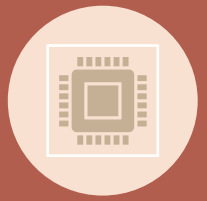

# ASM1

DATA STRUCTURE AND ALGORITHM

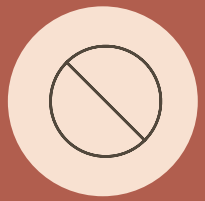
# TIME LINE
# Activity 1

Introduction

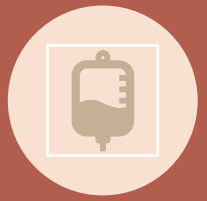
I. Create a design specification for data structures, explaining the valid operations that can be carried out on the structures. (**P1**)



II. Determine the operations of a memory stack and how it is used to implement function calls in a computer. (**P2**)



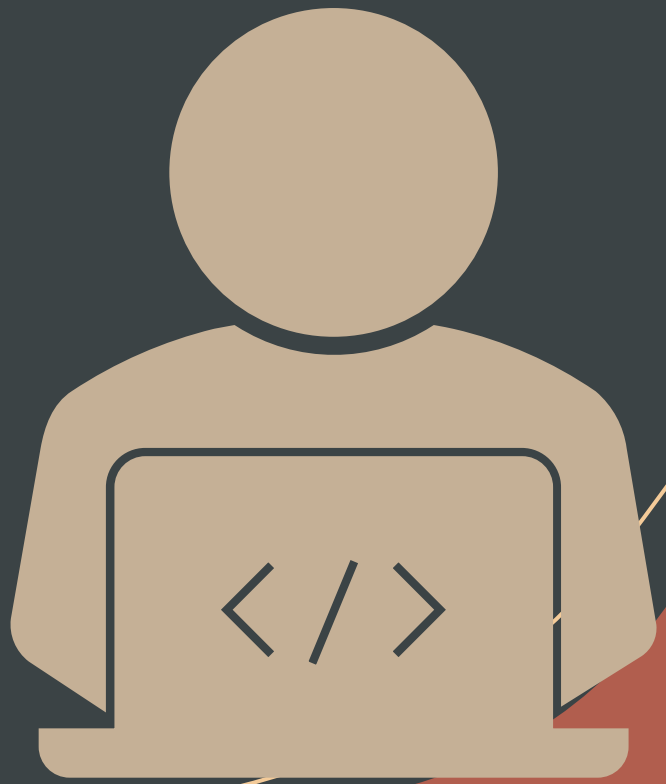
**III.** Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue. (**M1**)



**IV.** Compare the performance of two sorting algorithms. (**M2**)

**V. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each (D1).**

Conclusion



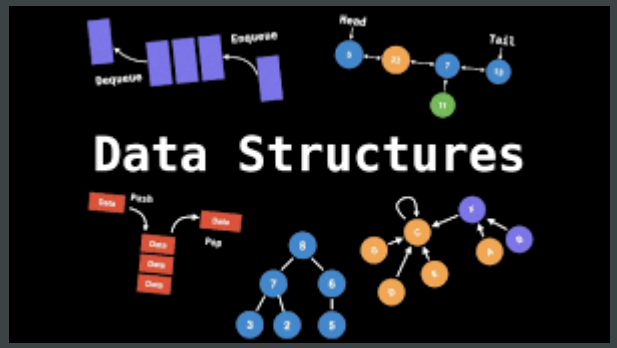
# INTRODUCTION

Welcome to the presentation on how to use data object types (ADTs) and algorithms in software design, development, and testing. In this section, we will explore ADTs such as Stack ADT and FIFO queue structures that are expected to improve the efficiency of data management in software.

In addition, I will introduce two popular sorting algorithms and two shortest path algorithms in the network. The goal is to learn how to apply these algorithms to optimize data processing, ensuring smoother and more efficient software operation.

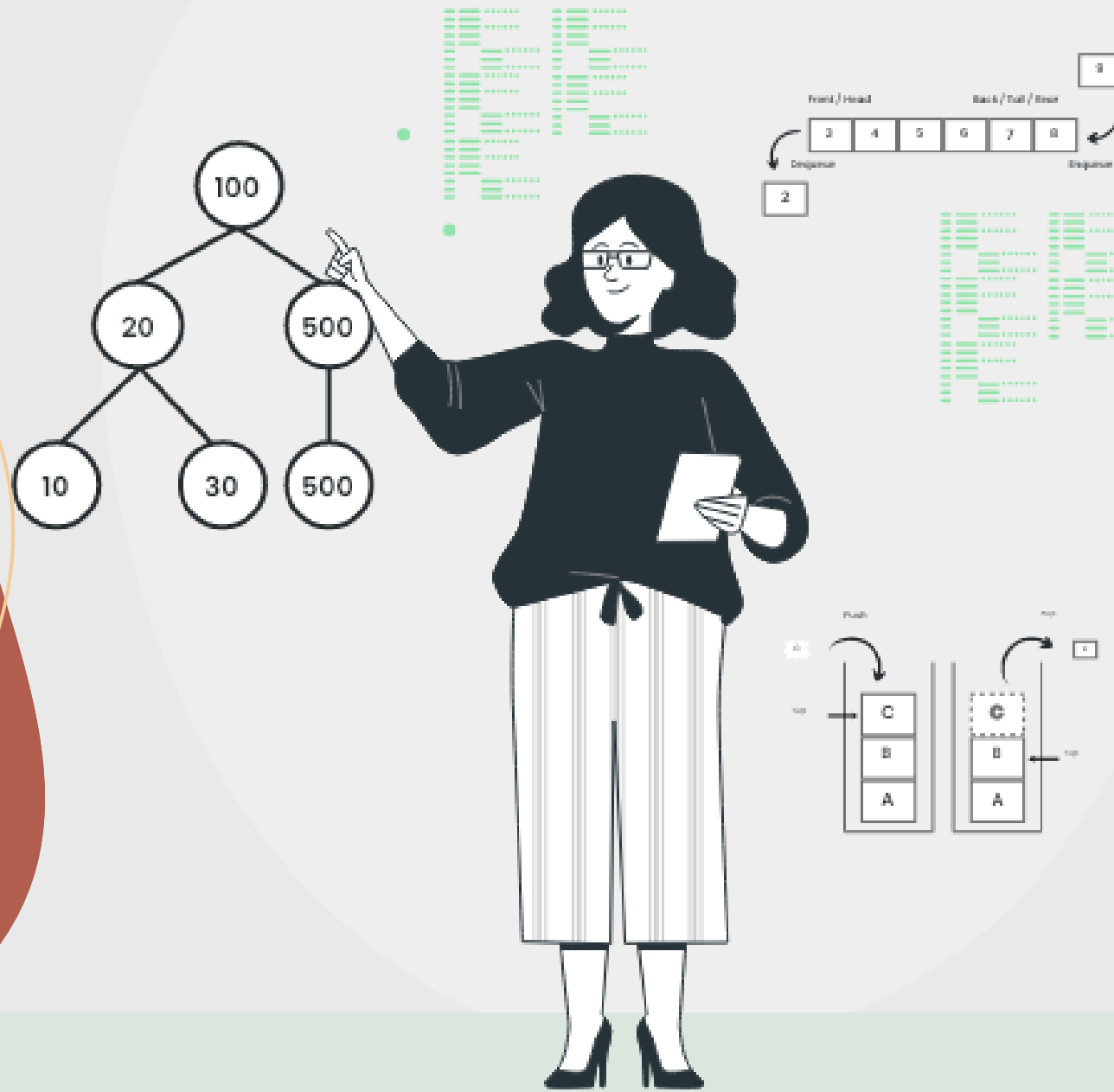
I. Create a design specification for data structures, explaining the valid operations that can be carried out on the structures. (**P1**)

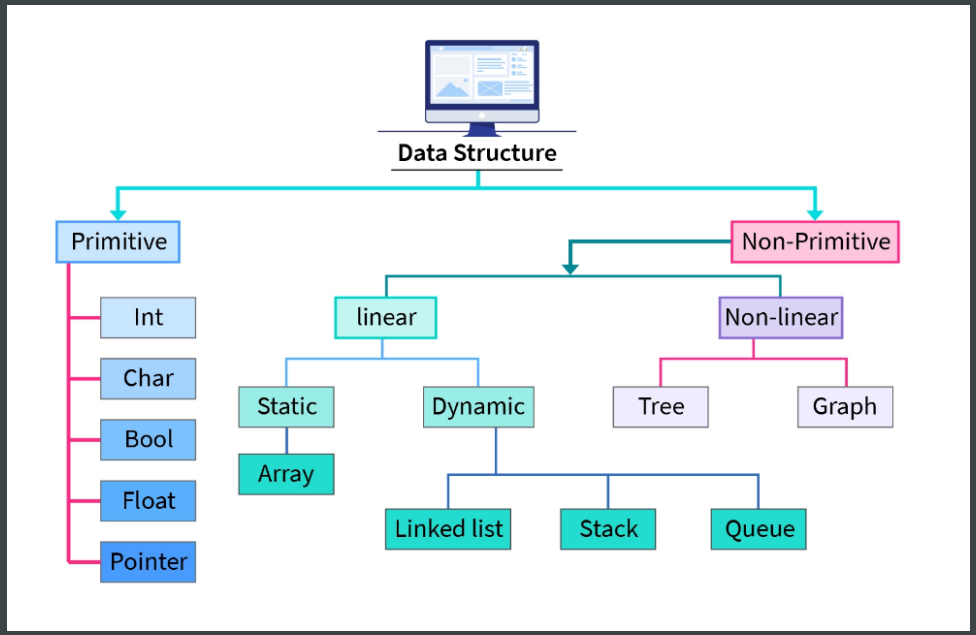
1. Identify the Data Structures
2. Define the Operations
3. Specify Input Parameters
4. Define Pre- and Post-conditions
5. Discuss Time and Space Complexity
6. Provide Examples and Code Snippets (if applicable)



# 1. Identify the Data Structures

- A **Data Structure** is the method of storing and organizing data in the computer memory.
- It is the branch of Computer Science that deals with arranging such large datasets in such a manner so that they can be accessed and modified as personal the requirements.






Data Structures are mainly classified into two categories:

1. Primitive Data Structures: Store data of only one type (e.g., integer, float, character, boolean).
2. Non-Primitive Data Structures: Derived from primitive types and can store more than one type of data (e.g., arrays, linked lists, trees). These are categorized as:

- Linear Data Structures: Data is arranged in a sequence.
  - Static Data Structures: Fixed size, memory allocated at compile time (e.g., arrays).
  - Dynamic Data Structures: Flexible size, memory allocated at runtime (e.g., linked lists, stacks, queues).
- Non-Linear Data Structures: Data forms a hierarchy, not a sequence (e.g., trees, graphs).

# 2. Define the Operations

Some of the basic operations performed on a data structure are as follows:

Traversing- Accessing or visiting each data element in a particular order in a data structure is called traversing.

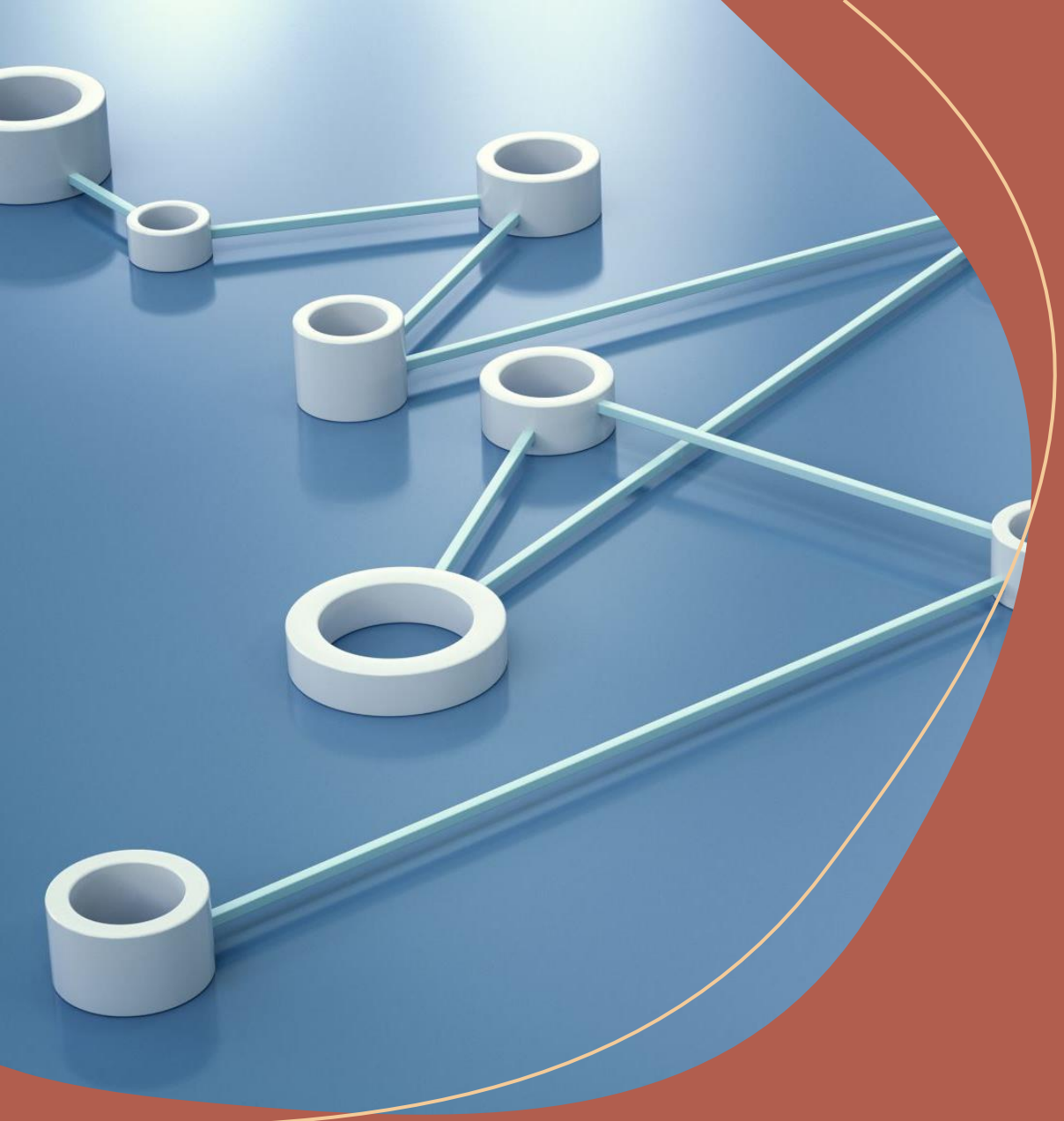
Searching- It is finding the location(s) of data that satisfy one or more conditions. We have various search techniques like linear search and binary search to search for elements in a data structure.

Inserting—Inserting new data into the data structure is called insertion. Data can be inserted at the initial (first), final (last), or anywhere between the first and final locations. This operation increases the size of the data structure.

Deleting—Removing existing data elements from the data structure is called deletion. Data is deleted from the initial (first), final (last), or anywhere between the first and final locations. This operation decreases the size of the data structure.

Sorting- Arranging the data in a specified order (ascending or descending) is sorting. A user uses different techniques to sort data, viz bubble sort, shell sort, etc.

Merging- Combining the data from two data structures (or files) into a single is called merging. This operation improves the competency in searching and ascertains correct data access to the users.



# 3. Specify Input Parameters

- Input parameters are essential elements that dictate how data is accessed,

manipulated, and managed in data structures.

Here is input parameters of each data structure:

- Array: Index (for access), Value (for insert/delete).
- Linked List: Node, Value (for operations like insert or delete).
- Stack/Queue: Value (for Push/Pop or Enqueue/Dequeue).
- BST: Key, Value (for insert/search).
- Hash Table: Key, Value (for insert/search).

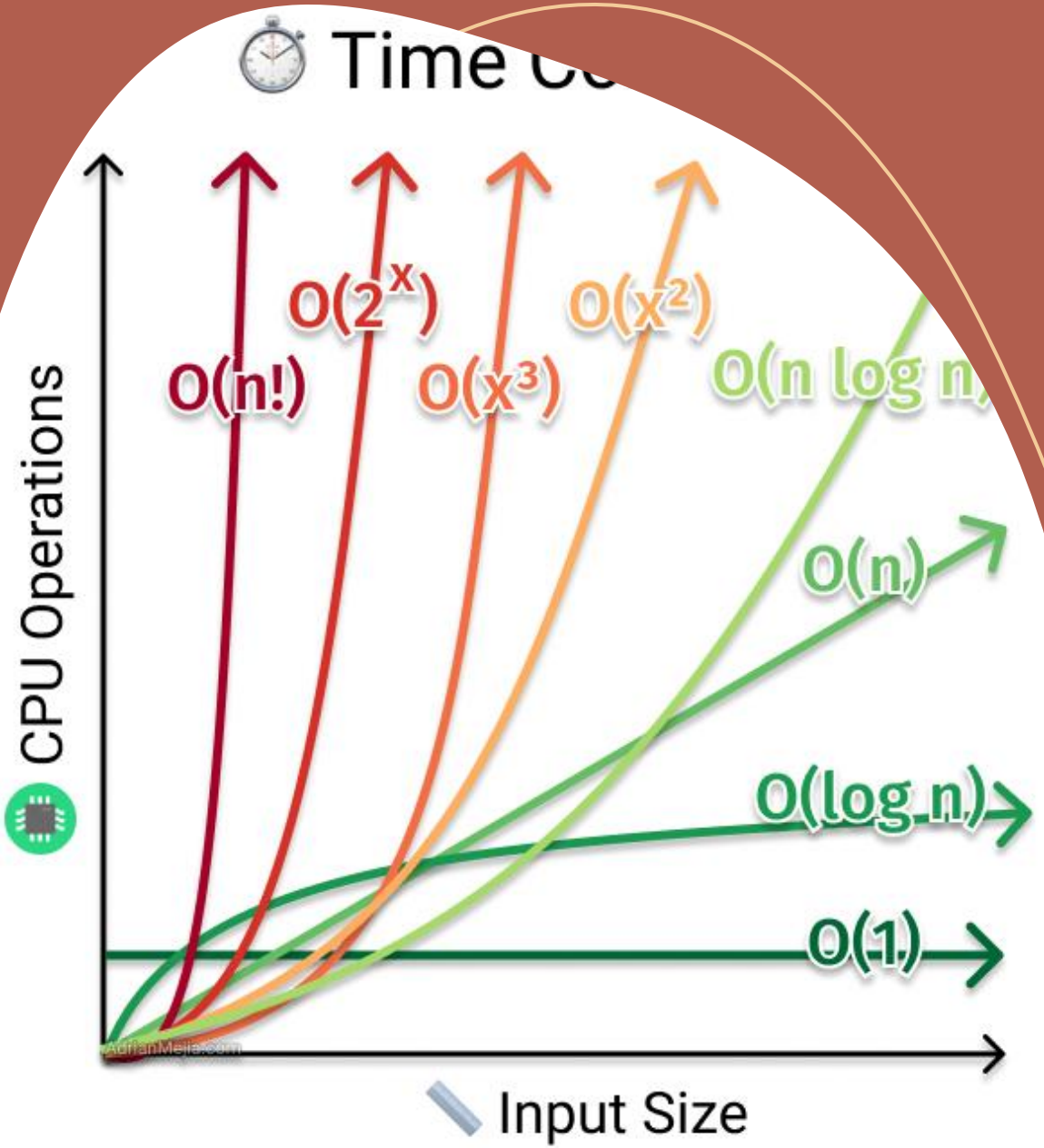
# 4. Define Pre- and Post-conditions

a. Pre-conditions

- Definition: Conditions that must be true before a function or op eration is executed.
- Purpose: Ensure necessary inputs and valid environment for exe cution.
- Examples:
    - Binary Search Tree: Tree is not null before inserting a nod e.
    - Array: Index is within bounds before accessing an elemen t.
    - Linked List: List is not empty before deleting a node.

**b. Post-conditions**

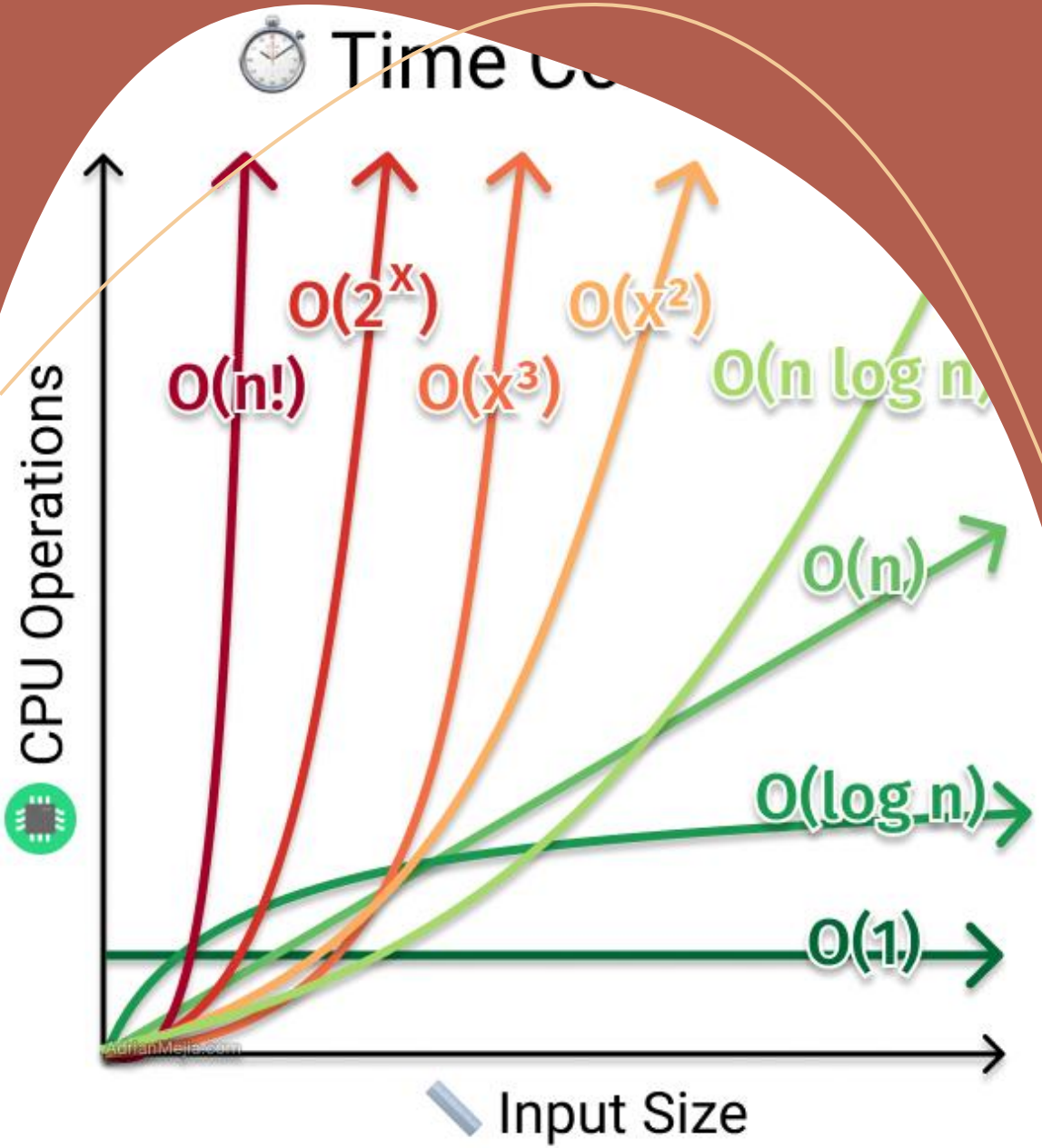
- **Definition**: Conditions that must be true after a function or op eration has been executed.
- **Purpose**: Verify successful completion and valid state of data st ructure.
- **Examples**:
    - Binary Search Tree: Tree remains valid after insertion.
    - Array: Returned value is correct after accessing an elem ent.
    - Linked List: Integrity maintained after node deletion.




# 5. Discuss Time and Space Complexity

a. Time complexity:

- Time complexity refers to the amount of time an algorithm takes to complete as a function of the input size. It provides an upper bound on the time required, which is often expressed using Big O notation (e.g., O(1), O(n), O(log n), O(n^2)).
- For example:
  - Linked list insert: O(1) for inserting at the beginning or end (if tail pointer exists), O(n) if inserting at a specific position.




b. Space complexity

- Space complexity refers to the amount of memory an algorithm uses as a function of the input size. Like time complexity, space complexity is also expressed using Big O notation.

- For example:

  Array: Space Complexity: O(n)

  - Requires space proportional to the number of elements stored.

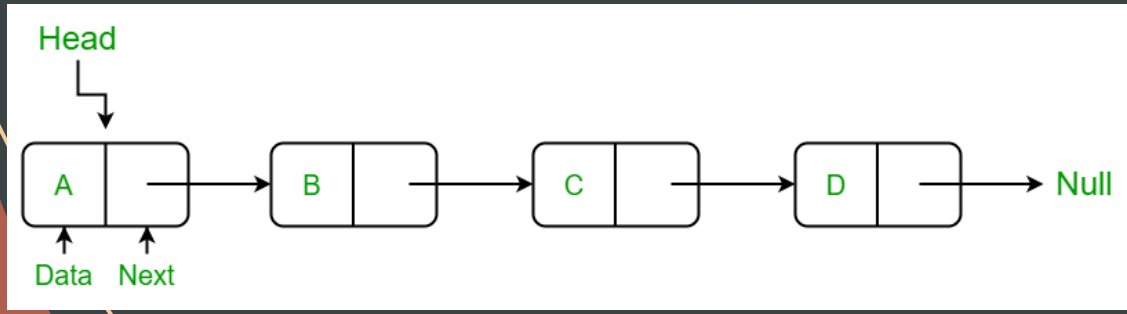
# 6. Provide Examples and Code Snippets (if applicable)

- So, here is example of specification for

Linked list:

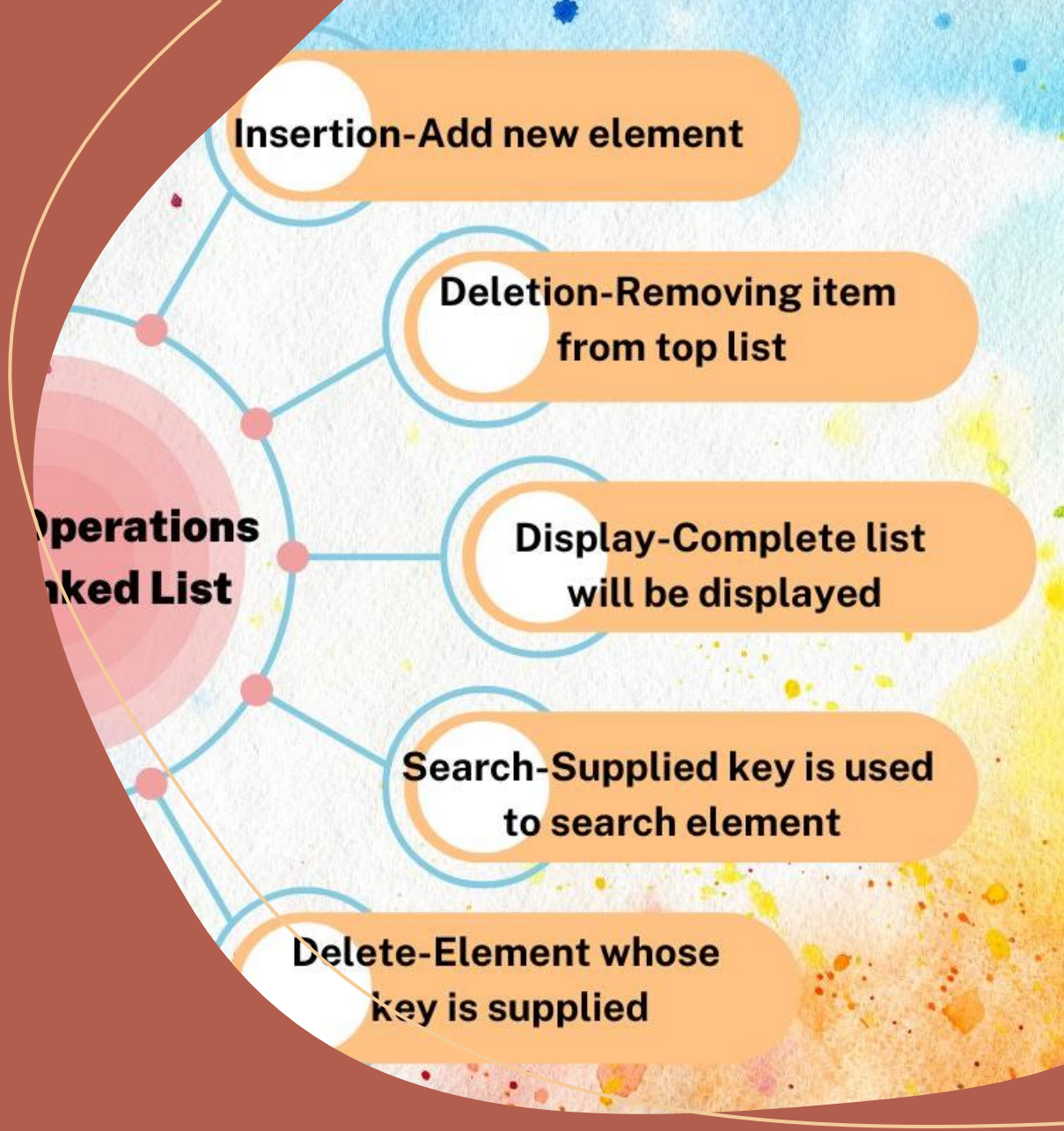
- Define:

  A Linked List is a linear collection of nod es, where each node points to the next node in the sequence.



The operations:

- **Insert (int data)**: Inserts a new node with the s pecified data at the beginning of the list.
- **Delete (int data)**: Deletes the first node with th e specified data.
- **Search (int data)**: Searches for a node with the specified data.
- **Display()**: Displays all the nodes in the list.



# - Define Pre- and Post-conditions:

- **Insert**:
  - Pre-condition: The list can be empty or non-empty.
  - Post-condition: A new node is added to the beginning of the list.
- **Delete**:
  - Pre-condition: The list should contain at least one node.
  - Post-condition: The first node with the specified value is removed from the list.
- **Search**:
  - Pre-condition: The list can be empty or non-empty.
  - Post-condition: Returns the node with the specified value if it exists; otherwise, returns null.
- **Display**:
  - Pre-condition: The list can be empty or non-empty.
  - Post-condition: Prints all nodes in the list.

# - Time and Space complexity:

- **Insert**:
  - Time Complexity: O(1)
  - Space Complexity: O(1)
- **Delete**:
  - Time Complexity: O(n) - In the worst case, we may need to traverse the entire list.
  - Space Complexity: O(1)
- **Search**:
  - Time Complexity: O(n)
  - Space Complexity: O(1)
- **Display**:
  - Time Complexity: O(n)
  - Space Complexity: O(1)

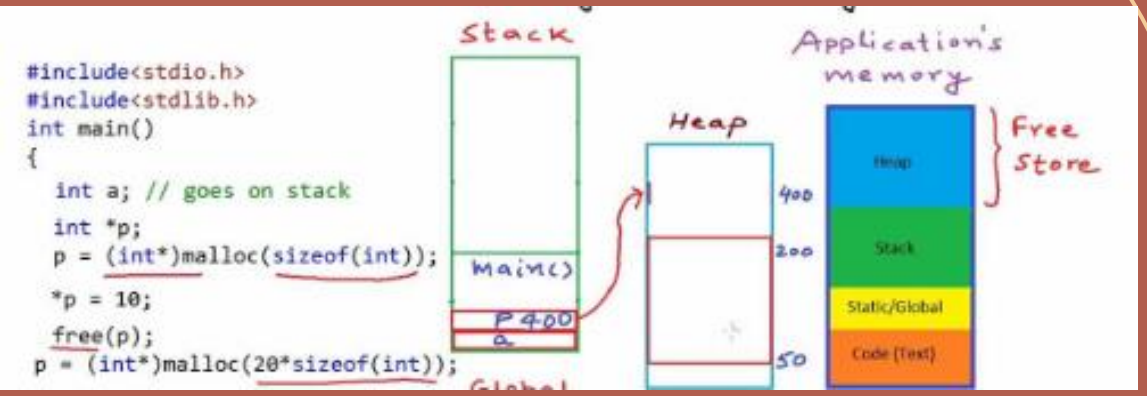
- Code of Linkedlist in Java:
- This code Create class Liskedlist store data type is integer.
- Linklist operations with two methods are insert and display

```java
public class Main {
    class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    class LinkedList {
        Node head;

        public LinkedList() {
            this.head = null;
        }

        public void insert(int data) {
            Node newNode = new Node(data);
            newNode.next = head;
            head = newNode;
        }

public void display() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " -> ");
        current = current.next;
    }
    System.out.println(x:"null");
```

```
Run main | Debug main | Run | Debug
public static void main(String[] args) {
    Main mainInstance = new Main();
    LinkedList list = mainInstance.new LinkedList();
    list.insert(data:10);
    list.insert(data:20);
    list.insert(data:30);
    list.display();
```

So, here is example results of Stack for two methods Insert and Display:

```
30 -> 20 -> 10 -> null
```

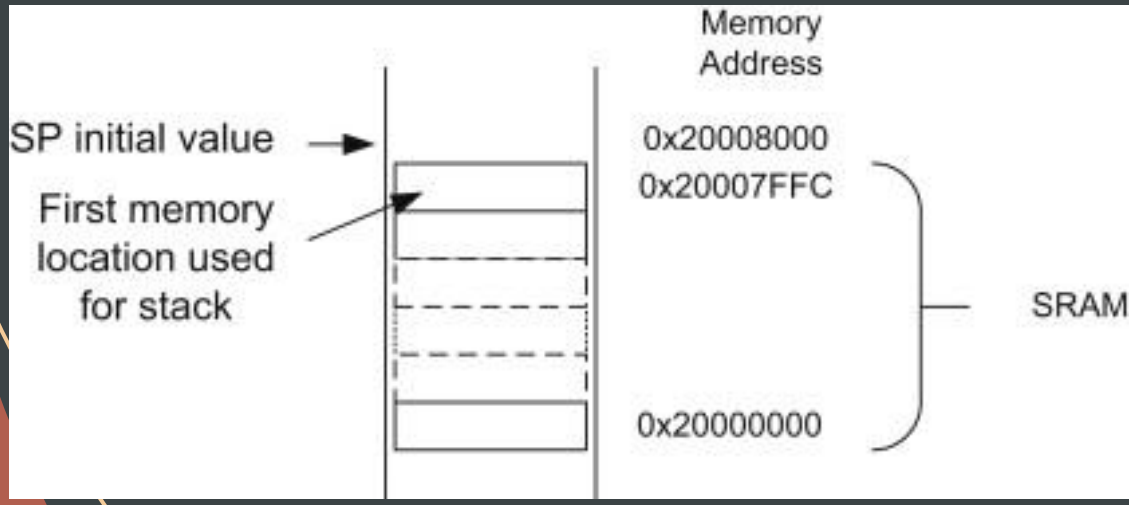



# II. Determine the operations of a memory stack and how it is used to implement function calls in a computer. (P2)

1. Define a Memory Stack
2. Identify Operations
3. Function Call Implementation
4. Demonstrate Stack Frames
5. Discuss the Importance

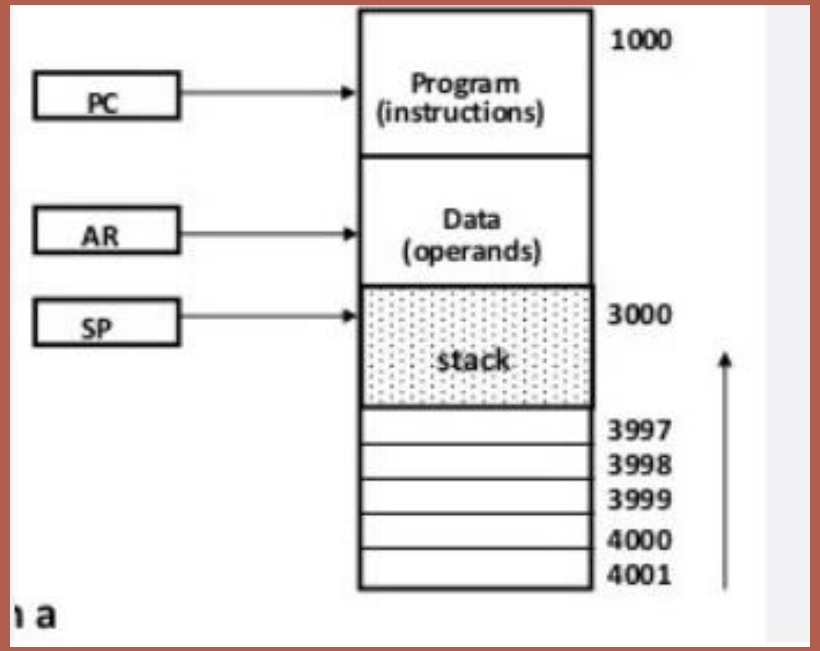
# 1. Define a Memory Stack

**Definition**: A memory stack is a special re gion of a computer’s memory that stores temporary data such as function paramet ers, return addresses, and local variables during function execution.



# 2. Identify Operations

- **Memory Stack Operations**
- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the top element from the stack.
- **Peek**: View the top element without removing it.

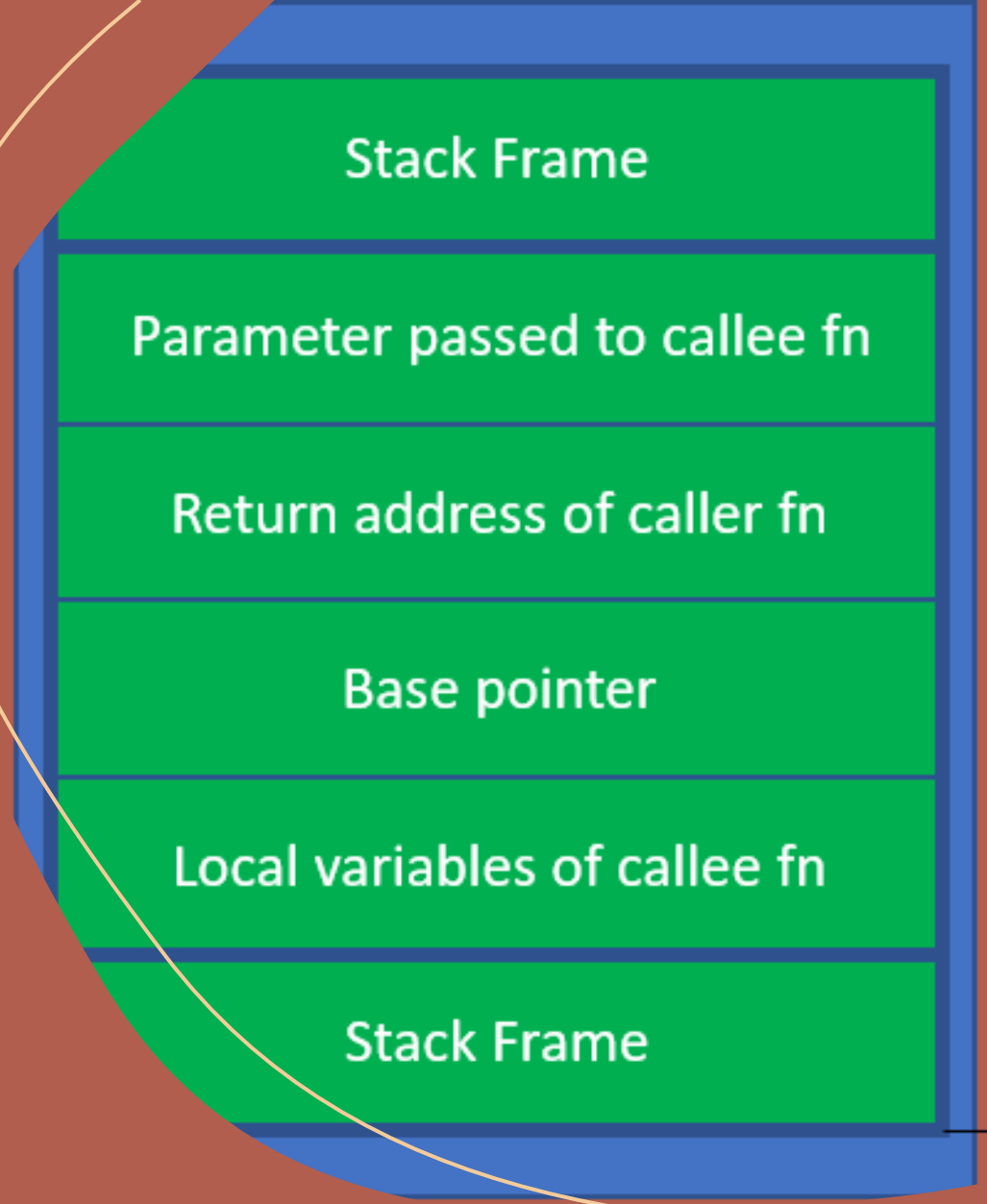


# 3. Function Call Implementation

- **Function Call Implementation**
- **Process**:
  - When a function is called, a new stack frame is created and pushed onto the st ack.
  - The stack frame contains the function’s return address, parameters, and local v ariables.
  - Once the function execution is complete, the stack frame is popped from the st ack, and control returns to the calling function.

# 4. Demonstrate Stack Frames

- A stack frame is an area of stack memory that is created each time a function is called. It stores the information needed to execute the function and returns to the correct place when the function ends.
- Components: Return address, function parameters, l ocal variables.
- Lifecycle:
  - Push: Create a new stack frame on function call
  - Pop: Remove the stack frame on function retur n.



# 5. Discuss the Importance

Importance

- **Resource Management**: Efficient use of memory for temporary data.
- **Function Nesting**: Supports nested function calls and recursion.
- **Data Integrity**: Ensures local variables are isolated and managed properly.

# **III.** Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue. (**M1**)

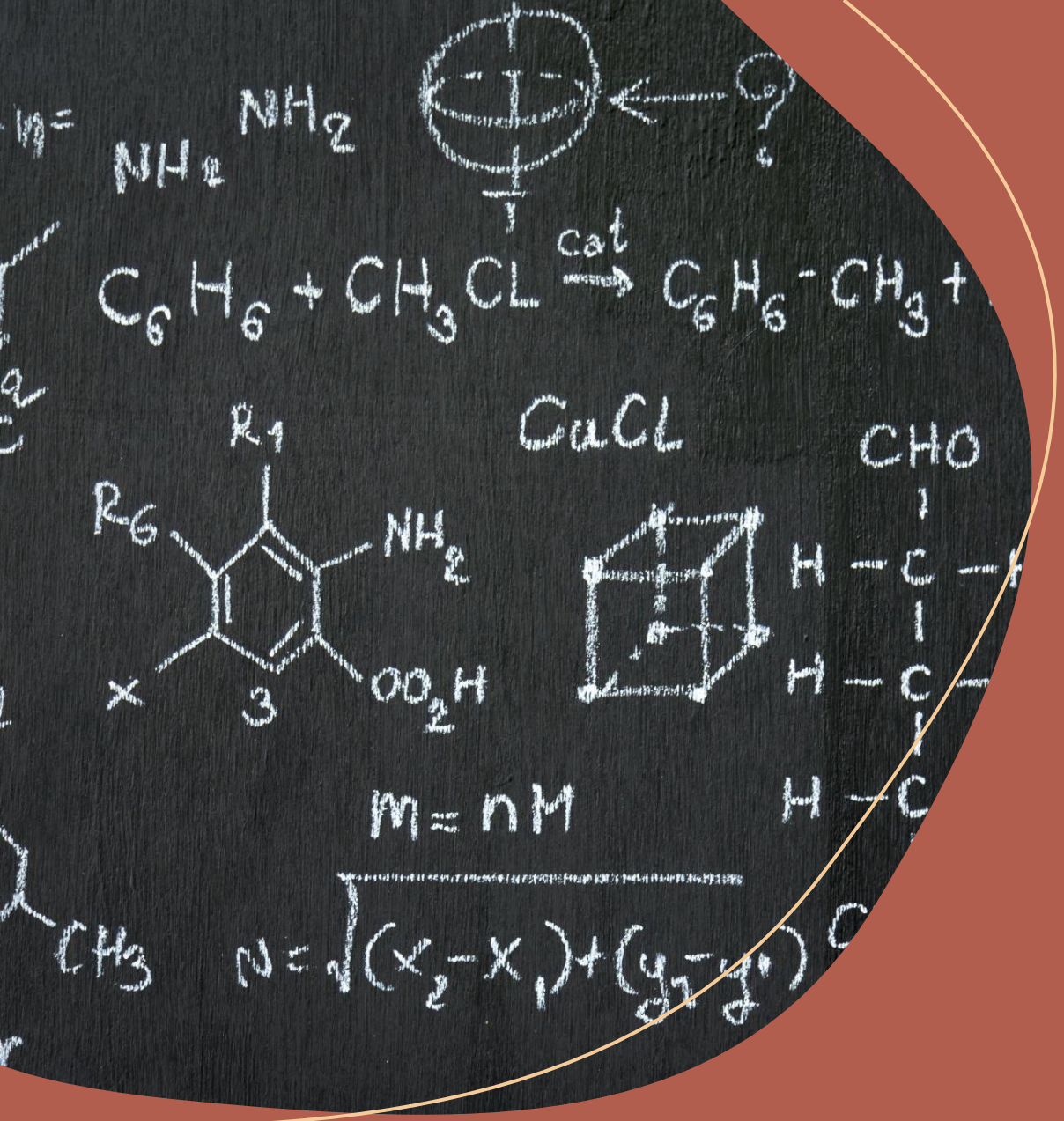
1. Introduction FIFO
2. Define the Structure
3. Array-Based Implementation
4. Linked List-Based Implementation
5. Provide a concrete example to illustrate how the FIFO queue works

# 1. Introduction FIFO

**Definition**: First In, First Out (FIFO) is a queue where the first element added is the first one removed. It mimics a realworld queue, like people waiting in line.

# 2. Define the Structure

- Queue Operations:
  - **Enqueue**: Add an element to the end.
  - **Dequeue**: Remove an element from the front.
  - **Peek**: View the front element without removing it.




# 3. Array-Based Implementation

- Structure: Uses a fixed-size array.
- Operations:
  - Enqueue: Add element at the rear, increment the rear index.
  - Dequeue: Remove element from the front, increment the front index.
- Example:
  - Initial State: [ ], Front = 0, Rear = -1.
  - Enqueue 1, 2, 3: [1, 2, 3], Front = 0, Rear = 2.
  - Dequeue: [ , 2, 3], Front = 1, Rear = 2.

# 4. Linked List-Based Implementation

- **Structure**: Uses nodes where each node points to the next.
- **Operations**:
  - **Enqueue**: Add a new node at the end.
  - **Dequeue**: Remove the node from the front.
- **Example**:
  - **Initial State**: Front -> null, Rear -> null.
  - **Enqueue 1, 2, 3**: Front -> 1 -> 2 -> 3 -> null.
  - **Dequeue**: Front -> 2 -> 3 -> null

# 5. Provide a concrete example to illustrate how the FIFO queue works

- Concrete Example of FIFO Queue
- Scenario: Customer Service
- Customer Arrivals: 1, 2, 3.
- Queue State:
  - **Enqueue 1**: Queue: [1].
  - **Enqueue 2**: Queue: [1, 2].
  - **Enqueue 3**: Queue: [1, 2, 3].
- Service:
  - **Dequeue**: Serve Customer 1. Queue: [2, 3].
  - **Dequeue**: Serve Customer 2. Queue: [3].
  - **Dequeue**: Serve Customer 3. Queue: [ ].

Code for Customer service:

```java
public class Main {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();

        queue.add(e:1);
        System.out.println("Enqueue 1: Queue: " + queue);
        queue.add(e:2);
        System.out.println("Enqueue 2: Queue: " + queue);
        queue.add(e:3);
        System.out.println("Enqueue 3: Queue: " + queue);

        queue.poll();
        System.out.println("Dequeue: Serve Customer 1. Queue: " + queue);
        queue.poll();
        System.out.println("Dequeue: Serve Customer 2. Queue: " + queue);
        queue.poll();
        System.out.println("Dequeue: Serve Customer 3. Queue: " + queue);
    }
}
```

```
Enqueue 1: Queue: [1]
Enqueue 2: Queue: [1, 2]
Enqueue 3: Queue: [1, 2, 3]
Dequeue: Serve Customer 1. Queue: [2, 3]
Dequeue: Serve Customer 2. Queue: [3]
Dequeue: Serve Customer 3. Queue: []
```

- This is result of example

- Explain:

- **Queue<Integer> queue = new LinkedList<>();**: Initia lizes a queue using a LinkedList.
- **queue.add(1);**: Enqueues customer 1 and prints th e current state of the queue.
- **queue.poll();**: Dequeues the first customer and pri nts the updated state of the queue.
- **System.out.println("...");**: Each operation's result is printed to display the queue's state.

# **IV.** Compare the performance of two sorting algorithms. (**M2**)

1. Introducing the two sorting algorithms you will be comparing
2. Time Complexity Analysis
3. Space Complexity Analysis
4. Stability
5. Comparison Table
6. Performance Comparison
7. Provide a concrete example to demonstrate the differences in performance between the two algorithms

# 1. Introducing the two sorting algorithms you will be comparing

In this part, I will compare Selection sort and Quick sort


Compare

# 2. Time Complexity Analysis

**a. Quick Sort**:

- Best: O(nlog(n))
- Average: O(nlog(n))
- Worst: O(n2)

**b. Selection Sort**:

- Best: O(n2)
- Average: O(n2)
- Worst: O(n2)

# 3. Space Complexity Analysis

**Quick Sort:**

- **Space Complexity**: O(log(n))
  - Quick Sort is an in place sorting algorithm, meaning it doesn't require additional memory for arrays. However, due to its recursive nature,
  - Quick Sort requires memory to store the recursive calls in the call stack. The nu mber of these recursive calls is proportional to the depth of the recursion tree, which is logn on average. Thus, the space complexity of Quick Sort is O(logn)

## Selection Sort:

- **Space Complexity**: O(1)
  - Selection Sort is also an in-
    place sorting algorithm. It performs the sorting by continually selecting the sm
    allest element from the unsorted portion of the array and swapping it with the
    first unsorted element. This process only requires a temporary variable to hol
    d a value during the swap, leading to a space complexity of O(1).
-

# 4. Stability

Quick Sort:

- Not Stable
    - Quick Sort is not a stable sorting algorithm. This means that the relative order of equal elements may cha nge during the sorting process. This instability arises due to the way Quick Sort partitions the array and s waps elements around the pivot.

Selection Sort:

- Not Stable
    - Selection Sort is also not a stable sorting algorithm. During the process of finding the minimum element a nd swapping it with the first unsorted element, the relative order of equal elements can be altered.

# 5. Comparison Table

| Algorithm | Time Complexity (Best) | Time Complexity (Average) | Time Complexity (Worst) | Space Complexity | Stability |
|---|---|---|---|---|---|
| Quick Sort | $O(n\log n)$ | $O(n\log n)$ | $O(n^2)$ | $O(\log n)$ | No |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | No |

# 6. Performance Comparison

- **Quick Sort**: Generally faster with better average case performance. Efficient for large datasets, but worst case can be mitigated with randomization.
- **Selection Sort**: Simpler but less efficient. Performance remains constant irrespective of dataset size, making it impractical for large datasets.

# 7. Provide a concrete example to demonstrate the differences in performance between the two algorithms

So, here is example to compare two algorithms:

**Dataset**: [38, 27, 43, 3, 9, 82, 10]

- **Quick Sort**:

**Step:**

1. **Initial Array**:
2. **Choose Pivot (First Element): 38**
3. **Partition**:
    1. Elements less than 38: [27, 3, 9, 10]
    2. Pivot: [38]
    3. Elements greater than 38: [43, 82]

Recursive Calls:

Sort [27, 3, 9, 10]:

Pivot: 27

Partition: [3, 9, 10] [27]

Recursive Sort on [3, 9, 10]

Pivot: 3

Partition: [ ] [3] [9, 10]

Sort [9, 10]

Pivot: 9

Partition: [ ] [9] [10]

Result: [3, 9, 10]

Sort [43, 82]:

Pivot: 43

Partition: [ ] [43] [82]

Result: [43, 82]

# Selection sort:

## Step:

1. **Initial Array**: [38, 27, 43, 3, 9, 82, 10]
2. **First Pass**:
    1. Find the minimum element in the entire array: 3
    2. Swap it with the first element: [3, 27, 43, 38, 9, 82, 10]
3. **Second Pass**:
    1. Find the minimum element in the remaining unsorted part: 9
    2. Swap it with the second element: [3, 9, 43, 38, 27, 82, 10]
4. **Third Pass**:
    1. Find the minimum element in the remaining unsorted part: 10
    2. Swap it with the third element: [3, 9, 10, 38, 27, 82, 43]
5. **Continue** until the entire array is sorted.

# Compare time and space complexity:

| Complexity | Quicksort | Selection sort |
|---|---|---|
| Time | O(nlog(n)) | O(n2) |
| Space | O(nlog(n)) | O(1) |

# Results:

```
Original array:
38 27 43 3 9 82 10
Sorted array:
3 9 10 27 38 43 82
Time call function:4
```

quicksort

```
Original array:
38 27 43 3 9 82 10
Sorted array:
3 9 10 27 38 43 82
Time call selections: 16
```

Selection sort

# Explain Time complexity of quicksort:

- **Initial Call**: O(7) // [38, 27, 43, 3, 9, 82, 10]
- **First Recursive Call (4 elements)**: O(4)
- **Second Recursive Call (3 elements)**: O(3)
- **Third Recursive Call (2 elements)**: O(2)
- **Fourth Recursive Call (2 elements)**: O(2)

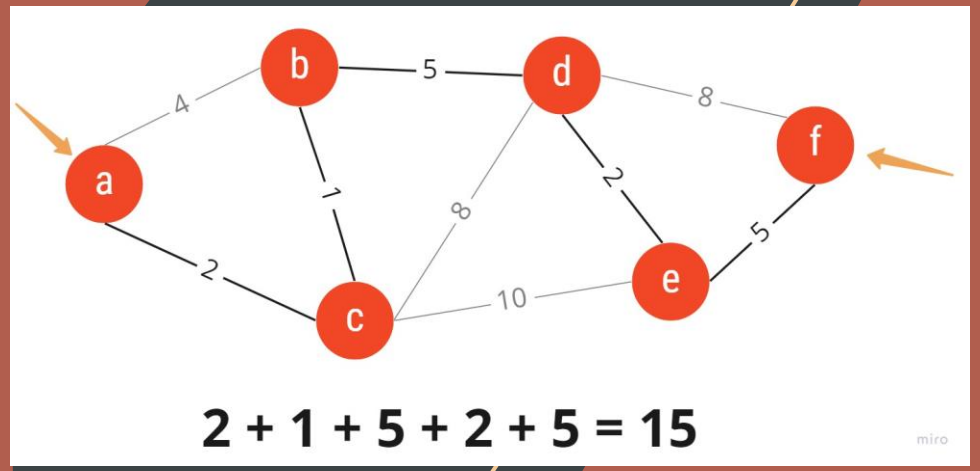
Summing these, the specific time complexity for this example can be approximated as:

O(7 + 4 + 3 + 2 + 2) = O(18)

So, O(18) quicksort better than O(16) of selection sort

# V. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each (D1)

1. Introducing the concept of network shortest path algorithms
2. Algorithm 1: Dijkstra's Algorithm
3. Algorithm 2: Prim-Jarnik Algorithm
4. Performance Analysis




# 1. Introducing the concept of network shortest path algorithms

Shortest path algorithms are important for finding the most efficient paths in a network, be it a road network, computer network or any graph-based system. Two widely used algorithms for this purpose are Dijkstra's Algorithm and Prim-Jarnik's Algorithm.

# 2. Algorithm 1: Dijkstra's Algorithm

## Overview

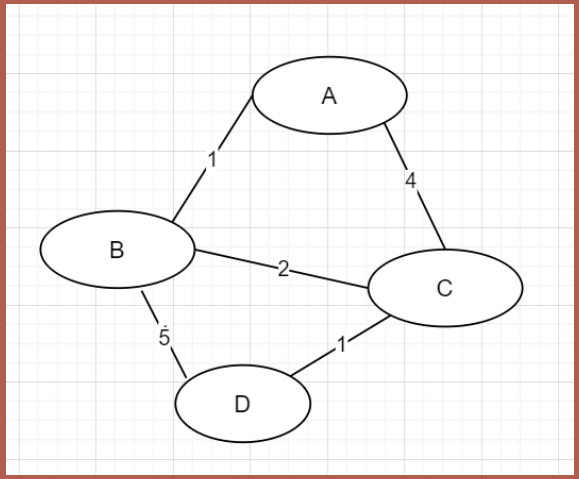
- **Purpose**: To find the shortest path from a single source node to all other nodes in a weighted graph.
- **Approach**: Greedy algorithm that iteratively selects the nearest unvisited node, updates distances, and marks nodes as visited.

Step:

- **Initialize**: Set the distance to the source node to 0 and all other nodes to infinity.
- **Priority Queue**: Use the priority queue to select the node with the smallest distance.
- **Relax**: For the current node, update the distance to its neighbors if a shorter path is found.
- **Repeat**: Mark the current node as visited and repeat until all nodes are processed.

For example:

- Consider a graph with nodes A, B, and C, and the following edges with weights:




- A - B: 1; C - D: 1
- A - C: 4; B - C: 2
- B - D: 5

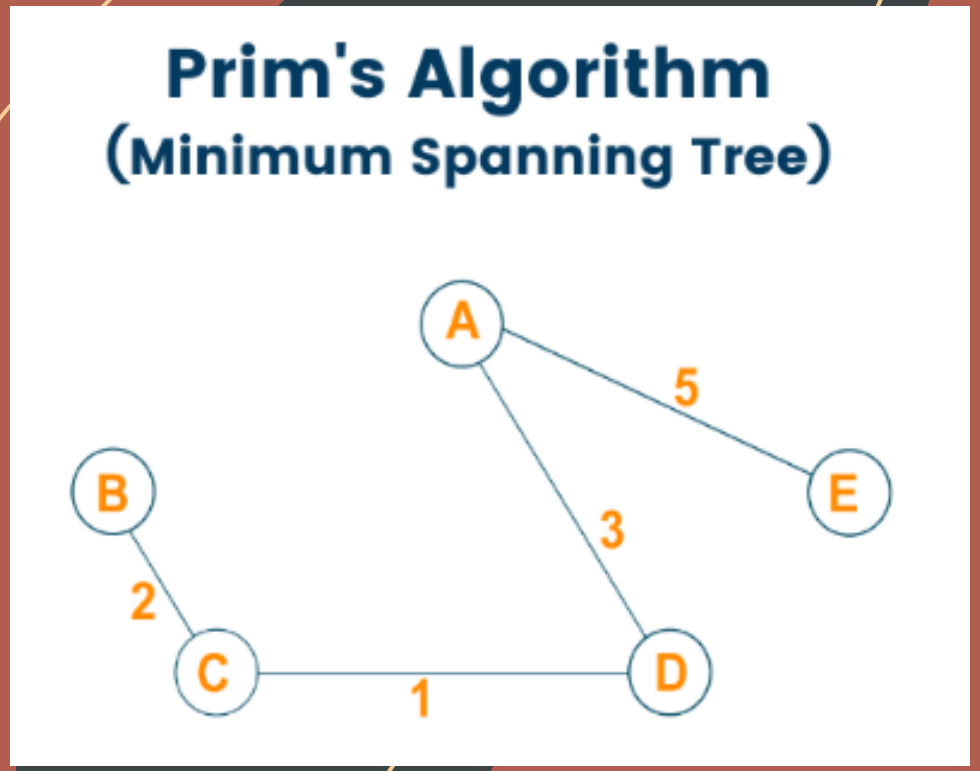
1. **Initialization**:
    1. Start from the source node A.
    2. Distance to A: 0, and all other nodes: ∞.
    3. Priority Queue: [(A, 0)].
2. **Step-by-Step Execution**:
    1. **Current Node: A** (distance 0)
        1. Update distances to neighbors: B (1), C (4).
        2. Priority Queue: [(B, 1), (C, 4)].
    2. **Current Node: B** (distance 1)
        1. Update distances to neighbors: C (1 + 2 = 3), D (1 + 5 = 6).
        2. Priority Queue: [(C, 3), (C, 4), (D, 6)]…

# Here is code and results:

```
Running Dijkstra's Algorithm from node A:
Shortest distance from A to A is 0
Shortest distance from A to B is 1
Shortest distance from A to C is 3
Shortest distance from A to D is 4
```

Results

```java
public static void dijkstra(Map<String, List<Node>> graph, String start) {
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
    queue.add(new Node(start, cost:0));
    Map<String, Integer> distances = new HashMap<>();
    distances.put(start, value:0);
    Set<String> visited = new HashSet<>();
    while (!queue.isEmpty()) {
        Node currentNode = queue.poll();
        String currentName = currentNode.name;

        if (!visited.add(currentName)) {
            continue;
        }

        for (Node neighbor : graph.getOrDefault(currentName, Collections.emptyList())) {
            if (visited.contains(neighbor.name)) {
                continue;
            }

            int newDist = distances.get(currentName) + neighbor.cost;
            if (newDist < distances.getOrDefault(neighbor.name, Integer.MAX_VALUE)) {
                distances.put(neighbor.name, newDist);
                queue.add(new Node(neighbor.name, newDist));
        for (Map.Entry<String, Integer> entry : distances.entrySet()) {
            System.out.println("Shortest distance from " + start + " to " + entry.getKey() + " is " + entry.getValue());
        }
    }


    static class Node {
        String name;
        int cost;

        Node(String name, int cost) {
            this.name = name;
            this.cost = cost;
        }
    }
```




# 3. Algorithm 2: Prim-Jarnik Algorithm

Overview

- Purpose: To find the minimum spanning tree (MST) for a weighted graph, which is a subset of the edges that connects all vertices without any cycles and with the minimum possible total edge weight.
- Approach: Greedy algorithm that grows the MST by adding the smallest edge from the tree to a vertex outside the tree.

Steps

1. **Initialize**: Start with an arbitrary node and an empty MST.
2. **Priority Queue**: Use a priority queue to select the smallest edge connecting t he tree to a new vertex.
3. **Update**: Add the selected edge and vertex to the MST and update the priorit y queue with edges connected to the new vertex.
4. **Repeat**: Continue until all vertices are included in the MST.

## For example:

- Consider a graph with nodes A, B, C, D, and E. The algorithm constructs an MST starting from A.
- **Initial State**:
  - MST: {A}, Edges: []
- **Processing**:
  - Select the smallest edge from A, update the MST.
  - Continue adding the smallest edges connecting the MST to new vertices.
- **Result**:
  - MST covering all nodes with the minimum total edge weight.

# 4. Performance Analysis

| Algorithm | Time Complexity | Space Complexity | Stability |
|---|---|---|---|
| Dijkstra's Algorithm | $O(V^2)$ (with array), $O(E + V\log V)$ (with priority queue) | $O(V)$ | Yes |
| Prim-Jarnik Algorithm | $O(V^2)$ (with adjacency matrix), $O(E + V\log V)$ (with priority queue) | $O(V)$ | Yes |

- **Dijkstra's Algorithm**: Efficient for sparse graphs using a priority queue. Suitable for single-source shortest path problems.
- **PrimJarnik Algorithm**: Efficient for dense graphs using an adjacency matrix. Ideal for constructing minimum spanning trees.

# Conclusion

Through this presentation, we have clearly seen how abstract data types (ADTs) and sorting and pathfinding algorithms play an important role in improving software performance. The correct application of data structures and algorithms helps optimize data processing, increase flexibility and ensure system efficiency. This knowledge will be a solid foundation for developing more complex software solutions in the future, especially in the working environment at Soft Development ABK.

# Thanks for watching