

# 第一周：代码随想录

## 数组部分

数组是存放在连续的内存空间上的相同类型数据的集合

数组的内存空间是连续的

正是因为数组的在内存空间的地址是连续的，所以我们在删除或者增添元素的时候，就难免要移动其他元素的地址。

Vector和array的区别：

严格来说vector是容器，不是数组

## 二分法

二分查找

左闭右闭和左闭右开

使用二分法的前提：**前提是数组为有序数组**，同时题目还强调**数组中无重复元素**，因为一旦有重复元素，使用二分查找法返回的元素下标可能不是唯一的

## 快慢指针&双指针

移除元素，有序元素的平凡

## 滑动窗口法

还需要做题

长度最小的子数组

## 螺旋矩阵

```
vector<vector<int>> res(n, vector<int>(n, 0));
```

`vector<vector<int>> res;`：声明了一个名为 `res` 的二维vector。

- `(n, vector<int>(n, 0))`：这是vector的构造函数调用。在这里，它创建了一个包含n个元素的vector，每个元素都是一个包含n个整数的vector，并且这些整数都初始化为0。
- 因此，`res` 是一个n行n列的二维矩阵，其中所有的元素都初始化为0。

保证严格的左闭右开原则，就不会乱套

# 第二周：代码随想录

## 链表

单链表中的指针域只能指向节点的下一个节点

双链表：每一个节点有两个指针域，一个指向下一个节点，一个指向上一个节点。

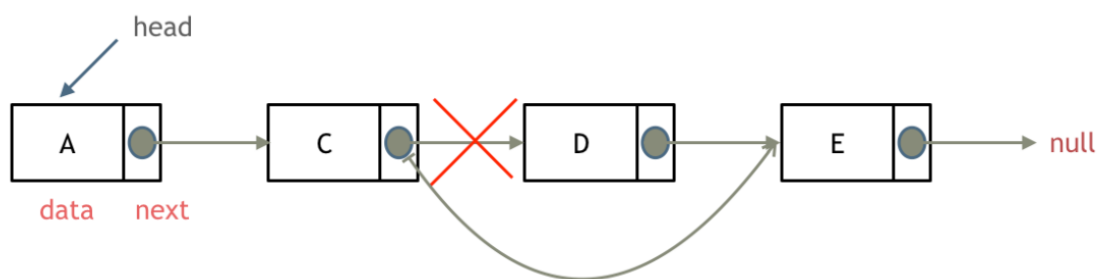
双链表 既可以向前查询也可以向后查询。

链表中的节点在内存中不是连续分布的，而是散乱分布在内存中的某地址上，分配机制取决于操作系统的内存管理。

### 面试时要自己手写链表的定义

c++里的定义链表方式：

```
// 单链表
struct ListNode {
    int val; // 节点上存储的元素
    ListNode *next; // 指向下一个节点的指针
    ListNode(int x) : val(x), next(NULL) {} // 节点的构造函数
};
```



只要将C节点的next指针 指向E节点就可以了。

那有同学说了，D节点不是依然存留在内存里么？只不过是没在这个链表里而已。

是这样的，所以在C++里最好是再手动释放这个D节点，释放这块内存。

Java、Python，就有自己的内存回收机制，就不用自己手动释放了

### 移除链表元素

移除头结点和移除其他节点的操作是不一样的

那么可不可以 以一种统一的逻辑来移除 链表的节点呢。

其实可以设置一个虚拟头结点，这样原链表的所有节点就都可以按照统一的方式进行移除了

### 反转链表

不需要再定义一个新的链表，只需要改变链表的next指针的指向

首先定义一个cur指针，指向头结点，再定义一个pre指针，初始化为null

把 cur->next 节点用tmp指针保存

将cur->next 指向pre，此时已经反转了第一个节点了。之后循环移动pre和cur指针

迭代是通过循环结构来实现的，和递归不同

## 两两交换链表中的节点

使用虚拟头节点会方便很多，因为不用额外处理头节点的逻辑

其中while循环的判定是`while(cur->next != nullptr && cur->next->next != nullptr)`

## 删除链表中的倒数第N个节点

依旧使用虚拟头节点，使用fast和slow节点

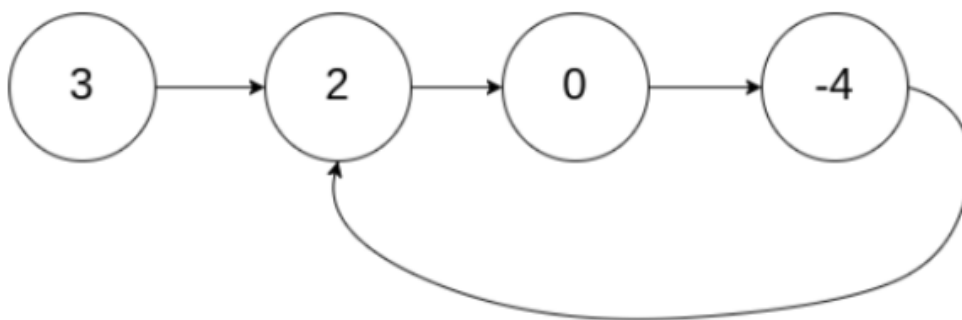
fast从dummyhead先走n+1步，然后slow，fast一起走，fast走到null即结尾时，slow正好走到倒数n+1个，即n的前一个

最后return head是错误的，当测试用例是[1]并且n=1时，会把1删掉，也就是head指针被删除了，直接return dummyhead->next才是正确的

## 环形链表

判断链表中是否有环，并且找到环的入口

因为是链表，所以如果有环一定呈现下面的样子

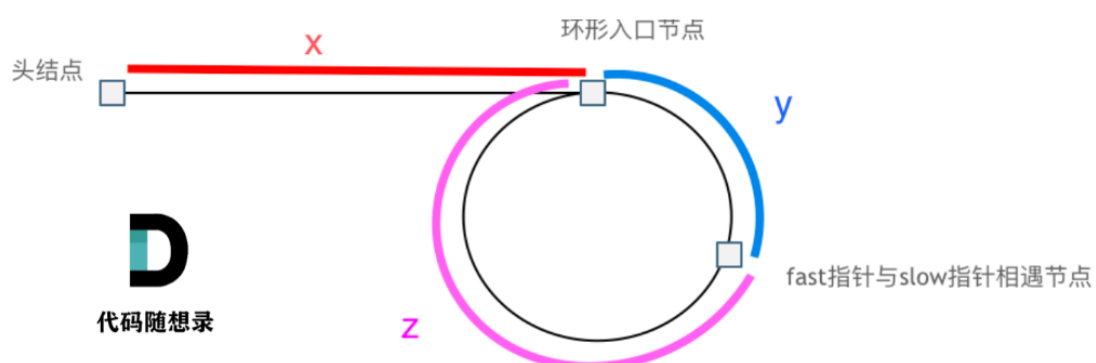


即环的后面不会再有节点了

使用fast和slow节点（fast走两步，slow1步）

如果有环，fast和slow一定会相遇，相当于fast追逐slow，至此可以判断出有没有环

找环的入口（需要数学证明）



那么相遇时：slow指针走过的节点数为： $x + y$ ，fast指针走过的节点数为： $x + y + n(y + z)$ ，n为fast指针在环内走了n圈才遇到slow指针， $(y+z)$ 为一圈内节点的个数A。

$$(x + y) * 2 = x + y + n(y + z)$$

两边消掉一个  $(x+y)$  :  $x + y = n (y + z)$

因为要找环形的入口，那么要求的是 $x$ ，因为 $x$ 表示 头结点到 环形入口节点的距离。

所以要求 $x$ ，将 $x$ 单独放在左面：  $x = n (y + z) - y$  ,

再从 $n(y+z)$ 中提出一个  $(y+z)$  来，整理公式之后为如下公式：  $x = (n - 1) (y + z) + z$  注意这里 $n$ 一定是大于等于1的，因为 fast指针至少要多走一圈才能相遇slow指针。

当  $n$ 为1的时候，公式就化解为  $x = z$  ,

这就意味着，从**头结点**出发一个指针，从**相遇节点**也出发一个指针，这两个指针每次只走一个节点，那么当这两个指针相遇的时候就是环形入口的节点

如果 $n$ 大于1情况也是一样的，不过就是index1在环里多转了  $(n-1)$  圈

## 哈希表

又叫散列表

在做面试题目的时候遇到需要**判断一个元素是否出现过的**场景也应该第一时间想到哈希法

哈希法是**牺牲了空间换取了时间**，因为我们要使用额外的数组，set或者是map来存放数据

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	$O(\log n)$	$O(\log n)$
std::multiset	红黑树	有序	是	否	$O(\log n)$	$O(\log n)$
std::unordered_set	哈希表	无序	否	否	$O(1)$	$O(1)$

std::unordered\_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	O(logn)	O(logn)
std::multimap	红黑树	key有序	key可重复	key不可修改	O(log n)	O(log n)
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

std::unordered\_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered\_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

那么再来看一下map，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。

## 题1

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。

示例 1: 输入: s = "anagram", t = "nagaram" 输出: true

示例 2: 输入: s = "rat", t = "car" 输出: false

说明: 你可以假设字符串只包含小写字母。

解:

因为是看他的出现次数——想到哈希表，而他指定了是26个字母之内，所以可以用数组（因为限制大小了）

此外，直接用字符串和'a'之间的相对差作数组下标即可，并不需要知道ASCII码

## 字符串

swap可以有两种实现。

一种就是常见的交换数值：

```
int tmp = s[i];
s[i] = s[j];
s[j] = tmp;
```

一种就是通过位运算：

```
s[i] ^= s[j];
s[j] ^= s[i];
s[i] ^= s[j];
```

如果是删除数组/字符串中的，如果是常规的从前往后删，时间复杂度为 $O(n^2)$

为了满足时间复杂度为 $O(n)$ ，且为原地改变，使用双指针

**如果是将某一个元素变为需要用更多元素容器装的，比如1变为number，那数组长度变大，则双指针从后往前替换， $s[fast]=s[slow]$ ，最后记得j减小**

**如果是删除某个元素，如空格或特定值，则用双指针的从前往后替换，for循环里令 $s[slow++]=s[fast]$**

## 151.翻转字符串里的单词

力扣题目链接 [🔗](#)

给定一个字符串，逐个翻转字符串中的每个单词。

示例 1:

输入: "the sky is blue"

输出: "blue is sky the"

示例 2:

输入: " hello world! "

输出: "world! hello"

解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。

示例 3:

输入: "a good example"

输出: "example good a"

解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

$s[fast]s[fast-1]\&\&s[fast]$  不可以连写，要写成两个条件

**思路:**

第一步：消除空格——和消除数组中消除特定元素是一样的——用双指针，原地消除，并且 $O(n)$ 的时间复杂度

第二步：全部reverse

第三步：遇到空格reverse单词

# 右旋字符串

卡码网[题目链接](#)

字符串的右旋转操作是把字符串尾部的若干个字符转移到字符串的前面。给定一个字符串  $s$  和一个正整数  $k$ ，请编写一个函数，将字符串中的后面  $k$  个字符移到字符串的前面，实现字符串的右旋转操作。

例如，对于输入字符串 "abcdefg" 和整数 2，函数应该将其转换为 "fgabcde"。

输入：输入共包含两行，第一行为一个正整数  $k$ ，代表右旋转的位数。第二行为字符串  $s$ ，代表需要旋转的字符串。

输出：输出共一行，为进行了右旋转操作后的字符串。

样例输入：

```
1 2
2 abcdefg
```

样例输出：

```
1 fgabcde
```

数据范围：  $1 \leq k < 10000$ ,  $1 \leq s.length < 10000$ ;

这个题就是上面一个题的简单版，只需要将他全部翻转，再按  $k$  局部反转就可以了

## KMP算法

实现字符串的匹配问题

首先获得next数组，然后用该数组进行匹配

## 时间复杂度分析

其中  $n$  为文本串长度， $m$  为模式串长度，因为在匹配的过程中，根据前缀表不断调整匹配的位置，可以看出匹配的过程是  $O(n)$ ，之前还要单独生成next数组，时间复杂度是  $O(m)$ 。所以整个KMP算法的时间复杂度是  $O(n+m)$  的。

暴力的解法显而易见是  $O(n \times m)$ ，所以KMP在字符串匹配中极大地提高了搜索的效率。

题目：

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`。

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。

解答:

直接背吧。。

```
class Solution {
public:
    //getNext函数
    void getNext(int *next,const string&s){
        int j=0;
        next[0]=0;
        for(int i=1;i<s.size();i++){
            while(j>0&&s[i]!=s[j]){
                j=next[j-1];
            }
            if(s[i]==s[j])
            {
                j++;
            }
            next[i]=j;
        }
    }

    int strStr(string haystack, string needle) {
        if(needle.size()==0){
            return 0;
        }
        int next[needle.size()];
        getNext(next,needle);
        //开始基于next数组的匹配
        int j=0;
        for (int i = 0; i < haystack.size(); i++) {
            while(j > 0 && haystack[i] != needle[j]) {
                j = next[j - 1];
            }
            if(haystack[i]==needle[j]){
                j++;
            }
            //检查j的大小是否等于needle.size(), 如果等于就匹配结束了
            if(j==needle.size())
                return (i-needle.size()+1);
        }
        return -1;
    }
};
```

衍生题:

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 = "abab"`

输出: `true`

解释: 可由子串 `"ab"` 重复两次构成。



解:

匹配字符串, KMP, 但是运用到数学推理公式

```
//是next[i]=j!!!!!!!!!!!!!!!!!!!!!!  
//不要再写成next[j]=j了  
    next[i]=j;
```

```
//这里是next数组来解题, 和你s没有什么关系  
//到底用哪个数组请分清  
if(next[len-1]!=0&&len%(len-(next[len-1]))==0)
```

## 第三周

开始写150题里面的题了

## 双指针法

### 回文判断

我被0P害惨啦! 有没有可能你是用的大小写ascii差32, 0和P刚好也差32!  
背一下ASCII表吧

### 判断子序列

双指针

### 两数之和

题目说了是非递归顺序排列。双指针不断判断大小向内收敛

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 **非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则  $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$ 。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 **只对应唯一的答案**，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

### 示例 1:

**输入:** `numbers = [2, 7, 11, 15]`, `target = 9`

**输出:** `[1, 2]`

**解释:** 2 与 7 之和等于目标数 9。因此 `index1 = 1`, `index2 = 2`。  
返回 `[1, 2]`。

个人感觉两数之和和盛水的题很像，不过一个是排好顺序的一个没有，没排好顺序的用贪心算法（不一定会更好但至少不会更坏），用 `res` 记录最大结果不断更新就好了


### 盛最多水的容器


双指针，不断向内收敛，主要就是想清数学性正确，收敛的时候一步一步来，不要考虑太多，类似于贪心算法


## 11. 盛最多水的容器

已解答 

中等

 相关标签

 相关企业

 提示

Aa

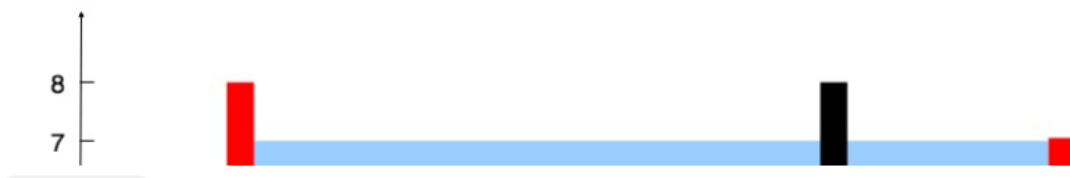
给定一个长度为  $n$  的整数数组 `height`。有  $n$  条垂线，第  $i$  条线的两个端点是  $(i, 0)$  和  $(i, \text{height}[i])$ 。

找出其中的两条线，使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

**说明：**你不能倾斜容器。

**示例 1：**



**三数之和**

给你一个整数数组 `nums`，判断是否存在三元组  $[\text{nums}[i], \text{nums}[j], \text{nums}[k]]$  满足  $i \neq j$ 、 $i \neq k$  且  $j \neq k$ ，同时还满足  $\text{nums}[i] + \text{nums}[j] + \text{nums}[k] == 0$ 。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]]`

先排序；三数之和，固定一个，剩下2个就用双指针法，同两数之和。

while里套while出问题了，所以把while套在一个可能情况比较少的if分支里了

记住 `sort(nums.begin(), nums.end());`，这是vector的函数

## 总结

犹豫不决先排序，步步逼近双指针

## 栈和队列

---

### 栈的应用：

编译器在 词法分析的过程中处理括号、花括号等这个符号的逻辑，也是使用了栈这种数据结构。

linux系统中，cd这个进入目录的命令我们应该再熟悉不过了。

```
cd a/b/c/../../
```

这个命令最后进入a目录，系统是如何知道进入了a目录呢，这就是栈的应用（其实可以出一道相应的面试题了）

### 用栈实现队列

设计一个输入栈和输出栈，只用一个栈实现队列是不可能的，因为他是单边的，所以必须要有两个栈

### 用队列实现栈

原本想着仿造上面想着用一个输入队列，一个输出队列，就可以模拟栈的功能，但是并不可以

队列是先进先出的规则，把一个队列中的数据导入另一个队列中，数据的顺序并没有变，并没有变成先进后出的顺序。

所以用栈实现队列，和用队列实现栈的思路还是不一样的

用两个队列模拟时：一个用来备份数据，操作完再还原回去

**队列模拟栈，其实一个队列就够了**，因为能知道队列的size，只要把队列pop的元素放到队列最后就行，记录好操作次数就行

"向零截断"是指在除法运算中，将除法结果舍入到最接近的整数，而不是四舍五入。具体来说，向零截断是指将小数部分丢弃，只保留整数部分

### 后缀的计算

用栈

还有后缀变中缀，中缀变后缀等题，可以看之前ppt

## 滑动窗口的最大值

给定一个数组 `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。

进阶：

你能在线性时间复杂度内解决此题吗？

示例：

输入： `nums = [1,3,-1,-3,5,3,6,7]`， 和 `k = 3`

输出： `[3,3,5,5,6,7]`

解释：

使用了单调栈

使用了deque双向队列。用法有：

```
deque<int>q
q.front();
q.back();
q.pop_front();    q.pop_back();
q.push_front();   q.push_back();
```

使用deque来实现单调栈最为合适，在文章[栈与队列：来看看栈和队列不为人知的一面 \(opens new window\)](#)中，我们就提到了常用的queue在没有指定容器的情况下，deque就是默认底层容器。

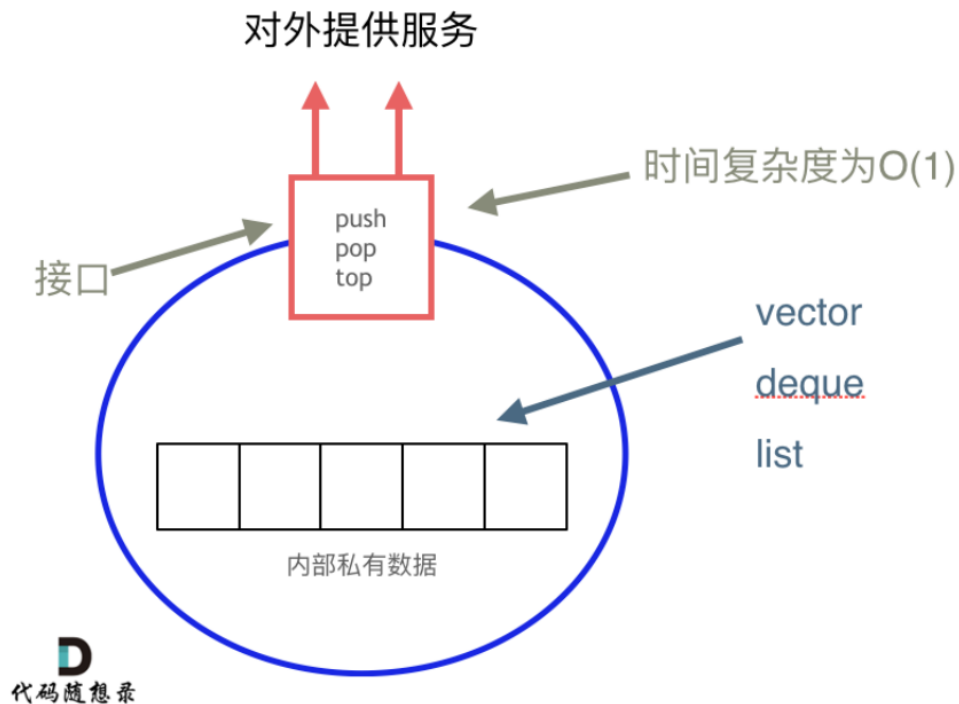
接下来介绍的栈和队列也是SGI STL里面的数据结构，知道了使用版本，才知道对应的底层实现

栈提供push 和 pop 等等接口，所有元素必须符合先进后出规则，所以栈

栈是以底层容器完成其所有的工作，对外提供统一的接口，底层容器是可插拔的（也就是说我们可以控制使用哪种容器来实现栈的功能）。

所以STL中栈往往不被归类为容器，而被归类为container adapter（容器适配器）。

从下图中可以看出，栈的内部结构，栈的底层实现可以是vector，deque，list 都是可以的，主要就是数组和链表的底层实现。



我们常用的SGI STL，如果没有指定底层实现的话，默认是以deque为栈和队列的底层结构

deque是一个双向队列，只要封住一段，只开通另一端就可以实现栈的逻辑了

```
stack<int, vector<int> > third; // 使用vector为底层容器的栈
```

队列中先进先出的数据结构，同样不允许有遍历行为，不提供迭代器

## 第三周

### 二叉树

满二叉树，完全二叉树，二叉搜索树，平衡二叉搜索树（AVL）—左右两个子树的高度差绝对值不超过1（空树也是平衡搜索二叉树）

- 深度优先遍历
  - 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）

看如下中间节点的顺序，就可以发现，中间节点的顺序就是所谓的遍历方式

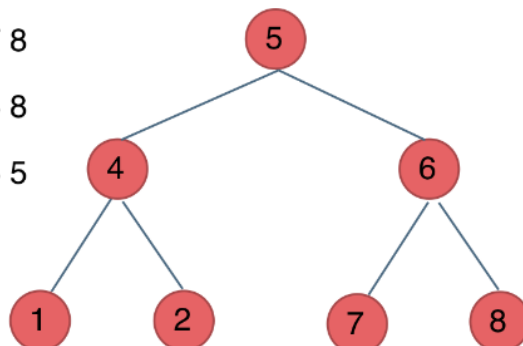
- 前序遍历：中左右
- 中序遍历：左中右
- 后序遍历：左右中

大家可以对着如下图，看看自己理解的前后中序有没有问题。

前序遍历（中左右）：5 4 1 2 6 7 8

中序遍历（左中右）：1 4 2 5 7 6 8

后序遍历（左右中）：1 2 4 7 8 6 5



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

对于链表存储的二叉树节点的定义方式：

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

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。

处理的时候一定记得树为空的情况，不然指针为空都不知道去哪里找错！！

## 遍历

- 递归型

三要素考虑：函数的参数和返回类型、函数终止条件，函数的单个逻辑

- 递归型

- 前序遍历

用栈。先将根节点放入栈中，然后将右孩子加入栈，再加入左孩子。

为什么要先加入右孩子，再加入左孩子呢？因为这样出栈的时候才是中左右的顺序。

注意下面这个地方，空节点不入栈

```
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）
```

- 中序

```

class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        TreeNode* cur = root;
        while (cur != NULL || !st.empty()) {
            if (cur != NULL) { // 指针来访问节点，访问到底底层
                st.push(cur); // 将访问的节点放进栈
                cur = cur->left; // 左
            } else {
                cur = st.top(); // 从栈里弹出的数据，就是要处理的数据（放进
                result数组里的数据）
                st.pop();
                result.push_back(cur->val); // 中
                cur = cur->right; // 右
            }
        }
        return result;
    }
};

```

#### ◦ 后序

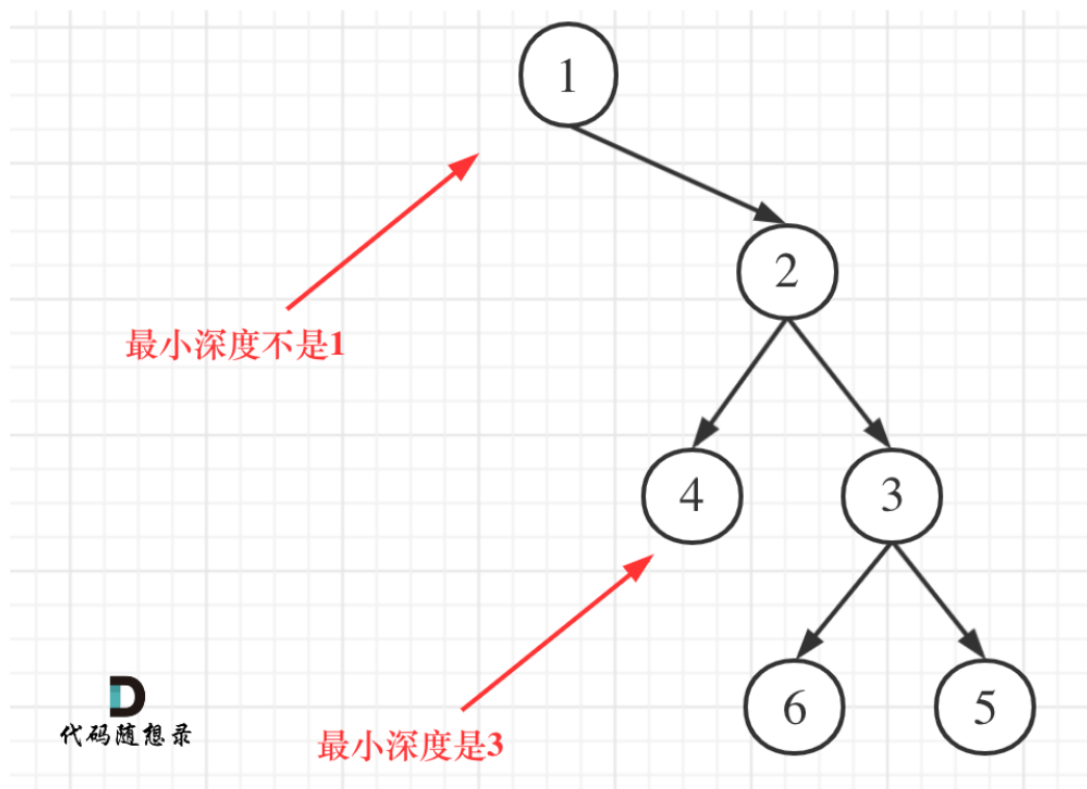
后序遍历为左右中，他的反向就是中右左，那么就是前序遍历的中左右调一下放在栈里的顺序即可

### 二叉树的统一迭代法

待看

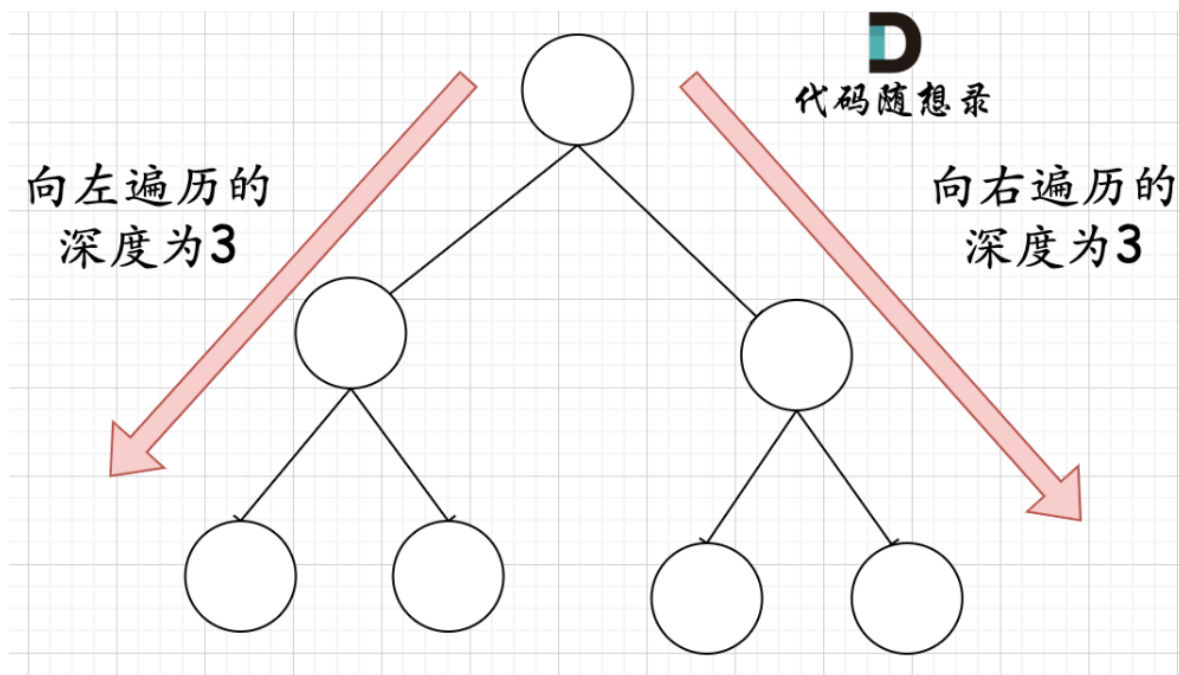
根节点的高度就是二叉树的最大深度

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





在完全二叉树中，如果递归向左遍历的深度等于递归向右遍历的深度，那说明就是满二叉树。



对一个数进行左移操作是指将这个数的二进制表示向左移动指定的位数

假设我们有一个数 5，其二进制表示为 101。左移一位会在原数的右侧添加一个零，相当于乘以 2。。所以 5 左移一位后变为 10，二进制表示为 1010。

- 1 左移 1 位:  $1 \ll 1 = 2$ ，相当于  $1 * 2^1 = 2$
- 1 左移 2 位:  $1 \ll 2 = 4$ ，相当于  $1 * 2^2 = 4$
- 2 左移 1 位:  $2 \ll 1 = 4$ ，相当于  $2 * 2^1 = 4$
- 3 左移 1 位:  $3 \ll 1 = 6$ ，相当于  $3 * 2^1 = 6$

### 平衡二叉树

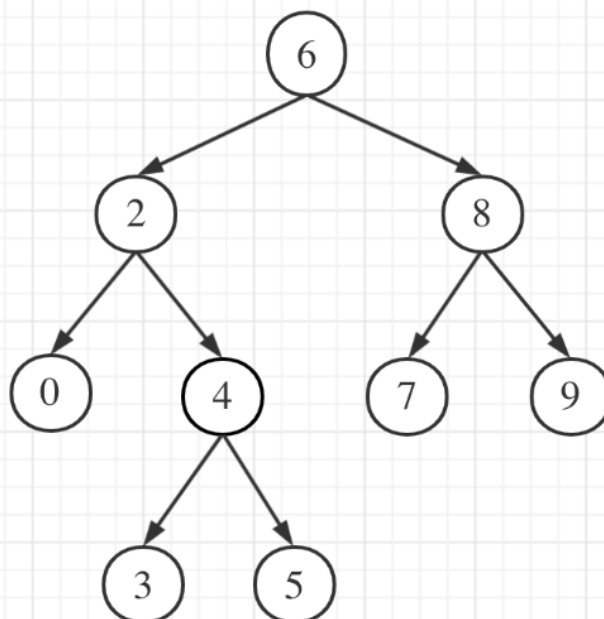
- 二叉树节点的深度：指从根节点到该节点的最长简单路径边的条数。
- 二叉树节点的高度：指从该节点到叶子节点的最长简单路径边的条数。

节点的高度：4，深度：1

节点的高度：3，深度：2

节点的高度：2，深度：3

节点的高度：1，深度：4



## 第四周

### 回溯算法

回溯是递归的副产品，只要有递归就会有回溯。

所以下讲解中，回溯函数也就是递归函数，指的都是一个函数。

组合是不强调元素顺序的，排列是强调元素顺序。

例如：{1, 2} 和 {2, 1} 在组合上，就是一个集合，因为不强调顺序，而要是排列的话，{1, 2} 和 {2, 1} 就是两个集合了。

```
void backtracking(参数) {  
    if (终止条件) {  
        存放结果;  
        return;  
    }  
  
    for (选择: 本层集合中元素 (树中节点孩子的数量就是集合的大小)) {  
        处理节点;  
        backtracking(路径, 选择列表); // 递归  
        回溯, 撤销处理结果  
    }  
}
```

#### 组合问题中的参考代码

## 第77题. 组合

[力扣题目链接](#)

给定两个整数  $n$  和  $k$ ，返回  $1 \dots n$  中所有可能的  $k$  个数的组合。

示例: 输入:  $n = 4, k = 2$  输出:  $[[2,4], [3,4], [2,3], [1,2], [1,3], [1,4], ]$

```
class Solution {  
private:  
    vector<vector<int>> result; // 存放符合条件结果的集合  
    vector<int> path; // 用来存放符合条件结果  
    void backtracking(int n, int k, int startIndex) {  
        if (path.size() == k) {  
            result.push_back(path);  
            return;  
        }  
        for (int i = startIndex; i <= n; i++) {  
            path.push_back(i); // 处理节点  
            backtracking(n, k, i + 1); // 递归  
            path.pop_back(); // 回溯, 撤销处理的节点  
        }  
    }  
};
```

```

    }
}
public:
    vector<vector<int>> combine(int n, int k) {
        result.clear(); // 可以不写
        path.clear();   // 可以不写
        backtracking(n, k, 1);
        return result;
    }
};

```

## 电话号码的字母组合

注意：输入1 \* #按键等等异常情况

代码中最好考虑这些异常情况，但题目的测试数据中应该没有异常情况的数据，所以我就没有加了。

**但是要知道会有这些异常，如果是现场面试中，一定要考虑到！**

```

string path;
path.erase(path.size()-1);
//string可以用加法，但是不可以用减法！

```

## 去重问题

**所谓去重，其实就是使用过的元素不能重复选取。**这么一说好像很简单！

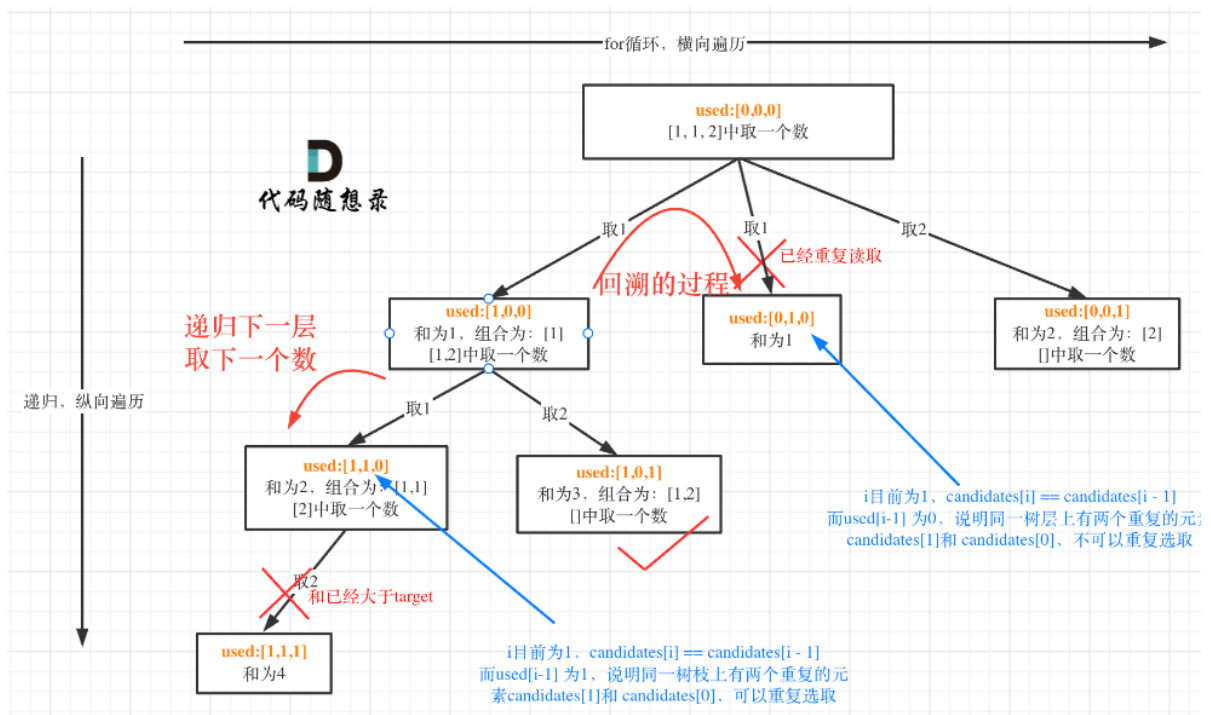
都知道组合问题可以抽象为树形结构，那么“使用过”在这个树形结构上是有两个维度的，**一个维度是同一树枝上使用过，一个维度是同一树层上使用过。**没有理解这两个层面上的“使用过”是造成大家没有彻底理解去重的根本原因。

**如果 `candidates[i] == candidates[i - 1]` 并且 `used[i - 1] == false`，就说明：前一个树枝，使用了 `candidates[i - 1]`，也就是说同一树层使用过 `candidates[i - 1]`。**

此时for循环里就应该做continue的操作。

- `used[i - 1] == true`，说明同一树枝 `candidates[i - 1]` 使用过
- `used[i - 1] == false`，说明同一树层 `candidates[i - 1]` 使用过

可能有的录友想，为什么 **`used[i - 1] == false` 就是同一树层呢**，因为同一树层，`used[i - 1] == false` 才能表示，当前取的 `candidates[i]` 是从 `candidates[i - 1]` 回溯而来的。而 `used[i - 1] == true`，说明是进入下一层递归，去下一个数，所以是树枝上，如下图



**记住使用该方法在树层上去重时必须先排序!!!!**

学到的一些string的新用法

```
string temp=s.substr(i,i-start+1);
//其中i是子串开始的位置, 第二个参数是子串的长度
```

```
string s;
s.erase(s.size()-1);
//删掉s的最后一个元素, 不可以用减法
```

如果把 子集问题、组合问题、分割问题都抽象为一棵树的话, 那么组合问题和分割问题都是收集树的叶子节点, 而子集问题是找树的所有节点!

遍历树的时候, 把所有节点都记录下来, 就是要求的子集集合。

```
unordered_set<int> uset;//去重版
if (uset.find(nums[i]) != uset.end()) {
    continue;
}
//表示没有找到, 没找到的时候就会返回uset.end()
```