【第二十课】专项面试题解析

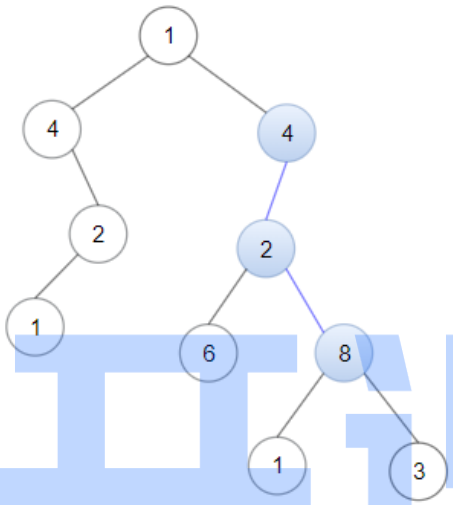
1367. 二叉树中的列表

给你一棵以 `root` 为根的二叉树和一个 `head` 为第一个节点的链表。

如果在二叉树中，存在一条一直向下的路径，且每个点的数值恰好一一对应以 `head` 为首的链表中每个节点的值，那么请你返回 `True` ，否则返回 `False` 。

一直向下的路径的意思是：从树中某个节点开始，一直连续向下的路径。

示例 1：




```
输入：head = [4,2,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]
输出：true
解释：树中蓝色的节点构成了与链表对应的子路径。
```

```
/**
 * 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) {}
 * };
 */
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool judge(TreeNode *root, ListNode *head) {
        if (head == nullptr) return true;
        if (root == nullptr) return false;
        if (root->val != head->val) return false;
        return judge(root->left, head->next) || judge(root->right, head->next);
```

```
    }
    bool isSubPath(ListNode* head, TreeNode* root) {
        if (head == nullptr) return true;
        if (root == nullptr) return false;
        if (root->val == head->val && judge(root, head)) return true;
        return isSubPath(head, root->left) || isSubPath(head, root->right);
    }
};
```

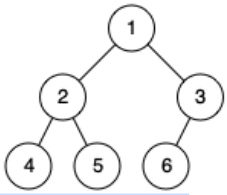
958. 二叉树的完全性检验

给定一个二叉树，确定它是否是一个*完全二叉树*。

百度百科中对完全二叉树的定义如下：

若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。（注：第 h 层可能包含 1~ 2h 个节点。）

示例 1：




```
输入：[1,2,3,4,5,6]
输出：true
解释：最后一层前的每一层都是满的（即，结点值为 {1} 和 {2,3} 的两层），且最后一层中的所有结点（{4,5,6}）都尽可能地向左。
```

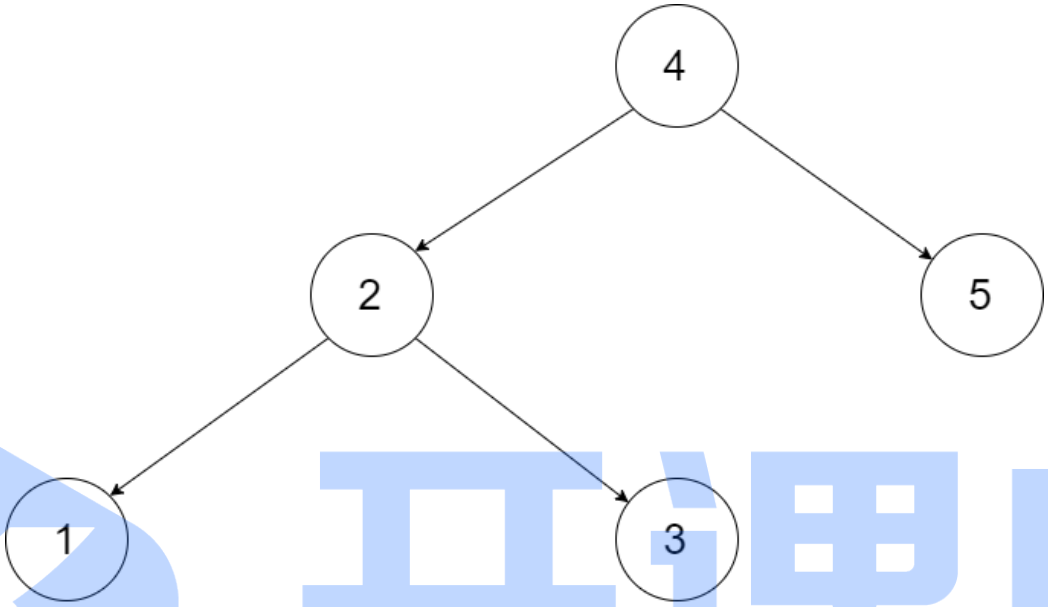
```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int nodeCount(TreeNode *root) {
        if (root == nullptr) return 0;
        return nodeCount(root->left) + nodeCount(root->right) + 1;
    }
    bool judge(TreeNode *root, int n, int m) {
        if (root == nullptr) return n == 0;
        if (n == 0) return false;
        if (n == 1) return root->left == nullptr && root->right == nullptr;
        int k = max(0, 2 * m - 1);
        int l = min(m, n - k), r = n - k - l;
        //cout << n << " : " << k << ", " << l << ", " << r << endl;
        return judge(root->left, (k - 1) / 2 + l, m / 2) && judge(root->right, (k - 1) / 2 + r, m / 2);
    }
    bool isCompleteTree(TreeNode* root) {
        if (root == nullptr) return true;
        int n = nodeCount(root), m = 1, cnt = 1;
        while (cnt + 2 * m <= n) {
            m *= 2;
            cnt += m;
        }
        return judge(root, n, m);
```

```
    }
};
```

剑指 Offer 36. 二叉搜索树与双向链表

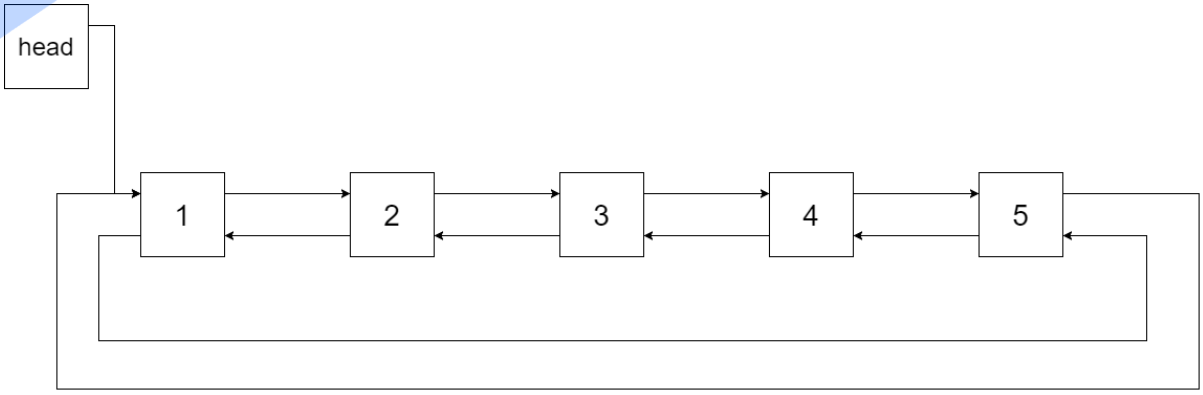
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

为了让您更好地理解问题，以下面的二叉搜索树为例：



我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。



特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
```

```
    Node() {}

    Node(int _val) {
        val = _val;
        left = NULL;
        right = NULL;
    }

    Node(int _val, Node* _left, Node* _right) {
        val = _val;
        left = _left;
        right = _right;
    }
};
*/
class Solution {
public:
    Node *head, *pre;
    void in_order(Node *root) {
        if (root == nullptr) return ;
        in_order(root->left);
        // do something
        if (pre == nullptr) {
            head = root;
        } else {
            pre->right = root;
        }
        root->left = pre;
        pre = root;
        in_order(root->right);
        return ;
    }
    Node* treeToDoublyList(Node* root) {
        if (root == nullptr) return nullptr;
        head = pre = nullptr;
        in_order(root);
        head->left = pre;
        pre->right = head;
        return head;
    }
};
```

464. 我能赢吗

在 "100 game" 这个游戏中，两名玩家轮流选择从 1 到 10 的任意整数，累计整数和，先使得累计整数和达到或超过 100 的玩家，即为胜者。

如果我们将游戏规则改为 “玩家不能重复使用整数” 呢？

例如，两个玩家可以轮流从公共整数池中抽取从 1 到 15 的整数（不放回），直到累计整数和 >= 100。

给定一个整数 `maxChoosableInteger` （整数池中可选择的最大数）和另一个整数 `desiredTotal`（累计和），判断先出手的玩家是否能稳赢（假设两位玩家游戏时都表现最佳）？

你可以假设 `maxChoosableInteger` 不会大于 20， `desiredTotal` 不会大于 300。

示例：

```
输入：
maxChoosableInteger = 10
desiredTotal = 11

输出：
false

解释：
无论第一个玩家选择哪个整数，他都会失败。
第一个玩家可以选择从 1 到 10 的整数。
如果第一个玩家选择 1，那么第二个玩家只能选择从 2 到 10 的整数。
第二个玩家可以通过选择整数 10（那么累积和为 11 >= desiredTotal），从而取得胜利.
同样地，第一个玩家选择任意其他整数，第二个玩家都会赢。
```

```
class Solution {
public:
    unordered_map<int,bool> h;
    bool dfs(int mask,int n,int total ){
        if(h.find(mask) != h.end()) return h[mask];
        for(int i = 1; i <= n; i++){
            if(mask & (1 << i)) continue;
            if(i >= total || !dfs(mask | (1 << i), n, total - i)){
                return h[mask] = true;
            }
        }
        return h[mask] = false;
    }
    bool canIWin(int maxChoosableInteger,int desiredTotal){
        int n = maxChoosableInteger,mask = 0;
        if((n + 1) * n / 2 < desiredTotal)return false;
        h.clear();
        return dfs(mask,maxChoosableInteger,desiredTotal);
    }
};
```

172. 阶乘后的零

给定一个整数 *n*，返回 *n*! 结果尾数中零的数量。

示例：

```
输入: 3
输出: 0
解释: 3! = 6, 尾数中没有零。
```

```
class Solution {
public:
    int trailingZeroes(int n) {
        int m = 5, cnt = 0;
        while (n / m) {
            cnt += n / m;
            m *= 5;
        }
        return cnt;
    }
};
```

384. 打乱数组

给你一个整数数组 nums ，设计算法来打乱一个没有重复元素的数组。

实现 `Solution` class:

- `Solution(int[] nums)` 使用整数数组 `nums` 初始化对象
- `int[] reset()` 重设数组到它的初始状态并返回
- `int[] shuffle()` 返回数组随机打乱后的结果

示例：

```
输入
["Solution", "shuffle", "reset", "shuffle"]
[[[1, 2, 3]], [], [], []]
输出
[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]

解释
```

```
Solution solution = new Solution([1, 2, 3]);
solution.shuffle();    // 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3]的排列返回的概率应该相同。例如，返回 [3, 1, 2]
solution.reset();      // 重设数组到它的初始状态 [1, 2, 3] 。返回 [1, 2, 3]
solution.shuffle();    // 随机返回数组 [1, 2, 3] 打乱后的结果。例如，返回 [1, 3, 2]
```

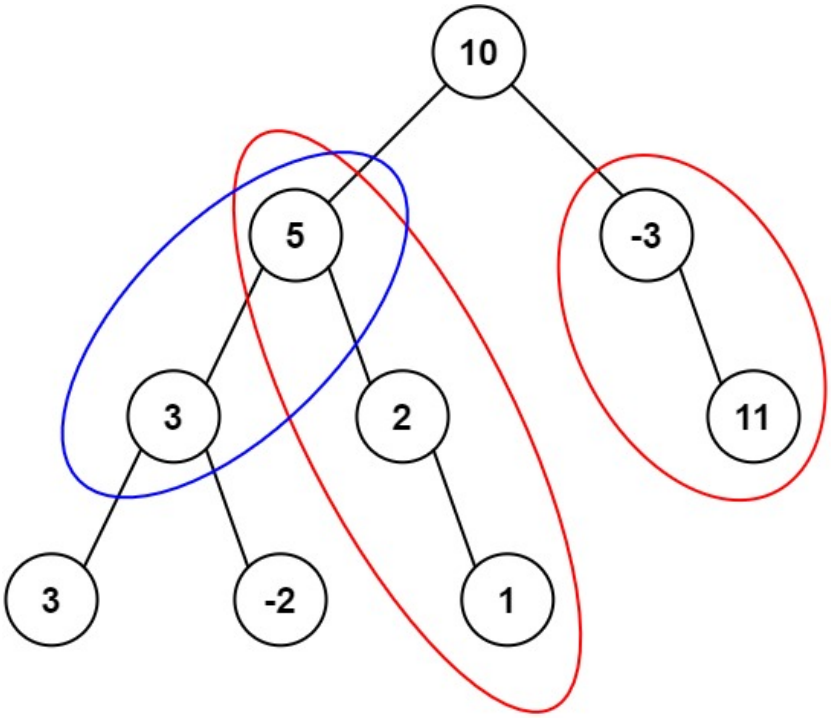
```
class Solution {
public:
    vector<int> nums;
    Solution(vector<int>& nums) : nums(nums) {
        srand(time(0));
    }

    /** Resets the array to its original configuration and return it. */
    vector<int> reset() {
        return nums;
    }

    /** Returns a random shuffling of the array. */
    vector<int> shuffle() {
        vector<int> ret(nums);
        for (int i = 0; i < ret.size(); i++) {
            swap(ret[i], ret[rand() % ret.size()]);
        }
        return ret;
    }
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution* obj = new Solution(nums);
 * vector<int> param_1 = obj->reset();
 * vector<int> param_2 = obj->shuffle();
 */
```

437. 路径总和 III

给定一个二叉树的根节点 `root` ，和一个整数 `targetSum` ，求该二叉树里节点值之和等于 `targetSum` 的 **路径** 的数目。

路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

示例 1：




```
输入：root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
输出：3
解释：和等于 8 的路径有 3 条，如图所示。
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    unordered_map<int, int> h;
    int count(TreeNode *root, int sum, int targetSum) {
        if (root == nullptr) return 0;
        sum += root->val;
        int ans = h[sum - targetSum];
        h[sum] += 1;
        ans += count(root->left,  sum, targetSum);
        ans += count(root->right, sum ,targetSum);
        h[sum] -= 1;
        return ans;
    }
    int pathSum(TreeNode* root, int targetSum) {
        h.clear();
        h[0] = 1;
        return count(root, 0, targetSum);
    }
};
```

395. 至少有 K 个重复字符的最长子串

给你一个字符串 `s` 和一个整数 `k` ，请你找出 `s` 中的最长子串， 要求该子串中的每一字符出现次数都不少于 `k` 。返回这一子串的长度。

示例 1：

```
输入：s = "aaabb", k = 3
输出：3
解释：最长子串为 "aaa" ，其中 'a' 重复了 3 次。
```

```
class Solution {
public:
    int longestSubstring(string s, int k) {
        unordered_map<char, int> cnt;
        vector<int> splits;
        for (auto x : s) cnt[x] += 1;
        for (int i = 0; s[i]; ++i) {
            if (cnt[s[i]] < k) splits.push_back(i);
        }
        splits.push_back(s.size());
        if (splits.size() == 1) return s.size();
        int pre = 0, ans = 0;
        for (auto p : splits) {
            int len = p - pre;
            if (len >= k) {
                ans = max(ans, longestSubstring(s.substr(pre, len), k));
            }
            pre = p + 1;
        }
        return ans;
    }
};
```

190. 颠倒二进制位

颠倒给定的 32 位无符号整数的二进制位。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 **示例 2** 中，输入表示有符号整数 `3`，输出表示有符号整数 `1073741825`。

进阶:如果多次调用这个函数，你将如何优化你的算法？

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t ret = 0;
        for (uint32_t i = 0, j = 1, k = (1 << 31); i < 32; i++, j <<= 1, k >>= 1) {
            if (n & j) ret |= k;
        }
        return ret;
    }
};
```

8. 字符串转换整数 (atoi)

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数（类似 C/C++ 中的 `atoi` 函数）。

函数 `myAtoi(string s)` 的算法如下：

- 读入字符串并丢弃无用的前导空格
- 检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。 确定最终结果是负数还是正数。 如果两者都不存在，则假定结果为正。
- 读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
- 将前面步骤读入的这些数字转换为整数（即，"123" -> 123， "0032" -> 32）。如果没有读入数字，则整数为 `0` 。必要时更改符号（从步骤 2 开始）。
- 如果整数数超过 32 位有符号整数范围 `[−2 31 , 2 31 − 1]` ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 `−2 31` 的整数应该被固定为 `−2 31` ，大于 `2 31 − 1` 的整数应该被固定为 `2 31 − 1` 。
- 返回整数作为最终结果。

注意：

- 本题中的空白字符只包括空格字符 `' '` 。
- 除前导空格或数字后的其余字符串外，**请勿忽略** 任何其他字符。

示例 1：

```
输入：s = "42"
输出：42
解释：加粗的字符串为已经读入的字符，插入符号是当前读取的字符。
第 1 步："42"（当前没有读入字符，因为没有前导空格）
         ^
第 2 步："42"（当前没有读入字符，因为这里不存在 '-' 或者 '+'）
          ^
第 3 步："42"（读入 "42"）
            ^
解析得到整数 42 。
由于 "42" 在范围 [-231, 231 - 1] 内，最终结果为 42 。
```

```
class Solution {
public:
    int myAtoi(string s) {
        int flag = 1, max_pre = INT_MAX / 10, d = INT_MAX % 10, ind = 0, num = 0;

        while (s[ind] == ' ') ++ind;

        if (s[ind] == '-') flag = -1, ind += 1;
        else if (s[ind] == '+') ind += 1;

        for (; s[ind]; ++ind) {
            if (s[ind] < '0' || s[ind] > '9') break;
            if (num > max_pre || (num == max_pre && (s[ind] - '0') > d)) {
                if (flag > 0) return INT_MAX;
                return INT_MIN;
            }
            num = num * 10 + (s[ind] - '0');
        }
        return num * flag;
    }
};
```

380. O(1) 时间插入、删除和获取随机元素

设计一个支持在*平均* 时间复杂度 **O(1)** 下，执行以下操作的数据结构。

1. `insert(val)`：当元素 val 不存在时，向集合中插入该项。
2. `remove(val)`：元素 val 存在时，从集合中移除该项。
3. `getRandom`：随机返回现有集合中的一项。每个元素应该有**相同的概率**被返回。

示例：

```
// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();

// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);

// 返回 false ，表示集合中不存在 2 。
randomSet.remove(2);

// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);

// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();

// 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomSet.remove(1);

// 2 已在集合中，所以返回 false 。
randomSet.insert(2);

// 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
randomSet.getRandom();
```

```
class RandomizedSet {
public:
    /** Initialize your data structure here. */
    unordered_map<int, int> h;
    vector<int> arr;
    RandomizedSet() {
        srand(time(0));
    }

    /** Inserts a value to the set. Returns true if the set did not already contain the specified element. */
    bool insert(int val) {
        if (h.find(val) != h.end()) return false;
        h[val] = arr.size();
        arr.push_back(val);
        return true;
    }

    void swap_item(int i, int j) {
        swap(arr[i], arr[j]);
        h[arr[i]] = i;
        h[arr[j]] = j;
        return ;
    }

    /** Removes a value from the set. Returns true if the set contained the specified element. */
    bool remove(int val) {
        if (h.find(val) == h.end()) return false;
        int n = h[val], m = arr.size() - 1;
        swap_item(n, m);
        h.erase(h.find(val));
        arr.pop_back();
        return true;
    }

    /** Get a random element from the set. */
    int getRandom() {
        return arr[rand() % arr.size()];
    }
};

/**
 * Your RandomizedSet object will be instantiated and called as such:
 * RandomizedSet* obj = new RandomizedSet();
 * bool param_1 = obj->insert(val);
 * bool param_2 = obj->remove(val);
```

```
 * int param_3 = obj->getRandom();
 */
```

402. 移掉 K 位数字

给你一个以字符串表示的非负整数 `num` 和一个整数 `k` ，移除这个数中的 `k` 位数字，使得剩下的数字最小。请你以字符串形式返回这个最小的数字。

示例 1 ：

```
输入：num = "1432219", k = 3
输出："1219"
解释：移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219 。
```

```
class Solution {
public:
    string removeKdigits(string num, int k) {
        if (k >= num.size()) return "0";
        string ret;
        for (auto x : num) {
            while (k && ret.size() && ret.back() > x) ret.pop_back(), k -= 1;
            ret.push_back(x);
        }
        if (k != 0) ret = ret.substr(0, ret.size() - k);
        int ind = 0;
        while (ret[ind] == '0') ++ind;
        ret = ret.substr(ind, ret.size());
        if (ret == "") ret = "0";
        return ret;
    }
};
```

1081. 不同字符的最小子序列

返回 `s` 字典序最小的子序列，该子序列包含 `s` 的所有不同字符，且只包含一次。

注意：该题与 316 https://leetcode.com/problems/remove-duplicate-letters/ 相同

示例 1：

```
输入：s = "bcabc"输出："abc"
```

```
class Solution {
public:
    string smallestSubsequence(string s) {
        string ret;
        unordered_map<char, int> cnt;
        for (auto x : s) cnt[x] += 1;
        unordered_set<char> h;
        for (auto x : s) {
            if (h.find(x) == h.end()) {
                while (ret.size() && cnt[ret.back()] && ret.back() > x) {
                    h.erase(h.find(ret.back()));
                    ret.pop_back();
                }
                h.insert(x);
                ret.push_back(x);
            }
            cnt[x] -= 1;
        }
        return ret;
    }
};
```

1499. 满足不等式的最大值

给你一个数组 `points` 和一个整数 `k` 。数组中每个元素都表示二维平面上的点的坐标，并按照横坐标 x 的值从小到大排序。也就是说 `points[i] = [x i , y i ]` ，并且在 `1 <= i < j <= points.length` 的前提下， `x i < x j` 总成立。

请你找出 `y i + y j + |x i - x j |` 的 **最大值**，其中 `|x i - x j | <= k` 且 `1 <= i < j <= points.length`。

题目测试数据保证至少存在一对能够满足 `|x i - x j | <= k` 的点。

```
class Solution {
public:
    int findMaxValueOfEquation(vector<vector<int>>& points, int k) {
        deque<int> q;
        q.push_back(0);
        int ans = INT_MIN;
        for (int i = 1; i < points.size(); i++) {
            while (q.size() && points[i][0] - points[q.front()][0] > k) q.pop_front();
            if (q.size()) {
                ans = max(ans, points[i][0] - points[q.front()][0] + points[i][1] + points[q.front()][1]);
            }
            while (q.size() && points[i][1] - points[i][0] > points[q.back()][1] - points[q.back()][0]) q.pop_back();
            q.push_back(i);
        }
        return ans;
    }
};
```

316. 去除重复字母

给你一个字符串 `s` ，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证 **返回结果的字典序最小**（要求不能打乱其他字符的相对位置）。

注意：该题与 1081 https://leetcode-cn.com/problems/smallest-subsequence-of-distinct-characters 相同

示例 1：

```
输入：s = "bcabc"输出："abc"
```

```
class Solution {
public:
    string removeDuplicateLetters(string s) {
        string ret;
        unordered_map<char, int> cnt;
        for (auto x : s) cnt[x] += 1;
        unordered_set<char> h;
        for (auto x : s) {
            if (h.find(x) == h.end()) {
                while (ret.size() && cnt[ret.back()] && ret.back() > x) {
                    h.erase(h.find(ret.back()));
                    ret.pop_back();
                }
                h.insert(x);
                ret.push_back(x);
            }
            cnt[x] -= 1;
        }
        return ret;
    }
};
```