Data Structures and Algorithms Viva Questions and Answers

Prepared by Rahul
For DSA Exam on 4th July

Contents

I	**Unit I: Introduction and Arrays**	**2**
II	**Unit II: Linked Lists**	**4**
III	**Unit III: Stacks and Queues**	**6**
IV	**Unit IV: Trees and Recursion**	**8**
V	**Unit V: AVL Trees and Heaps**	**10**
VI	**Unit VI: Graphs and Hashing**	**12**
VII	**Complexity Table**	**14**

I Unit I: Introduction and Arrays

1. **What is a data structure? Give examples.**
   A data structure is a method of organizing data to enable efficient operations like storage, retrieval, and modification. Examples include arrays (fixed-size collections), linked lists (dynamic node-based structures), stacks (LIFO), queues (FIFO), trees (hierarchical), and graphs (networks of nodes and edges).

2. **Define an algorithm and its characteristics.**
   An algorithm is a finite set of steps to solve a problem. Characteristics: finiteness (terminates after a finite number of steps), definiteness (clear instructions), effectiveness (feasible operations), and inputs/outputs (accepts input and produces output).

3. **What is time complexity, and why is it important?**
   Time complexity measures how an algorithm's runtime scales with input size, expressed in Big O notation. It's crucial for predicting performance and selecting efficient solutions for large datasets.

4. **Explain space complexity with an example.**
   Space complexity measures memory usage relative to input size. Example: Bubble sort uses O(1) extra space (in-place), while merge sort uses O(n) due to temporary arrays.

5. **What are Big O, Omega, and Theta notations?**
   Big O: upper bound (worst case), e.g., $O(n^2)$ for bubble sort. Omega: lower bound (best case), e.g., $\Omega(n)$ for bubble sort. Theta: tight bound (average case when upper and lower bounds match), e.g., $\Theta(n \log n)$ for merge sort.

6. **How does an array differ from other data structures?**
   Arrays store elements in contiguous memory with fixed size, enabling O(1) access but requiring O(n) for insertion/deletion due to shifting, unlike linked lists (dynamic, O(1) insertion).

7. **Describe memory allocation for arrays.**
   Arrays use contiguous memory blocks. For an integer array of size n, if the first element is at address x, the ith element is at x + i * sizeof(int).

8. **What is the significance of contiguous memory in arrays?**
   Contiguous memory allows fast index-based access (O(1)) via pointer arithmetic but limits dynamic resizing compared to linked lists.

9. **How do you access an element in an array?**
   Use its index: arr[i]. Time complexity is O(1) as it calculates the address directly.

10. **Explain array insertion with an example.**
    To insert 5 at index 2 in [1, 2, 3, 4]: shift 3 and 4 right to get [1, 2, 5, 3, 4]. Worst-case time is O(n).

11. **What challenges arise during array insertion?**
    Shifting elements (O(n) time) and potential overflow if the array is full, requiring resizing.

12. **Describe array deletion with an example.**
    To delete 3 from [1, 2, 3, 4]: shift 4 left to get [1, 2, 4]. Worst-case time is O(n).

13. **What is bubble sort, and how does it work?**
    Bubble sort compares adjacent elements and swaps if out of order, bubbling the largest to the end each pass. Example: [5, 3, 1] → [3, 5, 1] → [3, 1, 5] → [1, 3, 5].

14. **Analyze bubble sort's complexity.**
    Best: O(n) (sorted array), Average/Worst: $O(n^2)$ (n passes, up to n-1 comparisons each). Space: O(1).

15. **What is insertion sort, and how does it differ from bubble sort?**
    Insertion sort builds a sorted portion by inserting elements into their correct position. Unlike bubble sort (swaps adjacent), it shifts elements. Example: [5, 3, 1] → [3, 5, 1] → [1, 3, 5].

16. **Analyze insertion sort's complexity.**
    Best: O(n), Average/Worst: $O(n^2)$. Space: O(1). More efficient than bubble sort for small or nearly sorted arrays.

17. **Explain selection sort with an example.**
    Selection sort finds the minimum element and places it at the start each pass. Example: [5, 3, 1] → [1, 3, 5] (swap 5 with 1) → [1, 3, 5].

18. **Analyze selection sort's complexity.**
    Best/Average/Worst: $O(n^2)$ (n passes, n-i comparisons). Space: O(1). Not adaptive like insertion sort.

19. **What is linear search? Provide an example.**
    Linear search checks each element sequentially. Example: In [1, 3, 5], search 3: check 1 (no), 3 (yes). Time: O(n).

20. **What is binary search, and when can it be used?**
    Binary search halves the search space in a sorted array. Example: [1, 3, 5], search 3: mid=3 (found). Time: O(log n). Requires sorted data.

21. **Compare linear and binary search complexities.**
    Linear: Best O(1), Average/Worst O(n). Binary: Best O(1), Average/Worst O(log n). Binary is faster for large sorted arrays.

22. **What is stability in sorting, and why does it matter?**
    Stability preserves the relative order of equal elements. Example: In [(3, A), (3, B)], stable sort keeps A before B. Matters in multi-key sorting.

23. **Is insertion sort stable? Prove it.**
    Yes. It shifts elements without swapping equals past each other. Example: [(3, A), (3, B)] → [(3, A), (3, B)].

24. **How do you merge two sorted arrays?**
    Use two pointers: [1, 3] and [2, 4] → compare and build [1, 2, 3, 4]. Time: O(n+m).

25. **What is the space complexity of sorting algorithms?**
    In-place (bubble, insertion): O(1). Out-of-place (merge): O(n) due to extra arrays.

26. **Explain the adaptive nature of insertion sort.**
    It performs better (closer to O(n)) on partially sorted arrays by reducing shifts.

27. **What are multi-dimensional arrays?**
    Arrays of arrays, e.g., a 2D array (matrix) like [[1, 2], [3, 4]]. Used for grids or tables.

28. **How do you traverse a 2D array?**
    Use nested loops: for i (rows), for j (columns). Time: O(rows $\times$ cols).

29. **What is the difference between static and dynamic arrays?**
    Static arrays have fixed size (e.g., int arr[5]), while dynamic arrays (e.g., ArrayList) resize as needed.

30. **How can arrays be used to implement other data structures?**
    Stacks (top pointer), queues (front/rear pointers), heaps (parent-child index formulas).

II Unit II: Linked Lists

1. **What is a linked list, and how does it differ from an array?**
   A linked list is a chain of nodes, each with data and a pointer to the next. Unlike arrays (contiguous, fixed-size), it's dynamic and non-contiguous.

2. **List and describe types of linked lists.**
   Singly (one-way), Doubly (two-way), Circular (last links to first), Header (special first node).

3. **Explain the structure of a singly linked list node.**
   Contains data (e.g., int) and a next pointer (to the next node or null).

4. **How does a doubly linked list improve on a singly linked list?**
   Adds a previous pointer, enabling bidirectional traversal and easier deletion (O(1) with direct access).

5. **What is a circular linked list, and where is it used?**
   Last node links to the first. Used in round-robin scheduling or cyclic buffers.

6. **How is memory managed in linked lists?**
   Dynamically allocated per node (e.g., malloc in C), allowing flexibility but increasing overhead.

7. **What are the advantages of linked lists over arrays?**
   Dynamic resizing, O(1) insertion/deletion at known positions, no shifting required.

8. **What are the disadvantages of linked lists?**
   O(n) access time, extra memory for pointers, no cache locality.

9. **How do you traverse a singly linked list?**
   Start at head, follow next pointers until null. Time: O(n).

10. **Describe insertion at the beginning of a singly linked list.**
New node's next = head, head = new node. Time: O(1).

11. **How do you insert at the end of a singly linked list?**
Traverse to the last node (next = null), set its next to the new node. Time: O(n).

12. **How can insertion at the end be optimized?**
Maintain a tail pointer, making it O(1).

13. **Explain deletion from the beginning of a singly linked list.**
Head = head.next, free old head. Time: O(1).

14. **How do you delete a specific node in a singly linked list?**
Traverse to the previous node, set its next to the target's next. Time: O(n).

15. **Describe insertion in a doubly linked list.**
Adjust next and prev pointers of new node and its neighbors. Time: O(1) with direct access.

16. **How do you delete from a doubly linked list?**
Update prev.next and next.prev to skip the node. Time: O(1) with direct access.

17. **What is the time complexity of searching in a linked list?**
O(n), as it requires linear traversal.

18. **How do you reverse a singly linked list iteratively?**
Use three pointers (prev, curr, next): while curr, next = curr.next, curr.next = prev, prev = curr, curr = next. Time: O(n).

19. **How do you reverse a linked list recursively?**
Recursively reach the end, then reverse pointers on the way back. Time: O(n), Space: O(n).

20. **What is Floyd's cycle detection algorithm?**
Uses two pointers (slow moves 1, fast moves 2). If they meet, there's a cycle. Time: O(n).

21. **How do you find the middle of a linked list?**
Use two pointers: slow (1 step), fast (2 steps). When fast reaches the end, slow is at the middle. Time: O(n).

22. **What is a skip list, and how does it work?**
A layered linked list where higher layers skip nodes, enabling O(log n) search via probabilistic balancing.

23. **How do you merge two sorted linked lists?**
Compare nodes and link smaller values iteratively. Time: O(n+m).

24. **What is the space complexity of a linked list?**
O(n) for n nodes, including data and pointers.

25. **How do you implement a stack with a linked list?**
Push: insert at head. Pop: remove from head. Both O(1).

26. **How do you implement a queue with a linked list?**
    Enqueue: add to tail. Dequeue: remove from head. O(1) with head and tail pointers.

27. **What happens if you don't free deleted nodes?**
    Memory leaks occur, wasting resources.

28. **How do you detect and remove a loop in a linked list?**
    Use Floyd's algorithm to detect, then move one pointer to head and advance both at the same pace to find the loop start, then break it.

29. **What is the difference between singly and doubly linked list traversal?**
    Singly: forward only (O(n)). Doubly: forward and backward (O(n) but more flexible).

30. **How do you split a linked list into two parts?**
    Find the middle (e.g., slow/fast pointers), set the next of the middle to null, and return two heads.

III Unit III: Stacks and Queues

1. **What is a stack, and what is its principle?**
   A stack is a LIFO (Last In, First Out) structure where elements are added and removed from the top.

2. **List the primary operations of a stack.**
   Push (add to top), Pop (remove from top), Peek (view top), IsEmpty (check emptiness).

3. **How can a stack be implemented using an array?**
   Use an array with a top index. Push: arr[++top] = x. Pop: return arr[top–]. Time: O(1).

4. **How can a stack be implemented using a linked list?**
   Push: insert at head. Pop: remove head. Time: O(1).

5. **What is stack overflow and underflow?**
   Overflow: pushing to a full stack (array-based). Underflow: popping an empty stack.

6. **What is a queue, and what is its principle?**
   A queue is a FIFO (First In, First Out) structure where elements enter at the rear and exit from the front.

7. **List the primary operations of a queue.**
   Enqueue (add to rear), Dequeue (remove from front), Front (view front), IsEmpty (check emptiness).

8. **How can a queue be implemented using an array?**
   Use front and rear pointers. Enqueue: arr[rear++] = x. Dequeue: return arr[front++]. Time: O(1).

9. **What is a circular queue, and why is it useful?**
   Rear wraps to the start when it reaches the end, avoiding wasted space in array-based queues.

10. **How do you implement a circular queue?**
    Use modulo: rear = (rear + 1) % size. Time: O(1).

11. **What is a priority queue, and how does it differ from a regular queue?**
    Elements are dequeued by priority, not order of arrival. Implemented with heaps (O(log n) operations).

12. **What is a deque, and how does it work?**
    Double-ended queue allows insertion/deletion at both ends. Example: addFront, addRear, removeFront, removeRear.

13. **How do you implement a deque using a doubly linked list?**
    Use head and tail pointers, adjusting prev/next for operations at either end. Time: O(1).

14. **What are the applications of stacks?**
    Expression evaluation, backtracking (e.g., maze solving), function call management, undo operations.

15. **What are the applications of queues?**
    Job scheduling (e.g., printer queues), BFS, buffering (e.g., streaming).

16. **How do you reverse a stack?**
    Use recursion or an auxiliary stack: pop all to aux, then push back. Time: O(n).

17. **How do you check for balanced parentheses using a stack?**
    Push opening brackets, pop on closing brackets, ensuring matches. Time: O(n).

18. **What is postfix notation, and how is it evaluated?**
    Operators follow operands (e.g., 3 4 +). Use a stack: push operands, pop two and apply operator on encountering one.

19. **How do you convert infix to postfix?**
    Use a stack for operators, output operands, pop based on precedence. Example: A + B * C → A B C * +.

20. **What is the time complexity of stack and queue operations?**
    Push/Pop (stack), Enqueue/Dequeue (queue): O(1). Space: O(n).

21. **How do you implement a queue using two stacks?**
    Stack1 for enqueue, Stack2 for dequeue. Transfer from 1 to 2 when 2 is empty. Amortized O(1).

22. **How do you implement a stack using two queues?**
    Queue1 holds data, Queue2 assists. Push: enqueue to Q2, move Q1 to Q2, swap. Time: O(n).

23. **What is the role of a stack in recursion?**
    Manages function calls via the call stack, storing return addresses and local variables.

24. **How do you simulate recursion iteratively with a stack?**
    Push states (e.g., parameters) onto a stack and process them in a loop.

25. **What is the space complexity of a circular queue?**
    O(n), where n is the fixed array size.

26. **How do you handle queue overflow in an array implementation?**
    Check if rear == size-1 (linear) or front == (rear + 1) % size (circular).

27. **What is the difference between a stack and a deque?**
    Stack: single-end operations (top). Deque: both ends (front/rear).

28. **How do you implement a priority queue using an array?**
    Store (element, priority) pairs, sort by priority on dequeue. Time: O(n).

29. **What is the advantage of a linked list over an array for stacks/queues?**
    No fixed size, avoiding overflow; dynamic growth.

30. **How do you find the minimum element in a stack in O(1) time?**
    Use an auxiliary stack tracking minimums. Push/pop both stacks, updating min as needed.

IV Unit IV: Trees and Recursion

1. **What is a tree, and what are its key components?**
   A hierarchical structure with a root node, children, and no cycles. Components: root, nodes, edges, leaves.

2. **What is a binary tree, and what are its properties?**
   Each node has at most two children (left, right). Properties: max $2^h nodesatheighth, heightlog(n+1) - 1ton - 1$.

3. **What is a binary search tree (BST)?**
   Left child < parent < right child, enabling efficient search/insertion.

4. **How do you insert a node into a BST?**
   Recursively traverse based on value comparison, insert at null. Time: O(h), h = height.

5. **How do you delete a node from a BST?**
   Cases: no child (remove), one child (link parent to child), two children (replace with in-order successor).

6. **What are the time complexities of BST operations?**
   Search/Insert/Delete: O(log n) average (balanced), O(n) worst (skewed).

7. **What is in-order traversal, and what is its use in BST?**
   Left-Root-Right. In BST, it yields sorted order. Time: O(n).

8. **Explain pre-order and post-order traversals.**
   Pre: Root-Left-Right (tree copying). Post: Left-Right-Root (deletion). Time: O(n).

9. **What is the height of a binary tree, and how do you calculate it?**
   Longest root-to-leaf path. Recursively: height = 1 + max(left height, right height).

10. **What is a complete binary tree?**
    All levels filled except possibly the last, filled left to right.

11. **What is recursion, and how does it apply to trees?**
    A function calls itself. In trees, it processes subtrees (e.g., traversals).

12. **What is the base case and recursive case in recursion?**
    Base: stops recursion (e.g., null node). Recursive: breaks problem into subproblems.

13. **Explain the Tower of Hanoi problem.**
    Move n disks from source to target via auxiliary peg, never placing larger on smaller.

14. **Provide the recursive solution for Tower of Hanoi.**
    Move n-1 to aux, move nth to target, move n-1 to target. Time: O(2^n).

15. **What is merge sort, and how does it use recursion?**
    Divides array into halves, recursively sorts, then merges. Time: O(n log n).

16. **Describe quick sort's recursive mechanism.**
    Picks pivot, partitions array, recursively sorts subarrays. Time: O(n log n) average, O(n^2) worst.

17. **How do you implement binary search recursively?**
    Check mid, recurse on left (if target $<$ mid) or right (if target $>$ mid). Time: O(log n).

18. **What is the space complexity of recursive tree traversals?**
    O(h) for call stack, where h is height (O(log n) balanced, O(n) skewed).

19. **What is tail recursion, and how can it be optimized?**
    Recursive call is the last operation. Compilers can convert it to a loop, reducing space to O(1).

20. **How do you find the lowest common ancestor (LCA) in a BST?**
    Traverse from root: if both nodes $>$ root, go right; if both $<$ root, go left; else, root is LCA.

21. **What is a full binary tree?**
    Every node has 0 or 2 children.

22. **How do you check if a binary tree is balanced?**
    Recursively compute height of left and right subtrees, ensure |height difference| 1. Time: O(n).

23. **What is a perfect binary tree?**
    All internal nodes have 2 children, and all leaves are at the same level.

24. **How do you convert a binary tree to its mirror?**
    Recursively swap left and right children of each node. Time: O(n).

25. **What is the difference between recursion and iteration?**
    Recursion uses call stack (intuitive but space-intensive), iteration uses loops (space-efficient).

26. **How do you find the diameter of a binary tree?**
    Max path between any two nodes. Recursively: max(left height + right height + 1, subtrees' diameters).

27. **What is a threaded binary tree?**
    Uses null pointers to link to in-order predecessor/successor, speeding traversal.

28. **How do you construct a BST from a preorder traversal?**
    First element is root, partition rest into left (< root) and right (> root), recurse.

29. **What is the significance of tree height in complexity?**
    Determines operation times: O(log n) for balanced, O(n) for skewed.

30. **How do you level-order traverse a tree?**
    Use a queue: enqueue root, dequeue and enqueue children until empty. Time: O(n).

V Unit V: AVL Trees and Heaps

1. **What is a balanced BST, and why is it needed?**
   Height difference between subtrees is bounded (e.g., 1 in AVL), ensuring O(log n) operations vs. O(n) in skewed BSTs.

2. **What is an AVL tree, and what is its balance factor?**
   A self-balancing BST where balance factor (left height - right height) is -1, 0, or 1 for each node.

3. **How does an AVL tree maintain balance?**
   Uses rotations (left, right, left-right, right-left) after insertions/deletions.

4. **Describe a left rotation in an AVL tree.**
   Right child becomes root, original root becomes left child of new root. Used for right-heavy imbalance.

5. **When is a right-left rotation needed?**
   When inserting in the left subtree of a right child (e.g., zigzag pattern), requiring a right rotation then left.

6. **How do you insert into an AVL tree?**
   Insert as in BST, update heights, check balance, rotate if needed. Time: O(log n).

7. **How do you delete from an AVL tree?**
   Delete as in BST, update heights, rebalance via rotations. Time: O(log n).

8. **What is the time complexity of AVL tree operations?**
   Search/Insert/Delete: O(log n), guaranteed by balance.

9. **What is a heap, and what are its types?**
   A complete binary tree with heap property: max-heap (parent   children), min-heap (parent   children).

10. **How is a binary heap represented in an array?**
    Root at 0, left child at 2i+1, right at 2i+2. Parent of i at (i-1)/2.

11. **What is the heapify process?**
    Adjusts a subtree to maintain heap property by sifting down the root. Time: O(log n).

12. **How do you insert into a heap?**
    Add at the end, bubble up by swapping with parent if needed. Time: O(log n).

13. **How do you delete the root from a heap?**
    Replace with last element, heapify down. Time: O(log n).

14. **What is heap sort, and how does it work?**
    Build max-heap, repeatedly extract max (root) and heapify. Time: O(n log n).

15. **What is a B-tree, and where is it used?**
    A multi-way balanced tree for disk-based storage (e.g., databases). Each node has multiple keys and children.

16. **How does a B-tree differ from a B+ tree?**
    B-tree stores data in all nodes, B+ tree only in leaves, with internal nodes for navigation.

17. **What is the order of a B-tree?**
    Maximum number of children per node, controlling tree height.

18. **How do you insert into a B-tree?**
    Insert into leaf, split if full, propagate splits upward. Time: O(log n).

19. **What is the time complexity of B-tree operations?**
    Search/Insert/Delete: O(log n), due to balanced multi-way structure.

20. **What is a binomial heap?**
    A collection of binomial trees (each obeying heap property), optimized for merging.

21. **What is a Fibonacci heap, and its advantages?**
    A heap with lazy merging and cutting, offering O(1) amortized insert and O(log n) delete-min.

22. **How do you merge two heaps?**
    For binary heaps: concatenate arrays, rebuild heap (O(n)). Binomial heaps: merge trees (O(log n)).

23. **What is the space complexity of a heap?**
    O(n) for n elements, stored contiguously in array-based heaps.

24. **How do you convert a min-heap to a max-heap?**
    Negate all values or redefine comparison (parent > child). Time: O(n).

25. **What is the height of a heap with n elements?**
 log (n) , as it's a complete binary tree.

26. **How do you implement a priority queue with a heap?**
Use a max/min-heap: insert (bubble up), extract-max/min (heapify down). Time: O(log n).

27. **What is the difference between a heap and an AVL tree?**
Heap: priority-based, no order between siblings. AVL: search-based, strict ordering.

28. **How do you find the kth largest element using a heap?**
Build min-heap of size k, compare remaining elements, replace root if larger. Time: O(n + k log n).

29. **What is the significance of completeness in heaps?**
Ensures O(log n) height, optimizing insert/delete operations.

30. **How do you check if an array represents a valid heap?**
Verify heap property: for all i, arr[i]   arr[2i+1] and arr[2i+2] (max-heap).

VI Unit VI: Graphs and Hashing

1. **What is a graph, and what are its components?**
A set of vertices (nodes) and edges (connections). Directed (arrows) or undirected (lines).

2. **What is the difference between directed and undirected graphs?**
Directed: edges have direction (e.g., A $\rightarrow$ B). Undirected: bidirectional (A   B).

3. **What is a weighted graph?**
Edges have weights (e.g., distances). Used in shortest path problems.

4. **How do you represent a graph in memory?**
Adjacency Matrix (2D array, O(V^2) space) or Adjacency List (lists of neighbors, O(V+E)).

5. **What are the pros and cons of an adjacency matrix?**
Pros: O(1) edge lookup. Cons: O(V^2) space, inefficient for sparse graphs.

6. **What are the pros and cons of an adjacency list?**
Pros: O(V+E) space, efficient for sparse graphs. Cons: O(V) edge lookup.

7. **What is Breadth-First Search (BFS)?**
Explores level by level using a queue. Time: O(V+E).

8. **What is Depth-First Search (DFS)?**
Explores as far as possible down each branch using a stack/recursion. Time: O(V+E).

9. **How do BFS and DFS differ in application?**
BFS: shortest path in unweighted graphs. DFS: topological sorting, cycle detection.

10. **What is the Floyd-Warshall algorithm?**
    Finds all-pairs shortest paths in a weighted graph. Time: $O(V^3)$.
11. **How does Floyd-Warshall handle negative weights?**
    Works unless there's a negative cycle (detectable if diagonal < 0 after computation).
12. **What is Warshall's algorithm?**
    Computes transitive closure (reachability) in a directed graph. Time: $O(V^3)$.
13. **What is hashing, and why is it used?**
    Maps keys to indices via a hash function for O(1) average-time access in hash tables.
14. **What makes a good hash function?**
    Uniform distribution, minimal collisions, fast computation.
15. **What is a collision in hashing, and how does it occur?**
    Two keys map to the same index (e.g., hash(”cat”) = hash(”dog”)).
16. **Describe separate chaining for collision resolution.**
    Each slot holds a linked list of collided keys. Time: O(1 +), = load factor.
17. **Explain open addressing with linear probing.**
    Collided keys probe next slots (index + 1) until free. Time: O(1/(1-)) average.
18. **What is quadratic probing?**
    Probes at quadratic intervals (index + i^2). Reduces clustering vs. linear.
19. **What is double hashing?**
    Uses a second hash function for probe steps, minimizing clustering. Time: O(1/(1-)).
20. **What is the load factor in hashing?**
    Ratio of elements to table size (n/m). Higher increases collisions.
21. **How do you handle deletions in open addressing?**
    Mark slots as “deleted” (tombstones) to maintain probe chains.
22. **What is rehashing, and when is it needed?**
    Resizes table and reinserts elements when load factor exceeds a threshold (e.g., 0.7).
23. **What is the time complexity of hash table operations?**
    Insert/Search/Delete: O(1) average, O(n) worst (many collisions).
24. **How do you detect a cycle in a graph?**
    DFS: mark visited nodes, check for back edges. Time: O(V+E).
25. **What is a spanning tree?**
    A subgraph connecting all vertices with no cycles, having V-1 edges.
26. **How do you find the minimum spanning tree (MST)?**
    Kruskal's (sort edges, add greedily, O(E log E)) or Prim's (grow from vertex, O(E log V)).

27. **What is topological sorting?**
    Orders vertices in a DAG such that for every edge u→v, u precedes v. Uses DFS.

28. **How do you implement BFS using a queue?**
    Enqueue start, mark visited, dequeue and enqueue unvisited neighbors until empty.

29. **What is the significance of graph traversal?**
    Explores connectivity, finds paths, detects cycles, etc.

30. **How does hashing support fast data retrieval?**
    Direct index mapping bypasses linear search, achieving O(1) average time.

VII Complexity Table

Data Structure/Algorithm	Operation	Best Case	Average Case	Worst Case	Space
Array	Access	O(1)	O(1)	O(1)	
	Insertion	O(1)	O(n)	O(n)	
	Deletion	O(1)	O(n)	O(n)	
Singly Linked List	Access	O(1)	O(n)	O(n)	
	Insertion	O(1)	O(1)	O(1)	
	Deletion	O(1)	O(n)	O(n)	
Doubly Linked List	Access	O(1)	O(n)	O(n)	
	Insertion	O(1)	O(1)	O(1)	
	Deletion	O(1)	O(1)	O(1)	
Stack	Push	O(1)	O(1)	O(1)	
	Pop	O(1)	O(1)	O(1)	
Queue	Enqueue	O(1)	O(1)	O(1)	
	Dequeue	O(1)	O(1)	O(1)	
Binary Search Tree	Search	O(log n)	O(log n)	O(n)	
	Insertion	O(log n)	O(log n)	O(n)	
	Deletion	O(log n)	O(log n)	O(n)	
AVL Tree	Search	O(log n)	O(log n)	O(log n)	
	Insertion	O(log n)	O(log n)	O(log n)	
	Deletion	O(log n)	O(log n)	O(log n)	
Heap (Binary)	Search	O(n)	O(n)	O(n)	
	Insertion	O(1)	O(log n)	O(log n)	
	Deletion	O(log n)	O(log n)	O(log n)	
Hash Table	Search	O(1)	O(1)	O(n)	
	Insertion	O(1)	O(1)	O(n)	
	Deletion	O(1)	O(1)	O(n)	
Bubble Sort	Sorting	O(n)	O(n^2)	O(n^2)	
Insertion Sort	Sorting	O(n)	O(n^2)	O(n^2)	
Selection Sort	Sorting	O(n^2)	O(n^2)	O(n^2)	
Merge Sort	Sorting	O(n log n)	O(n log n)	O(n log n)	
Quick Sort	Sorting	O(n log n)	O(n log n)	O(n^2)	O
Linear Search	Search	O(1)	O(n)	O(n)	
Binary Search	Search	O(1)	O(log n)	O(log n)	

Data Structure/Algorithm	**Operation**	**Best Case**	**Average Case**	**Worst Case**	**Space**
BFS (Graph)	Traversal	O(V+E)	O(V+E)	O(V+E)	
DFS (Graph)	Traversal	O(V+E)	O(V+E)	O(V+E)	