23. Merge k Sorted Lists Heap (Priority Queue) **Linked List** Divide and Conquer **Merge Sort Leetcode Link** Hard

You are provided with an array that contains k sorted linked lists. Each linked-list in the array is sorted in ascending order. Your task

Problem Description

order and should include all the elements from the k linked-lists. Intuition

is to merge these k sorted linked-lists into a single sorted linked-list. The final linked-list should continue to be sorted in ascending

linked-lists.

The solution to this problem makes use of a min-heap (also known as a priority queue in some languages) to efficiently merge the k sorted linked-lists. Since each linked-list is sorted, the smallest element of the merge-list must be one of the k heads of the input

2. We go through the list of linked-lists, and if a linked-list is not empty, we insert its head into the min-heap. Since we are dealing with a linked-list, we only need a reference to the head node to access the entire list.

Here's the step-by-step intuition behind the solution:

3. We create a new dummy node that serves as the precursor to the merged linked-list, which we'll build one node at a time as we extract the smallest nodes from the min-heap.

1. First, we initialize a min-heap that will contain the current smallest node from each linked-list.

- 4. While the min-heap is not empty, we perform the following steps: Extract the smallest node from the min-heap (this is done efficiently for a min-heap since the smallest element is always the
- root). • If the extracted node has a next node, we insert the next node into the min-heap to replace the position of the extracted
- node. Append the extracted node to the merged linked-list by setting the next reference of the current node to this extracted
- Move the current node pointer forward to this extracted node, which is now the last node in the merged list.

Instead, at any point in time, we're only comparing the current smallest nodes of each list.

- node.
- from the dummy node's next node. 6. The use of the min-heap ensures that we're always processing nodes in ascending order since the heap is always sorted after

5. Once we've exhausted all the nodes (when the min-heap is empty), we've built the complete merged linked-list which starts

- each insertion and removal. This approach is efficient because it doesn't require us to look at each node in every list when we're appending the smallest node.
- Solution Approach The code provided is a direct implementation of the intuition behind the solution using Python's built-in heap functionality. Let's walk

1. The definition for the singly-linked list, ListNode, is provided. A __lt__ method is added to the ListNode class, which allows the comparison between two ListNode instances based on their val attribute. This is required for the heap to maintain the correct

2. The mergeKLists function is defined to take in the list of linked-lists. This function returns a merged sorted linked-list.

smallest element is always at the root, and heapify ensures that the invariant of the heap is maintained.

heappush. This ensures the next smallest node in that linked-list is now available for comparison.

sorted linked-list. This is the result that we return from the mergeKLists function.

the merged list.

property after the removal.

Consider the following linked lists:

1. Initialize Min-Heap: We start by creating an empty min-heap.

the next element from the same list to the min-heap (if there is one)

List 1: $1 \rightarrow 4 \rightarrow 5$

Step-by-step process:

through the implementation step by step:

order based on node values.

list, provided it is not a None. 4. The heapify function from the heapq module is called on pq, which transforms the list into a heap. In the context of a heap, the

5. A dummy node is created. This dummy node serves as the starting point of our merged linked-list. The dummy node does not hold

any data relevant to the final list but is used as a reference point. A cur pointer is also created to keep track of the current end of

We check if the node that was just popped off has a next node. If it does, this next node is then pushed onto the heap using

3. A pq (priority queue) is initialized which will act as our min-heap. We use a list comprehension to include the head of each linked-

- 6. A while loop is used to iterate until the heap pq is empty. • We use heappop to remove and return the smallest node from the heap. Remember, this operation maintains the heap
- The cur pointer's next is then set to this node, adding it to the final merged list. We then advance cur to point to the node we just added to the merged list. 7. Once the heap is empty, we have added all the nodes to our merged list. dummy next will be pointing to the head of the merged

By using a min-heap, the solution ensures that at every step, the node with the smallest value among the k lists is chosen and added

to the final list, which preserves the sorted order. The heap optimizes the process of finding the next smallest element, giving the

Example Walkthrough Let's take a simple example to illustrate the solution approach with k = 3 sorted linked lists:

algorithm a better time complexity compared to a naive approach of comparing every node in every list.

List 2: $1 \rightarrow 3 \rightarrow 4$ List 3: $2 \rightarrow 6$

2. Insert Initial Nodes into Min-Heap: We insert the head of each non-empty linked list into the min-heap. Min-heap: [1, 1, 2] (elements from List 1's head, List 2's head, and List 3's head) 3. Create Dummy Node: We create a dummy node as a placeholder for the merged list's head.

4. Merge Process: We continuously remove the smallest element from the min-heap and append it to the merged list, then we add

Extract min (1 from List 1), min-heap is now [1, 2, 4], and add the next element of List 1 (4) to min-heap.

Extract min (3 from List 2), min-heap is now [4, 4, 5], and add the next element of List 2 (4) to min-heap.

 Extract min (1 from List 2), min-heap is now [2, 4, 4], and add the next element of List 2 (3) to min-heap. Extract min (2 from List 3), min-heap is now [3, 4, 4], no element to add as List 3's next is null.

Merged list (in the process):

 $1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow ...$

 $1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 4 \rightarrow 5 \rightarrow 6$

1 from queue import PriorityQueue

self.val = val

def mergeKLists(self, lists):

for head in lists:

if head:

class ListNode:

class Solution:

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

45

46

47

Class definition for a singly-linked list node.

priority_queue = PriorityQueue()

dummy = current = ListNode()

current.next = node

return dummy.next

current = current.next

while not priority_queue.empty():

node = priority_queue.get()

priority_queue.put(head)

Get the node with the smallest value.

Merge k sorted linked lists and return it as one sorted list.

Adding the first node of each list to the priority queue.

Extract nodes from the priority queue and build the merged list.

Link the extracted node to the merged list and move the pointer.

// Connect the current node in the merged list to the smallest node.

// Comparator for the priority queue: it will prioritize the node with the smaller value.

// While the priority queue is not empty, extract the minimum element and link it to the current result list.

// Define a priority queue with the custom comparator to keep track of the nodes.

// Add the first node of each list to the priority queue if it is not null.

// If the extracted node has a next node, add it to the priority queue.

// Attach the minimum node to the current node of the result list and move forward.

priority_queue<ListNode*, vector<ListNode*>, decltype(compare)> min_heap(compare);

current.next = smallestNode;

current = current.next;

ListNode(): val(0), next(nullptr) {}

ListNode(int x) : val(x), next(nullptr) {}

return a->val > b->val;

for (ListNode* list_head : lists) {

min_heap.push(list_head);

// Create a dummy head for the result list.

ListNode* min_node = min_heap.top();

min_heap.push(min_node->next);

current_node->next = min_node;

ListNode* dummy_head = new ListNode();

ListNode* current_node = dummy_head;

if (list_head) {

while (!min_heap.empty()) {

if (min_node->next) {

min_heap.pop();

ListNode(int x, ListNode *next) : val(x), next(next) {}

ListNode* mergeKLists(vector<ListNode*>& lists) {

auto compare = [](ListNode* a, ListNode* b) {

return dummyHead.next;

// Move the current pointer forward.

// Return the merged list, skipping the dummy head.

Priority queue initialized to hold the list nodes.

:return: ListNode object that is the head of the merged sorted list.

def __init__(self, val=0, next=None):

Final merged list:

5. Completion: Once the min-heap is empty, all nodes have been added to the merged list.

Extract min (4 from List 2), continue the process until all elements are merged in the heap.

order throughout. **Python Solution**

By extracting the minimum element from the heap and maintaining each linked list's remaining nodes in the min-heap, we always

have access to the current smallest nodes that need to be compared. This is how we merge efficiently and maintain the ascending

self.next = next # This ensures the PriorityQueue can compare ListNode objects by their 'val' attribute. def __lt__(self, other): 10 return self.val < other.val</pre> 11 12

:param lists: A list of ListNode objects, where each ListNode is the head of a sorted linked list.

Creating a dummy node which will help in easily returning the head of the merged list.

```
37
38
               # If there's a next node in the list, add it to the priority queue.
39
                if node.next:
40
                    priority_queue.put(node.next)
41
```

```
Java Solution
    * Definition for singly-linked list.
   class ListNode {
       int value;
       ListNode next;
       ListNode() {}
       ListNode(int value) { this.value = value; }
       ListNode(int value, ListNode next) { this.value = value; this.next = next; }
10
11 }
12
   class Solution {
       public ListNode mergeKLists(ListNode[] lists) {
14
           // Initialize a min-heap (priority queue) to hold nodes and sort them by their value.
15
           PriorityQueue<ListNode> priorityQueue = new PriorityQueue<>((node1, node2) -> node1.value - node2.value);
16
           // Add the first node of each list to the priority queue, if it is not null.
18
           for (ListNode head : lists) {
19
               if (head != null) {
20
                   priorityQueue.offer(head);
21
22
23
24
25
           // Create a dummy node that will serve as the head of the merged list.
26
           ListNode dummyHead = new ListNode();
27
           // A pointer to track the current node's position in the merged list.
28
           ListNode current = dummyHead;
           // Continue merging until the priority queue is empty.
           while (!priorityQueue.isEmpty()) {
31
               // Poll the priority queue to get the node with the smallest value.
               ListNode smallestNode = priorityQueue.poll();
33
34
35
               // If the smallest node has a next node, add it to the priority queue.
               if (smallestNode.next != null) {
36
                   priorityQueue.offer(smallestNode.next);
37
38
39
```

44 current_node = current_node->next; 45 46 // The next node of dummy_head points to the head of the merged list. Return this node. 47 return dummy_head->next; 48

```
49
50 };
51
Typescript Solution
 1 // The ListNode class represents each node of a singly-linked list with a value and a link to the next node.
   class ListNode {
       val: number;
       next: ListNode | null;
       constructor(val?: number, next?: ListNode | null) {
           this.val = val === undefined ? 0 : val;
           this.next = next === undefined ? null : next;
 9
10 }
11
12
    * Merges k sorted linked lists into one sorted linked list and returns its head.
    * Uses a minimum priority queue to determine the next smallest element to be added to the merged list.
    * @param lists - An array of ListNode instances representing the heads of k sorted linked lists
    * @returns A ListNode instance representing the head of the merged sorted linked list, or null if all lists are empty
17
    */
   function mergeKLists(lists: Array<ListNode | null>): ListNode | null {
       // Initialize a minimum priority queue with a custom priority comparator based on ListNode's value
19
       const minPriorityQueue = new MinPriorityQueue({ priority: (node: ListNode) => node.val });
20
21
22
       // Enqueue the head of each non-empty list into the priority queue
23
       for (const head of lists) {
           if (head) {
24
               minPriorityQueue.enqueue(head);
26
27
28
       // Create a dummy head for the merged list
29
       const dummyHead = new ListNode();
30
       // 'current' keeps track of the last node in the merged list
31
32
       let current: ListNode = dummyHead;
33
       // Continue combining nodes until the priority queue is empty
34
       while (!minPriorityQueue.isEmpty()) {
35
           // Dequeue the smallest element and add it to the merged list
36
37
            const node = minPriorityQueue.dequeue().element;
           current.next = node;
39
           current = current.next;
           // If there is a next node in the dequeued element's list, enqueue it
           if (node.next) {
               minPriorityQueue.enqueue(node.next);
44
45
46
```

Time Complexity

return dummyHead.next;

Time and Space Complexity

48

50

49 }

// Return the merged list, which starts at dummyHead's next node

data structure, along with the initial construction of that queue.

• While loop: The while loop, runs until the priority queue is empty. In the worst case, this will be equal to the total number of nodes in all input lists, which can be represented as n*k (if each list has n nodes).

time complexity is 0(n*k*log k).

Thus, the space complexity is O(k).

• Heappop: Each heappop operation has a time complexity of O(log k) since, in the worst case, it may have to heapify a tree with k nodes after removing the smallest element (root of the heap). • Heappush: Each heappush operation also has a time complexity of O(log k) because it must maintain the heap property after

inserting a new element into the heap. Given there are n*k elements to process in total and for each we potentially have a heappop and heappush operation, the overall

The time complexity of the given code primarily involves the operations of heappush and heappop on the priority queue (min-heap)

• Heapify: The initial heapification of the list pq, which contains at most k nodes (where k is the number of linked lists), has a

complexity of O(k) because it is a linear time operation for building a heap from an existing list of n elements.

Space Complexity

• Heap: The space complexity is determined by the size of the heap, which at maximum can hold k elements (one from each

linked list). Hence, the space complexity is 0(k). • Output Linked List: The space for the output linked list does not count toward the space complexity since space complexity is typically analyzed with respect to additional space required by the algorithm, not including space needed for the output.

// Definition for singly-linked list. struct ListNode { int val;

C++ Solution

class Solution {

};

public:

ListNode *next;

40

41

42

43

45

46

47

48

50

49 }

8 };

13

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43