**Exercise 5: Task Management System**

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

**1. Singly Linked List**

**Definition:**

A **singly linked list** is a linear data structure in which each element (called a node) contains two parts:

* **Data**: The value or information the node holds.
* **Next Pointer**: A reference to the **next node** in the list.

The last node’s next pointer is null, indicating the end of the list.

**Structure:**

[Data|Next] → [Data|Next] → [Data|null]

**Characteristics:**

* **Unidirectional**: Traversal is possible only in one direction (from head to tail).
* **Dynamic size**: Grows or shrinks in size as needed (no fixed capacity).
* **Efficient insertion/deletion at the beginning**: These operations are O(1) if performed at the head.

**Advantages:**

* Uses **less memory** than a doubly linked list (only one pointer per node).
* Easier to implement.

**Disadvantages:**

* **No backward traversal** possible.
* Deleting a node requires **linear traversal** to find the previous node (if not at the head).
* Less efficient for certain operations like reverse traversal or bidirectional navigation.

**2. Doubly Linked List**

**Definition:**

A **doubly linked list** is a linear data structure in which each node contains three parts:

* **Data**: The value the node holds.
* **Previous Pointer**: A reference to the **previous node**.
* **Next Pointer**: A reference to the **next node**.

The first node’s previous pointer is null, and the last node’s next pointer is null.

**Structure:**

null ← [Prev|Data|Next] ↔ [Prev|Data|Next] ↔ [Prev|Data|null]

**Characteristics:**

* **Bidirectional**: Allows traversal in both directions (forward and backward).
* More flexible when dealing with complex insertion/deletion logic (e.g., inserting before or after a specific node).

**Advantages:**

* **Efficient deletion** of a given node if its reference is available (no need to find the previous node).
* **Reverse traversal** is possible and efficient.
* Suitable for complex data structures like **deques**, **navigation systems**, etc.

**Disadvantages:**

* **More memory usage** due to the extra pointer (for previous).
* Slightly more complex to implement (handling both pointers).
* Insertions and deletions are more prone to bugs due to needing to update two pointers for each neighboring node.

**Comparison Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Singly Linked List** | **Doubly Linked List** |
| Direction | One-way | Two-way |
| Memory Usage | Less (1 pointer per node) | More (2 pointers per node) |
| Traversal | Forward only | Forward and backward |
| Deletion (known node) | O(n) | O(1) |
| Insertion (middle) | Requires previous node | Easier with both directions |
| Reverse traversal | Not possible | Efficient and supported |
| Implementation | Simpler | More complex |

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

Assume n is the number of nodes in the linked list.

**a. Add Task (At the End)**

* **Time Complexity:** O(n)
* **Why:** To add a new node at the end, we must traverse the entire list to find the last node. If we maintained a tail pointer, this could be reduced to O(1).

**b. Search Task (By ID)**

* **Time Complexity:** O(n)
* **Why:** Each node must be checked one by one until the task is found or the end is reached. Linear search is required since there is no indexing.

**c. Traverse Tasks**

* **Time Complexity:** O(n)
* **Why:** Every node in the list must be visited and displayed. This is a linear-time operation.

**d. Delete Task (By ID)**

* **Time Complexity:** O(n)
* **Why:** The list must be traversed to find the node to delete, and also to find the previous node (for pointer adjustment). Even in the best case (first node), it still involves comparison and pointer reassignment.

**Summary Table**

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| Add (at end) | O(n) |
| Search (by ID) | O(n) |
| Traverse | O(n) |
| Delete (by ID) | O(n) |

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

**a. Dynamic Size**

* **Linked List:** Grows or shrinks as needed during runtime without reallocation.
* **Array:** Fixed size or requires manual resizing, which is costly (e.g., copying to a new array).

**b. Efficient Insertions and Deletions (at the beginning or middle)**

* **Linked List:** Insertion and deletion at the head or after a known node is O(1). No need to shift elements.
* **Array:** Insertion or deletion requires shifting all subsequent elements, which is O(n) time.

**c. No Wasted Space**

* **Linked List:** Allocates memory as needed. Ideal when the number of elements is unpredictable.
* **Array:** May have unused or wasted space if the capacity is larger than the actual usage.

**d. Ease of Implementing Abstract Data Types**

* **Linked List:** Suited for implementing stacks, queues, and other data structures where dynamic resizing and pointer manipulation are beneficial.
* **Array:** More suited for situations where fast random access is required.

**Limitations of Linked Lists (Compared to Arrays)**

To provide a balanced view:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Linked List** | **Array** |
| Random Access | Not supported (O(n)) | Fast (O(1)) |
| Memory Overhead | Higher (extra pointers) | Lower (pure data) |
| Cache Friendliness | Poor (non-contiguous) | Good (contiguous) |
| Traversal | Slower | Faster |

**Conclusion**

Linked lists are ideal for scenarios where:

* The size of the data changes frequently.
* Insertions and deletions are frequent, especially at arbitrary or beginning positions.
* Memory usage needs to be flexible.

Arrays are better when:

* Frequent access to elements by index is required.
* The size of the collection is known in advance or changes infrequently.