**Exercise 5: Task Management System**

### Step 1: Understanding Linked Lists

**Singly Linked List:**

* A singly linked list is a type of linked list where each node points to the next node in the sequence.
* Each node contains data and a reference (or pointer) to the next node.
* The last node has a reference to null, indicating the end of the list.
* Operations like insertion and deletion are generally efficient since they involve only updating pointers.

**Doubly Linked List:**

* A doubly linked list is a type of linked list where each node has two references: one to the next node and one to the previous node.
* Each node contains data, a reference to the next node, and a reference to the previous node.
* This allows traversal in both directions (forward and backward).
* Doubly linked lists are more flexible than singly linked lists but require more memory due to the additional pointer.

### Step 2: Setup

Let's create a Task class with attributes like taskId, taskName, and status.

public class Task {

private int taskId;

private String taskName;

private String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public int getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

@Override

public String toString() {

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

}

}

## Implementation

Now, we will implement a singly linked list to manage tasks. This includes methods to add, search, traverse, and delete tasks.

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

public class TaskLinkedList {

private Node head;

public TaskLinkedList() {

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(int taskId) {

Node current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null; // Task not found

}

// Traverse and print all tasks

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Delete a task by taskId

public boolean deleteTask(int taskId) {

if (head == null) {

return false; // List is empty

}

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

head = head.next; // Remove the head node

return true;

}

Node current = head;

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

current = current.next;

}

if (current.next == null) {

return false; // Task not found

}

current.next = current.next.next; // Remove the node

return true;

}

}

## Analysis:

### Time Complexity

* **Add Operation**:
  + Average and Worst Case: O(n)O(n)O(n) where nnn is the number of nodes, as you may need to traverse to the end of the list to add the new node.
* **Search Operation**:
  + Average and Worst Case: O(n)O(n)O(n) as it may require checking all nodes in the list to find the desired task.
* **Traverse Operation**:
  + Average and Worst Case: O(n)O(n)O(n) because you need to visit each node to print all tasks.
* **Delete Operation**:
  + Average and Worst Case: O(n)O(n)O(n) since you may need to traverse the list to find and remove the node.

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

* **Dynamic Size**: Linked lists can grow and shrink in size dynamically, whereas arrays have a fixed size once declared.
* **Ease of Insertion/Deletion**: In linked lists, insertion and deletion of nodes can be done more easily (especially at the beginning or middle of the list) compared to arrays, where elements may need to be shifted.
* **Memory Utilization**: Linked lists use memory efficiently by allocating space as needed. Arrays, on the other hand, may allocate more memory than required.

Overall, linked lists provide flexibility and efficient memory usage for dynamic data management, although they have higher overhead due to additional pointers and non-contiguous memory allocation.