Algorytmy i struktury danych I
Drzewa
IO, WIMiIP
2021/2022

Danuta Szeliga

AGH Kraków

Spis treści I

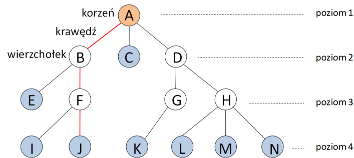
1 Drzewa
- Drzewa binarne

2 Drzewa poszukiwań binarnych - BST

Drzewo

Drzewo (tree)

Drzewo jest hierarchiczną strukturą danych.
Def. Drzewo jest to zbiór T jednego lub więcej elementów zwanych węzłami, takich że istnieje jeden wyróżniony węzeł zwany korzeniem drzewa i pozostałe węzły (z wyłączeniem korzenia) są podzielone na $m \geq 0$ rozłącznych zbiorów $T_1, \ldots, T_m$, z których każdy jest drzewem, zwanym poddrzewem korzenia.



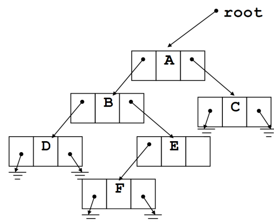
- Pierwszy obiekt zwany jest korzeniem, kolejne obiekty traktowane są jako jego potomstwo: węzły. Liście to węzły nie mające potomstwa
- Droga w drzewie – sekwencja węzłów w drzewie odpowiadających przejściu w kierunku od korzenia do liścia
- Pojęcia: rodzic, przodek, potomek, rodzeństwo (dwa węzły są rodzeństwem, gdy mają tego samego ojca)
- Warianty: drzewa AVL, drzewa czerwono-czarne, BST, ...

Drzewa binarne

- Drzewo binarne jest skończonym zbiorem węzłów, który jest albo pusty, albo zawiera korzeń oraz dwa drzewa binarne

- Każdy węzeł przechowuje dwa wskaźniki: do lewego poddrzewa `left` i prawego poddrzewa `right`
- `root` jest wskaźnikiem do drzewa
- Jeśli `root` $= \Lambda$ - drzewo puste

wpp `root` jest adresem korzenia drzewa, `left(root)` wskazuje lewe poddrzewo, `right(root)` wskazuje prawe poddrzewo



Implementacja wskaźnikowa

```
struct NODE{
 T val; // wartość
 NODE* left;  // wskaźnik do lewego syna
 NODE* right; // wskaźnik do prawego syna
} *root = null; // początkowo drzewo jest puste
```

Drzewa binarne - operacje

Podstawowe operacje dla drzew

- wyliczenie wszystkich elementów drzewa ("przejście"drzewa)
- wyszukanie elementu
- dodanie nowego elementu/poddrzewa w określonym miejscu drzewa
- usunięcie elementu/poddrzewa

Przechodzenie drzewa binarnego

- Jest to systematyczne przeglądanie węzłów w taki sposób, ze każdy węzeł jest odwiedzony dokładnie jeden raz
- Przejście drzewa wyznacza porządek liniowy w drzewie

6 sposób przechodzenia drzewa

VLR, LVR, LRV, VRL, RVL, RLV

gdzie: **V**isit = odwiedź węzeł, **L**eft = idź w lewo, **R**ight = idź w prawo

W szczególności wyróżnia się trzy pierwsze:

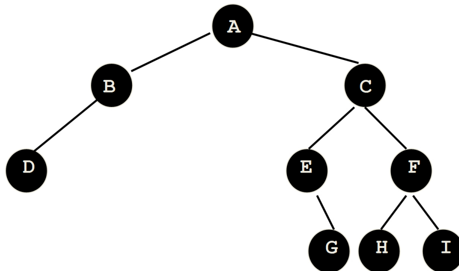
- VLR pre-order, wzdłużny: korzeń, lewe poddrzewo, prawe poddrzewo
- LVR in-order, poprzeczny: lewe poddrzewo, korzeń, prawe poddrzewo
- LRV post-order, wsteczny: lewe poddrzewo, prawe poddrzewo, korzeń

```
preorder(NODE* root){
 if(≠ root) return;
 visit(root->val)
 if(root->left)
  preorder(root->left);
 if(root->right);
  preorder(root->right);
}
```

```
inorder(NODE* root){
 if(≠ root) return;
 if(root->left)
  inorder(root->left);
 visit(root->val);
 if(root->right)
  inorder(root->right);
}
```

```
postorder(NODE* root){
 if(≠ root) return;
 if(root->left)
  postorder(root->left);
 if(root->right)
  postorder(root->right);
 visit(root->val);
}
```

Przykład



- Pre-order: A B D C E G F H I
- In-order: D B A E G C H F I
- Post-order: D B G E H I F C A

Porządek in-order - algorytm nierekurencyjny

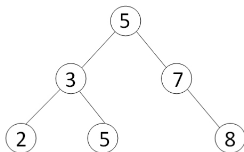
```
inorder(NODE* root){
S = Λ; //S - stos
p = root; // p - zmienna pomocnicza
while(1){
  while(p ≠ Λ){
     push(S,p);
     p = p->left;
  }
  if (S=Λ) return; //koniec algorytmu
  p = pop(S);
  visit(p);
  p = p->right;
}
```

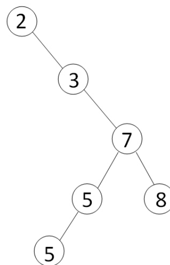
Drzewa poszukiwań binarnych - BST

Binarne drzewo poszukiwań (Binary search tree)

Binarne drzewo poszukiwań to drzewo binarne o następującej własności każdy element binarnego drzewa poszukiwań ma tę własność, że jego lewostronne potomstwo jest mniejsze bądź równe co do wartości od tego elementu, a prawostronne potomstwo jest większe bądź równe (drzewo BST)

Własność binarnego drzewa poszukiwań
dla każdego węzła `x` drzewa zachodzi:
`x->val` $\geq$ `x->left->val` oraz `x->val` $\leq$ `x->right->val`





Implementacja BST

- Implementacja wskaźnikowa

```
struct BST_N{
   T val; // wartość
   BST_N* left;   // wskaźnik do lewego syna
   BST_N* right;  // wskaźnik do prawego syna
   BST_N* parent; // opcjonalny wskaźnik do ojca
} * root = null; // początkowo drzewo jest puste
```

- Implementacja tablicowa

```
struct BST_N{
   T val; // wartość
   integer left;   // wskaźnik do lewego syna
   integer right;  // wskaźnik do prawego syna
   integer parent; // opcjonalny wskaźnik do ojca
} tree[N];
root = 0; // początkowo drzewo jest puste
```

Operacje na BST

- Przechodzenie drzewa
- Wyszukiwanie węzła
  - o podanym kluczu
  - największego/najmniejszego
  - następnika/poprzednika węzła
- Wstawianie węzła do drzewa
- Usuwanie węzła/poddrzewa

Przechodzenie BST I

- Własność BST umożliwia **wypisanie** wszystkich znajdujących się w nim elementów w uporządkowany sposób
- Wykorzystujemy algorytm przechodzenia drzewa metodą inorder (przechodzenie poprzeczne)

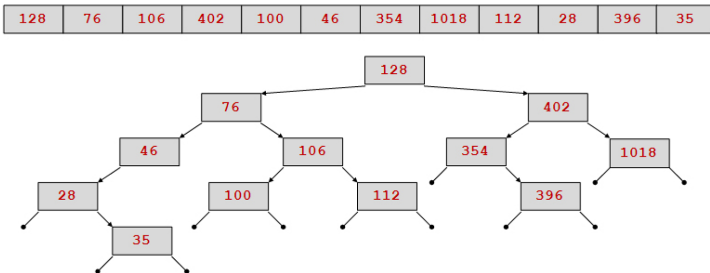
```
inorder(BST_N* root){
if(¬ root)
   stop;
if(root->left)
   inorder(root->left);

wypisz(root->val);

if(root->right)
  inorder(root->right);
}
```

- Złożoność: $\mathcal{O}(n)$, n — liczba węzłów drzewa

Przechodzenie BST II



- Niekiedy procedura przechodzenia drzewa BST nazywana jest śortowaniem"drzewiastym
- Algorytm śortowania"drzewiastego

```
TreeSort (val arr[]){
   BST_N * root ← arr; //przekształć listę wejściową w BST
   inorder(root);
```

- Procedura ta **nie zmienia** porządku w tablicy, a jedynie **wypisuje** elementy w sposób uporządkowany

Wyszukiwanie węzła w BST

- Wersja rekurencyjna

```
BST_N* find(NODE* root, T x) {
   if(¬ root) return 0;
   if(root->val = x) return root;
   if(root->val > x)
      return find(root->left,x);
   else
      return find(root->right,x);
}
```

- Wersja iteracyjna

```
NODE* find(BST_N* root, T x) {
   while(root ∧ root->val ≠ x)
      if(root->val > x) root = root->left;
      else root = root->right;
   return root;
}
```

- Złożoność obliczeniowa: $\mathcal{O}(h)$, gdzie h jest wysokością drzewa

Wyszukiwanie minimum i maksimum w BST

- Wyszukiwanie minimum: należy przejść od korzenia do najbardziej lewego węzła

```
BST_N* min(BST_N* root){
   while(root->left)
      root = root->left;
   return root;
}
```

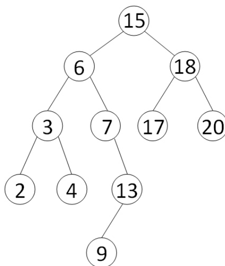
- Wyszukiwanie maksimum: należy przejść od korzenia do najbardziej prawego węzła

```
BST_N* max(BST_N* root){
   while(root->right)
      root = root->rigth;
   return root;
}
```

- Złożoność obliczeniowa: $\mathcal{O}(h)$, gdzie h jest wysokością drzewa

Następnik i poprzednik w BST

- Struktura BST umożliwia wyznaczenie następnika i poprzednika **bez konieczności porównywania kluczy**
- Konieczne jest wtedy przechowywanie w każdym węźle wskaźnika do ojca `NODE* parent`
- Jeśli wszystkie klucze są różne, to
    - następnikiem węzła `x` jest węzeł o najmniejszym kluczu większym od `x->val`
    - poprzednikiem węzła `x` jest węzeł o największym kluczu mniejszym od `x->val`



Następnik w BST

- Następnik:
  - jeżeli jest prawe poddrzewo, to następnikiem węzła `x` jest najmniejszy element tego poddrzewa
  - jeżeli brak prawego poddrzewa, to następnikiem węzła `x` jest jego najniższy przodek, którego lewy syn jest przodkiem `x`
- Funkcja `next`

```
BST_N* next(BST_N* x){
   if(x->right) // jeżeli jest prawe poddrzewo
      return min(x->right); // najmniejszy na prawo
   BST_N* y = x->parent; // brak prawego poddrzewa
   while(y ∧ x = y->right){ // cofamy się do góry
      x = y;
      y = y->parent;
   }
   return y;
}
```

- Złożoność obliczeniowa: $\mathcal{O}(h)$, gdzie h jest wysokością drzewa

Poprzednik w BST

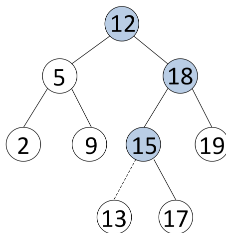
- Poprzednik:
  - jeżeli jest lewe poddrzewo, to następnikiem węzła x jest największy element tego poddrzewa
  - jeżeli brak lewego poddrzewa, to następnikiem węzła x jest jego najniższy przodek, którego prawy syn jest przodkiem x
- Funkcja prev

```
BST_N* prev(BST_N* x){
   if(x->left) // jeżeli jest lewe poddrzewo
      return max(x->left); // największy na prawo
   BST_N* y = x->parent; // brak prawego poddrzewa
   while(y ∧ x = y->left){ // cofamy się do góry
      x = y;
      y = y->parent;
   }
   return y;
}
```

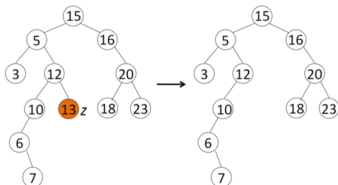
- Złożoność obliczeniowa: $\mathcal{O}(h)$, gdzie h jest wysokością drzewa

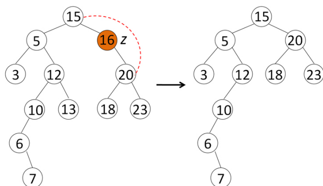
Wstawianie węzła w BST

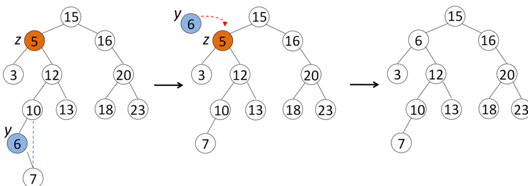
```
insert-BST(BST_N *&root, BST_N* x){
   if(¬ root) { // jeżeli drzewo było puste
   root = x; stop; // x->left = x->right = NULL;
   }
   BST_N* par = 0;
   BST_N* son = root;
   while(son){
      par = son;
      if(par->val > x->val) son = par->left;
      else son = par->right;
   }
   x->parent = par; // x->left = x->right = 0;
   if(par->val > x->val) par->left = x; // x - lewym synem
   else par->right = x; // x - prawym synem
}
```



Usuwanie węzła w BST (3 przypadki)







Usuwanie węzła w BST

```
remove-BST(BST_N* &root, BST_N* z){
   BST_N* y;
   if(z->left ∧ z->right) y = next(z); //do usunięcia
   else y = z;
   BST_N* x;
   if(y->left) x = y->left; // sprawdzenie, czy y ma lewego syna
   else x = y->right;
   if(x) x->parent = y->parent; // podpinamy x do ojca y-ka
   if(y->parent) { // ojciec y-ka pokaże na x
      if(y = y->parent->left)
         y->parent->left = x;
      else
         y->parent->right = x;
   }
   else root = x; // jeżeli z = y jest korzeniem
   if(y ≠ z) // nadpisanie usuniętego z
      z->val = y->val;
   delete y; // fizyczne usunięcie węzła
}
```

Usuwany element musi być zastąpiony przez

- swój następnik (w prawym poddrzewie)

lub

- swój poprzednik (w lewym poddrzewie)

Inne rodzaje drzew

- Drzewa BST z powtarzającymi się kluczami
- Drzewa pozycyjne:
  - porządek leksykograficzny
  - klucz każdego węzła można jednoznacznie wyznaczyć na podstawie ścieżki od korzenia do tego węzła $\Rightarrow$ nie ma potrzeby przechowywania klucza w węźle
- Drzewa AVL
- Drzewa czerwono-czarne