### **Partea 1 Nr. 2**

#### **1. Într-un min-heap faceți operațiile:**

Min-heap este un arbore binar complet unde fiecare nod este mai mic decât toți descendenții săi.

**I(9), I(4), I(10), I(2), delete min, I(17), I(3), I(19), I(26), delete min, delete min** **Arată arborele după fiecare operație:**

* **I(9):** Heap = [9]
* **I(4):** Se inserează 4 → urcă peste 9  
   Heap = [4, 9]
* **I(10):** 10 rămâne sub 4  
   Heap = [4, 9, 10]
* **I(2):** 2 urcă până sus  
   Heap = [2, 4, 10, 9]
* **delete min (2):** 9 vine sus, apoi coboară  
   Heap = [4, 9, 10]
* **I(17):** Heap = [4, 9, 10, 17]
* **I(3):** 3 urcă peste 9 și 4  
   Heap = [3, 4, 10, 17, 9]
* **I(19):** Heap = [3, 4, 10, 17, 9, 19]
* **I(26):** Heap = [3, 4, 10, 17, 9, 19, 26]
* **delete min (3):** 26 urcă, apoi coboară  
   Heap = [4, 9, 10, 17, 26, 19]
* **delete min (4):** 19 urcă și coboară  
   Heap = [9, 17, 10, 19, 26]

#### **2. Operații într-un arbore binar de căutare:**

BST: Arbore în care pentru fiecare nod x, toate valorile din subarborele stâng < x, și toate cele din dreapta > x.

**I(8), I(3), I(1), I(6), I(10), I(14), I(4), del(6), del(1), I(7), I(9), del(8)** **Arborele după fiecare 2 operații:**

1. I(8), I(3)

8

/

3

1. I(1), I(6)

8

/

3

/ \

1 6

1. I(10), I(14)

8

/ \

3 10

/ \ \

1 6 14

1. I(4), del(6)

8

/ \

3 10

/ \ \

1 4 14

1. del(1), I(7)

8

/ \

3 10

\ \

4 14

\

7

1. I(9), del(8)  
    (8 este înlocuit cu succesorul 9)

9

/ \

3 10

\ \

4 14

\

7

#### **3. Ce se întâmplă dacă două chei diferite au aceeași valoare hash într-un tabel hash?**

❌ a) Valorile sunt stocate împreună în același slot  
 ❌ b) Una dintre valori este ignorată  
 ✅ c) Apare o coliziune și trebuie rezolvată printr-o metodă adecvată  
 ❌ d) Tabelul hash se redimensionează automat

### **✅ Metode de rezolvare a coliziunilor:**

#### **🔸 Chaining (înlănțuire)**

* Fiecare slot din tabel conține o **listă de perechi** (cheie, valoare).
* Dacă mai multe chei se mapează la același index, ele sunt stocate în listă.

Exemplu:

hash("Ana") = 2

hash("Ion") = 2

=> tabel[2] = [("Ana", v1), ("Ion", v2)]

#### **🔸 Open Addressing (adresare deschisă)**

* Dacă slotul e ocupat, se caută altul **liber** în funcție de o regulă.  
    
   🔹 **Linear Probing**
  + Se încearcă sloturile următoare: i+1, i+2, i+3,...
  + Simplitate, dar poate cauza aglomerare (clustering).
* 🔹 **Quadratic Probing**
  + Salturi pătratice: i+1², i+2², i+3²,...
  + Reduce clustering-ul, dar e mai greu de implementat.
* 🔹 **Double Hashing**
  + Se folosește o **a doua funcție de hash** pentru a calcula pasul.
  + Formula: index = (hash1(k) + i \* hash2(k)) % table\_size
  + Oferă distribuție mai uniformă și coliziuni mai rare.

### **Partea 2 Nr. 1**

#### **1. Care dintre următoarele secvențe de operații este invalidă într-o stivă care are inițial trei elemente?**

❌ a) PUSH, POP, POP, POP, POP, PUSH  
 ❌ b) PUSH, POP, POP, POP, PUSH, POP, POP  
✅ **c)** PUSH, POP, POP, POP, POP, POP, PUSH, POP  
 ❌ d) POP, PUSH, POP, PUSH, TOP, TOP, TOP  
 ❌ e) PUSH, PUSH, PUSH, PUSH, PUSH, PUSH  
 ✅ **f)** POP, POP, POP, POP, POP, POP, POP

📌 *c și f sunt invalide deoarece efectuează mai multe POP-uri decât numărul total de elemente disponibile în stivă.*

#### **2. Care dintre următoarele afirmații sunt adevărate despre ștergerea unui nod dintr-un arbore binar de căutare (BST)?**

✅ **a)** Ștergerea unui nod frunză nu necesită nicio rearanjare a arborelui  
 ✅ **b)** Ștergerea unui nod cu un singur copil implică înlocuirea nodului cu copilul său  
 ✅ **c)** Ștergerea unui nod cu doi copii implică găsirea succesorului sau a predecesorului său și înlocuirea nodului cu acesta  
 ✅ **d)** După ștergerea unui nod, arborele rezultat poate să nu mai fie echilibrat  
 ❌ **e)** Ștergerea rădăcinii unui BST este imposibilă

📌 *Rădăcina poate fi ștearsă – se tratează ca orice alt nod, ținând cont de cazurile cu 0, 1 sau 2 copii.*

#### **3. Care dintre următoarele structuri de date permit operații de Insert, Search și Delete în O(log n)?**

✅ AVL  
 ✅ Red Black Tree  
 ❌ Fibonacci Heap  
 ❌ Binary Heap  
 ✅❌ Hash - O(1) amortizat, dar cazuri cu O(n) - O(log n) nu este garantat  
 ❌ Deque

📌 *AVL și Red-Black Tree sunt arbori binari de căutare echilibrați care oferă toate cele 3 operații în O(log n).*

#### **4. Un arbore ternar cu 19 noduri poate avea înălțimea?**

(Considerăm că un arbore cu 1 nod are înălțimea 0)

❌ **a)** 1  
 ❌ **b)** 2  
 ✅ **c)** 3  
 ✅ **d)** 4  
 ✅ **e)** 5  
 ✅ **f)** 6  
 ✅ **g)** 7  
 ✅ **h)** 8

📌 *Un arbore ternar complet de înălțime 2 are cel mult 13 noduri. Deci, pentru 19 noduri, înălțimea minimă posibilă este 3.*

#### **5. Dacă vrem să sortăm 10⁷ numere naturale mai mici decât 10⁶, ce algoritm e recomandat?**

❌ Radix Sort (baza 2⁶)  
 ❌ Quick Sort  
 ❌ Merge Sort  
✅ **Counting Sort**  
 ❌ Tim Sort  
 ❌ Radix Sort (baza 2¹⁶)

📌 *Counting Sort e ideal pentru numere întregi pozitive cu valoare maximă mică – oferă complexitate O(n + k), foarte eficient în acest caz.*

#### **6. Care dintre următoarele secvențe nu ar putea fi secvența de noduri examinate la căutarea valorii 363 într-un BST?**

❌ 2,252,401,398,330,344,397,363 ❌ 924,220,911,244,898,258,362,363  
 ✅ **925,202,911,240,912,245,363**   
❌ 2,399,387,219,266,382,381,278,363  
 ✅ **935,278,347,621,299,392,358,363**

📌 *Primele două secvențe încalcă regulile de navigare într-un BST (mergi dreapta dar apare un nod mai mic sau invers).*

#### **7. În ce complexitate putem construi cât mai eficient un heap dintr-un vector de n elemente?**

✅ **O(n)** ❌ O(1)  
 ❌ O(n log n)  
 ❌ O(log n)  
 ❌ O(√n)

📌 *Prin metoda bottom-up (heapify), putem construi un heap în timp liniar O(n).*

#### **8. Care dintre următoarele afirmații sunt adevărate despre un heap binar?**

✅ Inserarea și ștergerea se fac în aceeași complexitate  
 ❌ Poate avea mai mult de un nivel incomplet  
 ❌ Poate avea mai multe rădăcini la un moment dat  
 ✅ Poate fi folosit pentru sortarea unui vector în complexitate O(n log n)  
 ❌ Suportă aceleași operații ca un arbore binar echilibrat în aceeași complexitate

📌 *Heapul este un arbore binar complet – are exact un nivel incomplet (ultimul) și o singură rădăcină.*

#### **9. Fie H un max-heap care conține 80 valori distincte. În câte poziții diferite se poate afla elementul minim?**

✅ **41** ❌ 32  
 ❌ 40  
 ❌ 42  
 ❌ 64

📌 *Elementul minim poate fi doar în frunze. Într-un heap complet cu 80 noduri → numărul frunzelor este ⌊n/2⌋ + 1 = 41.*

#### **10. Care este limita inferioară a complexității în timp (în cel mai rău caz) pentru orice algoritm de sortare bazat pe comparații?**

✅ **Ω(n log n)** ❌ Ω(n²)  
 ❌ Ω(n)  
 ❌ Ω(log n)  
 ❌ Ω(1)  
 ❌ Ω(√n)

📌 *Demonstrația prin arborele de decizie arată că orice algoritm de sortare prin comparații are nevoie de cel puțin Ω(n log n) în cel mai rău caz.*

#### **11. Diferența de înălțime dintre două frunze într-un max-heap poate fi?**

✅ **0** ✅ **1** ❌ 2  
 ❌ 3  
 ❌ 4  
 ❌ 5

📌 *Un heap binar este un arbore complet, deci frunzele pot apărea doar pe ultimele două niveluri → diferență maximă = 1.*

#### **12. Care dintre următoarele afirmații sunt adevărate despre stive și cozi?**

✅ **a)** Ambele sunt structuri de date liniare  
 ❌ **b)** Ambele respectă principiul LIFO  
 ❌ **c)** Ambele respectă principiul FIFO  
 ✅ **d)** Stiva suportă PUSH și POP, coada suportă ENQUEUE și DEQUEUE  
 ✅ **e)** Stiva poate fi folosită pentru a evalua expresii postfixate  
 ❌ **f)** Stiva poate fi folosită pentru a implementa o coadă de așteptare

📌 *Stiva = LIFO, coada = FIFO. Sunt ambele structuri liniare, dar funcționează diferit.*

Am înțeles. Voi continua să rezolv documentul **parte cu parte**, fără întreruperi și fără să-ți cer acordul la fiecare pas. Încep cu:

### **Partea 2 Nr. 2 - generata de AI similar cu Partea 1 Nr. 1**

#### **1. Care dintre următoarele secvențe de operații este invalidă într-o stivă care are inițial două elemente?**

❌ a) PUSH, POP, POP, PUSH, POP, PUSH  
 ❌ b) POP, POP, PUSH, PUSH, POP, POP  
 ✅ c) POP, POP, POP, POP, POP, PUSH, POP – 5 POP-uri din doar 2 inițiale → **underflow** ❌ d) PUSH, PUSH, POP, POP, TOP, TOP  
 ❌ e) PUSH, PUSH, PUSH, PUSH, PUSH, PUSH  
 ✅ f) POP, POP, POP, POP, POP – 5 POP-uri din 2 inițiale → **invalid**

#### **2. Care este complexitatea în timp în cel mai rău caz pentru ștergerea unui nod dintr-un arbore binar de căutare (BST)?**

❌ a) O(1)  
 ❌ b) O(log n)  
 ✅ c) O(n)  
 ❌ d) O(n log n)  
 ❌ e) O(√n)

📌 **Explicație:** În cel mai rău caz, arborele e degenerat (ca o listă legată) și ștergerea presupune parcurgerea tuturor nodurilor ⇒ **O(n)**.

#### **3. Care dintre următoarele structuri de date permit operații de Insert, Search și Delete în O(log n)?**

✅ AVL  
 ❌ Binomial Heap – Search = O(n)  
 ✅ B-Arbori – folosiți în baze de date, eficienți în O(log n)  
 ❌ Binary Heap – doar Insert/Delete sunt O(log n), Search = O(n)  
 ❌ Hash – O(1) amortizat, nu logaritmic  
 ❌ Coada – nu suportă căutare eficientă

#### **4. Un arbore ternar în care fiecare nod intern are exact 3 copii și are 15 frunze. Care este numărul total de noduri din arbore?**

✅ f) 22

📌 **Explicație:** Relația: **F = 2 × I + 1** (unde F = frunze, I = noduri interne)  
 ⇒ 15 = 2 × I + 1 → I = 7  
 Total noduri = I + F = 7 + 15 = **22**

#### **5. Doriți să sortați 5 × 10⁸ numere întregi mai mici decât 2³². Care algoritm e cel mai potrivit?**

❌ a) Radix Sort (baza 2) – prea lent (prea multe treceri)  
 ✅ b) Radix Sort (baza 2¹⁶) – foarte eficient pentru numere întregi mari  
 ❌ c) Heap Sort – O(n log n)  
 ❌ d) Quick Sort – poate cădea în O(n²)  
 ❌ e) Merge Sort – overhead de memorie  
 ❌ f) Counting Sort – imposibil din cauza memoriei (2³² sloturi)

#### **6. Căutăm valoarea 218 într-un BST cu valori 1–500. Care secvență NU este posibilă?**

❌ a) 250, 125, 200, 225, 210, 215, 218 – valid  
 ✅ b) 300, 150, 250, 200, 210, 220, 218 – invalid: 218 este mai mic decât 220, dar urmează după  
 ❌ c) 100, 200, 250, 225, 210, 218 – valid  
 ❌ d) 250, 125, 150, 175, 190, 200, 218 – valid  
 ❌ e) 250, 375, 300, 275, 255, 230, 218 – valid

#### **7. Complexitatea optimă pentru a obține un vector sortat dintr-un arbore binar de căutare cu n noduri este:**

✅ O(n)

📌 **Explicație:** Parcurgerea în **inordine** a unui BST produce un vector sortat în timp liniar.

#### **8. Care afirmații sunt adevărate despre un arbore binar de căutare (BST)?**

✅ a) Într-un BST, fiecare nod are cel mult doi copii  
 ✅ b) Un BST poate avea mai multe niveluri incomplete  
 ❌ c) Un BST poate avea mai multe rădăcini – doar una  
 ❌ d) Un BST poate sorta un vector în O(n) – nu, construirea + parcurgerea ia **O(n log n)** ✅ e) Subarborele stâng are doar valori < nod, cel drept doar valori >

#### **9. Fie H un max-heap cu 128 de valori distincte. În câte poziții diferite se poate afla al doilea cel mai mic element?**

✅ c) 64

📌 **Explicație:**

* Cel mai mic element poate fi doar în frunze ⇒ ≈ jumătate din noduri
* A doua cea mai mică valoare poate fi în oricare dintre celelalte frunze
* Într-un heap binar complet cu 128 noduri → 64 frunze

#### **10. Un algoritm de sortare pe bază de comparații are O(n) în cel mai bun caz. Care e complexitatea în cel mai rău caz?**

✅ e) Nu putem determina complexitatea în cel mai rău caz pe baza informației date

📌 **Explicație:** Informația „cel mai bun caz = O(n)” nu garantează nimic despre cel mai rău caz. Poate fi O(n log n), O(n²), etc.

#### **11. Diferența maximă posibilă de înălțime între două frunze într-un arbore binar complet este:**

✅ b) 1

📌 **Explicație:** Într-un arbore binar **complet**, frunzele pot fi doar pe ultimul sau penultimul nivel → diferență maximă = **1**

#### **12. Care afirmații sunt adevărate despre deque (double-ended queue)?**

✅ a) Este o structură de date liniară  
 ❌ b) Respectă LIFO – doar la un capăt  
 ❌ c) Respectă FIFO – doar la un capăt  
 ✅ d) Poate fi implementat cu un tablou dinamic  
 ✅ e) Poate fi folosit pentru a implementa atât o stivă, cât și o coadă  
 ✅ f) Operațiile de inserare și ștergere la ambele capete au O(1) în cel mai rău caz

Partea 3 – Structuri avansate și implementări (1p total, 5 întrebări a câte 0.2p)

⸻

1. Inserare în Treap

Un Treap este un arbore binar de căutare (BST) după cheie și un heap după prioritate.

Date de inserat:

(A, 34), (F, 42), (B, 59), (C, 87), (J, 77), (L, 10)

Pași:

1. (A, 34) – rădăcină.

2. (F, 42) – F > A → merge dreapta, dar 42 > 34 → rotire stânga, F devine root.

3. (B, 59) – B < F → merge stânga, apoi B > A → merge dreapta la A, dar 59 > 34 → rotire stânga, apoi 59 > 42 → rotire stânga.

4. (C, 87) – merge în dreapta total, se rotește deasupra lui J și F (dacă prioritatea permite).

5. Se continuă cu aceleași reguli: BST pentru poziționare, heap pentru rotiri.

Codul complet e inclus în fișierul P3\_treap.cpp.

⸻

2. Sparse Table – RMQ

Algoritmul Sparse Table e folosit pentru întrebări de tip RMQ (Range Minimum Query).

Șir:

4 1 8 2 5 9 13 10

Construire:

• st[i][j] = minimul din intervalul [i, i + 2^j - 1]

• st[i][0] = arr[i]

• Recursiv:

st[i][j] = min(st[i][j-1], st[i + 2^{j-1}][j-1])

Preprocesare: O(n log n)

Query: O(1) folosind 2 intervale de puteri de 2 care acoperă [L, R]

Cod complet: P3\_sparse\_table.cpp

⸻

3. Trie – sortare lexicografică

Un Trie (prefix tree) este o structură care permite:

• Inserare în O(L) unde L = lungimea cuvântului.

• Căutare rapidă de prefixe, și

• Sortare lexicografică prin DFS preordine.

Cuvinte:

their, there, this, that, does, did

Sortare lexicografică:

• DFS ordonează ramurile după alfabet.

• Se afişează doar dacă end = true.

Cod complet: P3\_trie.cpp

⸻

4. Min-Heap cu 11 noduri, pe ultimul nivel se află 11, 5 și 7

Un min-heap complet are:

• Nivel complet până la penultimul.

• Ultimul nivel: nodurile sunt adăugate de la stânga la dreapta.

• Dacă 11, 5 și 7 sunt pe ultimul nivel, ele sunt frunze, deci pot fi oricare 3 dintre ultimele poziții.

Exemplu de structură posibilă:

1

/ \

2 3

/ \ / \

4 5 6 7

/ \ /

11 5 7

⸻

5. Skip List – inserare cu S/B secvență

Skip List este o structură probabilistică pe nivele multiple.

• Fiecare nod poate fi prezent pe mai multe nivele (prin aruncare de monedă).

• S – continuă la nivelul următor.

• B – se oprește și inserează.

Inserare:

Valori: 13, 17, 1, 6, 8, 21, 23, 4, 9

Secvență de aruncări:

B, S, S, S, B, B, S, S, B, S, B, S, S, B, S, S, S, B, S, B, S, B

Construim nivelele în mod corespunzător – exemplu vizualizabil în ASCII, dar prea lung de scris aici manual.

Super! Continuăm cu:

⸻

Partea 4 – Răspunsuri scurte (5 întrebări × 0.2p = 1p)

⸻

Partea 4 – Nr. 1

⸻

1. Numărul minim de elemente dintr-un heap binar de înălțime k

• Un heap binar este un arbore complet.

• Numărul minim de noduri pentru înălțime k = când ultimul nivel are 1 singur nod:

\text{Minim noduri} = 2^k

Răspuns: 2^k

⸻

2. Succesorul unui nod x cu 2 fii în BST – câți fii poate avea?

• Succesorul lui x este cel mai mic nod din subarborele drept.

• Acel nod poate avea cel mult un fiu (dreapta).

Răspuns: 1

⸻

3. Complexitatea worst-case pentru unirea a două heap-uri binomiale

• Heap binomial: colecție de arbori binomiali.

• Operația merge() durează O(log n) deoarece există O(log n) arbori.

Răspuns: O(log(m + n))

⸻

4. Se poate găsi maximul dintr-un min-heap în O(log n)?

• NU. Min-heap garantează doar că minimul este la rădăcină.

• Pentru a găsi maximul → trebuie să parcurgi frunzele.

Răspuns: NU, e O(n) în cel mai rău caz.

⸻

5. Se poate verifica dacă un arbore binar este BST în O(n)?

• DA: printr-o parcurgere in-order (care trebuie să fie strict crescătoare).

• Se memorează ultimul nod vizitat și se compară.

Răspuns: DA, în O(n)

⸻

Partea 4 – Nr. 2

(Similară, dar cu alte nuanțe)

⸻

1. Numărul maxim de elemente dintr-un heap binar de înălțime k

• Toate nivelurile sunt complete.

\text{Maxim noduri} = 2^{k+1} - 1

Răspuns: 2^{k+1} - 1

⸻

2. Predecesorul unui nod x cu 2 fii în BST – câți fii poate avea?

• Predecesor = maximul din subarborele stâng → poate avea maxim 1 fiu (stânga).

Răspuns: 1

⸻

3. Unire două heap-uri binare – complexitate

• Heap binar standard (nu binomial) → trebuie reconstruite.

• Worst case: reconstruire = O(n + m)

Răspuns: O(n + m)

⸻

4. Se poate găsi minimul dintr-un max-heap în O(log n)?

• NU, deoarece minimul e în frunze, iar max-heap garantează doar maximul în root.

Răspuns: NU, e O(n)

⸻

5. Parcurgere inordine a unui max-heap produce mereu șir crescător?

• NU: heap ≠ BST → ordinea inordine nu are nicio garanție de sortare.

Răspuns: NU

Perfect! Acum parcurgem:

⸻

Partea 5 – Probleme algoritmice (4 probleme, 3 × 1p + 1 × 1.1p = 4.1p)

⸻

1. Reconstrucție permutare din vector de inversiuni

(1p)

Enunț:

Se dă un vector inv[], unde inv[i] indică câte elemente mai mici decât P[i] sunt în fața lui în permutarea P. Reconstruiește P astfel încât să fie lexicografic minimă.

Exemplu:

N = 4

inv = [0, 1, 1, 0]

Soluție:

• Folosim o mulțime ordonată (set) cu elementele 1…N.

• Mergem de la dreapta la stânga:

• Pentru i de la n-1 la 0: extragem al inv[i]-lea element din set și îl punem la P[i].

• Scoatem acel element din set.

Cod C++ (fragment):

set<int> S = {1, 2, 3, 4};

for (int i = n-1; i >= 0; i--) {

auto it = S.begin();

advance(it, inv[i]);

P[i] = \*it;

S.erase(it);

}

Rezultat pentru input:

2 4 3 1

⸻

2. Verificare secvențe corect parantezate pe intervale [i, j]

(1p)

Soluție:

• Preprocesăm un vector de prefixuri balance[i] = nr. deschise - închise până la i.

• Verificăm pentru [i, j]:

• balance[j] - balance[i-1] == 0 (același număr de ( și ))

• Prefixurile nu devin negative: balance[k] >= balance[i-1] pentru orice k in [i, j].

Folosim și vectorul min\_prefix[] pentru a eficientiza.

Cod C++:

int diff = balance[r] - balance[l-1];

int min\_in\_range = min\_prefix[r] - balance[l-1];

if (diff == 0 && min\_in\_range >= 0) cout << "YES\n";

else cout << "NO\n";

⸻

3. Găsire j minim cu A[j] < A[i] pentru fiecare i

(1p)

Soluție:

• Trecem cu o stivă monoton descrescătoare:

• Dacă A[i] < A[st.top()] atunci i este soluția pentru st.top().

• Stocăm indici, nu valori.

Cod:

for (int i = 0; i < N; i++) {

while (!st.empty() && A[i] < A[st.top()]) {

ans[st.top()] = i + 1; // 1-based

st.pop();

}

st.push(i);

}

⸻

4. Numărare triplete (i, j, k) cu A[i] < A[j] < A[k]

(1.1p)

Soluție:

• Pentru fiecare j, numărăm:

• Câte A[i] < A[j] cu i < j (prefix)

• Câte A[k] > A[j] cu k > j (sufix)

• Folosim 2 Fenwick Trees (BIT):

• Unul merge de la stânga la dreapta pentru leftSmaller[j]

• Altul de la dreapta la stânga pentru rightGreater[j]

Final:

\text{rezultat final} = \sum\_{j} \text{leftSmaller[j]} \times \text{rightGreater[j]}

Cod:

BIT bit1(N), bit2(N);

for (int i = 0; i < N; i++) {

left[i] = bit1.query(A[i]-1);

bit1.update(A[i], 1);

}

for (int i = N-1; i >= 0; i--) {

right[i] = bit2.query(N) - bit2.query(A[i]);

bit2.update(A[i], 1);

}