

Resurse examen SD

Seria 15

June 26, 2025

Contents

1	Sortări	3
1.1	Clasificare	3
1.2	Stabilitate	3
1.3	Tabel de complexități	3
1.4	Informații generale	3
1.5	Counting sort	4
1.6	Bucket sort	4
1.7	Radix sort	4
1.8	Quicksort	4
1.9	Mergesort	4
2	Complexități	4
3	Hash-uri și Tabele de Dispersie	5
3.1	Definiții și noțiuni introductive	5
3.2	Tabelă cu adresare directă	5
3.3	Tabele de dispersie cu coliziuni	5
3.3.1	Chaining	5
3.3.2	Adresare deschisă	6
3.4	Dispersie universală	6
3.5	Algoritmul Rabin–Karp pentru pattern matching	6
3.6	Complexități	6
4	Structuri de date elementare	7
4.1	Vectori și liste	7
4.2	Stive	7
4.3	Cozi	7
4.4	Deque	7
5	Heapuri	8
5.1	Heapuri Binomiale	11
5.2	Heapuri Binomiale	11
5.3	Heapuri Fibonnaci	12
5.4	Arbori Huffman	13
6	Arbori binari de căutare	14
6.1	Definiții arbori	14
6.2	Parcurgerea unui arbore binar	14
6.3	Arbori binari de căutare	15
6.4	Minim și maxim	15

6.5	Succesor și predecesor	15
6.6	Inserare și ștergere	16
6.7	Arbori binari cu chei egale	16
7	Arbori binari de cautare echilibrati	17
7.1	Red Black Trees	17
7.2	AVL Trees	17
8	Skip Lists	17
9	Treap	18
10	Range Minimum Query	19
10.1	Lowest Ancestor (LA)	20
10.2	Lowest Common Ancestor (LCA)	21
11	Trie (Prefix Tree)	22

1 Sortări

Algoritmii de sortare pot fi clasificați după complexitate, spațiu, stabilitate și dacă se bazează pe comparații:

1.1 Clasificare

- **Elementari:** Insertion, Selection, Bubble
- **Divide et Impera:** Merge Sort, Quick Sort
- **Heap-based:** Heap Sort
- **Non-comparative:** Counting Sort, Radix Sort, Bucket Sort

1.2 Stabilitate

Un algoritm de sortare este *stabil* dacă menține ordinea relativă a elementelor egale. Acest lucru este important când sortăm obiecte cu multiple chei.

1.3 Tabel de complexități

Algorithm	Time Complexity			Space (worst)
	Best	Average	Worst	
Quicksort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
Mergesort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log n)$	$\Theta(n(\log n)^2)$	$O((n \log n)^2)$	$O(1)$
Bucket Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n^2)$	$O(n + k)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n + k)$
Counting Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$	$O(n + k)$
Cubesort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

Table 1: Array Sorting Algorithms: time and space complexities

1.4 Informații generale

- Sortările prin comparație au limită inferioară $\Omega(n \log n)$.
- Algoritmii non-comparativi (Counting, Radix) pot ajunge la $O(n)$ în condiții favorabile.
- Alegerea pivotului în Quick Sort influențează performanța:
 - pivot ales random
 - mediana din 3/5/7
 - mediana medianelor
- Pentru vectori mici, sortarea cu algoritmi $O(n^2)$ poate fi mai eficientă decât în $O(n \log n)$ (constanta poate fi mare).

1.5 Counting sort

Se creează un vector de frecvență cu valorile pe care trebuie să le sortăm. Valorile din vector sunt numărate, iar vectorul este reconstruit în ordine sortată prin parcurgerea vectorului de frecvență.

Se aplică când elementele sunt numere întregi într-un interval $[0, k]$, (până la 10^6)

1.6 Bucket sort

Se creează k buckets și fiecare element x din vectorul de intrare este distribuit în găleata indexată de o funcție de mapare (de ex. $\lfloor k \cdot \frac{x-a}{b-a} \rfloor$ pentru valori în $[a, b]$). Fiecare găleată este sortată intern (adesea prin Insertion Sort), apoi se concatenează conținutul găleților pentru a obține vectorul sortat.

1.7 Radix sort

Sortarea se face pe mai multe trepte de cifre: pentru fiecare poziție de cifră (de la LSD la MSD sau invers) se aplică Counting Sort pentru a grupa elementele după valoarea cifrei curente. După procesarea tuturor cifrelor, vectorul devine complet sortat.

1.8 Quicksort

Algoritmul Quick Sort folosește o abordare divide et impera: se selectează un pivot, iar vectorul este repartizat în două subsecțiuni, una cu elemente mai mici și cealaltă cu elemente mai mari decât pivotul. Se aplică recursiv același procedeu pe cele două subsecțiuni și, ulterior, se combină rezultatele pentru a obține vectorul sortat.

1.9 Mergesort

Merge Sort divide recursiv vectorul în jumătăți până se obțin secvențe cu un singur element (sau 2, depinde de implementare). Apoi, aceste secvențe sunt interclasate (merge) după ce au fost sortate, fiind combinate treptat pentru a forma vectorul sortat final.

2 Complexități

În studiul algoritmilor, folosim trei noțiuni fundamentale pentru a măsura creșterea funcției de timp $T(n)$ în raport cu dimensiunea de intrare n :

- **O** (Big-O): reprezintă o limită superioară asimptotică.

$$T(n) = O(f(n)) \iff \exists c > 0, n_0 : \forall n \geq n_0, T(n) \leq c f(n).$$

- **Ω** (Big-Omega): reprezintă o limită inferioară asimptotică.

$$T(n) = \Omega(f(n)) \iff \exists c > 0, n_0 : \forall n \geq n_0, T(n) \geq c f(n).$$

- **Θ** (Big-Theta): când există simultan limite superioară și inferioară de același ordin.

$$T(n) = \Theta(f(n)) \iff T(n) = O(f(n)) \wedge T(n) = \Omega(f(n)).$$

Astfel, $\Theta(f(n))$ indică creșterea „exactă” (în sens asimptotic), iar O și Ω doar conturul superior, respectiv inferior.

3 Hash-uri și Tabele de Dispersie

3.1 Definiții și noțiuni introductive

Definition 3.1 (Funcție de hash). O *funcție de hash* este o funcție matematică

$$h : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\},$$

care transformă orice element din universul de chei \mathcal{U} într-o valoare de dimensiune fixă. Rezultatul $h(x)$ se numește *cod hash* al lui x . Funcția trebuie să fie eficient de calculat și să distribuie uniform cheile.

- **Hash prin diviziune:** $h(x) = x \bmod p$, unde p este un număr prim apropiat de m .
- **Hash prin multiplicare:** Hash-ul prin multiplicare standard folosește formula

$$h_a(K) = \left\lfloor \frac{(aK \bmod W)}{W/M} \right\rfloor,$$

care produce o valoare în $\{0, \dots, M-1\}$. Parametrul a se alege coprim cu W și cu o reprezentare binară „aleatorie” a biților. În cazul special când $W = 2^w$ și $M = 2^m$, formula devine

$$h_a(K) = \left\lfloor \frac{(aK \bmod 2^w)}{2^{w-m}} \right\rfloor,$$

- **Proprietate dorită:** ipoteza dispersiei uniforme simple, adică pentru orice chei distincte x, y avem

$$\Pr[h(x) = h(y)] = \frac{1}{m}.$$

3.2 Tabelă cu adresare directă

Pentru N chei dintr-un univers mic $\{1, \dots, N\}$, putem implementa direct o tablă de dispersie printr-un vector binar:

$$A[1 \dots N], \quad A[i] = \begin{cases} 1 & \text{dacă } i \text{ este prezent,} \\ 0 & \text{altfel.} \end{cases}$$

Operații:

- **insert(x):** $A[x] \leftarrow 1$, cost $O(1)$.
- **find(x):** returnează $A[x]$, cost $O(1)$.
- **delete(x):** $A[x] \leftarrow 0$, cost $O(1)$.

3.3 Tabele de dispersie cu coliziuni

Pentru universuri mai mari sau chei arbitrare, folosim o funcție de hash h și rezolvăm coliziunile prin:

1. **Listă înlănțuită (chaining).**
2. **Adresare deschisă (open addressing).**

3.3.1 Chaining

Fie $T[0 \dots m-1]$ un vector de liste înlănțuite. La inserare, adăugăm x în lista $T[h(x)]$. Căutarea parcurge lista, cost amortizat $O(1 + \alpha)$, unde $\alpha = n/m$.

3.3.2 Adresare deschisă

Coliziunile se rezolvă prin sondaj:

Lineară: $h(x, i) = (h_0(x) + i) \bmod m$.

Dublu hashing: $h(x, i) = (h_1(x) + i \cdot h_2(x)) \bmod m$.

În cel mai rău caz, operațiile costă $O(m)$, dar dacă $\alpha < 1$ și funcțiile sunt bine alese, costul mediu este $O(1)$.

3.4 Dispersie universală

Definition 3.2 (Familie universală). O familie de funcții de dispersie \mathcal{H} este *universală* dacă pentru orice $x \neq y$ din \mathcal{U} ,

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| = \frac{|\mathcal{H}|}{m}.$$

Consecință: pentru $n \leq m$ și h ales aleator din \mathcal{H} , numărul mediu de coliziuni pentru o cheie fixă este mai mic decât 1.

3.5 Algoritmul Rabin–Karp pentru pattern matching

Pentru a găsi un șablon (pattern) P de lungime m într-un text T de lungime n , folosim un hash pentru substring.

- Alegem o bază b (de obicei dimensiunea alfabetului) și un număr M .
- Calculăm hash-ul pattern-ului (substring-ului)

$$h_P = \sum_{i=0}^{m-1} P[i] b^{m-1-i} \bmod M.$$

- Rolling hash-ul fiecărei ferestre de text de lungime m :

$$H_0 = \sum_{i=0}^{m-1} T[i] b^{m-1-i} \bmod M,$$

$$H_{i+1} = (b(H_i - T[i] b^{m-1}) + T[i+m]) \bmod M.$$

Dacă $H_i = h_P$, facem o verificare explicită $T[i..i+m-1] = P$ pentru a evita *false positives*. Complexitatea totală este $O(n+m)$ în medie.

3.6 Complexități

Structură	Insert	Find	Delete
Directă	$O(1)$	$O(1)$	$O(1)$
Chaining	$O(1 + \alpha)$	$O(1 + \alpha)$	$O(1 + \alpha)$
Open addressing	$O(1/(1 - \alpha))$	$O(1/(1 - \alpha))$	$O(1/(1 - \alpha))$

Table 2: Complexități medii pentru tabele de dispersie

4 Structuri de date elementare

4.1 Vectori si liste

Vectorii sunt structuri de date elementare. Vectorii se pot aloca static sau dinamic. Avem la dispozitie mai multe variante: vector static/dinamic din limbaj sau `std::vector` din STL sau `std::array` din STL (rar folosit).

Exista liste de mai multe tipuri: inlantuite, dublu inlantuite, circulare, dublu inlantuire circulare. O lista consta dintr-o inlantuire de noduri, fiecare nod avand o valoare si un pointer catre alt nod.

Comparison between lists and vectors:

Operație	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	$O(1)$	$O(n)$
Join	$O(1)$	$O(n)$

Table 3: Complexitatea operațiilor pentru Liste și Array-uri

4.2 Stive

Stiva functioneaza dupa principiul **LIFO** (Last In First Out). Operatiile de baza intr-o stiva sunt: **push**, **pop**, **peek/top**.

Aplicatie: Aplicatie de baza pentru stiva: skyline/ soldier's row/ trompeta. Problema cere sa determinam pentru fiecare indice i , indicele j maxim, $j \in 0, 1, \dots, i - 1$ astfel incat $v[j] \geq v[i]$.

Solutie: Problema se rezolva folosind o stiva. Pentru fiecare element nou intrat in stiva, acesta scoate toate elementele mai mici decat el din stiva. La final in stiva vom obtine un sir crescator.

4.3 Cozi

Coadă functioneaza dupa principiul **FIFO** (First In First Out). Operatiile de baza sunt: **push**, **pop**, **peek/top**.

Aplicatie: O aplicatie clasica pentru coada este *BFS* sau *Algoritmul lui Lee* pentru drum minim pe matrice.

Solutie: Scoatem un element din coada si bagam in coada elementele vecine nevizitate. Continuum procesul pana cand coada este vida.

4.4 Deque

Deque-ul (Double Ended Queue) este o coada care permite inserarea si eliminarea elementelor atat dintr-un capat al cozii cat si din celalalt. Deque-ul combina conceptele de stiva si coada.

Operatiile pe un deque sunt: **push-back**, **push-front**, **pop-back**, **pop-front**, **front**, **back**.

Aplicatie: O aplicatie de baza pentru deque este min-max deque. Aceasta presupune determinarea maximului sau minimului pe un window de o lungime k data.

Soluție: Pe masura ce deplasam window-ul de la stanga la dreapta facem acelasi lucru ca la stiva (elementul nou intrat scoate toate elementele mai mici ca el), dar, eliminam si elemntul din stanga din coada daca am trecut de el.

5 Heapuri

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

Un heap se poate reprezenta ca un vector astfel: Heap = [50, 30, 40, 10, 20, 35, 25]. Pentru o pozitie i avem: $2i$: fiul stang, $2i + 1$: fiul drept si $[i/2]$: tatal.

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

Urca/ Sift/ ReheapUp:

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

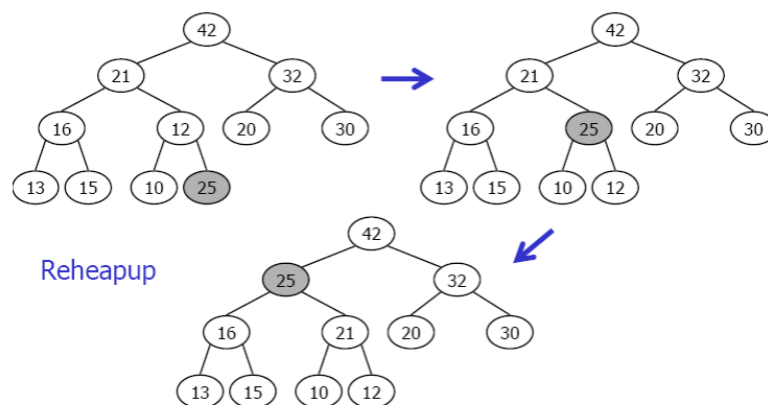


Figure 1: Urca

Coboara/ Percolate/ ReheapDown:

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

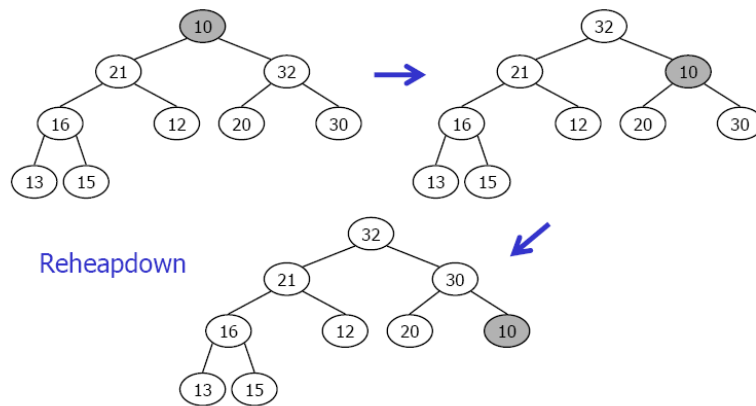
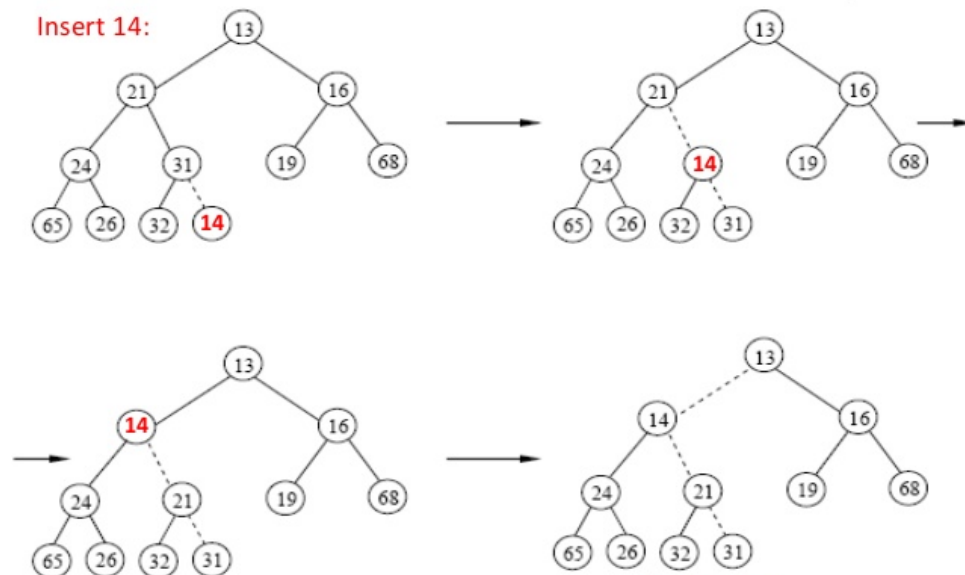


Figure 2: Coboara

Insert:

Insert new element into min-heap



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

11

Figure 3: Insert in min-heap.

Delete:

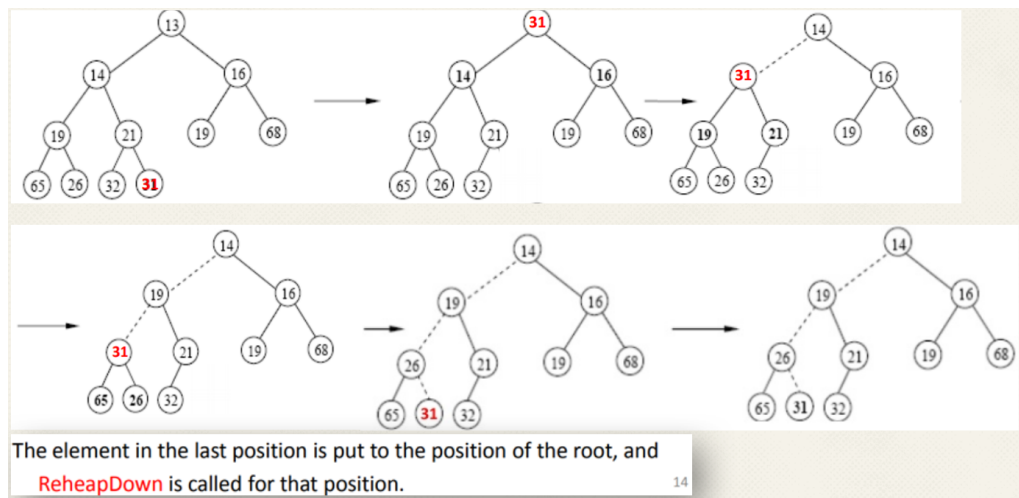


Figure 4: Delete root of the heap.

Analog se procedeaza si pentru stergerea oricarui nod din heap. Se interschimba cu ultimul, iar apoi facem **ReheapDown**.

Heapify

O alta problema este construirea unui heap. Aceasta poarta numele de *heapify*. Complexitatea este $O(n)$ si nu $O(n \log(n))$. Punem valorile in heap random. Pentru fiecare nod de la ultimul nivel in sus, facem **ReheapDown**.

Demonstratie complexitate $O(n)$:

Pentru fiecare coborare vom avea $O(h)$ complexitate. Sunt aprox. $n/2$ frunze in heap. Pentru frunze nu avem ce cobori. Apoi, frunzele au $n/4$ noduri parinte pentru care facem maxim 1 opearatie de swap. Apoi, $n/8$ noduri pentru care facem maxim 2 operatii de swap. Generalizand, obtinem ca numarul de operatii pe care il facem este:

$$\sum \frac{n}{2^i} (i - 1) = n \sum \frac{i - 1}{2^i}$$

Seria $\sum \frac{i-1}{2^i}$ converge la 1, deci obtinem complexitate $O(n)$.

Combinatorica, Diverse

Numărul minim de elemente dintr-un heap binar de înălțime k ? (Vom presupune că rădăcina heap-ului se află la înălțime 0) este: 2^k .

Se pot gasi elementele maxime / minime intr-un heap de minim / maxim? Elementele respective se vor afla in frunze, deci trebuie sa parcurgem frunzele heap-ului (aprox. $n/2$ frunze). Asadar, se poate gasi maximul intr-un min-heap sau invers in complexitate $O(n)$.

Merge Heaps

Deoarece operatia de merge este inceata, exista alte tipuri de heap-uri: *binomiale* si *fibonacci*.

	Căutare Min	Ștergere Min	Inserare	Update	Reuniune
Heap Binar	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Heap Binomial	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$ (amortizat)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Heap Fibonacci	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$ (amortizat)	$\mathcal{O}(1)$	$\mathcal{O}(1)$ (amortizat)	$\mathcal{O}(1)$

5.1 Heapuri Binomiale

Arbori binomiali

Un arbore binomial de ordin k :

- Are exact 2^k noduri.
- Înălțimea k
- Sunt exact C_i^k noduri de înălțime i .

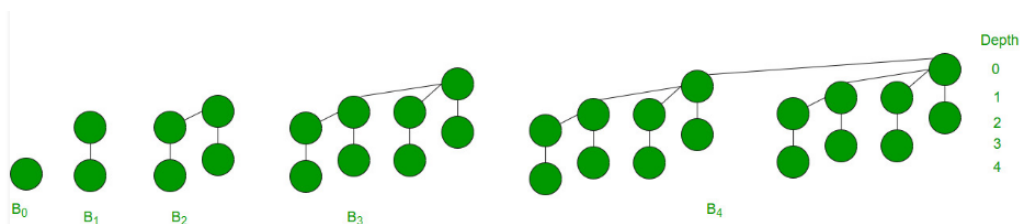


Figure 5: Arbori Binomiali

5.2 Heapuri Binomiale

Un heap binomial este o familie de arbori binomiali care au proprietatea de heap minim. Există o singură structură de heap binomial pentru orice mărime.

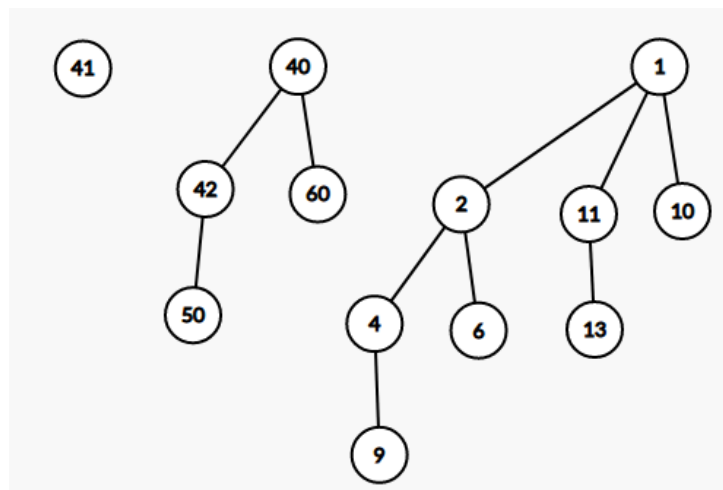


Figure 6: Heap Binomial

Heap-ul din figura este un heap binomial cu 13 noduri ($13 = 8 + 4 + 1$).

Extragerea minimului: Vom avea minimul dintre toate radacinile arborilor binomiali ($\mathcal{O}(\log(n))$ complexitate) sau retinem minimul si il updatam la fiecare alta operatie ($\mathcal{O}(1)$ complexitate).

Merge: Se face astfel: Parcurgem cele doua heap-uri binomiale si combinam arborii binomiali de acelasi rang. Cand obtinem un arbore binomial il putem folosi in reuniune cu alti

arbori. Este ca o suma pe biti si pozitiile bitilor setati din suma vor fi arborii binomiali pe care ii obtinem. Complexitate: $O(\log(m+n))$.

Insert: vom crea un arbore binomial de rang 1 cu un singur nod si vom da merge cu heap-ul initial. Complexitate: $O(\log(n))$ (vine din merge).

Stergere minim: Cand stergem, eliminam minimul, iar apoi facm reuniune. Complexitate: $O(\log(n))$ (vine din merge).

5.3 Heapuri Fibonnaci

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

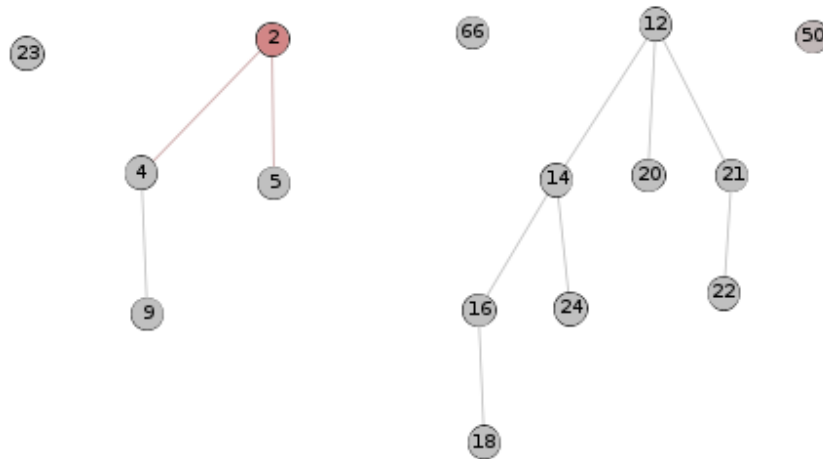


Figure 7: Fibonacci Heap

Structura unui heap fibonacci:

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

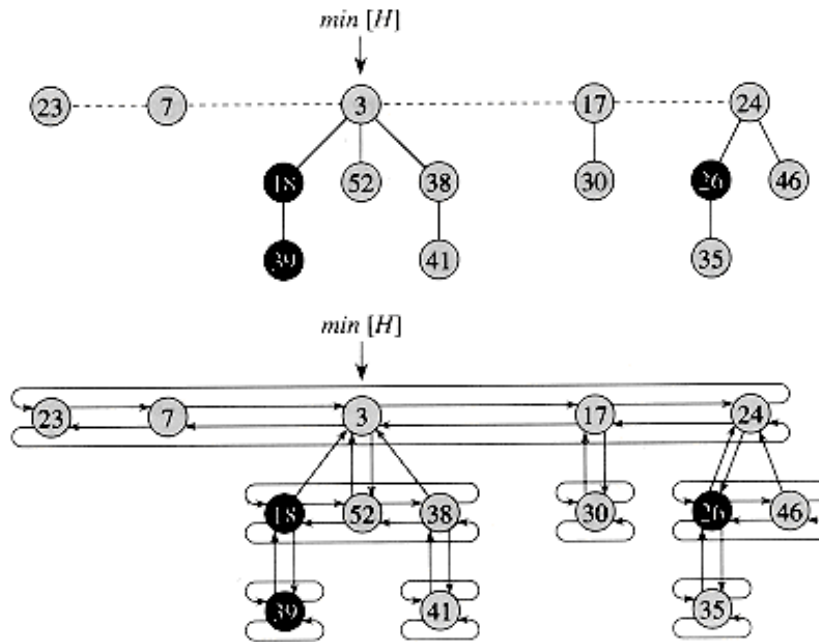


Figure 8: Fibonacci Heap Implementation

Extragere minim: reținem un pointer către minim care îl actualizăm la fiecare operație. Complexitate: $O(1)$.

Inserare: creăm un nou arbore cu un singur element și îl atașăm heap-ului. Nu facem reuniune. Complexitate: $O(1)$

Merge: concatenăm lista de radacini ale primului heap la cel de-al doilea. Modificăm pointer-ul către minim. Complexitate: $O(1)$.

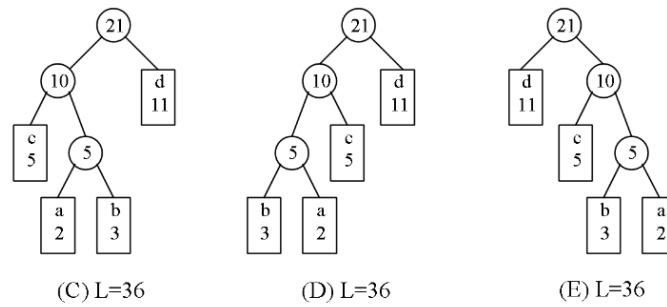
Stergere minim: Stergem minimul și rămânem cu doi arbori liberi (subarborii nodului sters). La extragerea minimului vom consolida heap-ul, adică se face reuniune similar cu heap-urile binomiale. **Consolidarea heap-ului:** Se calculează gradul unui arbore, ca fiind înălțimea arborelui și arborii cu același grad se reunesc.

5.4 Arbori Huffman

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

Construcția unui arbore Huffman Luăm frecvențele caracterelor în ordine crescătoare. Unim de fiecare dată cele mai mici 2 valori. Frecvența noului nod obținut o bagăm în lista și repetem procesul.

Observație: Arborii Huffman nu sunt unici!



Arbori Huffman pt. frunzele cu ponderi $\{(a, 2), (b, 3), (c, 5), (d, 11)\}$.
 $L = 36$ minima.

Figure 9: Arbori Huffman Exemplu

6 Arbori binari de căutare

6.1 Definiții arbori

Definition 6.1 (k -arbore). Un k -arbore T peste un tip de bază dat este definit recursiv după cum urmează:

1. T este vid (notat $T = \emptyset$), sau
2. T constă dintr-un nod rădăcină $root(T)$ de tipul de bază, împreună cu o secvență finită de subarbori disjuncți

$$T_1, T_2, \dots, T_k,$$

care sunt numiți *copiii* lui $root(T)$.

- k se numește **gradul arborelui**

Definition 6.2 (Arbore binar). Un arbore binar este un 2-arbore.

Definition 6.3 (Înălțimea unui arbore). Fiecărui arbore îi asociem un nivel în felul următor

- radacinii îi asociem nivelul 0
- dacă un nod se afla la nivelul i , îi asociem nivelul $i + 1$

Înălțimea este nivelul maxim.

Definition 6.4 (Arbore plin). Un arbore binar este plin dacă fiecare nod al său are 0 sau 2 fii.

Definition 6.5 (Arbore complet). Un arbore binar este complet dacă toate nivelurile sunt complete, exceptând ultimul nivel care e completat de la stânga la dreapta.

6.2 Parcurgerea unui arbore binar

- **Preordine - RSD** - (radacina - stanga - dreapta)
- **Inordine - SRD** - (stanga - radacina - dreapta)
- **Postordine - SDR** - (stanga - dreapta - radacina)

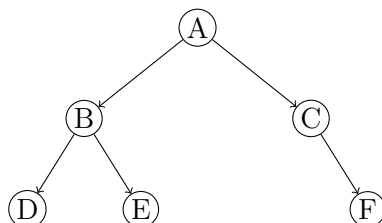
Exercise 6.1 (Reconstrucția arborelui din RSD și SRD). **Enunț.** Se dau două parcurgeri ale unui arbore binar cu noduri $\{A, B, C, D, E, F\}$:

RSD (preordine): A, B, D, E, C, F și SRD (inordine): D, B, E, A, C, F .

Reconstruiți arborele și reprezentați-l grafic.

Idee. 1. În preordine primul nod este rădăcina: A . 2. În inordine separăm subarborele stâng (D, B, E) de subarborele drept (C, F). 3. Pornind de la aceste două liste, repetăm recursiv pentru fiecare subarbore.

Rezolvare.



6.3 Arbori binari de căutare

Un ABC este un arbore binar care satisface următoarea condiție pentru fiecare nod x :

- dacă y e un nod din **subarborele stang** al lui x atunci $\text{cheie}[y] \leq \text{cheie}[x]$.
- dacă y e un nod din **subarborele drept** al lui x atunci $\text{cheie}[y] > \text{cheie}[x]$.

Definition 6.6 (Arbore binar strict). Un arbore binar **strict** e un arbore în care un nod fie nu are niciun fiu, fie are 2 fii.

Înălțimea arborelui

- Minim: Arbore binar complet - $\log n$
- Maxim: Lanț - n .

IMPORTANT Pargurgerea **inordine** a unui ABC ne dă vectorul **sortat**.

6.4 Minim și maxim

Minimul se află în cel mai din stânga nod. Complexitate $O(h)$, unde h e înălțimea arborelui.

Maximul se află în cel mai din dreapta nod. Complexitate $O(h)$.

Căutarea unui element Începem din rădăcină și dacă valoarea din nodul curent este mai mică decât cea ce căutăm, mergem în stânga, dacă valoarea e mai mare, mergem în dreapta.

6.5 Succesor și predecesor

Succesor = cea mai mică valoare din arbore, \geq decât valoarea din nod

Predecesor = cea mai mare valoare din arbore, \leq decât valoarea din nod

Pentru a găsi succesorul unui nod x într-un arbore binar de căutare, procedăm astfel: dacă x are subarbore drept, succesorul este nodul cel mai din stânga din acel subarbore (urmăm legăturile stângi până nu mai există); dacă nu, urcăm de la x spre rădăcină până întâlnim primul strămoș pentru care x se află în subarborele stâng – acel strămoș este succesorul.

Analog, pentru predecesor: dacă x are subarbore stâng, predecesorul este nodul cel mai din dreapta din subarborele stâng; în caz contrar, urcăm spre părinți până găsim un strămoș pentru care x face parte din subarborele drept, iar acel strămoș este predecesorul.

6.6 Inserare și ștergere

Inserare

- Similar cu căutarea.
- Începem din rădăcină și avem următoarele cazuri:
 - ☐ dacă valoarea dată este identică cu cheia rădăcinii, se renunță la inserare;
 - ☐ dacă valoarea dată este mai mică decât cheia rădăcinii, se continuă cu subarborele stâng;
 - ☐ dacă valoarea dată este mai mare decât cheia rădăcinii, se continuă cu subarborele drept;
- Se continuă acest proces recursiv până când se ajunge la un nod fără copii, moment în care noul nod este inserat ca fiu al acestui nod. **Obs.** După inserare, proprietatea de arbore binar de căutare trebuie să fie respectată.

Ștergere

- Avem 3 cazuri:
 1. Dacă nodul nu are fii, îl ștergem.
 2. Dacă nodul are un singur fiu, îl ștergem și creăm o legătură directă între tată și noul fiu.
 3. Dacă nodul are ambii fii, găsim succesorul, îl înlocuim pe nodul de șters și apoi eliminăm succesorul (care are cel mult un fiu).

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

Table 4: Complexitatea operațiilor într-un BST de înălțime h .

6.7 Arbori binari cu 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)

7 Arbori binari de cautare echilibrati

Cautam modalitati de a mentine adancimea unui arbore cat mai apropiata de $\log(n)$.

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

7.1 Red Black Trees

- 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 descendent NULL are același număr de noduri negre

7.2 AVL Trees

Pentru AVL-uri definim **factorul de echilibru** al unui nod: $BF(X) = h(subarbDrp(X)) - h(subarbStg(X))$, unde h este înălțimea unui arbore.

Un arbore AVL este un arbore care are $BF(x) \leq 1$, pentru orice nod x .

Pentru rebalansarea arborelui se aplica rotații: rotație stânga-stânga, rotație stânga-dreapta, rotație dreapta-stânga, rotație dreapta-dreapta.

8 Skip Lists

Sunt structuri de date echilibrate care ajuta la cautare rapida. Elementele **sunt sortate**.

Structura:

- Skip-listul este o lista inlatuita, dar extinsa pe mai multe nivele.
- La fiecare nivel adăugat, sărim peste o serie de elemente față de nivelul anterior
- Nivelele au legături între ele

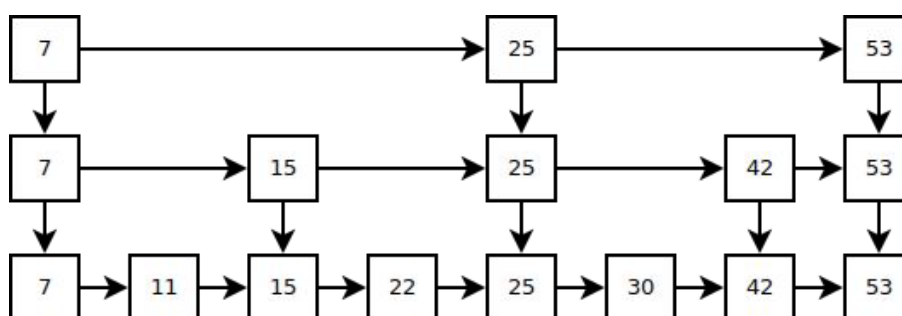


Figure 10: Skip List

Cautare

- 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)
- Ne mutăm în următoarea listă (mergem în jos)

- Reluăm algoritmul de la pasul 2)

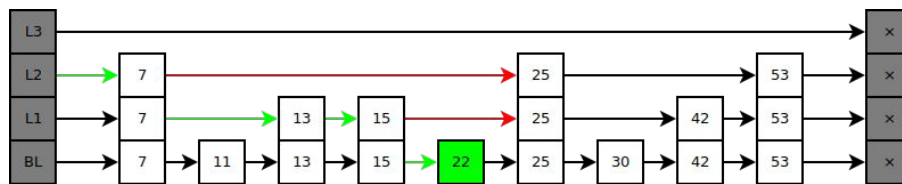


Figure 11: Cautare elementul 22

Inserare

- Vrem să inserăm elementul x . Observație: Lista de jos trebuie să conțină toate elementele, deci căutăm locul lui x în lista de jos $\rightarrow \text{search}(x)$ și adăugăm x în locul respectiv.
- 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

În medie vom avea $\frac{1}{2}$ elemente nepromovate, $\frac{1}{4}$ elemente promovate 1 nivel, $\frac{1}{8}$ elemente promovate 2 nivele și tot așa.

Deci, obținem o complexitate $O(\log(n))$

Stergere

Stergem elementul x din toate listele în care se afla. Complexitate: $O(\log(n))$.

9 Treap

Structură de date arborescentă care menține simultan proprietatea de arbore binar de căutare (BST) și cea de max-heap.

Puțin mai formal, fie (X_i, Y_i) nodurile arborelui. Treap-ul asigură structură de ABC pentru toți X_i , iar în Y_i , structura de max-heap.

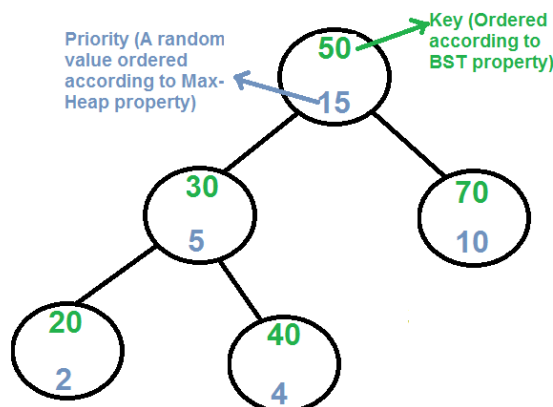


Figure 12: Treap

De ce treapuri? Sunt ușor de implementat. Cu puține modificări, permit abordarea multor tipuri de query-uri și update-uri (sume, cmmdc, rotații etc pe intervale).

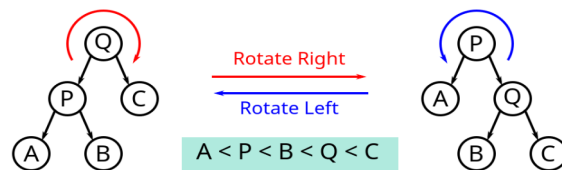


Figure 13: Rotatii pe arbori

Inserare

La inserare, inseram exact ca într-un BST obisnuit, iar după, pe recursie, rotim în sens invers subarboarele cu rădăcina în nodul curent, în funcție de tipul fiului (stâng/drept).

Cautare

Exact ca la un BST obisnuit.

Stergere

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

- E frunză \rightarrow Îl ștergem direct.
- Are numai un fiu \rightarrow Fiul ia locul nodului respectiv
- Are doi fii \rightarrow Rotim în locul rădăcinii subarboareului făcut din nodul curent fiul cu prioritatea cea mai mare. Repetăm algoritmul până când ne aflăm în unul dintre primele două cazuri.

10 Range Minimum Query

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

Operation	Segment Tree	RMQ	Batog(sqrt)
Build	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Query	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\sqrt{n})$
Point Update	$\mathcal{O}(\log n)$	N/A (not supported)	$\mathcal{O}(1)$

Table 5: Comparison of Segment Tree, Sparse Table (RMQ) Complexities and Sqrt Decomposition (Batog)

Atentie: RMQ se preteaza pentru situatia in care nu avem update-uri. Altfel, daca exista update-uri, folosim un arbore de intervale sau batog.

Exemplu:

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

Minimul pe $(0,3)$ este 2. Minimul pe $(5,9)$ este 2.

Pentru fiecare poziție i din v , vom calcula mai multe minime. Pentru fiecare secvență de lungime putere de 2 care începe la poziția i , vom calcula minimul dintre elementele ei.

Deci, pentru fiecare poziție din v vom avea maxim $\log(n)$ minime de calculat (atâtea puteri de 2 sunt mai mici sau egale decât n).

Astfel, vom obține o matrice $r[p][i] = \text{minimul din } v \text{ pentru elementele dintr-o secvență de lungime } 2^p \text{ care începe la poziția } i$.

v:	4	2	7	8	3	5	9	1	6	2	9
r:											
2^0	4	2	7	8	3	5	9	1	6	2	9
2^1	2	2	7	3	3	5	1	1	6	2	9
2^2	2	2	3	3	1	1	1	1	2	2	9
2^3	1	1	1	1	1	1	1	1	2	2	9

Construcție

Un element de pe linia p se calculează ca minimul a două elemente de pe linia $p - 1$:

$$r[p][i] = \min(r[p-1][i], r[p-1][i + 2p - 1]).$$

Având deja linia 0 cunoscută nu ne rămâne decât să calculăm valorile linie cu linie, în ordinea crescătoare a indicilor de linii.

Query

La fiecare interogare primim capetele unui interval, $[st, dr]$ care, evident, nu este neapărat de lungime putere de 2.

Pentru a răspunde la query putem face o descompunere în baza 2 și obținem $O(\log(n))$. Dar, putem să facem în $O(1)$. Pentru acest lucru, să considerăm cea mai mare putere de 2 mai mică sau egală decât L (notăm cu len această valoare).

Acum să considerăm, intervalul de lungime len care începe la poziția st și pe cel de lungime len care se termină la poziția dr .

Aceste două intervale acopera complet intervalul nostru, deci putem face minimul / maximul lor și acesta este răspunsul la query.

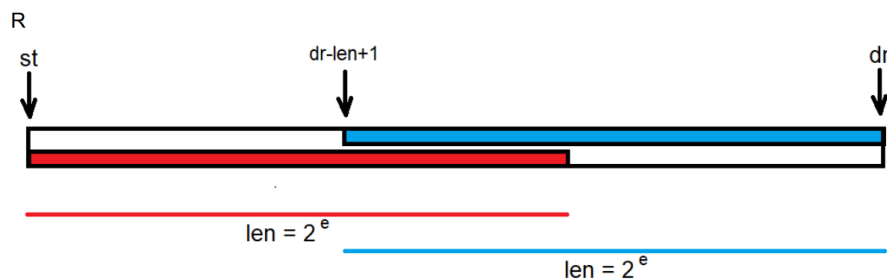


Figure 14: RMQ Query

10.1 Lowest Ancestor (LA)

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?

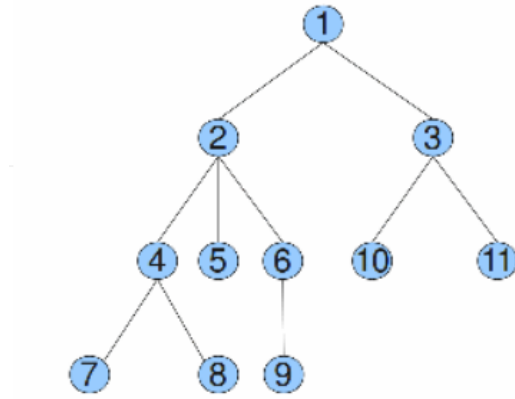


Figure 15: Tree Example

$$LA(2, 1) = 1, LA(9, 1) = 6, LA(9, 2) = 2, LA(6, 4) = -1$$

Mai multe rezolvari posibile:

- Parcurg din tata in tata. Complexitate: $O(n)$ pe query si $O(1)$ memorie.
- Retin un map unde stiu toti stramosii nodurilor. Preprocesare si memorie: $O(n \cdot h)$ si $O(1)$ pe query.
- Sqrt Decomposition (Batog). Țin pointer catre tatăl de ordin radical din n . Obținem $O(n)$ memorie suplimentara si $O(\sqrt{n})$ pe query.
- **Binary Lifting:** Pentru fiecare nod, țin tații de înălțime $1, 2, 4, 8, 16, \dots$. La query facem similar cu cautarea binara. Sarim cu cea mai mare putere de 2 posibila. Complexitate $O(n \cdot \log(n))$ preprocesoare, $O(n \cdot \log(n))$ memorie suplimentară si $O(\log(n))$ pe query.

10.2 Lowest Common Ancestor (LCA)

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

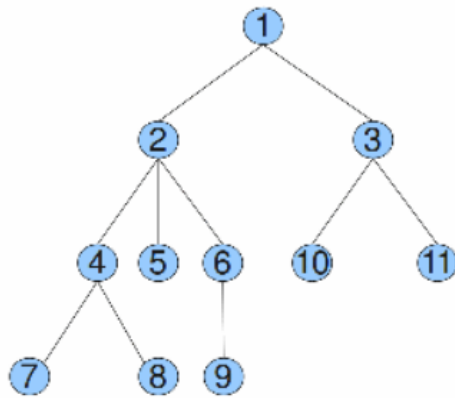


Figure 16: Tree Example

$$LCA(4, 9) = 2, LCA(4, 11) = 1, LCA(7, 6) = 2$$

Una dintre aplicatiile **structurii RMQ** este determinarea LCA-ului.

- **Liniazam arobrele printr-o parcurgere Euler:** 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, reținem și prima sa apariție în parcurgerea Euler.
- $LCA(i, j) = RMQ(first[i], first[j])$. Unde RMQ calculează minimul pe vectorul de distanțe până la rădăcină.
- Acum putem răspunde la query-uri $LCA(x, y)$ în $O(1)$.

Complexitate: $O(n \log(n) + q)$

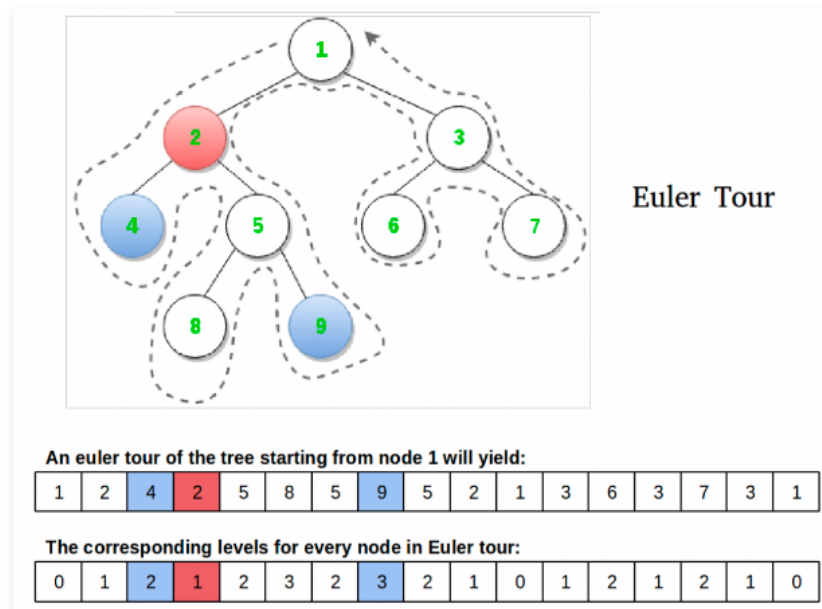


Figure 17: Euler Tour

11 Trie (Prefix Tree)

Problema: Am mai multe cuvinte pe care le tin minte și apoi am întrebări de genul: este cuvântul dat în acea listă sau nu? Soluții posibile:

- Hash-uri, $O(l)$ (unde l e lungimea cuvântului) pe query și $O(n \cdot l)$ memorie.

Problema: 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? Soluții posibile:

- Nu mai merge cu hash-uri.
- Sortăm toate cuvintele lexicografic și apoi căutăm binar. $O(n \cdot l)$ **memorie** și $O(\log(n) \cdot l)$

- Ținem toate cuvintele într-un arbore binar de căutare echilibrat. $O(n \cdot l)$ **memorie** și $O(\log(n) \cdot l)$

Implementare Trie:

- 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, ține pointer-ul către nodul cu acea literă.

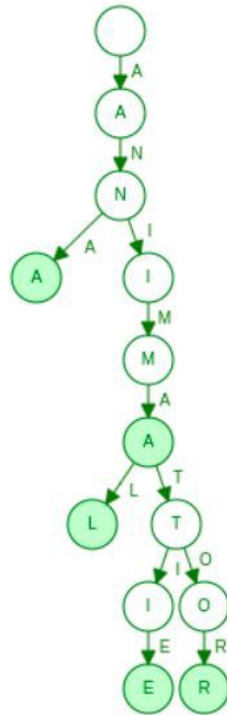


Figure 18: Trie cu cuvintele ana, animator, animație, animal, anima.

Inserare

Pornim din rădăcină și, la fiecare literă, mergem în nodul corespunzător literei. Eventual creăm acel nod. **Complexitate:** $O(l)$

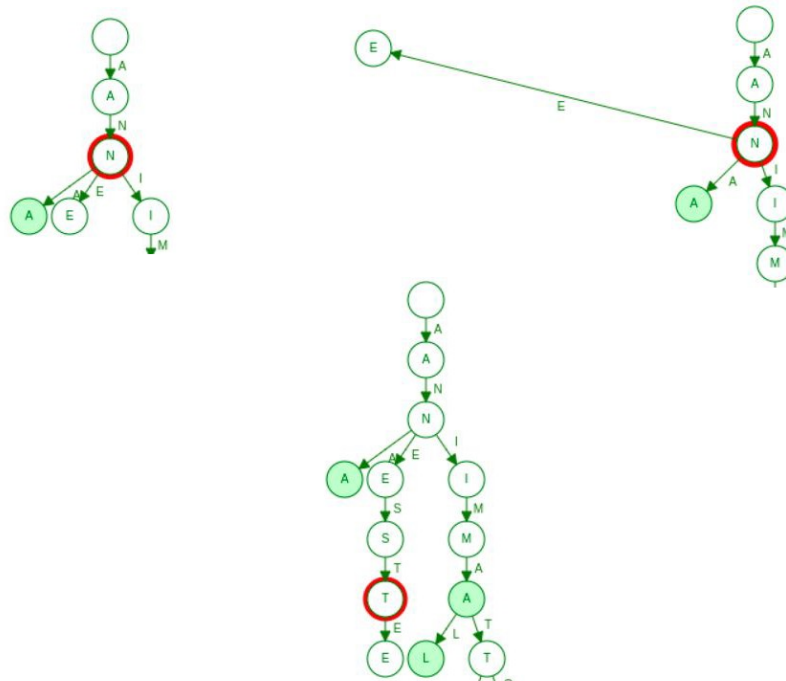


Figure 19: Inserare in trie

Cautare

Cautarea unui cuvânt din dictionar:

- 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.
Complexitate $O(l)$.

Sortarea lexicografică în trie

Odată ce trie-ul este construit, sortarea în ordine lexicografică poate fi realizată printr-un traversare în preordine (preorder traversal). Într-un trie, o traversare în preordine va vizita nodurile în ordine lexicografică, deoarece:

- Nodurile sunt vizitate de la rădăcină spre frunze.
- La fiecare nivel, vizităm nodurile în ordinea alfabetică a literelor.