31 合并K个排序链表

更新时间：2019-09-19 10:24:08

> 知识犹如人体的血液一样宝贵。
>
> ——高士其

刷题内容

> **难度: Hard**
>
> 原题连接：https://leetcode-cn.com/problems/merge-k-sorted-lists/

内容描述

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

输入:
[
  1->4->5,
  1->3->4,
  2->6
]
输出: 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6

题目详解

题目给我们一个列表，列表当中每一个元素都是一个链表的头部，且每一个链表都是排序过的。题目需要我们将这个列表当中所有的排序链表全部合并成一个排序链表并返回其头部

解题方案

思路 1：时间复杂度: O(NlgK) 空间复杂度: O(K)

这道题首先我们需要将所有的排序链表合并，直接合并必然不是一个好的选择。因为我们没有利用 每一个链表都是排序的 这个条件。那应该怎么做呢？

应该做的是每次取来一个，然后再把应该的下一个放入。一个最基本的思考就是我们将这k个链表的头结点都放在一个最小堆当中，然后每次取出一个值最小的结点a，如果结点a的next b仍然存在，就将b放进最小堆，以此类推，直到我们的最小堆被取空。

时间复杂度上的 N 代表所有结点的个数，K 是链表的个数。下面我们来看代码：

Python beats 96.50%

```python
from heapq import heappush, heappop

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        node_pools = [] # 这就是我们的最小堆
        lookup = collections.defaultdict(list)
        for head in lists:
            if head:
                heappush(node_pools, head.val)
                lookup[head.val].append(head)
        dummy = cur = ListNode(None)
        while node_pools: # 只要最小堆当中还有结点，我们就继续下去
            smallest_val = heappop(node_pools)
            smallest_node = lookup[smallest_val].pop(0)
            cur.next = smallest_node
            cur = cur.next
            if smallest_node.next: # 如果结点a的next b仍然存在，就将b放进最小堆
                heappush(node_pools, smallest_node.next.val)
                lookup[smallest_node.next.val].append(smallest_node.next)
        return dummy.next
```

上面每次我们都需要pop出一个元素，这些元素可能是在array的中间，这样的话就会消耗很多时间，我们能否做一些优化呢？

这里由于其他语言的数据结构中没有直接的最小堆的概念，我们直接在思路二中来写，他们多被称为优先队列

思路 2：时间复杂度: O(NlgK) 空间复杂度: O(K)

刚才我们每次都需要pop出一个list中最左边的一个结点，这样其实可能会挺慢的，因为取出array中的第一个元素需要我们将后面的所有元素都向前移动一格。所以我们可以用deque来代替这个list，这样pooleft就会更快

其他的代码逻辑都和思路一是一模一样的

Python beats 96.50%

```python
from heapq import heappush, heappop

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        node_pools = [] # 这就是我们的最小堆
        lookup = collections.defaultdict(collections.deque)
        for head in lists:
            if head:
                heappush(node_pools, head.val)
                lookup[head.val].append(head)
        dummy = cur = ListNode(None)
        while node_pools: # 只要最小堆当中还有结点，我们就继续下去
            smallest_val = heappop(node_pools)
            smallest_node = lookup[smallest_val].popleft()
            cur.next = smallest_node
            cur = cur.next
            if smallest_node.next: # 如果结点a的next b仍然存在，就将b放进最小堆
                heappush(node_pools, smallest_node.next.val)
                lookup[smallest_node.next.val].append(smallest_node.next)
        return dummy.next
```

很显然我们得到了一些优化，时间减少了一些

java beats 75.23 %

```java
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        //重载比较函数，PriorityQueue是小顶堆，越小越先出
        PriorityQueue<ListNode> Q = new PriorityQueue<ListNode>(new Comparator<ListNode>() {
            public int compare(ListNode e1, ListNode e2) {
                return e1.val - e2.val;
            }
        });
        ListNode dummy = new ListNode(-1);
        ListNode p = dummy;
        for (int i = 0;i < lists.length;i++) {
            if (lists[i] != null) {
                Q.add(lists[i]);
            }
        }
        while (Q.size() > 0) {
            //取当前的最小值
            ListNode least = Q.poll();
            ListNode next = least.next;
            p.next = least;
            p = p.next;
            //如果当前的最小值还有下一个节点，则加入到优先队列
            if (next != null) {
                Q.add(next);
            }
        }
        return dummy.next;
    }
}
```

c++ beats 73.95 %

```
class Solution {
public:
    //重载比较函数，c++的优先队列是大顶堆，我们需要变成小顶堆
    struct cmp {
        bool operator()(const ListNode* x, const ListNode* y) {
            return x->val > y->val;
        }
    };
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        ListNode* dummy = new ListNode(-1);
        ListNode* p = dummy;
        priority_queue<ListNode*, vector<ListNode*>, cmp> q;
        for (int i = 0;i < lists.size();i++) {
            if (lists[i] != NULL) {
                q.push(lists[i]);
            }
        }
        while (!q.empty()) {
            //取当前的最小值
            ListNode* next = q.top();
            q.pop();
            p->next = next;
            p = p->next;
            next = next->next;
            //如果当前的最小值还有下一个节点，则加入到优先队列
            if (next != NULL) {
                q.push(next);
            }
        }
        return dummy->next;
    }
};
```

go beats 52.09 %

```go
//以下是go语言自定义优先队列元素的模板
type Queue []*ListNode
func (q *Queue) Len() int {
    return len(*q)
}
func (q *Queue) Less(i, j int) bool {
    return (*q)[i].Val < (*q)[j].Val
}
func (q *Queue) Swap(i, j int) {
    (*q)[i], (*q)[j] = (*q)[j], (*q)[i]
}
func (q *Queue) Push(x interface{}) {
    *q = append(*q, x.(*ListNode))
}
func (q *Queue) Pop() interface{} {
    n := len(*q)
    e := (*q)[n - 1]
    *q = (*q)[:n - 1]
    return e
}
//以上是go语言自定义优先队列元素的模板

func mergeKLists(lists []*ListNode) *ListNode {
    //由于lists有空链表，所以不能直接&Queue{lists}
    q := new(Queue)
    for _, list := range lists {
        if list != nil {
            heap.Push(q, list)
        }
    }
    dummy := &ListNode{-1, nil}
    p := dummy
    for q.Len() > 0 {
        next := heap.Pop(q).(*ListNode)
        //next是当前优先队列中Val值最小的，直接链接再dummy后面
        p.Next = next
        p = p.Next
        next = next.Next
        //如果next还有值，继续进有限队列
        if next != nil {
            heap.Push(q, next)
        }
    }
    return dummy.Next
}
```

总结

只要知道能够减少时间或者空间的方式，哪怕它并不是一个很大的突破，我们在面试的时候就一定要告诉面试官，因为不说他不知道， 你说了但是别人没说的话，你就比别人强一些

}

30 正则表达式匹配

32 K 个一组翻转链表