作业 HW5 实验报告

姓名：段威呈 学号：2252109 日期：2023 年 12 月 28 日

1. 涉及数据结构和相关背景

本次实验主要涉及针对不同数据结构类型的数据查找算法。

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))。

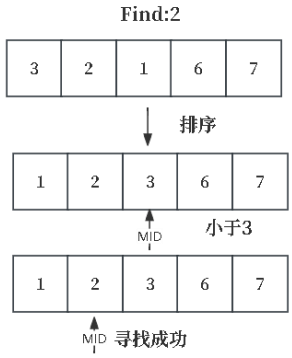

图 1.1 线性序列二分查找

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。

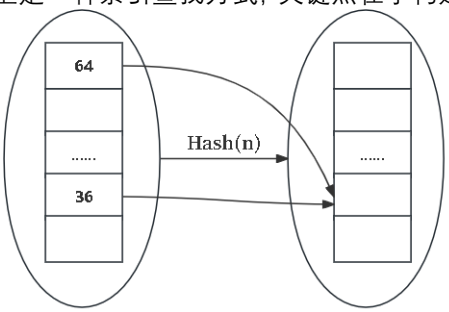

图 1.2 哈希查找与哈希冲突

哈希冲突的解决方式一般有两个，第一是通过线性探测散列实现，第二个是通过借助哈希链表+关键字查找实现。在本次实验中的第五题 哈希表 2 中即提供了一种平方探测的散列方法。

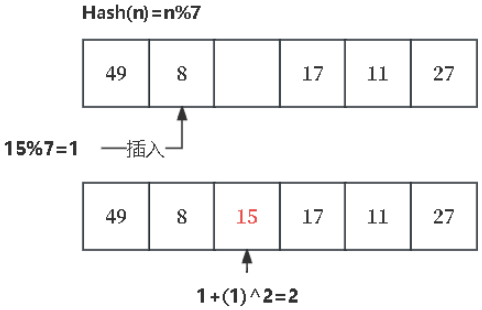

图 1.3 平方探测散列

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
输出
输出一行，包括 m 个整数，为 answer 中元素

2.1.3 算法设计

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。

前缀和序列的构建方式实际上就是用到了数列求和的公式:

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

因此这里不再开辟新的数列，直接在原顺序序列 queries 上操作，有递推通项:

$$quries[i]_{new} = queries[i]_{old} + quries[i-1]$$

从而得到前缀和序列，保存备用。

求解前缀和的算法时间复杂度为 ***O(n)***.

② 之后要根据 nums[i]的值从前缀和序列中寻找到：$\max\{j \mid queries[j] \le nums[i]\}$
即不超过 nums[i]的 queries[j]的最大值，返回 j+1 即为对应的最大和子序列长度。
如果逐个遍历对比，这样的方法时间复杂度是 O(n)。因此考虑使用二分法，则算法的时间复杂度为 O(logn)。
因此对 i 从 0 到 n，总时间复杂度为 ***O(nlogn)***。

因此总时间复杂度为 ***O(nlogn~ n^2 + n + nlogn)。***

2.1.4 算法实现

1.快速排序 QuickSort 实现

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：

```
void QuickSort(int low, int high) { //快排
    if (low >= high) {  //若待排序序列只有一个元素，返回空
        return;
    }
    int tmp;
    int i = low;    //i作为指针从左向右扫描
    int j = high;   //j作为指针从右向左扫描
    OrderElem key = (vex_order[low]);//第一个数作为基准数
    while (i < j) {
        while ((vex_order[j]).d >= key.d && i < j) {    //从右边找小于基准数的元素 （此处由于j值可能会变，所以仍需判断i是否小于j）
            j--;    //找不到则j减一
        }

        while ((vex_order[i]).d <= key.d && i < j) {    //从左边找大于基准数的元素
            i++;    //找不到则i加一
        }
        swap(i, j);
    }
    swap(low, i);//相遇时则交换
    QuickSort(low, i - 1);  //i左边的序列继续递归调用快排
    QuickSort(i + 1, high); //i右边的序列继续递归调用快排
}
```

2.前缀和求解的实现

根据前缀和公式，可以直接对 queries 数组进行原地求和操作：

```
void Sum_Calculate(int nums[NUMS_LEN]) {
    for (int i = 1; i < NUMS_LEN; i++) {
        nums[i] = nums[i - 1] + nums[i];
    }
}
```

3.二分查找实现

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：

$$j_{dest} = j \quad when: queries[j] \leq nums[i] < queries[j]$$

因此在最终判断条件终止的时候，要分别通过两侧数值的上下比较来判断是结束返回。也就是说，即便找到了比 nums[i]小的数值，也有可能不符合条件。如可能会有：

$$j_{dest} \neq j \quad when: queries[j] < queries[j] < nums[i]$$

因此二分搜索的代码应当如下编写：

```
//寻找不超过目标元素值的最大值的位置
void BinarySearch(int dst, int nums[NUMS_LEN], int low, int high) {

    if (low == high) {
        answer.push_back(low);
        return;
    }
    //条件成立的情况①
    if (nums[(low + high) / 2] > dst && nums[((low + high) / 2) - 1] <= dst) {
        answer.push_back(((low + high) / 2) - 1 + 1);
        return;
    }
    //条件成立的情况②
    if (nums[(low + high) / 2] <= dst && nums[((low + high) / 2) + 1] > dst) {
        answer.push_back(((low + high) / 2) + 1);
        return;
    }
    //往下找
    if (nums[(low + high) / 2] > dst && nums[((low + high) / 2) - 1] > dst) {
        BinarySearch(dst, nums, low, ((low + high) / 2) - 1);
    }
    //往上找
    if (nums[(low + high) / 2] < dst && nums[((low + high) / 2) + 1] < dst) {
        BinarySearch(dst, nums, ((low + high) / 2) + 1, high);
    }

}
```

2.1.5 调试分析（遇到的问题和解决方法）

本题在关于编写二分搜索结束判断条件时候出现了问题，即一开始没有考虑到如下情况：

$$\boldsymbol{j_{dest} \neq j} \quad when: queries[j] < queries[j] < nums[i]$$

因此在完成此类问题的时候，应当先从纸面上完成数学表达式的推演再编写代码。

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。另外也要着重感受在这里面前缀和方法的使用，大大降低了重复计算的时间成本，这是本题解决的关键。

2.2 题目二

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：

```
struct TreeNode {
    int data;
    TreeNode* lchild;
    TreeNode* rchild;
};
```

树根结点：

```
struct Tree {
    node root = NULL;
};
typedef Tree BiTree;
```

结点的创建与初始化：

```
node CreatNode(int e) {
    node S = (node)malloc(sizeof(TreeNode));
    S->data = e;
    S->lchild = NULL;
    S->rchild = NULL;
    return S;
}
```

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。

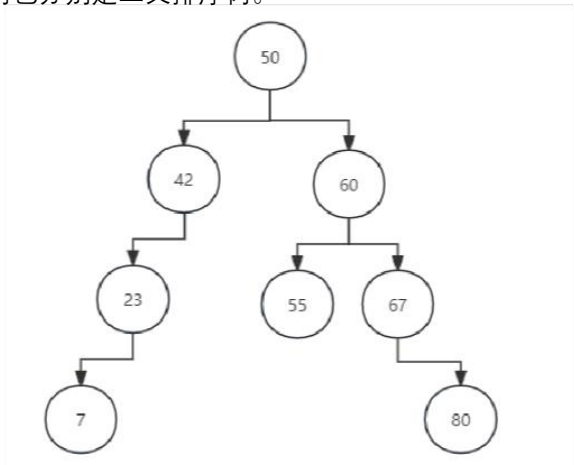

图 2.2.1 二叉排序树

2.2.4 功能实现

1.插入功能

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。

```
void Insert(BiTree& T, int e) {
    node S;
    S = CreatNode(e);
    node p = NULL, q = NULL;  //这里引入双指针，一个指向前驱一个指向后继
    if (!T.root) {
        T.root = S;
    }
    else {
        p = T.root;
        //寻找插入前驱
        while (p) {
            q = p;
            if (e <= p->data) {
                p = p->lchild;
            }
            else {
                p = p->rchild;
            }
        }
        if (e <= q->data) {
            q->lchild = S;
        }
        else {
            q->rchild = S;
        }
    }
}
```

发现这里把与父节点值相同的都连接到左孩子上了。

2.搜索功能

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。

```
int SearchAll(BiTree T, int e) {
    node p, q;
    BiTree tree;
    if (Search(T, e) == NULL) //不存在
        return 0;
    else {
        p = T.root;
        while (p != NULL && p->data >= 0) {
            q = p;
            if (q->data == e) {
                ++cnt;
                tree.root = q->lchild;
                SearchAll(tree, e);
                return 1;
            }
            if (e <= p->data)
                p = p->lchild;
            else
                p = p->rchild;
        }
    }
    return 0;
}
```

3.搜索点的前继

这里借助第一题中二分搜索前继值的思想，依旧是思考临界条件：

$$j_{dest} = j \quad when: tree[j] < dest < tree[j+1]$$

因此在判断是否满足条件的时候，应当考虑如下两种情况：

① 结点的值比查找的值大。先往右侧寻找大的值结点，如果此时寻找的结点结果都较大，则结果存在如下两种情况：

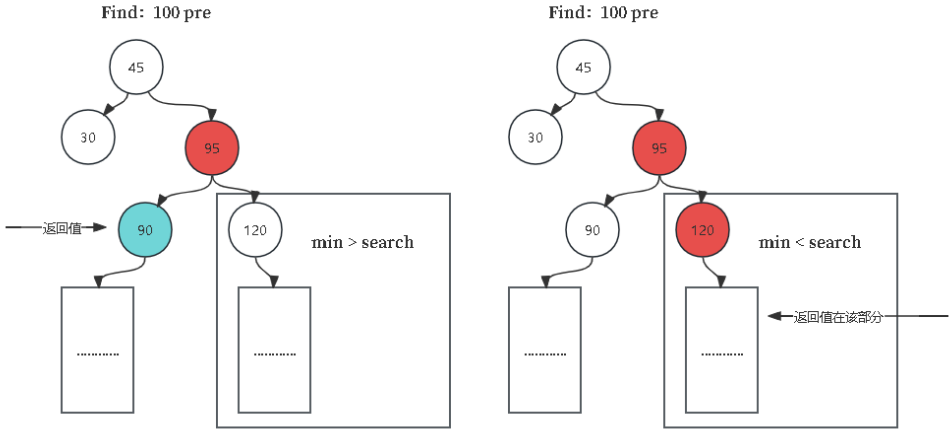

图 2.2.2 二叉排序树前继结点搜索(结点值大)

② 结点的值比查找的值小。则应当向右支寻找，如果一直寻找到最大也比待查找的值小，则直接返回最大值；若找到了更大值的结点，则同①情况更大值处理。

代码如下：

```
int SearchPre(BiTree& T, int e) {
    node p = NULL, q = NULL;
    if (!T.root) {
        return -1;
    }
    if (GetMin(T.root) >= e) {
        return -1;
    }
    else {
        p = T.root;
        while (p != NULL && p->data >= 0) {
            q = p;
            if (e <= p->data) {    //遇到有可能符合的部分
                p = p->lchild;
                if (p->data < e && (!p->rchild || GetMin(p->rchild) >= e))
                    return p->data;
            }
            else {                 //遇到不可能符合的部分
                p = p->rchild;
                if (!p)
                    return q->data;
                else
                    if (p->data >= e && (!p->lchild || GetMin(p) >= e))
                        return q->data;
            }
        }
    }
    return -1;
}
```

4.输出集合中的最小值

这里直接一直向左查找即可。

```
nt GetMin(node head) {
    node p = NULL, q = NULL;
    if (!head)
        return -1;
    else {
        p = head;
        q = p;
        while (p) {
            q = p;
            p = p->lchild;
        }
        return q->data;
    }
```

5.删除结点

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。

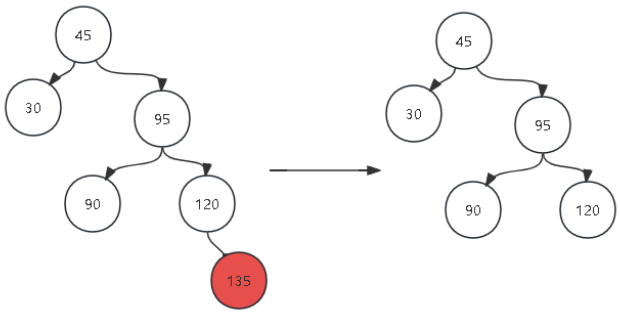

图 2.2.3 二叉排序树删除叶子结点

代码如下：

```
p = Search(T, e);
if (!p->lchild && !p->rchild) {  //是叶结点
    s = Search_Parent(T, e);
    if (s->rchild == p)
        s->rchild = NULL;
    if (s->lchild == p)
        s->lchild = NULL;
    free(p);
}
```

② 删除结点只有左/右支。

只需让后继结点继承即可。

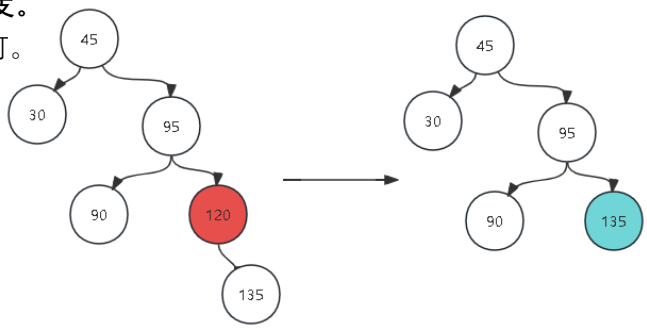

图 2.2.4 二叉排序树删除单支连接结点

代码如下：

```
else if (p->rchild&& !p->lchild) {  //没有左子树
    q = p->rchild;
    p->data = p->rchild->data;
    p->rchild = q->rchild;
    p->lchild = q->lchild;
    free(q);
}
```

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：

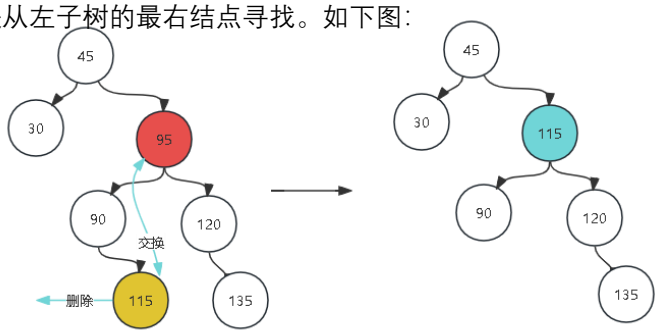

图 2.2.5 二叉排序树删除双连接结点

当然，在左子树不存在时候，可以考虑直接选取左节点（前继）作继承。如下图：

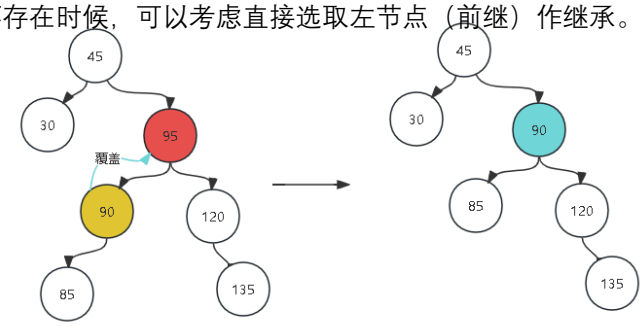

图 2.2.6 二叉排序树删除双连接结点2

代码如下：

```
else {              //同时具有左右子树
    q = p;
    s = p->lchild;
    //找到左子树最右侧元素
    while (s->rchild) {
        q = s;
        s = s->rchild;
    }
    p->data = s->data;//替换元素
    if (q != p) {       //左子树有右子树
        q->rchild = s->lchild;
    }
    else {              //左子树无右子树
        q->lchild = s->lchild;
        free(s);
        s = NULL;
    }
}
```

2.2.5 调试分析（遇到的问题和解决方法）

本题在调试的时候会出现删除的结点仍然存在指向的内容。尽管已经通过 free 的方式释放内存空间，但原本指向该内存空间的指针仍然右指向的内容。因此该过程需要自行手动指向 NULL 空间才能完全释放内存。否则在搜索过程中仍然会遍历已经删除的叶子结点的空间。

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；

此外，在前继结点搜素上，也用到了和二分前继查找相似的方法。同样需要依靠数学表达式找出搜索边界条件，并讨论在左右支可能出现的情况。

2.3 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

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

2.3.3 算法设计

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。

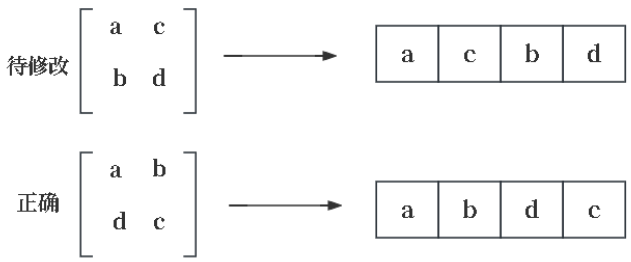

图 2.3.1 序列拉伸

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。
实现的方法是通过循环节调整，具体算法如下：

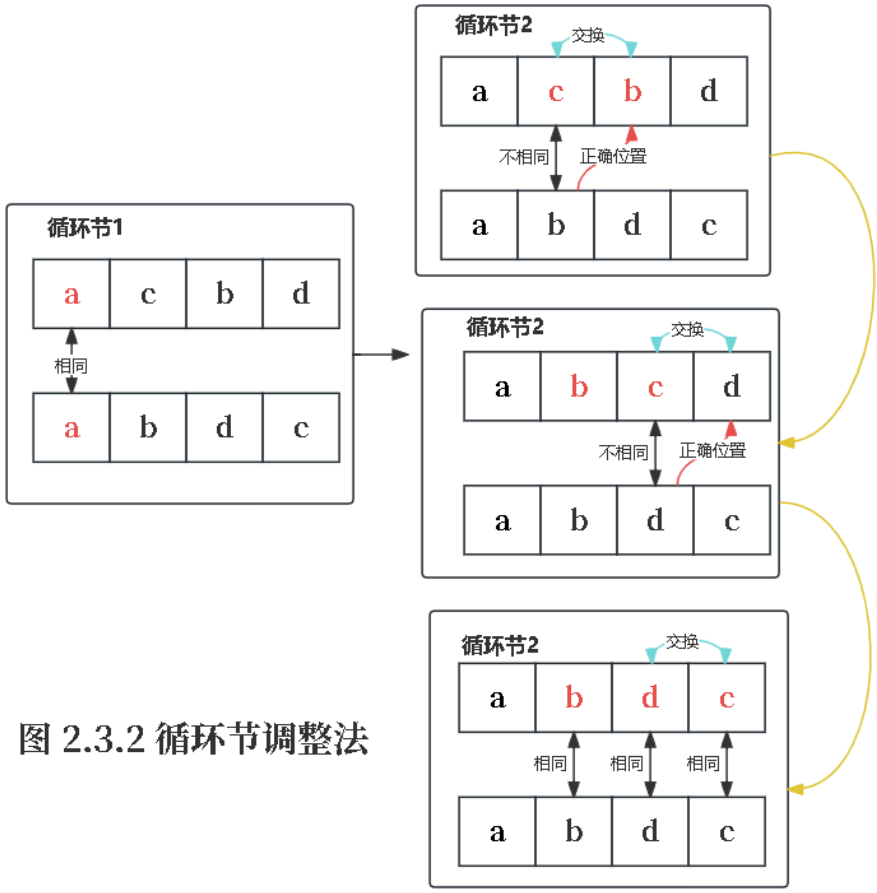

图 2.3.2 循环节调整法

这里发现，对于一个循环节，其中交换次数=循环节的元素个数-1。即：

$$Exchange(n_i)_i = n_i - 1$$

对所有的循环节中的调整交换次数求和：

$$Exchange(n) = \sum Exchange(n_i)_i = \sum n_i - \sum_1^m 1 = n - m$$

其中 m 为循环节的个数.

2.3.4 算法实现

① 序列拉伸

```
void Creatchart(int n, int m, std::vector<vector<std::string>> old_chart, std::vector<std::vector<std::string>> new_chart) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            Old.push_back(old_chart[i][j]);
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            New.push_back(new_chart[i][j]);
        }
    }
}
```

② 循环节查找

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：

```
map<string, int>oldmap;
map<string, int>newmap;
```

hashmap 的初始化：

```
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));
    }
    //创建new的hashmap
    for (int i = 0; i < New.size(); i++) {
        newmap.insert(pair<string,int>(New[i],i));
    }
}
```

并编写快速查找的函数：

```
int FindStrKey(int state, string str) {
    map<string, int>::iterator iter;
    if (state == 1) {
        return oldmap.find(str)->second;
    }
    if (state == 2) {
        return newmap.find(str)->second;
    }
}
```

之后可以依据循环节调整算法进行调整：

```
int Calculate_Circle() {
    int cnt=0;
    int j;   //标记指针
    string j_tmp;
    //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];
            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;
```

2.4 最大频率栈

2.4.1 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。

2.4.2 基本要求

输入
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
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

2.4.3 算法设计

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。

2.4.4 算法实现

push（int val）

```
void push(int val){
    stack<int> S;
    //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));
    }
    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);
        frestack[maxfre-1].push(val);
    }
    else {
        frestack[strmap[val] - 1].push(val);
    }
}
```

pop()

```
int pop() {
    int ret = frestack[maxfre-1].top();
    frestack[maxfre-1].pop();
    strmap[ret]--;
    if (frestack[maxfre-1].empty()) {
        maxfre--;

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

以上方法每次操作时间复杂度均为 O(1)

2.4.5 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。当元素出栈时，将频率最高的栈的栈顶元素出栈即可。要着重体会这种记录过程、分步维护的思路。

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。