重　庆　交　通　大　学

《算法与数据结构》课程

实　　验　　报　　告

班　　级：计算机专业 19 级　曙光班

姓名　学号：周迎川　631907060434

实验项目名称：二叉树及其操作

实验项目性质：验证性实验

实验所属课程：数据结构 A

实验室(中心)：B01 409

指 导 教 师 ：鲁云平

实验完成时间：2020 年 11 月 21 日

教师评阅意见： 签名：　　年　月　日
实验成绩：

一、实验目的

1、实现二叉树的存储结构（二叉链表或三叉链表等存储结构任选）
2、熟悉二叉树基本术语的含义
3、掌握二叉树相关操作的具体实现方法

二、实验内容及要求

内容：

1. 建立二叉树
2. 计算结点所在的层次
3. 统计结点数量和叶结点数量
4. 计算二叉树的高度
5. 计算结点的度
6. 找结点的双亲和子女
7. 二叉树前序、中序、后序遍历的递归实现和非递归实现及层次遍历
8. 二叉树的复制
9. 二叉树的输出等

其它操作可根据具体需要自行补充。

要求：

1.独立完成实验内容

2.自行实现二叉树的存储结构与相关操作，不得使用 STL(标准模板库)现成代码；

3.二叉树结点的数据类型自行定义；

4.编程语言：C++

5.使用类模板完成二叉树的定义

三、系统分析

（1）数据方面：

本实验需要建立一个二叉树的存储结构及其基本操作，要定义一个树的结点类，结点类中包含数据类和左子树指针和右子树指针。然后定义一个二叉树类，二叉树类中包含结点类的指针，这个指针定义为私有类型，体现类的封装性。

（2）功能方面：

1. 建立二叉树
2. 计算结点所在的层次
3. 统计结点数量和叶结点数量
4. 计算二叉树的高度
5. 计算结点的度
6. 找结点的双亲和子女
7. 二叉树前序、中序、后序遍历的递归实现和非递归实现及层次遍历
8. 二叉树的复制
9. 二叉树的输出
10. 二叉树的判空

四、系统设计

（1）设计的主要思路

说明整体设计思路

根据题目要求：

1. 建立二叉树
2. 计算结点所在的层次
3. 统计结点数量和叶结点数量
4. 计算二叉树的高度
5. 计算结点的度
6. 找结点的双亲和子女
7. 二叉树前序、中序、后序遍历的递归实现和非递归实现及层次遍历
8. 二叉树的复制
9. 二叉树的输出等
10 使用类模板完成二叉树的定义

通过链式二叉树储存，设计出一个结点类，存储二叉树的各个结点，定义一个数据域和两个指针域数据与用于存储数据，指针域用于存储二叉树左右子树的地址值。然后再定义一个二叉树类，里面定义两个一个数据域和一个指针域，指针域用于记录二叉树的头指针，数据域用于记录输入数据停止的位置。

（2）数据结构的设计

数据结构设计思路

```
#ifndef BINTREE_H
#define BINTREE_H
#include <iostream>
using namespace std;
template <typename T>
struct BinTreeNode
{
    T date;
    BinTreeNode *lchild,*rchild;
    BinTreeNode()
    {
        lchild=nullptr;rchild=nullptr;
    }
    BinTreeNode(T &x,BinTreeNode<T> *l=nullptr,BinTreeNode<T>
*r=nullptr)
    {
        date=x;rchild=r;lchild=l;
    }
};
template <typename T>
class BinTree
{
private:
    BinTreeNode<T> *root;
    T stop;
    //bool Insert(BinTreeNode<T> *&tree,T& x);//插入操作
    void CreateBinTree(BinTreeNode<T> *&tree);//前序创建二叉树
    void destroy(BinTreeNode<T>*& tree);//删除
    BinTreeNode<T>* Find(BinTreeNode<T>*& tree,T& x);//查找 x
    int Height(BinTreeNode<T>* tree);//求树的高度
    int Size(BinTreeNode<T>* tree);//求树的结点个数
    int LeavesCount(BinTreeNode<T> *tree);//求叶节点数量
    BinTreeNode<T>* Parent(BinTreeNode<T>* tree,BinTreeNode<T>*
now);//找到双亲结点
    void visit(BinTreeNode<T>*tree);//访问函数
    void preOrder(BinTreeNode<T> *tree);//前序遍历
    void inOrder(BinTreeNode<T> *tree);//中序遍历
    void postOrder(BinTreeNode<T> *tree);//后续遍历
    void PreOrder(BinTreeNode<T> *tree);//前序遍历
    void InOrder(BinTreeNode<T> *tree);//中序遍历
    void PostOrder(BinTreeNode<T> *tree);//后续遍历
    void NodeLevel(BinTreeNode<T>* tree,T &x,int lev,int &level);//某
个结点再哪一层
```

```
    //friend istream& operator>>(istream& in,BinTreeNode<T>& tr);//重
载>>
    //friend ostream& operator<<(ostream& out,BinTreeNode<T>& tr);//
重载<<
public:
    BinTree()//构造函数
    {
        root=nullptr;stop=-1;
    }
    BinTree(BinTree<T>& c);//复制构造函数
    void CreateBinTree()
    {
        CreateBinTree(this->root);
    }
    ~BinTree()//析构函数
    {
        destroy(root);
    }
    bool IsEmpty()//判空
    {
        return (root==nullptr)? true:false;
    }
    int NodeDeg(BinTreeNode<T>*now);//计算该节点的度
    BinTreeNode<T>* Parent(BinTreeNode<T> *now)//返回父节点
    {
        return (root==nullptr||now==root) ? nullptr:Parent(root,now);
    }
    BinTreeNode<T>* LeftChild(BinTreeNode<T>* now);//找到结点的左子女
    BinTreeNode<T>*RightChild(BinTreeNode<T>*now);//找到结点的有子女
    BinTreeNode<T>* Copy(BinTreeNode<T>* origin);//复制
    void Traverse(BinTreeNode<T>* tree);//前序遍历输出
//    BinTreeNode<T>*LeftChild(BinTreeNode<T> *now)//返回左子树结点地
址
//    {
//        return (now==nullptr)? nullptr:now->lchild;
//    }
//    BinTreeNode<T>* RightChild(BinTreeNode<T> *now)//返回右子树结点
地址
//    {
//        return (now==nullptr)? nullptr:now->rchild;
//    }
    int Height()//计算树的高度
    {
        return Height(root);
```

```
}
int Size()//计算树的结点数
{
    return Size(root);
}
BinTreeNode<T>* getRoot()//获取根结点
{
    return root;
}
void preOrder()//前序遍历
{
    preOrder(this->root);
}

void inOrder()//中序遍历
{
    inOrder(this->root);
}
void postOrder()//后续遍历
{
    postOrder(this->root);
}
void PreOrder()//前序遍历
{
    preOrder(this->root);
}

void InOrder()//中序遍历
{
    inOrder(this->root);
}
void PostOrder()//后续遍历
{
    postOrder(this->root);
}
//int Insert(T& x);
BinTreeNode<T>* Find(T& x)
{
    return Find(this->root,x);
}
int LeavesCount()
{
return LeavesCount(root);
}
```

```
    int NodeLevel(T& x)
    {
        int level=0;
        NodeLevel(root,x,0,level);
        return level;
    }
    void setRoot(BinTreeNode<T>*tree)
    {
        root=tree;
    }
};
#endif // BINTREE_H
```

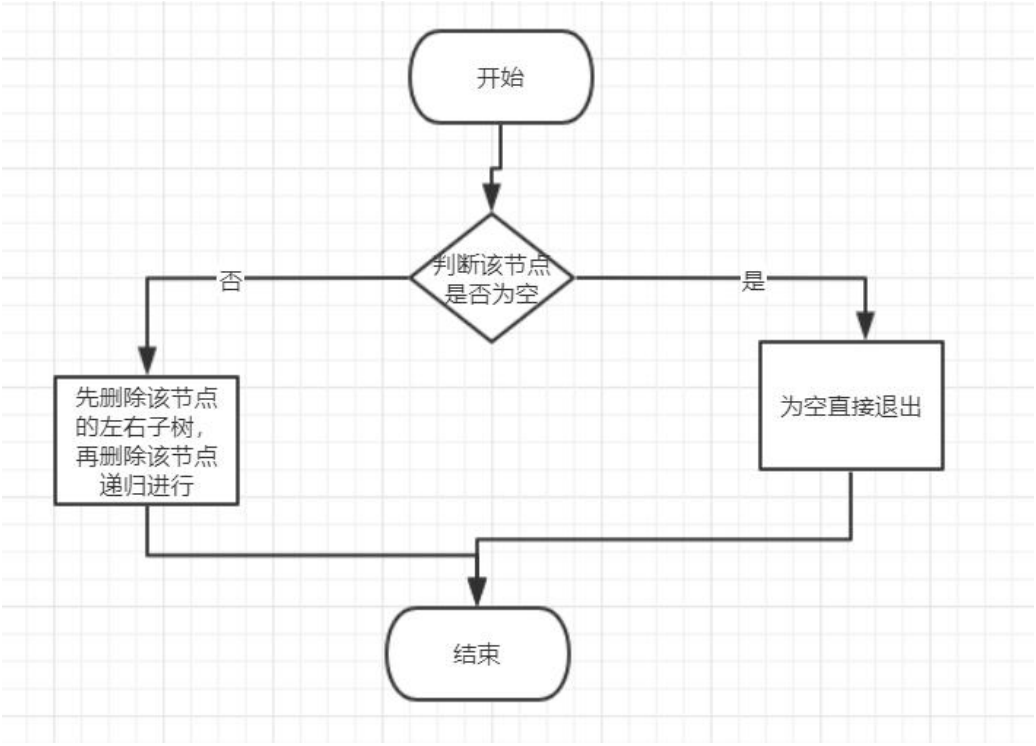
首先设计出一个结点类，存储二叉树的各个结点，定义一个数据域和两个指针域数据与用于存储数据，指针域用于存储二叉树左右子树的地址值。然后再定义一个二叉树类，里面定义两个一个数据域和一个指针域，指针域用于记录二叉树的头指针，数据域用于记录输入数据停止的位置。由于树的非递归遍历需要栈的存储形式，所以还需要定义一个栈的存储结构。

（3）基本操作的设计

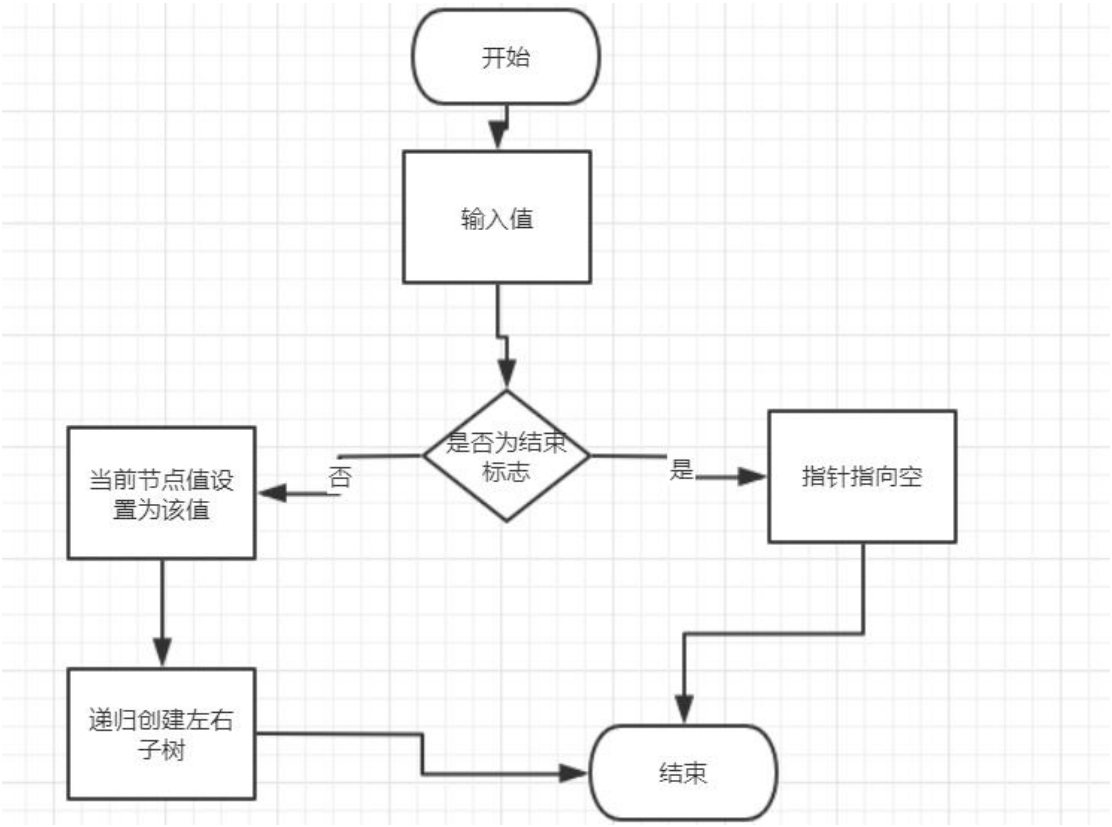
基本操作的抽象描述，关键算法的设计思路和算法流程图。

基本操作的抽象描述一般为操作名，初始条件，操作结构，参数说明等。

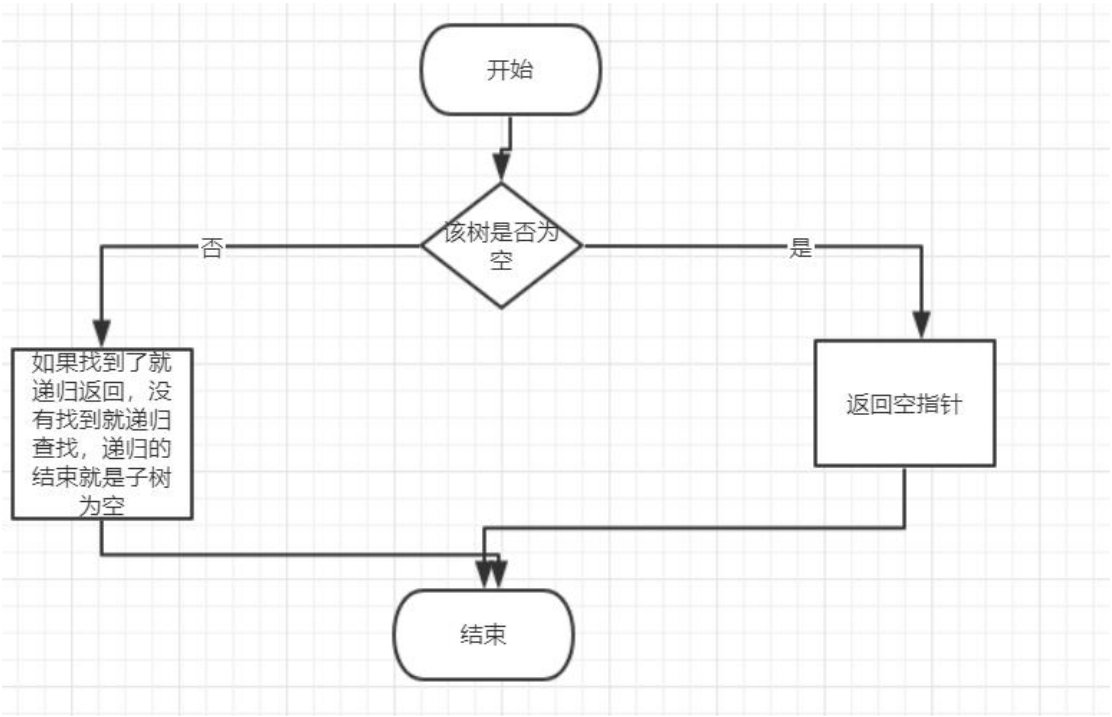
```
void BinTree<T>::CreateBinTree(BinTreeNode<T> *&tree)
{
    /*
     * 创建一个树，首先传递一个引用指针参数，不能只是指针，必须要引用
     * 输入数据，以-1 结束也就是 stop 变量
     * 如果不是-1，利用前序遍历新建一个树，先新建一个结点，然后再创建该节点的左子树和右子树，
     * 依次递归下去，该递归算法只有一个出口，那就是输入数据等于-1
     */
```



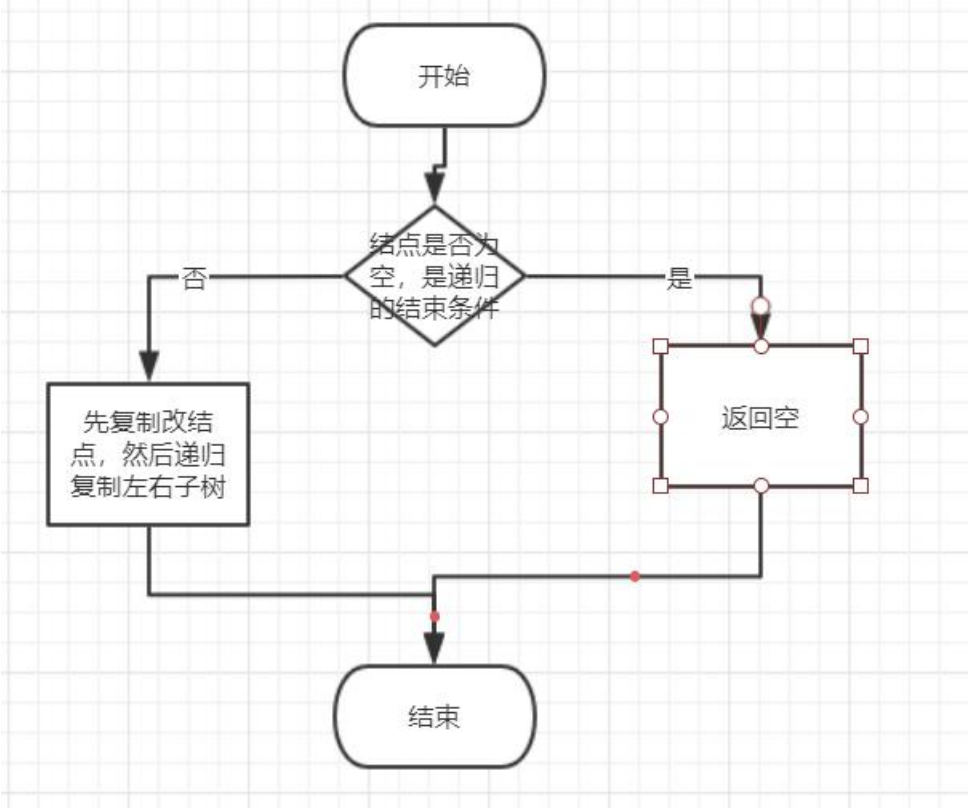

```
void BinTree<T>::destroy(BinTreeNode<T> *&tree)
{
    /*
     * 摧毁二叉树
     * 首先判断根结点是不是为空，如果不为空，利用递归，先删除左右子树，然后再删除结点本身
     * 如果全部删完了，结束递归
     */
```



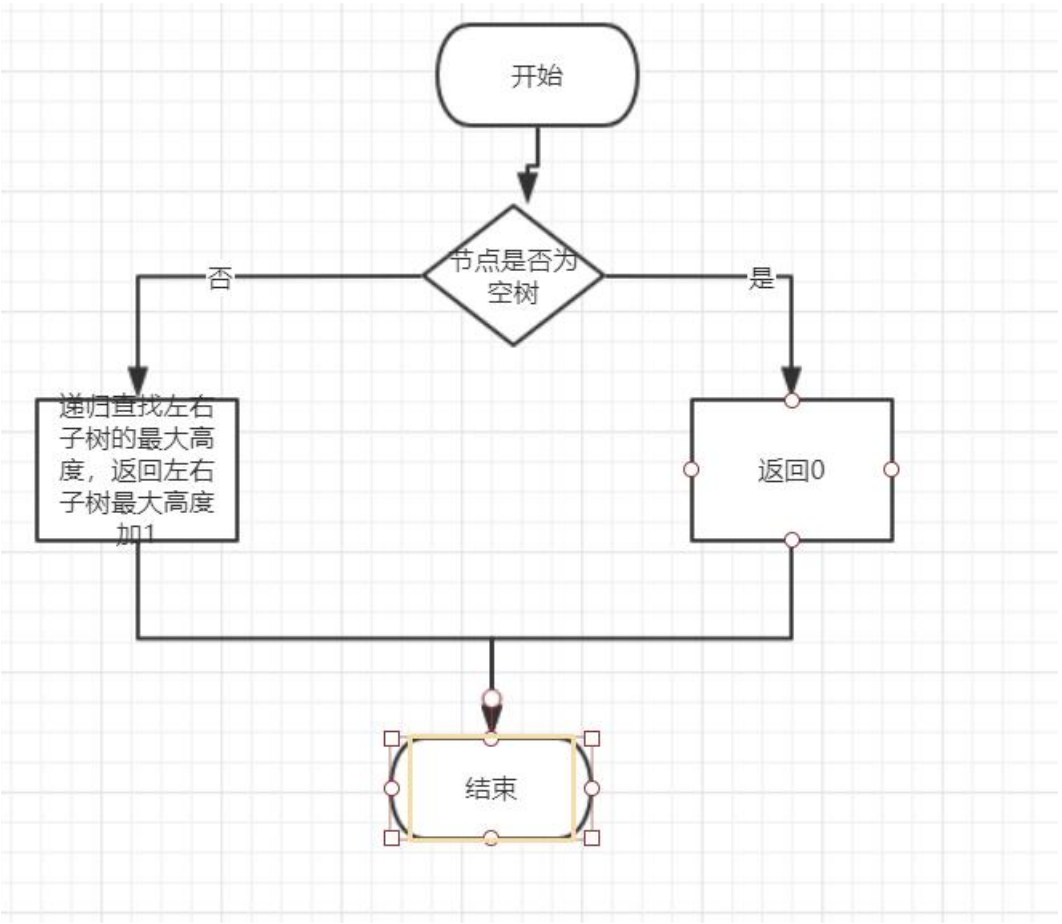

```
BinTreeNode<T>* BinTree<T>::Find(BinTreeNode<T>* &tree,T& x)
{
    /*
     * 根据 x 查找从树根开始依次往下找，如果找到了，返回结点地址，这个是递归的第一个出口
     * 如果该节点不是的值不是 x，则继续往下递归查找，找该节点的左右子树里面是否有 x
     * 没有找到返回空，这个是递归的第二个出口
     */
```



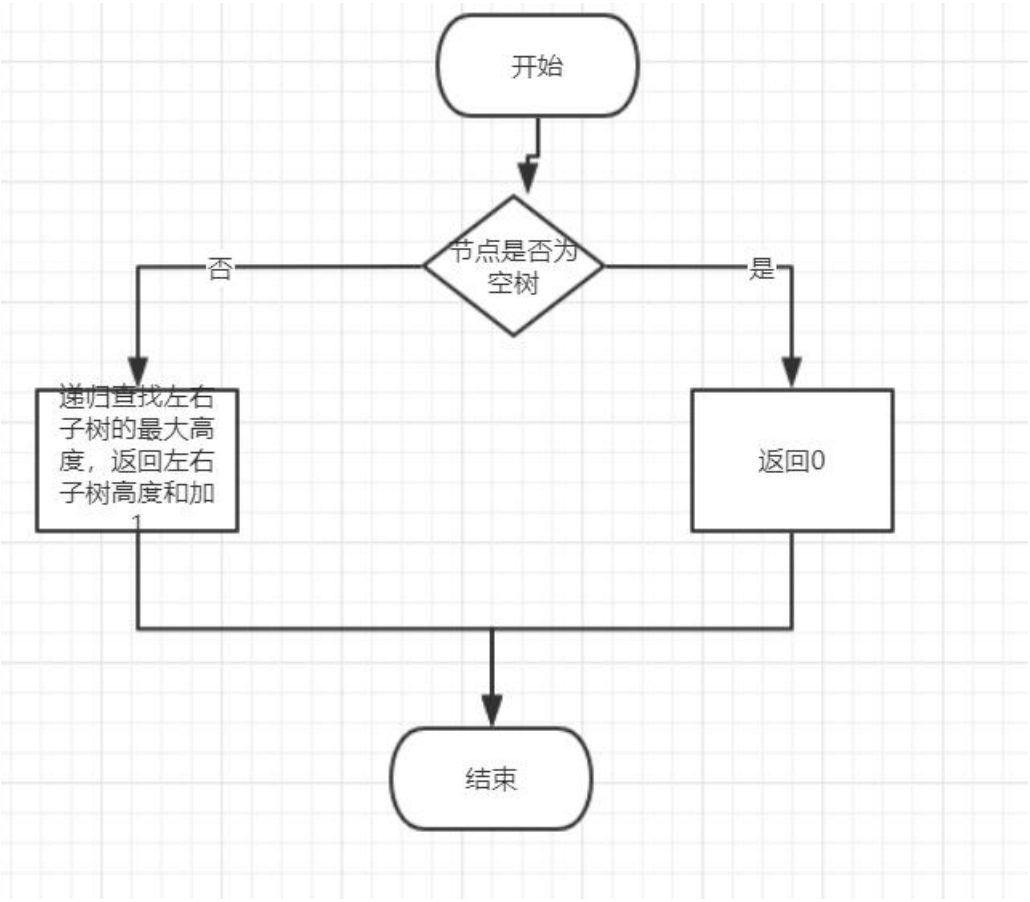

```
BinTreeNode<T>* BinTree<T>::Copy(BinTreeNode<T>*origin)
{
    /*
     * 树的复制，首先判断原来树是不是为空树，如果是空树直接返回空指针，这个也是这个递归算法的出口
     * 如果不是，首先定义一个结点，然后将原来树根的值复制给该节点，
     * 然后将原来左右子树赋值给现在的新书的左右子树，通过递归实现
     */
```



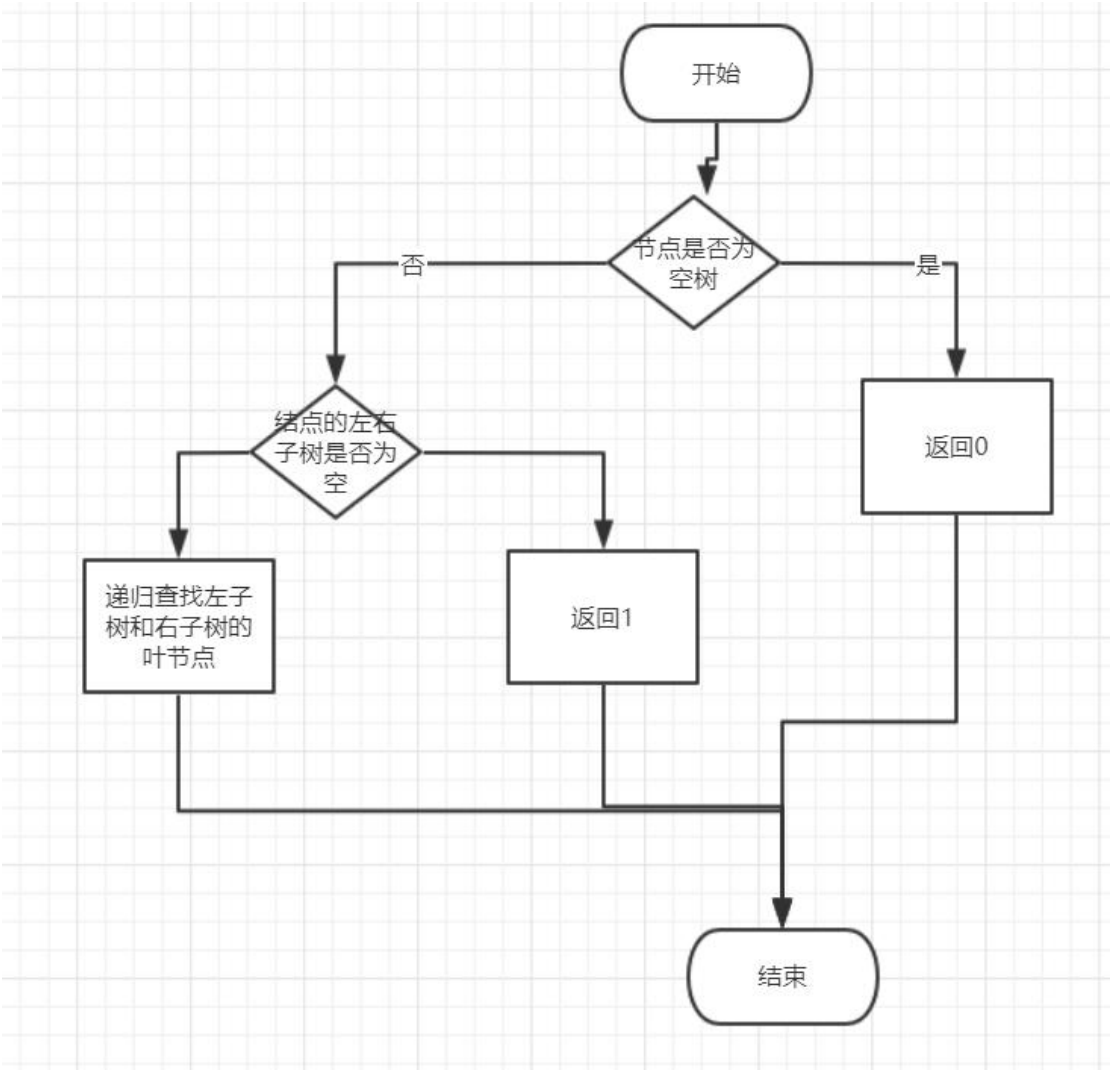

```
int BinTree<T>::Height(BinTreeNode<T>*tree)
{
    /*
     * 计算树的高度，即是树的层数
     * 首先判断树是否为空，这个是递归的出口
     * 如果是空直接返回 0;
     * 如果不是空，应该返回根节点左右子树的最大高度加 1
     */
```



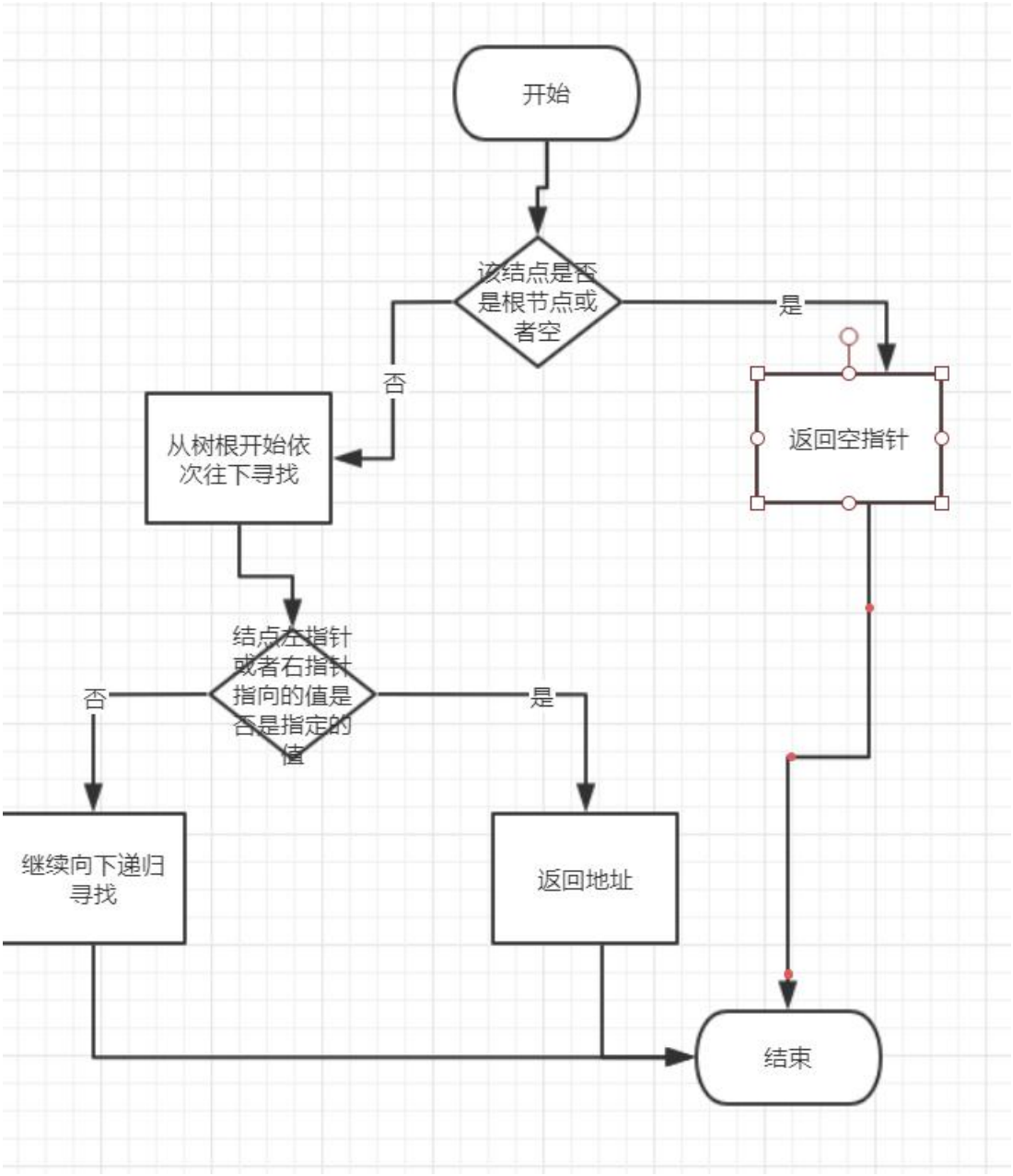

```
int BinTree<T>::Size(BinTreeNode<T>*tree)
{
    /*
     * 树的所有节点数
     * 如果数为空返回值 0，这个也是递归的出口
     * 如果不为空在，返回树的左右子树相加的值再加根节点，
     */
```



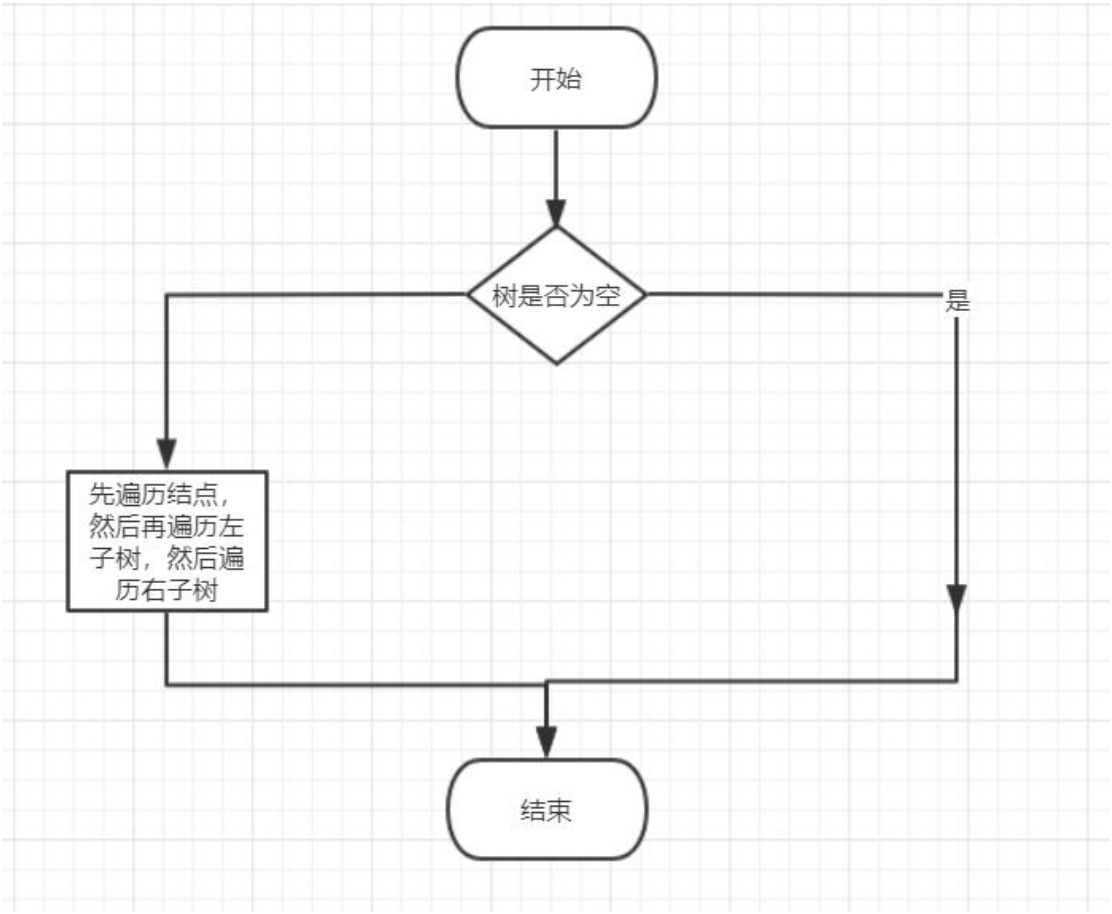

```
int BinTree<T>::LeavesCount(BinTreeNode<T> *tree)
{
    /*
     * 叶节点的个数
     * 如果树为空，返回 0，这个是递归的第一个出口
     * 如果结点的左右子树都为空，则说明该节点就是叶节点，返回值 1，这个是递归的第二个出口
     * 否则用递归，求出左子树的叶节点和右子树的叶节点，并且相加
     */
```



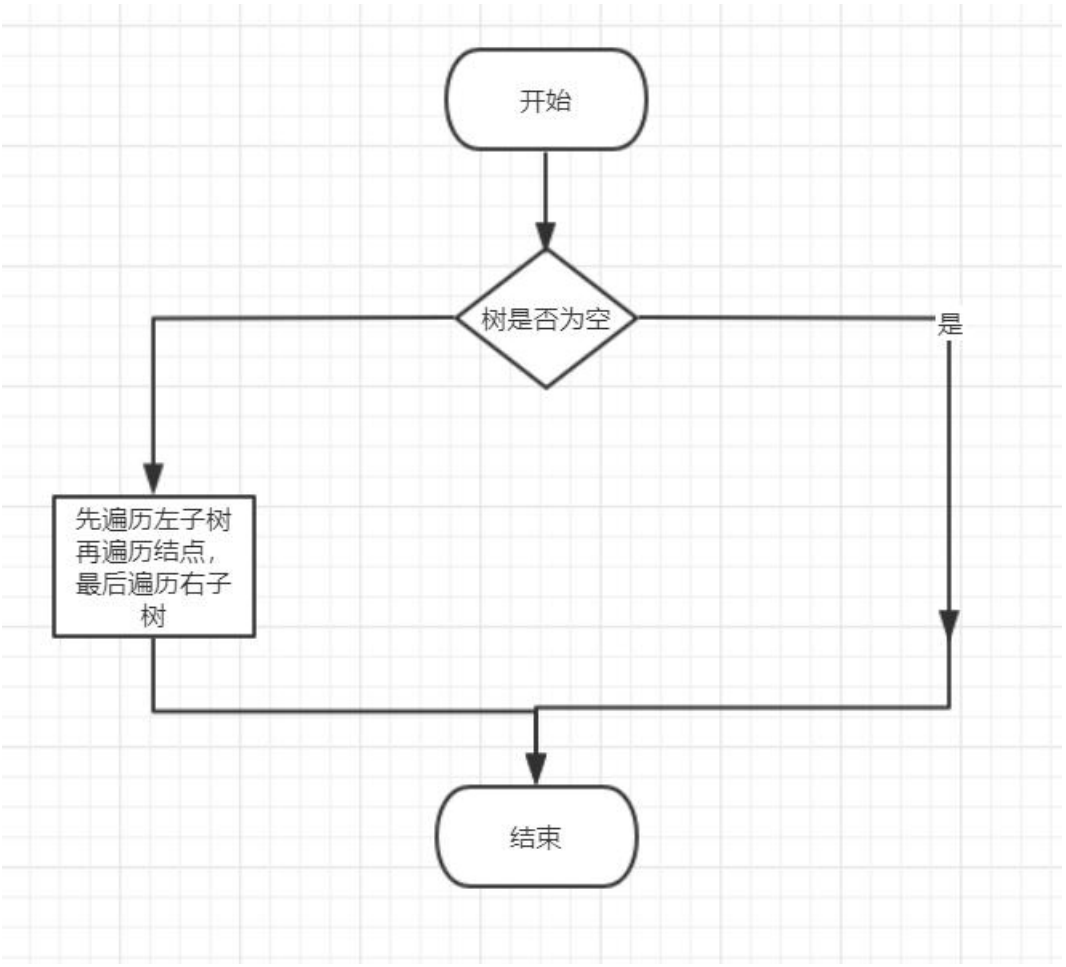

```
BinTreeNode<T>* BinTree<T>::Parent(BinTreeNode<T>
*tree,BinTreeNode<T>* now)
{
    /*
     * 找到指定结点的双亲结点，
     * 从树的根结点开始，如果根节点为空或者，指定结点为根节点，返回空指针，这个是递归的第一个出口
     * 如果树的结点的左子树或者右子树的结点时指定的结点，就说明该结点就是指定结点的双亲结点
     * 然后返回结点地址，这个是第二个出口
     * 如果当前结点不是以上情况，则从当前结点的左子树和右子树去递归寻找
     */
```



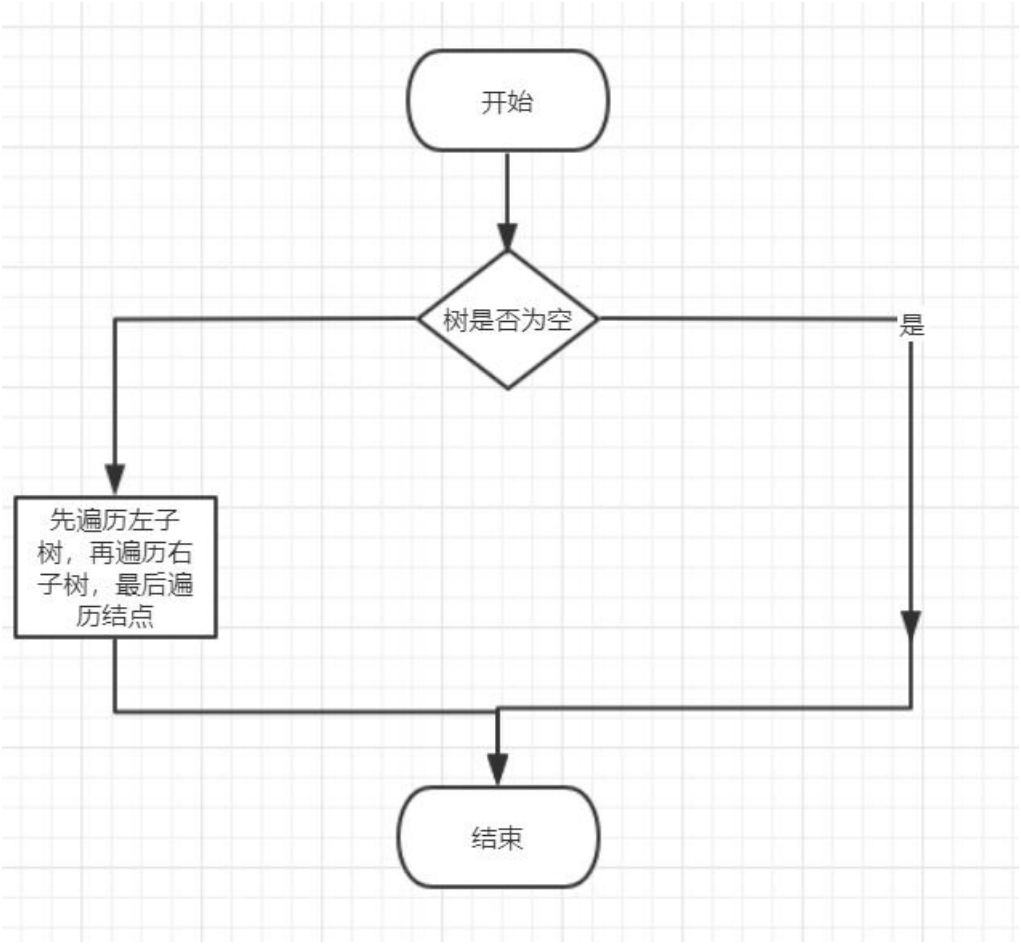

```
void BinTree<T>::preOrder(BinTreeNode<T> *tree)
{
    /*
     * 前序遍历的递归形式，
     * 如果树为空结束递归，
     * 先输出结点，在输出左子树，然后在输出右子树
     */
```




```
void BinTree<T>::inOrder(BinTreeNode<T>* tree)
{
    /*
     * 前序遍历的递归形式，
     * 如果树为空结束递归，
     * 先输出左子树，再输出结点，然后在输出右子树
     */
```




```
void BinTree<T>::postOrder(BinTreeNode<T>*tree)
{
    /*
     * 前序遍历的递归形式，
     * 如果树为空结束递归，
     * 先输出左子树，在输出右子树，最后输出结点
     */
```




```
void BinTree<T>::PreOrder(BinTreeNode<T>*tree)
{
    /*
     * 前序遍历的非递归算法，主要就是利用栈的存储结构来存储结点地址实现
     * 第一种，首先定义一个 p 指针，入栈一个空指针
     * 先输出根节点，如果该节点有右子树的根结点，先将右子树结点入栈，
     * 如果有左子树，将 p 指针指向左子树的根节点，然后输出 p 指针
     * 如果左子树遍历完全了，再遍历右子树，将右子树的结点依次出栈
     *
     * 第二种，首先将根结点入栈
     * 如果栈为空结束循环，将 p 出栈，并且输出
     * 如果右子树存在，先将该节点的右子树根节点入栈
     * 如果该节点的左子树存在，将该节点的左子树根结点入栈
     * 然后下一轮循环就会先输出左子树结点
     * 右子树就依次存留在栈空间中
     */
```

```
void BinTree<T>::InOrder(BinTreeNode<T> *tree)
{
    /*
     * 中序遍历的非递归遍历实现
     * 先通过栈存储存储节点地址
     * 首先将所有左子树结点入栈
     * 然后将最后一个左子树结点出栈，并且输出
     * 然后去寻找出栈的第一个结点的右子树
     * 如果没有继续出栈，如果右子树，然后继续去寻找右子树的左子树，继续入栈，然后出栈
     */
```

```
void BinTree<T>::PostOrder(BinTreeNode<T>* tree)
{
    /*
     *定义两个栈，用于存储结点的地址值
     * 第二个栈是用于存储最后输出结果的地址值
     * 第一个栈是是过度，将第二个栈中的输出全部换成需要想要的输出结果
     * 由于是先输出左子树-》右子树-》根节点，所以先入栈 1 的是根节点，然后是出栈进入栈 2，只要栈 1 为空，结束
     * 然后将左子树入栈 1，右子树入栈 1，下一次循环出栈的是右子树，然后入栈这个结点的左右子树，依次类推
     */
```

```
void BinTree<T>::NodeLevel(BinTreeNode<T>* tree,T &x,int lev,int &level)
{
    /*
     * 求出指定节点所在的层数
     * 如果数为空，返回 0，是第一个递归的出口
     * 如果不为空，如果当前节点的值等于指定值，层数等于递归层数加 1，是第二个递归出口
     * 如果层数一直是 0，说明还没有找到元素，继续在左子树和右子树递归寻找，递归层数加一，
     */
```

五、编程环境与实验步骤

（1）编程环境

主要是操作系统、编程工具软件

主要操作系统：

Windows 10

主要编程工具：

Qt Creator

（2）实验步骤

只说明程序相关的各种文件创建步骤及文件的作用，不需说明文件的具体内容。

Bintree.h　　二叉树的头文件

Bintree.cpp　　二叉树的实现文件

Stack.h　　栈的头文件

Stack.cpp　　栈的实现文件

Main.cpp　　测试文件

BintreeData.txt　　二叉树的数据文件

（3）编译参数

若有特殊的编译参数设置，需说明详细步骤。

若无特殊的编译参数设置，则只需简单说明操作步骤。

六、实现代码

主要功能的实现代码

```
#include "bintree.h"
#include "stack.h"
#include "stack.cpp"
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;
template <typename T>
void BinTree<T>::CreateBinTree(BinTreeNode<T> *&tree)
{
    /*
     * 创建一个树，首先传递一个引用指针参数，不能只是指针，必须要引用
     * 输入数据，以-1 结束也就是 stop 变量
     * 如果不是-1，利用前序遍历新建一个树，先新建一个结点，然后再创建该节点的左子树和右子树，
     * 依次递归下去，该递归算法只有一个出口，那就是输入数据等于-1
     */
```

```
    T item;
    cin>>item;
    if(item!=stop)
    {
        tree= new BinTreeNode<T>(item);
        //tree->date=item;
        //cout<<tree->date<<endl;
        //cout<<"00000"<<endl;
        if(tree==nullptr)
        {
            cerr<<"分配内存错误！！！"<<endl;
            return;
        }
        CreateBinTree(tree->lchild);
       // cout<<"11111"<<endl;
        CreateBinTree(tree->rchild);
    }
    else
        tree=nullptr;
}
template <typename T>
void BinTree<T>::destroy(BinTreeNode<T> *&tree)
{
    /*
     * 摧毁二叉树
     * 首先判断根结点是不是为空，如果不为空，利用递归，先删除左右子树，
然后再删除结点本身
     * 如果全部删完了，结束递归
     */
    if(tree!=nullptr)
    {
        destroy(tree->lchild);
        destroy(tree->rchild);
        delete tree;
    }
}
template <typename T>
BinTreeNode<T>* BinTree<T>::Find(BinTreeNode<T>* &tree,T& x)
{
    /*
     * 根据 x 查找从树根开始依次往下找，如果找到了，返回结点地址，这个
是递归的第一个出口
     * 如果该节点不是的值不是 x，则继续往下递归查找，找该节点的左右子
树里面是否有 x
```

```
     * 没有找到返回空，这个是递归的第二个出口
     */
    if(tree==nullptr)
        return nullptr;
    if(tree->date==x)
        return tree;
    BinTreeNode<T>* p;
    if((p=Find(tree->lchild,x))!=nullptr)
        return p;
    else
        return Find(tree->rchild,x);
}
template <typename T>
BinTreeNode<T>* BinTree<T>::Copy(BinTreeNode<T>*origin)
{
    /*
     * 树的复制，首先判断原来树是不是为空树，如果是空树直接返回空指针，
这个也是这个递归算法的出口
     * 如果不是，首先定义一个结点，然后将原来树根的值复制给该节点，
     * 然后将原来左右子树赋值给现在的新书的左右子树，通过递归实现
     */
    if(origin==nullptr)
        return nullptr;
    BinTreeNode<T>* newtree=new BinTreeNode<T>;
    newtree->date=origin->date;
    newtree->lchild=Copy(origin->lchild);
    newtree->rchild=Copy(origin->rchild);
    return newtree;
}
template <typename T>
int BinTree<T>::Height(BinTreeNode<T>*tree)
{
    /*
     * 计算树的高度，即是树的层数
     * 首先判断树是否为空，这个是递归的出口
     * 如果是空直接返回 0;
     * 如果不是空，应该返回根节点左右子树的最大高度加 1
     */
    if(tree==nullptr)
        return 0;
    int i=Height(tree->lchild);
    int j=Height(tree->rchild);
    return (i>j) ? i+1:j+1;
}
```

```
template <typename T>
int BinTree<T>::Size(BinTreeNode<T>*tree)
{
    /*
     * 树的所有节点数
     * 如果数为空返回值 0，这个也是递归的出口
     * 如果不为空在，返回树的左右子树相加的值再加根节点，
     */
    if(tree==nullptr)
        return 0;
    return 1+Size(tree->lchild)+Size(tree->rchild);
}
template <typename T>
int BinTree<T>::LeavesCount(BinTreeNode<T> *tree)
{
    /*
     * 叶节点的个数
     * 如果树为空，返回 0，这个是递归的第一个出口
     * 如果结点的左右子树都为空，则说明该节点就是叶节点，返回值 1，这个是递归的第二个出口
     * 否则用递归，求出左子树的叶节点和右子树的叶节点，并且相加
     */
    if(tree==nullptr)
        return 0;
    else if(tree->lchild==nullptr&&tree->rchild==nullptr)
        return 1;
    else
    {
        return LeavesCount(tree->rchild)+LeavesCount(tree->lchild);
    }
}
template <typename T>
int BinTree<T>::NodeDeg(BinTreeNode<T>*now)
{
    /*求一个结点的度，通常度指的是有多少个子树，在二叉树中，即是判断左右子树的存在情况
     * 都不存在，返回 0，存在一个返回 1，存在两个返回 2
     */
    if(now==nullptr)
        return 0;
    if(now->lchild!=nullptr&&now->rchild!=nullptr)
        return 2;
    if(now->lchild!=nullptr)
        return 1;
```

```
    if(now->rchild!=nullptr)
        return 1;
    else
        return 0;
}
template <typename T>
BinTreeNode<T>* BinTree<T>::Parent(BinTreeNode<T>
*tree,BinTreeNode<T>* now)
{
    /*
     * 找到指定结点的双亲结点，
     * 从树的根结点开始，如果根节点为空或者，指定结点为根节点，返回空指针，这个是递归的第一个出口
     * 如果树的结点的左子树或者右子树的结点时指定的结点，就说明该结点就是指定结点的双亲结点
     * 然后返回结点地址，这个是第二个出口
     * 如果当前结点不是以上情况，则从当前结点的左子树和右子树去递归寻找
     */
    if(tree==nullptr||now==root)
        return nullptr;
    else if(tree->lchild==now||tree->rchild==now)
        return tree;
    BinTreeNode<T> *p;
    if((p=Parent(tree->lchild,now))!=nullptr)
        return p;
    else
    {
        p=Parent(tree->rchild,now);
        return p;
    }
}
template <typename T>
BinTreeNode<T>* BinTree<T>::LeftChild(BinTreeNode<T> *now)
{
    return (now==nullptr) ? nullptr:now->lchild;
}
template <typename T>
BinTreeNode<T>* BinTree<T>::RightChild(BinTreeNode<T>*now)
{
    return (now==nullptr) ? nullptr:now->rchild;
}
template <typename T>
void BinTree<T>::Traverse(BinTreeNode<T> *tree)
```

```
{
    if(tree!=nullptr)
    {
        cout<<tree->date<<" ";
        Traverse(tree->lchild);
        Traverse(tree->rchild);
    }
}
template <typename T>
void BinTree<T>::visit(BinTreeNode<T>* tree)
{
    cout<<tree->date<<"    ";
}
template <typename T>
void BinTree<T>::preOrder(BinTreeNode<T> *tree)
{
    /*
     * 前序遍历的递归形式，
     * 如果树为空结束递归，
     * 先输出结点，在输出左子树，然后在输出右子树
     */
    if(tree==nullptr)
        return;
    visit(tree);
    preOrder(tree->lchild);
    preOrder(tree->rchild);
}
template <typename T>
void BinTree<T>::inOrder(BinTreeNode<T>* tree)
{
    /*
     * 前序遍历的递归形式，
     * 如果树为空结束递归，
     * 先输出左子树，再输出结点，然后在输出右子树
     */
    if(tree==nullptr)
        return;
    inOrder(tree->lchild);
    visit(tree);
    inOrder(tree->rchild);
}
template <typename T>
void BinTree<T>::postOrder(BinTreeNode<T>*tree)
{
```

```
    /*
     * 前序遍历的递归形式，
     * 如果树为空结束递归，
     * 先输出左子树，在输出右子树，最后输出结点
     */
    if(tree==nullptr)
        return;
    postOrder(tree->lchild);
    postOrder(tree->rchild);
    visit(tree);
}
template <typename T>
void BinTree<T>::PreOrder(BinTreeNode<T>*tree)
{
    /*
     * 前序遍历的非递归算法，主要就是利用栈的存储结构来存储结点地址实现
     * 第一种，首先定义一个 p 指针，入栈一个空指针
     * 先输出根节点，如果该节点有右子树的根结点，先将右子树结点入栈，
     * 如果有左子树，将 p 指针指向左子树的根节点，然后输出 p 指针
     * 如果左子树遍历完全了，再遍历右子树，将右子树的结点依次出栈
     *
     * 第二种，首先将根结点入栈
     * 如果栈为空结束循环，将 p 出栈，并且输出
     * 如果右子树存在，先将该节点的右子树根节点入栈
     * 如果该节点的左子树存在，将该节点的左子树根结点入栈
     * 然后下一轮循环就会先输出左子树结点
     * 右子树就依次存留在栈空间中
     */
    Stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p=tree;
    S.Push(nullptr);
    while(p!=nullptr)
    {
        visit(p);
        if(p->rchild!=nullptr)
            S.Push(p->rchild);
        if(p->lchild!=nullptr)
            p=p->lchild;
        else
            S.Pop(p);
    }
//    S.Push(p);
//    while(!S.IsEmpty())
```

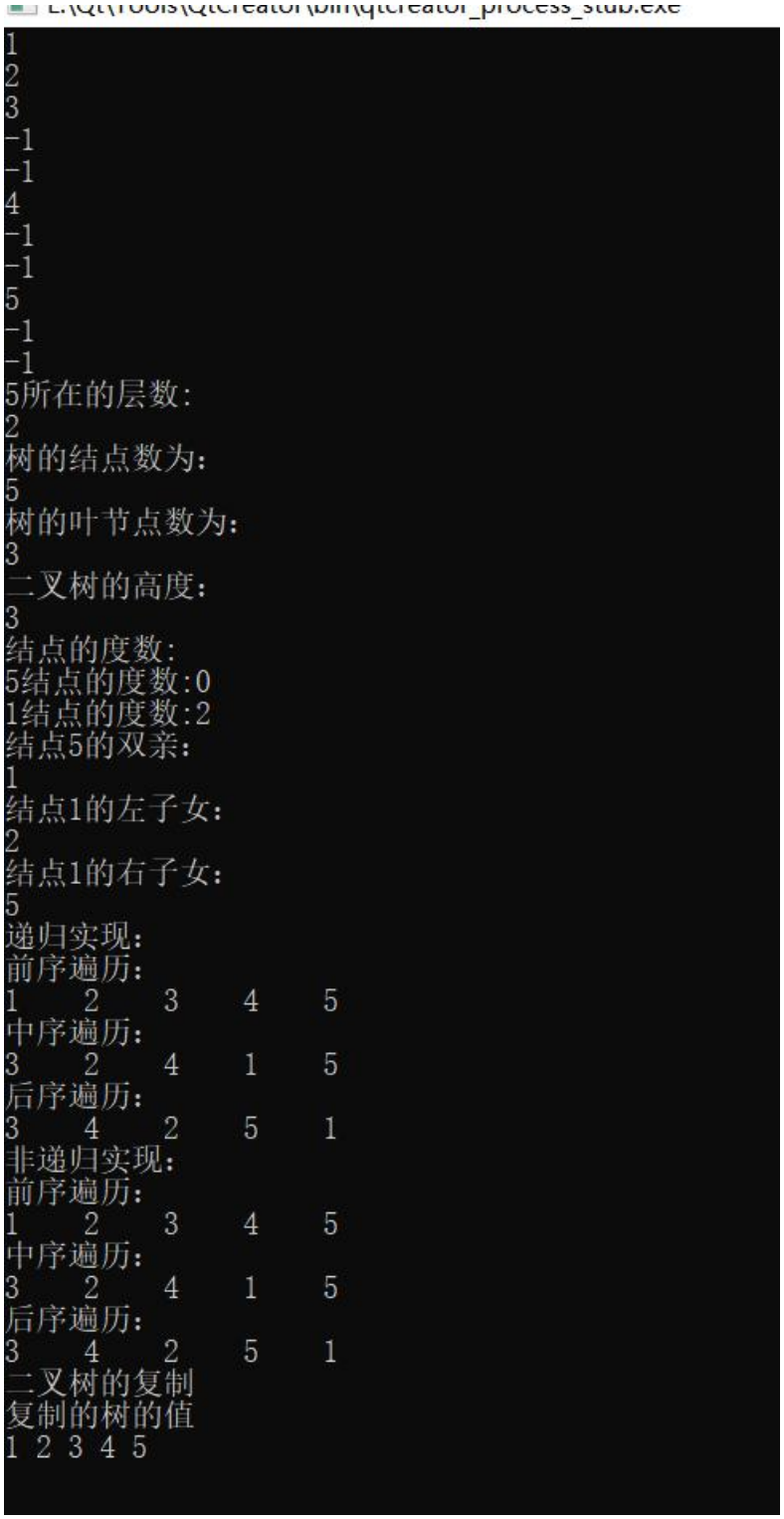
```
//    {
//        S.Pop(p);
//        visit(p);
//        if(p->rchild!=nullptr)
//            S.Push(p->rchild);
//        if(p->lchild!=nullptr)
//            S.Push(p->lchild);
//    }
}
template <typename T>
void BinTree<T>::InOrder(BinTreeNode<T> *tree)
{
    /*
     * 中序遍历的非递归遍历实现
     * 先通过栈存储存储节点地址
     * 首先将所有左子树结点入栈
     * 然后将最后一个左子树结点出栈，并且输出
     * 然后去寻找出栈的第一个结点的右子树
     * 如果没有继续出栈，如果右子树，然后继续去寻找右子树的左子树，继续入栈，然后出栈
     */
    Stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p=tree;
    do{
        while(p!=nullptr)
        {
            S.Push(p);
            p=p->lchild;
        }
        if(!S.IsEmpty())
        {
            S.Pop(p);
            visit(p);
            p=p->rchild;
        }
    }while(p!=nullptr||!S.IsEmpty());
}
template <typename T>
void BinTree<T>::PostOrder(BinTreeNode<T>* tree)
{
    /*
     *定义两个栈，用于存储结点的地址值
     * 第二个栈是用于存储最后输出结果的地址值
     * 第一个栈是是过度，将第二个栈中的输出全部换成需要想要的输出结果
```

```
     * 由于是先输出左子树-》右子树-》根节点，所以先入栈 1 的是根节点，然后是出栈进入栈 2，只要栈 1 为空，结束
     * 然后将左子树入栈 1，右子树入栈 1，下一次循环出栈的是右子树，然后入栈这个结点的左右子树，依次类推
     */
    Stack<BinTreeNode<T>*> S1;
    Stack<BinTreeNode<T>*> Res;
    BinTreeNode<T> *p=tree;
    S1.Push(p);
    while(!S1.IsEmpty())
    {
        S1.Pop(p);
        Res.Push(p);
        if(p->lchild!=nullptr)
            S1.Push(p->lchild);
        if(p->rchild!=nullptr)
            S1.Push(p->rchild);
    }
    while (!Res.IsEmpty()) {
        Res.Pop(p);
        visit(p);
    }
}
template <typename T>
void BinTree<T>::NodeLevel(BinTreeNode<T>* tree,T &x,int lev,int &level)
{
    /*
     * 求出指定节点所在的层数
     * 如果数为空，返回 0，是第一个递归的出口
     * 如果不为空，如果当前节点的值等于指定值，层数等于递归层数加 1，是第二个递归出口
     * 如果层数一直是 0，说明还没有找到元素，继续在左子树和右子树递归寻找，递归层数加一，
     */
    if(tree==nullptr)
        level=0;
    else
    {
        if(tree->date==x)
            level=lev+1;
        else
        {
            if(level==0)
```

```
                NodeLevel(tree->lchild,x,lev+1,level);
            if(level==0)
                NodeLevel(tree->rchild,x,lev+1,level);
        }
    }
}
//template <typename T>
//istream& operator>>(istream& in, BinTreeNode<T>&c)
//{
//    CreateBinTree(in,c.root);
//    return in;
//}
//template <typename T>
//ostream& operator<<(ostream& out,BinTreeNode<T>& c)
//{
//    out<<"二叉树前序遍历"<<endl;
//    c.Traversal(out,c.root);
//    out<<endl;
//    return out;
//}
template <typename T>
BinTree<T>::BinTree(BinTree<T>&s)
{
    root=Copy(s.root);
}
```

七、测试结果与说明

至少完成功能测试，使用测试数据测试相关功能是否符合设计要求。



八、实验分析

（1）算法的性能分析

主要针对增加、删除、搜索等算法。

没有增加和删除结点的算法，搜索算法的时间复杂度是 O(n)

（2）数据结构的分析

通过性能分析总结此种存储结构的优缺点，并说明其适用场景。

二叉树缺点：

1. 一共定义了 2*n 个指针，只有 n-1 个指针用到了，浪费了部分储存空间

2. 二叉树中有大量的递归算法，如果树的结点太多，递归层次过深，可能会爆栈。

二叉树的优点;

1. 有多个分支，可以存储有多个分支的数据

2. 数据与数据之间有祖先，上级和下属的关系，整体和分支的关系。

适用场景：

在计算机中有广泛的应用，文件系统和数据库系统中，树是组织信息的重要形式之一，在编译系统中，树用来表示源程序的语法结构，在算法设计与分析中，树是刻画程序动态性质的工具。

九、实验总结

主要针对本实验的分析、设计、实现、测试等环节进行总结，包含收获与不足，此部分的阐述应较为详细。

本实验采用了面向对象的思想编写了模板类，首先设计了一个树的结点类，因为树是有一个一个结点组成的，结点中定义了两个指针，分别指向左子树和右子树。然后在定义了一个二叉树类，用于保存指向树的根节点的指针，必须通过这个指针来访问整个树。

实现过程中，一开始拿到这个题目的时候，脑子一片空白，后面看了树上的代码和上课听讲之后，发现自己应该可以写出来。其中非递归前序遍历，非递归中序遍历，非递归后续遍历，都是通过查阅资料后写的，当时看到这个非递归实现这个遍历，直接有点不知所措，看了资料之后，好像懂了，但是有很多细节还是没有想明白，还需要进一步思考。二叉树用到了大量的递归算法，如果将这里面的算法全部改为非递归，感觉难度挺大的，不想去尝试了。

测试的时候我没有写文件操作，所以只能人工输入，我利用的是前序遍历的方法建树的，测试中我发现在求一个结点是第几层的时候，，没有输出值，原来是我想通过引用返回值，然后输出，忘了用引用，只传递了值，一开始建树的时候也是如此，没有用引用指针，导致建的树为空。

本次实验让我更加清楚的认识到了递归算法的思想，本次实验基本全部算法都是递归。递归可以减少代码量，递归的重点是递归函数的设计，每个递归函数都必须要有递归出口。
附录

参考文献：
1.
2.
3.