**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Different Types of Linked Lists**

**1. Singly Linked List:**

* **Structure:** Consists of nodes where each node contains a data part and a reference (or link) to the next node in the sequence.
* **Traversal:** Can be traversed in only one direction, from the head node to the last node.
* **Operations:**
  + Insertion at the beginning or end is straightforward.
  + Deletion requires traversal to find the node to delete.
* **Advantages:**
  + Simple and easy to implement.
  + Efficient for insertion and deletion operations at the beginning of the list.

**2. Doubly Linked List:**

* **Structure:** Each node contains two references: one to the next node and one to the previous node.
* **Traversal:** Can be traversed in both directions, from head to tail and tail to head.
* **Operations:**
  + Insertion and deletion are more flexible because we can traverse in both directions.
  + Requires more memory due to the additional reference for the previous node.
* **Advantages:**
  + Easier to implement bidirectional traversal.
  + More efficient for certain operations like deletion of a given node when the pointer to the node is provided.

**3. Circular Linked List:**

* **Structure:** Similar to singly linked lists, but the last node points back to the first node, forming a circle.
* **Traversal:** Can start from any node and traverse the entire list in a circular fashion.
* **Operations:**
  + Useful for implementing circular buffers or structures like round-robin scheduling.
* **Advantages:**
  + No need to keep a head or tail pointer separately; the list can be traversed indefinitely.

**4. Circular Doubly Linked List:**

* **Structure:** Similar to doubly linked lists, but the last node points back to the first node and the first node points to the last node, forming a circle.
* **Traversal:** Can be traversed in both directions in a circular manner.
* **Operations:**
  + Combines the benefits of doubly linked lists and circular linked lists.
* **Advantages:**
  + Efficient bidirectional circular traversal.

**Advantages of Linked Lists over Arrays for Dynamic Data**

1. **Dynamic Size:**
   * **Linked Lists:** Can grow and shrink in size dynamically as elements are added or removed. There is no need to specify the initial size.
   * **Arrays:** Have a fixed size once declared. If more elements are needed than the initial size, a new array must be created and the old elements copied over.
2. **Efficient Insertions and Deletions:**
   * **Linked Lists:** Inserting or deleting elements, especially at the beginning or middle, is efficient because it involves changing only a few pointers. There is no need to shift elements.
   * **Arrays:** Insertion or deletion in the middle requires shifting elements, which can be time-consuming, especially for large arrays.
3. **Memory Utilization:**
   * **Linked Lists:** Allocate memory as needed for each new element, which can be more memory-efficient for applications where the number of elements changes frequently.
   * **Arrays:** Require a contiguous block of memory, which can lead to memory wastage if the array is not fully utilized.
4. **Flexibility:**
   * **Linked Lists:** Provide greater flexibility for implementing complex data structures like stacks, queues, and graphs due to their dynamic nature.
   * **Arrays:** While simple and fast for fixed-size collections, they lack the flexibility needed for more complex data structures without significant overhead.

**Analysis**

1. **Add Task:**
   * Time Complexity: O(n) (since we traverse the list to add at the end)
2. **Search Task:**
   * Time Complexity: O(n) (since we may need to search through the entire list)
3. **Traverse Tasks:**
   * Time Complexity: O(n) (since we traverse the entire list)
4. **Delete Task:**
   * Time Complexity: O(n) (since we may need to traverse the list to find the node to delete)