**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Solutions:**

1. **Understanding:**

#### Types of Linked Lists

**Singly Linked List**

* 1. Each node contains data and a reference (or pointer) to the next node in the sequence.
  2. Allows traversal in one direction (forward).
  3. Operations such as insertion and deletion are efficient if the position is known.

**Doubly Linked List**

* 1. Each node contains data, a reference to the next node, and a reference to the previous node.
  2. Allows traversal in both directions (forward and backward).
  3. Requires more memory per node due to the extra reference.

1. **Setup and Implementation :**

public class Task {

private String taskId;

private String taskName;

private String status;

public Task(String taskId, String taskName, String status) {

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

@Override

public String toString() {

return "Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

public class TaskManagementSystem {

private class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

private Node head;

public TaskManagementSystem() {

this.head = null;

}

// Add a new task

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Search for a task by taskId

public Task searchTask(String taskId) {

Node current = head;

while (current != null) {

if (current.task.getTaskId().equals(taskId)) {

return current.task;

}

current = current.next;

}

return null;

}

// Traverse and display all tasks

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Delete a task by taskId

public void deleteTask(String taskId) {

if (head == null) {

System.out.println("List is empty.");

return;

}

if (head.task.getTaskId().equals(taskId)) {

head = head.next;

return;

}

Node current = head;

while (current.next != null && !current.next.task.getTaskId().equals(taskId)) {

current = current.next;

}

if (current.next == null) {

System.out.println("Task not found.");

} else {

current.next = current.next.next;

}

}

public static void main(String[] args) {

TaskManagementSystem tms = new TaskManagementSystem();

Task task1 = new Task("1", "Design system architecture", "Pending");

Task task2 = new Task("2", "Implement login feature", "In Progress");

Task task3 = new Task("3", "Test application", "Pending");

tms.addTask(task1);

tms.addTask(task2);

tms.addTask(task3);

System.out.println("All Tasks:");

tms.traverseTasks();

System.out.println("\nSearching for Task with ID 2:");

Task searchedTask = tms.searchTask("2");

System.out.println(searchedTask != null ? searchedTask : "Task not found");

System.out.println("\nDeleting Task with ID 2:");

tms.deleteTask("2");

System.out.println("\nAll Tasks after deletion:");

tms.traverseTasks();

}

}

1. **Analysis:**

#### Time Complexity of Each Operation

1. **Add Operation**: O(n) (since you need to traverse to the end of the list)
2. **Search Operation**: O(n) (linear search through the list)
3. **Traverse Operation**: O(n) (visiting each node once)
4. **Delete Operation**: O(n) (finding the node to delete requires traversal)

#### Advantages of Linked Lists Over Arrays for Dynamic Data

1. **Dynamic Size**: Linked lists can grow and shrink dynamically, making them more flexible than arrays which have a fixed size.
2. **Efficient Insertions/Deletions**: Insertion and deletion operations are more efficient (O(1) for inserting/deleting at the beginning) compared to arrays, where shifting elements can be costly (O(n)).
3. **Memory Usage**: Linked lists do not require contiguous memory allocation, which can be a limitation for very large arrays.

**Disadvantages of Linked Lists**:

1. **Memory Overhead**: Each node requires additional memory for storing the reference to the next node.
2. **Sequential Access**: Linked lists do not support direct access to elements by index, making some operations slower compared to arrays (O(n) for search).