Wykład 6 i 7 – drzewa

Drzewo jest bardziej skomplikowaną strukturą niż poprzednio omawiane.

Dla każdego drzewa wyróżniony jest jeden, charakterystyczny element- ***korzeń***.

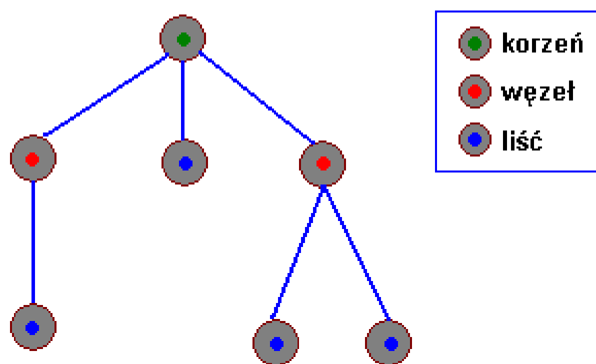
Korzeń jest jedynym elementem drzewa, który nie posiada elementów poprzednich.

Dla każdego innego elementu określony jest dokładnie **jeden** element poprzedni.

Dla każdego elementu oprócz ostatnich, tzw. **liści** istnieje co najmniej 1 element następny.

Jeżeli liczba następnych elementów wynosi dokładnie 2 (oprócz liści) to drzewo nazywamy **binarnym**.

Drzewo można zdefiniować, jako acykliczny **graf**.



Dla każdego drzewa można określić:

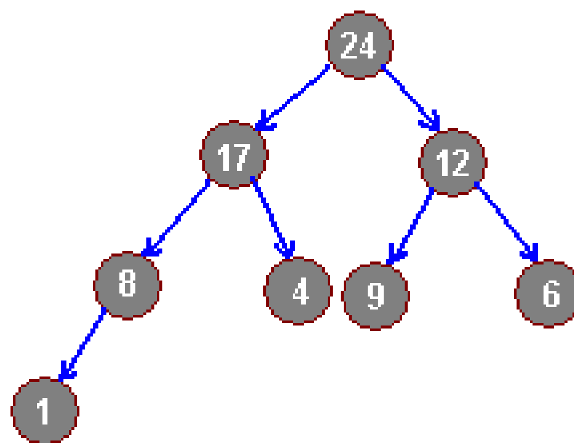
- **długość** drogi u (**głębokość**) - liczba wierzchołków, przez które należy przejść od korzenia do wierzchołka u,
- **wysokość** u - maksymalna liczba wierzchołków na drodze od u do pewnego liścia,
- **wysokość drzewa** = głębokość = wysokość korzenia +1
- **ścieżka** z u do v - sekwencja wierzchołków, przez które należy przejść z wierzchołka u do v ,
- **stopień wierzchołka** - liczba jego bezpośrednich następników
- **stopień drzewa** - maksymalny stopień wierzchołka

Kopiec inaczej zwany stogiem jest szczególnym przypadkiem **drzewa binarnego**, które spełnia tzw. *warunek kopca* tzn. każdy następnik jest niewiększy od swego poprzednika.

Własności kopca:

- w **korzeniu** kopca znajduje się największy element,
- na ścieżkach (połączeniach między węzłami), od korzenia do liścia, elementy są posortowane nierosnąco

Przykładowy kopiec:



Szczególne własności kopców zostały wykorzystane do stworzenia algorytmy do sortowania zwanego **HeapSort**.

Drzewa BST

Uogólnieniem wyszukiwania elementu w tablicy uporządkowanej jest wyszukiwanie elementu w tzw. drzewach poszukiwań binarnych, czyli w skrócie **BST** (od ang. Binary Search Trees).

W **BST** uzyskujemy możliwość szybszego wykonywania operacji wstawiania i usuwania elementu ze zbioru.

Każdy węzeł x drzewa, będący obiektem typu node, ma trzy atrybuty:

x: left(x), key(x), right(x)

W **BST** mamy porządek **symetryczny**,

tzn. dla każdego węzła x jest spełniony następujący warunek:

jeśli węzeł y leży w lewym poddrzewie x, to
key(y) <= key(x);

jeśli węzeł y leży w prawym poddrzewie x, to
key(y) >= key(x)

Drzewem poszukiwań binarnych (drzewem BST) nazywamy dowolne drzewo binarne, w którym elementy zbioru są wpisane do wierzchołków zgodnie z porządkiem symet-rycznym.

Operacja search:

```
function search(v : T; r: node) : node;
{T jest dowolnym typem liniowo uporządkowanym;
  r jest korzeniem drzewa BST}
var x : node;
begin
     x:= r;
     while (x < > nil) and (key(x) < > v) do
     if v < key(x) then x := left(x) else x := right (x);
     search := x
  {jeśli element v znajduje się w drzewie, to x < > nil
                                                  i key(x) = v:
   jeśli elementu v nie ma w drzewie, to x = nil}
end search;
```

Pierwszą czynnością w operacjach insert i delete jest wykonanie operacji search.

Potrzebny jest poprzednik końcowego węzła x. W tym celu modyfikujemy operację search.

```
function search(v : T; r: node, var y: node) : node;
 {T jest dowolnym typem liniowo uporządkowanym;
   r jest korzeniem drzewa BST;
  y jest poprzednikiem wierzchołka wyszukiwanego
    przez search}
var x : node;
begin
    x := r; y:=nil;
    while (x < > nil) and (key(x) < > v) do begin
        y:=x;
        if v < key(x) then x := left(x) else x := right (x);
    end;
    search := x
end search;
```

Operacja insert polega na:

wykonaniu search(v, r, y),
utworzeniu nowego wierzchołka x,
wstawieniu tam elementu v
i dowiązaniu x do y.

```
procedure insert (v : T; var r : node);
var x, y: node;
begin
    if search(v, r, y) = nil then
    begin
        new(x) ;
        left(x) := nil; key(x) := v; right(x) := nil;
        if y= nil then r := x else
          if v< key(y) then left(y) := x else right(y) := x
    end
end insert;
```

Usunięcie elementu ze zbioru wymaga usunięcia wierzchołka z drzewa.

Nie można swobodnie usuwać wierzchołków z drzewa, gdy mają one następniki !

Proponowane rozwiązanie:

Zastąpić wierzchołek x zawierający element v wierzchołkiem zawierającym albo element bezpośrednio poprzedzający v w zbiorze S, albo element bezpośrednio następujący po v.

W celu zagwarantowania losowej postaci drzewa BST po usunięciu wierzchołka, losowany jest wierzchołek, który zostanie wybrany.

Fizycznie usuwamy jest wierzchołek, który ma co najwyżej jeden wewnętrzny następnik (co najmniej jeden z jego następników jest równy nil).

Algorytm delete:

Pesymistyczna złożoność O(n)

Oczekiwana złożoność O(logn) – wynik empiryczny

Dwie wersje algorytmu

procedure **delete**(v ; T; var r : node);
var x, y, z, t : node; b: 0 .. 1;

```
begin
    x := search(v, r , y);
    if x < > nil then
     begin
        if (left(x) =nil) or (right(x) = nil) then
         begin
         if (left(x) =nil) and (right(x) =nil) then z :=nil
           else if left(x) = nil then z := right (x)
                   else z := left(x);
         if y = nil then r := z else
           if x= left(y) then left(y) := z else right(y) := z
        end
        else
        begin {left(x) < > nil i right(x) < > nil}
        b := random(2);
{jeśli b=0, to w miejsce v wstawiamy element bezpośrednio poprzedzający v w zbiorze S. Jeśli b = 1, to w miejsce v wstawiamy element bezpośrednio następujący po v w zbiorze S}
       if b = 0 then
       begin
           z := left(x) ;
           if right(z) = nil then left(x) := left(z) else
           begin
                repeat
                    t := z;
                    z := right(z)
                until right (z) = nil;
                right (t) := left (z)
            end
        end else
        begin
            z := right(x);
            if left(z) = nil then right(x) := right(z) else
            begin
                repeat
                   t := z;
                   z := left(z)
                until left (z) = nil;
       left(t) : = right(z)
     end
   end;
  key(x) := key(z)
end end end delete;
```

Możliwa jest prostsza wersja operacji delete, nazywana **opóźnionym usuwaniem**.

Zamiast usuwać wierzchołek x zawierający element v, przyjmuje się wierzchołek x za "usunięty" i pozostawia się go w drzewie.

Drzewa AVL

Poprawiają złożoność operacji na BST
z O(n) do O(logn).

Aby uzyskać czas działania O(logn), trzeba dodatkowo zadbać, żeby drzewa BST pozostawały w postaci gwarantującej wysokość O(logn), gdzie n jest liczbą wierzchołków.

Istnieje wiele odmian takich drzew.

Najprostsze z nich to drzewa AVL.

Drzewo BST jest drzewem AVL wtedy, kiedy dla każdego wierzchołka wysokości dwóch jego poddrzew różnią się co najwyżej o 1.

LEMAT
Wysokość drzewa AVL o n wierzchołkach
(n >= l) jest nie większa niż 1,45 log n.

Poza atrybutami left, key i right każdy wierzchołek drzewa AVL ma również atrybut bf():

x: bf(x) = h_L(x) – h_R(x)

bf(x) należy do {-1,0,1}

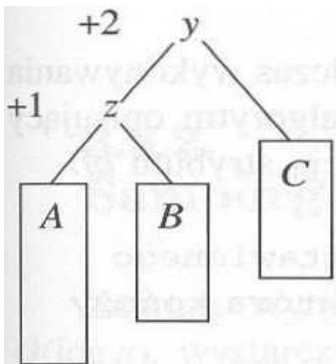
Search – jak dla BST

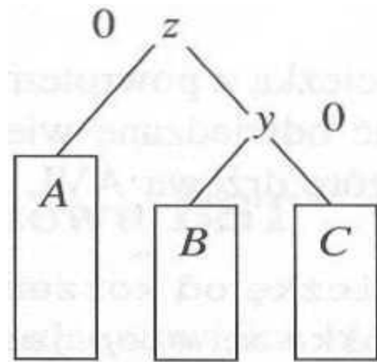
Insert i delete – początkowo takie same, ale potem drzewo trzeba przywrócić do postaci AVL.

Operacja insert(v,S) dla drzewa AVL.
Po wstawieniu nowego węzła x za pomocą zwykłego algorytmu insert dla drzew BST przesuwamy się z powrotem po ścieżce od x w stronę korzenia, dokonując odpowiednich zmian atrybutu bf (wierzchołki tej ścieżki stają się "cięższe") do chwili napotkania takiego wierzchołka y, że albo

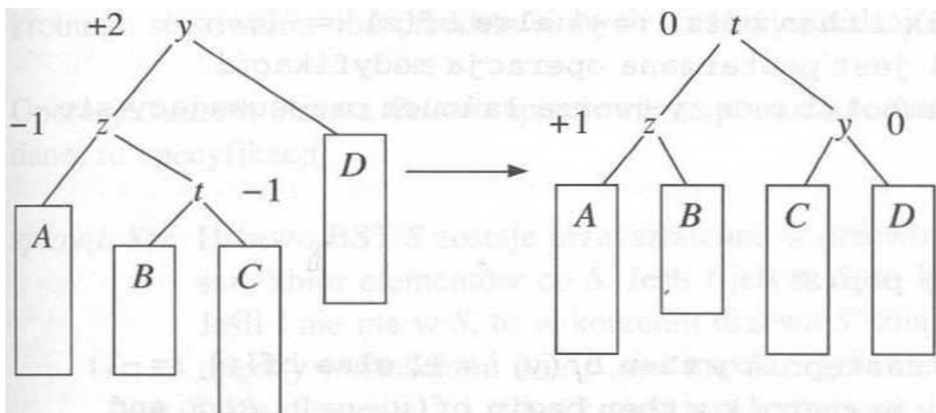
(a) y jest korzeniem, a nowa wartość bf(y) jest różna od 2 i -2, albo
(b) nowa wartość bf(y) = 0, albo
(c) nowa wartość bf(y) = 2 lub -2.

W przypadku c) dokonujemy **rotacji**.





Rotacja pojedyńcza w y



Rotacja podwójna w y

LEMAT
Rotacja pojedyncza i rotacja podwójna prowadzą od drzewa BST do drzewa BST.

Wykonanie rotacji przywraca danemu poddrzewu jego wysokość przed rozpoczęciem wykonywania operacji insert. Dla pozostałych wierzchołków na ścieżce do korzenia atrybut bf pozostaje nie zmieniony.

Aby umożliwić przejście ścieżką z powrotem do korzenia, należy podczas wykonywania operacji search umieszczać odwiedzane wierzchołki na stosie.

Algorytm opisujący przechodzenie ścieżką w górę drzewa AVL z odpowiednią aktualizacją atrybutu bf.

```
{Stos S zawiera ścieżkę od korzenia do poprzednika
  wstawionego do drzewa wierzchołka x;
  stop jest nazwą procedury, która kończy operację insert}

if S = pusty then stop;
t := x ;
{osobno rozpatrujemy poprzednik wstawianego wierzchołka}
z := front(S) ; pop(S);
if bf(z) <> 0 then begin bf(z) := 0; stop end;
if t-lewynastępnik z then bf(z) :=+l else bf(z) := -1;
{w poniższej pętli jest powtarzana operacja modyfikacji atrybutu bf;
wierzchołki t, z, y tworzą łańcuch przesuwający się w górę drzewa}
while S < > 0 do
begin
    y:= front(S) ; pop(S);
    case bf(y) of
          0: if z-lewy następnik y then bf(y) := +1 else bf(y) := -1;
        +1: if z-prawy następnik y then begin bf(y) := 0; stop end
              else if bf(z) = +1
              then begin rotacja pojedyncza(y, z); stop end
              else begin rotacja podwójna (y, z, t); stop end;
         -l: if z-lewy następnik y then begin bf(y) := 0; stop end
             else if bf( z) = -1
             then begin rotacja pojedyncza(y,z); stop end
             else begin rotacja podwójna (y, z, t); stop end;
     end;
  t := z; z := y

end
```

Dla drzew AVL każdą z operacji search, insert i delete można wykonać z pesymistyczną złożonością czasową O(log n).

Zaimplementowanie drzewa AVL wymaga O(n) dodatkowej pamięci na atrybuty left, right i bf, gdzie n jest maksymalną liczbą elementów w zbiorze S.

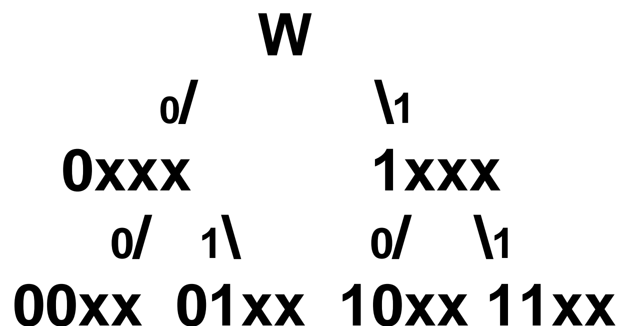
Drzewa RST (Radix Search Trees)

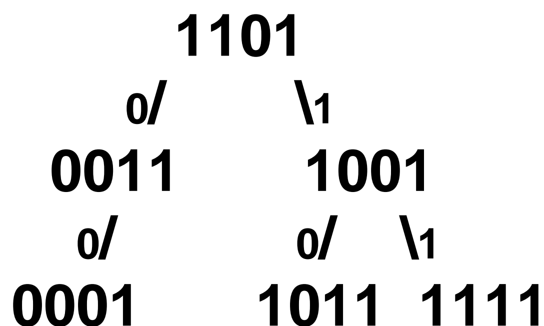
Rozgałęzienia na podstawie wartości kolejnego bitu poszukiwanego słowa.

Zaczynamy od najbardziej znaczącego bitu v_maxb.

v = (v_maxb,….,v_0)

bits(v,k,j) = (v_k+j-1,…,v_k+1,v_k)





Instrukcja Search:

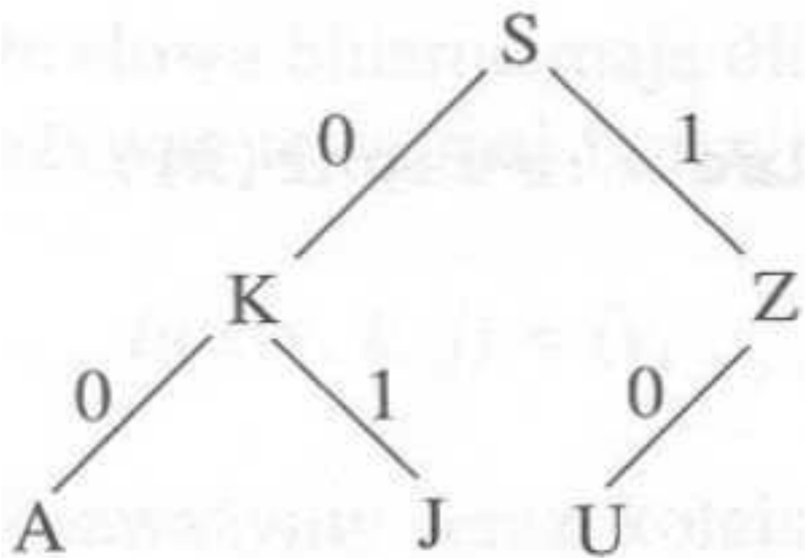
```
function digitalsearch{v : integer; r: node) : node;
  {r jest korzeniem drzewa RST}
var b : integer, x: node;
begin
     b := maxb;
     x := r;
     while (x < > nil) and (key(x) < > v) do
     begin
          if bits(v, b, 1) = 0 then x := left(x)
                                   else x := right(x);
          b:= b-1
     end;
     digitalsearch := x
{(v  jest w S i key(x) = v) lub (v nie jest w S i x=nil)}
end digitalsearch;
```

Oczekiwana złożoność czasowa operacji search i insert – O(logn). Złożoność pamięciowa: O(n)

```
function digitalinsert (v : integer; var r : node) : node;
{r jest korzeniem drzewa RST; v jest elementem
  do wstawienia w drzewie R5T}
var x,y : node;
      b: integer;
begin
{pierwsza część algorytmu to wyszukanie miejsca w
  drzewie do wstawienia elementu v}
     b := maxb;
     x := r;
     y := nil;
     while (x < > nil) and (key(x) <> v) do
     begin
          y:=x;
          if bits(v, b, l) = 0 then x := left(x) else x := right(x);
              b := b-1;
      end;
      if x =  nil then
     {jeśli v nie ma w drzewie, to zostaje wstawiony do nowego
      węzła, który jest doczepiany do y}
     begin
          new(x);
          key(x) := v; left(x) := nil; right(x) := nil;
          if y <> nil then
          if bits(v, b+l, 1) = 0 then left(y) := x else right(y) :=x
          else r := x;
     end;
     digitalinsert := x;
end digitalinsert;
```

Operacja delete – w miejsce usuwanego elementu można wstawić element z dowolnego liścia w poddrzewie.

Przykład



S – 10011
Z – 11010
U – 10101
K – 01011
A - 00000
J - 01010

Drzewa TRIE

W drzewie TRIE elementy zbioru S są zapisywane w liściach.

Węzły wewnętrzne mają tylko dwa atrybuty: left i right.

Liść ma tylko atrybut key.

Oprócz tego jest potrzebny atrybut logiczny typu Boolean:

leaf(x) = true wtw x jest liściem

```
function triesearch(v: integer; r: node): node;
 {r jest korzeniem niepustego drzewa TRIE}
var b : integer; x : node;
begin
   b := maxb, x := r;
   while not leaf(x) do
       begin
           if bits(v,b,1) = 0 then x := left(x) else x:= right(x);
            b:=b - 1
       end;
       triesearch := x
{(v jest w S i key(x) =v) lub (v nie jest S i key(x) <> v)}
end triesearch;
```

Dla wszystkich operacji (założenie: maxb + 1 >= log n)

T(n) = O(min(maxb,n) + maxb)
A(n) = O(logn)
S(n) = O(n(maxb – log n + 2))

Drzewa PATRICIA

PATICIA jest modyfikacją drzewa TRIE bez jego wad, czyli długich gałęzi i niejednorodności węzłów.

W drzewie PATRICIA jest tylko n węzłów do reprezentowania n elementów.

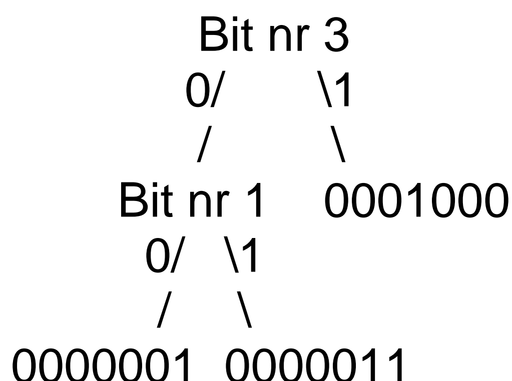
Ma ono zaletę drzewa TRIE:

porównanie kluczy odbywa się tylko raz (na koniec wyszukiwania).

Aby skasować długie gałęzie, na ścieżce od korzenia pozostawia się tylko porównania bitów, na których różnią się reprezentowane słowa.

Aby odróżnić na przykład trzy słowa bitowe:
0001000, 0000011, 0000001, wystarczy najpierw porównać bity nr 3;

w wypadku bitu 0 na tej pozycji kolejne bity porównujemy na pozycji nr 1.



Aby uniknąć niejednorodności węzłów, utożsamiamy każdy liść z pewnym wierzchołkiem wewnętrznym, wpisując zapisany w liściu element zbioru S do odpowiadającego mu wierzchołka wewnętrznego.

W ten sposób każdy węzeł wewnętrzny jest traktowany dwojako:

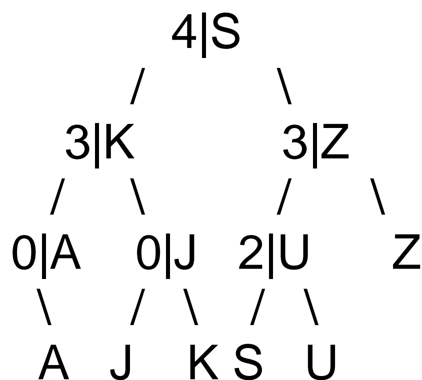
w fazie wyszukiwania jako węzeł określający rozgałęzienie,

w fazie identyfikacji (dojścia do liścia) - jako liść.

Trzeba przy tym dodać jeden węzeł, gdyż jest tylko n – 1 węzłów określających rozgałęzienie.

Przykład: Drzewo PATRICIA dla:

S – 10011
Z – 11010
U – 10101
K – 01011
A - 00000
J - 01010



Każdy węzeł drzewa PATRICIA ma 4 atrybuty: dowiązania drzewowe lefi i right,
atrybut key i
atrybut b (numer bitu, na podstawie którego następuje rozgałęzienie).

Zakładamy, że element zidentyfikowany (przez wyszukiwanie) w danym węźle jest wpisany do jednego z przodków danego węzła.

Dowiązanie od węzła x do y identyfikuje element wtw b(x) <= b(y).

```
function patriciasearch(v : integer; r: node) : node;
{r jest korzeniem drzewa PATRICIA}
var x, y: node;
begin
   x := r;
   repeat
       y:= x;
       if bits(v, b(x), 1) = 0  then x := left(x)
                                else x :=right(x)
   until b(y) <= b(x) ;
   patriciasearch := x
   {v jest w S wttw key(x) = v}
end patriciasearch;
```

```
function patriciainsert(v: integer; r: node) : node,
{r jest korzeniem drzewa PATRICIA}
var t, x, y, z : nade; i : .integer,
begin
    t :=patriciasearch(v, r);
    {element v daje te same wyniki porównań co key(t)}
    i :=maxb;
    while bits(v, i, 1) = bits(key(t), i, 1) do i := i - 1;
    {i jest numerem pierwszej pozycji,
      na której v i key(t) się różnią}
    x:= r;
    repeat
        z := x ;
        if bits(v, b(x), 1) = 0 then x := left(x)
                                  else x := right(x)
        until (b(z) <= b(x)) or (b(x) < i);
    {miejsce elementu v jest na dowiązaniu między z a x}
       new(y); key(y) := v; b(y) := i;
       if bits(v, b(y), 1) = 0 then
       begin
           left(y)   := y;
           right(y) := x
       end else
       begin
           right(y) := y;
           left (y)  := x
       end;
       if bits(v, b(z), 1)=0 then left(z):=y else right(z):= y;
       patriciainsert := y
end patriciainsert;
```

Dla operacji search i insert (założenie: maxb >= log n)

T(n) = O(min(maxb,n) + maxb)
A(n) = O(logn)
S(n) = O(n)