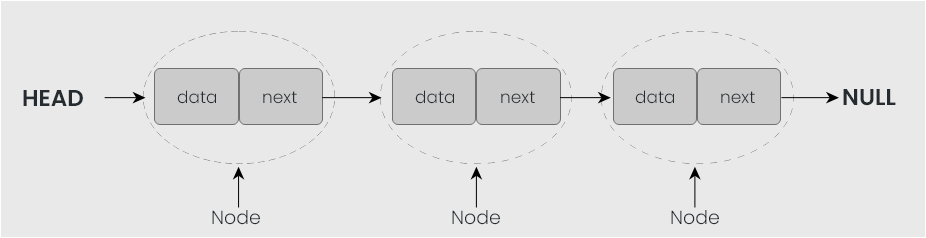
**Theory**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.



**When to Implement Linked List:**

**Dynamic Size:** When the size of the data structure needs to change dynamically during runtime.

**Frequent Insertions/Deletions**: When there is a frequent need for insertion or deletion of elements, as it can be done with O(1) complexity.

**Memory Efficiency:** When memory needs to be allocated dynamically and efficiently.

**No Random Access Requirement:** When there is no need for direct access to elements by index (like arrays).

**Implementation of Stacks and Queues:** Linked lists serve as the underlying data structure for implementing stacks and queues.

**Disadvantages :**

**Memory Overhead:** Each node in a linked list requires additional memory to store the data and the reference to the next node.

**Random Access Inefficiency:** Linked lists do not provide constant-time random access to elements. To access an element at a particular index, you have to traverse the list from the head, leading to O(n) time complexity.

**Cache Locality:** Due to scattered storage locations of nodes in memory, linked lists may not utilize cache memory effectively. This can result in slower performance compared to arrays for certain operations.

**Not Suitable for Small Data Sets:** For small datasets or applications with fixed-size requirements, the flexibility of linked lists may not be necessary, and the memory overhead can be considered wasteful.

**Vulnerability to Pointers Issues:** Manipulating pointers in linked lists can lead to pointer-related issues such as dangling pointers, memory leaks, and incorrect data connections.

**Singly Linked List :**

class Node<T> {

    T data;

    Node<T> next;

    public Node(T data) {

        this.data = data;

        this.next = null;

    }

}

class SinglyLinkedList<T>{

    Node<T> head;

    void insertAtBegining(T data) {

        Node<T> newNode = new Node<>(data);

        newNode.next = head;

        head = newNode;

    }

    void insertAtEnd(T data) {

        Node<T> newNode = new Node<>(data);

        if(head == null) {

            head = newNode;

        }

        Node<T> temp = head;

        while(temp.next != null) {

            temp = temp.next;

        }

        temp.next = newNode;

    }

    void insertAtPos(int pos, T data) {

        if(pos==1) {

            insertAtBegining(data);

            return;

        }

        Node<T> temp = head;

        // traversing till prev pos of where to add new node

        for(int i=0; i<pos-2 && temp!=null; i++) {

            temp = temp.next;

        }

        // position out of bound

        if(temp==null) {

            return;

        }

        Node<T> newNode = new Node<>(data);

        newNode.next = temp.next;

        temp.next = newNode;

    }

    void deletePos(int pos) {

        Node<T> temp = head;

        if(pos==1) {

            head = head.next;

            temp.next = null;

            return;

        }

        for(int i=0; i<pos-2 && temp!=null; i++) {

            temp = temp.next;

        }

        Node<T> t = temp.next;

        temp.next=temp.next.next;

        t.next=null;

    }

    void display() {

        if(head==null) {

            System.out.println("Empty linked list");

            return;

        }

        Node<T> temp = head;

        while(temp!=null) {

            System.out.print(temp.data+" -> ");

            temp=temp.next;

        }

    }

}

**Doubly Linked List :**

class Node<T> {

    T data;

    Node<T> prev;

    Node<T> next;

    public Node(T data) {

        this.data = data;

        this.prev = null;

        this.next = null;

    }

}

class DoublyLinkedList<T> {

    Node<T> head;

    void insertAtBeginning(T data) {

        Node<T> newNode = new Node<>(data);

        if (head == null) {

            head = newNode;

        } else {

            newNode.next = head;

            head.prev = newNode;

            head = newNode;

        }

    }

    void insertAtEnd(T data) {

        Node<T> newNode = new Node<>(data);

        if (head == null) {

            head = newNode;

        } else {

            Node<T> temp = head;

            while (temp.next != null) {

                temp = temp.next;

            }

            temp.next = newNode;

            newNode.prev = temp;

        }

    }

    void insertAtPos(int pos, T data) {

        if (pos == 1) {

            insertAtBeginning(data);

            return;

        }

        Node<T> temp = head;

        // traversing till prev pos of where to add new node

        for (int i = 0; i < pos - 2 && temp != null; i++) {

            temp = temp.next;

        }

        // position out of bound

        if (temp == null) {

            return;

        }

        Node<T> newNode = new Node<>(data);

        newNode.next = temp.next;

        newNode.prev = temp;

        if (temp.next != null) {

            temp.next.prev = newNode;

        }

        temp.next = newNode;

    }

    void deletePos(int pos) {

        if (head == null) {

            return;

        }

        Node<T> temp = head;

        if (pos == 1) {

            head = head.next;

            if (head != null) {

                head.prev = null;

            }

            temp.next = null;

            return;

        }

        for (int i = 0; i < pos - 2 && temp != null; i++) {

            temp = temp.next;

        }

        if (temp == null || temp.next == null) {

            return;

        }

        Node<T> t = temp.next;

        temp.next = t.next;

        if (t.next != null) {

            t.next.prev = temp;

        }

        t.prev = null;

        t.next = null;

    }

    void display() {

        if (head == null) {

            System.out.println("Empty linked list");

            return;

        }

        Node<T> temp = head;

        while (temp != null) {

            System.out.print(temp.data + " <-> ");

            temp = temp.next;

        }

    }

}

**Circular Linked List :**

class Node<T> {

    T data;

    Node<T> next;

    public Node(T data) {

        this.data = data;

        this.next = null;

    }

}

class CircularSinglyLinkedList<T> {

    Node<T> head;

    void insertAtBeginning(T data) {

        Node<T> newNode = new Node<>(data);

        if (head == null) {

            head = newNode;

            newNode.next = head;  // Make it circular

        } else {

            newNode.next = head;

            Node<T> temp = head;

            while (temp.next != head) {

                temp = temp.next;

            }

            temp.next = newNode;

            head = newNode;

        }

    }

    void insertAtEnd(T data) {

        Node<T> newNode = new Node<>(data);

        if (head == null) {

            head = newNode;

            newNode.next = head;  // Make it circular

        } else {

            Node<T> temp = head;

            while (temp.next != head) {

                temp = temp.next;

            }

            temp.next = newNode;

            newNode.next = head;  // Make it circular

        }

    }

    void insertAtPos(int pos, T data) {

        if (pos == 1) {

            insertAtBeginning(data);

            return;

        }

        Node<T> temp = head;

        // traversing till prev pos of where to add new node

        for (int i = 0; i < pos - 2 && temp != null; i++) {

            temp = temp.next;

        }

        // position out of bound

        if (temp == null) {

            return;

        }

        Node<T> newNode = new Node<>(data);

        newNode.next = temp.next;

        temp.next = newNode;

    }

    void deletePos(int pos) {

        if (head == null) {

            return;

        }

        Node<T> temp = head;

        if (pos == 1) {

            while (temp.next != head) {

                temp = temp.next;

            }

            if (temp == head) {

                head = null;  // List becomes empty

            } else {

                temp.next = head.next;

                head = head.next;

            }

            return;

        }

        for (int i = 0; i < pos - 2 && temp != null; i++) {

            temp = temp.next;

        }

        if (temp == null || temp.next == head) {

            return;

        }

        Node<T> t = temp.next;

        temp.next = t.next;

        t.next = null;

    }

    void display() {

        if (head == null) {

            System.out.println("Empty linked list");

            return;

        }

        Node<T> temp = head;

        do {

            System.out.print(temp.data + " -> ");

            temp = temp.next;

        } while (temp != head);

    }

}

**Floyds cycle detecting algorithm**

Floyd’s cycle detection algorithm is a pointer algorithm that uses only two pointers, moving through the sequence at different speeds.

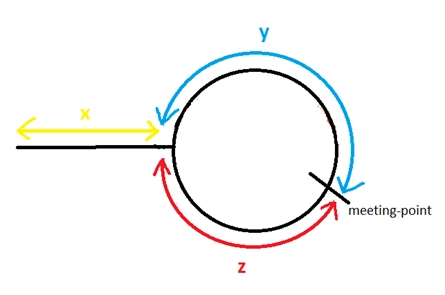
The purpose is to determine whether the linked list has a cycle or not. First, you keep two pointers of the head node. At each iteration, you move one of the pointers by two steps and the other one by one step. So you have two pointers tortoise and the hare.

Eventually one of the two cases will happen:

1. Hare will reach the tail of the linked list(null), which means that there is no cycle in it.
2. Hare will meet tortoise, which means that there is a cycle

**Detecting start node of loop :**

1. Once we know for sure that a loop is present.
2. Move the slowPointer to the start of the list,(i.e headNode) and let fastPointer remain there at the meeting point.
3. Now move both the pointers one node at a time(Yes! You heard it right. Now even fastPointer moves at one node at a time).
4. The point where both pointers will meet is our required start of the loop.



Distance travelled by slowPointer before meeting= x + y

Distance travelled by fastPointer before meeting = (x + y + z) + y

= x + 2y + z

Since fastPointer travels with double the speed of slowPointer, and time is constant for both when they reach the meeting point. So by using simple speed, time, and distance relations.

2(x+y)= x+2y+z

=> x+2y+z = 2x+2y

=> x=z

**Detect ad Delete loop in a linked list.**

Time complexity: O( N ) Space complexity: O(1)

**Detect loop**

public static boolean checkCycle(Node head){

    if(head==null || head.next==null){

      return false;

    }

    Node slow=head;

    Node fast=head;

    while(fast!=null && fast.next!=null){

      slow=slow.next;

      fast=fast.next.next;

      if(slow==fast){

        return true;

      }

    }

    return false;

  }

**Start node of loop**

public static Node startOfLoop(Node head){

    if(head==null || head.next==null){

      return head;

    }

    Node slow=head;

    Node fast=head;

    while(fast.next!=null && fast!=null){

      slow=slow.next;

      fast=fast.next.next;

      // meeting point if loop present

      if(slow==fast){

        break;

      }

    }

    slow=head;

    while(slow!=fast){

      slow=slow.next;

      fast=fast.next;

    }

    // slow contain startOfLoop node

    return slow;

  }

**Remove loop**

public static Node removeLoop(Node head) {

    // base case

    if(head==null || head.next==null){

      return head;

    }

    Node startNode=null;

    // checking for loop

    if(checkCycle(head)){

      //if loop present, store startOfLoop node to startNode

      startNode=startOfLoop(head);

    }else{

      return head;

    }

    Node temp=startNode.next;

    while(temp.next!=startNode){

      temp=temp.next;

    }

    temp.next=null;

    return head;

  }

**Merge Two Sorted Linked Lists**

**Ap 1: Iterative approach**

1. Create a dummy node which will be head of the merged list
2. T1=firstHead, t2=secondHead
3. Traverse through two list using t1 and t2
   1. Append to dummy accordingly the value

Time complexity: O( N+M ) Space complexity: O(1)

LinkedListNode<Integer> t1=first;

        LinkedListNode<Integer> t2=second;

        LinkedListNode<Integer> dummy=new LinkedListNode<Integer>(-1);

        LinkedListNode<Integer> temp=dummy;

        while(t1!=null && t2!=null){

            if(t1.data<t2.data){

                temp.next=t1;

                temp=t1;

                t1=t1.next;

            }else{

                temp.next=t2;

                temp=t2;

                t2=t2.next;

            }

        }

        if(t1!=null){

            temp.next=t1;

        }else{

            temp.next=t2;

        }

        dummy=dummy.next;

        return dummy;

**Merge Sort For Linked lists.[Very Important]**

1. Find middle node and divide list into two halves
2. Recursive call to merge sort for both halves
3. Merge two halves

Time complexity: O( Nlog(N) ) Space complexity: O(N)

static Node findMid(Node head){

        Node slow=head;

        Node fast=head.next;

        while(fast!=null && fast.next!=null){

            slow=slow.next;

            fast=fast.next.next;

        }

        return slow;

    }

    static Node merge(Node fir, Node sec){

        Node t1=fir;

        Node t2=sec;

        Node dummy=new Node(-1);

        Node temp=dummy;

        while(t1!=null && t2!=null){

            if(t1.data<t2.data){

                temp.next=t1;

                temp=temp.next;

                t1=t1.next;

            }else{

                temp.next=t2;

                temp=temp.next;

                t2=t2.next;

            }

        }

        if(t1!=null){

            temp.next=t1;

        }else{

            temp.next=t2;

        }

        return dummy.next;

    }

    public static Node sortList(Node head) {

        if(head==null || head.next==null){

            return head;

        }

        Node mid=findMid(head);

        Node right=mid.next;

        mid.next=null;

        Node left=head;

        left=sortList(left);

        right=sortList(right);

        Node mergeHead=merge(left,right);

        return mergeHead;

    }

**Why Merge Sort for LinkedLists ?**

If we have to access an ith index in a linked list using quicksort, we will have to travel every node from the head node to the ith node as we do not have a contiguous memory block. As a result, the cost of quicksort rises. Whereas merge sort sequentially accesses data, therefore the need for random access is low.

It might happen that the nodes in linked lists may not be present in nearby memory locations, therefore Merge Sort is preferred.

Unlike arrays, in linked lists, we can insert elements in the middle in O(1) extra space and O(1) time complexities if we are given a reference/pointer to the previous node. As a result, we can implement the merge operation in the merge sort without using any additional space.

Merge sort is preferred when stable sorting is needed. Stable Sorting means the order of elements with the same value remains the same after the elements have been sorted. This is significant as elements will have auxiliary data that is additional data associated with the element. Quick Sort is an unstable sorting algorithm, whereas Merge Sort is a stable method. Although Quick Sort can be adjusted to become a stable sorting algorithm, such modifications are inefficient and should not be employed.

**155\_Split a Circular linked list into two halves**

**Ap 1: Middle node by counting**

1. Find the middle point of the circular linked list by first counting the number of nodes in the circular linked list then traversing half of them from the head.
2. To make the second half of the circular linked list connect the last node, which is just before head, to the next of the middle node of the circular linked list.
3. To make the first half of the circular linked list connect the middle node of the circular linked list to the head of the circular linked list.

Time complexity: O( N ) Space complexity: O( 1 )

**Ap 2: Middle node by two pointers**

1. Find the middle point of the circular linked list using two pointers, where the 1st pointer will be traversing at twice the speed of the 2nd pointer.
2. Stop when the 1st pointer reaches two nodes prior to the head pointer.
3. To make the second half of the circular linked list connect the last node, which is just before the head, to the next of the 2nd pointer which is the middle of the circular linked list. Here, the last node will be the next node of the 1st pointer.
4. To make the first half of the circular linked list connect the 2nd pointer which is the middle of the circular linked list to the head of the circular linked list.

Time complexity: O( N ) Space complexity: O( 1 )

public static void splitCircularList(LinkedListNode<Integer> head) {

        LinkedListNode<Integer> slow=head, fast=head;

        while(fast.next.next!=head){

            fast=fast.next.next;

            slow=slow.next;

        }

        fast=fast.next;

        // connecting 2nd half

        fast.next=slow.next;

        // connecting first half

        slow.next=head;

    }