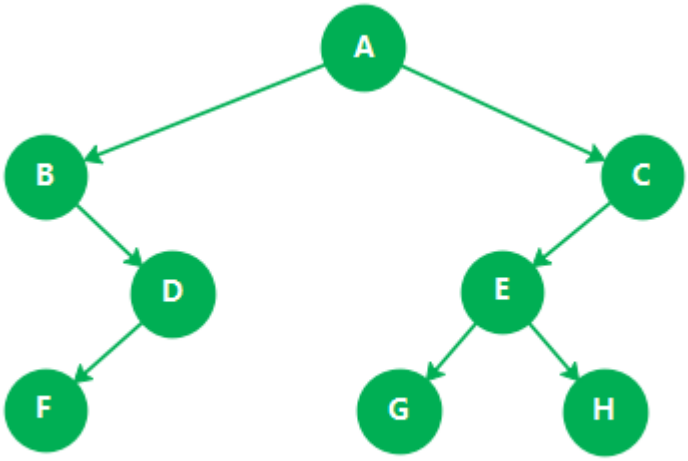
Homework:树和二叉树

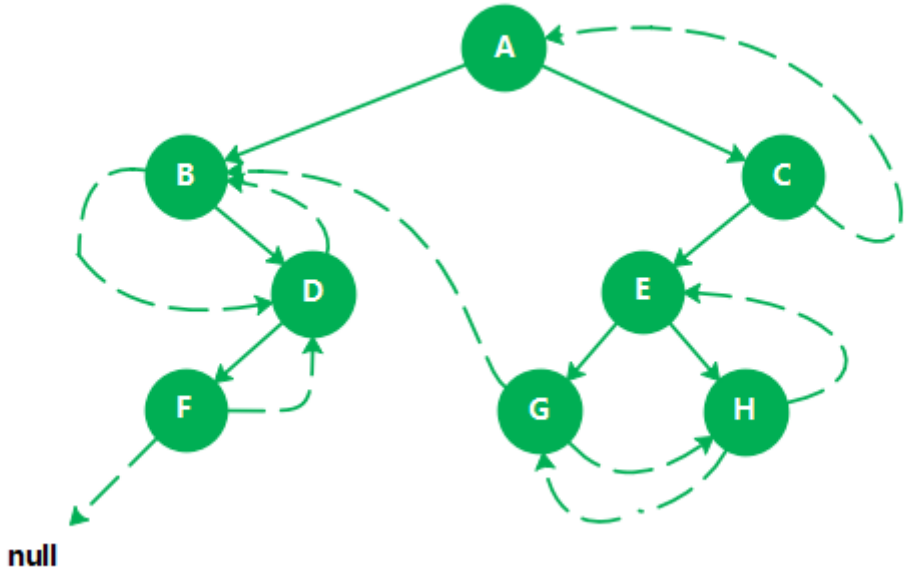
1.设一棵二叉树的先序序列:ABDFCEGH，中序序列:BFDAGEHC

（1）画出这棵二叉树。

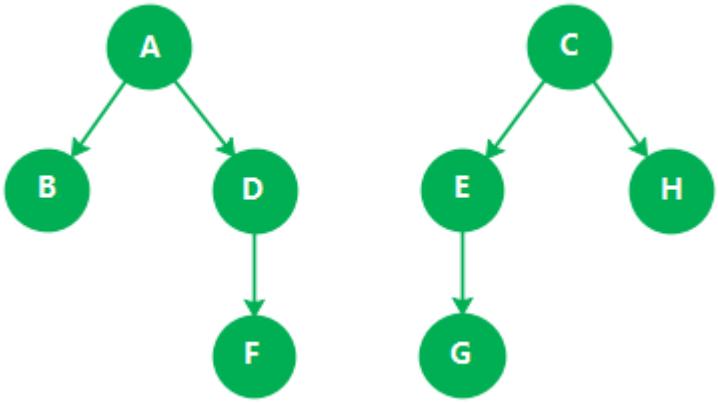


（2）画出这棵二叉树的后序线索树。

- 后续序列为：FDBGHECA



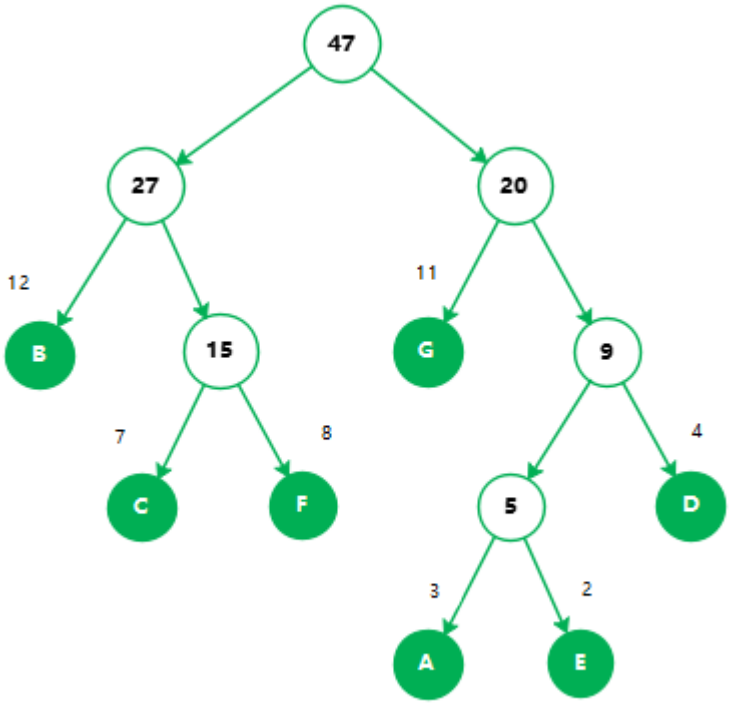
（3）将这棵二叉树转换成对应的树(或森林)。



2.已知下列字符A、B、C、D、E、F、G的权值分别为3、12、7、4、2、8，11，试填写出其对应哈夫曼树HT存储结构的初态和终态。

初态

终态



3.以二叉链表作为二叉树的存储结构，编写以下算法：

(1)统计二叉树的叶结点个数。

(2)判别两棵树是否相等。

(3)交换二叉树每个结点的左孩子和右孩子。

(4)设计二叉树的双序遍历算法(双序遍历是指对于二叉树的每一个结点来说，先访问这个结点，再按双序遍历它的左子树，然后再一次访问这个结点，接下来按双序遍历它的右子树)。

(5)计算二叉树最大的宽度(二叉的最大宽度是指二叉树所有层中结点个数的最大值)。

(6)用按层次顺序遍历二叉树的方法，统计树中度为1的结点数目。

(7)求任意二叉树中第一条最长的路径长度，并输出此路径上各结点的值。

(8)输出二叉树中从每个叶子结点到根结点的路径。

```
# include<iostream>
# include<queue>

using namespace std;

class BinaryTree{
public:
    typedef int value_type;

    BinaryTree()
        : _val(0)
        , _left(nullptr)
        , _right(nullptr)
    {}

    //(1)
    int leaf_num()
    {
        //是空结点
        if (this == nullptr)
        {
            return 0;
        }
        //是叶子
        if (_left == nullptr && _right == nullptr)
        {
            return 1;
        }
        else
        {
            return _left->leaf_num() + _right->leaf_num();
```

```
        }
    }
    //(2)
    bool operator== (BinaryTree* bt)
    {
        if (this == nullptr && bt == nullptr)
        {
            return true;
        }
        else if (this == nullptr || bt == nullptr)
        {
            return false;
        }
        if (_val == bt->_val)
        {
            return (_left == bt->_left) && (_right == bt->_right);
        }
        else
        {
            return false;
        }
    }
    //(3)
    void exchange_node()
    {
        if (this == nullptr)
            return;
        _left->exchange_node();
        _right->exchange_node();
        BinaryTree* tmp = _left;
        _left = _right;
        _right = tmp;
    }
    //(4)
    void double_order_traverse()
    {
        if (this)
        {
            cout << _val;
            _left->exchange_node();
            cout << _val;
            _right->exchange_node();
        }
    }
    //(5)
#define MAXDEPTH 100
    void _max_width(int level, vector<int>& v)
    {
        if (this)
        {
            v[level]++;
            _left->_max_width(level + 1, v);
            _right->_max_width(level + 1, v);
        }
```

```
    }
    int max_width()
    {
        if (this)
        {
            vector<int> v(MAXDEPTH, 0);
            int level = 1;
            _max_width(level, v);
            int max = 0;
            for (int i = 1; v[i] != 0; i++)
            {
                if (v[i] > max)
                    max = v[i];
            }
            return max;
        }
    }
    //(6)
    int degree_one()
    {
        queue<BinaryTree> q;
        q.push(*this);
        int ret = 0;
        while (!q.empty())
        {
            BinaryTree tmp = q.front();
            q.pop();
            if (tmp._left && tmp._right)
            {
                q.push(*tmp._left);
                q.push(*tmp._right);
            }
            else if (tmp._left || tmp._right)
            {
                ret++;
                if (tmp._left)
                {
                    q.push(*tmp._left);
                }
                else
                {
                    q.push(*tmp._right);
                }
            }
        }
        return ret;
    }
    //(7)
    vector<value_type> _max_route()
    {
        if (this)
        {
            vector<value_type> vl = _left ? _left->_max_route() :
vector<value_type>();
```

```
            vector<value_type> vr = _right ? _right->_max_route() :
vector<value_type>();
            if (vl.size() >= vr.size())
            {
                vl.push_back(_val);
                return vl;
            }
            else
            {
                vr.push_back(_val);
                return vr;
            }
        }
        return vector<value_type>();
    }
    void max_route()
    {
        if (this)
        {
            vector<value_type> v = _max_route();
            auto rit = v.rbegin();
            while (rit != v.rend())
            {
                cout << *rit << ' ';
                rit++;
            }
        }

    }
    //(8)
    void all_route(queue<value_type> q = queue<value_type>())
    {
        if (this)
        {
            q.push(_val);
            //叶子节点
            if (_left == nullptr && _right == nullptr)
            {
                while (!q.empty())
                {
                    value_type tmp = q.front();
                    cout << tmp;
                    q.pop();
                }
                cout << endl;
            }
            if (_left)
            {
                _left->all_route(q);
            }
            if (_right)
            {
                _right->all_route(q);
            }
```

```
        }
    }

private:
    value_type _val;
    BinaryTree* _left;
    BinaryTree* _right;
};
```