

Merge K Sorted Lists

Always pick the global minimum — Min-Heap across K list heads ■

● HARD

■ Hook

Database merge operations. External sorting of huge files. Combining sensor streams in real time. All of these need to merge K sorted sequences efficiently. Merging 2 lists is easy (two pointers). But K lists? A **min-heap keeps track of the current minimum across all K list heads simultaneously** — this is the canonical $O(N \log K)$ solution.

■ Core Intuition — Always Extract the Global Minimum

Load the head (first node) of each list into a min-heap. The heap root is always the smallest element across all K lists. Extract it, append to result, and push the next node from that same list. Repeat until heap is empty.

■ Why this is $O(N \log K)$

N = total nodes across all lists. K = number of lists. Each node is pushed once and popped once: $2N$ heap operations. Each heap operation costs $O(\log K)$ since heap size $\leq K$. Total: $O(N \log K)$. Compare to naive: $O(NK)$.

■ Visual — Merge 3 Lists

```
List1: 1 → 4 → 5
List2: 1 → 3 → 4
List3: 2 → 6
INIT heap with heads: [(1,L1), (1,L2), (2,L3)]
Step 1: Extract (1,L1) → result=[1] Push L1.next=4 → heap=[(1,L2), (2,L3), (4,L1)]
Step 2: Extract (1,L2) → result=[1,1] Push L2.next=3 → heap=[(2,L3), (3,L2), (4,L1)]
Step 3: Extract (2,L3) → result=[1,1,2] Push L3.next=6 → heap=[(3,L2), (4,L1), (6,L3)]
]
Step 4: Extract (3,L2) → result=[1,1,2,3] Push L2.next=4 → heap=[(4,L1), (4,L2), (6,L3)]
]
Step 5: Extract (4,L1) → result=[...,4] Push L1.next=5 → heap=[(4,L2), (5,L1), (6,L3)]
]
Step 6: Extract (4,L2) → result=[...,4] L2.next=null → heap=[(5,L1), (6,L3)]
Step 7: Extract (5,L1) → L1.next=null → heap=[(6,L3)]
Step 8: Extract (6,L3) → L3.next=null → heap=[] DONE
★ RESULT = 1→1→2→3→4→4→5→6
```

■ Code Implementation (Java)

```
import java.util.PriorityQueue;
public class MergeKLists {
    // ListNode definition
    static class ListNode {
        int val; ListNode next;
        ListNode(int v) { val = v; }
    }
    public ListNode mergeKLists(ListNode[] lists) {
        // Min-heap ordered by node value
        PriorityQueue<ListNode> minHeap =
            new PriorityQueue<>((a, b) -> a.val - b.val);
        // Add head of each non-empty list
        for (ListNode head : lists)
            if (head != null) minHeap.offer(head);
        ListNode dummy = new ListNode(0); // sentinel head
        ListNode curr = dummy;
        while (!minHeap.isEmpty()) {
            ListNode node = minHeap.poll(); // global minimum
            curr.next = node;
            curr = curr.next;
            if (node.next != null) // push next from same list
                minHeap.offer(node.next);
        }
        return dummy.next;
    }
}
```

■ Time & Space Complexity

■ TIME COMPLEXITY

$O(N \log K)$

■ SPACE COMPLEXITY

$O(K)$

N = total number of nodes. Each pushed/popped once: $O(N)$ ops $\times$ $O(\log K)$ each = $O(N \log K)$. Heap holds at most K nodes at any time $\rightarrow O(K)$ space.

■ Common Mistakes

■ Not using a dummy/sentinel head node

Building a linked list without a dummy head requires special-casing the first node. Always use: `ListNode dummy = new ListNode(0); curr = dummy;` — it simplifies the code immensely.

■ Pushing null nodes into the heap

Always check: `if (node.next != null)` before offering to heap. Pushing null causes `NullPointerException` in the comparator.

■ Practice Problems

#	Problem	Diff	Pattern
1	Merge K Sorted Lists (LC 23)	Hard	Min-Heap
2	Merge K Sorted Arrays	Medium	Min-Heap
3	Kth Smallest Element in a Sorted Matrix (LC 378)	Medium	Min-Heap
4	Find K Pairs with Smallest Sums (LC 373)	Medium	Min-Heap
5	Smallest Range Covering Elements from K Lists (LC 632)	Hard	Min-Heap + Max tracking

■ Quick Revision

- ✓ Load all K list heads into a min-heap
- ✓ Repeatedly: poll min → append to result → push next node from same list
- ✓ Use dummy head node to simplify linked list construction
- ✓ Always null-check before pushing next node
- ✓ Time: $O(N \log K)$ where N =total nodes, K =number of lists
- ✓ Space: $O(K)$ for heap

■ MENTAL MODEL — Remember This Forever

Think of K sorted conveyor belts, each delivering items in order. You have K slots in a 'priority dispatcher' — one slot per belt, always showing the belt's current front item. You always grab from the slot showing the smallest item, then that belt advances one step. The dispatcher (min-heap) always has exactly K items — one per active belt.