**Exercise 5: Task Management System**

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

**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.

1. Create a class **Task** 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 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 String toString(){

return this.taskId + " " + this.taskName + " " + this.status;

}

}

1. Implement a singly linked list to manage tasks and Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.

import java.util.LinkedList;

public class TaskManager {

Node head;

Node tail;

Node temp;

Node prev;

TaskManager(Node head) {

this.head = head;

this.tail = head;

}

void addTask(Task task){

temp = new Node(task);

tail.next = temp;

tail = tail.next;

}

Task searchTask(int id){

temp = head;

while (temp != null){

if(temp.task.getTaskId() == id)

return temp.task;

temp = temp.next;

}

return null;

}

void deleteTask(int id){

temp = head;

prev = null;

while(temp != null){

if(temp.task.getTaskId() == id){

if(prev == null)

head = temp.next;

else

prev.next = temp.next;

}

prev = temp;

temp = temp.next;

}

}

void traverse(){

System.out.printf("%-20s %-20s %s\n","Task Id","Task Name","Status");

temp = head;

while(temp != null){

Task currTask = temp.task;

System.out.printf("%-20s %-20s %s\n",currTask.getTaskId(),currTask.getTaskName(),currTask.getStatus());

temp = temp.next;

}

}

}

1. Analysis

**Time Complexity:**

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