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)

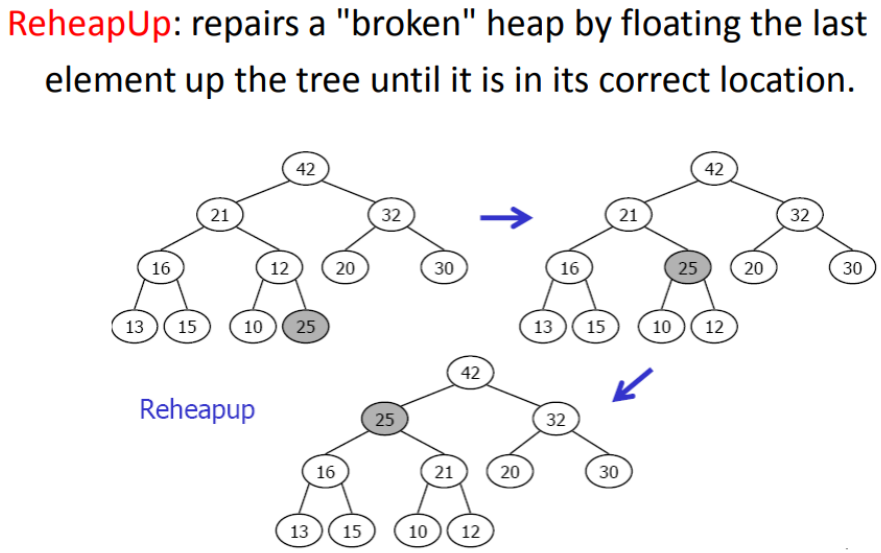


Figure 1: Urca

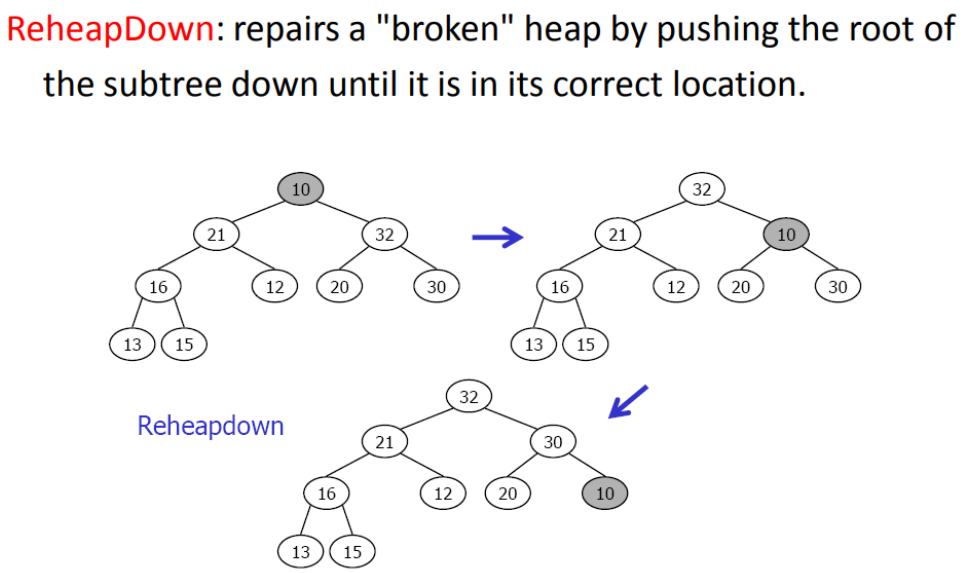


Figure 2: Coboara

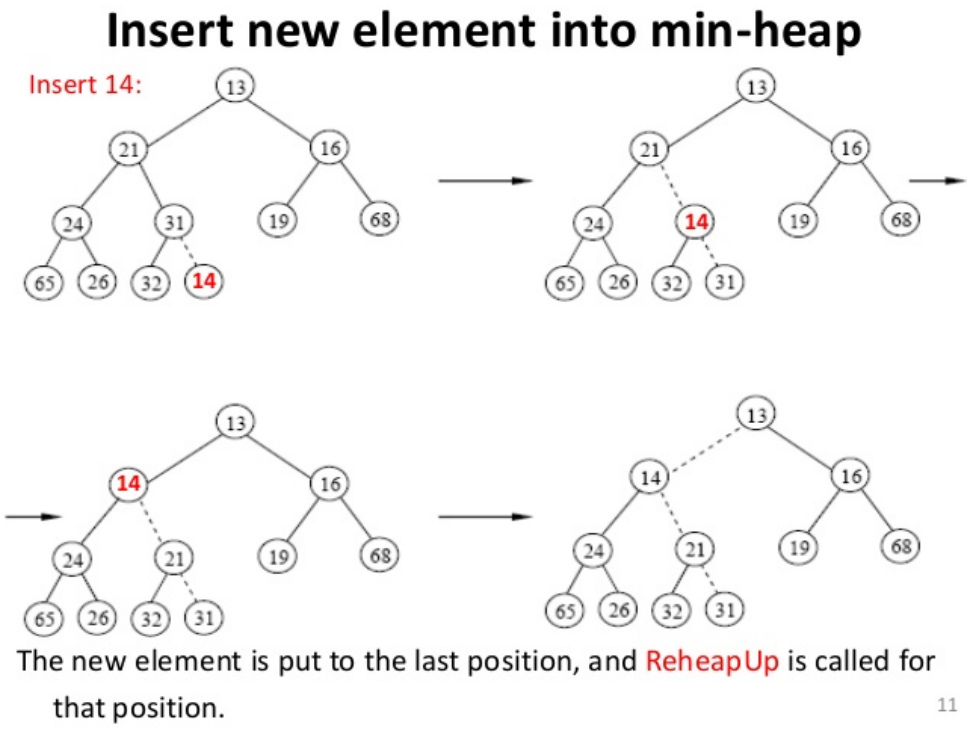


Figure 3: Insert in min-heap.

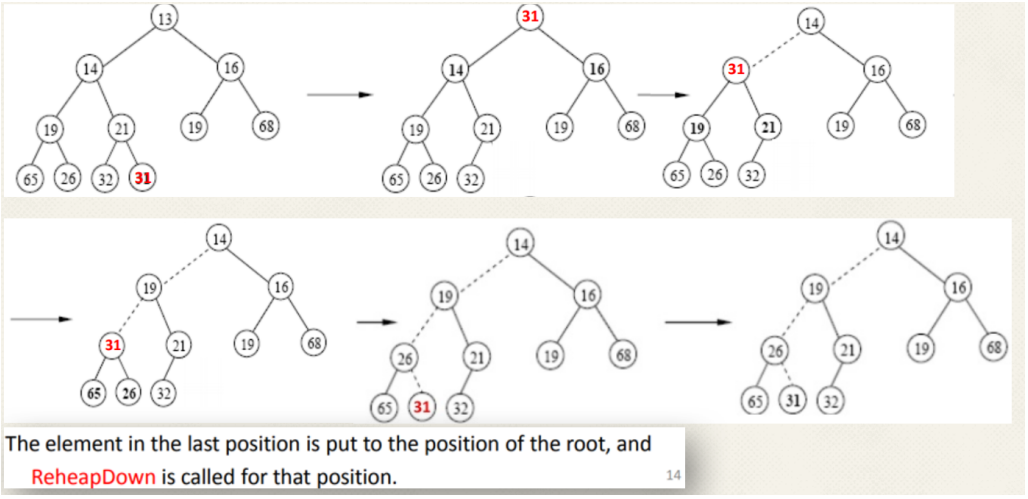


Figure 4: Delete root of the heap.

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).

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.

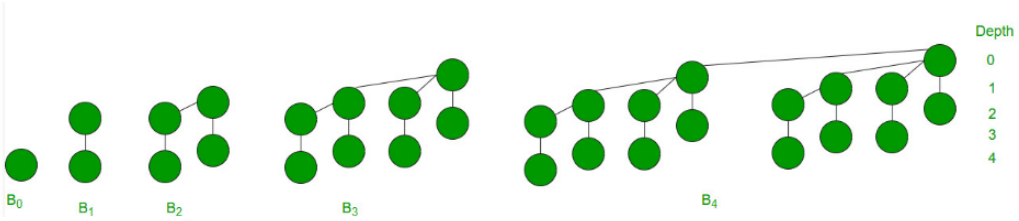


Figure 5: Arbori Binomiali

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.

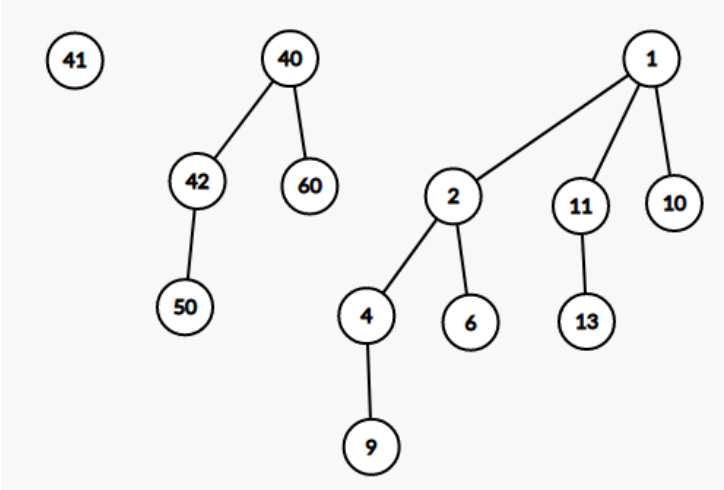


Figure 6: Heap Binomial

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(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).
- 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.

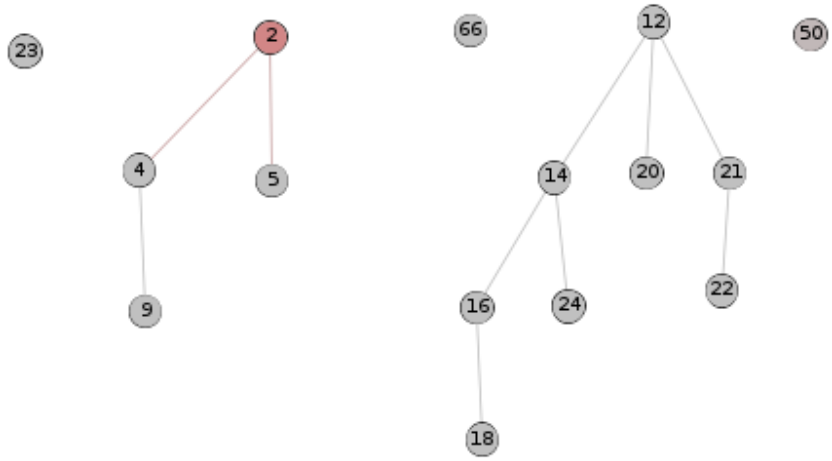


Figure 7: Fibonacci Heap

Structura unui heap fibonacci:

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

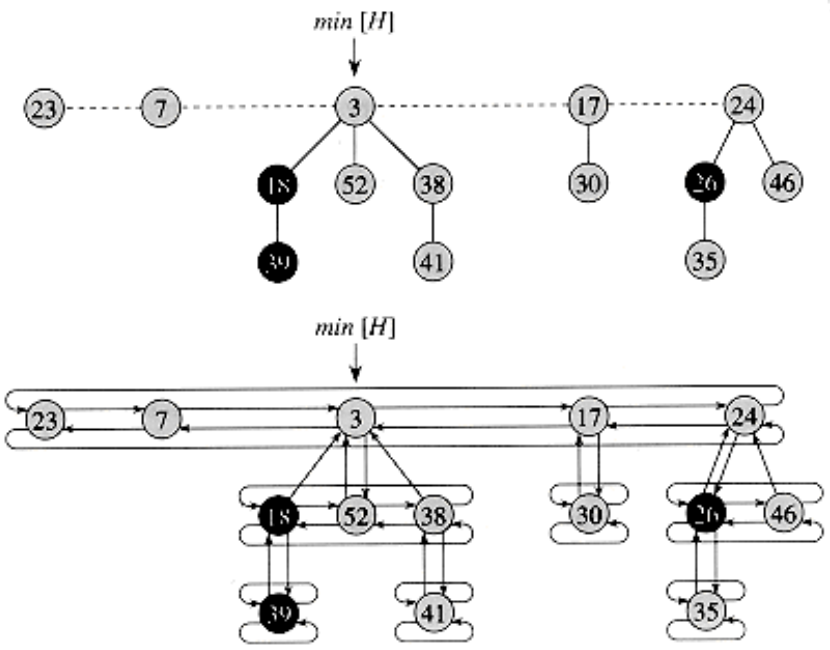


Figure 8: Fibonacci Heap Implementation

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.

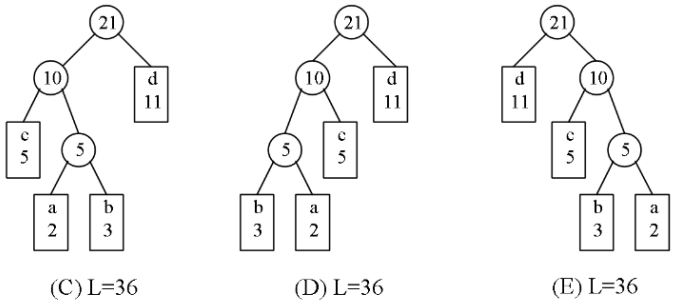
Stergere minim: Stergem minimul si ramanem cu doi arbori liberi (subarborii nodului sters). La extragerea minimului vom consolida heap-ul, adica se face reuniune similar cu heap-urile binomiale. **Consolidarea heap-ului**: Se calculeaza gradul unui arbore, ca fiind inaltimea arborelui si arborii cu acelasi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



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 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).

6.1 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

6.2 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.

7 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

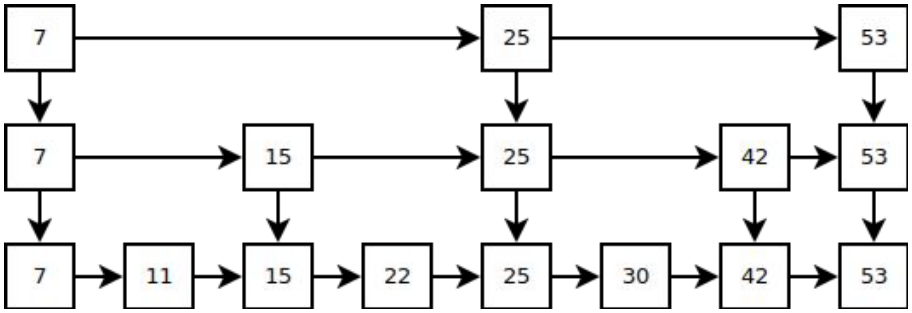


Figure 10: Skip List

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)

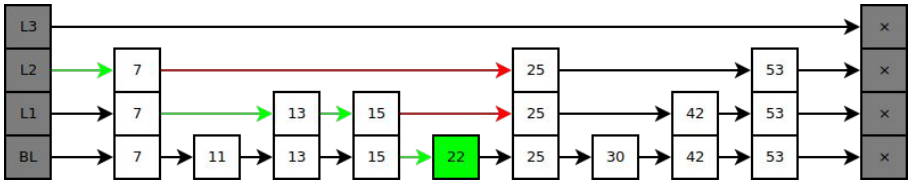


Figure 11: Cautare elementul 22

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)).

8 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.

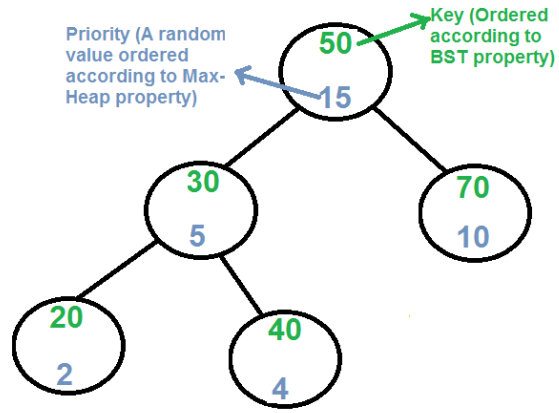


Figure 12: Treap

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).

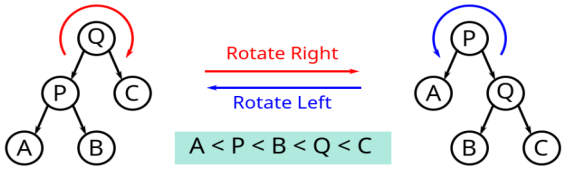


Figure 13: Rotatii pe arbori

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.