组号: 1

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：2021-2022 年春季

组　　长：李昀哲

学　　号：20123101

指导教师：朱能军

成绩评定：

二〇二二年 5 月 12 日

小组信息				
登记序号	姓名	学号	贡献比	签名
72	李昀哲	20123101	25%	
21	唐铭锋	20121489	25%	
20	刘沛根	20121483	25%	
22	李正宇	20121517	25%	

实验列表		
实验一	（熟悉上机环境、进度安排、评分制度；分组）	
实验二	*有向网的邻接矩阵验证及拓展*	✓
实验三	无向网的邻接表验证和拓展	✓
实验四	查找算法验证及设计	✓
实验五	(*实验题目*)	

实验四

一、实验题目

查找算法验证及设计

二、实验内容

模仿有向图的邻接表类模板，完成（带权：非负）无向网 的邻接表类模板的设计与实现。要求实现图的基本运算（如增加删除顶点和边等），并增加如下成员函数：

（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 2 种不同的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”，要求其中一种算法在最坏情况下的时间复杂度为 O(n)。

（2）求两个有序序列的中位数。

有两个长度为 n 的有序序列，如果将这两个序列合并成一个有序序列，则处于第 n 个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)。

（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，并实现了其中第一种算法，现要求完成后面 3 种算法的实现，并用多组测试数据对这 4 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

a) 查找 3 个数组的最小共同元素

方法一：①设三个数组下标 a,b,c 从 0 开始；②比较数组 1 中 a 下标的数与数组 2 中 b 下标的数，若两者不相等，则较小数的下标加一；③以此类推，比较 b,c 和 a,c；④循环比较，直到三个数相等，则输出三个数的位置；⑤若一个数组在比较的过程中超出了末尾，着代表没有最小相同元素。

方法二：①从最小的元素开始，遍历比较数组 a 和数组 b，若两个元素相等，则再遍历数组 c，与前两个数进行比较；②若三个数都相等，则输出此时位置；③若三个数组都超出范围则表示未找到。

b） 求两个有序序列的中位数

方法一：①设 a,b 为从零开始的两个数组的下标；②从第零个元素开始逐个比较两个数组，将小的那个放入新的数组中，并将对应的下标增一；③若下标有一个大于等于 n，则结束循环；④将没有遍历完的数组剩下的元素接到新数组的后面；⑤输出数组的第 n 个元素。

方法二：①设 l1=0 和 r1=n-1 是数组 1 的左右范围，mid1=(l1+r1)/2，同理，l2=0 和 r2=n-1 是数组 2 的左右范围，mid2=(l2+r2)/2；②比较对应 mid 下标的两个值，若相等，则直接返回这个值；③若不相等，则让值较小的 r=mid,值较大的则 l=mid，重复比较过程，直到两者相等或者有 l=r；④若最后有 l=r，则判断 r1 和 r2 哪个小，返回较小的值。

c） 二叉排序树的验证和拓展

方法一：在要删除节点的右子树中寻找关键字值最小的数据 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。

方法二：先把要删除节点的右子树作为左子树中关键字值最大的数据元素 x 的右子树，然后在删除结点。

方法三：先把要删除节点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，然后在删除结点。

2、源程序代码

a) 查找 3 个数组的最小共同元素

```cpp
/**
 ** Second efficient algorithm.
 ** [param] 3 array and 3 array size
 ** [out] Found: output min same number and their position, return true
 **/
bool FindMinNumber_elseif(const int *a, const int * b, const int *c, int aNum, int bNum, int cNum)
{
    int i = 0, j = 0, k = 0, same_min_number, cnt = 0;
    while(i < aNum && j < bNum && k < cNum)
    {
        cnt ++;
        if(a[i] < b[j]) i++;
        else if(b[j] < c[k]) j++;
        else if(c[k] < a[i]) k++;
        else
            {
                same_min_number = a[i];
                std::cout << same_min_number << " " << i + 1 << " " << j + 1 << " " << k + 1 << std::endl;
                std::cout << "cnt_elseif = " << cnt << std::endl;
                return true;
            }
    }
    return false;
}
```

```cpp
/**
 ** Most efficient algorithm.
 ** [param] 3 array and 3 array size
 ** [out] Found: output min same number and their position, return true
 **/
bool FindMinNumber_if(const int *a, const int * b, const int *c, int aNum, int bNum, int cNum)
{
    int i = 0, j = 0, k = 0, same_min_number, cnt = 0;
    while(i < aNum && j < bNum && k < cNum)
    {
        cnt ++;
        if(a[i] < b[j]) i++;
        if(b[j] < c[k]) j++;
        if(c[k] < a[i]) k++;
        if(a[i] == b[j] && b[j] == c[k])
        {
            same_min_number = a[i];
            std::cout << same_min_number << " " << i + 1 << " " << j + 1 << " " << k + 1 << std::endl;
            std::cout << "cnt_if = " << cnt << std::endl;
            return true;
        }
    }
    return false;
}
```

```cpp
/**
** Least efficient algorithm.
**Traverse a, b and c, if found same number in a and b, search in c.
** [param] 3 array and 3 array size
** [out] Found: output min same number and their position, return true
**/
bool FindMinNumber_slow(const int *a, const int * b, const int *c, int aNum, int bNum, int cNum)
{
    for(int i = 0; i < aNum; i++)
        for(int j = 0; j < bNum; j++)
            if(a[i] == b[j])
                for(int k = 0; k < cNum; k++)
                    if (c[k] == a[i]){std::cout << c[k] << " " << i + 1 << " " << j + 1 << " " << k + 1 << std::endl; return true;}

    return false;
}
```

```cpp
void Input(int *array, int arrayNum)
{
    std::cout << "Input " << arrayNum << " elements: ";
    for(int i = 0; i< arrayNum; i++)
        std::cin >> array[i];
}
```

```cpp
int main()
{
    int a[10], b[10], c[10], aNum, bNum, cNum;
    std::cout << "Input 3 numbers as the size of 3 array: "; std::cin >> aNum >> bNum >> cNum;

    Input(a, aNum);Input(b, bNum);Input(c, cNum);


    // O(n), but elseif num is too much
    std::cout << "=================================================================" << std::endl;
    std::cout << "O(n), second optimal algorithm" << std::endl;
    if ( FindMinNumber_elseif(a, b, c, aNum, bNum, cNum))
        std::cout << "Founded" << std::endl;
    else
        std::cout << "Not Founded" << std::endl;

    // O(n), optimal algorithm
    std::cout << "=================================================================" << std::endl;
    std::cout << "O(n), optimal algorithm" << std::endl;
    if ( FindMinNumber_if(a, b, c, aNum, bNum, cNum))
        std::cout << "Founded" << std::endl;
    else
        std::cout << "Not Founded" << std::endl;

    // O(n^3)
    std::cout << "=================================================================" << std::endl;
    std::cout << "O(n^3) algorithm" << std::endl;
    if ( FindMinNumber_slow(a, b, c, aNum, bNum, cNum))
        std::cout << "Founded" << std::endl;
    else
        std::cout << "Not Founded" << std::endl;

    return 0;
}
```

b) 求两个有序序列的中位数

```
int searchmid1(const int *num1,const int *num2,int n) //第一种查找中位数，算法时间复杂度为O(n)
{
    int *num;
    num=new int[2*n];
    int k=0,i,j;
    for(i=0,j=0;i<n && j<n;)  //将两个有序数组依次比较各元素，并有序插入新数组，即合并
    {
        if(num1[i]<num2[j])
        {
            num[k]=num1[i];
            k++;
            i++;
        }
        else if(num1[i]>num2[j])
        {
            num[k]=num2[j];
            k++;
            j++;
        }
        else
        {
            num[k]=num1[i];
            num[k+1]=num2[j];
            i++;
            j++;
            k=k+2;
        }
    }
    if(i==n && j<n)  //判断是否有数组的元素没遍历完，没遍历完的元素都依次加入新数组
    {
        for(;j<n;j++)
        {
            num[k]=num2[j];
            k++;
        }
    }
    else if(j==n&&i<n)
    {
        for(;i<n;i++)
        {
            num[k]=num1[i];
            k++;
        }
    }
    return num[n-1];  //返回合并后的有序数组的中位数
}
int searchmid2(int *num1,int *num2,int n)  //第二种求中位数，采用折半查找，算法时间复杂度为O(logn)
{
    int mid1,mid2;
    int l1=0,r1=n-1,l2=0,r2=n-1;
    while(l1!=r1||l2!=r2)
    {
        mid1=(l1+r1)/2;
        mid2=(l2+r2)/2;
        if(num1[mid1]==num2[mid2])  //比较两个数组区间的中位数
        {
            return num1[mid1];
        }
        else if(num1[mid1]<num2[mid2])
        {
            if((l1+r1)%2==0)  //判断是奇数还是偶数
            {
                l1=mid1;
                r2=mid2;
            }
            else
```

```
            {
                l1=mid1+1;
                r2=mid2;
            }
        }
        else
        {
            if((l1+r1)%2==0)
            {
                r1=mid1;
                l2=mid2;
            }
            else
            {
                r1=mid1;
                l2=mid2+1;
            }
        }
    }
    return num1[r1]<num2[r2]?num1[r1]:num2[r2];  //返回
}
```

```
int main() {

    int n;
    cin>>n;
    int *num1=new int[n];
    int *num2=new int[n];
    for(int i=0;i<n;i++)
    {
        cin>>num1[i];
    }
    for(int i=0;i<n;i++)
    {
        cin>>num2[i];
    }
    cout<<searchmid2(num1,num2,n)<<endl;
    cout<<searchmid1(num1,num2,n)<<endl;

    return 0;
}
```

c） 二叉排序树的验证和拓展

```
template <class ElemType>
bool BinarySortTree<ElemType>::Delete(const ElemType &key)
// 操作结果：删除关键字为key的数据元素
{
    BinTreeNode<ElemType> *p, *f;
    p = Find(key, f);
    if ( p == NULL) // 查找失败，删除失败
        return false;
    else    // 查找成功，插入失败
    if (f == NULL)  // 被删除结点为根结点
        Delete4( &p);
    else if (key < f->data) // elem.key更小,删除f的左孩子
        Delete4( &f->leftChild);
    else    // elem.key更大，删除f的右孩子
        Delete4( &f->rightChild);
    return true;
}
```

```cpp
template<class ElemType>
void BinarySortTree<ElemType>::Delete2(BinTreeNode<ElemType> *&p) {
    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只有右子树非空
        tmpPtr = p;
        p = p->leftChild;
        delete tmpPtr;
    }
    else {
        tmpF = p;
        tmpPtr = p->rightChild;
        while (tmpPtr->leftChild != 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->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->leftChild == tmpPtr)  // 删除tmpF的左孩子
            Delete2(&tmpF->leftChild);
        else    // 删除tmpF的右孩子
            Delete2(&tmpF->rightChild);
    }
}
```

```cpp
template<class ElemType>
void BinarySortTree<ElemType>::Delete3(BinTreeNode<ElemType> *&p) {
    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只有右子树非空
        tmpPtr = p;
        p = p->leftChild;
        delete tmpPtr;
    }
    else {
        tmpPtr = p->leftChild;
        while (tmpPtr->rightChild != NULL)      // 找到左子树中的最大值
            tmpPtr = tmpPtr->rightChild;
        tmpPtr->rightChild = p->rightChild;     // 将原右子树作为左子树中的最大值的右子树
        p->rightChild = NULL;                   // 删除结点
        if(p == root)
            root = p->leftChild;
        Delete3( &p);
    }
}
```

```cpp
template<class ElemType>
void BinarySortTree<ElemType>:: Delete4(BinTreeNode<ElemType> *&p) {
    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只有右子树非空
        tmpPtr = p;
        p = p->leftChild;
        delete tmpPtr;
    }
    else {
        tmpPtr = p->rightChild;
        while (tmpPtr->leftChild != NULL)   // 找到右子树中的最小值
            tmpPtr = tmpPtr->leftChild;
        tmpPtr->leftChild = p->leftChild;   // 将原左子树作为右子树中的最大值的左子树
        p->leftChild = NULL;                // 删除结点
        if(p == root)
            root = p->rightChild;
        Delete4( &p);
    }
}
```

3、实验结果

a） 查找 3 个数组的最小共同元素

可以看出，依次遍历两个数组，在其中找到相同的元素再去第三个数组查找，这样的效率最低，时间复杂度为 O(n^3);而修改版用 if else 实现，虽然时间复杂度有大幅减少，减少至 O(n)，但测试中发现仍会循环较多次数，因此，再次进行修改，将 if else 全部修改为 if，在一次循环中对数组中的元素做尽可能多次数的移动，从而达到较好的效果

```
Input 3 numbers as the size of 3 array:2 4 5
 Input 2 elements:3 4
 Input 4 elements:4 7 8 9
 Input 5 elements:1 2 3 4 5
 ==========================
======================================
O(n), second optimal algorithm
4 2 1 4
cnt_elseif = 5
Founded
======================================================================
O(n), optimal algorithm
4 2 1 4
cnt_if = 3
Founded
======================================================================
O(n^3) algorithm
4 2 1 4
Founded
```

b） 求两个有序序列的中位数

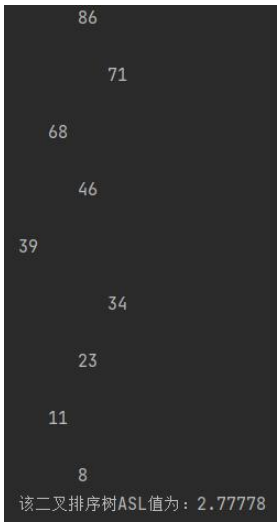
```
8

1 3 5 7 9 11 13 15

12 14 16 18 20 22 24 26
13
13
```

c） 二叉排序树的验证和拓展

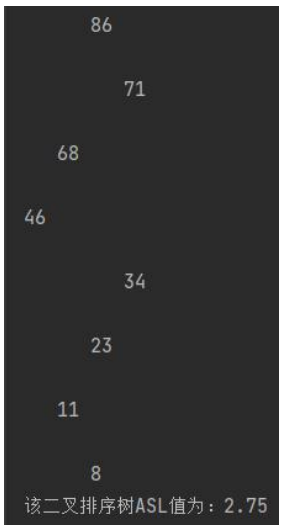
原始树为：

用方法一：

删除 75 后：

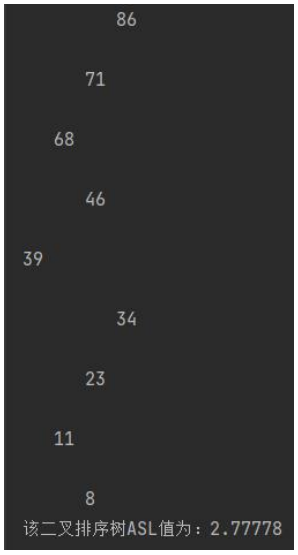


删除 39 后：

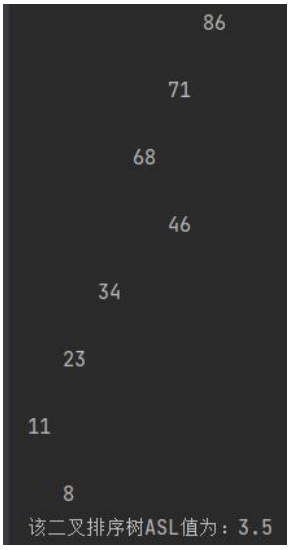


用方法二：

删除 75 后：

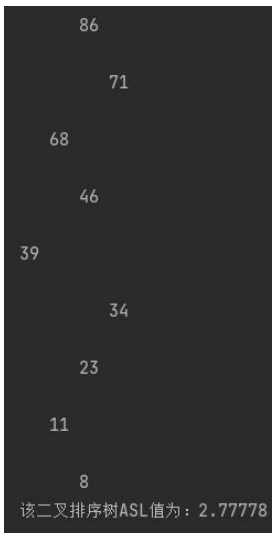


删除 39 后：

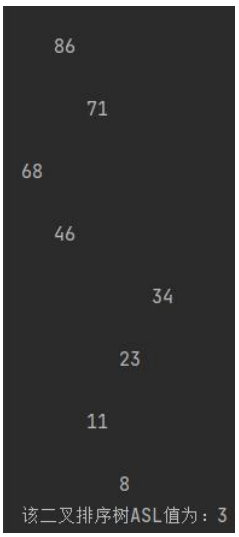


用方法三：

删除 75 后：



删除 39 后：



由此可见，对于子树比较复杂的节点而言，用方法一的查找效率要高于其他两种方法。

实验现象与预想的实验现象一致。

4、算法分析

a) 查找 3 个数组的最小共同元素

方法一，时间复杂度为 $\Theta(n)$；方法二为 $\Theta(n^3)$。

b) 求两个有序序列的中位数

方法一，时间复杂度为 $\Theta(n)$；方法二为 $\Theta(n^3)$。

5、总结与心得

在本次实验的过程中我们小组通过讨论，实践，查阅资料等方法完成了实验任务。通过实验，我们对于各种查找算法和二叉排序树的实现有了更深入的理解。此外，我们还复习了二叉树的实现等并运用到了解决问题中。总之，查找算法在实际应用、生活中都是十分常见的，通过此次实验，我们提高了算法能力，增加了查找方面数据结构的知识。

四、分工说明

李昀哲：算法设计，代码编写，ppt 制作

唐铭锋：算法设计，代码编写，撰写报告

刘沛根：算法设计，代码编写，ppt 制作

李正宇：算法设计，代码编写，撰写报告