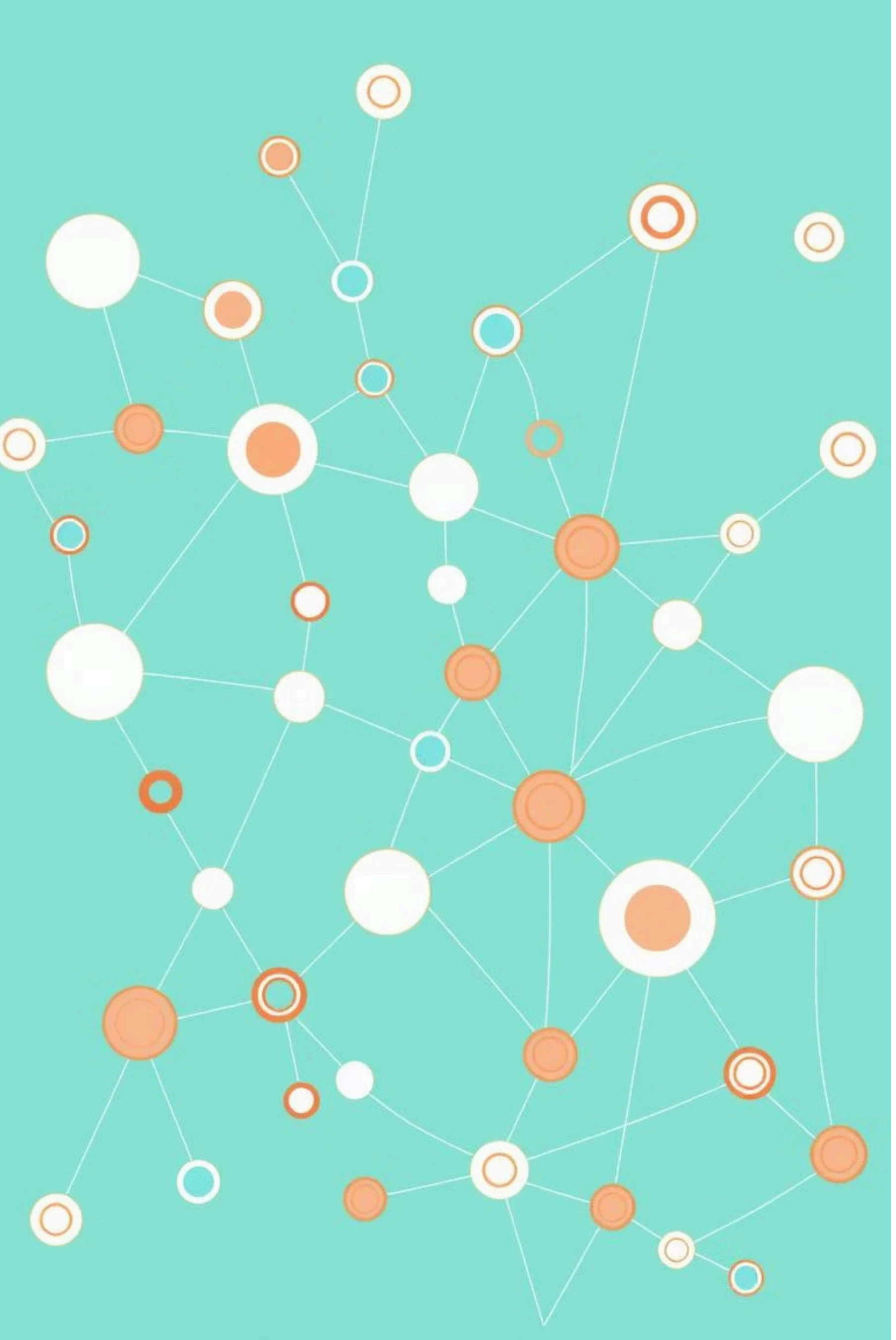


# Singly Linked List

Welcome to Session 2, where we delve into the world of Singly Linked Lists. This session will equip you with a fundamental understanding of this crucial data structure, its operations, and its practical applications.

Session Overview

# Learning Objectives

1. **Understand Singly Linked Lists**

   Grasp what they are and their utility.

2. **Define Nodes & Connections**

   Learn how individual nodes are structured and linked.

3. **Implement Core Operations**

   Master insertion, deletion, and traversal.

4. **Compare with Arrays**

   Analyze differences and use cases.

5. **Analyze Complexity**

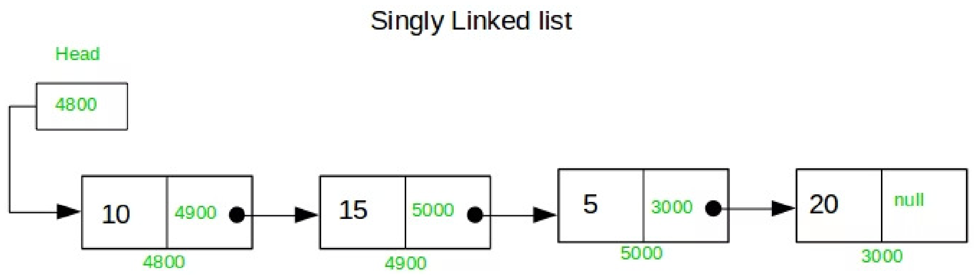
   Understand time and space efficiency.

# What is a Singly Linked List?

A Singly Linked List is a linear data structure where each element, called a node, contains two main parts:

- **Data:** The value you want to store.
- **Pointer (next):** The address of the next node in the list.

The list concludes when a node's 'next' pointer is **NULL**.



# Understanding the Node

A node is the fundamental building block of a linked list.

```
struct Node {
  int data; // Value of the node
  Node* next; // Pointer to the next node
};
```

**Example:** Creating a Node: `Node* newNode = new Node{10, nullptr};`

> Consider a linked list: **10 → 20 → 30 → NULL**
>
> Visually: **[10 | *] ---> [20 | *] ---> [30 | NULL]**

Each node holds data and a pointer to the subsequent node. The final node's 'next' pointer is **NULL**, signifying the end of the list.

# Core Operations: Traversal

**Traversal** involves printing all elements in the list.

```
void printList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
}
```

**Explanation:** This function begins at the head and iterates through each node using the 'next' pointer, printing its data until it reaches `nullptr` (the end of the list).

**Time Complexity:** O(n)

# Core Operations: Insertion

## Insertion at Beginning

```
void insertAtHead(Node*& head, int val) {
    Node* newNode = new Node{val, head};
    head = newNode;
}
```

**Explanation:** A new node is created with the given value and its 'next' pointing to the current head. The head is then updated to this new node. This is a fast O(1) operation.

**Time Complexity:** O(1)

## Insertion at End

```
void insertAtTail(Node*& head, int val) {
    Node* newNode = new Node{val, nullptr};
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next)
        temp = temp->next;
    temp->next = newNode;
}
```

**Explanation:** A new node is created and appended to the end. If the list is empty, it becomes the head. Otherwise, it traverses to the last node and links the new node.

**Time Complexity:** O(n)

# Core Operations: Deletion

## Deletion by Value

```
void deleteByValue(Node*& head, int val) {
    if (!head) return;
    if (head->data == val) {
        Node* del = head;
        head = head->next;
        delete del;
        return;
    }
    Node* temp = head;
    while (temp->next && temp->next->data != val)
        temp = temp->next;
    if (temp->next) {
        Node* del = temp->next;
        temp->next = temp->next->next;
        delete del;
    }
}
```

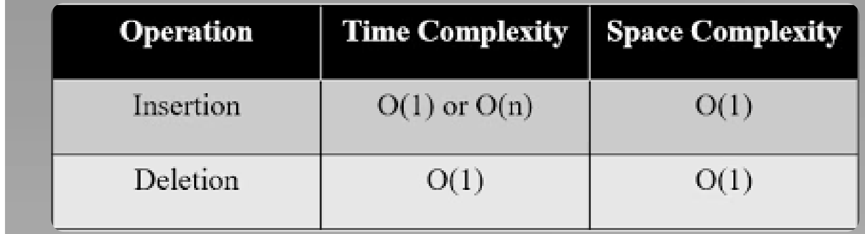
**Explanation:** This function searches for the first node with the specified value. If it's the head, the head pointer is adjusted. Otherwise, it finds the preceding node and re-links its 'next' pointer to bypass the deleted node.

ⓘ **Time Complexity:** O(n)

# Space Complexity

Each node in a singly linked list utilizes **O(1)** space for its data and pointer.

Therefore, for a list containing 'n' nodes, the total space complexity is **O(n)**.



| Operation | Time Complexity | Space Complexity |
|---|---|---|
| Insertion | O(1) or O(n) | O(1) |
| Deletion | O(1) | O(1) |

# Array vs. Singly Linked List

| Feature | Array | Singly Linked List |
|---|---|---|
| Size | Fixed | Dynamic |
| Memory | Contiguous | Non-contiguous |
| Insertion at Start | O(n) | O(1) |
| Deletion at Start | O(n) | O(1) |
| Access by Index | O(1) | O(n) |

# Further Resources & Practice

## Further Reading

- GeeksforGeeks – Singly Linked List
- VisualAlgo – Linked List Demo

## Recommended Problems

83. Remove Duplicates from Sorted List

206. Reverse Linked List

876. Middle of the Linked List

**Next Session:** We will explore other types of linked lists and their operations.