1. Introduction to Data Structures

1.1 Definition and Importance

A **data structure** is a systematic way of organizing and storing data to enable efficient access and modification.

Examples: Arrays, Lists, Stacks, Queues, Trees, Graphs, Hash Tables.

1.2 Classification of Data Structures

1. **Primitive Data Structures** – Integer, Float, Character, Boolean.
2. **Non-Primitive Data Structures**
    - **Linear:** Array, Stack, Queue, Linked List.
    - **Non-Linear:** Tree, Graph.
    - **Hash-based:** Hash Tables.

1.3 Abstract Data Types (ADTs)

An **ADT** defines what operations are possible on a data type without specifying how these operations are implemented.
Examples: Stack ADT, Queue ADT, List ADT.

2. Fundamental Data Structures

2.1 Arrays

An **array** is a collection of elements, each identified by an index or key.

- **Characteristics:**
    - Fixed size
    - Homogeneous elements
    - Random access

Example (C++):

```
int numbers[5] = {10, 20, 30, 40, 50};
```

Applications:

- Storing matrices
- Implementing other structures (lists, stacks)

2.2 Records (Structures)

A **record** is a collection of fields (possibly of different types).

Example:

```
struct Student {
    string name;
    int age;
    float gpa;
};
```

3. Implementation of Stacks, Queues, and Lists

3.1 Stack

- **Definition:** A **Last In, First Out (LIFO)** data structure.
- **Operations:**
  - `push(x)`: Insert element `x`
  - `pop()`: Remove top element
  - `peek()`: Retrieve top without removing

Implementation using Array:

```
int stack[SIZE];
int top = -1;
```

Applications:

- Function calls
- Expression evaluation
- Undo mechanisms

3.2 Queue

- **Definition:** A **First In, First Out (FIFO)** data structure.
- **Operations:** `enqueue(x)`, `dequeue()`

Types:

- Simple Queue
- Circular Queue
- Priority Queue
- Deque (Double-ended Queue)

3.3 Linked Lists

A **linked list** is a dynamic collection of nodes connected by pointers.

Node Structure:

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

Types:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Applications:
Dynamic memory allocation, stacks, queues, graphs.

4. Trees and Heaps

4.1 Trees

- **Definition:** A non-linear data structure with hierarchical relationships.
- **Terminology:** Root, Parent, Child, Leaf, Height, Depth.

Binary Tree: Each node has at most two children.

Traversal Methods:

- **Preorder (Root–Left–Right)**
- **Inorder (Left–Root–Right)**
- **Postorder (Left–Right–Root)**
- **Level Order (Breadth First)**

Applications:

- Expression trees
- Decision trees
- Binary Search Trees (BST)

4.2 Heap

- A **complete binary tree** satisfying the **heap property**:
    - **Max-Heap:** Parent ≥ Children
    - **Min-Heap:** Parent ≤ Children

Used in: Priority queues and heap sort.

5. Hash Tables

5.1 Concept

A **hash table** maps keys to values using a **hash function**.

Example:

```
index = hash(key)
```

Collision Handling:

- Chaining
- Open addressing (Linear probing, Quadratic probing)

Applications:

- Database indexing
- Symbol tables
- Caching

6. Graphs and Graph Algorithms

6.1 Graph Basics

A **graph** G = (V, E) is a set of vertices and edges.

Types:

- Directed / Undirected
- Weighted / Unweighted
- Cyclic / Acyclic

Representations:

- Adjacency Matrix
- Adjacency List

6.2 Common Graph Algorithms

- **Breadth-First Search (BFS)**
- **Depth-First Search (DFS)**
- **Dijkstra’s Algorithm** (Shortest Path)

- **Kruskal's / Prim's Algorithm** (Minimum Spanning Tree)

Applications:
Network routing, Social networks, AI pathfinding.

7. Run-Time Storage Management

7.1 Pointers and References

Pointers store memory addresses of other variables or structures.

```
int x = 10;
int *ptr = &x;
```

Uses:
Dynamic memory allocation, linked structures.

7.2 Linked Structures

Data stored non-contiguously, connected by pointers (used in linked lists, trees).

8. Searching Algorithms

8.1 Sequential (Linear) Search

Traverse the entire array sequentially.
Time Complexity: O(n)

8.2 Binary Search

Requires sorted data.
Time Complexity: O(log n)

Example:

```
int binarySearch(int arr[], int n, int x);
```

9. Sorting Algorithms

Algorithm	Type	Time Complexity	Remarks
Selection Sort	Simple	$O(n^2)$	Small datasets
Insertion Sort	Simple	$O(n^2)$	Nearly sorted data

Algorithm	Type	Time Complexity	Remarks
Quick Sort	Divide & Conquer	O(n log n) avg	Recursive
Heap Sort	Comparison-based	O(n log n)	Uses heap
Merge Sort	Divide & Conquer	O(n log n)	Stable
Parallel MergeSort	Distributed	O(n log n/p)	Uses concurrency

10. Recursion

10.1 Concept

A **function calling itself** until a base condition is reached.

Example:

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

10.2 Recursive Backtracking

Used for exploring multiple possibilities (e.g., maze solving, N-Queens problem).

10.3 Implementation of Recursion

Stack-based execution model — each recursive call creates a new stack frame.

11. Complexity Analysis

Understanding **Big O Notation** for analyzing:

- Time complexity
- Space complexity

Examples:

Operation	Best	Average	Worst
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

12. Practical Exercises

1. Implement a stack using a linked list.
2. Write a program to perform binary search.
3. Implement BFS and DFS for a given graph.
4. Compare sorting times for QuickSort and MergeSort.
5. Solve Tower of Hanoi recursively.