Resurse examen SD

Seria 15

June 26, 2025

Contents

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	$\Omega(n \log n)$	$\Theta\big(n(\log n)^2\big)$	$O\big((n \log n)^2\big)$	$O(1)$
Bucket Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n^2)$	$O(n + k)$
Radix Sort	$\Omega(n\,k)$	$\Theta(n\,k)$	$O(n\,k)$	$O(n + k)$
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 creşterea funcţiei 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 creşterea „exactă” (în sens asimptotic), iar O şi 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

$$h : \mathcal{U} \to \{0, 1, \ldots, m-1\},$$

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 $\{0, \ldots, M-1\}$. 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 $\{1, \ldots, N\}$, putem implementa direct o tăblă de dispersie printr-un vector binar:

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

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 $T[0 \ldots m-1]$ 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 $\mathcal{U}$,

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

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 (substringu-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:

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

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

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 inlatuire 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, $j \in 0, 1, ..., i-1$ astfel incat v[j] $>=$ 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

Coada 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. Continuam 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.

Solutie: 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 ... = 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:

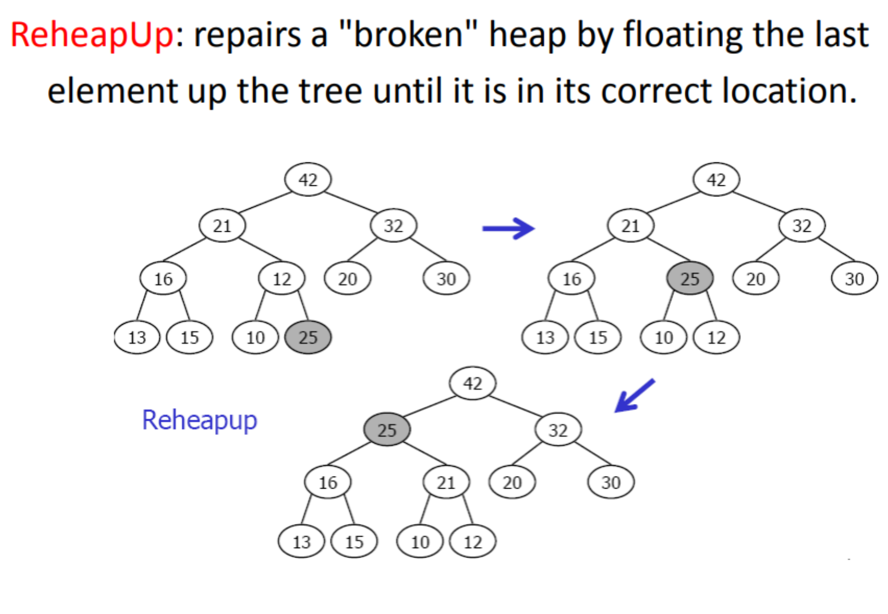


Figure 1: Urca

Coboara/ Percolate/ ReheapDown:

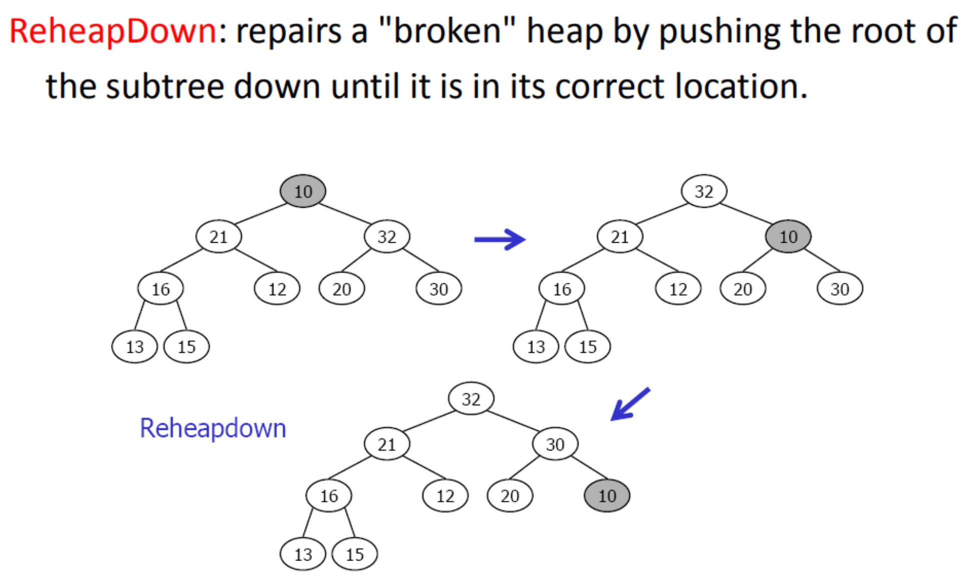


Figure 2: Coboara

Insert:

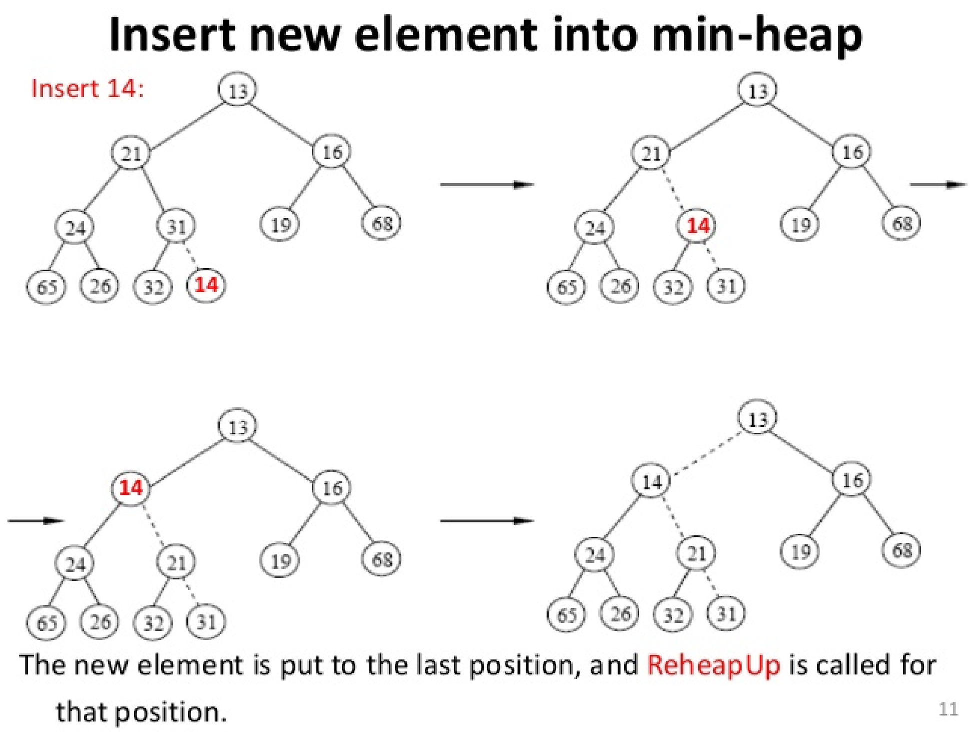


Figure 3: Insert in min-heap.

Delete:

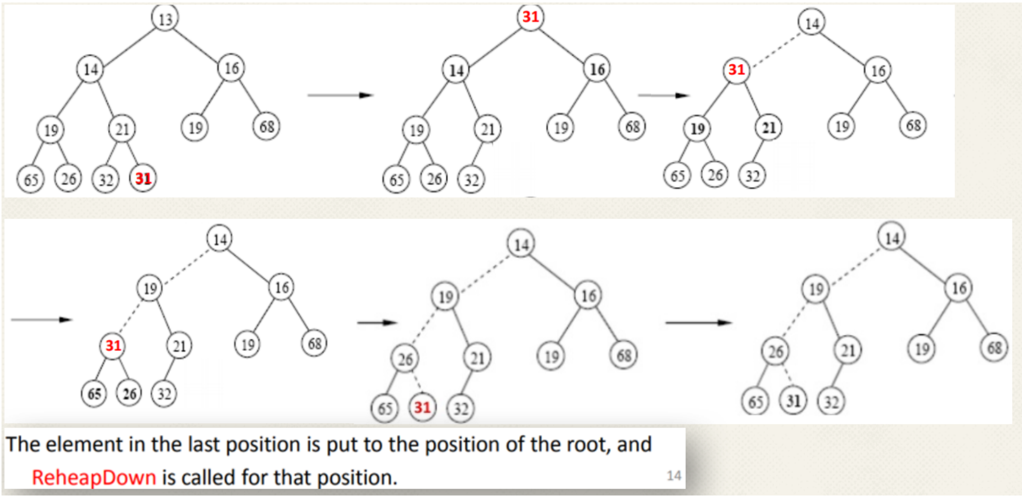


Figure 4: Delete root of the heap.

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(nlog(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 binominal de ordin k:

- Are exact 2^k noduri.
- Inaltimea k
- Sunt exact C_i^k noduri de inaltime i.

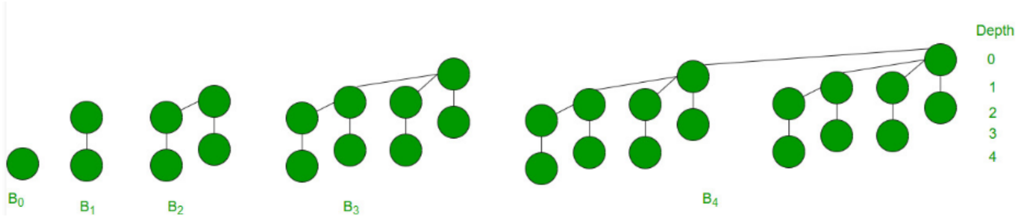


Figure 5: Arbori Binomiali

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.

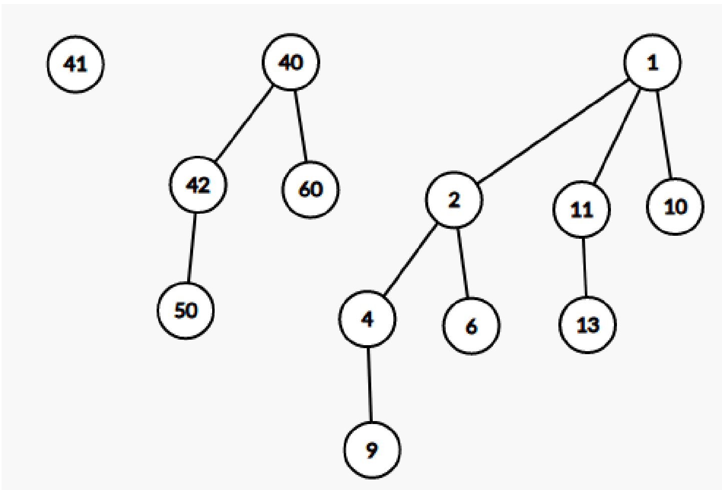


Figure 6: Heap Binomial

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 (O(log(n)) complexitate) sau retinem minimul si il updatam la fiecare alta operatie (O(1) complexitate).

Merge: Se face astfel: Parcurgem cele doua heap-uri binomiale si combinam arborii binomiali de acelasi rang. Cand obitnem un arbore binomial il putem folosi in reuniune cu alti arbori. Este ca o suma pe biti si pozitile 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 aceeasi dimesniune in heap.**
- Arborii dintr-un heap Fibonacci nu sunt ordonați.
- Arborii din componență au mărimi **puteri ale lui 2**. Fiii vor fi arbori de mărime $1, \ldots, k-1$, dar nu neapărat sortați de la stânga la dreapta.

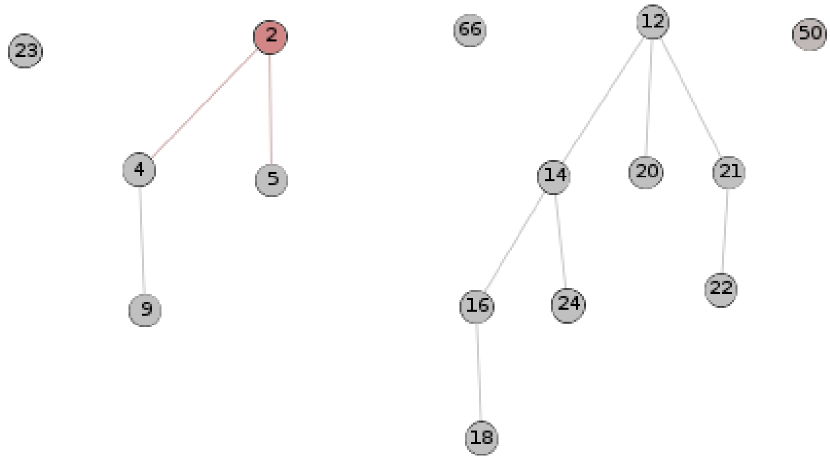


Figure 7: Fibonacci Heap

Structura unui heap fibonacci:

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

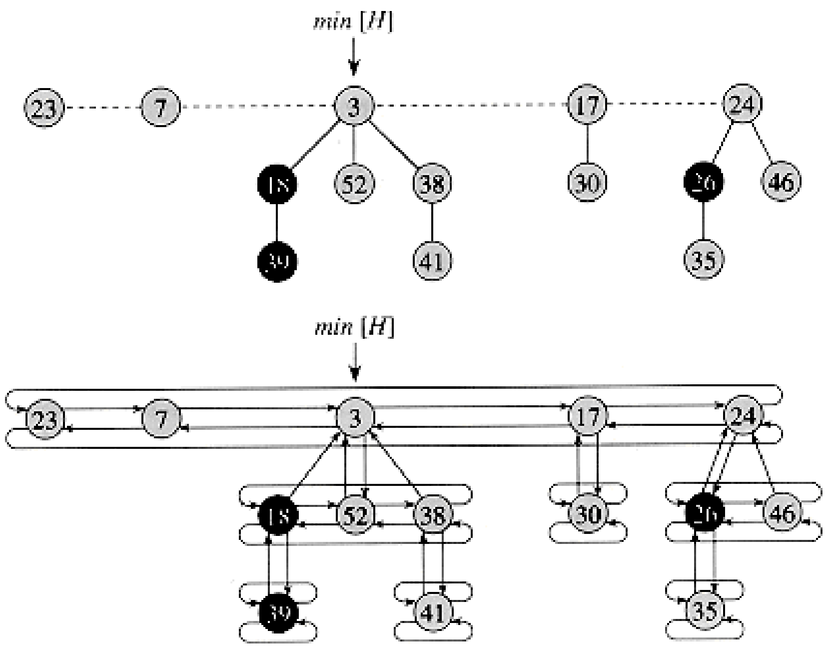


Figure 8: Fibonacci Heap Implementation

Extragere minim: retinem un pointer catre minim care il actualizam la fiecare operatie. Complexitate: O(1).

Inserare: cream un nou arbore cu un singur element si il atasam heap-ului. Nu facem reuniune. Complexitate: O(1)

Merge: concatenam lista de radacini ale primului heap la cel de-al doilea. Modificam pointer-ul catre minim. Complexitate: O(1).

Stergere minim: Stergem minimul si ramanem cu doi arbori liberi (subarborii nodului sters). La extragerea minimului vom consolida heap-ul, adica se face reuniune similar cu heap-urile binomiale. **Consolidarea heap-ului**: Se calculeaza gradul unui arbore, ca fiind inaltimea arborelui si arborii cu acelasi 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.

Constructia unui arbore Huffman Luam frecventele caracterelor in ordine crescatoare. Unim de fiecare data cele mai mici 2 valori. Frecventa noului nod obtinut o bagam in lista si repetem procesul.

Observatie: Arborii Huffman nu sunt unici!

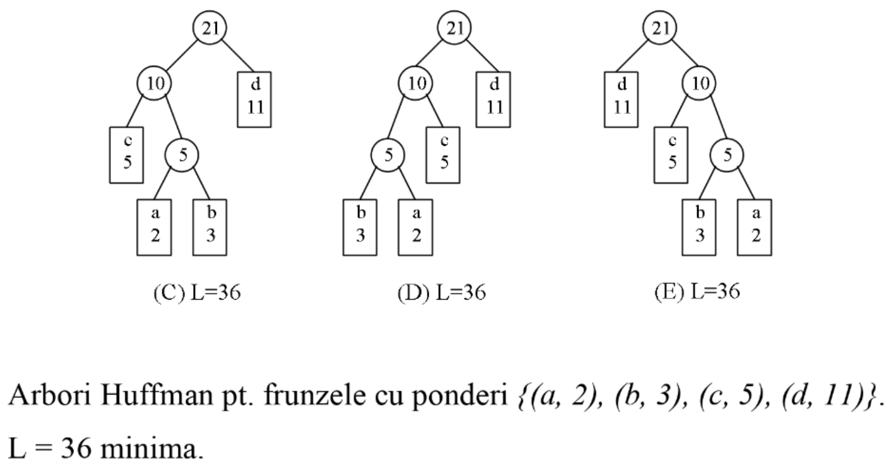


Figure 9: Arbori Huffman Exemplu

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

$$T_1, T_2, \ldots, T_k,$$

   care sunt numiţi *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 ii asociem nivelul 0
- daca un nod se afla la nivelul i, ii asociem nivelul $i + 1$

Inaltimea 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)

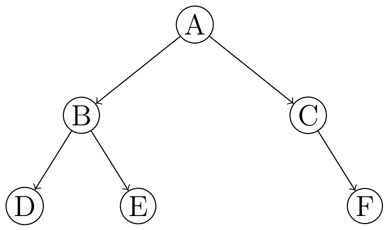
Exercise 6.1 (Reconstrucția arborelui din RSD și SRD). **Enunț.** Se dau două parcurgeri ale unui arbore binar cu noduri $\{A, B, C, D, E, F\}$:

$$\text{RSD (preordine): } A,\ B,\ D,\ E,\ C,\ F \quad \text{și} \quad \text{SRD (inordine): } D,\ B,\ E,\ A,\ C,\ F.$$

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 cheie[y] $\leq$ cheie[x].
- dacă y e un nod din **subarborele drept** al lui x atunci cheie[y] $>$ 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 ceea 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)

6.8 Selectarea celui de-al k-lea element într-un BST

Pentru a putea regăsi rapid al k-lea element (în ordine crescătoare) într-un arbore binar de căutare, adaugăm la fiecare nod x cu un câmp suplimentar

$$\mathit{size}(x) = \text{numărul de noduri din subarborele cu rădăcina în } x.$$

După fiecare inserare sau ștergere, actualizăm aceste valori urcând de la frunză spre rădăcină.

Pornind de la rădăcina arborelui, notăm prin $m = \mathit{size}(x.\mathit{left})$ numărul de elemente din subarborele stâng al nodului curent x. Pentru a găsi elementul de rang k (unde $k = 1$ corespunde celui mai mic element):

- dacă $k = m + 1$, atunci x însuși este al k-lea element;
- dacă $k \le m$, atunci al k-lea element se află în subarborele stâng;
- altfel (adică $k > m + 1$), continuăm căutarea în subarborele drept, cu rangul ajustat la $k - (m + 1)$.

În fiecare pas coborâm cu un singur nivel în arbore, deci complexitatea acestui procedeu este $O(h)$, unde h este înălțimea arborelui."‘

7 B-Arbori (B-Trees)

Un *B-arbore* de grad minim $t \ge 2$ este un arbore echilibrat în care fiecare nod intern poate avea între t și $2t$ fii, și în care toate frunzele se află pe același nivel. Mai precis:

Definition 7.1 (B-arbore)**.** Fie $t \ge 2$. Un arbore T este numit *B-arbore de grad minim* t dacă satisfice următoarele condiții:

1. Fiecare nod x are $n(x)$ chei, cu $t - 1 \le n(x) \le 2t - 1$, cu excepția rădăcinii care poate avea $1 \le n(\text{root}) \le 2t - 1$.
2. Dacă x nu este frunză, are $n(x) + 1$ pointeri către fiii săi.
3. Toate frunzele se găsesc la același nivel (înălțimea arborelui).
4. Cheile dintr-un nod x sunt stocate în ordine crescătoare:

   $$k_1(x) \le k_2(x) \le \cdots \le k_{n(x)}(x),$$

   iar domeniile subarborilor satisfac

   $$k_1(x) < \cdots < k_i(x) < \text{chei din subarborele } i + 1 < k_{i+1}(x) < \cdots .$$

Înălțimea unui B-arbore

Un B-arbore de înălțime h și cu minim de chei în fiecare nod are cel puțin

$$1 + (t) + (t)^2 + \cdots + (t)^h = \frac{t^{h+1} - 1}{t - 1}$$

chei, deci $h = O(\log_t n) = O(\log n)$.

Operații de bază

Căutare Începem de la rădăcină și, pentru un nod x, căutăm prin binsearch în cele $n(x)$ chei poziția la care k s-ar insera sau s-ar găsi. Dacă găsim, ne oprim; altfel continuăm recursiv în subarborele corespunzător. *Complexitate:* $O(\log_t n) \cdot O(\log t) = O(\log n)$.

Inserare 1. Căutăm frunza L în care s-ar afla cheia k. 2. Dacă L are strict mai puțin de $2t-1$ chei, adăugăm k și terminăm. 3. Dacă L este *plin* ($2t-1$ chei), îl *splitt-ăm* în două noduri cu câte $t-1$ chei, promovăm cheia mediană în părinte și repetăm recursiv pentru cheia promovată. *Complexitate:* $O(t \log_t n) = O(t \log n)$.

Ștergere 1. Dacă cheia k se află într-o frunză L și după ștergere rămâne cel puțin $t-1$ chei, o eliminăm direct. 2. Dacă în urma ștergerii L rămâne cu $< t-1$ chei, împrumutăm o cheie de la un frate cu t chei sau, dacă niciun frate nu poate împrumuta, *fuzionăm* L cu un frate și o cheie părinte, apoi continuăm recursiv. 3. Dacă k e într-un nod intern x, îl înlocuim cu predecesorul sau succesorul său (dintr-o frunză) și ștergem apoi recursiv din frunză. *Complexitate:* $O(t \log_t n) = O(t \log n)$.

Exerciții rezolvate

Exercise 7.1. Pentru ce valori ale lui t următorul nod poate apărea într-un B-arbore de grad minim t?

$$\big[\,[3,5] \;\big|\; 7 \;\big|\; [10,12,14]\big]$$

Solution. În nod avem $n = 3$ chei. Condiţiile pentru un nod nenul (ne-rădăcină) sunt

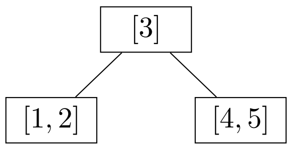
$$t-1 \;\le\; n \;\le\; 2t-1 \quad\Longrightarrow\quad t-1 \le 3 \le 2t-1.$$

Din prima: $t \le 4$. Din a doua: $2t-1 \ge 3 \implies t \ge 2$. **Răspuns:** orice $t \in \{2,3,4\}$.

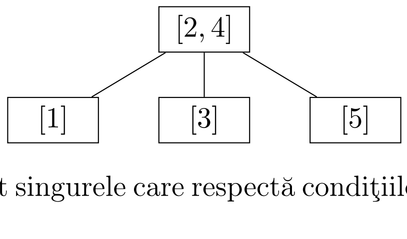
Exercise 7.2. Desenați toate B-arborii de grad minim $t = 2$ care conțin exact cheile $\{1,2,3,4,5\}$.

Solution. Pentru $t = 2$ fiecare nod nenul trebuie să aibă între 1 și 3 chei. Există două posibilităţi de structură:

(a) Rădăcină cu o singură cheie $[3]$ **și doi copii cu câte două chei:**



(b) Rădăcină cu două chei $[2,4]$ **și trei copii cu câte o cheie:**



Aceste două configuraţii sunt singurele care respectă condiţiile pentru $n(x) \in [\,1,3\,]$ și acoperă exact cele 5 chei.

8 Arbori binari de cautare echilibrati

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

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

8.1 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

8.2 AVL Trees

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

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

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

9 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

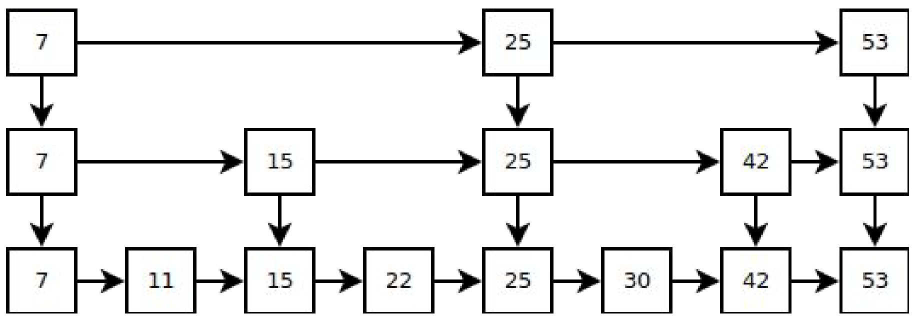


Figure 10: Skip List

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)

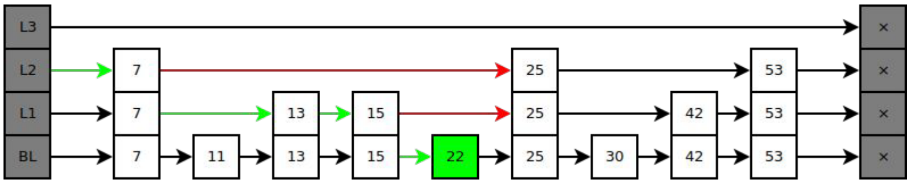


Figure 11: Cautare elementul 22

Inserare

- Vrem să inserăm elementul x. Observație: Lista de jos trebuie să conțină toate elementele, deci căutăm locul lui x în lista de jos → search(x) si adaugam x in 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 aveae $\frac{1}{2}$ elemente nepromovate, $\frac{1}{4}$ elemente promovate 1 nivel, $\frac{1}{8}$ elemente promovate 2 nivele si tot asa.

Deci, obtinem o complexitate O(log(n))

Stergere

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

10 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.

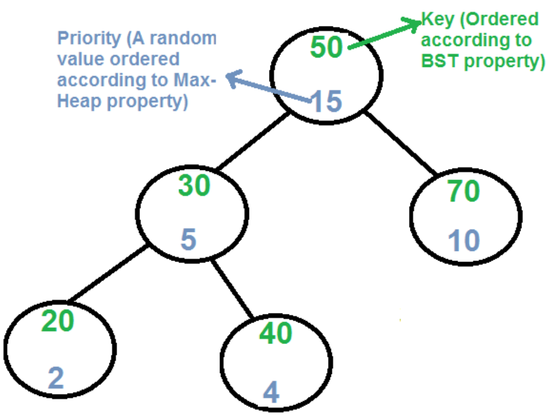


Figure 12: Treap

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

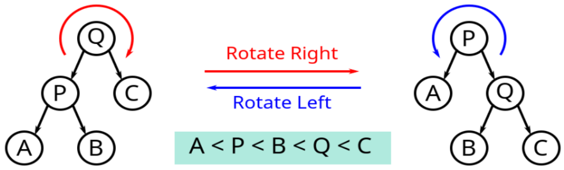


Figure 13: Rotatii pe arbori

Inserare

La inserare, inseram exact ca intr-un BST obisnuit, iar dupa, pe recursie, rotim în sens invers subarborele 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 -> Îl ștergem direct.
- Are numai un fiu -> Fiul ia locul nodului respectiv
- Are doi fii -> Rotim în locul rădăcinii subarborelui făcut din nodul curent fiul cu prioritatea cea mai mare. Repetăm algoritmul până când ne aflăm în unul dintre primele două cazuri.

11 Arbore de intervale, Batog

Problema de tip *Range Query* / *Point Update*

Se dă un vector $A[0 \ldots n-1]$ inițial cu valori numerice, iar apoi se efectuează două tipuri de operații:

- **Point Update:** adaugă valoarea x la poziția i, adică $A[i] \mathrel{+}= x$.
- **Range Query:** raportează maximul valorilor pe intervalul $[i, j]$, adică

$$\max_{k=i}^{j} A[k].$$

Exemplu:

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

Soluție prin decompoziție *sqrt* (batog)

Ideea principală. Împărțim tabloul $A[0 \ldots n-1]$ în blocuri de dimensiune aproximativă $B = \lceil \sqrt{n} \rceil$. Pentru fiecare bloc b (care acoperă pozițiile $[bB \ldots \min((b+1)B-1, n-1)]$) păstrăm în prealabil

$$M[b] = \max\{\, A[i] : i \in [bB,\, (b+1)B-1] \}.$$

Astfel:

- **Interogare** $\max(i, j)$**.**

1. Parcurgem „capătul stâng”: de la i până la sfârșitul blocului lui i, luând maximul direct din A.
2. Parcurgem blocurile întregi complet incluse între blocul lui i și cel al lui j, luând maximul dintre valorile $M[b]$.
3. Parcurgem „capătul drept”: de la începutul blocului lui j până la poziția j, din nou direct din A.

Complexitate: $O(B + n/B) = O(\sqrt{n})$.

- **Actualizare „point update” $A[p]$ += x.** Modificăm $A[p]$ și apoi recalculeăm în $O(B)$ noul maxim al blocului $b = \lfloor p/B \rfloor$. Complexitate: $O(B) = O(\sqrt{n})$.

Configurarea blocurilor și exemplu vizual

3	9	2	5	7	34	6	11	8
	bloc 0			bloc 1			bloc 2	
	$\max\{3, 9, 2\} = 9$			$\max\{5, 7, 34\} = 34$			$\max\{6, 11, 8\} = 11$	

Astfel, de exemplu $\max(2, 7)$ se obține prin:

$$\max\left\{\begin{array}{c} \max(A[2 \dots 2]), \\ M[1], \\ \max(A[6 \dots 7]) \end{array}\right\} = \max\{2,\ 34,\ \max\{6, 11\}\} = 34.$$

Arbori de intervale (Segment Trees)

Un arbore de intervale stochează agregări (sumă, minim, maxim etc.) pe subintervale ale unui vector $A[0 \dots n-1]$. Structura este un arbore binar complet implicit, în care fiecare nod ν corespunde unui interval $[L, R]$ din array.

- **Structură şi memorie.** Rădăcina reprezintă întregul interval $[0, n-1]$. Fiecare nod care acoperă $[L, R]$ are doi copii:

$$\text{stânga} \to [L, M], \quad \text{dreapta} \to [M+1, R], \quad M = \lfloor (L+R)/2 \rfloor.$$

  Toate frunzele sunt intervale unitare $[i, i]$. Se alocă aproximativ $4n$ noduri, deci spaţiul este $O(n)$.

- **Construire (build).** Pornim de la rădăcină şi, dacă $L = R$, setăm valoarea nodului la $A[L]$. Altfel, construim recursiv copiii şi apoi calculăm valoarea părintelui ca $\text{agg}(T[2\nu], T[2\nu + 1])$ (unde agg poate fi min, max, sumă etc.). Cost total: $O(n)$.

- **Interogare pe interval (query).** Pentru a răspunde la o interogare $\text{agg}(A[i \dots j])$, parcurgem arborele de la rădăcină:

  1. Dacă intervalul nodului $[L, R]$ este complet în afara lui $[i, j]$, returnăm valoarea neutră (ex: $+\infty$ pentru minim, $-\infty$ pentru maxim, 0 pentru sumă).
  2. Dacă $[L, R] \subseteq [i, j]$, returnăm valoarea stocată la nod.
  3. Altfel, combinăm rezultatele obținute recursiv din copiii stânga și dreapta prin agg.

Fiecare interogare vizitează cel mult $O(\log n)$ noduri, deci

$$\text{complexitate query} = O(\log n).$$

- **Actualizare punctuală (point update).** Pentru a modifica $A[p] \leftarrow A[p] + x$ (sau înlocuirea cu o valoare nouă), coborâm din rădăcină până la frunza $[p, p]$, actualizăm valoarea şi urcăm recursiv actualizând fiecare părinte:

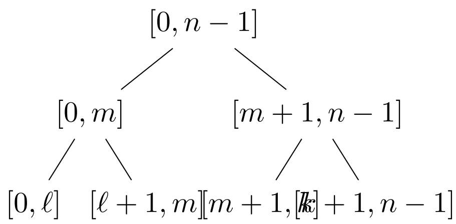
  $$T[\nu] \; = \; \text{agg}(T[2\nu],\, T[2\nu + 1]).$$

  Cost: $O(\log n)$.

- **Lazy propagation (actualizare pe interval).** Pentru operaţii de tip *range add* sau *range assign*, introducem în fiecare nod un „marcaj” (*lazy tag*) care păstrează actualizările întârziate ce trebuie propagate copiilor. - Când un nod ν este complet cuprins în intervalul de actualizare, aplicăm direct modificarea la $T[\nu]$ şi stocăm marcajul în lazy$[\nu]$ fără a coborî imediat. - La momentul interogării sau al necesităţii de a traversa copiii, propagăm marcajul de la ν la cei doi copii, actualizând valorile lor şi marcajele corespunzătoare. Astfel obținem:

  $$\text{complexitate query/update range} = O(\log n).$$

- **Aplicații și variații.**
  - *Range Sum Query (RSQ)*, *Range Maximum Query (RMQ)*, *Range GCD*, *Range XOR* etc.
  - Segment tree bidimensional pentru probleme pe matrice.
  - *Segment tree on array of vectors* pentru interogări de tip „count how many in $[i, j]$ lie in $[x, y]$”.



12 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 fiecarte 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] = minimul din v pentru elementele dintr-o secvență de lungime 2^p 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

Constructie

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 raspunde la query putem face o descompunere in baza 2 si obtinem O(log(n)). Dar, putem sa facem in 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 doua intervale acopera complet intervalul nostru, deci putem face minimul / maximul lor si acesta este raspunsul la query.

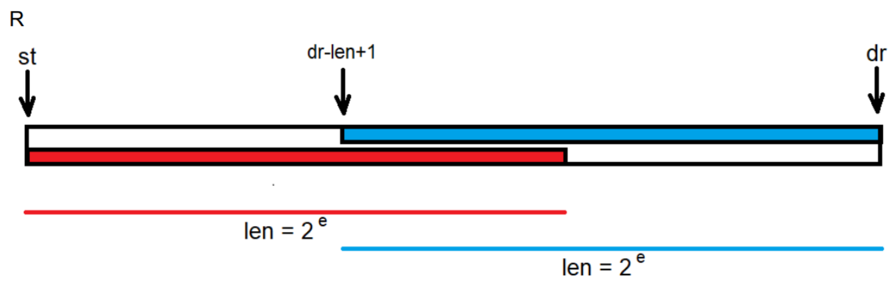


Figure 14: RMQ Query

12.1 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?

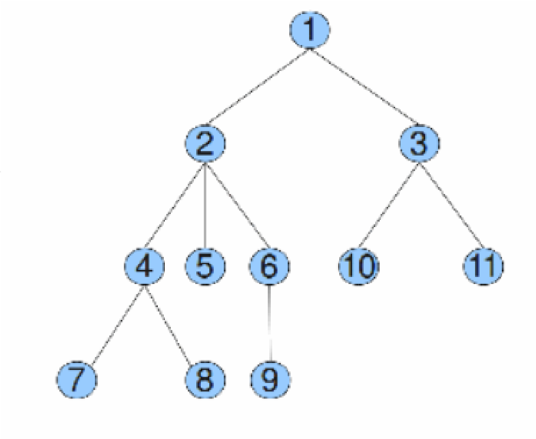


Figure 15: Tree Example

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

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. Obtinem $O(n)$ memorie suplimentara si $O(\sqrt{n})$ pe query.
- **Binary Lifting:** Pentru fiecare nod, țin tații de înălțime $1, 2, 4, 8, 16, \ldots$. 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.

12.2 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.

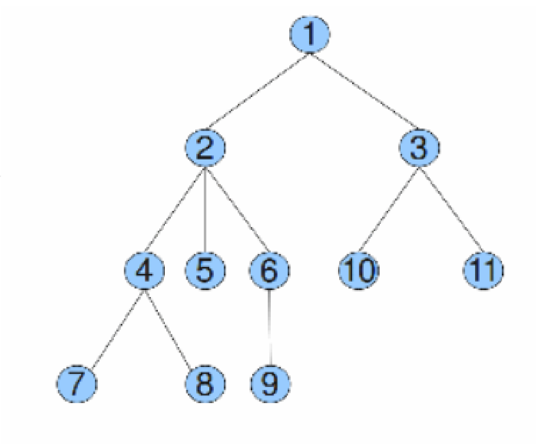


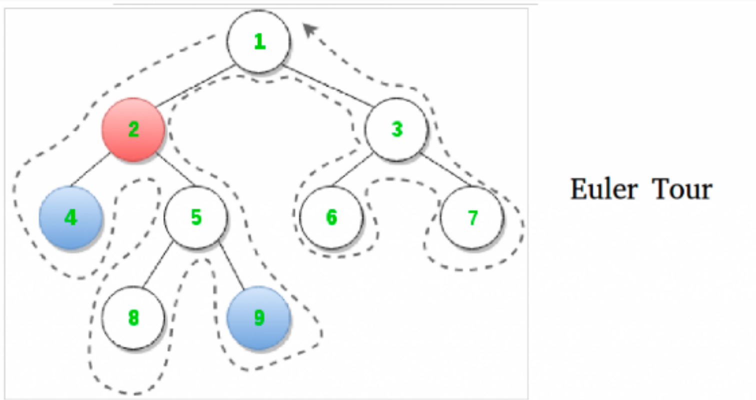
Figure 16: Tree Example

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

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

- **Liniarizam 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 calculeaza minimul pe vectorul de distante pana la radacina.
- Acum putem raspunde la query-uri $LCA(x,y)$ in $O(1)$.

Complexitate: $O(nlog(n) + q)$



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

Figure 17: Euler Tour

13 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? Solutii posibile:

- Hash-uri, $O(l)$ (unde l e lungimea cuvântului) pe query si $O(n \cdot l)$ 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? Solutii 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.

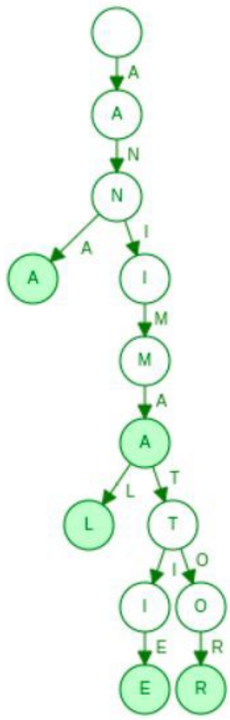


Figure 18: Trie cu cuvintele ana, animator, animație, animal, anima.

Inserare

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

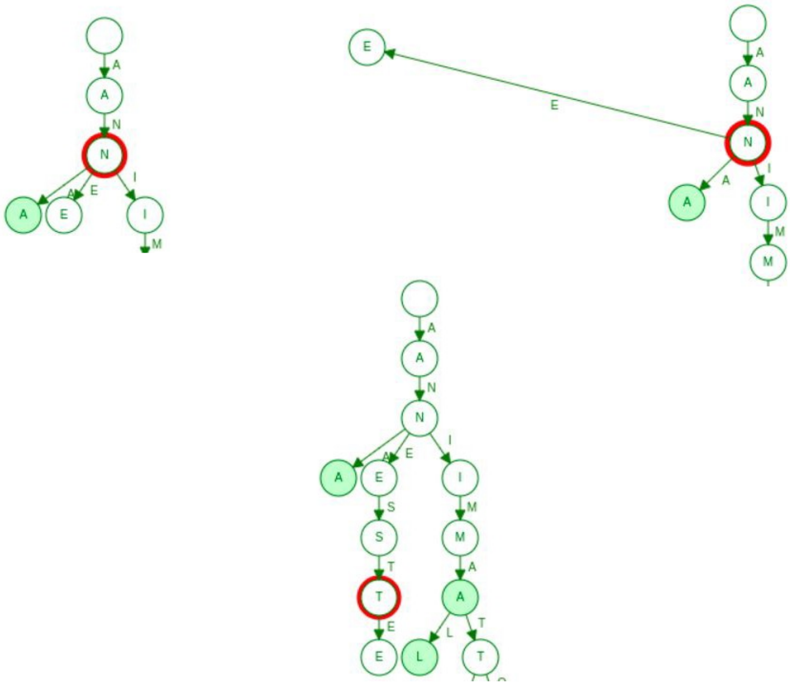


Figure 19: Inserare in trie

Cautare

Cautarea unui cuvant 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 lexicografica in 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.