# Coada cu priorități.
# Colecții de mulțimi disjuncte

SD 2019/2020

# Conținut

# Coada cu priorități – exemple

- Pasagerii unui avion

  *Priorități*:
  - clasa “business”;
  - persoane călatorind cu copii / cu mobilitate redusă;
  - ceilalți pasageri.

# Coada cu priorități – exemple

- Pasagerii unui avion

  *Priorități*:
  - clasa “business”;
  - persoane călatorind cu copii / cu mobilitate redusă;
  - ceilalți pasageri.

- Avioane care se pregătesc să aterizeze

  *Priorități*
  - urgențe;
  - nivelul carburantului;
  - distanța față de aeroport.

# Tipul abstract `Coada cu priorităţi`

- OBIECTE:
  - structuri de date în care elementele sunt numite atomi;
  - orice atom are un câmp *cheie* numit prioritate.

- Elementele sunt memorate în funcție de prioritate și nu de poziția lor.

# Coada cu priorităţi – operații

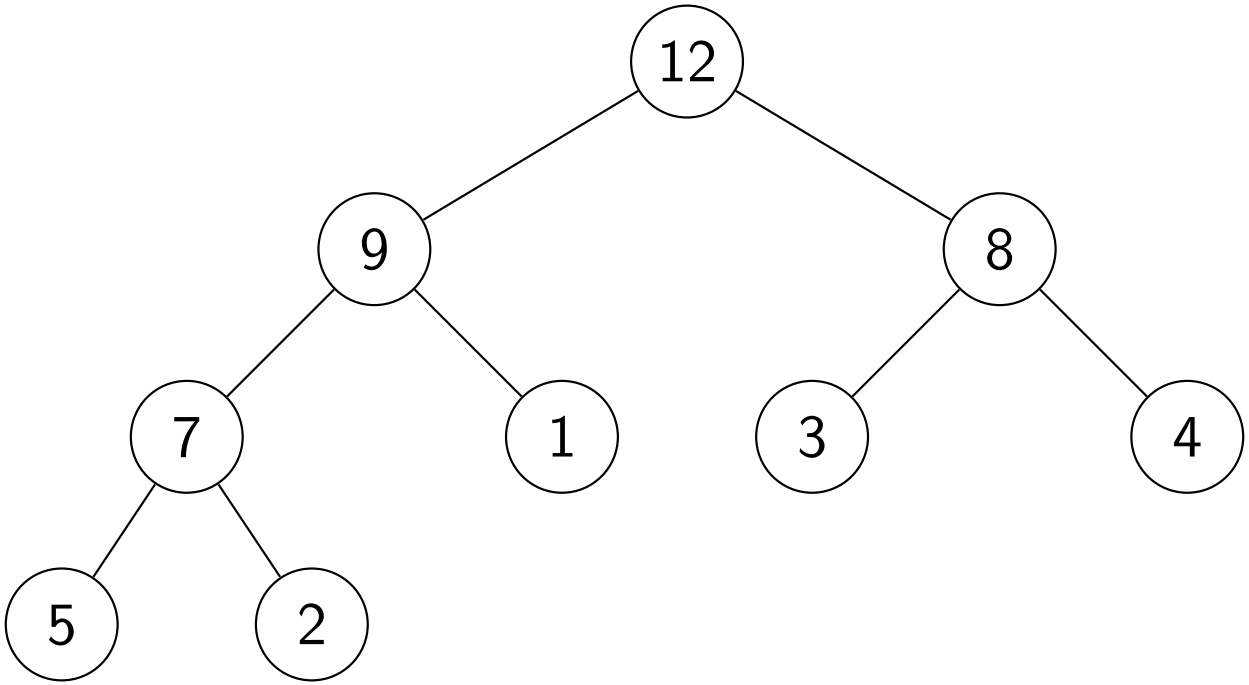
- `citeşte`
  - intrare: o coadă cu priorități $C$
  - ieșire: atomul din $C$ cu cheia cea mai mare.
- `elimină`
  - intrare: o coadă cu priorități $C$
  - ieșire: $C$ din care s-a eliminat atomul cu cheia cea mai mare.
- `inserează`
  - intrare: o coadă cu priorități $C$ și un atom $at$
  - ieșire: $C$ la care s-a adăugat $at$.

# maxHeap

- Implementează coada cu priorități.

- Arbori binari cu proprietățile:
  - Nodurile memorează câmpurile *cheie*;
  - Pentru orice nod, cheia din acel nod este mai mare decât sau egală cu cheile din nodurile fiu;
  - Arborele este complet. Fie $h$ înălțimea arborelui. Atunci,
    - Pentru $i = 0, \cdots, h-1$, sunt $2^i$ noduri cu adâncimea $i$;
    - Pe nivelul $h-1$ nodurile interne sunt situate la stânga nodurilor externe.
  - Ultimul nod al unui maxHeap este nodul cel mai la dreapta pe nivelul $h$.

# maxHeap – exemplu



# Înălțimea unui `maxHeap`

## Teoremă

*Un* `maxHeap` *care conține* $n$ *chei are înălțimea* $O(\log_2 n)$.

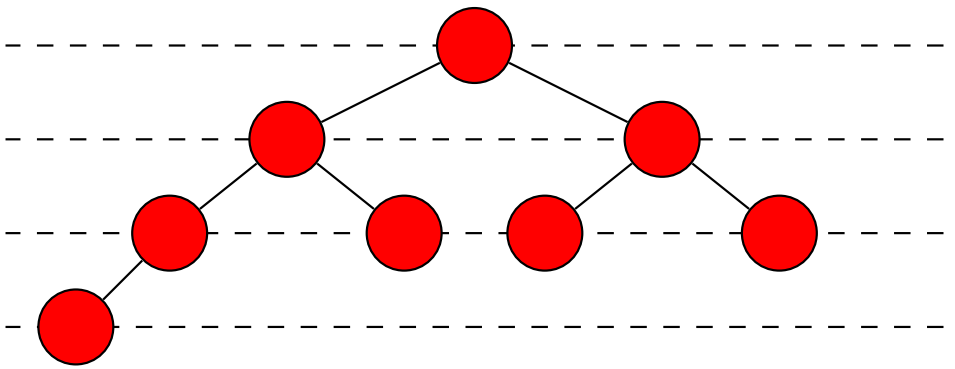
# Înălțimea unui `maxHeap`

## Teoremă

*Un* `maxHeap` *care conține $n$ chei are înălțimea* $O(\log_2 n)$.

## Demonstrație.

- Utilizăm proprietatea de arbore binar complet.
- Fie $h$ înălțimea unui `maxHeap` cu $n$ chei.
- Avem $2^i$ chei de adâncime $i$, pentru $i = 0, \cdots, h-1$ și cel puțin o cheie de adâncime $h$: $\Rightarrow n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 = 2^h$.
- Obținem: $h \leq \log_2 n$



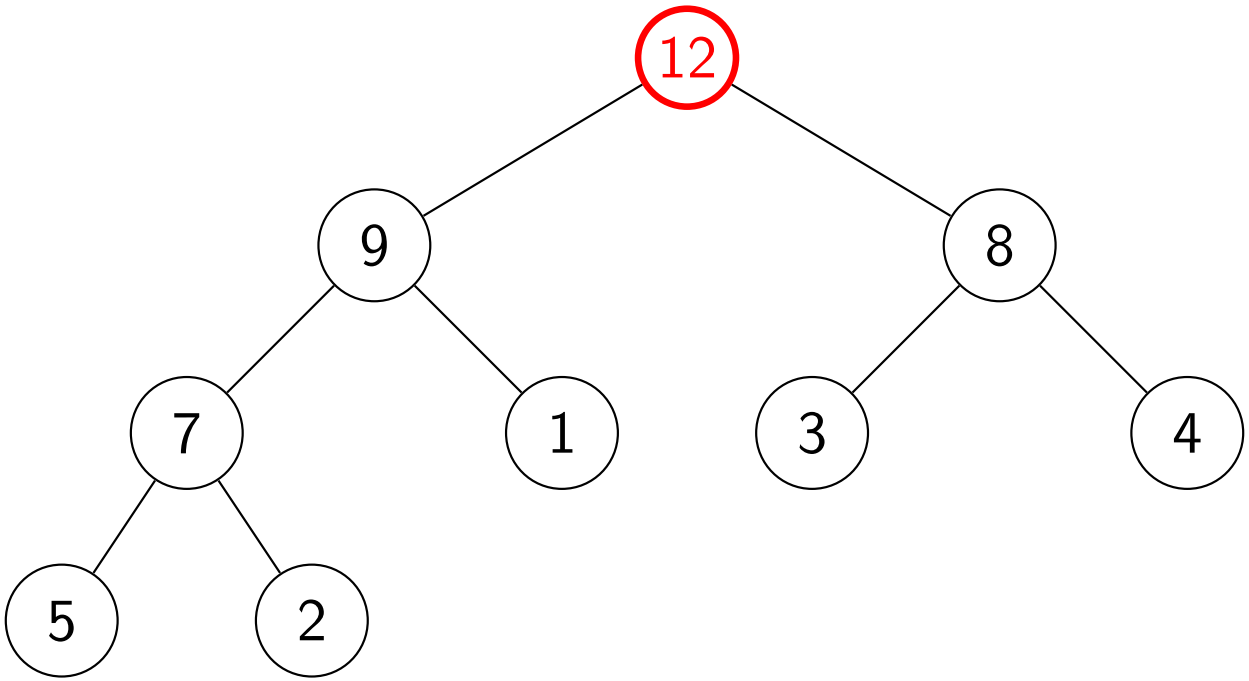
| nivel | chei |
|---|---|
| 0 | $2^0$ |
| 1 | $2^1$ |
| $h-1$ | $2^{h-1}$ |
| $h$ | 1 |

# maxHeap: eliminarea

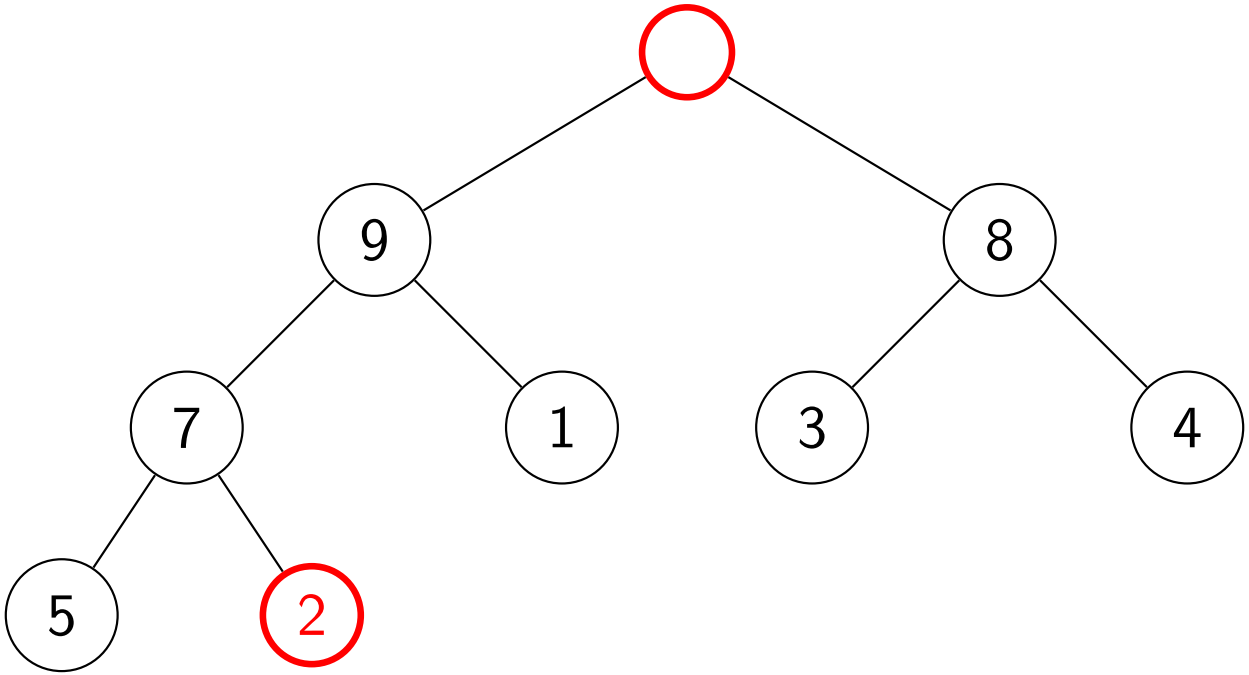
- Se elimină rădăcina heap-ului
  (corespunde elementului cel mai prioritar).

- Algoritmul are trei etape:
  - Se înlocuiește cheia rădăcinii cu cheia ultimului nod;
  - Se șterge ultimul nod (de pe ultimul nivel);
  - Se reface proprietatea de `maxHeap`.

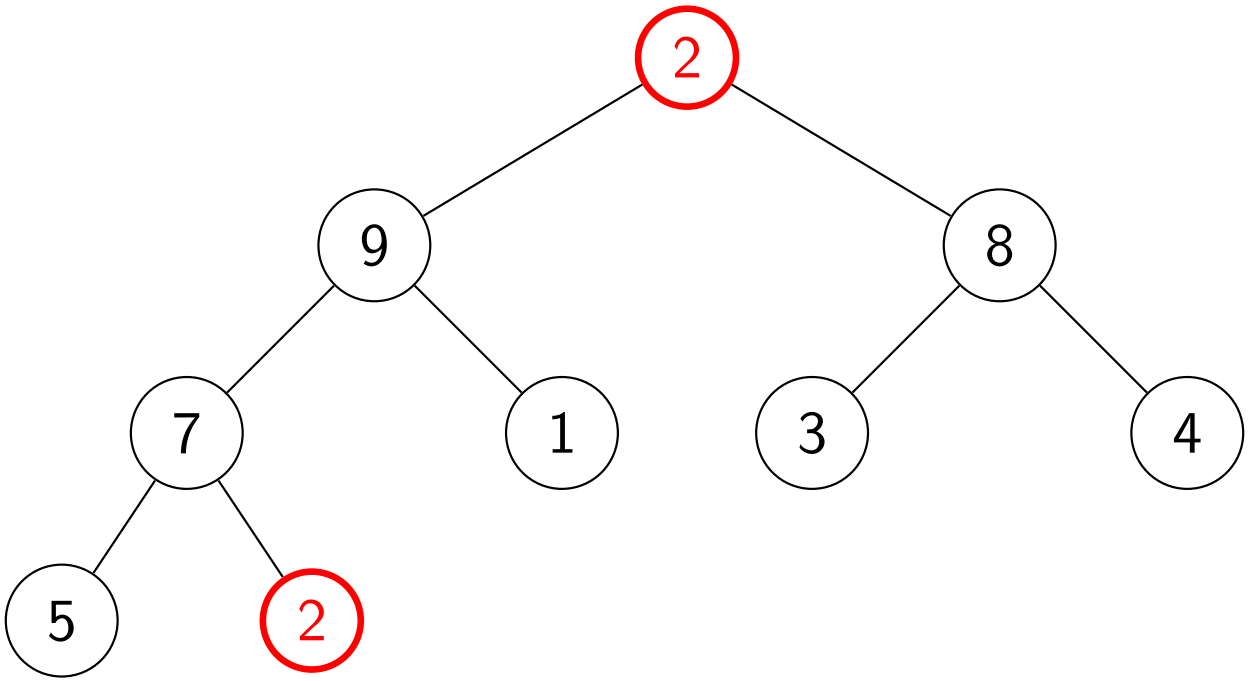
# maxHeap: eliminare – exemplu



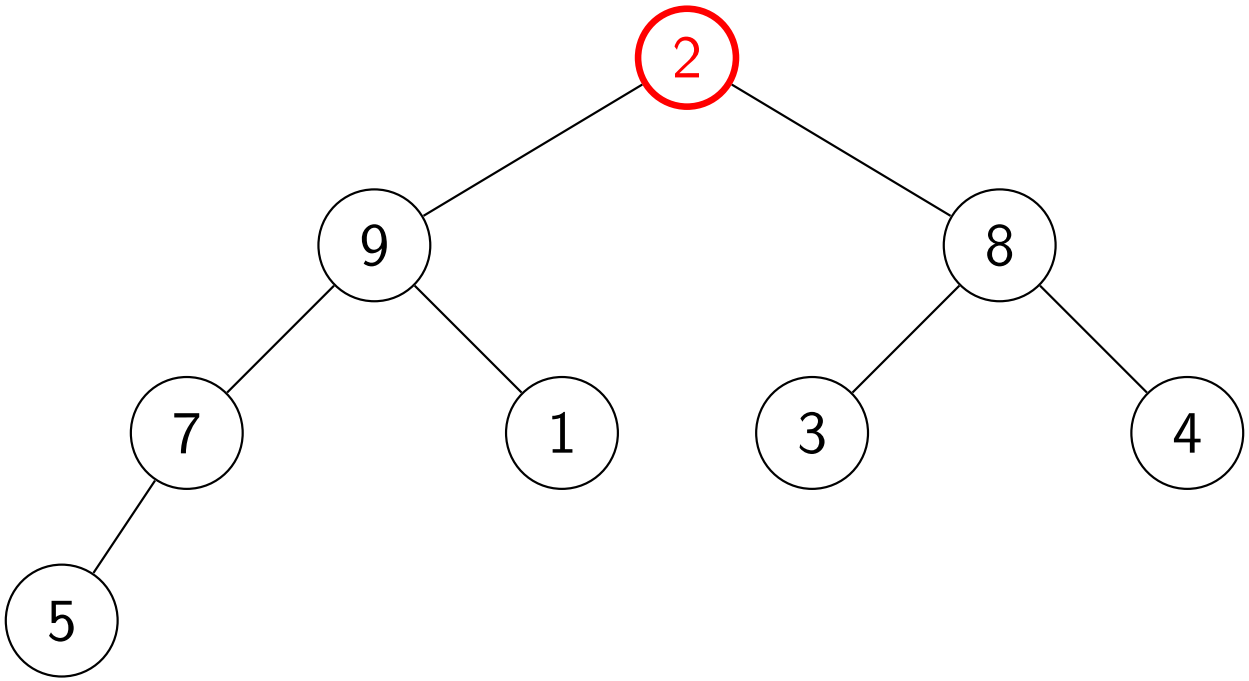
# maxHeap: eliminare – exemplu



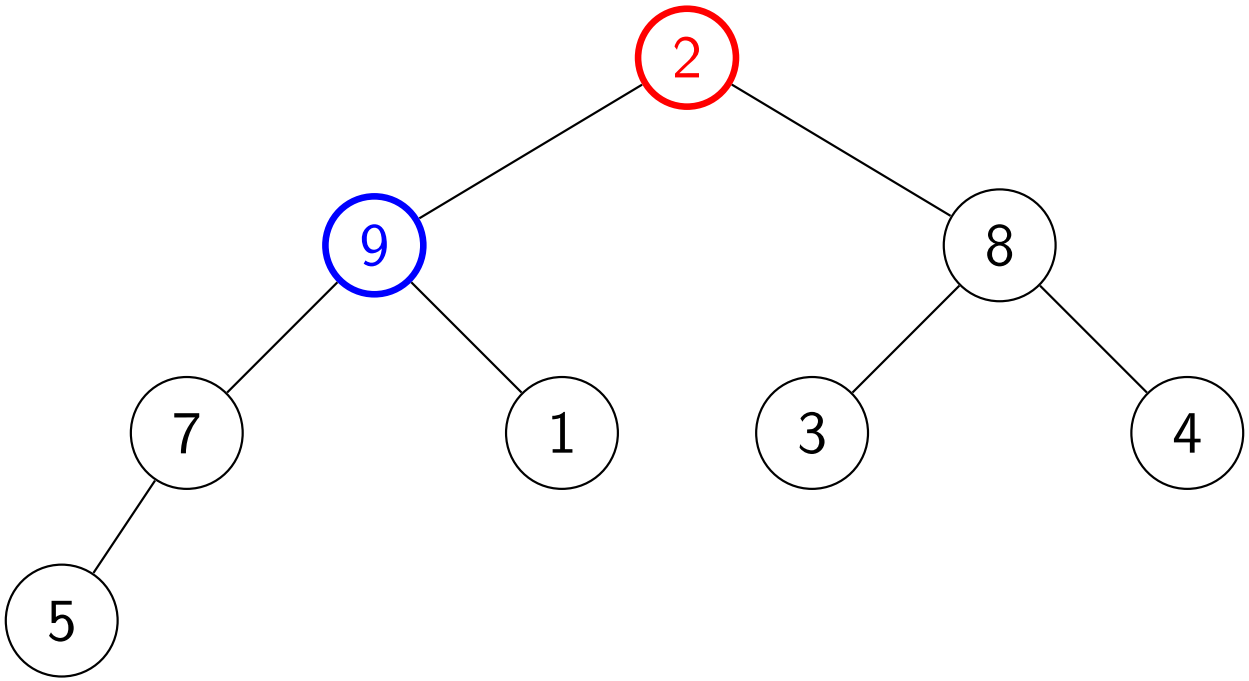
# maxHeap: eliminare – exemplu



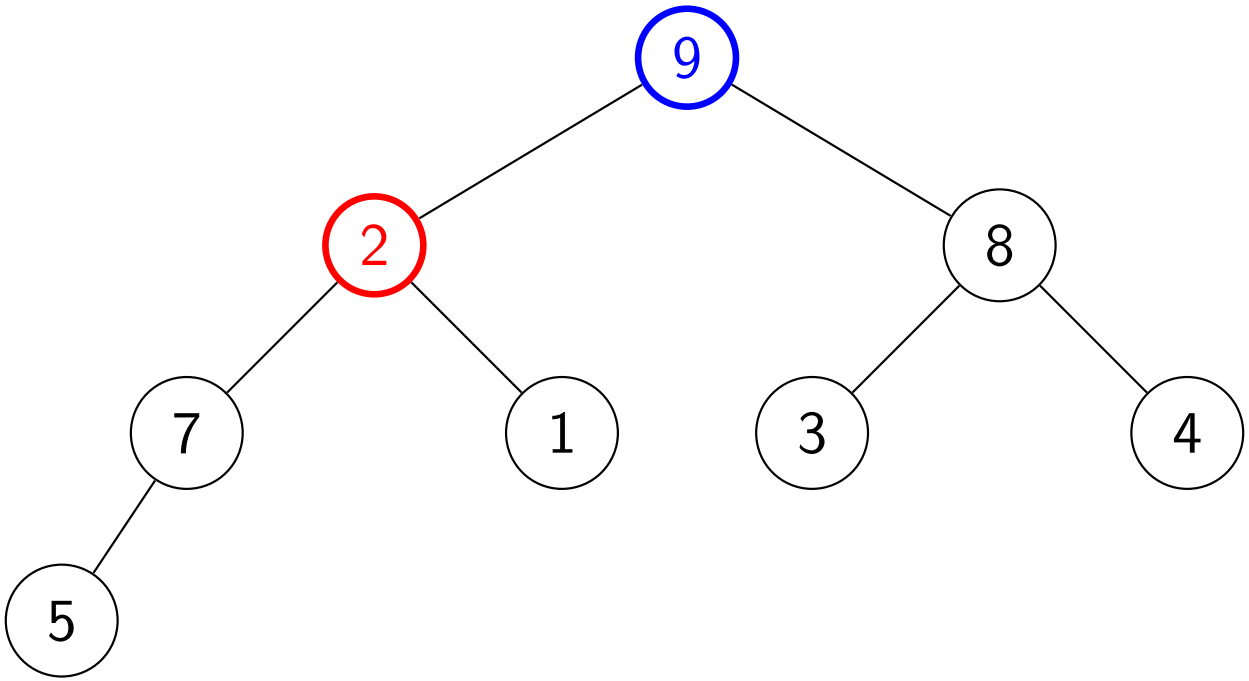
# maxHeap: eliminare – exemplu



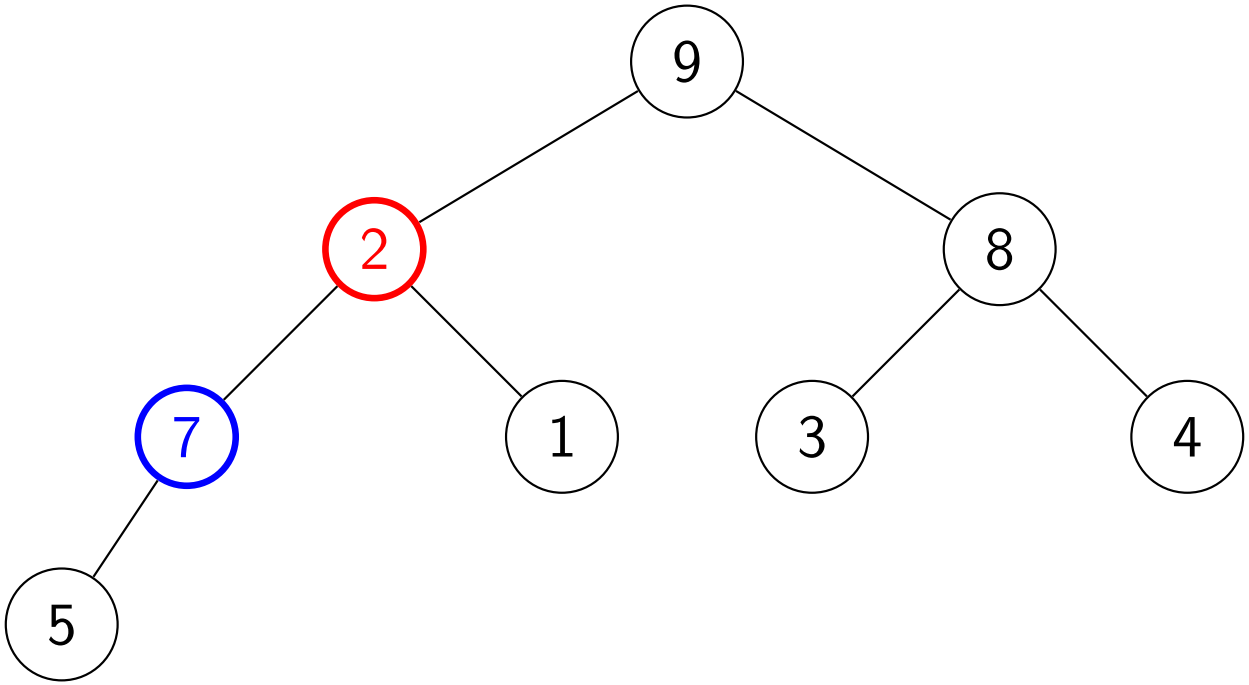
# maxHeap: eliminare – exemplu



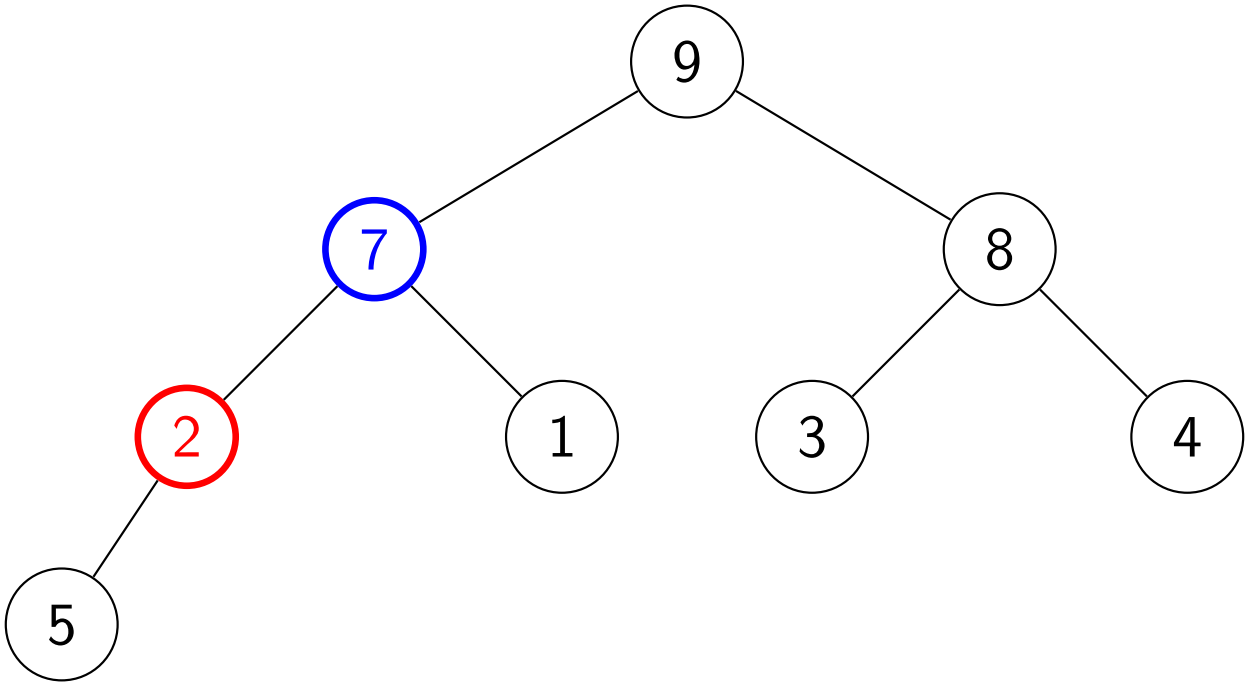
# maxHeap: eliminare – exemplu



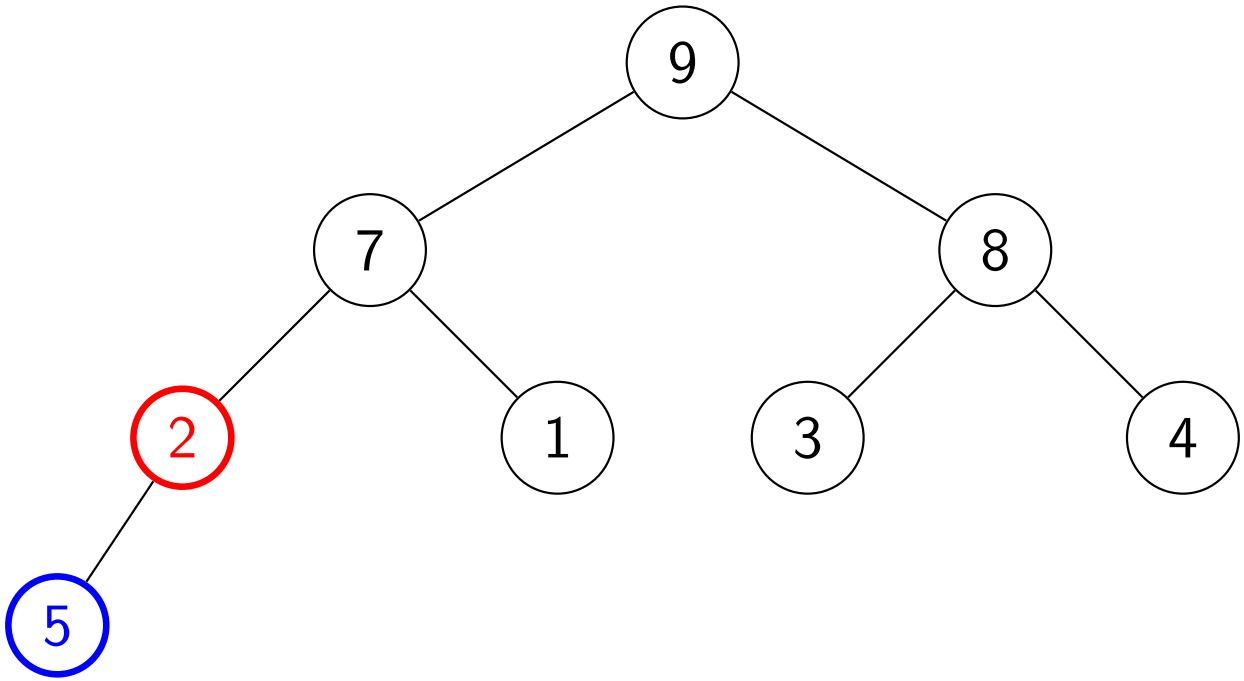
# maxHeap: eliminare – exemplu



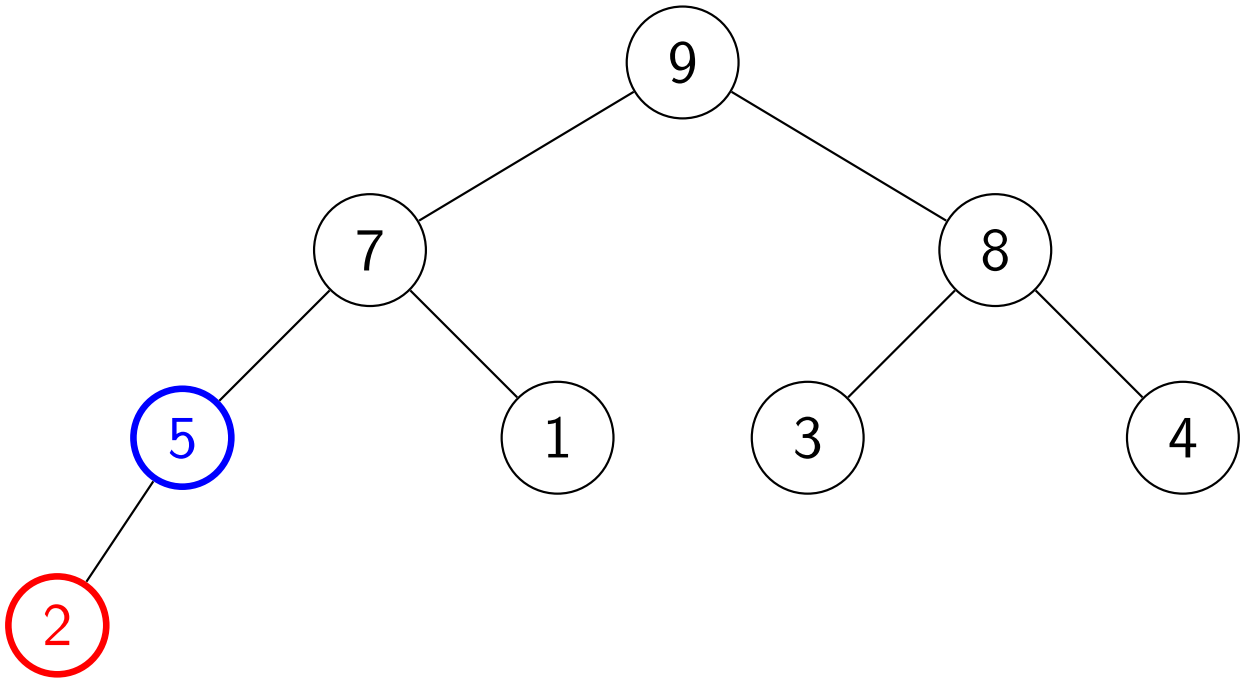
# maxHeap: eliminare – exemplu



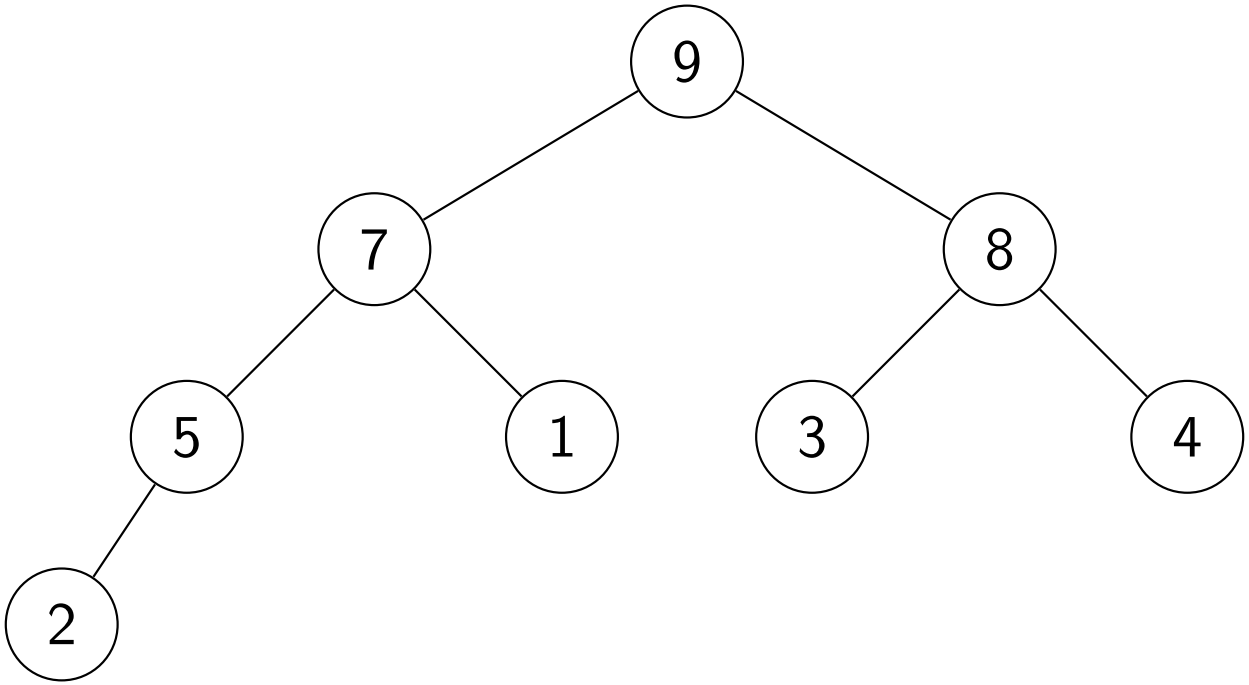
# maxHeap: eliminare – exemplu



# maxHeap: eliminare – exemplu



# maxHeap: eliminare – exemplu

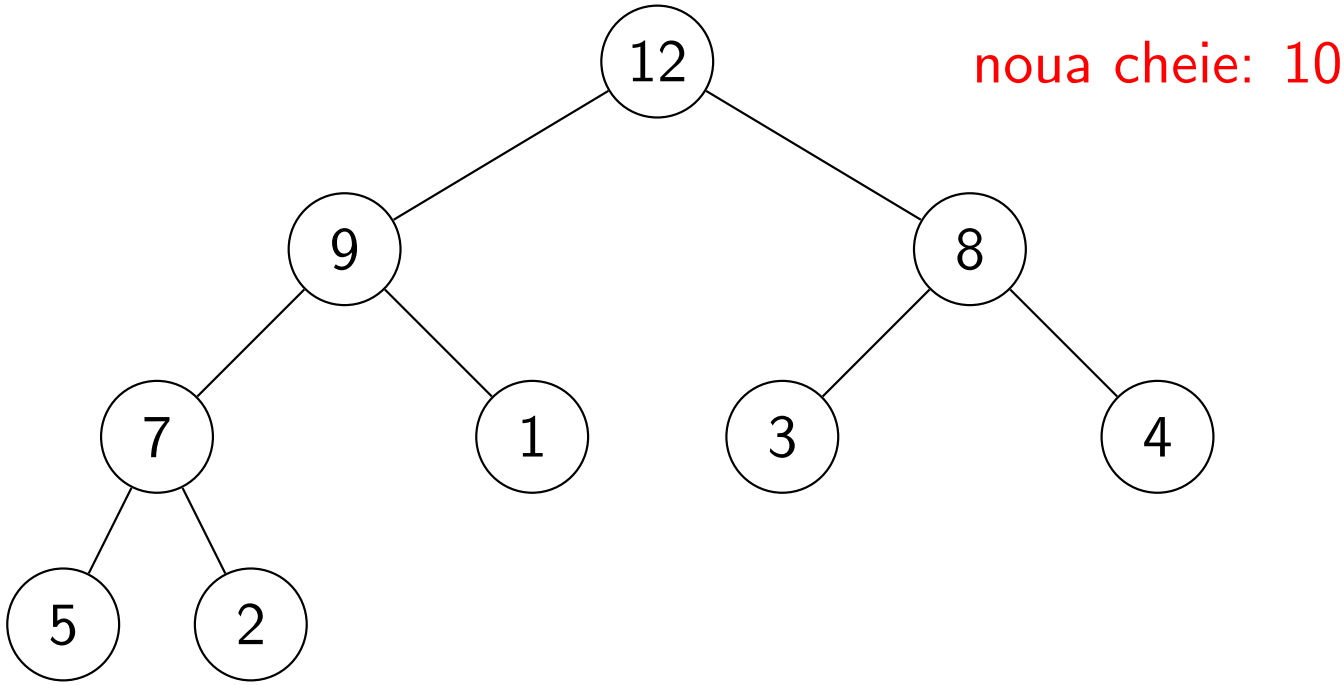


# maxHeap: inserarea

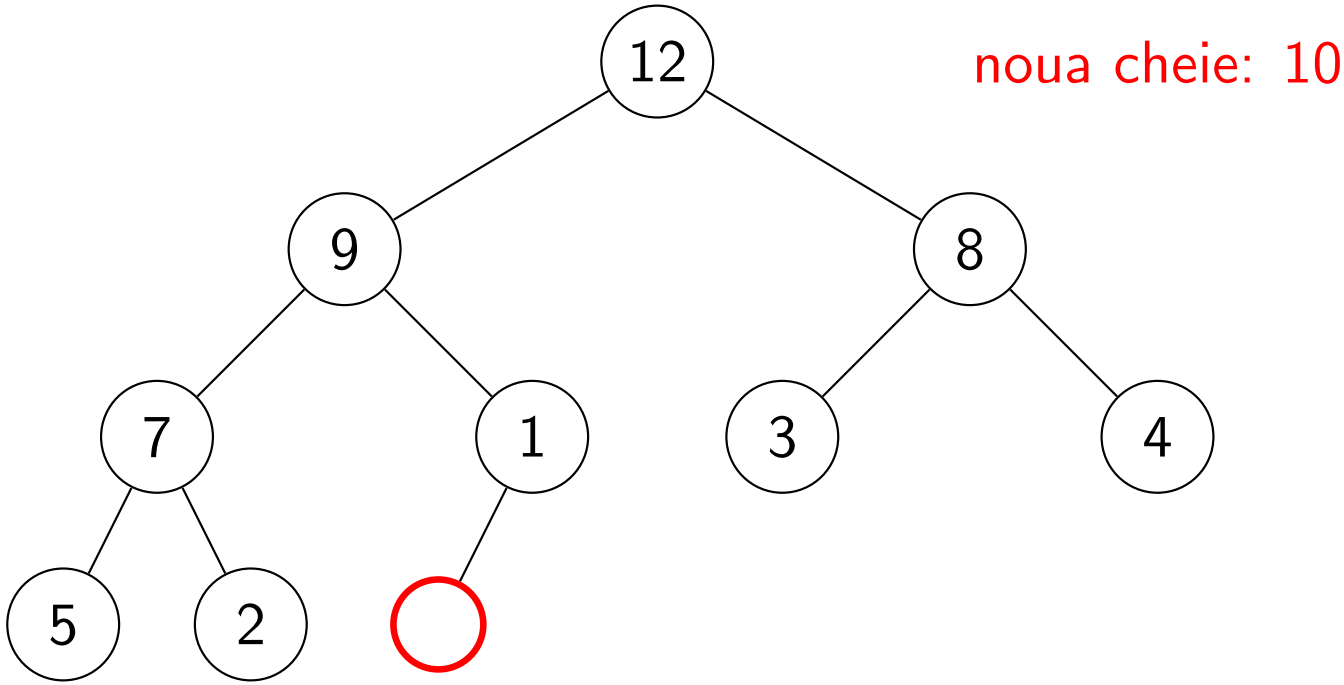
- Se inserează noua cheie într-un nou nod.

- Algoritmul are trei etape:
  - Se adaugă noul nod ca cel mai din dreapta pe ultimul nivel;
  - Se inserează noua cheie în acest nod;
  - Se reface proprietatea de maxHeap.

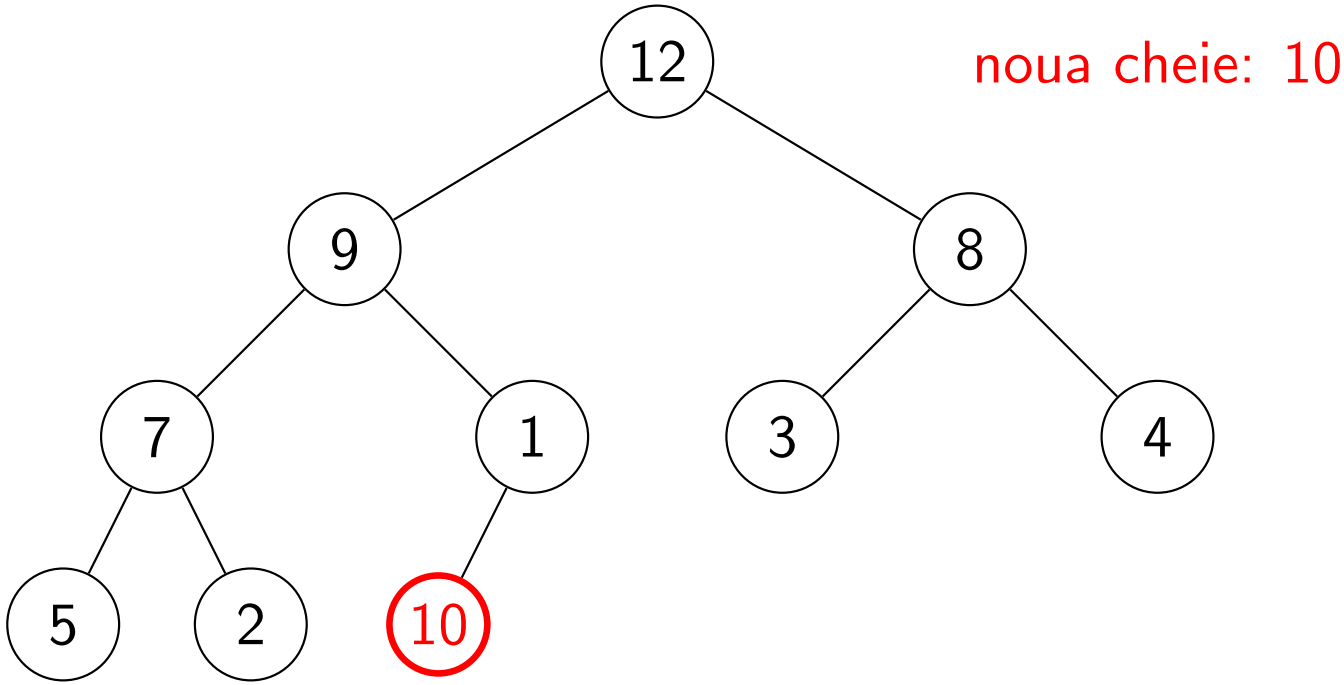
# maxHeap: inserare – exemplu



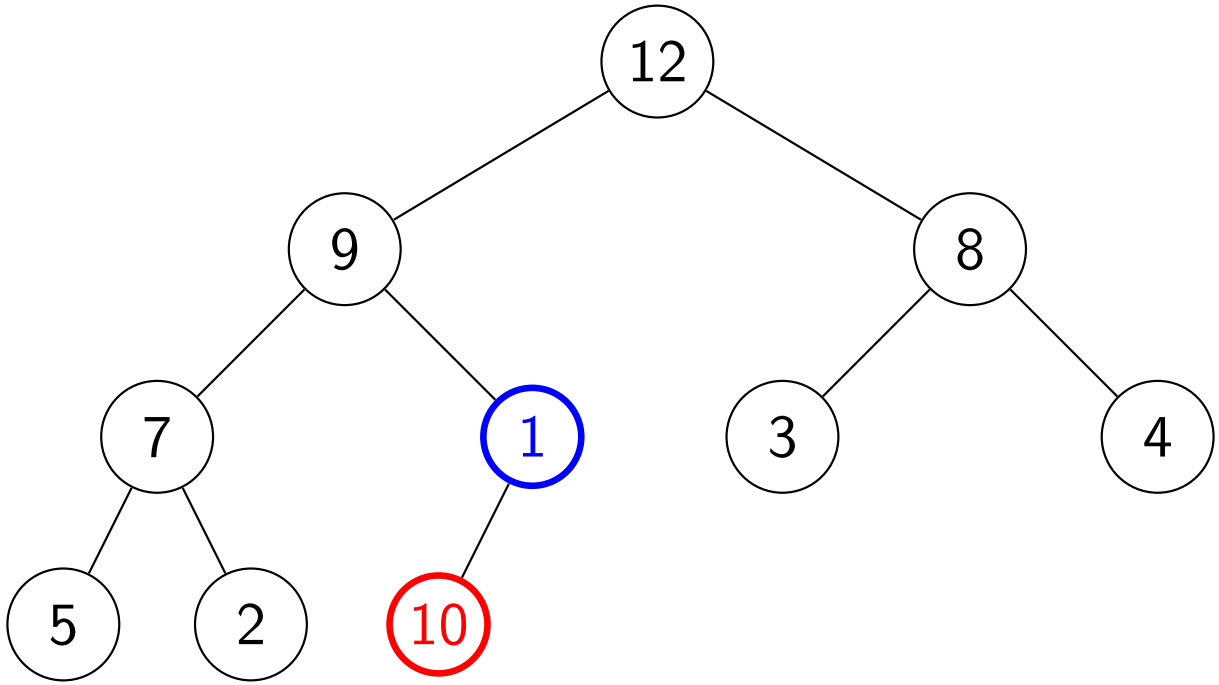
# maxHeap: inserare – exemplu



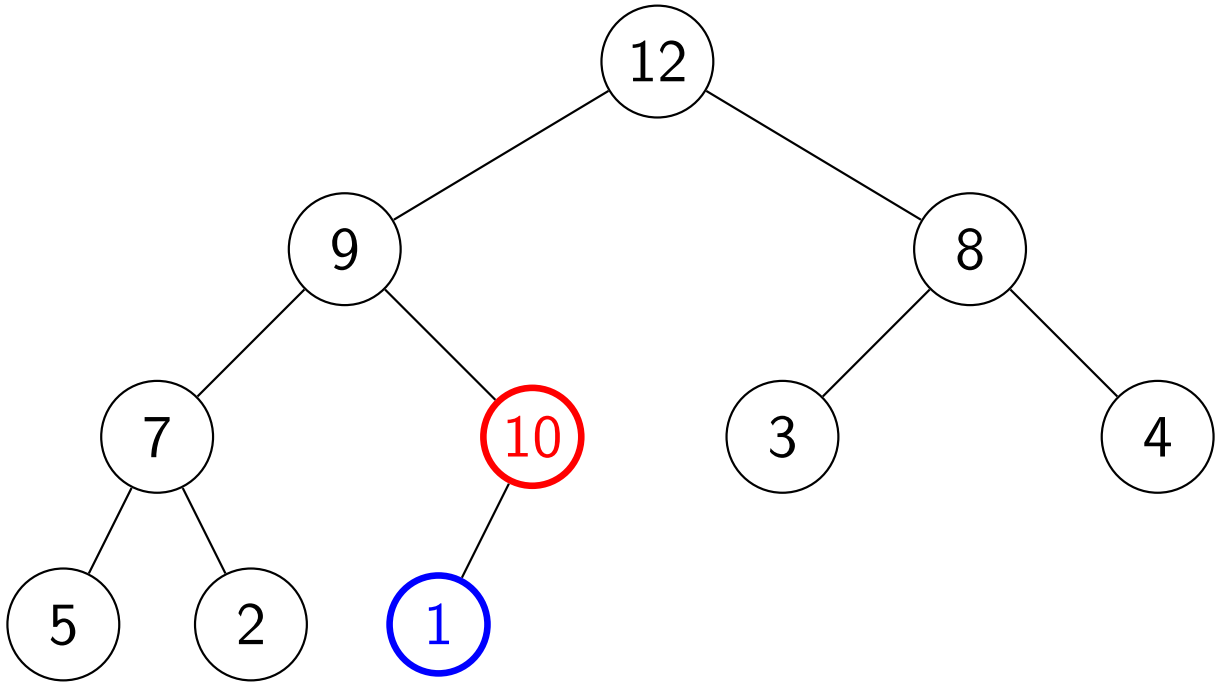
# maxHeap: inserare – exemplu



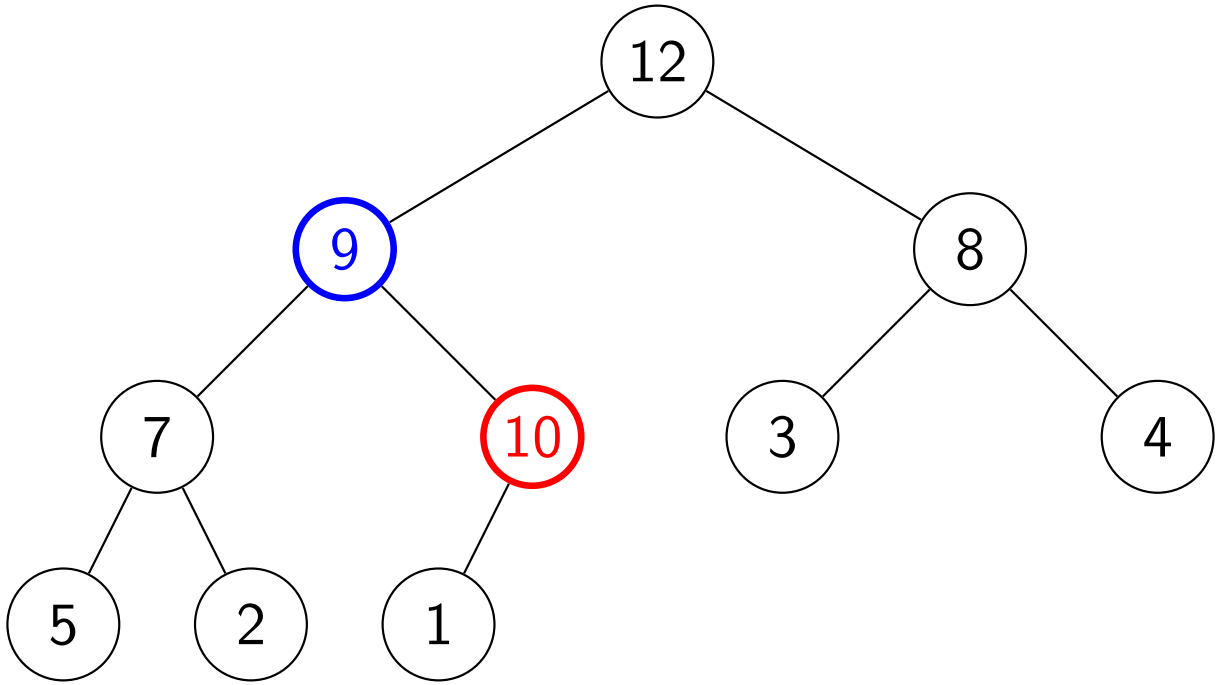
# maxHeap: inserare – exemplu



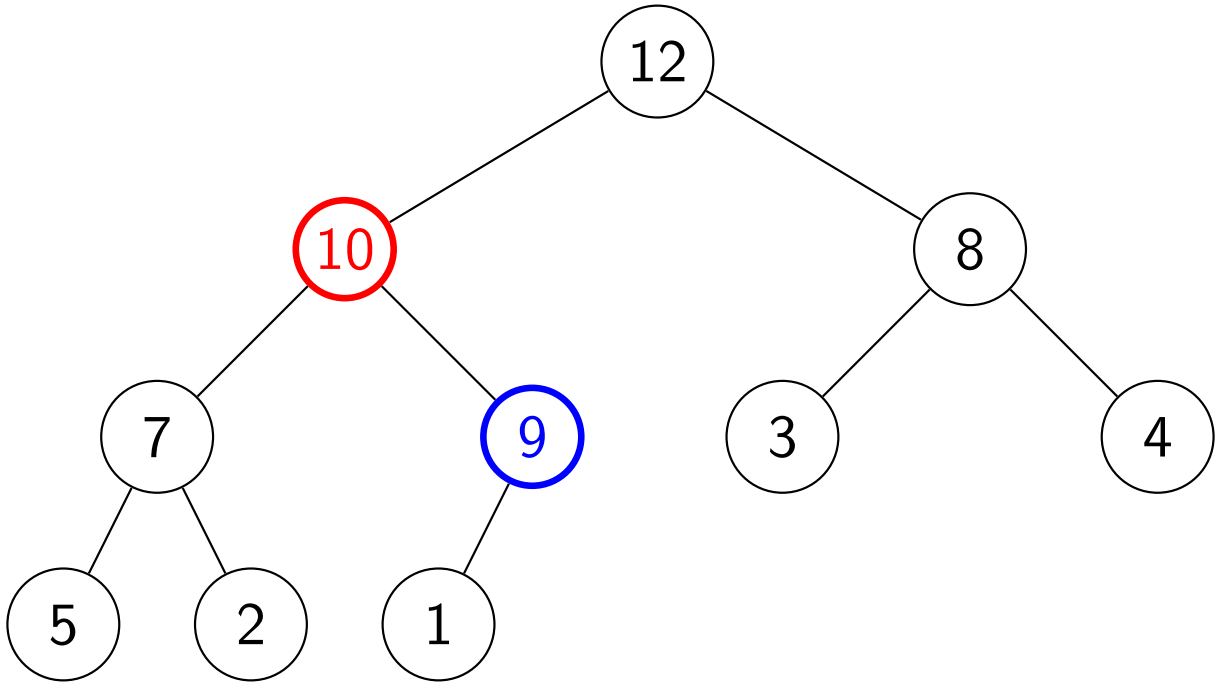
# maxHeap: inserare – exemplu



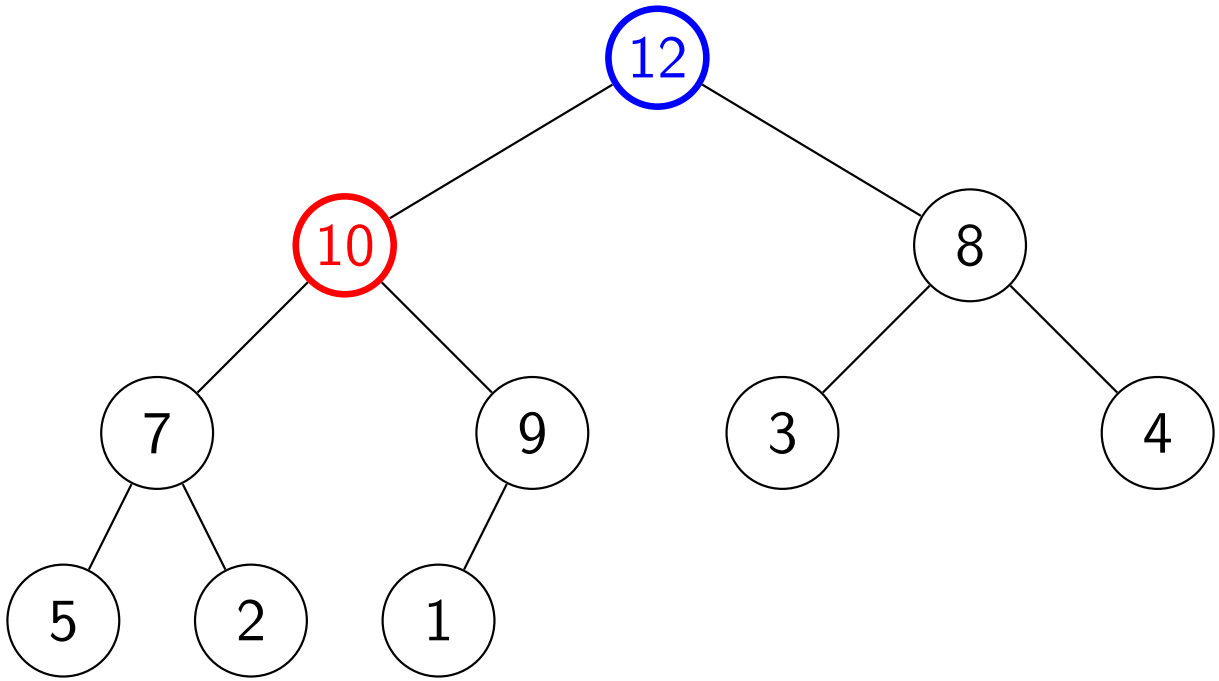
# maxHeap: inserare – exemplu



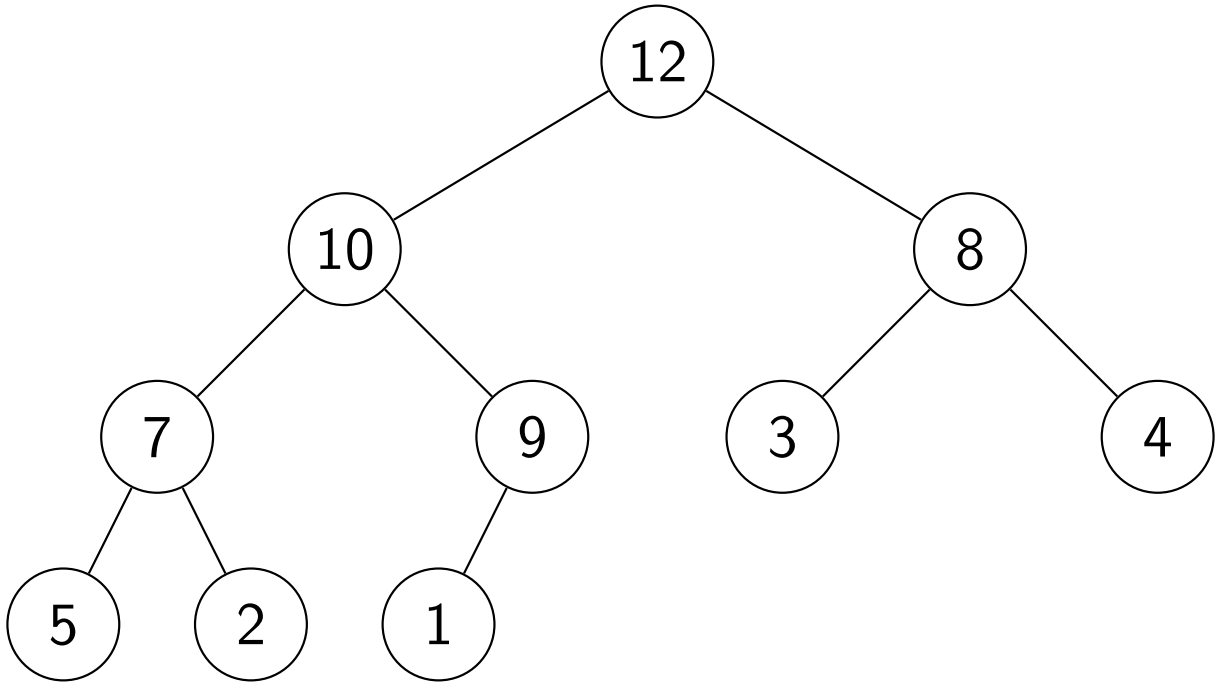
# maxHeap: inserare – exemplu



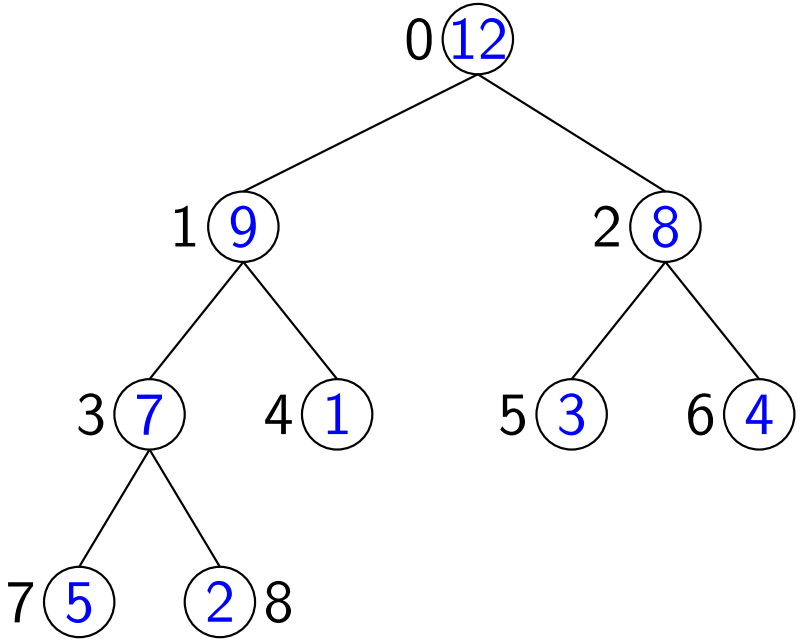
# maxHeap: inserare – exemplu



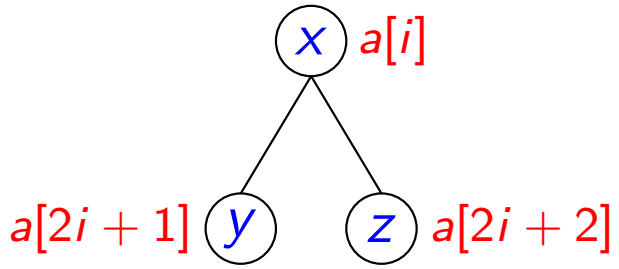
# maxHeap: inserare – exemplu



# maxHeap: implementarea cu tablouri




| 12 | 9 | 8 | 7 | 1 | 3 | 4 | 5 | 2 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |




$$\forall k : 1 \leq k \leq n-1 \Rightarrow a[k] \leq a[(k-1)/2]$$

# maxHeap: inserare

```
procedure insereaza(a, n, cheie)
begin
    n ← n+1
    a[n-1] ← cheie
    j ← n-1
    heap ← false
    while (j > 0  and not heap) do
        k ← [(j-1)/2]
        if (a[j] > a[k]) then
            swap(a[j], a[k])
            j ← k
        else
            heap ← true
end
```

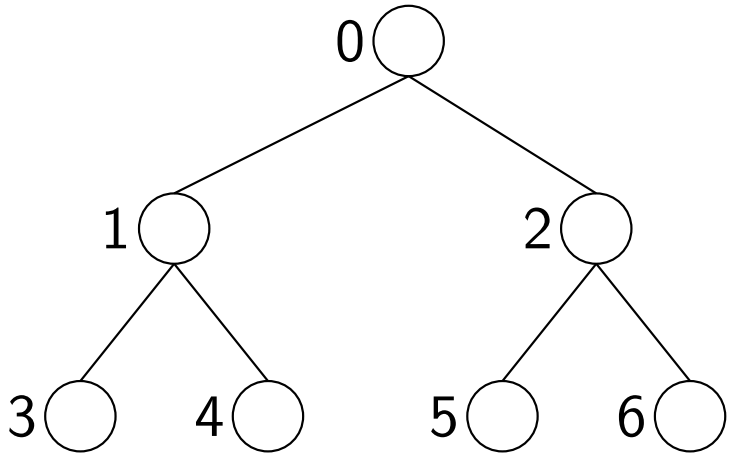
# maxHeap: elimina

```
procedure elimina(a, n)
begin
    a[0] ← a[n-1]
    n ← n-1
    j ← 0
    heap ← false
    while (2*j+1 < n  and not heap) do
        k ← 2*j+1
        if (k < n-1  and a[k] < a[k+1]) then
            k ← k+1
        if (a[j] < a[k]) then
            swap(a[j], a[k])
            j ← k
        else
            heap ← true
end
```

# maxHeap: timp de execuție

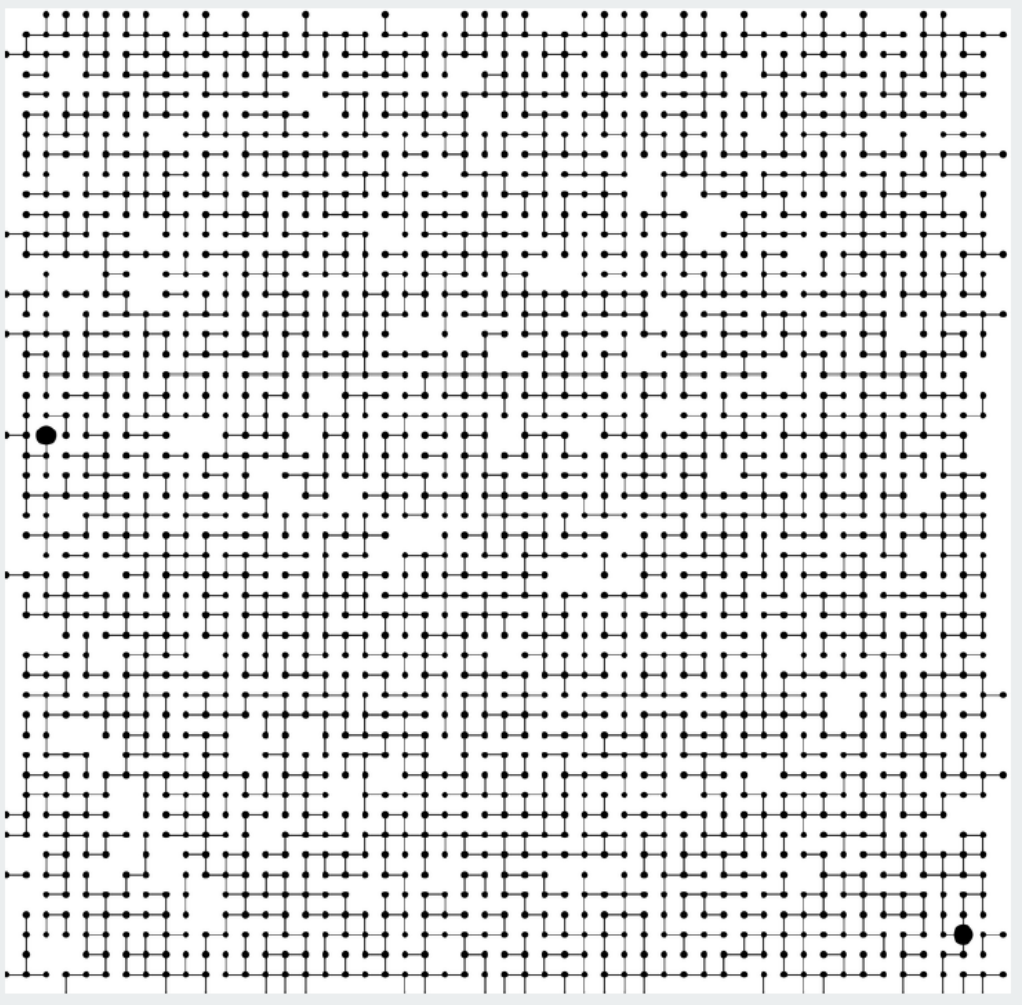
- Operațiile de `inserare` / `eliminare` au clasa de complexitate

$$O(h) = O(\log n)$$



# Conținut

# Colecții de mulțimi disjuncte



Aplicații:

- Rețele de calculatoare
- Pagini web (Internet)
- Pixeli într-o imagine digitală

# Tipul abstract Colecţii de mulţimi disjuncte

- OBIECTE:
  Colecții de submulțimi disjuncte (partiții) ale unei mulțimi univers.
- OPERAŢII:
  - `find`
    - intrare: o colecție $C$ și un element $i$ din mulțimea univers;
    - ieșire: submulțimea din $C$ căreia aparține $i$.
  - `union`
    - intrare: o colecție $C$ și două elemente $i$ și $j$ din mulțimea univers;
    - ieșire: $C$ în care s-au reunit componentele lui $i$ și $j$.
  - `singleton`
    - intrare: o colecție $C$ și un element $i$ din mulțimea univers;
    - ieșire: $C$ la care componenta lui $i$ are pe $i$ ca unic element.

# Colecții de mulțimi disjuncte: `union-find`

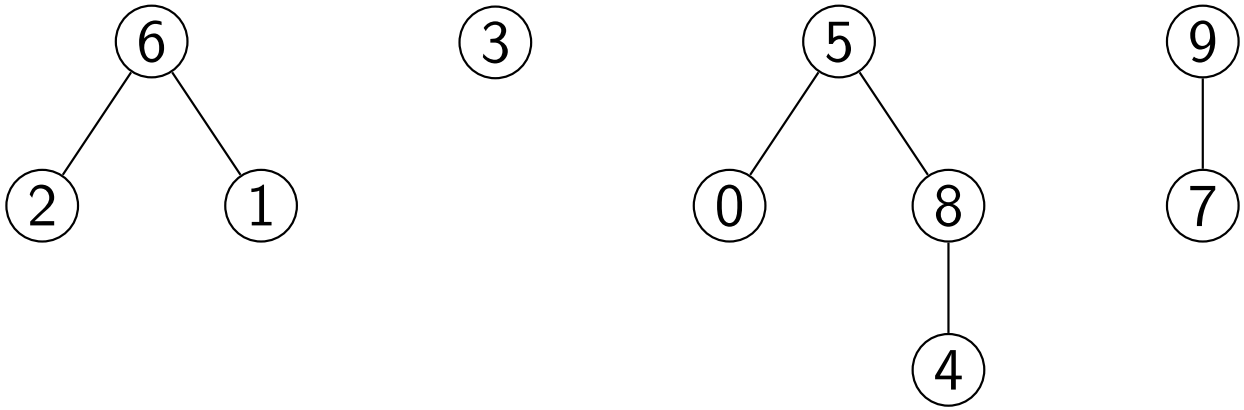
- Structura `union-find`:
  - mulțimea univers este $\{0, 1, \cdots, n-1\}$;
  - o submulțime este reprezentată printr-un arbore;
  - o colecție (partiție) este o colecție de arbori ("pădure");
  - reprezentarea unei "păduri" se face prin relația "părinte".

# union-find: exemplu

- $n = 10, \quad C = \{\{1, 2, 6\}, \{3\}, \{0, 4, 5, 8\}, \{7, 9\}\}$

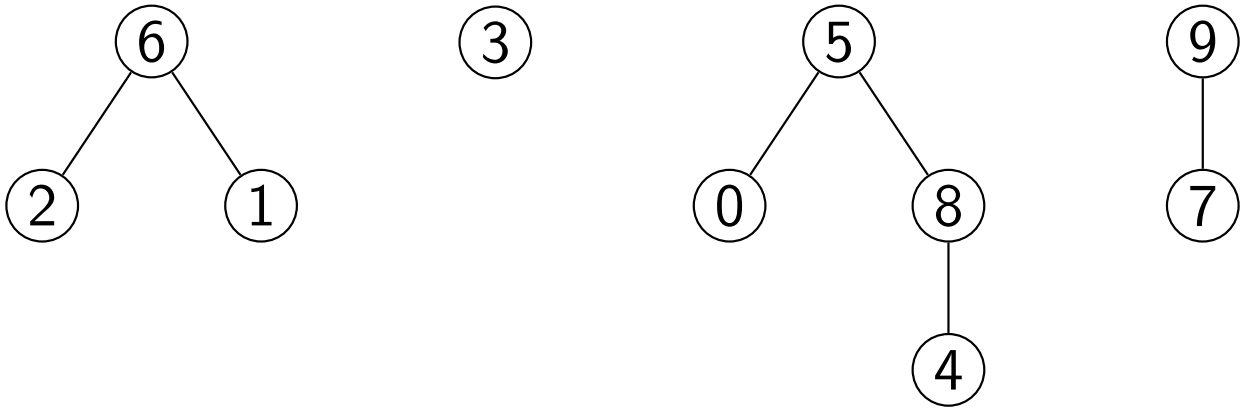
# union-find: exemplu

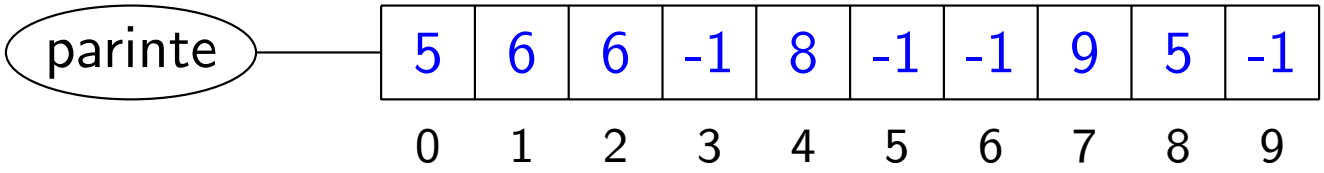
- $n = 10, \quad C = \{\{1, 2, 6\}, \{3\}, \{0, 4, 5, 8\}, \{7, 9\}\}$



# union-find: exemplu

- $n = 10, \quad C = \{\{1, 2, 6\}, \{3\}, \{0, 4, 5, 8\}, \{7, 9\}\}$



# union-find: singleton

**procedure** *singleton(*C, i*)*
**begin**
    C.parinte[i] $\leftarrow$ -1
**end**

# union-find: find

```
procedure find(C, i)
begin
    temp ← i
    while (C.parinte[temp] >= 0) do
        temp ← C.parinte[temp]
    return temp
end
```

# union-find: union

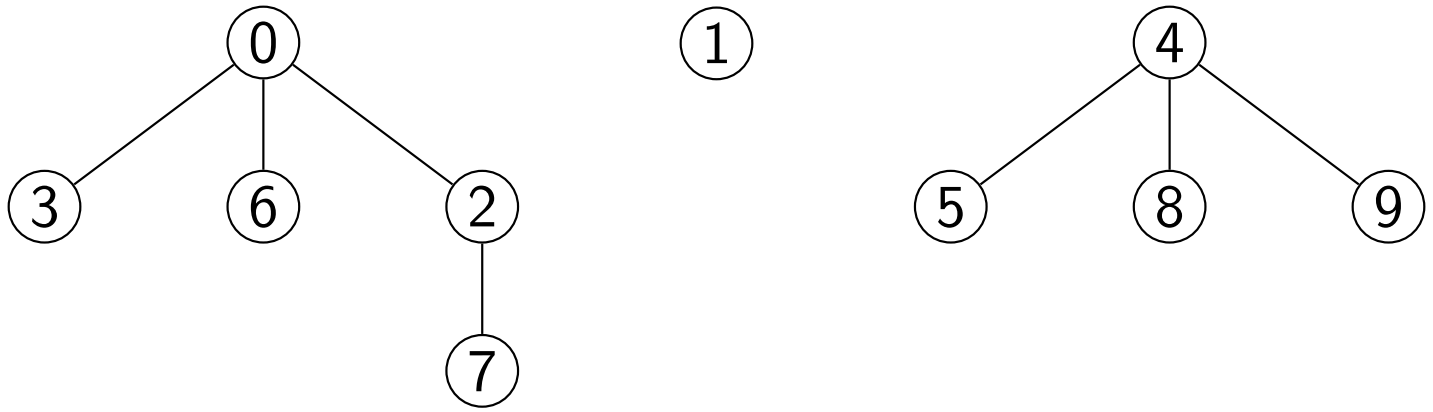
```
procedure union(C, i, j)
begin
    ri ← find(i)
    rj ← find(j)
    if ri != rj then
        C.parinte[rj] ← ri
end
```

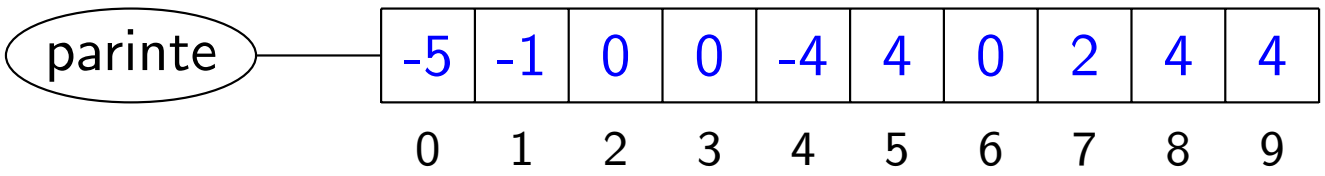
# Structură union-find ponderată

- Soluție la problema arborilor dezechilibrați.
- Mecanism:
  - Memorarea numărului de vârfuri din arbore (cu semn negativ).
  - Aplatizarea arborilor.

# Structură union-find ponderată: exemplu

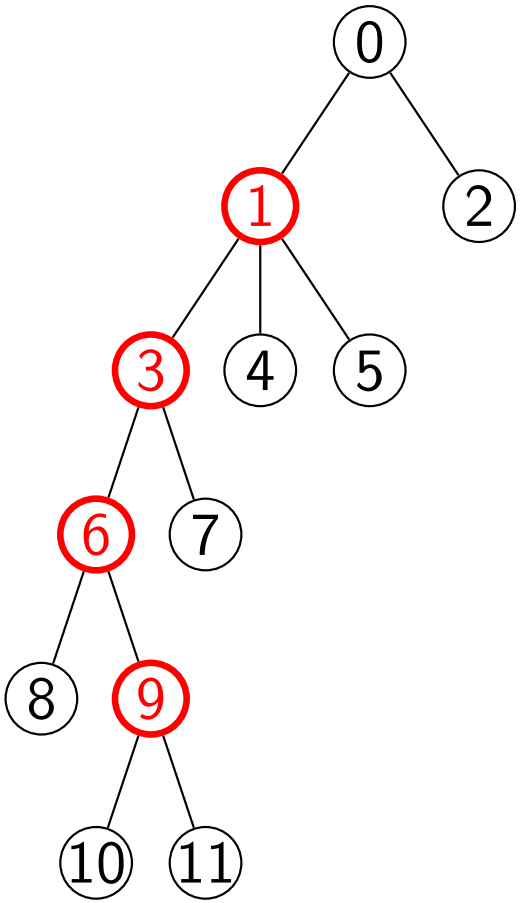
- $n = 10, \quad C = \{\{0, 2, 3, 6, 7\}, \{1\}, \{4, 5, 8, 9\}\}$

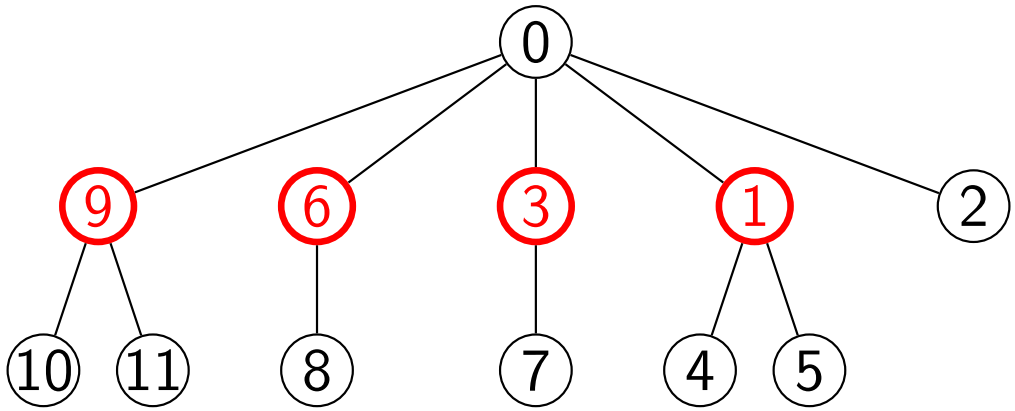


# Aplatizarea arborilor – exemplu

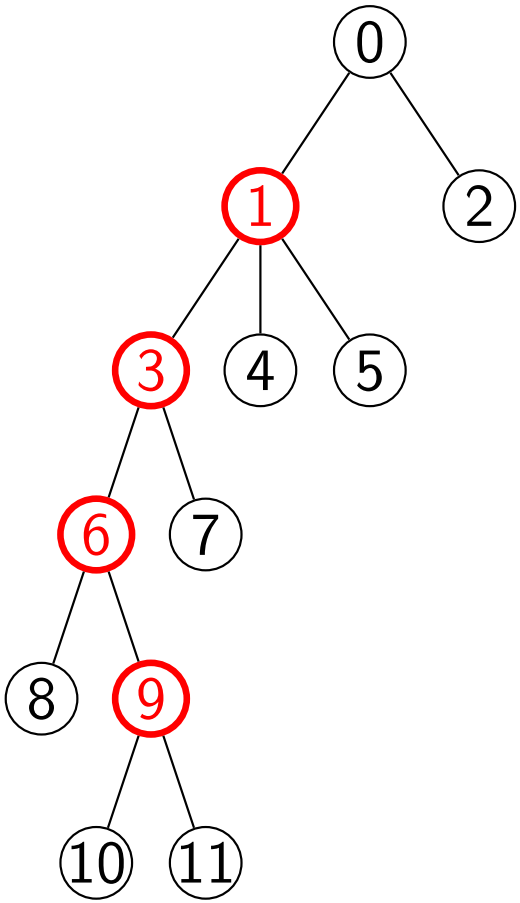
- `find(9)`

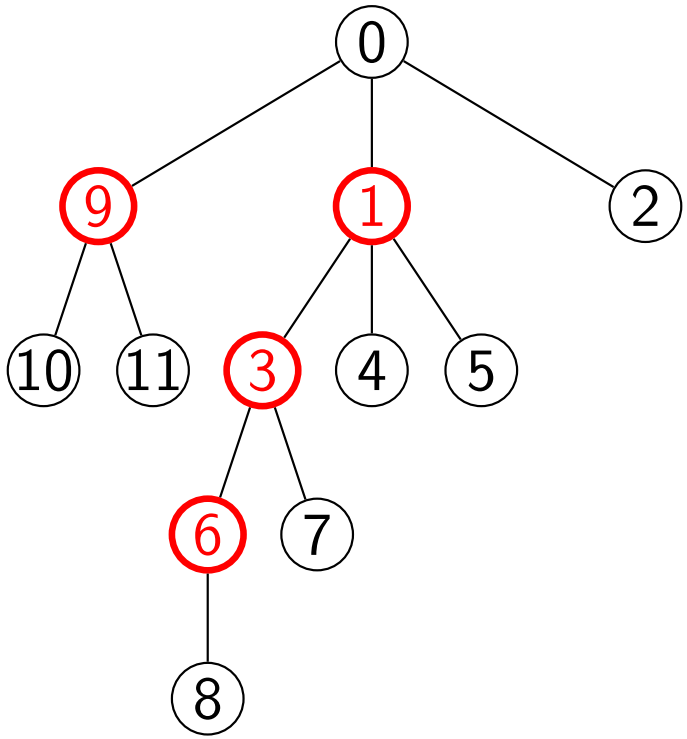


# Aplatizarea arborilor – exemplu

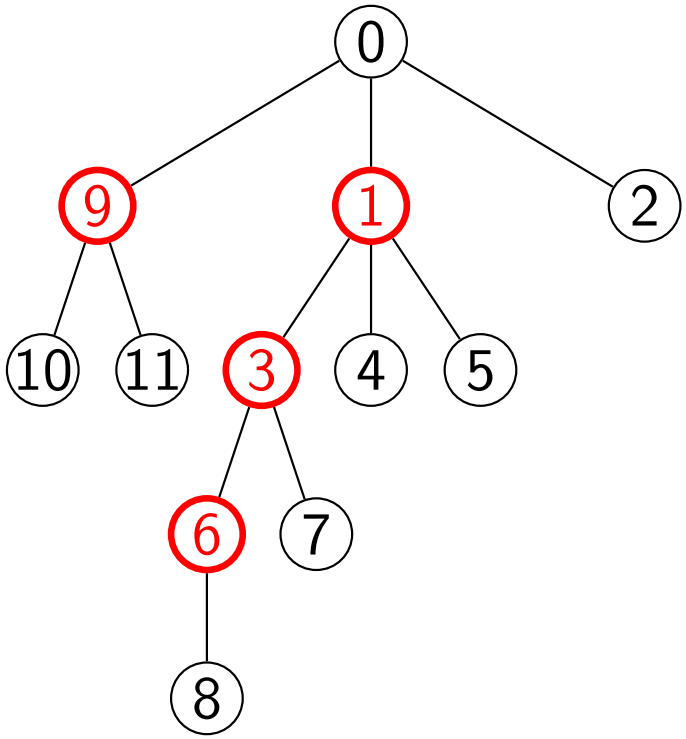
- `find(9)`

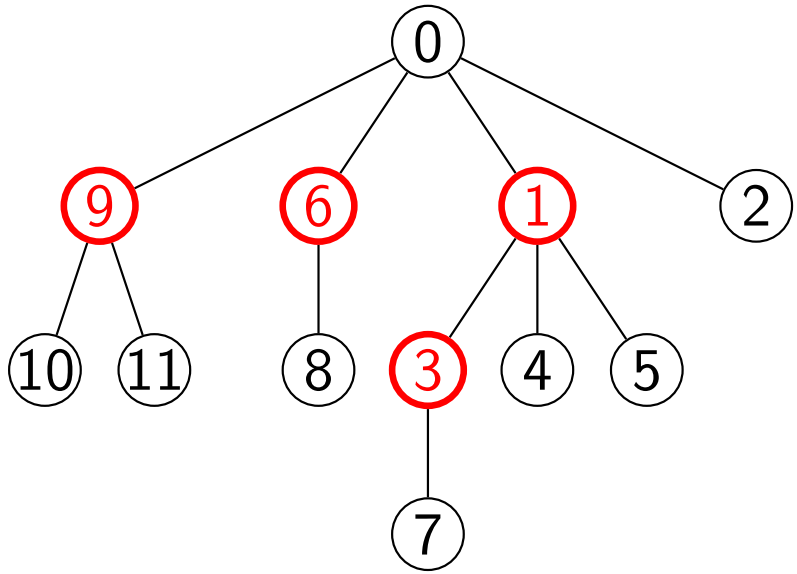


# Aplatizarea arborilor – exemplu

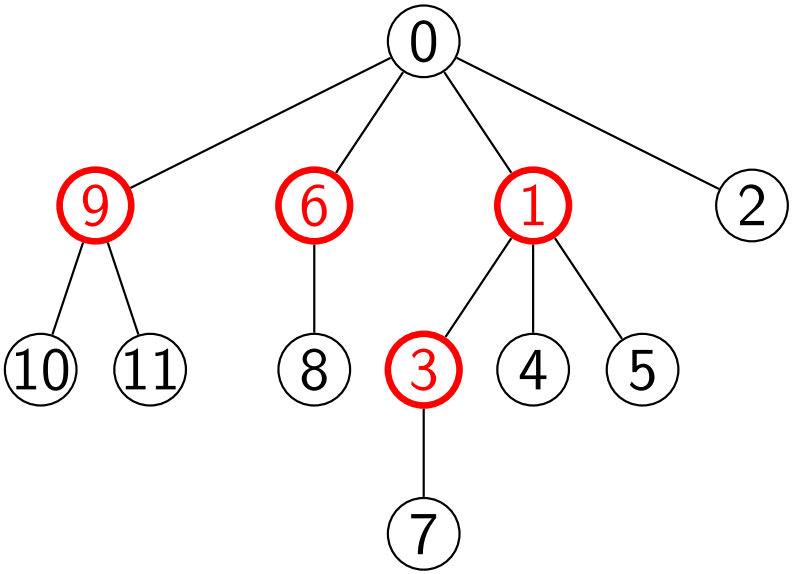
- `find(9)`

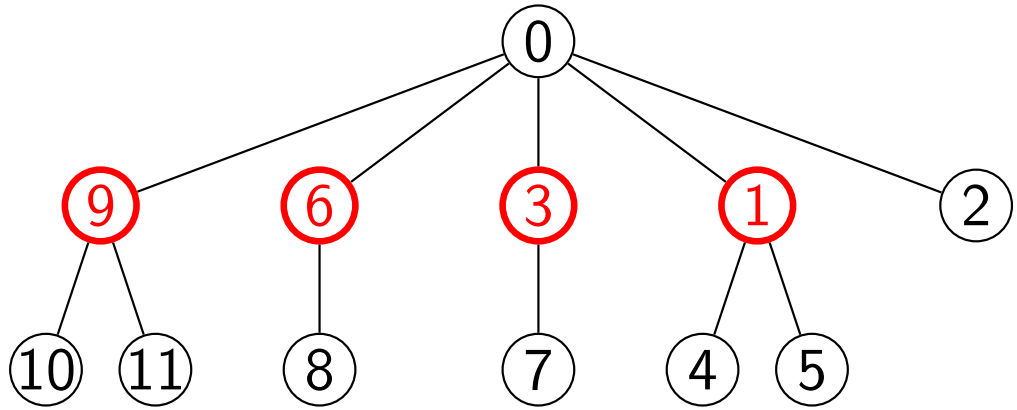


# Aplatizarea arborilor – exemplu

- `find(9)`



# Structură union-find ponderată

```
procedure union(C, i, j)
begin
    ri ← find(i);    rj ← find(j)

    while (C.parinte[i] >= 0) do
        temp ← i;    i ← C.parinte[i];    C.parinte[temp] ← ri
    while (C.parinte[j] >= 0) do
        temp ← j;    j ← C.parinte[j];    C.parinte[temp] ← rj

    if C.parinte[ri] > C.parinte[rj] then
        C.parinte[rj] ← C.parinte[ri] + C.parinte[rj]
        C.parinte[ri] ← rj
    else
        C.parinte[ri] ← C.parinte[ri] + C.parinte[rj]
        C.parinte[rj] ← ri
end
```

# Structură `union-find` ponderată

**Teoremă**

*Pornind de la o colecție vidă, orice secvență de $m$ operații* `union` *și* `find` *asupra $n$ elemente are complexitatea $O(n + m \log^* n)$.*

Observatie: $\log^* n$ este numărul de logaritmări până se obține valoarea 1.