Shakuntala Krishna Institute Of Technology KD - 64

Subject : - Data Structure Using C & C++

Top 15 Questions

1. Differentiate between linear and non-linear data structures with examples.
2. What is a stack? Explain its primitive operations (PUSH, POP, PEEK) with algorithms.
3. Explain applications of stack such as infix to postfix conversion with algorithm and example.
4. What is a queue? Explain its primitive operations with algorithms.
5. Differentiate between simple queue, circular queue, double-ended queue (deque), and priority queue with examples.
6. What is a linked list? Explain its types with diagrams (singly, doubly, circular).
7. Write algorithms for traversal, insertion, and deletion in a singly linked list.
8. Differentiate between sequential and linked lists with advantages and disadvantages.
9. What are two-way lists? Explain with example.
10. Define binary tree. Explain different tree terminologies (root, leaf, degree, height, etc.) with examples.
11. Write recursive algorithms for tree traversal methods: inorder, preorder, and postorder.
12. What is a binary search tree (BST)? Explain insertion and deletion operations with examples.
13. Differentiate between general tree and binary tree. Explain representation of binary trees in memory.
14. Define a graph. Explain different graph representations (adjacency matrix, adjacency list) with examples.
15. Write algorithms for Breadth First Search (BFS) and Depth First Search (DFS).

1. Differentiate between linear and non-linear data structures with examples.

A **linear data structure** is a structure where data elements are arranged **sequentially**, and each element has a unique **predecessor** and **successor** (except the first and last elements).

Examples of Linear Data Structures:

2. **Array**
   A fixed-size collection of elements of the same type stored in **contiguous memory**.
3. **Linked List**
   A dynamic collection where each element (node) contains **data + a pointer to the next node**.
4. **Stack** (LIFO – Last In, First Out)
   Example: Undo operations in text editors.
5. **Queue** (FIFO – First In, First Out)
   Example: Print queue, customer waiting line.

A **non-linear data structure** is one in which data elements are **not arranged sequentially**.
Here, one element may be connected to **multiple elements**.

1. **Tree**

   A **hierarchical structure** with a root node and child nodes.

   Example: File system (folders & subfolders).

2. **Graph**

   A collection of **nodes (vertices)** and **edges (connections)**.

   Example: Social networks (Facebook friends), Google Maps routes.

S.NO	Linear Data Structure	Non-linear Data Structure
1.	In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.

S.NO	Linear Data Structure	Non-linear Data Structure
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.
8.	Linear data structures are useful for simple data storage and manipulation.	Non-linear data structures are useful for representing complex relationships and data hierarchies, such as in social networks, file systems, or computer networks.
9.	Performance is usually good for simple operations like adding or removing at the ends, but slower for operations like searching or removing elements in the middle.	Performance can vary depending on the structure and the operation, but can be optimized for specific operations.

2. What is a stack? Explain its primitive operations (PUSH, POP, PEEK) with algorithms.

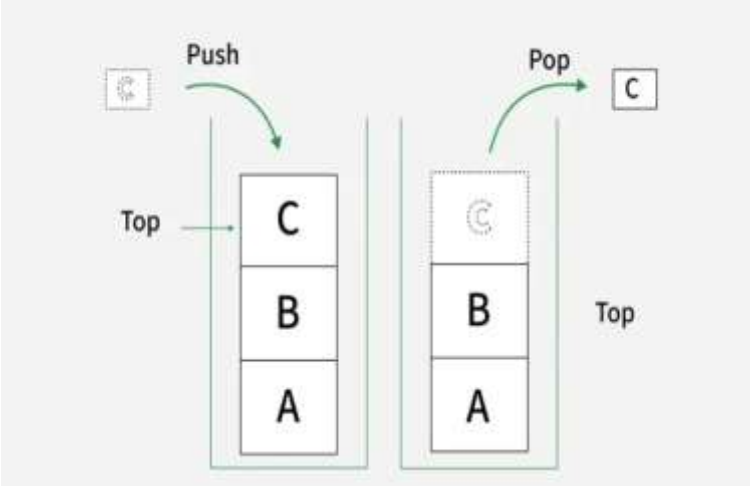
A **Stack** is a linear data structure that follows a particular order in which the operations are performed. The order may be **LIFO(Last In First Out)** or **FILO(First In Last Out)**. **LIFO** implies that the element that is inserted last, comes out first and **FILO** implies that the element that is inserted first, comes out last.

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

This means the **last element inserted** into the stack is the **first one to be removed**.

Real-life Example:

- A stack of plates: The plate placed last on the top is the first one removed.



Primitive Operations of Stack

The main operations performed on a stack are:

1. **PUSH (Insert an element into stack)**
    - Adds an element to the **top** of the stack.
    - If the stack is full → **Overflow condition**.
2. **POP (Remove an element from stack)**
    - Removes and returns the **top element** of the stack.
    - If the stack is empty → **Underflow condition**.
3. **PEEK / TOP (View the top element without removing it)**
    - Returns the value of the **top element** without deleting it.

Stack Representation

A stack can be represented using:

- **Array** (fixed size stack)

- **Linked List** (dynamic stack)

Algorithms of Stack Operations

1. PUSH Operation

Algorithm PUSH(stack, item)

1. IF TOP == MAX-1

    PRINT "Overflow"

    EXIT

2. ELSE

    TOP ← TOP + 1

    stack[TOP] ← item

3. END IF

---

2. POP Operation

Algorithm POP(stack)

1. IF TOP == -1

    PRINT "Underflow"

    EXIT

2. ELSE

    item ← stack[TOP]

    TOP ← TOP - 1

    RETURN item

3. END IF

---

3. PEEK Operation

Algorithm PEEK(stack)

1. IF TOP == -1

    PRINT "Stack is Empty"

    EXIT

2. ELSE

    RETURN stack[TOP]

3. END IF

C Program Example: Stack using Array

```
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;

// Function to push element
void push(int item) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        top++;
        stack[top] = item;
        printf("%d pushed to stack\n", item);
    }
}

// Function to pop element
void pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
    } else {
        printf("%d popped from stack\n", stack[top]);
        top--;
    }
}

// Function to peek
void peek() {
    if (top == -1) {
```

```
        printf("Stack is Empty\n");
    } else {
        printf("Top element is %d\n", stack[top]);
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    peek();
    pop();
    peek();
    return 0;
}
```

- **Output**

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
Top element is 30
30 popped from stack
Top element is 20
```

3. Explain applications of stack such as infix to postfix conversion with algorithm and example.

Infix expression is a common way of writing mathematical expressions where operators are written between the operands whereas postfix is a type of expression in which a pair of operands is followed by an operator.

Applications of Stack

Stacks are widely used in computer science. Some common applications are:

1. **Expression Evaluation** (Infix → Postfix / Prefix conversion)
2. **Function Calls & Recursion**
3. **Undo/Redo operations** in editors
4. **Backtracking** (Maze solving, Chess, etc.)
5. **Parenthesis Matching**
6. **Compiler Syntax Parsing**

Here, we will focus on **Infix to Postfix conversion**.

1. What is Infix and Postfix?

- **Infix Expression:** Operator comes **between operands**.
  Example: A + B * C
- **Postfix Expression (Reverse Polish Notation):** Operator comes **after operands**.
  Example: A B C * +

2. Algorithm: Infix to Postfix Conversion (Using Stack)

We use a **stack** to store operators temporarily until they can be added to the postfix expression.

Algorithm (Shunting Yard method):

1. Initialize an empty stack for operators
2. Scan the infix expression from left to right:
   a) If the symbol is an operand → Add it to postfix expression
   b) If the symbol is '(' → Push it onto the stack
   c) If the symbol is ')' → Pop from stack until '(' is encountered
   d) If the symbol is an operator (+, -, *, /, ^):
      - While stack is not empty AND precedence(top of stack) >= precedence(current operator):
        Pop from stack and add to postfix expression
      - Push the current operator onto stack
3. After scanning expression, pop and add all remaining operators from stack to postfix

Example: Convert Infix → Postfix

Expression:

A + B * C

Step by Step:

1. Read A → Operand → Postfix: A
2. Read + → Operator → Push to stack
3. Read B → Operand → Postfix: A B
4. Read * → Operator → Higher precedence than +, push to stack
5. Read C → Operand → Postfix: A B C
6. End → Pop all operators → * +

Final Postfix:

A B C * +

Example with Parentheses

Infix:

(A + B) * C

Steps:

1. (→ Push to stack
2. A → Operand → Postfix: A
3. + → Push to stack
4. B → Operand → Postfix: A B
5.) → Pop until (→ Postfix: A B +
6. * → Push to stack
7. C → Operand → Postfix: A B + C
8. End → Pop *

✅ **Final Postfix:**

A B + C *

4. What is a queue? Explain its primitive operations with algorithms.

A Queue Data Structure is a fundamental concept in computer science used for storing and managing data in a specific order.

- It follows the principle of "First in, First out" (FIFO), where the first element added to the queue is the first one to be removed.
- It is used as a buffer in computer systems where we have speed mismatch between two devices that communicate with each other. For example, CPU and keyboard and two devices in a network
- Queue is also used in Operating System algorithms like CPU Scheduling and Memory Management, and many standard algorithms like Breadth First Search of Graph, Level Order Traversal of a Tree.

Primitive Operations on Queue

The main operations performed on a queue are:

1. **ENQUEUE (Insert an element into queue)**
    - Adds an element at the **rear (end)** of the queue.
    - If the queue is full → **Overflow condition**.
2. **DEQUEUE (Remove an element from queue)**
    - Removes an element from the **front (beginning)** of the queue.
    - If the queue is empty → **Underflow condition**.
3. **PEEK / FRONT (View the front element without removing it)**
    - Returns the **first element** of the queue without deleting it.
4. **IsEmpty / IsFull**
    - Checks whether the queue is empty or full.

Queue Representation

A queue can be represented using:

- **Array** (fixed size)
- **Linked List** (dynamic size)

Algorithms for Queue Operations (Array Implementation)

1. ENQUEUE (Insert)

Algorithm ENQUEUE(Q, item)

1. IF REAR == MAX-1
   PRINT "Queue Overflow"
   EXIT
2. ELSE
   IF FRONT == -1
      FRONT ← 0
   REAR ← REAR + 1
   Q[REAR] ← item
3. END IF

---

2. DEQUEUE (Remove)

Algorithm DEQUEUE(Q)

1. IF FRONT == -1 OR FRONT > REAR
   PRINT "Queue Underflow"
   EXIT
2. ELSE
   item ← Q[FRONT]
   FRONT ← FRONT + 1
   RETURN item
3. END IF

---

3. PEEK (Front element)

Algorithm PEEK(Q)

1. IF FRONT == -1
   PRINT "Queue is Empty"
   EXIT
2. ELSE
   RETURN Q[FRONT]
3. END IF

C Program: Queue using Array

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

// Enqueue
void enqueue(int item) {
    if (rear == MAX - 1) {
        printf("Queue Overflow\n");
    } else {
        if (front == -1) front = 0;
        rear++;
        queue[rear] = item;
        printf("%d enqueued to queue\n", item);
    }
}

// Dequeue
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
    } else {
        printf("%d dequeued from queue\n", queue[front]);
        front++;
    }
}

// Peek
void peek() {
```

```
    if (front == -1 || front > rear) {
        printf("Queue is Empty\n");
    } else {
        printf("Front element is %d\n", queue[front]);
    }
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    peek();
    dequeue();
    peek();
    return 0;
}
```

Output

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Front element is 10
10 dequeued from queue
Front element is 20
```

5. Differentiate between simple queue, circular queue, double-ended queue (deque), and priority queue with examples.

1. Queue

Definition

A **queue** is a **linear data structure** that follows the **FIFO (First In, First Out)** principle.
The element inserted first will be removed first.

Operations

- **Enqueue** → Insert element at the **rear**
- **Dequeue** → Remove element from the **front**
- **Peek** → View the front element

Example (Normal Queue using array)

Enqueue(10) → [10]

Enqueue(20) → [10, 20]

Enqueue(30) → [10, 20, 30]

Dequeue() → [20, 30] (10 removed)

---

2. Circular Queue

Definition

In a **circular queue**, the **last position connects back to the first position**, forming a circle.
It solves the problem of wasted space in a normal queue.

How it works

- rear moves circularly: (rear + 1) % size
- front moves circularly: (front + 1) % size

Example (Size = 5)

Initial: [ ] [ ] [ ] [ ] [ ]

Enqueue(10) → [10] [ ] [ ] [ ] [ ]

Enqueue(20) → [10] [20] [ ] [ ] [ ]

Enqueue(30) → [10] [20] [30] [ ] [ ]

Dequeue() → [ ] [20] [30] [ ] [ ] (10 removed)

Enqueue(40) → [ ] [20] [30] [40] [ ]

Enqueue(50) → [ ] [20] [30] [40] [50]

Enqueue(60) → [60] [20] [30] [40] [50] (wraps around → circular)

---

3. Double-Ended Queue (Deque)

Definition

A **Deque (Double-Ended Queue)** is a linear data structure where insertion and deletion can be done **from both ends (front and rear)**.

Types of Deque

1. **Input Restricted Deque** → Insertion at **one end only**, deletion at **both ends**.
2. **Output Restricted Deque** → Deletion at **one end only**, insertion at **both ends**.

Example

Initial: [ ]

InsertRear(10) → [10]

InsertRear(20) → [10, 20]

InsertFront(5) → [5, 10, 20]

DeleteRear() → [5, 10]

DeleteFront() → [10]

---

4. Priority Queue

Definition

A **priority queue** is a special queue where each element has a **priority**.
The element with **higher priority** is dequeued first (not just based on arrival order).
If two elements have the same priority, then **FIFO** order is used.

Implementation

- Using **Array** (sorted/unsorted)
- Using **Heap** (efficient method)

Example

Insert(10, priority 2) → [(10,2)]

Insert(20, priority 1) → [(10,2), (20,1)]

Insert(30, priority 3) → [(10,2), (20,1), (30,3)]

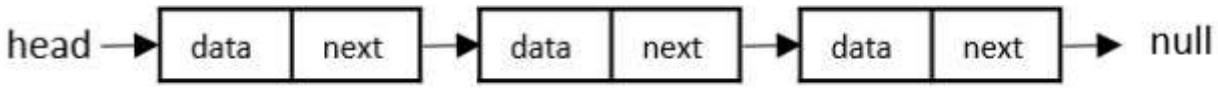
Dequeue() → removes (30, priority 3) → highest priority

Remaining: [(10,2), (20,1)]

6. **What is a linked list? Explain its types with diagrams (singly, doubly, circular).**

A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers. Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations. A node consists of the data value and a pointer to the address of the next node within the linked list.

A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently. Linked lists can be used to implment various data structures like a stack, queue, graph, hash maps, etc.



A linked list starts with a **head** node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to **null** indicating the end of the list.

A. Singly Linked List (SLL)

- Each node contains **data** and a **pointer to the next node**.
- Traversal is **forward only** (from head to last node).

Structure of Node

[Data | Next] -> [Data | Next] -> [Data | Next] -> NULL

Diagram

Head -> 10 -> 20 -> 30 -> NULL

B. Doubly Linked List (DLL)

- Each node contains: **Data**, **Pointer to Next**, **Pointer to Previous**.
- Traversal is **both forward and backward**.

Structure of Node

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

Diagram

NULL <- 10 <-> 20 <-> 30 -> NULL

C. Circular Linked List (CLL)

- The **last node points back to the first node**, forming a circle.
- Can be **singly circular** (last node points to first)
- Can be **doubly circular** (both next and prev pointers, last points to first and first points to last)
-

Singly Circular Linked List Diagram

Head -> 10 -> 20 -> 30

Doubly Circular Linked List Diagram

10 <-> 20 <-> 30

5. Example: Singly Linked List Node in C

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // Allocate nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    // Assign data and link nodes
    head->data = 10;
    head->next = second;
```

```
    second->data = 20;
    second->next = third;

    third->data = 30;
    third->next = NULL;

    // Print list
    struct Node* ptr = head;
    while (ptr != NULL) {
        printf("%d -> ", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");

    return 0;
}
```

Output:

```
10 -> 20 -> 30 -> NULL
```

7. Write algorithms for traversal, insertion, and deletion in a singly linked list.

1. Traversal in Singly Linked List

Algorithm

Algorithm: TraverseSLL(head)

1. Set temp = head
2. WHILE temp != NULL
    a. Print temp->data
    b. temp = temp->next
3. END WHILE

Explanation:

- Start from **head** node.
- Move to the **next node** until you reach NULL.
- Access or print each node's data during traversal.

---

2. Insertion in Singly Linked List

There are **3 main cases**:

A. Insertion at the Beginning

Algorithm: InsertAtBeginning(head, newData)

1. Create newNode
2. newNode->data = newData
3. newNode->next = head
4. head = newNode

B. Insertion at the End

Algorithm: InsertAtEnd(head, newData)

1. Create newNode
2. newNode->data = newData
3. newNode->next = NULL
4. IF head == NULL
    head = newNode
  ELSE

```
    temp = head
    WHILE temp->next != NULL
       temp = temp->next
    END WHILE
    temp->next = newNode
```

C. Insertion After a Given Node (or Position)

Algorithm: InsertAfterNode(prevNode, newData)

```
1. IF prevNode == NULL
       PRINT "Error: previous node cannot be NULL"
       EXIT
2. Create newNode
3. newNode->data = newData
4. newNode->next = prevNode->next
5. prevNode->next = newNode
```

---

3. Deletion in Singly Linked List

There are **3 main cases**:

A. Deletion at the Beginning

Algorithm: DeleteAtBeginning(head)

```
1. IF head == NULL
       PRINT "List is empty"
       EXIT
2. temp = head
3. head = head->next
4. FREE temp
```

B. Deletion at the End

Algorithm: DeleteAtEnd(head)

```
1. IF head == NULL
       PRINT "List is empty"
```

EXIT

2. IF head->next == NULL

FREE head

head = NULL

EXIT

3. temp = head

4. WHILE temp->next->next != NULL

temp = temp->next

5. FREE temp->next

6. temp->next = NULL

C. Deletion of a Given Node (by value)

Algorithm: DeleteNode(head, key)

1. temp = head, prev = NULL

2. IF temp != NULL AND temp->data == key

head = temp->next

FREE temp

EXIT

3. WHILE temp != NULL AND temp->data != key

prev = temp

temp = temp->next

4. IF temp == NULL

PRINT "Key not found"

EXIT

5. prev->next = temp->next

6. FREE temp

8. Differentiate between sequential and linked lists with advantages and disadvantages.

Data structures are used to **store** and **organize** data in a way that makes it easy to access, modify, and analyze.

Sequential List in Data Structure

Definition

A **Sequential List** is a **linear data structure** in which elements are stored in **contiguous memory locations** (continuous blocks of memory).

- It is usually implemented using **arrays**.
- Each element can be directly accessed using its **index (position)**.
- Example: A[0], A[1], A[2], … A[n-1].

Characteristics of Sequential List

1. **Contiguous Storage** → All elements are stored one after another in memory.
2. **Index-based Access** → Any element can be accessed directly using its index.
    - Example: A[4] gives the 5th element instantly.
3. **Fixed Size (Static Arrays)** → The size is defined at the time of creation and cannot change.
4. **Homogeneous Elements** → Stores same type of data (e.g., all integers, all floats).
5. **Order Preserved** → Elements are stored in a fixed order.

Operations on Sequential List

1. **Traversal** → Visiting each element one by one.
    - Time Complexity: **O(n)**
2. **Insertion** → Adding a new element.
    - If at end → **O(1)** (if space available)
    - If at beginning or middle → **O(n)** (shift elements to right)
3. **Deletion** → Removing an element.
    - If at end → **O(1)**
    - If at beginning or middle → **O(n)** (shift elements to left)
4. **Searching**
    - Linear Search → **O(n)**
    - Binary Search (only if sorted) → **O(log n)**
5. **Updation** → Changing value at a specific index → **O(1)**

Advantages of Sequential List

1. **Direct Access** → Any element can be accessed in constant time using index.
2. **Cache Friendly** → Since elements are stored continuously, CPU cache utilization is better → faster performance.
3. **Simplicity** → Easy to implement and use.
4. **Less Overhead** → No extra memory required for pointers (unlike linked list).
5. **Efficient for Small & Fixed Size Data** → Works very well when size of data is known in advance.

Disadvantages of Sequential List

1. **Fixed Size** → Cannot grow or shrink dynamically (static arrays).
2. **Wastage of Memory** → If allocated size is large but not fully used → memory wasted.
3. **Costly Insertion/Deletion** → Requires shifting of elements when inserting/deleting in middle.
    - Example: Inserting at index 0 requires shifting **all elements** right.
4. **Resizing is Expensive** → If array is full, new larger array must be created and old elements copied.
5. **Contiguous Memory Requirement** → Needs a large block of continuous memory which may not always be available.

Linked List in Data Structure

Definition

A **Linked List** is a **linear data structure** in which elements are stored in **non-contiguous memory locations**.

- Each element is called a **Node**.
- Every node has **two parts**:
    1. **Data** → actual value stored.
    2. **Pointer/Link** → address of the next node.

👉 Thus, nodes are connected through pointers, forming a chain-like structure.

Structure of a Node (in C-like pseudocode)

```
struct Node {
   int data;        // stores value
   struct Node* next; // pointer to next node
};
```

Types of Linked Lists

1. **Singly Linked List**
    - Each node has **data** + **pointer to next node**.
    - Last node's pointer = NULL.
    - Traversal is **only forward**.

Example: 10 → 20 → 30 → NULL

---

2. **Doubly Linked List**
    - Each node has **three fields**:
        - Data
        - Pointer to previous node
        - Pointer to next node
    - Traversal possible **both directions** (forward & backward).

Example: NULL ← 10 ↔ 20 ↔ 30 → NULL

---

3. **Circular Linked List**
    - Last node points back to **first node** instead of NULL.
    - Can be singly or doubly circular.
    - Traversal is continuous in a loop.

Example (Singly Circular): 10 → 20 → 30 → back to 10

---

4. **Header Linked List**
    - Special node called **header node** at start, stores metadata (like size of list).

Operations on Linked List

1. **Traversal**
    - Start from head node, visit each node until NULL.
    - Time Complexity: **O(n)**
2. **Insertion**
    - At beginning → Create new node, point it to head, update head. (**O(1)**)
    - At end → Traverse till last node, update pointer. (**O(n)**)
    - At given position → Traverse to position, adjust links. (**O(n)**)
3. **Deletion**

- At beginning → Move head to next node. (**O(1)**)
- At end → Traverse till second-last node, set its pointer = NULL. (**O(n)**)
- At given position → Traverse, unlink target node. (**O(n)**)

4. **Searching**
   - Traverse node by node until data found. (**O(n)**)
5. **Updation**
   - Traverse to node and update its data. (**O(n)**)

Advantages of Linked List

1. **Dynamic Size** → Can grow/shrink at runtime without memory wastage.
2. **Efficient Insertion/Deletion** → No shifting like arrays, just pointer changes.
3. **No Memory Wastage** → Uses only as much memory as required.
4. **Flexibility** → Easy to implement stacks, queues, hash chains, adjacency lists etc.
5. **Useful for Variable-Sized Data** → Strings, polynomials, etc.

Disadvantages of Linked List

1. **Sequential Access Only** → Random access not possible, must traverse from head. (**O(n)** access time).
2. **Extra Memory Overhead** → Each node stores pointer(s), increasing memory usage.
3. **Cache Performance Poor** → Nodes scattered in memory, less CPU cache friendly.
4. **Complex Implementation** → Pointer handling is error-prone.
5. **Reverse Traversal Difficult** in singly linked list (better in doubly list).

9. **What are two-way lists? Explain with example.**

Two-Way List (Doubly Linked List)

A Two-Way List (also called a Doubly Linked List) is a type of linked list in which each node contains two pointers:

- One pointer points to the previous node
- Another pointer points to the next node

Structure of a Node (in C-like pseudocode)

```
struct Node {
  int data;         // stores value
  struct Node* prev;   // pointer to previous node
  struct Node* next;   // pointer to next node
};
```

Diagram Representation

NULL ← [Prev|10|Next] ↔ [Prev|20|Next] ↔ [Prev|30|Next] → NULL

- **Here,**
  - Node 10 has prev = NULL (first node)
  - Node 20 has prev → 10 and next → 30
  - Node 30 has next = NULL (last node)

Operations on Two-Way List

1. **Traversal**
   - Forward traversal: Start from head → follow next pointers until NULL.

- Backward traversal: Start from tail → follow prev pointers until NULL.
- Time Complexity: O(n)

2. **Insertion**
   - At beginning: Create new node, adjust head's prev, update head. (O(1))
   - At end: Traverse to last, update last node's next. (O(n), or O(1) if tail pointer used).
   - In middle: Adjust prev and next pointers of neighboring nodes. (O(n))
3. **Deletion**
   - At beginning: Move head to head→next, set new head's prev = NULL. (O(1))
   - At end: Update second-last node's next = NULL. (O(n) or O(1) with tail pointer).
   - At middle: Update prev and next pointers of surrounding nodes. (O(n))
4. **Searching**
   - Sequential search (forward or backward).
   - Time Complexity: O(n)

Advantages of Two-Way List (over Singly Linked List)

1. Bi-directional Traversal → Forward and backward both possible.
2. Easier Deletion → Node can be deleted without traversing from head (if pointer to node is given).
3. Efficient Insertion/Deletion → No need to traverse for predecessor node.
4. More Flexible → Useful for applications like undo/redo, navigation (next & previous).

Disadvantages of Two-Way List

1. Extra Memory → Each node stores two pointers (prev and next).
2. More Complex → Insertion and deletion require careful updating of two links.

3. Overhead in Memory → Uses more memory compared to singly linked list.
4. Performance Impact → Slightly slower due to extra pointer management.

Example of Two-Way List

Suppose we want to store students' roll numbers: 101, 102, 103

NULL ← [NULL|101|*] ↔ [*|102|*] ↔ [*|103|NULL]

- First node (101) → prev = NULL, next → 102
- Middle node (102) → prev → 101, next → 103
- Last node (103) → next = NULL, prev → 102

👉 **Traversal forward: 101 → 102 → 103**

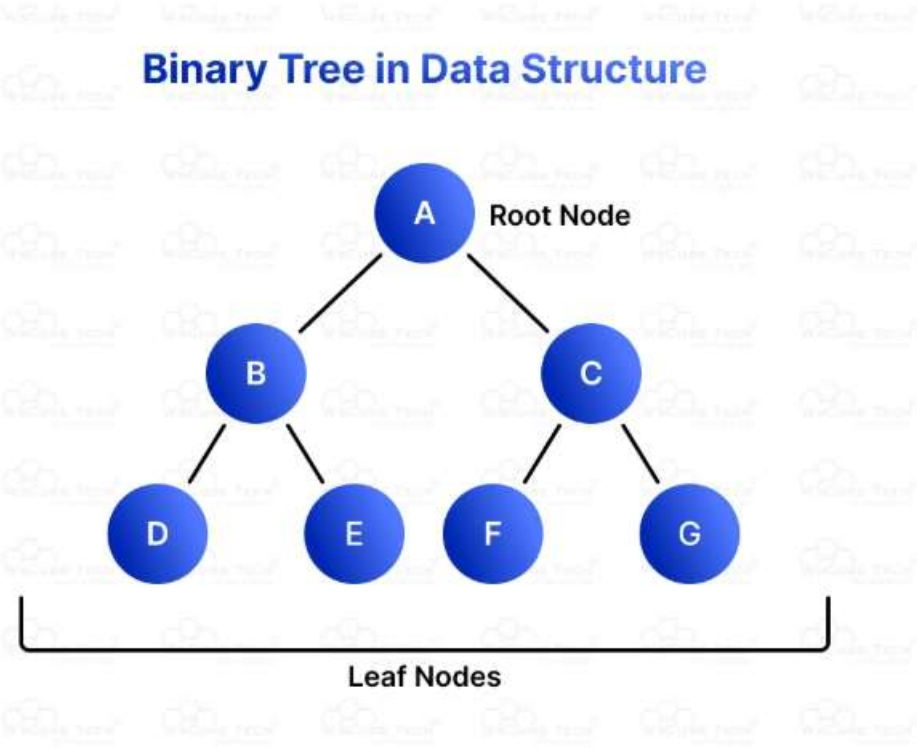
👉 **Traversal backward: 103 → 102 → 101**

Applications of Two-Way List

- Undo/Redo operations in text editors.
- Music/Video Playlists (previous and next track navigation).
- Browser History Navigation (backward and forward buttons).
- Deque (Double-Ended Queue) implementation.
- Complex Data Structures like Fibonacci heaps.

10. **Define binary tree. Explain different tree terminologies (root, leaf, degree, height, etc.)** A A binary tree in DSA (Data Structures and Algorithms) is a way to organize data in a hierarchical structure. In a binary tree, each node has at most two children, called the left child and the right child. The topmost node is called the root, and the nodes with no children are called leaves.

The basic idea of a binary tree is to have a parent-child relationship between nodes. Each node can have a left and a right child, and this pattern continues down the tree. This structure makes it easy to organize and find data quickly.



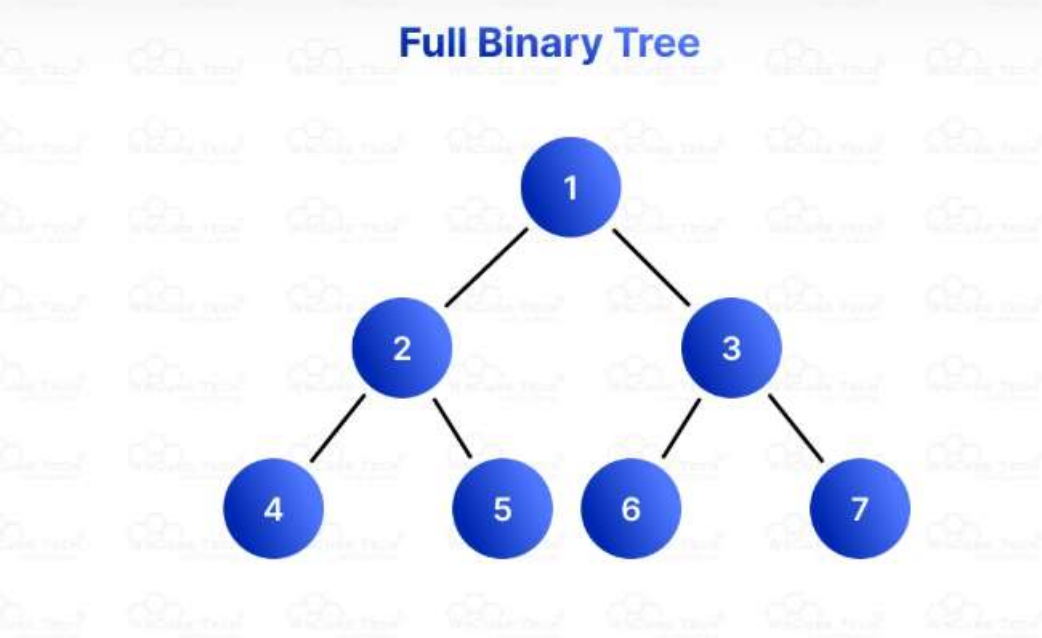
Types of Binary Tree in Data Structure

Let's understand what are the different types of binary tree with examples:

Full Binary Tree

A full binary tree is a tree where every node has either 0 or 2 children.

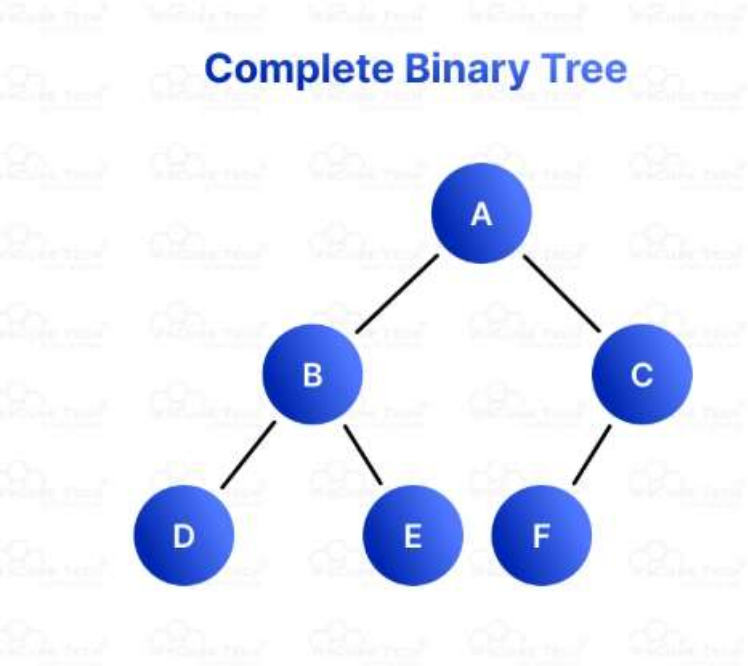
Example:



Complete Binary Tree

A **complete binary tree** is a tree where all levels are fully filled except possibly the last level, which is filled from left to right.

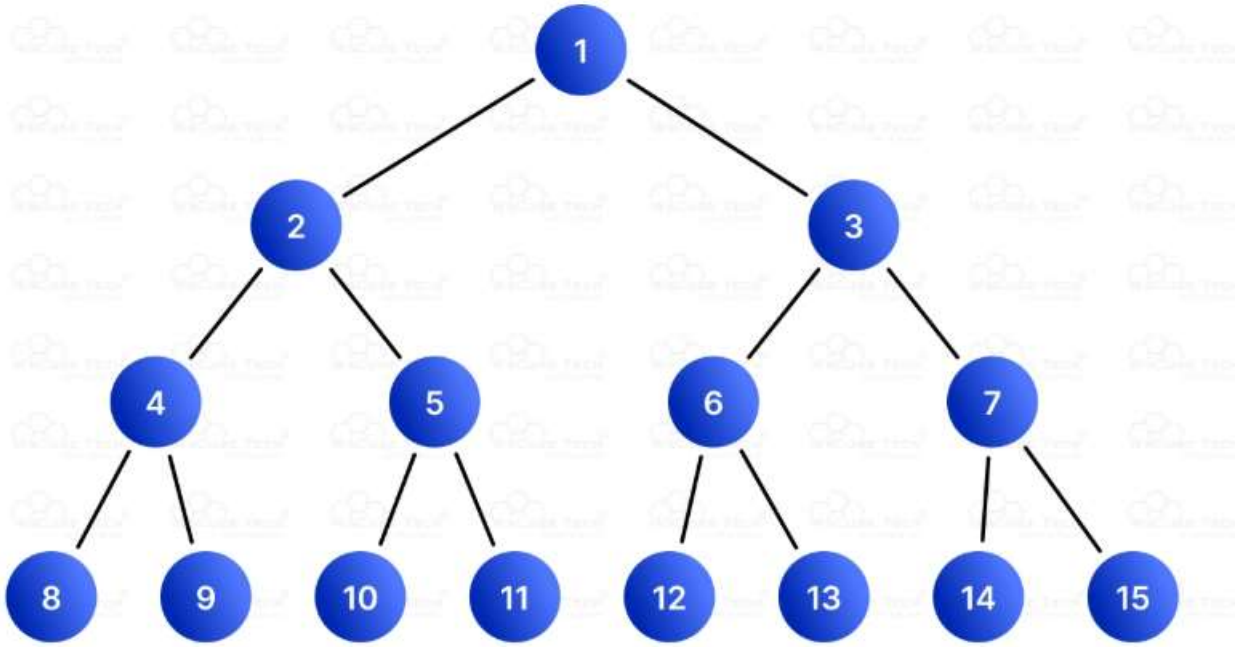
Example:



Perfect Binary Tree

A **perfect binary tree** is a tree where all internal nodes have exactly two children and all leaf nodes are at the same level.

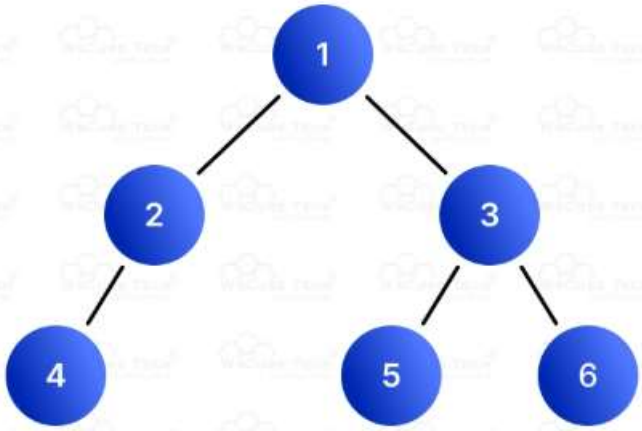
Example:



Balanced Binary Tree

A **balanced binary tree** is a tree where the height of the left and right subtrees of any node differ by at most one.

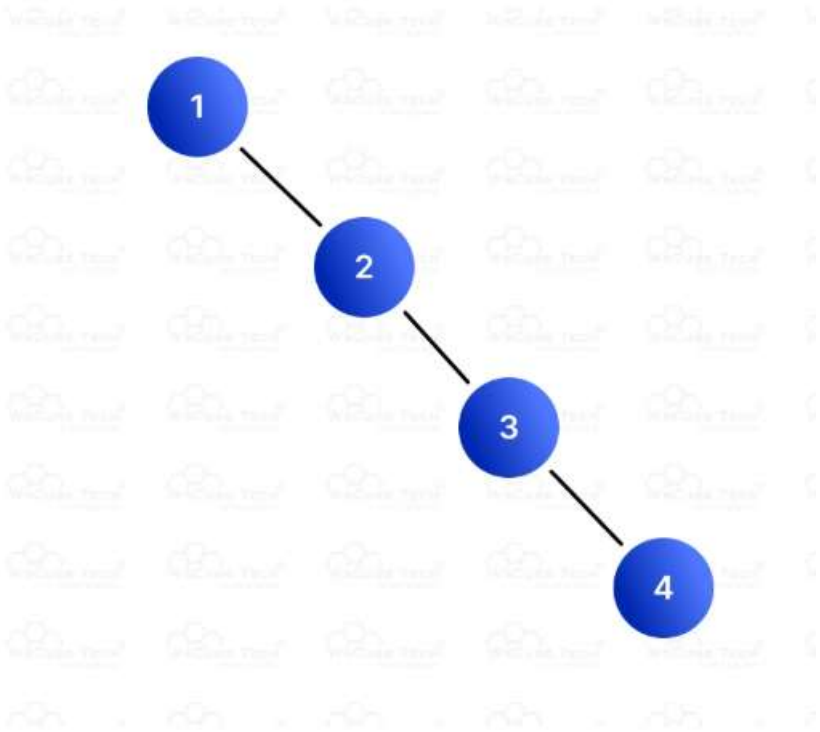
Example:



Degenerate (or Pathological) Binary Tree

A degenerate binary tree is a tree where each parent node has only one child. This makes the tree look like a **linked list**.

Example:



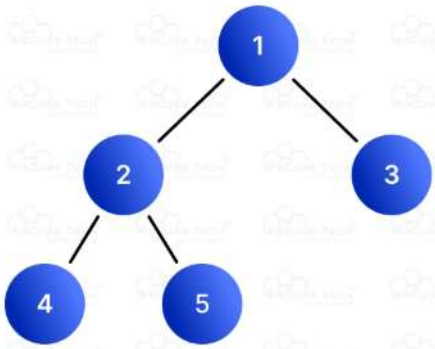
Binary Tree Traversals

Binary tree traversal refers to the process of visiting each node in the tree exactly once in a systematic way. There are four main types of binary tree traversals: in-order, pre-order, post-order, and level-order. Each traversal method visits nodes in a different order.

1. In-order Traversal (Left, Root, Right)

In in-order traversal, the nodes are recursively visited in this order: left subtree, root node, and then the right subtree.

Example:



In-order Traversal: 4, 2, 5, 1, 3

Steps:

Visit the left subtree: 4

Visit the root: 2

Visit the right subtree: 5

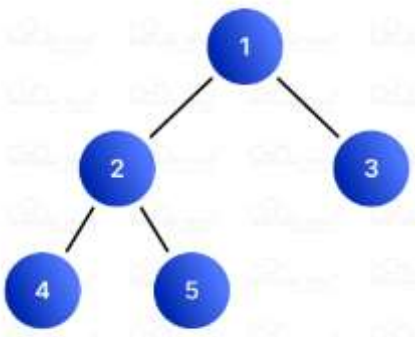
Visit the root: 1

Visit the right subtree: 3

2. Pre-order Traversal (Root, Left, Right)

In pre-order traversal, the nodes are recursively visited in this order: root node, left subtree, and then the right subtree.

Example:



Pre-order Traversal: 1, 2, 4, 5, 3

Steps:

Visit the root: 1

Visit the left subtree: 2

Visit the left subtree: 4

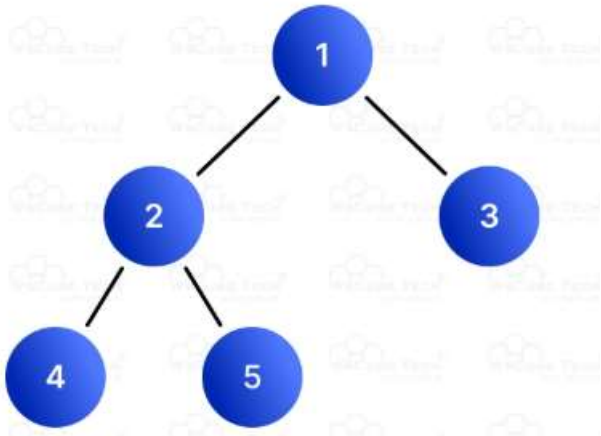
Visit the right subtree: 5

Visit the right subtree: 3

3. Post-order Traversal (Left, Right, Root)

In post-order traversal, the nodes are recursively visited in this order: left subtree, right subtree, and then the root node.

Example:



Post-order Traversal: 4, 5, 2, 3, 1

Steps:

Visit the left subtree: 4

Visit the right subtree: 5

Visit the root: 2

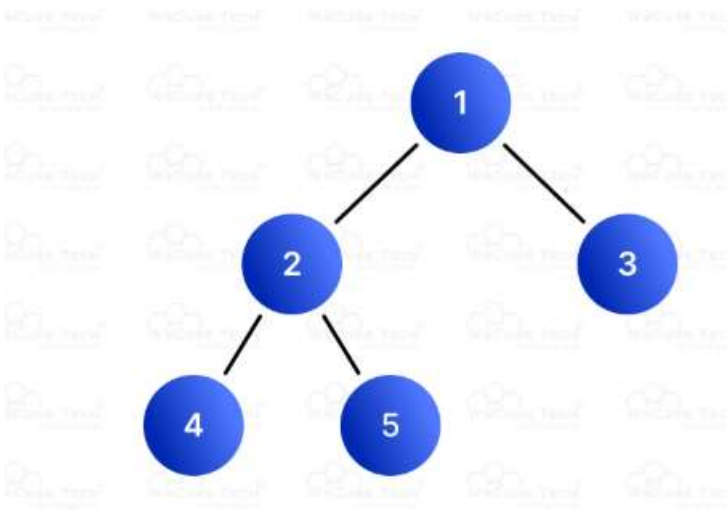
Visit the right subtree: 3

Visit the root: 1

Level-order Traversal (Breadth-First)

In level-order traversal, the nodes are visited level by level from left to right. This traversal uses a **queue data structure** to keep track of nodes.

Example:



Level-order Traversal: 1, 2, 3, 4, 5

Steps:

Visit the root: 1

Visit the nodes at the next level: 2, 3

Visit the nodes at the next level: 4, 5

Binary Tree Operations With Examples

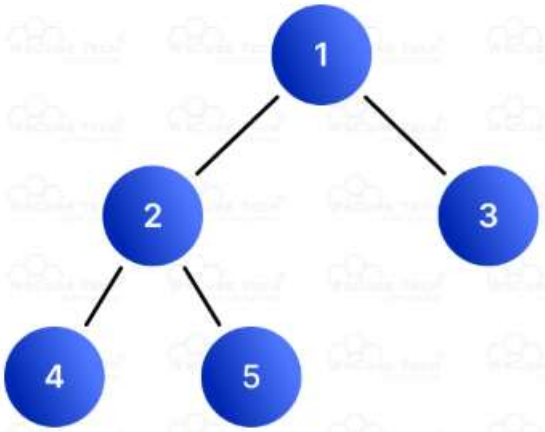
Binary trees support several fundamental operations, including insertion, deletion, searching, and traversal:

1. Insertion

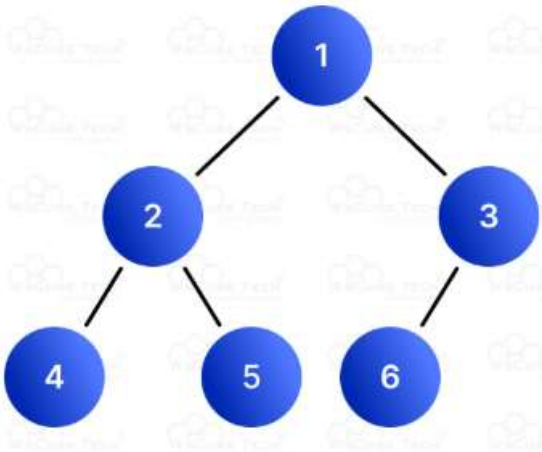
Insertion involves adding a new node to the binary tree. In a binary tree, a new node is usually inserted at the first available position in level order to maintain the completeness of the tree.

Example:

Let's insert the value 6 into the following binary tree:



After insertion, the binary tree will look like this:

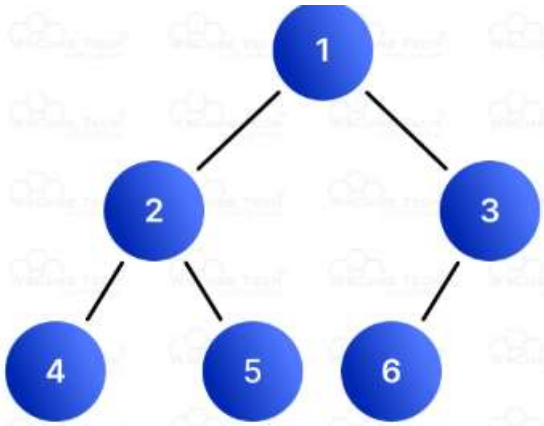


2. Deletion

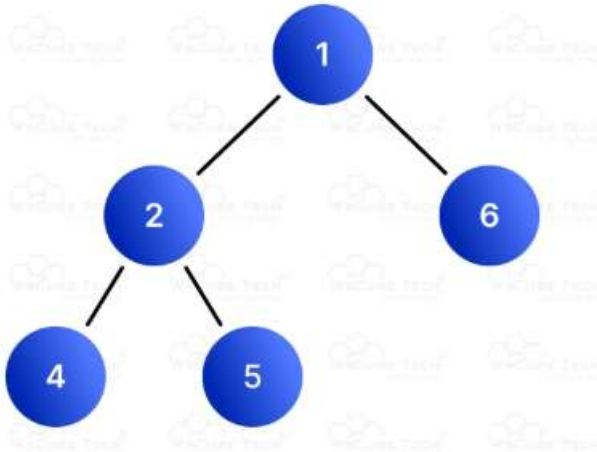
Deletion involves removing a node from the binary tree. In a binary tree, the node to be deleted is replaced by the deepest and rightmost node to maintain the tree's structure.

Example:

Let's delete the value 3 from the following binary tree:



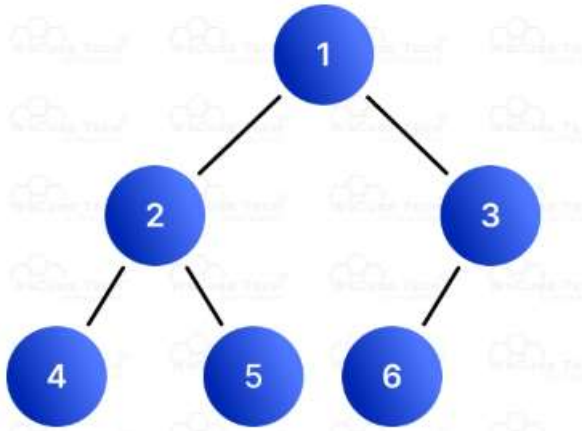
After deletion, the binary tree will look like this:



3. Search

Searching involves finding a node with a given value in the binary tree. The search operation can be implemented using any traversal method (in-order, pre-order, post-order, or level-order).

Example:



Let's search for the value 5 in the following binary tree:

Using level-order traversal:

Visit node 1

Visit node 2

Visit node 3

Visit node 4

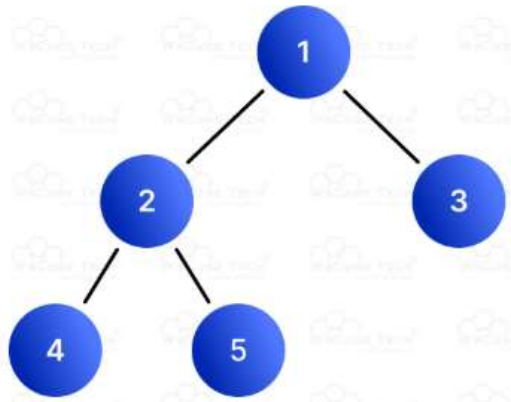
Visit node 5 (found)

4. Traversal

Traversal involves visiting all the nodes in the binary tree in a specific order. The main traversal methods are in-order, pre-order, post-order, and level-order.

Example:

Consider the following binary tree:



In-order Traversal (Left, Root, Right): 4, 2, 5, 1, 3

Pre-order Traversal (Root, Left, Right): 1, 2, 4, 5, 3

Post-order Traversal (Left, Right, Root): 4, 5, 2, 3, 1

Level-order Traversal (Breadth-First): 1, 2, 3, 4, 5

Time and Space Complexity of Binary Tree

11.Operation	12.Worst Case **Time Complexity**	13.Space Complexity
14.Insertion	15.O(n)	16.O(n)
17.Deletion	18.O(n)	19.O(n)
20.Search	21.O(n)	22.O(n)
23.Traversal (In-order, Pre-order, Post-order)	24.O(n)	25.O(h) (where h is the height of the tree)

11. Write recursive algorithms for tree traversal methods: inorder, preorder, and postorder.

Tree Traversals (Recursive Algorithms)

We assume a **binary tree node** has:

- data (value of the node)
- left (pointer to left child)
- right (pointer to right child)

---

1. Inorder Traversal (Left → Root → Right)

Algorithm:

Algorithm Inorder(node)

1. if node ≠ NULL then
2. Inorder(node.left) // Visit left subtree
3. Visit(node) // Process the root
4. Inorder(node.right) // Visit right subtree

In a **BST**, inorder traversal gives nodes in **sorted order**.

---

2. Preorder Traversal (Root → Left → Right)

Algorithm:

Algorithm Preorder(node)

1. if node ≠ NULL then
2. Visit(node) // Process the root
3. Preorder(node.left) // Visit left subtree
4. Preorder(node.right) // Visit right subtree

Used for **creating copy of tree** or **prefix expression**.

---

3. Postorder Traversal (Left → Right → Root)

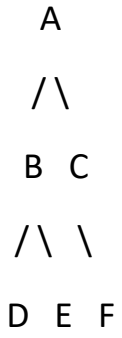
Algorithm:

Algorithm Postorder(node)

1. if node ≠ NULL then
2. Postorder(node.left) // Visit left subtree
3. Postorder(node.right) // Visit right subtree
4. Visit(node) // Process the root

Useful for **deleting/freeing tree** or **postfix expression** generation.

Example Tree



Traversals:

- **Inorder:** D B E A C F
- **Preorder:** A B D E C F
- **Postorder:** D E B F C A

12. What is a binary search tree (BST)? Explain insertion and deletion operations with examples.

A **Binary Search Tree (BST)** is a type of **binary tree** that maintains a specific order among its elements:

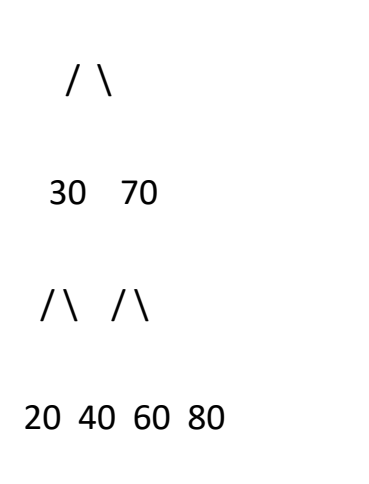
- For each node:
  - The **left subtree** contains only nodes with values **less than** the node's value.
  - The **right subtree** contains only nodes with values **greater than** the node's value.
- Both left and right subtrees are themselves **BSTs**.
- No duplicate nodes are usually allowed.

This property makes searching, insertion, and deletion very efficient.

Why BST?

- Searching in a normal array/list → **O(n)** time.
- Searching in a BST (balanced) → **O(log n)** time.
- Easy to perform sorted traversals (Inorder traversal of BST → **sorted output**).

Example BST



- Root = 50
- Left subtree = {30, 20, 40} → all < 50
- Right subtree = {70, 60, 80} → all > 50

This is a valid BST.

Operations in BST

1. Insertion in BST

Idea:

- Start from the root.
- If new key < root → move left.
- If new key > root → move right.
- Repeat until NULL spot is found → insert there.

Algorithm for Insertion

```
function insert(root, key):
    if root is NULL:
        create a new node with key
        return new node
    if key < root.data:
        root.left = insert(root.left, key)
```

```
    else if key > root.data:
        root.right = insert(root.right, key)
    return root
```

Example: Insert 25 into BST

Steps:

1. Compare with 50 → 25 < 50 → go left.
2. Compare with 30 → 25 < 30 → go left.
3. Compare with 20 → 25 > 20 → go right.
4. Right of 20 is empty → insert 25.

```
      50
     /  \
    30   70
   /\   /\
  20 40 60 80
    \
     25
```

---

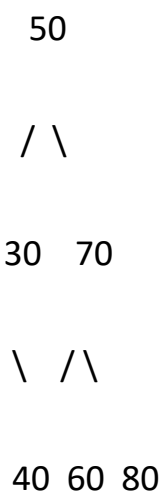
2. Deletion in BST

Deletion is more complex. There are **3 cases**:

Case 1: Node is a leaf (no child)

- Simply remove it.

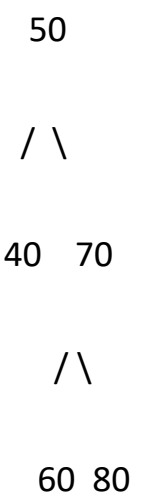
  Example: Delete 20



Case 2: Node has one child

- Replace node with its only child.

  Example: Delete 30 (has only one child 40)



Case 3: Node has two children

- Replace the node with either:

- **Inorder successor** (smallest in right subtree), or
- **Inorder predecessor** (largest in left subtree).

Example: Delete 50 (two children)

- Inorder successor of 50 is 60.
- Replace 50 with 60.
- Delete the original 60 node.

```
      60
     /  \
   30    70
  /  \     \
 20  40     80
```

C++ Program for BST (Insertion + Deletion + Traversal)

```
#include <iostream>
using namespace std;


// Node structure
class Node {
public:
    int data;
```

```
  Node* left;
  Node* right;


  Node(int value) {
    data = value;
    left = right = nullptr;
  }
};


// Insert a node in BST
Node* insert(Node* root, int key) {
  if (root == nullptr) {
    return new Node(key);
  }


  if (key < root->data)
    root->left = insert(root->left, key);
  else if (key > root->data)
    root->right = insert(root->right, key);
```

```
    return root;
}

// Find minimum value (used for deletion)
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr)
        current = current->left;
    return current;
}

// Delete a node from BST
Node* deleteNode(Node* root, int key) {
    if (root == nullptr) return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
```

```
    root->right = deleteNode(root->right, key);
  else {
    // Case 1 & 2: node with 0 or 1 child
    if (root->left == nullptr) {
        Node* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == nullptr) {
        Node* temp = root->left;
        delete root;
        return temp;
    }


    // Case 3: node with 2 children
    Node* temp = minValueNode(root->right);
    root->data = temp->data;
    root->right = deleteNode(root->right, temp->data);
  }
```

```
    return root;
}

// Inorder traversal (gives sorted order)
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

// Driver code
int main() {
    Node* root = nullptr;

    // Insertion
    root = insert(root, 50);
    insert(root, 30);
```

```
insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);



cout << "Inorder traversal: ";

inorder(root);

cout << endl;



cout << "Delete 20" << endl;

root = deleteNode(root, 20);

inorder(root);

cout << endl;



cout << "Delete 30" << endl;

root = deleteNode(root, 30);

inorder(root);

cout << endl;
```

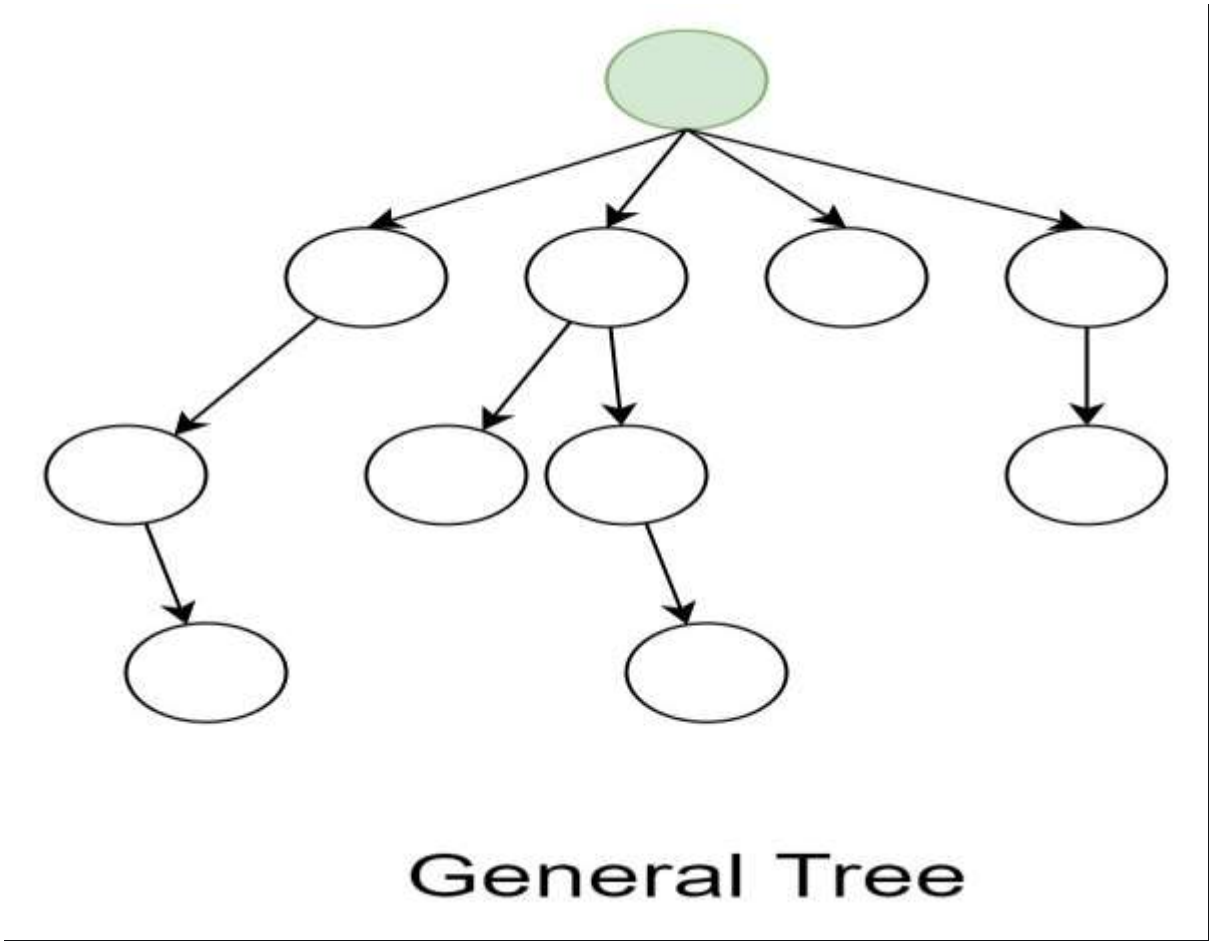
```
    cout << "Delete 50" << endl;

    root = deleteNode(root, 50);

    inorder(root);

    cout << endl;



    return 0;

}
```

Output

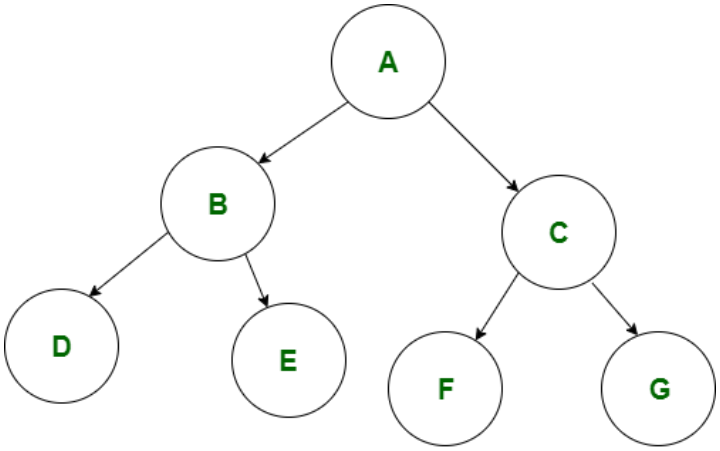
```
Inorder traversal: 20 30 40 50 60 70 80

Delete 20

30 40 50 60 70 80

Delete 30

40 50 60 70 80

Delete 50

                    40 60 70 80
```

13. Differentiate between general tree and binary tree. Explain representation of binary trees in memory.

General Tree**:** In the data structure, **General tree** is a tree in which each node can have either zero or many child nodes. It can not be empty. In general tree, there is no limitation on the degree of a node. The topmost node of a general tree is called the root node. There are many subtrees in a general tree. The subtree of a general tree is unordered because the nodes of the general tree can not be ordered according to specific criteria. In a general tree, each node has in-degree(number of parent nodes) one and maximum out-degree(number of child nodes) n.



Binary Tree**:** A binary tree is the specialized version of the General tree. A binary tree is a tree in which each node can have at most two nodes. In a binary tree, there is a limitation on the degree of a node because the nodes in a binary tree can't have more than two child node(or degree two). The topmost node of a binary tree is called root node and there are mainly two subtrees one is **left-subtree** and another is **right-subtree**. Unlike the general tree, the binary tree can be empty. Unlike the general tree, the subtree of a binary tree is ordered because the nodes of a binary tree can be ordered according to specific criteria



Binary Tree

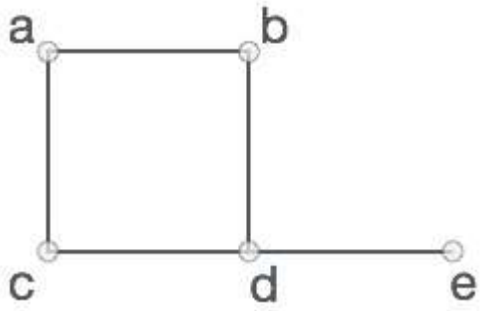
Difference between General tree and Binary tree

General tree	Binary tree
General tree is a tree in which each node can have many children or nodes.	Whereas in binary tree, each node can have at most two nodes.
The subtree of a general tree do not hold the ordered property.	While the subtree of binary tree hold the ordered property.
In data structure, a general tree can not be empty.	While it can be empty.
In general tree, a node can have at most **n(number of child nodes)** nodes.	While in binary tree, a node can have at most **2(number of child nodes)** nodes.
In general tree, there is no limitation on the degree of a node.	While in binary tree, there is limitation on the degree of a node because the nodes in a binary tree can't have more than two child node.
In general tree, there is either zero subtree or many subtree.	While in binary tree, there are mainly two subtree: **Left-subtree** and **Right-subtree**.

14. Define a graph. Explain different graph representations (adjacency matrix, adjacency list) with examples.

A graph is an abstract data type (ADT) which consists of a set of objects that are connected to each other via links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

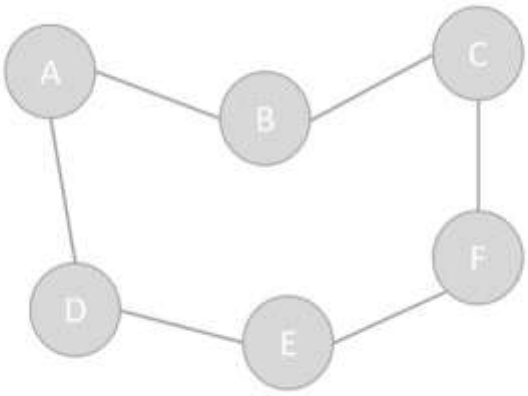
Graph Data Structure

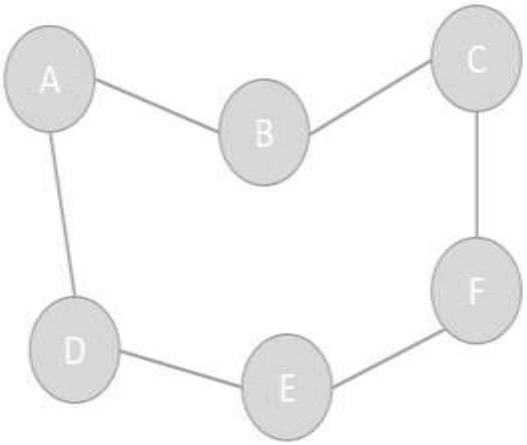
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to

represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.





Operations of Graphs

The primary operations of a graph include creating a graph with vertices and edges, and displaying the said graph. However, one of the most common and popular operation performed using graphs are Traversal, i.e. visiting every vertex of the graph in a specific order.

There are two types of traversals in Graphs −

- Depth First Search Traversal
- Breadth First Search Traversal

Depth First Search Traversal

Depth First Search is a traversal algorithm that visits all the vertices of a graph in the decreasing order of its depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed back and forth by marking unvisited adjacent nodes until all the vertices are marked.

The DFS traversal uses the stack data structure to keep track of the unvisited nodes.

Breadth First Search Traversal

Breadth First Search is a traversal algorithm that visits all the vertices of a graph present at one level of the depth before moving to the next level of depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed by visiting the adjacent vertices on the same depth level and marking them until there is no vertex left.

The DFS traversal uses the queue data structure to keep track of the unvisited nodes.

Representation of Graphs

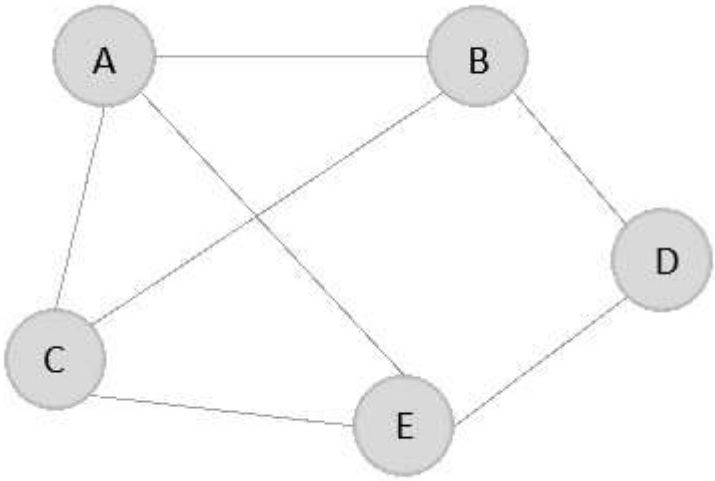
While representing graphs, we must carefully depict the elements (vertices and edges) present in the graph and the relationship between them. Pictorially, a graph is represented with a finite set of nodes and connecting links between them. However, we can also represent the graph in other most commonly used ways, like −

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

The Adjacency Matrix is a V x V matrix where the values are filled with either 0 or 1. If the link exists between Vi and Vj, it is recorded 1; otherwise, 0.

For the given graph below, let us construct an adjacency matrix −

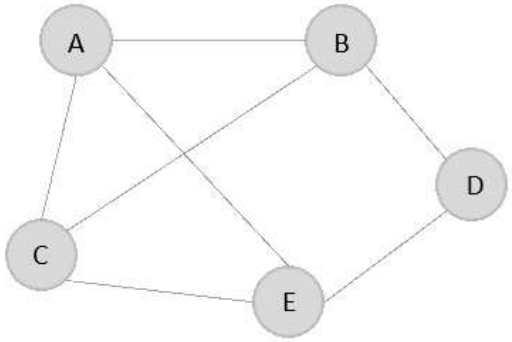


The adjacency matrix is –

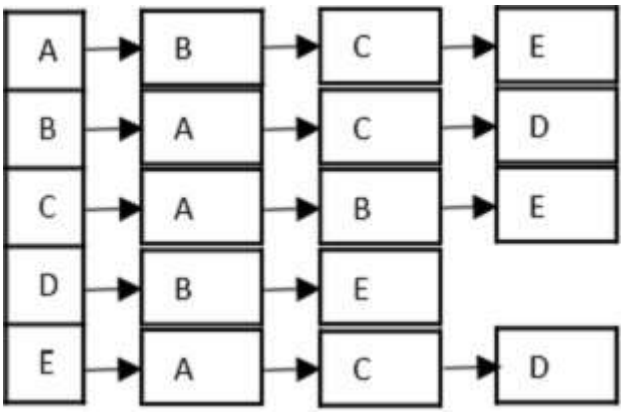
	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	0
C	1	1	0	0	1
D	0	1	0	0	1
E	1	0	1	1	0

Adjacency List

The adjacency list is a list of the vertices directly connected to the other vertices in the graph.



The adjacency list is –

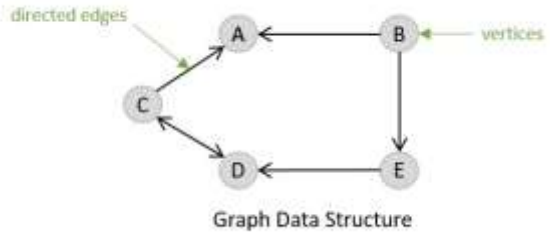


Types of graph

There are two basic types of graph –

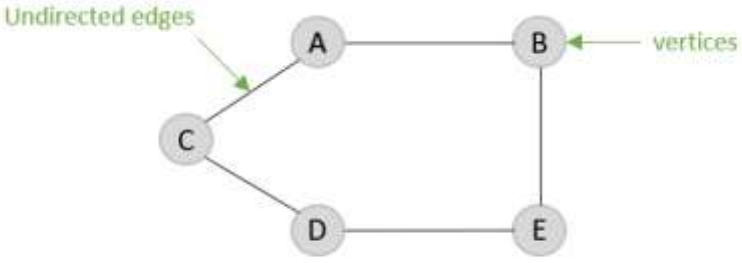
- Directed Graph
- Undirected Graph

Directed graph, as the name suggests, consists of edges that possess a direction that goes either away from a vertex or towards the vertex. Undirected graphs have edges that are not directed at all.



Graph Data Structure

Directed Graph



Graph Data Structure

Undirected Graph

15. Write algorithms for Breadth First Search (BFS) and Depth First Search (DFS).

1. Breadth First Search (BFS)

Definition

Breadth First Search (BFS) is a graph traversal algorithm that explores vertices **level by level**.

- It uses a **Queue (FIFO)**.
- First, visit the starting vertex, then all its neighbors, then their neighbors, and so on.

Algorithm for BFS

BFS(G, start_vertex):

1. Create a Queue Q

2. Mark all vertices as unvisited

3. Enqueue(start_vertex) into Q

4. Mark start_vertex as visited

5. While Q is not empty:

a. Dequeue a vertex 'u' from Q

b. Process 'u' (print/store it)

c. For each neighbor 'v' of 'u':

If 'v' is not visited:

Mark 'v' as visited

Enqueue 'v' into Q

Example of BFS

Graph edges:

A - B, A - C, B - D, C - E

Steps (Start BFS from **A**):

- Visit A → Queue = [A]
- Dequeue A → Visit neighbors B, C → Queue = [B, C]
- Dequeue B → Visit neighbor D → Queue = [C, D]
- Dequeue C → Visit neighbor E → Queue = [D, E]
- Dequeue D → No new neighbors
- Dequeue E → No new neighbors

BFS Order = **A → B → C → D → E**

2. Depth First Search (DFS)

Definition

Depth First Search (DFS) is a graph traversal algorithm that explores as **deep as possible** along each branch before backtracking.

- It uses **Stack (LIFO)** or **Recursion**.
- First visit a vertex, then recursively visit its unvisited neighbors.

Algorithm for DFS (Recursive)

DFS(G, vertex v):

1. Mark v as visited
2. Process v (print/store it)
3. For each neighbor 'u' of v:
   If 'u' is not visited:
      DFS(G, u)

Algorithm for DFS (Iterative using Stack)

DFS(G, start_vertex):

1. Create an empty Stack S
2. Mark all vertices as unvisited
3. Push(start_vertex) into S

4. While S is not empty:
   a. Pop vertex 'u' from S
   b. If 'u' is not visited:
      i. Mark 'u' as visited
      ii. Process 'u'
      iii. Push all unvisited neighbors of 'u' into S

Example of DFS

Graph edges:

A - B, A - C, B - D, C - E

Steps (Start DFS from **A**):

- Visit A → Go to B
- Visit B → Go to D
- Visit D → No new neighbor → Backtrack to B → Backtrack to A
- From A, go to C
- Visit C → Go to E
- Visit E → No new neighbor → Backtrack

DFS Order = **A → B → D → C → E**