

---

# Structuri de date

Lect. Dr. Marius Dumitran

---

# Organizatorice

- Notare
- Laboratoare / laboranți
- Curs live

# Notare

- 40% laborator
  - **Nota minim 5!!**
- 20% seminar
  - Prezență, activitate și teme
- 40% examen
  - Examen scris (10-11 iunie ?)
  - **Nota minim 5!!**
- 10% Kahoot
  - Vom face teste uneori (probabil)... vedem cum e cu hibridul (azi vom avea exemplu)
- Media minim 4:50

# Notare

- 40% laborator
  - **Nota minim 5!!**
  - Formatul urmeaza sa fie definit (nu sunt toti laboranzii definiti)
  - Nota laborator + bonus maxim 1p de la laborant (bonusul îl pot primi doar cei care au punctaj din teme)

# Curs live/online + schimbari seminar

- Eu nu trebuia sa tin cursul semestrul asta , a aparut o modificare de ultim moment.... Prin urmare primele saptamani trebuie sa facem mici modificari in orar..
- Saptamana viitoare 22 februarie face curs online de la 16 la 18, s-ar putea sa se mai intample asta la 1-3 cursuri.
- Grupa 134 Seminar Joi 16-18 saptamana asta ? + peste 2 saptamani
- La grupa 132 (poate 134) o sa vina altcineva la primele 2 seminarii

# Overview al materiei

- Curs 1-2 Sortări/Căutare binară
  - count sort, radix sort, quick sort, merge sort
- Curs 3 Vectori/Liste Înlănțuite
  - Cozi
  - Stive
  - Deque
- Curs 4 Heapuri
- Curs 5 Heapuri binomiale - fibonacci
- Curs 6 Huffman
- Curs 7 Arbori binari de căutare
- Curs 8 AVL / Red black
- Curs 9 Skip Lists / Treaps
- Curs 10 Arbori de intervale
- Curs 11 RMQ & LCA & LA
- Curs 12-13 Hashuri
- Curs 14 Tries / Suffix trees ?

# Overview al materiei

- Curs 1-2 Sortări/Căutare binară
  - count sort, radix sort, quick sort, merge sort
- Curs 3 Vectori/Liste înlănțuite
  - Cozi
  - Stive
  - Deque
- Curs 4 Heapuri
- Curs 5 Heapuri binomiale - fibonacci
- Curs 6 Huffman
- Curs 7 Arbori binari de căutare
- Curs 8 AVL / Red black
- Curs 9 Skip Lists / Treaps
- Curs 10 Arbori de intervale
- Curs 11 RMQ & LCA & LA
- Curs 12-13 Hashuri
- Curs 14 Tries / Suffix trees ?

Propuneri ?



# Algoritmi de sortare

Ce algoritmi de sortare cunoașteți?

# Algoritmi de sortare

Ce algoritmi de sortare cunoașteți?

- Bubble O( $n^2$ )
- Merge O( $n \log n$ )
- Interschimbare O( $n^2$ )
- Radix
- Quick O( $n \log n$ )?
- Heap O( $n \log n$ )
- Bucket Sort
- Count Sort
- Bogo Sort O( $n! * n$ )
- Gravity Sort O( $n^2$ )
- Selection Sort O( $n^2$ )
- Insert sort O( $n^2$ )
- Shell Sort O( $n \sqrt{n}$ ) ~ discutabil
- Intro Sort O( $n \log n$ ) alg hibrid
- Tim Sort O( $n \log n$ ) alg hibrid

Putem grupa după:

- Complexitate
- Complexitate spațiu
- Stabilitate
- Dacă se bazează pe comparații sau nu

# Algoritmi de sortare stabili

- Un algoritm de sortare este stabil dacă păstrează ordinea elementelor egale.
- $5 \text{ } 5 \text{ } 5 \rightarrow 5 \text{ } 5 \text{ } 5$  (sortare stabilă)
- $5 \text{ } 5 \text{ } 5 \rightarrow 5 \text{ } 5 \text{ } 5$  (sortare instabilă) sau oricare alta permutare

**Atenție:** Își unii algoritmi instabili pot sorta stabil uneori, algoritmii stabili garantează asta pentru orice input.

Pentru numere naturale nu este important, dar când sortăm altfel de obiecte acest lucru poate deveni important.

# Algoritmi de sortare

## Clasificare

Elementari	Prin comparație	Prin numărare
Insertion sort → $O(n^2)$	Quick sort → $O(n \log n)$	Bucket sort
Selection sort → $O(n^2)$	Merge sort → $O(n \log n)$	Counting sort
Bubble sort → $O(n^2)$	Heap sort → $O(n \log n)$	Radix sort
	Intro sort → $O(n \log n)$	

Tabel cu sortări:

[https://en.wikipedia.org/wiki/Sorting\\_algorithm#Comparison\\_of\\_algorithms](https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms)

# Sortare prin numărare / Counting Sort

- Algoritm de sortare a numerelor întregi mici
- Presupunem că vectorul de sortat **v** conține **n** elemente din multimea  $\{0, \dots, \text{max}\}$

## IDEE:

- Creem un vector de frecvență **fr**
- Numărăm aparițiile fiecărui element din **v**
- Modificăm vectorul **fr** a.î.
  - $\text{fr}[i] = \text{numărul de elemente cu valoare} = i$
- La final, iterăm prin vectorul  $\text{fr}[i]$  și afișăm  $i$  de  $\text{fr}[i]$  ori pentru toate numerele de la 1 la max.

# Sortare prin numărare / Counting Sort

Exemplu: sortăm note

Note	5	3	2	9	4	5	1	7	10	3	9	2	5
nota	1	2	3	4	5	6	7	8	9	10			
fr	1	2	2	1	3		1		2	1			

# Sortare prin numărare / Counting Sort

Exemplu: sortăm note

Note	5	3	2	9	4	5	1	7	10	3	9	2	5
nota	1	2	3	4	5	6	7	8	9	10			
fr	1	2	2	1	3	0	1	0	2	1			

# Sortare prin numărare / Counting Sort

Exemplu: sortăm note

nota	1	2	3	4	5	6	7	8	9	10		
fr	1	2	2	1	3	0	1	0	2	1		
soluție	1	2	2	3	3	4	5	5	5	7	9	9

# Sortare prin numărare / Counting Sort

Exemplu: sortăm note

Note	5	3	2	9	4	5	1	7	10	3	9	2	5
soluție	1	2	2	3	3	4	5	5	5	7	9	9	10

# Cod

```
// Pasul 1: Crestem frecventa fiecarui element din vector:  
  
for (int i = 0; i < n; ++i)    // O(n)  
    fr[note[i]]++, maxn = max(maxn, note[i]);  
  
// Pasul 2: afisam fiecare element de atatea ori cat apare in vectorul de frecventa  
// O(maxn * n)  
// O(maxn + n)  
  
for (int i = 0; i <= maxn; ++i) {    // pana la maxn  
    for (int j = 1; j <= fr[i]; ++j) {    // worst case se duce pana la n  
        // for-ul va face n + maxn  
        cout << i << " ";    // Afisam de fix n ori  
    }  
}
```

Complexitate? Spațiu? Timp?

# Counting Sort

## Complexitate

- Timp:
  - $O(n + \max)$
- Spațiu:
  - $O(\max)$

# Counting Sort

Vizualizare:

<https://visualgo.net/bn/sorting>

# Counting Sort

Ce ne facem dacă avem de sortat numere mari...

- Până la  $10^6$  ?
- Până la  $10^{18}$  ?
- Numere care nu sunt întregi ?

# Counting Sort

Ce ne facem dacă avem de sortat numere mari...

- Până la  $10^6$ ?
  - Depinde de N, dar **Count Sort** poate fi cea mai bună opțiune...
- Până la  $10^{18}$ ?
  - Nu mai putem folosi Count Sort. Putem folosi în schimb **Radix Sort**
- Numere care nu sunt întregi ?
  - Mai greu și cu Radix Sort (nu e imposibil, dacă sunt doar 1-2 zecimale putem înmulții cu 10, 100) ... altfel putem folosi **Bucket Sort**

# Kahoot

<https://create.kahoot.it/creator/2281f10b-f400-43fe-981c-85377fd66c12>

# Final

# Sortari

Bucket, Radix, Quick, Merge, Heap ?

# Bucket Sort

- Elementele vectorului sunt distribuite în bucket-uri după anumite criterii
- Bucket-urile sunt reprezentate de elemente ale unui vector de liste înlántuite
- Fiecare bucket conține elemente care îndeplinesc aceleasi condiții

## IDEE:

- Fie **v** vectorul de sortat și **b** vectorul de buckets
- Se inițializează vectorul auxiliar cu liste (buckets) goale
- Iterăm prin **v** și adăugăm fiecare element în bucket-ul corespunzător
- Sortam fiecare bucket (discutam cum)
- Iterăm prin fiecare bucket, de la primul la ultimul, adăugând elementele înapoi în **v**

# Bucket Sort

Vizualizare:

<https://www.cs.usfca.edu/~galles/visualization/BucketSort.html>

# Bucket Sort

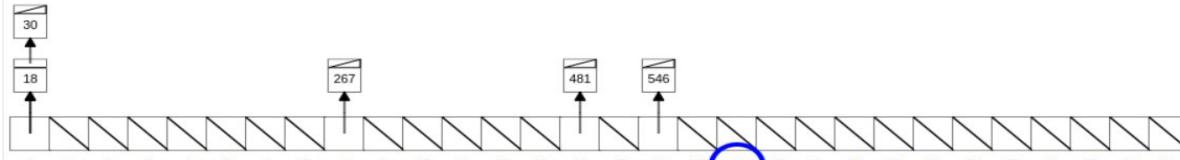
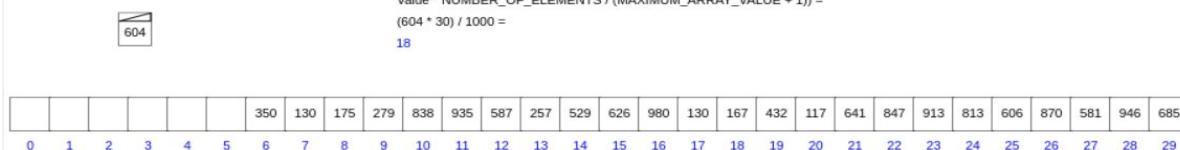
Cum adăugăm elementele în bucket-ul corespunzător?

-

# Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este să împărțim la o valoare și în funcție de catul împărțirii punem valoarea în bucketul corespunzător.
- În animație foloseam 30 de bucketuri și cum numerele erau pana la 1000, inmulteam cu 30 și imparteam la 1000



# Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este să împărțim la o valoare și în funcție de cat să punem în bucketul corespunzător

Cum sortam bucketurile ?

-

# Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este să împărțim la o valoare și în funcție de cat să punem în bucketul corespunzător

Cum sortam bucketurile ?

- Putem aplica recursiv tot bucketsort sau dacă avem puține elemente să folosim o sortare simplă (insertion/selection/bubble sort) ....
  - Cum adica să folosim bubble sort de ce nu quick sort ???

# Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este să împărțim la o valoare și în funcție de cat să punem în bucketul corespunzător

Cum sortam bucketurile ?

- Putem aplica recursiv tot bucketsort sau dacă avem putine elemente să folosim o sortare simplă (insertion/selection/bubble sort) ....
  - Cum adica să folosim bubble sort de ce nu quick sort ???
    - Pentru  $n$  mic constanta de la quicksort, mergesort face ca sortarea sa fie mai înceată

# Bucket Sort

- Cate bucketuri ?



# Bucket Sort

- Cate bucketuri ?
  - Dacă sunt foarte multe initializam spațiu prea mare
  - Dacă sunt prea puține nu dispersam suficient...
    - Ce se intampla dacă toate pica în același bucket ?
  - Conteaza foarte mult și distribuția inputului.

# Bucket Sort

Complexitate?

- Timp:

- 

- Spatiu:

-

# Bucket Sort

Complexitate?

- Timp:
  - Average  $O(n+k)$
  - Worst case  $O(n^2)$

Algoritm bun dacă avem o distribuție uniformă a numerelor...

- Spațiu:
  - $O(n+k)$

# Radix Sort

- Este un algoritm folosit în special pentru ordonarea sirurilor de caractere
  - Pentru numere - funcționează pe aceeași idee
- Asemănător cu bucket sort - este o generalizare pentru numere mari
- Împărțim în **B** bucketuri unde **B** este baza în care vrem sa considerăm numerele(putem folosi 10 sau 100 sau  $10^4$  sau 2 sau  $2^4$ ,  $2^6$  ...)
- Presupunem că vectorul de sortat **v** conține elemente întregi, cu cifre din multimea  $\{0, \dots, B-1\}$

# Radix Sort

- Cum sunt utilizate bucket-urile?
  - Elementele sunt sortate după fiecare cifră, pe rând
  - Bucket Urile sunt cifrele numerelor
  - Fiecare bucket  $b[i]$  conține, la un pas, elementele care au cifra curentă = i
- Numărul de bucket-uri necesare?
  - Baza în care sunt scrise numerele

# Radix Sort

Complexitate?

- Timp:
  - $O(n \log \max)$  (discutie mai lunga)
- Spațiu:
  - $O(n+b)$

# Radix Sort

Vizualizare:

<https://visualgo.net/bn/sorting>

# Radix Sort - LSD

- LSD = Least Significant Digit (iterativ rapid)

# Radix Sort - MSD

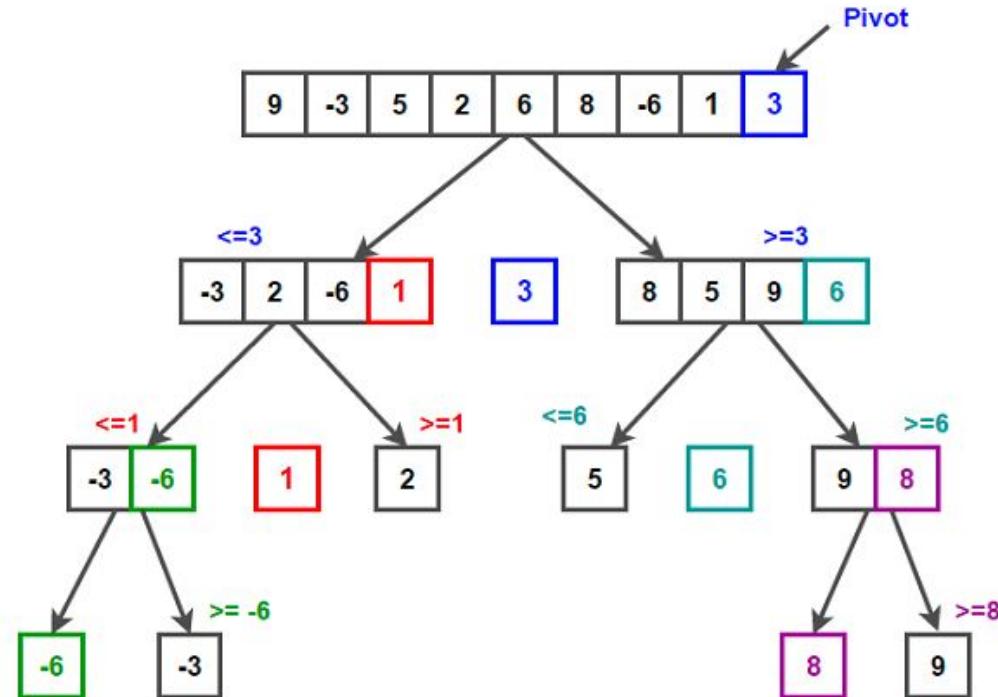
- MSD = Most Significant Digit (recursiv, ca bucket sort)

# Quick Sort

- Algoritm Divide et Impera
- Este un algoritm eficient în practica (implementarea este foarte importantă)
- **Divide:** se împarte vectorul în doi subvectori în funcție de un **pivot**  $x$  astfel încât elementele din subvectorul din stânga sunt  $\leq x \leq$  elementele din subvectorul din dreapta
- **Impera:** se sortează recursiv cei doi subvectori

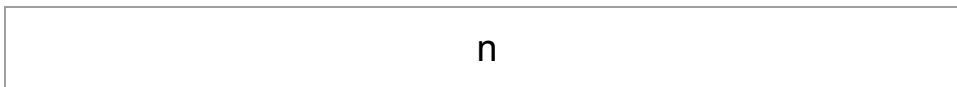
# Quick sort - exemplu

- Pivot ales la coada
- Contraexemplu ?



# Quick Sort

- În cel mai bun caz, pivotul  $x$  este chiar mediana, adică împarte vectorul în 2 subvectori de  $n/2$  elemente fiecare



1 partitie \*  $n = O(n)$



2 partitii \*  $n/2 = O(n)$



4 partitii \*  $n/4 = O(n)$

•  
•  
•

•  
•  
•



log n nivele,  $O(n) / nivel = O(n \ log n)$

# Quick Sort

Worst case?

- Când alegem cel mai mic sau cel mai mare element din vector la fiecare pas
- Una din cele două partiții va fi goală
- Cealaltă partiție are restul elementelor, mai puțin pivotul
- Număr de apeluri recursive?
  - $n - 1$
- Lungime partiție?
  - $n - k$  (unde  $k$  = numărul apelului recursiv) →  $O(n - k)$  comparații
- Complexitate finală?
  - $O(n^2)$

# Quick Sort

Cum alegem pivotul?

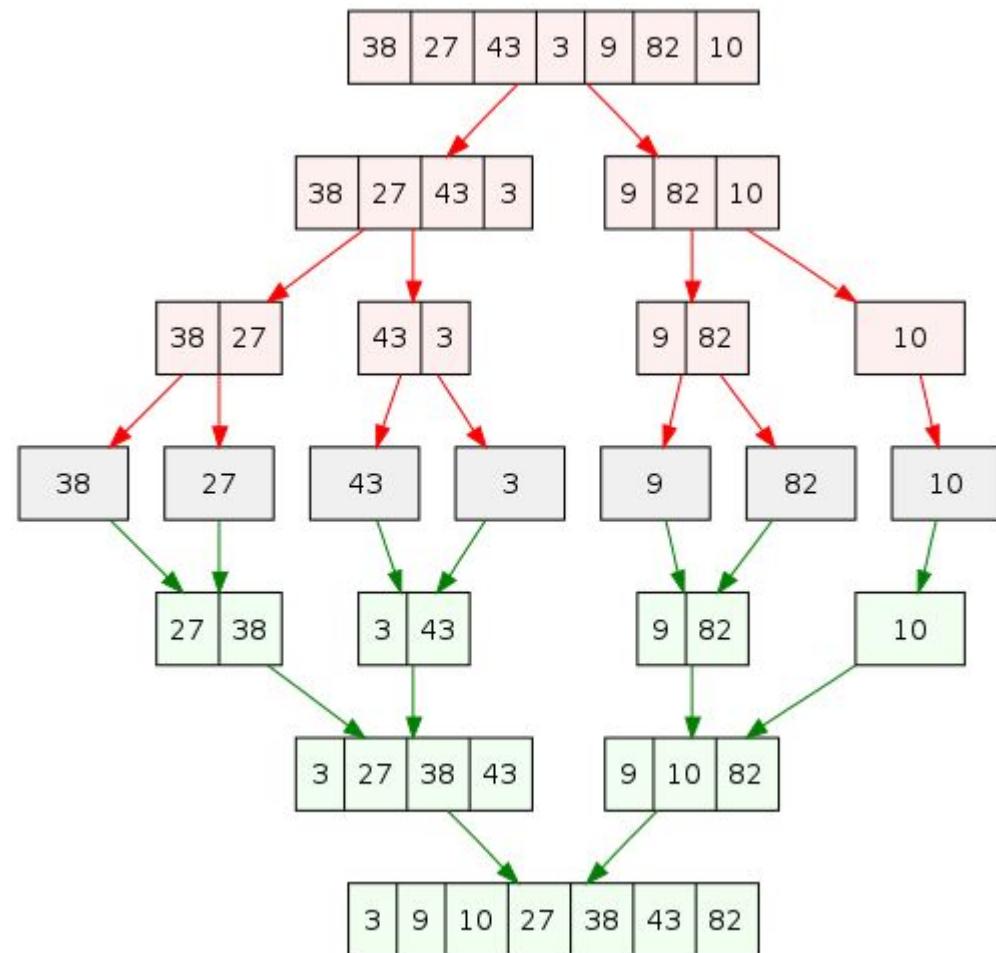
- Primul element
- Elementul din mijloc
- Ultimul element
- Un element random
- Mediana din 3
- Mediana din 5,7 (atenție cand vectorul devine mic, facem mult calcul pentru putin)

[https://en.wikipedia.org/wiki/Quicksort#Choice\\_of\\_pivot](https://en.wikipedia.org/wiki/Quicksort#Choice_of_pivot)

# Merge Sort

- Algoritm Divide et Impera
- **Divide:** se împarte vectorul în jumătate și se sortează independent fiecare parte
- **Impera:** se sortează recursiv cei doi subvectori

# Merge Sort - example



# Merge Sort

- Când se oprește recursivitatea?
  - Când vectorul ajunge de lungime 1 sau 2 (depinde de implementare)
  - La fel ca și la quicksort ne-am putea opri mai repede ca să evitam multe operații pentru puține numere
- Algoritm de merging
  - Creem un vector temporar
  - Iterăm cele două jumătăți sortate de la stânga la dreapta
  - Copiem în vectorul temporar elementul mai mic dintre cele două

# Merge Sort Vs Quick Sort

De ce e Quick Sort mai rapid în practica cand cazul ideal de la Quick Sort e ca împărțim în 2 exact ce face Merge Sortul?

- Merge Sortul are nevoie de un vector suplimentar și face multe mutari suplimentare.
- Quick Sortul e în place... memoria suplimentară e pentru stiva...

# In-Place Merge Sort

- Nu folosim vector suplimentar ca în cazul Merge Sort
  - Nu este  $O(n \log n)$
  - Mai complicat
  - O alta optiune este Block Sort

# Heap Sort

Vizualizare:

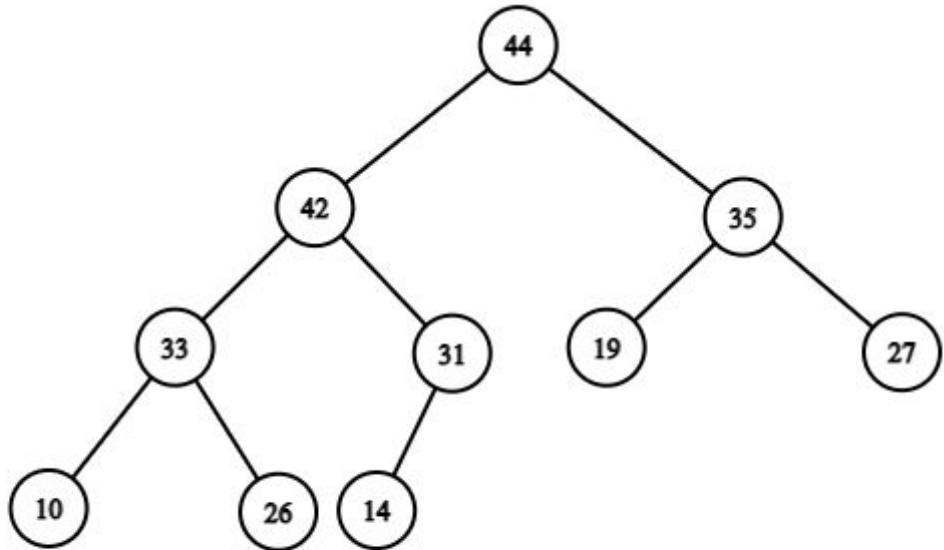
<https://www.cs.usfca.edu/~galles/visualization/HeapSort.html>

6 5 3 1 8 7 2 4

# Scurtă introducere în heap-uri

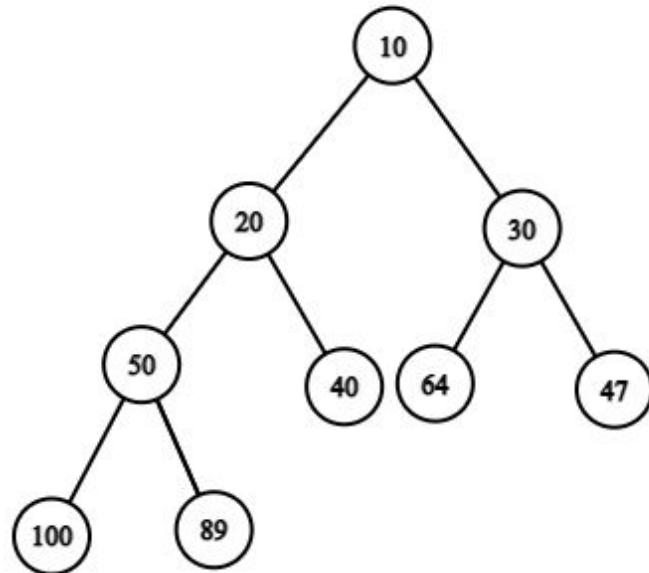
- Ce este un heap?
  - Arbore binar aproape complet
  - Are înălțimea  $h = \log n$
- Max-heap
  - Pentru orice nod  $X$ , fie  $T$  tatăl lui  $X$
  - $T$  are valoarea  $\geq$  decât valoarea lui  $X$
  - Elementul maxim este în rădăcină
- Min-heap
  - Pentru orice nod  $X$ , fie  $T$  tatăl lui  $X$
  - $T$  are valoarea  $\leq$  decât valoarea lui  $X$
  - Elementul minim este în rădăcină

# Scurtă introducere în heap-uri



**Max-heap**

Ultima poziție: 14



**Min-heap**

Ultima poziție: 89

# Heap Sort

- În funcție de sortarea dorită (ascendentă sau descendentă) - se folosește max-heap sau min-heap

## IDEE:

- Elementele vectorului inițial sunt adăugate într-un heap
- La fiecare pas, este reparat heap-ul după condiția de min/max-heap
- Cât timp mai sunt elemente în heap:
  - Fie X elementul maxim
  - X este interschimbat cu cel de pe ultima poziție în heap
  - X este adăugat la vectorul sortat (final)
  - X este eliminat din heap
  - Heap-ul este reparat după condiția de min/max-heap

# Intro Sort

- Se mai numește Introspective Sort
- Este sortarea din anumite implementări ale STL-ului
- Este un algoritm hibrid (combină mai mulți algoritmi care rezolvă aceeași problemă)
- Este format din Quick Sort, Heap Sort și Insertion Sort

## IDEE:

- Algoritmul începe cu Quick Sort
- Trece în Heap Sort dacă nivelul recursivității crește peste  $\log n$
- Trece în Insertion Sort dacă numărul de elemente de sortat scade sub o anumită limită

# TimSort

- Sortarea din python
- Este un algoritm hibrid care îmbina MergeSort cu sortare prin inserare.

## IDEE

- Algoritmul începe cu Merge Sort
- Trece în Insertion Sort dacă numărul de elemente de sortat scade sub o anumită limită (32, 64)

# Sortări prin comparație

Vizualizare:

<https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

- Quick Sort
- Merge Sort
- Algoritmi elementari de sortare

# Clase de complexitate

Urmatoarele slideuri sunt copiate de

la:<http://cadredidactice.ub.ro/simonavarlan/files/2018/02/Curs-2-2018.pdf>

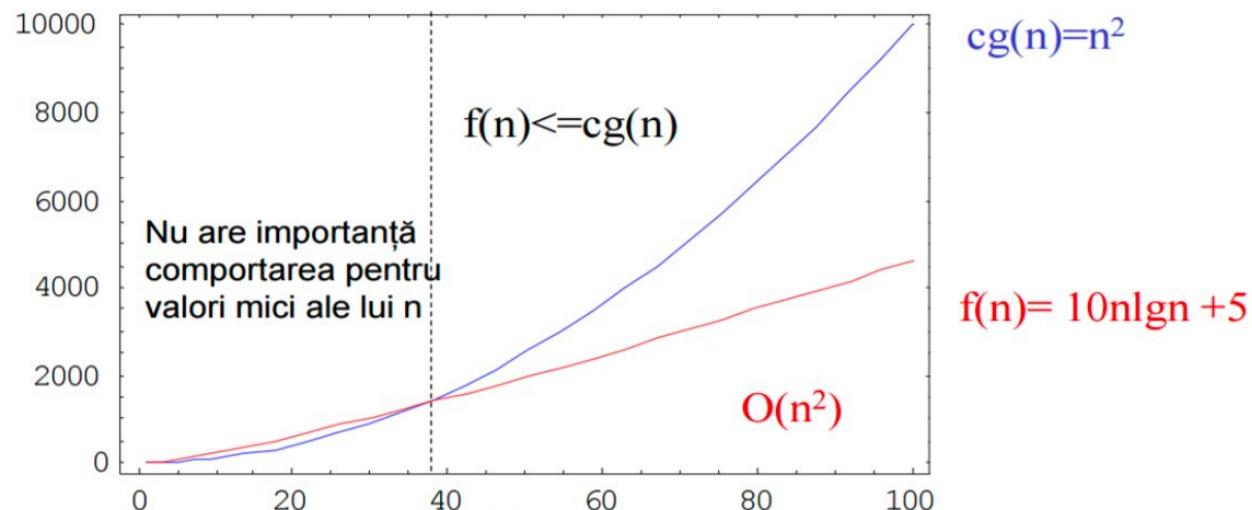
Ideal ar fi sa le refaci dacă ti-e rezonabil de usor .

# Clase de complexitate

## Complexitatea Algoritmilor

### Notatia O

Ilustrare grafica. Pentru valori mari ale lui  $n$ ,  $f(n)$  este marginită superior de  $g(n)$  multiplicată cu o constantă pozitivă



# Big O

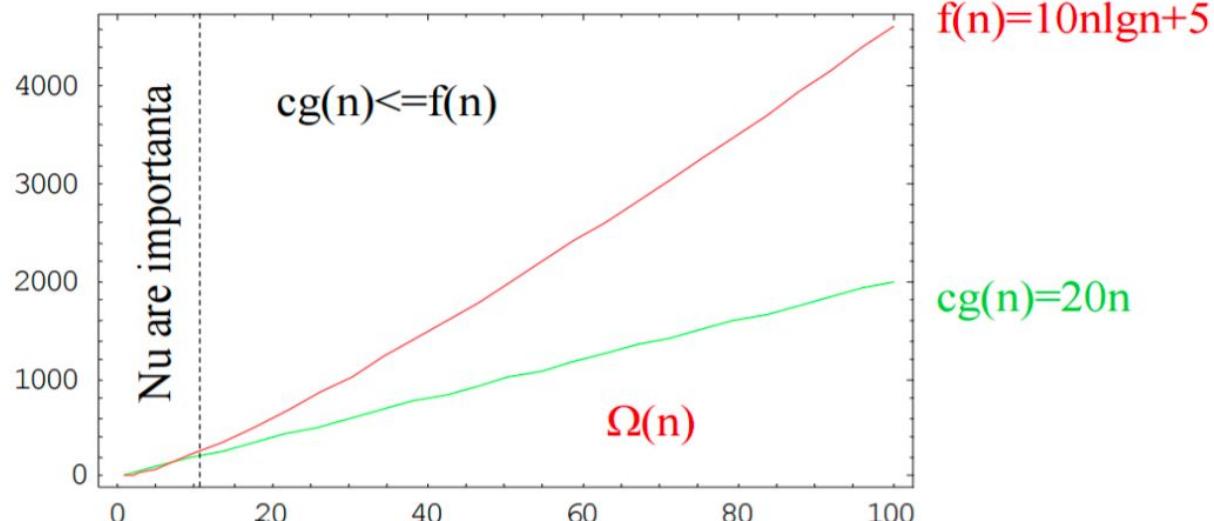
- $O \rightarrow$  marginire superioară
  - Un algoritm care face  $3n$  operații este și  $O(n)$  dar și  $O(n^2)$  și  $O(n!)$
  - În general vom vrea totusi marginea stransa care este de fapt  $\Theta$

# Clase de complexitate

## Complexitatea Algoritmilor

### Notăția $\Omega$

Ilustrare grafică. Pentru valori mari ale lui  $n$ , funcția  $f(n)$  este marginită inferior de  $g(n)$  multiplicată eventual de o constantă pozitivă

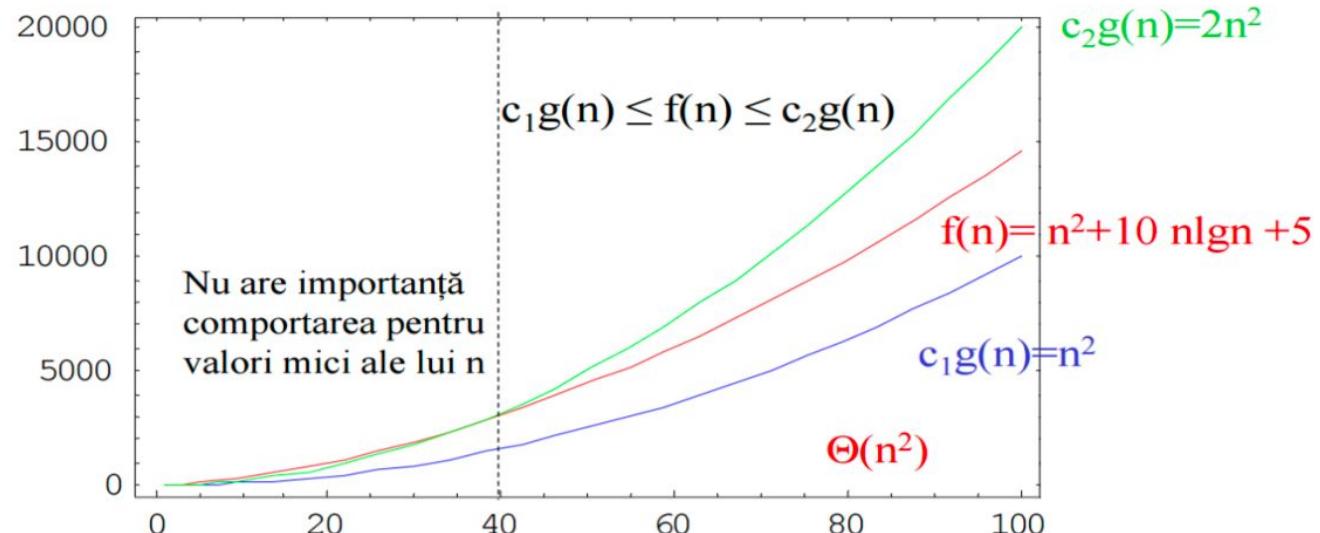


# Clase de complexitate

## Complexitatea Algoritmilor

### Notăția $\Theta$

Ilustrare grafică. Pentru valori mari ale lui  $n$ ,  $f(n)$  este mărginită, atât superior cât și inferior de  $g(n)$  înmulțit cu niște constante pozitive



# Complexitatea minima pentru o sortare prin comparație

**Teorema:** Orice algoritm de sortare care se bazeaza pe comparatii face cel putin  $\Omega(n \log n)$  comparatii.

**Schita Demonstrație:** Sunt în total  $n!$  permutari. Algoritmul nostru de sortare trebuie sa sorteze toate aceste  $n!$  permutări. La fiecare pas pe baza unei comparatii intre 2 elemente putem în funcție de răspuns sa eliminăm o parte din comparatii. La fiecare pas putem injumatatii numărul de permutări -> obținem minim:  $\log_2 (n!)$  comparatii,

$$\text{dar } \log_2 (n!) = \log_2 (n) + \log_2 (n - 1) + \dots + \log_2 (2) = \Omega(n \log n).$$

# Complexitatea minima pentru o sortare prin comparație

**Teorema:** Orice algoritm de sortare care se bazeaza pe comparatii face cel putin  $\Omega(n \log n)$  comparatii.

**Exemplu:**  $N = 3$ , vrem sa sortam orice permutare a vectorului {1,2,3}:

(1,2,3) (1,3,2) (2,1,3) (2,3,1) (3,1,2) (3,2,1)

Facem o prima comparație sa zicem  $a_1 ? a_2$ . Sa zicem ca  $a_1 > a_2 \rightarrow$  raman 3 posibilitati:  
(1,2,3) (1,3,2), (2,3,1)

Dacă ulterior comparam 1 cu 3 ... atunci dacă  $3 > 1$  am terminat dar dacă  $1 > 3$  ramanem cu (1,2,3) (1,3,2) și mai trebuie sa facem a 3-a comparație...

Final

---

# Structuri de Date Elementare

— Liste Vectori Stive Cozi —

---

# Exercițiu

- Se citesc numere de la tastatură până se citește 0. Sortați acele numere!
  - Opțiuni folosite:
    - Vector din STL în C++
      - Nu știm câtă memorie să alocăm
        - 1.000 → probabil prea puțin ⇒ segmentation fault
        - 10.000.000 → probabil prea mult ⇒ risipă de memorie
      - Array din C++ alocat dinamic
        - În cazul de față, nu e corect, pentru că nu știm câte elemente inserăm, dar în general ar putea fi o soluție bună
      - Liste în Python
      - Priority queue

# Exercițiu

- Se citesc  $n \leq 10^6$  numere, care fac parte din unul din cele  $m \leq 10^3$  grupuri. La final se pun întrebări de tipul: **care e al k-lea număr din grupul j?**
- $N = 8, M = 3$
- 9 3
- 12 3
- 13 1
- 4 2
- 6 2
- 7 2
- 11 1
- 12 3
- Q
- $2 \ 2 \rightarrow 6$
- $3 \ 3 \rightarrow 12$
- $1 \ 1 \rightarrow 13$  (nu în ordinea sortării, ci în ordinea citirii)

# Exercițiu

- Se citesc  $n \leq 10^6$  numere, care fac parte din unul din cele  $m \leq 10^3$  grupuri. La final se pun întrebări de tipul: **care e al k-lea număr din grupul j?**
- Soluții:
  - Matrice[n][m] → 4GB
    - Ocupă foarte multă memorie și dacă  $m = 10^5$ ! Clar soluția nu merge
    - Risipă de 99.9%!!
  - Listă de liste sau vectori de liste...
    - Soluție bună
  - Un vector lung care ține toate elementele cu un alt vector de next-uri

# Alocare statică vs Alocare Dinamică

- Alocare statică
  - C++
    - Array:
      - int v[1000]; int n = 733; → Trebuie să reținem noi lungimea
      - int v[1000][1000000]; → problematic
      - Static
    - Vector
      - vector <int> v;
      - for (int i = 0; i < n; ++i) {  
    cin>> x;  
    v.push\_back(x);  
}
      - vector <int> matrix[1000];
      - Nu prea static (vom discuta mai mult)

# Alocare statică vs Alocare Dinamică

- Alocare statică

- Python

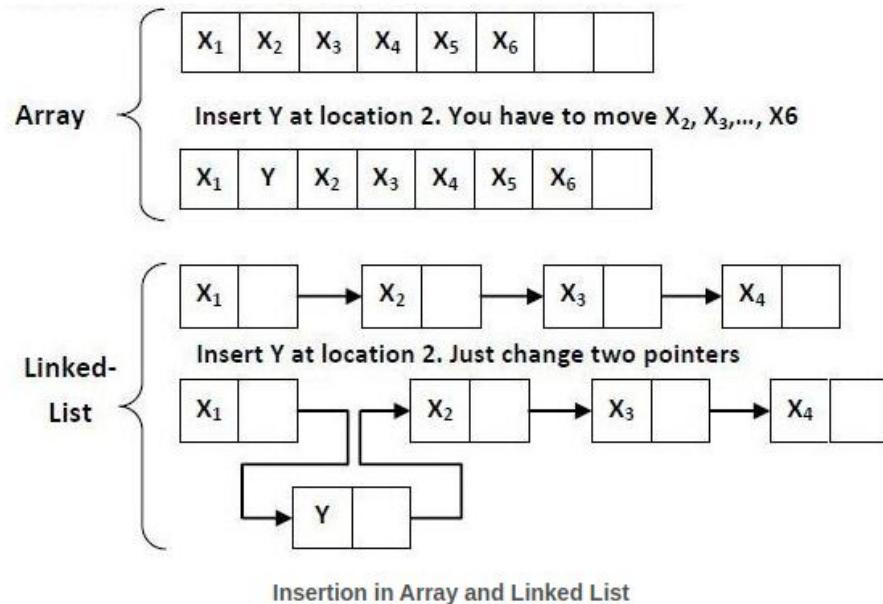
- `import array as arr` sau
    - `a = arr.array('d', [1.1, 3.5, 4.5])`
    - Sau direct: `array_2 = np.array(["numbers", 3, 6, 9, 12])`
    - Wrapper la array-ul din C++
    - Lumea folosește de obicei liste
    - Nu prea static...
    - `a.append(45)`

# Alocare statică vs Alocare Dinamică

- Array vs Liste
  - Păreri ?
    - Array sunt mai rapizi
    - Pot cauza risipă de memorie, că nu tot timpul știm câtă memorie să alocăm de la început
    - Putem avea probleme să alocăm o secvență continuă lungă sau să o extindem

# Alocare statică vs Alocare Dinamică

- Array vs Liste
  - Păreri ?
  - Inserare



# Array vs Liste

- Array-ul ocupă poziții consecutive din memorie și reține informații de același fel.
  - Ocupare optimă a memoriei
  - Mai rapizi
  - Probleme cu alocarea (trebuie să găsești un spațiu suficient de mare să aloci)...

## Exemplu:

- În tabelul de mai jos, nu putem aloca un vector de 4 elemente
- Putem ocupa memorie degeaba V[1000][100000]...
- Putem șterge și adăuga doar în capătul din dreapta în complexitate constantă
- Putem accesa în  $O(1)$  elemente de pe anumite poziții...

			O				O				O			
--	--	--	---	--	--	--	---	--	--	--	---	--	--	--

# Array vs Liste

- Lista permite alocarea memoriei când avem nevoie de ea
  - O(1) inserare/ștergere oriunde, **dacă** avem pointerul de care avem nevoie
  - Nu putem găsi ușor al k-lea element din listă
    - (skip lists can help)
  - Trebuie să avem grija să alocăm/ștergem memoria (cel puțin în C++)

# Array vs Liste

Complexitate: (sa completam impreuna)

	Liste	Array
Inserare oriunde		
Inserare/ștergere la capăt		
Afișarea celui de-al k-lea element		
Sortare		
Căutare în structura sortată		
Redimensionare		

# Array vs Liste

Complexitate:

	Liste	Array
Inserare oriunde	În caz bun, $O(1)$	$O(n)$
Inserare/ștergere la capăt	$O(1)$	$O(1)$
Afișarea celui de-al $k$ -lea element	$O(k)$	$O(1)$
Sortare	$O(n \log n)$	$O(n \log n)$
Căutare în structura sortată	$O(n)$	$O(\log n)$
Redimensionare	$O(1)$	$O(n)$

# Array vs Vector

- În array, alocăm de la început memoria
  - De obicei facem risipă
  - Trebuie să reținem noi câte elemente folosim
  - Foarte rapizi
  - Folosesc memoria eficient
- Vector
  - Alocăm **niște** memorie de la început
  - Redimensionăm

# Array vs Vector

- Vector

- Array alocat dinamic
- Putem aloca din start un număr de elemente: `vector<double> values(500, 3.14);`
- Putem rezerva locuri
  - `values.reserve(1000000);`

```
vector <int> linie;
int n;
linie.reserve(n);
cin >>n;
for (int i = 0; i < n; ++i) {
    int x;
    cin>>x;
    linie[i] = x;
}
```

# Array vs Vector

- Vector
  - Sau putem să adăugăm la final ... (de ce la final?)
    - Să nu mutăm toate elementele

```
vector <int> linie;
int n;
//linie.reserve(n);
cin >>n;
for (int i = 0; i < n; ++i) {
    int x;
    cin>>x;
    linie.push_back(x); // ce se întâmplă aici ?
}
```

# Vector

- Redimensionare
  - Vectorul începe cu un număr de locuri rezervate
  - Dacă vrem să adăugăm un element și nu mai avem spațiu
    - Mărim vectorul
      - Cu cât?
        - Dublăm sau  $1.5x$  sau  $3x$ , ca să rămânem amortizat în  $O(1)$  pe operație
      - Ce se întâmplă dacă nu mai e loc în continuare?
    - Dacă tot eliminăm elemente
      - Trebuie să micsorăm vectorul
        - Când?
        - Cu cât?
      - Dacă dublăm, complexitatea amortizată e  $O(1)$  pe operație!
- Avantajele array-urilor, dar cu alocare dinamică
  - Viteza este totuși mai mică decât array-urile. Dacă viteza e vitală, folosiți array!

# Liste înlăntuite

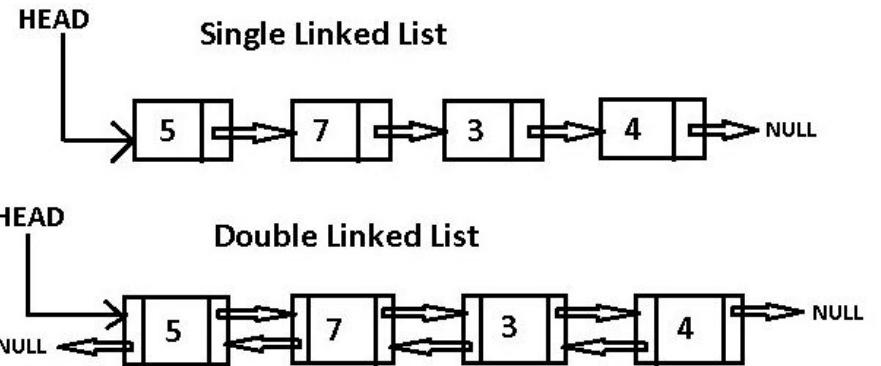
## Python

- `my_list = ["mouse", [8, 4, 6], ['a']]`
- `my_list.append("Primavara e frumoasa");`
- Putem avea elemente de tipuri diferite
- Alocarea/dealocarea se fac behind the scenes

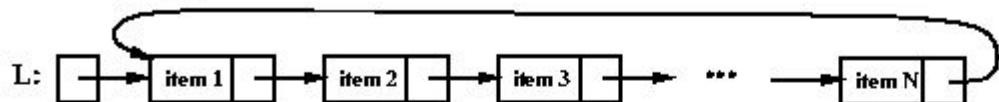
# Liste înlățuite

- Liste

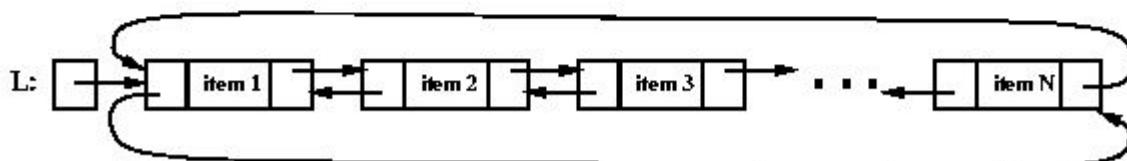
- Simplu înlățuite
- Dublu înlățuite
- Circulare



**Circular, singly linked list:**



**Circular, doubly linked list:**



# Liste înlántuite

C++

- Două opțiuni:
  - Container din C++ similar cu vectorul
  - Alocate de mâna

```
// list::begin
#include <iostream>
#include <list>

int main ()
{
    int myints[] = {75,23,65,42,13};
    std::list<int> mylist (myints,myints+5);

    std::cout << "mylist contains:";
    for (std::list<int>::iterator it=mylist.begin(); it != mylist.end(); ++it)
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

# Liste înlăntuite

C++

- Două opțiuni:
  - [Containere din STL](#)
  - [Liste alocate dinamic](#)

# Final

---

# Structuri de Date Elementare

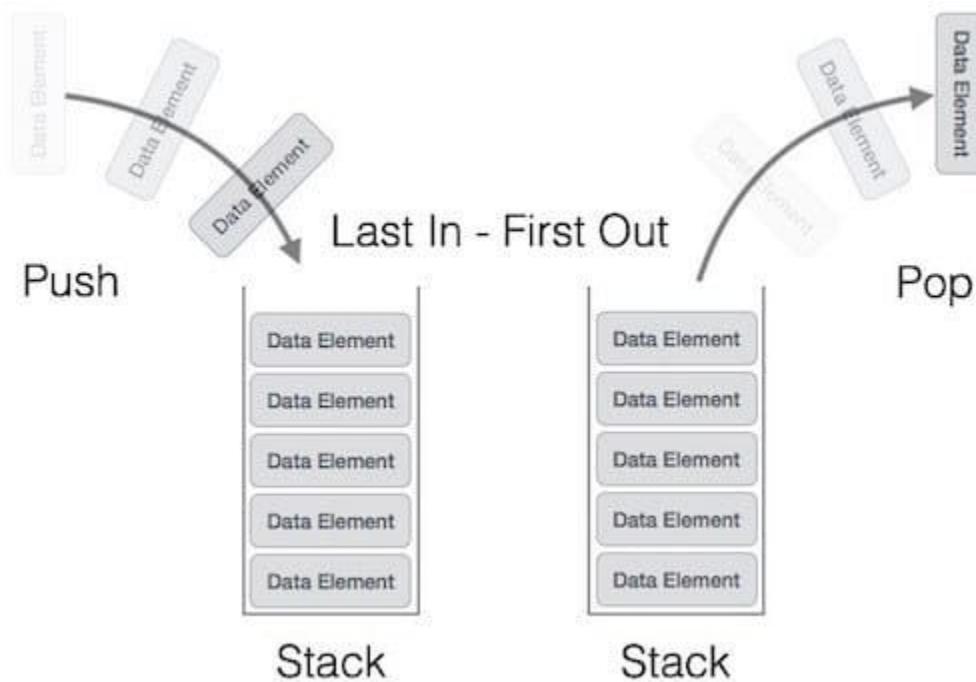
— Stive Cozi Deque Mars? Batog? —

---

# Stive (Stack)

- Sunt structuri de date de tip **LIFO** (**Last In First Out**)
- Avem acces numai la elementul din vârf (top)
- Operații de bază:
  - Push - adăugarea unui element (în vârf)
  - Pop - eliminarea elementului din vârf
- Operații suplimentare:
  - Size() - numărul de elemente
  - isEmpty() - returnează **true** dacă numărul de elemente este exact 0
  - Peek() - ne spune valoarea din vârf fără să o extragă

# Stive (Stack)



# Stive (Stack)

- Metode de implementare:
  - Stivă ca Vector
    - [Vizualizare](#)
    - [Implementare](#) (găsiți în secțiunea de implementare ca array)
  - Stivă ca Listă
    - [Vizualizare](#)
    - [Implementare](#) (găsiți în secțiunea de implementare ca linked list)
  - Stivă în C++ - <https://en.cppreference.com/w/cpp/container/stack>

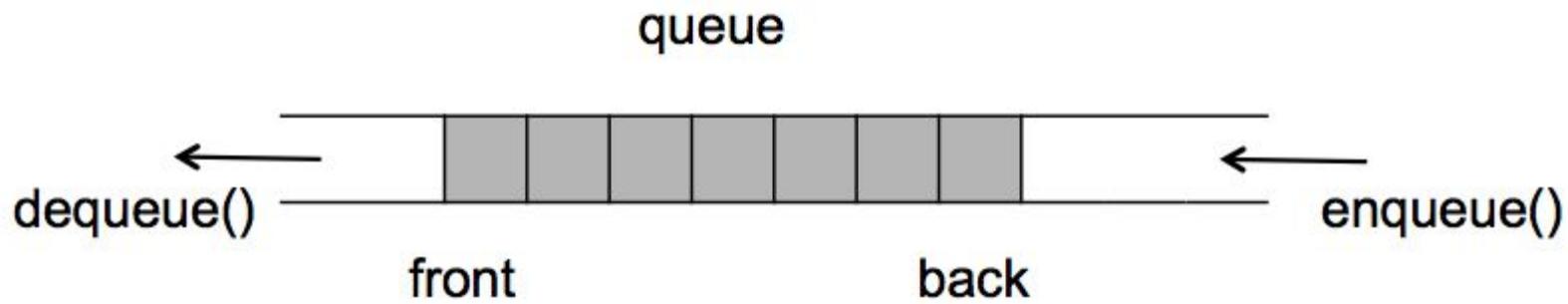
# Exerciții

- <https://www.infoarena.ro/problem/nrpits>
- Inversarea unui text
- Problema [parantezelor](#)

# Cozi (Queue)

- Sunt structuri de date de tip **FIFO** (**F**irst **I**n **F**irst **O**ut)
- Avem acces la primul și la ultimul element (head & tail / front & back)
- Operații de bază:
  - Push - adăugarea unui element la coadă
  - Pop - eliminarea unui element
- Operații suplimentare:
  - Size() - numărul de elemente
  - isEmpty() - returnează **true** dacă numărul de elemente este exact 0
  - First() - ne spune valoarea de la început fără să o extragă
  - Last() - ne spune valoarea de la sfârșit fără să o extragă

# Cozi (Queue)



# Cozi (Queue)

- Metode de implementare:
  - Coadă ca Vector
    - [Vizualizare](#)
    - [Implementare](#)
  - Coadă ca Listă
    - [Vizualizare](#)
    - [Implementare](#)
  - Coadă în C++ - <https://en.cppreference.com/w/cpp/container/queue>

# Deque

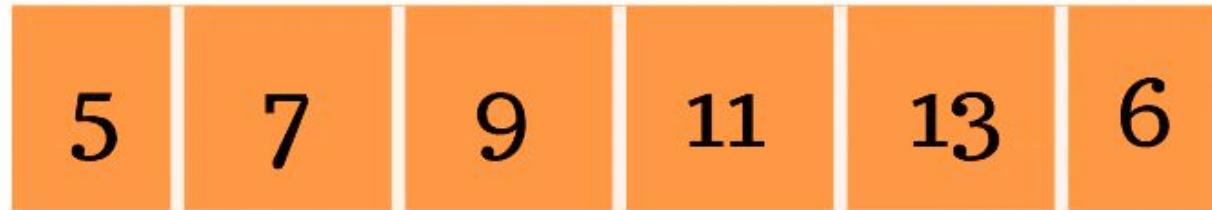
- Double ended queue (coadă cu două capete)
- Operații de bază:
  - Push Front
  - Push Back
  - Pop Front
  - Pop Back
- Operații suplimentare
  - Size()
  - Front()
  - Back()
  - isEmpty()

# Deque

Insert At Front



Deque Data Structure



Insert At Rear



Remove from Front



Remove from Back



# Deque

- Double ended queue (coadă cu două capete)
- Metode de implementare:
  - Deque ca Listă
    - [Vizualizare](#)
    - [Implementare](#)
  - Deque ca Array
    - [Implementare](#)
  - Deque în C++ - <https://en.cppreference.com/w/cpp/container/deque>

# Exerciții

- <https://infoarena.ro/problemă/deque>
- Book Pile -  
<https://codeforces.com/problemsets/acmsguru/problem/9999/271>

# Problemă

Se dă un vector cu n elemente și apoi n operații de genul:

- 1 i j → care este minimul din intervalul  $[i, j]$
- 2 i x → modificați elementul de pe poziția  $i$  în  $x$

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8

# Problemă

Se dă un vector cu  $n$  elemente și apoi  $n$  operații de genul:

- 1  $i \ j \rightarrow$  care este minimul din intervalul  $[i, j]$
- 2  $i \ x \rightarrow$  modificați elementul de pe poziția  $i$  în  $x$

Idei?

-

# Şmenul lui Batog

Se dă un vector cu  $n$  elemente și apoi  $n$  operații de genul:

- 1  $i \ j \rightarrow$  care este minimul din intervalul  $[i, j]$
- 2  $i \ x \rightarrow$  modificați elementul de pe poziția  $i$  în  $x$

**Idee:**

Împărțim vectorul în zone de lungime  $L$  și calculăm minimul pe fiecare zonă încă parte.

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
2			5			6		

# Şmenul lui Batog - SQRT Decomposition

Împărțim vectorul în zone de lungime L și calculăm minimul pe fiecare zonă în parte.

Linkuri externe:

- [Geeks for geeks](#)
- [CpAlgorithms](#)

# Şmenul lui Batog

Se dă un vector cu n elemente și apoi n operații de genul:

- 1 i j → care este minimul din intervalul  $[i, j]$
- 2 i x → modificați elementul de pe poziția  $i$  în  $x$

Cum răspundem la 1 0 8; 1 0 4; 1 1 7 ?

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
2			5			6		

# Şmenul lui Batog

Se dă un vector cu  $n$  elemente și apoi  $n$  operații de genul:

- 1  $i \ j \rightarrow$  care este minimul din intervalul  $[i, j]$
- 2  $i \ x \rightarrow$  modificați elementul de pe poziția  $i$  în  $x$

Cum răspundem la 1 1 7 ?

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
2			5			6		

# Şmenul lui Batog - Complexitate 1

Pentru query (operația de tip 1):

Împărțim vectorul în **n / L zone** de lungime L

Putem itera aproape complet **2 zone** (de la început și/sau de la final)  $\Rightarrow O(2*L)$

$\Rightarrow O(n/L + 2 * L)$

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
2			5			6		

# Şmenul lui Batog - Complexitate 1

Pentru query (operația de tip 1):

$$O(n/L + 2 * L)$$

Cât trebuie să fie L pentru o complexitate minimă?

- $L = \sqrt{n}$

$$\Rightarrow O(n/\sqrt{n} + 2 * \sqrt{n})$$

$$= O(\sqrt{n} + 2 * \sqrt{n})$$

$$= \textcolor{red}{O(\sqrt{n})}$$

# Şmenul lui Batog

Se dă un vector cu n elemente și apoi n operații de genul:

- 1 i j → care este minimul din intervalul  $[i, j]$
- 2 i x → modificați elementul de pe poziția  $i$  în  $x$

Cum răspundem la    2 2 1;    2 3 10 ?

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
2			5			6		

# Şmenul lui Batog

Se dă un vector cu n elemente și apoi n operații de genul:

- 1 i j → care este minimul din intervalul [i, j]
- 2 i x → modificați elementul de pe poziția i în x

Cum răspundem la 2 3 10 ?

0	1	2	3	4	5	6	7	8
3	9	2	5 10	7	34	6	11	8
2			7			6		

# Şmenul lui Batog - Complexitate 2

Pentru **update** (operăția de tip 2):

Modificăm elementul de pe poziția  $i$

Trebuie să facem update pe zona respectivă (să recalcuăm minimul)

$\Rightarrow O(L) = O(\sqrt{n})$

0	1	2	3	4	5	6	7	8
3	9	2	5 10	7	34	6	11	8
2			7			6		

# Şmenul lui Batog

Împărțim vectorul în zone de:

- $\text{sqrt}(n)$

- $\text{sqrt}(n) / 2$

- $\text{sqrt}(n) * 2$

- Variațiuni

- De ce?

# Şmenul lui Batog

Împărțim vectorul în zone de:

- $\text{sqrt}(n)$
- $\text{sqrt}(n) / 2$
- $\text{sqrt}(n) * 2$
- Variațiuni
- De ce?
  - În practică, algoritmul poate rula mai rapid pentru valori diferite de  $\text{sqrt}(n)$ , în funcție de operațiile care se fac pe segmente.

# Şmenul lui Batog - sortare

Se dă un vector cu n elemente. Sortați-l folosind şmenul lui Batog.

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
	2			5			6	

<https://leetcode.com/problems/sort-an-array/>

Complexitate?

- $O(n \sqrt{n})$

# Kahoot

Începem cursul 5 (Luni 15 ianuarie ora 14:00) cu **Kahoot dublu!**

# Final

# Hash-uri

Tabele cu adresare directă. Tabele de dispersie

# Kahoot



# Tabele de dispersie

Complexități pe operații ale unor structuri de date:

	Inserare	Ștergere min	Ștergere cu pointer	Ștergere fără pointer	Afișare minim	Căuare	Succesor	Afișare sortat
Heap	O(log n)	O(log n)	O(log n)	O(n)	O(1)	O(n) :(	O(n) :(	O(n logn)
Arbore de căutare echilibrați	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Vector	O(1)	O(n)	O(?) O(1) sau O(n)	O(n)	O(n)	O(n)	O(n)	O(n log n)
Listă înălțuită	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n log n)

# Tabele de dispersie

**Recapitulare:** heap-urile și arborii binari de căutare ţin ordine... prea complicat

	Inserare	Ștergere min	Ștergere cu pointer	Ștergere fără pointer	Afișare minim	Căuare	Succesor	Afișare sortat
Heap	O(log n)	O(log n)	O(log n)	O(n)	O(1)	O(n) :(	O(n) :(	O(n logn)
Arbore de căutare echilibrați	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Hashuri	O(1)	???	O(1)	O(1)	???	O(1)	???	???

# Tabele cu adresare directă

**Problemă:** Se dau 2 tipuri de operații pe numere de la 1 la N ( $N \leq 10^6$ ). Se dau până la M  $\leq 10^6$  operații.

- Inserați numărul x
- Întrebare: numărul x se află între numerele date?

Soluție?

- (ineficientă) Insertion sort pe inserare, apoi căutare binară
  - $O(n)$  pe inserare
  - $O(\log n)$  pe căutare

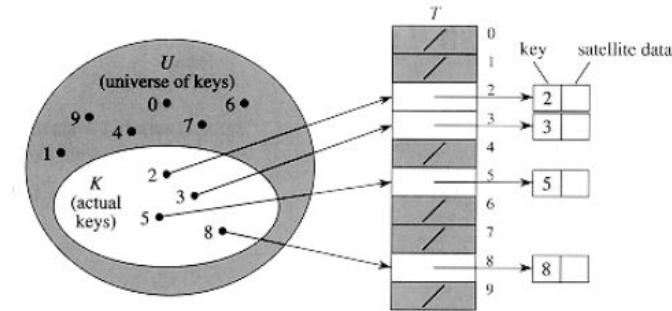
# Tabele cu adresare directă

**Problema:** Se dau 2 tipuri de operații pe numere de la 1 la N ( $N \leq 10^6$ ). Se dau până la  $M \leq 10^6$  operații.

- Inserați numărul x
- Întrebare: numărul x se află între numerele date?

Soluție?

- Putem ține un vector binar:  $a[i]=1$  dacă elementul s-a dat și  $a[i]=0$  dacă elementul nu a fost dat. Inițial este totul 0.
  - Complexitate?
    - **O(1) inserare și căutare!!!**



# Tabele cu adresare directă

**Problemă:** Se dau 2 tipuri de operații pe numere de la 1 la N ( $N \leq 10^6$ ).

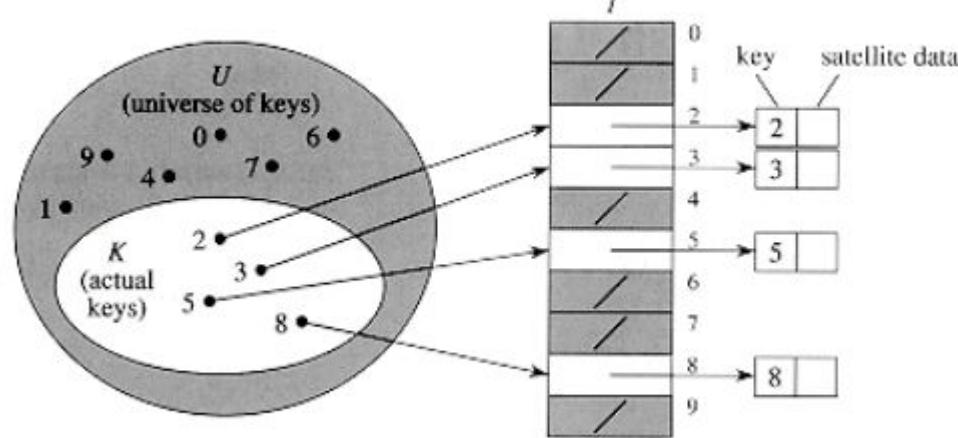
- Inserați numărul x
- Întrebare: numărul x se află între numerele date?
- **Ștergere: Elimin numărul x din numerele mele**

Soluție?

- Putem ține un vector binar:  $a[i]=1$  dacă elementul s-a dat și  $a[i]=0$  dacă elementul nu a fost dat. Inițial este totul 0.
  - Complexitate?
    - O(1) inserare și căutare!!! **Și ștergere!!!**

# Tabele cu adresare directă

Implementare O(1) și foarte scurtă!



DIRECT-ADDRESS-SEARCH( $T, k$ )

return  $T[k]$

DIRECT-ADDRESS-INSERT( $T, x$ )

$T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE( $T, x$ )

$T[key[x]] \leftarrow \text{NIL}$

# Tabele cu adresare directă

**Problemă:** Se dau 2 tipuri de operații pe numere de la 1 la N ( $N \leq 10^6$ ).

- Putem ține un vector binar:  $a[i]=1$  dacă elementul s-a dat și  $a[i]=0$  dacă elementul nu a fost dat. Inițial este totul 0.
  - Complexitate?
    - $O(1)$  inserare, căutare și ștergere

**Unde apare problema?**

# Tabele cu adresare directă

**Problemă:** Se dau 2 tipuri de operații pe numere de la 1 la N ( $N \leq 10^6$ ).

- Putem ține un vector binar:  $a[i]=1$  dacă elementul s-a dat și  $a[i]=0$  dacă elementul nu a fost dat. Inițial este totul 0.
  - Complexitate?
    - $O(1)$  inserare, căutare și ștergere

## Unde apare problema?

- **Limita de  $N \leq 10^6$ .**
  - Dacă am numere mai mari? Dacă am cuvinte sau altfel de obiecte? Dacă am numere negative?
- <https://leetcode.com/problems/two-sum/> (**problemă clasică de interviuri**) (cod1 va merge doar pt N mic  $\approx 10^7$  nu putem aloca oricata memorie)

# Tabele de dispersie

Trebuie să luăm elementele să le dispersăm

- Unde sunt problemele?
  - Mai multe elemente pică pe același câmp -> **coliziune**
  - Cam toate elementele pică în același loc -> **functie de dispersie proasta!**
    - Ex: %100 la prețuri de televizor
- Cum le rezolvăm?

Assume a table with 8 slots:

Hash key = key % table size

$$4 \quad = \quad 36 \% 8$$

$$2 \quad = \quad 18 \% 8$$

$$0 \quad = \quad 72 \% 8$$

$$3 \quad = \quad 43 \% 8$$

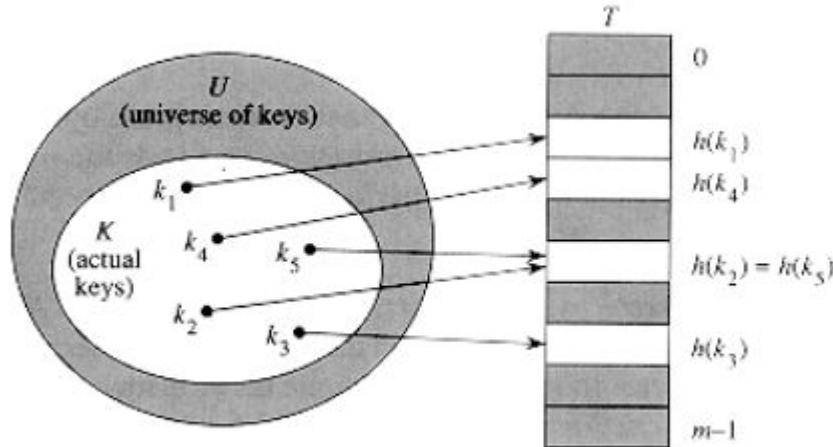
$$6 \quad = \quad 6 \% 8$$

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

# Tabele de dispersie

Trebuie să luăm elementele să le dispersăm

- În realitate elementele se suprapun



Assume a table with 8 slots:

Hash key = key % table size

$$4 = 36 \% 8$$

$$2 = 18 \% 8$$

$$0 = 72 \% 8$$

$$3 = 43 \% 8$$

$$6 = 6 \% 8$$

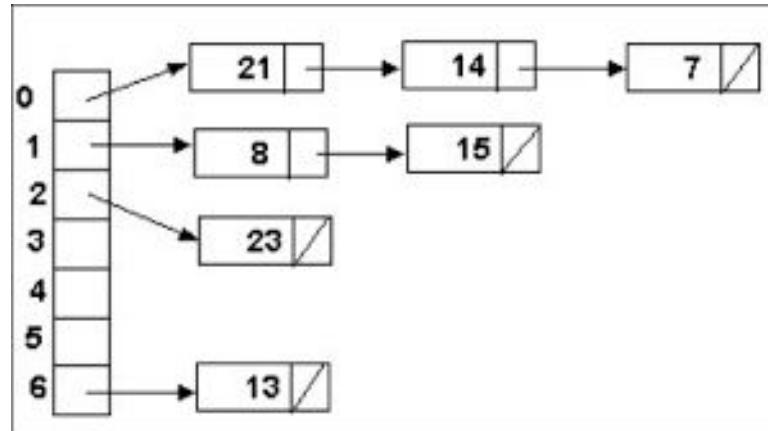
[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

# Funcție de dispersie

- Există mai multe metode de dispersie
- Astăzi ne vom axa pe una simplă și eficientă
- În practică,  $h(x) = x \% p$ , unde  $p$  este un număr prim
- Vom discuta în detaliu la cursul următor despre metodele de alegere ale funcției de dispersie

# Rezolvarea coliziunilor

- Vom discuta în cursul următor mai multe metode, pentru moment, voi alege înlănțuire.
- Hai să codăm :)
  - <https://leetcode.com/problems/two-sum/>
  - Practic implementăm de mână inserarea și căutarea într-un hash



# Problemă

- <https://www.infoarena.ro/problema/strmatch>
- Vrem să calculăm toate aparițiile unui sir mai mic (subșir) într-un sir mai mare
- Soluții?

Text : A A B A A C A A D A A A B A A B A

Pattern : A A B A

A A B A

A A B A

A A B A A C A A D A A B A A B A  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

A A B A

Pattern Found at 0, 9 and 12

# Pattern Matching

Algoritmul Rabin Karp!

1. Calculăm hash-ul pentru sirul mai mic
2. Calculăm hash-ul pentru toate sirurile de aceeași lungime din sirul mai mare

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A A B A

A A B A

A A B A A C A A D A A B A A B A  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

A A B A

Pattern Found at 0, 9 and 12

# Pattern Matching

Algoritmul Rabin Karp!

1. Calculăm hash-ul pentru sirul mai mic
2. Calculăm hash-ul pentru toate sirurile de aceeași lungime din sirul mai mare

set\_size = 10 (decimal)

1234

$$1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

$$(1 \times 10^2 + 2 \times 10^1) 10 + 3 \times 10^1 + 4 \times 10^0$$

$$((1 \times 10^2 + 2 \times 10^1) + 3) 10 + 4 \times 10^0$$

$$((1 \times 10^1 + 2) 10 + 3) 10 + 4 \times 10^0$$

<new hash = old hash \* alphabet\_size + letter>

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A A B A

A A B A

A A B A A C A A D A A B A A B A  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Pattern Found at 0, 9 and 12

# Pattern Matching

- Cum calculăm rolling hash?
- Ce probleme ar putea apărea?
  - Dacă două șiruri au hash-uri egale? Sunt egale?
- Soluții?
  - Verificăm fiecare potrivire
    - Ce se întâmplă dacă avem:
      - aaa
      - aaaaaaaaaa
      - O(n\*m)
    - Facem 2 hash-uri și vedem dacă ambele sunt egale
      - Dacă ambele sunt egale, atunci suntem OK.

set\_size = 10 (decimal)

1234

$$1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

$$(1 \times 10^2 + 2 \times 10^1) \cdot 10 + 3 \times 10^1 + 4 \times 10^0$$

$$((1 \times 10^2 + 2 \times 10^1) + 3) \cdot 10 + 4 \times 10^0$$

$$((1 \times 10^1 + 2) \cdot 10 + 3) \cdot 10 + 4 \times 10^0$$

<new hash = old hash \* alphabet\_size + letter>

Codăm:

- <https://www.infoarena.ro/problema/strmatch>
- implementare posibila

# Hash-uri

Tabele cu adresare directă. Tabele de dispersie

# Prewatch

Prewatch:

- Video1 (pentru sync-video: <https://youtu.be/JZHBa-rLrBA>)
  - Ideal tot videoul
  - **Măcar de la 29 la 1:02 (33 de minute)**
- Video2 (pentru sync-video: [https://youtu.be/0M\\_klqhwbf0](https://youtu.be/0M_klqhwbf0))
  - Ideal tot videoul :)

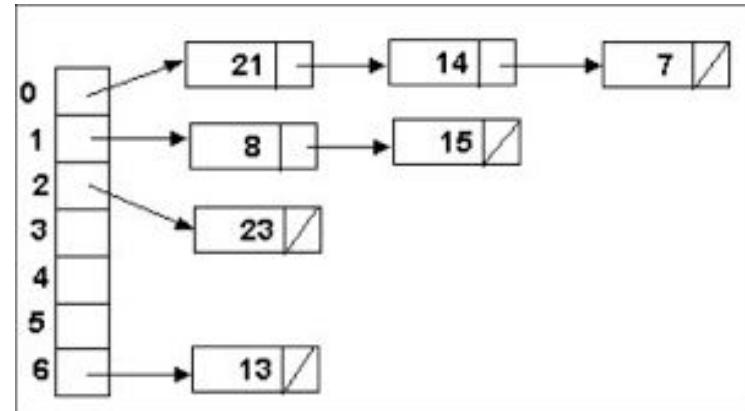
# Funcții de dispersie

Funcții de dispersie:

- Am zis săptămâna trecută că, pentru moment, folosim  $h(x) = x \% p$ , unde  $p$  este un număr prim.

Rezolvarea coliziunilor:

- Am spus că vom ține o listă înlățuită



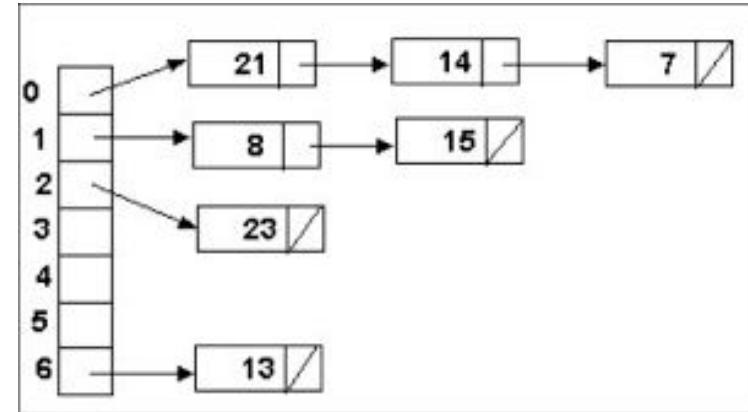
# Funcții de dispersie

Funcții de dispersie:

- Am zis săptămâna trecută că, pentru moment, folosim  $h(x) = x \% p$ , unde  $p$  este un număr prim.

Rezolvarea coliziunilor:

- Am spus că vom ține o listă înlățuită
- Complexitate:
  - $O(1)$  căutare ?
  - Ce se întâmplă dacă  $p$  este  $\sim \text{sqrt}(n)$ ?
    - $O(\text{sqrt } n)$  pe căutare
    - Dacă datele nu sunt rele avem  $O(n/p) \dots p$  nu trebuie să fie mult mai mic decât  $n$ , ideal mai mare



# Funcții de dispersie

Funcții de dispersie:

- Am zis săptămâna trecută că, pentru moment, folosim  $h(x) = x \% p$ , unde  $p$  este un număr prim.
- Ce ne dorim de la o funcție hash? **Ipoteza dispersiei uniforme simple:**
  - Fiecare cheie se poate dispersa cu aceeași probabilitate în oricare din cele  $m$  locații.
  - $f(x) = \text{cel mai reprezentativ bit a lui } x$  - nu e bună
    - $f(24) = f(18) = 16 \rightarrow$  cheile nu au aceeași probabilitate să ajungă pe cele  $m$  locații
  - În practică, nu putem satisface perfect regula, dar ne dorim să fim cât mai aproape
- [https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall  
2005/video-lectures/lecture-7-hashing-hash-functions/](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-7-hashing-hash-functions/) (28:38)
- <https://drive.google.com/drive/u/0/folders/1aNqJk0kfKZszEOzvPLh81hvH7OSKbl1g>

# Functii de dispersie

Functii de dispersie:

- Am zis săptămâna trecută că, pentru moment, folosim  $h(x) = x \% p$ , unde  $p$  este un număr prim.
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-7-hashing-hash-functions/> (28:38) (recomandare)
- Am vorbit despre Functii de dispersie și ne-am uitat la diverse metode
  - Metoda diviziunii (discutată și data trecută)
  - Metoda multiplicării (folosită în practică mult, pentru că este mai rapidă)

# Dispersie universală

Fie  $H$  o colecție finită de funcții de dispersie, care transformă un univers dat  $U$  al cheilor, în domeniul  $\{0, 1, \dots, m-1\}$ .

O astfel de colecție se numește **universală** dacă, pentru fiecare pereche de chei distincte  $x, y \in U$ , numărul de funcții de dispersie  $h \in H$  pentru care  $h(x) = h(y)$  este exact  $|H| / m$ .

Cu alte cuvinte, cu o funcție de dispersie aleasă aleator din  $H$ , șansa unei coliziuni între  $x$  și  $y$  când  $x \neq y$  este exact  $1/m$ , care este exact șansa unei coliziuni dacă  $h(x)$  și  $h(y)$  sunt alese aleator din mulțimea  $\{0, 1, \dots, m - 1\}$ .

# Dispersie universală

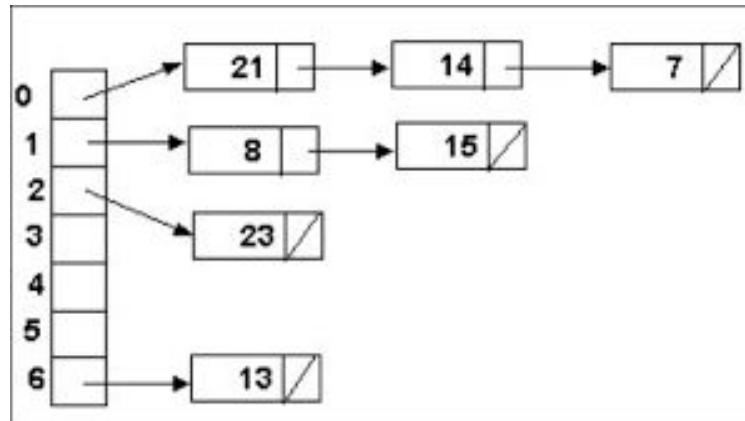
Următoarea teoremă arată că o clasă universală de funcții de dispersie dă un comportament bun în cazul mediu.

**Teorema 6.1:** Dacă  $\mathbf{h}$  este aleasă dintr-o colecție universală de funcții de dispersie și este folosită pentru a dispersa  $n$  chei într-o tabelă de dimensiune  $m$ , unde  $n \leq m$ , numărul mediu de coliziuni în care este implicată o cheie particulară  $x$  este mai mic decât 1.

# Rezolvarea coliziunilor

## Rezolvarea coliziunilor

- Am zis săptămâna trecută că, pentru moment, folosim **înlănțuirea**
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-7-hashing-hash-functions/> (50:00)



# Rezolvarea coliziunilor

## Rezolvarea coliziunilor

- Am zis săptămâna trecută că, pentru moment, folosim **înlănțuirea**
- Am urmărit 15 minute din cursul de la MIT, pornind cu 50:00, unde se vorbește despre metoda adresării directe
- În cazul adresării directe, au fost evidențiate 2 metode de calculare a poziției elementului în tabelul de dispersie de mărime  $m$ :
  - **Testare liniară:** 
$$h(x, i) = (h(x, 0) + i) \% m$$
  - **Hash dublu:** 
$$h(x, i) = (h1(x) + i * h2(x)) \% m$$

# Rezolvarea coliziunilor

Altă metodă:

- [https://en.wikipedia.org/wiki/Cuckoo\\_hashing](https://en.wikipedia.org/wiki/Cuckoo_hashing)

# HEAPURI

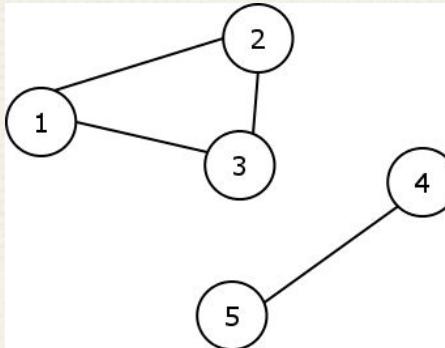


# Heapuri

- ❖ Definiții
  - Graf
  - Arbore
  - Arbore Binar
  - Heap
- ❖ Heapuri - inserare, stergere
- ❖ Heapify (creare heap în timp liniar)
- ❖ Lazy Deletion
- ❖ Binomial Heap
- ❖ Fibonacci Heap

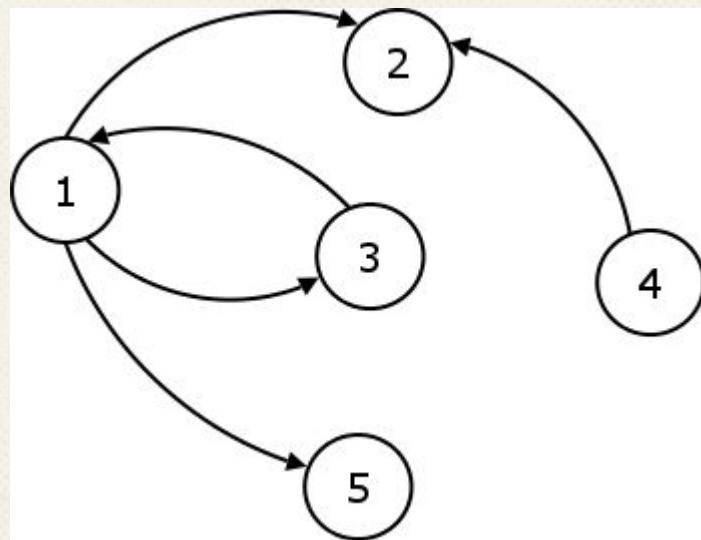
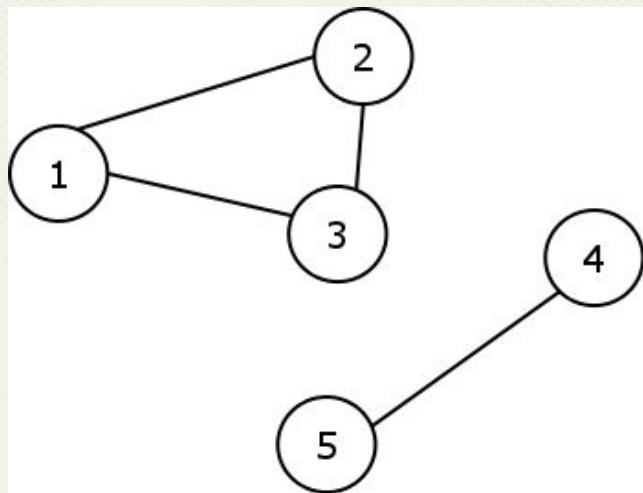
# Grafuri

- ❖ Ce este un graf?
- ❖ Un graf este o pereche de multimi  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ , unde:
  - $\mathbf{V}$  este multimea de noduri (vertex / vertices),
  - $\mathbf{E}$  este multimea de muchii



# Grafuri

- ❖ Graf orientat vs graf neorientat



# 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

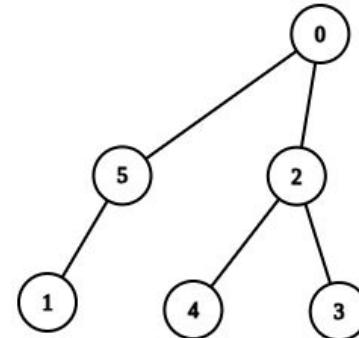
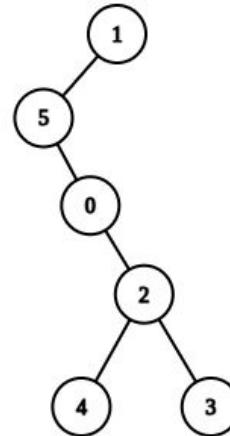
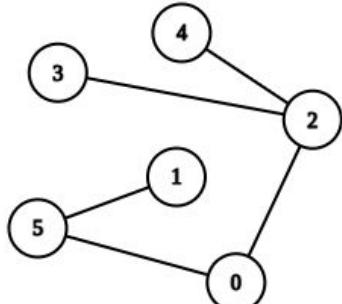
- ❖ Proprietăți:
  - Un arbore cu  $n \geq 2$  vârfuri conține minim 2 frunze
  - Ce este o frunză?
    - Un nod cu gradul 1 (și rădăcina poate să fie frunză)

# 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



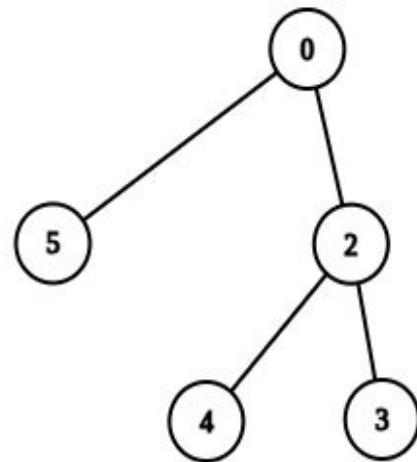
# Arbore binari

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

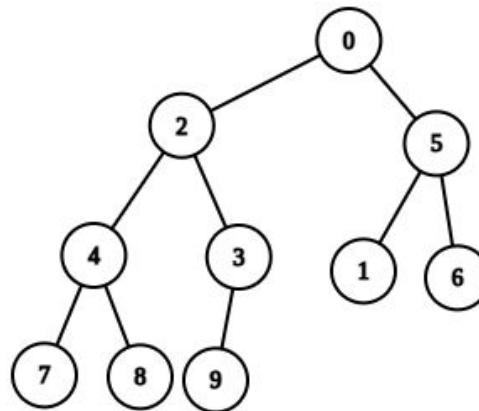
Copiii unui nod sunt numiți copilul stâng (Left, L) și copilul drept (Right, R).

# Arbore binari

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



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

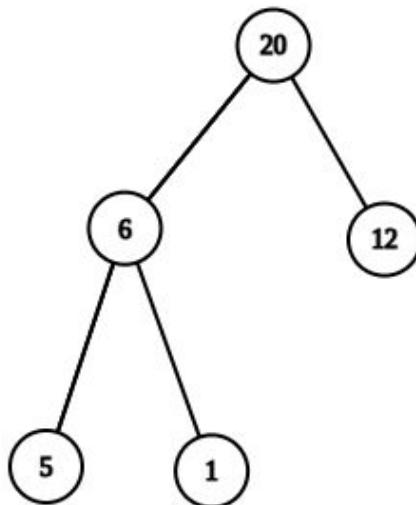


# Arbore binari - Proprietăți

- ❖ Exercițiu:
  - Numărul de noduri ale unui arbore binar cu înălțimea  $h$  este între (?) și (?)
    - $h$  (dacă este 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

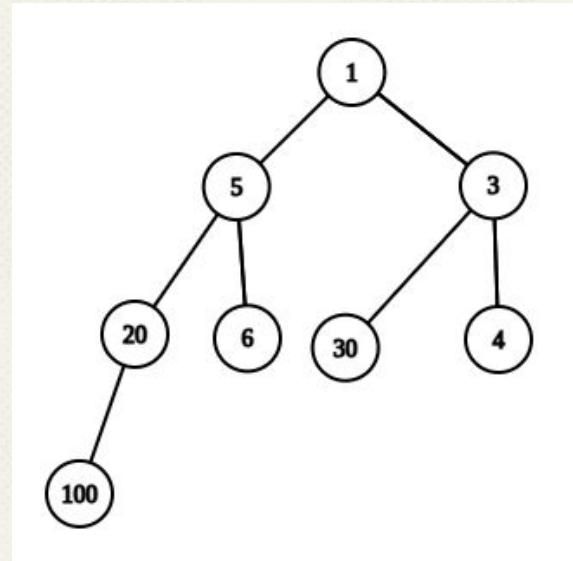
# Heapuri

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

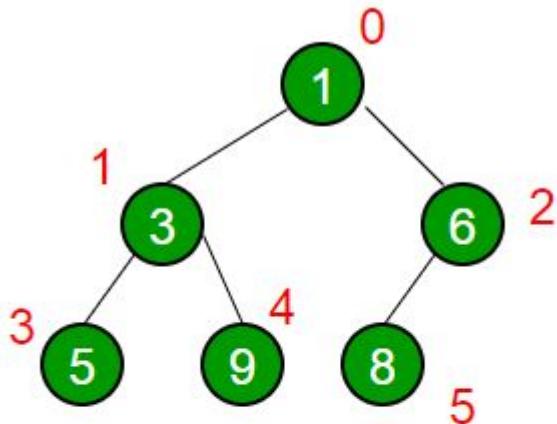


# Heapuri

- ❖ 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

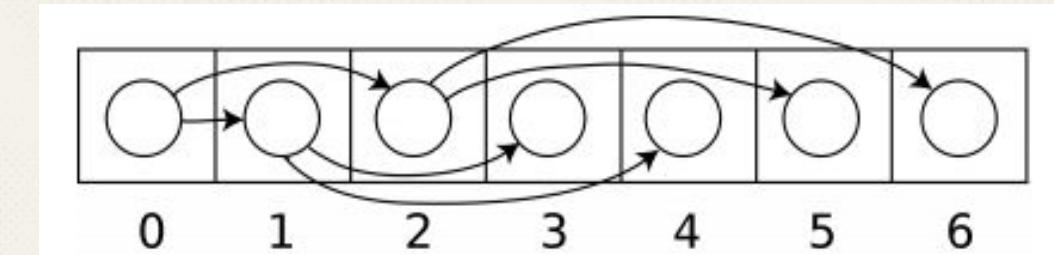
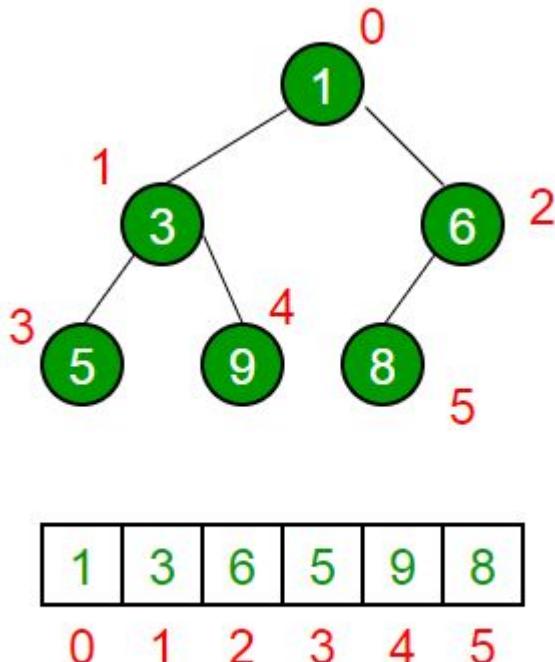


1	3	6	5	9	8
---	---	---	---	---	---

0 1 2 3 4 5

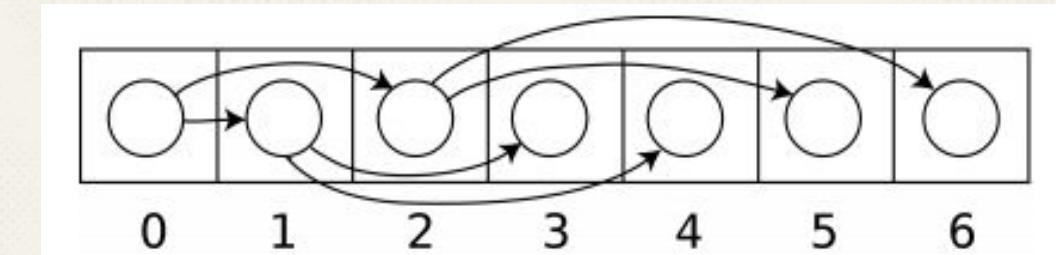
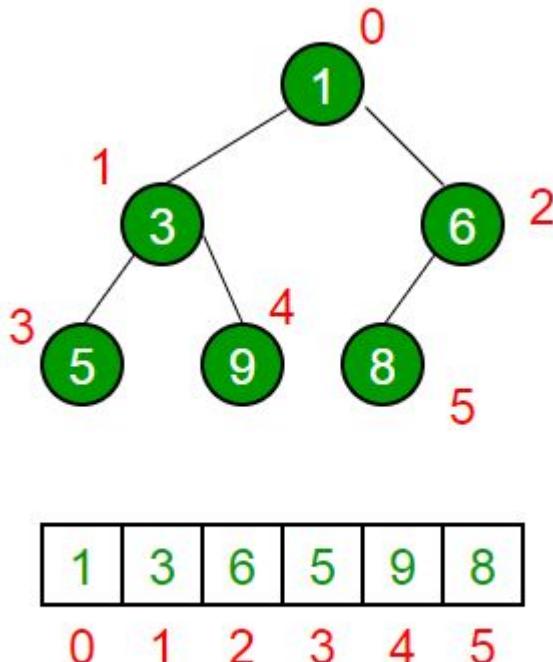
Un arbore binar complet poate fi reprezentat ca un vector!

# Heapuri - Reprezentare



- 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

# Heapuri - Reprezentare

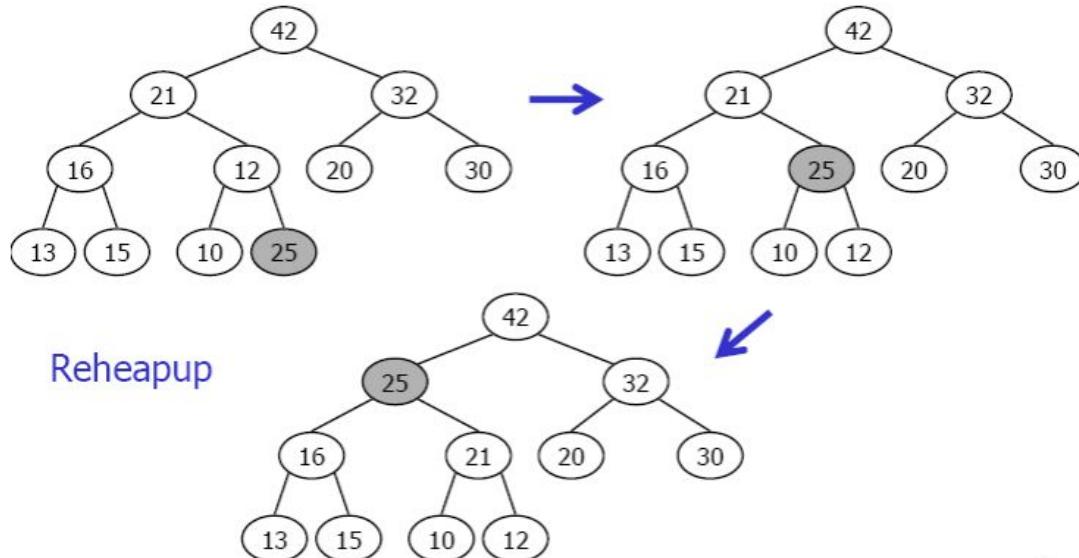


- Parinte( $i$ ) =  $(i - 1) / 2$ , unde  $i$  este indicele nodului curent
- IndexStanga( $i$ ) =  $2 * i + 1$ , unde  $i$  este indicele nodului curent
- IndexDreapta( $i$ ) =  $2 * i + 2$ , unde  $i$  este indicele nodului curent

Înălțime:  $\log n$  !!

# Heapuri - Urcă (percolate)

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



**O(log n)**

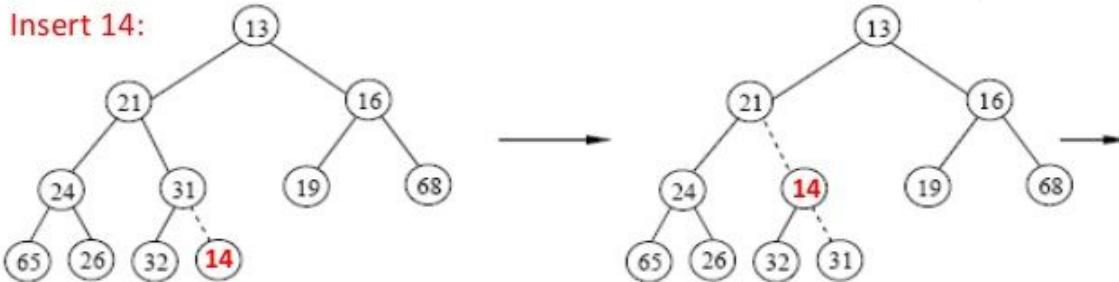
# Urca cod

```
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;  
        }  
    }  
}
```

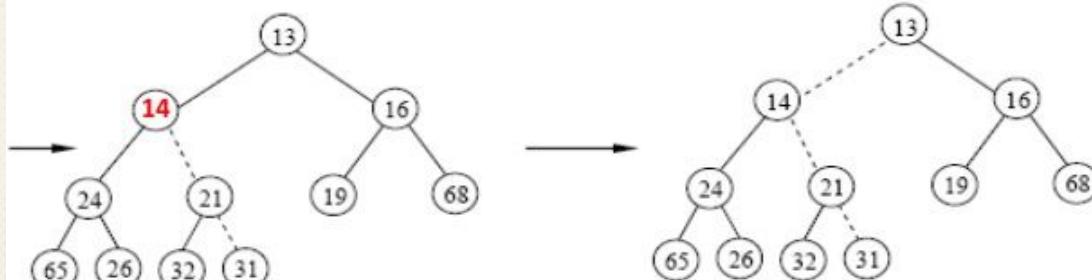
# Heapuri - Inserare

Insert new element into min-heap

Insert 14:



**O(log n)**



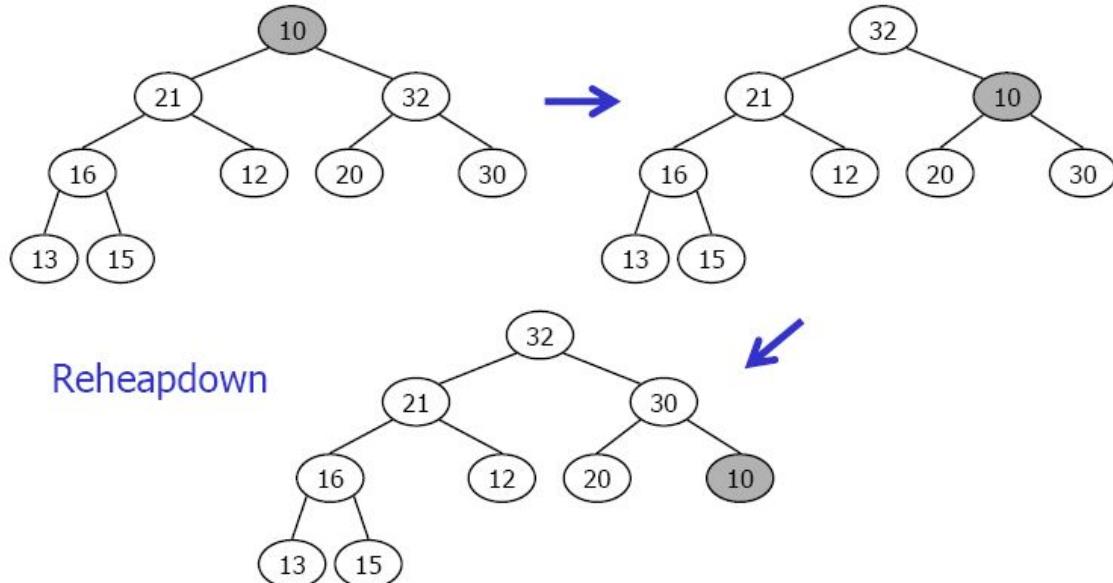
The new element is put to the last position, and **ReheapUp** is called for that position.

# 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)**

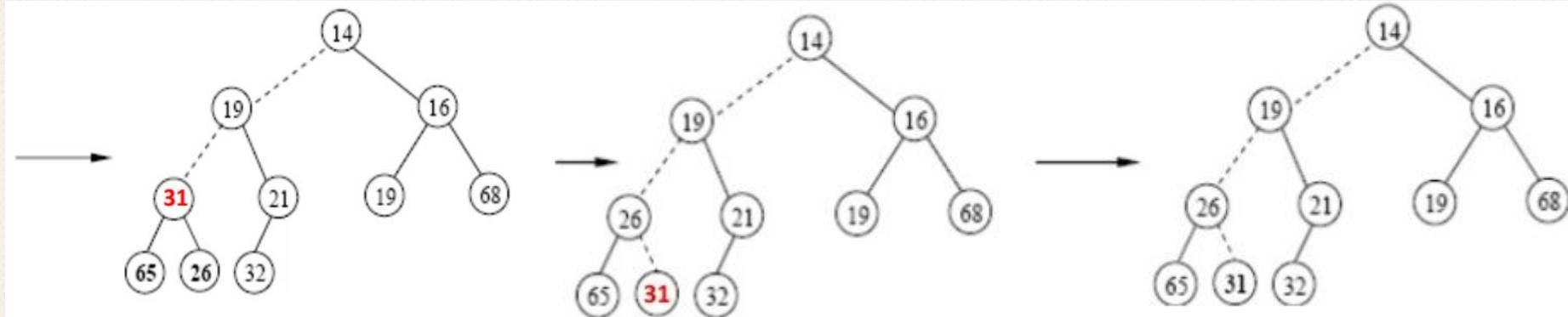
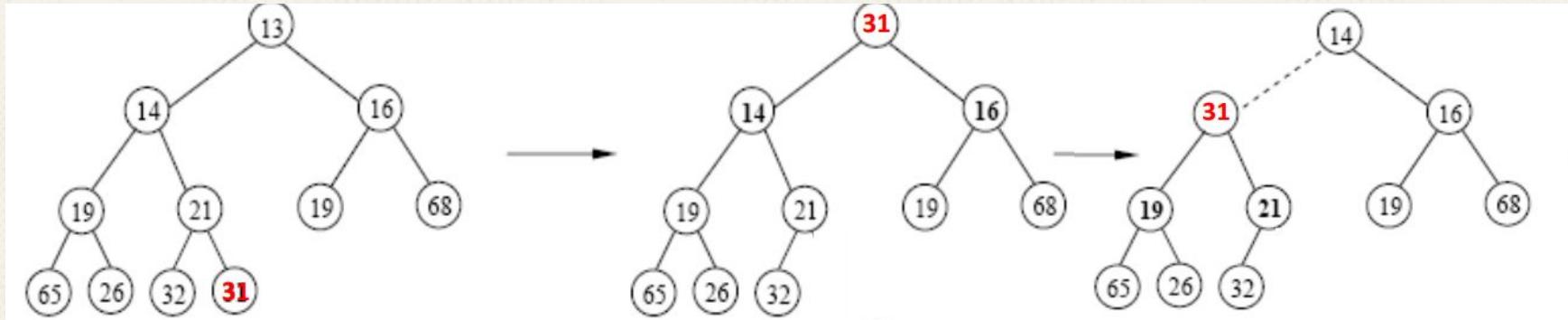
# Coboara cod

```
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]);  
        cobaara(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.

**O(log n)**

# 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)**

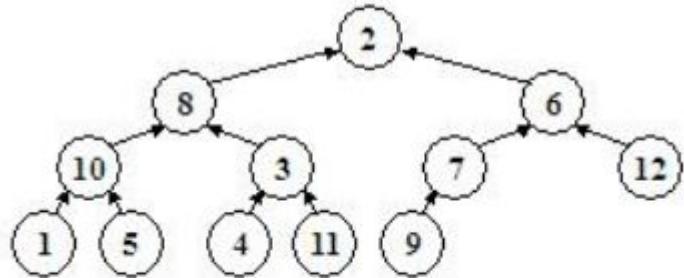
# Heapify

## Construire heap

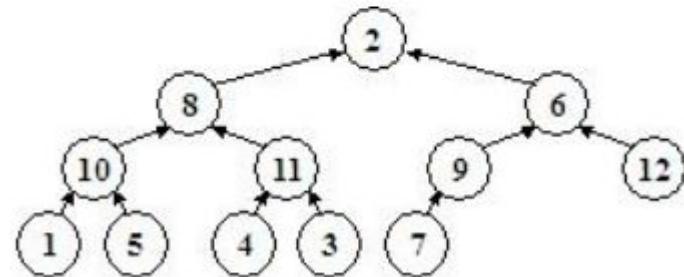
- ❖ Inserăm **n** elemente - **O(n log n)**
- ❖ Liniar:
  - Coborâm fiecare element începând de jos în sus

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

# Heapify

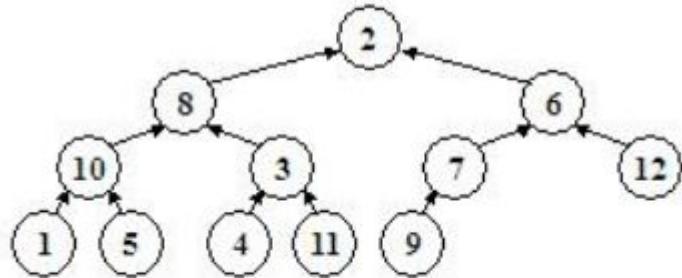


*Nivelul frunzelor este organizat*

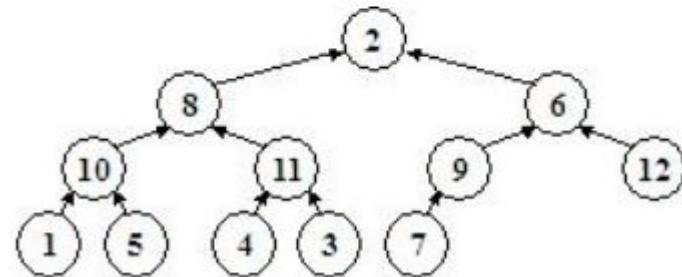


*Ultimele două niveluri sunt organizate*

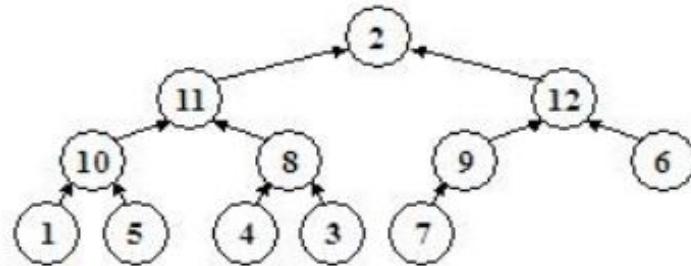
# Heapify



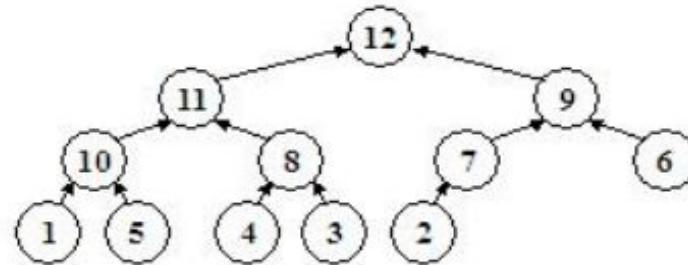
*Nivelul frunzelor este organizat*



*Ultimele doua niveluri sunt organizate*



*Ultimele trei 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
  - 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?

- ❖ **Problemă:** ?

# Problemă

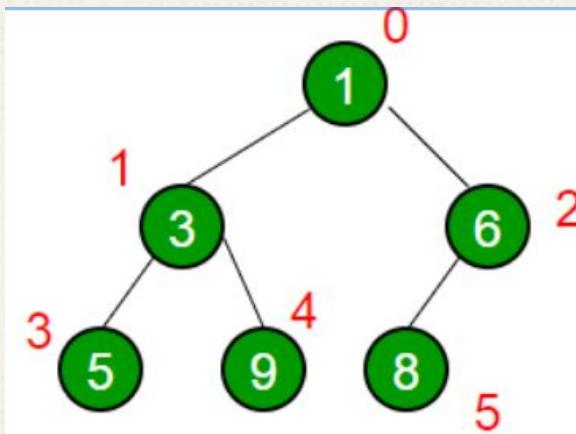
- ❖ Se dau multe operații de genul:
  - Inserare număr -  $O(\log n)$
  - Afisare minim -  $O(1)$
  - Elimină indice

Cum putem folosi un heap?

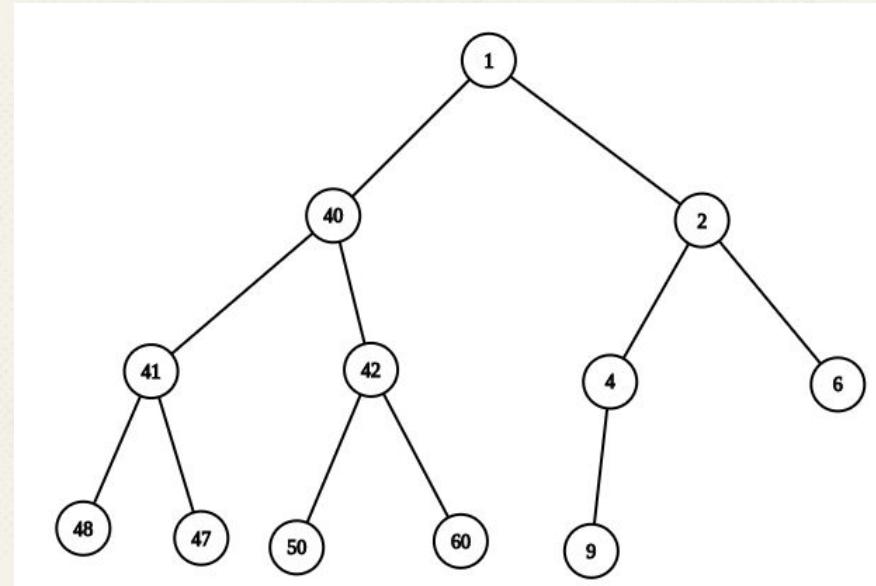
- ❖ **Problemă:** Eliminare număr (Nu știm indexul din heap și fără să știm indexul nu putem elmina în  $\log n$ )

# 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
  - Afişare 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 in heap e  $O(n)$
- ❖ Operație ce va fi folosită în general la arbori, nu doar pentru heapuri

# Heapuri - Complexitate

Operatie	Timp Mediu	Cel mai rau caz
Spatiu	$O(n)$	$O(n)$
Cautare	$O(n)$	$O(n)$
Inserare	$O(1)$ $n/2 * 0 + n/4 * 1 + n/8 * 2 \dots \approx 1$	$O(\log n)$
Stergere	$O(\log n)$	$O(\log n)$
Cautare 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

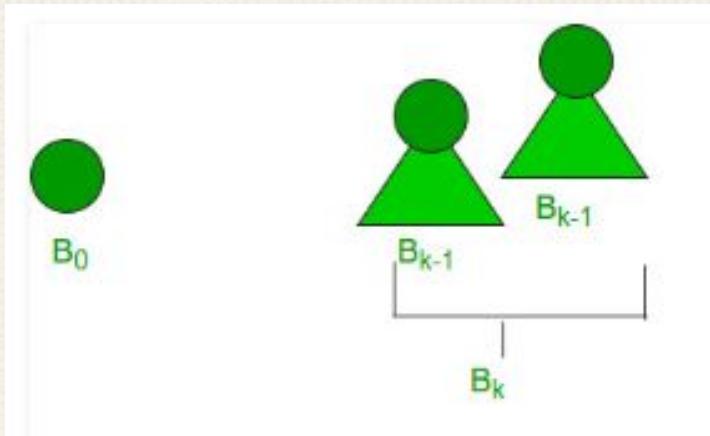
## ❖ Motivație:

- 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

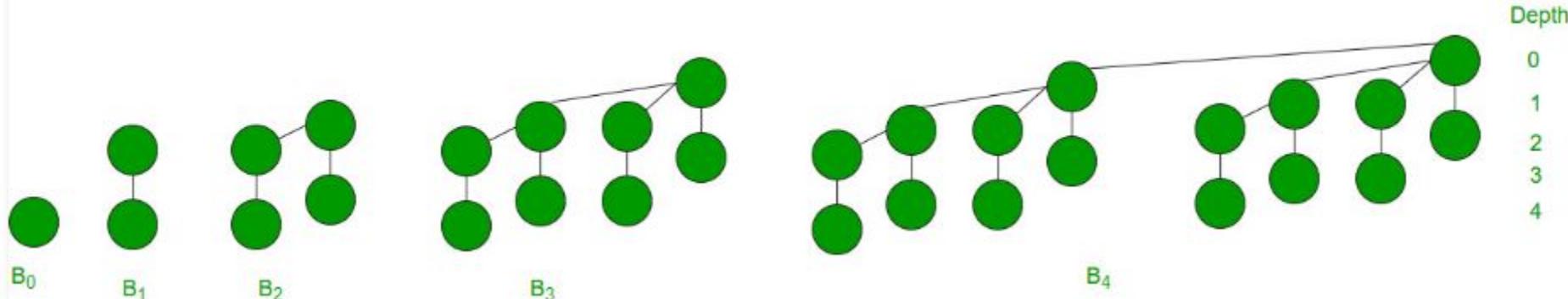
- ❖ Un arbore binomial de ordin 0 are un nod (rădăcina)
- ❖ Un arbore binomial de ordin K poate fi format prin reuniunea a doi arbori binomiali de mărime  $K-1$ , făcând pe unul dintre ei fiul stâng al celuilalt



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

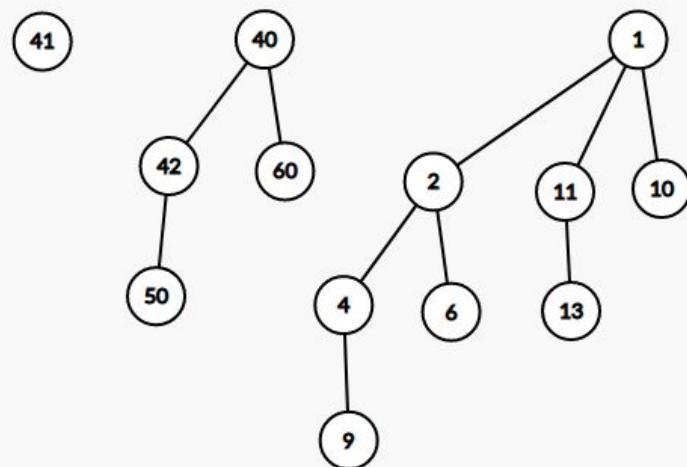


# Heapuri Binomiale

- ❖ Un Heap Binomial este o colecție de Arbori Binomiali, fiecare dintre ei având proprietatea de heap minim.
- ❖ Observatie: 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?

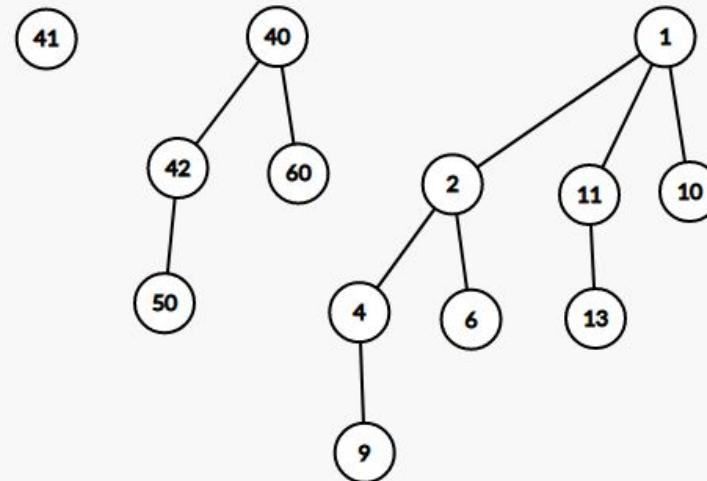
# Heapuri Binomiale

- ❖ Un Heap Binomial este o colecție de Arbori Binomiali, fiecare dintre ei având proprietatea de heap minim.
- ❖ Observatie: 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?



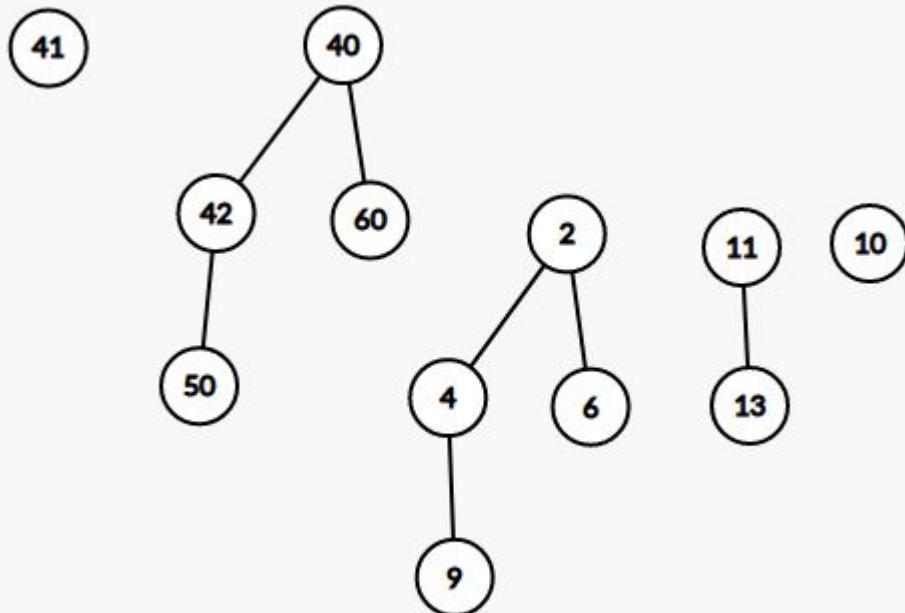
# Heapuri Binomiale - Căutare minim

- 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  $O(1)$



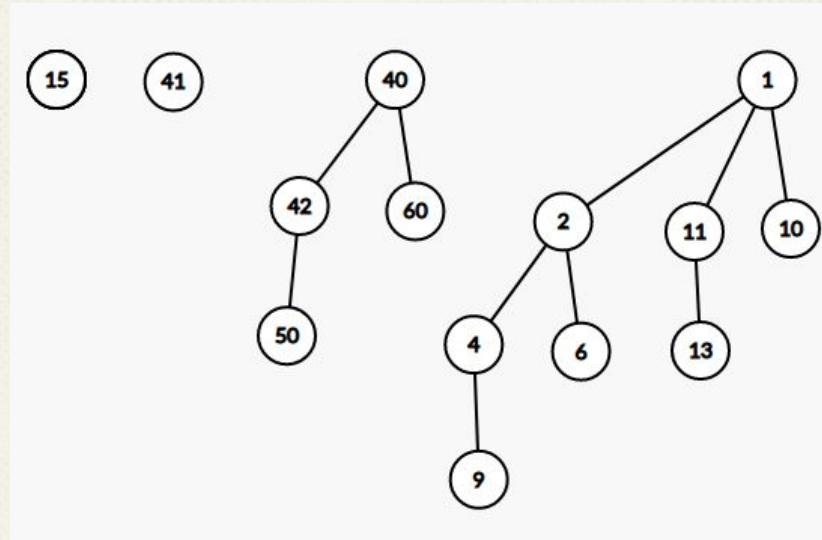
# Heapuri Binomiale - Extragerea minimului

- ❖ Eliminăm minimul
- ❖ Apoi facem reunire

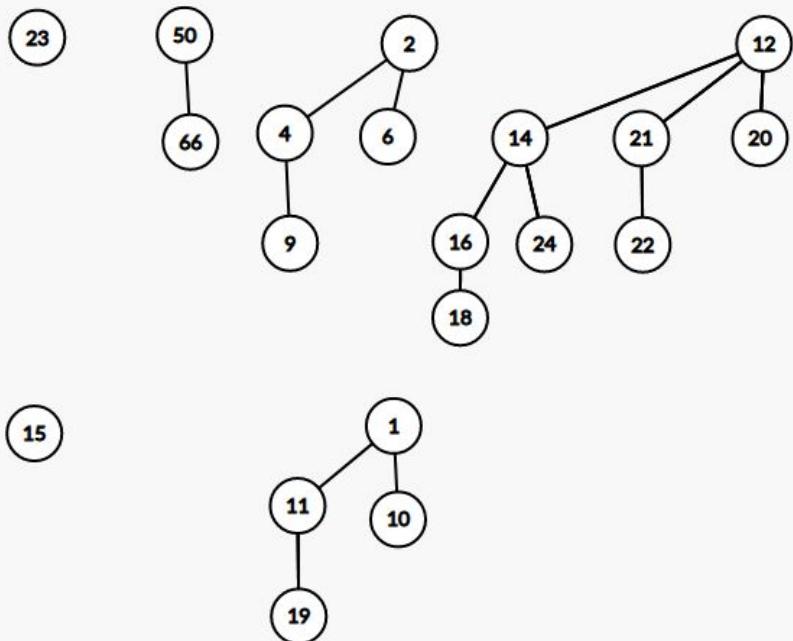


# Heapuri Binomiale - Inserare

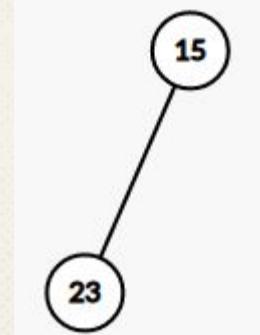
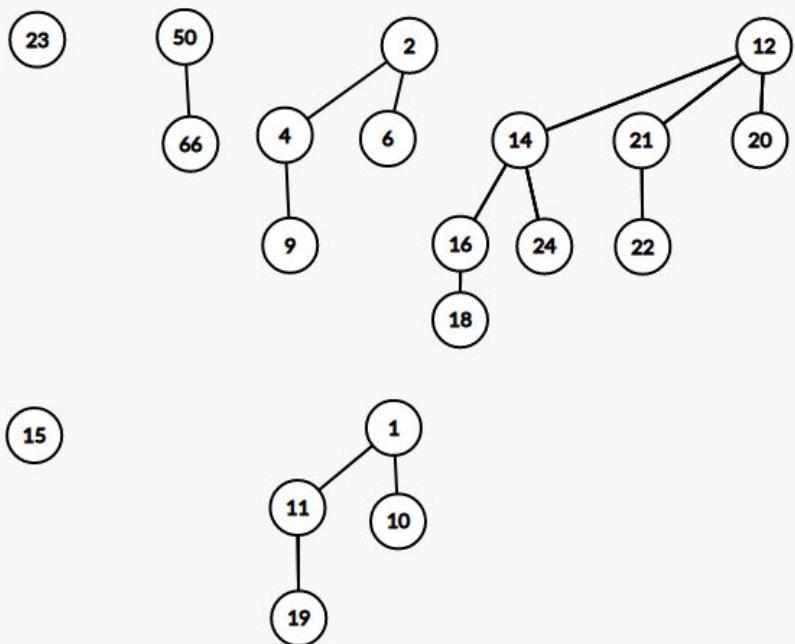
Adăugăm un arbore binomial de mărime 1, apoi apelăm reuniunea.



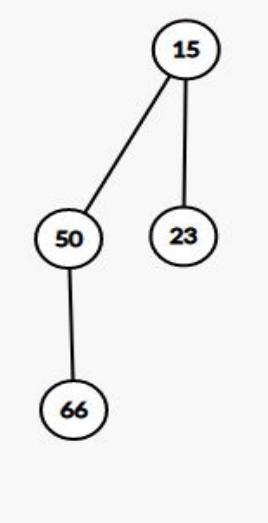
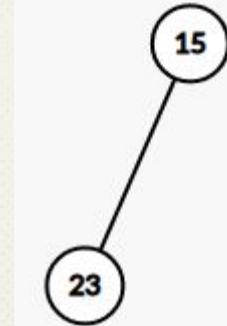
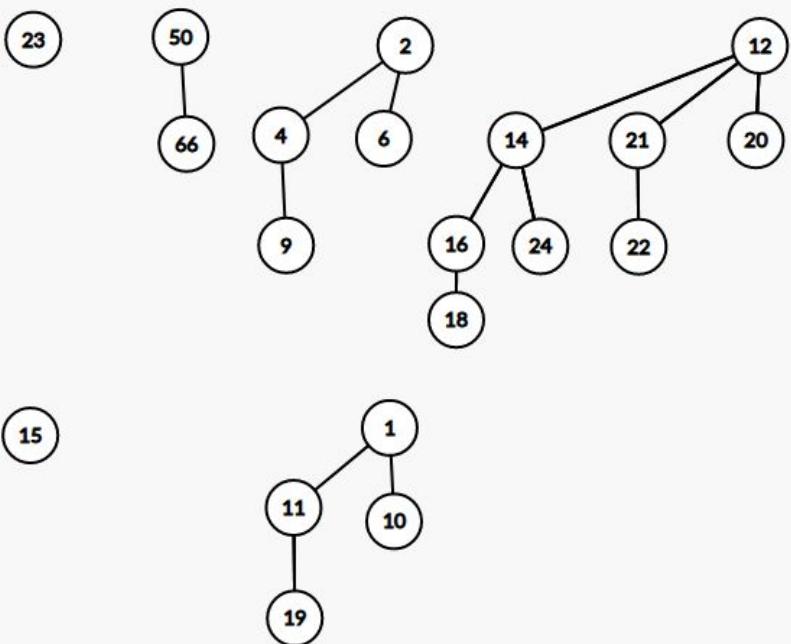
# Reuniune!



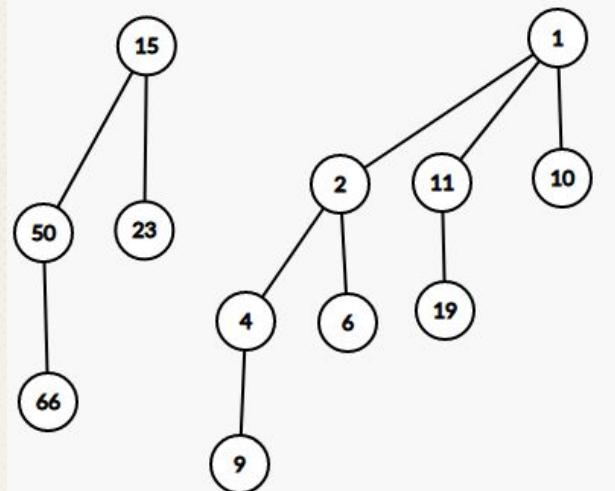
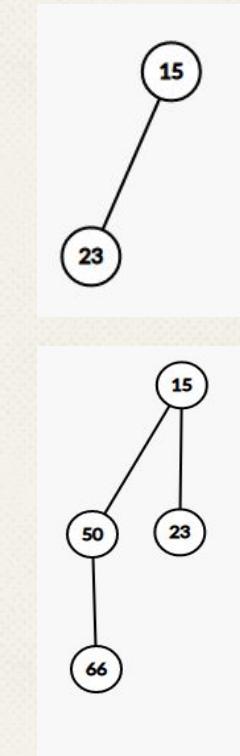
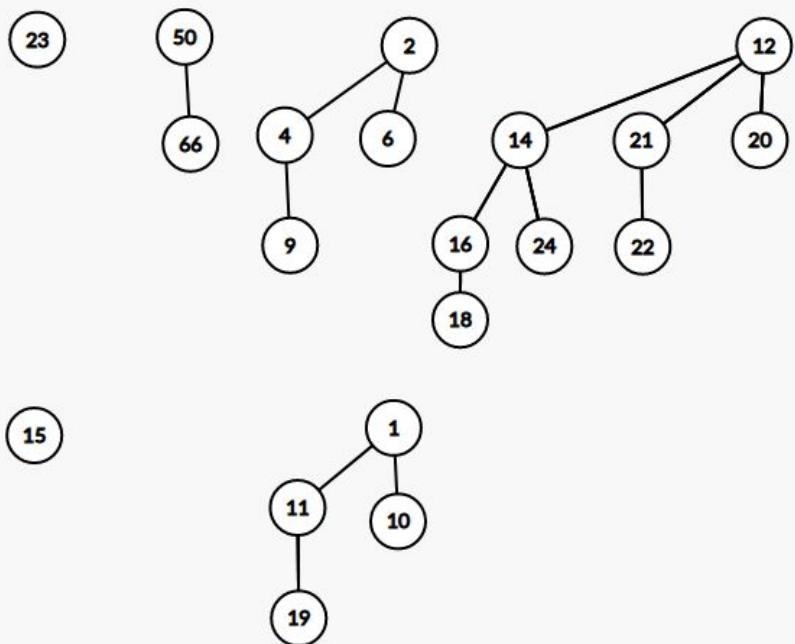
# Reuniune!



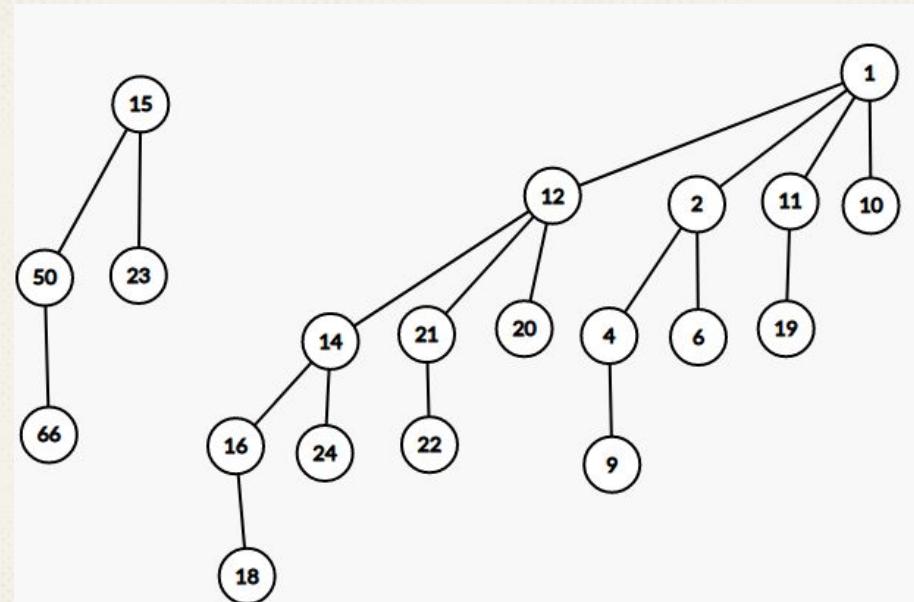
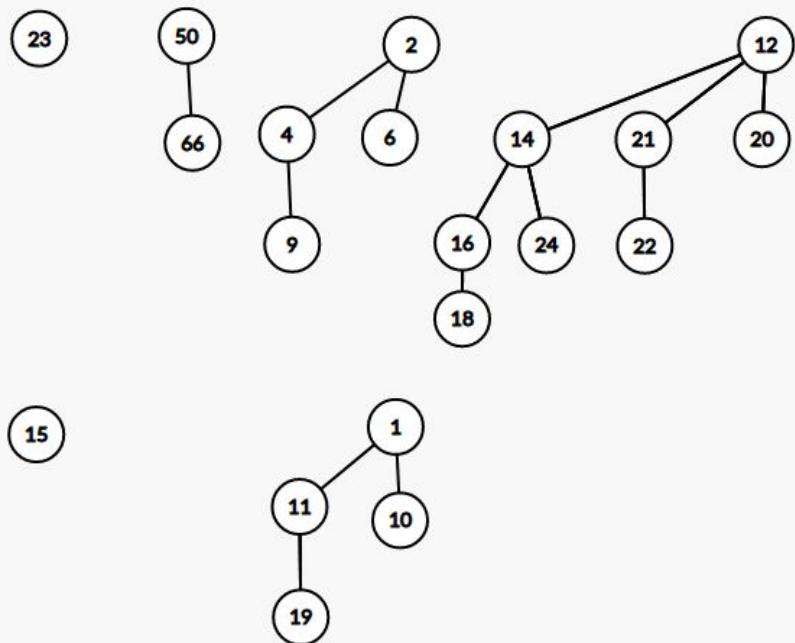
# Reuniune!



# 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.

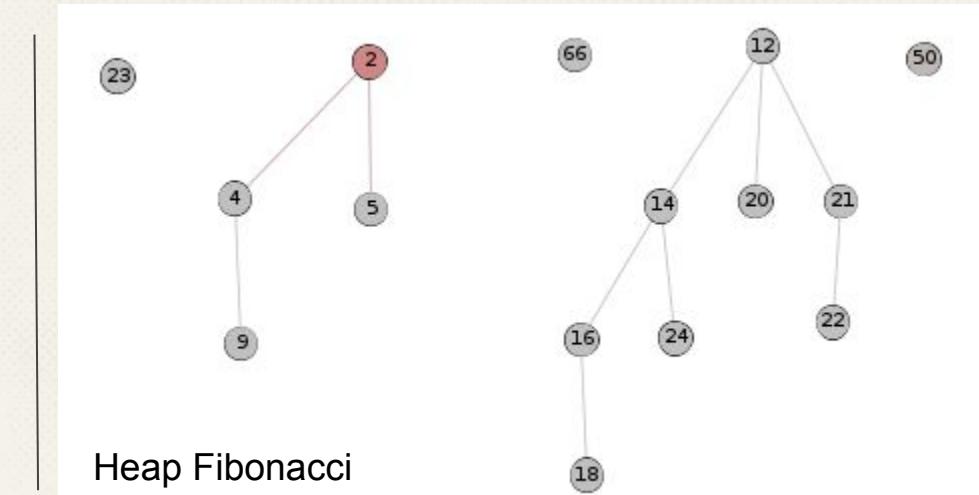
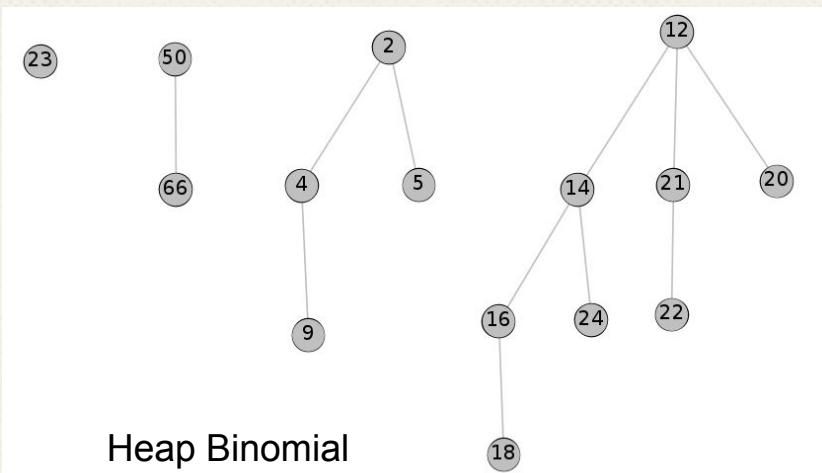
	Căutare Min	Ștergere Min	Inserare	Update	Reuniune
Heap Binar	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Heap Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$ (amortizat)	$\Theta(\log n)$	$O(\log n)$
Heap Fibonacci	$\Theta(1)$	$O(\log n)$ (amortizat)	$\Theta(1)$	$\Theta(1)$ (amortizat)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

# 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 compoziția sa au mărimi puteri ale lui 2. Acești vor fi arbori de mărime 1,..., k-1, dar nu neapărat sortați de la stânga la dreapta.

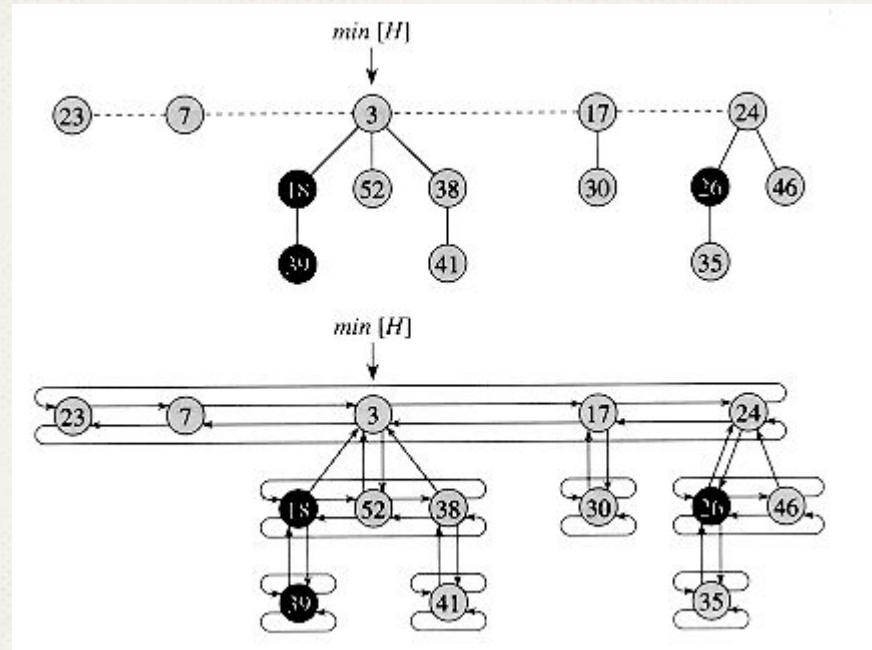
# Heapuri Fibonacci

- **Heapurile Fibonacci** sunt o colecție de arbori care au proprietatea de ordonare de heap (arborii nu trebuie să fie binomiali).
  - Arboarele dintr-un heap Fibonacci nu sunt ordonați.
  - Arboarele 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.



# Implementare

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

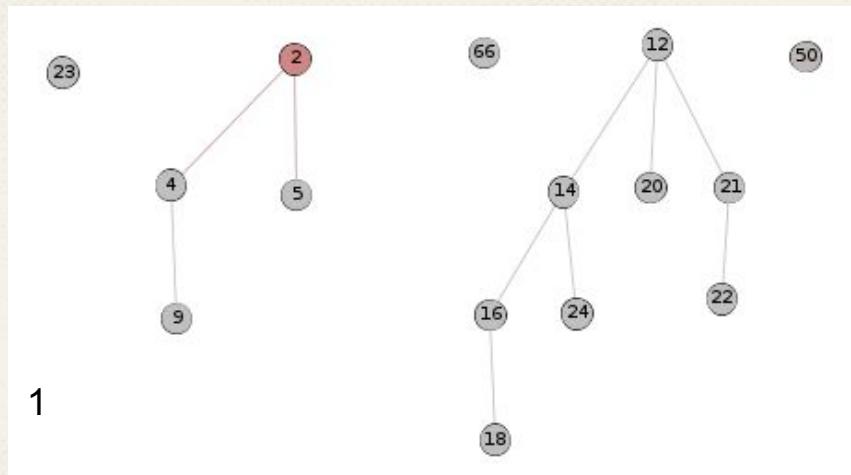


# Inserare nod

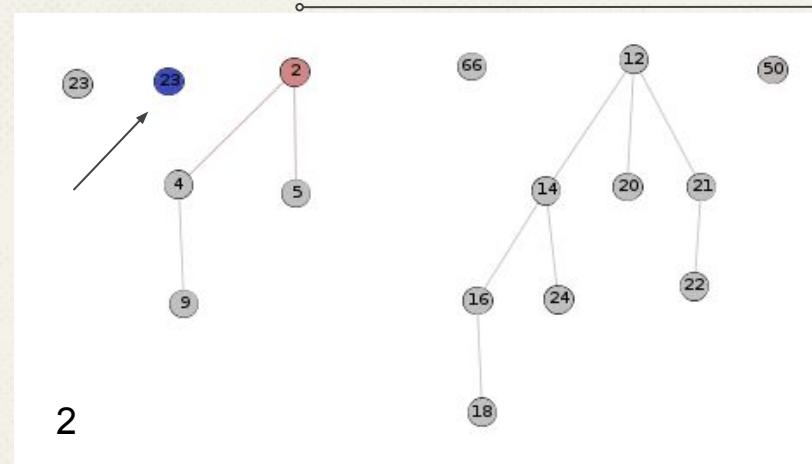
- Creăm un arbore cu un singur element
- Îl plasăm în stânga rădăcinii.
- **Nu facem reuniune!**

# Inserare nod

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



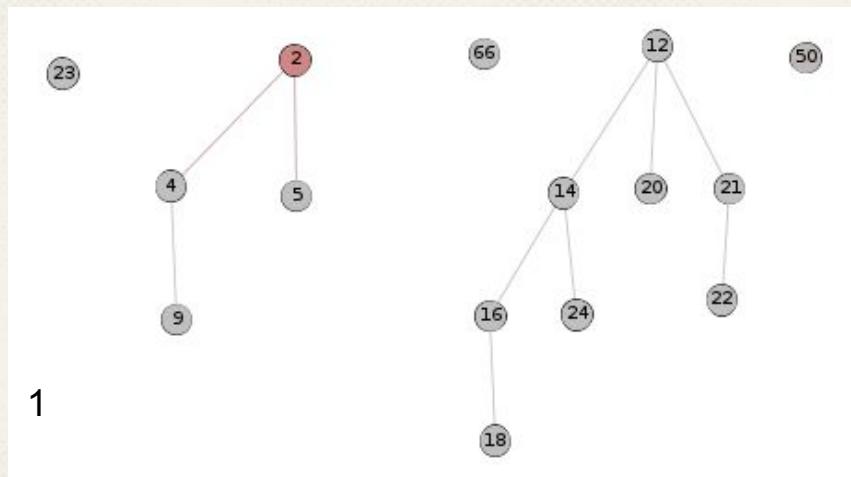
1



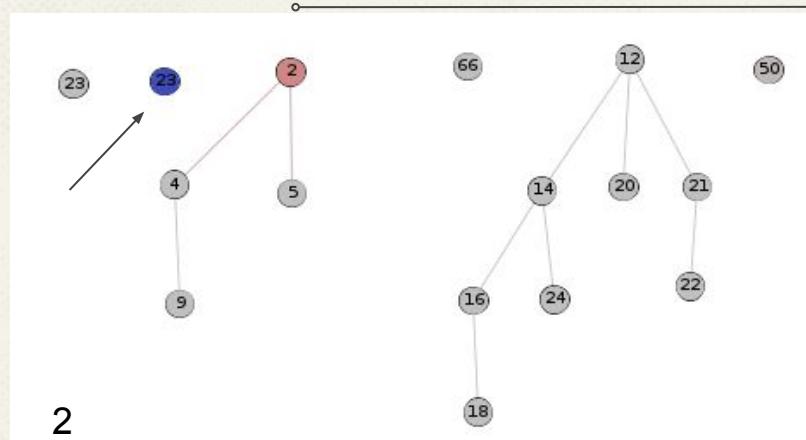
2

# Inserare nod

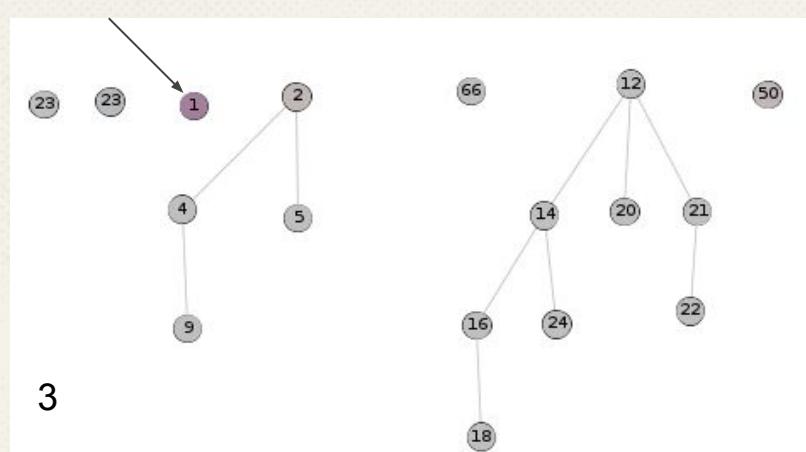
- Creăm un arbore cu un singur element
- Îl plasăm în stânga rădăcinii.
- **Nu facem reuniune!** →  $O(1)$



1



2



3

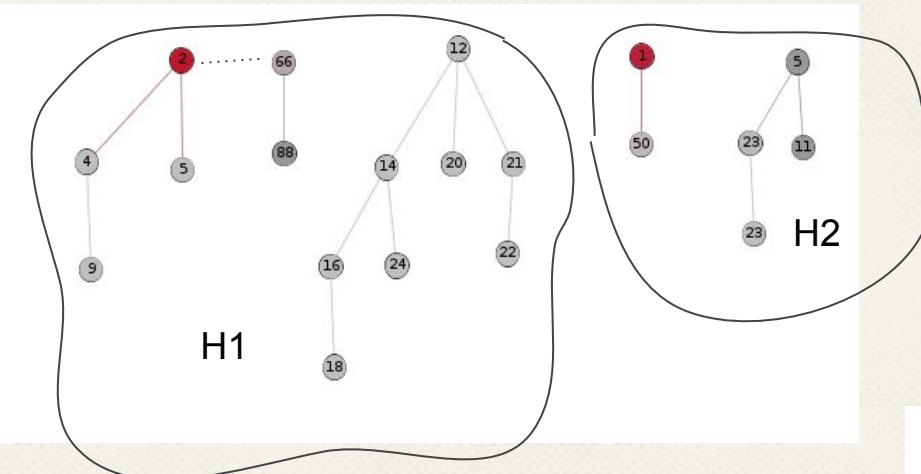
# Caută Minimul

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

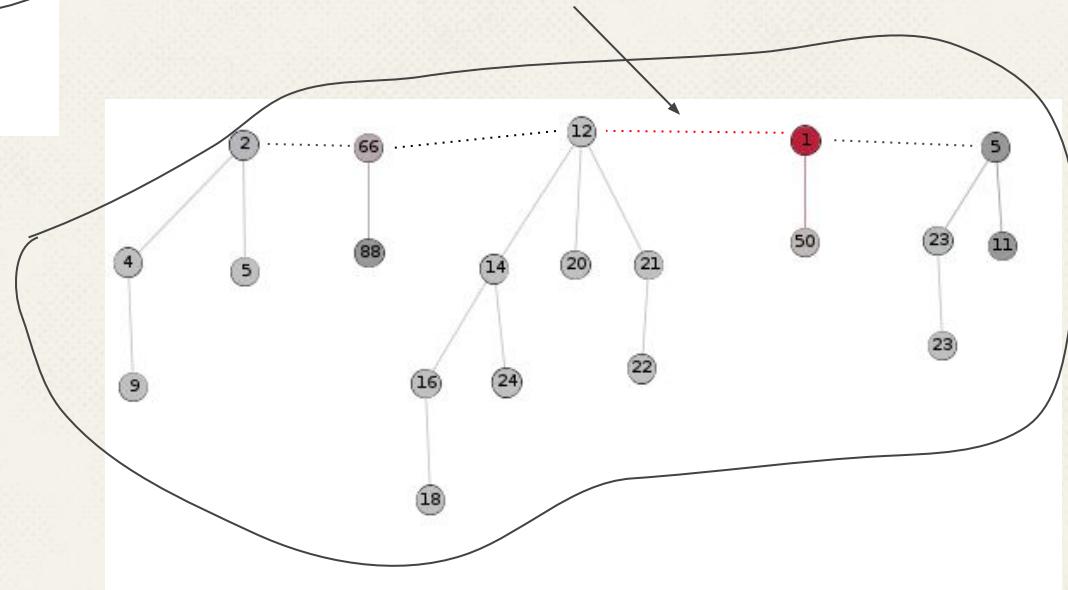
# Reuniune

- 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)!!**

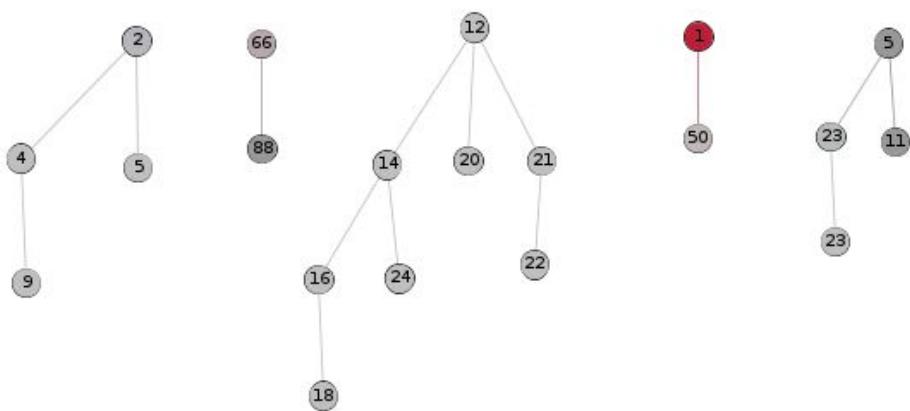
# Reuniune



$O(1)$

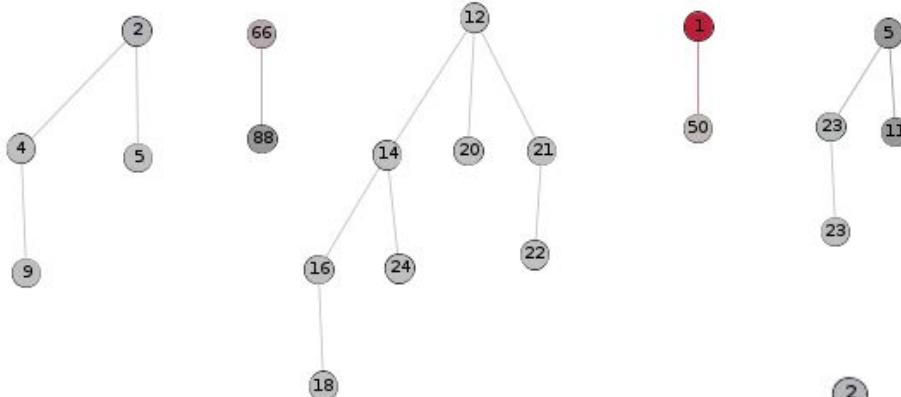


# Extragere minim



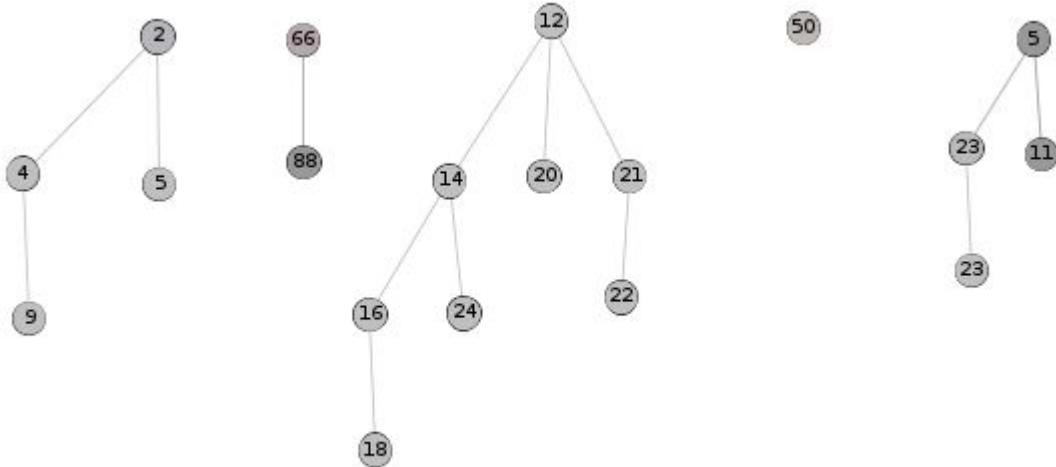
Extragem minim. Fiii să devin arbori liberi.

# Extragere minim

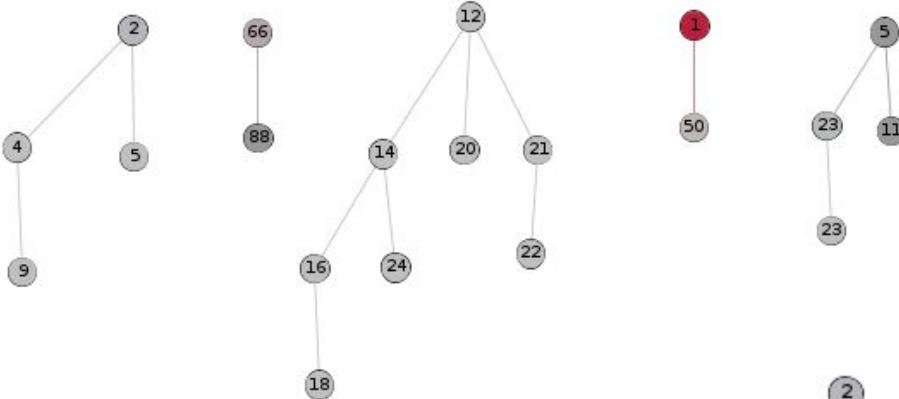


Unde e problema?

Extragem minim. Fiii să devin arbori liberi.



# Extragere minim

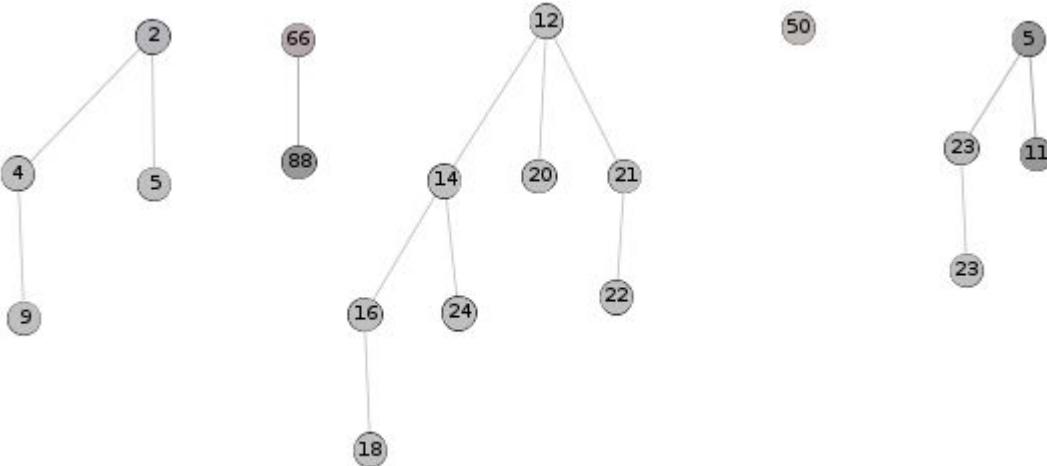


**Unde e problema?**

Nu știm care e minimul.  
Am putea avea **n** arbori  
cu 1 element.

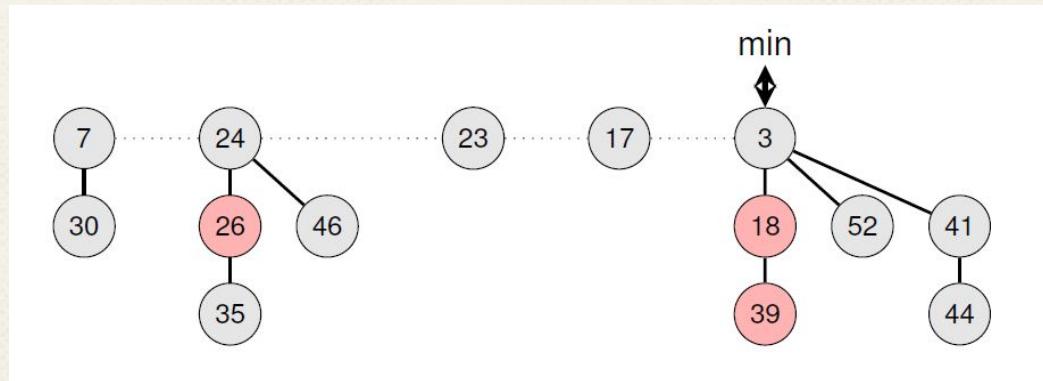
Dacă ștergem **n** elemente  
consecutive ne poate  
costa  $n^2$ ??

Extragem minim. Fiii săi devin  
arbori liberi.



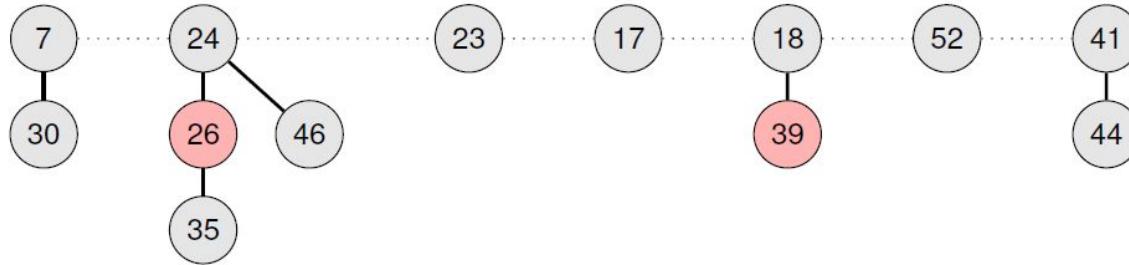
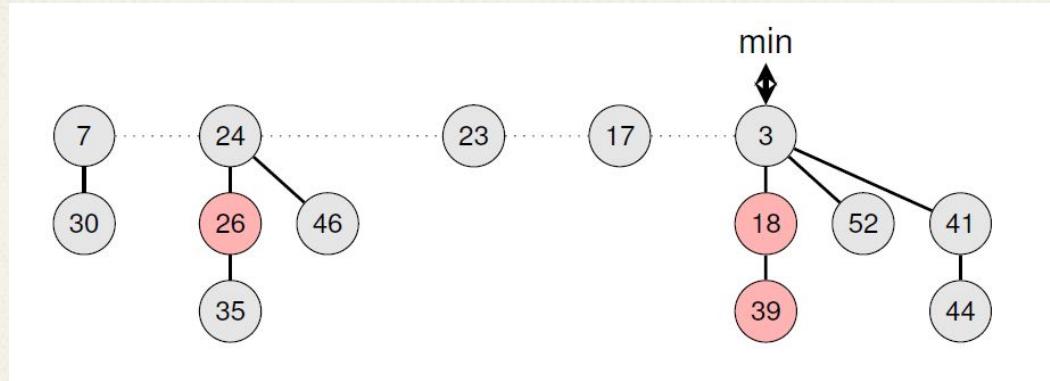
# Extragere minim

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



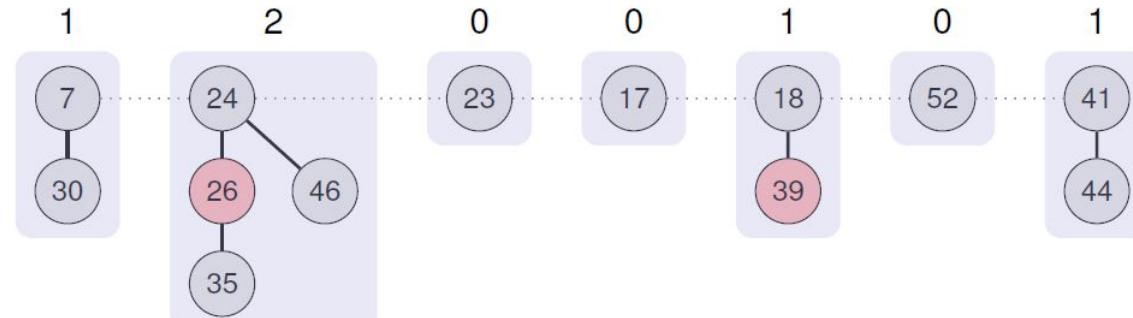
Eliminăm minimul, se creează multe “rădăcini”

# Extrageré minim

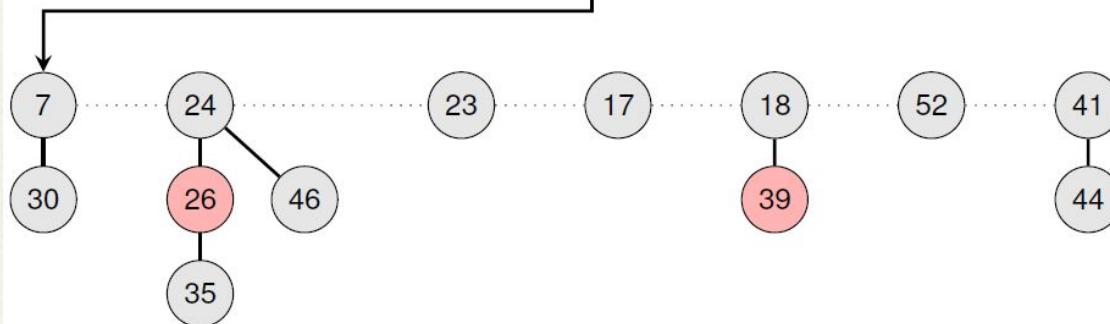


# Extragele minim

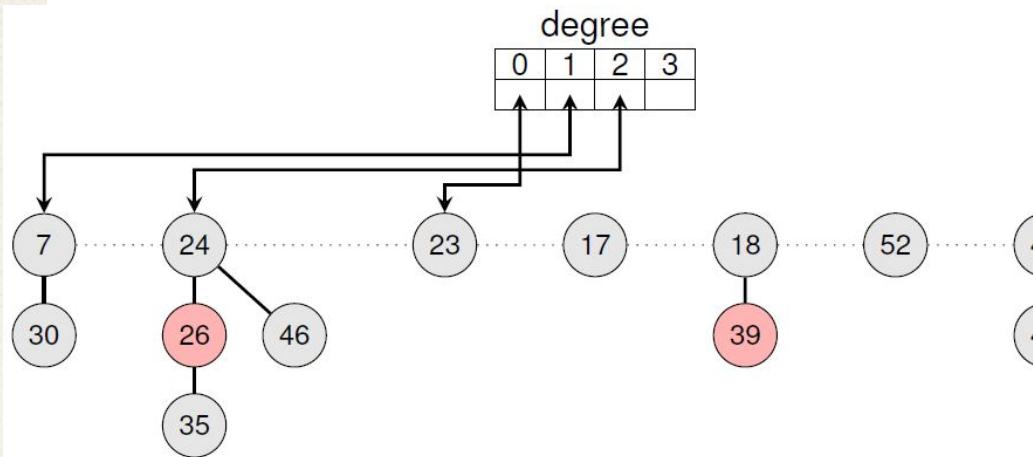
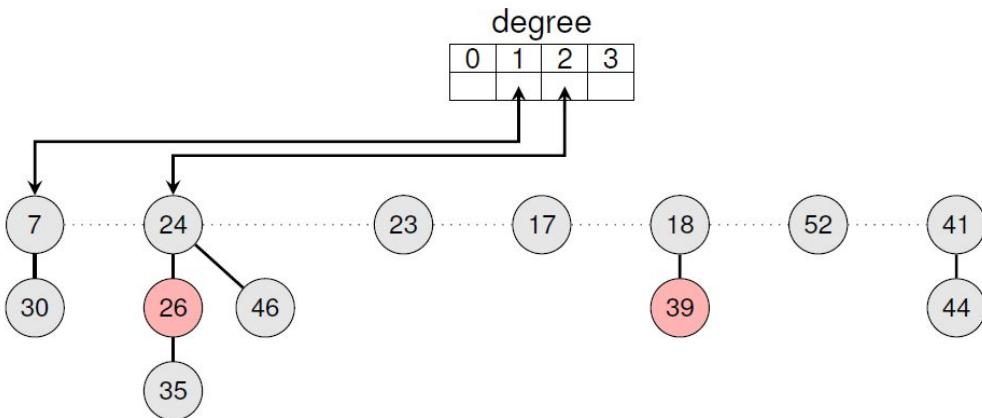
grad =



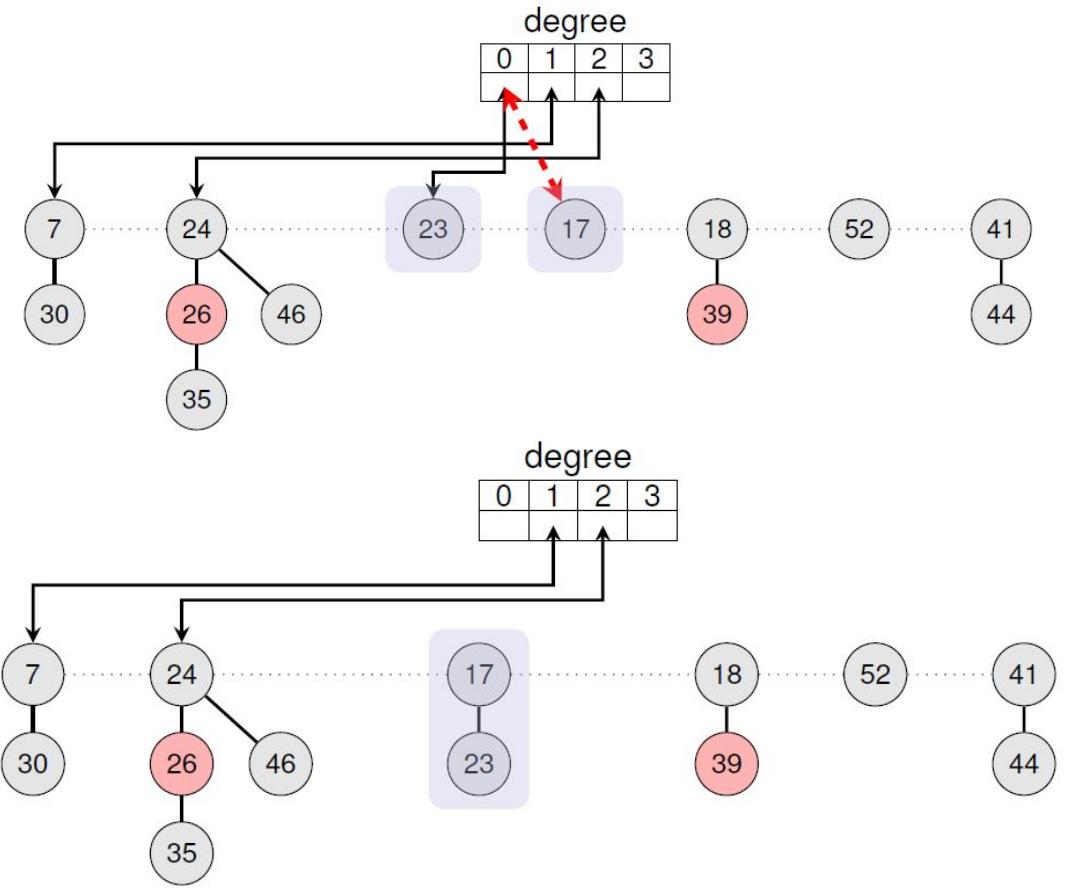
degree			
0	1	2	3



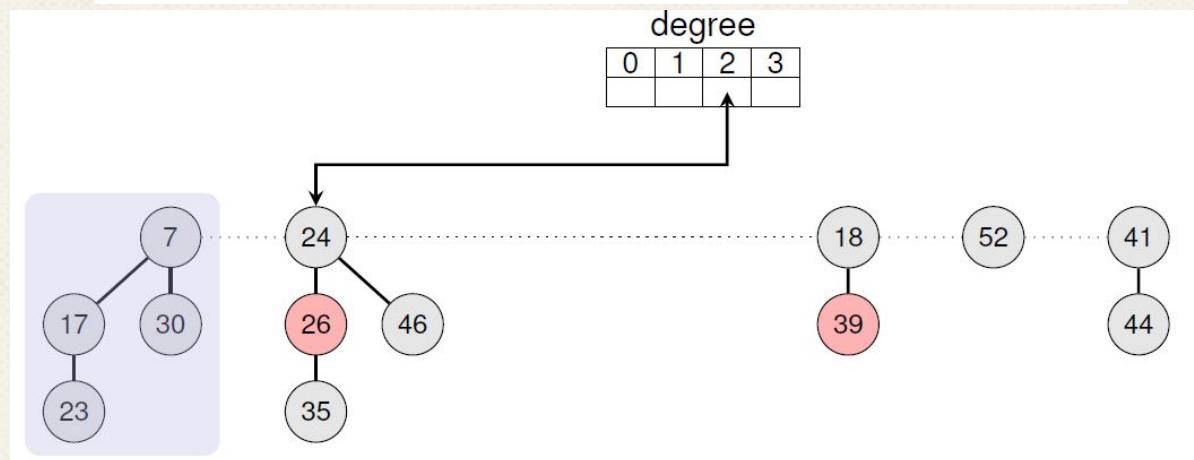
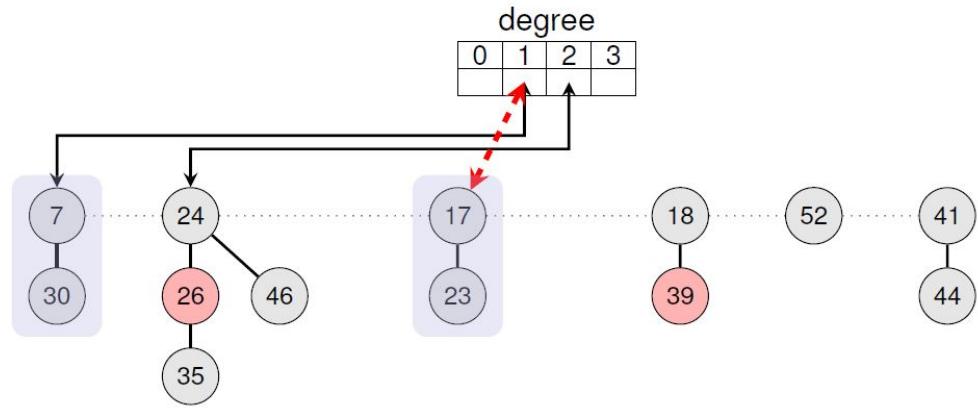
# Extragele minim



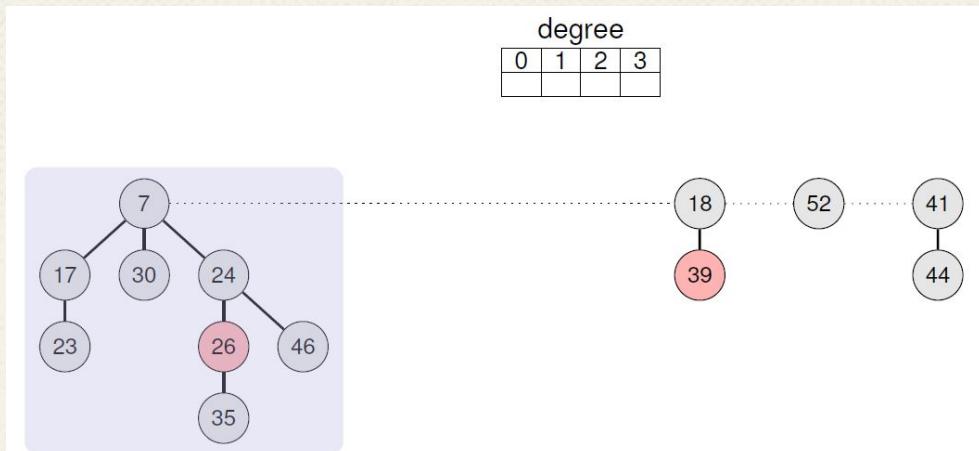
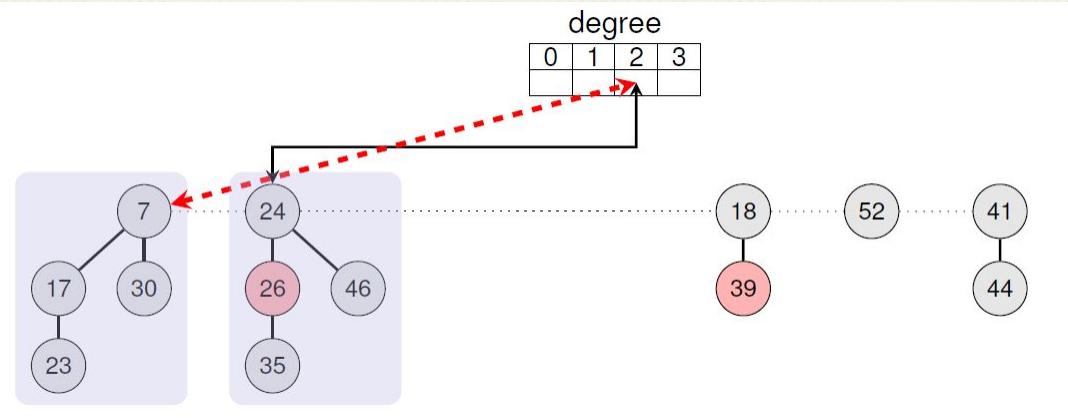
# Extragele minim



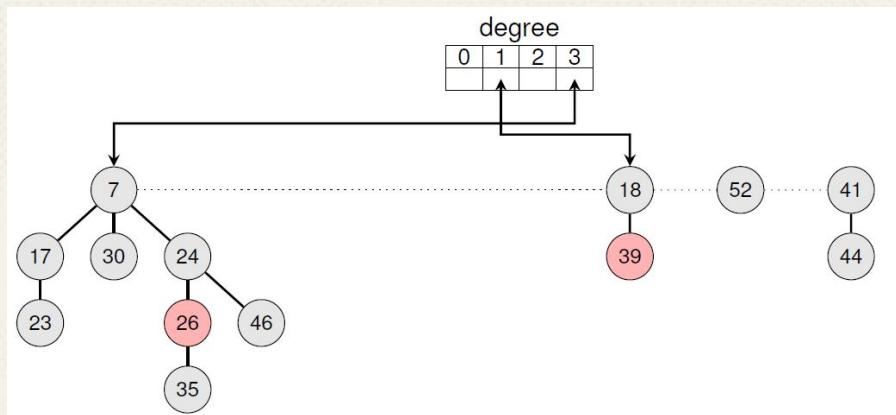
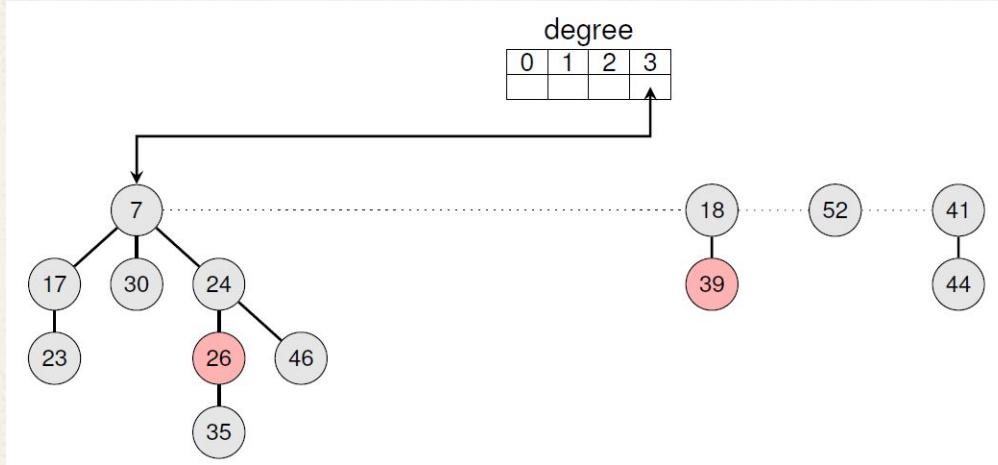
# Extragele minim



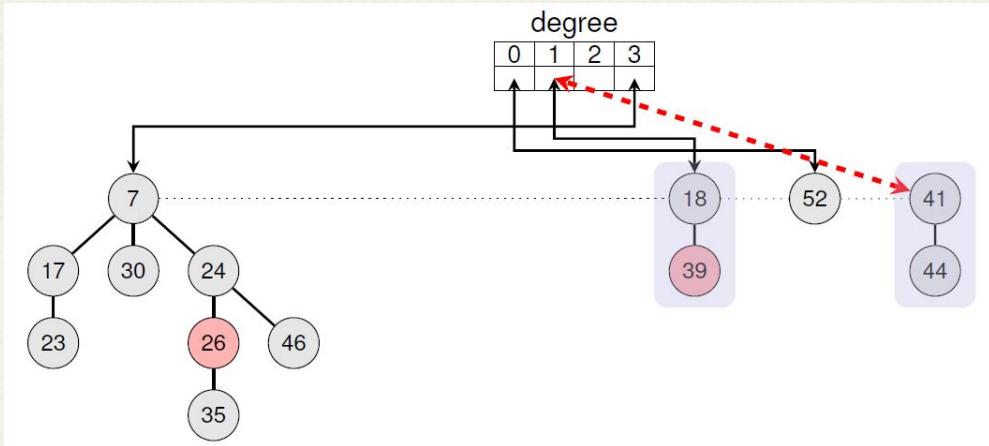
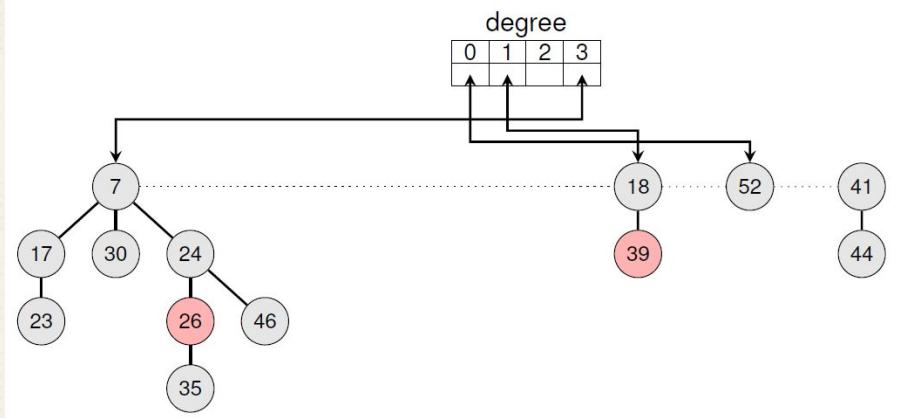
# Extragerere minim



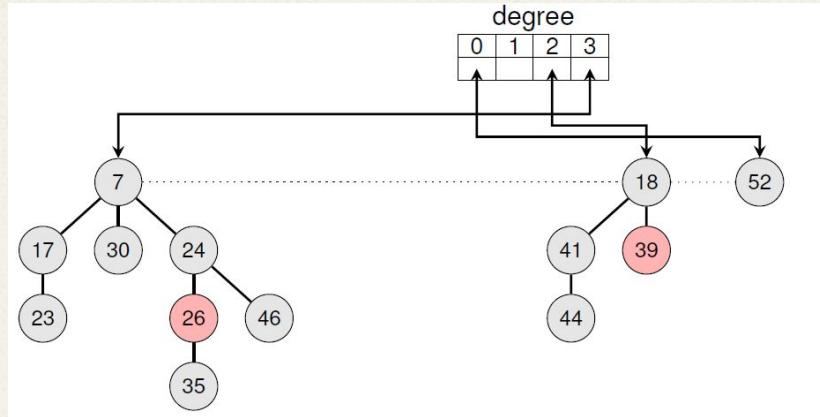
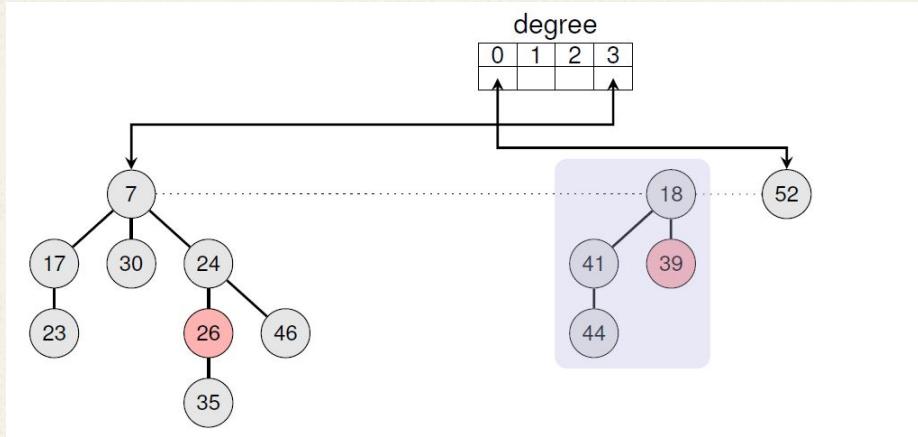
# Extragele minim



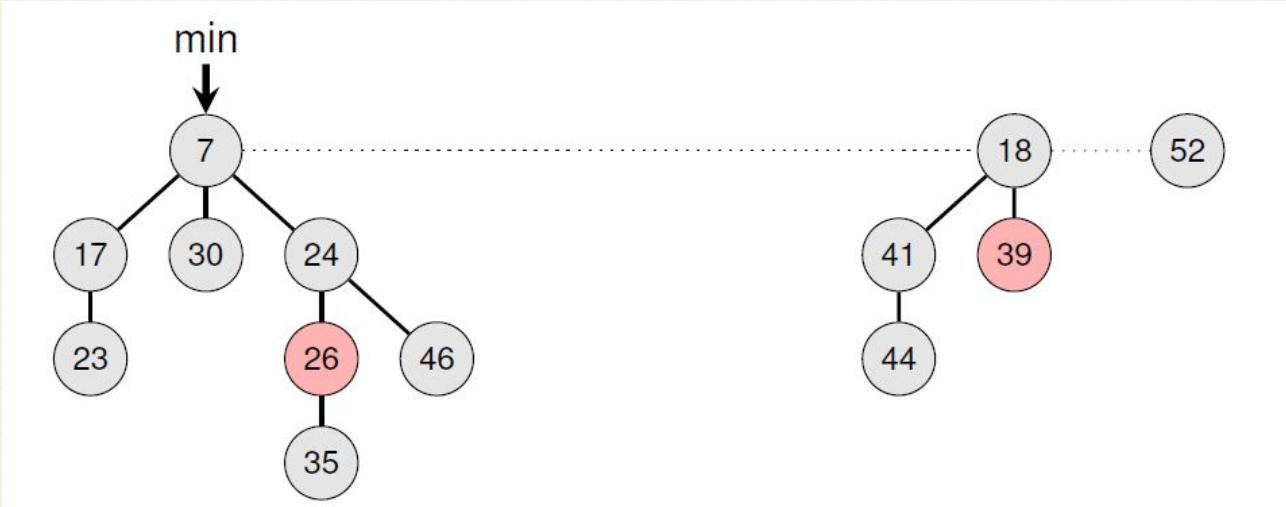
# Extragele minim



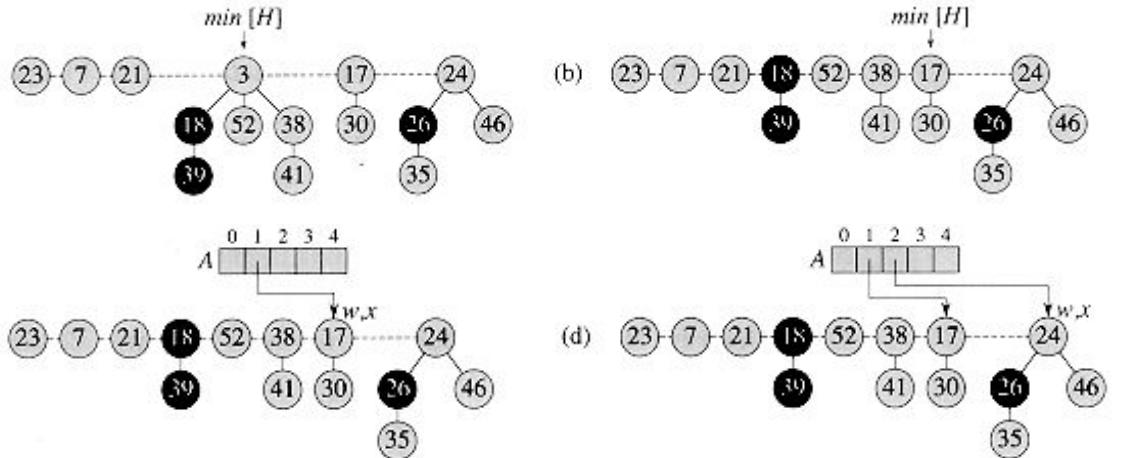
# Extragele minim



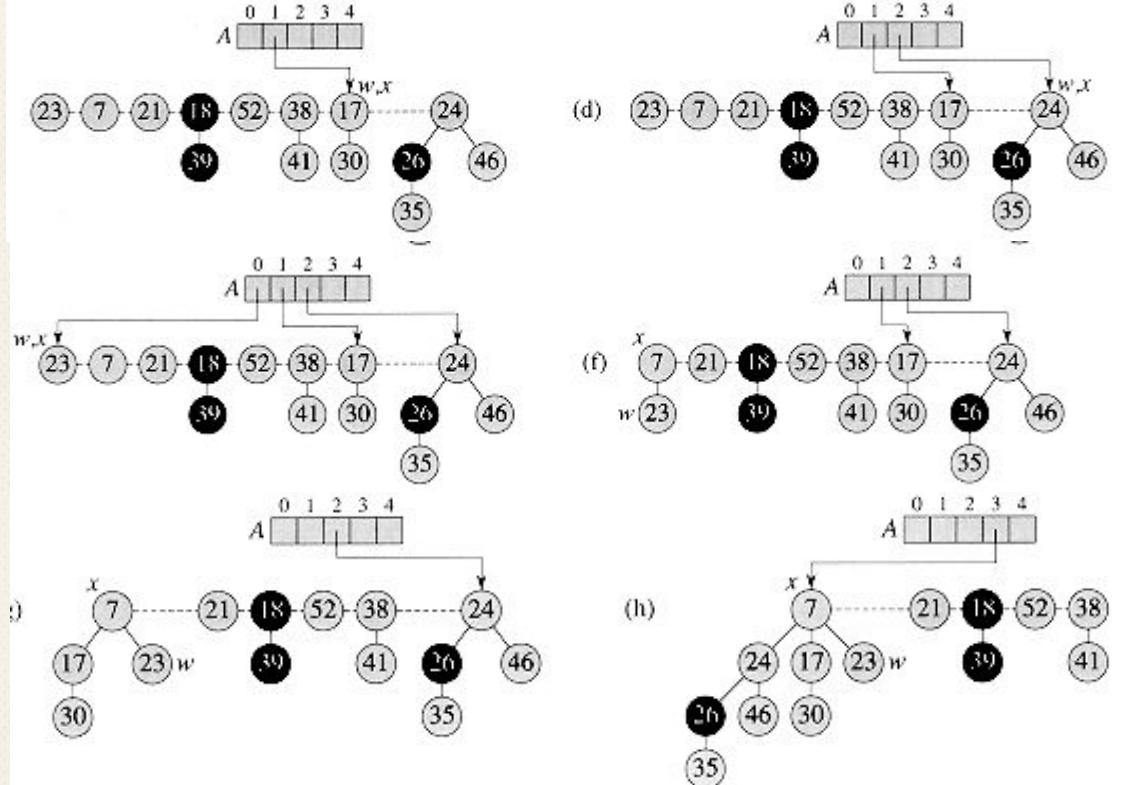
# Extragere minim



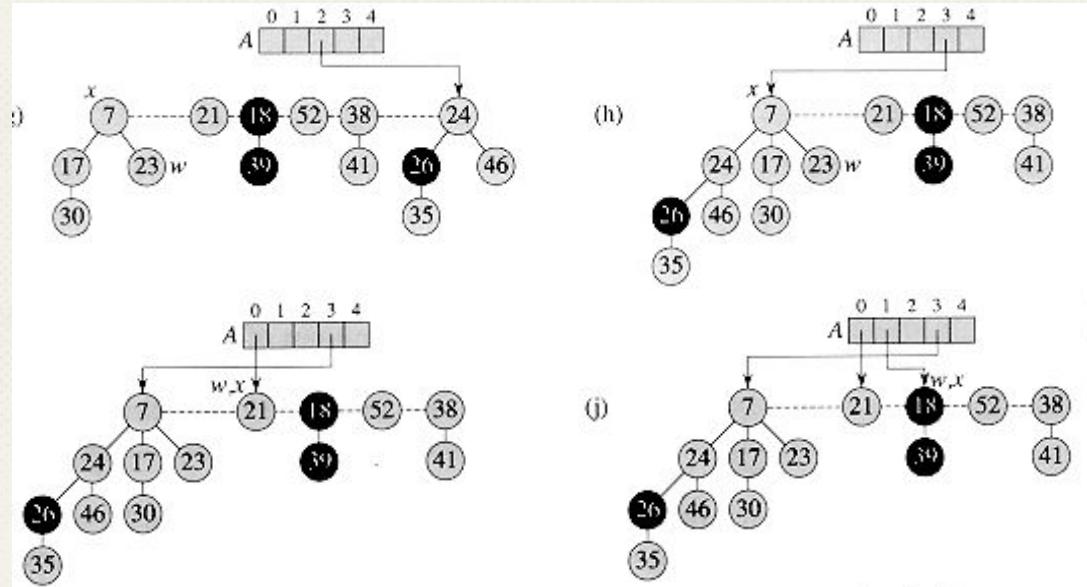
# Extrage $\min$



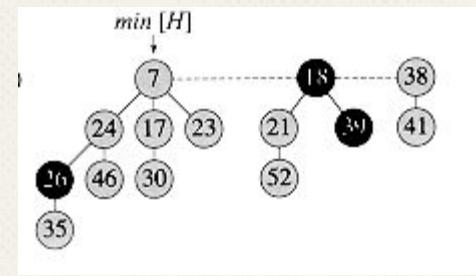
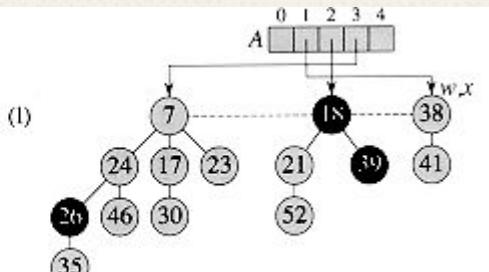
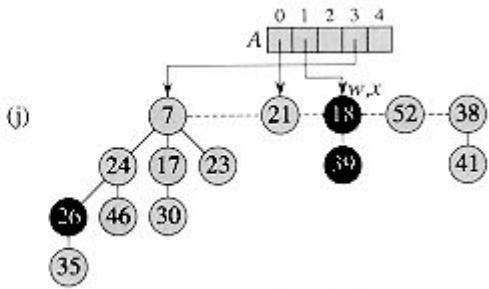
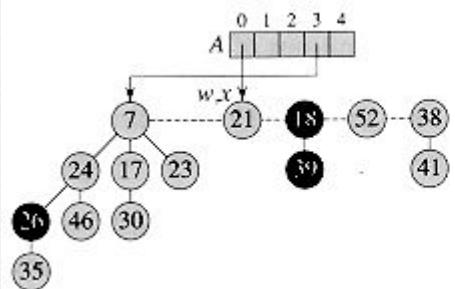
# Extragere minim



# 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 siruri.

Ex: 3 siruri 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 →  $O(n)$ .
  - 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

# Arbore binare de căutare



# Arbore Binari de Căutare

Un **arbore binar de căutare** este un arbore **binar** care satisface următoarea proprietate:

Pentru un nod  $x$ :

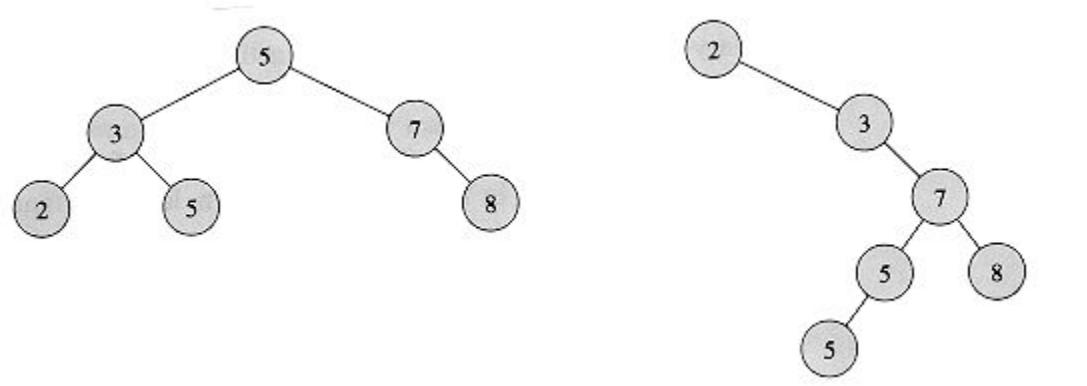
- Dacă  $y$  este un nod din subarborele stâng al lui  $x$ , atunci  $\text{cheie}[y] \leq \text{cheie}[x]$
- Dacă  $y$  este un nod din subarborele drept al lui  $x$ , atunci  $\text{cheie}[x] \leq \text{cheie}[y]$

# Arbore Binari de Căutare

Un **arbore binar de căutare** este un arbore **binar** care satisface următoarea proprietate:

Pentru un nod  $x$ :

- Dacă  $y$  este un nod din subarborele stâng al lui  $x$ , atunci  $\text{cheie}[y] \leq \text{cheie}[x]$
- Dacă  $y$  este un nod din subarborele drept al lui  $x$ , atunci  $\text{cheie}[x] \leq \text{cheie}[y]$



# Arbore Binari

Un *arbore binar strict* este un arbore binar în care fiecare nod fie nu are nici un fiu, fie are exact doi fii.

Nodurile cu doi copii se vor numi *noduri interne*, iar cele fără copii se vor numi *noduri externe* sau *frunze*.

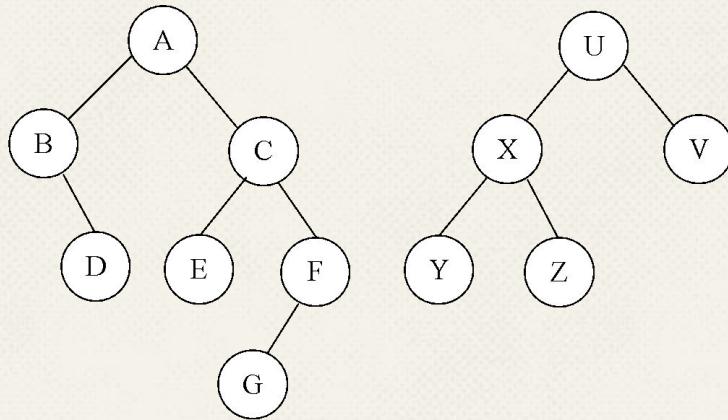
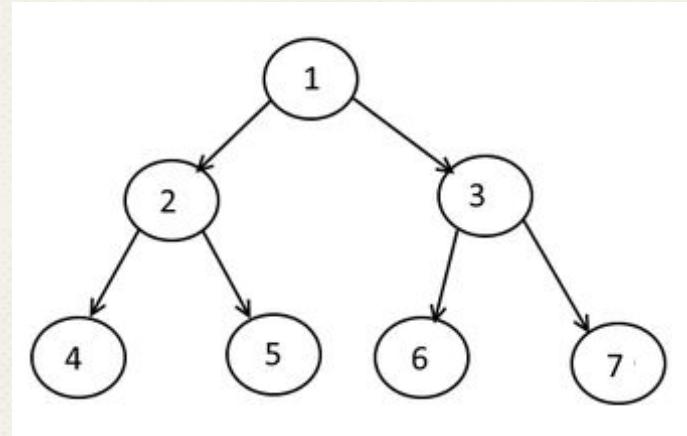


Fig.4.1.1. (a) Un arbore binar nestrict. (b) Arbore binar strict.

# Arbore Binari - Parcurgeri

Parcurgeri în arbori binari:

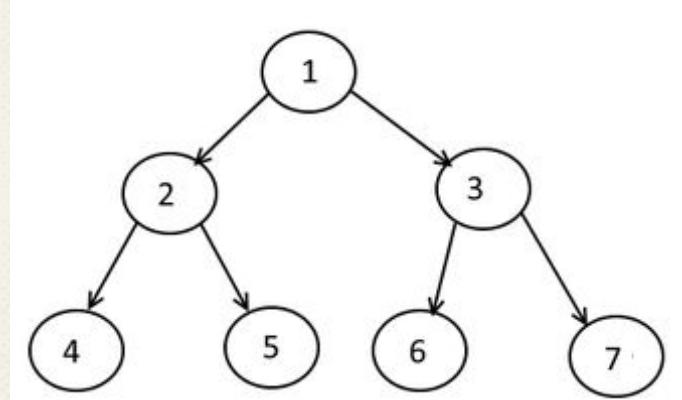
- **Inordine** (SRD, stânga rădăcină dreapta)
- **Preordine** (RSD, rădăcină stânga dreapta)
- **Postordine** (SDR, stânga dreapta rădăcină)



# Arbore Binari - Parcurgeri

Parcurgeri în arbori binari:

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



Inorder Traversal: 4 2 5 1 6 3 7  
Preorder Traversal: 1 2 4 5 3 6 7  
Postorder Traversal: 7 6 3 5 4 2 1

4 5 2 6 7 3 1

# Arbore Binari - Parcurgeri

```
void par_rsd (BTREE t) {  
    if (t != NULL) {  
        visit(t);  
        par_rsd(t->left);  
        par_rsd(t->right);  
    }  
}
```

```
void par_srd (BTREE t) {  
    if (t != NULL) {  
        par_srd(t->left);  
        visit(t);  
        par_srd(t->right);  
    }  
}
```

```
void par_sdr (BTREE t) {  
    if (t != NULL) {  
        par_sdr(t->left);  
        par_sdr(t->right);  
        visit(t);  
    }  
}
```

[Link pt vizualizare](#)

**TEMĂ:** Se dau SRD și RSD. Afişați arborele

# Arbore Binari de Căutare

- Înălțimea arborelui ?

- Minim ?

- Maxim ?

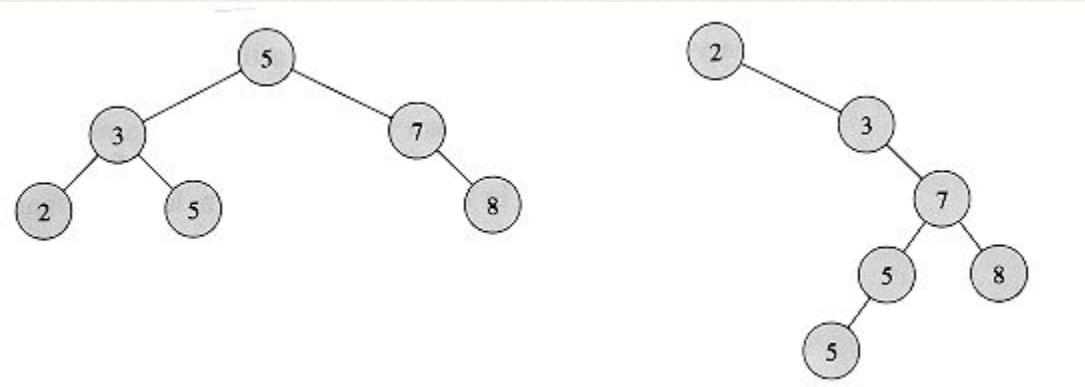
# Arbore Binari de Căutare

- Înălțimea arborelui ?
  - Minim
    - Arbore Binar Complet → Înălțime **log n**
  - Maxim
    - Dacă avem lanț (elementele sunt inserate în ordine crescătoare sau descrescătoare) → Înălțime **n**

# Arbore Binari de Căutare

○ Ce parcurgere ne oferă vectorul sortat?

- Preordine
- Inordine
- Postordine

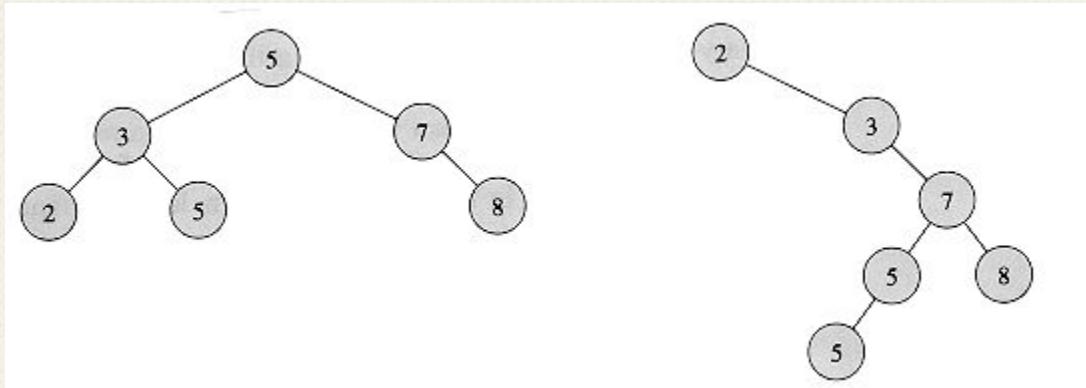


# Arbore Binare de Căutare

- Parcursarea **inordine** ne oferă vectorul sortat

- Preordine    5 3 2 5 7 8 | 2 3 7 5 5 8
- Inordine    2 3 5 5 7 8 | 2 3 5 5 7 8
- Postordine    2 5 3 8 7 5 | 5 5 8 7 3 2

- Restul parcurgerilor sunt diferite pentru cei 2 arbori.

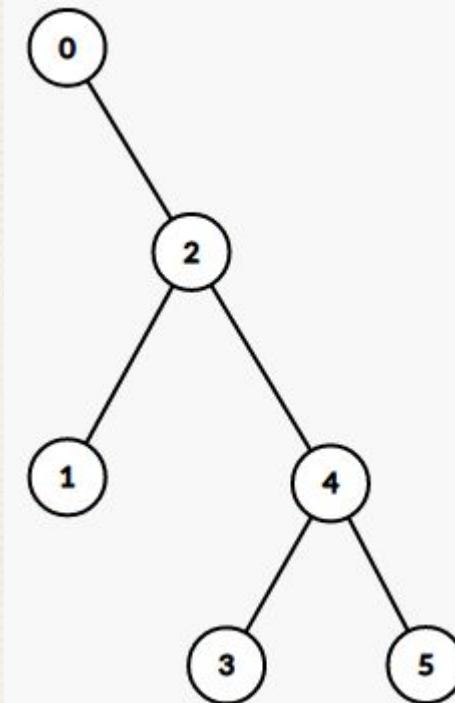
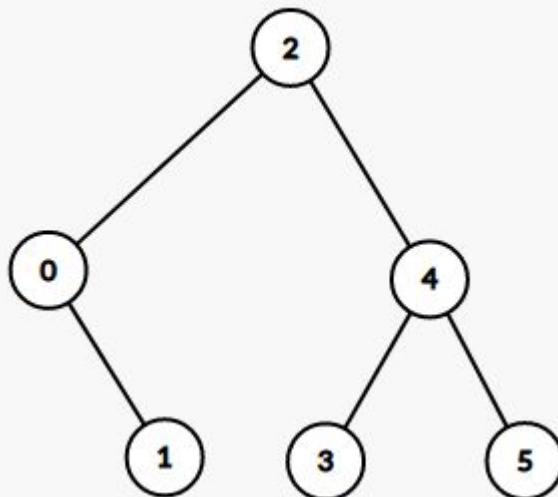


# Exercițiu

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

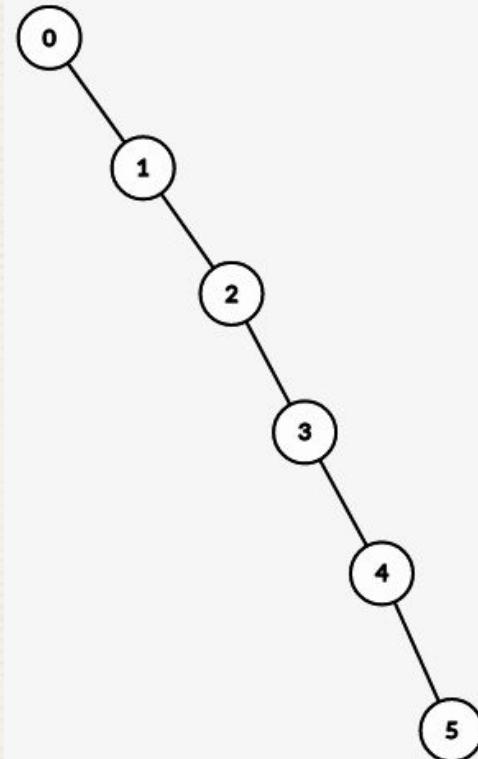
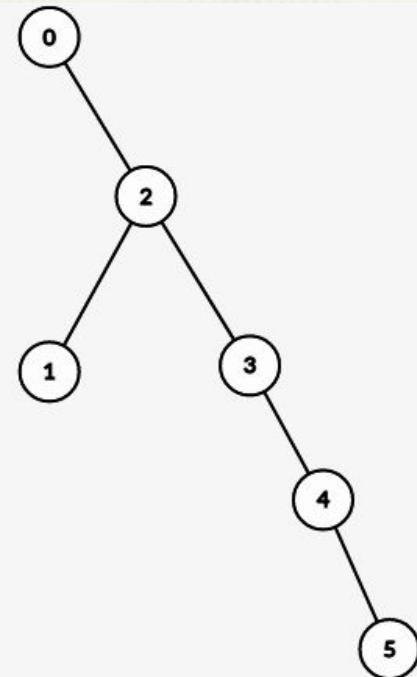
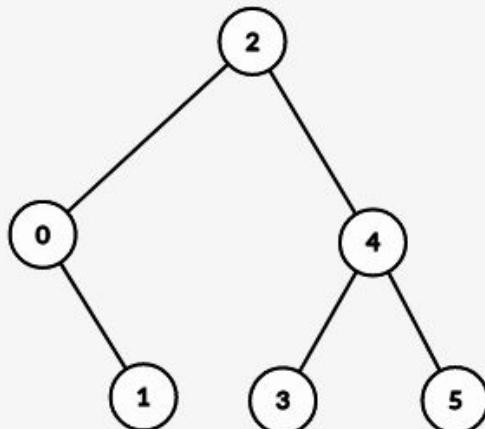
# Exercițiu

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



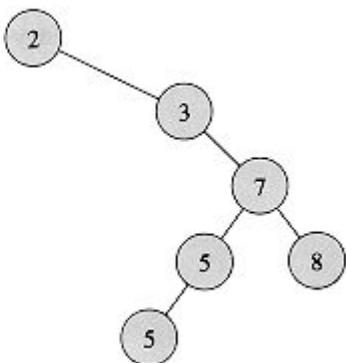
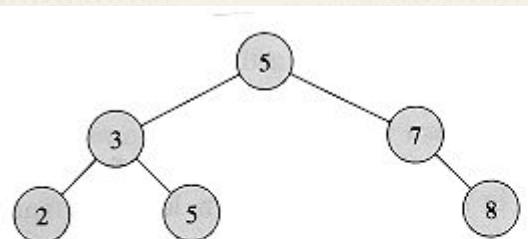
# Exercițiu

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



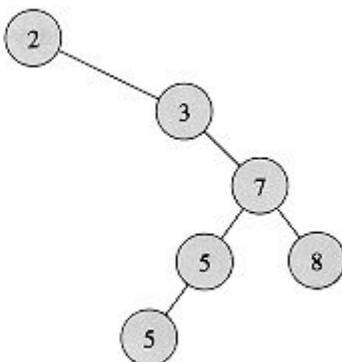
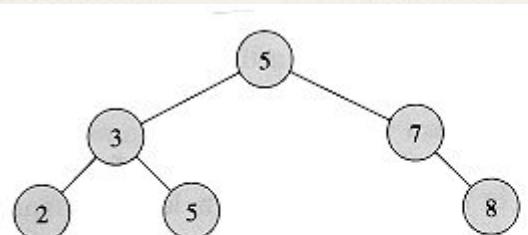
# Minim și Maxim

- Unde se află minimul?



# Minim și Maxim

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

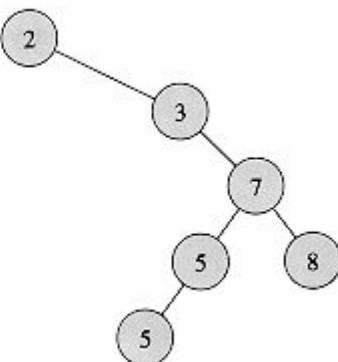
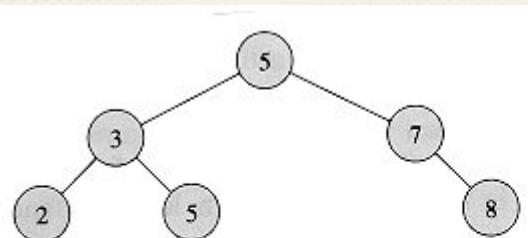


TREE-MINIMUM ( $x$ )

```
1 while left[ $x$ ] ≠ NIL  
2     do  $x \leftarrow \text{left}[x]$   
3 return  $x$ 
```

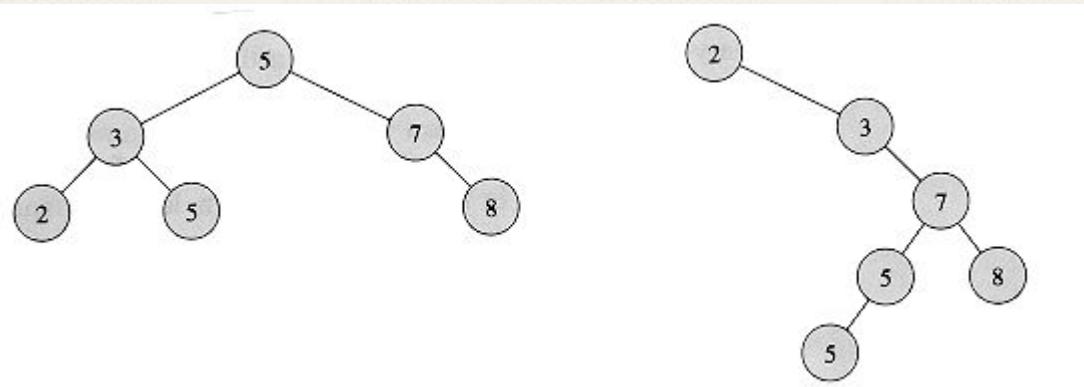
# Minim și Maxim

- Unde se află maximul?



# Minim și Maxim

- Unde se află maximul?
  - În cel mai din dreapta nod

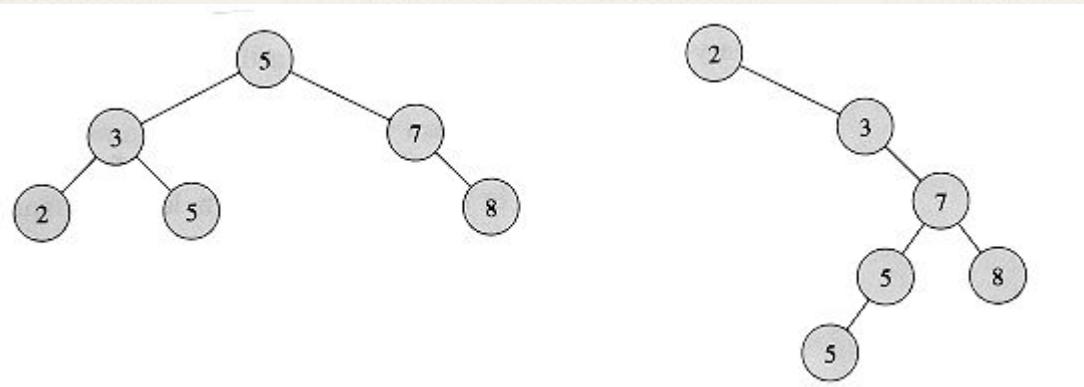


```
TREE-MAXIMUM (x)
1 while right[x] ≠ NIL
2   do x ← right[x]
3 return x
```

Complexitate?

# Minim și Maxim

- Unde se află maximul?
  - În cel mai din dreapta nod



Complexitate?  $O(h)$

```
TREE-MAXIMUM (x)
1 while right[x] ≠ NIL
2     do x ← right[x]
3 return x
```

# Căutare

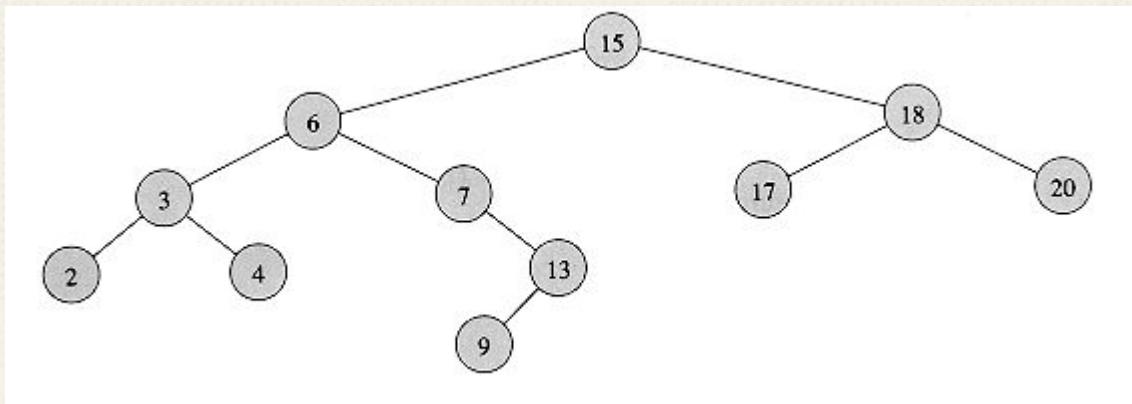
Minimul și maximul se găsesc mai greu decât într-un heap. Avantajul major al arborilor binari de căutare este că permit o căutare “relativ” eficientă.

Cum găsim un element?

# Căutare

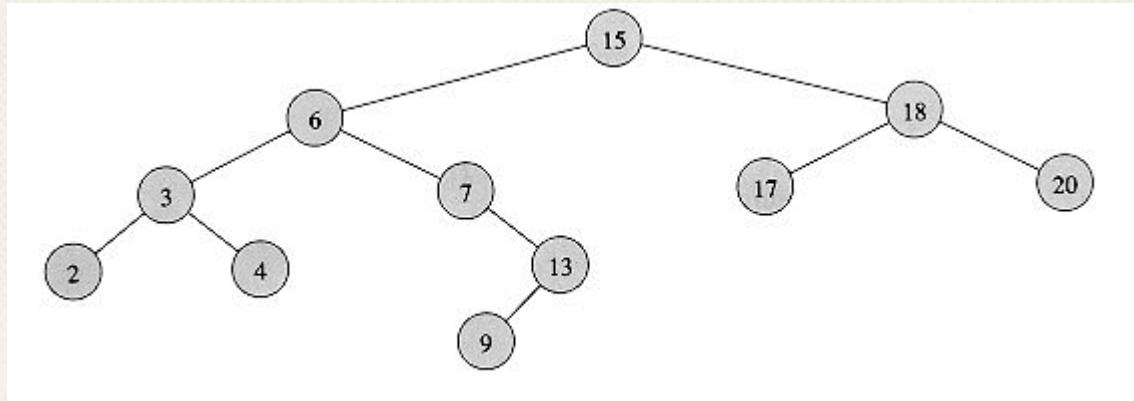
Minimul și maximul se găsesc mai greu decât într-un heap. Avantajul major al arborilor binari de căutare este că permit o căutare “relativ” eficientă.

Cum găsim un element?



# Căutare

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



# Căutare

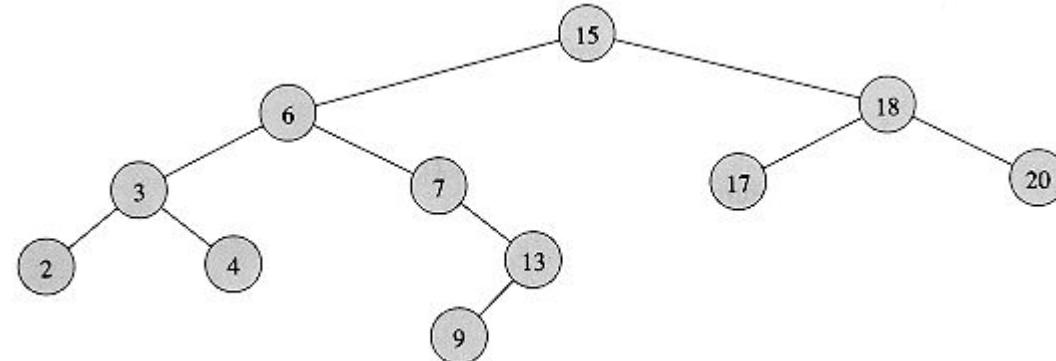
```
ITERATIVE-TREE-SEARCH (x, k)
```

```
1 while x ≠ NIL and k ≠ key[x]
2   do if k < key[x]
3     then x ← left[x]
4   else x ← right[x]
5 return x
```

```
TREE-SEARCH (x, k)
```

```
1 if x = NIL or k = key[x]
2   then return x
3 if k < key[x]
4   then return TREE-SEARCH (left[x], k)
5 else return TREE-SEARCH (right[x], k)
```

Complexitate:  $O(h)$



# Succesor / Predecesor

Până acum, puteam să ținem un dicționar și un heap și să facem aceleași operații.

**Succesor:** Se dă un nod din arbore.

Care este cea mai **mică** valoare din arbore  $\geq \text{val}[x]$  (valorea nodului)?

**Predecesor:** Se dă un nod din arbore.

Care este cea mai **mare** valoare din arbore  $\leq \text{val}[x]$  (valorea nodului)?

Cum facem?

# Sucesor / Predecesor

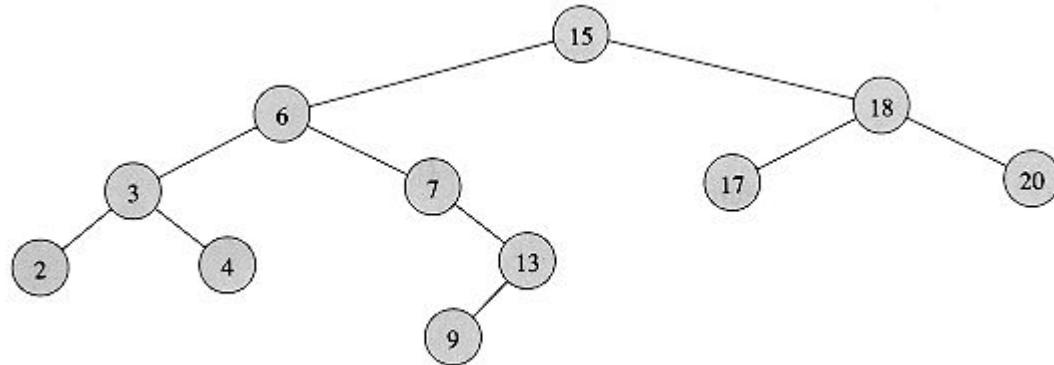
Sucesor de 3?

Sucesor de 6?

Sucesor de 15?

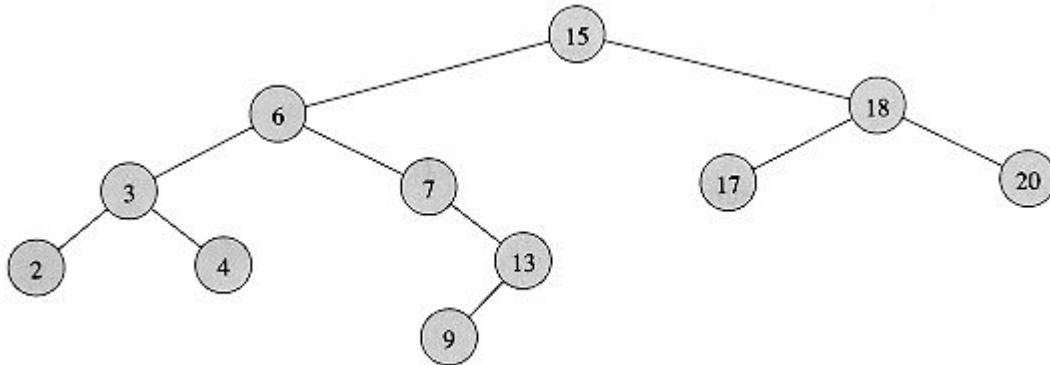
Sucesor de 13?

Sucesor de 4?



# Sucesor / Predecesor

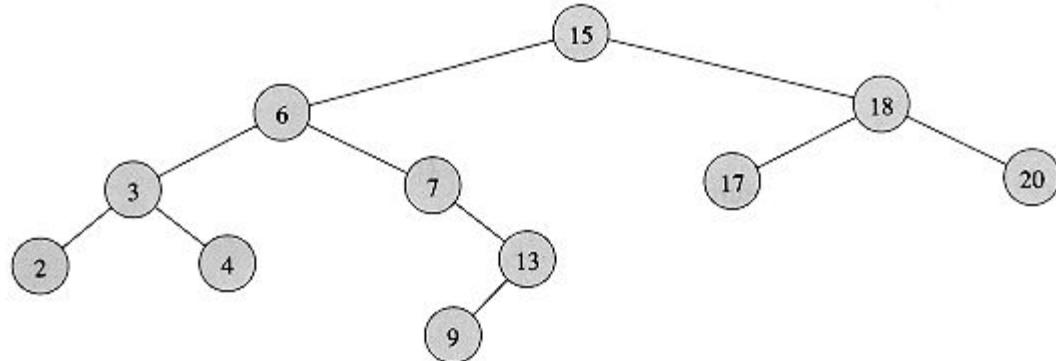
- Sucesor de 3? → 4
- Sucesor de 6? → 7
- Sucesor de 15? → 17
- Sucesor de 13? → 15
- Sucesor de 4? → 6



# Succesor / Predecesor

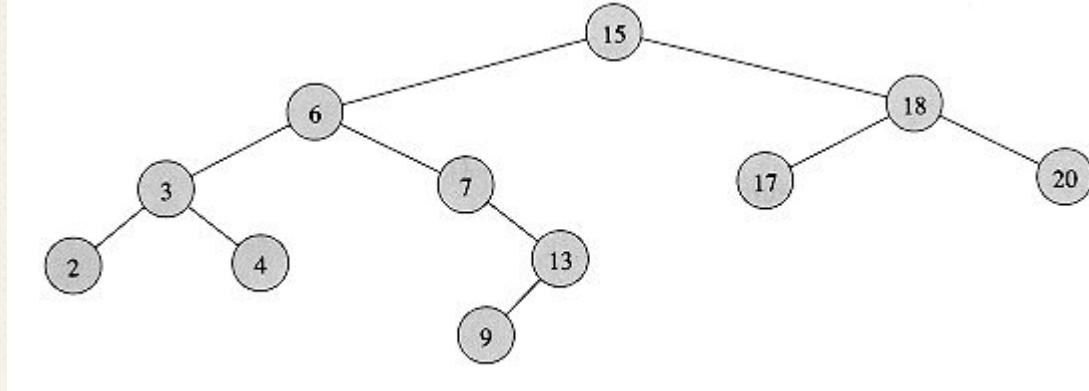
**Caz 1)** Dacă am fiu drept, atunci cel mai mic element va fi cel mai mic element din subarborele drept. Adică dreapta→stânga→stânga→...→stânga (vezi 7 sau 15)

**Caz 2)** Dacă **nu** am fiu drept, atunci va fi primul strămoș al meu în care eu sunt în subarborele stâng al său (vezi 13, 4, 17)



# Sucesor / Predecesor

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2      then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5      do x ← y
6          y ← p[y]
7  return y
```

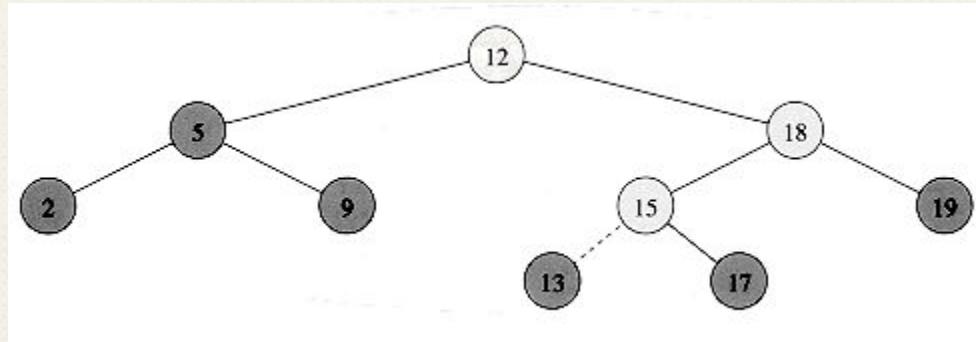


Complexitate: **O(h)**

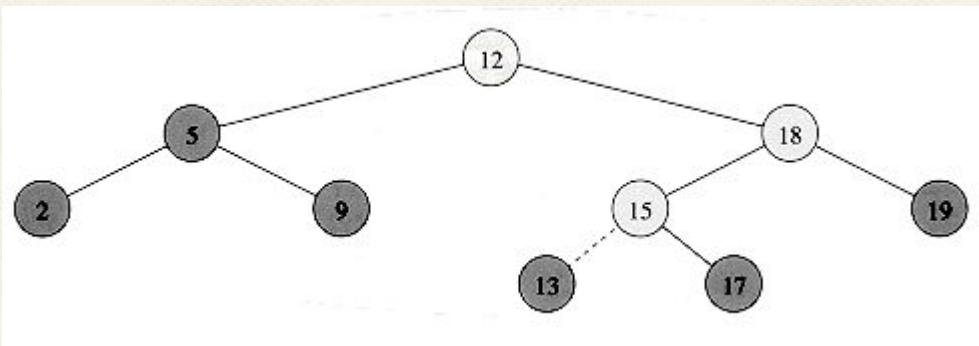
# Inserare

Similar cu căutarea.

Inserare 13:



# Inserare



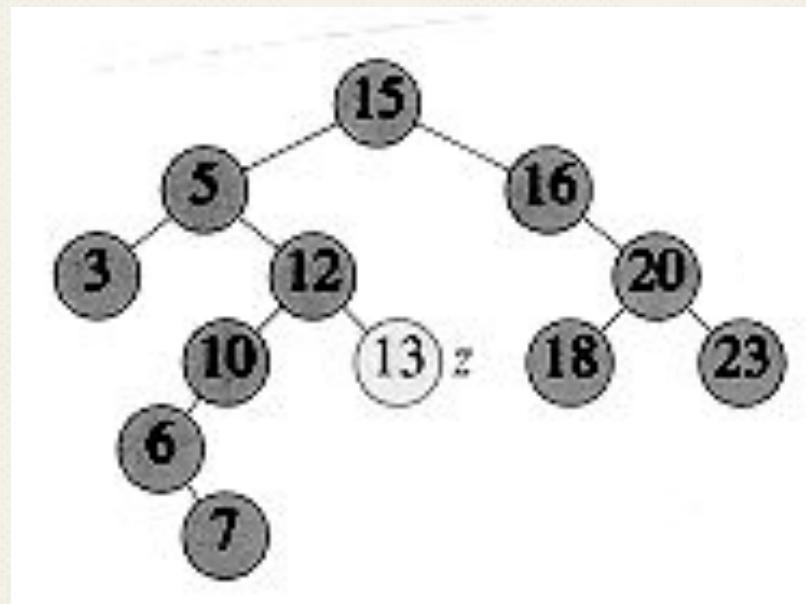
Complexitate: **O(h)**

TREE-INSERT( $T, z$ )

```
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{root}[T]$ 
3  while  $x \neq \text{NIL}$ 
4      do  $y \leftarrow x$ 
5          if  $\text{key}[z] < \text{key}[x]$ 
6              then  $x \leftarrow \text{left}[x]$ 
7              else  $x \leftarrow \text{right}[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow z$ 
11     else if  $\text{key}[z] < \text{key}[y]$ 
12         then  $\text{left}[y] \leftarrow z$ 
13         else  $\text{right}[y] \leftarrow z$ 
```

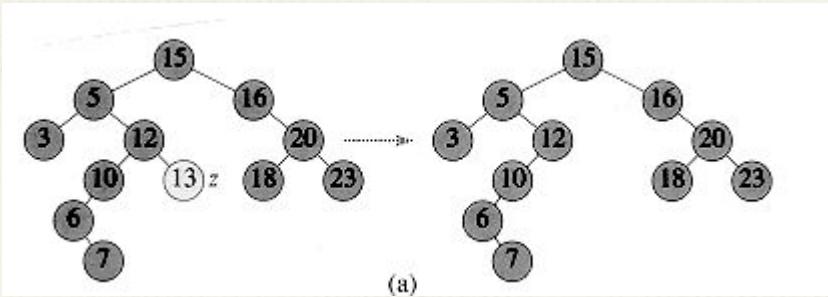
# Ştergere

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

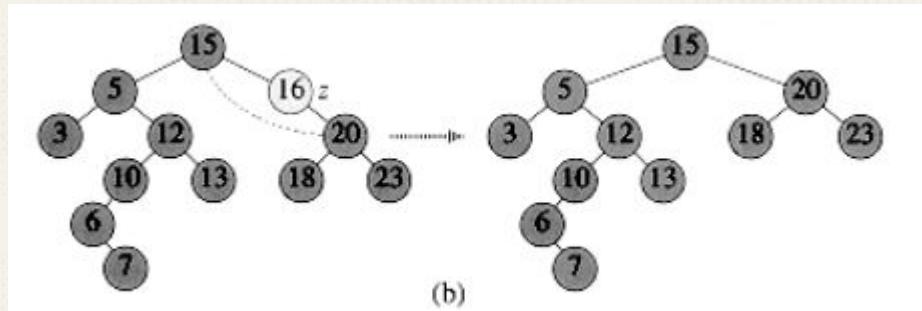


# Ştergere

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



(a)



(b)

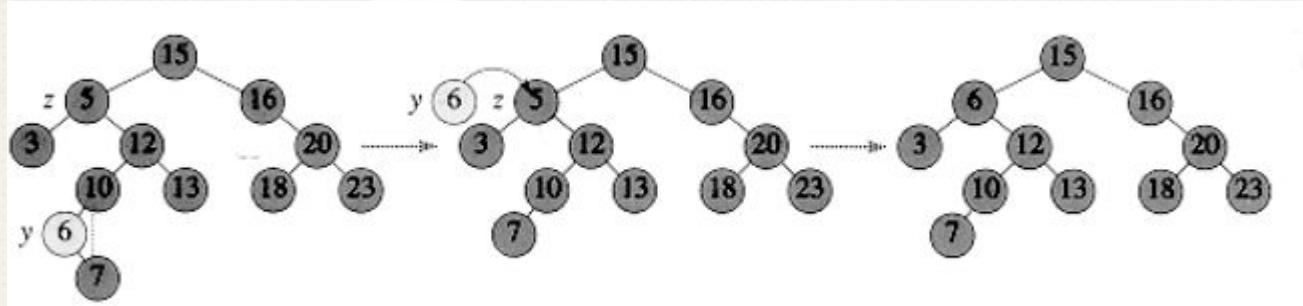
# Ştergere

*Exerciţiu:*

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

# Ştergere

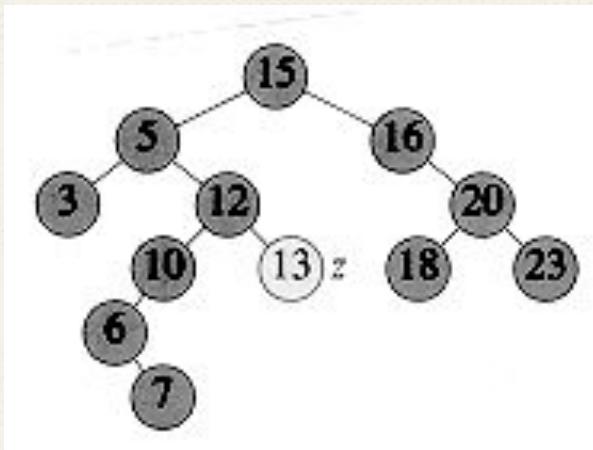
- Cum?
- Cum îl ştergem pe 13?
- Dar pe 7?
- Pe 5?
- Dar pe 15?



# Ştergere

Avem 3 cazuri:

- 1) Dacă nodul **nu are** fi, îl ștergem.
- 2) Dacă are **un** fiu, îl ștergem și creăm o tată și noul fiu.

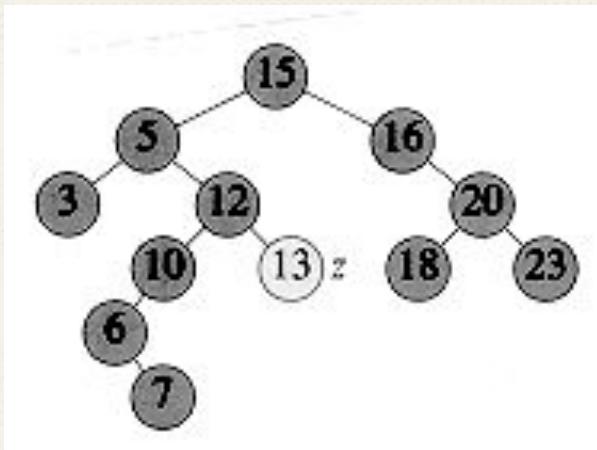


```
TREE-DELETE( $T, z$ )  
1 if  $left[z] = NIL$  or  $right[z] = NIL$   
2     then  $y \leftarrow z$   
3     else  $y \leftarrow TREE-SUCCESSOR(z)$   
4     if  $left[y] \neq NIL$   
5         then  $x \leftarrow left[y]$   
6         else  $x \leftarrow right[y]$   
7     if  $x \neq NIL$   
8 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

# Ştergere

Avem 3 cazuri:

- 3) Dacă are **ambii** fii, găsim succesorul său, punem în locul său și înlocuim legătura acestui nod cu singurul fiu (dacă există)



TREE-DELETE( $T, z$ )

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

# Complexitate

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

# Complexitate

Operăție	Complexitate
Căutare	$O(h)$
Găsire Minim	$O(h)$
Inserare	$O(h)$
Succesor / Predecesor	$O(h)$
Ștergere	$O(h)$

# **Arbore Binare de Căutare cu Chei Egale**

Ce facem dacă avem mai multe chei egale ?

# Arbore Binare de Căutare cu Chei Egale

Ce facem dacă avem mai multe chei egale ?

- În caz de egalitate, alegem tot timpul stânga sau dreapta și inserăm în aceeași direcție
- Ținem o listă cu toate elementele egale într-un singur nod (sau un contor care să numere aparițiile, dacă nu avem alte informații)

# Arbore Binari Echilibrați

- AVL
- Arboare Roșu-Negru
- Treap-uri
- Splay Trees
- B-arbori
  
- Skip Lists (nu sunt arbori binari de căutare, dar...)

Search trees  
(dynamic sets/associative arrays)

2-3 · 2-3-4 · AA · (a,b) · AVL · B · B+ · B\* · B<sup>X</sup> · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic ·  
(Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB · Weight-balanced

# Bibliografie

Introducere în Algoritmi Cormen Leiserson Rivest

# ARBORI BINARI ECHILIBRAȚI

,

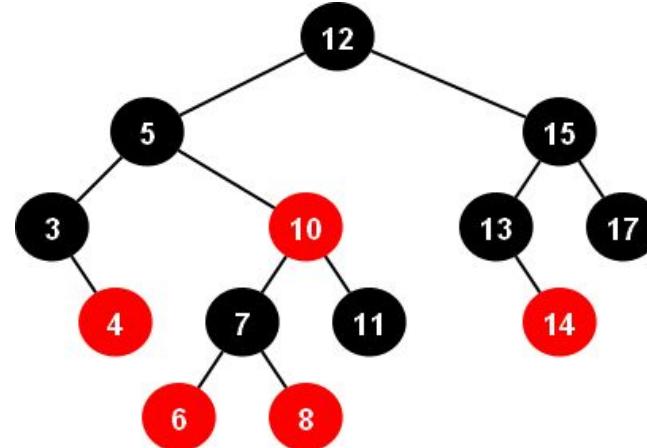


# Arbore Binari de Căutare Construiți Aleator

**Teorema 13.6.** Înălțimea medie a unui arbore binar de căutare construit aleator cu  $n$  chei distințe este  $O(\lg n)$ .

# Red Black Trees

- Reguli:
  - Fiecare nod e fie roșu, fie negru
  - Rădăcina e mereu neagră
  - Nu putem avea două noduri adiacente roșii
  - Orice drum de la un nod la un descendant NULL are același număr de noduri negre



# Red Black Trees

- Red Black Trees (nu veți avea la examen)
  - [MIT Video](#)
  - [MIT Lecture Notes](#)

# Red Black Trees

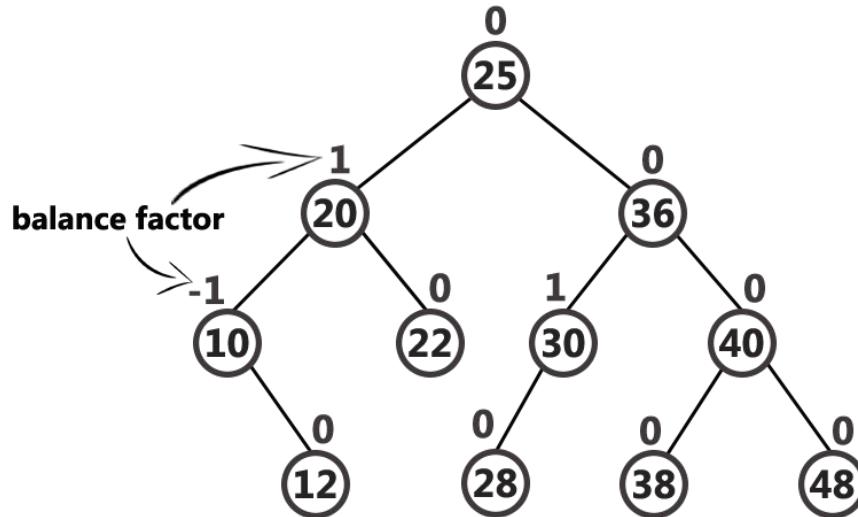


# ~~Red Black Trees~~ AVL



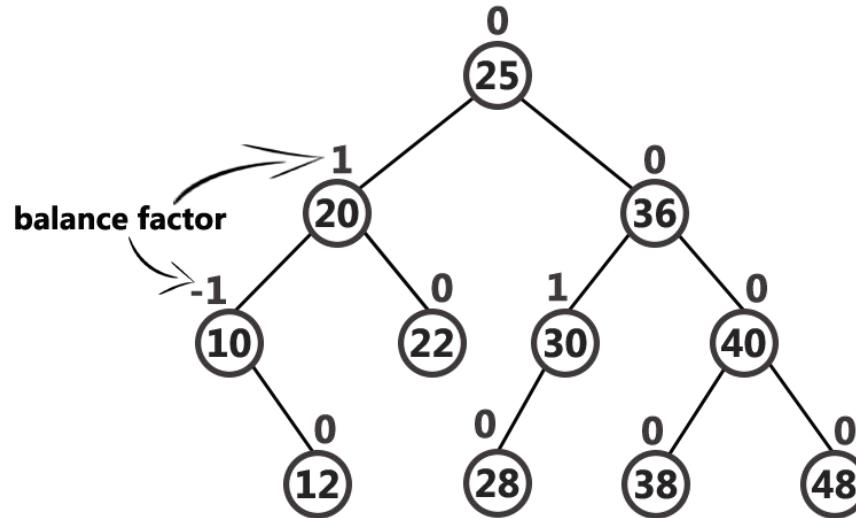
# AVL

- Construcția AVL-urilor:
  - pentru fiecare nod, diferența dintre înălțimile fiilor drept și stâng trebuie să fie maxim 1



# AVL

- Factorul de echilibru al unui nod:
  - $BF(X) = h(\text{subarbore_drept}(X)) - h(\text{subarbore_stang}(X))$



# AVL - Reechilibrare

- Rotații:

- 1) **Rotație stânga-stânga**
  - când un nod este inserat în stânga subarborelui stâng
  - se realizează o rotație la dreapta
- 2) **Rotație dreapta-dreapta**
  - când un nod este inserat în dreapta subarborelui drept
  - se realizează o rotație la stânga
- 3) **Rotație dreapta-stânga**
  - când un nod este inserat la dreapta subarborelui stâng
  - se realizează două rotații
- 4) **Rotație stânga-dreapta**
  - când un nod este inserat la stânga subarborelui drept
  - se realizează două rotații

Mai multe informații: <https://www.guru99.com/avl-tree.html>

# AVL

AVL (veți avea la examen)

- [Video](#) (MIT).
- [Lecture Notes](#)

# SKIP LISTS

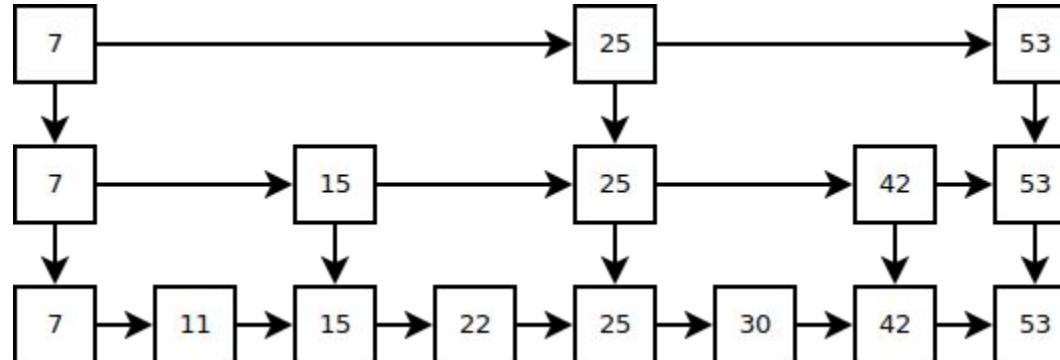


# Skip Lists

- Sunt structuri de date echilibrate
- Alte structuri de date eficiente (**log n** sau mai bun):
  - Tabele de dispersie (hash tables) - nu sunt sortate
  - Heap-uri - nu putem căuta în ei
  - Arbori binari echilibrați (AVL, Red Black Trees)
- Ajută la o căutare rapidă
- Elementele sunt sortate!

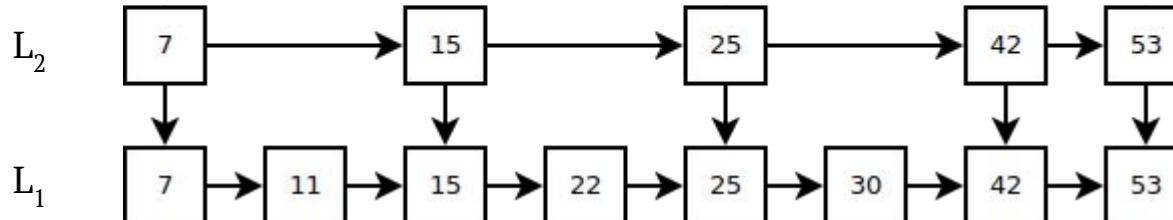
# Skip Lists

- Sunt implementate ca liste înlățuite
- Ideea de implementare:
  - este extinsă pe mai multe nivele (mai multe liste înlățuite)
  - la fiecare nivel adăugat, **sărim peste o serie de elemente** față de nivelul anterior
  - nivelele au legături între ele



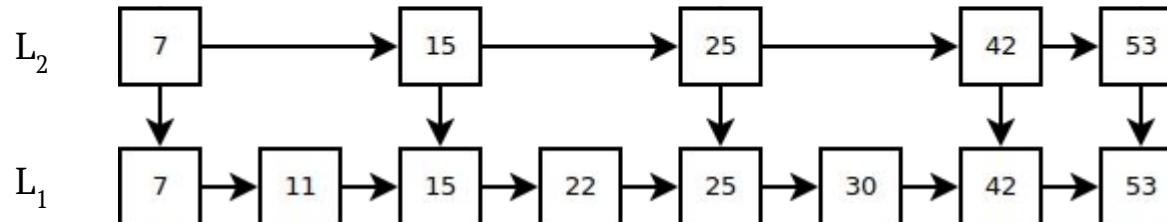
# Skip Lists

- Să presupunem că avem doar 2 liste
  - Cum putem alege ce elemente ar trebui transferate în nivelul următor?



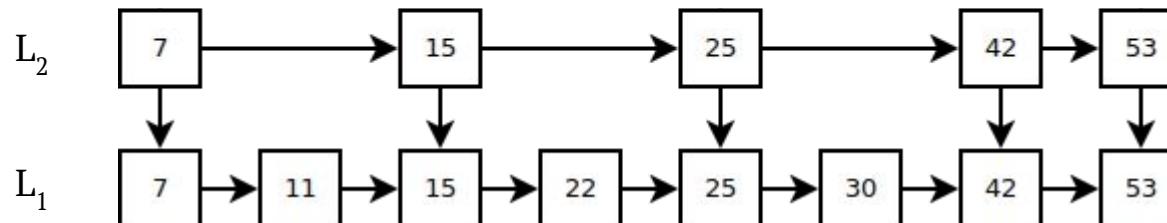
# Skip Lists - 2 liste

- Cum putem alege ce elemente ar trebui transferate în nivelul următor?
  - Cea mai bună metodă: elemente egal depărtate
  - Costul căutării =  $|L_2| + (|L_1| / |L_2|) = |L_2| + (n / |L_2|)$
  - Când este minim acest cost?



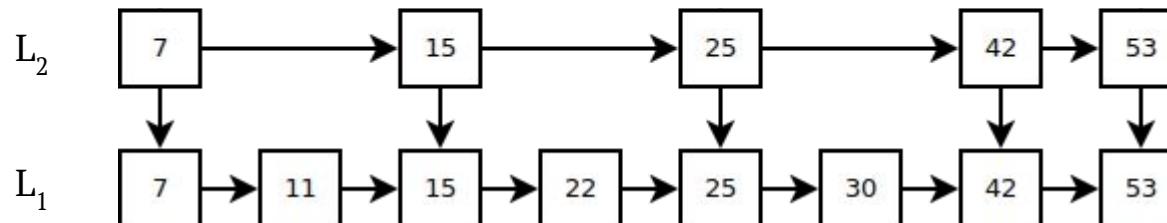
# Skip Lists - 2 liste

- Cum putem alege ce elemente ar trebui transferate în nivelul următor?
  - Cea mai bună metodă: elemente egal depărtate
  - Costul căutării =  $|L_2| + (|L_1| / |L_2|) = |L_2| + (n / |L_2|)$
  - Când este minim acest cost?
    - Când  $|L_2| = n / |L_2| \Rightarrow |L_2| = \sqrt{n}$



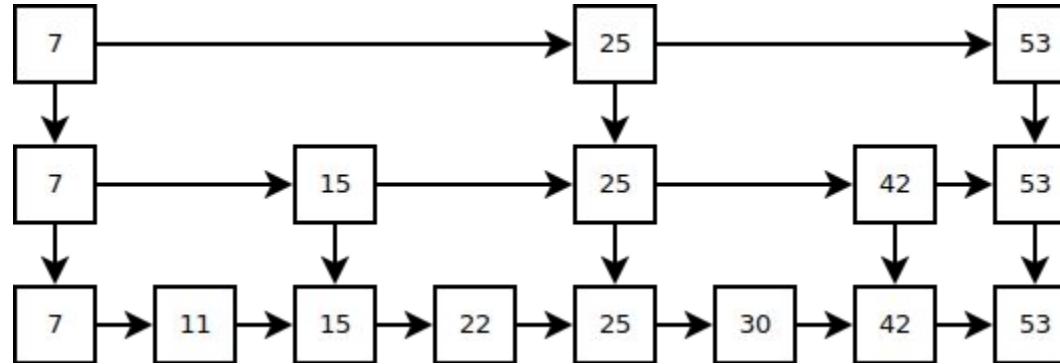
# Skip Lists - 2 liste

- Cum putem alege ce elemente ar trebui transferate în nivelul următor?
  - Cea mai bună metodă: elemente egal depărtate
  - Costul căutării =  $|L_2| + (|L_1| / |L_2|) = |L_2| + (n / |L_2|)$
  - Când este minim acest cost?
    - Când  $|L_2| = n / |L_2| \Rightarrow |L_2| = \sqrt{n}$
  - Deci, costul minim pentru căutare este  $\sqrt{n} + n / \sqrt{n} = 2 * \sqrt{n}$
  - Complexitate:  $O(\sqrt{n}) \rightarrow$  seamănă un pic cu **Batog**



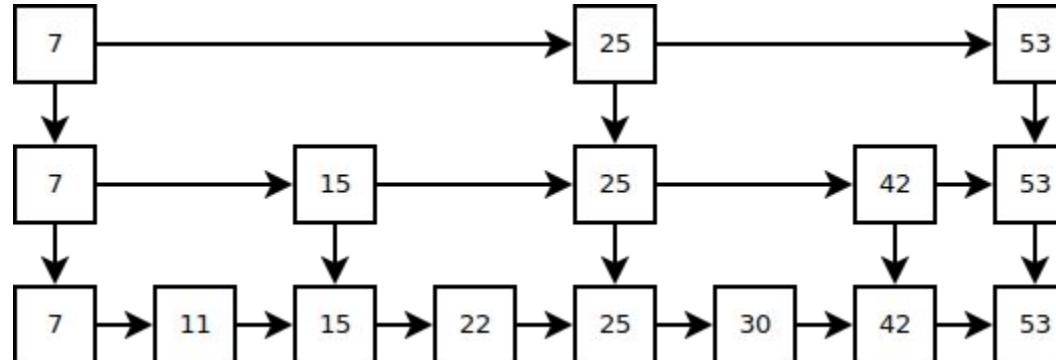
# Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlántuite?



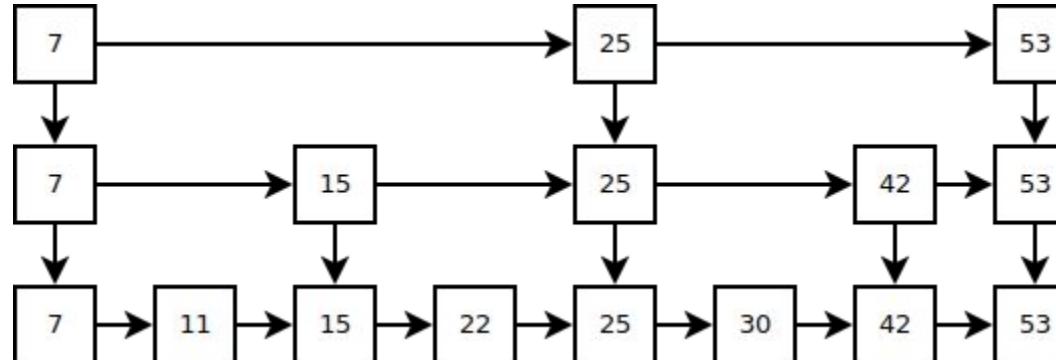
# Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlántuite?
  - Costul căutării  $2 * \sqrt{n}$  se modifică
  - 2 liste:
  - 3 liste: ?



# Skip Lists

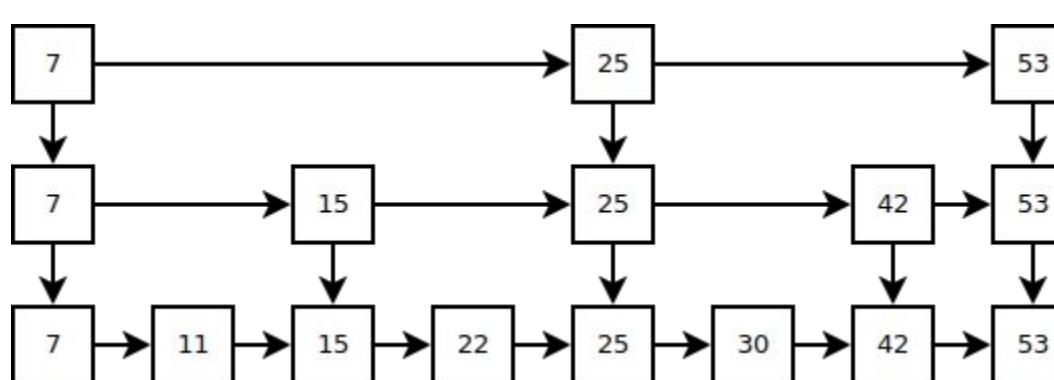
- Ce se întâmplă când avem mai mult de 2 liste înlántuite?
  - Costul căutării se modifică
  - 2 liste:  $2 * \sqrt{n}$
  - 3 liste:  $3 * \sqrt[3]{n}$
  - k liste:  $k * \sqrt[k]{n}$



# Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlántuite?

- Costul căutării se modifică
- 2 liste:  $2 * \sqrt{n}$
- 3 liste:  $3 * \sqrt[3]{n}$
- k liste:  $k * \sqrt[k]{n}$
- logn liste:  $\log n * \sqrt[\log n]{n}$



# Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlántuite?

- Costul căutării se modifică

- 2 liste:  $2 * \sqrt{n}$

- 3 liste:  $3 * \sqrt[3]{n}$

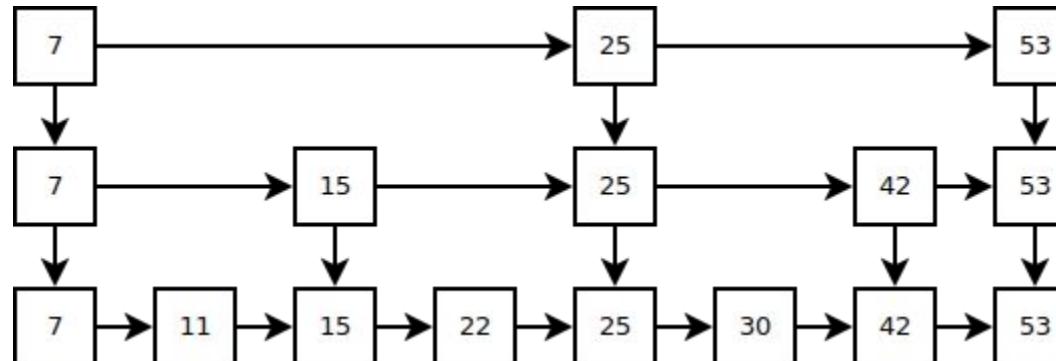
- k liste:

- logn liste:  $k * \sqrt[k]{n}$

= ? Cu cât este egal

?

$$\log n * \sqrt[\log n]{n}$$



# Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlántuite?

- Costul căutării se modifică

- 2 liste:  $2 * \sqrt{n}$

- 3 liste:  $3 * \sqrt[3]{n}$

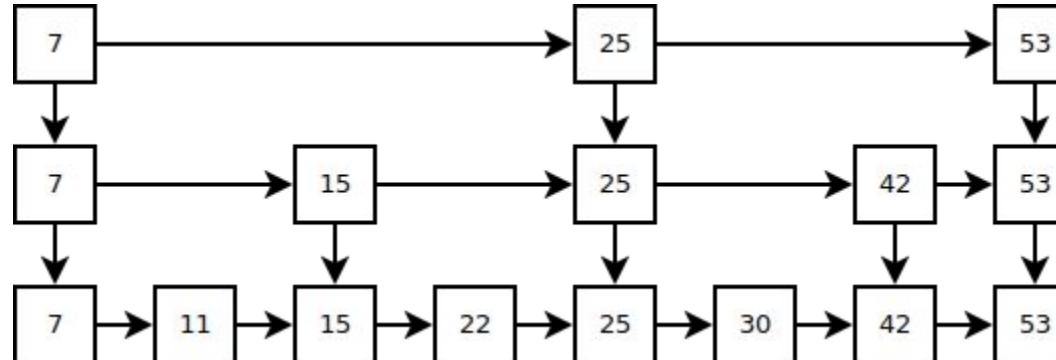
- k liste:

- logn liste:  $k * \sqrt[k]{n}$

 $=$ 

$\Rightarrow$  Complexitate:  $O(\log n)$  !

$$\log n * \sqrt[\log n]{n} = 2 * \log n$$



# Skip Lists - Căutare

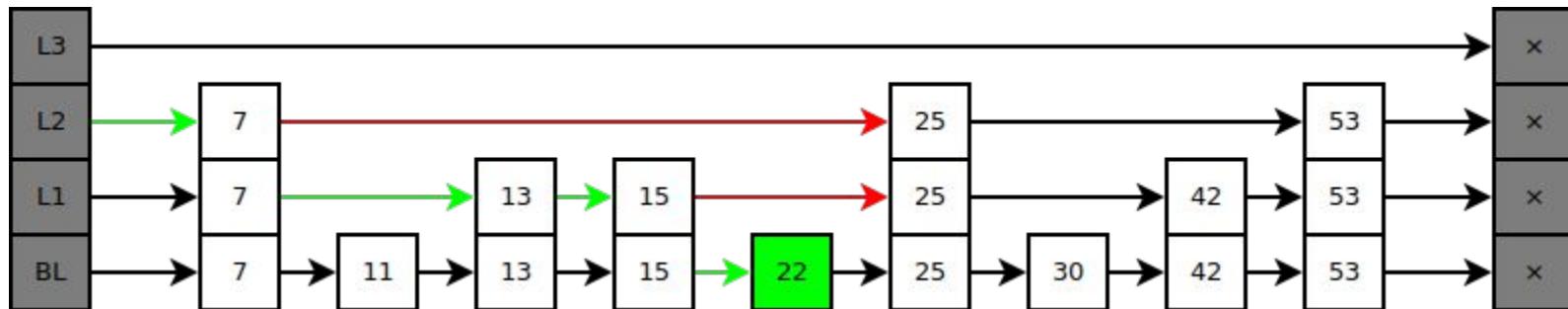
- 1) Începem căutarea cu primul nivel (cel mai de sus)
- 2) Avansăm în dreapta, până când, dacă am mai avansa, am merge prea departe (adică elementul următor este prea mare)
- 3) Ne mutăm în următoarea listă (mergem în jos)
- 4) Reluăm algoritmul de la pasul 2)

# Skip Lists - Căutare

- 1) Începem căutarea cu primul nivel (cel mai de sus)
- 2) Avansăm în dreapta, până când, dacă am mai avansat, am merge prea departe (adică elementul următor este prea mare)
- 3) Ne mutăm în următoarea listă (mergem în jos)
- 4) Reluăm algoritmul de la pasul 2)

Exemplu: search(22)

Complexitate:  $O(\log n)$



# Skip Lists - Inserare

- Vrem să inserăm elementul x
- **Observație:** Lista de jos trebuie să conțină toate elementele!
- x trebuie să fie inserat cu siguranță în nivelul cel mai de jos
  - căutăm locul lui x în lista de jos → search(x)
  - adăugăm x în locul găsit în lista cea mai de jos
- Cum alegem în câte liste să fie adăugat?

# Skip Lists - Inserare

- Vrem să inserăm elementul x
- x trebuie să fie inserat cu siguranță în nivelul cel mai de jos
- Cum alegem în ce altă listă să fie adăugat?
  - Alegem metoda probabilistică:
    - aruncăm o monedă
      - dacă pică Stema - o adăugăm în lista următoare și aruncăm din nou moneda
      - dacă pică Banul - ne oprim
    - probabilitatea să fie inserat și la nivelul următor:  $\frac{1}{2}$
- În medie:
  - $\frac{1}{2}$  elemente nepromovate
  - $\frac{1}{4}$  elemente promovate 1 nivel
  - $\frac{1}{8}$  elemente promovate 2 nivele
  - etc.
- Complexitate:  $O(\log n)$

## Skip Lists - Ștergere

- Ștergem elementul  $x$  din toate listele care îl conțin
- Complexitate:  $O(\log n)$

# Skip Lists

- [Articol](#)
- [Video MIT](#)
- [Notes](#)

# Bibliografie

<http://ticki.github.io/blog/skip-lists-done-right/>

<https://www.guru99.com/avl-tree.html>

<https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>

[MIT lecture notes on skip lists](#)

[Esoteric Data Structures: Skip Lists and Bloom Filters - Stanford University](#)

???

```
sol = 0;  
for (t = 1 << 30; t > 0; t>>=1) {  
    if (sol + t < v.size() && v[sol + t] <= x)  
        sol += t;  
}
```

???

```
sol = 0;  
for (t = 1 << 30; t > 0; t>>=1) {  
    if (sol + t < v.size() && v[sol + t] <= x)  
        sol += t;  
}
```

x=32

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	32	44	49	62	68	82	84	93	97

???

```
sol = 0; x = 32;  
for (t = 1 << 30; t > 0; t>>=1) {  
    if (sol + t < v.size() && v[sol + t] <= x)  
        sol += t;  
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	<b>32</b>	44	49	62	68	82	84	93	97

# Căutare binară

```
sol = 0; x = 32;  
for (t = 1 << 30; t > 0; t>>=1) {  
    if (sol + t < v.size() && v[sol + t] <= x)  
        sol += t;  
}
```

Complexitate?	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
3	7	11	20	24	28	30	<b>32</b>	44	49	62	68	82	84	93	97

# Căutare binară

```
sol = 0; x = 32;  
for (t = 1 << 30; t > 0; t>>=1) {  
    if (sol + t < v.size() && v[sol + t] <= x)  
        sol += t;  
}
```

Complexitate **O(log n)** - recomand cu căldură :)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	32	44	49	62	68	82	84	93	97

# AVL

AVL (veți avea la examen)

- Video (MIT de la minutul 29).
- Lecture Notes

---

Treapuri

---



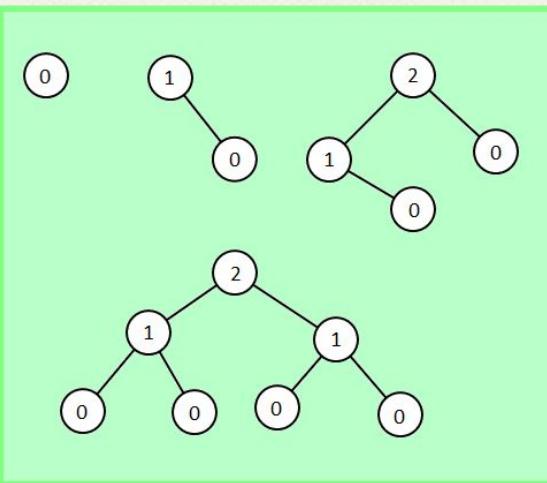
B-Arbori

# Arbore echilibrați

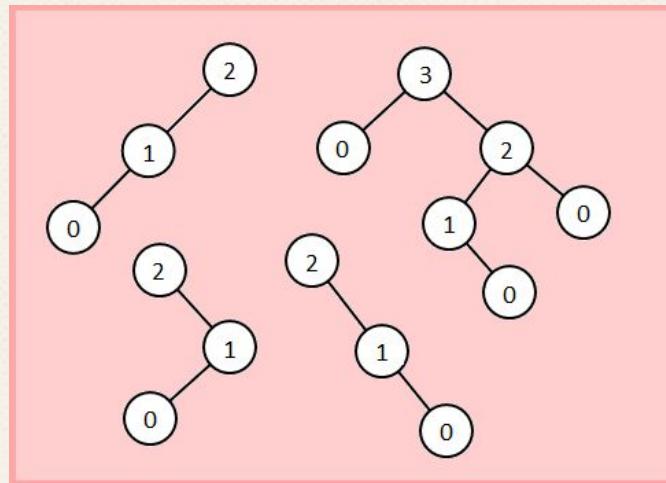
Un **arbore echilibrat** este un arbore în care, **pentru orice nod**, diferența dintre înălțimile subarborilor stâng și drept este de maxim 1.

Exemple:

Echilibrați



Neechilibrați

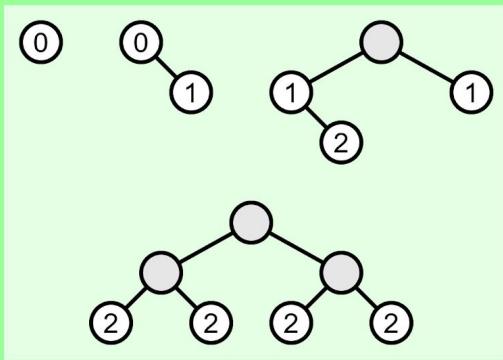


# Arbore echilibrați

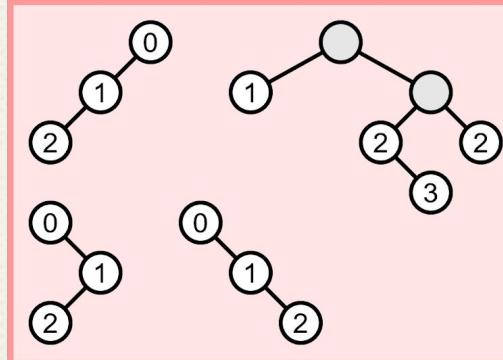
Un **arbore echilibrat** este un arbore în care, **pentru orice nod**, diferența dintre înălțimile subarborilor stâng și drept este de maxim 1.

Exemple:

Echilibrați



Neechilibrați



# B-Arbori

Un **B-Arbore** este un arbore echilibrat, destinat căutării eficiente de informație.

Un B-Arbore poate avea mai mult de 2 fii pentru un nod (se poate ajunge și la ordinul sutelor).

Totuși, înălțimea arborelui rămâne  $O(\log n)$ , datorită unei baze a logaritmului convenabilă.

În practică, B-Arborii sunt folosiți pentru baze de date și sisteme de fișiere, pentru citirea și scrierea eficientă pe discul de memorie.

O proprietate importantă a lor este faptul că rețin multă informație. De aceea, B-Arborii reduc numărul de accesări ale discului (accesarea discului este o operație costisitoare).

# B-Arbori

Un **B-Arbore** este un arbore echilibrat, destinat căutării eficiente de informație.

Un B-Arbore poate avea mai mult de 2 fii pentru un nod (se poate ajunge și la ordinul sutelor).

Totuși, înălțimea arborelui rămâne  $O(\log n)$ , datorită unei baze a logaritmului convenabilă.

## Proprietăți:

1. Un nod poate să conțină mai mult de o cheie
2. Numărul de chei ale unui nod  $x$  este  $n[x]$
3. Un nod  $x$  are  $n[x] + 1$  fii
4. Toate frunzele unui B-Arbore se află pe același nivel

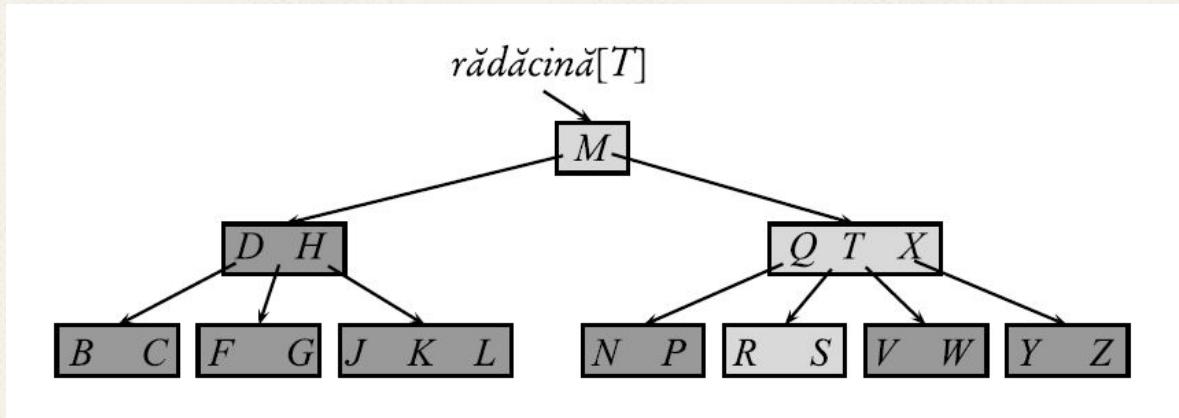
# B-Arbori

## Proprietăți:

1. Un nod poate să conțină mai mult de o cheie
2. Numărul de chei ale unui nod  $x$  este  $n[x]$
3. Un nod  $x$  are  $n[x] + 1$  fii
4. Toate frunzele unui B-Arbore se află pe același nivel

# B-Arbori

Exemplu:



Cheile acestui arbore sunt consoanele din alfabetul latin.

Observăm că un nod poate avea mai multe chei, iar fiecare nod **x** care are **n[x]** valori va avea **n[x] + 1** fii.

Căutarea literei "R" este exemplificată pe traseul hașurat cu o culoare mai deschisă.

# B-Arbori

## Câmpurile unui nod:

1. **n[x]** – numărul de chei memorate în nodul **x**
2. cele **n[x]** chei, memorate în ordine crescătoare:  
$$\text{cheie}_1[x] \leq \text{cheie}_2[x] \leq \text{cheie}_3[x] \leq \dots \leq \text{cheie}_{n[x]}[x]$$
3. o valoare booleană **frunză[x]** – **True**, dacă nodul **x** este frunză, **False**, dacă nodul **x** este nod intern

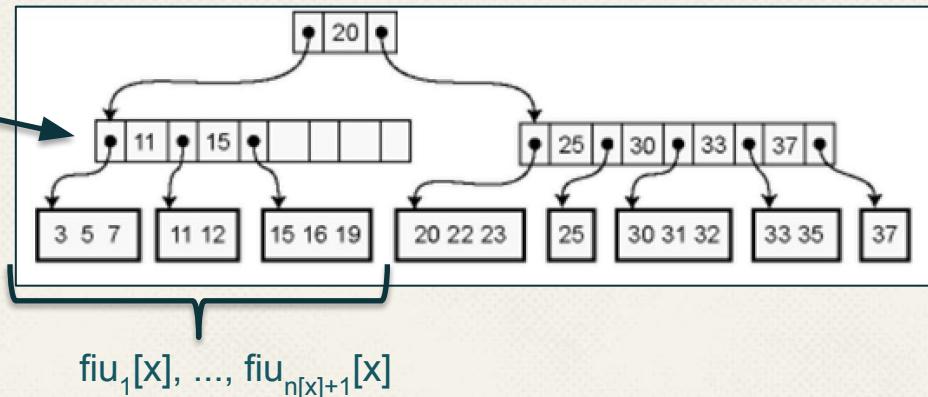
Dacă **x** este un nod intern, atunci el conține **n[x] + 1** pointeri către fiile săi. Nodurile frunză nu au fiile, deci nu au aceste câmpuri definite.

# B-Arbori

Cheile nodului  $x$  separă domeniile de chei aflate în fiecare subarbore astfel:

- dacă  $k_i$  este o cheie oarecare memorată într-un subarbore cu rădăcina  $\text{fiu}_i[x]$ , atunci  $k_1 \leq \text{cheie}_1[x] \leq k_2 \leq \text{cheie}_2[x] \leq \dots \leq \text{cheie}_{n[x]}[x] \leq k_{n[x]+1}$

Nodul  $x$



$$\begin{array}{l} 3, 5, 7 \leq 11 \\ 11 \leq 11, 12 \\ 11, 12 \leq 15 \\ 15 \leq 15, 16, 19 \end{array}$$

# B-Arbori

## Gradul unui B-Arbore

Există o limitare inferioară și una superioară a numărului de chei ce pot fi conținute într-un nod. Exprimăm aceste margini printr-un întreg fixat  $t \geq 2$ , numit **grad minim** al B-Arborelui.

# B-Arbori

## Restricții:

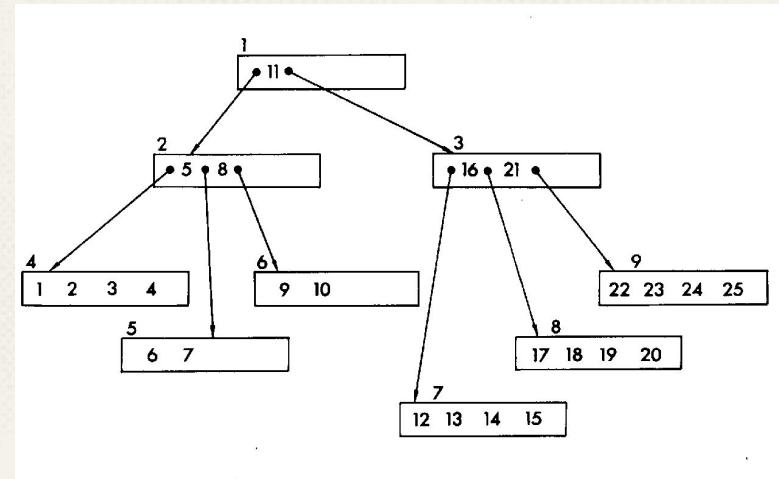
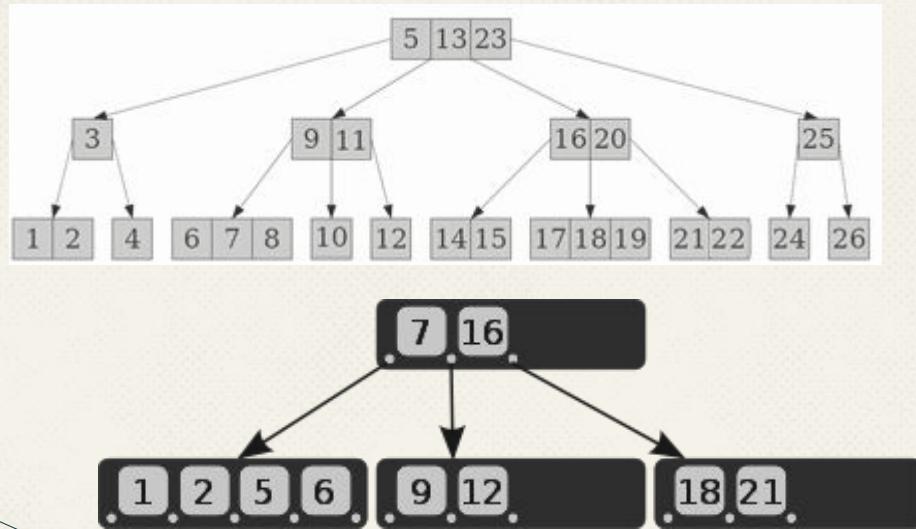
1. Fiecare nod, cu excepția rădăcinii, trebuie să aibă **cel puțin  $t - 1$  chei**.  
Consecintă: fiecare nod intern trebuie să aibă **cel puțin  $t$  fii**.
2. Dacă arborele este nevid, atunci rădăcina trebuie să aibă cel puțin o cheie.
3. Fiecare nod poate să aibă **cel mult  $2t - 1$  chei**.  
Consecintă: orice nod intern poate să aibă **cel mult  $2t$  fii**.

Un nod cu  $2t - 1$  chei se numește **nod plin**.

# B-Arbori

Exemple de B-Arbori:

B-Arbore de ordin 2 (arbore 2-3-4)



# Înălțimea unui B-Arbore

**Teoremă:** Dacă  $n \geq 1$ , atunci, pentru orice B-Arbore  $T$  cu  $n$  chei și grad minim  $t \geq 2$ :

$$h \leq \log_t \frac{n+1}{2} - \text{înălțimea arborelui}$$

**Demonstrație:** Dacă un B-Arbore are înălțimea  $h$ , atunci va avea număr minim de chei dacă rădăcina conține o singură cheie, iar toate celelalte noduri câte  $t - 1$  chei.

În acest caz, există:

- pe nivelul 1: 2 noduri
- pe nivelul 2:  $2t$  noduri
- pe nivelul 3:  $2t^2$  noduri
- ...
- pe nivelul  $h$ :  $2t^{h-1}$

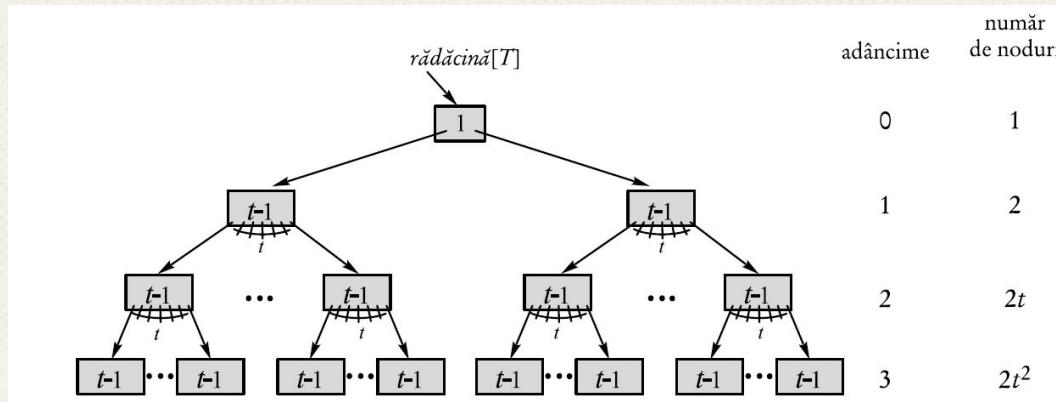
Deci:

$$n \geq 1 + (t - 1) \sum_{i=1}^h 2t^{i-1} = 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right) = 2t^h - 1$$

# Înăltimea unui B-Arbore

Exemplu:

Pentru un B-Arbore de înăltime 3, cu număr minim de chei, care are, în fiecare nod, numărul de chei reținute  $n[x]$ , avem următorul desen:



# Înălțimea unui B-Arbore

**Concluzie:**

Înălțimea unui arbore este

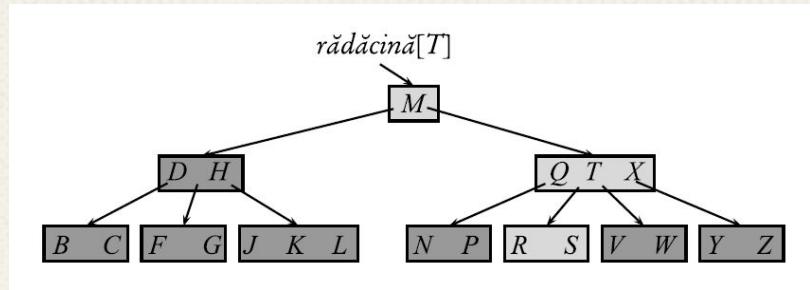
$$h \leq \log_t \frac{n+1}{2}$$

Deci, înălțimea unui B-Arbore crește proporțional cu  **$O(\log n)$** .

# Discuție

## Exerciții:

1. De ce nu putem permite gradul minim  $t = 1$  ?
2. Pentru ce valori ale lui  $t$ , arborele de mai jos este un B-Arbore, conform definiției?



3. Desenați toti B-Arborii corecți cu grad minim 2 care să reprezinte mulțimea  $\{1, 2, 3, 4, 5\}$ .

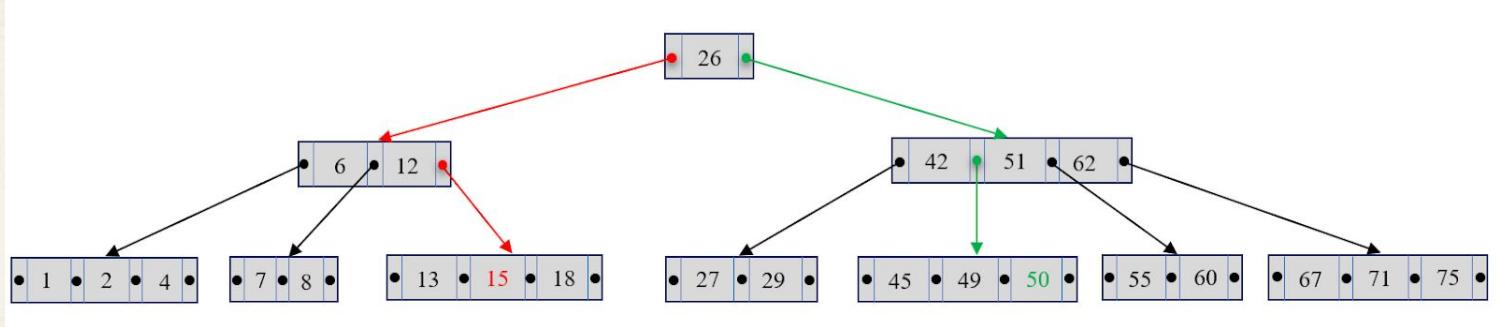
# Operații de bază

# Căutarea În B-Arbore

Căutarea într-un B-Arbore este asemănătoare cu o căutare într-un arbore binar.

Într-un B-Arbore, căutarea se realizează comparând cheia căutată **x** cu cheile nodului curent, plecând de la nodul rădăcină.

Căutare reușită pentru 50 și nereușită pentru 17.

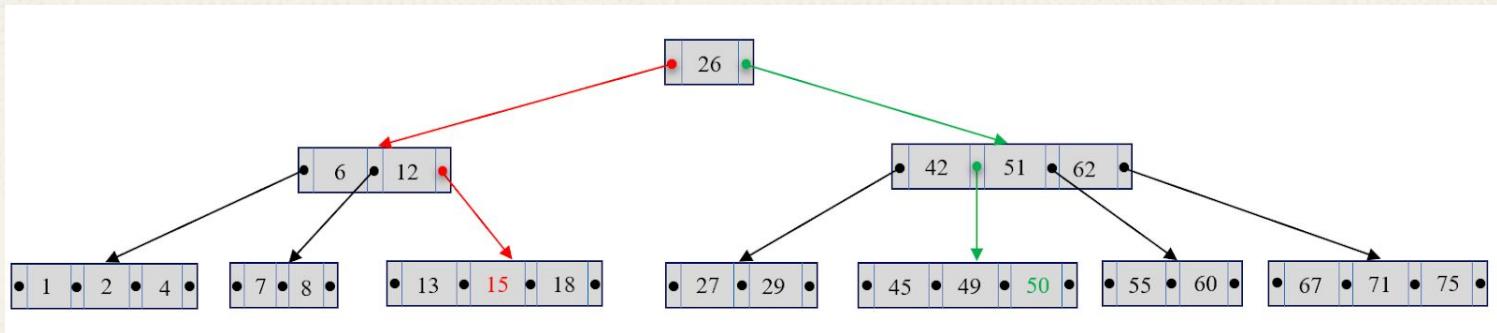


# Căutarea în B-Arbore

Algoritm:

1. Căutăm cheia **x** în rădăcină
2. Dacă nu o găsim, atunci continuăm căutarea în fiul corespunzător valorii **x**
3. Dacă găsim cheia, returnăm perechea de valori (**y, i**), reprezentând nodul, respectiv poziția în nod pe care s-a găsit valoarea **x**.

Putem afla indicele fiului care trebuie explorat în continuare la pasul 2 folosind **căutarea binară dacă numărul de valori din fiecare nod este mare**.



# Căutarea în B-Arbore

Căutarea într-un B-Arbore este asemănătoare cu o căutare într-un arbore binar.

Într-un B-Arbore, căutarea se realizează comparând cheia căutată **x** cu cheile nodului curent, plecând de la nodul rădăcină.

Algoritm:

1. Căutăm cheia **x** în rădăcină
2. Dacă nu o găsim, atunci continuăm căutarea în fiul corespunzător valorii **x**
3. Dacă găsim cheia, returnăm perechea de valori  $(y, i)$ , reprezentând nodul, respectiv poziția în nod pe care s-a găsit valoarea **x**.

Puteți afla indicele fiului care trebuie explorat în continuare la pasul 2 folosind căutarea binară.

# Căutarea în B-Arbore

## Complexitate:

Procesul se repetă de cel mult **O(h)** ori, în cazul în care valoarea căutată se află într-o frunză.

Căutarea valorii într-un nod se realizează (folosind căutarea binară) în **O(log t)**.

Complexitate finală:

$$\begin{aligned} O(h * \log t) &= O(\log t * \log_t n) \\ &= O(\log t * (\log n / \log t)) \\ &= \mathbf{O(\log n)} \end{aligned}$$

# Inserarea În B-Arbore

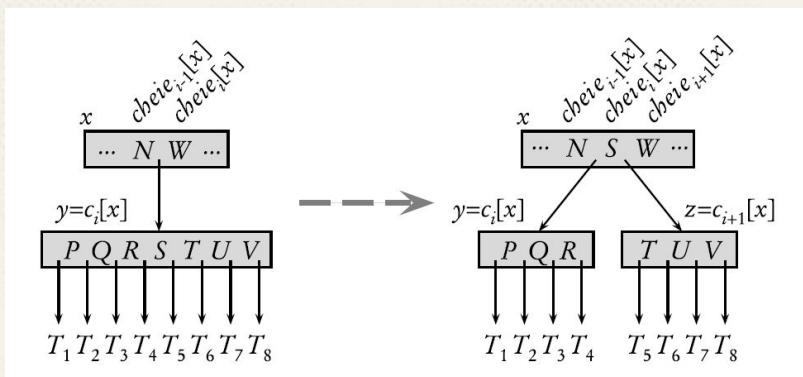
Pentru a inseră o cheie  $x$  într-un B-Arbore, trebuie distinse două cazuri: când nodul unde trebuie introdus are mai puțin de  $2t-1$  chei, respectiv când are  $2t-1$  chei.

Algoritm:

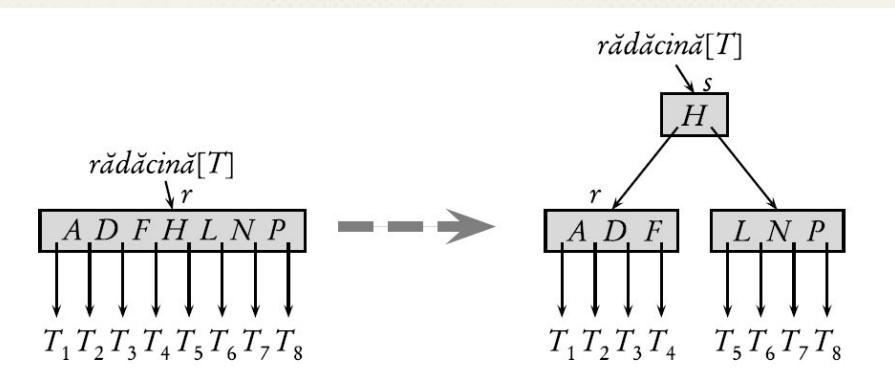
1. Aplicăm operația de căutare pentru a găsi nodul unde trebuie introdusă cheia. Notăm acest nod cu  $X$  și va fi o **frunză**.
2. Dacă  $X$  are mai puțin de  $2t-1$  chei, atunci inserarea se efectuează fără a modifica structura arborelui.
3. Dacă  $X$  are  $2t-1$  chei, atunci acesta trebuie divizat. Rezultă, astfel, două noduri noi,  $F_s$  (fiul din stânga) și  $F_d$  (fiul din dreapta).
4. Eliminăm cea mai mare cheie din  $F_s$  (cheia mediană). O notăm cu  $M$ .
5.  $M$  devine părintele celor două noduri  $F_s$  și  $F_d$ .
6. Se încearcă (recursiv) adăugarea lui  $M$  în părintele lui  $X$ .

# Inserarea în B-Arbore

Divizarea unui nod ( $t = 4$ ):



Nod intermediar



Rădăcină

# Inserarea în B-Arbore

## Complexitate:

Căutarea nodului în care trebuie introdusă cheia:  $O(\log_t n)$

Pentru un nivel:  $O(t)$

Recursivitatea:  $O(h) = O(\log_t n)$

Complexitatea finală:  $O(t \log_t n)$



# **Arbore Binari de Căutare**

## **Construiți Aleator**

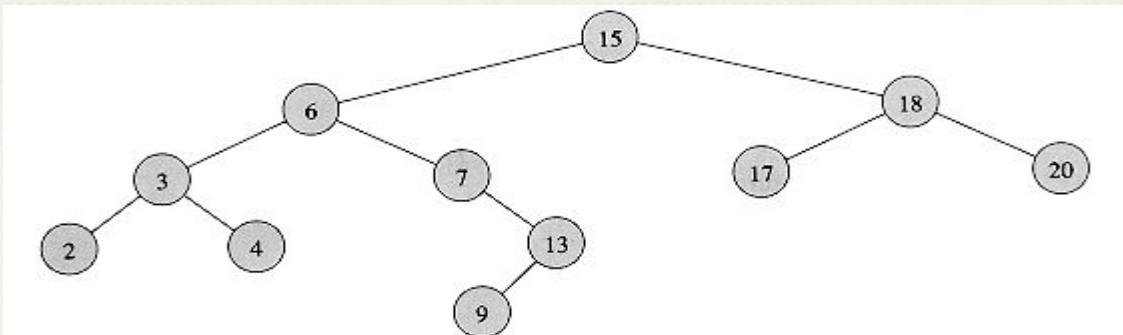
- Amestecăm bine de tot și inserăm elementele în arborele binar de căutare. Ce înălțime va avea?

# Arbore Binari de Căutare

## Construiți Aleator

**Lema 13.3.** Notăm cu  $T$  arborele care rezultă prin inserarea a  $n$  chei distincte  $k_1, k_2, \dots, k_n$  (în ordine) într-un arbore binar de căutare inițial vid. Cheia  $k_i$  este un strămoș al cheii  $k_j$  în  $T$ , pentru  $1 \leq i < j \leq n$ , dacă și numai dacă:

- $k_i = \min\{ k_l : 1 \leq l \leq i \text{ și } k_l > k_i \}$  //  $k_i$  este cel mai mic număr mai mare decât  $k_i$  din primele  $i$ , practic în procesul de inserare a lui  $j$  vom ajunge în  $k_i$  și vom merge în stânga
- SAU  $k_i = \max\{ k_l : 1 \leq l \leq i \text{ și } k_l < k_i \}$ 
  - 13 e fiu al lui 7 (până la momentul inserării lui 13, 7 era cel mai mare număr mai mic)
  - Ulterior, proprietatea e valabilă și pentru 9 și 14.
  - Ce trebuia să se întâpte ca 18 să fie fiu al lui 7?



# Arbore Binari de Căutare

## Construiți Aleator

*Demonstrație:*

‘ $\Rightarrow$ ’: Presupunem că  $k_i$  este un strămoș al lui  $k\Box$ . Notăm cu  $T_i$  arborele care rezultă după ce au fost inserate în ordine cheile  $k_1, k_2, \dots, k_i$ . Drumul de la rădăcină la nodul  $k_i$  în  $T_i$  este același cu drumul de la rădăcină la nodul  $k_i$  în  $T$ . De aici, rezultă că, dacă s-ar insera în arborele  $T_i$  nodul  $k\Box$ , acesta ( $k\Box$ ) ar deveni fie fiu stâng, fie fiu drept al nodului  $k_i$ . Prin urmare (vezi exercițiul 13.2-6),  $k_i$  este fie cea mai mică valoare dintre  $k_1, k_2, \dots, k_i$  care este mai mare decât  $k\Box$ , fie cea mai mare valoare dintre cheile  $k_1, k_2, \dots, k_i$  care este mai mică decât  $k\Box$ .

‘ $\Leftarrow$ ’: Presupunem că  $k_i$  este cea mai mică valoare dintre  $k_1, k_2, \dots, k_i$  care este mai mare decât  $k\Box$ . (Cazul când  $k_i$  este cea mai mare cheie dintre  $k_1, k_2, \dots, k_i$  care este mai mică decât  $k\Box$  se tratează simetric). Compararea cheii  $k\Box$  cu oricare dintre cheile de pe drumul de la rădăcină la  $k_i$  în arborele  $T$  produce aceleași rezultate ca și compararea cheii  $k_i$  cu cheile respective. Prin urmare, pentru inserarea lui  $k\Box$ , se va parcurge drumul de la rădăcină la  $k_i$ , apoi  $k\Box$  se va insera ca descendenter al lui  $k_i$ .

# Arbore Binari de Căutare

## Construiți Aleator

**Corolarul 13.4.** Fie  $T$  arborele care rezultă prin inserarea a  $n$  chei distințe  $k_1, k_2, \dots, k_n$  (în ordine) într-un arbore binar de căutare inițial vid. Pentru o cheie  $k_j$  dată, cu  $1 \leq j \leq n$ , definim mulțimile:

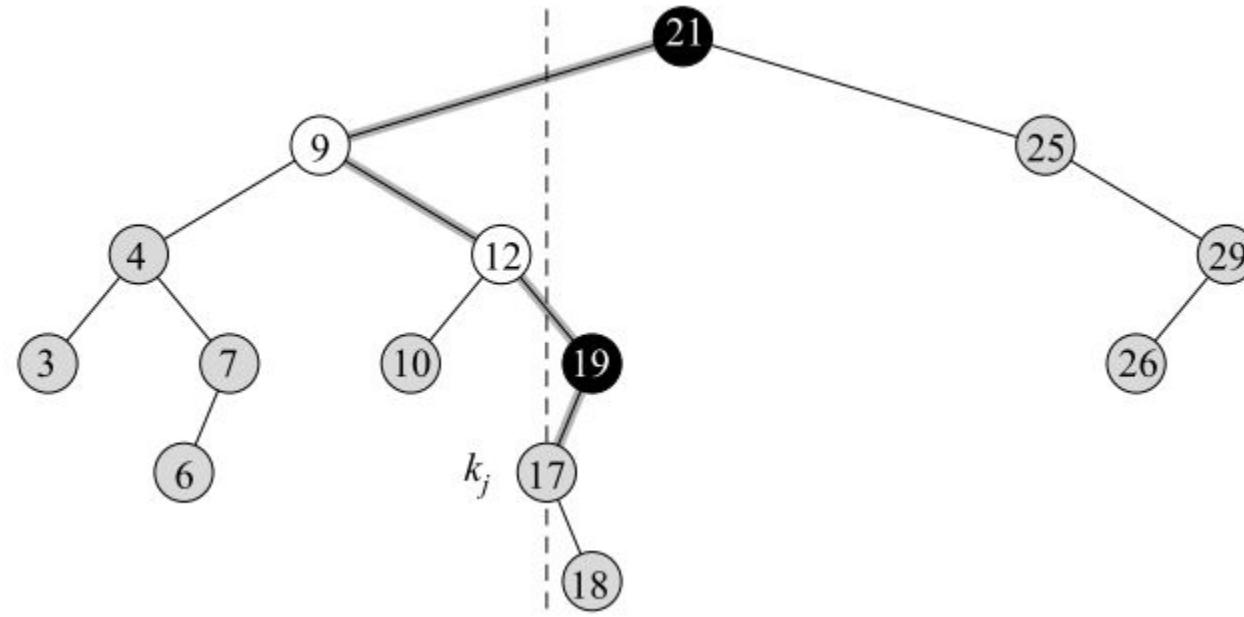
- $G_j = \{ k_i : 1 \leq i < j \text{ și } k_j > k_i > k_j \}$  pentru toți indicii  $l < i$  cu  $k_l > k_j$
- $L_j = \{ k_i : 1 \leq i < j \text{ și } k_j < k_i < k_j \}$  pentru toți indicii  $l < i$  cu  $k_l < k_j$

Atunci cheile de pe drumul de la rădăcină la  $k_j$  sunt chiar cheile din  $G_j \cup L_j$ , iar adâncimea oricărei chei  $k_j$  din  $T$  este  $d(k_j, T) = |G_j| + |L_j|$ .

# Arbore Binari de Căutare

## Construiri Aleator

Cu negru sunt nodurile care sunt, la inserarea lor, cel mai mic element mai mare decât 19 ( $G \rightarrow$  greater). Similar, cele cu alb sunt elemente care, la inserarea lor, erau cele mai mari elemente mai mici decât 19 ( $L \rightarrow$  lower).



# Arbore Binari de Căutare Construiți Aleator

Practic, pentru a calcula înălțimea unui arbore, trebuie să calculăm  $\max_{1 \leq j \leq n} (|G_j| + |L_j|)$ .

Simplificăm și discutăm cum calculăm de câte ori se modifică, în medie, minimul, dacă inserăm **n** elemente pe rând.

**Exercițiu:** Care este probabilitatea ca  $k_i$  să fie minimul primelor  $i$  numere?

# Arbore Binari de Căutare

## Construiți Aleator

Practic, pentru a calcula înălțimea unui arbore, trebuie să calculăm  $\max_{1 \leq j \leq n} (|G_j| + |L_j|)$ .

Simplificăm și discutăm cum calculăm de câte ori se modifică, în medie, minimul, dacă inserăm **n** elemente pe rând.

**Răspuns:** Probabilitatea ca  $k_i$  să fie minimul primelor  $i$  numere este  $1/i$ .

$$\sum_{i=1}^n \frac{1}{i} = H_n$$

Prin urmare, numărul mediu de modificări este ,

unde  $H_n = \ln(n) + O(1)$  este al  $n$ -lea număr armonic.

→ Avem  $\log(n)$  modificări.

# Arbore Binari de Căutare

## Construiri Aleator

**Lema 13.5.** Fie  $k_1, k_2, \dots, k_n$  o permutare oarecare a unei mulțimi de  $n$  numere distincte și fie  $|S|$  variabilă aleatoare reprezentând cardinalul mulțimii.

$$S = \{ k_i : 1 \leq i \leq n \text{ și } k_l > k_i \text{ pentru orice } l < i \} \quad (13.1)$$

Atunci  $\Pr\{ |S| \geq (\beta + 1)H_n \} \leq 1/(n^2)$ , unde  $H_n$  este al  $n$ -lea număr armonic, iar  $\beta \approx 4,32$  verifică ecuația  $(\ln \beta - 1)\beta = 2$ .

**Prin urmare, e foarte probabil să avem maxim  $O(\log(n))$  modificări ale minimului.**

# Arbore Binari de Căutare Construiți Aleator

**Teorema 13.6.** Înălțimea medie a unui arbore binar de căutare construit aleator cu  $n$  chei distincte este  $O(\lg n)$ .

# Arbore Binari de Căutare

## Construiți Aleator

**Teorema 13.6.** Înălțimea medie a unui arbore binar de căutare construit aleator cu  $n$  chei distincte este  $O(\lg n)$ .

Demonstrație: Fie  $k_1, k_2, \dots, k_n$  o permutare oarecare a celor  $n$  chei și fie  $T$  arborele binar de căutare care rezultă prin inserarea cheilor în ordinea specificată, pornind de la un arbore inițial vid. Vom discuta prima dată probabilitatea ca adâncimea  $d(k_i, T)$  a unei chei date  $k_i$  să fie cel puțin  $t$ , pentru o valoare  $t$  arbitrară. Conform caracterizării adâncimii  $d(k_i, T)$  din **corolarul 13.4**, dacă adâncimea lui  $k_i$  este cel puțin  $t$ , atunci cardinalul uneia dintre cele două mulțimi  $G_i$  și  $L_i$  trebuie să fie cel puțin  $t/2$ .

Prin urmare,  $\Pr\{ d(k_i, T) \geq t \} \leq \Pr\{ |G_i| \geq t/2 \} + \Pr\{ |L_i| \geq t/2 \}.$

# Arbore Binari de Căutare

## Construiri Aleator

Să examinăm la început  $\Pr\{ |G| \geq t/2 \}$ . Avem

$$\begin{aligned}\Pr\{ |G| \geq t/2 \} &= \Pr\{ |\{k_i : 1 \leq i < j \text{ și } k_j > k_i > k_l, \forall l < i\}| \geq t/2 \} \\ &\leq \Pr\{ |\{k_i : i \leq n \text{ și } k_i > k_l, \forall l < i\}| \geq t/2 \} \\ &= \Pr\{ |S| \geq t/2 \},\end{aligned}$$

unde S este definit în relația (13.1.)  $S = \{ k_i : 1 \leq i \leq n \text{ și } k_i > k_l, \forall l < i \}$ .

În sprijinul acestei afirmații, să observăm că probabilitatea nu va descrește dacă vom extinde intervalul de variație al lui i de la  $i < j$  la  $i \leq n$ , deoarece, prin extindere, se vor adăuga elemente noi la mulțime. Analog, probabilitatea nu va descrește dacă se renunță la condiția  $k_i > k_l$ , deoarece, prin aceasta, se înlocuiește o permutare a (de regulă) mai puțin de **n** elemente (și anume acele chei  $k_i$  care sunt mai mari decât  $k_l$ ) cu o altă permutare oarecare de n elemente. Folosind o argumentare similară, putem demonstra că

$$\Pr\{ |L| \geq t/2 \} \leq \Pr\{ |S| \geq t/2 \}.$$

# Arbore Binare de Căutare

## Construiri Aleator

Folosind o argumentare similară, putem demonstra că

$$\Pr\{ |L_k| \geq t/2 \} \leq \Pr\{ |S| \geq t/2 \}$$

și apoi, folosind inegalitatea (13.2), obținem:

$$\Pr\{ d(k, T) \geq t \} \leq 2 * \Pr\{ |S| \geq t/2 \}.$$

Dacă alegem  $t = 2(\beta + 1)H_n$ , unde  $H_n$  este al n-lea număr armonic, iar  $\beta \approx 4.32$  verifică ecuația  $(\ln \beta - 1)\beta = 2$ , putem aplica **lema 13.5** pentru a concluziona că

$$\Pr\{ d(k, T) \geq 2(\beta + 1)H_n \} \leq 2 * \Pr\{ |S| \geq (\beta + 1)H_n \} \leq 2/n^2.$$

# Arbore Binari de Căutare

## Construiți Aleator

Deoarece discutăm despre un arbore binar de căutare construit aleator și cu cel mult  $n$  noduri, probabilitatea ca adâncimea oricărui dintre noduri să fie cel puțin  $2(\beta + 1)H\Box$  este, folosind *inegalitatea lui Boole*\*, de cel mult  $n^*(2/n^2) = 2/n$ . Prin urmare, în cel puțin  $1 - 2/n$  din cazuri, înălțimea arborelui binar de căutare construit aleator este mai mică decât  $2(\beta + 1)H\Box$  și în cel mult  $2/n$  din cazuri înălțimea este cel mult  $n$ . În concluzie, înălțimea medie este cel mult

$$(2(\beta + 1)H\Box)(1 - 2/n) + n(2/n) = O(\lg n).$$

\**Inegalitatea lui Boole*:

Fie  $A_1, A_2, \dots, A\Box$  în  $K$  cu  $\bigcap_{i=1}^{n-1} A_i \neq 0$ . Atunci:  $\Pr(\bigcap_{i=1}^{n-1} A_i) \geq (\sum_{i=1}^n \Pr(A_i)) - n - 1$

# Treapuri

---

PĂTRAȘCU ADRIAN-OCTAVIAN

GRUPA 131

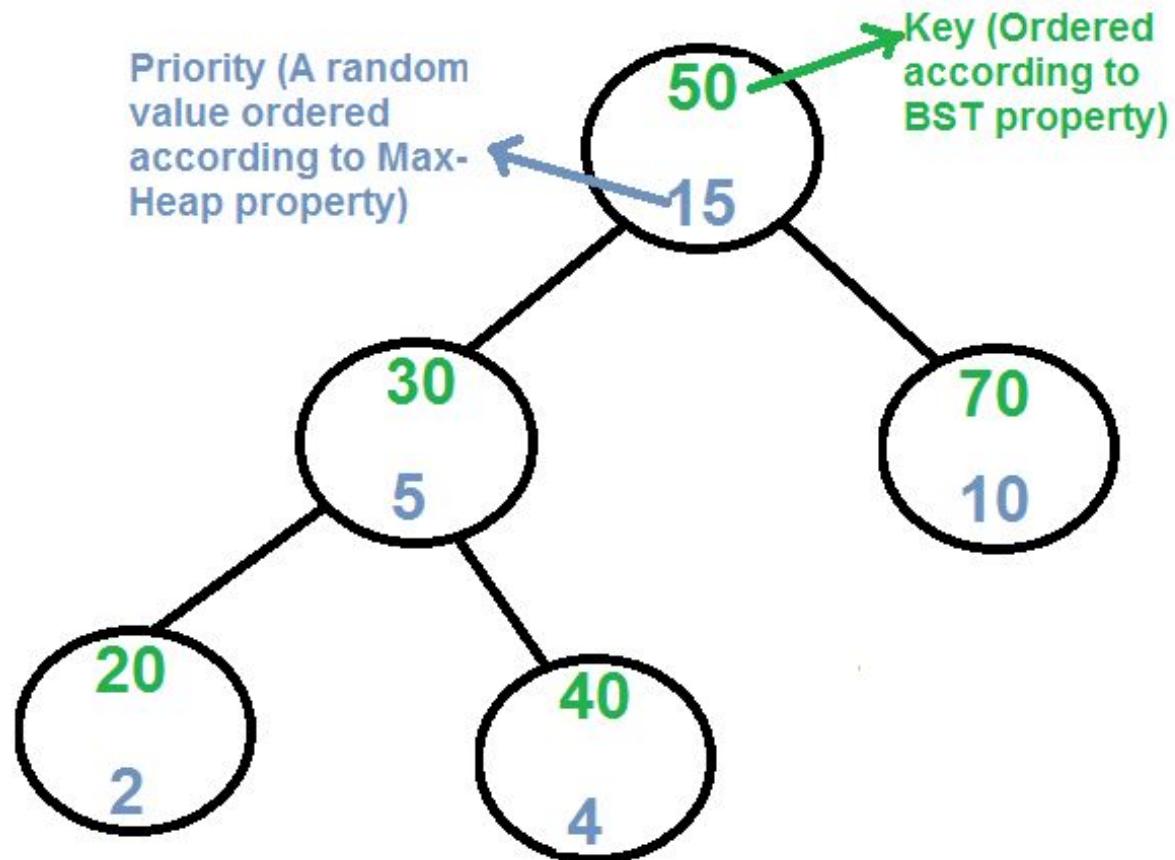
# Cuprins

---

- Ce-i un treap? Motivație (Definiții + discuție)
- Operații:
  1. Rotații pe arbori
  2. Inserție  $O(\log N)$
  3. Căutare  $O(\log N)$
  4. Ștergere  $O(\log N)$
- Bibliografie

# Treap

- Structură de date arborescentă care menține simultan proprietatea de arbore binar de căutare (ABC) și cea de max-heap.
- Puțin mai formal, fie  $(X_i, Y_i)$  nodurile arborelui. Treapul asigură structură de ABC pentru toți  $X_i$ , iar în  $Y_i$  structura de max-heap.



# De ce ne plac treapurile?

---

- Sunt ușor de implementat
- Sunt mai rapizi decât arborii roșu-negru (ARN) și decât skip-listurile [\[1.1\]](#), [\[1.2\]](#)
- Cu puține modificări permit abordarea multor tipuri de query-uri și update-uri (sume, cmmdc, rotații etc pe intervale)

```
struct Treap
{
    int key, p;
    Treap *left, *right;
};

Treap* New_node(int x)
{
    Treap* nod = new Treap;
    nod->key = x;
    nod->p = rand();
    nod->left = nod->right = NULL;
    return nod;
}
```

The diagram illustrates the structure of a Treap node and its creation. It shows a class definition for 'Treap' with members: 'key' (int), 'p' (int), 'left' (Treap\*), and 'right' (Treap\*). Below it, a constructor function 'New\_node' is defined, which creates a new node ('nod') using 'new', initializes its 'key' to 'x', sets its 'p' value to the result of 'rand()', and sets both 'left' and 'right' pointers to 'NULL'. A vertical line connects the two code snippets.

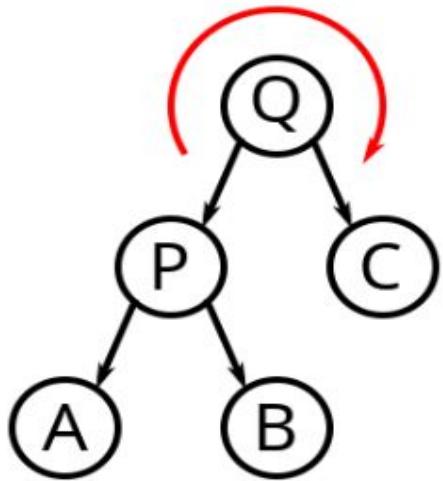


Arbore  
binar de  
căutare + max-heap

---

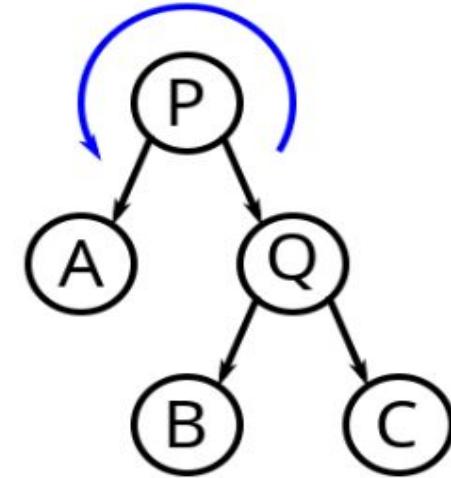
Treap

# Rotății pe arbori



Rotate Right  
← Rotate Left

A < P < B < Q < C



```
Treap* rot_right(Treap* y)  
{  
    Treap *x = y->left;  
    y->left = x->right;  
    x->right = y;  
    return x;  
}
```

```
Treap* rot_left(Treap* y)  
{  
    Treap *x = y->right;  
    y->right = x->left;  
    x->left = y;  
    return x;  
}
```

# Insetție

---

Inserăm, recursiv, nodul ca într-un ABC

Odată inserat, la fiecare pas înapoi în recursivitate rotim în sens invers subarborele cu rădăcina în nodul curent în funcție de tipul fiului (stâng/drept)

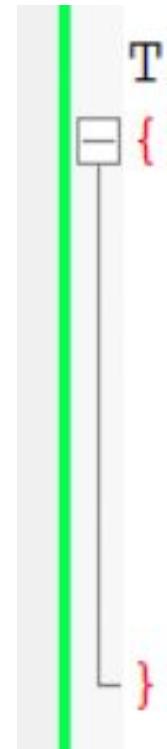
```
void Insert(Treap* &node, int key)
{
    if(node->key == key)
        return;
    if(node == NULL)
        node = New_node(key);
    else
    {
        if(key < node->key)
        {
            Insert(node->left, key);
            if(node->p < node->left->p)
                node = rot_right(node);
        }
        else
        {
            Insert(node->right, key);
            if(node->p < node->right->p)
                node = rot_left(node);
        }
    }
}
```

# Căutare

---

Exact ca la un ABC,  
anume:

- 1) Dacă valoarea căutată este mai mică decât cea pe care o caut, atunci cobor în subarborele stâng
- 2) Analog dacă valoarea căutată este mai mare



```
Treap* Search(Treap* node, int key)
{
    if(node->key == key)
        return node;
    if(key < node->key)
        Search(node->left, key);
    else
        Search(node->right, key);
```

# Ștergere

Există trei cazuri în care se poate afla un nod pe care vrem să-l ștergem:

1. E frunză -> îl ștergem direct.
2. Are numai un fiu -> Fiul ia locul nodului respectiv
3. Are doi fii -> Rotim în locul rădăcinii subarborelui făcut din nodul curent fiul cu prioritatea cea mai mare. Repetăm algoritmul până când ne aflăm într-unul dintre primele două cazuri.

```
Treap* Delete(Treap* node, int key)
{
    if (!node)
        return node;
    if (key < node->key)
        node->left = Delete(node->left, key);
    else if (key > node->key)
        node->right = Delete(node->right, key);
    else if (!node->left || !node->right)
    {
        Treap* aux = (node->left ? node->left : node->right);
        delete node;
        node = aux;
    }
    else if (node->left->p > node->right->p)
    {
        node = rot_right(node);
        node->right = Delete(node->right, key);
    }
    else
    {
        node = rot_left(node);
        node->left = Delete(node->left, key);
    }
    return node;
}
```

# Bibliografie

---

- [1.1] [pinporelmundo: Skip Lists compared with Treaps and Red-Black Trees](#)
- [1.2] [7.2 Treap: A Randomized Binary Search Tree \(opendatastructures.org\)](#) (La secțiunea 7.2.1)
- [2] [Treapuri \(infoarena.ro\)](#)
- [3] [Treap | Set 2 \(Implementation of Search, Insert and Delete\) – GeeksforGeeks](#)

Despre rotiri de arbori: [Balanced binary search tree rotations – YouTube](#)

Implementarea operațiilor: [Treap - Pastebin.com](#)

---

Discuti examen saptamana viitoare

Deadline laborator... ramane la fel, vreti o saptamana in plus -10%..

# Arbore de intervale



# Arbore de Intervale

**Problema.** Se dă un vector cu n numere și operații de genul:

- Adăugăm la poziția i valoarea x (x poate fi și negativ)
- Cerem maximul pe intervalul i, j (ex 3 6)

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8

Cum putem face asta?

# Şmenul lui Batog

**Problemă.** Se dă un vector cu n numere și operații de genul:

- Adăugăm la poziția i valoarea x (x poate fi și negativ)
- Cerem minimul pe intervalul i, j (ex 3 6)

0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
9			34			11		

Împărțim vectorul în zone de L (?) și calculăm minimul pe fiecare zonă în parte.

# Şmenul lui Batog

**Problemă.** Se dă un vector cu n numere și operații de genul:

- Adăugăm la poziția i valoarea x (x poate fi și negativ)
  - Pentru că, dacă facem maximul mai mic, trebuie să găsim noul maxim  $O(\sqrt{n})$
- Cerem maximul pe intervalul i, j (ex 3 6)

0	1	2	3	4	5	6	7	8
3	9	2	5	7	3	6	11	8
9			7			11		

Cum răspundem la 0, 8? Dar la 0, 4? Dar la 1, 7 ?

Care este complexitatea ?

# Şmenul lui Batog

**Complexitate query:** Împărțim în  $n/L$  zone de lungime  $L$

$O(n/L(\text{nr de zone}) + 2 * L(2 \text{ zone le pot itera aproape complet})) \rightarrow L = \sqrt{n}$

$O(\sqrt{n} + 2 * \sqrt{n}) = O(\sqrt{n})$

0	1	2	3	4	5	6	7	8
3	9	2	5	7	3	6	11	8
9			7			11		

# Şmenul lui Batog

Împărțim în zone de:

- $\text{sqrt}(n)$  sau...
- $\text{sqrt}(n)/2$
- $\text{sqrt}(n) * 2 .. \text{ și}$
- Variațiuni
- De ce?
  - Pentru că, în practică, nu  $\text{sqrt}(n)$  va fi cel mai rapid. Totuși,  $\text{sqrt}(n)$  este o alegere buna în general.

# Şmenul lui Batog

**Problemă.** Se dă un vector cu n numere. Sortați-l!

problemă: <https://leetcode.com/problems/sort-an-array/submissions/>

cod: <https://pastebin.com/bFHYephh>

0	1	2	3	4	5	6	7	8
3	9	50001	5	7	34	6	11	8
3			5			6		

# Arbore de Intervale

**Problema.** Se dă un vector cu n numere și operații de genul:

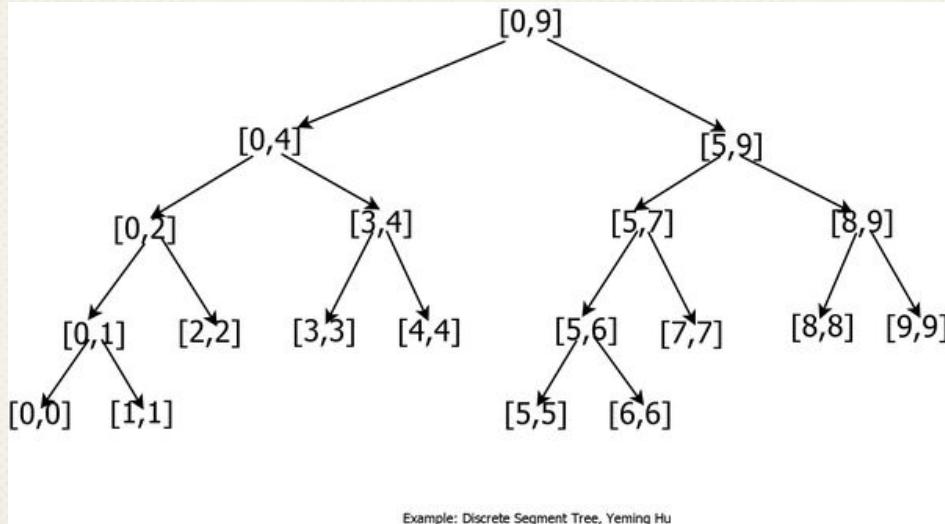
- Adăugăm la poziția i valoarea x (x poate fi și negativ)
- Cerem minimul pe intervalul i, j (ex 3 6)

0	1	2	3	4	5	6	7	8	9
3	9	2	5	7	34	6	11	8	44

# Arbore de Intervale

Arbore cu rădăcina ținând intervalul  $[0, n]$

Pentru un nod ce ține intervalul  $[L, R] \rightarrow$  fiul stâng ține  $[L, (L+R)/2]$ , cel drept  $[(L+R)/2, R]$

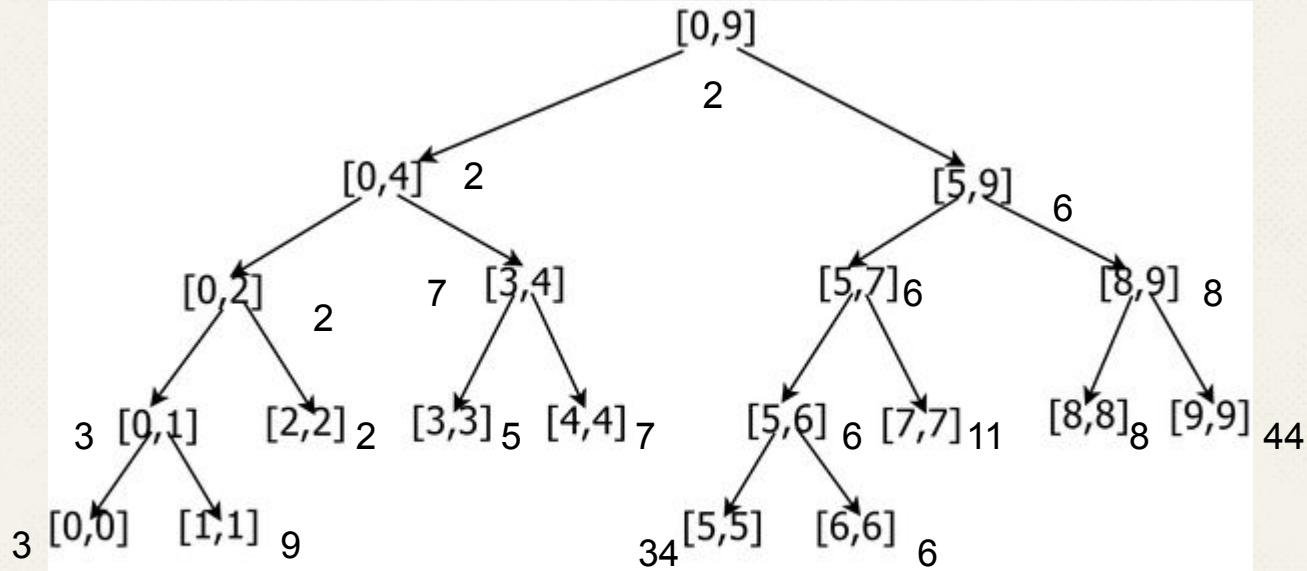


0	1	2	3	4	5	6	7	8	9
3	9	2	<b>5</b>	7	<b>34</b>	<b>6</b>	11	8	44

# Arbore de Intervale

0	1	2	3	4	5	6	7	8	9
3	9	2	<b>5</b>	<b>7</b>	<b>34</b>	<b>6</b>	11	8	44

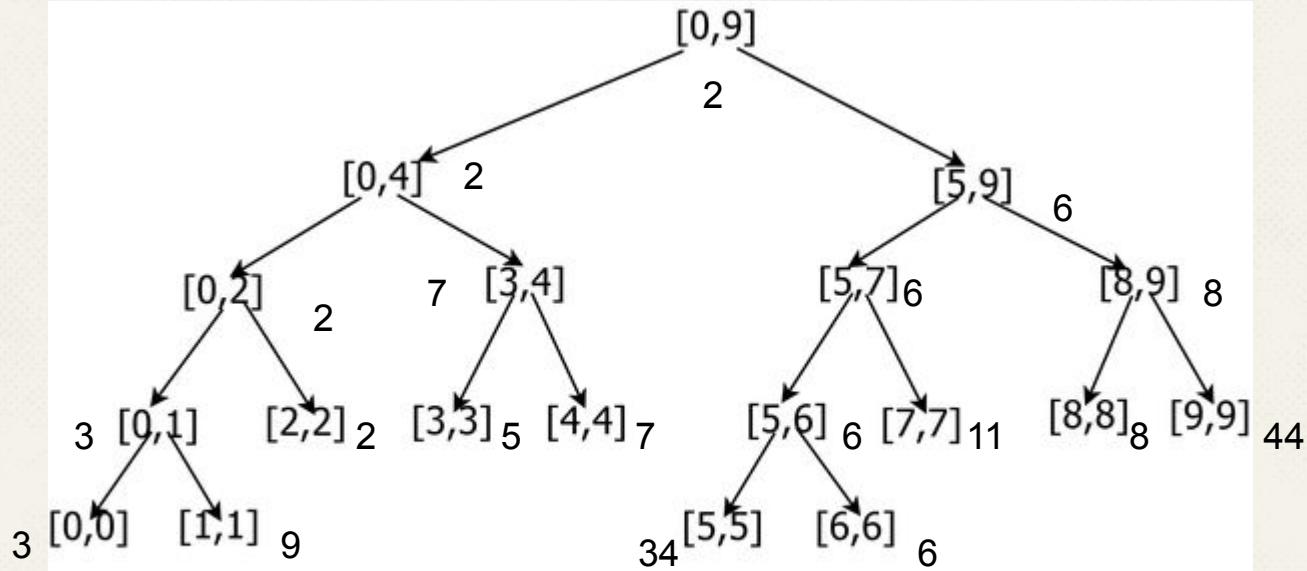
Tinem minimul!



# Arbore de Intervale

0	1	2	3	4	5	6	7	8	9
3	9	2	<b>5</b>	<b>7</b>	<b>34</b>	<b>6</b>	11	8	44

Cum îl  
implementăm?

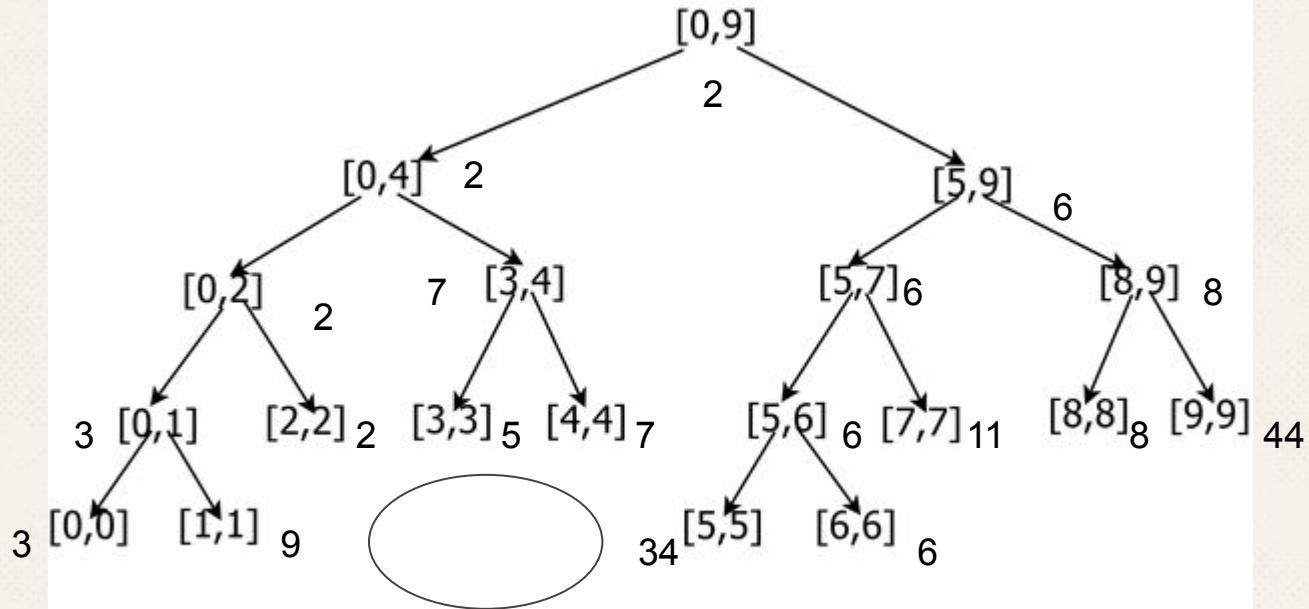


# Arbore de Intervale

0	1	2	3	4	5	6	7	8	9
3	9	2	<b>5</b>	<b>7</b>	<b>34</b>	<b>6</b>	11	8	44

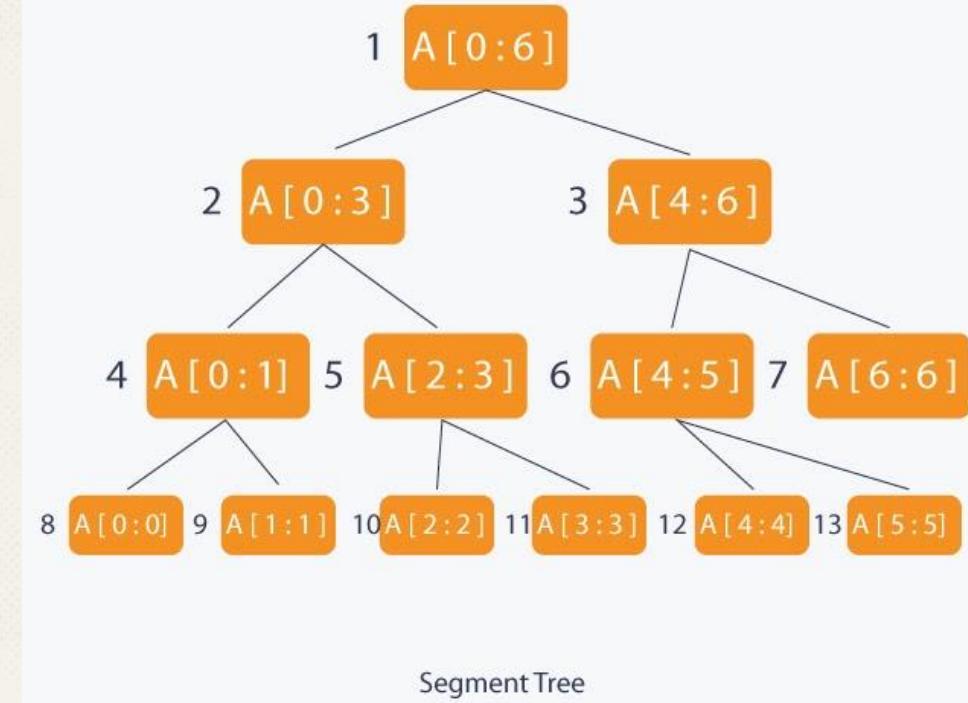
Cum îl implementăm?

- Arbore like
- **Vector!**



# Arbore de Intervale

```
tree [1] = A[0:6]
tree [2] = A[0:3]
tree [3] = A[4:6]
tree [4] = A[0:1]
tree [5] = A[2:3]
tree [6] = A[4:5]
tree [7] = A[6:6]
tree [8] = A[0:0]
tree [9] = A[1:1]
tree [10] = A[2:2]
tree [11] = A[3:3]
tree [12] = A[4:4]
tree [13] = A[5:5]
```



Segment Tree represented as linear array

# Arbore de Intervale

Reprezentare similară cu heapul:

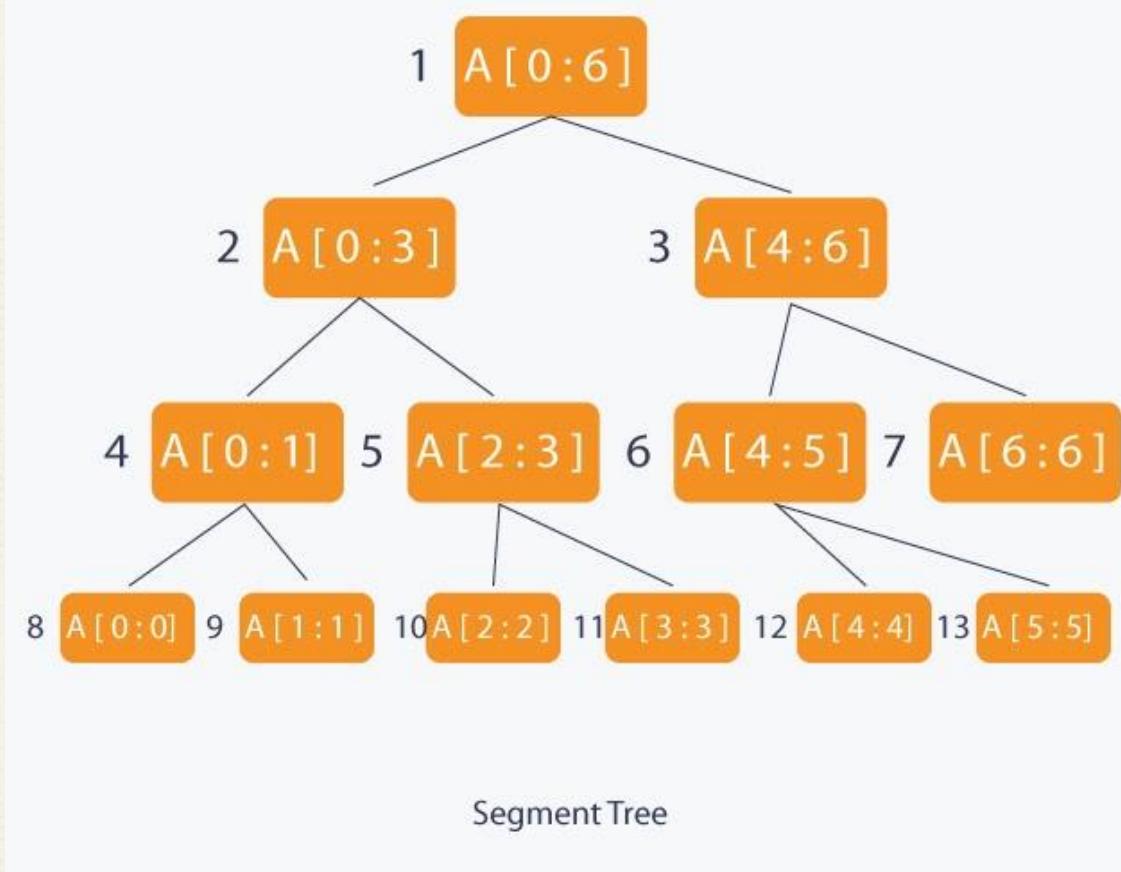
- Rădăcina (1 de multe ori) are intervalul  $[0,n)$   $[L,R)$ 
  - Fiul stâng are  $[L, (L+R)/2]$ ; el are poziția în vector  $i^*2$
  - Fiul drept are  $[(L+R)/2 + 1, R]$ ; el are poziția în vector  $i^*2+1$
  - Vectorul poate avea niște elemente lipsă pe ultimul rând (vezi 2 slide-uri mai sus).

În total vectorul are  $2^*n$  noduri “active”, dar avem nevoie de mai mult de  $2^*n$  memorie.  
 $4^*n$  e safe

**O(n)** memorie.

# Operații

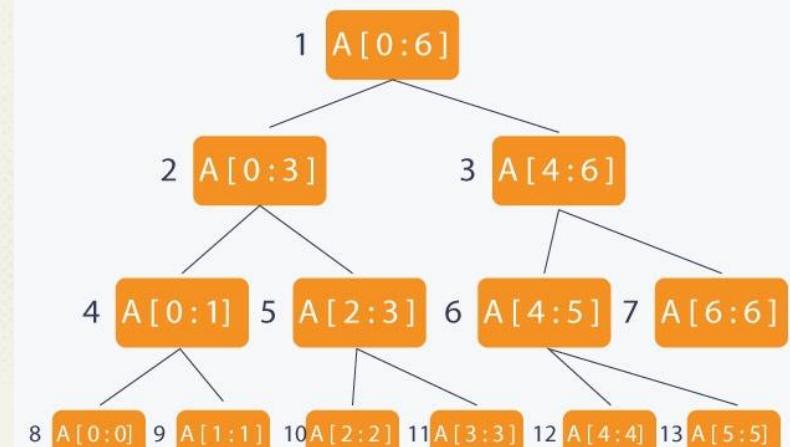
- Query pe index
- Query pe interval
  - Min
  - Sum
- Modificare element
- Modificare interval



# Operații

- Query pe index
  - Ori avem “pointeri” spre frunze și răspund direct
  - Ori pornim top down

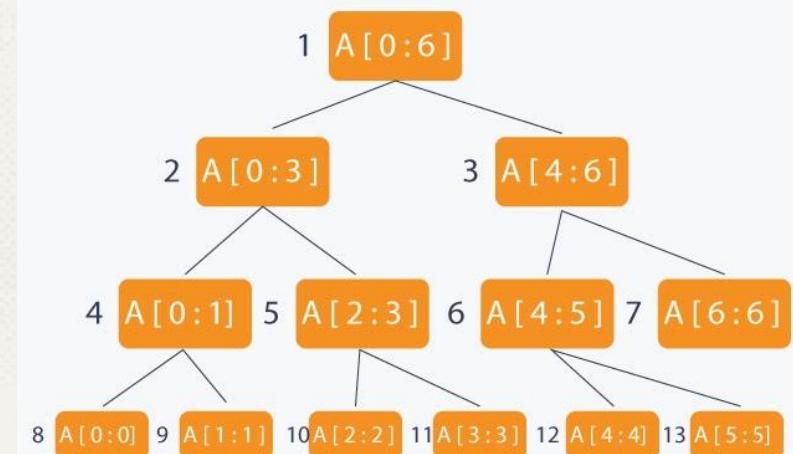
```
getValue(vector<int> arb_int, int index, int n) {  
    int L = 0, R = n, poz = 1;  
    while (L != R) {  
        if (index > (L + R)/2) {  
            L = (L + R)/2, poz = poz*2 + 1;  
        }  
        else {  
            R = (L+R)/2, poz *=2;  
        }  
    }  
    return arb_int[poz]; // L = R;  
}
```



Segment Tree

# Operații

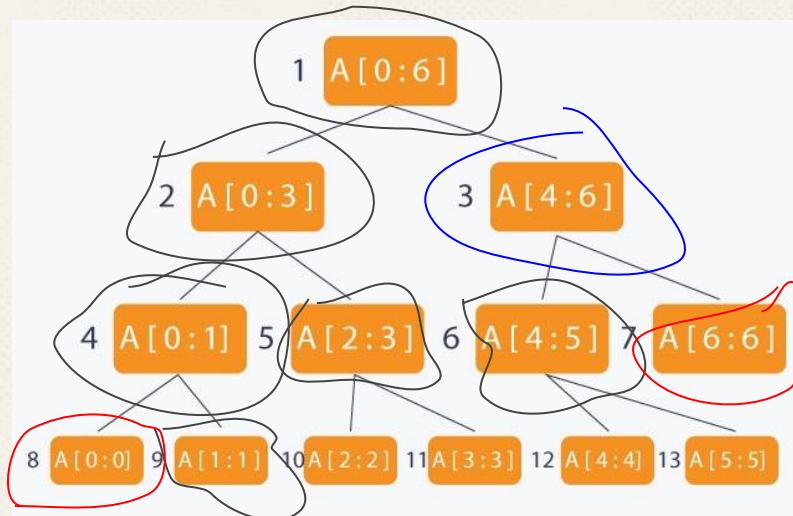
- Query pe interval
  - Evident, nu luăm toate valorile; ar putea fi liniar
  - $Q(1,5)$  min



Segment Tree

# Operații

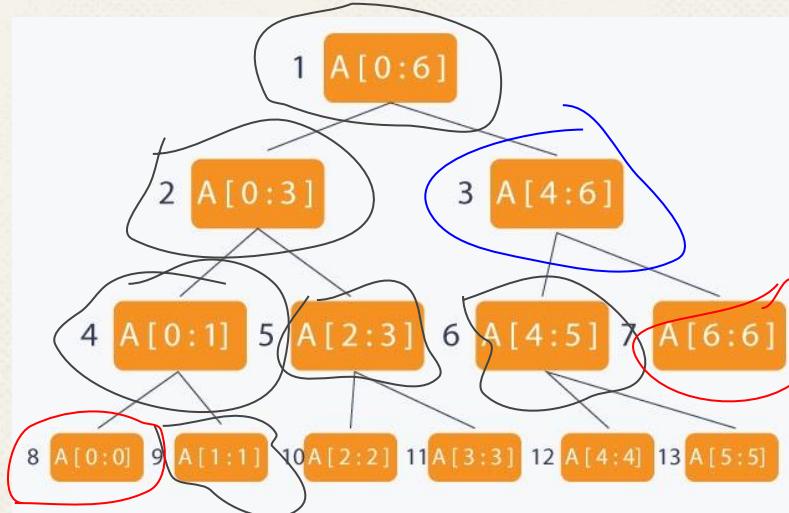
- Query pe interval
  - Evident, nu luăm toate valorile; ar putea fi liniar
  - $Q(1,5)$  min
  - Pornim din rădăcina și mergem recursiv și L și R
  - Dacă intervalul nodului nu se intersectează, oprim
  - **Dacă intervalul e inclus complet, luăm info & ne opri**
  - Câte noduri putem parcurge?



Segment Tree

# Operații

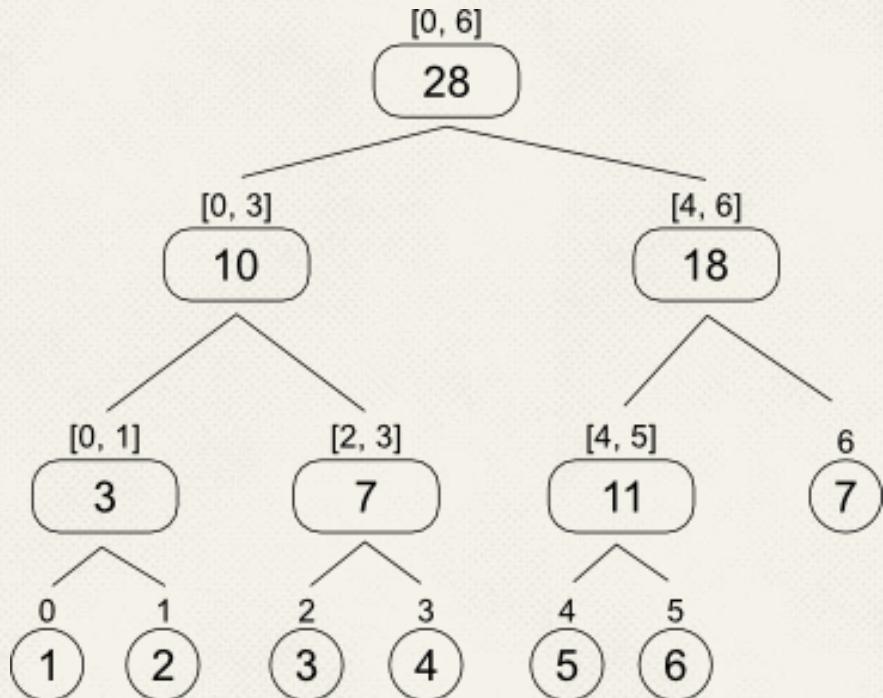
- Query pe interval
  - Evident, nu luăm toate valorile; ar putea fi liniar
  - $Q(1,5)$  min
  - Pornim din rădăcina și mergem recursiv și L și R
- Caz I    □ Dacă intervalul nodului nu se intersectează, oprim
- Caz II    □ **Dacă intervalul e inclus complet, luăm info & ne oprim**
- Câte noduri putem parcurge?
  - Doar  $4^* \log n$
  - Coborâm pe o ramură până facem un split
    - După split, în fiecare parte, unul dintre ei va fi ori cazul I, ori cazul II, deci se va cobori pe maxim 2 drumuri până jos.



Segment Tree

# Operații

- Query pe index
- Query pe interval
- Sum (1,5)
  - ?

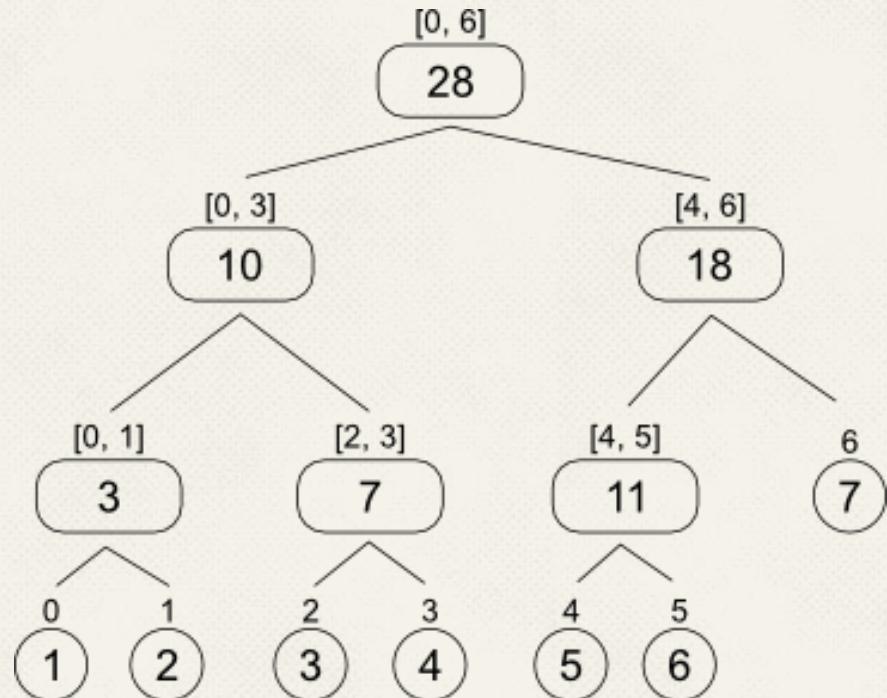


# Operații

- Modificare element
  - Dacă țin suma, pot face top-down
  - Dacă țin minim, pot face ori:
    - Top down up
      - coborâm din rădăcină până găsim frunza pe care o modificăm
      - La urcare, facem update tata = min(cei 2 fi)
    - Bottom up
      - Exact ca mai sus, dar avem deja indexul ținut
  - Înapoi la sortare (<https://leetcode.com/problems/sort-an-array/submissions/>)

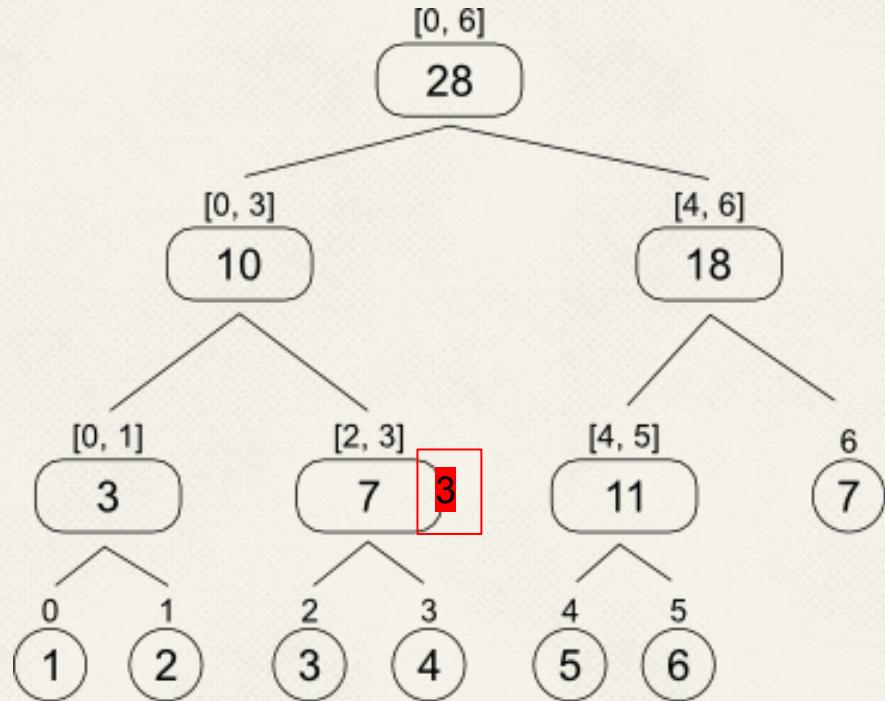
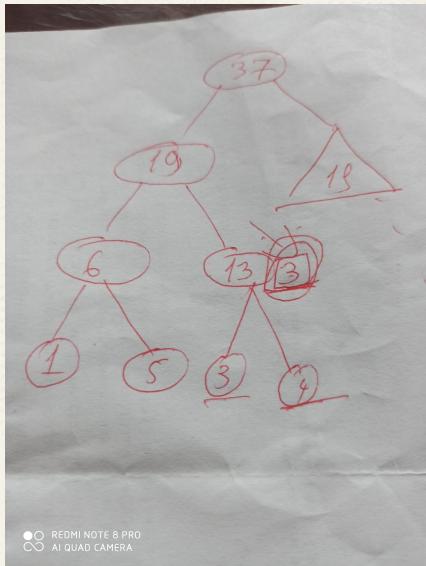
# Operații

- Modificare pe interval
  - Similar cu query pe interval
  - Merg recursiv în ambii ffi
    - Mă opresc dacă nu am intersecție
    - Modific doar nodul actual dacă este inclus de tot în interval
      - Aici trebuie să ținem în nod o informație suplimentară (toate nodurile cresc cu o anumită valoare)
    - Cobor dacă e intersecție parțială



# Operații

- Modificare pe interval
  - Add(3, 1, 3) (adaugă 3 la fiecare element din intervalul 1, 3)
  - O mică atenție la query-uri



# RMQ, LCA, LA



# Definirea problemelor

**Range Minimum Query (RMQ):**

Se dă un vector. Răspundeți cât mai eficient la întrebări de genul: **Care este cel mai mic element din intervalul i, j?**

0	1	2	3	4	5	6	7	8	9
3	9	2	8	5	3	8	7	6	11

<https://www.infoarena.ro/problema/rmq>

0 3 → 2

5 9 → 3

# LCA

**Lowest Common Ancestor (LCA):**

Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dau două noduri într-un arbore. Găsiți cel mai apropiat strămoș comun.**

(<https://www.infoarena.ro/problema/lca>)

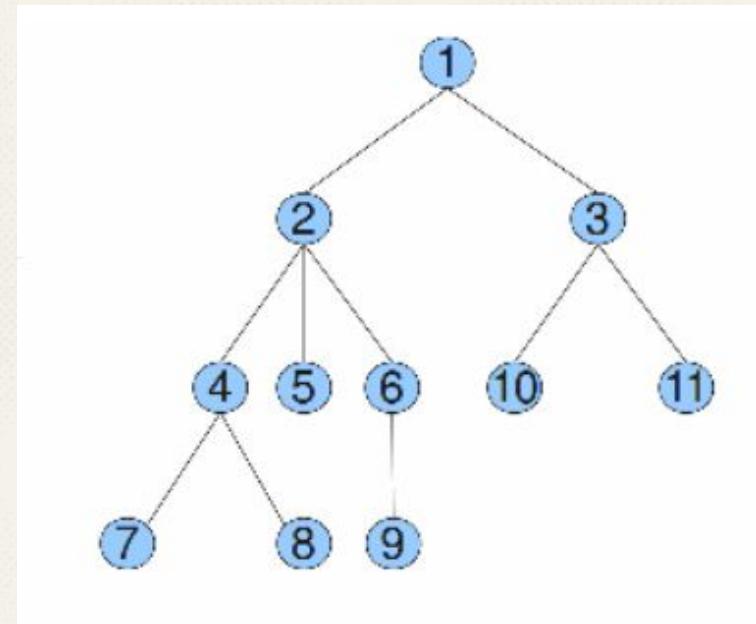
4 9 → 2

4 11 → 1

7 6 → 2

8 9 → 2

8 4 → 4



# Lowest Ancestor

Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dă un nod și un întreg k. Care este strămoșul de nivel k al nodului dat?**

<https://www.infoarena.ro/problema/stramosi> (adăugată cu 1 punct la temă)

2 1 → 1

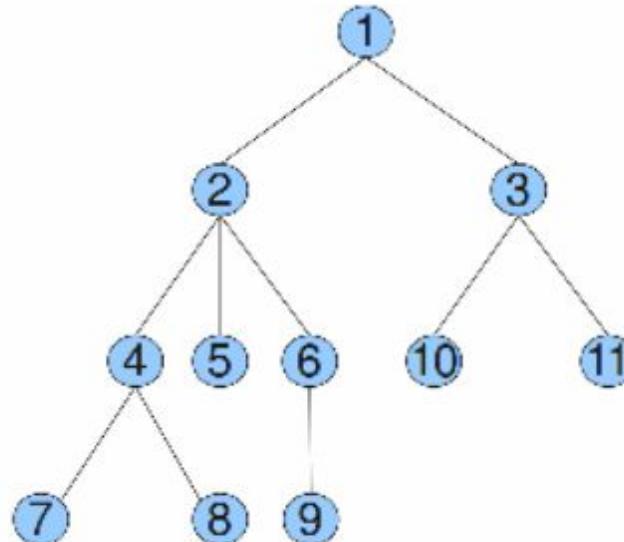
9 1 → 6

9 2 -> 2

9 3 -> 1

6 4 → -1

10 1 → 3



# Lowest Ancestor - soluții

Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dă un nod și un întreg k. Care este strămoșul de nivel k al nodului dat?**

2 1 → 1      9 1 → 6

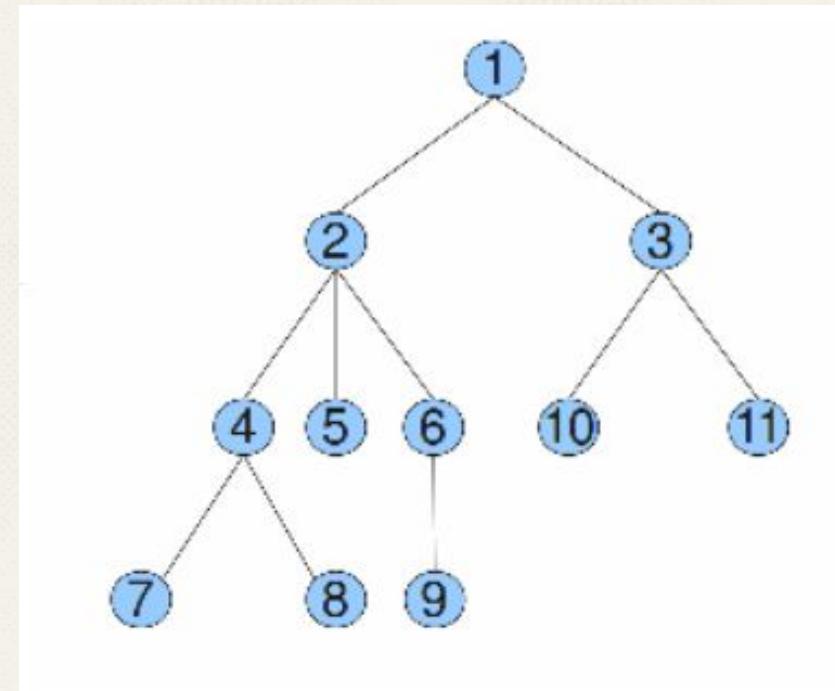
Cum facem?

Putem răspunde în  $O(h)$ , parcurgând din tata în tata la fiecare querry.

Sau putem răspunde în  $O(1)$ , dacă pentru fiecare nod reținem

$D[i][j] = \text{strămoșul de nivel } j \text{ a lui } i$

$D[9] = \{9, 6, 2, 1\}$



# Lowest Ancestor - soluții

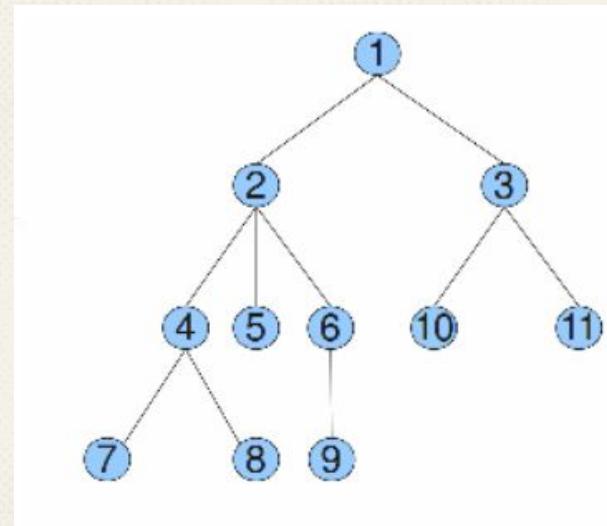
Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dă un nod și un întreg k. Care este strămoșul de nivel k al nodului dat?**

2 1 → 1      9 1 → 6

$D[i][j]$  = strămoșul de nivel  $j$  a lui  $i$

$D[9] = \{9, 6, 2, 1\}$

Memorie și procesare  $O(n*h)$  și răspuns  $O(1)$ .



# Lowest Ancestor - soluții

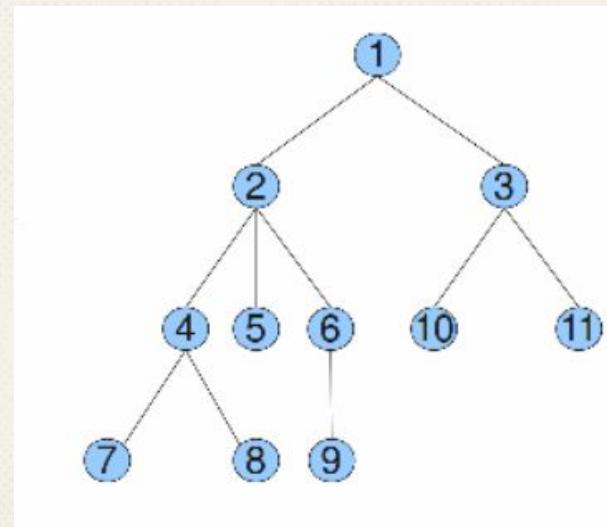
Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dă un nod și un întreg k. Care este strămoșul de nivel k al nodului dat?**

Sau pot folosi sqrt decomposition:

Țin tatăl de ordin radical din n.

Dacă radical din n este 100 și eu țin din 100 în 100:

Tatăl 300 este tata100[tata100[tata100[x]]];



# Lowest Ancestor - soluții

Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dă un nod și un întreg k. Care este strămoșul de nivel k al nodului dat?**

2 1 → 1      9 1 → 6

Tin tatăl de ordin radical din n.

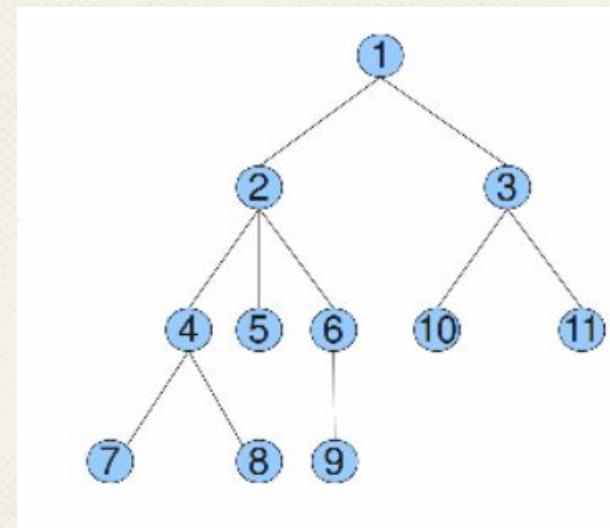
Dacă radical din n este 100 și eu tin din 100 în 100:

Tatăl 301 este

```
tata[tata100[tata100[tata100[x]]]];
```

Soluție cu  $O(n)$  memorie suplimentară,

$O(1)$  pe nod și  $O(\sqrt{n})$  pe query.



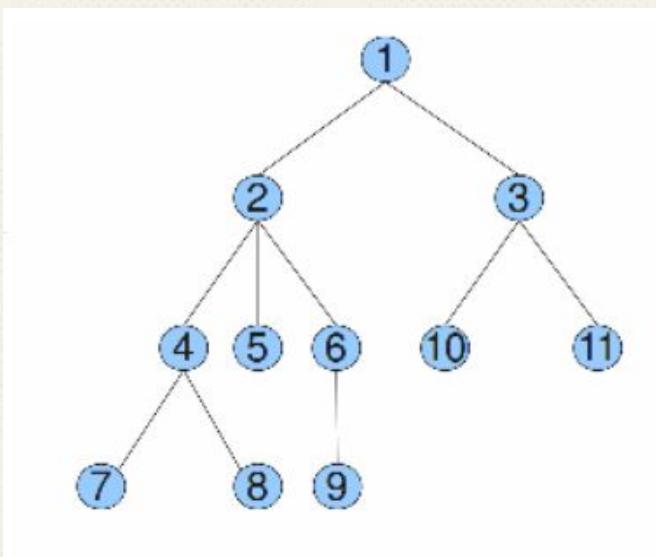
# Lowest Ancestor - soluții

Se dă un arbore. Răspundeți cât mai eficient la întrebări de genul: **Se dă un nod și un întreg k. Care este strămoșul de nivel k al nodului dat?**

2 1 → 1      9 1 → 6

Cum facem ?

- O(n) query, O(1) memorie
- O(sqrt n) query și O(n) memorie (Balog)
- **O(log n) query și O(n log n) memorie**



# Lowest Ancestor

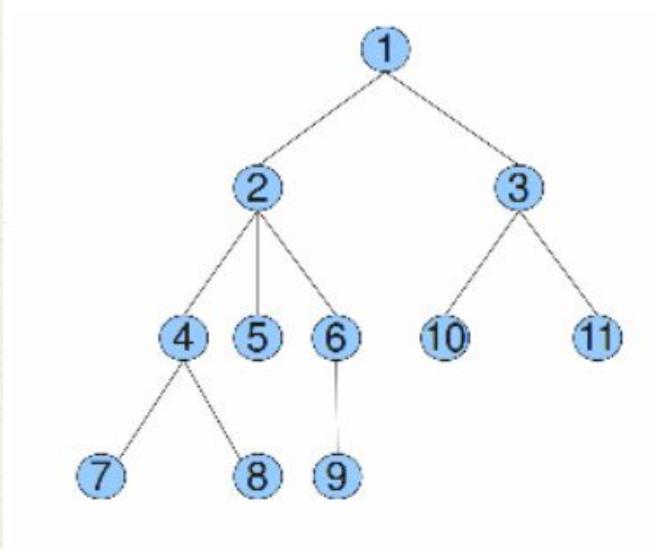
$O(\log n)$  query și  $O(n \log n)$  memorie

Pentru fiecare nod, țin tații de înălțime 1, 2, 4, 8, 16...

Pentru 7 → 4, 2, -1, -1 ....

Pentru 6 → 2, 1, -1, -1 ....

Cum calculăm vectorul de tați?



# Lowest Ancestor

$O(\log n)$  query și  $O(n \log n)$  memorie

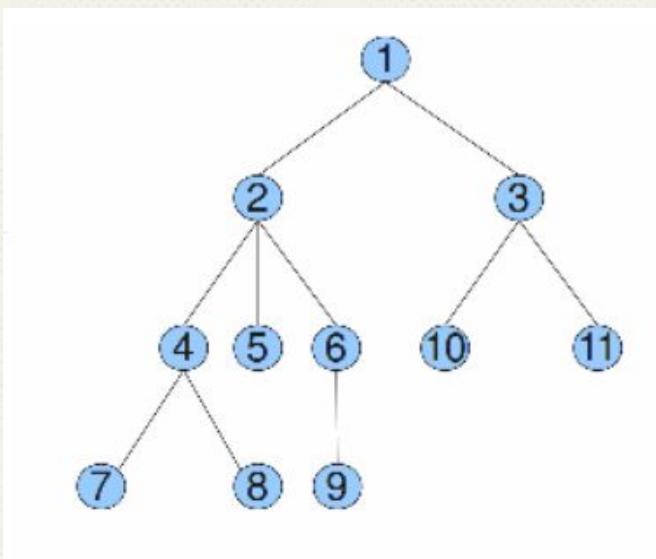
Pentru fiecare nod, țin tații de înălțime 1, 2, 4, 8, 16...

Pentru 7 → 4, 2, -1, -1 ....

Pentru 6 → 2, 1, -1, -1 ....

Cum calculăm vectorul de tați?

```
for (int i = 1; i < log n; ++i) {  
    for (int j = 1; j < n; ++j)  
        tata[j][i] = tata[tata[j][i-1]][i-1];  
}
```



# Lowest Ancestor

$O(\log n)$  query și  $O(n \log n)$  memorie

Pentru fiecare nod, țin tații de înălțime 1, 2, 4, 8, 16...

Pentru 7 → 4, 2, -1, -1 ....

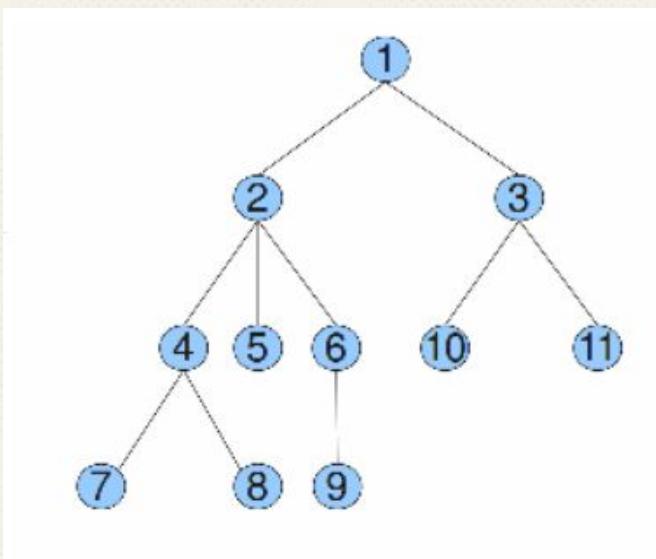
Pentru 6 → 2, 1, -1, -1 ....

Cum calculăm al k-lea strămoș?

- Similar cu căutarea binară discutată la curs
- Sărim cu puterea lui 2 cea mai mare

7 3 → 7 sărim 2 pași până la 2

Apoi 2 1 → sărim 1 pas → 1



# Lowest Ancestor

$O(\log n)$  query și  $O(n \log n)$  memorie

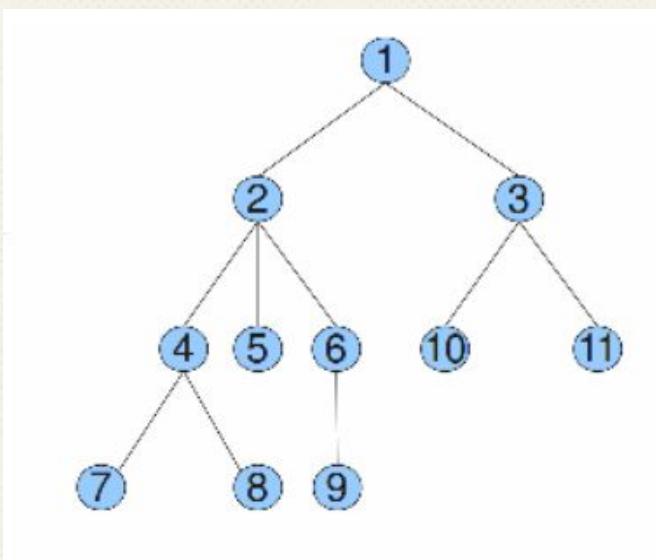
Pentru fiecare nod, țin tații de înălțime 1, 2, 4, 8, 16...

Pentru 7 → 4, 2, -1, -1 ....

Pentru 6 → 2, 1, -1, -1 ....

Cum calculăm al k-lea strămoș?

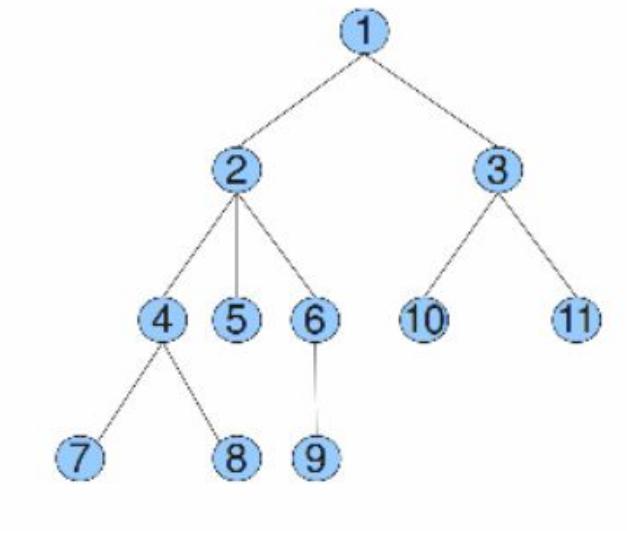
$\text{tata}(x, 14) = \text{tata}(\text{tata}8[x], 6) = \text{tata}(\text{tata}4[\text{tata}8[x]], 2)$   
 $= \text{tata}2[\text{tata}4[\text{tata}8[x]]]$



# Lowest Ancestor

$O(\log n)$  query și  $O(n \log n)$  memorie

Complexitate ?

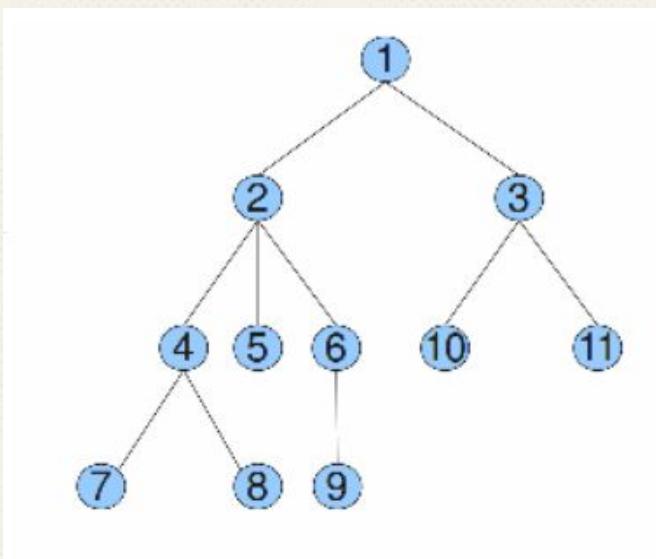


# Lowest Ancestor

$O(\log n)$  query și  $O(n \log n)$  memorie

Complexitate

- $O(n \log n)$  preprocesoare
- $O(n \log n)$  memorie suplimentară
- $O(\log n)$  pe query
- Se poate obține  $O(n)$  memorie suplimentară
  - (vezi cursul de la [MIT](#))



# Range Minimum Query

## Soluții

**Range Minimum Query (RMQ):**

Se dă un vector. Răspundeți cât mai eficient la întrebări de genul: **Care este cel mai mic element din intervalul  $i,j$ ?**

0	1	2	3	4	5	6	7	8	9
3	9	2	8	5	3	8	7	6	11

Soluții ?

- $O(n)$  pe query
- Șmenul lui Batog -  $O(\sqrt{n})$  pe query
- **Ținem pentru fiecare element puterile lui 2 și răspundem similar LA în  $\log n$ .**

# Range Minimum Query

## Soluții

- Ținem pentru fiecare element puterile lui 2 și răspundem similar LA în  $\log n$ .

	0	1	2	3	4	5	6	7	8	9
min										
min2	3	2	2	5	3	3	7	6	6	11
min4	2	2	2	3	3	3	6	6	6	11
min8	2	2	2	3	3	3	6	6	6	11

DE SCHIMBAT EXEMPLUL sa nu fie crescator!

# Range Minimum Query

## Soluții

- Ținem pentru fiecare element puterile lui 2 și răspundem similar LA în  $\log n$ .

	0	1	2	3	4	5	6	7	8	9
min	3	9	2	8	5	3	8	7	6	11
min2	3	2	2	5	3	3	7	6	6	11
min4	2	2	2	3	3	3	6	6	6	11
min8	2	2	2	3	3	3	6	6	6	11

- Query în  $\log(n)$ 
  - 1 6 min4(1) 1-4 + min2(5) 5-6
  - 2 9 min8(2)
  - 3 6 min4(3) ->alte exemple si aici

# Problemă adițională

Se dă un nr  $n \leq 10^9$ . Cum calculez  $\log n$  în  $O(1)$  ?



# Problemă adițională

Se dă un nr n  $\leq 10^9$ . Cum calculez logn în O(1) ?

- Pot ține, pentru fiecare număr de la 1 la 256, care e cel mai semnificativ bit
  - 14 → 8
  - 230 → 128
  - ....
- Pentru un număr pe 32 de biți, găsesc primul byte > 0 și aplic ce am calculat mai sus
- Pot ține rezultatul pt 2 bytes și atunci am nevoie de doar 2 operații

# Range Minimum Query

## Soluții

- Ținem pentru fiecare element puterile lui 2 și răspundem în O(1)

	0	1	2	3	4	5	6	7	8	9
min	3	9	2	8	5	3	8	7	6	11
min2	3	2	2	5	3	3	7	6	6	11
min4	2	2	2	3	3	3	6	6	6	11
min8	2	2	2	3	3	3	6	6	6	11

- Query în O(1)? Cum?
  - 1 6 → min(min(1,4), min(3,6)) - prin urmare, putem face 2 query-uri [a, a + log(b-a)], [b - log(b-a) + 1, b].
  - 20, 1000 → min [Q(20, 531), Q(489, 1000)] → 2 query-uri de mărime 512

# Range Minimum Query

## Soluții

- **Ținem pentru fiecare element puterile lui 2 și răspundem în O(1)**
- Query în **O(1)**? Cum?
  - $1 \rightarrow \min(\min(1,4), \min(3,6))$  - prin urmare, putem face 2 query-uri  $[a, a + \log(b-a)]$ ,  $[b - \log(b-a) + 1, b]$ .
  - $20, 1000 \rightarrow \min [Q(20, 531), Q(489, 1000)] \rightarrow$  2 query-uri de mărime 512
  - **Atenție! Ideea funcționează doar pentru minim**, nu și pentru sumă, deoarece o parte din interval  $(489, 531)$  este inclus în ambele query-uri. Dacă vrem să calculăm minimul, acest lucru nu este o problemă, dar pentru sume da!
  - Pentru sumă, trebuie să facem  $O(\log n)$  query-uri, deci probabil arborii de intervale sunt mai buni, deoarece au tot  $O(\log n)$  pe query, dar au  $O(n)$  memorie suplimentară și  $O(n)$  construcție.

# Range Minimum Query

## Soluții

- Complexitate **O(n log n)** memorie și procesare și **O(1)** query
  - Se poate obține O(n) procesare și memorie suplimentară și O(1) pe query.
    - [Link](#)
    - Implementare
      - RMQ pe Infoarena: <https://pastebin.com/7a8uVdtP>
      - <https://leetcode.com/problems/range-sum-query-immutable/>
        - am realizat la un seminar că problema nu cerea minim, prin urmare nu se putea rezolva în O(1) pe query. Vă dau două rezolvări diferite
        - cu Batog: <https://pastebin.com/5RUrVpVi>
        - Totuși, problema se rezolvă cu sume partiale în O(1) pe query

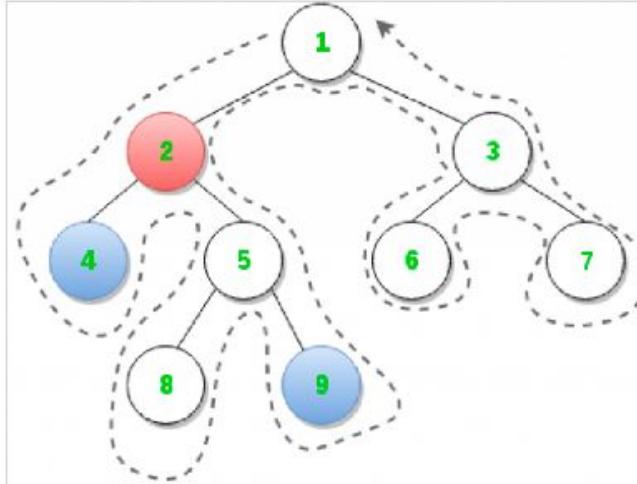
# LCA → RMQ

Problema LCA se poate reduce la RMQ

- Descriere pe larg
- Principiul este o liniarizare a arborelui

# LCA → RMQ

- Începem o parcurgere RSD din rădăcină și scriem fiecare nod **de fiecare dată când trecem prin el.**
- Pentru fiecare nod, reținem și distanța de la el la rădăcină.



Euler Tour

An euler tour of the tree starting from node 1 will yield:

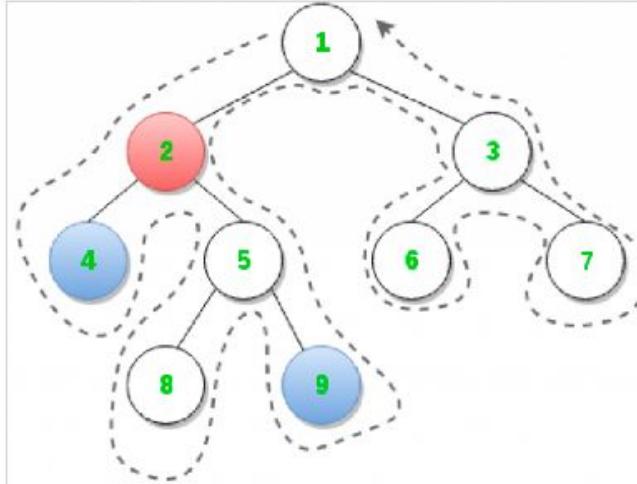
1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# LCA → RMQ

- Începem o parcurgere RSD din rădăcină și scriem fiecare nod **de fiecare dată când trecem prin el.**
- Pentru fiecare nod, reținem și distanța de la el la rădăcină
- Pentru fiecare nod, mai reținem și prima sa apariție în parcurgerea Euler...
- De exemplu, pentru 4 e poziția 2, pentru 9 este 7



Euler Tour

An euler tour of the tree starting from node 1 will yield:

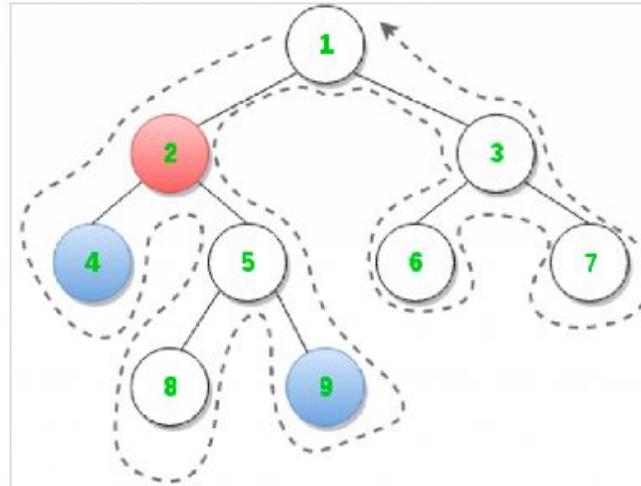
1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# LCA → RMQ

- LCA( $i, j$ ) este RMQ(first[i], first[j])...
- LCA(4,9) va fi RMQ pe parcurgerea Euler între primele apariții ale lui 4 și 9  
Deci RMQ(2,7)...
- RMQ se va face pe vectorul de distanțe, până la rădăcină (2, 7), prin urmare obținem distanța 1 către rădăcina care corespunde nodului 2.
- Orice drum între 4 și 9 trece prin 2, dar nu mai sus de 2!



An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# TRIE



---

Kahoot

---

# Discuții Examen

- Ca de obicei, mă găsiți online, unde răspund la întrebări
- Poate facem un Q&A ? Cam cand ati vrea ? Nu promit nimic inca... Incercam pe 8
- Puteți pune întrebări aici și eventual eu voi răspunde la ele

# Subiecte Examen

- Aveți voie cu materiale scrise (imi pare rau pt natura)! Mobilele pe catedra! Daca aveti mobil la voi dupa ce incepe examenul -> frauda -> restanta & posibila exmatriculare!
- Doua parti:
  - Exemplificare cum funcționează structuri de date/algoritmi
  - Probleme ca la seminar
  - Pauza 15 minute între ele

# Subiecte Examen

- Exemplificare cum funcționează structuri de date/algoritmi & Complexitate
  - Count Sort, radix sort, quick sort, merge sort
  - Cozi, Stive, Deque
  - Hashuri
    - Inserare/Cautare/Stergere
    - Tratarea coliziunilor: Inlantuire/Adresare Directa
    - Functii de dispersie: metoda diviziunii/metoda multiplicarii
    - Rabin Karp
  - Heapuri, Heapuri Binomiale, Heapuri Fibonacci

# Subiecte Examen

- Exemplificare cum funcționează structuri de date/algoritmi & Complexitate
  - Arbori binari de cautare
    - Inserare, Stergere, Cautare, Succesor, Predesor, k-th element
    - Parcugeri Preordine, Inordine, Postordine
  - Arbori binari de cautare echilibrati:
    - inserare, stergere, cautare, succesor, k-lea cel mai mare...
    - La alegere din
      - AVL/Red Black/Skip lists/B-arbori/Treaps
  - Arbori de Intervale/ Batog
    - Inserare/Cautare min/Stegere/Sortare/ Calcularea suma pe interval/Update pe interval
  - RMQ&LCA&LA
    - ce rezolva, cum functioneaza pe un exemplu, complexitate..
  - Trie

# Subiecte Examen

- Desenati un arbore binar complet de inaltime 2
- Desenati un heap cu 5 noduri
- Desenati un arbore binar de cautare cu 6 noduri. Ce inalimi poate sa aiba?
- Inserati pe rand intr-un heap fibonacii de minim valorile 1, 2, 9, 5, 7, 3
- Cum folosim un arbore de intervale sa sortam un vector?
- Se da un arbore, care este LA intre 3 si 9, dar 2 si 8, cum se calculeaza ? Ce complexitate are ?
- Cum gasim succesorul intr-un arbore binar de cautare
- Construiti un TRIE cu cuvintele : ala, bala, portocala
- Bonus:
  - Demonstrati ca orice algoritm care construieste un arbore binar de cautare cu n numere ruleaza in timp  $\Omega(n \log n)$ .

# Subiecte Examen

- Probleme ca la seminar
  - 3-4 probleme in o ora (o sa fac un mic test la ultimul seminar)
    - O sa va dau la fiecare grupa o problema in 15-20 minute la seminarul 7 si o sa incerc sa va dau feedback ulterior... Daca o rezolvati bine -> puncte activitate...
  - Va trebui sa scrieti cum o rezolvati si ce complexitate are solutia voastră:
  - Gen: Se dau n numere, cate perechi de numere au suma un patrat perfect
    -
  -

# Subiecte Examen

- Gen: Se dau n numere, cate perechi de numere au suma un patrat perfect
  - Iau toate numerele de la 1 la max le calculez patratul, apoi iau toate perechile de la 1 la n si le fac suma si vad daca da fix patratul la care sunt
    - Nota 2,
    - Daca adaug si complexitate corect ??
      - $O(n^2 * \text{max})$  -> nota 3
  - Iau toate numere de la 1 la  $\sqrt{\text{max}} * 2$  ..... si la fel -> nota 3, respectiv 4
    - $O(n^2 * \sqrt{\text{max}})$
  - Iau toate perechile de numere le fac suma si vad daca rezultatul e un patrat perfect in O(1) (gen  $\sqrt{x} * \sqrt{x} == x$ )... -> 5 cu complexitate  $O(n^2)$  7
  - Iau toate numerele si toate patratele  $\leq \text{max1} + \text{max2}$  si vad daca Patrat-nr exista intre numerele mele cu hashuri  $O(n * \sqrt{x})$  -> 5 cu complexitate  $O(n * \sqrt{\text{max}})$  7
  - Impreuna 10...

# Trie

- Am mai multe cuvinte pe care le tin minte și apoi am întrebări de genul:
  - este cuvântul dat in aceea lista sau nu?
- Cum putem rezolva?
  - Hash-uri!
  - Cât mă costă un query?
    - $O(l)$ , unde  $l$  e lungimea cuvântului
  - Câtă memorie mă costă să rețin hash-ul?
    - $O(n*l)$
  - Ce credeți că am putea optimiza?
    - Memoria (poate)
    - Timpul pentru query-uri nereușite ... oarecum

# Trie

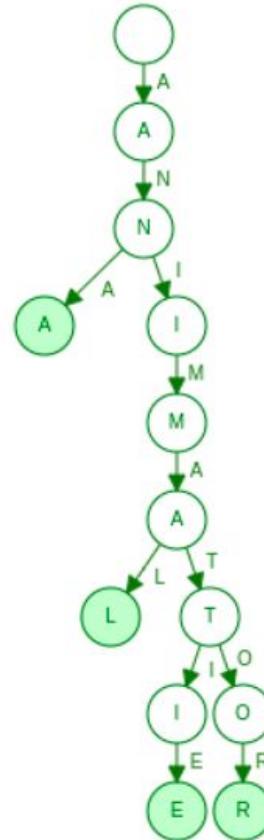
- Am mai multe cuvinte și apoi am întrebări de genul:
  - este cuvântul în dicționar sau nu?
  - care este cel mai lung prefix al cuvântului în dicționar?
- Mai merge cu hash-uri ?
  - Nu prea ...
- Alte soluții?
  - Sortăm toate cuvintele lexicografic și apoi căutăm binar
  - Îinem toate cuvintele într-un arbore binar de căutare echilibrat
- Ambele soluții au  $O(n^*l)$  memorie și  $O(\log n^*l)$  complexitate pe search
- Arboarele binar permite, totuși, și inserări și ștergeri!!

# Trie

- Dacă avem cuvintele **anima**, **animal**, **animație**, **animator**, **animare**, reținem, pentru fiecare, prefixul **anim** comun
- Cum credeți că putem îmbunătăți memoria folosită?
  - Am putea, când le ținem sortate, să le ținem ceva de genul
    - anima
    - 51
    - 5tie
  - Adică, să ținem lungimea prefixului față de elementul anterior
  - Putem duce o idee similară și spre arbori binari de căutare, dar să nu ne mai complicăm :)

# Trie

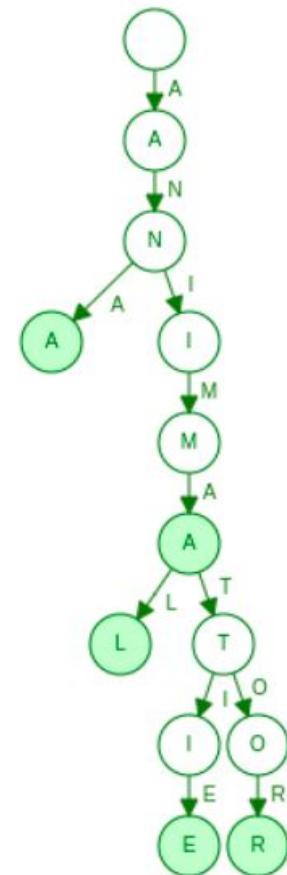
Trie cu cuvintele **ana, animator, animație, animal**



# Trie

Trie cu cuvintele **ana, animator, animație, animal, anima**

vizualizare trie



# Trie - Memorare

- Cum îl reținem?
  - Fiecare nod are un vector cu 26 de vecini, una pentru fiecare literă (sau mărimea alfabetului)
  - Ce facem dacă alfabetul e mare?
  - Fiecare nod ține un hash\_map care pentru fiecare literă tine pointerul către nodul cu acea literă

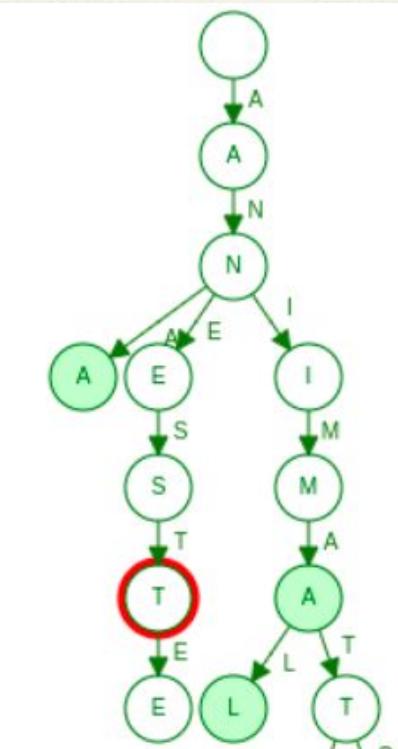
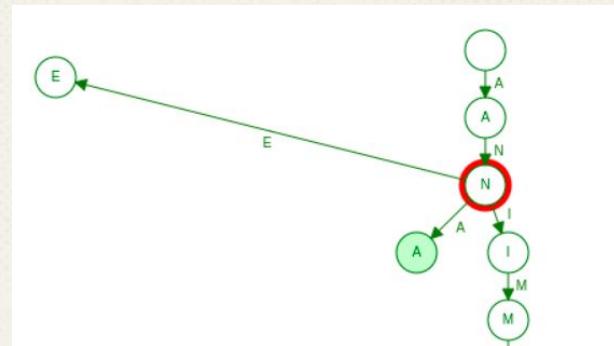
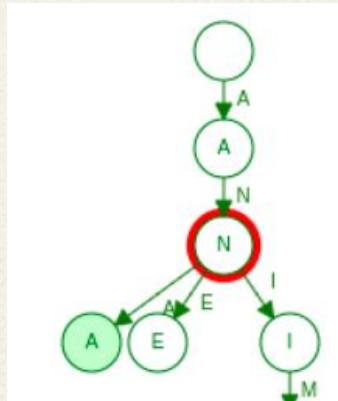
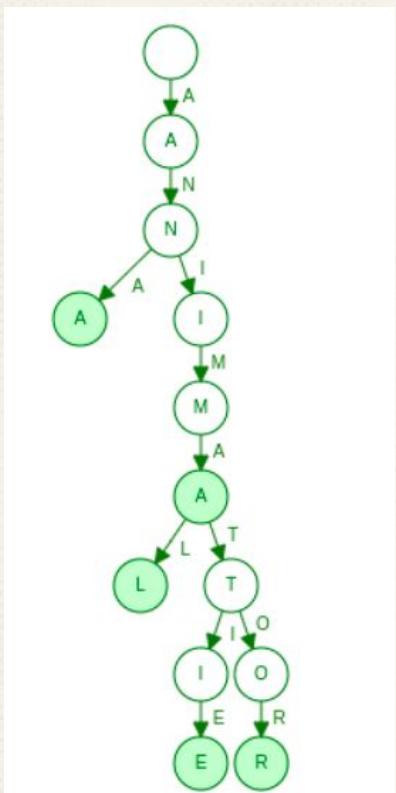
# Trie - Inserare

Pornim din rădăcină și, la fiecare literă, mergem în nodul corespunzător literei, eventual creăm acel nod

<https://www.cs.usfca.edu/~galles/visualization/Trie.html>

# Trie - Inserare

Inserăm anestezie



# Trie - Inserare

Complexitate:  $O(l)$

# Trie - Căutare

Pornim din rădăcină și mergem, la fiecare pas, pe litera corespunzătoare

Complexitate  $O(l)$  pentru căutare reușită

În practică, mai rapid pentru căutare nereușită

Căutare prefix maxim:

- Căutăm elementul până nu găsim nod corespunzător acelei litere

**Succes în sesiune :)**