**Exercise 5: Task Management System**

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

Types of Linked Lists:

1. Singly Linked List
   * Each node points to the next node only.
   * Memory efficient.
   * Operations: forward traversal only.
   * Example: Node -> Node -> Node -> null
2. Doubly Linked List
   * Each node has a next and a prev pointer.
   * Allows forward and backward traversal.
   * Slightly more memory usage.
   * Example: null <- Node <-> Node <-> Node -> null

**Code:**

**Task Class: -**

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 void setTaskId(int 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;  
 }  
  
 public void display(){  
 System.*out*.println("Task ID: "+taskId+", Name: "+taskName+", Status: "+status);  
 }  
}

**TaskManager Class: -**

class Node{  
 Task task;  
 Node next;  
 public Node(Task task){  
 this.task=task;  
 this.next=null;  
 }  
}  
public class TaskManager {  
 private Node head;  
 public void add(Task task){  
 if(task==null){  
 System.*out*.println("Cannot add as Task is Empty");  
 return;  
 }  
 Node newNode=new Node(task);  
 if(head==null){  
 head=newNode;  
 }else{  
 Node temp=head;  
 while(temp.next!=null){  
 temp=temp.next;  
 }  
 temp.next=newNode;  
 }  
 System.*out*.println("Task added");  
 }  
  
 public void traverse(){  
 if(head==null){  
 System.*out*.println("No tasks available");  
 return;  
 }  
 Node temp=head;  
 while(temp!=null){  
 temp.task.display();  
 temp=temp.next;  
 }  
 }  
  
 public void search(int taskId){  
 Node temp=head;  
 while(temp!=null){  
 if(temp.task.getTaskId()==taskId){  
 System.*out*.println("Task found:");  
 temp.task.display();  
 return;  
 }  
 temp=temp.next;  
 }  
 System.*out*.println("Task with ID "+taskId+" not found.");  
 }  
  
 public void delete(int taskId){  
 if(head==null){  
 System.*out*.println("List is Empty.");  
 return;  
 }  
 if(head.task.getTaskId()==taskId){  
 head=head.next;  
 }  
 else{  
 Node prev=null;  
 Node curr=head;  
 while(curr!=null && curr.task.getTaskId()!=taskId){  
 prev=curr;  
 curr=curr.next;  
 }  
 if(curr==null){  
 System.*out*.println("Task with ID "+taskId+" not found.");  
 return;  
 }  
 if(curr.next==null) prev.next=null;  
 else prev.next=curr.next;  
 }  
  
 System.*out*.println("Deleted task with ID "+taskId);  
 }  
  
  
}

**Main Class: -**

public class Main {  
 public static void main(String[] args){  
 TaskManager manager=new TaskManager();  
 manager.add(new Task(1,"Complete the UI","Pending"));  
 manager.add(new Task(2,"Implement Backend","In Progress"));  
 manager.add(new Task(3,"Write Test Cases","Pending"));  
  
 System.*out*.println("\nAll Tasks:");  
 manager.traverse();  
  
 System.*out*.println("\nSearching Task:");  
 manager.search(2);  
  
 System.*out*.println("\nDeleting Task:");  
 manager.delete(1);  
  
 System.*out*.println("\nTasks After Deletion:");  
 manager.traverse();  
  
 }  
}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Q)** **Analyze the time complexity of each operation.**

1.Add Task: The time complexity is O(n) because to add a task at the end, the program needs to traverse the entire list to find the last node.

2.Search Task: The time complexity is O(n) since it performs a linear search by traversing through each node until it finds the matching task ID or reaches the end.

3.Traverse: The time complexity is O(n) because it visits each node in the linked list one by one to display or process the tasks.

4.Delete Task: The time complexity is also O(n) because it must traverse the list to locate the node to be deleted, and then adjust pointers accordingly.

**Q)Discuss the advantages of linked lists over arrays for dynamic data.**

1.Size: Arrays have a fixed size, meaning the number of elements must be known in advance or resizing must be handled manually. Linked lists, on the other hand, are dynamic in size and can grow or shrink as needed.

2.Insertion and Deletion: In arrays, insertion and deletion operations can be expensive (with a time complexity of O(n)) because elements may need to be shifted. In linked lists, these operations are more efficient—especially at the head—where they can be done in constant time (O(1)).

3.Memory Usage: Arrays require memory to be pre-allocated, which may lead to unused space if the array isn't full. Linked lists allocate memory as needed for each element, which makes them more flexible with memory usage.

4.Random Access: Arrays provide fast random access to elements using indices in constant time (O(1)). Linked lists do not support random access and require traversal from the head node, resulting in slower access with a time complexity of O(n).