

# HEAPURI



# Heapuri

## ❖ Definiții

- Graf
- Arbore
- Arbore Binar
- Heap

## ❖ Heapuri - inserare, ștergere

## ❖ Heapify (creare heap în timp liniar)

## ❖ Lazy Deletion

## ❖ Binomial Heap

## ❖ Fibonacci Heap

# Grafuri



Ce este un graf?



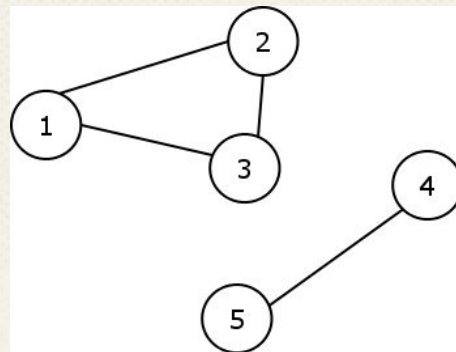
Un graf este o pereche de mulțimi  $G = (V, E)$ , unde:



$V$  este mulțimea de noduri (vertex / vertices),

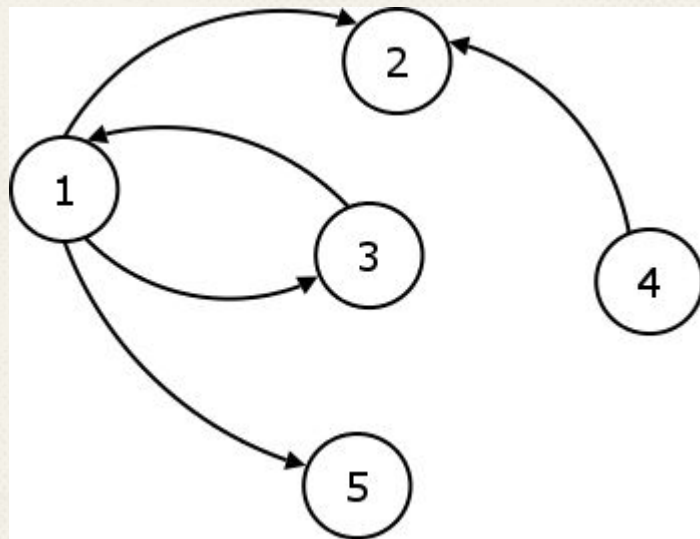
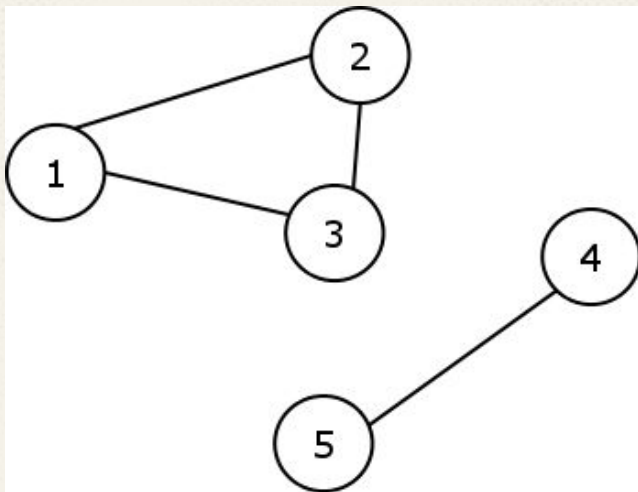


$E$  este mulțimea de muchii



# Grafuri

## ❖ Graf orientat vs graf neorientat



# Arbori



## Definiții:

- Un arbore este un graf conex aciclic
- Un arbore este un graf aciclic maximal
- Un arbore este un graf conex minimal
- Un arbore este un graf aciclic cu  **$n-1$**  muchii
- Un arbore este un graf conex cu  **$n-1$**  muchii
- ...
- Într-un arbore există un singur drum simplu între oricare 2 noduri

# Arbori



## Proprietăți:



Un arbore cu  $n \geq 2$  vârfuri conține minim 2 frunze



Ce este o frunză?



Un nod cu gradul 1 (și rădăcina poate să fie frunză)



# Arbori

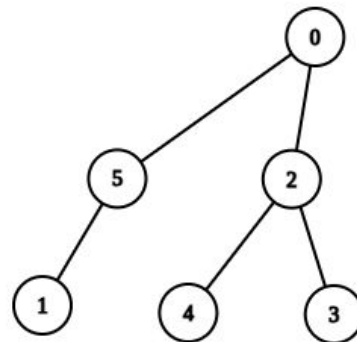
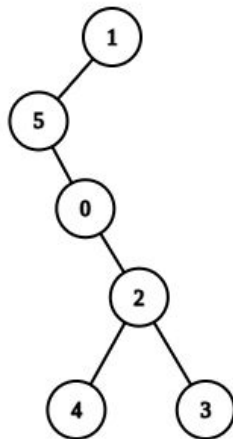
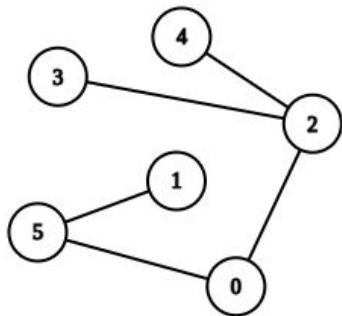


## Rădăcina:



Ce este rădăcina unui arbore?

- Putem alege un nod de care să agățăm arborele; acel nod este rădăcina
- În funcție de ce rădăcina avem, înălțimea arborelui poate fi diferită



# Arbori binari

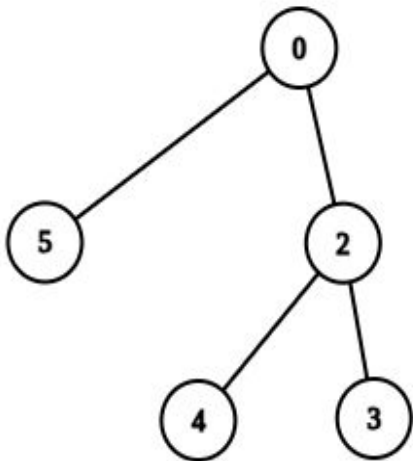
Un **arbore binar** este un arbore cu rădăcină, în care fiecare nod are cel mult 2 copii.

Copiii unui nod sunt numiți copilul stâng (Left, L) și copilul drept (Right, R).

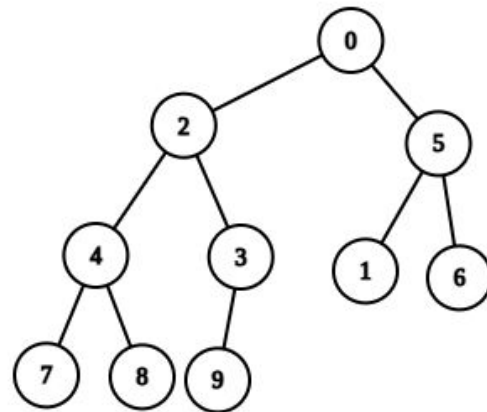


# Arbori binari

Un arbore binar este **plin** dacă fiecare nod are 0 sau 2 fii



Un arbore binar este **complet** dacă toate nivelurile sunt complete, exceptând ultimul nivel care e completat de la S→D



# Arbori binari - Proprietăți



## Exercițiu:



Numărul de noduri ale unui arbore binar cu înălțimea  **$h$**  este între (?) și (?)

- $h$  (dacă este lanț)

- $2^{h+1} - 1$

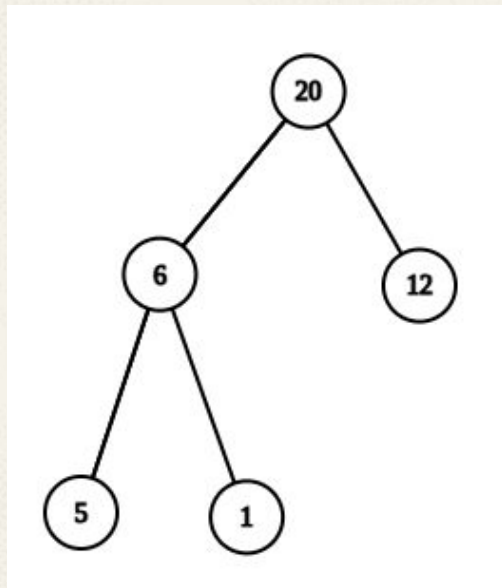
- 1 pe primul nivel, 2 pe al doilea, ...,  $2^h$  pe al  $h$ -lea



Un arbore binar este **balansat** dacă, pentru orice nod, diferența între fiul stâng și cel drept este maxim 1

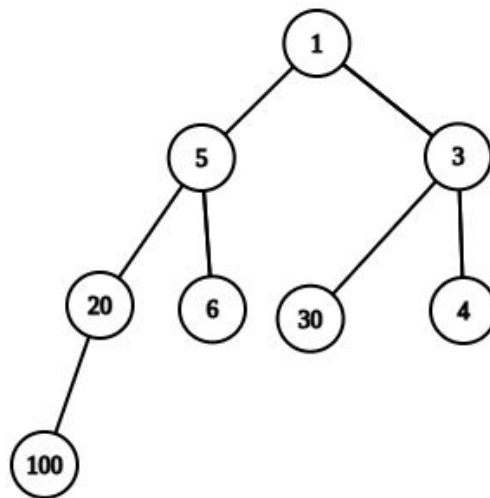
# Heapuri

- ❖ Un heap de maxim este un **arbore binar complet** cu proprietatea că fiecare nod este mai mare decât fii săi

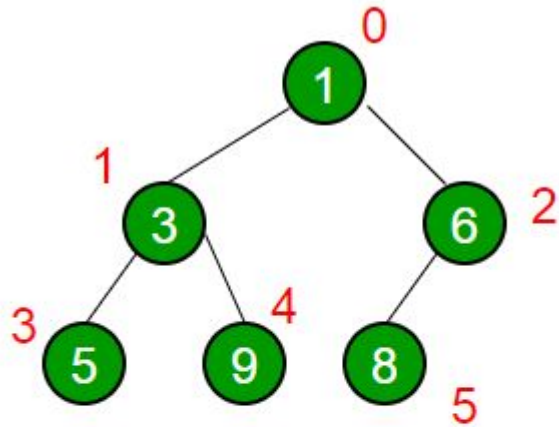


# Heapuri

- ❖ Un heap de minim este un **arbore binar complet** cu proprietatea că fiecare nod este mai mic decât fiii săi
- ❖ Unchiul poate fi mai mare decât nepotul (vezi 5 și 4). Nu există o ordonare pe nivele!! Doar între descendenți!



# Heapuri - Reprezentare

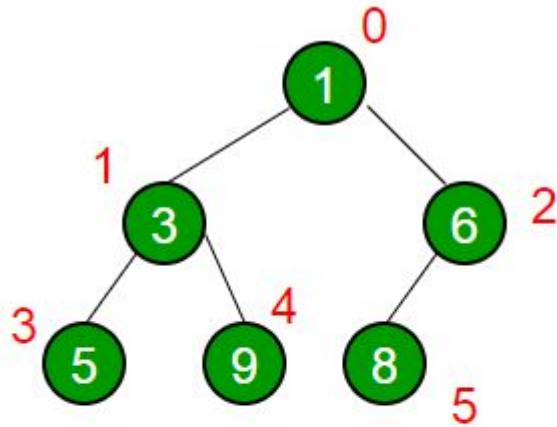


1	3	6	5	9	8
0	1	2	3	4	5

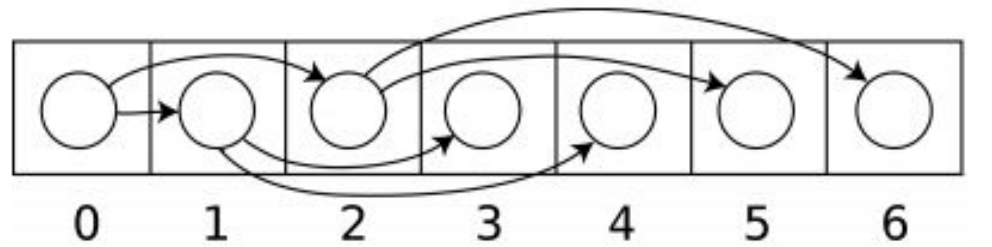
Un arbore binar complet poate fi reprezentat ca un vector!



# Heapuri - Reprezentare



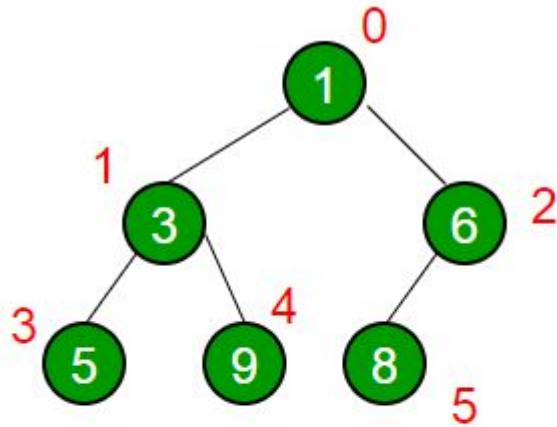
1	3	6	5	9	8
0	1	2	3	4	5



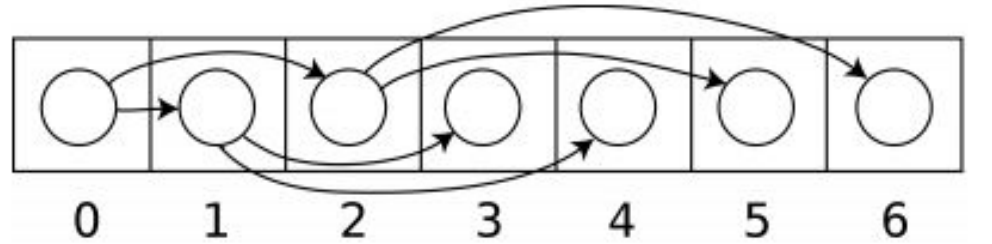
- $\text{Parinte}(i) = (i - 1) / 2$ , unde  $i$  este indicele nodului curent
- $\text{IndexStanga}(i) = 2 * i + 1$ , unde  $i$  este indicele nodului curent
- $\text{IndexDreapta}(i) = 2 * i + 2$ , unde  $i$  este indicele nodului curent



# Heapuri - Reprezentare



1	3	6	5	9	8
0	1	2	3	4	5

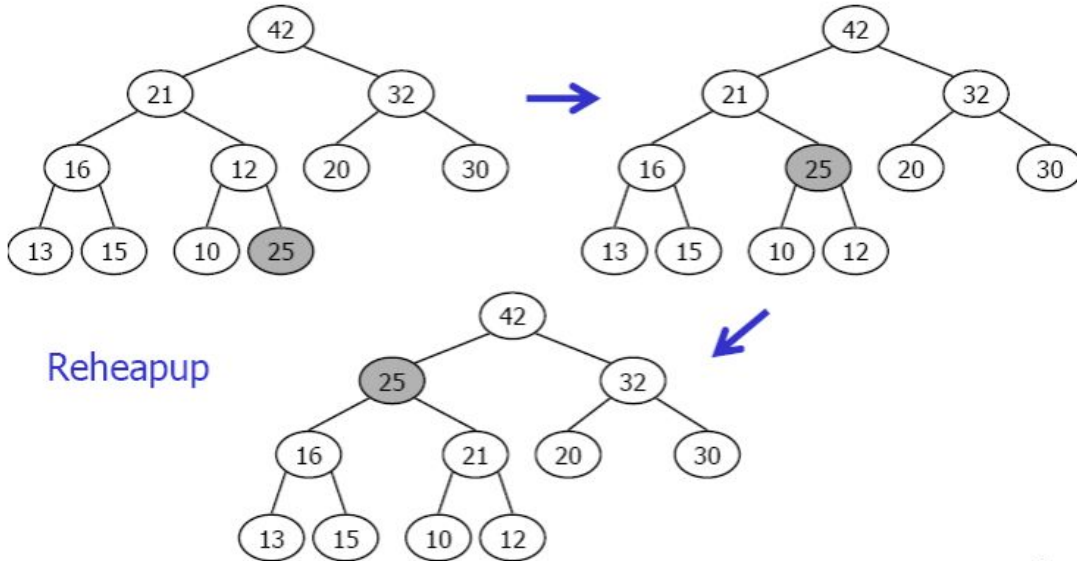


- $\text{Parinte}(i) = (i - 1) / 2$ , unde  $i$  este indicele nodului curent
- $\text{IndexStanga}(i) = 2 * i + 1$ , unde  $i$  este indicele nodului curent
- $\text{IndexDreapta}(i) = 2 * i + 2$ , unde  $i$  este indicele nodului curent

**Înălțime:**  $\log n$  !!

# Heapuri - Urcă (percolate)

**ReheapUp**: repairs a "broken" heap by floating the last element up the tree until it is in its correct location.



**$O(\log n)$**

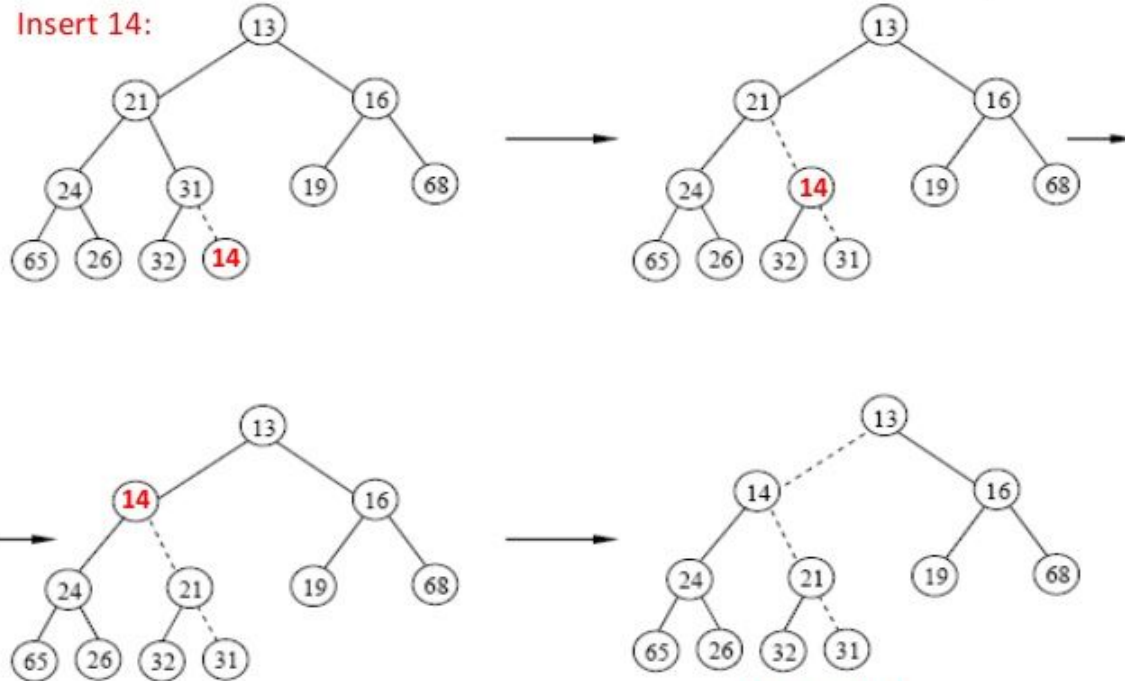
# Urca cod

```
void urca(int poz) {  
    while (poz) {  
        int tata = (poz - 1) / 2;  
        if (heap[tata] < heap[poz]) {  
            swap(heap[tata], heap[poz]);  
            poz = tata;  
        } else {  
            break;  
        }  
    }  
}
```

# Heapuri - Inserare

## Insert new element into min-heap

Insert 14:



The new element is put to the last position, and **ReheapUp** is called for that position.

**$O(\log n)$**

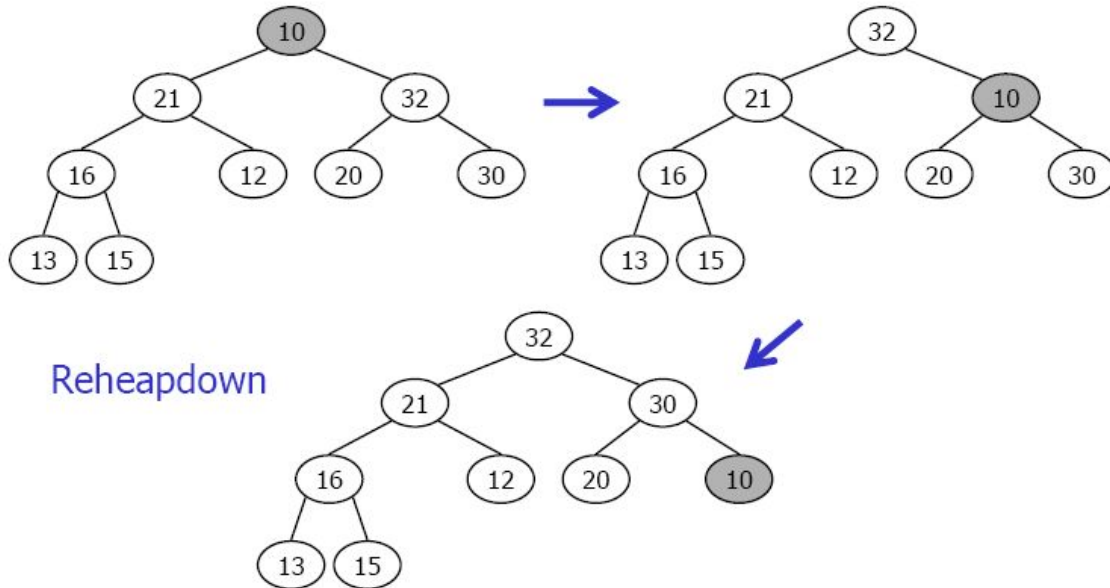
# Inserare cod

```
void push(int x) {  
    heap.push_back(x);  
    urca(heap.size()-1);  
}
```



# Heapuri - Coboară (sift)

**ReheapDown**: repairs a "broken" heap by pushing the root of the subtree down until it is in its correct location.



**$O(\log n)$**



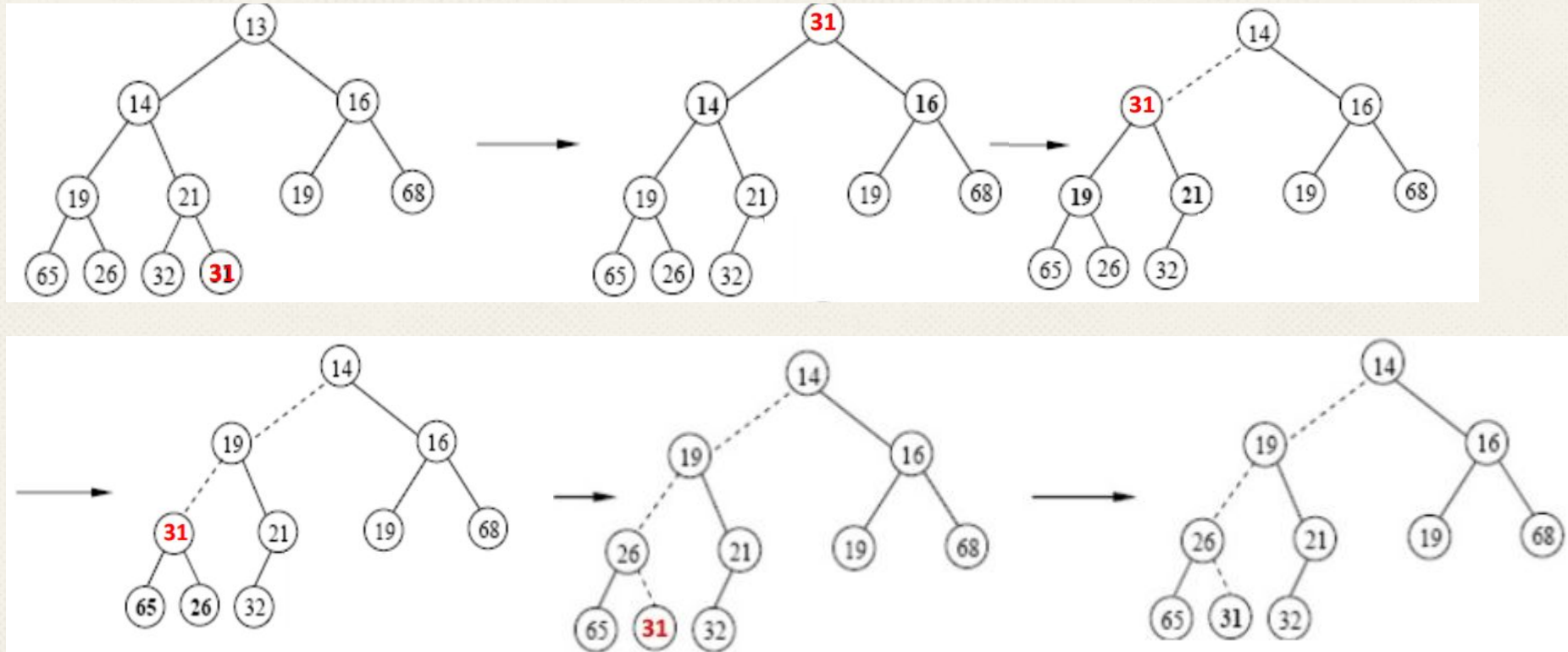
# Coboara cod

```
void coboara(int poz) {  
    if (poz * 2 + 1 >= heap.size())  
        return;  
    int fiu_st = heap[poz * 2 + 1];  
    if ((poz * 2 + 2 == heap.size()) || fiu_st > heap[poz * 2 + 2]) {  
        if (fiu_st > heap[poz]) {  
            swap(heap[poz], heap[poz * 2 + 1]);  
            coboara(poz * 2 + 1);  
        }  
        return;  
    }  
}
```

## Partea 2

```
else { return;
    }
} else {
    if (heap[poz * 2 + 2] > heap[poz]) {
        swap(heap[poz], heap[poz * 2 + 2]);
        coboara(poz * 2 + 2);
        return;
    } else {
        return;
    }
}
```

# Heapuri - Elimină



The element in the last position is put to the position of the root, and **ReheapDown** is called for that position.

# Pop cod

```
int pop() {  
    if (heap.size() == 0)  
        return -1;  
    int vf = heap[0];  
    heap[0] = heap[heap.size()-1];  
    heap.pop_back();  
    coboara(0);  
    return 0;  
}
```

# Heapify

Construire heap



Inserăm  $n$  elemente -  $O(n \log n)$



# Heapify

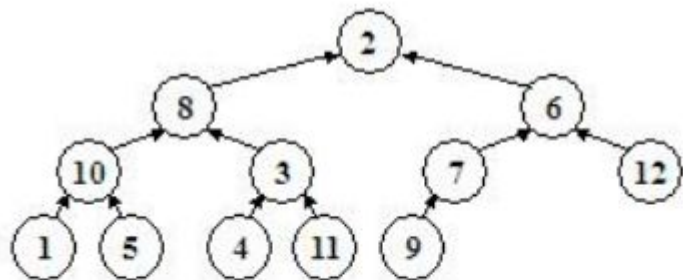
## Construire heap

- ❖ Inserăm  $n$  elemente -  $O(n \log n)$
- ❖ Liniar:
  - Coborâm fiecare element începând de jos în sus

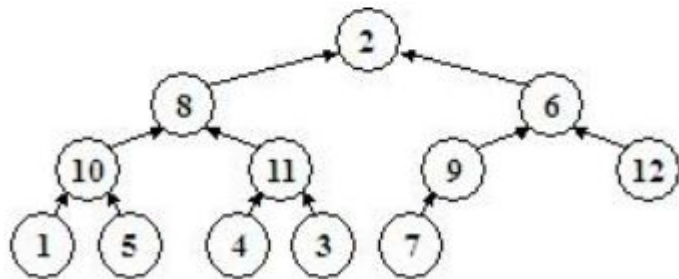
```
void build_heap(Heap H) {  
    for (int i = H.size() / 2; i >= 0; i--) {  
        cobora(H, i);  
    }  
}
```



# Heapify

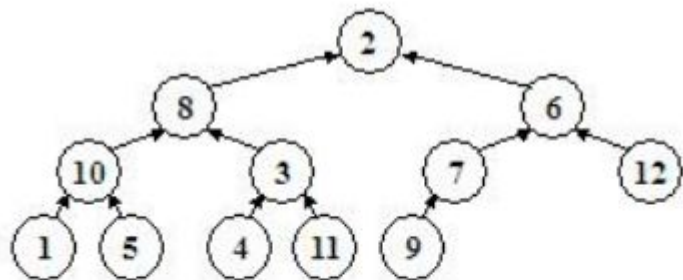


*Nivelul frunzelor este organizat*

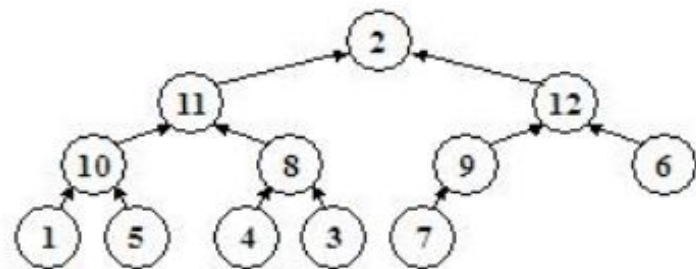


*Ultimele doua niveluri sunt organizate*

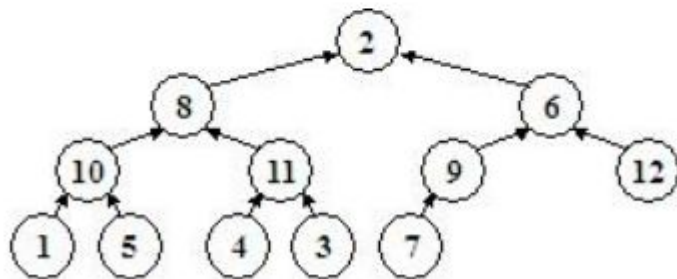
# Heapify



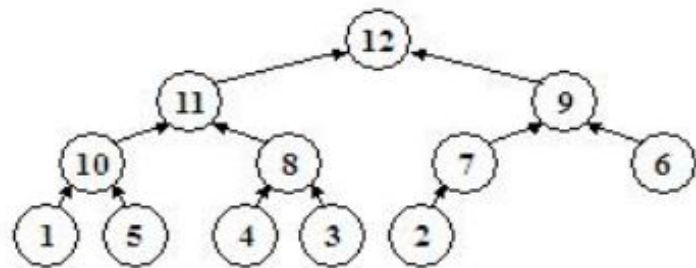
*Nivelul frunzelor este organizat*



*Ultimele trei niveluri sunt organizate*



*Ultimele doua niveluri sunt organizate*



*Structura de heap*

# Heapify

Complexitate

❖  $n$  noduri: coborâm fiecare nod în  **$\log n$**

→  **$O(n \log n)$**

# Heapify

## Complexitate

❖ n noduri: coborâm fiecare nod în  **$\log n$**

→  **$O(n \log n)$**

❖ Sau calculăm pentru fiecare nod ce efort depunem

➤ Pentru jumătate nu facem nimic (cazul frunzelor)

➤ Pentru un sfert, coboară maxim un nivel

➤ Ş.a.m.d.

$$\begin{aligned}\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) &= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\end{aligned}$$

# Problemă

❖ Se dau multe operații de genul:

➤ Inserare număr -  $O(\log n)$

➤ Afișare minim -  $O(1)$

➤ Elimină indice

Cum putem folosi un heap?

❖ **Problemă: ?**



# Problemă

❖ Se dau multe operații de genul:

➤ Inserare număr -  $O(\log n)$

➤ Afișare minim -  $O(1)$

➤ Elimină indice

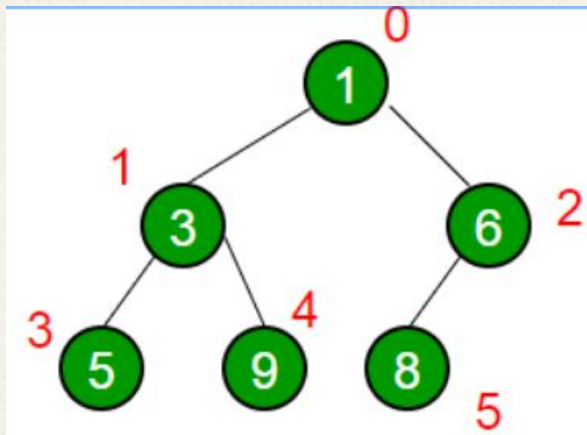
Cum putem folosi un heap?

❖ **Problemă:** Eliminare număr (Nu știm indexul din heap și fără să știm indexul nu putem elimina în  **$\log n$** )

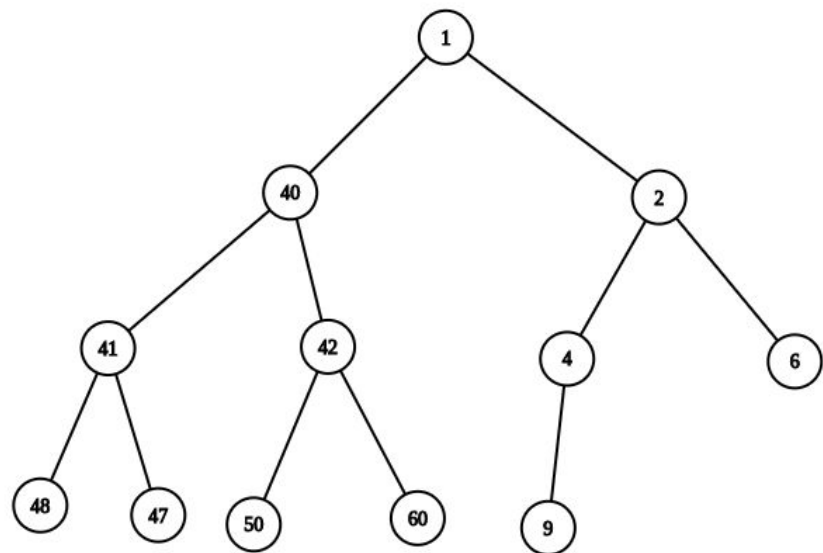


# Eliminare element cunoscând poziția

```
elimina(i) {  
    heap[i] = heap[n--];  
    coboara(i);  
    urca(i);  
}
```



Eliminăm 3, respectiv 41



# Problemă

❖ Se dau multe operații de genul:

➤ Inserare număr

➤ Afișare minim

➤ Elimină indice

Cum putem folosi un heap?

❖ **Problemă:** Eliminare indice

➤ Totuși, nu avem poziția în heap. Putem să o reținem (niște pointeri dubli... un pic dureros).

# Lazy deletion

- ❖ Marcăm un nod spre ștergere, dar nu-l ștergem decât când ajunge în vârf
  - Mai simplu
  - Trebuie să folosim mai multă memorie ca să ținem minte elementele marcate
  - Căutarea în heap e  $O(n)$
- ❖ Operație ce va fi folosită în general la arbori, nu doar pentru heapuri

# Heapuri - Complexitate

Operație	Timp Mediu	Cel mai rău caz
Spațiu	$O(n)$	$O(n)$
Căutare	$O(n)$	$O(n)$
Inserare	$O(1)$ $n/2 * 0 + n/4 * 1 + n/8 * 2 \dots \approx 1$	$O(\log n)$
Ștergere	$O(\log n)$	$O(\log n)$
Căutare minim	$O(1)$	$O(1)$
Construcție n elemente	$O(n)$	$O(n)$
Uniune (2 heapuri de n elemente)	$O(n)$	$O(n)$

# Heapuri Binomiale și Heapuri Fibonacci



Motivație:

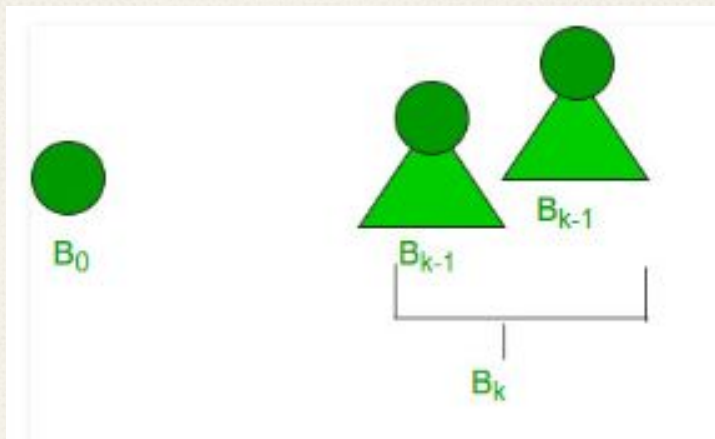
➤ Reuniunea este înceată și alte operații pot fi îmbunătățite

	Căutare Min	Ștergere Min	Inserare	Update	Reuniune
Heap Binar	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Heap Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$ (amortizat)	$\Theta(\log n)$	$O(\log n)$
Heap Fibonacci	$\Theta(1)$	$O(\log n)$ (amortizat)	$\Theta(1)$	$\Theta(1)$ (amortizat)	$\Theta(1)$



# Arbori binomiali

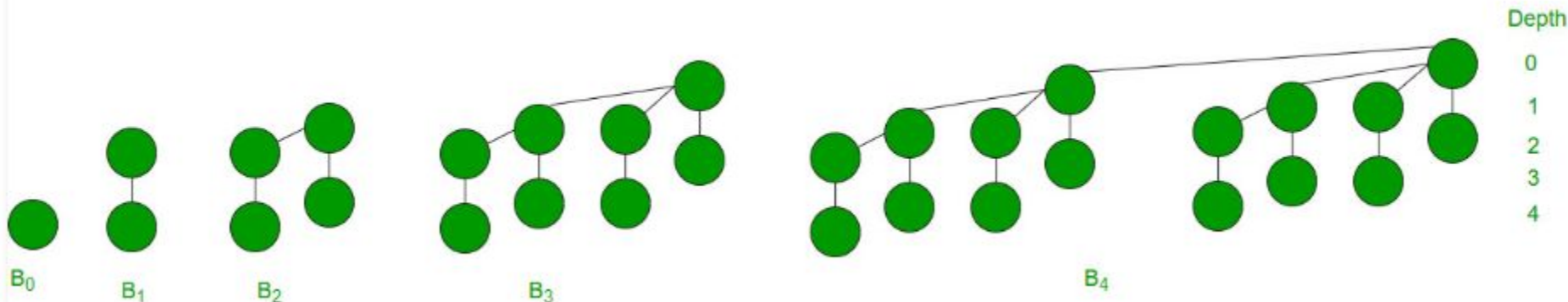
- ❖ Un arbore binomial de ordin 0 are un nod (rădăcina)
- ❖ Un arbore binomial de ordin  $K$  poate fi format prin reuniunea a doi arbori binomiali de mărime  $K-1$ , făcând pe unul dintre ei fiul stâng al celuilalt



# Arbori binomiali

Proprietăți ale unui arbore binomial de ordin  $k$ :

- ❖ Are exact  $2^k$  noduri
- ❖ Are înălțimea  $k$
- ❖ Sunt exact  $C_i^k$  (combinări de  $i$  luate câte  $k$ ) noduri de înălțime  $i$  pentru  $i = 0, 1, \dots, k$
- ❖ Rădăcina are gradul  $k$  și copiii săi sunt arbori binomiali de tip  $k-1, k-2, \dots, 0$

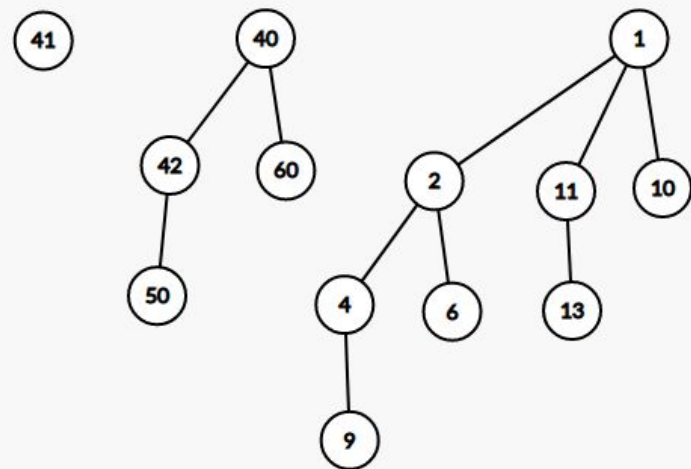


# Heapuri Binomiale

- ❖ Un Heap Binomial este o colecție de Arbori Binomiali, fiecare dintre ei având proprietatea de heap minim.
- ❖ Observație: Există o singură structură de heap binomial pentru orice mărime.
- ❖ Exemplu:
  - Cum arată un heap binomial cu 13 noduri?
  - Câți arbori binomiali are?
  - Ce tipuri?

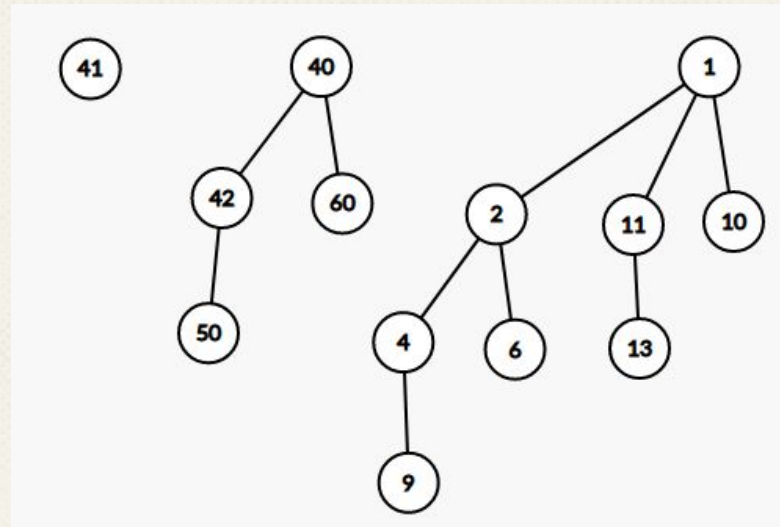
# Heapuri Binomiale

- ❖ Un Heap Binomial este o colecție de Arbori Binomiali, fiecare dintre ei având proprietatea de heap minim.
- ❖ Observație: Există o singură structură de heap binomial pentru orice mărime.
- ❖ Exemplu:
  - Cum arată un heap binomial cu 13 noduri?
  - Câți arbori binomiali are?
  - Ce tipuri?



# Heapuri Binomiale - Căutare minim

- 1) Minimul se află în rădăcina unui arbore binomial. Putem parcurge toți arborii binomiali, să ne uităm la rădăcina lor și să reținem minimul  
→  **$O(\log n)$**
- 2) Totuși, putem ține minte valoarea când facem orice fel de operație și să răspundem în  **$O(1)$**





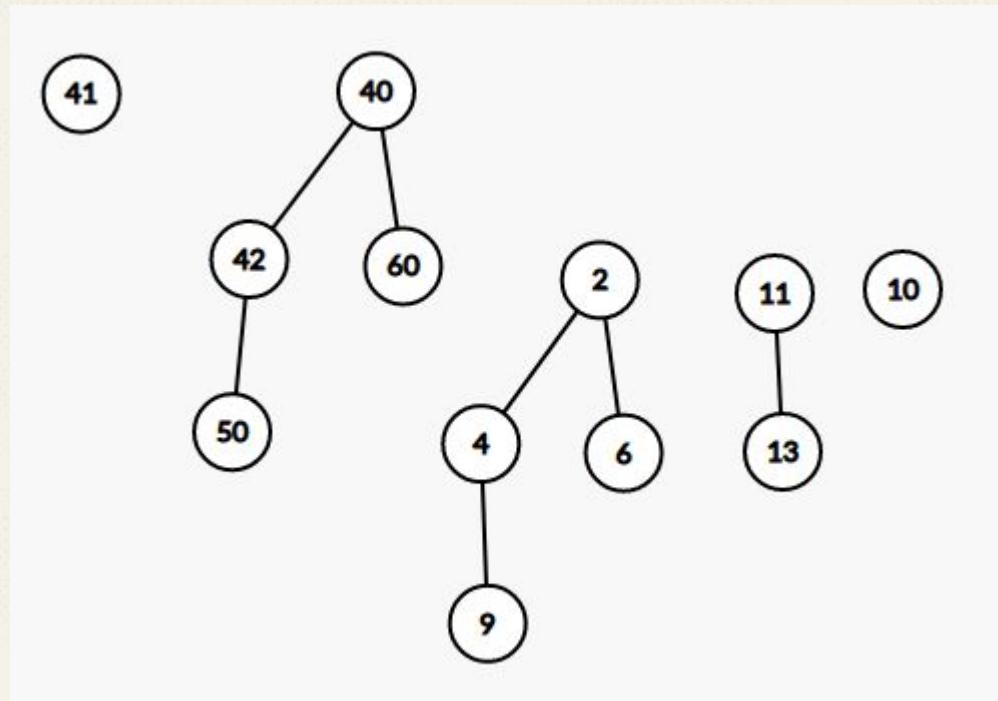
# Heapuri Binomiale - Extragerea minimului



Eliminăm minimul

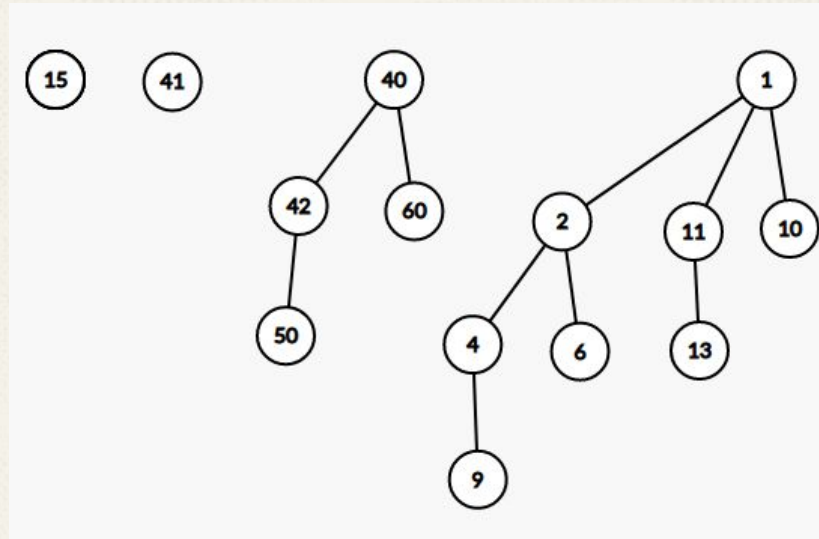


Apoi facem reuniune

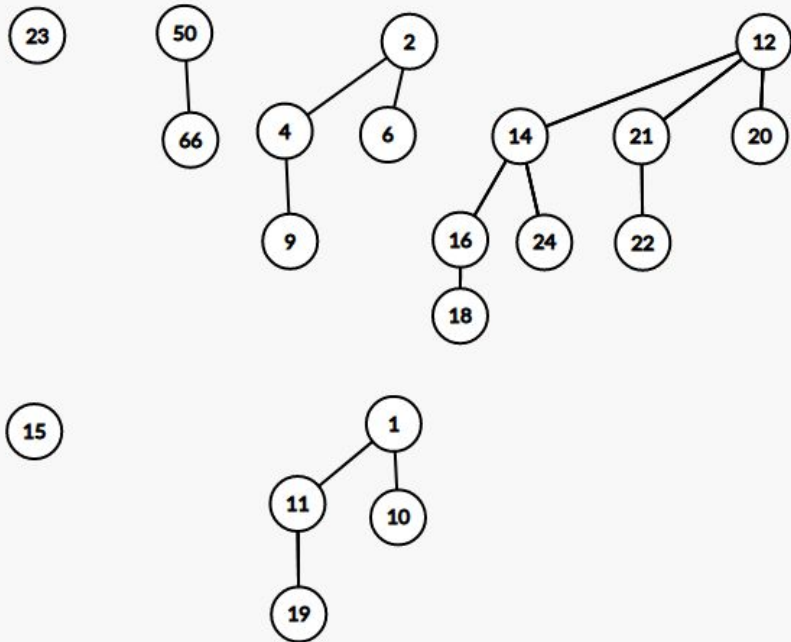


# Heapuri Binomiale - Inserare

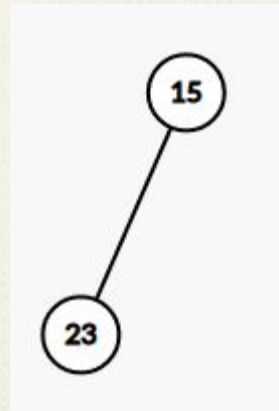
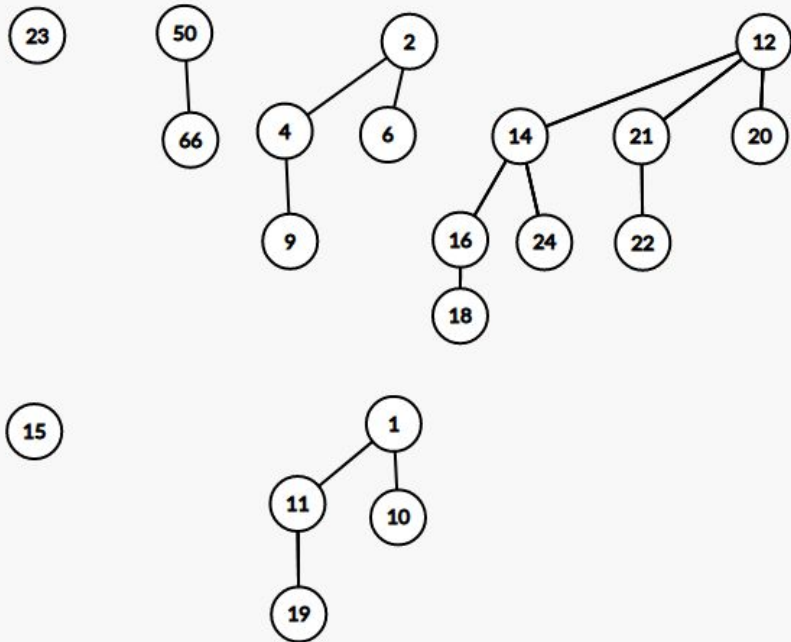
Adăugăm un arbore binomial de mărime 1, apoi apelăm reuniunea.



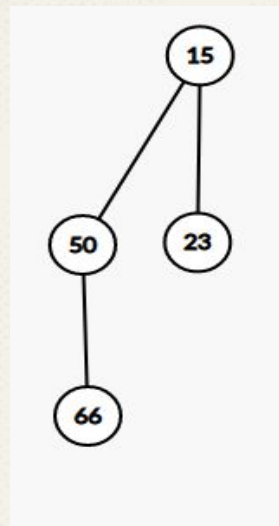
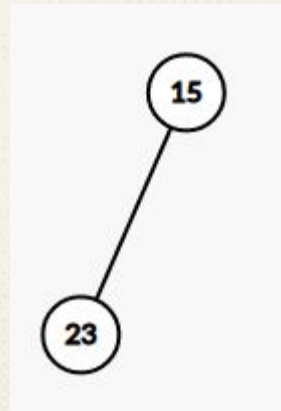
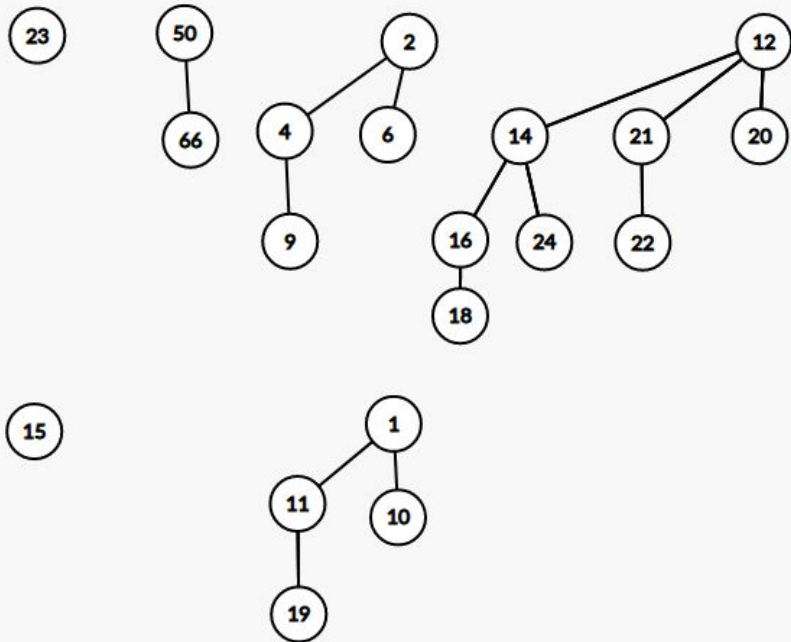
# Reuniune!



# Reuniune!

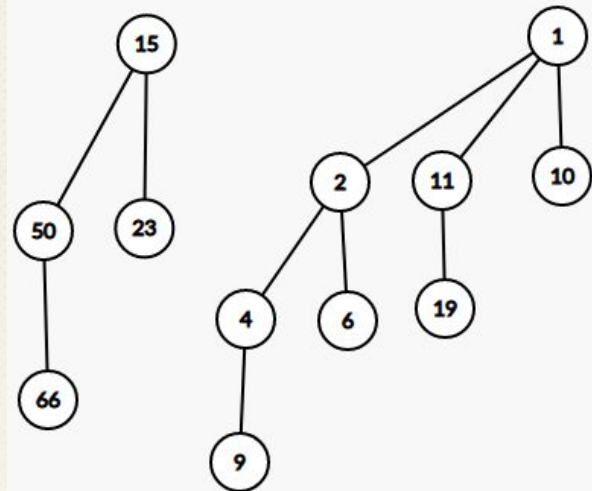
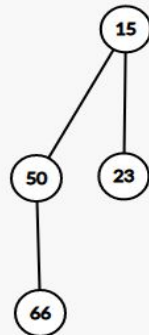
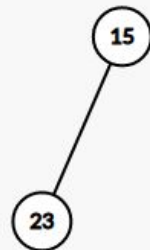
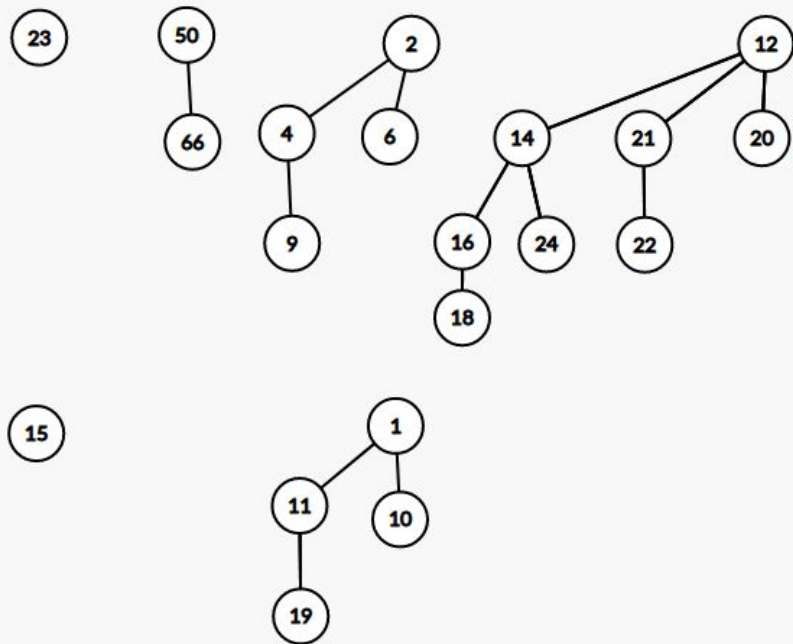


# Reuniune!

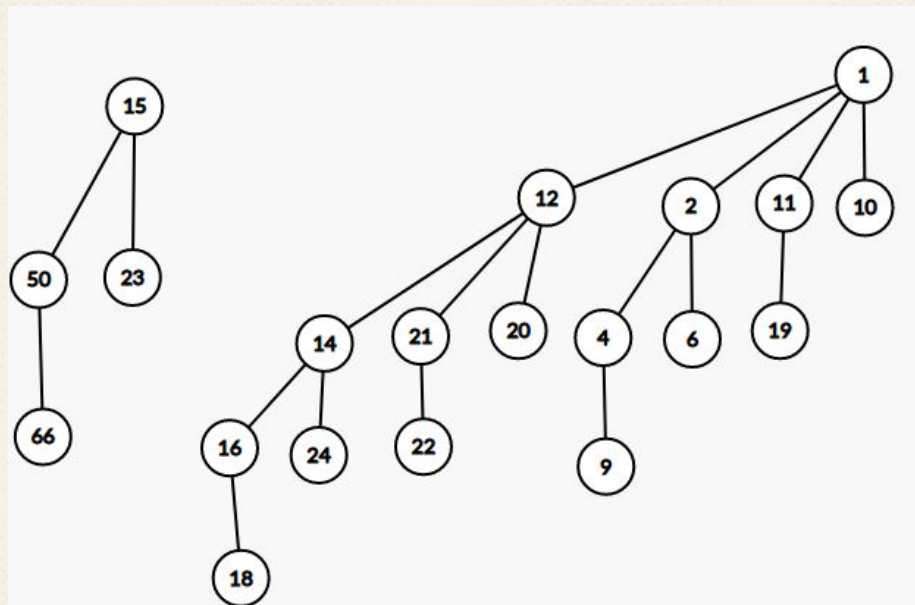
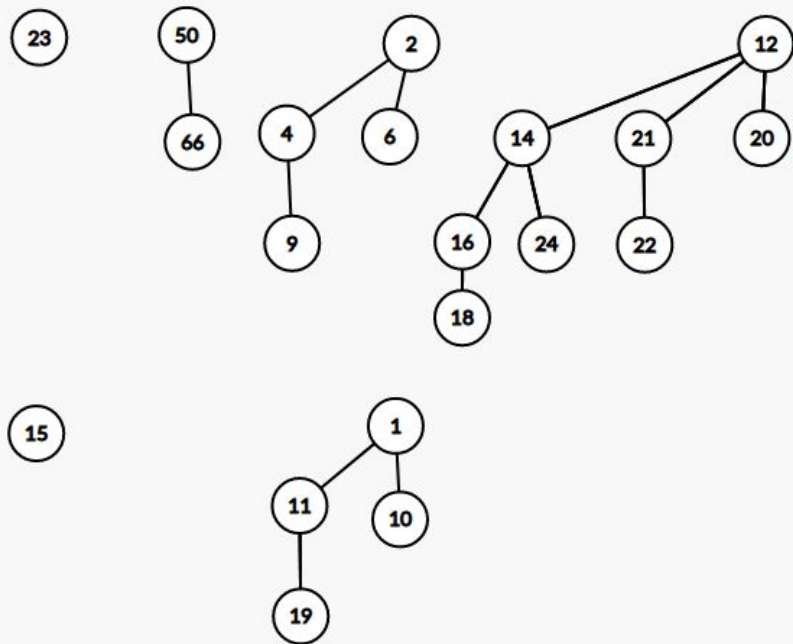




# Reuniune!



# Reuniune!



# Reuniune!

Complexitate:  $O(\log n)$

Pentru fiecare mărime a arborilor binomiali de la **0** la  **$\log n$**  trebuie “eventual” să fac o reuniune a doi arbori.

Reuniunea a doi arbori se face în  **$O(1)$** .

# Heap-uri binomiale și Fibonacci

- Motivație:

- Reuniunea este înceată și alte operații pot fi îmbunătățite.

	Căutare Min	Ștergere Min	Inserare	Update	Reuniune
Heap Binar	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Heap Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$ (amortizat)	$\Theta(\log n)$	$O(\log n)$
Heap Fibonacci	$\Theta(1)$	$O(\log n)$ (amortizat)	$\Theta(1)$	$\Theta(1)$ (amortizat)	$\Theta(1)$

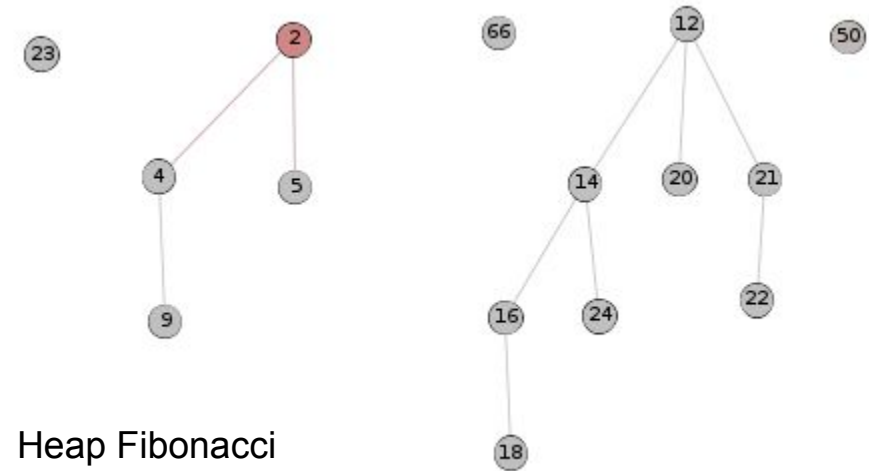
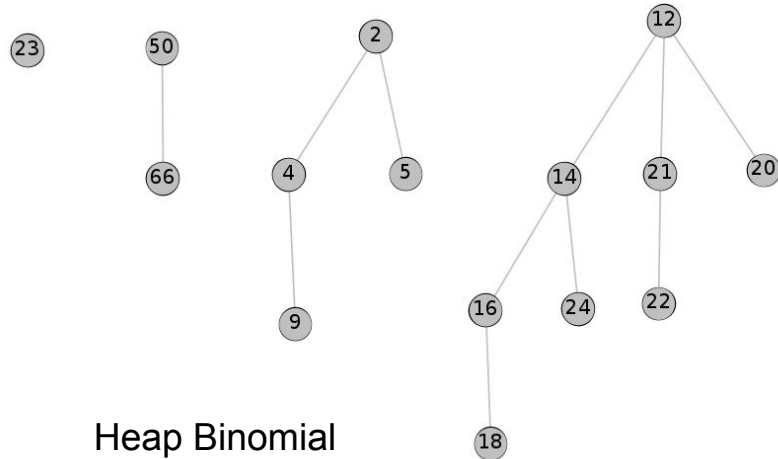
# Heapuri Fibonacci

- **Heapurile Fibonacci** sunt o colecție de arbori care au proprietatea de ordonare de heap (arborii nu trebuie să fie binomiali).
- Arborii dintr-un heap Fibonacci nu sunt ordonați.
- Arborii din componență au mărimi puteri ale lui 2. Fiii vor fi arbori de mărime  $1, \dots, k-1$ , dar nu neapărat sortați de la stânga la dreapta.



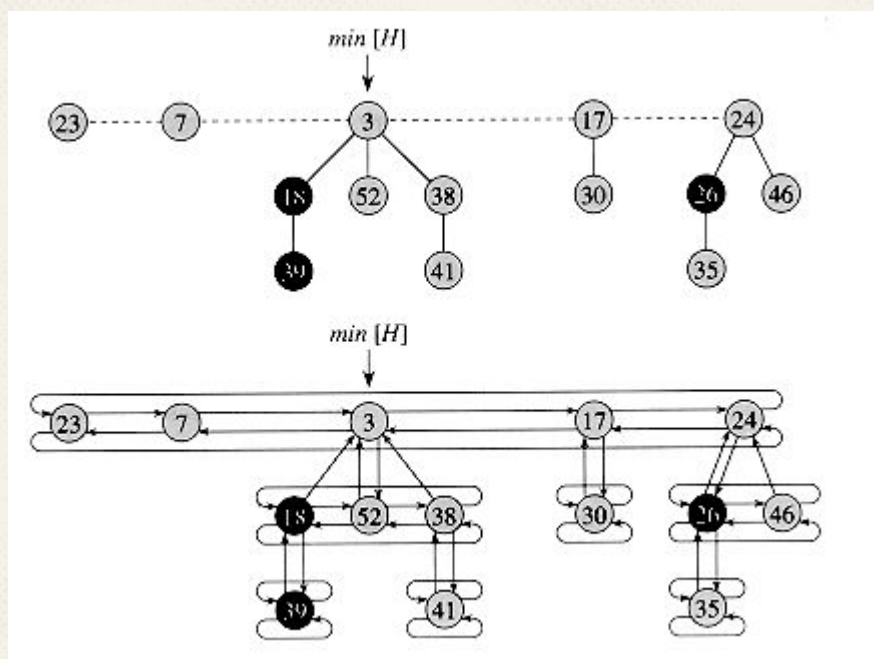
# Heapuri Fibonacci

- **Heapurile Fibonacci** sunt o colecție de arbori care au proprietatea de ordonare de heap (arborii nu trebuie să fie binomiali).
- Arborii dintr-un heap Fibonacci nu sunt ordonați.
- Arborii din componentă au mărimi puteri ale lui 2. Fiii vor fi arbori de mărime  $1, \dots, k-1$ , dar nu neapărat sortați de la stânga la dreapta.



# Implementare

- Listă dublu înlănțuită între rădăcini
- Link către un fiu
- Listă dublu înlănțuită între frați
- Link către tată

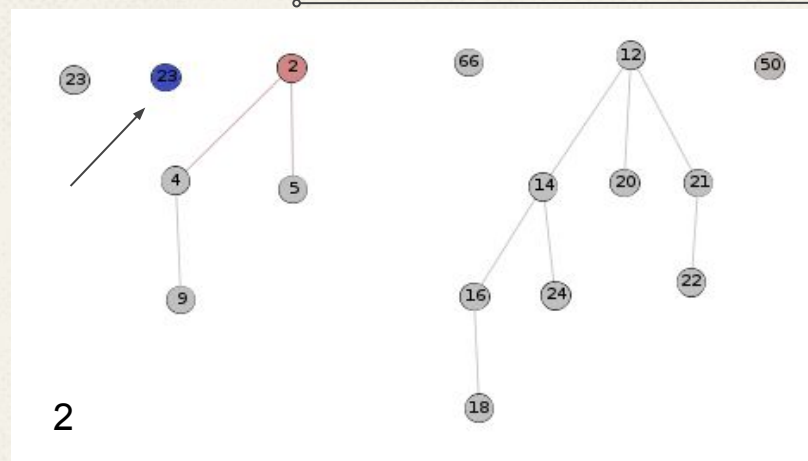
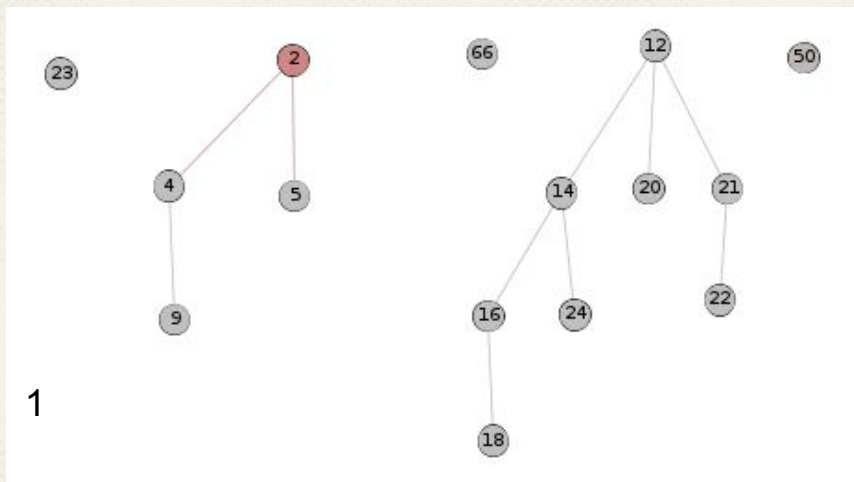


# Inserare nod

- Creăm un arbore cu un singur element
- Îl plasăm în stânga rădăcinii.
- **Nu facem reuniune!**

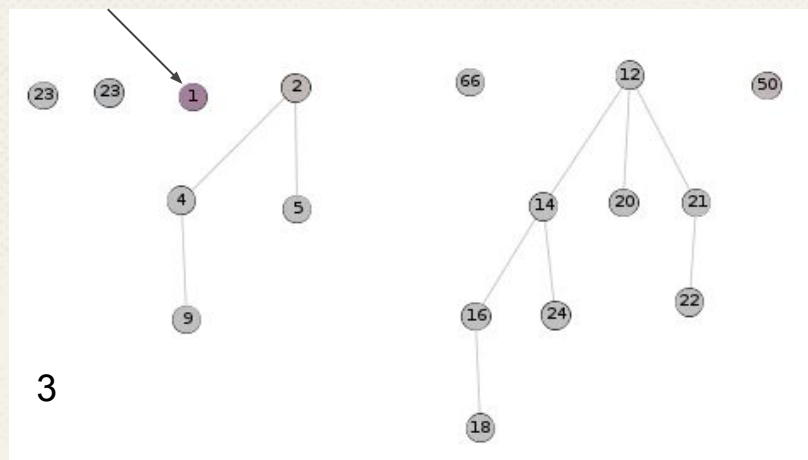
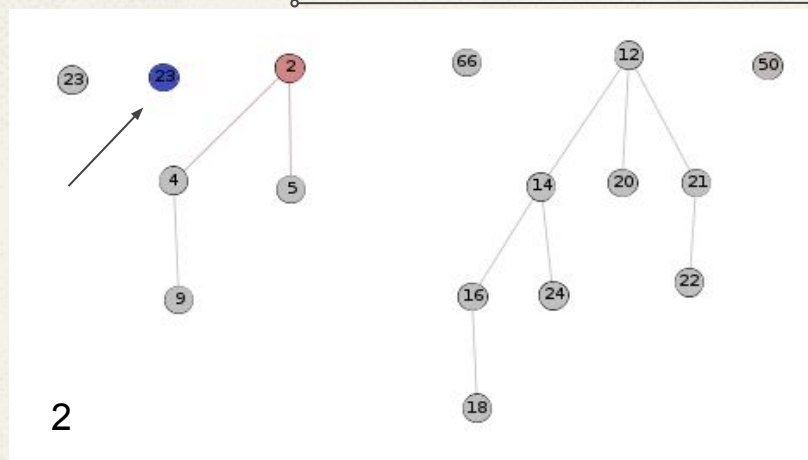
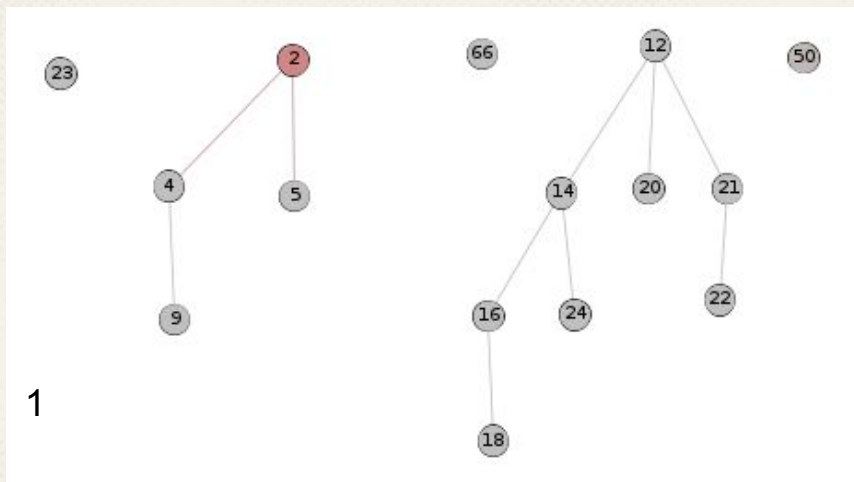
# Inserare nod

- Creăm un arbore cu un singur element
- Îl plasăm în stânga rădăcinii.
- **Nu facem reuniune!**



# Inserare nod

- Creăm un arbore cu un singur element
- Îl plasăm în stânga rădăcinii.
- Nu facem reuniune!  $\rightarrow O(1)$





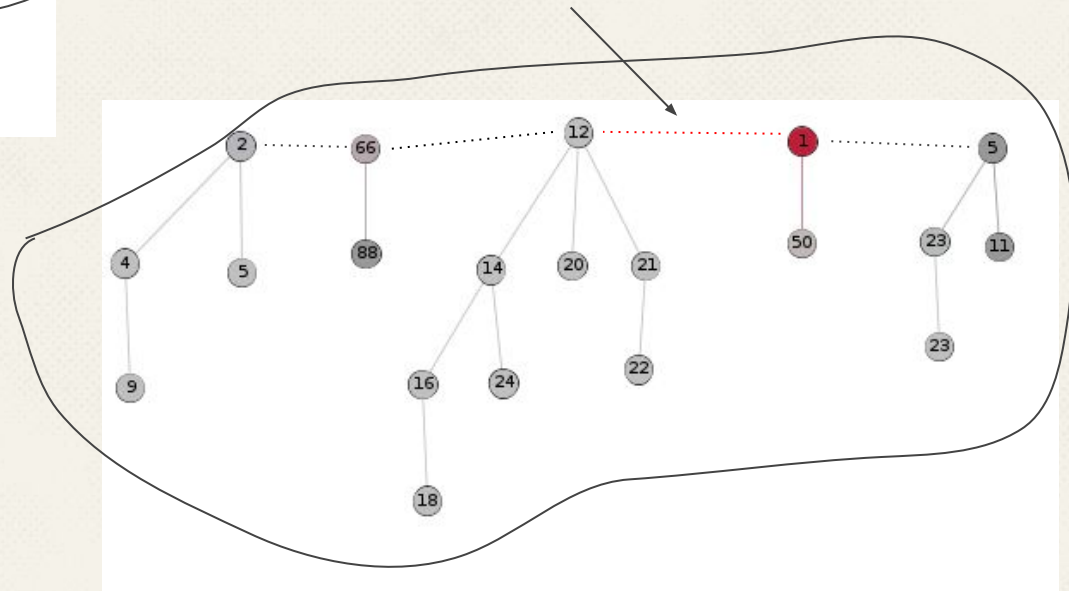
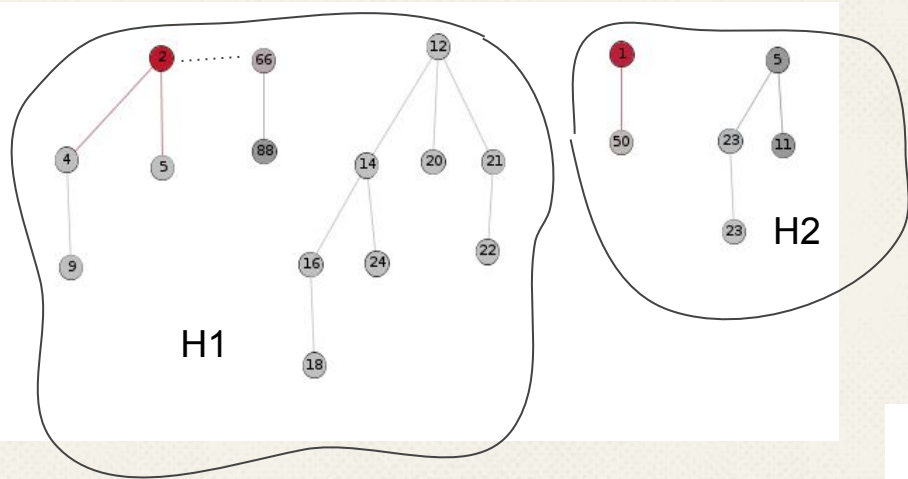
# Caută Minimul

- La fiecare pas ținem pointer spre minim.
- **Complexitate  $O(1)$ !**

# Reuniune

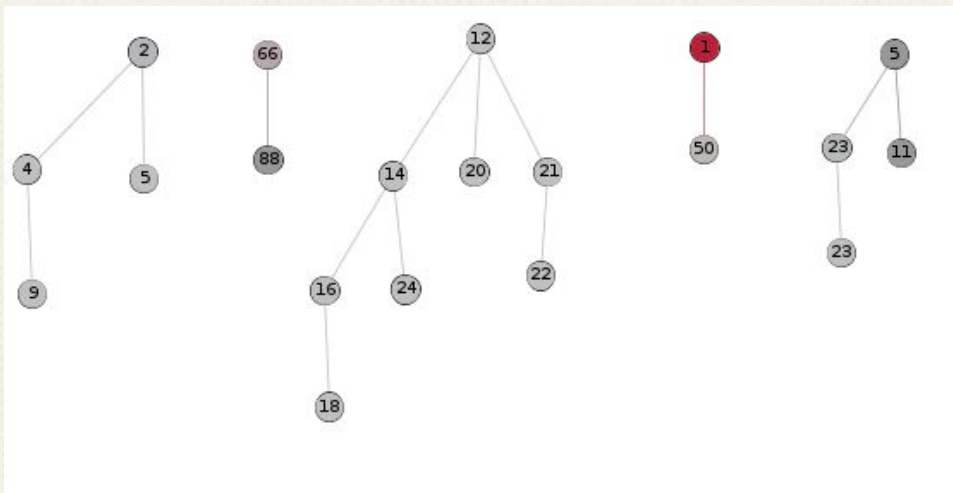
- Concatenăm rădăcinile lui  $H_2$  la cele ale lui  $H_1$ .
- Avem grijă să păstrăm lista dublu înlănțuită.
- Avem grijă să păstrăm minimul (poate fi unul din cei 2 minimi)
- Nu facem consolidare (putem să avem mai mulți arbori de aceeași mărime).
- **Complexitate  $O(1)!!$**

# Reuniune



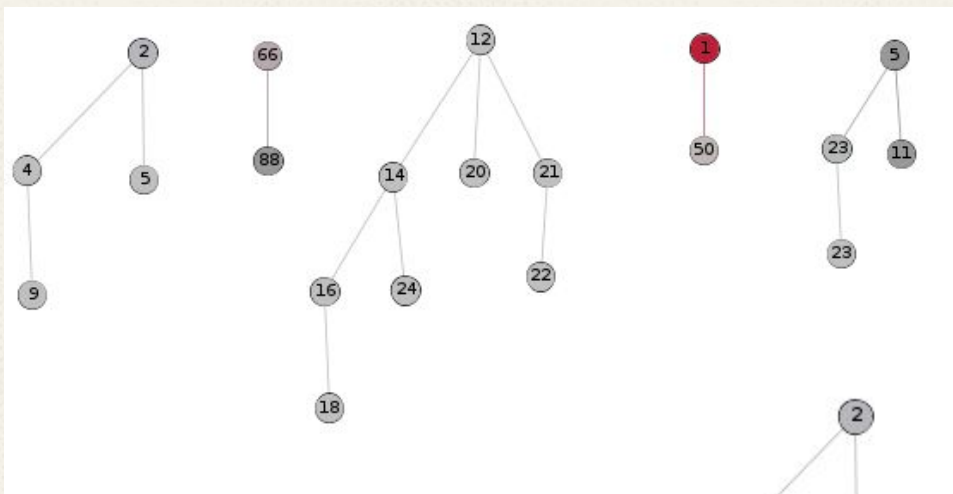
$O(1)$

# Extragere minim



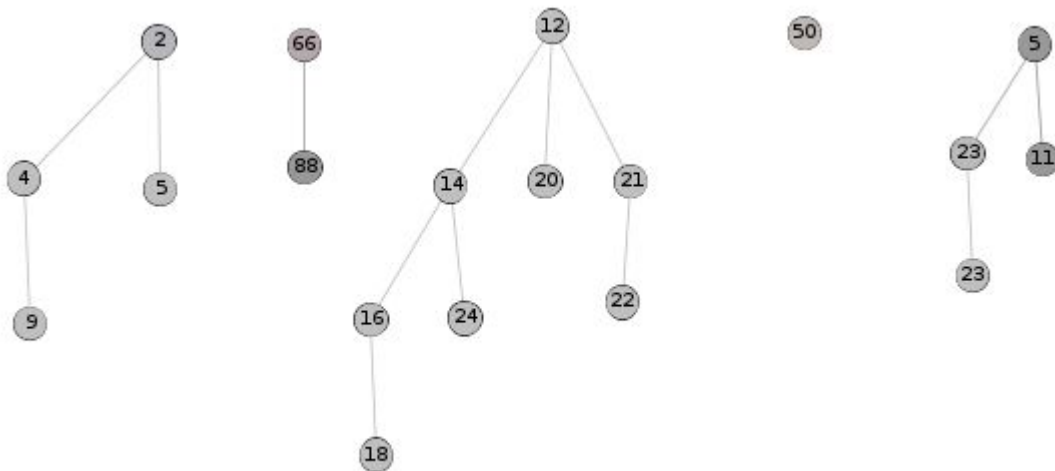
Extragem minim. Fiii săi devin arbori liberi.

# Extragere minim



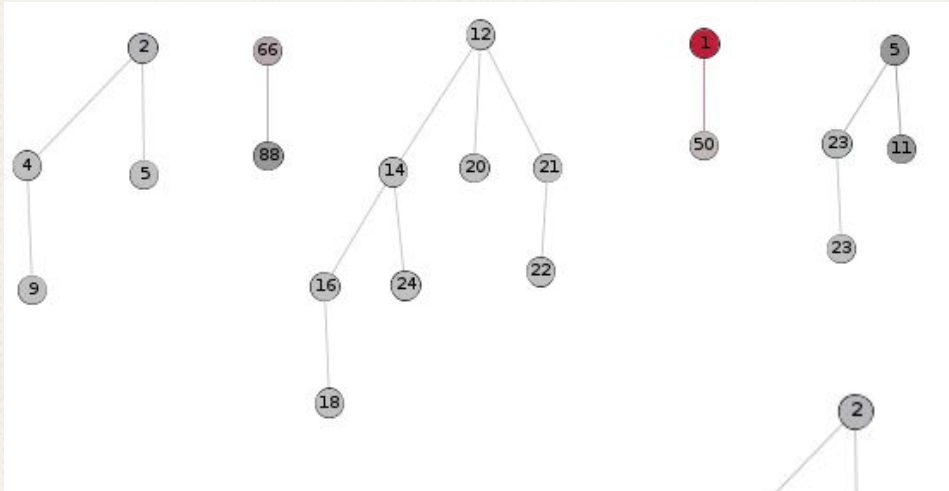
Extragem minim. Fiii săi devin arbori liberi.

Unde e problema?





# Extragere minim



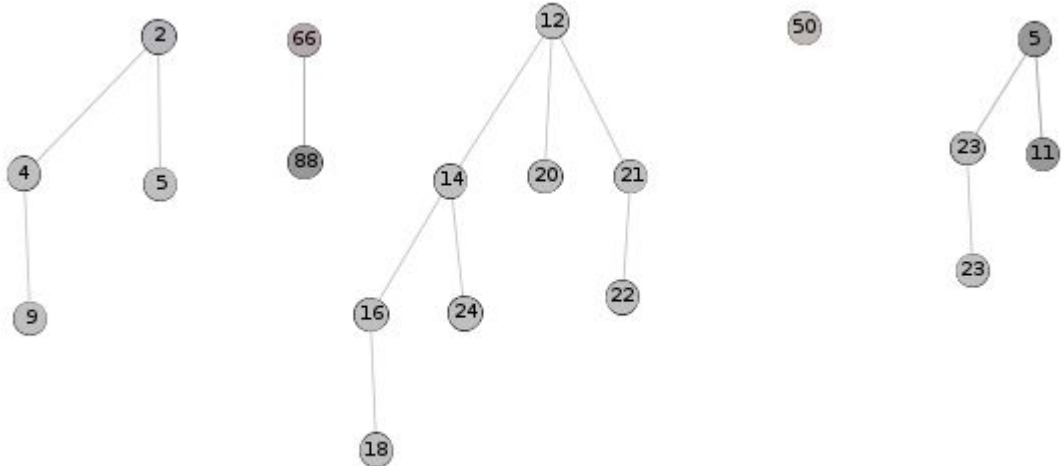
Extragem minim. Fiii săi devin arbori liberi.

## Unde e problema?

Nu știm care e minimul.

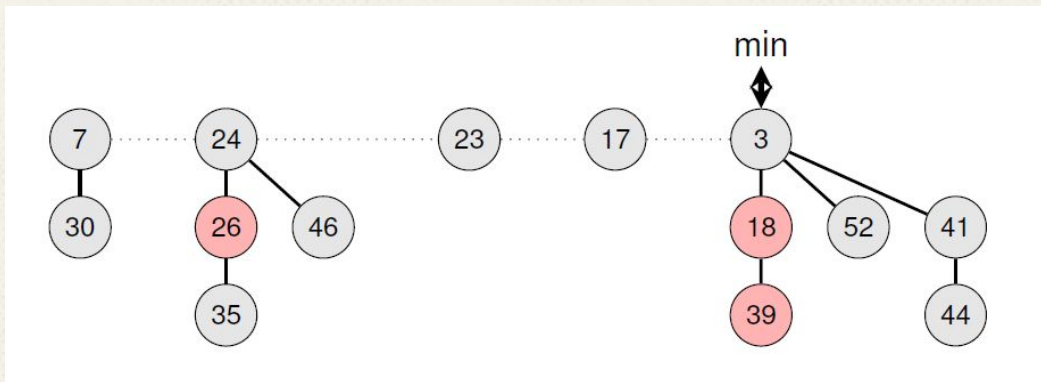
Am putea avea  $n$  arbori cu 1 element.

Dacă ștergem  $n$  elemente consecutive ne poate costa  $n^2$ ??



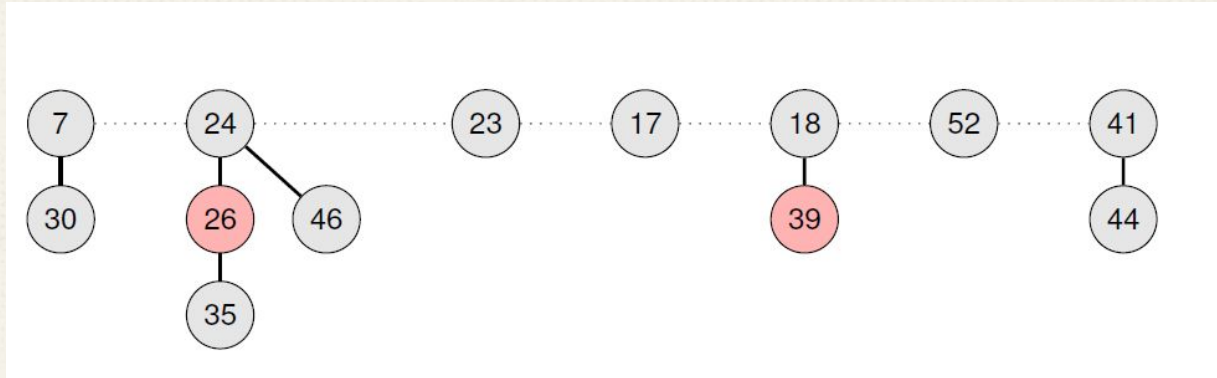
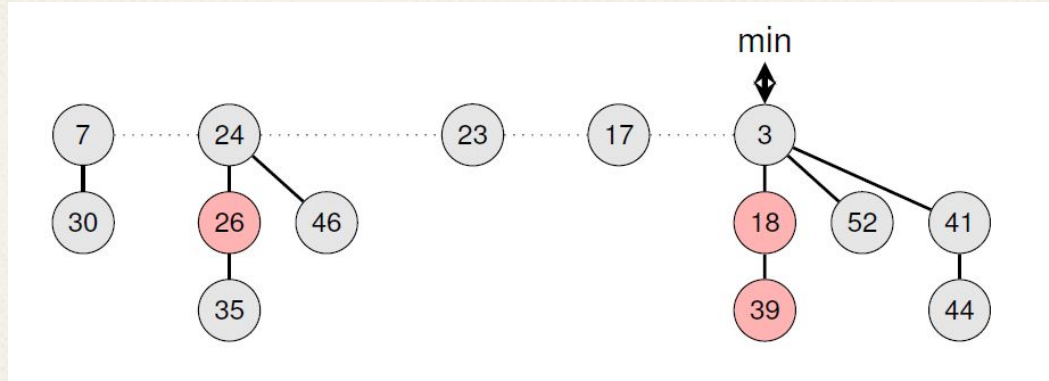
# Extragere minim

- Ca să evităm să avem de mai multe ori cost mare pentru extragerea minimului, vom consolida heapul (“reuniunea” de la heapul binomial).



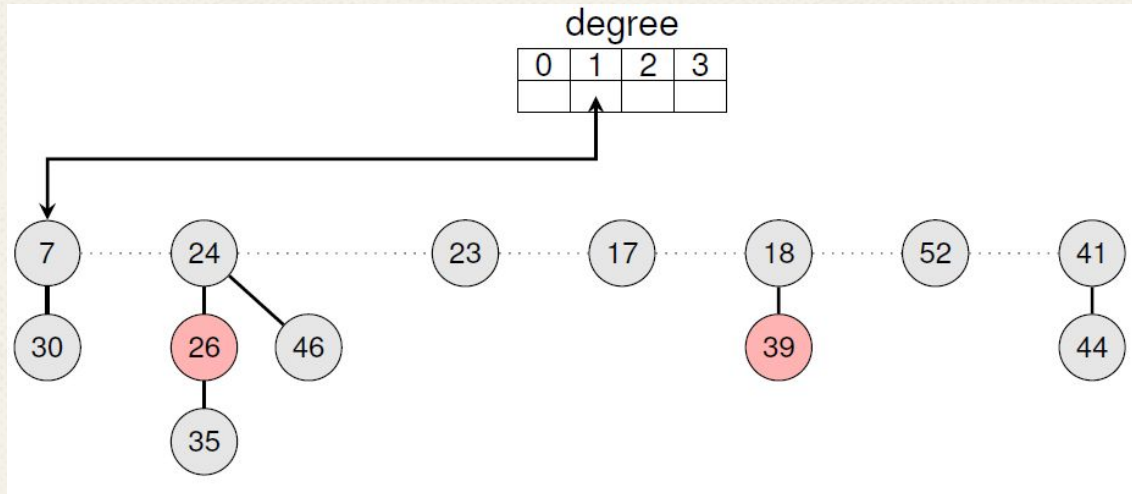
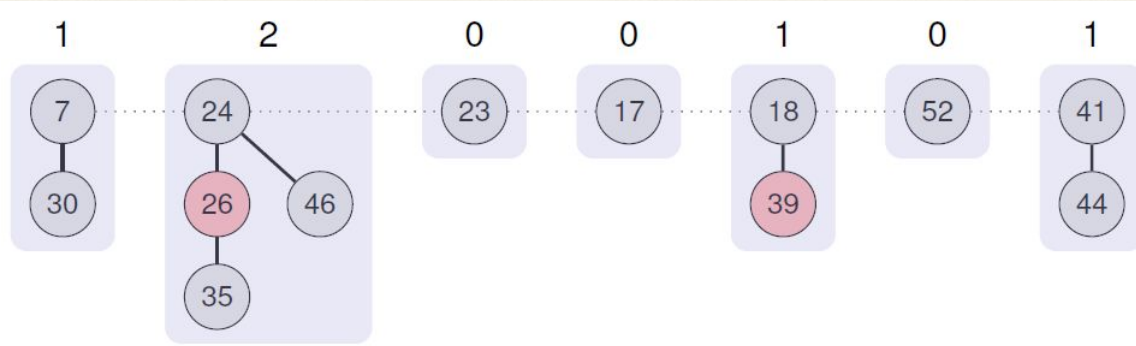
Eliminăm minimul, se creeaza multe “rădăcini”

# Extragere minim

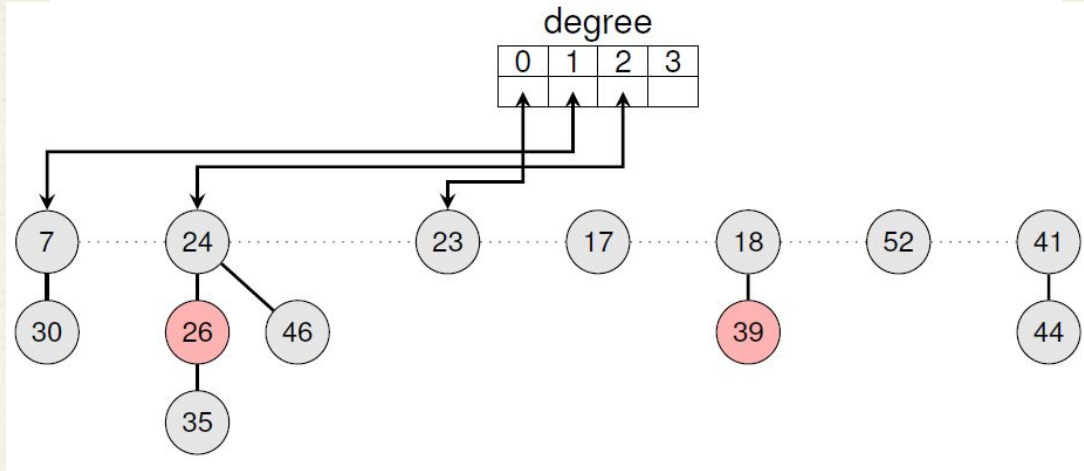
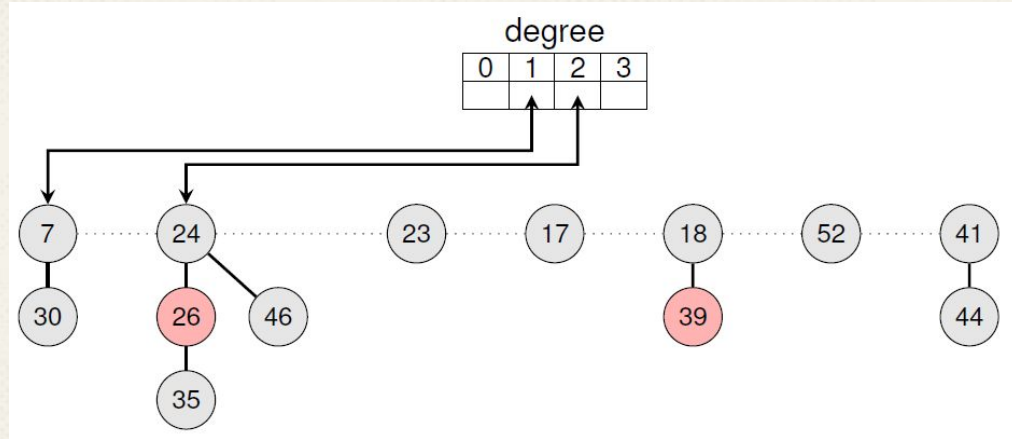


# Extragere minim

grad =

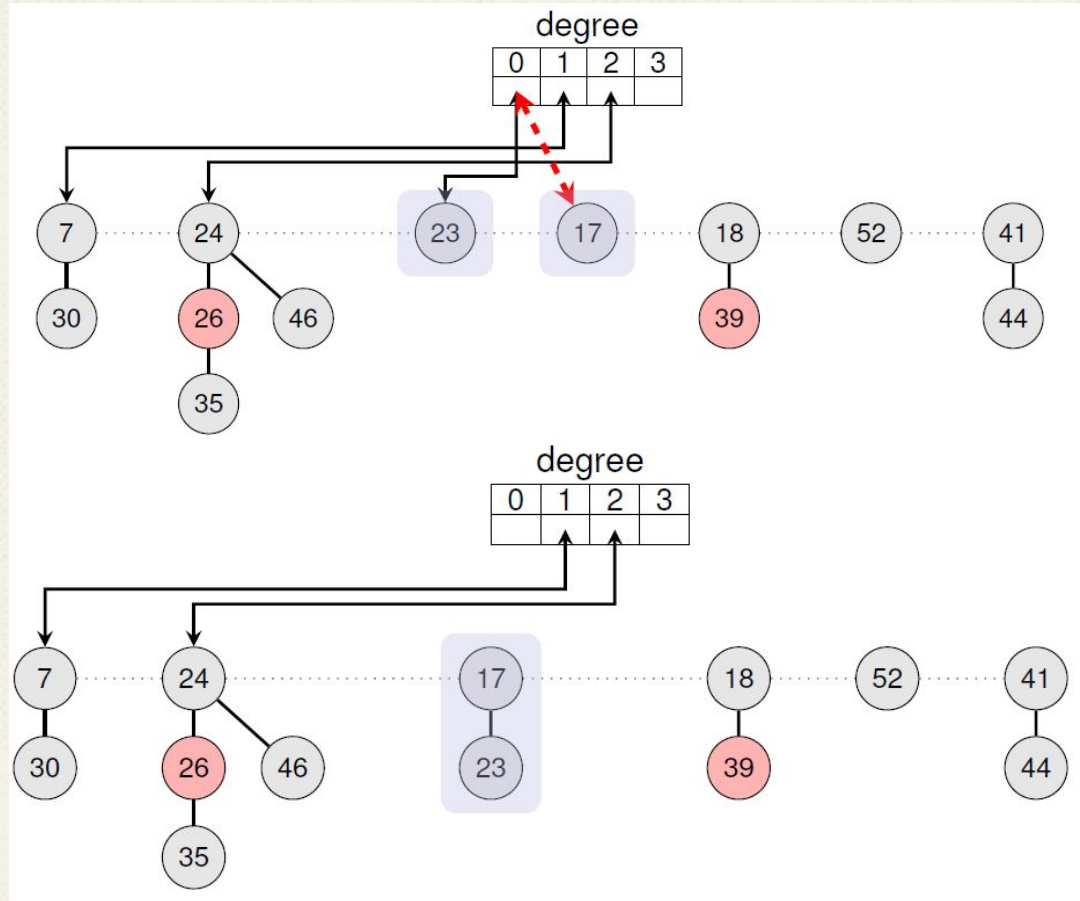


# Extragere minim

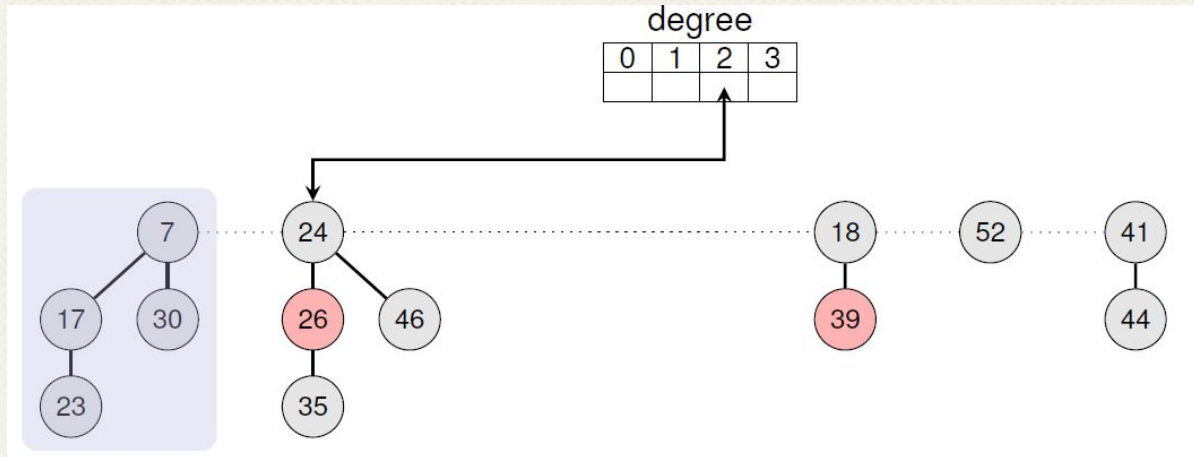
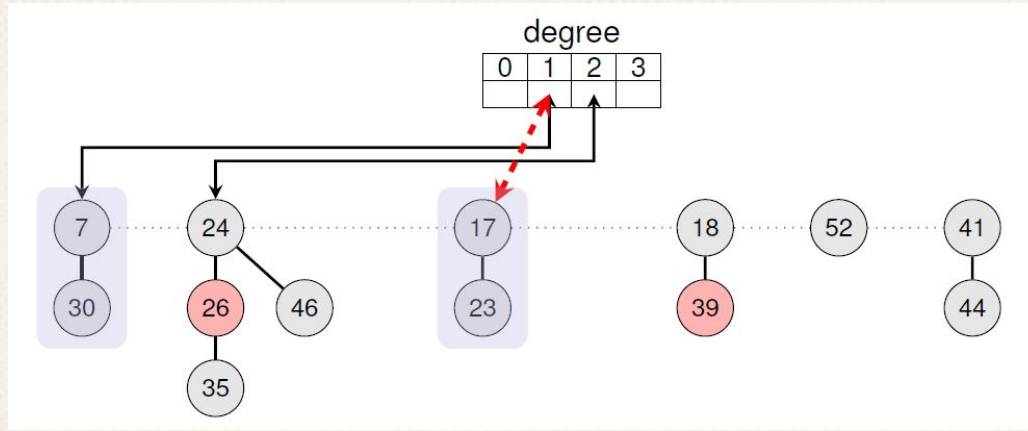




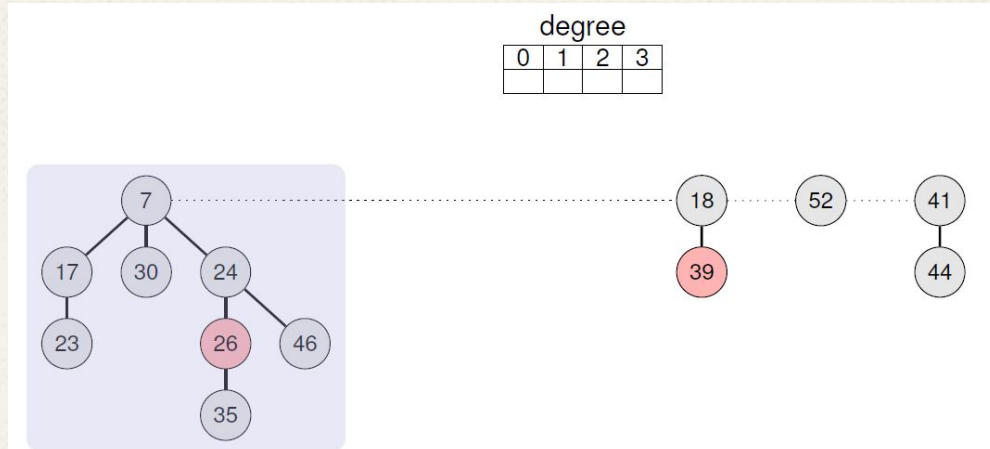
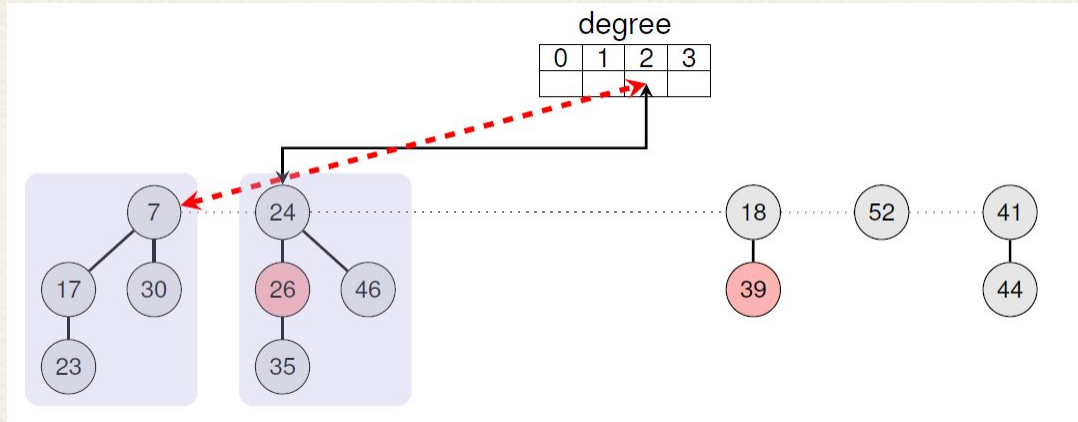
# Extragere minim



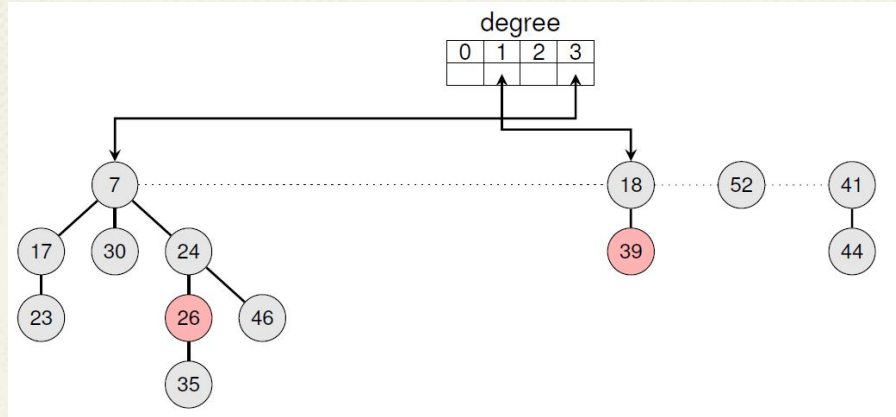
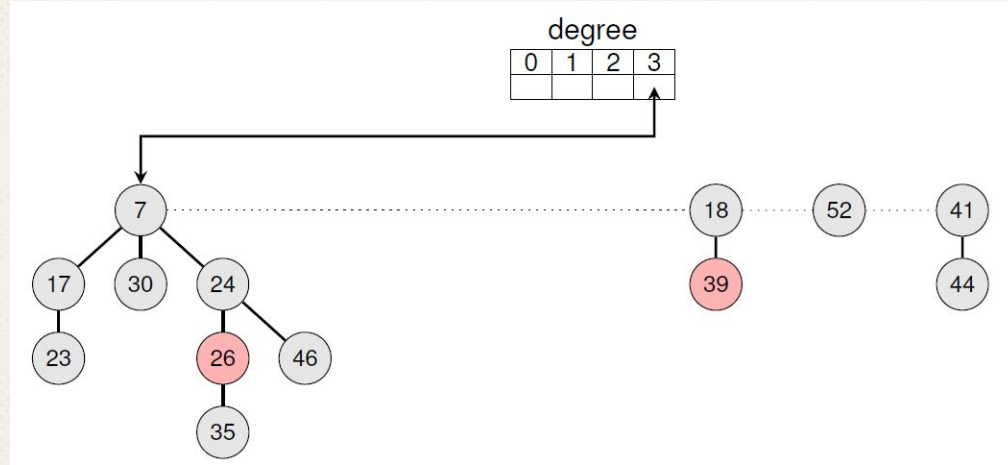
# Extragere minim



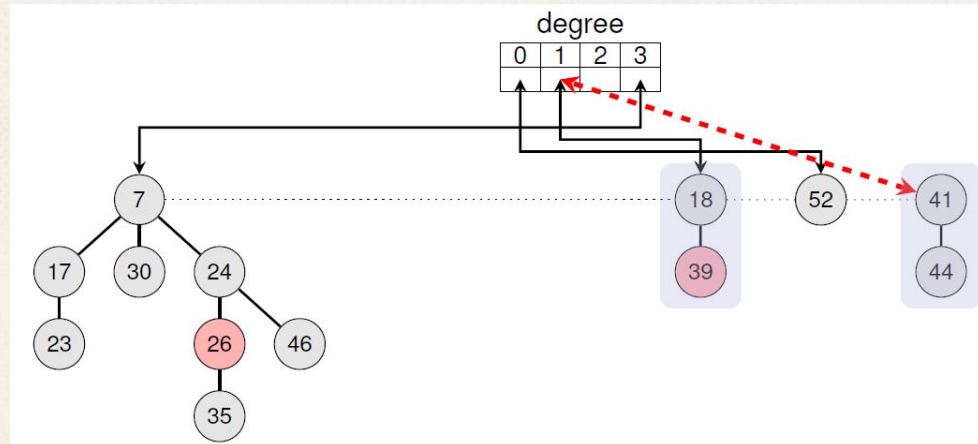
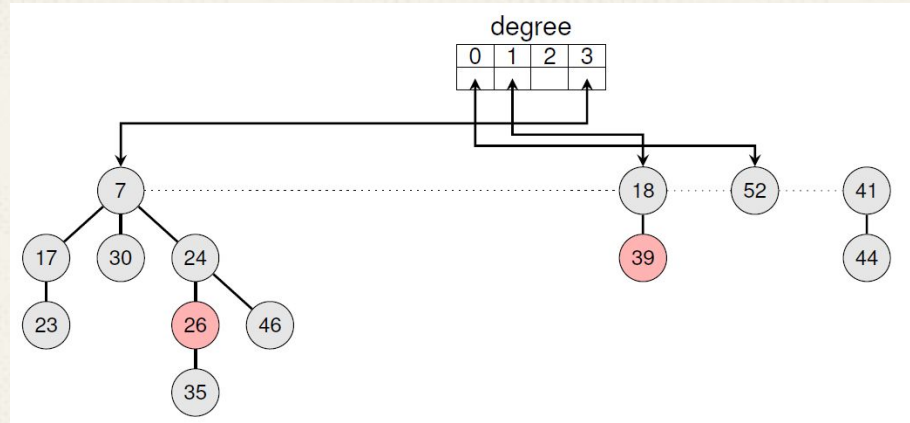
# Extragere minim



# Extragere minim

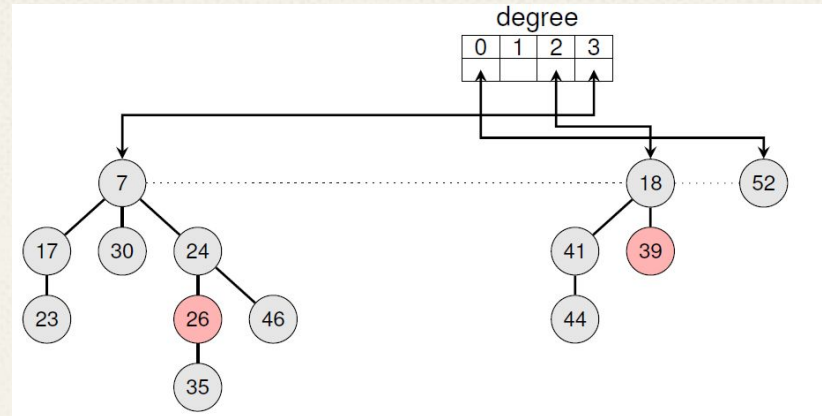
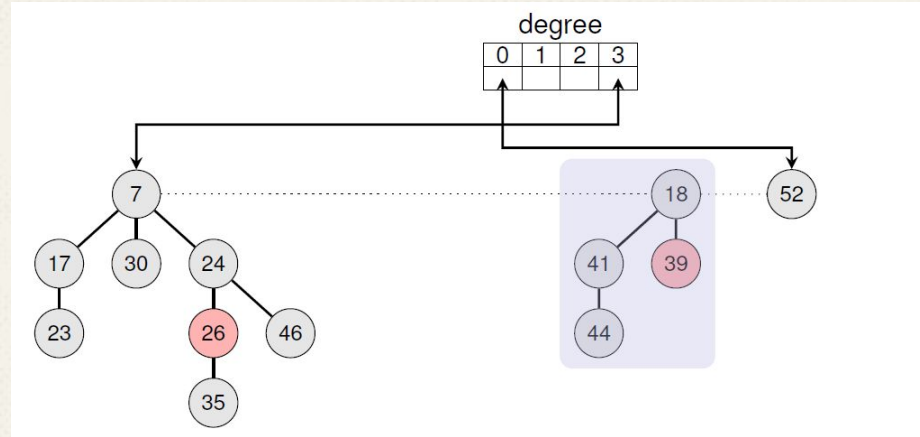


# Extragere minim

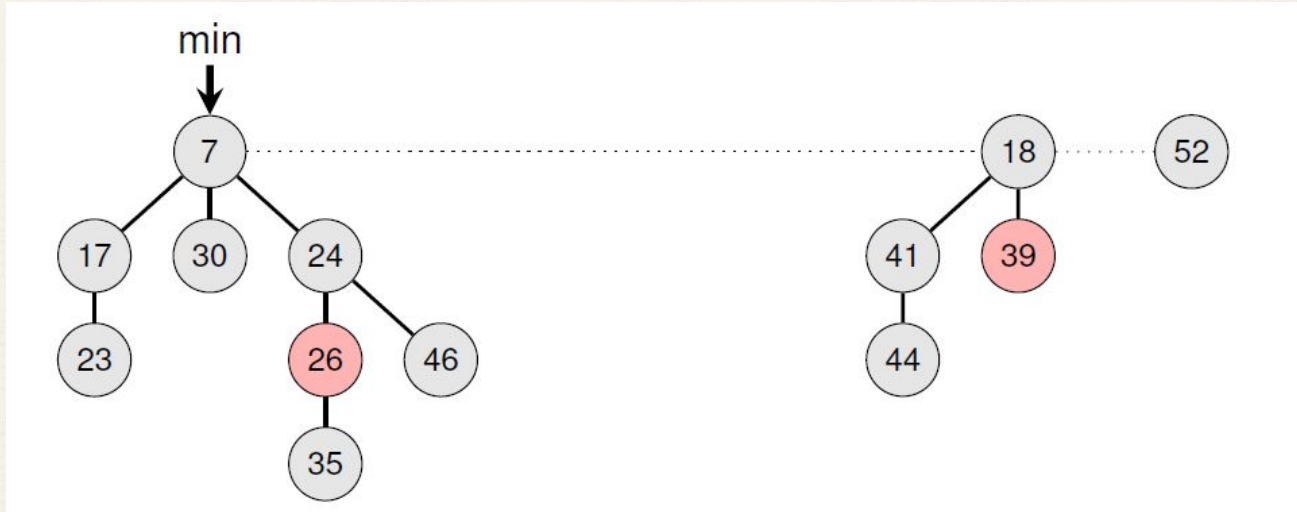




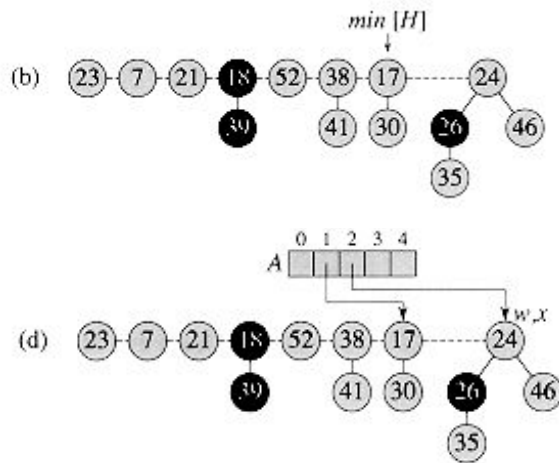
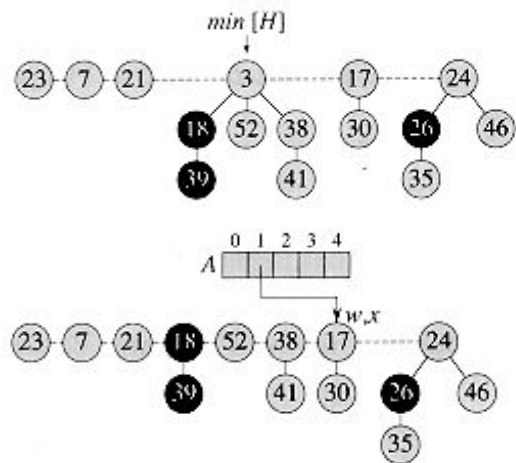
# Extragere minim



# Extragere minim



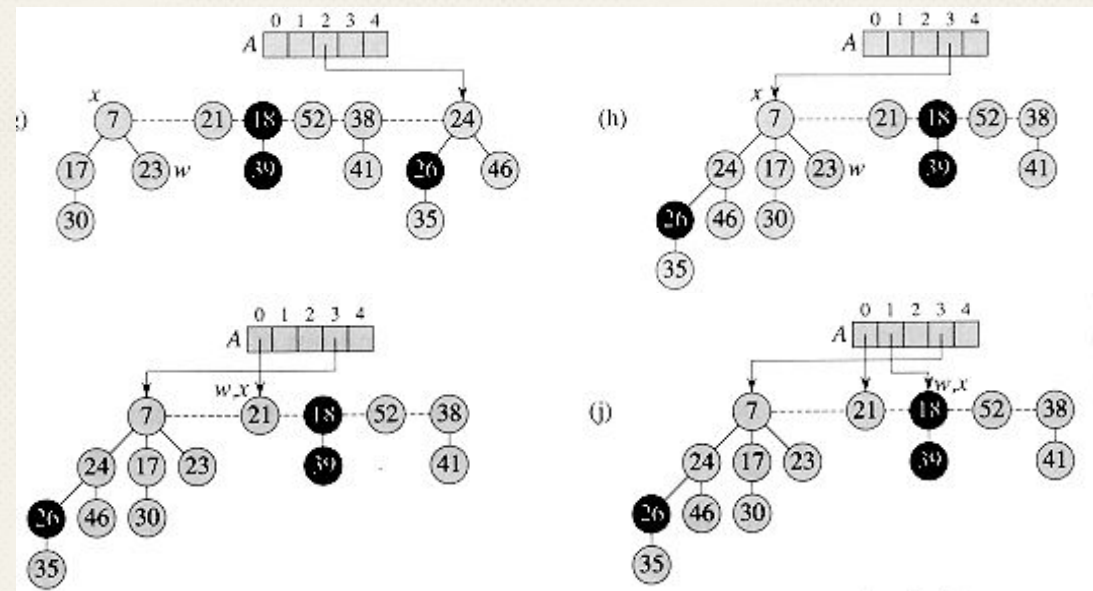
# Extragere minim



---

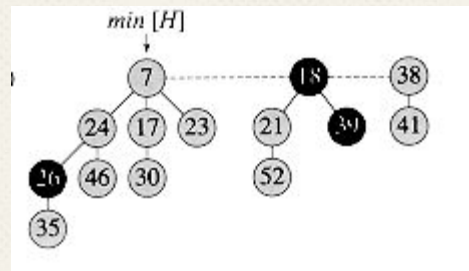
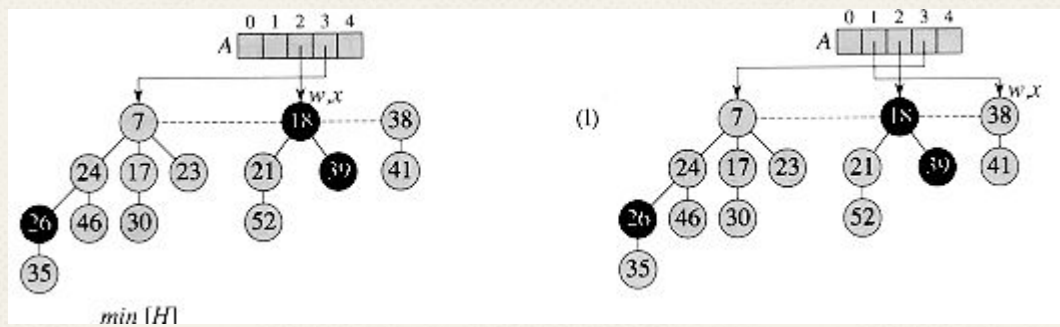
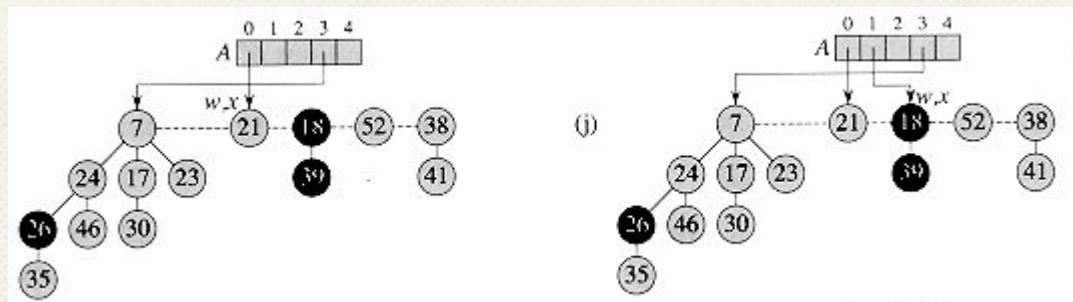


# Extragere minim





# Extragere minim



# Extragere minim

- Complexitate?



# Extragere minim

- Complexitate:
  - $O(n)$  pentru prima
  - $O(?)$  pentru următoarele, dacă nu facem alte operații

# Extragere minim

- Complexitate:

- $O(n)$  pentru prima
- $O(\log n)$  pentru următoarele, dacă nu facem alte operații
- $O(\log n)$  amortizat
- Pentru mai multe detalii despre complexitate urmăriți textul

# Utilitate

Dijkstra (nu ați facut oficial, nu? e timp :) ):

- Cu matrice de adiacență:  $O(n^2)$
- Cu heapuri binare  $O(m \log n)$
- Cu heapuri fibonacci  $O(m + n \log n)$



# Problema

Interclasarea optimală a mai multor șiruri.

Ex: 3 șiruri de lungimi 10, 40 și 90

Interclasarea lui 10 cu 90 → mă costă 100.  $100 + 40 \rightarrow 140$

Total: 240

Interclasarea lui 10 cu 40 → mă costă 50.  $50 + 90 \rightarrow 140$

Total: 180

Interclasarea lui 40 cu 90 → mă costă 130.  $130 + 10 \rightarrow 140$

Total: 270

# Discuție Problema

- Cum rezolvăm ?
- La fiecare pas, trebuie să alegem cele mai mici 2 elemente
- 10 20 30 40 40 60 80
- Optim (10 cu 20) cu 30, 40 cu 40, 60 (primele 3) cu 80 ultimele 2 etc.
- Demonstrație mai târziu

# Discuție Problema

Complexitate?

# Discuție Temă

Complexitate?

- $O(n^2)$  dacă la fiecare pas găsim cele mai mici 2 elemente iterând prin toate elementele rămase.

# Discuție Problema

## Complexitate?

- $O(n^2)$  dacă la fiecare pas găsim cele mai mici 2 elemente iterând prin toate elementele rămase.
- $O(n \log n)$  dacă folosim heapuri să reținem toate valorile (inclusiv cele obținute prin uniune).
- Dacă elementele sunt deja sortate sau putem folosi Counting Sort  $\rightarrow O(n)$ .
  - Folosim 2 cozi: una cu valorile inițiale sortate, a doua cu valorile sumelor în ordinea care vin (vor fi și ele sortate)



# Discuție Problema

## Complexitate?

- Dacă elementele sunt deja sortate sau putem folosi Counting Sort  $\rightarrow O(n)$ .
  - Folosim 2 cozi una cu valorile inițiale sortate, a doua cu valorile sumelor în ordinea care vin (vor fi și ele sortate)
  - 10 20 30 40 40 70 | nimic
  - 30 40 40 70 | 30 (după ce am unit 10 cu 20)
  - 40 40 70 | 60 (după ce am unit 30 cu 30)
  - 70 | 60 80 (după ce am unit 40 cu 40)
  - nimic | 80 130 (după ce am unit 60 cu 80)
  - Nimic | 210

# Bibliografie

<https://ocw.cs.pub.ro/courses/sd-ca/laboratoare/laborator-11>

<https://www.slideshare.net/HoangNguyen446/heaps-61679009>

<https://www.infoarena.ro/heapuri>

<https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/heaps.pdf>

[https://en.wikipedia.org/wiki/Binary\\_heap](https://en.wikipedia.org/wiki/Binary_heap)

[https://en.wikipedia.org/wiki/Heap\\_\(data\\_structure\)](https://en.wikipedia.org/wiki/Heap_(data_structure))

<https://www.geeksforgeeks.org/binomial-heap-2/>

Cursuri Structuri de Date și Algoritmi Rodica Ceterchi