组号: 13

上海大学计算机工程与科学学院

实 验 报 告

（数据结构 2）

学　　期：2023-2024 年春季

组　　长：高焕景

学　　号：22120650

指导教师：朱能军

成绩评定：　　（教师填写）

二〇二 X 年 X 月 X 日

小组信息				
登记序号	姓名	学号	贡献比	签名
1	孔馨怡	22122128	33%	
2	徐亚妮	22122075	33%	
3	高焕景	22120650	33%	

实验概述	
实验 0	（熟悉上机环境、进度安排、评分制度；确定小组成员）
实验一	有向网的邻接矩阵验证及拓展
实验二	无向网的邻接表验证和拓展
实验三	查找算法验证及设计
实验四	(*实验题目*)

实验三

一、实验题目

查找算法验证及设计

二、实验内容

（1）查找 3 个数组的最小共同元素。

有 3 个整数数组 a[]、b[]和 c[], 各有 aNum、bNum 和 cNum 个元素（aNum, bNum, cNum <= n），而且三者都已 经从小到大排列。设计并编写算法找出最小共同元素以及该元素在 3 个数组中出现的位置，若没有共同元素，则显示“NOT FOUND”，要求算法在最坏情况下的时间复杂度为 O(n)。

【输入】

第一行三个数，表示三个数组元素个数，接下来三行是各有序数组。

【输出】

若存在，则输出四个数，分别表示共同元素以及在 3 个数组中出现的位置；否则输出“NOT FOUND”。

【输入样例】

2 4 5

3 4

4 7 8 9

1 2 3 4 5

【输出样例】

4 2 1 4

（2）求两个有序序列的中位数。**（在原有题目上做出改进）**

有两个长度为 m, n 的有序序列，如果将这两个序列合并成一个有序序列，则处于中间位置的元素称为这两个序列的中位数。请设计 2 种求两个有序序列的中位数的算法，要求其中一种算法在最坏情况下的时间复杂度为 O(logn)。

【输入】

第一行两个正整数 m，n，表示两个序列的长度，接下来两行是各有 m，n 个元素的有序序列。

【输出】

一个数，就是所求中位数的值。

【输入样例】

8 8

1 3 5 7 9 11 13 15

12 14 16 18 20 22 24 26

【输出样例】

13

（3）二叉排序树的验证和拓展

对于在二叉排序树上删除结点的问题，教材中介绍了 4 种算法，并实现了其中第 1 种 算法，现要求完成另 1 种：即移动当前节点左子树到其后继左子树的方式，并用多组测试数据对教材实现的和本题实现的这 2 种算法进行性能测试，分析比较它们的查找性能。

三、解决方案

1、 算法设计（*主要描述数据结构、算法思想、主要操作、用例分析、改进方法等*）

（1）查找 3 个数组的最小共同元素。

- 数据结构和算法思想：

哈希表（Hash Table）：

使用了 map<int, struct location> 类型的哈希表，其中键为整数，表示数组中出现的元素本身（数值），值为包含两个整数的结构体 location，用来存储元素及其在两个输入数组 a，b 中的位置。

遍历：

通过遍历第三个输入数组，在哈希表中查找元素，如果找到对应的位置信息，则表示该元素在前两个数组中都存在，并输出其位置信息。

- 主要操作：

建立结构体 location：

用来在哈希表的值中存储数值在多个数组中的位置。

建立哈希表：

读取输入数组 a 和 b，将元素及其位置存入哈希表中。

读取输入数组 c。

遍历数组 c，在哈希表中查找元素，输出对应的位置信息或者 "NOT FOUND"。

- 改进方法：

就数据结构（类）进行了探讨，思考 hash 表在值的存储（数组位置存储）时，到底该使用 map 还是 multimap，在比较二者后选择了更好的类。

二者异同在于：

•map：一个键一个值，multimap：一个键多个值，他们在“键”上是一样的，在存储，“查找”数值的过程是一样的

•值的不同：map 通过 struct 来存在数组中出现的位置，可以通过 hash[数值].a 的方式直接可以得到是否在 a 数组中存在该数字，但是 multimap 中没有直接可以得到值个数的函数，必须再自己遍历一遍才可以知道此时存了几个值

综上所述，我们最后选择了 map 作为存储容器。

（2）求两个有序序列的中位数

- 数据结构和算法思想：

数据结构：

使用了两个 vector<int> 类型的数组 nums1 和 nums2，分别表示两个已排序的数组。

算法思想：

采用了中位数的概念，并通过二分查找的方法在两个数组中找到合适的切割点，使得左半部分的元素都小于右半部分的元素。最终根据切割点处的元素确定中位数的值。

- 主要操作：

判断数组大小：

确保 nums1 的长度小于等于 nums2 的长度。一是避免数组越界，二是可以

缩小范围，查找更快。

二分查找、计算左右边界值、比较对角线上的关系的大小：

通过二分查找确定 nums1 中切割点的位置 cut1，进而由中位数左右两边确定的个数计算出 nums2 中切割点的位置 cut2。

根据切割点的位置计算出左半部分的最大值和右半部分的最小值。并在切割点位置在边界时对边界进行处理，即补足够大或者足够小的数字来保证后续比较大小的操作可以顺利完成，不会越界且不会出现错误或异常。

根据左右边界值的大小关系，调整二分查找的范围。在此过程中体现二分的思想，每次根据对角线上关系的大小可以知道我们此时是需要往左移还是往右移，然后更新范围，达到减小一半范围的效果。

计算中位数：

根据切割点及左右边界值，再结合两个数组的个数总和，确定中位数到底是中间的一个数还是中间两个数的平均值，计算出中位数的值。

（3）二叉排序树的验证和拓展

重现书上方法，算法思想略。

2、 源程序代码（*要求有必要注释、格式整齐、命名规范，利于阅读*）

（1） 查找 3 个数组的最小共同元素。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;
struct location // 用来存元素位置
{
    int a;
    int b;
};
int main()
{
    int count_a, count_b, count_c; // 每个数组的个数
    vector<int> c;              // c 数组
    int temp;
    map<int, struct location> hash; // hash 表 键：元素 值：位置
    cin >> count_a >> count_b >> count_c;
```

```
    c.resize(count_c);              // 为c 数组调整大小
    for (int i = 0; i < count_a; i++) // a 数组
    {
        cin >> temp;
        hash[temp].a = i + 1; // 将位置存入hash 表
    }
    for (int i = 0; i < count_b; i++) // b 数组
    {
        cin >> temp;
        hash[temp].b = i + 1; // 将位置存入hash 表
    }

    for (int i = 0; i < count_c; i++) // 存c 数组
    {
        cin >> c[i];
    }

    for (int i = 0; i < count_c; i++) // 遍历c 数组
    {
        if (hash[c[i]].a && hash[c[i]].b) // 如果此时的元素在hash 表中a，b 都存过
        {
            cout << c[i] << " " << hash[c[i]].a << " " << hash[c[i]].b << " "
<< i + 1 << endl;
            break; // 此时找到共同最小元素 跳出循环
        }
        if (i == count_c - 1) // 遍历到最后，说明没找到 输出NOT FOUND
            cout << "NOT FOUND" << endl;
    }
    return 0;
}
```

（2） 求两个有序序列的中位数。

算法一（不是主要算法，时间复杂度 O（n））：

```
int main()
{
    vector<int> nums1, nums2;
    int size;
    cin >> size;
    nums1.resize(size), nums2.resize(size); // 调整大小
    for (int i = 0; i < size; i++)
    {
        cin >> nums1[i]; // 存数组
    }
```

```
    for (int i = 0; i < size; i++)
    {
        cin >> nums2[i]; // 存数组
    }

    int mid = size;           // 表示两个数组的中间位置（中位数的位置）
    int i = 0, j = 0;         // 标记nums1,nums2 移动的位置
    int left = -1, right = -1; // mid/中位数的左右到数字
    for (int k = 0; k <= mid; k++)
    {
        left = right; // 更新left 为上一次移动后的数组
        // 开始下一次移动
        if (i < size && (j >= size || nums1[i] < nums2[j]))
        // 如果nums1 没移动完 ， 并且nums2 移动完了，或者nums1 此时更小
        {
            right = nums1[i]; // 这一次的往后移动一个数 存的是nums1 的
            i++;            // 移动nums1
        }
        else
        {
            right = nums2[j];
            j++;
        }
    }

    if (mid % 2 == 0)
    {
        cout << (int)((left + right) / 2.0);
        // 如果是偶数 中位数是两边数的平均
    }
    else
    {
        cout << right;
    }
    return 0;
}
```

算法二：二分法（上述算法思想描述对象）

```
class Solution
{
public:
    double findMedianSortedArrays(vector<int> &nums1, vector<int> &nums2)
    {
```

```
        int size1 = nums1.size();
        int size2 = nums2.size();
        if (size1 > size2) // 保证一数组的个数一定小于二数组
            return findMedianSortedArrays(nums2, nums1);
        if (!size1) // 空数组的情况直接处理
        {
            return (size2 % 2)
                     ? nums2[size2 / 2]
                     : (nums2[size2 / 2 - 1] + nums2[size2 / 2]) / 2.0;
        }
        int lmax1, lmax2, rmin1, rmin2; // 左边最大值 右边最小值
        int c1, c2;                  // 切割的位置
        int left = 0, right = size1;    // 表示当前c1存在的区间
        while (left <= right)          // 范围没有缩小到一个位置的时候继续循环
        {
            c1 = (left + right) / 2;              // c1是这个范围二分的结果
            c2 = (size1 + size2) / 2 - c1;          // 算出c2的位置
            lmax1 = (!c1) ? INT_MIN : nums1[c1 - 1]; // 对边界进行处理
            rmin1 = (c1 == size1) ? INT_MAX : nums1[c1];
            lmax2 = (!c2) ? INT_MIN : nums2[c2 - 1];
            rmin2 = (c2 == size2) ? INT_MAX : nums2[c2];
            if (lmax1 > rmin2)  // 开始比较对角线上的关系
                right = c1 - 1; // 如果第一个数组左边太大了，说明此时中位数比c1要往左移
            else if (lmax2 > rmin1)
                left = c1 + 1; // 如果第二个数组的左边太大了，c2需要往左走，那么c1往右
走
            else             // 找到中位数了
                break;
        }
        double ou = (max(lmax1, lmax2) + min(rmin1, rmin2)) / 2.0;
        double ji = min(rmin1, rmin2);
        return (size1 + size2) % 2 ? ji : ou;
    }
};
```

（3）二叉排序树的验证和拓展

教材第四种方法：

```
template <class ElemType>
void BinarySortTree<ElemType>::Delete(BinTreeNode<ElemType> *&p)
// 操作结果：删除p指向的结点------第四种方法
{
    BinTreeNode<ElemType> *tmpPtr, *tmpF;
    if (p->leftChild == NULL && p->rightChild == NULL)  {   // p为叶结点
```

```
        delete p;
        p = NULL;
    }
    else if (p->leftChild == NULL)  {   // p 只有左子树为空
        tmpPtr = p;
        p = p->rightChild;
        delete tmpPtr;
    }
    else if (p->rightChild == NULL) {   // p 只有右子树为空
        tmpPtr = p;
        p = p->leftChild;
        delete tmpPtr;
    }
    else    {   // p 左右子非空
        tmpF = p;
        tmpPtr = p->rightChild;
        while (tmpPtr->leftChild != NULL)   {   // 查找p 在中序序列中直接后继 tmpPtr
及其双亲 tmpF,直到 tmpPtr 左子树为空
            tmpF = tmpPtr;
            tmpPtr = tmpPtr->leftChild;
        }
        tmpPtr->leftChild=p->leftChild;
        tmpPtr=p;
        p = p->rightChild;
        delete tmpPtr;
    }
}
```

教材第二种方法：

```
template <class ElemType>
void BinarySortTree<ElemType>::Delete(BinTreeNode<ElemType> *&p)
// 操作结果: 删除 p 指向的结点------第二种方法
{
    BinTreeNode<ElemType> *tmpPtr, *tmpF;
    if (p->leftChild == NULL && p->rightChild == NULL)  {   // p 为叶结点
        delete p;
        p = NULL;
    }
    else if (p->leftChild == NULL)  {   // p 只有左子树为空
        tmpPtr = p;
        p = p->rightChild;
        delete tmpPtr;
    }
```

```
    else if (p->rightChild == NULL) {   // p 只有右子树为空
        tmpPtr = p;
        p = p->leftChild;
        delete tmpPtr;
    }
    else    {   // p 左右子非空
        tmpF = p;
        tmpPtr = p->rightChild;
        while (tmpPtr->leftChild != NULL)   {   // 查找 p 在中序序列中直接后继 tmpPtr
及其双亲 tmpF,直到 tmpPtr 左子树为空
            tmpF = tmpPtr;
            tmpPtr = tmpPtr->leftChild;
        }
        p->data = tmpPtr->data;
            // 将 tmpPtr 指向结点的数据元素值赋值给 tmpF 指向结点的数据元素值
        // 删除 tmpPtr 指向的结点
        if (tmpF->rightChild == tmpPtr) // 删除 tmpF 的右孩子
            Delete(tmpF->rightChild);
        else    // 删除 tmpF 的左孩子
            Delete(tmpF->leftChild);
    }
}
```

教材第三种方法：

```
template <class ElemType>
void BinarySortTree<ElemType>::Delete(BinTreeNode<ElemType> *&p)
// 操作结果: 删除 p 指向的结点------第三种方法
{
    BinTreeNode<ElemType> *tmpPtr, *tmpF;
    if (p->leftChild == NULL && p->rightChild == NULL)  {   // p 为叶结点
        delete p;
        p = NULL;
    }
    else if (p->leftChild == NULL)  {   // p 只有左子树为空
        tmpPtr = p;
        p = p->rightChild;
        delete tmpPtr;
    }
    else if (p->rightChild == NULL) {   // p 只有右子树为空
        tmpPtr = p;
        p = p->leftChild;
        delete tmpPtr;
    }
```

```
    else    {   // p 左右子非空
        tmpF = p;
        tmpPtr = p->leftChild;
        while (tmpPtr->rightChild != NULL)  {   // 查找 p 在中序序列中直接前驱
tmpPtr 及其双亲 tmpF,直到 tmpPtr 右子树为空
            tmpF = tmpPtr;
            tmpPtr = tmpPtr->rightChild;
        }
        tmpPtr->rightChild=p->rightChild;
        tmpPtr=p;
        p = p->leftChild;
        delete tmpPtr;
    }
}
```

3、 实验结果（*展示实验结果、测试情况、结果分析等*）

（1） 查找 3 个数组的最小共同元素。

```
2 4 5
3 4
4 7 8 9
1 2 3 4 5
4 2 1 4
```

（2） 求两个有序序列的中位数。

Leetcode 上有此题，该代码全部测试案例运行通过。

通过 执行用时: 3 ms

• Case 1 • Case 2 • Case 3 • Case 4 • Case 5

• Case 6

输入

nums1 =

[1]

nums2 =

[1,2,3,4,5,6]

输出

3.00000

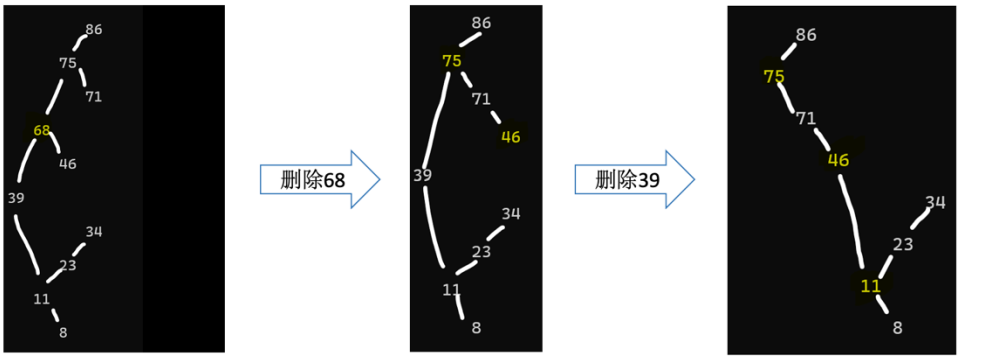
预期结果

3.00000

（3）二叉排序树的验证和拓展

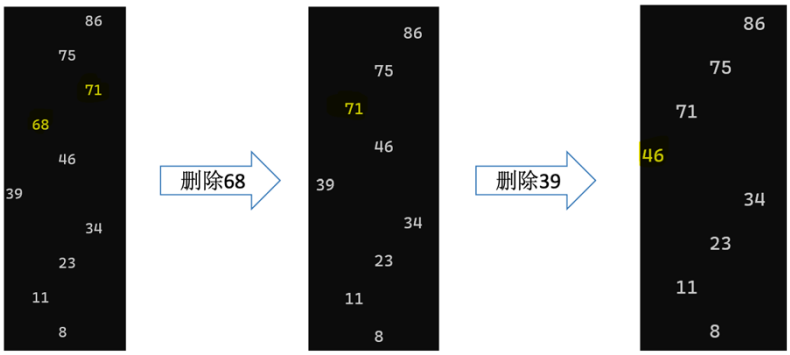
案例测试-方法4

先把它的左子树作为右子树中关键字值最小的数据元素x的左子树，然后再删除结点。



案例测试-方法2

在它的右子树中寻找关键字值最小的数据元素(其右子树中序遍历中第一个被访问的数据元素)x，用x的值代替被删除数据元素的值，再来删除数据元素x (x没有左子树)。



4、 算法分析（*对算法空间、时间效率进行必要分析，可能的改进建议等*）

（1）查找 3 个数组的最小共同元素。

时间复杂度：

- 哈希表空间：哈希表用于存储元素及其位置，因此其空间占用与元素数量成正比。在这个程序中，使用了一个 map<int, struct location> 类型的哈希表，其中键为整数，值为包含两个整数的结构体。因此，哈希表的空间复杂度可以看作是 O(count_a + count_b)，其中 count_a 和 count_b 分别是两个输入数组的大小。
- 输入数组空间：程序中使用了一个额外的大小为 count_c 的 vector<int> 类型的数组来存储 c 数组。因此，这部分的空间复杂度是 O(count_c)，其中 count_c 是 c 数组的大小。

由于 a,b,c 的大小都小于 n，所以时间复杂度满足 O（n）。

空间复杂度为 ：O(1)

（2）求两个有序序列的中位数

- 时间复杂度：$O(\log \min(m,n)))$，其中 m 和 n 分别是数组 $nums_1$ 和 $nums_2$ 的长度。查找的区间是 $[0,m]$，而该区间的长度在每次循环之后都会减少为原来的一半。所以，只需要执行 $\log m$ 次循环。由于每次循环中的操作次数是常数，所以时间复杂度为 $O(\log m)$。由于我们可能需要交换 $nums_1$ 和 $nums_2$ 使得 $m \leq n$，因此时间复杂度是 $O(\log \min(m,n)))$。
- 空间复杂度：$O(1)$。

（3）二叉排序树的验证和拓展

<u>教材方法一：</u>

- 时间复杂度：

寻找左子树中关键字值最大的节点，需要遍历整个左子树的右分支，其时间复杂度为 O(h)，其中 h 是树的高度。删除节点及其调整操作的时间复杂度也为 O(h)。

- 空间复杂度：

使用递归或迭代删除节点，空间复杂度也为 O(1)

- 查找：

该方法会将左子树中关键字最大的节点作为替代，不会增加左子树的深度，查找性能不变。

教材方法四：

- 时间复杂度：

寻找右子树中关键字值最小的节点，需要遍历整个右子树的左分支，其时间复杂度为 O(h)。删除节点本身及其调整操作也为 O(h)，因此总体时间复杂度为 O(h)。

- 空间复杂度：

使用递归删除操作，可能在递归过程中占用栈空间，但总体空间复杂度依然为 O(1)

- 查找：

该方法倾向于在右子树中寻找关键字最小的节点，然后将左子树连接在该节点上，此操作会增加右子树的深度，特别是如果右子树是原本较平衡的。这可能导致右偏的倾向，降低查找性能。