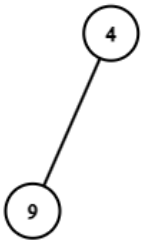
Subiectul 2

Partea 1

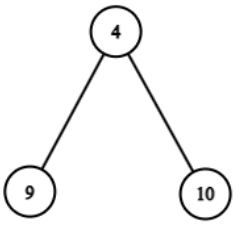
1. **Într-un min-heap faceți operațiile I(9), I(4), I(10), I(2), delete min, I(17), I(3), I(19), I(26) delete min, delete min. Arată arborele după fiecare operație.**

I(9)

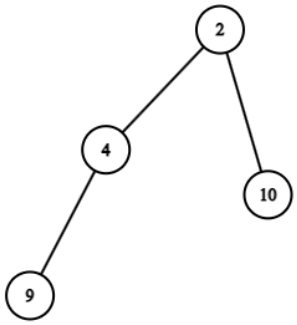
I(4)



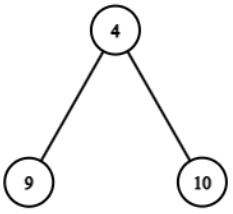
I(10)



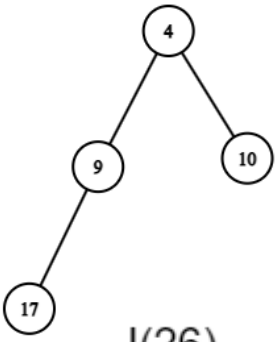
I(2)



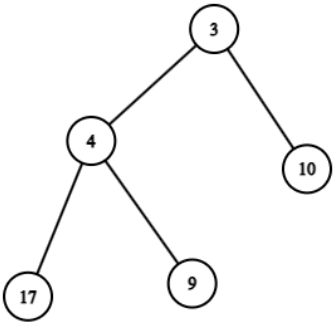
delete min



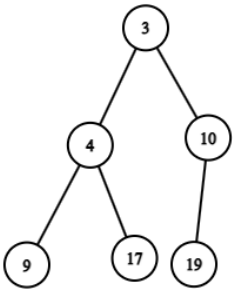
I(17)



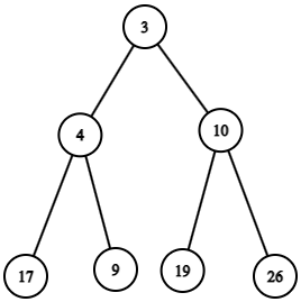
I(3)



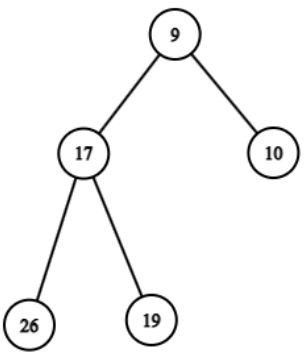
I(19)



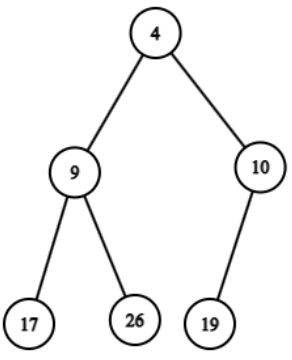
I(26)



delete min

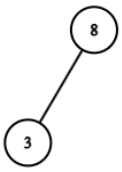


delete min

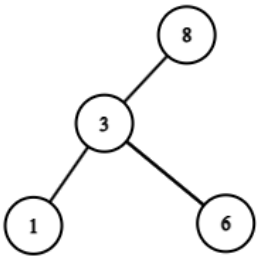


2. **Într-un arbore binar de căutare faceți operațiile I(8), I(3), I(1), I(6), I(10), I(14), I(4), del(6), del(1), I(7), I(9), del(8). Aratati arborele după fiecare 2 operații.**

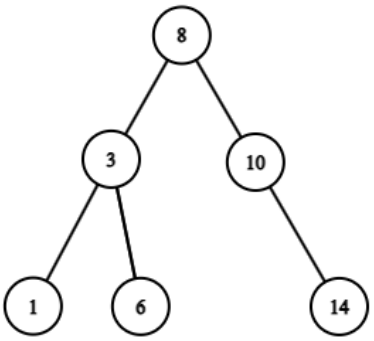
I(8), I(3)



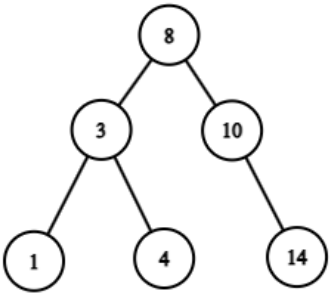
I(1), I(6)



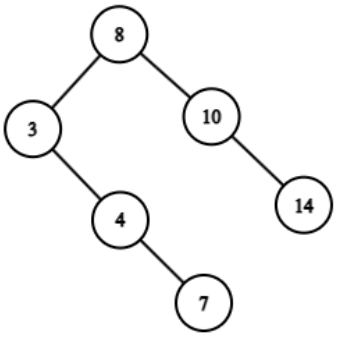
I(10), I(14)



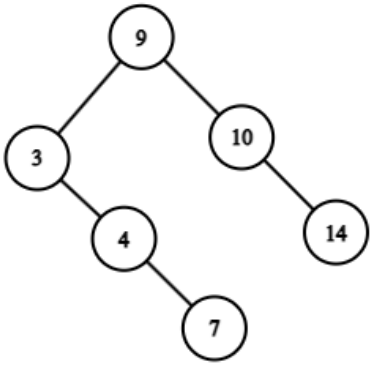
I(4), del(6)



del(1), I(7)



I(9), del(8)



3. **Ce se întâmplă dacă două chei diferite au aceeași valoare hash într-un tabel hash?**
    - **Răspuns corect: c) Apare o coliziune și trebuie rezolvată printr-o metodă adecvată.**
    - **Explicație:** Când două chei diferite au aceeași valoare hash, se produce o coliziune care necesită o tehnică de rezolvare, cum ar fi chaining sau open addressing.

Partea 2:

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

- **Răspuns corect: c), f)**

- **Explicație:** Într-o stivă, operațiile POP nu pot fi efectuate dacă stiva este goală, ceea ce face ca c) și f) să fie secvențe invalide.

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

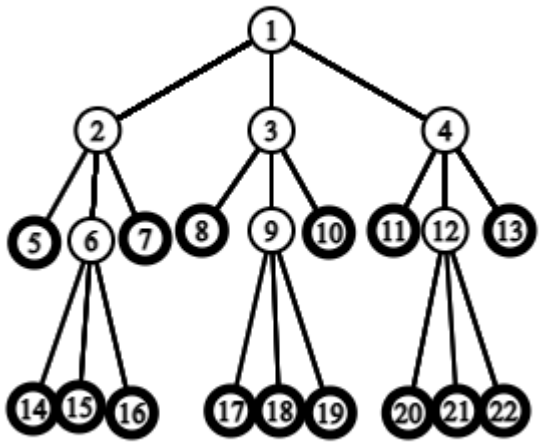
- **Răspuns corect: c) O(n)**
- **Explicație:** În cel mai rău caz, arborele poate fi degenerat (similar cu o listă), caz în care complexitatea pentru ștergere este liniară, O(n).

6. Care din următoarele structuri de date permit operații de Insert, Search, și Delete în O(log n)?

- **Răspuns corect: a), c), e)**
- **De ce nu sunt corecte celelalte opțiuni:**
  - b) Binomial Heap: permite operații în O(log n) doar pentru unele operații, nu pentru toate trei.
  - d) Binary Heap: are operații de căutare ineficiente (O(n)).
  - f) Coada: operațiile de căutare și ștergere sunt ineficiente (O(n)).

7. Un arbore ternar în care fiecare nod intern are exact trei copii, are 15 frunze. Care este numărul total de noduri din arbore?

- **Răspuns corect: f) 22 sau b) 22**
- **Explicație:** 1(primul nivel)+3(al doilea nivel)+9(al treilea nivel)+9(al patrulea nivel)=22



Nodurile evidentiate sunt cele 15 frunze.

8. Doriți să sortați 5 x 10^8 numere întregi mai mici decât 2^32. Care algoritm de sortare ar fi cel mai potrivit să utilizați?

- **Răspuns corect: b) Radix Sort (baza 2^16)** (Complexitate O(log(baza 2^16)(max)*(n+baza)), n=5*10^8, max=2^32)
- **De ce nu sunt corecte celelalte opțiuni:**
    - a) Radix Sort (baza 2): mai lent din cauza numărului mai mare de pași.Complexitate O(log(baza 2)(max)*(n+baza))*
    - c) Heap Sort: are complexitate O(n log n) și este mai lent pentru seturi mari de date.
    - d) Quick Sort: în cel mai rău caz O(n^2), deși în medie O(n log n).
    - e) Merge Sort: O(n log n) și necesită spațiu suplimentar.
    - f) Counting Sort: O(max+n)

9. Care dintre următoarele secvențe NU ar putea fi secvența de noduri examinate?

- **Răspuns corect: e) 250, 375, 300, 275, 255, 230, 218**
- **Explicație:** Secvența e) nu respectă proprietatea de ordine a unui arbore binar de căutare. Initial, merge pe partea dreapta a arborelui la numerele mai mari de 250, spre final ajunge la 230 care e mai mic de 250.

10. Dat fiind un arbore binar de căutare cu n noduri, care este complexitatea în timp optimă pentru a obține un vector sortat în ordine crescătoare conținând toate elementele din arbore?

- **Răspuns corect: b) O(n)**
- **Explicație:** Traversarea în ordine (inorder traversal) a unui arbore binar de căutare dă un vector sortat în timp liniar, O(n).

11. Care dintre următoarele afirmații sunt adevărate despre un arbore binar de căutare (BST)?

- **Răspuns corect: a), b), e)**
- **De ce nu sunt corecte celelalte opțiuni:**
    - c) Un BST poate avea mai multe rădăcini la un moment dat: incorect, un BST are o singură rădăcină.
    - d) Un BST poate fi folosit pentru sortarea unui vector în complexitate O(n): incorect, sortarea are O(n log n) în cel mai bun caz.

12. Fie H un max-heap care conține 128 de valori distincte. În câte poziții diferite se poate afla al doilea cel mai mic element?

- **Răspuns corect: d) 65**
- **Explicație:** Într-un max-heap, al doilea cel mai mic element poate fi în pozițiile penultimului sau ultimului nivel(pe toate mai putin una, cea a celui mai mic numar), adică 65 de poziții.

13. Se dă un algoritm de sortare care folosește doar comparații între elemente. În cel mai bun caz, acest algoritm sortează un șir de n elemente în timp liniar O(n). Care dintre următoarele afirmații este adevărată despre complexitatea în timp în cel mai rău caz a acestui algoritm?

- **Răspuns corect: e) Nu putem determina complexitatea în cel mai rău caz pe baza informației date.**
- **Explicație:** Informația că algoritmul sortează în timp O(n) în cel mai bun caz nu oferă suficiente detalii pentru a determina complexitatea în cel mai rău caz. Depinde de implementare si de tipul de date, putand fi si mai mare de O(n^2).

14. Care este diferența maximă posibilă de înălțime dintre două noduri frunză într-un arbore binar complet?

- **Răspuns corect: b) 1**
- **Explicație:** Într-un arbore binar complet, toate frunzele sunt pe ultimul sau penultimul nivel, deci diferența maximă de înălțime este 1.

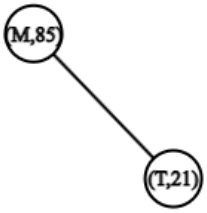
15. Care dintre următoarele afirmații sunt adevărate despre deque (double-ended queue)?

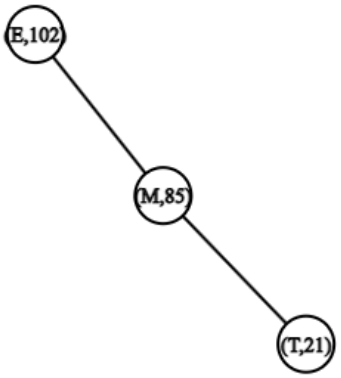
- **Răspuns corect: a), d), e), f)**
- **De ce nu sunt corecte celelalte opțiuni:**
  - b) Respectă principiul LIFO (Last In, First Out): incorect, deoarece deque permite operații la ambele capete.
  - c) Respectă principiul FIFO (First In, First Out): incorect, deoarece deque permite operații la ambele capete.

Partea 3

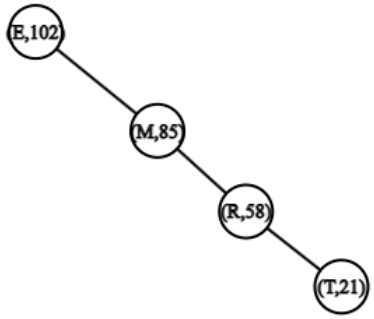
16. Inserați următoarele chei și priorități într-un treap(cu heap de maxim): (M, 85), (T, 21), (E, 102), (R, 58), (P, 16), (S, 93). Aratati arborele după fiecare inserare.

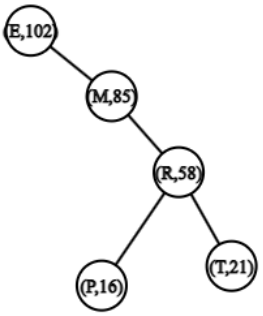
(M, 85) **(T, 21)** **(E, 102)**

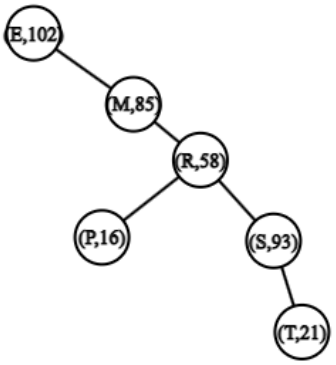




(R, 58) **(P, 16)** **(S, 93)**







17. Construiți sparse table-ul (matricea din algoritmul RMQ) pentru șirul: 7, 3, 9, 1, 4, 6, 10, 2. Presupunem că în cadrul unui query am vrea să determinăm minimul pe un interval.

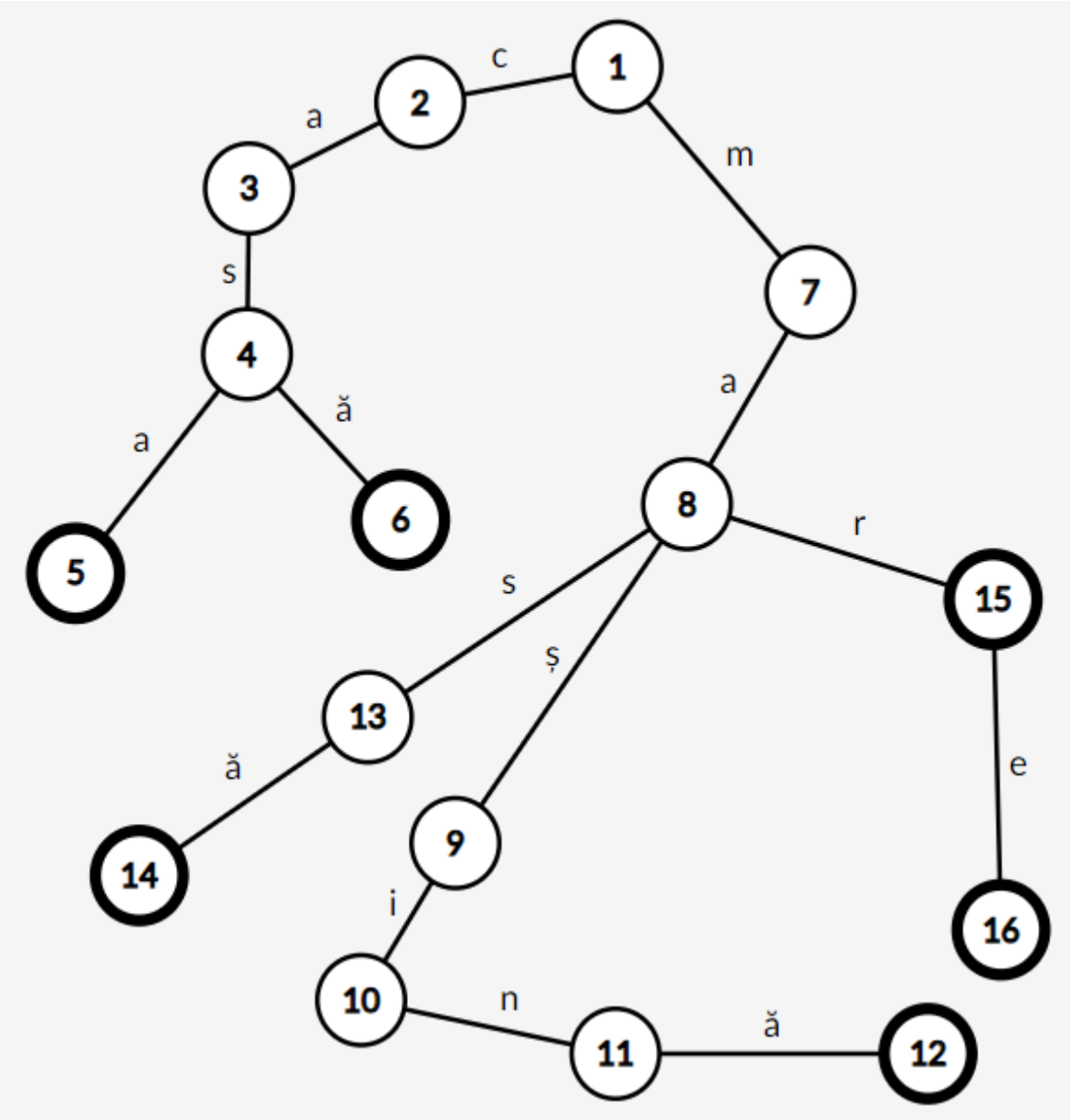
7 3 9 1 4 6 10 2

3 3 1 1 4 6 2

1 1 1 1 2

1

18. Inserați într-un trie cuvintele: casa, casă, mașină, masă, mare, mar. Explicați cum puteți găsi prefixul maxim comun al cuvântului "cascada" cu oricare cuvânt din lista inițială folosind trie-ul construit.

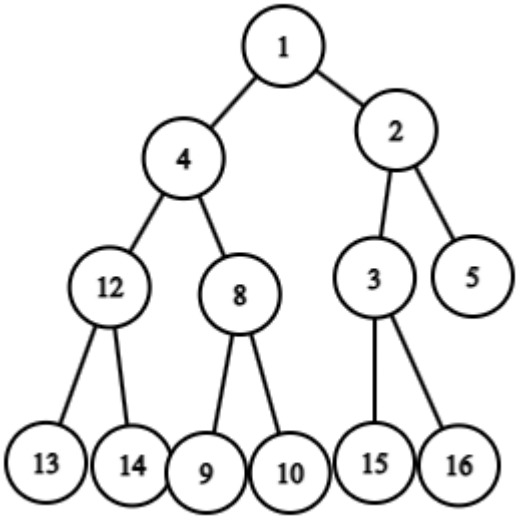


nodurile incercuite cu negru sunt terminale

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.

19. Construiți un min-heap cu 13 noduri în care pe penultimul nivel se află printre altele valorile 12, 8 și 3.



20. Inserați în skip list următoarele valori: 1, 9, 3, 14, 6, 11, 19, 15, 22 La aruncarea monedei obțineți următoarele valori: B, S, S, B, S, B, S, S, B, S, S, S, S, B, S, B, S, S, B, B, S, B. Când obțineți B vă opriți și inserați până la acel nivel. Când obțineți S continuați la nivelul următor.

Level 4: 6

Level 3: 6

Level 2: 6-----9----------14----------19

Level 1: 3-----6-----9-----11-----14----------19------22

Level 0: 1-----3-----6-----9-----11-----14-----15-----19-----22

Partea 4

21. Care este numărul maxim de elemente dintr-un heap binar de înălțime k? Vom presupune ca rădăcina heap-ului se afla la înălțime 0.

- **Răspuns corect: 2^(k+1)-1**
- **Explicație:** Un heap binar complet de înălțime k are 2^(k+1)-1 elemente. 1+2^1+2^2+...+2^k = 2^(k+1)-1

22. Fie T un arbore binar de căutare și x un nod din T cu 2 fii. Care este numărul maxim de fii pe care îi poate avea predecesorul lui x?

- **Răspuns corect: 1**
- **Explicație:** Predecesorul lui x într-un BST este in arborele stang a lui x, maxim in dreapta, deci nu poate avea copil drept. Asadar, poate avea maxim 1 fiu, pe cel stang.

23. Care este complexitatea în timp, în cel mai rău caz, pentru a uni (merge) două heap-uri binare?

- **Răspuns corect: O(m+n)**
- **Explicație:** Combinarea a două heap-uri binare are complexitatea O(m+n) deoarece putem face comparatia de complexitate intre task-ul de a interclasa doi vectori sortati care are complexitate o(m + n) si task-ul de a combina doua heap-uri binare.

24. Se poate găsi elementul minim dintr-un max-heap în O(log(n))?

- **Răspuns corect: Nu**
- **Explicație:** Elementul minim într-un max-heap pe orice frunza, necesitând parcurgerea a cel puțin n/2 elemente.

25. O parcurgere in inordine a unui max-heap produce mereu un șir crescător?

- **Răspuns corect: Nu**
- **Explicație:** Parcurgerea inordine a unui max-heap nu produce un șir crescător deoarece heap-ul nu respectă ordinea BST.

Partea 5

26) Fie P o permutare cu n elemente. Scopul este sa determinam aceasta permutare știind pentru fiecare poziție i (1 <= i <= n) cate elemente aflate înaintea ei sunt mai mici decat P[i]. Dacă exista mai multe permutări posibile, se va afla cea minim lexicografica. Input: N = 4 0 1 1 0 Output: 2 4 3 1

Solutie:

Fie cnt[i] = vectorul citit în datele de intrare, și p[i] = permutarea pe care trebuie să o aflăm. În primul rând, soluția, dacă există, este unică, iar condițiile de existență sunt că 0 <= cnt[i] <= i - 1 pentru orice i de la 1 la n. Dacă condiția menționată anterior nu se respectă, soluția nu există.

Presupunem ca solutia exista. Vom afla pe rând elementele în ordine descrescătoare a indicilor (de la n la 1). Ultimul element poate fi găsit instantaneu, acesta fiind egal cu cnt[n] + 1.

Inițial, mulțimea elementelor posibile pe care acestea le pot pune pe pozitia curenta este egală cu {1, 2, 3, …, n - 1, n}, iar după ce aflăm că p[n] = cnt[n] + 1, voi șterge acest element din mulțime (deoarepe p este o permutare si la un pas ulterior nu voi putea pune din nou alelasi element). Pentru valoarea n - 1, trebuie să găsim al 1 + cnt[n-1] -lea cel mai mic element din mulțimea rămasă, și după aceea să ștergem p[n-1]. Deci, procesul este următorul:

Notăm cu S(i) = mulțimea elementelor de la pasul i din proces. Inițial S(n) = {1, 2, 3, …, n - 1, n} iar mai apoi avem că p[i] = al 1 + cnt[i] - lea element din mulțimea S(i), iar S(i - 1) = S(i) / {p[i]}. De aici ne dăm seama și de unde vine condiția de existență și de ce soluția este unică. Procesul poate fi simulat cu un AINT și utilizând căutarea binară pe AINT în log(n) în felul următor: inițial facem: addAint(v = 1, tl = 1, tr = n, pos = 1, val = +1), addAint(v = 1, tl = 1, tr = n, pos = 2, val = +1), …, addAint(v = 1, tl = 1, tr = n, pos = n - 1, val = +1), addAint(v = 1, tl = 1, tr = n, pos = n, val = +1). Iar pe parcurs ce vrem să găsim al k-lea element din mulțimea rămasă trebuie doar să căutăm binar până unde suma elementelor este <= k - 1, iar elementul nostru este fix următorul. După ce ștergem un element din mulțime facem: addAint(v = 1, tl = 1, tr = n, pos = p[i], val = -1).

Complexitatea: O(n*log(n)).

27) Se considera un șir S de paranteze rotunde (închise sau deschise) de lungime n și q query-uri (i, j). Pentru fiecare query, sa se raspunda dacă subsecventa de la i la j (S[i…j]) este corect parantezata.

Solutie:

Initial, trebuie să verificăm dacă numărul de paranteze deschise este egal cu numărul de paranteze închise. Acest lucru se poate face folosind sume parțiale. Definim bal(i) ca „balanța pe care o are parantezarea în primele i elemente”, adică numărul de '(' minus numărul de ')' din primele i elemente. Avem că bal(i) = bal(i - 1) + 1 dacă al i-lea element este '(', sau bal(i) = bal(i - 1) - 1 dacă al i-lea element este ')'. Ca să avem un număr egal de paranteze deschise și închise în parantezarea de la i la j, trebuie ca bal[j] - bal[i - 1] = 0.

Acum trebuie și să nu avem niciodată un k cu proprietatea că k aparține intervalului {i, i+1, …, j - 1, j} cu bal[k] - bal[i - 1] < 0, adică să nu avem niciodată un prefix cu proprietatea că avem mai multe paranteze închise decât deschise pe el. Dacă nu există niciun astfel de k și parantezarea este echilibrată, atunci există soluție. Stim că abs(bal[i] - bal[i - 1]) = 1. Deci, dacă există un k cu proprietatea că bal[k] - bal[i - 1] < 0, atunci există unul cu proprietatea că bal[k] = bal[i - 1] - 1. Dacă pentru fiecare indice z găsim următorul indice u[z] (u[z] > z și bal[u[z]] = bal[z] - 1). u[z] reprezinta urmatoarea pozitie la dreapta cu balansul mai mic decat pozitia curenta. Atunci trebuie doar să verificăm că u[z] > j pentru a nu avea un balans negativ pe parcursul secvenței de la i la j. Acest indice z se poate să fie găsit fie cu o stivă (de la capăt la început), fie cu un vector de frecvență care menține ultimul indice de o anumită valoare, abuzând de faptul că bal are valori <= n în modul.

Complexitatea fiind in ambele cazuri: O(n)

Solutie 2:

Alternativ, se poate folosi si rmq. Ideea de balans se pastreaza (bal[j] - bal[i - 1] trebuie sa fie egal cu 0) ceea ce se verifica folosind sume partiale, dar in loc de stiva folosim un rmq pentru pentru a afla balansul minim pe fiecare interval si se verifica daca acesta e >= bal[i-1].

Complexitatea: n*log(n)

28) Se da un vector A cu N elemente. 1 <= A[i] <= N. pentru fiecare i sa se gaseasca j minim a.i. A[j] < A[i].

Soluție:

Vom profita de faptul că 1 <= A[i] <= N și vom utiliza un vector de frecvență în care, pe poziția i, vom pune indicele minim al unui element care are valoarea egală cu i. După aceea, vom face un nou vector sau vom reconstrui vectorul nostru în așa fel încât acesta să mențină indicele minim al unei poziții care are valoarea <= i, acest lucru realizându-se foarte ușor, doar înlocuind vectorul cu minimul dat pe prefix. Dacă numim acest vector cu T[i], atunci în momentul în care trebuie să răspundem cu soluția pentru indicele i, va trebui să afișăm T[a[i]-1].

Complexitatea: O(n).

Soluție 2:

Alternativ, putem folosi un arbore de intervale. Pentru poziția i, cel mai mic j cu proprietatea A[j]<A[i] este query(început=1,final= i). Apoi adăugăm în arbore pe poziția A[i] elementul i, dacă pe poziția A[i] nu se află deja ceva (add(poz=A[i] val= i)).

Complexitatea: n*log(n)

29) Se da un vector A cu N elemente. 1 <= A[i] <= N. Să se numere tripletele (i, j, k) a.i. 1 <= i < j < k <= N si A[i] < A[j] < A[k].

Soluție:

Observăm următorul fapt de care putem să abuzăm: odată ce fixăm J, adică indicele din mijloc, calculele pe care le facem la stânga sunt complet independente de calculele pe care le facem la dreapta, adică trebuie doar să numărăm câți indici i există la stânga cu proprietatea că a(i) < a(j) și trebuie să numărăm câți indici k există cu proprietatea că a(k) > a(j). Odată ce calculăm aceste două numere și le numim x(j) și y(j), trebuie să adăugăm la soluția noastră x(j) * y(j), deci soluția finală va fi sum(j=1…n | x(j) * y(j)).

Voi descrie cum se poate calcula x(j), y(j) calculându-se aproape la fel, mai puțin că vom parcurge vectorul în ordine inversă și că în loc să numărăm câte sunt mai mici, trebuie să numărăm câte sunt mai mari. Voi calcula vectorul X cu un arbore de intervale: parcurgem vectorul de la stânga la dreapta și în momentul în care trecem prin dreptul indicelui i, va trebui să zicem că x(i) = query(1, a(i)-1), iar după aceea facem add(a(i), +1), profitând de faptul că elementele sunt între 1 și n.

Vectorul Y se calculează asemănător, dar cu un alt arbore de intervale. Parcurgem vectorul de la dreapta la stânga și în momentul în care trecem prin dreptul indicelui i, y(i) = query(a(i)-1,n), iar după aceea facem add(a(i), +1), profitând de faptul că elementele sunt între 1 și n.

Complexitatea: O(n*log(n)).