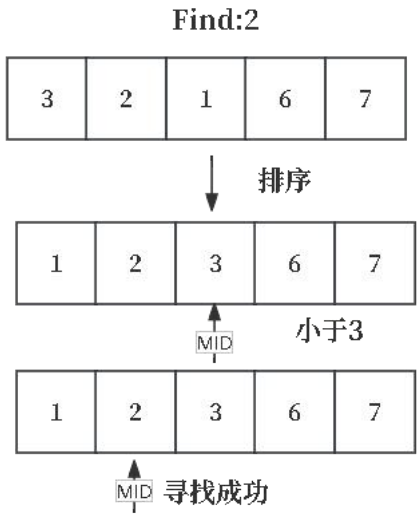
作业 HW5* 实验报告

姓名：朱俊泽 学号：2351114 日期：2024 年 12 月 11 日

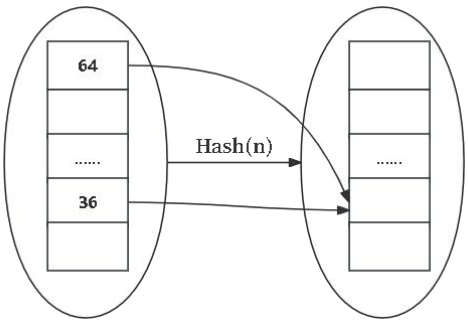
1. 涉及数据结构和相关背景

按照思想大体可以分为三类结构：线性序列表查找、树结构查找及哈希查找（包括索引查找）。其中线性序列表查找主要用于数组、顺序表、链表等数据结构的元素查找。除了朴素的顺序查找方法（O(n^2)），有序查找也为常用的线性序列表查找。构建有序表需要先对序列进行排序
（八大排序算法，优先推荐使用快速排序），常用查找方法为二分(O(logn))。



树结构查询主要用到了二叉排序树(BST)，其中 BST 的搜索原理与二分查找一致。在树结构的动态插入/删除过程中需要特别注意。

哈希表的查询方法实际上是一种索引查找方式，关键点在于构建映射函数以及避免哈希冲突。



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。

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

下载编译并运行 p126_data.cpp 以生成随机测试数据

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

本题实质是从序列中从小到大选取 queries 序列中的数字并求和，并要求这个和不能超过某个 nums[i]。
① 因此首先先要对 queries 进行排序，这里可供选择的排序算法有很多，选择对大数效率更高的快速排序 QuickSort 方法。
注意到 QuickSort 的时间复杂度为 O(nlogn)~O(n^2)的不稳定排序算法。
②排序后就需要依次探测求和值。这里发现由于需要比较多次，就不妨一次性将所有的顺序序列的前缀子列求和并保存备用。
最后做一个前缀和来计算和。

$$S_n = S_{n-1} + a_n$$

最后用二分算法来查询 idx。
因此总时间复杂度为 O(nlogn~ n^2 + n + nlogn)。

2.1.4 功能说明（函数、类）

1.二分查找：

这里所用到二分查找与朴素二分查找有所区别，这里需要寻找的是比目标数小的最大值。
因此最终结束查找条件是 nums[mid]<=dst;

```
// 二分查找：返回不超过目标值 dst 的最大位置
int BinarySearch(int dst, int nums[], int low, int high) {
    int result = -1;  // 用于记录符合条件的最大位置
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (nums[mid] <= dst) {  // 目标值大于等于 mid 位置的值，符合条件
            result = mid;  // 更新result
            low = mid + 1;  // 继续查找右半部分
        } else {
            high = mid - 1;    // 查找左半部分
        }
    }
    return result + 1;  // 返回的是位置，所以需要加1
}
```

2. 前缀和

$S_n = S_{n-1} + a_n$

```
// 计算nums的前缀和
vector<int> prefix_sum(n);
prefix_sum[0] = nums[0];
for (int i = 1; i < n; i++) {
    prefix_sum[i] = prefix_sum[i - 1] + nums[i];
}
```

3. Quicksort

这里选择效率较高的快速排序方法进行排列，其用到了分治思想的方法，QuickSort 具体操作流程如下：① 选择序列的第一个元素为枢轴元素，并用 middle 指针指向该枢轴元素的位置。
以下步骤的目的是将序列调整顺序，直到枢轴元素左侧的元素的值都小于枢轴元素，枢轴元素右侧的所有元素的值都大于枢轴元素。
② 头尾 i，j 指针开始移动。
i)先移动尾指针 j，从尾向头移动，直到指向一个小于枢轴元素的值；
ii)之后开始移动头指针 i，从头向尾移动，直到指向一个大于枢轴元素的值；
iii)执行完成 i)、ii)后，交换 i，j 指针所指向位置的元素；
重复以上 i)、ii)、iii)步骤，直到 i，j 指针重合，交换 i（或 j）指向位置与 middle 指针指向的枢轴元素。在该过程中 iii)步骤可能一直不会执行；
③ 执行完毕①、②后，枢轴元素已经调整到序列的中间部分，其左侧的元素都比枢轴元素更小，其右侧的元素都不枢轴元素更大。这时候把两端的序列也当成新序列，重新传入 QuickSort（）函数进行排序（分治思想），每次重复执行①、②步骤；
④ 直到最终传入数列的长度为 1，结束排序。
QuickSort()代码如下：

```cpp
// 快速排序的划分函数
int Partition(int nums[], int low, int high) {
    int pivot = nums[high];  // 选择最后一个元素作为基准
    int i = low - 1;  // i 是小于基准的元素的最右边的索引

    for (int j = low; j < high; j++) {
        if (nums[j] <= pivot) {  // 如果当前元素小于等于基准
            i++;
            swap(nums[i], nums[j]);  // 交换小于等于基准的元素到左侧
        }
    }
    swap(nums[i + 1], nums[high]);  // 将基准元素放到正确位置
    return i + 1;  // 返回基准元素的索引
}

// 快速排序函数
void QuickSort(int nums[], int low, int high) {
    if (low < high) {
        int pivotIndex = Partition(nums, low, high);  // 获取基准元素的索引
        QuickSort(nums, low, pivotIndex - 1);  // 对左侧子数组排序
        QuickSort(nums, pivotIndex + 1, high);  // 对右侧子数组排序
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

2.1.6 总结和体会

本题方法基础，思维量也很小，但是可以提醒完成的时候要养成良好的编程习惯，尤其是推演前缀和公式以及二分查找结束条件的部分，侧面说明了递推思想以及函数思想在算法设计中的作用。

2.2 题目二 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 结构设计：基本的树结构：结点结构定义：数据域+左右孩子指针域：树根结点：结点的创建与初始化：

```
// 创建新节点
Node CreateNode(int value) {
    Node newNode = (Node)malloc(sizeof(TreeNode));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
typedef TreeNode* Node;
struct BinaryTree {
    Node root = NULL;
};
```

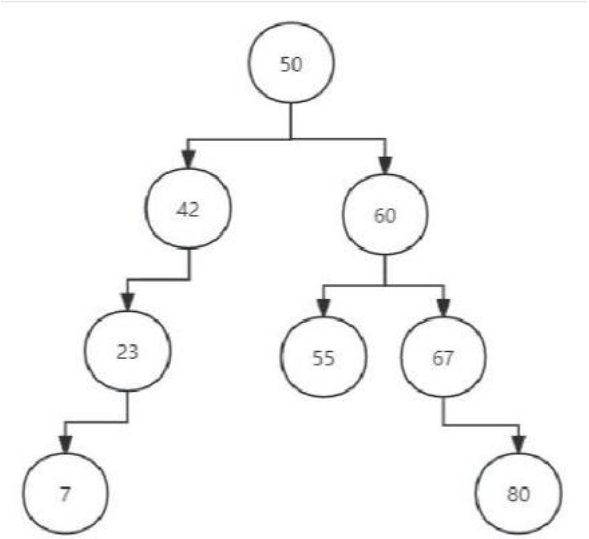
```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};
```

BST 二叉排序搜索树是的结构要求如下：

（1）若根节点有左子树，则左子树的所有节点都比根节点小。

（2）若根节点有右子树，则右子树的所有节点都比根节点大。

（3）根节点的左，右子树也分别是二叉排序树。



2.2.4 功能说明（函数、类）

1.插入

确定插入位置的方式是找到待连接的叶子结点。从上往下依次找。从树根开始，如果待插入的结点的值比结点小，则往左寻找；如果插入结点值比结点大，则往右寻找。直到找到最终的叶子结点停止（叶子结点的左右孩子指针都为空），并插入树中，成为新的叶子结点。

```
// 插入新节点
void Insert(BSTree& tree, int value) {
    Node newNode = CreateNode(value);
    Node current = NULL, parent = NULL;

    if (!tree.root) {
        tree.root = newNode;
    } else {
        current = tree.root;
        // 找到插入位置
        while (current) {
            parent = current;
            if (value <= current->data) {
                current = current->left;
            } else {
                current = current->right;
            }
        }

        if (value <= parent->data) {
            parent->left = newNode;
        } else {
            parent->right = newNode;
        }
    }
}
```

2. 搜索

搜索的逻辑与插入类似，BST 的搜索与二分查找实际上是类似的思路。

但是注意到，这里搜索需要找到全部的目标值相同的结点，因此在搜索成功后应当继续搜索。这里选用的方法是在选取到目标结点后，把目标结点当做根结点继续检索，直到检索完整棵树。

```
// 查找节点
Node Search(BSTree tree, int value) {
    Node current = NULL;
    if (!tree.root) {
        return NULL;
    } else {
        current = tree.root;
        while (current) {
            if (current->data == value) {
                return current;
            }
            if (value <= current->data) {
                current = current->left;
            } else {
                current = current->right;
            }
        }
        return NULL;
    }
}
```

3. 输出最小

这里直接一直向左查找即可。

```
// 获取最小值节点
int GetMin(Node root) {
    Node current = NULL;
    if (!root) {
        return -1;
    } else {
        current = root;
        while (current->left) {
            current = current->left;
        }
        return current->data;
    }
}
```

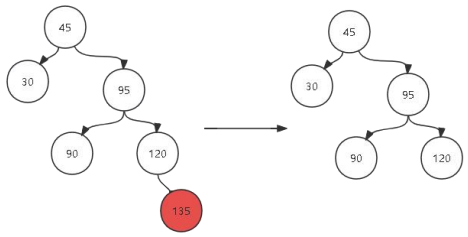
4. 删除

删除结点的关键是要保持二叉排序树的形式。

这里可以分为三种情况：

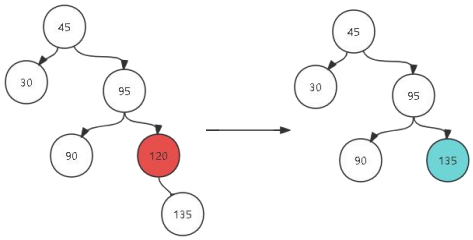
① 删除结点为叶子结点。

该情况可以直接删除。



② 删除结点只有左/右支。

只需让后继结点继承即可。

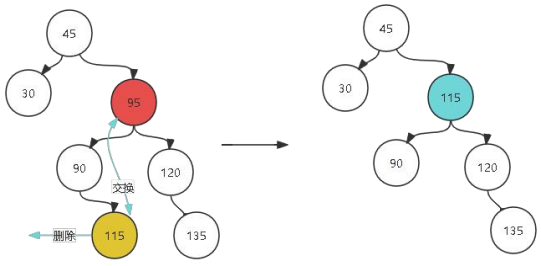


③ 删除结点同时有左/右支。

本情况考虑如下：

例如删除结点序列 30,55,95,90,115,120,135 中的 95，则为了保证结构不改变，应当考虑用该结点右支部分的最小值来代替，即用该结点的前/后继值来代替，在次序列中就是用 115 来继承 95 的位置。

首先后继值可以是从左子树的最右结点寻找。如下图：




```
// 删除节点
int Delete(BSTree& tree, int value) {
    Node current = NULL, parent = NULL, temp = NULL;
    if (Search(tree, value) == NULL) { // 如果节点不存在
        return 0;
    } else {
        current = Search(tree, value);
        if (!current->left && !current->right) {  // 叶子节点
            parent = SearchParent(tree, value);
            if (parent->right == current) parent->right = NULL;
            if (parent->left == current) parent->left = NULL;
            free(current);
        } else if (current->right && !current->left) {  // 只有右子树
            temp = current->right;
            current->data = temp->data;
            current->right = temp->right;
            current->left = temp->left;
            free(temp);
        } else if (current->left && !current->right) {  // 只有左子树
            temp = current->left;
            current->data = temp->data;
            current->left = temp->left;
            current->right = temp->right;
            free(temp);
        } else {  // 左右子树都有
            temp = current;
            parent = current->left;
            // 找到左子树最右侧的节点
            while (parent->right) {
                temp = parent;
                parent = parent->right;
            }
            current->data = parent->data;  // 替换值
            if (temp != current) {  // 左子树有右子树
                temp->right = parent->left;
            } else {  // 左子树无右子树
                temp->left = parent->left;
                free(parent);
                parent = NULL;
```

2.2.5 调试分析（遇到的问题和解决方法）

2.2.6 总结和体会

本题目实现了简单的 BST 二叉排序树，并实现了动态查找的功能。在这个过程中需要特别

注意有关二叉排序树删除的三种情况，针对不同情况选择好不同的继承覆盖结点；

2.3 题目三 换座位

2.3.1 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。

2.3.2 基本要求

输入描述

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

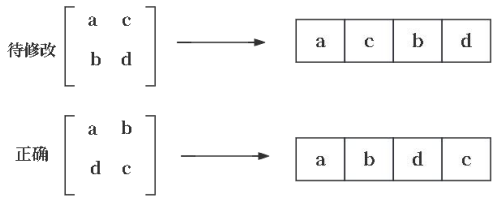
下载编译并运行 p138_data.cpp 生成随机测试数据

输出描述

一个整数，表示最少交换次数

2.3.3 数据结构设计

本题目算法的实现与线性序列交换问题相同。这里的两个矩阵可以通过拉伸变换成两个线性序列，其中现在座位表所得的线性序列表示的是待修改的序列，而新座位表所得的线性序列表示的是正确的序列。



拉伸后，这里按照线性序列的位置调整交换方法即可得到交换次数。

2.3.4 功能说明（函数、类）

1. 序列拉伸

2. 循环查找

由于需要多次查找正确元素的位置索引，这里需要采取一种高效的查找方式。因此这里选择使用哈希表 map 结构来存储正确元素和其位置索引构成的键值对关系：

首先定义 hashmap：

```
void Creatmap(int n, int m) {
    //创建old的hashmap
    for (int i = 0; i < Old.size(); i++) {
        oldmap.insert(pair<string, int>(Old[i], i));    未定义标识符 "pair"
    }
    //创建new的hashmap
    for (int i = 0; i < New.size(); i++) {
        newmap.insert(pair<string, int>(New[i], i));    未定义标识符 "pair"
    }
}
```

然后快速查找

```
int FindStrKey(int state, string str) {    "string" 不是类型名
    map<string, int>::iterator iter;    未定义标识符 "map"
    if (state == 1) {
        return oldmap.find(str)->second;
    }
    if (state == 2) {
        return newmap.find(str)->second;
    }
}
```

最后计算环

```
int Calculate_Circle() {
    int cnt = 0;
    int j;   //标记指针
    string j_tmp;    应输入";"
    //遍历记录数组Checked归0
    for (int i = 0; i < 50000; i++) {
        Checked[i] = 0;
    }
    for (int i = 0; i < Old.size(); i++) {
        j = i;
        while (Checked[i] != 1) {
            j_tmp = Old[j];    未定义标识符 "j_tmp"
            if (Old[j] == New[j]) {
                Checked[j] = 1;
                cnt++;
                break;
            }
            else {
                swap(j, FindStrKey(1, New[j]), Old[j], New[j]);
                j = FindStrKey(1, j_tmp);
                Checked[j] = 1;
            }
        }
    }
    return cnt;
}
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.6 总结和体会

对于表格相关的问题，将二维数据结构展开为一维后可以简化操作逻辑。通过索引映射和线性处理，大幅减少复杂度。

2.4 题目四 家族树

2.4.1 问题描述

人类学教授对居住在孤岛上的人们及其历史感兴趣。他收集了他们的家谱进行一些人类学实验。为了实验，他需要用电脑来处理家谱。为此，他把它们翻译成文本文件。以下是表示家谱的文本文件的示例。

John

 Robert

  Frank

  Andrew

Nancy
David
每行包含一个人的名字。第一行的名字是这个家谱中最古老的祖先。家谱只包含该祖先的后代。他们的丈夫或妻子没有在家庭树上显示。一个人的孩子比父母缩进一个空格。例如，Robert 和 Nancy 是 John 的孩子，Frank 和 Andrew 是 Robert 的孩子。David 比 Robert 缩进了一个空格，但他不是 Robert 的孩子，而是 Nancy 的孩子。为了用这一方法表示一个家谱，教授将一些人从家谱中排除，使得家谱中没有人有两个父母。
为了实验，教授还收集了家属的文件，并提取了每个家谱中关于两人关系的陈述语句。以下是关于上述家庭陈述语句的一些例子。
John is the parent of Robert.
Robert is a sibling of Nancy.
David is a descendant of Robert.
对于实验，他需要判断每个陈述语句是否正确。例如，上面的前两个语句是正确的，最后一个语句是错误的。但是这个任务是十分无聊的，他想通过电脑程序来判断。
需要支持的查询有以下几种：
X is a child of Y.
X is the parent of Y.
X is a sibling of Y.
X is a descendant of Y.
X is an ancestor of Y.
注意：一个人的祖先、后代、兄弟可以是自己，但父母、孩子则不行。

2.4.2 基本要求

输入

输入包含若干组测试用例，每个测试用例由一个家谱和一个陈述句集合组成。

每个测试用例的第一行给出两个整数 n,m,(0<n,m<1000)，分别表示家谱中的名字和陈述语句的数目。

接下来输入的每行少于 70 个字符。名字的字符串仅由不超过 20 个字母字符组成。在家谱的第一行中给出名字前没有前导空格，在家谱中其他的名字至少缩进一个空格，表示是第一行给出的那个名字（家族的最早祖先）的后代。若家谱中一个名字缩进 k 个空格，则下一行最多缩进 k+1 个空格。

本题假定，除了最早的祖先外，在家谱中，每个人都有其父母。在同一个家谱中同样的名字不会出现两次。家谱的每一行结束的时候无多余的空格。

再接下来每句陈述句占一行。在家谱中没有出现的名字不会出现在陈述句中。陈述句中连续的单词被一个空格分开。每句陈述句在行首和行尾没有多余的空格。

用两个 0 表示输入的结束。

对于 20%的数据，有 0<n<=20, 0<m<=40，且只有 1 个测试集

对于 40%的数据，有 0<=n<=100, 0<=m<=200，且超过一个数据集

对于 100%的数据，有 0<=n<=5000, 0<=m<=10000，且超过一个数据集

本题拿满分需要使用哈希表，可以使用 std::map

输出

对于测试用例中的每句陈述语句，程序输出一行，给出 True 或 False。

每个测试用例后要给出一个空行，就算只有一个测试用例。

2.4.3 数据结构设计

设计了两个 map
第一个 children map 用于存储名字和对应的 vector《string》儿子序列，存贮每个节点的子节点
第二个 parent map 用于存储每一个节点的父亲节点。

```
// 树节点关系
map<string, vector<string>> children; // 每个节点的子节点
map<string, string> parent;           // 每个节点的父节点
```

2.4.4 功能说明（函数、类）

1. 解析家谱

· 循环读取 n 行输入。
· 统计每行前的空格数，作为缩进层级。
· 去掉缩进部分，提取名字并存储到 nodes 中。

遍历 nodes 中每个节点的层级和名字。将所有层级大于或等于当前节点的祖先出栈。 这样可以确保栈顶始终是当前节点的直接父节点。

· 如果栈不为空，栈顶节点是当前节点的父节点。
· 更新 children 和 parent 映射表。将当前节点（层级和名字）压入栈，作为下一次迭代的祖先链一部分。

```
// 解析家谱
void parseGenealogy(int n) {
    children.clear();
    parent.clear();

    vector<pair<int, string>> nodes; // 存储每个节点的缩进层级和名字
    string line;

    for (int i = 0; i < n; i++) {
        getline(cin, line);
        int level = 0;

        // 统计缩进空格数
        while (level < line.size() && line[level] == ' ') {
            level++;
        }

        string name = line.substr(level);
        nodes.emplace_back(level, name);
    }

    stack<pair<int, string>> ancestors; // 栈用于维护当前祖先链
    for (const auto& node : nodes) {
        int level = node.first;
        string name = node.second;

        // 弹出所有比当前节点层级高的节点
        while (!ancestors.empty() && ancestors.top().first >= level)
        {
            ancestors.pop();
        }

        // 如果栈不为空，当前节点的父节点就是栈顶
        if (!ancestors.empty()) {
            string parentName = ancestors.top().second;
            children[parentName].push_back(name);
            parent[name] = parentName;
        }

        // 将当前节点入栈
        ancestors.push({level, name});
    }
```

2.分析关系问题

· 使用 y = y.substr(0, y.size() - 1) 去掉末尾的逗号。
· 如果查询字符串结构不符合预期，则后续处理可能出错。
如果查询不符合格式要求，直接返回 false。
Child:
· parent.find(x)：检查 x 是否有父节点。
· parent[x] == y：验证 x 的父节点是否为 y。
Parent:
· children.find(x)：检查 x 是否有子节点。
· find(children[x].begin(), children[x].end(), y)：搜索 y 是否在 x 的子节点列表中。
Sibling:
· 检查 x 和 y 是否都有父节点。
· 比较 parent[x] 和 parent[y] 是否相同。
Descendant:
· 特殊情况：x 是自己的后代，直接返回 true。
· 从 x 开始，沿着 parent 映射向上遍历，直到根节点。
· 如果遍历过程中遇到 y，返回 true。

```cpp
bool checkQuery(const string& query) {
    istringstream iss(query);
    string x, is, a_or_the, relation, of, y;

    // 读取查询的格式
    iss >> x >> is >> a_or_the >> relation >> of >> y;
    y=y.substr(0,y.size()-1); // 去掉末尾的逗号
    //cout<<x<<" "<<y<<endl;
    //y=y.substr(0,y.size()-3); // 去掉末尾的逗号

    if (is != "is" || (of != "of" && relation != "sibling"))
    {
        return false; // 如果格式不正确，直接返回 False
    }
    // 根据关系类型处理查询
    if (relation == "child")
    {
        // X is a child of Y
        return (parent.find(x) != parent.end())&& parent[x] == y;
    }
    else if (relation == "parent")
    {
        // X is the parent of Y
        return children.find(x) != children.end() &&
               find(children[x].begin(), children[x].end(), y) != children[x].end();
    }
    else if (relation == "sibling")
    {
        // X is a sibling of Y
        if (parent.find(x) == parent.end() || parent.find(y) == parent.end()) return false;
        return parent[x] == parent[y] ;
    }
    else if (relation == "descendant")
    {
        // X is a descendant of Y
        if(x==y)return true;
        string current = x;
        while (parent.find(current) != parent.end()) {
            current = parent[current];
            if (current == y) return true;
        }
        return false;
```

```cpp
    else if (relation == "descendant")
    {
        // X is a descendant of Y
        if(x==y)return true;
        string current = x;
        while (parent.find(current) != parent.end()) {
            current = parent[current];
            if (current == y) return true;
        }
        return false;
    }
    else if (relation == "ancestor")
    {
        // X is an ancestor of Y
        if(x==y)return true;
        string current = y;
        while (parent.find(current) != parent.end()) {
            current = parent[current];
            if (current == x) return true;
        }
        return false;
    }

    return false;
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.6 总结和体会

这是一道围绕树形数据结构（家谱）展开的关系查询问题。输入是一颗家谱树和若干关系查询，目标是利用高效的数据结构和算法验证查询的正确性。题目涉及了树的构建、基本树关系的判断（父子、兄弟、祖先、后代等）、输入输出格式解析等多个层面的考验。

简单考察了双向的 map 查找

2.5 题目五 哈希表 2

2.5.1 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:

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：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。

2.5.2 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

2.5.3 数据结构设计

散列表
长度：MMM
每个位置存储整数（标记是否被占用）

冲突解决路径:

哈希值 h(key)%Mh(key) \% Mh(key)%M

冲突后，按平方探测路径递增：(pos+i2)%M(pos + i^2) \% M(pos+i2)%M。

```
    vector<int> hashTable(M, -1); // 初始化哈希表，-1 表示空
    vector<string> keys(N);
```

2.5.4 功能说明（函数、类）

素数调整 getNextPrime：

检查并调整散列表长度 PPP，确保其为素数若 PPP 不是素数，返回大于等于 PPP 的最小素数。

```
// 判断是否是素数
bool isPrime(int num) {
    if (num <= 1) return false;
    for (int i = 2; i * i <= num; ++i) {
        if (num % i == 0) return false;
    }
    return true;
}

// 获取大于等于P的最小素数
int getNextPrime(int P) {
    while (!isPrime(P)) {
        P++;
    }
    return P;
}
```

哈希值计算 computeHash：将字符串转换为整数，计算其哈希值。使用累积乘法和累加法，避免冲突集中。

```
// 计算哈希值
unsigned long long computeHash(const string& key) {
    unsigned long long hash = 0;
    for (char ch : key) {
        hash = hash * 37 + ch; // 按照公式累积计算
    }
    return hash;
}
```

冲突处理（平方探测法）：

初始位置由哈希值确定。

发生冲突时，按照平方探测法重新计算插入位置。

```
int origin = pos;
int k = time + 1;
if (k % 2 != 0)
{
    pos += ((k + 1) / 2) * ((k + 1) / 2);
    pos = pos % P;

}
else
{
    pos -= (k / 2) * (k / 2);
    pos = (pos % P + P) % P;
}
if (list[pos] == "\0")
    return;
pos = origin;
time++;
```

2.5.5 调试分析（遇到的问题和解决方法）

2.5.6 总结和体会

哈希表的核心在于通过哈希函数快速定位数据，同时利用冲突解决策略来应对数据分布不均的问题。本题中，主要应用了平方探测法解决冲突，体现了哈希表性能的关键因素。

2.6 题目六 最大频率栈

2.6.1 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。

2.6.2 基本要求

输入描述

第一行包含一个整数 n

接下来 n 行每行包含一个字符串（push 或 pop）表示一个操作，若操作为 push，则该行额外包含一个整数 val，表示压入堆栈的元素

对于 100%的测试数据，1<=n<=20000，0<=val<=10^9，且当堆栈为空时不会输入 pop 操作

输出描述

输出包含若干行，每有一个 pop 操作对应一行，为弹出堆栈的元素

2.6.3 数据结构设计

题目需要实现 FreqStack，模拟类似栈的数据结构的操作的一个类。
FreqStack 有两个方法:
push(int val)，将 val 推入栈中；
pop()，它移除并返回栈中出现最频繁的元素；如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。
首先，用一个哈希表来统计 push 的 val 的频次。同时再创建一个哈希表，键是不同的频次，值是一个 Stack，用以存储具体相同频次的不同的 val，这是为了能满足 pop 时相同频次仍能先进后出，即靠近栈顶的 val 先被 pop。
还要用一个 maxFreq 来指向最大频次，因为 pop 时要先从 maxFreq 的栈中出栈 val。因为 pop 是每次弹出一个 val，并且不是把所有的 val 都弹出了，而且 push 也一次一个 val 的 push，因此 maxFreq 一定以 1 为步长在变化。频次对应的是一个栈，栈为空说明此频次没有 val 了，总

是从 maxFreq 对应的栈 pop，所以当 maxFreq 对应的栈为空时，maxFreq 就要减 1。

2.6.4 功能说明（函数、类）

1.Push 函数

```
void push(int val) {
    stack<int> S;     不允许使用类型名
    //将键值对记录，检测是否需要新建一个键值对
    if (strmap.find(val) == strmap.end()) {
        strmap.insert(pair<int, int>(val, 1));     未定义标识符 "pair"
    }
    else {
        strmap[val]++;
    }
    //将获得的结果放入新栈中，检测是否需要新建一个频率栈
    if (strmap[val] > maxfre) {
        maxfre = strmap[val];
        frestack.push_back(S);     未定义标识符 "frestack"
        frestack[maxfre - 1].push(val);
    }
    else {
        frestack[strmap[val] - 1].push(val);     未定义标识符 "frestack"
    }
}
```

2.pop 函数

```
int pop() {
    int ret = frestack[maxfre - 1].top();     未定义标识符 "frestack"
    frestack[maxfre - 1].pop();
    strmap[ret]--;
    if (frestack[maxfre - 1].empty()) {
        maxfre--;

        frestack.pop_back();
    }
    return ret;
}
```

2.6.5 调试分析（遇到的问题和解决方法）

2.6.6 总结和体会

在本题中，每次需要优先弹出频率最大的元素，如果频率最大元素有多个，则优先弹出靠近栈顶的元素。因此，我们可以考虑将栈序列分解为多个频率不同的栈序列，每个栈维护同一频率的元素。当元素入栈时频率增加，将元素加入到更高频率的栈中，低频率栈中的元素不需要出栈。

3. 实验总结

本次实验旨在比较顺序查找、二叉搜索树和哈希查找三种常见的查找算法的性能差异。通过设计相应的上机程序，对这些算法进行了实际测试和分析。

首先，实现了顺序查找算法。该算法简单直观，适用于小规模数据集。然而，在大规模数据集中，其时间复杂度较高，性能相对较差。这使得顺序查找在处理大型数据时可能不是最优选择。

其次，研究了二叉搜索树的性能。二叉搜索树通过有序排列的节点构建树结构，使得查找操作的平均时间复杂度为 O(log n)。但是，树的平衡性可能受到影响，导致性能下降。因此，在实际应用中，需要考虑平衡二叉搜索树的使用，如 AVL 树。

最后，探讨了哈希查找算法。哈希表通过哈希函数将关键字映射到数组索引，实现了 O(1) 的平均查找时间。然而，在处理冲突方面需要谨慎选择解决方法，以充分发挥哈希表的性能。实验中采用了开放地址法解决冲突，但也要注意其可能带来的性能问题。

综合比较，不同场景下选择不同的查找算法是至关重要的。顺序查找适用于小型数据，而二叉搜索树和哈希查找则更适合大规模数据集。在实际应用中，应该根据数据规模、插入删除频率等因素选择合适的查找算法，以取得更好的性能。