✅ A1: Hash Table (Linear Probing & Double Hashing)

Q1: What is a hash function?
→ A hash function maps a key to an index in the hash table using a formula like `key % table_size`.

Q2: Linear probing with vs. without replacement?
→ *Without replacement:* Insert next empty slot if collision occurs.
→ *With replacement:* Replace if current slot's hashed key is different and reinsert the old one.

Q3: How does double hashing reduce clustering?
→ It uses two hash functions to find new positions, avoiding clusters from linear probing.

Q4: Time complexity of searching?
→ **Average:** O(1); **Worst:** O(n), when many collisions occur.

Q5: Comparisons in worst case?
→ Linear probing may need up to `n` comparisons. Double hashing usually needs fewer due to better distribution.

✅ A4: Set ADT

Q1: What is a set?
→ A collection of unique elements with no duplicates.

Q2: Difference from a list?
→ Sets don't allow duplicates, and order doesn't matter.

Q3: What is union, intersection, difference?
→ *Union:* All unique elements from both sets.
→ *Intersection:* Only common elements.
→ *Difference:* Elements in one but not in the other.

Q4: What does contains() do?
→ Returns True if element exists in the set.

Q5: Subset check?
→ All elements of B are in A ⇒ B is a subset of A.

✅ A5: General Tree (Book Structure)

Q1: What is a general tree?
→ A tree where each node can have multiple children.

Q2: How is memory allocated?
→ Typically using arrays or pointers to child nodes (e.g., `child[10]`).

Q3: Time complexity to build tree?
→ O(n), where `n` is number of nodes (chapters + sections + subsections).

Q4: Space complexity?
→ Also O(n), storing all node pointers and labels.

Q5: Recursive implementation possible?
→ Yes, tree creation and traversal are naturally recursive.

✅ B7: Expression Tree

Q1: What is an expression tree?
→ A binary tree that represents an arithmetic expression.

Q2: Build tree from prefix?
→ Scan from right to left; use stack for operands, then pop for operator's children.

Q3: Postorder traversal meaning?
→ Left → Right → Root (used to convert prefix to postfix).

Q4: Why use stack?
→ To simulate recursive function call order non-recursively.

Q5: How is tree deleted?
→ Using postorder traversal: delete left, right, then root.

✅ B11: BST Dictionary

Q1: What is a BST?
→ A binary tree where left < root < right.

Q2: Insert vs. search?
→ Insert: place node in sorted order.
→ Search: follow BST rules until node found or NULL.

Q3: Handle duplicates?
→ Usually duplicates are not allowed or are handled via counters or lists.

Q4: Time complexity (best/worst)?
→ Best: O(log n), Worst: O(n) if tree is skewed.

Q5: How is sorting done?
→ In-order traversal gives ascending order.

✅ C13: DFS/BFS on Graph

Q1: DFS vs. BFS?
→ DFS: depth-first (stack or recursion), BFS: level-order (queue).

Q2: Matrix vs. List?
→ Matrix: uses 2D array, good for dense graphs.
→ List: space-efficient for sparse graphs.

Q3: BFS preferred when?
→ When shortest path or level-wise traversal is needed.

Q4: What's used for DFS and BFS?
→ DFS: stack (explicit or implicit via recursion)
→ BFS: queue.

✅ C15: Prim's MST

Q1: What is MST?
→ A tree connecting all vertices with minimum total weight and no cycles.

Q2: How does Prim's work?
→ Start with any node, always add the cheapest edge connecting a new node.

Q3: Why sort edges by weight?
→ To ensure minimum cost at each step.

Q4: What is the data structure used?
→ Vector or priority queue for sorted edge selection.

Q5: Time complexity?
→ Using adjacency matrix: $O(V^2)$; with heap: $O(E \log V)$.

✅ D18: OBST

Q1: What is OBST?
→ A binary search tree that minimizes expected search cost based on probabilities.

Q2: Why use probabilities?
→ To model frequent vs. rare searches; more frequent keys should be nearer root.

Q3: How to choose root?
→ Try all possible roots and pick the one minimizing cost: DP-based.

Q4: Cost function?
→ `C[i][j] = min over k (C[i][k-1] + C[k][j]) + weight[i][j]`

Q5: Time & space complexity?
→ Time: $O(n^3)$, Space: $O(n^2)$

✅ D19: AVL Tree Dictionary

Q1: What is AVL tree?
→ A balanced BST where the height difference (balance factor) of each node is ≤1.

Q2: Rotations in AVL?
→ LL, RR, LR, RL – used to maintain balance after insert/delete.

Q3: AVL vs BST?
→ AVL maintains balance; guarantees O(log n) operations.

Q4: Balance ensured how?
→ After each insertion/deletion, check balance and rotate if needed.

Q5: Height complexity?
→ O(log n), better than unbalanced BST.

✅ E20: Priority Queue (Hospital)

Q1: What is a priority queue?
→ A queue where each element has a priority and highest priority is served first.

Q2: How patients prioritized?
→ Based on an integer (3 = serious, 2 = medium, 1 = general).

Q3: Data structure used?
→ Singly linked list (with sorted insertion based on priority).

Q4: Time complexity?
→ Insertion: O(n), Deletion (front): O(1)

Q5: Can we use arrays?
→ Yes, but linked list is more flexible for dynamic sizes.

✅ F23: Sequential File - Student Info

Q1: What is sequential file?
→ File where records are stored in order of arrival.

Q2: Add/delete/search – how?
→ Read/write line by line; for delete, copy all except deleted one to new file.

Q3: Limitation?
→ Search is slow: O(n); not good for large datasets.

Q4: Display matching records?
→ Compare roll no. while reading file line by line.

Q5: How stored on disk?
→ Usually as plain text or binary; one record per line.

✅ F24: Indexed Sequential File - Employee Info

Q1: What is indexed sequential file?
→ Combines index (for faster access) and sequential file.

Q2: How is it better than sequential?
→ Uses index to jump to approximate location, reducing search time.

Q3: What is an index?
→ A table mapping keys (like employee ID) to positions in file.

Q4: Can we use binary search?
→ Yes, on the index array.

Q5: How is deletion done?
→ Mark record as deleted, or remove from file and rebuild index.