链表

jask

09/30/2024

2 两数之和

给你两个非空的链表，表示两个非负的整数。它们每位数字都是按照逆序的方式存储的，并且每个节点只能存储一位数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 **0** 之外，这两个数都不会以 **0** 开头。

```
class Solution {
 public:
  ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode dummy;
    auto cur=&dummy;
    int carry=0;
    while(l1||l2||carry){
        int sum=carry+(l1?l1->val:0)+(l2?l2->val:0);
        cur=cur->next=new ListNode(sum%10);
        carry=sum/10;
        if(l1) l1=l1->next;
        if(l2) l2=l2->next;
    }
    return dummy.next;
   }
};
```

递归法

```
class Solution {

public:

    // l1 和 l2 为当前遍历的节点，carry 为进位

    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2, int carry = 0) {

        if (l1 == nullptr && l2 == nullptr) { // 递归边界：l1 和 l2 都是空节点

            return carry ? new ListNode(carry) : nullptr; // 如果进位了，就额外创建一个节点

        }

        if (l1 == nullptr) { // 如果 l1 是空的，那么此时 l2 一定不是空节点

            swap(l1, l2); // 交换 l1 与 l2，保证 l1 非空，从而简化代码

        }

        int sum = carry + l1->val + (l2 ? l2->val : 0); // 节点值和进位加在一起

        l1->val = sum % 10; // 每个节点保存一个数位

        l1->next = addTwoNumbers(l1->next, (l2 ? l2->next : nullptr), sum / 10); // 进位
```

```
        return l1;

    }

};
```

24 两两交换链表中的节点

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if(head==nullptr||head->next==nullptr){
            return head;
        }
        auto newHead=head->next;
        head->next=swapPairs(newHead->next);
        newHead->next=head;
        return newHead;
    }
};
```

25 K 个一组反转链表

给你链表的头节点 `head` ，每 `k` 个节点一组进行翻转，请你返回修改后的链表。

`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，那么请将最后剩余的节点保持原有顺序。

你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        int n=0;
        for(auto cur=head;cur!=nullptr;cur=cur->next){
            n++;
        }
        ListNode dummy{0,head};
        auto p0=&dummy;
        ListNode* pre=nullptr;
        auto cur=head;
        for(;n>=k;n-=k){
            for(int i=0;i<k;i++){
                auto nxt=cur->next;
                cur->next=pre;
                pre=cur;
                cur=nxt;
            }
            auto nxt=p0->next;
            p0->next->next=cur;
            p0->next=pre;
            p0=nxt;
        }
        return dummy.next;
    }
};
```

92 反转链表

给你单链表的头指针 `head` 和两个整数 `left` 和 `right` ，其中 `left <= right` 。请你反转从位置 `left` 到位置 `right` 的链表节点，返回反转后的链表。

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        auto dummy=std::make_shared<ListNode>(0,head);
        auto p0=dummy.get();
        for(int i=0;i<left-1;i++){
            p0=p0->next;
        }
        ListNode* pre=nullptr;
        ListNode* cur=p0->next;
        for(int i=0;i<right-left+1;i++){
            auto next=cur->next;
            cur->next=pre;
            pre=cur;
            cur=next;
        }
        p0->next->next=cur;
        p0->next=pre;
        return dummy->next;
    }
};
```

61 旋转链表

给你一个链表的头节点 `head` ，旋转链表，将链表每个节点向右移动 `k` 个位置。

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
                if(!head||!head->next) return head;
        auto getTail=head;
        int n=0;
        while(true){
            n++;
            if(getTail->next==nullptr){
                break;
            }
            getTail=getTail->next;
        }
        int add=n-k%n;
        if(add==n) return head;
        getTail->next=head;//拼成环，在合适位置断开
        auto cur=head;
        while(add--){
            getTail=getTail->next;
        }
        auto ret=getTail->next;
        getTail->next=nullptr;
        return ret;
    }
};
```

82 删除链表中的重复元素 2

给定一个已排序的链表的头 `head` ，删除原始链表中所有重复数字的节点，只留下不同的数字。返回已排序的链表。

注意是删除重复元素不是去重。

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode dummy{0,head};
        auto cur=&dummy;
        while(cur->next&&cur->next->next){
            int val=cur->next->val;
            if(cur->next->next->val==val){
                while(cur->next&& cur->next->val==val){
                    cur->next=cur->next->next;
                }
            }else{
                cur=cur->next;
            }
        }
        return dummy.next;
    }
};
```

86 分割链表

给你一个链表的头节点 `head` 和一个特定值 `x` ，请你对链表进行分隔，使得所有小于 `x` 的节点都出现在大于或等于 `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。

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode *small=new ListNode();
        ListNode *large=new ListNode();
        auto small_cur=small;
        auto large_cur=large;
        for(auto cur=head;cur!=nullptr;cur=cur->next){
            if(cur->val<x){
                small_cur->next=cur;
                small_cur=small_cur->next;
            }
            else{
             large_cur->next=cur;
             large_cur=large_cur->next;
            }
        }
        small_cur->next=large->next;
        large_cur->next=nullptr;
        return small->next;
    }
};
```

109 有序链表转换二叉搜索树

给定一个单链表的头节点 `head` ，其中的元素按升序排序，将其转换为平衡二叉搜索树。

```
class Solution {
public:
    ListNode* find_mid(ListNode* left,ListNode* right){
        auto fast=left;
        auto slow=left;
        while(fast!=right&&fast->next!=right){
            fast=fast->next->next;
            slow=slow->next;
        }
```

```
        return slow;
    }
    TreeNode* build_tree(ListNode* left,ListNode* right){
        if(left==right){
            return nullptr;
        }
        auto mid=find_mid(left,right);
        auto root=new TreeNode(mid->val);
        root->left=build_tree(left,mid);
        root->right=build_tree(mid->next,right);
        return root;
    }
    TreeNode* sortedListToBST(ListNode* head) {
        return build_tree(head,nullptr);
    }
};
```

141 环形链表

给你一个链表的头节点 `head` ，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 `0` 开始）。注意：`pos` 不作为参数进行传递。仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true` 。否则，返回 `false` 。

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head==nullptr||head->next==nullptr) return false;
        auto fast=head->next;
        auto slow=head;
        while(slow!=fast){
            if(fast==nullptr||fast->next==nullptr) return false;
            fast=fast->next->next;
            slow=slow->next;
        }
        return 1;
    }
};
```

142 环形链表 2

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

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 `0` 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

不允许修改链表。

```
class Solution {
public:
    ListNode* detectCycle(ListNode* head) {
        if (head == nullptr || head->next == nullptr)
            return nullptr;
        auto fast = head;
        auto slow = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                while (slow != head) {
```

```
                    slow = slow->next;
                    head = head->next;
                }
                return slow;
            }
        }

        return nullptr;
    }
};
```

143 重排链表

给定一个单链表 `L` 的头节点 `head` ，单链表 `L` 表示为：

`L0 → L1 → …→ Ln - 1 → Ln`

请将其重新排列后变为：

`L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …`

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。

```
class Solution {
public:
    void reorderList(ListNode* head) {
        std::vector<ListNode*> list;
        while(head){
            list.push_back(head);
            head=head->next;
        }
        int i=0,j=list.size()-1;
        while(i<j){
            list[i]->next=list[j];
            i++;
            if(i==j){
                break;
            }
            list[j]->next=list[i];
            j--;
        }
        list[i]->next=nullptr;
        head=list[0];
    }
};
```

148 排序链表

给你链表的头结点 `head` ，请将其按升序排列并返回排序后的链表。

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        return sortList(head,nullptr);
    }
    ListNode* sortList(ListNode *head,ListNode *tail){
        if(head==nullptr) return head;
        if(head->next==tail){
            head->next=nullptr;
            return head;
        }
        ListNode *slow=head,*fast=head;
        while (fast!=tail)
```

```
        {
            slow=slow->next;
            fast=fast->next;
            if(fast!=tail) fast=fast->next;
            /* code */
        }
        auto mid=slow;
        return merge(sortList(head,mid),sortList(mid,tail));
    }
    ListNode *merge(ListNode* head1,ListNode* head2){
        ListNode* dummyHead=new ListNode(0);
        auto temp=dummyHead,t1=head1,t2=head2;
        while (t1!=nullptr&&t2!=nullptr)
        {
            if(t1->val<=t2->val){
                temp->next=t1;
                t1=t1->next;
            }else{
                temp->next=t2;
                t2=t2->next;
            }
            temp=temp->next;
            /* code */
        }
        if(t1!=nullptr){
            temp->next=t1;
        }else if(t2!=nullptr){
            temp->next=t2;
        }
        return dummyHead->next;
    }

};
```

160 相交链表

给你两个单链表的头节点 `headA` 和 `headB` ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 `null` 。

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if(headA==nullptr||headB==nullptr) return nullptr;
        auto t1=headA,t2=headB;
        while(t1!=t2){
            t1=t1==nullptr?headB:t1->next;
            t2=t2==nullptr?headA:t2->next;
        }
        return t1;
    }
};
```

移除链表元素

给你一个链表的头节点 `head` 和一个整数 `val` ，请你删除链表中所有满足 `Node.val == val` 的节点，并返回新的头节点。

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
       ListNode dummy{0};
       auto cur=&dummy;
       while(head){
```

```
        while(head&&head->val==val){
            head=head->next;
        }
        cur->next=head;
        cur=cur->next;
        if(head)
        head=head->next;
       }
       return dummy.next;
    }
};
```

138 随即链表的复制

给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random ，该指针可以指向链表中的任何节点或空节点。

构造这个链表的深拷贝。深拷贝应该正好由 n 个全新节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 next 指针和 random 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。复制链表中的指针都不应指向原链表中的节点。

例如，如果原链表中有 X 和 Y 两个节点，其中 X.random -> Y 。那么在复制链表中对应的两个节点 x 和 y ，同样有 x.random -> y 。

返回复制链表的头节点。

用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：

val：一个表示 Node.val 的整数。
random_index：随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为 null 。

你的代码只接受原链表的头节点 head 作为传入参数。

```
class Solution {
public:
unordered_map<Node*,Node*> cache;
    Node* copyRandomList(Node* head) {
        if(head==nullptr) return nullptr;
        if(!cache.count(head)){
            Node* newHead=new Node(head->val);
            cache[head]=newHead;
            newHead->next=copyRandomList(head->next);
            newHead->random=copyRandomList(head->random);
        }
        return cache[head];
    }
};
```

328 奇偶链表

给定单链表的头节点 head ，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起，然后返回重新排序的列表。

第一个节点的索引被认为是奇数，第二个节点的索引为偶数，以此类推。

请注意，偶数组和奇数组内部的相对顺序应该与输入时保持一致。

你必须在 O(1) 的额外空间复杂度和 O(n) 的时间复杂度下解决这个问题。

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(head==nullptr) return head;
        auto evenHead=head->next;
        auto odd=head;
        auto even=evenHead;
        while(even!=nullptr&&even->next!=nullptr){
            odd->next=even->next;
            odd=odd->next;
```

```
            even->next=odd->next;
            even=even->next;
        }
        odd->next=evenHead;
        return head;
    }
};
```