**Types of Linked Lists**

**1. Singly Linked List**

**Description:**

* Each node contains data and a reference (or link) to the next node in the sequence.
* **Structure:** Node → Node → Node → null

**Advantages:**

* Simple implementation.
* Efficient insertion and deletion at the beginning.

**Disadvantages:**

* Only allows traversal in one direction (forward).
* Deletion and insertion in the middle or end are less efficient compared to doubly linked lists.

**Use Case:**

* Suitable for applications where only forward traversal is needed and simplicity is key.

**2. Doubly Linked List**

**Description:**

* Each node contains data and two references: one to the next node and one to the previous node.
* **Structure:** null ← Node ↔ Node ↔ Node → null

**Advantages:**

* Allows traversal in both directions (forward and backward).
* Easier to delete nodes from the middle since it has a reference to the previous node.

**Disadvantages:**

* More complex to implement compared to singly linked lists.
* Requires additional memory for the extra reference to the previous node.

**Use Case:**

* Ideal for applications requiring bidirectional traversal, such as navigation in a web browser or managing a playlist.

**Summary**

* **Singly Linked List:** Each node points to the next node, simple and efficient for forward traversal.
* **Doubly Linked List:** Each node points to both the next and previous nodes, allowing bidirectional traversal but with increased complexity and memory usage.

**Analysis:**

**Time Complexity Analysis**

Here is the time complexity analysis of each operation in my code:

**1. Add Task (addTask method)**

* **Operation:** Adding a new task to the end of the linked list.
* **Time Complexity:** O(n)
  + **Explanation:** To add a task, the method traverses the entire list to find the last node. In the worst case, it must visit all n nodes.

**2. Search Task (searchTask method)**

* **Operation:** Searching for a task by taskId.
* **Time Complexity:** O(n)
  + **Explanation:** The method traverses the linked list from the head to the end, checking each node's taskId. In the worst case, it may need to examine all n nodes.

**3. Traverse Tasks (traverseTasks method)**

* **Operation:** Printing all tasks in the linked list.
* **Time Complexity:** O(n)
  + **Explanation:** The method iterates through all nodes once, printing each node's details. It requires visiting every node in the list.

**4. Delete Task (deleteTask method)**

* **Operation:** Deleting a task by taskId.
* **Time Complexity:** O(n)
  + **Explanation:** The method first checks if the head node needs to be deleted (O(1)). If not, it traverses the list to find the node to delete, which could involve checking all n nodes in the worst case.

**Advantages of Linked Lists Over Arrays for Dynamic Data:**

1. **Dynamic Size:**
   * **Linked Lists:** Can easily grow and shrink in size as needed by adding or removing nodes dynamically.
   * **Arrays:** Have a fixed size, requiring resizing and copying of elements if the size needs to change.
2. **Efficient Insertions and Deletions:**
   * **Linked Lists:** Insertion and deletion operations can be done efficiently without shifting elements. Nodes can be added or removed from any position in O(1) time if the position is known.
   * **Arrays:** Inserting or deleting elements requires shifting other elements, resulting in O(n) time complexity for these operations.
3. **Memory Utilization:**
   * **Linked Lists:** Allocate memory as needed for each node, which can be more efficient in terms of memory usage when the size is unpredictable or varies significantly.
   * **Arrays:** May waste memory if allocated size is larger than the actual number of elements used.
4. **Flexibility:**
   * **Linked Lists:** Can handle dynamic changes in the dataset without the need for reallocation or resizing.
   * **Arrays:** Require resizing or reallocation if the dataset size exceeds the initial allocation.
5. **Ease of Implementation for Certain Data Structures:**
   * **Linked Lists:** Serve as a foundation for implementing more complex data structures like stacks, queues, and dequeues.
   * **Arrays:** While versatile, more complex data structures require additional management or hybrid approaches.