Untitled

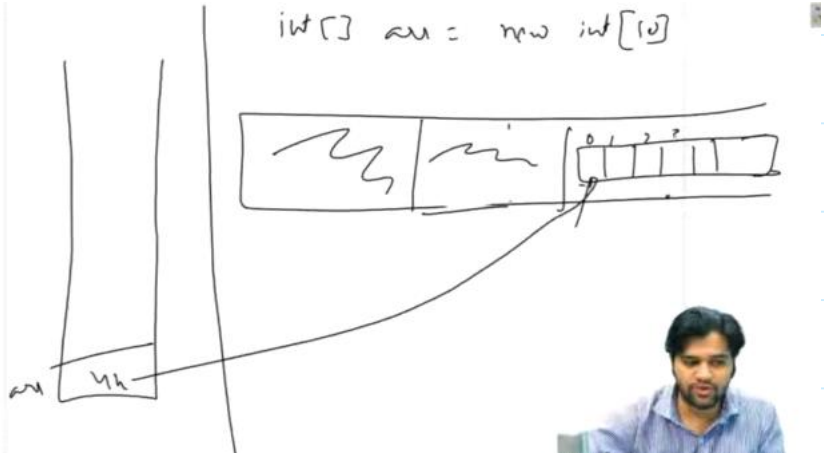
25 October 2025 18:49

Introduction.

Compare array vs linkedlist.

Array has continuous storage.

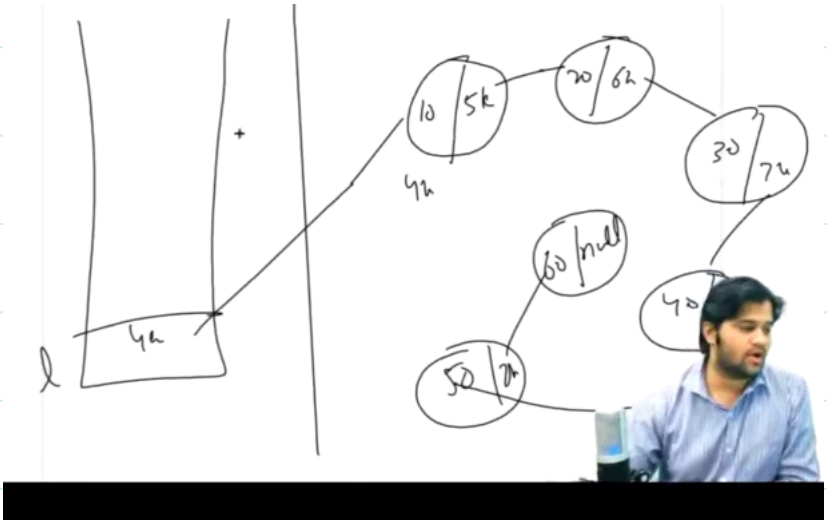
Data stored in heap, and its address is stored in stack.



So we need continuous (10 * 4) = 40 40 bytes available.

But in case of linkedlist, we could do even if 40 bytes of space is not available, but there is scattered space available.

As linkedlist is made of nodes.
And each node has data and address of next node.



So when there is fragmented memory available, then linkedlist can be formed and can make use of it.

But space taken by LL nodes would be more as it stores both data and address.

25 October 2025 19:02

How to make linkedlist.

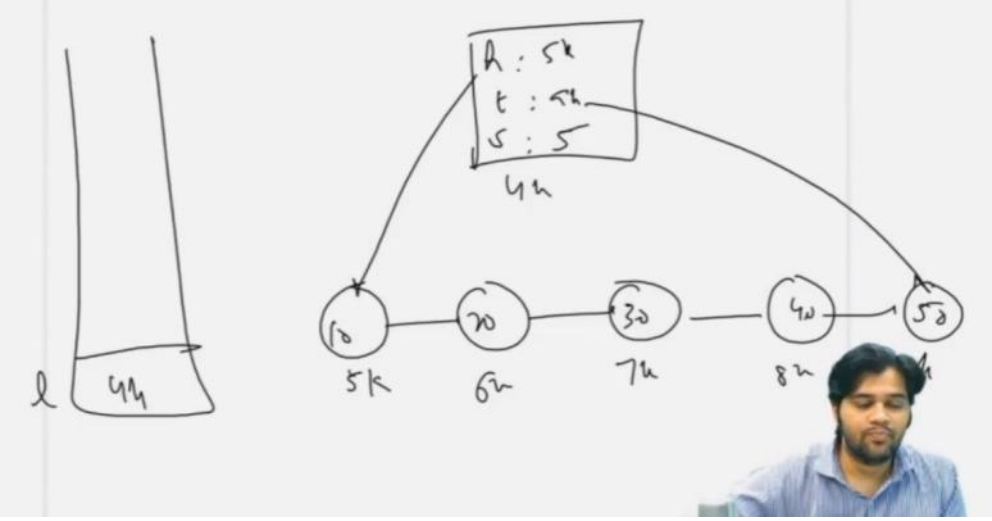
```java
import java.util.*;

public class Main {
    public static class Node {
        int data;
        Node next;
    }

    public static void main(String[] args) {

    }

}
```

Linked List Structure

The implementation uses two nested classes:

- `Node` : Represents an individual element in the list, containing an `int data` value and a `Node next` reference (a pointer) to the subsequent node.
- `LinkedList` : Represents the entire list, managing the `head` (the first node), the `tail` (the last node), and the `size` (the total count of nodes).

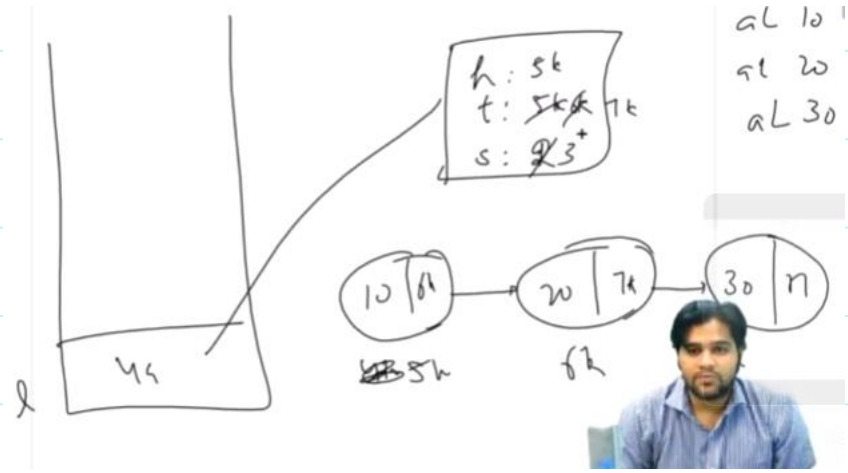
```java
public class App {

    public static class Node {
        int data;
        Node next;
    }

    public static class LinkedList {
        Node head;
        Node tail;
        int size;
    }
    Run | Debug
```

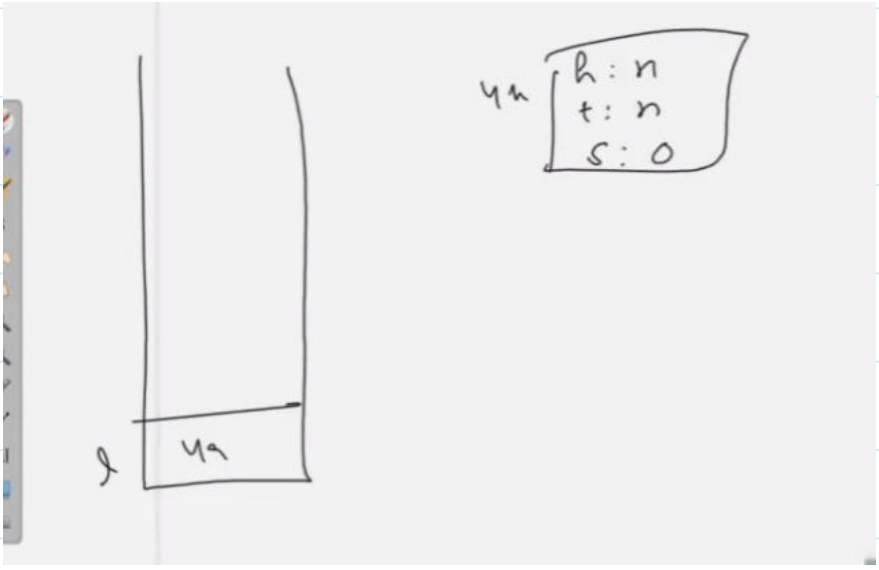


Always use this code to write anything on linkedlist
https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/Main.java

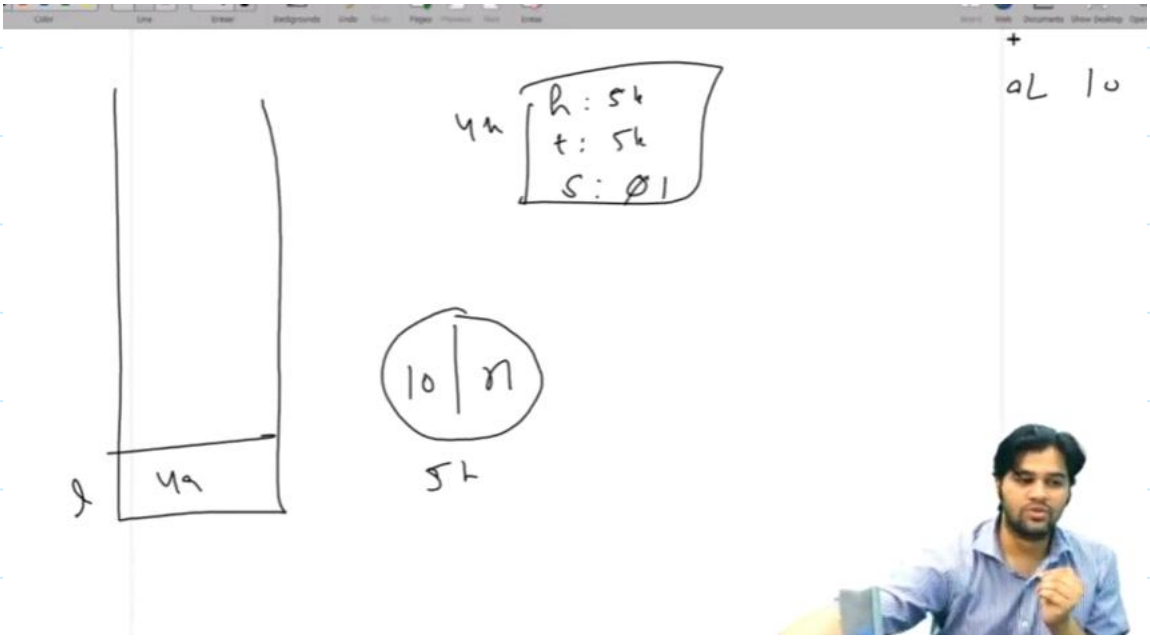
Add last in the linkedlist.

25 October 2025 19:31





After addlast 10



Logic of the `addLast(int val)` Function

Input

The integer value (`val`) to be added to the list.

Steps

1. **Create New Node**:
   - A new `Node` object is created to hold the input `val` .
2. **Handle Empty List (Initialization Case)**:
   - **Check**: If the list's `size` is **0** (i.e., `head` is `null`).
   - **Action**: This new node is the very first node. Therefore, both the list's `head` and the list's `tail` pointers are set to reference this new node.
3. **Handle Non-Empty List (Append Case)**:
   - **Check**: If the list's `size` is greater than 0.
   - **Action 1 (Link)**: The `next` pointer of the **current** `tail` node (`tail.next`) is updated to point to the newly created node. This links the new node to the end of the existing list.
   - **Action 2 (Update Tail)**: The list's main `tail` pointer is then updated to reference the new node, officially making it the last element.
4. **Update Size**:
   - Regardless of whether the list was empty or not, the list's `size` is incremented by 1.

In essence, the function efficiently handles both list creation and appending by leveraging the readily available `tail` pointer.

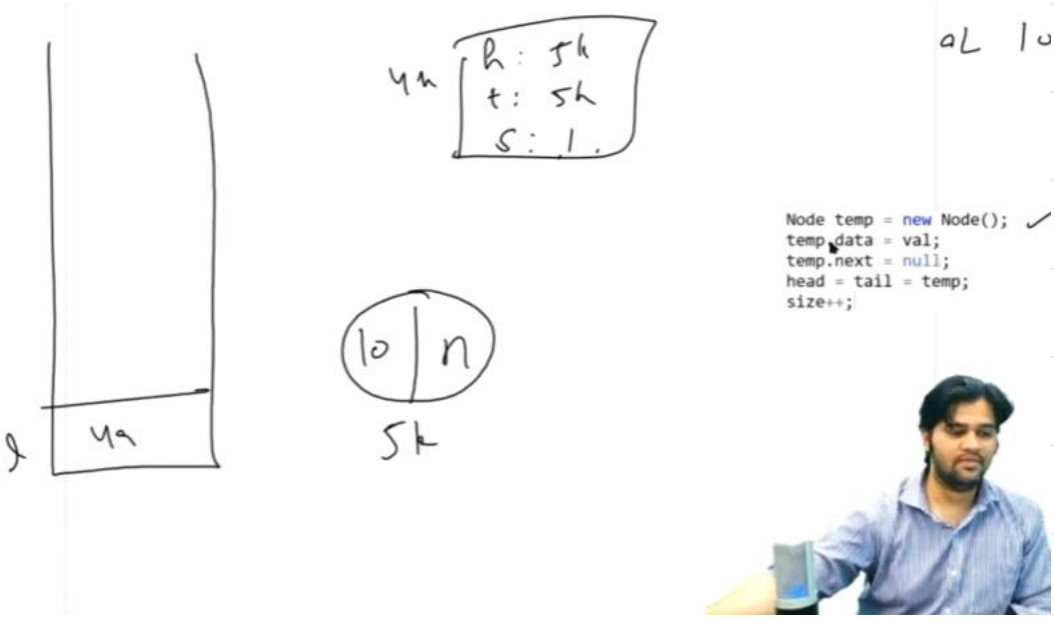
This will happen.

First a node is created, data is set, next address is null.

Then address of node updated in head and tail
And size updated to 1.

```
void addLast(int val) {
  // Write your code here
  if(size == 0){
      Node temp = new Node();
      temp.data = val;
      temp.next = null;
      head = tail = temp;
      size++;
  }
 }
}
```

Then run code side by side



```
void addLast(int val) {
  // Write your code here
  if(size == 0){
      Node temp = new Node();
      temp.data = val;
      temp.next = null;

      head = tail = temp;
      size++;
  } else {
      Node temp = new Node();
      temp.data = val;
      temp.next = null;

      tail.next = temp;
      tail = temp;

      size++;
  }
 }
}
```

Simplify the code.

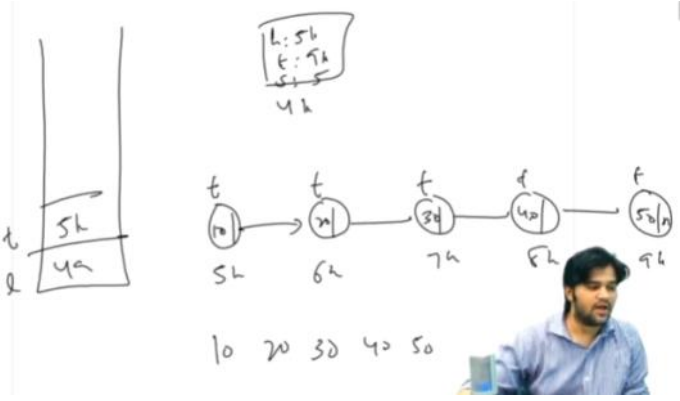
```
15    void addLast(int val) {
16      Node temp = new Node();
17      temp.data = val;
18      temp.next = null;
19
20      if(size == 0){
21          head = tail = temp;
22      } else {
23          tail.next = temp;
24          tail = temp;
25      }
26
27      size++;
28    }
29  }
30
```

Use this template to write code

Untitled

25 October 2025 21:48

Display a linkedlist;



💻 `display()` Function Logic Summary

The `display()` method is designed to traverse the list and print all node data values, separated by arrows, without a trailing arrow after the last element.

1. **Initialization**: A temporary pointer, `temp`, is set to the `head` to start iteration.
2. **Empty Check**: If `size` is 0, it prints `"List is empty"`.
3. **Traversal**: It enters a `while` loop that continues as long as `temp` is not `null` (meaning it hasn't passed the last node).
4. **Printing**:
   - It prints the `data` of the current node (`temp.data`).
   - It then checks if `temp.next` is **not null**. If it isn't, it prints the separator `"->"`. This crucial check ensures the arrow is **not** printed after the last node.
5. **Advance**: It moves the `temp` pointer to the next node (`temp = temp.next`).
6. **Newline**: After the loop finishes, a newline is printed to format the output.

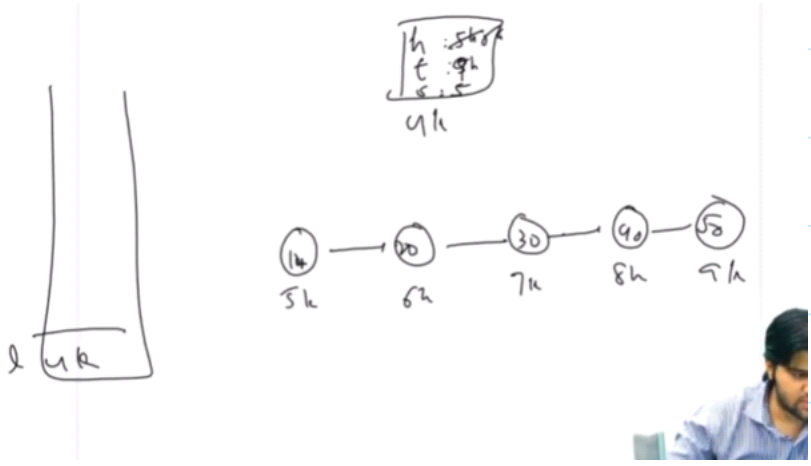
Java

```
        // Loop as long as the current node (temp) is not null.
        // This ensures the loop processes the last node as well.
        while (temp != null) {
            System.out.print(temp.data);

            // Print a separator if it's not the last element
            if (temp.next != null) {
                System.out.print(" -> ");
            }

            // Move to the next node
            temp = temp.next;
        }

        // Print a newline after displaying all elements
        System.out.println();
    }
```

Use this template to practice

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/addLastTemplate.java

Remove first from linkedlist.



⚙️ Logic of the `removeFirst()` Function

Input

None (it operates directly on the `LinkedList` object).

Output

The linked list is modified, and the `size` is decreased.

Steps

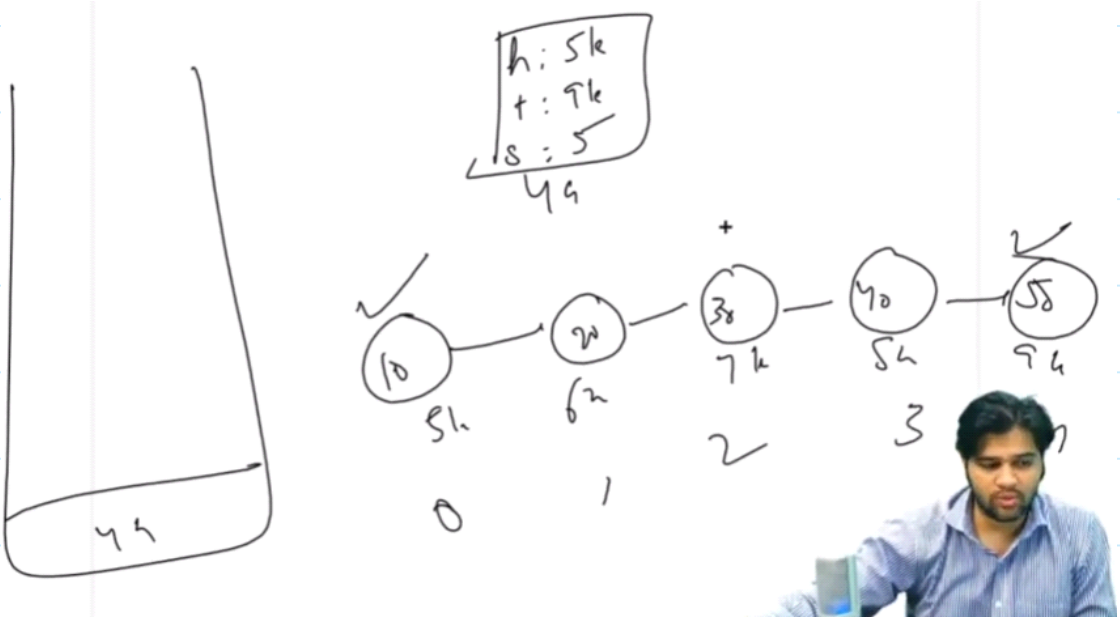
1. **Case 1: Empty List (Size = 0)**:
   - **Check**: If the list's `size` is equal to **0**.
   - **Action**: Print the error message `"List is empty"` and the function terminates.
2. **Case 2: Single Node List (Size = 1)**:
   - **Check**: If the list's `size` is equal to **1**.
   - **Action**:
     - Both the `head` and the `tail` pointers are set to `null`. This ensures the list is properly reset to an empty state.
     - The `size` is then **decremented** by 1 (it becomes 0).
3. **Case 3: Multiple Node List (Size > 1)**:
   - **Action**:
     - The `head` pointer is updated to point to the **next node** in the sequence (`head = head.next`).
     - *Note: The old head node is automatically garbage collected by Java since no references are pointing to it anymore.*
     - The `size` is then **decremented** by 1.

Change head and decrease size

Handle special cases, if size is 0
Or size is 1

```
    }

    public void removeFirst(){
      if(size == 0){
          System.out.println("List is empty");
      } else if(size == 1){
          head = tail = null;
          size = 0;
      } else {
          head = head.next;
          size--;
      }
    }
}
```

Get value in linkedlist.



0 based indexing

If size =0 , then list empty
If idx is invalid then invalid arg
Else return the data.

💻 Algorithm for `getAt(int idx)`

The primary goal of this algorithm is to **return the data element** stored in the node at the given **index (`idx`)** of the linked list, while also handling **error conditions**.

1. Check for an Empty List (Edge Case)

1. **Start:** Begin the `getAt(idx)` operation.
2. **Condition Check:** Check if the size of the list (`size`) is equal to **0**.
3. **Action if Empty:** If it is empty:
   - Print the message: "List is empty".
   - **Return -1** (or throw an exception in a production environment).

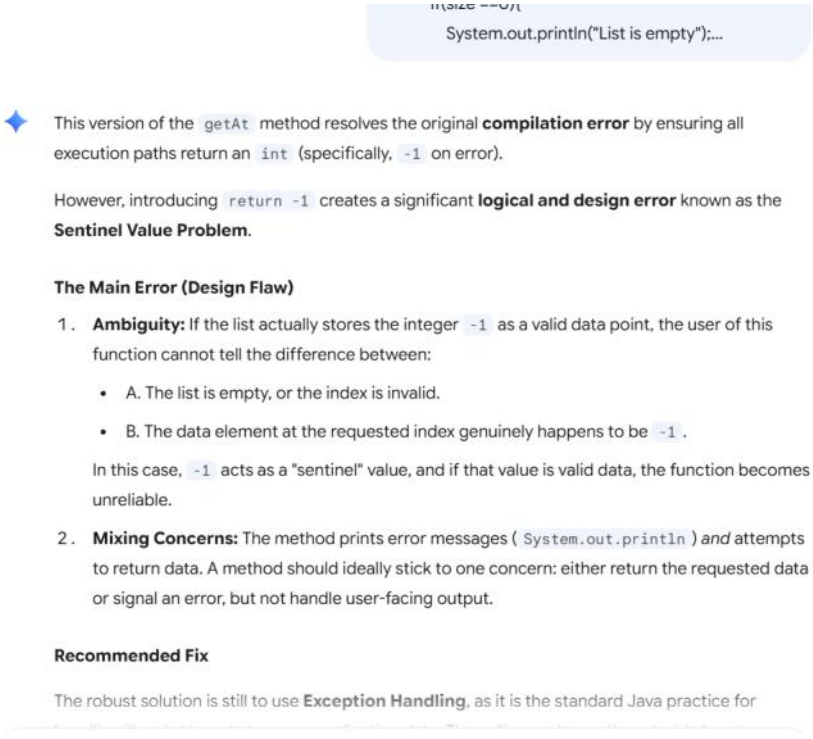
2. Check for an Invalid Index (Edge Case)

1. **Condition Check:** If the list is not empty, check if the provided index (`idx`) is **out of bounds**.
   - An index is invalid if $idx < 0$ **OR** $idx \geq size$.
2. **Action if Invalid:** If the index is invalid:
   - Print the message: "Invalid argument".
   - **Return -1**.

3. Traverse to the Target Index (Main Logic)

1. **Initialization:** If the index is valid, initialize a temporary reference, let's call it `temp` , and set it to the `head` of the linked list.
2. **Iteration (Traverse):** Use a loop to move the `temp` reference forward.
   - The loop should run idx times (i.e., from $i = 0$ up to $i < idx$).
   - In each iteration, update `temp` to point to the **next node**: `temp = temp.next;` .
   - **Goal:** After the loop completes, the `temp` reference will be pointing directly to the node at the desired index.
3. **Return Value:** Access the **data** field of the node currently pointed to by `temp` .
4. **End: Return** the value `temp.data` .

```java
public int getAt(int idx){
    if(size==0){
        System.out.println(x: "List is empty");
        return -1;
    } else if(idx<0 || idx>=size){
        System.out.println(x: "Invalid argument");
        return -1;
    } else{
        Node temp= head;
        for(int i=0; i<idx; i++){
            temp =temp.next;
        }
        return temp.data;
    }
}
```



https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/getAtTemplate.java

https://github.com/AlgoMagnet/0or1/blob/main/Linked

List/getAt.java

Untitled

25 October 2025 22:03

Add first in linkedlist

The problem your code solves can be framed as:

> **"Implement a method, `addFirst(int val)`, to insert a new node containing `val` at the head of the list. This operation must be performed in constant time, $O(1)$."**

---

2. Steps to Write the Code Next Time

Here are the conceptual steps for inserting a node at the beginning of a singly linked list. This process is highly predictable and should always be done in this order:

Step	Goal (What to Do)	Code Equivalent (Your Logic)
1. Creation	Create the new node (`temp`) and set its data.	`Node temp = new Node(); temp.data = val;`
2. Link Forward (CRITICAL)	Make the new node point to the current list. This is done by setting the new node's `next` pointer to the current `head` . **This must happen first!**	`temp.next = head;`
3. Update Head	Update the list's official `head` pointer to point to the new node.	`head = temp;`
4. Edge Case: Empty List	Check if the list was empty (`size == 0`) before the insertion. If so, the `tail` also needs to be updated to point to the new node (`temp`).	`if(size == 0){ tail = temp; }`
5. Size Update	Increment the list size.	`size++;`

```java
public void addFirst(int val){
    Node temp = new Node();
    temp.data = val;
    temp.next = null;

    if(size == 0){
        head =tail =temp;

    }else{
        temp.next = head;
        head =temp;

    }
    size++;
}
```

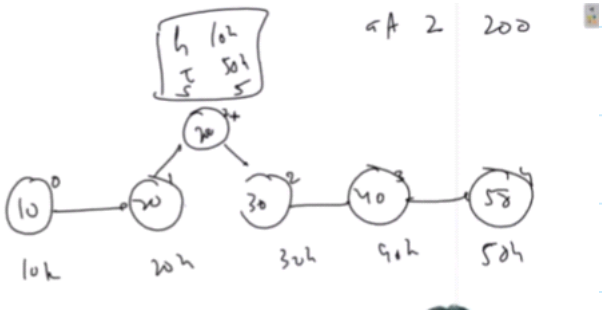
https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/addFirstTemplate.java

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/addFirst.java

Untitled

25 October 2025 22:05

Add at index in linkedlist



What is add at 2?

1. The Frame Question

> **"Implement a method, `addAt(int idx, int val)`, that inserts a new node containing `val` at the specified zero-based index `idx` in the singly linked list. This method must handle all index bounds and delegate to `addFirst` or `addLast` when appropriate, while also throwing an `IndexOutOfBoundsException` for invalid indices."**

✚ Algorithm for `addAt(int idx, int val)`

The primary goal is to insert a new node at the position specified by `idx`, handling boundary conditions (start, end) and invalid indices.

---

1. Handle Boundary and Invalid Cases (Edge Cases)

1. **Start of List (Index 0):**
   - **Condition:** If $idx == 0$.
   - **Action:** Call the existing `addFirst(val)` method.
   - **Return.**
2. **End of List (Index `size`):**
   - **Condition:** If $idx == size$.
   - **Action:** Call the existing `addLast(val)` method.
   - **Return.**
3. **Invalid Index:**
   - **Condition:** If $idx < 0$ **OR** $idx > size$.
   - **Action:** Print "Invalid index".
   - **Return.**

---

2. Conceptual Steps for Implementation

The complexity of `addAt` comes from handling the different index cases, which determine whether you can use an existing method or if you need to traverse the list.

Case	Condition	Action	Why?
Case 1: Invalid Index	`idx < 0` or `idx > size`	Throw `IndexOutOfBoundsException`.	The list cannot be inserted before index 0 or after the current `size` (i.e., the new element becomes the new `size`).
Case 2: At Head	`idx == 0`	Call `addFirst(val)`.	This is the most efficient way to handle the head insertion, which is already implemented.
Case 3: At Tail	`idx == size`	Call `addLast(val)`.	This is the most efficient way to handle appending, which is already implemented.
Case 4: In Middle	`0 < idx < size`	**Traverse** to the node *before* the insertion point, and update the links.	You need to find the node at `idx - 1` to correctly redirect its `next` pointer.

Export to Sheets

3. Step-by-Step Guide for Case 4 (In the Middle)

This is the only section of the code you will have to write from scratch, as the other cases can use the pre-existing methods.

Goal: Insert the new node at the position `idx`.

1. **Find the Predecessor:** You need to stop one step *before* the insertion point, at index `idx - 1`.
   - Initialize a temporary pointer: `Node prev = head;`
   - Loop `idx - 1` times: `for (int i = 0; i < idx - 1; i++) { prev = prev.next; }`
2. **Create the New Node:**
   - `Node temp = new Node();`
   - `temp.data = val;`
3. **Link the New Node Forward:** The new node's `next` pointer should point to the node that was *originally* at the insertion point (`idx`). This node is currently pointed to by `prev.next`.
   - `temp.next = prev.next;`
4. **Link the Predecessor Forward:** The predecessor node (`prev`) must now point to the new node (`temp`).
   - `prev.next = temp;`
5. **Update Size:**
   - `size++;`

```java
public void addAt(int idx, int val){
    if(idx==0){
        addFirst(val);
    } else if(idx == size){
        addLast(val);
    } else if(idx<0 || idx>size){
        System.out.println(x: "Invalid index");
    } else{
        Node temp = new Node();
        temp.data =val;

        Node traverse= head;
        for(int i=0;i<idx-1; i++){
            traverse = traverse.next;
        }
        temp.next = traverse.next;
        traverse.next =temp;

        size++;
    }
}
```

Make use of already existing function.

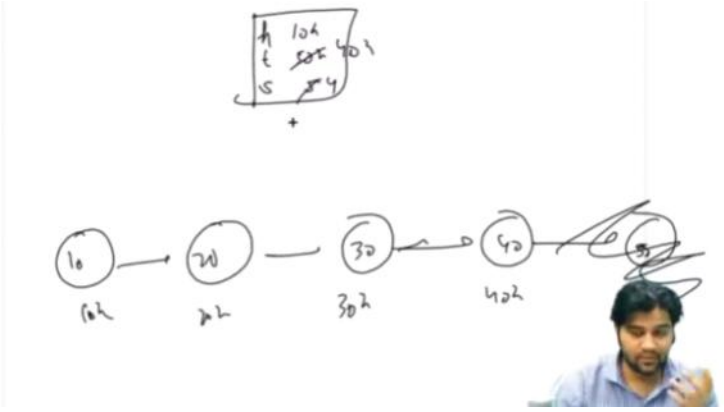
https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/addAtTemplate.java

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/addAt.java

Untitled

25 October 2025 22:07

Remove last in linked list.



Handle special case when size is 0 and 1.

1. The Frame Question

> **"Implement a method, `removeLast()`, to delete the last node (tail) of the list. This method must handle the empty list and single-node list edge cases, and ensure the `head`, `tail`, and `size` are correctly updated. What is the time complexity of this operation?"**

(Answer: The time complexity is $O(N)$, because a singly linked list must traverse from the head to find the second-to-last node.)

2. Steps to Write the Code Next Time

The `removeLast` method has three distinct cases that must be handled sequentially.

Case	Condition	Goal (What to Do)	Code Logic (Conceptual)
Case 1: Empty List	`size == 0`	Signal that the operation cannot be performed.	`if (size == 0) { // print error or throw exception }`
Case 2: Single Node	`size == 1`	Remove the only node by resetting the list pointers.	`else if (size == 1) { head = tail = null; size = 0; }`
Case 3: General Case	`size > 1`	Find the node *before* the tail, declare it the new tail, and break the link to the old tail.	*(See detailed steps below)*

Export to Sheets

Detailed Steps for Case 3 (Multiple Nodes)

This is the core logic that requires list traversal. The objective is to stop the temporary pointer (`temp`) at the node directly preceding the current `tail` .

1. **Initialize Traversal:** Start a temporary pointer at the head.
   - `Node temp = head;`
2. **Find the Second-to-Last Node:** Traverse until `temp` reaches the node at index `size - 2` .
   - If `size` is 5, you stop at index 3 (which is $5 - 2$).
   - `for (int i = 0; i < size - 2; i++) { temp = temp.next; }`
3. **Update Tail Reference:** The node `temp` is the new tail.
   - `tail = temp;`
4. **Break the Link (Deletion):** Set the new tail's `next` pointer to `null` . This effectively deletes the old tail, making it available for garbage collection. * `temp.next = null;`
5. **Update Size:**
   - `size--;`

```java
public void removeLast(){
    if(size==0){
        System.out.println(x: "List is empty");

    } else if(size ==1){
        head =tail =null;
        size--;

    } else {
        Node traverse = head;
        for(int i=0; i<size -2; i++){
            traverse = traverse.next;
        }
        tail = traverse;
        traverse.next=null;
        size--;

    }
}
```

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/removeLast.java

Untitled

25 October 2025 22:09

Reverse a linkedlist
Data iterative.

Only touch data property, don't touch next property

O(N)

Here is the algorithm for the two provided methods, which together implement **Data-Iterative Reversal** on a Singly Linked List.

The key idea of this method is to reverse the list by **swapping the *data* fields** of the nodes, rather than re-linking the *pointers* (which is the more common and efficient way).

⚙️ Algorithm for `getNodeAt(int idx)`

This is a helper method used to locate and return the node at a specific index.

1. **Initialization:** Initialize a traversal pointer, `traverse`, to the `head` of the linked list.
2. **Traverse:** Iterate in a loop from $i = 0$ up to $i < idx$.
3. **Update Pointer:** In each iteration, move the `traverse` pointer one step forward: `traverse = traverse.next;`.
4. **Return Node:** After the loop, the `traverse` pointer points to the node at the specified index. **Return** the `traverse` node.

🔄 Algorithm for `reverseDI()` (Data-Iterative Reversal)

This method reverses the order of the elements in the linked list by swapping data from the outside inward, similar to reversing an array.

1. **Initialization:**
   - Initialize a **left index pointer** `li` to **0** (start of the list).
   - Initialize a **right index pointer** `ri` to `size - 1` (end of the list).
2. **Iterative Swapping (Loop):**
   - Start a `while` **loop** that continues as long as the left index (`li`) is **less than** the right index (`ri`). This ensures we swap each pair exactly once and stop when the pointers meet or cross.
3. **Fetch Nodes:**
   - Inside the loop, use the helper method `getNodeAt(int idx)` to fetch the two nodes:
     - `Node left = getNodeAt(li);`
     - `Node right = getNodeAt(ri);`
4. **Swap Data:**
   - Swap the data values of the `left` node and the `right` node using a temporary variable:
     - Store the left node's data: `int temp = left.data;`
     - Assign the right node's data to the left node: `left.data = right.data;`
     - Assign the stored temporary data (original left data) to the right node: `right.data = temp;`
5. **Move Pointers:**
   - Move the indices toward the center:
     - Increment the left index: `li++;`
     - Decrement the right index: `ri--;`
6. **Loop Continues:** The loop repeats, swapping the next pair of nodes until the list is fully reversed.

Here's a breakdown of why and how `private` methods are used:

🔒 1. Encapsulation and Data Hiding

The core principle of OOP is **Encapsulation**, which means binding data (fields) and the methods that operate on that data into a single unit (the class) and restricting direct access from outside the class.

- **Access Control:** A `private` method can only be called from **within the same class**. This creates a secure boundary.

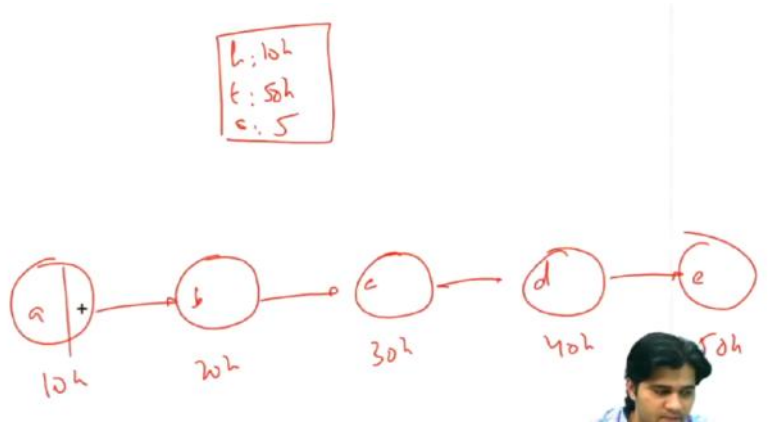
```java
private Node getNodeAt(int idx){
    Node traverse= head;
    for(int i=0; i<idx; i++){
        traverse= traverse.next;
    }
    return traverse;
}

public void reverseDI(){
    int li = 0;
    int ri = size -1;

    while(li< ri){
        Node left = getNodeAt(li);
        Node right = getNodeAt(ri);

        int temp = left.data;
        left.data = right.data;
        right.data = temp;

        li++;
        ri--;
    }
}
```

Untitled

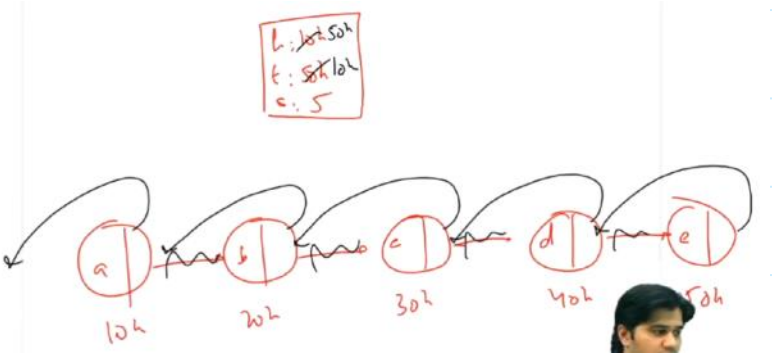
25 October 2025 22:12

Reverse a linkedlist

Pointer iterative.



From this achieve this



🔄 Pointer-Iterative Reversal Algorithm

The algorithm uses three key pointers— `prev`, `curr`, and `after_curr` —to redirect the `next` links of the nodes one by one.

1. Initialization

1. **Start:** Begin the operation.
2. `prev` **(New Tail Anchor):** Initialize a pointer `prev` to `null`. This pointer will track the part of the list that has already been reversed.

$$Node\ prev = null;$$

3. `curr` **(Current Node):** Initialize a pointer `curr` to the `head` of the list. This pointer is the node being actively processed in the loop.

$$Node\ curr = head;$$

---

2. Iterative Reversal Loop

The `while(curr != null)` loop ensures every node is visited and its pointer is reversed.

1. **Save the Next Node:** Create a temporary pointer, `after\_curr`, and set it to `curr.next`. This is a **crucial step** because we must save the reference to the rest of the list **before** we change `curr.next`.

$$Node\ after\_curr = curr.next;$$

2. **Reverse the Link:** Redirect the `next` pointer of the current node (`curr`) to point backward to the previous node (`prev`).

$$curr.next = prev;$$

3. **Advance** `prev`**:** Move the `prev` pointer forward to the current node (`curr`). The node `curr` is now fully reversed.

$$prev = curr;$$

4. **Advance** `curr`**:** Move the `curr` pointer forward to the node that was saved in `after\_curr`.

$$curr = after\_curr;$$

---

3. Final Head and Tail Update

Once the loop terminates (`curr` becomes `null`), the `prev` pointer is at the original last node (which is the new head), and the original `head` is the new tail.

1. **Save Old Head:** Temporarily save the current (old) `head` reference.

$$Node\ temp = head;$$

2. **Update `head` :** Set the list's `head` pointer to the original `tail` (which is the node where `prev` ended up, making the swap clean).

$$head = tail;$$

3. **Update `tail` :** Set the list's `tail` pointer to the original `head` (saved in `temp`).

$$tail = temp;$$

```java
public void pointerIterative(){
    Node prev = null;
    Node curr = head;

    while(curr != null)
    {
        // 1. Save the rest of the list (before overwriting curr.next)
        Node after_curr = curr.next;
        // 2. Reverse the link: Make curr point backward to prev
        curr.next = prev;
        // 3. Advance prev: curr (now reversed) becomes the new prev
        prev = curr;
        // 4. Advance curr: Move to the next saved node
        curr = after_curr;
    }

    // --- Final Head and Tail Swap (Crucial for list utility) ---
    Node temp = head;
    head = tail;
    tail = temp;
}
}
```

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/pointerIterative.java

Remove at index

Algorithm for `removeAt(int idx)`

The goal of this operation is to delete a node at a specified index (`idx`), efficiently handling boundary conditions (empty list, head, tail) before performing the intermediate deletion.

---

1. Handle Edge Cases and Validation

1. **Start:** Begin the `removeAt(idx)` operation.
2. **Empty List Check:**
   - **Condition:** If $size == 0$.
   - **Action:** Print "List is empty" and **return**.
3. **Invalid Index Check:**
   - **Condition:** If $idx < 0$ **OR** $idx \geq size$.
   - **Action:** Print "Invalid index" and **return**.
4. **Remove Head (Index 0):**
   - **Condition:** If $idx == 0$.
   - **Action:** Call the helper method `removeFirst()` and **return**.
5. **Remove Tail (Index `size - 1`):**
   - **Condition:** If $idx == size - 1$.
   - **Action:** Call the helper method `removeLast()` and **return**.

---

2. Remove Intermediate Node (Main Logic)

This logic applies when $0 < idx < size - 1$.

1. **Find Preceding Node:**
   - Initialize a traversal pointer, `prev_node` , to the `head` .
   - **Traverse:** Loop $idx - 1$ times to move `prev_node` to the node *immediately before* the one to be deleted (i.e., the node at index $idx - 1$).
   - The loop runs from $i = 0$ up to $i < idx - 1$: `prev_node = prev_node.next;`
2. **Bypass/Delete Node:**
   - Update the `next` pointer of the `prev_node` to skip over the node at index idx. The link now jumps from the node at $idx - 1$ directly to the node at $idx + 1$.

$$prev\_node.next = prev\_node.next.next;$$

3. **Update Size:** Decrement the list size.

$$size - -;$$

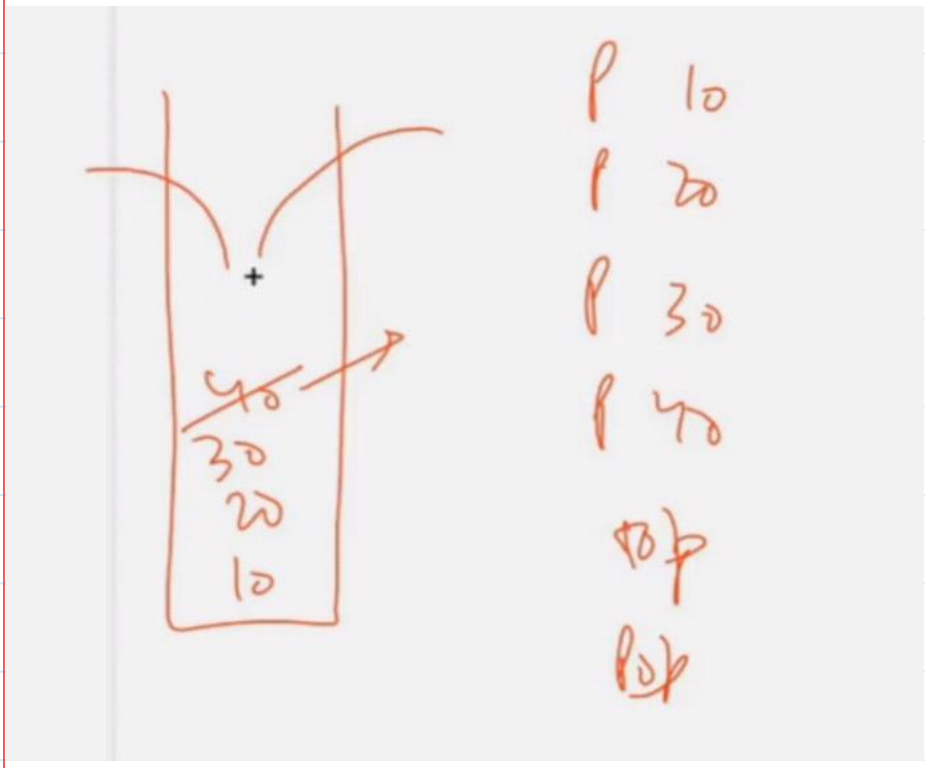
```java
public void removeAt(int idx){
    if (size == 0) {
        System.out.println("List is empty");
        return;
    }

    // Check for invalid index (0 <= idx < size is valid)
    if (idx < 0 || idx >= size) { // Must use >= size for an invalid index
        System.out.println("Invalid index");
        return;
    }

    // Special Case 1: Remove Head
    if (idx == 0) {
        removeFirst();
        return;
    }

    // Special Case 2: Remove Tail
    if (idx == size - 1) {
        removeLast(); // Assumes removeLast() handles the single-node case correctl
        return;
    }

    // Main Logic: Remove Intermediate Node (0 < idx < size - 1)
    {
        // Find the node *just before* the one to be deleted (at idx-1)
        Node prev_node = head;
        // Loop runs idx - 1 times
        for(int i = 0; i < idx - 1; i++){
            prev_node = prev_node.next;
        }

        // Let the preceding node bypass the current node (the one to be deleted).
        // prev_node.next is the node to delete.
        // prev_node.next.next is the node after the one to delete.
        prev_node.next = prev_node.next.next;

        size--;
    }
}
```

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/removeAt.java

Linkedlist to stack adapter.

18 November 2025 19:27

How stack works



Same end addition, same end removal.

addFirst()
removeFirst()
getFirst()

between a **conceptual implementation** (like the one you were reviewing) and the **optimized, built-in implementation** found in a language's standard library.

Here is the explanation for why `removeLast()` is $O(n)$ in your typical Singly Linked List implementation, but $O(1)$ in Java's standard library lists.

🔗 1. Why Your `removeLast()` is $O(n)$

Your implementation of `removeLast()` is based on a standard **Singly Linked List**.

In a Singly Linked List, each node only stores a pointer to the **next** node.

1. To remove the last node, you must set the `next` pointer of the **second-to-last** node to `null`.
2. Because you can only move forward, there is no direct pointer to the second-to-last node.
3. You must start at the `head` and **traverse** the entire list until you reach the node whose `next` pointer is the `tail`. This traversal requires visiting N nodes, making the time complexity $O(n)$ **(Linear Time)**.

💻 2. Why Java's `removeLast()` is $O(1)$

The built-in list classes in Java, specifically the `LinkedList` class, are not implemented as simple Singly Linked Lists. Instead, they use a **Doubly Linked List** structure.

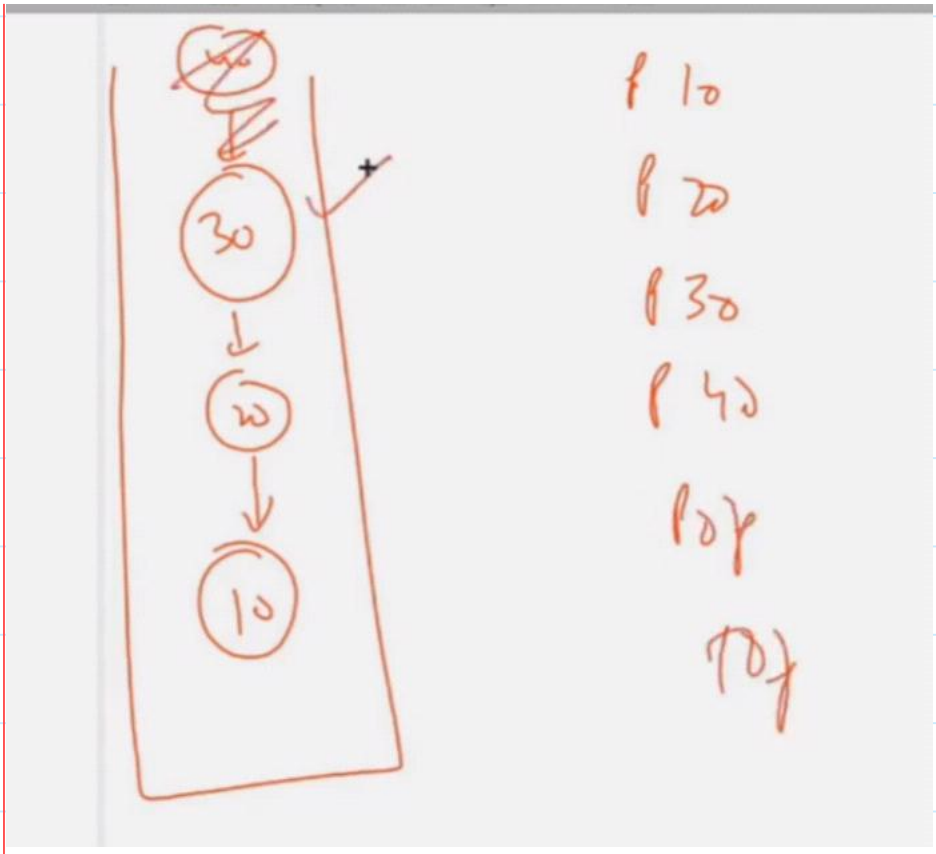
In a Doubly Linked List:

1. Each node has a pointer to the `next` node *and* a pointer to the `prev` (previous) node.
2. The list itself maintains a direct reference to both the `head` and the `tail` node.

To perform `removeLast()` on a Doubly Linked List:

1. Access the `tail` node in $O(1)$ time.
2. Access the second-to-last node via the `tail.prev` pointer in $O(1)$ time.
3. Set the new tail's `next` pointer to `null` and update the list's `tail` reference, all in $O(1)$ **(Constant Time)**.

Linked List Type	`removeLast()` Complexity	Reason
Singly Linked List (Your implementation)	$O(n)$	Must traverse from the **Head** to find the second-to-last node.
Doubly Linked List (Java's `LinkedList`)	$O(1)$	Can access the second-to-last node directly via the `tail.prev` pointer.

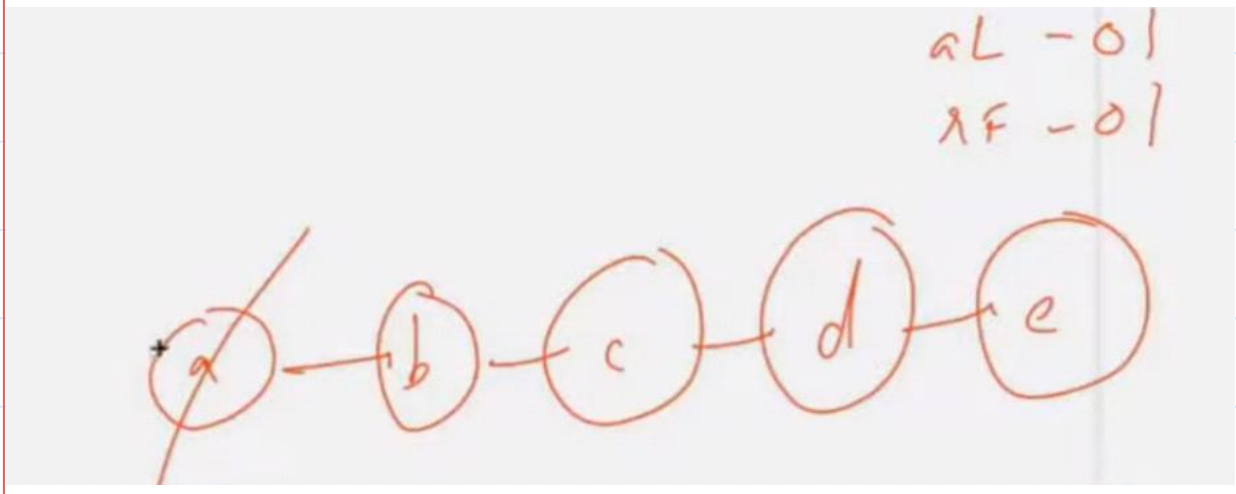


https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/LLToStackAdapterFunctionality.java

Linkedlist to queue adapter

18 November 2025 20:45

Queue is FIFO



https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/LLToQueueAdapterFunctionality.java

Kth element from the end in LL

18 November 2025 20:50

❓ Problem Statement: Kth Element From the End of a Linked List

Objective: Write an iterative function to find the K^{th} element from the end of a Singly Linked List (LL).

Constraints:

- The solution must be **iterative**.
- The solution **must not** use the list's `size()` function or a pre-calculated `size` data member.

Input Parameter:

- K: An integer representing the offset from the end of the list.

Output (Index Definition):

- If $K = 0$, the function should return the **last node**.
- If $K = 1$, the function should return the **second-to-last node**.

💡 Algorithm Hint (Two-Pointer Method)

Since you cannot use the size, the standard efficient iterative approach uses **two pointers** separated by K nodes.

1. Initialize two pointers, `slow` and `fast`, both starting at the head.
2. Move the `fast` pointer K steps ahead.
3. Move both `slow` and `fast` one step at a time until `fast` reaches the end of the list (`null`).
4. When `fast` reaches `null`, `slow` will be pointing to the K^{th} element from the end. Would

```java
public Node kthFromEnd(int k) {
    if (k < 0) {
        System.out.println(x: "K must be non-negative.");
        return null;
    }

    Node slow = head;
    Node fast = head;

    // --- 1. Establish the K-node gap ---
    for (int i = 0; i < k; i++) {
        if (fast == null) {
            // Error: K is greater than or equal to the size of the list
            System.out.println(x: "Invalid K. List is too short.");
            return null;
        }
        fast = fast.next;
    }

    // --- 2. Move both pointers together until 'fast' is null ---
    // Since k=0 means the last node,
    //'fast' points to null when 'slow' points to the last node.
    while (fast != null) {
        slow = slow.next;
        fast = fast.next;
    }

    // --- 3. Return the result ---
    // 'slow' is now at the Kth node from the end.
    return slow;
}
```

🧐 Comparison of Loop Conditions

1. Standard Method: `while (fast != null)` (Preferred)

This is the standard, robust, and mathematically cleaner approach.

- **Logic:** The loop runs until the `fast` pointer moves **one step past the last node**.
- **Result:** When the loop terminates, the `slow` pointer has moved exactly $N - K - 1$ steps, placing it correctly at the K^{th} element from the end.
- **Example:**
  - If you want the **last node** ($K = 0$), you need the `slow` pointer to stop when `fast` is `null`.
  - If you want the **second-to-last node** ($K = 1$), you need `slow` to stop when `fast.next` is `null` (i.e., when `fast` is the last node, and the loop continues one more step until `fast` becomes `null`).
- **Safety:** It works correctly regardless of whether the `tail` pointer is null, points to the correct node, or is even tracked at all.

2. Proposed Method: `while (fast != tail)` (Problematic)

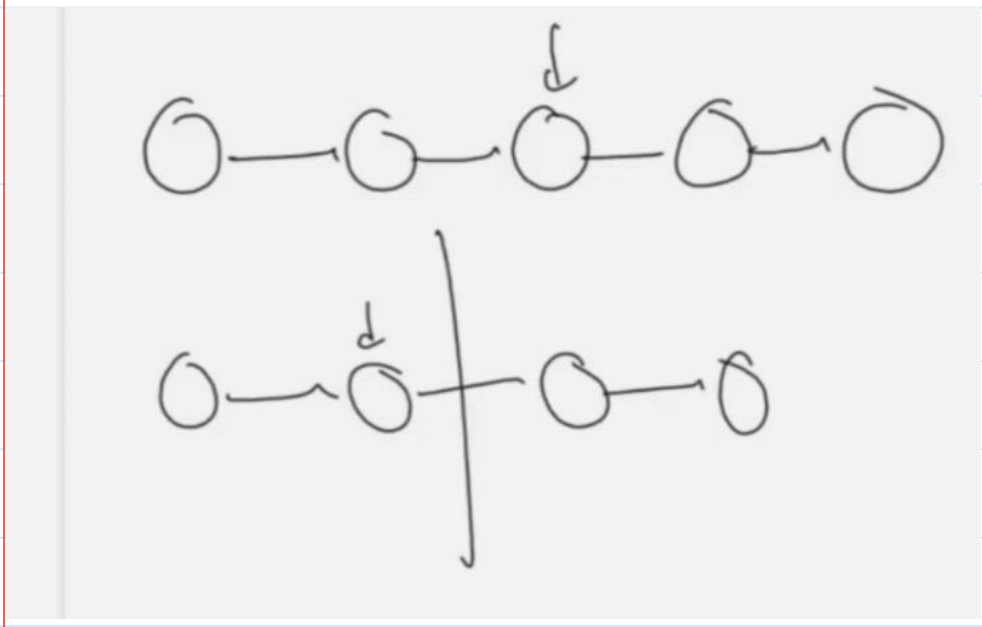
This condition causes the loop to stop too early, only placing the `slow` pointer at the correct position if you adjust the initial K steps.

- **Logic:** The loop stops when the `fast` pointer is pointing *at* the last node (the `tail`).
- **Result:** When the loop stops, the `slow` pointer is **one node short** of the correct position.
  - The total steps taken by `slow` is $N - K - 2$.
- **Example:**
  - If $K = 0$ (last node), you want `slow` to stop at the last node. `fast` must initially be K steps ahead, and the loop stops when `fast` is at the `tail`. This means you need the loop to run $N - K - 1$ times, which it does not.
- **Requirement:** To use this condition, you would typically need to initialize the `fast` pointer to be $K - 1$ steps ahead instead of K steps, which complicates the logic and the error checking. Furthermore, you must ensure the `tail` pointer is always correctly maintained by the list.

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/kthFromEnd.java

Middle of linkedlist

18 November 2025 21:03



In case of even no of node, first one is mid.

🔍 Problem Statement: Finding the Middle Node of a Linked List

Objective: Design an iterative algorithm to efficiently find the middle node of a given Singly Linked List (LL).

Constraints:

- The solution must be **iterative**.
- The algorithm must be completed in a **single pass** through the list (i.e., $O(n)$ time complexity).
- The solution **must not** rely on calculating the size/length of the list beforehand.

Input: The `head` pointer of a Singly Linked List.

Output (Definition of Middle):

- If the list has an **odd** number of nodes (e.g., 5 nodes), return the **single center node**.
- If the list has an **even** number of nodes (e.g., 6 nodes), return the **first of the two middle nodes** (the node at the $\frac{N}{2}$ position).

🚓 Algorithm: Middle Node of Linked List

The algorithm uses two pointers, `slow` and `fast`, where `fast` moves twice as quickly as `slow`. When `fast` reaches the end, `slow` will be exactly in the middle.

1. Initialization

1. **Start:** Begin the operation.
2. **Pointers:** Initialize two pointers, `slow` and `fast`, both pointing to the `head` of the linked list.

2. Traversal Loop

1. **Condition:** Start a `while` loop that continues as long as the `fast` pointer has a valid **next** step. This ensures we stop at the right moment regardless of whether the list length is odd or even.

$$\text{while } (\text{fast} \neq \text{null AND fast.next} \neq \text{null})$$

2. **Advance `slow`:** Move the `slow` pointer by **one** step.

$$\text{slow} = \text{slow.next};$$

3. **Advance `fast`:** Move the `fast` pointer by **two** steps.

$$\text{fast} = \text{fast.next.next};$$

3. Result

1. **Stop Condition:** When the loop terminates, the `fast` pointer has either reached the last node's `next` (for odd length) or the node just before the last (for even length, before the final step is taken).
2. **Return:** The `slow` pointer will be positioned exactly at the **middle node** of the linked list. **Return** the node or its data.

⏳ Time Complexity

The time complexity is $O(n)$ **(Linear Time)** because the list is traversed only once.

```java
public int mid(){
    Node slow = head;
    Node fast = head;

    while(fast.next != null){
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow.data;
}
```

❌ 2. Failure: Even-Length List Traversal

The loop condition is too restrictive for lists with an **even number of nodes** (e.g., 4, 6, 8...).

- **Scenario (4 nodes: 1 → 2 → 3 → 4):** | Pointers | **slow** | **fast** | `fast.next != null`? | Action | | :--- | :--- | :--- | :--- | :--- | | **Start** | 1 | 1 | True (1.next = 2) | Loop | | **Iter 1** | 2 | 3 | True (3.next = 4) | Loop | | **Iter 2** | 3 | **null** | **FAIL!** |
- **Failure Point:** In the second iteration, `fast` is assigned `fast.next.next` (which is $1 \rightarrow 3 \rightarrow \text{null}$). So, `fast` becomes the **4th node's** `next` (i.e., null).
- **Result:** The code will throw a `NullPointerException` when it tries to evaluate the condition *at the start of the next iteration* because it attempts to access `fast.next` when `fast` is already `null`.

✅ The Correct Loop Condition

To fix the issue and make the code robust for all list lengths, you must include a check for whether `fast` itself is `null` before trying to access its `next` pointer.

The correct, standard loop condition for the Fast and Slow Pointer technique is:

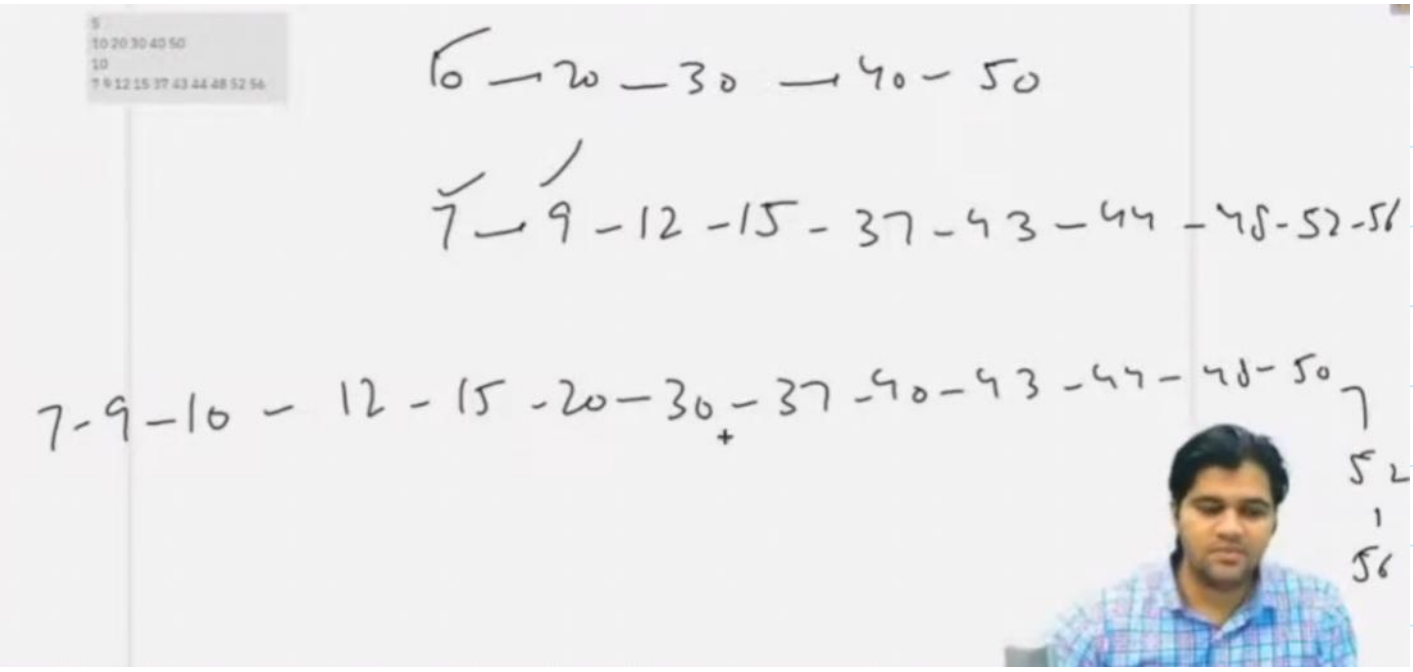
$$\text{while } (\text{fast} \neq \text{null AND fast.next} \neq \text{null})$$

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/mid.java

Untitled

18 November 2025 21:10

Merge two sorted linkedlist.



Algorithm: Merging Two Sorted Linked Lists

This algorithm uses a single-pass approach to combine the elements from two sorted lists, $L1$ and $L2$, into a new sorted list, $L3$. This approach maintains the overall sorted order and achieves $O(N + M)$ time complexity.

1. Initialization

1. Initialize two traversal pointers, `one` and `two`, to the `head` of the input lists $L1$ and $L2$, respectively.
   - `Node one = l1.head;`
   - `Node two = l2.head;`
2. Create a new, empty `LinkedList` object, $L3$, to store the merged result.
   - `LinkedList l3 = new LinkedList();`

---

2. Main Comparison Loop

Start a `while` **loop** that continues as long as **both** traversal pointers (`one` and `two`) are not null.

1. **Compare Data:** Check which pointer points to the smaller element: `if (one.data < two.data)`.
2. **Add Smaller Element:**
   - **If** `one` **is smaller:** Add the data from the `one` node to the end of $L3$ (`l3.addLast(one.data)`). Then, advance the `one` pointer to the next node (`one = one.next`).
   - **If** `two` **is smaller or equal:** Add the data from the `two` node to the end of $L3$ (`l3.addLast(two.data)`). Then, advance the `two` pointer to the next node (`two = two.next`).

---

3. Append Remaining Elements

After the main loop terminates, one of the input lists is completely processed (`one` or `two` is null). The remaining list contains elements that are all larger than any element already in $L3$.

1. **Append $L1$ Remainder:** Start a `while` loop that runs as long as the pointer `one` is not null.
   - Add the data of the `one` node to $L3$ (`l3.addLast(one.data)`).
   - Advance the `one` pointer (`one = one.next`).
2. **Append $L2$ Remainder:** Start a `while` loop that runs as long as the pointer `two` is not null.
   - Add the data of the `two` node to $L3$ (`l3.addLast(two.data)`).
   - Advance the `two` pointer (`two = two.next`).

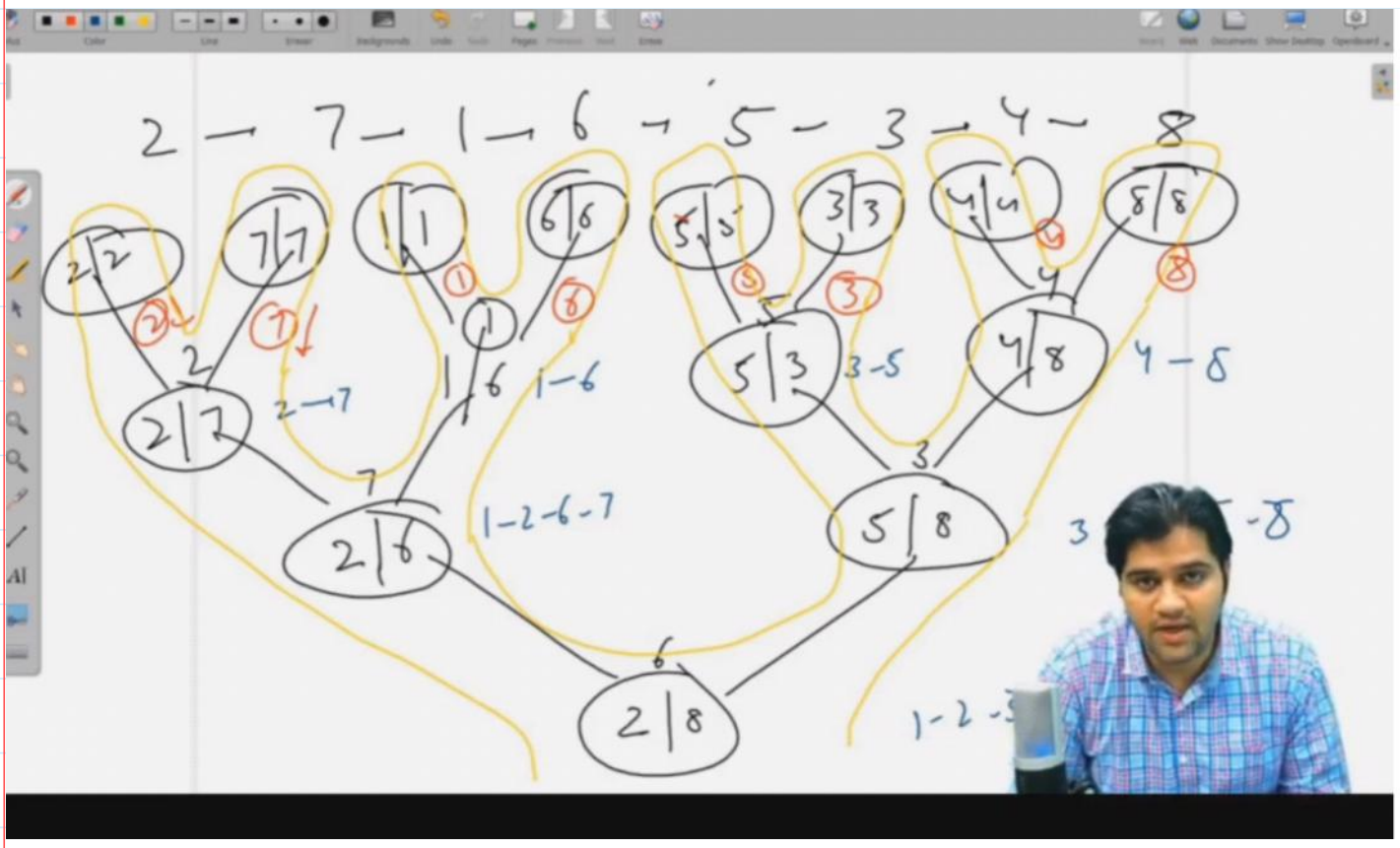
4. Return Result

1. **Return** the new merged list, $L3$.

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/mergeTwoSortedLists.java

Merge sort on LL

19 November 2025 18:05



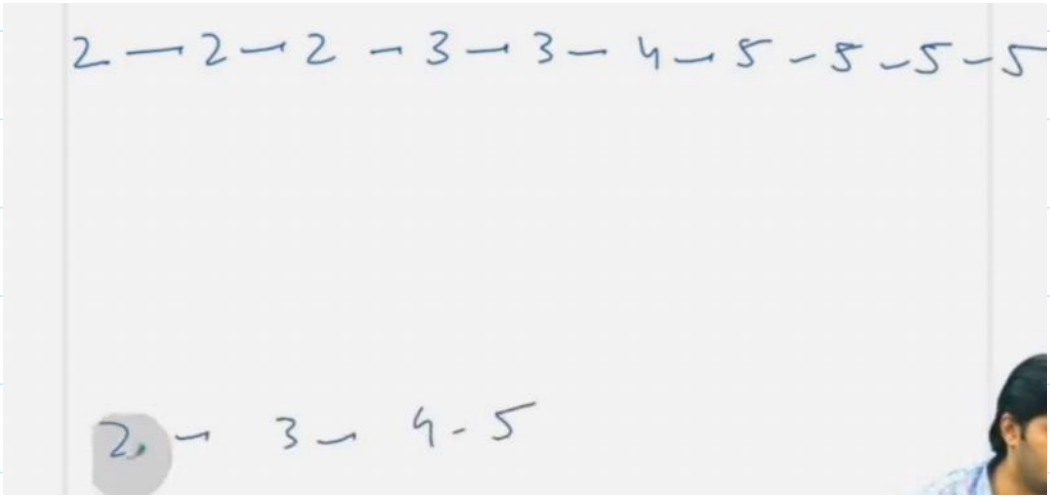
https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/mergeSort.java

Untitled

19 November 2025 18:29

Remove duplicate in sorted linkedlist

Time complexity linear
Space complexity: constant.




```
public class removeDuplicates {
    public static class Node{
        int data;
        Node next;
    }
    public static class LinkedList{
        Node head;
        Node tail;
        int size;
        void addLast(int val){
            // ... (Correct and unchanged)
            Node temp = new Node();
            temp.data=val;
            temp.next=null;
            if(size==0){
                head=tail=temp;
            }else{
                tail.next =temp;
                tail=temp;
```

```
            }
            size++;
        }
        public void display() {
            // ... (Correct and unchanged)
            Node temp=head;
            if(size==0){
                System.out.println("List is
empty");
            } else{
                while(temp!=null){
                    System.out.print(temp.data);
                    if(temp.next!=null){
                        System.out.print("->");
                    }
                    temp=temp.next;
                }
                System.out.println();
            }
        }
        /**
         * 🛠️ CORRECTION: Changed to return int to
support removeDuplicates logic.
         */
        public int removeFirst(){
            if(size==0){
                System.out.println("List is
empty");
                return -1; // Indicate error
            }

            int removedData = head.data; // Save
data before removal

            if(size==1){
                head=tail=null;
                size=0; // size should be 0
            } else{
                head = head.next;
                size--;
            }
            return removedData;
        }
```

```
        // 🛠 ADDITION: Necessary method for
removeDuplicates
        public int getFirst(){
             if(size==0){
                System.out.println("List is
empty");
                return -1;
            }
            return head.data;
        }

        // ... (getAt, addFirst, addAt,
removeLast, getNodeAt, reverseDI,
        // pointerIterative, removeAt, kthFromEnd
are assumed correct and unchanged) ...

        // --- Remaining methods for
completeness ---
        // (Copying existing methods here for a
single block of runnable code)
        public int getAt(int idx){
            if(size==0){
                System.out.println("List is
empty");
                return -1;
            } else if(idx<0 || idx>=size){
                System.out.println("Invalid
argument");
                return -1;
            } else{
                Node temp= head;
                for(int i=0; i<idx; i++){
                    temp =temp.next;
                }
                return temp.data;
            }
        }

        public void addFirst(int val){
            Node temp = new Node();
            temp.data = val;
            temp.next = null;
            if(size == 0){
```

```
                head =tail =temp;

            }else{
                temp.next = head;
                head =temp;
            }
            size++;
        }
        public void addAt(int idx, int val){
            if(idx==0){
                addFirst(val);
            } else if(idx == size){
                addLast(val);
            } else if(idx<0 || idx>size){
                System.out.println("Invalid
index");
            } else{
                Node temp = new Node();
                temp.data =val;
                Node traverse= head;
                for(int i=0;i<idx-1; i++){
                    traverse = traverse.next;
                }
                temp.next = traverse.next;
                traverse.next =temp;
                size++;
            }
        }
        public void removeLast(){
            if(size==0){
                System.out.println("List is
empty");
            } else if(size ==1){
                head =tail =null;
                size--;
            } else {
                Node traverse = head;
                for(int i=0; i<size -2; i++){
                    traverse = traverse.next;
                }
                tail = traverse;
                traverse.next=null;
                size--;
```

```
        }
    }
    private Node getNodeAt(int idx){
        Node traverse= head;
        for(int i=0; i<idx; i++){
            traverse= traverse.next;
        }
        return traverse;
    }
    public void reverseDI(){
        int li = 0;
        int ri = size -1;
        while(li< ri){
            Node left = getNodeAt(li);
            Node right = getNodeAt(ri);
            int temp = left.data;
            left.data = right.data;
            right.data = temp;
            li++;
            ri--;
        }
    }
    public void pointerIterative(){
        Node prev = null;
        Node curr = head;
        if(size <= 1) return; // Handle 0 or 1
node lists
        while(curr != null)
        {
            Node after_curr = curr.next;
            curr.next = prev;
            prev = curr;
            curr = after_curr;
        }

        // Final Head and Tail Swap
        Node temp = head;
        head = tail;
        tail = temp;
    }
    public void removeAt(int idx){
        if (size == 0) {
            System.out.println("List is
```

```
empty");
                return;
            }
            if (idx < 0 || idx >= size) {
                System.out.println("Invalid
index");
                return;
            }

            if (idx == 0) {
                removeFirst(); // Already size--
handled inside
                return;
            }
            if (idx == size - 1) {
                removeLast(); // Already size--
handled inside
                return;
            }

            Node prev_node = head;
            for(int i = 0; i < idx - 1; i++){
                prev_node = prev_node.next;
            }
            prev_node.next = prev_node.next.next;
            size--; // Only decrement here since
removeFirst/Last handle size
        }
        public Node kthFromEnd(int k) {
            if (k < 0) {
                System.out.println("K must be non-
negative.");
                return null;
            }

            Node slow = head;
            Node fast = head;
            for (int i = 0; i < k; i++) {
                if (fast == null) {
                    System.out.println("Invalid K.
List is too short.");
                    return null;
                }
```

```
                fast = fast.next;
            }
            while (fast != null) {
                slow = slow.next;
                fast = fast.next;
            }
            return slow;
        }
        public int mid(){
            if (head == null) {
                System.out.println("List is
empty.");
                return -1;
            }

            Node slow = head;
            Node fast = head;
            // This condition is robust for both
odd and even lengths
            while(fast.next != null &&
fast.next.next != null){
                slow = slow.next;
                fast = fast.next.next;
            }

            return slow.data;
        }
        /**
         * 🛠 CORRECTION: The logic is a
functional way to remove duplicates by rebuilding
the list.
         * The only needed correction was to link
the result list (res) back to the current list's
head/tail/size.
         */
        public void removeDuplicates(){
            LinkedList res= new LinkedList();
            while(this.size > 0){
                int val = this.getFirst(); // Use
the newly defined getFirst()
                this.removeFirst(); // Use the
newly defined removeFirst() (returns int)
                if(res.size==0 || res.tail.data !=
```

```
val){
                    res.addLast(val);
                }
            }

            // Link the result list (res) back to
the current list ('this')
            this.head = res.head;
            this.tail = res.tail;
            this.size = res.size;
        }
    }

    public static void main(String[] args) {
        System.out.println("🚀 Testing LinkedList
Functionality 🚀");
        System.out.println("---");
        // Initialize the LinkedList
        LinkedList list = new LinkedList();

        list.addLast(10);
        list.addLast(10);
        list.addLast(20);
        list.addLast(30);
        list.addLast(30);
        list.addLast(30);
        list.addLast(40);

        System.out.print("Original List: ");
        list.display(); // Output:
10->10->20->30->30->30->40
        System.out.println("\nRunning
removeDuplicates()...");
        list.removeDuplicates();
        System.out.print("Modified List: ");
        list.display(); // Output: 10->20->30->40
        System.out.println("---");
    }
}
```

Odd even list

19 November 2025 19:07

❓ Problem Statement: Odd-Value First, Even-Value Last

Objective: Write an efficient iterative algorithm to rearrange the nodes of a given singly linked list such that all nodes containing **odd integer data** appear before all nodes containing **even integer data**. The relative order within the odd group and the even group must be preserved. The list's `head` and `tail` pointers must be correctly updated after the rearrangement.

Constraints:

- The solution must be **in-place** (by changing pointers, not by creating new nodes or changing data).
- The solution should be **iterative**.
- Time complexity must be $O(N)$, and space complexity must be $O(1)$.

Input: The `head` and `tail` pointers of a singly linked list.

Output: The updated `head` and `tail` pointers of the rearranged list.

Example

Input List (Data)	Odd Group (Preserved Order)	Even Group (Preserved Order)	Output List (Data)	New Tail
1 → 4 → 3 → 2 → 5 → 6 → null	1 → 3 → 5	4 → 2 → 6	1 → 3 → 5 → 4 → 2 → 6 → null	Node 6
10 → 1 → 8 → 3 → 4 → null	1 → 3	10 → 8 → 4	1 → 3 → 10 → 8 → 4 → null	Node 4
2 → 4 → 6 → 8 → null	(Empty)	2 → 4 → 6 → 8	2 → 4 → 6 → 8 → null	Node 8

Export to Sheets

Input List

$$\text{Head} \to 2 \to 9 \to 7 \to 8 \to 1 \to 6 \to 5 \to 7 \to \text{null}$$

Final Output List

The rearranged list:

$$\mathbf{9} \to 7 \to 1 \to 5 \to 7 \to 2 \to 8 \to \mathbf{6} \to \text{null}$$

Here we will make use of linkedlist function instead of dealing with nodes, we will make use of oops, abstraction of linkedlist class which we have written.

```
public void oddEven(){
            LinkedList odd = new LinkedList();
            LinkedList even = new LinkedList();
            while(this.size >0){
                int val = this.getFirst();
                this.removeFirst();
                if(val%2==0){
                    even.addLast(val);
                } else{
                    odd.addLast(val);
                }
            }

            if(odd.size>0 && even.size >0){
                odd.tail.next = even.head;
                this.head = odd.head;
                this.tail = even.tail;
                this.size = odd.size +even.size;
            }else if(odd.size>0){
                this.head = odd.head;
                this.tail = odd.tail;
                this.size = odd.size;
            }else if(even.size>0){
                this.head = even.head;
                this.tail = even.tail;
                this.size = even.size;
            }
        }
```

K reverse in a linkedlist

19 November 2025 19:47

❓ Problem Statement: K-Reverse in a Linked List

Objective: Given the `head` of a singly linked list and an integer K, reverse the nodes of the list in groups of size K. If the number of nodes remaining is less than K, the remaining nodes should stay in their original relative order.

Constraints:

- K is a positive integer.
- The solution must only change the nodes' next pointers (i.e., it must be an **in-place** pointer-based reversal).
- The algorithm should be $O(N)$ time complexity and $O(1)$ space complexity.

Example

Input List	Group Size (K)	Output List (Grouped Reverse)
$1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to \text{null}$	$K = 3$	$3 \to 2 \to 1 \to 6 \to 5 \to 4 \to 9 \to 8 \to 7 \to 10 \to 11 \to \text{null}$

Step-by-Step Reversal for $K = 3$:

Group	Original	Reversed
Group 1	$1 \to 2 \to 3$	$3 \to 2 \to 1$
Group 2	$4 \to 5 \to 6$	$6 \to 5 \to 4$
Group 3	$7 \to 8 \to 9$	$9 \to 8 \to 7$
Remaining	$10 \to 11$	$10 \to 11$ (Not enough nodes to reverse)

Export to Sheets

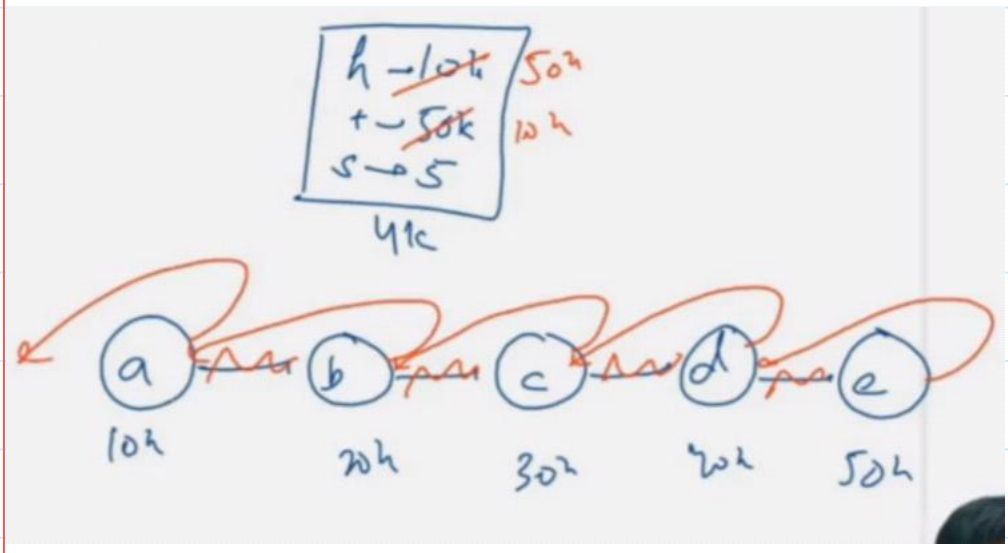
Output:

$$\text{Head} \to \mathbf{3} \to \mathbf{2} \to \mathbf{1} \to \mathbf{6} \to \mathbf{5} \to \mathbf{4} \to \mathbf{9} \to \mathbf{8} \to \mathbf{7} \to 10 \to 11 \to \text{null}$$

```
public void kReverse(int k){
            LinkedList prev = null;
            while(this.size>0){
                LinkedList curr = new LinkedList();
                if(this.size>=k){
                    for(int i=0; i<k; i++){
                        int val = this.getFirst();
                        this.removeFirst();
                        curr.addFirst(val);
                    }
                }else{
                    int s = this.size;
                    for(int i=0; i<s; i++){
                        int val = this.getFirst();
                        this.removeFirst();
                        curr.addLast(val);
                    }
                }
                if(prev==null){
                    prev=curr;
                } else{
                    prev.tail.next = curr.head;
                    prev.tail = curr.tail;
                    prev.size += curr.size;
                }
            }
            this.head = prev.head;
            this.tail = prev.tail;
            this.size = prev.size;
        }
```

Reverse a linked list Using pointer recursion.

20 November 2025 18:52

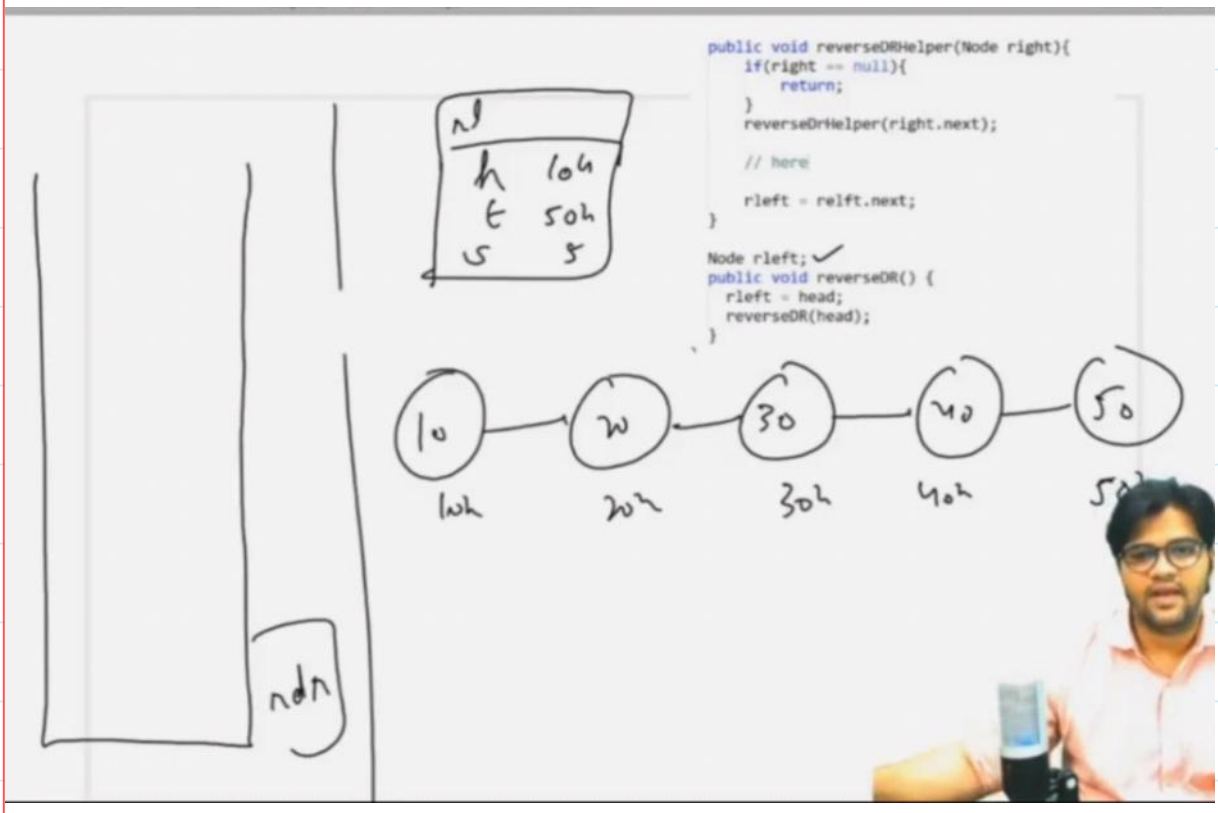



```
private void reversePRHelper(Node node){
    if(node==null){
        return;
    }
    reversePRHelper(node.next);
    if(node == tail){
        //nothing to do
    } else{
        node.next.next = node;
    }
}
public void reversePR(){
    reversePRHelper(head);
    head.next = null;
    Node temp = head;
    head = tail;
    tail = temp;
}
```

Reverse a linkedlist - data recursive

20 November 2025 19:04

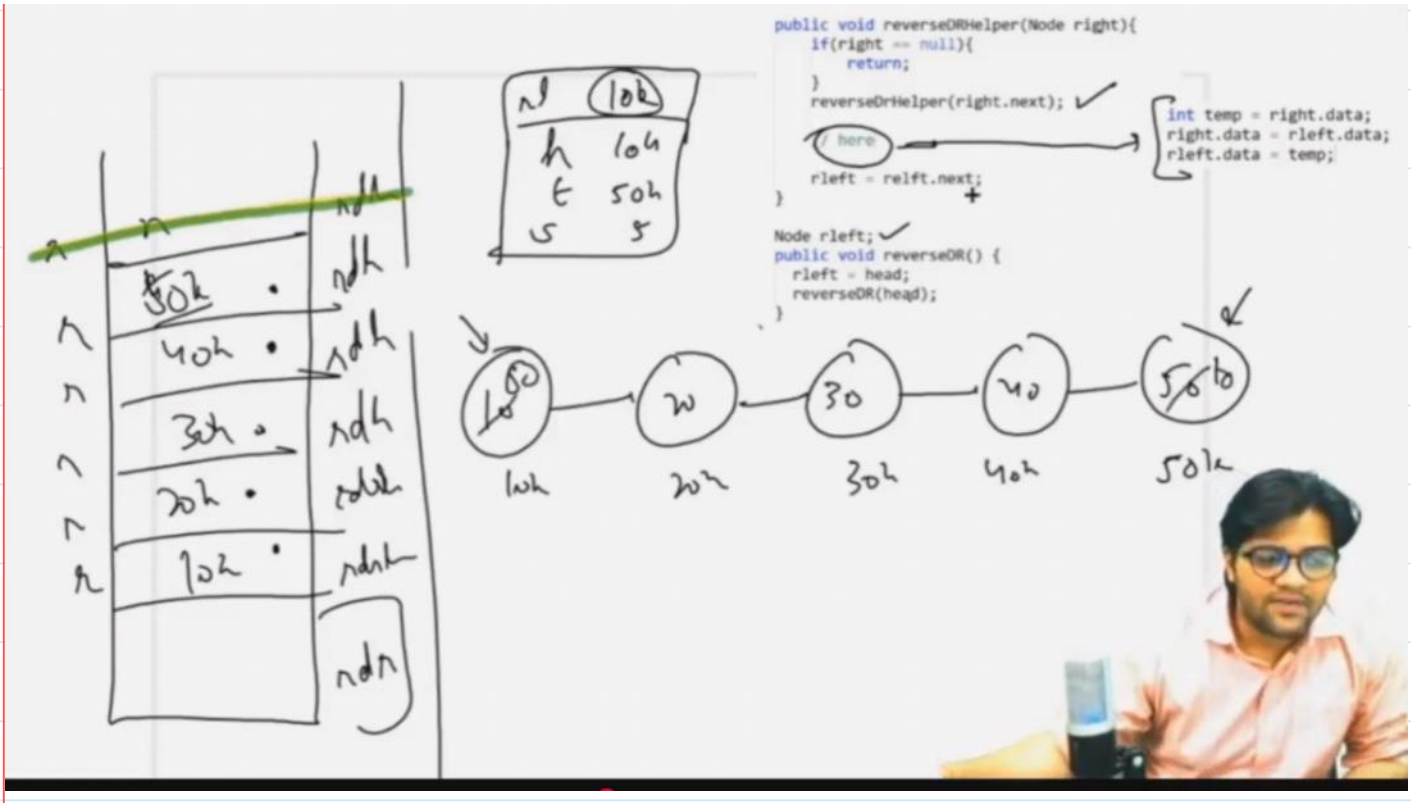
Change data , but don't make changes in pointer



Explain this much code.

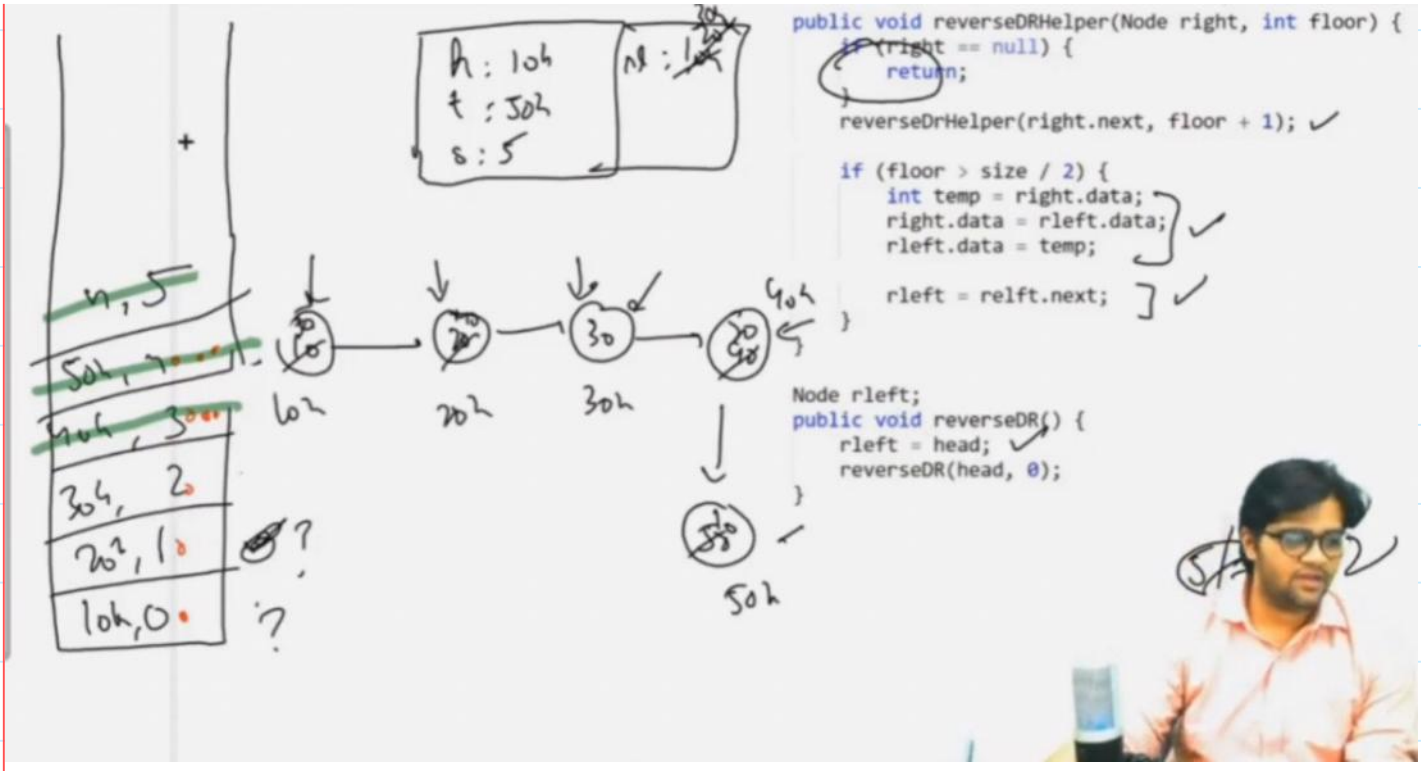
How rleft is moving fromleft to right (the variable made outside, just like head, tail and size)

and right will keep coming back with help of recursion.




```
public void reverseDRHelper(Node right, int floor) {
    if (right == null) {
        return;
    }
    reverseDrHelper(right.next, floor + 1);

    if (floor > size / 2) {
        int temp = right.data;
        right.data = rleft.data;
        rleft.data = temp;

        rleft = relft.next;
    }
}

Node rleft;
public void reverseDR() {
    rleft = head;
    reverseDR(head, 0);
}
```

Adding a new argument floor, so that the entire operation happens only till both reach the mid of the linkedlist, so that it doesn't keep on happening for the rest of LL(because that would again create the original list).



Then do dry run for full code.

Then do for even case(when even no of nodes are there), then there will be change

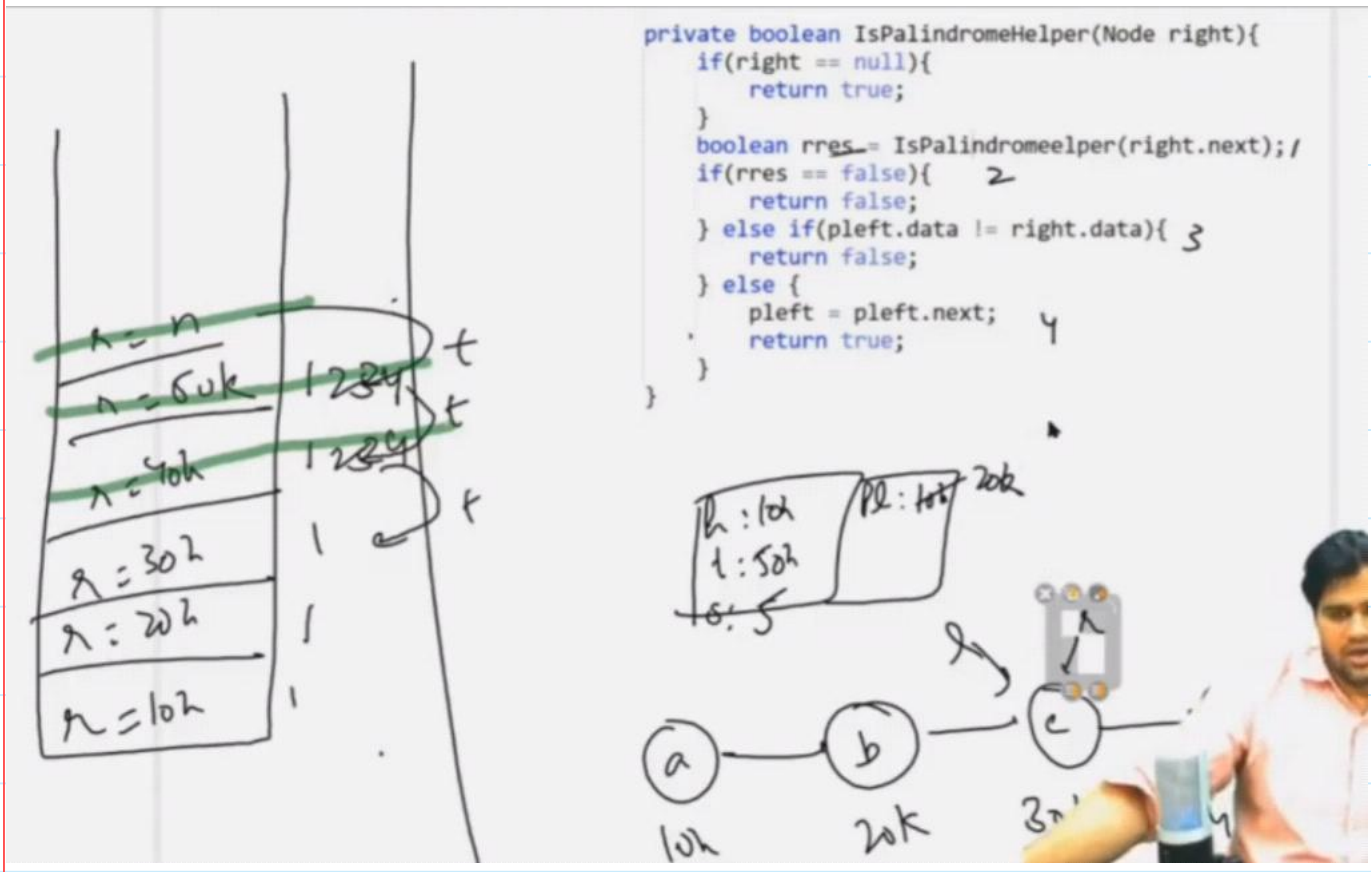
If(floor **>=** size/2)

```
public void reverseDRHelper(Node right, int floor) {
    if (right == null) {
        return;
    }
    reverseDrHelper(right.next, floor + 1);

    if (floor >= size / 2) {
        int temp = right.data;
        right.data = rleft.data;
        rleft.data = temp;

        rleft = relft.next;
    }
}

Node rleft;
public void reverseDR() {
    rleft = head;
    reverseDR(head, 0);
}
```

Is a linkedlist palindrome

20 November 2025 19:17

Use data recursive logic

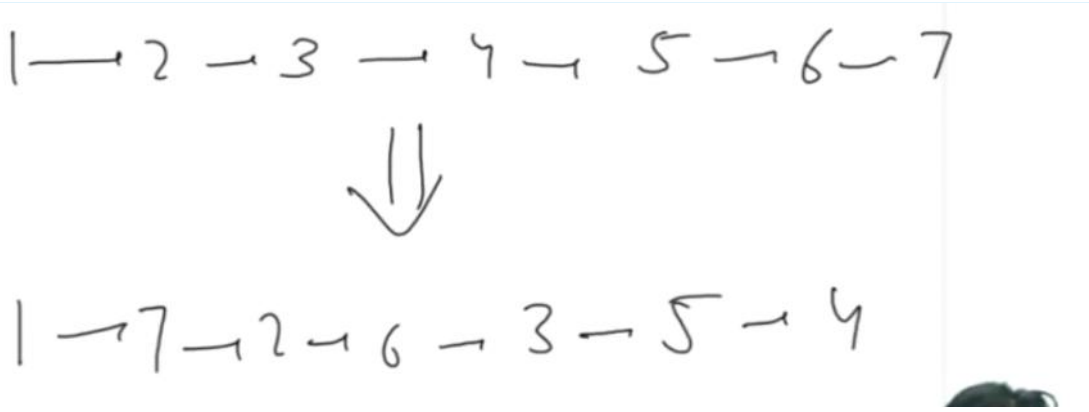


And do dry run as many times as possible.

```
    private boolean IsPalindromeHelper(Node right){
        if(right == null){
            return true;
        }
        boolean rres = IsPalindromeHelper(right.next);
        if(rres == false){
            return false;
        } else if(pleft.data != right.data){
            return false;
        } else {
            pleft = pleft.next;
            return true;
        }
    }

    Node pleft;
    public boolean IsPalindrome() {
      pleft = head;
      return IsPalindromeHelper(head);
    }
}
```

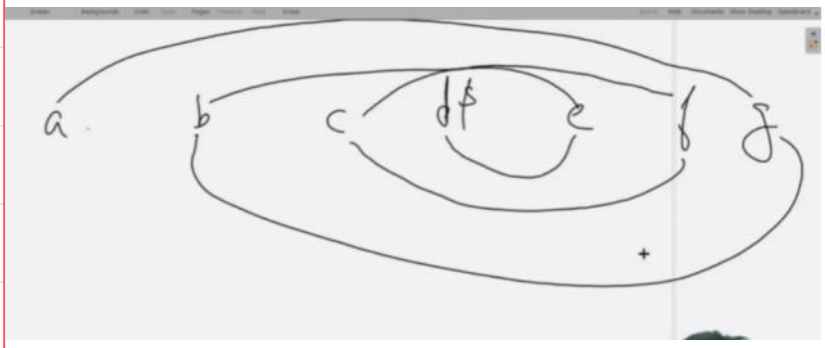
Fold a linkedlist

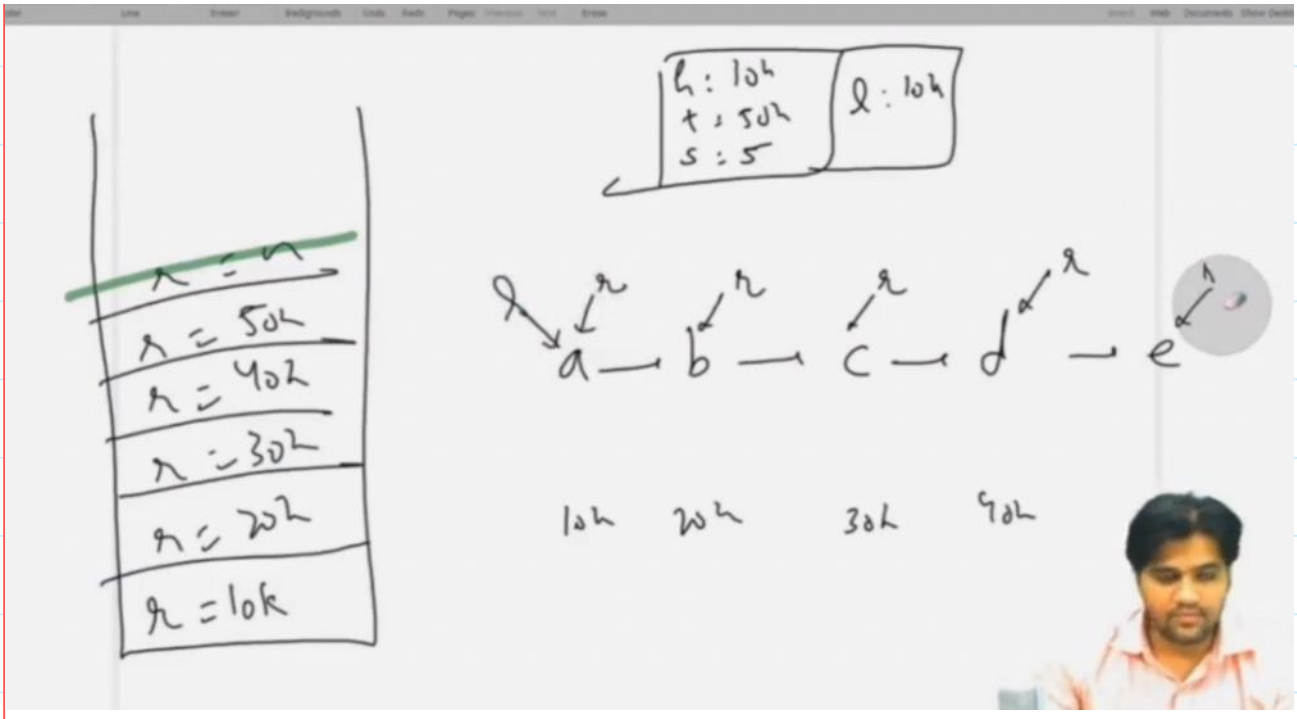
20 November 2025 19:23



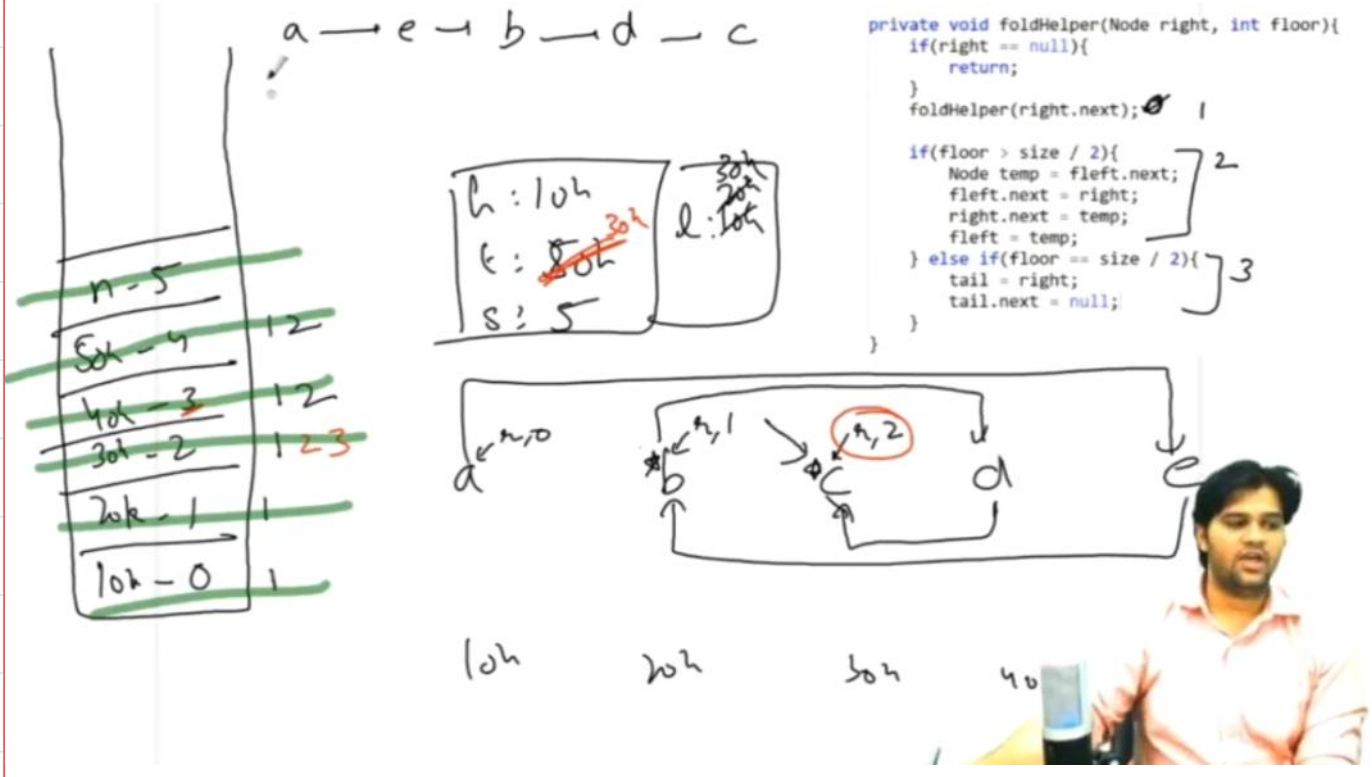
After first->last->second->secondLast-> …

How about doing this





Do full dry run




```
private void foldHelper(Node right, int floor){
    if(right == null){
        return;
    }
    foldHelper(right.next);

    if(floor > size / 2){
        Node temp = fleft.next;
        fleft.next = right;
        right.next = temp;
        fleft = temp;
    } else if(floor == size / 2){
        tail = right;
        tail.next = null;
    }
}
```

Final code

```
    private void foldHelper(Node right, int floor){
        if(right == null){
            return;
        }
        foldHelper(right.next, floor + 1);

        if(floor > size / 2){
            Node temp = fleft.next;
            fleft.next = right;
            right.next = temp;
            fleft = temp;
        } else if(floor == size / 2){
            tail = right;
            tail.next = null;
        }
    }

    Node fleft;
    public void fold() {
      fleft = head;
      foldHelper(head, 0);
    }
  }
```

Last 3question to understand, which variable tomake data member and what to make as parameter.

Datamember one will be in heap(rleft) and the one passed as parameter will keep coming down with help of recursion.

Add two linkedlist

20 November 2025 19:32

Do convert to array, or to number, or reverse the linkedlist.

Do using linkedlist. (as most significant bit is on head, so it will be challenging).

Question Statement

You are given the heads of two singly linked lists, `List L1` and `List L2`, where each node contains a single digit. The digits are stored in **forward order** (i.e., the head node represents the most significant digit, MSD).

Implement the complete Java solution, including the `addListHelper` recursive function, to calculate the sum of the two numbers represented by the linked lists and return the resulting sum as a new linked list.

Input Specification

📜 Algorithm: Recursive Addition (MSD First)

I. Main Function: `addTwoLists(list1, list2)`

This function prepares the environment and handles the final step of the addition.

1. **Initialize Result:** Create a new empty linked list, rcs, to store the sum.
2. **Get Lengths/Positions:** Retrieve the size of the two input lists ($\text{pv}_1 = \text{list}_1.\text{size}$ and $\text{pv}_2 = \text{list}_2.\text{size}$). These lengths act as position trackers.
3. **Recursive Call:** Call the recursive helper function, passing the heads of the lists, their sizes, and the result list:

$$\text{finalCarry} = \text{addListHelper}(\text{list}_1.\text{head}, \text{pv}_1, \text{list}_2.\text{head}, \text{pv}_2, \text{rcs})$$

4. **Handle Final Carry:** After the recursion completes, check if the returned finalCarry is greater than 0.
   - If $\text{finalCarry} > 0$, add it as the new most significant digit to the result list by calling $\text{rcs}.\text{addFirst}(\text{finalCarry})$.
5. **Return:** Return the resulting linked list, rcs.

II. Recursive Helper: `addListHelper(one, pv1, two, pv2, res)`

This function handles alignment during the recursive descent (pre-computation) and performs the addition during the recursive ascent (post-computation).

Step 1: Base Case (Termination)

1. If both current nodes (one and two) are null, return $\mathbf{0}$ (no carry).

Step 2: Alignment (Pre-computation/Recursive Descent)

This section ensures that we traverse the longer list until its length matches the shorter one. The recursive call ensures the addition starts from the LSD.

1. **If $\mathbf{pv_1} > \mathbf{pv_2}$ (List 1 is currently longer):**
   - Recursively call the helper, moving to the next node in List 1 (`one.next`) but keeping the position of List 2 (`two`) constant.
   - The returned value (oc, or *old carry*) holds the carry from the subsequent addition.
   - Calculate the temporary sum: $\text{data} = \text{one.data} + \text{oc}$.
2. **If $\mathbf{pv_2} > \mathbf{pv_1}$ (List 2 is currently longer):**
   - Recursively call the helper, moving to the next node in List 2 (`two.next`) but keeping the position of List 1 (`one`) constant.
   - Calculate the temporary sum: $\text{data} = \text{two.data} + \text{oc}$.
3. **If $\mathbf{pv_1} = \mathbf{pv_2}$ (Lists are aligned):**
   - Recursively call the helper, moving to the next node in *both* lists.
   - Calculate the temporary sum: $\text{data} = \text{one.data} + \text{two.data} + \text{oc}$.

Step 3: Addition and Result Construction (Post-computation/Recursive Ascent)

This section executes after the recursive call returns (when we are moving back up the call stack, from LSD to MSD).

1. **Calculate New Digit and Carry:**
   - New Digit: $\text{nd} = \text{data} \pmod{10}$ (The digit for the current position).
   - New Carry: $\text{nc} = \text{data}/10$ (The carry to pass up to the parent call).
2. **Build Result:** Add the new digit to the result list: $\text{res.addFirst}(\text{nd})$.
3. **Return Carry:** Return nc.

Complexity Analysis

- **Time Complexity:** $O(\max(m, n))$, where m and n are the lengths of the two lists. The recursive calls effectively perform two full traversals (one to find the alignment and one to perform the addition), making the total time linear with respect to the total number of nodes.
- **Space Complexity:** $O(\max(m, n))$ due to the **recursion stack depth**, which is proportional to the length of the longer linked list.

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/addTwoLists.java

Intersection point of linkedlist

21 November 2025 18:33

Problem Description

Given the heads of two singly linked lists, `listA` and `listB`, find the node at which the two lists intersect. If the two linked lists have no intersection at all, return `null`.

Key Constraints & Conditions

1. **Singly Linked Lists:** The lists are singly linked.
2. **Structure:** The lists must retain their original structure after the function returns.
3. **Efficiency:** The solution should run in $O(m + n)$ time complexity and use only $O(1)$ extra space, where m and n are the lengths of `listA` and `listB`, respectively.
4. **Intersection Definition:** The intersection is defined by **reference**, not value. For instance, if the k^{th} node of `listA` is the exact same node object as the j^{th} node of `listB`, they intersect. After this point, they must share the entire tail segment.

Example

Example 1: Intersecting Lists

Input:

- `List A`: 4 -> 1 -> **8** -> 4 -> 5 -> null
- `List B`: 5 -> 6 -> 1 -> **8** -> 4 -> 5 -> null

Output: The node with value **8**.

Explanation: List A has a length of 5. List B has a length of 6. The lists merge at the node with value 8.

📜 Algorithm: Difference in Length Method

This algorithm works by normalizing the starting points of the two lists so they are **equidistant** from the potential intersection point (or the end of the lists).

1. Initialization and Length Calculation

1. Get the length of the first linked list (len_1) and the second linked list (len_2). (In your code, these are already available as `list1.size` and `list2.size`).
2. Initialize two traversal pointers, ptr_1 and ptr_2, to the heads of the two lists, head_1 and head_2.
3. Calculate the absolute difference in their lengths, $\text{diff} = |\text{len}_1 - \text{len}_2|$.

---

2. Alignment (Advancing the Longer List)

The goal is to move the starting pointer of the longer list forward by diff nodes.

1. **Identify the Longer List:** Compare len_1 and len_2.
2. **Advance the Pointer:**
   - If $\text{len}_1 > \text{len}_2$, advance ptr_1 by diff steps.
   - If $\text{len}_2 > \text{len}_1$, advance ptr_2 by diff steps.
   - If $\text{len}_1 = \text{len}_2$, no advancement is needed, as they are already aligned.

After this step, ptr_1 and ptr_2 are guaranteed to start traversing from points that are an equal number of steps away from the intersection node.

---

3. Simultaneous Traversal and Comparison

Traverse both lists simultaneously, comparing the pointer references at each step.

1. **Iterate:** Loop while both ptr_1 and ptr_2 are not `null`.
2. **Check for Intersection:** Inside the loop, check if ptr_1 is the same object as ptr_2 (`head1 == head2` in your code).
   - If they are the same, **return the node** (this is the intersection point).
3. **Advance:** Move both pointers one step forward: $\text{ptr}_1 = \text{ptr}_1.\text{next}$ and $\text{ptr}_2 = \text{ptr}_2.\text{next}$.

---

4. Termination

1. If the loop finishes (meaning one or both pointers reached `null` at the same time) without finding an intersection, **return** `null`.

📊 Complexity Analysis

- **Time Complexity:** $O(m + n)$
  - We calculate the lengths (which is $O(m + n)$).
  - We advance the longer pointer by diff steps (at most $O(\max(m, n))$).
  - We traverse the common section simultaneously (at most $O(\min(m, n))$).
  - Total time is dominated by traversing each list at most a constant number of times.
- **Space Complexity:** $O(1)$
  - Only a few extra pointers and integer variables (`len1`, `len2`, `diff`) are used, which is constant space regardless of the list sizes.

https://github.com/AlgoMagnet/0or1/blob/main/LinkedList/findIntersection.java