237. Delete Node in a Linked List删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为: 4 -> 5 -> 1 -> 9

|  |
| --- |
| 输入: head = [4,5,1,9], node = 5  输出: [4,1,9]  解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9. |
| 输入: head = [4,5,1,9], node = 1  输出: [4,5,9]  解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9. |

说明：链表至少包含两个节点。链表中所有节点的值都是唯一的。给定的节点为非末尾节点并且一定是链表中的一个有效节点。不要从你的函数中返回任何结果。

【分析】删除链表的一个节点，没有给我们链表的起点，只给我们了一个要删的节点，我们之前要删除一个节点的方法是要有其前一个节点的位置，然后将其前一个节点的next连向要删节点的下一个，然后delete掉要删的节点即可。这道题的处理方法是先把当前节点的值用下一个节点的值覆盖了，然后我们删除下一个节点即可。

【CODE】class Solution {

public:

void deleteNode(ListNode\* node) {

node->val = node->next->val;

ListNode \*tmp = node->next;

node->next = tmp->next;

delete tmp;

}

};

206. Reverse Linked List反转链表

反转一个单链表。

输入: 1->2->3->4->5->NULL 输出: 5->4->3->2->1->NULL

【分析】迭代的解法，思路是在原链表之前建立一个dummy node，因为首节点会变，然后从head开始，将之后的一个节点移到dummy node之后，重复此操作知道head成为末节点为止。

【CODE】// Iterative

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if (!head) return head;

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*cur = head;

while (cur->next) {

ListNode \*tmp = cur->next;

cur->next = tmp->next;

tmp->next = dummy->next;

dummy->next = tmp;

}

return dummy->next;

}

};

class Solution { //简洁写法

public:

ListNode\* reverseList(ListNode\* head) {

ListNode\* pre = NULL;

ListNode\* cur = head;

while (cur != NULL) {

ListNode\* next = cur->next; // forward

cur->next = pre; // reverse

pre = cur; // forward

cur = next; // forward

}

return pre; // cur == NULL;

}

};

【分析】递归解法，代码量更少。思路是不断的进入递归函数，直到head指向最后一个节点，p指向之前一个节点，然后调换head和p的位置，再返回上一层递归函数，再交换p和head的位置，每次交换后，head节点后面都是交换好的顺序，直到p为首节点，然后再交换，首节点就成了为节点，此时整个链表也完成了翻转。

【CODE】// Recursive

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*p = head;

head = reverseList(p->next);

p->next->next = p;

p->next = NULL;

return head;

}

};

21. Merge Two Sorted Lists合并两个有序链表

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

输入：1->2->4, 1->3->4 输出：1->1->2->3->4->4

【分析】具体思想是新建一个链表，然后比较两个链表中的元素值，把较小的那个链到新链表中，由于两个输入链表的长度可能不同，所以最终会有一个链表先完成插入所有元素，则直接另一个未完成的链表直接链入新链表的末尾。

【CODE】

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

ListNode \*dummy = new ListNode(-1), \*cur = dummy;

while (l1 && l2) {

if (l1->val < l2->val) {

cur->next = l1;

l1 = l1->next;

} else {

cur->next = l2;

l2 = l2->next;

}

cur = cur->next;

}

cur->next = l1 ? l1 : l2;

return dummy->next;

}

};

【分析】递归写法，当某个链表为空了，就返回另一个。然后核心还是比较当前两个节点值大小，如果l1的小，那么对于l1的下一个节点和l2调用递归函数，将返回值赋值给l1.next，然后返回l1；否则就对于l2的下一个节点和l1调用递归函数，将返回值赋值给l2.next，然后返回l2。

【CODE】

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

if (!l1 || (l2 && l1->val > l2->val)) swap(l1, l2);

if (l1) l1->next = mergeTwoLists(l1->next, l2);

return l1;

}

};

23. Merge k Sorted Lists合并K个排序链表

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。

【分析】不管合并几个，基本还是要两两合并。那么我们首先考虑的方法是能不能利用上题的解法来解答此题。答案是肯定的，但是需要修改，怎么修改，最先想到的就是两两合并，就是前两个先合并，合并好了再跟第三个，然后第四个直到第k个。这样的思路是对的，但是效率不高，没法通过OJ，所以我们只能换一种思路，这里就需要用到分治法 Divide and Conquer Approach。简单来说就是不停的对半划分，比如k个链表先划分为合并两个k/2个链表的任务，再不停的往下划分，直到划分成只有一个或两个链表的任务，开始合并。举个例子来说比如合并6个链表，那么按照分治法，我们首先分别合并1和4,2和5,3和6。这样下一次只需合并3个链表，我们再合并1和3，最后和2合并就可以了。

【CODE】class Solution {

public:

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

if (lists.size() == 0) return NULL;

int n = lists.size();

while (n > 1) {

int k = (n + 1) / 2;

for (int i = 0; i < n / 2; ++i) {

lists[i] = mergeTwoLists(lists[i], lists[i + k]);

}

n = k;

}

return lists[0];

}

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode \*head = new ListNode(-1);

ListNode \*cur = head;

while (l1 && l2) {

if (l1->val < l2->val) {

cur->next = l1;

l1 = l1->next;

} else {

cur->next = l2;

l2 = l2->next;

}

cur = cur->next;

}

if (l1) cur->next = l1;

if (l2) cur->next = l2;

return head->next;

}

};

【分析】另一种解法，这种解法利用了最小堆这种数据结构，我们首先把k个链表的首元素都加入最小堆中，它们会自动排好序。然后我们每次取出最小的那个元素加入我们最终结果的链表中，然后把取出元素的下一个元素再加入堆中，下次仍从堆中取出最小的元素做相同的操作，以此类推，直到堆中没有元素了，此时k个链表也合并为了一个链表，返回首节点即可。

【CODE】struct cmp {

bool operator () (ListNode \*a, ListNode \*b) {

return a->val > b->val;

}

};

class Solution {

public:

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

priority\_queue<ListNode\*, vector<ListNode\*>, cmp> q;

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

if (lists[i]) q.push(lists[i]);

}

ListNode \*head = NULL, \*pre = NULL, \*tmp = NULL;

while (!q.empty()) {

tmp = q.top();

q.pop();

if (!pre) head = tmp;

else pre->next = tmp;

pre = tmp;

if (tmp->next) q.push(tmp->next);

}

return head;

}

};

19. Remove Nth Node From End of List删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

说明：给定的 n 保证是有效的。

进阶：你能尝试使用一趟扫描实现吗？

【分析】移除链表倒数第N个节点，限定n一定是有效的，即n不会大于链表中的元素总数。还要求一次遍历解决问题，那么就得想些比较巧妙的方法了。比如我们首先要考虑的时，如何找到倒数第N个节点，由于只允许一次遍历，所以我们不能用一次完整的遍历来统计链表中元素的个数，而是遍历到对应位置就应该移除了。那么我们需要用两个指针来帮助我们解题，pre和cur指针。首先cur指针先向前走N步，如果此时cur指向空，说明N为链表的长度，则需要移除的为首元素，那么此时我们返回head->next即可，如果cur存在，我们再继续往下走，此时pre指针也跟着走，直到cur为最后一个元素时停止，此时pre指向要移除元素的前一个元素，我们再修改指针跳过需要移除的元素即可。

【CODE】

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

if (!head->next) return NULL;

ListNode \*pre = head, \*cur = head;

for (int i = 0; i < n; ++i) cur = cur->next;

if (!cur) return head->next;

while (cur->next) {

cur = cur->next;

pre = pre->next;

}

pre->next = pre->next->next;

return head;

}

};

24. Swap Nodes in Pairs两两交换链表中的节点

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

示例:给定 1->2->3->4, 你应该返回 2->1->4->3.

说明:你的算法只能使用常数的额外空间。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

【分析】基本的链表操作题，我们可以分别用递归和迭代来实现。对于迭代实现，还是需要建立dummy节点，注意在连接节点的时候，最好画个图，以免把自己搞晕。

【CODE】

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

ListNode \*dummy = new ListNode(-1), \*pre = dummy;

dummy->next = head;

while (pre->next && pre->next->next) {

ListNode \*t = pre->next->next;

pre->next->next = t->next;

t->next = pre->next;

pre->next = t;

pre = t->next;

}

return dummy->next;

}

};

【分析】递归的写法更简洁，实际上利用了回溯的思想，递归遍历到链表末尾，然后先交换末尾两个，然后依次往前交换。

【CODE】

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*t = head->next;

head->next = swapPairs(head->next->next);

t->next = head;

return t;

}

};

25. Reverse Nodes in k-Group k个一组翻转链表

给出一个链表，每 k 个节点一组进行翻转，并返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么将最后剩余节点保持原有顺序。

给定这个链表：1->2->3->4->5

当 k = 2 时，应当返回: 2->1->4->3->5

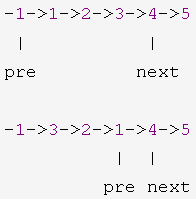
当 k = 3 时，应当返回: 3->2->1->4->5

说明 :你的算法只能使用常数的额外空间。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

【分析】以每k个为一组来翻转链表，实际上是把原链表分成若干小段，然后分别对其进行翻转，那么肯定总共需要两个函数，一个是用来分段的，一个是用来翻转的，我们就以题目中给的例子来看，对于给定链表1->2->3->4->5，一般在处理链表问题时，我们大多时候都会在开头再加一个dummy node，因为翻转链表时头结点可能会变化，为了记录当前最新的头结点的位置而引入的dummy node，那么我们加入dummy node后的链表变为-1->1->2->3->4->5，如果k为3的话，我们的目标是将1,2,3翻转一下，那么我们需要一些指针，pre和next分别指向要翻转的链表的前后的位置，然后翻转后pre的位置更新到如下新的位置。以此类推，只要next走过k个节点，就可以调用翻转函数来进行局部翻转了。

【CODE】

class Solution {

public:

ListNode \*reverseKGroup(ListNode \*head, int k) {

if (!head || k == 1) return head;

ListNode \*dummy = new ListNode(-1);

ListNode \*pre = dummy, \*cur = head;

dummy->next = head;

int i = 0;

while (cur) {

++i;

if (i % k == 0) {

pre = reverseOneGroup(pre, cur->next);

cur = pre->next;

} else {

cur = cur->next;

}

}

return dummy->next;

}

ListNode \*reverseOneGroup(ListNode \*pre, ListNode \*next) {

ListNode \*last = pre->next;

ListNode \*cur = last->next;

while(cur != next) {

last->next = cur->next;

cur->next = pre->next;

pre->next = cur;

cur = last->next;

}

return last;

}

};

【分析】也可以在一个函数中完成，我们首先遍历整个链表，统计出链表的长度，然后如果长度大于等于k，我们开始交换节点，当k=2时，每段我们只需要交换一次，当k=3时，每段需要交换2此，所以i从1开始循环，注意交换一段后更新pre指针，然后num自减k，直到num<k时循环结束。

【CODE】class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode \*dummy = new ListNode(-1), \*pre = dummy, \*cur = pre;

dummy->next = head;

int num = 0;

while (cur = cur->next) ++num;

while (num >= k) {

cur = pre->next;

for (int i = 1; i < k; ++i) {

ListNode \*t = cur->next;

cur->next = t->next;

t->next = pre->next;

pre->next = t;

}

pre = cur;

num -= k;

}

return dummy->next;

}

};

【分析】使用递归来做，我们用head记录每段的开始位置，cur记录结束位置的下一个节点，然后我们调用reverse函数来将这段翻转，然后得到一个new\_head，原来的head就变成了末尾，这时候后面接上递归调用下一段得到的新节点，返回new\_head即可。

【CODE】class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode \*cur = head;

for (int i = 0; i < k; ++i) {

if (!cur) return head;

cur = cur->next;

}

ListNode \*new\_head = reverse(head, cur);

head->next = reverseKGroup(cur, k);

return new\_head;

}

ListNode\* reverse(ListNode\* head, ListNode\* tail) {

ListNode \*pre = tail;

while (head != tail) {

ListNode \*t = head->next;

head->next = pre;

pre = head;

head = t;

}

return pre;

}

};

61. Rotate List旋转链表

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。

|  |  |
| --- | --- |
| 输入: 1->2->3->4->5->NULL, k = 2  输出: 4->5->1->2->3->NULL  解释:  向右旋转 1 步: 5->1->2->3->4->NULL  向右旋转 2 步: 4->5->1->2->3->NULL | 输入: 0->1->2->NULL, k = 4  输出: 2->0->1->NULL  解释:  向右旋转 1 步: 2->0->1->NULL  向右旋转 2 步: 1->2->0->NULL  向右旋转 3 步: 0->1->2->NULL  向右旋转 4 步: 2->0->1->NULL |

【分析】先遍历整个链表获得链表长度n，然后此时把链表头和尾链接起来，在往后走n - k % n个节点就到达新链表的头结点前一个点，这时断开链表即可。（还有一种利用快慢指针的做法，需要考虑空链表和k>n的情况。）

【CODE】class Solution {

public:

ListNode \*rotateRight(ListNode \*head, int k) {

if (!head) return NULL;

int n = 1;

ListNode \*cur = head;

while (cur->next) {

++n;

cur = cur->next;

}

cur->next = head;

int m = n - k % n;

for (int i = 0; i < m; ++i) {

cur = cur->next;

}

ListNode \*newhead = cur->next;

cur->next = NULL;

return newhead;

}

};

82. Remove Duplicates from Sorted List II删除排序链表中的重复元素 II

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 *没有重复出现* 的数字。

|  |  |
| --- | --- |
| 输入: 1->2->3->3->4->4->5  输出: 1->2->5 | 输入: 1->1->1->2->3  输出: 2->3 |

【分析】要删掉所有的重复项，由于链表开头可能会有重复项，被删掉的话头指针会改变，而最终却还需要返回链表的头指针。所以需要定义一个新的节点，然后链上原链表，然后定义一个前驱指针和一个现指针，每当前驱指针指向新建的节点，现指针从下一个位置开始往下遍历，遇到相同的则继续往下，直到遇到不同项时，把前驱指针的next指向下面那个不同的元素。如果现指针遍历的第一个元素就不相同，则把前驱指针向下移一位。

【CODE】

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

if (!head || !head->next) return head;

ListNode \*start = new ListNode(0);

start->next = head;

ListNode \*pre = start;

while (pre->next) {

ListNode \*cur = pre->next;

while (cur->next && cur->next->val == cur->val) cur = cur->next;

if (cur != pre->next) pre->next = cur->next;

else pre = pre->next;

}

return start->next;

}

};

86. Partition List分隔链表

给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。你应当保留两个分区中每个节点的初始相对位置。

Given 1->4->3->2->5->2 and x = 3,

return 1->2->2->4->3->5.

【分析】首先找到第一个大于或等于给定值的节点，用题目中给的例子来说就是先找到4，然后再找小于3的值，每找到一个就将其取出置于4之前即可。

还有一种解法，就是将所有小于给定值的节点取出组成一个新的链表，此时原链表中剩余的节点的值都大于或等于给定值，只要将原链表直接接在新链表后即可。

【CODE】方法1

class Solution {

public:

ListNode \*partition(ListNode \*head, int x) {

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*pre = dummy, \*cur = head;;

while (pre->next && pre->next->val < x) pre = pre->next;

cur = pre;

while (cur->next) {

if (cur->next->val < x) {

ListNode \*tmp = cur->next;

cur->next = tmp->next;

tmp->next = pre->next;

pre->next = tmp;

pre = pre->next;

} else {

cur = cur->next;

}

}

return dummy->next;

}

};

92. Reverse Linked List II反转链表 II

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。

说明:1 ≤ m ≤ n ≤ 链表长度。

输入: 1->2->3->4->5->NULL, m = 2, n = 4

输出: 1->4->3->2->5->NULL

【分析】对于链表的问题，根据以往的经验一般都是要建一个dummy node，连上原链表的头结点，这样的话就算头结点变动了，我们还可以通过dummy->next来获得新链表的头结点。这道题的要求是只通过一次遍历完成，就拿题目中的例子来说，变换的是2,3,4这三个点，那么我们可以先取出2，用front指针指向2，然后当取出3的时候，我们把3加到2的前面，把front指针前移到3，依次类推，到4后停止，这样我们得到一个新链表4->3->2, front指针指向4。对于原链表连说，有两个点的位置很重要，需要用指针记录下来，分别是1和5，因为当2,3,4被取走时，原链表就变成了1->5->NULL，要把新链表插入的时候需要这两个点的位置。1的位置很好找，因为知道m的值，我们用pre指针记录1的位置，5的位置最后才能记录，当4结点被取走后，5的位置需要记下来，这样我们就可以把倒置后的那一小段链表加入到原链表中。

【CODE】class Solution {

public:

ListNode \*reverseBetween(ListNode \*head, int m, int n) {

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*cur = dummy;

ListNode \*pre, \*front, \*last;

for (int i = 1; i <= m - 1; ++i) cur = cur->next;

pre = cur;

last = cur->next;

for (int i = m; i <= n; ++i) {

cur = pre->next;

pre->next = cur->next;

cur->next = front;

front = cur;

}

cur = pre->next;

pre->next = front;

last->next = cur;

return dummy->next;

}

};

109. Convert Sorted List to Binary Search Tree有序链表转换二叉搜索树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

【分析】数组方便就方便在可以通过index直接访问任意一个元素，而链表不行。由于二分查找法每次需要找到中点，而链表的查找中间点可以通过快慢指针来操作，可参见之前的Reorder List 链表重排序和Linked List Cycle II 单链表中的环二有关快慢指针的应用。找到中点后，要以中点的值建立一个数的根节点，然后需要把原链表断开，分为前后两个链表，都不能包含原中节点，然后再分别对这两个链表递归调用原函数，分别连上左右子节点即可。

【CODE】

class Solution {

public:

TreeNode \*sortedListToBST(ListNode \*head) {

if (!head) return NULL;

if (!head->next) return new TreeNode(head->val);

ListNode \*slow = head;

ListNode \*fast = head;

ListNode \*last = slow;

while (fast->next && fast->next->next) {

last = slow;

slow = slow->next;

fast = fast->next->next;

}

fast = slow->next;

last->next = NULL;

TreeNode \*cur = new TreeNode(slow->val);

if (head != slow) cur->left = sortedListToBST(head);

cur->right = sortedListToBST(fast);

return cur;

}

};

138. Copy List with Random Pointer复制带随机指针的链表

给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。

要求返回这个链表的深度拷贝。

【分析】链表的深度拷贝题的难点就在于如何处理随机指针的问题，由于每一个节点都有一个随机指针，这个指针可以为空，也可以指向链表的任意一个节点，如果我们在每生成一个新节点给其随机指针赋值时，都要去遍历原链表的话，OJ上肯定会超时，所以我们可以考虑用Hash map来缩短查找时间，第一遍遍历生成所有新节点时同时建立一个原节点和新节点的哈希表，第二遍给随机指针赋值时，查找时间是常数级。

【CODE】

class Solution {

public:

RandomListNode \*copyRandomList(RandomListNode \*head) {

if (!head) return NULL;

RandomListNode \*res = new RandomListNode(head->label);

RandomListNode \*node = res;

RandomListNode \*cur = head->next;

map<RandomListNode\*, RandomListNode\*> m;

m[head] = res;

while (cur) {

RandomListNode \*tmp = new RandomListNode(cur->label);

node->next = tmp;

m[cur] = tmp;

node = node->next;

cur = cur->next;

}

node = res;

cur = head;

while (node) {

node->random = m[cur->random];

node = node->next;

cur = cur->next;

}

return res;

}

};

【分析】如果使用哈希表占用额外的空间，如果这道题限制了空间的话，就要考虑别的方法。下面这个方法很巧妙分以下三个步骤：

1. 在原链表的每个节点后面拷贝出一个新的节点

2. 依次给新的节点的随机指针赋值，

而且这个赋值非常容易 cur->next->random = cur->random->next

3. 断开链表可得到深度拷贝后的新链表

【CODE】

class Solution {

public:

RandomListNode \*copyRandomList(RandomListNode \*head) {

if (!head) return NULL;

RandomListNode \*cur = head;

while (cur) {

RandomListNode \*node = new RandomListNode(cur->label);

node->next = cur->next;

cur->next = node;

cur = node->next;

}

cur = head;

while (cur) {

if (cur->random) {

cur->next->random = cur->random->next;

}

cur = cur->next->next;

}

cur = head;

RandomListNode \*res = head->next;

while (cur) {

RandomListNode \*tmp = cur->next;

cur->next = tmp->next;

if(tmp->next) tmp->next = tmp->next->next;

cur = cur->next;

}

return res;

}

};

142. Linked List Cycle II环形链表 II

给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

说明：不允许修改给定的链表。

进阶：你是否可以不用额外空间解决此题？

【分析】设快慢指针，不过这次要记录两个指针相遇的位置，当两个指针相遇了后，让其一指针从链表头开始，一步两步，此时再相遇的位置就是链表中环的起始位置。

【CODE】class Solution {

public:

ListNode \*detectCycle(ListNode \*head) {

ListNode \*slow = head, \*fast = head;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) break;

}

if (!fast || !fast->next) return NULL;

slow = head;

while (slow != fast) {

slow = slow->next;

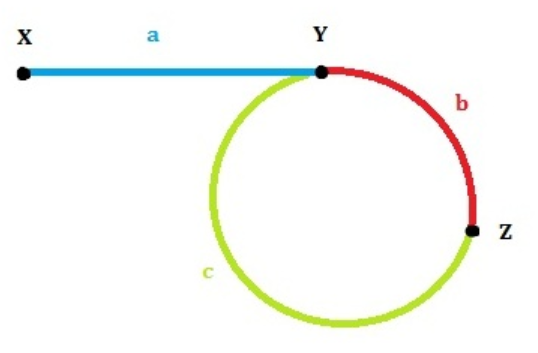
fast = fast->next;

}

return fast;

}

};

**补.环链表的相关问题**

1. 环的长度是多少？

2. 如何找到环中第一个节点（即Linked List Cycle II）？

3. 如何将有环的链表变成单链表（解除环）？

4. 如何判断两个单链表是否有交点？如何找到第一个相交的节点？

设：链表头是X，环的第一个节点是Y，slow和fast第一次的交点是Z。各段的长度分别是a,b,c，如图所示。环的长度是L。slow和fast的速度分别是qs,qf。

1. 方法一（网上都是这个答案）：

第一次相遇后，让slow,fast继续走，记录到下次相遇时循环了几次。因为当fast第二次到达Z点时，fast走了一圈，slow走了半圈，而当fast第三次到达Z点时，fast走了两圈，slow走了一圈，正好还在Z点相遇。

方法二：

第一次相遇后，让fast停着不走了，slow继续走，记录到下次相遇时循环了几次。

方法三（最简单）：

第一次相遇时slow走过的距离：a+b，fast走过的距离：a+b+c+b。

因为fast的速度是slow的两倍，所以fast走的距离是slow的两倍，有 2(a+b) = a+b+c+b，可以得到a=c（这个结论很重要！）。

L=b+c=a+b，也就是说，从一开始到二者第一次相遇，循环的次数就等于环的长度。

2. 我们已经得到了结论a=c，那么让两个指针分别从X和Z开始走，每次走一步，那么正好会在Y相遇！也就是环的第一个节点。

3. 在上一个问题的最后，将c段中Y点之前的那个节点与Y的链接切断即可。

4. 如何判断两个单链表是否有交点？先判断两个链表是否有环，如果一个有环一个没环，肯定不相交；如果两个都没有环，判断两个列表的尾部是否相等；如果两个都有环，判断一个链表上的Z点是否在另一个链表上。

如何找到第一个相交的节点？求出两个链表的长度L1,L2（如果有环，则将Y点当做尾节点来算），假设L1<L2，用两个指针分别从两个链表的头部开始走，长度为L2的链表先走(L2-L1)步，然后两个一起走，直到二者相遇。

143. Reorder List重排链表

给定一个单链表 L：L0→L1→…→Ln-1→Ln ，

将其重新排列后变为： L0→Ln→L1→Ln-1→L2→Ln-2→…

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

【分析】这道链表重排序问题可以拆分为以下三个小问题：

1. 使用快慢指针来找到链表的中点，并将链表从中点处断开，形成两个独立的链表。

2. 将第二个链翻转。

3. 将第二个链表的元素间隔地插入第一个链表中。

【CODE】

class Solution {

public:

void reorderList(ListNode \*head) {

if (!head || !head->next || !head->next->next) return;

ListNode \*fast = head;

ListNode \*slow = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

ListNode \*mid = slow->next;

slow->next = NULL;

ListNode \*last = mid;

ListNode \*pre = NULL;

while (last) {

ListNode \*next = last->next;

last->next = pre;

pre = last;

last = next;

}

while (head && pre) {

ListNode \*next = head->next;

head->next = pre;

pre = pre->next;

head->next->next = next;

head = next;

}

}

};

147. Insertion Sort List对链表进行插入排序

插入排序算法：

1.插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。

2.每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。

3.重复直到所有输入数据插入完为止。

【分析】链表的插入排序实现原理很简单，就是一个元素一个元素的从原链表中取出来，然后按顺序插入到新链表中，时间复杂度为O(n2)，是一种效率并不是很高的算法，但是空间复杂度为O(1)，以高时间复杂度换取了低空间复杂度

【CODE】

class Solution {

public:

ListNode\* insertionSortList(ListNode\* head) {

ListNode \*dummy = new ListNode(-1), \*cur = dummy;

while (head) {

ListNode \*t = head->next;

cur = dummy;

while (cur->next && cur->next->val <= head->val) {

cur = cur->next;

}

head->next = cur->next;

cur->next = head;

head = t;

}

return dummy->next;

}

};

148. Sort List排序链表

在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

【分析】常见排序方法有很多，插入排序，选择排序，堆排序，快速排序，冒泡排序，归并排序，桶排序等等。。它们的时间复杂度不尽相同，而这里题目限定了时间必须为O(nlgn)，符合要求只有快速排序，归并排序，堆排序，而根据单链表的特点，最适于用归并排序。

【CODE】

class Solution {

public:

ListNode\* sortList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*slow = head, \*fast = head, \*pre = head;

while (fast && fast->next) {

pre = slow;

slow = slow->next;

fast = fast->next->next;

}

pre->next = NULL;

return merge(sortList(head), sortList(slow));

}

ListNode\* merge(ListNode\* l1, ListNode\* l2) {

if (!l1) return l2;

if (!l2) return l1;

if (l1->val < l2->val) {

l1->next = merge(l1->next, l2);

return l1;

} else {

l2->next = merge(l1, l2->next);

return l2;

}

}

};

369. Plus One Linked List 链表加一运算

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example: Input:1->2->3 Output:1->2->4

【分析】给了我们一个链表，用来模拟一个三位数，表头是高位，现在让我们进行加1运算，这道题的难点在于链表无法通过坐标来访问元素，只能通过遍历的方式进行，而这题刚好让我们从链尾开始操作，从后往前，遇到进位也要正确的处理，最后还有可能要在开头补上一位。那么我们反过来想，如果链尾是高位，那么进行加1运算就方便多了，直接就可以边遍历边进行运算处理，那么我们可以做的就是先把链表翻转一下，然后现在就是链尾是高位了，我们进行加1处理运算结束后，再把链表翻转回来即可。

【CODE】

class Solution {

public:

ListNode\* plusOne(ListNode\* head) {

if (!head) return head;

ListNode \*rev\_head = reverse(head), \*cur = rev\_head, \*pre = cur;

int carry = 1;

while (cur) {

pre = cur;

int t = cur->val + carry;

cur->val = t % 10;

carry = t / 10;

if (carry == 0) break;

cur = cur->next;

}

if (carry) pre->next = new ListNode(1);

return reverse(rev\_head);

}

ListNode\* reverse(ListNode \*head) {

if (!head) return head;

ListNode \*dummy = new ListNode(-1), \*cur = head;

dummy->next = head;

while (cur->next) {

ListNode \*t = cur->next;

cur->next = t->next;

t->next = dummy->next;

dummy->next = t;

}

return dummy->next;

}

};

【分析】可以通过递归来实现，这样我们就不用翻转链表了，通过递归一层一层的调用，最先处理的是链尾元素，我们将其加1，然后看是否有进位，返回进位，然后回溯到表头，加完进位，如果发现又产生了新的进位，那么我们在最开头加上一个新节点即可。

【CODE】class Solution {

public:

ListNode\* plusOne(ListNode\* head) {

if (!head) return head;

int carry = helper(head);

if (carry == 1) {

ListNode \*res = new ListNode(1);

res->next = head;

return res;

}

return head;

}

int helper(ListNode \*node) {

if (!node) return 1;

int carry = helper(node->next);

int sum = node->val + carry;

node->val = sum % 10;

return sum / 10;

}

};

445. Add Two Numbers II两数相加 II

给定两个非空链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储单个数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶:如果输入链表不能修改该如何处理？即你不能对列表中的节点进行翻转。

输入: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4) 输出: 7 -> 8 -> 0 -> 7

【分析】我们可以看到这道题的最高位在链表首位置，如果我们给链表翻转一下的话就跟之前的题目一样了，这里我们来看不修改链表顺序的方法。由于加法需要从最低位开始运算，而最低位在链表末尾，链表只能从前往后遍历，没法取到前面的元素，那怎么办呢？我们可以利用栈来保存所有的元素，然后利用栈的后进先出的特点就可以从后往前取数字了，我们首先遍历两个链表，将所有数字分别压入两个栈s1和s2中，我们建立一个值为0的res节点，然后开始循环，如果栈不为空，则将栈顶数字加入sum中，然后将res节点值赋为sum%10，然后新建一个进位节点head，赋值为sum/10，如果没有进位，那么就是0，然后我们head后面连上res，将res指向head，这样循环退出后，我们只要看res的值是否为0，为0返回res->next，不为0则返回res即可。

【CODE】

class Solution {

public:

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

stack<int> s1, s2;

while (l1) {

s1.push(l1->val);

l1 = l1->next;

}

while (l2) {

s2.push(l2->val);

l2 = l2->next;

}

int sum = 0;

ListNode \*res = new ListNode(0);

while (!s1.empty() || !s2.empty()) {

if (!s1.empty()) {sum += s1.top(); s1.pop();}

if (!s2.empty()) {sum += s2.top(); s2.pop();}

res->val = sum % 10;

ListNode \*head = new ListNode(sum / 10);

head->next = res;

res = head;

sum /= 10;

}

return res->val == 0 ? res->next : res;

}

};

328. Odd Even Linked List奇偶链表

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

|  |  |
| --- | --- |
| 输入: 1->2->3->4->5->NULL  输出: 1->3->5->2->4->NULL | 输入: 2->1->3->5->6->4->7->NULL  输出: 2->3->6->7->1->5->4->NULL |

【分析】pre指向奇节点，cur指向偶节点，然后把偶节点cur后面的那个奇节点提前到pre的后面，然后pre和cur各自前进一步，此时cur又指向偶节点，pre指向当前奇节点的末尾，以此类推直至把所有的偶节点都提前了即可。

【CODE】比较好

class Solution {

public:

ListNode\* oddEvenList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*pre = head, \*cur = head->next;

while (cur && cur->next) {

ListNode \*tmp = pre->next;

pre->next = cur->next;

cur->next = cur->next->next;

pre->next->next = tmp;

cur = cur->next;

pre = pre->next;

}

return head;

}

};

【分析】用两个奇偶指针分别指向奇偶节点的起始位置，另外需要一个单独的指针even\_head来保存偶节点的起点位置，然后把奇节点的指向偶节点的下一个(一定是奇节点)，此奇节点后移一步，再把偶节点指向下一个奇节点的下一个(一定是偶节点)，此偶节点后移一步，以此类推直至末尾，此时把分开的偶节点的链表连在奇节点的链表后即可。

【CODE】

class Solution {

public:

ListNode\* oddEvenList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode \*odd = head, \*even = head->next, \*even\_head = even;

while (even && even->next) {

odd = odd->next = even->next;

even = even->next = odd->next;

}

odd->next = even\_head;

return head;

}

};