ETHNUS™

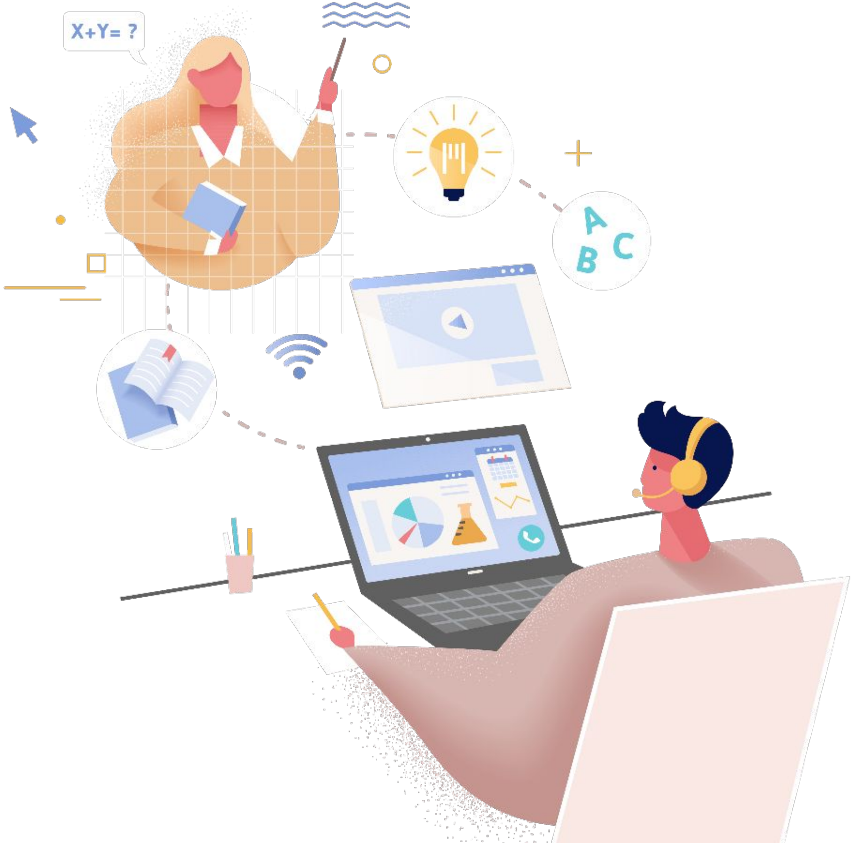
Explore | Expand | Enrich

<Codemithra />™

Merge Two Sorted Lists

TOPICS

- Introduction
- Algorithm Pseudocode
- Example
- Code
- Time and space complexity
- Advantages and disadvantages
- Interview Questions



Merge Two Sorted Lists

Definition: Sorting of Two Lists
"Sorting of two lists" refers to the process of arranging the elements of each list in a specific order, usually ascending or descending, either individually or as part of a combined operation such as merging.

If the Lists Are Already Sorted:

- You can **directly merge** them using an efficient algorithm like **two-pointer technique**.

If the Lists Are Unsorted:

- You need to **sort each list** first using algorithms like:
  - Merge Sort (best for linked lists)
  - Quick Sort (for arrays or arraylists)

- Then proceed with merging if needed.

Merge Two Sorted Lists

Merging two sorted linked lists means combining them into a single sorted linked list.
The input lists must already be sorted in ascending order.

Why is it useful?

- Common subroutine in merge sort.
- Frequently asked in coding interviews.
- Helps practice linked list traversal and pointer manipulation.

Algorithm:

Use two pointers, each for one list.
Compare current nodes:

- Append the smaller node to the result list.
- Move that list’s pointer forward.

If one list ends, append the remaining part of the other list.

Pseudo code

```
Function mergeSortedLists(list1, list2):
    Create a dummy node to act as the start
    Set current = dummy

    While list1 != null and list2 != null:
        If list1.data <= list2.data:
            current.next = list1
            list1 = list1.next
        Else:
            current.next = list2
            list2 = list2.next
        current = current.next

    If list1 is not null:
        current.next = list1
    Else:
        current.next = list2

    Return dummy.next
```

Input

```
List1: 1 → 3 → 5
List2: 2 → 4 → 6
```

Output

```
Merged List: 1 → 2 → 3 → 4 → 5 → 6
```

Example :

Input Lists:

- List 1: 1 → 3 → 5
- List 2: 2 → 4 → 6

Step-by-Step Merging Process:

1. Start comparing the first nodes of both lists: 1 (List1) and 2 (List2).
2. Since 1 < 2, add 1 to the merged list. Move the pointer of List1 to 3.
3. Compare 3 (List1) and 2 (List2).
4. Since 2 < 3, add 2 to the merged list. Move the pointer of List2 to 4.
5. Compare 3 (List1) and 4 (List2).
6. Since 3 < 4, add 3 to the merged list. Move List1 to 5.
7. Compare 5 (List1) and 4 (List2).
8. Since 4 < 5, add 4 to the merged list. Move List2 to 6.
9. Compare 5 (List1) and 6 (List2).
10. Since 5 < 6, add 5 to the merged list. Move List1 to null.
11. Now List1 is exhausted (null), so append the remaining nodes of List2 to the merged list — which is just 6.

Final Merged List:
1 → 2 → 3 → 4 → 5 → 6

```
import java.util.Scanner;

class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}

public class MergeSortedLists {

    static Node insert(Node head, int data) {
        Node newNode = new Node(data);
        if (head == null) return newNode;

        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
 temp.next = newNode;
        return head;
    }
```

```
static void printList(Node head) {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}

static Node merge(Node l1, Node l2) {
    Node dummy = new Node(0);
    Node current = dummy;

    while (l1 != null && l2 != null) {
        if (l1.data <= l2.data) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }
```

```
 if (l1 != null) current.next = l1;
        else current.next = l2;

        return dummy.next;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        Node head1 = null, head2 = null;


        System.out.print("Enter number of elements in
List 1: ");
        int n1 = sc.nextInt();
        System.out.println("Enter sorted elements for
List 1:");
        for (int i = 0; i < n1; i++) {

  int val = sc.nextInt();
            head1 = insert(head1, val);
        }
```

```
        System.out.print("Enter number of elements in List 2: ");
        int n2 = sc.nextInt();
        System.out.println("Enter sorted elements for List 2:");
        for (int i = 0; i < n2; i++) {
            int val = sc.nextInt();
            head2 = insert(head2, val);
        }

        System.out.print("List 1: ");
        printList(head1);
        System.out.print("List 2: ");
        printList(head2);

        Node mergedHead = merge(head1, head2);

        System.out.print("Merged List: ");
        printList(mergedHead);

        sc.close();
    }
}
```

Sample Input:

```
Enter number of elements in List 1: 3
Enter sorted elements for List 1:
1 3 5
Enter number of elements in List 2: 3
Enter sorted elements for List 2:
2 4 6
```

Output:

```
List 1: 1 3 5
List 2: 2 4 6
Merged List: 1 2 3 4 5 6
```

Code Explanation:

1. Node Class

- Defines a structure for a node in the linked list.
- Contains:
  - int data: to store the value.
  - Node next: to store reference to the next node.
- Constructor initializes data and sets next to null.

2. Insert(Node head, int data) Method

- Adds a node at the end of the list.
- If the list is empty (head == null), it returns a new node.
- Otherwise, traverses to the end and links the new node.

Returns the updated head of the list.

3. printList(Node head) Method

- Traverses the list starting from head.
- Prints the data of each node followed by a space.
- Ends with a newline.

4. merge(Node l1, Node l2) Method

- Merges two sorted linked lists into one sorted list.
- Uses a dummy node to simplify the logic.
- Uses a current pointer to build the merged list.
- Compares nodes of both lists and links the smaller one.
- If one list ends, appends the remaining part of the other list.
- Returns dummy.next, which is the head of the merged list.

5. Main Method Steps

- Uses Scanner to read user input.
- Takes size and elements of List 1, builds using insert().
- Takes size and elements of List 2, builds using insert().
- Prints both lists using printList().
- Calls merge() to merge the two lists.
- Prints the merged list.

Time Complexity: O(n + m)

- Let n be the length of the first linked list.
- Let m be the length of the second linked list.
- The algorithm iterates through **each node exactly once** from both lists.

Merging Process:

Each comparison takes constant time, and we perform (n + m) comparisons in the worst case.

Total Time Complexity: O(n + m)

Space Complexity:

Iterative Solution (as in our Java code):

- Uses only a few pointers (dummy, current, l1, l2) regardless of input size.
- No extra space is used for storing nodes or creating new ones (in-place merge).

Space Complexity: O(1)

Advantages:

1. **Efficient Time Complexity – O(n + m)**

- Merging is done in a single pass through both lists.
- Ideal for large sorted lists.

1. **In-Place Merge (O(1) Space)**

- No additional memory is needed (except for a few pointers).
- Nodes are reused, which saves space.

1. **Foundational Interview Concept**

- This problem is commonly asked in coding interviews.
- Helps assess understanding of linked lists and pointer manipulation.

Disadvantages

1. **Only Works if Lists Are Pre-Sorted**
   - The solution assumes both lists are sorted.
   - If not, the merged output will not be sorted.
2. **Singly Linked List Limitation**
   - Only applies to singly linked lists.
   - Additional logic is needed for doubly or circular lists.
3. **Cannot Handle Null/Invalid Inputs Gracefully**
   - No validation for invalid inputs (e.g., non-integer values or null heads).
   - Could throw exceptions if not handled properly.

QUESTION 1

Can the merge be done recursively?

Answer:

Yes, but it uses extra space on the call stack: O(n + m) space.

Explanation:

Yes, the merge of two sorted linked lists can be implemented **recursively** by:

- Comparing the head nodes of both lists.
- Recursively calling the function with the next node of the chosen list and the other list.

QUESTION 2

Do you need to create new nodes during the merge?

Answer:

No, we can reuse existing nodes and just rearrange the next pointers.

Explanation:

- Since the original lists are already sorted, there's **no need to create new nodes** or copy data.
- You just rearrange the next pointers so that one sorted list flows into the next.

QUESTION 3

What happens if one of the lists is null?

Answer:

Return the other list as the merged result.

Explanation:

- If one list is null, then the merged list is simply the other list because:
    - An empty list merged with a sorted list = the sorted list.
    - No elements to compare or rearrange.

QUESTION 4

Can this logic be extended to merge more than two sorted lists?

Answer:

Yes, but we need to use techniques like:

- ➢ Repeated pairwise merging (less efficient)
- ➢ Min-heap (efficient for K lists): Time = O(N log K)

THANK YOU