**Exercise 5: Task Management System**

**Linked-List Fundamentals**

**🔹 Singly Linked List**

Each node stores **data + a single reference to the next node**. Traversal is strictly forward; there is no direct way back.  
*Typical uses*: lightweight stacks, queues, and any collection that grows and shrinks at the front or middle.

**🔹 Doubly Linked List**

Every node keeps **two references**—one to the next node and one to the previous—allowing bidirectional traversal and constant-time deletion when you already hold the node reference.  
*Trade-off*: extra memory per node plus slightly more pointer housekeeping.

For a simple task tracker that only needs forward iteration, a **singly linked list** is sufficient and lean.

**Source Code:**

**Task.java**

public class Task {

private final 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; }

public void setTaskName(String taskName) { this.taskName = taskName; }

public void setStatus(String status) { this.status = status; }

@Override

public String toString() {

return taskId + " | " + taskName + " | " + status;

}

}

**TaskList.java**

public class TaskList {

private static class Node {

Task data;

Node next;

Node(Task data) { this.data = data; }

}

private Node head;

public void add(Task t) { // insert at front

Node n = new Node(t);

n.next = head;

head = n;

}

public Task search(String id) { // linear scan

for (Node cur = head; cur != null; cur = cur.next) {

if (cur.data.getTaskId().equals(id)) return cur.data;

}

return null;

}

public boolean delete(String id) { // remove first node with matching id

Node cur = head, prev = null;

while (cur != null) {

if (cur.data.getTaskId().equals(id)) {

if (prev == null) head = cur.next;

else prev.next = cur.next;

return true;

}

prev = cur;

cur = cur.next;

}

return false;

}

public void traverse() {

for (Node cur = head; cur != null; cur = cur.next)

System.out.println(cur.data);

}

}

**Demo.java**

public class Demo {

public static void main(String[] args) {

TaskList list = new TaskList();

list.add(new Task("T101", "Design schema", "OPEN"));

list.add(new Task("T102", "Write API", "IN-PROGRESS"));

list.add(new Task("T103", "Deploy app", "PENDING"));

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

list.traverse();

System.out.println("\nSearch T102:");

System.out.println(list.search("T102"));

System.out.println("\nDelete T101:");

list.delete("T101");

System.out.println("\nAfter deletion:");

list.traverse();

}

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Complexity Analysis**

* **Add (front insertion)** – constant time **O(1)** because only the head pointer changes.
* **Search** – linear **O(n)**; you may examine every node in the worst case.
* **Traverse** – also **O(n)** because each node is visited once.
* **Delete** – requires a search to find the node, plus at most one pointer update, so **O(n)** overall.

**Why Linked Lists for Dynamic Tasks?**

* **No pre-allocation** – the list grows or shrinks exactly as tasks are created or removed; there is no wasted or reallocated block as with arrays.
* **Cheap inserts/deletes** at the head or immediately after a known node: only pointer rewiring, no element shifting.
* **Stable node addresses** – references to a task node remain valid after other insertions or removals, unlike arrays that may move data during resizing.

**When arrays still win**

If you need **random index access** or plan to traverse far more often than you modify, an array or an ArrayList is faster due to CPU-cache friendliness and O(1) indexing. For predominantly add/delete workloads where order is flexible, the linked list shines.