**5 Task Management System**

**Linked Lists in Task Management Systems**

Linked lists are foundational data structures widely used in dynamic applications such as task management systems. Unlike arrays, linked lists consist of nodes where each node contains data and a reference to the next node. This allows efficient insertion and deletion of tasks without the need for shifting elements or resizing memory.

**1. Singly Linked List**

**Definition:**  
A singly linked list is a unidirectional list where each node connects to the next via a single reference called next. The final node points to null, signifying the end of the list.

**Structure of Node:**

* taskId – Unique identifier for each task
* taskName – Description or title of the task
* status – Completion state (e.g., "Pending", "Done")
* next – Reference to the next task node

**Operations:**

* **Add** – Insert a task at the end of the list by traversing from the head to the last node.
* **Search** – Locate a task by traversing and comparing taskId.
* **Delete** – Remove a task by updating the next pointer of the preceding node.
* **Traverse** – Print or process each task from start to end.

**2. Doubly Linked List**

**Definition:**  
A doubly linked list is a bi-directional structure where each node contains references to both the next and previous nodes, enabling traversal in both directions.

**Structure of Node:**

* taskId, taskName, status
* next – Link to the next node
* prev – Link to the previous node

**Advantages over Singly Linked List:**

* Supports backward traversal
* Easier deletion when a reference to a node is known
* Facilitates operations at both ends efficiently

**Comparison**

| **Feature** | **Singly Linked List** | **Doubly Linked List** |
| --- | --- | --- |
| Direction | Forward only | Forward and backward |
| Memory usage | Less (1 pointer) | More (2 pointers) |
| Complexity | Easier to implement | Requires managing two links |
| Insertion/Deletion | Simple at head | Flexible anywhere |

**Time Complexity Analysis**

**1. Adding a Task**

* **Operation:** Add at the end
* **Time Complexity:** O(n)
* **Reason:** Requires traversal from the head to the last node

**2. Searching a Task**

* **Operation:** Locate by taskId
* **Time Complexity:** O(n)
* **Reason:** Each node is checked until match is found

**3. Traversing the List**

* **Operation:** Display or process all tasks
* **Time Complexity:** O(n)
* **Reason:** Every node must be visited once

**4. Deleting a Task**

* **Operation:** Remove task by taskId
* **Time Complexity:** O(n)
* **Reason:** Finding the task requires traversal; deletion is O(1) once found

**Advantages of Linked Lists Over Arrays**

1. **Dynamic Growth**
   * Linked lists grow as needed without reallocating memory, unlike arrays which have a fixed size or require manual resizing.
2. **Efficient Modifications**
   * Inserting or deleting tasks does not require shifting elements as in arrays. Only references are updated.
3. **No Fixed Memory Block Required**
   * Nodes can be scattered in memory and don’t need contiguous blocks like arrays do.
4. **Optimized Insertions/Deletions**
   * Especially efficient at adding/removing from the beginning or end, without expensive memory operations.

**Disadvantages of Linked Lists**

* **Extra Memory:**  
  Each node stores an additional pointer, leading to more memory usage compared to arrays.
* **Slower Access:**  
  Cannot directly access elements by index like arrays; must traverse sequentially.
* **Poor Cache Performance:**  
  Due to scattered memory allocation, they may lead to frequent cache misses compared to arrays.