**Exercise 5: Task Management System**

**Explain the different types of linked lists (Singly Linked List, Doubly Linked List).**

Linked lists are essential structures in computer science used to organize data in a linear sequence. Each element, or node, stores a value and a reference to the next node in the chain.

**1. Singly Linked List**

**Definition:** A singly linked list links each node to the next one, with the final node pointing to null.

**Structure:**

* **Node contains:**
  + Data: The stored value.
  + Next: A pointer to the next node.

**Key Features:**

* One-way traversal from the first node (head) to the last.
* Simple structure, easy to build.
* Fast insertion or deletion at the beginning, but inserting or deleting elsewhere requires traversing the list.

**Operations:**

* **Traversal:** Start at the head, follow next references.
* **Insertion:** Modify pointers to insert at start, end, or a specific spot.
* **Deletion:** Update the next pointer of the previous node to skip over the one being removed.

**2. Doubly Linked List**

A doubly linked list gives each node two references—one to the next node and one to the previous—allowing movement in both directions.

**Structure:**

* **Node contains:**
  + Data
  + Next: Points forward.
  + Previous: Points backward.

**Key Features:**

* **Two-way navigation** is possible.
* Slightly more memory overhead due to the extra pointer.
* Inserting or removing nodes is **more flexible**, especially in the middle of the list.

**Operations:**

* **Traversal:** You can move forward using next or backward using previous.
* **Insertion/Deletion:** Easier and more efficient when pointers are adjusted from both directions.

| **Feature** | **Singly Linked List** | **Doubly Linked List** |
| --- | --- | --- |
| **Direction** | Forward only | Both forward and backward |
| **Memory Use** | Lower (1 pointer per node) | Higher (2 pointers per node) |
| **Complexity** | Easier to implement | More complex but more versatile |
| **Efficiency** | Fast at head operations | Better at both ends and the middle |

**Analyze the time complexity of each operation.**

* 1. **Adding a Task - O(n)** To append a task, we must first scan the entire list to locate the final node, which requires linear time relative to the number of tasks.
  2. **Searching for a Task - O(n)** In the worst case, the search must scan every node to find a match, resulting in linear time.
  3. **Traversing the List - O(n)** We must visit each node once to access every task, making the traversal linear in time.
  4. **Deleting a Task** - **O(n)** Although deleting a node is an O(1) action, locating the node first takes O(n), which determines the overall complexity.

**Discuss the advantages of linked lists over arrays for dynamic data.**

**1. Flexible Sizing**

* **Linked Lists:** They can dynamically grow or shrink, allowing elements to be added or removed without resizing the entire structure.
* **Arrays:** Their size is fixed once created. Expanding them requires allocating a larger array and copying the elements, which is resource-intensive.

**2. Efficient Insertions and Deletions**

* **Linked Lists:** Inserting or deleting a node is efficient—especially if you already know the preceding node—since the operation only involves updating pointers, often in **O(1)** time.
* **Arrays:** These operations are slower due to the need to shift elements to preserve order, costing **O(n)** in time.

**3. Memory Management**

* **Linked Lists:** Nodes are created only when needed, using memory efficiently without reserving unused space.
* **Arrays:** Allocate a continuous block up front, which can lead to wasted space if underutilized, or resizing if too small.

**4. No Resizing Penalty**

* **Linked Lists:** There's no need for copying or reallocating memory as the structure expands.
* **Arrays:** Growing an array involves creating a new one and moving existing elements, which adds overhead.

**5. Simplicity in Building Other Structures**

* **Linked Lists:** Ideal for implementing advanced structures like stacks, queues, and deques with minimal extra logic.
* **Arrays:** Can also be used but often require additional tracking and code for dynamic behaviour.

However, even linked lists have a few disadvantages which are –

1. **Extra Memory Usage:** Each node holds not just data but also a reference to the next node, increasing memory consumption.
2. **Weaker Cache Performance:** Arrays benefit from being stored in contiguous memory blocks, making them faster for operations like looping, thanks to better cache utilization.