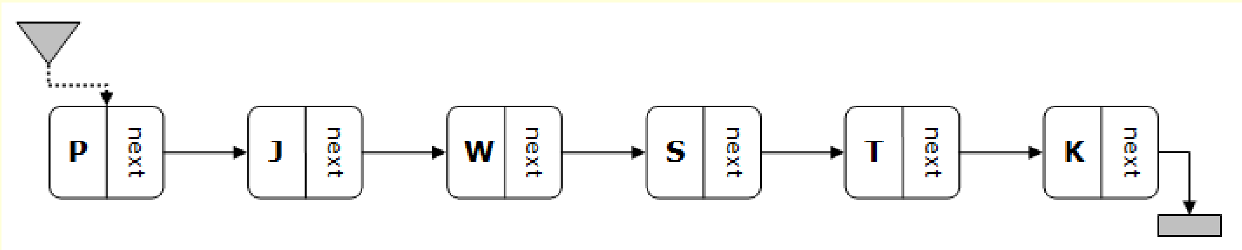
- sprawdzenie, czy lista jest pusta,
- sprawdzenie, czy element znajduje się w liście,
- wstawienie elementu na początek/koniec/w dowolne miejsce listy,
- usunięcie elementu z listy,
- utworzenie podlisty,
- połączenie list,
- itd.

Lista – implementacja list

Implementacja statyczna. Tablica, w której zgodnie z porządkiem listy, zapisane są jej elementy, np.:



Implementacja dynamiczna. Struktura dowiązaniowa, w której zgodnie z porządkiem listy, zapisane są jej elementy, np.:

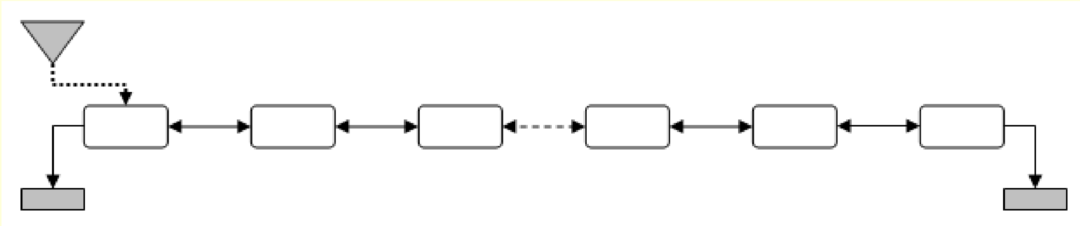


Pytanie. Jaka jest złożoność przedstawionych operacji na listach w zależności od rodzaju implementacji struktury danych?

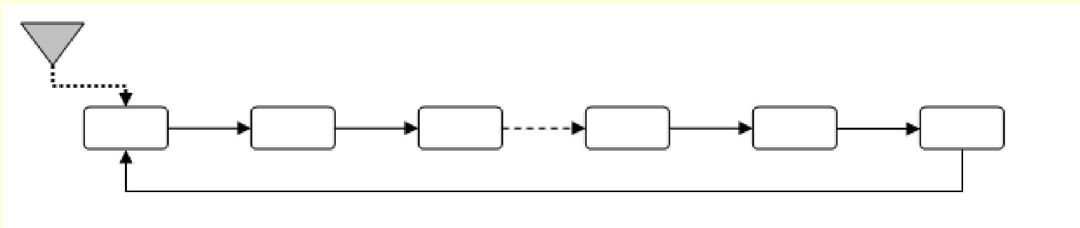
Lista – implementacja list

Dla struktury dowiązaniowej wyróżniamy także listy:

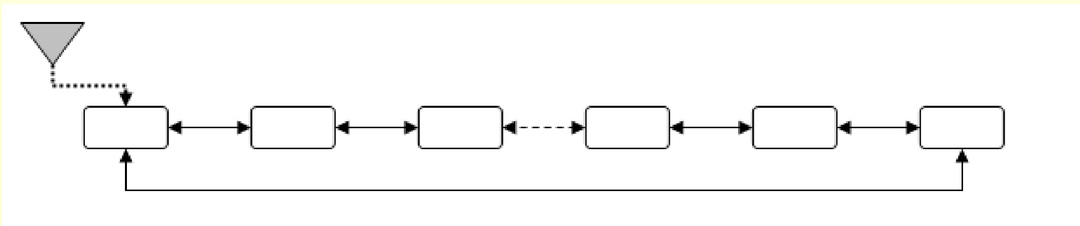
- dwustronne



- cykliczne



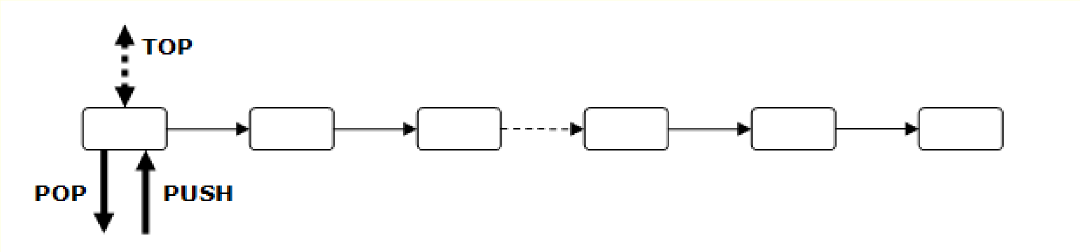
- dwustronne i cykliczne.



Stos

Stos

Idea:



Specyfikacja stosu:

- sygnatura:
  - $\langle E \cup S \cup \{true, false\}, empty, top, push, pop\rangle$, gdzie S jest uniwersum multizbiorów,
  - $empty : S \to \{true, false\}$,
  - $top : S \to E$,
  - $push : S \times E \to S$,
  - $pop : S \to S$,

Stos

Specyfikacja stosu (c.d.):

- aksjomaty:
  - $top\,(push\,(s,e)) = e$,
  - $pop\,(push\,(s,e)) = s$,
  - $\neg empty\,(s) \Rightarrow push\,(pop\,(s)\,, top\,(s)) = s$,
  - program `while (!empty(s)) s=pop(s);` ma własność stopu.

Twierdzenie. *Dowolna struktura, która spełnia aksjomaty specyfikacji stosu jest izomorficzna z pewną standardową strukturą stosów.*

Stos – algorytm DFS przechodzenia grafu

Rozwiązanie.

```
void DFS(Graph g, Vertex v) {
   Stack s; // stos jest początkowo pusty
   Vertex tmp;

   s=wstaw_i_oznacz_wierzchołek(s,v);
   while (!empty(s)) {
      tmp=top(s);
      s=pop(s);
      wypisz_wierzchołek(tmp);
      wstaw_i_oznacz_nieoznaczone_wierzchołki_sąsiednie(s,tmp,g);
         // wstawianie odbywa się w ustalonym porządku
   }
}
```

Pytanie. Jaka jest złożoność czasowa algorytmu DFS względem operacji na stosie?

Zadanie. Prześledź działanie algorytmu DFS na dowolnym 8-wierzchołkowym grafie wejściowym, którego wierzchołki etykietowane są literami, a kolejność wstawiania wierzchołków nieoznaczonych do stosu jest zgodna z porządkiem alfabetycznym etykiet.

Stos – algorytm obliczania wartości wyrażeń arytmetycznych

Założenia. Obliczamy wartość niepustego, poprawnego, w pełni nawiasowanego wyrażenia arytmetycznego zdefiniowanego nad alfabetem znaków

$$\Sigma = \{0, 1, 2, \ldots, 9\} \cup \{+, -, \cdot, /\} \cup \{(,)\}\,.$$

Idea algorytmu. Niech w będzie wyrażeniem wejściowym, wtedy:

- utwórz dwa puste stosy arg i opr, odpowiednio *stos argumentów* i *stos operatorów*,
- dopóki w nie jest wyrażeniem pustym wczytaj z początku wyrażenia w *token* t, jeżeli:
  - t jest liczbą, wykonaj $push\,(arg, t)$,
  - t jest operatorem, wykonaj $push\,(opr, t)$,
  - t jest znakiem), wykonaj kolejno
    $x = top\,(arg)$, $arg = pop(arg)$,
    $y = top\,(arg)$, $arg = pop(arg)$,
    $push\,(arg, y\ top\,(opr)\ \ x)$, $pop\,(opr)$,
  - usuń token t z początku wyrażenia w,
- wartość wyrażenia w jest równa $top\,(arg)$.

Stos – algorytm obliczania wartości wyrażeń arytmetycznych

Zadanie. Prześledź działanie algorytmu obliczania wartości wyrażenia arytmetycznego dla dowolnego 5-cio operatorowego wyrażenia wejściowego.

Pytanie. Jaka jest złożoność czasowa przedstawionego algorytmu wyliczania wartości wyrażeń arytmetycznych mierzona liczbą operacji na obu stosach?

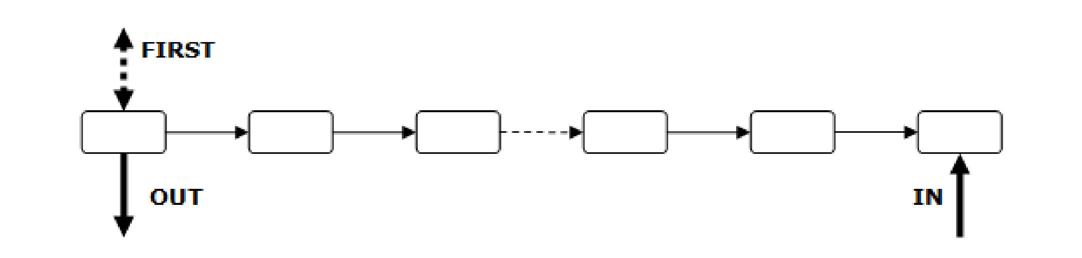
Pytanie. Jaka jest maksymalna możliwa wysokość stosu argumentów arg, dla wyrażenia zbudowanego z n argumentów?

Pytanie. Jaka jest maksymalna możliwa wysokość stosu operatorów opr, dla wyrażenia zbudowanego z n operatorów?

Kolejka

Kolejka

Idea:



Specyfikacja kolejki:

- sygnatura:
  - $\langle E \cup Q \cup \{true, false\}, empty, first, in, out\rangle$, gdzie Q jest uniwersum multizbiorów,
  - $empty : Q \to \{true, false\}$,
  - $first : Q \to E$,
  - $in : Q \times E \to Q$,
  - $out : Q \to Q$,

Kolejka

Specyfikacja kolejki (c.d.):

- aksjomaty:
  - $\neg empty\,(in\,(q,e))$,
  - $empty\,(q) \Rightarrow first\,(in\,(q,e)) = e$,
  - $empty\,(q) \Rightarrow out\,(in\,(q,e)) = q$,
  - $\neg empty\,(q) \Rightarrow first\,(in\,(q,e)) = first\,(q)$,
  - $\neg empty\,(q) \Rightarrow in\,(out\,(q)\,,e) = out\,(in\,(q,e))$,
  - program `while (!empty(q)) q=out(q);` ma własność stopu.

Twierdzenie. *Dowolna struktura, która spełnia aksjomaty specyfikacji kolejki jest izomorficzna z pewną standardową strukturą kolejek.*

Kolejka – algorytm BFS przechodzenia grafu

Rozwiązanie.

```
void BFS(Graph g, Vertex v) {
   Queue q; // kolejka jest początkowo pusta
   Vertex tmp;

   q=wstaw_i_oznacz_wierzchołek(q,v);
   while (!empty(q)) {
      tmp=first(q);
      q=out(q);
      wypisz_wierzchołek(tmp);
      wstaw_i_oznacz_nieoznaczone_wierzchołki_sąsiednie(q,tmp,g);
         // wstawianie odbywa się w ustalonym porządku
   }
}
```

Pytanie. Jaka jest złożoność czasowa algorytmu BFS względem operacji na kolejce?

Zadanie. Prześledź działanie algorytmu BFS na dowolnym 8-wierzchołkowym grafie wejściowym, którego wierzchołki etykietowane są literami, a kolejność wstawiania wierzchołków nieoznaczonych do stosu jest zgodna z porządkiem alfabetycznym etykiet.

Słownik

Słownik

Idea (model standardowy słownika):

- $\langle E \cup D \cup \{true, false\}, empty, member, insert, delete, any\rangle$, gdzie D jest uniwersum zbiorów,
- $empty\,(d) \equiv_{df} (d = \emptyset)$,
- $member\,(d, e) \equiv_{df} (e \in d)$,
- $insert\,(d, e) =_{df} (d \cup \{e\})$,
- $delete\,(d, e) =_{df} (d \setminus \{e\})$,
- $any\,(d)$, dowolna operacja, której rezultatem jest pewien element zbioru d.

Specyfikacja słownika:

- sygnatura:
  - $\langle E \cup D, empty, member, insert, delete, any\rangle$,
  - $empty : D \to \{true, false\}$,
  - $member : D \times E \to \{true, false\}$,

Słownik

Specyfikacja słownika (c.d.):

- sygnatura:
  - $insert : D \times E \longrightarrow D$,
  - $delete : D \times E \longrightarrow D$,
  - $any : D \longrightarrow E$,
- aksjomaty:
  - $member\,(d, e) \equiv P\,(d, e)$, gdzie P jest następującym programem

```
E tmp;

while (!empty(d)) {
   tmp=any(d);
   if (tmp==e) return true; else d=delete(d,tmp);
}
return false;
```

Słownik

Specyfikacja słownika (c.d.):

- aksjomaty:
  - $empty\,(d) \equiv \neg\exists e \in E\,(member\,(d,e))$,
  - $\neg empty\,(d) \Rightarrow member\,(d, any\,(d))$,
  - $member\,(insert\,(d,e)\,,e)$,
    $e \neq e' \Rightarrow member\,(d,e') \equiv member\,(d, insert\,(d,e))\,,$
  - $\neg member\,(delete\,(d,e)\,,e)$,
    $e \neq e' \Rightarrow member\,(d,e) \equiv member\,(delete\,(d,e')\,,e)\,,$
  - program `while (!empty(d)) d=delete(d,any(d));` ma własność stopu.

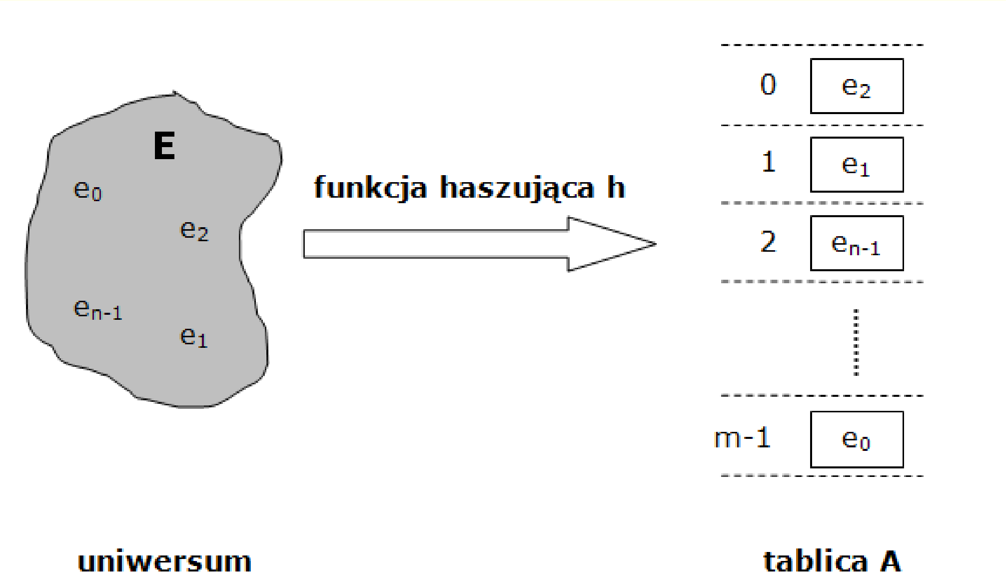
Twierdzenie. *Dowolna struktura, która spełnia aksjomaty specyfikacji słownika jest izomorficzna z pewną standardową strukturą słowników z dokładnością do operacji* $any : D \rightarrow E$.

Słownik

(implementacja w tablicy haszującej)

Słownik – implementacja w tablicy haszującej

Idea. Niech E będzie uniwersum elementów słownika d, gdzie $|E| = n$. *Tablicą haszującą* nazywamy parę (A, h), gdzie A jest m-elementową tablicą, a $h : E \rightarrow \{0, 1, \ldots, m-1\}$ *funkcją haszującą.*



Zadanie. Podaj przykład tablicy haszującej dla implementacji słownika, którego uniwersum elementów to zbiór liczb $\{0, 1, 2, \ldots, 999\}$.

Słownik – implementacja w tablicy haszującej

Trudności w implementacji. Jak dobrać długość m tablicy A? Przypadki do rozważenia:

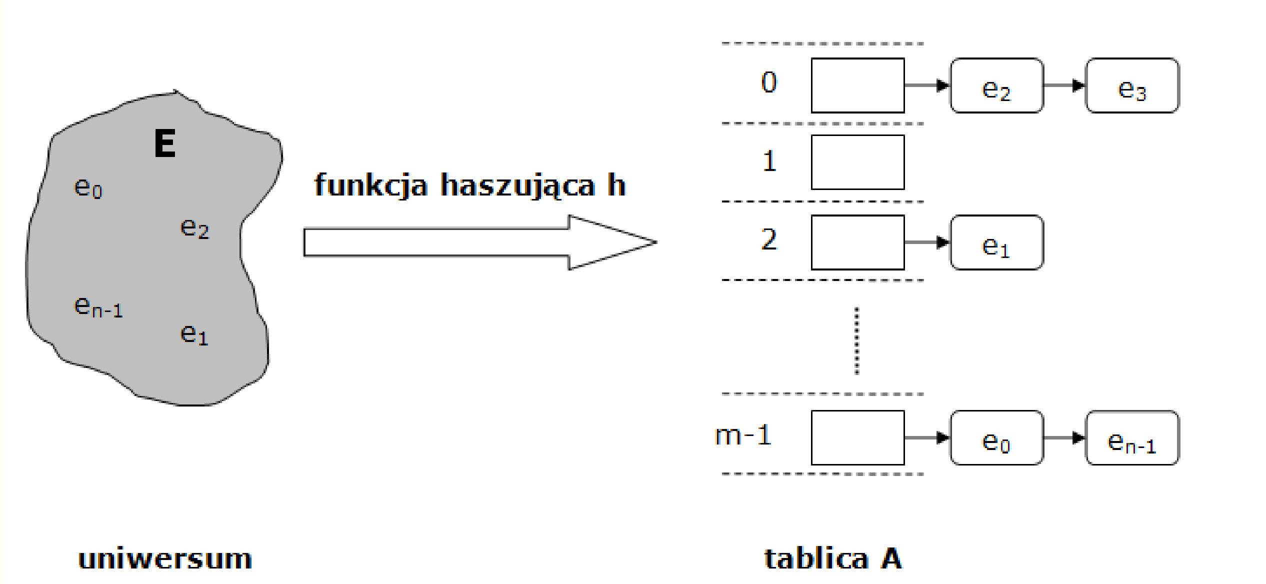
- $m > n$, wtedy istnieje funkcja haszująca h różnowartościowa, struktura tablicy haszującej jest nieefektywna pamięciowo – nie wszystkie elementy tablicy A są wykorzystywane,
- $m = n$, wtedy istnieje funkcja haszująca h różnowartościowa i „na”, dla słowników „rzadkich”, tj. gdy nie wszystkie elementy uniwersum słownika są wykorzystywane (np. słowniki językowe dla słów ograniczonej długości nad ustalonym alfabetem) struktura tablicy haszującej jest nieefektywna pamięciowo – nie wszystkie elementy tablicy A są wykorzystywane,
- $m < n$, wtedy:
  - dla słowników „rzadkich” udaje się w praktyce znaleźć funkcję haszująca h różnowartościową i „na”,
  - dla słowników „gęstych”, tj. gdy prawie wszystkie elementy uniwersum słownika są wykorzystywane, nie udaje się w praktyce znaleźć funkcji haszującej h różnowartościowej i „na” albo taka funkcja nie istnieje,

  W tym przypadku bardzo często występuje tzw. *problem kolizji*, tj. dla dwóch różnych elementów $e_1, e_2 \in E$ zachodzi $h(e_1) = h(e_2)$.

Słownik – implementacja w tablicy haszującej

Trudności w implementacji. Jak rozwiązać problem kolizji?

Rozwiązanie. W miejsce tablicy elementów uniwersum słownika stosujemy tablicę kolejek, w których to kolejkach przechowujemy wszystkie elementy, dla których wartość funkcji haszującej jest identyczna.



Słownik – implementacja w tablicy haszującej

Trudności w implementacji. Jak znaleźć właściwą funkcję haszującą?

Fakt. W przypadku ogólnym problem ten jest nadal otwarty!

Rozwiązanie. Podejście standardowe:

- ustalić odwzorowanie różnowartościowe zbioru E elementów uniwersum w zbiór liczb naturalnych, tj. dla każdego $e \in E$ ustalamy w sposób jednoznaczny liczbę naturalną i_e,
- użyć haszowania:
  - *modularnego*, gdzie $h(i_e) = i_e \bmod m$, dla m będącego liczbą pierwszą nie zbyt bliską potędze 2,
  - *przez mnożenie*, gdzie $h(i_e) = \lfloor m(a \cdot k - \lfloor a \cdot k \rfloor) \rfloor$, dla $0 < a < 1$,
  - *uniwersalnego*, *z adresowaniem otwartym*, itd.

Słownik – implementacja w tablicy haszującej

Twierdzenie. *Niech (A, h) będzie tablicą haszującą długości m z kolejkami, będącą implementacją słownika d dla uniwersum elementów E, gdzie $|E| = n$. Jeżeli koszt wyznaczenia wartości funkcji haszującej h, dla dowolnego elementu $e \in E$ jest stały, to:*

- *$member\,(d, e)$ – sprawdź, czy element e należy do kolejki $A\,[h\,(e)]$, złożoność średnia $O\left(1 + \frac{n}{m}\right)$, pesymistyczna $O\,(n)$,*
- *$insert\,(d, e)$ – jeżeli $member\,(d, e) = false$, wstaw element e do kolejki $A\,[h\,(e)]$, złożoność średnia $O\left(1 + \frac{n}{m}\right)$, pesymistyczna $O\,(n)$,*
- *$delete\,(d, e)$ – jeżeli $member\,(d, e) = true$, usuń element e z kolejki $A\,[h\,(e)]$, złożoność średnia $O\left(1 + \frac{n}{m}\right)$, pesymistyczna $O\,(n)$.*

Wniosek. Tablica haszująca może być implementacją słownika o stałym oczekiwanym czasie działania operacji słownikowych.

Wniosek. Tablica haszująca może być implementacją słownika o liniowym pesymistycznym czasie działania operacji słownikowych.

Pytanie. Jak zaimplementować operację słownikową $empty$ w przypadku tablicy haszującej tak, aby złożoność tej operacji była stała?

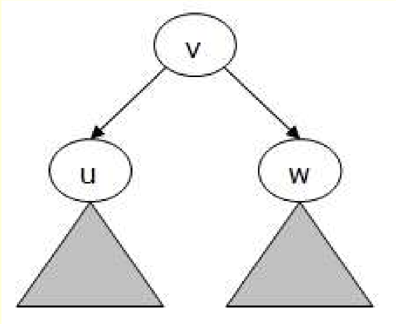
Słownik

(drzewa poszukiwań binarnych BST)

Słownik – drzewa poszukiwań binarnych BST

Definicja. *Drzewem binarnych poszukiwań BST* nazywamy drzewo binarne $G = (V_G, E_G, et)$, gdzie:

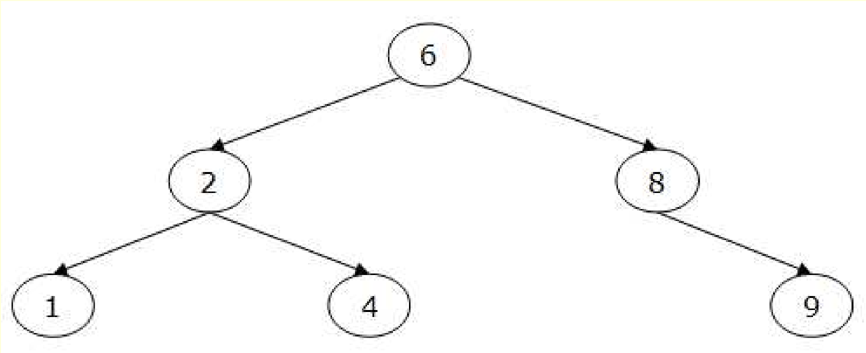
- $et : V_G \to E$ jest różnowartościową funkcją etykietowania wierzchołków i E jest pewnym niepustym, liniowo uporządkowanym zbiorem etykiet $\langle E, \leq \rangle$,
- dla każdej trójki wierzchołków u, v, w, jeżeli:
  - u jest lewym następnikiem wierzchołka v, to dla dowolnego wierzchołka x należącego do poddrzewa o korzeniu w wierzchołku u zachodzi $et\,(x) \leq et\,(v)$,
  - w jest prawym następnikiem wierzchołka v, to dla dowolnego wierzchołka y należącego do poddrzewa o korzeniu w wierzchołku w zachodzi $et\,(v) \leq et\,(y)$.



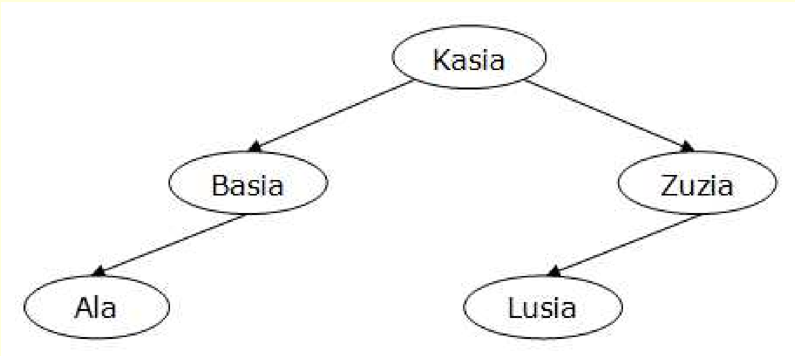
Słownik – drzewa poszukiwań binarnych BST

Przykłady:

- zbiór etykiet $\langle \mathbb{N}, \leq \rangle$:



- zbiór etykiet $\langle \Pi, \leq_{leks} \rangle$, gdzie Π jest zbiorem słów języka polskiego:

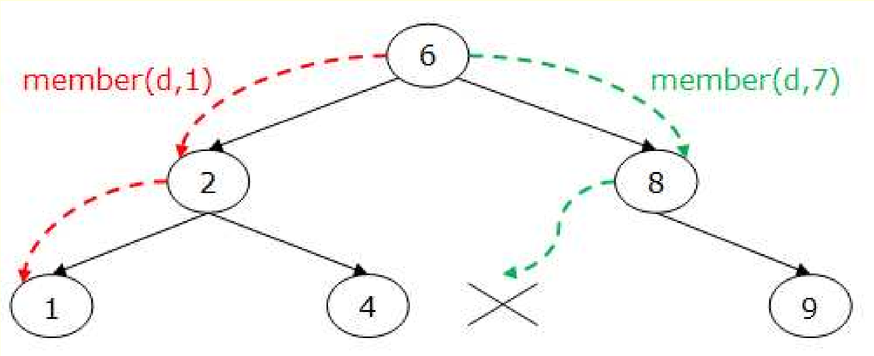


Słownik – drzewa poszukiwań binarnych BST

Operacji $member\,(d, e)$ **– idea.** Niech T będzie drzewem BST, będącym implementacją słownika d dla uniwersum elementów E, i niech e będzie elementem uniwersum E. Rozpoczynając od korzenia drzewa BST:

- sprawdź, czy dowiązanie do aktualnie rozważanego wierzchołka v jest puste, wtedy $member\,(d, e) = false$,
- sprawdź dla aktualnie rozważanego wierzchołka, czy $et\,(v) = e$, jeżeli tak, to $member\,(d, e) = true$,
- jeżeli $et\,(v) > e$, przejdź do wierzchołka $v.left$ i powtórz powyższe postępowanie,
- jeżeli $et\,(v) < e$, przejdź do wierzchołka $v.right$ i powtórz powyższe postępowanie.

Przykład. Wyszukujemy w słowniku $d = \{1, 2, 4, 6, 8, 9\}$ elementu 1 oraz 7.



Słownik – drzewa poszukiwań binarnych BST

Operacji $member\,(d, e)$ **– implementacja.**

```
bool Member(TreeNode root, E e) {

   while (root!=NULL)
      if (root.Et==e) return true;
      else if (root.Et>e) root=root.left;
         else root=root.right;

   return false;
}
```

Pytanie. Jaka jest pesymistyczna złożoność czasowa algorytmu Member względem liczby odwiedzonych wierzchołków?

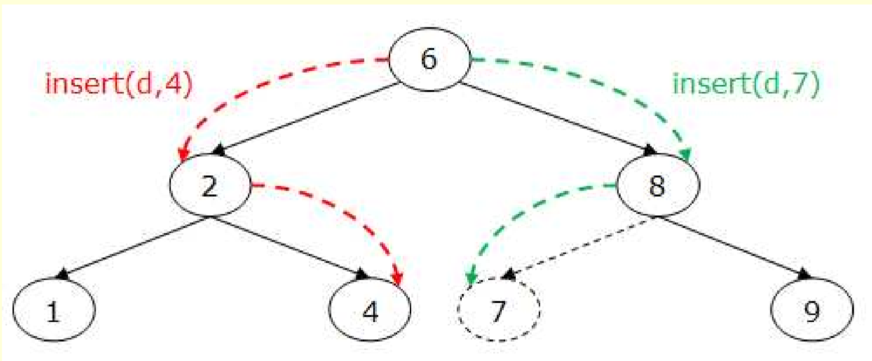
Pytanie. Jaka jest złożoność pamięciowa algorytmu Member?

Słownik – drzewa poszukiwań binarnych BST

Operacji $insert\,(d, e)$ **– idea.** Niech T będzie drzewem BST, będącym implementacją słownika d dla uniwersum elementów E, i niech e będzie elementem uniwersum E. Rozpoczynając od korzenia drzewa BST:

- sprawdź, czy dowiązanie do aktualnie rozważanego wierzchołka v jest puste, jeżeli tak, to utwórz (w aktualnym dowiązaniu) nowy wierzchołek z etykietą e
- sprawdź dla aktualnie rozważanego wierzchołka, czy $et\,(v) = e$, jeżeli tak, to zakończ działanie algorytmu,
- jeżeli $et\,(v) > e$, przejdź do wierzchołka $v.left$ i powtórz powyższe postępowanie,
- jeżeli $et\,(v) < e$, przejdź do wierzchołka $v.right$ i powtórz powyższe postępowanie.

Przykład. Wstawiamy do słownika $d = \{1, 2, 4, 6, 8, 9\}$ element 4 oraz 7.



Słownik – drzewa poszukiwań binarnych BST

Operacji $insert\,(d, e)$ **– implementacja.**

```
TreeNode Insert(TreeNode root, E e) {
   TreeNode tmp=root;

   if (root==NULL) return NewVertex(e);

   while (true) {
      if (tmp.Et==e) return root;
      if (tmp.Et>e)
         if (tmp.left!=NULL) tmp=tmp.left;
         else {
            tmp.left=NewVertex(e);
            return root;
         }
```

Słownik – drzewa poszukiwań binarnych BST

Operacji $insert\,(d, e)$ **– implementacja (c.d.).**

```
          else
             if (tmp.right!=NULL) tmp=tmp.right;
             else {
                tmp.right=NewVertex(e);
                return root;
             }
       }

       return false;
    }
```

Pytanie. Jaka pesymistyczna złożoność czasowa algorytmu Insert względem liczby odwiedzonych wierzchołków?

Pytanie. Jaka jest złożoność pamięciowa algorytmu Insert?

Słownik – drzewa poszukiwań binarnych BST

Twierdzenie. *Niech T będzie drzewem poszukiwań binarnych BST, będącym implementacją słownika d dla uniwersum elementów E, gdzie $|E| = n$. Jeżeli drzewo T powstało przez kolejne wstawienie elementów zbioru E w losowej kolejności, to oczekiwana wysokość tego drzew jest rzędu $O(\lg n)$.*

Wniosek. Oczekiwana złożoność operacji słownika $memeber$ w implementacji drzewa BST wynosi $O(\lg n)$, gdzie n jest liczbą wierzchołków drzewa.

Wniosek. Oczekiwana złożoność operacji słownika $insert$ w implementacji drzewa BST wynosi $O(\lg n)$, gdzie n jest liczbą wierzchołków drzewa.

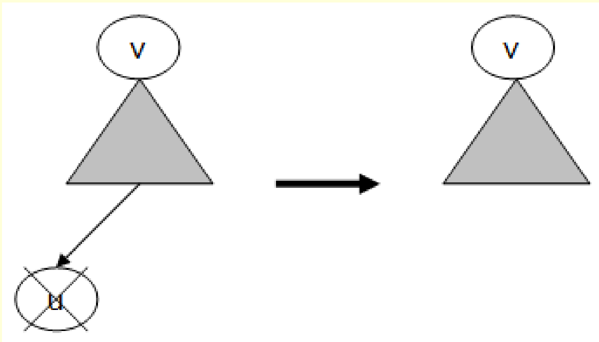
Słownik – drzewa poszukiwań binarnych BST

Operacji $delete\,(d, e)$ **– idea.** Niech T będzie drzewem BST, będącym implementacją słownika d dla uniwersum elementów E, i niech e będzie elementem uniwersum E. Rozpoczynając od korzenia drzewa BST:

- sprawdź, czy dowiązanie do aktualnie rozważanego wierzchołka u jest puste, wtedy zakończ działanie algorytmu,
- jeżeli $et\,(u) > e$, przejdź do wierzchołka $u.left$ i powtórz powyższe postępowanie,
- jeżeli $et\,(u) < e$, przejdź do wierzchołka $u.right$ i powtórz powyższe postępowanie.
- sprawdź dla aktualnie rozważanego wierzchołka, czy $et\,(u) = e$, jeżeli tak, to wykonaj jeden z trzech wariantów operacji usuwania wierzchołka:
  - wierzchołek u jest liściem w drzewie BST,
  - wierzchołek u ma tylko jednego syna w drzewie BST,
  - wierzchołek u ma dwóch synów w drzewie BST.

Słownik – drzewa poszukiwań binarnych BST

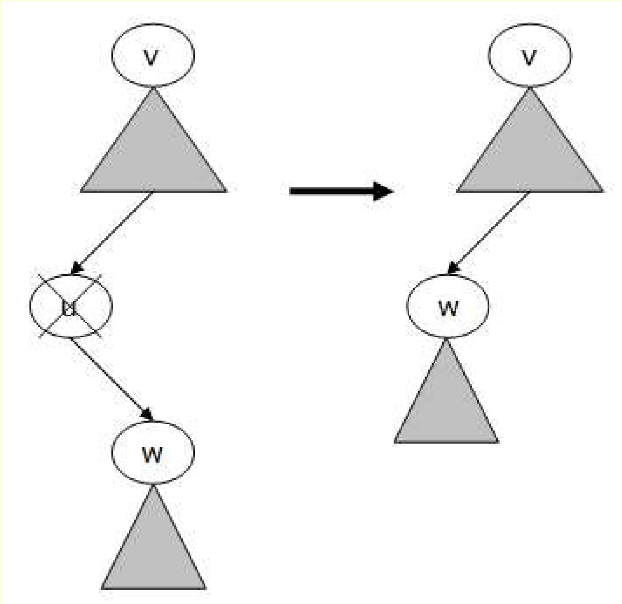
Operacji $delete\,(d, e)$ **– idea (c.d.).** Wierzchołek u jest liściem w drzewie BST



Pytanie. Jaka oczekiwana i pesymistyczna złożoność czasowa algorytmu Delete w tym przypadku?

Słownik – drzewa poszukiwań binarnych BST

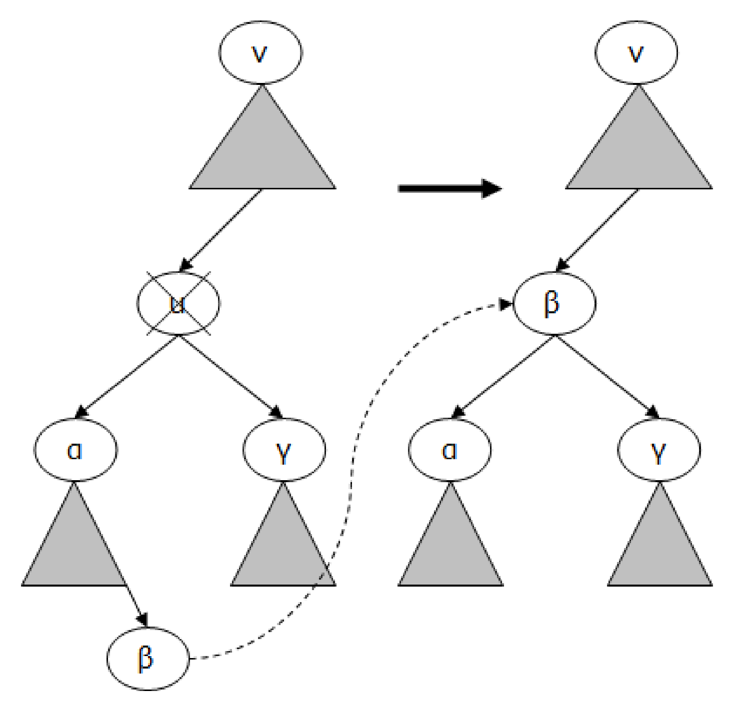
Operacji $delete\,(d,e)$ – **idea (c.d.).** Wierzchołek u ma tylko jednego syna w drzewie BST



Pytanie. Jaka oczekiwana i pesymistyczna złożoność czasowa algorytmu Delete w tym przypadku?

Słownik – drzewa poszukiwań binarnych BST

Operacji $delete\,(d, e)$ **– idea (c.d.).** Wierzchołek u ma dwóch synów w drzewie BST (wierzchołek β jest bezpośrednim poprzednikiem w sensie porządku etykiet wierzchołka u, analogicznie dla bezpośredniego następnika w poddrzewie wierzchołka γ):



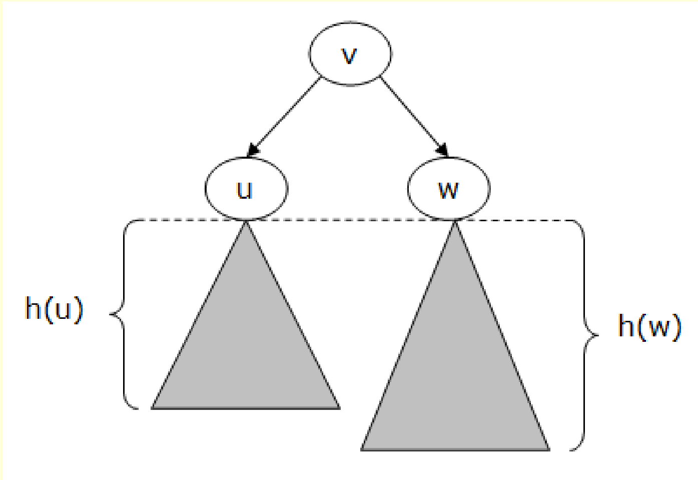
Pytanie. Jaka oczekiwana i pesymistyczna złożoność czasowa algorytmu Delete w tym przypadku?

Słownik

(drzewa poszukiwań binarnych AVL)

Słownik – drzewa poszukiwań binarnych AVL

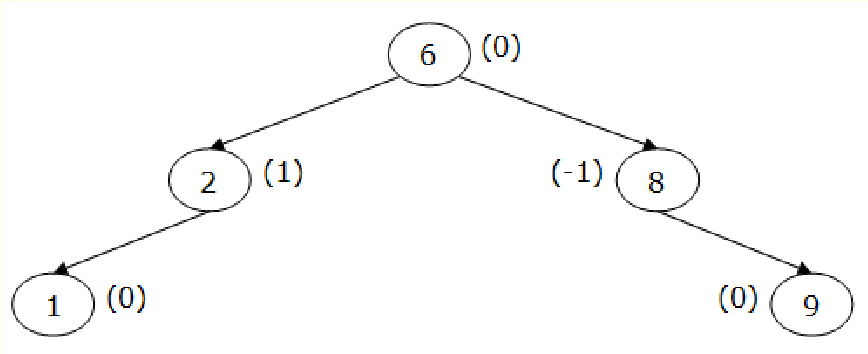
Definicja. *Drzewem binarnych poszukiwań AVL (Adelson, Velskii, Landis)* nazywamy drzewo BST $G = (V_G, E_G, et)$, gdzie dla każdej trójki wierzchołków u, v, w różnica wysokości poddrzew wierzchołka v, tj. $h(u) - h(w)$ jest równa -1, 0, albo 1.



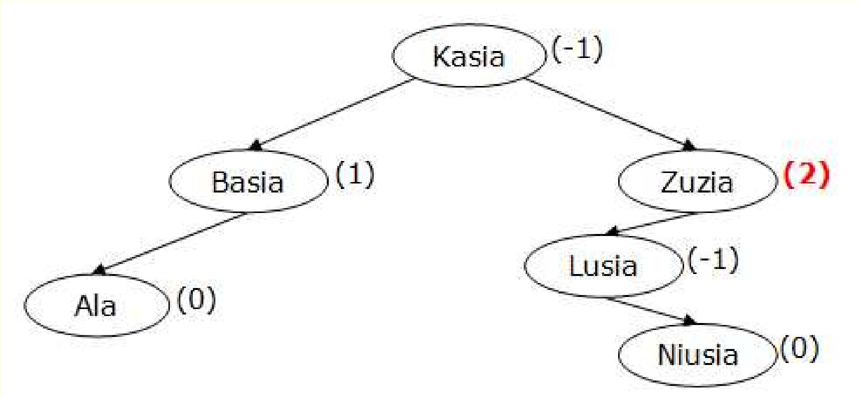
Słownik – drzewa poszukiwań binarnych AVL

Przykłady:

- zbiór etykiet $\langle \mathbb{N}, \leq \rangle$ – drzewo BST i AVL:

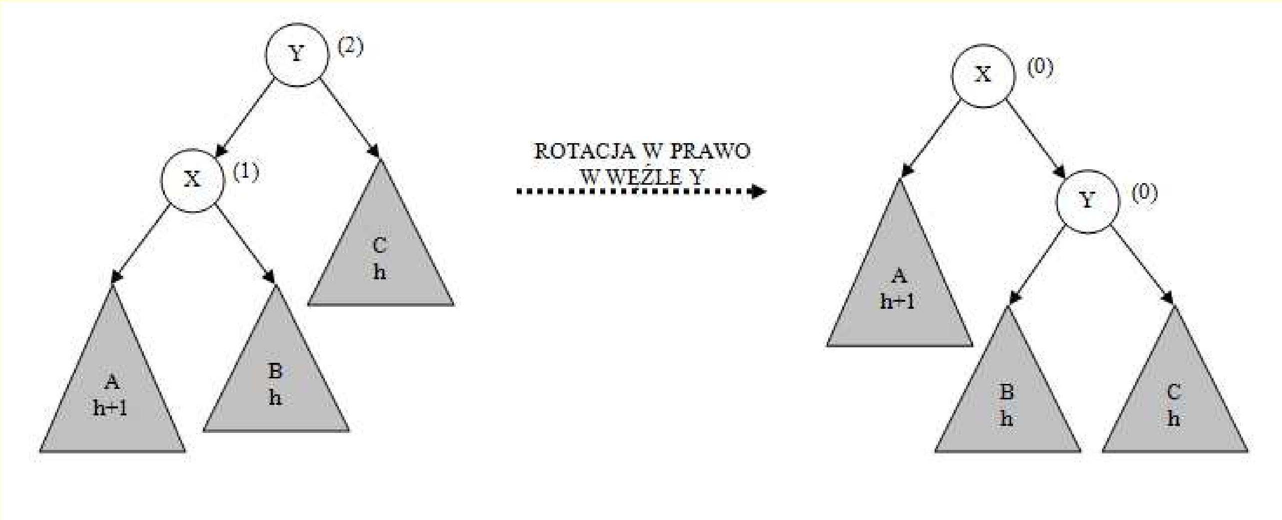


- zbiór etykiet $\langle \Pi, \leq_{leks} \rangle$ – drzewo BST, ale nie AVL:



Słownik – drzewa poszukiwań binarnych AVL

Schemat rotacji pojedynczej w prawo (rotacja w lewo analogicznie).

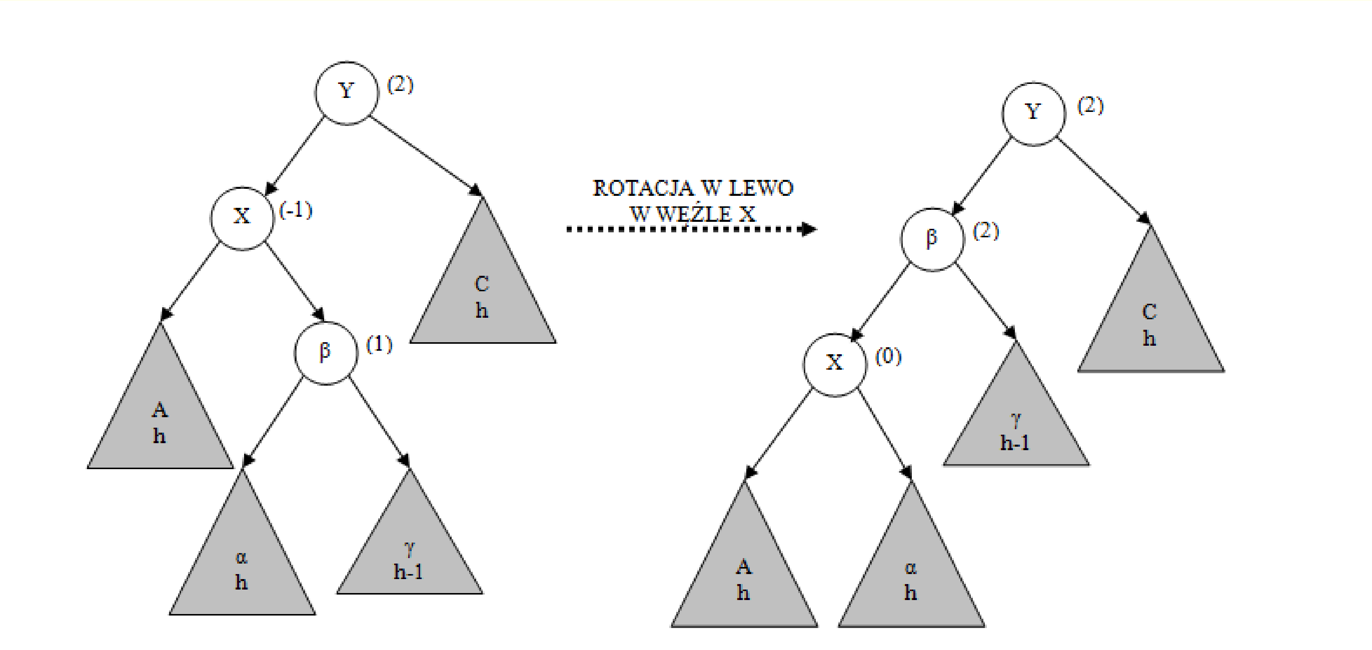


Złożoność. Koszt czasowy pojedynczej rotacji względem operacji na strukturze drzewa AVL jest stały.

Zadanie. Podaj przykład drzewa BST, ale nie AVL, dla którego wykonanie pojedynczej rotacji w prawo spowoduje, że drzewo stanie się drzewem AVL.

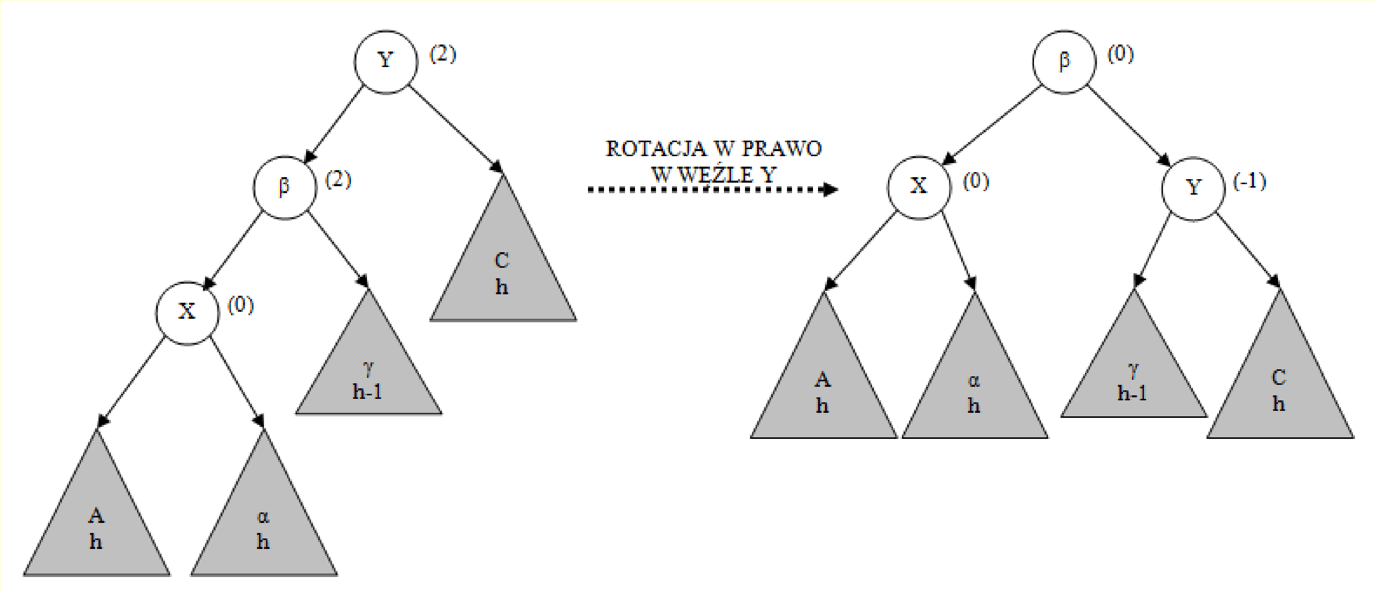
Słownik – drzewa poszukiwań binarnych AVL

Schemat rotacji podwójnej lewo-prawo (rotacja w prawo-lewo analogicznie)



Słownik – drzewa poszukiwań binarnych AVL

Schemat rotacji podwójnej lewo-prawo (rotacja w prawo-lewo analogicznie) c.d.



Złożoność. Koszt czasowy podwójnej rotacji względem operacji na strukturze drzewa AVL jest stały.

Zadanie. Podaj przykład drzewa BST, ale nie AVL, dla którego wykonanie podwójnej rotacji w lewo-prawo spowoduje, że drzewo stanie się drzewem AVL.

Słownik – drzewa poszukiwań binarnych AVL

Operacja $member\,(d, e)$ **– idea:**

- wyszukujemy wierzchołek zgodnie z procedurą $member$ dla drzew BST,
- rezultatem operacji jest drzewo AVL,

Złożoność operacji $W\,(n) = A\,(n) = O\,(\lg n)$, gdzie n jest liczbą wierzchołków drzewa .

Operacja $insert\,(d, e)$ **– idea:**

- wstawiamy wierzchołek zgodnie z procedurą $insert$ dla drzew BST,
- przeglądamy ścieżkę wierzchołek-korzeń drzewa i wykonujemy stosowne rotacje,
- rezultatem jest drzewo AVL.

Złożoność operacji $W\,(n) = A\,(n) = O\,(\lg n)$, gdzie n jest liczbą wierzchołków drzewa.

Fakt. Operacja $insert$ dla drzew AVL wymaga wykonania co najwyżej jednej co najwyżej podwójnej rotacji bez względu na liczbę wierzchołków drzewa.

Słownik – drzewa poszukiwań binarnych AVL

Operacja $delete\,(d, e)$ **– idea:**

- usuwamy wierzchołek u zgodnie z procedurą $delete$ dla drzew BST, jeżeli:
  - wierzchołek u jest liściem w drzewie AVL albo wierzchołek u ma tylko jednego syna w drzewie AVL, to przeglądamy ścieżkę wierzchołek-korzeń drzewa i wykonujemy stosowne rotacje,
  - wierzchołek u ma dwóch synów w drzewie AVL, to przeglądamy ścieżkę wierzchołek bezpośredni poprzednik-korzeń (analogicznie wierzchołek bezpośredni następnik-korzeń) drzewa i wykonujemy stosowne rotacje,
- rezultatem jest drzewo AVL.

Złożoność operacji $W\,(n) = A\,(n) = O\,(\lg n)$, gdzie n jest liczbą wierzchołków drzewa.

Fakt. Operacja $delete$ dla drzew AVL wymaga wykonania co najwyżej $\lg n$ co najwyżej podwójnych rotacji, gdzie n jest liczbą wierzchołków drzewa.

Zadanie (**). Podaj przykład drzewa AVL o $n > 16$ wierzchołkach, w którym usunięcie pewnego wierzchołka wymusi wykonanie $\Theta\,(h)$ rotacji, gdzie h jest wysokością drzewa.

Sortowanie przy użyciu drzew poszukiwań binarnych

Sortowanie przy użyciu drzew poszukiwań binarnych

Idea algorytmu TreeSort. Niech E będzie zbiorem n elementów z określoną relacją $\leq$ porządku częściowego oraz T pustym drzewem poszukiwań binarnych:

- wstaw (w dowolnej kolejności) wszystkie elementy zbioru E do drzewa T,
- wypisz wierzchołki drzewa T w kolejności InOrder.

Otrzymany ciąg elementów zbioru E jest posortowany rosnąco względem relacji porządku częściowego $\leq$.

Zadanie (*). Udowodnij, że etykiety dowolnego drzewa poszukiwań binarnych odczytane w kolejności InOrder tworzą ciąg niemalejący.

Fakt. Złożoność czasową i pamięciową algorytmu TreeSort w strukturze drzew AVL można oszacować kolejno przez $\Theta(n \lg n)$ i $\Theta(n)$, gdzie n jest rozmiarem danych wejściowych.

Wniosek. Algorytm TreeSort jest optymalnym algorytmem dla problemu sortowania przez porównania.

Pytanie. Jaka jest złożoność czasowa algorytmu TreeSort w strukturze drzew BST?