

Lists Structure

- struct Lists

```
// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

index

- [remove-duplicates-from-sorted-list](#)
- [remove-duplicates-from-sorted-list-ii](#)
- [reverse-linked-list](#)

remove-duplicates-from-sorted-list

linkage: [leetcode](#)

- 给定一个**排序链表**，删除所有重复的元素，使得每个元素只出现一次
- 迭代版本（直接法）

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head)
    {
        ListNode* _head = head;
        if(head == nullptr)
        {
            return _head;
        }
        // 注意1：判断当前head->next是否为空，并非head为空
        while(head->next != nullptr)
        {
            if(head->next->val == head->val)
            {
                //注意2：删除操作
                head->next = head->next->next;
            }
            else
            {
                head = head->next;
            }
        }
        return _head;
    }
};
```

```
    }
};
```

- 递归版本一
- 有重先去重，头结点定位到重复元素最后一个

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if(head == nullptr || head->next == nullptr)
        {
            return head;
        }
        if(head->val == head->next->val)
        {
            // 如果有重复，头结点定位到重复元素的最后一个，相当于去重
            head = deleteDuplicates(head->next);
        }
        else
        {
            // 无重复后连接到下一个节点，再考虑下个节点
            head->next = deleteDuplicates(head->next);
        }
        return head;
    }
};
```

- 递归版本二
- 删除头节点后面挂接的链表中的重复元素

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if(head == nullptr || head->next == nullptr)
        {
            return head;
        }
        // 删除头节点后面挂接的链表中的重复元素
        head->next = deleteDuplicates(head->next);
        if(head->val == head->next->val)
        {
            head = head->next;
        }
        return head;
    }
};
```

- 快慢指针

```

class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head)
    {
        ListNode* slow = head;
        ListNode* fast = head;
        if(head == nullptr)
        {
            return slow;
        }
        // 注意while判断的条件
        while(fast != nullptr)
        {
            if(slow->val != fast->val)
            {
                // 若不相等,则将慢指针slow指向快指针fast的地址
                slow->next = fast;
                slow = slow->next;
            }
            // fast指向下一个
            fast = fast->next;
        }
        // 最后slow指向空
        slow->next = nullptr;
        return head;
    }
};

```

remove-duplicates-from-sorted-list-ii

linkage: [leetcode](#)

- 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
- 方式一: 迭代方法(注意元素去重以及边界条件处理)

```

class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head)
    {
        // 注意一: 哨兵节点: 指向链表, 用于最后返回结果
        ListNode* tmp = new ListNode();
        tmp->next = head;
        ListNode* fast = head;
        ListNode* slow = tmp;
        if(head == NULL)
        {
            return fast;
        }
        // 注意二: 一定要包含fast!= nullptr, 需要考虑边界条件

```

```

while(fast != nullptr&&fast->next != NULL)
{
    // 注意三：找重复节点
    if(fast->val == fast->next->val)
    {
        int tmp_node = fast->val;
        // 注意四：一定要包含fast->next!= nullptr, 需要考虑边界条件
        while(fast->next != nullptr &&fast->next->val
==tmp_node)
        {
            fast = fast->next;
        }
        // 注意五：循环后返回重复元素的末尾
        fast = fast->next;
        slow->next = fast;
    }
    else
    {
        fast = fast->next;
        slow = slow->next;
    }
}
return tmp->next;
}
};

```

- 方式二：Recursion

```

class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head)
    {
        if(head == nullptr || head->next == nullptr)
        {
            return head;
        }
        // 注意：下一个元素
        ListNode * next = head->next;
        if(next->val == head->val)
        {
            while(next != nullptr && next->val == head->val)
            {
                next = next -> next;
            }
            // 因为要将重复的都删了，所以直接返回递归函数
            return deleteDuplicates(next);
        }
        else
        {
            //如果不重复就将当前节点指向递归函数
            head->next = deleteDuplicates(head->next);
            return head;
        }
    }
};

```

```
    }  
  }  
};
```

reverse-linked-list

linkage: [leetcode](#)

- 反转一个单链表
- 定义两个指针：prepre 和 curcurcur ；prepre 在前 curcurcur 在后
- 每次让 prepre 的 nextnextnext 指向 curcurcur ，实现一次局部反转
- 局部反转完成之后， prepre 和 curcurcur 同时往前移动一个位置
- 循环上述过程，直至 prepre 到达链表尾部

开始



- 思路一：迭代双指针方式

```
class Solution {  
public:  
    ListNode* reverseList(ListNode* head)  
    {  
        if(head == nullptr)  
        {  
            return head;  
        }  
        // 注意：此处pre不能等于new ListNode()  
        ListNode* pre = nullptr;  
        ListNode* cur = head;  
        while(cur != nullptr)  
        {  
            // 注意：此处用临时变量指向cur->next  
            ListNode* tmp_next = cur->next;
```

```
        cur->next = pre;
        // 双指针移动
        pre = cur;
        cur = tmp_next;
    }
    return pre;
};
```

- 思路二：递归方式
- 使用递归函数，一直递归到链表的最后一个结点，该结点就是反转后的头结点，记作ret
- 每次函数在返回的过程中，让当前结点的下一个结点的next指针指向当前节点
- 同时让当前结点的next指针指向NULL，从而实现从链表尾部开始的局部反转
- 当递归函数全部出栈后，链表反转完成

开始

