**LINKED LIST**

# Introduction

## Problem with Arrays and Hash Table

**Arrays:**

- Static Arrays: Fixed size, limited memory allocation, and performance issues when resizing.

- Dynamic Arrays: Can grow in size but resizing (doubling) can be costly (O(n) time).

- Insertion/Deletion Issues: Shifting elements when inserting/deleting anywhere but the end of the array.

**Hash Tables:**

- Store data efficiently without worrying about memory allocation.

- Drawback: Do not maintain order of elements.

# Linked list

- A linked list is a data structure made up of nodes, where each node contains:

* A value (the data you want to store, e.g., a number).
* A pointer (a reference to the next node in the list).

- Singly Linked List: In a singly linked list, each node points to the next node, and the last node points to null, indicating the end of the list.

## Key Terms

- Head: The first node in the list.

- Tail: The last node in the list, which points to null.

- Pointer: A reference to the location of the next node in memory.

A diagram of a chart

Description automatically generated with medium confidence

Slide : Linked list

## How It Works

- Nodes link to the next node in the sequence, forming a chain of nodes.

- The list can be sorted or unsorted, and the nodes can store any type of data.

## Advantages

- Efficient Insertions/Deletions: Insert or delete nodes in the middle by adjusting pointers, no need to shift other elements like array.

- Dynamic Memory Allocation: Nodes are scattered in memory, allowing flexible growth/shrinkage without requiring contiguous memory.

- Maintains Order: Linked lists maintain an order of elements, unlike hash tables.

## Disadvantages

- Traversal for access: Accessing elements requires traversing the list from the head, taking O(n) time.

- Slower iteration: Iteration through nodes is slower than arrays due to non-contiguous memory locations.

- No random access: Cannot directly access elements by index like arrays can, requiring traversal to reach specific nodes.

## Operations

- Insert/Prepend: O(1) (at the beginning).

- Append (Add to End): O(1).

- Lookup/Search: O(n) (requires traversal).

- Insert/Delete: O(n) (requires finding the position).

- Delete: O(n) (requires finding the node).

# Pointer

- A pointer is simply a reference to another memory location, object, or node.

- In programming, a pointer allows you to refer to the memory address where an object or data is stored, rather than copying the data itself.

A black background with white text

Description automatically generated

- Example: In this case, *objectTwo* is a pointer to the same memory location as *objectOne*. Both *objectOne* and *objectTwo* point to the same object in memory.

- In languages like JavaScript, memory management (garbage collection) is automatic, which makes it easier to work with memory.

- In low-level languages, you must manually manage memory to avoid issues like memory leaks.

# Doubly Linked List

- A Doubly Linked List is a type of linked list where each node contains:

* A value (data).
* A next pointer (link to the next node).
* A previous pointer (link to the previous node).

## Key Features

- Bi-directional traversal: Can move forward (using next) and backward (using prev).

- Head: First node with prev = null.

- Tail: Last node with next = null.

A diagram of a graph

Description automatically generated with medium confidence

Slide : Doubly Linked list

## Advantages

- Backward Traversal: Can traverse the list in both directions.

- Improved Search:

* Can start search from both the head and the tail.
* Slightly faster when you know the general position of the data.

- Efficient Insertions/Deletions: Easier to add or remove nodes from both ends of the list.

## Disadvantages

- Extra Memory Usage: Each node requires more memory to store the prev pointer.

- More Complex Implementation: Need to manage both next and prev pointers when adding or removing nodes.

## Performance

- Traversal: O(n) (still linear time).

- Insertions/Deletions: O(1) at the head or tail.

- Search: Technically O(n) but can optimize by starting from head or tail.

# Compare Linked list vs. Doubly Linked list

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Singly Linked List** | **Doubly Linked List** |
| Memory Usage | Requires less memory (no prev pointer). | Requires more memory (additional prev pointer). |
| Implementation | Simpler to implement. | More complex due to prev pointer management. |
| Traversal | Only forward (cannot traverse backward). | Bi-directional: Can traverse forward and backward. |
| Insertion/Deletion | Faster (fewer operations, no need to update prev). | Slightly slower (must update both next and prev). |
| Use Case | - Limited memory or memory is expensive.  - Fast insertion/deletion, especially at the beginning | - Memory is not a major concern.  - Need to traverse or search elements in both directions. |
| Drawbacks | Cannot traverse backward; losing head loses the list. | Higher memory usage and complexity. |
| Common Use | More common in coding interviews. | Used when bi-directional operations are required. |