Exercise 5: Task Management System

1. Understanding the Problem

In a task management system where users need to add, delete, update, or view tasks efficiently, the choice of the right data structure is **crucial**. Linked lists are particularly useful here due to their **dynamic memory allocation** and **ease of insertion/deletion operations**.

Why Are Linked Lists Important?

➤ Efficient Insertion and Deletion:

In arrays, inserting or deleting a task (especially in the middle) requires shifting elements, which is time-consuming (O(n)). But in a linked list, tasks can be added or removed without shifting — only pointer changes are needed (O(1) for head, O(n) for others).

➤ Dynamic Memory Allocation:

Linked lists grow and shrink as needed. There's no need to declare the size in advance (unlike arrays), making them perfect for applications where the number of tasks is unknown or frequently changing.

➤ Traversal Support:

Linked lists allow easy traversal from the first task to the last — ideal for displaying all tasks in order.

2. Understanding Types of Linked Lists

Choosing the right type of linked list depends on the needs of the system:

1. Singly Linked List

- Each node stores a reference to the **next** node only.
- Best for one-way traversal.
- Simple and memory-efficient.

Structure:

[Task1] → [Task2] → [Task3] → null

2. Doubly Linked List

- Each node stores a reference to **both the next and previous** nodes.
- Allows **two-way traversal** and **faster deletions**, especially when the node reference is available.

Structure:

null ← [Task1] ↔ [Task2] ↔ [Task3] → null

For most task management systems, **singly linked lists** are sufficient unless backward traversal or fast middle deletions are required.

3. Suitable Data Structures for Task Management

- Add new tasks
- Delete existing tasks

- Search tasks by ID or name
- Display all tasks in order

4. Time Complexity Analysis (Singly Linked List)

```
class Task {
   int taskId;
   String taskName;
   String status;
   Task next;
}
```

➤ Adding a Task

- **At the head**: O(1)
- **At the end**: O(n) — need to traverse to the last node

```
task.next = head;
head = task;
```

➤ Deleting a Task by ID

- Requires traversal to find the task and its previous node.
- **Time Complexity**: O(n)

```
 Task current = head, prev = null;
while (current != null && current.taskId != id) {
   prev = current;
   current = current.next;
}
```

➤ **Searching a Task by ID**

- Sequential traversal required
- **Time Complexity**: O(n)

➤ **Traversing All Tasks**

- Simply loop through all nodes from head to null
- **Time Complexity**: O(n)

6. Optimizations and Best Practices

1. **Efficient Head Operations:**

   Insert/delete at the head is always O(1). Use it when order is not strict.

2. **Avoid Memory Leaks:**

   Always set removed nodes' next pointers to null.

3. **Use Object-Oriented Design:**

   Wrap logic in a class like TaskManager for clean code.

4. **Future-Proofing:**

If you later need backward traversal, switch to a doubly linked list.