# Exercise 5: Task Management System

### 

### **Understand Linked Lists**

**Singly Linked List:**

* **Description:** A linked list where each node contains data and a reference (or link) to the next node in the sequence. The last node points to null, indicating the end of the list.
* **Structure:**
  + **Node:** Contains the data and a pointer to the next node.
  + **Head:** The starting node of the list.

**Doubly Linked List:**

* **Description:** A linked list where each node contains data and two references: one to the next node and one to the previous node. This allows traversal in both directions.
* **Structure:**
  + **Node:** Contains the data, a pointer to the next node, and a pointer to the previous node.
  + **Head:** The starting node of the list.
  + **Tail:** The ending node of the list.

### **Step 2: Setup**

**Task Class:**

class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

// Getters

public int getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

@Override

public String toString() {

return "Task [taskId=" + taskId + ", taskName=" + taskName + ", status=" + status + "]";

}

}

### **Step 3: Implementation**

**Singly Linked List for Task Management:**

public class TaskManagementSystem {

private class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

private Node head;

public TaskManagementSystem() {

head = null;

}

// Method to add a task

public void addTask(Task task) {

Node newNode = new Node(task);

newNode.next = head;

head = newNode;

}

// Method to search for a task by taskId

public Task searchTask(int taskId) {

Node current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null; // Task not found

}

// Method to traverse and print all tasks

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Method to delete a task by taskId

public void deleteTask(int taskId) {

if (head == null) return;

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

head = head.next;

return;

}

Node current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

}

}

}

### **Step 4: Analysis**

**Time Complexity:**

* **Add Operation:**
  + Best Case: O(1)O(1)O(1)
  + Average Case: O(1)O(1)O(1)
  + Worst Case: O(1)O(1)O(1)
* **Search Operation:**
  + Best Case: O(1)O(1)O(1) (if the task is at the head)
  + Average Case: O(n)O(n)O(n)
  + Worst Case: O(n)O(n)O(n)
* **Traverse Operation:**
  + Best Case: O(n)O(n)O(n)
  + Average Case: O(n)O(n)O(n)
  + Worst Case: O(n)O(n)O(n)
* **Delete Operation:**
  + Best Case: O(1)O(1)O(1) (if the task to be deleted is the head)
  + Average Case: O(n)O(n)O(n)
  + Worst Case: O(n)O(n)O(n)

**Advantages of Linked Lists over Arrays:**

* **Dynamic Size:** Linked lists can grow and shrink in size dynamically, making them more flexible than arrays, which have a fixed size.
* **Efficient Insertions/Deletions:** Insertions and deletions can be more efficient as they do not require shifting elements, as is the case with arrays.
* **Memory Utilization:** Linked lists do not need to allocate memory in advance; they use memory as needed for each node.

**Limitations of Linked Lists:**

* **Memory Overhead:** Each node requires extra memory for storing the reference to the next node.
* **Sequential Access:** Linked lists do not support efficient random access; elements must be accessed sequentially from the head, which can be slower than indexed access in arrays.
* **Cache Performance:** Due to non-contiguous memory allocation, linked lists may have poorer cache performance compared to arrays.

Linked lists are particularly useful when you need a dynamic data structure with frequent insertions and deletions. Arrays, on the other hand, are better suited for scenarios where fast indexed access is required and the size of the data set is known in advance.