1) Ordonat 2) Neordonat

C(t)

adăugare - ca la ștergere - $\Theta(1)$ dacă avem size amortizat

ștergere - $\Theta(\log_2 n + n) = \Theta(n)$

căutare - $\Theta(\log_2 n)$ - $\Theta(n)$

V. dinamic

$C(t) = fr.ad * C(ad) + fr.st. * C(st.) + fr.caut. * C(c.)$

Av.
- cap. din.

Des.
- când resize
- memory leaks

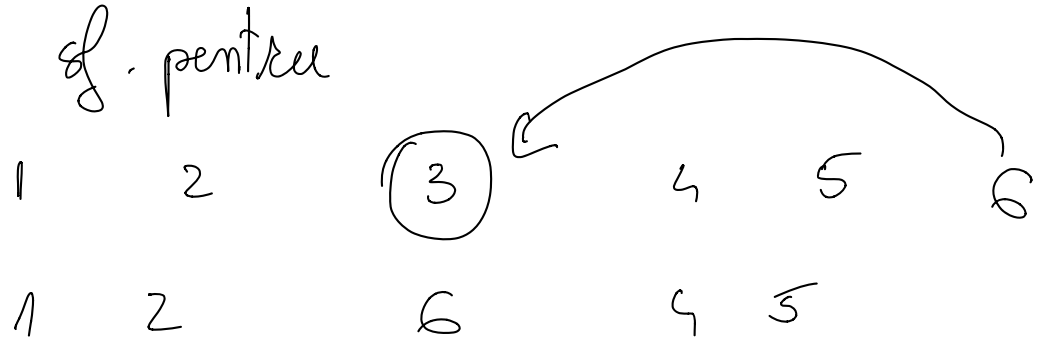
comp. ad. : $\Theta(1)$

comp. resize : $\Theta(n)$

$\Theta(am.) = \Theta(1) + \frac{1}{n} \cdot \Theta(n) = \Theta(1) + \Theta(1) = \Theta(1)$

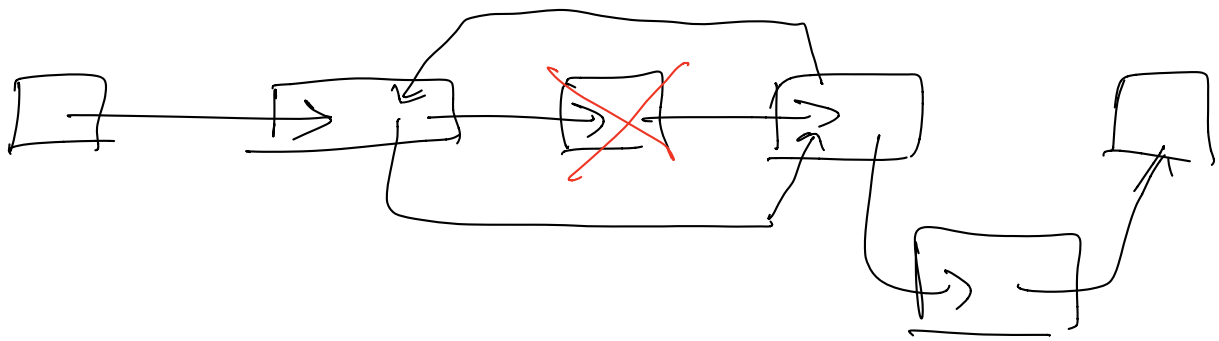
găsim pe poz. i: $\Theta(i)$ + mutăm: $\Theta(n-i)$ = $\Theta(n)$

```
șterge(v, e):
    {pre: v - vect. dinamic, e: TElem}
    {post: e ∈ v}
    pentru i = 1..v.size:
        dacă v[i] = e atunci
            v[i] ← v[v.size]
            v.size ← v.size - 1
            return
        sf. dacă
```



Listă înlănțuită

	Ord.		Nord.
ad.	$-O(n)$		$-\Theta(1)$ (la început ll final)
ș.	$-O(n)$	$<$	$-O(n)$
căut.	$-O(n)$	$<$	$-O(n)$



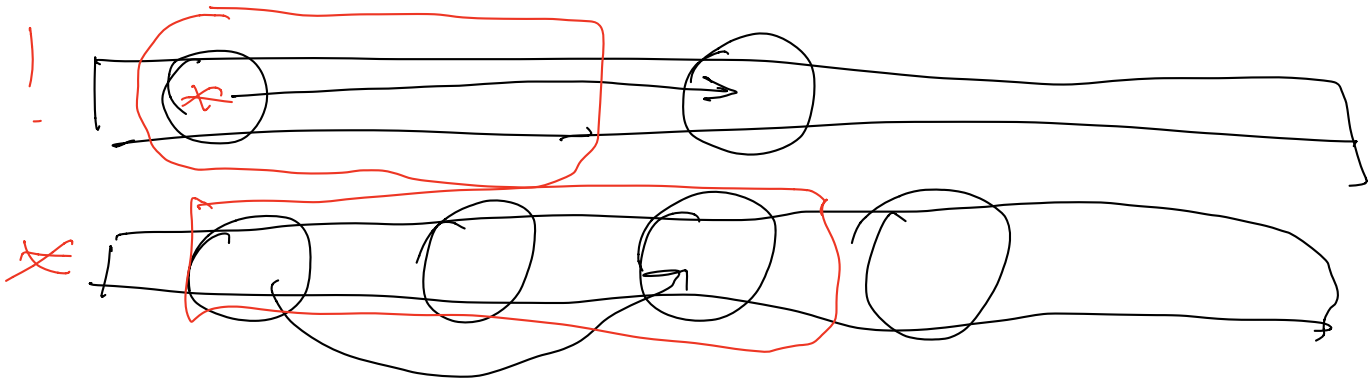
I dinamic

```
Nod {
  TElem data
  ↑Nod next
}
LL {
  ↑Nod start
}
```

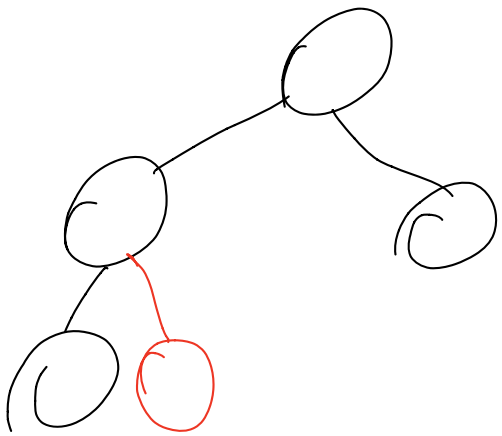
II pe tablou

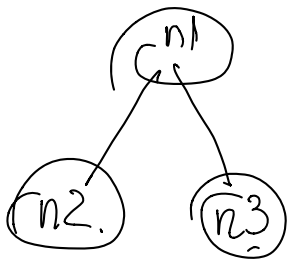
```
LL {
  data: TElem[]
  next: Integer[]
  prev: Integer[]
}
```

urm: p → next l.data[l.next[p]]



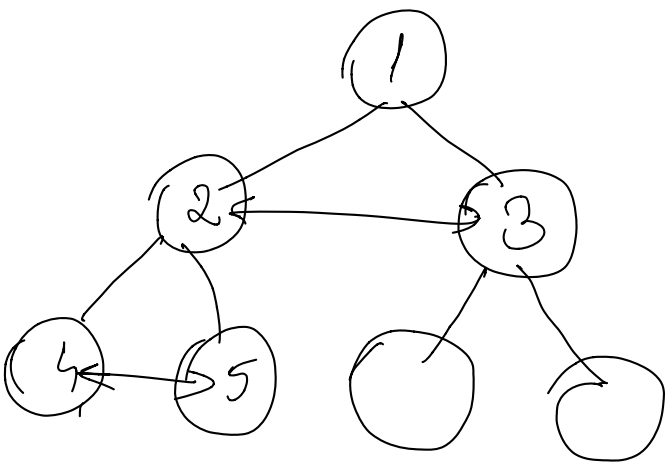
Ansamblu = Heap





R ≤

$$\begin{cases} n_1 \le n_2 \\ n_1 \le n_3 \end{cases}$$

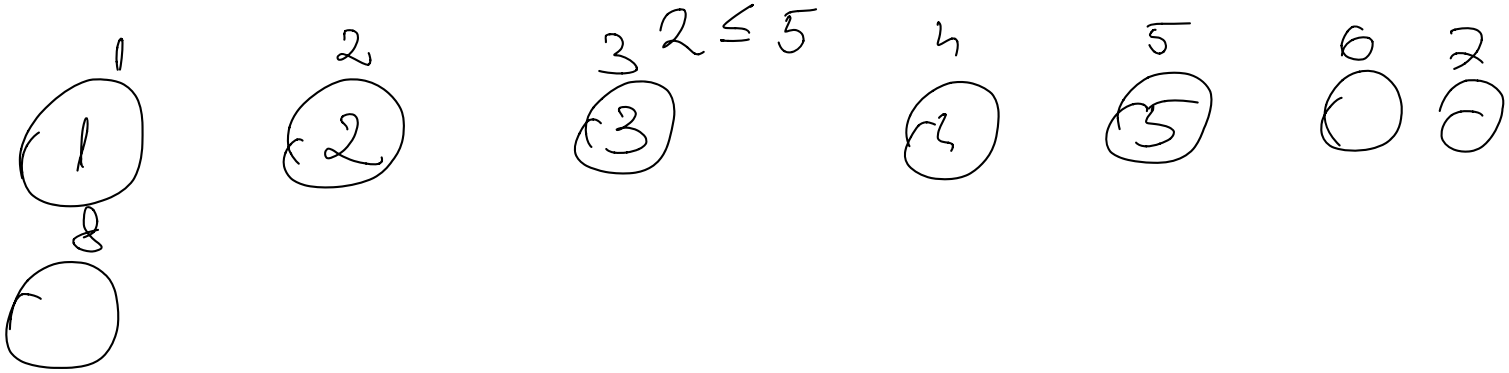


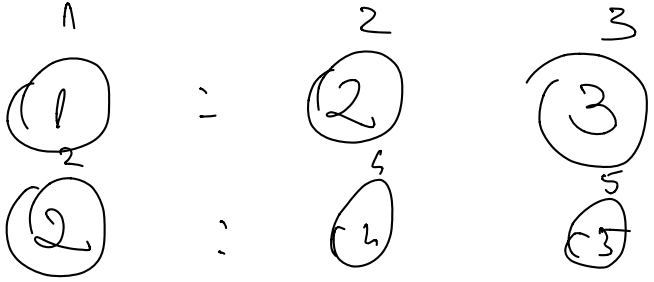
$1 \le 2$

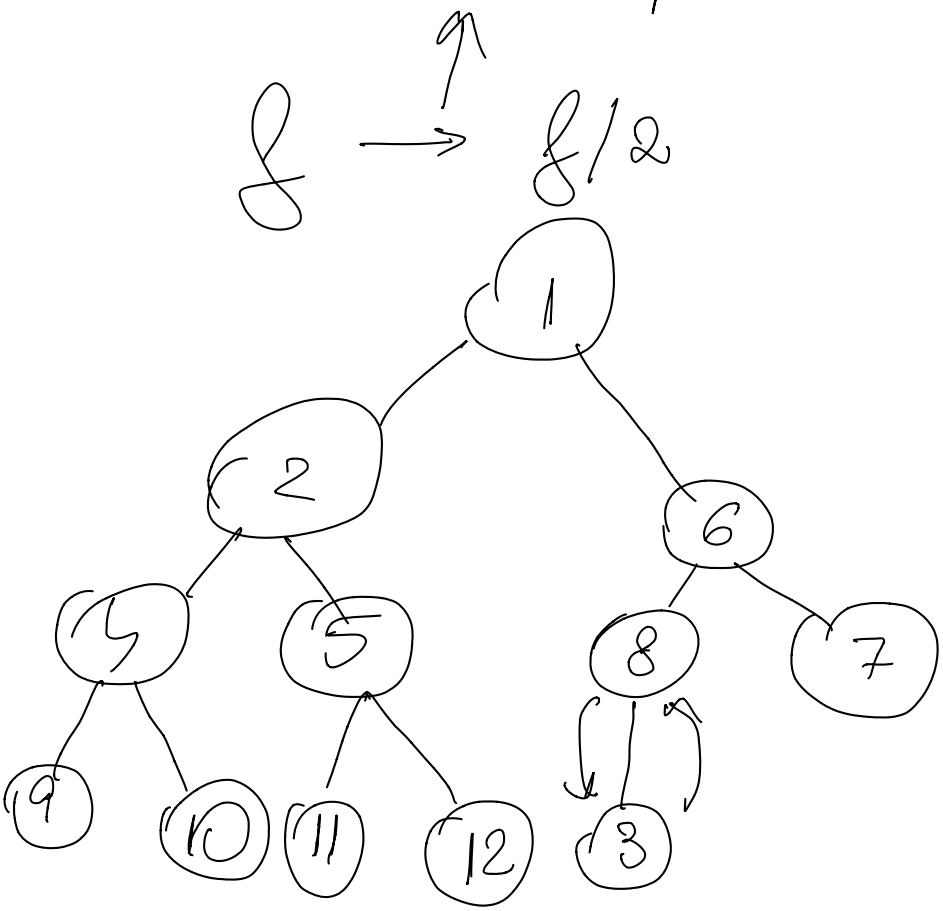
$1 \le 3$

$2 \le 4$

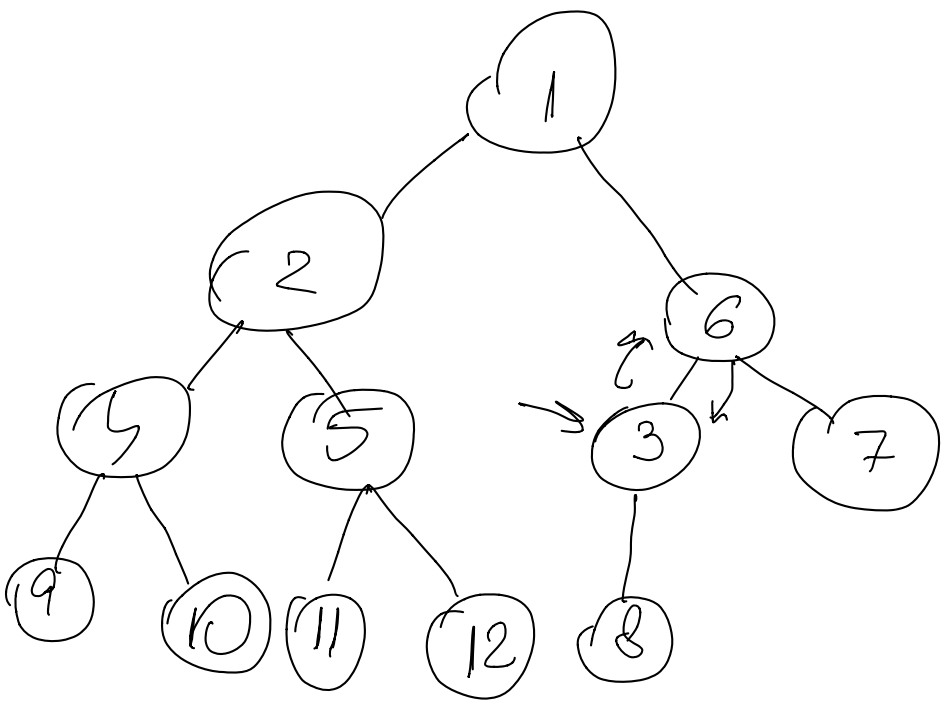
$2 \le 5$



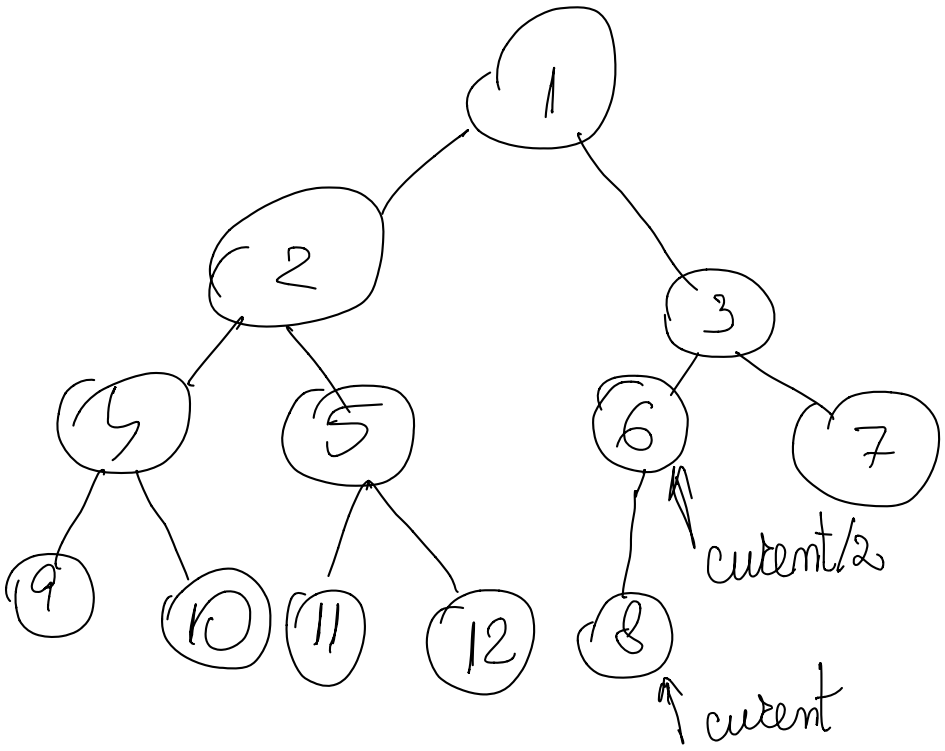




$8 \leq 3$?



$6 \leq 3$?



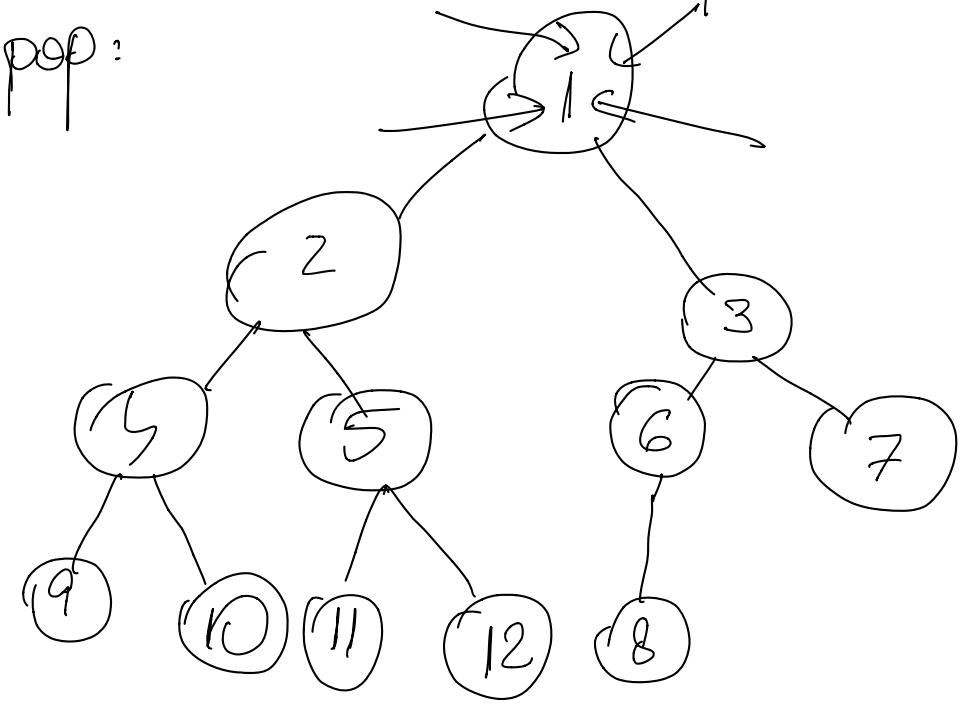
$1 \leq 3$

ad. : $O(\log_2 n)$

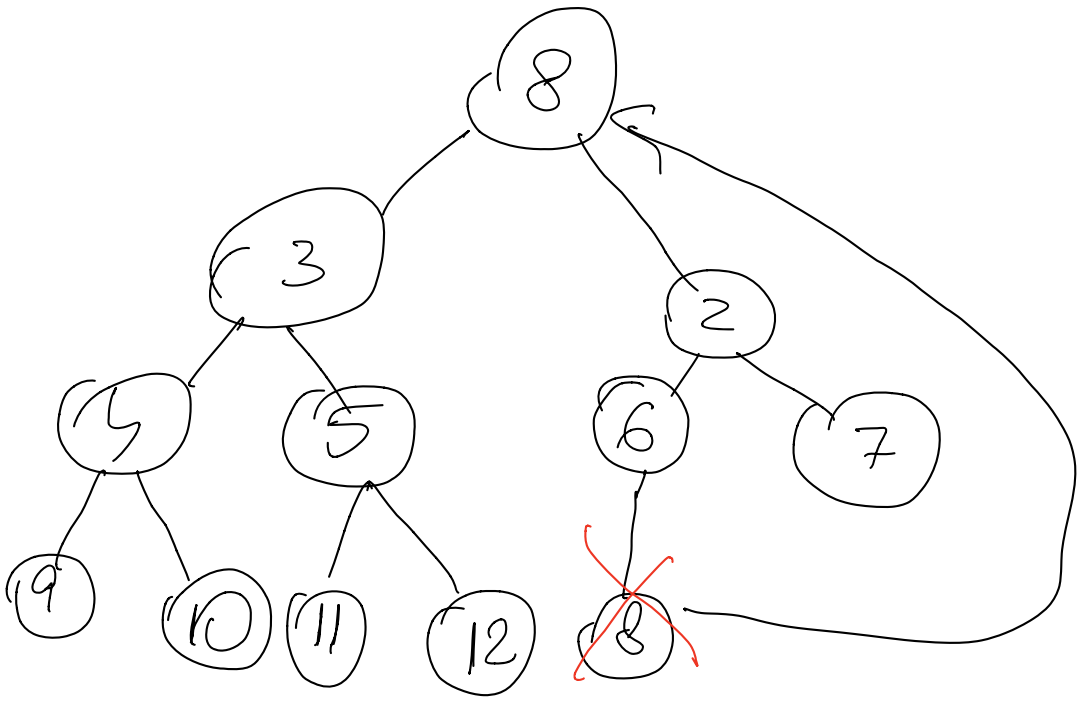
```
adauga(h, e) {
    h[h.size] ← e
    curent ← h.size
    h.size ← h.size + 1
    CatTimp curent>1 si !R(curent/2, curent):
        Interschimba(h.elem[curent/2], h.elem[curent])
        curent ← curent/2
    sfCatTimp
}
```

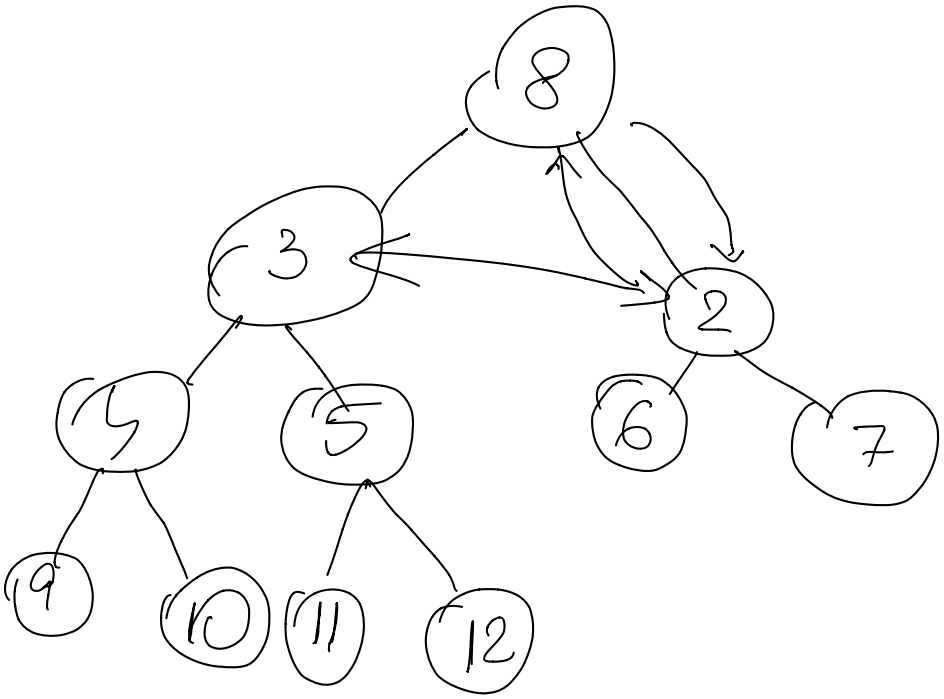
min heap

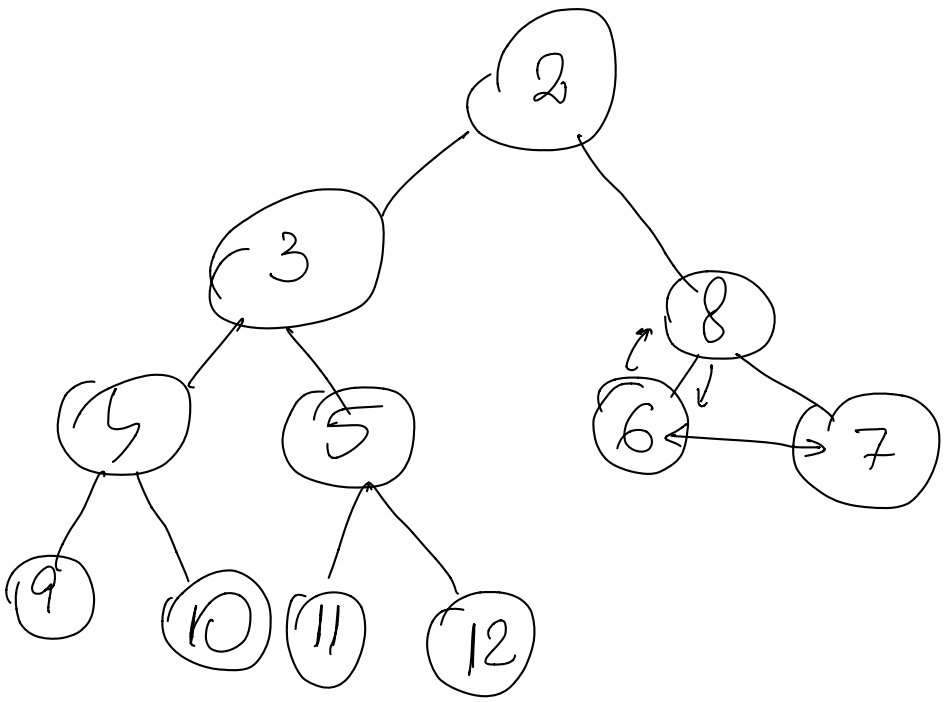
pop:

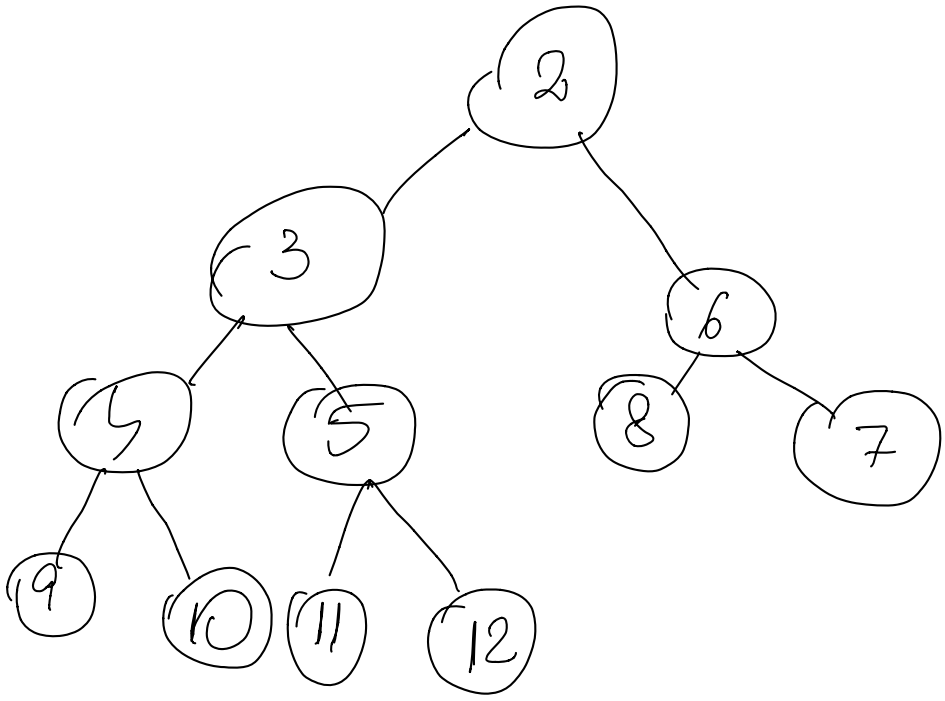


R ≤




8
3
2
4
5
6
7
9
10
11
12


2
3
8
4
5
6
7
9
10
11
12




```
pop ( h ) h   // O(log2 n)
    returnValue ← h.elem[1]
    current ← 1
    h.elem[1] ← h.elem[h.size]
    h.size ← h.size - 1
    CâtTimp current <= h.size   și
        (!R(current, 2*current) sau !R(current, 2*current+1)):
        Dacă R(2*current, 2*current+1):
            Interschimbă(h.elem[current], h.elem[2*current])
            current ← 2*current
        altfel
            Interschimbă(h.elem[current], h.elem[2*current+1])
            current ← 2*current+1
        Sf.Dacă
    Sf.CâtTimp
```

pop ← return value

}

$C(t) = \Theta(k * \log_2 n)$

//scoatem al k-lea element cel mai prioritate

```
for(int i=0; i<k-1; i++)
    v[i] = pop(h)
r = pop(h)
for(int i=0; i<k-1; i++)
    push(h, v[i])
return r
```

$C(t) = (k-1) \cdot O(\log_2 n) + O(\log_2 n) + (k-1) \cdot \Theta(\log_2 n) =$

$= \Theta(k * \log_2 n)$

$g(\log_2 n) < \Theta(\log_2 n) < f(\log_2 n)$

$h(\log_2 n) < O(\log_2 n) < f'(\log_2 n)$

(+)

$g(\log_2 n) + h(\log_2 n) < \Theta + O < f''(\log_2 n)$

$g(\log_2 n) + h(\log_2 n) = g'(\log_2 n)$

$\Downarrow$

Θ

Av: extragere minim rapidă + inserare

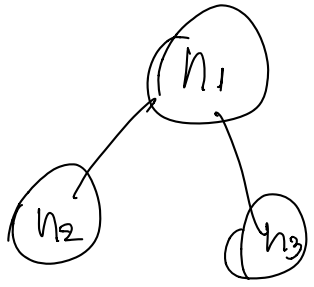
Dez: nu ai versatilitate

W13 : 2 teste SO + practică Gr + test lab Gr

⇕

W14 : pr. SO + examen Gr

ABC preview :



$$\begin{cases} n_1 > n_2 \\ n_1 < n_3 \\ n_2 < n_3 \end{cases}$$

Tabelă de dispersie = Hash table

liste independente

LL[hash(c)]

cu pp. că LL are puține elem

vector de LL

HasTable {

list : LL[]

}

presupunem că hash e bună

0 --- allsize - 1 — dispersează bine

hash(c) = f(c) % allsize

LL:
1) localizezi LL[hash(c)] } O(1)
2) Ad. LL

coliziune
!!
h(c1) = h(c2), deși c1 ≠ c2

adauga(ht, e, c) {
poz = hash(c)
ht.list[poz] = Nod_nou(e, c, ht.list[poz])
}
data key next

Nod_nou(e, c, next) {
@Alocă n;
n → data = e
n → key = c
n → next = next
Nod_nou = n { așa se returnează }
}

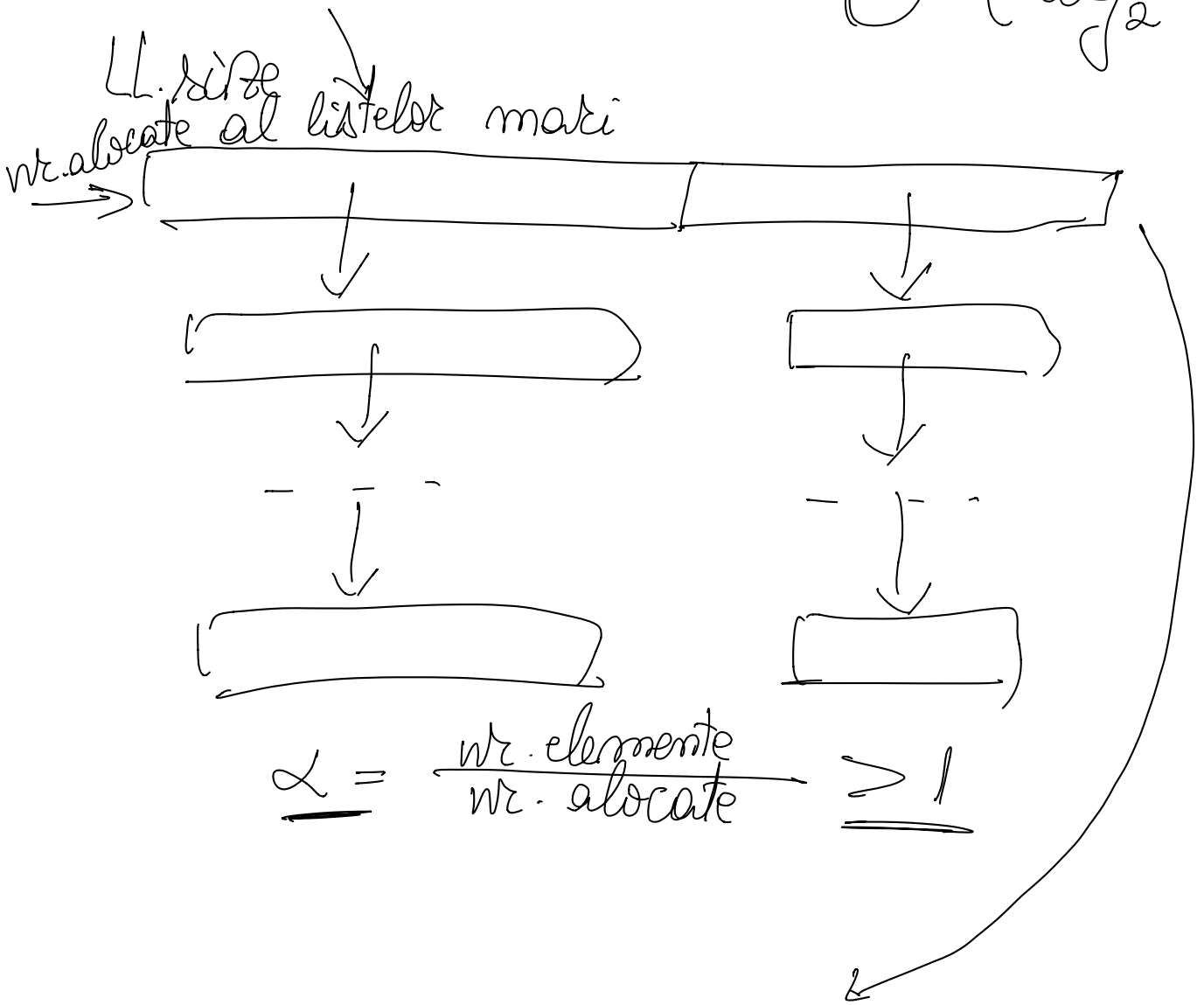
caută:
1) localizezi LL[hash(c)] } O(LL.size)
2) parcurgere LL[hash(c)] }

$\Theta(1)$ $\Theta(2) = \Theta(1)$

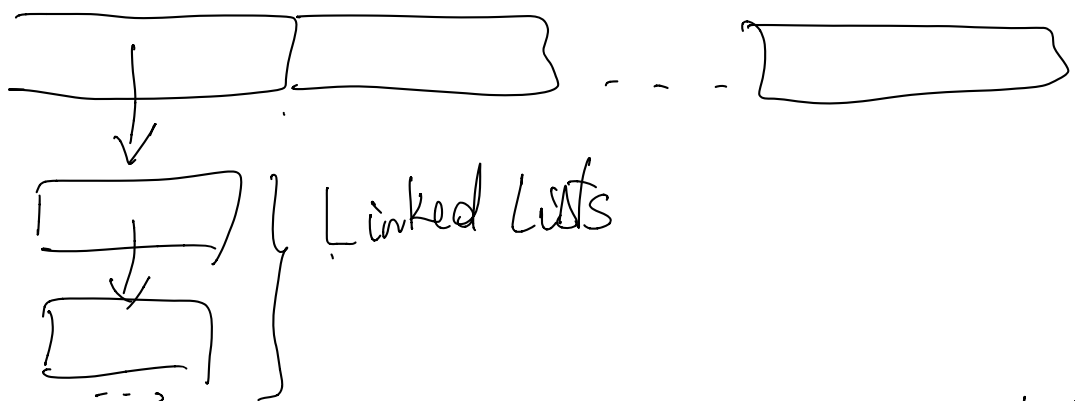
$\Theta(\log_2 n) \neq \Theta(\log_2 n^2)$

$\Theta(2\log_2 n)$

$\Theta(\log_2 n)$



$\alpha = \frac{\text{nr. elemente}}{\text{nr. alocate}} \geq 1$



Rehash fiecare element în locul său

ad. $\Rightarrow \Theta(1)$ $\quad n$ = allocated size

ad * n $\Rightarrow \Theta(n)$

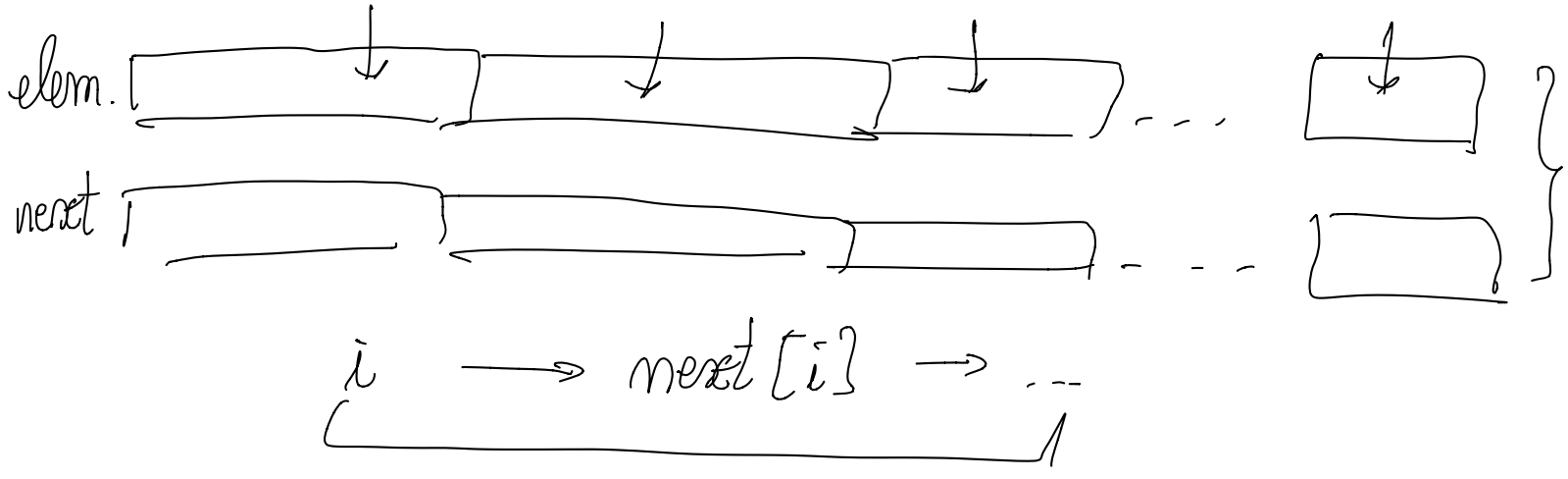
ad. amortizate $= \Theta(1) + \frac{1}{n}\Theta(n) = \Theta(1)$

căutare = $O(LL[hash(c)].size)$,
pentru un size mic, e $\Theta(1)$

ștergere = $O(LL[hash(c)].size) + \Theta(1)$

accesare element: căutare

Hash Table cu liste intrepatrunse
- cu adresare deschisă.



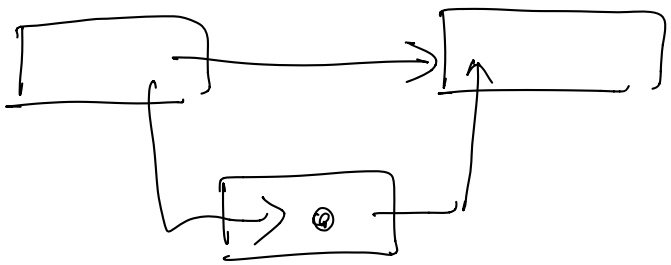
i $\longrightarrow$ next[i] $\longrightarrow$ ...

adăugare (ht, e, c)
@ resize eventual
poz $\leftarrow$ hash(c)
dacă ht.elem[poz] = NIL atunci
ht.elem[poz] $\leftarrow$ (e, c)
dacă poz = primLiber atunci
@update primLiber
sf dacă
return
sf dacă

comentarii neesențiale → } Cât Timp ht.next[poz] != NIL atunci {
} poz ← ht.next[poz] {
} Sf. Cât Timp {

ht.elem[primLiber] = ht.elem[poz]
ht.next[primLiber] = ht.next[poz]
ht.elem[poz] = (e, c)
ht.next[poz] = primLiber
@update primLiber ; parcurge intervalul (primLiber, ht.size]
// până găsește primul el. liber

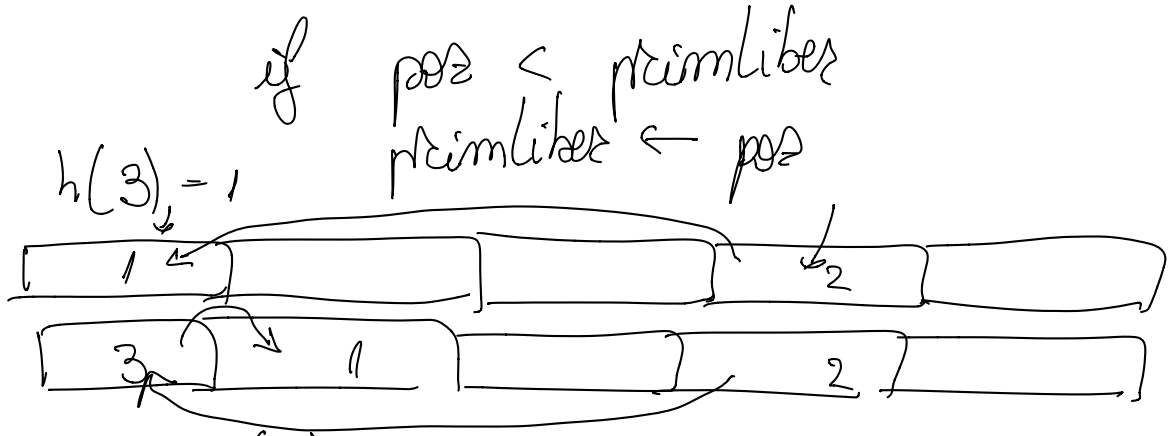
}



Ștergere {

- - -

if poz < primLiber
primLiber ← poz



$h(3) \neq h(2) = h(1)$

$h(c, i) = (h'(c) + i * h''(c)) \% size$

```
for (int i = 0; i < size; i++)
    // daca h(c, i) este libera
    // adaug pe h(c, i) si return
```

resize

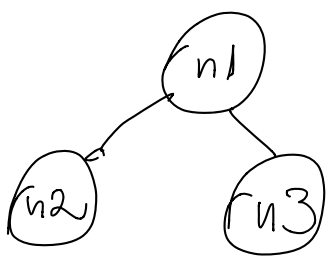
adaugă.

$h(c, size) = (h'(c) + size * h''(c)) \% size =$

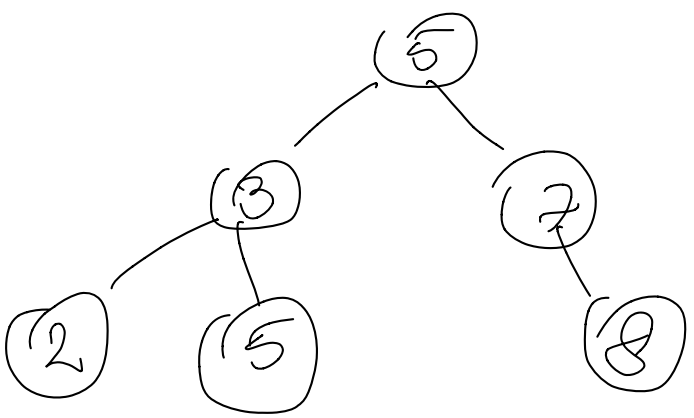
$= h'(c) \% size + (size * h''(c)) \% size =$

$= h'(c) \% size = h(c, 0)$ (unde $i = 0$)

ABC



$n_2 < n_1 < n_3$



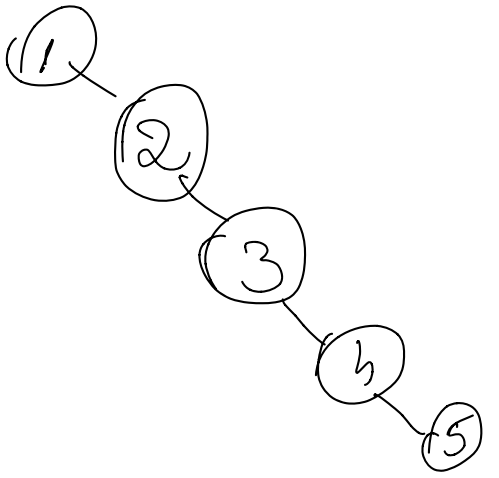
echilibrat (ex: AVL)

$|nr\ stg - nr\ dr| \leq 1$

ad: $O(\log_2 n)$

căutare: $O(\log_2 n)$

ștergere: $O(\log_2 n)$



degenerat

ad: $O(n)$

căutare: $O(n)$

ștergere: $O(n)$

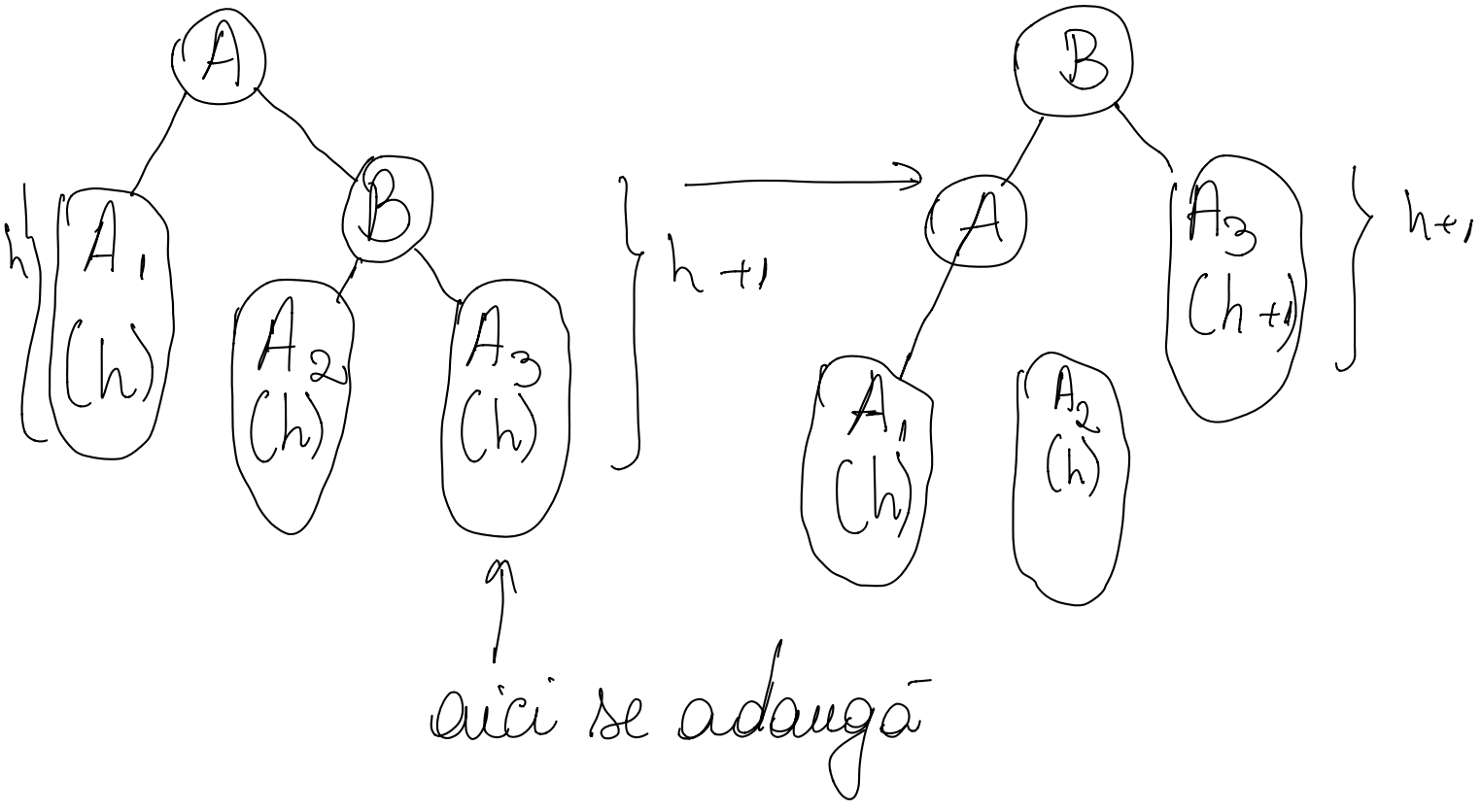
AVL:

4 tipuri de rotații:

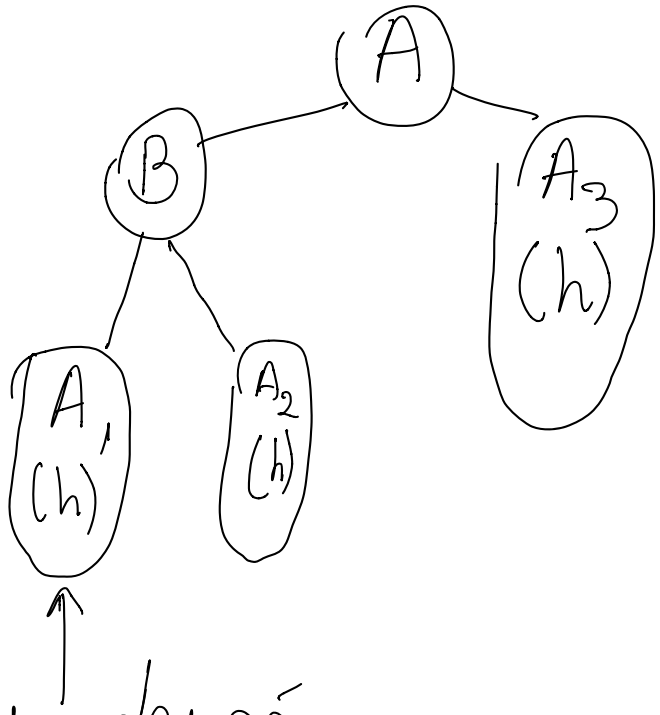
- SRS
- SRD
- DRS
- DRD

S/D	R	S/D
↑ simplă/ dublă	↑ rotație	↑ stânga/ dreapta

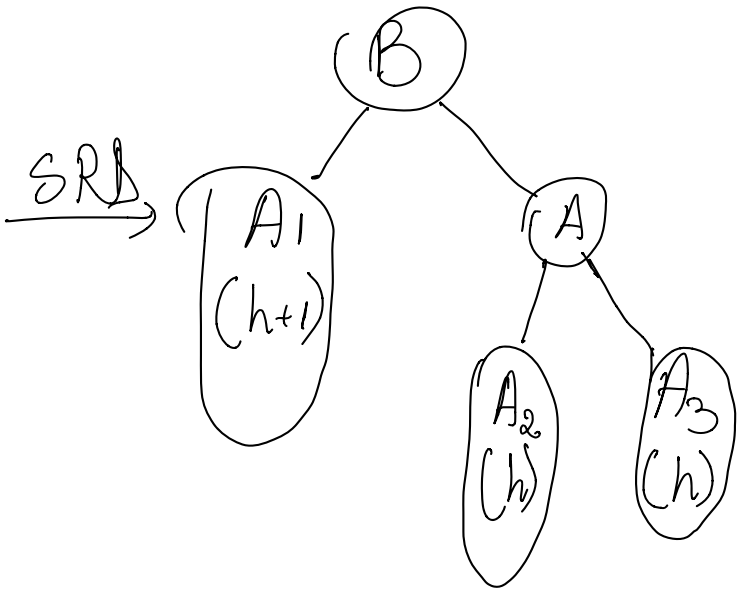
SRS



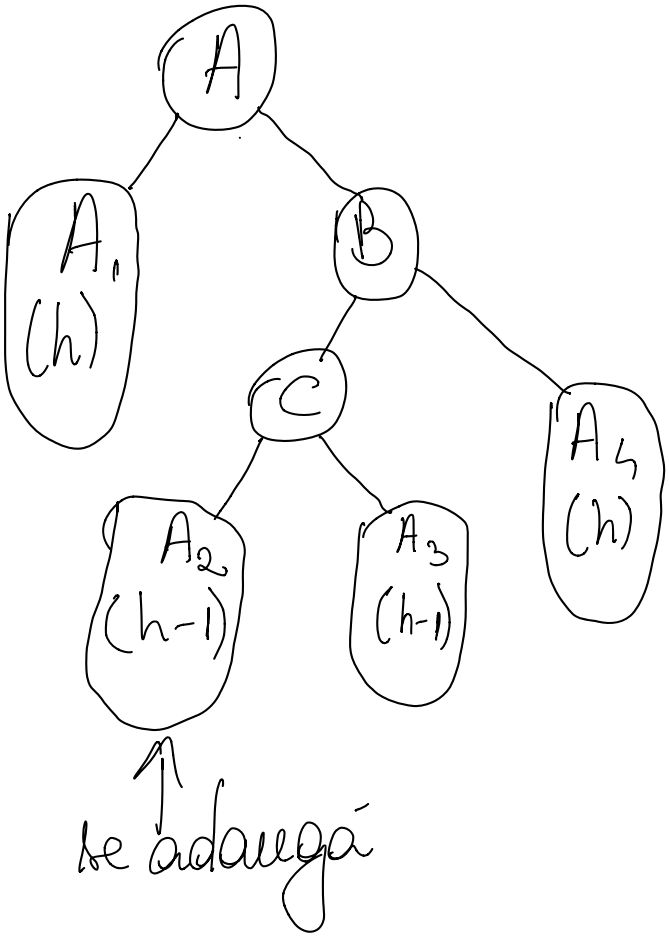
SRD



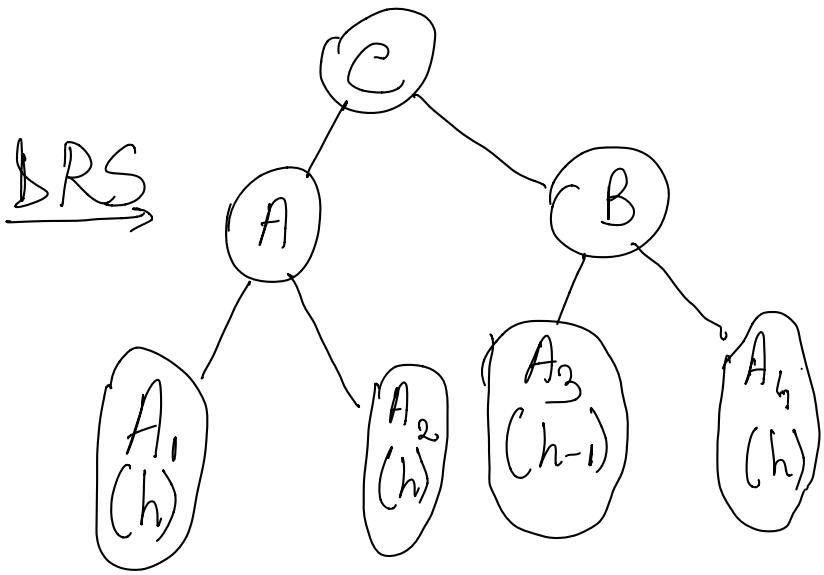
se adaugă



DRS



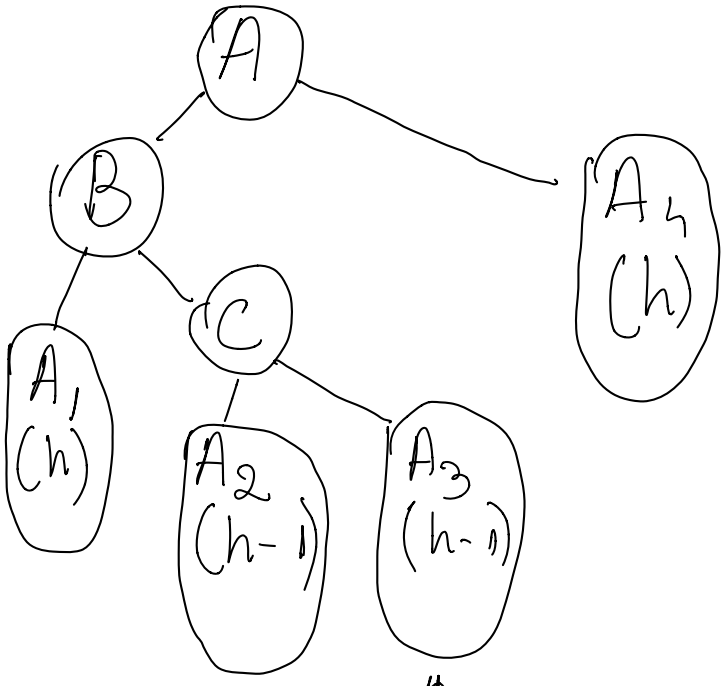
se adaugă



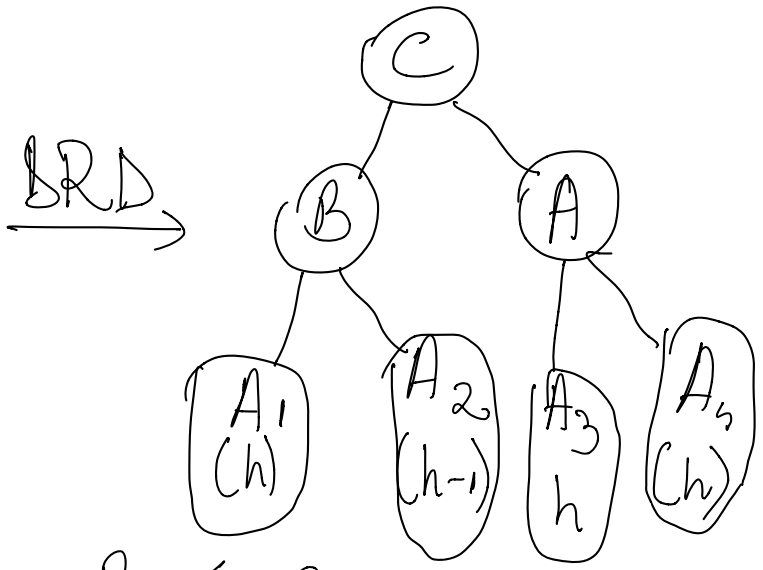
$C < B$

$A < C$

LRD

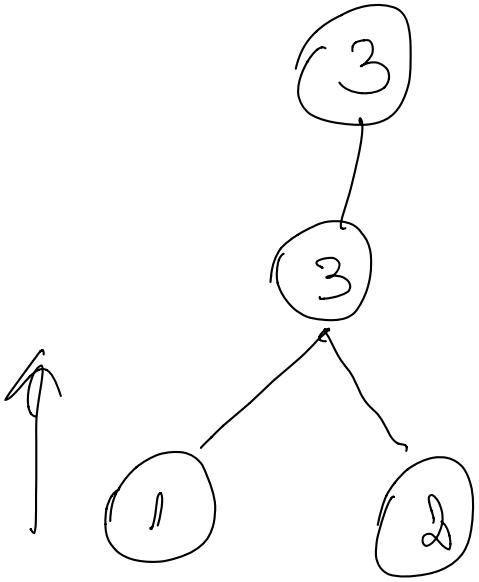


se adauga

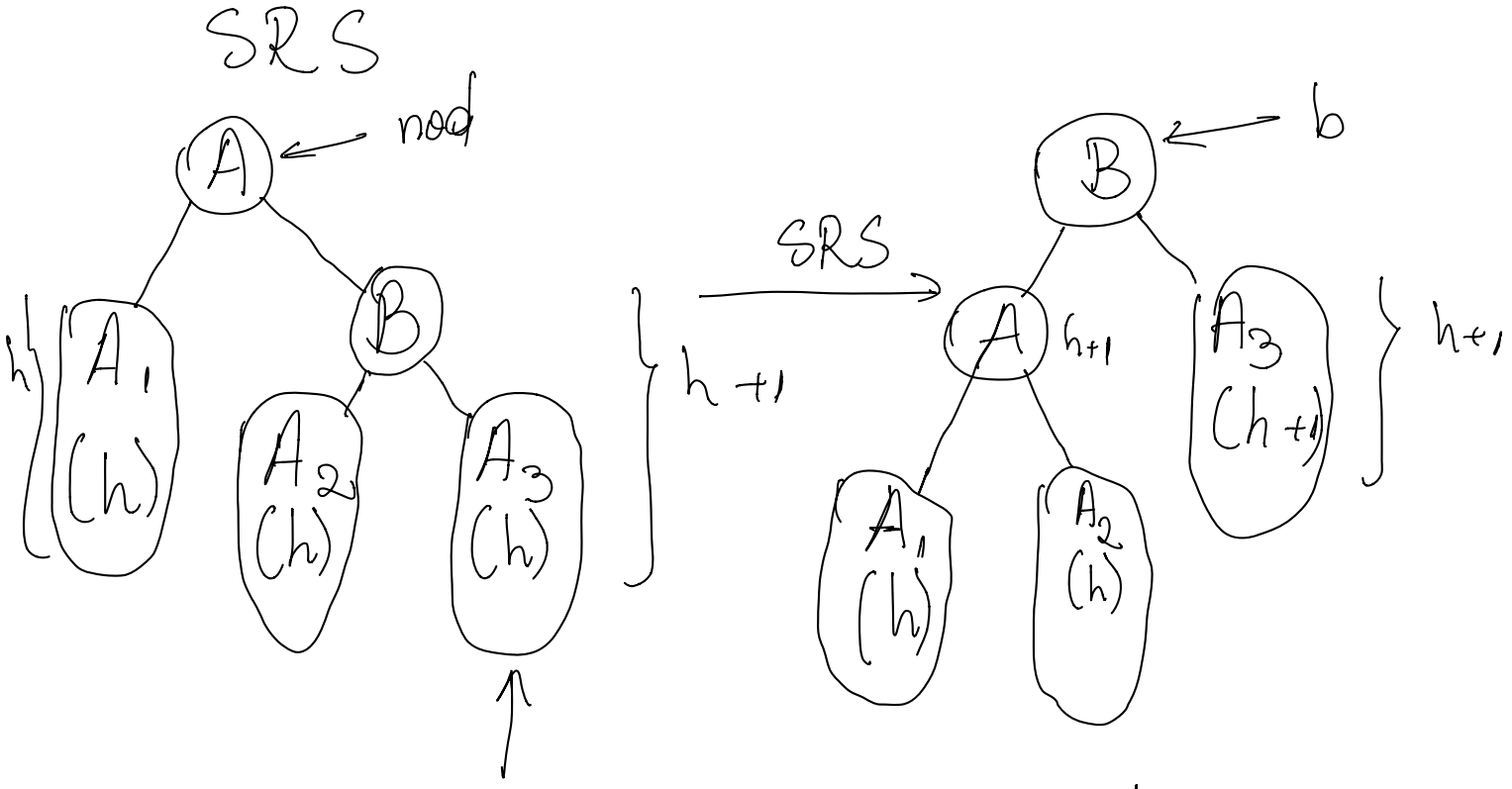


$B < C$

$C < A$



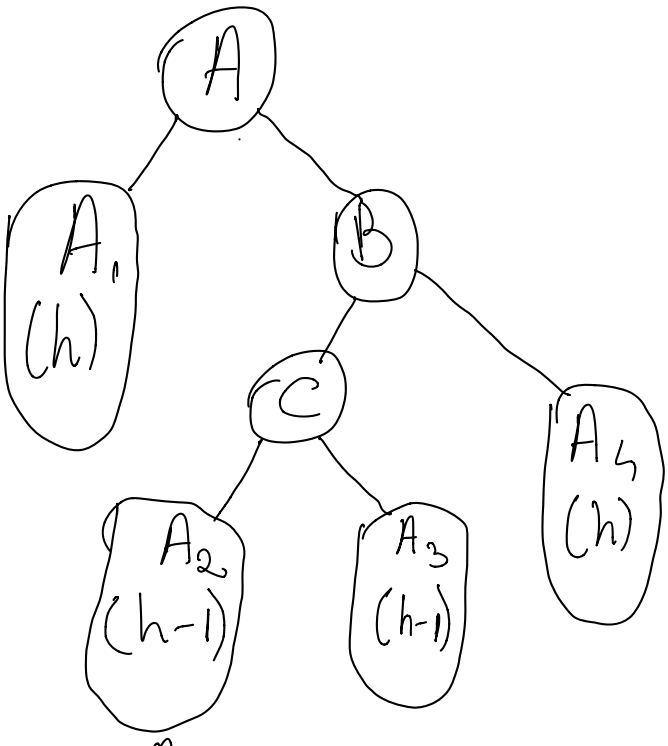
$h = max(st.h, dr.h) + 1$




```
Nod }
    data : TElem
    h    : Integer
    left : ↑Nod
    right: ↑Nod
}
```

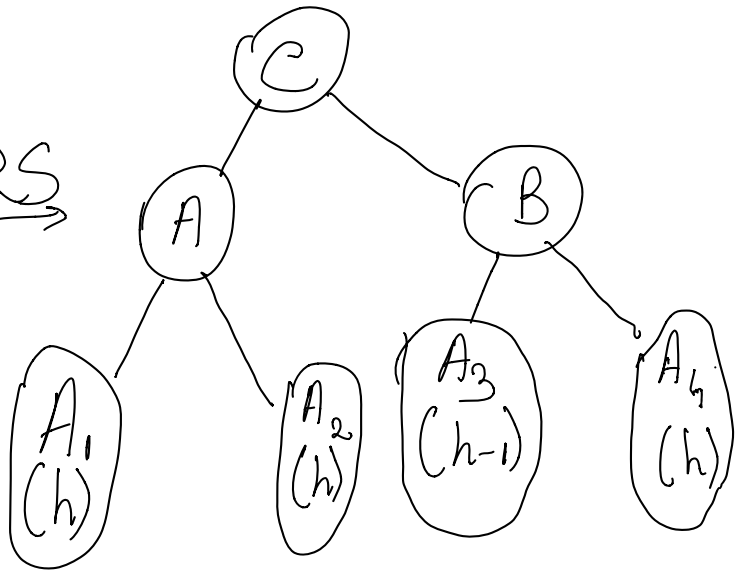
```
SRS(nod) }
    aux = [nod].right
    [aux].left = nod
    //[aux].right = [aux].right
    [nod].right = [nod].right.left
    aux.h = 1 + [aux].left.h
    SRS = aux
}
```

DRS



se adauga

DRS →



C < B

A < C

```
DRS(nod) {
    a3 = [nod].[right].[left].right
    aux = [nod].[right].left
    [aux].left = nod
    [aux].right = [nod].right
    [aux].[right].left = a3
    [aux].[left].right = [nod].[right].[left].left
    DRS = aux
}
```