第四题 二叉树的应用

【问题描述】

二叉搜索树（BST）定义为具有以下属性的二叉树：

- 任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值
- 任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值
- 任意节点的左、右子树也分别为二叉查找树

完全二叉树（CBT）的深度为 k，除第 k 层外，其他各层（1～（k-1）层）的节点数都达到最大值，且第 k 层所有的节点都连续集中在最左边。

现在，给定一个键值互不相同的非负整数序列，构造一颗树既是 CBT，也是 BST。请你输出这个 BST 的层次遍历序列。

【输入形式】

每个输入文件的第一行为一个正整数 N（≤20），即二叉树中结点的总数。第二行给出了 N 个不同的非负键值序列。

注意：每一行中的所有数字都用一个空格隔开，并且不大于 50。

【输出形式】

相应完全搜索二叉树的层次遍历序列输出在一行中。一行中的所有数字必须由一个空格隔开，并且行首和行尾不得有多余的空格。

【样例输入】

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

【样例输出】

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

一、 问题分析和任务定义

本题给定一个非负整数序列，要求构建出一棵既是二叉搜索树，又是完全二叉树的树，然后输出这棵树的层次遍历序列。

任务一：输入给定序列

任务二：构建完全二叉搜索树

任务三：层次遍历输出

二、 样例推导

先将输入序列按照升序排列，得到0 1 2 3 4 5 6 7 8 9，然后通过计算得到根结点的左子树的结点数，为6，所以0 1 2 3 4 5为左子树部分，6为根结点，7 8 9为右子树部分。在分别对左右子树重复上述操作，直到遇到叶结点递归结束。层次遍历输出后得到6 3 8 1 5 7 9 0 2 4，经过验证，符合完全二叉树和二叉搜索树的要求。

三、 数据结构设计和定义

```
template <typename Elem>
class BinTree
{
   public:
      BinTree() {} //构造函数
      ~BinTree() {} //析构函数
      virtual void setRoot(BinNode<Elem>* rt) = 0; //设置根结点
      virtual BinNode<Elem>* getRoot() = 0; //返回根结点
      virtual void clear(BinNode<Elem>* rt) = 0; //清空二叉树
      virtual bool BinTreeEmpty(BinNode<Elem>* rt) = 0; //判断二叉树是否为空树
      virtual void preorder(BinNode<Elem>* rt) = 0; //前序遍历
      virtual void Inorder(BinNode<Elem>* rt) = 0; //中序遍历
      virtual void postorder(BinNode<Elem>* rt) = 0; //后序遍历
      virtual void LevelOrderTraverse(BinNode<Elem>* rt) = 0; //层次遍历
      virtual int BinTreeDepth(BinNode<Elem>* rt) = 0; //获得二叉树的深度
      virtual int count(BinNode<Elem>* rt) = 0; //获得二叉树的结点数

      virtual bool find(BinNode<Elem>* rt,const Elem& e) = 0; //查找二叉树中是否含有元素e
```

virtual　int countLeft(vector<Elem>& v) = 0; //计算二叉树的左子树的结点数

virtual BinNode<Elem>* CreateCBST(vector<Elem>& v) = 0; //创建完全二叉搜索树

};

物理存储结构采用二叉链表来实现。

四、 算法思想

1. 首先将输入序列存储在 vector 容器中。
2. 再对 vector 中数据从小到大排序。
3. 根据二叉搜索树的性质，某一结点的左子树都小于该结点，右子树都大于该结点，根据完全二叉树的性质，给定一个固定大小的结点总数 n，就可以计算出根结点左子树的结点数 x，于是有序序列 vector 的前 x 个就是左子树，第 x+1 就是根结点，剩下的就是右子树。
4. 从根结点开始对序列递归划分，得到完全二叉搜索树 T，然后通过队列实现层次遍历输出。

五、 关键功能的算法设计

```
//数据输入
vector<int>  v;
int  value;
for(int  i=0;i<N;i++)
{
    cin>>value;
    v.push_back(value);
}
sort(v.begin(),v.end());  //将输入序列升序排列
//数据处理
int  countLeft(vector<Elem>&  v)  //计算二叉树的左子树的结点数
    {
        int  N=v.size();
        int  n=0,x=0;
```

```
        n=log(N+1)/log(2);  //除叶结点外满二叉树的层数
        x=N-(pow(2,n)-1);  //叶结点数
        x=min(x,int(pow(2,n-1)));  //左子树的叶结点数
        x=x+pow(2,n-1)-1;  //左子树的结点数
        return  x;
    }

    BinNode<Elem>*  CreateCBST(vector<Elem>&  v)  //创建完全二叉搜索
树
    {
        BinNode<Elem>*  rt=new  BinNode<Elem>;
        if(v.size()==1)  //如果是叶结点
        {
            rt->setValue(v[0]);
            return  rt;
        }
        int  x=countLeft(v);  //计算二叉树的左子树的结点数
        rt->setValue(v[x]);  //根结点
        vector<int>  vl,vr;
        vl.assign(v.begin(),v.begin()+x);  //左子树
        if(x+1<=v.size()-1)  //如果右子树不为空
            vr.assign(v.begin()+x+1,v.end());  //右子树
        rt->setLeft(CreateCBST(vl));  //创建左子树的完全二叉搜索树
        if(vr.size())  //如果右子树不为空
            rt->setRight(CreateCBST(vr));  //创建右子树的完全二叉搜索树
        return  rt;
    }
//数据输出
void  LevelOrderTraverse(BinNode<Elem>*  rt)  //层次遍历
    {
```

```
        if(rt==NULL)  return;
        queue<BinNode<Elem>*>  q;
        q.push(rt);
        BinNode<Elem>*  curr;
        while(!q.empty())
        {
            curr=q.front();
            if(curr->left()!=NULL)
                q.push(curr->left());
            if(curr->right()!=NULL)
                q.push(curr->right());
            printf("%d  ",curr->getValue());
            q.pop();
        }
    }
```

六、 性能分析

1. 数据输入部分

    for 循环共执行 n 次，所以时间复杂度为 O(n)，空间复杂度为 O(n)。

2. 数据处理部分

    countLeft 函数，时间复杂度为 O(1)，空间复杂度为 O(1)。

    CreateCBST 函数，时间复杂度为 $O(\log_2 n)$，空间复杂度为 O(n)。

3. 数据输出部分

    LevelOrderTraverse 函数，时间复杂度为 O(n)，空间复杂度为 O(n)。