<https://leetcode.com/problems/merge-k-sorted-lists/description/>

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

*Merge all the linked-lists into one sorted linked-list and return it.*

**Example 1:**

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

Explanation: The linked-lists are:

[

  1->4->5,

  1->3->4,

  2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

**Example 2:**

Input: lists = []

Output: []

**Example 3:**

Input: lists = [[]]

Output: []

**Constraints:**

k == lists.length

0 <= k <= 10^4

0 <= lists[i].length <= 500

-10^4 <= lists[i][j] <= 10^4

lists[i] is sorted in **ascending order**.

The sum of lists[i].length will not exceed 10^4.

**Attempt 1: 2023-02-19**

**Solution 1:  Priority Queue Solution (30 min)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\*    int val;

\*    ListNode next;

\*    ListNode() {}

\*    ListNode(int val) { this.val = val; }

\*    ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

    public ListNode mergeKLists(ListNode[] lists) {

        if(lists == null || lists.length == 0) {

            return null;

        }

        ListNode dummy = new ListNode();

        ListNode iter = dummy;

        PriorityQueue<ListNode> minPQ = new PriorityQueue<ListNode>(lists.length, (a, b) -> a.val - b.val);

        for(ListNode listHead : lists) {

            if(listHead != null) {

                minPQ.offer(listHead);

            }

        }

        while(!minPQ.isEmpty()) {

            ListNode curPeek = minPQ.poll();

            iter.next = curPeek;

            iter = iter.next;

            if(curPeek.next != null) {

                minPQ.offer(curPeek.next);

            }

        }

        return dummy.next;

    }

}

Time complexity:O(Nlog⁡k) where k is the number of linked lists.

- The comparison cost will be reduced to O(log⁡k) for every pop and insertion to priority queue. But finding the node with the smallest value just costs O(1) time.

- There are N nodes in the final linked list.

Space complexity:O(n) Creating a new linked list costs O(n)space.

- O(k) The code above present applies in-place method which cost O(1) space. And the priority queue (often implemented with heaps) costs O(k) space (it's far less than N in most situations).

**Refer to**

<https://leetcode.com/problems/merge-k-sorted-lists/solutions/10528/a-java-solution-based-on-priority-queue/>

public class Solution {

    public ListNode mergeKLists(List<ListNode> lists) {

        if (lists==null||lists.size()==0) return null;

        PriorityQueue<ListNode> queue= new PriorityQueue<ListNode>(lists.size(),new Comparator<ListNode>(){

            @Override

            public int compare(ListNode o1,ListNode o2){

                if (o1.val<o2.val)

                    return -1;

                else if (o1.val==o2.val)

                    return 0;

                else

                    return 1;

            }

        });

        ListNode dummy = new ListNode(0);

        ListNode tail=dummy;

        for (ListNode node:lists)

            if (node!=null)

                queue.add(node);

        while (!queue.isEmpty()){

            tail.next=queue.poll();

            tail=tail.next;

            if (tail.next!=null)

                queue.add(tail.next);

        }

        return dummy.next;

    }

}

-------------------------------------------------------------------------------------------------------

Using Java 8:-

class Solution {

    public ListNode mergeKLists(ListNode[] lists) {

        if (lists==null || lists.length==0) return null;

        PriorityQueue<ListNode> queue= new PriorityQueue<ListNode>(lists.length, (a,b)-> a.val-b.val);

        ListNode dummy = new ListNode(0);

        ListNode tail=dummy;

        for (ListNode node:lists)

            if (node!=null)

                queue.add(node);

        while (!queue.isEmpty()){

            tail.next=queue.poll();

            tail=tail.next;

            if (tail.next!=null)

                queue.add(tail.next);

        }

        return dummy.next;

    }

}

**Refer to**

<https://leetcode.com/problems/merge-k-sorted-lists/solutions/127466/merge-k-sorted-list/>

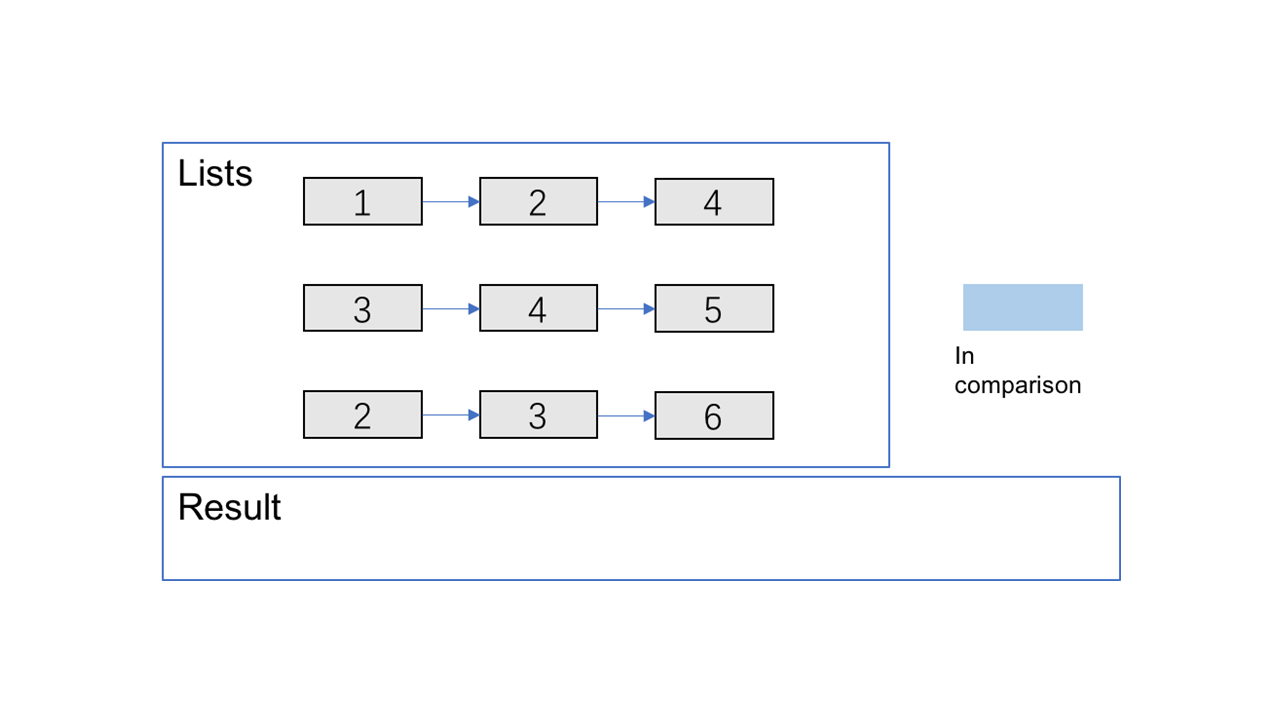
**Approach 2: Compare one by one**

**Algorithm**

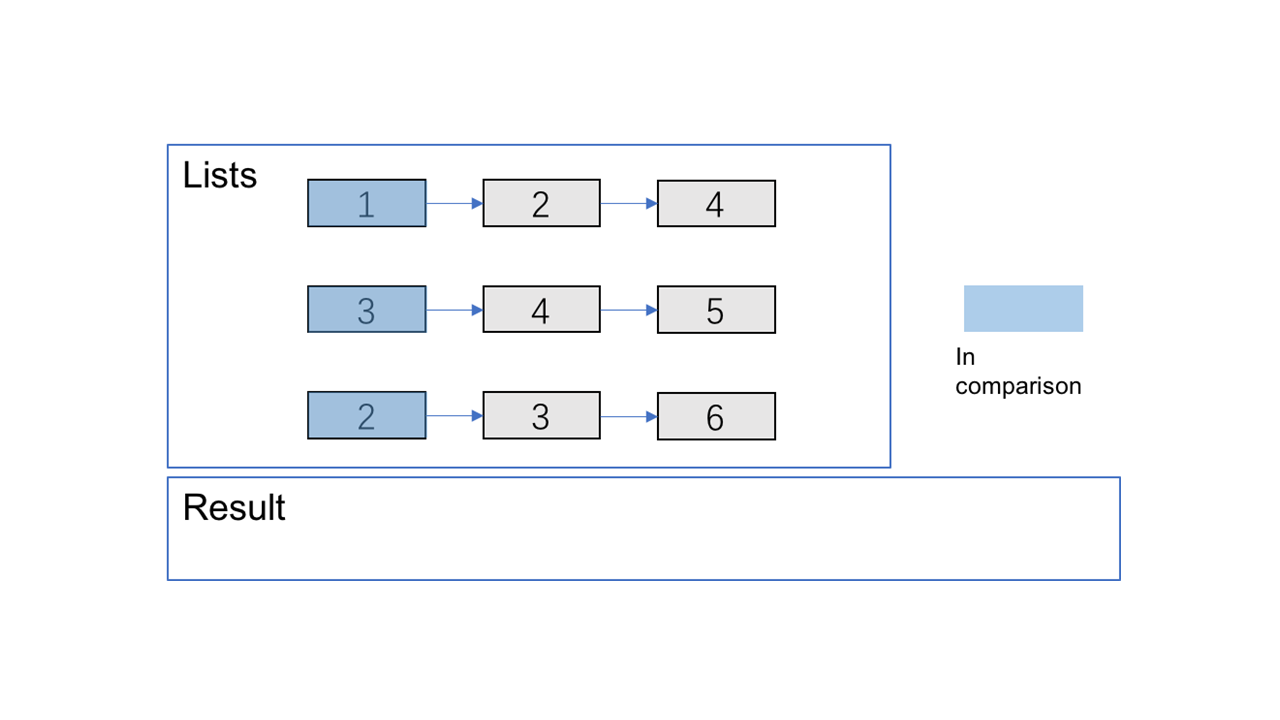
Compare every k nodes (head of every linked list) and get the node with the smallest value.

Extend the final sorted linked list with the selected nodes.

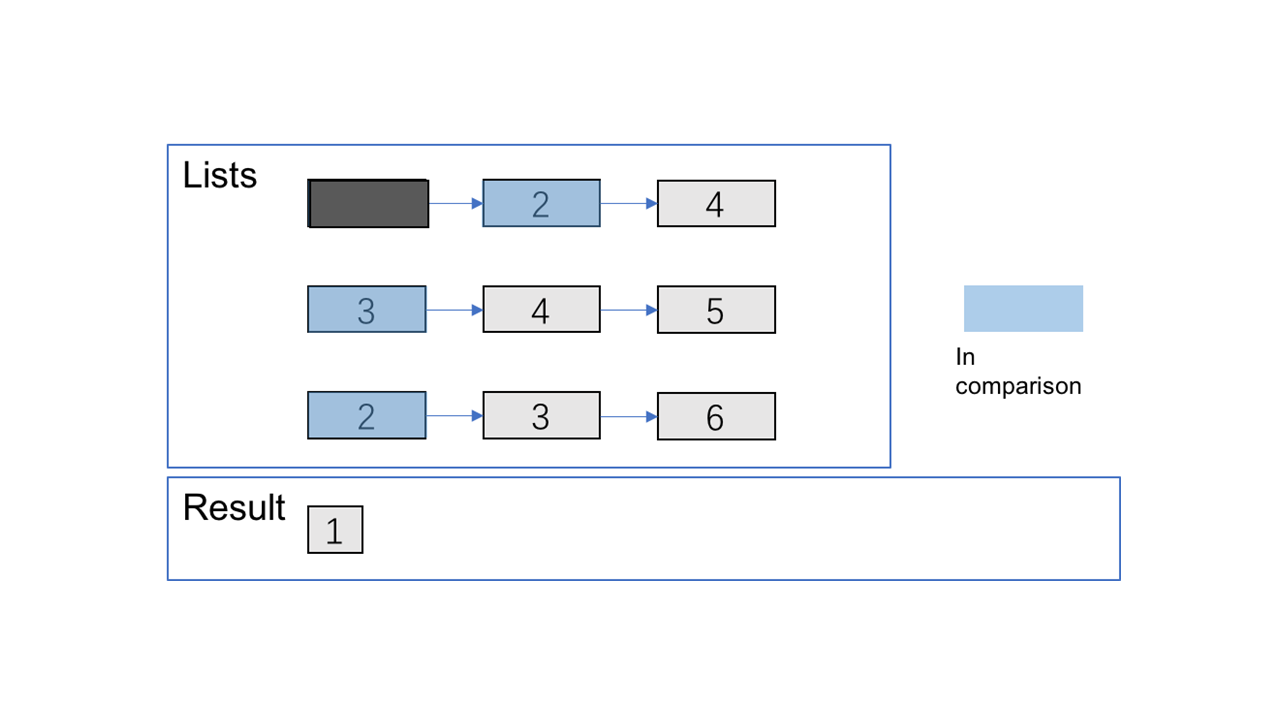
Step 1



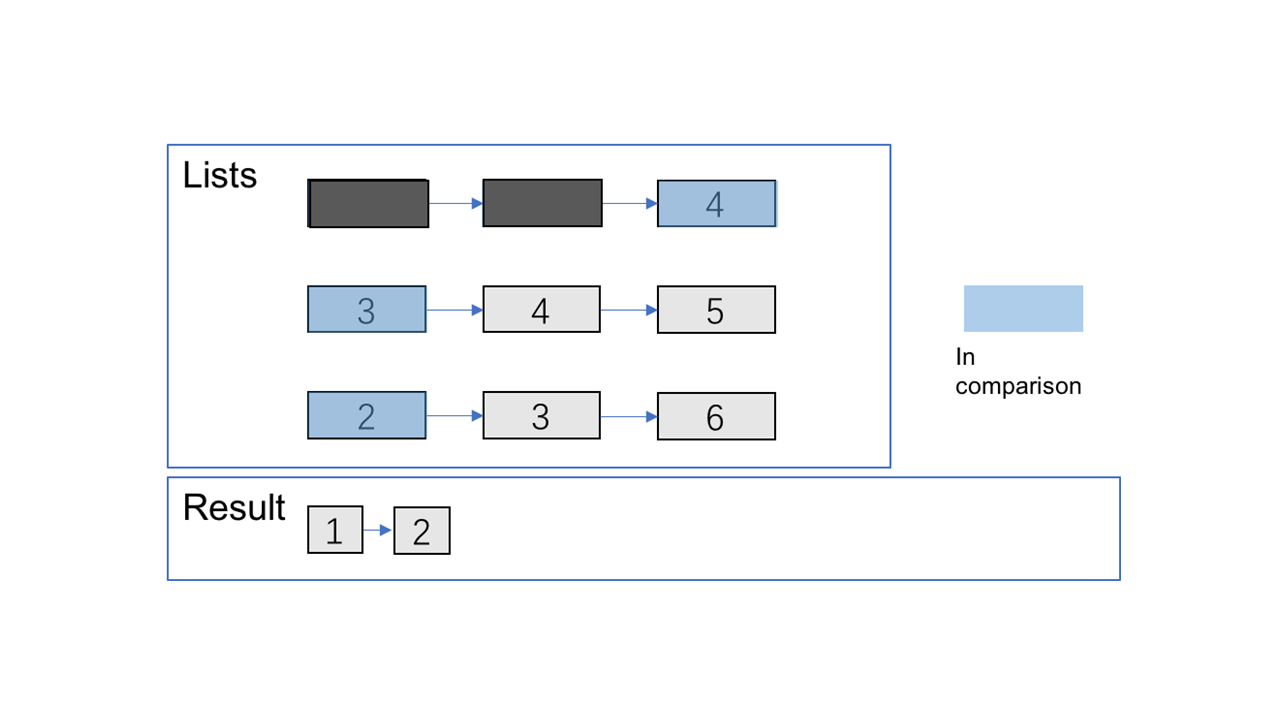
Step 2



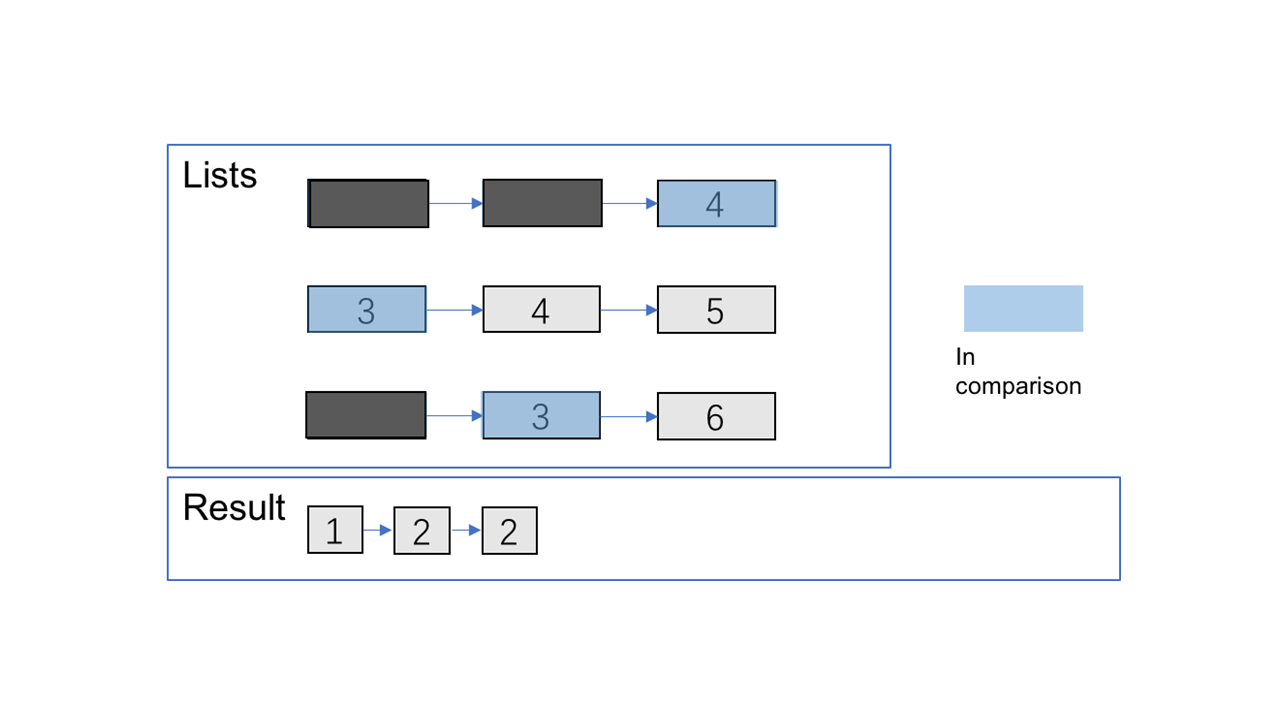
Step 3



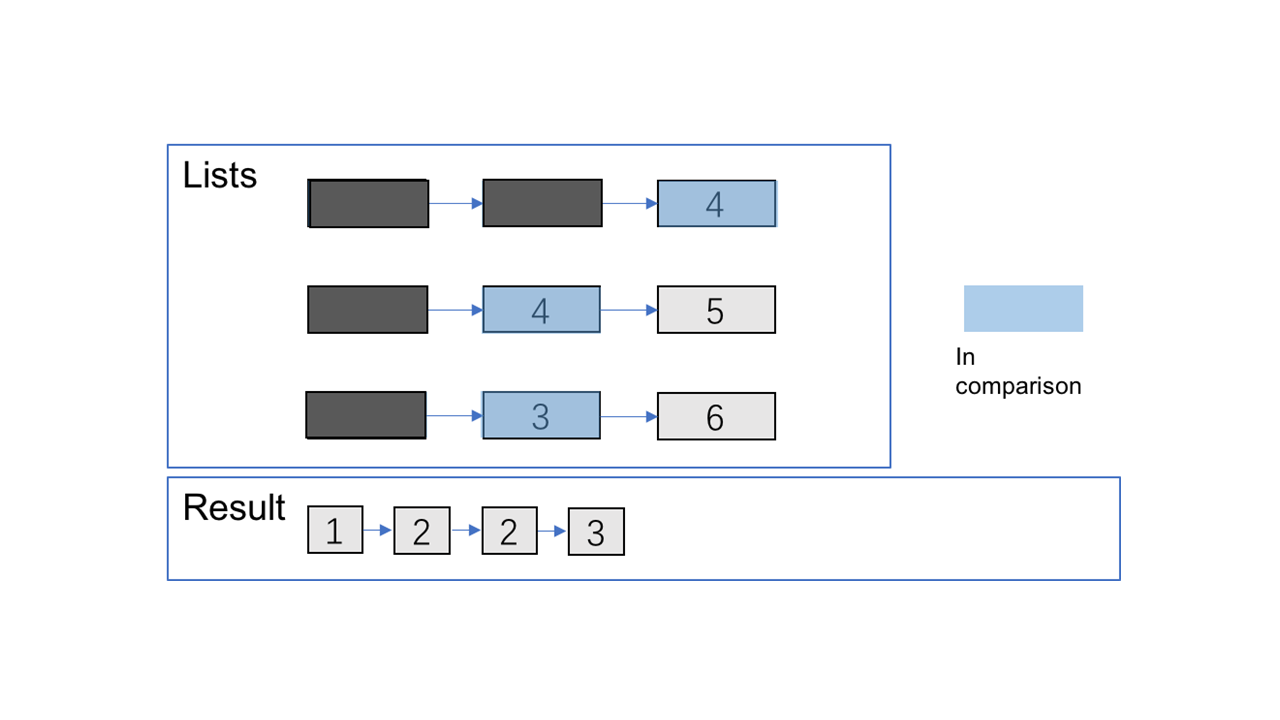
Step 4



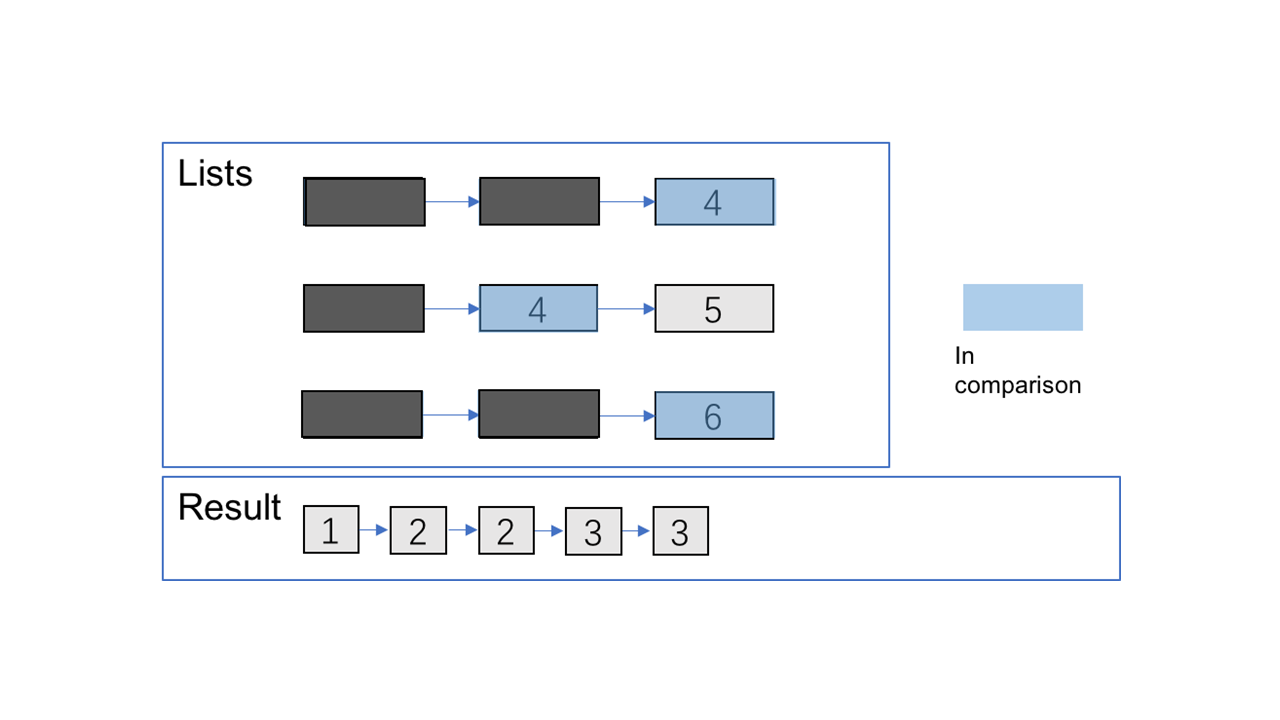
Step 5



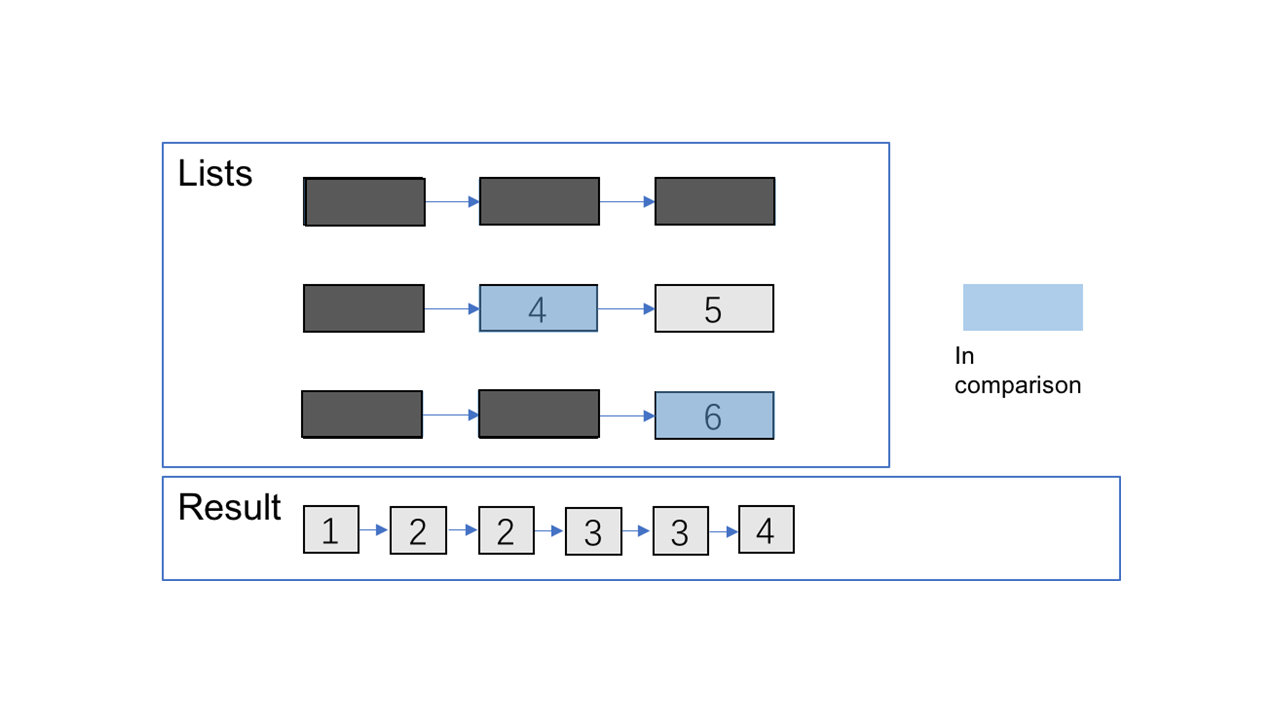
Step 6



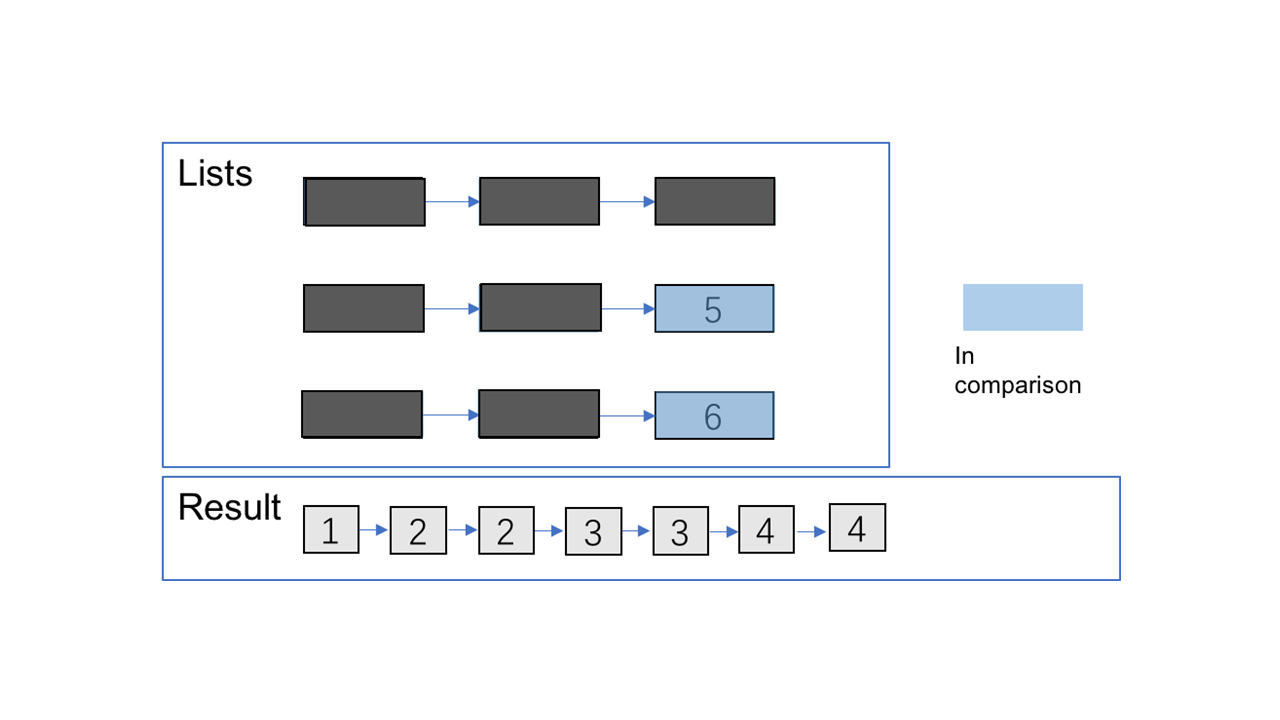
Step 7



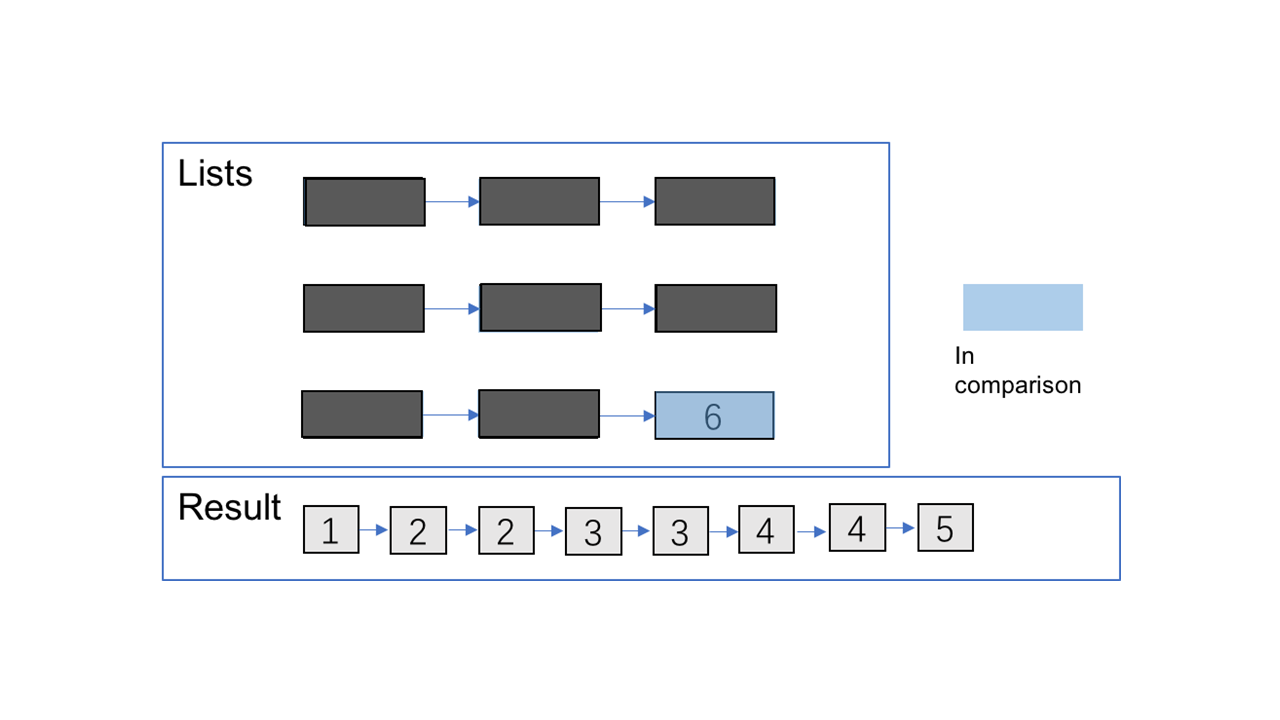
Step 8



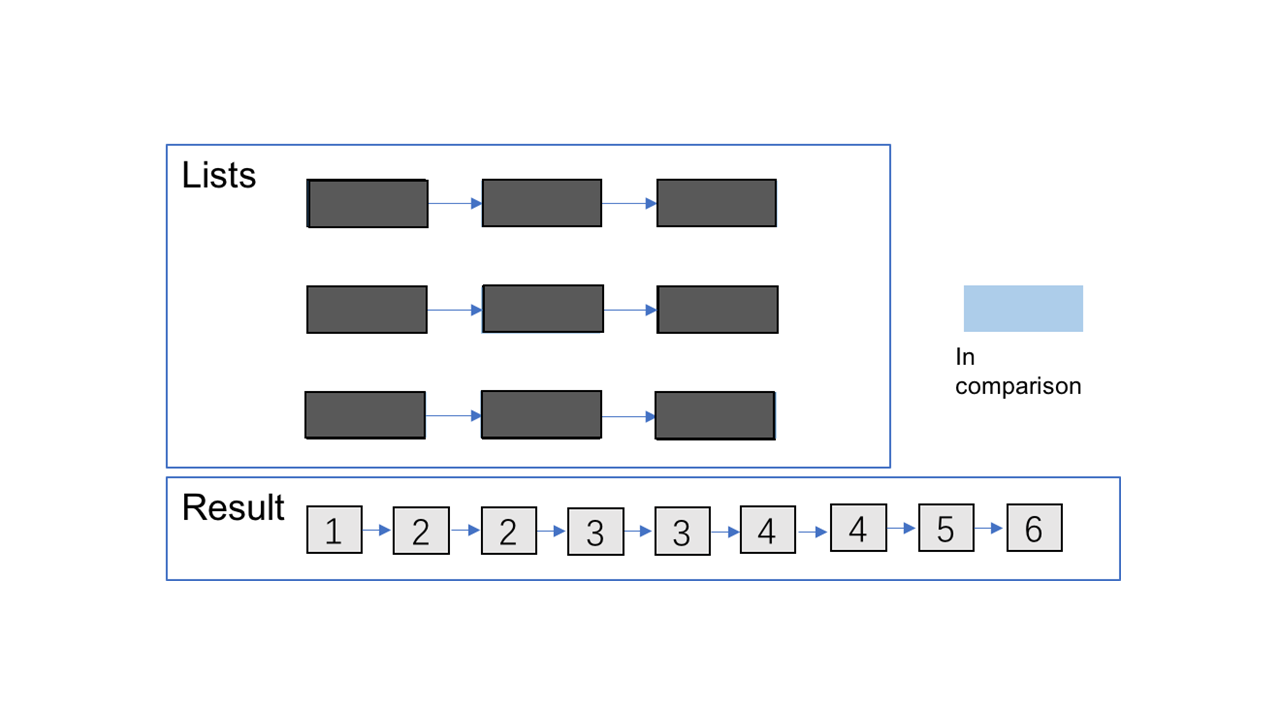
Step 9



Step 10



Step 11



**Approach 3: Optimize Approach 2 by Priority Queue**

**Algorithm**

Almost the same as the one above but optimize the **comparison process** by **priority queue**. You can refer [here](https://en.wikipedia.org/wiki/Priority_queue) for more information about it.

from Queue import PriorityQueue

class Solution(object):

    def mergeKLists(self, lists):

        """

        :type lists: List[ListNode]

        :rtype: ListNode

        Note: this is Python2 code

        """

        head = point = ListNode(0)

        q = PriorityQueue()

        for l in lists:

            if l:

                q.put((l.val, l))

        while not q.empty():

            val, node = q.get()

            point.next = ListNode(val)

            point = point.next

            node = node.next

            if node:

                q.put((node.val, node))

        return head.next

**Complexity Analysis**

Time complexity : O(Nlog⁡k) where k is the number of linked lists.

The comparison cost will be reduced to O(log⁡k)for every pop and insertion to priority queue. But finding the node with the smallest value just costs O(1) time.

There are N nodes in the final linked list.

Space complexity :

O(n) Creating a new linked list costs O(n)space.

O(k) The code above present applies in-place method which cost O(1) space. And the priority queue (often implemented with heaps) costs O(k) space (it's far less than N in most situations).