**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

# “Merge k Sorted Lists”

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

## BACHELOR OF ENGINEERING

## IN COMPUTER SCIENCE

**Submitted by**

**N. Baby Keerthi Reddy (192211433)**

**Under the Supervision of**

**K V Kanimozhi**

## DECLARATION

I, **N. Baby Keerthi Reddy, a** student of **Bachelor of Engineering in Computer Science Engineering** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, as a result of this, declare that the work presented in this Capstone Project Work entitled **"Merge k Sorted Lists"** is the outcome of my bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

(N. Baby Keerthi Reddy-192211433)

Date:

Place: Saveetha School of Engineering, Thandalam.

## CERTIFICATE

This is to certify that the project entitled **“Merge k Sorted Lists”** submitted by Baby Keerthi Reddy has been carried out under my supervision. The project has been submitted as per the requirements for the current semester of the B.E Computer Science Engineering and B.Tech Artificial Intelligence in Data Science.

Faculty-in-charge

K V Kanimozhi

**ABSTRACT**

In this project, we explore the problem of merging multiple sorted linked lists into one sorted linked list. The problem is a classic example of efficient algorithm design and optimization. The approach involves utilizing a priority queue (min-heap) to maintain the smallest current elements across the lists, merging them in ascending order. This solution is ideal for scenarios involving large-scale data that needs to be combined and sorted efficiently.

Merging sorted linked lists is a common task in systems that handle data streams, such as search engines, databases, and distributed systems. The efficient handling of multiple sorted lists, especially when the number of lists and their size increases, is critical to maintaining performance. By using a priority queue, we ensure that the merging process operates at optimal time complexity, reducing unnecessary comparisons. This project demonstrates the use of key data structures like heaps and linked lists, highlighting their importance in algorithmic problem-solving and system optimization. The final solution is both time-efficient and scalable, making it suitable for real-world applications involving large datasets.

**Keywords:** Linked Lists, Merge k Sorted Lists, Min-Heap, Priority Queue, Algorithm Optimization, Scalable Solutions

**INTRODUCTION**

The "Merge k Sorted Lists" problem is a fundamental task in data structures and algorithms. Given an array of k linked lists, where each list is sorted in ascending order, the objective is to merge all of them into a single sorted linked list. This problem arises frequently in many real-world applications, such as external sorting algorithms, data merging, and in systems where large datasets need to be combined in a sorted order efficiently.

In addition to its practical relevance, the "Merge k Sorted Lists" problem also serves as a great exercise in algorithm optimization, requiring the designer to balance space and time complexity. A naïve approach would involve repeatedly merging two lists at a time, leading to an inefficient solution, especially as the number of lists increases. More optimal solutions make use of advanced data structures, such as **min-heaps (priority queues)**, which help minimize unnecessary comparisons and maintain efficiency even as the number of lists grows.

For example, given k = 3 sorted linked lists:

* List 1: 1 -> 4 -> 5
* List 2: 1 -> 3 -> 4
* List 3: 2 -> 6

The goal is to merge these lists to produce the result:  
1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6.

This problem can be approached in several ways, but the most efficient method leverages a **priority queue** to ensure that at every step, the smallest element is selected and added to the merged list. This approach ensures a time complexity of **O(N log k)**, where N is the total number of elements across all lists and k is the number of lists. This makes the solution scalable and practical, even for large values of k and large lists, making it ideal for handling extensive datasets in real-world applications.

Furthermore, the problem provides an opportunity to explore the intricate workings of linked lists and priority queues, fundamental concepts in data structures. It highlights the power of **divide-and-conquer** strategies when faced with complex data manipulation tasks and underscores the importance of choosing the right algorithm for optimal performance.

**Divide and Conquer Method:**

**Divide:** Divide the given array into sub-arrays with the help of mid-value.

**Mid = ( start + end ) / 2**

**Conquer:** Repeat the process until the sub-array contains exactly one element.

{ i.e., The final step of dividing the array into sub-arrays should contain exactly one element.}

**Combine:** Sort and combine the two sub-arrays until the sub-arrays combine into one single array with sorted elements.

### CODING

### #include <stdio.h>

### #include <stdlib.h>

### struct ListNode {

### int val;

### struct ListNode\* next;

### };

### struct ListNode\* createNode(int val) {

### struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

### newNode->val = val;

### newNode->next = NULL;

### return newNode;

### }

### struct ListNode\* mergeTwoLists(struct ListNode\* l1, struct ListNode\* l2) {

### if (!l1) return l2;

### if (!l2) return l1;

### if (l1->val < l2->val) {

### l1->next = mergeTwoLists(l1->next, l2);

### return l1;

### } else {

### l2->next = mergeTwoLists(l1, l2->next);

### return l2;

### }

### }

### struct ListNode\* mergeKLists(struct ListNode\*\* lists, int listsSize) {

### if (listsSize == 0) return NULL;

### if (listsSize == 1) return lists[0];

### int mid = listsSize / 2;

### struct ListNode\* left = mergeKLists(lists, mid);

### struct ListNode\* right = mergeKLists(lists +mid, listsSize - mid);

### return mergeTwoLists(left, right);

### }

### void printList(struct ListNode\* head) {

### while (head) {

### printf("%d -> ", head->val);

### head = head->next;

### }

### printf("NULL\n");

### }

### int main() {

### // Example: creating the lists [[1, 4, 5], [1, 3, 4], [2, 6]]

### struct ListNode\* list1 = createNode(1);

### list1->next = createNode(4);

### list1->next->next = createNode(5);

### struct ListNode\* list2 = createNode(1);

### list2->next = createNode(3);

### list2->next->next = createNode(4);

### struct ListNode\* list3 = createNode(2);

### list3->next = createNode(6);

### struct ListNode\* lists[] = {list1, list2, list3};

### struct ListNode\* result = mergeKLists(lists, 3);

### printf("Merged list: ");

### printList(result);

### return 0;

### }

### OUTPUT

### 

**COMPLEXITY ANALYSIS**

The complexity of the "Merge k Sorted Lists" problem can be analyzed based on different scenarios: best case, worst case, and average case. The key to understanding the complexity lies in the usage of a priority queue (min-heap), which allows us to efficiently manage and merge elements from multiple sorted lists.

***Best Case: O(N log k)***

In the best-case scenario, the priority queue is used optimally, and the merging of the k lists happens efficiently. Here, N represents the total number of nodes across all the lists, and k represents the number of linked lists. Each insertion and removal operation in the priority queue takes O(log k) time due to the binary heap structure.

Since there are N elements to merge, and each element is inserted into and removed from the priority queue exactly once, the time complexity for processing each element is O(log k). Therefore, in the best case, where no additional delays or inefficiencies occur, the overall time complexity is **O(N log k)**.

***Worst Case: O(N log k)***

The worst-case complexity remains **O(N log k)**, which is similar to the best case. In the worst case, all N nodes are inserted and removed from the priority queue, with each operation taking O(log k) time due to the heap's structure. The worst case may occur when the input lists are distributed unevenly, with one list having far fewer elements than others. However, the efficiency of the min-heap ensures that even in the worst case, the merging process is handled optimally. Every insert and delete operation on the heap maintains a logarithmic time complexity in terms of k, as the heap maintains the smallest current elements from the lists.

Thus, the overall time complexity remains **O(N log k)**, where N is the total number of nodes, and k is the number of lists, even in the worst case.

***Average Case: O(N log k)***

On average, each element from the k lists is processed exactly once. When we use a min-heap, each element is pushed into and popped from the heap exactly once. Since the heap contains k elements at any given time (one from each list), the cost for each insertion or removal is **O(log k)**. Given that we are merging a total of N nodes, the total complexity is **O(N log k)**.

In practical scenarios, the lists are often balanced in terms of the number of elements. In such cases, the priority queue (min-heap) ensures that the smallest current element from all the lists is efficiently chosen and merged into the final sorted list, leading to an average case complexity of **O(N log k)**.

***Overall Complexity***

The overall time complexity of the algorithm is **O(N log k)**, where N is the total number of nodes across all linked lists, and k is the number of linked lists. The space complexity of the algorithm is **O(k)** because the priority queue will store at most k elements at any given time, one from each linked list.

* **Time Complexity**:
  + Insertion and removal from the heap take O(log k) time.
  + With N elements to process and each requiring a heap operation, the total time complexity is **O(N log k)**.
* **Space Complexity**:
  + The priority queue requires **O(k)** space to store the smallest element from each list during the merge process.

### CONCLUSION

The "Merge k Sorted Lists" problem demonstrates effective use of advanced data structures, such as priority queues, to optimize the process of merging multiple sorted linked lists into one. By utilizing a **min-heap**, the algorithm efficiently handles the merging process with a time complexity of **O(N log k)**, where N is the total number of nodes across all lists, and k is the number of lists. This approach not only minimizes the number of comparisons needed but also ensures that the smallest elements are consistently added to the merged list in each step.

This solution is highly scalable and well-suited for handling large datasets, making it ideal for real-world applications that require merging sorted sequences. Examples of such applications include external sorting, merging multiple sorted files in databases, and consolidating real-time data streams from multiple sources.

The efficient handling of multiple sorted inputs and maintaining the overall complexity within **O(N log k)** proves the power of choosing the right algorithmic approach. As data continues to grow in size and complexity, solutions like this become essential for maintaining performance and efficiency in modern computing environments. Thus, this project highlights the critical role that algorithm design and data structure optimization play in solving fundamental problems and improving computational efficiency in diverse applications.