## Exercise 5: Task Management System

### Step 1: Understand Linked Lists

### Explain the different types of linked lists (Singly Linked List, Doubly Linked List).

#### Singly Linked List

* **Description**: A singly linked list is a collection of nodes where each node contains data and a reference (or pointer) to the next node in the sequence.
* **Structure**: Each node has two fields:
  + data: The value or information the node holds.
  + next: A reference to the next node in the list.
* **Operations**: Basic operations include insertion, deletion, traversal, and search.

#### Doubly Linked List

* **Description**: A doubly linked list is similar to a singly linked list, but each node has two references: one to the next node and one to the previous node.
* **Structure**: Each node has three fields:
  + data: The value or information the node holds.
  + next: A reference to the next node in the list.
  + prev: A reference to the previous node in the list.
* **Operations**: Supports the same operations as a singly linked list but allows for more efficient deletion and backward traversal.

### Step 2: Setup

#### Define the Task Class

// Java implementation

public 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;

}

// Getters and setters

public String getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

}

### Step 3: Implementation

#### Singly Linked List Implementation

// Java implementation of Singly Linked List

public class SinglyLinkedList {

private class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.next = null;

}

}

private Node head;

public SinglyLinkedList() {

this.head = null;

}

// Add a task to the linked list

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;

}

}

// Search for a task by taskId

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;

}

// Traverse the linked list

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task.getTaskName() + " - " + current.task.getStatus());

current = current.next;

}

}

// Delete a task by taskId

public boolean deleteTask(String taskId) {

if (head == null) return false;

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

head = head.next;

return true;

}

Node current = head;

while (current.next != null) {

if (current.next.task.getTaskId().equals(taskId)) {

current.next = current.next.next;

return true;

}

current = current.next;

}

return false;

}

}

### Step 4: Analysis

#### Time Complexity of Operations

* **Add Task**:
  + Best Case: O(1) (if adding at the beginning)
  + Average/Worst Case: O(n) (if adding at the end)
* **Search Task**: O(n) (linear search through the list)
* **Traverse Tasks**: O(n) (iterating through all nodes)
* **Delete Task**:
  + Best Case: O(1) (if deleting the first node)
  + Average/Worst Case: O(n) (if deleting a node in the middle or end)

#### Advantages of Linked Lists over Arrays for Dynamic Data

* **Dynamic Size**: Linked lists can grow and shrink dynamically, whereas arrays have a fixed size.
* **Efficient Insertions/Deletions**: Linked lists can efficiently insert and delete nodes, especially at the beginning, without needing to shift elements.

**Memory Utilization**: Linked lists do not need contiguous memory allocation, which can be advantageous for large datasets or when memory is fragmented.