Task Management System with Singly Linked List -EX 5

1. Understanding Linked Lists

Types of Linked Lists:

- **Singly Linked List**: Each node contains data and a reference to the next node. It's unidirectional, meaning traversal is possible only in one direction, from head to the last node.
- **Doubly Linked List**: Each node contains data and references to both the next and previous nodes. It allows traversal in both directions (forward and backward). It requires extra memory for storing an additional reference per node.

2. Setup

Class Definition: Task

The Task class includes attributes like taskId, taskName, and status.

```
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 + ", Task Name: " + taskName + ", Status: " + status;
    }
}
```

3. Implementation

Singly Linked List for Managing Tasks

```
class Node {
    Task task;
    Node next;

    public Node(Task task) {
        this.task = task;
        this.next = null;
    }
}

public class TaskManagement {
    private Node head;

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

    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;
    }
    System.out.println("Task added successfully.");
}

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;
}

public void traverseTasks() {
    if (head == null) {
        System.out.println("No tasks in the list.");
    } else {
        Node current = head;
```

```
        while (current != null) {
            System.out.println(current.task);
            current = current.next;
        }
    }
}

public void deleteTask(String taskId) {
    if (head == null) {
        System.out.println("No tasks in the list.");
        return;
    }

    if (head.task.getTaskId().equals(taskId)) {
        head = head.next;
        System.out.println("Task deleted successfully.");
        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;
        System.out.println("Task deleted successfully.");
```

```
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    TaskManagement tms = new TaskManagement();

    while (true) {
        System.out.println("\nTask Management System:");
        System.out.println("1. Add Task");
        System.out.println("2. Search Task");
        System.out.println("3. Traverse Tasks");
        System.out.println("4. Delete Task");
        System.out.println("5. Exit");
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();
        scanner.nextLine();

        switch (choice) {
            case 1:
                System.out.print("Enter Task ID: ");
                String id = scanner.nextLine();
                System.out.print("Enter Task Name: ");
                String name = scanner.nextLine();
                System.out.print("Enter Task Status: ");
                String status = scanner.nextLine();

                Task newTask = new Task(id, name, status);
                tms.addTask(newTask);
```

```
            break;

        case 2:
            System.out.print("Enter Task ID to search: ");
            String searchId = scanner.nextLine();
            Task task = tms.searchTask(searchId);
            if (task != null) {
                System.out.println("Task found: " + task);
            } else {
                System.out.println("Task not found.");
            }
            break;

        case 3:
            System.out.println("Task List:");
            tms.traverseTasks();
            break;

        case 4:
            System.out.print("Enter Task ID to delete: ");
            String deleteId = scanner.nextLine();
            tms.deleteTask(deleteId);
            break;

        case 5:
            System.out.println("Exiting the system. Goodbye!");
            scanner.close();
            return;
```

```
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    }
}
}
```

4. Analysis

Time Complexity Analysis:

- **Add**: O(n) in the worst case, as it requires traversal to the end of the list to add a new node.
- **Search**: O(n) in the worst case, since it may require checking each node until the desired task is found.
- **Traverse**: O(n) as it involves visiting each node in the list.
- **Delete**: O(n) in the worst case, because it may require traversal to find the node before the one to be deleted.

Advantages of Linked Lists over Arrays:

1. **Dynamic Size**: Linked lists can grow or shrink dynamically, unlike arrays which have a fixed size. This flexibility makes linked lists more suitable for scenarios where the number of elements can change frequently.
2. **Efficient Insertions/Deletions**: Inserting or deleting elements in a linked list does not require shifting elements, as in arrays. This can be more efficient, especially for large lists where such operations are frequent.
3. **Memory Utilization**: Linked lists allocate memory as needed for each element, potentially saving memory compared to arrays, which may reserve extra space in anticipation of future additions.