ARBORI

Arbori

Definiţie. Se numeşte **graf** $G = (X, V)$ o pereche formată din două mulţimi, mulţimea X a nodurilor sau vârfurilor grafului şi mulţimea V a muchiilor grafului, unde o muchie $v \in V$ este o pereche ordonată de noduri $v=(x,y)$, $x,y \in X$.

Un graf neorientat este un graf în care perechea (x,y) se identifică cu perechea (y,x).
Un graf fără cicluri este un graf în care, pornind de la un vârf dat, nu putem ajunge din nou la el folosind muchii.

Definiţie. Se numeşte **arbore** un graf $H = (X, V)$ care este neorientat, conex, fără cicluri, cu un nod precizat numit rădăcină. Pentru orice vârf $x \in X$, există un număr finit de vârfuri $x_1,...,x_n \in X$ asociate lui x, numite descendenţi direcţi (sau fiii) lui x.

Arbori - definiție recursivă

O structură de arbore (***k***-arbore), T, de un anume tip de bază, este

- fie o structură vidă (adică $T = \emptyset$);
- fie este nevidă, deci conţine un nod de tipul de bază, pe care-l vom numi rădăcină şi îl vom nota *root(T)*, plus un număr finit de structuri disjuncte de arbori de acelaşi tip, T_1, T_2, …, T_k, numiţi subarborii lui T (sau fiii lui *root(T)*).

Reprezentarea ca graf a unui ***k***-arbore:

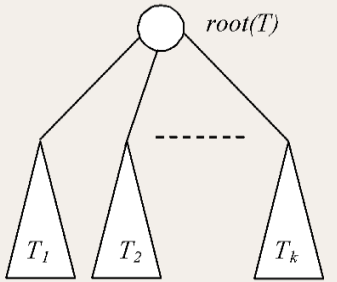


Fig.3.1.1. Un k-arbore T, cu nodul rădăcină *root(T)* şi fii săi, $T_1, T_2, \ldots, T_k$.

Arbori - reprezentări

- Reprezentarea unui arbore **cu legătură *tată - fiu:*** din fiecare nod, avem acces la oricare dintre fii (pentru arbori ordonaţi, pentru arbori la care accesul la noduri se face "de sus în jos”)

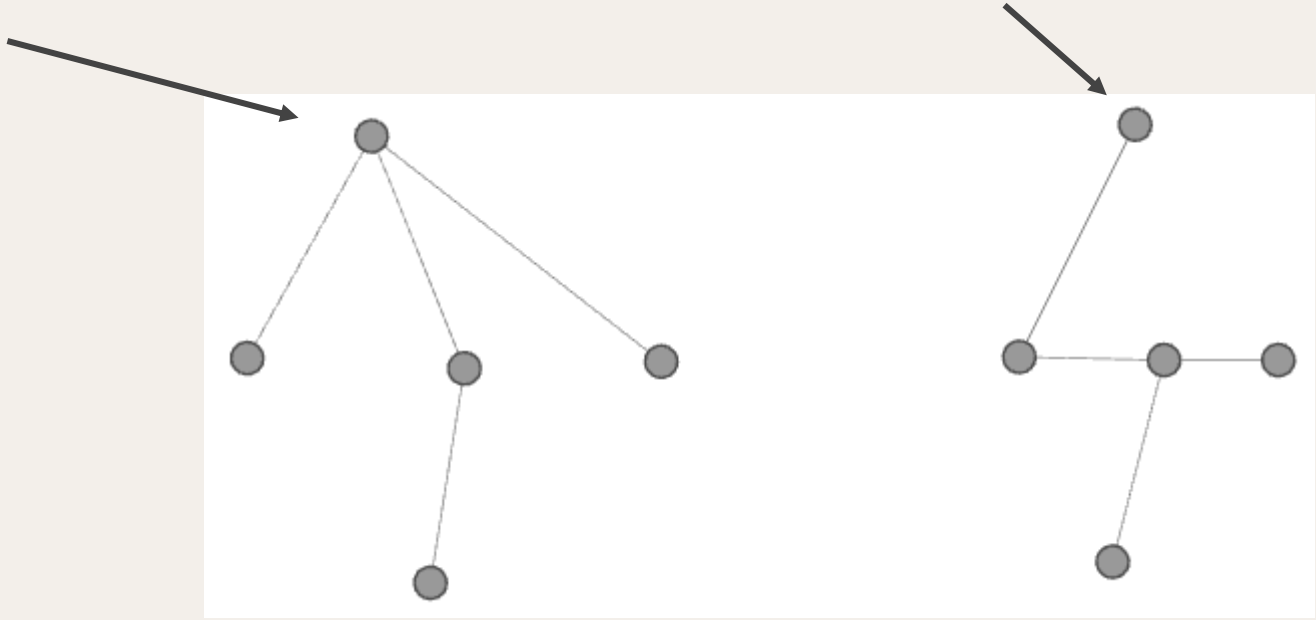
- **Cu legături de tip *tată*** (fiecare nod "ştie" cine este tatăl său): se pot reprezenta arbori neordonaţi (tip de legătură frecvent pentru probleme în care nodurile arborelui reprezintă elementele unei mulţimi şi suntem interesaţi să facem operaţii pe mulţimi: teste de apartenenţă şi reuniuni de mulţimi)

- Un alt tip de reprezentare este cu **tip de legătură *fiu - frate***. Aceasta înseamnă că, pentru fiecare nod al arborelui, avem acces la primul său fiu şi la fraţii săi (nodurile de pe acelaşi nivel se numesc fraţi şi îi organizăm ca listă)

- Pentru arborii binari (mai ales compleți) putem folosi un **vector** cum am făcut la heapuri

Arbori - reprezentări

- relația ***tată - fiu*** (sau doar tată) reprezentată ca muchie sau arc
- legături între frați (heapuri Fibonacci)



Arbori - terminologie

Gradul arborelui = întregul *k* care reprezintă numărul maxim de fii ai unui nod. Un *2*-arbore ordonat se numeşte ***arbore binar*** (fiu stâng, respectiv fiu drept)

Fiecărui nod al arborelui îi vom asocia un ***nivel*** în felul următor:

(a) rădăcina se află la nivelul 0
(b) dacă un nod se află la nivelul *i*, atunci fiii săi sunt la nivelul *i+1*

- Numim ***înălţime*** (sau *adâncime*) a unui arbore nivelul maxim al nodurilor sale.
- Se numeşte ***frunză*** un nod cu gradul 1(și rădăcina poate să fie frunză).
- Se numeşte ***nod interior*** orice nod care nu e terminal.

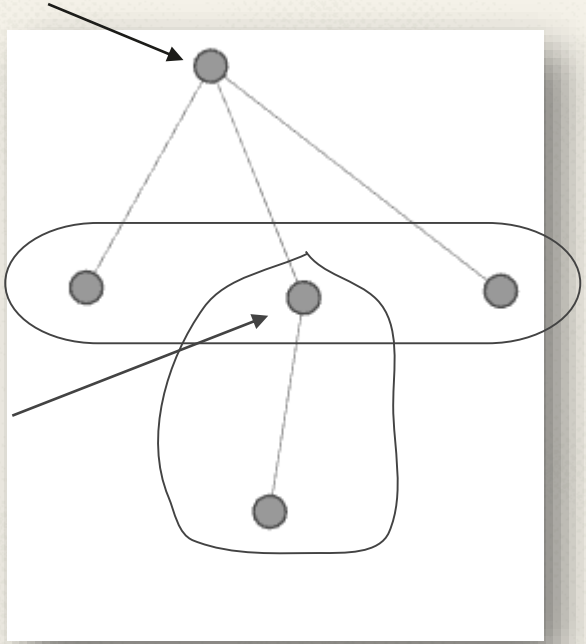
Arbori - terminologie

Rădăcina: nodul fără părinte

Descendenții unui nod: fii, nepoti șamd.

Frați: toate nodurile care au același părinte

Subarbore: arbore format dintr-un nod și descendenții lui.



Arbori binari

Un arbore binar (2-arbore ordonat) T este:

(a) fie un arbore vid $(T=\emptyset)$

(b) fie un arbore nevid ce conţine un nod numit rădăcină, împreună cu doi subarbori binari disjuncţi numiţi subarborele stâng, respectiv subarborele drept

Dacă un nod are un singur fiu, de multe ori va trebui să menționăm dacă este fiul stâng sau drept.

Arbori binari - parcurgeri

- în Preordine (RSD) (*R*ădăcină *S*tânga *D*reapta)
- în Inordine (SRD) (*Stânga R*ădăcină *D*reapta)
- în Postordine (SDR) (*Stânga D*reapta *R*ădăcină)

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);
        }
}
```

TEMĂ (se dau SRD și RSD, trebuie să afișați arborele)

Link pt vizualizare

Bibliografie

Introducere în Algoritmi - Cormen Leiserson Rivest

Curs *Fibonacci Heaps* - University of Cambridge

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:

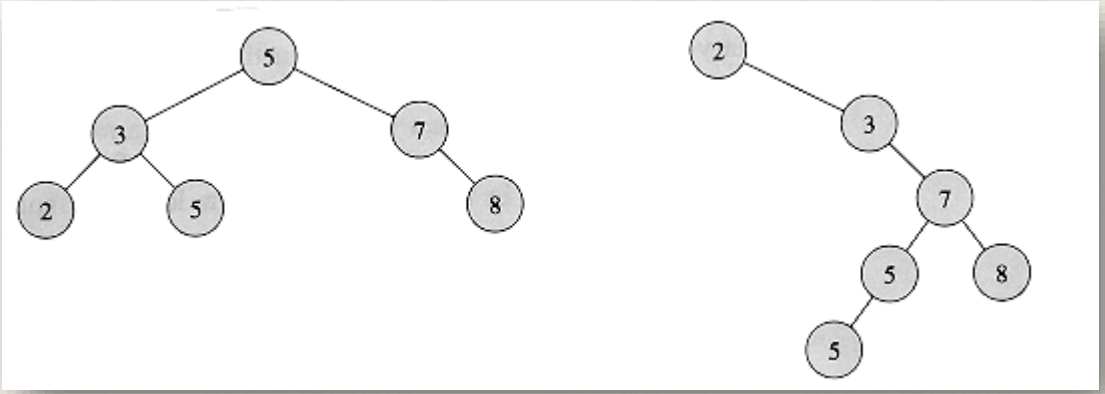
- Dacă **y** este un nod din subarborele ***stâng*** al lui **x**, atunci `cheie[y] ≤ cheie[x]`
- Dacă **y** este un nod din subarborele ***drept*** al lui **x**, atunci `cheie[x] ≤ cheie[y]`

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 `cheie[y] ≤ cheie[x]`
- Dacă **y** este un nod din subarborele ***drept*** al lui **x**, atunci `cheie[x] ≤ 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**.

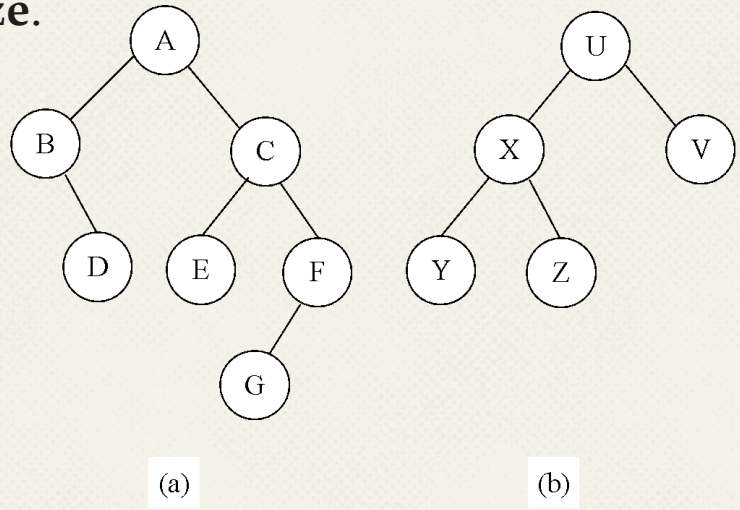
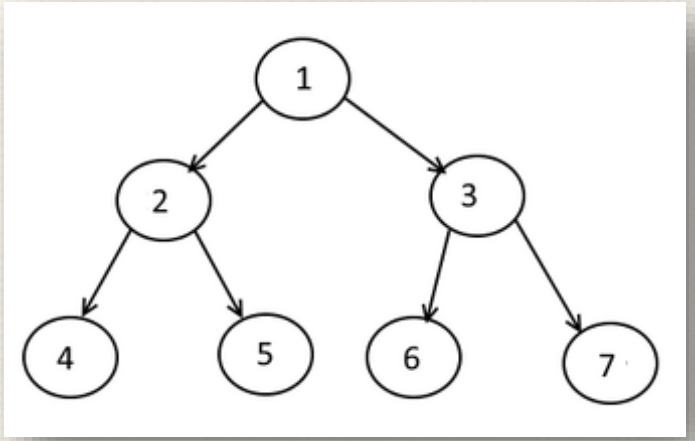


Fig.4.1.1. (a) Un arbore binar nestrict. (b) Arbore binar strict.

Arbori Binari - Parcurgeri

Parcurgeri în arbori binari:

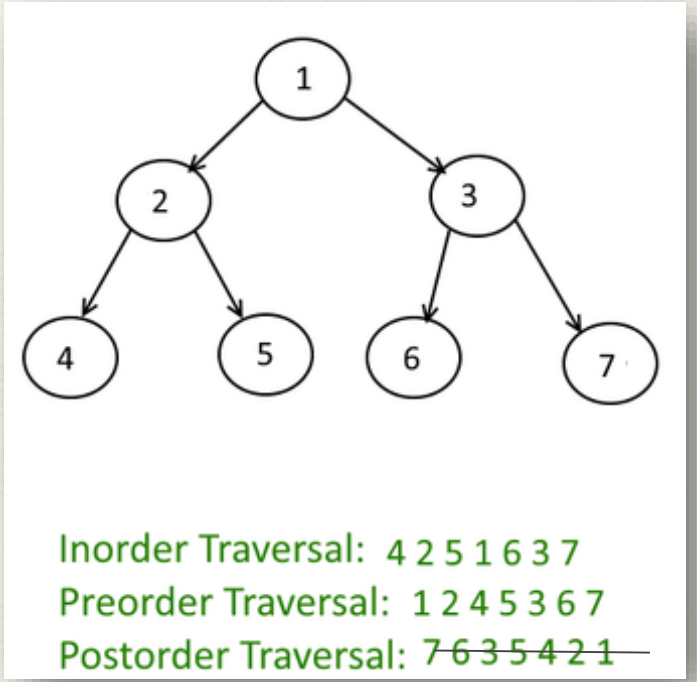
- **Inordine** (SRD, stânga rădăcină dreapta)
- **Preordine** (RSD, rădăcină stânga dreapta)
- **Postordine** (SDR, stânga dreapta rădăcină)



Arbori Binari - Parcurgeri

Parcurgeri în arbori binari:

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



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

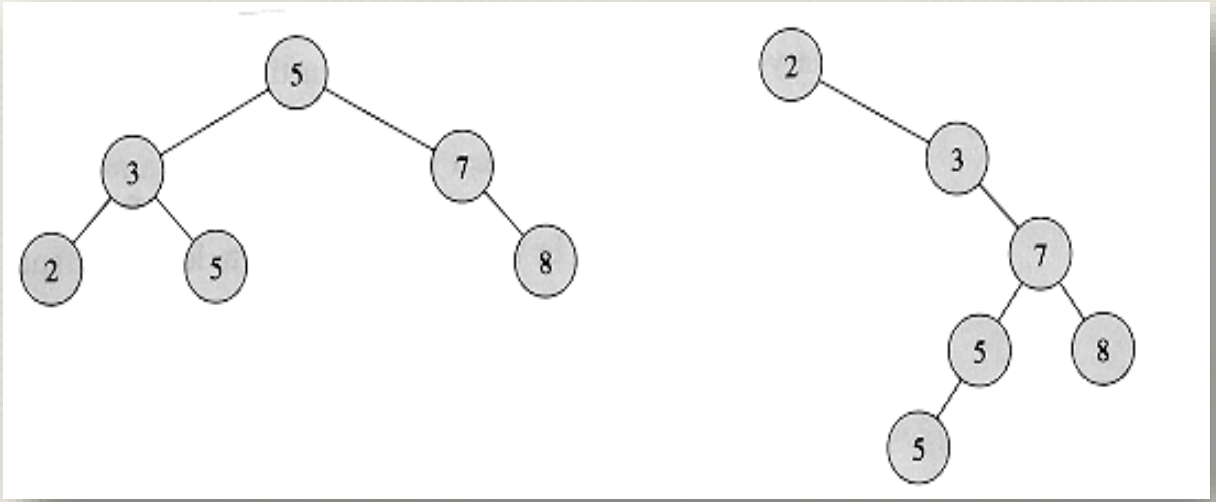
TEMĂ: Se dau SRD și RSD. Afișati arborele

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**

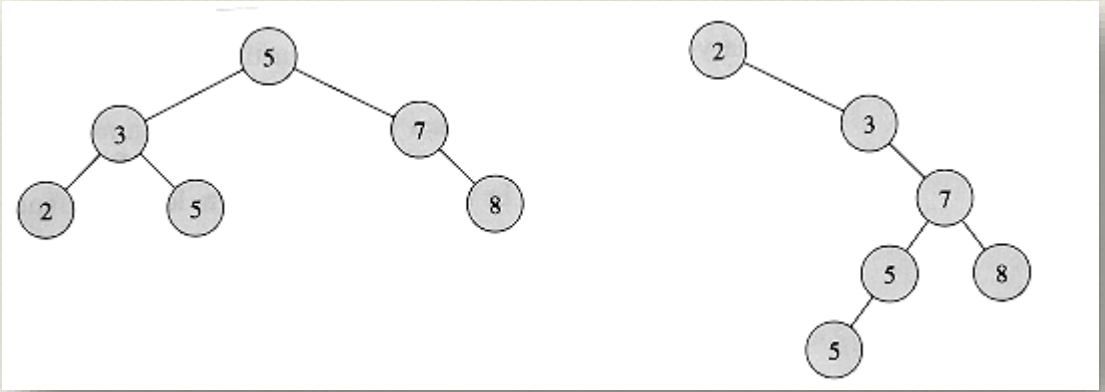
Arbori Binari de Căutare

- Ce parcurgere ne oferă vectorul sortat?
  - Preordine
  - Inordine
  - Postordine



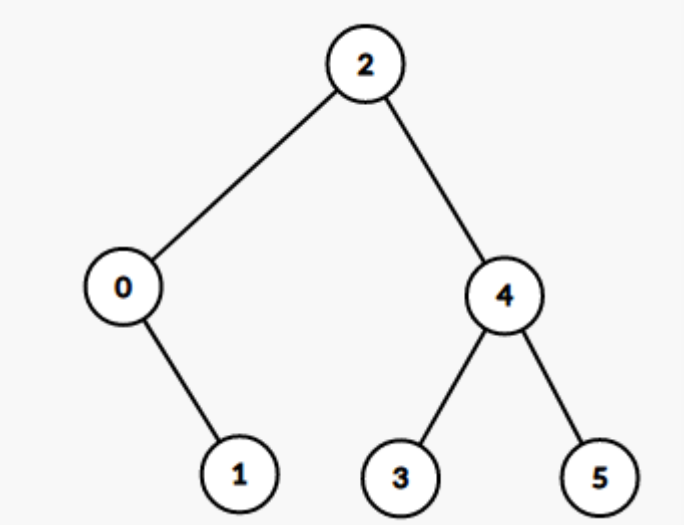
Arbori Binari de Căutare

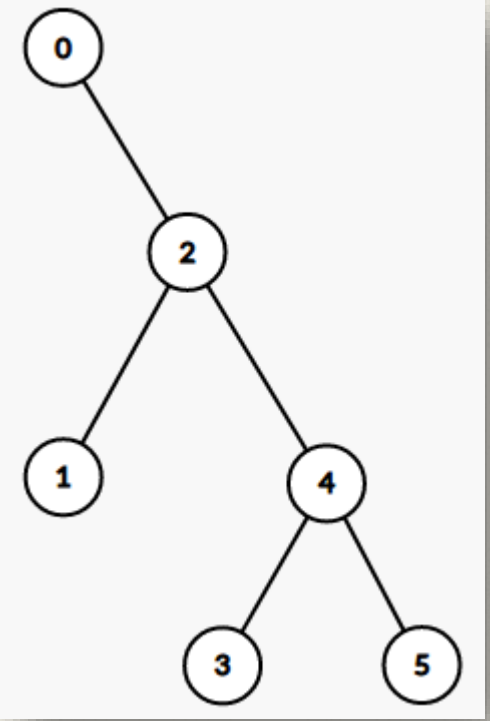
- Parcurgerea **inordine** ne oferă vectorul sortat
  - Preordine 5 3 2 5 7 8 | 2 3 7 5 5 8
  - Inordine 2 3 5 5 7 8 | 2 3 5 5 7 8
  - Postordine 2 5 3 8 7 5 | 5 5 8 7 3 2
- Restul parcurgerilor sunt diferite pentru cei 2 arbori.

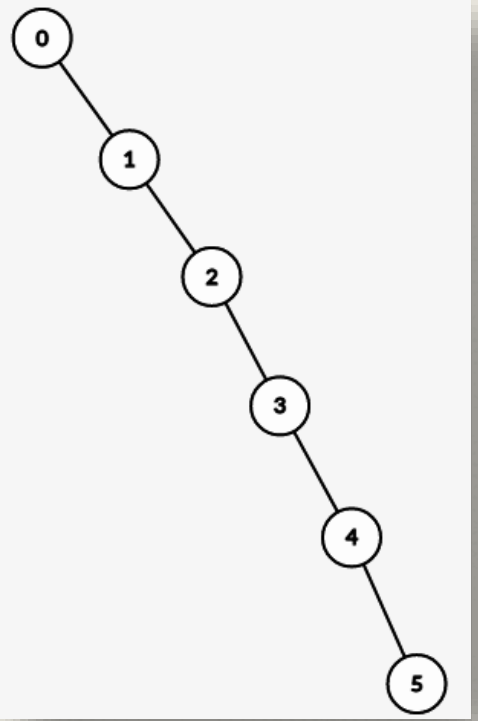


Exercițiu

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

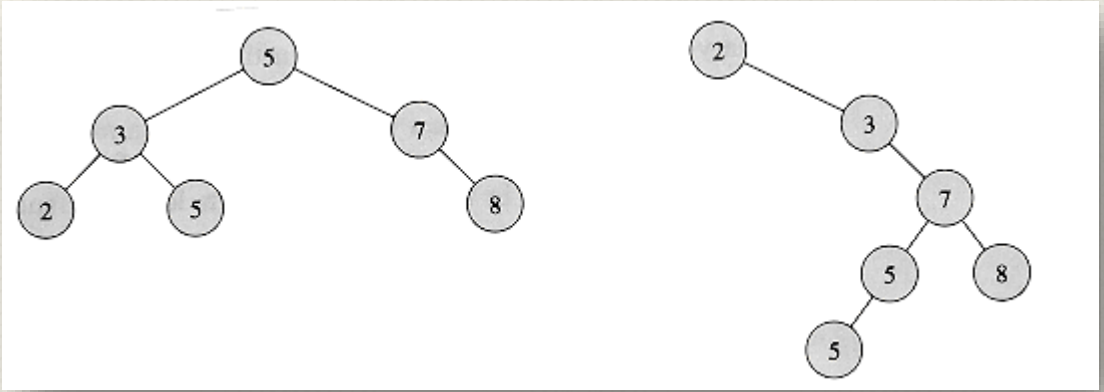






Minim și Maxim

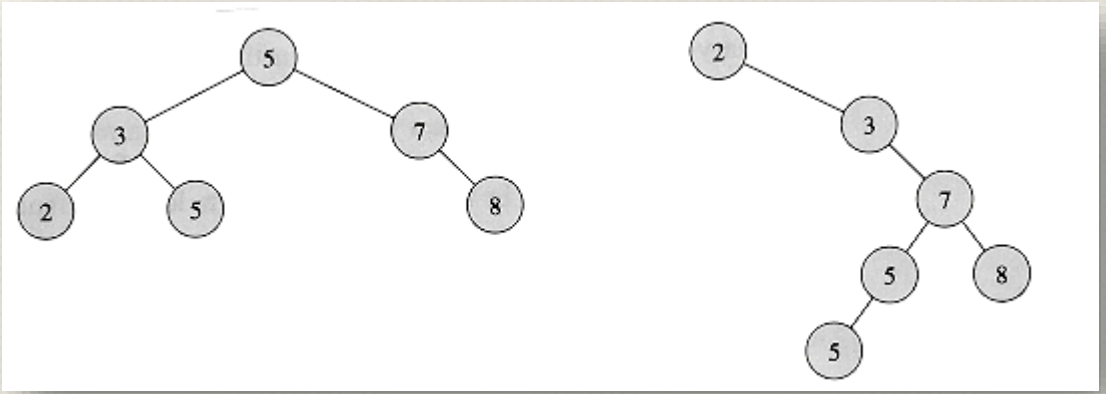
- Unde se află minimul?
  - În cel mai din stânga nod




```
TREE-MINIMUM (x)

1  while left[x] ≠ NIL

2       do x ← left[x]

3  return x
```

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
```

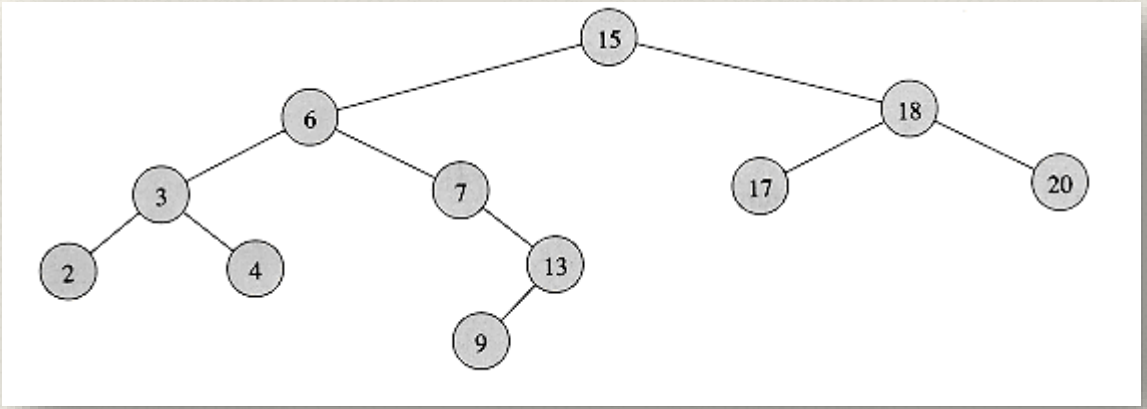
Complexitate?

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.

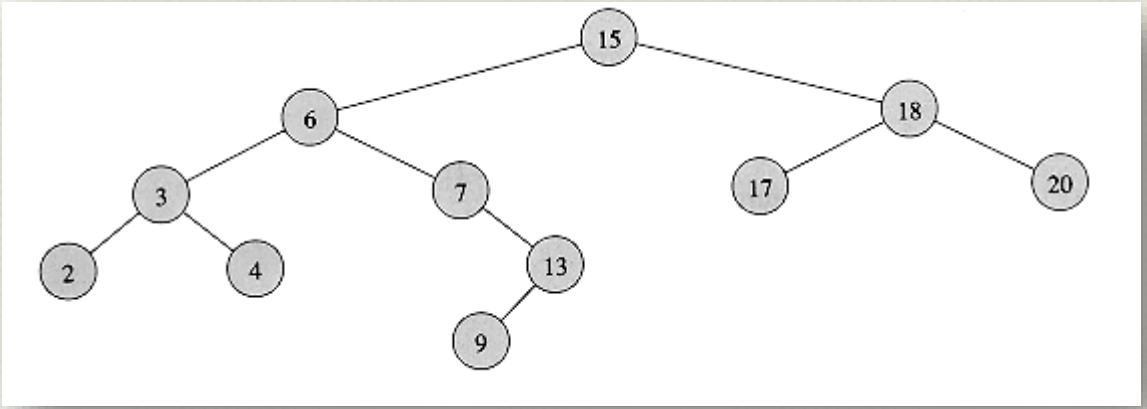
Cum găsim un element?



Căutare

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

Evident, ne oprim dacă am găsit valoarea.

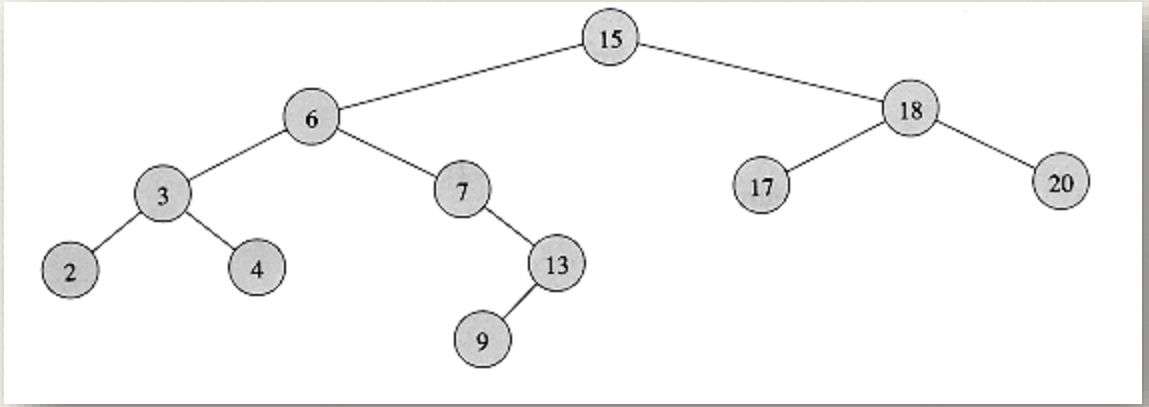


Căutare

```
ITERATIVE-TREE-SEARCH (x,k)
1  while x ≠ NIL and k ≠ key[x]
2      do if k < key[x]
3            then x ← left[x]
4            else x ← right[x]
5  return x
```

Complexitate: **O(h)**

```
TREE-SEARCH (x, k)
1 if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH (left[x], k)
5     else return TREE-SEARCH (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 $\geqslant$ val[x] (valorea nodului)?

Predecesor: Se dă un nod din arbore.
Care este cea mai **mare** valoare din arbore $\leqslant$ val[x] (valorea nodului)?

Cum facem?

Succesor / Predecesor

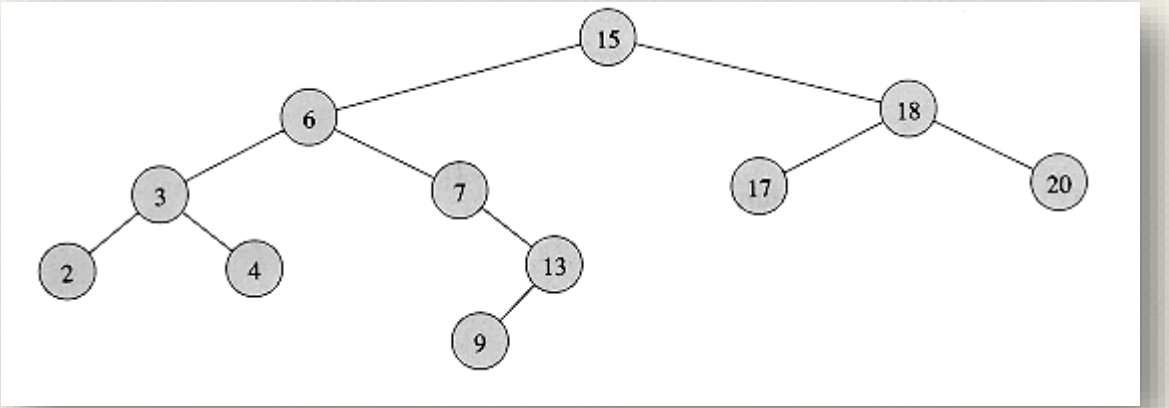
Succesor de 3?

Succesor de 6?

Succesor de 15?

Succesor de 13?

Succesor de 4?



Succesor / Predecesor

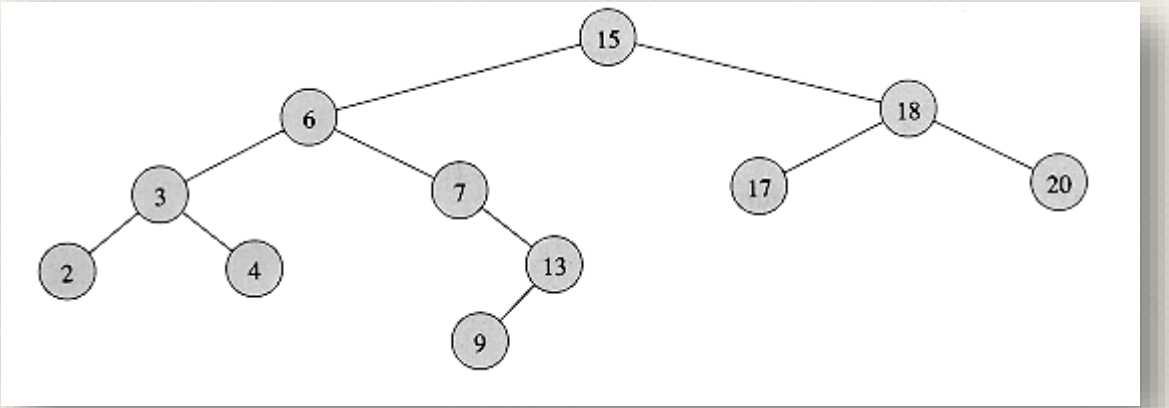
Succesor de 3? → 4

Succesor de 6? → 7

Succesor de 15? → 17

Succesor de 13? → 15

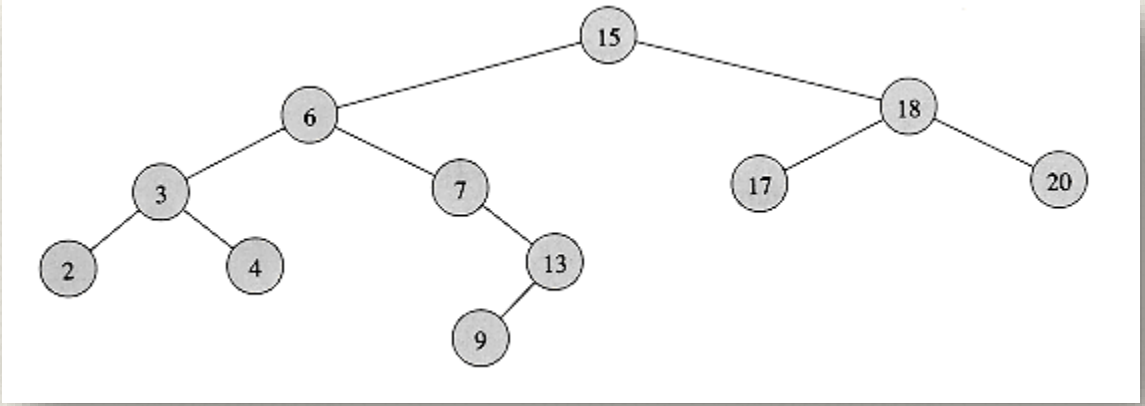
Succesor de 4? → 6



Succesor / Predecesor

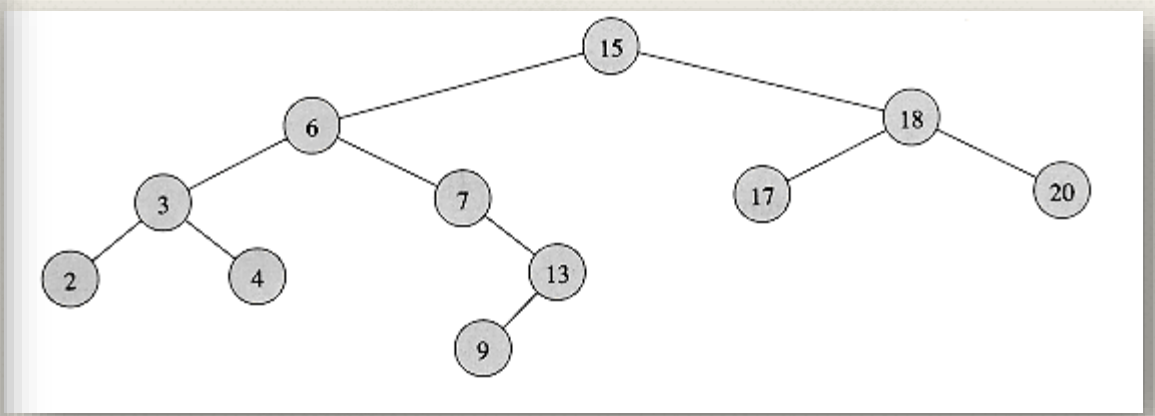
Caz 1) Dacă am fiu drept, atunci cel mai mic element va fi cel mai mic element din subarborele drept. Adică dreapta→stânga→stânga→ … →stânga (vezi 7 sau 15)

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)



Succesor / Predecesor

```
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
```



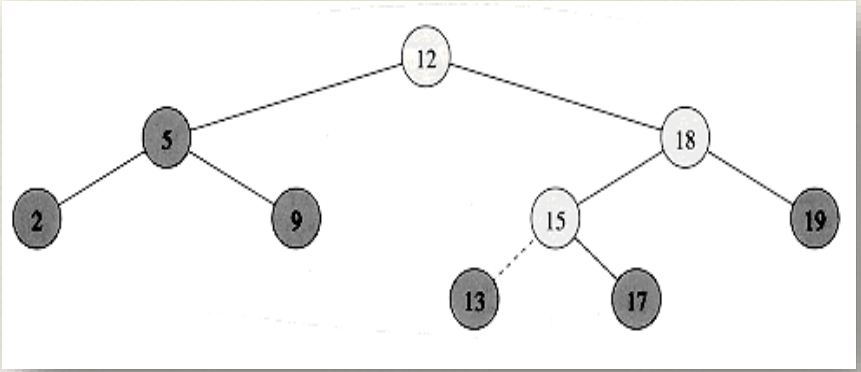
Complexitate: **O(h)**

Inserare

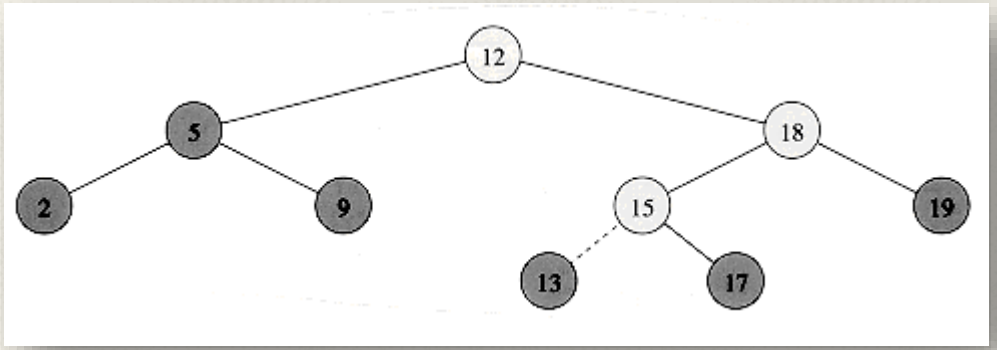
- Similar cu căutarea.
- Începem din rădăcină și avem următoarele cazuri:
  - dacă valoarea dată este identică cu cheia rădăcinii se renunță la inserare;
  - dacă valoarea dată este mai mică decât cheia rădăcinii, se continuă cu subarborele stâng;
  - dacă valoarea dată este mai mare decât cheia rădăcinii, se continuă cu subarborele drept;
- Se continuă acest proces recursiv până când se ajunge la un nod fără copii, moment în care noul nod este inserat ca fiu al acestui nod.

  Obs. După inserare, proprietatea de arbore binar de căutare trebuie să fie respectată

Inserare 13:



Inserare

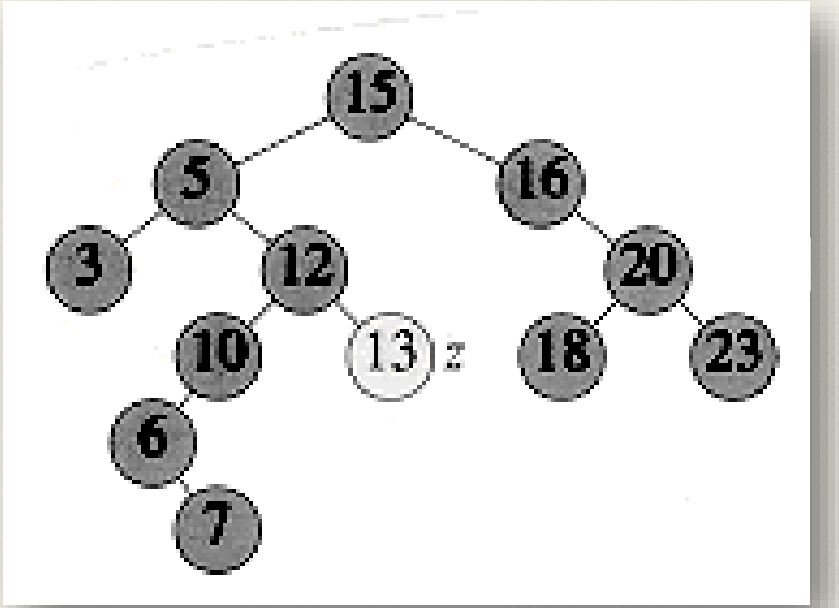


Complexitate: **O(h)**

```
TREE-INSERT(T,z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4        do y ← x
5           if key[z] < key[x]
6              then x ← left[x]
7              else x ← right[x]
8  p[z] ← y
9  if y = NIL
10       then root[T] ← z
11       else if key[z] < key[y]
12                then left[y] ← z
13                else right[y] ← z
```

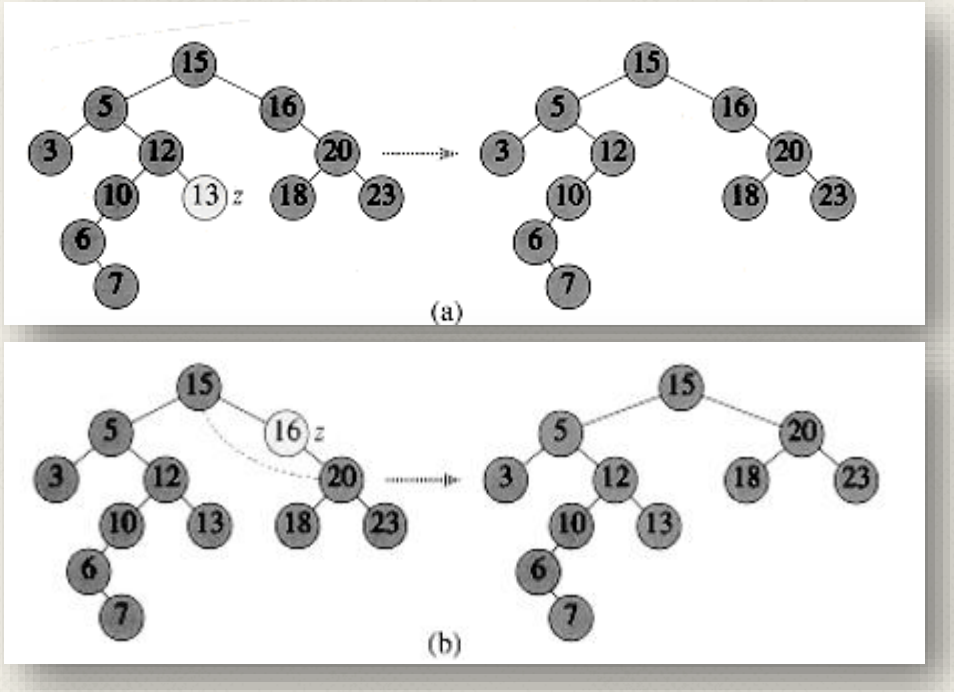
Ștergere

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



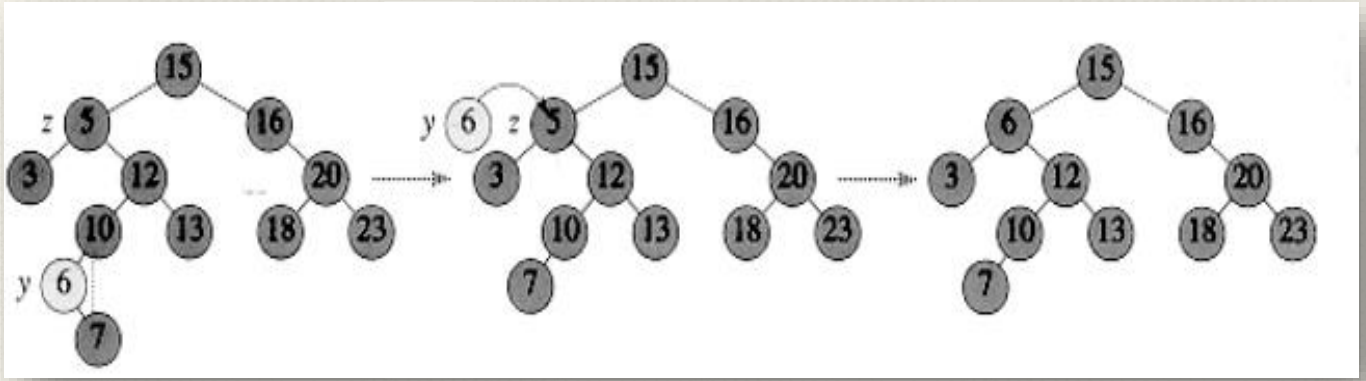
Ștergere

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



Ștergere

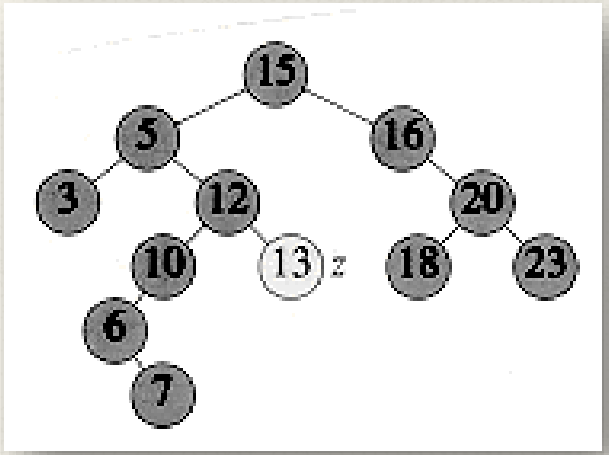
- Cum?
- Cum îl ștergem pe 13?
- Dar pe 7?
- Pe 5?
- Dar pe 15?



Ștergere

Avem 3 cazuri:

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




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

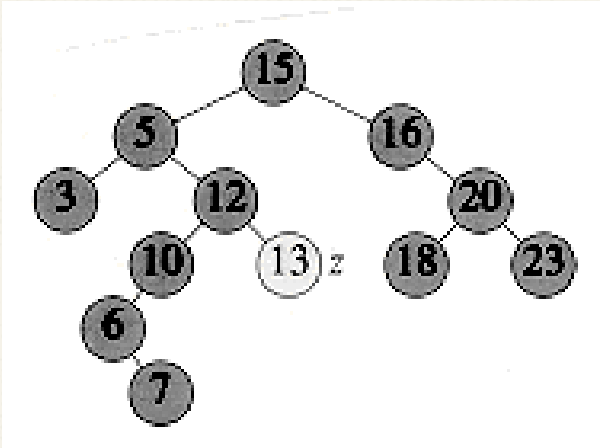
Ștergere

Avem 3 cazuri:

3) Dacă are **ambii** fii, găsim succesorul său,

   îl punem în locul său și înlocuim legătura tatălui acestui nod cu singurul fiu

   (dacă există)




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

Ștergere

Exercițiu:

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

Complexitate

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

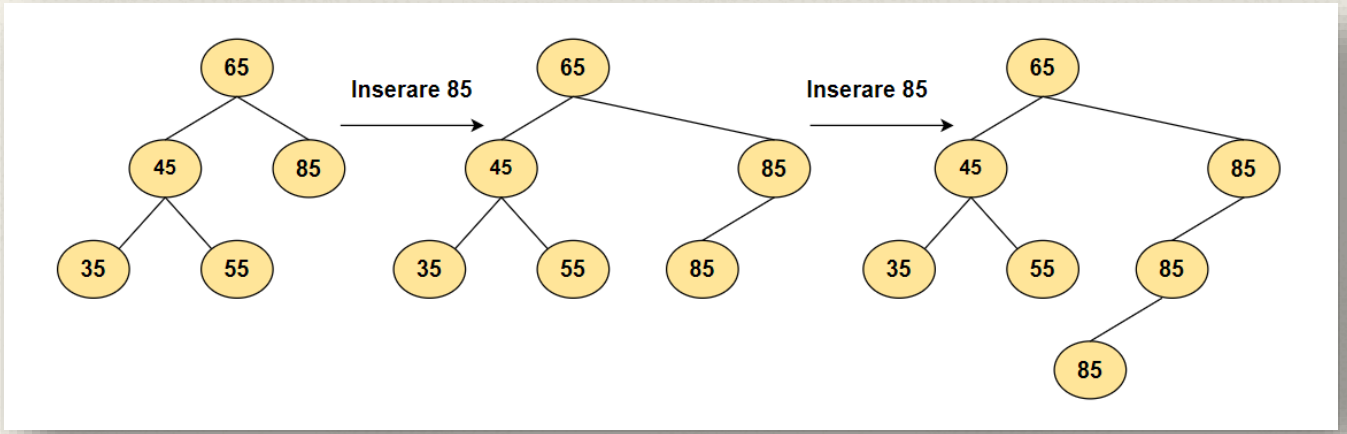
Complexitate

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 ?

- 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

Final