**VIVA**

**1) What is a Data Structure?**

Data structures are ways of organizing, storing and managing data in a computer so that it can be accessed and used efficiently. They provide a means to manage and process data effectively for various applications.

Example: A Dictionary like a **phone book**, where you can quickly look up someone's phone number using their name.

**Types of Data Structures:**

1. **Linear Data Structures**: These structures organize data in a linear (sequential) order. E.g.: arrays linked lists

2. **Non-linear Data Structures**: These structures store data in a hierarchical or interconnected way. E.g. : Trees and graphs

**1. Sorting**

Sorting is the process through which data is arranged according to its value in a specific order typically in **ascending** (smallest to largest) or **descending** (largest to smallest) order.

Sorting makes it easier to search, analyze, or work with data efficiently.

**1. Bubble Sort**

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. This process continues until the list is sorted.

So for n elements we repeat the process (n-1) times.

**Steps:**

* Start from the beginning of the list.
* Compare adjacent elements.
* Swap them if the first element is larger than the second.
* Move to the next pair and repeat until the entire list is traversed.
* After each pass, the largest element bubbles up to its correct position.

**Average Case Complexity: O(n²)**

* In the average case, the algorithm must perform multiple comparisons and swaps for every pair of elements, resulting in a time complexity of O(n²).

**2. Selection Sort**

In Selection Sort the array is divided into two parts the sorted and the unsorted region.

In each pass, the smallest element is selected from the unsorted region and exchanged with the element at the beginning of unsorted region.

This process continues until the entire list is sorted.

**Steps:**

* Start with the first element.
* Find the smallest element in the unsorted part of the list.
* Swap it with the first unsorted element.
* Move the boundary between sorted and unsorted parts one step forward.
* Repeat until all elements are sorted.

**Average Case Complexity: O(n²)**

* Selection Sort always scans the entire unsorted part of the list to find the smallest element, resulting in O(n²) time complexity in the average case.

**3. Insertion Sort**

**Insertion Sort** is a simple sorting algorithm that works by building the sorted list one element at a time. At each step, it takes one element from the unsorted part and **inserts** it into the correct position in the sorted part.

**Steps:**

* Start with the second element (the first is considered already sorted).
* Compare it with the element before it.
* Shift all elements larger than the current element to the right.
* Insert the current element into its correct position.
* Repeat for all elements.

**Average Case Complexity: O(n²)**

* The algorithm takes quadratic time in the average case because, on average, each element needs to be compared with half of the elements before it and then shifted.

**4. Shell Sort**

Shell Sort is an optimized version of Insertion Sort. It works by sorting elements that are far apart and gradually reducing the gap between the elements being compared. Hence it is also called Diminishing Insertion sort. The idea is to move elements into place more efficiently than just comparing adjacent elements, which helps improve performance over Insertion Sort.

**Steps:**

* Start by selecting a gap (typically half the list length).
* Perform a regular Insertion Sort on elements that are spaced by this gap.
* Gradually reduce the gap (e.g., divide by 2).
* Continue sorting until the gap is reduced to 1, at which point a final Insertion Sort is performed.

**Average Case Complexity: O(n log n)**

* The time complexity depends on the gap sequence used, but it is generally better than O(n²) and can approach O(n log n) for certain gap sequences.

**Q3:** What is the difference between Selection Sort and Bubble Sort?

* **A3:** In Selection Sort, the minimum (or maximum) element is selected and swapped with the first unsorted element, whereas in Bubble Sort, adjacent elements are repeatedly swapped if they are in the wrong order.

**Q4:** Explain the concept of Shell Sort.

* **A4:** Shell Sort is an improvement over Insertion Sort that works by comparing elements that are far apart, reducing the gap progressively until it performs a regular Insertion Sort.

**2. Searching**

**Searching** is the process of finding the location of a specific element a collection of data, such as an array, list, or database.

**1. Linear Search:**

* **How It Works:** Start at the first element and check each element of the list one by one until the target element is found or the list ends.

**2. Binary Search (For Sorted Arrays Only):**

* **How It Works:** Binary Search divides the list into halves to find the target element.
* **Steps:**

1. Check the middle element of the list.

2. If it matches the target, return its position.

3. If the target is smaller than the middle element, search in the left half.

4. If the target is larger, search in the right half.

5. Repeat until the target is found or the list is empty.

**Q1:** What is the time complexity of Linear Search?

* **A1:** The time complexity of Linear Search is O(n), where n is the number of elements in the array.

**Q2:** What is the advantage of Binary Search over Linear Search?

* **A2:** Binary Search has a time complexity of O(log n) and is much faster than Linear Search (O(n)), but it requires the array to be sorted.

**3. Hashing Techniques**

**Hashing** is a technique used to store and retrieve data efficiently using a special function called a **hash function**.

Hash function maps input data (keys) to a unique code or output (called a hash value or hash code/ address) that represents where the data is stored in the hash table.

**Collision**

A collision is the event that occurs when a hashing algorithm produces and address for a key that is already occupied.

Keys that hash to the same location are called synonyms.

**Hashing Methods**

**Modulo Division**

In this method, the **hash value** is calculated by applying the modulo operator on the key with respect to the size of the hash table

**Digit Extraction**

In the **Digit Extraction** method, a hash value is obtained by extracting digits from the key (such as from a number or a string) and using them to calculate the hash. This is generally useful for more complex keys, such as strings or large numbers.

**Linear Probing: Collision Resolution Technique**

Linear probing is a method of handling **collisions** in a hash table. A **collision** occurs when two keys hash to the same index in the hash table. Linear probing addresses this by finding the next available slot in the table (horizontally) to store the new key.

In simpler terms, when the intended position is already occupied, linear probing moves to the next index, continuing in a linear fashion, until an empty slot is found.

**Clustering** in the context of hashing refers to the situation where a group of hash table slots become consecutively occupied

**Clustering:** Over time, a **primary clustering** issue may arise where consecutive elements form clusters, making it harder to find empty slots. This increases the number of probes.

**Q1:** What is the difference between Modulo Division and Digit Extraction in Hashing?

* **A1:** Modulo Division uses the modulus operation to calculate the hash, while Digit Extraction involves extracting digits from the input key and using them to form a hash.

**Q2:** What is Linear Probing in Hashing?

* **A2:** Linear Probing is a collision resolution technique in hashing where, when a collision occurs, the algorithm checks the next slot (or next available slot) sequentially until an empty one is found.

**4. Stacks & Queues**

**Stacks**

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed.

**Key Operations:**

1. **Push**: Adds an element to the top of the stack.

2. **Pop**: Removes the element from the top of the stack.

3. **Peek/Top**: Returns the element at the top of the stack without removing it.

4. **isEmpty**: Checks if the stack is empty.

5. **Size**: Returns the number of elements in the stack.

**Applications of Stack:**

1. **Undo/Redo**: In text editors, stacks are used to implement undo and redo functionality.

2. **Expression Evaluation**: Used to evaluate expressions, such as infix, prefix, and postfix expressions.

3. **Function Calls**: The system uses a stack to manage function calls (call stack).

4. **Backtracking**: In algorithms like DFS (Depth First Search), stacks are used to remember the previous nodes.

**1. Parentheses Balancing**

Parentheses balancing refers to the process of checking whether parentheses (round, square, or curly) in an expression are correctly paired and nested. It's typically done using a **stack** data structure.

**Steps to check for balanced parentheses**:

1. Traverse the given string.

2. Push every opening parenthesis onto the stack.

3. For every closing parenthesis encountered:

o If the stack is empty, it indicates an unmatched closing parenthesis, so the expression is unbalanced.

o If the stack is not empty, pop the top element from the stack.

4. After the traversal, if the stack is empty, the parentheses are balanced. If the stack is not empty, the expression is unbalanced.

**2. Infix to Postfix Conversion**

Infix expressions are those where the operator is between the operands, like A + B. To evaluate such expressions programmatically, it's more efficient to convert them into **postfix notation** (Reverse Polish Notation), where the operator follows the operands, like A B +.

**Steps for Infix to Postfix Conversion**:

1. Use a **stack** to temporarily hold operators and parentheses.

2. Traverse the infix expression character by character.

3. If the character is an operand (like a variable or number), append it directly to the postfix expression.

4. If the character is an operator, pop operators from the stack and append them to the postfix expression until an operator with less precedence is found, then push the current operator onto the stack.

5. If the character is an opening parenthesis (, push it onto the stack.

6. If the character is a closing parenthesis ), pop operators from the stack and append them to the postfix expression until an opening parenthesis ( is encountered.

7. After processing the entire infix expression, pop all remaining operators from the stack and append them to the postfix expression.

**3. Postfix Evaluation**

Once an infix expression is converted to postfix, it can be easily evaluated using a stack. The idea is to traverse the postfix expression and:

1. Push operands (numbers/variables) onto the stack.

2. When an operator is encountered, pop the required operands from the stack, apply the operator, and push the result back onto the stack.

3. After processing all tokens, the final result will be the only value remaining in the stack.

**Queues**

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means that the first element added to the queue will be the first one to be removed.

**Key Operations:**

1. **Enqueue**: Adds an element to the rear (or end) of the queue.

2. **Dequeue**: Removes an element from the front of the queue.

3. **Front/Peek**: Returns the element at the front of the queue without removing it.

4. **isEmpty**: Checks if the queue is empty.

5. **Size**: Returns the number of elements in the queue.

**Applications of Queue:**

1. **Order Processing**: In print queues or task scheduling, the first task to arrive is the first to be processed.

2. **Breadth-First Search (BFS)**: In graph traversal, queues are used to visit nodes level by level.

3. **Call Center Systems**: Calls are processed in the order they are received, where the first call is answered first.

4. **Handling Requests**: In operating systems, queues are used for managing processes or tasks.

**1. Simple Queue (Linear Queue)**

* **Definition**: A basic queue that follows the **First In, First Out (FIFO)** principle, where elements are added at the rear and removed from the front.
* **Operations**:
  + **Enqueue**: Adds an element to the rear.
  + **Dequeue**: Removes an element from the front.
* **Limitation**: Once the queue is full, the space may remain underutilized even if there is space at the front of the queue. This is known as the problem of **wasting space**.

**2. Circular Queue**

* **Definition**: A **circular queue** overcomes the limitations of the simple queue. The rear of the queue is connected back to the front when the queue is full, forming a circular structure.
* **Operations**:
  + **Enqueue**: Adds an element at the rear, but if the rear reaches the end, it moves to the front if space is available.
  + **Dequeue**: Removes an element from the front.
* **Advantage**: Space is used more efficiently since the unused space at the front can be reused once elements are dequeued.

**3. Priority Queue**

* **Definition**: In a **priority queue**, each element is assigned a **priority**. Elements with higher priority are dequeued before elements with lower priority, regardless of their order in the queue.
* **Operations**:
  + **Enqueue**: Adds an element with an associated priority.
  + **Dequeue**: Removes the element with the highest priority.

**4. Double-Ended Queue (Deque)**

* **Definition**: A **deque** (pronounced "deck") allows elements to be added or removed from **both ends**—the front and the rear—hence the name **double-ended queue**.
* **Operations**:
  + **EnqueueFront**: Adds an element to the front.
  + **EnqueueRear**: Adds an element to the rear.
  + **DequeueFront**: Removes an element from the front.
  + **DequeueRear**: Removes an element from the rear.
* **Use Case**: Deques are used when you need to perform operations at both ends of the queue efficiently.

**Q1:** How is a stack implemented using an array?

* **A1:** In an array-based stack, elements are stored in an array, and a pointer (top) keeps track of the last element inserted. Push and Pop operations are done by modifying the top pointer.

**Q2:** What is a Circular Queue?

* **A2:** A Circular Queue is a queue where the last element is connected back to the first element to form a circle, allowing for efficient utilization of space.

**Q3:** What is a Priority Queue?

* **A3:** A Priority Queue is a type of queue where each element is associated with a priority. Elements with higher priority are dequeued before those with lower priority.

**Q4:** How is Parenthesis Balancing implemented using a stack?

* **A4:** Parenthesis balancing is implemented using a stack by pushing opening parentheses onto the stack and popping them when closing parentheses are encountered. If there is a mismatch, the parentheses are unbalanced.

**Q5:** What is Infix to Postfix conversion?

* **A5:** Infix to Postfix conversion involves converting an expression from infix notation (where operators are between operands) to postfix notation (where operators follow operands) using a stack to temporarily hold operators.

**5. Linked Lists**

A **Linked List** is a linear data structure in which elements, called **nodes**, are stored in a sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations. Each node contains two parts:

1. **Data**: The actual value or information stored in the node.

2. **Reference (or Pointer)**: A reference to the next node in the sequence.

**Types of Linked Lists:**

1. **Singly Linked List**:

o Each node contains data and a reference (pointer) to the next node in the sequence.

o The last node's reference is set to null, indicating the end of the list.

**Structure**:

[Data | Next] -> [Data | Next] -> [Data | null]

2. **Doubly Linked List**:

o Each node contains data, a reference to the next node, and a reference to the previous node.

o It allows traversal in both directions (forward and backward).

**Structure**:

null <- [Prev | Data | Next] <-> [Prev | Data | Next] <-> [Prev | Data | Next] -> null

3. **Circular Linked List**:

o In a circular singly linked list, the last node’s reference points back to the first node.

o In a circular doubly linked list, the first and last nodes are connected, allowing traversal in both directions.

**Structure (Singly Circular)**:

[Data | Next] -> [Data | Next] -> [Data | Next] -> (back to the first node)

**Advantages of Linked Lists:**

1. **Dynamic Size**: Linked lists can grow or shrink in size dynamically, as memory is allocated when a new node is added.

2. **Efficient Insertions/Deletions**: Insertions and deletions are more efficient compared to arrays, especially for operations at the beginning or in the middle of the list. Only pointers need to be adjusted, without shifting elements.

**Disadvantages of Linked Lists:**

1. **Memory Overhead**: Each node requires extra memory to store the pointer/reference in addition to the data.

2. **Sequential Access**: Unlike arrays, linked lists do not support random access to elements. You must traverse the list to access a specific element.

**Q1:** What is the difference between Singly Linked List and Doubly Linked List?

* **A1:** In a Singly Linked List, each node points to the next node, whereas in a Doubly Linked List, each node points to both the next and the previous node, allowing for bidirectional traversal.

**Q2:** What is a Circular Linked List?

* **A2:** A Circular Linked List is a variation where the last node points back to the first node, forming a circle, making it easier to traverse from any point.

**6. Trees**

**Q1:** What is a **Binary Search Tree (BST)?**

* A Binary Search Tree is a tree where each node has at most two children, and the left child is smaller, and the right child is larger than the parent node.

**Properties of a Binary Search Tree:**

1. **Node Structure**:

o Each node in a BST contains three parts:

§ **Data**: The value stored in the node.

§ **Left Child**: A pointer/reference to the left child node (all values in the left subtree are smaller than the node's data).

§ **Right Child**: A pointer/reference to the right child node (all values in the right subtree are greater than the node's data).

2. **Ordering Rule**:

o For each node:

§ All values in the left subtree are **less than** the node’s value.

§ All values in the right subtree are **greater than** the node’s value.

3. **No Duplicates**:

o In a standard binary search tree, duplicate values are not allowed.

**Basic Operations on a BST:**

1. **Search**:

o To find a value in the tree, start from the root:

§ If the value is less than the current node's value, move to the left child.

§ If the value is greater, move to the right child.

§ Continue until the value is found or you reach a leaf node (null).

2. **Insertion**:

o To insert a new node, compare the value to be inserted with the root and recursively move to the left or right child based on the comparison, until a null position is found.

3. **Deletion**:

o Deleting a node in a BST involves three cases:

1. **Node has no children**: Simply remove the node.

2. **Node has one child**: Remove the node and replace it with its child.

3. **Node has two children**: Find the **in-order successor** (the smallest value in the right subtree) or **in-order predecessor** (the largest value in the left subtree), replace the node with this successor/predecessor, and delete the successor/predecessor.

4. **Traversal**:

o **In-order Traversal**: Visit the left subtree, the current node, and then the right subtree. This traversal returns the values in sorted order. (Left,Right,Root)

o **Pre-order Traversal**: Visit the current node, then the left subtree, and finally the right subtree. (Root,Left,Right)

o **Post-order Traversal**: Visit the left subtree, then the right subtree, and finally the current node.(Left, Right, Root)

**Min-Heap**

A **Min-Heap** is a Data Structure with the following properties.

* It is a complete [Complete Binary Tree](https://www.geeksforgeeks.org/complete-binary-tree/).
* The value of the root node must be the smallest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

**Properties of Min Heap:**

1. **Complete Binary Tree**: A min heap is a complete binary tree, meaning all levels of the tree are fully filled except possibly the last, which is filled from left to right.

2. **Min-Heap Property**: For every node, the value of the parent node is smaller than or equal to the values of its children. Formally, if P is the parent node and L and R are its left and right children, then:

P <= L

P <= R

**Max Heap**

A **Max Heap** is a complete binary tree-based data structure that satisfies the **max-heap property**: In a max heap, the value of each parent node is **greater than or equal to the values of its children**. This means that the largest element is always at the **root** of the tree.

**Properties of Max Heap:**

1. **Complete Binary Tree**: A max heap is a complete binary tree, meaning all levels of the tree are fully filled except possibly the last, which is filled from left to right.

2. **Max-Heap Property**: For every node, the value of the parent node is greater than or equal to the values of its children. Formally, if P is the parent node and L and R are its left and right children, then:

P >= L

P >= R

**Q1:** What is the difference between Min-Heap and Max-Heap?

* **A2:** In a Min-Heap, the parent node has a smaller value than its children, whereas in a Max-Heap, the parent node has a larger value than its children.

**7. Graphs**

A **graph** is a data structure used to represent relationships between pairs of objects. It consists of a set of **vertices (or nodes)** and a set of **edges** that connect pairs of vertices. Graphs are used to model many real-world structures, such as social networks, maps, and computer networks.

**Key Components of a Graph:**

1. **Vertices (or Nodes)**: These are the fundamental units in a graph. Each vertex represents an object or entity.

2. **Edges**: An edge is a connection between two vertices. It can represent a relationship or a pathway between the vertices.

**Types of Graphs:**

1. **Directed Graph (Digraph)**: In this type of graph, the edges have a direction. Each edge is an ordered pair, meaning it points from one vertex to another. For example, in a social network, a directed edge might represent a "follower" relationship where person A follows person B.

o Example: A → B means A is connected to B, but not necessarily the other way around.

2. **Undirected Graph**: In an undirected graph, the edges do not have a direction. An edge simply connects two vertices without specifying a direction. This means that if vertex A is connected to vertex B, then vertex B is also connected to vertex A.

o Example: A — B means A is connected to B, and B is also connected to A.

**Adjacency Matrix**

An **Adjacency Matrix** is a way to represent a graph in the form of a 2D matrix (array). In this matrix:

* Each row and column corresponds to a vertex in the graph.
* If 2 vertices are adjacent i.e. there is an edge between them then the matrix intersect value is 1 . If there is no edge the intersect is 0
  + For **directed graphs**, if there is a directed edge from vertex **i** to vertex **j**, the matrix entry at (i, j) is typically **1**, and **0** otherwise.
  + For **undirected graphs**, if there is an edge between vertex **i** and vertex **j**, the matrix entry at (i, j) and (j, i) is **1**, since the edge is bidirectional.

**Breadth-First Traversal (BFS)**

Breadth-First Traversal (BFS) is an algorithm used to traverse or search through a graph or tree. The idea behind BFS is to explore the graph level by level, starting from the root (or any arbitrary node). It uses a queue to explore nodes in the order they are discovered.

**BFS Algorithm (Step-by-Step):**

1. **Initialize**:

o Create an empty queue.

o Mark the starting node as visited.

o Enqueue the starting node.

2. **While the queue is not empty**:

o Dequeue the node from the queue.

o Visit the node and perform the required operation (e.g., print the node's value).

o Enqueue all unvisited adjacent nodes of the dequeued node.

o Mark these adjacent nodes as visited.

**3.** **Repeat until the queue is empty.**

**Depth-First Search (DFS)**

**Depth-First Search (DFS)** is an algorithm for traversing or searching through a graph or tree. Unlike Breadth-First Search (BFS), which explores all nodes at the present depth level before moving to the next level, DFS explores as far as possible along each branch before backtracking.

**How DFS Works:**

1. **Start at the root (or an arbitrary node)**.

2. **Mark the node as visited**.

3. Visit the first unvisited adjacent node and recursively repeat the process for each node.

4. If a node has no unvisited adjacent nodes, backtrack to the previous node and continue the process.

5. Repeat the above steps until all nodes are visited.

DFS can be implemented using **recursion** or using a **stack**. The recursive version naturally mimics the stack structure, which is why it is often the preferred approach.

**Q2:** What is the difference between Breadth-First Traversal (BFS) and Depth-First Traversal (DFS)?

* **A2:** BFS explores all neighbors of a node level by level, while DFS explores as far as possible along each branch before backtracking.