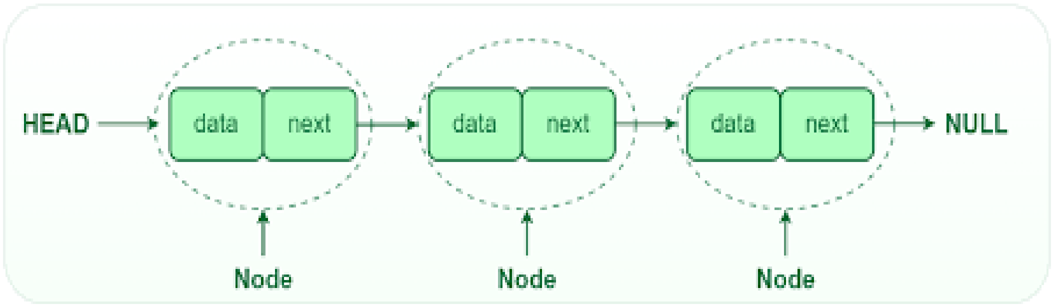
LinkedList

A linked list is a fundamental data structure in computer science used for storing a sequence of elements, called nodes. Each node contains two parts: data and a reference (or link) to the next node in the sequence. Here are some key points about linked lists:



1. **Types of Linked Lists**:
    - **Singly Linked List**: Each node points to the next node in the sequence.
    - **Doubly Linked List**: Each node has pointers to both the next and the previous nodes.
    - **Circular Linked List**: The last node points back to the first node, forming a circle.

2. **Basic Operations**:
    - **Insertion**: Adding a new node at the beginning, end, or middle of the list.
    - **Deletion**: Removing a node from the list.
    - **Traversal**: Iterating through the list to access or manipulate nodes.
    - **Search**: Finding a node with a specific value or index.

3. **Advantages**:
    - Dynamic size: Linked lists can grow or shrink in size during execution.

    - Ease of insertion and deletion: Adding or removing nodes doesn't require shifting elements, unlike arrays.
    - Versatility: Different types of linked lists offer flexibility based on the application needs.

4. **Disadvantages**:
    - **Memory Overhead**: Each node in a linked list requires extra memory for storing the link.
    - **Traversal**: Sequential access is slower compared to arrays because elements are not stored contiguously in memory.
    - **No Random Access**: Unlike arrays, accessing an element at a particular index requires traversing from the beginning.

Real Examples:-

1. Operating Systems: Memory Management
2. Music and Video Playback
3. Web Browsers: History Management
4. Databases: Transaction Management
5. Job Scheduling
6. Text Editors: Undo Functionality
7. Symbol Tables in Compilers
8. Data Packet Processing in Networking

Linked lists are commonly used in situations where the size of the data structure is unpredictable or where efficient insertion and deletion of elements are more critical than random access. Understanding linked lists is crucial for mastering data structures and algorithms, as they serve as building blocks for more complex data structures like stacks, queues, and hash tables.

// Node class represents each element in the linked list

```
class Node {
    int data;

    Node next;


    // Constructor to initialize data and next pointer

    public Node(int data) {

        this.data = data;

        this.next = null;

    }

}


// LinkedList class manages the linked list operations

class LinkedList {

    Node head; // Head of the list


    // Constructor to initialize an empty linked list

    public LinkedList() {

        this.head = null;

    }


    // Method to insert a new node at the beginning of the linked list

    public void insertAtBeginning(int data) {

        Node newNode = new Node(data); // Create a new node with the given data

        newNode.next = head; // Set the next of new node as head
        head = newNode; // Move the head to point to the new node
    }

    // Method to insert a new node at the end of the linked list
    public void insertAtEnd(int data) {
        Node newNode = new Node(data); // Create a new node with the given data
        if (head == null) { // If the list is empty, make the new node as head
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null) { // Traverse to the last node
            current = current.next;
        }
        current.next = newNode; // Link the new node to the last node
    }

    // Method to insert a new node after a given node in the linked list
    public void insertAfter(Node prevNode, int data) {
        if (prevNode == null) {
            System.out.println("Previous node cannot be null");
            return;
        }
```

```
        Node newNode = new Node(data); // Create a new node with the given data
        newNode.next = prevNode.next; // Set the next of new node as next of prevNode
        prevNode.next = newNode; // Set the next of prevNode as new node
    }

    // Method to print all nodes of the linked list
    public void printList() {
        Node current = head; // Start traversal from the head node
        while (current != null) {
            System.out.print(current.data + " "); // Print current node's data
            current = current.next; // Move to the next node
        }
        System.out.println(); // Print a new line at the end
    }
}

// Main class to test the LinkedList implementation
public class Main {
    public static void main(String[] args) {
        // Create a new linked list
        LinkedList list = new LinkedList();

        // Insert some nodes at the beginning
```

```
list.insertAtBeginning(5);

list.insertAtBeginning(10);

list.insertAtBeginning(15);


// Print the linked list after insertion at the beginning

System.out.print("Linked List after insertion at the beginning: ");

list.printList();


// Insert nodes at the end

list.insertAtEnd(20);

list.insertAtEnd(25);


// Print the linked list after insertion at the end

System.out.print("Linked List after insertion at the end: ");

list.printList();


// Find the node with data 10

Node prevNode = list.head.next; // This should point to the node with data 10


// Insert node after the node with data 10

list.insertAfter(prevNode, 12);

System.out.print("Linked List after insertion after node with data 10: ");

list.printList();
```

```
    }
}
```

/// Delete ///

```
// Node class represents each element in the linked list
class Node {
    int data;
    Node next;

    // Constructor to initialize data and next pointer
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

// LinkedList class manages the linked list operations
class LinkedList {
    Node head; // Head of the list

    // Constructor to initialize an empty linked list
    public LinkedList() {
        this.head = null;
    }

    // Method to insert a new node at the beginning of the linked list
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data); // Create a new node with the given data
        newNode.next = head; // Set the next of new node as head
        head = newNode; // Move the head to point to the new node
    }

    // Method to delete the node at the end of the linked list
    public void deleteAtEnd() {
        if (head == null) {
            System.out.println("Linked list is empty. Nothing to delete.");
            return;
        }
        if (head.next == null) {
            head = null; // If there is only one node, set head to null
            return;
        }
        Node current = head;
        Node prev = null;
        // Traverse to the last node and keep track of the previous node
        while (current.next != null) {
            prev = current;
            current = current.next;
        }

        // Now current is the last node, prev is the second-to-last node
        prev.next = null; // Remove the link to the last node
    }

    // Method to print all nodes of the linked list
    public void printList() {
        Node current = head; // Start traversal from the head node
        while (current != null) {
            System.out.print(current.data + " "); // Print current node's data
            current = current.next; // Move to the next node
        }
        System.out.println(); // Print a new line at the end
    }
}

// Main class to test the LinkedList implementation
public class Main {
```

```
public static void main(String[] args) {
    // Create a new linked list
    LinkedList list = new LinkedList();

    // Insert nodes at the beginning
    list.insertAtBeginning(5);
    list.insertAtBeginning(10);
    list.insertAtBeginning(15);

    // Print the linked list after insertion at the beginning
    System.out.print("Linked List after insertion at the beginning: ");
    list.printList();

    // Delete node at the end
    list.deleteAtEnd();
    System.out.print("Linked List after deletion at the end: ");
    list.printList();

    // Delete node at the end again
    list.deleteAtEnd();
    System.out.print("Linked List after deletion at the end again: ");
    list.printList();
```

```
        // Delete node at the end again

        list.deleteAtEnd();

        System.out.print("Linked List after deletion at the end again: ");

        list.printList();


        // Try to delete from an empty list

        list.deleteAtEnd();

        System.out.print("Linked List after deletion from an empty list: ");

        list.printList();

    }

}
```

Easy Questions

1. Write a program to create a linked list in all three ways.

2. Write a program to delete linked lists in all the three ways.

3. Write a program to find the sum of all elements in a linked list.

4. Two Linked Lists are identical when they have the same data and the arrangement of data is also the same. Write a function to check if the given two linked lists are identical.

Examples:

Input: *a = 1->2->3->4, b = 1->2->3->4*
Output: *Identical*

Input: *a = 1->3->5->2, b = 1->3->4->6*
Output: *Not Identical*

5. *Given a Singly Linked List, the task is to find the middle of the linked list. If the number of nodes are even, then there would be two middle nodes, so return the second middle node.*

   Example:

   Input: *linked list = 1 -> 2 -> 3 -> 4 -> 5*
   Output: *3*
   Explanation: *There are 5 nodes in the linked list and there is one middle node whose value is 3.*

   Input: *linked list = 1 -> 2 -> 3 -> 4 -> 5 -> 6*
   Output: *4*
   Explanation: *There are 6 nodes in the linked list, so we have two middle nodes: 3 and 4, but we will return the second middle node which is 4.*

6. Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

7. Write a function that takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.
   For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

Medium Level

1. Given a linked list, check if the linked list has a loop (cycle) or not. The below diagram shows a linked list with a loop.

2. Given an unsorted Linked List, the task is to remove duplicates from the list.

   Examples:

   Input: *linked_list = 12 -> 11 -> 12 -> 21 -> 41 -> 43 -> 21*
   Output: *12 -> 11 -> 21 -> 41 -> 43*
   Explanation: *Second occurrence of 12 and 21 are removed.*

   Input: *linked_list = 12 -> 11 -> 12 -> 21 -> 41 -> 43 -> 21*
   Output: *12 -> 11 -> 21 -> 41 -> 43*

3. Write a program to reverse a linked list

4. Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.

Input: *R->A->D->A->R->NULL*
Output: *Yes*

Input: *C->O->D->E->NULL*
Output: *No*

5. *Given two sorted linked lists consisting of* ***N*** *and* ***M*** *nodes respectively. The task is to merge both of the lists (in place) and return the head of the merged list.*

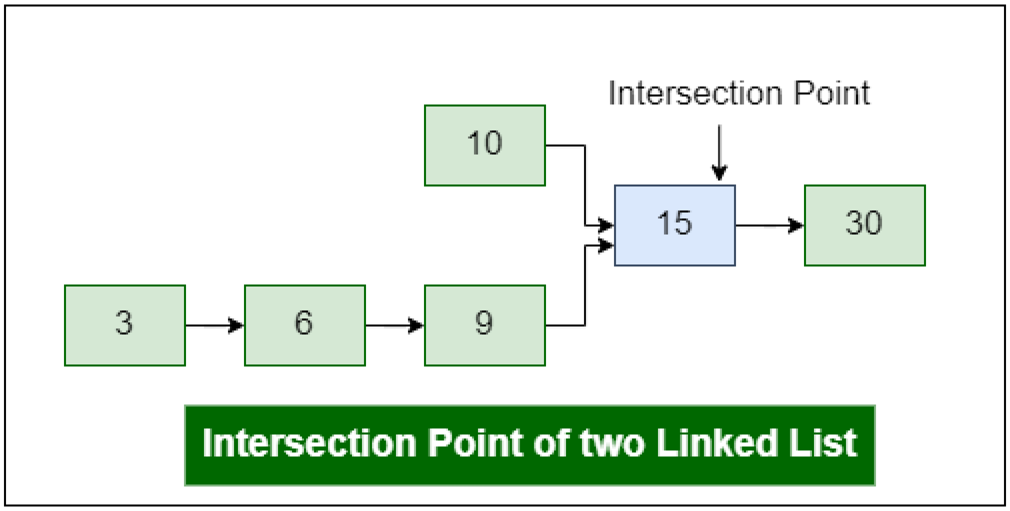
   Examples:

   Input: *a: 5->10->15, b: 2->3->20*
   Output: *2->3->5->10->15->20*

   Input: *a: 1->1, b: 2->4*
   Output: *1->1->2->4*

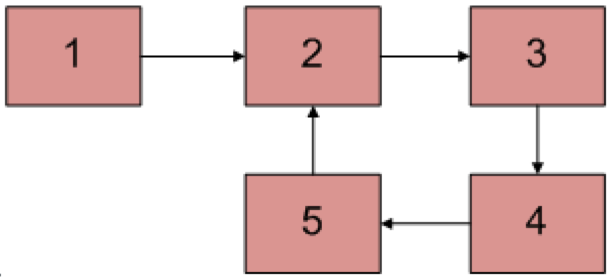
Hard

1. There are two singly linked lists in a system. By some programming error, the end node of one of the linked lists got linked to the second list, forming an inverted Y-shaped list. Write a program to get the point where two linked lists merge.



2. Given the head of a linked list. The task is to find if a loop exists in the linked list if **yes** then return the **length of the loop** in the linked list else return **0**.

   Examples:

   Input: *linked list =*
   Output: *4*
   Explanation: The loop is present in the below-linked list and the length of the loop is 4.

   Input: *linked list = 4 -> 3 -> 7 -> 9 -> 2*
   Output: *0*