

HEAPURI

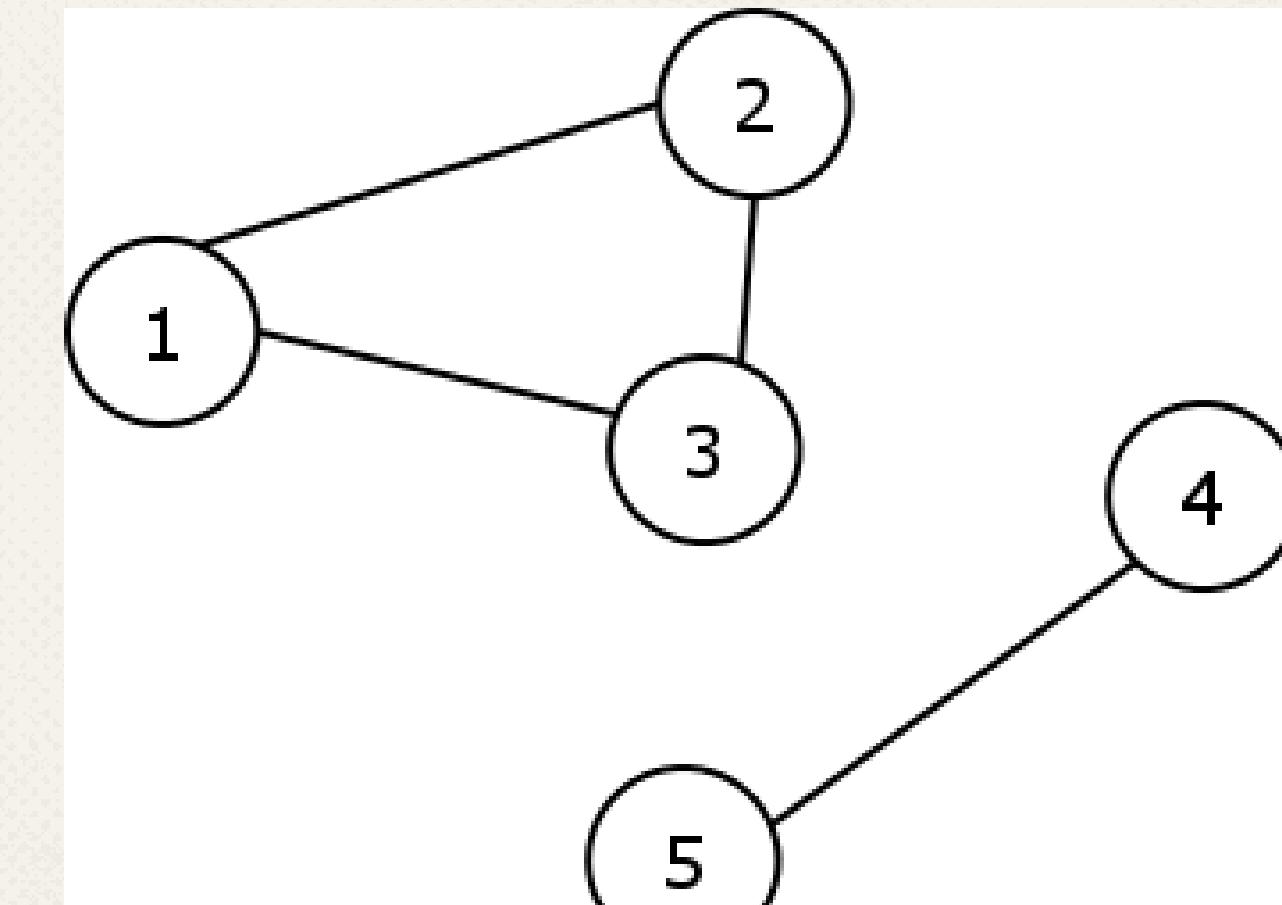


Heapuri

- 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 multimea de noduri (vertex / vertices),
 - E este multimea de muchii

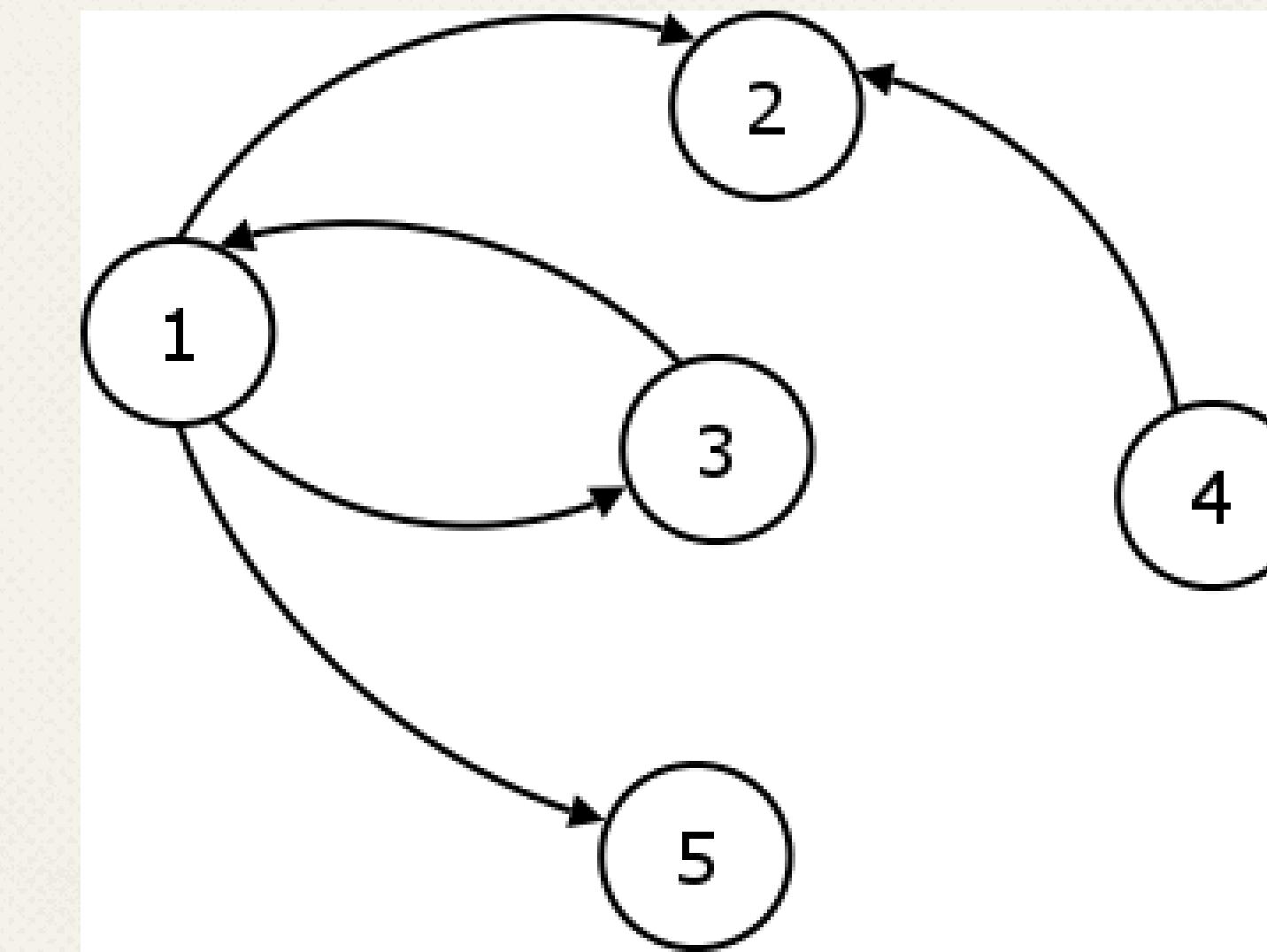
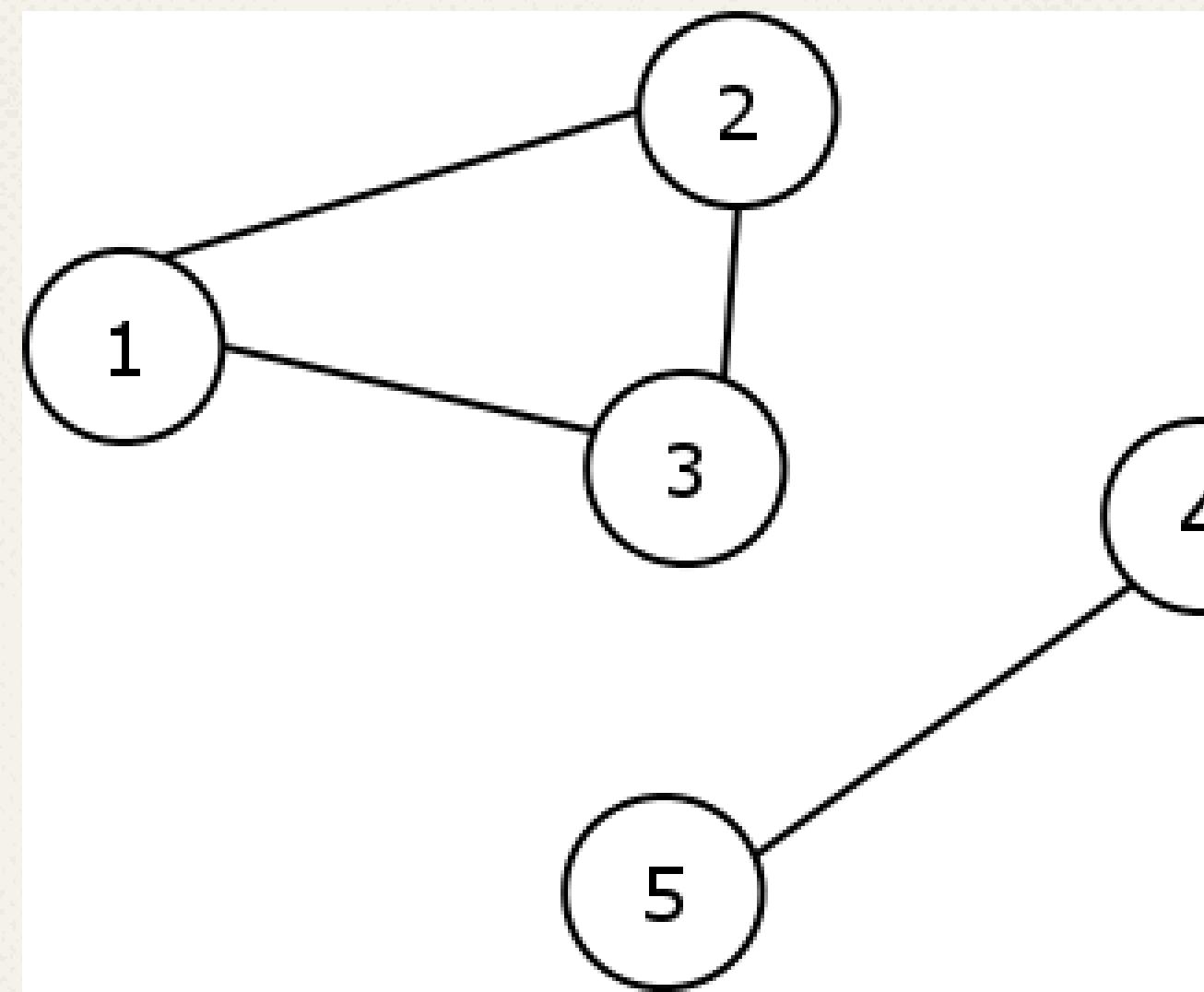


Grafuri

- Graf neorientat

vs

- graf orientat



Arbore

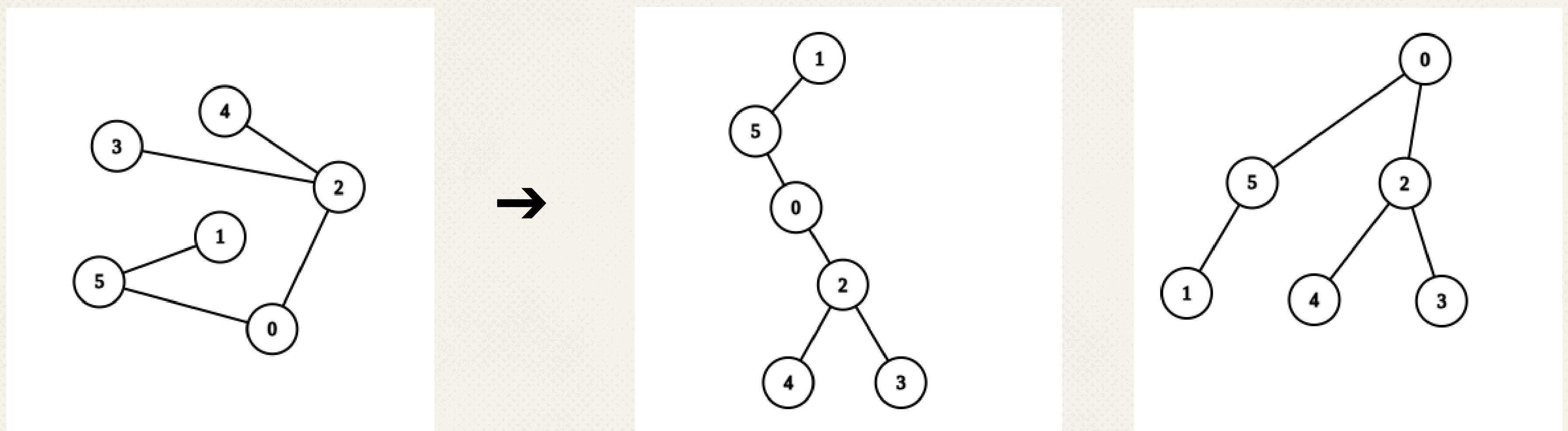
- Definiții:
 - Un arbore este un graf conex aciclic
 - Un arbore este un graf aciclic maximal
 - Un arbore este un graf conex minimal
 - Un arbore este un graf aciclic cu **n-1** muchii
 - Un arbore este un graf conex cu **n-1** muchii
 - ...
 - Într-un arbore există un singur drum simplu între oricare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

Arbore

- 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)

Arbore

- 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



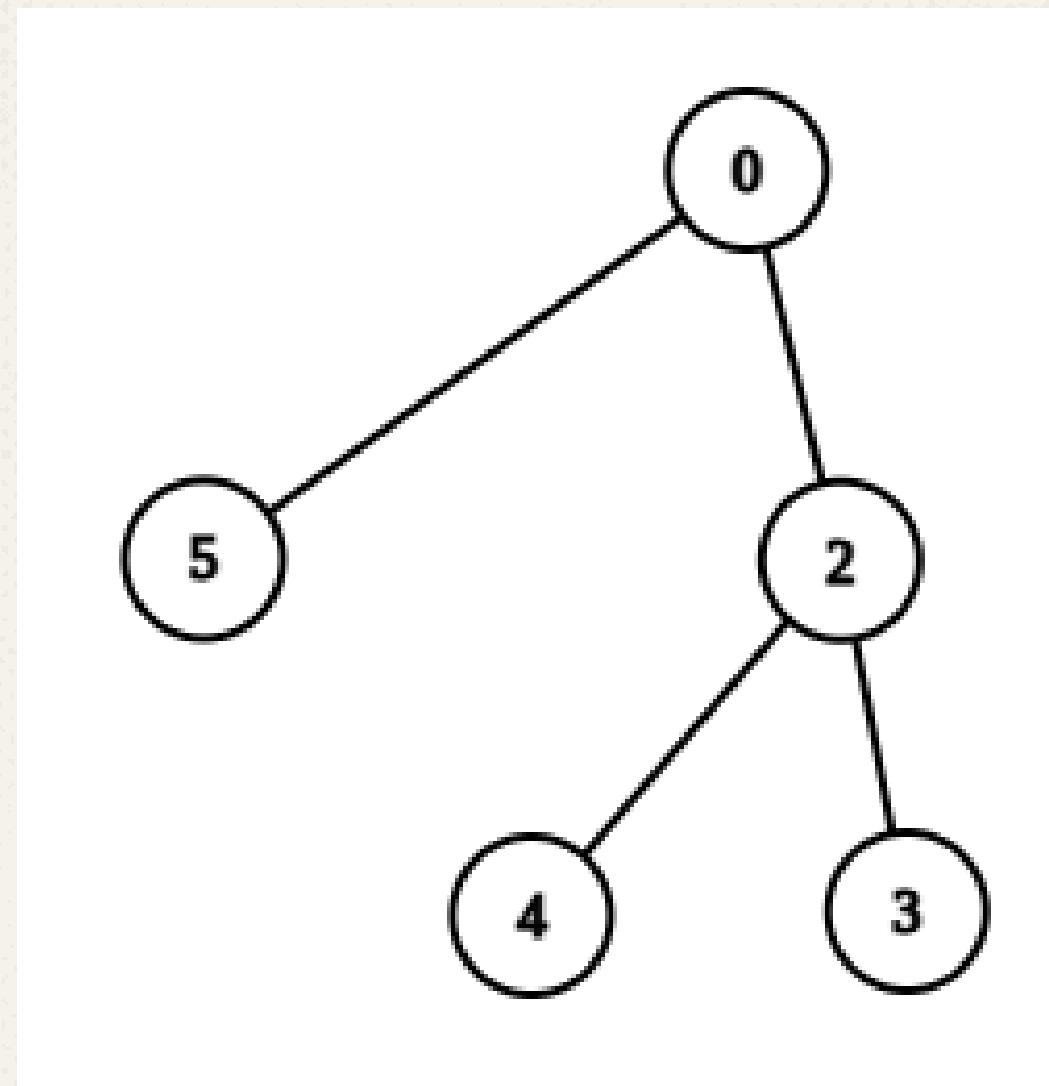
Arbore binari

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

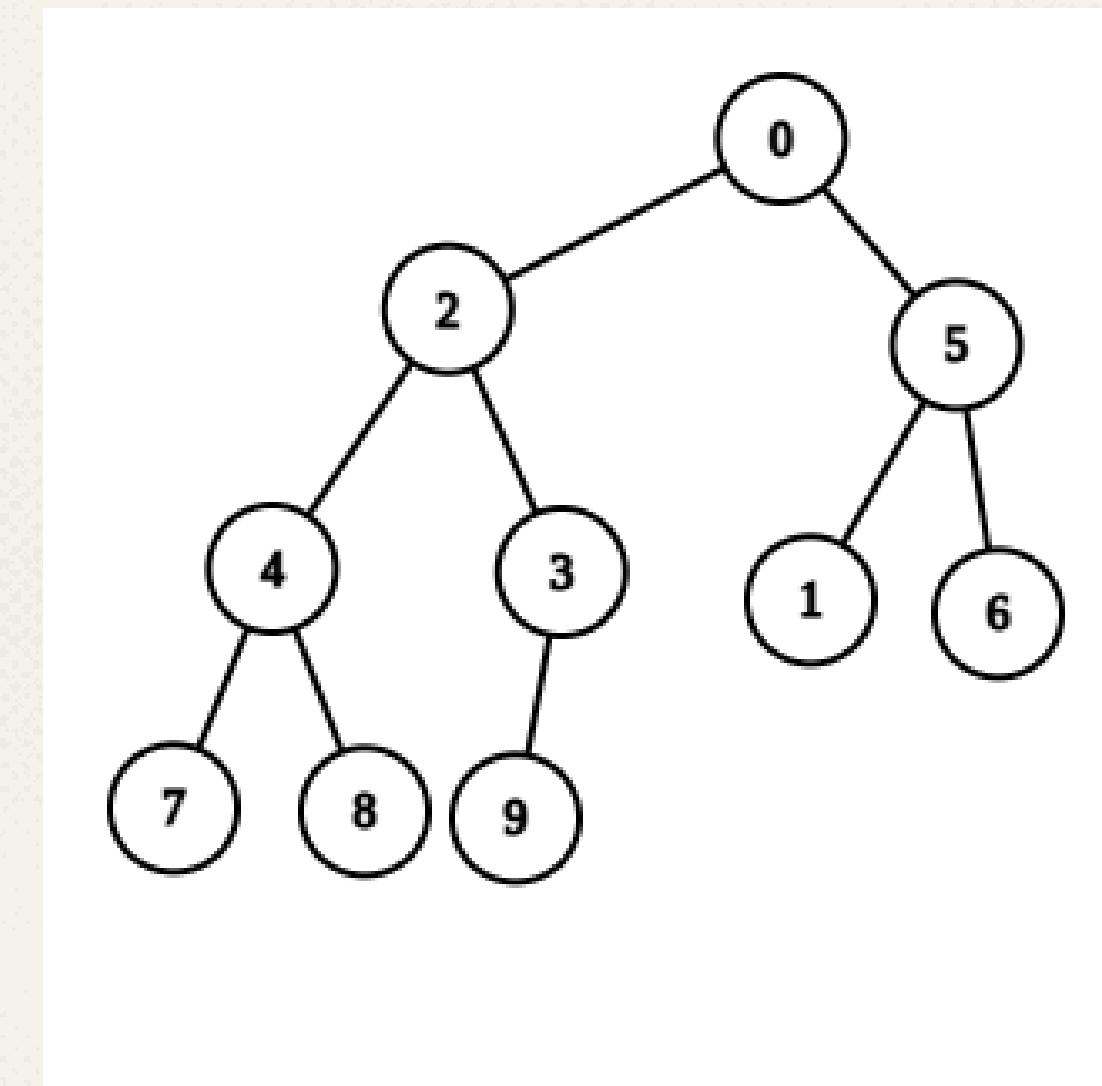
Copiii unui nod sunt numiți copilul stâng (Left, L) și copilul drept (Right, R).

Arborei 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

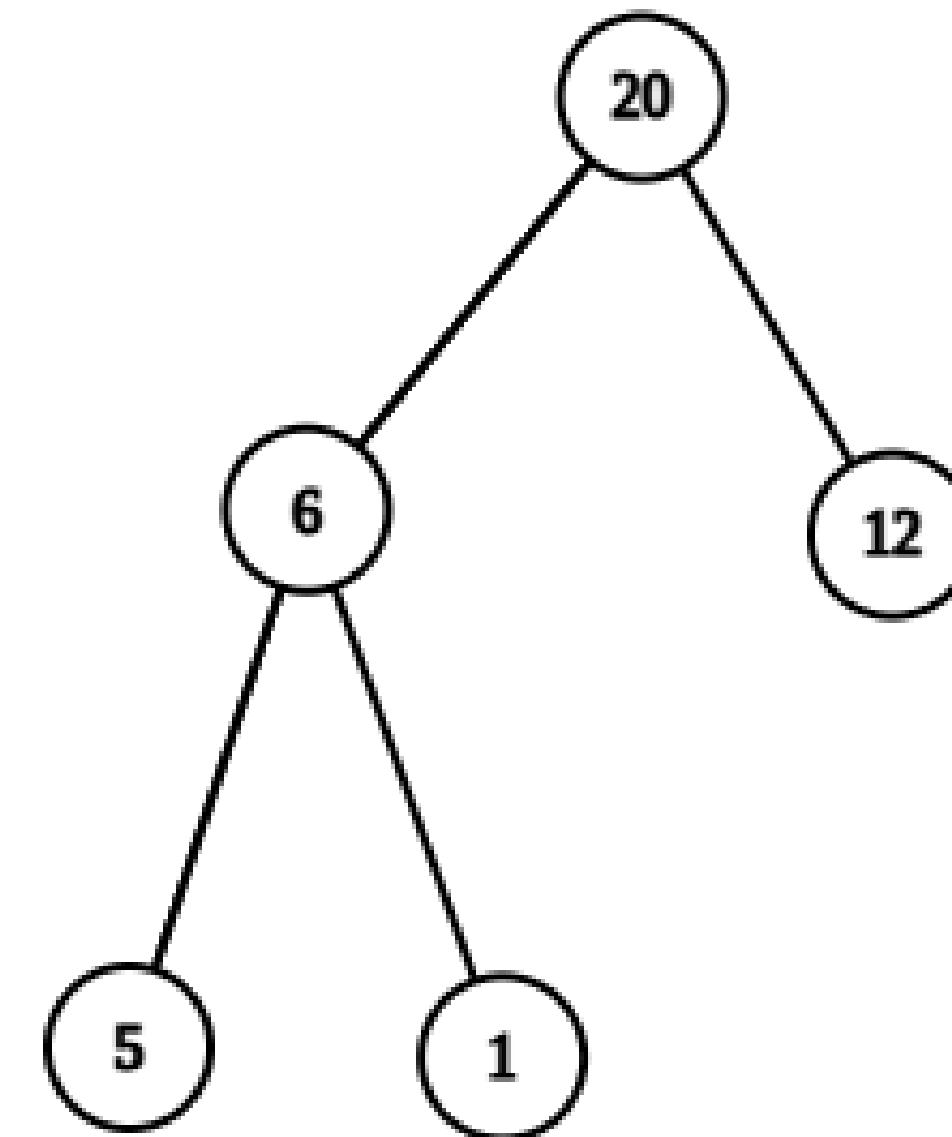


Arbore binari - Proprietăți

- Exercițiu:
 - Numărul de noduri ale unui arbore binar cu înălțimea h este între (?) și (?)
 - h (dacă este lant)
 - $2^{h+1} - 1$
 - 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

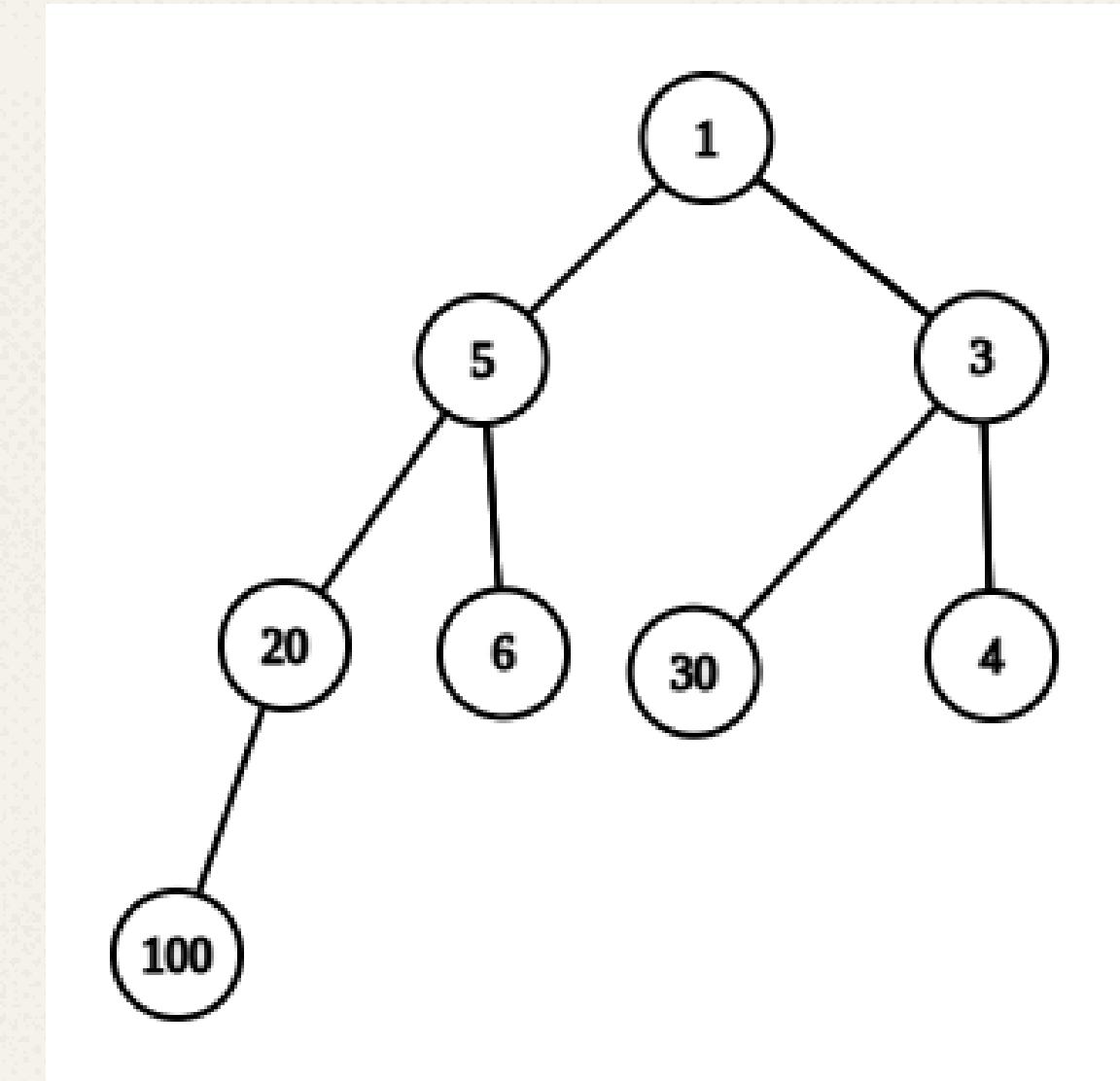
Heapuri

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



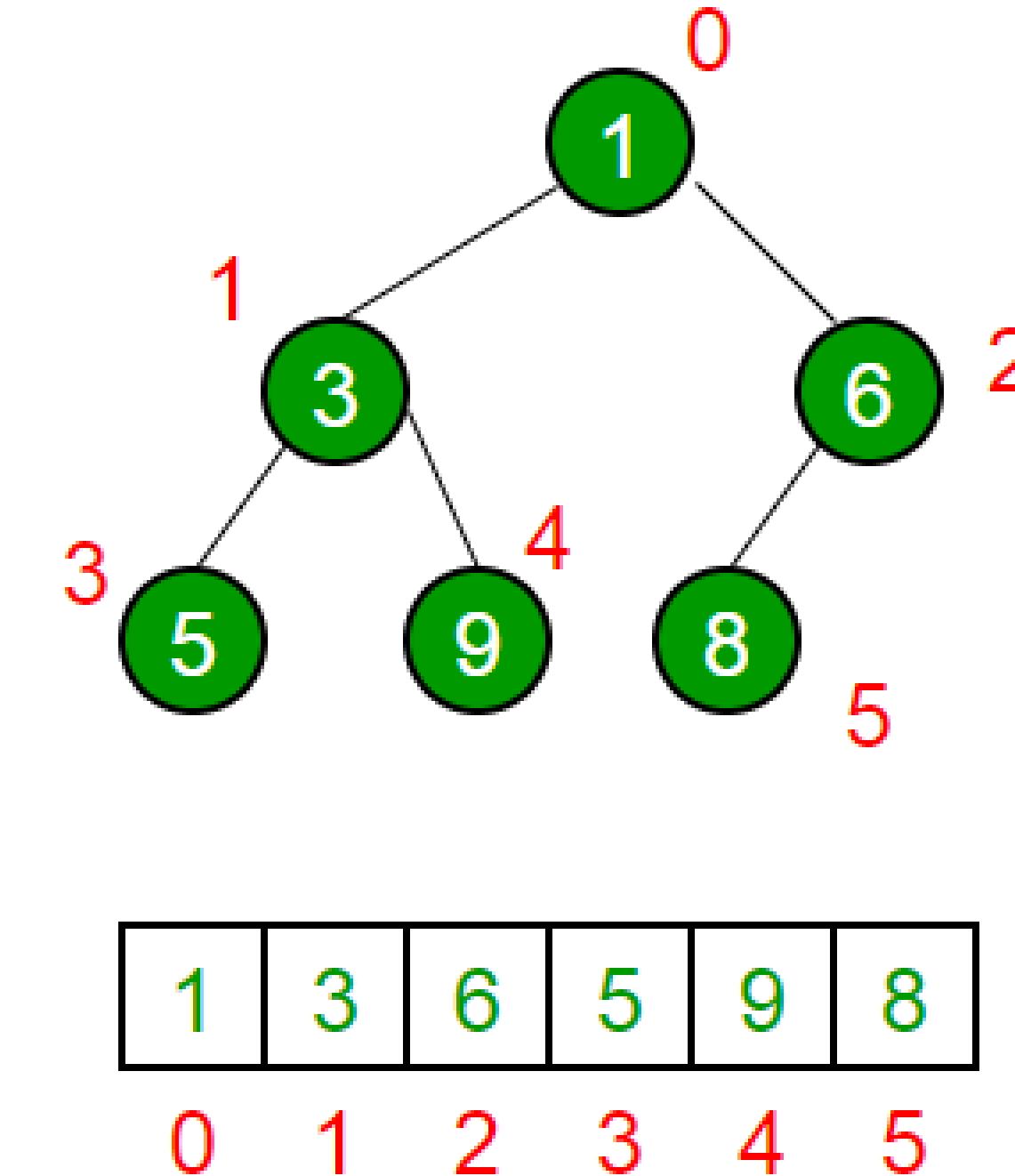
Heapuri

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

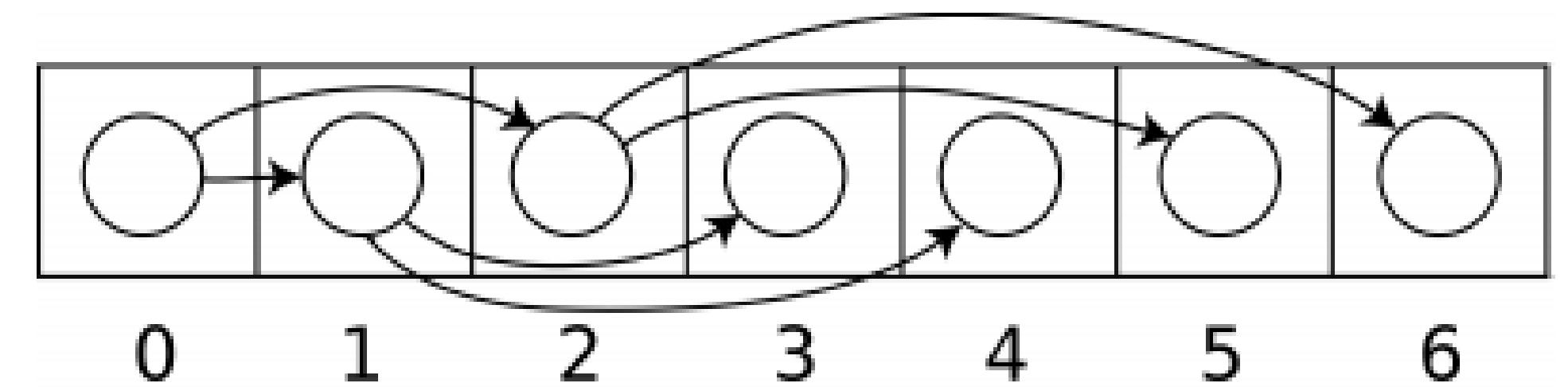
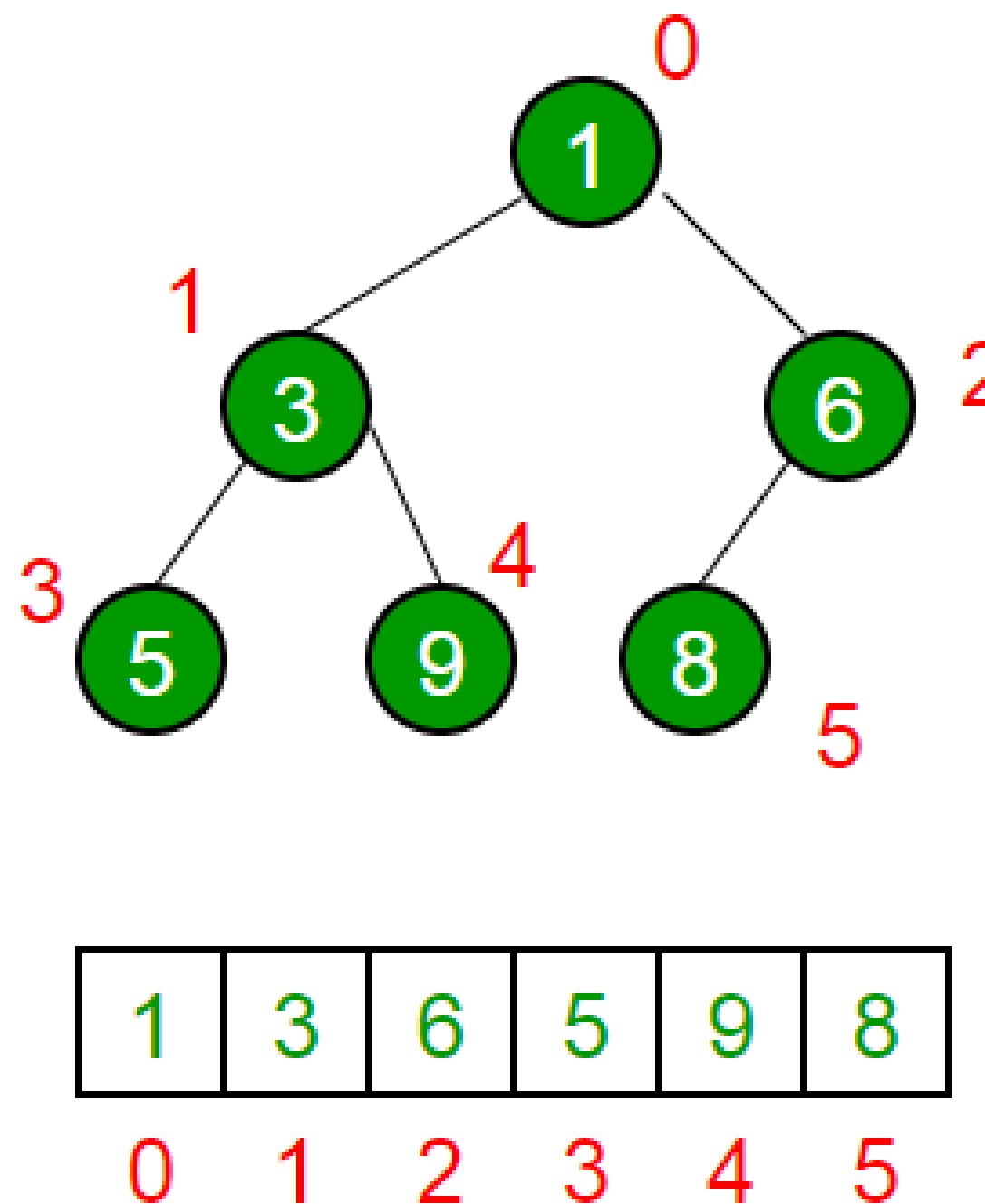


Heapuri - Reprezentare

Un arbore binar complet poate fi reprezentat ca un vector!



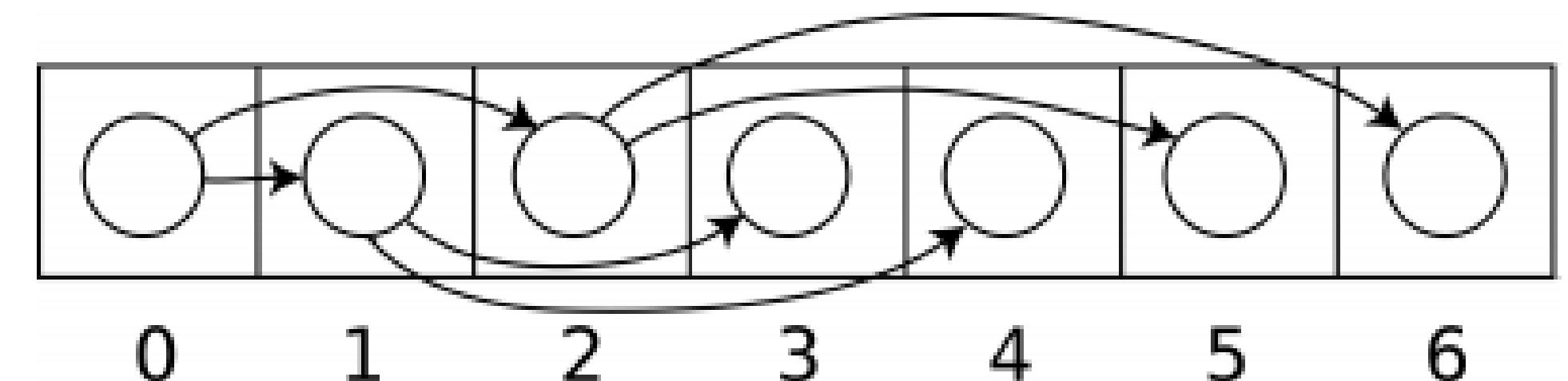
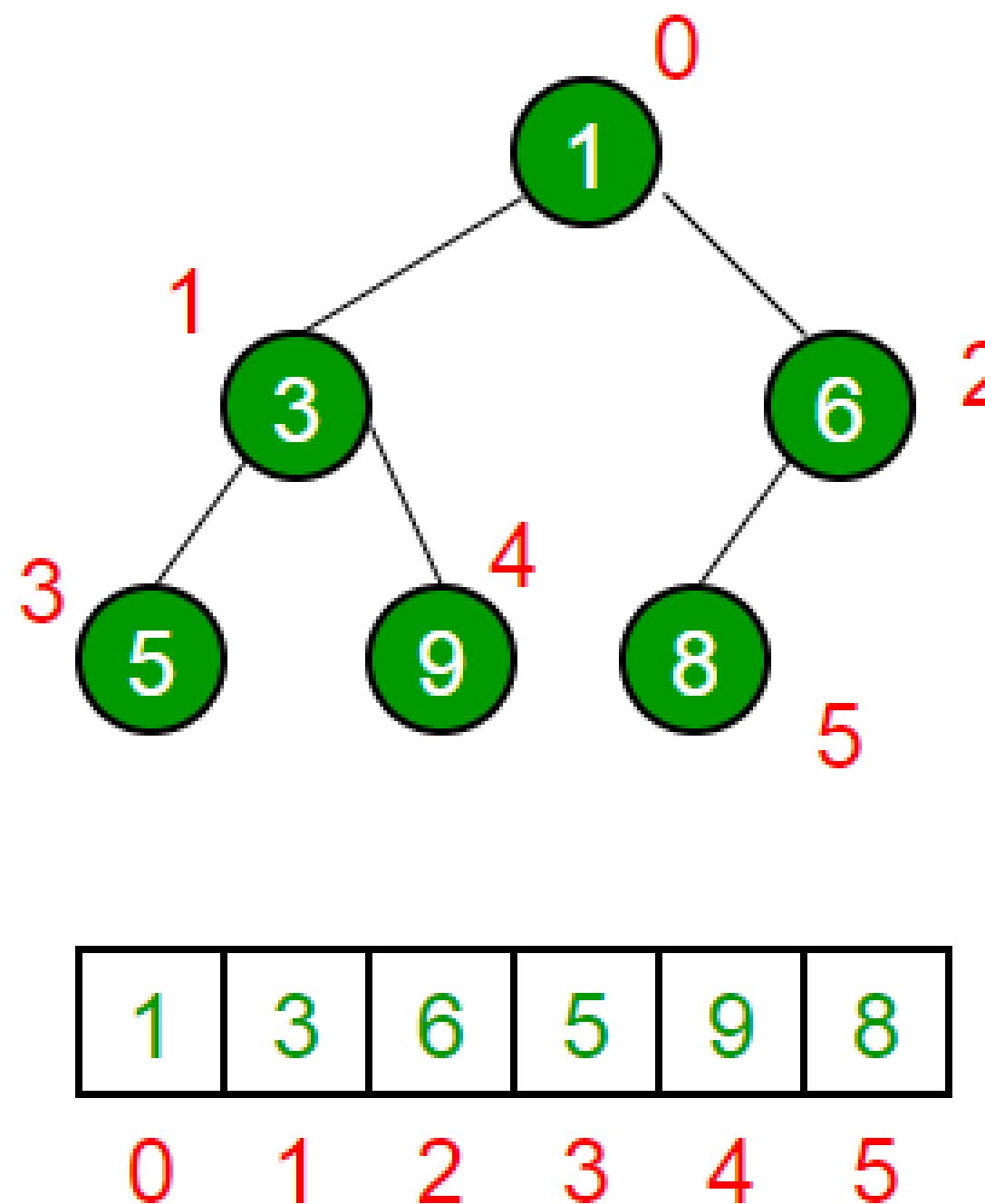
Heapuri - Reprezentare



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

Înălțime?

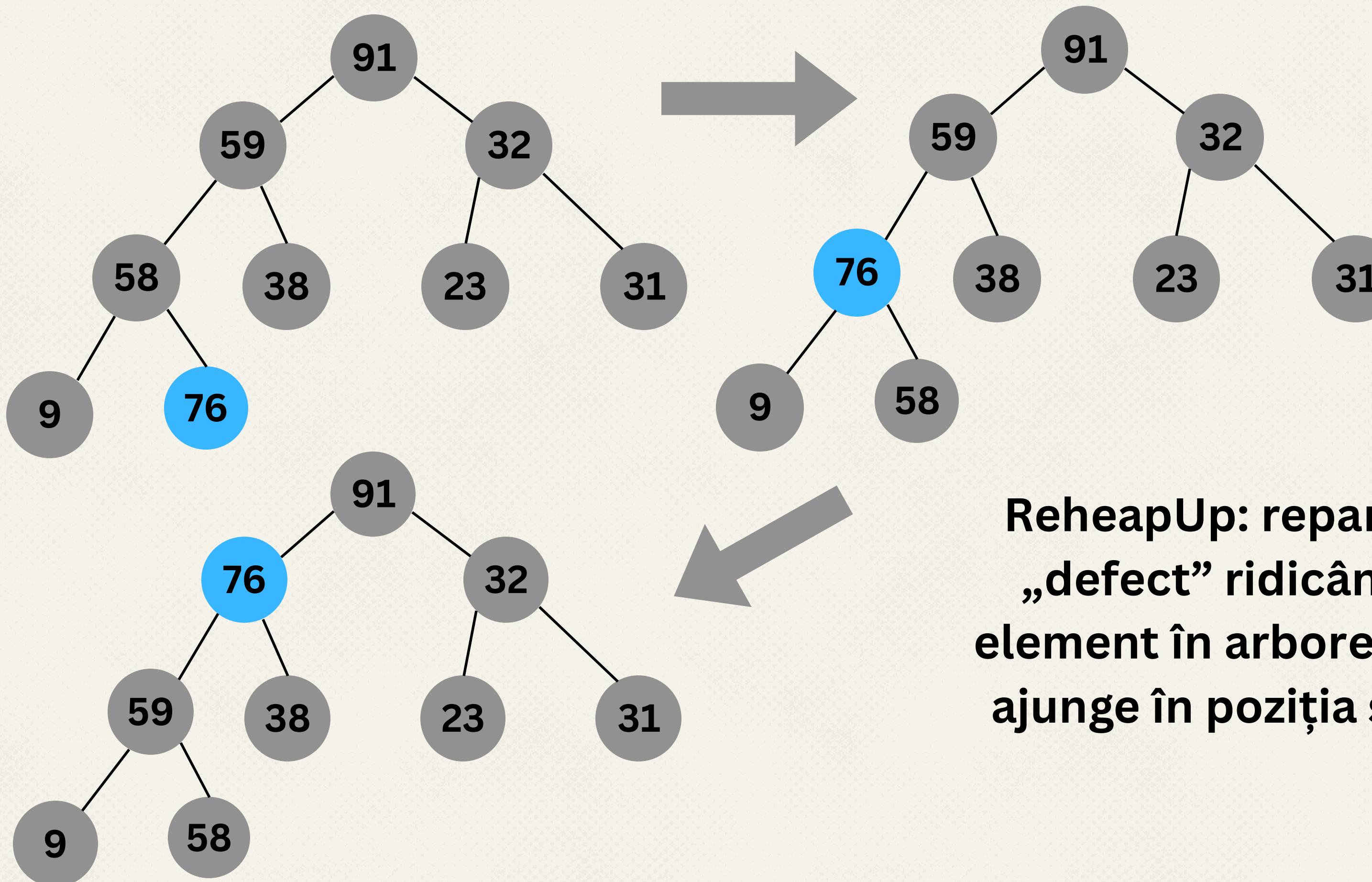
Heapuri - Reprezentare



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

Înălțime?
 $\log n$!!

Heapuri - Urcă (percolate)

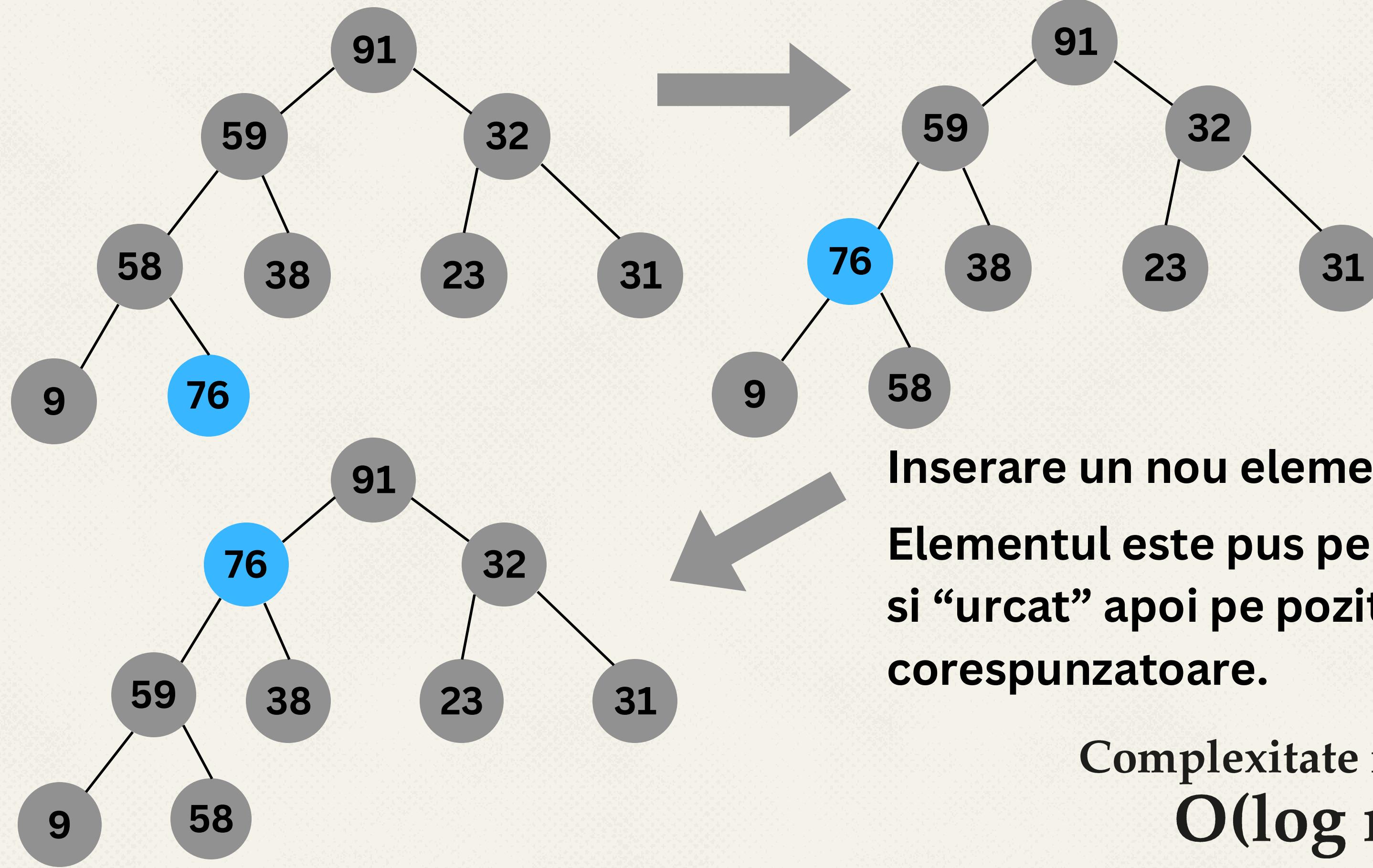


ReheapUp: repară un heap „defect” ridicând ultimul element în arbore, până când ajunge în poziția sa corectă.

Cod ‘urcă’

```
void urca(int poz) {
    while (poz) {
        // Calculează indicele părintelui nodului curent
        int tata = (poz - 1) / 2;
        // Verifică dacă valoarea nodului părinte este mai mică decât valoarea nodului
        // curent
        if (heap[tata] < heap[poz]) {
            // Dacă da, schimbă valorile nodurilor între ele (swap)
            swap(heap[tata], heap[poz]);
            // Actualizează indicele la nodul părinte
            poz = tata;
        } else {
            //iese din while dacă proprietatea heap este deja satisfăcută
            break;
        }
    }
}
```

Heapuri - Inserare



Inserare un nou element in max-heap

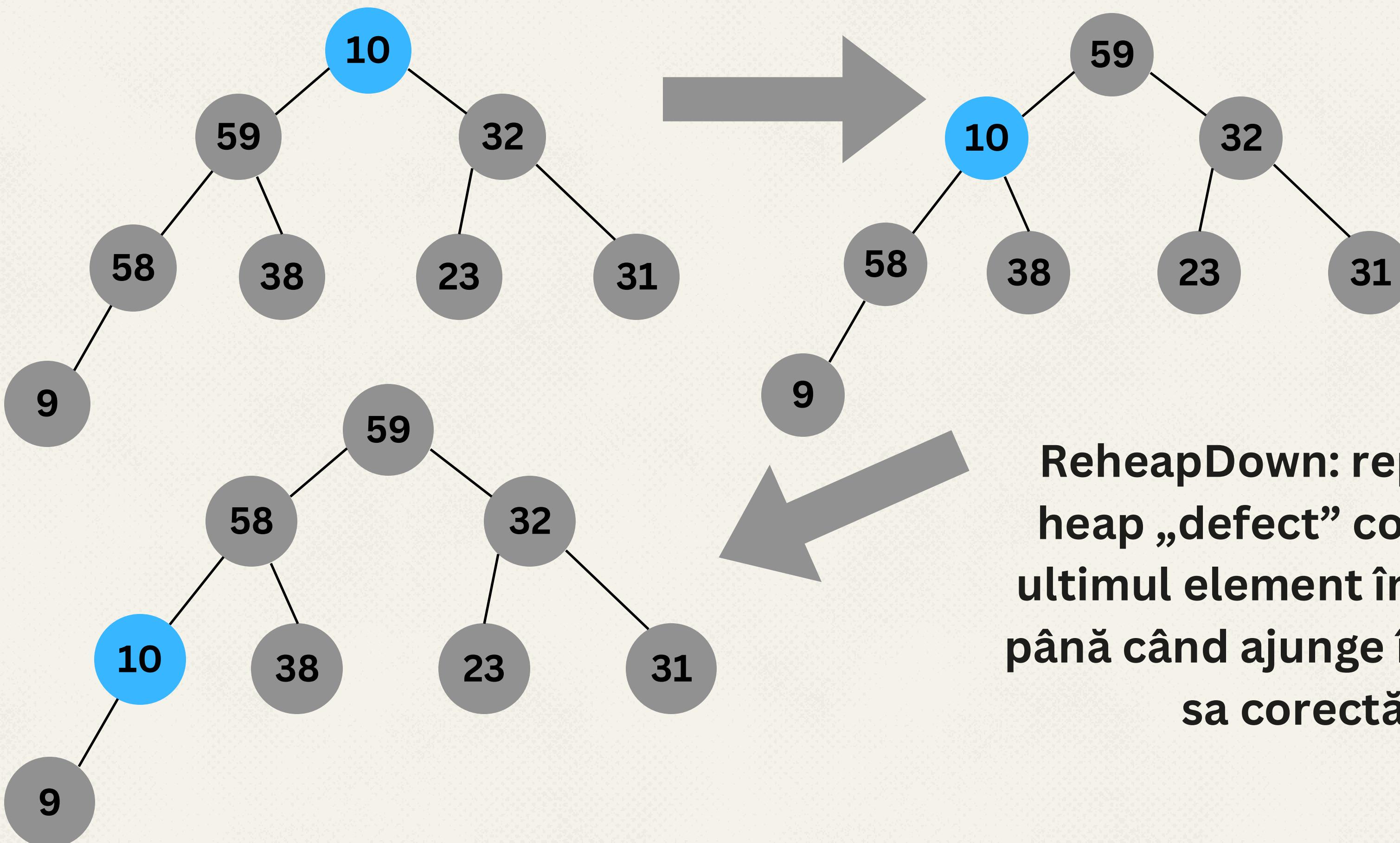
Elementul este pus pe ultima pozitie
si “urcat” apoi pe pozitia
corespunzatoare.

Complexitate inserare
 $O(\log n)$

Cod ‘inserare’

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

Heapuri - Coboară (sift)



ReheapDown: repară un heap „defect” coborând ultimul element în arbore, până când ajunge în poziția sa corectă.

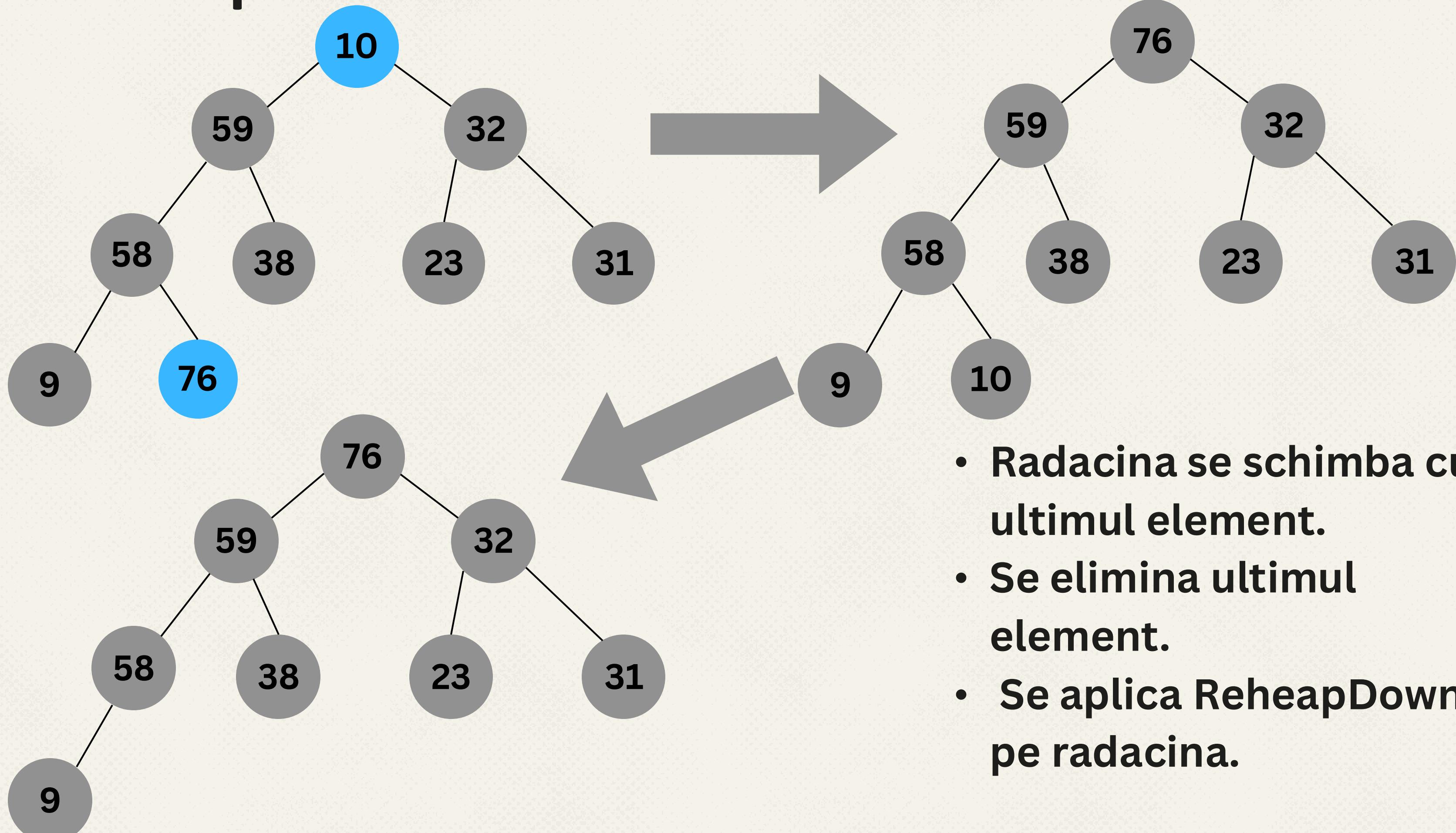
Cod ‘coboară’ - partea 1

```
void coboara(int poz) {  
    if (poz * 2 + 1 >= heap.size()) return; // Verifică dacă indicele 'poz' indică un nod  
    care nu are copii  
    int fiu_st = heap[poz * 2 + 1]; // Calculează indicele fiului stâng al nodului curent  
    // Verifică dacă nodul curent are doar un fiu sau dacă fiul stâng este mai mare decât  
    fiul drept  
    if ((poz * 2 + 2 == heap.size()) || fiu_st > heap[poz * 2 + 2]) {  
        // Verifică dacă fiul stâng este mai mare decât nodul curent  
        if (fiu_st > heap[poz]) {  
            //interschimbă valorile nodului curent cu cele ale fiului stâng  
            swap(heap[poz], heap[poz * 2 + 1]);  
            // se continuă coborârea în arbore  
            coboara(poz * 2 + 1);  
            return;  
        } else {  
            return;  
        }  
    }  
}
```

Cod ‘coboară’ - partea 2

```
else {
    // Verifică dacă fiul drept este mai mare decât nodul curent
    if (heap[poz * 2 + 2] > heap[poz]) {
        //interschimbă valorile nodului curent cu cele ale fiului drept
        swap(heap[poz], heap[poz * 2 + 2]);
        // se continuă coborârea în arbore
        coboara(poz * 2 + 2);
        return;
    } else {
        // În caz contrar, oprește recursivitatea
        return;
    }
}
```

Heapuri - Elimină radacina



- Radacina se schimba cu ultimul element.
- Se elimina ultimul element.
- Se aplica ReheapDown pe radacina.

Pop cod

```
int pop() {  
    if (heap.size() == 0)  
        return -1;  
  
    int vf = heap[0]; // Salvăm valoarea rădăcinii (valoarea maximă)  
    // Suprascriem rădăcina cu ultimul element din heap  
    heap[0] = heap[heap.size() - 1];  
    // Eliminăm ultimul element din heap (elementul care a fost mutat în locul  
    rădăcinii)  
  
    heap.pop_back();  
    // Reechilibrăm heap-ul, coborând noua rădăcină pentru a respecta  
    proprietatea de heap  
  
    coboara(0);  
    // Returnăm valoarea maximă care a fost extrasă din heap  
  
    return vf;  
}
```

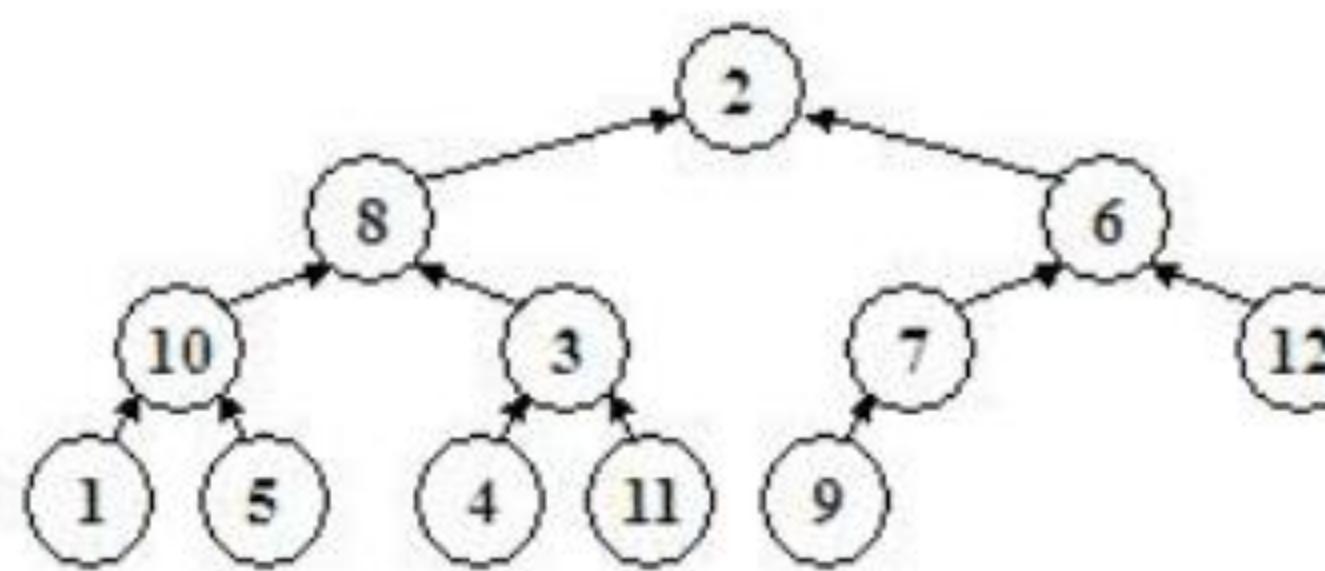
Heapify

Construire heap

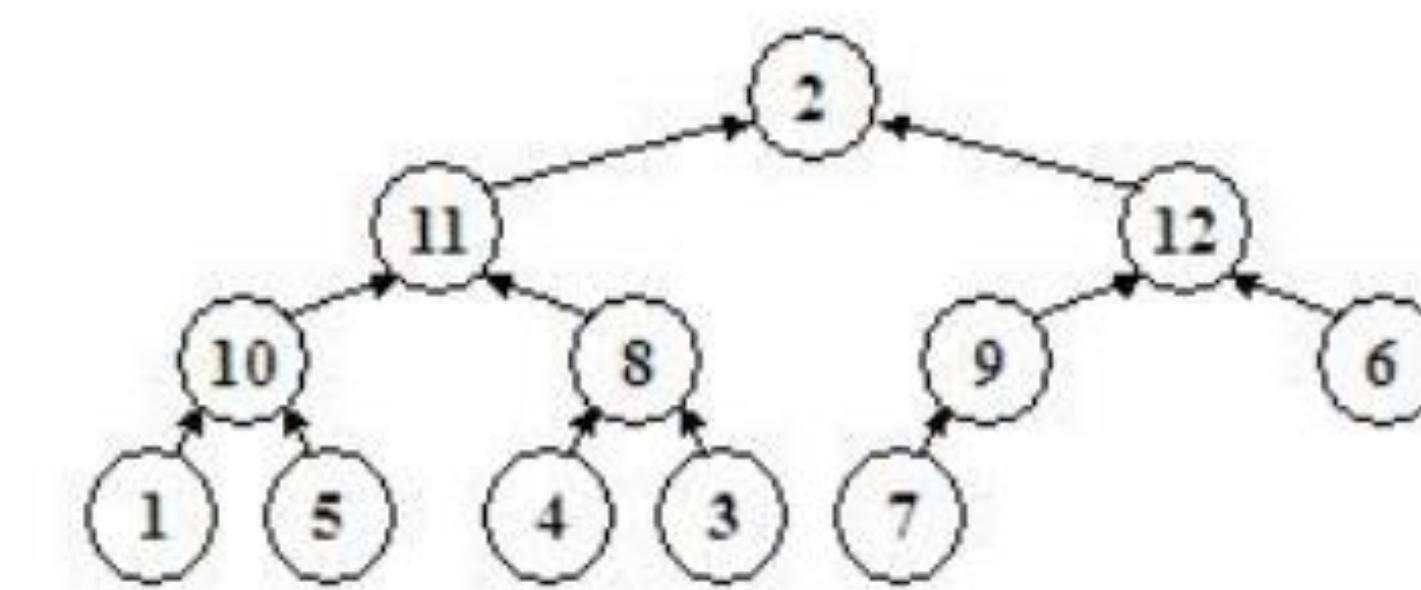
- Inserăm n elemente - $O(n \log n)$
- Liniar (heapify):
 - Coborâm fiecare element începând de jos în sus

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

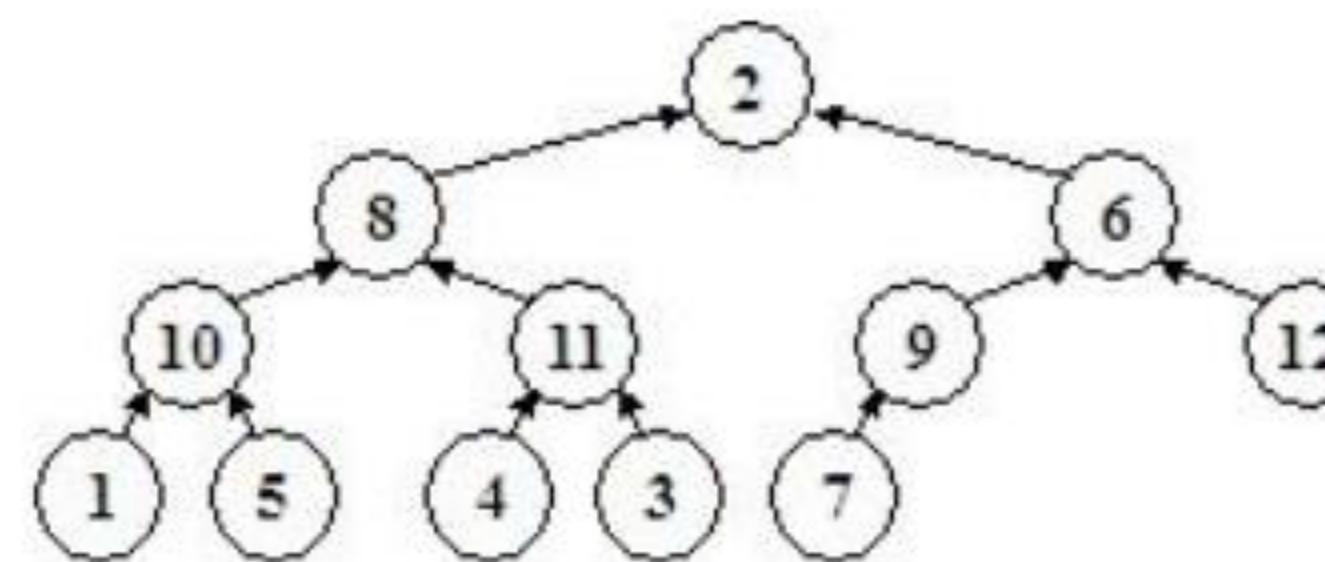
Heapify



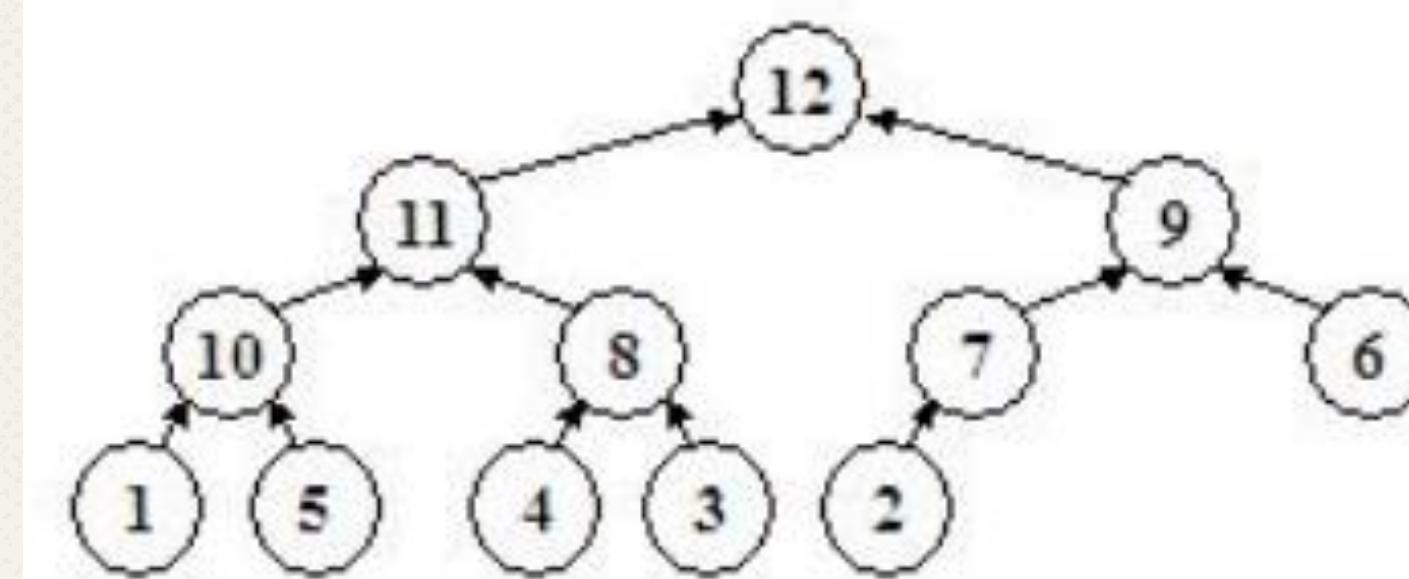
Nivelul frunzelor este organizat



Ultimele trei niveluri sunt organizate



Ultimele doua niveluri sunt organizate



Structura de bază

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
 - S.a.m.d.

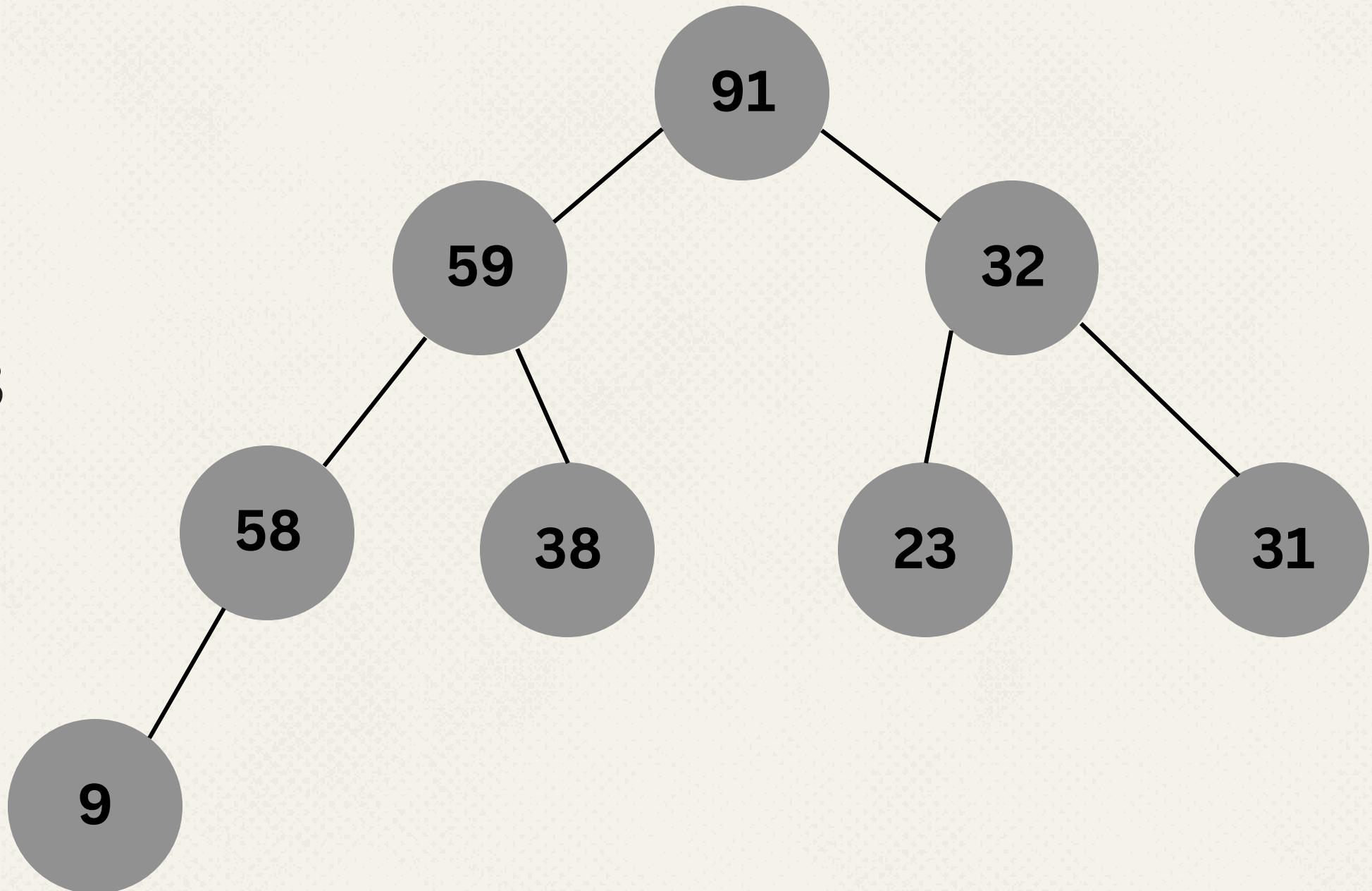
$$\begin{aligned}\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} O(h) &= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\end{aligned}$$

Problemă

- Se dau multe operații de genul:
 - Inserare număr - $O(\log n)$
 - Afisare minim - $O(1)$
 - Elimină indice
- Cum putem folosi un heap?
- De ce nu e evident ?
- Eliminare număr (Nu știm indexul din heap și fără să știm indexul nu putem elmina în $\log n$)

Eliminare element cunoscând pozitia

```
elimina(i) {  
    heap[i] = heap[n--];  
    Eliminăm pozitia 3, respectiv val 58  
    coboara(i);  
    urca(i);  
}
```



Problemă

- Se dau multe operații de genul:
 - Inserare număr
 - Afisare 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 stergere, dar nu-l stergem 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 minim	$O(\log n)$	$O(\log n)$
Căutare minim	$O(1)$	$O(1)$
Construcție n elemente	$O(n)$	$O(n)$
Uniune (2 heapuri de n elemente)	$O(n)$	$O(n)$

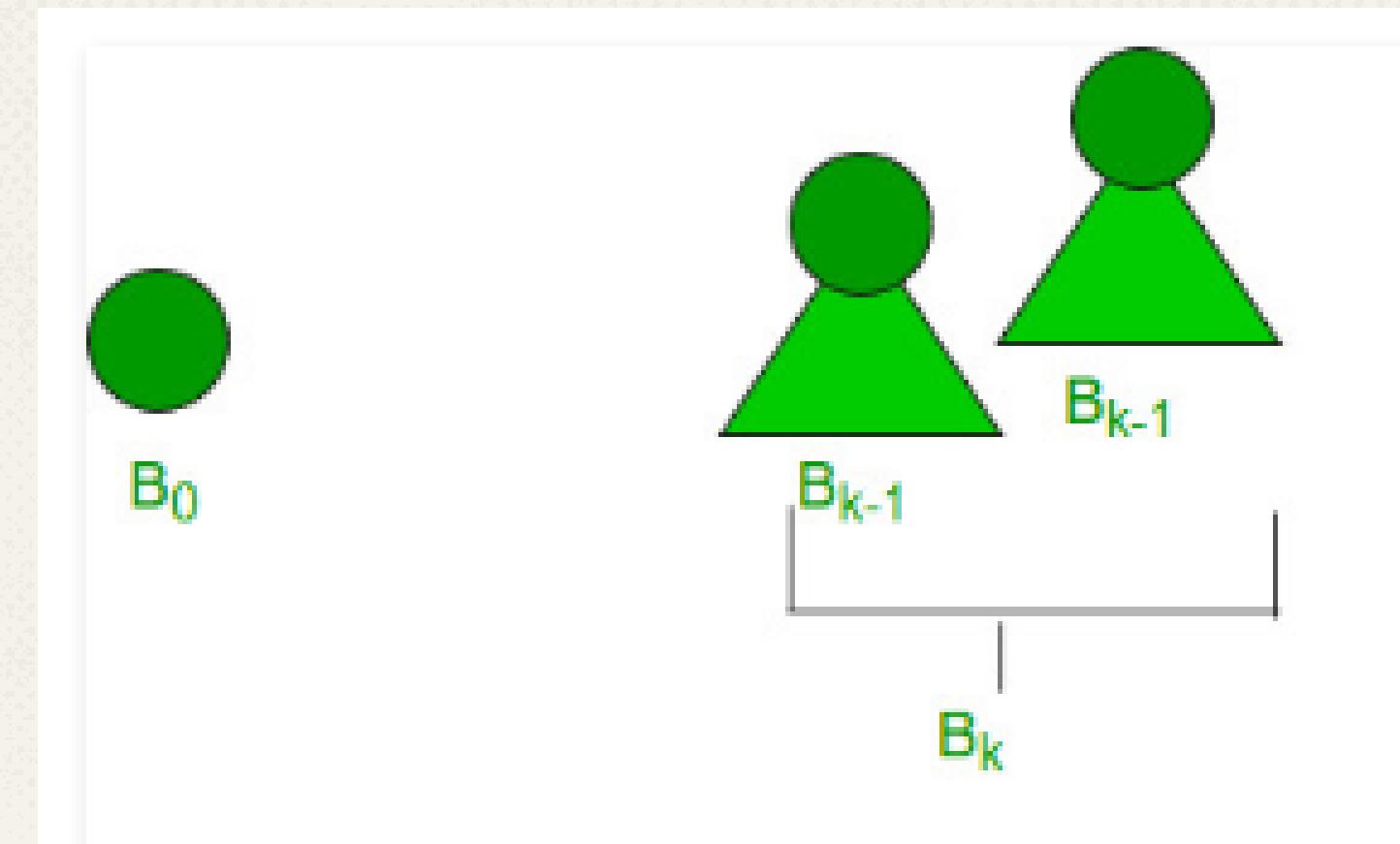
Heapuri Binomiale și Heapuri Fibonacci

- 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)	$\Theta(1)$

Arbore binomiali

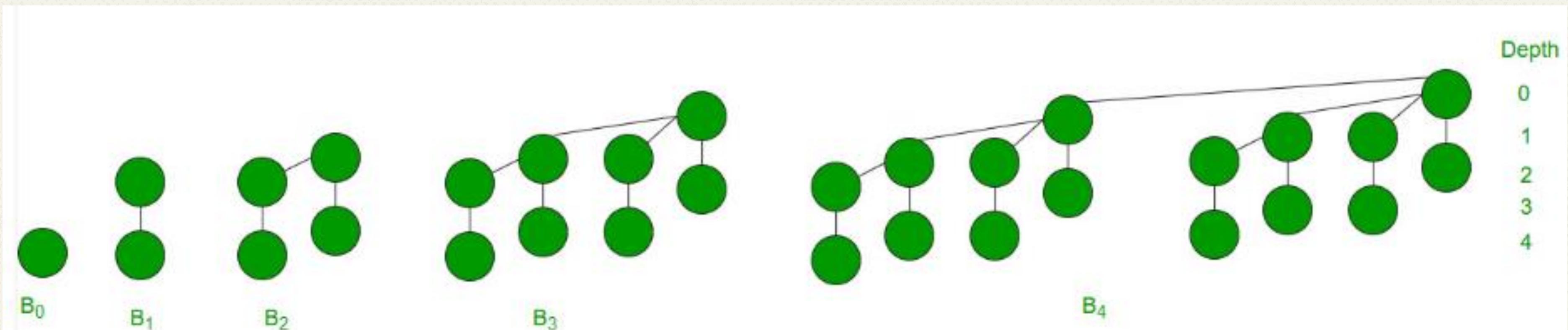
- Un arbore binomial de ordin 0 are un nod (rădăcina)
- Un arbore binomial de ordin K poate fi format prin reuniunea a doi arbori binomiali de mărime $K-1$, făcând pe unul dintre ei fiul stâng al celuilalt



Arbore 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 $k-1, k-2, \dots, 0$

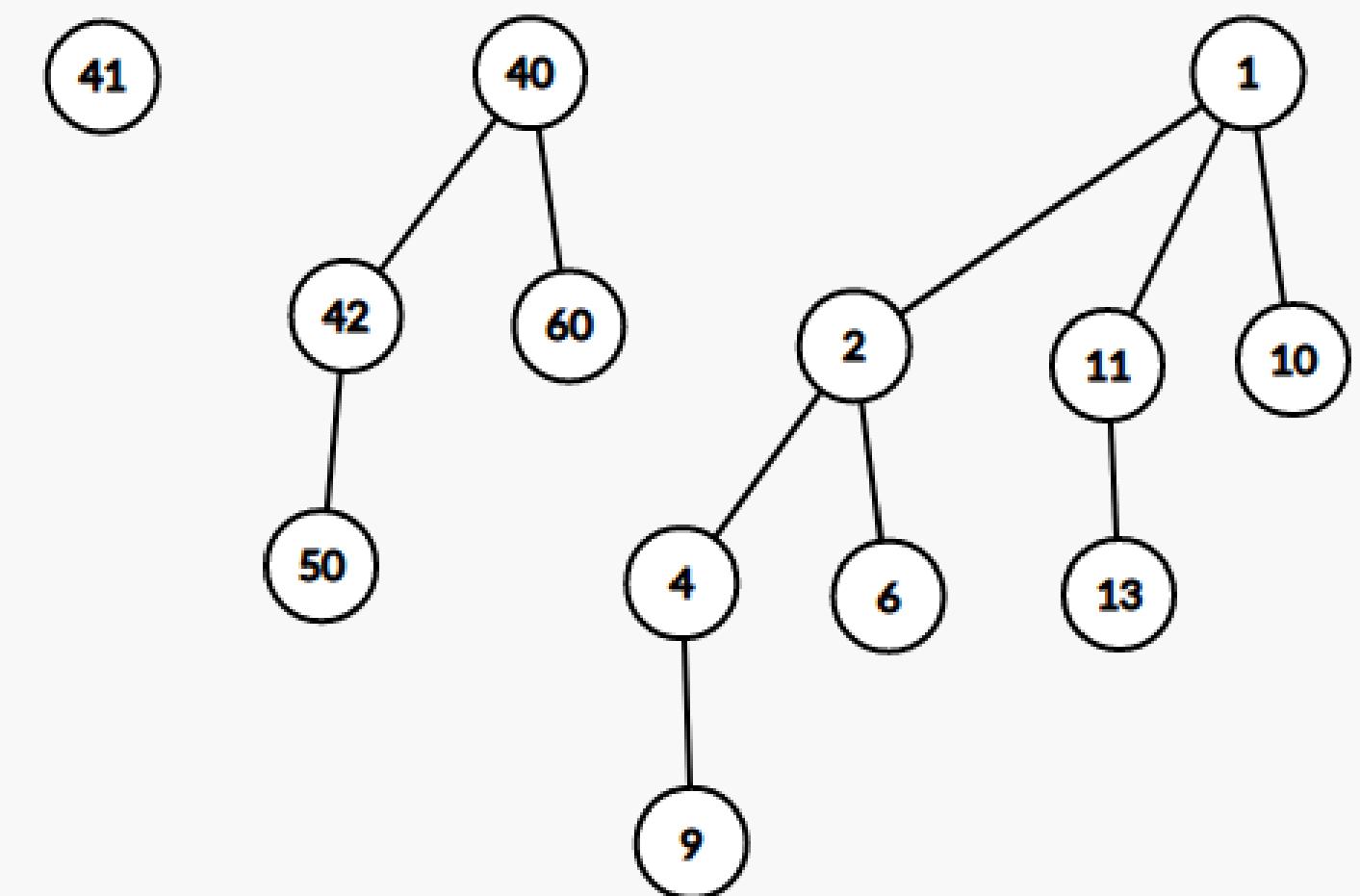


Heapuri Binomiale

- 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?

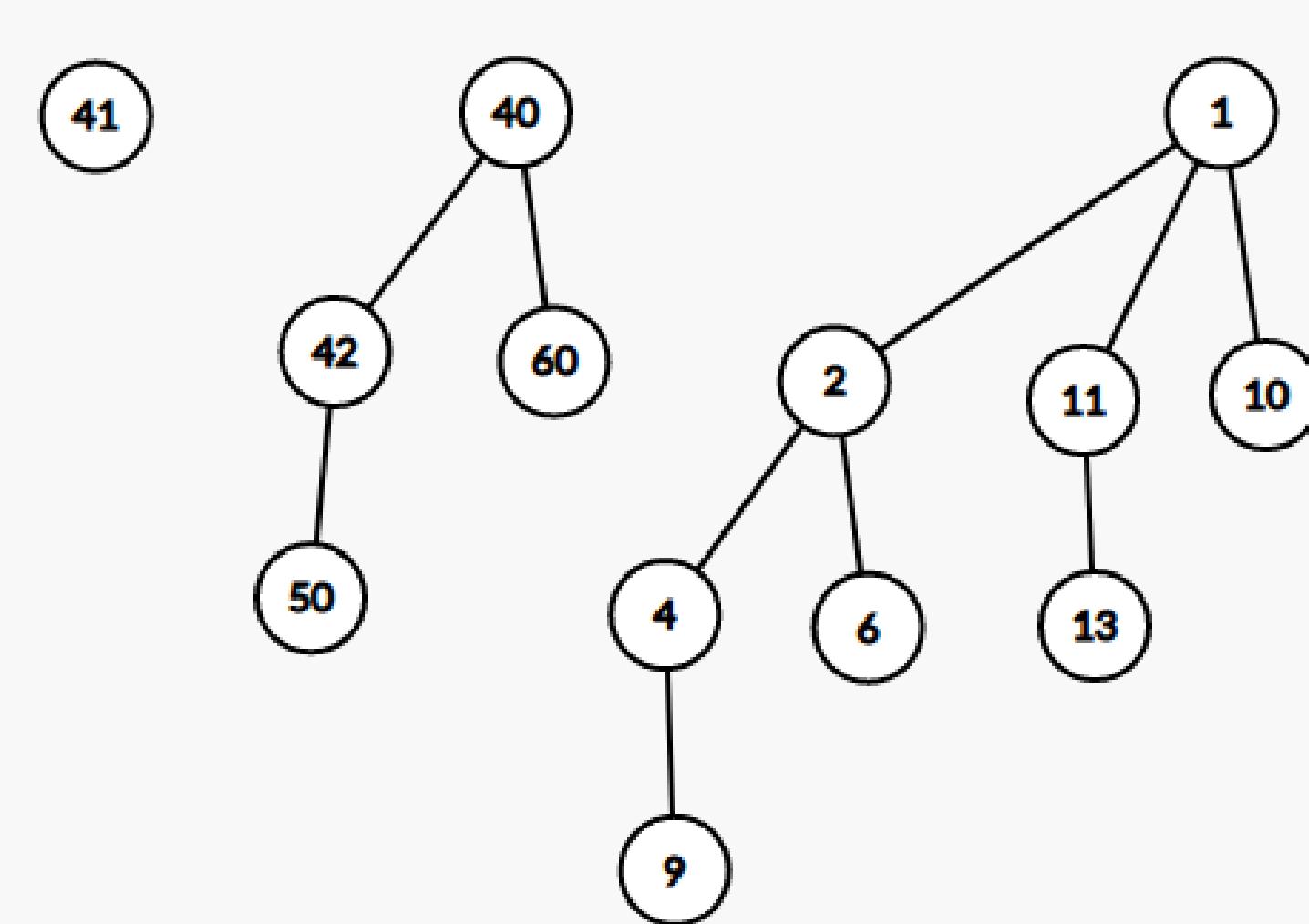
Heapuri Binomiale

- 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

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

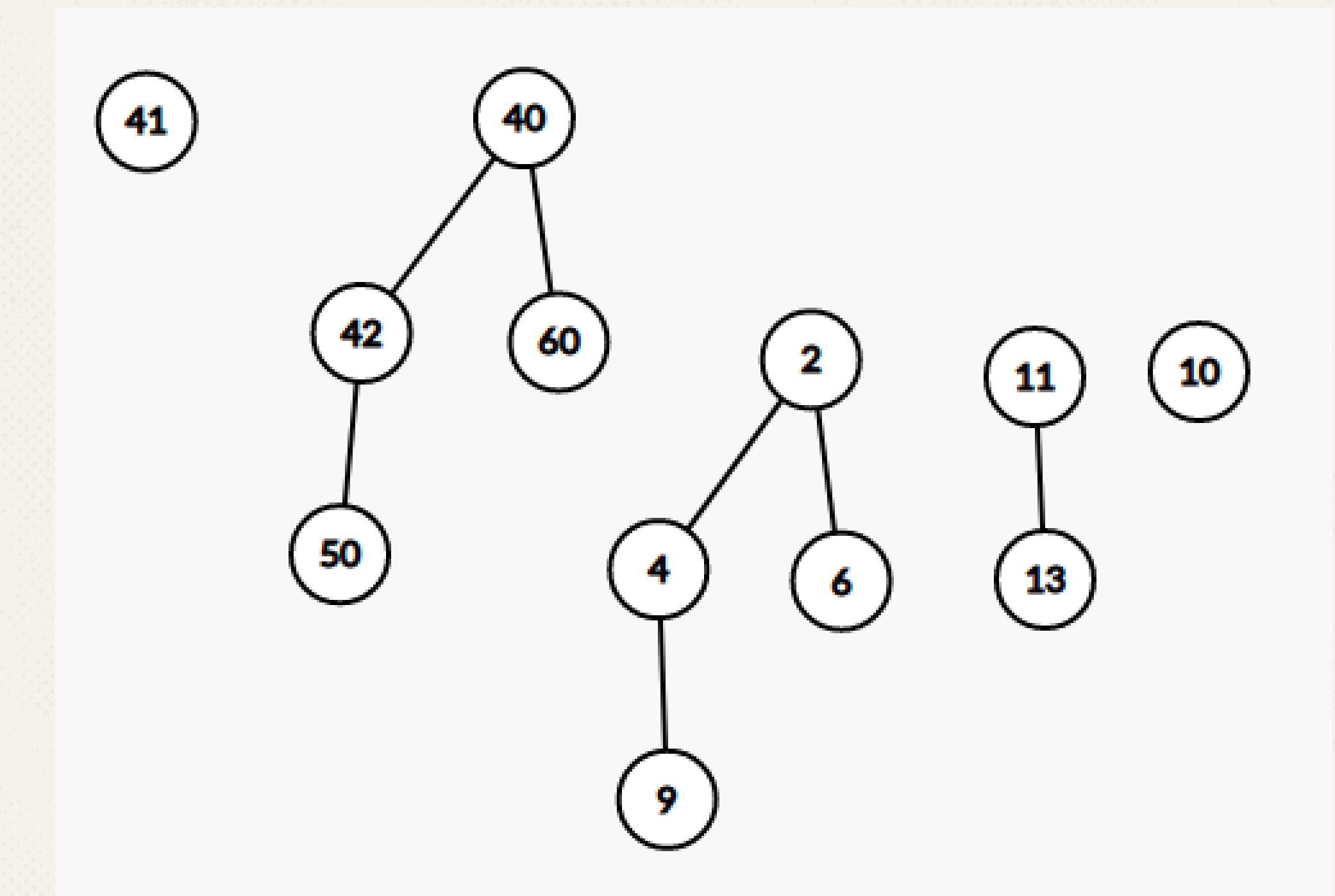


Heapuri Binomiale - Căutare minim

```
void _find_min() {
    // Inițializăm min_node ca fiind nullptr pentru a începe căutarea
    min_node = nullptr;
    // Iterăm prin fiecare arbore din lista de arbori
    for (Node* tree : trees) {
        // Verificăm dacă min_node este nullptr sau dacă valoarea nodului curent este mai mică decât valoarea
        // min_node-ului curent
        if (min_node == nullptr || tree->value < min_node->value) {
            // Dacă una dintre condiții este îndeplinită, actualizăm min_node pentru a fi nodul curent
            min_node = tree;
        }
    }
}
```

Heapuri Binomiale - Extragerea minimului

- Eliminăm minimul
- Apoi facem reunire

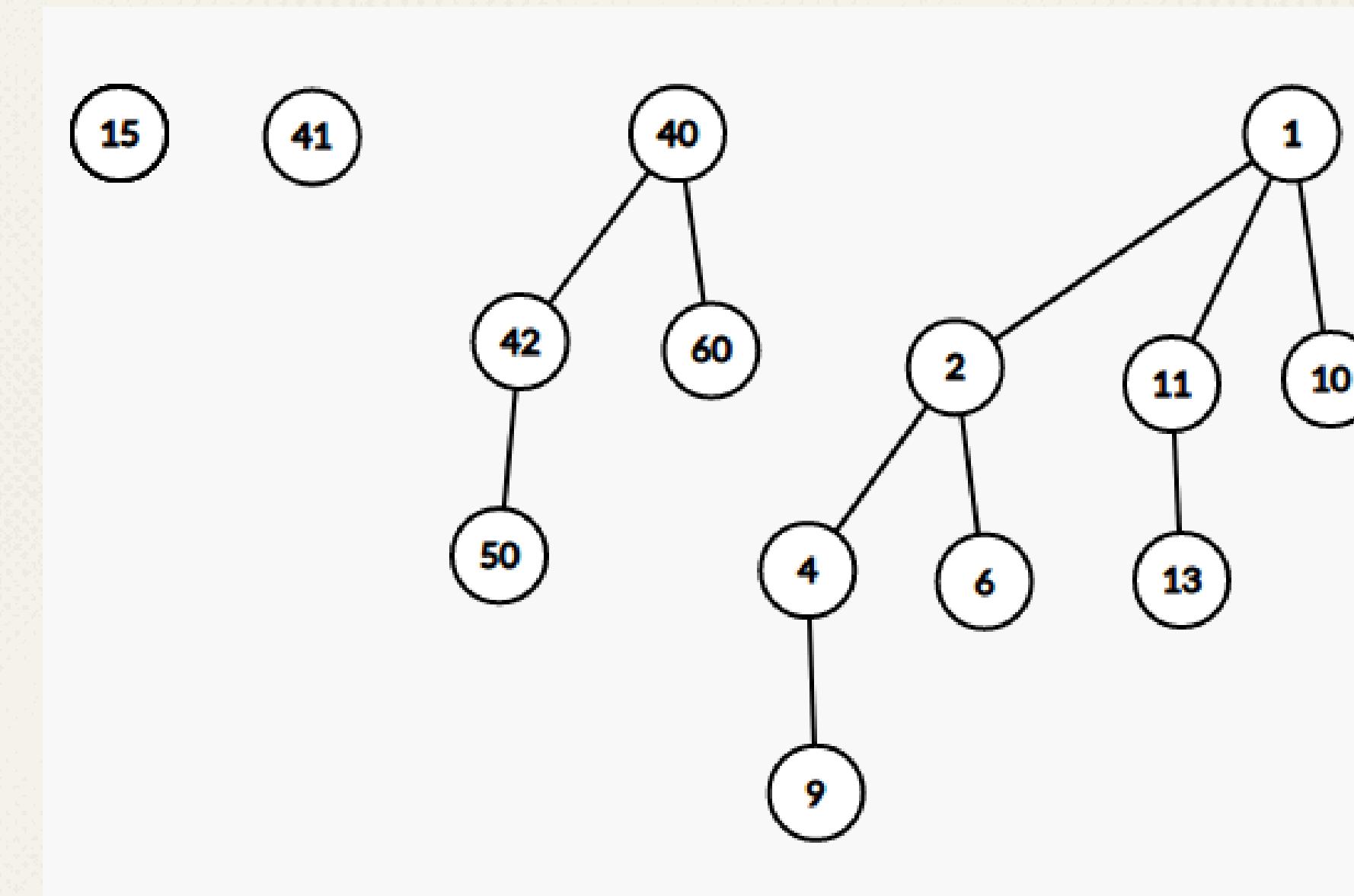


Heapuri Binomiale - Extragerea minimului

```
int extract_min() {
    Node* minNode = min_node; // Salvăm referința către nodul minim
    // Ștergem nodul minim din lista de arbori
    trees.erase(remove(trees.begin(), trees.end(), minNode), trees.end());
    BinomialHeap heap; // Creăm un heap binomial ce conține copiii nodului minim
    heap.trees = minNode->children;
    merge(heap); // Unim heap-ul creat cu cel temporar
    _find_min(); // Găsim noul nod minim în heap
    count -= 1; // Actualizăm contorul pentru numărul de elemente din heap
    return minNode->value; // Returnăm valoarea minimă extrasă
}
```

Heapuri Binomiale - Inserare

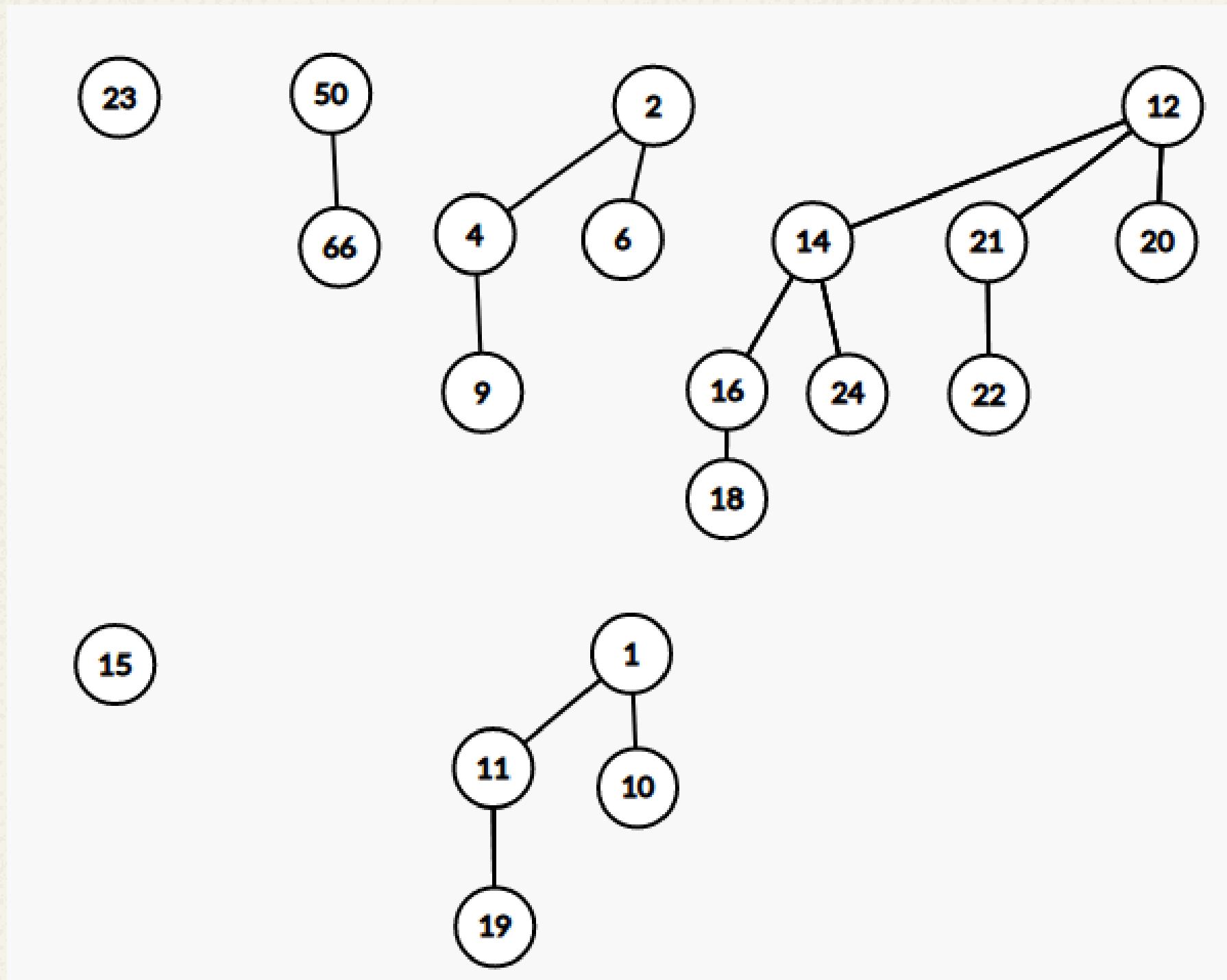
Adăugăm un arbore binomial de mărime 1, apoi apelăm reunirea.



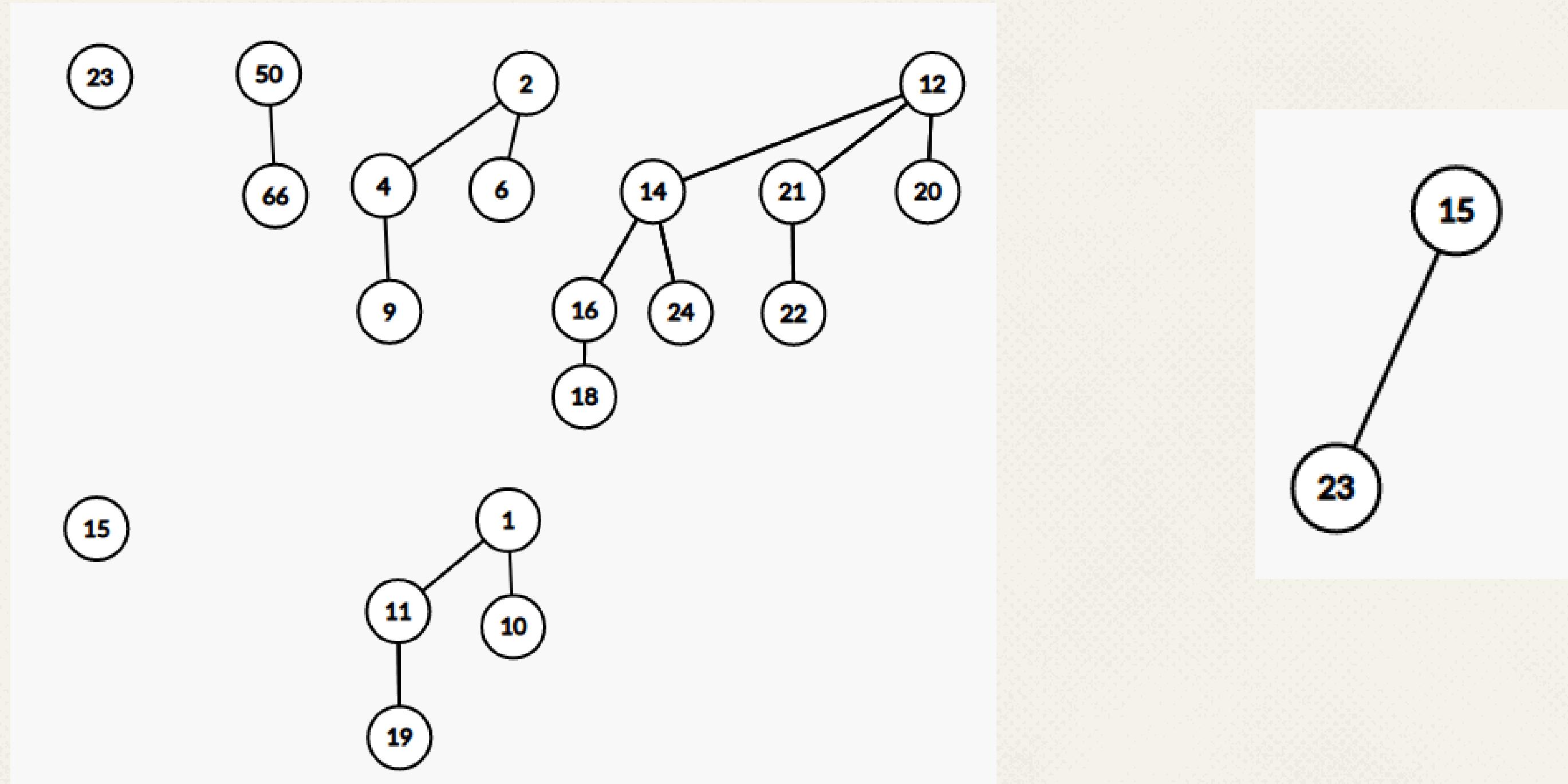
Heapuri Binomiale - Inserare

```
void insert(int value) {  
    // Creăm un nod nou pentru valoarea dată  
    Node* node = new Node(value);  
  
    // Creăm un heap binomial care conține doar nodul nou creat  
    BinomialHeap heap;  
    heap.trees.push_back(node);  
  
    // Unim heap-ul creat cu heap-ul principal  
    merge(heap);  
}
```

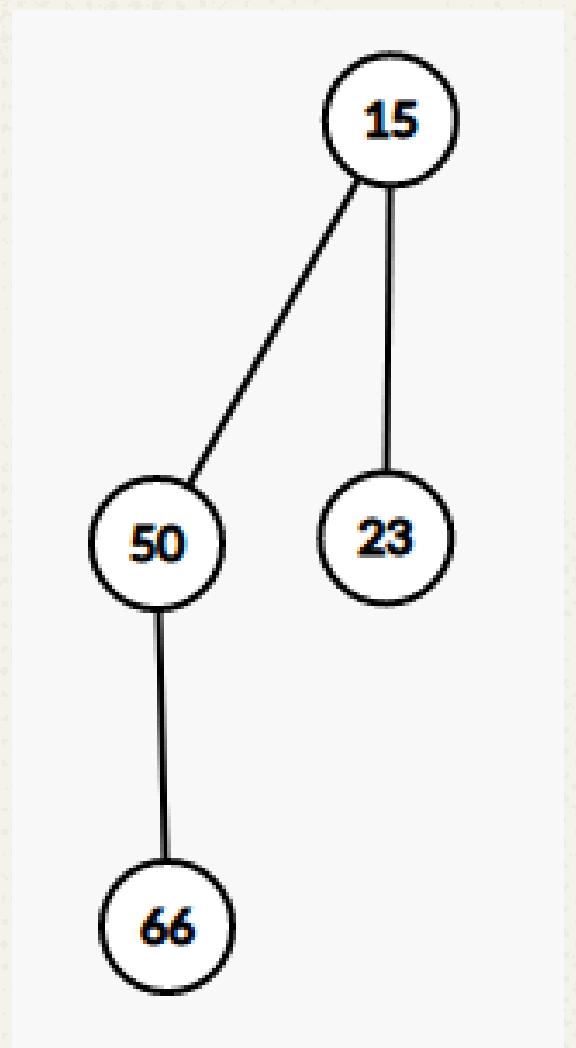
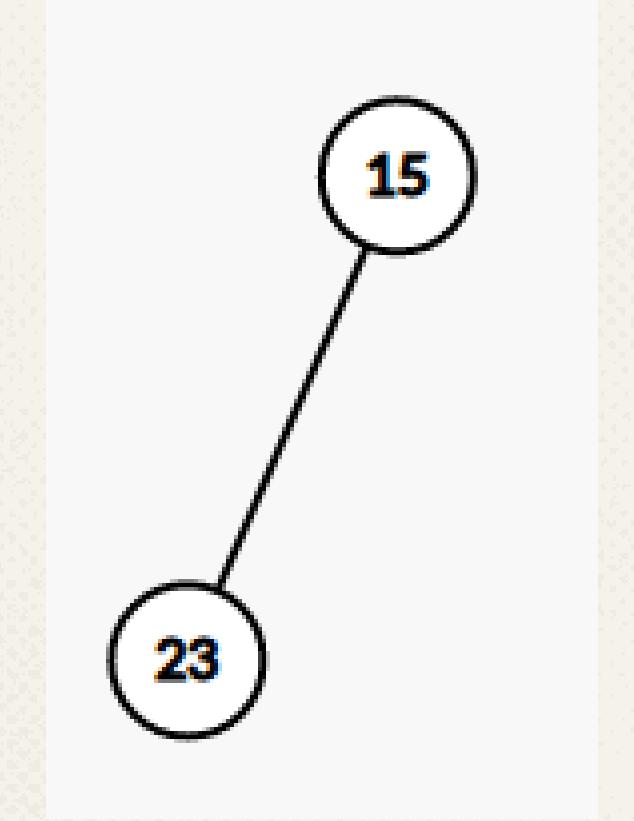
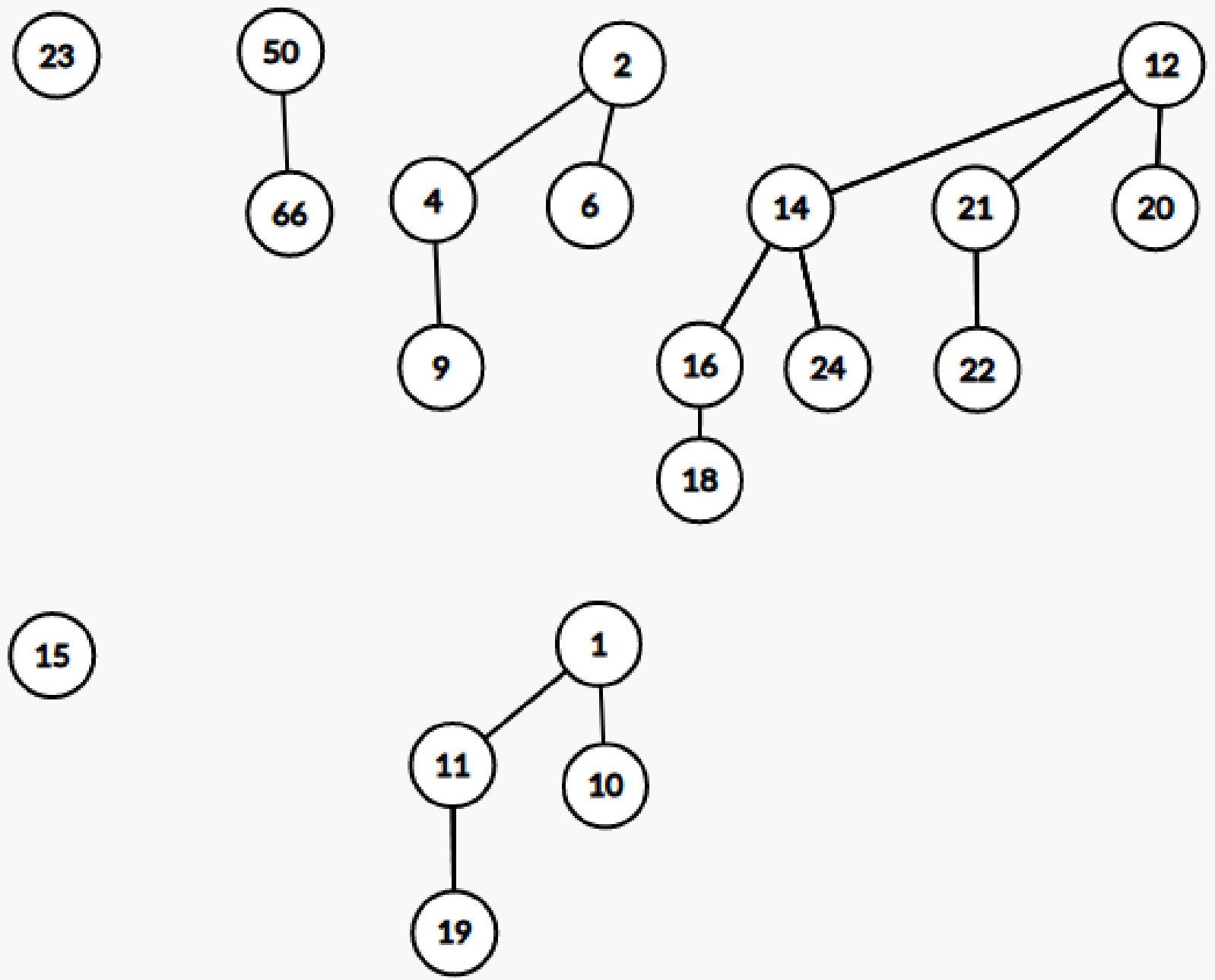
Reuniune!



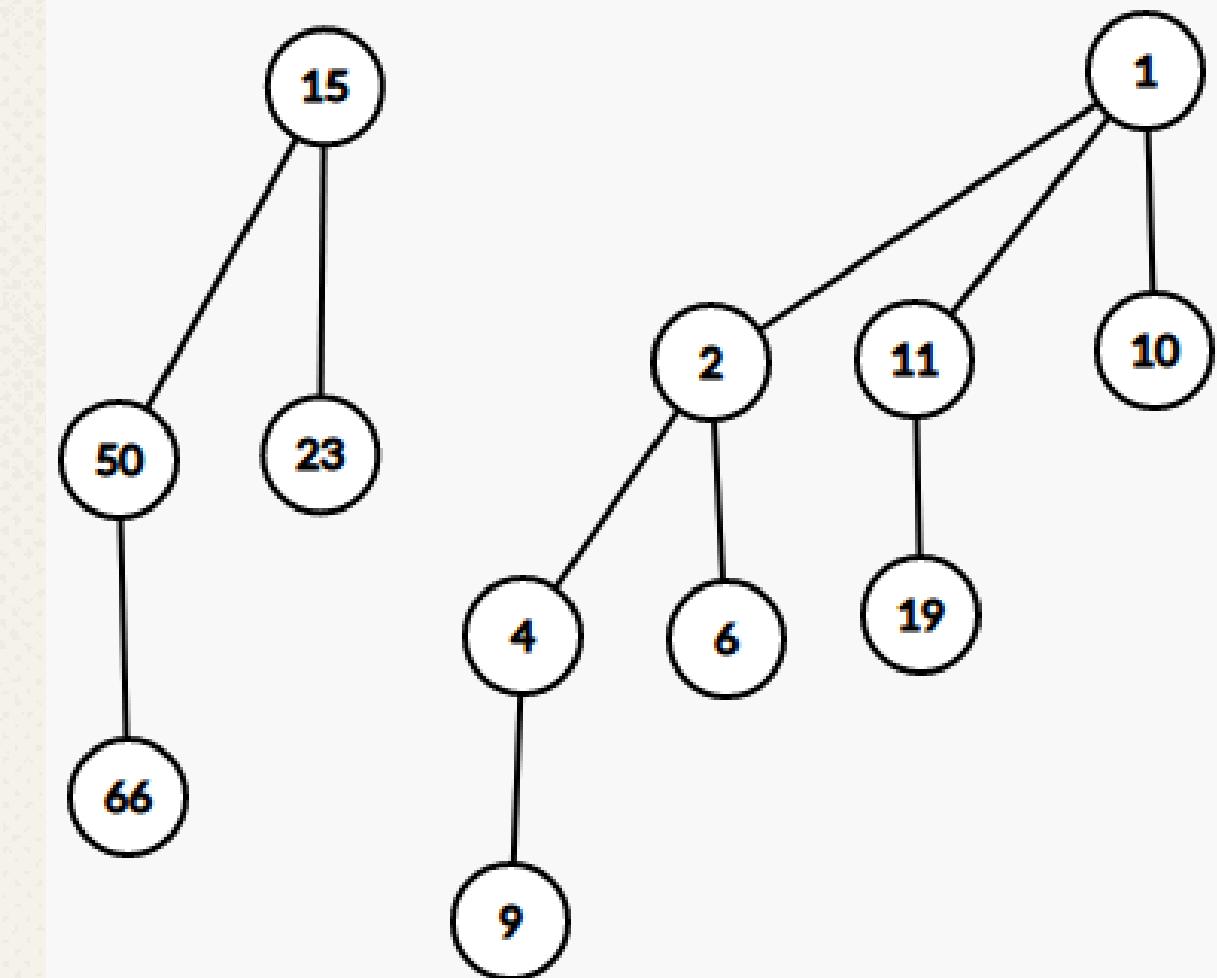
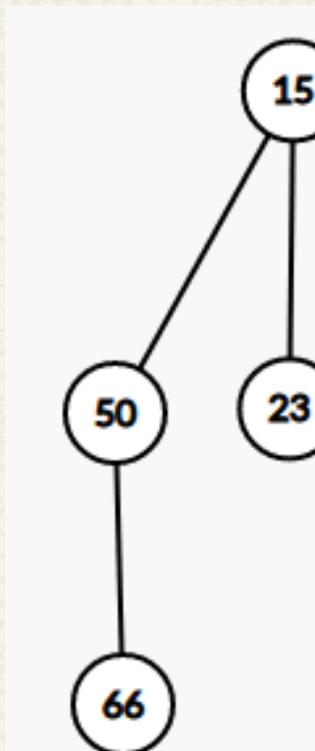
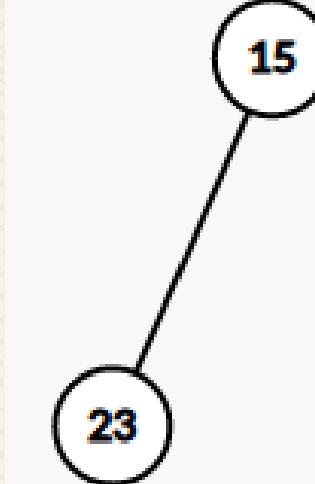
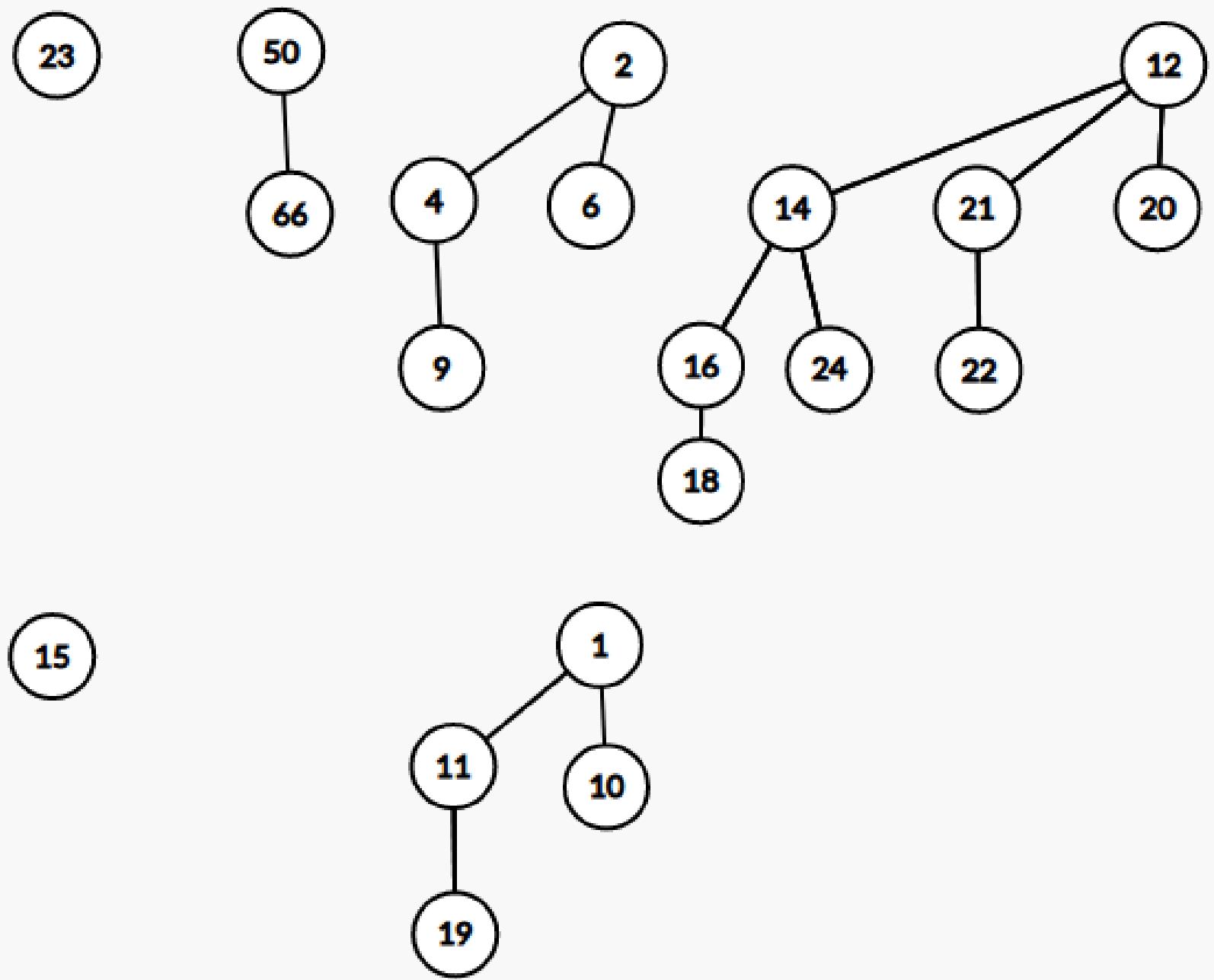
Reuniune!



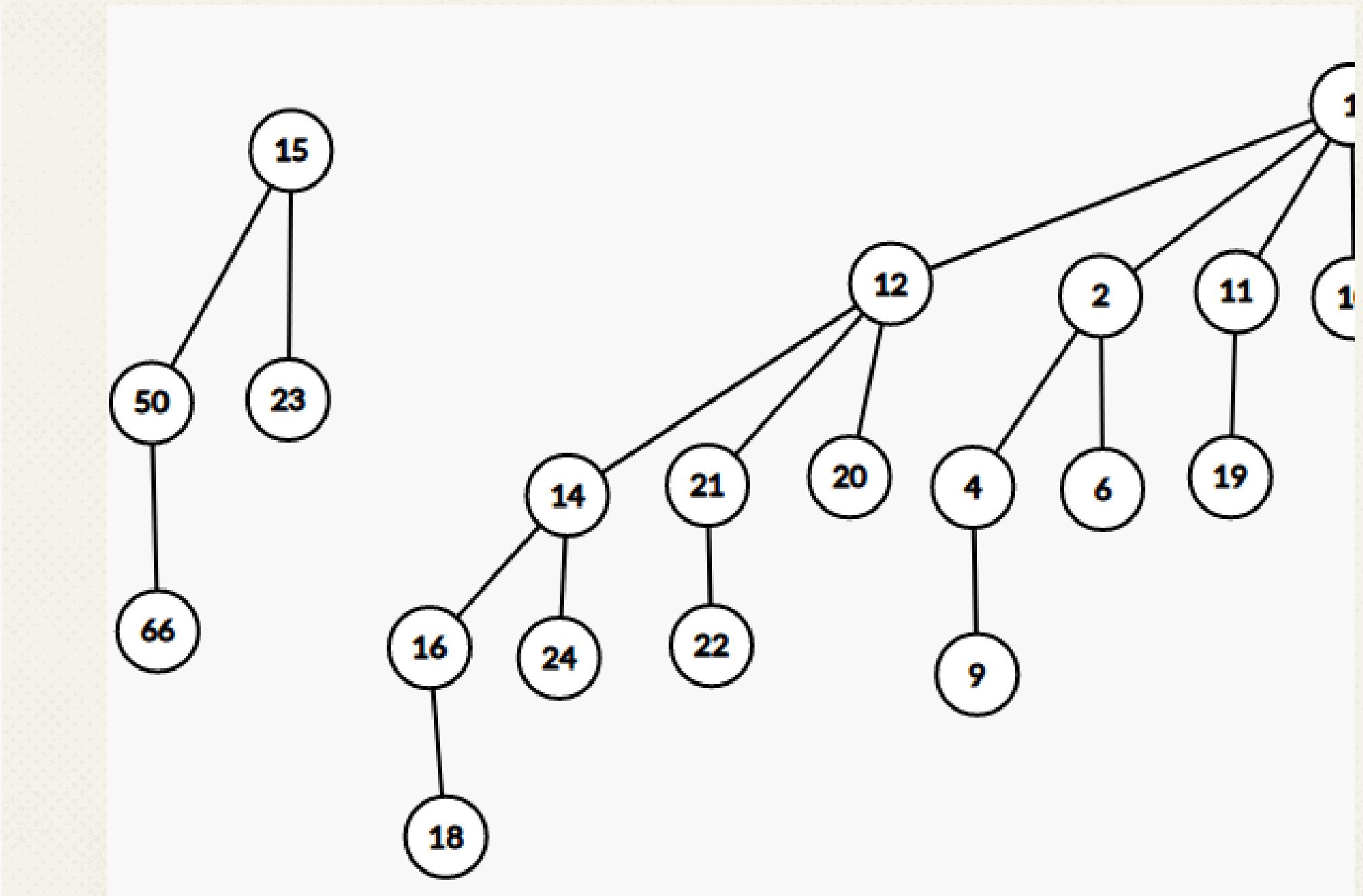
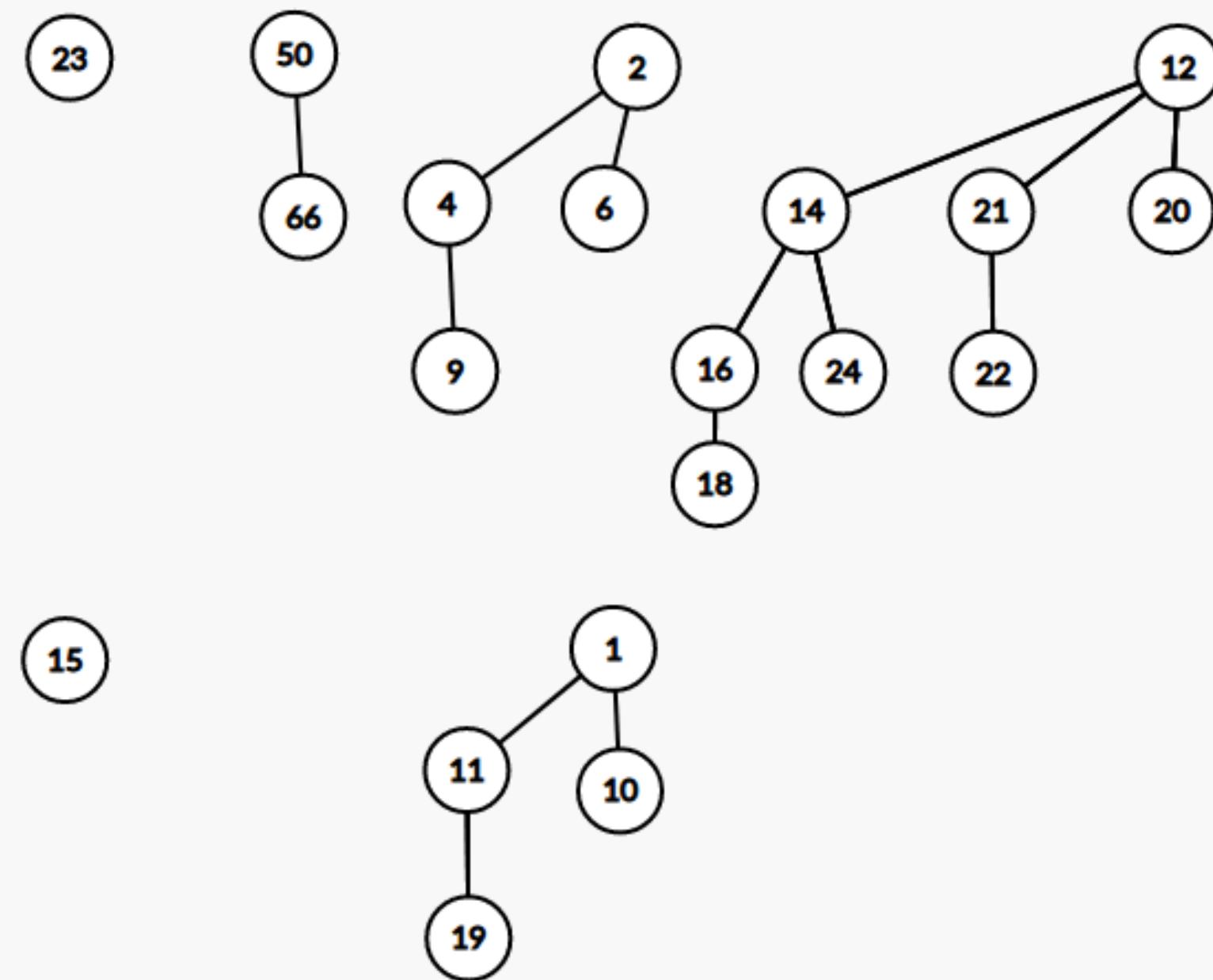
Reuniune!



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.

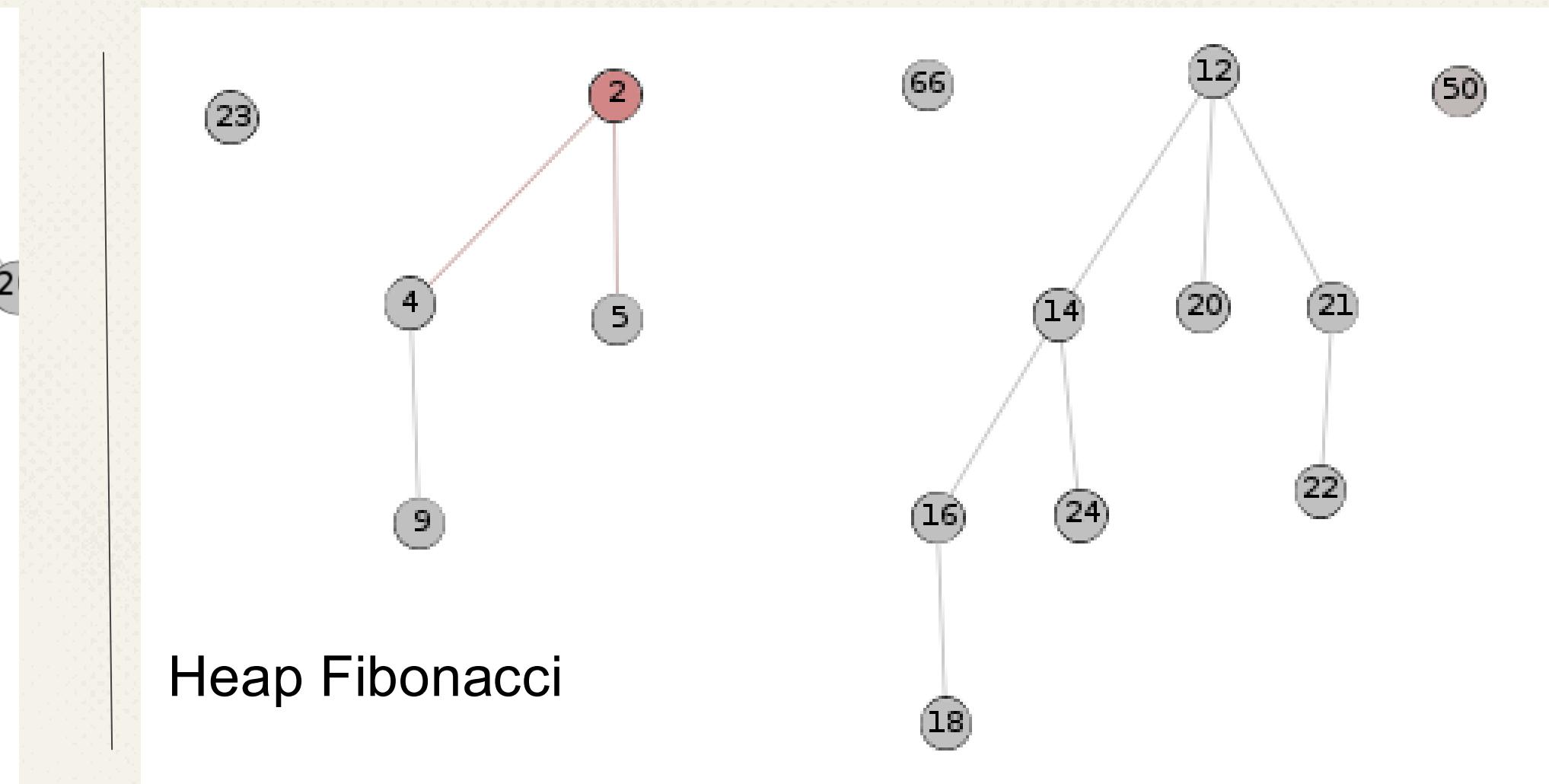
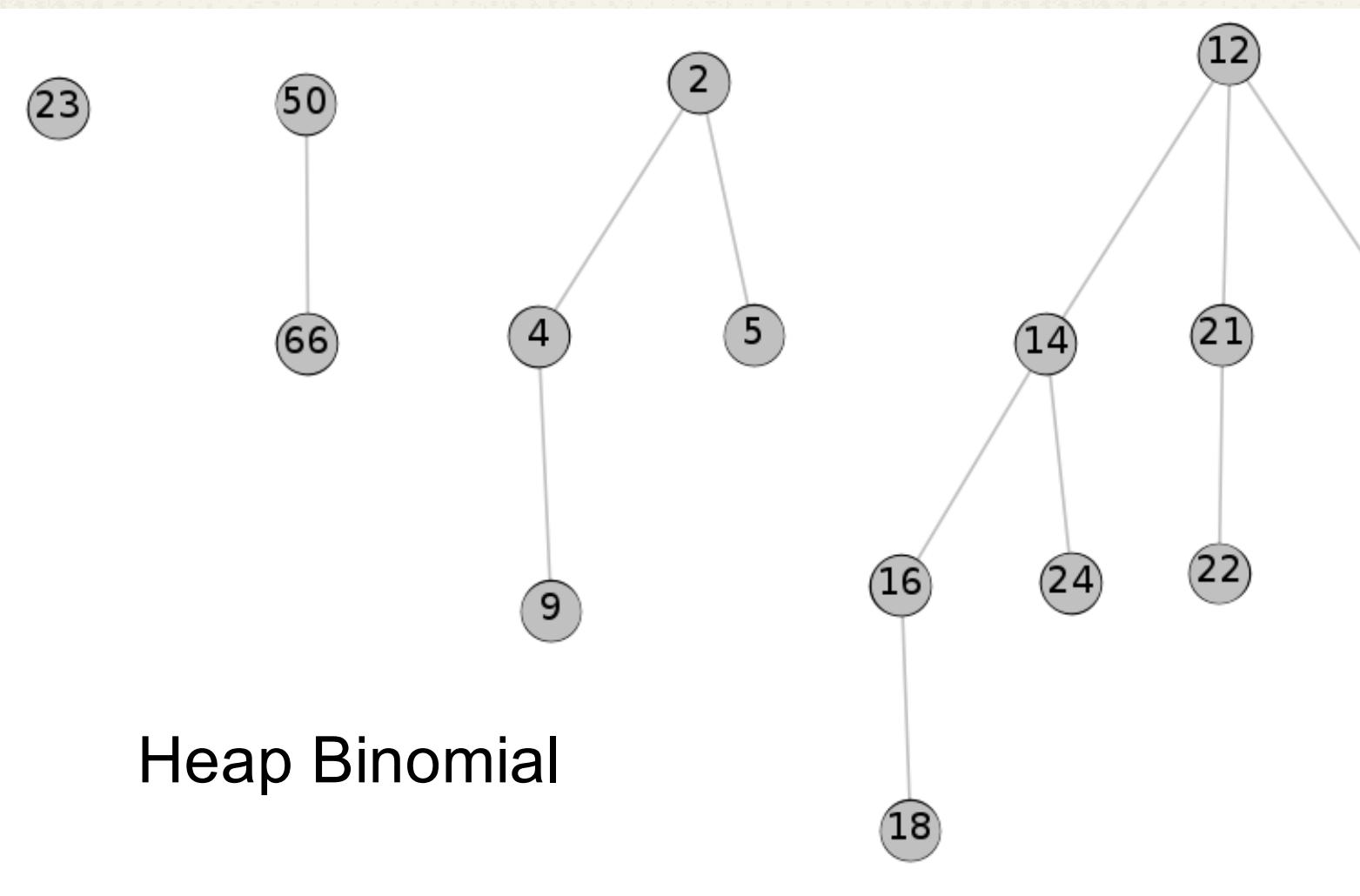
Reuniunea a doi arbori se face în $O(1)$.

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,..., k-1, dar nu neapărat sortați de la stânga la dreapta.

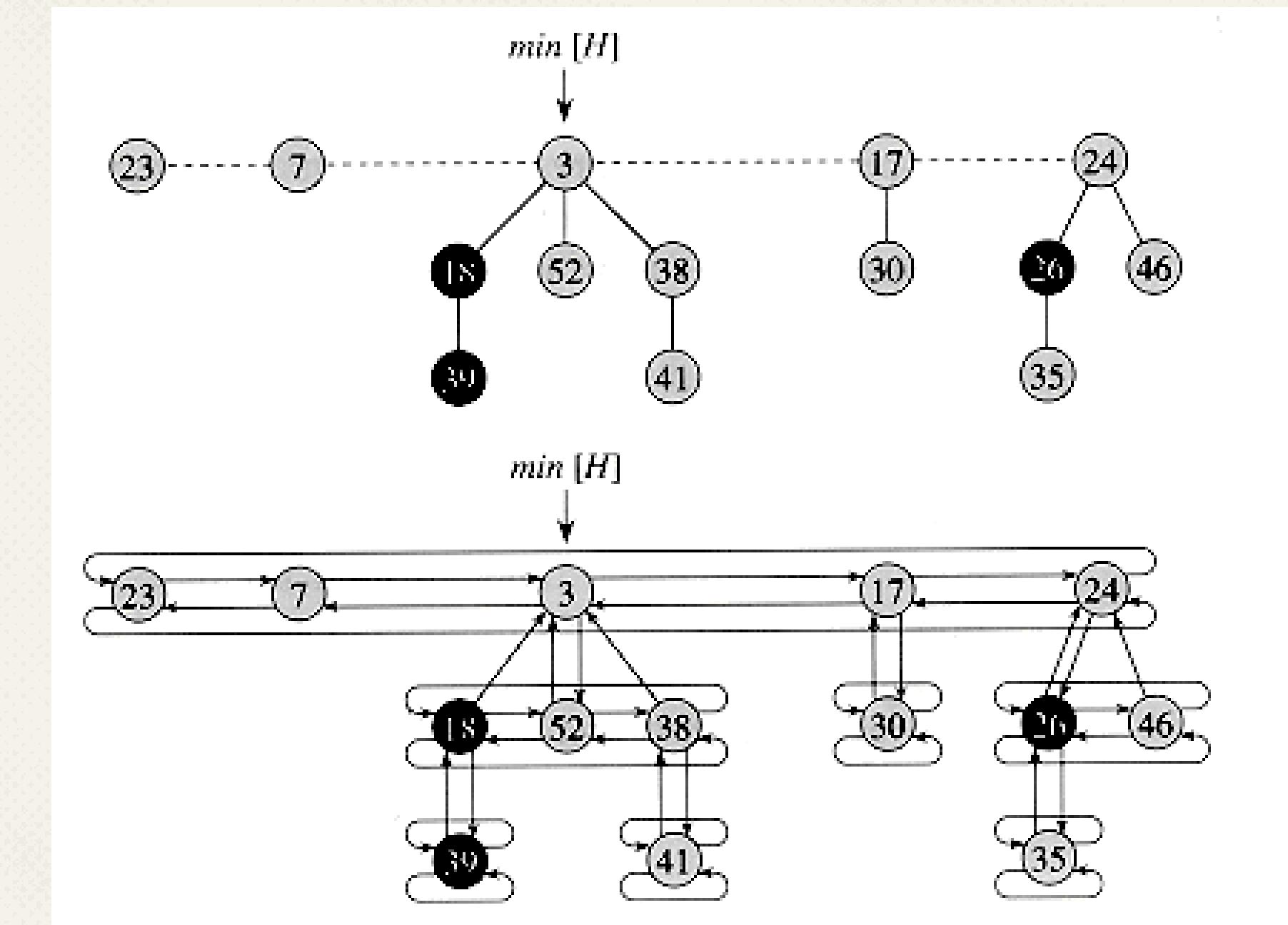
Heapuri Fibonacci

- 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 compoñă 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.



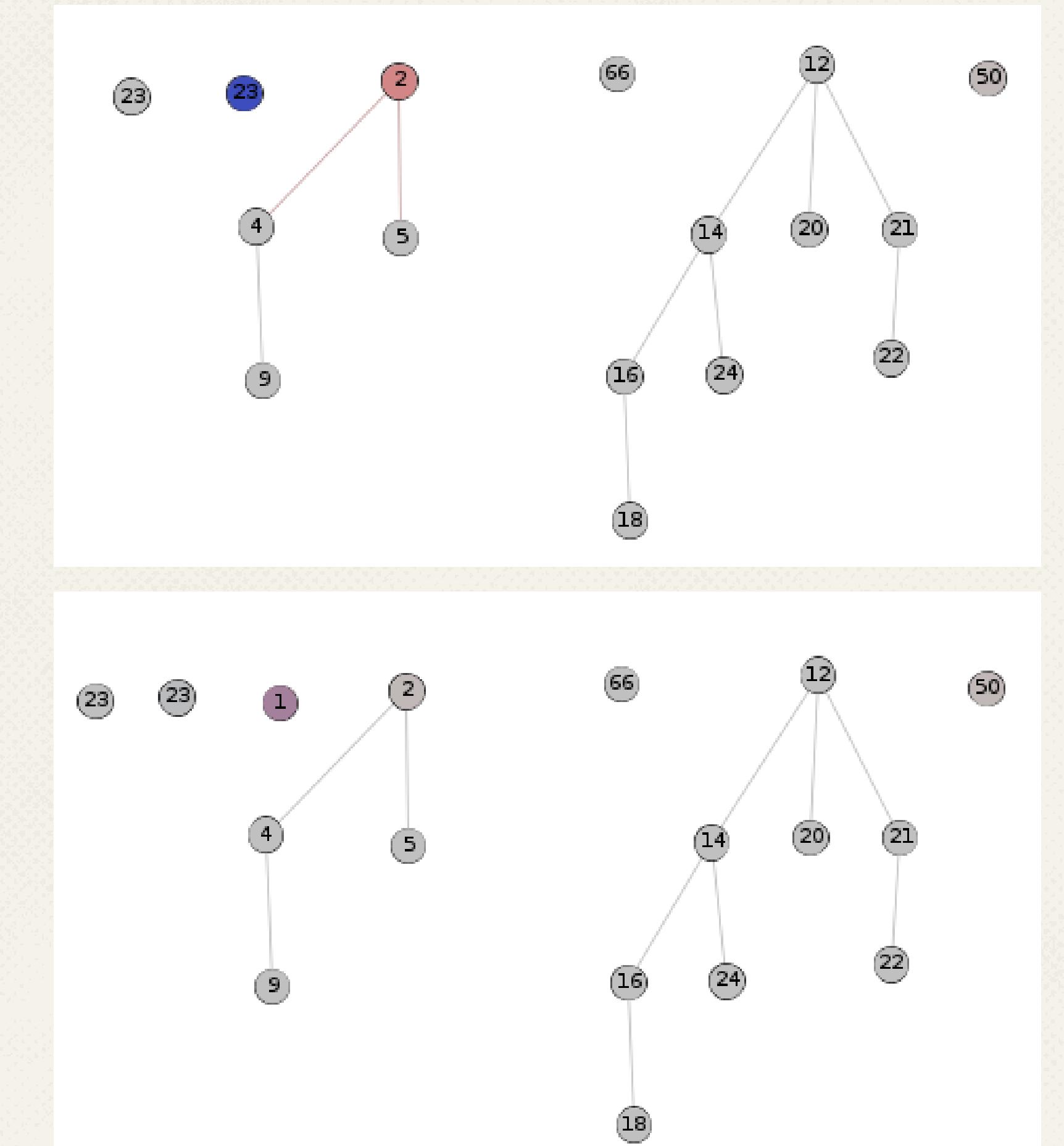
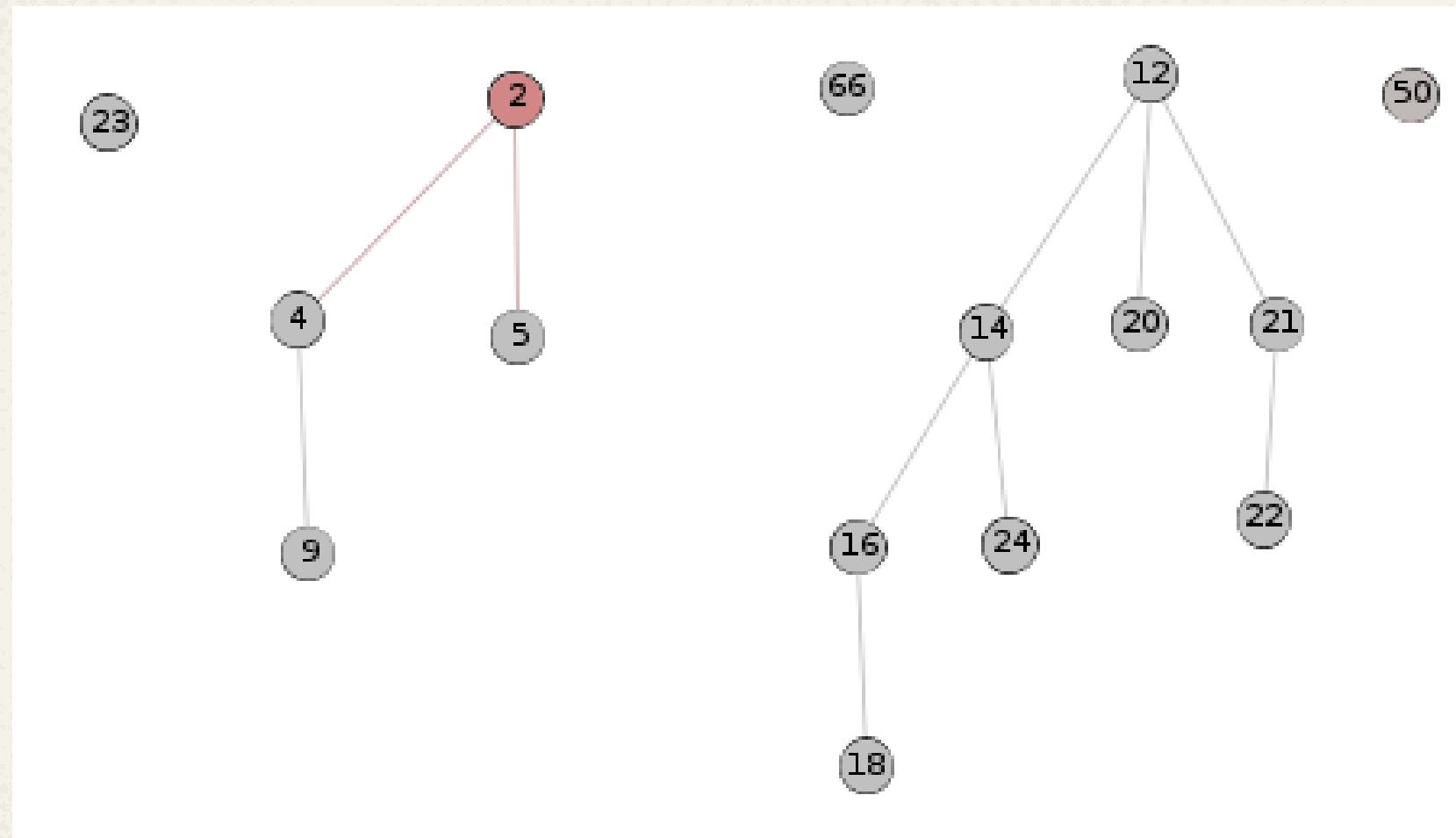
Implementare

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



Inserare nod

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



Inserare nod

```
• void insertion(int val) {  
•     Node* new_node = createNode(val);  
•     // Verificăm dacă heap-ul are deja un nod minim  
•     if (mini != nullptr) {  
•         (mini->left)->right = new_node;  
•         new_node->right = mini;  
•         new_node->left = mini->left;  
•         mini->left = new_node;  
•     }  
•  
•     if (new_node->key < mini->key) // Verificăm dacă valoarea noului nod este mai mică decât valoarea nodului minim curent  
•         mini = new_node;  
•     }  
•     else {  
•         mini = new_node; // Dacă heap-ul este gol, noua valoare devine nodul minim  
•     }  
• }
```

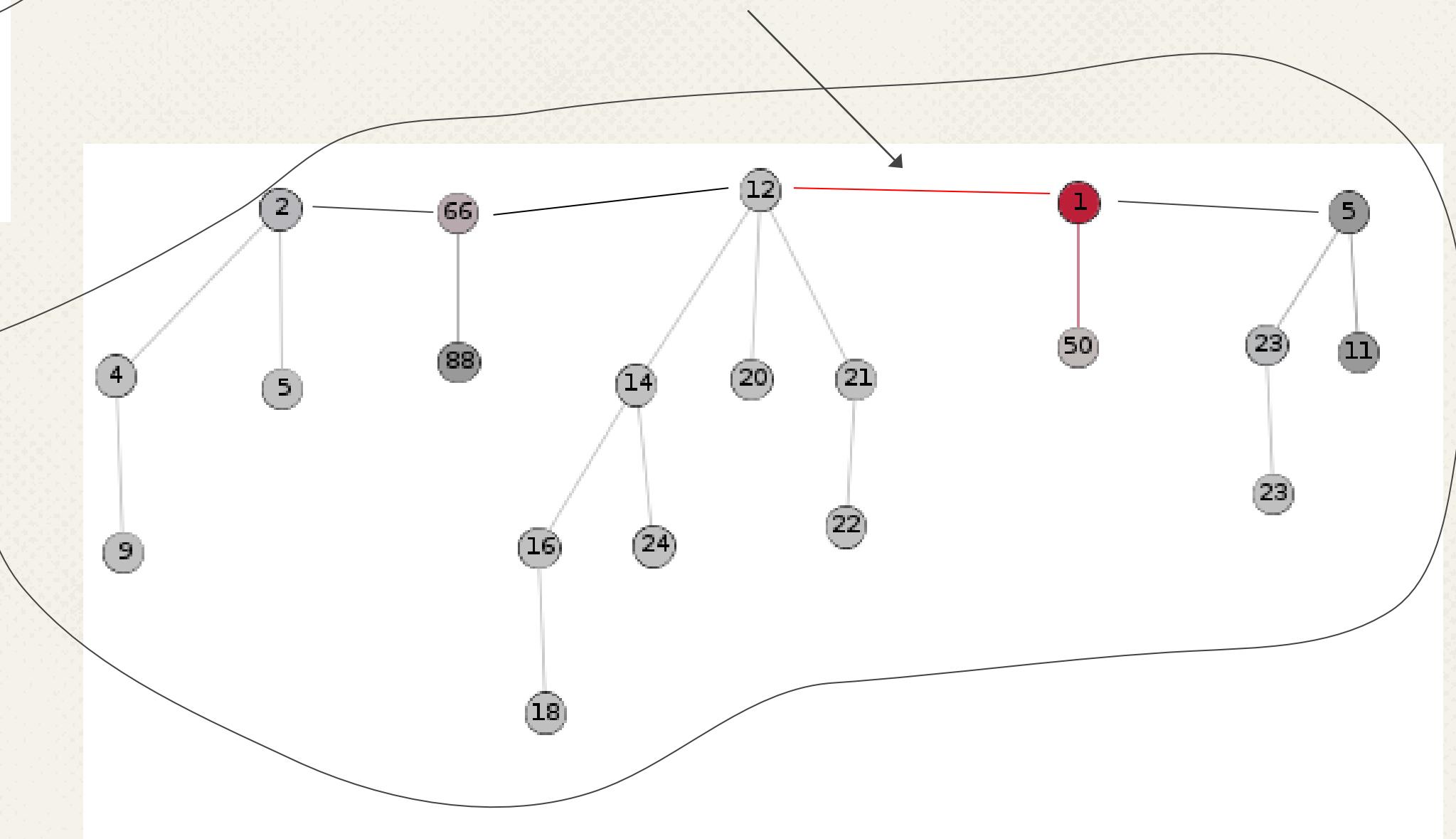
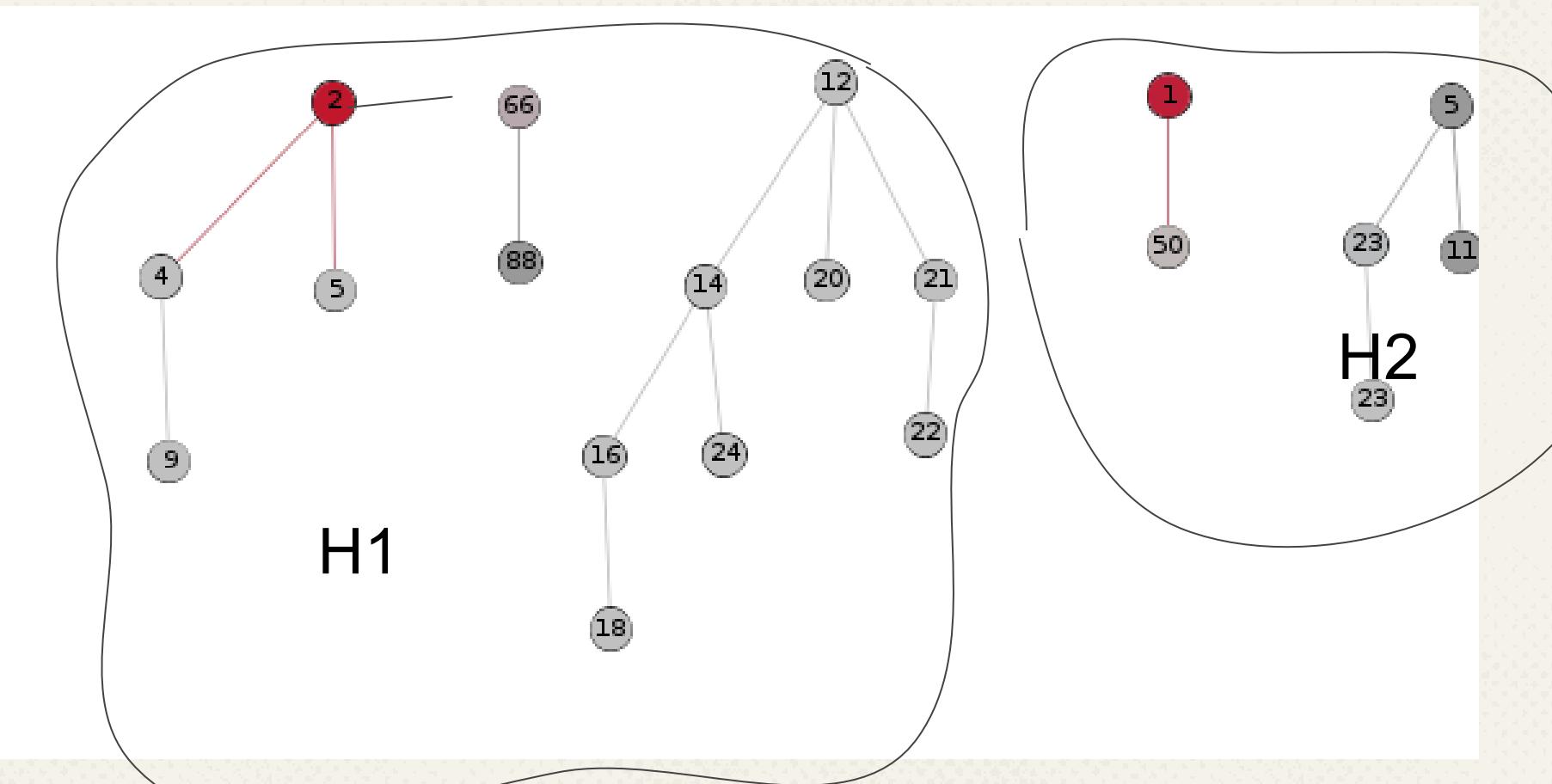
Caută Minimul

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

Reuniune

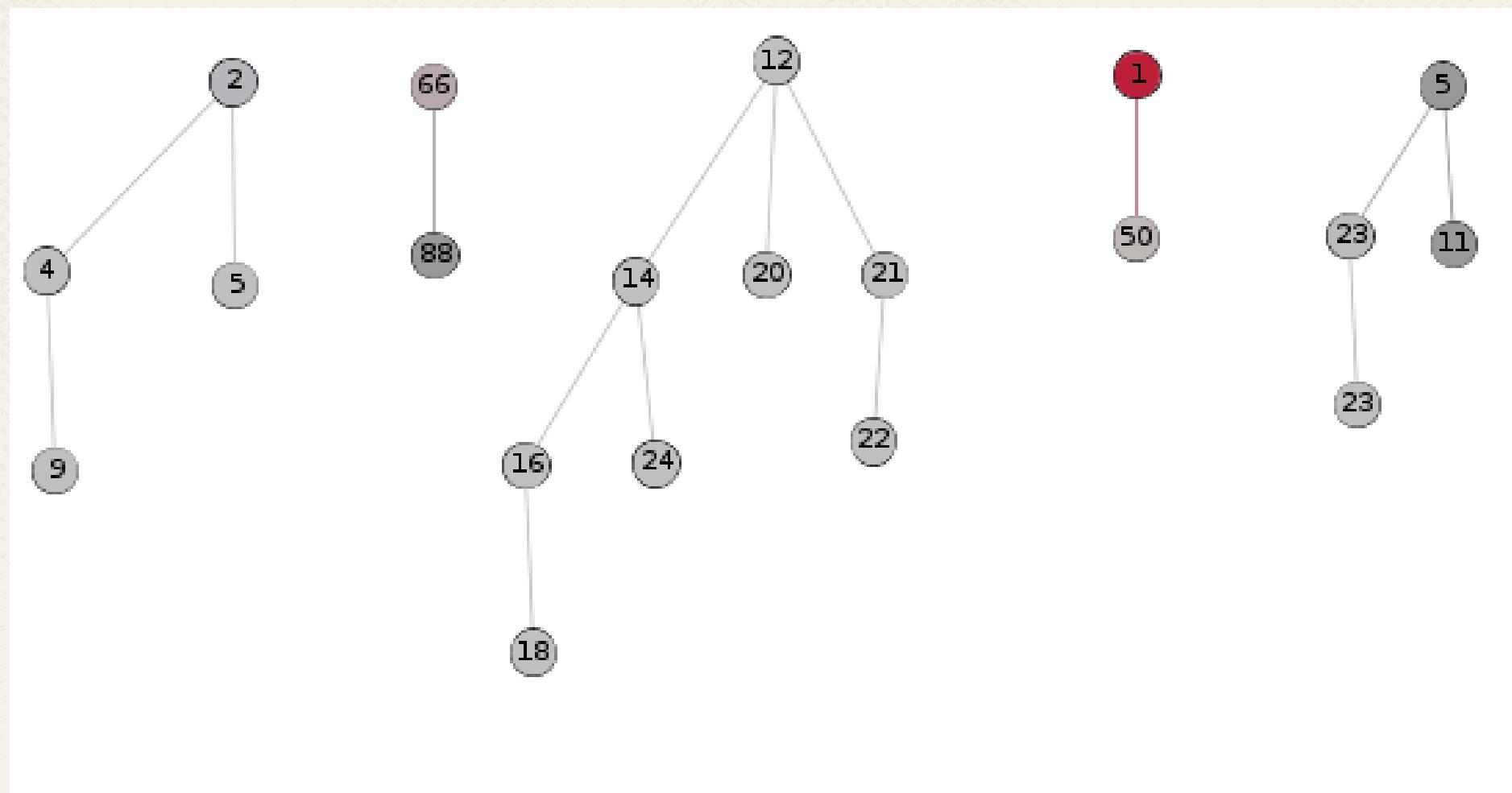
- Concatenăm rădăcinile lui H_2 la cele ale lui H_1 .
- Avem grija să păstrăm lista dublu înlănțuită.
- Avem grija să păstrăm minimul (poate fi unul din cei 2 minimi).
- Nu facem consolidare (putem să avem mai mulți arbori de aceeași mărime).
- **Complexitate $O(1)!!$**

Reuniune



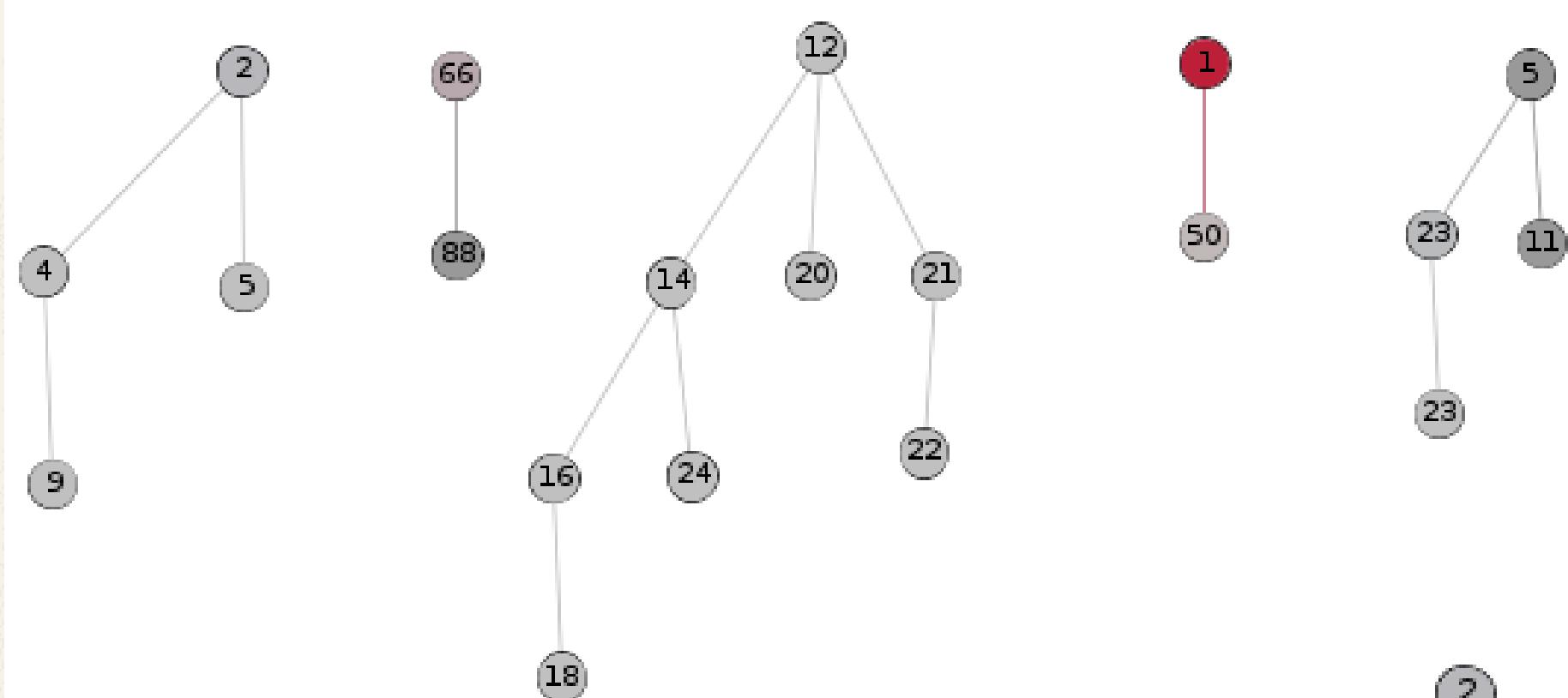
O(1)

Extragere minim



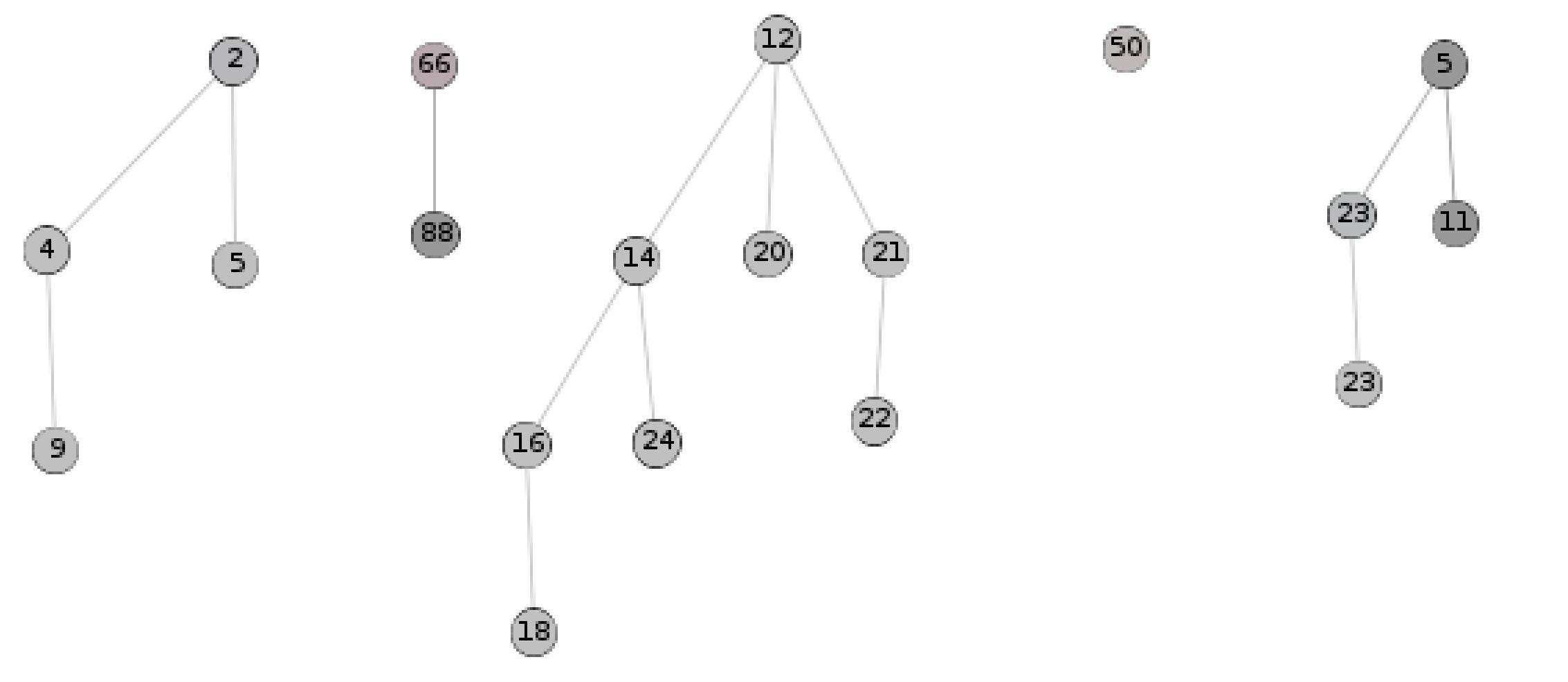
Extragem minim. Fiii să devin arbori liberi.

Extragere minim

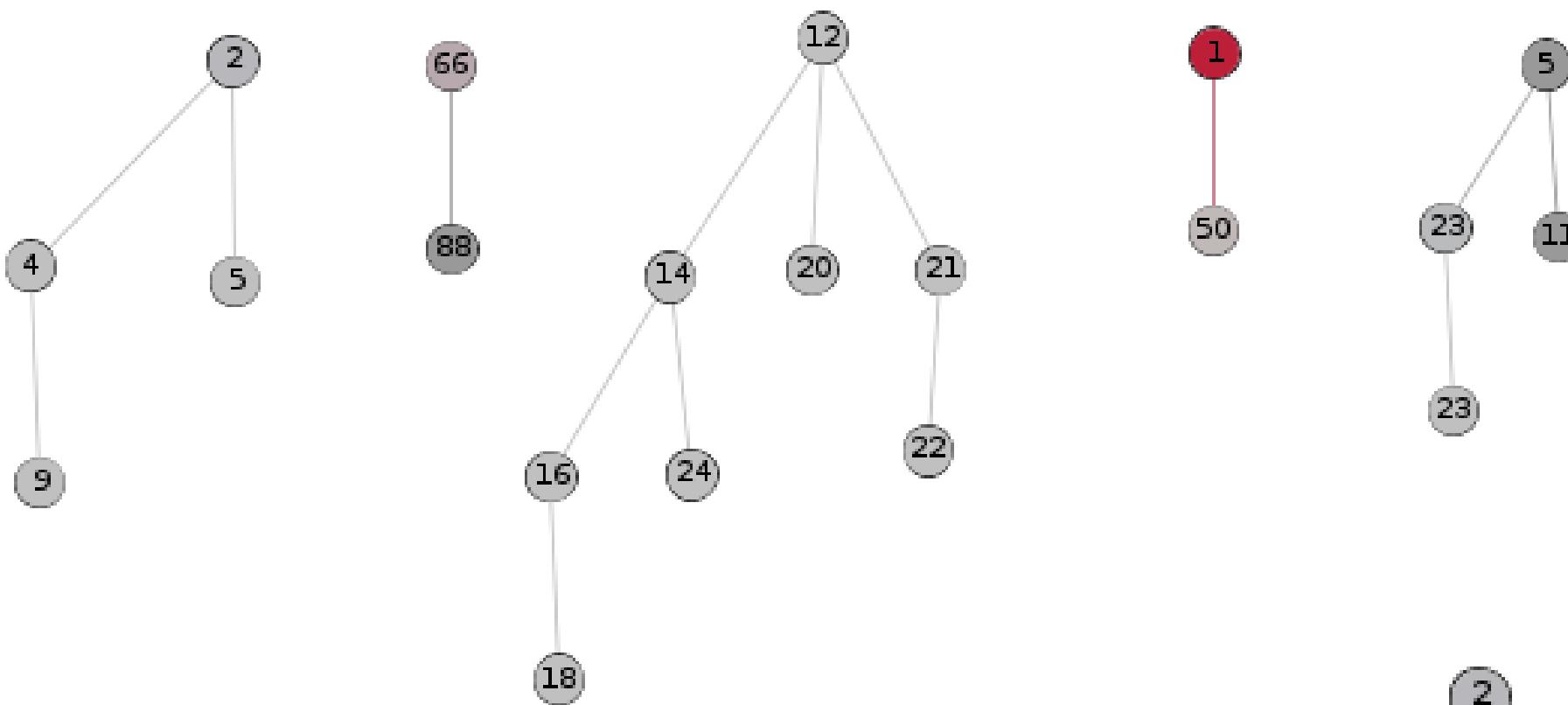


Unde e problema?

Extragem minimul. Fiii săi devin arbori liberi.



Extragere minim



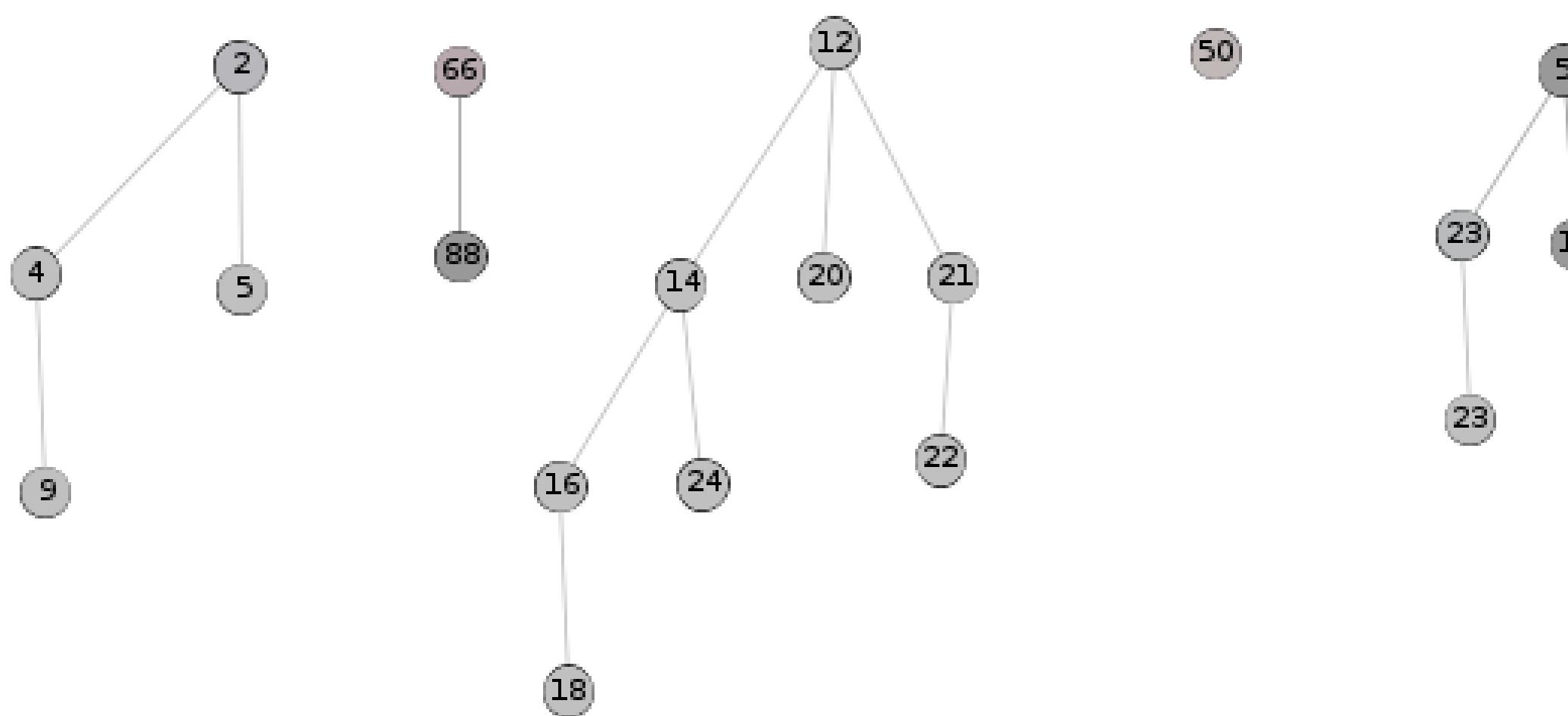
Unde e problema?

Nu știm care e minimul.

Am putea avea **n** arbori cu 1 element.

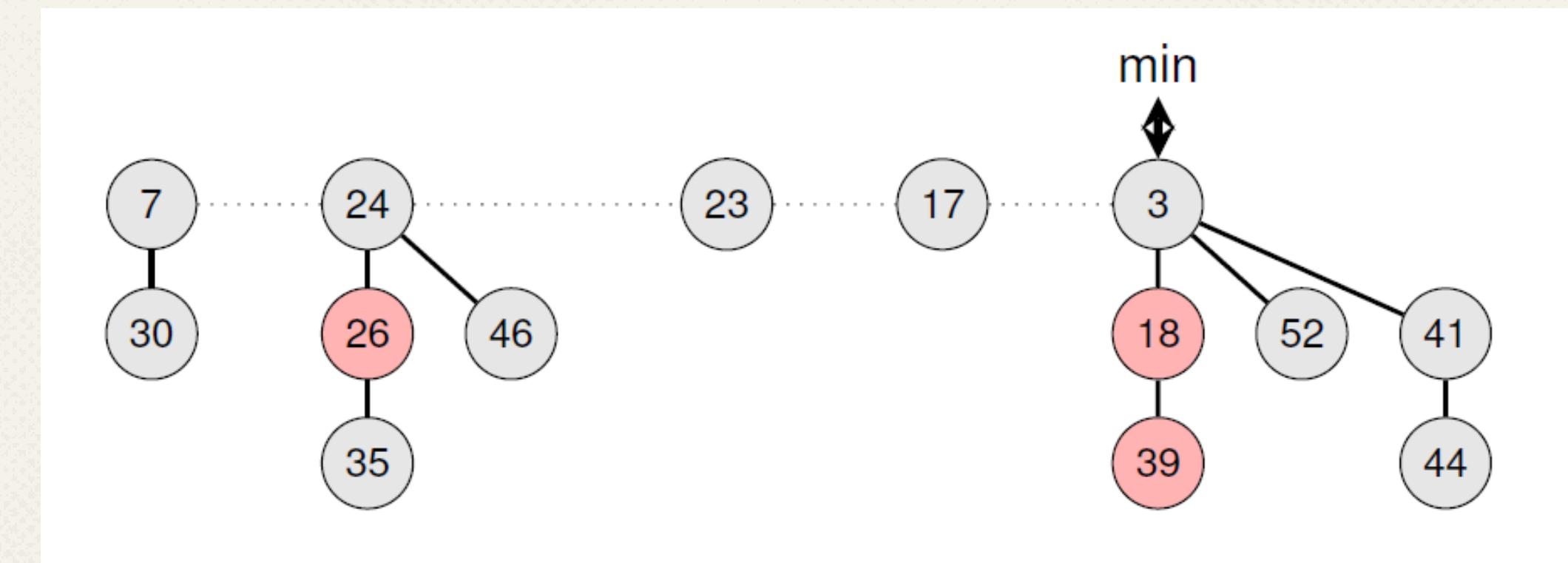
Dacă ștergem **n** elemente consecutive ne poate costa $n^2??$

Extragem minim. Fiii săi devin arbori liberi.



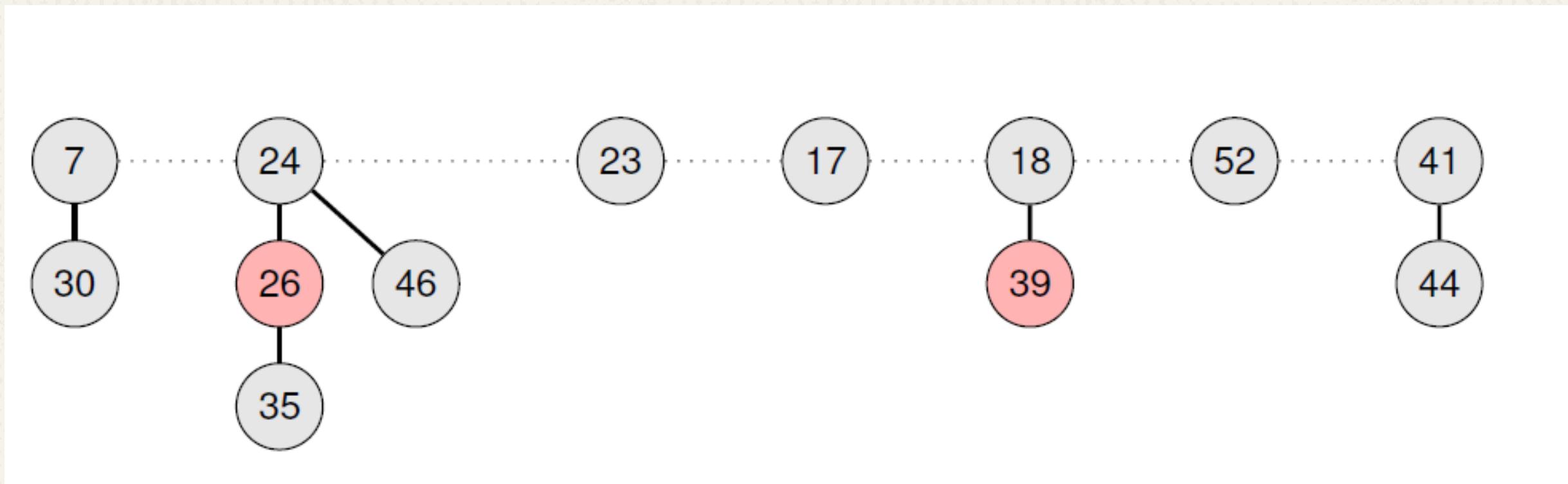
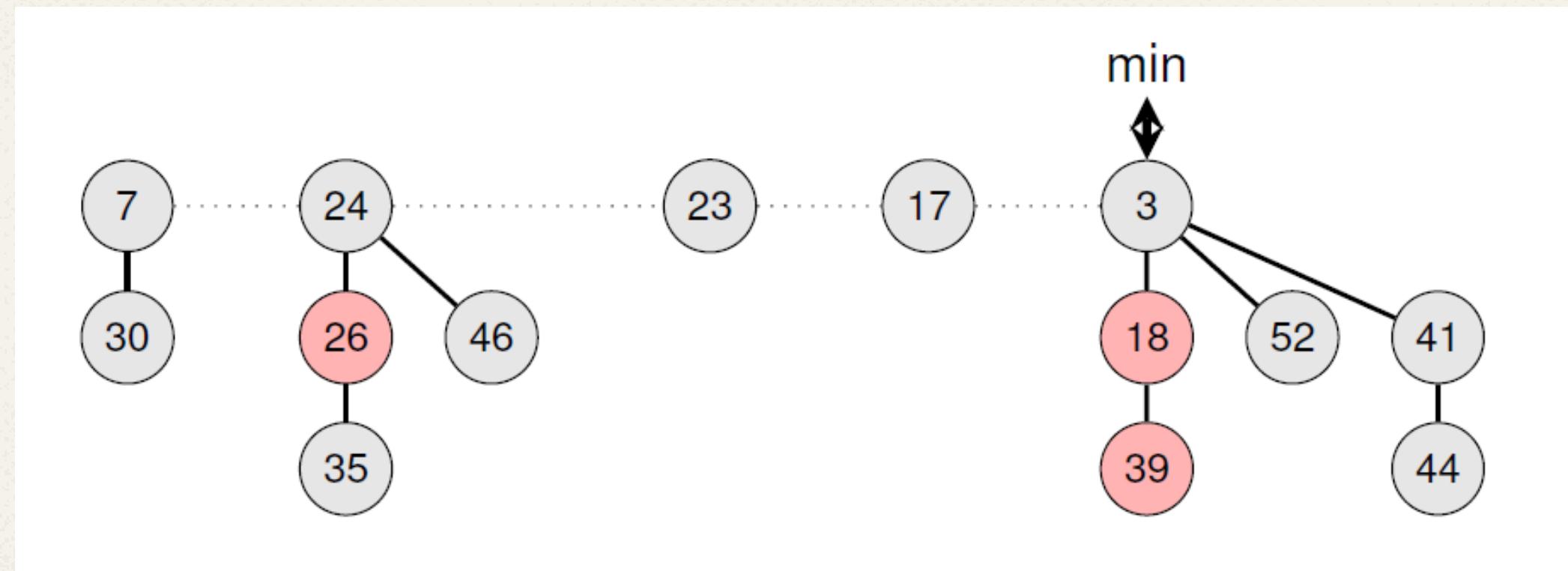
Extragere minim

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



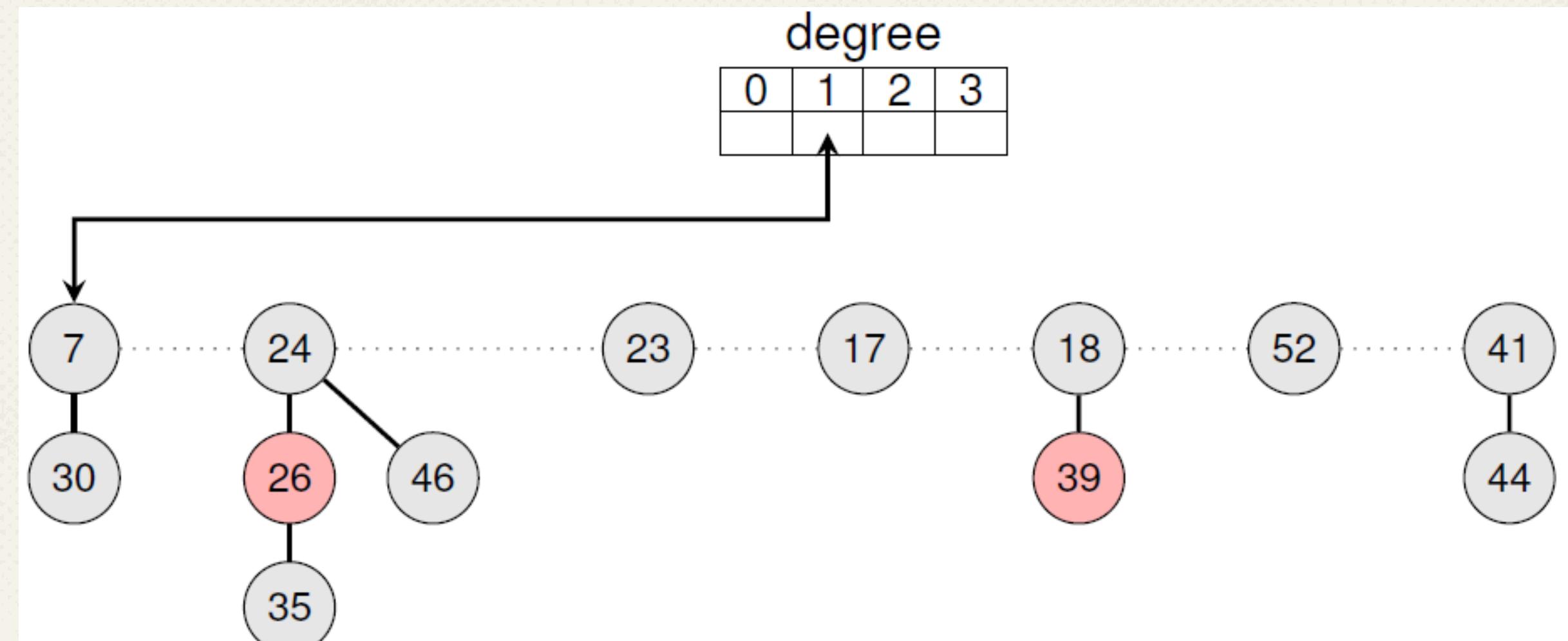
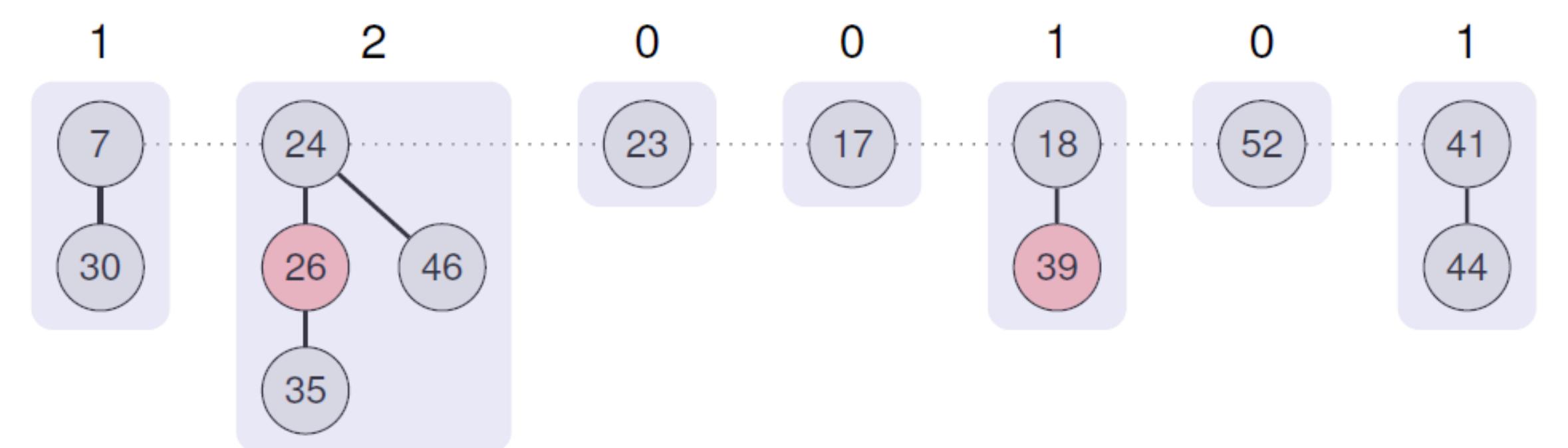
Eliminăm minimul, se creează multe “rădăcini”

Extragere minim

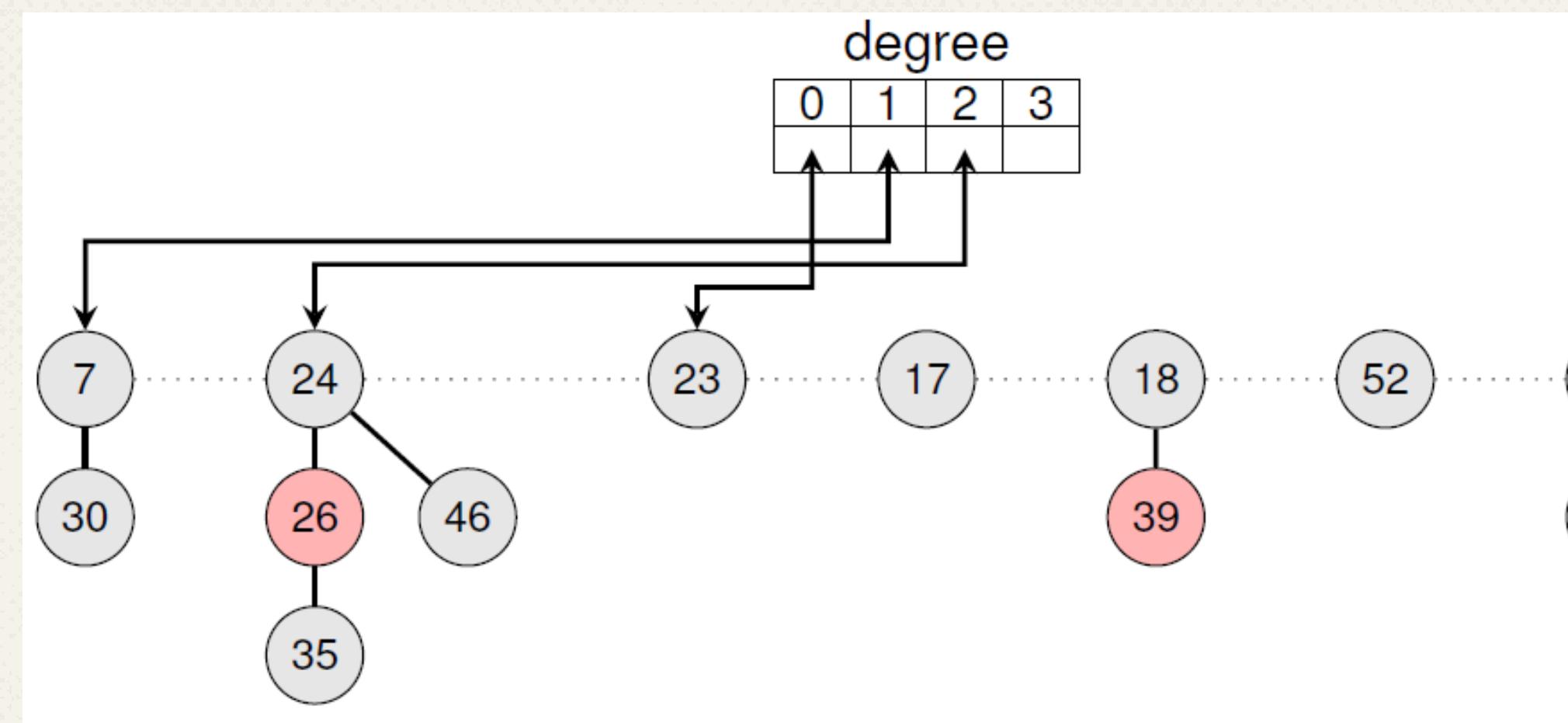
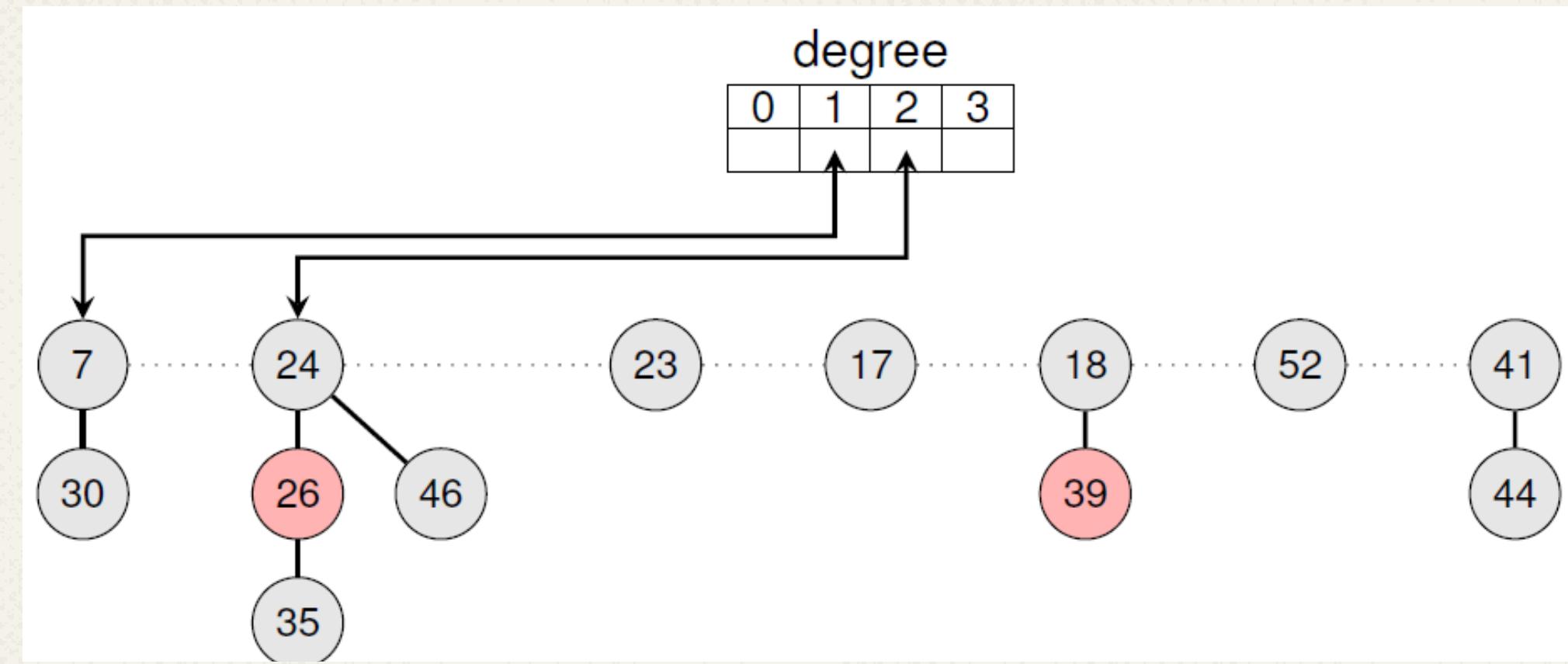


Extragere minim

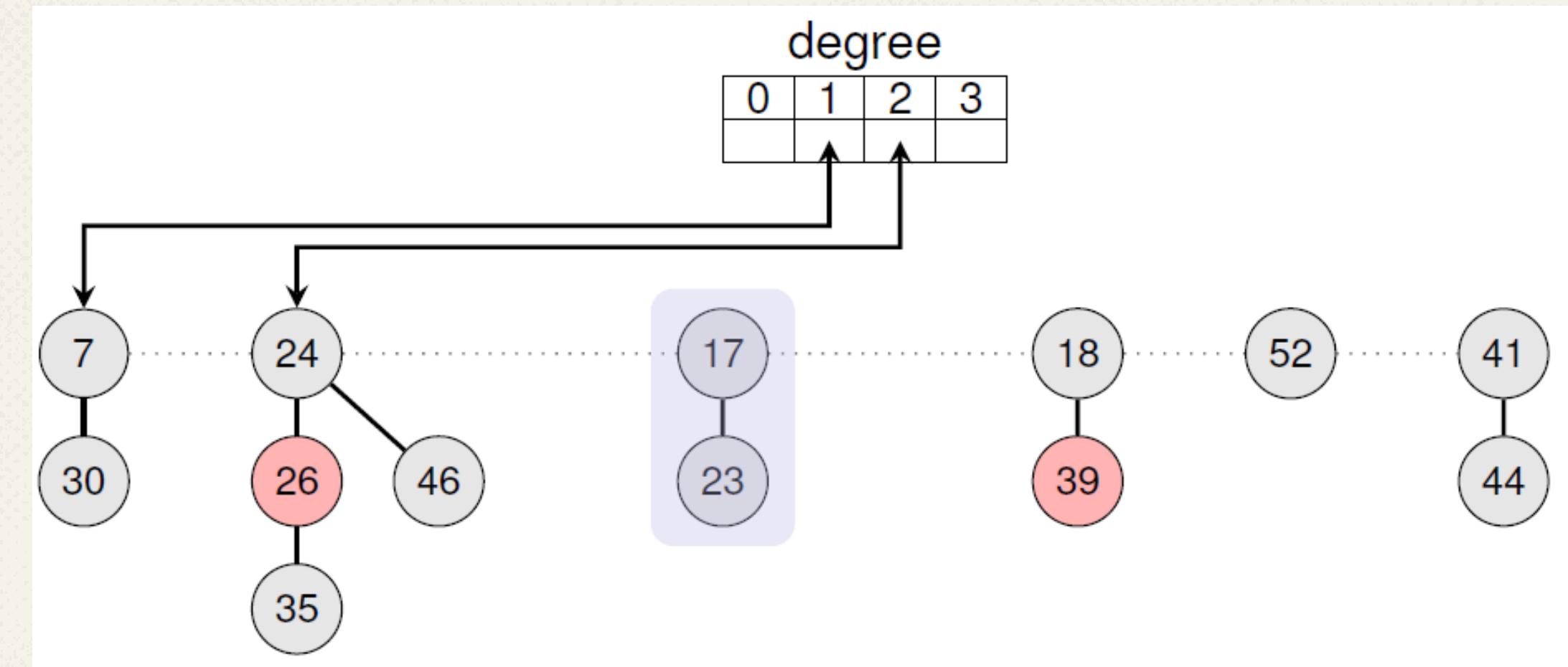
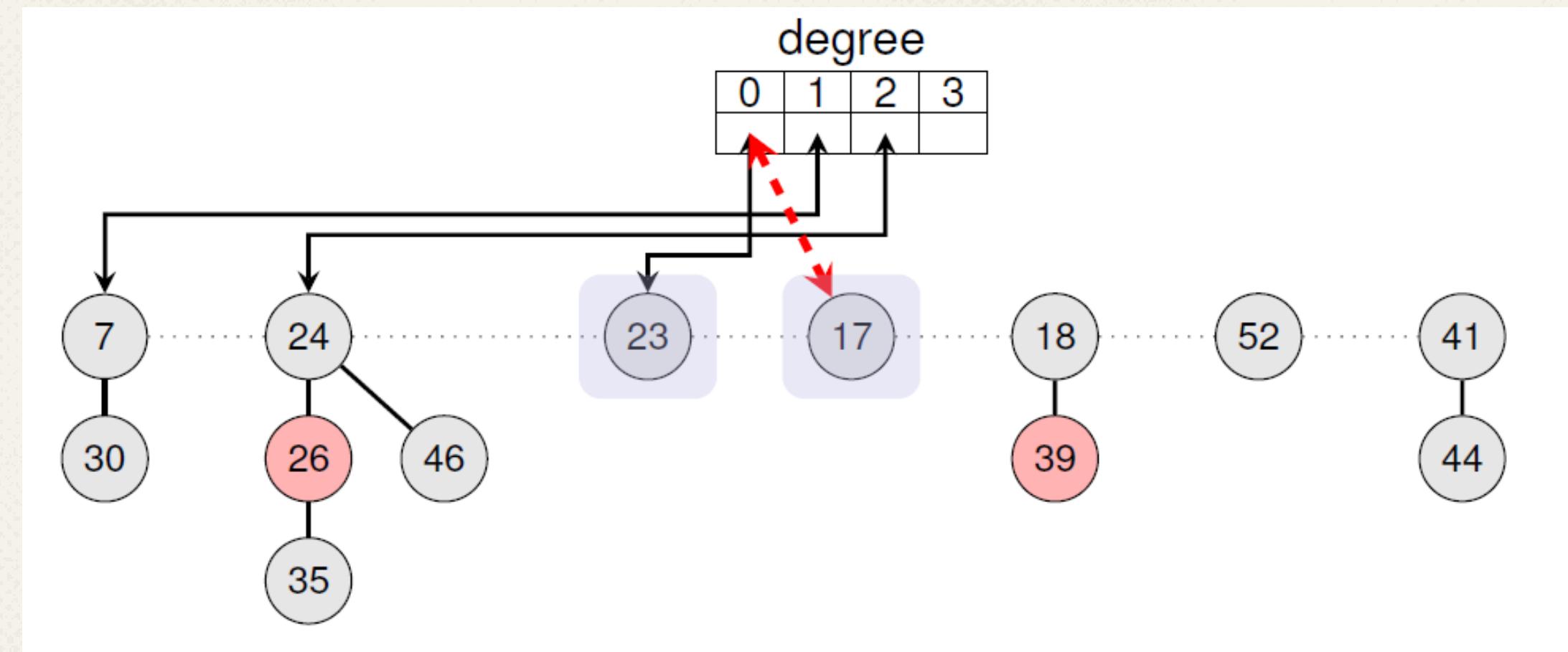
grad =



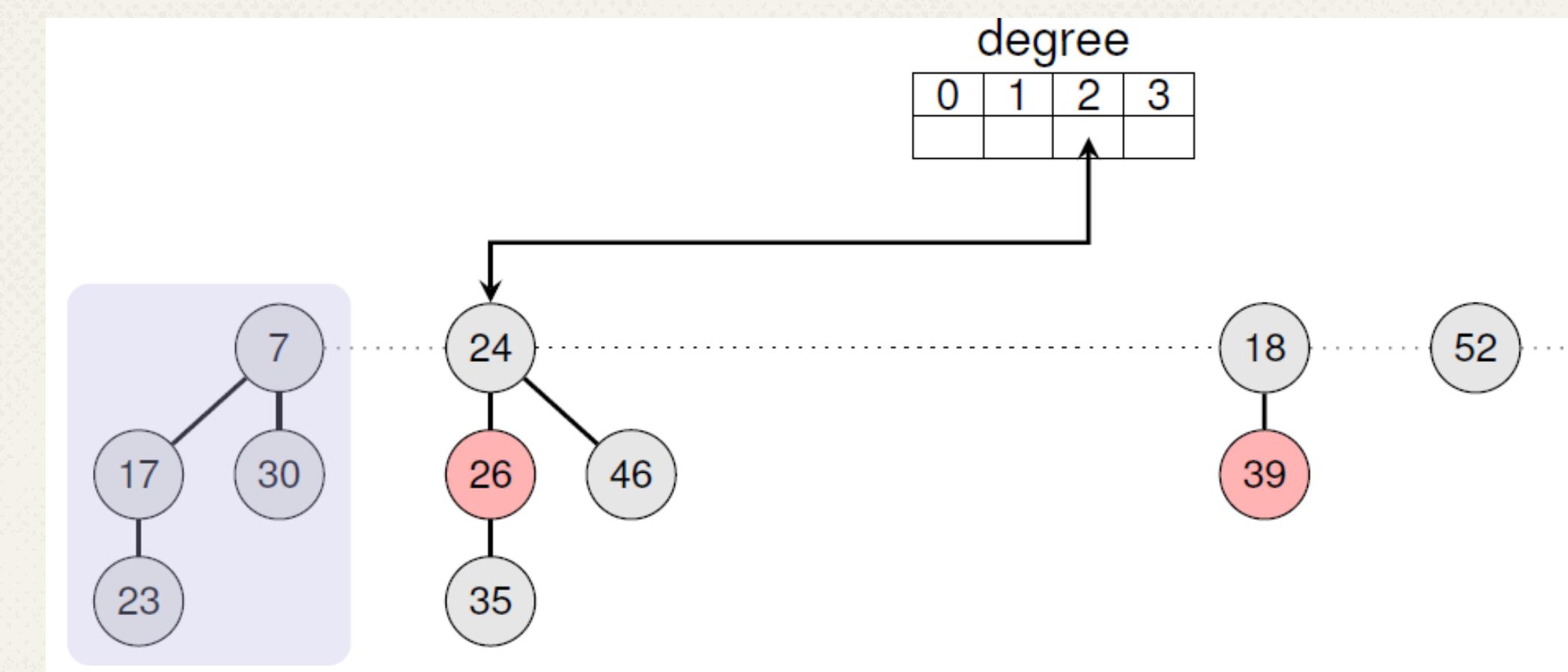
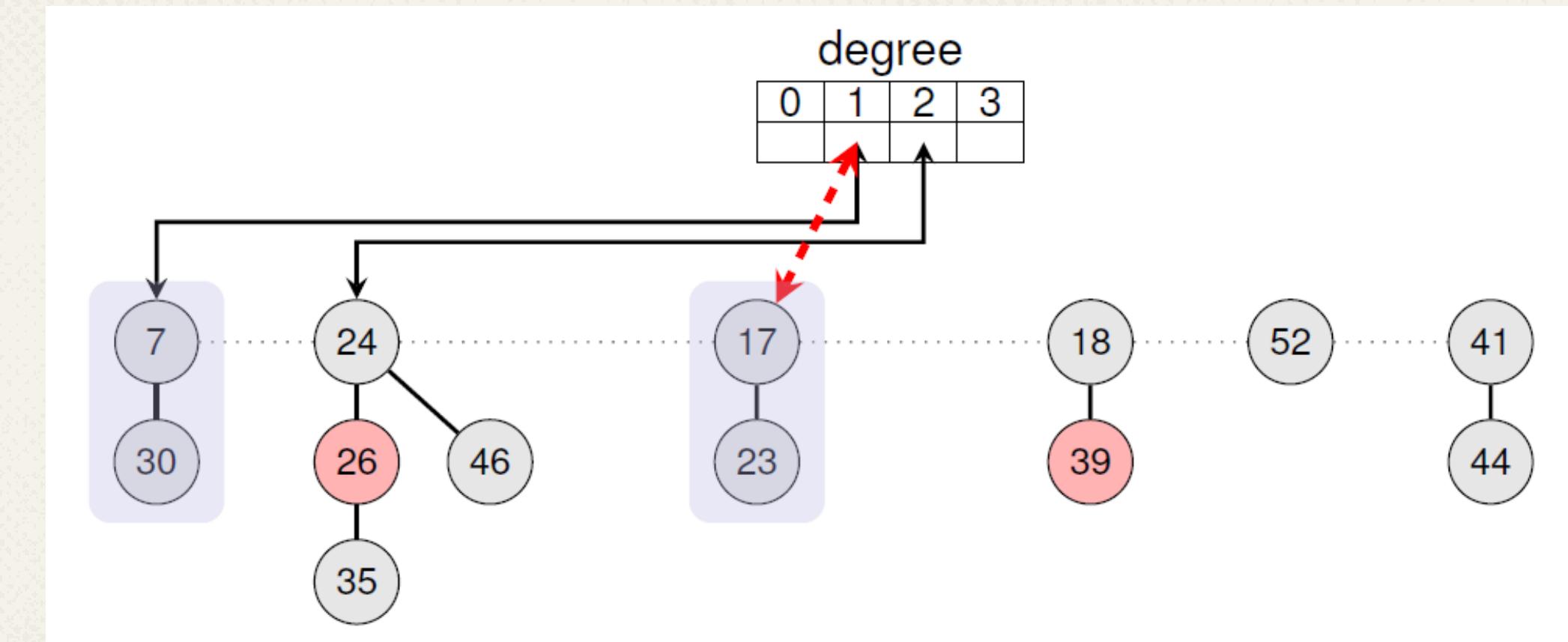
Extragere minim



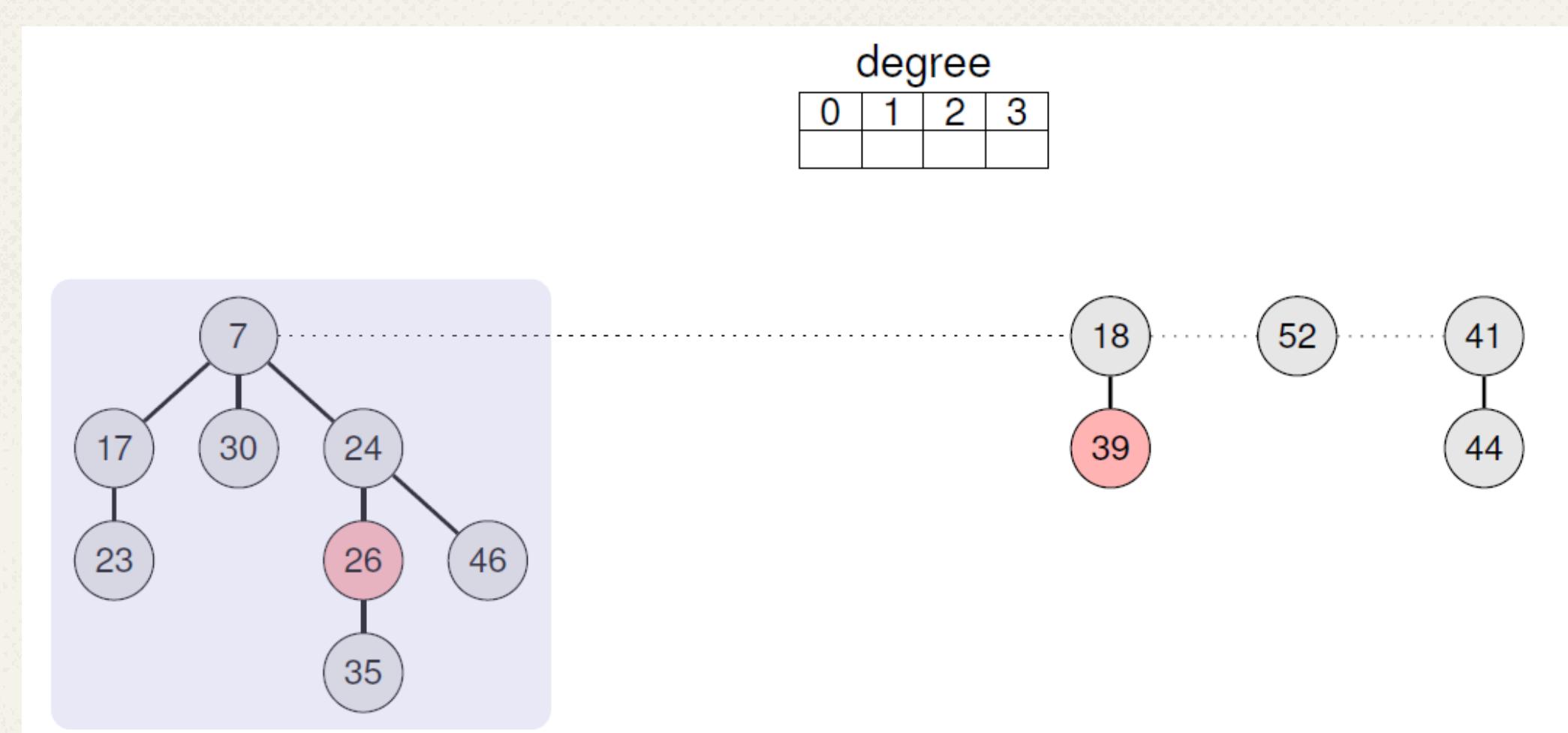
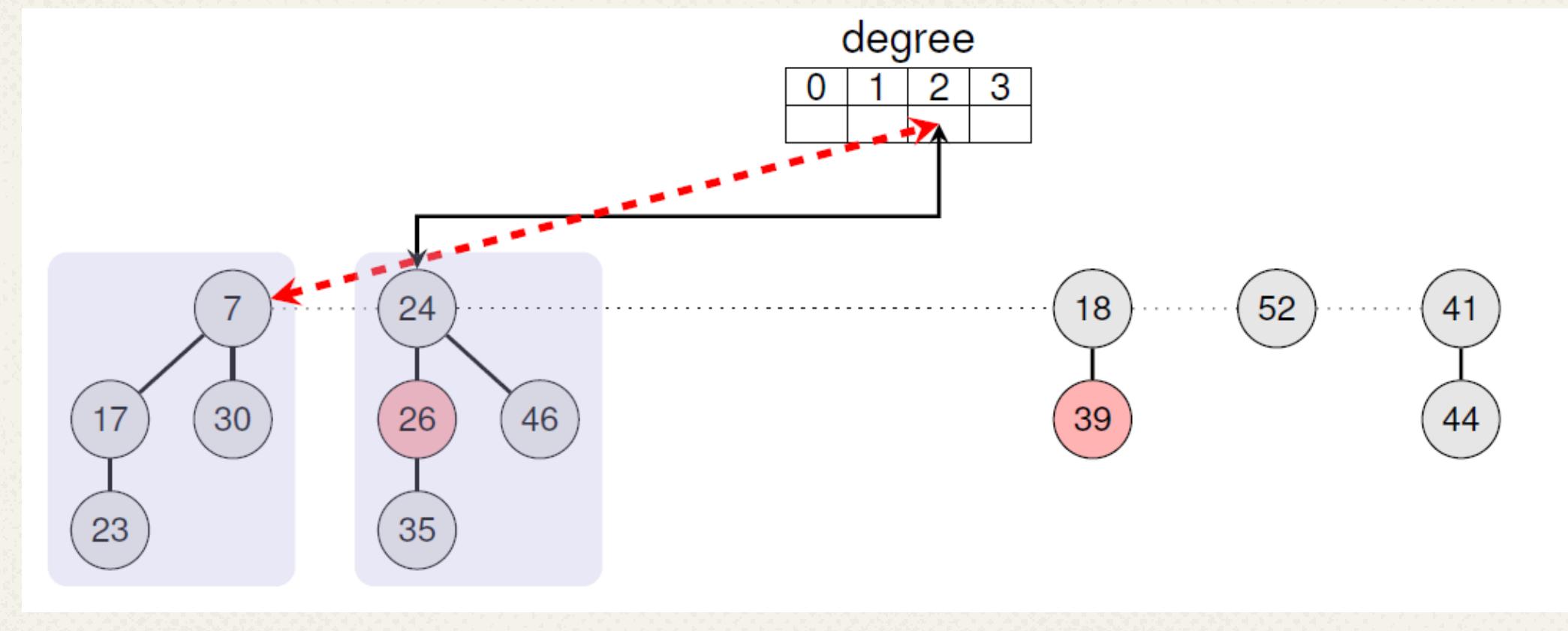
Extragere minim



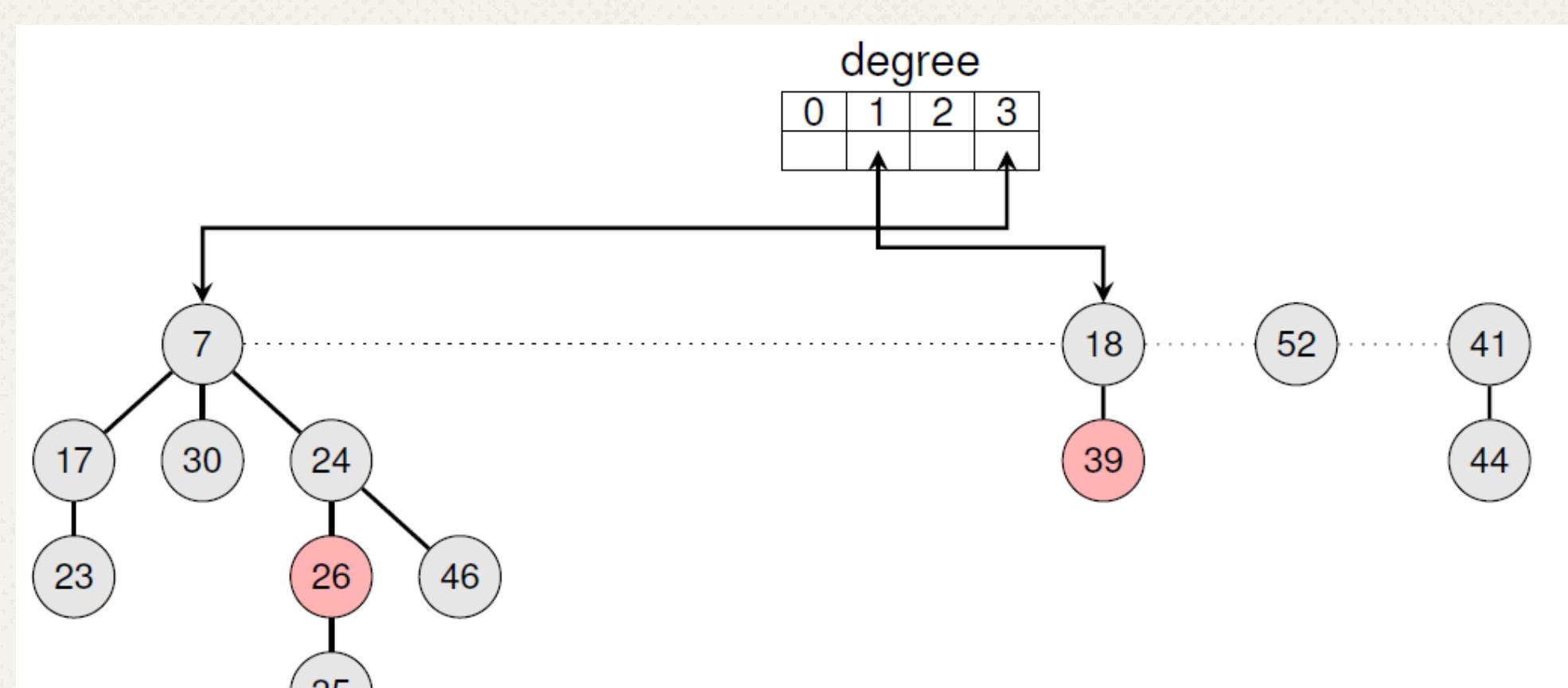
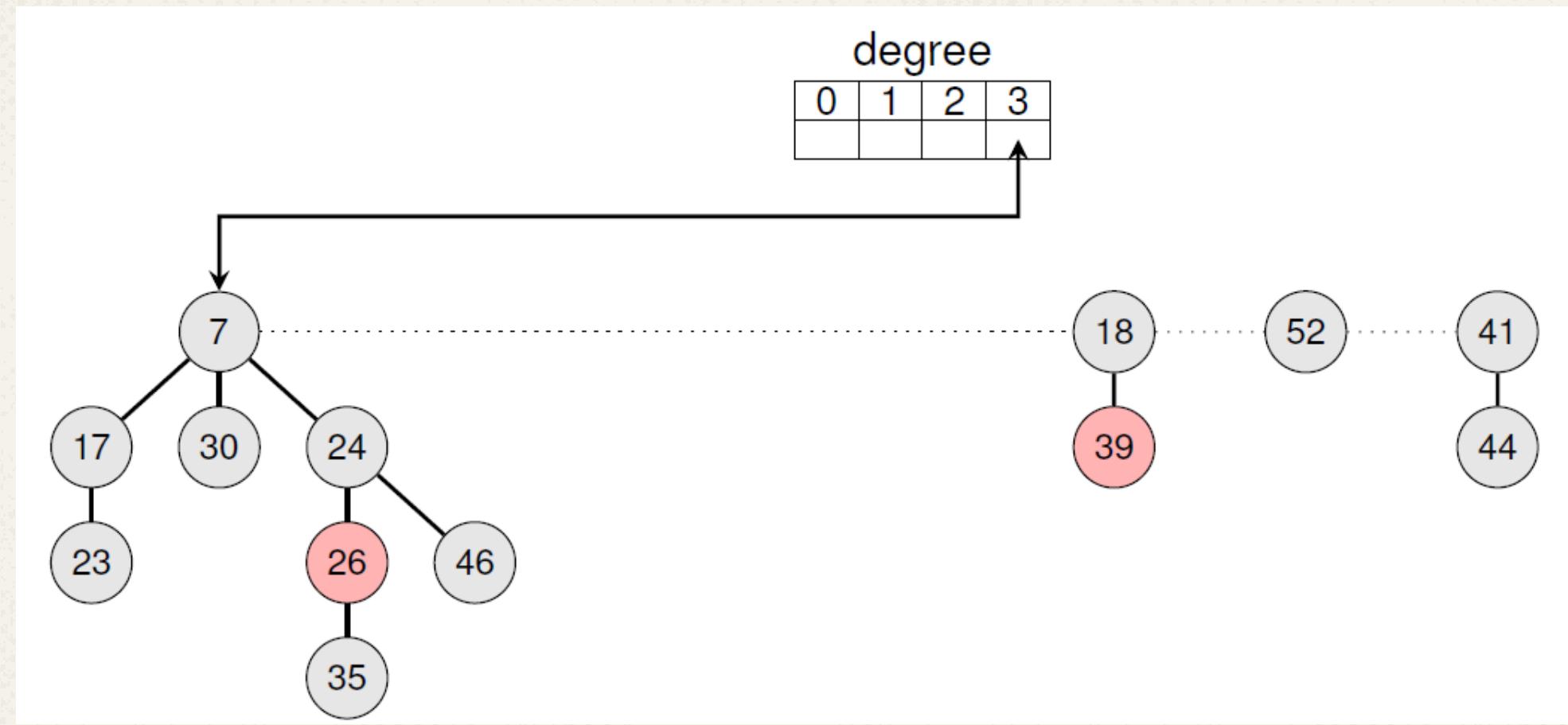
Extragere minim



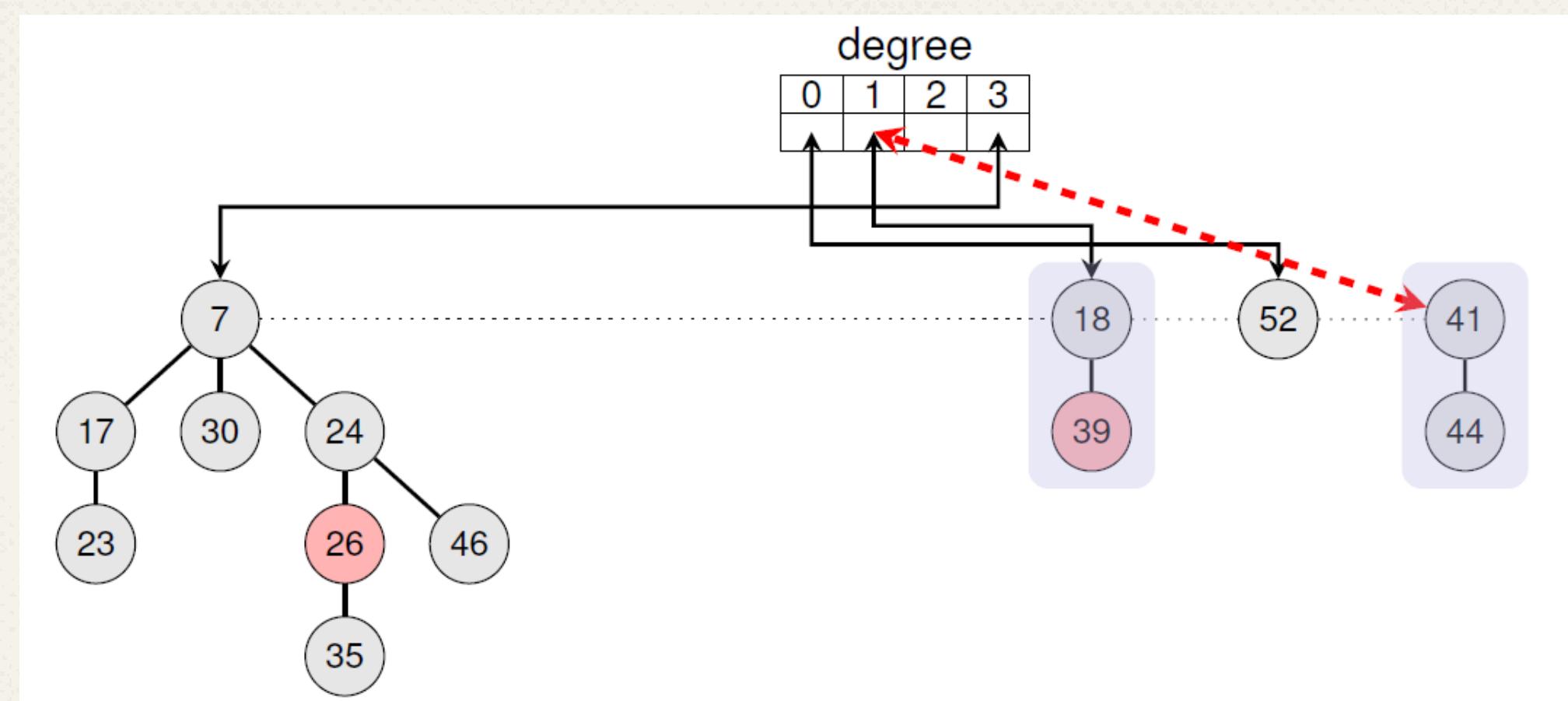
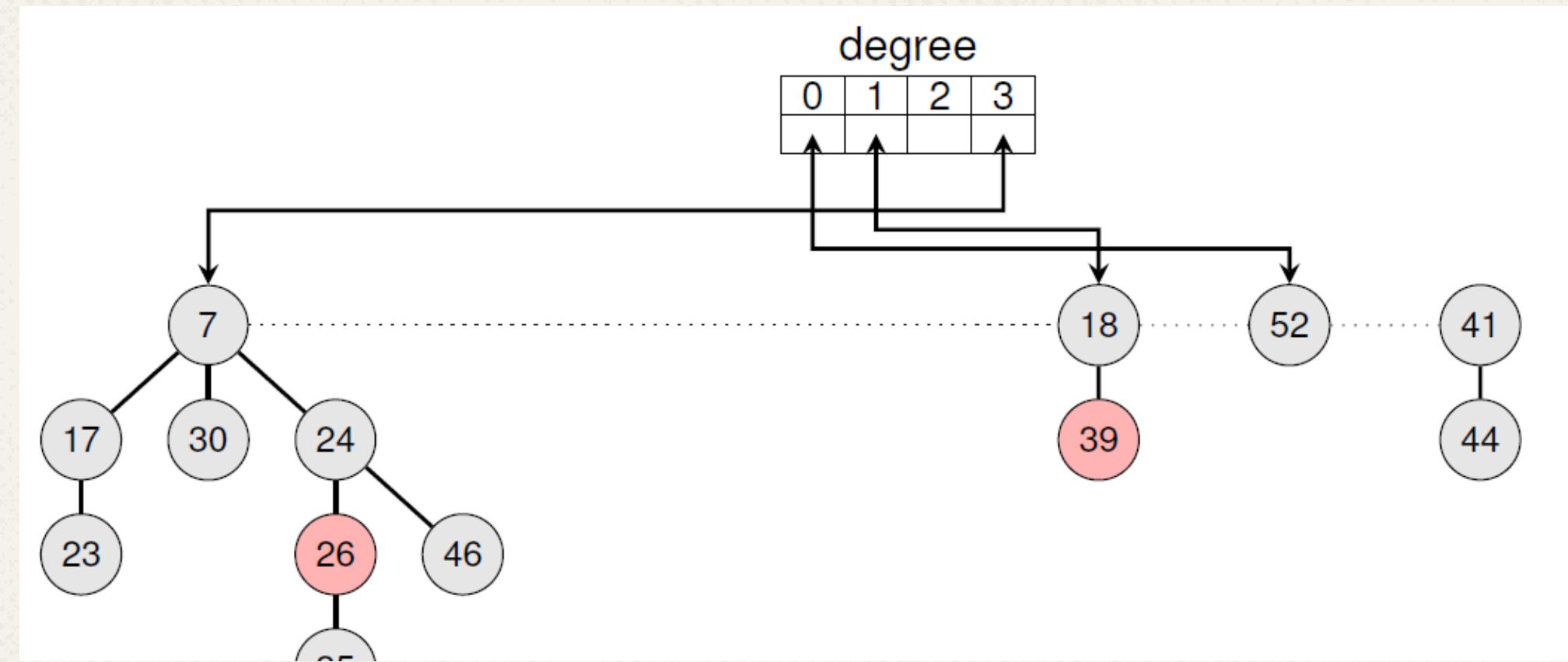
Extragere minim



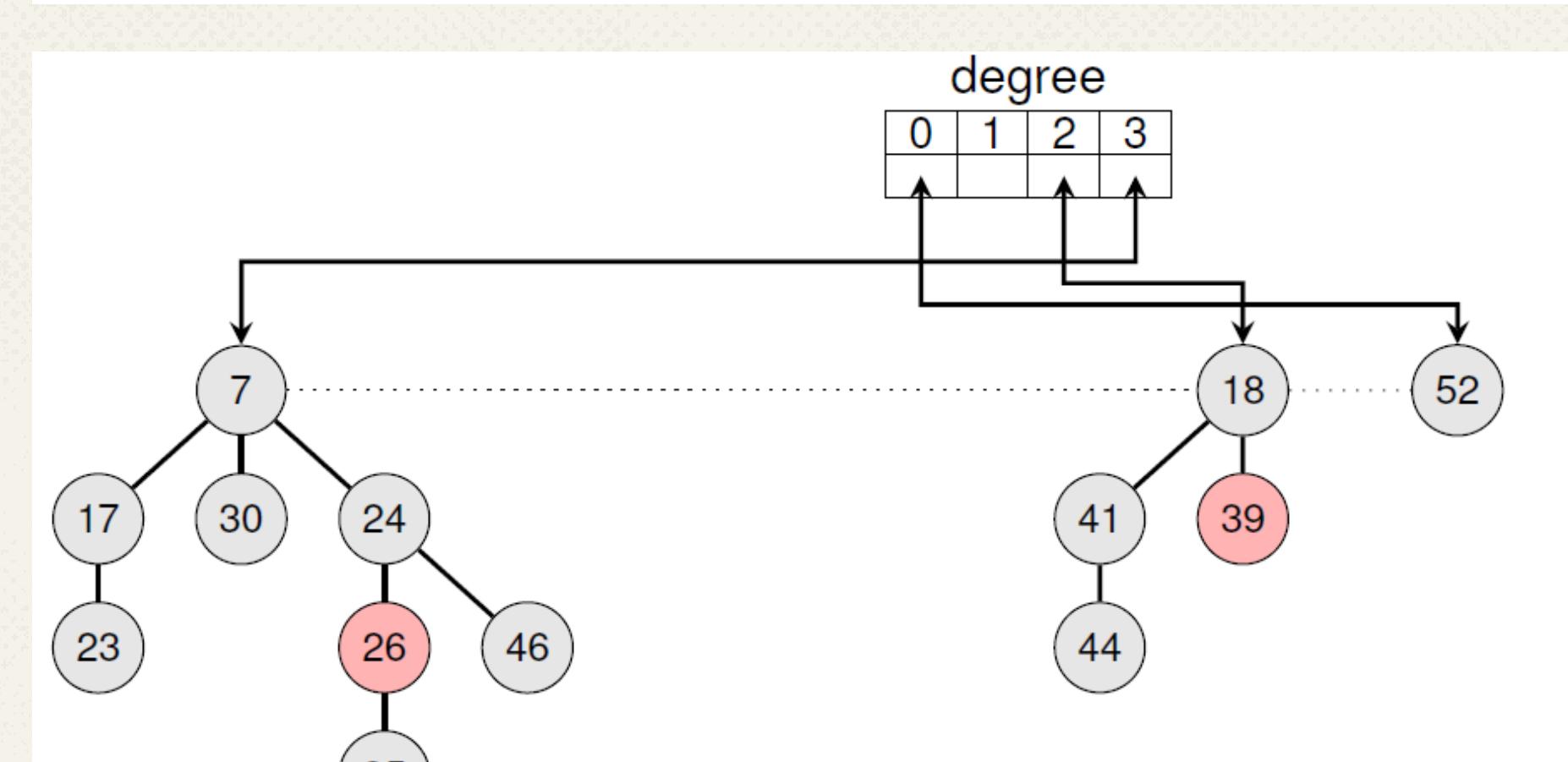
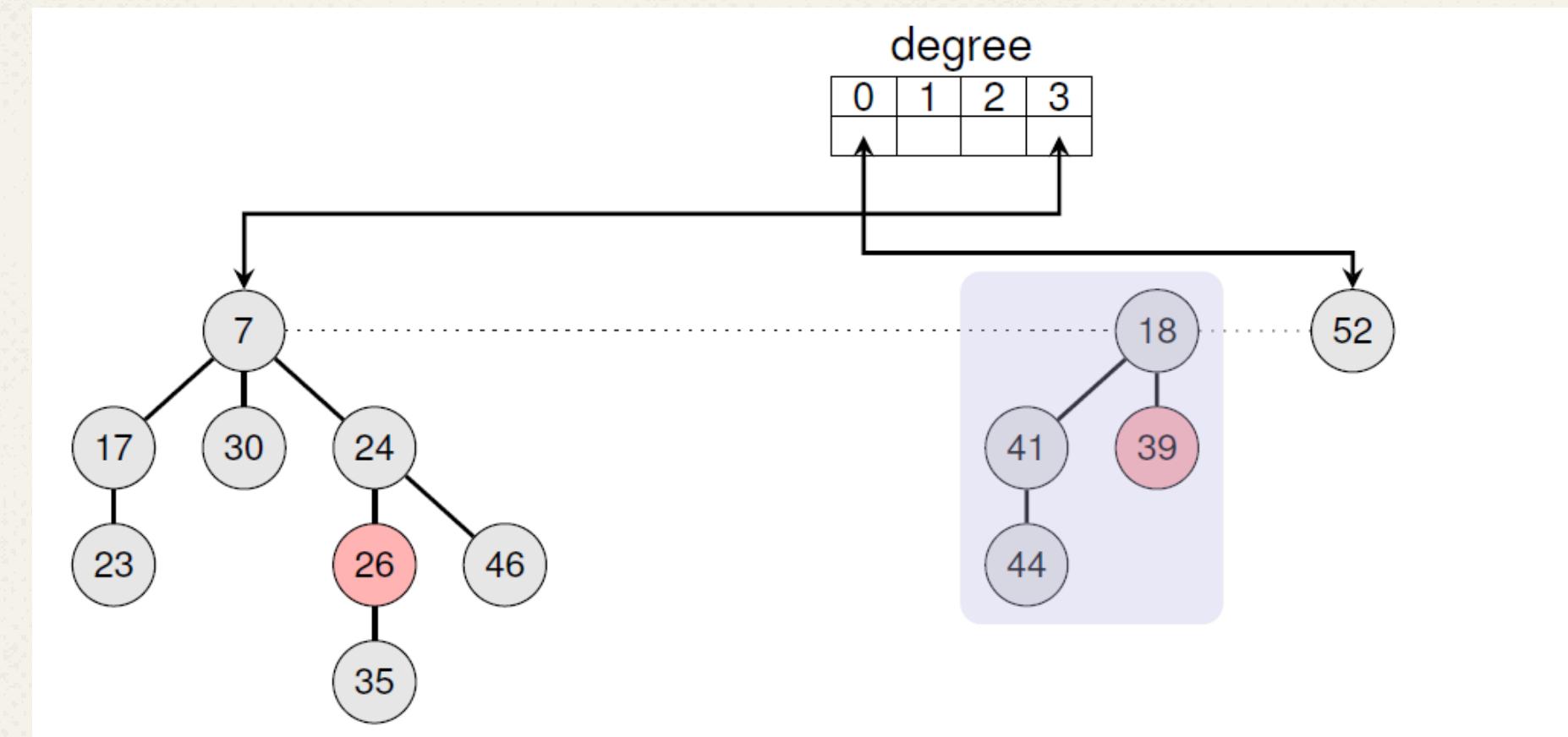
Extragere minim



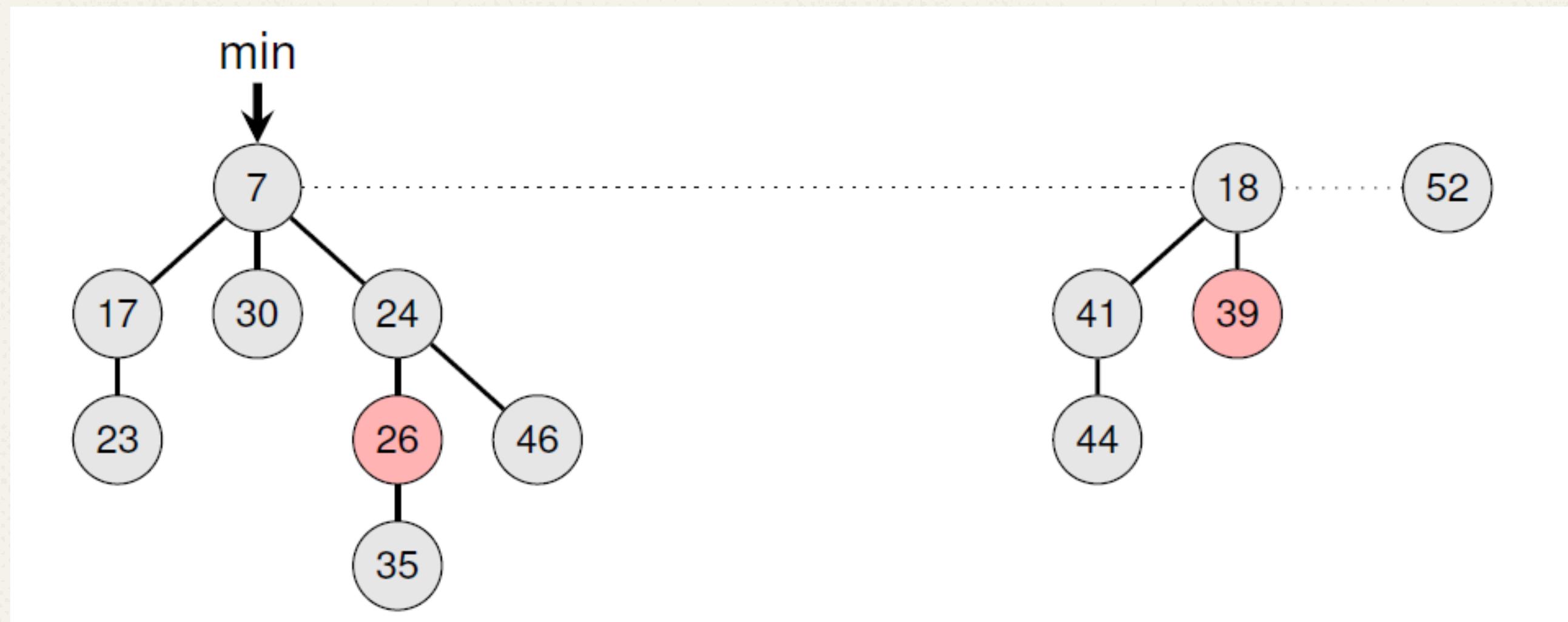
Extragere minim



Extragere minim



Extragere minim



Extragere minim – Partea I

```
void Extract_min()
{
    if (mini == NULL) // Verificăm dacă heap-ul este gol
        cout << "Heap-ul este gol" << endl;
    else {
        node* temp = mini;
        node* pntr;
        pntr = temp;
        node* x = NULL;

        // Verificăm dacă nodul minim are copii
        if (temp->child != NULL) {
            x = temp->child; // Iterăm prin toți copiii nodului minim
            do {
                pntr = x->right;
                (mini->left)->right = x; // Rearanjăm legăturile pentru a include copiii în lista de rădăcini
                x->right = mini;
                x->left = mini->left;
                mini->left = x;
                if (x->key < mini->key) // Actualizăm nodul minim, dacă este necesar
                    mini = x;
                x->parent = NULL;
                x = pntr;
            } while (pntr != temp->child);
        }
    }
}
```

Extragere minim – Partea II

```
// Ștergem nodul minim din lista de rădăcini
    (temp->left)->right = temp->right;
    (temp->right)->left = temp->left;

// Actualizăm nodul minim
mini = temp->right;
if (temp == temp->right && temp->child == NULL)
    mini = NULL;
else {
    mini = temp->right;
    // Consolidăm heap-ul
    Consolidate();
}
// Scădem numărul de noduri din heap
no_of_nodes--;
}
```

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 ati facut oficial, nu? e timp :))

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

Problemă

Interclasarea optimală a mai multor siruri.

Ex: 3 şiruri de lungimi 10, 40 şi 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: 190

Interclasarea lui 40 cu 90 → mă costă 130. $130 + 10 \rightarrow 140$ Total: 270

Discuție problemă

- 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 problemă

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 în care vin (vor fi și ele sortate)

Discuție problemă

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

CODURI HUFFMAN



Coduri Huffman

- Codurile Huffman reprezintă o tehnică eficientă pentru compactarea datelor. Scopul este ca, pentru fiecare caracter, să alegem o metodă optimă pentru a scrie în binar.

	a	b	c	d	e	f	
Frequency (in thousands)	45	13	12	16	9	5	
• Care	000	001	010	011	100	101	
◦	Variable-length codeword	0	101	100	111	1101	1100

- Cum decodificăm?

Coduri Huffman

- Codurile Huffman reprezintă o tehnică eficientă pentru compactarea datelor. Scopul este ca, pentru fiecare caracter, să alegem o metodă optimă pentru a scrie în binar.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Care
- Fixed-length codeword
- Variable-length codeword
- Cum decodificăm?
 - Primul caz e simplu: fiecare 3 caractere sunt o literă
 - În al doilea caz, trebuie ca nicio codificare să nu fie prefix al altrei codificări.

Coduri Prefix

Codificările în care nicio codificare nu este prefix al altrei codificări se numesc prefix. Se poate demonstra că, pentru o compresie optimă a datelor, există tot timpul și o codificare prefix. Prin urmare, vom căuta o codificare prefix optimă.

Codificare și decodificare

Textul *adefa* devine

0111110111000

Codificarea este ușoară, trebuie să înlocuim fiecare literă cu codul ei.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	3	1
Fixed-length codeword	000	001	010	011	10	11
Variable-length codeword	0	101	100	111	11	

Coduri Prefix

Textul *a**d**e**f**a* devine 011110111000

Decodificarea:

011110111000 → șirul începe cu 0 și doar *a* începe cu 0 → prima literă e *a*

111101111000 → șirul începe cu 111 și doar *d* începe cu 111 → a doua literă e *d*

....

Cum găsim litera următoare optim?

	a	b	c	d
Frequency (in thousands)	45	13	12	16
Fixed-length codeword	000	001	010	011
Variable-length codeword	0	101	100	111

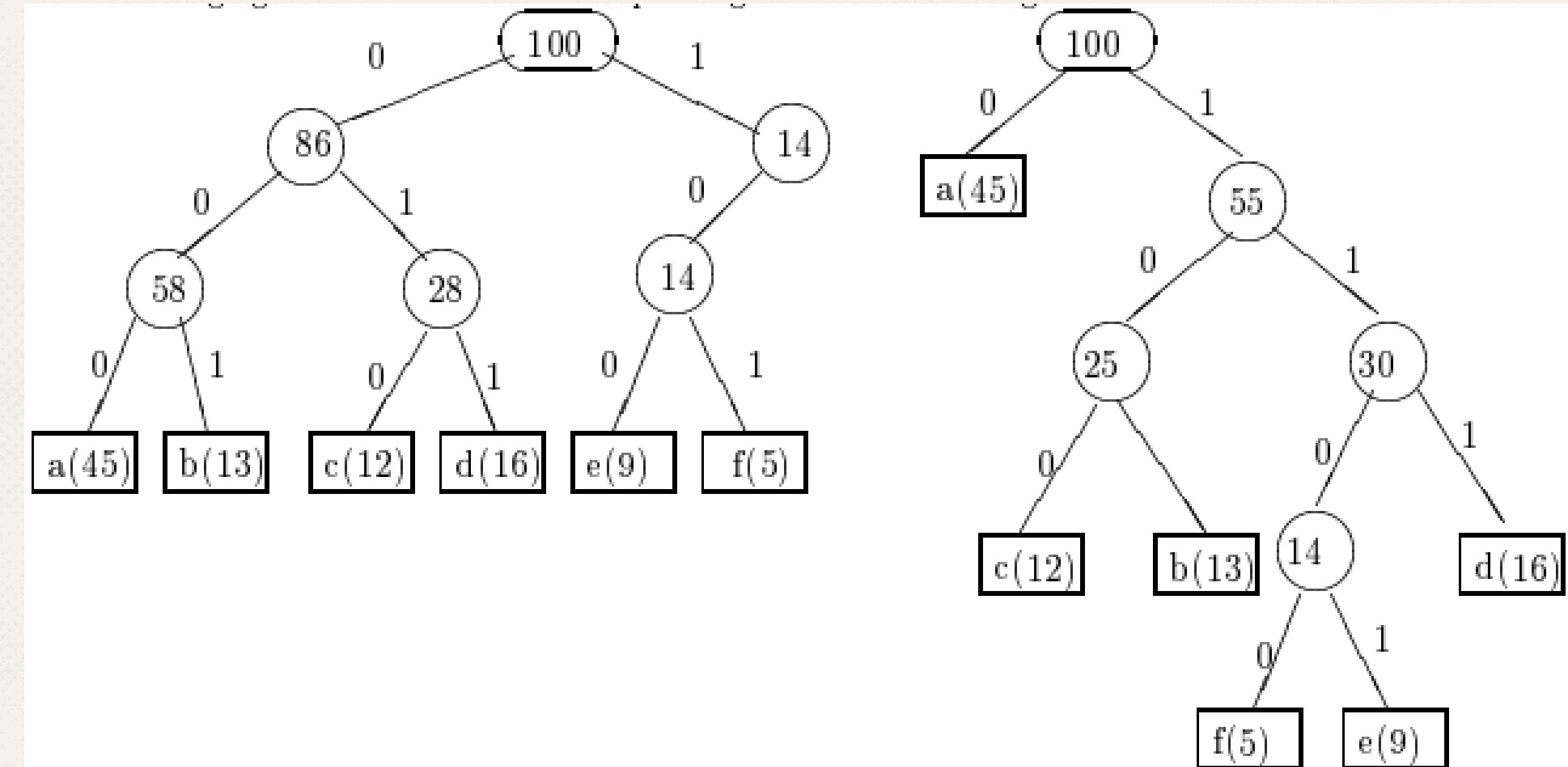
Coduri Prefix - Arbori de Codificare

(Trie like)

0111110111000

0 → 111 → 1101...

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



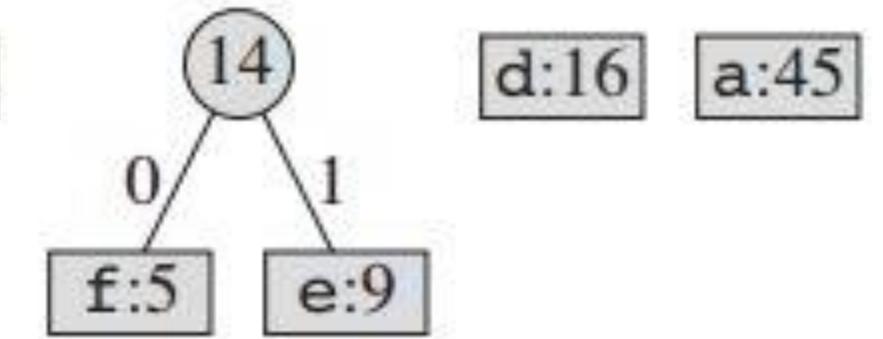
Coduri Huffman - Construcție

La fel ca în temă :)

Unim, de fiecare dată, cele mai mici 2 valori.

(a) [f:5] [e:9] [c:12] [b:13] [d:16] [a:45]

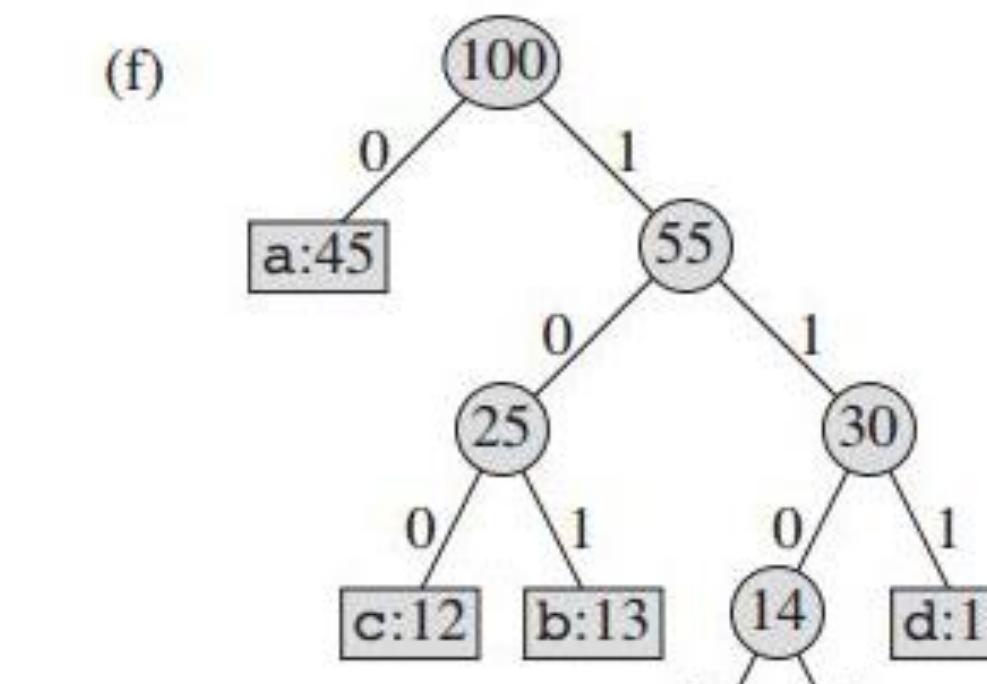
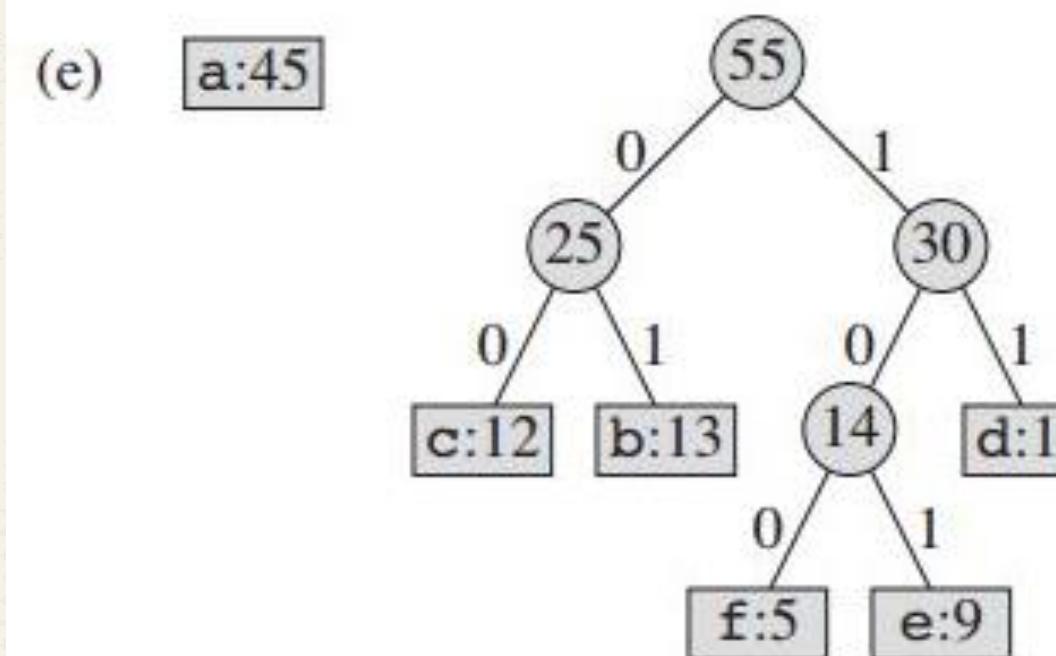
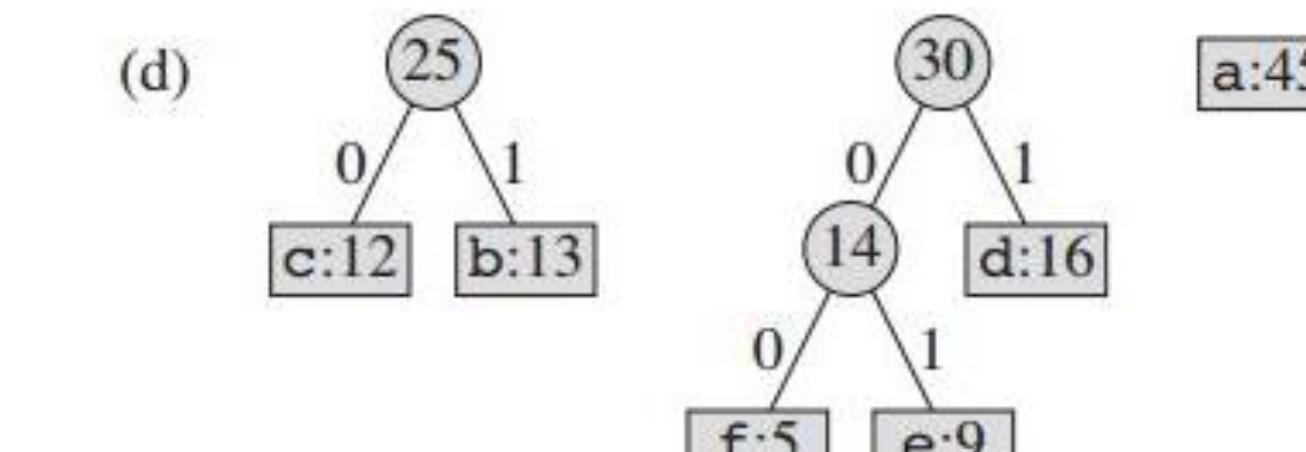
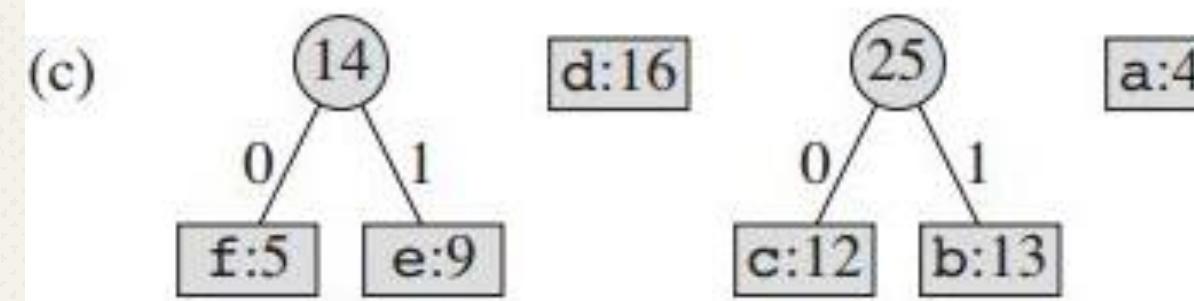
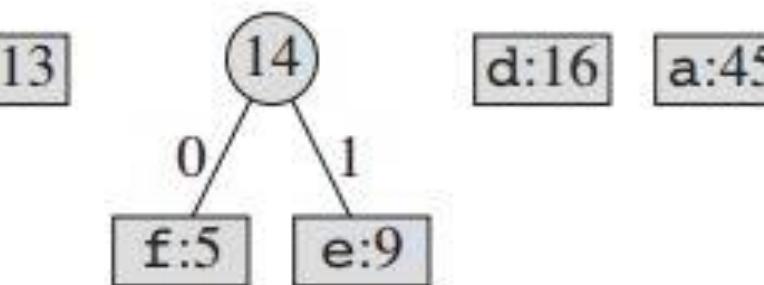
(b) [c:12] [b:13] [d:16] [a:45]



Coduri Huffman - Construcție

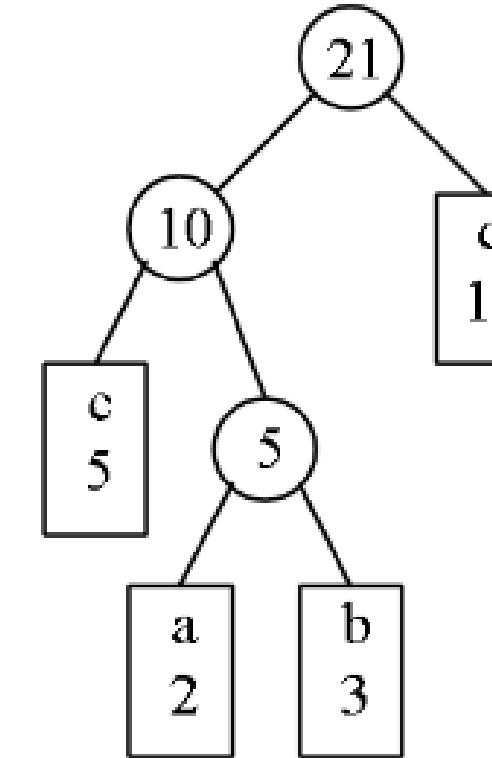
(a) f:5 e:9 c:12 b:13 d:16 a:45

(b) c:12 b:13 d:16 a:45

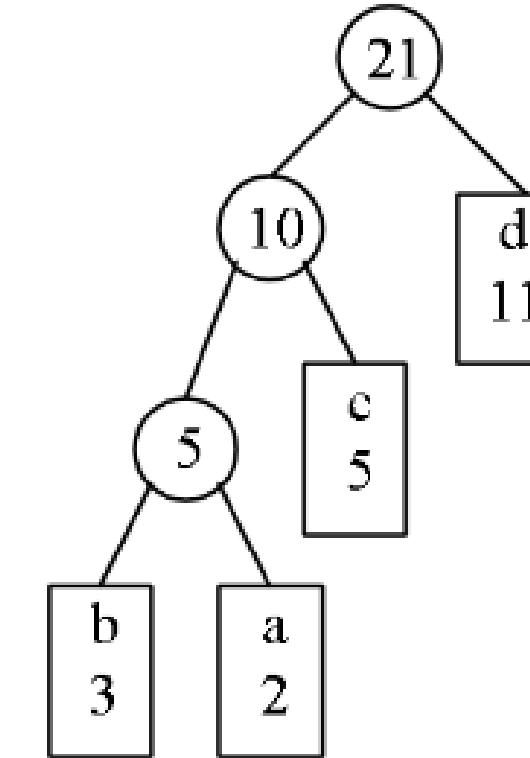


Coduri Huffman

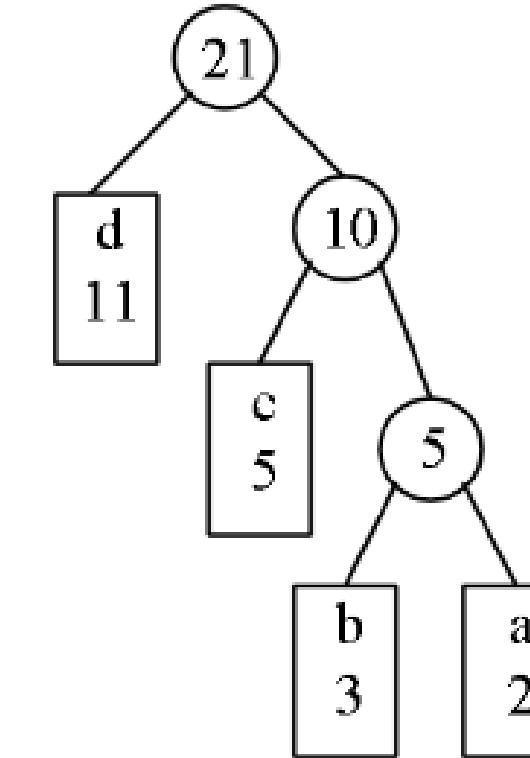
Nu e unic!



(C) L=36



(D) L=36



(E) L=36

Arborei Huffman pt. frunzele cu ponderi $\{(a, 2), (b, 3), (c, 5), (d, 11)\}$.

L = 36 minima.

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/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

Final



Temă (Partial optională)

Deadline: Cursul 8 → după vacanță

- Demonstrați că un arbore binar, care nu este complet, nu poate corespunde unei codificări prefix optime.
- **Optional:** Optimalitate Huffman + prezentare scurtă (5-10 minute)
- Stabiliți o codificare Huffman optimă pentru primele 8 numere din sirul lui Fibonacci (1, 1, 2, 3, ...)
- <https://www.infoarena.ro/problema/huffman> (cu heapuri, nu cu 2 cozi) → 70 de puncte