二叉树算法设计练习

设计算法按前序次序打印二叉树中的叶子结点。

```
void PreOrder(BiNode *root)
{
    if (root == NULL) return;
    else {
        if (root->lchild ==NULL && root->rchild==NULL)
              cout<<root->data;
        PreOrder(root->lchild);
        PreOrder(root->rchild);
    }
}
```

二叉树算法设计练习

设计算法求二叉树中的叶子结点个数。

```
int  CountLeaf (BiNode * T)
{// 先序遍历二叉树，以 count 返回二叉树中叶子结点数
    if ( T ==null) return 0;
    else  if ((!T->Lchild)&& (!T->Rchild))
                return 1;       // 对叶子结点计数
          else{
            int a=CountLeaf( T->Lchild)
            int b = CountLeaf( T->Rchild);
            return a+b;   }
             }
} // CountLeaf
```

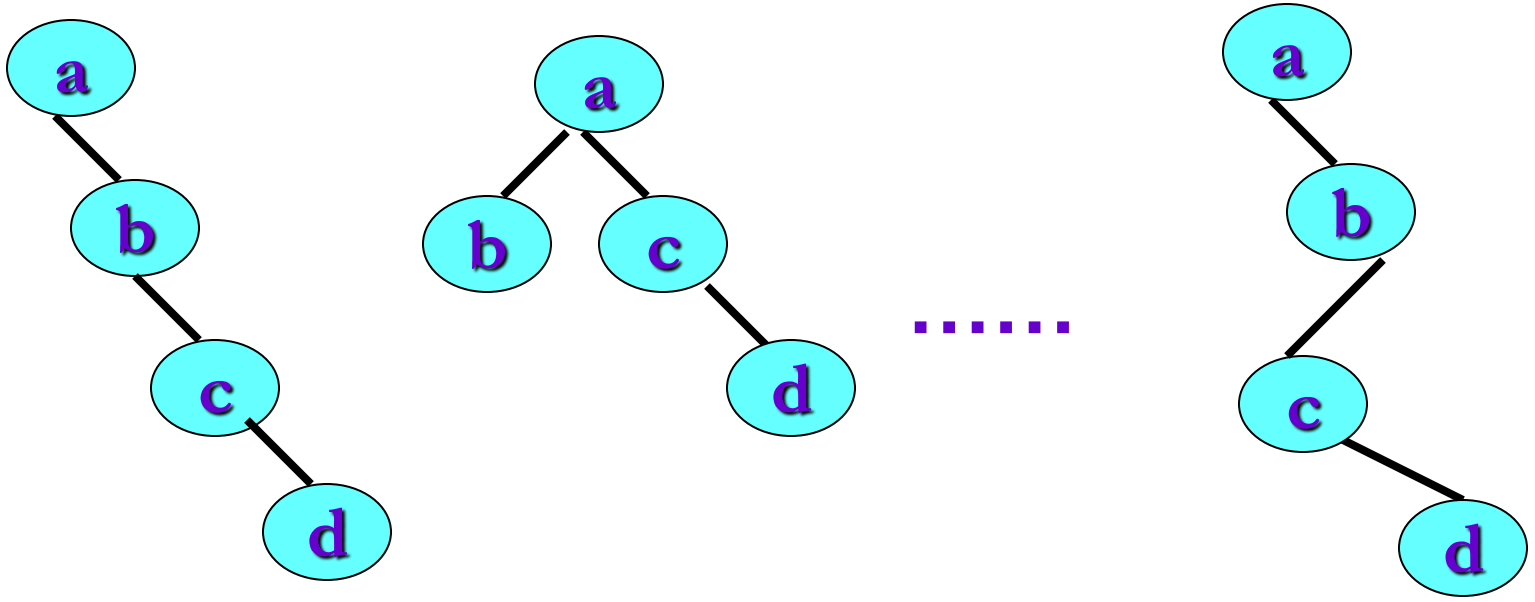
二叉树算法设计练习

设计算法求二叉树中的叶子结点个数。

```
void CountLeaf (BiNode * T, int& count)
{// 先序遍历二叉树，以 count 返回二叉树中叶子结点数
    if ( T ) {
        if ((!T->Lchild)&& (!T->Rchild))
            count++;        // 对叶子结点计数
        else{
            CountLeaf( T->Lchild, count);
            CountLeaf( T->Rchild, count);
                }
        } // if
} // CountLeaf
```

由二叉树的先序和中序序列建树

- 若已知先序序列abcd，则可知道a为树的根结点但哪个是左子树，哪个是右子树，无法确定，有多棵符合条件的树与之相对应



C.由二叉树的先序和中序序列建树

- 若已知中序序列，则无法确定根，也就无法确定左子树和右子树，也无法唯一确定一棵二叉树。如abcde

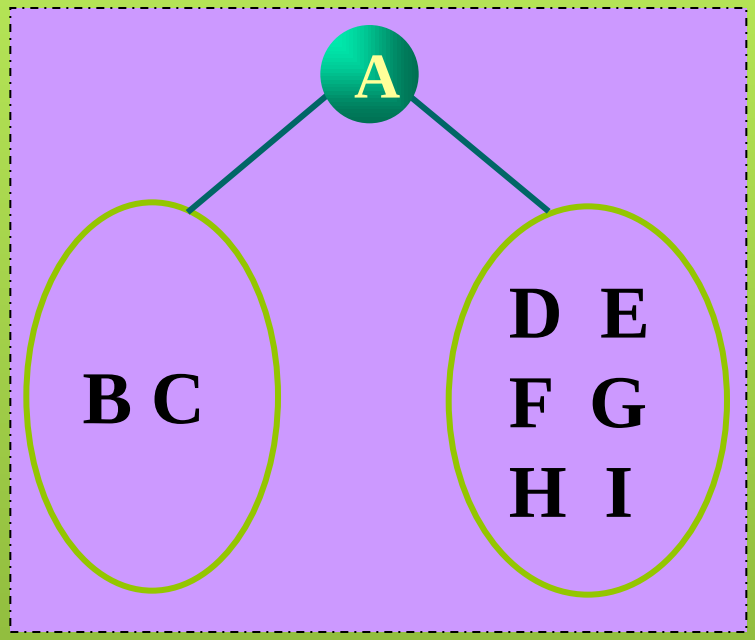
怎么办呢?

将先序与中序结合起来!

由先序序列确定根结点

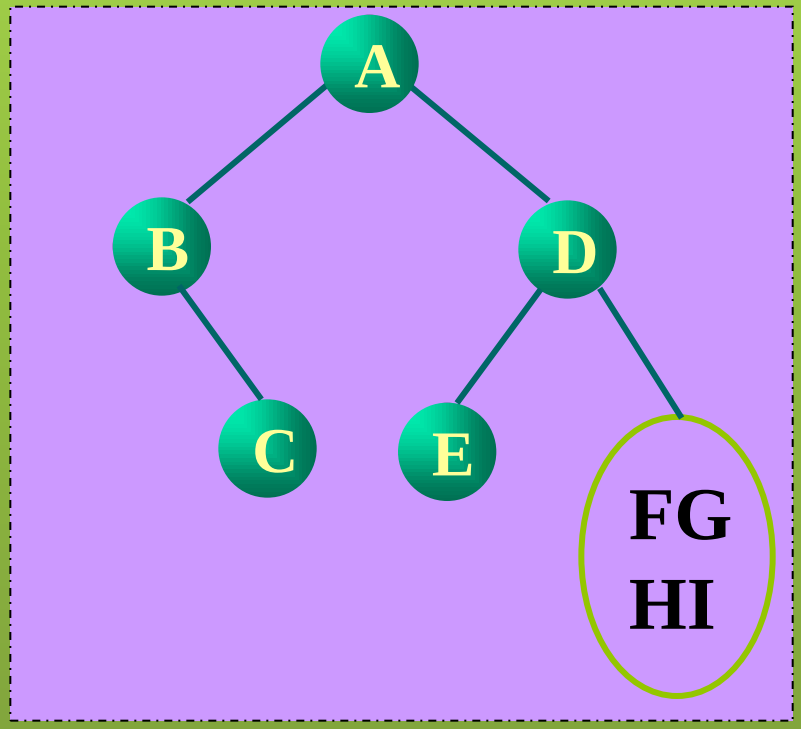
再由中序序列找出左右子树。

可唯一确定一棵二叉树!



前序：A B C D E F G H I

中序：B C A E D G H F I

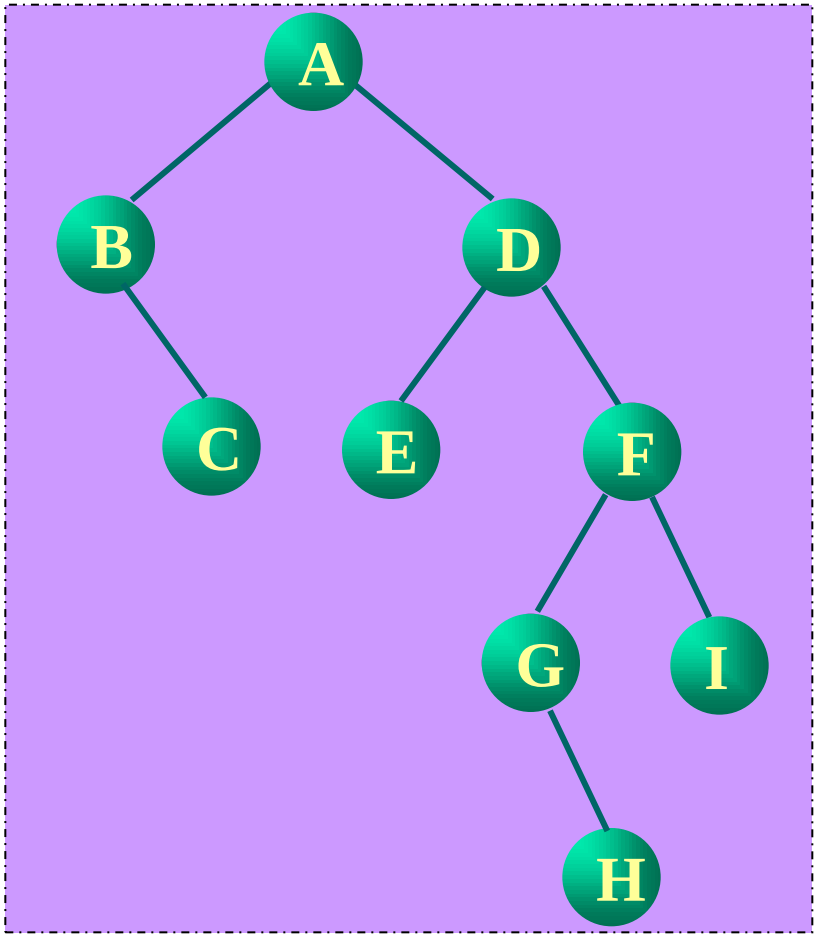


前序：B C

中序：B C

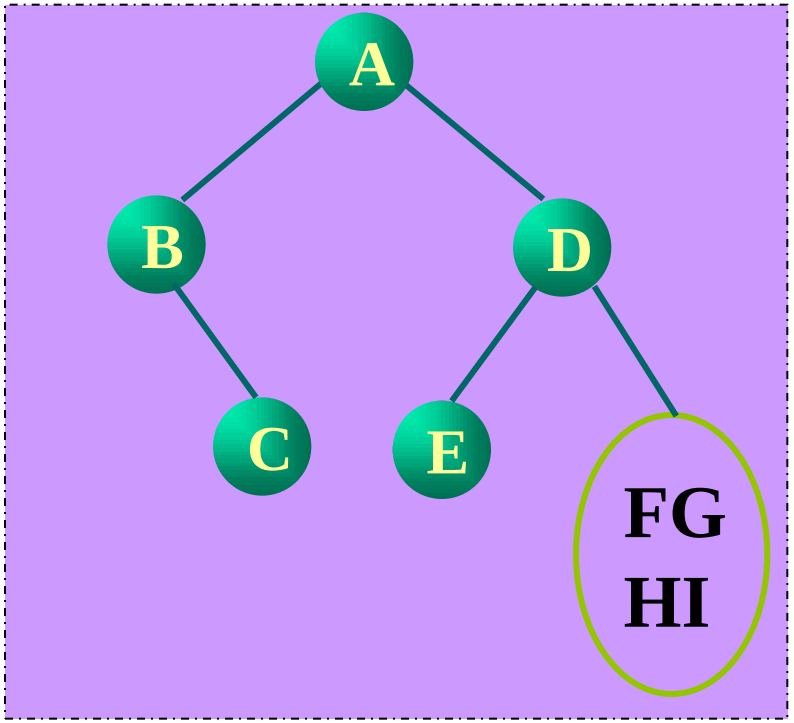
前序：D E F G H I

中序：E D G H F I



前序：F G H I

中序：G H F I



前序：D E F G H I

中序：E D G H F I

- 思考：若已知后序序列和中序序列是否可以唯一确定一棵二叉树？
- 可以！
- 例：已知一棵二叉树的中序序列和后序序列分别是BDCEAFHG 和DECBHGFA，请画出这棵二叉树。
- 分析：
- ①由后序遍历特征，根结点必在后序序列尾部（即A）；
- ②由中序遍历特征，根结点必在其中间，而且其左部必全部是左子树子孙（即BDCE），其右部必全部是右子树子孙（即FHG）；
- ③继而，根据后序中的DECB子树可确定B为A的左孩子，根据HGF子串可确定F为A的右孩子；以此类推。

二叉树遍历的非递归算法

前序遍历——非递归算法

二叉树前序遍历的非递归算法的关键：在前序遍历过某结点的整个左子树后，如何找到该结点的右子树的根指针。

解决办法：在访问完该结点后，将该结点的指针保存在栈中，以便以后能通过它找到该结点的右子树。

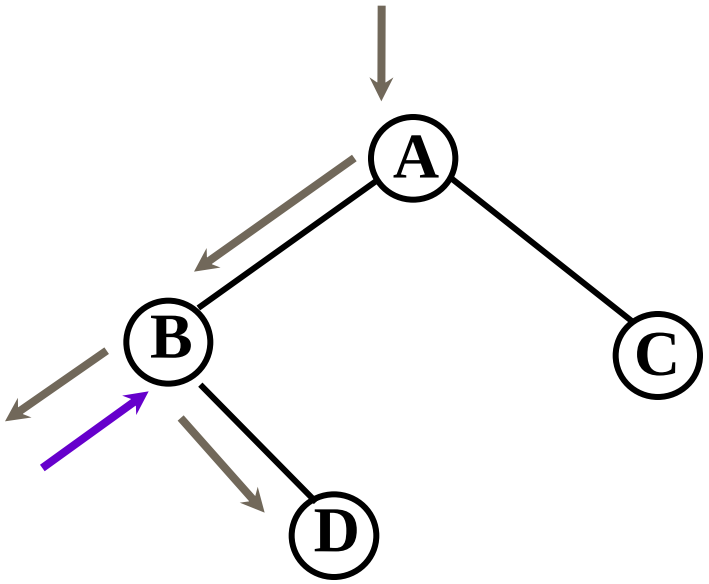
在前序遍历中，设要遍历二叉树的根指针为root，则有两种可能：

(1) 若root!=NULL，则表明？如何处理？

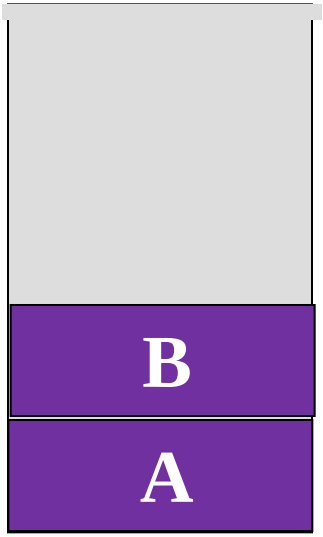
(2) 若root=NULL，则表明？如何处理？

二叉树遍历的非递归算法

前序遍历的非递归实现



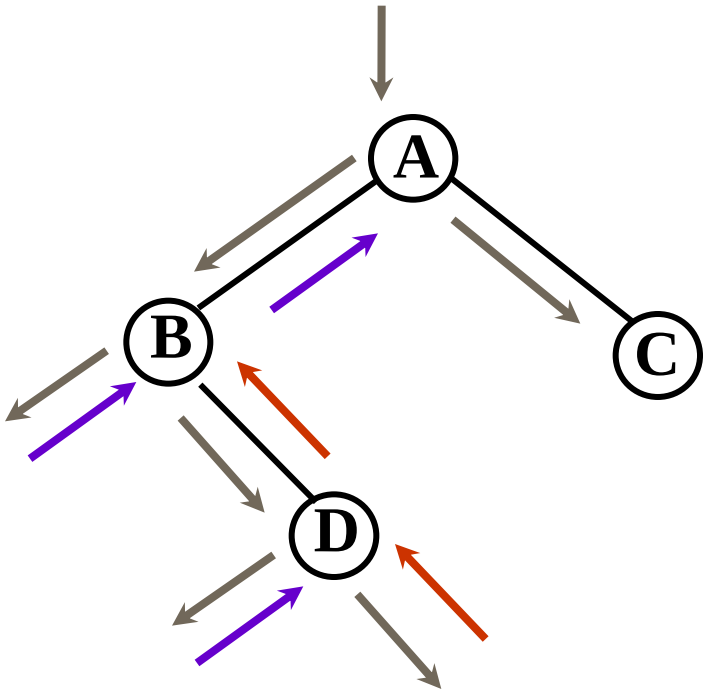
栈S内容:



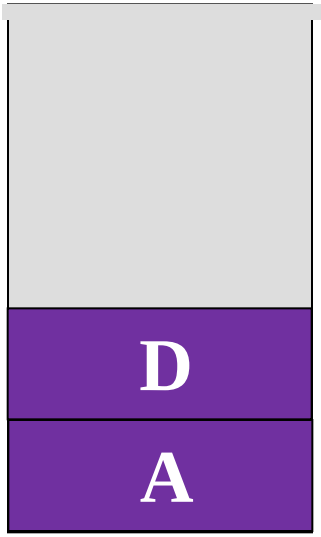
访问结点序列: A B D

二叉树遍历的非递归算法

前序遍历的非递归实现



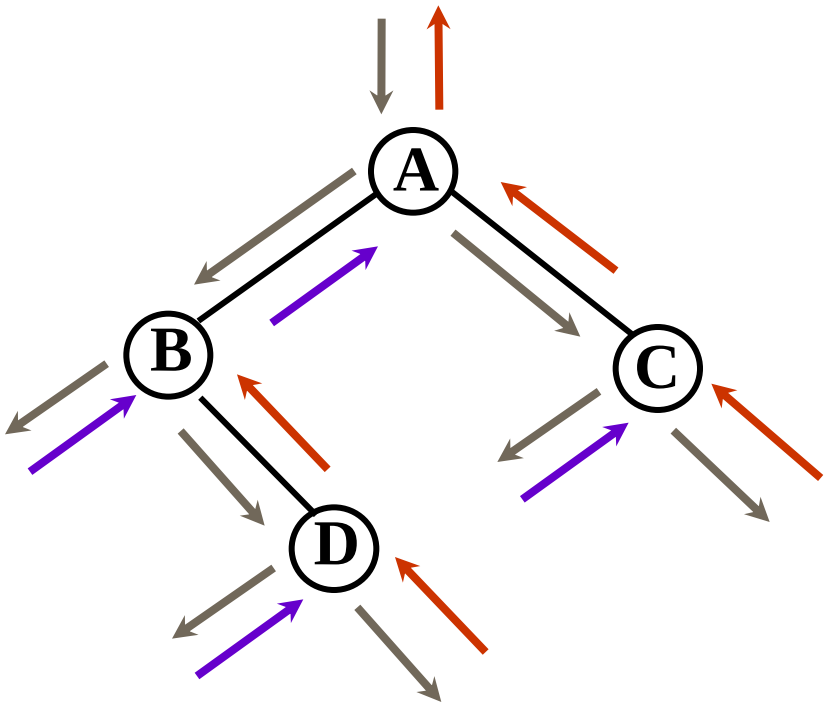
栈S内容:



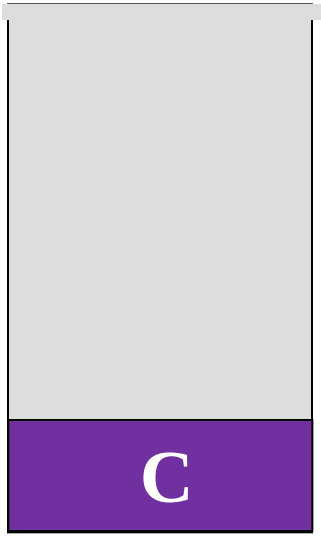
访问结点序列: A B D

二叉树遍历的非递归算法

前序遍历的非递归实现



栈S内容:



访问结点序列: A B D C

二叉树遍历的非递归算法

前序遍历——非递归算法（伪代码）

1.栈s初始化；

2.循环直到p为空且栈s为空

2.1 当p不空时循环

2.1.1 输出p->data;

2.1.2 将指针p的值保存到栈中；

2.1.3 继续遍历p的左子树

2.2 如果栈s不空，则

2.2.1 将栈顶元素弹出至p；

2.2.2 准备遍历p的右子树；

二叉树遍历的非递归算法（自学）

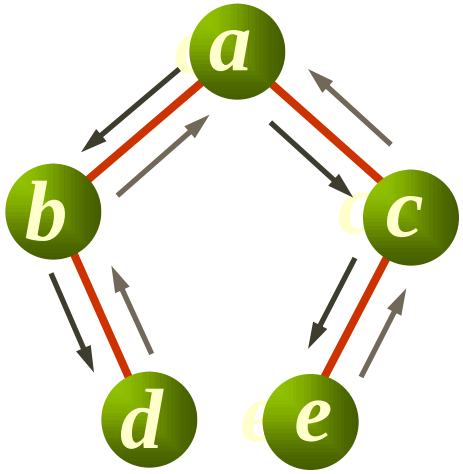
```
template <class T>
void BinaryTree<T>::preOrder (void (*visit) (BinTreeNode<T> *t))
{
    stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p = root;
  do {
        while (p != NULL) {    //遍历指针向左下移动
            visit(p);
            S.Push (p);              //该子树沿途结点进栈
            p = p->leftChild;
              }
        if (!S.IsEmpty()) {                  //栈不空时退栈
            S.Pop (p);    //退栈
            p = p->rightChild; //遍历指针进到右子女
        }
    } while (p != NULL || !S.IsEmpty ());
};
```

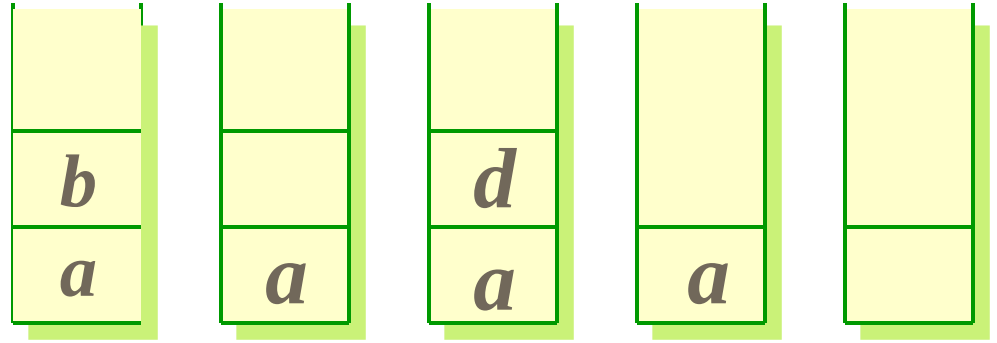
二叉树遍历的非递归算法

中序遍历——非递归算法

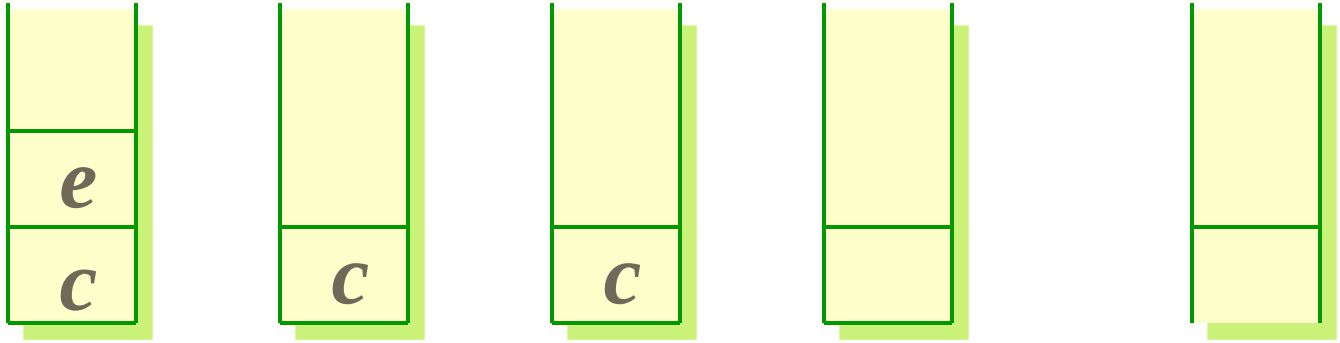
- 在二叉树的中序遍历中，访问结点的操作发生在该结点的左子树遍历完毕并准备遍历右子树时，
- 所以，在遍历过程中遇到某结点时并不能立即访问它，而是将它压栈，等到它的左子树遍历完毕后，再从栈中弹出并访问之。
- 中序遍历的非递归算法只需将前序遍历的非递归算法中的访问语句，移到S.Pop (p)出栈之后即可。

利用栈的中序遍历非递归算法





左空　退栈访问　左空　退栈访问　退栈访问



左空　退栈访问　右空　退栈访问　栈空结束

```
template <class T>
void BinaryTree<T>::InOrder (void (*visit) (BinTreeNode<T> *t)) {
    stack<BinTreeNode<T>*> S;
   BinTreeNode<T> *p = root;
    do {
        while (p != NULL) {   //遍历指针向左下移动
            S.Push (p);           //该子树沿途结点进栈
            p = p->leftChild;
        }
        if (!S.IsEmpty()) {                 //栈不空时退栈
            S.Pop (p);  visit (p);          //退栈, 访问
            p = p->rightChild;              //遍历指针进到右子女
        }
    } while (p != NULL || !S.IsEmpty ());
};
```

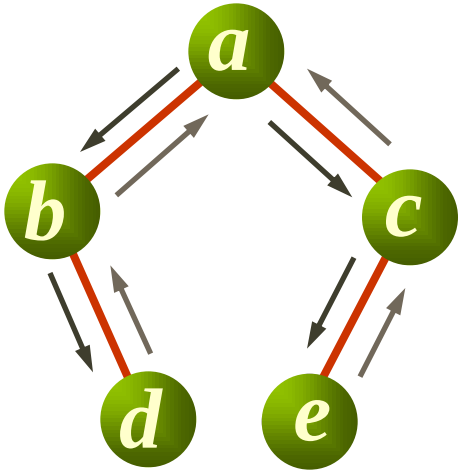
利用栈的后序遍历非递归算法

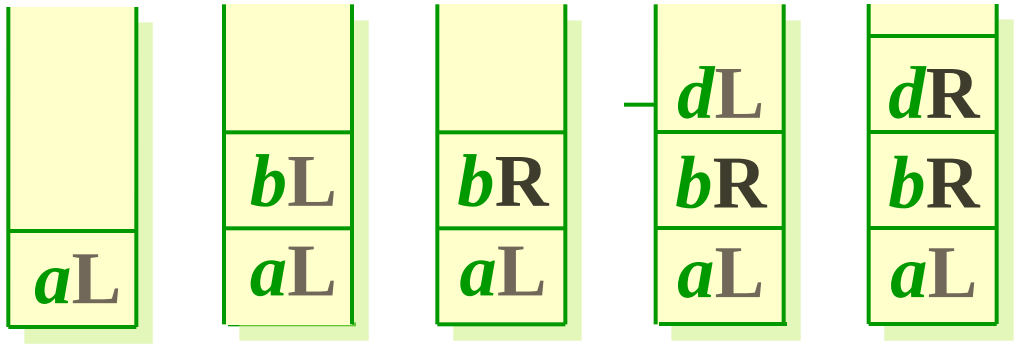
- 在后序遍历过程中所用栈的结点定义

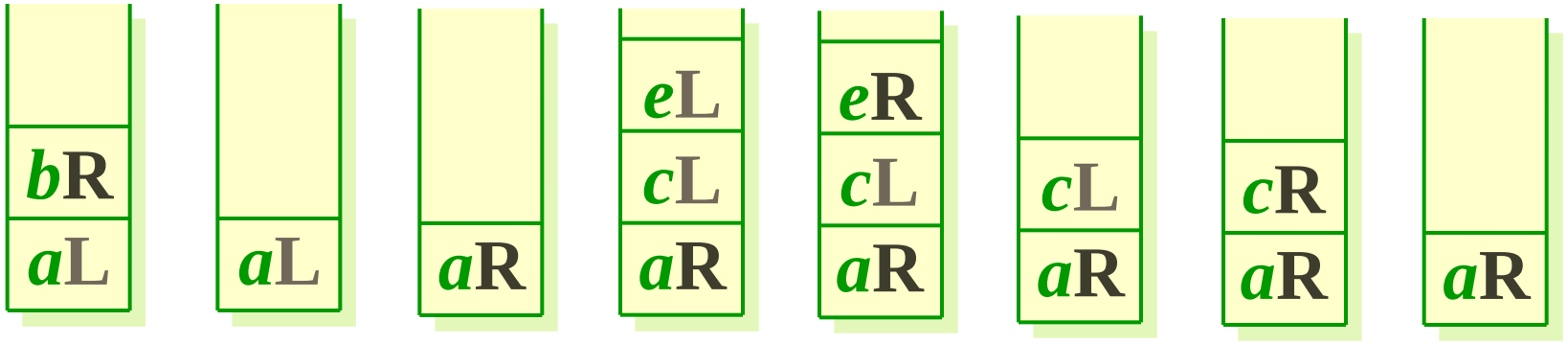
```
template <class T>
 struct stkNode {
     BinTreeNode<T> *ptr;      //树结点指针
      enum tag {L, R};         //退栈标记
     stkNode (BinTreeNode<T> *N = NULL) :
         ptr(N), tag(L) { }                //构造函数
};
```

ptr	tag{L,R}

- tag = L，表示从左子树退回还要遍历右子树;
  tag = R，表示从右子树退回要访问根结点。


a
b
c
d
e


aL
bL
aL
bR
aL
dL
bR
aL
dR
bR
aL


bR
aL
aL
aR
eL
cL
aR
eR
cL
aR
cL
aR
cR
aR
aR

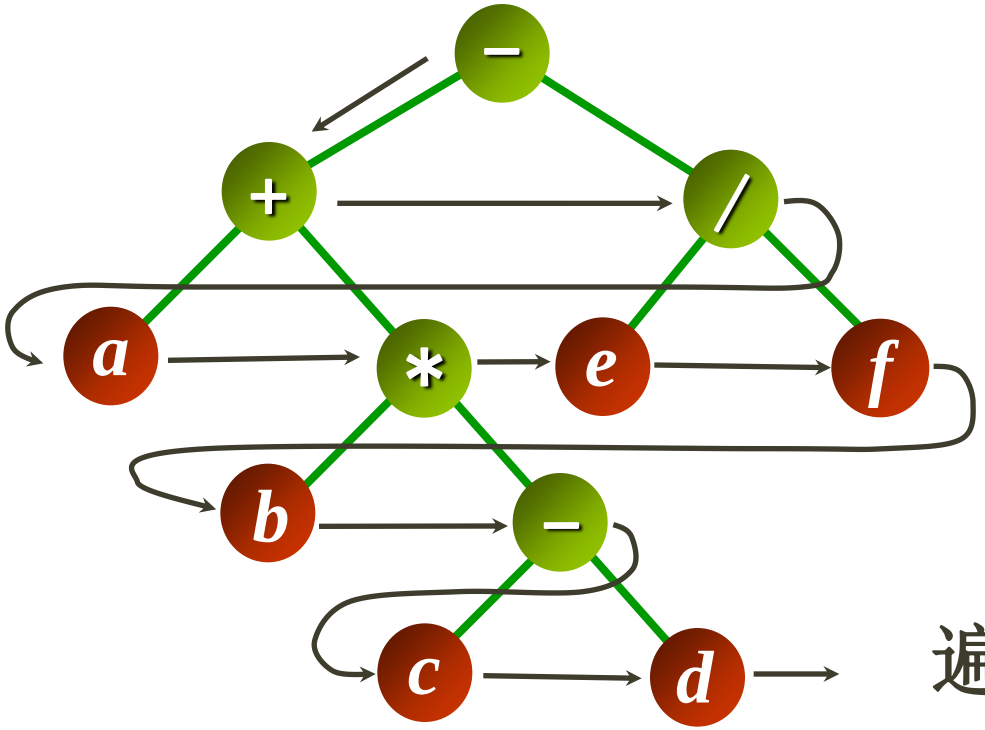
后序遍历的非递归算法（自学）

```
template <class T>
void BinaryTree<T>::
PostOrder (void (*visit) (BinTreeNode<T> *t) {
    Stack<stkNode<T>> S;   stkNode<T> w;
    BinTreeNode<T> * p = root;     //p是遍历指针
   do {
       while (p != NULL) {
           w.ptr = p;  w.tag = L;  S.Push (w);
           p = p->leftChild;
       }
       int continue1 = 1;     //继续循环标记, 用于R
```

```
        while (continue1 && !S.IsEmpty ()) {
            S.Pop (w);  p = w.ptr;
            switch (w.tag) {          //判断栈顶的tag标记
                case L:  w.tag = R;  S.Push (w);
                         continue1 = 0;
                         p = p->rightChild;  break;
                case R:  visit (p);   break;
            }
        }
    } while (!S.IsEmpty ());      //继续遍历其他结点
    cout << endl;
};
```

层次序遍历二叉树的算法

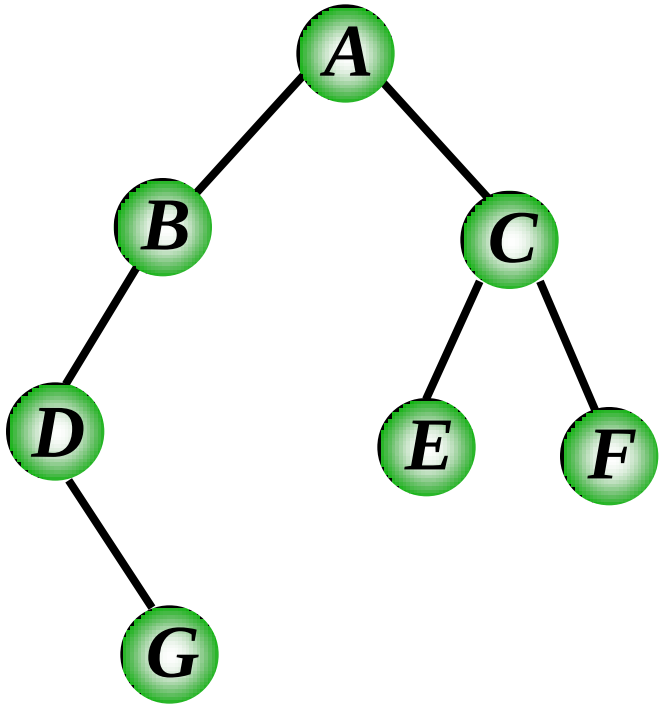
- 层次序遍历二叉树就是从根结点开始，按层次逐层遍历，如图：



- 这种遍历需要使用一个先进先出的队列，在处理上一层时，将其下一层的结点直接进到队列（的队尾）。在上一层结点遍历完后，下一层结点正好处于队列的队头，可以继续访问它们。
- 算法是非递归的。

5.4 二叉树的存储结构及实现

层序遍历



A B C D E F G

遍历序列：A B C D E F G

层序遍历

1. 队列Q初始化；
2. 如果二叉树非空，将根指针入队；
3. 循环直到队列Q为空
   - 3.1 q=队列Q的队头元素出队；
   - 3.2 访问结点q的数据域；
   - 3.3 若结点q存在左孩子，则将左孩子指针入队；
   - 3.4 若结点q存在右孩子，则将右孩子指针入队；

层次序遍历的算法

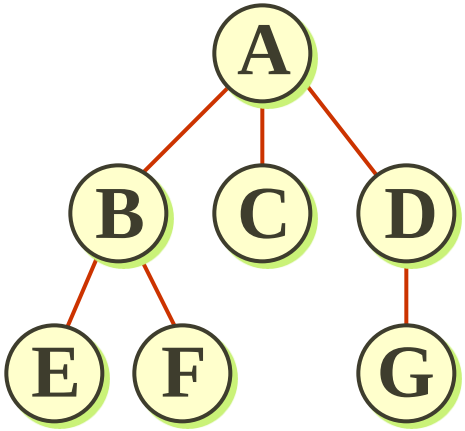
```
template <class T>
void BinaryTree<T>::
levelOrder (void (*visit) (BinTreeNode<T> *t)) {
    if (root == NULL) return;
    Queue<BinTreeNode<T> * > Q;
    BinTreeNode<T> *p = root;
     Q.EnQueue (p);
    while (!Q.IsEmpty ()) {
        Q.DeQueue (p);
         visit(p);
        if (p->leftChild != NULL) Q.EnQueue (p->leftChild);
        if (p->rightChild != NULL) Q.EnQueue (p->rightChild);
        }
    };
```

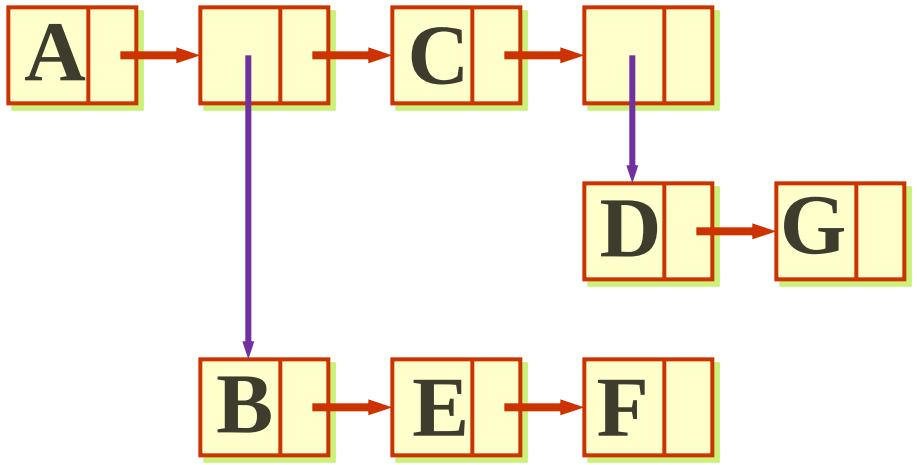
```
template <class T>//判断是否完全二叉树,是则返回1,否则返回0
int levelOrder (BitTreeNode<T> * root) {
    if (root == NULL) return 0;
    Queue<BinTreeNode<T> *> Q;
    BinTreeNode<T> *p = root;
     Q.EnQueue (p);
    while (!Q.IsEmpty ())
  {    Q.DeQueue (p);
        if(!p) flag=1;
       else if(flag)return 0;
          else {  Q.EnQueue (p->leftChild);
                  Q.EnQueue (p->rightChild);  } }
     return 1;
}
```

5.6 树与森林

5.6.1 树的存储表示

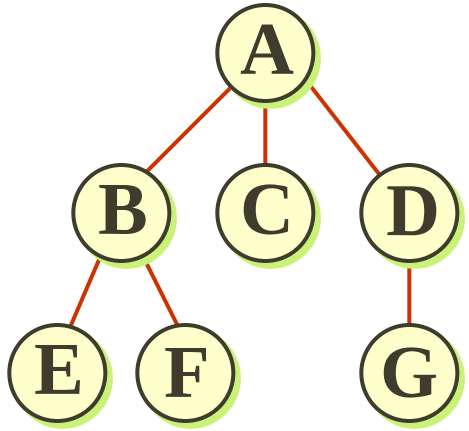
1、广义表表示





表示为：A(B(E, F), C, D(G))

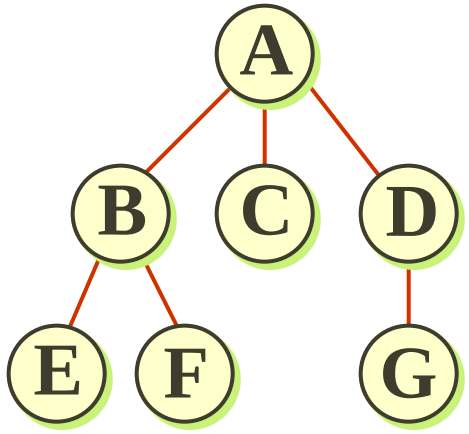
2、父指针表示

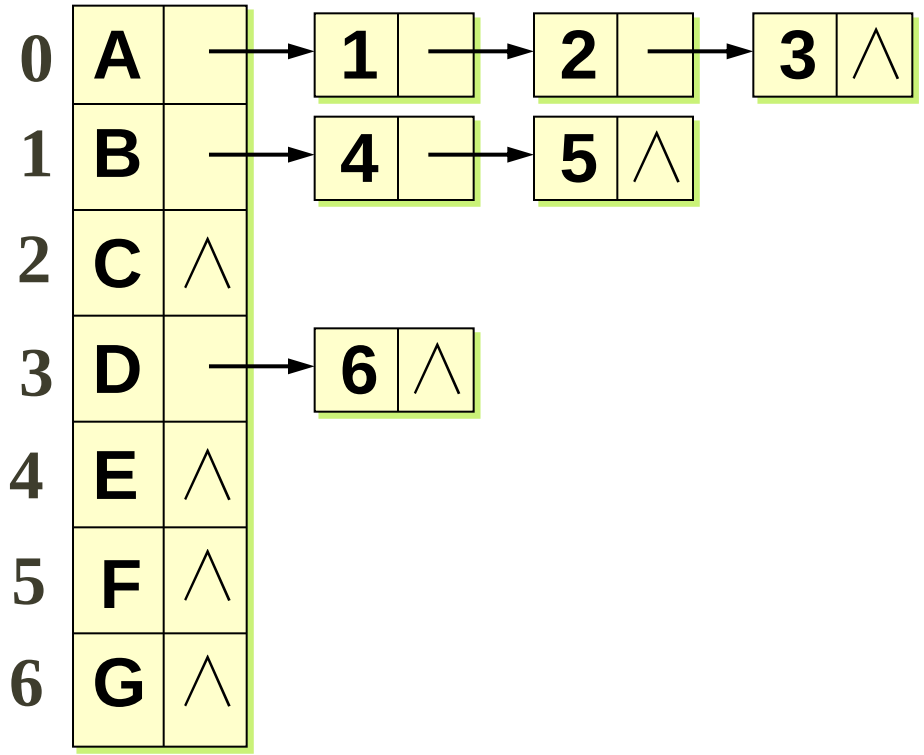


	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	−1	0	0	0	1	1	3

- 树中结点的存放顺序一般不做特殊要求，但为了操作实现的方便，有时也会规定结点的存放顺序。
- 此存储结构找父节点的时间复杂度为O(1),但找子女需遍历整个数组。

3、子女链表表示





- 无序树情形链表中各结点顺序任意，有序树必须自左向右链接各个子女结点。

4、子女-兄弟表示

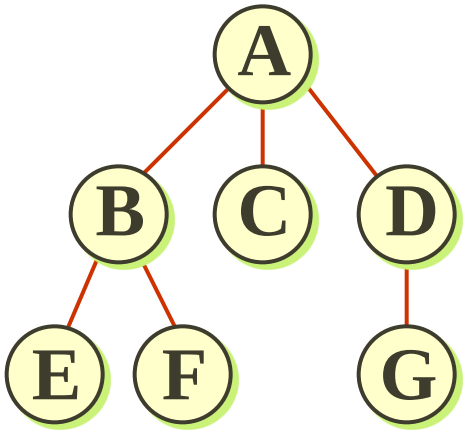
- 也称为树的二叉树表示。结点构造为：

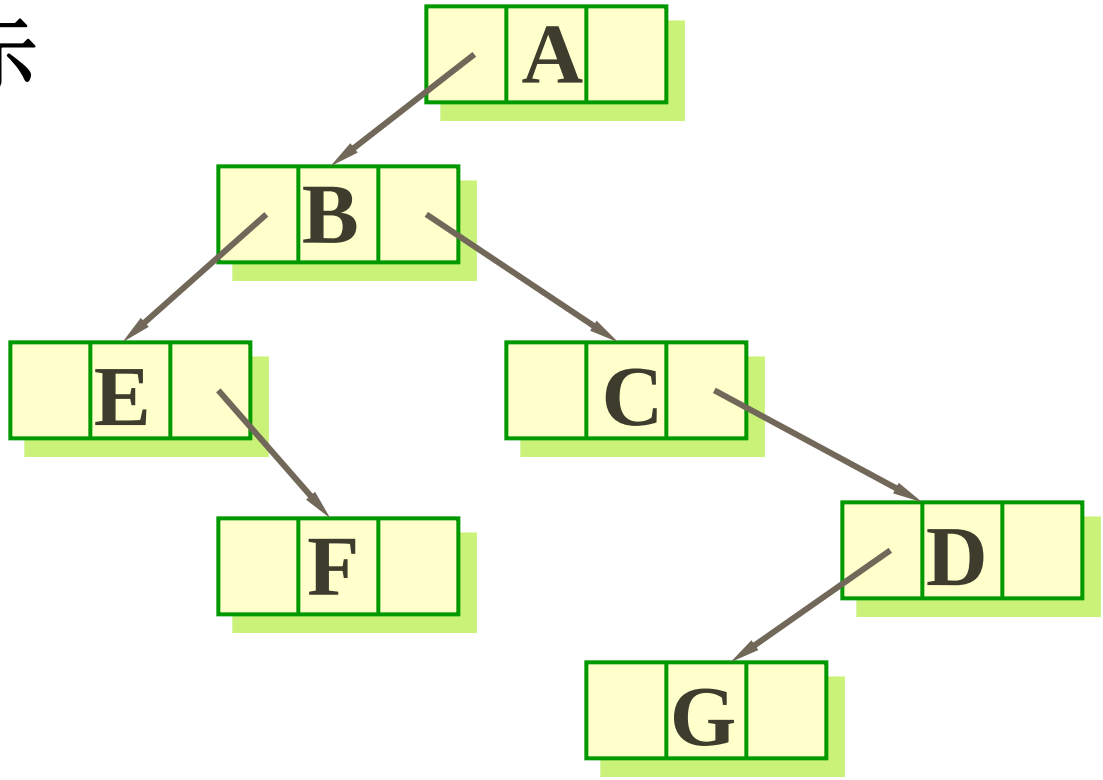
- **firstChild** 指向该结点的第一个子女结点。无序树时，可任意指定一个结点为第一个子女。
- **nextSibling** 指向该结点的下一个兄弟。任一结点在存储时总是有顺序的。
- 若想找某结点的所有子女，可先找**firstChild**，再反复用 **nextSibling** 沿链扫描。

4、子女-兄弟表示

data	firstChild	nextSibling

树的子女-兄弟表示





用子女-兄弟表示的树的类定义

```
template <class T>
struct TreeNode {                          //树的结点类
    T data;                                //结点数据
    TreeNode<T> *firstChild, *nextSibling;
                                           //子女及兄弟指针
   TreeNode (T value = 0, TreeNode<T> *fc = NULL,
      TreeNode<T> *ns = NULL)     //构造函数
     : data (value), firstChild (fc), nextSibling (ns) { }
};
```

用子女-兄弟表示的树的类定义

```
template <class T>
class Tree {                    //树类
private:
    TreeNode<T> *root, *current;
                                        //根指针及当前指针
    int Find (TreeNode<T> *p, T value);
                                    //在以p为根的树中搜索value
    void RemovesubTree (TreeNode<T> *p);
                                        //删除以p为根的子树
    bool FindParent (TreeNode<T> *t, TreeNode<T> *p);
public:
```

用子女-兄弟表示的树的类定义

```
Tree () { root = current = NULL; }        //构造函数
bool Root ();                  //置根结点为当前结点
bool IsEmpty () { return root == NULL; }
bool FirstChild ();
   //将当前结点的第一个子女置为当前结点
bool NextSibling ();
   //将当前结点的下一个兄弟置为当前结点
bool Parent ();    //将当前结点的双亲置为当前结点
bool Find (T value);
   //搜索含value的结点, 使之成为当前结点
……                             //树的其他公共操作
};
```

子女-兄弟链表常用操作的实现

```
template <class T>
bool Tree<T>::Root () {
//让树的根结点成为树的当前结点
    if (root == NULL) {
        current = NULL;  return false;
    }
    else {
        current = root;  return true;
    }
};
```

子女-兄弟链表常用操作的实现

```
template <class T>
bool Tree<T>::Parent () {
//置当前结点的双亲结点为当前结点
    TreeNode<T> *p = current;
    if (current == NULL || current == root)
      { current = NULL;  return false; }
        //空树或根无双亲
    return FindParent (root, p);
                //从根开始找*p的双亲结点
};
```

子女-兄弟链表常用操作的实现

```
template <class T>
bool Tree<T>::
FindParent (TreeNode<T> *t, TreeNode<T> *p) {
//在根为*t的树中找*p的双亲, 并置为当前结点
    TreeNode<T> *q = t->firstChild;     //*q是*t长子
    bool succ;
    while (q != NULL && q != p) {     //扫描兄弟链
        if ((succ = FindParent (q, p)) == true)
            return succ;     //递归搜索以*q为根的子树
        q = q->nextSibling;
    }
```

子女-兄弟链表常用操作的实现

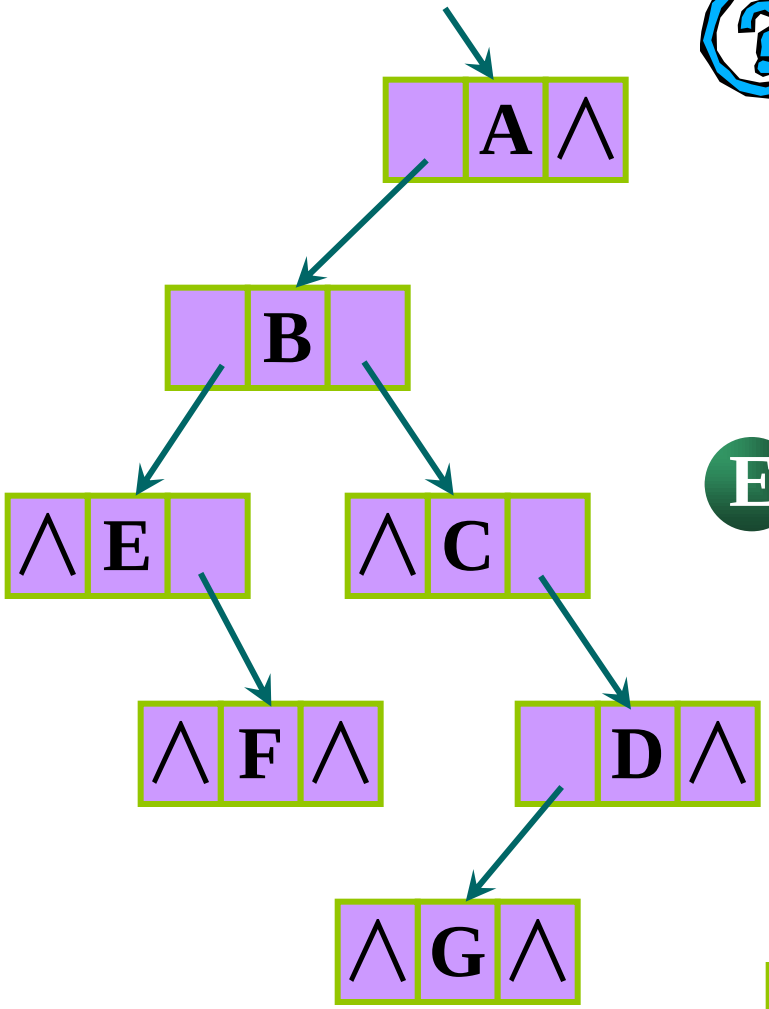
```
    if (q != NULL && q == p) {
        current = t;  return true;
    }
    else { current = NULL;  return false; }   //未找到
};

template <class T>
bool Tree<T>::FirstChild () {
//在树中找当前结点的长子, 并置为当前结点
    if (current && current->firstChild )
       { current = current->firstChild;  return true; }
    current = NULL;  return false;
};
```

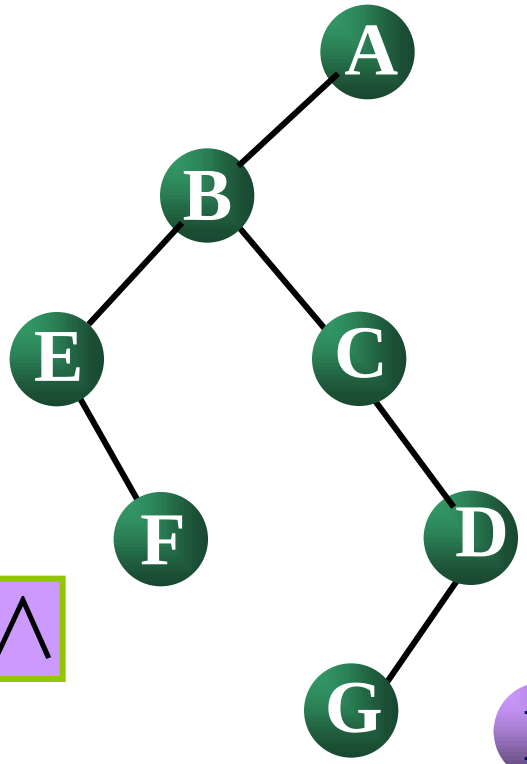
子女-兄弟链表常用操作的实现

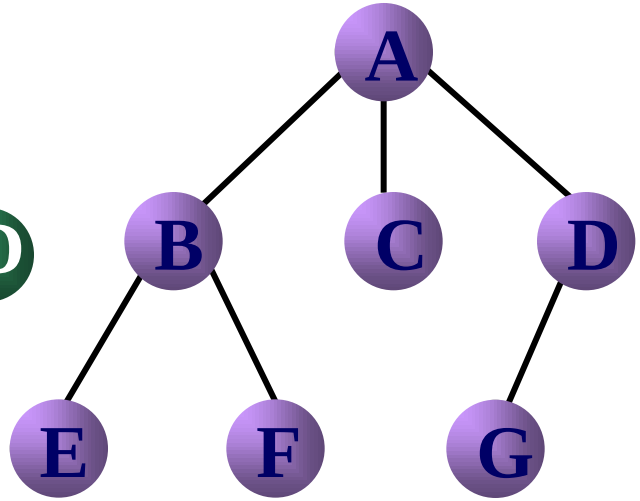
```
template <class T>
bool Tree<T>::NextSibling () {
//在树中找当前结点的兄弟,并置为当前结点
    if (current && current->nextSibling) {
        current = current->nextSibling;
        return true;
    }
    current = NULL;  return false;
};
```

5.6 树、森林与二叉树的转换

是哪些树结构的存储结构?







树和二叉树之间具有对应关系

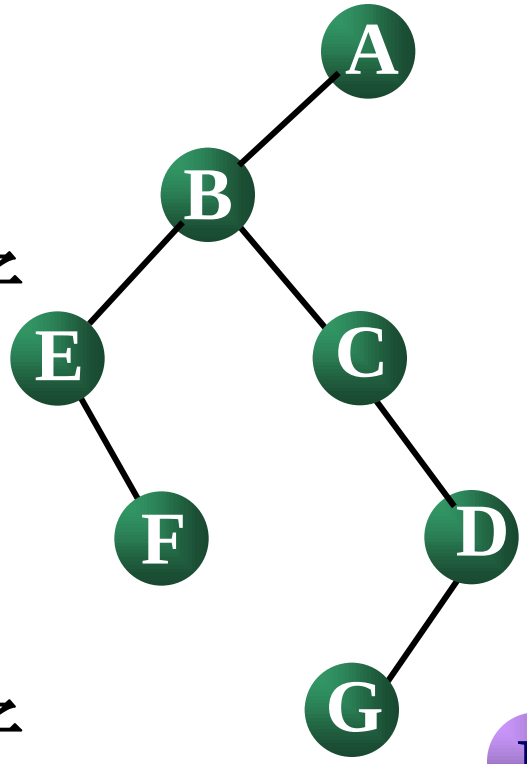
5.6 树、森林与二叉树的转换

树和二叉树之间的对应关系

树：兄弟关系

⇕

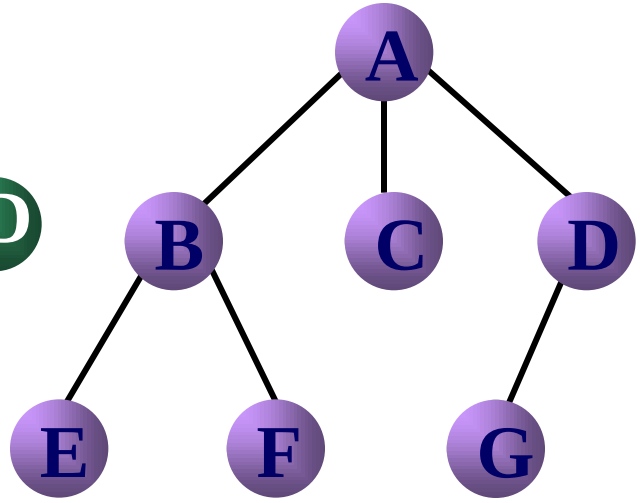
二叉树：双亲和右孩子



树：双亲和长子

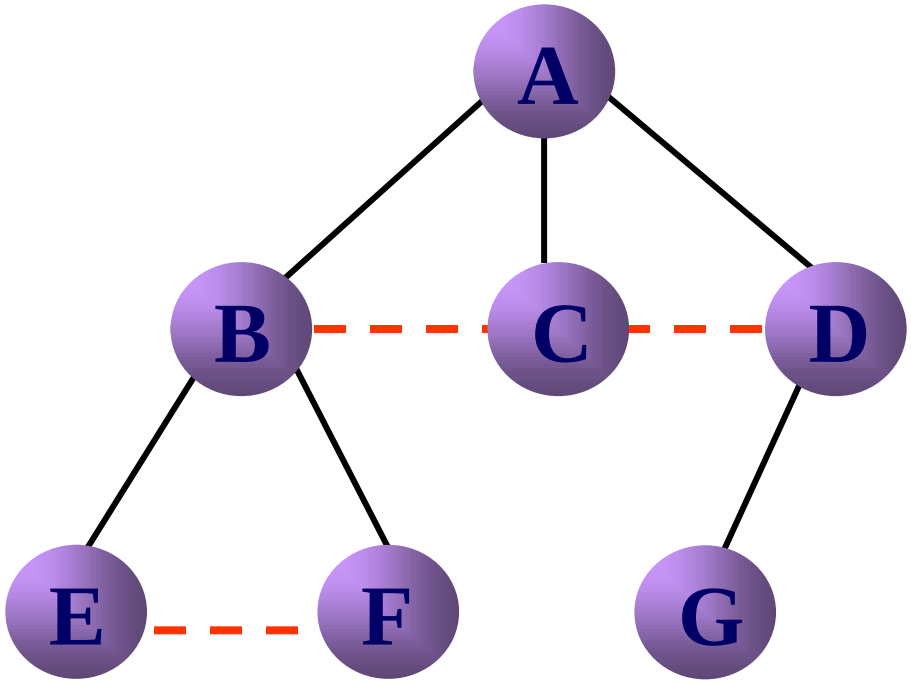
⇕

二叉树：双亲和左孩子



5.6 树、森林与二叉树的转换

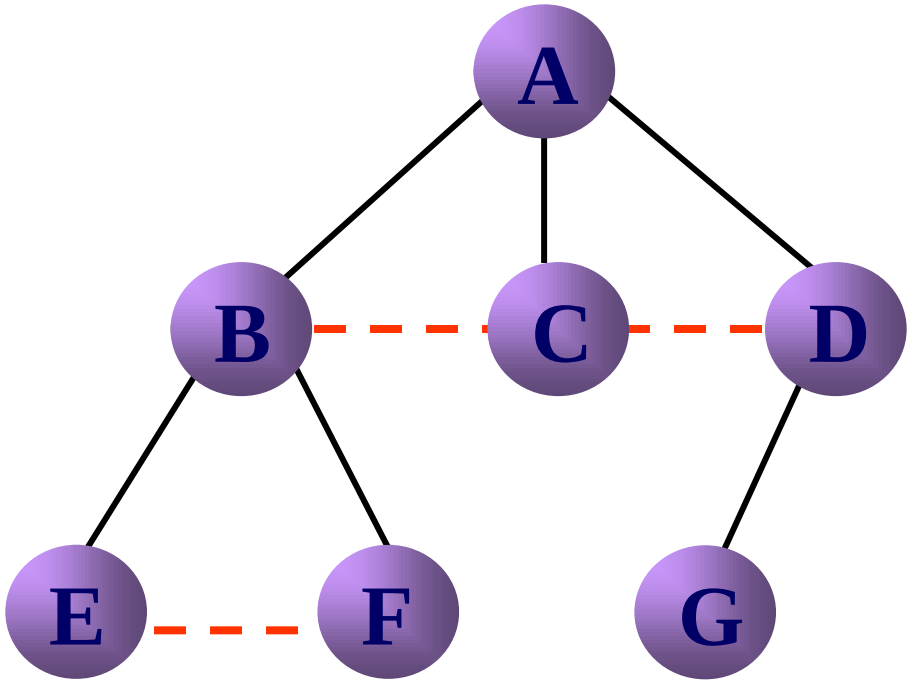
树和二叉树之间的对应关系



1.兄弟加线.

5.6 树、森林与二叉树的转换

树和二叉树之间的对应关系

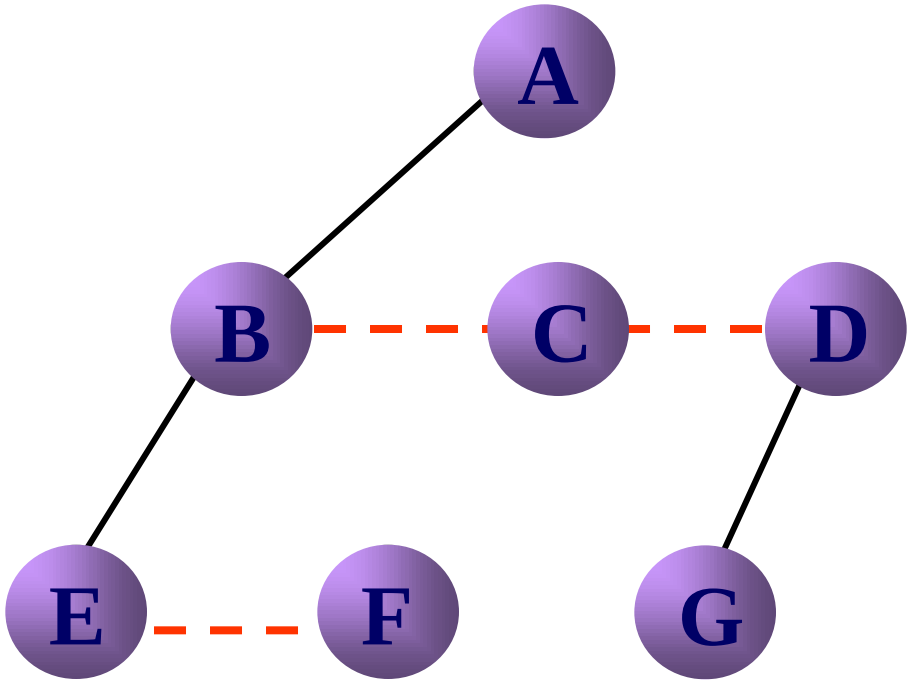


1.兄弟加线.

2.保留双亲与第一孩子连线,删去与其他孩子的连线.

5.6 树、森林与二叉树的转换

树和二叉树之间的对应关系



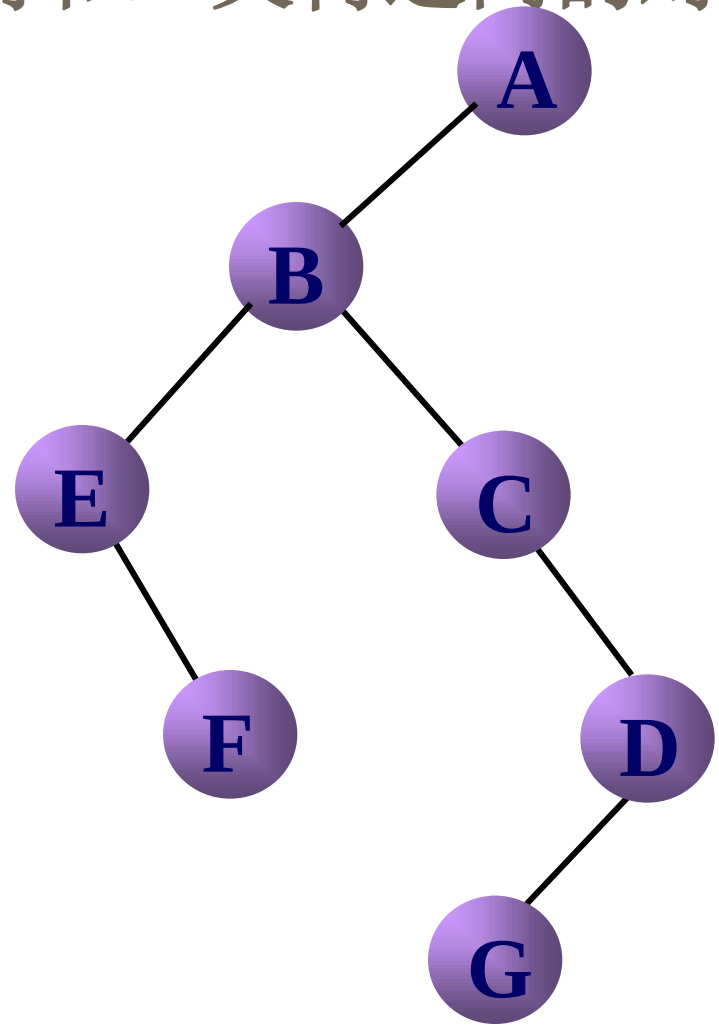
1.兄弟加线.

2.保留双亲与第一孩子连线,删去与其他孩子的连线.

3.顺时针转动,使之层次分明.

5.6 树、森林与二叉树的转换

树和二叉树之间的对应关系



1.兄弟加线.

2.保留双亲与第一孩子连线,删去与其他孩子的连线.

3.顺时针转动,使之层次分明.

5.6 树、森林与二叉树的转换

树转换为二叉树

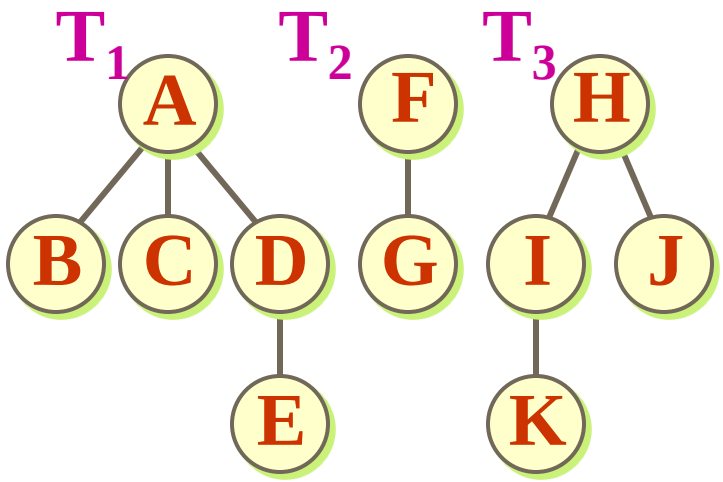
(1)加线——树中所有相邻兄弟之间加一条连线。

(2)去线——对树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其它孩子结点之间的连线。

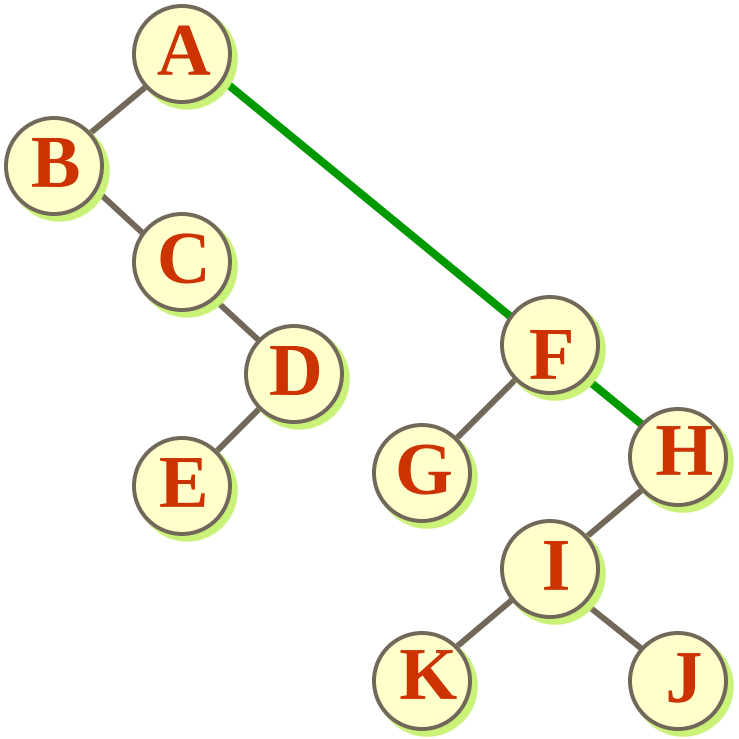
(3)层次调整——以根结点为轴心，将树顺时针转动一定的角度，使之层次分明。

5.6.2 森林与二叉树的转换

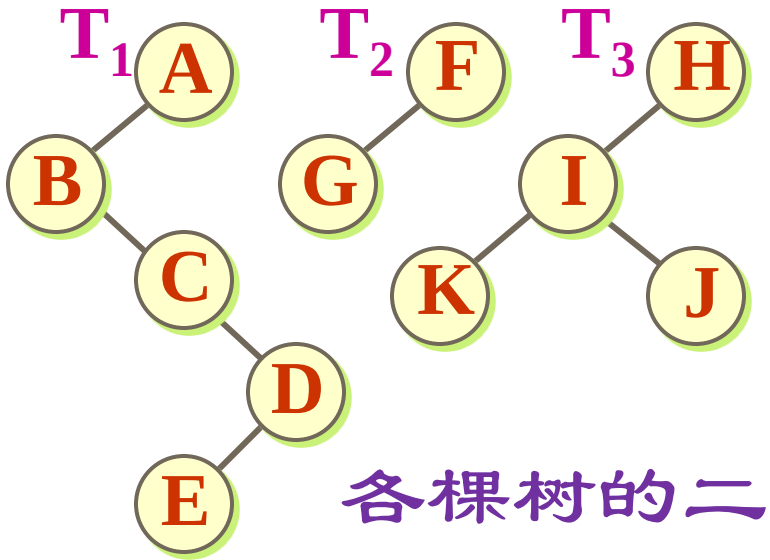
- 将一般树化为二叉树表示就是用树的子女-兄弟表示来存储树的结构。
- 森林与二叉树表示的转换可以借助树的二叉树表示来实现。



3 棵树的森林



森林的二叉树表示



各棵树的二叉树表示

森林转化成二叉树的规则

森林转换为二叉树

(1) 将森林中的每棵树转换成二叉树；

(2) 从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连起来后，此时所得到的二叉树就是由森林转换得到的二叉树。

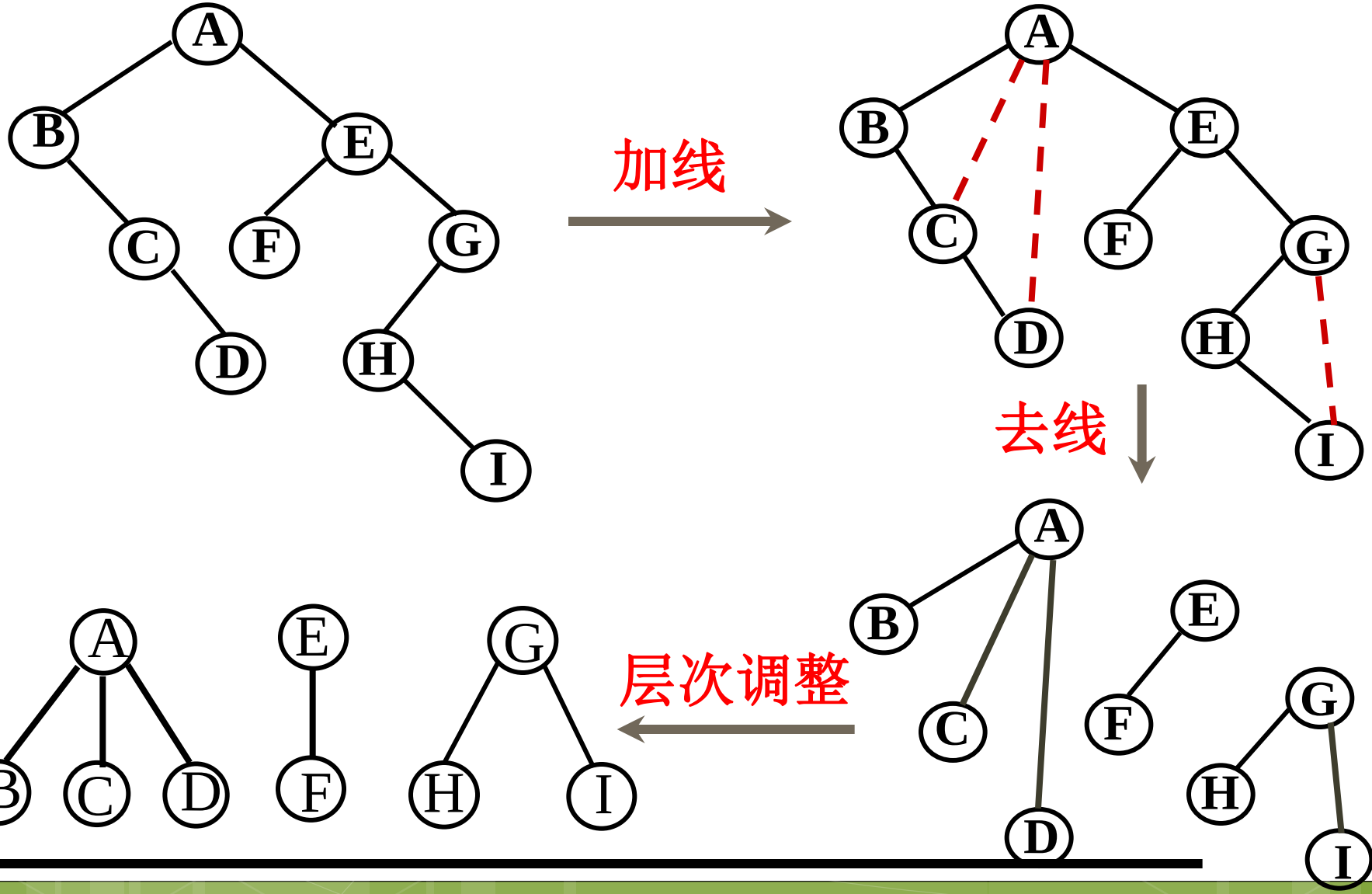
二叉树转换为树或森林

(1) 加线——若某结点x是其双亲y的左孩子，则把结点x的右孩子、右孩子的右孩子、……，都与结点y用线连起来；

(2) 去线——删去原二叉树中所有的双亲结点与右孩子结点的连线；

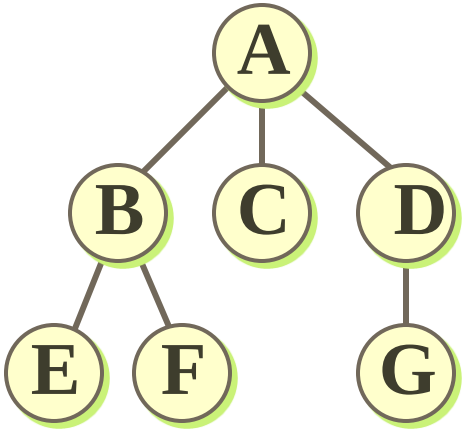
(3) 层次调整——整理由(1)、(2)两步所得到的树或森林，使之层次分明。

5.6 树、森林与二叉树的转换

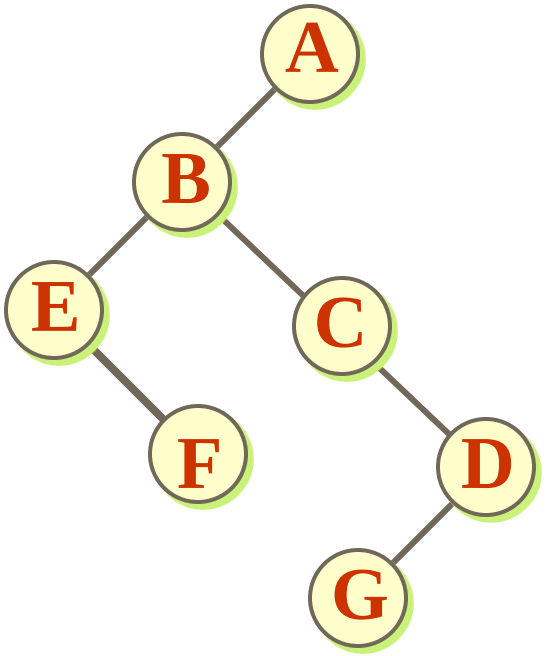


5.7 树与森林的遍历

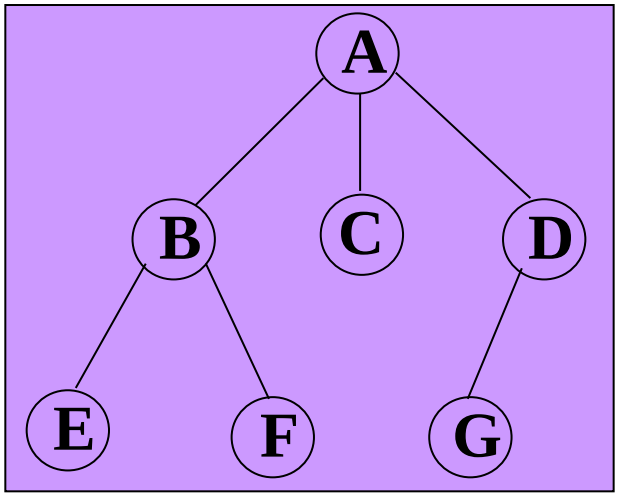
- 深度优先遍历
  - 先根次序遍历
  - 后根次序遍历
- 广度优先遍历



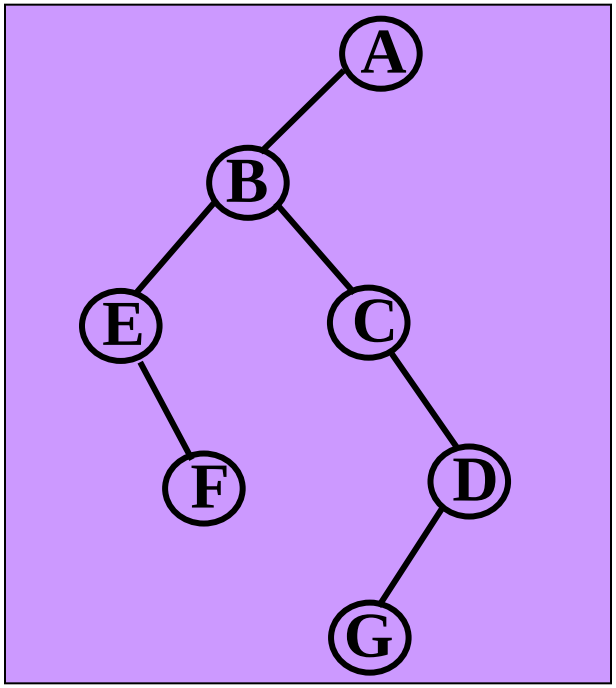
树的二叉树表示



5.7.1 树的深度优先遍历



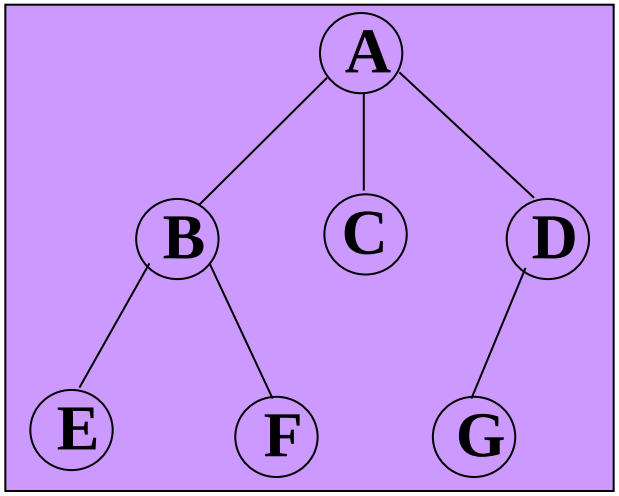
前序遍历 → ABEFCDG



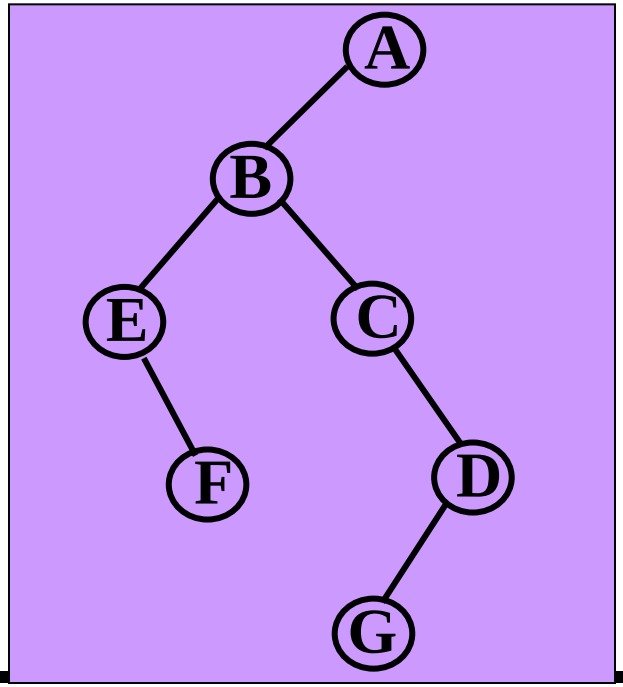
树的前序遍历等价于二叉树的前序遍历！

前序遍历 → ABEFCDG

5.7.1 树的深度优先遍历



后序遍历 → **EFBCGDA**



树的后序遍历等价于二叉树的中序遍历！

中序遍历 → **EFBCGDA**

树的先根次序遍历的递归算法

```
template <class T>
void Tree<T>::
PreOrder ( void (*visit) (BinTreeNode<T> *t) ) {
//以当前指针current为根, 先根次序遍历
    if (!IsEmpty ()) {          //树非空
        visit (current);                    //访问根结点
        TreeNode<T> *p = current;     //暂存当前指针
        current = current->firstChild;  //第一棵子树
        while (current != NULL) {
            PreOrder (visit);          //递归先根遍历子树
            current = current->nextSibling;
 }
        current = p;                        //恢复当前指针
    }
}
```

树的后根次序遍历的递归算法

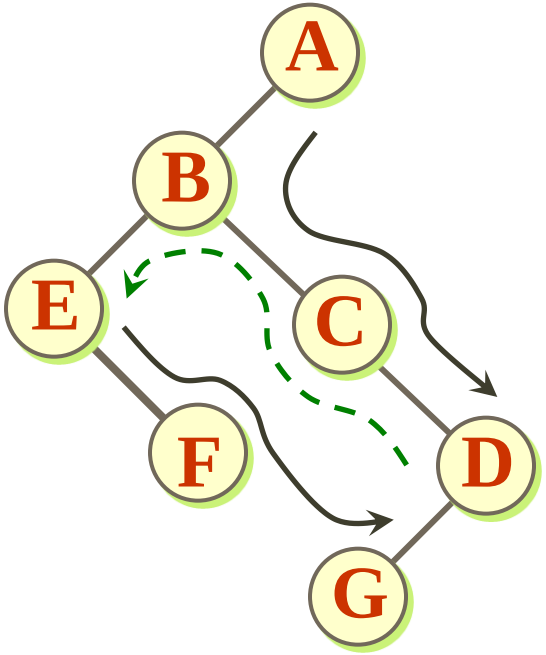
```
template <class T>
void Tree<T> ::
PostOrder (void (*visit) (BinTreeNode<T> *t)) {
//以当前指针current为根, 按后根次序遍历树
    if ( ! IsEmpty () ) {                          //树非空
        TreeNode<T> *p = current;     //保存当前指针
        current = current->firstChild;   //第一棵子树
        while (current != NULL) {        //逐棵子树
            PostOrder (visit);
            current = current->nextSibling;
        }
        current = p;                              //恢复当前指针
        visit (current);                          //访问根结点
    }
};
```

5.7.2 广度优先（层次次序）遍历

- 按广度优先次序遍历树的结果

  ABCDEFG

- 遍历算法用到一个队列。



树的广度优先遍历算法p2-1

```
template <class T>
void Tree<T>::
LevelOrder(void (*visit) (BinTreeNode<T> *t) ) {
//按广度优先次序分层遍历树,树的根结点是
//当前指针current。
Queue<TreeNode<T>*> Q;
TreeNode<T> *p;
if (current != NULL) {        //树不空
        p = current;                        //保存当前指针
        Q.EnQueue (current);          //根结点进队列
        }
```

树的广度优先遍历算法p2-1

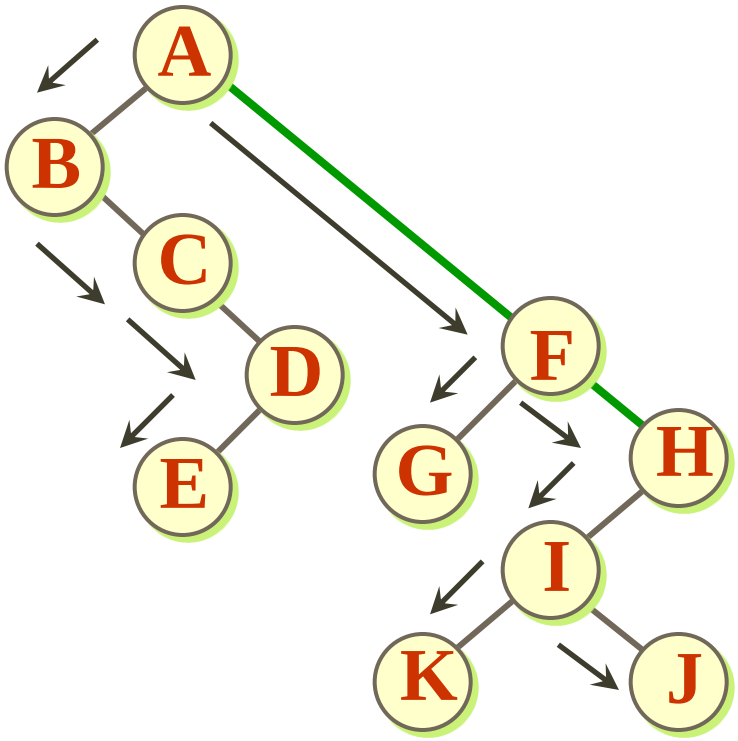
```
    while (!Q.IsEmpty ()) {
            Q.DeQueue (current);        //退出队列
            visit (current);            //访问之
        current = current->firstChild;
        while (current != NULL) {
            Q.EnQueue (current);
            current = current->nextSibling;
    }
            current = p; //恢复算法开始的当前指针
    }
};
```

森林的遍历

- 森林的遍历也分为深度优先遍历和广度优先遍历，深度优先遍历又可分为先根次序遍历和后根次序遍历。

1.深度优先遍历

（1）森林的先根次序遍历

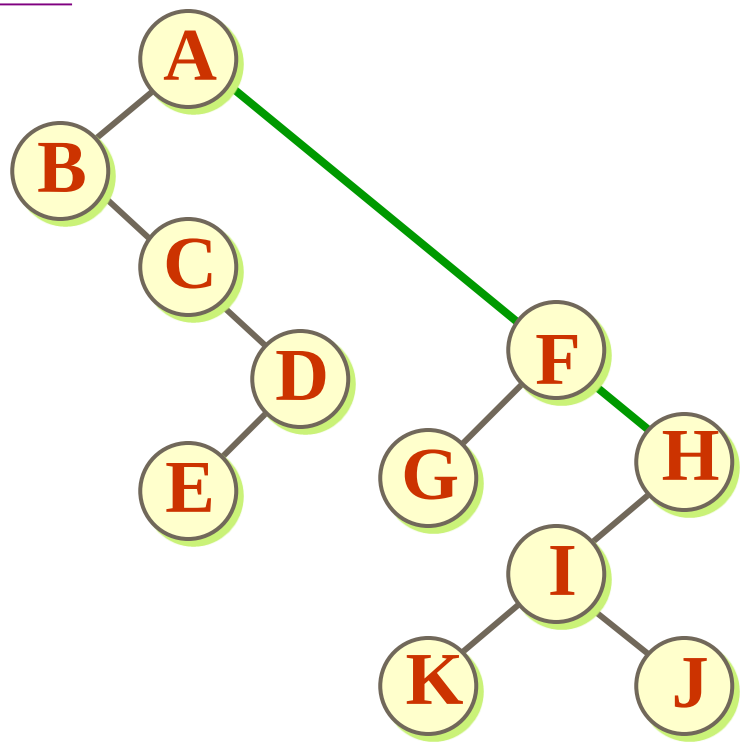


- 森林的先根次序遍历的结果序列

  ABCDE FG HIKJ

- 这相当于对应二叉树的前序遍历结果。

(2) 森林的后根次序遍历



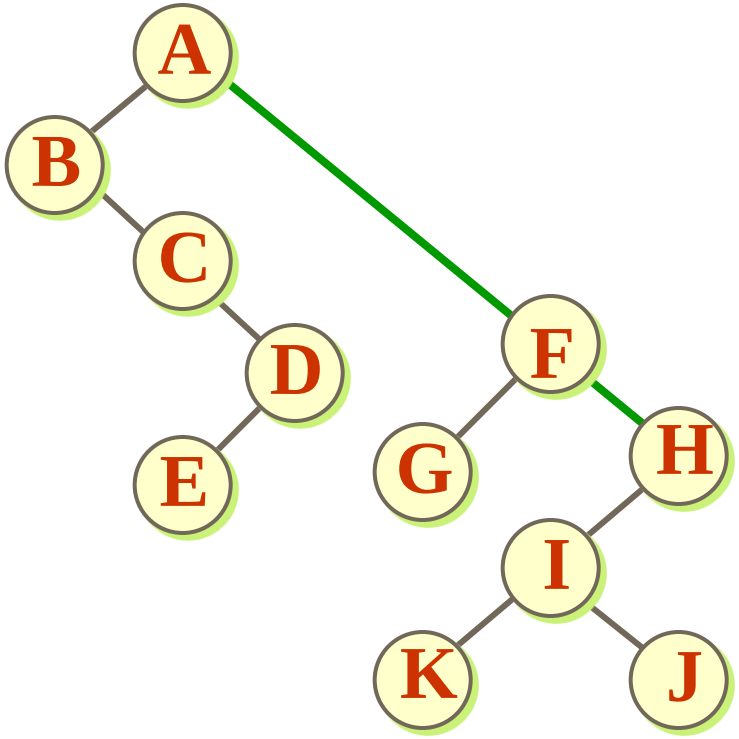
- 森林的后根次序遍历的结果序列

  BCEDA GF KIJH

- 这相当于对应二叉树中序遍历的结果。

森林的广度优先遍历（层次序遍历）

- 若森林 *F* 为空，返回；否则
  - ✓ 依次遍历各棵树的根结点；
  - ✓ 依次遍历各棵树根结点的所有子女；
  - ✓ 依次遍历这些子女结点的子女结点；
  - ✓ ……



AFH BCDGIJ EK