# III Semester
# Course 6: Data Structures using C
# UNIT-I
# Introduction to Data structures

## Introduction to Data Structures in C

### *1. What is a Data Structure?*

A **data structure** is a way of organizing, managing, and storing data efficiently so that it can be accessed and modified easily. In C, data structures help in implementing algorithms efficiently.

### *2. Types of Data Structures*

Data structures in C can be broadly classified into two types:

1. **Linear Data Structures** – Elements are arranged sequentially.
    - **Arrays** – Fixed-size, contiguous memory storage.
    - **Linked Lists** – Dynamic, node-based storage.
    - **Stacks** – LIFO (Last In, First Out) structure.
    - **Queues** – FIFO (First In, First Out) structure.
2. **Non-Linear Data Structures** – Elements are connected in a hierarchical manner.
    - **Trees** – Hierarchical data structure (e.g., Binary Trees, BST, Heaps).
    - **Graphs** – Nodes connected by edges, used in networking and pathfinding.

---

## Basic Data Structures in C

### *1. Arrays*

- Collection of elements of the same data type stored in contiguous memory.
- Example:
- `int arr[5] = {1, 2, 3, 4, 5};`
- Supports indexing but has a fixed size.

### *2. Linked List*

- Collection of nodes where each node contains data and a pointer to the next node.
- Example (Singly Linked List Node):
- `struct Node {`
- `    int data;`
- `    struct Node* next;`
- `};`
- More flexible than arrays (dynamic size).

### *3. Stack*

- **LIFO (Last In, First Out)**
- Operations: `push()`, `pop()`, `peek()`
- Example using array:
- `#define MAX 100`
- `int stack[MAX];`
- `int top = -1;`
- Used in recursion, undo operations, etc.

### *4. Queue*

- **FIFO (First In, First Out)**
- Operations: `enqueue()`, `dequeue()`
- Example using array:
- `#define SIZE 10`
- `int queue[SIZE];`
- `int front = 0, rear = -1;`
- Used in scheduling, buffering, etc.

### *5. Trees*

- Hierarchical structure consisting of nodes.
- Example: **Binary Tree**
- `struct Node {`
- `    int data;`
- `    struct Node* left;`
- `    struct Node* right;`
- `};`
- Used in searching, decision-making (e.g., BST, AVL Trees).

### *6. Graphs*

- Nodes (vertices) connected by edges.
- Example:
- `struct Graph {`
- `    int vertices;`
- `    int **adjMatrix;`
- `};`
- Used in networking, pathfinding algorithms.

---

# Definition of Data Structures

A **data structure** is a specialized way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently. It defines the relationship between data, operations on data, and how data is stored.

### *Key Features of Data Structures*

1. **Efficient Data Access** – Organizes data for fast retrieval and updates.
2. **Memory Utilization** – Helps in optimal memory usage.
3. **Operations** – Supports operations like insertion, deletion, searching, and sorting.
4. **Reusability** – Can be used across multiple applications.

### *Types of Data Structures*

1. **Linear Data Structures** – Data elements are arranged sequentially (e.g., Arrays, Linked Lists, Stacks, Queues).
2. **Non-Linear Data Structures** – Data elements are arranged hierarchically (e.g., Trees, Graphs).

## Types of Data Structures in C

In C, data structures are classified into two major categories:

### 1. Linear Data Structures

In **linear data structures**, elements are arranged sequentially, and each element is connected to its previous and next elements.

#### *a) Arrays*

- A collection of elements of the same data type stored in contiguous memory locations.
- **Example:**
- `int arr[5] = {1, 2, 3, 4, 5};`
- **Operations:** Insertion, Deletion, Traversal, Searching, Sorting
- **Use Cases:** Used in matrices, tables, and fixed-size storage requirements.

#### *b) Linked Lists*

- A collection of nodes, where each node contains data and a pointer to the next node.
- **Example (Node Structure):**
- `struct Node {`
- `    int data;`
- `    struct Node* next;`
- `};`
- **Types:** Singly Linked List, Doubly Linked List, Circular Linked List
- **Use Cases:** Dynamic memory allocation, efficient insertion/deletion

### *c) Stack*

- A **LIFO (Last In, First Out)** structure.
- **Example using an array:**
- `#define MAX 100`
- `int stack[MAX];`
- `int top = -1;`
- **Operations:** `push(), pop(), peek()`
- **Use Cases:** Function calls (recursion), Undo operations, Expression evaluation

### *d) Queue*

- A **FIFO (First In, First Out)** structure.
- **Example using an array:**
- `#define SIZE 10`
- `int queue[SIZE];`
- `int front = 0, rear = -1;`
- **Types:** Simple Queue, Circular Queue, Priority Queue, Double-ended Queue (Deque)
- **Use Cases:** Scheduling, Buffer management, Print queue

---

## 2. Non-Linear Data Structures

In **non-linear data structures**, elements are not arranged sequentially.

### *a) Trees*

- A hierarchical structure consisting of **nodes**.
- **Example (Binary Tree Node):**
- `struct Node {`
- `    int data;`
- `    struct Node* left;`
- `    struct Node* right;`
- `};`
- **Types:** Binary Tree, Binary Search Tree (BST), AVL Tree, Heap
- **Use Cases:** Database indexing, AI decision trees, File system organization

### *b) Graphs*

- A collection of **nodes (vertices)** connected by **edges**.
- **Example (Graph Representation using Adjacency Matrix):**
- `struct Graph {`
- `    int vertices;`
- `    int **adjMatrix;`
- `};`
- **Types:** Directed Graph, Undirected Graph, Weighted Graph
- **Use Cases:** Networking (Internet, Social Media), Pathfinding (Google Maps), AI

---

### 3. Other Data Structures

These are additional data structures used in advanced programming.

#### *a) Hash Tables*

- Stores data in **key-value pairs** for fast retrieval.
- **Example (Simple Hash Table in C using an array):**
- `#define TABLE_SIZE 10`
- `int hashTable[TABLE_SIZE];`
- **Use Cases:** Database indexing, Caching, Cryptography

#### *b) Heap*

- A **complete binary tree** used for priority-based operations.
- **Types:** Min-Heap, Max-Heap
- **Use Cases:** Priority Queue, Heap Sort, Memory Management

---

### Conclusion

C provides powerful data structures that help in **efficient data organization and manipulation**. The choice of data structure depends on the problem requirements.

## Abstract Data Types (ADT)

### What is an Abstract Data Type (ADT)?

An **Abstract Data Type (ADT)** is a **mathematical model** of a data structure that specifies:

1. **What operations** can be performed on the data.
2. **How the data behaves** (without specifying implementation details).

ADTs **define the behavior** of data structures without describing how they are implemented in memory.

---

### Examples of ADTs and Their Operations

| ADT | Operations |
|---|---|
| **List** | Insert, Delete, Search, Traverse |

| ADT | Operations |
|---|---|
| **Stack** | Push, Pop, Peek, isEmpty |
| **Queue** | Enqueue, Dequeue, Peek, isEmpty |
| **Deque** | Insert Front, Insert Rear, Delete Front, Delete Rear |
| **Priority Queue** | Insert, Delete (based on priority) |
| **Set** | Insert, Delete, Union, Intersection, Difference |
| **Map (Dictionary)** | Insert (key-value pair), Search (by key), Delete |

---

## Common ADTs and Their Implementations in C

### *1. List ADT*

- A **list** is a collection of elements that can be accessed sequentially.
- **Operations:** Insert, Delete, Search, Traverse
- **Implementation:** Arrays or Linked Lists

### *2. Stack ADT (LIFO - Last In, First Out)*

- **Operations:**
  - `push(element)`: Insert an element at the top.
  - `pop()`: Remove the top element.
  - `peek()`: Retrieve the top element without removing it.
  - `isEmpty()`: Check if the stack is empty.
- **Implementation in C (Using Arrays)**:
- `#define MAX 100`
- `int stack[MAX], top = -1;`
- 
- `void push(int value) {`
- `    if (top == MAX - 1) {`
- `        printf("Stack Overflow\n");`
- `        return;`
- `    }`
- `    stack[++top] = value;`
- `}`
- **Use Cases:** Function calls (Recursion), Undo operations, Expression evaluation

### *3. Queue ADT (FIFO - First In, First Out)*

- **Operations:**

    - `enqueue(element)`: Add an element at the rear.
    - `dequeue()`: Remove an element from the front.
    - `peek()`: View the front element without removing it.
    - `isEmpty()`: Check if the queue is empty.
- **Implementation in C (Using Arrays)**:
- `#define SIZE 10`
- `int queue[SIZE], front = 0, rear = -1;`
- 
- `void enqueue(int value) {`
- `    if (rear == SIZE - 1) {`
- `        printf("Queue is Full\n");`
- `        return;`
- `    }`
- `    queue[++rear] = value;`
- `}`
- **Use Cases:** Scheduling, Print Queue, CPU Task Management

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

- **Operations:** Insert and delete from both front and rear.
- **Types:** Input-restricted Deque, Output-restricted Deque
- **Use Cases:** Palindrome checking, Sliding Window Problems

### *5. Priority Queue ADT*

- **Operations:** Insert elements based on priority, Remove highest/lowest priority element.
- **Use Cases:** CPU Scheduling, Dijkstra's Algorithm

### *6. Set ADT*

- **Operations:** Insert, Delete, Union, Intersection, Difference
- **Use Cases:** Database operations, Membership checking

### *7. Map (Dictionary) ADT*

- **Operations:** Store key-value pairs, Search by key, Delete by key
- **Use Cases:** Hash Tables, Symbol Tables

---

## Key Points About ADTs

✅ ADTs focus on **what operations** are performed, **not how** they are implemented.
✅ Implementation can be done using **arrays, linked lists, or other structures**.
✅ ADTs help in **modular programming** by providing reusable interfaces.

# Difference Between Abstract Data Types (ADT), Data Types, and Data Structures in C

| Aspect | Abstract Data Type (ADT) | Data Type | Data Structure |
|---|---|---|---|
| **Definition** | A conceptual model that defines a data structure and its operations without specifying implementation. | A classification of data that defines the type of values a variable can hold. | A way of organizing and storing data in memory. |
| **Focus** | Focuses on **what operations** can be performed rather than how they are implemented. | Focuses on defining **data values** and their valid operations. | Focuses on **how data is stored and managed** efficiently. |
| **Implementation** | Implemented using **data structures** (e.g., stack using an array or linked list). | Built into the C language (primitive) or user-defined (derived). | Implemented using memory structures such as arrays, linked lists, trees, etc. |
| **Examples** | Stack, Queue, List, Set, Priority Queue, Map | `int`, `float`, `char`, `double`, `struct`, `enum` | Arrays, Linked Lists, Stacks, Queues, Trees, Graphs |
| **Usage** | Provides **abstraction** for data handling and algorithm design. | Defines the **nature** of data being stored and processed. | Optimizes **data access, manipulation, and storage**. |

---

## Detailed Explanation

### *1. Data Type*

- A **data type** specifies the kind of data a variable can store.
- **Built-in Data Types (Primitive)**:
  - `int`, `float`, `char`, `double`
- **User-defined Data Types**:
  - `struct`, `union`, `enum`, `typedef`

### Example:

```
int a = 10;   // 'int' is a primitive data type
struct Student {
    char name[50];
    int age;
};  // 'struct' is a user-defined data type
```

---

### *2. Abstract Data Type (ADT)*

- An **abstract model** of a data structure that defines **operations** but does not specify how they are implemented.
- ADTs can be implemented using **arrays, linked lists, or other structures**.

**Example (Stack ADT Implementation Using an Array in C):**

```
#define MAX 100
int stack[MAX], top = -1;

void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = value;
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}
```

👉 Here, **"push" and "pop"** define stack operations (ADT), while the **array** is the underlying data structure.

---

### *3. Data Structure*

- A **concrete implementation** that defines how data is organized and stored in memory.
- Helps in **efficient data access and manipulation**.

**Example (Linked List Implementation in C):**

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

struct Node* head = NULL;
```

👉 A **linked list** is a **data structure** that can be used to implement **ADT like a stack or queue**.

---

### Conclusion

✅ **Data Types** define the nature of values.
✅ **Data Structures** define how data is stored and accessed.
✅ **Abstract Data Types (ADTs)** define the behavior of a data structure without focusing on implementation.

## Arrays in Data Structures

### What is an Array?

An **array** is a collection of elements of the **same data type**, stored in **contiguous memory locations**. It allows **random access** to elements using an index.

### Key Features of Arrays

✅ **Fixed Size** – The size is declared at the time of creation.
✅ **Same Data Type** – All elements must be of the same type (e.g., `int`, `char`, `float`).
✅ **Indexed Access** – Elements are accessed using an index (starting from 0).
✅ **Efficient Traversal** – Elements can be accessed quickly using loops.

---

### Types of Arrays in C

#### 1. One-Dimensional (1D) Array

A **1D array** stores data in a **single row**.

*Declaration & Initialization*

```
int arr[5] = {10, 20, 30, 40, 50}; // Declaring and initializing
```

*Accessing Elements*

```
printf("%d", arr[2]); // Output: 30 (Index starts from 0)
```

*Traversing a 1D Array*

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

---

## 2. Two-Dimensional (2D) Array

A **2D array** stores data in **rows and columns** (matrix format).

*Declaration & Initialization*

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

*Accessing Elements*

```
printf("%d", matrix[1][2]); // Output: 6 (Row 1, Column 2)
```

*Traversing a 2D Array*

```
#include <stdio.h>
int main() {
    int matrix[2][2] = {{1, 2}, {3, 4}};

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

---

## 3. Multi-Dimensional Arrays

C allows arrays of **more than two dimensions**, but they are less commonly used.

```
int arr[2][2][2] = {{{1,2}, {3,4}}, {{5,6}, {7,8}}};
```

---

## Operations on Arrays

| Operation | Description |
|---|---|
| **Traversal** | Visiting each element one by one |
| **Insertion** | Adding an element at a specific index |
| **Deletion** | Removing an element from an index |
| **Searching** | Finding an element (Linear/Binary Search) |
| **Sorting** | Arranging elements in order (Bubble, Quick, Merge Sort) |

---

### Advantages of Arrays

✅ **Fast Access** – Constant time complexity **O(1)** for element access.
✅ **Memory Efficiency** – Uses contiguous memory.
✅ **Easy Sorting & Searching** – Works well with algorithms like **Binary Search** and **Sorting**.

### Disadvantages of Arrays

❌ **Fixed Size** – Cannot grow dynamically (Use `malloc()` for dynamic allocation).
❌ **Insertion/Deletion Overhead** – Requires shifting elements.

---

### Example: Searching an Element in an Array (Linear Search)

```
#include <stdio.h>

int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key)
            return i; // Return index if found
    }
    return -1; // Not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = linearSearch(arr, size, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

## Concept of Arrays

### What is an Array?

An **array** is a collection of elements of the **same data type**, stored in **contiguous memory locations**. It allows **random access** to elements using an index.

### Key Characteristics of Arrays

✅ **Fixed Size** – The size is declared at the time of creation.
✅ **Same Data Type** – All elements must be of the same type (e.g., `int`, `char`, `float`).
✅ **Indexed Access** – Elements are accessed using an index (starting from 0).
✅ **Efficient Traversal** – Elements can be accessed quickly using loops.
✅ **Contiguous Memory Allocation** – Stored in continuous memory blocks.

---

## Declaration and Initialization of Arrays

### 1. One-Dimensional (1D) Array

A **1D array** is a linear collection of elements.

*Declaration & Initialization*

```
int arr[5] = {10, 20, 30, 40, 50}; // Declaring and initializing
```

*Accessing Elements*

```
printf("%d", arr[2]); // Output: 30 (Index starts from 0)
```

*Traversing a 1D Array*

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

---

### 2. Two-Dimensional (2D) Array

A **2D array** is used to store data in rows and columns (like a matrix).

*Declaration & Initialization*

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

*Accessing Elements*

```
printf("%d", matrix[1][2]); // Output: 6 (Row 1, Column 2)
```

*Traversing a 2D Array*

```
#include <stdio.h>
int main() {
    int matrix[2][2] = {{1, 2}, {3, 4}};
```

```
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

---

### 3. Multi-Dimensional Arrays

C allows arrays of **more than two dimensions**, but they are less commonly used.

```
int arr[2][2][2] = {{{1,2}, {3,4}}, {{5,6}, {7,8}}};
```

---

## Operations on Arrays

| Operation | Description |
|---|---|
| **Traversal** | Visiting each element one by one |
| **Insertion** | Adding an element at a specific index |
| **Deletion** | Removing an element from an index |
| **Searching** | Finding an element (Linear/Binary Search) |
| **Sorting** | Arranging elements in order (Bubble, Quick, Merge Sort) |

---

## Advantages of Arrays

✅ **Fast Access** – Constant time complexity **O(1)** for element access.
✅ **Memory Efficiency** – Uses contiguous memory.
✅ **Easy Sorting & Searching** – Works well with algorithms like **Binary Search** and **Sorting**.

## Disadvantages of Arrays

❌ **Fixed Size** – Cannot grow dynamically (Use `malloc()` for dynamic allocation).
❌ **Insertion/Deletion Overhead** – Requires shifting elements.

---

### Example: Searching an Element in an Array (Linear Search)

```
#include <stdio.h>

int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key)
            return i; // Return index if found
    }
    return -1; // Not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = linearSearch(arr, size, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

# Single Dimensional Array in C

## What is a Single-Dimensional Array?

A **single-dimensional array** (1D array) is a collection of elements of the **same data type** stored in **contiguous memory locations** and accessed using an **index**.

### Key Features of 1D Arrays

✅ **Stores elements of the same type**
✅ **Elements are accessed using an index (starting from 0)**
✅ **Fixed size** (declared at the time of creation)
✅ **Efficient for data storage and retrieval**

---

## Declaration and Initialization of a 1D Array

### 1. Declaration of a 1D Array

```
data_type array_name[size];
```

Example:

```
int arr[5];  // Declares an array of 5 integers
```

### 2. Initialization of a 1D Array

```
int arr[5] = {10, 20, 30, 40, 50}; // Declare and initialize
```

👉 If we don't initialize an array, it will contain **garbage values**.

### 3. Accessing Elements in a 1D Array

```
printf("%d", arr[2]); // Output: 30 (Index starts from 0)
```

---

## Example Program: Traversing a 1D Array

```
#include <stdio.h>
int main() {
    int arr[] = {1, 2, 3, 4, 5}; // Declare and initialize
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate size

    // Traversing the array
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

**Output:**

```
1 2 3 4 5
```

---

## Operations on a 1D Array

| Operation | Description | Time Complexity |
|---|---|---|
| **Traversal** | Accessing each element one by one | O(n) |
| **Insertion** | Adding an element at a specific position | O(n) |
| **Deletion** | Removing an element from a specific position | O(n) |
| **Searching** | Finding an element (Linear/Binary Search) | O(n) / O(log n) |
| **Sorting** | Arranging elements in ascending/descending order | O(n log n) |

---

## Example 1: Insert an Element in a 1D Array

```
#include <stdio.h>
int main() {
    int arr[6] = {1, 2, 4, 5, 6}; // 6th element is uninitialized
    int size = 5;  // Current size
    int pos = 2, value = 3; // Insert 3 at index 2
```

```
    // Shifting elements to the right
    for (int i = size; i > pos; i--) {
        arr[i] = arr[i - 1];
    }

    arr[pos] = value; // Insert value
    size++;

    // Print updated array
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

**Output:**

```
1 2 3 4 5 6
```

---

## Example 2: Searching an Element (Linear Search)

```
#include <stdio.h>
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key)
            return i; // Return index if found
    }
    return -1; // Not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = linearSearch(arr, size, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

**Output:**

```
Element found at index 2
```

---

### Advantages of a 1D Array

✅ **Fast access (O(1) time complexity) for element retrieval**
✅ **Efficient for storing a collection of similar elements**
✅ **Easy to implement searching and sorting algorithms**

### Disadvantages of a 1D Array

❌ **Fixed size** – Cannot dynamically grow or shrink (Use dynamic memory allocation for flexibility).
❌ **Insertion/Deletion Complexity** – Requires shifting elements (O(n) complexity).

Would you like **sorting algorithms** or **more advanced array operations**? 😊

# Two-Dimensional Array in C

## What is a Two-Dimensional Array?

A **two-dimensional (2D) array** is an array of arrays, where data is stored in a **row-column format**, similar to a table or matrix. Each element is accessed using two indices:

- The **first index** represents the row.
- The **second index** represents the column.

### Key Features of 2D Arrays

✅ **Stores elements of the same type**
✅ **Accessed using two indices** `[row][column]`
✅ **Fixed size** (declared at the time of creation)
✅ **Useful for matrices, tables, and grid-based data**

---

## Declaration and Initialization of a 2D Array

### 1. Declaration of a 2D Array

```
data_type array_name[rows][columns];
```

Example:

```
int matrix[3][3];  // Declares a 3x3 integer matrix
```

### 2. Initialization of a 2D Array

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
```

```
    {7, 8, 9}
};
```

👉 You can also initialize without specifying row size:

```
int matrix[][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

### 3. Accessing Elements in a 2D Array

```
printf("%d", matrix[1][2]); // Output: 6 (Row index = 1, Column index = 2)
```

---

## Example Program: Traversing a 2D Array

```
#include <stdio.h>

int main() {
    int matrix[2][2] = {{1, 2}, {3, 4}};

    // Traversing the 2D array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n"); // New line for matrix format
    }

    return 0;
}
```

**Output:**

```
1 2
3 4
```

---

## Operations on a 2D Array

| Operation | Description | Time Complexity |
|---|---|---|
| **Traversal** | Accessing each element one by one | O(rows × cols) |
| **Insertion** | Adding an element at a specific position | O(1) |
| **Deletion** | Removing an element from a specific position | O(1) |
| **Searching** | Finding an element (Linear Search) | O(rows × cols) |

---

## Example 1: Input and Display a 2D Array

```
#include <stdio.h>

int main() {
    int matrix[3][3];

    // Taking input for a 3x3 matrix
    printf("Enter 9 elements:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    // Printing the matrix
    printf("The Matrix is:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

**Input:**

```
1 2 3
4 5 6
7 8 9
```

**Output:**

```
The Matrix is:
1 2 3
4 5 6
7 8 9
```

---

## Example 2: Searching an Element in a 2D Array

```
#include <stdio.h>

int searchMatrix(int matrix[3][3], int key) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (matrix[i][j] == key) {
                printf("Element found at row %d, column %d\n", i, j);
                return 1;
            }
        }
    }
    return 0;
}
```

```
int main() {
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int key = 5;

    if (!searchMatrix(matrix, key)) {
        printf("Element not found\n");
    }

    return 0;
}
```

**Output:**

```
Element found at row 1, column 1
```

---

### Advantages of a 2D Array

✅ **Best for matrix-based operations**
✅ **Organized storage for tabular data**
✅ **Direct access to elements using index `[row][column]`**

### Disadvantages of a 2D Array

❌ **Fixed size** – Cannot grow dynamically (Use dynamic memory allocation for flexibility).
❌ **Complexity** – Operations like searching require `O(rows × cols)` time.

Would you like an **example of matrix addition, subtraction, or multiplication**? 😊

## Operations on Arrays with Algorithms

Arrays support multiple operations, including **traversing, searching, inserting, and deleting elements**. Below are the explanations, algorithms, and example implementations in **C** for each operation.

---

### 1. Traversing an Array

👉 **Definition:** Traversing means accessing each element of the array one by one.
👉 **Time Complexity: O(n)** (where `n` is the number of elements in the array).

### Algorithm for Traversal

```
1. Start
2. Initialize an array arr[] of size n
3. Loop from i = 0 to n-1
     a. Print arr[i]
4. End
```

### C Program for Traversing

```
#include <stdio.h>

void traverse(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Array Elements: ");
    traverse(arr, size);

    return 0;
}
```

**Output:**

```
Array Elements: 10 20 30 40 50
```

---

## 2. Searching in an Array

👉 **Definition:** Searching means finding the index of a given element in an array.

👉 **Types of Searching Algorithms:**

- **Linear Search (O(n))** – Used for unsorted arrays.
- **Binary Search (O(log n))** – Used for sorted arrays.

### A. Linear Search Algorithm

```
1. Start
2. Initialize an array arr[] of size n
3. Loop from i = 0 to n-1
     a. If arr[i] == key, return i (index)
4. If key is not found, return -1
5. End
```

### C Program for Linear Search

```
#include <stdio.h>

int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
```

```
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = linearSearch(arr, size, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

**Output:**

```
Element found at index 2
```

## B. Binary Search Algorithm (for Sorted Arrays)

```
1. Start
2. Initialize low = 0, high = n-1
3. Repeat while low ≤ high:
     a. mid = (low + high) / 2
     b. If arr[mid] == key, return mid
     c. If arr[mid] < key, set low = mid + 1
     d. Else, set high = mid - 1
4. If key is not found, return -1
5. End
```

## C Program for Binary Search

```
#include <stdio.h>

int binarySearch(int arr[], int size, int key) {
    int low = 0, high = size - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50}; // Sorted array
```

```
    int key = 40;
    int size = sizeof(arr) / sizeof(arr[0]);

    int result = binarySearch(arr, size, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

**Output:**

```
Element found at index 3
```

---

## 3. Insertion in an Array

👉 **Definition:** Inserting an element at a specific position in an array.
👉 **Time Complexity: O(n)** (since elements after the position must be shifted).

### Algorithm for Insertion

```
1. Start
2. Check if the array is full
3. Move elements from the last index to the desired position
4. Insert the new element at the given position
5. Increase the array size
6. End
```

### C Program for Insertion

```
#include <stdio.h>

void insertElement(int arr[], int *size, int pos, int value) {
    for (int i = *size; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
    (*size)++;
}

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int size = 5, pos = 2, value = 25;

    printf("Before Insertion: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    insertElement(arr, &size, pos, value);

    printf("\nAfter Insertion: ");
    for (int i = 0; i < size; i++) {
```

```
        printf("%d ", arr[i]);
    }

    return 0;
}
```

**Output:**

```
Before Insertion: 10 20 30 40 50
After Insertion: 10 20 25 30 40 50
```

---

## 4. Deletion in an Array

👉 **Definition:** Deleting an element at a specific position in an array.
👉 **Time Complexity: O(n)** (since elements after the deleted one must be shifted).

### Algorithm for Deletion

```
1. Start
2. Find the index of the element to delete
3. Shift all elements after the index to the left
4. Decrease the array size
5. End
```

### C Program for Deletion

```
#include <stdio.h>

void deleteElement(int arr[], int *size, int pos) {
    for (int i = pos; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*size)--;
}

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int size = 5, pos = 2;

    printf("Before Deletion: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    deleteElement(arr, &size, pos);

    printf("\nAfter Deletion: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

**Output:**

```
Before Deletion: 10 20 30 40 50
After Deletion: 10 20 40 50
```

---

## Summary of Time Complexities

| Operation | Best Case | Worst Case |
|---|---|---|
| **Traversal** | O(n) | O(n) |
| **Searching (Linear)** | O(1) | O(n) |
| **Searching (Binary)** | O(1) | O(log n) |
| **Insertion** | O(1) | O(n) |
| **Deletion** | O(1) | O(n) |

Would you like an **example of sorting (Bubble, Selection, Quick Sort)?** 😊

# UNIT-II

## Linked List in Data Structures

A **Linked List** is a linear data structure where elements are stored in **nodes**, and each node contains a **data field** and a **pointer (or reference) to the next node** in the sequence. Unlike arrays, linked lists provide **dynamic memory allocation**, meaning their size can change at runtime.

---

### Types of Linked Lists

1. **Singly Linked List (SLL)** – Each node points to the next node.
2. **Doubly Linked List (DLL)** – Each node has a pointer to both **next** and **previous** nodes.
3. **Circular Linked List (CLL)** – The last node points back to the first node.

---

### 1. Structure of a Singly Linked List

A **node** in a linked list is defined as:

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

Each node contains:

- `data` (value stored in the node)
- `next` (pointer to the next node)

#### Example of a Singly Linked List

```
Head → [10 | *] → [20 | *] → [30 | NULL]
```

---

### 2. Basic Operations on a Singly Linked List

#### A. Insertion in a Linked List

1. **At the beginning**
2. **At the end**
3. **At a specific position**

#### Algorithm for Insertion at the Beginning

```
1. Create a new node
```

```
2. Set newNode->next = head
3. Update head = newNode
```

### C Program for Inserting a Node at the Beginning

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

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

// Function to insert at the beginning
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = *head;
    *head = newNode;
}

// Function to print the list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

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

    insertAtBeginning(&head, 30);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 10);

    printf("Linked List: ");
    printList(head);

    return 0;
}
```

**Output:**

```
Linked List: 10 -> 20 -> 30 -> NULL
```

---

## B. Deletion from a Linked List

1. **From the beginning**
2. **From the end**
3. **A specific node**

## Algorithm for Deletion from the Beginning

1. Check if the list is empty
2. Store the head node in a temp variable
3. Move head to the next node
4. Free the temp node

## C Program for Deleting the First Node

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

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

// Function to delete the first node
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to print the list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
```

```
    struct Node* head = NULL;

    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    printf("Linked List before deletion: ");
    printList(head);

    deleteFromBeginning(&head);

    printf("Linked List after deletion: ");
    printList(head);

    return 0;
}
```

**Output:**

```
Linked List before deletion: 10 -> 20 -> 30 -> NULL
Linked List after deletion: 20 -> 30 -> NULL
```

---

## C. Searching in a Linked List

### 👉 Algorithm:

1. Start from the head.
2. Traverse each node and check if `data == key`.
3. If found, return the position.
4. If not, return -1.

### C Program for Searching an Element

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

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

// Function to search an element
int search(struct Node* head, int key) {
    int position = 0;
    struct Node* temp = head;

    while (temp != NULL) {
        if (temp->data == key)
            return position;
        temp = temp->next;
        position++;
    }
    return -1;
```

```
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

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

    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    int key = 20;
    int result = search(head, key);

    if (result != -1)
        printf("Element %d found at position %d\n", key, result);
    else
        printf("Element not found\n");

    return 0;
}
```

**Output:**

```
Element 20 found at position 1
```

---

## Comparison: Linked List vs. Array

| Feature | Linked List | Array |
|---|---|---|
| **Memory Allocation** | Dynamic | Static |
| **Insertion/Deletion** | Efficient (O(1) at head) | Expensive (O(n)) |
| **Access Time** | O(n) (Sequential) | O(1) (Direct access) |

| Feature | Linked List | Array |
|---|---|---|
| **Wasted Space** | Extra pointer storage | None |

---

## Time Complexities

| Operation | Best Case | Worst Case |
|---|---|---|
| **Insertion at Head** | O(1) | O(1) |
| **Insertion at Tail** | O(n) | O(n) |
| **Deletion at Head** | O(1) | O(1) |
| **Deletion at Tail** | O(n) | O(n) |
| **Searching** | O(1) (Best) | O(n) (Worst) |

---

## Conclusion

- Linked lists are **dynamic** and **efficient** for frequent insertions and deletions.
- Arrays offer **fast access** but require **fixed memory** allocation.
- Use **linked lists** when memory usage is a concern, and **arrays** when direct access is needed.

---

## Concept of Linked Lists

### What is a Linked List?

A **Linked List** is a linear data structure in which elements (called **nodes**) are connected using **pointers**. Each node consists of two parts:

1. **Data**: Stores the actual value.
2. **Pointer** (or **Next**): Holds the address of the next node in the sequence.

Unlike **arrays**, which store elements in contiguous memory locations, **linked lists** store elements **dynamically** in different memory locations, making insertion and deletion more efficient.

---

## Why Use a Linked List?

🔷 **Dynamic Size**: Unlike arrays, linked lists do not require a predefined size.
🔷 **Efficient Insertions/Deletions**: Adding or removing elements in a linked list is faster than in arrays (O(1) at the beginning).
🔷 **No Memory Wastage**: Arrays may have unused spaces, whereas linked lists use only required memory.

🚨 **Drawbacks:**
❌ **Slower Access Time**: Accessing an element in a linked list requires traversing from the head (O(n)), whereas arrays provide direct access (O(1)).
❌ **Extra Memory Usage**: Each node requires an additional pointer, which increases memory usage.

---

## Types of Linked Lists

1. **Singly Linked List (SLL)**: Each node points to the next node.
2. **Doubly Linked List (DLL)**: Each node has pointers to both the previous and next nodes.
3. **Circular Linked List (CLL)**: The last node points back to the first node.

---

## Structure of a Singly Linked List

### Node Representation in C

```
struct Node {
    int data;          // Stores value
    struct Node* next; // Pointer to next node
};
```

### Example:

Consider the linked list:

```
Head → [10 | *] → [20 | *] → [30 | NULL]
```

- The **Head** points to the first node (10).
- Each node contains:
    - **Data** (10, 20, 30).
    - **Pointer** to the next node.

---

## Basic Operations on a Linked List

### 1. Insertion

- **At the Beginning** → O(1)
- **At the End** → O(n)
- **At a Specific Position** → O(n)

### 2. Deletion

- **From the Beginning** → O(1)
- **From the End** → O(n)
- **At a Specific Position** → O(n)

### 3. Traversing

- **Visiting each node and printing its data** → O(n)

### 4. Searching

- **Finding an element in the list** → O(n)

---

## Comparison: Linked List vs. Array

| Feature | Linked List | Array |
|---|---|---|
| Memory Usage | Extra memory for pointers | Fixed size |
| Access Time | O(n) (Sequential) | O(1) (Direct Access) |
| Insertion/Deletion | Fast (O(1) at beginning) | Expensive (O(n)) |
| Memory Allocation | Dynamic | Static |

---

## When to Use a Linked List?

✅ When **frequent insertions and deletions** are needed.
✅ When memory usage is unpredictable.
✅ When you do not need **random access** to elements.

❌ **Use arrays** when frequent access is required.

Would you like me to provide **detailed C code implementations** for linked list operations? 🚀

# Representation of Linked Lists in Memory

In a **linked list**, elements (nodes) are dynamically allocated in memory and connected using **pointers**. Unlike **arrays**, which store elements in contiguous memory locations, linked lists use **non-contiguous** memory allocation.

---

## Memory Structure of a Linked List Node

Each **node** in a linked list consists of:

1. **Data** → Stores the actual value.
2. **Pointer (Next)** → Stores the address of the next node.

### C Representation of a Node

```
struct Node {
    int data;          // Stores value
    struct Node* next; // Pointer to next node
};
```

Each node occupies **memory space** for:

1. **Integer data** (4 bytes)
2. **Pointer to next node** (8 bytes on a 64-bit system or 4 bytes on a 32-bit system)

---

## 1. Memory Representation of a Singly Linked List

Consider a **singly linked list** with the elements **10 → 20 → 30 → NULL**.

### Visual Representation in Memory

**Address Data Pointer (Next)**

```
1000   10   2000

2000   20   3000

3000   30   NULL
```

### Linked Representation in Memory

```
Head → [10 | 2000] → [20 | 3000] → [30 | NULL]
```

- The **head** points to the first node.
- Each node contains:
    - **Data (10, 20, 30)**

    - **Pointer to the next node's memory address.**
- The last node points to **NULL**, indicating the end of the list.

---

## 2. Memory Representation of a Doubly Linked List

A **Doubly Linked List (DLL)** contains:

1. **Data**
2. **Pointer to the next node**
3. **Pointer to the previous node**

### C Representation of a Node

```
struct Node {
    int data;
    struct Node* prev; // Pointer to the previous node
    struct Node* next; // Pointer to the next node
};
```

### Memory Representation of a DLL with 10 ⇄ 20 ⇄ 30

**Address Prev Pointer Data Next Pointer**

```
1000   NULL      10   2000

2000   1000      20   3000

3000   2000      30   NULL
```

### Linked Representation in Memory

```
NULL ← [10 | 2000] ⇄ [20 | 3000] ⇄ [30 | NULL]
```

- Each node contains **two pointers**:
    - `prev` → Points to the previous node.
    - `next` → Points to the next node.

---

## 3. Memory Representation of a Circular Linked List

- In a **Circular Linked List (CLL)**, the last node's pointer does not point to NULL.
- Instead, it points to the **first node**, forming a loop.

### C Representation of a Circular Node

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

### Circular Linked List (10 → 20 → 30 → back to 10)

**Address Data Pointer (Next)**

```
1000   10   2000

2000   20   3000

3000   30   1000
```

### Linked Representation in Memory

```
[10 | 2000] → [20 | 3000] → [30 | 1000] → (back to 10)
```

- The last node's pointer connects back to the **head**, creating a circular structure.

---

## Comparison of Memory Usage

| Type | Pointers Per Node | Extra Memory Overhead |
|---|---|---|
| **Singly Linked List** | 1 (`next`) | Low |
| **Doubly Linked List** | 2 (`prev`, `next`) | Higher |
| **Circular Linked List** | 1 or 2 (depends on SLL or DLL) | Medium |

---

## Key Takeaways

✓ **Linked lists use dynamic memory allocation**, allowing flexible memory usage.
✓ **Each node is stored separately in memory**, unlike arrays.
✓ **Singly Linked Lists** use **one pointer**, **Doubly Linked Lists** use **two pointers**, and **Circular Linked Lists** use **a looping pointer**.

Would you like a **detailed C program** to illustrate memory representation? 🚀

# Comparison Between Linked List and Array

Linked lists and arrays are both **linear data structures**, but they differ significantly in terms of **memory allocation, access time, insertion, deletion, and flexibility**. Below is a detailed comparison:

---

## 1. Basic Differences

| Feature | Array | Linked List |
|---|---|---|
| **Definition** | A collection of elements stored in **contiguous memory locations**. | A collection of **nodes**, each storing data and a pointer to the next node. |
| **Memory Allocation** | **Static** (Fixed Size) | **Dynamic** (Grows as needed) |
| **Access Method** | **Direct (Random Access)** using index | **Sequential Access** (Must traverse nodes) |
| **Insertion/Deletion Complexity** | Costly (Shifting required) | Efficient (Only pointer changes) |
| **Extra Memory Usage** | No extra space required except for data | Requires extra memory for pointers |
| **Size Flexibility** | Fixed at declaration | Grows or shrinks dynamically |

---

## 2. Memory Allocation & Usage

| Feature | Array | Linked List |
|---|---|---|
| **Storage** | Uses **continuous** memory blocks. | Uses **non-contiguous** memory locations. |
| **Memory Overhead** | No extra memory needed. | Needs extra memory for storing **pointers**. |
| **Memory Efficiency** | Can have **wasted space** if size is overestimated. | No wastage since it grows dynamically. |

### 📌 Example:

- **Array**: `int arr[5] = {10, 20, 30, 40, 50};` → Allocates 5 consecutive memory blocks.
- **Linked List**: Each node is dynamically allocated and connected via pointers.

---

## 3. Access Time & Searching Efficiency

| Feature | Array | Linked List |
|---|---|---|
| **Access Time** | O(1) **(Direct Access using index)** | O(n) **(Traversing Required)** |

| Feature | Array | Linked List |
|---|---|---|
| **Searching** | O(n) (Linear Search), O(log n) (Binary Search) | O(n) (Must traverse nodes) |

### 📌 Example:

- **Array:** Accessing `arr[3]` is **instantaneous**.
- **Linked List:** To access the **3rd node**, we must traverse from the head.

### 🚀 When to use arrays?

- When **fast access** to elements using an index is required.

---

## 4. Insertion & Deletion Efficiency

| Feature | Array | Linked List |
|---|---|---|
| **Insertion (Beginning)** | O(n) (Shifting required) | O(1) (Only pointer update) |
| **Insertion (Middle)** | O(n) | O(n) |
| **Insertion (End)** | O(1) if space is available, O(n) otherwise | O(n) for SLL, O(1) for DLL |
| **Deletion (Beginning)** | O(n) (Shifting required) | O(1) (Only pointer update) |
| **Deletion (Middle)** | O(n) | O(n) |
| **Deletion (End)** | O(1) | O(n) for SLL, O(1) for DLL |

### 📌 Example:

- **Array:** Deleting the first element requires shifting **all elements** left.
- **Linked List:** Deleting the first node only requires changing the `head` pointer.

### 🚀 When to use linked lists?

- When **frequent insertions and deletions** are required.

---

## 5. Flexibility & Resizing

| Feature | Array | Linked List |
|---|---|---|

| Feature | Array | Linked List |
|---|---|---|
| **Size Modification** | Cannot be changed once declared (unless using **dynamic arrays** like `malloc` in C). | Can grow/shrink dynamically. |
| **Expansion Cost** | Requires **copying elements to a larger array**. | No need for resizing, as new nodes are added dynamically. |

### 📌 Example:

- **Array:** If `arr[5]` is full, a new array of **larger size** must be created.
- **Linked List:** New nodes can be added without affecting existing elements.

### 🚀 Use a linked list when dynamic memory allocation is needed.

---

## 6. Variants & Types

| Feature | Array | Linked List |
|---|---|---|
| **Variants** | One-dimensional, Two-dimensional, Multi-dimensional arrays | Singly Linked List, Doubly Linked List, Circular Linked List |

### 📌 Example:

- **Array:** `int arr[3][3]` for a **matrix**.
- **Linked List:** A **Doubly Linked List (DLL)** allows **traversal in both directions**.

### 🚀 Use linked lists when bidirectional traversal is required.

---

## 7. Applications

| Use Case | Array | Linked List |
|---|---|---|
| **Fast Access (Indexing)** | ✅ Yes | ❌ No |
| **Dynamic Memory Allocation** | ❌ No | ✅ Yes |
| **Frequent Insertions/Deletions** | ❌ No | ✅ Yes |

| Use Case | Array | Linked List |
|---|---|---|
| **Fixed Number of Elements** | ✅ Yes | ❌ No |
| **Memory Optimization** | ❌ No (Memory wastage if overestimated) | ✅ Yes (Uses exact memory) |
| **Stack Implementation** | ✅ Yes (Using array-based stack) | ✅ Yes (Using linked list-based stack) |
| **Queue Implementation** | ✅ Yes (Using circular queue) | ✅ Yes (Using linked list-based queue) |

📌 **Example Use Cases:**
✓ **Use Arrays** for **searching, sorting, and indexing** (like databases, lookup tables).
✓ **Use Linked Lists** for **dynamic memory allocation** (like memory management, undo operations).

---

## Final Summary: When to Use Which?

| Scenario | Use Array | Use Linked List |
|---|---|---|
| **Fast element access required** | ✅ Yes | ❌ No |
| **Frequent insertions/deletions** | ❌ No | ✅ Yes |
| **Memory efficiency needed** | ❌ No | ✅ Yes |
| **Fixed-size storage needed** | ✅ Yes | ❌ No |
| **Data structure like stacks/queues** | ✅ Yes | ✅ Yes |

---

## Conclusion

✓ **Arrays** are better for **fast access** and **fixed-size** storage.
✓ **Linked Lists** are better for **dynamic memory allocation**, **frequent insertions and deletions**.

Would you like **C programs** demonstrating these differences? 🚀

# Types of Linked Lists in Data Structures

A **linked list** is a **linear data structure** where elements (nodes) are stored in **non-contiguous** memory locations and connected using **pointers**. Unlike arrays, linked lists **dynamically allocate memory** and allow efficient insertions and deletions.

## 1. Singly Linked List (SLL)

### Definition:

A **singly linked list** consists of **nodes** where each node contains:

- **Data** (Stores the value)
- **Pointer to the next node**

### 🚀 Characteristics:

✓ Unidirectional (Can be traversed only **forward**)
✓ Uses **less memory** (only one pointer per node)
✓ Insertions and deletions are **faster** than arrays
✓ Cannot be traversed backward

### Structure in C:

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

### Example Representation:

```
Head → [10 | *] → [20 | *] → [30 | NULL]
```

Here, `*` stores the **address** of the next node, and the last node points to `NULL`.

---

## 2. Doubly Linked List (DLL)

### Definition:

A **doubly linked list** is similar to an SLL but each node contains:

- **Data**
- **Pointer to the next node**
- **Pointer to the previous node**

### 🚀 Characteristics:

✓ **Bidirectional traversal** (Can move forward & backward)
✓ More **memory usage** (Two pointers per node)

✓ Easier deletion of a node since it has a **backward pointer**
✓ More complex than an SLL

### Structure in C:

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

### Example Representation:

```
NULL ← [10 | * | *] ⇄ [20 | * | *] ⇄ [30 | * | NULL]
```

Each node has **two pointers**: `prev` (previous node) and `next` (next node).

---

## 3. Circular Linked List (CLL)

### Definition:

In a **circular linked list**, the last node **points back** to the first node instead of `NULL`.

### Types:

1. **Singly Circular Linked List**
   - The last node's `next` pointer points to the **head**.
   - No `NULL` at the end.
2. **Doubly Circular Linked List**
   - Both `next` and `prev` pointers form a circular connection.

### 🚀 Characteristics:

✓ No `NULL`, traversal is **continuous**
✓ Can start from any node and traverse the entire list
✓ Used in **buffered systems** (e.g., CPU scheduling, multimedia playlists)

### Structure in C (Singly CLL):

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

### Example Representation (Singly CLL):

```
Head → [10 | *] → [20 | *] → [30 | *] → (Back to Head)
```

Last node's `next` points back to the **head**, forming a loop.

---

## 4. Circular Doubly Linked List (CDLL)

### Definition:

A **circular doubly linked list** is a **doubly linked list** where:

- The **last node's next** points to the first node.
- The **first node's prev** points to the last node.

🚀 **Characteristics:**
✓ Supports **bidirectional traversal**
✓ **No NULL** pointers, continuous looping
✓ Useful in **operating systems** (e.g., round-robin scheduling)

### Structure in C:

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

### Example Representation (CDLL):

```
[10 | * | *] ⇄ [20 | * | *] ⇄ [30 | * | *] ⇄ (Back to Head)
```

Both `next` and `prev` pointers form a **loop**.

---

## Comparison of Different Types of Linked Lists

| Feature | Singly Linked List | Doubly Linked List | Circular Linked List | Circular Doubly Linked List |
|---|---|---|---|---|
| **Traversal** | One direction (Forward) | Both directions (Forward & Backward) | One direction (Forward, circular) | Both directions (Forward & Backward, circular) |
| **Memory Usage** | Less (1 pointer/node) | More (2 pointers/node) | Less (1 pointer/node) | More (2 pointers/node) |
| **Insertion/Deletion** | Fast but limited (Need to track previous node) | Easy (prev & next pointers available) | Fast (No NULL pointers) | Easy & efficient |
| **Complexity** | Simple | Moderate | Moderate | Complex |

| Feature | Singly Linked List | Doubly Linked List | Circular Linked List | Circular Doubly Linked List |
|---|---|---|---|---|
| **End of List** | `NULL` | `NULL` | Points back to `head` | Points back to `head` |
| **Use Cases** | Basic lists, stacks | Advanced lists, undo operations | Buffers, scheduling | Music playlists, OS scheduling |

---

## Conclusion

✓ **Use Singly Linked Lists** when **memory is limited** and **only forward traversal is needed**.
✓ **Use Doubly Linked Lists** when **bidirectional traversal** and **fast deletions** are required.
✓ **Use Circular Linked Lists** in **buffered systems, scheduling algorithms**.
✓ **Use Circular Doubly Linked Lists** for **complex applications like music players, OS scheduling**.

Would you like an example **C program** for each type? 🚀

# Singly Linked List (SLL) in C

A **Singly Linked List (SLL)** is a linear data structure where:

- Each node contains **data** and a **pointer to the next node**.
- The last node's pointer is set to `NULL`, marking the end of the list.

---

## Structure of a Node in C

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

Each node consists of:
✓ `data` → Stores the value.
✓ `next` → Pointer to the next node.

---

## Basic Operations on a Singly Linked List

### 1. Insertion

- At the **beginning**
- At the **end**
- At a **specific position**

### 2. Deletion

- From the **beginning**
- From the **end**
- From a **specific position**

### 3. Traversal

- Display all elements of the list.

---

## C Program for Singly Linked List

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

// Structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a node at the beginning
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = *head;
    *head = newNode;
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;
```

```
    temp->next = newNode;
}

// Function to delete a node from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

// Function to delete a node from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

    if (prev == NULL) { // Only one node in the list
        *head = NULL;
    } else {
        prev->next = NULL;
    }

    free(temp);
}

// Function to display the linked list
void displayList(struct Node* head) {
    struct Node* temp = head;

    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
```

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

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);

    displayList(head);

    deleteFromBeginning(&head);
    displayList(head);

    deleteFromEnd(&head);
    displayList(head);

    return 0;
}
```

---

## Explanation of the Code

1. **Insertion at Beginning:**
   - Create a new node.
   - Point `next` of the new node to the existing head.
   - Update head to the new node.
2. **Insertion at End:**
   - Create a new node.
   - Traverse to the last node.
   - Point the `next` of the last node to the new node.
3. **Deletion from Beginning:**
   - Update head to the second node.
   - Free the previous head node.
4. **Deletion from End:**
   - Traverse to the second-last node.
   - Set its `next` to `NULL`.
   - Free the last node.
5. **Display List:**
   - Traverse the list and print each node's `data`.

---

## Example Output

```
Linked List: 20 -> 10 -> 30 -> 40 -> NULL
Linked List: 10 -> 30 -> 40 -> NULL
Linked List: 10 -> 30 -> NULL
```

---

### Complexity Analysis

| Operation | Time Complexity |
|---|---|
| Insertion at Beginning | O(1) |
| Insertion at End | O(n) |
| Deletion from Beginning | O(1) |
| Deletion from End | O(n) |
| Traversal | O(n) |

---

### Advantages of Singly Linked List

✓ **Dynamic Size** (Can grow or shrink as needed)
✓ **Efficient Insertions/Deletions** (No shifting required like arrays)
✓ **Memory Utilization** (No pre-allocation needed)

### Disadvantages

❌ **Sequential Access Only** (Cannot access nodes directly like arrays)
❌ **Extra Memory for Pointers** (Each node stores an additional pointer)

Would you like modifications to this program (e.g., searching, inserting at a position)? 🚀

## Doubly Linked List (DLL) in C

A **Doubly Linked List (DLL)** is a linear data structure where each node contains:

- **Data** (Value stored in the node)
- **Pointer to the next node** (`next`)
- **Pointer to the previous node** (`prev`)

Unlike a **Singly Linked List (SLL)**, DLL allows **bidirectional traversal**.

---

### Structure of a Node in C

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

```
    struct Node* prev;
};
```

Each node consists of: ✓ `data` → Stores the value
✓ `next` → Pointer to the next node
✓ `prev` → Pointer to the previous node

---

## Operations on a Doubly Linked List

### 1 Insertion

- At the **beginning**
- At the **end**
- At a **specific position**

### 2 Deletion

- From the **beginning**
- From the **end**
- From a **specific position**

### 3 Traversal

- **Forward Traversal**
- **Backward Traversal**

---

## C Program for Doubly Linked List

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

// Structure for a node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to insert a node at the beginning
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = *head;
    newNode->prev = NULL;

    if (*head != NULL)
        (*head)->prev = newNode;
```

```
    *head = newNode;
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = NULL;

    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;
    newNode->prev = temp;
}

// Function to delete a node from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;

    if (*head != NULL)
        (*head)->prev = NULL;

    free(temp);
}

// Function to delete a node from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;

    while (temp->next != NULL)
        temp = temp->next;

    if (temp->prev != NULL)
        temp->prev->next = NULL;
    else
        *head = NULL; // Only one node in the list

    free(temp);
```

```
}

// Function to traverse the linked list forward
void displayForward(struct Node* head) {
    struct Node* temp = head;
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Forward Traversal: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to traverse the linked list backward
void displayBackward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;

    printf("Backward Traversal: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);

    displayForward(head);
    displayBackward(head);

    deleteFromBeginning(&head);
    displayForward(head);

    deleteFromEnd(&head);
    displayForward(head);

    return 0;
}
```

## Explanation of the Code

### Insertion at Beginning

- Create a new node.
- Set its `next` pointer to the current `head`.
- Update the `prev` pointer of the old `head` (if not NULL).
- Set the new node as the `head`.

### Insertion at End

- Create a new node.
- Traverse to the last node.
- Set the `next` pointer of the last node to the new node.
- Set the `prev` pointer of the new node to the last node.

### Deletion from Beginning

- Update `head` to point to the second node.
- Free the old head node.
- Set `prev` of the new head to `NULL`.

### Deletion from End

- Traverse to the last node.
- Set `next` of the second-last node to `NULL`.
- Free the last node.

### Forward Traversal

- Traverse from the head to the end.
- Print each node's data.

### Backward Traversal

- Traverse to the last node.
- Move backward using the `prev` pointer.

---

## Example Output

```
Forward Traversal: 20 -> 10 -> 30 -> 40 -> NULL
Backward Traversal: 40 -> 30 -> 10 -> 20 -> NULL
Forward Traversal: 10 -> 30 -> 40 -> NULL
Forward Traversal: 10 -> 30 -> NULL
```

## Complexity Analysis

| Operation | Time Complexity |
|---|---|
| Insertion at Beginning | O(1) |
| Insertion at End | O(n) |
| Deletion from Beginning | O(1) |
| Deletion from End | O(n) |
| Forward Traversal | O(n) |
| Backward Traversal | O(n) |

---

## Advantages of Doubly Linked List

✓ **Bidirectional Traversal** (Can move forward and backward)
✓ **Efficient Deletion** (Easier to delete a node without needing traversal)
✓ **More Flexibility** in implementation compared to SLL

### Disadvantages

✗ **More Memory Usage** (Each node has two pointers)
✗ **More Complex Implementation** than SLL

---

## Comparison: Singly vs. Doubly Linked List

| Feature | Singly Linked List | Doubly Linked List |
|---|---|---|
| **Memory** | Less (One pointer per node) | More (Two pointers per node) |
| **Traversal** | Only Forward | Forward & Backward |
| **Insertion/Deletion** | Slower (Need to track previous node) | Faster (Can delete node using `prev` pointer) |
| **Implementation** | Simpler | More Complex |

---

## Conclusion

✓ **Use a Singly Linked List** when memory is **limited**, and **only forward traversal** is needed.
✓ **Use a Doubly Linked List** when **bidirectional traversal** and **fast deletions** are required.

Would you like additional operations (e.g., insert at a specific position, searching)? 🚀

# Circular Singly Linked List (CSLL) in C

A **Circular Singly Linked List (CSLL)** is a variation of a **Singly Linked List** where the **last node points back to the first node**, forming a **circular structure**.

---

## Structure of a Node in C

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

Each node contains:

- **Data** → Stores the value
- **Pointer (`next`)** → Points to the next node (or the first node in case of the last node)

---

## Operations on Circular Singly Linked List

### 1 Insertion

- **At the Beginning**
- **At the End**
- **At a Specific Position**

### 2 Deletion

- **From the Beginning**
- **From the End**
- **From a Specific Position**

### 3 Traversal

- **Forward Traversal**

## C Program for Circular Singly Linked List

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

// Structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert at the beginning
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;

    if (*head == NULL) {
        newNode->next = newNode; // Points to itself (first node)
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != *head)
            temp = temp->next;

        temp->next = newNode;
        newNode->next = *head;
        *head = newNode;
    }
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;

    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != *head)
            temp = temp->next;

        temp->next = newNode;
        newNode->next = *head;
    }
}

// Function to delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
```

```
    struct Node* last = *head;

    while (last->next != *head)
        last = last->next;

    if (*head == last) { // Only one node
        free(temp);
        *head = NULL;
    } else {
        *head = temp->next;
        last->next = *head;
        free(temp);
    }
}

// Function to delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    while (temp->next != *head) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == *head) { // Only one node
        free(temp);
        *head = NULL;
    } else {
        prev->next = *head;
        free(temp);
    }
}

// Function to display the list
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("Circular Linked List: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(Back to Head)\n");
}

// Main function
int main() {
```

```
    struct Node* head = NULL;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);

    display(head);

    deleteFromBeginning(&head);
    display(head);

    deleteFromEnd(&head);
    display(head);

    return 0;
}
```

---

## Explanation of the Code

### Insertion

#### ✓ At the Beginning:

- If the list is empty, the new node **points to itself**.
- If not, traverse to the **last node** and update its `next` pointer.

#### ✓ At the End:

- If the list is empty, the new node **points to itself**.
- If not, traverse to the **last node** and update its `next` pointer.

### Deletion

#### ✓ From the Beginning:

- If the list has **only one node**, set `head = NULL`.
- Otherwise, update the `head` and set the last node's `next` to the new `head`.

#### ✓ From the End:

- If the list has **only one node**, set `head = NULL`.
- Otherwise, traverse to the **second-last node** and update its `next` pointer.

### Traversal

✓ Start from `head` and print values until we reach `head` again.

## Example Output

```
Circular Linked List: 20 -> 10 -> 30 -> 40 -> (Back to Head)
Circular Linked List: 10 -> 30 -> 40 -> (Back to Head)
Circular Linked List: 10 -> 30 -> (Back to Head)
```

---

## Complexity Analysis

| Operation | Time Complexity |
|---|---|
| Insertion at Beginning | O(n) |
| Insertion at End | O(n) |
| Deletion from Beginning | O(n) |
| Deletion from End | O(n) |
| Traversal | O(n) |

---

## Advantages of Circular Singly Linked List

✓ **Circular nature** eliminates the need for `NULL` at the end.
✓ **Efficient traversal** from any node to any other node.
✓ **Useful for applications like** CPU Scheduling, Multiplayer Games, etc.

### Disadvantages

✗ **More complex implementation** than SLL.
✗ **Requires careful handling of `next` pointers** during insertions and deletions.

---

## Comparison: Singly vs. Circular Singly Linked List

| Feature | Singly Linked List | Circular Singly Linked List |
|---|---|---|
| **Last Node Points to** | `NULL` | First Node |
| **Traversal** | One-directional | Circular (No end) |

| Feature | Singly Linked List | Circular Singly Linked List |
|---|---|---|
| **Memory Usage** | Less (Only one pointer per node) | Slightly more (Looping required) |
| **Best Use Case** | Basic linked list operations | Circular data structures (Schedulers, Buffers) |

---

## Conclusion

- **Use a Singly Linked List** when traversal **stops at the last node**.
- **Use a Circular Singly Linked List** when **continuous looping** is required (e.g., CPU Scheduling).

Would you like additional operations (insertion at a specific position, searching)? 🚀

# Circular Doubly Linked List (CDLL) in C

A **Circular Doubly Linked List (CDLL)** is a variation of a **Doubly Linked List**, where:

- **Each node contains two pointers**:
  - `next` → Points to the **next node**.
  - `prev` → Points to the **previous node**.
- **The last node connects to the first node**, making the list circular in both directions.

---

## Structure of a Node in C

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

Each node consists of:

- **Data** → Stores the value.
- **Pointer (`next`)** → Points to the **next node**.
- **Pointer (`prev`)** → Points to the **previous node**.

---

## Operations on Circular Doubly Linked List

### 1 Insertion

- **At the Beginning**
- **At the End**
- **At a Specific Position**

## 2. Deletion

- **From the Beginning**
- **From the End**
- **From a Specific Position**

## 3. Traversal

- **Forward Traversal**
- **Backward Traversal**

---

## C Program for Circular Doubly Linked List

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

// Structure for a node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to insert at the beginning
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;

    if (*head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        *head = newNode;
    } else {
        struct Node* last = (*head)->prev;
        newNode->next = *head;
        newNode->prev = last;
        last->next = newNode;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;

    if (*head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        *head = newNode;
    } else {
        struct Node* last = (*head)->prev;
```

```
        newNode->next = *head;
        newNode->prev = last;
        last->next = newNode;
        (*head)->prev = newNode;
    }
}

// Function to delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* last = (*head)->prev;

    if (*head == last) { // Only one node
        free(temp);
        *head = NULL;
    } else {
        *head = temp->next;
        (*head)->prev = last;
        last->next = *head;
        free(temp);
    }
}

// Function to delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* last = (*head)->prev;

    if (*head == last) { // Only one node
        free(last);
        *head = NULL;
    } else {
        struct Node* secondLast = last->prev;
        secondLast->next = *head;
        (*head)->prev = secondLast;
        free(last);
    }
}

// Function to display the list in forward direction
void displayForward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("Circular Doubly Linked List (Forward): ");
```

```
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(Back to Head)\n");
}

// Function to display the list in backward direction
void displayBackward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* last = head->prev;
    struct Node* temp = last;
    printf("Circular Doubly Linked List (Backward): ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->prev;
    } while (temp != last);
    printf("(Back to Last)\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);

    displayForward(head);
    displayBackward(head);

    deleteFromBeginning(&head);
    displayForward(head);

    deleteFromEnd(&head);
    displayForward(head);

    return 0;
}
```

---

## Explanation of the Code

### Insertion

#### ✓ At the Beginning:

- If the list is empty, the new node **points to itself**.
- If not, update `prev` and `next` pointers of the **head and last node**.

### ✓ At the End:

- If the list is empty, the new node **points to itself**.
- Otherwise, update `prev` and `next` pointers of the **last node**.

## Deletion

### ✓ From the Beginning:

- If only **one node exists**, free it and set `head = NULL`.
- Otherwise, update `prev` and `next` pointers of the **head and last node**.

### ✓ From the End:

- If only **one node exists**, free it and set `head = NULL`.
- Otherwise, update `prev` and `next` pointers of the **second-last node**.

## Traversal

✓ **Forward Traversal:** Start from `head` and print values until reaching `head` again.
✓ **Backward Traversal:** Start from `last` node and move backward.

---

## Example Output

```
Circular Doubly Linked List (Forward): 20 -> 10 -> 30 -> 40 -> (Back to Head)
Circular Doubly Linked List (Backward): 40 -> 30 -> 10 -> 20 -> (Back to
Last)
Circular Doubly Linked List (Forward): 10 -> 30 -> 40 -> (Back to Head)
Circular Doubly Linked List (Forward): 10 -> 30 -> (Back to Head)
```

---

## Complexity Analysis

| Operation | Time Complexity |
|---|---|
| Insertion at Beginning | O(1) |
| Insertion at End | O(1) |
| Deletion from Beginning | O(1) |
| Deletion from End | O(1) |
| Traversal | O(n) |

---

### Advantages of Circular Doubly Linked List

✓ **Efficient Insertion/Deletion** → O(1) at both ends.
✓ **Traversal in Both Directions** → `next` and `prev` pointers allow **bidirectional movement**.
✓ **No `NULL` Pointers** → Continuous circular structure.

### Disadvantages

✗ **More Memory Usage** → Extra `prev` pointer.
✗ **More Complex Operations** → Need to update **both `next` and `prev` pointers**.

---

### Comparison: Singly, Doubly & Circular Doubly Linked List

| Feature | Singly Linked List | Doubly Linked List | Circular Doubly Linked List |
|---|---|---|---|
| **Next Pointer** | ✓ | ✓ | ✓ |
| **Prev Pointer** | ✗ | ✓ | ✓ |
| **Circular Nature** | ✗ | ✗ | ✓ |
| **Bidirectional Traversal** | ✗ | ✓ | ✓ |
| **Use Case** | Basic list | Efficient traversal | Circular applications |

---

### Conclusion

- **Use a Circular Doubly Linked List** when you need **efficient bidirectional traversal** with a circular structure.
- It is **widely used in applications like** buffer management, scheduling, and undo-redo operations.

Would you like additional operations, such as searching or inserting at a specific position? 🚀

## Implementation of Linked List as an Abstract Data Type (ADT) in C

### What is an Abstract Data Type (ADT)?

An **Abstract Data Type (ADT)** is a **logical description** of a data structure that defines operations without specifying the implementation details.

## Linked List as an ADT

A **Linked List ADT** provides:

- **Encapsulation** → Hides implementation details.
- **Standard operations** → Insert, delete, search, traverse.

---

## Structure of a Node in C

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

Each node contains:

- `data` → Stores the value.
- `next` → Pointer to the next node.

---

## Operations on Linked List ADT

### 1 Basic Operations

✓ **Insertion**
✓ **Deletion**
✓ **Search**
✓ **Traversal**

### 2 ADT Definition

```
struct LinkedList {
    struct Node* head;
    void (*insert)(struct LinkedList*, int);
    void (*delete)(struct LinkedList*, int);
    struct Node* (*search)(struct LinkedList*, int);
    void (*display)(struct LinkedList*);
};
```

- `head` → Points to the first node.
- **Function pointers** for operations.

---

## C Implementation of Linked List ADT

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

// Node structure
```

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

// Linked List ADT structure
struct LinkedList {
    struct Node* head;
    void (*insert)(struct LinkedList*, int);
    void (*delete)(struct LinkedList*, int);
    struct Node* (*search)(struct LinkedList*, int);
    void (*display)(struct LinkedList*);
};

// Function to insert a node at the end
void insertNode(struct LinkedList* list, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (list->head == NULL) {
        list->head = newNode;
        return;
    }

    struct Node* temp = list->head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to delete a node by value
void deleteNode(struct LinkedList* list, int value) {
    struct Node* temp = list->head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == value) {
        list->head = temp->next;
        free(temp);
        return;
    }

    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

// Function to search for a node by value
struct Node* searchNode(struct LinkedList* list, int value) {
    struct Node* temp = list->head;
```

```
    while (temp != NULL) {
        if (temp->data == value) {
            return temp;
        }
        temp = temp->next;
    }
    return NULL;
}

// Function to display the linked list
void displayList(struct LinkedList* list) {
    struct Node* temp = list->head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to initialize the Linked List ADT
void initializeLinkedList(struct LinkedList* list) {
    list->head = NULL;
    list->insert = insertNode;
    list->delete = deleteNode;
    list->search = searchNode;
    list->display = displayList;
}

// Main function to demonstrate Linked List ADT
int main() {
    struct LinkedList list;
    initializeLinkedList(&list);

    list.insert(&list, 10);
    list.insert(&list, 20);
    list.insert(&list, 30);
    list.display(&list);

    list.delete(&list, 20);
    list.display(&list);

    struct Node* found = list.search(&list, 30);
    if (found) {
        printf("Element %d found in the list.\n", found->data);
    } else {
        printf("Element not found.\n");
    }

    return 0;
}
```

## Explanation of the Code

### 1 ADT Definition

- **Encapsulated structure** `LinkedList`.
- **Function pointers** for `insert`, `delete`, `search`, and `display`.

### 2 Operations

**✓ Insertion:**

- If the list is **empty**, set `head = newNode`.
- Else, traverse to the **end** and attach the new node.

**✓ Deletion:**

- If the node to be deleted is the **head**, update `head = head->next`.
- Else, find the **previous node** and update `prev->next = temp->next`.

**✓ Search:**

- Traverse the list to **find the value**.

**✓ Traversal (Display):**

- Start from `head` and print each node.

---

## Example Output

```
Linked List: 10 -> 20 -> 30 -> NULL
Linked List: 10 -> 30 -> NULL
Element 30 found in the list.
```

---

## Time Complexity Analysis

| Operation | Time Complexity |
|---|---|
| **Insertion** | O(n) (At end) |
| **Deletion** | O(n) |
| **Search** | O(n) |
| **Traversal** | O(n) |

---

## Advantages of Linked List ADT

✓ **Encapsulation** → Implementation details hidden.
✓ **Flexible** → Can **modify operations easily**.
✓ **Reusability** → Same ADT can be used in **different applications**.

---

## Conclusion

🔷 **Linked List ADT** helps structure linked list operations in a clean, reusable, and maintainable way.
🔷 Using **function pointers** ensures **flexibility and modularity**.
🔷 Useful in **memory management, dynamic data structures, and real-time applications**.

Would you like to see **more operations like reversing or sorting**? 🚀

## Creating a Linked List in C

A **linked list** is a dynamic data structure consisting of nodes, where each node contains:

1. **Data** → Stores the actual value.
2. **Pointer** → Stores the address of the next node.

---

## Structure of a Node

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

Each node contains an integer `data` and a pointer `next` that points to the next node.

---

## Steps to Create a Linked List

### 1️⃣ Create a Node

- Allocate memory dynamically using `malloc()`.
- Assign data to the node.
- Set the `next` pointer to `NULL`.

### 2. Insert Nodes

- Append new nodes at the end or at the beginning.

### 3. Traverse and Display the List

- Start from `head` and print each node.

---

## C Program to Create a Simple Linked List

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

// Define Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to display the linked list
void displayList(struct Node* head) {
    struct Node* temp = head;

    if (temp == NULL) {
        printf("List is empty.\n");
```

```
        return;
    }

    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL; // Initialize an empty list

    // Insert elements
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    // Display the linked list
    displayList(head);

    return 0;
}
```

---

## Explanation of Code

### 1 Creating a Node

- `createNode(int value)` → Allocates memory for a new node and initializes it.

### 2 Inserting a Node

- `insertAtEnd(&head, value)` → Adds a new node at the end of the list.

### 3 Displaying the List

- `displayList(head)` → Traverses the list and prints all values.

---

## Output

```
Linked List: 10 -> 20 -> 30 -> NULL
```

---

## Time Complexity

| Operation | Complexity |
|---|---|

| Operation | Complexity |
|---|---|
| Create a Node | O(1) |
| Insert at End | O(n) |
| Display List | O(n) |

---

## Summary

✅ **Dynamic** → No need to define array size.
✅ **Efficient Memory Usage** → Uses only required space.
✅ **Insertion & Deletion** → Easier compared to arrays.

Would you like to see **insertion at the beginning or deletion operations**? 🚀

## Traversing a Linked List in C

**Traversal** means visiting each node of the linked list and processing its data.

---

## Steps for Traversal

1. **Start from `head`** → Initialize a pointer to the head node.
2. **Move through each node** → Use a loop to visit each node.
3. **Print/process each node** → Access and display the `data` of each node.
4. **Stop when `NULL` is reached** → The last node's `next` pointer is `NULL`.

---

## C Program to Traverse a Linked List

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

// Define Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
```

```
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to traverse and display the linked list
void traverseList(struct Node* head) {
    struct Node* temp = head;

    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL; // Initialize an empty list

    // Insert elements
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    // Traverse and display the linked list
    traverseList(head);

    return 0;
}
```

---

## Explanation

1. **`traverseList(struct Node* head)`**
   - Starts from the `head`.
   - Uses a `while` loop to visit each node.
   - Prints the `data` of each node.
   - Stops when `NULL` is reached.
2. **Main Function**
   - Creates an empty linked list.
   - Inserts `10, 20, 30`.
   - Calls `traverseList(head)` to display the list.

---

## Output

```
Linked List: 10 -> 20 -> 30 -> NULL
```

---

## Time Complexity

**Operation Complexity**

Traversing **O(n)**

---

## Summary

✅ **Simple & Efficient** → Visits each node once.
✅ **O(n) Time Complexity** → Proportional to the number of nodes.
✅ **Used for Searching, Displaying, Counting Nodes, etc.**

Would you like to implement **reverse traversal or searching**? 🚀

## Searching in a Linked List

Searching in a **linked list** involves traversing the list node by node and checking if a specific value exists.

### Types of Searching in a Linked List

1. **Linear Search** → Start from the `head` and compare each node's data with the key.
2. **Binary Search (Not Possible)** → Since linked lists are not indexed, binary search is inefficient.

---

## C Program for Searching in a Linked List

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

// Define Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to search for an element in the linked list
int search(struct Node* head, int key) {
    struct Node* temp = head;
    int position = 1;

    while (temp != NULL) {
        if (temp->data == key) {
            return position;  // Return the position if found
        }
        temp = temp->next;
        position++;
    }
    return -1; // Return -1 if not found
}

// Main function
int main() {
    struct Node* head = NULL;
```

```
    // Insert elements into the linked list
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);

    // Search for a key in the linked list
    int key = 30;
    int result = search(head, key);

    if (result != -1)
        printf("Element %d found at position %d\n", key, result);
    else
        printf("Element %d not found in the list\n", key);

    return 0;
}
```

---

## Explanation

1. **`search(struct Node* head, int key)`**
   - Starts from the `head`.
   - Checks if `data` matches `key`.
   - If found, returns the position.
   - If `NULL` is reached, returns `-1`.
2. **Main Function**
   - Inserts `10, 20, 30, 40`.
   - Searches for `30`.
   - Displays the result.

---

## Output

```
Element 30 found at position 3
```

---

## Time Complexity

**Operation Complexity**

Searching **O(n)**

### Why O(n)?

- In the worst case, we may have to check all `n` nodes.

---

## Summary

✅ **Simple and Effective**
✅ **Best for Unsorted Lists**
✅ **O(n) Time Complexity**

Would you like to implement **deletion or sorting** next? 🚀

## Insertion and Deletion in a Linked List

Insertion and deletion operations in a **singly linked list** can be performed at:

1. **Beginning (First Node)**
2. **Specified Position**
3. **End (Last Node)**

---

## 🔷 C Program for Insertion & Deletion in Linked List

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

// Define Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning
void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
}

// Function to insert a node at a specific position
void insertAtPosition(struct Node** head, int value, int position) {
    if (position < 1) {
```

```
        printf("Invalid position!\n");
        return;
    }

    struct Node* newNode = createNode(value);

    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds!\n");
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to delete the first node
void deleteFirstNode(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

// Function to delete a node at a specific position
void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL || position < 1) {
```

```
        printf("Invalid position!\n");
        return;
    }

    struct Node* temp = *head;

    if (position == 1) {
        *head = temp->next;
        free(temp);
        return;
    }

    struct Node* prev = NULL;
    for (int i = 1; temp != NULL && i < position; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds!\n");
        return;
    }

    prev->next = temp->next;
    free(temp);
}

// Function to delete the last node
void deleteLastNode(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp->next == NULL) {
        free(temp);
        *head = NULL;
        return;
    }

    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

    prev->next = NULL;
    free(temp);
}

// Function to display the linked list
void displayList(struct Node* head) {
    struct Node* temp = head;

    if (temp == NULL) {
```

```
        printf("List is empty.\n");
        return;
    }

    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL; // Initialize an empty list

    // Insert elements
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    displayList(head);

    // Insert at beginning
    insertAtBeginning(&head, 5);
    displayList(head);

    // Insert at position 3
    insertAtPosition(&head, 15, 3);
    displayList(head);

    // Insert at end
    insertAtEnd(&head, 40);
    displayList(head);

    // Delete first node
    deleteFirstNode(&head);
    displayList(head);

    // Delete node at position 2
    deleteAtPosition(&head, 2);
    displayList(head);

    // Delete last node
    deleteLastNode(&head);
    displayList(head);

    return 0;
}
```

---

## 📌 Explanation

### 1️⃣ Insertion Functions

- `insertAtBeginning()` → Inserts at the head.

- `insertAtPosition()` → Inserts at a specific position.
- `insertAtEnd()` → Appends at the tail.

### 2️⃣ Deletion Functions

- `deleteFirstNode()` → Removes the first node.
- `deleteAtPosition()` → Removes from a given position.
- `deleteLastNode()` → Removes the last node.

---

## 📌 Output

```
Linked List: 10 -> 20 -> 30 -> NULL
Linked List: 5 -> 10 -> 20 -> 30 -> NULL
Linked List: 5 -> 10 -> 15 -> 20 -> 30 -> NULL
Linked List: 5 -> 10 -> 15 -> 20 -> 30 -> 40 -> NULL
Linked List: 10 -> 15 -> 20 -> 30 -> 40 -> NULL
Linked List: 10 -> 20 -> 30 -> 40 -> NULL
Linked List: 10 -> 20 -> 30 -> NULL
```

---

## 🔷 Time Complexity

| Operation | Complexity |
|---|---|
| Insert at Beginning | O(1) |
| Insert at End | O(n) |
| Insert at Position | O(n) |
| Delete First Node | O(1) |
| Delete Last Node | O(n) |
| Delete at Position | O(n) |

---

## 🔷 Summary

✅ **Insertion & Deletion** → Efficient for dynamic memory.
✅ **O(n) Complexity** → Traversing required for middle or end operations.
✅ **Better than Arrays** → No shifting required.

Would you like **Circular Linked List** or **Doubly Linked List** next? 🚀

# 📌 Applications of Linked Lists

Linked lists are widely used in various real-world applications due to their **dynamic memory allocation**, **efficient insertions/deletions**, and **flexibility**. Below are some common applications:

---

## 1️⃣ Dynamic Memory Allocation

- Linked lists allow **efficient memory utilization** as memory is allocated dynamically.
- Unlike arrays, linked lists **do not require a predefined size**, reducing memory wastage.
- Used in **Operating Systems, File Management Systems, and Memory Management**.

---

## 2️⃣ Implementation of Data Structures

- **Stacks and Queues** → Implemented using singly or doubly linked lists.
- **Graphs** → Adjacency list representation of graphs uses linked lists.
- **Hash Tables** → Chaining method for collision handling uses linked lists.

---

## 3️⃣ Undo & Redo Operations in Editors

- **Text editors (e.g., Notepad, Word)** store **Undo/Redo** actions as a linked list.
- Every change is stored as a new node, allowing easy traversal for **undo/redo operations**.

---

## 4️⃣ Music & Video Playlists

- **Media Players (Spotify, VLC, YouTube Playlists)** use linked lists to navigate between songs/videos.
- **Doubly Linked List** is used for **next & previous song navigation**.

---

## 5️⃣ Web Browser Forward & Backward Navigation

- **Browser History (Chrome, Firefox, Edge)** is implemented using **Doubly Linked Lists**.
- Clicking "Back" moves to the previous page, and "Forward" moves to the next page.

---

## 6️⃣ Implementation of File Systems

- **File Allocation Table (FAT) and NTFS (New Technology File System)** use linked lists to store file locations.
- Each file block is stored as a node in a linked list, making it easy to insert/delete blocks.

---

## 7️⃣ Polynomial Arithmetic

- **Linked lists** help represent and process large **polynomials** efficiently.
- Example: Storing $5x^2 + 3x + 7$ as nodes $(5,2) \rightarrow (3,1) \rightarrow (7,0)$.
- Useful in **Scientific Computing & Symbolic Mathematics**.

---

## 8️⃣ Image Viewer Applications

- Applications like **Photos and Gallery Apps** use linked lists to navigate images.
- **Next/Previous** buttons work using **Doubly Linked Lists**.

---

## 9️⃣ Scheduling in Operating Systems

- **CPU Scheduling Algorithms (Round-Robin, Multi-level Queue)** use linked lists to maintain process queues.
- **Task Management & Process Scheduling** are efficiently handled using circular linked lists.

---

## 🔷 Summary

| Application | Linked List Type Used |
|---|---|
| Dynamic Memory Allocation | Singly Linked List |
| Stacks & Queues | Singly or Doubly Linked List |
| Undo/Redo in Editors | Doubly Linked List |
| Music & Video Playlists | Doubly Linked List |
| Browser History Navigation | Doubly Linked List |
| File Systems (FAT, NTFS) | Singly Linked List |

| Application | Linked List Type Used |
|---|---|
| Polynomial Arithmetic | Singly Linked List |
| Image Viewers | Doubly Linked List |
| CPU Scheduling (Round-Robin) | Circular Linked List |

---

## 🚀 Conclusion

✅ **Efficient Memory Management**
✅ **Faster Insertions & Deletions** (compared to arrays)
✅ **Widely Used in OS, Editors, Playlists, and Scheduling**

Would you like **real-world coding examples** for any of these applications? 🚀

# UNIT-III

## 📌 Stacks in Data Structures

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

---

### 🔹 Basic Operations of Stack

| Operation | Description |
|---|---|
| `Push(x)` | Inserts (pushes) an element `x` onto the stack. |
| `Pop()` | Removes the top element from the stack. |
| `Peek()/Top()` | Returns the top element without removing it. |
| `isEmpty()` | Checks if the stack is empty. |
| `isFull()` | Checks if the stack is full (for an array-based stack). |

---

### 🔹 Stack Implementation

Stacks can be implemented in **two ways**:

1. **Using Arrays (Static Stack)**
2. **Using Linked Lists (Dynamic Stack)**

---

## 1️⃣ Stack Implementation Using Arrays

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

#define MAX 5  // Maximum size of the stack

// Stack structure
struct Stack {
    int arr[MAX];
    int top;
};
```

```
// Function to initialize the stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

// Function to push an element onto the stack
void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    s->arr[++(s->top)] = value;
}

// Function to pop an element from the stack
int pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    }
    return s->arr[(s->top)--];
}

// Function to return the top element
int peek(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    }
    return s->arr[s->top];
}

// Function to display the stack
void display(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack: ");
    for (int i = s->top; i >= 0; i--) {
        printf("%d ", s->arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
```

```
    struct Stack s;
    initialize(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    display(&s);

    printf("Popped: %d\n", pop(&s));
    display(&s);

    printf("Top element: %d\n", peek(&s));

    return 0;
}
```

### 📌 Output

```
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

---

## 2 Stack Implementation Using Linked List

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

// Define a node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to push an element onto the stack
void push(struct Node** top, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack Overflow!\n");
        return;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
}

// Function to pop an element from the stack
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow!\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedValue;
```

```
}

// Function to return the top element
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return -1;
    }
    return top->data;
}

// Function to display the stack
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* top = NULL;

    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    display(top);

    printf("Popped: %d\n", pop(&top));
    display(top);

    printf("Top element: %d\n", peek(top));

    return 0;
}
```

### 📌 Output

```
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

---

# 📌 Applications of Stack

## 1️⃣ Function Call Management (Recursion)

- Each function call is stored in a **stack frame** in memory.

- When a function returns, the stack unwinds (LIFO).

### 2️⃣ Undo/Redo Operations

- Used in **text editors (Notepad, Word)** and **IDEs**.

### 3️⃣ Expression Evaluation

- Used to evaluate **postfix, infix, and prefix expressions**.
- Example: `2 3 +` (Postfix Expression)

### 4️⃣ Backtracking (Maze Solving, Game AI)

- Used in **DFS (Depth-First Search)** in graphs.
- Example: **Solving a maze** by storing paths.

### 5️⃣ Browser Back/Forward Navigation

- **Web browsers (Chrome, Firefox, Edge)** use **two stacks**:
  - One for back navigation.
  - Another for forward navigation.

### 6️⃣ Reversing a String

- Push each character onto a stack, then pop to get the reversed string.

---

## 📌 Comparison: Array vs Linked List Implementation

| Feature | Array-Based Stack | Linked List-Based Stack |
|---|---|---|
| Memory Usage | Fixed (static) | Dynamic (grows as needed) |
| Speed | Faster (O(1) push/pop) | Slightly slower (O(1) push/pop, but needs memory allocation) |
| Size Limitation | Fixed (predefined size) | No limitation (depends on available memory) |
| Complexity | O(1) for push/pop | O(1) for push/pop |

---

## 🚀 Summary

- ✅ **Stacks follow LIFO (Last In, First Out)**
- ✅ **Implemented using arrays or linked lists**
- ✅ **Used in Recursion, Undo/Redo, Browser Navigation, Function Calls, DFS, etc.**

# 📌 Introduction to Stack ADT (Abstract Data Type)

## 🔹 What is a Stack?

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

---

## 🔹 What is an ADT (Abstract Data Type)?

An **Abstract Data Type (ADT)** is a data structure defined by its **operations** rather than its implementation. It defines **what** operations can be performed on the data but not **how** they are implemented.

**Example:**
A **Stack ADT** defines operations like `push()`, `pop()`, and `peek()`, but it doesn't specify whether the stack is implemented using **arrays** or **linked lists**.

---

## 🔹 Stack ADT Operations

| Operation | Description |
|---|---|
| `push(x)` | Inserts an element `x` onto the stack. |
| `pop()` | Removes and returns the top element of the stack. |
| `peek()/top()` | Returns the top element without removing it. |
| `isEmpty()` | Checks if the stack is empty. |
| `isFull()` | Checks if the stack is full (for array implementation). |

## 💡 Stack ADT Representation

A stack stores elements **sequentially**, with insertion and deletion happening **only at the top**.

```
Stack (Top at 30)
+----+
| 30 |  <-- Top
+----+
| 20 |
+----+
| 10 |
+----+
```

**Push(40)** → Add 40 at the top
**Pop()** → Remove 40 from the top

---

## 🔷 Stack ADT Implementation

Stack can be implemented using:

1. **Array (Static Stack)**
2. **Linked List (Dynamic Stack)**

### 1️⃣ Stack ADT using Arrays

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

#define MAX 5  // Maximum stack size

struct Stack {
    int arr[MAX];
    int top;
};

// Initialize stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Check if stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Check if stack is full
int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

// Push operation
```

```
void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow!\n");
        return;
    }
    s->arr[++(s->top)] = value;
}

// Pop operation
int pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    }
    return s->arr[(s->top)--];
}

// Peek operation
int peek(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    }
    return s->arr[s->top];
}

// Display stack
void display(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack: ");
    for (int i = s->top; i >= 0; i--) {
        printf("%d ", s->arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    struct Stack s;
    initialize(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    display(&s);

    printf("Popped: %d\n", pop(&s));
    display(&s);

    printf("Top element: %d\n", peek(&s));

    return 0;
}
```

### 📌 Output

```
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

---

### 2️⃣ Stack ADT using Linked List

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

// Define a node
struct Node {
    int data;
    struct Node* next;
};

// Push operation
void push(struct Node** top, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack Overflow!\n");
        return;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
}

// Pop operation
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow!\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedValue;
}

// Peek operation
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return -1;
    }
    return top->data;
}

// Display stack
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
```

```
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* top = NULL;

    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    display(top);

    printf("Popped: %d\n", pop(&top));
    display(top);

    printf("Top element: %d\n", peek(top));

    return 0;
}
```

### 📌 Output

```
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

---

## 🔷 Applications of Stack ADT

### 1️⃣ Function Call Management (Recursion)

- Every function call is stored in a **stack frame** in memory.
- When a function returns, the last called function is popped from the stack.

### 2️⃣ Undo/Redo Operations

- Used in **text editors (Notepad, Word, IDEs)**.
- Each edit is pushed onto a stack, allowing undo and redo operations.

### 3️⃣ Expression Evaluation

- Used to evaluate **Postfix, Infix, and Prefix expressions**.
- Example: `2 3 +` (Postfix Evaluation)

### 4️⃣ Backtracking (Maze Solving, Game AI)

- Used in **Depth-First Search (DFS)** to store visited nodes.

### 5️⃣ Browser Back/Forward Navigation

- Web browsers use **two stacks**:
  - One stack for "Back" navigation.
  - Another stack for "Forward" navigation.

---

## 📌 Comparison: Array vs Linked List Stack Implementation

| Feature | Array-Based Stack | Linked List-Based Stack |
|---|---|---|
| Memory Usage | Fixed (static) | Dynamic (grows as needed) |
| Speed | Faster (O(1) push/pop) | Slightly slower (O(1) push/pop, but needs memory allocation) |
| Size Limitation | Fixed (predefined size) | No limitation (depends on available memory) |
| Complexity | O(1) for push/pop | O(1) for push/pop |

---

## 🚀 Summary

✅ **Stack ADT follows LIFO (Last In, First Out)**
✅ **Common operations: push(), pop(), peek(), isEmpty()**
✅ **Used in Recursion, Undo/Redo, Browser Navigation, Function Calls, DFS, etc.**

## 📌 Representation of Stacks using Arrays and Linked Lists

A **stack** is a **linear data structure** that follows the **LIFO (Last In, First Out)** principle. There are two primary ways to represent a stack in memory:

1️⃣ **Using Arrays (Static Stack)**
2️⃣ **Using Linked Lists (Dynamic Stack)**

---

# 1️⃣ Stack Representation using Arrays

## 📌 Characteristics

- ✅ **Fixed size** (predefined size)
- ✅ **Stored in contiguous memory locations**
- ✅ **Fast access** (O(1) for push and pop)
- ✅ **Prone to overflow** (if the array is full)

## 🛠️ Implementation of Stack using Array in C

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5  // Maximum size of stack

struct Stack {
    int arr[MAX];
    int top;
};

// Initialize stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Check if stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Check if stack is full
int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

// Push operation
void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow!\n");
        return;
    }
    s->arr[++(s->top)] = value;
}

// Pop operation
int pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow!\n");
        return -1;
    }
    return s->arr[(s->top)--];
}

// Peek operation
```

```
int peek(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    }
    return s->arr[s->top];
}

// Display stack
void display(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack: ");
    for (int i = s->top; i >= 0; i--) {
        printf("%d ", s->arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    struct Stack s;
    initialize(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    display(&s);

    printf("Popped: %d\n", pop(&s));
    display(&s);

    printf("Top element: %d\n", peek(&s));

    return 0;
}
```

## 📌 Output

```
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

## 📌 Stack Representation using Array

```
Initial Stack:       After Push(10, 20, 30):     After Pop():
[ ]  <- top = -1      [10] <- top = 0            [10]
[ ]                   [20] <- top = 1            [20] <- top = 1
[ ]                   [30] <- top = 2            [ ]  <- top = 0
```

---

## 2 Stack Representation using Linked List

### Characteristics

- ✅ **Dynamic size** (no fixed limit)
- ✅ **Elements are stored in non-contiguous memory locations**
- ✅ **Efficient memory usage**
- ✅ **Slower than array-based stack (due to memory allocation overhead)**

### Implementation of Stack using Linked List in C

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

// Define a node
struct Node {
    int data;
    struct Node* next;
};

// Push operation
void push(struct Node** top, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack Overflow!\n");
        return;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
}

// Pop operation
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow!\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedValue;
}

// Peek operation
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return -1;
    }
    return top->data;
}
```

```
// Display stack
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* top = NULL;

    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    display(top);

    printf("Popped: %d\n", pop(&top));
    display(top);

    printf("Top element: %d\n", peek(top));

    return 0;
}
```

### 📌 Output

```
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

### 📌 Stack Representation using Linked List

```
Initial Stack:       After Push(10, 20, 30):     After Pop():
NULL                 NULL                        NULL
                       ↓                           ↓
                     [10]                        [10]
                       ↓                           ↓
                     [20]                        [20]
                       ↓
                     [30] <- top
```

---

## 📌 Comparison: Array vs Linked List Stack

| Feature | Array-Based Stack | Linked List-Based Stack |
|---|---|---|
| Memory Usage | Fixed (predefined size) | Dynamic (grows as needed) |

| Feature | Array-Based Stack | Linked List-Based Stack |
|---|---|---|
| Speed | Faster (O(1) push/pop) | Slightly slower (O(1) push/pop, but needs memory allocation) |
| Size Limitation | Fixed (predefined size) | No limitation (depends on available memory) |
| Overflow | Can occur if array is full | No overflow unless memory is exhausted |
| Underflow | Occurs if stack is empty | Occurs if stack is empty |
| Implementation | Uses a contiguous array | Uses dynamically allocated nodes |
| Preferred When | Memory is limited, fast access needed | Flexible size, memory efficiency |

---

## 📌 Summary

✅ **Stack can be implemented using both arrays and linked lists.**
✅ **Array-based stacks are faster but have a fixed size.**
✅ **Linked List-based stacks use dynamic memory but are slightly slower due to memory allocation.**
✅ **Both implementations support `push()`, `pop()`, and `peek()` operations.**
✅ **Stacks are used in recursion, undo/redo operations, expression evaluation, and function calls.**

Would you like **more real-world applications of stacks** or **detailed explanations of push/pop algorithms**? 🚀

### 📌 Implementation of Stacks in C

A **stack** is a **linear data structure** that follows the **LIFO (Last In, First Out)** principle. There are two common ways to implement a stack in **C**:

1️⃣ **Using Arrays (Static Implementation)**
2️⃣ **Using Linked Lists (Dynamic Implementation)**

---

## 1️⃣ Stack Implementation using Arrays

### 📌 Characteristics

✅ Uses a **fixed-size** array
✅ **Fast operations** (O(1) for push and pop)
✅ **Risk of overflow** if the stack is full

### 🛠️ Code Implementation

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5  // Define the maximum size of stack

// Define Stack structure
struct Stack {
    int arr[MAX];  // Array to hold stack elements
    int top;       // Top pointer
};

// Function to initialize stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Check if stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Check if stack is full
int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

// Push operation
void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    s->arr[++(s->top)] = value;
    printf("Pushed %d into stack\n", value);
}

// Pop operation
int pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    }
    return s->arr[(s->top)--];
}

// Peek operation
```

```
int peek(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    }
    return s->arr[s->top];
}

// Display stack
void display(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack: ");
    for (int i = s->top; i >= 0; i--) {
        printf("%d ", s->arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    struct Stack s;
    initialize(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    display(&s);

    printf("Popped: %d\n", pop(&s));
    display(&s);

    printf("Top element: %d\n", peek(&s));

    return 0;
}
```

### 📌 Output

```
Pushed 10 into stack
Pushed 20 into stack
Pushed 30 into stack
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

---

## 🔷 2️⃣ Stack Implementation using Linked List

### 📌 Characteristics

✅ Uses **dynamic memory allocation** (no size limit)
✅ **No overflow** unless memory is exhausted
✅ **Slower than array implementation** due to memory allocation overhead

### 🛠️ Code Implementation

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

// Define a Node structure
struct Node {
    int data;
    struct Node* next;
};

// Push operation
void push(struct Node** top, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d into stack\n", value);
}

// Pop operation
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    }
    struct Node* temp = *top;
    int poppedValue = temp->data;
    *top = (*top)->next;
    free(temp);
    return poppedValue;
}

// Peek operation
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return -1;
    }
    return top->data;
}

// Display stack
```

```
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack: ");
    struct Node* temp = top;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* top = NULL;

    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    display(top);

    printf("Popped: %d\n", pop(&top));
    display(top);

    printf("Top element: %d\n", peek(top));

    return 0;
}
```

### 📌 Output

```
Pushed 10 into stack
Pushed 20 into stack
Pushed 30 into stack
Stack: 30 20 10
Popped: 30
Stack: 20 10
Top element: 20
```

---

## 📌 Comparison: Stack using Array vs Linked List

| Feature | Stack Using Array | Stack Using Linked List |
|---|---|---|
| **Memory** | Fixed size, preallocated | Dynamic memory allocation |
| **Speed** | Faster (O(1) for push/pop) | Slightly slower (O(1) for push/pop due to memory allocation) |
| **Size Limit** | Limited by array size | No fixed limit (until memory runs out) |

| Feature | Stack Using Array | Stack Using Linked List |
|---|---|---|
| **Overflow** | Possible if the array is full | No overflow unless memory is exhausted |
| **Underflow** | Happens when the stack is empty | Happens when the stack is empty |
| **Implementation** | Uses a contiguous array | Uses dynamically allocated nodes |
| **Use Case** | When a fixed maximum size is known | When flexible size is needed |

---

## 📌 Summary

✅ **Stack can be implemented using both arrays and linked lists.**
✅ **Array-based stacks** are **faster** but have a **fixed size**.
✅ **Linked list-based stacks** are **dynamic** but slightly **slower**.
✅ **Both implementations support push(), pop(), and peek() operations.**
✅ **Stacks are widely used in recursion, undo/redo operations, expression evaluation, function calls, and backtracking.**

Would you like more examples or explanations on **real-world applications of stacks**? 🚀

## 📌 Applications of Stacks

A **stack** is a **LIFO (Last In, First Out)** data structure, and its unique behavior makes it useful in various applications in **computer science** and **real-world scenarios**. Below are some key applications:

---

## 1️⃣ Function Call Management (Recursion)

### 📌 How it Works?

- The **call stack** manages function calls in programming languages.
- Each function call **pushes** an activation record onto the stack.
- When a function returns, its activation record is **popped** from the stack.

### 🔷 Example

```
void fun(int n) {
    if (n == 0) return;
    printf("%d ", n);
```

```
    fun(n - 1); // Recursive call
    printf("%d ", n);
}
```

🔷 **Call Stack Behavior for `fun(3)`:**

```
Push fun(3) → Push fun(2) → Push fun(1) → Push fun(0) (Base Case) → Pop fun(1) → Pop fun(2) → Pop fun(3)
```

📌 **Used In:** Function calls, recursion, OS process execution.

---

## 2️⃣ Undo/Redo Operations in Text Editors

### 📌 How it Works?

- **Undo:** The last performed operation is stored in a stack and can be popped to revert to the previous state.
- **Redo:** The undone operations are stored in another stack and can be re-applied.

📌 **Used In:** MS Word, Notepad, Photoshop, Web Browsers (Back/Forward navigation)

---

## 3️⃣ Expression Evaluation (Postfix & Prefix Notation)

### 📌 How it Works?

- **Infix expression:** `(A + B) * C`
- **Postfix expression:** `A B + C *`
- Stack helps in converting **infix expressions** to **postfix/prefix** and evaluates them.

📌 **Used In:** Compilers, calculators.

---

## 4️⃣ Balancing Parentheses & Syntax Checking

### 📌 How it Works?

- The stack helps check if parentheses, brackets, and braces are balanced in expressions.
- Example: `{ [ ( ) ] }` is balanced, but `{ [ ( ] ) }` is not.

📌 **Used In:** Programming language parsers, HTML/XML tag validation.

---

## 5️⃣ Backtracking (Maze Solving, Puzzle Solving)

### 📌 How it Works?

- Stack stores previous choices so we can go **back** if a wrong path is chosen.
- Example: Solving a **maze** using DFS (Depth First Search).

📌 **Used In:** Game AI, pathfinding, puzzle solving.

---

## 6️⃣ Browser Back & Forward Navigation

### 📌 How it Works?

- When a user visits a webpage, the URL is **pushed** onto a stack.
- Clicking "Back" **pops** the last visited page.
- Clicking "Forward" **pushes** the next visited page.

📌 **Used In:** Google Chrome, Firefox, Safari.

---

## 7️⃣ Memory Management (Stack vs Heap)

### 📌 How it Works?

- **Stack Memory:** Stores local variables, function calls.
- **Heap Memory:** Stores dynamically allocated memory.

📌 **Used In:** Operating systems, memory allocation.

---

## 8️⃣ String Reversal

### 📌 How it Works?

- Push each character of a string onto a stack.
- Pop characters one by one to get the reversed string.

### 🔷 Example

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

void reverseString(char str[]) {
    int n = strlen(str);
    char stack[n];
```

```
    int top = -1;

    for (int i = 0; i < n; i++) stack[++top] = str[i]; // Push

    for (int i = 0; i < n; i++) str[i] = stack[top--]; // Pop
}

int main() {
    char str[] = "hello";
    reverseString(str);
    printf("Reversed: %s\n", str);  // Output: "olleh"
    return 0;
}
```

📌 **Used In:** Reverse words, undo operations.

---

## 9️⃣ Expression Conversion (Infix to Postfix/Prefix)

### 📌 How it Works?

- Stack is used to convert **infix expressions** (`A + B * C`) into **postfix (`A B C * +`)**.
- Used by **compilers and interpreters**.

📌 **Used In:** Compilers, calculators.

---

## 🔷 Conclusion

Stacks are a fundamental data structure widely used in **memory management, algorithm design, and application development**. 🚀

## 📌 Polish Notation (Prefix and Postfix)

Polish notation is a way of writing mathematical expressions without the need for parentheses. It is commonly used in compilers, calculators, and expression evaluation algorithms.

There are **two types** of Polish notation:
1️⃣ **Prefix Notation (Polish Notation)**
2️⃣ **Postfix Notation (Reverse Polish Notation - RPN)**

---

## 1️⃣ Prefix Notation (Polish Notation)

- **Operator appears before the operands**.

- No need for parentheses to specify the order of operations.
- Evaluated from **right to left**.

### 🔷 Example

*Infix Expression:*

`(A + B) * C`

*Prefix (Polish) Notation:*

`* + A B C`

### 🔷 Evaluation

1. Start from the right: `* + A B C`
2. Evaluate `+ A B` first
3. Multiply result with `C`

---

## 2 Postfix Notation (Reverse Polish Notation - RPN)

- **Operator appears after the operands**.
- No need for parentheses.
- Evaluated from **left to right** using a **stack**.

### 🔷 Example

*Infix Expression:*

`(A + B) * C`

*Postfix (Reverse Polish) Notation:*

`A B + C *`

### 🔷 Evaluation Using Stack

| Step | Stack |
|---|---|
| Read `A` | `A` |
| Read `B` | `A B` |
| Read `+` (Apply operator) | `(A + B)` |
| Read `C` | `(A + B) C` |
| Read `*` (Apply operator) | `(A + B) * C` |

📌 **Final Result:** `(A + B) * C`

---

## 📌 Advantages of Polish Notation

✅ **No Parentheses Needed** – Operator precedence is inherently followed.
✅ **Efficient for Stack-Based Evaluations** – Used in compilers and calculators.
✅ **Easier for Computers to Parse** – No need to deal with operator precedence.

---

## 📌 Converting Infix to Postfix (Algorithm)

1. **If operand** → Add to output.
2. **If operator** → Push to stack (consider precedence).
3. **If '('** → Push to stack.
4. **If ')'** → Pop from stack until '(' is found.
5. **Pop remaining operators**.

---

## 📌 Application of Polish Notation

✅ **Compilers & Expression Parsing**
✅ **Calculators**
✅ **Stack-based Evaluations**
✅ **Used in Programming Languages (Lisp, Forth, PostScript)**

Would you like to see a **C program to evaluate postfix expressions**? 🚀

## 📌 Converting Infix to Postfix Notation

To convert an **infix expression** (where operators appear between operands, e.g., `A + B * C`) into a **postfix expression** (Reverse Polish Notation, RPN, e.g., `A B C * +`), we use a **stack-based algorithm**.

---

## 🔷 Steps to Convert Infix to Postfix

1️⃣ **If the character is an operand (A-Z, 0-9), add it to the output.**
2️⃣ *If the character is an operator (+, -, , /, ^), push it onto the stack*

- If the stack is not empty and the top of the stack has an operator of **higher or equal precedence**, pop and add it to the output.
- Then push the current operator onto the stack.

3️⃣ **If the character is '(' (left parenthesis), push it onto the stack.**

4️⃣ **If the character is ‘)’ (right parenthesis), pop from the stack and add to output until ‘(’ is found.**
5️⃣ **After scanning the entire infix expression, pop and add all remaining operators from the stack to the output.**

---

## 📌 Operator Precedence Table

| Operator | Associativity | Precedence |
|---|---|---|
| `^` | Right to Left | Highest (3) |
| `* / %` | Left to Right | Medium (2) |
| `+ -` | Left to Right | Lowest (1) |

---

## 🔷 Example

### Example 1

*Infix Expression:*

`A + B * C`

*Step-by-Step Conversion*

| Step | Symbol | Stack | Postfix Output |
|---|---|---|---|
| 1 | A | - | A |
| 2 | + | + | A |
| 3 | B | + | A B |
| 4 | * | + * | A B |
| 5 | C | + * | A B C |
| 6 | (End) | - | A B C * + |

✅ **Postfix Expression:** `A B C * +`

---

### Example 2

*Infix Expression:*

`(A + B) * C`

*Step-by-Step Conversion*

| Step | Symbol | Stack | Postfix Output |
|---|---|---|---|
| 1 | ( | ( | - |
| 2 | A | ( | A |
| 3 | + | (+ | A |
| 4 | B | (+ | A B |
| 5 | ) | | A B + |
| 6 | * | * | A B + |
| 7 | C | * | A B + C |
| 8 | (End) | | A B + C * |

✅ **Postfix Expression:** `A B + C *`

---

## 📌 C Program to Convert Infix to Postfix

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

#define MAX 100

// Stack structure
struct Stack {
    int top;
    char items[MAX];
};

// Function to initialize stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Function to check if stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to push an element onto stack
void push(struct Stack *s, char c) {
```

```
    if (s->top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    s->items[++(s->top)] = c;
}

// Function to pop an element from stack
char pop(struct Stack *s) {
    if (isEmpty(s)) {
        return '\0';
    }
    return s->items[(s->top)--];
}

// Function to return the top element of stack
char peek(struct Stack *s) {
    if (isEmpty(s)) {
        return '\0';
    }
    return s->items[s->top];
}

// Function to check precedence of operators
int precedence(char op) {
    if (op == '^') return 3;
    if (op == '*' || op == '/' || op == '%') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;
}

// Function to convert infix to postfix
void infixToPostfix(char *infix, char *postfix) {
    struct Stack s;
    initialize(&s);
    int i = 0, j = 0;

    while (infix[i] != '\0') {
        char token = infix[i];

        // If operand, add to output
        if (isalnum(token)) {
            postfix[j++] = token;
        }
        // If '(', push to stack
        else if (token == '(') {
            push(&s, token);
        }
        // If ')', pop until '(' is found
        else if (token == ')') {
            while (!isEmpty(&s) && peek(&s) != '(') {
                postfix[j++] = pop(&s);
            }
            pop(&s);  // Remove '(' from stack
        }
        // If operator, pop operators of higher precedence and push new
operator
```

```
        else {
            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(token))
{
                postfix[j++] = pop(&s);
            }
            push(&s, token);
        }
        i++;
    }

    // Pop all remaining operators
    while (!isEmpty(&s)) {
        postfix[j++] = pop(&s);
    }

    postfix[j] = '\0';  // Null terminate the postfix expression
}

// Main function
int main() {
    char infix[MAX], postfix[MAX];

    printf("Enter an infix expression: ");
    scanf("%s", infix);

    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

---

## 📌 Output Example

### Input:

`(A+B)*C`

### Output:

`Postfix expression: AB+C*`

---

## 📌 Summary

✓ **Infix Notation:** Operators between operands (e.g., `A + B * C`).
✓ **Postfix Notation:** Operators after operands (e.g., `A B C * +`).
✓ **Algorithm:** Uses **stacks** to handle operator precedence and associativity.
✓ **Implementation:** Efficient in **compilers, calculators, and expression evaluations**.

Would you like a program to **evaluate a postfix expression** as well? 🚀

## 📌 Evaluation of Postfix (Reverse Polish) Notation

Postfix notation (Reverse Polish Notation - RPN) is evaluated using a **stack-based algorithm**. It eliminates the need for parentheses and follows a simple left-to-right evaluation process.

---

## 🔷 Steps to Evaluate a Postfix Expression

1. **Scan the expression from left to right.**
2. **If the character is an operand (number), push it onto the stack.**
3. ***If the character is an operator (+, -, , /, ^), pop two operands from the stack.***
4. **Apply the operator on the two popped operands.**
5. **Push the result back onto the stack.**
6. **After scanning the entire expression, the final result is on top of the stack.**

---

## 📌 Example 1

### Postfix Expression:

```
5 3 2 * + 8 -
```

### Step-by-Step Evaluation

| Step | Symbol | Stack (Top → Bottom) |
|---|---|---|
| 1 | 5 | 5 |
| 2 | 3 | 3, 5 |
| 3 | 2 | 2, 3, 5 |
| 4 | * | (3 * 2) → 6, 5 |
| 5 | + | (5 + 6) → 11 |
| 6 | 8 | 8, 11 |
| 7 | - | (11 - 8) → 3 |

✅ **Final Answer:** `3`

---

## 📌 C Program to Evaluate a Postfix Expression

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

#define MAX 100

// Stack structure
struct Stack {
    int top;
    double items[MAX];
};

// Function to initialize stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Function to check if stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to push an element onto stack
void push(struct Stack *s, double value) {
    if (s->top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    s->items[++(s->top)] = value;
}

// Function to pop an element from stack
double pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return s->items[(s->top)--];
}

// Function to evaluate postfix expression
double evaluatePostfix(char* postfix) {
    struct Stack s;
    initialize(&s);

    for (int i = 0; postfix[i] != '\0'; i++) {
        char token = postfix[i];

        // If operand, push it to stack
        if (isdigit(token)) {
            push(&s, token - '0');  // Convert char to integer
        }
        // If operator, pop two elements and apply operation
        else {
```

```
            double val2 = pop(&s);
            double val1 = pop(&s);

            switch (token) {
                case '+': push(&s, val1 + val2); break;
                case '-': push(&s, val1 - val2); break;
                case '*': push(&s, val1 * val2); break;
                case '/': push(&s, val1 / val2); break;
                case '^': push(&s, pow(val1, val2)); break;
                default: printf("Invalid operator\n"); exit(1);
            }
        }
    }

    // Final result
    return pop(&s);
}

// Main function
int main() {
    char postfix[MAX];

    printf("Enter a postfix expression (single-digit operands only): ");
    scanf("%s", postfix);

    double result = evaluatePostfix(postfix);
    printf("Result: %.2f\n", result);

    return 0;
}
```

---

## 📌 Example Run

### Input:

```
5 3 2 * + 8 -
```

### Output:

```
Result: 3.00
```

---

## 📌 Summary

✓ **Postfix Notation** eliminates parentheses and follows a simple stack-based evaluation.
✓ **Algorithm**: Push operands onto the stack, pop two operands for each operator, compute the result, and push it back.
✓ **Applications**: Used in compilers, calculators, and expression evaluation.

Would you like a **program to convert infix to postfix and then evaluate it?** 🚀

## Tower of Hanoi Problem

The **Tower of Hanoi** is a mathematical puzzle involving three rods and multiple disks of different sizes. The objective is to move all the disks from the **source rod** to the **destination rod**, following these rules:

1. **Only one disk can be moved at a time.**
2. **A larger disk cannot be placed on a smaller disk.**
3. **You can only use the auxiliary rod as a temporary storage.**

---

## 📌 Recursive Solution

The problem follows a recursive approach:

- Move **(n-1) disks** from **source** to **auxiliary** using the **destination** as a helper.
- Move the **nth (largest) disk** from **source** to **destination**.
- Move **(n-1) disks** from **auxiliary** to **destination** using the **source** as a helper.

The **base case** occurs when there is only one disk to move.

---

## 📌 Recursive Algorithm

```
TOWER(n, source, destination, auxiliary):
    If n == 1:
        Move disk from source to destination
        Return
    TOWER(n-1, source, auxiliary, destination)  # Move (n-1) disks to
auxiliary
    Move disk from source to destination
    TOWER(n-1, auxiliary, destination, source)  # Move (n-1) disks to
destination
```

---

## 📌 C Program for Tower of Hanoi

```
#include <stdio.h>

// Function to implement Tower of Hanoi
void towerOfHanoi(int n, char source, char destination, char auxiliary) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }
    towerOfHanoi(n - 1, source, auxiliary, destination);
    printf("Move disk %d from %c to %c\n", n, source, destination);
    towerOfHanoi(n - 1, auxiliary, destination, source);
}
```

```
// Main function
int main() {
    int n; // Number of disks
    printf("Enter the number of disks: ");
    scanf("%d", &n);
    printf("Steps to solve Tower of Hanoi:\n");
    towerOfHanoi(n, 'A', 'C', 'B'); // A = Source, C = Destination, B =
Auxiliary
    return 0;
}
```

---

## 📌 Example Run

### Input:

```
Enter the number of disks: 3
```

### Output:

```
Steps to solve Tower of Hanoi:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

---

## 📌 Time Complexity

The Tower of Hanoi follows the recurrence relation:

$$T(n)=2T(n−1)+1T(n) = 2T(n-1) + 1$$

Solving this gives the time complexity:

$$T(n)=O(2n)T(n) = O(2^n)$$

This exponential complexity makes it impractical for large values of `n`.

---

## 📌 Applications of Tower of Hanoi

✓ **Data Structure Concept** - Stack-based recursion.
✓ **Algorithm Design** - Helps in understanding recursion and divide & conquer techniques.
✓ **Real-World Analogy** - Used in problems related to disk swapping, robotics, and CPU scheduling.

## 📌 Recursion in C

### What is Recursion?

Recursion is a technique in which a function calls itself to solve a problem. It is used to break down complex problems into smaller, more manageable subproblems.

---

## 📌 Key Features of Recursion

✅ **Base Case** – A condition where the recursion stops.
✅ **Recursive Case** – The function calls itself with a smaller problem.

---

## 📌 Example 1: Factorial using Recursion

Factorial of a number nn is defined as:

n!=n×(n−1)×(n−2)×...×1n! = n \times (n-1) \times (n-2) \times ... \times 1

### C Code:

```
#include <stdio.h>

// Recursive function to calculate factorial
long long factorial(int n) {
    if (n == 0)  // Base case
        return 1;
    return n * factorial(n - 1);  // Recursive case
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Factorial of %d is %lld\n", num, factorial(num));
    return 0;
}
```

### Example Output:

```
Enter a number: 5
Factorial of 5 is 120
```

---

## 📌 Example 2: Fibonacci Series using Recursion

The Fibonacci series is:

0,1,1,2,3,5,8,13,...0, 1, 1, 2, 3, 5, 8, 13, ...

Each term is the sum of the previous two terms:

F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)

### C Code:

```
#include <stdio.h>

// Recursive function to find nth Fibonacci number
int fibonacci(int n) {
    if (n == 0) return 0; // Base case
    if (n == 1) return 1; // Base case
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);

    printf("Fibonacci Series: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```

### Example Output:

```
Enter the number of terms: 7
Fibonacci Series: 0 1 1 2 3 5 8
```

---

## 📌 Types of Recursion

1. **Direct Recursion** – A function calls itself directly.
2. **Indirect Recursion** – Two or more functions call each other.

### Example of Indirect Recursion

```
#include <stdio.h>

void functionA(int n);
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        printf("%d ", n);
        functionB(n - 1);
    }
}

void functionB(int n) {
    if (n > 0) {
```

```
        printf("%d ", n);
        functionA(n - 2);
    }
}

int main() {
    functionA(5);
    return 0;
}
```

Output:

```
5 4 2 1
```

---

## 📌 Recursion vs Iteration

| Feature | Recursion | Iteration |
|---|---|---|
| **Speed** | Slower due to function calls | Faster |
| **Memory Usage** | Uses more memory (stack) | Uses less memory |
| **Code Simplicity** | More elegant for complex problems | Requires loops and extra variables |

---

## 📌 When to Use Recursion?

✓ **When the problem can be broken into smaller subproblems** (e.g., Fibonacci, Factorial).
✓ **When using Divide & Conquer techniques** (e.g., Merge Sort, Quick Sort).
✓ **For tree and graph traversal** (e.g., DFS, Binary Search Tree).

---

## 📌 Common Recursion Problems

✓ **Tower of Hanoi**
✓ **Binary Search**
✓ **Palindrome Checking**
✓ **Graph Traversals (DFS)**

Would you like a **detailed explanation of recursion in sorting algorithms** like **Merge Sort and Quick Sort**? 🚀

# Recursion vs Iteration: Concept and Comparison

## 📌 Concept of Recursion

**Recursion** is a process where a function **calls itself** to solve a problem by breaking it into smaller subproblems. Each recursive call reduces the problem size until a **base case** is reached.

### Example: Factorial using Recursion

Factorial of a number nn is:

n!=n×(n−1)×(n−2)×...×1n! = n \times (n-1) \times (n-2) \times ... \times 1

*C Code for Recursive Factorial*

```
#include <stdio.h>

// Recursive function to calculate factorial
long long factorial(int n) {
    if (n == 0)  // Base case
        return 1;
    return n * factorial(n - 1);  // Recursive case
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Factorial of %d is %lld\n", num, factorial(num));
    return 0;
}
```

### Output:

```
Enter a number: 5
Factorial of 5 is 120
```

---

## 📌 Concept of Iteration

**Iteration** is a process where a loop (like `for`, `while`, or `do-while`) **repeats a block of code** until a certain condition is met. It avoids function calls and is generally more efficient in terms of memory and execution speed.

### Example: Factorial using Iteration

```
#include <stdio.h>

// Iterative function to calculate factorial
long long factorial(int n) {
    long long fact = 1;
    for (int i = 1; i <= n; i++) {
        fact *= i;
    }
```

```
    return fact;
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printf("Factorial of %d is %lld\n", num, factorial(num));
    return 0;
}
```

### Output:

```
Enter a number: 5
Factorial of 5 is 120
```

---

## 📌 Comparison Between Recursion and Iteration

| Feature | Recursion | Iteration |
|---|---|---|
| **Definition** | Function calls itself to solve smaller subproblems. | Uses loops (`for`, `while`, etc.) to repeat a task. |
| **Termination** | Ends when the base case is met. | Ends when the loop condition becomes false. |
| **Memory Usage** | Requires extra memory (stack) for each function call. | Uses less memory (only loop control variables). |
| **Execution Speed** | Slower due to function calls and stack overhead. | Faster as it avoids function call overhead. |
| **Code Simplicity** | Simple and elegant for problems like DFS, factorial, Fibonacci, etc. | More efficient for simple repetitive tasks like summation. |
| **Use Case** | Suitable for problems that are naturally recursive (tree traversal, divide & conquer algorithms). | Suitable for tasks requiring repetitive execution with fixed steps. |

---

## 📌 When to Use Recursion vs Iteration?

**✓ Use Recursion when:**

✅ The problem can be divided into smaller subproblems (e.g., **Merge Sort, Tower of Hanoi, DFS**).

✅ The problem is easier to express in a recursive way (e.g., **Fibonacci, Factorial**).

✓ **Use Iteration when:**
✅ Memory efficiency is important (e.g., **Looping over an array**).
✅ The problem does not naturally break down into smaller subproblems.

---

## 📌 Example: Fibonacci Series

**Using Recursion:**

```
#include <stdio.h>

int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);

    printf("Fibonacci Series: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```

✓ **Time Complexity:** O(2n)O(2^n) (Exponential, very slow!)

**Using Iteration:**

```
#include <stdio.h>

void fibonacci(int n) {
    int a = 0, b = 1, temp;
    printf("Fibonacci Series: %d %d ", a, b);
    for (int i = 2; i < n; i++) {
        temp = a + b;
        printf("%d ", temp);
        a = b;
        b = temp;
    }
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    fibonacci(n);
```

```
    return 0;
}
```

✓ **Time Complexity:** O(n)O(n) (Much faster than recursion!)

---

## 📌 Conclusion

- **Recursion** makes code simpler but can be inefficient due to function calls and memory usage.
- **Iteration** is efficient but may require additional variables and careful loop management.
- Choose **recursion** for **divide & conquer problems** and **iteration** for **simple loops**.

Would you like more **examples or detailed explanations** of recursion in sorting algorithms like **Merge Sort or Quick Sort**? 🚀

# 📌 Queues in Data Structures

## 1️⃣ What is a Queue?

A **queue** is a linear **First In, First Out (FIFO)** data structure, meaning the element added **first** is the **first** to be removed.

### Example of FIFO Behavior:

Imagine a queue at a ticket counter:

1. The first person to arrive gets served first.
2. The last person to arrive has to wait until others are served.

---

## 2️⃣ Basic Queue Operations

| Operation | Description |
|---|---|
| **Enqueue** | Adds an element to the **rear** of the queue. |
| **Dequeue** | Removes an element from the **front** of the queue. |
| **Front (Peek)** | Returns the front element without removing it. |
| **isEmpty** | Checks if the queue is empty. |
| **isFull** | Checks if the queue is full (for a fixed-size array implementation). |

---

## 3 Representation of Queues in C

Queues can be implemented using **Arrays** and **Linked Lists**.

### Queue Implementation Using Arrays

```
#include <stdio.h>
#define SIZE 5  // Define maximum size of Queue

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

// Function to check if the queue is full
int isFull() {
    return rear == SIZE - 1;
}

// Function to check if the queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to add an element to the queue (Enqueue)
void enqueue(int value) {
    if (isFull()) {
        printf("Queue is Full!\n");
        return;
    }
    if (front == -1) front = 0;
    queue[++rear] = value;
    printf("%d enqueued to queue\n", value);
}

// Function to remove an element from the queue (Dequeue)
void dequeue() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("%d dequeued from queue\n", queue[front++]);
}

// Function to display the queue
void display() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
```

```
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

### Example Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

## 📌 Queue Implementation Using Linked List

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

// Structure for a queue node
struct Node {
    int data;
    struct Node* next;
};

struct Node *front = NULL, *rear = NULL;

// Function to enqueue an element
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
}

// Function to dequeue an element
void dequeue() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    front = front->next;

    if (front == NULL)
        rear = NULL;

    printf("%d dequeued from queue\n", temp->data);
```

```
    free(temp);
}

// Function to display the queue
void display() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

---

### 4️⃣ Types of Queues

1. **Simple Queue (FIFO Queue)** – Basic queue where insertion is done at the **rear** and deletion at the **front**.
2. **Circular Queue** – The last position is connected to the first, overcoming the limitation of a fixed-size array.
3. **Deque (Double-Ended Queue)** – Insertion and deletion can occur at both ends.
4. **Priority Queue** – Elements are removed based on priority rather than order of arrival.

Would you like implementations of **Circular Queue or Priority Queue** as well? 🚀

## 📌 Introduction to Queue ADT

### 1️⃣ What is a Queue ADT?

A **Queue** is an **Abstract Data Type (ADT)** that follows the **First In, First Out (FIFO)** principle. This means that elements are added from the **rear** and removed from the **front**.

#### 📌 Real-Life Example of a Queue:

- A **line of people at a ticket counter**:

- The first person in the queue is served first.
- The last person has to wait until everyone before them is served.

---

## 2 Queue ADT Operations

A Queue ADT supports the following operations:

| Operation | Description |
|---|---|
| **Enqueue(x)** | Inserts an element `x` at the **rear** of the queue. |
| **Dequeue()** | Removes an element from the **front** of the queue. |
| **Front()** | Returns the element at the front without removing it. |
| **isEmpty()** | Checks if the queue is empty. |
| **isFull()** | Checks if the queue is full (for a fixed-size implementation). |

---

## 3 Implementation of Queue ADT

Queues can be implemented using:

✅ **Arrays**
✅ **Linked Lists**

---

### 📌 Queue ADT Implementation Using Arrays

```
#include <stdio.h>
#define SIZE 5  // Define maximum queue size

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

// Function to check if queue is full
int isFull() {
    return rear == SIZE - 1;
}

// Function to check if queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to add an element to the queue (Enqueue)
void enqueue(int value) {
```

```
    if (isFull()) {
        printf("Queue is Full!\n");
        return;
    }
    if (front == -1) front = 0;
    queue[++rear] = value;
    printf("%d enqueued to queue\n", value);
}

// Function to remove an element from the queue (Dequeue)
void dequeue() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("%d dequeued from queue\n", queue[front++]);
}

// Function to display queue elements
void display() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}

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

### 📌 Example Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

### 📌 Queue ADT Implementation Using Linked List

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

// Structure for a queue node
struct Node {
    int data;
```

```
    struct Node* next;
};

struct Node *front = NULL, *rear = NULL;

// Function to enqueue an element
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
}

// Function to dequeue an element
void dequeue() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    front = front->next;

    if (front == NULL)
        rear = NULL;

    printf("%d dequeued from queue\n", temp->data);
    free(temp);
}

// Function to display queue elements
void display() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
```

```
    display();
    return 0;
}
```

---

### 4 Advantages of Queue ADT

✓ **FIFO Order:** Ensures elements are processed in the order they arrive.
✓ **Efficient Operations:** `O(1)` time complexity for **Enqueue** and **Dequeue** operations.
✓ **Used in Many Applications:** CPU scheduling, Printer Queue, Network Buffers, etc.

Would you like explanations on **Circular Queue or Priority Queue** as well? 🚀

## 📌 Representation of Queues Using Arrays and Linked Lists

### 1 What is a Queue?

A **queue** is a linear **First In, First Out (FIFO)** data structure, meaning the element added **first** is the **first** to be removed.

---

### 2 Representation of Queue Using an Array

#### 📌 Implementation of Queue Using Array

- The queue is stored in a **fixed-size** array.
- Two variables, `front` and `rear`, are used to track the first and last elements in the queue.
- **Enqueue:** Insert an element at the `rear` of the queue.
- **Dequeue:** Remove an element from the `front` of the queue.

#### 📌 Array Implementation in C

```
#include <stdio.h>
#define SIZE 5  // Define maximum queue size

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

// Function to check if queue is full
int isFull() {
    return rear == SIZE - 1;
}

// Function to check if queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to add an element to the queue (Enqueue)
void enqueue(int value) {
```

```
    if (isFull()) {
        printf("Queue is Full!\n");
        return;
    }
    if (front == -1) front = 0;
    queue[++rear] = value;
    printf("%d enqueued to queue\n", value);
}

// Function to remove an element from the queue (Dequeue)
void dequeue() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("%d dequeued from queue\n", queue[front++]);
}

// Function to display queue elements
void display() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}

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

### 📌 Example Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

## 3️⃣ Representation of Queue Using Linked List

A **linked list** implementation of a queue is more dynamic and does not require a fixed-size array.

## 📌 Features of Linked List Representation

- **Nodes** store the data and a pointer to the next element.
- `front` points to the **first** element.
- `rear` points to the **last** element.

## 📌 Linked List Implementation in C

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

// Structure for a queue node
struct Node {
    int data;
    struct Node* next;
};

struct Node *front = NULL, *rear = NULL;

// Function to enqueue an element
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
}

// Function to dequeue an element
void dequeue() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    front = front->next;

    if (front == NULL)
        rear = NULL;

    printf("%d dequeued from queue\n", temp->data);
    free(temp);
}

// Function to display queue elements
void display() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
```

```
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

---

## 4️⃣ Comparison of Queue Using Array vs. Linked List

| Feature | Array-Based Queue | Linked List-Based Queue |
|---|---|---|
| **Memory** | Requires a fixed-size array | Uses dynamic memory allocation |
| **Insertion (Enqueue)** | `O(1)` if space available, else `O(n)` for resizing | `O(1)` |
| **Deletion (Dequeue)** | `O(1)` but shifting required in some cases | `O(1)` |
| **Flexibility** | Limited by predefined size | Dynamically grows/shrinks |
| **Implementation Complexity** | Simple | Requires extra pointer handling |

Would you like me to explain **Circular Queue** or **Priority Queue** next? 🚀

## 📌 Implementation of Queues in C

Queues can be implemented using **Arrays** and **Linked Lists**. Below, I'll provide implementations for both.

---

## 1️⃣ Queue Implementation Using Array

### ✅ Steps:

1. Use an array to store queue elements.
2. Maintain `front` and `rear` pointers.
3. Perform **enqueue**, **dequeue**, and **display** operations.

### 📌 C Program for Queue Using Array

```
#include <stdio.h>
#define SIZE 5  // Maximum size of the queue

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

// Function to check if the queue is full
int isFull() {
    return rear == SIZE - 1;
}

// Function to check if the queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to insert an element (enqueue)
void enqueue(int value) {
    if (isFull()) {
        printf("Queue is Full!\n");
        return;
    }
    if (front == -1) front = 0; // Initialize front if queue was empty
    queue[++rear] = value;
    printf("%d enqueued to queue\n", value);
}

// Function to remove an element (dequeue)
void dequeue() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("%d dequeued from queue\n", queue[front++]);
}

// Function to display the queue elements
void display() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}
```

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

### 📌 Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

## 2 Queue Implementation Using Linked List

### ✅ Steps:

1. Use a **linked list** with **front** and **rear** pointers.
2. Dynamically allocate memory for queue elements.
3. Perform **enqueue**, **dequeue**, and **display** operations.

### 📌 C Program for Queue Using Linked List

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

// Structure for queue node
struct Node {
    int data;
    struct Node* next;
};

struct Node *front = NULL, *rear = NULL;

// Function to enqueue an element
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
    printf("%d enqueued to queue\n", value);
}
```

```
// Function to dequeue an element
void dequeue() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    front = front->next;

    if (front == NULL)
        rear = NULL;

    printf("%d dequeued from queue\n", temp->data);
    free(temp);
}

// Function to display queue elements
void display() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

### 📌 Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

## 3️⃣ Comparison: Array vs Linked List Implementation

| Feature | Array-Based Queue | Linked List-Based Queue |
|---|---|---|

| Feature | Array-Based Queue | Linked List-Based Queue |
|---|---|---|
| **Memory** | Fixed size | Dynamic memory allocation |
| **Insertion (Enqueue)** | `O(1)`, but resizing may take `O(n)` | `O(1)` |
| **Deletion (Dequeue)** | `O(1)`, but shifting may be needed | `O(1)` |
| **Flexibility** | Limited by array size | No size limitation |
| **Implementation Complexity** | Easier | Requires pointer management |

---

## 4 Special Types of Queues

1. **Circular Queue** (Optimized array-based queue)
2. **Priority Queue** (Elements have priorities)
3. **Deque (Double-Ended Queue)** (Insertion & deletion from both ends)

Would you like me to explain **Circular Queue** next? 🚀

## 📌 Implementation of Queues in C

Queues can be implemented using **Arrays** and **Linked Lists**. Below, I'll provide implementations for both.

---

## 1 Queue Implementation Using Array

### ✅ Steps:

1. Use an array to store queue elements.
2. Maintain `front` and `rear` pointers.
3. Perform **enqueue**, **dequeue**, and **display** operations.

### 📌 C Program for Queue Using Array

```
#include <stdio.h>
#define SIZE 5  // Maximum size of the queue

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

// Function to check if the queue is full
int isFull() {
    return rear == SIZE - 1;
}
```

```
// Function to check if the queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Function to insert an element (enqueue)
void enqueue(int value) {
    if (isFull()) {
        printf("Queue is Full!\n");
        return;
    }
    if (front == -1) front = 0; // Initialize front if queue was empty
    queue[++rear] = value;
    printf("%d enqueued to queue\n", value);
}

// Function to remove an element (dequeue)
void dequeue() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("%d dequeued from queue\n", queue[front++]);
}

// Function to display the queue elements
void display() {
    if (isEmpty()) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}

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

### 📌 Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

## 2 Queue Implementation Using Linked List

### ✅ Steps:

1. Use a **linked list** with **front** and **rear** pointers.
2. Dynamically allocate memory for queue elements.
3. Perform **enqueue**, **dequeue**, and **display** operations.

### 📌 C Program for Queue Using Linked List

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

// Structure for queue node
struct Node {
    int data;
    struct Node* next;
};

struct Node *front = NULL, *rear = NULL;

// Function to enqueue an element
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
        front = rear = newNode;
        return;
    }

    rear->next = newNode;
    rear = newNode;
    printf("%d enqueued to queue\n", value);
}

// Function to dequeue an element
void dequeue() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    front = front->next;

    if (front == NULL)
        rear = NULL;

    printf("%d dequeued from queue\n", temp->data);
    free(temp);
}

// Function to display queue elements
void display() {
    if (front == NULL) {
```

```
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

### 📌 Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
Queue elements: 10 20 30
10 dequeued from queue
Queue elements: 20 30
```

---

## 3️⃣ Comparison: Array vs Linked List Implementation

| Feature | Array-Based Queue | Linked List-Based Queue |
|---|---|---|
| **Memory** | Fixed size | Dynamic memory allocation |
| **Insertion (Enqueue)** | `O(1)`, but resizing may take `O(n)` | `O(1)` |
| **Deletion (Dequeue)** | `O(1)`, but shifting may be needed | `O(1)` |
| **Flexibility** | Limited by array size | No size limitation |
| **Implementation Complexity** | Easier | Requires pointer management |

---

## 4️⃣ Special Types of Queues

1. **Circular Queue** (Optimized array-based queue)
2. **Priority Queue** (Elements have priorities)
3. **Deque (Double-Ended Queue)** (Insertion & deletion from both ends)

Would you like me to explain **Circular Queue** next? 🚀

# 📌 Applications and Types of Queues in Data Structures

---

## 1️⃣ What is a Queue?

A **queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. Elements are added from the **rear** and removed from the **front**.

---

## 2️⃣ Applications of Queues

Queues are widely used in **computer science** and **real-world applications**:

### 📌 Computer Science Applications

✅ **Process Scheduling in Operating Systems** – CPU scheduling uses a queue to manage processes (e.g., **Round Robin Scheduling**).
✅ **Disk Scheduling** – Managing read/write requests in a hard drive.
✅ **Network Packet Processing** – Routers use queues to manage data packets.
✅ **I/O Buffers** – Print queue, keyboard buffer, etc.
✅ **Tree and Graph Traversals** – BFS (Breadth-First Search) uses queues.
✅ **Job Scheduling** – Task execution in batch processing systems.

### 📌 Real-World Applications

✅ **Call Center Management** – Calls are queued and answered in order.
✅ **Ticket Booking Systems** – Movie tickets, railway tickets, etc.
✅ **Banking System** – Customers wait in line at ATMs or customer service counters.
✅ **Waiting Lists** – Hospitals, airports, and other services use queues.

---

## 3️⃣ Types of Queues

Queues come in different variations based on how elements are inserted and removed.

## 📌 1. Simple Queue (Linear Queue)

✅ **Follows FIFO** – Elements enter from the rear and leave from the front.
✅ **Limitation** – Wasted space if elements are dequeued.
✅ **Example:** Normal ticket booking queue.

**Example:**

```
Enqueue(10) -> Enqueue(20) -> Enqueue(30)
Dequeue() -> Removes 10
Queue: 20 -> 30
```

---

## 📌 2. Circular Queue

✅ **Overcomes the limitation of a simple queue.**
✅ **Front and Rear wrap around** when they reach the end of the array.
✅ **More efficient memory utilization.**
✅ **Used in:** CPU scheduling, memory management.

**Example:**

```
Enqueue(10) -> Enqueue(20) -> Enqueue(30) -> Dequeue() (Removes 10)
New element is added at the empty spot (instead of shifting all elements).
```

---

## 📌 3. Double-Ended Queue (Deque)

✅ **Insertion and deletion are allowed at both ends** (Front & Rear).
✅ Two types:

- **Input-Restricted Deque** – Insertion at only one end, deletion at both.
- **Output-Restricted Deque** – Deletion at only one end, insertion at both.
  ✅ **Used in:** Undo operations, scheduling algorithms.

---

## 📌 4. Priority Queue

✅ **Elements are dequeued based on priority instead of FIFO.**
✅ **Higher priority elements are removed first.**
✅ Implemented using **Heaps or Linked Lists**.
✅ **Used in:**

- **CPU Scheduling**

- **Dijkstra's Algorithm (Shortest Path Algorithm)**
- *A Search Algorithm (AI & Robotics)**

**Example:**

```
Enqueue(5, Priority=2) -> Enqueue(10, Priority=1) -> Enqueue(15, Priority=3)
Dequeue() -> Removes 10 (Highest priority)
```

---

## 4️⃣ Comparison of Queue Types

| Queue Type | Insertion | Deletion | Special Feature |
|---|---|---|---|
| **Simple Queue** | Rear | Front | FIFO |
| **Circular Queue** | Rear | Front | Uses memory efficiently |
| **Deque** | Both Ends | Both Ends | Flexible insertion/deletion |
| **Priority Queue** | Based on priority | Based on priority | Elements processed by priority |

---

Would you like an implementation of **Circular Queue or Priority Queue** in C? 🚀

# 📌 Double-Ended Queue (Deque) in Data Structures

## 1️⃣ What is a Deque?

A **Deque (Double-Ended Queue)** is a linear data structure that allows insertion and deletion from **both ends (front and rear)**.

Unlike a simple queue (FIFO) or stack (LIFO), a deque provides **greater flexibility** in data handling.

---

## 2️⃣ Types of Deques

There are two types of deques:

### 📌 1. Input-Restricted Deque

✅ **Insertion allowed only at one end** (rear).
✅ **Deletion can happen at both ends** (front and rear).

### 📌 2. Output-Restricted Deque

✅ **Deletion allowed only at one end** (front).
✅ **Insertion can happen at both ends** (front and rear).

---

## 3️⃣ Operations on a Deque

A deque supports the following operations:

| Operation | Description |
|---|---|
| `insertFront()` | Insert an element at the front |
| `insertRear()` | Insert an element at the rear |
| `deleteFront()` | Remove an element from the front |
| `deleteRear()` | Remove an element from the rear |
| `isEmpty()` | Check if the deque is empty |
| `isFull()` | Check if the deque is full |
| `getFront()` | Get the front element |
| `getRear()` | Get the rear element |

---

## 4️⃣ Applications of Deques

✅ **Task Scheduling** – OS process scheduling, Disk scheduling.
✅ **Undo/Redo Functionality** – Used in text editors, IDEs.
✅ **Sliding Window Problems** – Used in problems like **maximum/minimum in a window**.
✅ **Palindrome Checking** – Used to check if a string is a palindrome.
✅ **Browser History** – Moving back and forth between pages.

---

## 5 Implementation of Deque in C

Deques can be implemented using **Arrays** or **Doubly Linked Lists**.
Below is an **Array-based Deque implementation**.

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

#define SIZE 5  // Maximum size of the deque

int deque[SIZE];
int front = -1, rear = -1;

// Check if deque is empty
int isEmpty() {
    return (front == -1);
}

// Check if deque is full
int isFull() {
    return ((front == 0 && rear == SIZE - 1) || (front == rear + 1));
}

// Insert an element at the front
void insertFront(int value) {
    if (isFull()) {
        printf("Deque is Full!\n");
        return;
    }
    if (front == -1) {  // First element
        front = rear = 0;
    } else if (front == 0) {
        front = SIZE - 1;  // Wrap around
    } else {
        front--;
    }
    deque[front] = value;
    printf("%d inserted at front\n", value);
}

// Insert an element at the rear
void insertRear(int value) {
    if (isFull()) {
        printf("Deque is Full!\n");
        return;
    }
    if (rear == -1) {  // First element
        front = rear = 0;
    } else if (rear == SIZE - 1) {
        rear = 0;  // Wrap around
    } else {
        rear++;
    }
    deque[rear] = value;
    printf("%d inserted at rear\n", value);
```

```
}

// Delete an element from the front
void deleteFront() {
    if (isEmpty()) {
        printf("Deque is Empty!\n");
        return;
    }
    printf("%d deleted from front\n", deque[front]);
    if (front == rear) {  // Only one element left
        front = rear = -1;
    } else if (front == SIZE - 1) {
        front = 0;  // Wrap around
    } else {
        front++;
    }
}

// Delete an element from the rear
void deleteRear() {
    if (isEmpty()) {
        printf("Deque is Empty!\n");
        return;
    }
    printf("%d deleted from rear\n", deque[rear]);
    if (front == rear) {  // Only one element left
        front = rear = -1;
    } else if (rear == 0) {
        rear = SIZE - 1;  // Wrap around
    } else {
        rear--;
    }
}

// Display elements in the deque
void display() {
    if (isEmpty()) {
        printf("Deque is Empty!\n");
        return;
    }
    printf("Deque elements: ");
    int i = front;
    while (1) {
        printf("%d ", deque[i]);
        if (i == rear)
            break;
        i = (i + 1) % SIZE;  // Wrap around
    }
    printf("\n");
}

int main() {
    insertRear(10);
    insertRear(20);
    insertFront(5);
    display();
    deleteFront();
```

```
    display();
    insertFront(15);
    deleteRear();
    display();
    return 0;
}
```

---

## 6 Output of the Program

```
10 inserted at rear
20 inserted at rear
5 inserted at front
Deque elements: 5 10 20
5 deleted from front
Deque elements: 10 20
15 inserted at front
20 deleted from rear
Deque elements: 15 10
```

---

## 7 Comparison: Deque vs Other Queues

| Feature | Deque | Queue | Stack |
|---|---|---|---|
| Insertion | Front & Rear | Rear only | Top only |
| Deletion | Front & Rear | Front only | Top only |
| Access Order | No fixed order | FIFO | LIFO |
| Flexibility | High | Medium | Low |

---

## 8 Summary

✓ **Deque (Double-Ended Queue)** allows **insertion & deletion from both ends**.
✓ Used in **task scheduling, undo/redo, palindrome checking, browser history**.
✓ Can be implemented using **arrays or linked lists**.
✓ Provides more **flexibility** than a normal queue.

Would you like me to implement **Deque using Linked List**? 🚀

# 📌 Priority Queue in Data Structures

## 1️⃣ What is a Priority Queue?

A **Priority Queue** is a special type of queue where each element has a **priority level**, and elements are **dequeued based on priority** rather than the FIFO (First In, First Out) principle.

✅ **Higher priority elements are dequeued first.**
✅ If two elements have the same priority, they follow **FIFO order**.

---

## 2️⃣ Types of Priority Queues

### 📌 1. Max Priority Queue

- The **element with the highest priority** is removed first.
- Example: **CPU Scheduling**, where higher-priority tasks execute first.

### 📌 2. Min Priority Queue

- The **element with the lowest priority** is removed first.
- Example: **Dijkstra's Algorithm**, where the shortest path is found first.

---

## 3️⃣ Applications of Priority Queues

✅ **CPU Scheduling** – Higher priority tasks execute first.
✅ **Graph Algorithms** – Dijkstra's shortest path, Prim's algorithm.
✅ **Event Scheduling** – Used in simulation systems.
✅ **Data Compression** – Huffman coding.
✅ **Networking** – Routers prioritize data packets based on importance.

---

## 4️⃣ Implementation of Priority Queue in C (Using Arrays)

We store elements in a **sorted order** to ensure **efficient dequeuing**.

```
#include <stdio.h>

#define SIZE 5  // Max size of priority queue

struct PriorityQueue {
    int data;
    int priority;
```

```
};

struct PriorityQueue pq[SIZE];
int rear = -1;

// Function to check if queue is empty
int isEmpty() {
    return (rear == -1);
}

// Function to check if queue is full
int isFull() {
    return (rear == SIZE - 1);
}

// Function to insert an element in priority order
void insert(int value, int priority) {
    if (isFull()) {
        printf("Priority Queue is Full!\n");
        return;
    }

    int i;
    for (i = rear; i >= 0 && pq[i].priority > priority; i--) {
        pq[i + 1] = pq[i];  // Shift elements to the right
    }

    pq[i + 1].data = value;
    pq[i + 1].priority = priority;
    rear++;
    printf("Inserted: %d with priority: %d\n", value, priority);
}

// Function to remove the highest priority element
void delete() {
    if (isEmpty()) {
        printf("Priority Queue is Empty!\n");
        return;
    }
    printf("Removed: %d with priority: %d\n", pq[0].data, pq[0].priority);

    // Shift elements to the left
    for (int i = 0; i < rear; i++) {
        pq[i] = pq[i + 1];
    }
    rear--;
}

// Function to display the priority queue
void display() {
    if (isEmpty()) {
        printf("Priority Queue is Empty!\n");
        return;
    }

    printf("Priority Queue: ");
    for (int i = 0; i <= rear; i++) {
```

```
        printf("[%d, %d] ", pq[i].data, pq[i].priority);
    }
    printf("\n");
}

// Main function
int main() {
    insert(10, 2);
    insert(20, 1);
    insert(30, 3);
    display();
    delete();
    display();
    return 0;
}
```

---

## 5 Output of the Program

```
Inserted: 10 with priority: 2
Inserted: 20 with priority: 1
Inserted: 30 with priority: 3
Priority Queue: [20, 1] [10, 2] [30, 3]
Removed: 20 with priority: 1
Priority Queue: [10, 2] [30, 3]
```

---

## 6 Comparison: Priority Queue vs Normal Queue

| Feature | Priority Queue | Normal Queue |
|---|---|---|
| Order | Based on priority | FIFO (First In, First Out) |
| Insertion | Anywhere (Sorted) | Always at Rear |
| Deletion | Highest/Lowest priority first | Always from Front |
| Usage | Scheduling, Graphs, Networks | Simple queuing systems |

---

## 7 Summary

✓ A **Priority Queue** processes elements based on **priority**, not FIFO.
✓ Used in **CPU scheduling, graph algorithms, and networking.**
✓ Can be implemented using **arrays, linked lists, or heaps (efficient).**
✓ Sorting during insertion ensures **fast deletion.**

Would you like me to implement **Priority Queue using Linked List or Heap**? 🚀

# UNIT-IV

## Searching

Searching in data structures refers to the process of finding a specific element within a data structure, such as an array, linked list, tree, or graph. There are various searching algorithms, each suited to different types of data structures and use cases.

---

### Types of Searching Algorithms

#### 1. Linear Search

- Works on: **Unsorted arrays, linked lists**
- Approach: Traverses each element one by one until the target element is found.
- Time Complexity: **O(n)**
- Best for: Small or unsorted datasets.

**Example in Java:**

```
public class LinearSearch {
    public static int search(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Element found, return index
            }
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int target = 30;
        int result = search(arr, target);
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

#### 2. Binary Search (Efficient)

- Works on: **Sorted arrays**
- Approach: Uses the divide-and-conquer strategy by repeatedly dividing the search space in half.
- Time Complexity: **O(log n)**
- Best for: Large, sorted datasets.

**Example in Java:**

```
import java.util.Arrays;

public class BinarySearch {
    public static int search(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) return mid;
            else if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int target = 30;
        int result = search(arr, target);
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

## 3. Hashing (Constant Time Search)

- Works on: **Hash tables, HashMaps**
- Approach: Uses a **hash function** to compute an index where the element is stored.
- Time Complexity: **O(1) (on average)**
- Best for: Fast lookups when hashing is implemented well.

### Example in Java (Using HashMap):

```
import java.util.HashMap;

public class HashSearch {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Cherry");

        int targetKey = 2;
        if (map.containsKey(targetKey)) {
            System.out.println("Element found: " + map.get(targetKey));
        } else {
            System.out.println("Element not found");
        }
    }
}
```

---

## 4. Search in Trees

*Binary Search Tree (BST) Search*

- Works on: **Binary Search Trees (BST)**
- Approach: Recursively or iteratively compares the target value with the root and decides whether to go left or right.
- Time Complexity: **O(log n) (average case), O(n) (worst case for unbalanced trees)**

### Example in Java:

```
class Node {
    int value;
    Node left, right;

    Node(int value) {
        this.value = value;
        left = right = null;
    }
}

class BST {
    Node root;

    public boolean search(Node root, int target) {
        if (root == null) return false;
        if (root.value == target) return true;
        return target < root.value ? search(root.left, target) :
search(root.right, target);
    }

    public static void main(String[] args) {
        BST tree = new BST();
        tree.root = new Node(50);
        tree.root.left = new Node(30);
        tree.root.right = new Node(70);
        tree.root.left.left = new Node(20);
        tree.root.left.right = new Node(40);

        System.out.println(tree.search(tree.root, 40) ? "Element found" :
"Element not found");
    }
}
```

---

## Comparison of Searching Algorithms

| Algorithm | Best Case | Average Case | Worst Case | Works On |
|---|---|---|---|---|
| **Linear Search** | O(1) | O(n) | O(n) | Unsorted arrays, linked lists |
| **Binary Search** | O(1) | O(log n) | O(log n) | Sorted arrays |

| Algorithm | Best Case | Average Case | Worst Case | Works On |
|---|---|---|---|---|
| **Hashing** | O(1) | O(1) | O(n) (collisions) | Hash tables (HashMap in Java) |
| **BST Search** | O(1) | O(log n) | O(n) (if unbalanced) | Binary search trees |

---

### Conclusion

- Use **Linear Search** for small or unsorted data.
- Use **Binary Search** if the data is **sorted**.
- Use **Hashing** for fast lookups when hashing is well-implemented.
- Use **BST Search** if you have tree-based data.

Let me know if you need more details or explanations! 🚀

## Linear Search (Sequential Search)

Linear Search, also known as **Sequential Search**, is the simplest searching algorithm. It checks each element in the data structure one by one until the target element is found or the entire structure is traversed.

---

### How It Works

1. Start from the **first** element.
2. Compare each element with the target.
3. If a match is found, return the index.
4. If the search reaches the end of the data structure without finding the target, return **-1** (element not found).

---

### Time Complexity

| Case | Complexity |
|---|---|
| Best Case | **O(1)** (Element found at the first position) |
| Average Case | **O(n)** |
| Worst Case | **O(n)** (Element not found or at the last position) |

- **Space Complexity**: **O(1)** (No extra space needed)

---

## When to Use Linear Search?

✅ **Unsorted or unordered data**
✅ **Small datasets**
✅ **Linked lists (since binary search doesn't work efficiently on linked lists)**
❌ **Not efficient for large datasets compared to Binary Search**

---

## Linear Search in Java (Iterative Approach)

```
public class LinearSearch {
    public static int search(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Element found, return index
            }
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int target = 30;
        int result = search(arr, target);
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

## Linear Search in Java (Recursive Approach)

```
public class LinearSearchRecursive {
    public static int search(int[] arr, int target, int index) {
        if (index >= arr.length) return -1; // Base case: Not found
        if (arr[index] == target) return index; // Found, return index
        return search(arr, target, index + 1); // Recursively search next
element
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int target = 40;
        int result = search(arr, target, 0);
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

## Advantages & Disadvantages

### ✅ Advantages

- **Simple & easy to implement**
- **Works for both sorted and unsorted data**
- **Good for small datasets**

### ❌ Disadvantages

- **Slow for large datasets** (O(n) time complexity)
- **Inefficient compared to binary search for sorted arrays**

---

## Comparison with Binary Search

| Feature | Linear Search | Binary Search |
|---|---|---|
| Works on | Unsorted & Sorted Data | Sorted Data Only |
| Time Complexity (Worst) | O(n) | O(log n) |
| Extra Space | O(1) | O(1) |
| Best For | Small datasets, Linked Lists | Large Sorted Arrays |

---

## Conclusion

- **Use Linear Search when the data is small or unsorted.**
- **Use Binary Search when the data is large and sorted.**

Let me know if you need further clarification! 🚀

# Binary Search (Efficient Searching Algorithm)

Binary Search is an efficient algorithm used to find an element in a **sorted** array. It follows the **divide-and-conquer** approach by repeatedly dividing the search space in half.

---

## How Binary Search Works

1. **Start with a sorted array.**

2. Find the **middle** element.
3. Compare it with the target:
    - If it's equal → **Element found, return index.**
    - If it's smaller → **Search in the right half.**
    - If it's larger → **Search in the left half.**
4. Repeat until the element is found or the search space is empty.

---

## Time Complexity

| Case | Complexity |
|---|---|
| **Best Case** | **O(1)** (Element is found in the middle) |
| **Average Case** | **O(log n)** |
| **Worst Case** | **O(log n)** (Element not found) |

- **Space Complexity**: **O(1)** (Iterative) or **O(log n)** (Recursive, due to function call stack)

---

## When to Use Binary Search?

✅ **Only works for sorted arrays**
✅ **Efficient for large datasets (O(log n) vs. O(n) in Linear Search)**
❌ **Doesn't work for unsorted data or linked lists (unless converted to an array)**

---

## Binary Search in Java (Iterative Approach)

```
public class BinarySearch {
    public static int search(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2; // Avoids overflow
            if (arr[mid] == target) return mid; // Found
            if (arr[mid] < target) left = mid + 1; // Search right half
            else right = mid - 1; // Search left half
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50, 60};
        int target = 30;
        int result = search(arr, target);
```

```
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

## Binary Search in Java (Recursive Approach)

```
public class BinarySearchRecursive {
    public static int search(int[] arr, int left, int right, int target) {
        if (left > right) return -1; // Base case: not found

        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid; // Found
        return (arr[mid] < target) ? search(arr, mid + 1, right, target) :
search(arr, left, mid - 1, target);
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50, 60};
        int target = 50;
        int result = search(arr, 0, arr.length - 1, target);
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

## Advantages & Disadvantages

### ✅ Advantages

- **Much faster than Linear Search (O(log n) vs. O(n))**
- **Efficient for large datasets**
- **Can be implemented iteratively or recursively**

### ❌ Disadvantages

- **Only works on sorted data**
- **Not efficient for small arrays (due to extra calculations)**
- **Does not work well with linked lists (due to O(n) access time for middle elements)**

---

## Comparison: Binary Search vs. Linear Search

| Feature | Binary Search | Linear Search |
|---|---|---|
| **Works on** | Sorted arrays | Any list (sorted/unsorted) |
| **Time Complexity (Worst)** | O(log n) | O(n) |

| Feature | Binary Search | Linear Search |
|---|---|---|
| **Best for** | Large datasets | Small datasets or unsorted data |
| **Implementation** | More complex | Simple |
| **Search Method** | Divide & Conquer | Sequential |

---

### Conclusion

- **Use Binary Search for large, sorted datasets.**
- **Use Linear Search for small or unsorted datasets.**

Let me know if you have any questions! 🚀

## Indexed Sequential Search

Indexed Sequential Search is a combination of **Sequential Search** and **Binary Search** (or Jump Search). It is used when dealing with **large datasets** that are **sorted** but require faster searching than a purely sequential approach.

---

### How Indexed Sequential Search Works

1. **Create an Index Table:**
   - The dataset is divided into **blocks**.
   - Each block has a **starting index** stored in an index table.
   - The index table contains key values that help in searching.
2. **Search in the Index Table (Binary Search or Linear Search on Indexes):**
   - Find the block where the target value might be present.
3. **Perform Sequential Search in the Block:**
   - Once the correct block is located, use **Linear Search** within that block.

---

### Time Complexity

| Case | Complexity |
|---|---|
| **Best Case** | **O(1)** (If the element is found in the index table) |
| **Average Case** | **O(√n)** (If indexing is optimal) |

| Case | Complexity |
|---|---|
| **Worst Case** | **O(n)** (If indexing fails and full search is needed) |

- **Space Complexity: O(√n)** (Extra space needed for the index table)

---

## When to Use Indexed Sequential Search?

✅ **Large sorted datasets**
✅ **Databases & File Systems**
✅ **When a balance between Linear and Binary Search is needed**
❌ **Not efficient for small datasets**
❌ **Extra memory required for indexing**

---

## Indexed Sequential Search in Java

```
public class IndexedSequentialSearch {
    public static int indexedSearch(int[] arr, int[] index, int target) {
        int blockSize = index.length;

        // Step 1: Find the correct block
        int block = -1;
        for (int i = 0; i < blockSize; i++) {
            if (index[i] >= target) {
                block = i;
                break;
            }
        }

        if (block == -1) return -1; // Element is larger than the largest
index value

        // Step 2: Perform Linear Search in the block
        int start = (block == 0) ? 0 : (block * (arr.length / blockSize));
        int end = Math.min(start + (arr.length / blockSize), arr.length);

        for (int i = start; i < end; i++) {
            if (arr[i] == target) return i; // Found
        }

        return -1; // Not found
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}; // Sorted
array
        int[] index = {30, 60, 90}; // Index table
```

```
        int target = 70;
        int result = indexedSearch(arr, index, target);
        System.out.println(result != -1 ? "Element found at index: " + result
: "Element not found");
    }
}
```

---

## Advantages & Disadvantages

### ✅ Advantages

- Faster than **Linear Search** (because of indexing)
- Doesn't require fully sorted data (only indexed portions)
- Suitable for **database searching** and **large file systems**

### ❌ Disadvantages

- Extra **storage required** for the index table
- Index must be **maintained and updated** when data changes
- Not as fast as **pure Binary Search** for fully sorted data

---

## Comparison with Other Searching Techniques

| Feature | Indexed Sequential Search | Binary Search | Linear Search |
|---|---|---|---|
| Works on | Sorted data with index | Sorted arrays | Unsorted or sorted data |
| Speed | O(√n) | O(log n) | O(n) |
| Extra Space | O(√n) | O(1) | O(1) |
| Best Use Case | Large datasets (databases, files) | Fully sorted arrays | Small datasets |

---

## Conclusion

- **Use Indexed Sequential Search when working with large datasets where indexing is possible.**
- **Use Binary Search when the data is fully sorted and stored in an array.**
- **Use Linear Search when the dataset is small or unsorted.**

Let me know if you need further explanations! 🚀

# Sorting in Data Structures

Sorting is the process of **arranging elements** in a specific order, usually **ascending** or **descending**. Sorting is important because it helps in **faster searching, efficient data processing, and better readability**.

---

## Types of Sorting Algorithms

Sorting algorithms are broadly classified into two types:

1. **Comparison-Based Sorting** (Bubble Sort, Selection Sort, Quick Sort, Merge Sort, etc.)
2. **Non-Comparison-Based Sorting** (Counting Sort, Radix Sort, Bucket Sort)

---

# Selection Sort in C Language

Selection Sort is a **simple** and **efficient** sorting algorithm that works by **repeatedly finding the smallest (or largest) element** from the unsorted part of the array and placing it in its correct position.

---

## How Selection Sort Works

1. **Find the Minimum Element**:
   - Start from the first element and find the smallest element in the unsorted part of the array.
2. **Swap the Minimum with the First Unsorted Element**:
   - Swap the found minimum element with the first element of the unsorted part.
3. **Repeat** the process for the remaining elements until the entire array is sorted.

---

## Selection Sort Algorithm in C

```
#include <stdio.h>

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, minIdx, temp;

    // Traverse through all elements
    for (i = 0; i < n - 1; i++) {
        minIdx = i; // Assume the minimum is at index i
```

```
        // Find the minimum element in the remaining array
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j; // Update minIdx
            }
        }

        // Swap the found minimum element with the first element
        temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

---

## Example Output

```
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64
```

---

## Time and Space Complexity

| Case | Time Complexity |
|---|---|
| **Best Case** | **$O(n^2)$** |
| **Average Case** | **$O(n^2)$** |

| Case | Time Complexity |
|---|---|
| **Worst Case** | **$O(n^2)$** |

- **Space Complexity**: **O(1)** (Sorting is done in-place, using no extra memory)
- **Stable?** ✗ (Selection Sort is **not stable** because it swaps elements)

---

## Advantages & Disadvantages

### ✅ Advantages

- **Easy to implement**.
- **Does not require extra memory** (works in-place).
- **Performs well on small datasets**.

### ✗ Disadvantages

- **Slow for large datasets** ($O(n^2)$ complexity).
- **Not a stable sort** (may change the order of equal elements).

---

## Comparison with Other Sorting Algorithms

| Algorithm | Time Complexity (Worst) | Space Complexity | Stable? |
|---|---|---|---|
| **Selection Sort** | $O(n^2)$ | O(1) | ✗ |
| **Bubble Sort** | $O(n^2)$ | O(1) | ✅ |
| **Insertion Sort** | $O(n^2)$ | O(1) | ✅ |
| **Merge Sort** | O(n log n) | O(n) | ✅ |
| **Quick Sort** | $O(n^2)$ | O(log n) | ✗ |

---

## Conclusion

Selection Sort is a simple sorting algorithm that is **good for small datasets** but inefficient for large datasets due to its **$O(n^2)$ complexity**. For **large datasets**, algorithms like **Merge Sort or Quick Sort** are preferred.

# Bubble Sort in C Language

**Bubble Sort** is a simple sorting algorithm that repeatedly **compares adjacent elements** and **swaps them** if they are in the wrong order. The process continues until the array is completely sorted.

---

## How Bubble Sort Works

1. Compare adjacent elements.
2. Swap them if they are in the wrong order.
3. Repeat this process for each element in the array.
4. The largest elements "bubble" to the end of the array after each pass.

---

## Bubble Sort Algorithm in C

```
#include <stdio.h>

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    int swapped;

    // Loop through each element in the array
    for (i = 0; i < n - 1; i++) {
        swapped = 0; // Optimization: Check if any swap happens

        // Compare adjacent elements
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

                swapped = 1; // Mark that a swap occurred
            }
        }

        // If no swap occurred, array is already sorted
        if (!swapped) break;
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
// Main function
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

---

## Example Output

```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

---

## Time and Space Complexity

| Case | Time Complexity |
|---|---|
| **Best Case** | **O(n)** (Already sorted) |
| **Average Case** | **O(n²)** |
| **Worst Case** | **O(n²)** |

- **Space Complexity**: **O(1)** (Sorting is done in-place)
- **Stable? ✅ (Bubble Sort is stable)**

---

## Advantages & Disadvantages

### ✅ Advantages

- **Simple to implement**.
- **Stable sorting algorithm** (does not change the order of equal elements).
- **Works well for small datasets**.

### ❌ Disadvantages

- **Inefficient for large datasets** ($O(n^2)$ complexity).

- **Slower than other sorting algorithms** like Quick Sort and Merge Sort.

---

### Comparison with Other Sorting Algorithms

| Algorithm | Time Complexity (Worst) | Space Complexity | Stable? |
|---|---|---|---|
| **Bubble Sort** | $O(n^2)$ | O(1) | ✅ |
| **Selection Sort** | $O(n^2)$ | O(1) | ❌ |
| **Insertion Sort** | $O(n^2)$ | O(1) | ✅ |
| **Merge Sort** | O(n log n) | O(n) | ✅ |
| **Quick Sort** | $O(n^2)$ | O(log n) | ❌ |

---

### Conclusion

Bubble Sort is an **easy-to-understand sorting algorithm** that works well for **small or nearly sorted datasets**, but it is **inefficient for large datasets**. Algorithms like **Quick Sort and Merge Sort** are much faster for large datasets.

## Insertion Sort in C Language

**Insertion Sort** is a simple and efficient sorting algorithm that builds the final sorted array **one item at a time** by placing each element in its correct position.

---

### How Insertion Sort Works

1. Start from the second element (index 1), assuming the first element is already sorted.
2. Compare the selected element with the elements before it.
3. Shift larger elements one position to the right.
4. Insert the selected element into its correct position.
5. Repeat for all elements in the array.

---

### Insertion Sort Algorithm in C

```
#include <stdio.h>

// Function to perform Insertion Sort
```

```
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i]; // Pick the current element
        j = i - 1;

        // Shift elements of arr[0..i-1] that are greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key; // Insert key at the correct position
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    insertionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

---

## Example Output

```
Original array: 12 11 13 5 6
Sorted array: 5 6 11 12 13
```

---

## Time and Space Complexity

| Case | Time Complexity |
|---|---|
| **Best Case** | **O(n)** (Already sorted) |
| **Average Case** | **O(n²)** |

| Case | Time Complexity |
|---|---|
| **Worst Case** | **$O(n^2)$** |

- **Space Complexity**: **O(1)** (Sorting is done in-place)
- **Stable? ✅ (Insertion Sort is stable)**

---

## Advantages & Disadvantages

### ✅ Advantages

- **Efficient for small and nearly sorted datasets**.
- **Stable sorting algorithm** (does not change the order of equal elements).
- **Works well when only a few elements are out of order**.

### ❌ Disadvantages

- **Inefficient for large datasets** ($O(n^2)$ complexity).
- **Slower than algorithms like Quick Sort and Merge Sort for large datasets**.

---

## Comparison with Other Sorting Algorithms

| Algorithm | Time Complexity (Worst) | Space Complexity | Stable? |
|---|---|---|---|
| **Insertion Sort** | $O(n^2)$ | O(1) | ✅ |
| **Selection Sort** | $O(n^2)$ | O(1) | ❌ |
| **Bubble Sort** | $O(n^2)$ | O(1) | ✅ |
| **Merge Sort** | O(n log n) | O(n) | ✅ |
| **Quick Sort** | $O(n^2)$ | O(log n) | ❌ |

---

## Conclusion

Insertion Sort is an **easy-to-implement algorithm** that works well for **small or nearly sorted datasets**, but it is **inefficient for large datasets**. If performance is a concern, **Merge Sort or Quick Sort** is a better choice.

# Quick Sort in C Language

**Quick Sort** is a highly efficient **divide-and-conquer** sorting algorithm. It selects a **pivot element** and partitions the array around the pivot such that **smaller elements are placed on the left** and **larger elements on the right**. This process is repeated recursively for the left and right subarrays until the entire array is sorted.

---

## How Quick Sort Works

1. **Choose a Pivot**: Pick an element as the pivot (e.g., last element, first element, or median).
2. **Partition the Array**:
    - Rearrange elements so that **smaller elements** are placed **before the pivot** and **larger elements** after it.
    - The pivot is now at its correct position.
3. **Recursively Apply Quick Sort**:
    - Quick sort is applied to the left and right subarrays.
4. **Base Case**:
    - The recursion stops when subarrays contain one or zero elements.

---

## Quick Sort Algorithm in C

```
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function to rearrange elements
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as pivot
    int i = low - 1; // Index for smaller elements

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]); // Place pivot at correct position
    return (i + 1);
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
```

```
    if (low < high) {
        int pi = partition(arr, low, high); // Get partition index

        quickSort(arr, low, pi - 1); // Sort left subarray
        quickSort(arr, pi + 1, high); // Sort right subarray
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

---

## Example Output

```
Original array: 10 7 8 9 1 5
Sorted array: 1 5 7 8 9 10
```

---

## Time and Space Complexity

| Case | Time Complexity |
|---|---|
| **Best Case** | **O(n log n)** (Balanced partitions) |
| **Average Case** | **O(n log n)** |
| **Worst Case** | **$O(n^2)$** (Unbalanced partitions, e.g., already sorted array) |

- **Space Complexity**: **O(log n)** (Recursive calls)
- **Stable?** ✗ (Quick Sort is **not stable**)

## Advantages & Disadvantages

### ✅ Advantages

- **Very fast** (O(n log n) on average).
- **Efficient for large datasets**.
- **In-place sorting** (does not require extra memory).

### ❌ Disadvantages

- **Worst-case $O(n^2)$ performance** (if pivot is poorly chosen).
- **Not a stable sort** (does not preserve the relative order of equal elements).
- **Recursive algorithm**, which may cause stack overflow for very large inputs.

---

## Comparison with Other Sorting Algorithms

| Algorithm | Time Complexity (Worst) | Space Complexity | Stable? |
|---|---|---|---|
| **Quick Sort** | $O(n^2)$ | O(log n) | ❌ |
| **Merge Sort** | O(n log n) | O(n) | ✅ |
| **Bubble Sort** | $O(n^2)$ | O(1) | ✅ |
| **Insertion Sort** | $O(n^2)$ | O(1) | ✅ |
| **Selection Sort** | $O(n^2)$ | O(1) | ❌ |

---

## Conclusion

Quick Sort is one of the **fastest sorting algorithms** for large datasets, with an average time complexity of **O(n log n)**. However, it is **not stable** and may perform **poorly in the worst case ($O(n^2)$)** if not optimized (e.g., using **randomized pivots** or **median-of-three pivot selection**).

# Merge Sort in C Language

**Merge Sort** is a highly efficient, **divide-and-conquer** sorting algorithm. It **recursively divides** the array into two halves, sorts them, and then **merges** the sorted halves.

---

## How Merge Sort Works

1. **Divide**: Split the array into two halves until each subarray contains a single element.
2. **Conquer**: Recursively sort the two halves.
3. **Merge**: Combine the sorted halves back into a single sorted array.

---

## Merge Sort Algorithm in C

```
#include <stdio.h>

// Function to merge two halves of the array
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2]; // Temporary arrays

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the two subarrays back into arr[]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy any remaining elements of L[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy any remaining elements of R[]
    while (j < n2) {
```

```
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Merge Sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

---

## Example Output

```
Original array: 38 27 43 3 9 82 10
Sorted array: 3 9 10 27 38 43 82
```

---

## Time and Space Complexity

| Case | Time Complexity |
| --- | --- |

| Case | Time Complexity |
|---|---|
| **Best Case** | **O(n log n)** |
| **Average Case** | **O(n log n)** |
| **Worst Case** | **O(n log n)** |

- **Space Complexity**: **O(n)** (extra space needed for merging)
- **Stable? ✅ (Merge Sort is stable)**

---

## Advantages & Disadvantages

### ✅ Advantages

- **Guaranteed O(n log n) time complexity**.
- **Stable sorting algorithm** (preserves the relative order of equal elements).
- **Works well for large datasets**.

### ❌ Disadvantages

- **Uses extra memory (O(n))**.
- **Slower for small datasets compared to Quick Sort or Insertion Sort**.

---

## Comparison with Other Sorting Algorithms

| Algorithm | Time Complexity (Worst) | Space Complexity | Stable? |
|---|---|---|---|
| **Merge Sort** | O(n log n) | O(n) | ✅ |
| **Quick Sort** | $O(n^2)$ | O(log n) | ❌ |
| **Bubble Sort** | $O(n^2)$ | O(1) | ✅ |
| **Insertion Sort** | $O(n^2)$ | O(1) | ✅ |
| **Selection Sort** | $O(n^2)$ | O(1) | ❌ |

---

# UNIT-V

## Binary Trees in Data Structures

A **Binary Tree** is a hierarchical data structure where each node has at most **two children**: a **left child** and a **right child**.

---

### Basic Terminology

1. **Root**: The topmost node in the tree.
2. **Parent Node**: A node that has children.
3. **Child Node**: A node that is a descendant of another node.
4. **Leaf Node**: A node with **no children**.
5. **Depth**: The length of the path from the **root** to a specific node.
6. **Height**: The length of the longest path from a **node to a leaf**.
7. **Subtree**: A tree consisting of a node and its descendants.
8. **Binary Search Tree (BST)**: A binary tree where the **left child** contains **smaller values** and the **right child** contains **larger values**.

---

### Types of Binary Trees

1. **Full Binary Tree**: Every node has either **0 or 2 children**.
2. **Complete Binary Tree**: All levels are completely filled **except possibly the last**, which is filled from **left to right**.
3. **Perfect Binary Tree**: All internal nodes have **exactly two children**, and all leaf nodes are at the **same level**.
4. **Balanced Binary Tree**: The height difference between the left and right subtrees is **at most 1**.
5. **Degenerate (Skewed) Tree**: Every node has **only one child**, forming a linked list-like structure.

---

### Binary Tree Implementation in C

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

// Structure for a node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Inorder Traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}

// Preorder Traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// Postorder Traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}

// Main function
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

---

## Example Output

```
Inorder traversal: 4 2 5 1 3
```

```
Preorder traversal: 1 2 4 5 3
Postorder traversal: 4 5 2 3 1
```

---

## Binary Tree Traversal Methods

1. **Inorder Traversal (Left, Root, Right)**
   - Visits **left subtree**, then **root**, then **right subtree**.
   - Used in **Binary Search Trees (BSTs)** to retrieve values in **sorted order**.
2. **Preorder Traversal (Root, Left, Right)**
   - Visits **root**, then **left subtree**, then **right subtree**.
   - Used in **tree cloning** and **prefix expressions**.
3. **Postorder Traversal (Left, Right, Root)**
   - Visits **left subtree**, then **right subtree**, then **root**.
   - Used in **deleting trees** and **postfix expressions**.

---

## Time Complexity

| Operation | Time Complexity |
|---|---|
| **Insertion** | **O(log n)** (Balanced Tree), **O(n)** (Unbalanced Tree) |
| **Deletion** | **O(log n)** (Balanced Tree), **O(n)** (Unbalanced Tree) |
| **Search** | **O(log n)** (Balanced Tree), **O(n)** (Unbalanced Tree) |
| **Traversal (Inorder, Preorder, Postorder)** | **O(n)** |

---

## Applications of Binary Trees

- **Binary Search Trees (BSTs)** for **efficient searching and sorting**.
- **Expression Trees** for **evaluating mathematical expressions**.
- **File Systems** (Hierarchical storage).
- **Heap Data Structure** (Priority Queue implementation).
- **Decision Trees** (Machine Learning & AI).

---

## Conclusion

Binary Trees are **fundamental** in data structures, with various applications in **searching, sorting, and hierarchical data organization**. If you need further explanations or code examples, let me know! 🚀

## Concept of Non-Linear Data Structures

A **Non-Linear Data Structure** is a type of data structure where data elements are not arranged sequentially (like arrays or linked lists). Instead, they follow a hierarchical or interconnected structure, allowing complex relationships between elements.

---

## Key Characteristics of Non-Linear Data Structures

1. **Multiple Connections**: Elements can be connected to multiple elements.
2. **Efficient Searching & Sorting**: Faster than linear structures for certain operations (e.g., BST for search).
3. **Hierarchical Representation**: Data is organized in a tree or graph-like fashion.
4. **Complex Relationships**: Elements can have multiple parents or children.

---

## Types of Non-Linear Data Structures

### 1 Trees

A **tree** is a hierarchical data structure consisting of nodes. Each node contains:

- **Data**
- **A pointer to its child nodes**
  A **Binary Tree** is the most common type of tree.

#### *Common Types of Trees*

- **Binary Tree**: Each node has at most **two children**.
- **Binary Search Tree (BST)**: Left child < Parent < Right child.
- **Balanced Tree (AVL, Red-Black Tree)**: Self-balancing BST for optimal performance.
- **Heap (Min-Heap, Max-Heap)**: Used for priority queues.
- **Trie (Prefix Tree)**: Used in dictionaries, autocomplete, etc.

#### *Tree Example*

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

---

### 2 Graphs

A **Graph** is a collection of **nodes (vertices)** connected by **edges**.

### *Types of Graphs*

- **Directed Graph**: Edges have a direction.
- **Undirected Graph**: Edges do not have a direction.
- **Weighted Graph**: Edges have weights (used in shortest path algorithms like Dijkstra's).
- **Cyclic & Acyclic Graphs**: Whether cycles exist or not.

### *Graph Representation*

1. **Adjacency Matrix**
   - A 2D matrix representation of a graph.
   - Space Complexity: **$O(V^2)$**
2. **Adjacency List**
   - A list where each node points to its connected nodes.
   - Space Complexity: **O(V + E)** (Efficient for sparse graphs)

### *Graph Example in C*

```
struct Graph {
    int V;  // Number of vertices
    struct Node* adjList[];
};
```

---

## Comparison: Linear vs Non-Linear Data Structures

| Feature | Linear DS (Array, Linked List) | Non-Linear DS (Tree, Graph) |
|---|---|---|
| Structure | Sequential | Hierarchical/Interconnected |
| Traversal | Sequential (One way) | Multiple paths |
| Memory Usage | Less (fixed allocation) | More (dynamic allocation) |
| Searching Complexity | O(n) | O(log n) (BST), O(1) (Hash) |
| Examples | Array, Linked List | Tree, Graph, Heap |

---

## Applications of Non-Linear Data Structures

- **Trees** → Database indexing (B-Trees, BST), HTML DOM.
- **Graphs** → Social networks (Facebook, LinkedIn), Shortest path (Google Maps).
- **Tries** → Autocomplete, Search engines.
- **Heaps** → Priority queues, Scheduling systems.

---

# Introduction to Binary Trees

## What is a Binary Tree?

A **Binary Tree** is a type of **non-linear data structure** where each node has at most **two children**:

1. **Left Child**
2. **Right Child**

Binary trees are widely used in **searching, sorting, hierarchical data representation**, and **graph traversal**.

---

## Basic Structure of a Binary Tree

Each node in a binary tree consists of:

- **Data (Value)**
- **Pointer to Left Child**
- **Pointer to Right Child**

### Binary Tree Node Structure in C

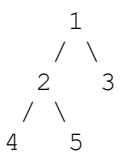
```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

---

## Types of Binary Trees

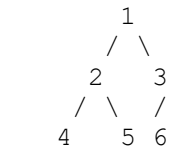
### 1 Full Binary Tree

- Every node has either **0 or 2 children**.
- **Example**:



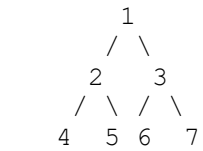

### 2 Complete Binary Tree

- All levels are **completely filled**, except possibly the **last level**, which is filled **from left to right**.
- **Example**:



### 3 Perfect Binary Tree

- All **internal nodes** have **two children**, and all **leaf nodes** are at the **same level**.
- **Example**:

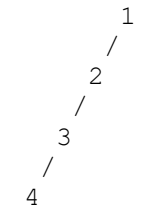


### 4 Balanced Binary Tree

- The height difference between the left and right subtree is **at most 1**.
- Example: **AVL Tree, Red-Black Tree**.

### 5 Degenerate (Skewed) Tree

- Each parent node has only **one child**, forming a **linked list-like structure**.
- **Example (Left Skewed Tree)**:



---

## Binary Tree Traversal Techniques

### 1. Inorder Traversal (Left → Root → Right)

```
void inorder(struct Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
```

**✓ Used for Binary Search Trees (BSTs) to get sorted order.**

### 2. Preorder Traversal (Root → Left → Right)

```
void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
```

```
    preorder(root->right);
}
```

**✓ Used for copying trees, and prefix expressions.**

### 3. Postorder Traversal (Left → Right → Root)

```
void postorder(struct Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

**✓ Used for deleting trees and evaluating postfix expressions.**

---

## Binary Tree Implementation in C

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

// Structure for a Node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Inorder Traversal
void inorder(struct Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Main Function
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder Traversal: ");
```

```
    inorder(root);
    printf("\n");

    return 0;
}
```

✓ **Example Output**:

```
Inorder Traversal: 4 2 5 1 3
```

---

## Applications of Binary Trees

✅ **Binary Search Trees (BSTs)** – Fast searching and sorting.
✅ **Expression Trees** – Evaluating mathematical expressions.
✅ **Heap Data Structure** – Used in priority queues.
✅ **Decision Trees** – Used in AI and Machine Learning.
✅ **File Systems** – Directory structure in OS.

---

## Conclusion

Binary Trees are fundamental in computer science and are used in various applications like **searching, sorting, and hierarchical data representation**. If you need more details on **Binary Search Trees (BSTs) or Heaps**, let me know! 🚀

# Types of Trees in Data Structures 🌳

A **Tree** is a **non-linear data structure** that consists of **nodes** connected in a **hierarchical manner**. The topmost node is called the **root**, and each node may have **child nodes**.

---

## 1️⃣ General Tree

- A tree where **each node can have any number of children**.
- Example: File System, XML Parsing.

### Example Structure

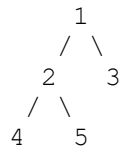
```
      A
    / | \
   B  C  D
  / \
 E   F
```

---

## 2 Binary Tree

- A tree where **each node can have at most two children**:
  - **Left Child**
  - **Right Child**
- Example: Binary Search Tree (BST), Heap.

### Example Structure

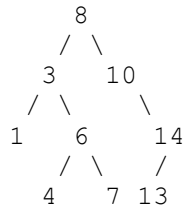



---

## 3 Binary Search Tree (BST)

- A **Binary Tree** with the property:
  - **Left subtree** contains values **less than** the root.
  - **Right subtree** contains values **greater than** the root.
- Used for **fast searching, insertion, and deletion** operations.

### Example Structure




### ⏳ Time Complexity:

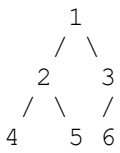
| Operation | Average Case | Worst Case (Skewed Tree) |
|---|---|---|
| Search | O(log n) | O(n) |
| Insert | O(log n) | O(n) |
| Delete | O(log n) | O(n) |

---

## 4 Complete Binary Tree

- All levels are **completely filled**, except possibly the **last level**, which is filled **from left to right**.

### Example Structure



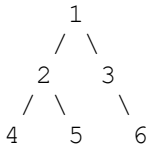

✅ **Used in Heap Data Structures**.

---

## 5 Full Binary Tree (Strictly Binary Tree)

- Every node has **either 0 or 2 children** (no node has only 1 child).

### Example Structure

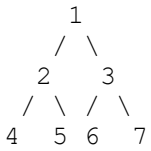



---

## 6 Perfect Binary Tree

- All **internal nodes have exactly two children**, and **all leaf nodes are at the same level**.

### Example Structure



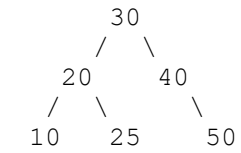

📌 **Nodes at level `h` = `2^h - 1`**
(Example: A **perfect binary tree of height 3** has **7 nodes**)

---

## 7️⃣ Balanced Binary Tree

- The height difference between the **left and right subtrees** is **at most 1**.

### Example: AVL Tree (Self-Balancing BST)



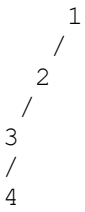

✅ **Operations like insertion, deletion, and search take O(log n) time.**

---

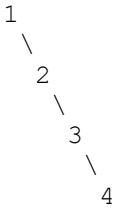
## 8️⃣ Degenerate (Skewed) Tree

- Every node has **only one child**, making it behave like a **linked list**.

### Left-Skewed Tree




### Right-Skewed Tree




🚨 **Worst-case search time = O(n) (like a linked list).**

---

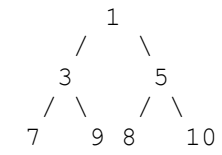
## 9️⃣ Heap Tree

- A **complete binary tree** where **parent nodes follow a specific order**.
- Used in **Priority Queues**.

### Types of Heaps

✅ **Min Heap**: Parent node is **smaller** than children.
✅ **Max Heap**: Parent node is **greater** than children.

### Min Heap Example

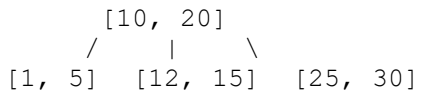



✅ **Used in Heap Sort, Priority Queues, Scheduling Algorithms.**

---

## 🔟 B-Trees (Balanced Trees for Databases)

- Used in **database indexing** and **file systems**.
- Self-balancing trees where each node can have **multiple children**.
- **Example**: B-Tree of order 3 (Each node can have 2-3 keys).




✅ **Used in databases (MySQL, MongoDB) and file systems (NTFS, HFS+).**

---

## Conclusion

Different tree structures serve various **computational needs**:

- **BSTs** → Efficient Searching (O(log n))
- **AVL Trees** → Self-Balancing BSTs
- **Heaps** → Priority Queues
- **B-Trees** → Database Indexing

💡 Let me know if you need **detailed implementations in C**! 🚀

# Basic Definition of Binary Trees 🌳

A **Binary Tree** is a **non-linear data structure** in which **each node has at most two children**:

1. **Left Child**
2. **Right Child**

## Key Characteristics of Binary Trees

✓ **Hierarchical structure** (Unlike arrays or linked lists).
✓ **Each node contains**:

- **Data (Value)**
- **Left child pointer**
- **Right child pointer**

✓ **Maximum nodes at level 'h'** = $2^h - 1$ (Perfect Binary Tree).

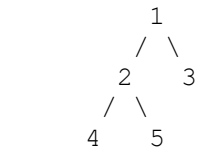
---

## Binary Tree Node Structure in C

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

---

## Example Structure of a Binary Tree




---

## Applications of Binary Trees

✅ **Binary Search Trees (BSTs)** → Fast Searching & Sorting.
✅ **Heap Trees** → Priority Queues (Min Heap, Max Heap).
✅ **Expression Trees** → Parsing Mathematical Expressions.
✅ **File Systems & Databases** → B-Trees, B+ Trees.

Would you like an implementation in C? 🚀

# Properties of Binary Trees

A **Binary Tree** is a tree data structure where each node has at most **two children**. Understanding its properties helps in analyzing **space complexity, height, and efficiency** in searching, insertion, and deletion operations.

---

## 1 Maximum Number of Nodes at Each Level

- A binary tree at **level 'h'** can have **at most** $2^h$ nodes.
- A binary tree of **height 'h'** can have **at most** $(2^h - 1)$ nodes.

### Example

```
Height = 3
        1
      /  \
     2    3
    / \   / \
   4  5  6   7

Total Nodes = 2³ - 1 = 7
```

---

## 2 Minimum Height of a Binary Tree

- The minimum height **h ($\log_2$ N) for N nodes** is: h=⌈log2(N+1)⌉−1h = $\lceil \log_2(N + 1) \rceil$ - 1
- **Example**: A perfect binary tree with $N = 7$ has $h = \log_2(7+1) - 1 = 2$.

---

## 3 Types of Binary Trees Based on Properties

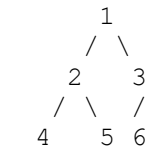
### ✅ Full Binary Tree

- Every node **has either 0 or 2 children**.
- Maximum nodes in a full binary tree of height **h**: 2h−12$^h$ - 1

```
      1
    /  \
   2    3
  / \  / \
 4  5 6  7
```

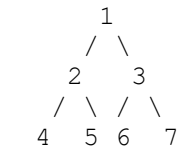
### ✅ Complete Binary Tree

- All levels **except possibly the last** are completely filled.
- Last level is filled **from left to right**.



### ☑ Perfect Binary Tree

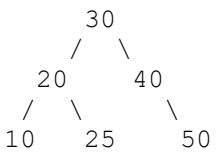
- All **internal nodes have two children**.
- **Leaf nodes are at the same level**.



- **Total nodes**: $2^h - 1$
- **Height**: $\log_2(N + 1) - 1$

### ☑ Balanced Binary Tree

- Height difference between left and right subtree is **≤ 1**.
- Example: **AVL Tree, Red-Black Tree**.



- **Time Complexity**: $O(\log n)$

---

## 4 Maximum Number of Nodes in a Binary Tree of Height 'h'

- The **maximum** number of nodes in a binary tree of height $h$ is: 2h−12^h - 1
- Example:
  - **Height = 3** → Max nodes = $2^3 - 1 = 7$
  - **Height = 4** → Max nodes = $2^4 - 1 = 15$

---

## 5 Minimum Number of Nodes in a Binary Tree of Height 'h'

- **Minimum nodes in a tree of height 'h'** is: h+1h + 1
- Example:
  - **Height = 3** → Min nodes = $3 + 1 = 4$

```
\
 2
  \
   3
    \
     4
```

---

## 6 Traversal Properties

There are three main **traversal** techniques: 1 **Inorder (Left → Root → Right)** → Sorted order in BST.
2 **Preorder (Root → Left → Right)** → Used in tree cloning.
3 **Postorder (Left → Right → Root)** → Used in deleting a tree.

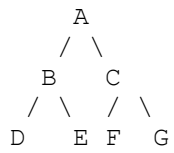
---

## 7 Relationship Between Internal & Leaf Nodes

For a **Full Binary Tree** with **N internal nodes**, the number of **leaf nodes (L)** is:

$L=N+1$

Example:




- **Internal Nodes (N) = 3**
- **Leaf Nodes (L) = 4**

---

## 8 Sum of Degrees in a Binary Tree

- The **total number of edges in a binary tree** is always: TotalNodes−1Total Nodes - 1
- **Example**: A tree with 7 nodes has 6 edges.

---

## 9 Depth & Height Relationship

- **Depth of a node** = Number of edges from root to the node.
- **Height of a node** = Number of edges from the node to the deepest leaf.

Example:

```
      1   (Depth = 0)
     / \
    2   3  (Depth = 1)
   / \
  4   5  (Depth = 2)
```

- **Height of node 2 = 1** (max path to leaf node).
- **Depth of node 5 = 2**.

---

## 10 Applications of Binary Tree Properties

✅ **Binary Search Trees (BSTs)** → Searching & Sorting.
✅ **Expression Trees** → Mathematical expression evaluation.
✅ **Heaps (Min/Max Heap)** → Priority queues, heap sort.
✅ **B-Trees & B+ Trees** → Database indexing, file system organization.
✅ **Huffman Tree** → Data compression algorithms.

---

## Conclusion

Binary trees have **specific properties** that make them useful in **data storage, searching, and hierarchical structures**.
Would you like an example **implementation in C**? 🚀

# Representation of Binary Trees 🌳

Binary trees can be represented in **two main ways** in computer memory:

1. **Linked Representation** (Using pointers)
2. **Array Representation** (Using index-based storage)

---

## 1️⃣1️⃣ Linked Representation (Using Pointers)

- Each node contains:
    1. **Data** (Value stored in the node)
    2. **Pointer to Left Child**
    3. **Pointer to Right Child**
- **Dynamic memory allocation** is used.

### Structure of a Node in C

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

// Define a Node structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Example Usage
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);

    printf("Root Node: %d\n", root->data);
    printf("Left Child: %d\n", root->left->data);
    printf("Right Child: %d\n", root->right->data);

    return 0;
}
```

### Example Binary Tree Structure in Memory

```
      [1]
     /   \
   [2]   [3]
```

✓ **Advantages**

✅ Efficient in memory allocation.

✅ Good for dynamic tree structures.

❌ **Disadvantages**

❌ Extra space required for pointers.

---

## 2 Array Representation (Using Index-Based Storage)

- Nodes are stored in **sequential memory locations**.
- Parent-child relationships are maintained using **index calculations**:
    - **Parent at index** `i`

- **Left Child** at index `2*i + 1`
- **Right Child** at index `2*i + 2`

### Example Binary Tree

```
       1  →  Stored at index 0
      / \
     2   3  →  Stored at index 1 & 2
    / \
   4   5  →  Stored at index 3 & 4
```

### Array Representation

```
 Index:  0   1   2   3   4
 Values: [1] [2] [3] [4] [5]
```

### C Implementation

```c
#include <stdio.h>

// Function to print binary tree stored in an array
void printTree(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("Node at index %d: %d\n", i, arr[i]);
    }
}

int main() {
    int tree[] = {1, 2, 3, 4, 5}; // Array representation
    int size = sizeof(tree) / sizeof(tree[0]);

    printTree(tree, size);
    return 0;
}
```

✓ **Advantages**

✅ Fast **access** using index.

✅ **No extra space** for pointers.

❌ **Disadvantages**

❌ **Wastes space** for incomplete trees.

❌ Difficult to **insert/delete dynamically**.

---

## Comparison of Representations

| Representation | Storage Type | Best for | Space Complexity |
|---|---|---|---|
| **Linked Representation** | Pointers | Dynamic Trees | O(n) (Extra pointer storage) |
| **Array Representation** | Array | Complete Trees | O(n) (Efficient for full trees) |

---

### Conclusion

- **Use Linked Representation** when trees are **dynamic**.
- **Use Array Representation** for **complete binary trees** (like Heaps).

## Operations on a Binary Search Tree (BST) 🌳

A **Binary Search Tree (BST)** is a **sorted binary tree** where:

- **Left subtree** contains values **less than** the root.
- **Right subtree** contains values **greater than** the root.
- **No duplicate values** are allowed.

---

### 1️⃣ Basic Structure of a BST Node in C

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

// Define a Node structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

---

## 🔷 Operations on BST

### 1️⃣ Insertion in BST

- Insert a new node by following **BST property**.
- **Time Complexity:**
  - **Best/Average Case:** `O(log n)`
  - **Worst Case (Skewed Tree):** `O(n)`

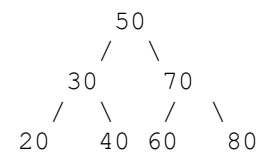
### C Implementation

```
// Function to insert a node in BST
struct Node* insert(struct Node* root, int value) {
    if (root == NULL) return createNode(value);
```

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

    return root;
}
```

✅ **Example:**
Inserting **50, 30, 70, 20, 40, 60, 80**




---

## 2 Searching in BST

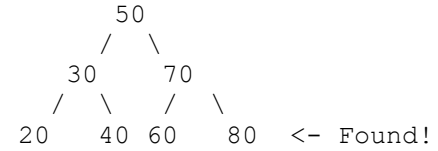
- Compare value with **root**.
- If **smaller**, search in the **left subtree**.
- If **greater**, search in the **right subtree**.
- **Time Complexity:** `O(log n)`, worst case `O(n)`.

### C Implementation

```
// Function to search a value in BST
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key) return root;

    if (key < root->data)
        return search(root->left, key);
    return search(root->right, key);
}
```

✅ **Example:** Searching for **40**




---

## 3 Deletion in BST

- **Case 1:** Node has **no child** → Simply delete.
- **Case 2:** Node has **one child** → Replace with child.
- **Case 3:** Node has **two children** → Replace with **inorder successor** (smallest in right subtree).
- **Time Complexity:** `O(log n)`, worst case `O(n)`.

## C Implementation

```
// Function to find the minimum node in a subtree
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Function to delete a node in BST
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node with one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: Get inorder successor
        struct Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

✅ **Example:** Deleting **30**

```
Before:
       50
      /  \
    30    70
   /  \   /  \
 20   40 60   80

After:
       50
      /  \
    40    70
   /     /  \
 20     60   80
```

---

## 4️⃣ Traversals in BST

BST can be traversed using:
1️⃣ **Inorder Traversal** (Left → Root → Right) → Sorted order.
2️⃣ **Preorder Traversal** (Root → Left → Right) → Used in copying trees.
3️⃣ **Postorder Traversal** (Left → Right → Root) → Used in deleting trees.

### C Implementation

```
// Inorder Traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

### ✅ Example Output (Inorder for BST)

```
20 30 40 50 60 70 80
```

---

## 5️⃣ Finding Minimum and Maximum in BST

- **Minimum** = Leftmost node.
- **Maximum** = Rightmost node.
- **Time Complexity:** `O(log n)`, worst case `O(n)`.

### C Implementation

```
// Find Minimum Value in BST
int findMin(struct Node* root) {
    struct Node* current = root;
    while (current->left != NULL)
        current = current->left;
    return current->data;
}

// Find Maximum Value in BST
int findMax(struct Node* root) {
    struct Node* current = root;
    while (current->right != NULL)
        current = current->right;
    return current->data;
}
```

### ✅ Example Output

```
Min: 20
Max: 80
```

---

## 6️⃣ Finding Height of BST

- **Height of BST** = Longest path from root to leaf.
- **Time Complexity:** `O(n)` (Unbalanced), `O(log n)` (Balanced).

### C Implementation

```
// Function to find height of BST
int height(struct Node* root) {
    if (root == NULL) return -1;
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}
```

### ✅ Example

```
Height: 2
```

---

## 🔷 Summary of Operations

| Operation | Best Case | Worst Case |
|---|---|---|
| **Insertion** | O(log n) | O(n) |
| **Search** | O(log n) | O(n) |
| **Deletion** | O(log n) | O(n) |
| **Traversal** | O(n) | O(n) |
| **Find Min/Max** | O(log n) | O(n) |
| **Height** | O(log n) | O(n) |

---

## 🚀 Conclusion

- **BST is efficient for searching & sorting.**
- **Balancing BST (like AVL Tree) improves performance.**
- **Used in databases, file systems, and search algorithms.**

# Binary Tree Traversal in C

Tree traversal is the process of **visiting each node** in a **binary tree** exactly once in a specific order.

---

## Types of Binary Tree Traversals

There are **three main types** of tree traversal methods:

### 1 Depth-First Traversal (DFS)

- **Inorder Traversal** `(Left → Root → Right)`
- **Preorder Traversal** `(Root → Left → Right)`
- **Postorder Traversal** `(Left → Right → Root)`

### 2 Breadth-First Traversal (BFS)

- **Level Order Traversal** `(Level by Level)`

---

## 1 Depth-First Traversal (DFS)

### (i) Inorder Traversal (L → Root → R)

- **Steps**:
  1. Traverse **left** subtree.
  2. Visit **root** node.
  3. Traverse **right** subtree.
- **Output**: Nodes are printed in **sorted order** if it is a **Binary Search Tree (BST)**.

#### C Implementation

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

// Node Structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Create New Node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Inorder Traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Driver Code
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder Traversal: ");
    inorder(root);
    return 0;
}
```

### ✅ Output

```
Inorder Traversal: 4 2 5 1 3
```

---

## ✅ (ii) Preorder Traversal (Root → L → R)

- **Steps**:
    1. Visit **root** node.
    2. Traverse **left** subtree.
    3. Traverse **right** subtree.

### C Implementation

```
// Preorder Traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

### ✅ Output

```
Preorder Traversal: 1 2 4 5 3
```

---

### ✅ (iii) Postorder Traversal (L → R → Root)

- **Steps**:
    1. Traverse **left** subtree.
    2. Traverse **right** subtree.
    3. Visit **root** node.

#### C Implementation

```
// Postorder Traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
```

#### ✅ Output

```
Postorder Traversal: 4 5 2 3 1
```

---

## 2 Breadth-First Traversal (BFS)

### ✅ Level Order Traversal

- **Uses a Queue** (FIFO approach).
- **Steps**:
    1. Start from **root** and enqueue it.
    2. Dequeue a node, print it, and enqueue its **left & right** children.
    3. Repeat until queue is empty.
- **Time Complexity**: `O(n)`

#### C Implementation

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

// Node Structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Create New Node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
```

```
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Find Height of Tree
int height(struct Node* root) {
    if (root == NULL) return 0;
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
}

// Print Nodes at Given Level
void printLevel(struct Node* root, int level) {
    if (root == NULL) return;
    if (level == 1) printf("%d ", root->data);
    else {
        printLevel(root->left, level - 1);
        printLevel(root->right, level - 1);
    }
}

// Level Order Traversal
void levelOrder(struct Node* root) {
    int h = height(root);
    for (int i = 1; i <= h; i++)
        printLevel(root, i);
}

// Driver Code
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Level Order Traversal: ");
    levelOrder(root);
    return 0;
}
```

### ✅ Output

```
Level Order Traversal: 1 2 3 4 5
```

---

## 🔷 Summary of Traversals

| Traversal Type | Order | Use Case |
|---|---|---|
| **Inorder (L → Root → R)** | 4 2 5 1 3 | BST sorting |

| Traversal Type | Order | Use Case |
|---|---|---|
| **Preorder (Root → L → R)** | 1 2 4 5 3 | Copying tree structure |
| **Postorder (L → R → Root)** | 4 5 2 3 1 | Deleting tree nodes |
| **Level Order (BFS)** | 1 2 3 4 5 | Shortest path in trees |

---

## 🚀 Conclusion

- **DFS (Inorder, Preorder, Postorder)** are **recursive** and efficient for **small trees**.
- **BFS (Level Order)** is useful for **finding shortest paths** in **graphs and trees**.
- **Applications**:
  - ✅ **Inorder** - Sorting in BST
  - ✅ **Preorder** - Expression Trees
  - ✅ **Postorder** - Deleting nodes
  - ✅ **Level Order** - Shortest path problems

Would you like a **full program** with all traversals together? 😊

## Applications of Binary Trees 🌳🚀

A **Binary Tree** is a hierarchical data structure in which each node has at most **two children**. It has numerous real-world applications in **computer science**, **data processing**, and **networking**.

---

### 🔷 1. Binary Search Trees (BST) – Efficient Searching & Sorting

- A **BST** is a special type of binary tree where left nodes contain smaller values and right nodes contain larger values.
- **Application**:
  - ✅ **Databases** – Used for indexing large amounts of data efficiently.
  - ✅ **File Systems** – Searching and storing hierarchical file directories.
  - ✅ **Symbol Tables in Compilers** – Store variable names efficiently.
- **Example**: Searching in BST takes **O(log n)** time in a balanced tree.

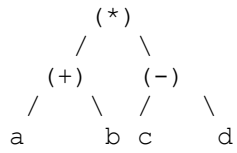
---

### 🔷 2. Expression Trees – Evaluating Mathematical Expressions

- Used to **evaluate algebraic expressions** with **operators (+, -, *, /)**.

- **Application**:
  - ✅ **Compilers** – Convert infix expressions into postfix or prefix for computation.
  - ✅ **Calculators** – Solve arithmetic expressions.

### Example of Expression Tree for (a + b) * (c - d)

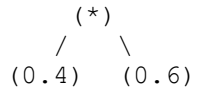



---

## 🔷 3. Huffman Coding – Data Compression (Used in ZIP, MP3, JPEG)

- Huffman Trees are **Binary Trees** used for **lossless data compression**.
- **Application**:
  - ✅ **File Compression** – ZIP, GZIP, JPEG, MP3, MPEG.
  - ✅ **Text Compression** – Reduces file sizes for efficient storage.

### Example of Huffman Tree



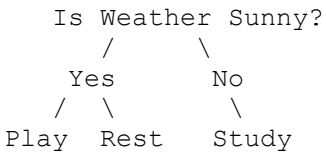

Smaller frequency characters are stored **deeper**, giving a compressed encoding.

---

## 🔷 4. Decision Trees – AI & Machine Learning 🤖

- A **Decision Tree** is a type of binary tree used in **machine learning algorithms**.
- **Application**:
  - ✅ **AI & ML Models** – Used for decision-making (classification and regression).
  - ✅ **Medical Diagnosis** – Predicting diseases based on symptoms.
  - ✅ **Fraud Detection** – Identifying suspicious transactions.

### Example of a Decision Tree




---

## 🔷 5. File Systems – Hierarchical Storage

- File systems in operating systems use **binary trees** for efficient **file management**.

- **Application**:
  ✅ **Linux File System (EXT, NTFS)** – Stores directories as tree structures.
  ✅ **Windows File Explorer** – Uses hierarchical trees to organize files.

### Example

```
Root
 ├── Documents
 │    ├── Notes.txt
 │    ├── Report.docx
 ├── Pictures
 │    ├── Photo1.jpg
 │    ├── Photo2.png
```

---

## 🔷 6. Network Routing & Internet Protocols

- Binary trees are used in **networking** for efficient routing of **data packets**.
- **Application**:
  ✅ **IP Address Routing** – Uses Binary Tries (Trie Trees) for fast lookups.
  ✅ **DNS (Domain Name System)** – Uses a hierarchical tree structure for domain resolution.

### Example:

```
         Root (.)
        /        \
   .com          .org
   /   \         /    \
google  yahoo wikipedia opensource
```

---

## 🔷 7. Game Trees – AI in Chess, Tic-Tac-Toe

- Used in **game development** and **AI decision-making**.
- **Application**:
  ✅ **Chess AI (Minimax Algorithm)** – Evaluates possible moves.
  ✅ **Tic-Tac-Toe** – Represents all possible game states.

### Example of a Tic-Tac-Toe Game Tree

```
    Root (Empty Board)
       /       \
   X Move      O Move
    /   \       /   \
 Next Moves  Next Moves
```

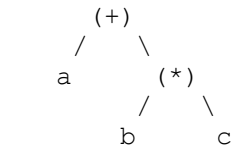
---

## 🔷 8. Parsing Trees – Used in Compilers & Interpreters

- **Syntax trees** (Parse Trees) are used in compilers to **parse and execute** programming languages.

- **Application**:
  - ✅ **Compilers (C, Java, Python)** – Convert source code into an executable.
  - ✅ **Natural Language Processing (NLP)** – Sentence parsing in AI.

### Example of Parse Tree for "a + (b * c)"

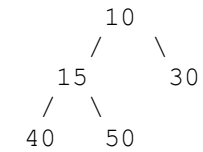



---

## 🔷 9. Binary Heaps – Used in Priority Queues

- **Binary Heaps** are a special type of binary tree used to implement **Priority Queues**.
- **Application**:
  - ✅ **Heap Sort Algorithm** – Efficient sorting in `O(n log n)`.
  - ✅ **Dijkstra's Algorithm** – Finds the shortest path in graphs.
  - ✅ **Operating Systems (Process Scheduling)** – Uses Min Heap for task scheduling.

### Example of a Min-Heap




---

## 🚀 Summary – Where Are Binary Trees Used?

| Application | Type of Tree Used | Used In |
|---|---|---|
| **Searching & Sorting** | **Binary Search Tree (BST)** | Databases, File Systems |
| **Expression Evaluation** | **Expression Tree** | Compilers, Calculators |
| **Compression (ZIP, JPEG, MP3)** | **Huffman Tree** | File Compression |
| **AI & Machine Learning** | **Decision Tree** | Medical Diagnosis, Fraud Detection |
| **File Management** | **General Binary Tree** | Windows, Linux File Systems |
| **Network Routing** | **Binary Trie** | IP Address Lookup, DNS |
| **Game AI (Chess, Tic-Tac-Toe)** | **Game Trees** | AI, Gaming |

| Application | Type of Tree Used | Used In |
|---|---|---|
| **Parsing Source Code** | **Syntax Tree** | Compilers, NLP |
| **Priority Queues & Scheduling** | **Binary Heap** | CPU Scheduling, Graph Algorithms |

---

## 🎯 Conclusion

✅ **Binary Trees are fundamental in computer science** and have real-world applications in **AI, networking, databases, operating systems, and data compression**.
✅ Understanding their applications **helps in designing efficient algorithms** and **optimizing performance** in various domains.

# Introduction to Graphs 📊

A **graph** is a **non-linear data structure** that consists of **nodes (vertices) and edges**. Graphs are widely used in **computer science**, **networking**, **AI**, **social media**, and **many real-world applications**.

---

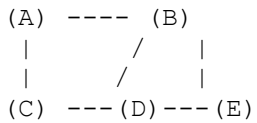
## 🔷 Definition of a Graph

A **graph G(V, E)** is defined as:

- **V** = Set of vertices (nodes)
- **E** = Set of edges (connections between nodes)

A graph is represented as **G = (V, E)**.

### Example Graph Representation




Here, **A, B, C, D, and E** are **vertices**, and the **lines** connecting them are **edges**.

---

# 🔷 Types of Graphs

Graphs are classified into different types:

## 1️⃣ Directed vs. Undirected Graphs

- **Undirected Graph**: Edges have **no direction**.
  - Example: **Social Networks (Facebook Friendships)**
- **Directed Graph (Digraph)**: Edges have **direction** (one-way connections).
  - Example: **Website Links (Hyperlinks), Twitter Follow System**

## 2️⃣ Weighted vs. Unweighted Graphs

- **Unweighted Graph**: All edges are **equal in cost**.
- **Weighted Graph**: Each edge has a **weight (cost or distance)**.
  - Example: **Google Maps (Shortest Path Calculation)**

## 3️⃣ Cyclic vs. Acyclic Graphs

- **Cyclic Graph**: Contains **at least one cycle**.
  - Example: **Computer Networks with Loops**
- **Acyclic Graph**: **No cycles**.
  - Example: **Tree Data Structures**

## 4️⃣ Connected vs. Disconnected Graphs

- **Connected Graph**: Every vertex is **reachable**.
- **Disconnected Graph**: Some vertices are **not reachable**.

## 5️⃣ Special Types of Graphs

- **Tree**: A **connected acyclic** graph.
- **Bipartite Graph**: Vertices can be **divided into two groups** with no edges within a group.
- **Complete Graph**: Every node is **connected to every other node**.

---

# 🔷 Graph Representation in Computer Science

Graphs can be represented using:

## 1️⃣ Adjacency Matrix

A **2D array** of size `V x V`, where:

- `graph[i][j] = 1` if there's an edge between `i` and `j`.
- **Space Complexity**: `O(V²)`

✅ **Example** (for 4 nodes):

```
   A  B  C  D
A [0, 1, 1, 0]
B [1, 0, 1, 1]
C [1, 1, 0, 1]
D [0, 1, 1, 0]
```

## 2️⃣ Adjacency List

- Each vertex stores a **list of its neighbors**.
- **Efficient for sparse graphs**.
- **Space Complexity**: `O(V + E)`

✅ **Example (Adjacency List Representation)**

```
A → B, C
B → A, C, D
C → A, B, D
D → B, C
```

## 3️⃣ Edge List

A list of **edges (pairs of nodes)**.

- Example: `{(A, B), (A, C), (B, C), (B, D), (C, D)}`

---

# 🔷 Graph Traversal Algorithms

Graph traversal is used to **visit** all nodes in a graph.

## 1️⃣ Breadth-First Search (BFS)

- Uses a **Queue (FIFO)**
- Explores **level by level** (like waves).
- Used in **shortest path algorithms** (e.g., Google Maps).

### C Implementation of BFS

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

#define MAX 100

int adj[MAX][MAX]; // Adjacency matrix
int visited[MAX];

void BFS(int start, int vertices) {
    int queue[MAX], front = 0, rear = 0;
    queue[rear++] = start;
    visited[start] = 1;

    while (front < rear) {
        int node = queue[front++];
        printf("%d ", node);

        for (int i = 0; i < vertices; i++) {
            if (adj[node][i] == 1 && !visited[i]) {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
    }
}

int main() {
    int vertices = 4;

    adj[0][1] = adj[1][0] = 1;
    adj[0][2] = adj[2][0] = 1;
    adj[1][2] = adj[2][1] = 1;
    adj[1][3] = adj[3][1] = 1;

    printf("BFS Traversal: ");
    BFS(0, vertices); // Start BFS from node 0
    return 0;
}
```

### ✅ Output

```
BFS Traversal: 0 1 2 3
```

---

## 2️⃣ Depth-First Search (DFS)

- Uses a **Stack (LIFO)** (or recursion).
- Explores **deepest paths first** before backtracking.
- Used in **solving mazes, cycle detection, topological sorting**.

### C Implementation of DFS

```
void DFS(int node, int vertices) {
    printf("%d ", node);
    visited[node] = 1;

    for (int i = 0; i < vertices; i++) {
        if (adj[node][i] == 1 && !visited[i]) {
            DFS(i, vertices);
        }
    }
}
```

### ✅ Output

```
DFS Traversal: 0 1 2 3
```

---

## 🔷 Applications of Graphs 🚀

| Application | Graph Type | Example |
|---|---|---|
| **Social Networks** | Undirected | Facebook Friends, LinkedIn |
| **Web Crawling** | Directed | Google Search Engine |
| **Google Maps** | Weighted | Shortest Path (Dijkstra's Algorithm) |
| **Computer Networks** | Directed | Routing Algorithms (OSPF, BGP) |
| **AI (Game Trees)** | Directed | Chess, Tic-Tac-Toe AI |
| **Database Indexing** | Directed | Graph Databases (Neo4j, ArangoDB) |
| **Operating Systems** | Directed | Deadlock Detection (Resource Allocation Graph) |
| **Cryptography** | Directed | Blockchain Transactions |

| Application | Graph Type | Example |
|---|---|---|
| **Compilers** | Directed | Topological Sorting (Dependency Resolution) |

---

## 🚀 Conclusion

✅ **Graphs** are fundamental data structures used in networking, AI, web crawling, and routing.
✅ **Graph algorithms like BFS and DFS** help in navigation, searching, and problem-solving.
✅ **Choosing the right graph representation** (adjacency list/matrix) is important for efficiency.

## 📌 Terms Associated with Graphs

Graphs are an essential data structure used in **computer science, networking, AI, and more**. Below are the key terms associated with graphs:

---

### 1️⃣ Graph (G)

A **Graph (G)** is a collection of:

- **Vertices (Nodes, V)**
- **Edges (Connections, E)**
  It is represented as **G = (V, E)**.

#### Example Graph

```
(A) ---- (B)
 |      /   |
 |     /    |
(C) ---(D)---(E)
```

Here, **A, B, C, D, and E** are **vertices**, and the lines connecting them are **edges**.

---

### 2️⃣ Vertex (Node)

- A **vertex** is a point in the graph.
- Represented as `V = {A, B, C, D, E}`.

✅ **Example**: In a social network, each **person** is a vertex.

---

## 3 Edge

- An **edge** is a connection between two vertices.
- Represented as **E = {(A, B), (A, C), (B, C), (B, D), (D, E)}**.

✅ **Example**: In a road network, each **road** is an edge.

---

## 4 Degree of a Vertex

- **Degree (deg(v))** of a vertex is the **number of edges** connected to it.

✅ **Example**:

```
(A) -- (B) -- (C)
```

- **deg(A) = 1**
- **deg(B) = 2**
- **deg(C) = 1**
- **In-degree**: Number of incoming edges.
- **Out-degree**: Number of outgoing edges (only in directed graphs).

---

## 5 Directed vs. Undirected Graphs

- **Undirected Graph**: Edges have **no direction**. ✅ Example: Facebook Friendships.
- **Directed Graph (Digraph)**: Edges have **direction** (one-way connections). ✅ Example: Twitter Follow System.

✅ **Example**

```
Directed Graph:
    (A) → (B) → (C)
```

---

## 6 Weighted vs. Unweighted Graphs

- **Unweighted Graph**: All edges are **equal in cost**.
- **Weighted Graph**: Each edge has a **weight (cost or distance)**.

✅ **Example**

```
(A) --5-- (B) --3-- (C)
```

Here, **5 & 3** are weights (cost of travel).

---

## 7 Path

- A **path** is a sequence of vertices connected by edges.
- **Path Length** = Number of edges in the path.

✅ **Example**: **Path from A to E**:
`A → B → D → E`

---

## 8 Cycle

- A **cycle** is a path where the **starting and ending vertex are the same**.

✅ **Example**:

```
A -- B
|    |
D -- C
```

**Cycle**: `A → B → C → D → A`

---

## 9 Connected vs. Disconnected Graph

- **Connected Graph**: All vertices are **reachable**.
- **Disconnected Graph**: Some vertices are **not connected**.

✅ **Example**

```
Connected Graph:
    A -- B -- C -- D

Disconnected Graph:
    A    B -- C -- D
```

---

## 10 Bipartite Graph

- A graph where vertices can be **divided into two groups** with **no edges inside a group**.

### ✅ Example

```
Group 1: {A, C}
Group 2: {B, D}
Edges: (A-B), (A-D), (C-B), (C-D)
```

---

## 11 Complete Graph

- A graph where **every vertex is connected to every other vertex**.

### ✅ Example

```
  (A) -- (B)
   |  \/  |
   |  /\  |
  (C) -- (D)
```

Each node is **connected to all others**.

---

## 12 Graph Traversal

- **Breadth-First Search (BFS)**: Visits nodes **level by level** (Queue-based).
- **Depth-First Search (DFS)**: Visits nodes **deep first** before backtracking (Stack/Recursion).

### ✅ Example

```
BFS: A → B → C → D
DFS: A → B → D → C
```

---

## 13 Adjacency Matrix vs. Adjacency List

- **Adjacency Matrix**: `V x V` matrix storing edges.
- **Adjacency List**: Each vertex stores **a list of its neighbors**.

### ✅ Example (Adjacency List for Graph)

```
A → B, C
B → A, C, D
C → A, B, D
D → B, C
```

---

## 14️⃣ Spanning Tree

- A **subgraph** that connects all vertices **with minimum edges**.

### ✅ Example: Minimum Spanning Tree (MST)

```
Original Graph:
    A -- B -- C
     \       /
       D

MST:
    A -- B
    |    |
    D -- C
```

---

## 15️⃣ Shortest Path Algorithms

- **Dijkstra's Algorithm**: Finds the shortest path in weighted graphs.
- **Floyd-Warshall Algorithm**: Solves all-pairs shortest paths.
- **Bellman-Ford Algorithm**: Works for graphs with **negative weights**.

### ✅ Example

```
Graph:
    A --5-- B --3-- C

Shortest Path (A to C): **A → B → C (8)**
```

---

## 🚀 Summary of Graph Terms

| Term | Definition | Example |
|---|---|---|
| **Graph (G)** | Collection of nodes & edges | `G = (V, E)` |
| **Vertex (V)** | Node in the graph | A, B, C, D |
| **Edge (E)** | Connection between vertices | (A, B), (B, C) |
| **Degree** | Number of edges connected to a vertex | deg(A) = 2 |
| **Path** | Sequence of vertices | `A → B → C` |
| **Cycle** | Path where start & end are same | `A → B → C → A` |

| Term | Definition | Example |
|---|---|---|
| **Connected Graph** | All vertices are reachable | Road Networks |
| **Disconnected Graph** | Some vertices are isolated | Separate Social Groups |
| **Bipartite Graph** | Two groups, no internal edges | Job Matching |
| **Complete Graph** | Every node is connected to every other node | Fully Connected Network |
| **Graph Traversal** | BFS (Queue), DFS (Stack) | Searching Algorithms |
| **Adjacency Matrix** | `V x V` matrix storing edges | Used for dense graphs |
| **Adjacency List** | Each vertex stores its neighbors | Used for sparse graphs |
| **Spanning Tree** | Subgraph with minimum edges | Network Cabling |
| **Shortest Path** | Finds the least-cost path | Google Maps |

---

## 🚀 Conclusion

✅ Graphs are a powerful **non-linear data structure** with **real-world applications** in networking, AI, maps, social networks, and more.
✅ Understanding **graph terminology** is **essential** for algorithms like **BFS, DFS, Dijkstra's, and Prim's Algorithm**.
✅ If you're interested, I can provide **C programs** for graph traversal, shortest path algorithms, or spanning trees! 😊

Would you like to dive into **Graph Algorithms next**? 🚀

# 📌 Sequential Representation of Graphs

Graphs can be represented in different ways in **computer memory**. One of the most common representations is the **sequential representation**, which uses **arrays or matrices** to store graph data.

---

## 🔷 What is Sequential Representation?

In **Sequential Representation**, graphs are stored using **contiguous memory locations** like **arrays** and **matrices**. It is useful for representing **dense graphs** (graphs with many edges).

The two main sequential representations of graphs are:

1. **Adjacency Matrix Representation**
2. **Edge List Representation**

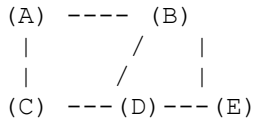
---

## 1️⃣ Adjacency Matrix Representation

An **Adjacency Matrix** is a **2D array** (`V x V`) where:

- `graph[i][j] = 1` → if an edge exists between `i` and `j`
- `graph[i][j] = 0` → if no edge exists

### ✅ Example Graph




### ✅ Adjacency Matrix Representation

```
   A  B  C  D  E
A [0, 1, 1, 0, 0]
B [1, 0, 1, 1, 1]
C [1, 1, 0, 1, 0]
D [0, 1, 1, 0, 1]
E [0, 1, 0, 1, 0]
```

### ✅ C Program to Implement Adjacency Matrix

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

```
#define V 5 // Number of vertices

void printGraph(int graph[V][V]) {
    printf("Adjacency Matrix Representation:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = { {0, 1, 1, 0, 0},
                        {1, 0, 1, 1, 1},
                        {1, 1, 0, 1, 0},
                        {0, 1, 1, 0, 1},
                        {0, 1, 0, 1, 0} };

    printGraph(graph);
    return 0;
}
```

### ✅ Output

```
Adjacency Matrix Representation:
0 1 1 0 0
1 0 1 1 1
1 1 0 1 0
0 1 1 0 1
0 1 0 1 0
```

### ✅ Advantages of Adjacency Matrix

- **Fast edge lookup** (`O(1)`)
- **Easy to implement**
- **Good for dense graphs** (many edges)

### ✅ Disadvantages of Adjacency Matrix

- **Wastes space** for sparse graphs (`O(V²)`)
- **Slow to find neighbors** (`O(V)` time complexity)

---

## 2 Edge List Representation

An **Edge List** stores only the **edges**, instead of a full matrix.

### ✅ Example Graph

```
    (A) ---- (B)
     |      /   |
     |     /    |
    (C) ---(D)---(E)
```

## ✅ Edge List Representation

```
(A, B)
(A, C)
(B, C)
(B, D)
(B, E)
(C, D)
(D, E)
```

## ✅ C Program to Implement Edge List

```
#include <stdio.h>

#define E 7 // Number of edges

void printEdges(int edges[E][2]) {
    printf("Edge List Representation:\n");
    for (int i = 0; i < E; i++) {
        printf("(%d, %d)\n", edges[i][0], edges[i][1]);
    }
}

int main() {
    int edges[E][2] = { {0, 1}, {0, 2}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {3,
4} };

    printEdges(edges);
    return 0;
}
```

## ✅ Output

```
Edge List Representation:
(0, 1)
(0, 2)
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(3, 4)
```

## ✅ Advantages of Edge List

- **Efficient for sparse graphs** (few edges)
- **Uses less space (O(E))**
- **Easy to iterate over edges**

### ✅ Disadvantages of Edge List

- **Finding neighbors is slow** (`O(E)`)
- **Checking edge existence takes** `O(E)`

---

## 🚀 Comparison of Graph Representations

| Representation | Best for | Space Complexity | Edge Lookup Time |
|---|---|---|---|
| **Adjacency Matrix** | Dense Graphs | `O(V²)` | `O(1)` |
| **Edge List** | Sparse Graphs | `O(E)` | `O(E)` |

---

## 🚀 Conclusion

✅ **Sequential Representation** of graphs is efficient for memory storage and operations.
✅ **Adjacency Matrix** is best for **dense graphs** (`O(V²)` space).
✅ **Edge List** is best for **sparse graphs** (`O(E)` space).
✅ Would you like me to explain **Adjacency List Representation** too? 🚀

## 📌 Linked Representation of Graphs (Adjacency List Representation)

In **Linked Representation**, we use **linked lists** to store the graph, making it **efficient for sparse graphs** (graphs with fewer edges).

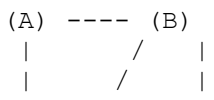
---

### 🔷 What is Linked Representation?

- Uses an **array of linked lists** to store neighbors of each vertex.
- Each **vertex** has a **linked list** storing its adjacent vertices.
- **Efficient for memory**: Only stores edges that exist (`O(V + E)` space).

---

### ✅ Example Graph



```
    (C) ---(D)---(E)
```

## ✅ Adjacency List Representation

```
A → B → C
B → A → C → D → E
C → A → B → D
D → B → C → E
E → B → D
```

---

## ✅ C Program for Adjacency List Representation

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

// Structure for a Node in Linked List
struct Node {
    int vertex;
    struct Node* next;
};

// Structure for Graph
struct Graph {
    int numVertices;
    struct Node** adjLists;
};

// Function to create a node
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a Graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    // Allocate memory for adjacency lists
    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}

// Function to add an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge src → dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge dest → src (Since it's undirected)
```

```
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to print adjacency list
void printGraph(struct Graph* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        struct Node* temp = graph->adjLists[i];
        printf("Vertex %d: ", i);
        while (temp) {
            printf(" -> %d", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

// Main Function
int main() {
    int vertices = 5;
    struct Graph* graph = createGraph(vertices);

    // Adding edges
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // Print adjacency list
    printGraph(graph);

    return 0;
}
```

### ✅ Output

```
Vertex 0:  -> 2 -> 1
Vertex 1:  -> 4 -> 3 -> 2 -> 0
Vertex 2:  -> 3 -> 1 -> 0
Vertex 3:  -> 4 -> 2 -> 1
Vertex 4:  -> 3 -> 1
```

---

## ✅ Advantages of Linked Representation (Adjacency List)

✓ **Saves space** (`O(V + E)`) for sparse graphs.
✓ **Efficient traversal** (BFS, DFS).
✓ **Adding a new edge is fast** (`O(1)`).

---

## ❌ Disadvantages

❌ **Checking edge existence takes $O(V)$** (instead of $O(1)$ in an adjacency matrix).
❌ **Extra memory required for pointers**.

---

## 🚀 Summary: Matrix vs. Linked Representation

| Representation | Best for | Space Complexity | Edge Lookup | Traversal Complexity |
|---|---|---|---|---|
| **Adjacency Matrix** | Dense Graphs | $O(V^2)$ | $O(1)$ | $O(V^2)$ |
| **Adjacency List** | Sparse Graphs | $O(V + E)$ | $O(V)$ | $O(V + E)$ |

---

## 🚀 Conclusion

✅ **Adjacency List (Linked Representation)** is best for **sparse graphs**.
✅ **Adjacency Matrix (Sequential Representation)** is better for **dense graphs**.

Would you like an explanation of **Graph Traversal (BFS & DFS)?** 🚀

## 📌 Graph Traversal: DFS & BFS

Graph traversal is the process of **visiting all the vertices** in a graph.
Two common methods are:
1️⃣ **Depth-First Search (DFS)** 🔍
2️⃣ **Breadth-First Search (BFS)** 🌐

---

## 1️⃣ Depth-First Search (DFS) 🔍

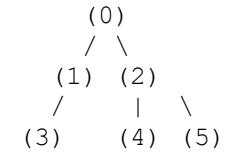
DFS explores **as deep as possible** along a branch before backtracking. It uses:
✓ **Stack (Explicit or Recursion)**

## ✅ DFS Algorithm

1. Start from any vertex.
2. **Mark** the vertex as **visited**.
3. **Move to an adjacent unvisited vertex** and repeat.
4. If no adjacent vertex is unvisited, **backtrack**.
5. Continue until all vertices are visited.

---

## ✅ Example Graph




### DFS Traversal (Starting from `0`)

`0 → 1 → 3 → 2 → 4 → 5`

---

## ✅ C Program for DFS

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

#define MAX 100

// Graph representation using adjacency list
struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

// Function to create a new node
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));
```

```
    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

// Function to add an edge
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Depth-First Search
void DFS(struct Graph* graph, int vertex) {
    printf("%d ", vertex);
    graph->visited[vertex] = 1;

    struct Node* temp = graph->adjLists[vertex];
    while (temp != NULL) {
        int adjVertex = temp->vertex;
        if (graph->visited[adjVertex] == 0) {
            DFS(graph, adjVertex);
        }
        temp = temp->next;
    }
}

// Main function
int main() {
    int vertices = 6;
    struct Graph* graph = createGraph(vertices);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 2, 5);

    printf("DFS Traversal: ");
    DFS(graph, 0);  // Start DFS from vertex 0

    return 0;
}
```

### ✅ Output

```
DFS Traversal: 0 1 3 2 4 5
```

---

## 2️⃣ Breadth-First Search (BFS) 🌐

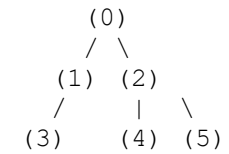
BFS explores **all neighbors first** before moving to the next level.
✓ **Uses a Queue (FIFO)**

### ✅ BFS Algorithm

1. Start from any vertex.
2. **Mark** it as **visited**.
3. **Push** it into the **queue**.
4. **Dequeue** a vertex, **visit all unvisited adjacent vertices**, and enqueue them.
5. Repeat until the queue is empty.

---

### ✅ BFS Example




#### BFS Traversal (Starting from `0`)

`0 → 1 → 2 → 3 → 4 → 5`

---

### ✅ C Program for BFS

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

#define MAX 100

// Queue implementation
struct Queue {
    int items[MAX];
    int front, rear;
};

struct Queue* createQueue() {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

int isEmpty(struct Queue* q) {
    return q->front == -1;
}

void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX - 1) return;
    if (q->front == -1) q->front = 0;
```

```
    q->items[++q->rear] = value;
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) return -1;
    int item = q->items[q->front];
    if (q->front >= q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front++;
    }
    return item;
}

// Graph representation using adjacency list
struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

```
// BFS Function
void BFS(struct Graph* graph, int startVertex) {
    struct Queue* queue = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(queue, startVertex);

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        printf("%d ", currentVertex);

        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(queue, adjVertex);
            }
            temp = temp->next;
        }
    }
}

// Main function
int main() {
    int vertices = 6;
    struct Graph* graph = createGraph(vertices);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 2, 5);

    printf("BFS Traversal: ");
    BFS(graph, 0);  // Start BFS from vertex 0

    return 0;
}
```

### ✅ Output

```
BFS Traversal: 0 1 2 3 4 5
```

---

## 🚀 Summary: DFS vs BFS

| Feature | DFS | BFS |
|---|---|---|
| **Uses** | Stack (or Recursion) | Queue |
| **Explores** | Deep first | Level by level |

| Feature | DFS | BFS |
|---|---|---|
| **Time Complexity** | `O(V + E)` | `O(V + E)` |
| **Space Complexity** | `O(V)` | `O(V)` |
| **Best for** | Pathfinding, Maze Solving | Shortest Path (Unweighted Graphs) |

---

Would you like explanations on **Shortest Path Algorithms** like **Dijkstra's Algorithm**? 🚀

## 📌 Applications of Graphs in Real Life

Graphs are widely used in various fields such as **computer science, engineering, social networks, and transportation**. They help **model complex relationships** between objects.

---

### 🚀 1. Computer Science & Networking

#### ✅ Social Networks (Facebook, Instagram, LinkedIn, Twitter)

- **Users → Vertices**
- **Friendships/Connections → Edges**
- Graph algorithms help **suggest friends**, detect communities, and recommend content.

#### ✅ Web Crawling & Search Engines (Google, Bing)

- **Webpages → Nodes**
- **Hyperlinks → Edges**
- **PageRank Algorithm** uses graphs to rank websites.

#### ✅ Computer Networks & Routing (Internet, LAN, WAN)

- **Devices (Routers, Computers) → Nodes**
- **Connections → Edges**
- **Routing Algorithms (Dijkstra's Algorithm, Bellman-Ford)** find the shortest path for data packets.

---

## 🚀 2. Transportation & Logistics

### ✅ Google Maps, GPS Navigation

- **Cities/Locations → Vertices**
- **Roads/Flights → Edges**
- Uses *Shortest Path Algorithms (Dijkstra, A)** to find the best route.

### ✅ Airline & Metro Systems

- Airports/stations as **nodes**, flights/trains as **edges**.
- Used for **scheduling and optimization**.

### ✅ Delivery & Supply Chain (Amazon, FedEx, UPS, DHL)

- Used for **route optimization** and **supply chain management**.
- **Graph-based heuristics** improve **cost & efficiency**.

---

## 🚀 3. Artificial Intelligence & Machine Learning

### ✅ Neural Networks

- **Neurons → Nodes**
- **Connections → Edges**
- Used in **Deep Learning & AI** models.

### ✅ Recommendation Systems (Netflix, YouTube, Spotify, Amazon)

- Graphs help analyze **user preferences** and recommend **movies, songs, and products**.

### ✅ Image Processing & Computer Vision

- **Graph Cut Algorithm** is used for **image segmentation** and **object recognition**.

---

## 🚀 4. Bioinformatics & Chemistry

### ✅ DNA Sequencing & Protein Structures

- **Genes, proteins → Nodes**
- **Interactions → Edges**
- Helps in **disease research** and **drug discovery**.

### ✅ Chemical Reactions & Molecular Structures

- Atoms as **nodes**, bonds as **edges**.
- Used in **drug design** and **chemical analysis**.

---

## 🚀 5. Electrical & Civil Engineering

### ✅ Circuit Design & VLSI

- Components as **nodes**, wires as **edges**.
- Graphs optimize **chip design & connectivity**.

### ✅ Water Supply & Power Grids

- **Pipelines, power stations → Nodes**
- **Connections → Edges**
- Used for **load balancing** and **network reliability**.

---

## 🚀 6. Cybersecurity & Fraud Detection

### ✅ Network Security

- Graphs detect **intrusions, malware, phishing attacks**.

### ✅ Fraud Detection (Banking & E-Commerce)

- Graphs help find **suspicious transactions & money laundering**.

---

## 🚀 7. Project Management & Dependency Analysis

### ✅ PERT & Critical Path Method (CPM)

- Used for **task scheduling** in projects.

### ✅ Software Dependencies (Package Managers like NPM, Maven, Pip)

- Packages as **nodes**, dependencies as **edges**.

---

## Summary Table

| Application | Graph Type | Example |
|---|---|---|
| Social Networks | Undirected Graph | Facebook, Instagram |
| Web Search | Directed Graph | Google PageRank |
| Road Networks | Weighted Graph | Google Maps, GPS |
| AI & ML | Neural Network | Deep Learning |
| DNA Sequencing | Biological Graph | Genome Mapping |
| Cybersecurity | Graph Analytics | Fraud Detection |
| Circuit Design | Directed Graph | VLSI, PCB Layout |
| Supply Chain | Weighted Graph | Amazon, FedEx Logistics |

---

## Conclusion

Graphs are **powerful tools** used in **many real-world applications**. They help model relationships, optimize paths, detect patterns, and solve complex problems.

# LAB PROGRAMS

## 1. Write a program to read 'N' numbers of elements into an array and also perform the following operation on an array

## a. Add an element at the beginning of an array

## b. Insert an element at given index of array

## c. Update an element using a values and index

## d. Delete an existing element

```
#include <stdio.h>


// Function to display the array
void displayArray(int arr[], int size) {
    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}


// Function to add an element at the beginning
void addAtBeginning(int arr[], int *size, int element) {
    for (int i = *size; i > 0; i--) {
        arr[i] = arr[i - 1];
    }
    arr[0] = element;
    (*size)++;
}
```

```
// Function to insert an element at a specific index
void insertAtIndex(int arr[], int *size, int index, int element) {
    if (index < 0 || index > *size) {
        printf("Invalid index!\n");
        return;
    }
    for (int i = *size; i > index; i--) {
        arr[i] = arr[i - 1];
    }
    arr[index] = element;
    (*size)++;
}

// Function to update an element at a given index
void updateElement(int arr[], int size, int index, int newValue) {
    if (index < 0 || index >= size) {
        printf("Invalid index!\n");
        return;
    }
    arr[index] = newValue;
}

// Function to delete an element at a given index
void deleteElement(int arr[], int *size, int index) {
    if (index < 0 || index >= *size) {
```

```
        printf("Invalid index!\n");
        return;
    }
    for (int i = index; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*size)--;
}

// Main function
int main() {
    int arr[100], size, choice, element, index;

    // Read the size of the array
    printf("Enter the number of elements: ");
    scanf("%d", &size);

    // Read the array elements
    printf("Enter %d elements: ", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    // Display initial array
    displayArray(arr, size);
```

```
while (1) {
    printf("\nOperations:\n");
    printf("1. Add element at beginning\n");
    printf("2. Insert element at index\n");
    printf("3. Update element\n");
    printf("4. Delete element\n");
    printf("5. Display array\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter element to add at beginning: ");
            scanf("%d", &element);
            addAtBeginning(arr, &size, element);
            displayArray(arr, size);
            break;

        case 2:
            printf("Enter index and element to insert: ");
            scanf("%d %d", &index, &element);
            insertAtIndex(arr, &size, index, element);
            displayArray(arr, size);
```

```
    break;

case 3:
    printf("Enter index and new value to update: ");
    scanf("%d %d", &index, &element);
    updateElement(arr, size, index, element);
    displayArray(arr, size);
    break;

case 4:
    printf("Enter index to delete element: ");
    scanf("%d", &index);
    deleteElement(arr, &size, index);
    displayArray(arr, size);
    break;

case 5:
    displayArray(arr, size);
    break;

case 6:
    printf("Exiting...\n");
    return 0;

default:
```

```
            printf("Invalid choice! Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Output :-

```
Enter the number of elements: 5

Enter 5 elements: 10 20 30 40 50

Array elements: 10 20 30 40 50

Operations:
1. Add element at beginning
2. Insert element at index
3. Update element
4. Delete element
5. Display array
6. Exit
Enter your choice: 1
Enter element to add at beginning: 5
Array elements: 5 10 20 30 40 50

Enter your choice: 2
Enter index and element to insert: 3 25
Array elements: 5 10 20 25 30 40 50

Enter your choice: 3
Enter index and new value to update: 2 15
Array elements: 5 10 15 25 30 40 50

Enter your choice: 4
Enter index to delete element: 4
Array elements: 5 10 15 25 40 50

Enter your choice: 6
Exiting...
```

## Write Program to implement Single Linked List with insertion, deletion and traversal operations

```
#include <stdio.h>

#include <stdlib.h>


// Define Node structure

struct Node {

   int data;

   struct Node* next;

};


// Function to insert at the beginning

void insertAtBeginning(struct Node** head, int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->data = value;

   newNode->next = *head;

   *head = newNode;

}


// Function to insert at the end

void insertAtEnd(struct Node** head, int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->data = value;

   newNode->next = NULL;
```

```
    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to insert at a specific position
void insertAtPosition(struct Node** head, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
```

```
        return;
    }

    struct Node* temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of range!\n");
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

// Function to delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
```

```
    free(temp);
}

// Function to delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp->next == NULL) {
        free(temp);
        *head = NULL;
        return;
    }

    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

    prev->next = NULL;
```

```
    free(temp);
}

// Function to delete from a specific position
void deleteFromPosition(struct Node** head, int position) {
    if (*head == NULL || position < 1) {
        printf("Invalid position or empty list!\n");
        return;
    }

    struct Node* temp = *head;

    if (position == 1) {
        *head = temp->next;
        free(temp);
        return;
    }

    struct Node* prev = NULL;
    for (int i = 1; temp != NULL && i < position; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
```

```
        printf("Position out of range!\n");
        return;
    }

    prev->next = temp->next;
    free(temp);
}

// Function to display the linked list
void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
```

```
struct Node* head = NULL;
int choice, value, position;

while (1) {
   printf("\nOperations:\n");
   printf("1. Insert at Beginning\n");
   printf("2. Insert at End\n");
   printf("3. Insert at Position\n");
   printf("4. Delete from Beginning\n");
   printf("5. Delete from End\n");
   printf("6. Delete from Position\n");
   printf("7. Display List\n");
   printf("8. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);

   switch (choice) {
      case 1:
         printf("Enter value to insert at beginning: ");
         scanf("%d", &value);
         insertAtBeginning(&head, value);
         break;

      case 2:
         printf("Enter value to insert at end: ");
```

```
    scanf("%d", &value);
    insertAtEnd(&head, value);
    break;

case 3:
    printf("Enter position and value to insert: ");
    scanf("%d %d", &position, &value);
    insertAtPosition(&head, value, position);
    break;

case 4:
    deleteFromBeginning(&head);
    break;

case 5:
    deleteFromEnd(&head);
    break;

case 6:
    printf("Enter position to delete: ");
    scanf("%d", &position);
    deleteFromPosition(&head, position);
    break;

case 7:
```

```
            displayList(head);
            break;

        case 8:
            printf("Exiting...\n");
            return 0;

        default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   }

   return 0;
}
```

Output :-

```
Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Exit
Enter your choice: 1
```

Enter value to insert at beginning: 10

Enter your choice: 1

Enter value to insert at beginning: 20

Enter your choice: 2

Enter value to insert at end: 30

Enter your choice: 7

Linked List: 20 -> 10 -> 30 -> NULL

Enter your choice: 3

Enter position and value to insert: 2 15

Linked List: 20 -> 15 -> 10 -> 30 -> NULL

Enter your choice: 6

Enter position to delete: 3

Linked List: 20 -> 15 -> 30 -> NULL

Enter your choice: 8

Exiting...

## Write Program to implement Circular doubly Linked List with insertion, deletion and traversal operations

```
#include <stdio.h>

#include <stdlib.h>


// Define Node structure

struct Node {

   int data;

   struct Node* next;

   struct Node* prev;

};


// Function to insert at the beginning

void insertAtBeginning(struct Node** head, int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->data = value;


   if (*head == NULL) {

      newNode->next = newNode;

      newNode->prev = newNode;

      *head = newNode;

      return;

   }


   struct Node* last = (*head)->prev;
```

```
    newNode->next = *head;
    newNode->prev = last;
    (*head)->prev = newNode;
    last->next = newNode;
    *head = newNode;
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (*head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
        *head = newNode;
        return;
    }

    struct Node* last = (*head)->prev;
    newNode->next = *head;
    newNode->prev = last;
    last->next = newNode;
    (*head)->prev = newNode;
}
```

```
// Function to insert at a specific position
void insertAtPosition(struct Node** head, int value, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
    }

    if (position == 1) {
        insertAtBeginning(head, value);
        return;
    }

    struct Node* temp = *head;
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    for (int i = 1; temp->next != *head && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp->next == *head && position != 2) {
        printf("Position out of range!\n");
        return;
    }
```

```
    newNode->next = temp->next;
    newNode->prev = temp;
    temp->next->prev = newNode;
    temp->next = newNode;
}

// Function to delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* last = (*head)->prev;

    if (*head == (*head)->next) {
        free(temp);
        *head = NULL;
        return;
    }

    *head = (*head)->next;
    (*head)->prev = last;
```

```
    last->next = *head;
    free(temp);
}

// Function to delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* last = (*head)->prev;

    if (*head == (*head)->next) {
        free(last);
        *head = NULL;
        return;
    }

    struct Node* secondLast = last->prev;
    secondLast->next = *head;
    (*head)->prev = secondLast;
    free(last);
}
```

```
// Function to delete from a specific position
void deleteFromPosition(struct Node** head, int position) {
    if (*head == NULL || position < 1) {
        printf("Invalid position or empty list!\n");
        return;
    }

    if (position == 1) {
        deleteFromBeginning(head);
        return;
    }

    struct Node* temp = *head;
    for (int i = 1; temp->next != *head && i < position; i++) {
        temp = temp->next;
    }

    if (temp->next == *head && position != 2) {
        printf("Position out of range!\n");
        return;
    }

    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    free(temp);
```

```
}

// Function to display the linked list
void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = head;
    printf("Circular Doubly Linked List: ");
    do {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(Back to Head)\n");
}

// Main function
int main() {
    struct Node* head = NULL;
    int choice, value, position;

    while (1) {
        printf("\nOperations:\n");
```

```
printf("1. Insert at Beginning\n");
printf("2. Insert at End\n");
printf("3. Insert at Position\n");
printf("4. Delete from Beginning\n");
printf("5. Delete from End\n");
printf("6. Delete from Position\n");
printf("7. Display List\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter value to insert at beginning: ");
      scanf("%d", &value);
      insertAtBeginning(&head, value);
      break;

   case 2:
      printf("Enter value to insert at end: ");
      scanf("%d", &value);
      insertAtEnd(&head, value);
      break;

   case 3:
```

```
        printf("Enter position and value to insert: ");
        scanf("%d %d", &position, &value);
        insertAtPosition(&head, value, position);
        break;

    case 4:
        deleteFromBeginning(&head);
        break;

    case 5:
        deleteFromEnd(&head);
        break;

    case 6:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        deleteFromPosition(&head, position);
        break;

    case 7:
        displayList(head);
        break;

    case 8:
        printf("Exiting...\n");
```

```
            return 0;


        default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   }


   return 0;
}
```

Output :-

```
Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display List
8. Exit
Enter your choice: 1
Enter value to insert at beginning: 10


Enter your choice: 1
Enter value to insert at beginning: 20
```

Enter your choice: 2

Enter value to insert at end: 30

Enter your choice: 7

Circular Doubly Linked List: 20 <-> 10 <-> 30 <-> (Back to Head)

Enter your choice: 6

Enter position to delete: 2

Circular Doubly Linked List: 20 <-> 30 <-> (Back to Head)

Enter your choice: 8

Exiting...

## Write Programs to implement the Stack operations using an array

```
#include <stdio.h>

#define MAX 100  // Maximum stack size


int stack[MAX];  // Stack array

int top = -1;    // Top of stack


// Function to push an element into the stack

void push(int value) {

   if (top == MAX - 1) {

      printf("Stack Overflow! Cannot push %d\n", value);

      return;

   }

   stack[++top] = value;

   printf("%d pushed into stack\n", value);

}


// Function to pop an element from the stack

void pop() {

   if (top == -1) {

      printf("Stack Underflow! No elements to pop\n");

      return;

   }

   printf("%d popped from stack\n", stack[top--]);

}
```

```
// Function to peek (view top element)
void peek() {
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Top element is %d\n", stack[top]);
}

// Function to display the stack
void display() {
    if (top == -1) {
        printf("Stack is empty!\n");
        return;
    }

    printf("Stack elements: ");
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

// Main function
```

```
int main() {
    int choice, value;

    while (1) {
        printf("\nOperations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display Stack\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;

            case 2:
                pop();
                break;

            case 3:
```

```
            peek();
            break;

        case 4:
            display();
            break;

        case 5:
            printf("Exiting...\n");
            return 0;

        default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   }

   return 0;
}
```

Output :-

Operations:

1. Push
2. Pop
3. Peek
4. Display Stack
5. Exit

Enter your choice: 1

Enter value to push: 10

10 pushed into stack

Enter your choice: 1

Enter value to push: 20

20 pushed into stack

Enter your choice: 4

Stack elements: 20 10

Enter your choice: 3

Top element is 20

Enter your choice: 2

20 popped from stack

Enter your choice: 4

Stack elements: 10

Enter your choice: 5

Exiting...

## Write a program using stacks to convert a given infix expression to postfix

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


#define MAX 100  // Maximum stack size


// Stack implementation

char stack[MAX];

int top = -1;


// Function to push an element onto the stack

void push(char ch) {

   if (top == MAX - 1) {

      printf("Stack Overflow!\n");

      return;

   }

   stack[++top] = ch;

}


// Function to pop an element from the stack

char pop() {

   if (top == -1) {

      return '\0';  // Return null character if stack is empty
```

```
    }
    return stack[top--];
}

// Function to get the precedence of an operator
int precedence(char ch) {
    switch (ch) {
        case '+': case '-': return 1;
        case '*': case '/': return 2;
        case '^': return 3;
        default: return 0;  // Lower precedence for operands and brackets
    }
}

// Function to convert infix to postfix
void infixToPostfix(char infix[]) {
    char postfix[MAX];  // To store the postfix expression
    int i, j = 0;

    for (i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];

        // If character is an operand, add it to postfix expression
        if (isalnum(ch)) {
            postfix[j++] = ch;
```

```
    }
    // If character is '(', push to stack
    else if (ch == '(') {
        push(ch);
    }
    // If character is ')', pop from stack until '(' is found
    else if (ch == ')') {
        while (top != -1 && stack[top] != '(') {
            postfix[j++] = pop();
        }
        pop();  // Remove '(' from stack
    }
    // If character is an operator
    else {
        while (top != -1 && precedence(stack[top]) >= precedence(ch)) {
            postfix[j++] = pop();
        }
        push(ch);
    }
}

// Pop remaining operators from the stack
while (top != -1) {
    postfix[j++] = pop();
}
```

```
    postfix[j] = '\0';  // Null terminate the string
    printf("Postfix Expression: %s\n", postfix);
}

// Main function
int main() {
    char infix[MAX];

    printf("Enter an infix expression: ");
    scanf("%s", infix);

    infixToPostfix(infix);

    return 0;
}
```

Output :-

```
Enter an infix expression: A+B*C
Postfix Expression: ABC*+

Enter an infix expression: (A+B)*C
Postfix Expression: AB+C*
```

## Write Programs to implement the Stack operations using Liked List.

```
#include <stdio.h>

#include <stdlib.h>


// Define Node structure

struct Node {

   int data;

   struct Node* next;

};


// Pointer to the top of the stack

struct Node* top = NULL;


// Function to push an element onto the stack

void push(int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   if (!newNode) {

      printf("Heap Overflow! Cannot push %d\n", value);

      return;

   }

   newNode->data = value;

   newNode->next = top;

   top = newNode;

   printf("%d pushed into stack\n", value);

}
```

```
// Function to pop an element from the stack
void pop() {
    if (top == NULL) {
        printf("Stack Underflow! No elements to pop\n");
        return;
    }
    struct Node* temp = top;
    printf("%d popped from stack\n", top->data);
    top = top->next;
    free(temp);
}

// Function to peek (view top element)
void peek() {
    if (top == NULL) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Top element is %d\n", top->data);
}

// Function to display the stack
void display() {
    if (top == NULL) {
```

```
        printf("Stack is empty!\n");
        return;
    }

    struct Node* temp = top;
    printf("Stack elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    int choice, value;

    while (1) {
        printf("\nOperations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display Stack\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
```

```
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter value to push: ");
      scanf("%d", &value);
      push(value);
      break;

   case 2:
      pop();
      break;

   case 3:
      peek();
      break;

   case 4:
      display();
      break;

   case 5:
      printf("Exiting...\n");
      return 0;
```

```
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Output :-

```
Operations:
1. Push
2. Pop
3. Peek
4. Display Stack
5. Exit
Enter your choice: 1
Enter value to push: 10
10 pushed into stack

Enter your choice: 1
Enter value to push: 20
20 pushed into stack

Enter your choice: 4
Stack elements: 20 -> 10 -> NULL
```

Enter your choice: 3

Top element is 20

Enter your choice: 2

20 popped from stack

Enter your choice: 4

Stack elements: 10 -> NULL

Enter your choice: 5

Exiting...

## Write Programs to implement the Queue operations using an array

```
#include <stdio.h>

#define MAX 100  // Maximum queue size


int queue[MAX];  // Queue array

int front = -1, rear = -1;  // Front and Rear pointers


// Function to enqueue (insert) an element into the queue

void enqueue(int value) {

   if (rear == MAX - 1) {

      printf("Queue Overflow! Cannot enqueue %d\n", value);

      return;

   }

   if (front == -1) {

      front = 0;

   }

   queue[++rear] = value;

   printf("%d enqueued into queue\n", value);

}


// Function to dequeue (remove) an element from the queue

void dequeue() {

   if (front == -1 || front > rear) {

      printf("Queue Underflow! No elements to dequeue\n");

      front = rear = -1;  // Reset queue when empty
```

```
        return;
    }
    printf("%d dequeued from queue\n", queue[front++]);
}

// Function to peek (view front element)
void peek() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Front element is %d\n", queue[front]);
}

// Function to display the queue
void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
        return;
    }

    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
```

```
    printf("\n");
}

// Main function
int main() {
    int choice, value;

    while (1) {
        printf("\nOperations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display Queue\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
```

```
            dequeue();
            break;

        case 3:
            peek();
            break;

        case 4:
            display();
            break;

        case 5:
            printf("Exiting...\n");
            return 0;

        default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   }

   return 0;
}
```

Output:-

Operations:

1. Enqueue

2. Dequeue

3. Peek

4. Display Queue

5. Exit

Enter your choice: 1

Enter value to enqueue: 10

10 enqueued into queue

Enter your choice: 1

Enter value to enqueue: 20

20 enqueued into queue

Enter your choice: 4

Queue elements: 10 20

Enter your choice: 3

Front element is 10

Enter your choice: 2
10 dequeued from queue

Enter your choice: 4
Queue elements: 20

Enter your choice: 5
Exiting...

## Write Programs to implement the Queue operations using Liked List.

```
#include <stdio.h>

#include <stdlib.h>


// Define Node structure

struct Node {

   int data;

   struct Node* next;

};


// Front and Rear pointers

struct Node *front = NULL, *rear = NULL;


// Function to enqueue (insert) an element into the queue

void enqueue(int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   if (!newNode) {

      printf("Heap Overflow! Cannot enqueue %d\n", value);

      return;

   }

   newNode->data = value;

   newNode->next = NULL;


   if (rear == NULL) {  // If queue is empty

      front = rear = newNode;
```

```
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d enqueued into queue\n", value);
}

// Function to dequeue (remove) an element from the queue
void dequeue() {
    if (front == NULL) {
        printf("Queue Underflow! No elements to dequeue\n");
        return;
    }

    struct Node* temp = front;
    printf("%d dequeued from queue\n", front->data);
    front = front->next;

    if (front == NULL) { // If queue becomes empty
        rear = NULL;
    }

    free(temp);
}
```

```
// Function to peek (view front element)
void peek() {
    if (front == NULL) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Front element is %d\n", front->data);
}

// Function to display the queue
void display() {
    if (front == NULL) {
        printf("Queue is empty!\n");
        return;
    }

    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

```
// Main function
int main() {
    int choice, value;

    while (1) {
        printf("\nOperations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display Queue\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                dequeue();
                break;
```

```
            case 3:
                peek();
                break;

            case 4:
                display();
                break;

            case 5:
                printf("Exiting...\n");
                return 0;

            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Output :-

Operations:

1. Enqueue
2. Dequeue
3. Peek
4. Display Queue

5. Exit

Enter your choice: 1

Enter value to enqueue: 10

10 enqueued into queue

Enter your choice: 1

Enter value to enqueue: 20

20 enqueued into queue

Enter your choice: 4

Queue elements: 10 -> 20 -> NULL

Enter your choice: 3

Front element is 10

Enter your choice: 2

10 dequeued from queue

Enter your choice: 4

Queue elements: 20 -> NULL

Enter your choice: 5

Exiting...

## Write a program for Binary Search Tree Traversals

```
#include <stdio.h>

#include <stdlib.h>


// Define Node structure

struct Node {

   int data;

   struct Node* left;

   struct Node* right;

};


// Function to create a new node

struct Node* createNode(int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->data = value;

   newNode->left = newNode->right = NULL;

   return newNode;

}


// Function to insert a node into BST

struct Node* insert(struct Node* root, int value) {

   if (root == NULL) {

      return createNode(value);

   }
```

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

    return root;
}

// Inorder Traversal (Left -> Root -> Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Preorder Traversal (Root -> Left -> Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
```

```
}

// Postorder Traversal (Left -> Right -> Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Main function
int main() {
    struct Node* root = NULL;
    int choice, value;

    while (1) {
        printf("\nOperations:\n");
        printf("1. Insert\n");
        printf("2. Inorder Traversal\n");
        printf("3. Preorder Traversal\n");
        printf("4. Postorder Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
switch (choice) {
    case 1:
        printf("Enter value to insert: ");
        scanf("%d", &value);
        root = insert(root, value);
        break;

    case 2:
        printf("Inorder Traversal: ");
        inorder(root);
        printf("\n");
        break;

    case 3:
        printf("Preorder Traversal: ");
        preorder(root);
        printf("\n");
        break;

    case 4:
        printf("Postorder Traversal: ");
        postorder(root);
        printf("\n");
        break;
```

```
            case 5:
                printf("Exiting...\n");
                return 0;

            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    }

    return 0;
}
```

Output :-

```
Operations:
1. Insert
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Exit
Enter your choice: 1
Enter value to insert: 50

Enter your choice: 1
Enter value to insert: 30
```

Enter your choice: 1

Enter value to insert: 70

Enter your choice: 1

Enter value to insert: 20

Enter your choice: 1

Enter value to insert: 40

Enter your choice: 2

Inorder Traversal: 20 30 40 50 70

Enter your choice: 3

Preorder Traversal: 50 30 20 40 70

Enter your choice: 4

Postorder Traversal: 20 40 30 70 50

Enter your choice: 5

Exiting...

## Write a program to search an item in a given list using the following Searching Algorithms

## a. Linear Search

## b. Binary Search.

```
#include <stdio.h>


// Function for Linear Search
int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key)
         return i;  // Return index if found
   }
   return -1;  // Return -1 if not found
}


// Function for Binary Search (array must be sorted)
int binarySearch(int arr[], int left, int right, int key) {
   while (left <= right) {
      int mid = left + (right - left) / 2;


      if (arr[mid] == key)
         return mid;  // Return index if found
      else if (arr[mid] < key)
         left = mid + 1;  // Search right half
      else
         right = mid - 1; // Search left half
```

```
    }
    return -1;  // Return -1 if not found
}

// Function to sort the array for Binary Search
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to display array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
// Main function
int main() {
    int n, key, choice, result;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n]; // Array declaration

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    while (1) {
        printf("\nSearch Methods:\n");
        printf("1. Linear Search\n");
        printf("2. Binary Search (Array will be sorted)\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to search: ");
```

```
    scanf("%d", &key);
    result = linearSearch(arr, n, key);
    if (result != -1)
        printf("Element found at index %d (position %d)\n", result, result + 1);
    else
        printf("Element not found!\n");
    break;

case 2:
    bubbleSort(arr, n);  // Sorting before Binary Search
    printf("Sorted array: ");
    displayArray(arr, n);

    printf("Enter the element to search: ");
    scanf("%d", &key);

    result = binarySearch(arr, 0, n - 1, key);
    if (result != -1)
        printf("Element found at index %d (position %d)\n", result, result + 1);
    else
        printf("Element not found!\n");
    break;

case 3:
    printf("Exiting...\n");
```

```
            return 0;


        default:
            printf("Invalid choice! Please try again.\n");
      }
   }


   return 0;
}
```

Output :-

```
Enter the number of elements: 5
Enter 5 elements: 12 5 8 20 15


Search Methods:
1. Linear Search
2. Binary Search (Array will be sorted)
3. Exit
Enter your choice: 1
Enter the element to search: 20
Element found at index 3 (position 4)


Enter your choice: 2
Sorted array: 5 8 12 15 20
Enter the element to search: 8
Element found at index 1 (position 2)

Enter your choice: 3
Exiting...
```

## Write a program for implementation of the following Sorting Algorithms

## a. Bubble Sort

## b. Insertion Sort

## c. Quick Sort

```
#include <stdio.h>


// Function to swap two elements
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}


// Bubble Sort
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            swap(&arr[j], &arr[j + 1]);
         }
      }
   }
}


// Insertion Sort
void insertionSort(int arr[], int n) {
```

```
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements that are greater than key to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Quick Sort Partition Function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]); // Place pivot in correct position
```

```
    return i + 1;
}

// Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to display an array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int n, choice;

    printf("Enter the number of elements: ");
```

```
scanf("%d", &n);

int arr[n], copyArr[n]; // Original & copy for multiple sorting
printf("Enter %d elements: ", n);
for (int i = 0; i < n; i++) {
   scanf("%d", &arr[i]);
   copyArr[i] = arr[i]; // Copy for later sorting
}

while (1) {
   printf("\nSorting Algorithms:\n");
   printf("1. Bubble Sort\n");
   printf("2. Insertion Sort\n");
   printf("3. Quick Sort\n");
   printf("4. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);

   // Restore original array before sorting
   for (int i = 0; i < n; i++) {
      arr[i] = copyArr[i];
   }

   switch (choice) {
      case 1:
```

```
            bubbleSort(arr, n);
            printf("Sorted array using Bubble Sort: ");
            displayArray(arr, n);
            break;

        case 2:
            insertionSort(arr, n);
            printf("Sorted array using Insertion Sort: ");
            displayArray(arr, n);
            break;

        case 3:
            quickSort(arr, 0, n - 1);
            printf("Sorted array using Quick Sort: ");
            displayArray(arr, n);
            break;

        case 4:
            printf("Exiting...\n");
            return 0;

        default:
            printf("Invalid choice! Please try again.\n");
    }
}
```

```
    return 0;
}
```

Output :-

```
Enter the number of elements: 5
Enter 5 elements: 12 5 8 20 15

Sorting Algorithms:
1. Bubble Sort
2. Insertion Sort
3. Quick Sort
4. Exit
Enter your choice: 1
Sorted array using Bubble Sort: 5 8 12 15 20

Enter your choice: 2
Sorted array using Insertion Sort: 5 8 12 15 20

Enter your choice: 3
Sorted array using Quick Sort: 5 8 12 15 20

Enter your choice: 4
Exiting...
```

# VIVA – QUESTIONS

Here's a **detailed introduction to Data Structures** along with **Viva Questions & Answers** to help you prepare.

---

## 📌 Introduction to Data Structures

### 🔷 Definition of Data Structure

A **data structure** is a **way of organizing, storing, and managing data** efficiently to perform operations like searching, sorting, inserting, and deleting.

Example: **Arrays, Linked Lists, Stacks, Queues, Trees, Graphs, etc.**

---

### 🔷 Types of Data Structures

1️⃣ **Linear Data Structures** – Elements are stored sequentially.

- **Array**
- **Linked List**
- **Stack**
- **Queue**

2️⃣ **Non-Linear Data Structures** – Elements are connected in a hierarchical or networked manner.

- **Trees (Binary Tree, BST, AVL Tree, etc.)**
- **Graphs (Directed, Undirected, Weighted, etc.)**

---

### 🔷 Abstract Data Types (ADT)

An **ADT (Abstract Data Type)** defines the behavior of a data structure **without specifying implementation details**.

✅ **Example:** Stack ADT (Operations: Push, Pop, Peek, IsEmpty, IsFull)

| Concept | Definition |
|---|---|
| **Data Type** | Specifies the kind of data (int, float, char, etc.). |

| Concept | Definition |
| --- | --- |
| **Abstract Data Type (ADT)** | Specifies the operations on data without implementation. |
| **Data Structure** | The actual implementation of ADT in programming (Arrays, Linked Lists, etc.). |

---

## 📌 Concept of Arrays

An **array** is a **collection of elements of the same data type** stored in contiguous memory locations.

- 🔹 **Single Dimensional Array:** A linear collection of elements (e.g., `arr[5]`).
- 🔹 **Two Dimensional Array:** A matrix-like collection (e.g., `arr[3][3]`).

---

## 📌 Operations on Arrays with Algorithms

### 🔷 Traversing an Array

#### Algorithm for Traversing an Array

```
void traverse(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}
```

---

### 🔷 Searching in an Array

#### ✅ Linear Search Algorithm

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

#### ✅ Binary Search Algorithm (for sorted arrays)

```
int binarySearch(int arr[], int left, int right, int key) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) return mid;
```

```
        else if (arr[mid] < key) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

---

### ◈ Insertion in an Array

#### ✅ Algorithm to Insert an Element at a Given Position

```
void insertElement(int arr[], int *n, int pos, int value) {
    for (int i = *n; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
    (*n)++;
}
```

---

### ◈ Deletion in an Array

#### ✅ Algorithm to Delete an Element from a Given Position

```
void deleteElement(int arr[], int *n, int pos) {
    for (int i = pos; i < *n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*n)--;
}
```

---

## 📌 Viva Questions & Answers

### ◈ General Questions on Data Structures

1️⃣ **Q:** What is a Data Structure?
**A:** A data structure is a way to store and organize data efficiently.

2️⃣ **Q:** What are the types of Data Structures?
**A:** Linear (Arrays, Linked Lists, Stacks, Queues) and Non-Linear (Trees, Graphs).

3️⃣ **Q:** What is the difference between ADT and Data Structure?
**A:** ADT defines only operations, while Data Structure is the actual implementation.

---

## ◈ Questions on Arrays

4 **Q:** What is an array?
**A:** An array is a collection of elements of the same type stored in contiguous memory.

5 **Q:** What is the difference between a 1D and 2D array?
**A:** A 1D array is a single list of elements, while a 2D array is a matrix with rows and columns.

6 **Q:** What is the time complexity of searching in an array?
**A:** Linear Search: **O(N)**, Binary Search: **O(log N)** (only for sorted arrays).

7 **Q:** How is an array stored in memory?
**A:** Arrays are stored in contiguous memory locations.

---

## ◈ Questions on Searching & Sorting

8 **Q:** What is the difference between Linear and Binary Search?
**A:** Linear Search works on **unsorted** data (**O(N)**), while Binary Search requires **sorted** data (**O(log N)**).

9 **Q:** What is the best sorting algorithm?
**A: Quick Sort (O(N log N))** is best for large datasets, but **Bubble Sort (O($N^2$))** is simpler.

10 **Q:** What is the worst-case time complexity of Quick Sort?
**A: O($N^2$)** when the pivot is always the smallest or largest element.

---

## ◈ Questions on Array Operations

11 **Q:** How do you insert an element in an array?
**A:** Shift elements to the right and insert at the desired position (**O(N)**).

12 **Q:** How do you delete an element from an array?
**A:** Shift elements to the left and reduce the size (**O(N)**).

13 **Q:** What is the time complexity of traversing an array?
**A: O(N)** (Each element is accessed once).

---

### 🔷 Advanced Questions

14️⃣ **Q:** What are the advantages of using an array?
**A:**

- **Fast access** using index (**O(1)**).
- **Memory efficiency** (no extra overhead).

15️⃣ **Q:** What are the disadvantages of arrays?
**A:**

- **Fixed size** (wastes memory if not fully used).
- **Insertion & Deletion is costly** (**O(N)**).

16️⃣ **Q:** What is the difference between a Static and Dynamic Array?
**A:**

- **Static Array**: Fixed size, allocated at compile time.
- **Dynamic Array**: Size can change, allocated at runtime (**e.g., malloc() in C**).

---

## 📌 Summary

✓ **Data Structures** help organize data efficiently.
✓ **Arrays** are the simplest data structures with **fast access but costly insertions/deletions**.
✓ **Searching Algorithms** – Linear Search (**O(N)**), Binary Search (**O(log N)**).
✓ **Sorting Algorithms** – Bubble Sort (**$O(N^2)$**), Quick Sort (**O(N log N)**).
✓ **Viva Questions** cover basics, implementations, and efficiency comparisons.

Would you like more **Viva Questions** on **Linked Lists, Stacks, or Trees**? 😊

Here's a **detailed guide** on **Linked Lists** along with **Viva Questions & Answers** to help you prepare.

---

## 📌 Introduction to Linked Lists

### 🔷 What is a Linked List?

A **Linked List** is a **linear data structure** where elements (nodes) are **linked using pointers**.

Each **node** consists of:

1. **Data** (Stores the actual value)
2. **Pointer (Next)** (Points to the next node)

📌 Unlike arrays, linked lists **don't require contiguous memory allocation**.

---

## 📌 Representation of Linked List in Memory

Linked lists use **dynamic memory allocation** (`malloc()` in C) to create nodes at runtime.

Example structure in C:

```
struct Node {
    int data;
    struct Node* next; // Pointer to the next node
};
```

---

## 📌 Comparison: Linked List vs. Array

| Feature | Array | Linked List |
|---|---|---|
| **Memory Allocation** | Contiguous (fixed size) | Non-contiguous (dynamic size) |
| **Insertion/Deletion** | Costly (`O(N)`) | Efficient (`O(1)` at head) |
| **Access Time** | Fast (`O(1)`) | Slow (`O(N)`) |
| **Wastage** | Can waste memory (if not fully used) | No wastage |

---

## 📌 Types of Linked Lists

### 1️⃣ Singly Linked List

Each node has **one pointer** to the next node.

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

🔹 **Traversal: Forward only**
🔹 **Insertion/Deletion: Easy at head, harder at other positions**

---

### 2️⃣ Doubly Linked List (DLL)

Each node has **two pointers**: `prev` (to previous node) and `next` (to next node).

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

- 🔹 **Traversal: Both Forward & Backward**
- 🔹 **Insertion/Deletion: More efficient than Singly LL**

---

### 3️⃣ Circular Singly Linked List

- 🔹 The **last node** points back to the **first node**.
- 🔹 **Used in:** CPU Scheduling, Multiplayer Games.

---

### 4️⃣ Circular Doubly Linked List

- 🔹 The **last node** points to **first**, and **first node** points to **last**.
- 🔹 **Used in:** Music Playlists, Web Browsers (Back/Forward navigation).

---

## 📌 Implementation of Linked List ADT

### ✅ Creating a Linked List

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

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

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

---

## ✅ Traversing a Linked List

```
void traverse(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

---

## ✅ Searching in a Linked List

```
int search(struct Node* head, int key) {
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data == key) return 1;
        temp = temp->next;
    }
    return 0;
}
```

---

## ✅ Insertion Operations

### *Insert at Beginning*

```
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}
```

### *Insert at a Given Position*

```
void insertAtPosition(struct Node** head, int data, int position) {
    struct Node* newNode = createNode(data);
    if (position == 0) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    for (int i = 0; i < position - 1 && temp != NULL; i++)
        temp = temp->next;

    if (temp == NULL) return;
    newNode->next = temp->next;
    temp->next = newNode;
}
```

### *Insert at End*

```
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
```

```
    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
}
```

---

## ☑ Deletion Operations

### *Delete from Beginning*

```
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) return;
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

### *Delete from a Given Position*

```
void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) return;

    struct Node* temp = *head;
    if (position == 0) {
        *head = temp->next;
        free(temp);
        return;
    }

    struct Node* prev = NULL;
    for (int i = 0; i < position && temp != NULL; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;
    prev->next = temp->next;
    free(temp);
}
```

### *Delete from End*

```
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) return;

    struct Node* temp = *head;
    struct Node* prev = NULL;

    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

    if (prev == NULL) {
        *head = NULL;
    } else {
        prev->next = NULL;
    }
```

```
    free(temp);
}
```

---

## 📌 Applications of Linked Lists

✓ **Dynamic Memory Allocation** (OS uses linked lists for heap management).
✓ **Implementing Stack & Queue**.
✓ **Music Playlist & Web Browser Back/Forward navigation**.
✓ **Graph Representation** (Adjacency List).
✓ **Undo/Redo functionality in Software**.

---

## 📌 Viva Questions & Answers

1️⃣ **Q:** What is a Linked List?
**A:** A Linked List is a **dynamic data structure** where elements are connected using pointers.

2️⃣ **Q:** What is the difference between Array and Linked List?
**A:** Arrays use **contiguous memory**, while Linked Lists use **dynamic memory with pointers**.

3️⃣ **Q:** What are the advantages of Linked Lists over Arrays?
**A:** Dynamic size, Efficient insertion/deletion, No memory wastage.

4️⃣ **Q:** What are the types of Linked Lists?
**A:** Singly, Doubly, Circular Singly, Circular Doubly.

5️⃣ **Q:** What is a Circular Linked List?
**A:** A Linked List where the **last node points to the first node**.

6️⃣ **Q:** What is the time complexity of insertion at the beginning of a Linked List?
**A: O(1)** (Constant time).

7️⃣ **Q:** How can a Linked List be used to implement a Stack?
**A:** Using `push()` for insertion at **head** and `pop()` for deletion from **head**.

---

## 📌 Summary

✓ **Linked Lists** are efficient for **dynamic memory allocation**.
✓ **Types:** Singly, Doubly, Circular LLs.

✓ **Operations:** Insertion, Deletion, Searching, Traversal.
✓ **Applications:** OS, Graphs, Playlists, Undo/Redo.

Would you like me to add **more viva questions** on **Stacks, Queues, or Trees**? 😊

Here's a **detailed guide** on **Stacks & Queues** along with **Viva Questions & Answers** to help you prepare. 🚀

---

# 📌 UNIT-III: STACKS & QUEUES

## 🔹 STACKS

### 🔹 Introduction to Stack ADT

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

#### 🔷 Basic Operations:

1. **Push** → Insert an element at the top
2. **Pop** → Remove the top element
3. **Peek (Top)** → Get the top element without removing it
4. **isEmpty()** → Check if the stack is empty
5. **isFull()** → Check if the stack is full (for array implementation)

---

### 🔹 Representation of Stacks

*1️⃣ Using an Array*

```
#define MAX 100
int stack[MAX];
int top = -1;

void push(int data) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = data;
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}
```

🔷 **Pros:** Easy implementation, fast access
🔷 **Cons:** Fixed size, memory wastage

---

### 2️⃣ *Using a Linked List*

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

struct Node* top = NULL;

void push(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = top;
    top = newNode;
}

int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        return -1;
    }
    int data = top->data;
    struct Node* temp = top;
    top = top->next;
    free(temp);
    return data;
}
```

🔷 **Pros:** Dynamic size, no memory wastage
🔷 **Cons:** Extra memory for pointers

---

## 🔹 Applications of Stacks

✓ **Reversing Strings**
✓ **Backtracking (Maze Solving, Sudoku Solver)**
✓ **Function Call Stack (Recursion)**
✓ **Undo/Redo Operations in Text Editors**
✓ **Expression Evaluation (Postfix, Infix to Postfix Conversion)**

---

## ◈ Polish Notations

📌 **Infix Expression** → Operators are in between operands (e.g., `A + B`).
📌 **Postfix Expression** → Operators are **after** operands (`AB+`).
📌 **Prefix Expression** → Operators are **before** operands (`+AB`).

### ✔ *Infix to Postfix Conversion (Using Stack)*

```
// Function to convert infix to postfix using stack
int precedence(char ch) {
    if (ch == '+' || ch == '-') return 1;
    if (ch == '*' || ch == '/') return 2;
    return 0;
}

void infixToPostfix(char* infix) {
    char postfix[100];
    int j = 0;
    Stack s;

    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        if (isalnum(ch))
            postfix[j++] = ch;
        else if (ch == '(')
            push(&s, ch);
        else if (ch == ')') {
            while (!isEmpty(&s) && peek(&s) != '(')
                postfix[j++] = pop(&s);
            pop(&s);
        } else {
            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(ch))
                postfix[j++] = pop(&s);
            push(&s, ch);
        }
    }

    while (!isEmpty(&s))
        postfix[j++] = pop(&s);

    postfix[j] = '\0';
    printf("Postfix Expression: %s\n", postfix);
}
```

---

## ◈ Tower of Hanoi (Using Recursion)

**Problem:** Move `n` disks from **Source** to **Destination** using an **Auxiliary** peg.
**Rules:**
1. Only one disk can be moved at a time.
2. A larger disk **cannot** be placed on a smaller disk.

```
void towerOfHanoi(int n, char source, char aux, char dest) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, dest);
        return;
    }
    towerOfHanoi(n - 1, source, dest, aux);
    printf("Move disk %d from %c to %c\n", n, source, dest);
    towerOfHanoi(n - 1, aux, source, dest);
}
```

---

### ◈ Recursion vs. Iteration

| Feature | Recursion | Iteration |
|---|---|---|
| **Speed** | Slower (function calls overhead) | Faster |
| **Memory** | Uses stack (extra memory) | Less memory |
| **Complexity** | Easy for tree/graph problems | Hard to implement |

---

# 📌 QUEUES

### ◈ Introduction to Queue ADT

A **Queue** is a **FIFO (First In, First Out)** structure.

#### 🔷 Basic Operations:

1. **Enqueue** → Insert element at rear
2. **Dequeue** → Remove element from front
3. **Peek** → Get front element without removing
4. **isEmpty()** → Check if queue is empty
5. **isFull()** → Check if queue is full (for array implementation)

---

### ◈ Representation of Queues

*1️⃣ Using an Array*

```
#define MAX 100
int queue[MAX];
int front = -1, rear = -1;

void enqueue(int data) {
    if (rear == MAX - 1) {
        printf("Queue Overflow\n");
        return;
    }
```

```
    if (front == -1) front = 0;
    queue[++rear] = data;
}

int dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return -1;
    }
    return queue[front++];
}
```

---

### 2️⃣ *Using a Linked List*

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

struct Node *front = NULL, *rear = NULL;

void enqueue(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
        return;
    }
    rear->next = newNode;
    rear = newNode;
}

int dequeue() {
    if (front == NULL) return -1;
    struct Node* temp = front;
    front = front->next;
    free(temp);
    if (front == NULL) rear = NULL;
}
```

---

### 🔷 Types of Queues

✓ **Circular Queue** → Last node is connected to first.
✓ **Dequeue (Double-Ended Queue)** → Insert & delete from both ends.
✓ **Priority Queue** → Elements with higher priority dequeued first.

---

## 📌 Viva Questions & Answers

### 🔑 Stacks

1️⃣ **Q:** What is a stack?
**A:** A **LIFO** data structure where insertion & deletion occur at **top**.

2️⃣ **Q:** What are the applications of stacks?
**A:** Expression evaluation, Function call stack, Undo/Redo.

3️⃣ **Q:** How do you convert an infix expression to postfix?
**A:** Use a **stack** to handle operator precedence.

---

### 🔑 Queues

1️⃣ **Q:** What is a queue?
**A:** A **FIFO** data structure where insertion happens at **rear** and deletion at **front**.

2️⃣ **Q:** What are the types of queues?
**A:** Simple Queue, Circular Queue, Priority Queue, Dequeue.

3️⃣ **Q:** Where are queues used?
**A: CPU Scheduling, Print Spooling, Call Center Systems**.

---

🔥 Would you like more viva questions on **Recursion, Priority Queue, or Sorting Algorithms?** 😊

# 📌 UNIT-IV: SEARCHING & SORTING

A complete guide with **concepts, programs, and viva questions**! 🚀

---

## 🔷 SEARCHING ALGORITHMS

### 🔷 1. Linear Search (Sequential Search)

**📌 Concept:**

- Checks each element **one by one** until the target is found.

- Works for **unsorted arrays**.

### 📌 Time Complexity:

- **Best Case:** `O(1)` (First element is the target)
- **Worst Case:** `O(N)` (Last element or not present)

### 📌 C Program for Linear Search

```
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) return i; // Element found
    }
    return -1; // Not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30, n = 5;
    int index = linearSearch(arr, n, key);
    if (index != -1)
        printf("Element found at index %d\n", index);
    else
        printf("Element not found\n");
    return 0;
}
```

---

## 🔷 2. Binary Search

### 📌 Concept:

- **Works only on sorted arrays**.
- Repeatedly divides the search interval **by half**.
- **Middle element** is compared with the target.

### 📌 Time Complexity:

- **Best Case:** `O(1)` (Middle element is the target)
- **Worst Case:** `O(log N)` (Repeated halving of array)

### 📌 C Program for Binary Search

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int key) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
```

```
        if (arr[mid] == key) return mid; // Found
        else if (arr[mid] < key) left = mid + 1; // Search right half
        else right = mid - 1; // Search left half
    }
    return -1; // Not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int key = 30, n = 5;
    int index = binarySearch(arr, 0, n - 1, key);
    if (index != -1)
        printf("Element found at index %d\n", index);
    else
        printf("Element not found\n");
    return 0;
}
```

---

### 🔷 3. Indexed Sequential Search

#### 📌 Concept:

- Improves **Linear Search** by **dividing data into blocks**.
- Index stores the **maximum values** for each block.
- First, searches in **index**, then searches in the **block**.

#### 📌 Time Complexity:

- **Worst Case:** $O(\sqrt{N})$ (For large datasets, faster than Linear Search)

---

## 🔷 SORTING ALGORITHMS

### 🔷 1. Selection Sort

#### 📌 Concept:

- Selects the **smallest element** in each pass and swaps it.
- **Unstable** sorting algorithm.

#### 📌 Time Complexity:

- **Best & Worst Case:** $O(N^2)$

#### 📌 C Program for Selection Sort

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

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) minIdx = j;
        }
        int temp = arr[i];
        arr[i] = arr[minIdx];
        arr[minIdx] = temp;
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = 5;
    selectionSort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

---

## 🔷 2. Bubble Sort

### 📌 Concept:

- Repeatedly swaps adjacent elements if they are in the wrong order.

### 📌 Time Complexity:

- **Best Case:** $O(N)$ (Already sorted)
- **Worst Case:** $O(N^2)$

### 📌 C Program for Bubble Sort

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22};
    int n = 5;
    bubbleSort(arr, n);
```

```
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

---

## 🔷 3. Insertion Sort

### 📌 Concept:

- Picks an element and **places it in the correct position**.

### 📌 Time Complexity:

- **Best Case:** $O(N)$
- **Worst Case:** $O(N^2)$

### 📌 C Program for Insertion Sort

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i], j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22};
    int n = 5;
    insertionSort(arr, n);
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

---

## 🔷 4. Quick Sort

### 📌 Concept:

- **Pivot element** is chosen, and elements are partitioned into two halves.
- **Fastest sorting algorithm for large datasets**.

### 📌 Time Complexity:

- **Best & Avg Case:** $O(N \log N)$
- **Worst Case:** $O(N^2)$ (When pivot is smallest/largest)

### 📌 C Program for Quick Sort

```
#include <stdio.h>

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = arr[high], i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
        int pi = i + 1;
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

---

## ◈ 5. Merge Sort

### 📌 Concept:

- Divides array into two halves and merges them.
- **Stable sorting algorithm**.

### 📌 Time Complexity:

- **Best & Worst Case:** $O(N \log N)$

---

# 📌 Viva Questions & Answers

1⃣ **Q:** What is the difference between Linear & Binary Search?
**A:** Linear Search works on **unsorted data**, Binary Search works on **sorted data**.

2⃣ **Q:** Why is Quick Sort faster than Bubble Sort?
**A:** Quick Sort has $O(N \log N)$ complexity, while Bubble Sort is $O(N^2)$.

3⃣ **Q:** What is a stable sorting algorithm?
**A:** It maintains the relative order of equal elements (e.g., Merge Sort, Insertion Sort).

---

🔥 Do you want more **examples** or **advanced viva questions**? 😊

# 📌 UNIT-V: BINARY TREES & GRAPHS

A complete guide with **concepts, programs, and viva questions**! 🌳📎

---

## 🔷 BINARY TREES

### 🔷 1. Concept of Non-Linear Data Structures

**📌 Definition:**

- Unlike arrays, stacks, and queues (which are **linear**), **non-linear** data structures allow multiple connections between elements (e.g., **Trees** and **Graphs**).
- **Examples:** Binary Trees, BST, AVL Tree, Graphs, Heap, B-Trees.

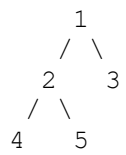
---

### 🔷 2. Introduction to Binary Trees

**📌 Definition:**
A **Binary Tree** is a tree where **each node has at most two children**:

- **Left Child**
- **Right Child**

**📌 Example:**




---

### 🔷 3. Types of Trees

1️⃣ **Full Binary Tree:** Every node has **0 or 2 children**.
2️⃣ **Complete Binary Tree:** All levels are completely filled **except the last one**.
3️⃣ **Perfect Binary Tree:** All **leaf nodes** are at the same level.
4️⃣ **Balanced Binary Tree:** Difference between **left & right subtree heights** is at most `1`.
5️⃣ **Binary Search Tree (BST):** Left child < Root < Right child.

---

## 🔷 4. Properties of Binary Trees

📌 **1. Maximum Nodes:** At level `L`, max nodes = `2^L - 1`.
📌 **2. Minimum Height:** $\log_2(N + 1) - 1$.
📌 **3. Leaf Nodes:** `N = Internal Nodes + 1`.

---

## 🔷 5. Representation of Binary Trees

📌 **1. Using Arrays** (for Complete Binary Trees)
📌 **2. Using Linked Lists** (Dynamic Allocation)

### 🔹 C Code for Binary Tree Representation Using Linked List

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

struct Node {
    int data;
    struct Node *left, *right;
};

// Creating a new node
struct Node* newNode(int value) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = value;
    temp->left = temp->right = NULL;
    return temp;
}

int main() {
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    printf("Root Node: %d\n", root->data);
    return 0;
}
```

---

## 🔷 6. Binary Search Tree (BST) Operations

📌 **1. Insertion**
📌 **2. Deletion**
📌 **3. Searching**

### 🔷 C Code for Insertion in BST

```
struct Node* insert(struct Node* root, int key) {
    if (root == NULL) return newNode(key);
    if (key < root->data) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    return root;
}
```

---

## ◈ 7. Binary Tree Traversal

📌 **1. Inorder (LNR)** → Left → Root → Right
📌 **2. Preorder (NLR)** → Root → Left → Right
📌 **3. Postorder (LRN)** → Left → Right → Root
📌 **4. Level Order** (BFS - Using Queue)

### 🔷 C Code for Inorder Traversal

```
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

---

## ◈ 8. Applications of Binary Trees

✅ Expression Trees (Compiler Design)
✅ Binary Search Trees (Fast Searching)
✅ Huffman Coding (Compression)
✅ Heap Data Structure (Priority Queues)

---

# ◈ GRAPHS

## ◈ 1. Introduction to Graphs

### 📌 Definition:
A **graph** is a set of **nodes (vertices)** connected by **edges (links)**.

- **Types:** Directed, Undirected, Weighted, Unweighted.

### 📌 Example:

```
A ---- B
|      |
C ---- D
```

---

## 🔷 2. Terms in Graphs

📌 **1. Vertex (V)** → A point (node).
📌 **2. Edge (E)** → A connection between vertices.
📌 **3. Adjacency List** → Stores neighbors of a vertex.
📌 **4. Degree of Vertex** → Number of edges touching a vertex.
📌 **5. Path** → Sequence of vertices from source to destination.

---

## 🔷 3. Representation of Graphs

📌 **1. Adjacency Matrix (Sequential Representation)**
📌 **2. Adjacency List (Linked Representation)**

### 🔹 C Code for Adjacency Matrix

```
#define V 4

void printGraph(int graph[V][V]) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = { {0, 1, 1, 0},
                        {1, 0, 1, 1},
                        {1, 1, 0, 1},
                        {0, 1, 1, 0} };
    printGraph(graph);
    return 0;
}
```

---

## 🔷 4. Graph Traversal Algorithms

📌 **1. Depth-First Search (DFS)**
📌 **2. Breadth-First Search (BFS)**

### 🔷 C Code for DFS

```
void DFS(int v, int visited[], int graph[V][V]) {
    visited[v] = 1;
    printf("%d ", v);
    for (int i = 0; i < V; i++) {
        if (graph[v][i] == 1 && !visited[i]) {
            DFS(i, visited, graph);
        }
    }
}
```

---

## 🔷 5. Applications of Graphs

✅ Google Maps (Shortest Path)
✅ Social Networks (Facebook, LinkedIn)
✅ Internet Routing (TCP/IP)
✅ AI Pathfinding (A* Algorithm)

---

## 📌 Viva Questions & Answers

### Binary Trees

1️⃣ **Q:** What is a Binary Tree?
**A:** A tree where each node has **at most 2 children**.

2️⃣ **Q:** What is the difference between a Binary Tree and a BST?
**A:** BST maintains a sorted order: `Left < Root < Right`.

3️⃣ **Q:** What are the types of Binary Tree traversal?
**A:** Inorder, Preorder, Postorder, Level Order.

---

### Graphs

4️⃣ **Q:** What is the difference between BFS and DFS?
**A:** BFS explores **level by level**, DFS explores **depth first**.

5️⃣ **Q:** What are the ways to represent a Graph?
**A:** Adjacency Matrix & Adjacency List.

6️⃣ **Q:** What is the use of graphs in real life?
**A:** Used in **navigation, networks, AI, and social media**.

# IMPORTANT QUESTIONS

## 📌 UNIT-I: Important Questions (5 Marks & 10 Marks)

---

### 🔷 5-Marks Questions (Short Answer Type)

1. **Define Data Structures. What are its types?**
2. **Differentiate between Linear and Non-Linear Data Structures.**
3. **What is an Abstract Data Type (ADT)? Give an example.**
4. **How is an ADT different from a Data Type and a Data Structure?**
5. **Explain the concept of an array with an example.**
6. **What are the advantages and disadvantages of arrays?**
7. **Explain single-dimensional and two-dimensional arrays with an example.**
8. **Write an algorithm for traversing an array.**
9. **Write an algorithm for inserting an element into an array.**
10. **Explain the concept of searching in an array with an example.**

---

### 🔷 10-Marks Questions (Long Answer Type)

1. **Explain the types of Data Structures with examples.**
2. **What is an ADT? Explain Stack ADT and Queue ADT with operations.**
3. **Describe the differences between ADT, Data Types, and Data Structures with examples.**
4. **Explain the concept of arrays in detail. Discuss its types and memory representation.**
5. **Write and explain algorithms for different array operations (searching, traversing, inserting, and deleting).**
6. **Explain Linear Search and Binary Search with algorithms and examples.**
7. **Discuss the advantages and disadvantages of arrays over linked lists.**
8. **Write an algorithm for insertion and deletion in a one-dimensional array and explain it with an example.**
9. **Explain two-dimensional arrays in detail with an example and memory representation.**
10. **Compare and contrast static and dynamic data structures with examples.**

---

## 📌 UNIT-II: Important Questions (5 Marks & 10 Marks)

---

### 🔷 5-Marks Questions (Short Answer Type)

1. **Define a linked list. How is it different from an array?**
2. **Explain the memory representation of a linked list.**

3. **List the types of linked lists with a brief description.**
4. **Differentiate between a singly linked list and a doubly linked list.**
5. **What is a circular linked list? How is it different from a normal linked list?**
6. **What are the advantages and disadvantages of linked lists?**
7. **Explain the concept of a self-referential structure in a linked list.**
8. **Write an algorithm for traversing a singly linked list.**
9. **How do you insert a node at the beginning of a linked list?**
10. **What are the real-world applications of linked lists?**

---

## 🔷 10-Marks Questions (Long Answer Type)

1. **Explain the differences between arrays and linked lists with examples.**
2. **Describe the types of linked lists (Singly, Doubly, Circularly Singly, Circularly Doubly) with diagrams.**
3. **Explain the memory representation of linked lists and the role of pointers in linked lists.**
4. **Write and explain the algorithm for searching an element in a linked list.**
5. **Write an algorithm for insertion and deletion at different positions in a linked list.**
6. **Explain how to insert a node at the first position, a specified position, and the last position in a linked list with algorithms.**
7. **Discuss the operations of a doubly linked list with insertion and deletion algorithms.**
8. **Explain circular linked lists in detail with insertion and deletion operations.**
9. **Write a program to implement a singly linked list with insertion, deletion, and traversal operations.**
10. **Describe the applications of linked lists in real-world scenarios.**

---

## 📌 UNIT-III: Important Questions (5 Marks & 10 Marks)

---

## 🔷 5-Marks Questions (Short Answer Type)

Stacks:

1. **Define a Stack and explain its ADT.**
2. **What are the applications of Stacks?**
3. **Explain Stack representation using an array.**
4. **Explain Stack representation using a linked list.**
5. **What are Polish Notations? Explain Infix, Prefix, and Postfix notation.**
6. **Write an algorithm to convert an infix expression to postfix.**
7. **Explain the algorithm to evaluate a postfix expression.**
8. **What is the Tower of Hanoi problem? Explain its significance.**

9. **What is recursion? Give an example.**
10. **Differentiate between recursion and iteration.**

### Queues:

1. **Define a Queue and explain its ADT.**
2. **What are the applications of Queues?**
3. **Explain Queue representation using an array.**
4. **Explain Queue representation using a linked list.**
5. **What is a Circular Queue? How is it different from a normal queue?**
6. **Define a Dequeue and its applications.**
7. **What is a Priority Queue? Where is it used?**
8. **Write an algorithm for inserting an element in a Queue.**
9. **Write an algorithm for deleting an element from a Queue.**
10. **Compare Stack and Queue with examples.**

---

## ◈ 10-Marks Questions (Long Answer Type)

### Stacks:

1. **Explain Stack ADT in detail with operations (Push, Pop, Peek).**
2. **Describe the array and linked list representations of Stacks with examples.**
3. **Write an algorithm and program to implement Stack operations using an array.**
4. **Write an algorithm and program to implement Stack operations using a linked list.**
5. **Explain the conversion of Infix to Postfix notation with an example.**
6. **Describe the evaluation of a Postfix expression with an example and algorithm.**
7. **Explain the Tower of Hanoi problem and solve it using recursion.**
8. **Differentiate between recursion and iteration with examples and advantages.**

### Queues:

1. **Explain Queue ADT in detail with operations (Enqueue, Dequeue, Peek).**
2. **Describe the array and linked list representations of Queues with examples.**
3. **Write an algorithm and program to implement Queue operations using an array.**
4. **Write an algorithm and program to implement Queue operations using a linked list.**
5. **What is a Circular Queue? Explain its operations with an example and algorithm.**
6. **Discuss the types of Queues: Circular Queue, Dequeue, and Priority Queue with applications.**
7. **Explain the advantages and disadvantages of different Queue types.**
8. **Compare Stack and Queue based on their working principles and applications.**

---

## 📌 UNIT-IV: Important Questions (5 Marks & 10 Marks)

---

### 🔷 5-Marks Questions (Short Answer Type)

Searching:

1. **Define Searching. What are its types?**
2. **Explain the working of Linear Search with an example.**
3. **Explain the working of Binary Search with an example.**
4. **What is Indexed Sequential Search? How is it different from Binary Search?**
5. **Compare Linear Search and Binary Search in terms of efficiency.**
6. **What is the best, worst, and average-case time complexity of Linear Search?**
7. **What is the best, worst, and average-case time complexity of Binary Search?**
8. **In which conditions is Binary Search preferred over Linear Search?**
9. **Write an algorithm for Linear Search.**
10. **Write an algorithm for Binary Search.**

Sorting:

1. **What is Sorting? Why is it important?**
2. **Explain the Selection Sort algorithm briefly.**
3. **Explain the Bubble Sort algorithm briefly.**
4. **Explain the Insertion Sort algorithm briefly.**
5. **What is Quick Sort? Explain its basic concept.**
6. **What is Merge Sort? Explain its basic concept.**
7. **Compare Selection Sort, Bubble Sort, and Insertion Sort based on efficiency.**
8. **What is the best, worst, and average-case time complexity of Quick Sort?**
9. **What is the best, worst, and average-case time complexity of Merge Sort?**
10. **Write an algorithm for Selection Sort.**

---

### 🔷 10-Marks Questions (Long Answer Type)

Searching:

1. **Explain Linear Search and Binary Search in detail with algorithms and examples.**
2. **Describe Indexed Sequential Search and compare it with Linear and Binary Search.**
3. **Compare Linear and Binary Search based on time complexity and efficiency.**
4. **Write a program to implement Linear and Binary Search in C.**
5. **Explain the importance of searching in data structures and real-world applications.**

### Sorting:

1. **Explain Selection Sort with a step-by-step example and algorithm.**
2. **Explain Bubble Sort with a step-by-step example and algorithm.**
3. **Explain Insertion Sort with a step-by-step example and algorithm.**
4. **Describe Quick Sort in detail, including its partitioning logic and recursive nature.**
5. **Describe Merge Sort in detail, including its divide-and-conquer approach.**
6. **Compare Selection Sort, Bubble Sort, and Insertion Sort in terms of efficiency and applications.**
7. **Write a program to implement Selection Sort, Bubble Sort, and Insertion Sort in C.**
8. **Write a program to implement Quick Sort and Merge Sort in C.**
9. **Compare Merge Sort and Quick Sort based on performance and applications.**
10. **Discuss the practical applications of different sorting algorithms in real-world scenarios.**

---

## 📌 UNIT-V: Important Questions (5 Marks & 10 Marks)

---

## 🔹 5-Marks Questions (Short Answer Type)

### Binary Trees:

1. **What is a Binary Tree? Explain with an example.**
2. **Differentiate between Linear and Non-Linear Data Structures.**
3. **What are the different types of Trees in Data Structures?**
4. **List and explain the properties of Binary Trees.**
5. **Explain the different ways of representing a Binary Tree in memory.**
6. **What is a Binary Search Tree (BST)? Explain with an example.**
7. **List the different operations that can be performed on a Binary Search Tree.**
8. **Explain Preorder, Inorder, and Postorder Traversal in Binary Trees.**
9. **What are the applications of Binary Trees?**
10. **Compare Binary Trees and Binary Search Trees.**

### Graphs:

1. **Define Graph and explain its components.**
2. **What are Directed and Undirected Graphs? Give examples.**
3. **Explain different ways of representing a Graph in memory.**
4. **Compare the Sequential and Linked Representation of Graphs.**
5. **What is Depth-First Search (DFS)? Explain briefly.**
6. **What is Breadth-First Search (BFS)? Explain briefly.**
7. **Compare DFS and BFS in terms of efficiency and applications.**

8. **List the different applications of Graphs in real life.**
9. **What is a Weighted Graph? Give an example.**
10. **Differentiate between Trees and Graphs.**

---

## ◈ 10-Marks Questions (Long Answer Type)

### Binary Trees:

1. **Explain Binary Trees in detail with examples and their types.**
2. **Discuss the properties of Binary Trees with mathematical proof.**
3. **Explain the different ways of representing Binary Trees in memory with examples.**
4. **Describe the operations of a Binary Search Tree (Insertion, Deletion, Searching) with examples.**
5. **Write a program to implement Binary Search Tree operations in C.**
6. **Explain the different types of Binary Tree traversals with algorithms and examples.**
7. **Compare Preorder, Inorder, and Postorder Traversals with examples.**
8. **Discuss the real-world applications of Binary Trees in computing.**
9. **What are Balanced Binary Trees? Explain AVL Trees and their role in maintaining balance.**
10. **Compare Binary Trees, Binary Search Trees, and AVL Trees with advantages and disadvantages.**

### Graphs:

1. **Explain Graphs in detail, along with their types and representations.**
2. **Describe the terms associated with Graphs such as Degree, Path, Cycle, Connected Graph, etc.**
3. **Compare and contrast the Sequential and Linked Representations of Graphs.**
4. **Explain Depth-First Search (DFS) in detail with an algorithm and example.**
5. **Explain Breadth-First Search (BFS) in detail with an algorithm and example.**
6. **Compare DFS and BFS in terms of efficiency, working mechanism, and applications.**
7. **Write a program to implement Graph traversal using DFS and BFS in C.**
8. **Discuss the real-world applications of Graphs in networking, social media, and AI.**
9. **What is a Minimum Spanning Tree (MST)? Explain Prim's and Kruskal's algorithms.**
10. **Compare Trees and Graphs based on their structure, properties, and applications.**

---