UNIT I: Introduction to Data Structures and Algorithms

---

1. Introduction

Definition:
Data structures are ways of organizing and storing data in a computer so that it can be accessed and modified efficiently. They play a critical role in enhancing the performance of algorithms and programs.

Features:

- Efficient data organization.
- Optimized data retrieval and manipulation.
- Used for solving real-world problems like searching, sorting, and graph traversal.

---

2. Types of Data Structures

1. **Primitive Data Structures**:
    - Directly operated upon by machine-level instructions.
    - Examples: Integers, Floats, Characters, Pointers.
2. **Non-Primitive Data Structures**:
    - Complex structures built using primitive types.
    - Examples: Arrays, Linked Lists, Stacks, Queues, Trees, Graphs.

Subcategories:

- **Linear Data Structures**: Organized sequentially (e.g., Arrays, Stacks, Queues).
- **Non-Linear Data Structures**: Hierarchical relationships (e.g., Trees, Graphs).

---

3. Algorithm

Definition:
An algorithm is a finite set of well-defined instructions performed in a specific sequence to solve a problem or perform a computation.

Characteristics:

- **Finiteness**: Must terminate after a finite number of steps.
- **Definiteness**: Each step must be precisely defined.
- **Input**: Takes zero or more inputs.
- **Output**: Produces at least one output.
- **Effectiveness**: Each step should be feasible and efficiently executable.

---

4. Pseudocode

Definition:
Pseudocode is a high-level description of an algorithm using simple and natural language, often resembling programming syntax.

Example:

```
BEGIN
   SET total to 0
   FOR each number in the list
       ADD number to total
   END FOR
   RETURN total
END
```

---

5. Algorithm Analysis

Algorithm analysis helps in evaluating the efficiency of an algorithm, primarily focusing on resource usage.

1. **Time Complexity**:
    - Measures the time taken by an algorithm to run as a function of input size.
    - Common notations:
        - O(1): Constant time
        - O(n): Linear time
        - O(n^2): Quadratic time
2. **Space Complexity**:
    - Evaluates the memory consumed by an algorithm during its execution.
    - Includes memory for:
        - Program instructions

- Variables
- Recursion stack

---

6. Abstract Data Types (ADT)

Definition:
An ADT is a mathematical model for data types where only behavior (operations) is defined, not implementation details.

Examples:

- **List ADT**: Operations include insertion, deletion, traversal.
- **Queue ADT**: Operations include enqueue, dequeue, peek.

---

7. String Processing

Definition:
String processing involves operations performed on sequences of characters.

1. **Basic Terminology**:
   - **String**: A finite sequence of characters.
   - **Substring**: A smaller sequence derived from a string.
2. **String Operations**:
   - **Concatenation**: Combining two strings.
   - **Substring Extraction**: Extracting a part of the string.
   - **Pattern Matching**: Finding a substring within a string.

Example of Pattern Matching Algorithm:

```
def naive_pattern_matching(text, pattern):
    for i in range(len(text) - len(pattern) + 1):
        if text[i:i+len(pattern)] == pattern:
            return i
    return -1
```

---

8. Arrays

Definition:
An array is a collection of elements stored in contiguous memory locations, each identified by an index.

1. **Representation in Memory**:
    - Array elements are stored sequentially.
    - Address of an element:
      `Base Address + (Index × Size of each element)`
2. **Operations**:
    - **Insertion**: Adding an element.
    - **Deletion**: Removing an element.
    - **Searching**:
        - **Linear Search**: Sequentially checks each element.
        - **Binary Search**: Divides array and checks half.

Example of Binary Search:

```
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

3. **Sorting**:
    - **Bubble Sort**:
        - Repeatedly swaps adjacent elements if they are in the wrong order.

Example of Bubble Sort:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

4. **Multidimensional Arrays**:
    - Arrays with more than one dimension.
    - Example: `Matrix = [[1, 2], [3, 4]]`
5. **Pointers and Pointer Arrays**:
    - Pointer: A variable that stores the address of another variable.
    - Pointer Array: An array of pointers pointing to other elements or arrays.

UNIT II: Sorting and Linked Lists

---

1. Sorting

Sorting refers to the process of arranging data in a specific order (ascending or descending). There are two categories of sorting:

1. **Internal Sorting**:
    - Sorting takes place entirely in the main memory (RAM).
    - Example algorithms: Bubble Sort, Quick Sort, Merge Sort.
2. **External Sorting**:
    - Involves sorting large amounts of data that cannot fit into the main memory.
    - Data is stored in external storage like disks, and only a small portion is loaded into memory at any time.
    - Example: Sorting large databases.

Common Sorting Algorithms:

- **Bubble Sort**: A simple algorithm where adjacent elements are compared and swapped until the entire list is sorted.
- **Insertion Sort**: Builds the sorted array one element at a time by inserting the next element into its correct position.
- **Selection Sort**: Repeatedly selects the smallest (or largest) element from the unsorted part and places it in the sorted part.
- **Merge Sort**: A divide-and-conquer algorithm that divides the array into two halves, recursively sorts them, and then merges the sorted halves.

- **Quick Sort**: Another divide-and-conquer algorithm that selects a 'pivot' element, partitions the array around the pivot, and recursively sorts the subarrays.
- **Radix Sort**: A non-comparative integer sorting algorithm that sorts numbers digit by digit, starting from the least significant digit.

---

2. Linked Lists

A linked list is a linear data structure where elements (nodes) are stored in non-contiguous memory locations. Each node contains data and a reference (pointer) to the next node.

1. **Linked List vs Arrays**:
    - **Memory**: Linked lists do not require contiguous memory, unlike arrays. This makes them more flexible for dynamic data storage.
    - **Insertion/Deletion**: Insertion and deletion of elements can be done easily in a linked list as compared to arrays, where shifting of elements is required.
    - **Access time**: Arrays provide constant-time access, whereas linked lists provide linear-time access due to traversal.
2. **Types of Linked Lists**:
    - **Singly Linked List**: Each node points to the next node in the list. The last node points to `NULL`.
    - **Doubly Linked List**: Each node has two references: one to the next node and one to the previous node. This allows traversal in both directions.
    - **Circular Linked List**: The last node points back to the first node, creating a circular structure. It can be either singly or doubly circular.
    - **Doubly Circular Linked List**: A combination of doubly linked list and circular list, where the last node points to the first node and vice versa.
3. **Operations on Singly Linked List**:
    - **Insertion**: Add a new node at the beginning, end, or at a specified position.
    - **Deletion**: Remove a node from the beginning, end, or at a specified position.
    - **Traversal**: Visit each node in the list, typically used for displaying elements or searching.
    - **Searching**: Traverse the list to find a specific element.

UNIT III: Stacks and Queues

---

1. Stacks

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

1. **Array and Linked List Representation of Stacks**:
    - **Array-based Stack**: Uses a fixed-size array to store elements. The top of the stack is represented by an index.
    - **Linked List-based Stack**: Uses a linked list where the top of the stack corresponds to the first node in the list.
2. **Operations on Stacks**:
    - **Push**: Adds an element to the top of 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.
3. **Applications of Stacks**:
    - **Recursion**: Stacks are used in the implementation of recursive function calls.
    - **Polish Notation**: Stacks are used to evaluate expressions in Polish notation (prefix and postfix).

---

2. Queues

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

1. **Array and Linked List Representation of Queues**:
    - **Array-based Queue**: Uses a circular array to store elements.
    - **Linked List-based Queue**: Uses a linked list where the front and rear pointers keep track of the beginning and end of the queue.
2. **Types of Queues**:
    - **Simple Queue**: Basic FIFO queue.
    - **Circular Queue**: The last element points to the first element, forming a circle. Helps in efficiently utilizing space.

- **Priority Queue**: Each element is assigned a priority, and elements are dequeued in order of priority rather than arrival.
- **Double Ended Queue (Deque)**: Allows insertion and deletion from both ends of the queue.

3. **Operations on Queues**:

- **Enqueue**: Adds an element to the rear of the queue.
- **Dequeue**: Removes an element from the front of the queue.
- **Front/Peek**: Returns the element at the front without removing it.
- **IsEmpty**: Checks if the queue is empty.

4. **Applications of Queues**:

- **Scheduling**: Queues are used in job scheduling in operating systems.
- **Buffering**: Queues are used in buffering data between producers and consumers in data streaming.
- **Breadth-First Search (BFS)**: Queues are used to implement BFS in graph traversal.

UNIT IV: Trees and Graphs

---

1. Trees

A tree is a hierarchical data structure consisting of nodes connected by edges. It is widely used for representing relationships and hierarchies.

1. **Basic Tree Concepts**:

- **Root**: The topmost node of a tree.
- **Parent**: A node that has one or more child nodes.
- **Child**: A node that is a descendant of another node.
- **Leaf**: A node with no children.
- **Subtree**: A tree formed by a node and all its descendants.

2. **Binary Trees**:

- A binary tree is a tree in which each node has at most two children, typically referred to as the left and right children.

- **Representation of Binary Tree in Memory**: Nodes are typically represented as objects with data and pointers to left and right children.

3. **Binary Tree Traversals**:
   - **In-order Traversal**: Left subtree → Root → Right subtree.
   - **Pre-order Traversal**: Root → Left subtree → Right subtree.
   - **Post-order Traversal**: Left subtree → Right subtree → Root.
   - **Level-order Traversal**: Traverses the tree level by level.
4. **Binary Search Trees (BST)**:
   - A binary tree where the left child is smaller than the parent, and the right child is larger.
   - Operations like searching, insertion, and deletion can be done in O(log n) time in a balanced BST.
5. **Heapsort**:
   - A comparison-based sorting algorithm based on a binary heap data structure.
   - Heaps are complete binary trees where each parent node is either greater than or less than its children (max-heap or min-heap).

---

2. Graphs

A graph is a non-linear data structure consisting of nodes (vertices) and edges (connections between nodes).

1. **Representations of Graphs**:
   - **Adjacency Matrix**: A 2D array where each cell indicates whether there is an edge between two nodes.
   - **Adjacency List**: A list where each node has a list of all nodes connected to it.
2. **Sequential Representation**:
   - A form of adjacency matrix or list representation where graph data is stored in a sequential manner for easier access.
3. **Warshall's Algorithm**:
   - A graph algorithm used to find the transitive closure of a directed graph, i.e., determining which vertices are reachable from others.

4. **Linked Representation of Graphs**:
    - In this representation, each vertex in the graph contains a list of adjacent vertices, typically implemented using linked lists.
5. **Operations on Graphs**:
    - **Insertion**: Add a vertex or an edge.
    - **Deletion**: Remove a vertex or an edge.
    - **Traversal**: Explore all the vertices and edges in a graph. Two common traversal methods are Depth-First Search (DFS) and Breadth-First Search (BFS).
