中山大学计算机学院本科生实验报告

（2021 学年第 1 学期）

课程名称：Data structures and algorithms　　任课教师：张子臻

年级	20 级	专业（方向）	软件工程
学号	20337270	姓名	钟海财
电话	13397996670	Email	2940599563@qq.com
开始日期	2021/11/11	完成日期	2021/11/17

1. 实验题目

1）二叉搜索树的遍历

2）postorder

2.实验目的

1）输入一系列数字及个数，通过构建二叉搜索树，输出该二叉搜索树的中序遍历和前序遍历的结果

2）输入一系列数字的个数、这些数字构成的二叉树的前序遍历的结果 A[]和中序遍历的结果 B[]，输出这些数字构成的二叉树的后序遍历的结果。

3.程序设计

1）二叉搜索树的遍历

设计思路：

首先，定义节点的结构体：

```
1. struct BitNode{
2.     int data;
3.     BitNode* lchild;//左儿子
```

```
    BitNode* rchild;//右儿子
};
```

由二叉搜索树的结构特点可知：对任意一个节点，其左儿子节点的值比其小，其右儿子节点的值比其大，这样才能使中序遍历的结果是从小到大排好序的。

开始第一个数我们设置为根节点，于是我们可使用递归来将节点插入二叉搜索树中：首先比较该数与根节点的大小，若更小，递归插入左子树（左儿子不存在时直接插入成左儿子），若更大，递归插入右子树（右儿子不存在时直接插入成右儿子）：

```
void build(BitNode* T, int num){
        if(T->data>num)//比根节点小就往左儿子节点插入
        {
            if(!T->lchild)  //左儿子不存在直接插入
            {
                BitNode* lt=new BitNode();
                lt->data=num;
                T->lchild=lt;
            }
             else// 左儿子存在插入左子树
                 build(T->lchild, num);//调用递归
         }
         else   //比根节点大就往左儿子节点插入
         {
            if(!T->rchild)//右儿子不存在直接插入
            {
                BitNode* rt=new BitNode();
                rt->data=num;
                T->rchild=rt;
            }
            else    // 右儿子存在插入右子树
                build(T->rchild, num);
         }
    }
```

同时，二叉树的前序遍历和中序遍历也使用递归实现：

```
//前序遍历：根节点->左子树->右子树
void preorder(BitNode* T){
    if(T){
        cout<<T->data<<" ";
```

```
        preorder(T->lchild);
        preorder(T->rchild);
    }
}
//中序遍历：左子树->根节点->右子树
void inorder(BitNode* T){
    if(T){
        inorder(T->lchild);
        cout<<T->data<<" ";
        inorder(T->rchild);
    }
}
```

最后，防止内存泄漏，节点的删除也使用递归实现：删除左子树->右子树->根节点：

```
void destory(BitNode* T){ //释构函数，递归删除所有节点
    if(T->lchild) destory(T->lchild);
    if(T->rchild) destory(T->rchild);
    delete T ;
}
```

最后，把这些函数封装到 BStree 类中，并在类中实现上述函数的根节点版本。

代码：

```
#include<iostream>
using namespace std;

struct BitNode{
    int data;
    BitNode* lchild;//左儿子
    BitNode* rchild;//右儿子
};

class  BStree{
private:
    BitNode * T0; //根节点的指针
public:
    BStree(){
    T0 = NULL ;
    }
    BStree(int t){
```

```
    BitNode* p=new BitNode();
    p->data = t ;
    T0 = p ;
    }
    ~BStree(){
    destory(T0);
    }
    void  buildT0(int num){
        build(T0,num);
    }
    void preorderT0(){
        preorder(T0);
    }
    void inorderT0(){
        inorder(T0);
    }
    //构建二叉查找树
    void build(BitNode* T, int num){
        if(T->data>num)//比根节点小就往左儿子节点插入
        {
            if(!T->lchild)  //左儿子不存在直接插入
            {
                BitNode* lt=new BitNode();
                lt->data=num;
                T->lchild=lt;
            }
            else// 左儿子存在构建左子树
                build(T->lchild, num);//调用递归
        }
        else   //比根节点大就往左儿子节点插入
        {
            if(!T->rchild)//右儿子不存在直接插入
            {
                BitNode* rt=new BitNode();
                rt->data=num;
                T->rchild=rt;
            }
            else    // 右儿子存在构建右子树
                build(T->rchild, num);
        }
    }

    //前序遍历：根节点->左子树->右子树
    void preorder(BitNode* T){
        if(T){
            cout<<T->data<<" ";
```

```
69.             preorder(T->lchild);
70.             preorder(T->rchild);
71.         }
72.     }
73.     //中序遍历：左子树->根节点->右子树
74.     void inorder(BitNode* T){
75.         if(T){
76.             inorder(T->lchild);
77.             cout<<T->data<<" ";
78.             inorder(T->rchild);
79.         }
80.     }
81.     void destory(BitNode* T){ //释构函数，递归删除所有节点
82.         if(T->lchild) destory(T->lchild);
83.         if(T->rchild) destory(T->rchild);
84.         delete T ;
85.     }
86. };
87.
88.     int main(){
89.         int n;
90.         while(cin>>n && n!=0 ){
91.             int num;
92.             cin>>num;
93.             BStree Tree(num);//根节点
94.             for(int i = 0 ; i<n-1;++i){
95.                 cin>>num;
96.                 Tree.buildT0(num);
97.             }
98.             Tree.inorderT0();//中序遍历
99.             cout<<endl;
100.               Tree.preorderT0();//前序遍历
101.               cout<<endl;
102.             }
103.           return 0;
104.       }
```

2）postorder

设计思路：

根据三种遍历的特点：

前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

已知前序遍历的结果和后序遍历的结果，要求后序遍历的结果，可根据前序遍历的结果找到对应根节点在中序遍历结果中的位置，将中序遍历的结果划分为 3 个部分：左子树 ，根结点 ，右子树。再用后序遍历的方式输出：左子树 ---> 右子树 ---> 根结点。我们可以发现，前序遍历中第 k 个根节点，就是中序遍历中第 k 次递归划分时的根节点，也是后序遍历中第 k 次递归要输出的根节点。

于是构造递归函数 post_cout，A[]是前序遍历的结果，B[]是中序遍历的结果，start 和 end 是该子树在 B[]中的开始位置和结束位置，root 是该子树的根节点在 A[]中的位置：

```
1.void post_cout(int A[] , int B[],int start,int end,int& root){
2.    if(start>end) return;
3.    int key;//记录根节点 A[root]在 B[]的位置
4.    for(int i = start; i<=end ;++i){
5.        if(A[root]==B[i]) {
6.            key = i ;
7.            break;
8.        }
9.    }
10.     ++root;//记录 A[]中下一个要访问的根节点位置
11.     post_cout(A,B,start,key-1,root);//后序输出左子树
12.     post_cout(A,B,key+1,end,root);//后序输出右子树
13.     cout<<B[key]<<" ";//输出根节点
14. }
```

代码：

```
#include<iostream>
using namespace std;

void post_cout(int A[] , int B[],int start,int end,int& root){
    if(start>end) return;
    int key;//记录根节点 A[root]在 B[]的位置
    for(int i = start; i<=end ;++i){
        if(A[root]==B[i]) {
            key = i ;
             break;
        }
    }
    ++root;//记录 A[]中下一个要访问的根节点位置
    post_cout(A,B,start,key-1,root);//后序输出左子树
    post_cout(A,B,key+1,end,root);//后序输出右子树
    cout<<B[key]<<" ";//输出根节点
}
int main(){
    int n;
    cin>>n;
    int A[n];//前序遍历：根节点->左子树->右子树
    int B[n];//中序遍历：左子树->根节点->右子树
    for(int i = 0 ; i < n ; ++i) cin >>A[i];
    for(int i = 0 ; i < n ; ++i) cin>>B[i];
    int root = 0;//记录 A[]中根节点遍历的根节点的位置
    post_cout(A,B,0,n-1,root);//后序遍历：左子树->右子树->根节点
    return 0 ;
}
```

4.程序运行与测试

1）二叉搜索树的遍历

测试输入 1：

```
9
10 4 16 9 8 15 21 3 12
6
20 19 16 15 45 48 0
```

测试输出 1：

```
3 4 8 9 10 12 15 16 21
10 4 3 9 8 16 15 12 21
15 16 19 20 45 48
20 19 16 15 45 48
```

测试输出 1：通过

测试输入 2：

```
16
10 4 16 9 8 15 21 3 12 20 19 1 17 45 48 0
```

测试输出 2：

```
0 1 3 4 8 9 10 12 15 16 17 19 20 21 45 48
10 4 3 1 0 9 8 16 15 12 21 20 19 17 45 48
```

测试输出 2：通过

2）postorder

测试输入 1：

```
10
7 2 0 5 8 4 9 6 3 1
7 5 8 0 4 2 6 3 9 1
```

测试输出 1：

```
8 5 4 0 3 6 1 9 2 7
```

测试输出 1：通过

测试输入 2：

```
16
10 4 3 1 0 9 8 16 15 12 21 20 19 17 45 48
0 1 3 4 8 9 10 12 15 16 17 19 20 21 45 48
```

测试输出 2：

```
0 1 3 8 9 4 12 15 17 19 20 48 45 21 16 10
```

测试输出 2：通过

5.实验总结与心得

解决二叉树的遍历问题，关键是要理解二叉树三种遍历方式：

前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

值得注意的是，在遍历子树时也要采用该种遍历方式，这时我们就可以使用递归来进行遍历。

附录、提交文件清单