2. 两数相加

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。

循环执行的条件是只要链表不为空或有进位就执行，每次统计节点（节点不为空）加进位的和，更新进位

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode();
        ListNode* p = dummy;
        int carry = 0;
        while (l1 || l2 || carry != 0) {
            int val = carry;
            if (l1 != nullptr) {
                val += l1->val;
                l1 = l1->next;
            }
            if (l2 != nullptr) {
                val += l2->val;
                l2 = l2->next;
            }
            carry = val / 10;
            val = val % 10;
            p->next = new ListNode(val);
            p = p->next;
        }
        return dummy->next;
    }
};
```

另一种写法，单独判断最后的carry

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* pre = new ListNode();
        ListNode* cur = pre;
        int carry = 0;
        while (l1 != nullptr || l2 != nullptr) {
            int x = l1 == nullptr ? 0 : l1->val;
            int y = l2 == nullptr ? 0 : l2->val;
            int val = x + y + carry;
            carry = val / 10;
            val = val % 10;
            cur->next = new ListNode(val);
            cur = cur->next;
            if (l1) l1 = l1->next;
            if (l2) l2 = l2->next;
        }
        if (carry > 0) {
            cur->next = new ListNode(carry);
        }
        return pre->next;
    }
};
```

146. LRU 缓存

请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。
实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以 **正整数** 作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1` 。
- `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值 `value` ；如果不存在，则向缓存中插入该组 `key-value` 。如果插入操作导致关键字数量超过 `capacity` ，则应该 **逐出** 最久未使用的关键字。

函数 `get` 和 `put` 必须以 `O(1)` 的平均时间复杂度运行。

```
class LRUCache {
public:
    class ListNode{
    public:
        int val, key;
        ListNode* pre, *next;
        ListNode() : val(-1), key(-1), pre(nullptr), next(nullptr) {}
        ListNode(int x, int y) : key(x), val(y), pre(nullptr), next(nullptr) {}
    }*head, *tail;
    int cap;
    int count;
    unordered_map<int, ListNode*> mp;
    LRUCache(int capacity) {
        count = 0;
        cap = capacity;
        head = new ListNode();
        tail = new ListNode();
        head->next = tail;
        tail->pre = head;
    }

    int get(int key) {
        if (mp.count(key) > 0) {
            remove(mp[key]);
            setHead(mp[key]);
            return mp[key]->val;
        }
        return -1;
    }

    void put(int key, int value) {
        if (mp.count(key) > 0) {
            mp[key]->val = value;
            remove(mp[key]);
            setHead(mp[key]);
        } else {
            mp[key] = new ListNode(key, value);
            count++;
            setHead(mp[key]);
        }
        if (count > cap) {
            mp.erase(tail->pre->key);
            ListNode* tmp = tail->pre;
            remove(tmp);
            delete(tmp);
            count--;
        }
    }
    void setHead(ListNode* node) {
        node->pre = head;
        node->next = head->next;
        head->next->pre = node;
        head->next = node;
    }
    void remove(ListNode* node) {
        node->pre->next = node->next;
        node->next->pre = node->pre;
    }
};
```

206. 反转链表

递归（好理解版本）

```
class Solution {
public:
    ListNode* reverse(ListNode* pre, ListNode* head) {
        if (head == nullptr) return pre;  //递归出口不要忘记
        ListNode* tmp = head->next;
        head->next = pre;
        pre = head;
        return reverse(pre, tmp);
    }
    ListNode* reverseList(ListNode* head) {
        return reverse(nullptr, head);
    }
};
```

递归（不好理解版本）

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) return head;
        ListNode* ret = reverseList(head->next);
        head->next->next = head;
        head->next = nullptr;  //别忘了给当前最后节点next置空
        return ret;   //每次返回的都是最后一个节点
    }
};
```

迭代

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr) return head;
        ListNode* pre = nullptr;
        while (head != nullptr) {
            ListNode* tmp = head->next;
            head->next = pre;
            pre = head;
            head = tmp;
        }
        return pre;
    }
};
```

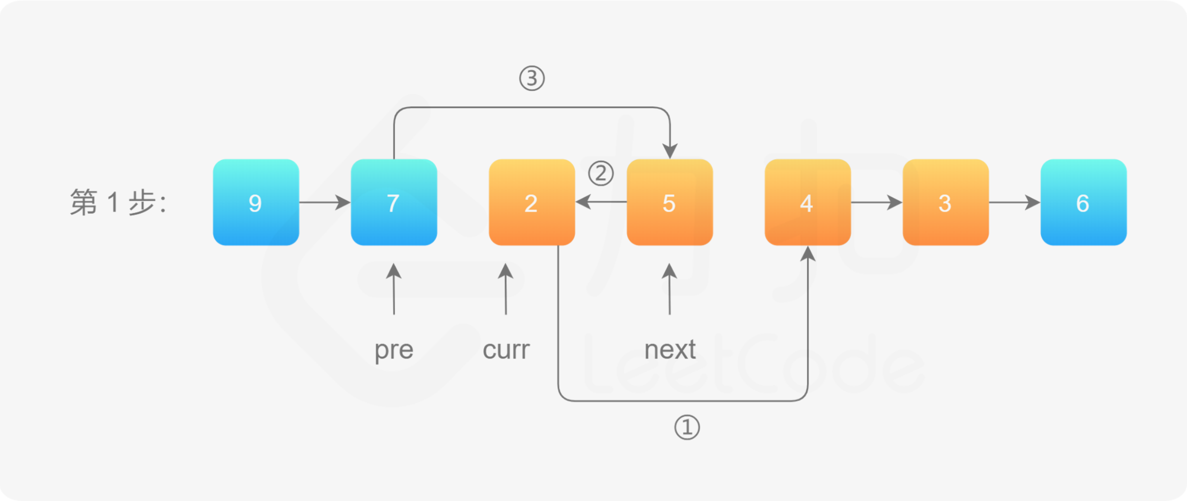
92. 反转链表 II

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** 。

递归法

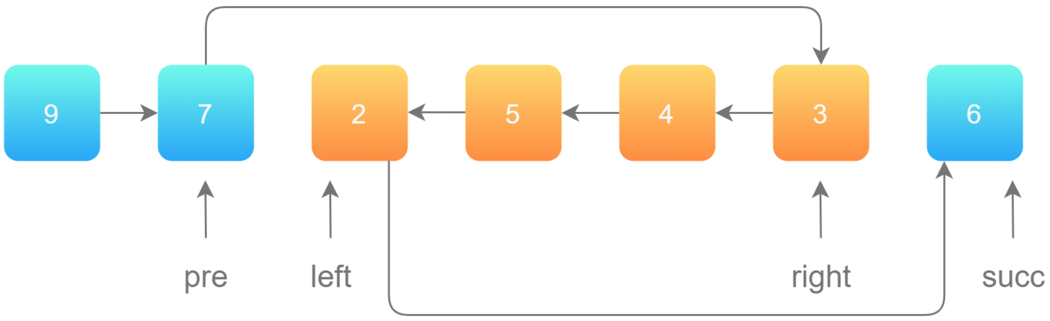
```
class Solution {
public:
    ListNode* succ = nullptr;
    ListNode* reverseN(ListNode* head, int n) {
        if (n == 1) {
            succ = head->next;
            return head;
        }
        ListNode* last = reverseN(head->next, n - 1);
        head->next->next = head;
        head->next = succ;
        return last;
    }
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if (left == 1) return reverseN(head, right);
        head->next = reverseBetween(head->next, left - 1, right - 1);
        return head;
    }
};
```

头插法



```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        ListNode* dummyNode = new ListNode(-1);
        dummyNode->next = head;
        ListNode* pre = dummyNode;
        for (int i = 0; i < left - 1; i++) {
            pre = pre->next;
        }
        ListNode* curr = pre->next;
        ListNode* next = curr->next;
        for (int i = 0; i < right - left; i++) {
            curr->next = next->next;
            next->next = pre->next;
            pre->next = next;
            next = curr->next;
        }
        return dummyNode->next;
    }
};
```

分割+翻转+拼接



```cpp
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr) return head;
        ListNode* pre = nullptr;
        while (head != nullptr) {
            ListNode* tmp = head->next;
            head->next = pre;
            pre = head;
            head = tmp;
        }
        return pre;
    }
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        ListNode* dummyNode = new ListNode(-1);
        dummyNode->next = head;
        ListNode* pre = dummyNode;
        for (int i = 0; i < left - 1; i++) {
            pre = pre->next;
        }
        ListNode* leftNode = pre->next;
        ListNode* rightNode = pre;
        for (int i = 0; i < right - left + 1; i++) {
            rightNode = rightNode->next;
        }
        ListNode* curr = rightNode->next;
        //切割链表
        pre->next = nullptr;
        rightNode->next = nullptr;
        //反转链表
        reverseList(leftNode);
        //拼接链表
        pre->next = rightNode;
        leftNode->next = curr;
        return dummyNode->next;

    }
};
```

25. K 个一组翻转链表

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。

利用翻转链表前N个节点做，统计好需要翻转几部分，迭代的去翻转即可

需要注意的点，head节点反翻转后就会到翻转部分的末尾，head的下一个节点即下一次翻转的开始

```
class Solution {
public:
    ListNode* curr = nullptr;
    ListNode* reverseN(ListNode* head, int N) {
        if (N == 1) {
            curr = head->next;
            return head;
        }
        ListNode* ret = reverseN(head->next, N - 1);
        head->next->next = head;
        head->next = curr;
        return ret;
    }
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* dummyNode = new ListNode();
        dummyNode->next = head;
        int count = 0;
        ListNode* cur = dummyNode;
        while (cur->next != nullptr) {
            cur = cur->next;
            ++count;
        }
        int a = count / k;
        ListNode* pre = dummyNode;
        while (a--) {
            pre->next = reverseN(head, k);
            pre = head;
            head = head->next;
        }
        return dummyNode->next;
    }
};
```

利用翻转链表做，分割大小为k的链表，翻转后再拼接，迭代执行

```
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        ListNode* ret = reverse(head->next);
        head->next->next = head;
        head->next = nullptr;
        return ret;
    }
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* dummyhead = new ListNode();
        dummyhead->next = head;
        ListNode* pre, *start, *end, *next;
        pre = dummyhead;
        end = dummyhead;
        while (end->next != nullptr) {
            for (int i = 0; i < k && end != nullptr; i++) {
                end = end->next;
            }
            if (end == nullptr) break;
            start = pre->next;
            next = end->next;
            end->next = nullptr;
            pre->next = reverse(start);
            start->next = next;
            pre = start;
            end = pre;
        }
        return dummyhead->next;
    }
};
```

21. 合并两个有序链表

迭代

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummynode = new ListNode();
        ListNode* cur = dummynode;
        while (l1 != NULL && l2 != NULL) {
            if (l1->val <= l2->val) {
                cur->next = l1;
                l1 = l1->next;
            } else {
                cur->next = l2;
                l2 = l2->next;
            }
            cur = cur->next;
        }
        cur->next = !l1 ? l2 : l1;
        return dummynode->next;
    }
};
```

递归

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;
        if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

23. 合并K个升序链表

分治合并

```cpp
class Solution {
public:
    ListNode* merge2Lists(ListNode* head1, ListNode* head2) {
        if (head1 == nullptr) return head2;
        if (head2 == nullptr) return head1;
        if (head1->val < head2->val) {
            head1->next = merge2Lists(head1->next, head2);
            return head1;
        } else {
            head2->next = merge2Lists(head1, head2->next);
            return head2;
        }
    }
    ListNode* merge(vector<ListNode*>& lists, int left, int right) {
        if (left == right) return lists[left];
        if (left > right)  return nullptr;
        int mid = (left + right) / 2;
        return merge2Lists(merge(lists, left, mid), merge(lists, mid + 1, right));
    }
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        return merge(lists, 0, lists.size() - 1);
    }
};
```

优先队列

关于优先队列自定义比较器：

如果直接将结构体放入priority_queue中，则需在结构体中重载<号（或是在结构体外重载<号），优先队列默认使用less<>

如果放入的不是某个结构体，则需定义结构体cmp并在其内重载小括号，并将cmp写到优先队列的第三个参数

关于sort自定义比较器：

如果使用结构体，需在结构体内重载小括号（返回值为bool），并将匿名对象（类名()）或对象实例放到sort第三个参数

如果使用函数，需自定义一个返回类型为bool值的函数，并将函数名放到sort第三个参数

好理解版本

```cpp
class Solution {
public:
    struct cmp{
        bool operator () (ListNode* a, ListNode* b) {
            return a->val > b->val;
        }
    };
    priority_queue<ListNode*, vector<ListNode*>, cmp> pq;
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        for (auto p : lists) {
            if (p) pq.push(p);
        }
        ListNode head, *cur = &head;
        while (!pq.empty()) {
            ListNode* f = pq.top(); pq.pop();
            cur->next = f;
            cur = cur->next;
            if (f->next) pq.push(f->next);
        }
        return head.next;
    }
};
```

不好理解版本

```cpp
class Solution {
public:
    struct Status {
        int val;
        ListNode *ptr;
        bool operator < (const Status &rhs) const {
            return val > rhs.val;
        }
    };

    priority_queue <Status> q;

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        for (auto node: lists) {
            if (node) q.push({node->val, node});
        }
        ListNode head, *tail = &head;
        while (!q.empty()) {
            auto f = q.top(); q.pop();
            tail->next = f.ptr;
            tail = tail->next;
            if (f.ptr->next) q.push({f.ptr->next->val, f.ptr->next});
        }
        return head.next;
    }
};
```

剑指 Offer 22. 链表中倒数第k个节点

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。

例如，一个链表有 `6` 个节点，从头节点开始，它们的值依次是 `1、2、3、4、5、6`。这个链表的倒数第 `3` 个节点是值为 `4` 的节点。

哈希表

```

class Solution {
public:
    ListNode* getKthFromEnd(ListNode* head, int k) {
        unordered_map<int, ListNode*> map;
        int i = 0;
        while (head != NULL) {
            map[++i] = head;
            head = head->next;
        }
        return map[i - k + 1];
    }
};
```

快慢指针

```
class Solution {
public:
    ListNode* getKthFromEnd(ListNode* head, int k) {
        ListNode* fast = head;
        ListNode* slow = head;
        while (k--) fast = fast->next;
        while (fast != NULL) {
            fast = fast->next;
            slow = slow->next;
        }
        return slow;
    }
};
```

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：

$L_0 \to L_1 \to … \to L_{n - 1} \to L_n$

请将其重新排列后变为：

$L_0 \to L_n \to L_1 \to L_{n - 1} \to L_2 \to L_{n - 2} \to …$

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。

寻找链表中点 + 链表逆序 + 合并链表

```cpp
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        ListNode* ret = reverse(head->next);
        head->next->next = head;
        head->next = nullptr;
        return ret;
    }
    void reorderList(ListNode* head) {
        if (head == nullptr) return;
        ListNode* fast = head;
        ListNode* slow = head;
        while (fast->next != nullptr && fast->next->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }
        ListNode* midHead = slow->next;
        slow->next = nullptr;
        midHead = reverse(midHead);
        ListNode* p1 = head;
        ListNode* p2 = midHead;
        while (p2 != nullptr) {
            ListNode* tmp1 = p1->next;
            ListNode* tmp2 = p2->next;
            p1->next = p2;
            p2->next = tmp1;
            p1 = tmp1;
            p2 = tmp2;
        }
        return;
    }
};
```

线性表

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

707. 设计链表

设计链表的实现。您可以选择使用单链表或双链表。单链表中的节点应该具有两个属性：`val` 和 `next`。`val` 是当前节点的值，`next` 是指向下一个节点的指针/引用。如果要使用双向链表，则还需要一个属性 `prev` 以指示链表中的上一个节点。假设链表中的所有节点都是 0-index 的。

在链表类中实现这些功能：

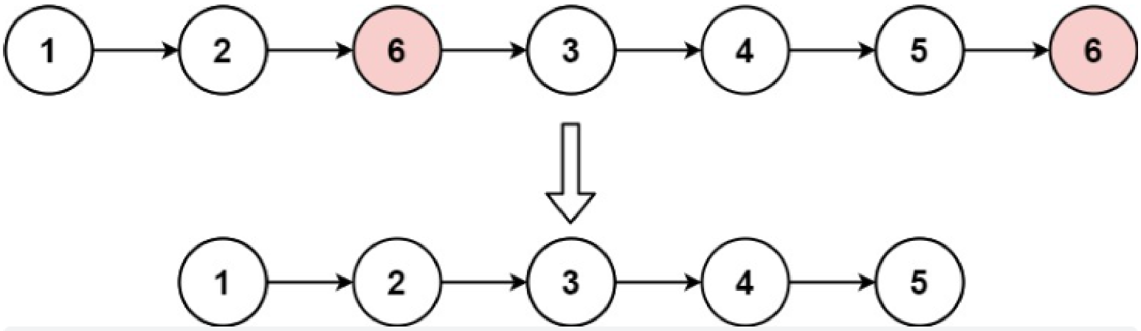
- get(index)：获取链表中第 `index` 个节点的值。如果索引无效，则返回 `-1` 。
- addAtHead(val)：在链表的第一个元素之前添加一个值为 `val` 的节点。插入后，新节点将成为链表的第一个节点。
- addAtTail(val)：将值为 `val` 的节点追加到链表的最后一个元素。
- addAtIndex(index,val)：在链表中的第 `index` 个节点之前添加值为 `val` 的节点。如果 `index` 等于链表的长度，则该节点将附加到链表的末尾。如果 `index` 大于链表长度，则不会插入节点。如果`index`小于0，则在头部插入节点。
- deleteAtIndex(index)：如果索引 `index` 有效，则删除链表中的第 `index` 个节点。

```
class MyLinkedList {
public:
    struct LinkedNode {
        int val;
        LinkedNode* next;
        LinkedNode(int x) : val(x), next(nullptr) {}
        };

    MyLinkedList() {
        dummyhead = new LinkedNode(0);
        size = 0;
    }

    int get(int index) {
        if (index > (size - 1) || index < 0) {
            return -1;
        }
        LinkedNode* cur = dummyhead->next;
        while (index--) {
            cur = cur->next;
        }
        return cur->val;
    }

    void addAtHead(int val) {
        LinkedNode* newNode = new LinkedNode(val);
        newNode->next = dummyhead->next;
        dummyhead->next = newNode;
        size++;
    }

    void addAtTail(int val) {
        LinkedNode* newNode = new LinkedNode(val);
        LinkedNode* cur = dummyhead;
        while (cur->next != nullptr) {
            cur = cur->next;
        }
        cur->next = newNode;
        size++;
    }

    void addAtIndex(int index, int val) {
        if (index > size) {
            return;
        }
        LinkedNode* newNode = new LinkedNode(val);
        LinkedNode* cur = dummyhead;
        while (index--) {
            cur = cur->next;
        }
        newNode->next = cur->next;
        cur->next = newNode;
        size++;
    }

    void deleteAtIndex(int index) {
        if (index >= size || index < 0) {
            return;
        }
        LinkedNode* cur = dummyhead;
        while(index--) {
            cur = cur->next;
        }
        LinkedNode* tmp = cur->next;
        cur->next = cur->next->next;
        delete tmp;
        size--;
    }
private:
    int size;
    LinkedNode* dummyhead;
};
```

203. 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** 。

示例 1：



递归

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        if (head == nullptr) return head;
        if (head->val == val) {
            return removeElements(head->next, val);
        } else {
            head->next = removeElements(head->next, val);
            return head;
        }
    }
};
```

迭代

```
class Solution {
```

```cpp
public:
    ListNode* removeElements(ListNode* head, int val) {
        if (head == nullptr) return head;
        ListNode* dummy = new ListNode();
        dummy->next = head;
        ListNode* cur = dummy;
        while (cur->next != nullptr) {
            if (cur->next->val == val) {
                cur->next = cur->next->next;
            } else {
                cur = cur->next;
            }
        }
        return dummy->next;
    }
};
```

83. 删除排序链表中的重复元素

给定一个已排序的链表的头 `head` ， *删除所有重复的元素，使每个元素只出现一次* 。返回 *已排序的链表* 。

双指针

```cpp
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return head;
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != nullptr) {
            if (fast->val != slow->val) {
                slow->next = fast;
                slow = slow->next;
                fast = fast->next;
            } else {
                fast = fast->next;
            }
        }
        slow->next = nullptr;
        return head;
    }
};
```

迭代

```cpp
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head)  return head;
        ListNode* cur = head;
        while (cur->next) {
            if (cur->val == cur->next->val) {
                cur->next = cur->next->next;
            } else {
                cur = cur->next;
            }
        }
        return head;
    }
};

```

递归（没必要）

```
class Solution {
public:
    ListNode* dfs(ListNode* head, int x) {
        while (head && head->val == x) {
            head = head->next;
        }
        if (!head) return head;
        head->next = dfs(head->next, head->val);
        return head;
    }
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return head;
        head->next = dfs(head->next, head->val);
        return head;
    }
};
```

82. 删除排序链表中的重复元素 II

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* 。

迭代

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == nullptr) return head;
        ListNode* dummy = new ListNode(101);
        dummy->next = head;
        ListNode* cur = dummy;
        ListNode* pre = dummy;
        while (cur != nullptr && cur->next != nullptr) {
            if (cur->val != cur->next->val) {
                pre = cur;
                cur = cur->next;
            } else {
                int x = cur->val;
                while (cur != nullptr && cur->val == x) {
                    cur = cur->next;
                }
                pre->next = cur;
            }
        }
        return dummy->next;
    }
};
```

递归

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head || !head->next) {
            return head;
        }
        if (head->val != head->next->val) {
            head->next = deleteDuplicates(head->next);
        } else {
            ListNode* move = head->next;
            while (move != nullptr && head->val == move->val) {
                move = move->next;
            }
            return deleteDuplicates(move);
        }
        return head;
    }
};
```

利用计数，两次遍历

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        unordered_map<int, int> m;
        ListNode dummy(0);
        ListNode* dummy_move = &dummy;
        ListNode* move = head;
        while (move) {
            m[move->val]++;
            move = move->next;
        }
        move = head;
        while (move) {
            if (m[move->val] == 1) {
                dummy_move->next = move;
                dummy_move = dummy_move->next;
            }
            move = move->next;
        }
        dummy_move->next = nullptr;
        return dummy.next;
    }
};
```

19. 删除链表的倒数第 N 个结点

给你一个链表，删除链表的倒数第 `n` 个结点，并且返回链表的头结点。

双指针

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode();
        dummy->next = head;
        ListNode* p1 = dummy;
        ListNode* p2 = dummy;
        for (int i = 0; i < n; ++i) {
            p1 = p1->next;
        }
        while (p1->next != nullptr) {
            p1 = p1->next;
            p2 = p2->next;
        }
        ListNode* tmp = p2->next;
        p2->next = p2->next->next;
        delete(tmp);
        return dummy->next;
    }
};
```

递归

```
class Solution {
public:
    int dfs(ListNode* head, int n) {
        if (head == nullptr) return 0;
        int cnt = dfs(head->next, n);
        if (cnt == n) {
            head->next = head->next->next;
        }
        return cnt + 1;
    }
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        int cnt = dfs(head, n);
        if (cnt == n) {
            return head->next;
        }
        return head;
    }
};
```

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）。

迭代

```cpp
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if (head == NULL || head->next == NULL) return head;
        ListNode* dummyHead = new ListNode();
        dummyHead->next = head;
        ListNode* cur = dummyHead;
        while (cur->next != NULL && cur->next->next != NULL) {
            ListNode* tmp1 = cur->next;
            ListNode* tmp2 = cur->next->next;
            tmp1->next = tmp2->next;
            cur->next = tmp2;
            tmp2->next = tmp1;
            cur = tmp1;
        }
        return dummyHead->next;
    }
};
```

递归

```cpp
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if (head == NULL || head->next == NULL) return head;
        ListNode* tmp = head->next;
        head->next = swapPairs(head->next->next);
        tmp->next = head;
        return tmp;
    }
};
```

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` 。

快慢指针

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

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

哈希表

```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        unordered_set<ListNode*> seen;
        while (head != nullptr) {
            if (seen.count(head)) {
                return true;
            }
            seen.insert(head);
            head = head->next;
        }
        return false;
    }
};
```

142. 环形链表 II

双指针

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

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast = head, slow = head;
        while (true) {
            if (fast == null || fast.next == null) return null;
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        fast = head;
        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }
        return fast;
    }
}
```

234. 回文链表

234. 回文链表 labuladong 题解 思路

难度 简单 1483

给你一个单链表的头节点 `head` ，请你判断该链表是否为回文链表。如果是，返回 `true` ；否则，返回 `false` 。

快慢指针+反转链表

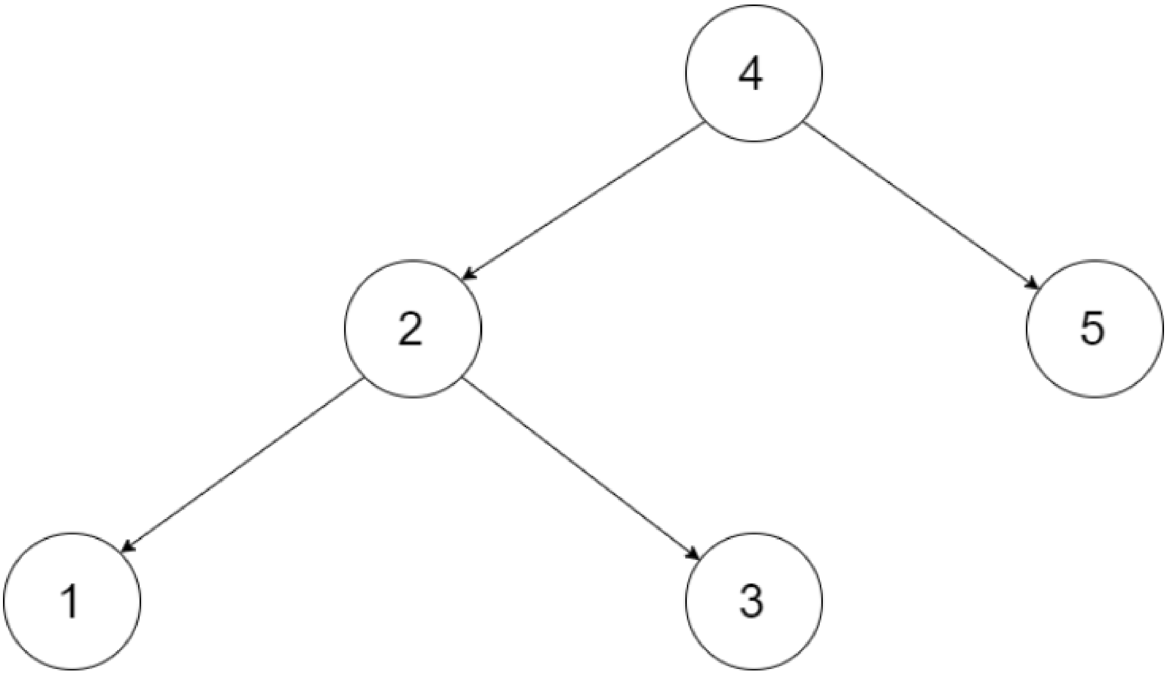
```cpp
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        ListNode* cur = head, *pre = NULL;
        while (cur != NULL) {
            ListNode* tmp = cur->next;
            cur->next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }

    bool isPalindrome(ListNode* head) {
        ListNode *fast = head, *slow = head;
        while (fast != NULL && fast->next != NULL) {
            fast = fast->next->next;
            slow = slow->next;
        }
        if (fast != NULL) slow = slow->next;
        ListNode* right =  reverse(slow);
        ListNode* left = head;
        while (right != NULL) {
            if (left->val != right->val) return false;
            left = left->next;
            right = right->next;
        }
        return true;
    }
};
```

使用辅助数组

```cpp
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        vector<int> v;
        while (head != NULL) {
            v.push_back(head->val);
            head = head->next;
        }
        int l = 0, r = v.size() - 1;
        while (l < r) {
            if (v[l] == v[r]) {
                l++, r--;
                continue;
            }
            return false;
        }
        return true;
    }
};
```

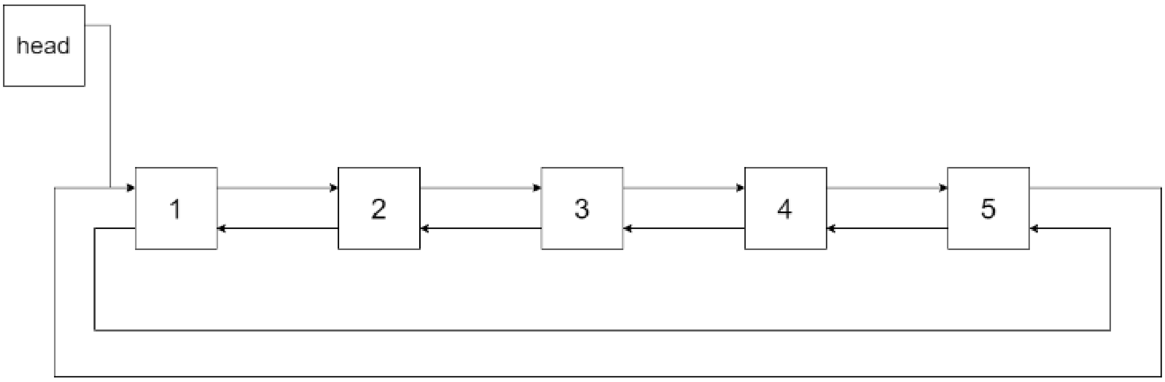
剑指 Offer 36. 二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。



我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。



特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

宏观递归

```
class Solution {
public:
    Node* treeToDoublyList(Node* root) {
        if (!root) return root;
        Node* leftHead = treeToDoublyList(root->left);
        Node* rightHead = treeToDoublyList(root->right);
        Node* leftTail, *rightTail;
        if (leftHead) {
            leftTail = leftHead->left;
            leftTail->right = root;
            root->left = leftTail;
        } else {
            leftTail = leftHead = root;
        }
        if (rightHead) {
            rightTail = rightHead->left;
            root->right = rightHead;
            rightHead->left = root;
        } else {
            rightTail = rightHead = root;
        }
        leftHead->left = rightTail;
        rightTail->right = leftHead;
        return leftHead;
    }
};
```

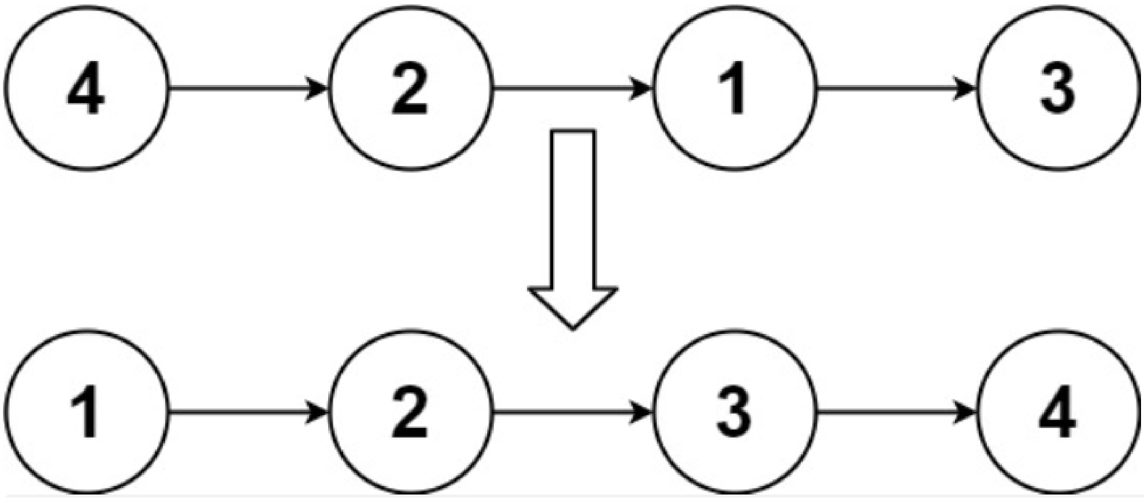
中序遍历

```
class Solution {
public:
    Node* treeToDoublyList(Node* root) {
        if (!root) return NULL;
        dfs(root);
        head->left = pre;
        pre->right = head;
        return head;
    }
private:
    Node* pre, *head;
    void dfs(Node* node) {
        if (!node) return;
        dfs(node->left);
        if (pre != NULL) pre->right = node;
        else head = node;
        node->left = pre;
        pre = node;
        dfs(node->right);
    }
};
```

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** 。

示例 1：



哈希表

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr) return head;
        multiset<int> set;
        ListNode* cur = head;
```

```cpp
        while (cur != nullptr) {
            set.insert(cur->val);
            cur = cur->next;
        }
        cur = head;
        for (auto& i : set) {
            cur->val = i;
            cur = cur->next;
        }

        return head;
    }
};
```

```cpp
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr) return head;
        multimap<int, ListNode*> map;
        ListNode* cur = head;
        while (cur != nullptr) {
            ListNode* tmp = cur;
            cur = cur->next;
            tmp->next = nullptr;
            map.insert({tmp->val, tmp});

        }
        ListNode* Head = new ListNode();
        cur = Head;
        for (auto pair : map) {
            cur->next = pair.second;
            cur = cur->next;
        }

        return Head->next;
    }
};
```

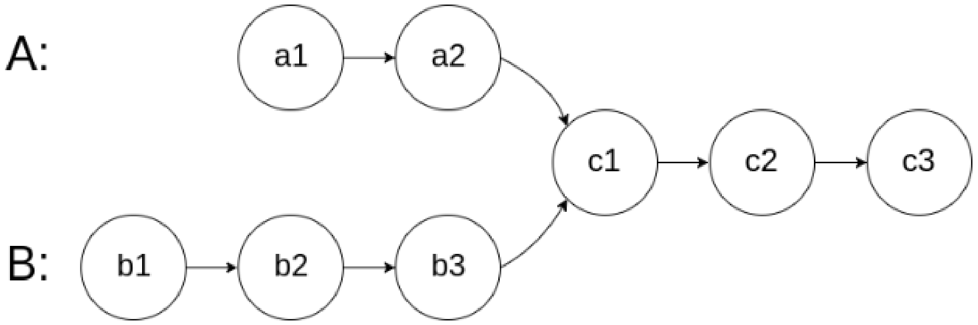
归并排序

```cpp
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) return head;
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }
        ListNode* tmp = slow->next;
        slow->next = nullptr;
        ListNode* left = sortList(head);
        ListNode* right = sortList(tmp);
        ListNode* res = new ListNode();
        ListNode* cur = res;
        while (right != nullptr && left != nullptr) {
            if (left->val < right->val) {
                cur->next = left;
                left = left->next;
            } else {
                cur->next = right;
                right = right->next;
            }
            cur = cur->next;
        }
        cur->next = left == nullptr ? right : left;
        return res->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` 。

图示两个链表在节点 `c1` 开始相交：



题目数据 **保证** 整个链式结构中不存在环。

注意，函数返回结果后，链表必须 **保持其原始结构** 。

双指针

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode* curA = headA;
        ListNode* curB = headB;
        while (curA != curB) {
            curA = curA == NULL ? headB : curA->next;
            curB = curB == NULL ? headA : curB->next;
        }
        return curA;
    }
};
```

138. 复制带随机指针的链表

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` 。

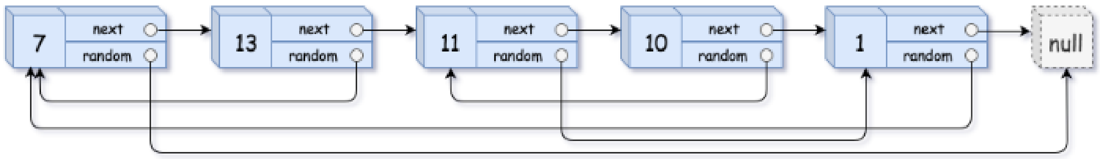
返回复制链表的头节点。

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。

示例 1：



```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

拆分链表

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return NULL;
        //交叉连接到一起
        for (Node* node = head; node != NULL; node = node->next->next) {
            Node* nodeNew = new Node(node->val);
            nodeNew->next = node->next;
            node->next = nodeNew;
        }
        //构造random
        for (Node* node = head; node != NULL; node = node->next->next) {
            Node* nodeNew = node->next;
            nodeNew->random = (node->random == NULL) ? NULL : node->random->next;
        }
        //拆分链表，构建next
        Node* headNew = head->next;
        for (Node* node = head; node != NULL; node = node->next) {
            Node* nodeNew = node->next;
            node->next = nodeNew->next;
            nodeNew->next = (node->next == NULL) ? NULL : node->next->next;
        }
        return headNew;
    }
};
```

哈希表

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

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (!head) return NULL;
        unordered_map<Node*, Node*> map;
        Node* cur = head;
        while (cur != NULL) {
            map[cur] = new Node(cur->val);
            cur = cur->next;
        }
        cur = head;
        while (cur != NULL) {
            map[cur]->next = map[cur->next];
            map[cur]->random = map[cur->random];
            cur = cur->next;
        }
        return map[head];
    }
};
```