**🧩 1. Reverse Linked List**

**💡 Problem:**

We are given the **head** of a singly linked list (that is, the first node).  
We need to **reverse the list** — meaning the first node should become last, and the last node should become first.

**📘 Example:**

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

**Output:**  
[5 → 4 → 3 → 2 → 1]

**🔍 Step-by-step explanation:**

Let’s say the list is:

1 → 2 → 3 → 4 → 5 → null

We want to make it:

5 → 4 → 3 → 2 → 1 → null

We do this by **reversing each node’s pointer** one by one.

**⚙️ Logic:**

We keep track of:

* prev → previous node (starts as null)
* curr → current node (starts as head)

Steps:

1. Save the next node → next = curr.next
2. Reverse the pointer → curr.next = prev
3. Move forward →  
   prev = curr  
   curr = next
4. Continue until curr becomes null  
   Then, prev is the new head.

**💻 Java Code:**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public class ReverseLinkedList {

public static ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextNode = curr.next; // save next node

curr.next = prev; // reverse link

prev = curr; // move prev forward

curr = nextNode; // move curr forward

}

return prev; // new head

}

public static void main(String[] args) {

// Creating Linked List: 1 -> 2 -> 3 -> 4 -> 5

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

ListNode reversed = reverseList(head);

// Printing reversed list

while (reversed != null) {

System.out.print(reversed.val + " ");

reversed = reversed.next;

}

}

}

✅ **Output:**

5 4 3 2 1

**🔁 2. Linked List Cycle**

**💡 Problem:**

We are given the **head** of a linked list and we need to check if the list contains a **cycle**.

A **cycle** happens when a node’s next pointer points back to a previous node instead of null.

**📘 Example:**

**Input:**  
head = [3 → 2 → 0 → -4]  
and the tail connects to the node with index 1 (which is value 2).

That means:

3 → 2 → 0 → -4

↑ |

|\_\_\_\_\_\_\_\_|

**Output:**  
true  
(because the list goes in a loop)

**🔍 Step-by-step explanation:**

We can imagine two runners on the track:

* slow moves one step at a time
* fast moves two steps at a time

If there’s **no cycle**, fast will reach null.  
If there **is a cycle**, fast and slow will eventually meet.

**💻 Java Code:**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public class LinkedListCycle {

public static boolean hasCycle(ListNode head) {

if (head == null) return false;

ListNode slow = head;

ListNode fast = head;

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

slow = slow.next; // move 1 step

fast = fast.next.next; // move 2 steps

if (slow == fast) { // if they meet, cycle exists

return true;

}

}

return false; // if fast reaches null → no cycle

}

public static void main(String[] args) {

// Create Linked List: 3 -> 2 -> 0 -> -4

ListNode head = new ListNode(3);

head.next = new ListNode(2);

head.next.next = new ListNode(0);

head.next.next.next = new ListNode(-4);

// create cycle (tail connects to node 2)

head.next.next.next.next = head.next;

System.out.println(hasCycle(head)); // Output: true

}

}

✅ **Output:**

true

**Perfect 👍 Let’s now clearly explain both linked list problems — “Merge Two Sorted Lists” and “Merge k Sorted Lists” — in a very simple, easy-to-understand way with examples and Java code.**

**🧩 3. Merge Two Sorted Lists**

**💡 Problem:**

**You are given the heads of two sorted linked lists — list1 and list2.  
You must merge them into a single sorted linked list.**

**📘 Example:**

**Input:  
list1 = [1 → 2 → 4]  
list2 = [1 → 3 → 4]**

**Output:  
[1 → 1 → 2 → 3 → 4 → 4]**

**🔍 Step-by-step Explanation:**

**We compare the values from both lists one by one:**

| **Step** | **list1 value** | **list2 value** | **Smaller node added to result** |
| --- | --- | --- | --- |
| **1** | **1** | **1** | **1 (from list1)** |
| **2** | **2** | **1** | **1 (from list2)** |
| **3** | **2** | **3** | **2** |
| **4** | **4** | **3** | **3** |
| **5** | **4** | **4** | **4 (from list1)** |
| **6** | **null** | **4** | **4 (from list2)** |

**✅ Final merged list → 1 → 1 → 2 → 3 → 4 → 4**

**⚙️ Logic:**

1. **Use a dummy node to simplify merging.**
2. **Use a pointer tail that always points to the end of the merged list.**
3. **Compare the current nodes of list1 and list2.**
4. **Attach the smaller node to tail.**
5. **Move that list’s pointer forward.**
6. **Continue until both lists are done.**
7. **Return dummy.next (the head of the merged list).**

**💻 Java Code:**

**class ListNode {**

**int val;**

**ListNode next;**

**ListNode(int val) {**

**this.val = val;**

**}**

**}**

**public class MergeTwoSortedLists {**

**public static ListNode mergeTwoLists(ListNode list1, ListNode list2) {**

**ListNode dummy = new ListNode(0);**

**ListNode tail = dummy;**

**while (list1 != null && list2 != null) {**

**if (list1.val < list2.val) {**

**tail.next = list1;**

**list1 = list1.next;**

**} else {**

**tail.next = list2;**

**list2 = list2.next;**

**}**

**tail = tail.next;**

**}**

**// Add the remaining nodes from either list**

**if (list1 != null) tail.next = list1;**

**else tail.next = list2;**

**return dummy.next;**

**}**

**public static void main(String[] args) {**

**ListNode list1 = new ListNode(1);**

**list1.next = new ListNode(2);**

**list1.next.next = new ListNode(4);**

**ListNode list2 = new ListNode(1);**

**list2.next = new ListNode(3);**

**list2.next.next = new ListNode(4);**

**ListNode result = mergeTwoLists(list1, list2);**

**// Print merged list**

**while (result != null) {**

**System.out.print(result.val + " ");**

**result = result.next;**

**}**

**}**

**}**

**✅ Output:**

**1 1 2 3 4 4**

**🔁 4. Merge k Sorted Lists (Without Priority Queue)**

**💡 Problem:**

You’re given an array of k linked lists.  
Each linked list is **sorted in ascending order**.  
You need to **merge all of them into one sorted linked list**.

**📘 Example:**

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

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

**💭 Idea:**

We already know how to merge **two** sorted lists (from question 3).  
So, we can reuse that logic.

To merge **k** lists:

* Merge the first two lists → get a new sorted list.
* Then merge that result with the third list → get another sorted list.
* Continue this process until all lists are merged.

**🔍 Step-by-Step Explanation:**

Let’s take the input lists:

List1 = 1 → 4 → 5

List2 = 1 → 3 → 4

List3 = 2 → 6

**Step 1:**

Merge List1 and List2  
→ Result = [1,1,3,4,4,5]

**Step 2:**

Now merge that result with List3  
→ Final Result = [1,1,2,3,4,4,5,6]

**⚙️ Logic (in simple steps)**

1. Start with result = null
2. For each list in lists:
   * Merge result with current list using our “mergeTwoLists” function.
3. After the loop ends → result contains the fully merged sorted list.

**💻 Java Code (Without Priority Queue)**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public class MergeKSortedLists {

// Step 1: Merge two sorted lists

public static ListNode mergeTwoLists(ListNode list1, ListNode list2) {

ListNode dummy = new ListNode(0);

ListNode tail = dummy;

while (list1 != null && list2 != null) {

if (list1.val < list2.val) {

tail.next = list1;

list1 = list1.next;

} else {

tail.next = list2;

list2 = list2.next;

}

tail = tail.next;

}

if (list1 != null) tail.next = list1;

if (list2 != null) tail.next = list2;

return dummy.next;

}

// Step 2: Merge all k lists

public static ListNode mergeKLists(ListNode[] lists) {

if (lists == null || lists.length == 0) return null;

ListNode result = lists[0]; // start with first list

for (int i = 1; i < lists.length; i++) {

result = mergeTwoLists(result, lists[i]);

}

return result;

}

public static void main(String[] args) {

// List1: 1 -> 4 -> 5

ListNode l1 = new ListNode(1);

l1.next = new ListNode(4);

l1.next.next = new ListNode(5);

// List2: 1 -> 3 -> 4

ListNode l2 = new ListNode(1);

l2.next = new ListNode(3);

l2.next.next = new ListNode(4);

// List3: 2 -> 6

ListNode l3 = new ListNode(2);

l3.next = new ListNode(6);

ListNode[] lists = {l1, l2, l3};

ListNode result = mergeKLists(lists);

// Print merged list

while (result != null) {

System.out.print(result.val + " ");

result = result.next;

}

}

}

**✅ Output:**

1 1 2 3 4 4 5 6

**⚡ Time Complexity:**

* Each merge of two lists takes **O(n + m)**
* Doing it sequentially for k lists → **O(k × n)** approximately (if all lists are similar in size)

**⚙️ Space Complexity:**

* **O(1)** (only pointers are used; no extra data structures)

**🧩 5. Remove Nth Node From End of List**

**💡 Problem:**

You are given the head of a linked list.  
Remove the **n-th node from the end** of the list and return the new head.

**📘 Example:**

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

**🔍 Step-by-step Explanation:**

We want to remove the **2nd node from the end**, which is **node 4**.

1. Count nodes from the end:
2. 1 → 2 → 3 → 4 → 5
3. ↑
4. (2nd from end)
5. Remove node 4 → list becomes:
6. 1 → 2 → 3 → 5

**⚙️ Logic (Two Pointer Technique):**

We can do this in **one pass** using two pointers:

1. Create **two pointers**: first and second, both start at head.
2. Move first **n steps ahead**.
3. Then move both pointers **together** until first reaches null.
4. Now, second is **just before the node to remove**.
5. Skip that node → second.next = second.next.next.

**💻 Java Code:**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public class RemoveNthNode {

public static ListNode removeNthFromEnd(ListNode head, int n) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode first = dummy;

ListNode second = dummy;

// Move first n+1 steps ahead

for (int i = 0; i <= n; i++) {

first = first.next;

}

// Move both pointers until first reaches end

while (first != null) {

first = first.next;

second = second.next;

}

// Skip the nth node

second.next = second.next.next;

return dummy.next;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

ListNode result = removeNthFromEnd(head, 2);

while (result != null) {

System.out.print(result.val + " ");

result = result.next;

}

}

}

✅ **Output:**

1 2 3 5

**🔁 6. Reorder List**

**💡 Problem:**

You are given the head of a singly linked list.  
You must reorder it to follow the pattern:

L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → ...

You cannot change the node values — only the links (pointers).

**📘 Example:**

**Input:**  
head = [1 → 2 → 3 → 4]  
**Output:**  
[1 → 4 → 2 → 3]

**🔍 Step-by-step Explanation:**

**Original:**

1 → 2 → 3 → 4

We want to reorder as:

1 → 4 → 2 → 3

**⚙️ Logic (3 Steps):**

1. **Find the middle** of the list (slow & fast pointer).  
   → mid = 2 (for [1,2,3,4])
2. **Reverse the second half**
3. Second half: 3 → 4 → null
4. After reverse: 4 → 3 → null
5. **Merge both halves alternately**
6. First half: 1 → 2
7. Second half (reversed): 4 → 3
8. Merge them:
9. 1 → 4 → 2 → 3

**💻 Java Code:**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public class ReorderList {

public static void reorderList(ListNode head) {

if (head == null || head.next == null) return;

// Step 1: Find middle (slow-fast pointer)

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

// Step 2: Reverse second half

ListNode second = reverse(slow.next);

slow.next = null; // break link

// Step 3: Merge two halves

ListNode first = head;

while (second != null) {

ListNode temp1 = first.next;

ListNode temp2 = second.next;

first.next = second;

second.next = temp1;

first = temp1;

second = temp2;

}

}

private static ListNode reverse(ListNode head) {

ListNode prev = null, curr = head;

while (curr != null) {

ListNode nextNode = curr.next;

curr.next = prev;

prev = curr;

curr = nextNode;

}

return prev;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

reorderList(head);

while (head != null) {

System.out.print(head.val + " ");

head = head.next;

}

}

}

✅ **Output:**

1 4 2 3

QUESTION FROM **DSA Case Study1.txt**

**✅ Full Java Code: Doubly Linked List for Student Management**

// Student.java

class Student {

private int stuID;

private String name;

private int age;

private float percentage;

// Constructor

public Student(int stuID, String name, int age, float percentage) {

this.stuID = stuID;

this.name = name;

this.age = age;

this.percentage = percentage;

}

// Getters and Setters

public int getStuID() {

return stuID;

}

public void setStuID(int stuID) {

this.stuID = stuID;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public float getPercentage() {

return percentage;

}

public void setPercentage(float percentage) {

this.percentage = percentage;

}

@Override

public String toString() {

return "Student ID: " + stuID + ", Name: " + name + ", Age: " + age + ", Percentage: " + percentage;

}

}

// Node.java

class Node {

Student student;

Node next;

Node prev;

public Node(Student student) {

this.student = student;

this.next = null;

this.prev = null;

}

}

// DoublyLinkedList.java

class DoublyLinkedList {

private Node head;

private Node tail;

// Insert student at end

public void insertAtEnd(Student student) {

Node newNode = new Node(student);

if (head == null) {

head = tail = newNode;

return;

}

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

// Delete student by ID

public void deleteByID(int stuID) {

Node current = head;

while (current != null && current.student.getStuID() != stuID) {

current = current.next;

}

if (current == null) {

System.out.println("❌ Student with ID " + stuID + " not found.");

return;

}

// If it's the only node

if (head == tail) {

head = tail = null;

}

// If deleting head

else if (current == head) {

head = head.next;

head.prev = null;

}

// If deleting tail

else if (current == tail) {

tail = tail.prev;

tail.next = null;

}

// If deleting middle node

else {

current.prev.next = current.next;

current.next.prev = current.prev;

}

System.out.println("✅ Deleted Student ID: " + stuID);

}

// Search by ID

public void searchByID(int stuID) {

Node current = head;

while (current != null) {

if (current.student.getStuID() == stuID) {

System.out.println("✅ Found: " + current.student);

return;

}

current = current.next;

}

System.out.println("❌ Student with ID " + stuID + " not found.");

}

// Display from head to tail

public void displayForward() {

if (head == null) {

System.out.println("List is empty.");

return;

}

Node current = head;

System.out.println("📘 Students (Head → Tail):");

while (current != null) {

System.out.println(current.student);

current = current.next;

}

}

// Display from tail to head

public void displayBackward() {

if (tail == null) {

System.out.println("List is empty.");

return;

}

Node current = tail;

System.out.println("📕 Students (Tail → Head):");

while (current != null) {

System.out.println(current.student);

current = current.prev;

}

}

// Sort by percentage (Bubble Sort style)

public void sortByPercentage() {

if (head == null) return;

boolean swapped;

do {

swapped = false;

Node current = head;

while (current.next != null) {

if (current.student.getPercentage() > current.next.student.getPercentage()) {

// Swap the student objects

Student temp = current.student;

current.student = current.next.student;

current.next.student = temp;

swapped = true;

}

current = current.next;

}

} while (swapped);

System.out.println("✅ Sorted Students by Percentage (Ascending):");

displayForward();

}

}

// Main.java

public class Main {

public static void main(String[] args) {

DoublyLinkedList list = new DoublyLinkedList();

// Insert students

list.insertAtEnd(new Student(101, "Alice", 20, 89.5f));

list.insertAtEnd(new Student(102, "Bob", 21, 76.8f));

list.insertAtEnd(new Student(103, "Charlie", 19, 92.3f));

list.insertAtEnd(new Student(104, "David", 22, 68.4f));

// Display

list.displayForward();

System.out.println();

// Search

list.searchByID(102);

list.searchByID(999);

System.out.println();

// Delete

list.deleteByID(104);

list.displayForward();

System.out.println();

// Display in reverse

list.displayBackward();

System.out.println();

// Sort by percentage

list.sortByPercentage();

}

}

**🧠 Output Example:**

📘 Students (Head → Tail):

Student ID: 101, Name: Alice, Age: 20, Percentage: 89.5

Student ID: 102, Name: Bob, Age: 21, Percentage: 76.8

Student ID: 103, Name: Charlie, Age: 19, Percentage: 92.3

Student ID: 104, Name: David, Age: 22, Percentage: 68.4

✅ Found: Student ID: 102, Name: Bob, Age: 21, Percentage: 76.8

❌ Student with ID 999 not found.

✅ Deleted Student ID: 104

📘 Students (Head → Tail):

Student ID: 101, Name: Alice, Age: 20, Percentage: 89.5

Student ID: 102, Name: Bob, Age: 21, Percentage: 76.8

Student ID: 103, Name: Charlie, Age: 19, Percentage: 92.3

📕 Students (Tail → Head):

Student ID: 103, Name: Charlie, Age: 19, Percentage: 92.3

Student ID: 102, Name: Bob, Age: 21, Percentage: 76.8

Student ID: 101, Name: Alice, Age: 20, Percentage: 89.5

✅ Sorted Students by Percentage (Ascending):

Student ID: 102, Name: Bob, Age: 21, Percentage: 76.8

Student ID: 101, Name: Alice, Age: 20, Percentage: 89.5

Student ID: 103, Name: Charlie, Age: 19, Percentage: 92.3