作业 HW5 实验报告

姓名　　学号　　日期：2024 年 12 月 17 日

1. 涉及数据结构和相关背景

本次选做的四道题目—和有限的最长子序列、二叉排序树、换座位、哈希表主要涉及了以下相关知识：

1. 二分查找：

必须是对有序数据进行处理。它通过每次将查找范围缩小一半来快速定位目标元素。二分查找的实现依赖于递归或迭代地计算中间位置，并将目标值与中间元素进行比较。如果目标值小于中间元素，则继续在左半部分查找；如果大于，则在右半部分查找。这种方法的时间复杂度为 O(log n)，是查找有序数据的最优方法之一。

2. 二叉排序树 BST：

二叉排序树是一种特殊的二叉树，其中每个节点的左子树只包含小于节点值的元素，而右子树只包含大于节点值的元素。这种结构使得查找、插入和删除操作都能以平均 O(log n)的时间复杂度完成。然而，在最坏情况下（如树退化为链表），这些操作的时间复杂度会上升到 O(n)。因此，在实际应用中，通常会和二叉平衡树一起使用。

3. 哈希表：

哈希表是一种基于哈希函数实现的数据结构，它能够在平均情况下以 O(1)的时间复杂度完成查找、插入和删除操作。哈希表通过将键映射到表中的位置来存储数据，这种映射关系由哈希函数确定。有时会涉及哈希冲突的处理（比如二次探测等）。

以及之前的树等数据结构知识。

2. 实验内容

2.1 和有限的最长子序列

2.1.1 问题描述

给一个长度为 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。
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；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：输出一行，包括 m 个整数，为 answer 中元素

2.1.3 数据结构设计

主要涉及的是数组，无需自己定义，直接使用即可。

2.1.4 功能说明（函数、类）

```
/**
 * @brief           计算整数数组的前缀和
 * @param   n       输入数组的长度
 * @param   nums    指向输入整数数组的指针
 * @param   sum     指向用于存储前缀和的整数数组的指针
 */
void input_sum(int n, int* nums, int* sum) {
    memset(sum, 0, n * sizeof(int)); // 初始化sum数组为零
    for (int i = 0; i < n; i++) {
        if (i == 0) {
            sum[i] = nums[i]; // 第一个元素的前缀和就是它本身
        } else {
            sum[i] = sum[i - 1] + nums[i]; // 其他元素的前缀和是前一个元素的前缀和加上当前元素
        }
    }
}
```

```
/**
 * @brief           在有序数组中搜索目标值的插入位置
 * @param   n       输入数组的长度
 * @param   nums    指向输入有序整数数组的指针
 * @param   tgt     需要搜索的目标值
 * @return          目标值应插入的位置索引，如果数组中已存在目标值，则返回其第一个出现的索引
 */
int search(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;
}
```

```
/**
 * @brief           使用选择排序算法对整数数组进行升序排序
 * @param   nums    指向需要排序的整数数组的指针
 * @param   n       数组的长度
 */
void paixu(int nums[], int n)
{
    int k=0,tmp;
    for (int i = 0; i < n; ++i) {
        k = i;
        for (int j = i+1; j < n; ++j) {
            if (nums[k] > nums[j])
            {
                k = j;
            }
        }
        tmp = nums[k];
        nums[k] = nums[i];
        nums[i] = tmp;
    }
    return;
}
```

2.1.5 调试分析（遇到的问题和解决方法）

调试过程描述：

1. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
2. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

问题：

```
Compile Error
Error Messages: (line numbers are not correct)

 In function 'void input_sum(int, int*, int*)':
6:2: error: 'memset' was not declared in this scope
  memset(sum, 0, n * sizeof(int));
  ^~~~~~
6:2: note: 'memset' is defined in header ''; did you forget to '#include '?
2:1:
+#include
 using namespace std;
6:2:
  memset(sum, 0, n * sizeof(int));
  ^~~~~~
```

解决方案：

```
#include <iostream>
#include<string.h>
#include<cstring>
using namespace std;
```

加上<string.h><cstring>的头文件即可

2.1.6 总结和体会

本题整体上来说是简单题，但有一个很巧妙的点是，和有限的最长子序列由最小的前几个数组成。所以 nums 本身的元素次序对结果没有影响，所以要对 nums 进行排序。想到这一点，这个题目就很容易解决了。

2.2 二叉排序树

2.2.1 问题描述

二叉排序树 BST（二叉查找树）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

```
//BST结点
struct node {
    int value;      //结点数值
    int times;      //出现的次数
    node* lchild;
    node* rchild;
};
```

2.2.4 功能说明（函数、类）

```
/**
 * @brief           创建一个新的树节点，并初始化其值、出现次数和左右子节点指针。
 * @param   n       指向需要创建的树节点指针的引用（双重指针）。
 * @param   value   新节点的值。
 */
void createnode(node*& n, int value)
{
    n = new(nothrow) node;//分配新的 node 结构体内存
    if (!n)//检查内存分配是否成功。
        exit(-1);//若失败，退出程序。
    //若成功，初始化新节点的 value、times和左右子节点指针
    n->value = value;
    n->times = 1;
    n->lchild = n->rchild = NULL;
}
```

```
/**
 * @brief           向二叉搜索树中插入一个值，若值已存在，则增加其出现次数。
 * @param   T       指向二叉搜索树根节点的指针的引用（双重指针）。
 * @param   value   需要插入的值。
 */
void insert(node*& T, int value)
{
    if (!T)//T是根结点，如果根结点空
        createnode(T, value);
    //非空
    else if (T->value == value) {//若当前节点值等于待插入值，则增加其出现次数（times）
        T->times++;
    }
    else if (T->value > value) {//若当前节点值大于待插入值，则递归地在左子树中插入。
        if (T->lchild)
            insert(T->lchild, value);
        else
            createnode(T->lchild, value);
    }
    else {//若当前节点值小于待插入值，则递归地在右子树中插入。
        if (T->rchild)
            insert(T->rchild, value);
        else
            createnode(T->rchild, value);
    }
}
```

```
/**
 * @brief           从二叉搜索树中删除一个值，若值存在且出现次数大于1，则减少其出现次数；否则删除该节点。
 * @param   T       指向二叉搜索树根节点的指针的引用（双重指针）。
 * @param   value   需要删除的值。
 * @return          删除操作是否成功（找到并删除了值）。
 */
bool delete_node(node*& T, int value)
{
    if (!T) {
        return false;  //空树，没有该元素
    }
    else if (T->value == value) {
        //a. 若出现次数大于1，则减少出现次数。
        if (T->times != 1)
            T->times--;
        //若无右子节点，则用左子节点替换当前节点并删除当前节点。
        else if (!T->rchild) {//只有单边子树
            node* q = T;
            T = T->lchild;
            delete q;
        }
        //若无左子节点，则用右子节点替换当前节点并删除当前节点。
        else if (!T->lchild) {
            node* q = T;
            T = T->rchild;
            delete q;
        }
        else {//有左右两子树
            node* q = T, * s = T->lchild;
            while (s->rchild) {
                q = s;
                s = s->rchild;
            }
            T->value = s->value;
            T->times = s->times;
            if (q != T)
                q->rchild = s->lchild;
            else
                q->lchild = s->lchild;
            delete s;
```

```
        }
        return true;
    }
    // 若当前节点值大于待删除值，则递归地在左子树中删除。
    else if (T->value > value)
        return delete_node(T->lchild, value);
    //若当前节点值小于待删除值，则递归地在右子树中删除.
    else
        return delete_node(T->rchild, value);
}
```

2.2.5 调试分析（遇到的问题和解决方法）

调试过程描述：

1.整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。

2.生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

问题：提交到 oj 系统上后出现 runtime error 问题，但明明测试了几组数据都是对的，为什么？（如下图）

解决方案：由于编译器的不同（我用的 VStudio），在 VS 上，我的 Search 函数忘记加 return 是正确的，但是在 oj 平台就会报错，添上 return 就好了。

```
int Search(node* T, int value)
{
    if (!T)//空
        return 0;
    else if (T->value == value)
        return T->times;
    else if (T->value > value)
        return Search(T->lchild, value);
    else
        return Search(T->rchild, value);
}
```

```
Test 1
Runtime Error
Test 2
Runtime Error
Test 3
Runtime Error
Test 4
Runtime Error
Test 5
Runtime Error
Test 6
Runtime Error
Test 7
Runtime Error
Test 8
Runtime Error
Test 9
Runtime Error
Test 10
Runtime Error
```

2.2.6 总结和体会

本题主要考察了二叉排序树的相关知识，既涉及了之前所学的树的相关知识，又涉及了最近所学的查找知识，综合性较强。但细细分解下来，也还是比较简单的。但根据函数的具体实现，不难看出 delete 操作是最复杂的，因为要考虑删除结点后依然保持排序树的有序性。通过这道题的练习，也算是加强了我对具体操作的代码实现的理解吧。

2.3 和有限的最长子序列

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。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）
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。

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

本题主要的是数组，未自定义结构体、类等数据结构。

2.3.4 功能说明（函数、类）

```
/**
 * @brief           判断一个整数是否为质数。
 * @param   n       需要判断的整数。
 * @return          若整数为质数，则返回 true；否则返回 false。
 */
bool isPrime(int n)
{
    if (n == 1)
        return false;
    /*质数检测--从 2 遍历到 sqrt(n) (n 的平方根)
    , 检查是否存在能整除 n 的数*/
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0)
            return false;
    return true;
}
```

```
/**
 * @brief           根据字符串生成一个哈希键。
 * @param   str     用于生成哈希键的字符串。
 * @param   mod     哈希键取模的基数。
 * @return          生成的哈希键。
 */
int getKey(char* str, int mod)
{
    const int weight = 37;
    unsigned long long int key = 0;
    //遍历字符串 str 中的每个字符，得到key值
    for (int i = 0; str[i]!=0; i++) {
        key *= weight;
        key += int(str[i]);
    }
    //对 key 取 mod 的余数，作为最终的哈希键返回
    key %= mod;
    return key;
}

/**
 * @brief           在哈希表中解决哈希冲突，找到一个可用的哈希键。
 * @param   hash    哈希表的布尔数组表示，用于记录哪些键已被占用。
 * @param   key     初始哈希键。
 * @param   mod     哈希表的大小（即取模的基数）。
 * @return          可用的哈希键；若无法找到可用键，则返回 ERROR=-1
 */
int avoidConflict(bool* hash, int key, int mod)
{
    //若初始哈希键 key 对应的哈希表位置未被占用，则标记为已占用，并返回 key
    if (!hash[key]) {
        hash[key] = true;
        return key;
    }
    //否则，二次探测 解决哈希冲突：
    int newkey=key;
    bool breakout = false;
    for (int i = 1; i <= mod / 2; i++) {
        for (int j = 1; j >= -1; j -= 2) {
            newkey = (((key + j * i * i) % mod) + mod) % mod;
            if (!hash[newkey]) {
                breakout = true;
                break;
            }
        }
        if (breakout)
            break;
    }
    if (!hash[newkey]) {
        hash[newkey] = true;
        return newkey;
    }
    else
        return ERROR;
}
```

2.3.5 调试分析（遇到的问题和解决方法）

调试过程描述：

1.整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。

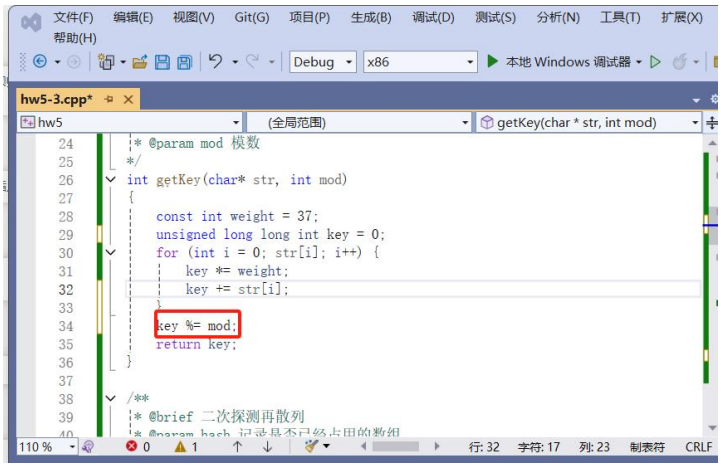
2.生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

问题：测试了几组数据，提交后却还是不通过。

解决方案：其实是因为我偷懒只测试了几组简单的数据，简单数据可以，但是数据一旦计算出来的 key 值太大，就会反应出程序的逻辑问题。经过反复查看，发现是 key 值取模的执行语句放错了位置，应该放在如图所示位置，但我前面的程序都是放在了循环里。

result8.html

```
Test 1
WRONG
Test 2
ACCEPT
Test 3
WRONG
Test 4
WRONG
Test 5
WRONG
Test 6
WRONG
Test 7
WRONG
Test 8
WRONG
Test 9
WRONG
Test 10
WRONG
```



2.3.6 总结和体会

本题主要基于哈希函数，考察了哈希冲突的解决方法之一—二次探测。通过这道题目，加深了我对哈希函数本身执行逻辑的理解，也加深了我对解决哈希冲突的方法的理解。

我想这道题的难点就在于题目中给出的提示“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”，如果没有这句提示，其实很难查出程序不通过的错误。

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。假设一次交换可以选择两个学生并让他们交换位置，给出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

```
/**
 * @class Solution
 * @brief 此类用于解决一个特定的问题，给定两个字符串矩阵（二维向量），
 *        将第二个矩阵（新矩阵）转换为与第一个矩阵（旧矩阵）相同的顺序，
 *        并记录交换操作的次数。转换是通过交换新矩阵中的元素来实现的，
 *        直到新矩阵与旧矩阵完全相同。（即本题的换座位问题）
 */
class Solution {
private:
    vector<string> old_vec, new_vec;
    void changeIntoVector(std::vector<vector<std::string>>& old_chart, std::vector<std::vector<std::string>>& new_chart);
public:
    int solve(std::vector<vector<std::string>>& old_chart, std::vector<std::vector<std::string>>& new_chart);
};
```

2.4.4 功能说明（函数、类）

```
/**
    * @brief 将二维字符串矩阵转换为一维字符串向量。
    * @param old_chart 旧矩阵（二维向量）。
    * @param new_chart 新矩阵（二维向量）。
    *        此方法将旧矩阵和新矩阵的所有元素分别存储到 old_vec 和 new_vec 中。
    */
void changeIntoVector(std::vector<std::vector<std::string>>& old_chart, std::vector<std::vector<std::string>>& new_chart) {
    for (int i = 0; i < old_chart.size(); i++) // 遍历旧矩阵的每一行
        old_vec.insert(old_vec.end(), old_chart[i].begin(), old_chart[i].end()); // 将当前行的元素添加到 old_vec 的末尾
    for (int i = 0; i < new_chart.size(); i++) // 遍历新矩阵的每一行
        new_vec.insert(new_vec.end(), new_chart[i].begin(), new_chart[i].end()); // 将当前行的元素添加到 new_vec 的末尾
}
```

```
/**
    * @brief 解决将新矩阵转换为旧矩阵顺序的问题，并返回交换操作的次数。
    * @param old_chart 旧矩阵（二维向量）。
    * @param new_chart 新矩阵（二维向量）。
    * @return 返回将新矩阵转换为旧矩阵所需的最小交换次数。
    */
int solve(std::vector<std::vector<std::string>>& old_chart, std::vector<std::vector<std::string>>& new_chart) {
    changeIntoVector(old_chart, new_chart); // 将二维矩阵转换为一维向量
    int cnt = 0; // 初始化交换次数计数器
    std::map<std::string, int> mp; // 创建一个映射，用于存储旧向量中每个元素的位置

      // 填充映射表，将旧向量中的每个元素映射到其索引位置
    for (int i = 0; i < old_vec.size(); i++)
        mp[old_vec[i]] = i;

        // 遍历新向量的每个元素，并尝试将其与旧向量的对应元素匹配
    for (int i = 0; i < new_vec.size(); i++) {
        // 如果新向量中的当前元素与旧向量中的对应元素不匹配
        while (new_vec[i] != old_vec[i]) {
            // 交换新向量中的当前元素与其在旧向量中对应位置上的元素
            swap(new_vec[i], new_vec[mp[new_vec[i]]]);
            cnt++; // 增加交换次数计数器
        }
    }
    return cnt; // 返回总的交换次数
}
```

2.4.5 调试分析（遇到的问题和解决方法）

调试过程描述：

1.整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。

2.生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

问题：测试的时候通过，oj 上没通过

解决方案：依旧是因为只测试了几组简单数据，当数据量大的时候，程序的逻辑问题就会显现出来。排查出逻辑问题即可。

体会：测试数据一定一定要有代表性！！！！

2.4.6 总结和体会

本题 n 最大能够达到 200，意味着座位总数最多为 40000 个，所以如何快速查找就是要考虑的问题，也是与查找相关联的地方。因此采用哈希表来快速查找学生的位置，以此提高算法的效率。

3. 实验总结

本次实验主要围绕查找的相关知识，设置了一系列情景来运用这些知识。通过本次实验的练习，加深了我对各种查找算法的理解，并且在不同实际问题下选择不同的算法的过程中，也加深了我对这些不同查找的时间复杂度等特点的理解，更好地学以致用。通过复杂的实际问题，选择合适的数据结构，提高了自身的编程能力。