作业 HW5 实验报告

姓名：郑博远 学号：2154312 日期：2022 年 11 月 30 日

1. 涉及数据结构和相关背景

本次作业涉及到的数据结构为查找表，包含二叉排序树、哈希表等。查找表(Search Table）是由同一类型的数据元素（或记录）构成的集合。由于“集合”中的数据元素之间存在着完全松散的关系，因此查找表是一种非常灵便的数据结构。

对查找表经常进行的操作有：（1）查询某个“特定的”数据元素是否在查找表中；（2）检素某个“特定的”数据元素的各种属性；（3）在查找表中插人一个数据元素；（4）从查找表中删去某个数据元素。若对查找表只作前两种统称为“查找”的操作，则称此类查找表为静态查找表（Static Search Table）。若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存在的某个数据元素，则称此类表为动态查找表（Dynamic Search Table）。

关键字（Key）是数据元素（或记录）中某个数据项的值，用它可以标识（识别）一个数据元素（或记录）。若此关键字可以唯一地标识一个记录，则称此关键字为主关键字（Primary Key）（对不同的记录，其主关键字均不同）。反之，称用以识别若干记录的关键字为次关键字（Secondary Key）。当数据元素只有一个数据项时，其关键字即为该数据元素的值。

查找（Searching）根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素。若表中存在这样的一个记录，则称查找是成功的，此时查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置；若表中不存在关键字等于给定值的记录，则称查找不成功，此时查找的结果可给出一个“空”记录或“空”指针。查找的方法随数据结构不同而不同。

2. 实验内容

2.1 和有限的最长子序列

2.1.1 问题描述

给你一个长度为 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 中元素之和小于等于 queries[i] 的子序列的最大长度。

下面给出子序列的定义：子序列是由一个数组删除某些元素（也可以不删除元素）但不改变元素顺序所得到的一个新数组。

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；

第二行包括 n 个整数，为数组 nums 中元素；

第三行包含 m 个整数，为数组 queries 中元素。

对于 20%的数据，有 1<=n,m<=10；

对于 40%的数据，有 1<=n,m<=100；

对于 100%的数据，有 1<=n,m<=1000。

对于所有数据，1<=nums[i],queries[i]<=106。

输出：一行，包括 m 个整数，为 answer 中元素。

2.1.3 数据结构设计

```
//nums用于存储读入的数组，并进行排序
//preSum用于存储排序后的前缀和
int nums[1000], preSum[1000];
```

2.1.4 功能说明（函数、类）

```
/**
 * @brief 求前缀和数组
 * @param n 数组元素个数
 * @param nums 原数组
 * @param presum 前缀和数组
*/
void getPrefixSum(int n, int* nums, int* presum)

/**
 * @brief 二分查找最小长度
 * @param n 数组元素个数
 * @param nums 前缀和数组
 * @param tgt 所求的序列和
 * @return
*/
int searchBin(int n, int* nums, int tgt)
```

2.1.5 调试分析

在查找不大于 queries[i]的最大前缀和时，可以采用直接遍历的方式，时间复杂度为 O(n)。由于每个元素均为正整数，queries 元素递增，因此我使用二分查找的方法降低时间复杂度，由于不够熟练遇到了一些问题。

1. 在给定每次 l、r 与 mid 的转移关系时出错。一开始由于二分的转移关系写错，导致查找的结果与实际不符；在区间大小为 2 时，还有可能因此陷入死循环（由于 mid = (l+r)/2 – l，l 每次都被赋值为自身）。要解决问题，首先要确定最后找到的区间的左右开闭问题，从而确定 while 循环的终止条件，也确定每次 mid 的转移情况。如本题中我设定最终找到的区间为左闭右开区间，确定初始时 l 为 0、r 为 n。每次 mid 下标的数组值若小于等于 queries[i]则 l=mid（由于左开，l 在区间中），否则 r=mid（r 由于右闭，被移出区间）。最终得到的结果 l+1（下标从 0 开始，因此补 1）便是最长的子序列长度；

2. 题目中给定的 queries[i] 有可能小于数组中最小的元素。由于二分的初始条件是 l = 0、r = n，实际上初始的 l 值对应的答案就不满足小于等于 queries[i] 的条件。因此，要对于 queries[i] 小于所有元素的情况进行特判。

2.1.6 总结和体会

本题中所求的是满足和小于等于 queries[i] 的子序列的最大长度，实际上与元素的排列顺序无关。其次，本题中涉及到数列求和问题，适合使用前缀和的方式来完成。容易想到，首先将输入的数列进行从小到大排序，再对数列求前缀和。此时，前缀和数列满足单调递增，从前往后遍历，最后一个小于等于 queries[i]的 i+1（下标从 0 开始）即为最大的子序列长度。

由于本题数据量较小，在查找不大于 queries[i]的最大前缀和时，可以采用时间复杂度为 O(n)的直接遍历方式。由于每个元素均为正整数，queries 元素递增，因此我使用二分查找的方法降低时间复杂度至 O(logn)。

```
/**
 * @brief 二分查找最小长度
 * @param n 数组元素个数
 * @param nums 前缀和数组
 * @param tgt 所求的序列和
 * @return
*/
int searchBin(int n, int* nums, int tgt)
{
    int l = 0, r = n;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        if (nums[mid] <= tgt)
            l = mid;
        else
            r = mid;
    }

    //特殊处理 数组内所有元素都小于queries[i]的情况
    if (nums[l] > tgt)
        return l - 1;

    return l;
}
```

图 1　二分查找的实现

在使用二分查找时，本次作业中由于起初不够熟练遇到了答案错误、死循环等情况。针对不同的题目背景，要特别注意每次 l 与 r 从 mid 转移的方式、最终所得区间的左右开闭问题等，避免出现错误。

2.2 二叉排序树

2.2.1 问题描述

二叉排序树 BST (二叉排序树) 是一种动态查找表，基本操作集包括：创建、查找，插入，删除，查找最大值，查找最小值等。

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

输入：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）；

输出：一个操作输出 1 行（除了插入操作没有输出）。

2.2.3 数据结构设计

```
//二叉排序树的结点
struct node {
    int value;    //结点数值
    int times;    //出现的次数
    node* lchild, * rchild;
};
```

2.2.4 功能说明（函数、类）

```
/**
 * @brief 创建新结点
 * @param n 新结点指针
 * @param value 数值
*/
void CreateNode(node*& n, int value)

/**
 * @brief 在二叉排序树插入新结点(dfs)
 * @param T 当前搜索的结点
 * @param value 新结点的数值
*/
void Insert(node*& T, int value)

/**
 * @brief 在二叉排序树中查找结点(dfs)
 * @param T 当前搜索的结点
 * @param value 查找的结点数值
*/
int Search(node* T, int value)

/**
 * @brief 删除二叉排序树中的结点(dfs)
 * @param T 当前搜索的结点
 * @param value 删除的结点值
 * @return 是否成功删除
```

```
*/
bool Delete(node*& T, int value)

/**
 * @brief 查询二叉排序树内最小值(dfs)
 * @param T 当前搜索的结点
*/
int GetMin(node* T)

/**
 * @brief 中序遍历二叉排序树 记录到队列q中(dfs)
 * @param T 当前遍历的结点
 * @param q 用于记录的队列
*/
void Traverse(node* T, queue<int>& q)
```

2.2.5 调试分析

1. 忽略了题目中删除数字的操作一次只删除一个，每次都直接将数字对应的结点释放，导致涉及到出现多次数字的删除均出错。按照题目修改，若某数字出现的 times 大于 1，则将 times--；否则将该结点删除释放即可；

2. 删除二叉排序树的某个节点时，若左右子树均存在，有两种方式。本次我使用的方式是将该结点的直接前驱替代所删除的结点，再在二叉排序树中删除其直接前驱。之后，将直接前驱原先的左子树（若存在）接到其直接前驱的双亲结点上（接在左子树或右子树取决于原先直接前驱的位置）。在将直接前驱取代所删除结点时，我只拷贝了结点的值，而忘记拷贝记录该数值出现次数的 times 变量。因此删除结点后若再通过 opt3 功能查询则会出错。修改后，将 times 与 value 值均进行拷贝即可。也可以考虑将 times 与 value 定义为一个结构体 data，对该结构体进行等于重载。这样，在结点存储信息有所变化时，能够更为直观地对结

点所存储的数据信息进行拷贝，不容易遗漏。

2.2.6 总结和体会

本题考察二叉排序树这一数据结构，包含其插入、删除、查找等，具体如下：

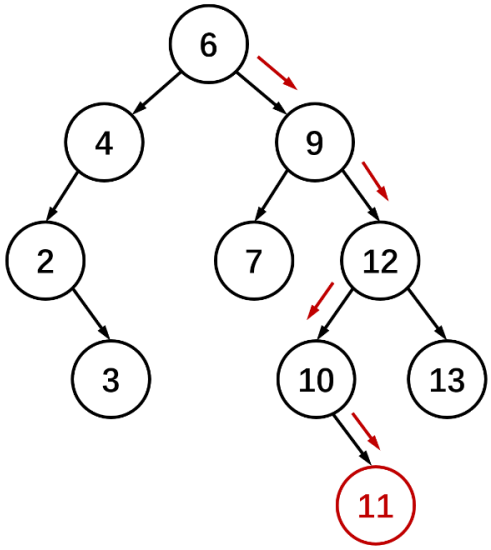

图 2　二叉排序树的插入

对于二叉排序树的插入：从根结点开始遍历，若当前结点元素值小于要插入的元素，则遍历其左子树；若当前结点元素值大于要插入的元素，则遍历其右子树。直到遍历的子树为空，则将元素插入到该位置。如图 2 所示，在原有二叉排序树中插入新元素 11。

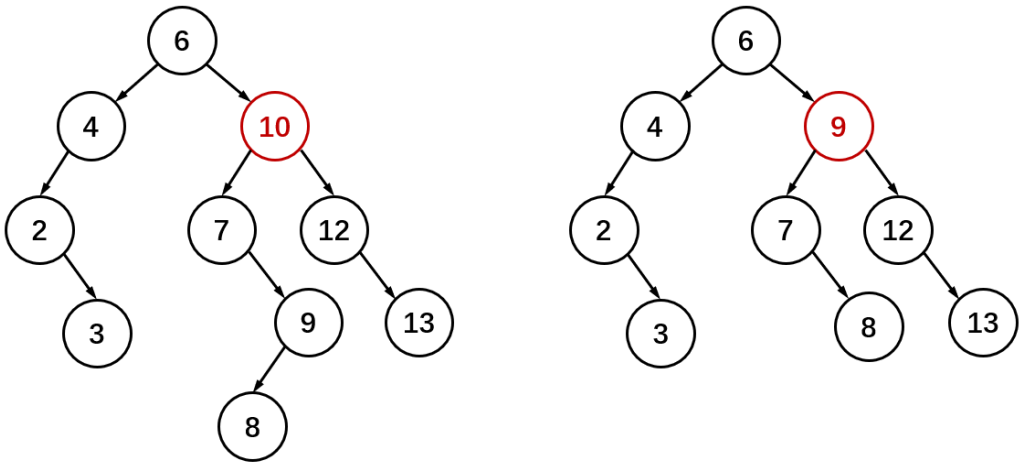

图 3　二叉排序树的删除

对于二叉排序树的删除：若无子树，则直接删除；若仅有左或右子树，则将其接到原先删除结点的位置；若左右子树均存在，将该结点的直接前驱替代所删除的结点，再在二叉排序树中删除其直接前驱。之后，将直接前驱原先的左子树（若存在）接到其直接前驱的双亲结点上（接在左子树或右子树取决于原先直接前驱的位置）。如图 3 所示，在原有二叉树中删除元素 10。

在二叉排序树中查找某元素方法与插入类似。对于一个有 n 个结点的二叉搜索树中，上述基本操作的最优时间复杂度为 O(logn)，最坏为 O(n)。

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。

要求如下：

1. 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)；

2. 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；

3. 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）。

注意：计算 h(key)时会发生溢出，需要先取模再计算。

2.3.2 基本要求

输入：第 1 行输入 2 个整数 N、P，分别为待插入关键字总数、散列表的长度。若 P 不是素数，则取大于 P 的最小素数作为表长。

第 2 行给出 N 个字符串，每一个字符串表示一个人名。

输出：在 1 行内输出每个字符串插入到散列表中的位置，以空格分割，若探测后始终找不到插入位置，输出一个'-'。

2.3.3 数据结构设计

```
//存储该key值是否被占用
bool hash[10010];
```

2.3.4 功能说明（函数、类）

```
/**
 * @brief 判断是否是素数
 * @param n 判断的数
*/
bool isPrime(int n)
/**
 * @brief 获取字符串key值
 * @param str 字符串
 * @param mod 模数
*/
int getKey(char* str, int mod)

/**
 * @brief 二次探测再散列
 * @param hash 记录是否已经占用的数组
 * @param key 原先的key
 * @param mod 模数
 * @return 处理冲突后的key
*/
int avoidConflict(bool* hash, int key, int mod)
```

2.3.5 调试分析

1. 判断一个数是否是质数时，我从 2 开始遍历到 sqrt(n)，若有某个数能够整除 n，则 n 是合数，返回 false；否则在循环结束后返回 true，即 n 是质数。但是当 n 为 1 时会被认为是质数，因此需要进行特判；

2. 采用平方探测法，分别对±1，$\pm2^2$，…，$\pm k^2$（k ≤ m/2）等位置进行探测。因为新探测的位置可能大于表长，因此需要取模保证不会越界。起初采用简单的取模，但由于原先的 key 值可能为负值，取模后仍然会产生负数，无法保证不会产生数组越界；若取模后再加上一个 n，在大多数情况都能正确，但当原先为 n 的负倍数时，会得到 n 仍然越界。因此，需要对新得到的 key 值先取模后，加 n，再次取模，以保证新生成的 key 值在 0 到 n 之间。

2.3.6 总结和体会

本题中考察哈希表的使用。哈希表又称散列表，一种以「key-value」形式存储数据的数据结构，即任意的键值 key 都唯一对应到内存中的某个位置。只需要输入查找的键值，就可以快速地找到其对应的 value。

本题中哈希函数的构造采用经典的 h(key) = ((...(key[0] * 37+key[1]) * 37+...)*37+key[n-2])* 37+key[n-1] 方式；并且用除留余数法，即 h(key) = h(key) % p。处理冲突时，常用的方法有开放定址法、再哈希法、链地址法、建立公共溢出区等等。本题中采用的方法是开放地址法中的二次探测再散列，即 H_i=(H(key) + d_i) MOD m，其中 di =±1，$\pm2^2$，…，$\pm k^2$（k ≤ m/2）。相较于线性探测总能够在哈希表未填满时找到不发生冲突的地址 H_k，二次探测再散列只有在哈希表长 m 为形如 4j+3 的素数时才可能填满，因此题目中存在元素最终找不到 key 值。

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

输入：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;

对于 100%的数据，1<=n,m<=200;

人名为仅由小写英文字母组成的字符串，长度不大于 5。

输出：一个整数，表示最少交换次数。

2.4.3 数据结构设计

```
//将二维数组转换为一维进行存储便于排序
vector<string> old_vec, new_vec;
```

2.4.4 功能说明（函数、类）

```
/**
 * @brief 把二维数组vector数组转一维
 * @param old_chart 错误座位二维数组
 * @param new_chart 正确座位二维数组
*/
void changeIntoVector(std::vector<vector<std::string>>& old_chart, std::vector<std::vector<std::string>>& new_chart)
```

2.4.5 调试分析

1. 对于提交类 Solution 的方式不熟悉，导致上传的 cpp 文件 Compilation

Error 编译出错。注释掉头文件与 main 函数部分提交即可；

2. 我采用循环排序的方法来分析最少交换次数，循环排序中的交换次数即为所求。但每一次循环排序遍历到的元素，都需要找寻其在 old_chart 中的位置来判断，即每一次搜索都额外需要 O(n)的时间复杂度，会导致超时。优化后，采用 map 记录以字符串为 key 值，所在位置为 value 的信息，使得每次查找在 old_chart 中位置的时间复杂度降为 O(logn)，通过该题。

```
Test 1
ACCEPT
Test 2
ACCEPT
Test 3
ACCEPT
Test 4
ACCEPT
Test 5
Time Limit Exceeded
Test 6
Time Limit Exceeded
Test 7
Time Limit Exceeded
Test 8
Time Limit Exceeded
Test 9
Time Limit Exceeded
Test 10
Time Limit Exceeded
```

图 4 每次都遍历 old_chart 查找位置导致的 TLE

2.4.6 总结和体会

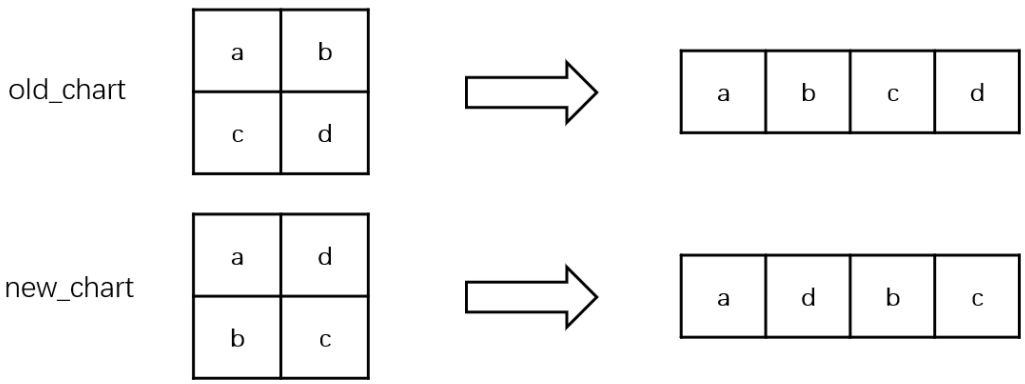

图 5 以样例输入为例将二维数组转为一维数组

本题欲求重新排座所需要的最少交换次数。由于二维数组中每个元素的排列转换到一维数组中仍然保持有序，能够一一对应；且一维数组更方便以排序的思想对待，所以首先将二维数组按行拼接成一维数组。

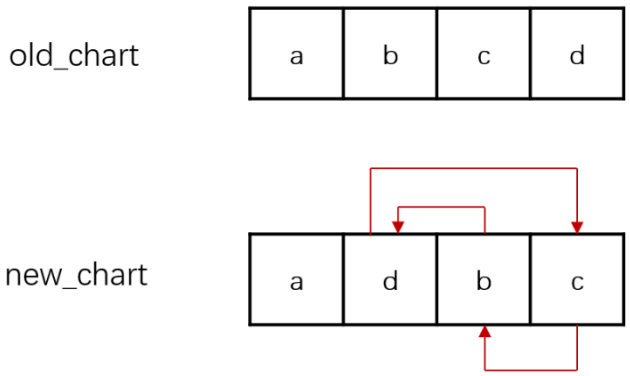

图 6 循环排序将 new_chart 恢复到 old_chart

循环排序（Cyclic Sort）能够通过最少的元素交换次数将数列恢复到有序，在本题中即将打乱后的 new_chart 恢复到 old_chart。这是由于循环排序的特性，即每次都将当前元素放到其正确位置（若不在正确位置，与正确位置所在元素进行交换），能保证最少的交换次数。

在寻找某字符串在 old_chart 中的位置时，若采取直接遍历进行搜索，则每一次搜索都额外需要 O(n)的时间复杂度，会导致超时。我采用 map 记录以字符串为 key 值、所在位置为 value 的信息进行优化，使得每次查找在 old_chart 中位置的时间复杂度降为 O(logn)。

如图 6 所示，能通过循环排序将 new_chart 恢复到 old_chart。上述红色箭头所指向的 d、b、c 三个元素分别占用其他元素所在的正确位置，经过几轮互换之后能够全部到达正确位置，可称之为一个循环节（元素 a 在自己的正确位置上，也可以视作一个循环节）。若数组中不存在循环节，则每个元素都需要一次交换

到达所在的正确位置。若存在循环节，则循环节中的元素只需要元素个数-1 次的次数即可到达正确位置（最后一个元素的位置随前几个数而确定）。因此，最少的交换次数为元素个数-循环节个数。

3. 实验总结

这次作业中，我巩固了对查找表掌握。在题目“和有限的最长子序列”、“二叉排序树”以及“哈希表”中，我分别体会了静态查找表（有序表的查找）、动态查找表（二叉排序树）以及哈希表的使用，同时在最后一题中探究了循环排序（Cyclic Sort）的方法及其交换次数最少的性质特点。

在第一题中，我通过排序+前缀和的方式将原数组化为有序的查找表。尽管O(n)的方式遍历数组也可以在数据量为 1000 的情况下通过本题，但我锻炼了自己在有序表中通过二分查找来找到对应值的能力。在二分的过程中，我在不断地试错中学会了分析意每次 l 与 r 的转移方式、最终所得区间的左右开闭问题等；在第二题中，我练习了二叉查找树中结点的插入、删除、遍历，以及求最小值、求直接前驱节点等基本操作。这是我自研讨课以来第一次编写二叉查找树的代码，但由于整体难度不高通过得比较顺利；在第三题中，我练习了哈希表这一数据结构的基本使用，并通过二次探测再散列的方式来解决哈希表的冲突，较为顺利；在第四题中，我通过学习循环排序解决了交换次数最少这一问题，了解了循环排序的基本性质及其排序方法，并通过 map 优化了时间复杂度。

总体来说，本次作业中我巩固了对静态、动态查找表以及哈希表的了解，让我对查找部分的相关知识了解更为深入，知识点掌握比较熟练、完成比较顺利。希望在以后的编程过程中，能熟练地将这些数据结构知识学以致用。