**DATASTRUCTURES AND ALGORITHMS:**

**Exercise 5-Task Management System**

**Step 1: Understanding Linked Lists**

**Linked List** is a linear data structure where each element (called a node) contains two parts:

* Data (the value)
* A reference (or pointer) to the next node

There are mainly two types of linked lists:

1. **Singly Linked List (SLL):**
   * Each node points to the next node.
   * Traversal is **one-way**, from head to tail.
   * Memory-efficient but cannot go backward.
2. **Doubly Linked List (DLL):**
   * Each node contains two pointers: one to the next node and one to the previous node.
   * Can be traversed **both forward and backward**.
   * Slightly more complex and uses more memory.

For this exercise, we will use a **Singly Linked List** to manage tasks.

**✅ Step 2: Setup – Task Class**

We create a class Task with these attributes:

* taskId (int)
* taskName (String)
* status (String) → Example: "Pending", "Completed"

**Task.java**

class Task {

int taskId;

String taskName;

String status;

Task next;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

void display() {

System.out.println("ID: " + taskId + ", Task: " + taskName + ", Status: " + status);

}

}

**TaskManagementSystem.java**

public class TaskManagementSystem {

Task head;

// Add a new task at the end

public void addTask(int id, String name, String status) {

Task newTask = new Task(id, name, status);

if (head == null) {

head = newTask;

} else {

Task temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newTask;

}

}

// Search a task by ID

public Task searchTask(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id) {

return temp;

}

temp = temp.next;

}

return null;

}

// Traverse all tasks

public void displayTasks() {

Task temp = head;

while (temp != null) {

temp.display();

temp = temp.next;

}

}

// Delete a task by ID

public void deleteTask(int id) {

if (head == null) {

System.out.println("Task list is empty.");

return;

}

if (head.taskId == id) {

head = head.next;

System.out.println("Task deleted.");

return;

}

Task prev = head;

Task curr = head.next;

while (curr != null) {

if (curr.taskId == id) {

prev.next = curr.next;

System.out.println("Task deleted.");

return;

}

prev = curr;

curr = curr.next;

}

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

}

// Main method to test

public static void main(String[] args) {

TaskManagementSystem tms = new TaskManagementSystem();

tms.addTask(1, "Design Homepage", "Pending");

tms.addTask(2, "Build Login API", "Completed");

tms.addTask(3, "Test Signup Flow", "Pending");

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

tms.displayTasks();

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

Task found = tms.searchTask(2);

if (found != null) {

found.display();

} else {

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

}

System.out.println("\nDeleting Task with ID 1:");

tms.deleteTask(1);

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

tms.displayTasks();

}

}

**OUTPUT:**

A black screen with white text

AI-generated content may be incorrect.

**Time Complexity:**

| **Operation** | **Time Complexity** | **Reason** |
| --- | --- | --- |
| Add | O(n) | Need to traverse till the end of the list |
| Search | O(n) | Might have to check each node |
| Traverse | O(n) | Each node is visited once |
| Delete | O(n) | May need to find and unlink a node |

**✅ Advantages of Linked Lists over Arrays**

* **Dynamic Size:** Linked lists can grow or shrink as needed without wasting memory.
* **Efficient Deletion/Insertion:** Adding/removing elements doesn’t require shifting, unlike arrays.
* **No need to know size in advance:** Arrays need a fixed size, linked lists don’t.

**✅ When to Use Linked Lists**

* When data size changes frequently.
* When frequent insertions and deletions are required.
* When memory allocation needs to be optimized.