**TASK MANAGEMENT SYSTEM**

**Types of Linked Lists:**

**Singly Linked List**

A Singly Linked List is a data structure where each element, called a node, contains a value and a reference (or link) to the next node in the sequence. It starts with a head node and ends with a tail node, which has a reference pointing to `null`. This unidirectional linkage allows for efficient sequential access and insertion operations at the beginning of the list but makes operations like deletion and insertion at arbitrary positions less efficient, as it requires traversal from the head node. Singly Linked Lists are memory efficient because each node only requires storage for one reference. However, they are less efficient for operations requiring reverse traversal, as there is no backward link.

**Doubly Linked List**

A Doubly Linked List enhances the Singly Linked List by adding an additional reference in each node, pointing to the previous node. This bidirectional linking allows for more flexible navigation through the list, making both forward and backward traversal possible. Each node thus contains a value, a reference to the next node, and a reference to the previous node. This structure facilitates more efficient insertion and deletion operations at both ends and in the middle of the list, as it can directly access both neighboring nodes. However, the trade-off is increased memory usage due to the extra reference in each node. Doubly Linked Lists are useful when frequent bidirectional traversal is required.

**Circular Linked List:**

A Circular Linked List is similar to a Singly Linked List, but with a crucial difference: the last node in the list points back to the first node, forming a circular structure. This means there is no end node with a null reference. Circular Linked Lists can be either singly or doubly linked. For a singly circular linked list, the last node's `next` reference points to the head, while in a doubly circular linked list, the last node's `next` points to the head and the head’s `prev` points to the last node. Circular Linked Lists are useful for applications that require a cyclic iteration over the elements, such as round-robin scheduling.

**Doubly Circular Linked List:**

A Doubly Circular Linked List is an extension of the Doubly Linked List, where both the head node and the tail node are linked in a circular manner. This means the `next` reference of the tail node points to the head node, and the `prev` reference of the head node points to the tail node. This structure supports efficient traversal in both directions and facilitates operations such as insertion and deletion at both ends. It is particularly useful in applications requiring constant-time access to both ends of the list and cyclic traversals, such as in certain buffering techniques.

**Time Complexity Analysis**

**1. Add Operation (`addTask`):**

- Time Complexity: O(1)

- Explanation: Adding a task to the end of the linked list involves updating the `tail` pointer and linking the new task. Since this operation only involves a few pointer assignments and does not require traversing the list, it has a constant time complexity.

**2. Search Operation (`searchTask`):**

- Time Complexity: O(n)

- Explanation: To find a task by its ID, the method traverses the linked list from the head node to the end, checking each node. In the worst case, it may need to traverse the entire list if the task is at the end or not present, leading to linear time complexity.

**3. Display Operation (`displayTask`):**

- Time Complexity: O(n)

- Explanation: Displaying all tasks requires traversing the entire linked list, printing each task's details. The time complexity is linear with respect to the number of tasks in the list.

**4. Delete Operation (`deleteTask`):**

- Time Complexity: O(n)

- Explanation: Deleting a task involves searching for the task first, which requires traversing the list. Once found, the method updates the pointers to remove the task. The search step dominates the time complexity, resulting in linear time complexity.

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

1. Dynamic Size: Linked lists can easily grow or shrink in size without the need to allocate or reallocate memory. This flexibility makes them well-suited for scenarios where the number of elements changes frequently.

2. Efficient Insertions and Deletions: Inserting or deleting elements in a linked list is generally more efficient than in an array, especially for operations at the beginning or middle of the list. Unlike arrays, which require shifting elements, linked lists only involve updating a few pointers, resulting in O(1) time complexity for these operations when the position is known.

3. No Wasted Space: Linked lists use only as much memory as needed for the current number of elements, avoiding the issue of wasted space that can occur with arrays when the allocated size exceeds the number of used elements.

4. Flexibility: Linked lists are inherently flexible due to their dynamic nature. They do not require knowing the maximum size in advance, and their nodes can be scattered throughout memory, reducing fragmentation concerns compared to arrays.