一维数据合并篇

**合并是一维数据结构中很常见的操作，通常是排序，分布式算法中的子操作。**

**这篇总结主要介绍LeetCode中关于合并的几个题目：**

**1.** [**Merge Sorted Array**](http://blog.csdn.net/linhuanmars/article/details/19712333)

**2.** [**Merge Two Sorted Lists**](http://blog.csdn.net/linhuanmars/article/details/19712593)

**3.** [**Sort List**](http://codeganker.blogspot.com/2014/03/sort-list-leetcode.html)

**4.** [**Merge k Sorted Lists**](http://blog.csdn.net/linhuanmars/article/details/19899259)

**1. Merge Sorted Array，思路比较明确，就是维护三个index，分别对应数组A，数组B，和结果数组。然后A和B同时从后往前扫，每次迭代中A和B指向的元素大的便加入结果数组中，然后index-1，另一个不动。这里从后往前扫是因为这个题目中结果仍然放在A中，如果从前扫会有覆盖掉未检查的元素的可能性。算法的时间复杂度是O(m+n),m和n分别是两个数组的长度，空间复杂度是O(1)。**

**代码如下：**

public void merge(int A[], int m, int B[], int n) {

if(B == null || n == 0){

return;

}

int idx = m + n - 1;

int idxA = m - 1;

int idxB = n - 1;

while(idxA >= 0 && idxB >= 0){

if(A[idxA] > B[idxB]){

A[idx--] = A[idxA--];

} else {

A[idx--] = B[idxB--];

}

}

while(idxB >=0){

A[idx--] = B[idxB--];

}

}

**做题时的感悟:**

**对于这种类型的题目，感觉使用while循环要优于for循环，因为比较好控制。**

**2. Merge Two Sorted Lists，一般来说合并的思路就是以一个为主参考，然后逐项比较，如果较小元素在参考链表中，则继续前进，否则把结点插入参考链表中，前进另一个链表， 最后如果另一个链表还没到头就直接接过来就可以了。维护两个指针对应两个链表，因为一般会以一条链表为基准，比如说l1, 那么如果l1当前那的元素比较小，那么直接移动l1即可，否则将l2当前的元素插入到l1当前元素的前面。算法时间复杂度是O(m+n),m和n分别是两条链表的长度，空间复杂度是O(1).**

**dummy是一种链表比较常见的技巧，就是在链表头构造一个空节点，这样是有利于链表操作中需要改动链表头时不需要分情况讨论.**

**代码如下：**

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode helper = new ListNode(0);

helper.next = l1;

ListNode pre = helper;

while(l1 != null && l2 != null){

if(l1.val <= l2.val){

l1 = l1.next;

} else {

ListNode next = l2.next;

l2.next = pre.next;

pre.next = l2;

l2 = next;

}

pre = pre.next;

}

if(l2 != null){

pre.next = l2;

}

return helper.next;

}

**做题时的感悟:**

**1. 在我们需要改变链表头部的时候，我们可以创建一个虚拟结点在头部前面，这样可以避免讨论，而且最后返回dummy.next即为新链表的头部，非常方便。**

**2. pre结点开始指向dummy，当l1指向空时，pre正好指向最后一个结点。所以如果l2还没空，直接把l2接到pre后面即可。**

**3. 这种以1个链表为参照的方法思想非常好，值得借鉴。**

**3. Sort List，对于链表，用**[**Merge Two Sorted Lists**](http://blog.csdn.net/linhuanmars/article/details/19712593)**作为合并的子操作，然后用**[**归并排序**](http://zh.wikipedia.org/wiki/%E5%BD%92%E5%B9%B6%E6%8E%92%E5%BA%8F)**的递归进行分割合并就可以了。我们需要做的就是每次找到中点，然后对于左右进行递归，最后用**[**Merge Two Sorted Lists**](http://blog.csdn.net/linhuanmars/article/details/19712593)**把他们合并起来。不过用归并排序有个问题就是这里如果把栈空间算上的话还是需要O(logn)的空间的。**

**排序是面试中比较基础的一个主题，所以对于各种常见的排序算法大家还是要熟悉，不了解的朋友可以参见**[**排序算法 - Wiki**](http://zh.wikipedia.org/wiki/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95)**。特别是算法的原理，很多题目虽然没有直接考察排序的实现，但是用到了其中的思想，比如非常经典的topK问题，就用到了快速排序的原理，关于这个问题在**[**Median of Two Sorted Arrays**](http://blog.csdn.net/linhuanmars/article/details/19905515)**中有提到。**

**代码如下：**

public ListNode sortList(ListNode head) {

if(head == null || head.next == null){

return head;

}

ListNode walker = head;

ListNode runner = head;

while(runner.next != null && runner.next.next != null){

walker = walker.next;

runner = runner.next.next;

}

ListNode head1 = head;

ListNode head2 = walker.next;

walker.next = null;

head1 = sortList(head1);

head2 = sortList(head2);

return merge(head1, head2);

}

private ListNode merge(ListNode l1, ListNode l2){

ListNode dummy = new ListNode(0);

dummy.next = l1;

ListNode pre = dummy;

while(l1 != null && l2 != null){

if(l1.val <= l2.val){

l1 = l1.next;

} else {

ListNode next = l2.next;

l2.next = pre.next;

pre.next = l2;

l2 = next;

}

pre = pre.next;

}

if(l2 != null){

pre.next = l2;

}

return dummy.next;

}

**做题时的感悟:**

**1. 循环条件要灵活多变，根据题目要求来设定。该题需要取walker.next作为head2，要提前1步结束，否则不能均分链表为2段，所以循环条件为runner.next != null && runner.next.next != null。**

**2. 找到中点后需将walker.next指针置空，walker.next = null;即切断原链表为2个子链表。**

**3. 用merge作为subroutine，divide之后conquer，最后完成merge sort。**

**核心代码代码如下：**

**ListNode head1 = head;**

**ListNode head2 = walker.next;**

**walker.next = null;**

**head1 = sortList(head1);**

**head2 = sortList(head2);**

**return merge(head1, head2);**

**4. Merge k Sorted Lists，这道题在分布式系统中非常常见，来自不同client的sorted list要在central server上面merge起来。这道题一般有两种做法，下面一一介绍并分析复杂度。**

**第一种做法比较容易想到，有点类似于MergeSort的思路，就是分治法。思路是先分成两个子任务，然后递归求子任务，最后回溯回来。这道题也是这样，先把k个list分成两半，然后继续划分，直到剩下两个list就合并起来，合并时会用到**[**Merge Two Sorted Lists**](http://blog.csdn.net/linhuanmars/article/details/19712593)**这道题。**

**我们来分析一下上述算法的时间复杂度。假设总共有k个list，每个list的最大长度是n，那么运行时间满足递推式T(k) = 2T(k/2)+O(n\*k)。根据主定理，可以算出算法的总复杂度是O(nklogk)。如果不了解主定理的朋友，可以参见**[**主定理-维基百科**](http://zh.wikipedia.org/wiki/%E4%B8%BB%E5%AE%9A%E7%90%86)**。空间复杂度的话是递归栈的大小O(logk)。**

**代码如下：**

// Solution 1 - Divide and Conquer

public ListNode mergeKLists(List<ListNode> lists) {

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

return null;

}

return helper(lists, 0, lists.size() - 1);

}

private ListNode helper(List<ListNode> lists, int l, int r){

if(l == r){

return lists.get(l);

}

int m = (l + r) / 2;

ListNode head1 = helper(lists, l, m);

ListNode head2 = helper(lists, m + 1, r);

return merge(head1, head2);

}

private ListNode merge(ListNode l1, ListNode l2){

ListNode dummy = new ListNode(0);

dummy.next = l1;

ListNode pre = dummy;

while(l1 != null && l2 != null){

if(l1.val <= l2.val){

l1 = l1.next;

} else {

ListNode next = l2.next;

l2.next = pre.next;

pre.next = l2;

l2 = next;

}

pre = pre.next;

}

if(l2 != null){

pre.next = l2;

}

return dummy.next;

}

**第二种方法用到了堆的数据结构，思路比较难想到，但是其实原理比较简单。维护一个大小为k的堆，每次取堆顶的最小元素放到结果中，然后读取该元素的下一个元素放入堆中，重新维护好。因为每个链表是有序的，每次又是取当前k个元素中最小的，所以当所有链表都读完时结束，这个时候所有元素按从小到大放在结果链表中。这个算法每个元素要读取一次，即是k\*n次，然后每次读取元素要把新元素插入堆中要logk的复杂度，所以总时间复杂度是O(nklogk)。空间复杂度是堆的大小，即为O(k)。**

**可以看出两种方法有着同样的时间复杂度，都是可以接受的解法，但是却代表了两种不同的思路，数据结构也不用。个人觉得两种方法都掌握会比较好，因为在实际中比较有应用。**

**代码如下：**

// Solution 2 - Heap

public ListNode mergeKLists(List<ListNode> lists) {

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

return null;

}

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

public int compare(ListNode l1, ListNode l2){

return l1.val - l2.val;

}

});

for(int i = 0; i < lists.size(); i++){

ListNode node = lists.get(i);

if(node != null){

heap.offer(node);

}

}

ListNode helper = new ListNode(0);

ListNode pre = helper;

while(!heap.isEmpty()){

ListNode cur = heap.poll();

pre.next = cur;

pre = pre.next;

if(cur.next != null){

heap.offer(cur.next);

}

}

return helper.next;

}

**做题时的感悟:**

**1. mergeSort解法中，divide部分中，如果l == r的时候，就说明只剩下单一的一个链表，直接返回即可。否则继续递归左面l -> m，右面是m + 1 - > r。这里需要注意的是，左边取m，右边取m + 1，这样链表才可以均分成2部分。如果左面l -> m - 1，右面m -> r在链表长度是偶数的情况下就是不正确的解法。**

**2. 这道题中heap是用一个PriorityQueue来实现的，PriorityQueue中的方法与queue中的基本相同。如果我们知道heap的准确size，可以在创建heap的时候指定，而且在创建heap的时候，可以指定comparator来确定heap中的排序规则。**

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

**public int compare(ListNode l1, ListNode l2){**

**return l1.val - l2.val;**

**}**

**});**

**3. 在加入heap之前，可以先判断一下node是否为空，为空代表对应链表已经把所有结点都加入结果链表了，就不用再加入heap中去了。**