

链表算法

```
class MyLinkedList {
public:
    // 定义链表节点结构体
    struct ListNode {
        int val;
        ListNode* next;
        ListNode(int val):val(val), next(nullptr){}
    };

    // 初始化链表
    MyLinkedList() {
        _dummyHead = new ListNode(0); // 这里定义的头结点 是一个虚拟头结点，而不是真正的链表头结点
        _size = 0;
    }

    // 获取到第index个节点数值，如果index是非法数值直接返回-1， 注意index是从0开始的，第0个节点就是头结点
    int get(int index) {
        if (index > (_size - 1) || index < 0) {
            return -1;
        }
        ListNode* cur = _dummyHead->next;
        while(index--){ // 如果--index 就会陷入死循环
            cur = cur->next;
        }
        return cur->val;
    }

    // 在链表最前面插入一个节点，插入完成后，新插入的节点为链表的新的头结点
    void addAtHead(int val) {
        ListNode* newNode = new ListNode(val);
        newNode->next = _dummyHead->next;
        _dummyHead->next = newNode;
        _size++;
    }
}
```

// 在链表最后面添加一个节点

```
void addAtTail(int val) {  
    ListNode* newNode = new ListNode(val);  
    ListNode* cur = _dummyHead;  
    while (cur->next != nullptr) {  
        cur = cur->next;  
    }  
    cur->next = newNode;  
    _size++;  
}
```

// 在第index个节点之前插入一个新节点，例如index为0，那么新插入的节点为链表的新头节点。

// 如果index 等于链表的长度，则说明是新插入的节点为链表的尾结点

// 如果index大于链表的长度，则返回空

```
void addAtIndex(int index, int val) {  
    if (index > _size) {  
        return;  
    }  
    ListNode* newNode = new ListNode(val);  
    ListNode* cur = _dummyHead;  
    while (index-- > 0) {  
        cur = cur->next;  
    }  
    newNode->next = cur->next;  
    cur->next = newNode;  
    _size++;  
}
```

```
// 删除第index个节点, 如果index 大于等于链表的长度, 直接return, 注意index是从0开始的
void deleteAtIndex(int index) {
    if (index >= _size || index < 0) {
        return;
    }
    ListNode* cur = _dummyHead;
    while(index-- > 0) {
        cur = cur->next;
    }
    ListNode* tmp = cur->next;
    cur->next = cur->next->next;
    delete tmp;
    _size--;
}
```cpp
private:
 int _size;
 ListNode* _dummyHead;

};
```

```
/* 双链表 */

class MyLinkedList {
public:
 /** Initialize your data structure here. */
 MyLinkedList() {
 size = 0;
 dummy_head = new DoublyListNode(0);
 dummy_tail = new DoublyListNode(0);
 dummy_head->next = dummy_tail;
 dummy_tail->prev = dummy_head;
 }

 /** Get the value of the index-th node in the linked list. If the index is invalid, return -1. */
 int get(int index) {
 if (index < 0 || index >= size) {
 return -1;
 }

 DoublyListNode *curr = nullptr;
 if (index + 1 < size - index) {
 curr = dummy_head;
 for (int i = 0; i < index + 1; ++i) {
 curr = curr->next;
 }
 } else {
 curr = dummy_tail;
 for (int i = 0; i < size - index; ++i) {
 curr = curr->prev;
 }
 }

 return curr->val;
 }

 /** Add a node of value val before the first element of the linked list. After the insertion, the new node
 void addAtHead(int val) {
 DoublyListNode *pred = dummy_head; // Predecessor
 DoublyListNode *succ = dummy_head->next; // Successor

 DoublyListNode *add_node = new DoublyListNode(val);

 add_node->next = succ;
 add_node->prev = pred;
 succ->prev = add_node;
 pred->next = add_node;

 size++;
 }
}
```

```
}

/** Append a node of value val to the last element of the linked list. */
void addAtTail(int val) {
 DoublyListNode *pred = dummy_tail->prev; // Predecessor
 DoublyListNode *succ = dummy_tail; // Successor

 DoublyListNode *add_node = new DoublyListNode(val);

 add_node->next = succ;
 add_node->prev = pred;
 succ->prev = add_node;
 pred->next = add_node;

 size++;
}

/** Add a node of value val before the index-th node in the linked list. If index equals to the length of
void addAtIndex(int index, int val) {
 if (index > size) return;

 if(index < 0) index = 0;

 DoublyListNode *curr = nullptr;
 if (index + 1 < size - index) {
 curr = dummy_head;
 for (int i = 0; i < index + 1; ++i) {
 curr = curr->next;
 }
 } else {
 curr = dummy_tail;
 for (int i = 0; i < size - index; ++i) {
 curr = curr->prev;
 }
 }

 DoublyListNode *pred = curr->prev; // Predecessor
 DoublyListNode *succ = curr; // Successor

 DoublyListNode *add_node = new DoublyListNode(val);

 add_node->next = succ;
 add_node->prev = pred;
 succ->prev = add_node;
 pred->next = add_node;

 size++;
}
```

```

 }

 /** Delete the index-th node in the linked list, if the index is valid. */
 void deleteAtIndex(int index) {
 if (index < 0 || index >= size) return;

 DoublyListNode *curr = nullptr;
 if (index + 1 < size - index) {
 curr = dummy_head;
 for (int i = 0; i < index + 1; ++i) {
 curr = curr->next;
 }

 } else {
 curr = dummy_tail;
 for (int i = 0; i < size - index; ++i) {
 curr = curr->prev;
 }
 }

 DoublyListNode *deletedNode = curr;
 DoublyListNode *pred = curr->prev; // Predecessor
 DoublyListNode *succ = curr->next; // Successor

 succ->prev = pred;
 pred->next = succ;

 delete deletedNode;
 deletedNode = nullptr;

 --size;
 }

private:
 // Definition for doubly-linked list.
 struct DoublyListNode {
 int val;
 DoublyListNode *next, *prev;
 DoublyListNode(int x) : val(x), next(NULL), prev(NULL) {}
 };

 DoublyListNode *dummy_head; // 哑头结点
 DoublyListNode *dummy_tail; // 哑尾结点

 int size; // 链表的长度
};

```

```
/** 迭代 删除值为val的结点 */
class Solution {
public:
 ListNode* deleteNode(ListNode* head, int val) {
 ListNode* dummy = new ListNode(-1);
 dummy->next = head; //建立虚拟头节点
 ListNode* cur = dummy;
 while(cur->next)
 if(cur->next->val == val) cur->next = cur->next->next;
 else cur = cur->next;
 return dummy->next;
 }
};

// 递归 删除 值为val的节点
class Solution {
public:
 ListNode* removeElements(ListNode* head, int val) {
 //1、递归边界
 if(!head) return nullptr;
 //2、递去：直到到达链表尾部才开始删除重复元素
 head->next = removeElements(head->next, val);
 //3、递归式：相等就是删除head，不相等就不用删除
 return head->val == val ? head->next : head;
 }
};
```

```
/* 用delete删除被删节点 */
class Solution {
public:
 ListNode* removeElements(ListNode* head, int val) {
 ListNode* sentinel = new ListNode(0);
 sentinel->next = head;

 ListNode *prev = sentinel, *curr = head, *toDelete = nullptr;
 while (curr != nullptr) {
 if (curr->val == val) {
 prev->next = curr->next;
 toDelete = curr;
 } else prev = curr;

 curr = curr->next;

 if (toDelete != nullptr) {
 delete toDelete;
 toDelete = nullptr;
 }
 }

 ListNode *ret = sentinel->next;
 delete sentinel;
 return ret;
 }
};
```



//反转链表

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

// 递归解 (反转链表)
class Solution {
public:
 ListNode* reverseList(ListNode* head) {
 // 递归终止条件是当前为空, 或者下一个为空
 if(!head || !head -> next) return head;
 // 递归调用来反转每一个结点
 ListNode *curr = reverseList(head -> next);
 // 每一个结点都是这样反转的
 head -> next -> next = head;
 // 防止链表循环, 需要将head->next置为空
 head -> next = nullptr;
 // 每层递归函数都返回cur, 也就是最后一个结点
 return curr;
 }
};
```

```

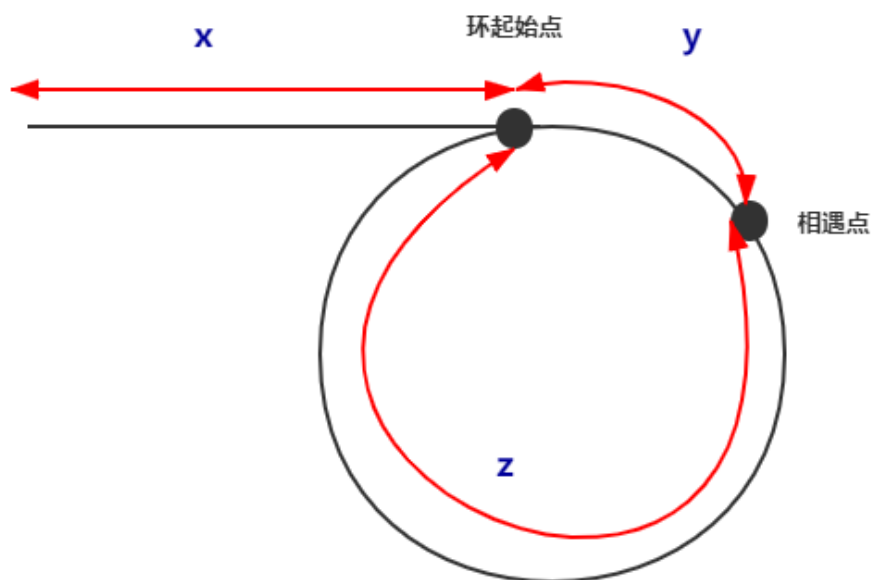
/**
 * 判断链表中是否有环
 */
class Solution {
public:
 //哈希表
 bool hasCycle(ListNode *head) {
 set<ListNode*> s;
 while(head != nullptr){
 if(s.insert(head).second==false){
 return true;
 }else{
 s.insert(head);
 }
 head = head->next;
 }
 return false;
 }

 // 快慢指针
 bool hasCycle(ListNode *head) {
 if(!head || !head->next) return false;
 ListNode *slow = head;
 ListNode *fast = head;
 while(fast != nullptr && fast->next != nullptr){
 slow = slow->next;
 fast = fast->next->next;
 if(fast == slow){
 return true;
 }
 }
 return false;
 }
};

/**
 * 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。
 * 为了表示给定链表中的环，我们使用整数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。
 * 说明：不允许修改给定的链表。
 * 示例 1：
 * 输入：head = [3,2,0,-4], pos = 1
 * 输出：tail connects to node index 1
 * 解释：链表中有一个环，其尾部连接到第二个节点。

 * 算法思想：快慢指针相遇的时候，此时从node1从head出发，node2从fast出发，相遇即为入口
 */

```



相遇时:

**slow**走过的路程:  $x+y$

**fast**走过的路程:  $x+y+z+y$

又因为**fast**走过的路程是**slow**的两倍

$$2*(x+y) = x+y+z+y \Rightarrow x=z$$

所以从head到环起始点的距离  
= 从相遇点到环起始点的距离

相遇时, 令其中一个指针指向head, 然后两个指针同步往后指, 再相遇时即为环起始点

```
class Solution {
public:
 ListNode *detectCycle(ListNode *head) {
 ListNode* slow = head;
 ListNode* fast = head;
 while(fast != nullptr && fast->next != nullptr){
 slow = slow->next;
 fast = fast->next->next;
 if(slow == fast){
 ListNode* node1 = fast;
 ListNode* node2 = head;
 while(node1 != node2){
 node1 = node1->next;
 node2 = node2->next;
 }
 return node2;
 }
 }
 return NULL;
 }
};

/* 编写一个程序，找到两个单链表相交的起始节点。 */
//算法思想：为了和你相遇，即使我走过了我所有的路，我也愿意走一遍你走过的路。

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

```
/* 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。

*示例：
*给定一个链表： 1->2->3->4->5， 和 n = 2。
*当删除了倒数第二个节点后，链表变为 1->2->3->5。
*/

//算法思想：快慢指针，相距 n个单位,当fast到达表尾时,slow->next 即为要删除的点
class Solution {
public:
 ListNode* removeNthFromEnd(ListNode* head, int n) {
 ListNode *dummyHead = new ListNode(0);
 dummyHead->next = head;
 ListNode *slow = dummyHead;
 ListNode *fast = dummyHead;
 for(int i=0; i<n+1 ; i++){
 fast = fast->next;
 }
 while(fast){
 fast = fast->next;
 slow = slow->next;
 }
 ListNode *delNode = slow->next;
 slow->next = delNode->next;
 delete delNode;

 ListNode *newHead = dummyHead->next;
 delete dummyHead;
 return newHead;
 }
};
```

```
/* 合并两个有序链表 */

class Solution {
public:
 ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
 ListNode *dummyHead = new ListNode(-1);
 ListNode *p = dummyHead;

 while(l1 != nullptr && l2 != nullptr){
 if(l1->val < l2->val)
 {
 p->next = l1;
 l1=l1->next;
 }else{
 p->next = l2;
 l2=l2->next;
 }
 p=p->next;
 }
 p->next = l1==nullptr ? l2 : l1;
 ListNode *newHead = dummyHead->next;
 delete(dummyHead);
 return newHead;
 }
};
```

/\*  
 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。

示例：

输入：head = 1->4->3->2->5->2,  $x = 3$

输出：1->2->2->4->3->5

```
*/

class Solution {
public:
 ListNode* partition(ListNode* head, int x) {
 ListNode *lowdummyhead = new ListNode(-1); // 小于x的结点的虚拟头结点
 ListNode *plow = lowdummyhead;
 ListNode *highdummyhead = new ListNode(-1); // 大于等于x的结点的虚拟头结点
 ListNode *phigh = highdummyhead;
 while(head) {
 ListNode *next = head->next; // 记录head->next 防止链表断裂
 head->next = nullptr;
 if(head->val < x) { // 小于x的值连接到lowdummyhead
 plow->next = head;
 plow = plow->next;
 } else {
 phigh->next = head; // 大于等于x的值链接到highdummyhead
 phigh = phigh->next;
 }
 head = next; // head 向前走一步
 }
 plow->next = highdummyhead->next;
 ListNode *newhead = lowdummyhead->next;
 delete lowdummyhead;
 delete highdummyhead;
 return newhead;
 }
};
```

```
/*
 * 给定一个带有头结点 head 的非空单链表，返回链表的中间结点。
 * 如果有两个中间结点，则返回第二个中间结点。
 */
class Solution {
public:
 ListNode* middleNode(ListNode* head) {
 if(!head) return head;
 ListNode *slow = head;
 ListNode *fast = head;
 while(fast->next && fast->next->next){
 slow = slow->next;
 fast = fast->next->next;
 }
 return fast->next==nullptr ? slow : slow->next; //奇偶个数分别处理
 }
};

/*
 * 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。
 * 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。
 */
class Solution {
public:
 ListNode* swapPairs(ListNode* head) {
 if(!head) return nullptr;
 ListNode *dummyhead = new ListNode(-1);
 dummyhead->next = head;
 ListNode *pre = dummyhead;
 ListNode *p = head;
 ListNode *after = head->next;
 while(p && p->next){
 pre->next = p->next;
 p->next = after->next;
 after->next = p;
 pre = p;
 p = p->next;
 if(p) after = p->next;
 }
 ListNode *newHead = dummyhead->next;
 delete dummyhead;
 return newHead;
 }
};
```



```
/*
*给你两个 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一位数字。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。
*复杂度分析
*时间复杂度： $O(\max(m, n))$ ，其中 m 与 n 分别为两个链表的长度。我们需要遍历每个链表。
*空间复杂度： $O(m + n)$ ，其中 m 与 n 分别为两个链表的长度。这是我们把链表内容放入栈中所用的空间。
*/
```

```
class Solution {
public:
 ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
 stack<int> s1,s2;
 while(l1){ // 入栈
 s1.push(l1->val);
 l1=l1->next;
 }
 while(l2){
 s2.push(l2->val);
 l2=l2->next;
 }
 int carry=0;
 ListNode *ans=NULLPTR;
 while(!s1.empty() or !s2.empty() or carry > 0){
 int x = s1.empty() ? 0 : s1.top();
 int y = s2.empty() ? 0 : s2.top();
 if(!s1.empty())s1.pop();
 if(!s2.empty())s2.pop();
 int curr = x + y +carry;
 carry = curr/10;
 curr%=10;
 ListNode *temp = new ListNode(curr);
 temp->next = ans;
 ans = temp;
 }
 return ans;
 }
};
```

```
class Solution {
public:
 ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
 ListNode *p1 = l1;
 ListNode *p2 = l2;
 ListNode *dummyhead = new ListNode(-1);
 ListNode *cur = dummyhead;
 int carry = 0;
 while(p1 || p2){ // 不论长短 一起遍历
```

```

 int x = p1 != nullptr ? p1->val : 0; // 若p1到尾部, 按照加法规则, 从低位对齐, 高位视为0
 int y = p2 != nullptr ? p2->val : 0;
 int sum = x + y + carry;
 carry = sum / 10;
 cur->next = new ListNode(sum%10); // sum % 10 十进制, 满十进位, 剩余保留
 cur = cur->next; // 结果链表指针移到当前尾部
 if(p1) p1 = p1->next; // 如果p1不空, p1前进一步。防止空指针报错, 因为本循环无视长度
 if(p2) p2 = p2->next;
 }
 if(carry > 0){ // 如果最高位满十, 则进位, 逻辑同其他
 cur->next = new ListNode(carry);
 }
 ListNode *newhead = dummyhead->next;
 delete dummyhead;
 return newhead;
}

};

/* 删除链表中最小的结点 */
ListNode *del_min(ListNode *head){
 ListNode *pre = head, *p = pre->next;
 ListNode *minpre = pre, *min = p;
 while(p){
 if(p->data < min->data){
 min = p;
 minpre = pre;
 }
 p = p->next;
 pre = pre->next;
 }
 minpre->next = min->next;
 delete(min);
 return head;
}

```

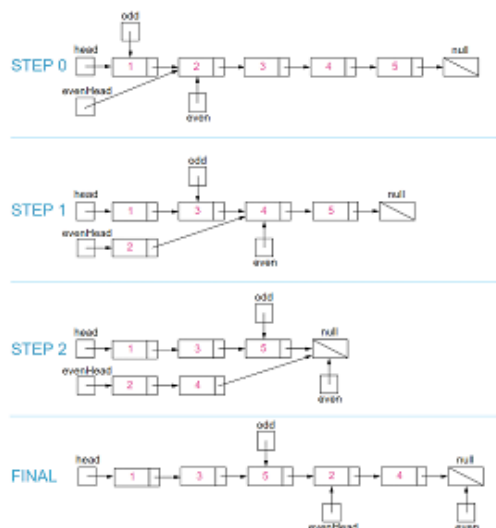
```

/* sort链表使其递增 */
/* 插入排序思想 */
void sort(LinkList &L){
 LNode *p = L->next,*pre;
 LNode *r = p->next; // 保证链表不断链
 p->next = null; // 构建有序链表（只有第一个结点）
 p = r; // p来到无需部分的头部
 while(p){
 r = p->next; // 保存p的后继结点，防止链表断链
 pre = L;
 while(pre->next && pre->next->data < p->data){
 pre = pre->next; // 找到p点的插入位置
 }
 p->next = pre->next; // 尾插
 pre->next = p;
 p = r;
 }
}

/* 每次找到链表中最小的结点，输出val，并将其free()，直到只剩head，最后将head free() */
void Min_del(LinkList &head){
 while(head->next){
 pre = head;
 p = pre->next;
 while(p->next){
 if(p->next->val < pre->next->val)
 pre = p; //记录最小值的前驱
 p = p->next;
 }
 printf(pre->next->val);
 u = pre->next;
 pre->next = u->next;
 free(u);
 }//while
 free(head);
}

/*
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，而不是节点的值奇偶性。
*/

```



```
class Solution {
public:
 ListNode* oddEvenList(ListNode* head) {
 if(!head) return nullptr;
 // odd是奇链表的当前节点，先初始化为head（初始化为奇链表头）
 ListNode *odd = head;
 // even是偶链表的当前节点，初始化为第二个节点也就是head.next
 ListNode *even = head->next;
 // evenHead是偶链表的头节点，初始化为链表第二个节点（初始化为奇链表头的下一个节点）
 ListNode *evenhead = even;
 while(even && even->next){ //涉及到链表长度为奇或偶，用双指针，条件好像都是这个
 odd->next = even->next;
 odd = odd->next;
 even->next = odd->next;
 even = even->next;
 }
 odd->next = evenhead; // 奇(拼接)偶
 return head;
 }
};
```

```
/* 删除 有序 链表中的重复项 */
```

```
void Del_Same(LinkList &L){
 LNode* p = L->next, *q;
 if(p==null) return;
 while(p->next){
 q = p->next;
 if(p->data == q->data){
 p->next = q->next;
 free(q);
 }else{
 p = p->next;
 }
 }
}
```

```
/* 请判断一个链表是否为回文链表。(栈) */
```

```
class Solution {
public:
 bool isPalindrome(ListNode* head) {
 stack<int> st;
 ListNode *cur = head;
 int length = 1;
 while(cur){
 st.push(cur->val);
 cur = cur->next;
 length++;
 }
 int cnt = length/2;
 cur = head;
 while(cnt--){
 if(cur->val != st.top()) return false;
 st.pop();
 cur = cur->next;
 }
 return true;
 }
};
```

```
/* 顺序表法 */
class Solution {
public:
 bool isPalindrome(ListNode* head) {
 vector<int> res;
 while(head) {
 res.push_back(head->val);
 head = head->next;
 }
 int i = 0; int j = res.size() - 1;
 while(i<j){ // 啊啊啊啊 边界值真的是佛了啊
 if(res[i++]!=res[j--]) return false;
 }
 return true;
 }
};

/* 双指针, 反转链表 */
public:
 bool isPalindrome(ListNode *head){
 ListNode *cur = head;
 ListNode *pre = nullptr;
 ListNode *fast = head;
 while(fast && fast->next){
 fast = fast->next->next; //位置不可以变, 因为链表在后面断了
 ListNode *temp = cur->next;
 cur->next = pre;
 pre = cur;
 cur = temp;
 }
 if(fast) cur = cur->next;
 while(pre){
 if(cur->val != pre->val) return false;
 cur = cur->next;
 pre = pre->next;
 }
 return true;
 }
};
```

```
/* 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。 */
class Solution {
public:
 ListNode* rotateRight(ListNode* head, int k) {
 if(!head || !head->next || k==0) return head;
 ListNode *phead = head;
 int length = 1;
 while(phead->next){length++;phead = phead->next;}
 int move = k % length;
 if(move == 0) return head;
 ListNode *cur = head;
 for(int i=0;i<length-move-1;i++) cur = cur->next;
 ListNode *tmp = cur->next;
 cur->next = nullptr;
 phead->next = head;
 return tmp;
 }
};
```