3rd semester, Computer Science & Engineering

Data Structures (TH-3)

MODULE I — Introduction to Data Structures

2 Marks Questions

1. Define data structure?
A data structure is a systematic way of organizing and storing data so that it can be accessed and modified efficiently.

2. What are primitive data types?
Primitive data types are the basic built-in types such as int, float, char, and double that store a single value.

3. List two operations on data structures?
Insertion and Deletion.

4. Define algorithm?
An algorithm is a finite step-by-step sequence of instructions to solve a specific problem.

5. What is asymptotic notation?
Asymptotic notation describes the running time behavior of an algorithm as input size grows, using Big O, Omega (Ω), and Theta (Θ).

6. What is the worst-case complexity?
It is the maximum time taken by an algorithm to execute for any possible input of size n.

7. Define time complexity?
Time complexity represents the total time required by an algorithm as a function of the input size.

8. Give an example of a non-linear data structure?
Trees and Graphs are examples of non-linear data structures.

9. Define space complexity?
Space complexity is the total memory space required by an algorithm to execute completely.

10. What is meant by Abstract Data Type (ADT)?
An ADT defines a data type by its behavior (operations and properties) rather than its implementation.

5 Marks Questions

1. Explain the classification of data structures?

Ans:-

Data structures are ways to organize, store, and manage data efficiently. They are broadly classified as:

Primitive Data Structures

- These are the basic building blocks provided by a programming language.
- **Examples:** int, char, float, double.
- **Characteristics:**
    - Directly supported by the programming language.
    - Used for simple data storage.

Non-Primitive Data Structures

These are more complex and are derived from primitive data types. They can be further classified into:

i. Linear Data Structures

- Elements are stored in a sequential manner, one after another.

- Each element has a unique predecessor and successor (except the first and last elements).
- **Examples:**
    - **Arrays:** Fixed-size collection of elements of the same type.
    - **Stacks:** Follow LIFO (Last In First Out) principle.
    - **Queues:** Follow FIFO (First In First Out) principle.
    - **Linked Lists:** Each element (node) contains data and a pointer to the next element.

ii. Non-Linear Data Structures

- Elements are stored in a hierarchical or interconnected manner.
- Relationships among elements are complex.
- **Examples:**
    - **Trees:** Hierarchical structure with a root node and child nodes (e.g., Binary Trees, AVL Trees).
    - **Graphs:** Represent relationships as nodes (vertices) and connections (edges).

2. Describe various operations performed on data structures?

Ans:

Data structures support several fundamental operations to manipulate and manage data:

1. **Traversal** – Visiting all elements in a specific order (e.g., in-order, pre-order, post-order for trees).
2. **Insertion** – Adding a new element at a specified position.
3. **Deletion** – Removing an element from a data structure.
4. **Searching** – Finding the location of an element.
5. **Sorting** – Arranging elements in a specific order (ascending or descending).
6. **Merging** – Combining two data structures into one.

3. Explain algorithm analysis?

Ans:

Algorithm analysis evaluates the efficiency of an algorithm in terms of:

- **Time Complexity:** How fast an algorithm runs relative to the input size.
- **Space Complexity:** How much memory an algorithm uses relative to the input size.

Asymptotic Notations:

- **Big O (O):** Worst-case performance (upper bound).
- **Omega (Ω):** Best-case performance (lower bound).
- **Theta (Θ):** Average-case performance (tight bound).

4. Differentiate between primitive and non-primitive data structures?

Ans:

Feature	Primitive	Non-Primitive
Definition	Basic built-in data types	Complex or user-defined structures
Examples	int, float, char, double	Array, Stack, Queue, Linked List, Tree, Graph
Complexity	Simple, direct usage	More complex, often requires algorithms to manage
Memory	Occupies fixed memory	Memory may vary depending on structure and size.
Usage	Stores individual values	Stores collection of values and supports operations

5. Discuss the importance of data structures in programming?

Ans:

1. **Efficient Data Management:** Helps store and retrieve data efficiently.
2. **Algorithm Optimization:** Proper data structure choice improves algorithm performance.
3. **Scalability:** Supports handling of large datasets in applications.
4. **Foundation for Advanced Systems:** Essential for databases, operating systems, AI, networks, and more.
5. **Problem Solving:** Enables the design of solutions for complex computational problems.

10 Marks Questions

1. Explain asymptotic analysis and its types with examples?

Asymptotic analysis is a method to evaluate the **performance of an algorithm** as the **input size (n) approaches infinity**.

- Focuses on the **growth rate** of an algorithm rather than exact execution time.
- Helps in **comparing algorithms** and understanding their efficiency for large inputs.

Purpose of Asymptotic Analysis:

- Determines the **time complexity** (how fast an algorithm runs) and **space complexity** (memory usage).
- Ignores constants and lower-order terms as they are insignificant for large inputs.
- Helps in **choosing the most efficient algorithm** for large datasets.

Types of Asymptotic Analysis:

A. Big O Notation (O)

- Represents the **upper bound** of an algorithm's running time.
- Describes the **worst-case scenario**, i.e., the maximum time the algorithm may take.
- Ensures the algorithm will **never take longer than this time** for input of size n.
- **Example:** Bubble Sort → $O(n^2)$
    - Worst-case occurs when the array is in reverse order.
    - Two nested loops → maximum comparisons = $n \times n = n^2$.

B. Omega Notation (Ω)

- Represents the **lower bound** of an algorithm's running time.
- Describes the **best-case scenario**, i.e., minimum time the algorithm will take.
- Guarantees that the algorithm will take **at least this much time**.
- **Example:** Linear Search → $\Omega(1)$
    - Best-case occurs when the element is at the first position.
    - Only one comparison needed.

C. Theta Notation (Θ)

- Represents the **tight bound** of an algorithm's running time.
- Bounds the algorithm from **both above and below**.
- Represents the **average-case performance**.
- **Example:** Binary Search → $\Theta(\log n)$
    - In a sorted array, the search space is halved in each step.
    - Number of steps ≈ $\log_2 n$.

2. Explain all type of Data Structures in detail and write the application of each type?

A Data Structure is a way of organizing, storing, and managing data in a computer so it can be used efficiently.
Example: Array, Stack, Queue, Linked List, Tree, Graph, etc.

Types of Data Structures

1. Linear Data Structure

In a linear data structure, elements are arranged in a sequential manner — one after another.

a) Array

- Definition: Collection of elements of the same data type stored in contiguous memory locations.
- Example: int marks[5] = {90, 80, 70, 85, 95};
- Applications:
    - Used to store lists of data such as marks, salaries, or temperatures.
    - Useful in matrix operations and image processing.
    - Used in searching and sorting algorithms.

b) Linked List

- Definition: Collection of nodes where each node contains data and a reference (link) to the next node.
- Types: Singly Linked List, Doubly Linked List, Circular Linked List.
- Applications:
    - Dynamic memory allocation (no fixed size).
    - Implementation of stacks and queues.
    - Used in navigation systems (previous/next buttons).

c) Stack

- Definition: Linear structure that follows LIFO (Last In, First Out) principle.
- Example: A pile of plates – last plate placed is the first one removed.
- Applications:

    - Function call management in programming (recursion).
    - Undo/Redo operations in text editors.
    - Expression evaluation and syntax parsing.

d) Queue

- Definition: Linear structure that follows FIFO (First In, First Out) principle.
- Example: People standing in a line at a ticket counter.
- Applications:
    - Process scheduling in operating systems.
    - Managing requests in printers or network data packets.
    - Used in BFS (Breadth First Search) algorithm.

2. Non-Linear Data Structure

In a non-linear data structure, elements are not stored sequentially — they are connected in a hierarchical or network manner.

a) Tree

- Definition: Hierarchical structure where data is stored in nodes connected by edges.
- Example: Family tree or folder structure.
- Types: Binary Tree, Binary Search Tree, AVL Tree, etc.
- Applications:
    - Used in databases and file systems.
    - Used in decision-making (Decision Trees).
    - Used in searching and sorting (Binary Search Trees).

b) Graph

- Definition: Set of nodes (vertices) connected by edges.
- Example: Social network (users as nodes, friendships as edges).
- Applications:
    - Used in networking (routing algorithms).
    - Representing maps and paths (shortest path algorithms).
    - Used in social networks, recommendation systems.

MODULE II — Linear Data Structures (Stacks and Queues)

MODULE II — Linear Data Structures (Stacks and Queues)

2 Marks Questions

1. Define stack?
A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.

2. What is a push operation?
It inserts an element onto the top of the stack.

3. What is a pop operation?
It removes the top element from the stack.

4. List two applications of stack?
Function call management and expression conversion (infix to postfix).

5. Define queue?
A queue is a linear data structure that follows the First In, First Out (FIFO) principle.

6. What is a circular queue?
It is a queue in which the last position connects back to the first, forming a circle.

7. Write postfix form of (A + B) * C?
AB+C*

8. What is an overflow condition?
It occurs when an element is pushed into a full stack or queue.

9. What is underflow?
It occurs when an element is removed from an empty stack or queue.

10. Define dequeue?

A double-ended queue that allows insertion and deletion from both ends.

5 Marks Questions

1. Explain array representation of stack?

- A **stack** is a linear data structure that follows **LIFO (Last In First Out)** principle.
- It can be implemented using an **array**.
- A variable top keeps track of the **index of the top element**.

  Initially, top = -1 indicates an empty stack.

- **Push Operation:**
  - Increment top by 1.
  - Insert the new element at stack[top].
- **Pop Operation:**
  - Remove the element at stack[top].
  - Decrement top by 1.
- **Overflow Condition:** top == n-1 (stack full).
- **Underflow Condition:** top == -1 (stack empty).

Example:-

stack[5]; top = -1;

Push 10 → top = 0; stack[0] = 10

Push 20 → top = 1; stack[1] = 20

Pop → element = 20; top = 0

2. Explain applications of stack?

1. **Expression Evaluation:**
   - Used in converting **infix → postfix/prefix** and evaluating expressions.
2. **Function Call Management:**
   - Handles **recursive function calls** using a call stack.
3. **Undo/Redo Operations:**
   - Text editors and software use stacks to **undo or redo actions**.
4. **Syntax Parsing:**
   - Used in **compilers** to check for balanced parentheses.
5. **Backtracking Algorithms:**
   - In **maze solving** or **game AI**, stacks store paths for backtracking.

3. Explain the differences between linear queue and circular queue?

Feature	Linear Queue	Circular Queue
Memory Utilization	May waste space after deletions	Efficient; reuses empty space
Rear Movement	Rear moves linearly	Rear wraps around using modulo (rear + 1) % size
Overflow Condition	rear == n-1	(rear + 1) % size == front
Implementation	Simple	Slightly complex (modulo required)
Use Case	Simple FIFO tasks	Real-time scheduling, buffer management

4. Explain circular queue with example?

1. A **circular queue** is a type of queue in which the **last position is connected back to the first position**, forming a circle.
2. It is implemented using an **array** with two pointers:
   - front → points to the first element of the queue
   - rear → points to the last element of the queue
3. **Purpose:**
   - To **reuse empty spaces** left by dequeued elements in a linear queue.
   - Prevents the **wastage of memory**.
4. **Operations:**

Enqueue (Insertion)

rear = (rear + 1) % size

queue[rear] = element

Dequeue (Deletion):

front = (front + 1) % size

element = queue [front]

5. **Overflow Condition:** (rear + 1) % size == front → Queue is full.
6. **Underflow Condition:** front == -1 → Queue is empty.

5. Differentiate between stack and queue?

Feature	Stack	Queue
Principle	LIFO (Last In First Out)	FIFO (First In First Out)
Insertion	Only at the **top**	At the **rear**
Deletion	Only from the **top**	From the **front**
Access	Only the top element is accessible	Only the front element is accessible
Examples	Undo operation, function call stack	Printer queue, ticket booking system
Use Case	Backtracking, expression evaluation	Task scheduling, buffering

10 Marks Questions

1. Explain expression evaluation using stack?
Ans:

Expression evaluation is the process of **computing the value of a mathematical expression**.

- Expressions can be represented in three forms:
    1. **Infix:** Operators are between operands (e.g., A + B)
    2. **Prefix:** Operators appear **before** operands (e.g., + A B)
    3. **Postfix:** Operators appear **after** operands (e.g., A B +)

- **Stacks** are commonly used for evaluating expressions because they follow **LIFO (Last In First Out)** principle, which is suitable for handling operator precedence and parentheses.

Need for Expression Evaluation

- Direct evaluation of **infix expressions** is difficult due to operator precedence and parentheses.
- Converting to **postfix (or prefix)** simplifies evaluation.
- Stack provides a way to **store operators or operands temporarily** and process them in correct order.

Algorithm for Evaluating Postfix Expression

1. Initialize an empty stack.
2. **Scan the postfix expression from left to right**.
3. **If the symbol is an operand:**
    - Push it onto the stack.
4. **If the symbol is an operator:**
    - Pop the top two elements from the stack.
    - Perform the operation: operand1 operator operand2
    - Push the result back onto the stack.
5. **After scanning the entire expression:**
    - The stack contains the **final result**.

Example of Postfix Evaluation

Expression: 5 6 2 + * 12 4 / -

Step-by-Step Evaluation

Step	Symbol	Action	Stack
1	5	Operand → push	5
2	6	Operand → push	5 6
3	2	Operand → push	5 6 2
4	+	Pop 2,6 → 6+2=8 → push	5 8
5	*	Pop 8,5 → 5*8=40 → push	40
6	12	Operand → push	40 12
7	4	Operand → push	40 12 4
8	/	Pop 4,12 → 12/4=3 → push	40 3
9	-	Pop 3,40 → 40-3=37 → push	37

2. Explain Queue, its Types, Operations, and Applications with Example?

A **queue** is a linear data structure that follows the **FIFO (First In First Out)** principle.

The **first element inserted** is the **first to be removed**.

A queue has two ends:

Front: For deletion

Rear: For insertion

Types of Queues

1. **Linear Queue:**
   - Elements are inserted at rear and removed from front.
   - Simple to implement using arrays.
   - Disadvantage: After some deletions, empty spaces at the front cannot be reused.
2. **Circular Queue:**
   - The last position is connected back to the first position forming a circle.
   - Reuses empty spaces efficiently.
   - Rear and Front positions are updated using modulo operation:
   - rear = (rear + 1) % size
   - front = (front + 1) % size
3. **Priority Queue:**
   - Each element has a **priority**.
   - Element with **highest priority** is served first.
   - Can be implemented using arrays, linked lists, or heaps.
4. **Double-Ended Queue (Deque):**
   - Insertion and deletion can occur at **both ends**.

Operations on Queue

1. **Enqueue (Insertion):**
   - Add an element at the **rear**.

    - Check for **overflow** (queue full).
2. **Dequeue (Deletion):**
    - Remove an element from the **front**.
    - Check for **underflow** (queue empty).
3. **Peek / Front Operation:**
    - Returns the element at the **front** without removing it.
4. **Display:**
    - Print all elements from **front** to **rear**.

Applications of Queue

1. **CPU Scheduling:** Tasks are scheduled in FIFO order.
2. **Printer Queue:** Print jobs are executed in the order they arrive.
3. **Call Center Systems:** Calls are handled in order of arrival.
4. **Buffers in Data Communication:** Circular queues store data temporarily in network buffers.

MODULE III — Linked Lists

MODULE III — Linked Lists

2 Marks Questions

1. Define linked list?
A linked list is a collection of nodes where each node contains data and a pointer to the next node.

2. What is a node?
A node is a structure containing data and address of the next node.

3. Define singly linked list?
Each node points only to the next node in the list.

4. What is a circular linked list?
The last node points back to the first node.

5. Define doubly linked list?
Each node has pointers to both its previous and next nodes.

6. Mention one advantage of linked list over array?
Linked lists allow dynamic memory allocation.

7. What is a NULL pointer?
A pointer that doesn't point to any valid memory location.

8. What are the fields of a node in a singly linked list?
Data field and next pointer.

9. Define self-referential structure?
A structure that contains a pointer to the same type of structure.

10. Mention one application of linked list?
Implementation of stacks, queues, and dynamic memory management.

5 Marks Questions

1. Explain types of linked lists?

Linked lists are linear data structures where elements, called **nodes**, are connected using pointers. Each node contains **data** and a **pointer** to the next node (or previous node). There are three main types:

1. **Singly Linked List (SLL)**
   - Each node has two fields: data and next.
   - next points to the next node.
   - Last node points to NULL.
   - Traversal is **forward only**.
   - Example: head → 10 → 20 → 30 → NULL
2. **Doubly Linked List (DLL)**
   - Each node has three fields: prev, data, and next.
   - prev points to the previous node, next points to the next.
   - Supports **forward and backward traversal**.
   - Example: NULL ← 10 ↔ 20 ↔ 30 → NULL
3. **Circular Linked List (CLL)**
   - Last node points back to the first node.
   - Can be **singly or doubly** circular.
   - Traversal can continue indefinitely in a loop.
   - Example (Singly Circular): 10 → 20 → 30 → 10 → …

2. Write algorithm to insert a node at beginning of singly linked list?

Algorithm:

1. Create a new node (newnode).
2. Assign data to newnode->data.
3. Make newnode->next point to current head.
4. Update head = newnode.

Example:

```
newnode = allocate ()          // Step 1

newnode->data = value          // Step 2

newnode->next = head           // Step 3

head = newnode                 // Step 4
```

3. Explain deletion in single linked list?

Deletion involves removing a node from the list and adjusting pointers so that the list remains connected.

Deleting the First Node (Beginning)

Steps:

1. Check if the list is empty (head == NULL). If yes, deletion is not possible.
2. Create a temporary pointer temp = head.
3. Move the head to the next node: head = head->next.
4. Free memory of the old first node: free(temp).

Diagram:
Before deletion: head -> 10 -> 20 -> 30 -> NULL

After deletion: head -> 20 -> 30 -> NULL

Deleting the Last Node (End)

Steps:

1. Check if list is empty.
2. If only one node exists, free it and set head = NULL.
3. Otherwise, traverse to the second last node (temp).
4. Set temp->next = NULL.
5. Free the last node.

Deleting a Node at a Given Position (Middle)

Steps:

1. Check if list is empty.
2. If deleting the first node, follow **Step 1**.
3. Traverse to the node just before the target position (prev).
4. Set prev->next = node_to_delete->next.
5. Free memory of node_to_delete.

4. Explain linked list representation of stack?

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

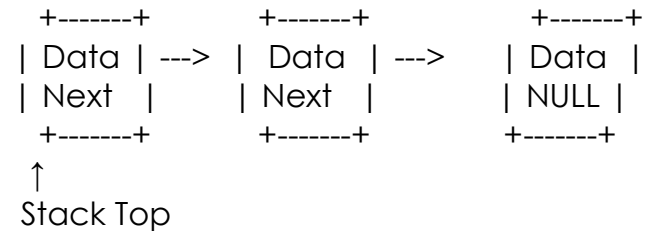
When a stack is implemented using an **array**, its size is **fixed**.
But using a **linked list**, the stack can **grow or shrink dynamically**, depending on the available memory.
No overflow unless memory is full.
Memory is used efficiently.

Representation

Each node in the linked list contains:

- **Data** — the value stored in the stack
- **Pointer (next)** — the address of the next node

The **top pointer** always points to the **topmost node** of the stack.



Basic Operations

a) PUSH (Insert)

- Create a new node.
- Store the data in it.
- Make the new node point to the current top.
- Update top to point to the new node.

b) POP (Delete)

- Check if the stack is empty.
- Store the top node in a temporary pointer.
- Move top to the next node.
- Delete the old top node.

5. List advantages and disadvantages of linked list?

Advantages:

1. **Dynamic Size**: Memory is allocated as needed, unlike arrays.
2. **Efficient Insertion/Deletion**: Adding/removing nodes is fast, especially at beginning/middle.
3. **Flexible Memory Usage**: Nodes can be scattered in memory, no contiguous allocation required.

Disadvantages:

1. **Extra Memory**: Each node stores a pointer, increasing memory usage.
2. **Sequential Access Only**: To access a node, traversal from the head is required.
3. **Complex Implementation**: Requires careful pointer management to avoid memory leaks.

10 Marks Questions

1. Explain operations on singly linked list?

A **singly linked list (SLL)** is a collection of nodes where each node contains **data** and a pointer next pointing to the next node. The list is accessed through a pointer called **head**.

The main operations are **Creation, Insertion, Deletion, and Traversal**.

a) Creation of Singly Linked List

Steps:

1. **Allocate memory for nodes:**
   - Use malloc () in C or new in C++ to create new nodes dynamically.
   - Example: Node* newNode = (Node*)malloc(sizeof(Node));
2. **Assign data to the node:**
   - Store the value to be inserted in the data field of the node.
   - Example: newNode->data = 10;
3. **Set the head pointer:**
   - The first node's pointer next is set to NULL.
   - Head points to the first node: head = newNode;
4. **Add more nodes if needed:**
   - For each new node, link the previous node's next to the new node.

b) Insertion in Singly Linked List

1. Insertion at Beginning:

- Create a new node.
- Set newNode->next = head.
- Update head = newNode.

2. Insertion at End:

- Create a new node.
- Traverse to the last node.
- Set lastNode->next = newNode.
- Set newNode->next = NULL.

3. Insertion at Middle (Position pos):

- Traverse to the node **before** the target position.
- Set newNode->next = prevNode->next.
- Set prevNode->next = newNode.

Deletion in Singly Linked List

1. Deletion at Beginning:

- Use a temporary pointer temp = head.
- Move head = head->next.
- Free memory of temp.

2. Deletion at End:

- Traverse to the second last node.
- Set secondLast->next = NULL.
- Free memory of last node.

3. Deletion at Middle (Position pos):

- Traverse to the node **before** the target node.
- Set prevNode->next = nodeToDelete->next.
- Free memory of nodeToDelete.

d) Traversal of Singly Linked List

Purpose:

- To access and process all nodes in the list.

Steps:

1. Start from head.
2. Repeat until current node is NULL:
    - Access/process current->data.
    - Move to next node: current = current->next.

Uses of Traversal:

- Display list elements.
- Count number of nodes.
- Search for a value in the list.

2. Explain how queues are implemented using linked lists.

Queue Implementation Using Linked List

A **queue** is a **FIFO (First In First Out)** data structure.

- **Insertion** is done at the **rear** (enqueue).
- **Deletion** is done at the **front** (dequeue).

Using a **linked list** allows **dynamic memory allocation**, so the queue can grow or shrink as needed.

1. Structure of a Node

Each node contains:

- data → stores the value.
- next → pointer to the next node.
  Example:-

```
struct Node {

   int data;
```

```
    struct Node* next;
};
```

2. Queue Pointers

- **Front** → points to the first node (for deletion).
- **Rear** → points to the last node (for insertion).
- Initially: front = rear = NULL.

3. Operations on Queue

a) Enqueue (Insertion at Rear)

Steps:

1. Create a new node and assign data.
2. Set newNode->next = NULL.
3. If the queue is empty (front == NULL), set front = rear = newNode.
4. Else, set rear->next = newNode and update rear = newNode.

b) Dequeue (Deletion from Front)

Steps:

1. Check if queue is empty (front == NULL).
2. Store front node in a temporary pointer.
3. Move front = front->next.
4. Free the memory of the deleted node.
5. If front becomes NULL, set rear = NULL (queue is empty).

c) Display Queue

Steps:

1. Start from front.
2. Traverse using next pointer until NULL.
3. Print each node's data.MODULE IV — Non-Linear Data Structures (Trees and Graphs)

MODULE IV — Non-Linear Data Structures (Trees and Graphs)

2 Marks Questions

1. Define tree.
A tree is a hierarchical data structure of nodes connected by edges.

2. Define binary tree.
The binary tree is a hierarchical structure where each node has at most two children — left and right.

3. What is a leaf node?
A node with no children.

4. Define root node.
The topmost node in a tree.

5. What is degree of a node?
Number of children a node has.

6. Define graph.
A graph consists of vertices (nodes) and edges connecting them.

7. What is adjacency matrix?
A 2D matrix showing vertex connections — 1 if connected, else 0.

8. List applications of trees.
Expression evaluation, hierarchical data storage, and searching.

9. What is a complete binary tree?
A binary tree where all levels are completely filled except possibly the last.

10. Define graph traversal.
Systematic visiting of all vertices using BFS or DFS.

5 Marks Questions

1. Explain array representation of binary tree.

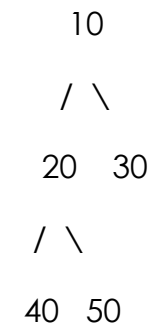
- A **binary tree** can be stored in an array without using pointers.
- Nodes are stored **level by level** (top to bottom, left to right).

Rules:

- Root node is at **index 1**.
- For a node at index i:
  - **Left child** → index = 2 × i
  - **Right child** → index = 2 × i + 1
- Parent of node at index i → floor(i / 2)

Example:

Binary Tree:



Array Representation (index 1-based):

Index: 1 2 3 4 5

Data: 10 20 30 40 50

Advantages:

- Simple to implement.
- Good for **complete binary trees**.

Disadvantages:

- Wastes space for **sparse trees**.
- Insertion/deletion is costly for dynamic trees.

2. Describe preorder, inorder, and postorder traversals.

Traversal: Visiting all nodes of a tree in a systematic way.

1. **Preorder (Root → Left → Right)**
    - Process the **root first**, then traverse left subtree, then right subtree.
    - **Example Output:** 10 20 40 50 30
2. **Inorder (Left → Root → Right)**
    - Traverse **left subtree first**, then process root, then right subtree.
    - **Example Output:** 40 20 50 10 30
    - Produces **sorted order** for a binary search tree (BST).
3. **Postorder (Left → Right → Root)**
    - Traverse **left**, then **right**, then **process root**.
    - **Example Output:** 40 50 20 30 10

Use :

- Preorder → Create a copy of tree, prefix expression.
- Inorder → Get sorted sequence (BST).
- Postorder → Delete tree, postfix expression.

3. What is adjacency matrix? Explain with example.

Definition:

An **adjacency matrix** is a **2D array** used to represent a graph.

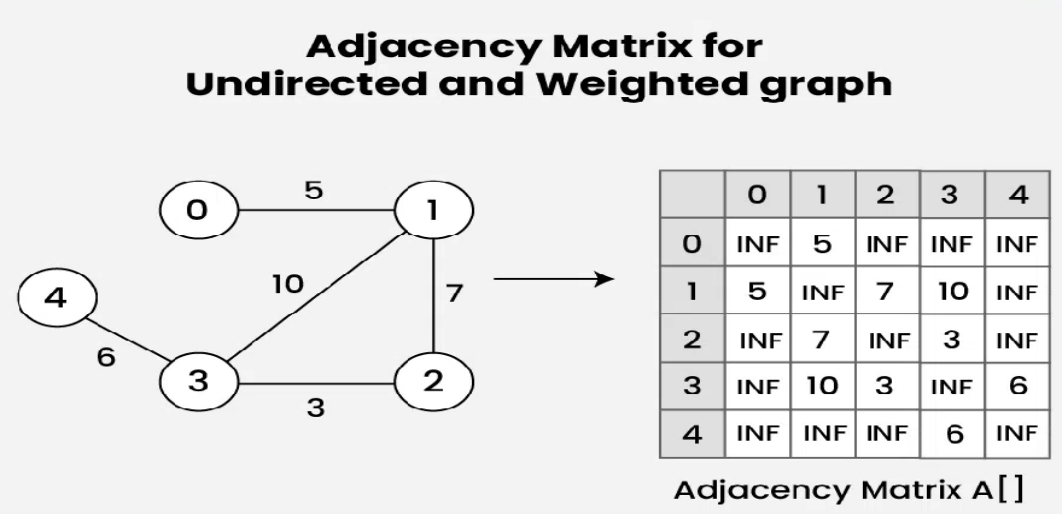
- If the graph has **V vertices**, the matrix is of size **V × V**.
- Each **row and column** corresponds to a vertex.
- The element matrix[i][j] indicates whether there is an **edge from vertex i to vertex j**:
    - 1 (or weight) → edge exists
    - 0 → no edge

Features:

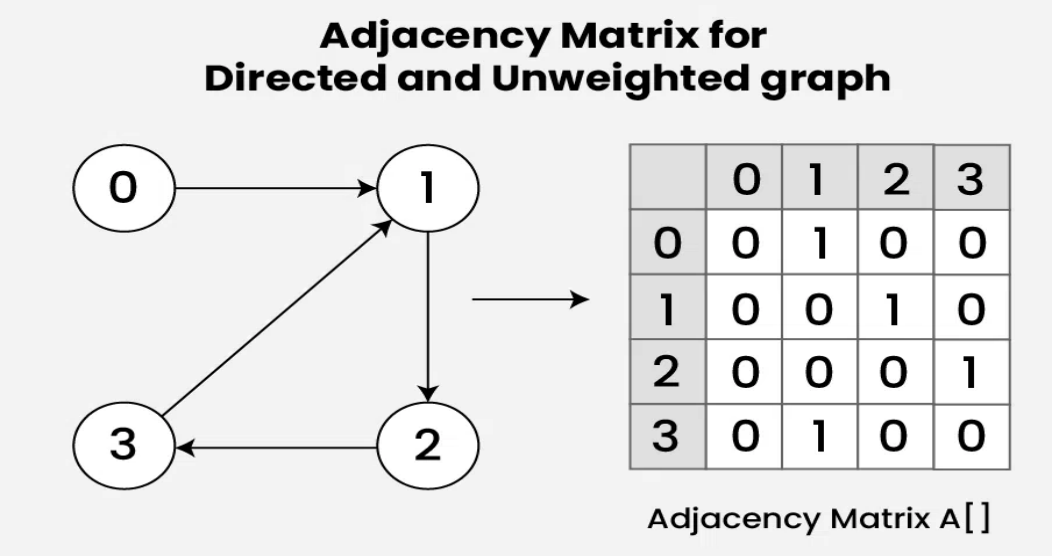
- Works for both **directed** and **undirected** graphs.

- Easy to check if an edge exists between two vertices.
- Space complexity: **O (V^2)**.

Example:-



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

4. Explain the adjacency list representation of a graph?

An **adjacency list** represents a graph using **linked lists**. Each vertex maintains a list of **all vertices directly connected to it by edges**.

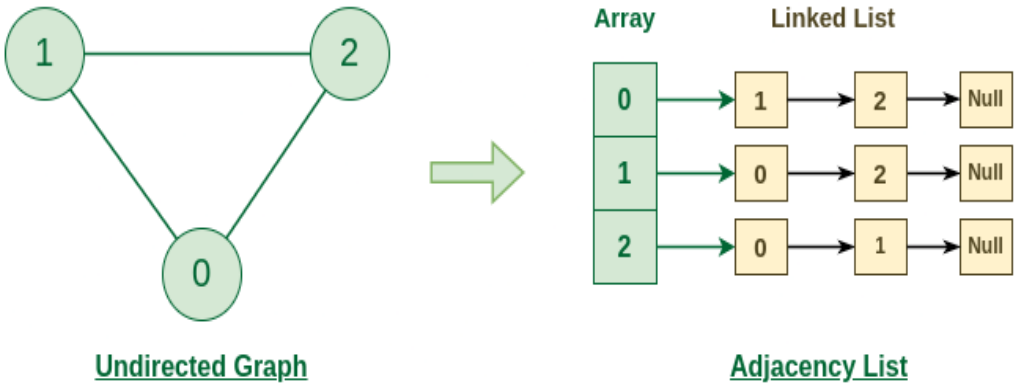
- Efficient for **sparse graphs** (graphs with fewer edges than vertices squared).
- Can represent **directed or undirected graphs**.

Structure

1. **Vertex Array:**
    - An array of size V (number of vertices).
    - Each index corresponds to a vertex.
2. **Linked List for Each Vertex:**
    - Stores all adjacent vertices (neighbors).
    - Each node contains:
        - data → value of adjacent vertex
        - next → pointer to next adjacent vertex

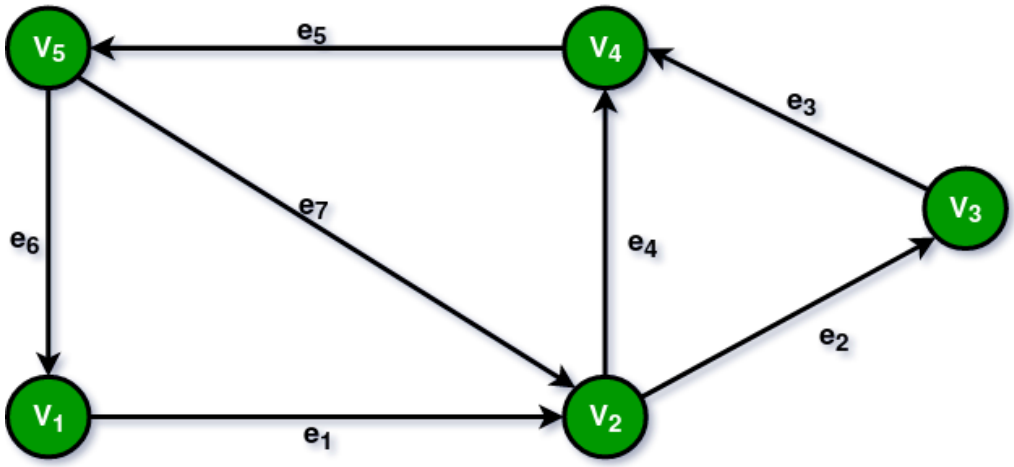
Advantages

1. **Space Efficient:** Uses **O(V + E)** space instead of O(V²).
2. **Dynamic:** Easy to add or remove edges.
3. **Efficient Traversal:** Can efficiently visit all neighbors of a vertex.



Graph Representation of Undirected graph to Adjacency List

5. Calculate the indegree of the following graph?



Indegree(V5) =1 as there is only one arrow with edge e5.
Indegree(V1) =1 as there is only one arrow with edge e6.
Indegree(V2) =2 as there are two arrows with edges e1 and e7.
Indegree(V3) =1 as there is only one arrow with edge e2.

10 Marks Questions

1. What is binary search tree? Explain the insertion operation with an example?

Insertion in a BST means adding a new node to the tree while maintaining the **BST property**:

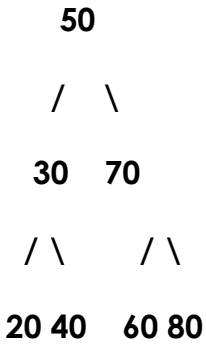
- Left child < Parent
- Right child > Parent

Algorithm (Steps):

1. **Start** from the root node.
2. **Compare** the value to be inserted (key) with the current node's value.
   - If key < current node → go to the **left subtree**.
   - If key > current node → go to the **right subtree**.
3. **Repeat** the process until a NULL position is found.
4. **Insert** the new node at that position.
5. The BST property will remain intact.

Example:

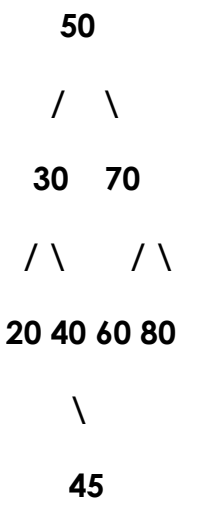
Insert **45** into this BST:



Step-by-step:

- 45 < 50 → go left
- 45 > 30 → go right
- 45 > 40 → insert as **right child of 40**

Result:



2. Discuss BFS algorithms with an example?.

BFS explores level by level using queue.

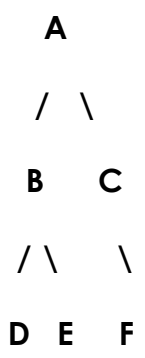
Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices of a graph **in breadth-wise (level-by-level) order**, starting from a given **source vertex**.

It uses a **queue (FIFO)** to keep track of the vertices to visit next.

Algorithm (Using Queue):

1. **Start** with the starting (root) node.
2. **Visit** the node and mark it as **visited**.
3. **Insert** (enqueue) it into a **queue**.
4. **Repeat** until the queue becomes empty:
    - **Dequeue** a node from the front of the queue.
    - **Visit** all unvisited neighbors of that node.
    - **Mark** each as visited and **enqueue** them.

Example Graph



BFS Traversal (starting from A):

Step	Queue Content	Visited Order
1	A	A
2	B, C	A
3	C, D, E	A, B
4	D, E, F	A, B, C
5	E, F	A, B, C, D
6	F	A, B, C, D, E
7	—	A, B, C, D, E, F

Final BFS Order:
A → B → C → D → E → F

****