作业 HW05 实验报告

姓名：闫浩扬 学号：2253156 日期：2023 年 12 月 24 日

1. 涉及数据结构和相关背景

本次作业涉及查找的相关知识。需要掌握各种查找数据结构与算法的实现，同时掌握各种算法的特点，效率和适用情况。

1.1 基本术语

查找表：由同一类型的数据元素(或记录)构成的集合。

查找表分为动态查找表和静态查找表

静态查找表：只作查找操作的查找表。主要操作：查询某个“特定的”数据元素是否在查找表中。检索某个“特定的”数据元素和各种属性。

动态查找表： 在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已经存在的某个数据元素。主要操作：查找时插入不存在的数据元素。查找时删除已存在的数据元素。

主关键字：在每个对象中有若干属性，其中有一个属性，其值可唯一地标识这个对象。

次关键字：可以标识若干个数据元素。

平均查找长度：为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值，称为查找算法。在查找成功时的平均查找长度(Average Search Length, ASL)。

$$ASL=\sum_{i=1}^{n} p_i c_i$$

1.2 存储结构及算法

1.2.1 线性结构的查找

(一) 顺序查找

查找过程：从表的一端开始，依次将记录的关键字和给定值进行比较，若某个记录的关键字和给定值相等，则查找成功；反之，若扫描整个表后，仍未找到关键字和给定值相等的记录，则查找失败。

应用范围：顺序查找法适用于顺序表或链表表示的静态查找表，同时表内元素之间无序。

算法思想：把待查关键字 key 存入表头（哨兵），从后向前逐个比较，可免去查找过程中每一步都要检测是否查找完毕，加快速度。

性能分析：空间复杂度：一个辅助空间

时间复杂度：

① 查找成功时的平均查找长度（设表中各记录查找概率相等）

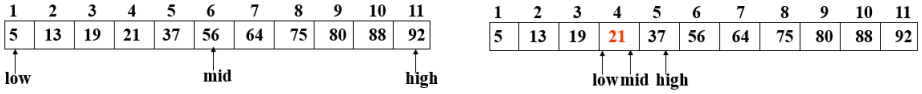
$$\boldsymbol{ASLs(n)=(1+2+\ldots+n)/n=(n+1)/2;}$$

②查找不成功时的平均查找长度 ASLf =n+1。

(二) 折半查找

折半查找也叫二分查找，采用了分治的思想，是一种效率较高的查找方法。

查找过程：不断与表中部元素的关键字进行比较，缩小范围直到找到为止。



应用范围：折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列（递增或递减）；不宜用于链式结构。

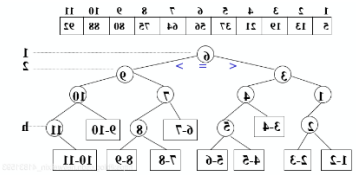
算法思想：设表长为 n，low、high 和 mid 分别指向待查元素所在区间的上界、下界和中点,k 为给定值。 初始时，令$low=1,high=n,mid=(low+high)/2$让 k 与 mid 比较：

①若 k==R[mid].key，查找成功；

②若 k<R[mid].key，则 high=mid-1；

③若 k>R[mid].key，则 low=mid+1。重复上述操作，直至 low>high 时，查找失败。

性能分析：折半查找使用判定树进行分析。该判定树是一颗二叉排序树，中序有序。



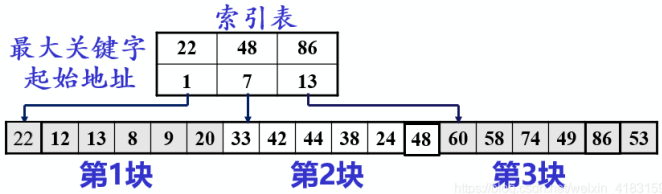
查找成功时，$ASL=(1*1+2*2+4*3+4*4)/11.$

查找成功时比较次数：为该结点在判定树上的层次数，不超过树的深度$d = log\,n+1$, 查找不成功的过程就是走了一条从根结点到外部结点的路径 d 或 d-1。

每次将待查记录所在区间缩小一半，比顺序查找效率高, 时间复杂度 O(log2n)

(三) 分块查找

分块有序，即分成若干子表，要求每个子表中的数值都比后一块中数值小（但子表内部未必有序）。然后将各子表中的最大关键字构成一个索引表，表中还要包含每个子表的起始地址（即头指针）。



查找过程：对索引表使用折半查找法（索引表是有序表）；确定了待查关键字所在的子表后，在子表内采用顺序查找法（各子表内部是无序表）

应用范围：如果线性表既要快速查找又经常动态变化，则可采用分块查找。

算法思想：可以发现，分块查找实际是顺序查找和二分查找额的简单合成。

性能分析：$ASL=Lb+Lw$（Lb 是对索引表查找的 ASL，Lw 是对块内查找的 ASL）

分块查找插入和删除比较容易无需进行大量移动，但是要增加一个索引表的存储空间并对初始索引表进行排序运算。

1.2.2 树形结构的查找

线性表的查找更适用于静态查找表，若要对动态查找表进行高效率的查找，可以采用几种特殊的二叉树，称作树表。

(一) 二叉排序树 BST

BST 有以下性质：

①若其左子树非空，则左子树上所有结点的值均小于根结点的值；

②若其右子树非空，则右子树上所有结点的值均大于等于根结点的值；

③其左右子树本身又各是一棵二叉排序树

特点：中序遍历二叉排序树得到一个关键字递增的有序序列。

二叉排序树的查找：

算法思想：若二叉排序树为空，则查找不成功，在二叉排序树中插入 该元素；否则

（1）若给定值等于根结点的关键字，则查找成功；

（2）若给定值小于根结点的关键字，则继续在左子树上进行查找；

（3）若给定值大于根结点的关键字，则继续在右子树上进行查找。

平均查找长度和二叉树的形态有关，最好：$logn$（形态匀称，与二分查找的判定树相似）；

最坏:$(n+1)/2$（单支树）.

二叉排序树的插入删除及生成

(二) 平衡二叉树 AVL 树

为尽量提高二叉排序树的查找效率，尽量让二叉树的形状均衡。左、右子树是平衡二叉树；所有结点的左、右子树深度之差的绝对值≤ 1。任一结点的平衡因子只能取：-1、0 或 1；如果树中任意一个结点的平衡因子的绝对值大于 1，则这棵二叉树就失去平衡，不再是 AVL 树；

平衡因子：该结点左子树与右子树的高度差。

算法分析：对于有 n 个结点的 AVL 树，其高度保持在$O(logn)$，ASL 也保持在$O(logn)$.

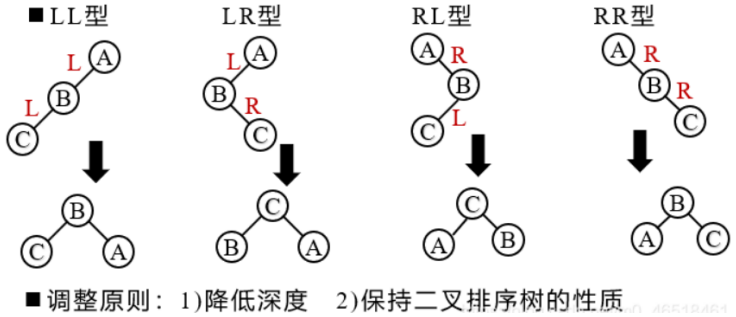
平衡旋转：如果在一棵 AVL 树中插入一个新结点，就有可能造成失衡，此时必须重新调整树的结构，使之恢复平衡。

LL：若在 A 的左子树的左子树上插入结点，使 A 的平衡因子从 1 增加至 2，需要进行一次顺时针旋转(以 B 为旋转轴）

RR：若在 A 的右子树的右子树上插入结点，使 A 的平衡因子从-1 增加至-2，需要进行一次逆时针旋转(以 B 为旋转轴）

LR：若在 A 的左子树的右子树上插入结点，使 A 的平衡因子从 1 增加至 2，需要先进行逆时针旋转，再顺时针旋转(以插入的结点 C 为旋转轴）

RL：若在 A 的右子树的左子树上插入结点，使 A 的平衡因子从-1 增加至-2，需要先进行顺时针旋转，再逆时针旋转(以插入的结点 C 为旋转轴）



键树/B-树/B+树

1.2.3 散列表的查找

哈希方法：选取某个函数，依该函数按关键字计算元素的存储位置，并按此存放；查找时，由同一个函数对给定值 k 计算地址，将 k 与地址单元中元素关键码进行比，确定查找是否成功。

哈希表：使用哈希函数构造的表

基本思想：记录的存储位置与关键字之间存在对应关系，Loc(i)=H(keyi)→哈希函数

优点：查找速度极快，时间复杂度为 O(1),查找效率与元素个数 n 无关

哈希函数的构造：

直接定址法：$Hash(key) = a \cdot key + b$ (a、b为常数)

优点：以关键码 key 的某个线性函数值为哈希地址，不会产生冲突。

缺点：要占用连续地址空间，空间效率低。

数字分析法：选取数码分布较为均匀的若干位作为散列地址，这种方法适合于已知的关键字集合，若更换了关键字，则需要重新构造新的散列函数。

平方取中法：取关键字的平方值的中间几位作为散列地址，具体取多少位要视实际情况而定。这种方法得到的散列地址与关键字的每位都有关系，因此使得散列地址分布比较均匀,适用于关键字的每位取值都不够均匀或均小于散列地址所需的位数。

除留余数法：$Hash(key) = key \bmod p$ (p是一个整数)。该方法的关键是选取有个合适的 p 值，一般情况下，若表长为 m，取 p 为小于等于 m 的最大质数。

冲突处理：

冲突：不同的关键码映射到同一个哈希地址。$key1 \neq key2$，但$H(key1) = H(key2)$

开放定址法：开放定址法的基本思想为有冲突时就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。

其冲突函数为$Hi = (Hash(key) + di) \bmod m$ ($1 \leq i < m$)

线性探测法：di 为增量序列 1,2,3，…，n

二次探测法：di 为增量序列 1^2，-1^2…

伪随机探测法：di 为随机数

链地址法：相同哈希地址的记录链成一单链表，m 个哈希地址就设 m 个单链表，然后用一个数组将 m 个单链表的表头指针存储起来，形成一个动态的结构。

①取数据元素的关键字 key，计算其哈希函数值（地址）。若该地址对应的链表为空，则

将该元素插入此链表；否则执行②解决冲突。
②利用链表的前插法或后插法将该元素插入此链表。
优点①非同义词不会冲突，无“聚集”现象；②链表上结点空间动态申请，更适合于表长不确定的情况。
效率分析： ASL 取决于以下几个因素：①哈希函数；②处理冲突的方法；③哈希表的装填因子。α 越大，表中记录数越多，说明表装得越满，发生冲突的可能性就越大，查找时比较次数就越多。哈希表技术具有很好的平均性能，优于一些传统的技术；链地址法优于开地址法；除留余数法作哈希函数优于其它类型函数。

2. 实验内容

2.1 和有限的最长子序列

2.1.1 问题描述

一个长度为 n 的整数数组 nums 和一个长度为 m 的整数数组 queries，返回一个长度为 m 的数组 answer，其中 answer[i]是 nums 中元素和小于等于 queries[i]的子序列的最大长度。子序列是由一个数组删除某些元素（也可以不删除）但不改变元素顺序得到的一个数组。

2.1.2 基本要求

输入：第一行包括两个整数 n 和 m，分别表示数组 nums 和 queries 的长度
输出：输出一行，包括 m 个整数，为 answer 中元素

2.1.3 数据结构设计

这道题中 queries 中的每个元素代表一个长度，我们需要在 nums 中找到元素和不超过 queries[i]的最大长度。因为题目要选的是子序列，为了使子序列尽量长。就应该优先选择较小的数字。另一方面，对于较大的 query[i]，其对应的子序列一定是在较小的 query 对应的子序列的基础上再加一些 nums 中未使用的较小的元素。所以，大概有以下思路:

① 我们可以把 nums 和 queries 先按照从小到大进行排序，使 nums 和 queries 变成两个单调的序列，然后遍历排序后的 queries，利用单调性在一趟遍历内确定每个 query 对应的子序列之和。
② 前缀和的思想：先计算出排序后数组的前缀和，对于每个 query 再运用二分查找。
③ std::upper_bound,std::lower_bound
④ 由于 queries 的输出有顺序，所以先对 nums 进行排序，这样长度为 n 的子序列最小和一定是前 n 位。随后可以通过其他查找算法挨个对 queries 中的元素进行查找。这里可以使用多种算法，比方二分查找等，这里我使用双指针法进行查找。

双指针法：适用于多种场景，其中之一就是在一个序列中寻找一个连续的空间，在本题中，我们可以使用指针 i 遍历整个序列，另一个指针 j 维护以 i 为右端点的最长的不重复的子序列，优势在于时间复杂度较低，适用于有序数组。

数据结构设计：使用 STL 模版中的 vector 向量容器存放原始 nums，排序后的 nums，queries 和 answer。

queries/nums

```
vector<int> queries(m);
vector<int> nums(n);
```

sortedNums

```
vector<int> sortedNums = nums;   //原始的 nums
sort(sortedNums.begin(), sortedNums.end()); //从小到大排序的 nums
```

result/answer

```
vector<int> answer(queries.size()); //一个等同于 queries 大小的容器
// ...
answer[i] = left; //左指针即代表最长子序列右端点位置
```

2.1.4 功能说明（函数、类）

为使代码整洁易读，将函数封装起来放在 Solution 类中，函数返回类型为 vector<int>

```
/*description: 通过双指针法找出满足条件的子序列长度
```

```
    *param1   nums : 原始数字序列
    *param2   queries : 查询序列
    *return   answer :查询结果
    */
    vector<int> solveQueries(const vector<int>& nums, const vector<int>& queries) {
        vector<int> sortedNums = nums;
        sort(sortedNums.begin(), sortedNums.end());// 排序原始数字序列
        vector<int> answer(queries.size());// 存储结果
        for (int i = 0; i < queries.size(); ++i) {
            int sum = 0;
            int left = 0, right = sortedNums.size() - 1;
            // 在排序后的序列中使用双指针查找
            while (left <= right) {
                // 如果当前元素加上 sum 不超过查询值，将当前元素加到 sum 中，同时左指针右移
                if (sortedNums[left] + sum <= queries[i]) {
                    sum += sortedNums[left];
                    ++left;
                }
                else
                    --right; // 超过查询值，右指针左移
            }
            answer[i] = left; // 存储查询的结果（左指针的位置即代表最长子序列的右端点）
        }
        return answer;
    }
```

设计思想：现将 nums 进行排序，随后使用双指针法处理每个查询，找出满足条件的子序列长度。
时间复杂度：排序使用内置的 sort 函数，时间复杂度为$O(nlogn)$，每个 query 的处理为$O(n)$，其中 n 是 nums 长度。综合时间复杂度为$O(nlogn+n)$，因为排序时间复杂度更高，可以整体近似为$O(nlogn)$
空间复杂度：需要一个和 queries 等长的容器来存储结果，一个和 nums 等长的存储排序后的 nums，所以总的空间复杂度为$O(n)$

2.1.5 调试分析（遇到的问题和解决方法）

1. 思考不全面，急于做题

读完题后发现比较简单，于是想用暴力破解法，嵌套循环解决，但是仔细想，想到算法题肯定会有更好地方法。画了一下过程，发现要找最长的子序列，肯定是在前一个最长子序列的基础上加一个未使用的且较小的元素，所以，发现首先需要将 nums 先排序，但是 queries 不可以，因为不知道 queries 是否有序。排完序后可以用许多算法查找，这里选用了双指针法。

2. 算法的优化

双指针法对于每个 query 都需要从头开始，实际上有一些重复工作。后来发现有前缀和，或许可以将 queries 和 nums 都排序，然后使用前缀和求出每个 queries 对应的序列长度，填充在原始的 queries 数组中，这样可以保证顺序不变。

另外，想起老师提到的 upper_bound 函数， 从数组的 begin 位置到 end-1 位置二分查找第一个大于 num 的数字，找到返回该数字的地址，不存在则返回 end。通过返回的地址减去起始地址 begin,得到找到数字在数组中的下标。发现这种内置函数很便捷。

```
class Solution {
public:
    vector<int> answerQueries(vector<int>& nums, vector<int>& queries) {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        vector<int> prefix_sums(n + 1);
        for (int i = 0; i < n; ++i) {
            prefix_sums[i + 1] = prefix_sums[i] + nums[i];
        }
        for (int& query : queries) {
            query = upper_bound(prefix_sums.begin(), prefix_sums.end(), query)-
prefix_sums.begin() - 1;
        }
        return queries;
```

```
    }
};
```

2.1.6 总结和体会

在这道题中，我学习了许多有趣的知识，比方 upper_bound 函数，双指针法等，我也意识到采用合适的算法可以带来极大的效率提升(相比暴力破解).

2.2 二叉排序树 BST

2.2.1 问题描述

本题实现一个维护整数集合（允许有重复关键字）的 BST，并具有以下功能：1.插入一个整数 2.删除一个整数 3.查询某个整数有多少个 4.查询最小值 5.查询某个数字的前驱（集合中比该数字小的最大值）。

2.2.2 基本要求

第 1 行一个整数 n，表示操作的个数；接下来 n 行，每行一个操作，第一个数字 op 表示操作种类：若 op=1，后面跟着一个整数 x，表示插入数字 x 若 op=2，后面跟着一个整数 x，表示删除数字 x（若存在则删除，否则输出 None，若有多个则只删除一个），若 op=3，后面跟着一个整数 x，输出数字 x 在集合中有多少个（若 x 不在集合中则输出 0）若 op=4，输出集合中的最小值（保证集合非空）若 op=5，后面跟着一个整数 x，输出 x 的前驱（若不存在前驱则输出 None，x 不一定在集合中）

2.2.3 数据结构设计

二叉排序树是一种实现动态查找表额的树形存储结构，说明不仅需要查找目标元素，还需要执行插入和删除元素的操作。它本质是一棵二叉树，特别之处在于：对于树中的每个结点，如果它有左子树，那么左子树上所有结点的值都比该结点小；对于树中的每个结点，如果它有右子树，那么右子树上所有结点的值都比该结点大。

由于设计动态查找表，所以选择链式结构表示更有利于删除等操作。

本题中涉及二叉排序树的几种操作：

op=1，插入

二叉排序树的创建实际也是多个插入操作，所以在开始仅需初始化即可，同时在插入函数中需要加以判断 BST 是否为空。

同时，本题实现一个维护整数集合，允许有重复的关键字，但是二叉排序树中不可以有结点关键字相同，所以意思即为在节点中存一个 count，用于存放频次信息，还需要有一个 val 存放关键字信息。又因为是二叉树，所以还需要有指向左右孩子的指针。

插入的过程同二叉搜索类似，与根节点比较，小往左走，大往右走，直到最终查找失败的位置就是新节点放置的位置，如果有相同节点就给节点增加频次 count，最后返回根节点地址。

op=2，删除

删除操作较为复杂因为删除需要保证整棵树还是一棵二叉排序树，假设被删除的元素是 P，删除的同时需要妥善处理它的左、右子树。根据结点 P 是否有左、右孩子，可以归结为以下 3 种情况：

①P 是叶子结点：可以直接摘除，整棵树还是二叉排序树。

②P 只有一个孩子（左孩子或右孩子）：若 P 是双亲结点（用 F 表示）的左孩子，直接将 P 的孩子结点作为 F 的左孩子；反之若 P 是 F 的右孩子，直接将 P 的孩子结点作为 F 的右孩子，整棵树还是二叉排序树。

③P 有两个孩子：中序遍历整棵二叉排序树，在中序序列里找到 P 的直接前驱结点 S，将 P 结点修改成 S 结点，然后再将之前的 S 结点从树中摘除。

在二叉排序树中，对于拥有两个孩子的结点，它的直接前驱结点要么是叶子结点，要么是没有右孩子的结点，所以删除直接前驱结点可以套用前面两种情况的实现思路。

另外，本题中节点有频次信息，所以 remove 函数的思路就是先判空，然后遍历找到要删除的节点，随后对频次进行判断，如果>1，则这个节点不会消失，给它的 count--即可，如果 count==1，则这个节点要消失，执行上述删除操作即可。

op=3，计数

较为简单，直接使用递归遍历找到要求的节点，然后读取节点中的 count 信息即可。

op=4，找最小值

根据二叉排序树的特点，最左下角的元素最小，所以遍历到最左下角即可。

op=5，输出前驱

前驱节点是指在 BST 中比给定值 val 小的最大节点。在每次循环中，比较当前节点的值与目标值 val：如果目标值小于等于当前节点的值，说明前驱节点应该在当前节点的左子树中，因此将当前节点移动到左子树。如果目标值大于当前节点的值，说明前驱节点应该在当前节点的右子树中，此时将 pred 更新为当前节点的值，并将当前节点移动到右子树。

通过上面的分析，设计出以下的链式存储结构表示二叉排序树：

二叉树节点：

```
struct TreeNode {
    int val; //关键字
    int count; //频次
    TreeNode* left; //左孩子
    TreeNode* right;//右孩子
    TreeNode(int x) : val(x), count(1), left(nullptr), right(nullptr) {} //初始化
};
```

```
class BST {
private:
    TreeNode* root; //根节点
    TreeNode* insert(TreeNode* root, int val) {} //插入
    TreeNode* findMin(TreeNode* root) {}          //找最小值
    TreeNode* remove(TreeNode* root, int val) {} //删除
    int count(TreeNode* root, int val) {}         //计数
    int predecessor(TreeNode* root, int val) {}  //求前驱结点
public: //使用函数重载，private 处理函数内部逻辑，public 处理外部逻辑
    BST() : root(nullptr) {}//构造函数
    void insert(int val) { }//插入
    void remove(int val) { }//删除
    int count(int val) { }  //计数
    int findMin() {}         //找最小值
    int predecessor(int val) { }
};
```

2.2.4 功能说明（函数、类）

插入函数 insert

```
/**
 * @brief 在二叉搜索树中插入节点
 * @param root 树的根节点
 * @param val 待插入的节点值
 * @return TreeNode* 插入后的根节点
 */
TreeNode* insert(TreeNode* root, int val) {
    if (!root) //判空
        return new TreeNode(val);
    // 如果值与当前节点值相等，增加计数
    if (val == root->val)
        root->count++;
    // 如果值小于当前节点值，递归插入左子树
    else if (val < root->val)
        root->left = insert(root->left, val);
    // 如果值大于当前节点值，递归插入右子树
    else
        root->right = insert(root->right, val);
    // 返回插入后的根节点
    return root;
}
```

思路：判空，不断于根节点比较，递归插入。

时间复杂度：对于平衡的二叉搜索树，ASL 可以近似为 $\log_2(n)$，其中 n 是树中节点的数量。最坏情况下，当树变得非常不平衡时，时间复杂度趋向于 O(n)。这是因为在不平衡树中，每次查找并不总是将搜索范围减半，而可能需要遍历大部分节点才能找到目标节点。

寻找最小值 findMin

```
/**
 * @brief 寻找二叉搜索树中的最小节点
 *
 * @param root 树的根节点
 * @return TreeNode* 最小节点的指针
 */
TreeNode* findMin(TreeNode* root) {
    // 当树不为空且存在左子树时，一直往左走，直到找到最小节点
    while (root && root->left)
        root = root->left;
    return root; // 返回最小节点的指针
}
```

思路：判空，不断于根节点比较，递归插入。
时间复杂度：对于平衡的二叉搜索树，ASL 可以近似为 $\log_2(n)$，其中 n 是树中节点的数量。最坏情况下，当树变得非常不平衡时，时间复杂度趋向于 O(n)。这是因为在不平衡树中，每次查找并不总是将搜索范围减半，而可能需要遍历大部分节点才能找到目标节点。

删除 remove

```
/**
 * @brief 从二叉搜索树中删除节点
 * @param root 树的根节点
 * @param val 待删除节点的值
 * @return TreeNode* 删除后的根节点
 */
TreeNode* remove(TreeNode* root, int val) {
    判空
    if (待删除值小于当前节点值) {
        root->left = remove(root->left, val);
    else if (待删除值大于当前节点值)
        root->right = remove(root->right, val);
    else {
        if (root->count > 1)
            root->count--;
        else {
            // 如果左子树为空，用右子树替换当前节点
            if (!root->left) {
                TreeNode* temp = root->right;
                delete root;
                return temp;
            }
            // 如果右子树为空，用左子树替换当前节点
            else if (!root->right) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            }
            // 找到右子树中最小的节点，用该节点替换当前节点
            TreeNode* temp = findMin(root->right);
            root->val = temp->val;
            root->count = temp->count;
            root->right = remove(root->right, temp->val);
        }
    }
    return root; // 返回删除后的根节点
}
```

删除操作分为三种情况：节点无子树、节点只有左子树、节点有左右子树。此外还需要对频次 count 进行判断，只有在 count==1 时才需要删除。
时间复杂度：在平均情况下，时间复杂度为 O(logn)，其中 n 是树中节点的数量。在最坏情况下，当树变得非常不平衡时，时间复杂度趋向于 O(n)。

计数 count

```
/**
 * @brief 计算二叉搜索树中指定值的节点数量
  * @param root 树的根节点
 * @param val 待计数的节点值
 * @return int 节点数量
 */
int count(TreeNode* root, int val) {
    判空
    // 如果待计数值等于当前节点值，返回当前节点的计数
    if (val == root->val)
        return root->count;
    // 如果待计数值小于当前节点值，递归计数左子树
    else if (val < root->val)
        return count(root->left, val);
    // 如果待计数值大于当前节点值，递归计数右子树
    else
        return count(root->right, val);
}
```

递归查找到指定元素，然后返回其频次 count
时间复杂度：在平均情况下，时间复杂度为 O(logn)，其中 n 是树中节点的数量。在最坏情况下，当树变得非常不平衡时，时间复杂度趋向于 O(n)。

寻找前驱 predecessor

```
/**
 * @brief 寻找二叉搜索树中指定值的前驱节点值
* @param root 树的根节点
 * @param val 待寻找前驱节点的值
 * @return int 前驱节点的值，如果不存在则返回-1
 */
int predecessor(TreeNode* root, int val) {
    int pred = -1; // 初始化前驱节点值为-1
    while (root) {
        // 如果待寻找值小于或等于当前节点值，说明前驱节点在左子树
        if (val <= root->val)
            root = root->left; // 移动到左子树继续寻找
        // 如果待寻找值大于当前节点值，说明前驱节点在右子树
        else
            pred = root->val; // 更新前驱节点值
            root = root->right; // 移动到右子树继续寻找
    }
    return pred; // 返回找到的前驱节点值
}
```

时间复杂度：在平均情况下，时间复杂度为 O(logn)，其中 n 是树中节点的数量。在最坏情况下，当树变得非常不平衡时，时间复杂度趋向于 O(n)。

2.2.5 调试分析（遇到的问题和解决方法）

1. 删除操作

本题节点中有 count 信息，所以可以根据它的大小来判断需要删除还是--。同时删除的节点有三种情况，P 为叶子节点，只有一个孩子和有两个孩子，需要考虑全面.同时对根节点等特殊情况需要加以判断，防止越界。

2. 前驱节点的理解

错把前驱结点和父节点搞混，它们是不同的概念。前驱结点：对于二叉搜索树中的每个节点，其前驱节点是小于该节点值的最大节点。即，在中序遍历中，前驱节点是当前节点的前一个被访问的节点。查找规则：如果节点有左子树，前驱节点是其左子树中的最右节点。如果节点没有左子树，需要沿着父节点向上遍历，找到第一个比当前节点值小的父节点，该父节点即为前驱节点。

2.2.6 总结和体会

这道题考察二叉排序树的相关操作，插入删除查找等。我熟悉了 BST 的相关操作，同时也认识到，我们可以根据需要灵活地对数据结构进行调整，比方在此处添加 count 计数。

2.3 哈希表

2.3.1 问题描述

本题针对字符串设计哈希函数。假定有一个班级的人名名单，用汉语拼音（英文字母）表示。要求：首先把人名转换成整数，采用函数 $h(key) = ((...(key[0] * 37 + key[1]) * 37+...) * 37 + key[n-2]) * 37 + key[n-1]$，其中 key[i]表示人名从左往右的第 i 个字母的 ascii 码值(i 从 0 计数,字符串长度为 n，1<=n<=100)。采取除留余数法将整数映射到长度为 P 的散列表中，$h(key) = h(key)\%M$，若 P 不是素数，则 M 是大于 P 的最小素数，并将表长 P 设置成 M。采用平方探测法（二次探测再散列）解决冲突。（有可能找不到插入位置，当探测次数>表长时停止探测）

2.3.2 基本要求

注意计算 h(key)时会发生溢出，需要先取模再计算。

2.3.3 数据结构设计

这道题是一个典型的哈希表题目，要求给定的哈希函数将人名转化为整数，这涉及到 ASCII 的知识，然后使用二次探测(平方探测法)的方法解决冲突，同时，注意里有提出计算 h(key)时会发生溢出，需要先取模再计算。对于表长 P 的选择，使用 ispRrime 判断是否为素数，通过调用 nextPrime 函数找到大于等于 P 的下一个素数作为表长。

我封装了一个哈希表类，包含散列表大小 M，平方探测参数 K 以及使用 vector 存储的散列表 hashTable，同时包含哈希值计算函数，插入与转换函数等。

```
class HashTable {
public:
    HashTable(int P) : M(P), K(P / 2), hashTable(P, 0) {}
    int myhash(const char s[MAX_N]) { }// 哈希函数，将人名转换成整数，并解决冲突
    void insertNames(int N) {}// 插入人名并输出哈希值
private:
    int M; // 散列表大小
    int K; // 平方探测法的参数
    vector<int> hashTable; // 散列表
};
```

2.3.4 功能说明（函数、类）

```
/* 哈希函数，将人名转换成整数，并解决冲突
* param string s
* return newh 新哈希值
*/
int myhash(const char &s) {
    int h = 0;                  // 初始化哈希值
    int i = 0;                  // 初始化字符串遍历索引
    while (s[i])
        h = (h * 37 + static_cast<int>(s[i++])) % M; // 使用给定的哈希函数 h(key)计算哈希值
    // 如果哈希值对应位置未被占用，直接返回
    if (!hashTable[h]) {
        hashTable[h] = 1;
        return h;
    }
    else {
        int newh;
        // 平方探测法解决冲突
        for (int i = 1; i <= K; i++) {
            newh = (h + i * i) % M;         //加
            // 如果找到合适的位置，标记并返回
            if (!hashTable[newh]) {
                hashTable[newh] = 1;
                return newh;
            }
            newh = (h - i * i + M * i) % M; //减
            // 如果找到合适的位置，标记并返回
```

```
            if (!hashTable[newh]) {
                hashTable[newh] = 1;
                return newh;
            }
        }
        return -1; // 表示未找到合适的位置
    }
}
```

使用给定的哈希函数计算哈希值，同时先取模再计算，防止溢出。使用平方探测法解决冲突。
时间复杂度：计算哈希值的时间复杂度为 O(n)，其中 n 是人名的长度。平方探测法的时间复杂度为 O(k)，其中 k 是表长 M。最终插入一个人名的时间复杂度为 O(n + k)
空间复杂度：使用了一个大小为 M 的哈希表，因此空间复杂度为 O(M)
在实际应用中，哈希表的大小 M 可能需要适当调整，以便平衡空间利用率和时间效率。选择合适的哈希函数和解决冲突的方法也是哈希表设计的重要考虑因素。
使用开放定址法，二次探测再散列方法处理冲突的平均查找长度为$-ln(1-a)/a$。 其中 a 是装填因子，$a = N/P$。因此，若设计合理，哈希表的最优和平均插入和查找复杂度为常数级。哈希表的最坏复杂度是 O(n)。此时所有散列值全部冲突，退化为线性表。本题中由于对字符串每一位操作，复杂度为 O(n),n 为字符串长度。

2.3.5 调试分析（遇到的问题和解决方法）

1. 本该输出 1 但是实际输出-表示未找到

发现在 P 小时出现这种错误，猜测应该是 P 的判断不合适，导致哈希表被错误理解为满了。若 P 不是素数，则 M 是大于 P 的最小素数，并将表长 P 设置成 M。

2. 散列值越界

采用开放定址法，二次探测再散列方法处理冲突。注意对哈希值取模加模，防止散列值越界或为负值。采用了比较笨的办法将情况进行列举。

2.3.6 总结和体会

通过这道题，我学习了哈希表的构造及冲突的解决办法。哈希函数可以有很多，冲突可以通过开放定址法或链址法解决。查资料发现当哈希表的长度为素数 4N+3 的形式时可以有效减少哈希冲突的产生

2.4 换座位

2.4.1 问题描述

期末考试，监考老师粗心拿错了座位表，学生已经落座，现在需要按正确的座位表给学生重新排座。假设一次交换你可以选择两个学生并让他们交换位置，给你原来错误的座位表和正确的座位表，问给学生重新排座需要最少的交换次数。

2.4.2 基本要求

两个 n*m 的字符串数组，表示错误和正确的座位表old_chart和new_chart，$old_chart[i][j]$为原来坐在第 i 行第 j 列的学生名字

2.4.3 数据结构设计

这个题目虽然是二维数组，其实按照二维数组实际存储的地址，和一维数组最少交换次数是相同的， 可以证明当遇到与标准排布不同的位置时，循环执行将当前位置的元素与其正确位置交换，直到当前位置正确，这样交换的次数将会是最小的。

查询资料发现这个结论：

最少交换次数证明

1. 问题介绍

问题：对第一行数据进行排序，可以任意交换两个元素，求最少的交换次数是多少？

结论：

$$F(N) = N_{node} - N_{ring}$$

其中，$F(N)$为最少交换次数，N_{node}为数组长度（或节点数），N_{ring}为交换环（或称可交换环）的个数。

交换环：对于元素a_{ij}，其中 i 表示该元素**排序前的下标**，j 表示**排序后的下标**，若存在一个 $n(n>0)$ 个元素序列 $\{a_{ij}\}$，满足：1. $j_n = i_1$；2. $j_k = i_{k+1}(1 \le k < n)$，则称序列 $\{a_{ij}\}$ 为**交换环**。

对于座位交换这类问题，我们关心的是最小交换次数，而不是具体的交换过程，可以将二维座位表看作是一个一维数组，并按照一维数组的方式进行计算最小交换次数。这种思路可以有效地减小空间占用，并简化计算逻辑。

```
class Solution {
public:
    int solve(std::vector<std::vector<std::string>>& old_chart,
std::vector<std::vector<std::string>>& new_chart) {
        int n = old_chart.size();
        int m = old_chart[0].size();
        // 将二维座位表映射到一维数组
        std::vector<std::string> flat_old_chart(n * m);
        std::vector<std::string> flat_new_chart(n * m);
        // 构建从学生名字到座位编号的映射
        std::map<std::string, int> old_seat_map;
        int swaps = 0;
        // 计算最小交换次数，使用类似冒泡排序的思想
        return swaps;
    }
};
```

2.4.4 功能说明（函数、类）

```
/*
 * description: 将二维座位表映射到一维数组
 * param1: n，座位表的行数
 * param2: m，座位表的列数
 * return: 无
 */
vector<std::string> flat_old_chart(n * m); // 存储映射后的错误座位表
vector<std::string> flat_new_chart(n * m); // 存储映射后的正确座位表
// 遍历原始的二维座位表，将其映射到一维数组中
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
        flat_old_chart[i * m + j] = old_chart[i][j];
        flat_new_chart[i * m + j] = new_chart[i][j];
    }
}
```

将原始的二维座位表映射到一维数组中。通过遍历座位表的每个元素，使用公式 i * m + j 将其映射到一维数组的对应位置。这样就把一个二维数组排序最小次数问题转换成了一维数组的，这样我们就可以用交换环的思路来解决了。
时间复杂度：遍历二维座位表并映射到一维数组的时间复杂度为 $O(n * m)$，其中 n 和 m 分别是座位表的行数和列数。
空间复杂度：由于使用了两个一维数组来存储映射后的座位表，空间复杂度为 $O(n * m)$。

```
/*
 * description: 构建从学生名字到座位编号的映射
 * param1: flat_old_chart，映射后的错误座位表的一维数组
 * param2: n，座位表的行数
 * param3: m，座位表的列数
 * return: 无
 */
map<std::string, int> old_seat_map; // 存储学生名字到座位编号的映射
// 遍历映射后的错误座位表的一维数组，构建映射关系
for (int i = 0; i < n * m; ++i) {
    old_seat_map[flat_old_chart[i]] = i;
}
```

构建一个从学生名字到座位编号的映射。通过遍历映射后的错误座位表的一维数组，将每个学生的名字与其对应的座位编号建立映射关系，以便后续能够根据学生名字快速定位到其在座位表中的位置。使 map 数据结构，将学生名字作为键，座位编号作为值，构建映射关系。时间复杂度为 O(n * m)

```
/*
 * description: 计算最小交换次数，使用类似冒泡排序的思想
 * param1: flat_old_chart，映射后的错误座位表的一维数组
 * param2: flat_new_chart，映射后的正确座位表的一维数组
 * param3: old_seat_map，从学生名字到座位编号的映射
 * param4: n，座位表的行数
 * param5: m，座位表的列数
 * return: swaps，最小交换次数
 */
int swaps = 0; // 交换次数计数器
// 遍历映射后的正确座位表的一维数组
for (int i = 0; i < n * m; ++i) {
    // 如果当前位置学生不在正确位置
    while (old_seat_map[flat_new_chart[i]] != i) {
        // 交换当前位置和正确位置的学生
        std::swap(flat_old_chart[i], flat_old_chart[old_seat_map[flat_new_chart[i]]]);
        std::swap(old_seat_map[flat_old_chart[i]],
old_seat_map[flat_old_chart[old_seat_map[flat_new_chart[i]]]]);
        swaps++; // 交换次数加一
    }
}
```

采用类似冒泡排序的思想。通过遍历映射后的正确座位表的一维数组，对每个学生检查其是否在正确的位置上，若不在则执行交换操作，直到所有学生都在正确的位置上。
时间复杂度：遍历正确座位表的一维数组并执行交换操作的时间复杂度为 O(n * m)，其中 n 和 m 分别是座位表的行数和列数。

2.4.5 调试分析（遇到的问题和解决方法）

二维的位置交换可以通过 vector 将其映射到一维的，这样降维后处理起来就很方便了。在已知数学原理后，通过类似冒泡排序的思想，遍历一维数组，对于每个位置，不断交换当前位置的学生和正确位置的学生，直到当前位置正确。在每次成功交换后，更新映射关系并增加交换次数的计数器。

2.4.6 总结和体会

这道题给我最大的启发是降维的思想，二维数组在内存中的存储是连续的，这和一维数组一致，我们可以降维，将复杂问题简单化。

2.5 最大频率栈

2.5.1 问题描述

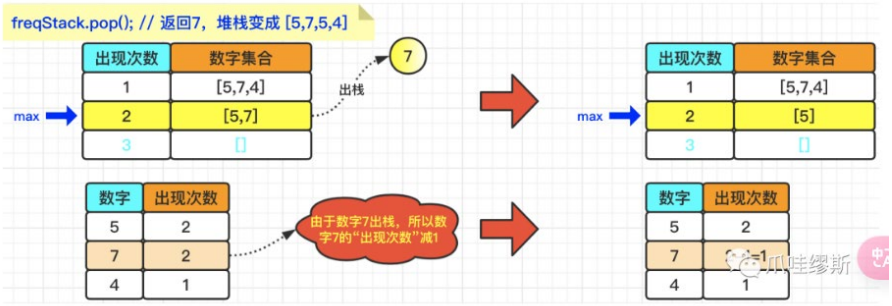
设计一个类似堆栈的数据结构，将元素推入堆栈，并从堆栈中弹出出现频率最高的元素。实现 $FreqStack$ 类:$FreqStack()$ 构造一个空的堆栈。$void\ push(int\ val)$ 将一个整数 val 压入栈顶。$int\ pop()$ 删除并返回堆栈中出现频率最高的元素。如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素。

2.5.2 基本要求

本题需要提交类 FreqStack 的实现，类中需要包含三个名为 FreqStack、push、pop 的函数，下载 tamplate.cpp 查看详细信息，完成类的定义，提交时仅提交类的定义相关代码

2.5.3 数据结构设计

频率表：对于每个元素，我们要统计出其出现的频率，这自然的就会想到哈希表；用一个频率哈希表存储 val 和对应的 freq。对于频率相同的，要按照离顶近的先出，也就是说后入的先出，所以可以再来一个哈希表，key 值是频率，value 是一个栈，栈里存放着元素，所以每次弹出的时候都是弹出最大频率的栈顶。注意一个点，就是这个 maxFreq 一定是连续的，因为在入栈的时候，对于在最大频率处的元素，其也一定存在于比它小的栈里，都是逐步增大才到最大频率栈里的。或许这题的难点也在这里,数据的物理结构和逻辑结构并不需要完全对应,甚至可以完全不一样.因为对外来说,内部的结构都是不可见的,所以只需要能完成指定的行为即可,无需拘泥于特定的形式.




```
class FreqStack {
public:
    FreqStack() : maxFreq(0) {}  // 构造函数，初始化最大频率为 0
    void push(int val) {}// 将元素压入栈
    int pop() {   } // 弹出出现频率最高的元素
private:
    int maxFreq; // 最大频率
    vector<int> elems; // 元素顺序向量
    map<int, int> freq; // 元素频率哈希表
    map<int, stack<int>> group; // 频率对应的元素栈
};
```

2.5.4 功能说明（函数、类）

```
  /*
 * description: 最大频率栈
 * param1: val - 待推入栈的元素
 * return: 无
 */
void push(int val) {
    freq[val]++; // 更新元素频率
    maxFreq = max(maxFreq, freq[val]); // 更新最大频率
    group[freq[val]].push(val); // 将元素推入对应频率的栈和顺序向量
    elems.push_back(val);
}
```

使用两个哈希表和一个向量来管理元素和频率
时间复杂度：主要受到哈希表和栈的操作，推入和弹出的复杂度都接近 $O(1)$
空间复杂度：主要受到哈希表和向量的影响，整体上为 $O(N)$，其中 N 为元素数量

```
/*
 * description: 弹出堆栈中出现频率最高的元素
* return: 返回弹出的元素
 */
int pop() {
    int val = group[maxFreq].top();// 获取当前最高频率的栈顶元素
    group[maxFreq].pop();// 弹出元素
    if (group[maxFreq].empty())// 如果弹出后当前频率的栈为空，降低最大频率
        maxFreq--;
    // 从元素顺序向量中移除弹出的元素
    elems.erase(remove(elems.begin(), elems.end(), val), elems.end());
    freq[val]--; // 更新元素频率
    return val; // 返回弹出的元素}
```

弹出操作主要基于频率的管理，通过哈希表记录元素频率，使用栈记录同一频率的元素，通过顺序向量保留元素的推入顺序。在弹出操作中，根据最大频率获取对应栈的栈顶元素，弹出后更新最大频率，从元素顺序向量中移除该元素，更新元素频率。
时间复杂度：主要受到栈和向量的操作，整体上较为高效，接近 O(1)
空间复杂度：主要受到哈希表和向量的影响，整体上为 O(N)，其中 N 为元素数量

2.5.5 调试分析（遇到的问题和解决方法）

leetcode 的思路大多基于 unordered_map，通过记录元素的频率和对应的频率组来实现，我使用 vector 保存所有元素，用 map<int，int>保存每个元素的频率，使用 map<int,stack<int>>保存每个频率对应的元素组，这两者的作用类似于哈希表中键值对的存储。unordered_map 是哈希表实现的，提供了平均情况下常数时间的查找和插入操作，vector 和 mao 则使用红黑树等数据结构，查找和插入时间在平均情况下为对数时间。两者功能相近，但是 unordered_map 性能会更好些。

2.5.6 总结和体会

通过这道题，我意识到了哈希表的优点和适用情况，通过哈希表中键值对的存储，我们可以实现常数时间的查找和插入操作。

2.6 哈希表 2

题目要求：同哈希表 1 相同，但是注意：第 1 步计算$h(key)$时得到的整数可能很大，需要采用数据类型$usigned\ long\ long\ int$存储，产生的溢出不需处理，其结果相当于对2^64取模的结果。

思路与问题：这道题以为只是将数据结构换成$usigned\ long\ long\ int$，然后不处理溢出即可。但是修改后发现测试样例出错，发现在修改后哈希值可能会很大，需要模表长处理，进行了相关修改，使用 p111_data.cpp 生成的测试样例均通过，但是 OJ 后三个测试点过不了。分析发现可能是在表长 K 的处理上存在问题，我写死成 P/2，但是改了也没有用，可能是还有其他别的错误，因为时间原因，后续再研究。

3. 实验总结

本次作业，涉及了许多数据结构与算法，也学到了许多小知识与技巧。查找又称检索，是数据处理中经常使用的一种重要的运算，查找离不开查找表，分为两大类，静态查找表和动态查找表，也有线性表的查找，如二分查找，分块查找等；树状结构的查找，二叉搜索树，平衡二叉树等；还有哈希表等键值对的查找方法。

在写题的过程中，熟悉了各种数据结构与算法的特点和注意事项，在这个过程中踩了不少坑，也学到了不少知识与技巧，锻炼了我的编程能力。