常见双指针的操作

倒数第 k 个节点

```cpp
class Solution {
public:
    ListNode* getKthFromEnd(ListNode* head, int k) {
        ListNode *p = head, *q = head; //初始化
        while(k--) {   //将 p指针移动 k 次
            p = p->next;
        }
        while(p != nullptr) {//同时移动，直到 p == nullptr
            p = p->next;
            q = q->next;
        }
        return q;
    }
};
```

判断链表是否有环

```cpp
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {

        ListNode *slow = head;
        ListNode *fast = head;
        while(fast != nullptr && fast->next!= nullptr) {
            //慢指针每次"迈一步"
            slow = slow->next;
            //快指针每次"迈两步"
            fast = fast->next->next;
            if(fast == slow) {
                // 有环
                //指针p从链表首节点出发
                fast = head;
                while( fast != slow ){
                    //指针p和慢指针每次都迈一步
                    slow = slow->next;
                    fast = fast->next;
                }
                //指针p和慢指针会在“环开始的节点”相遇
                return fast;
            }
        }
        return nullptr;
    }
};
```

删除链表中间的元素

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;

    }
};
```

清除链表中的重复元素

```
struct ListNode {
        int val;
        ListNode *next;
         ListNode(int x) : val(x), next(NULL) {}
     };

class Solution {
public:
    ListNode* removeDuplicateNodes(ListNode* head) {
        ListNode* pre= nullptr;
        ListNode* cur= head;
        unordered_set<int> visited;
        while (cur!= nullptr){
            if(visited.find(cur->val)!=visited.end()){
                // 节点在set已经存在
                pre->next=cur->next;
            }else{
                visited.emplace(cur->val);
                pre=cur;
            }
            cur=cur->next;
        }
        return head;
    }
};
```

分割链表（经典双指针操作）

```
// https://leetcode-cn.com/problems/partition-list-lcci/submissions/
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {

        ListNode* small = new ListNode(0);
        ListNode* smallHead = small;
        ListNode* large = new ListNode(0);
        ListNode* largeHead = large;


        while (head!= nullptr) {
            if(head->val<x){
                small->next = head;
                small = small->next;
            }else{
                large->next = head;
                large = large->next;
            }
            head=head->next;
        }
        large->next = nullptr;
        small->next = largeHead->next;
        return  smallHead->next;
    }
};
```

链表求和

```
// 从两个链表头开始相加，处理进位（单位之和大于10的问题）。创建新的链表节点。然后连接节点
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* head = new ListNode(0);
        ListNode *cur = head;
        int carry = 0, sum = 0;
        while (l1 || l2 || carry ){
            sum=0;
           if(l1!= nullptr){
               sum+=l1->val;
               l1=l1->next;
           }
            if(l2!= nullptr){
                sum+=l2->val;
                l2=l2->next;
            }
            sum+=carry;
            ListNode * tmp=new ListNode(sum % 10);
            carry = sum / 10;
            cur->next=tmp;
            cur=cur->next;
        }

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

回文链表

> 找链表中点和反转链表部分节点

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        // 快慢指针找中点
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }
        // slow is the mid of list
        // 反转后半部分
        ListNode* pre = nullptr;
        while (slow != nullptr) {
            ListNode* tmp = slow->next;
            slow->next = pre;
            pre = slow;
            slow = tmp;
        }
        ListNode* node=head;
        while (pre!= nullptr){
            if(pre->val!=node->val){
                return false;
            }
            pre=pre->next;
            node=node->next;

        }
        return true;
    }
};
```

反转链表

```
ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* curr = head;
    while (curr != nullptr) {
        ListNode* nextTemp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}
```

K个一组翻转链表

```

// 反转链表前 N 个节点
class Solution3 {
private:
    ListNode* nextCur= nullptr; // 后驱节点
public:
    ListNode *reverseList(ListNode *head,int n) {
       reverseN(head,n);
    }
    ListNode * reverseN(ListNode* head, int n){
        if(n==1){
            nextCur=head->next;
            return head;
        }
        // 以 head.next 为起点，需要反转前 n - 1 个节点
        ListNode * last= reverseN(head->next,n-1);
        head->next->next=head;
        head->next=nextCur;
        return last;
    }
};



// 反转链表的一部分
class Solution4 {
private:
    ListNode* nextCur= nullptr; // 后驱节点
public:
    // [m,n] 索引从1开始
    ListNode *reverseBetween(ListNode *head,int m,int n) {
        if(m==1){

            return reverseN(head,n);
        }
        head->next= reverseBetween(head->next,m-1,n-1);
        return head;
    }

    ListNode * reverseN(ListNode* head, int n){
        if(n==1){
            nextCur=head->next;
            return head;
        }
        // 以 head.next 为起点，需要反转前 n - 1 个节点
        ListNode * last= reverseN(head->next,n-1);
        head->next->next=head;
        head->next=nextCur;
        return last;
    }
};
```

两个链表是否相交

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

删除链表中的最小值

```
//
// Created by yjs on 23-7-29.
//
#include <bits/stdc++.h>
#include <iostream>

using namespace std;
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};


class Solution{

public:

    ListNode * deleteListNode(ListNode* head){
        ListNode * prevMin=head;
        ListNode * Min=head;
        ListNode * cur=head;
        while(cur->next!= nullptr){
            cout << " node is "<<cur->val<<endl;
            if(Min->val>cur->next->val){
                prevMin=cur;
                Min=prevMin->next;
            }
            cur=cur->next;
        }
        cout << "delete node is "<<Min->val<<endl;
        cout << "prev min is " << prevMin->val <<endl;
        prevMin->next=Min->next;

        return  head;
    }



};
```

合并两个升序链表

```
ListNode* merge(ListNode* head1, ListNode* head2) {
        ListNode* dummyHead = new ListNode(0);
        ListNode* temp = dummyHead, *temp1 = head1, *temp2 = head2;
        while (temp1 != nullptr && temp2 != nullptr) {
            if (temp1->val <= temp2->val) {
                temp->next = temp1;
                temp1 = temp1->next;
            } else {
                temp->next = temp2;
                temp2 = temp2->next;
            }
            temp = temp->next;
        }
        if (temp1 != nullptr) {
            temp->next = temp1;
        } else if (temp2 != nullptr) {
            temp->next = temp2;
        }
        return dummyHead->next;
    }
```