Analysis

Time Complexity Analysis

1. **Add Task**
    - Time Complexity: O(1) if adding at the beginning
    - Time Complexity: O(n) if adding at the end (without a tail pointer)
    - Explanation: In a singly linked list, inserting at the head is very efficient. Inserting at the end requires traversal unless a reference to the tail is maintained.
2. **Search Task**
    - Time Complexity: O(n)
    - Explanation: Each node may need to be checked sequentially until the matching task is found.
3. **Traverse Tasks**
    - Time Complexity: O(n)
    - Explanation: Each node is visited once during traversal.
4. **Delete Task**
    - Time Complexity: O(n)
    - Explanation: Searching for the node and adjusting links requires linear time, especially when the node is in the middle or end.

Advantages of Linked Lists Over Arrays

- **Dynamic Size:**
  Linked lists can grow or shrink at runtime without needing to resize or allocate memory in advance, unlike arrays which have fixed sizes or require costly resizing.
- **Efficient Insertions/Deletions:**
  Inserting or deleting elements (especially at the beginning) is faster in linked lists, as there is no need to shift elements like in arrays.
- **Memory Utilization:**
  Memory allocation happens per node, reducing waste compared to over-allocated arrays.

When to Use Linked Lists

- When the number of elements is unpredictable or frequently changes.
- When frequent insertions and deletions are required, particularly at the head or tail.
- When minimal memory reallocations are preferred.

Limitations

- No direct access by index (unlike arrays).
- Extra memory is required for pointers in each node.
- Linear search time for most operations unless additional data structures are used.