

题目一：

[题目描述](#) | [笔记](#) × | [题解](#) | [提交记录](#)

817. 链表组件

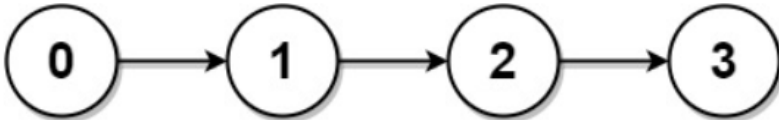
已解答 ✓

中等 相关标签 相关企业 Aa

给定链表头结点 `head`，该链表上的每个结点都有一个 **唯一的整型值**。同时给定列表 `nums`，该列表是上述链表中整型值的一个子集。

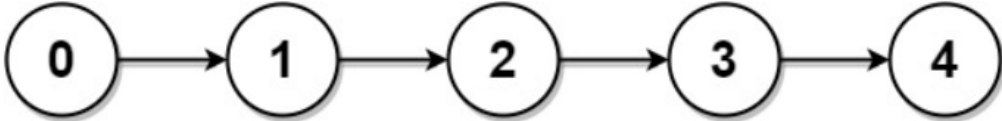
返回列表 `nums` 中组件的个数，这里对组件的定义为：链表中一段最长连续结点的值（该值必须在列表 `nums` 中）构成的集合。

示例 1：



输入： `head = [0,1,2,3]`, `nums = [0,1,3]`
输出： 2
解释： 链表中, 0 和 1 是相连接的, 且 `nums` 中不包含 2, 所以 `[0, 1]` 是 `nums` 的一个组件, 同理 `[3]` 也是一个组件, 故返回 2。

示例 2：



输入： `head = [0,1,2,3,4]`, `nums = [0,3,1,4]`
输出： 2

198 562 ☆ ↗ ?

解题思路一：

利用数组下标，由于此题中，所有数据都是连续的，那么我们可以遍历链表获取数据的长度，根据长度设置一个dp数组，初始化结果为0，对于输入的数据nums，由于它是链表数据的子集，那么一定有dp[nums[i]]，对于这种出现在nums的数据，我们把它置为一，那么只要当前数据的dp为1且它后续无1或者无数据，当前组件数目++

```
class Solution {
public:
    int numComponents(ListNode* head, vector<int>& nums) {
        int len = 0;
        ListNode* t = head;
        while (t != nullptr)
```

```

    {
        len++;
        t = t->next;
    }
    vector<int>dp(len, 0);
    int count = 0;
    //待处理组件集合中的数据置为1
    for (auto i : nums)
        dp[i] = 1;
    t = head;
    while (t != nullptr)
    {
        //后续有1说明这个组件长度是大于1的，直到后续为0或者空这个组件才结束
        if (dp[t->val] == 1 && (t->next == nullptr || dp[t->next->val] ==
0))
            count++;
        t = t->next;
    }
    return count;
}
};

```

解题思路二：

同时判断当前元素与当前元素的下一个元素。如果当前元素在nums中，且下一个元素不在Nums中或者为空时，当前组件数目+1,由于链表中的每个元素都要遍历数组进行判断，所以时间复杂度是 $O(n^2)$ ，在数据量大的时候容易超时，但是这份代码不局限于链表中的数据是连续的整型，链表的数据可以是无序的不等值

```

class Solution {
public:
    int numComponents(ListNode* head, vector<int>& nums) {
        int flag=0;
        while(head)
        {
            //当前元素在，下一个元素不在
            if(show(head,nums)&&!show(head->next,nums))
            {
                flag+=1;
            }
            head=head->next;
        }
        return flag;
    }
    bool show(ListNode* s,vector<int>& nums)
    {
        //为空说明没有后续元素，那么组件长度已经不能再长了
        if(s==nullptr)
            return false;
        for(int i=0;i<nums.size();i++)
        {
            if(nums[i]==s->val)
            {
                return true;
            }
        }
    }
};

```

```

    }
    return false;
}
};

```

解题思路三：

利用哈希表记忆化搜索提高搜索效率，设置一个标志位incompent,初始位false，如果当前链表的值在nums中且不在组件中，count++,incompent变为ture,如果下一个元素还在nums中，由于这个时候它的组件标志位是ture，所以计数器不用修改，直到下一个元素不在组件中或者链表遍历结束

```

class Solution {
public:
    int numComponents(ListNode* head, vector<int>& nums) {
        // 使用unordered_set提高查找效率
        unordered_set<int> numSet(nums.begin(), nums.end());
        int count = 0;
        bool inComponent = false;

        // 遍历链表
        while (head != nullptr) {
            if (numSet.find(head->val) != numSet.end()) {
                // 如果当前值在nums中，且不在组件中，则计数加1
                if (!inComponent) {
                    count++;
                    inComponent = true;
                }
            }
            else {
                // 如果当前值不在nums中，重置组件状态
                inComponent = false;
            }
            head = head->next;
        }

        return count;
    }
};

```

题目二：

114. 二叉树展开为链表

已解答 ✓

中等

🔖 相关标签

🏢 相关企业

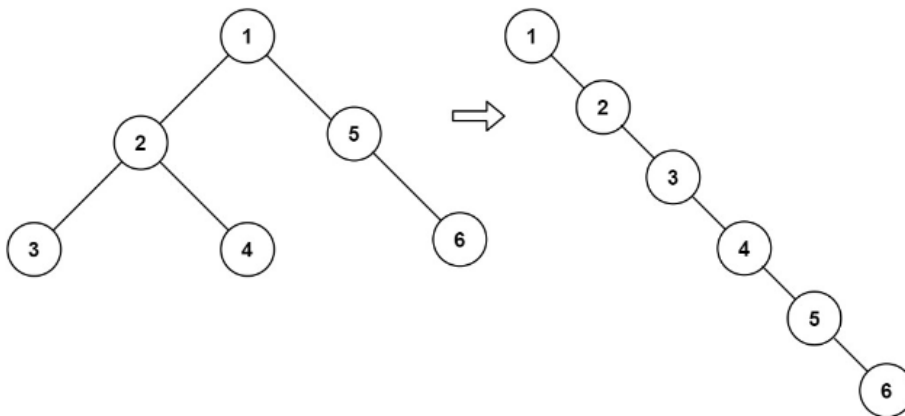
💡 提示

Aa

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。
- 展开后的单链表应该与二叉树 **先序遍历** 顺序相同。

示例 1:



输入: `root = [1,2,5,3,4,null,6]`

输出: `[1,null,2,null,3,null,4,null,5,null,6]`

示例 2:

输入: `root = []`

解题思路一：

题目的结果是展开后与前序遍历结果相同，那么我们先前序遍历，将前序遍历的结果存入到向量中，再根据向量的数据构建一颗单链表的二叉树即可

```

class Solution {
public:
    //前序遍历存入结果
    void preorderTraversal(TreeNode* root, vector<TreeNode*>& nodes) {
        if (!root) return;
        nodes.push_back(root);
        preorderTraversal(root->left, nodes);
        preorderTraversal(root->right, nodes);
    }

    void flatten(TreeNode* root) {
        if (!root) return;

        vector<TreeNode*> nodes;
        preorderTraversal(root, nodes);
    }
}
    
```

```

//遍历向量构建链表
for (int i = 0; i < nodes.size() - 1; ++i) {
    nodes[i]->left = nullptr;//左子树全部为空
    nodes[i]->right = nodes[i + 1];
}
//对最后的尾指针进行处理
if (!nodes.empty()) {
    nodes.back()->left = nullptr;
    nodes.back()->right = nullptr;
}
};

```

解题思路二：

由于需要前序遍历的结果作为新链表，且新链表的构成就是原树的左子为空，右子树为前序遍历结果，那么我们只需在遍历节点时，如果左子树不为空，找到左子树的最右子树，将当前左子树的右子树移到最右子树，成为最右子树的右子树，然后把当前子树的右子树变成左子树，左子树赋空，右子树是原本左子树的值，变更节点指向右子树，即current=current->right,这颗树是原来的左子树，对这颗树进行处理，直到所有树的处理完毕，最后的结果就是我们需要的值

```

class Solution {
public:
    void flatten(TreeNode* root) {
        if (!root) return;

        TreeNode* current = root;
        while (current) {
            if (current->left) {
                // 找到左子树中的最右节点
                TreeNode* rightmost = current->left;
                while (rightmost->right) {
                    rightmost = rightmost->right;
                }
                // 将当前节点的右子树接到左子树的最右节点的右孩子
                rightmost->right = current->right;
                // 将左子树移动到右子树位置
                current->right = current->left;
                current->left = nullptr;
            }
            // 继续处理右子树
            current = current->right;
        }
    }
};

```

解题思路三：

使用辅助栈，一边展开前序一边替换节点，在这里我们依然是替换左右子树，不同的是，右子树我们不再拼接到最右子树，而是入栈，对于每个节点，如果它的前序节点不为空，前序节点的左值赋值为空，右值为当前节点。由于每次入栈的时候都是右子树先入栈，因此只要左子树不为空，所有的右子树都无法出栈，这就保证了遍历的顺序一定是前序

```

//一边进行前序遍历一边展开

```

```
class Solution {
public:
    void flatten(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        auto stk = stack<TreeNode*>();
        //根节点入栈
        stk.push(root);
        //前序节点初始化
        TreeNode* prev = nullptr;
        while (!stk.empty()) {
            TreeNode* curr = stk.top();
            //弹出节点
            stk.pop();
            if (prev != nullptr) {
                //构建链表，左值为空，右值为前序遍历值
                prev->left = nullptr;
                prev->right = curr;
            }
            TreeNode* left = curr->left, * right = curr->right;
            if (right != nullptr) {
                stk.push(right);
            }
            if (left != nullptr) {
                stk.push(left);
            }
            //前序节点变成当前节点
            prev = curr;
        }
    }
};
```