### **1. Min-Heap**

🟢 **Ce este:** O structură unde cel mai mic număr e mereu în vârf (prima poziție). Gândește-te la o piramidă cu cel mai mic număr în vârf.

🔧 **Ce poți face cu el:**

* Adaugi un număr nou → îl pui la sfârșit și apoi îl „urci” până e la locul potrivit.
* Scoți numărul cel mai mic → iei vârful, apoi muți ultimul număr sus și îl „cobori” unde trebuie.

📌 **Exemplu simplu:**

* Adaugi 4 → [4]
* Adaugi 2 → [2, 4] (2 urcă, că e mai mic)
* Adaugi 7 → [2, 4, 7]

### **2. Arbore Binar de Căutare (BST)**

🟢 **Ce e:** Un fel de arbore (ca o schemă cu noduri), unde fiecare nod are:

* în stânga: numere mai mici
* în dreapta: numere mai mari

🔧 **Ce faci cu el:**

* Poți să adaugi ușor numere.
* Le cauți repede (dai mereu stânga/dreapta ca la un joc „mai mare sau mai mic”).

📌 **Exemplu:**

* Adaugi 10 → rădăcină
* Adaugi 5 → se duce în stânga
* Adaugi 12 → se duce în dreapta

### **3. Tabel Hash + Coliziuni**

🟢 **Ce e:** O tehnică să găsești rapid o valoare (ex: ca un sertar în care pui ceva folosind un cod scurt).

🔧 **Problema:** Uneori două lucruri au același „sertar” → asta e o **coliziune**.

🛠 **Soluții:**

* Pui mai multe în același sertar (listă).
* Cauti următorul sertar liber.

📌 **Exemplu:** Vrei să pui „Ana” și „Bob” în sertare → ambele ajung la sertarul 2 → trebuie să le ții cumva pe amândouă.

### **4. Stivă (Stack)**

🟢 **Ce e:** O grămadă de cărți – pui deasupra, iei de deasupra.  
 📘 Ultimul pus, primul scos – ca o stivă de farfurii.

🔧 **Operații:**

* PUSH: pui ceva deasupra
* POP: scoți ce e deasupra

📌 **Exemplu:**

* PUSH 5 → [5]
* PUSH 3 → [5, 3]
* POP → scoți 3, rămâne [5]

### **5. Cum ștergi dintr-un arbore (BST)**

🔧 3 cazuri:

1. **Frunză** (n-are nimic sub ea): o ștergi direct.
2. **Are un copil:** îl muți în locul ei.
3. **Are doi copii:** cauți următorul număr mai mare și îl pui în loc.

📌 **Imaginează-ți** că tai o ramură. Dacă are alte crengi sub ea, trebuie să le muți.

### **6. Structuri care sunt rapide (O(log n))**

🟢 Unele structuri (ca arborele AVL, Red-Black) sunt **echilibrate** și găsesc lucruri repede.

📌 Gândește-te că arborele e tuns bine, nu e înclinat pe o parte – ca să ajungi repede la orice număr.

### **7. Arbore Ternar**

🟢 **Ce e:** Un arbore unde fiecare nod poate avea **3 copii**, nu doar 2.

📌 Ex: Dacă ai 19 noduri, înălțimea poate fi mică sau mare, depinde cum e făcut. Înălțimea înseamnă câți pași faci de la rădăcină la ultima frunză.

### **8. Alegerea algoritmului de sortare**

🔧 Dacă ai **puține valori distincte** dar multe în total → folosești Counting Sort sau Radix Sort (foarte rapide).

📌 Ex: ai 100 milioane de numere, dar toate sunt între 0 și 1000 → folosești Counting Sort.

### **9. Secvență Invalidă în BST**

🟢 Când cauți un număr în arbore, trebuie să mergi **numai în direcția corectă**:

* Dacă cauți 3 și ești la 10 → mergi în stânga.
* Dacă ești la 2 → mergi în dreapta.

❌ Dacă încerci să mergi aiurea (ex: înapoi la un nod mai mare), e greșit → secvență invalidă.

### **10. Cum construiești un heap dintr-un vector**

🟢 Ai un vector cu numere. Le „aranjezi” ca un heap (cu reguli de heap – ex: min-heap are cel mai mic sus).

🔧 Python are funcția heapify care face asta direct:

* import heapq
* nums = [5, 2, 8]
* heapq.heapify(nums)
* print(nums) # [2, 5, 8] ca min-heap

### **11. Heap Binar – Caracteristici**

🟢 **Ce e:** Un fel de arbore unde numerele sunt puse astfel încât:

* fiecare părinte e mai mic (min-heap) sau mai mare (max-heap) decât copiii lui.
* arborele e complet – toate rândurile sunt pline, în afară de ultimul.

📌 **Gândește-te** la un joc cu reguli stricte: mai micul sus, mai marele jos.

### **12. Unde poate fi minimul într-un Max-Heap**

📘 Într-un Max-Heap, cel mai mare e în vârf. Deci:

* minimul sigur NU e sus,
* poate fi doar într-o frunză (jos de tot),
* și acolo pot fi mai multe frunze, deci mai multe poziții posibile.

### **13. Cât de rapid poate fi cel mai bun algoritm de sortare cu comparații**

🟢 Dacă folosești doar „mai mic / mai mare” între numere, **cel mai rapid posibil** este:

* **n log n** în cel mai rău caz.

📌 Nu poți avea unul „magie” care le sortează mereu în O(n), doar dacă nu compari (ex: Counting Sort).

### **14. Diferența de înălțime între frunze în Max-Heap**

🟢 Înălțimea = câți pași ai de la rădăcină la frunză.

Într-un Max-Heap:

* toate frunzele sunt jos, dar nu toate la același nivel,
* diferența maximă e de **1** dacă e „complet”, și **2** dacă nu e perfect plin.

### **15. Stivă vs Cozi**

📦 **Stivă (Stack):**

* pui deasupra, iei deasupra → LIFO (Last In, First Out)

🚪 **Coada (Queue):**

* pui la coadă, scoți de la început → FIFO (First In, First Out)

🧠 Folosiri:

* Stiva → pentru lucruri ca paranteze sau backtracking.
* Coada → pentru ordine de procesare.

### **16. Cât durează ștergerea într-un BST (caz rău)**

🛠 Dacă arborele e „înclinat” (ca o listă):

* durează **n pași** să găsești și ștergi un nod.
* Asta e **cel mai rău caz**: O(n)

📌 În mod normal e O(log n), dar nu întotdeauna.

### **17. Structuri ca Binomial Heap, B-Arbori**

📚 **Binomial Heap:** bun la „îmbinări” rapide între două heap-uri.  
 📘 **B-Arbori:** folosiți în baze de date (lucrează bine cu fișiere mari).

🧠 Sunt mai avansate, dar ideea e că pot face unele lucruri **mai repede sau eficient** decât altele.

### **18. Câte noduri are un arbore ternar dacă are 15 frunze**

🧮 Formulă: număr total de noduri = 2 × frunze – 1 → 2×15 – 1 = 29

📌 **Important:** în arborele ternar perfect, fiecare nod intern are 3 copii.

### **19. Ce algoritm de sortare folosești la numere mari**

🟢 Ai milioane de numere între 0 și 2³² (foarte mari valori):

* Dacă nu vrei să compari → **Radix Sort (bază mare)** merge excelent.
* Dacă sunt răspândite → Merge Sort sau Quick Sort.

### **20. Secvență invalidă de căutare în BST**

📘 Când cauți un număr:

* Dacă e mai mic → mergi stânga.
* Dacă e mai mare → mergi dreapta.

❌ Dacă secvența merge la dreapta, apoi la stânga, apoi la dreapta din nou → nu e posibil → **e invalidă**.

Perfect, hai să continuăm cu următoarele **10 concepte (21–30)** explicate clar, ca să le înțeleagă oricine:

### **21. Sortarea valorilor dintr-un BST cu „inordine”**

🟢 **Inordine** = vizitezi mai întâi tot ce e în stânga, apoi nodul curent, apoi tot ce e în dreapta.

📌 Dacă faci asta într-un BST, o să obții un **șir sortat**.

🔧 **Exemplu:**

5

/ \

3 7

Inordine → 3, 5, 7

### **22. Reguli importante pentru BST**

📘 Fiecare nod are:

* maxim 2 copii,
* stânga = valori mai mici,
* dreapta = valori mai mari,
* doar o rădăcină (nod de sus).

✅ Dacă se respectă asta, ai un arbore binar de căutare valid.

### **23. Unde poate fi al doilea cel mai mic într-un Max-Heap**

📘 Max-Heap = cel mai mare număr e sus.

* Al doilea cel mai mic **trebuie să fie tot jos**, într-o frunză.
* Poate fi în **mai multe locuri**, nu există o poziție sigură pentru el.

### **24. Algoritm de sortare: cazul cel mai bun vs cel mai rău**

📘 Ex: un algoritm sortează foarte repede dacă datele sunt aproape sortate (ex: QuickSort).

* Dar dacă ai ghinion și datele sunt într-o ordine proastă, poate fi **lent**.

✅ Important: un algoritm bun trebuie să se descurce bine în cazuri proaste, nu doar bune.

### **25. Diferența între frunze într-un arbore binar complet**

🌲 Arbore binar complet = toate rândurile sunt pline, ultimul rând e completat de la stânga.

📏 Frunzele pot fi la aceeași înălțime sau la cel mult o diferență de **1**.

### **26. Deque (Double-Ended Queue)**

🟢 Deque = coadă unde poți să adaugi sau scoți **și din față și din spate**.

🔧 E ca un autobuz:

* Poți să urci și să cobori atât prin față cât și prin spate.

✅ Super flexibil: poți să-l folosești ca stivă sau ca coadă.

### **27. Treap (BST + Heap)**

🌳 Un „Treap” e o combinație între:

* **arbore binar de căutare** (după cheie)
* **heap** (după prioritate aleatoare)

📌 Ajută să păstrezi echilibrul în arbore fără reguli complicate.

### **28. Sparse Table – pentru întrebări de tip „minim pe interval”**

📘 Ai un șir de numere și vrei să afli rapid:

* Care e cel mai mic număr între poziția i și j?

🔧 Sparse Table e o tehnică de pre-calculare.

* Pre-calculăm niște tabele → apoi răspundem **foarte repede** (într-o fracțiune de secundă).

### **29. Trie – pentru cuvinte**

📘 Trie = un arbore special pentru cuvinte.

* Fiecare literă e un pas în arbore.
* Foarte util pentru: căutări rapide, sortare după alfabet, autocomplete.

🔧 Exemplu:  
 Adaugi „car”, „care”, „cart” → toate împart rădăcina „car”.

### **30. Construirea unui Heap**

📘 Ai un șir de numere și vrei să le transformi într-un min-heap sau max-heap.

🔧 Se face printr-un proces numit heapify, care rearanjează numerele.

📌 E ca atunci când faci ordine într-o grămadă: cel mai mic sus (la min-heap).

Super! Hai să terminăm cu ultimele **10 concepte (31–40)**, explicate tot simplu și pe înțelesul oricui:

### **31. Skip List**

📘 O listă specială care are mai multe „niveluri” ca un lift:

* Nivelul de jos = o listă normală.
* Fiecare nivel de sus „sare” peste mai multe elemente ca să ajungă mai repede unde vrei.

🔧 Când inserezi un număr:

* Arunci o monedă: dacă iese „S”, îl pui și mai sus.
* Dacă iese „B”, te oprești.

✅ E o metodă „cu noroc” să cauți repede, dar fără arbore.

### **32. Câte noduri poate avea un heap de înălțime k**

📏 Dacă înălțimea (adică câți pași ai de la rădăcină până jos) e k:

* Număr **minim** de noduri: k + 1 → toate pe o linie (ca un șir).
* Număr **maxim**: 2^(k+1) – 1 → dacă e complet, ca o piramidă.

### **33. Câți copii poate avea succesorul unui nod cu 2 copii (în BST)**

📘 Succesorul = următorul număr mai mare decât nodul ales.

🔧 Dacă un nod are 2 copii, succesorul lui e cel mai mic din dreapta → deci:

* poate avea **cel mult un copil în dreapta**,
* sau poate să fie o frunză.

### **34. Unirea a două heap-uri**

🔧 Dacă vrei să unești două heap-uri:

* Dacă sunt **heap-uri binomiale** (structură specială) → se face repede, în O(log n).
* Dacă sunt **heap-uri normale** → trebuie să le combini și apoi heapify tot → durează O(n).

### **35. Poți găsi minimul într-un Max-Heap repede?**

📘 Max-Heap are cel mai mare număr în vârf → cel mai mic e jos.

❌ Nu poți să-l găsești rapid → trebuie să te uiți **la toate frunzele**, deci durează **mult (O(n))**.

### **36. Cum verifici dacă un arbore e BST în O(n)**

✅ Parcurgi arborele în **inordine** și verifici dacă valorile sunt **în ordine crescătoare**.

🔧 Dacă da → e BST.  
 📌 Se poate face în **O(n)**, adică o singură trecere prin arbore.

### **37. Dacă parcurgi un Heap în inordine, iese ceva sortat?**

❌ NU!

📘 Chiar dacă parcurgi un Heap în ordine stânga → rădăcină → dreapta,

* NU garantează că obții un șir sortat.

🔧 Doar BST are această proprietate.

### **38. Găsirea permutării după „câte mai mici înainte”**

📘 Ai un șir ca: 0 1 1 0 → înseamnă că:

* primul număr are 0 mai mici înaintea lui,
* al doilea are 1 mai mic înaintea lui, și așa mai departe.

🔧 Vrei să afli cum arăta permutarea originală (care numere, în ce ordine).

📌 Se face cu o structură care ține evidența pozițiilor libere – cum ar fi un arbore special sau o listă.

### **39. Verificare paranteze corecte între 2 poziții**

📘 Ai un șir de paranteze, de exemplu: (()())

🔧 Între poziția i și j → vrei să știi dacă parantezele sunt „corect închise”.

* Se poate face cu un **stack** sau cu un contor care adună +1 la '(' și -1 la ')'.

📌 Dacă la final e 0 și nu ai avut valori negative → e bine.

### **40. Pentru fiecare A[i], găsește cel mai apropiat A[j] < A[i]**

📘 Ai un șir de numere și pentru fiecare vrei să știi:

* „Unde e primul număr din stânga mea care e mai mic decât mine?”

🔧 Se rezolvă cu o **stivă specială** care reține ultimele numere în ordine descrescătoare.

📌 Foarte util pentru programare competitivă sau interviuri.

### **41. Numărare de triplete crescătoare (i, j, k)**

📘 Ai un șir de numere și vrei să numeri de câte ori poți alege **trei poziții** așa încât:

i < j < k și A[i] < A[j] < A[k]

🔧 Adică găsești toate grupurile de **trei numere crescătoare** care respectă ordinea din șir.

📌 E o problemă un pic mai grea, dar:

* Se poate face eficient cu două treceri prin șir,
* sau cu un **truc matematic**: la fiecare poziție j, vezi câte numere mai mici ai înainte și câte mai mari ai după → înmulțești.

## **✅ Rezumat Final**

🔵 **Structuri importante**:

* **Stivă** = ultimul intrat, primul ieșit (LIFO)
* **Coada** = primul intrat, primul ieșit (FIFO)
* **Heap** = cel mai mic/mare sus
* **BST** = în stânga valori mai mici, în dreapta mai mari
* **Trie** = pentru cuvinte
* **Skip list** = listă cu „niveluri” pentru căutare rapidă

🔵 **Algoritmi de sortare**:

* **Counting/Radix** → pentru valori mici
* **Merge/Quick** → pentru comparații
* **Timsort** (folosit în Python) → eficient în general

🔵 **Tehnici utile**:

* **Heapify** → transformi un șir într-un heap
* **Inordine pe BST** = obții numere sortate
* **Stack pentru paranteze**
* **Sparse Table** = pentru întrebări de genul „care e minimul între i și j?”

🔵 **Complexități de reținut**:

* Căutare în BST echilibrat: O(log n)
* Heap: insert/delete → O(log n)
* Tabel hash: O(1) în medie
* Sortare cu comparații: cel mai bine → O(n log n)