作业 HW5 试验报告

姓名: 范潇　学号: 2254298

2023年 12月 21日

1　涉及数据结构和相关背景

查找，即在数据集合中寻找满足某种条件的数据对象。查找表则是由同一类型的数据元素组成的数据集合，可以按照查找过程中是否有元素发生变化而区分为静态查找表和动态查找表。关键字则是数据元素中的某个数据项的值，用于标记一个数据元素，主关键字可以唯一标识一个数据元素，而可以识别若干记录的关键字称为次关键字。

除了算法的时间复杂度和空间复杂性外，评价查找算法的一个标准是平均比较次数，也称平均查找长度ASL。

顺序查找，又称线性查找，适用于顺序表。实现需要注意的细节是：从顺序表结尾开始向前查找。将数组下标为0的位置置为要查找的元素。如果查找失败，对于长度为n的数组，需要比较n次。如果查找成功，则最少需要1次比较，最多需要n次比较。在查找成功的情况下，平均查找长度为(n+1)/2。

二分查找适用于已经排序好的顺序表。当所查找的顺序表按照升序排列时，先求位于查找区间正中的对象的下标mid，如果该位置是所给关键字，则查找成功。如果该关键字在欲查找的关键字的前面，则将查找区间的最小值设为mid+1，否则将最大值设为 mid-1。当low¿high时，查找失败。

分块查找适用于分块有序的顺序表，即块与块之间存在有序关系，而块内中不存在。在块与块之间使用二分查找，在块中使用顺序查找。

二叉排序树（二叉查找树）或者是一颗空树，或者是具有下列性质的二叉树：

1. 每个结点都有一个作为查找依据的关键字，所有结点的关键字互不相同。

2. 左子树（若非空）上所有结点的关键字都小于根节点的关键字

3. 右子树（若非空）上所有结点的关键字都大于根节点的关键字

4. 左子树和右子树也是二叉排列树

如果对一颗二叉排列树进行中序遍历，则可以按从小到大的顺序，将各结点关键字排列起来。因此可以使用中序遍历来求最大值、最小值，以及给定值的前驱和后继。

在查找中，和给定值比较的关键字个数等于待查找结点所在的层数。与给定值比较的关键字个数不超过树的高度，所以二叉排列树的性能和其深度相关。最好的情况是二叉排列树的形态和二分查找的判定树相同，其平均查找长度和 $\log_2 n$成正比。

对二叉排序树中的结点进行删除时，要求做到删除一个结点后，仍然保持二叉排序树的有序性。对于叶子结点，直接删除即可。对于有两个孩子结点的结点，若要删除它，则可以将它用左子树中的最大元素，或者右子树中的最小元素来代替。对于只有一个孩子结点的结点，要想删除它，可以直接用它的非空子树来代替它。

AVL树即高度平衡的二叉查找树。一颗AVL树或者是空树，或者是具有下列性质的二叉查找树：它的左子树和右子树都是AVL树，且左子树和右子树的高度之差的绝对值不超过1。任一结点的左子树高度减去右子树高度所得的值即为该结点的平衡因子，由AVL树的定义可知，只能取-1，0和1。AVL树还有如下性质：

1. 对于有n个结点的AVL树，它的高度保持在 $O(log_2 n)$，平均查找长度也保持在 $O(log_2 n)$

2. 高度为h的AVL树的最小结点数为$F_{h+3} - 1$

3. 适合于组织在内存中较小的索引

为了保持AVL树的平衡，当插入一个新结点时，插入后需要通过旋转来调整左右子树的高度。

哈希查找的基本思想为：在记录的存储地址和它的关键字之间建立一个确定的对应关系；使得不经过比较，依次存取就能得到所查元素的查找方法。优点为插入、查找的速度快”，缺点有：

1. 计算哈希地址需要一定的时间

2. 哈希表需要一定的冗余从而占用空间多

3. 只能按照关键字查找

相关的基本概念有：

1. 哈希函数：在记录的关键字与记录的存储地址之间建立的一种对应关系

2. 哈希表：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表

3. 哈希查找：利用哈希函数进行查找的过程

4. 冲突：不同关键词对应的哈希函数值相同的现象

5. 同义词：发生冲突的不同关键词

6. 负载因子：填入哈希表的结点数与哈希表空间的比值

哈希表的设计包括三个步骤：

1. 确定散列函数的值域，从而确定散列表的空间范围

2. 构造合适的散列函数，使得冲突发生概率尽可能小

3. 处理冲突的方法

哈希函数的评价因素有：

1. 是否简单

2. 是否均匀，从而冲突发生的概率小

常用的哈希函数构造法有：

1. 直接定址法

    - 取‘key‘的线性函数
    - 地址集合与关键字集合大小相等

2. 数字分析法

    - 取关键字若干位或其组合作为哈希地址

- 适用于关键字位数比哈希地址大，且可能出现的关键字事先知道的情况

3. 平方取中法

- 将关键字平方后取中间几位作为哈希地址
- 所取位数由散列表的长度决定
- 适用于不知道全部关键字的情况
- 较为常用

4. 折叠法

- 将关键字分割为位数相同的几部分，然后取这几部分的叠加
- 位移叠加：顺序不变
- 间界叠加：蛇形排序
- 适用于关键字位数很多，且每一位上数字分布大致均匀的情况

5. 除留余数法

- 取关键字除某个数p得到的余数作为哈希地址
- p不大于表长，常取素数或素数之积，来减少冲突发生的可能性

6. 乘余取整法

- 先让关键字乘以一个0到1之间的常数，然后取乘积的小数部分；然后再用散列表长度乘以该小数部分并向下取整。

7. 基数转换法

- 将原先 A进制的码视为B进制的码，然后转换为A进制的码，再取其中的若干位。
- B一般大于A，且互素

选取哈希函数主要考虑一下因素：

1. 计算哈希函数所需时间

2. 关键字长度

3. 哈希表长度

4. 关键字分布情况

5. 记录的查找频率

冲突处理的常用方法有：

1. 开放定址法

   - 发生冲突时，按照某个探测序列逐个探测散列表中的其他地址，直到遇到空地址为止，将发生冲突的记录存放在该地址中
   - 常用探测序列有：
     (a) 线性循环
       - 地址递增，将散列表视为是循环的来计算
       - 只要散列表未满，总能找到一个不冲突的地址
       - 会导致更多的冲突可能，即导致“集聚”现象

     (b) 二次探测
       - 增量序列为$1^2, -1^2, 2^2, -2^2, \cdots$
       - 不易产生“聚集”现象
       - 不能保证探测到散列表中的所有地址

     (c) 伪随机探测法
       - 增量序列由伪随机函数生成

2. 再哈希法

   - 构造多个哈希函数，依次使用，知道不发生冲突为止
   - 不易产生“聚集”现象
   - 计算时间增加

3. 链地址法

   - 哈希表中存储的是链表的头指针
   - 不易产生“聚集”
   - 删除记录简单

4. 建立公共溢出区

- 在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录

2 实验内容

2.1 和有限的最长子序列

2.1.1 问题描述

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。

2.1.2 基本要求

输入要求： 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。

输出要求： 输出一行，包括m个整数，为answer中元素。

2.1.3 数据结构设计

```
typedef int ElemType;
vector<ElemType> data,query,ans;//分别存储整数数组、询问、以及答案
```

2.1.4 功能说明

Algorithm 1 求和有限的最长子序列

输入: 最大和序列query、整数数组data

输出: 最长子序列长度数组ans

1: **function** GET_MAX_LENGTH(vector<int>query,vector<int>data)
2: **while** 遍历data **do**
3: data[i]+=data[i-1]//求前缀和

```
    end while
    Sort(data)//升序排列
    l = 0
    h = data.size()-1
    while 遍历data do
        while l≤h do
            mid = (l+h)/2
            if data[mid]≤query[i]&&((mid==data.size()-1)
     || (data[mid+1]>query[i])) then
                ans.push_back(mid+1)
                break
            end if
            if data[mid]<query[i] then
                l=mid+1
            elseh=mid-1
            end if
        end while
    end while
    return ans
end function
```

2.1.5 调试分析

本题调试过程较短，使用输出中间变量法来辅助调试。所输出的中间变量是搜索区间端点 l和 h。

2.1.6 总结与体会

该题题目中强调“不改变元素顺序”，但是由于子序列并不要求连续，所以实际上该题中元素的顺序并不会影响结果。该题的易错点之一是本题中 mid上的元素符合要求的标准是“小于给定值的最大值”，而非“与给定值相等”，所以成立的条件是 data[mid]≤query[i]&&((mid==data.size()-1)||(data[mid+1]>query[i])).

2.2 二叉排序树

2.2.1 问题描述

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：

1. 若op=1，后面跟着一个整数x，表示插入数字x
2. 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）
3. 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）
4. 若op=4，输出集合中的最小值（保证集合非空）
5. 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）

输出要求： 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

```
typedef int ElemType;
typedef struct BST {
    ElemType key;
    int n;
    BST* lchild, * rchild;
}BST;
```

2.2.4 功能说明

Algorithm 2 插入元素

输入: 二叉排序树，待插入元素

输出: 插入元素后的二叉排序树

```
1:  function INSERT(BST*&T,ElemType x)
2:      if T非空 then
3:          为T申请内存空间
4:          左右子树置空
5:          key=x
6:          n=1
7:          return
8:      end if
9:      if x==key then
10:         n++
11:         return
12:     end if
13:     p = T
14:     pre = NULL
15:     while p&& p− >key!=x do
16:         pre更新为p
17:         if p− >key<x then
18:             p更新为右子树
19:         else
20:             p更新为左子树
21:         end if
22:     end while
23:     if p非空 then
```

```
        n++
    else
        为p申请内存空间
        左右子树置空
        key=x
        n=1
        return
    end if
end function
```

Algorithm 3 求指定元素个数

输入: 二叉排序树，指定元素

输出: 指定元素个数

```
function GET_NUM(BST*&T,ElemType x)
    if T为空 then
        输出0
        return
    end if
    p = T
    while p&& p− >key!=x do
        pre更新为p
        if p− >key<x then
            p更新为右子树
        else
            p更新为左子树
        end if
    end while
    if 如果p为空 then
        输出0
    else
        输出n
    end if
end function
```

Algorithm 4 求最小元素

输入: 二叉排序树
输出: 二叉排序树中的最小元素

```
1: function GET_MIN(BST*T)
2:     if T为空 then
3:         输出-1
4:         return
5:     end if
6:     p = T
7:     while 左子树不为空 do
8:         p更新为左子树
9:     end while
10:    输出key
11: end function
```

```cpp
//求指定元素的前驱
void pre(BST* T, ElemType x)
{
    BST* target = NULL;//用于存储前驱所在的结点
    get_pre(T, target, x);
    if (!target)//并未找到前驱
        cout << "None" << endl;
    else//找到前驱
        cout << target->key << endl;
    return;
}
void get_pre(BST* T, BST*& target, ElemType x)
{
    if (!T)//树为空
        return;
    //中序遍历
    get_pre(T->lchild, target, x);
    if (T->key < x)//保持始终为的前驱targetT
        target = T;
    get_pre(T->rchild, target, x);
    return;
}
```

Algorithm 5 删除指定元素

输入: 当前结点，指定元素，当前结点的前驱，flag
输出: 更新后的结点

```
1: function DEL(BST*&T,ELemType x,BST*&pre,int remove)
2:     if T为空 then
3:         输出None
4:         return
5:     end if
6:     p = T
7:     while p&& p− >key!=x do//寻找待删除元素
8:         pre更新为p
9:         if p− >key<x then
10:            p更新为右子树
11:        else
12:            p更新为左子树
13:        end if
14:    end while
15:    if p非空 then
16:        输出None
17:        return
18:    end if
19:    if n>1&&!remove then//找到但是有多个，且不移除
20:        n-=1
21:        return
22:    end if
23:    if p的左右孩子均为空 then
24:        if 前驱不为空，且p为前驱的右子树 then
25:            前驱右子树置空
26:        end if
27:        if 前驱不为空，且p为前驱的左子树 then
28:            前驱左子树置空
29:        end if
30:        释放p
31:    end if
32:    if p没有左孩子 then
```

```
33:         if 前驱为空 then
34:             T置为p的右子树
35:         end if
36:         if 前驱不为空，且p为前驱的右子树 then
37:             前驱右子树置为p的右子树
38:         end if
39:         if 前驱不为空，且p为前驱的左子树 then
40:             前驱左子树置为p的左子树
41:         end if
42:         释放p
43:         return
44:     end if
45:     if p没有右孩子 then
46:         if 前驱为空 then
47:             T置为p的左子树
48:         end if
49:         if 前驱不为空，且p为前驱的左子树 then
50:             前驱左子树置为p的左子树
51:         end if
52:         if 前驱不为空，且p为前驱的右子树 then
53:             前驱右子树置为p的左子树
54:         end if
55:         释放p
56:         return
57:     end if
58:     辅助指针q置为p的左子树
59:     pre置为p
60:     while q的右子树不为空 do
61:         pre置为q
62:         q置为q的右子树
63:     end while
64:     p− >key=q− >key
65:     p− >n=q− >n
66:     del(q,q− >key,pre,1)//此时不再是删除，而是直接移动
```

67: **end function**

2.2.5 调试分析

在基本错误全部解决后，可以通过前9个样例，但是第10个样例无法通过。因为第10个样例的输入过多，难以找出出错原因，所以我编写脚本循环使用生成测试样例的数据。发现当输入数据达到50多条时才会出现错误。经过排查，发现是因为我将“移动”一个结点的操作等同于删除一个结点了，当排序树中的元素没有多份时，这样是可以的，但是在本题中不行，所以我添加了remove形参来区分“移动”结点还是“删除”结点。

2.2.6 总结与体会

本题和普通的排序树的最大差别在于树中的元素可以有多份，当删除一个元素时，实际上是减去该元素的一份，当全部份数都被减去后，才释放该结点。但是当对排序树进行调整时，是要将元素的作为一个整体进行移动，要把待移动的元素的所有份数一起移动，因此不能用“删除”这个操作实现，这是和普通排序树有所不同，也是本题的易错点。

在设计数据结构时，需要注意各个数据结构的特点。在本题中，便利用了“中序遍历二叉排序树得到的是递增序列”这一特性来实现“查询最小值”，“查询某个数字的前驱”这些功能。

2.3 换座位

2.3.1 问题描述

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。

输出要求： 一个整数，表示最少交换次数。

2.3.3 数据结构设计

```
1  std::vector<vector<std::string>> &old_chart;
2  std::vector<std::vector<std::string>> &new_chart;
3  map<string, bool> flag;//旧座位表中的元素到其被访问状态之间的的哈希表
4   map<string, string>old_map;//旧座位表中的元素到新座位表中对应位置的元素之
       间的哈希表
5   map<string, string>new_map;//新座位表中的元素到旧座位表中对应位置的元素之
       间的哈希表
```

2.3.4 功能说明

Algorithm 6 求最少交换次数

输入: 旧座位表old_chart,新座位表new_chart

输出: 最少交换次数sum

1: **function** SOLUTION(std::vector<vector<std::string>> &old_chart, std::vector<std::vector<std::string>> &new_chart)
2: 初始化并置零计数器sum
3: **while** 遍历old_chart **do**
4: 构建old_chart中的元素到new_chart中对应位置的元素之间的哈希表old_map
5: 构建new_chart中的元素到old_chart中对应位置的元素之间的哈希表new_map
6: 构建old_chart中的元素到其被访问状态之间的的哈希表flag
7: **end while**
8: **while** 遍历old_chart **do**
9: **if** old_chart[i][j] == new_chart[i][j] **then**
10: flag[old_chart[i][j]] = 1
11: continue
12: **end if**
13: **if** 当前循环元素old_chart[i][j]已被访问过 **then**
14: continue
15: **end if**
16: string src = old_chart[i][j], now =old_map[src]

```
17:         更新src,now中的元素的被访问状态
18:         while now!=src do
19:             sum++
20:             now= old_map[now]
21:             更新now的被访问状态
22:         end while
23:     end while
24:     return sum
25: end function
```

2.3.5 调试分析

一开始使用顺序查找表实现，但是会超时，后来改为了用stl容器map来实现，便能顺利通过样例。

2.3.6 总结与体会

在进行大规模的查找时，数据结构的选取便显得十分重要。本题采用了哈希表，降低了查找所需的时间复杂度。

2.4 哈希表

2.4.1 问题描述

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。

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）

2.4.2 基本要求

输入要求： 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。

输出要求： 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'-'。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 数据结构设计

```
1 bool* vec;//用于记录哈希表中位置的使用情况
```

2.4.4 功能说明

Algorithm 7 判断是否为质数

输入: 待判断正整数n

输出: 判断结果

```
1: function IS_PRIME(unsigned long long n)
2:     if n<2 then
3:         return 否
4:     end if
5:     while 遍历大于2，且比n小的正整数 do
6:         if 循环变量i能整除n then
7:             return 否
8:         end if
9:     end while
10:    return 是
11: end function
```

```
1 //获得输入字符串的键值
2 unsigned long long get_key(string str)
3 {
```

```
    unsigned long long key=0;
    for(int i = 0;i<str.length();i++){
        key*=37;
        key+=(str[i]);
    }
    return key;
}
```

```
//二次探测
void detect(bool*vec,unsigned long long key,unsigned long long len)
{
    int i = 0;//记录探测次数
    int d = 2;
    while(i<len){
        if(d%2){//偏移量的符号按照奇偶次序交换
            if(!vec[(key-(d/2)*(d/2)+len*len)%len]){
                //位置未被占用
                vec[(key-(d/2)*(d/2)+len*len)%len]=1;
                cout<<(key-(d/2)*(d/2)+len*len)%len<<"␣";
                //存储成功
                return;
            }
        }
        else{
            if(!vec[(key+(d/2)*(d/2))%len]){
                //位置未被占用
                vec[(key+(d/2)*(d/2))%len]=1;
                cout<<(key+(d/2)*(d/2))%len<<"␣";
                //存储成功
                return;
            }
        }
        d++;
        i++;
    }
    //存储失败
    cout<<"-"<<"␣";
    return ;
}
```

Algorithm 8 创建哈希表

输入: 关键字总数N，散列表长度P

输出: 存储结果序列

```
1: function INIT(int N,unsigned long long P)
2:     while !is_prime(P) do
3:         P++
4:     end while
5:     初始化辅助数组vec并置零
6:     while 读取关键字数量少于N do
7:         读入关键字并存储至变量in中
8:         key = get_key(in)
9:         key%=p
10:        if !vec[key] then
11:            vec[key]=1
12:            输出key
13:        else
14:            detect(vec,key,p)
15:        end if
16:    end while
17: end function
```

2.4.5 调试分析

一开始我所编写的程序无法通过最后几个样例。经过分析，发现是因为哈希表长度过大，在解决冲突时，会产生对负数取余数的操作，无法得到正确结果，因此我在模len的操作之前，先加上len*len。最终通过全部样例。

2.4.6 总结与体会

本题的易错点在于，在通过模p得到key的过程中，要确保被模p的数字非负。同时，从本题也可以看出，哈希表在时间复杂度上的优势在一定程度上是用空间复杂度换取的。

3 实验总结

查找表以多种存储结构为基础，并利用了它们的特点。例如二分查找、哈希表利用了顺序表的随机存取这一特点，而AVL树中的旋转操作也利用了二叉链表的灵活性。在实际应用中，首先要根据所给条件确定大致的查找表类型，然后在此基础上进一步进行设计和优化，例如在“二叉排序树”这一道题中，便在结点中增加了用于记录重复个数的数据域；在使用哈希表时更是如此，如果没有选择合适的哈希函数或者解决冲突的方案，则会大大降低查找的效率。