HUNAN UNIVERSITY

数据结构与算法分析

实验报告

学生姓名/学号　梅炳寅　202108010206

专业班级　计科 2102

指导老师　夏艳

2022 年 4 月 19 日

目录

1.问题分析

1.1 处理的对象（数据）

处理的对象为用字符表示的前序遍历二叉树，每个输入文件的第一行为二叉树 A 的前序遍历顺序表示法（N≤30）。第二行为二叉树 B 的前序遍历顺序表示法。其中用“#”代表空指针 NULL。

1.2 实现的功能

对输入的两棵二叉树 A 和 B，判断 B 是不是 A 的子树

1.3 处理后的结果如何显示

yes 代表二叉树 B 就是二叉树 A 的一棵子树。
no 代表二叉树 B 不是二叉树 A 的一棵子树。

1.4 请用题目中样例，详细给出样例求解过程。

【样例输入 1】

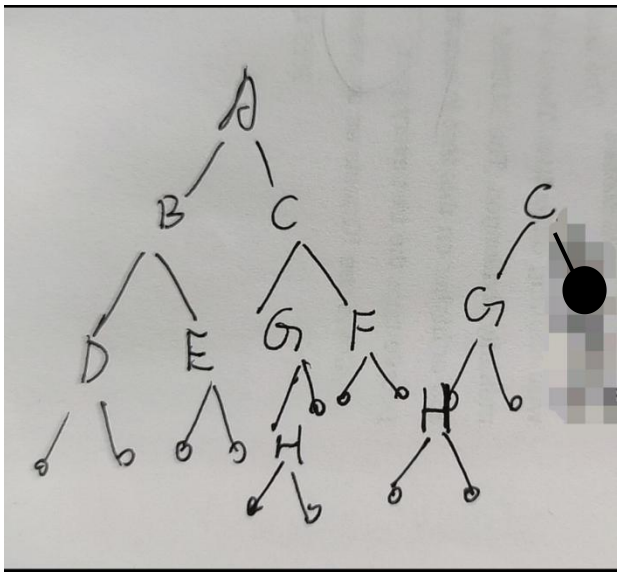
AB##C##
AB##C##

1、通过前序遍历读入树的各节点，存入树结构中；
2、通过前序遍历在 A 树中寻找 B 树的根节点 ，找到为 A；
3、通过前序遍历，分别将 A 中的 A 和 B 中的 A 的子树分别存入容器中；
4、比较容器内元素个数，发现 7==7，即元素个数相同；
5、逐个比较容器内各元素的值，发现全部相同，返回结果 yes。

【样例输入 2】

ABD##E##CGH###F##
CGH####

1、通过前序遍历读入树的各节点，存入树结构中；
2、通过前序遍历在 A 树中寻找 B 树的根节点 ，找到为 C；
3、通过前序遍历，分别将 A 中的 C 和 B 中的 C 的子树分别存入容器中；
4、比较容器内元素个数，发现 9！=7，即元素个数不同，返回结果 no；



【样例输入 3】

ABD##E##CGH###F##

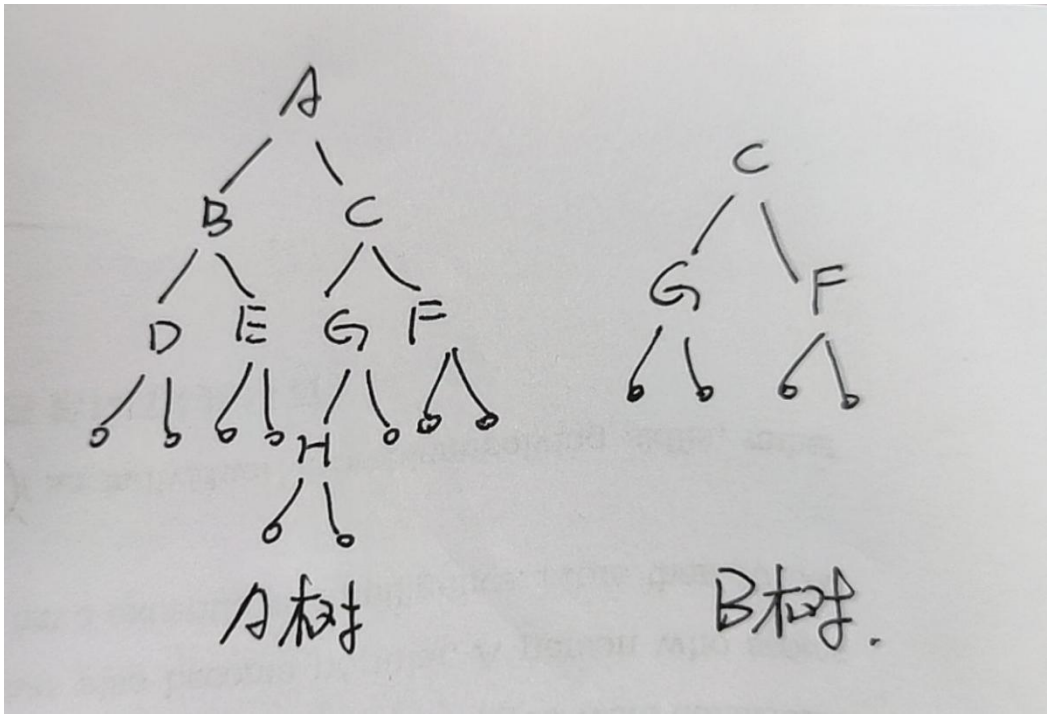
CG##F##

1、通过前序遍历读入树的各节点，存入树结构中；

2、通过前序遍历在 A 树中寻找 B 树的根节点 ，找到为 C；

3、通过前序遍历，分别将 A 中的 C 和 B 中的 C 的子树分别存入容器中；

4、比较容器内元素个数，发现 9！=7，即元素个数不同，返回结果 no；



2.数据结构和算法设计

2.1 抽象数据类型设计

```
template<typename    E>
class    BinNode//结点类
{
private:
        BinNode*lc;//左孩子
        BinNode*rc;//右孩子
        E    elem;
public:
        BinNode();//默认构造函数，设置左右孩子为空
        BinNode(E    tmp,    BinNode*l=NULL,    BinNode*r=NULL);
//带参构造函数
        BinNode*left();//返回左孩子
        BinNode*right();//返回右孩子
        void    setLeft(BinNode*l);//设置左孩子
```

```
        void    setRight(BinNode*r);//设置右孩子
        void    setValue(E    tmp);//设置当前结点的值
        E    getValue();//获得当前结点的值
};

template<typename    E>
class    BinTree//二叉树类
{
private:
        BinNode<E>*root;//根结点
        void    clear(BinNode<E>*r);//清空二叉树
public:
        BinTree();//默认构造函数
        ~BinTree();//析构函数
        bool    BinTreeEmpty();//判断二叉树是否为空
        BinNode<E>*getRoot();//获得根节点
        void    setRoot(BinNode<E>*r);//设置根节点
//下面的函数是对外的函数，所以内部还会有一些同名的函数，但是参数列表不一样，实现数据的封装，外部的调用不会涉及到内部的数据对象
        void    preOrder(void(*visit)(BinNode<E>*node));
//先序遍历，传入相对应的访问函数即可对该当前结点实现不同功能的访问（本程序为输出）
        bool    find(E    e);//查找二叉树中是否存在名为 e 的结点
        BinNode<E>* find(BinNode<E>* tmp, E e);
//查找二叉树中是否含有某个名为 e 的结点
};
```

2.2 物理数据对象设计（不用给出基本操作的实现）

```
//BinaryTreeNode 二叉树的节点的实现
BinNode<E>::BinNode()//默认构造函数，设置左右孩子为空
BinNode<E>::BinNode(E    tmp,    BinNode*l,    BinNode*r)//带参构造函数
//模板函数的默认参数似乎必须在第一次声明的时候给出,也就是说，这里不能有两个=NULL，详情参见 https://blog.csdn.net/u013457310/article/details/89510406
BinNode<E>* BinNode<E>::left()//返回左孩子
BinNode<E>* BinNode<E>::right()//返回右孩子
void    BinNode<E>::setLeft(BinNode*l)//设置左孩子
void    BinNode<E>::setRight(BinNode*r)//设置右孩子
void    BinNode<E>::setValue(E    tmp)//设置当前结点的值
E    BinNode<E>::getValue()//获得当前结点的值

//BinaryTree 二叉树的实现
BinNode<E>* BinTree<E>::find(BinNode<E>* tmp, E e)
```

BinTree<E>::BinTree()//默认构造函数
BinTree<E>::~BinTree()//析构函数
BinNode<E>* BinTree<E>::getRoot()//获得根节点
void BinTree<E>::setRoot(BinNode<E>*r)//设置根节点

//下面的函数是对外的函数，所以内部还会有一些同名的函数，但是参数列表不一样，实现数据的封装，外部的调用不会涉及到内部的数据对象
bool BinTree<E>::find(E e)//查找二叉树中是否存在名为 e 的结点

//类外函数
BinNode<E>* creatBinaryTree(string s[], int& x, int n)
//建立树结构
void creatBinaryTree(BinTree<string>* BT)
//读入数据，为存入树结构做准备工作
void preOrder1(BinNode<E>* tmp, vector< string>& ve)
//通过前序遍历的方式，在 A 树中寻找 B 树的根节点 C
bool compare(BinNode<string>* s, BinNode<string>* s1)
//逐个比较取入容器中的 A 中 C 的子树与 B 中 C 的子树各个节点是否一致

2.3 算法思想的设计

记输入的两个二叉树为分别为 a1，a2
记 a2 的根节点对应的值为 h
1）先创建二叉树的类来储存输入的元素
2）a1 中进行遍历查找 h。
3）在未查找到 h，则输出‘no’并 return
4）若查到 h 则返回以 h 为根节点的二叉树记做 e
5）对 a2 和 e 分别进行先序遍历，并将遍历到的节点装入两个容器中，若遍历到的左节点或右节点为空则将‘#’装入容器
（若不进行插“#”的操作，不能保证两个二叉树前序遍历相同则两个二叉树相同）
6）对两个容器进行 size 的比较，
若不同则输出“no”并返回
若相同则对两个容器进行遍历逐个比较若有不同输出“no”并返回
直到遍历完容器中所有元素，此时说明两个容器元素相同，即两个二叉树相同，也说明 a2 为 a1 的子树，输出“yes”并 return

2.4 关键功能的算法步骤（不能用源码）

1、二叉树节点类与二叉树类的实现
这是本次实验的底层代码，二叉树节点类与二叉树的类，以及相关的部分成员函数，这些在实验三中有体现了，这里主要是将实验三中部分有用的代码进行迁移，并将无关的代码删除。主要是 preorder 前序遍历函数的改写，因为是对于该子树所有节点的一个保存，因此需要在前序遍历时将节点全部存入容器 vector 中，以便之后的比较。

2、在 A 树中寻找 B 树的根节点 C

这是本题的关键算法，倘若不能在 A 树中找到 B 树的根节点 C，那么说明一定不存在这样的一个树 B 作为 A 的子树。领悟这一点也是解答本题的关键。至于找到 C 之后对于两边的 C 的子树的储存与注意比较，就是自然而然的了。

3、对于容器中元素的逐一比较

考虑到有可能因为元素数量不同而本身就不可能是成立的，故需要在一开始就检查是否两个 vector 内元素个数相同，若不相同则直接退出。若相同，则用循环逐个检查每一个元素，若仍然相同，则返回答案。

4、特别注意"#"的意义

特别注意对于空节点的处理，因为对于仅有前序遍历，是没有办法唯一确定出一棵二叉树的，所以本题的特殊性在于它同时保存了每个空节点（也就是叶子节点）。若不进行插"#"的操作，不能保证两个二叉树前序遍历相同则两个二叉树相同。

3.算法性能分析

3.1 时间复杂度

该算法的时间复杂度是 O（n），下面是具体分析。由于大多是递归函数，我们主要采取功能的方法来分析。

1、准备部分：

creatBinaryTree();的两次调用，为 0(1)，但在 creatBinaryTree()中又调用了 creatBinaryTree<string>(s, now, n)，而这个的功能类似于遍历，是 0(n)的，故准备阶段的时间复杂度是 c1n；

2、在 A 树中查找 B 树的根节点 C：

BinNode<string>* tem = BT->find(BT->getRoot(), BT1->getRoot()->getValue());这句的作用在于在 A 树中查找 B 树的根节点 C。这相当于是遍历一遍，为 c2n;

3、分别存储 A 中 C 的子树与 B 中 C 的子树的各个节点：

这里在函数 compare 中调用了两次 preorder1 函数，preorder1 函数的功能是在前序遍历的同时以前序遍历的方式存储各个节点，这种遍历方式是 O(n)的，记作 c3n；

4、compare 函数主体的逐个比较：

for 循环逐个比较，显然为 O(n),记作 c4n;

综上所述，时间复杂度为

$$T = c1n + c2n + c3n + c4n = O(n)$$

3.2 空间复杂度

由于使用链表结构存储二叉树，而又不是 morris 算法遍历，仅仅使用递归来遍历，所以该算法的空间复杂度是 O（n）

4.不足与反思

在算法上本题应该已经是比较优化的了，包括采取先判断个数是否相同等方法来剪枝，达到了很好的优化效果。但在遍历上，有一种 morris 算法可以将空间复杂度降到 O(1)，可以考虑。此外如果还有其他不足或可优化的地方，请务必向我提出。