目录

数据结构基础

一、二叉树

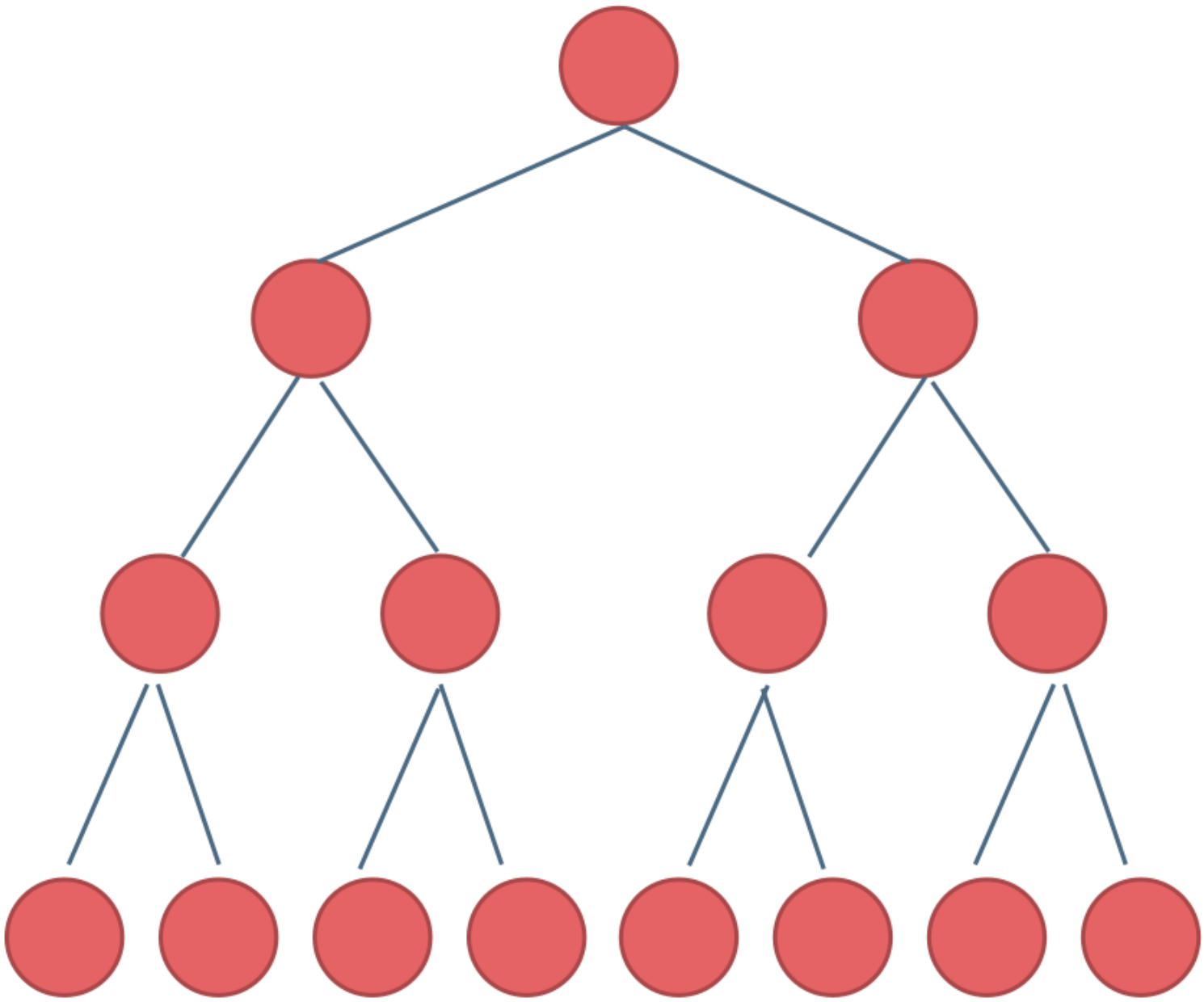
1.1 二叉树的种类

在我们解题过程中二叉树有两种主要的形式：满二叉树和完全二叉树。

1.1.1满二叉树

满二叉树：如果一棵二叉树只有度为0的结点和度为2的结点，并且度为0的结点在同一层上，则这棵二叉树为满二叉树。

如图所示：



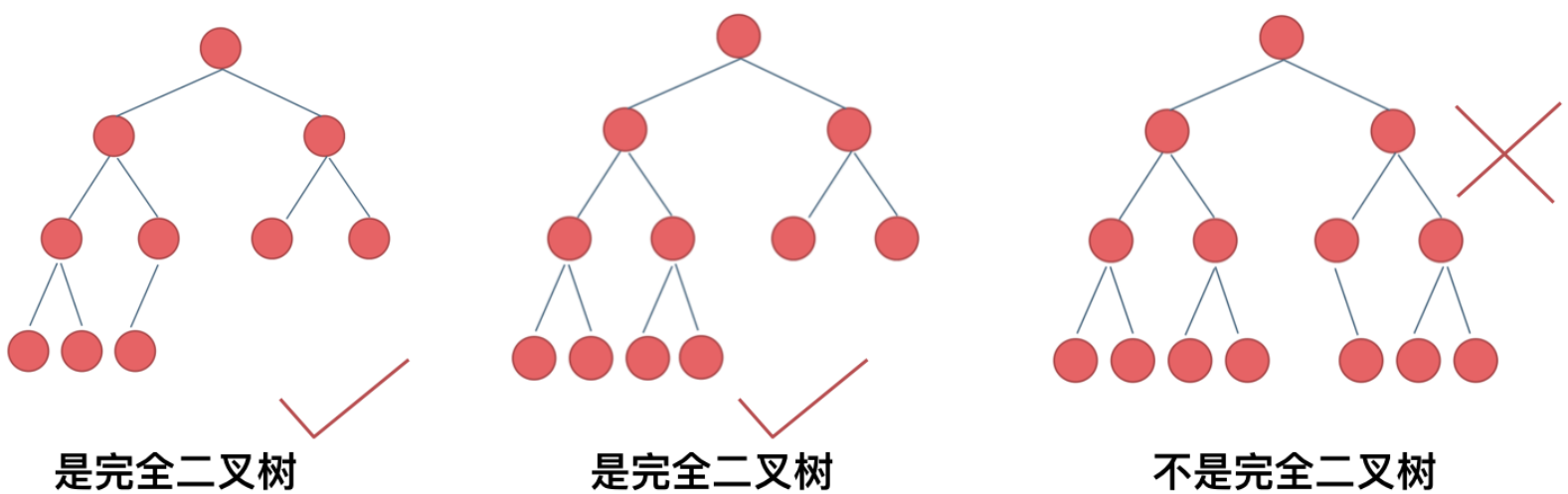
这棵二叉树为满二叉树，也可以说深度为k，有2^k-1个节点的二叉树。

1.1.2 完全二叉树

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层（h从1开始），则该层包含 1~ 2^(h-1) 个节点。说白了就是最下面**从左到右连续**，不存在左孩子为空的情况。

大家要自己看完全二叉树的定义，很多同学对完全二叉树其实不是真正的懂了。

我来举一个典型的例子如题：



相信不少同学最后一个二叉树是不是完全二叉树都中招了。

之前我们刚刚讲过优先级队列其实是一个堆，堆就是一棵完全二叉树，同时保证父子节点的顺序关系。

1.1.3 二叉搜索树

前面介绍的树，都没有数值的，而二叉搜索树是有数值的了，**二叉搜索树是一个有序树。**

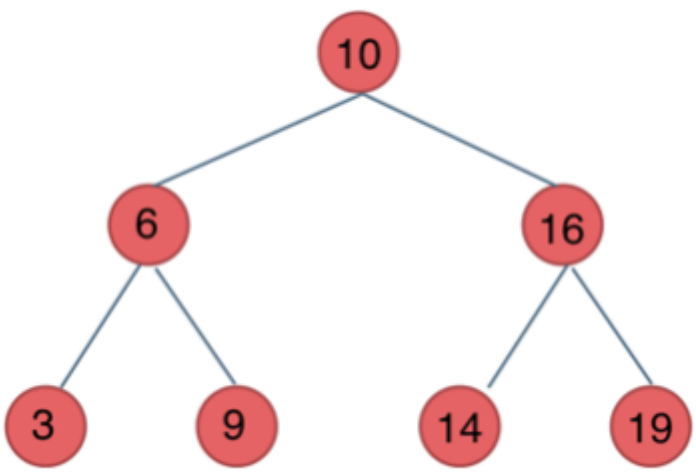
- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉排序树

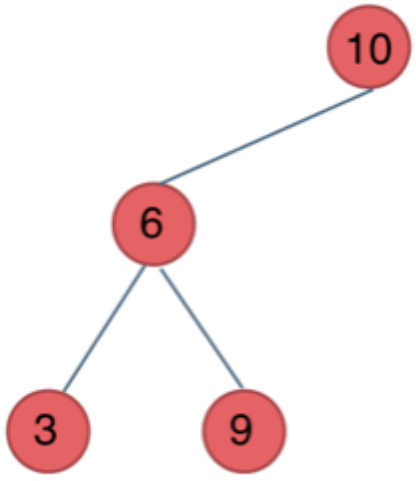
我的理解就是：

中

小　　大

下面这两棵树都是搜索树

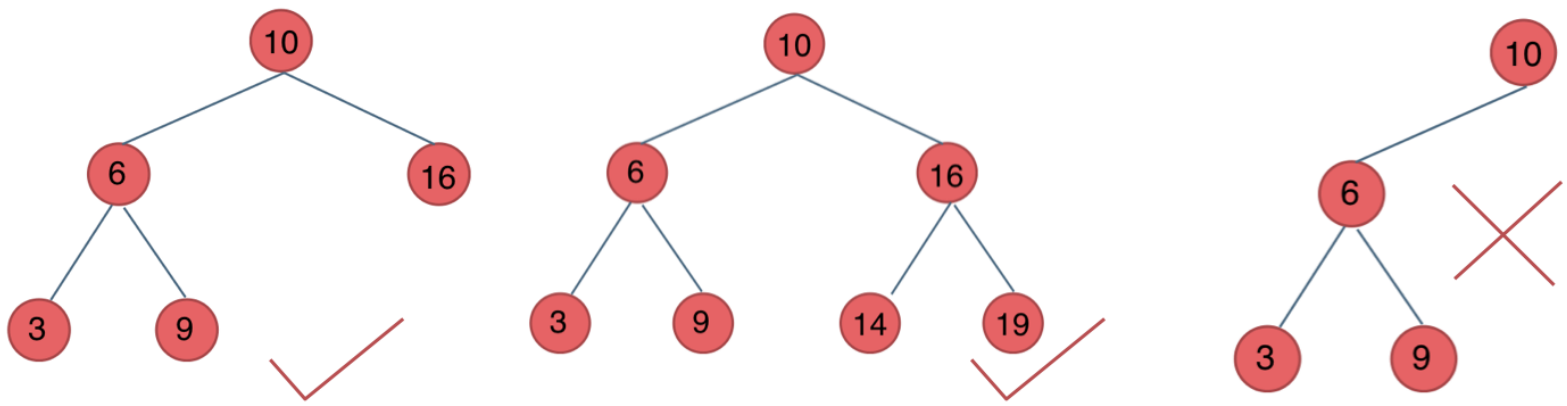




1.1.4 平衡二叉搜索树

平衡二叉搜索树：又被称为AVL（Adelson-Velsky and Landis）树，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

如图：



最后一棵 不是平衡二叉树，因为它的左右两个子树的高度差的绝对值超过了1。

C++中map、set、multimap，multiset的底层实现都是平衡二叉搜索树，所以map、set的增删操作时间时间复杂度是log(n)，注意我这里没有说unordered_map、unordered_set，unordered_map、unordered_set底层实现是哈希表。

所以大家使用自己熟悉的编程语言写算法，一定要知道常用的容器底层都是如何实现的，最基本的就是map、set等等，否则自己写的代码，自己对其性能分析都分析不清楚！

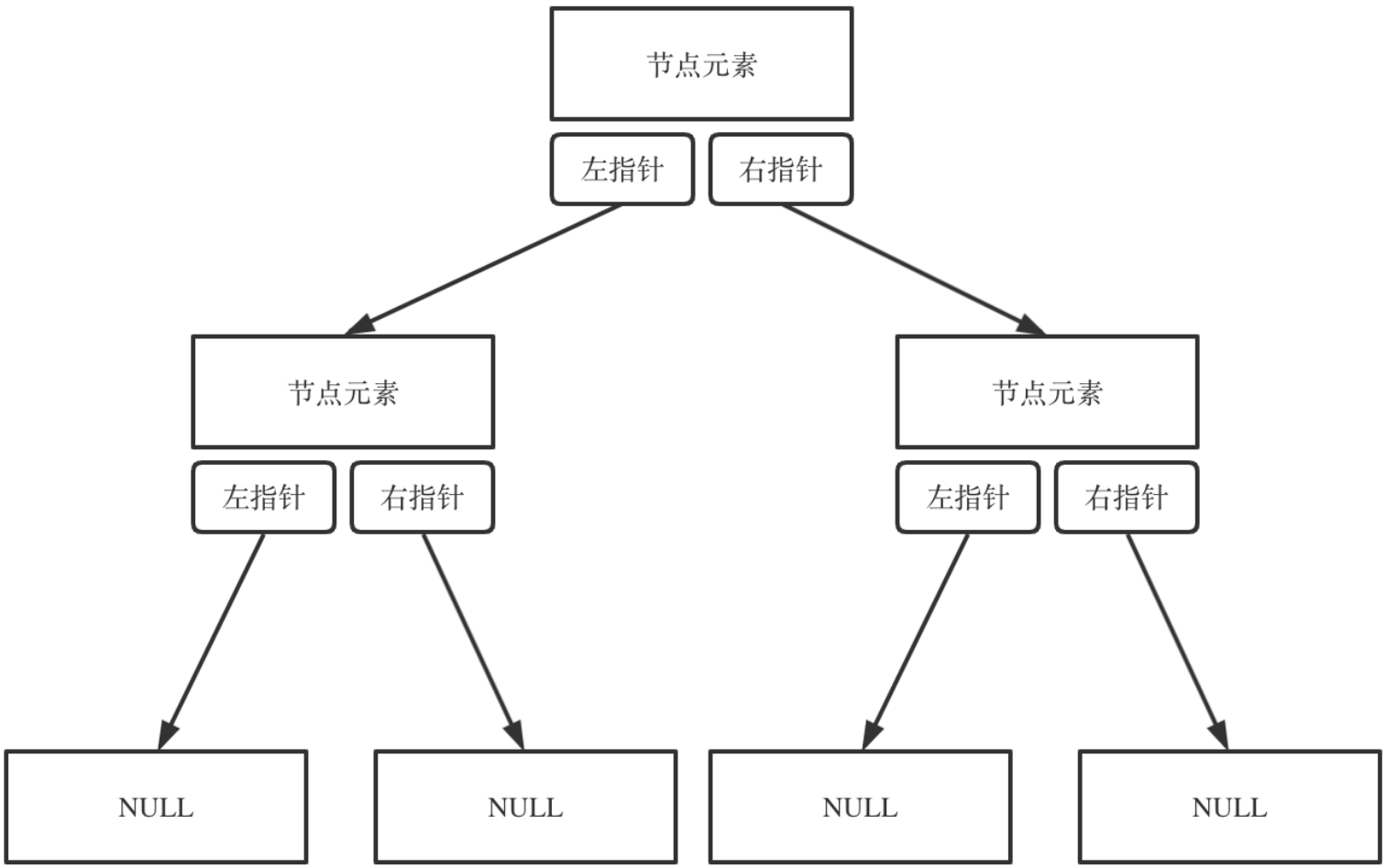
1.2 二叉树的存储方式

二叉树可以链式存储，也可以顺序存储。

那么链式存储方式就用指针， 顺序存储的方式就是用数组。

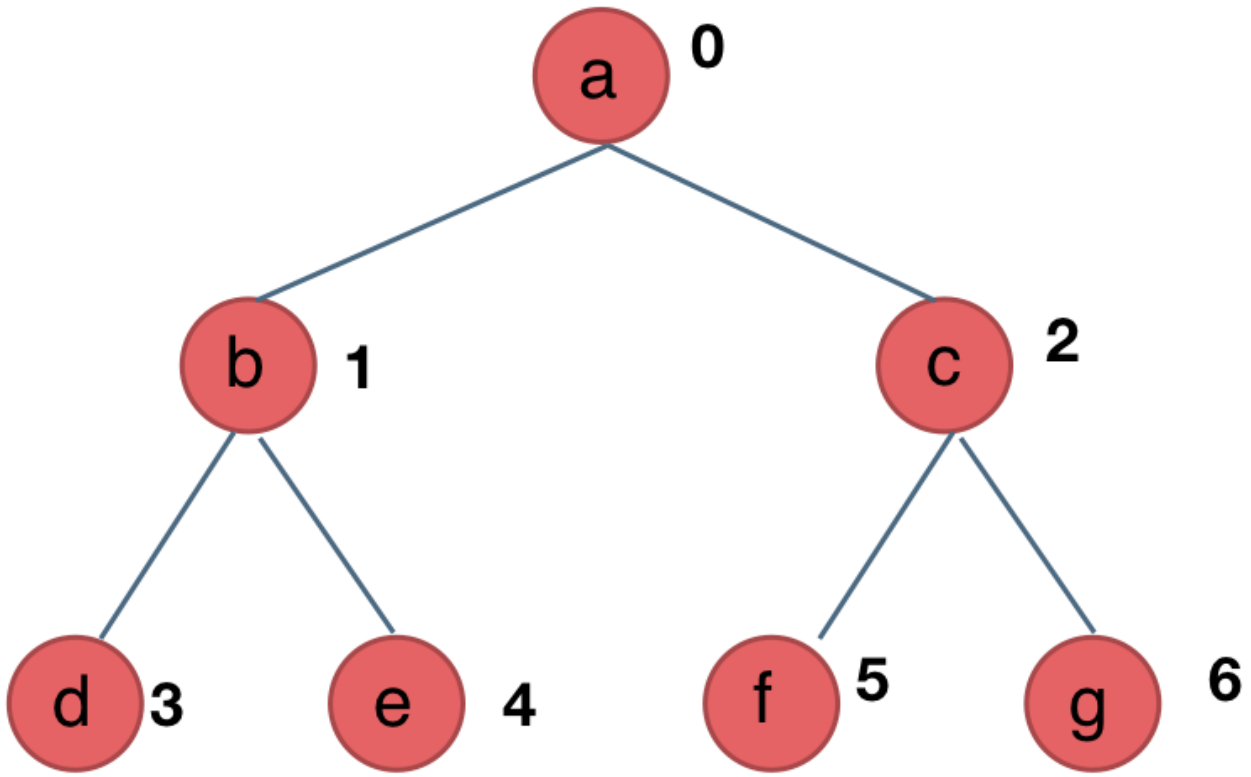
顾名思义就是顺序存储的元素在内存是连续分布的，而链式存储则是通过指针把分布在各个地址的节点串联一起。

链式存储如图：



链式存储是大家很熟悉的一种方式，那么我们来看看如何顺序存储呢？

其实就是用数组来存储二叉树，顺序存储的方式如图：



数组中的树：	a	b	c	d	e	f	g
下标：	0	1	2	3	4	5	6

用数组来存储二叉树如何遍历的呢？

如果父节点的数组下标是 i，那么它的左孩子就是 i * 2 + 1，右孩子就是 i * 2 + 2。

但是用链式表示的二叉树，更有利于我们理解，所以一般我们都是用链式存储二叉树。

所以大家要了解，用数组依然可以表示二叉树。

1.3 二叉树的遍历方式

关于二叉树的遍历方式，要知道二叉树遍历的基本方式都有哪些。

一些同学用做了很多二叉树的题目了，可能知道前中后序遍历，可能知道层序遍历，但是却没有框架。

我这里把二叉树的几种遍历方式列出来，大家就可以一一串起来了。

二叉树主要有两种遍历方式：

1. **深度优先遍历**：先往深走，遇到叶子节点再往回走。

2. **广度优先遍历**：一层一层的去遍历。

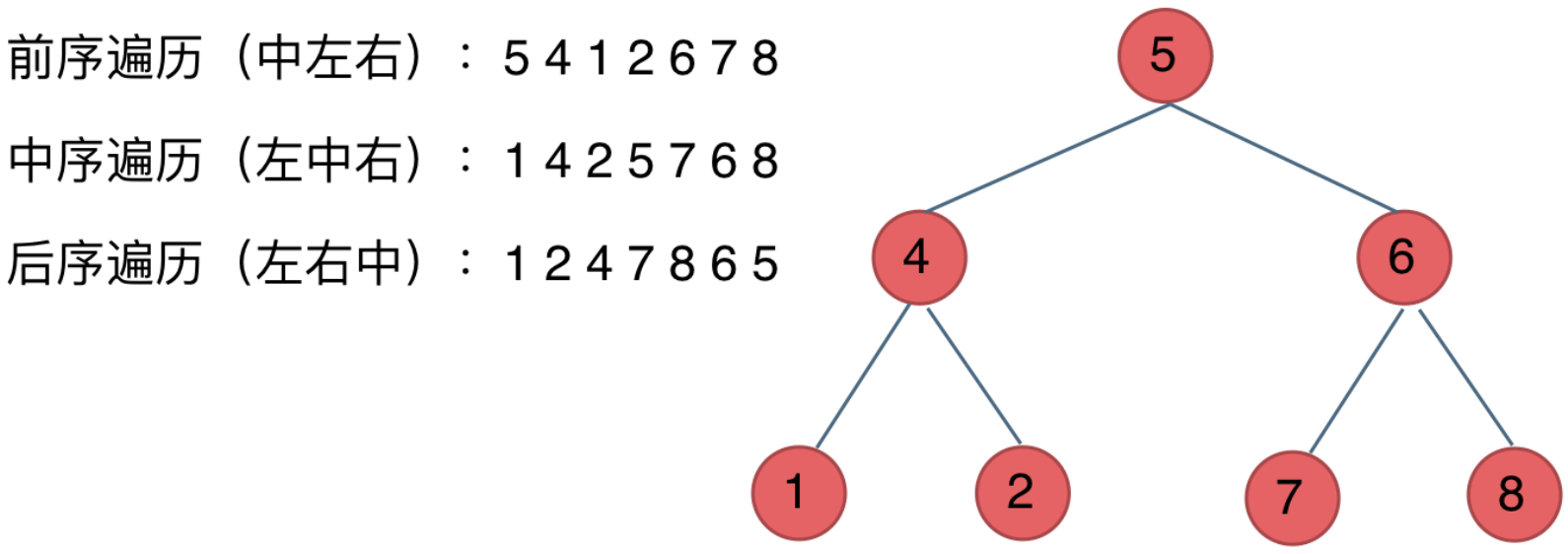
这两种遍历是图论中最基本的两种遍历方式，

那么从深度优先遍历和广度优先遍历进一步拓展，才有如下遍历方式：

- 深度优先遍历(见下图)
  - 前序遍历（递归法，迭代法）
  - 中序遍历（递归法，迭代法）
  - 后序遍历（递归法，迭代法）
- 广度优先遍历
  - 层次遍历（迭代法）

在深度优先遍历中：有三个顺序，前中后序遍历， 有同学总分不清这三个顺序，经常搞混，我这里有一个技巧。就是去看”中“在哪个位置就是什么序遍历。

- 前序遍历：中左右
- 中序遍历：左中右
- 后序遍历：左右中



最后再说一说二叉树中深度优先和广度优先遍历实现方式，我们做二叉树相关题目，经常会使用递归的方式来实现深度优先遍历，也就是实现前中后序遍历，使用递归是比较方便的。

之前我们讲栈与队列的时候，就说过栈其实就是递归的一种实现结构，也就说前中后序遍历的逻辑其实都是可以借助栈使用递归的方式来实现的。

而广度优先遍历的实现一般使用队列来实现，这也是队列先进先出的特点所决定的，因为需要先进先出的结构，才能一层一层的来遍历二叉树。

这里其实我们又了解了栈与队列的一个应用场景了。

1.4 二叉树的定义

刚刚我们说过了二叉树有两种存储方式顺序存储，和链式存储，顺序存储就是用数组来存，这个定义没啥可说的，我们来看看链式存储的二叉树节点的定义方式。**在现场面试的时候 面试官可能要求手写代码，所以数据结构的定义以及简单逻辑的代码一定要锻炼白纸写出来。**

C++代码如下：

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

大家会发现二叉树的定义 和链表是差不多的，相对于链表 ，二叉树的节点里多了一个指针， 有两个指针，指向左右孩子。

这里要提醒大家要注意二叉树节点定义的书写方式。

简单难度{8/28}

20. 有效的括号

给定一个只包括 `'('`，`')'`，`'{'`，`'}'`，`'['`，`']'` 的字符串 `s` ，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例 1：

```
输入：s = "()"
输出：true
```

示例 2：

```
输入：s = "()[]{}"
输出：true
```

示例 3：

```
输入：s = "(]"
输出：false
```

提示：

- `1 <= s.length <= 104`
- `s` 仅由括号 `'()[]{}'` 组成

解答

1.

```
bool checkBracket(const string s) {
    stack <char> stackRuan;
    for (char temp : s) {
        if (temp == '[' || temp == '{' || temp == '(') {      //如果是左边的括号，就入栈。
            stackRuan.push(temp);
        }
        else if (temp == '}' || temp == ']' || temp == ')')
        {
            if ((temp == '}' && stackRuan.top() != '{') || (temp == ']' && stackRuan.top() != '[') || (temp == ')' && stackRuan.top() != '(')) {    //判断括号和栈顶的括号是否匹配，如果不匹配，则返回错误
                return false;
            }
            stackRuan.pop();                                    //如果匹配，则弹出栈顶元素
        }
    }
      return stackRuan.empty();                                 //现在栈是空的了
};
```

栈，是先入先出的结构，底层基于vector、deque、list，就是数组，队列，链表，默认情况下是deque

> 主要特性

- **LIFO 数据结构**：`std::stack` 是一种先进后出（LIFO，Last In First Out）的数据结构，这意味着最后插入的元素将是第一个被移除的元素。
- **容器适配器**：`std::stack` 是一种容器适配器，它封装了其他序列容器，提供了一组特定的成员函数来管理堆栈中的元素。

> 主要成员函数

以下是 `std::stack` 的一些主要成员函数及其功能：

- **构造函数**：用于创建 `std::stack` 对象。
- **empty()**：检查堆栈是否为空。　空的话返回true
- **size()**：返回堆栈中元素的数量。
- **top()**：返回堆栈顶部元素的引用。

- **push(const T& val)**：将元素 `val` 压入堆栈。
- **push(T&& val)**：将元素 `val` 移动到堆栈（C++11 起）。
- **pop()**：移除堆栈顶部的元素。
- **emplace(Args&&... args)**：在堆栈顶部创建一个新元素（C++11 起）。

> 底层容器

默认情况下，`std::stack` 使用 `std::deque` 作为底层容器。可以在声明 `std::stack` 时指定其他序列容器（如 `std::vector` 或 `std::list`）作为底层容器。

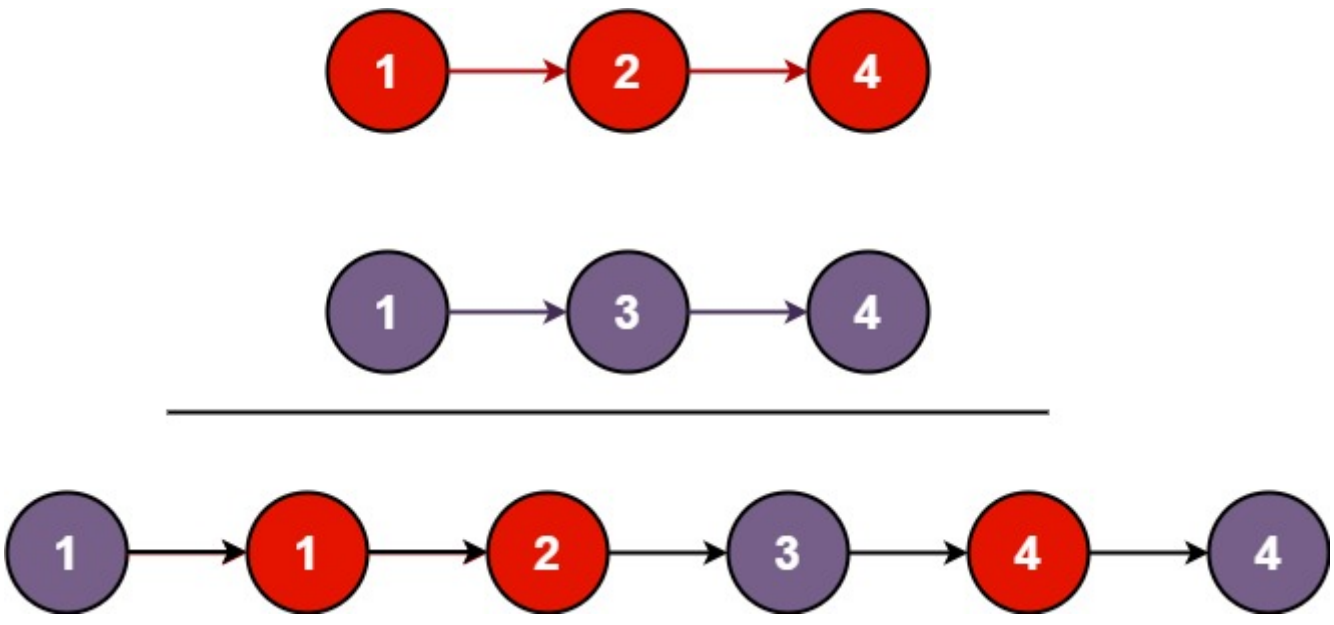
```
#include <iostream>
#include <stack>
#include <vector>
#include <list>

int main() {
    // 使用 std::vector 作为底层容器
    std::stack<int, std::vector<int>> stackWithVector;
    stackWithVector.push(1);
    stackWithVector.push(2);
    std::cout << "Top element with vector: " << stackWithVector.top() << std::endl;

    // 使用 std::list 作为底层容器
    std::stack<int, std::list<int>> stackWithList;
    stackWithList.push(3);
    stackWithList.push(4);
    std::cout << "Top element with list: " << stackWithList.top() << std::endl;

    return 0;
}
```

---

21. 合并两个有序链表

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：



```
输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]
```

示例 2：

```
输入：l1 = [], l2 = []
输出：[]
```

示例 3：

```
输入：l1 = [], l2 = [0]
输出：[0]
```

提示：

- 两个链表的节点数目范围是 `[0, 50]`
- `-100 <= Node.val <= 100`
- `l1` 和 `l2` 均按 **非递减顺序** 排列

题解

```cpp
#include <iostream>

// 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* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode dummy; // 虚拟头结点
        ListNode* tail = &dummy; // 尾指针指向虚拟头结点

        while (l1 && l2) {
            if (l1->val <= l2->val) {
                tail->next = l1;
                l1 = l1->next;
            } else {
                tail->next = l2;
                l2 = l2->next;
            }
            tail = tail->next;
        }

        // 将剩余的节点连接到新链表中
        tail->next = l1 ? l1 : l2;

        return dummy.next;        //因为dummy是一个虚拟头节点，所以返回dummy的下一个节点
```

```
    }
};

void printList(ListNode* node) {
    while (node) {
        std::cout << node->val << " ";
        node = node->next;
    }
    std::cout << std::endl;
}

int main() {
    // 创建链表 l1: [1, 2, 4]
    ListNode* l1 = new ListNode(1);
    l1->next = new ListNode(2);
    l1->next->next = new ListNode(4);

    // 创建链表 l2: [1, 3, 4]
    ListNode* l2 = new ListNode(1);
    l2->next = new ListNode(3);
    l2->next->next = new ListNode(4);

    Solution solution;
    ListNode* mergedList = solution.mergeTwoLists(l1, l2);

    // 输出合并后的链表
    printList(mergedList);

    return 0;
}
```

双向链表List

> 什么是 `std::list`

`std::list` 是一个双向链表容器。每个节点包含一个数据域和两个指针，一个指向前一个节点，一个指向后一个节点。双向链表允许在常数时间内在序列的两端进行插入和删除操作。

> 主要特性

- **双向链表**：每个节点包含前驱和后继指针。
- **常数时间插入和删除**：在任何位置插入或删除元素的时间复杂度为 O(1)。
- **线性时间访问**：访问元素的时间复杂度为 O(n)。
- **不支持随机访问**：不像 `std::vector`，`std::list` 不支持 `operator[]` 或 `at()` 进行随机访问。

要使用 `std::list`，需要包含头文件 `<list>`

> 创建和初始化

可以通过多种方式创建和初始化 `std::list`：

```
#include <iostream>
#include <list>
```

```cpp
int main() {
    // 创建一个空的 std::list
    std::list<int> mylist;

    // 用初始值列表初始化
    std::list<int> mylist2 = {1, 2, 3, 4, 5};

    // 使用特定数量的默认值初始化
    std::list<int> mylist3(5, 10); // 5 个元素，每个值为 10

    // 复制构造函数
    std::list<int> mylist4(mylist2);

    return 0;
}
```

> 常用成员函数

插入和删除

- `push_back(const T& val)`：在列表末尾插入元素。
- `push_front(const T& val)`：在列表开头插入元素。
- `pop_back()`：删除列表末尾的元素。
- `pop_front()`：删除列表开头的元素。
- `insert(iterator pos, const T& val)`：在指定位置插入元素。
- `erase(iterator pos)`：删除指定位置的元素。
- `clear()`：清空列表。

访问元素

- `front()`：返回列表第一个元素的引用。
- `back()`：返回列表最后一个元素的引用。

其他操作

- `size()`：返回列表中元素的数量。
- `empty()`：检查列表是否为空。
- `sort()`：对列表元素进行排序。
- `reverse()`：反转列表元素的顺序。

> 示例代码

以下是一些示例代码，展示了如何使用 `std::list` 进行基本操作：

```cpp
#include <iostream>
#include <list>

int main() {
    // 创建一个 std::list 并插入一些元素
    std::list<int> mylist = {2, 1, 4, 3};

    // 在列表末尾插入元素
```

```
    mylist.push_back(5);

    // 在列表开头插入元素
    mylist.push_front(0);

    // 遍历并打印列表元素
    for (int val : mylist) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // 删除列表开头的元素
    mylist.pop_front();

    // 删除列表末尾的元素
    mylist.pop_back();

    // 使用迭代器插入元素
    auto it = mylist.begin();
    std::advance(it, 2);
    mylist.insert(it, 8);

    // 使用迭代器删除元素
    mylist.erase(it);

    // 排序列表
    mylist.sort();

    // 反转列表
    mylist.reverse();

    // 打印修改后的列表
    for (int val : mylist) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

而在次题目，用的是单向链表

单向链表

单链表（Singly Linked List）是一种链式数据结构，其中每个节点包含数据和指向下一个节点的指针。与数组不同，单链表的节点不必在内存中连续存储。

> 单链表的结构

在单链表中，每个节点包含两个部分：

1. **数据部分**：存储节点的数据。
2. **指针部分**：存储指向下一个节点的指针。

链表的最后一个节点指向 `nullptr`，表示链表的结束。

> 单链表的基本操作

节点的定义

在 C++ 中，可以通过定义一个结构体或类来表示单链表的节点：

```
struct Node {
    int data;       // 数据部分
    Node* next;     // 指针部分
    Node(int val) : data(val), next(nullptr) {}
};
```

创建和初始化单链表

以下是如何创建和初始化单链表的示例：

```
#include <iostream>

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}          //构造函数，头节点
};

int main() {
    // 创建单向链表的节点
    Node* head = new Node(1); // 头节点
    head->next = new Node(2); // 第二个节点
    head->next->next = new Node(3); // 第三个节点

    // 打印链表中的元素
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }

    // 释放链表中的节点
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

> 插入节点

在单链表的开头、中间和结尾插入节点的示例：

```
void insertAtBeginning(Node*& head, int newData) {      //从头部插入
    Node* newNode = new Node(newData);
    newNode->next = head;
    head = newNode;
}
```

```
void insertAfter(Node* prevNode, int newData) {          //从某个位置插入
    if (prevNode == nullptr) {                           //首先判断插入的这个位置是不是空指针
        std::cout << "Previous node cannot be null." << std::endl;
        return;
    }
    Node* newNode = new Node(newData);
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}
//从尾部插入
void insertAtEnd(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    if (head == nullptr) {             //如果是一个空的链表,要单独写出来
        head = newNode;                //因为空的链表的指针是指向的nullptr，也就是空指针，空指针的下一个指向
        return;
    }
    Node* last = head;             //用last而不是直接移动head，是为了让last去找链表最后面的那个节点
    while (last->next != nullptr) {
        last = last->next;
    }
    last->next = newNode;
}
```

> 删除节点

删除单链表中的节点示例：

```
void deleteNode(Node*& head, int key) {   //这个key是按值搜索用的关键字
    Node* temp = head;                    //创建一个搜索用的临时指针，初始化搜索指针指向链表的头节点
    Node* prev = nullptr;                 //这个prev用来指代temp的前面的那个节点

    // 如果头节点本身是要删除的节点
    if (temp != nullptr && temp->data == key) {
        head = temp->next;                        //头指针往后移动
        delete temp;                              //删除这个节点
        return;
    }

    // 搜索要删除的节点
    while (temp != nullptr && temp->data != key) {  //条件：删除的临时指针不断在链表中后移，当且仅当没有到达链表末尾（空指向）以及与要删除的值不匹配的时候。    都进行搜索
        prev = temp;                        //这个prev也跟随着不断移动，永远指向temp的前面的那个节点
        temp = temp->next;
    }

    // 如果找不到要删除的节点
    if (temp == nullptr) {
        return;
    }

    // 从链表中删除节点
    prev->next = temp->next;              //直接跳过这个要删除的节点
    delete temp;                          //删除这个搜索用的临时指针
}
```

> 单链表的优势和劣势

优势

1. **动态大小**：链表大小可以动态增长或缩减，不需要预先定义大小。
2. **高效的插入和删除操作**：在已知位置进行插入和删除操作只需要常数时间。

劣势

1. **顺序访问**：链表不支持随机访问，查找某个特定元素需要线性时间。
2. **额外内存开销**：每个节点需要额外的指针内存。
3. **复杂性**：实现链表操作相对复杂，需要处理指针操作和边界情况。

> 完整示例代码

以下是一个完整的示例，展示了如何创建、插入、删除和遍历单链表：

```
#include <iostream>

struct Node {
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    newNode->next = head;
    head = newNode;
}

void insertAfter(Node* prevNode, int newData) {
    if (prevNode == nullptr) {
        std::cout << "Previous node cannot be null." << std::endl;
        return;
    }
    Node* newNode = new Node(newData);
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}

void insertAtEnd(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* last = head;
    while (last->next != nullptr) {
        last = last->next;
    }
    last->next = newNode;
}

void deleteNode(Node*& head, int key) {
```

```
    Node* temp = head;
    Node* prev = nullptr;

    // 如果头节点本身是要删除的节点
    if (temp != nullptr && temp->data == key) {
        head = temp->next;
        delete temp;
        return;
    }

    // 搜索要删除的节点
    while (temp != nullptr && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // 如果找不到要删除的节点
    if (temp == nullptr) {
        return;
    }

    // 从链表中删除节点
    prev->next = temp->next;
    delete temp;
}

void printList(Node* node) {
    while (node != nullptr) {
        std::cout << node->data << " ";
        node = node->next;
    }
    std::cout << std::endl;
}

int main() {
    Node* head = nullptr;    //创建一个头指针节点，它的指向为空

    insertAtEnd(head, 1);
    insertAtEnd(head, 2);
    insertAtEnd(head, 3);
    insertAtBeginning(head, 0);
    insertAfter(head->next, 5);

    std::cout << "Linked list: ";
    printList(head);

    deleteNode(head, 2);
    std::cout << "After deletion of 2: ";
    printList(head);

    // 释放链表中的节点
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
```

```
    return 0;
}
```

在这个示例中，我们展示了如何定义单链表节点，如何在链表的开头、中间和结尾插入节点，如何删除节点，以及如何遍历并打印链表中的元素。

---

26. 删除有序数组中的重复项

给你一个 **非严格递增排列** 的数组 `nums` ，请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ，返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 `nums` 中唯一元素的个数。

考虑 `nums` 的唯一元素的数量为 `k` ，你需要做以下事情确保你的题解可以被通过：

- 更改数组 `nums` ，使 `nums` 的前 `k` 个元素包含唯一元素，并按照它们最初在 `nums` 中出现的顺序排列。`nums` 的其余元素与 `nums` 的大小不重要。
- 返回 `k` 。

判题标准:

系统会用下面的代码来测试你的题解:

```
int[] nums = [...]; // 输入数组
int[] expectedNums = [...]; // 长度正确的期望答案

int k = removeDuplicates(nums); // 调用

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有断言都通过，那么您的题解将被 **通过。**

题解：

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.empty()) return 0; // 如果数组为空，直接返回 0

        int i = 0; // 用于指向当前处理好的唯一元素的末尾
        for (int j = 1; j < nums.size(); j++) { // 快指针，遍历整个数组
            if (nums[j] != nums[i]) { // 如果当前元素和慢指针指向的元素不同
                i++; // 慢指针向前移动一位
                nums[i] = nums[j]; // 将当前元素赋值给慢指针新位置
            }
        }

        return i + 1; // 返回唯一元素的个数
    }
};
```

Vector容器的介绍

`std::vector` 是 C++ 标准库中的一个动态数组容器，能够在运行时自动调整大小。它提供了许多便利的方法来操作元素，是 C++ 中常用的数据结构之一。

> 特点和优势：

1. **动态大小**：
   - `std::vector` 可以动态增长或缩减，根据需要自动调整存储空间大小。
   - 当向 `vector` 添加元素时，如果当前存储空间不足，`vector` 会自动分配更多内存，以容纳新元素。
2. **随机访问**：
   - `std::vector` 支持通过索引直接访问元素，具有常数时间复杂度（O(1)）。
   - 可以使用 `operator[]` 或 `at()` 方法来访问元素。
3. **元素存储连续**：
   - `std::vector` 的元素在内存中是连续存储的，这有利于缓存性能。
4. **标准库支持**：
   - `std::vector` 是 C++ 标准库中的一部分，提供了丰富的操作和算法，如排序、查找等。

> 使用示例：

创建和初始化 `vector`

```cpp
#include <iostream>
#include <vector>

int main() {
    // 创建一个空的 vector
    std::vector<int> vec1;

    // 初始化带有初始值的 vector
    std::vector<int> vec2 = {1, 2, 3, 4, 5};

    // 添加元素到 vector
    vec1.push_back(10);
    vec1.push_back(20);
    vec1.push_back(30);

    // 访问 vector 中的元素
    std::cout << "vec1 elements:";
    for (int i = 0; i < vec1.size(); ++i) {
        std::cout << " " << vec1[i];
    }
    std::cout << std::endl;

    std::cout << "vec2 elements:";
    for (auto& elem : vec2) {
        std::cout << " " << elem;
    }
    std::cout << std::endl;

    return 0;
}
```

> 常用操作：

- **访问元素**：使用 `operator[]` 或 `at()` 方法。

```
int value = vec1[0]; // 访问第一个元素
```

- **添加元素**：使用 `push_back()` 方法在尾部添加元素。

```
vec1.push_back(40); // 在尾部添加元素 40
```

- **删除元素**：使用 `pop_back()` 方法删除尾部元素。

```
vec1.pop_back(); // 删除尾部元素
```

- **获取大小**：使用 `size()` 方法获取 `vector` 的当前大小。

```
int size = vec1.size(); // 获取 vector 的大小
```

- **遍历元素**：使用范围-based for 循环或迭代器遍历 `vector` 中的元素。

```
for (auto& elem : vec2) {
    std::cout << elem << " ";
}
```

---

27. 移除元素

给你一个数组 `nums` 和一个值 `val`，你需要 **原地** 移除所有数值等于 `val` 的元素。元素的顺序可能发生改变。然后返回 `nums` 中与 `val` 不同的元素的数量。

假设 `nums` 中不等于 `val` 的元素数量为 `k`，要通过此题，您需要执行以下操作：

- 更改 `nums` 数组，使 `nums` 的前 `k` 个元素包含不等于 `val` 的元素。`nums` 的其余元素和 `nums` 的大小并不重要。
- 返回 `k`。

用户评测：

评测机将使用以下代码测试您的解决方案：

```
int[] nums = [...]; // 输入数组
int val = ...; // 要移除的值
int[] expectedNums = [...]; // 长度正确的预期答案。
                            // 它以不等于 val 的值排序。

int k = removeElement(nums, val); // 调用你的实现

assert k == expectedNums.length;
sort(nums, 0, k); // 排序 nums 的前 k 个元素
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有的断言都通过，你的解决方案将会 **通过。**

示例 1：

```
输入：nums = [3,2,2,3], val = 3
输出：2, nums = [2,2,_,_]
解释：你的函数函数应该返回 k = 2, 并且 nums 中的前两个元素均为 2。
你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。
```

示例 2：

```
输入：nums = [0,1,2,2,3,0,4,2], val = 2
输出：5, nums = [0,1,4,0,3,_,_,_]
解释：你的函数应该返回 k = 5，并且 nums 中的前五个元素为 0,0,1,3,4。
注意这五个元素可以任意顺序返回。
你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。
```

提示：

- `0 <= nums.length <= 100`
- `0 <= nums[i] <= 50`
- `0 <= val <= 100`

题解：

```cpp
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int i = 0;      //慢指针
        for (int j = 0; j < nums.size(); j++) {
            nums[i] = nums[j];
            if (nums[i] != val) {
                i++;
            }
        }
        return i;
    }
};
```

---

28. 找出字符串中第一个匹配项的下标(KMP)

给你两个字符串 `haystack` 和 `needle` ，请你在 `haystack` 字符串中找出 `needle` 字符串的第一个匹配项的下标（下标从 0 开始）。如果 `needle` 不是 `haystack` 的一部分，则返回 `-1` 。

示例 1：

```
输入：haystack = "sadbutsad", needle = "sad"
输出：0
解释："sad" 在下标 0 和 6 处匹配。
第一个匹配项的下标是 0 ，所以返回 0 。
```

示例 2：

```
输入：haystack = "leetcode", needle = "leeto"
输出：-1
解释："leeto" 没有在 "leetcode" 中出现，所以返回 -1 。
```

提示：

- `1 <= haystack.length, needle.length <= 104`
- `haystack` 和 `needle` 仅由小写英文字符组成

题解：

```
class Solution {
public:
    //求next 数组
    void buildPrefixTable(const string& pattern) {
        int i = 1;
        int prefix_length = 0;
        this->next.push_back(0);  //第一个是0
        while (i < pattern.length())
        {
            if (pattern[i] == pattern[prefix_length]) {
                i++;
                prefix_length++;
                this->next.push_back(prefix_length);
            }
            else
            {
                if (prefix_length != 0) {            //可以查表
                    prefix_length = this->next[prefix_length - 1];
                }
                else
                {
                    this->next.push_back(0);
                    i++;
                }
            }
        }
    }
    //kmp搜索
    void kmp_Search(string& pattern) {
        this->buildPrefixTable(pattern);
        int i = 0;
        int j = 0;
        while (i < this->essay.length())
        {
            if (this->essay[i] == pattern[j]) {
                i++;
                j++;
            }
            else
            {
                if (j > 0) {
                    j = this->next[j - 1];
                }
```

```
                else
                {
                    i++;
                }
            }
            if (j == pattern.length()) {
                this->kmp_Find = i - j;
                return;
            }
        }
    }
public:

    vector<int> next;          //这是所求的next数组
    string essay;              //这是输入的原文
    int kmp_Find;              //这是最终找到的下标的位置
};
```

KMP算法讲解

KMP算法（Knuth-Morris-Pratt算法）是一种用于在字符串中查找子串的高效算法。它在进行匹配时避免了重复的比较，因此在最坏情况下时间复杂度为O(n + m)，其中n是主字符串的长度，m是子串的长度。

KMP算法的核心思想是通过预处理模式串，构建一个部分匹配表（又称为失配函数、前缀表、Next数组），用来记录模式串的前缀和后缀的匹配情况。这样，当出现不匹配时，可以根据前缀表跳过一些不必要的匹配，从而提高效率。

> KMP算法步骤

1. **构建前缀表（Next数组）**：
    - 计算模式串的前缀和后缀的最长公共部分长度，用数组`next`记录。
    - `next[i]`表示模式串中前`i`个字符的最长相等前缀后缀的长度。
2. **利用前缀表进行匹配**：
    - 使用`next`数组在主字符串中查找模式串，遇到不匹配时，利用`next`数组跳过一些字符继续匹配。

---

35. 搜索插入位置

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请必须使用时间复杂度为 `O(log n)` 的算法。

示例 1:

```
输入: nums = [1,3,5,6], target = 5
输出: 2
```

示例 2:

```
输入: nums = [1,3,5,6], target = 2
输出: 1
```

示例 3:

```
输入: nums = [1,3,5,6], target = 7
输出: 4
```

提示:

- `1 <= nums.length <= 104`
- `-104 <= nums[i] <= 104`
- `nums` 为 **无重复元素** 的 **升序** 排列数组
- `-104 <= target <= 104`

题解（简单）

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {

        int i = 0;
        while (i<nums.size())
        {
            if (nums[i] >= target) {
                return i;
            }
            i++;
        }
        return nums.size();
    }
};
```

---

58. 最后一个单词的长度

给你一个字符串 `s`，由若干单词组成，单词前后用一些空格字符隔开。返回字符串中 **最后一个** 单词的长度。

单词 是指仅由字母组成、不包含任何空格字符的最大子字符串

示例 1：

```
输入：s = "Hello World"
输出：5
解释：最后一个单词是“World”，长度为 5。
```

示例 2：

```
输入：s = "   fly me   to   the moon  "
输出：4
解释：最后一个单词是“moon”，长度为 4。
```

示例 3：

```
输入：s = "luffy is still joyboy"
输出：6
解释：最后一个单词是长度为 6 的“joyboy”。
```

提示：

- `1 <= s.length <= 104`
- `s` 仅有英文字母和空格 `' '` 组成
- `s` 中至少存在一个单词

题解

```
class Solution {
public:
    int lengthOfLastWord(string s) {
        int i = 0;
        int j = 0;
        while ( i<(s.length() - 1))
        {


            if (s[i] != ' ') {
                j ++;
            }
            if (s[i] == ' ' && s[i + 1] != ' ') {
                j = 0;
                i++;
            }
        }
        if (s[s.length()-1] != ' ') { j++; };
        return j;
    }
};
```

---

66. 加一

给定一个由 **整数** 组成的 **非空** 数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储**单个**数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1：

```
输入：digits = [1,2,3]
输出：[1,2,4]
解释：输入数组表示数字 123。
```

示例 2：

```
输入：digits = [4,3,2,1]
输出：[4,3,2,2]
解释：输入数组表示数字 4321。
```

示例 3：

```
输入：digits = [0]
输出：[1]
```

提示：

- `1 <= digits.length <= 100`
- `0 <= digits[i] <= 9`

题解

```
    vector<int> plusOne(vector<int>& digits) {
        int n = digits.size();
        for (int i = n - 1; i >= 0; i--) {
            digits[i]++;
            digits[i] %= 10;
            if (digits[i] != 0) {
                return digits;
            }
        }
                //只有全是9的情形，才会走到这一步
        vector<int> result;
        result.push_back(1);
        for (int j = 0; j < n; j++) {
            result.push_back(digits[j]);
        }
        return result;
    }
};
```

---

67. 二进制求和

给你两个二进制字符串 `a` 和 `b` ，以二进制字符串的形式返回它们的和。

示例 1：

```
输入:a = "11", b = "1"
输出："100"
```

示例 2：

```
输入：a = "1010", b = "1011"
输出："10101"
```

提示：

- `1 <= a.length, b.length <= 104`

- `a` 和 `b` 仅由字符 `'0'` 或 `'1'` 组成
- 字符串如果不是 `"0"` ，就不含前导零

题解

```
class Solution {
public:
    string addBinary(string a, string b) {
        //这几行代码是对短的那一条进行补零
        int c= a.length();
        int d = b.length();
        while (c<d)                    //while和for的区别就是while会先做一次判断     while = if+循环   for =循环+if
        {
            a = '0' + a;
            c++;
        }
        while (c>d)
        {
            b = '0' + b;
            d++;
        }
        //先计算，后判断，所以这里使用for
        for (int i = d - 1; i > 0; i--) {
            a[i] = a[i] - '0' + b[i];
            if (a[i] > '1') {                       //如果要进位就这样写
                a[i] = (a[i] - '0') % 2 + '0';              //这里之所以要这样写是因为
                a[i - 1] = a[i - 1] + 1;                //正是因为for循环先计算后判断，所以存在i-1
            }
           // cout << "此时的a[" << i << "]=" << a[i] << "   a[" << i << "-1]=" << a[i - 1] << endl;
        };
      //单独判断第一个相加需不需要进位
        a[0] = a[0] - '0' + b[0];
        if (a[0] > '1') {
            a[0] = (a[0] - '0') % 2 + '0';
            a = '1' + a;                        //在前面拼接1
        }
        return a;
    }
```

for循环和while循环

for循环的顺序：

A:执行初始化语句
B : 执行判断条件语句，看其结果是true还是false
如果是false，循环结束。
如果是true，继续执行。
C : 执行循环体语句
D : 执行控制条件语句
E : 回到B继续

> `for` 循环和 `while` 循环的比较

特征	`for` 循环	`while` 循环
初始化语句	在循环开始前执行一次，用于初始化循环控制变量。	通常在循环前执行，但不在循环头中。
判断条件语句	在每次迭代开始前执行。如果为 `false`，循环结束。	在每次迭代开始前执行。如果为 `false`，循环结束。
循环体语句	在判断条件为 `true` 的情况下执行。	在判断条件为 `true` 的情况下执行。
控制条件语句（增量）	在每次迭代后执行，通常用于更新循环控制变量。	通常在循环体内手动执行。
使用场景	当循环次数已知或容易确定时使用。	当循环次数未知或依赖动态条件时使用。

---

69. x 的平方根

给你一个非负整数 `x` ，计算并返回 `x` 的 **算术平方根** 。

由于返回类型是整数，结果只保留 **整数部分** ，小数部分将被 **舍去** 。

注意：不允许使用任何内置指数函数和算符，例如 `pow(x, 0.5)` 或者 `x ** 0.5` 。

示例 1：

```
输入：x = 4
输出：2
```

示例 2：

```
输入：x = 8
输出：2
解释：8 的算术平方根是 2.82842..., 由于返回类型是整数，小数部分将被舍去。
```

提示：

- `0 <= x <= 231 - 1`

题解：

```cpp
class Solution {
public:
    int mySqrt(int x) {
      int left = 0, right = x, ans = -1;
        while (left <= r) {
            int mid = left + (right - left) / 2;
            if ((long long)mid * mid <= x) {
                ans = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return ans;
```

```
    }
};
```

实现步骤：测试 `solution.mySqrt(2147395599);`

```
输出结果:
left = 0  mid = 1073697799  right = 1073697798  anwser = -1
left = 0  mid = 536848899  right = 536848898  anwser = -1
left = 0  mid = 268424449  right = 268424448  anwser = -1
left = 0  mid = 134212224  right = 134212223  anwser = -1
left = 0  mid = 67106111  right = 67106110  anwser = -1
left = 0  mid = 33553055  right = 33553054  anwser = -1
left = 0  mid = 16776527  right = 16776526  anwser = -1
left = 0  mid = 8388263  right = 8388262  anwser = -1
left = 0  mid = 4194131  right = 4194130  anwser = -1
left = 0  mid = 2097065  right = 2097064  anwser = -1
left = 0  mid = 1048532  right = 1048531  anwser = -1
left = 0  mid = 524265  right = 524264  anwser = -1
left = 0  mid = 262132  right = 262131  anwser = -1
left = 0  mid = 131065  right = 131064  anwser = -1
left = 0  mid = 65532  right = 65531  anwser = -1
left = 32766  mid = 32765  right = 65531  anwser = 32765
left = 32766  mid = 49148  right = 49147  anwser = 32765
left = 40957  mid = 40956  right = 49147  anwser = 40956
left = 45053  mid = 45052  right = 49147  anwser = 45052
left = 45053  mid = 47100  right = 47099  anwser = 45052
left = 46077  mid = 46076  right = 47099  anwser = 46076
left = 46077  mid = 46588  right = 46587  anwser = 46076
left = 46333  mid = 46332  right = 46587  anwser = 46332
left = 46333  mid = 46460  right = 46459  anwser = 46332
left = 46333  mid = 46396  right = 46395  anwser = 46332
left = 46333  mid = 46364  right = 46363  anwser = 46332
left = 46333  mid = 46348  right = 46347  anwser = 46332
left = 46333  mid = 46340  right = 46339  anwser = 46332
left = 46337  mid = 46336  right = 46339  anwser = 46336
left = 46339  mid = 46338  right = 46339  anwser = 46338
left = 46340  mid = 46339  right = 46339  anwser = 46339
46339
```

（未解决）70. 爬楼梯

假设你正在爬楼梯。需要 `n` 阶你才能到达楼顶。

每次你可以爬 `1` 或 `2` 个台阶。你有多少种不同的方法可以爬到楼顶呢?

示例 1：

```
输入：n = 2
输出：2
解释：有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2：

```
输入：n = 3
输出：3
解释：有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

提示：

- `1 <= n <= 45`

题解：

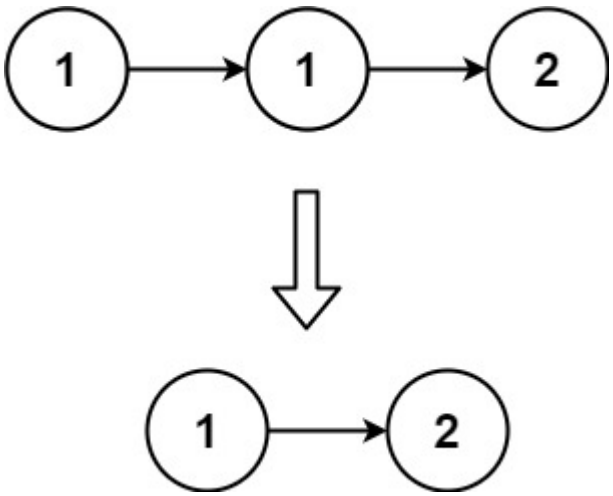
```
Microsoft Visual Studio 调试控制台
Leecode Problem 70:爬梯子
矩阵快速幂
调用vector<vector<long long>> res = matrixPow(ret, n);
c[0][0] = 1       c[0][1] = 1
c[1][0] = 1       c[1][1] = 0
ret = multiply(ret, a);
++++++++++++++++++++++++++++++
a = multiply(a, a);
c[0][0] = 2       c[0][1] = 1
c[1][0] = 1       c[1][1] = 1

a = multiply(a, a);
c[0][0] = 5       c[0][1] = 3
c[1][0] = 3       c[1][1] = 2

c[0][0] = 8       c[0][1] = 5
c[1][0] = 5       c[1][1] = 3
ret = multiply(ret, a);
++++++++++++++++++++++++++++++
a = multiply(a, a);
c[0][0] = 34       c[0][1] = 21
c[1][0] = 21       c[1][1] = 13

最后打印一下，res数组:
res[0][0] = 8     res[0][1] = 5
res[1][0] = 5     res[1][1] = 3
8
D:\Work\workspace\C++\leeCode\test01\LeecodeTest01\x64\Debug\LeecodeTest01.exe (进程 10076)已退出，代码为 0。
按任意键关闭此窗口. . .
```

---

83. 删除排序链表中的重复元素

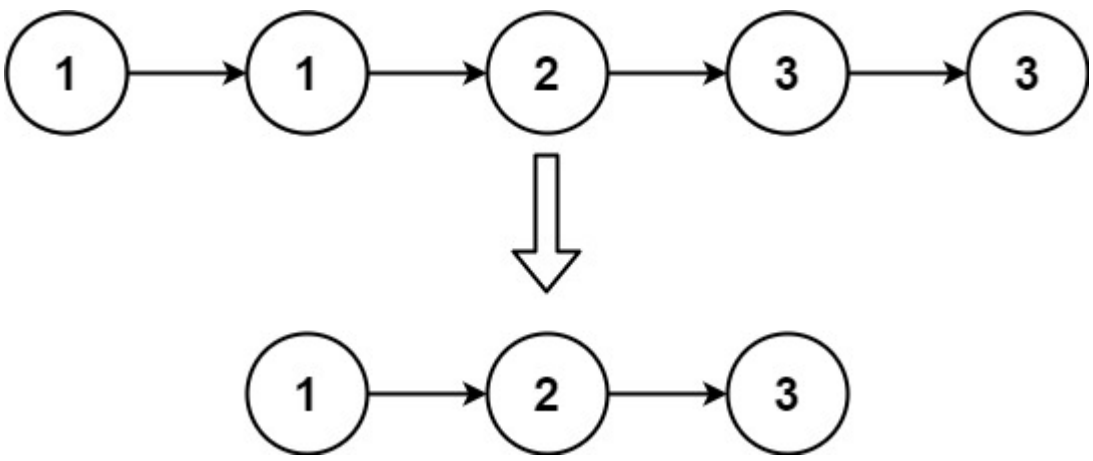
给定一个已排序的链表的头 `head` ， *删除所有重复的元素，使每个元素只出现一次* 。返回 *已排序的链表* 。

示例 1：




```
输入：head = [1,1,2]
输出：[1,2]
```

示例 2：




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

提示：

- 链表中节点数目在范围 `[0, 300]` 内
- `-100 <= Node.val <= 100`
- 题目数据保证链表已经按升序 **排列**

题解：

```cpp
/**
 * 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* deleteDuplicates(ListNode* head) {

    }
};
```

---

704. 二分查找

给定一个 `n` 个元素有序的（升序）整型数组 `nums` 和一个目标值 `target` ，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。

示例 1:

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

示例 2:

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不存在 nums 中因此返回 -1
```

提示:

1. 你可以假设 `nums` 中的所有元素是不重复的。
2. `n` 将在 `[1, 10000]` 之间。
3. `nums` 的每个元素都将在 `[-9999, 9999]` 之间。

题解

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int i = nums.size()-1;    //高指针
        int j = 0;               //低指针
        int mid = i;
        if (nums[0] == target  ) {
            return 0;
        }
        if (nums[i] == target)
        {
            return i;
        }
        while (i>=j)
        {
            mid = (i + j) / 2;
            if (nums[mid]<target)    //
            {
                j = mid+1;
            }
            else if (nums[mid] > target) {
                i = mid-1;
```

```
            }
            else
            {
                return mid;
            }
        }
        return -1;
    }
};
```

[977. 有序数组的平方](#)（有待优化）

给你一个按 **非递减顺序** 排序的整数数组 `nums`，返回 **每个数字的平方** 组成的新数组，要求也按 **非递减顺序** 排序。

示例 1：

```
输入：nums = [-4,-1,0,3,10]
输出：[0,1,9,16,100]
解释：平方后，数组变为 [16,1,0,9,100]
排序后，数组变为 [0,1,9,16,100]
```

示例 2：

```
输入：nums = [-7,-3,2,3,11]
输出：[4,9,9,49,121]
```

提示：

- `1 <= nums.length <= 104`
- `-104 <= nums[i] <= 104`
- `nums` 已按 **非递减顺序** 排序

进阶：

- 请你设计时间复杂度为 `O(n)` 的算法解决本问题

题解：

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        int left = 0;
        int right = nums.size() - 1;
        int i = 0;
        int j = 0;
        vector<int> temp;
        while (left<=right)
        {
            i = nums[left] * nums[left];
            j = nums[right] * nums[right];
            if (i<= j) {
                temp.insert(temp.begin(), j);
                right -= 1;
```

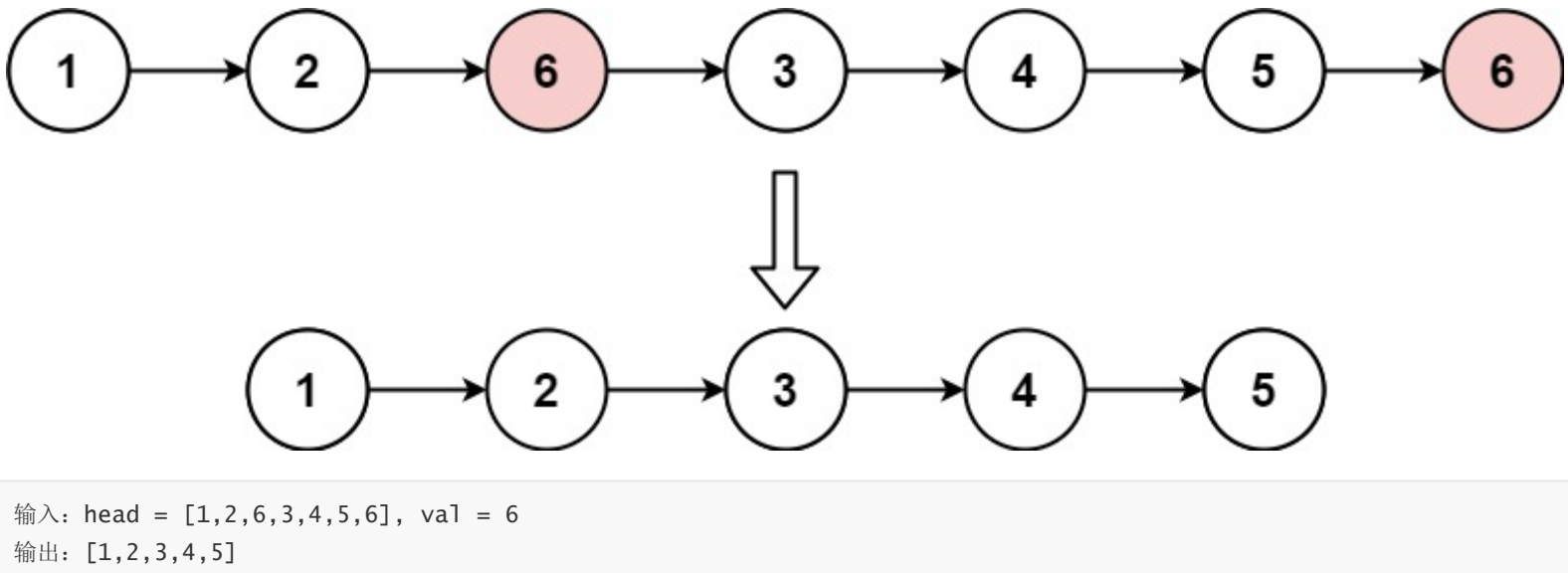
```
            }
            else
            {
                temp.insert(temp.begin(), i);
                left += 1;
            }
        }
        return temp;
    }
};
```

---

203. 移除链表元素

给你一个链表的头节点 `head` 和一个整数 `val` ，请你删除链表中所有满足 `Node.val == val` 的节点，并返回 **新的头节点** 。

示例 1：




```
输入：head = [1,2,6,3,4,5,6], val = 6
输出：[1,2,3,4,5]
```

示例 2：

```
输入：head = [], val = 1
输出：[]
```

示例 3：

```
输入：head = [7,7,7,7], val = 7
输出：[]
```

提示：

- 列表中的节点数目在范围 `[0, 104]` 内
- `1 <= Node.val <= 50`
- `0 <= val <= 50`

题解：

虚拟头节点的方式(因为循环内是删除下一个节点，不方便对头节点进行操作，所以我们不妨在头节点前端加上一个虚拟头节点，再让这个新的链表放入while循环中遍历)

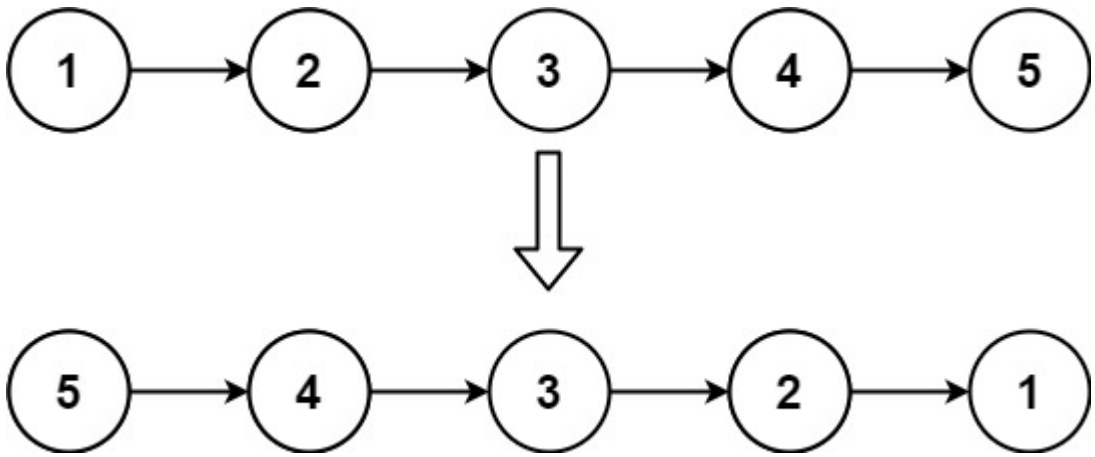
```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
 ListNode* dummy =new  ListNode(0,head);           //使用一个 dummy来临时加到head前面
 ListNode* current = dummy;                       //使用current来指向这个新的头节点，让current来完成遍历的工作
 while (current->next!=nullptr)
 {
     if (current->next->val == val) {
         ListNode* temp = current->next;            //用一个临时指针来保存这个即将被释放掉的内存空间
         current->next = current->next->next;          //为了保证这一条语句能正常执行，所以，相当于是从第二个元素开始遍历
         delete temp;
     }
     else
     {
         current = current->next;
     }
 }
 //接下来的工作就是删除我们临时创建的dummy头节点 ,为什么直接不直接返回head，是因为head也会被删除(假如有删除头节点的情况)
 ListNode* newHead = dummy->next;
 delete dummy;
 return newHead;
    }
};
```

---

206. 反转链表

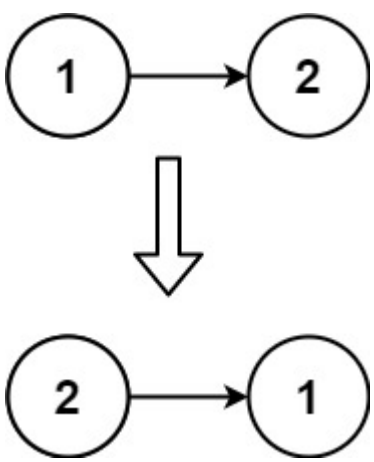
给你单链表的头节点 `head` ，请你反转链表，并返回反转后的链表。

示例 1：




```
输入：head = [1,2,3,4,5]
输出：[5,4,3,2,1]
```

示例 2：




```
输入：head = [1,2]
输出：[2,1]
```

示例 3：

```
输入：head = []
输出：[]
```

提示：

- 链表中节点的数目范围是 `[0, 5000]`
- `-5000 <= Node.val <= 5000`

进阶：链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题？

题解：（双指针）

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

```
        ListNode* preNode = new ListNode(0, head);
        ListNode* current = head;
        ListNode* temp = head;
        while (current!=nullptr)
        {
            temp = current->next;
            current->next = preNode;
            preNode = current;
            current = temp;
        }
        head->next = nullptr;
        return preNode;
    }
};
```

题解: (递归)

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        // 基本情况：如果链表为空或只有一个节点，直接返回该节点
        if (head == nullptr || head->next == nullptr) {
            return head;
        }

        // 递归地反转剩余的链表
        ListNode* newHead = reverseList(head->next);

        // 反转当前节点的指针
        head->next->next = head;
        head->next = nullptr;

        return newHead;
    }
};
```

---

242. 有效的字母异位词

给定两个字符串 `*s*` 和 `*t*` ，编写一个函数来判断 `*t*` 是否是 `*s*` 的字母异位词。

注意：若 `*s*` 和 `*t*` 中每个字符出现的次数都相同，则称 `*s*` 和 `*t*` 互为字母异位词。

示例 1:

```
输入: s = "anagram", t = "nagaram"
输出: true
```

示例 2:

```
输入: s = "rat", t = "car"
输出: false
```

提示:

- `1 <= s.length, t.length <= 5 * 104`
- `s` 和 `t` 仅包含小写字母

进阶: 如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

题解：

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        vector<int> ruan(26, 0);
        for (int i = 0; i < s.length(); i++) {
            ruan[int(s[i])-97]++;
        }
        for (int i = 0; i < t.length(); i++) {
            ruan[int(t[i])-97]--;
        }
        for (int i = 0; i < ruan.size(); i++)
        {
            if (ruan[i] != 0) {
                return false;
            }
        }
        return true;
    }
};
```

笔记：这是第一个关于哈希表的题目

如果要涉及的数组比较短，就用数组，如果比较长，就用set，如果涉及键值对value，那就用map。

解法思路：创建一个长度为26，默认值为0的数组，遍历第一个数组，在每一个存在的字符对应vector数组的位置下面加一，遍历第二个数组，对每一个存在的字符对应的vector数组的位置下面减一，最后遍历整个vector数组，看数组的内容是不是全部都是0；

---

349. 两个数组的交集

给定两个数组 `nums1` 和 `nums2` ，返回 *它们的*

交集

。输出结果中的每个元素一定是 **唯一** 的。我们可以 **不考虑输出结果的顺序** 。

示例 1：

```
输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]
```

示例 2：

```
输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]
解释：[4,9] 也是可通过的
```

提示：

- `1 <= nums1.length, nums2.length <= 1000`
- `0 <= nums1[i], nums2[i] <= 1000`

题解：

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {

    }class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        vector<int> ruan(1001,0);
        vector<int> anwser;
        for (int i = 0; i < nums1.size(); i++) {
            ruan[nums1[i] ]++;
    }
        for (int i = 0; i < nums2.size(); i++) {
            if (ruan[nums2[i] ] != 0) {
                ruan[nums2[i] ] = -1;             //第二次遍历的时候，直接将重复的数组修改为-1
            }
        }
        for (int i = 0; i < 1001; i++)           //将值为-1的数添加到anwser数组中输出
        {
            if (ruan[i] == -1) {
                anwser.push_back(i );
            }
        }
        return anwser;
    }
};
};
```

还有一种使用过set数据类型的解法，set的特点就是存入的数据不重复，并且还会给存入的数据自动排序

---

1. 两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值** *`target`* 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 1：

```
输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
```

示例 2：

```
输入：nums = [3,2,4], target = 6
输出：[1,2]
```

示例 3：

```
输入：nums = [3,3], target = 6
输出：[0,1]
```

提示：

- `2 <= nums.length <= 104`
- `-109 <= nums[i] <= 109`
- `-109 <= target <= 109`
- **只会存在一个有效答案**

进阶：你可以想出一个时间复杂度小于 `O(n2)` 的算法吗?

尝试使用hashmap求解：

题解：

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
     int n = nums.size();
     for(int i = 0;i<n;i++){
        for(int j =i+1; j<n;j++){
            if(nums[i]+nums[j]==target){
                return {i,j};
            }
        }
     }
    return {};
    }
};
```

---

344. 反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 `s` 的形式给出。

不要给另外的数组分配额外的空间，你必须**原地修改输入数组**、使用 O(1) 的额外空间解决这一问题。

示例 1：

```
输入：s = ["h","e","l","l","o"]
输出：["o","l","l","e","h"]
```

示例 2：

```
输入：s = ["H","a","n","n","a","h"]
输出：["h","a","n","n","a","H"]
```

提示：

- `1 <= s.length <= 105`

- `s[i]` 都是 ASCII 码表中的可打印字符

题解：

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        char temp;
        int left = 0;
        int right = s.size()-1;
        while ( left<right )
        {
            temp = s[left];
            s[left] = s[right];
            s[right] = temp;
            left++;
            right--;
        }
    }
};
```

笔记：太简单

---

541. 反转字符串 II

给定一个字符串 `s` 和一个整数 `k`，从字符串开头算起，每计数至 `2k` 个字符，就反转这 `2k` 字符中的前 `k` 个字符。

- 如果剩余字符少于 `k` 个，则将剩余字符全部反转。
- 如果剩余字符小于 `2k` 但大于或等于 `k` 个，则反转前 `k` 个字符，其余字符保持原样。

示例 1：

```
输入：s = "abcdefg", k = 2
输出："bacdfeg"
```

示例 2：

```
输入：s = "abcd", k = 2
输出："bacd"
```

提示：

- `1 <= s.length <= 104`
- `s` 仅由小写英文组成
- `1 <= k <= 104`

题解：

```
class Solution {
public:
    string reverseStr(string s, int k) {
        int size = s.size();
```

```
        for (int i = 0; i < size; i += (2 * k))
        {
            if (i + k <= size)
            {
                reverse(s.begin() + i, s.begin() + i + k);
                continue;
            }
            reverse(s.begin() + i, s.end());
        }
        return s;
    }
};
```

笔记：它的方法使用了string 下的reverse 函数，做判断：

1.当截取长度在K到2K之间：都是反转前K个数

2.当截取长度已经不足一个K的长度，那么直接反转剩下的截取长度的字符串。

---

459. 重复的子字符串

给定一个非空的字符串 `s` ，检查是否可以通过由它的一个子串重复多次构成。

示例 1:

```
输入: s = "ab'ab"
输出: true
解释: 可由子串 "ab" 重复两次构成。
```

示例 2:

```
输入: s = "aba"
输出: false
```

示例 3:

```
输入: s = "abcabcabcabc"
输出: true
解释: 可由子串 "abc" 重复四次构成。 (或子串 "abcabc" 重复两次构成。)
```

提示:

- `1 <= s.length <= 104`
- `s` 由小写英文字母组成

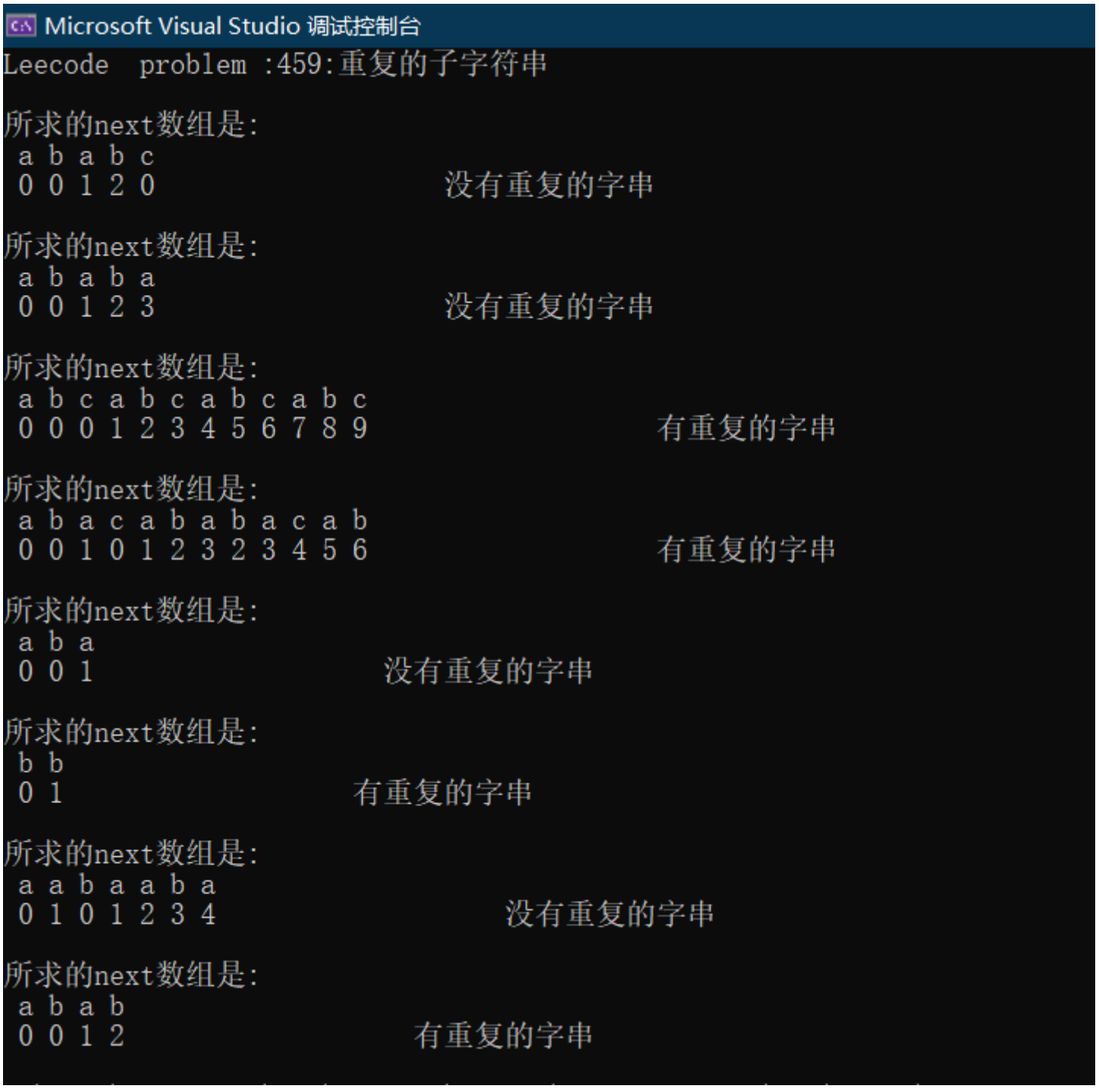
题解：(459.1.cpp)

```
class Solution {
public:
    void buildPrefixTable(const string& pattern) {
                    this->next.clear();                 //每一次都要清空表
                    int i = 1;
```

```
                int prefix_length = 0;
                this->next.push_back(0);
                while (i<pattern.length())
                {
                    if (pattern[i] == pattern[prefix_length]) {
                        i++;
                        prefix_length++;
                        this->next.push_back(prefix_length);
                    }
                    else
                    {
                        if (prefix_length != 0) {            //可以查表
                            prefix_length = this->next[prefix_length-1];
                        }
                        else
                        {
                            this->next.push_back(0);
                            i++;
                        }
                    }
                }
                //printThisNextStruct(pattern);
            }

    bool repeatedSubstringPattern(string s) {
        this->buildPrefixTable(s);
         if (s.length()%(s.length()-this->next[s.length()-1])==0 && this->next[s.length() - 1]!= 0)
            {
                return true;
            }
        return false;
    }

public:
    vector<int> next;
};
```

笔记:

请观察：



这道题首先用KMP求出next数组。观察上图想出next数组的用途。

然后观察next数组：

1.满足next数组的最后一位不是0

2.单个最短重复的子字符串的长度为：**s.length() - next数组的最后一位**，这个长度能够被s.length()整除。

---

232. 用栈实现队列

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（`push`、`pop`、`peek`、`empty`）：

实现 `MyQueue` 类：

- `void push(int x)` 将元素 x 推到队列的末尾
- `int pop()` 从队列的开头移除并返回元素

- `int peek()` 返回队列开头的元素
- `boolean empty()` 如果队列为空，返回 `true` ；否则，返回 `false`

说明：

- 你 **只能** 使用标准的栈操作 —— 也就是只有 `push to top`, `peek/pop from top`, `size`, 和 `is empty` 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

示例 1：

```
输入：
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
输出：
[null, null, null, 1, 1, false]

解释：
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1                                  返回
myQueue.pop(); // return 1, queue is [2]                    移除队头元素，并返回
myQueue.empty(); // return false
```

提示：

- `1 <= x <= 9`
- 最多调用 `100` 次 `push`、`pop`、`peek` 和 `empty`
- 假设所有操作都是有效的 （例如，一个空的队列不会调用 `pop` 或者 `peek` 操作）

进阶：

- 你能否实现每个操作均摊时间复杂度为 `O(1)` 的队列？换句话说，执行 `n` 个操作的总时间复杂度为 `O(n)` ，即使其中一个操作可能花费较长时间。

题解：

```
class MyQueue {
public:
    MyQueue() {    //啥也不用做

    }

    void push(int x) {
        this->ruan.push(x);
    }

    int pop() {
        stack<int> temp;
        while (!ruan.empty())
        {
            temp.push(ruan.top());
            ruan.pop();
        }
```

```
        int tempNum = temp.top();
        temp.pop();
        while (!temp.empty())
        {
            ruan.push(temp.top());
            temp.pop();
        }
        return tempNum;
    }

    int peek() {            //返回栈底元素
        stack<int> temp;
        while (!ruan.empty())
        {
            temp.push(ruan.top());
            ruan.pop();
        }
        int tempNum = temp.top();
        while (!temp.empty())
        {
            ruan.push(temp.top());
            temp.pop();
        }
        return tempNum;
    }

    bool empty() {
        return ruan.empty();
    }
public:
    stack<int> ruan;
};
```

笔记：就是用一个栈来存储，然后创建一个临时栈用来反转和输出。

---

225. 用队列实现栈

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（`push`、`top`、`pop` 和 `empty`）。

实现 `MyStack` 类：

- `void push(int x)` 将元素 x 压入栈顶。
- `int pop()` 移除并返回栈顶元素。
- `int top()` 返回栈顶元素。
- `boolean empty()` 如果栈是空的，返回 `true` ； 否则，返回 `false` 。

注意：

- 你只能使用队列的标准操作 —— 也就是 `push to back`、`peek/pop from front`、`size` 和 `is empty` 这些操作。
- 你所使用的语言也许不支持队列。 你可以使用 list （列表）或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。

示例：

```
输入：
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
输出：
[null, null, null, 2, 2, false]

解释：
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // 返回 2
myStack.pop(); // 返回 2
myStack.empty(); // 返回 False
```

提示：

- `1 <= x <= 9`
- 最多调用`100`次`push`、`pop`、`top`和`empty`
- 每次调用`pop`和`top`都保证栈不为空

进阶：你能否仅用一个队列来实现栈。

题解：

```
class MyStack {
public:
    MyStack() {

    }

    void push(int x) {
        ruan.push(x);
    }

    int pop() {
        int tempNum = 0;
        for (int i = 0; i < ruan.size()-1; i++)
        {
            tempNum = ruan.front();
            ruan.pop();
            ruan.push(tempNum);
        }
        tempNum = ruan.front();
        ruan.pop();
        return tempNum;
    }

    int top() {
        int tempNum = 0;
        for (int i = 0; i < ruan.size();i++) {
            tempNum = ruan.front();
            ruan.pop();
            ruan.push(tempNum);
       }
```

```
        return tempNum;
    }

    bool empty() {
        return ruan.empty();
    }
public:
    queue<int> ruan;
};
```

笔记：其实就是让队列不停弹出元素，然后将弹出的元素再追加到自己的末尾，当弹出myStack.size()-1个元素的时候，此时队列的对头元素就是之前队列要找的最后一个元素。

1047. 删除字符串中的所有相邻重复项

给出由小写字母组成的字符串 `S`，**重复项删除操作**会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

```
输入："abbaca"
输出："ca"
解释：
例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，
其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。
```

提示：

1. `1 <= S.length <= 20000`
2. `S` 仅由小写英文字母组成。

题解：

```
class Solution {
public:
    string removeDuplicates(string s) {
        stack<char> ruan;
        string aws = "";
        for (int i = 0; i < s.length(); i++) {
            if (ruan.empty()) {
                ruan.push(s[i]);
                i++;
            }
            if (i >= s.length()) continue;

            if (s[i]!=ruan.top()) {
                ruan.push(s[i]);
            }
            else
            {
                ruan.pop();
```

```
            }
        }
        //接下来是反转ruan中的字符并加入到结果字符串当中去
        stack<char> temp;
        while (!ruan.empty())
        {
            temp.push(ruan.top());
            ruan.pop();
        }
        while (!temp.empty())
        {
            aws += temp.top();
            temp.pop();
        }
        return aws;
    }
};
```

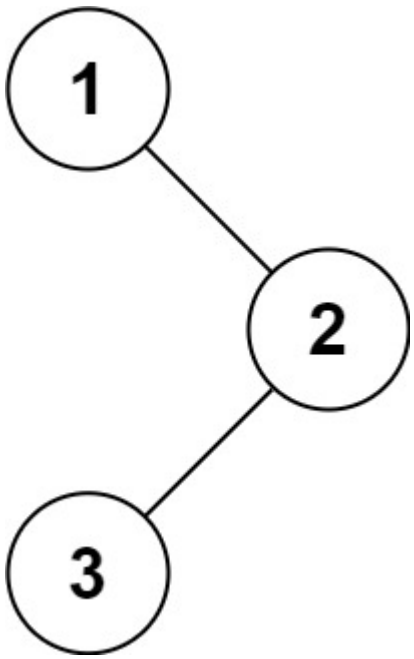
思路：这道题用栈来解决，遇到栈顶元素相同的就出栈，遇到不一样的就入栈。

---

144. 二叉树的前序遍历（94 题， 145题）中序和后序

给你二叉树的根节点 `root` ，返回它节点值的 **前序** 遍历。

示例 1：




```
输入：root = [1,null,2,3]
输出：[1,2,3]
```

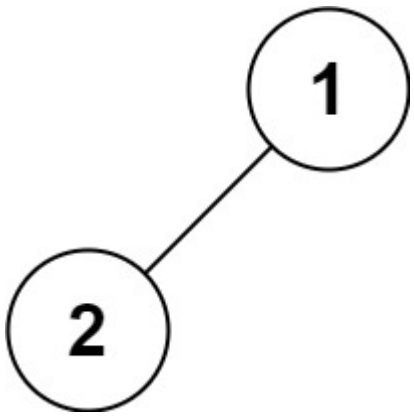
示例 2：

```
输入：root = []
输出：[]
```

示例 3：

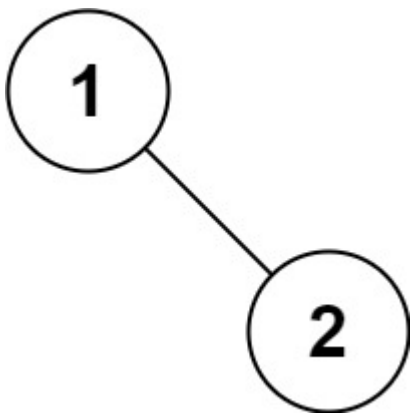
```
输入：root = [1]
输出：[1]
```

示例 4：




```
输入：root = [1,2]
输出：[1,2]
```

示例 5：




```
输入：root = [1,null,2]
输出：[1,2]
```

提示：

- 树中节点数目在范围 `[0, 100]` 内
- `-100 <= Node.val <= 100`

进阶：递归算法很简单，你可以通过迭代算法完成吗？

题解：　　（递归）144.cpp

```cpp
class Solution {
public:
    void traversal(TreeNode*  ruan, vector<int> &aws) {
        if (ruan == nullptr) {
            return;
```

```
        }
        aws.push_back(ruan->val);      //中
        traversal(ruan->left, aws);    //左
        traversal(ruan->right, aws);   //后
    }


    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> aws;
        traversal(root, aws);
        return aws;
    }
};
```

中序:

```
class Solution {
public:
    void traversal(TreeNode*  ruan, vector<int> &aws) {
        if (ruan == nullptr) {
            return;
        }
        traversal(ruan->left, aws);     //左
        aws.push_back(ruan->val);       //中
        traversal(ruan->right, aws);    //右
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> aws;
        traversal(root, aws);
        return aws;
    }
};
```

后序:

```
class Solution {
public:
    void traversal(TreeNode*  ruan, vector<int> &aws) {
        if (ruan == nullptr) {
            return;
        }
        traversal(ruan->left, aws);     //左
        traversal(ruan->right, aws);    //右
        aws.push_back(ruan->val);       //中
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> aws;
        traversal(root, aws);
        return aws;
    }
};
```

题解（非递归遍历） 用栈

先序遍历

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {            //前序遍历  中左右
        stack<TreeNode*> temp;    //用来存放指针的栈
        vector<int> aws;          //用来存放结果
        temp.push(root);
        while (!temp.empty())
        {
            TreeNode* ruan = temp.top();
            aws.push_back(temp.top()->val);
            temp.pop();       //存哪个就放哪个
            if (ruan->right) temp.push(ruan->right);
            if (ruan->left)  temp.push(ruan->left);
        }
        return aws;
    }
};
```

中序遍历 (94.cpp)

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {            //中序遍历  左中右
        stack<TreeNode*> temp;
        vector<int> result;
        //temp.push(root);
        TreeNode* ruan = root;     //用这个指针来从左往下遍历
        while (ruan!= nullptr||!temp.empty())
        {
            if (ruan!=nullptr) {          //只要ruan指针不为空
                temp.push(ruan);
                ruan = ruan->left;      //ruan就一直往下跑    //左
            }
            else
            {
                ruan = temp.top();
                result.push_back(temp.top()->val);        //存储左
                temp.pop();                               //弹出左
                ruan =ruan->right;                        //右一步
            }
        }
        return result;
    }
};
```

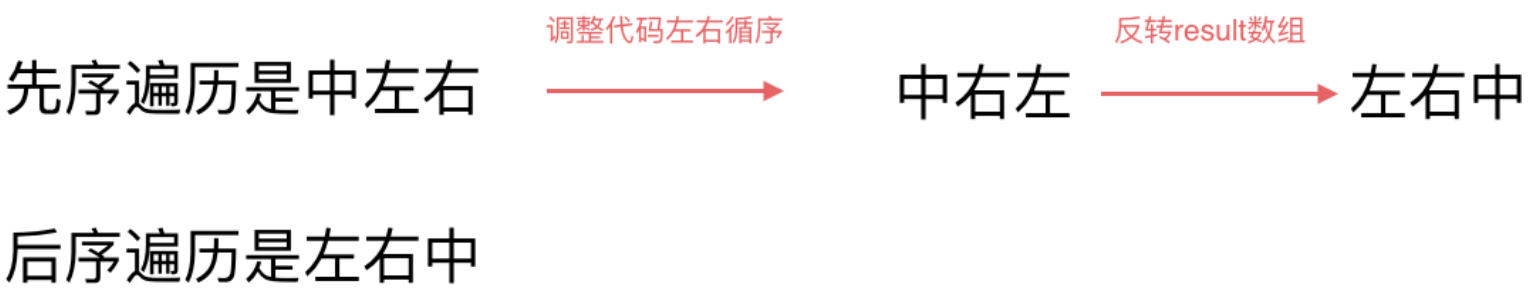
往右，如果不行就会弹出栈顶

后序遍历(左右中) 145.1.cpp

```
class Solution {
```

```
public:
    vector<int> inorderTraversal(TreeNode* root) {            //中序遍历  左中右
        stack<TreeNode*> temp;
        vector<int> result;
        //temp.push(root);
        TreeNode* ruan =root;     //用这个指针来从左往下遍历
        temp.push(ruan);
        while (!temp.empty())
        {
            ruan = temp.top();
            temp.pop();
            result.push_back(ruan->val);
            if(ruan->left) temp.push(ruan->left);
            if(ruan->right)temp.push(ruan->right);
        }
        reverse(result.begin(), result.end());
        return result;
    }
};
```

先序遍历是中左右，后序遍历是左右中，那么我们只需要调整一下先序遍历的代码顺序，就变成中右左的遍历顺序，然后在反转result数组，输出的结果顺序就是左右中了，如下图：

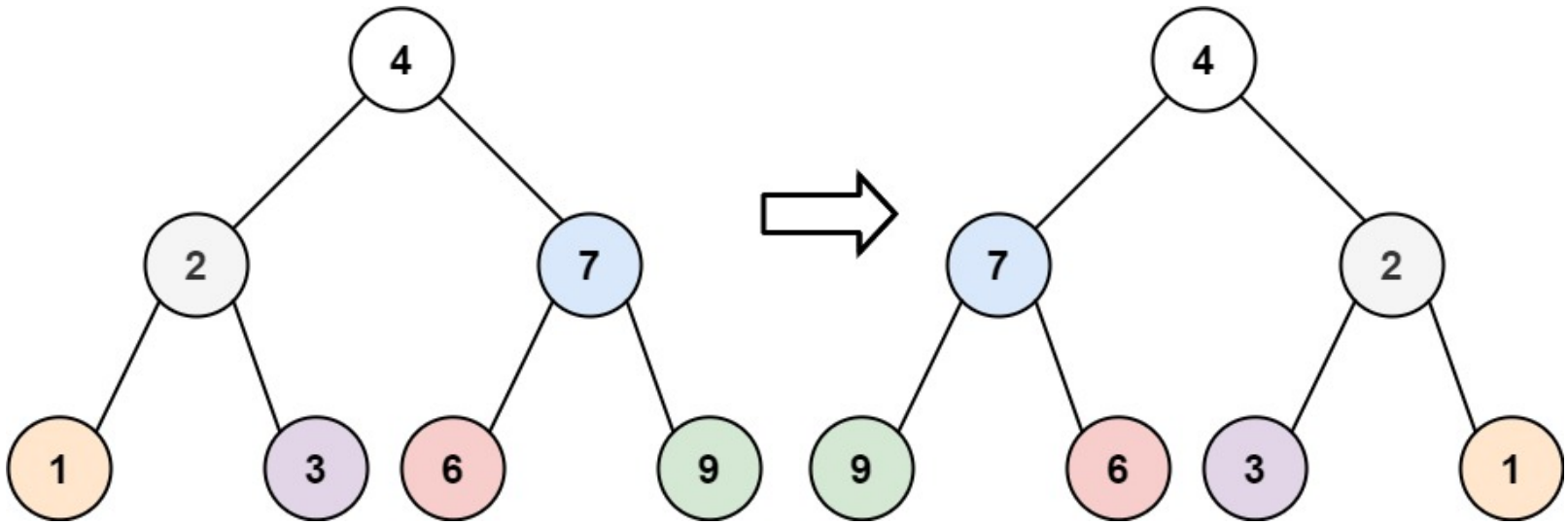


---

226. 翻转二叉树

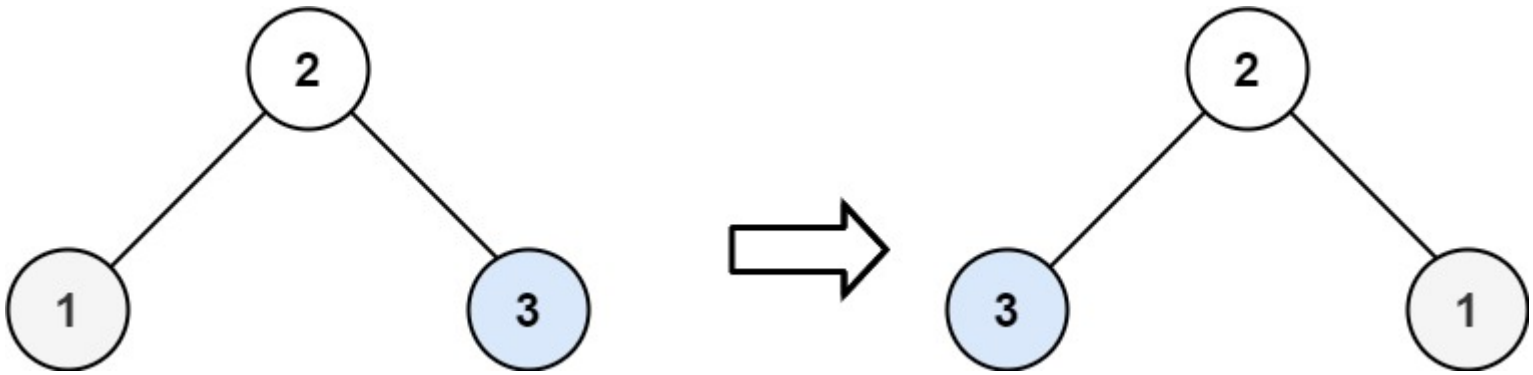
给你一棵二叉树的根节点 `root` ，翻转这棵二叉树，并返回其根节点。

示例 1：




```
输入：root = [4,2,7,1,3,6,9]
输出：[4,7,2,9,6,3,1]
```

示例 2：




```
输入：root = [2,1,3]
输出：[2,3,1]
```

示例 3：

```
输入：root = []
输出：[]
```

提示：

- 树中节点数目范围在 `[0, 100]` 内
- `-100 <= Node.val <= 100`

题解：(226.cpp)

```cpp
class Solution {
public:
    void invert(TreeNode* root) {
        TreeNode* temp = root;
        if (root->left || root ->right)
        {
            temp = root->left;
            root->left = root->right;
```

```
            root->right = temp;
        }
    }
    void invertTreeRuan(TreeNode* root) {
        TreeNode* temp = root;
        if (!temp) return;
        if (temp->left) invertTreeRuan(temp->left);
        if (temp->right) invertTreeRuan(temp->right);
        invert(temp);
    }
    TreeNode* invertTree(TreeNode* root) {
         invertTreeRuan(root);
        return root;
    }
};
```

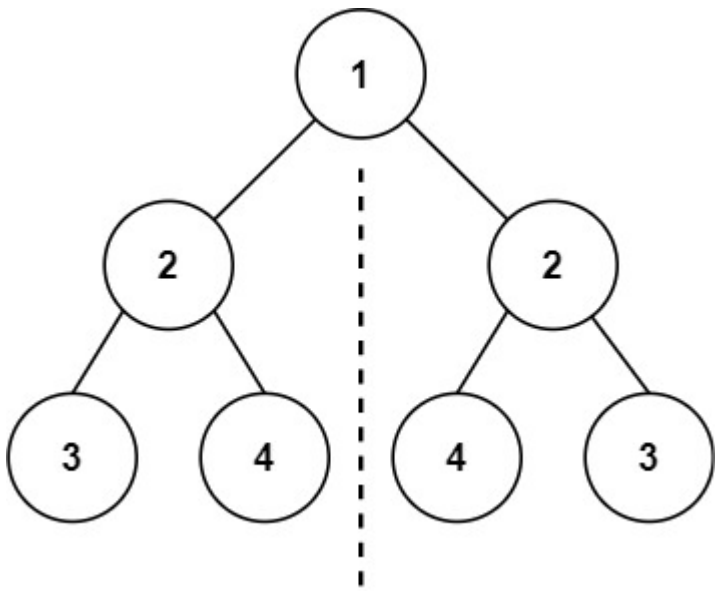
笔记：递归法(我的方法是前序递归)，别忘了 `if (!temp) return;` 不然会访问到空指针。

---

101. 对称二叉树

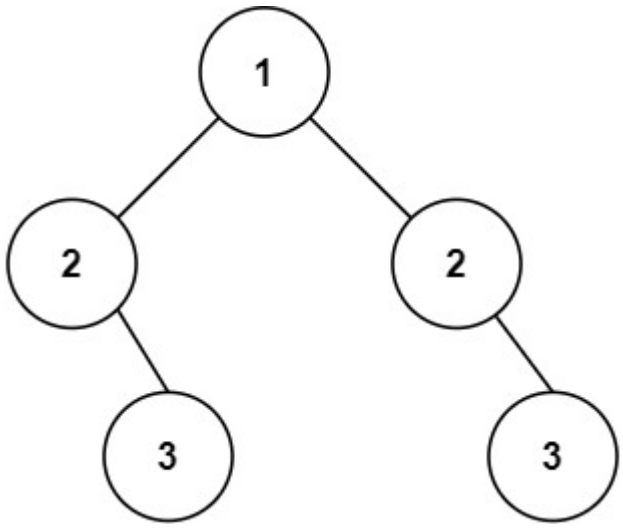
给你一个二叉树的根节点 `root` ， 检查它是否轴对称。

示例 1：




```
输入：root = [1,2,2,3,4,4,3]
输出：true
```

示例 2：




```
输入：root = [1,2,2,null,3,null,3]
输出：false
```

提示：

- 树中节点数目在范围 `[1, 1000]` 内
- `-100 <= Node.val <= 100`

进阶：你可以运用递归和迭代两种方法解决这个问题吗？(好问题)

题解（递归）：

题解(迭代法):

迭代法有两种解法：一种是栈，另一种是队列，这里使用队列的解法

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (root == NULL) return true;
        queue<TreeNode*> que;
        que.push(root->left);   // 将左子树头结点加入队列
        que.push(root->right);  // 将右子树头结点加入队列

        while (!que.empty()) {  // 接下来就要判断这两个树是否相互翻转
            TreeNode* leftNode = que.front(); que.pop();
            TreeNode* rightNode = que.front(); que.pop();
            if (!leftNode && !rightNode) {  // 左节点为空、右节点为空，此时说明是对称的
                continue;
            }

            // 左右一个节点不为空，或者都不为空但数值不相同，返回false
            if ((!leftNode || !rightNode || (leftNode->val != rightNode->val))) {
                return false;
            }
            que.push(leftNode->left);   // 加入左节点左孩子
            que.push(rightNode->right); // 加入右节点右孩子
            que.push(leftNode->right);  // 加入左节点右孩子
```

```
            que.push(rightNode->left);  // 加入右节点左孩子
        }
        return true;
    }
};
```

我的笔记：

---

104. 二叉树的最大深度

给定一个二叉树 `root` ，返回其最大深度。

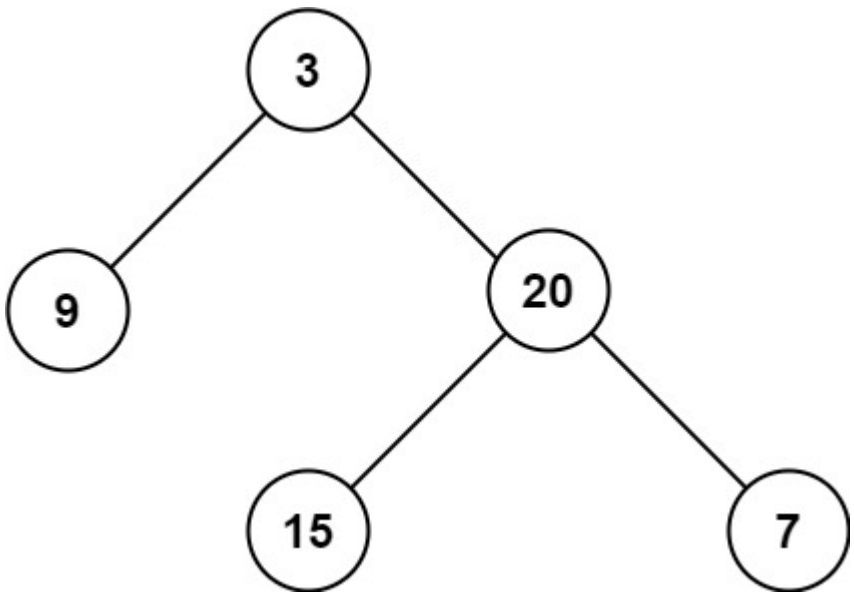
> **二叉树的深度**

二叉树的 **最大深度** 是指从根节点到最远叶子节点的最长路径上的节点数。

> **二叉树的高度**

二叉树的高度是二叉树中任意一个节点，到叶子节点的距离。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：3
```

示例 2：

```
输入：root = [1,null,2]
输出：2
```

提示：

- 树中节点的数量在 `[0, 104]` 区间内。

- `-100 <= Node.val <= 100`

题解：

这是递归的解法，通过

```
class Solution {
public:
    void getMaxDepth(TreeNode* root , int coutNum) {
        if (!root->left &&!root->right)
        {
            if (coutNum > this->maxDepthNum) this->maxDepthNum = coutNum;
            coutNum -= 1;
            return;
        }
        coutNum++;
        if(root->right) getMaxDepth(root->right, coutNum);

        if(root->left) getMaxDepth(root->left, coutNum);

    }

    int maxDepth(TreeNode* root) {
        if (!root) return 0;
        this->maxDepthNum = 1;
        int coutNum = 1;
        getMaxDepth(root, coutNum);
        return this->maxDepthNum;
    }

public:
    int maxDepthNum;
};
```

第二种方法是迭代法，就是层序遍历的改版，每一次加一层，计数器加一。

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (!root) return 0;
        deque<TreeNode*> ruanDeque;
        ruanDeque.push_back(root);
        int sizeRuan = 1;
        int coutDepth = 0;
        while (!ruanDeque.empty())
        {
            sizeRuan = ruanDeque.size();
            coutDepth++;     //记录深度
            for (size_t i = 0; i < sizeRuan; i++)
            {
                TreeNode* ruan = ruanDeque.front();
                ruanDeque.pop_front();
                if (ruan->left) ruanDeque.push_back(ruan->left);
```

```
                if (ruan->right) ruanDeque.push_back(ruan->right);
            }
        }
        return coutDepth;
    }
};
```

我的笔记：这道题就是帮助你回忆，二叉树的递归遍历和层序遍历。

---

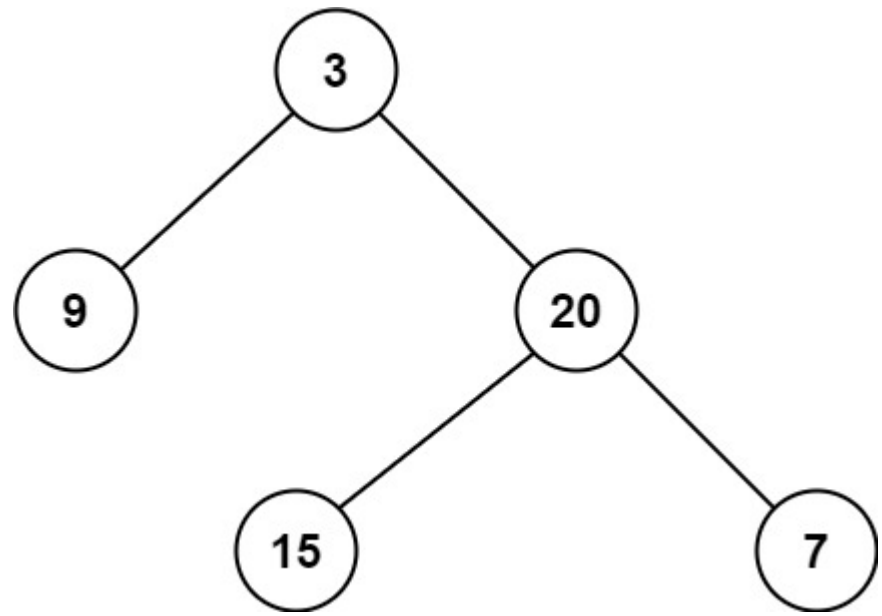
111. 二叉树的最小深度

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：2
```

示例 2：

```
输入：root = [2,null,3,null,4,null,5,null,6]
输出：5
```

提示：

- 树中节点数的范围在 `[0, 105]` 内
- `-1000 <= Node.val <= 1000`

题解：

```
class Solution {
public:
    void getShortDepth(TreeNode* root , int coutNum) {
        if (!root->left &&!root->right)
        {
            cout << "\n到" << root->val << "了,coutNum=" << coutNum;
            if (coutNum < this->maxDepthNum) this->maxDepthNum = coutNum;   //把这里判断大小的改了
            coutNum -= 1;
            return;
        }
        coutNum++;
        if(root->right) getShortDepth(root->right, coutNum);

        if(root->left) getShortDepth(root->left, coutNum);

    }

    int minDepth(TreeNode* root) {
        if (!root) return 0;
        this->maxDepthNum = 10000;    //把这里的初值改了
        int coutNum = 1;
        getShortDepth(root, coutNum);
        return this->maxDepthNum;
    }

public:
    int maxDepthNum;
};
```

我的笔记：就是把求最大深度的改一改，在做过上一道题的基础上这道题就是简单题。

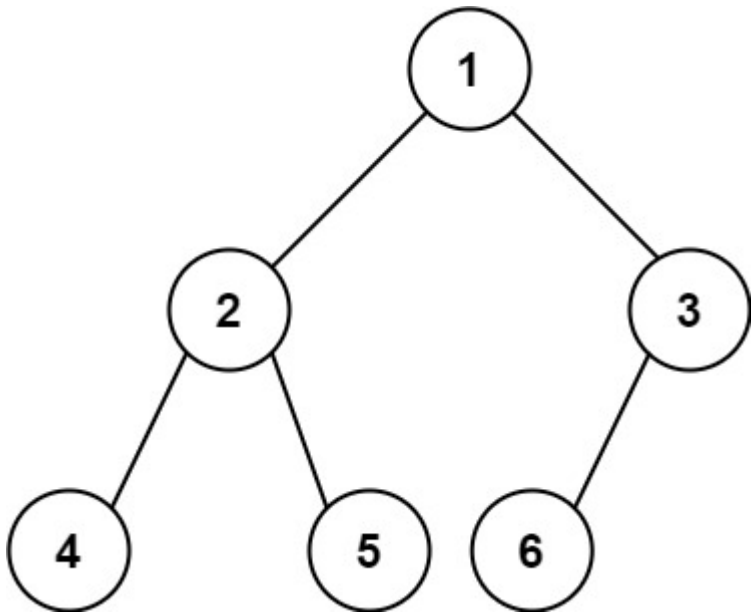
---

222. 完全二叉树的节点个数

给你一棵 **完全二叉树** 的根节点 `root` ，求出该树的节点个数。

完全二叉树 的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 `h` 层，则该层包含 `1~ 2h` 个节点。

示例 1：




```
输入：root = [1,2,3,4,5,6]
输出：6
```

示例 2：

```
输入：root = []
输出：0
```

示例 3：

```
输入：root = [1]
输出：1
```

提示：

- 树中节点的数目范围是`[0, 5 * 104]`
- `0 <= Node.val <= 5 * 104`
- 题目数据保证输入的树是 **完全二叉树**

进阶：遍历树来统计节点是一种时间复杂度为 `O(n)` 的简单解决方案。你可以设计一个更快的算法吗？

题解：

这个办法比较直接，就是把求最大深度改了改，加上了一个计数器

```
class Solution {
public:
    void findDepth(TreeNode* root, int depth) {
        if (!root->left && !root->right) {
            if (this->maxDepth < depth) this->maxDepth = depth;
            if (depth >= this->maxDepth) this->ruanCouter++;    //最后一行，触发计数器++
            depth--;
            return;
        }
        depth++;
        if (root->left) findDepth(root->left, depth);
```

```
        if (root->right) findDepth(root->right, depth);
        return ;
    }
    int countNodes(TreeNode* root) {
        if (!root) return 0;
        int depth = 1;
        int aws = 0;
        this->maxDepth = 0;
        this->ruanCouter = 0;
        findDepth(root, depth);
        for (int i = 0; i < this->maxDepth - 1; i++) {
            aws +=pow(2,i);                 //前面几行满二叉树的节点数量和
        }
        aws += this->ruanCouter;            //加上最后一行的节点的数量
        return aws;
    }
public:
    int maxDepth;
    int ruanCouter;    //统计最后一行的个数
};
```

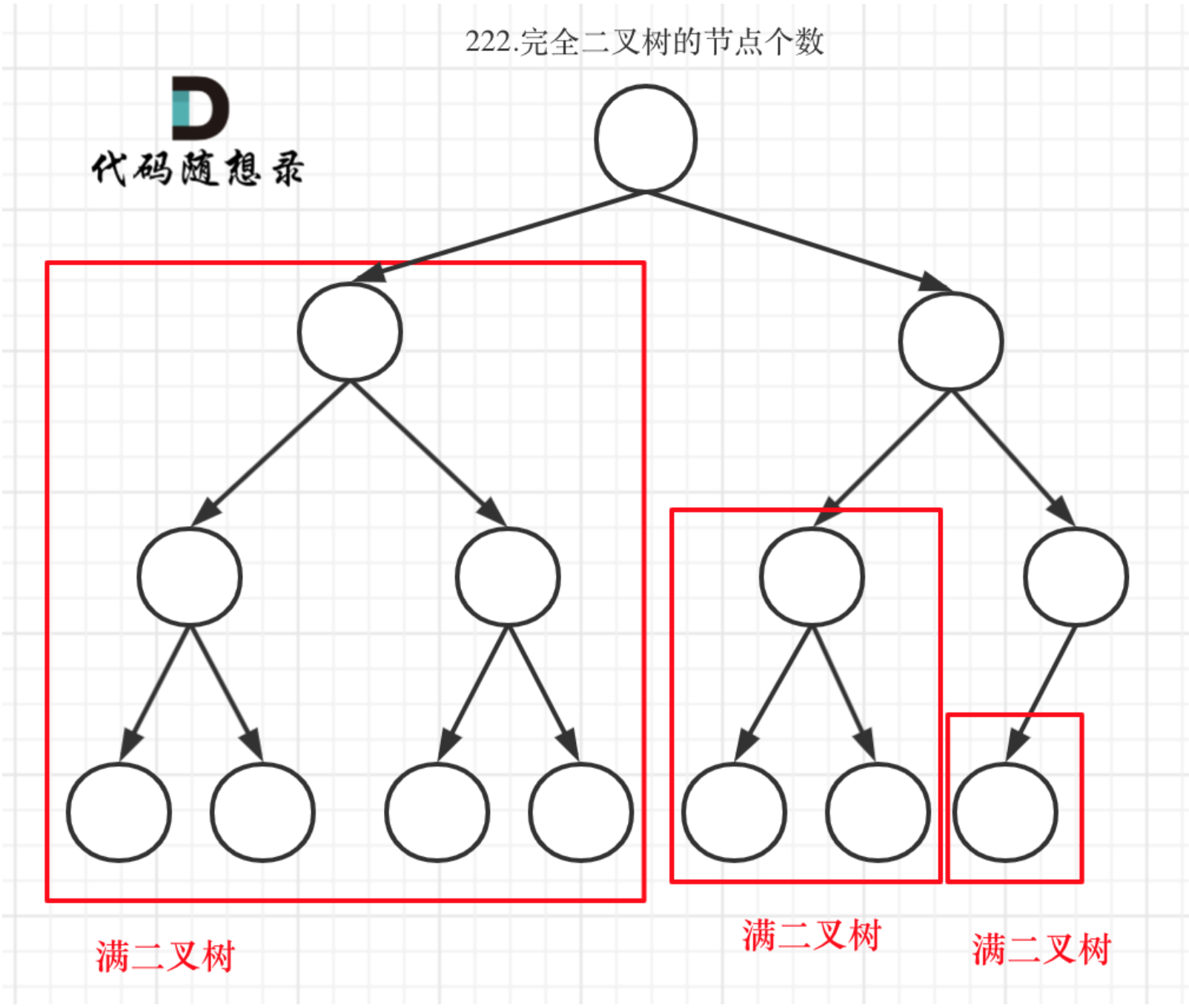
题解：

完全二叉树只有两种情况，情况一：就是满二叉树，情况二：最后一层叶子节点没有满。

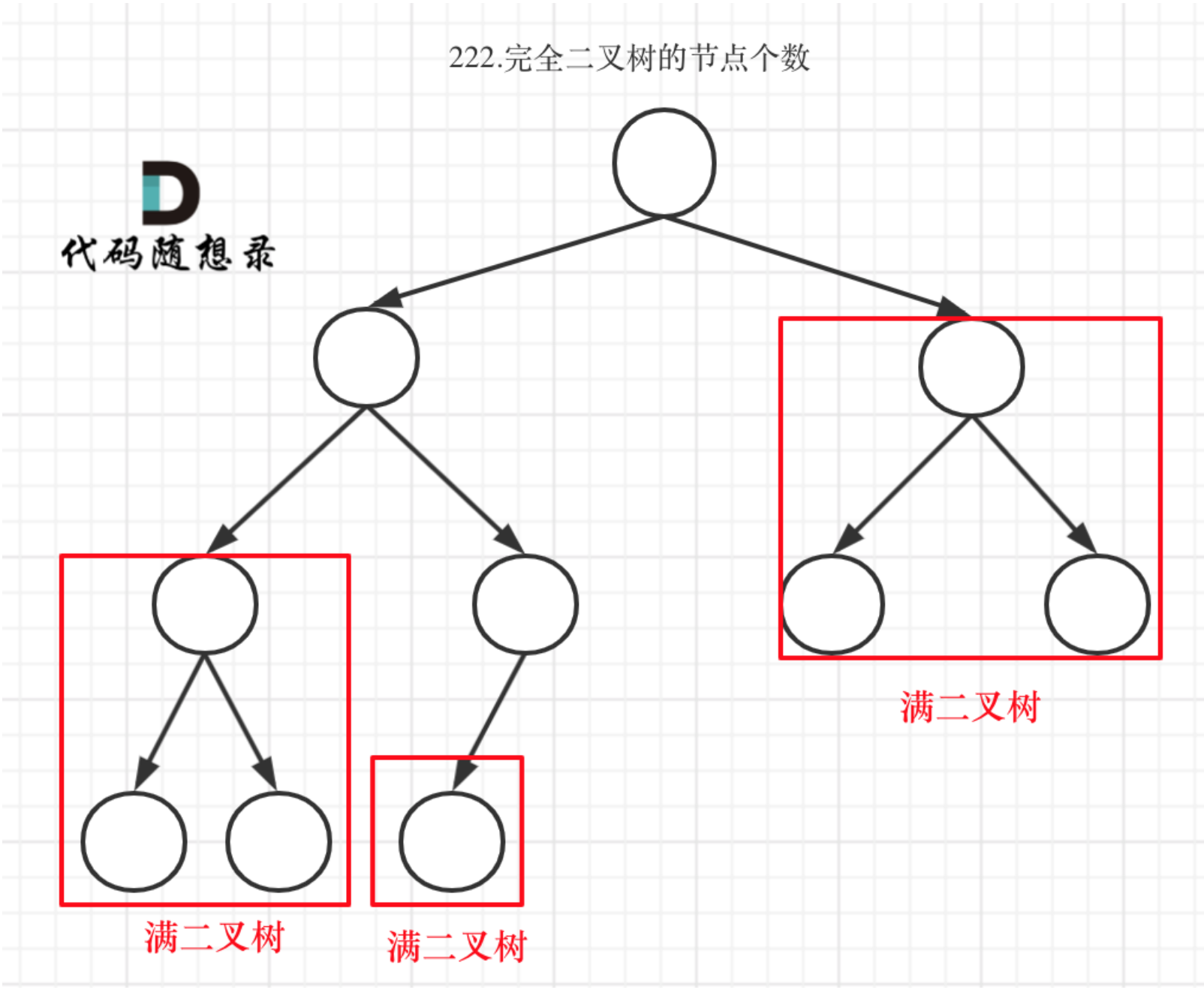
对于情况一，可以直接用 2^树深度 - 1 来计算，注意这里根节点深度为1。

对于情况二，分别递归左孩子，和右孩子，递归到某一深度一定会有左孩子或者右孩子为满二叉树，然后依然可以按照情况1来计算。

完全二叉树（一）如图：



完全二叉树（二）如图：



可以看出如果整个树不是满二叉树，就递归其左右孩子，直到遇到满二叉树为止，用公式计算这个子树（满二叉树）的节点数量。

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        if (root == nullptr) return 0;
        TreeNode* left = root->left;
        TreeNode* right = root->right;
        int leftDepth = 0, rightDepth = 0; // 这里初始为0是有目的的，为了下面求指数方便
        while (left) {  // 求左子树深度
            left = left->left;
            leftDepth++;
        }
        while (right) { // 求右子树深度
            right = right->right;
            rightDepth++;
        }
        if (leftDepth == rightDepth) {
```

```
            return (2 << leftDepth) - 1; // 注意(2<<1) 相当于2^2，所以leftDepth初始为0
        }
        return countNodes(root->left) + countNodes(root->right) + 1;
    }
};
```

> 这里的 `2 << leftDepth` 的含义解读

`2 << leftDepth` 是一个位运算表达式，表示将数字 2 的二进制表示向左移动 `leftDepth` 位。

位运算符 `<<` 是位移运算符，用于将一个数的二进制表示向左或向右移动指定的位数。向左移动意味着在数的二进制表示的右边添加零位，这相当于将原数乘以 2 的幂。

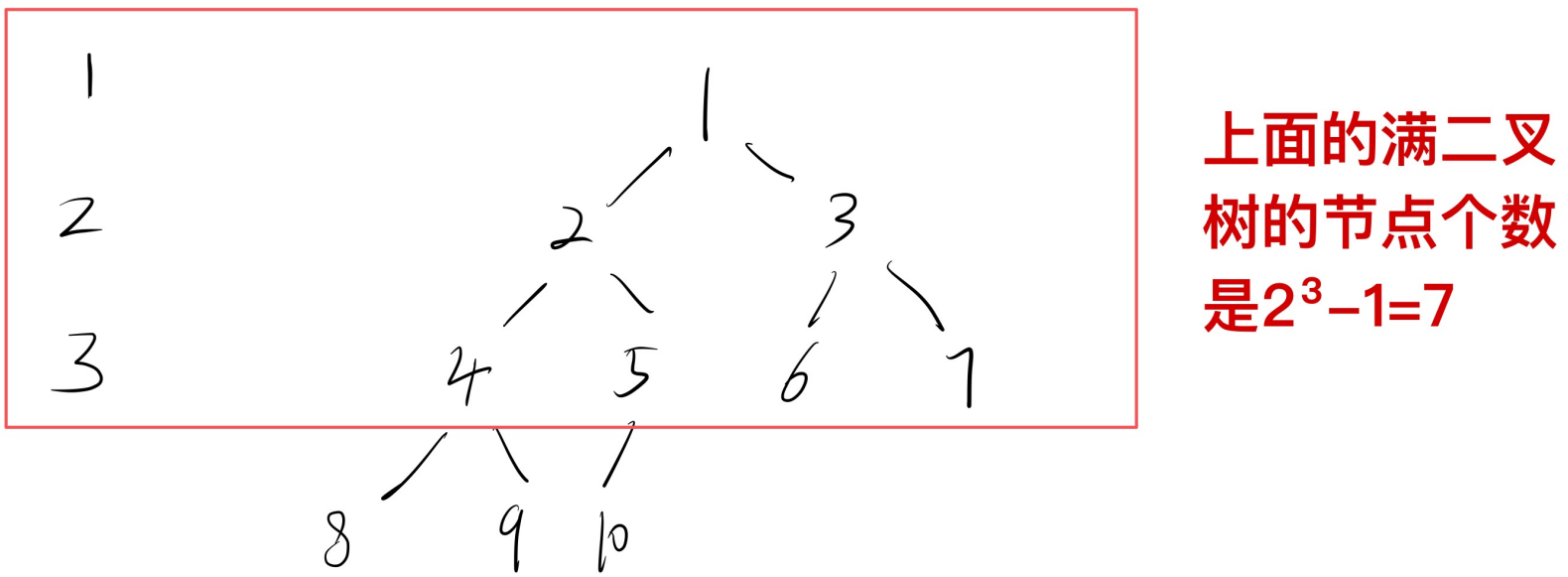
具体来说，`2 << leftDepth` 的含义是：

- 将数字 2 的二进制表示（即 `10`）向左移动 `leftDepth` 位。
- 每向左移动一位，相当于将 2 乘以 2（即倍增）。
- 因此，`2 << leftDepth` 相当于计算 2(leftDepth+1)2(leftDepth+1)，因为 2 的二进制表示是 `10`，向左移动一位变成 `100`（即 4），向左移动两位变成 `1000`（即 8），依此类推。

例如：

- `2 << 0` 的结果是 `2`（二进制 `10` 变成 `10`）。
- `2 << 1` 的结果是 `4`（二进制 `10` 变成 `100`）。
- `2 << 2` 的结果是 `8`（二进制 `10` 变成 `1000`）。

在这里的`(2 << leftDepth) - 1;`就是计算$2^n - 1$，例如下图

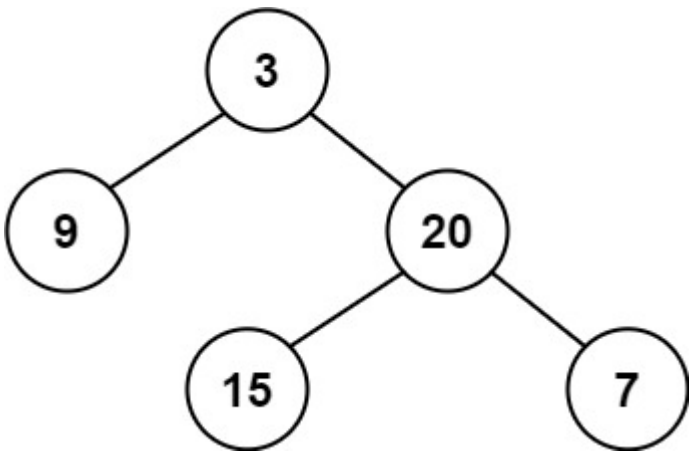


110. 平衡二叉树

给定一个二叉树，判断它是否是 平衡二叉树

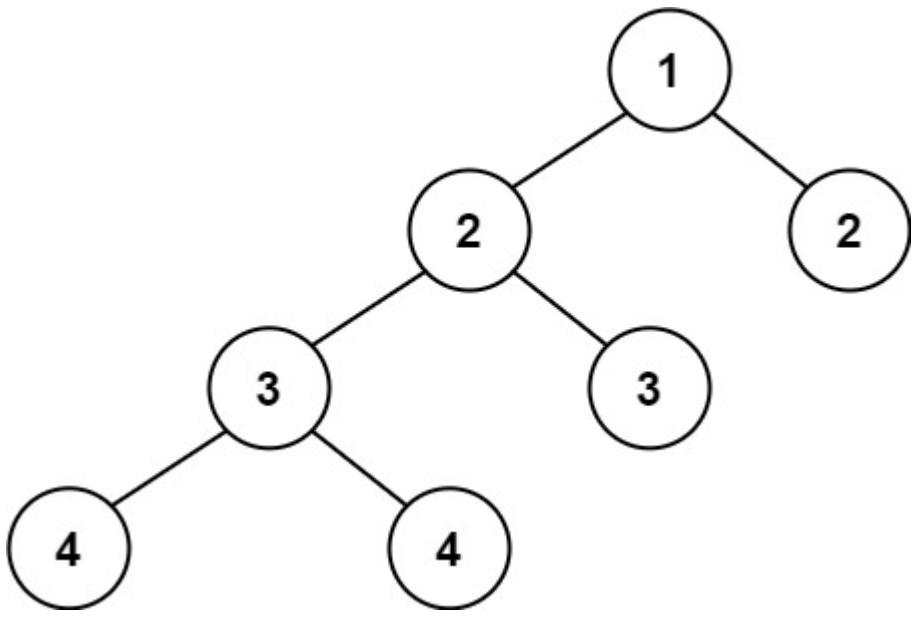
> **平衡二叉树** 是指该树所有节点的左右子树的深度相差不超过 1。
>
> 【平衡二叉树(AVL树)】 https://www.bilibili.com/video/BV1tZ421q72h/?share_source=copy_web&vd_source=6715ab9baaaef26ce4b66bf7ce958418

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：true
```

示例 2：




```
输入：root = [1,2,2,3,3,null,null,4,4]
输出：false
```

示例 3：

```
输入：root = []
输出：true
```

提示：

- 树中的节点数在范围 `[0, 5000]` 内
- `-104 <= Node.val <= 104`

题解：

递归

```
class Solution {
public:
    // 返回以该节点为根节点的二叉树的高度，如果不是平衡二叉树了则返回-1
    int getHeight(TreeNode* node) {
        if (node == NULL) {
            return 0;
        }
        int leftHeight = getHeight(node->left);
        if (leftHeight == -1) return -1;
        int rightHeight = getHeight(node->right);
        if (rightHeight == -1) return -1;
        return abs(leftHeight - rightHeight) > 1 ? -1 : 1 + max(leftHeight, rightHeight);
    }
    bool isBalanced(TreeNode* root) {
        return getHeight(root) == -1 ? false : true;
    }
};
```

迭代：

```
class Solution {
private:
    int getDepth(TreeNode* cur) {
        stack<TreeNode*> st;
        if (cur != NULL) st.push(cur);
        int depth = 0; // 记录深度
        int result = 0;
        while (!st.empty()) {
            TreeNode* node = st.top();
            if (node != NULL) {
                st.pop();
                st.push(node);                          // 中
                st.push(NULL);
                depth++;
                if (node->right) st.push(node->right);  // 右
                if (node->left) st.push(node->left);    // 左

            } else {
                st.pop();
                node = st.top();
                st.pop();
                depth--;
            }
            result = result > depth ? result : depth;
        }
        return result;
    }

public:
    bool isBalanced(TreeNode* root) {
        stack<TreeNode*> st;
```

```
        if (root == NULL) return true;
        st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();                      // 中
            st.pop();
            if (abs(getDepth(node->left) - getDepth(node->right)) > 1) {
                return false;
            }
            if (node->right) st.push(node->right);          // 右（空节点不入栈）
            if (node->left) st.push(node->left);            // 左（空节点不入栈）
        }
        return true;
    }
};
```

当然此题用迭代法，其实效率很低，因为没有很好的模拟回溯的过程，所以迭代法有很多重复的计算。

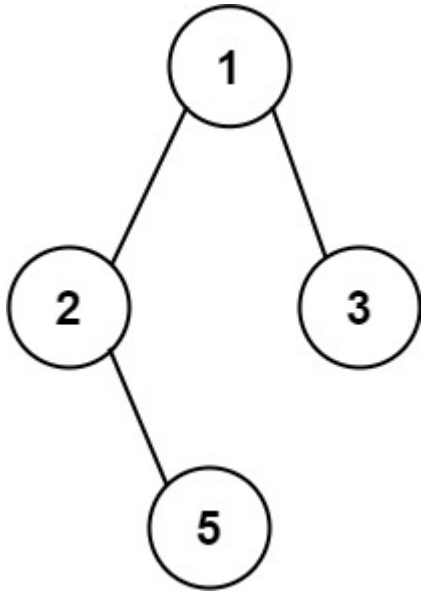
虽然理论上所有的递归都可以用迭代来实现，但是有的场景难度可能比较大。

257. 二叉树的所有路径

给你一个二叉树的根节点 `root` ，按 **任意顺序** ，返回所有从根节点到叶子节点的路径。

叶子节点 是指没有子节点的节点。

示例 1：




```
输入：root = [1,2,3,null,5]
输出：["1->2->5","1->3"]
```

示例 2：

```
输入：root = [1]
输出：["1"]
```

提示：

- 树中节点的数目在范围 `[1, 100]` 内
- `-100 <= Node.val <= 100`

题解：

这个算法的核心是使用栈（`stack`）来实现广度优先搜索（BFS）的逻辑，从而找到二叉树中从根节点到所有叶子节点的路径。以下是对算法的点评和总结：

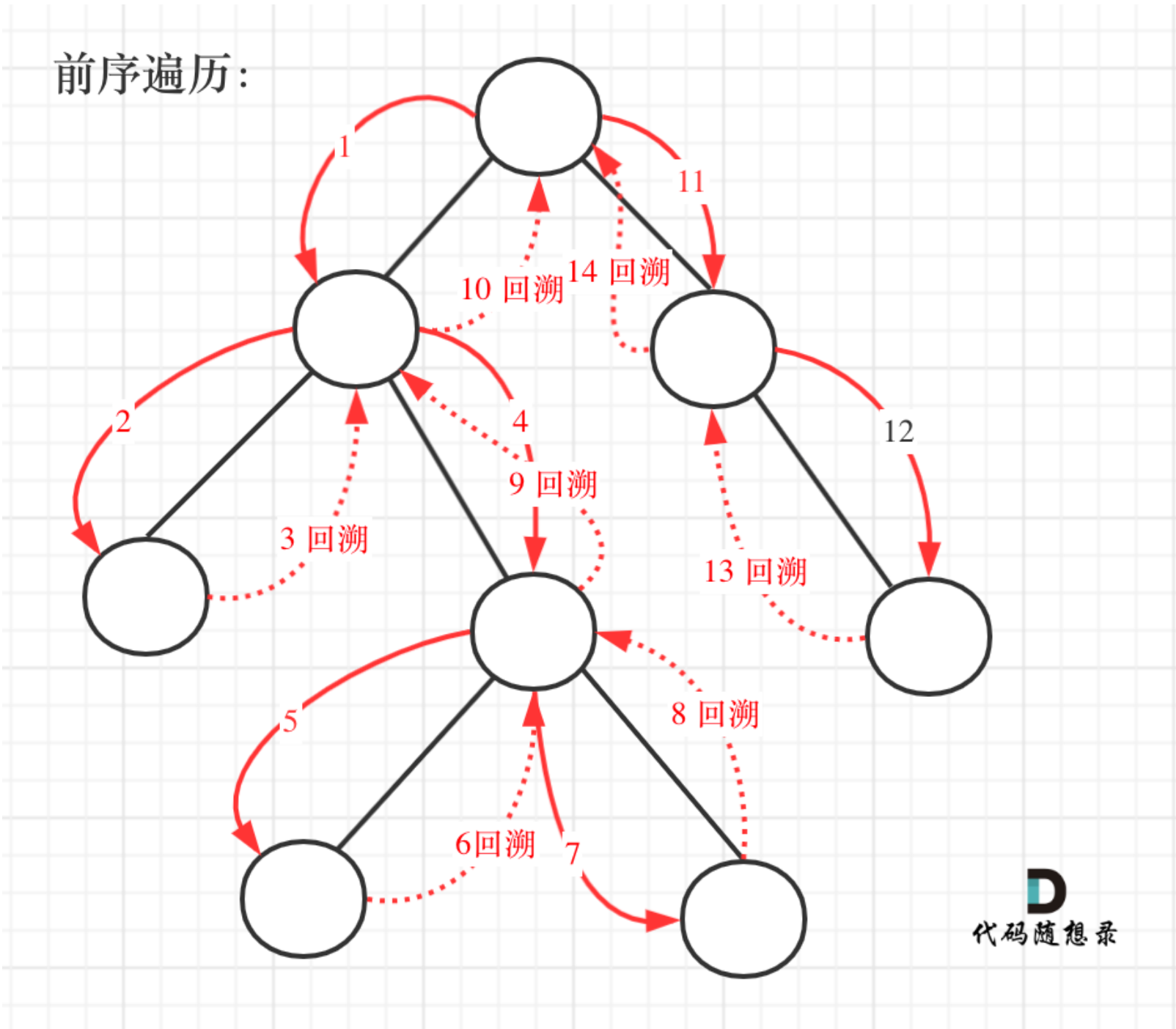
```
vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> paths;
    if (!root) return paths;
    stack<pair<TreeNode*, string>> nodeStack;
    nodeStack.push({ root, to_string(root->val) });
    while (!nodeStack.empty()) {
        pair<TreeNode*, string> top = nodeStack.top();
        TreeNode* node = top.first;  // 从 pair 中手动提取 node
        string path = top.second;    // 从 pair 中手动提取 path
        nodeStack.pop();
        // 如果是叶子节点(也就是左右都为空的情况)，将路径保存
        if (!node->left && !node->right) {
            paths.push_back(path);
        }
        // 先压右子树，再压左子树，以保证左子树先遍历
        if (node->right) {
            nodeStack.push({ node->right, path + "->" + to_string(node->right->val) });
        }
        if (node->left) {
            nodeStack.push({ node->left, path + "->" + to_string(node->left->val) });
        }
    }
    return paths;
}
```

> 递归解法

```
class Solution {
public:
    void digui(TreeNode* root,string path) {
        if (!root) return;
        if (!root->left && !root->right) {
            aws.push_back(path+=to_string(root->val));
            return;
        }
        if (root->left) {
            digui(root->left, path + to_string(root->val) + "->");
        }
        if (root->right) {
            digui(root->right, path + to_string(root->val) + "->");
        }
    }
    vector<string> binaryTreePaths(TreeNode* root) {
        if (!root) return this->aws;
        digui(root, "");
        return this->aws;
    }
public:
    vector<string> aws;
```

```
};
```

前序遍历以及回溯的过程如图：



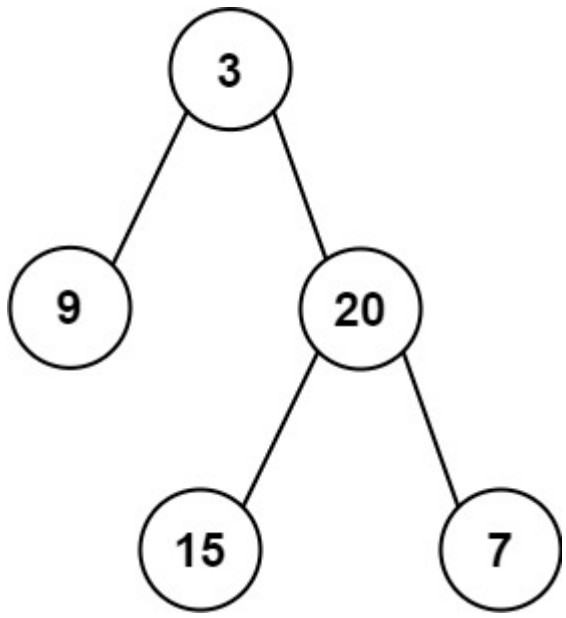
我的笔记：简单题才是难题

---

404. 左叶子之和

给定二叉树的根节点 `root` ，返回所有左叶子之和。

示例 1：




```
输入: root = [3,9,20,null,null,15,7]
输出: 24
解释: 在这个二叉树中，有两个左叶子，分别是 9 和 15，所以返回 24
```

示例 2:

```
输入: root = [1]
输出: 0
```

提示:

- 节点数在 `[1, 1000]` 范围内
- `-1000 <= Node.val <= 1000`

题解:

```
class Solution {
public:
    int sumOfLeftLeaves(TreeNode* root) {
        if (!root) return 0;
        int sum = 0;
        stack<TreeNode*> ruanStack;
        ruanStack.push(root);
        while (!ruanStack.empty())
        {
            TreeNode* temp = ruanStack.top();
            ruanStack.pop();
            if (temp->left) {
                if (!temp->left->left && !temp->left->right) {
                    sum += temp->left->val;
                }
                ruanStack.push(temp->left);
            }
            if (temp->right) {
                ruanStack.push(temp->right);
            }
        }
```

```
        return sum;
    }
};
```

我的笔记：很简答的题，就是在遍历的时候加上一个判断，用一个数目来保存符合条件的数的总和，只要把左叶子节点统计出来，就可以了

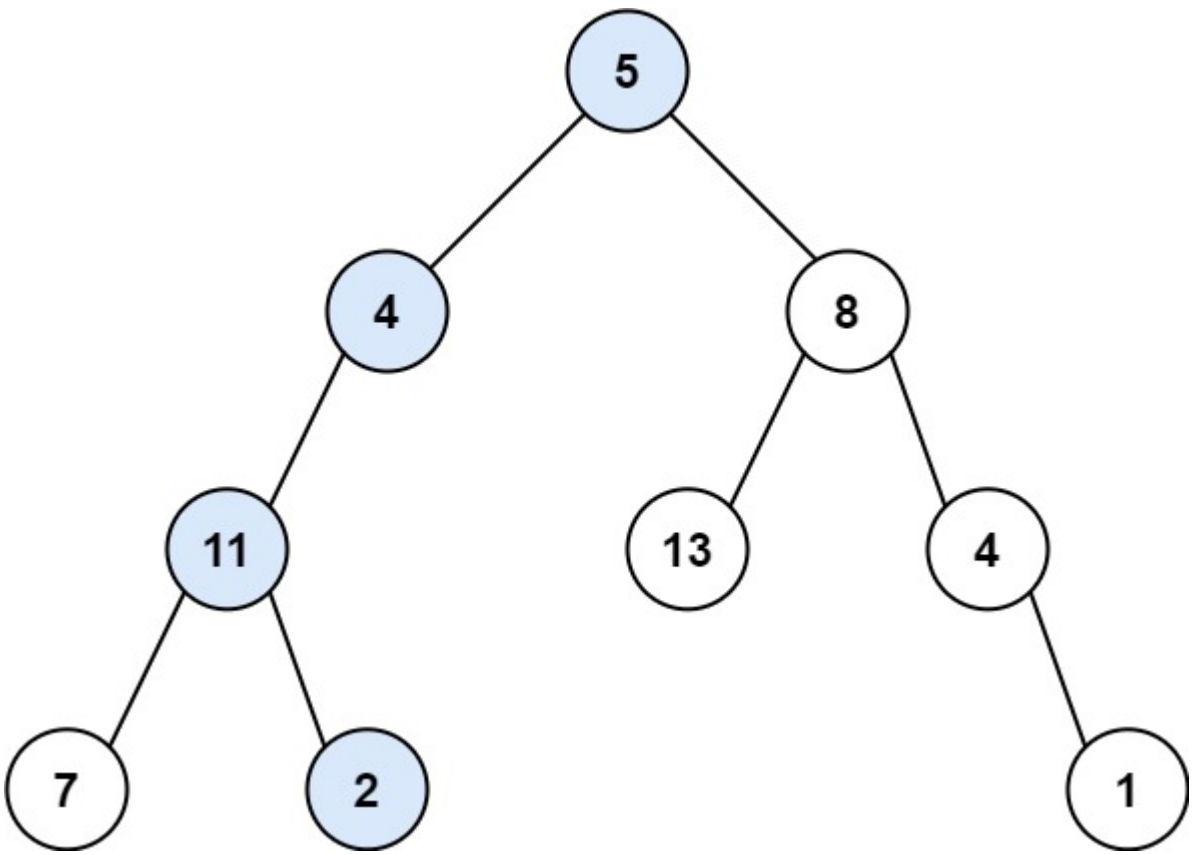
---

112. 路径总和

给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum` 。判断该树中是否存在 **根节点到叶子节点** 的路径，这条路径上所有节点值相加等于目标和 `targetSum` 。如果存在，返回 `true` ；否则，返回 `false` 。

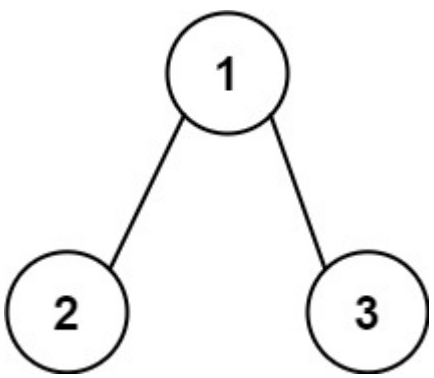
叶子节点 是指没有子节点的节点。

示例 1：




```
输入：root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
输出：true
解释：等于目标和的根节点到叶节点路径如上图所示。
```

示例 2：



```
输入：root = [1,2,3], targetSum = 5
输出：false
解释：树中存在两条根节点到叶子节点的路径：
(1 --> 2): 和为 3
(1 --> 3): 和为 4
不存在 sum = 5 的根节点到叶子节点的路径。
```

示例 3：

```
输入：root = [], targetSum = 0
输出：false
解释：由于树是空的，所以不存在根节点到叶子节点的路径。
```

提示：

- 树中节点的数目在范围 `[0, 5000]` 内
- `-1000 <= Node.val <= 1000`
- `-1000 <= targetSum <= 1000`

题解：

```cpp
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        stack<pair<TreeNode*, int>> ruanStack;
        //if (!root)  return (targetSum == 0)? true : false;
        if (!root) return false;
        ruanStack.push(make_pair(root,0));
        while (!ruanStack.empty())
        {
            pair<TreeNode*, int> top = ruanStack.top();
            ruanStack.pop();
            TreeNode* topNode = top.first;
            int NowSum = top.second;
            if (!topNode->left && !topNode->right) {                    //发现叶子节点，进行判断
                if (NowSum+topNode->val == targetSum)    return true;
            }
            if (topNode->right) {
                ruanStack.push(make_pair(topNode->right, NowSum + topNode->val));
            }
            if (topNode->left) {
                ruanStack.push(make_pair(topNode->left, NowSum + topNode->val));
            }
        }
        return false;
    }
};
```

我的笔记：在熟练掌握了之前257题目对二叉树的路径遍历的题目后，这种题就只是加上了一个值去统计而已

617. 合并二叉树

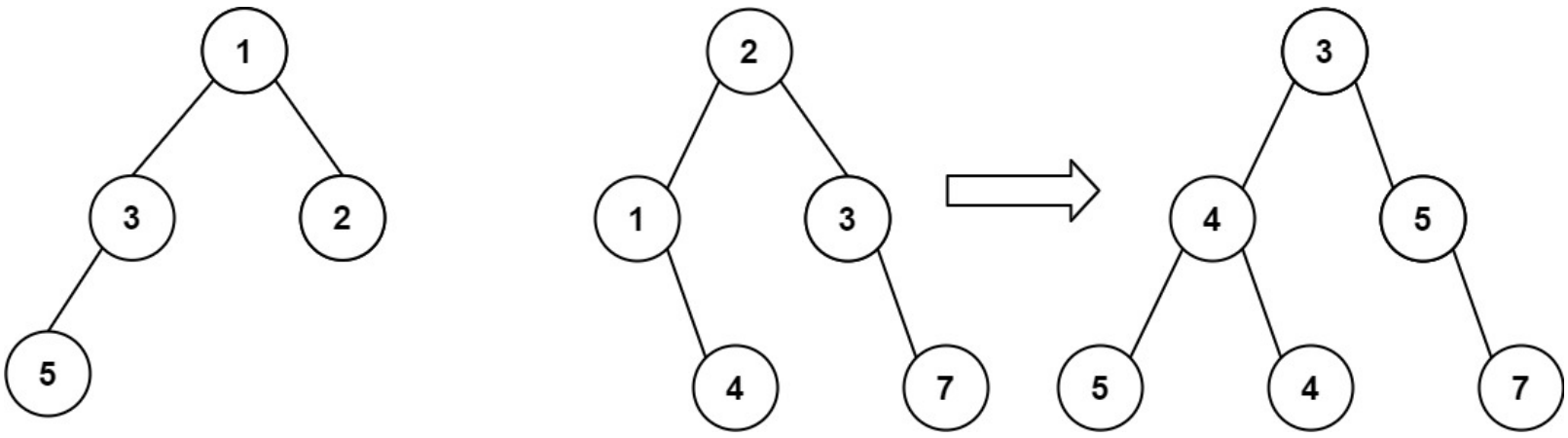
给你两棵二叉树： `root1` 和 `root2` 。

想象一下，当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠（而另一些不会）。你需要将这两棵树合并成一棵新二叉树。合并的规则是：如果两个节点重叠，那么将这两个节点的值相加作为合并后节点的新值；否则，**不为** null 的节点将直接作为新二叉树的节点。

返回合并后的二叉树。

注意: 合并过程必须从两个树的根节点开始。

示例 1:




```
输入：root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]
输出：[3,4,5,5,4,null,7]
```

示例 2:

```
输入：root1 = [1], root2 = [1,2]
输出：[2,2]
```

提示:

- 两棵树中的节点数目在范围 `[0, 2000]` 内
- `-104 <= Node.val <= 104`

题解:

```cpp
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        stack<pair<TreeNode*, TreeNode*>> nodeStack;
        if (!root1 && !root2) return nullptr;
        if (!root1) return root2;
        if (!root2) return root1;
        nodeStack.push({ root1,root2 });
        while (!nodeStack.empty())
        {
            pair<TreeNode*, TreeNode*> tempPair = nodeStack.top();
            nodeStack.pop();
            TreeNode* ruan1 = tempPair.first;
            TreeNode* ruan2 = tempPair.second;
```

```
            ruan1->val += ruan2->val;
            //只有当两棵树都是有值的时候进行相加，并且还需要继续向下遍历
            if (ruan1->right && ruan2->right) {
                nodeStack.push(make_pair( ruan1->right,ruan2->right));
            }
            else    //如果仅仅是root2对应的树有值，那就拼接上root2的分枝
            {
                if (!ruan1->right && ruan2->right) {
                    ruan1->right = ruan2->right;
                }
            }
            if (ruan1->left && ruan2->left) {
                nodeStack.push(make_pair(ruan1->left, ruan2->left));
            }
            else
            {
                if (!ruan1->left && ruan2->left) {
                    ruan1->left = ruan2->left;
                }
            }
        }
        return root1;
    }
};
```

我的笔记：熟悉二叉树遍历，这道题不难。

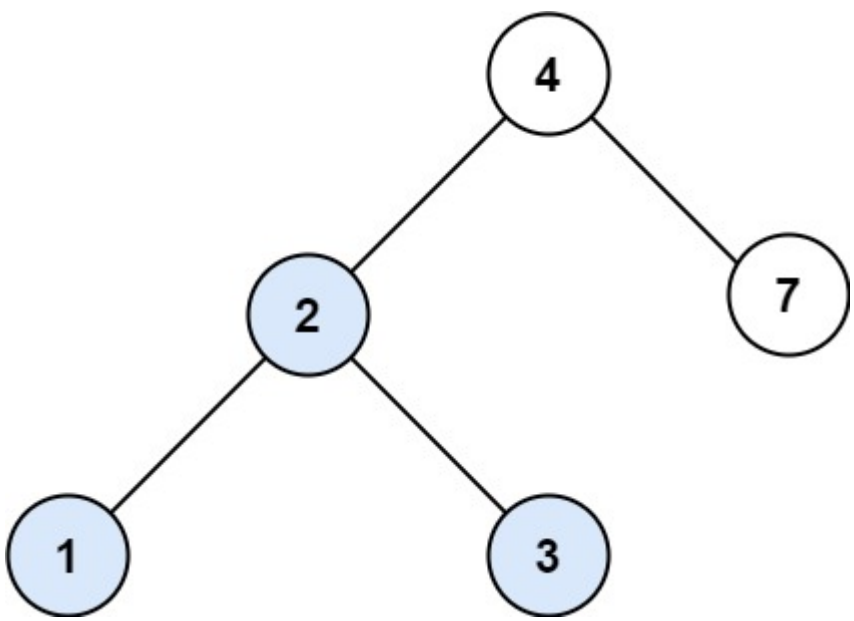
---

700. 二叉搜索树中的搜索

给定二叉搜索树（BST）的根节点 `root` 和一个整数值 `val`。

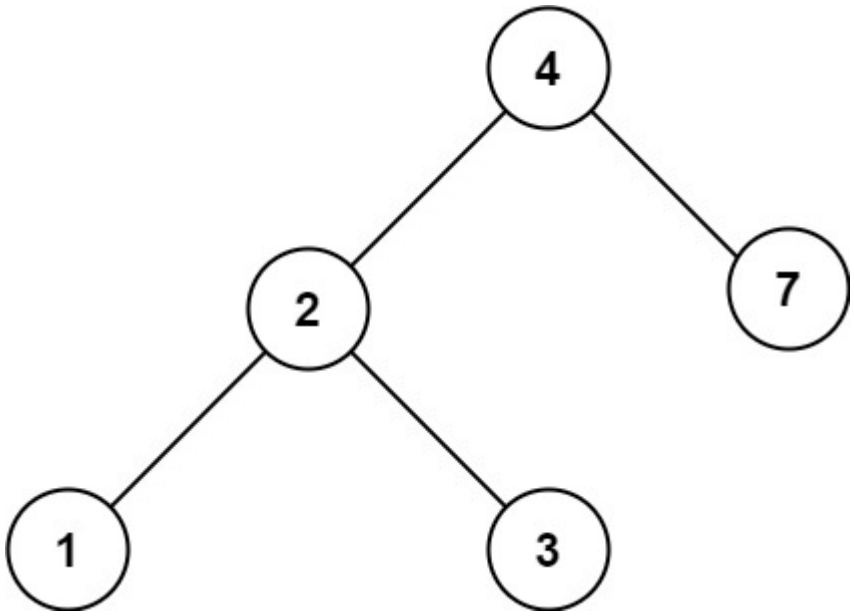
你需要在 BST 中找到节点值等于 `val` 的节点。 返回以该节点为根的子树。 如果节点不存在，则返回 `null` 。

示例 1:




```
输入：root = [4,2,7,1,3], val = 2
输出：[2,1,3]
```

示例 2:




```
输入：root = [4,2,7,1,3], val = 5
输出：[]
```

提示：

- 树中节点数在 `[1, 5000]` 范围内
- `1 <= Node.val <= 107`
- `root` 是二叉搜索树
- `1 <= val <= 107`

题解：

```cpp
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        TreeNode* ruan = root;
        while (ruan)
        {
            if (ruan->val == val) return ruan;
            if (ruan->val > val) { ruan = ruan->left; }
            else ruan = ruan->right;
        }
        return NULL;
    }
};
```

我的笔记：没啥看头，很简单，就是让你直到这么一个二叉搜索树的概念。

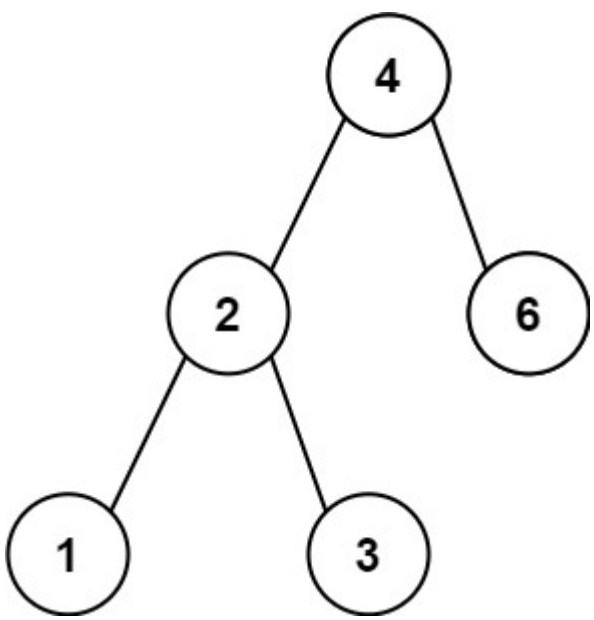
---

530. 二叉搜索树的最小绝对差

给你一个二叉搜索树的根节点 `root` ，返回 **树中任意两不同节点值之间的最小差值** 。

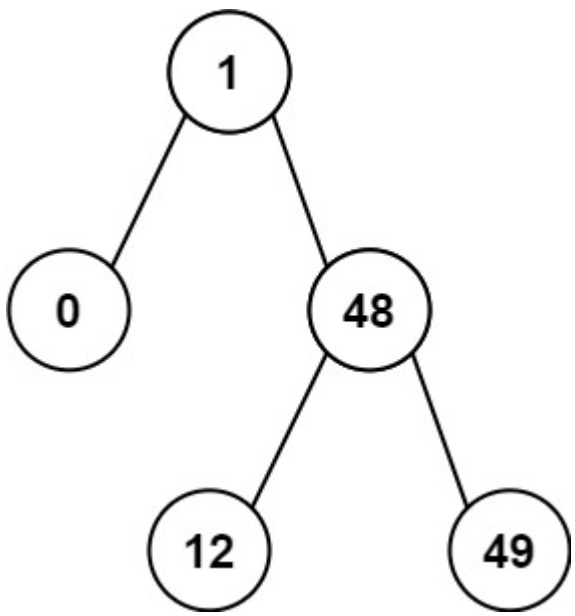
差值是一个正数，其数值等于两值之差的绝对值。

示例 1：




```
输入：root = [4,2,6,1,3]
输出：1
```

示例 2：




```
输入：root = [1,0,48,null,null,12,49]
输出：1
```

提示：

- 树中节点的数目范围是 `[2, 104]`
- `0 <= Node.val <= 105`

题解：

```
class Solution {
public:
    int getMinimumDifference(TreeNode* root) {
        if (!root || (!root->left && !root->right)) return 0;
        int maxNumber = 0;
        int minNumber = -2000000;
        int aws = 99999999;
        stack<TreeNode*> nodeRuan;
        TreeNode* ruan = root;
        //if (ruan) cout << "root为空" << endl;
        while (ruan||!nodeRuan.empty())
        {
            if (ruan) {
                nodeRuan.push(ruan);
                ruan = ruan->left;
            }
            else
            {
                ruan = nodeRuan.top();
                nodeRuan.pop();

                maxNumber = ruan->val;
                if ((maxNumber - minNumber) < aws) {
                    aws = (maxNumber - minNumber);
                }
                //cout << "    当前为" << maxNumber << "     上一个值为" << minNumber<<"   此时差为"<<maxNumber-
minNumber << endl;
                ruan = ruan->right;
                minNumber = maxNumber;
            }
        }
        return aws;
    }
};
```

我的笔记：就是单纯中序遍历，然后每一次统计上一次和这一次的差值

PS：其实不需要像我这样设置初始化，加一个栈或者队列就能解决问题

---

501. 二叉搜索树中的众数

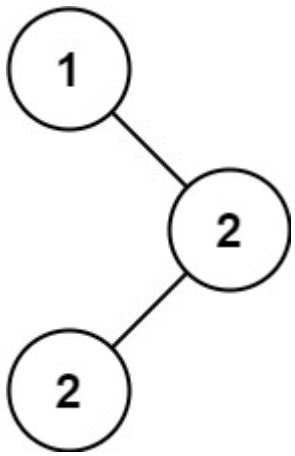
给你一个含重复值的二叉搜索树（BST）的根节点 `root` ，找出并返回 BST 中的所有 众数（即，出现频率最高的元素）。

如果树中有不止一个众数，可以按 **任意顺序** 返回。

假定 BST 满足如下定义：

- 结点左子树中所含节点的值 **小于等于** 当前节点的值
- 结点右子树中所含节点的值 **大于等于** 当前节点的值
- 左子树和右子树都是二叉搜索树

示例 1：




```
输入：root = [1,null,2,2]
输出：[2]
```

示例 2：

```
输入：root = [0]
输出：[0]
```

提示：

- 树中节点的数目在范围 `[1, 104]` 内
- `-105 <= Node.val <= 105`

进阶：你可以不使用额外的空间吗？（假设由递归产生的隐式调用栈的开销不被计算在内）

题解：

1.我的方法：写了一个计数器,简直是狗屎

```
class Solution {
public:

    void jishuqi(TreeNode* root) {          //计数器函数
        if (aws.empty()) {
            aws.push_back(root->val);
            this->tempNumber= root;
            this->tempCounter = 1;
            this->tempMax = 1;            //初始化最大数量
        }
        else
        {
            if (this->tempNumber->val == root->val) {    //和上一个一样，计数器++
                this->tempCounter++;
            }
            else    //和上一个不一样，上一个统计结束
            {
                if (this->tempCounter > this->tempMax) {    //超过了有史以来的所有数
                    this->tempMax = this->tempCounter;     //新的记录被添加
                    aws.clear();                        //结果数组被清空
                    aws.push_back(this->tempNumber->val);        //新的数被记录
                }
                else if (this->tempCounter == this->tempMax)     //和当前记录持平
```

```
                {
                    if(this->tempNumber->val!=aws[0])aws.push_back(this->tempNumber->val);            //该数也被添加
                }
                this->tempCounter = 1;                //计数器被重置为1
                this->tempNumber->val = root->val;          //更新当前数
            }
        }
    }
    void inOrder(TreeNode* root) {
        if (!root) return;
        if (root->left)   inOrder(root->left);
        jishuqi(root);

        if (root->right)   inOrder(root->right);
    }
    vector<int> findMode(TreeNode* root) {
        if (!root) return this->aws;
         inOrder(root);
        TreeNode* tempMou = new TreeNode(999999);
        jishuqi(tempMou);
        return this->aws;
    }
public:
    vector<int> aws;    //结果
    int tempMax;       //最大值
    TreeNode* tempNumber;    //保存上一值
    int tempCounter;      //保存上一值的计数器
};
```

我的笔记：这个题目想到有两种解法，第一种是使用哈希表，遍历整个数组，把统计出来的数的频率（键值对）进行排序。

第二种就是我这种，既然题目说了是二叉搜索树，那么它的中序遍历就是有序的，前面一个元素和后面一个元素就是指相邻的两个节点。

遇到的问题：1.第一个元素会被添加两次，所以我增加了一个让它只添加一次的设定。2.末尾的节点没有被统计上，所以我在结束完遍历+计数器函数调用后，单独拿了一个让它多执行1次计数器。

---

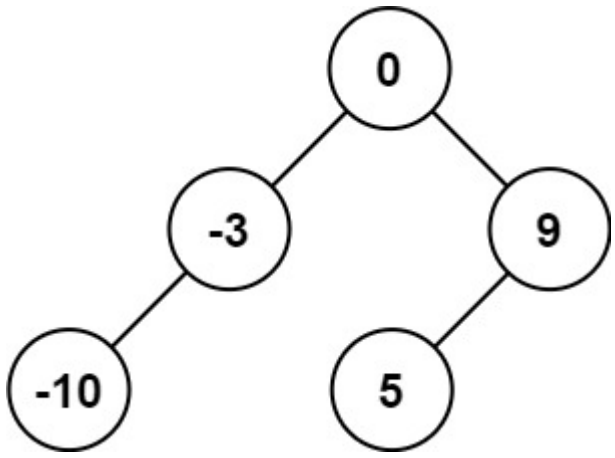
108. 将有序数组转换为二叉搜索树

给你一个整数数组 `nums` ，其中元素已经按 **升序** 排列，请你将其转换为一棵 平衡二叉搜索树。

> 平衡二叉搜索树

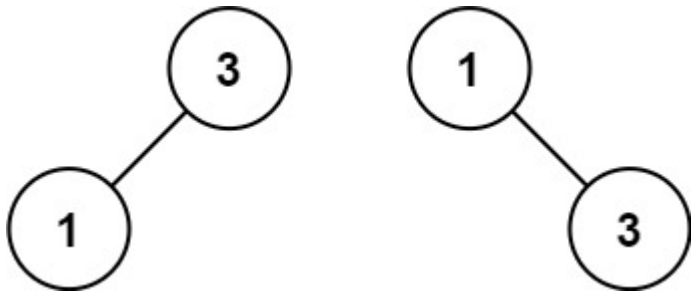
它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

示例 1：




```
输入：nums = [-10,-3,0,5,9]
输出：[0,-3,9,-10,null,5]
解释：[0,-10,5,null,-3,null,9] 也将被视为正确答案：
```

示例 2：




```
输入：nums = [1,3]
输出：[3,1]
解释：[1,null,3] 和 [3,1] 都是高度平衡二叉搜索树。
```

提示：

- `1 <= nums.length <= 104`
- `-104 <= nums[i] <= 104`
- `nums` 按 **严格递增** 顺序排列

题解：

```cpp
class Solution {
private:
    TreeNode* traversal(vector<int>& nums, int left, int right) {
        if (left > right) return nullptr;
        int mid = left + ((right - left) / 2);
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = traversal(nums, left, mid - 1);
        root->right = traversal(nums, mid + 1, right);
        return root;
    }
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        TreeNode* root = traversal(nums, 0, nums.size() - 1);
        return root;
```

```
    }
};
```

我的笔记：在推导演练的时候不够严谨。

我之前的做法是计算：

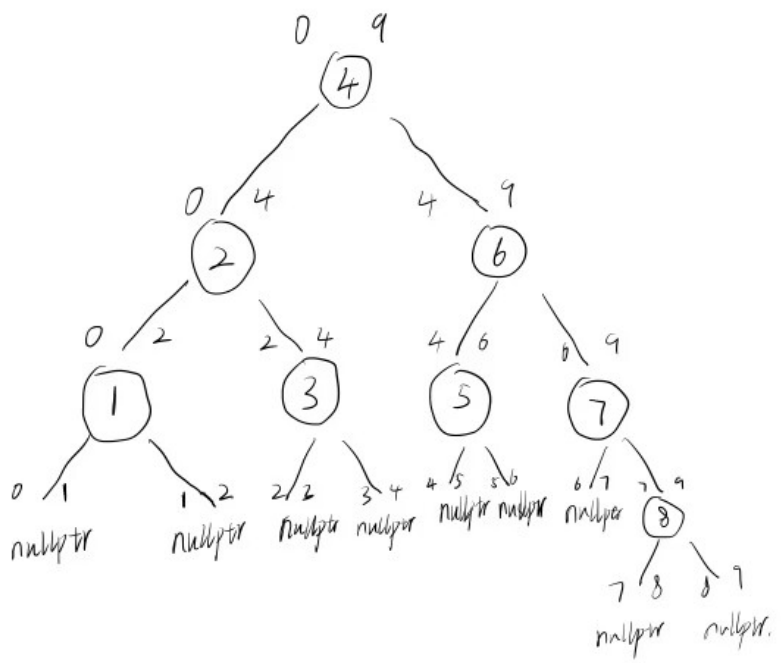
```
if(mid==left) return nullptr
    root->left = traversal(nums, left, mid );
        root->right = traversal(nums, mid , right);
```

这样存在一个缺陷，就是nums[0]和nums[nums.size()-1]就需要手动添加 会出现严重的bug（图2）

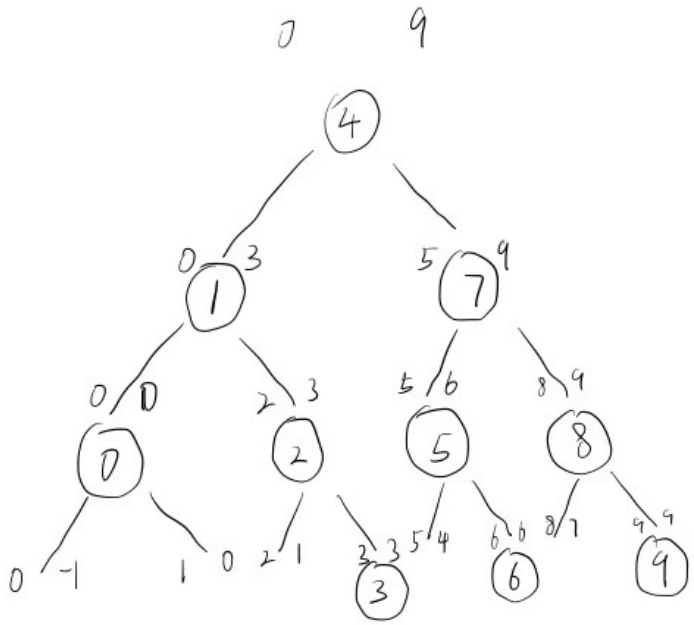
正确答案执行过程见图一

0 1 2 3 4 5 6 7 8 9

1.



2、



中等难度

209. 长度最小的子数组

给定一个含有 `n` 个正整数的数组和一个正整数 `target` **。**

找出该数组中满足其总和大于等于 `target` 的长度最小的

子数组

`[numsl, numsl+1, ..., numsr-1, numsr]` ，并返回其长度。如果不存在符合条件的子数组，返回 `0` 。

示例 1：

```
输入：target = 7, nums = [2,3,1,2,4,3]
输出：2
解释：子数组 [4,3] 是该条件下的长度最小的子数组。
```

示例 2：

```
输入：target = 4, nums = [1,4,4]
输出：1
```

示例 3：

```
输入：target = 11, nums = [1,1,1,1,1,1,1,1]
输出：0
```

提示：

- `1 <= target <= 109`
- `1 <= nums.length <= 105`
- `1 <= nums[i] <= 105`

进阶：

- 如果你已经实现 `O(n)` 时间复杂度的解法, 请尝试设计一个 `O(n log(n))` 时间复杂度的解法。

题解

滑动窗口算法

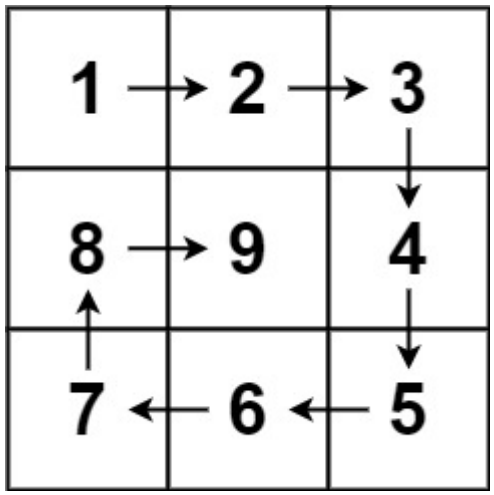
```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
       //求第一段最短的长度
        int len = 0;
        int sum = 0;
        while (sum<target)
        {
            if ( len>= nums.size() )
            {
                return 0;
            }
            sum += nums[len];
            len++;
        }
        //return len;
        //开始最初的滑动窗口
        int left = 0;
        int right = left + len-1;
        while ((sum - nums[left]) >= target)
        {
            sum = sum - nums[left];
            left++;
        }                                          //窗口最初的大小
        while (right<nums.size()-1) {
            sum = sum - nums[left] + nums[right + 1];
            right++;
            left++;
                while ((sum - nums[left]) >= target)
                {
                    sum = sum - nums[left];
                    left++;
                }
        }
        len = right - left+1;
        return len;
    }
};
```

---

59. 螺旋矩阵 II

给你一个正整数 `n` ，生成一个包含 `1` 到 `n2` 所有元素，且元素按顺时针顺序螺旋排列的 `n x n` 正方形矩阵 `matrix` 。

示例 1：




```
输入：n = 3
输出：[[1,2,3],[8,9,4],[7,6,5]]
```

示例 2：

```
 输入：n = 1
输出：[[1]]
```

提示：

- `1 <= n <= 20`

题解：

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        vector<vector<int>> Ruan(n, vector<int>(n, 0));
        int number = 1;     //填入的数字
        int len = 0;
        int j = 0;
        int layer = 0;  //记录层数
        int maxLayer = (n%2 !=0)?(n/2+1):n/2;     //统计有多少层
        while (layer<=maxLayer)
        {
            //横向--->
            for (int i = layer; i < n-layer; i++)
            {
                Ruan[layer][i] = number;
                number++;
            }
            number--;
            //纵向向下
            for (int j = layer; j < n - layer; j++) {
                Ruan[j][n-layer-1] = number;
                number++;
            }
            number--;
            //横向向左
            for (int j = layer; j < n - layer; j++) {
                Ruan[n-layer-1][n-j-1] = number;
```

```
                number++;
            }
            number--;
            //纵向向上
            for (int j = layer; j < n - layer-1; j++) {
                Ruan[n - j - 1][layer] = number;
                number++;
            }
            layer++;
        }
        return Ruan;
    }
};
```

---

707. 设计链表

你可以选择使用单链表或者双链表，设计并实现自己的链表。

单链表中的节点应该具备两个属性：`val` 和 `next` 。`val` 是当前节点的值，`next` 是指向下一个节点的指针/引用。

如果是双向链表，则还需要属性 `prev` 以指示链表中的上一个节点。假设链表中的所有节点下标从 **0** 开始。

实现 `MyLinkedList` 类：

- `MyLinkedList()` 初始化 `MyLinkedList` 对象。
- `int get(int index)` 获取链表中下标为 `index` 的节点的值。如果下标无效，则返回 `-1` 。
- `void addAtHead(int val)` 将一个值为 `val` 的节点插入到链表中第一个元素之前。在插入完成后，新节点会成为链表的第一个节点。
- `void addAtTail(int val)` 将一个值为 `val` 的节点追加到链表中作为链表的最后一个元素。
- `void addAtIndex(int index, int val)` 将一个值为 `val` 的节点插入到链表中下标为 `index` 的节点之前。如果 `index` 等于链表的长度，那么该节点会被追加到链表的末尾。如果 `index` 比长度更大，该节点将 **不会插入** 到链表中。
- `void deleteAtIndex(int index)` 如果下标有效，则删除链表中下标为 `index` 的节点。

示例：

```
输入
["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]
[[], [1], [3], [1, 2], [1], [1], [1]]
输出
[null, null, null, null, 2, null, 3]

解释
MyLinkedList myLinkedList = new MyLinkedList();
myLinkedList.addAtHead(1);
myLinkedList.addAtTail(3);
myLinkedList.addAtIndex(1, 2);    // 链表变为 1->2->3
myLinkedList.get(1);              // 返回 2
myLinkedList.deleteAtIndex(1);    // 现在，链表变为 1->3
myLinkedList.get(1);              // 返回 3
```

提示：

- `0 <= index, val <= 1000`
- 请不要使用内置的 LinkedList 库。
- 调用 `get`、`addAtHead`、`addAtTail`、`addAtIndex` 和 `deleteAtIndex` 的次数不超过 `2000` 。

题解：

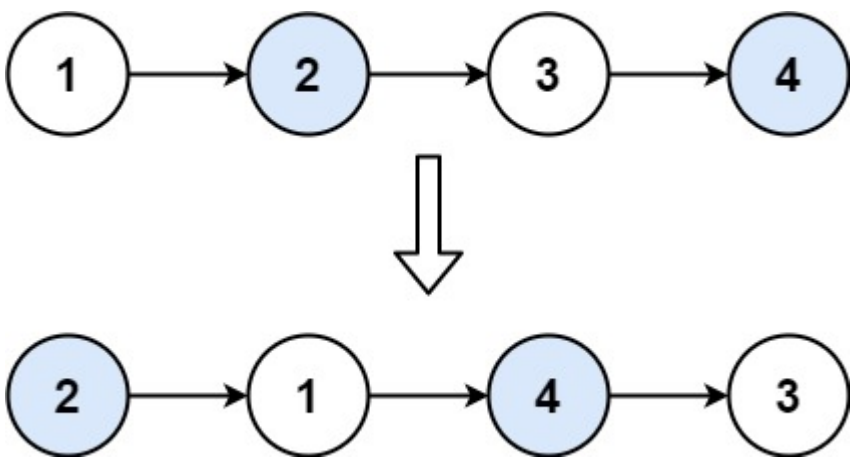
```
class MyLinkedList {
public:
    // 定义链表节点结构体
    struct LinkedNode {
        int val;
        LinkedNode* next;
        LinkedNode(int val):val(val), next(nullptr){}
    };

    // 初始化链表
    MyLinkedList() {
        _dummyHead = new LinkedNode(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;
        }
        LinkedNode* cur = _dummyHead;
        index++;
        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) {
        LinkedNode* newNode = new LinkedNode(val);
        LinkedNode* cur=_dummyHead;
        newNode->next = cur->next;
        cur->next = newNode;
        _size++;
    }

    // 在链表最后面添加一个节点
    void addAtTail(int val) {
        LinkedNode* newNode = new LinkedNode(val);
        LinkedNode* cur = _dummyHead;
        while(cur->next != NULL){
            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
// 如果index小于0，则在头部插入节点
void addAtIndex(int index, int val) {

    if(index > _size) return;
    if(index < 0) index = 0;
    LinkedNode* newNode = new LinkedNode(val);
    LinkedNode* cur = _dummyHead;
    while(index--) {
        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;
    }
    LinkedNode* cur = _dummyHead;
    while(index--) {
        cur = cur ->next;
    }
    LinkedNode* tmp = cur->next;
        cur->next = cur->next->next;
        delete tmp;
        //delete命令指示释放了tmp指针原本所指的那部分内存，
        //被delete后的指针tmp的值（地址）并非就是NULL，而是随机值。也就是被delete后，
        //如果不再加上一句tmp=nullptr,tmp会成为乱指的野指针
        //如果之后的程序不小心使用了tmp，会指向难以预想的内存空间
        tmp=nullptr;
        _size--;
    }

    // 打印链表
    void printLinkedList() {
        LinkedNode* cur = _dummyHead;
        while (cur->next != nullptr) {
            cout << cur->next->val << " ";
            cur = cur->next;
        }
        cout << endl;
    }
private:
    int _size;
    LinkedNode* _dummyHead;

};
```

24. 两两交换链表中的节点

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

示例 1：




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

示例 2：

```
输入：head = []
输出：[]
```

示例 3：

```
输入：head = [1]
输出：[1]
```

提示：

- 链表中节点的数目在范围 `[0, 100]` 内
- `0 <= Node.val <= 100`

题解：

```cpp
/**
 * 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* swapPairs(ListNode* head) {
```

```
        ListNode* ruan = head;
        while (ruan==nullptr || ruan->next == nullptr)        //当为空或者仅仅为1个元素的时候
        {
            return head;
        }
        ListNode* mou = head->next;
        head=head->next;
        ListNode* temp = mou;
    while (ruan!=nullptr && ruan->next!=nullptr)
    {
        temp= ruan->next->next;
        mou= ruan->next;
        if(temp==nullptr || temp->next==nullptr){   //AB型或者ABC型
        ruan->next=mou->next;
        mou->next=ruan;
        }
        else{    //ABCD型
        ruan->next=temp->next;
        mou->next=ruan;
        }
        if(temp==nullptr){
            return head;
        }
        ruan=temp;
    }
    return head;
    }
};
```

笔记：这道题无非就考虑一个：

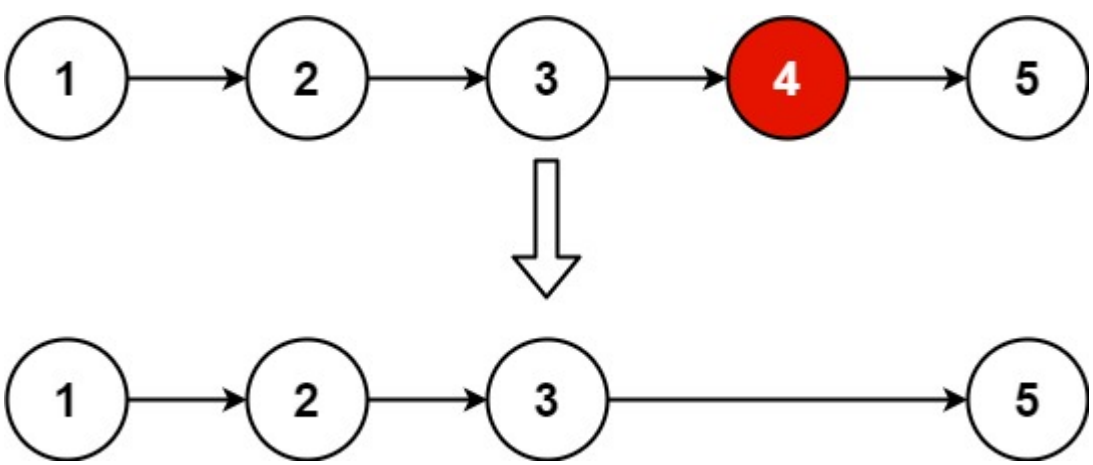
AB型和ABC型:改变AB结果都无需考虑C，但是如果是ABCD型，当一旦改变了AB的顺序，新的顺序BA中的A的下一个指向是D，别忘了。

---

19. 删除链表的倒数第 N 个结点

给你一个链表，删除链表的倒数第 `n` 个结点，并且返回链表的头结点。

示例 1：




```
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]
```

示例 2：

```
输入：head = [1], n = 1
输出：[]
```

示例 3：

```
输入：head = [1,2], n = 1
输出：[1]
```

提示：

- 链表中结点的数目为 `sz`
- `1 <= sz <= 30`
- `0 <= Node.val <= 100`
- `1 <= n <= sz`

进阶：你能尝试使用一趟扫描实现吗？

题解：

```cpp
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        int sz = 0;
        ListNode* p = head;
        while (p) {
            p = p->next;
            sz++;
        }
        n = sz - n + 1;
        if (n == 1) {
            head = head->next ? head->next : nullptr;
            return head;
        }
        p = head;
        for (int i = 1; i < n - 1; i++)
            p = p->next;

        p->next = p->next ? p->next->next : nullptr;
        return head;
    }
};
```

笔记：就是使用两个快慢指针同步来进行移动，快指针刚刚好比满指针快了n个位置，当快指针到达末尾的时候，慢指针也就到达了对应的删除的位置

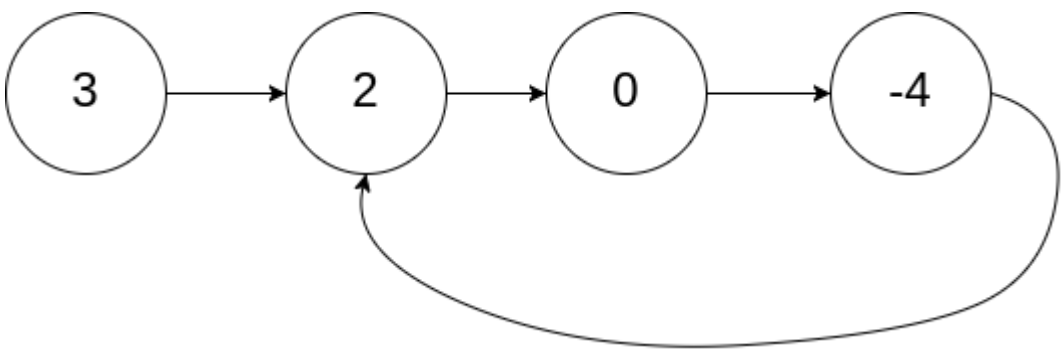
142. 环形链表 II

给定一个链表的头节点 `head` ，返回链表开始入环的第一个节点。 *如果链表无环，则返回 `null`*。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。 为了表示给定链表中的环，评测系统内部使用整数 `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。**注意：** `pos` **不作为参数进行传递**，仅仅是为了标识链表的实际情况。

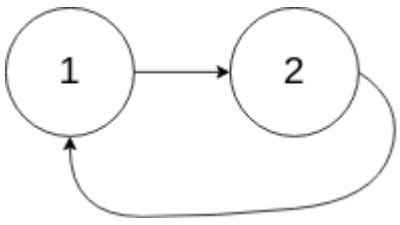
不允许修改 链表。

示例 1：




```
输入：head = [3,2,0,-4], pos = 1
输出：返回索引为 1 的链表节点
解释：链表中有一个环，其尾部连接到第二个节点。
```

示例 2：




```
输入：head = [1,2], pos = 0
输出：返回索引为 0 的链表节点
解释：链表中有一个环，其尾部连接到第一个节点。
```

示例 3：


```
输入：head = [1], pos = -1
输出：返回 null
解释：链表中没有环。
```

提示：

- 链表中节点的数目范围在范围 `[0, 104]` 内
- `-105 <= Node.val <= 105`
- `pos` 的值为 `-1` 或者链表中的一个有效索引

进阶：你是否可以使用 `O(1)` 空间解决此题？

题解：

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* detectCycle(ListNode* head) {
        ListNode* ruan = head;   //定义慢指针
        ListNode* mou = head;    //定义快指针
        ListNode* current = head;   //其实这两个指针可以循环利用，没必要创建新的指针current和newHead
        ListNode* newHead = head;
        int pos = 0;
        while (mou!=nullptr)
        {
            ruan = ruan->next;
            mou = mou->next->next;
            if (ruan == mou) {    //快慢指针相遇了
                current = ruan;
                break;
            }
            if (mou->next == nullptr) {    //碰到空指针了，不存在循环
                pos = -1;
                return nullptr;
            }
        }
        while (current != newHead)
        {
            pos++;
            current = current->next;
            newHead = newHead->next;
        }
        //cout << "pos=" << pos << endl;
        return current;
    }
};
```

笔记：

整体思路：定义一个快指针和一个慢指针，慢指针一次走1个元素，而快指针一次走2个元素，如果链表中存在循环，则快指针一定会与慢指针相遇（追击问题）

然后通过推理可得第一次相遇时，慢指针还没有完成一圈的剩下距离就等于达到这个开始循环开始的距离。

---

454. 四数相加 II

给你四个整数数组 `nums1`、`nums2`、`nums3` 和 `nums4`，数组长度都是 `n`，请你计算有多少个元组 `(i, j, k, l)` 能满足：

- `0 <= i, j, k, l < n`
- `nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0`

示例 1：

```
输入：nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
输出：2
解释：
两个元组如下：
1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0
```

示例 2：

```
输入：nums1 = [0], nums2 = [0], nums3 = [0], nums4 = [0]
输出：1
```

提示：

- `n == nums1.length`
- `n == nums2.length`
- `n == nums3.length`
- `n == nums4.length`
- `1 <= n <= 200`
- `-228 <= nums1[i], nums2[i], nums3[i], nums4[i] <= 228`

题解：

```
class Solution {
public:
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3, vector<int>& nums4) {
        map<int, int> ruanMap;
        int aws = 0;
        for (int i = 0; i < nums1.size(); i++)
        {
            for (int j = 0; j < nums2.size(); j++) {
                ruanMap[nums1[i] + nums2[j]]++;       //如此实现让键值对自增
            }
        }
        //printMap(ruanMap);
        for (int i = 0; i < nums3.size(); i++)
        {
            for (int j = 0; j < nums4.size(); j++) {
                map<int,int>::iterator  itRuan = ruanMap.find(-nums3[i] - nums4[j]);
                       if (itRuan!=ruanMap.end())
                       {
                        //cout<<"now the nums1[i] + nums2[j] is "<<nums3[i] + nums4[j]<<endl;;
                        aws+=ruanMap[-nums3[i] - nums4[j]];
```

```
                    }
                }
            }
            return aws;
        }
};
```

Map常用知识点回顾

> 常用成员函数

成员函数	功能介绍
`insert()`	插入一个键值对。如果键已存在，插入操作不会生效。
`erase()`	删除指定键或迭代器位置的元素。
`find()`	查找指定键的元素，返回指向该元素的迭代器。如果元素不存在，返回 `end()`。
`at()`	访问指定键的元素，若键不存在则抛出异常。
`operator[]`	访问或插入指定键的元素。若键不存在，则插入该键并返回对应的值。
`size()`	返回 `map` 中元素的数量。
`empty()`	检查 `map` 是否为空。
`clear()`	清空 `map` 中的所有元素。
`begin()`	返回指向第一个元素的迭代器。
`end()`	返回指向最后一个元素后一个位置的迭代器。

```
#include <iostream>
#include <map>
#include <string>

int main() {
    // 创建一个 std::map
    std::map<std::string, int> myMap;

    // 插入键值对
    myMap.insert(std::make_pair("apple", 3));
    myMap["banana"] = 5;
    myMap["orange"] = 2;

    // 访问元素
    std::cout << "apple: " << myMap["apple"] << std::endl;
    std::cout << "banana: " << myMap.at("banana") << std::endl;

    // 查找元素
    auto it = myMap.find("orange");
    if (it != myMap.end()) {
        std::cout << "orange found: " << it->second << std::endl;
    }
```

```
    // 删除元素
    myMap.erase("banana");

    // 遍历 map 中的元素
    std::cout << "Remaining elements:" << std::endl;
    for (const auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

---

15. 三数之和

给你一个整数数组 `nums` ，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k` ，同时还满足 `nums[i] + nums[j] + nums[k] == 0` 。请你返回所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
解释：
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。
不同的三元组是 [-1,0,1] 和 [-1,-1,2] 。
注意，输出的顺序和三元组的顺序并不重要。
```

示例 2：

```
输入：nums = [0,1,1]
输出：[]
解释：唯一可能的三元组和不为 0 。
```

示例 3：

```
输入：nums = [0,0,0]
输出：[[0,0,0]]
解释：唯一可能的三元组和为 0 。
```

提示：

- `3 <= nums.length <= 3000`
- `-105 <= nums[i] <= 105`

题解：

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> aws;
        sort(nums.begin(), nums.end());
        int left = 0;
        int right = 0;
        int i = 0;
        while ( i < nums.size()-2)
        {

            left = i + 1;
            right = nums.size() - 1;

            while (left<right)
            {
                if ((nums[i] + nums[left] + nums[right])<0) {
                    left++;
                }
                else if ((nums[i] + nums[left] + nums[right]) > 0)
                {
                    right--;
                }
                else
                {
                    aws.push_back({ nums[i],nums[left],nums[right] });
                    //cout << "i=" << i << endl;
                    right--;
                    left++;
                    while (nums[left] == nums[left -1 ] && nums[right]==nums[right+1]&&left<right) {
                        right--;
                        left++;
                    }
                }
            }
            while (nums[i] == nums[i+1]&&i<nums.size()-2) {
                i++;
            }
            i++;
        }
        return aws;
    }
};
```

18. 四数之和

给你一个由 `n` 个整数组成的数组 `nums` ，和一个目标值 `target` 。请你找出并返回满足下述全部条件且**不重复**的四元组 `[nums[a], nums[b], nums[c], nums[d]]` （若两个四元组元素一一对应，则认为两个四元组重复）：

- `0 <= a, b, c, d < n`
- `a`、`b`、`c` 和 `d` **互不相同**
- `nums[a] + nums[b] + nums[c] + nums[d] == target`

你可以按 **任意顺序** 返回答案 。

示例 1：

```
输入：nums = [1,0,-1,0,-2,2], target = 0
输出：[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

示例 2：

```
输入：nums = [2,2,2,2,2], target = 8
输出：[[2,2,2,2]]
```

提示：

- `1 <= nums.length <= 200`
- `-109 <= nums[i] <= 109`
- `-109 <= target <= 109`

题解：

```
vector<vector<int>> aws;

        if (nums.size() < 4) {
            return aws;
        }

        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size() - 3; ++i) {
            // 跳过重复的数
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            for (int j = i + 1; j < nums.size() - 2; ++j) {
                // 跳过重复的数
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;

                int left = j + 1;
                int right = nums.size() - 1;

                while (left < right) {
                    long long sum = (long long)nums[i] + nums[j] + nums[left] + nums[right];
                    if (sum < target) {
```

```
                    ++left;
                } else if (sum > target) {
                    --right;
                } else {
                    aws.push_back({ nums[i], nums[j], nums[left], nums[right] });
                    ++left;
                    --right;
                    // 跳过重复的数
                    while (left < right && nums[left] == nums[left - 1]) ++left;
                    while (left < right && nums[right] == nums[right + 1]) --right;
                }
            }
        }
    }

    return aws;
```

---

151. 反转字符串中的单词

给你一个字符串 `s` ，请你反转字符串中 **单词** 的顺序。

单词 是由非空格字符组成的字符串。`s` 中使用至少一个空格将字符串中的 **单词** 分隔开。

返回 **单词** 顺序颠倒且 **单词** 之间用单个空格连接的结果字符串。

注意：输入字符串 `s`中可能会存在前导空格、尾随空格或者单词间的多个空格。返回的结果字符串中，单词间应当仅用单个空格分隔，且不包含任何额外的空格。

示例 1：

```
输入：s = "the sky is blue"
输出："blue is sky the"
```

示例 2：

```
输入：s = "  hello world  "
输出："world hello"
解释：反转后的字符串中不能存在前导空格和尾随空格。
```

示例 3：

```
输入：s = "a good   example"
输出："example good a"
解释：如果两个单词间有多余的空格，反转后的字符串需要将单词间的空格减少到仅有一个。
```

提示：

- `1 <= s.length <= 104`
- `s` 包含英文大小写字母、数字和空格 `' '`
- `s` 中 **至少存在一个** 单词

进阶：如果字符串在你使用的编程语言中是一种可变数据类型，请尝试使用 `O(1)` 额外空间复杂度的 **原地** 解法。

题解：

1.从末尾开始抄录单个字符，写入temp字符串中与' '(这里是#)拼接为一个新的字符串"#naur"，然后再将temp字符串反转存入到结果中(ruan#)，最后需要删除末尾的那个空格

```
class Solution {
public:
    string reverseWords(string s) {
        string temp;
        string aws;
        for (int i = s.length()-1; i>=0; i--) {
            if (s[i] == ' ') continue;
            temp = ' ';
            if (i==0 && s[i]!=' ')
            {
                temp += s[i];
            }
            while (s[i]!=' ' && i>0)
            {
                temp += s[i];
                if (i == 1 && s[0] != ' ') {
                    temp += s[0];
                }
                i--;
            }
            reverse(temp.begin(),temp.end());
            aws += temp;
    }
        aws = aws.substr(0 aws.length() - 1);
        return aws;
    }
};
```

---

150. 逆波兰表达式求值(string转换int)

给你一个字符串数组 `tokens` ，表示一个根据 逆波兰表示法 表示的算术表达式。

请你计算该表达式。返回一个表示表达式值的整数。

注意：

- 有效的算符为 `'+'`、`'-'`、`'*'` 和 `'/'` 。
- 每个操作数（运算对象）都可以是一个整数或者另一个表达式。
- 两个整数之间的除法总是 **向零截断** 。
- 表达式中不含除零运算。
- 输入是一个根据逆波兰表示法表示的算术表达式。
- 答案及所有中间计算结果可以用 **32 位** 整数表示。

示例 1：

```
输入：tokens = ["2","1","+","3","*"]
输出：9
解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9
```

示例 2：

```
输入：tokens = ["4","13","5","/","+"]
输出：6
解释：该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6
```

示例 3：

```
输入：tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
输出：22
解释：该算式转化为常见的中缀算术表达式为：
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

提示：

- `1 <= tokens.length <= 104`
- `tokens[i]` 是一个算符（`"+"`、`"-"`、`"*"` 或 `"/"`），或是在范围 `[-200, 200]` 内的一个整数

逆波兰表达式：

逆波兰表达式是一种后缀表达式，所谓后缀就是指算符写在后面。

- 平常使用的算式则是一种中缀表达式，如 `( 1 + 2 ) * ( 3 + 4 )` 。
- 该算式的逆波兰表达式写法为 `( ( 1 2 + ) ( 3 4 + ) * )` 。

逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成 `1 2 + 3 4 + *` 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中

题解：

```cpp
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> Number;
        int tempNumA= 0;
        int tempNumB = 0;
        for (int i = 0; i < tokens.size(); i++)
        {
            if (tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "*" || tokens[i] == "/") {
                tempNumA= Number.top();
                Number.pop();
                tempNumB = Number.top();
```

```
                Number.pop();
                if (tokens[i]=="+")
                {
                    Number.push(tempNumA + tempNumB);
                }
                else if (tokens[i] == "-")
                {
                    Number.push(tempNumB - tempNumA);
                }
                else if (tokens[i] == "*")
                {
                    Number.push(tempNumB * tempNumA);
                }
                else if (tokens[i] == "/")
                {
                    Number.push(tempNumB / tempNumA);
                }
            }
            else
            {
                Number.push(atoi(tokens[i].c_str()));
            }
        }
        return Number.top();
    }
};
```

解题思路：用栈解决

---

347. 前 K 个高频元素

给你一个整数数组 `nums` 和一个整数 `k` ，请你返回其中出现频率前 `k` 高的元素。你可以按 **任意顺序** 返回答案。

示例 1:

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

提示：

- `1 <= nums.length <= 105`
- `k` 的取值范围是 `[1, 数组中不相同的元素的个数]`
- 题目数据保证答案唯一，换句话说，数组中前 `k` 个高频元素的集合是唯一的

进阶：你所设计算法的时间复杂度 **必须** 优于 `O(n log n)` ，其中 `n` 是数组大小。

题解：(程序347.2.cpp)

```
class Solution {
public:
    class compairRuan {
    public:
        bool operator() (const pair<int, int>& NumberPair1, const pair<int, int>& NumberPair2) {
            return NumberPair1.second > NumberPair2.second;
        }
    };
    vector<int> topKFrequent(vector<int>& nums, int k) {
        vector<int> aws(k);                 //先就定义一个长度为k的vector数组
        //先将所有的数据存放到无序表中
        unordered_map<int, int> ruanMap;
        for (int i = 0; i < nums.size(); i++)
        {
            ruanMap[nums[i]]++;
        }
        //printMap(ruanMap);

        //然后创建一个小顶堆表进行排序：看起来像是一个队列，内部能够根据仿函数进行排序
        priority_queue<pair<int, int>, vector<pair<int, int>>, compairRuan> ruanPriority_queue;
        //现在开始遍历这个无需表，将按照无需表的第二个值进行排序
        for (unordered_map<int, int>::iterator itRuan = ruanMap.begin(); itRuan != ruanMap.end(); itRuan++)
        {
            ruanPriority_queue.push(*itRuan);
            if (ruanPriority_queue.size()>k)          //控制表的长度为K
            {
                ruanPriority_queue.pop();
            }
        }
       // printPriority_queue(ruanPriority_queue);
        //现在开始弹出这个优先级表中的数，并保存到结果中
        for (int i = ruanPriority_queue.size()-1; i >-1 ; i--)
        {
            aws[i] = ruanPriority_queue.top().first;
            ruanPriority_queue.pop();
        }
        return aws;
    }
};
```

> **题解：**

使用一种 容器适配器就是**优先级队列。**

什么是优先级队列呢?

其实**就是一个披着队列外衣的堆**，因为优先级队列对外接口只是从队头取元素，从队尾添加元素，再无其他取元素的方式，看起来就是一个队列。而且优先级队列内部元素是自动依照元素的权值排列。那么它是如何有序排列的呢?

缺省情况下priority_queue利用max-heap（大顶堆）完成对元素的排序，这个大顶堆是以vector为表现形式的complete binary tree（完全二叉树）。

什么是堆呢?

堆是一棵完全二叉树，树中每个结点的值都不小于（或不大于）其左右孩子的值。

所以大家经常说的大顶堆（堆头是最大元素），小顶堆（堆头是最小元素），如果懒得自己实现的话，就直接用priority_queue（优先级队列）就可以了，底层实现都是一样的，从小到大排就是小顶堆，从大到小排就是大顶堆。

第二个问题就是关于这个**大小堆的实现**（其实就是实现了一个自定义优先级的优先级队列）

正常情况下的优先级队列：

`priority_queue ruanPriority_queue` 即可。

但是如本题目，要 想实现这个自定义优先级的优先级队列，需要传入三个参数

```
priority_queue(T,container,Compair) ruanPriority_queue;
```

1. `T`：存储的元素类型（元素的类型）。

   这个参数指定优先级队列中存储的元素类型。它可以是任何数据类型，比如 `int`、`float`、`std::string` 或用户定义的类型

2. `Container`：用于存储元素的底层容器类型，默认是 `std::vector<T>`。

   这个参数指定用来存储优先级队列元素的底层容器类型。默认情况下是 `std::vector<T>`，但也可以使用其他标准容器，比如 `std::deque<T>`。这个容器必须支持随机访问迭代器和以下操作：`push_back`、`pop_back`、`front`、`size` 和 `empty`。

3. `Compare`：用于比较元素优先级的比较函数对象类型，默认是 `std::less<T>`（用于最大堆）。

   这个参数指定用于比较优先级的函数对象，也被称为仿函数。默认情况下是 `std::less<T>`，用于最大堆。如果要实现最小堆，可以使用 `std::greater<T>`。此外，也可以传入自定义的比较函数对象。

其次补充一点就是仿函数的写法：

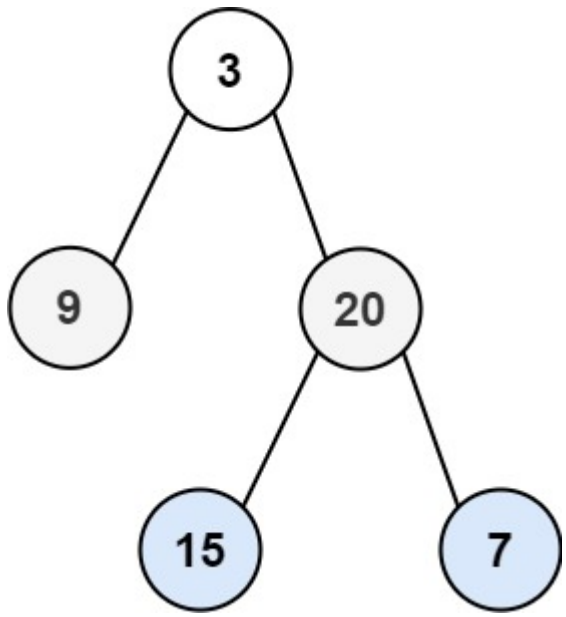
```
class 排序函数名 {
   public:
       bool operator()(数据结构传入的参数A，数据结构传入的参数B) {      //常见都是重写()运算符
           return A和B的某种比较结果
       }
   };
```

---

102. 二叉树的层序遍历

给你二叉树的根节点 `root` ，返回其节点值的 **层序遍历** 。 （即逐层地，从左到右访问所有节点）。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：[[3],[9,20],[15,7]]
```

示例 2：

```
输入：root = [1]
输出：[[1]]
```

示例 3：

```
输入：root = []
输出：[]
```

提示：

- 树中节点数目在范围 `[0, 2000]` 内
- `-1000 <= Node.val <= 1000`

题解：（102.cpp）

```cpp
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        queue<TreeNode*> temp;
        if (!root) return result;
        temp.push(root);
        TreeNode* ruan = root;

        int coutNum = 1;
        int RuanNum = 1;
        vector<int> tempVector;
        while (!temp.empty())
        {
            tempVector.clear();
            RuanNum = coutNum;
```

```
            coutNum = 0;
            while(RuanNum>0) {
                ruan = temp.front();
                if (temp.empty()) ruan = NULL;
                tempVector.push_back(ruan->val);
                temp.pop();
                if (ruan->left) {
                    temp.push(ruan->left);
                    coutNum++;
                };
                if (ruan->right) {
                    temp.push(ruan->right);
                    coutNum++;
                };
                RuanNum--;
            }
            result.push_back(tempVector);
        }
        return result;
    }
};
```

笔记：这其实就是广度优先搜索，迭代法和递归法都是深度优先算法。

---

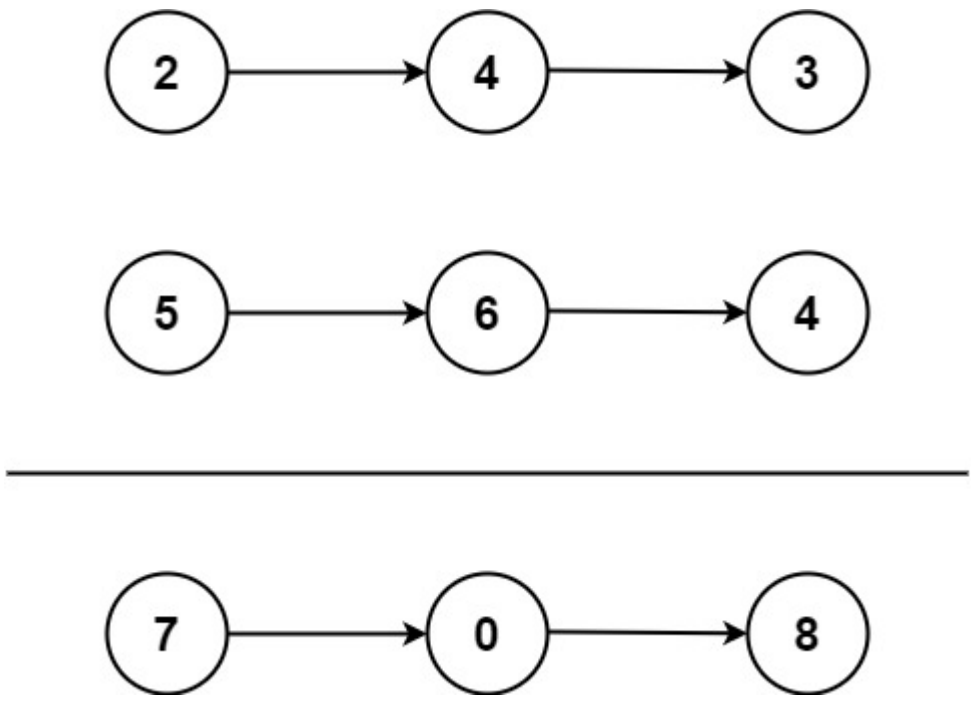
2. 两数相加

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储 **一位** 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：



```
输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]
解释：342 + 465 = 807.
```

示例 2：

```
输入：l1 = [0], l2 = [0]
输出：[0]
```

示例 3：

```
输入：l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
输出：[8,9,9,9,0,0,0,1]
```

提示：

- 每个链表中的节点数在范围 `[1, 100]` 内
- `0 <= Node.val <= 9`
- 题目数据保证列表表示的数字不含前导零

题解：

```
/**
 * 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:
    void doLongList(ListNode* longList, bool carryNumber, ListNode* tempHead) {
        int tempNum = 0;
        while (carryNumber)   //不断有进位的情况发生
        {
            if (!longList) {   //常量表也到头了，但是还能进位
                tempHead->next = new ListNode(1);
                return;
            }
            tempNum = longList->val + 1;
            tempHead->next = new ListNode(tempNum % 10);
            tempHead = tempHead->next;
            carryNumber = false;
            if (tempNum > 9) carryNumber = true;
            longList = longList->next;
        }
        //终于不需要进位了
        while (longList)
```

```
        {
            tempHead->next = new ListNode(longList->val);
            longList = longList->next;
            tempHead = tempHead->next;
        }
    }

    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* p1 = l1;
        ListNode* p2 = l2;
        ListNode* Head = new ListNode();
        ListNode* tempHead = Head;
        bool carryNumber = false;    //确定进位情况
        int tempNum = 0;
        while (p1 && p2)   //就是还没有到底的情况
        {
            if (carryNumber)      //这一头就是负责进位
            {
                tempNum = p1->val + p2->val + 1;
            }
            else
            {
                tempNum = p1->val + p2->val;
            }
            carryNumber = false;
            if (tempNum > 9) carryNumber = true;
            tempHead->next = new ListNode(tempNum%10);
            tempHead = tempHead->next;
            p1 = p1->next;
            p2 = p2->next;
        }
        //现在开始写到底的情况
        if (!p1) {          //l1.length<=l2.length
            if (!p2) {
                if (carryNumber) {
                    tempHead->next = new ListNode(1);
                }
                return Head->next;
            }    //如果l1.length=l2.length  直接返回
            doLongList(p2, carryNumber, tempHead);
        }
        else
        {
            doLongList(p1, carryNumber, tempHead);
        }
        return Head->next;


    }
};
```

我的笔记：

将短的链表和长的链表的同长度部分按位相加，然后再将长链表的部分抄过来即可

这里如果直接让长链表的剩余部分不抄过来（也就是如这个代码），而是直接改成`tempHead->next=longList`，那么如果要释放内存的时候就会出现重复释放内存的情况，所以不建议。

这道题主要要处理的情况就是，假设一个链表比另一个链表的长度更长，此时我就需要将处理剩下部分。剩下部分分为：1.需要受进位影响 2.没有受进位影响。

---

3. 无重复字符的最长子串

给定一个字符串 `s` ，请你找出其中不含有重复字符的 **最长子串**的长度。

示例 1:

```
输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

示例 3:

```
输入: s = "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

提示:

- `0 <= s.length <= 5 * 104`
- `s` 由英文字母、数字、符号和空格组成

题解:

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int longNuM = 0;
        if (s.empty()) return 0;
        if (s.length() == 1) return 1;
        deque<char> ruan;
        ruan.push_back(s[0]);
        int bigNum = 1;
        for (int i = 1; i < s.length(); i++) {
            ruan.push_back(s[i]);
```

```
            for (int j = 0; j < ruan.size()-1; j++)
            {
                if (s[i] == ruan[j]) {
                    for (int  k = 0; k < j+1; k++)
                    {
                        ruan.pop_front();
                    }
                    break;
                }
            }
            if (ruan.size() > bigNum)  bigNum = ruan.size();
        }
        return bigNum;
    }
};
```

我的笔记：

我的思路是，搞一个队列，每一次读取到一个字符就插入进队列，然后遍历这个队列，如果发现又和刚插入字符一样的字符，就pop和这个字符一样的字符和它以前的所有字符，然后再用一个数不断去记录最长的字符串

---

6. Z 字形变换

将一个给定字符串 `s` 根据给定的行数 `numRows` ，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 `"PAYPALISHIRING"` 行数为 `3` 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如：`"PAHNAPLSIIGYIR"`。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1：

```
输入：s = "PAYPALISHIRING", numRows = 3
输出："PAHNAPLSIIGYIR"
```

示例 2：

```
输入：s = "PAYPALISHIRING", numRows = 4
输出："PINALSIGYAHRPI"
解释：
P     I    N
A   L S  I G
Y A   H R
P     I
```

示例 3：

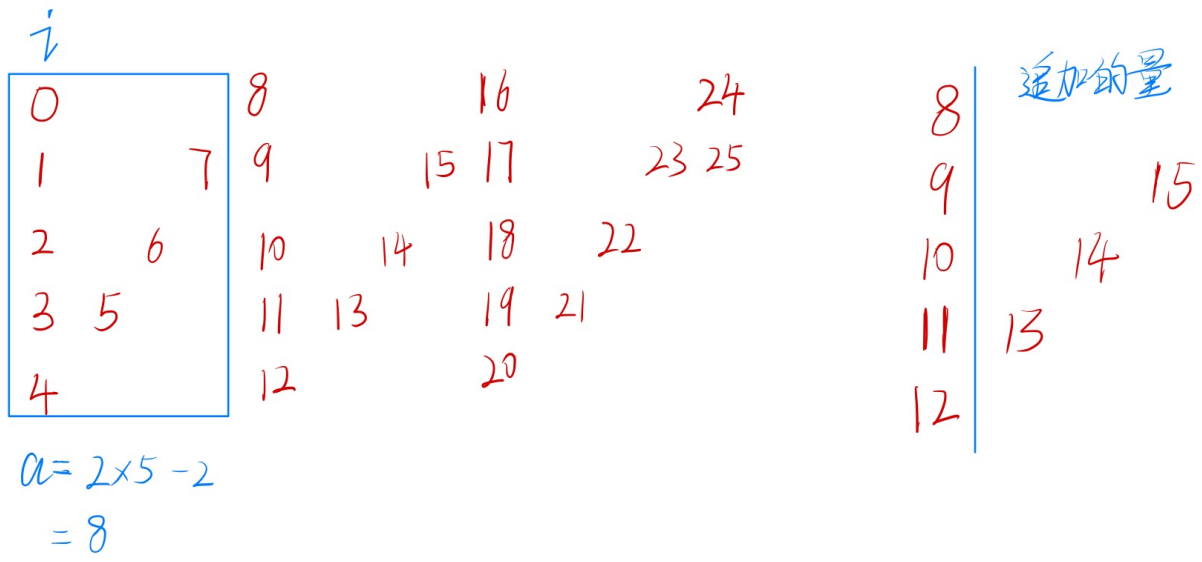
```
输入：s = "A", numRows = 1
输出："A"
```

提示：

- `1 <= s.length <= 1000`
- `s` 由英文字母（小写和大写）、`','` 和 `'.'` 组成
- `1 <= numRows <= 1000`

题解：

这是一种数学解法，将数组分为两个部分




```cpp
class Solution {
public:
    string convert(string s, int numRows) {
        if (numRows == 1) return s;
        int a = 2 * numRows - 2;    //单个组的长度
        string aws = "";
        for (int i = 0; i < numRows; i++) {
            int x = 0;
            int y = a * x + i;
            int y_2 = a * x + a - i;
            while (y<s.size())
            {
                aws += s[y];
                cout << s[y]<<y<<" ";
                y_2 = a * x + a - i;
                if (i != 0 && i != numRows - 1 && y_2<s.size()) {  //需要追加的列
                    aws += s[y_2];
```

```
                    cout << s[y_2]<<y_2 << " ";
                }
                x+=1;
                y = a * x + i;
            }
        }
        return aws;
    }
};
```

其中，a是单个分组的数长度

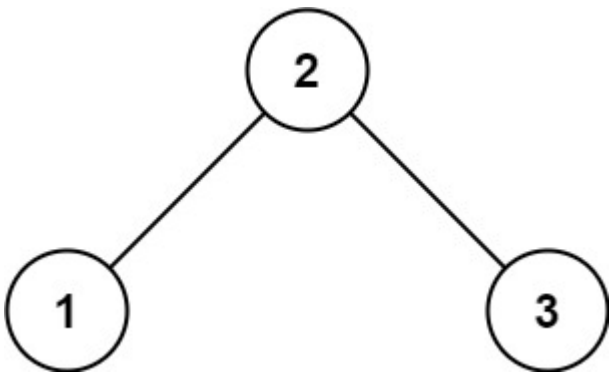
---

513. 找树左下角的值

给定一个二叉树的 **根节点** `root`，请找出该二叉树的 **最底层 最左边** 节点的值。

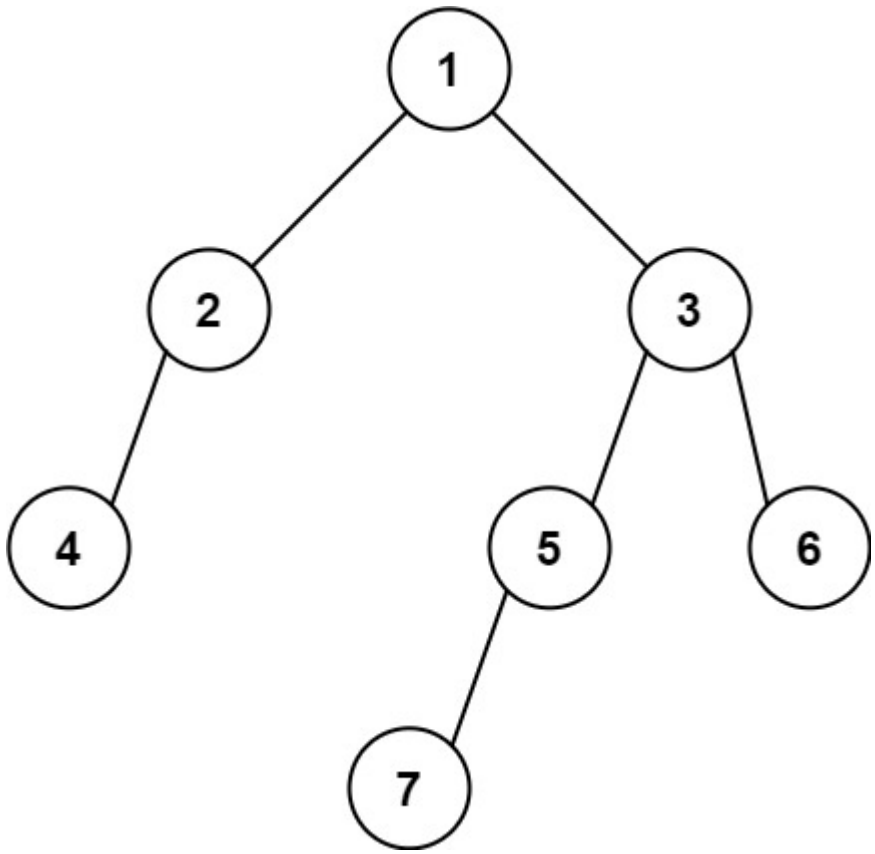
假设二叉树中至少有一个节点。

示例 1:




```
输入: root = [2,1,3]
输出: 1
```

示例 2:




```
输入: [1,2,3,4,null,5,6,null,null,7]
输出: 7
```

提示:

- 二叉树的节点个数的范围是 `[1,104]`
- `-231 <= Node.val <= 231 - 1`

题解:

```
class Solution {
public:
    void  findBottom(TreeNode* root,int deepth) {
        if (root->left||root->right) {
            if (root->left) {
                findBottom(root->left, deepth+1);
            }
            if (root->right) {
                findBottom(root->right, deepth + 1);
            }
        }
        else
        {
            if (deepth > this->bottomDepth) {
                this->bottomDepth = deepth;
                this->mostLeft = root->val;
            }
            return;
        }

    }
```

```
    int findBottomLeftValue(TreeNode* root) {
        if(!root) return 0;
        findBottom(root,1);
        return this->mostLeft;
    }
public:
    int bottomDepth;
    int mostLeft;
};
```

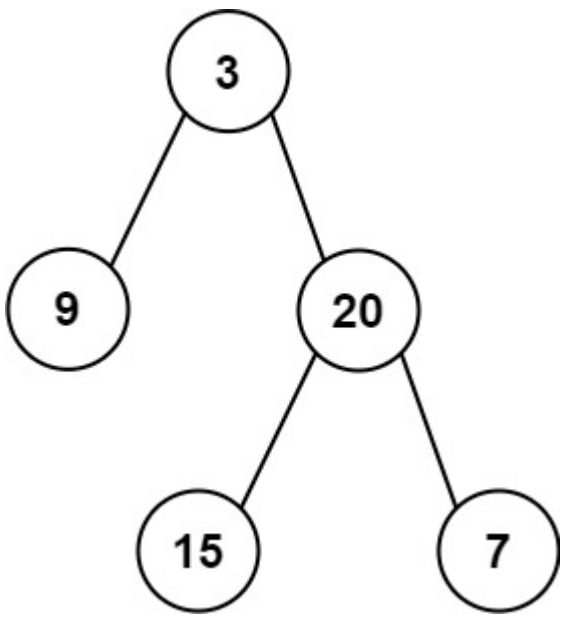
我的笔记：不是很难，只要好好学了上面的求最大深度，这道题只需要保存最大深度的点（同时前序遍历）

---

106. 从中序与后序遍历序列构造二叉树

给定两个整数数组 `inorder` 和 `postorder` ，其中 `inorder` 是二叉树的中序遍历， `postorder` 是同一棵树的后序遍历，请你构造并返回这颗 *二叉树* 。

示例 1:




```
输入：inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
输出：[3,9,20,null,null,15,7]
```

示例 2:

```
输入：inorder = [-1], postorder = [-1]
输出：[-1]
```

提示:

- `1 <= inorder.length <= 3000`
- `postorder.length == inorder.length`
- `-3000 <= inorder[i], postorder[i] <= 3000`
- `inorder` 和 `postorder` 都由 **不同** 的值组成

- `postorder` 中每一个值都在 `inorder` 中
- `inorder` **保证**是树的中序遍历
- `postorder` **保证**是树的后序遍历

题解：

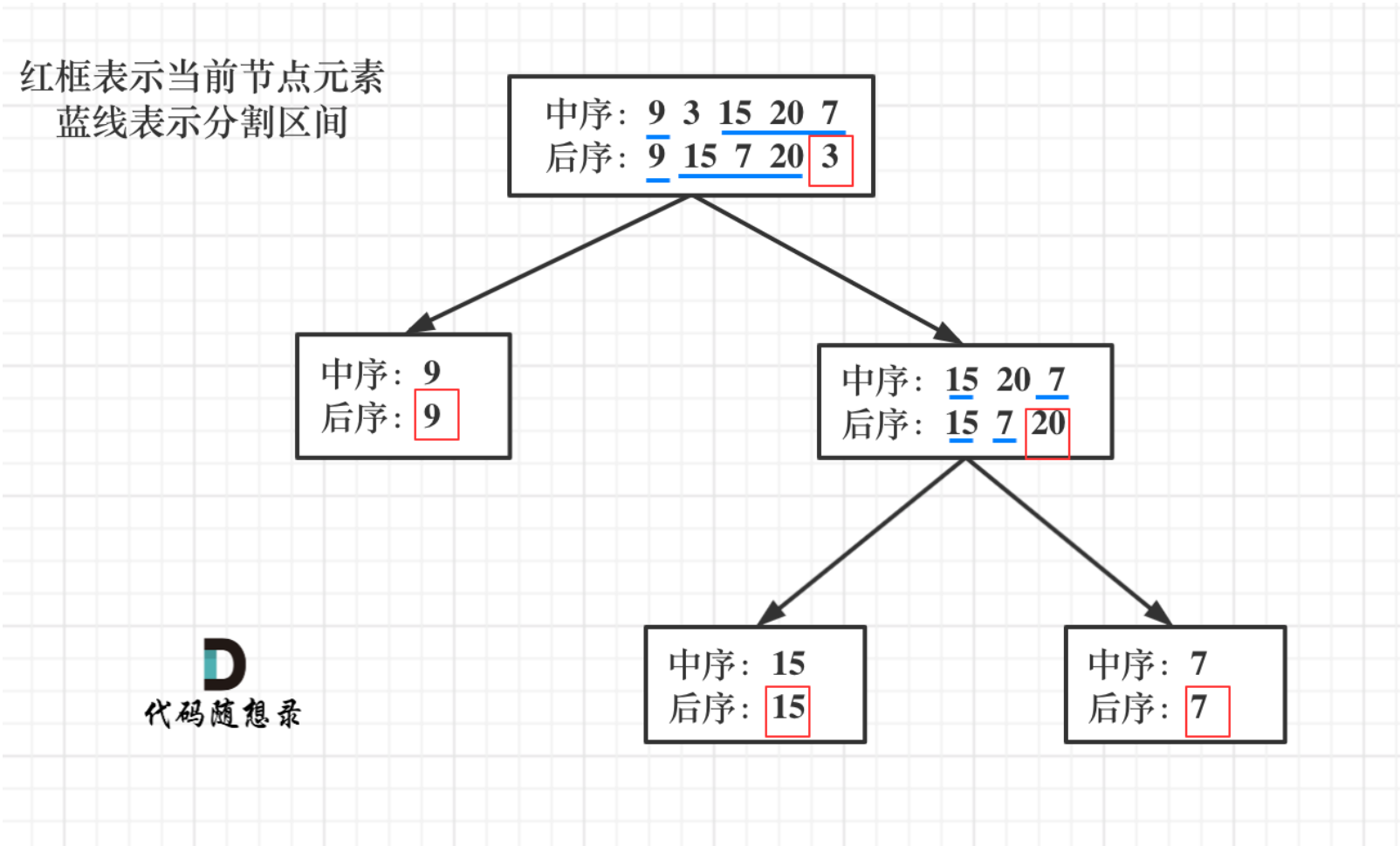
```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if (postorder.empty()) return nullptr; //证明是空树
        int rootNumber = postorder.back();
        TreeNode* root = new TreeNode(rootNumber);  //后序数组的最后一个元素一定是当前的根节点
        if (postorder.size() == 1)  return root;    //如果这是根节点，直接返回
        //接下来开始查找这个节点在中序数组中的位置  并且开始切割中序数组
        auto itRuan = find(inorder.begin(), inorder.end(), rootNumber);
        vector<int> leftInorder(inorder.begin(), itRuan);
        vector<int> rightInorder(itRuan + 1, inorder.end());
        //接下来开始切割后序数组
        vector<int> leftPostorder(postorder.begin(), postorder.begin() + leftInorder.size());
        vector<int> rightPostorder(postorder.begin() + leftInorder.size(), postorder.end() - 1);
        root->left = buildTree(leftInorder, leftPostorder);
        root->right = buildTree(rightInorder, rightPostorder);
        return root;
    }
};
```

思路：

首先回忆一下如何根据两个顺序构造一个唯一的二叉树，相信理论知识大家应该都清楚，就是以 后序数组的最后一个元素为切割点，先切中序数组，根据中序数组，反过来再切后序数组。一层一层切下去，每次后序数组最后一个元素就是节点元素。

如果让我们肉眼看两个序列，画一棵二叉树的话，应该分分钟都可以画出来。

流程如图：



那么代码应该怎么写呢？

说到一层一层切割，就应该想到了递归。

来看一下一共分几步：

- 第一步：如果数组大小为零的话，说明是空节点了。
- 第二步：如果不为空，那么取后序数组最后一个元素作为节点元素。
- 第三步：找到后序数组最后一个元素在中序数组的位置，作为切割点
- 第四步：切割中序数组，切成中序左数组和中序右数组 （顺序别搞反了，一定是先切中序数组）
- 第五步：切割后序数组，切成后序左数组和后序右数组
- 第六步：**递归**处理左区间和右区间

我的笔记：巧啊，妙啊，你他娘的可真是个人才。

---

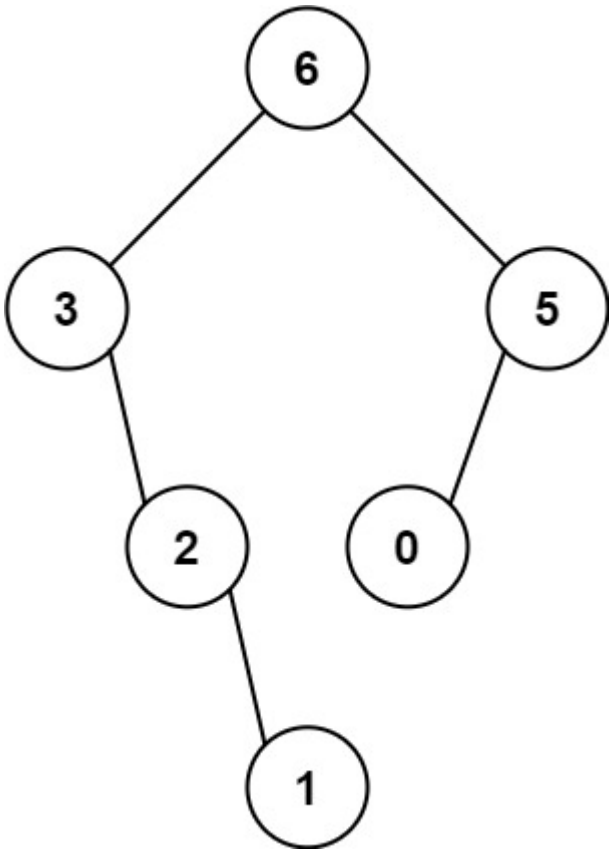
654. 最大二叉树

给定一个不重复的整数数组 `nums` 。 **最大二叉树** 可以用下面的算法从 `nums` 递归地构建:

1. 创建一个根节点，其值为 `nums` 中的最大值。
2. 递归地在最大值 **左边** 的 **子数组前缀上** 构建左子树。
3. 递归地在最大值 **右边** 的 **子数组后缀上** 构建右子树。

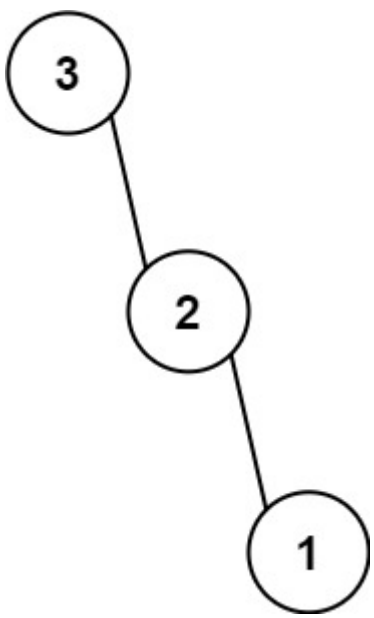
返回 *`nums` 构建的* ***最大二叉树*** 。

示例 1：



输入：nums = [3,2,1,6,0,5]
输出：[6,3,5,null,2,0,null,null,1]
解释：递归调用如下所示：
- [3,2,1,6,0,5] 中的最大值是 6 ，左边部分是 [3,2,1] ，右边部分是 [0,5] 。
    - [3,2,1] 中的最大值是 3 ，左边部分是 [] ，右边部分是 [2,1] 。
        - 空数组，无子节点。
        - [2,1] 中的最大值是 2 ，左边部分是 [] ，右边部分是 [1] 。
            - 空数组，无子节点。
            - 只有一个元素，所以子节点是一个值为 1 的节点。
    - [0,5] 中的最大值是 5 ，左边部分是 [0] ，右边部分是 [] 。
        - 只有一个元素，所以子节点是一个值为 0 的节点。
        - 空数组，无子节点。

示例 2:



```
输入：nums = [3,2,1]
输出：[3,null,2,null,1]
```

提示：

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <= 1000`
- `nums` 中的所有整数 **互不相同**

题解：

```
class Solution {
public:
    TreeNode* leftBulid(vector<int> &nums, int index) {
        if(index==this->maxIndex) return nullptr;
        TreeNode* root = new TreeNode(nums[index]);
        root->left = leftBulid(nums, index - 1);
        return root;
    }

    TreeNode* rightBuild(vector<int> &nums, int index) {
        if (index == this->maxIndex) return nullptr;
        TreeNode* root = new TreeNode(nums[index]);
            root->right = rightBuild(nums, index +1);
            return root;
    }

    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        if (nums.empty()) return nullptr;
         int maxNumber = 0;
        this->maxIndex = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] > maxNumber) {
                maxNumber = nums[i];
                this ->maxIndex = i;
            }
        }
        TreeNode* root = new TreeNode(maxNumber);
        if(this->maxIndex>0) root->left = rightBuild(nums, 0);
        if(this->maxIndex<nums.size()) root->right = leftBulid(nums, nums.size()-1);
        return root;
    }
public:
    int maxIndex;
};
```

我的笔记：我不知道错哪儿了，反正我能跑，代码也能跑

98. 验证二叉搜索树

给你一个二叉树的根节点 `root` ，判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

- 节点的左

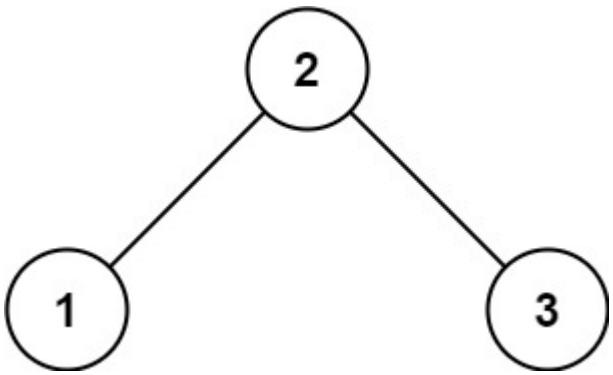
  子树

  只包含

  小于

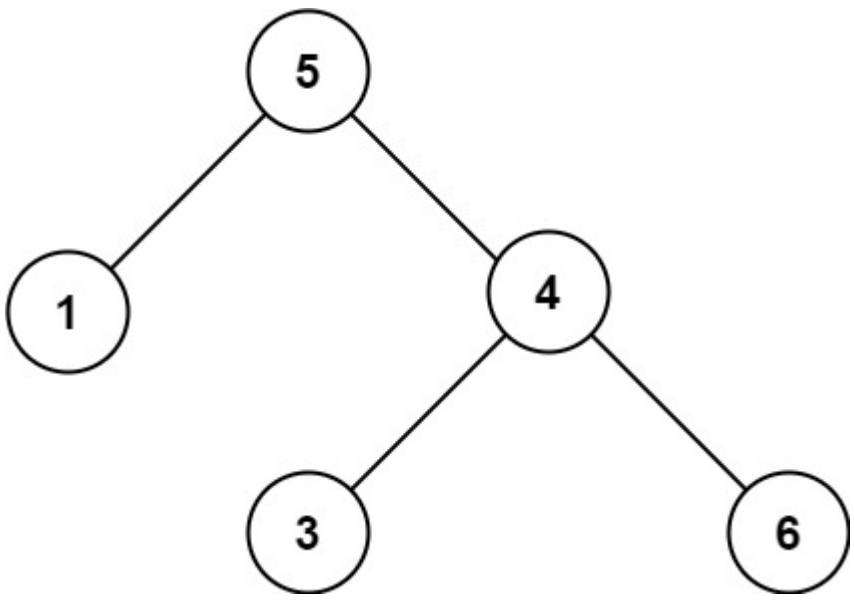
  当前节点的数。
- 节点的右子树只包含 **大于** 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1：




```
输入：root = [2,1,3]
输出：true
```

示例 2：




```
输入：root = [5,1,4,null,null,3,6]
输出：false
解释：根节点的值是 5 ，但是右子节点的值是 4 。
```

提示：

- 树中节点数目范围在`[1, 104]`内
- `-231 <= Node.val <= 231 - 1`

题解：

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {  //使用中序遍历来排序
        if (!root) return true;
        if (!root->left && !root->right) return true;
        stack<TreeNode*> ruanStack;
        long long tempMax = 999;
        TreeNode* ruan = root;
        while (ruan!=nullptr||!ruanStack.empty()) {
            if (ruan!=nullptr) {
                ruanStack.push(ruan);
                ruan = ruan->left;
            }
            else
            {
                ruan = ruanStack.top();
                ruanStack.pop();
                if (tempMax == 999) tempMax = ruan->val - 1LL;   //面向结果答题
                if(tempMax >= ruan->val) return false;
                    else tempMax = ruan->val;
                ruan = ruan->right;

            }

        }
        return true;
    }
};
```

我的笔记：这道题就是将通过中序遍历树，然后进行最大值的比较

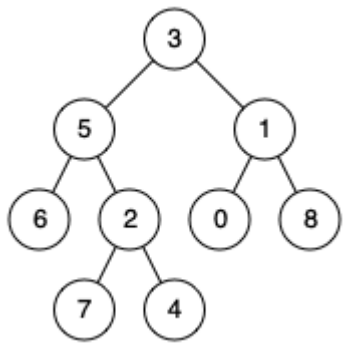
---

236. 二叉树的最近公共祖先（important）

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

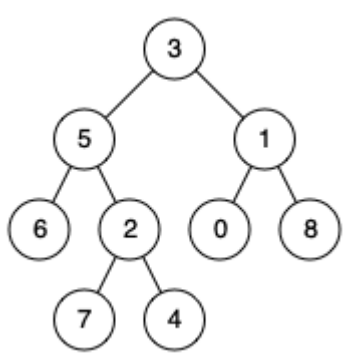
百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

示例 1：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出：3
解释：节点 5 和节点 1 的最近公共祖先是节点 3 。
```

示例 2：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出：5
解释：节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。
```

示例 3：

```
输入：root = [1,2], p = 1, q = 2
输出：1
```

提示：

- 树中节点数目在范围 `[2, 105]` 内。
- `-109 <= Node.val <= 109`
- 所有 `Node.val` `互不相同` 。
- `p != q`
- `p` 和 `q` 均存在于给定的二叉树中。

题解：

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == q || root == p || root == NULL) return root;
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left != NULL && right != NULL) return root;
        if (left == NULL) return right;
        return left;
    }
};
```

1. 求最小公共祖先，需要从底向上遍历，那么二叉树，只能通过后序遍历（即：回溯）实现从底向上的遍历方式。
2. 在回溯的过程中，必然要遍历整棵二叉树，即使已经找到结果了，依然要把其他节点遍历完，因为要使用递归函数的返回值（也就是代码中的left和right）做逻辑判断。
3. 要理解如果返回值left为空，right不为空为什么要返回right，为什么可以用返回right传给上一层结果。

可以说这里每一步，都是有难度的，都需要对二叉树，递归和回溯有一定的理解。

本题没有给出迭代法，因为迭代法不适合模拟回溯的过程。理解递归的解法就够了。

我的笔记：之后一定要多看多理解这道题。

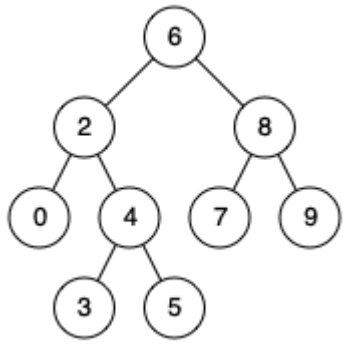
---

235. 二叉搜索树的最近公共祖先

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

示例 2:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

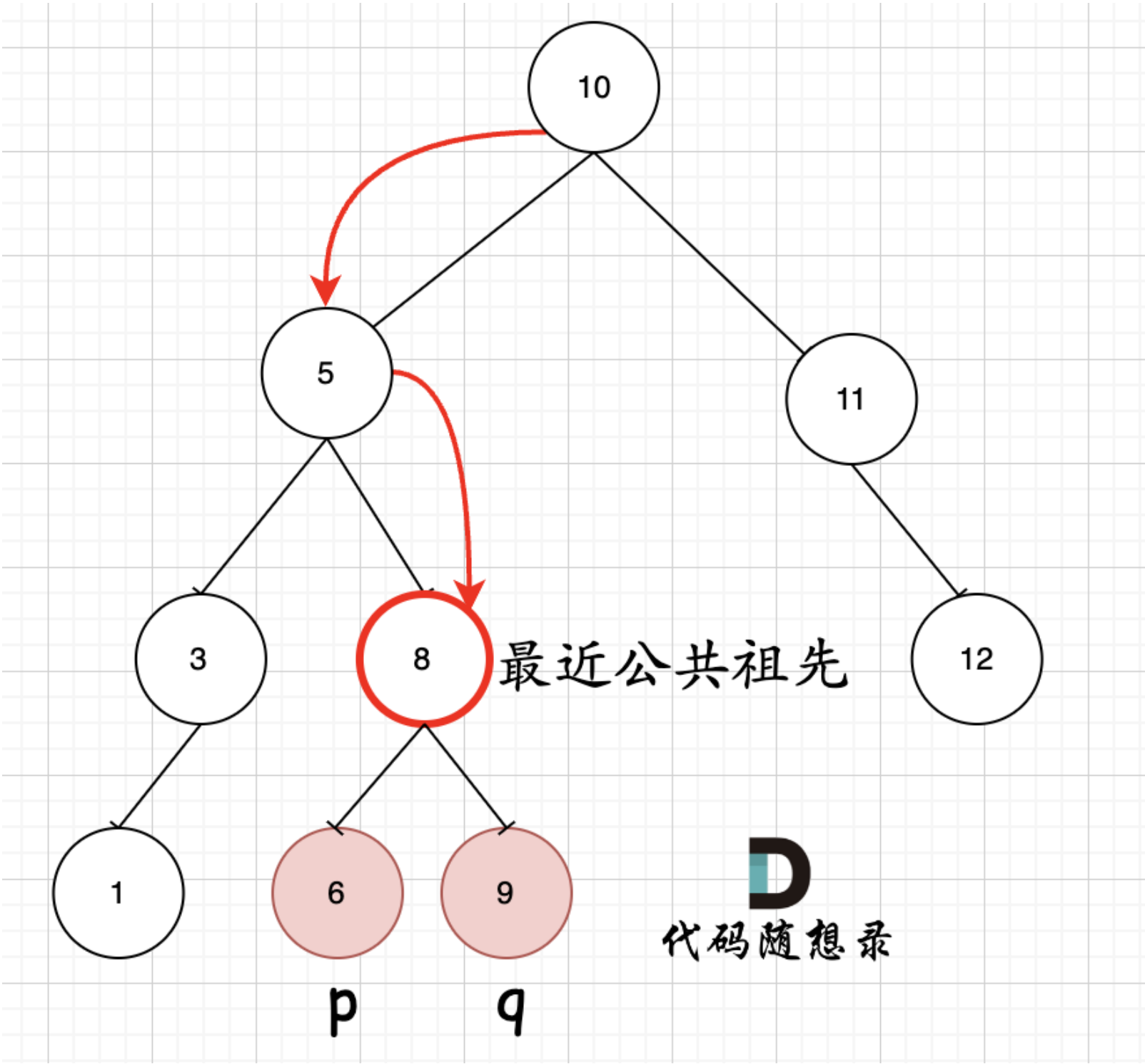
说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

题解:

```cpp
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root) return nullptr;
        if (root->val>p->val&&root->val>q->val) return lowestCommonAncestor(root->left, p, q);
        else if (root->val < p->val && root->val < q->val)  return lowestCommonAncestor(root->right, p, q);
        else  return root;
    }
};
```

思路：



我的笔记：这道题甚至没有设计前中后序的排列。如果根节点小于所有的比较节点，那么就往右寻找，如果大于所有的比较节点，那么就往左寻找，最近的公共祖先节点的值一定是在 [p->val,q->val]之间的

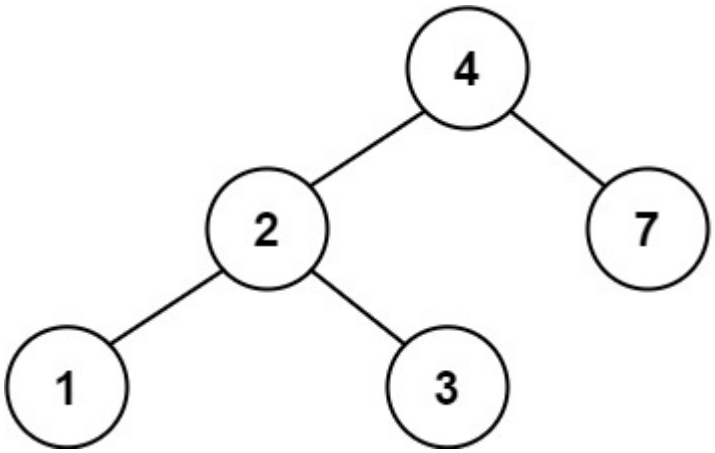
---

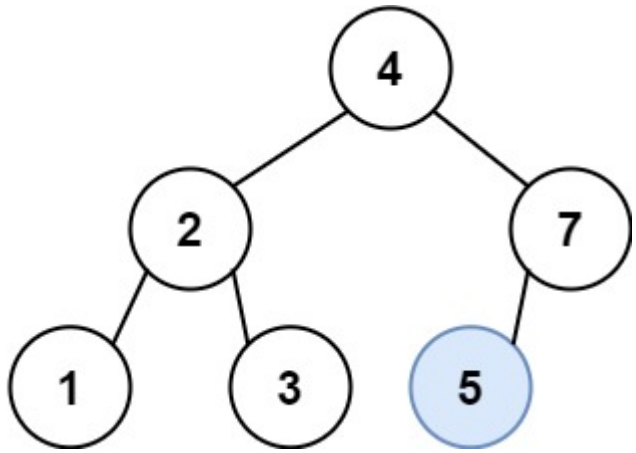
701. 二叉搜索树中的插入操作

给定二叉搜索树（BST）的根节点 `root` 和要插入树中的值 `value` ，将值插入二叉搜索树。 返回插入后二叉搜索树的根节点。 输入数据 **保证** ，新值和原始二叉搜索树中的任意节点值都不同。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。 你可以返回 **任意有效的结果** 。

示例 1：






```
输入：root = [4,2,7,1,3], val = 5
输出：[4,2,7,1,3,5]
解释：另一个满足题目要求可以通过的树是：
```

示例 2：

```
输入：root = [40,20,60,10,30,50,70], val = 25
输出：[40,20,60,10,30,50,70,null,null,25]
```

示例 3：

```
输入：root = [4,2,7,1,3,null,null,null,null,null,null], val = 5
输出：[4,2,7,1,3,5]
```

提示：

- 树中的节点数将在 `[0, 104]` 的范围内。
- `-108 <= Node.val <= 108`
- 所有值 `Node.val` 是 **独一无二** 的。
- `-108 <= val <= 108`
- **保证** `val` 在原始BST中不存在。

题解：

```cpp
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if (!root)return new TreeNode(val);
        if (root->val > val) root->left = insertIntoBST(root->left, val);
        else if (root->val < val) root->right = insertIntoBST(root->right, val);
        return root;
    }
};
```

我的笔记：很简单的一道题

---

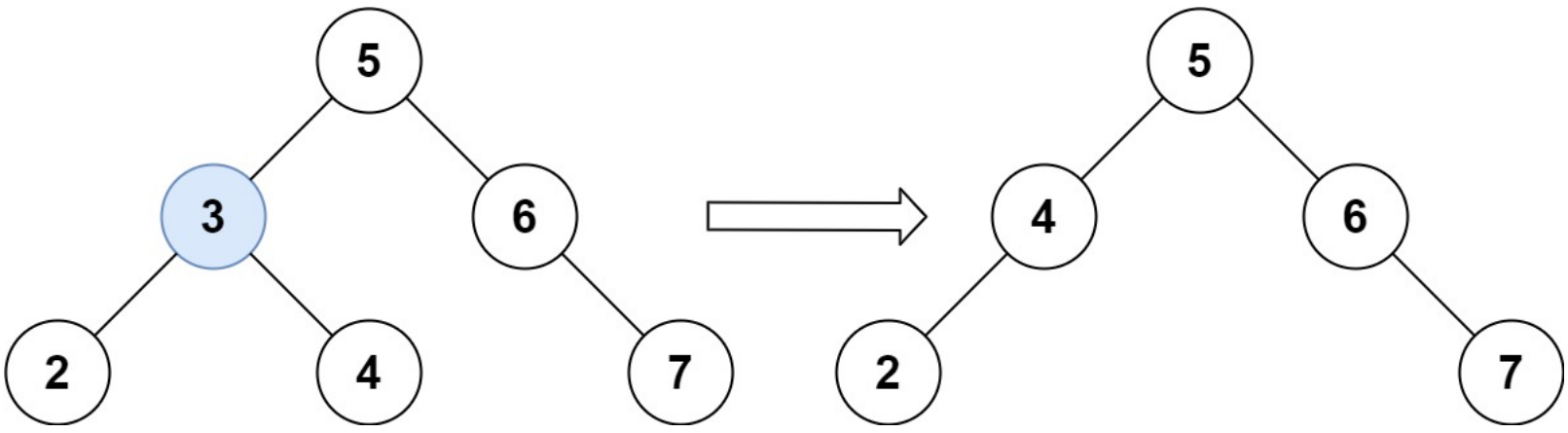
450. 删除二叉搜索树中的节点

给定个二叉搜索树的根节点 **root** 和一个值 **key**，删除二叉搜索树中的 **key** 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

1. 首先找到需要删除的节点；
2. 如果找到了，删除它。

示例 1:




```
输入：root = [5,3,6,2,4,null,7], key = 3
输出：[5,4,6,2,null,null,7]
解释：给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。
一个正确的答案是 [5,4,6,2,null,null,7], 如下图所示。
另一个正确答案是 [5,2,6,null,4,null,7]。
```

示例 2:

```
输入: root = [5,3,6,2,4,null,7], key = 0
输出: [5,3,6,2,4,null,7]
解释: 二叉树不包含值为 0 的节点
```

示例 3:

```
输入: root = [], key = 0
输出: []
```

提示:

- 节点数的范围 `[0, 104]`.
- `-105 <= Node.val <= 105`
- 节点值唯一
- `root` 是合法的二叉搜索树
- `-105 <= key <= 105`

进阶： 要求算法时间复杂度为 O(h)，h 为树的高度。

题解：

```
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if (root == nullptr) return root; // 第一种情况：没找到删除的节点，遍历到空节点直接返回了
        if (root->val == key) {
            // 第二种情况：左右孩子都为空（叶子节点），直接删除节点， 返回NULL为根节点
            if (root->left == nullptr && root->right == nullptr) {
                ///! 内存释放
                delete root;
                return nullptr;
            }
            // 第三种情况：其左孩子为空，右孩子不为空，删除节点，右孩子补位 ，返回右孩子为根节点
            else if (root->left == nullptr) {
                auto retNode = root->right;
                ///! 内存释放
                delete root;
                return retNode;
            }
            // 第四种情况：其右孩子为空，左孩子不为空，删除节点，左孩子补位，返回左孩子为根节点
            else if (root->right == nullptr) {
                auto retNode = root->left;
                ///! 内存释放
                delete root;
                return retNode;
            }
            // 第五种情况：左右孩子节点都不为空，则将删除节点的左子树放到删除节点的右子树的最左面节点的左孩子的位置
            // 并返回删除节点右孩子为新的根节点。
            else {
                TreeNode* cur = root->right; // 找右子树最左面的节点
                while(cur->left != nullptr) {
                    cur = cur->left;
                }
                cur->left = root->left; // 把要删除的节点（root）左子树放在cur的左孩子的位置
                TreeNode* tmp = root;   // 把root节点保存一下，下面来删除
                root = root->right;     // 返回旧root的右孩子作为新root
                delete tmp;             // 释放节点内存（这里不写也可以，但C++最好手动释放一下吧）
                return root;
            }
        }
        if (root->val > key) root->left = deleteNode(root->left, key);
        if (root->val < key) root->right = deleteNode(root->right, key);
        return root;
    }
};
```

我的笔记：

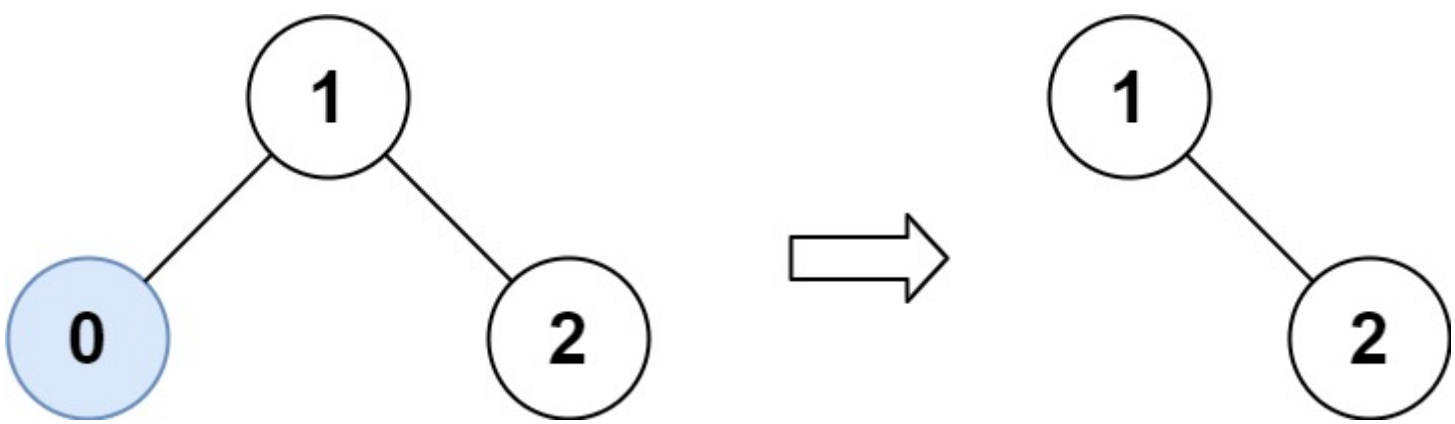
---

669. 修剪二叉搜索树

给你二叉搜索树的根节点 `root` ，同时给定最小边界`low` 和最大边界 `high`。通过修剪二叉搜索树，使得所有节点的值在`[low, high]`中。修剪树 **不应该** 改变保留在树中的元素的相对结构 (即，如果没有被移除，原有的父代子代关系都应当保留)。 可以证明，存在 **唯一的答案** 。

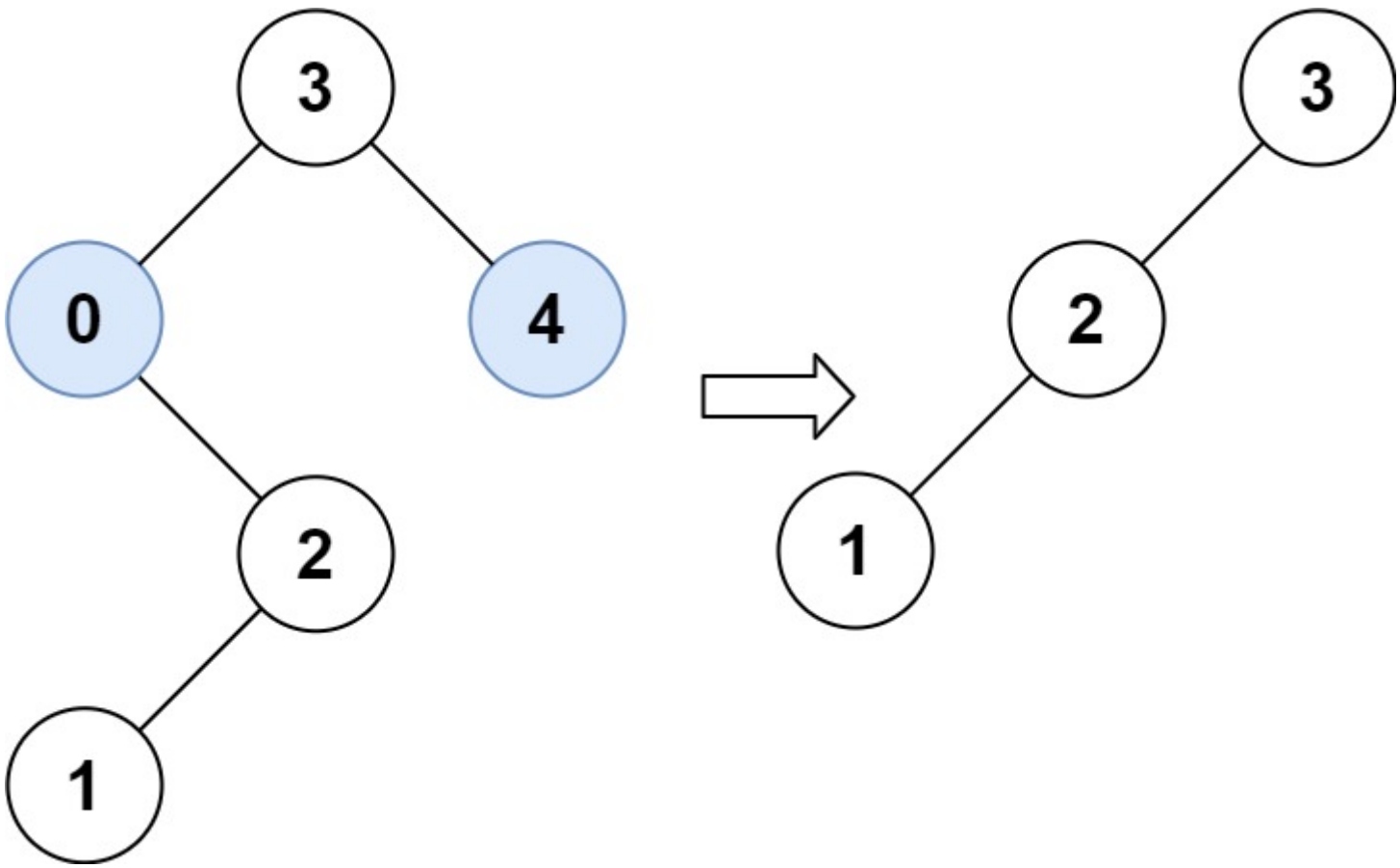
所以结果应当返回修剪好的二叉搜索树的新的根节点。注意，根节点可能会根据给定的边界发生改变。

示例 1：




```
输入：root = [1,0,2], low = 1, high = 2
输出：[1,null,2]
```

示例 2：




```
输入：root = [3,0,4,null,2,null,null,1], low = 1, high = 3
输出：[3,2,null,1]
```

提示：

- 树中节点数在范围 `[1, 104]` 内
- `0 <= Node.val <= 104`
- 树中每个节点的值都是 **唯一** 的
- 题目数据保证输入是一棵有效的二叉搜索树
- `0 <= low <= high <= 104`

题解：

递归：

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        if (root == nullptr) return nullptr;
        if (root->val < low) return trimBST(root->right, low, high);
        if (root->val > high) return trimBST(root->left, low, high);
        root->left = trimBST(root->left, low, high);
        root->right = trimBST(root->right, low, high);
        return root;
    }
};
```

迭代：

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int L, int R) {
        if (!root) return nullptr;

        // 处理头结点，让root移动到[L, R] 范围内，注意是左闭右闭
        while (root != nullptr && (root->val < L || root->val > R)) {
            if (root->val < L) root = root->right; // 小于L往右走
            else root = root->left; // 大于R往左走
        }
        TreeNode *cur = root;
        // 此时root已经在[L, R] 范围内，处理左孩子元素小于L的情况
        while (cur != nullptr) {
            while (cur->left && cur->left->val < L) {
                cur->left = cur->left->right;
            }
            cur = cur->left;
        }
        cur = root;

        // 此时root已经在[L, R] 范围内，处理右孩子大于R的情况
        while (cur != nullptr) {
            while (cur->right && cur->right->val > R) {
                cur->right = cur->right->left;
            }
            cur = cur->right;
        }
```

```
        return root;
    }
};
```

我的笔记：

---

538. 把二叉搜索树转换为累加树

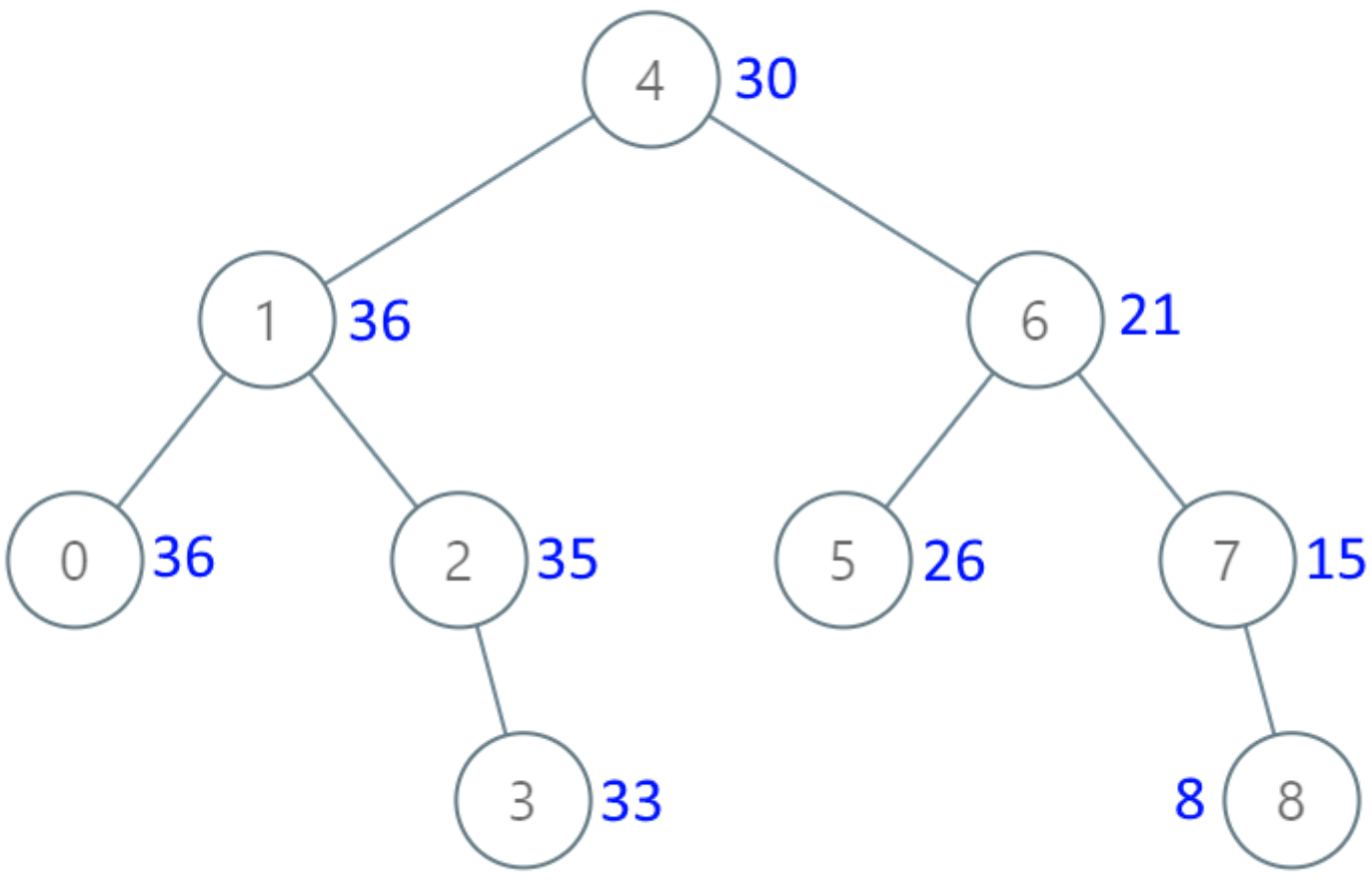
给出二叉 **搜索** 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

提醒一下，二叉搜索树满足下列约束条件：

- 节点的左子树仅包含键 **小于** 节点键的节点。
- 节点的右子树仅包含键 **大于** 节点键的节点。
- 左右子树也必须是二叉搜索树。

注意：本题和 1038: https://leetcode-cn.com/problems/binary-search-tree-to-greater-sum-tree/ 相同

示例 1：




```
输入：[4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出：[30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

示例 2：

```
输入：root = [0,null,1]
输出：[1,null,1]
```

示例 3：

```
输入：root = [1,0,2]
输出：[3,3,2]
```

示例 4：

```
输入：root = [3,2,4,1]
输出：[7,9,4,10]
```

提示：

- 树中的节点数介于 `0` 和 `104` 之间。
- 每个节点的值介于 `-104` 和 `104` 之间。
- 树中的所有值 **互不相同** 。
- 给定的树为二叉搜索树。

题解：

```cpp
class Solution {
private:
    int pre = 0; // 记录前一个节点的数值
    void traversal(TreeNode* cur) { // 右中左遍历
        if (cur == NULL) return;
        traversal(cur->right);
        cur->val += pre;
        pre = cur->val;
        traversal(cur->left);
    }
public:
    TreeNode* convertBST(TreeNode* root) {
        pre = 0;
        traversal(root);
        return root;
    }
};
```

我的：

```cpp
class Solution {
private:
    void changeTree(int tempNumber) {
        if (this->ruanStack.empty()) return;
        this->ruanStack.top()->val = this->ruanStack.top()->val+ tempNumber;
        tempNumber = this->ruanStack.top()->val;
        this->ruanStack.pop();
        changeTree(tempNumber);
    }
    TreeNode* buildStack(TreeNode* root) {
```

```
        if (root->left) root->left = buildStack(root->left);
        this->ruanStack.push(root);
        if (root->right) root->right = buildStack(root->right);
        return root;
    }
public:
    TreeNode* convertBST(TreeNode* root) {
        if (!root) return nullptr;
        TreeNode* aws = buildStack(root);
        int k = this->ruanStack.top()->val;
        this->ruanStack.pop();
        changeTree(k);
        return aws;
    }
private:
    stack<TreeNode*> ruanStack;
};
```

我的笔记：正确做法：直接反中序遍历。我的做法：正常中序遍历，然后把每一个元素保存到一个栈中，然后依次弹出修改。（慢了）

困难难度

239. 滑动窗口最大值

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回 *滑动窗口中的最大值* 。

示例 1：

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

示例 2：

```
输入：nums = [1], k = 1
输出：[1]
```

提示：

- `1 <= nums.length <= 105`

- `-104 <= nums[i] <= 104`
- `1 <= k <= nums.length`

题解：

```
class Solution {
private:
    class MyQueue { //单调队列（从大到小）
    public:
        deque<int> que; // 使用deque来实现单调队列
        // 每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。
        // 同时pop之前判断队列当前是否为空。
        void pop(int value) {
            if (!que.empty() && value == que.front()) {
                que.pop_front();
            }
        }
        // 如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口元素的数值为止。
        // 这样就保持了队列里的数值是单调从大到小的了。
        void push(int value) {
            while (!que.empty() && value > que.back()) {
                que.pop_back();
            }
            que.push_back(value);

        }
        // 查询当前队列里的最大值 直接返回队列前端也就是front就可以了。
        int front() {
            return que.front();
        }
    };
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        MyQueue que;
        vector<int> result;
        for (int i = 0; i < k; i++) { // 先将前k的元素放进队列
            que.push(nums[i]);
        }
        result.push_back(que.front()); // result 记录前k的元素的最大值
        for (int i = k; i < nums.size(); i++) {
            que.pop(nums[i - k]); // 滑动窗口移除最前面元素
            que.push(nums[i]); // 滑动窗口前加入最后面的元素
            result.push_back(que.front()); // 记录对应的最大值
        }
        return result;
    }
};
```

笔记：本题目是使用栈和队列解决的经典问题。

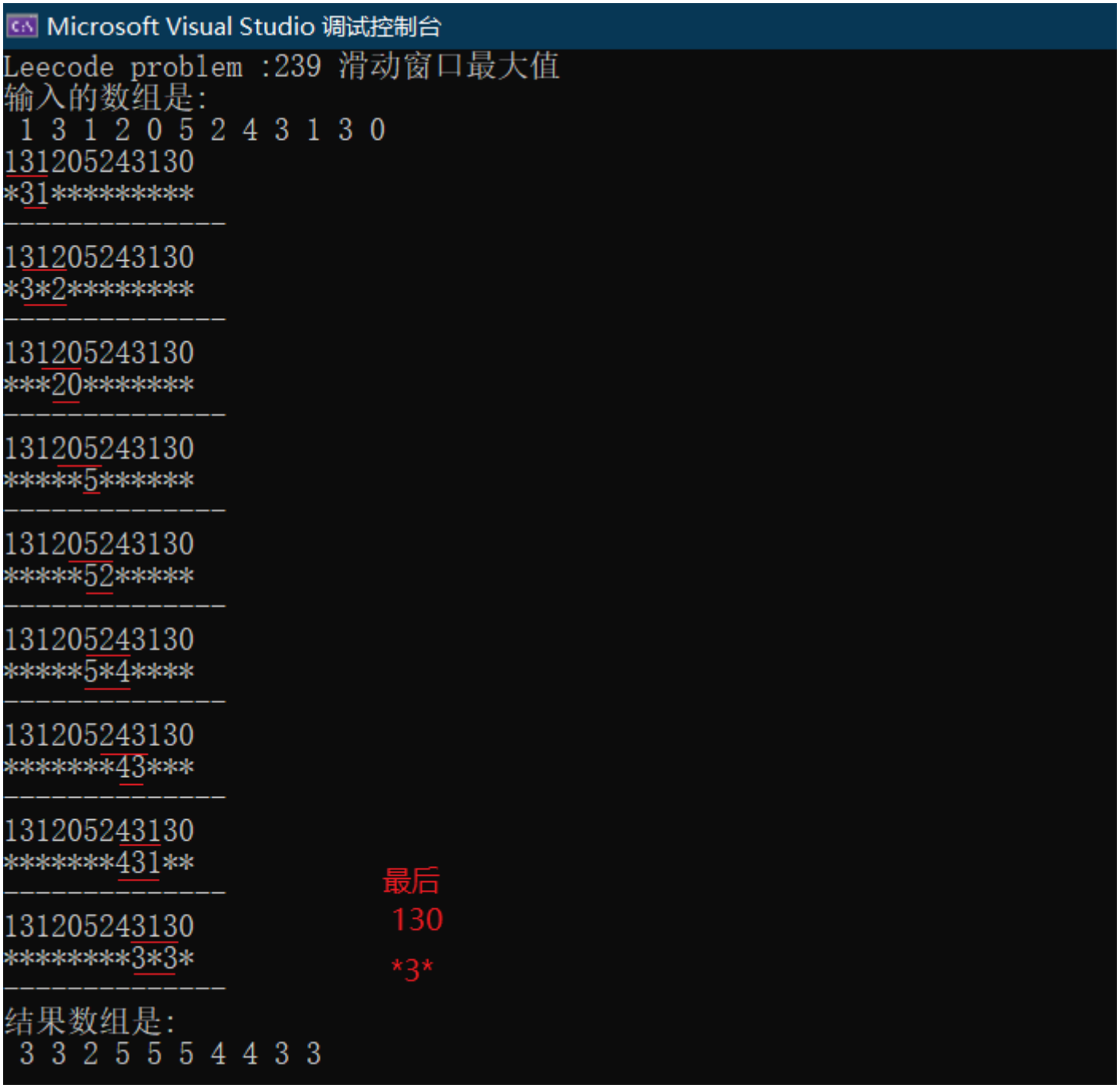
这道题目的思路是使用**双端队列deque**解决(因为要从队尾弹出元素)

其实队列没有必要维护窗口里的所有元素，只需要维护有可能成为窗口里最大值的元素就可以了，同时保证队列里的元素数值是由大到小的。

那么这个维护元素单调递减的队列就叫做**单调队列，即单调递减或单调递增的队列。C++中没有直接支持单调队列，需要我们自己来实现一个单调队列**

不要以为实现的单调队列就是 对窗口里面的数进行排序，如果排序的话，那和优先级队列又有什么区别了呢。

请看代码的详细示例：



所以所做的事情，就是维护一个窗口队列，它的内容按照降序排列，第一个永远是最大的，当后面新增一个数的时候，就会和队列末端（也就是当前队列最小的）比较，如果比它大，则弹出最后那个，如果比它还小，就弹出，最后还是要加上末尾的数字。然后前端稳定弹出当前队列中最大的，并存放到结果数组中。