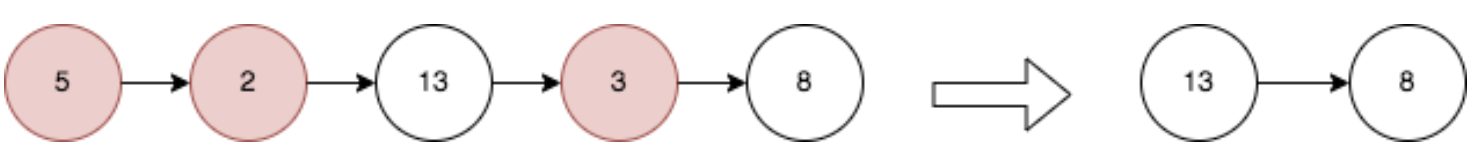
# 题目

给你一个链表的头节点 head 。

移除每个右侧有一个更大数值的节点。

返回修改后链表的头节点 head 。

示例 1：



输入：head = [5,2,13,3,8]

输出：[13,8]

解释：需要移除的节点是 5 ，2 和 3 。

- 节点 13 在节点 5 右侧。

- 节点 13 在节点 2 右侧。

- 节点 8 在节点 3 右侧。

示例 2：

输入：head = [1,1,1,1]

输出：[1,1,1,1]

解释：每个节点的值都是 1 ，所以没有需要移除的节点。

提示：

给定列表中的节点数目在范围 [1, 105] 内

1 <= Node.val <= 105

# 分析

## 方法一：递归

由题意可知，节点对它右侧的所有节点都没有影响，因此对于某一节点，我们可以对它的右侧节点递归地进行移除操作：

1、该节点为空，那么递归函数返回空指针。

2、该节点不为空，那么先对它的右侧节点进行移除操作，得到一个新的子链表，如果子链表的表头节点值大于该节点的值，那么移除该节点，否则将该节点作为子链表的表头节点，最后返回该子链表。

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* removeNodes(ListNode\* head) {

if (head == nullptr) return nullptr;

head->next = removeNodes(head->next);

if (head->next && head->val < head->next->val) {

return head->next;

} else {

return head;

}

}

};

复杂度分析：

时间复杂度：O(n)，其中n是链表节点个数。

空间复杂度：O(n)。

## 方法二：栈

类似于方法一，我们可以用栈来模拟递归，具体为：

1、将所有链表节点按从左到右的顺序压入栈中，同时新链表初始为空。

2、不断地从栈中弹出节点，如果节点的值大于等于新链表的表头节点值，那么将该节点插入新链表的表头，否则移除该节点。

最后返回该链表。

代码：

class Solution {

public:

ListNode\* removeNodes(ListNode\* head) {

stack<ListNode \*> st;

for (; head != nullptr; head = head->next) {

st.push(head);

}

for(; !st.empty(); st.pop()) {

if (head == nullptr || st.top()->val >= head->val) {

st.top()->next = head;

head = st.top();

}

}

return head;

}

};

复杂度分析：

时间复杂度：O(n)，其中n是链表节点个数。

空间复杂度：O(n)。

## 方法三：反转链表