### Analysis of Task Management System Using Singly Linked List

1. Time Complexity Analysis:

- Add Task:

- Best Case: O(1) - Adding the first task when the list is empty.

- Worst Case: O(n) - Adding a task requires traversing the entire list to find the end.

- Search Task:

- Best Case: O(1) - The task to be found is the first task.

- Worst Case: O(n) - The task to be found is the last task, or it does not exist.

- Traverse Tasks:

- Best, Average, and Worst Case: O(n) - All tasks need to be visited and printed.

- Delete Task:

- Best Case: O(1) - Deleting the first task.

- Worst Case: O(n) - Deleting a task requires traversing the list to find the task, which could be at the end or not present at all.

2. Advantages of Linked Lists over Arrays for Dynamic Data:

- Dynamic Size: Linked lists can grow and shrink dynamically, whereas arrays have a fixed size.

- Efficient Insertions/Deletions: Insertions and deletions at the beginning or middle of the list are more efficient in linked lists (O(1)

for beginning) compared to arrays (O(n) for shifting elements).

- Memory Allocation: Linked lists do not require contiguous memory allocation, which is beneficial when dealing with fragmented memory.

3. Discussion:

- Memory Overhead: Linked lists have additional memory overhead for storing pointers/reference to the next nodes, which is not present in arrays.

- Access Time: Accessing an element by index is O(n) in a linked list, compared to O(1) in an array. This makes linked lists less suitable for scenarios requiring frequent random access.

- Use Cases: Linked lists are ideal for applications with dynamic and frequently changing datasets where insertions and deletions are common, and direct access by index is not required.

Example Analysis Using Code:

- Adding Tasks:

- The `addTask` method traverses the list to find the end and adds the new task there. This is O(n) in the worst case.

- Searching for a Task:

- The `searchTask` method traverses the list to find the task with the given `taskId`. This is O(n) in the worst case.

- Traversing Tasks:

- The `traverseTasks` method traverses the entire list to print each task, which is O(n).

- Deleting a Task:

- The `deleteTask` method traverses the list to find the task with the given `taskId`. Deleting the task involves updating the `next` pointer of the previous node, which is O(n) in the worst case.

Conclusion:

- Linked lists offer flexibility and efficiency for dynamic data management, particularly when frequent insertions and deletions are required.

- However, they come with a trade-off in terms of memory overhead and access time for indexed elements.

- Choosing between arrays and linked lists depends on the specific requirements of the application, such as the frequency of insertions/deletions versus the need for fast access by index.