

Arbori binari de căutare



Arbori Binari de Căutare

Un **arbore binar de căutare** este un arbore **binar** care satisface următoarea proprietate:

Pentru un nod x :

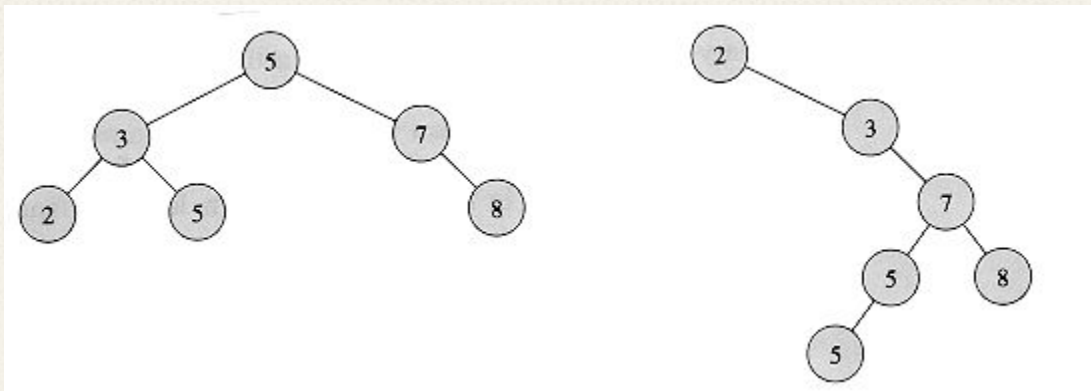
- Dacă y este un nod din subarborele stâng al lui x , atunci $\text{cheie}[y] \leq \text{cheie}[x]$
- Dacă y este un nod din subarborele drept al lui x , atunci $\text{cheie}[x] \leq \text{cheie}[y]$

Arbori Binari de Căutare

Un **arboare binar de căutare** este un arbore **binar** care satisface următoarea proprietate:

Pentru un nod x :

- Dacă y este un nod din subarborele stâng al lui x , atunci $\text{cheie}[y] \leq \text{cheie}[x]$
- Dacă y este un nod din subarborele drept al lui x , atunci $\text{cheie}[x] \leq \text{cheie}[y]$



Arbori Binari

Un *arbore binar strict* este un arbore binar în care fiecare nod fie nu are nici un fiu, fie are exact doi fii.

Nodurile cu doi copii se vor numi *noduri interne*, iar cele fără copii se vor numi *noduri externe* sau *frunze*.

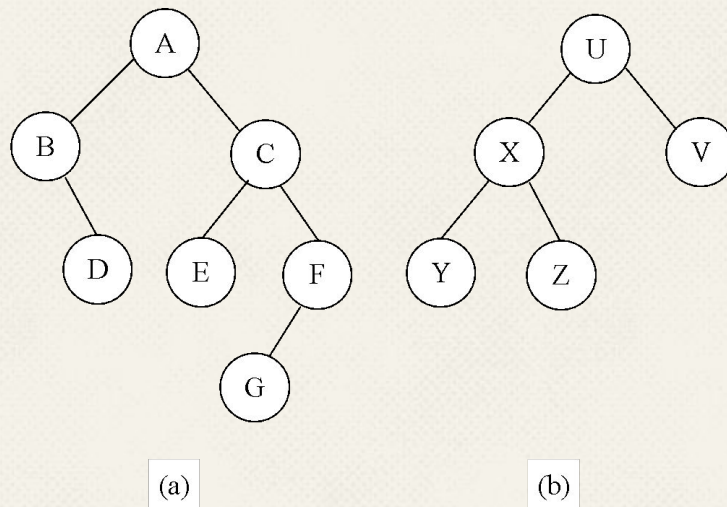
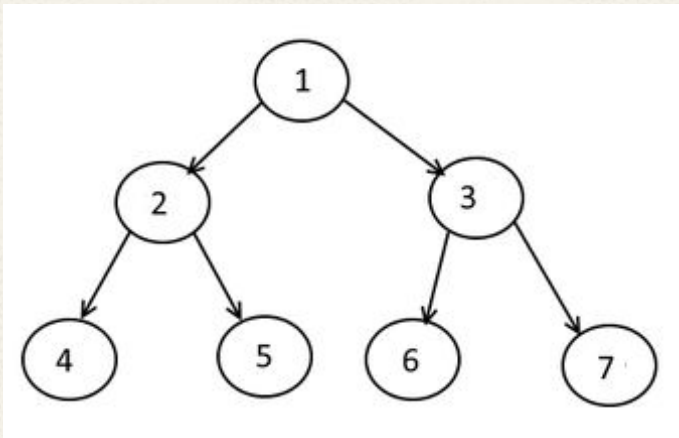


Fig.4.1.1. (a) Un arbore binar nestrict. (b) Arbore binar strict.

Arbori Binari - Parcurgeri

Parcurgeri în arbori binari:

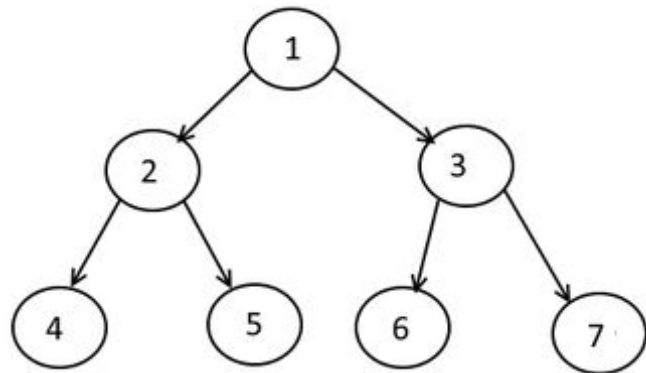
- **Inordine** (SRD, stânga rădăcină dreapta)
- **Preordine** (RSD, rădăcină stânga dreapta)
- **Postordine** (SDR, stânga dreapta rădăcină)



Arbori Binari - Parcurgeri

Parcurgeri în arbori binari:

- **Inordine** (SRD, stânga rădăcină dreapta)
- **Preordine** (RSD, rădăcină stânga dreapta)
- **Postordine** (SDR, stânga dreapta rădăcină)



Inorder Traversal: 4 2 5 1 6 3 7

Preorder Traversal: 1 2 4 5 3 6 7

Postorder Traversal: 7 6 3 5 4 2 1

4 5 2 6 7 3 1

Arbori Binari - Parcurgeri

```
void par_rsd (BTREE t) {  
    if (t != NULL) {  
        visit(t);  
        par_rsd(t->left);  
        par_rsd(t->right);  
    }  
}
```

```
void par_srd (BTREE t) {  
    if (t != NULL) {  
        par_srd(t->left);  
        visit(t);  
        par_srd(t->right);  
    }  
}
```

```
void par_sdr (BTREE t) {  
    if (t != NULL) {  
        par_sdr(t->left);  
        par_sdr(t->right);  
        visit(t);  
    }  
}
```

[Link pt vizualizare](#)

TEMĂ: Se dau SRD și RSD. Afișați arborele

Arbori Binari de Căutare

- Înălțimea arborelui ?

- Minim ?

- Maxim ?

Arbori Binari de Căutare

- Înălțimea arborelui ?

- Minim

- Arbore Binar Complet → Înălțime $\log n$

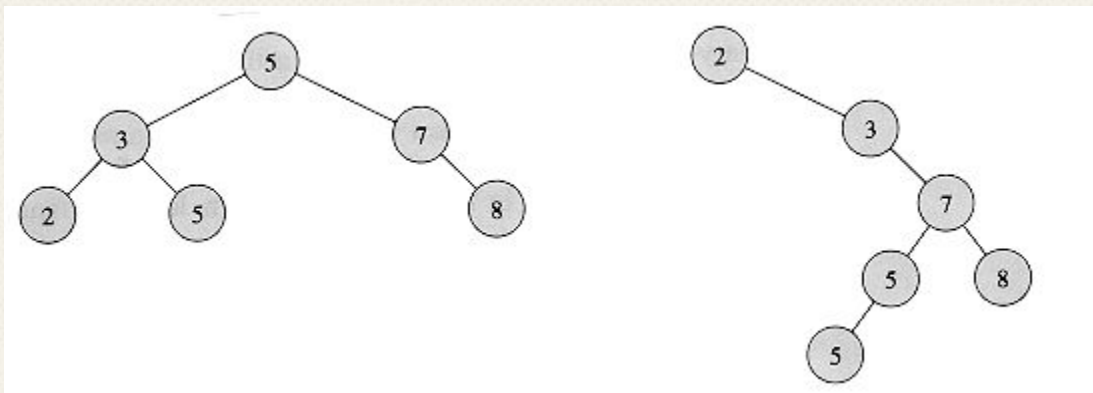
- Maxim

- Dacă avem lanț (elementele sunt inserate în ordine crescătoare sau descrescătoare) → Înălțime n

Arbori Binari de Căutare

○ Ce parcurgere ne oferă vectorul sortat?

- ☐ Preordine
- ☐ Inordine
- ☐ Postordine



Arbori Binari de Căutare

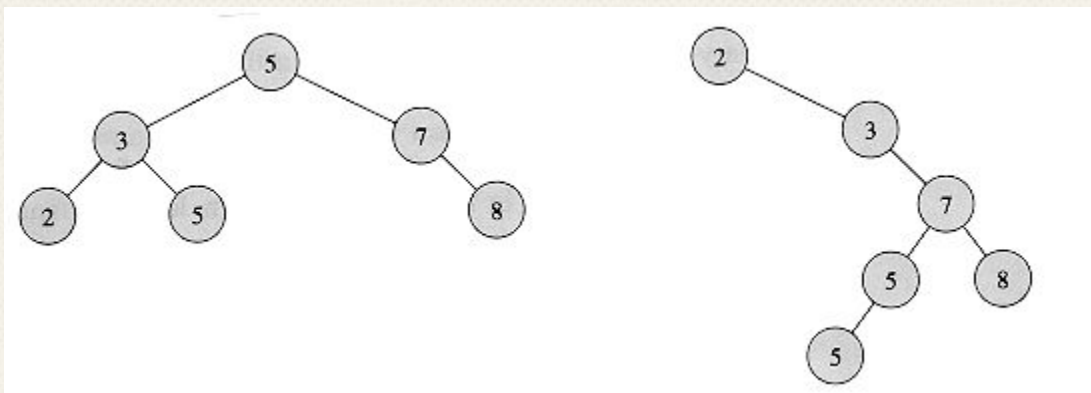
- Parcurgerea **inordine** ne oferă vectorul sortat

- Preordine 5 3 2 5 7 8 | 2 3 7 5 5 8

- Inordine 2 3 5 5 7 8 | 2 3 5 5 7 8

- Postordine 2 5 3 8 7 5 | 5 5 8 7 3 2

- Restul parcurgerilor sunt diferite pentru cei 2 arbori.

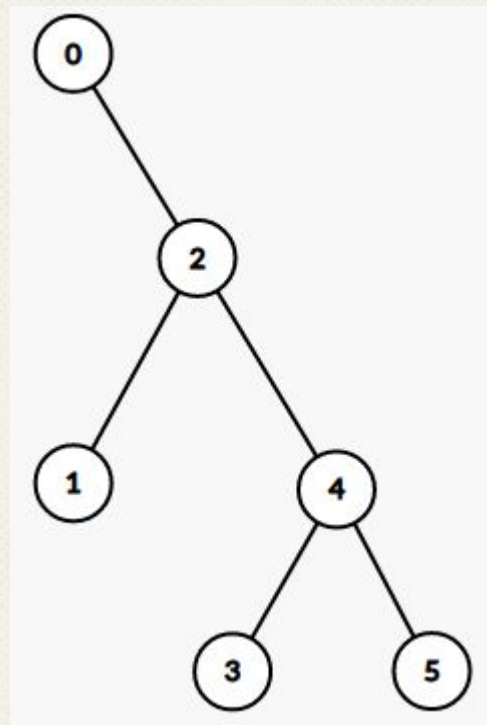
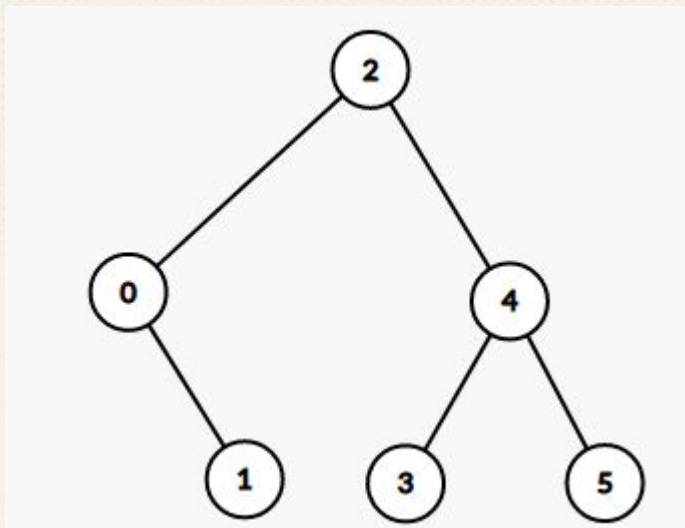


Exercițiu

Desenați arbori binari de înălțime 2, 3, 4, 5 pentru valorile {1, 2, 3, 4, 5}.

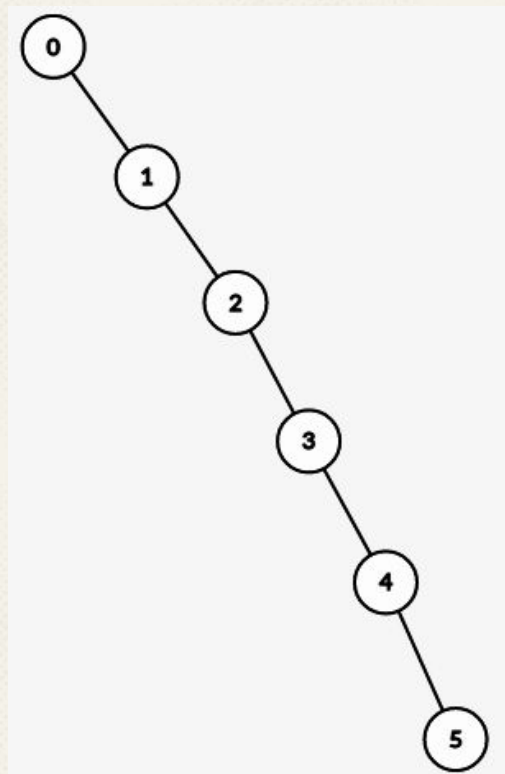
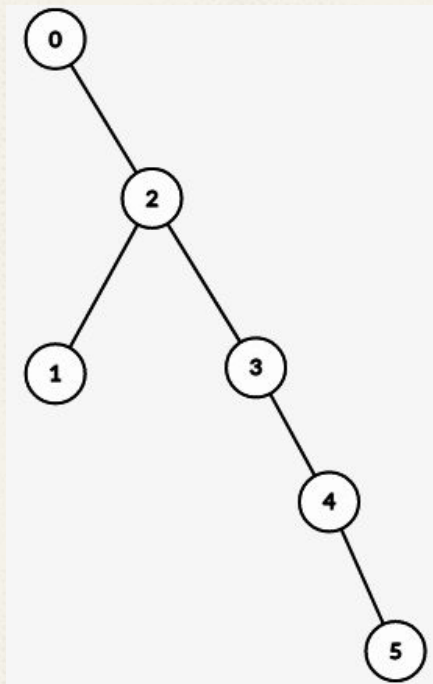
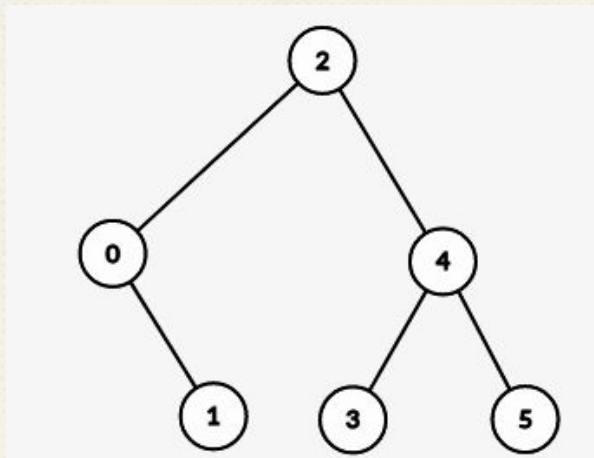
Exercițiu

Desenați arbori binari de înălțime 2, 3, 4, 5 pentru valorile $\{0, 1, 2, 3, 4, 5\}$.



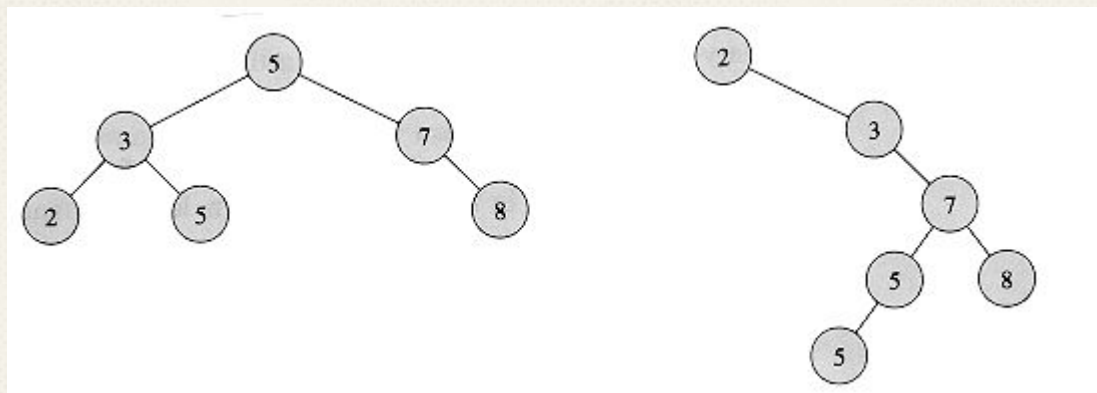
Exercițiu

Desenați arbori binari de înălțime 2, 3, 4, 5 pentru valorile $\{0, 1, 2, 3, 4, 5\}$.



Minim și Maxim

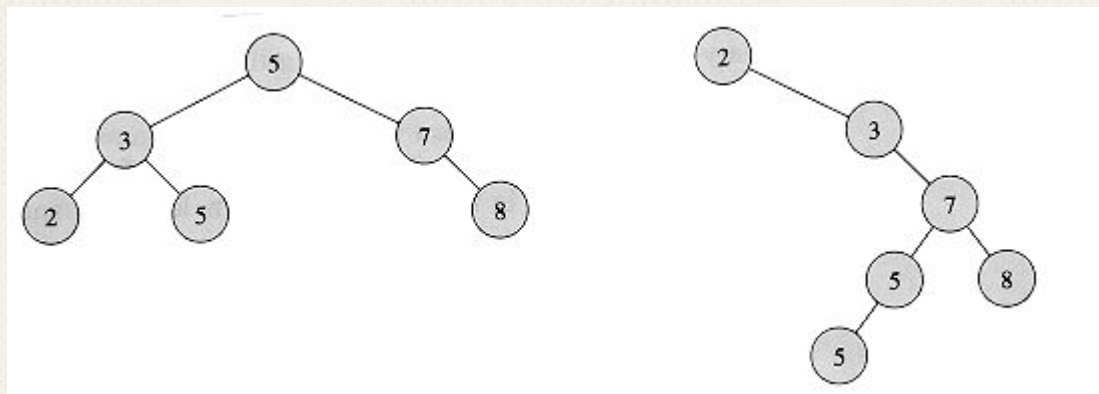
- Unde se află minimul?



Minim și Maxim

○ Unde se află minimul?

□ În cel mai din stânga nod

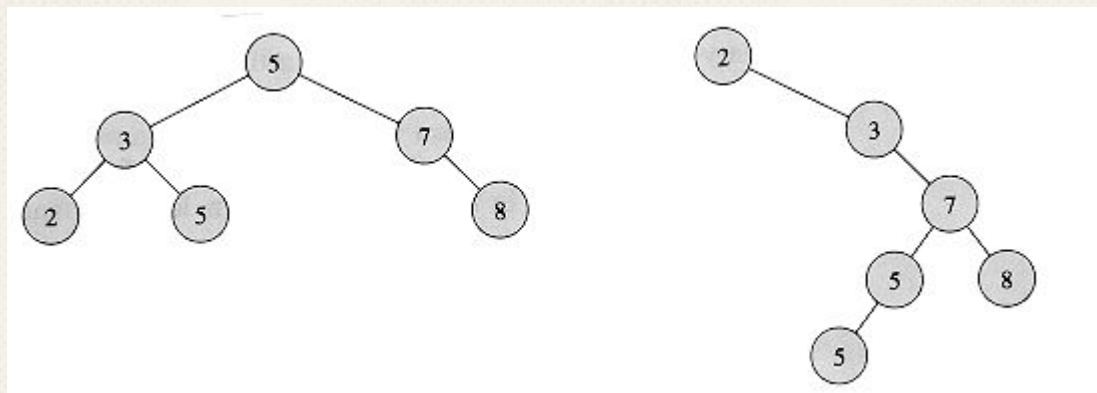


TREE-MINIMUM (x)

```
1 while left[x]  $\neq$  NIL
2   do  $x \leftarrow$  left[x]
3 return x
```

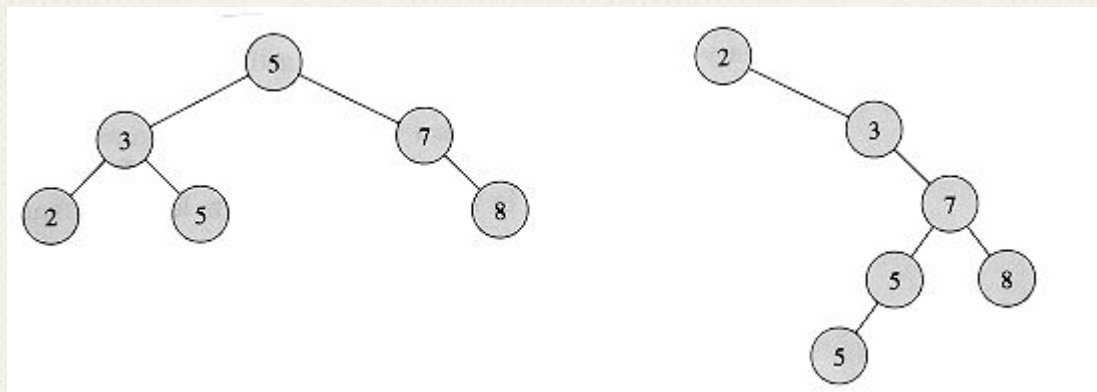
Minim și Maxim

- Unde se află maximul?



Minim și Maxim

- Unde se află maximul?
- În cel mai din dreapta nod

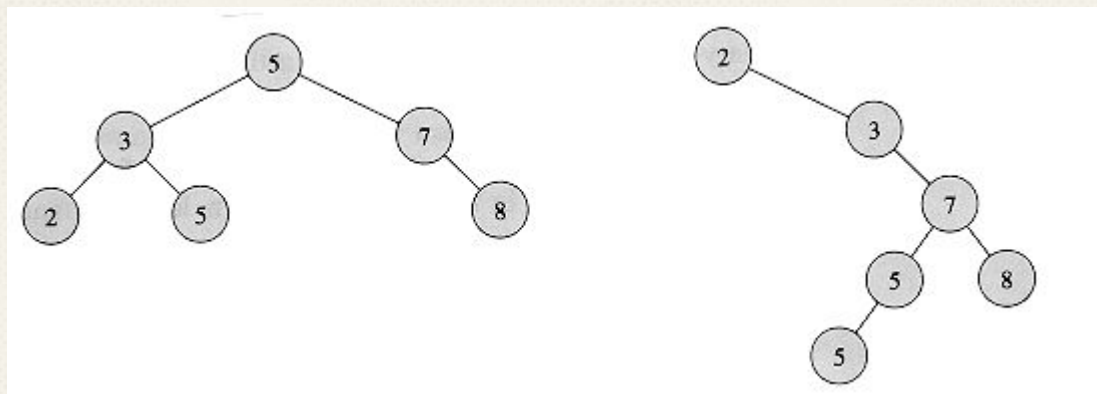


```
TREE-MAXIMUM (x)  
1  while right[x] ≠ NIL  
2      do x ← right[x]  
3  return x
```

Complexitate?

Minim și Maxim

- Unde se află maximul?
- În cel mai din dreapta nod



```
TREE-MAXIMUM (x)  
1  while right[x] ≠ NIL  
2      do x ← right[x]  
3  return x
```

Complexitate? $O(h)$

Căutare

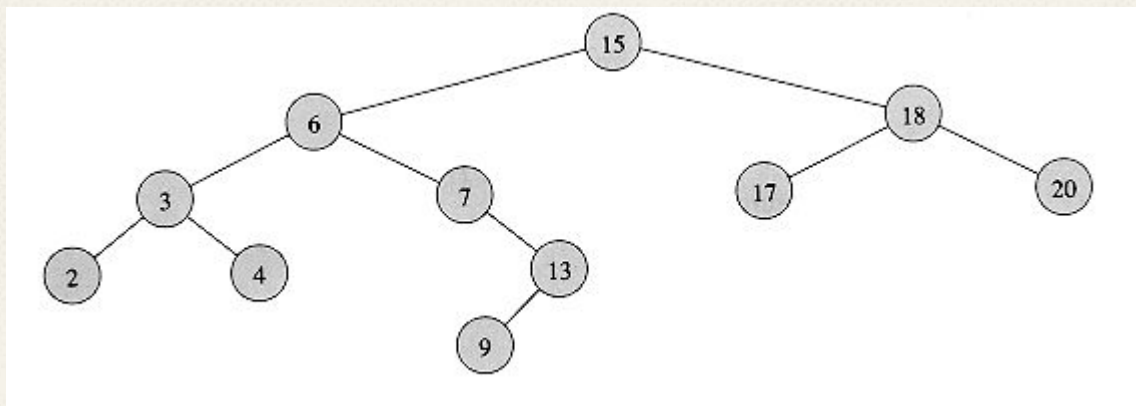
Minimul și maximul se găsesc mai greu decât într-un heap. Avantajul major al arborilor binari de căutare este că permit o căutare “relativ” eficientă.

Cum găsim un element?

Căutare

Minimul și maximul se găsesc mai greu decât într-un heap. Avantajul major al arborilor binari de căutare este că permit o căutare “relativ” eficientă.

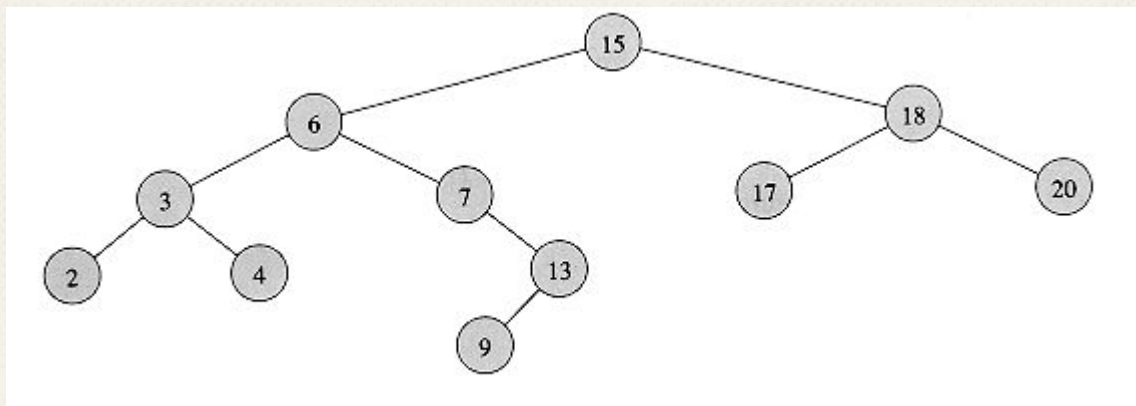
Cum găsim un element?



Căutare

Începem din rădăcină și dacă valoarea din nodul curent este mai mică decât cea ce căutăm, mergem în stânga, dacă valoarea e mai mare, mergem în dreapta.

Evident, ne oprim dacă am găsit valoarea.



Căutare

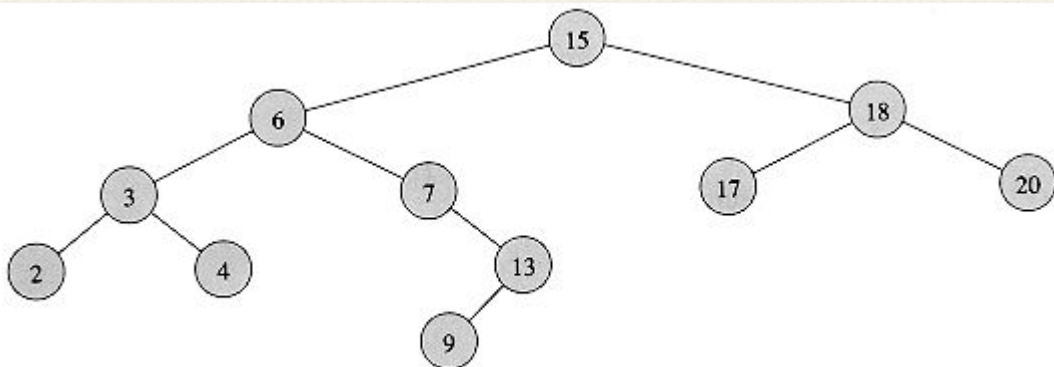
ITERATIVE-TREE-SEARCH (x, k)

```
1 while  $x \neq \text{NIL}$  and  $k \neq \text{key}[x]$ 
2   do if  $k < \text{key}[x]$ 
3       then  $x \leftarrow \text{left}[x]$ 
4       else  $x \leftarrow \text{right}[x]$ 
5 return  $x$ 
```

Complexitate: $O(h)$

TREE-SEARCH (x, k)

```
1 if  $x = \text{NIL}$  or  $k = \text{key}[x]$ 
2   then return  $x$ 
3 if  $k < \text{key}[x]$ 
4   then return TREE-SEARCH ( $\text{left}[x], k$ )
5   else return TREE-SEARCH ( $\text{right}[x], k$ )
```



Succesor / Predecesor

Până acum, puteam să ținem un dicționar și un heap și să facem aceleași operații.

Succesor: Se dă un nod din arbore.

Care este cea mai **mică** valoare din arbore $\geq \text{val}[x]$ (valoarea nodului)?

Predecesor: Se dă un nod din arbore.

Care este cea mai **mare** valoare din arbore $\leq \text{val}[x]$ (valoarea nodului)?

Cum facem?

Sucesor / Predecesor

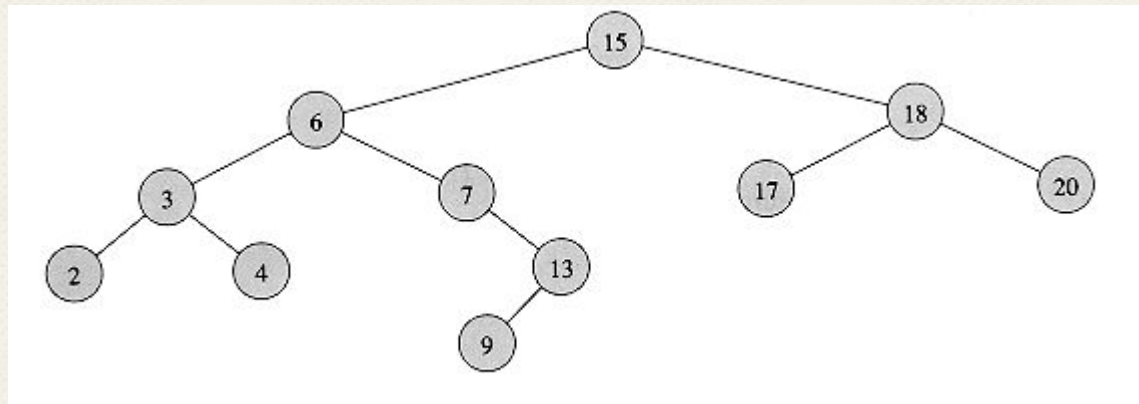
Sucesor de 3?

Sucesor de 6?

Sucesor de 15?

Sucesor de 13?

Sucesor de 4?



Sucesor / Predecesor

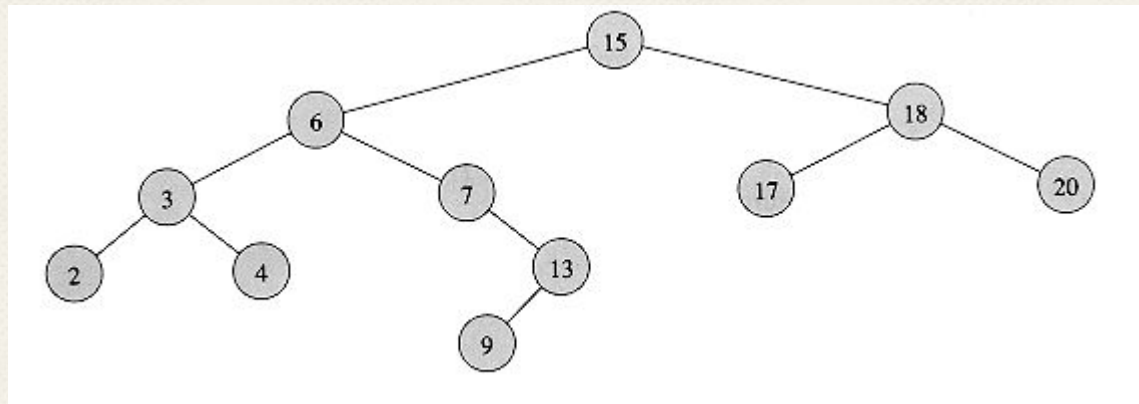
Sucesor de 3? → 4

Sucesor de 6? → 7

Sucesor de 15? → 17

Sucesor de 13? → 15

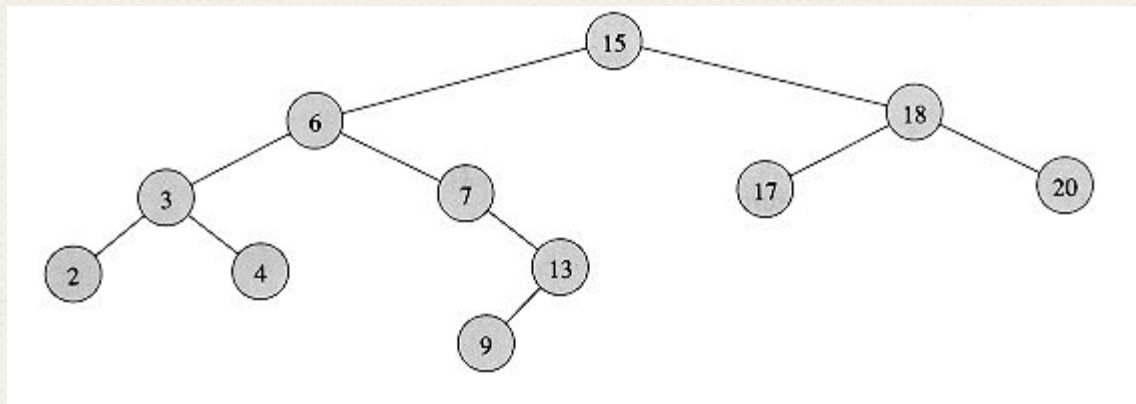
Sucesor de 4? → 6



Succesor / Predecesor

Caz 1) Dacă am fiu drept, atunci cel mai mic element va fi cel mai mic element din subarborele drept. Adică dreapta \rightarrow stânga \rightarrow stânga $\rightarrow \dots \rightarrow$ stânga (vezi 7 sau 15)

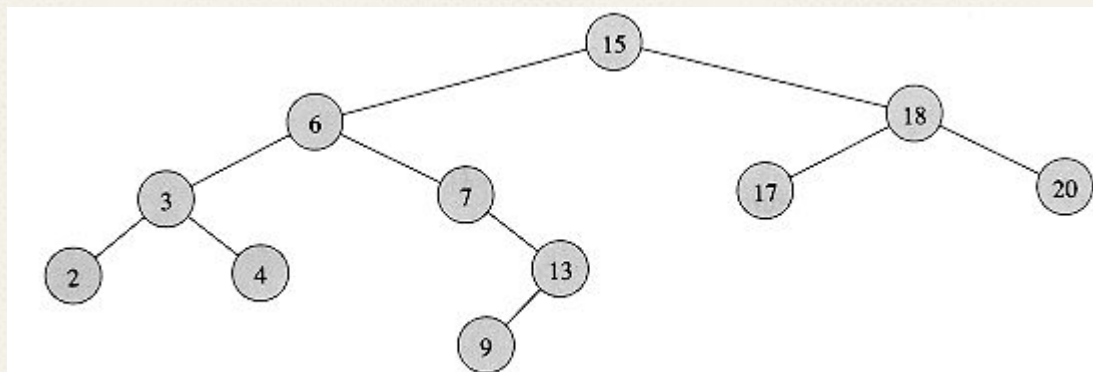
Caz 2) Dacă **nu** am fiu drept, atunci va fi primul strămoș al meu în care eu sunt în subarborele stâng al său (vezi 13, 4, 17)



Successor / Predecessor

TREE SUCCESSOR(*x*)

```
1  if right[x] ≠ NIL
2      then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5      do x ← y
6      y ← p[y]
7  return y
```

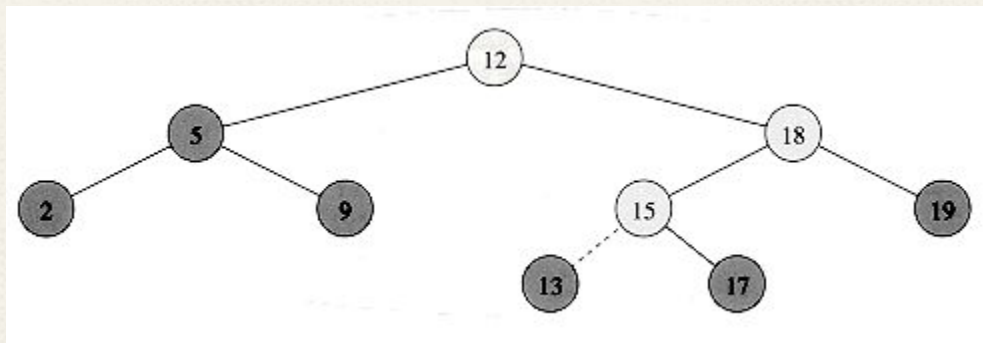


Complexitate: **O(h)**

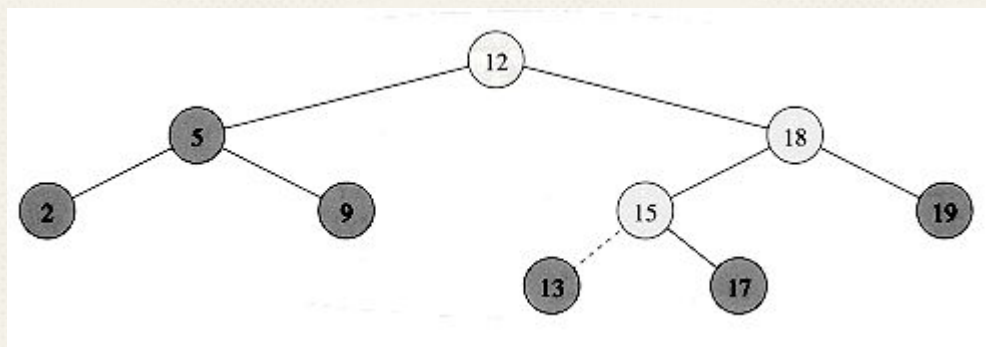
Inserare

Similar cu căutarea.

Inserare 13:



Inserare



Complexitate: $O(h)$

TREE-INSERT(T, z)

1 $y \leftarrow \text{NIL}$

2 $x \leftarrow \text{root}[T]$

3 **while** $x \neq \text{NIL}$

4 **do** $y \leftarrow x$

5 **if** $\text{key}[z] < \text{key}[x]$

6 **then** $x \leftarrow \text{left}[x]$

7 **else** $x \leftarrow \text{right}[x]$

8 $p[z] \leftarrow y$

9 **if** $y = \text{NIL}$

10 **then** $\text{root}[T] \leftarrow z$

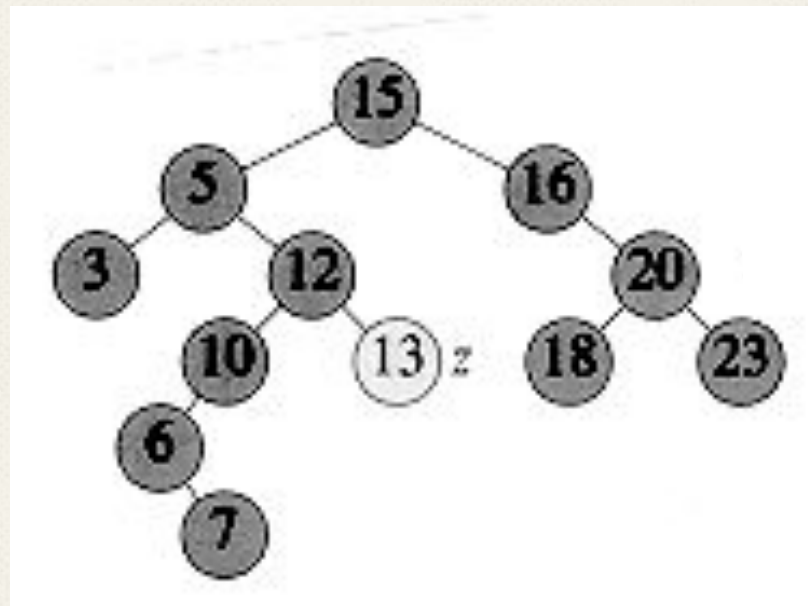
11 **else if** $\text{key}[z] < \text{key}[y]$

12 **then** $\text{left}[y] \leftarrow z$

13 **else** $\text{right}[y] \leftarrow z$

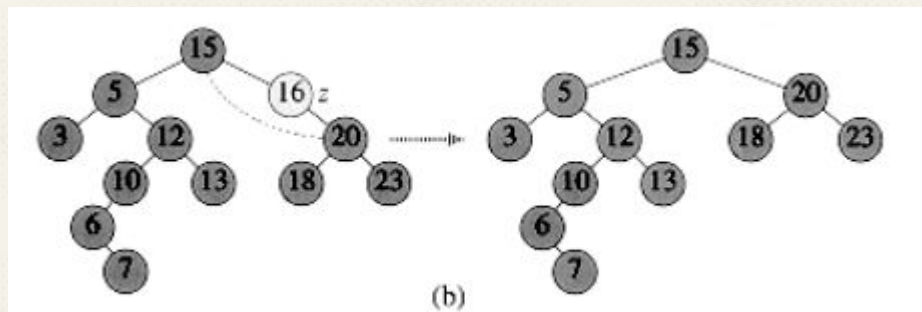
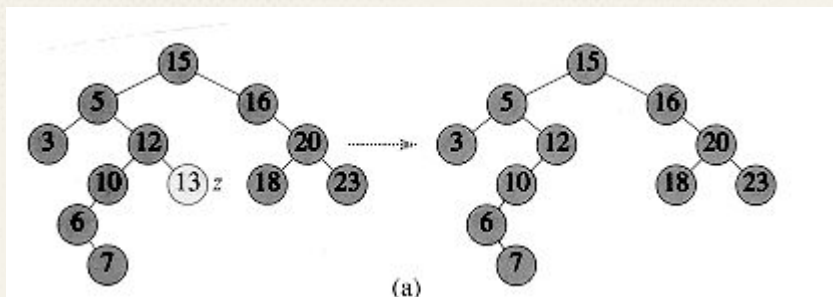
Ștergere

- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7? Pe 16?
- Pe 5?
- Dar pe 15?



Ștergere

- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7? Pe 16?
- Pe 5?
- Dar pe 15?



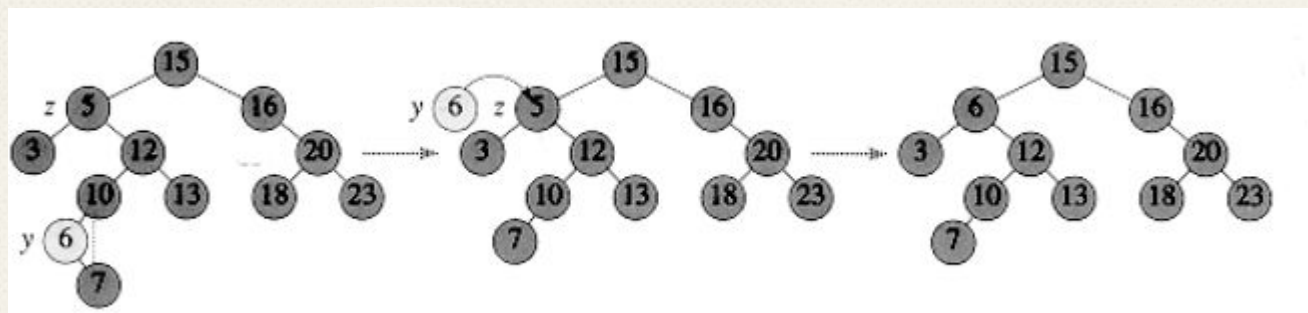
Ștergere

Exercițiu:

- Demonstrați că succesorul unui nod cu 2 fiu are maxim un fiu.

Ștergere

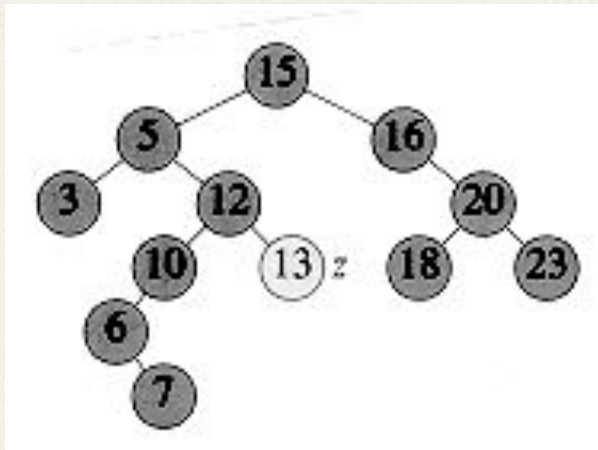
- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7?
- Pe 5?
- Dar pe 15?



Ștergere

Avem 3 cazuri:

- 1) Dacă nodul **nu are** fii, îl ștergem.
- 2) Dacă are **un** fiu, îl ștergem și creăm o tată și noul fiu.



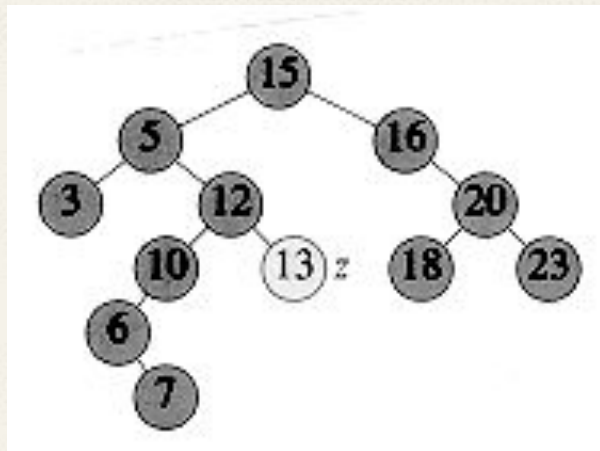
TREE-DELETE(T, z)

```
1  if left[z] = NIL or right[z] = NIL
2      then  $y \leftarrow z$ 
3      else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
4  if left[y]  $\neq$  NIL
5      then  $x \leftarrow \text{left}[y]$ 
6      else  $x \leftarrow \text{right}[y]$ 
7  if  $x \neq \text{NIL}$ 
8      then  $p[x] \leftarrow p[y]$ 
9  if  $p[y] = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow x$ 
11     else if  $y = \text{left}[p[y]]$ 
12         then  $\text{left}[p[y]] \leftarrow x$ 
13         else  $\text{right}[p[y]] \leftarrow x$ 
14  if  $y \neq z$ 
15     then  $\text{key}[z] \leftarrow \text{key}[y]$ 
16         ▷ If  $y$  has other fields, copy them, too.
17  return  $y$ 
```

Ștergere

Avem 3 cazuri:

- 3) Dacă are **ambii** fii, găsim succesorul său, punem în locul său și înlocuim legătura acestui nod cu singurul fiu (dacă există)



TREE-DELETE(T, z)

1 if $left[z] = NIL$ or $right[z] = NIL$

2 then $y \leftarrow z$

3 else $y \leftarrow \text{TREE-SUCCESSOR}(z)$

4 if $left[y] \neq NIL$

5 then $x \leftarrow left[y]$

6 else $x \leftarrow right[y]$

7 if $x \neq NIL$

8 then $p[x] \leftarrow p[y]$

9 if $p[y] = NIL$

10 then $root[T] \leftarrow x$

11 else if $y = left[p[y]]$

12 then $left[p[y]] \leftarrow x$

13 else $right[p[y]] \leftarrow x$

14 if $y \neq z$

15 then $key[z] \leftarrow key[y]$

16 ▷ If y has other fields, copy them, too.

17 return y

Complexitate

Operație	Complexitate
Căutare	$O(?)$
Găsire Minim	$O(?)$
Inserare	$O(?)$
Succesor / Predecesor	$O(?)$
Ștergere	$O(?)$

Complexitate

Operație	Complexitate
Căutare	$O(h)$
Găsire Minim	$O(h)$
Inserare	$O(h)$
Succesor / Predecesor	$O(h)$
Ștergere	$O(h)$

Arbori Binari de Căutare cu Chei Egale

Ce facem dacă avem mai multe chei egale ?

Arbori Binari de Căutare cu Chei Egale

Ce facem dacă avem mai multe chei egale ?

- În caz de egalitate, alegem tot timpul stânga sau dreapta și inserăm în aceeași direcție
- Ținem o listă cu toate elementele egale într-un singur nod (sau un contor care să numere aparițiile, dacă nu avem alte informații)

Arbori Binari Echilibrați

- AVL
 - Arbori Roșu-Negri
 - Treap-uri
 - Splay Trees
 - B-arbori
-
- Skip Lists (nu sunt arbori binari de căutare, dar...)

Search trees

(dynamic sets/associative arrays)

2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^x · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · **Treap** · UB · Weight-balanced

Bibliografie

Introducere în Algoritmi Cormen Leiserson Rivest