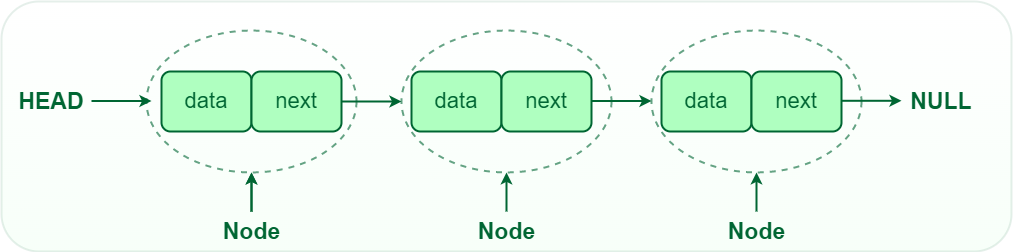
**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

1. Understand Linked Lists:

* Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
* **Singly Linked List:** **In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.**



* **Doubly Linked List:** In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.



2. Setup:

* + **Class Creation**: Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.

class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

@Override

public String toString() {

return "Task{" +

"taskId=" + taskId +

", taskName='" + taskName + '\'' +

", status='" + status + '\'' +

'}';

}}

* **class Task:** Defines a class named Task. class Task

**Instance variables (or attributes) of the Book class:**

* **int taskId:** An integer attribute to store the unique identifier for each task.
* **String taskName:** A string attribute to store the name of the task.
* **String status:** A string attribute to store the status of the task (e.g., "Pending", "Completed").

**Constructor:**

* **public Task(int taskId, String taskName, String status):** A constructor that initializes a Task object with the provided taskId, taskName, and status.
* **this.taskId = taskId**: Assigns the value of the parameter taskId to the instance variable taskId.
* **this.taskName = taskName:** Assigns the value of the parameter taskName to the instance variable taskName.
* **this.status = status:** Assigns the value of the parameter status to the instance variable status.

3. Implementation:

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

  class TaskLinkedList {

    private TaskNode head;

    public TaskLinkedList() {

        this.head = null;

    }

    public void addTask(Task task) {

        TaskNode newNode = new TaskNode(task);

        if (head == null) {

            head = newNode;

        } else {

            TaskNode current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newNode;

        }

    }

    public Task searchTask(int taskId) {

        TaskNode current = head;

        while (current != null) {

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

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    public void traverseTasks() {

        TaskNode current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    public boolean deleteTask(int taskId) {

        if (head == null) {

            return false;

        }

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

            head = head.next;

            return true;

        }

        TaskNode current = head;

        while (current.next != null) {

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

                current.next = current.next.next;

                return true;

            }

            current = current.next;

        }

        return false;

    }

}

Here is the github repo link –

4. Analysis:

* Analyze the time complexity of each operation.
* **add:**

1. **Creating a New Node:**
   * **TaskNode newNode = new TaskNode(task);**
   * **This operation creates a new TaskNode with the given Task. The time complexity for this operation is O(1), as it involves simple assignment and initialization.**
2. **Checking if the List is Empty:**
   * **if (head == null)**
   * **This operation checks if the list is empty (i.e., if head is null). The time complexity is O(1).**
3. **Adding the Node to the End of the List:**
   * **If the list is not empty, the method traverses the list to find the last node:**
     + **TaskNode current = head;**
       - **Sets the current node to the head of the list. Time complexity is O(1).**
     + **while (current.next != null)**
       - **This loop iterates through the linked list to find the last node. The time complexity of this loop is O(n), where n is the number of nodes in the list.**
     + **current = current.next;**
       - **Moves to the next node in each iteration. This operation is performed in the loop, and its time complexity is O(n).**
4. **Adding the New Node to the End:**
   * **current.next = newNode;**
   * **This operation sets the next reference of the last node to the new node. The time complexity for this operation is O(1).**

**Overall Time Complexity**

* **Best Case: When the list is empty (head is null), the time complexity is O(1) because it directly sets the head to the new node.**
* **Average Case: When the list has n nodes, the method must traverse all nodes to find the end. Therefore, the time complexity is O(n).**
* **Worst Case: Similar to the average case, where the method needs to traverse the entire list to find the last node. The time complexity is O(n).**
* Search:

1. **Initialization**:
   * **TaskNode current = head;**
   * This operation initializes a reference to the head of the list. The time complexity is O(1).
2. **Traversing the List**:
   * **while (current != null)**
     + The while loop iterates through the linked list from the head to the end.
     + Each iteration involves checking if the taskId of the current node matches the desired taskId and moving to the next node.
     + **if (current.task.taskId == taskId)**
       - This operation involves a comparison, which is O(1) per iteration.
     + **current = current.next;**
       - Moving to the next node is O(1) per iteration.
3. **Termination**:
   * The loop terminates when either the task is found or the end of the list (current becomes null) is reached.

**Overall Time Complexity**

* **Best Case**: The task is found in the first node (head). The time complexity is O(1) because only one comparison is needed.
* **Average Case**: The task is found somewhere in the middle of the list. On average, the method will need to check half of the nodes. Thus, the time complexity is O(n), where n is the number of nodes in the list.
* **Worst Case**: The task is not found in the list, or it is found at the last node. The method will need to traverse all n nodes. Thus, the time complexity is O(n).
* Traverse:

1. **Initialization**:
   * **TaskNode current = head;**
     + This operation initializes a reference to the head of the list. The time complexity is O(1).
2. **Traversing the List**:
   * **while (current != null)**
     + The while loop iterates through the linked list from the head to the end.
     + **System.out.println(current.task);**
     + This operation prints the task associated with the current node. The time complexity for printing is O(1) for each node, though in practice the time taken to print can vary depending on the output.
     + **current = current.next;**
     + This operation moves to the next node in the list and is O(1) for each node.
3. **Termination**:
   * The loop terminates when current becomes null, indicating the end of the list.

**Overall Time Complexity**

* **Best Case**: The best case occurs when the list has only one node. The loop will run once, and the time complexity is O(1) in terms of traversing and printing that single node.
* **Average Case**: The average case assumes the list contains n nodes. The loop will run n times, with each iteration having O(1) operations. Thus, the time complexity is O(n).
* **Worst Case**: The worst case is the same as the average case, where the list contains n nodes, and the loop needs to traverse all n nodes. The time complexity is O(n).
* Delete:

1. **Checking if the List is Empty**:
   * **if (head == null)**
     + This operation checks if the list is empty (i.e., if head is null). The time complexity is O(1).
2. **Deleting the Head Node**:
   * **if (head.task.taskId == taskId)**
     + If the head node contains the task to be deleted, it is removed by updating the head pointer. This operation has O(1) time complexity, as it only involves updating a reference.
3. **Deleting a Node in the Middle or End**:
   * **TaskNode current = head;**
     + Initializes the current pointer to the head of the list. Time complexity is O(1).
   * **while (current.next != null)**
     + The while loop iterates through the list to find the node just before the one to be deleted.
     + **if (current.next.task.taskId == taskId)**
       - Checks if the next node contains the task to be deleted. Time complexity is O(1) per iteration.
     + **current.next = current.next.next;**
       - Updates the next reference to bypass the node to be deleted. Time complexity is O(1) for this operation.
     + **current = current.next;**
       - Moves to the next node in each iteration. Time complexity is O(1) per iteration.
4. **Termination**:
   * The loop terminates when the task is found and deleted, or when the end of the list is reached (current.next becomes null).

**Overall Time Complexity**

* **Best Case**: The task to be deleted is at the head of the list. The time complexity is O(1) because it involves a single check and update of the head pointer.
* **Average Case**: The task is somewhere in the middle of the list. On average, the method will need to traverse half of the list. Thus, the time complexity is O(n), where n is the number of nodes in the list.
* **Worst Case**: The task is not in the list or is at the end of the list. The method needs to traverse the entire list. Thus, the time complexity is O(n).
* Discuss the advantages of linked lists over arrays for dynamic data.

 **Dynamic Size**:

* **Linked Lists**: Can grow or shrink in size dynamically. You can easily add or remove elements without needing to know the size in advance.
* **Arrays**: Have a fixed size once allocated. To handle dynamic data, you would need to create a new array with a larger size and copy elements from the old array, which can be inefficient.

 **Efficient Insertions and Deletions**:

* **Linked Lists**: Insertion and deletion operations are efficient. You only need to update a few pointers to add or remove elements. These operations can be performed in O(1) time if the position is known.
* **Arrays**: Insertions and deletions require shifting elements to maintain order, which can be costly and result in O(n) time complexity for these operations, especially in the middle of the array.

 **Memory Utilization**:

* **Linked Lists**: Allocate memory for each element dynamically. There is no need to allocate a large block of memory in advance. This can be more memory-efficient, especially if the size of the data is not known beforehand.
* **Arrays**: Require allocating a fixed amount of memory upfront. This can lead to wasted space if the array is over-provisioned or insufficient space if the array is under-provisioned.

 **Flexibility in Memory Allocation**:

* **Linked Lists**: Memory for elements can be allocated and deallocated as needed, which helps in managing memory more flexibly and can reduce fragmentation.
* **Arrays**: Require contiguous memory allocation. If the array grows too large, finding a contiguous block of memory can become problematic.

 **Efficient Use of Space**:

* **Linked Lists**: Do not require reallocation of memory. Each element is allocated independently, so space is utilized as needed.
* **Arrays**: Might need to be resized and copied when elements are added beyond the initial capacity. This can lead to performance overhead and wasted space.

 **Implementation of Other Data Structures**:

* **Linked Lists**: Serve as the foundation for implementing various advanced data structures such as stacks, queues, and deques. They provide the flexibility needed to efficiently support these structures.
* **Arrays**: While arrays are a good base for simple data structures, they are less flexible for implementing structures that require frequent insertions and deletions.

Output:

