Welcome to my presentation.

1. Identify Data Structures

Data Structure: Stack

- Characteristics:
  - LIFO (Last In, First Out).
  - Supports operations to add, remove, and retrieve the top element.

2. Define the Operations

Operations to Perform:

1. **Push: Add a student to the stack.**
2. **Pop: Remove and return the student at the top of the stack.**
3. **Peek/Top: Retrieve the student at the top of the stack without removing it.**
4. **Search: Look for a student by their ID.**
5. **Edit: Modify the name and marks of a student.**
6. **Delete: Remove a student from the stack.**
7. **Sort: Sort students by marks.**

3. Specify Input Parameters

- **Push:**
  - **Parameters: StudentID, StudentName, Marks**
- **Search:**
  - **Parameter: StudentID**
- **Edit:**
  - **Parameters: StudentID, NewStudentName, NewMarks**
- **Delete:**
  - **Parameter: StudentID**
- **Sort:**
  - **No input parameters (sorts all students in the stack).**

4. Define Pre- and Post-conditions

Pre-conditions:

- The stack must not be empty when performing Pop, Peek, or Search.
- A student with StudentID must exist in the stack when performing Edit or Delete.

Post-conditions:

- Push: Stack size increases.
- Pop: Stack size decreases, and the student is returned.
- Peek: Stack remains unchanged, and the student is returned.
- Search: Returns student information if found, or indicates not found.
- Edit: Updates the name and marks of the student without changing stack size.
- Delete: Stack size decreases, and the student is removed from the stack.
- Sort: Stack is sorted by marks.

5. Discuss Time and Space Complexity

- Time Complexity
- Push / Pop / Peek: O(1) – No need to traverse the stack.
- Search / Edit / Delete: O(n) – Must traverse the stack to find a specific student.
- Sort: O(n log n) – Sorting using algorithms like Quick Sort or Merge Sort.
- Space Complexity
- O(n) – Requires memory proportional to the number of students in the stack.

Examples and Code Snippets

```java
class Student {  no usages
    String studentID;  2 usages
    String studentName;  2 usages
    double marks;  2 usages

    public Student(String studentID, String studentName, double marks) {  no usages
        this.studentID = studentID;
        this.studentName = studentName;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return "ID: " + studentID + ", Name: " + studentName + ", Marks: " + marks;
    }
}
```

Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1. Define a Memory Stack

A Memory Stack is a data structure that operates in a Last In, First Out (LIFO) manner, meaning the last item added to the stack is the first one to be removed. It is used primarily for managing function calls, local variables, and program execution flow in a computer's memory.

- **Structure:**
  - **The stack consists of a series of memory addresses that point to stored data. Each entry in the stack is called a "stack frame" or "activation record," which contains the information needed to manage the function calls.**

2. Identify Operations

The primary operations that can be performed on a memory stack include:

1. **Push:**
    - **Adds an item (e.g., a stack frame) to the top of the stack.**
    - **Operation: Increases the stack size by one.**
2. **Pop:**
    - **Removes the item from the top of the stack and returns it.**
    - **Operation: Decreases the stack size by one.**
3. **Peek/Top:**
    - **Retrieves the item at the top of the stack without removing it.**
    - **Operation: The stack size remains unchanged.**
4. **IsEmpty:**
    - **Checks if the stack is empty (i.e., whether there are any items in the stack).**
5. **Size:**
    - **Returns the current number of items in the stack.**

3. Function Call Implementation

When a function is called in a programming language, the following steps typically occur:

1. **Creating a Stack Frame:**
   - **When a function is called, a new stack frame is created and pushed onto the stack. This frame contains:**
     - **Local Variables: Variables defined within the function.**
     - **Parameters: Input values passed to the function.**
     - **Return Address: The point in the code where execution should resume after the function completes.**
2. **Executing the Function:**
   - **The program executes the function code, using the local variables and parameters stored in the stack frame.**
3. **Returning from the Function:**
   - **When the function completes, it returns a value (if applicable), and the stack frame is popped from the stack. The return address is then used to resume execution in the calling function.**

Function Call Example

```java
public class FunctionCallExample {

    public static void main(String[] args) {
        int result = calculateSum(5, 10);
        System.out.println("Result: " + result);
    }

    public static int calculateSum(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
```

4. Demonstrate Stack Frames

Stack Frame Components

A stack frame typically contains:

- **Return Address**: Where to return after the function call.
- **Parameters**: Values passed to the function.
- **Local Variables**: Variables defined within the function.
- **Saved Registers**: If necessary, to restore CPU state after returning

Illustration of Stack Frame

```java
public class RecursiveFunctionExample {

    public static void main(String[] args) {
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + " is: " + result);
    }

    public static int factorial(int n) {  2 usages
        if (n == 0) {
            return 1;
        } else {
            return n * factorial( n: n - 1);
        }
    }
}
```

5. Discuss the Importance

The memory stack is crucial for several reasons:

1. **Function Management: It efficiently manages function calls and returns, allowing for recursive function calls and deep nesting.**
2. **Memory Allocation: Local variables are allocated on the stack, providing a quick and temporary storage area that automatically deallocates when the function exits.**
3. **Control Flow: The stack maintains the return addresses, enabling the program to resume execution accurately after function calls.**
4. **Recursion Support: Each recursive call creates a new stack frame, allowing multiple instances of a function to run concurrently without interfering with each other.**
5. **Error Handling: When a function encounters an error, the stack can be unwound to find the appropriate error handling routines, preserving the execution context.**

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

1. Introduction to FIFO

FIFO (First In First Out) is a data structure that operates in such a way that the first element added to the queue is the first one to be removed. This principle is similar to a line of people waiting at a ticket counter: the person who arrives first is the one who is served first.

Key Characteristics of FIFO Queue

- Enqueue: Adds an element to the end of the queue.
- Dequeue: Removes an element from the front of the queue.
- Peek: Retrieves the element at the front without removing it.
- IsEmpty: Checks whether the queue is empty.

2. Define the Structure

A FIFO queue can be implemented using two main structures:

1. **Array-Based Implementation**
2. **Linked List-Based Implementation**

Array-Based Implementation

```java
class ArrayQueue {  no usages
    private int[] queue;  4 usages
    private int front, rear, capacity, size;  5 usages

    public ArrayQueue(int capacity) {  no usages
        this.capacity = capacity;
        queue = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    public void enqueue(int item) {  no usages
        if (size == capacity) {
            System.out.println("Queue is full!");
            return; // Queue is full
        }
        rear = (rear + 1) % capacity; // Circular increment
        queue[rear] = item;
        size++;
    }
```

```java
    public int dequeue() {  no usages
        if (size == 0) {
            System.out.println("Queue is empty!");
            return -1; // Queue is empty
        }
        int item = queue[front];
        front = (front + 1) % capacity; // Circular increment
        size--;
        return item;
    }

    public int peek() {  no usages
        if (size == 0) {
            System.out.println("Queue is empty!");
            return -1; // Queue is empty
        }
        return queue[front];
    }

    public boolean isEmpty() {  no usages
        return size == 0;
    }
}
```

Linked List-Based Implementation

```java
class Node {  4 usages
    int data;  3 usages
    Node next;  3 usages

    public Node(int data) {  1 usage
        this.data = data;
        this.next = null;
    }
}

class LinkedListQueue {  no usages
    private Node front, rear;  10 usages

    public LinkedListQueue() {  no usages
        front = rear = null;
    }

    public void enqueue(int item) {  no usages
        Node newNode = new Node(item);
        if (rear == null) {
            front = rear = newNode; // Queue was empty
            return;
        }
        rear.next = newNode; // Link the old rear to the new node
        rear = newNode; // Move rear to the new node
    }
```

```java
    public int dequeue() {  no usages
        if (front == null) {
            System.out.println("Queue is empty!");
            return -1; // Queue is empty
        }
        int item = front.data;
        front = front.next; // Move front to the next node
        if (front == null) {
            rear = null; // If the queue is now empty
        }
        return item;
    }

    public int peek() {  no usages
        if (front == null) {
            System.out.println("Queue is empty!");
            return -1; // Queue is empty
        }
        return front.data;
    }

    public boolean isEmpty() {  no usages
        return front == null;
    }
}
```

Provide a Concrete Example to Illustrate How the FIFO Queue Works

Example Using Array-Based Implementation

```java
public class ArrayQueue {
    private int[] queue;
    private int front, rear, size, capacity;

    public ArrayQueue(int capacity) {
        this.capacity = capacity;
        queue = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full!");
            return;
        }
        rear = (rear + 1) % capacity; // Circular increment
        queue[rear] = item;
        size++;
    }
```

```java
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return -1; // Or throw an exception
        }
        int item = queue[front];
        front = (front + 1) % capacity; // Circular increment
        size--;
        return item;
    }

    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return -1; // Or throw an exception
        }
        return queue[front];
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean isFull() {
        return size == capacity;
    }
}
```

```java
public class Example {
    public static void main(String[] args) {
        ArrayQueue queue = new ArrayQueue(5);

        // Enqueue elements
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.enqueue(40);
        queue.enqueue(50);

        // Attempt to enqueue another element (Queue is full)
        queue.enqueue(60); // Output: Queue is full!

        // Dequeue elements
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 10
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 20

        // Current front element
        System.out.println("Front element: " + queue.peek()); // Output: 30

        // Enqueue another element
        queue.enqueue(60);
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 30
    }
}
```

result

```
Queue is full!
Dequeued: 10
Dequeued: 20
Front element: 30
Dequeued: 30

Process finished with exit code 0
```

Example Using Linked List-Based Implementation

```java
class Node {  4 usages
    int data;  3 usages
    Node next;  3 usages

    public Node(int data) {  1 usage
        this.data = data;
        this.next = null;
    }
}

// Linked List Queue implementation
public class LinkedListQueue {  2 usages
    private Node front, rear;  10 usages

    public LinkedListQueue() {  1 usage
        front = rear = null;
    }

    public void enqueue(int item) {  6 usages
        Node newNode = new Node(item);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }
```

```java
public int dequeue() {  3 usages
    if (front == null) {
        System.out.println("Queue is empty!");
        return -1; // Or throw an exception
    }
    int item = front.data;
    front = front.next;
    if (front == null) {
        rear = null; // If the queue becomes empty
    }
    return item;
}

public int peek() {  1 usage
    if (front == null) {
        System.out.println("Queue is empty!");
        return -1; // Or throw an exception
    }
    return front.data;
}

public boolean isEmpty() {  no usages
    return front == null;
}
```

```java
// Example usage
class Example {
    public static void main(String[] args) {
        LinkedListQueue queue = new LinkedListQueue();

        // Enqueue elements
        queue.enqueue( item: 10);
        queue.enqueue( item: 20);
        queue.enqueue( item: 30);
        queue.enqueue( item: 40);
        queue.enqueue( item: 50);

        // Dequeue elements
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 10
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 20

        // Current front element
        System.out.println("Front element: " + queue.peek()); // Output: 30

        // Enqueue another element
        queue.enqueue( item: 60);
        System.out.println("Dequeued: " + queue.dequeue()); // Output: 30
    }
}
```

result

```
Dequeued: 10
Dequeued: 20
Front element: 30
Dequeued: 30


Process finished with exit code 0
```

Compare the performance of two sorting algorithms.

1. Introducing the Two Sorting Algorithms

A. Quick Sort

- **Overview: Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. It recursively sorts the sub-arrays.**
- **Best Use Cases: Efficient for large datasets, works well in practice and is often faster than other O(nlogn)O(n \log n)O(nlogn) algorithms.**

B. Merge Sort

- **Overview: Merge Sort is also a divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves back together.**
- **Best Use Cases: Suitable for linked lists and external sorting algorithms.**

2. Time Complexity Analysis

Algorithm	Best Case	Average Case	Worst Case
Quick Sort	O(nlogn)	O(nlogn)	O(n2)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)

- **Quick Sort:**
- **Best/Average: Balanced partitions.**
- **Worst: Unbalanced partitions with poor pivot choices.**
- **Merge Sort:**
- **Consistent performance in all cases.**

3. Space Complexity Analysis

Algorithm	Space Complexity
Quick Sort	O(logn) (in-place, due to recursion stack)
Merge Sort	O(n) (requires additional space for merging)

- **Quick Sort: It requires space on the stack for recursive calls, leading to a space complexity of O(logn)O(\log n)O(logn). It is considered an in-place sorting algorithm since it does not require extra space for a secondary array.**
- **Merge Sort: It requires additional space proportional to the size of the input array for temporary arrays during the merging process, resulting in a space complexity of O(n)O(n)O(n).**

4. Stability

Algorithm	Space Complexity
Quick Sort	Unstable
Merge Sort	Stable

Quick Sort: It is not stable, meaning that the relative order of equal elements may not be preserved after sorting.
Merge Sort: It is stable, preserving the relative order of equal elements, which can be important for certain applications.

5. Comparison Table

Feature	Quick Sort	Merge Sort
Time Complexity	O(nlogn) (avg)	O(nlogn)
Space Complexity	O(logn)	O(n)
Stability	Unstable	Stable
In-Place	Yes	No
Best for	Large datasets	Linked lists, external sorting

6. Performance Comparison

Concrete Example

Let's sort the following array:

Array: [38,27,43,3,9,82,10]

A. Quick Sort Implementation in Java

```java
public class Test {  no usages
    public class QuickSort {  no usages
        public static void quickSort(int[] arr, int low, int high) {  2 usages
            if (low < high) {
                int pi = partition(arr, low, high);
                quickSort(arr, low,  high: pi - 1);
                quickSort(arr,  low: pi + 1, high);
            }
        }

        private static int partition(int[] arr, int low, int high) {  1 usage
            int pivot = arr[high];
            int 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;
            return i + 1;
        }
    }

}
```

B. Merge Sort Implementation in Java

```java
public class Test {
    public class MergeSort {
        public static void mergeSort(int[] arr, int l, int r) {
            if (l < r) {
                int m = (l + r) / 2;
                mergeSort(arr, l, m);
                mergeSort(arr, m + 1, r);
                merge(arr, l, m, r);
            }
        }
        private static void merge(int[] arr, int l, int m, int r) {
            int n1 = m - l + 1;
            int n2 = r - m;
            int[] L = new int[n1];
            int[] R = new int[n2];
            for (int i = 0; i < n1; i++) L[i] = arr[l + i];
            for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
            int i = 0, j = 0;
            int k = l;
            while (i < n1 && j < n2) {
                if (L[i] <= R[j]) {
                    arr[k++] = L[i++];
                } else {
                    arr[k++] = R[j++];
                }
            }
            while (i < n1) arr[k++] = L[i++];
            while (j < n2) arr[k++] = R[j++];
        }
    }
```

result

```
Sorted array using Quick Sort: [3, 9, 10, 27, 38, 43, 82]
Sorted array using Merge Sort: [3, 9, 10, 27, 38, 43, 82]
```

Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

1. Introduction to Network Shortest Path Algorithms

Network shortest path algorithms are essential for determining the shortest paths between nodes in a graph, which can represent various real-world scenarios such as road networks, telecommunication networks, and data routing. These algorithms help optimize routes, reduce costs, and improve efficiency in numerous applications.

2. Algorithm 1: Dijkstra's Algorithm

Overview

Dijkstra's Algorithm finds the shortest path from a starting node (source) to all other nodes in a weighted graph. It works on both directed and undirected graphs but requires that all edge weights be non-negative.

Steps

1. Initialize distances from the source to all nodes as infinity and the distance to the source itself as zero.
2. Use a priority queue to repeatedly extract the node with the smallest distance.
3. Update the distances of the neighboring nodes.
4. Repeat until all nodes are processed.

result

Illustration

Example Graph for Dijkstra's Algorithm:

- **Starting Node: A**
- **Shortest Paths:**
  - **A to B: 4**
  - **A to C: 1**
  - **A to D: 3**

```
Vertex Distance from Source
0        0
1        4
2        12
3        19
4        21
5        11
6        9
7        8
8        14
```

3. Algorithm 2: Prim-Jarnik Algorithm

Overview

Prim's Algorithm (or Prim-Jarnik) is used to find the Minimum Spanning Tree (MST) of a weighted undirected graph. It starts from an arbitrary node and grows the MST by adding the smallest edge that connects a vertex in the MST to a vertex outside it.

Steps

1. Initialize a priority queue and select an arbitrary starting vertex.
2. Mark the vertex as included in the MST.
3. Add all edges from the selected vertex to the priority queue.
4. Repeat until all vertices are included.

result

Illustration
Example Graph for Prim's Algorithm:
Starting Node: A
Edges Included:
A to B
A to C
B to D

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

4. Performance Analysis

Dijkstra's Algorithm

- **Time Complexity:**
  - Using a priority queue (binary heap): O((V+E)logV)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges.
- **Space Complexity:**
  - O(V)O(V)O(V) for storing distances and priority queue.

Prim-Jarnik Algorithm

- **Time Complexity:**
  - Using a priority queue (binary heap): O(ElogV)O(E \log V)O(ElogV).
- **Space Complexity:**
  - O(V)O(V)O(V) for storing keys and MST.

Thanks for watching