**Exercise 5: Task Management System**

**1. Understand Linked Lists**

**Singly Linked List: Each node points to the next node and has no reference to the previous node.**

**Doubly Linked List: Each node has references to both the next and previous nodes, allowing traversal in both directions.**

**2. Setup**

**// Task.java**

**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 void setTaskId(String taskId) {**

**this.taskId = taskId;**

**}**

**public String getTaskName() {**

**return taskName;**

**}**

**public void setTaskName(String taskName) {**

**this.taskName = 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 + '\'' +**

**'}';**

**}**

**}**

**3. Implementation**

**// Node.java**

**class Node {**

**Task task;**

**Node next;**

**Node(Task task) {**

**this.task = task;**

**this.next = null;**

**}**

**}**

**// SinglyLinkedList.java**

**public class SinglyLinkedList {**

**private Node head;**

**public SinglyLinkedList() {**

**this.head = null;**

**}**

**// Add task**

**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 task by ID**

**public Task searchTaskById(String taskId) {**

**Node current = head;**

**while (current != null) {**

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

**return current.task;**

**}**

**current = current.next;**

**}**

**return null; // Task not found**

**}**

**// Traverse tasks**

**public void traverseTasks() {**

**Node current = head;**

**while (current != null) {**

**System.out.println(current.task);**

**current = current.next;**

**}**

**}**

**// Delete task by ID**

**public void deleteTaskById(String taskId) {**

**if (head == null) {**

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

**return;**

**}**

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

**head = head.next;**

**return;**

**}**

**Node current = head;**

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

**current = current.next;**

**}**

**if (current.next != null) {**

**current.next = current.next.next;**

**} else {**

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

**}**

**}**

**}**

**4. Analysis**

**Time Complexity:**

**Add: O(n) - Inserting a task at the end requires traversing the list if the list is not empty. If inserting at the beginning, it is O(1).**

**Search: O(n) - Searching for a task requires scanning through the list.**

**Traverse: O(n) - Traversing the entire list involves visiting each node once.**

**Delete: O(n) - Deleting a task involves finding the node first (linear time), and then adjusting pointers.**

**Advantages of Linked Lists Over Arrays:**

**Dynamic Size: Linked lists can grow or shrink dynamically, unlike arrays which have a fixed size.**

**Efficient Insertions/Deletions: Inserting or deleting elements in a linked list does not require shifting elements, as in arrays. This is particularly advantageous when performing these operations frequently.**

**Memory Utilization: Linked lists use memory more efficiently for dynamic data compared to arrays that may allocate more space than necessary.**