**Why linked list is needed , even though we already have a linear data structure array?**

Linked lists are needed even though arrays exist because they offer several advantages in scenarios where arrays have limitations. Here are the key reasons why linked lists are useful:

**1. Dynamic Size**

* **Arrays:** Have a fixed size, meaning you need to allocate memory in advance.
* **Linked List:** Can grow and shrink dynamically as needed, making efficient use of memory.

**2. Efficient Insertions & Deletions**

* **Arrays:** Inserting or deleting an element requires shifting elements, making it **O(n)** in the worst case.
* **Linked List:** Insertion and deletion can be **O(1)** if you have a pointer to the node.

**3. Memory Utilization**

* **Arrays:** May have unused allocated memory if not fully utilized.
* **Linked List:** Uses memory only for required elements (plus extra for pointers), making it more space-efficient for unpredictable data sizes.

**4. No Contiguous Memory Requirement**

* **Arrays:** Require a **continuous** block of memory, which may lead to memory allocation issues.
* **Linked List:** Nodes can be scattered across memory, improving flexibility.

**5. Different Variants for Specific Use Cases**

* **Singly Linked List:** Efficient for simple insertion/deletion.
* **Doubly Linked List:** Allows traversal in both directions.
* **Circular Linked List:** Useful in scenarios like scheduling (e.g., CPU process scheduling).

**When to Use a Linked List Over an Array?**

✔ When frequent insertions and deletions are required.  
✔ When the size of the data structure is unknown or changes dynamically.  
✔ When memory allocation in contiguous blocks is difficult.

However, **linked lists have drawbacks too**—extra memory for pointers and slower access times (O(n) vs O(1) for arrays).

**Dynamic arrays** (like ArrayList in Java, Vector in C++, and List in Python) address some of the limitations of static arrays. However, they still differ from linked lists in key ways. Here's how they compare:

## **Dynamic Array vs. Linked List: Key Differences**

| **Feature** | **Dynamic Array** | **Linked List** |
| --- | --- | --- |
| **Size flexibility** | Grows/shrinks dynamically but requires resizing. | Fully dynamic, grows/shrinks without resizing. |
| **Insertion (Middle)** | Costly (O(n)) due to shifting elements. | Efficient (O(1)) if pointer is known. |
| **Insertion (End)** | O(1), but may trigger resizing (O(n)). | O(1) if tail pointer is maintained. |
| **Deletion (Middle)** | Costly (O(n)) due to shifting elements. | Efficient (O(1)) if pointer is known. |
| **Random Access** | O(1) (Direct indexing). | O(n) (Sequential traversal). |
| **Memory Overhead** | Uses less memory (no extra pointers). | Uses more memory (extra pointers per node). |
| **Contiguous Memory** | Requires a single continuous memory block. | No need for contiguous memory allocation. |

### ****When to Use Dynamic Arrays vs. Linked Lists?****

✔ **Use a Dynamic Array when:**

* You need **fast random access** (O(1) indexing).
* Insertions & deletions mostly happen **at the end**.
* Memory efficiency is a concern (no extra pointers).

✔ **Use a Linked List when:**

* You have **frequent insertions & deletions** in the middle (O(1) when pointer is known).
* You **don't know the size** in advance and want **no resizing overhead**.
* You **don't need fast random access**, but efficient traversal and modification.

### ****Why Not Just Use Dynamic Arrays Always?****

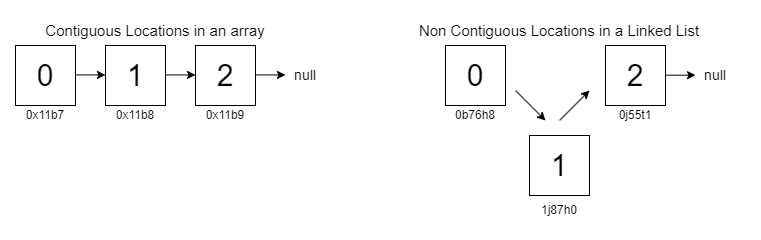
* Dynamic arrays still require **resizing (doubling size)** when full, which takes **O(n)** time.
* If frequent insertions/deletions occur **in the middle**, shifting elements is costly.
* If memory is fragmented, a dynamic array might fail to allocate a **large contiguous block**, while a linked list works fine.

So, while **dynamic arrays solve many problems of static arrays**, linked lists still shine in scenarios requiring **frequent modifications** and **non-contiguous memory storage**.

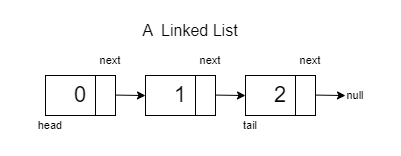
### What is a Linked List?

It is a linear data structure that can be visualized as a chain with different nodes connected, where each node represents a different element. The difference between arrays and linked lists is that, unlike arrays, the elements are not stored at a contiguous location.

Since for any element to be added in an array, we need the exact next memory location to be empty, and it is impossible to guarantee that it is possible. Hence adding elements to an array is not possible after the initial assignment of size.



A linked list is a data structure containing two crucial pieces of information, the first being the data and the other being the pointer to the next element. The ‘head’ is the first node, and the ‘tail’ is the last node in a linked list.



### ****Creating a Linked List****

There are two information sets to store at every node, thus there is a need to create a self-defined data type to handle them. Therefore, we will use the help of classes.

To understand linked lists better, let’s take the help of an example:

class Node<T>  
{  
 T data;  
 Node<T> next;  
 Node(T data,Node<T> next)  
 {  
 this.data = data;  
 this.next = next;  
 }  
 Node(T data)  
 {  
 this.data = data;  
 this.next = null;  
 }  
}

The struct has two data types: data which contains the value of the node and a pointer next, which points to the next node in the list.

There is a constructor which assigns the values to a new node.

A new keyword is used to dynamically allocate memory to a node with data as arr[0].

The combination of the given parameters and functions initializes a linked list.

### ****Understanding Pointers****

A pointer is a variable that stores the memory address of another variable. In simpler terms, it "points" to the location in memory where data is stored. This allows you to indirectly access and manipulate data by referring to its memory address.

Java does not explicitly use pointers or take the address of variables as you do in C++. Instead, we have reference variables. These reference variables do not directly contain memory addresses like pointers in languages such as C or C++. Instead, they hold references to objects in memory.

### ****Memory Space****:

Let’s talk about assuming the data stored is integer. Another main difference between an array and a linked list is the memory used. In the case of an array, we are storing integers that consume 4 Bytes for every int, whereas in a linked list, we are storing data and a pointer at every node, so the memory used up will depend on the configuration of the system.

32 Bit System 64 Bit System

Int - 4 Bytes Int - 4 Bytes

Pointer - 4 Bytes Pointer - 8 Bytes

Overall - 8 Bytes Overall - 12 Bytes

Therefore, in the case of a 64 Bit system, it occupies or consumes more space than a 32 Bit system.

**Applications of Linked Lists:**

Creating Data Structures: Linked lists serve as the foundation for building other dynamic data structures, such as stacks and queues.

Dynamic Memory Allocation: Dynamic memory allocation relies on linked lists to manage and allocate memory blocks efficiently.

***Web Browser is one important application of Linked List.***

A **web browser** is a great real-world example of a **linked list**, specifically when handling the **back and forward navigation** of web pages.

### ****How a Browser Uses a Linked List?****

When you browse the internet, the browser maintains a **history** of visited pages. This history is stored using a **doubly linked list** to allow smooth **backward and forward navigation**.

1. **Each webpage you visit is a node** in the linked list.
2. **The "Back" button moves to the previous node**, while the **"Forward" button moves to the next node**.
3. If you visit a new page after pressing back, the forward history is deleted, and a new node is created.

### ****How It Works Internally?****

#### **1. Browsing a Website (Adding Nodes)**

* When you visit a new page, a **new node is created** in the doubly linked list.
* The **previous page points to the new node**, and the new node points back to the previous one.

#### **2. Pressing the "Back" Button**

* Moves the pointer to the **previous node** (previous webpage).
* The **next pointer is still available** to move forward.

#### **3. Pressing the "Forward" Button**

* Moves the pointer to the **next node** (if available).
* Only possible if you have gone back before.

#### **4. Visiting a New Page After Going Back**

* If you visit a new page **after going back**, all "forward" history is deleted.
* A **new node is added**, breaking the previous forward connection.

### ****Why Use a Doubly Linked List Instead of a Singly Linked List?****

✔ **Efficient Bidirectional Navigation** – Moving **both backward and forward** is easy.  
✔ **No Need to Re-traverse** – In a singly linked list, moving back would require starting from the head.  
✔ **Better Memory Management** – Old pages can be removed dynamically to save space.

### ****Real-World Summary****

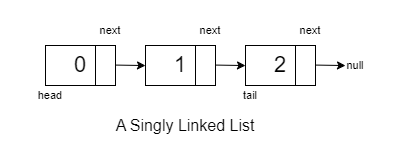
* **Nodes** represent **webpages**.
* **Back button** moves to the **previous node**.
* **Forward button** moves to the **next node**.
* **Visiting a new page after going back** deletes the forward history.

Thus, a **browser history is an excellent real-time example of a doubly linked list!**

### ****Types of Linked Lists:****

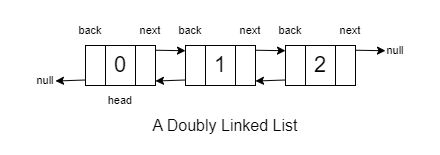
**Singly Linked Lists**:

In a singly linked list, each node points to the next node in the sequence. Traversal is straightforward but limited to moving in one direction, from the head to the tail.

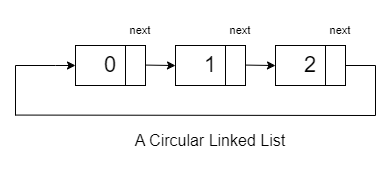


**Doubly Linked Lists:**

In this each node points to both the next node and the previous node, thus allowing it for bidirectional connectivity.



**Circular Linked Lists**:

In a circular linked list, the last node points back to the head node, forming a closed loop. 

**Singly Linked Lists**:

**Conversion of Array to LinkedList :**

public static Node converttoll(int[] arr)  
{  
 Node head = new Node(arr[0]);  
 Node mover = head;  
 for(int i=1;i<arr.length;i++)  
 {  
 Node temp = new Node(arr[i]);  
 mover.next = temp;  
 mover = temp;  
 }  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Traversel in a LinkedList :**

Node data = head;  
while(data!=null)  
{  
 System.*out*.println(data.data);  
 data = data.next;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Length of a LinkedList :**

private static int lengthofaLL(Node head){  
 int cnt=0;  
 Node temp=head;  
 while(temp!=null){  
 temp = temp.next;  
 cnt++;  
 }  
 return cnt;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Search in a LinkedList :**

public static int checkifPresent(Node head, int desiredElement) {  
 Node temp = head;  
 while (temp != null) {  
 if (temp.data == desiredElement)  
 return 1;  
 temp = temp.next;  
 }  
 return 0;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Deletion in a LinkedList :**

**1.Head:**

public static Node deleteHead(Node Head)  
{  
 if(Head == null)return null;  
 Head = Head.next;  
 return Head;  
}

**Time Complexity : O(1) Space Complexity:O(1)**

**2.Position:**

public static Node deletePosition(Node head,int k)  
{  
 if(head == null||head.next==null)return null;  
 if(k==1) return *deleteHead*(head);  
 int cnt = 0;  
 Node temp = head;  
 Node prev = null;  
 while(cnt<=k)  
 {  
 cnt++;  
 if(cnt == k)  
 {  
 prev.next = prev.next.next;  
 break;  
 }  
 prev = temp;  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(K) Space Complexity:O(1)**

**3.Element:**

public static Node deleteElement(Node head,int el)  
{  
 if(head == null)return null;  
 if(head.data.equals(el)) return *deleteHead*(head);  
 Node temp = head;  
 Node prev = null;  
 while(temp!=null)  
 {  
 if(temp.data.equals(el))  
 {  
 prev.next = prev.next.next;  
 break;  
 }  
 prev = temp;  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**4.Tail:**

public static Node deleteTail(Node head)  
{  
 if(head == null || head.next == null)return null;  
 Node temp = head;  
 while(temp.next.next != null)  
 {  
 temp = temp.next;  
 }  
 temp.next = null;  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Insertion in a LinkedList :**

**1.Head:**

public static Node insertHead(Node head,int value)  
{  
 Node temp = new Node(value,head);  
 return temp;  
}

**Time Complexity : O(1) Space Complexity:O(1)**

**2.Position:**

public static Node insertPosition(Node head,int ele,int pos)  
{  
 if(head == null)  
 {  
 if(pos == 1)  
 return new Node(ele);  
 else  
 return null;  
 }  
 if(pos == 1)  
 {  
 Node temp = new Node(ele,head);  
 return temp;  
 }  
 int cnt = 0;  
 Node temp = head;  
 while(temp != null)  
 {  
 cnt++;  
 if(cnt == pos-1)  
 {  
 Node x = new Node(ele,temp.next);  
 temp.next = x;  
 break;  
 }  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(k) Space Complexity:O(1)**

**3.Element:**

public static Node insertElement(Node head,int y,int ele)  
{  
 if(head == null)  
 {  
 return null;  
 }  
 if(head.data.equals(y))  
 {  
 return new Node(ele,head);  
 }  
 Node temp = head;  
 while(temp.next != null)  
 {  
 if(temp.next.data.equals(y))  
 {  
 Node x = new Node(ele,temp.next);  
 temp.next = x;  
 break;  
 }  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**4.Tail:**

public static Node insertTail(Node head,int value)  
{  
 if(head == null)return new Node(value);  
 Node temp = head;  
 while(temp.next != null)  
 {  
 temp = temp.next;  
 }  
 Node last = new Node(value);  
 temp.next = last;  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Doubly Linked Lists:**

### ****Creating a Doubly Linked List****

There are three information sets to store at every node, thus there is a need to create a self-defined data type to handle them. Therefore, we will use the help of classes.

To understand linked lists better, let’s take the help of an example:

class DoublyLinkedList<T>  
{  
 T data;  
 DoublyLinkedList<T> prev;  
 DoublyLinkedList<T> next;  
  
 DoublyLinkedList(T data,DoublyLinkedList<T> prev,DoublyLinkedList<T> next)  
 {  
 this.data = data;  
 this.prev = prev;  
 this.next = next;  
 }  
 DoublyLinkedList(T data)  
 {  
 this.data = data;  
 this.prev = null;  
 this.next = null;  
 }  
}

**Conversion of Array to Doubly LinkedList :**

public static DoublyLinkedList convert(int[] arr)  
{  
 DoublyLinkedList head = new DoublyLinkedList(arr[0]);  
 DoublyLinkedList prev = head;  
 for(int i=1;i<arr.length;i++)  
 {  
 DoublyLinkedList temp = new DoublyLinkedList(arr[i],prev,null);  
 prev.next = temp;  
 prev = temp;  
 }  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Traversal of a LinkedList :**

**Same as Singly linked list.**

**Length of a LinkedList :**

**Same as Singly linked list.**

**Search in a LinkedList :**

**Same as Singly Linked List.**

**Deletion in a LinkedList :**

**1.Head:**

public static DoublyLinkedList deleteHead(DoublyLinkedList head)  
{  
 if(head == null || head.next == null)  
 {  
 return null;  
 }  
 DoublyLinkedList temp = head;  
 head = head.next;  
 temp.next = null;  
 head.prev = null;  
 return head;  
}

**Time Complexity : O(1) Space Complexity:O(1)**

**2.Tail:**

public static DoublyLinkedList deleteTail(DoublyLinkedList head)  
{  
 if(head == null || head.next == null)  
 {  
 return null;  
 }  
 DoublyLinkedList temp = head;  
  
 while(temp.next != null)  
 {  
 temp = temp.next;  
 }  
 temp.prev.next = null;  
 temp.prev = null;  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**3.Position:**

public static DoublyLinkedList deleteK(DoublyLinkedList head,int k)  
{  
 if(head == null)  
 {  
 return null;  
 }  
 int cnt = 1;  
 DoublyLinkedList temp = head;  
 while(temp.next != null)  
 {  
 if(cnt == k)break;  
 cnt++;  
 temp = temp.next;  
 }  
 DoublyLinkedList back = temp.prev==null?null:temp.prev;  
 DoublyLinkedList front = temp.next==null?null:temp.next;  
 if(cnt<k)  
 {  
 return head;  
 }  
 else if(temp.prev == null && temp.next == null)  
 {  
 return null;  
 }  
 else if(temp.prev == null)  
 {  
 return *deleteHead*(head);  
 }  
 else if(temp.next == null)  
 {  
 return *deleteTail*(head);  
 }  
 back.next = front;  
 front.prev=back;  
 temp.next=null;  
 temp.prev=null;  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**4.Node:**

public static void deleteNode(DoublyLinkedList node)  
{  
 if(node == null)  
 {  
 return;  
 }  
 DoublyLinkedList back = node.prev;  
 DoublyLinkedList front = node.next;  
 if(front == null)  
 {  
 back.next = null;  
 node.prev = null;  
 return;  
 }  
 back.next = front;  
 front.prev = back;  
 node.next = null;  
 node.prev = null;  
}

**Time Complexity : O(1) Space Complexity:O(1)**

**Insertion in a LinkedList :**

**1.Head:**

public static DoublyLinkedList insertHead(DoublyLinkedList head,int ele)  
{  
 DoublyLinkedList newHead = new DoublyLinkedList(ele,null,head);  
 head.prev = newHead;  
 return newHead;  
}

**Time Complexity : O(1) Space Complexity:O(1)**

**2.Tail:**

public static DoublyLinkedList insertTail(DoublyLinkedList head,int k)  
{  
 if(head.next == null)  
 {  
 return *insertHead*(head,k);  
 }  
 DoublyLinkedList front = head;  
 while(front.next!=null)  
 {  
 front = front.next;  
 }  
 DoublyLinkedList back = front.prev;  
 DoublyLinkedList current = new DoublyLinkedList(k,back,front);  
 back.next = current;  
 front.prev = current;  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**3.Position:**

public static DoublyLinkedList insertK(DoublyLinkedList head,int k,int ele)  
{  
 if(head == null)  
 {  
 return null;  
 }  
 if(k == 1)  
 {  
 return *insertHead*(head,ele);  
 }  
 DoublyLinkedList front = head;  
 int cnt = 0;  
 while(front != null)  
 {  
 cnt++;  
 if(cnt == k)break;  
 front = front.next;  
 }  
 if(cnt<k)  
 {  
 return head;  
 }  
 DoublyLinkedList back = front.prev;  
 DoublyLinkedList current = new DoublyLinkedList(ele,back,front);  
 back.next = current;  
 front.prev = current;  
 return head;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**4.Node:**

public static void insertNode(DoublyLinkedList next,int ele)  
{  
 if(next == null)  
 {  
 return;  
 }  
 DoublyLinkedList back = next.prev;  
 DoublyLinkedList curr = new DoublyLinkedList(ele,back,next);  
 back.next = curr;  
 next.prev = curr;  
}

**Time Complexity : O(1) Space Complexity:O(1)**

**Reversal in a Singly LinkedList :**

public ListNode reverseList(ListNode head)  
{  
 ListNode prev = null;  
 ListNode curr = head;  
 ListNode next = null;  
 while(curr!=null)  
 {  
 next = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = next;  
 }  
 return prev;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Reversal in a Doubly LinkedList :**

**Approach 1:**

**A brute-force approach involves replacing data in a doubly linked list. First, we traverse the list and store node data in a stack. Then, in a second pass, we assign elements from the stack to nodes, ensuring a reverse order replacement since stacks follow the Last-In-First-Out (LIFO) principle.**

public DoublyLinkedList reverseDLL1(DoublyLinkedList head)  
{  
 Stack<Integer> st = new Stack<>();  
 DoublyLinkedList temp = head;  
 while(temp!=null){  
 st.push((Integer) temp.data);  
 temp = temp.next;  
 }  
 temp = head;  
 while(temp!=null){  
 temp.data = st.pop();  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(2n) Space Complexity:O(n)**

**Approach 2:**

public DoublyLinkedList reverseDLL(DoublyLinkedList head)  
{  
 if(head == null||head.next==null)return head;  
 DoublyLinkedList temp = null;  
 DoublyLinkedList curr = head;  
 while(curr != null)  
 {  
 temp = curr.prev;  
 curr.prev = curr.next;  
 curr.next = temp;  
 curr = curr.prev;  
 }  
 return temp.prev;  
}

**Time Complexity : O(n) Space Complexity:O(1)**

**Singly LinkedList :**

**Find the middle of the linked list.**

*Given the head of a singly linked list, return the middle node of the linked list.*

*If there are two middle nodes, return the second middle node.*

**Example 1:**

Input: head = [1,2,3,4,5]

Output: [3,4,5]

Explanation: The middle node of the list is node 3.

**Example 2:**

Input: head = [1,2,3,4,5,6]

Output: [4,5,6]

Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.

**Brute Force:**

Using the brute force approach, we can find the middle node of a linked list by traversing the linked list and finding the total number of nodes as `count`. Then we reset the traversal pointer and traverse to the node at the [count/2 + 1]th position. That will be the middle node.

public static Node findMiddle(Node head)   
{  
 if (head == null || head.next == null) {  
 return head;  
 }  
 Node temp = head;  
 int count = 0;  
  
 while (temp != null) {  
 count++;  
 temp = temp.next;  
 }  
  
 int mid = count / 2;  
 temp = head;  
 while (mid != 0) {  
 mid = mid - 1;  
 temp = temp.next;  
 }  
 return temp;  
}

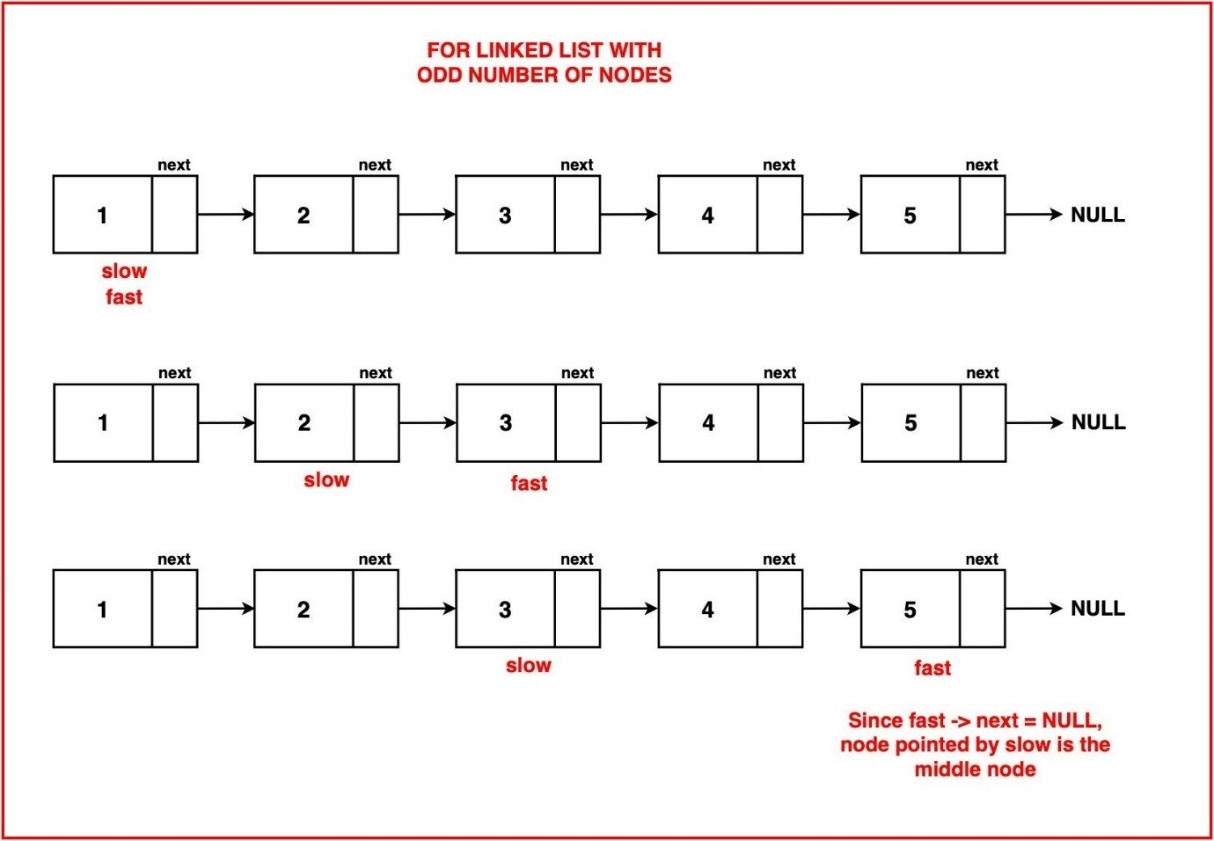
**Time Complexity : O(2n) Space Complexity:O(1)**

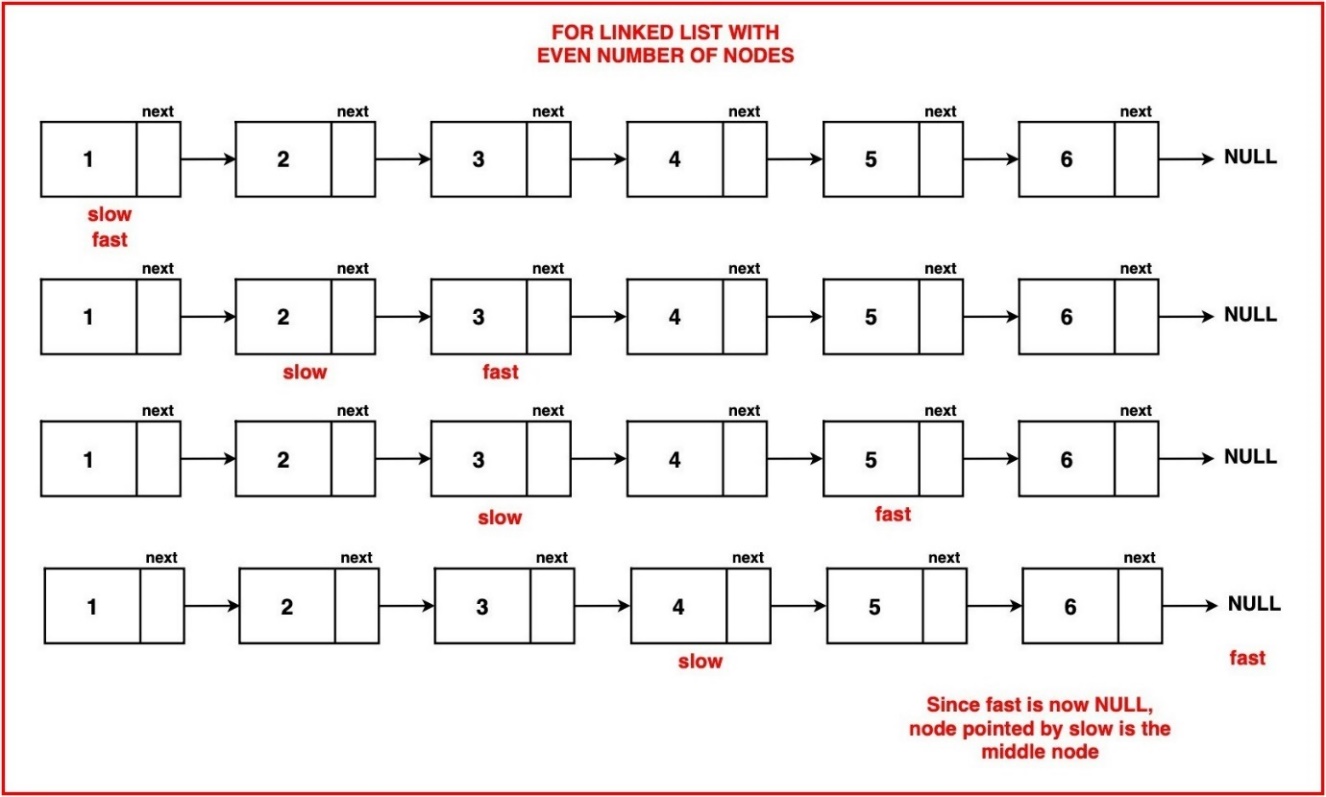
**Optimal (TortoiseHare Algo):**

**The previous method requires the traversal of the linked list twice. To enhance efficiency, the Tortoise and Hare Algorithm is introduced as an optimization where the middle node can be found in just one traversal.**

**The Tortoise and Hare algorithm leverages two pointers, 'slow' and 'fast', initiated at the beginning of the linked list. The 'slow' pointer advances one node at a time, while the 'fast' pointer moves two nodes at a time.**

The Tortoise and Hare algorithm works because the fast-moving hare reaches the end of the list in exactly the same time it takes for the slow-moving tortoise to reach the middle. When the hare reaches the end, the tortoise is guaranteed to be at the middle of the list.





public static Node findMiddle(Node head)  
{  
 Node slow = head;  
 Node fast = head;  
 while(fast!=null&&fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next.next;  
 }  
 return slow;  
}

**Time Complexity : O(n/2) Space Complexity:O(1)**

**Reverse a linked list.**

**Given the head of a singly linked list, reverse the list, and return the reversed list .**

**Example 1:**

**Input: head = [1,2,3,4,5]**

**Output: [5,4,3,2,1]**

**Example 2:**

**Input: head = [1,2]**

**Output: [2,1]**

**Example 3:**

**Input: head = []**

**Output: []**

**Brute Force:**

**A straightforward approach to reversing a singly linked list requires an additional data structure to temporarily store the values. We can use a stack for this. By pushing each node onto the stack as we move through the list, we effectively reverse the order of the nodes. Once all the nodes are stored in the stack, we rebuild the reversed linked list by popping nodes from the stack and assigning them to the nodes. The result is a new linked list with the elements in the opposite order of the original list.**

public static Node reverseLinkedList(Node head) {  
 Node temp = head;  
 Stack<Integer> stack = new Stack<>();  
 while (temp != null) {  
 stack.push(temp.data);  
 temp = temp.next;  
 }  
 temp = head;  
  
 while (temp != null) {  
 temp.data = stack.pop();  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(2n) Space Complexity : O(n)**

**Optimal 1: Iterative**

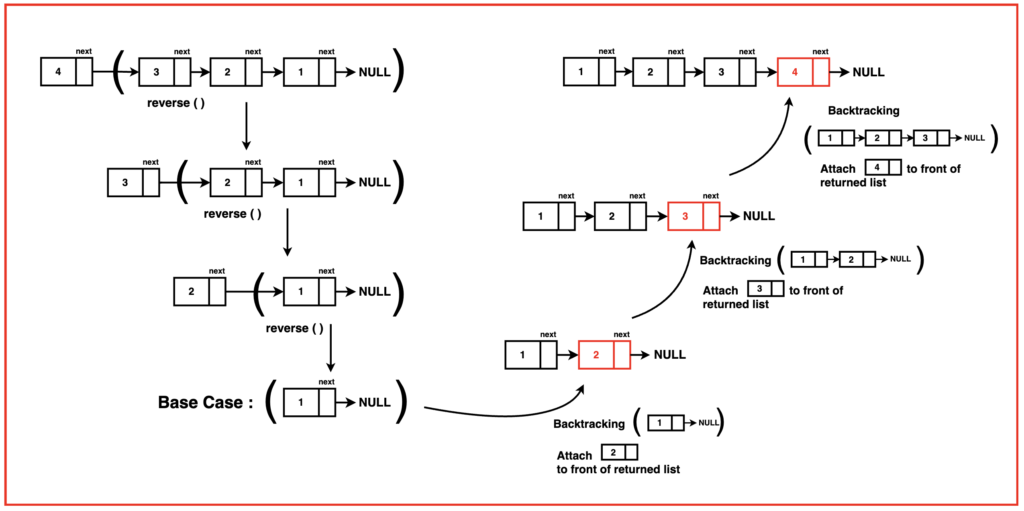
**The main idea is to flip the order of connections in the linked list, which changes the direction of the arrows. When this happens, the last element becomes the new first element of the list. This in-place reversal allows us to efficiently transform the original list without using extra space.**

public Node reverseList(Node head)  
{  
 Node prev = null;  
 Node curr = head;  
 Node next = null;  
 while(curr!=null)  
 {  
 next = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = next;  
 }  
 return prev;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

**Optimal 2: Recursive**

In this case, tackling the larger problem involves reversing a linked list with N = 4 nodes. Recursion allows us to break this task down into progressively smaller subproblems, starting with the case of 3 nodes, then the last 2 nodes, and ultimately reaching the base case where only 1 node remains. In the base case, reversing the linked list is straightforward, as a list with just one node is already in its reversed form, and we can simply return it as is.



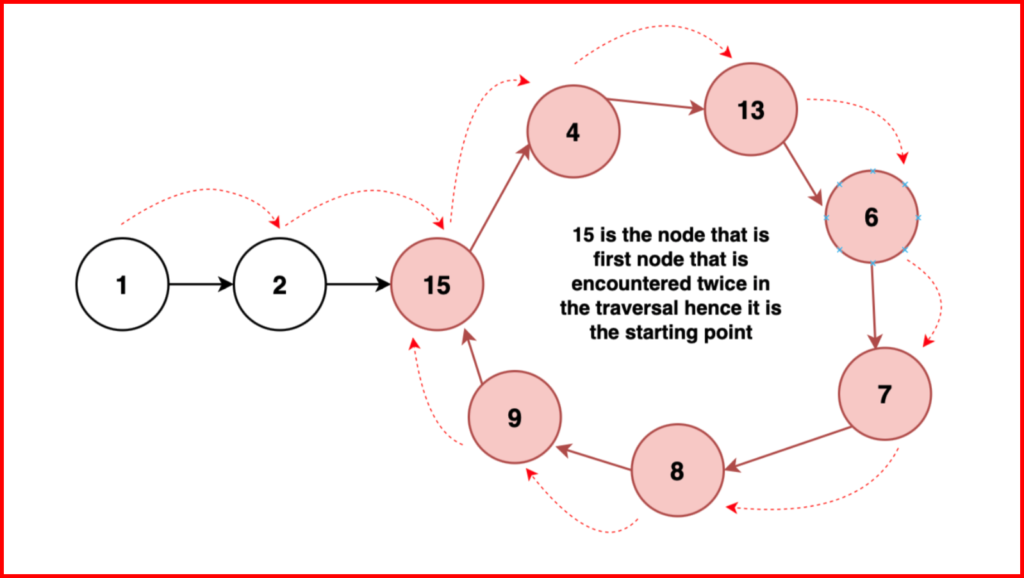
public static Node reverseList1(Node head)  
{  
 if(head == null || head.next == null)return head;  
 Node resultHead = *reverseList1*(head.next);  
 Node front = head.next;  
 front.next = head;  
 head.next = null;  
 return resultHead;  
}

**Time Complexity : O(n) Space Complexity : O(n)**

**Detect a cycle in linked list.**

**Given head, the head of a linked list, determine if the linked list has a cycle in it.**

**There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.**



**Return true if there is a cycle in the linked list. Otherwise, return false.**

**Example 1:**



**Input: head = [3,2,0,-4], pos = 1**

**Output: true**

**Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).**

**Example 2:**



**Input: head = [1,2], pos = 0**

**Output: true**

**Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.**

**Example 3:**



**Input: head = [1], pos = -1**

**Output: false**

**Explanation: There is no cycle in the linked list.**

**Brute Force:**

**A loop in a linked list occurs when there's a node that, when followed, brings you back to it, indicating a closed loop in the list.**

Hence it's important to keep track of nodes that have already been visited so that loops can be detected. One common way to do this is by using hashing.

public boolean hasCycle(ListNode head)  
{  
 ListNode temp = head;  
 Map<ListNode,Integer> map = new HashMap<>();  
 while(temp != null)  
 {  
 if(map.containsKey(temp))  
 {  
 return true;  
 }  
 map.put(temp,1);  
 temp = temp.next;  
 }  
 return false;  
}

**Time Complexity : O(n) Space Complexity : O(n)**

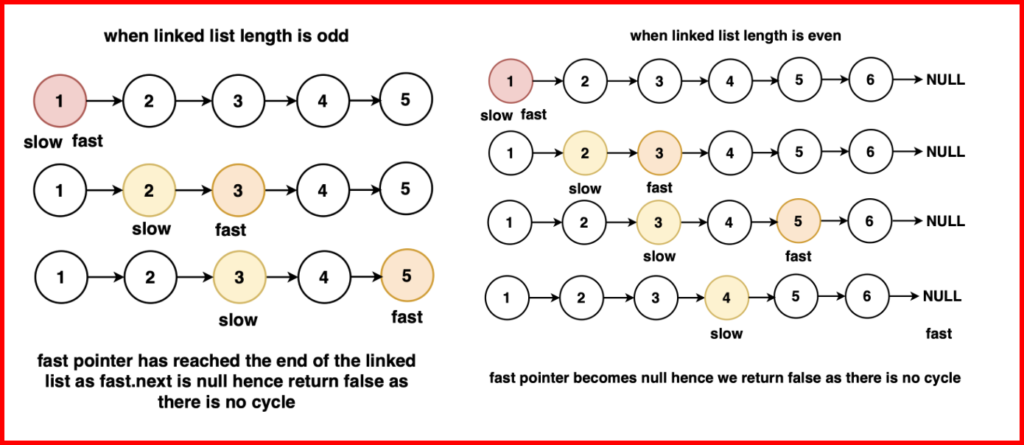
**Optimal :**

**The previous method uses O(N) additional memory, which can become quite large as the linked list length grows. To enhance efficiency, the Tortoise and Hare Algorithm is introduced as an optimization.**

**The Tortoise and Hare approach has been discussed in this article.**

**When the tortoise and hare enter the loop, they may be at different positions within the loop due to the difference in their speeds. The hare is moving faster, so it will traverse a greater distance in the same amount of time.**

**If there is no loop in the linked list, the hare will eventually reach the end, and the algorithm will terminate without a meeting occurring.**



### ****Intuition:****

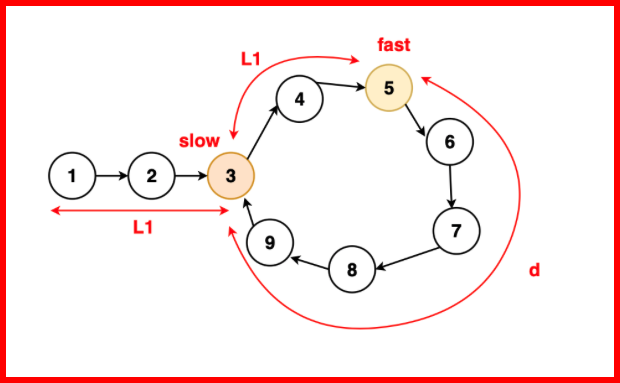
**In a linked list with a loop, consider two pointers: one that moves one node at a time (slow) and another that moves two nodes at a time (fast). If we start moving these pointers with their defined speed they will surely enter the loop and might be at some distance 'd' from each other within the loop.**

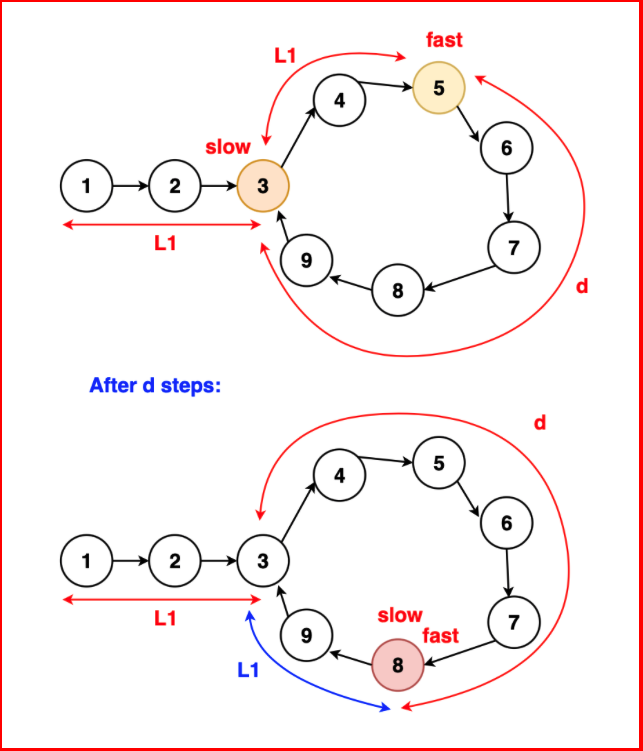
**The key insight here is the relative speed between these pointers. The fast pointer, moving at double the speed of the slow one, closes the gap between them by one node in every iteration. This means that with each step, the distance decreases by one node.**

**Imagine a race where one runner moves at twice the speed of another. The faster runner covers the ground faster and closes the gap, resulting in a reduction in the distance between them. Similarly, the fast pointer catches up to the slow pointer in the looped linked list, closing in the gap between them until it reaches zero.**

### ****Proof:****

**Let 'd' denote the initial distance between the slow and fast pointers inside the loop. At each step, the fast pointer moves ahead by two nodes while the slow pointer advances by one node.**





**The relative speed between them causes the gap to decrease by one node in each iteration (fast gains two nodes while slow gains one node). This continuous reduction ensures that the difference between their positions decreases steadily. Mathematically, if the fast pointer gains ground twice as fast as the slow pointer, the difference in their positions reduces by one node after each step. Consequently, this reduction in the distance between them continues until the difference becomes zero.**

**Hence, the proof lies in this iterative process where the faster rate of the fast pointer leads to a continual decrease in the gap distance, ultimately resulting in their collision within the looped linked list.**

public boolean hasCycle1(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;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Starting point of a cyclic linked list:****

**Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.**

**There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.**

**Do not modify the linked list.**

**Example 1:**



**Input: head = [3,2,0,-4], pos = 1**

**Output: tail connects to node index 1**

**Explanation: There is a cycle in the linked list, where tail connects to the second node.**

**Example 2:**



**Input: head = [1,2], pos = 0**

**Output: tail connects to node index 0**

**Explanation: There is a cycle in the linked list, where tail connects to the first node.**

**Example 3:**



**Input: head = [1], pos = -1**

**Output: no cycle**

**Explanation: There is no cycle in the linked list.**

**Brute Force:**

**A loop in a linked list occurs when there's a node that, when followed, brings you back to it, indicating a closed loop in the list.**

Hence it's important to keep track of nodes that have already been visited so that loops can be detected. One common way to do this is by using hashing.

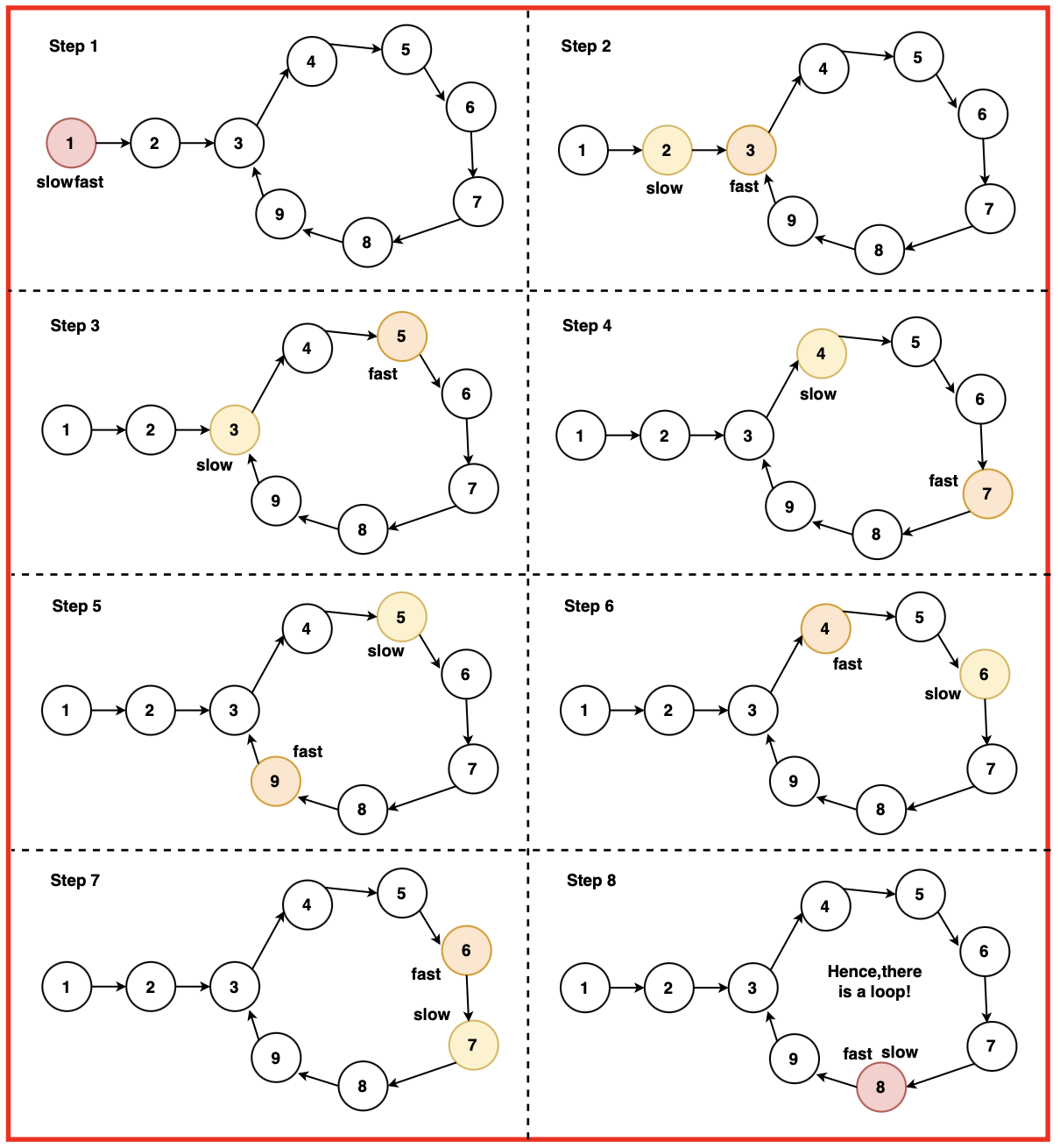
public Node detectCycle(Node head)  
{  
 Map<Node,Integer> map = new HashMap<>();  
 Node temp = head;  
 while(temp != null)  
 {  
 if(map.containsKey(temp))  
 {  
 return temp;  
 }  
 map.put(temp,1);  
 temp = temp.next;  
 }  
 return null;  
}

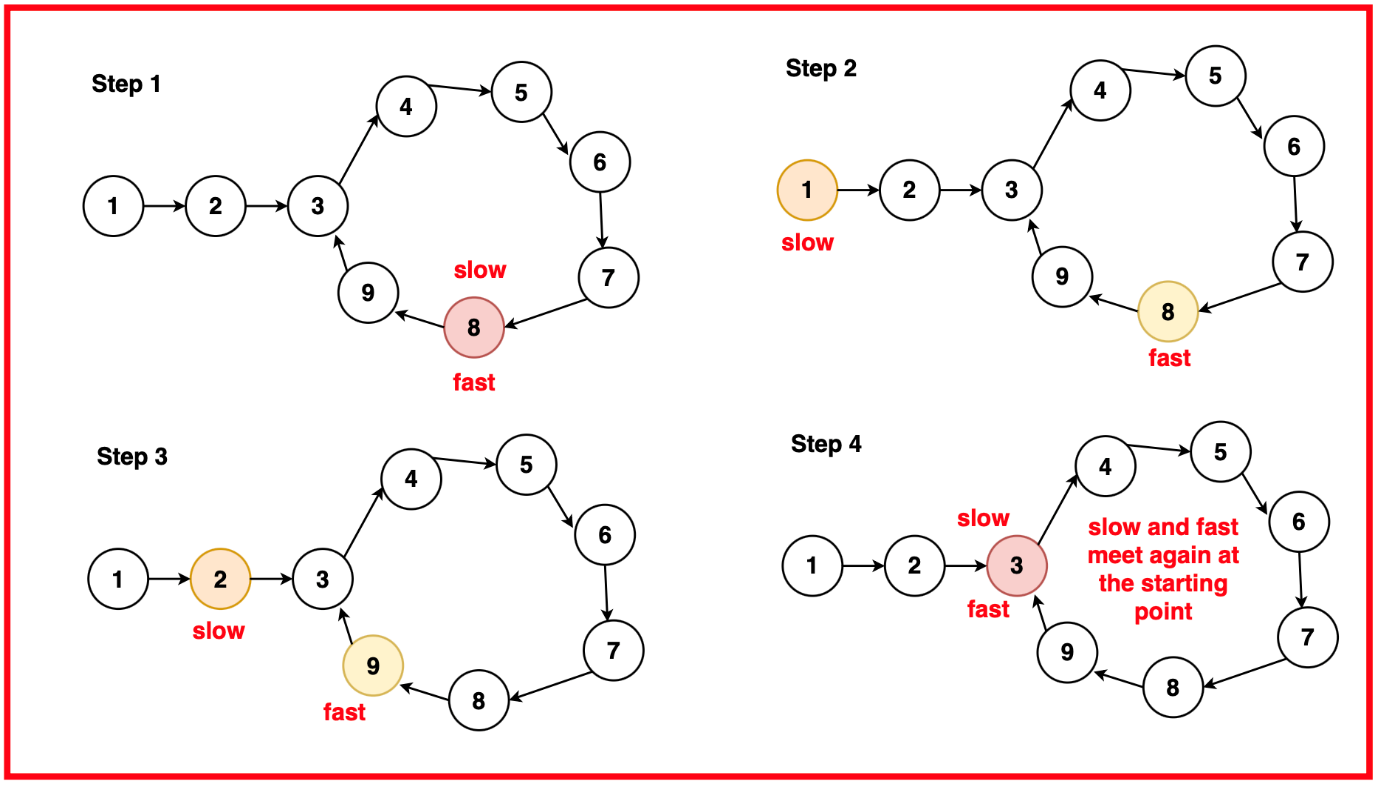
**Time Complexity : O(n) Space Complexity : O(n)**

**Optimal :**

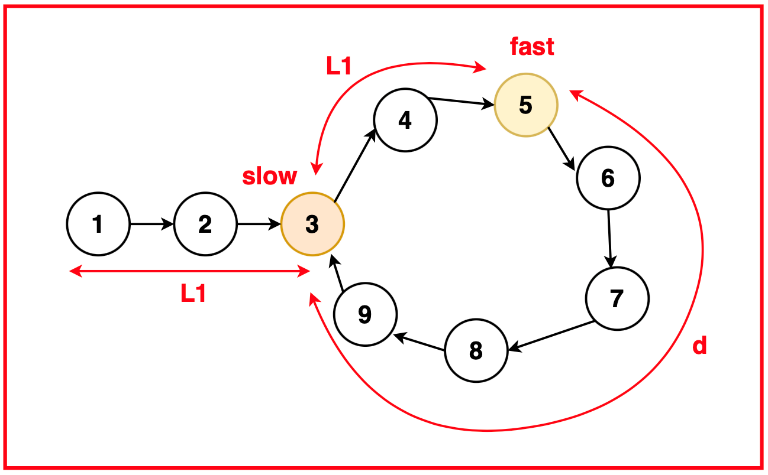
**The previous method uses O(N) additional memory, which can become quite large as the linked list length grows. To enhance efficiency, the Tortoise and Hare Algorithm is introduced as an optimization.**

**The Tortoise and Hare approach has been discussed in this article. The key insight is that when the slow and fast pointers meet inside the loop, the distance travelled by each pointer can be used to calculate the starting point of the loop.**





**You may be curious about the proof for this algorithm, and it hinges on the idea that the point where the slow and fast pointers converge can be leveraged to determine the starting point of the loop.**



**In the "tortoise and hare" algorithm for detecting loops in a linked list, when the slow pointer (tortoise) reaches the starting point of the loop, the fast pointer (hare) is positioned at a point that is twice the distance travelled by the slow pointer. This is because the hare moves at double the speed of the tortoise.**

**If slow has travelled distance L1 then fast has travelled 2 x L1. Now that slow and fast have entered the loop, the distance fast will have to cover to catch up to slow is the total length of loop minus L1. Let this distance be d.**

**Distance travelled by slow = L1**

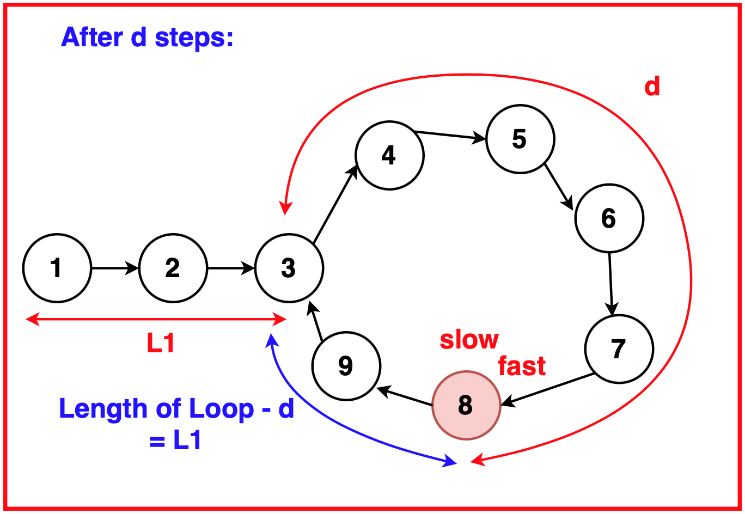
**Distance travelled by fast = 2 \* L1**

**Total length of loop = L1 + d**

**In this configuration, the fast pointer advances toward the slow pointer with two jumps per step, while the slow pointer moves away with one jump per step. As a result, the gap between them decreases by 1 with each step. Given that the initial gap is d, it takes exactly d steps for them to meet.**

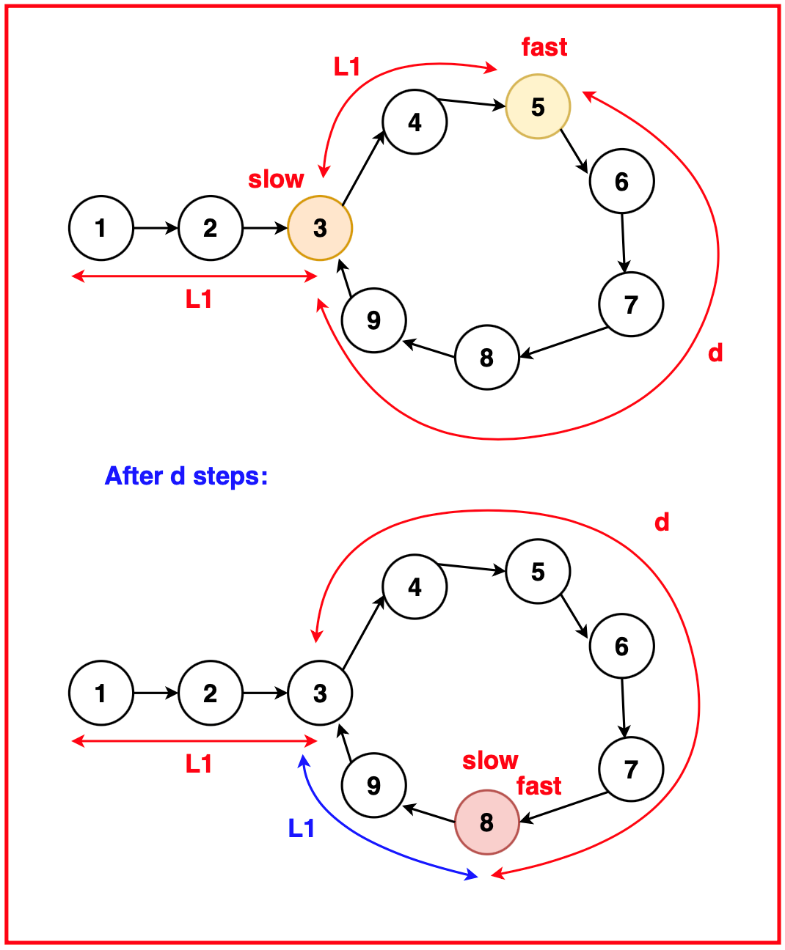
**Total length of loop = L1 + d**

**Distance between slow and fast= d**



**During these d steps, the slow pointer effectively travels d steps from the starting point within the loop and fast travels 2 x d and they meet a specific point. Based on our previous calculations, the total length of the loop is L1 + d. And since the distance covered by the slow pointer within the loop is d, the remaining distance within the loop is equal to L1.**

**Therefore, it is proven that the distance between the starting point of the loop and the point where the two pointers meet is indeed equal to the distance between the starting point and head of the linked list.**

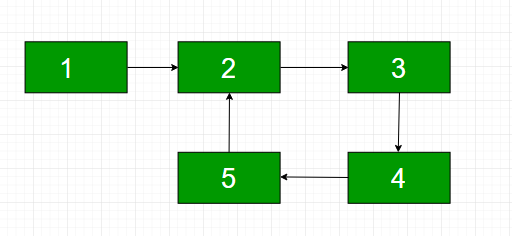


public Node detectCycle1(Node head)  
{  
 Node slow = head;  
 Node fast = head;  
 while(fast!=null && fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next.next;  
 if(slow == fast)  
 {  
 slow = head;  
 while(slow!=fast)  
 {  
 slow = slow.next;  
 fast = fast.next;  
 }  
 return slow;  
 }  
 }  
 return null;  
}

**Time Complexity : O(2n) Space Complexity : O(1)**

### ****Find the length of the loop in a linked list:****

Given the head of a linked list, determine whether the list contains a loop. If a loop is present, **return the number of nodes** in the loop, otherwise **return 0**.

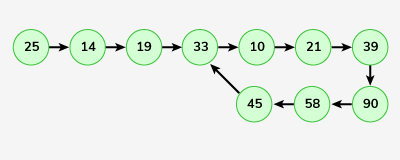


**Note: '**c**'**is the position of the node which is the next pointer of the last node of the linkedlist. If c is 0, then there is no loop.

**Examples:**

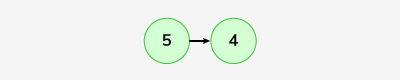
**Input:** LinkedList: 25->14->19->33->10->21->39->90->58->45, c = 4

**Output:** 7

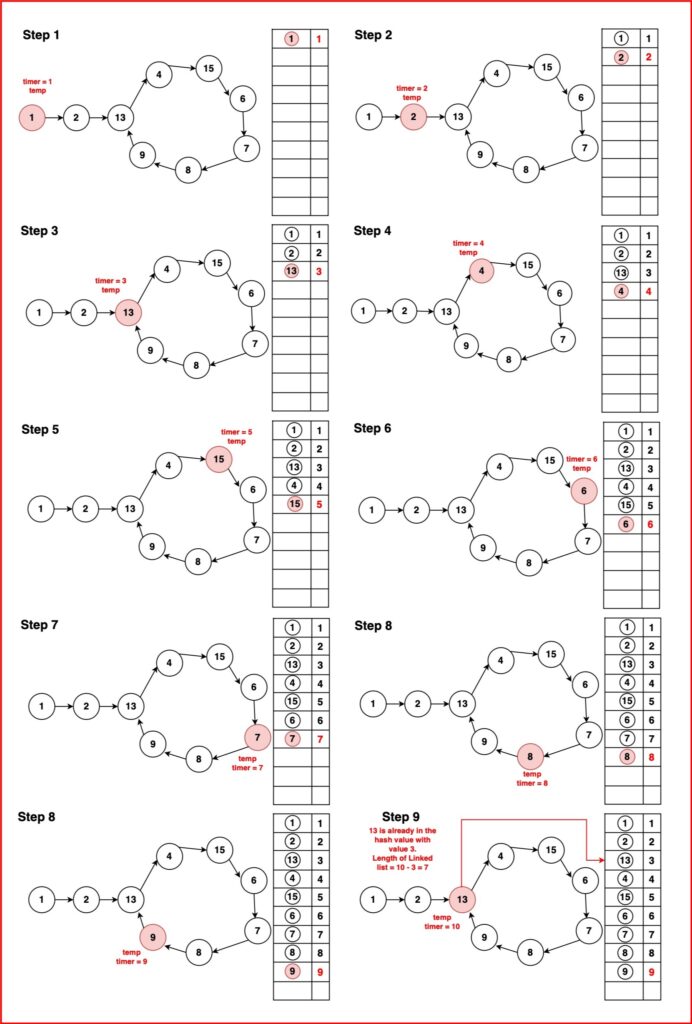
**Explanation:** The loop is from 33 to 45. So length of loop is 33->10->21->39-> 90->58->45 = **7.**The number 33 is connected to the last node of the linkedlist to form the loop because according to the input the 4th node from the beginning(1 based indexing)   
will be connected to the last node for the loop.  


**Input:** LinkedList: 5->4, c = 0

**Output:** 0

**Explanation:** There is no loop.  


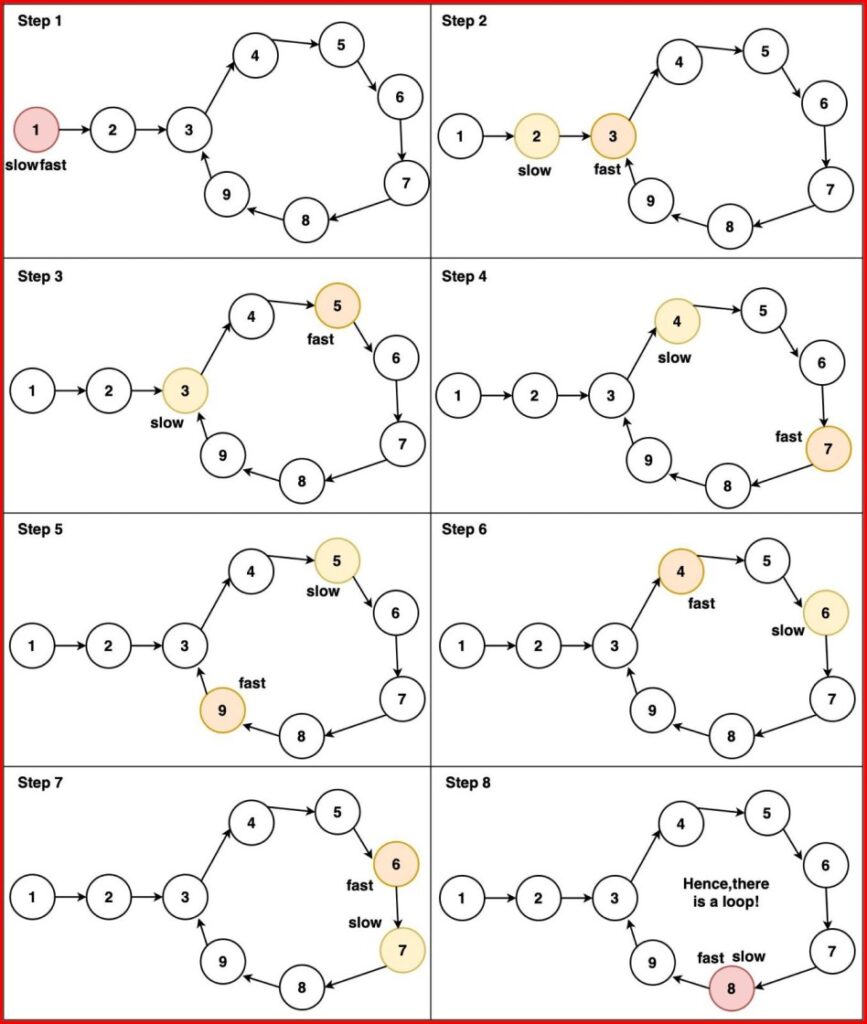
**Brute Force:**

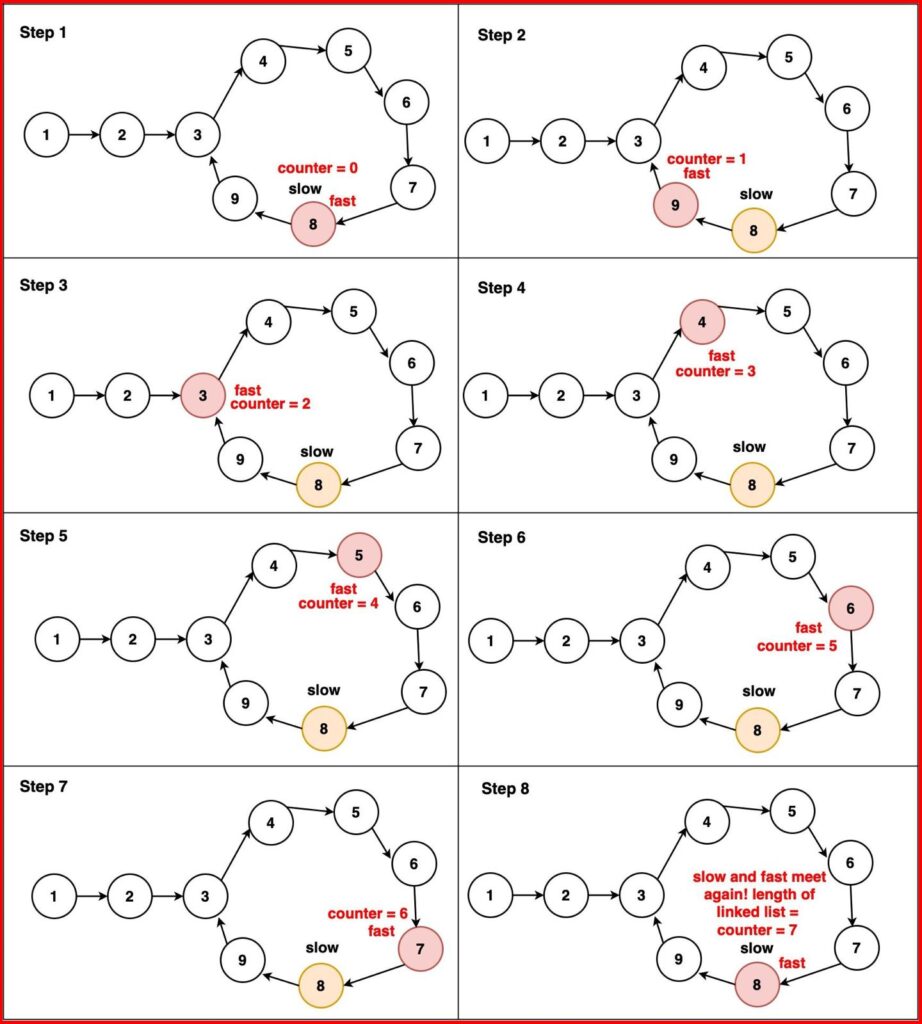


public int countNodesinLoop(Node head)  
{  
 Map<Node,Integer> map = new HashMap<>();  
 Node temp = head;  
 int cnt = 1;  
 while(temp != null)  
 {  
 if(map.containsKey(temp))  
 {  
 return cnt-map.get(temp);  
 }  
 map.put(temp,cnt);  
 cnt++;  
 temp = temp.next;  
 }  
 return 0;  
}

**Time Complexity : O(n) Space Complexity : O(n)**

**Optimal:**





public int countNodesinLoop1(Node head)  
{  
 Node slow = head;  
 Node fast = head;  
 while(fast!=null && fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next.next;  
 if(slow == fast)  
 {  
 int cnt = 1;  
 fast = fast.next;  
 while(slow!=fast)  
 {  
 cnt++;  
 fast = fast.next;  
 }  
 return cnt;  
 }  
 }  
 return 0;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Check if LinkedList Is Palindrome or not:****

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

**Example 1:**

Input: head = [1,2,2,1]

Output: true

**Example 2:**

Input: head = [1,2]

Output: false

**Brute Force:**

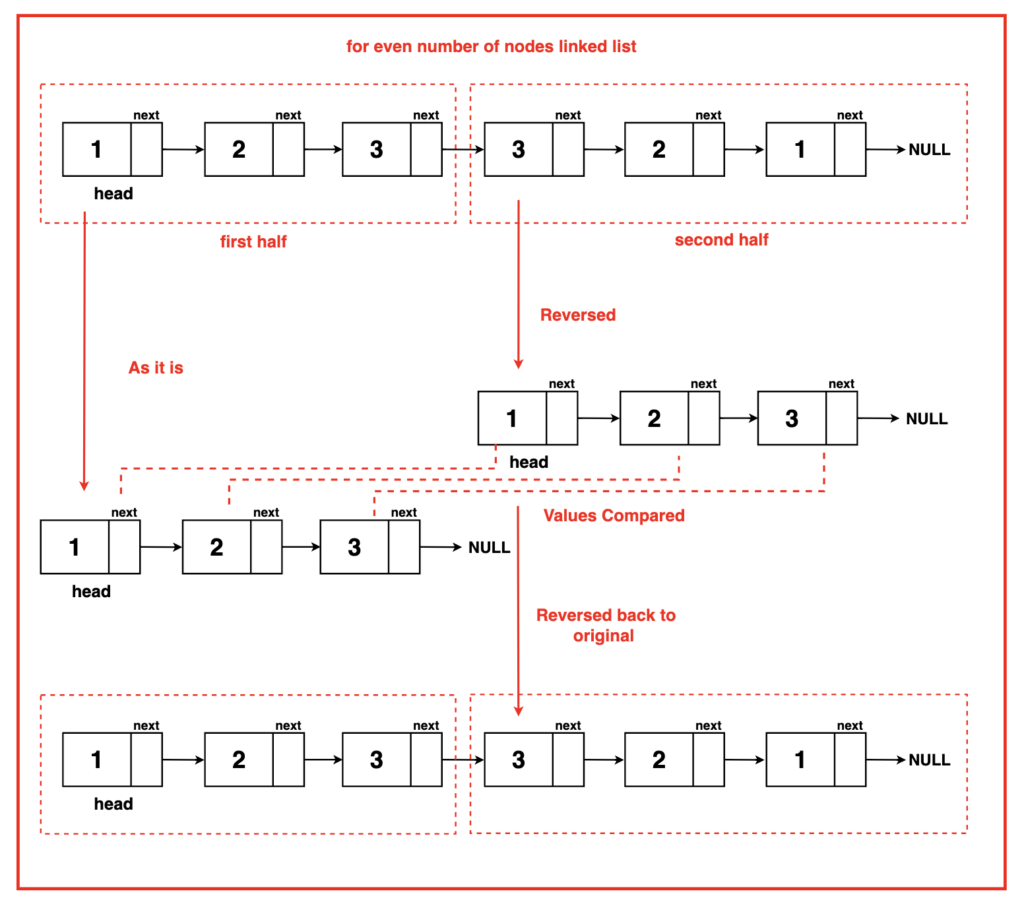
A straightforward approach to checking if the given linked list is a palindrome or not is to temporarily store the values in an additional data structure. We can use a stack for this. By pushing each node onto the stack as we traverse the list, we effectively store the data values in the reverse order. Once all the nodes are stored in the stack, we traverse the linked list again comparing each node's value with the values popped from the top of the stack

public boolean isPalindrome(Node head)  
{  
 Stack<Integer> st = new Stack<>();  
 Node temp = head;  
 while(temp != null)  
 {  
 st.push((Integer) temp.data);  
 temp = temp.next;  
 }  
 temp = head;  
 while(temp != null)  
 {  
 if(temp.data != st.peek())  
 {  
 return false;  
 }  
 st.pop();  
 temp = temp.next;  
 }  
 return true;  
}

**Time Complexity : O(2n) Space Complexity : O(n)**

.**Optimal:**

The previous approach uses O(N) additional space, which can be avoided by reversing only half of the linked list and comparing the first and second halves. If they match, reverse the portion that was originally reversed, and then return true else return false.



public static boolean isPalindrome1(Node head)  
{  
 Node mid = *middle*(head);  
 mid = *reverseHalf*(mid);  
 Node temp = head;  
 Node midTemp = mid;  
 while(midTemp != null)  
 {  
 if(temp.data != midTemp.data)  
 {  
 *reverseHalf*(mid);  
 return false;  
 }  
 temp = temp.next;  
 midTemp = midTemp.next;  
 }  
 *reverseHalf*(mid);  
 return true;  
}  
public static Node middle(Node head)  
{  
 Node slow = head;  
 Node fast = head;  
 while(fast!=null && fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next.next;  
 }  
 return slow;  
}  
public static Node reverseHalf(Node mid)  
{  
 Node prev = null;  
 Node next = null;  
 Node curr = mid;  
 while(curr != null)  
 {  
 next = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = next;  
 }  
 return prev;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Segregate odd and even nodes in LL:****

Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The first node is considered odd, and the second node is even, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in O(1) extra space complexity and O(n) time complexity.

**Example 1:**

Input: head = [1,2,3,4,5]

Output: [1,3,5,2,4]

**Example 2:**

Input: head = [2,1,3,5,6,4,7]

Output: [2,3,6,7,1,5,4]

**Brute Force:**

Using extra array , store the odd values and even values In the order and then replace the linkedlist with the corresponding values.

public ListNode oddEvenList(ListNode head)  
{  
 if(head == null || head.next == null)return head;  
 ListNode odd = head;  
 ListNode even = head.next;  
 List<Integer> order = new ArrayList<>();  
 while(odd!=null && odd.next!=null)  
 {  
 order.add(odd.val);  
 odd = odd.next.next;  
 }  
 if(odd!=null)order.add(odd.val);  
 while(even!=null && even.next != null)  
 {  
 order.add(even.val);  
 even = even.next.next;  
 }  
 if(even!=null)order.add(even.val);  
 ListNode temp = head;  
 for(int i=0;i<order.size();i++)  
 {  
 temp.val = order.get(i);  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(2n) Space Complexity : O(n)**

.**Optimal:**

Simultaneously altering the even and odd nodes , and then connecting the odd with even and then returing the LinkedList.

public Node oddEvenList(Node head)  
{  
 if(head == null || head.next == null)return head;  
 Node odd = head;  
 Node even = head.next;  
 Node evenHead = head.next;  
 while(even != null && even.next != null)  
 {  
 odd.next = odd.next.next;  
 even.next = even.next.next;  
 odd = odd.next;  
 even = even.next;  
 }  
 odd.next = evenHead;  
 return head;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Remove Nth node from the back of the LL:****

Given the head of a linked list, remove the nth node from the end of the list and return its head.

**Example 1:**

Input: head = [1,2,3,4,5], n = 2

Output: [1,2,3,5]

**Example 2:**

Input: head = [1], n = 1

Output: []

**Example 3:**

Input: head = [1,2], n = 1

Output: [1]

**Brute Force:**

The simplest way to delete the Nth node from the end is to delete the (L-N+1)th node from the start of the linked list, where L is the total length of the linked list.

public ListNode removeNthFromEnd(ListNode head, int n)  
{  
 if(head == null)return null;  
 int length = findLength(head);  
 if(n==1)//tail  
 {  
 return deleteTail(head);  
 }  
 else if(n == length)  
 {  
 return deleteHead(head);  
 }  
 else  
 {  
 int len = length - n;  
 return deleteMiddle(head,len);  
 }  
}  
public ListNode deleteMiddle(ListNode head,int n)  
{  
 int cnt = 1;  
 ListNode temp = head;  
 while(temp != null && cnt != n)  
 {  
 cnt++;  
 temp = temp.next;  
 }  
 ListNode del = temp.next;  
 temp.next = temp.next.next;  
 del = null;  
 return head;  
}  
public ListNode deleteHead(ListNode head)  
{  
 if(head.next == null)return null;  
 ListNode newHead = head.next;  
 head = null;  
 return newHead;  
}  
public ListNode deleteTail(ListNode head)  
{  
 if(head.next == null)return null;  
 ListNode temp = head;  
 while(temp.next.next != null)  
 {  
 temp = temp.next;  
 }  
 temp.next = null;  
 return head;  
}  
public int findLength(ListNode head)  
{  
 ListNode temp = head;  
 int cnt = 0;  
 while(temp != null)  
 {  
 cnt++;  
 temp = temp.next;  
 }  
 return cnt;  
}

**Time Complexity : O(L)+O(L-N) Space Complexity : O(1)**

**Optimal:**

To enhance efficiency, we will involve two pointers, a fast pointer and a slow pointer. The fast-moving pointer will initially be exactly N nodes ahead of the slow-moving pointer. After which, both of them will move one step at a time. When the fast pointer reaches the last node.

public ListNode removeNthFromEnd(ListNode head, int n)  
{  
 if(head == null)return null;  
 ListNode slow = head;  
 ListNode fast = head;  
 for(int i=0;i<n;i++)  
 fast = fast.next;  
 if(fast == null)  
 return head.next;  
 while(fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next;  
 }  
 ListNode del = slow.next;  
 slow.next = slow.next.next;  
 del = null;  
 return head;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Delete the middle node of the LL:****

**You are given the head of a linked list. Delete the middle node, and return the head of the modified linked list.**

**The middle node of a linked list of size n is the ⌊n / 2⌋th node from the start using 0-based indexing, where ⌊x⌋ denotes the largest integer less than or equal to x.**

**For n = 1, 2, 3, 4, and 5, the middle nodes are 0, 1, 1, 2, and 2, respectively.**

**Example 1:**

**Input: head = [1,3,4,7,1,2,6]**

**Output: [1,3,4,1,2,6]**

**Example 2:**

**Input: head = [1,2,3,4]**

**Output: [1,2,4]**

**Example 3:**

**Input: head = [2,1]**

**Output: [2]**

**Brute Force:**

**Using the brute force approach, the middle node of a linked list can be determined by traversing the entire linked list initially to find the total number of nodes, denoted as `n`. Then we reset the temporary node `temp` to the head of the linked list and proceed to traverse to the node positioned at index `[n/2]`, which represents either the middle node in an odd-lengthed list or the node just before the middle node in an even-lengthed linked list.**

public Node deleteMiddle1(Node head) {  
 Node temp = head;  
 int n = 0;  
 while (temp != null) {  
 n++;  
 temp = temp.next;  
 }  
 int res = n / 2;  
 temp = head;  
 while (temp != null) {  
 res--;  
 if (res == 0) {  
 Node middle = temp.next;  
 temp.next = temp.next.next;  
 middle = null;  
 break;  
 }  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(n + n/2) Space Complexity : O(1)**

**Optimal:**

**To enhance efficiency, the Tortoise and Hare Algorithm is introduced as an optimization where the middle node can be found in just one traversal.**

**The 'slow' pointer advances one node at a time while the 'fast' pointer moves two nodes at a time. By doing so, when the 'fast' pointer reaches the end of the list, the 'slow' pointer will be at the middle node.**

**Since we require `slow` to reach the node before the middle node, we give `fast` a head start!**

public Node deleteMiddle(Node head)  
{  
 if(head == null||head.next == null)return null;  
 Node slow = head;  
 Node fast = head.next.next;  
 Node prev = null;  
 while(fast!=null && fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next.next;  
 }  
 Node del = slow.next;  
 slow.next = slow.next.next;  
 del = null;  
 return head;  
}

**Time Complexity : O(n/2) Space Complexity : O(1)**

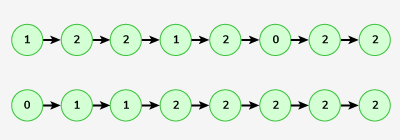
### ****Sort a LL of 0's 1's and 2's by changing links:****

Given a linked list where nodes can contain values **0s**, **1s,** and **2s**only. The task is to segregate **0s**, **1s,** and **2s** linked list such that all zeros segregate to the head side, 2s at the end of the linked list, and 1s in the middle of 0s and 2s.

**Examples:**

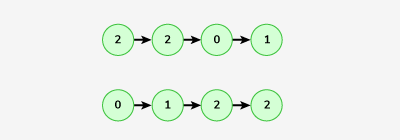
**Input:** LinkedList:1->2->2->1->2->0->2->2

**Output:** 0->1->1->2->2->2->2->2

**Explanation:** All the 0s are segregated to the left end of the linked list, 2s to the right end of the list, and 1s in between.  


**Input:** LinkedList: 2->2->0->1

**Output:** 0->1->2->2

**Explanation:** After arranging all the 0s,1s and 2s in the given format, the output will be 0 1 2 2.  


**Brute Force:**

This code sorts a **linked list** containing only 0, 1, and 2, following a **counting sort** approach. Instead of rearranging nodes, it modifies the **node values** in a structured way.

1. **Counting the occurrences of 0s, 1s, and 2s:**
   * Traverse the list once and count how many nodes have values 0, 1, and 2.
   * Store these counts in cnt0, cnt1, and cnt2.
2. **Overwriting the linked list:**
   * Traverse the list again and **overwrite** the first cnt0 nodes with 0.
   * Then, overwrite the next cnt1 nodes with 1.
   * Finally, overwrite the remaining cnt2 nodes with 2.
   * This effectively sorts the linked list in **O(N) time**.

public static Node segregate(Node head)  
{  
 int cnt0 = 0;  
 int cnt1 = 0;  
 int cnt2 = 0;  
 Node temp = head;  
 while(temp!=null)  
 {  
 if(temp.data == 0)cnt0++;  
 else if(temp.data == 1)cnt1++;  
 else cnt2++;  
 temp = temp.next;  
 }  
 temp = head;  
 for(int i=0;i<cnt0;i++)  
 {  
 temp.data = 0;  
 temp = temp.next;  
 }  
 for(int i=0;i<cnt1;i++)  
 {  
 temp.data = 1;  
 temp = temp.next;  
 }  
 for(int i=0;i<cnt2;i++)  
 {  
 temp.data = 2;  
 temp = temp.next;  
 }  
 return head;  
}

**Time Complexity : O(2n) Space Complexity : O(1)**

**Optimal:**

This code sorts a **linked list** containing only 0s, 1s, and 2s **without modifying node values**. Instead of counting occurrences and rewriting node values (like in the counting sort approach), it **rearranges the node links directly** using three separate lists.

This approach follows the **Dutch National Flag Algorithm**, where:

* **One list stores nodes with value 0**.
* **One list stores nodes with value 1**.
* **One list stores nodes with value 2**.

After distributing the nodes into three separate lists, the code **connects these lists in order** (0s → 1s → 2s) to produce a sorted linked list.

static Node segregate(Node head)  
{  
 if(head == null || head.next == null)return head;  
 Node zh = new Node(-1);  
 Node zt = zh;  
 Node oh = new Node(-1);  
 Node ot = oh;  
 Node th = new Node(-1);  
 Node tt = th;  
 Node temp = head;  
 while(temp != null)  
 {  
 if(temp.data == 0)  
 {  
 zt.next = temp;  
 zt = zt.next;  
 }  
 else if(temp.data == 1)  
 {  
 ot.next = temp;  
 ot = ot.next;  
 }  
 else  
 {  
 tt.next = temp;  
 tt = tt.next;  
 }  
 temp = temp.next;  
 }  
 zt.next = oh.next != null?oh.next:th.next;  
 ot.next = th.next;  
 tt.next = null;  
 return zh.next;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Intersection of two linked lists:****

Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:

The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must retain their original structure after the function returns.

**Example 1:**

Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

Output: Intersected at '8'

Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

- Note that the intersected node's value is not 1 because the nodes with value 1 in A and B (2nd node in A and 3rd node in B) are different node references. In other words, they point to two different locations in memory, while the nodes with value 8 in A and B (3rd node in A and 4th node in B) point to the same location in memory.

**Example 2:**

Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

Output: Intersected at '2'

Explanation: The intersected node's value is 2 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.

**Example 3:**

Input: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

Output: No intersection

Explanation: From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA and skipB can be arbitrary values.

Explanation: The two lists do not intersect, so return null.

**Brute Force:**

**Solution 1:**

This approach **uses a HashMap to detect the intersection** of two linked lists. The key idea is to **store all nodes of one linked list in a HashMap** and then check if any node from the second linked list already exists in the map.

If a node exists in both lists, that is the **intersection point**.

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {  
 Map<ListNode,boolean> map = new HashMap<>();  
 ListNode temp = headA;  
 while(temp != null)  
 {  
 map.put(temp,true);  
 temp = temp.next;  
 }  
 temp = headB;  
 while(temp != null)  
 {  
 if(map.containsKey(temp))  
 {  
 return temp;  
 }  
 temp = temp.next;  
 }  
 return null;  
}

**Time Complexity : O(m+n) Space Complexity : O(m)**

**Solution 2:**

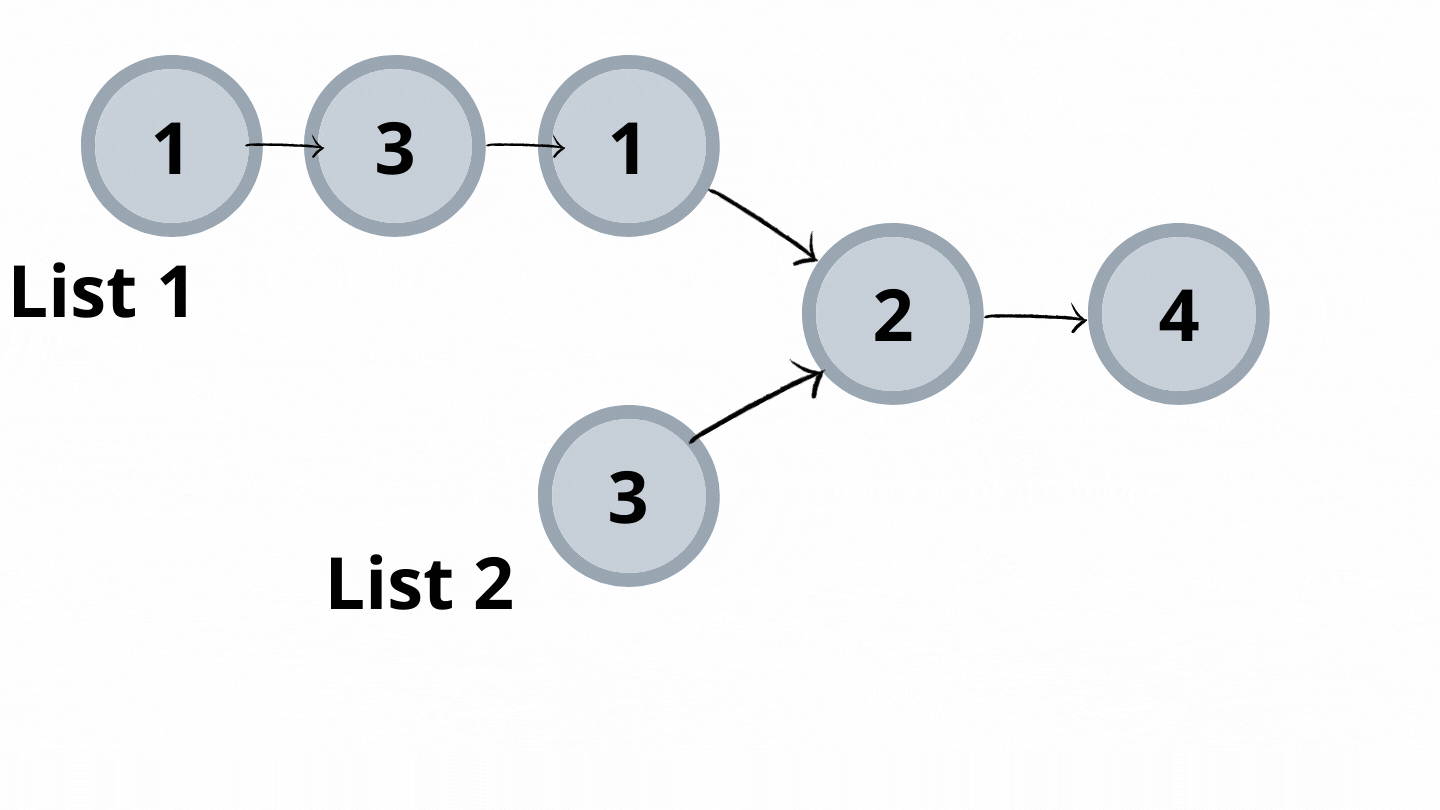
**We will reduce the search length. This can be done by searching the length of the shorter linked list. How? Let’s see the process.**

**Find the length of both lists.**

**Find the positive difference between these lengths.**

**Move the dummy pointer of the larger list by the difference achieved. This makes our search length reduced to a smaller list length.**

**Move both pointers, each pointing two lists, ahead simultaneously if both do not collide.**



static int getDifference(Node head1,Node head2) {  
 int len1 = 0,len2 = 0;  
 while(head1 != null || head2 != null) {  
 if(head1 != null) {  
 ++len1; head1 = head1.next;  
 }  
 if(head2 != null) {  
 ++len2; head2 = head2.next;  
 }  
  
 }  
 return len1-len2;//if difference is neg-> length of list2 > length of list1 else vice-versa  
}  
//utility function to check presence of intersection  
static Node intersectionPresent(Node head1,Node head2) {  
 int diff = *getDifference*(head1,head2);  
 if(diff < 0)  
 while(diff++ != 0) head2 = head2.next;  
 else while(diff-- != 0) head1 = head1.next;  
 while(head1 != null) {  
 if(head1 == head2) return head1;  
 head2 = head2.next;  
 head1 = head1.next;  
 }  
 return head1;  
  
}

**Time Complexity:**

O(2max(length of list1,length of list2))+O(abs(length of list1-length of list2))+O(min(length of list1,length of list2))

Reason: Finding the length of both lists takes max(length of list1, length of list2) because it is found simultaneously for both of them. Moving the head pointer ahead by a difference of them. The next one is for searching.

**Space Complexity: O(1)**

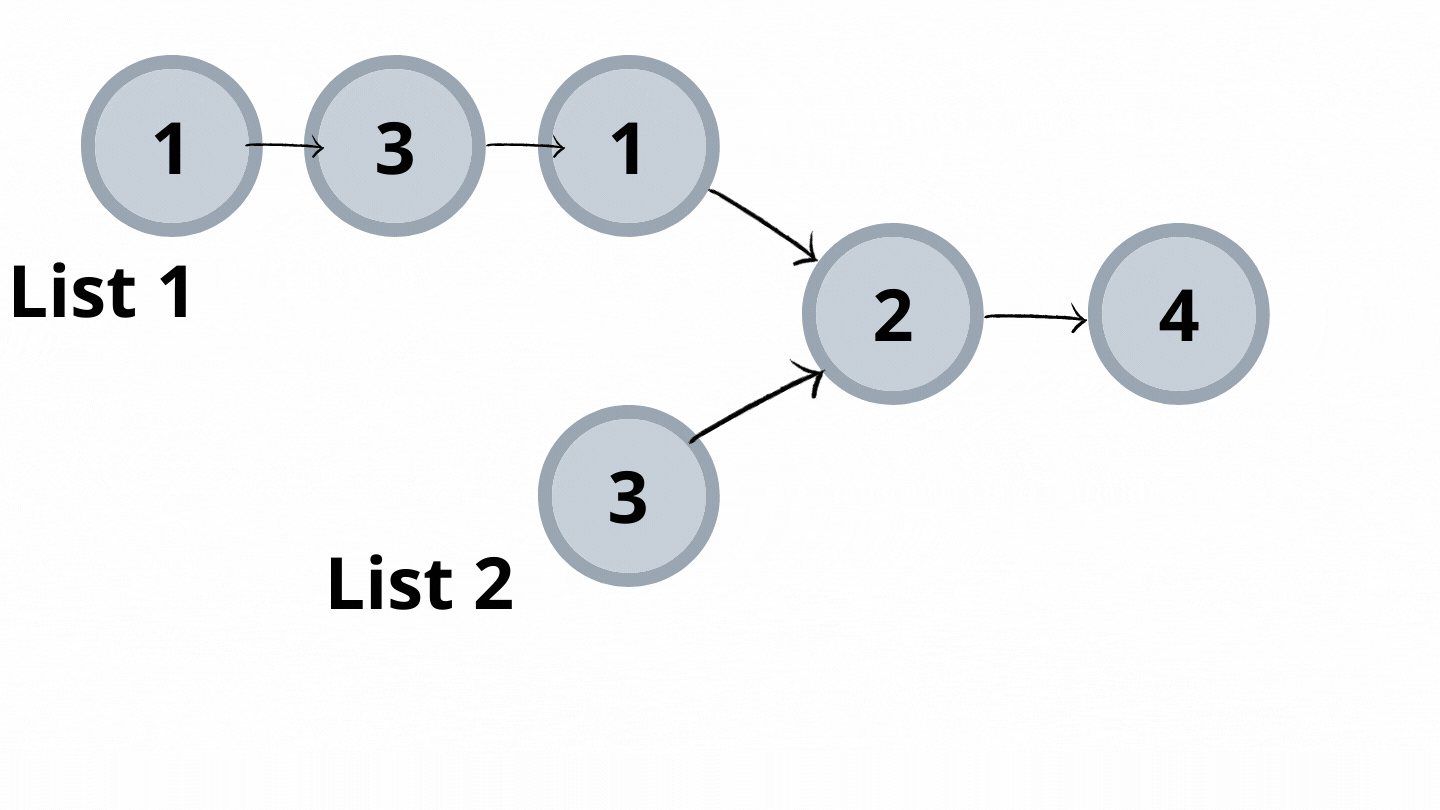
Reason: No extra space is used.

**Optimal:**

The difference of length method requires various steps to work on it. Using the same concept of difference of length, a different approach can be implemented. The process is as follows:-

Take two dummy nodes for each list. Point each to the head of the lists.

Iterate over them. If anyone becomes null, point them to the head of the opposite lists and continue iterating until they collide.



public ListNode getIntersectionNode(ListNode headA, ListNode headB) {  
 ListNode t1 = headA;  
 ListNode t2 = headB;  
 while(t1!=t2)  
 {  
 t1 = t1.next;  
 t2 = t2.next;  
 if(t1 == t2) return t1;  
 if(t1 == null)t1 = headB;  
 if(t2 == null)t2 = headA;  
 }  
 return t1;  
}

**Time Complexity : O(m+n) Space Complexity : O(1)**

### ****Add 1 to the number represented in LL:****

**Solution 1:**  
just reverse the ll, and do the math.

public Node addOne(Node head) {  
 Node temp = head;  
 temp = reverseLL(temp);  
 Node revtemp = temp;  
 int carry = 1;  
 while(temp != null)  
 {  
 if(temp.data == 9 && carry == 1)  
 {  
 temp.data = 0;  
 carry = 1;  
 }  
 else if(carry == 1)  
 {  
 temp.data = temp.data+1;  
 carry = 0;  
 }  
 else  
 {  
 break;  
 }  
 temp = temp.next;  
 }  
 temp = reverseLL(revtemp);  
 if(carry == 1)  
 {  
 Node ans = new Node(1);  
 ans.next = temp;  
 return ans;  
 }  
 return temp;  
}  
public Node reverseLL(Node head)  
{  
 Node curr = head;  
 Node prev = null;  
 Node next = null;  
 while(curr!=null)  
 {  
 next = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = next;  
 }  
 return prev;  
}

**Time Complexity : O(3n) Space Complexity : O(1)**

**Optimal:**

Using recursion:

public Node addOne(Node head) {  
 Node temp = head;  
 int carry = addone(temp);  
 if(carry == 1)  
 {  
 Node newHead = new Node(1);  
 newHead.next = head;  
 return newHead;  
 }  
 return head;  
}  
public int addone(Node temp)  
{  
 if(temp == null)return 1;  
 int carry = addone(temp.next);  
 temp.data = temp.data + carry;  
 if(temp.data < 10)return 0;  
 temp.data = 0;  
 return 1;  
}

**Time Complexity : O(2n) Space Complexity : O(n) – stack space.**

### ****Add 2 numbers in LL:****

**Problem Statement:** Given the heads of two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

**Examples:**

Input Format:

num1 = 243, num2 = 564

l1 = [2,4,3] l2 = [5,6,4]

Result: sum = 807; L = [7,0,8]

Explanation: Since the digits are stored in reverse order, reverse the numbers first to get the or original number and then add them as → 342 + 465 = 807. Refer to the image below.

**Solution :**

Keep track of the carry using a variable and simulate digits-by-digits sum starting from the head of the list, which contains the least significant digit.

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  
 ListNode t1 = l1;  
 ListNode t2 = l2;  
 ListNode dummyNode = new ListNode(-1);  
 ListNode curr = dummyNode;  
 int sum = 0;  
 int carry = 0;  
 while(t1!=null || t2!=null)  
 {  
 sum = carry;  
 if(t1!=null)sum = sum+t1.val;  
 if(t2!=null)sum = sum+t2.val;  
 if(sum<10)  
 {  
 curr.next = new ListNode(sum);  
 carry = 0;  
 }  
 else  
 {  
 curr.next = new ListNode(sum%10);  
 carry = sum/10;  
 }  
 curr = curr.next;  
 if(t1!=null)t1=t1.next;  
 if(t2!=null)t2=t2.next;  
 }  
 if(carry == 1)  
 {  
 curr.next = new ListNode(1);  
 }  
 return dummyNode.next;  
}

**Time Complexity : O(n+m) Space Complexity : O(1)**

### ****Delete all occurrences of a key in DLL:****

You are given the **head\_ref** of a doubly Linked List and a **Key**. Your task is to **delete all occurrences** of the given key if it is present and return the new DLL.

**Example1:**

**Input:**

2<->2<->10<->8<->4<->2<->5<->2

2

**Output:**

10<->8<->4<->5

**Explanation:**

All Occurences of 2 have been deleted.

**Example2:**

**Input:**

9<->1<->3<->4<->5<->1<->8<->4

9

**Output:**

1<->3<->4<->5<->1<->8<->4

**Explanation:**

All Occurences of 9 have been deleted.

**Solution :**

static Node deleteAllOccurOfX(Node head, int x) {  
  
 while(head.data == x)head = head.next;  
 Node temp = head;  
 while(temp!=null)  
 {  
 if(temp.data == x)  
 {  
 *deleteNode*(temp);  
 }  
 temp = temp.next;  
 }  
 return head;  
}  
static void deleteNode(Node curr)  
{  
 Node prev = curr.prev;  
 Node next = curr.next;  
 if(prev!=null)prev.next = next;  
 if(next!=null)next.prev = prev;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Find pairs with given sum in DLL:****

Given a sorted doubly linked list of positive distinct elements, the task is to find pairs in a doubly-linked list whose sum is equal to given value **target**.

**Example 1:**

**Input:**

1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9

target = 7

**Output:** (1, 6), (2,5)

**Explanation:** We can see that there are two pairs

(1, 6) and (2,5) with sum 7.

**Example 2:**

**Input:**

1 <-> 5 <-> 6

target = 6

**Output:** (1,5)

**Explanation:** We can see that there is one pairs (1, 5) with sum 6.

**Solution :**

**Using two pointers.**

public static ArrayList<ArrayList<Integer>> findPairsWithGivenSum(int target, Node head) {  
 ArrayList<ArrayList<Integer>> ans = new ArrayList<>();  
 Node end = *findTail*(head);  
 Node start = head;  
 while(start.data<end.data)  
 {  
 if(start.data+end.data == target)  
 {  
 ans.add(new ArrayList<>(Arrays.asList(start.data,end.data)));  
 start = start.next;  
 end = end.prev;  
 }  
 else if(start.data+end.data>target)  
 {  
 end = end.prev;  
 }  
 else  
 {  
 start = start.next;  
 }  
 }  
 return ans;  
}  
public static Node findTail(Node head)  
{  
 Node temp = head;  
 while(temp.next != null)  
 temp = temp.next;  
 return temp;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Remove duplicates from sorted DLL:****

Given a doubly linked list of **n**nodes sorted by values, the task is to remove duplicate nodes present in the linked list.

**Example 1:**

**Input:**

n = 6

1<->1<->1<->2<->3<->4

**Output:**

1<->2<->3<->4

**Explanation:**

Only the first occurance of node with value 1 is

retained, rest nodes with value = 1 are deleted.

**Example 2:**

**Input:**

n = 7

1<->2<->2<->3<->3<->4<->4

**Output:**

1<->2<->3<->4

**Explanation:**

Only the first occurance of nodes with values 2,3 and 4 are

retained, rest repeating nodes are deleted.

**Solution :**

Traversing through the linked list , and checking with the previous element since the doubly linked list is sorted.

Node removeDuplicates(Node head){  
 if (head == null || head.next == null) return head;  
  
 Node prev = head;  
 Node curr = head.next;  
  
 while (curr != null) {  
 if (prev.data == curr.data) {  
 deleteNode(curr); // Remove duplicate node  
 } else {  
 prev = curr; // Move forward only if no deletion  
 }  
 curr = curr.next;  
 }  
 return head;  
}  
void deleteNode(Node curr)  
{  
 Node prev = curr.prev;  
 Node next = curr.next;  
 prev.next = next;  
 if(next != null)  
 next.prev = prev;  
}

**Time Complexity : O(n) Space Complexity : O(1)**

### ****Rotate a LL:****

Problem Statement: Given the head of a linked list, rotate the list to the right by k places.

Examples:

**Example 1:**

Input:

head = [1,2,3,4,5]

k = 2

Output:

head = [4,5,1,2,3]

**Example 2:**

Input:

head = [1,2,3]

k = 4

Output:

head = [3,1,2]

Let’s take an example.

head = [1,2,3,4,5] k = 2000000000

If we see a brute force approach, it will take O(5\*2000000000) which is not a good time complexity when we can optimize it.

We can see that for every k which is multiple of the length of the list, we get back the original list. Try to operate brute force on any linked list for k as a multiple of the length of the list.

This gives us a hint that for k greater than the length of the list, we have to rotate the list for k%length of the list. This reduces our time complexity.

Steps to the algorithm:-

Calculate the length of the list.

Connect the last node to the first node, converting it to a circular linked list.

Iterate to cut the link of the last node and start a node of k%length of the list rotated list.

Dry Run:

Let’s calculate the length of the list by iterating on it until it reaches null and increasing the count. Once the length is calculated we will connect the last node to the first node.

**Solution :**

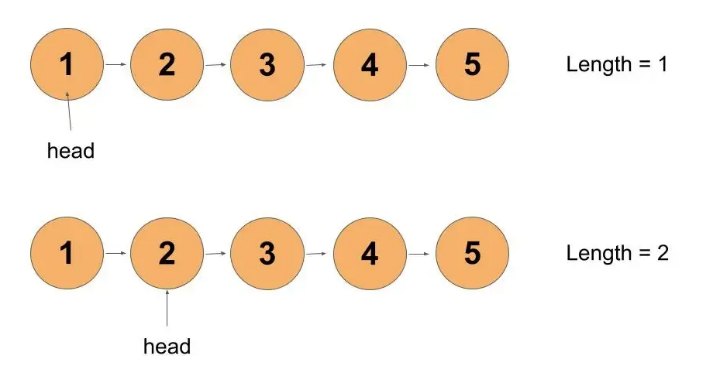
public ListNode rotateRight(ListNode head, int k) {  
 if(head == null || k == 0)return head;  
 ListNode tail = head;  
 int len = 1;  
 while(tail.next != null)  
 {  
 tail = tail.next;  
 len++;  
 }  
 if(k%len == 0)return head;  
 ListNode temp = findNthNode(head,len-(k%len));  
 tail.next = head;  
 head = temp.next;  
 temp.next = null;  
 return head;  
}  
public ListNode findNthNode(ListNode head,int nth)  
{  
 ListNode temp = head;  
 int cnt = 1;  
 while(cnt != nth)  
 {  
 temp = temp.next;  
 cnt++;  
 }  
 return temp;  
}

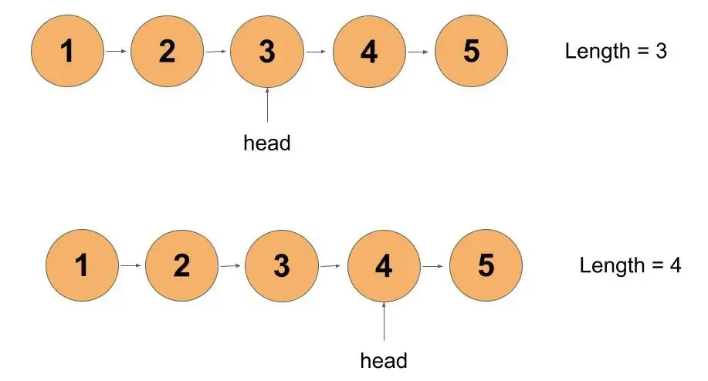
**Time Complexity:** O(length of list) + O(length of list - (length of list%k))

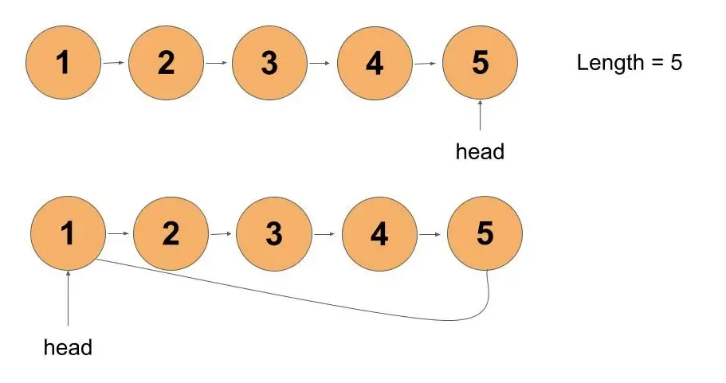
Reason: O(length of the list) for calculating the length of the list. O(length of the list - (length of list%k)) for breaking link.

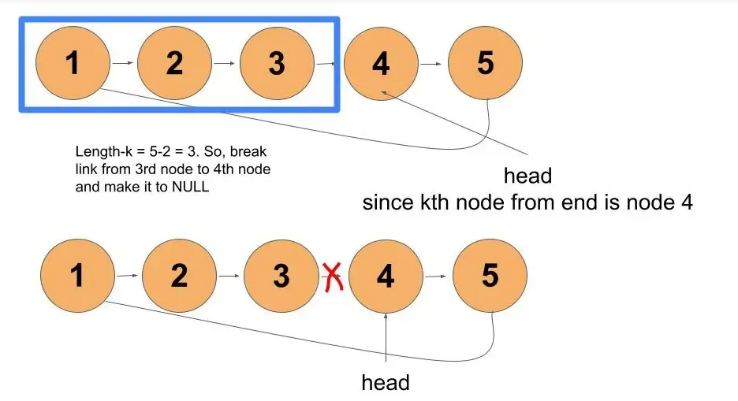
**Space Complexity:** O(1)

Reason: No extra data structure is used for computation.









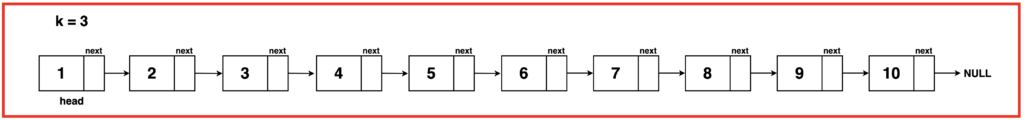
### ****Reverse LL in group of given size K:****

: Given the head of a singly linked list of `n` nodes and an integer `k`, where k is less than or equal to `n`. Your task is to reverse the order of each group of `k` consecutive nodes, if `n` is not divisible by `k`, then the last group of remaining nodes should remain unchanged.

Examples

**Example 1:**

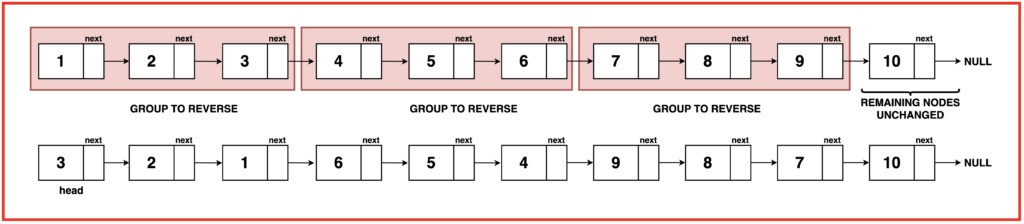
Input Format:



LL: 1 2 3 4 5 6 7 8 9 10

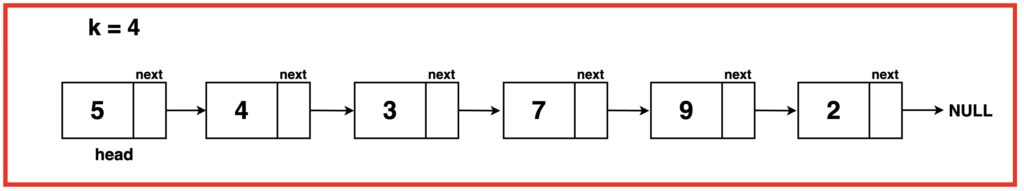
K = 3

Output: 3 2 1 6 5 4 9 8 7 10



**Example 2:**

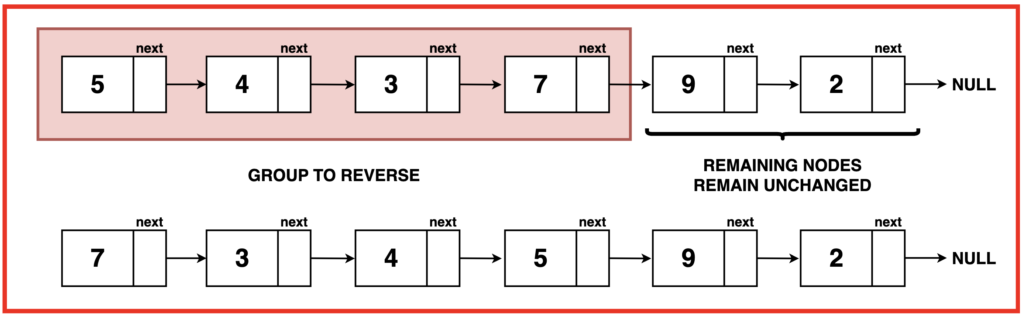
Input Format:



LL: 5 4 3 7 9 2

K = 4

Output: 7 3 4 5 9 2



**Solution :**

The approach simplifies reversing linked list nodes by breaking the list into segments of K nodes and reversing each segment individually. Starting from the head, the algorithm traverses the list to identify segments of K nodes. Upon finding a segment, it reverses it, returning the modified list. If a segment has less than K nodes left (ie. remaining nodes at the end), they are left unaltered.

To implement this (complex) algorithm we can break down the process into three parts:

`reverseLinkedList`: This function takes the head of a segment as input and reverses the linked list formed by that segment. It operates by utilizing the classic iterative 3-pointer method to reverse the direction of pointers within the segment. Read about this algorithm in detail here Reverse Linked List.

`getKthNode`: The purpose of this function is to identify the end of a segment of K nodes in the linked list. Given a starting node, it traverses K nodes in the list and returns the Kth node, allowing the segmentation of the list into smaller parts for reversal.

`kReverse`: The main function orchestrates the reversal process. It iterates through the linked list and identifies segments of K nodes using getKthNode. For each identified segment, it utilizes reverseLinkedList to reverse the nodes within that segment. This iterative approach efficiently reverses the linked list nodes in groups of K.

public ListNode reverseKGroup(ListNode head, int k) {  
 ListNode temp = head;  
 ListNode prev = null;  
 ListNode nextNode = null;  
 ListNode kthNode ;  
 while(temp!=null)  
 {  
 kthNode = findk(temp,k);  
 if(kthNode == null)  
 {  
 if(prev!=null)prev.next = temp;  
 break;  
 }  
 nextNode = kthNode.next;  
 kthNode.next = null;  
 reverse(temp);  
 if(temp == head)  
 {  
 head = kthNode;  
 }  
 else  
 {  
 prev.next = kthNode;  
 }  
 prev = temp;  
 temp = nextNode;  
 }  
 return head;  
}  
public ListNode findk(ListNode head,int k)  
{  
 ListNode temp = head;  
 k--;  
 while(temp!=null && k>0)  
 {  
 k--;  
 temp = temp.next;  
 }  
 return temp;  
}  
public void reverse(ListNode head)  
{  
 ListNode curr = head;  
 ListNode prev = null;  
 ListNode next = null;  
 while(curr != null)  
 {  
 next = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = next;  
 }  
}

**Time Complexity:** O(2N) The time complexity consists of actions of reversing segments of K and finding the Kth node which operates in linear time. Thus, O(N) + O(N) = O(2N), which simplifies to O(N).

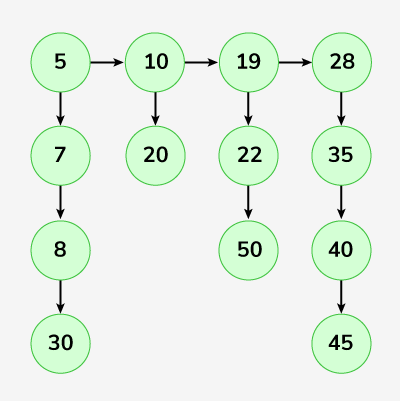
**Space Complexity:** O(1) The space complexity is O(1) as the algorithm operates in place without any additional space requirements.

### ****Flattening a LinkedList:****

Given a linked list containing **n** head nodes where every node in the linked list contains two pointers:  
(i) **next**points to the next node in the list.  
(ii) **bottom**pointer to a sub-linked list where the current node is the head.  
Each of the sub-linked lists nodes and the head nodes are sorted in **ascending**order based on their data.  
Your task is to **flatten**the linked list such that all the nodes appear in a single level while maintaining the sorted order.

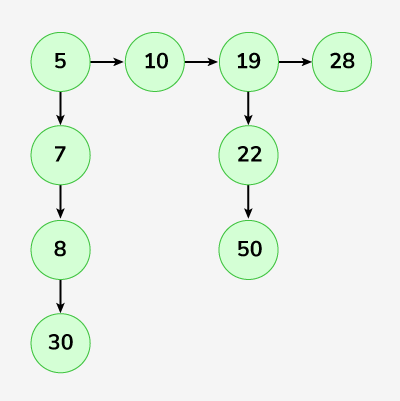
**Note:**1. **↓**represents the bottom pointer and **->** represents the next pointer.  
2. The flattened list will be printed using the **bottom** pointer instead of the next pointer.

**Examples:**

**Input:**  


**Output:** 5-> 7-> 8-> 10 -> 19-> 20-> 22-> 28-> 30-> 35-> 40-> 45-> 50.

**Explanation**:   
Bottom pointer of 5 is pointing to 7.  
Bottom pointer of 7 is pointing to 8.  
Bottom pointer of 8 is pointing to 10 and so on.

**Input:**  
   
**Output:** 5-> 7-> 8-> 10-> 19-> 22-> 28-> 30-> 50

**Explanation:**Bottom pointer of 5 is pointing to 7.  
Bottom pointer of 7 is pointing to 8.  
Bottom pointer of 8 is pointing to 10 and so on.

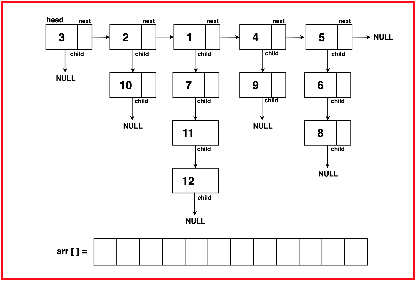
**Brute Force:**

To transform the given linked list into a single level sorted list ensuring that the nodes are arranged in an ascending order, we initialise an array to temporarily hold the extracted nodes during the traversal.

We iterate over the array by first going over the top-level next pointers of the linked list then accessing each node within its child pointers adding all to the array. Then the array is sorted to arrange all values sequentially and a new linked list from that array is created and returned.

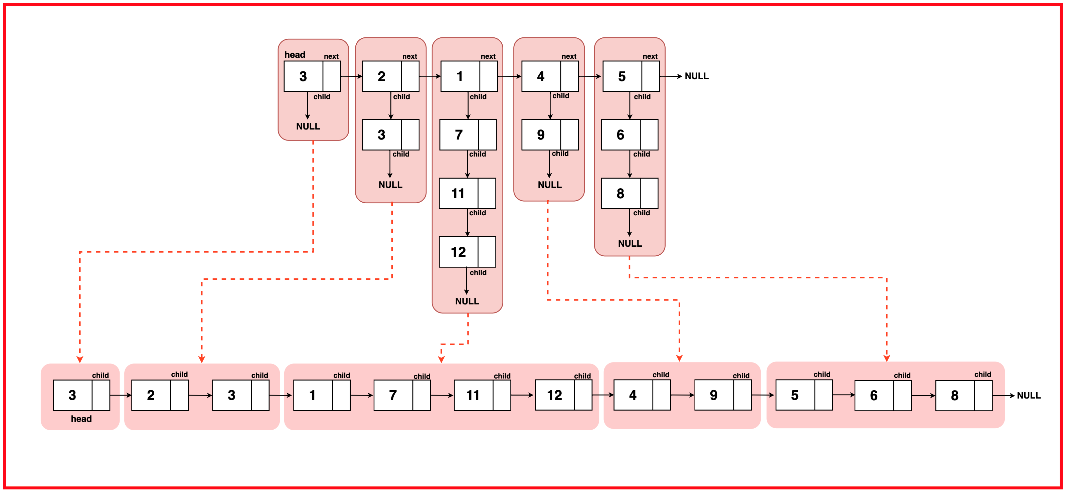
**Algorithm:**

**Step 1** :Initialise an empty array to store the data extracted during the traversal.

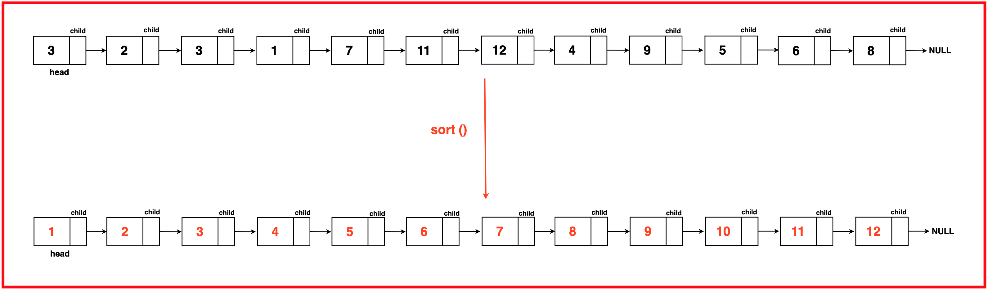


**Step 2:** Start traversing through the top-level ‘next’ pointers of the linked list and for each node accessed by the ‘next’ pointer, traverse its ‘child’ nodes.

Iterate all the nodes until reaching the end of the child pointer list appending each node’s value to the array. Move to the next primary node and repeat the process of traversing the child nodes.



**Step 3:** Sort the array to arrange its collected node data in ascending order.



**Step 4:** Create a new linked list from the sorted array and return the flattened linked list.

static Node convertArrToLinkedList(ArrayList<Integer> arr) {  
 // Create a dummy node to serve as  
 // the head of the linked list  
 Node dummyNode = new Node(-1);  
 Node temp = dummyNode;  
  
 // Iterate through the ArrayList and  
 // create nodes with elements  
 for (int i = 0; i < arr.size(); i++) {  
 // Create a new node with the element  
 temp.child = new Node(arr.get(i));  
 // Move the temporary pointer  
 // to the newly created node  
 temp = temp.child;  
 }  
 // Return the linked list starting  
 // from the next of the dummy node  
 return dummyNode.child;  
}  
  
// Function to flatten a linked list with child pointers  
static Node flattenLinkedList(Node head) {  
 ArrayList<Integer> arr = new ArrayList<>();  
  
 // Traverse through the linked list  
 while (head != null) {  
 // Traverse through the child  
 // nodes of each head node  
 Node t2 = head;  
 while (t2 != null) {  
 // Store each node's data in the ArrayList  
 arr.add(t2.data);  
 // Move to the next child node  
 t2 = t2.child;  
 }  
 // Move to the next head node  
 head = head.next;  
 }  
  
 // Sort the ArrayList containing  
 // node values in ascending order  
 Collections.sort(arr);  
  
 // Convert the sorted ArrayList  
 // back to a linked list  
 return *convertArrToLinkedList*(arr);  
}

**Time Complexity:** O(N\*M) + O(N\*M log(N\*M)) + O(N\*M)where N is the length of the linked list along the next pointer and M is the length of the linked list along the child pointer.

O(N\*M) as we traverse through all the elements, iterating through ‘N’ nodes along the next pointer and ‘M’ nodes along the child pointer.

O(N\*M log(N\*M)) as we sort the array containing N\*M (total) elements.

O(N\*M) as we reconstruct the linked list from the sorted array by iterating over the N\*M elements of the array.

**Space Complexity :** O(N\*M) + O(N\*M)where N is the length of the linked list along the next pointer and M is the length of the linked list along the child pointer.

O(N\*M) for storing all the elements in an additional array for sorting.

O(N\*M) to reconstruct the linked list from the array after sorting

**Optimal:**

The time and space complexity of the previous approach can be optimised as we have not yet leveraged the given property that the child linked lists are sorted. We can eliminate the additional space and time complexity generated by sorting by using these sorted vertical linked lists.

Instead of collecting all node values into an array and then sorting them, we can merge these pre-sorted lists directly during the traversal, eliminating the need for additional sorting steps. This merge operation can be performed efficiently in place without allocating extra space for the combined linked list.s

Read more about Merging Sorted Linked Lists. The base case ensures the termination of the recursion when there's either no list or only a single node remaining. The recursive function then handles the merging of the remaining lists after recursive flattening, creating a sorted flattened linked list.

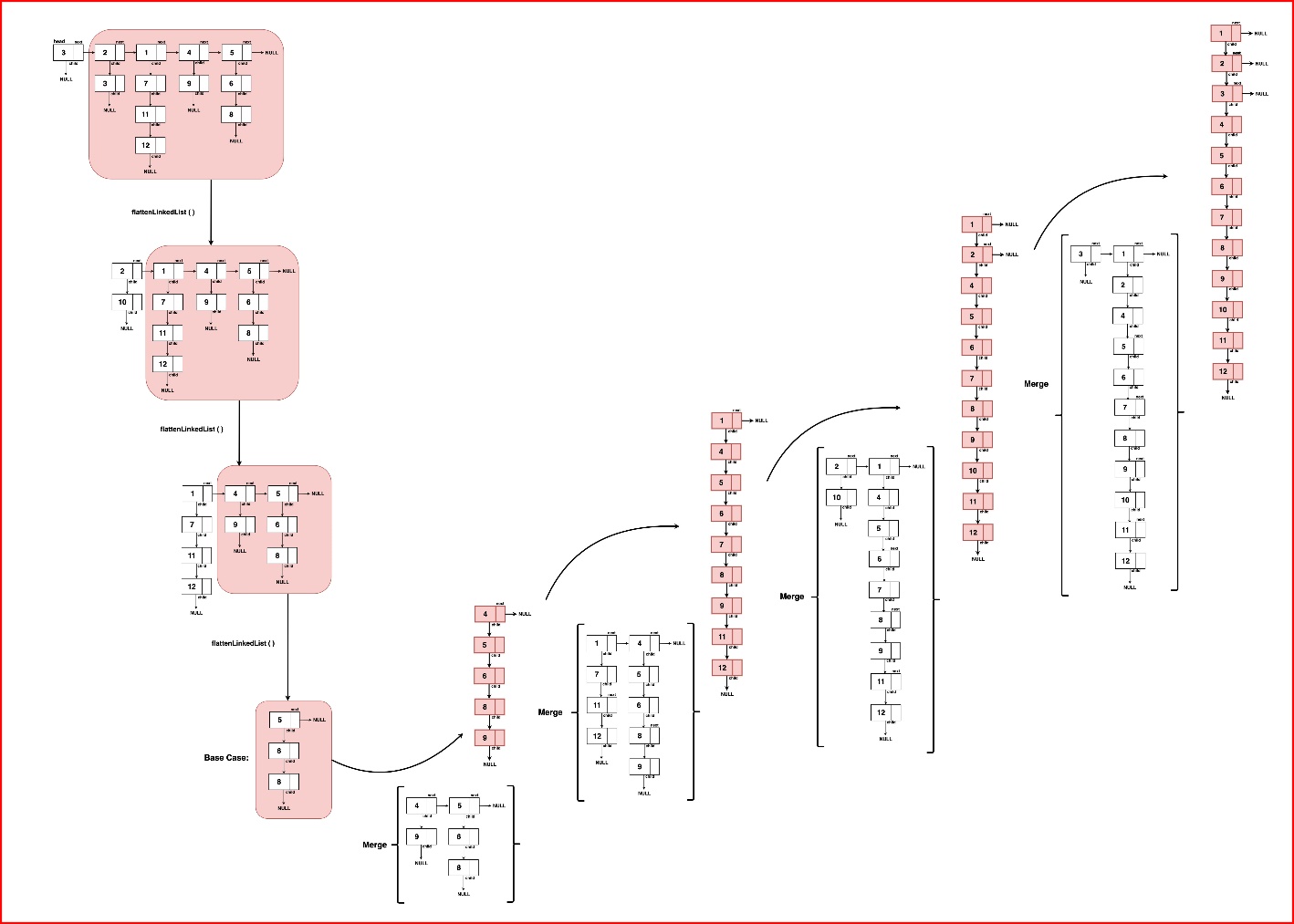
Node flatten(Node root) {  
 if(root.next == null)  
 {  
 return root;  
 }  
 Node mergedhead = flatten(root.next);  
 return merge(root,mergedhead);  
}  
  
Node merge(Node head,Node mergedhead)  
{  
 Node temp1 = head;  
 Node temp2 = mergedhead;  
 Node dummyNode = new Node(-1);  
 Node ans = dummyNode;  
 while(temp1!=null&&temp2!=null)  
 {  
 if(temp1.data <= temp2.data)  
 {  
 ans.bottom = temp1;  
 ans = ans.bottom;  
 temp1 = temp1.bottom;  
 }  
 else  
 {  
 ans.bottom = temp2;  
 ans = ans.bottom;  
 temp2 = temp2.bottom;  
 }  
 }  
 if(temp1!=null)ans.bottom=temp1;  
 else ans.bottom = temp2;  
 return dummyNode.bottom;  
}

**Time Complexity:** O( N\*(2M) ) ~ O(2 N\*M)where N is the length of the linked list along the next pointer and M is the length of the linked list along the child pointers.

The merge operation in each recursive call takes time complexity proportional to the length of the linked lists being merged as they have to iterate over the entire lists. Since the vertical depth of the linked lists is assume to be M, the time complexity for a single merge operation is proportional to O(2\*M).

This operation operation is performed N number of times (to each and every node along the next pointer list) hence the resultant time complexity becomes: O(N\* 2M).

**Space Complexity :** O(1) as this algorithm uses no external space or additional data structures to store values. But a recursive stack uses O(N) space to build the recursive calls for each node along the next pointer list.



### ****Clone LinkedList with random and next pointer:****

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a deep copy of the list. The deep copy should consist of exactly n brand new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return the head of the copied linked list.

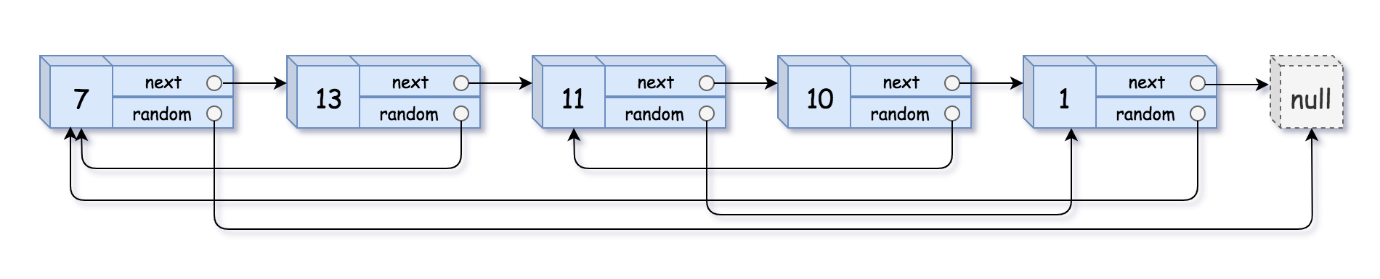
The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

val: an integer representing Node.val

random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will only be given the head of the original linked list.

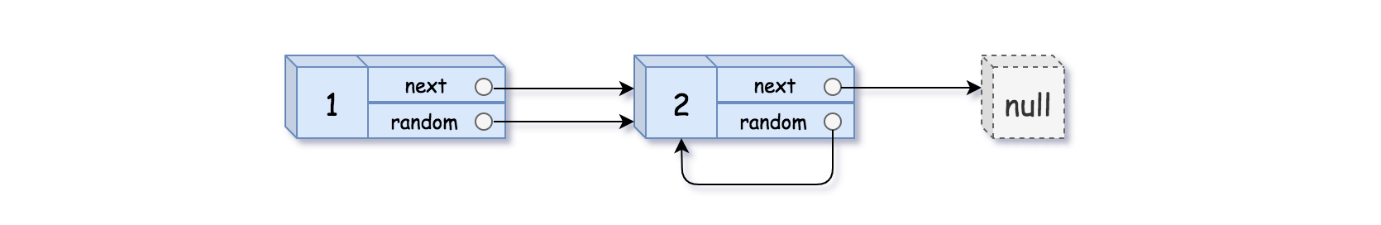
**Example 1:**



Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]

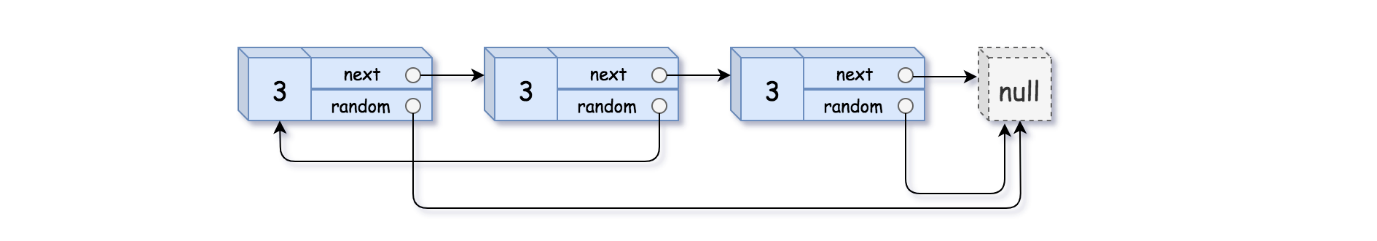
**Example 2:**



Input: head = [[1,1],[2,1]]

Output: [[1,1],[2,1]]

**Example 3:**



Input: head = [[3,null],[3,0],[3,null]]

Output: [[3,null],[3,0],[3,null]]

**Brute Force :**

To create a deep copy of the original linked list we can use a map to establish a relationship between original nodes and their copied nodes.

We traverse the list first to create a copied node for each original node then traverse and establish the correct connections between the copied nodes similar to the arrangement of next and random pointers of the original pointers. In the end, return the head of the copied list obtained from the map.

public Node copyRandomList(Node head) {  
 HashMap<Node,Node> map = new HashMap<>();  
 Node temp = head;  
 while(temp != null)  
 {  
 map.put(temp,new Node(temp.val));  
 temp = temp.next;  
 }  
 temp = head;  
 while(temp!=null)  
 {  
 Node currNode = map.get(temp);  
 currNode.next = map.get(temp.next);  
 currNode.random = map.get(temp.random);  
 temp = temp.next;  
 }  
 return map.get(head);  
}

**Time Complexity:** O(2N) where N is the number of nodes in the linked list. The linked list is traversed twice, once for creating copies of each node and for the second time to set the next and random pointers for each copied node. The time to access the nodes in the map is O(1) due to hashing.

**Space Complexity :** O(N)+O(N)where N is the number of nodes in the linked list as all nodes are stored in the map to maintain mappings and the copied linked lists takes O(N) space as well.

**Optimal:**

The previous approach uses an extra space complexity of creating mappings between the original and copied nodes. Instead of creating duplicate nodes and storing them in a map, insert it in between the original node and the next node for quick access without the need for additional space.

Traverse the list again to set the random pointer of copied nodes to the corresponding copied node duplicating the original arrangement. As a final traversal, separate the copied and original nodes by detaching alternate nodes.

public Node copyRandomList(Node head) {  
 Node temp = head;  
 Node copyNode = null;  
 Node dummyNode = new Node(-1);  
 Node res = dummyNode;  
 while(temp!=null)  
 {  
 Node nextNode = temp.next;  
 copyNode = new Node(temp.val);  
  
 temp.next = copyNode;  
 copyNode.next = nextNode;  
  
 temp = temp.next.next;  
 }  
 temp = head;  
 while(temp != null)  
 {  
 copyNode = temp.next;  
 if(temp.random != null)  
 {  
 copyNode.random = temp.random.next;  
 }  
 else  
 {  
 copyNode.random = null;  
 }  
 temp = temp.next.next;  
 }  
 temp = head;  
 while(temp != null)  
 {  
 res.next = temp.next;  
 temp.next = temp.next.next;  
  
 res = res.next;  
 temp = temp.next;  
 }  
 return dummyNode.next;  
}

**Time Complexity:** O(3N)

**Space Complexity :** O(N) where N is the number of nodes in the linked list as the only extra additional space allocated it to create the copied list without creating any other additional data structures.

### ****Sort LL:****

Merge Sort:

public ListNode sortList(ListNode head)  
{  
 if(head == null || head.next == null)return head;  
 ListNode midhead = findMiddle(head);  
 ListNode lefthead = head;  
 ListNode righthead = midhead.next;  
 midhead.next = null;  
 lefthead = sortList(lefthead);  
 righthead = sortList(righthead);  
 return merge(lefthead,righthead);  
}  
public ListNode merge(ListNode left1,ListNode right1)  
{  
 ListNode dummyNode = new ListNode(-1);  
 ListNode res = dummyNode;  
 while(left1!=null && right1!=null)  
 {  
 if(left1.val <= right1.val)  
 {  
 res.next = left1;  
 res = res.next;  
 left1 = left1.next;  
 }  
 else  
 {  
 res.next = right1;  
 res = res.next;  
 right1 = right1.next;  
 }  
 }  
 if(left1!=null)res.next=left1;  
 else res.next = right1;  
 return dummyNode.next;  
}  
public ListNode findMiddle(ListNode head)  
{  
 ListNode slow = head;  
 ListNode fast = head;  
 fast = fast.next.next;  
 while(fast!=null&&fast.next!=null)  
 {  
 slow = slow.next;  
 fast = fast.next.next;  
 }  
 return slow;  
}

**Time Complexity:** O(N log N)where N is the number of nodes in the linked list. Finding the middle node of the linked list requires traversing it linearly taking O(N) time complexity and to reach the individual nodes of the list, it has to be split log N times (continuously halve the list until we have individual elements).

**Space Complexity :** O(1) as no additional data structures or space is allocated for storage during the merging process. However, space proportional to O(log N) stack space is required for the recursive calls. THe maximum recursion depth of log N height is occupied on the call stack.