数据结构与算法

计算机学院

朱晨阳 副教授

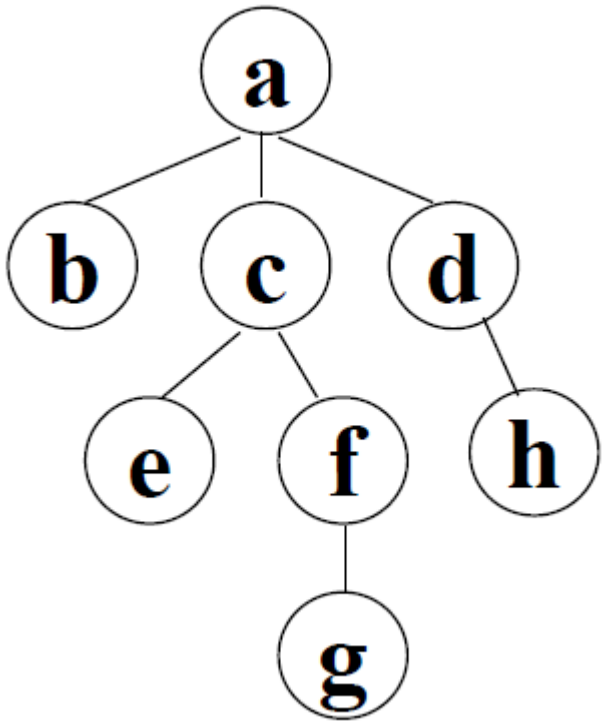
zhuchenyang07@nudt.edu.cn

树和二叉树

1. **树的概念**
2. 二叉树
3. 二叉树、树、树林的遍历
4. BinaryTree结构及递归遍历算法
5. 非递归遍历算法及线索化二叉树

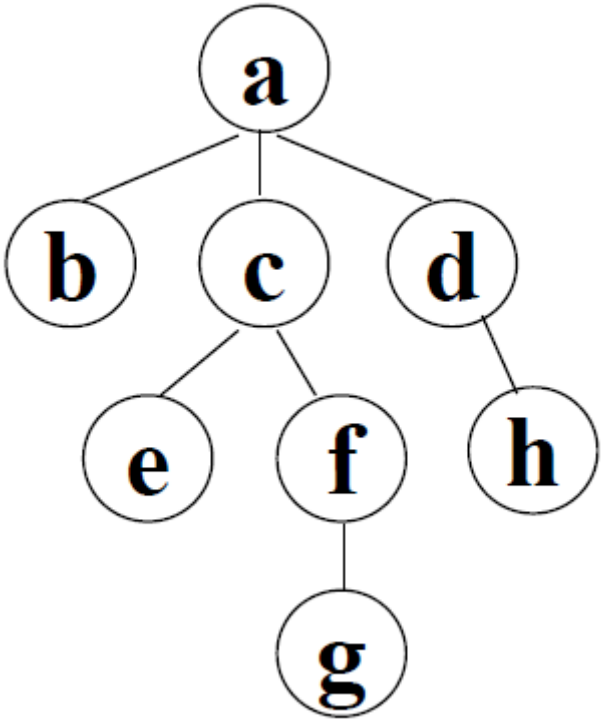
树的概念

- **基本术语**
  - 根：没有前驱的结点
  - 儿子：结点的后继是该结点的儿子
  - 父亲/父母：若s是p的儿子，则p是s的父亲/父母
  - 度：一个结点的儿子个数为该结点的度。
  - 叶结点：没有儿子的结点称为叶结点。
  - 分支结点：有儿子的结点。



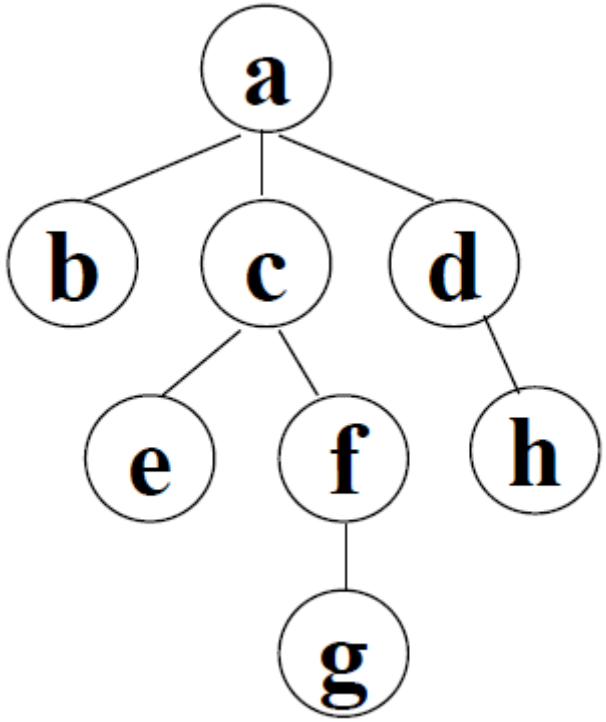
树的概念

- **基本术语**
  - 兄弟：有相同父亲的结点。
  - 子孙：根为r的树的所有结点都是r的子孙，除r之外的所有结点是r的真子孙。
  - 祖先：从根r到结点p的路径上的所有结点都是p的祖先，其中除p之外是p的真祖先。
  - 有序树：树中各结点的儿子是有序的。



树的概念

- **基本术语**
  - 层数：根结点层数为1，其它结点的层数是其父结点的层数加1。
  - 高度(深度)：树中结点的最大层数。空树的高度为0。
  - 树林(森林)：n(n>=0)个互不相交的树的集合。



树和二叉树

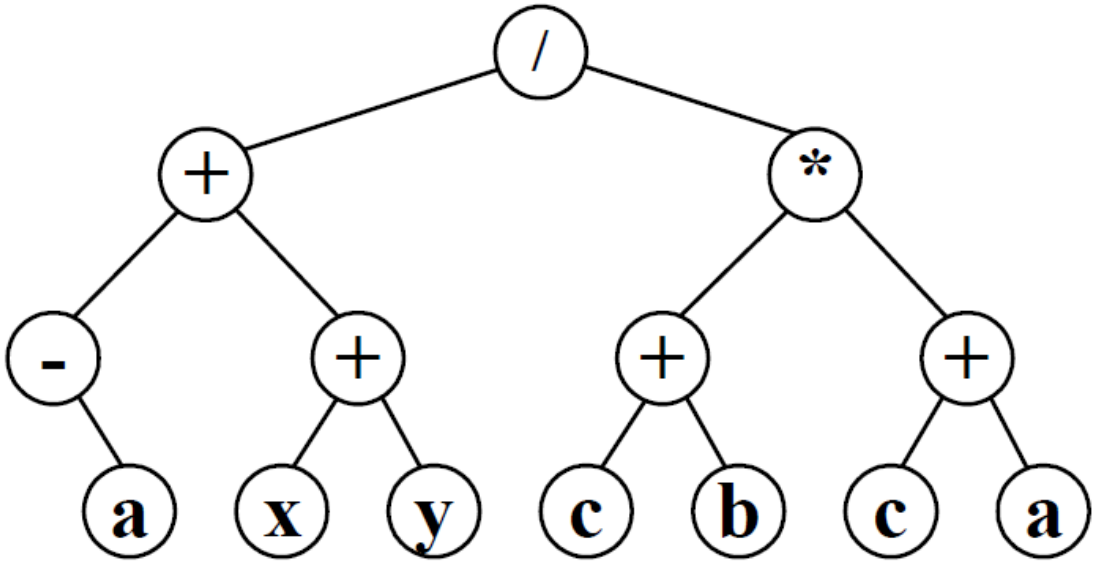
- 7.2 二叉树
  - 7.2.1 二叉树的概念
  - 7.2.2 二叉树的性质
  - 7.2.3 二叉树的存储方式
  - 7.2.4 树(树林)与二叉树的相互转换

二叉树的概念

- 二叉树的每个结点至多有2个子女，而且有左、右之分。
- 二叉树的递归定义：二叉树由结点的有限集合构成，这个有限集合或者为空，或者由一个根结点以及两棵不相交的分别称作这个根的左子树和右子树的二叉树组成。
- 二叉树的存储结构简单，算法实现简单。

二叉树的概念

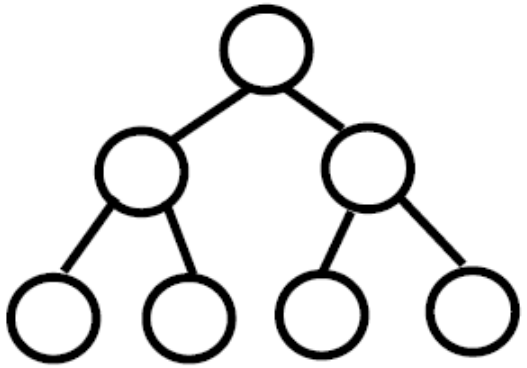
- 二叉树表示算术表达式：分支结点对应运算符，叶结点对应操作数。



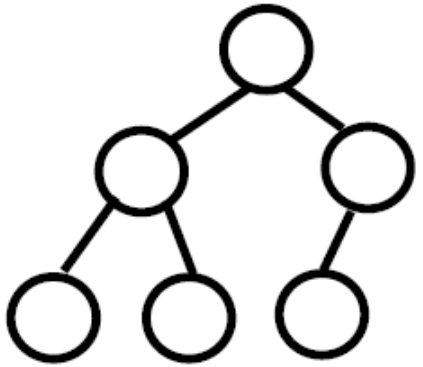
((-a)+(x + y)) / ((c+b)*(c+a))

二叉树的概念

- 满二叉树：一棵深度为h且有2^h-1个结点的二叉树。(每个结点的度为0或2，且度为0的结点只出现在最后一层)



- 完全二叉树：深度为h且有n个结点的二叉树，当且仅当每个结点都与深度为h的满二叉树中编号从1~n的结点一一对应时，称之为完全二叉树。



二叉树的性质

- 性质1: 任何一棵含有n(n>0)个结点的二叉树恰有n-1条边。
  - 任何一棵含有n个结点的二叉树，除了根结点外的其它n-1个结点都有且只有一条边与其父母结点相连。
- 性质2：深度为h的二叉树至多有2^h-1个结点(h>=0).
  - 二叉树的第i层最多有2^{i-1}个结点。
- 性质3：设二叉树的结点数为n，深度为h，则$\lceil \log_2(n+1) \rceil \le h \le n$。

二叉树的性质

- 性质4：如果一棵有n个结点的完全二叉树的结点，按层次序编号(每层从左至右)，则对任意结点i(1≤i≤n):
  - (1)若2i>n，则结点i无左子女；否则，结点2i为结点i的左子女。
  - (2)若2i+1>n，则结点i无右子女；否则，结点2i+1为结点i的右子女。
  - (3)若i=1，则结点i为二叉树的根结点；若i>1，则结点[i/2]为其父母结点。

二叉树的性质

- 证：证(1)和(2)，用归纳法。
- 当i=1时， (1)和(2)显然成立。
- 假设i=k时(1)和(2)成立。当i=k+1时：
- 若结点k+1有左儿子，则结点k有左儿子和右儿子(否则不是完全二叉树)，其编号分别为2k和2k+1(归纳假设)，所以结点k+1的左儿子的编号为(2k+1)+1 = 2(k+1)，从而，若结点k+1有右儿子，则其编号为2(k+1)+1，所以当i=k+1时(1)和(2)成立。
- (3)是(1)和(2)推论。

二叉树的性质

- 性质5：如果一颗二叉树每个分支结点都两个儿子，设叶结点个数为n，则分支结点的个数为n-1。

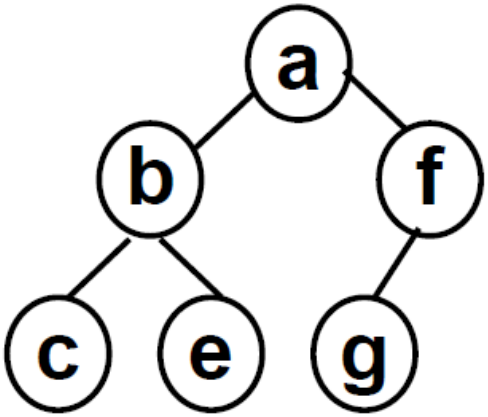
证明：设分支结点的个数为x，则该二叉树有2x条边，结点总数为n+x。因为二叉树的边数为其结点数减1，所以2x=n+x-1，所以x=n-1。

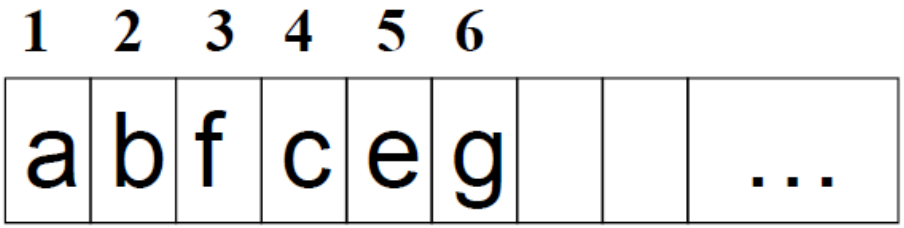
二叉树的性质

- 性质6：设二叉树中叶结点个数为n0, 只有一个儿子的分支结点个数为n1, 有两个儿子的分支结点个数为n2，边的数量为e。则：
  - e=2*n2+n1
  - e=(n0+n1+n2)-1
  - n2=n0-1

二叉树的顺序存储

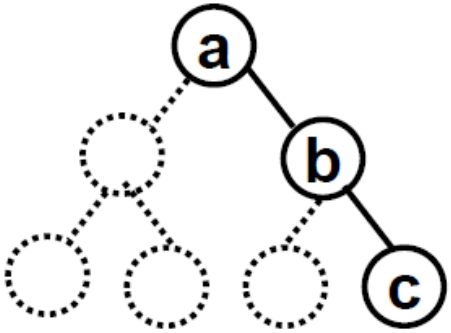
- 完全二叉树的顺序存储（由性质4）
  - 结点相对地址=(结点编号–1)×每个结点所占单元数





二叉树的顺序存储

- 非完全二叉树的顺序存储
  - 补设一些虚结点，使它成为一棵完全二叉树，对这棵通过补设虚结点而形成的完全二叉树按完全二叉树的顺序存储方式存储。(虚结点需要对应存储位置，并设立虚结点标志)



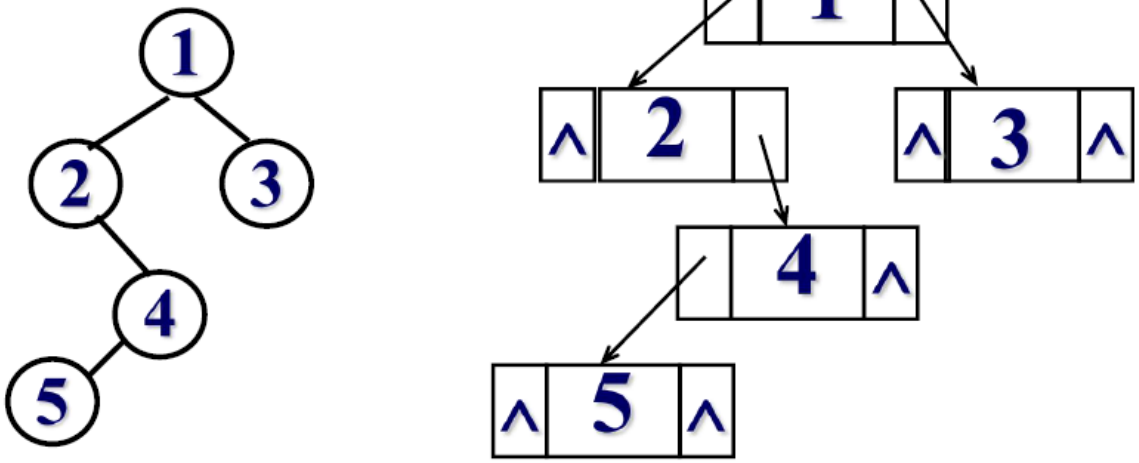
1	2	3	4	5	6	7	…
a	x	b	x	x	x	c	…

一般二叉树的连续存储(x表示虚结点)

二叉树的链接存储

- **LeftChild-RighChild表示法(二指针式)**

结点结构**(LeftChild, data, RightChild)**



二叉树的链接存储

- LeftChild-RighChild表示法(二指针式)
  结点结构(LeftChild, data, RightChild)

```
Struct BinaryTreeNode
{   T data;
    struct BinaryTreeNode * LeftChild,*RightChild;
}
void InitBinaryTreeNode(BinaryTreeNode *btree, T e, BinaryTre
    eNode *l, BinaryTreeNode *r)
{   btree->LeftChild = l;
    btree->RightChild = r;
    btree->data = e;
}
```

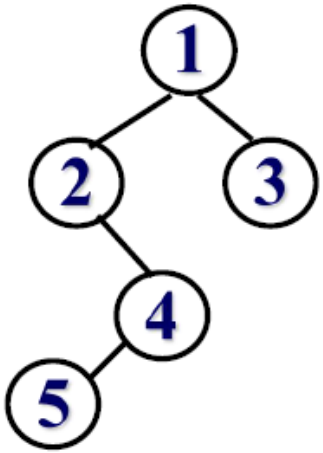
二叉树的链接存储

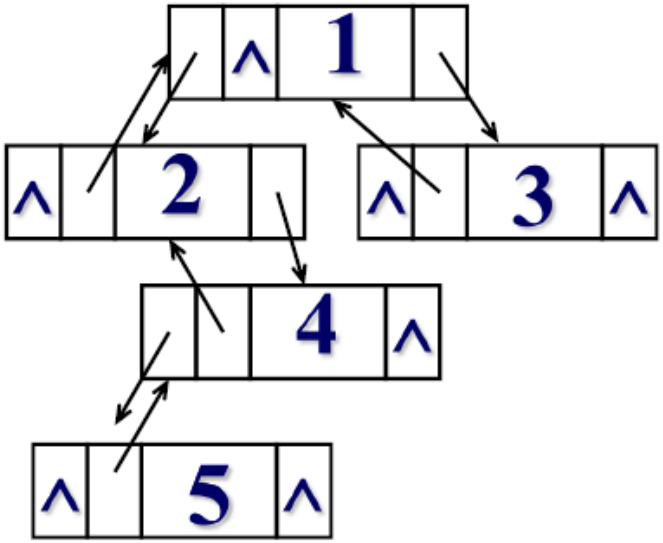
```
BinaryTreeNode *CreateBTreeNode(T item,BinaryTreeNo
    de *lptr, BinaryTreeNode *rptr)
{
    BinaryTreeNode *p;
    p = (BinaryTreeNode *)malloc(sizeof(BinaryTreeNode));
    if(p==NULL)
        printf("Memory allocation failure!\n");
    else
        InitBinaryTreeNode(p, item, lptr, rptr);
    return p;
}
```

二叉树的链接存储

- 三重链表示(三指针式)

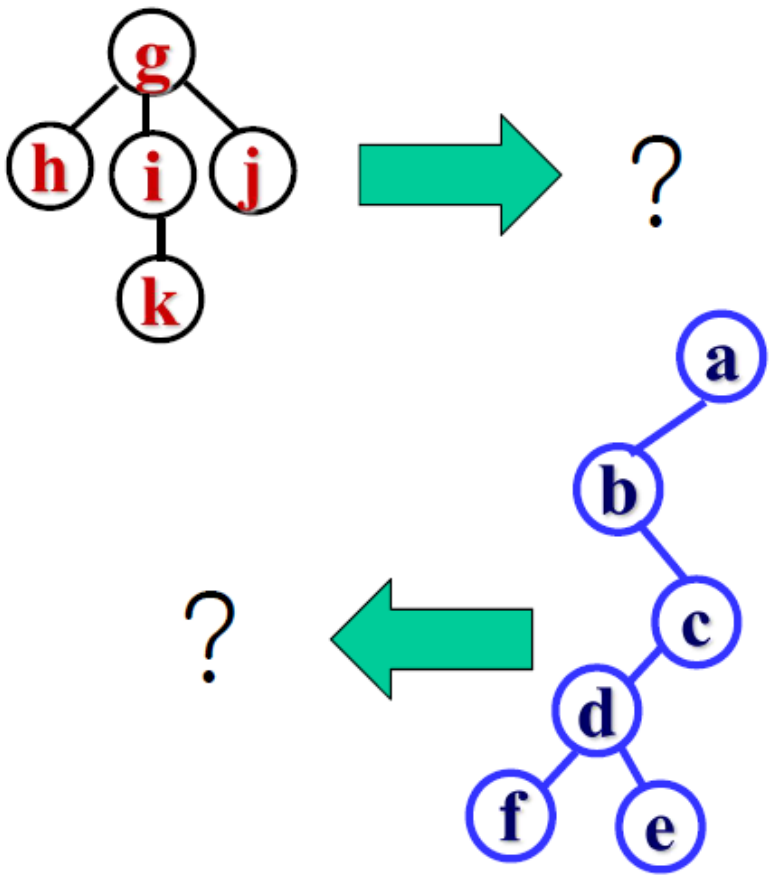
结点结构(LeftChild, Parent, Data, RightChild)





树与二叉树的相互转换

- 树与二叉树的相互转换
- 树的根与转换二叉树的根对应
- 树中结点x的第一个儿子与转换二叉树中结点x的左儿子对应
- 树中结点x的下一个兄弟与转换二叉树中结点x的右儿子对应



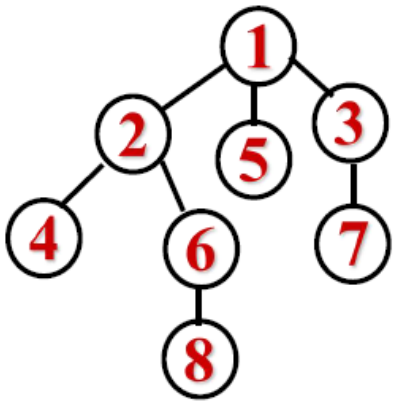
树与二叉树的相互转换

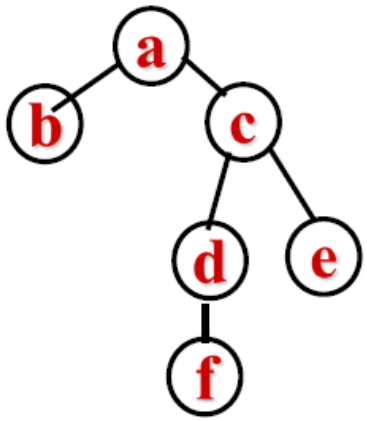
- 树林转换为二叉树
  定义树林$F=(T_1,T_2,\ldots,T_n)$到二叉树$B(F)$的转换：
  - 若n=0，则$B(F)$为空的二叉树
  - 若n>0，则$B(F)$的根是T_1的根W_1，$B(F)$的左子树是$B(T_{11}, T_{12}, \ldots, T_{1m})$，其中$T_{11},T_{12},\ldots,T_{1m}$是$W_1$的子树；$B(F)$的右子树是$B(T_2,\ldots,T_n)$

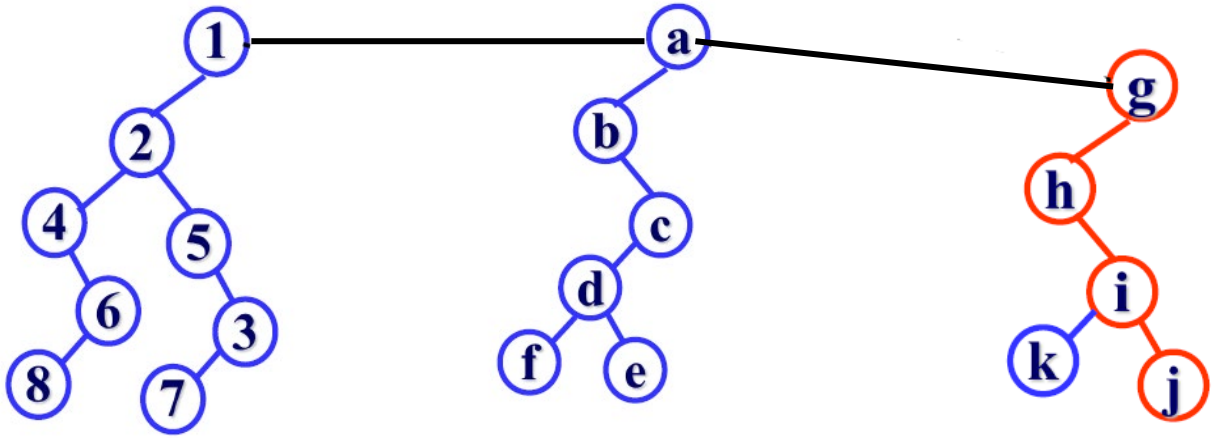
大儿子->左儿子

i号儿子->i-1号儿子的右儿子 （弟->右儿子）

树与二叉树的相互转换







树与二叉树的相互转换

- 设B是一棵二叉树，r是B的根，L是r的左子树，R是r的右子树，则对应于B的树林F(B)的定义为：
  - 若B为空，则F(B)是空的树林；
  - 若B不为空，则F(B)是一棵树T1加上树林F(R)，其中T1的根为r，r的子树为F(L)。
  - 左儿子->大儿子
  - 右儿子->弟

树和二叉树

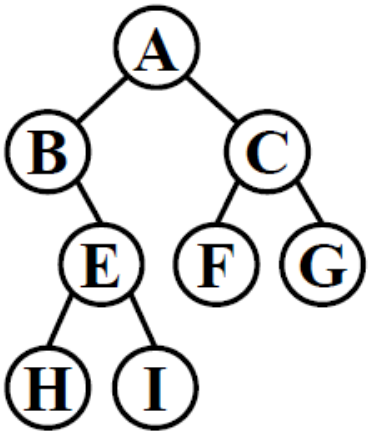
- 树：什么是根、子结点、父结点、叶结点、分支结点、兄第、有序树？
- 二叉树：什么是左子树、右子树、满二叉树、完全二叉树？
- n个结点的树/二叉树有多少条边？深度为h的满二叉树有多少结点？n个结点的二叉树最少多少层？层序编号的完全二叉树父子结点间编号有何关系？有n个叶结点的二叉树，有多少度为2的分支结点？
- 二叉树如何存储？
- 树如何转换为二叉树？
- 如何遍历二叉树、树、森林？
- 什么是线索化二叉树？

树和二叉树

- 7.1 树的概念
- 7.2 二叉树
- 7.3 二叉树、树、树林的遍历
- 7.4 BinaryTree结构及递归遍历算法
- 7.5 非递归遍历算法及线索化二叉树

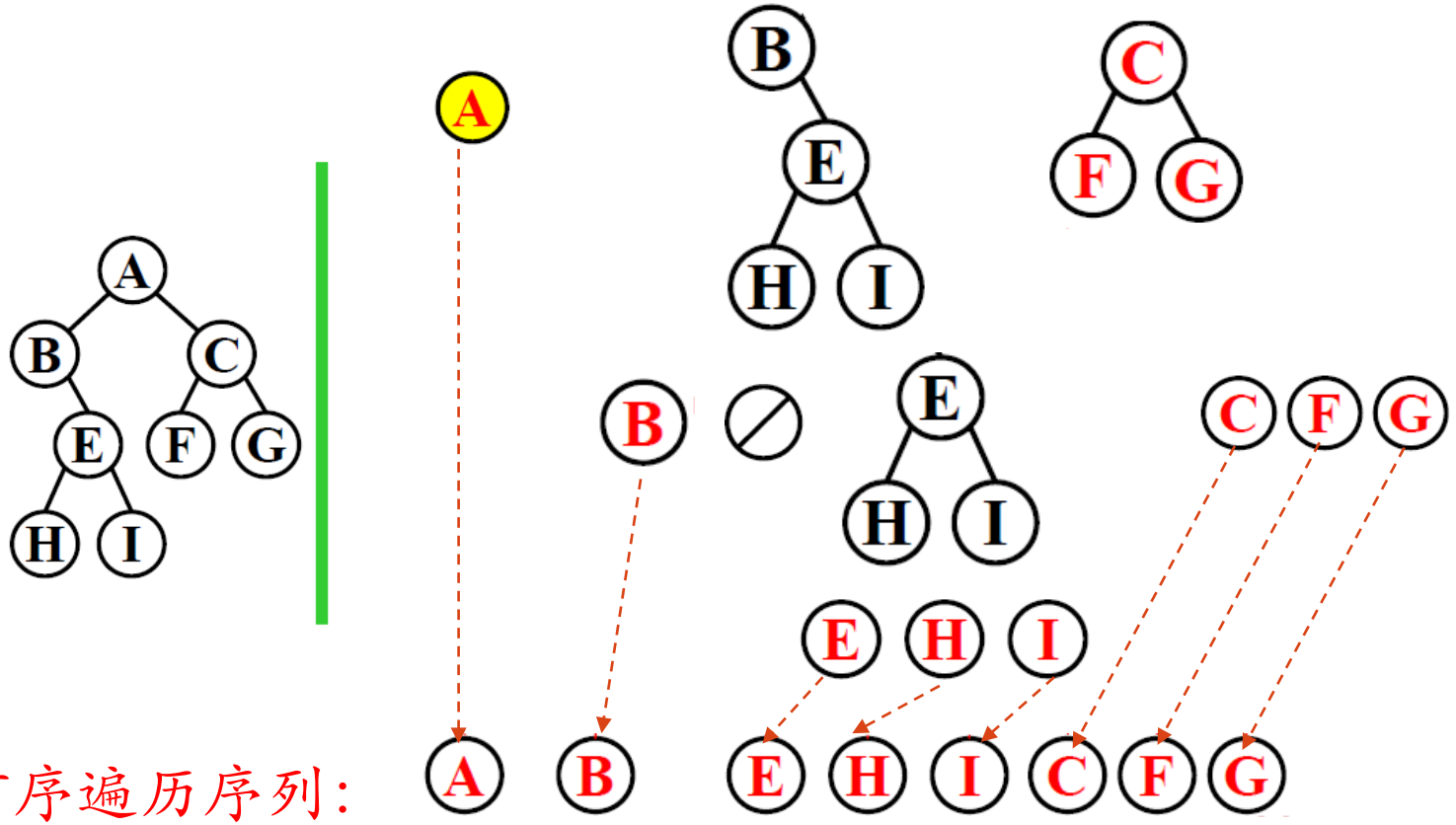
遍历

- 何谓遍历?
  - 按照一定方式访问树中的结点，每个结点恰好被访问一次。
- 四种遍历方式:
  - 前序遍历、中序遍历、后序遍历、层序遍历



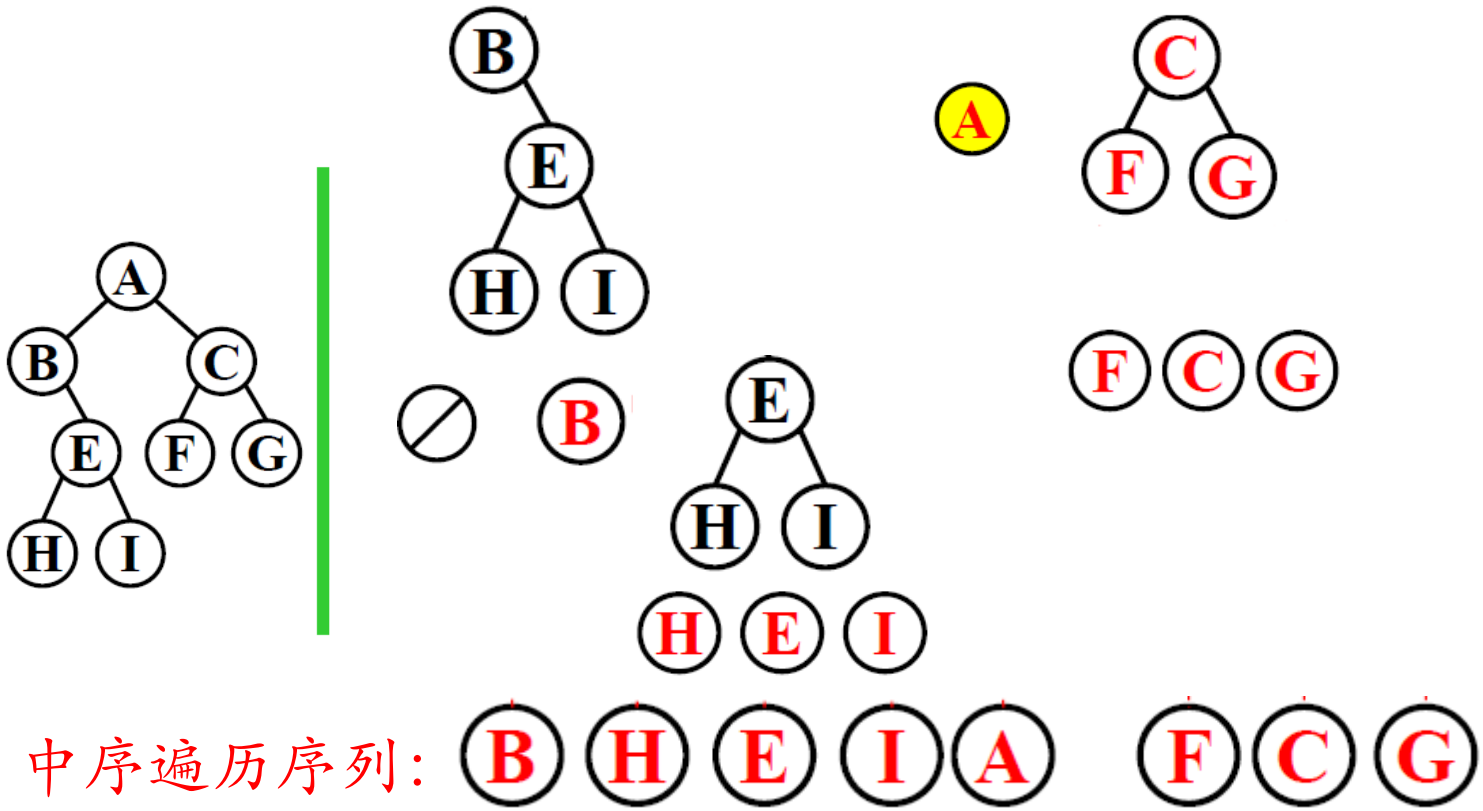
前序遍历二叉树

- 前序遍历：
  - (1)访问根结点；(2)前序遍历左子树；(3)前序遍历右子树



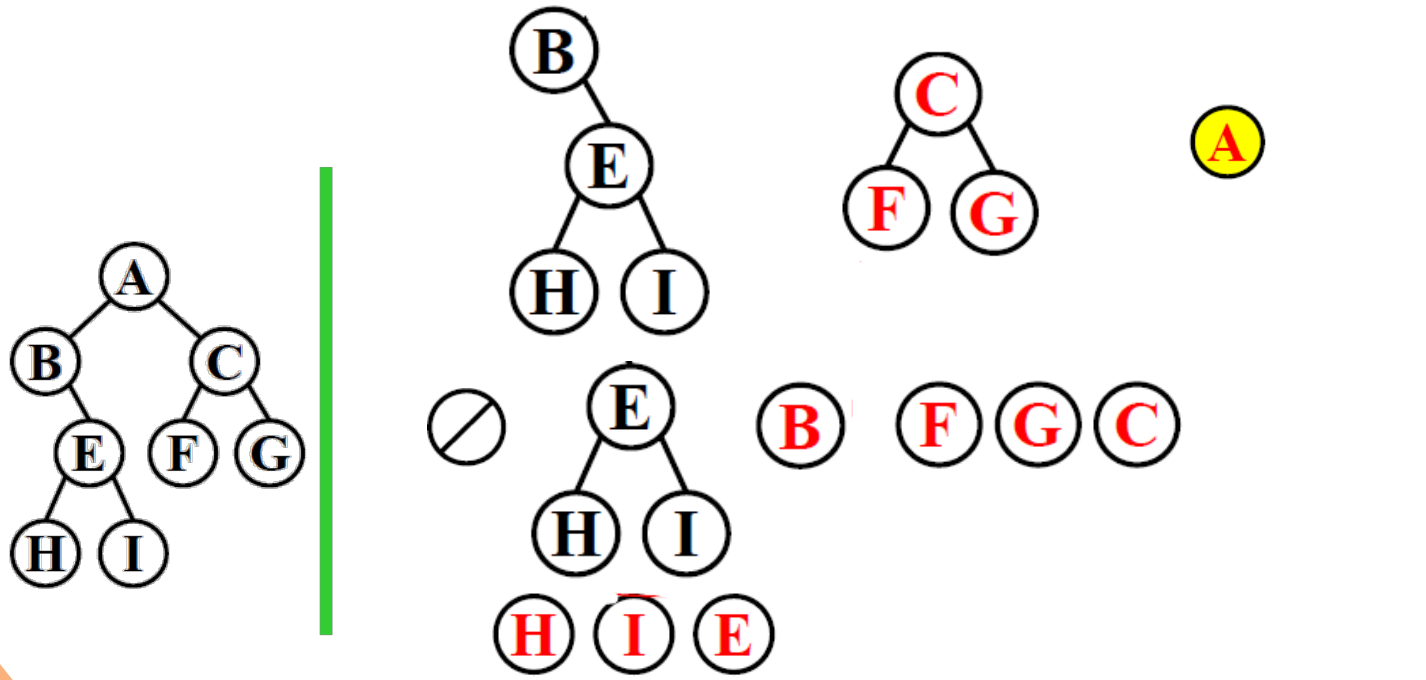
中序遍历二叉树

- 中序遍历：
  - (1)中序遍历左子树；(2)访问根结点；(3)中序遍历右子树



后序遍历二叉树

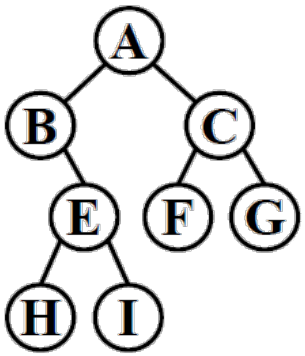
- 后序遍历：
  - (1)后序遍历左子树；(2)后序遍历右子树； (3)访问根结点；



后序遍历序列：H I E B F G C A

层序遍历二叉树

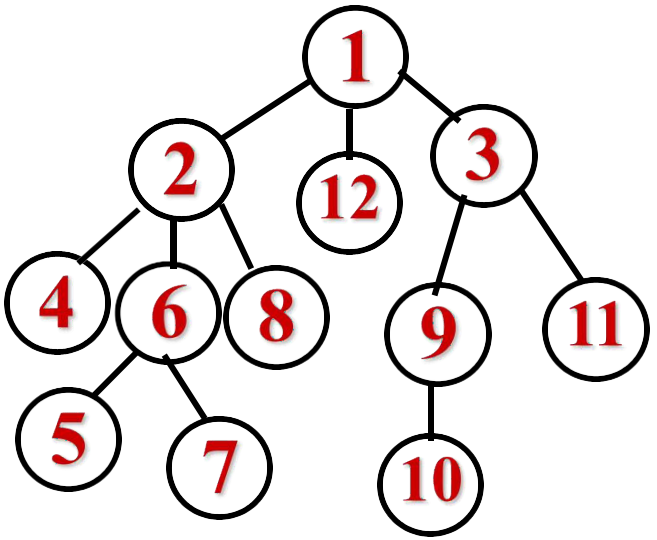
- 层序遍历：
  - (1)从左到右访问第1层；(2)从左到右访问第2层； (3)……；



A B C E F G H I

树的遍历

- 树的先根遍历
  - (1)访问根
  - (2)依次先根遍历根的各个子树
- 树的后根遍历
  - (1)依次后根遍历根的各个子树，
  - (2)访问根
- 树的层序遍历
  - 依次(从上到下）访问每层（从左到右）



先根： 1 2 4 6 5 7 8 12 3 9 10 11

后根： 4 5 7 6 8 2 12 10 9 11 3 1

由遍历结果还原出二叉树

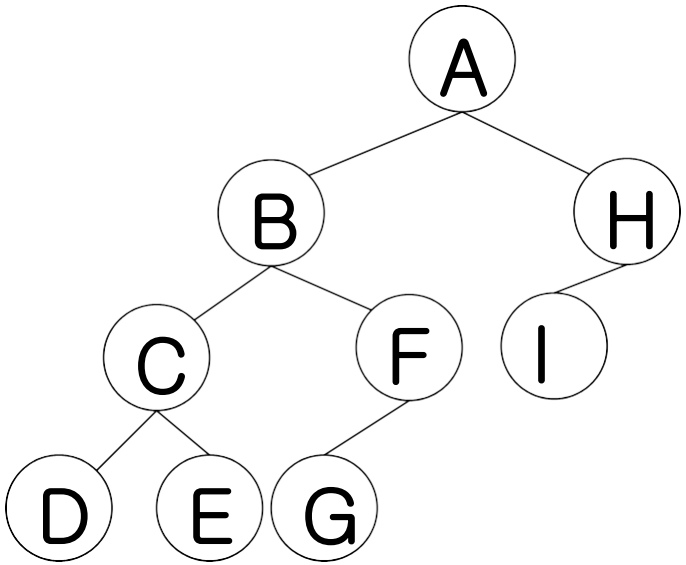
- 哪些组合可还原出二叉树(不能则给示例)
  - 前序+中序
  - 前序+后序
  - 前序+层序
  - 中序+后序
  - 中序+层序
  - 后序+层序

遍历算法

```
void PrintPostTravel(TNode* t)
{
    if(t==NULL) return;
    if(t->left) PrintPostTravel(t->left);
    if(t->right) PrintPostTravel(t->right);
    printf("%c", t->data);
}
```

非递归的遍历算法

- 效率比递归算法高。
- 从根起，顺左分枝往下，依次访问每个结点，直到遇到没有左分枝的结点为止。然后，回到已访问过的最近一个有右儿子的且该右儿子尚未被访问过的结点，同样地顺着该结点的右子树的左分枝往下依次访问每个结点。如此反复，直到所有结点均已处理完。



非递归的遍历算法

- 在访问完某结点后，不能立即丢弃它，因为遍历完它的左子树后，要从它的右儿子起遍历，所以在访问完某结点后，应保存它的指针，以便以后能从它找到它的右儿子。
- 由于较先访问到的结点较后才重新利用，故应将访问到的结点按栈的方式保存。

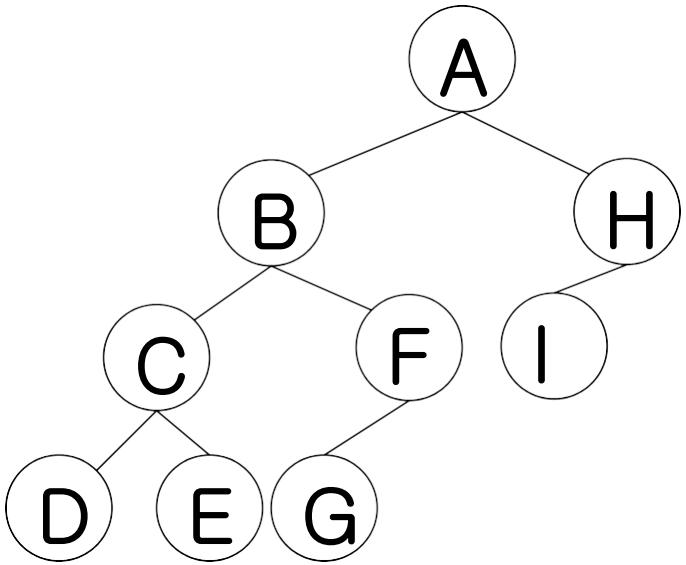
非递归的遍历算法

- stack:{}
- visit sequence:

if TOP != NULL,
then visit TOP, push left(TOP)

if TOP==NULL
then POP, new TOP=right(new TOP)

TOP==NULL and top==0
END



非递归的遍历算法

- 前序遍历非递归算法：

```
push(root);
while( stack[top]!=NULL or top>0 )
{
    if (stack[top]!=NULL)
    { visit(stack[top]); push(left(stack[top])); }
    else
    {pop; stack[top]=right(stack[top]);}
}
```

非递归的遍历算法

- 中序遍历非递归算法与前序遍历相似。
- 在前序遍历中，每搜索到一个结点，就访问它，并将它推进栈。
- 在中序遍历中，搜索到结点时并不能立即访问，只是将它推进栈，等到左分枝搜索完后再从栈中弹出并访问之。由此可知，中序遍历的非递归算法，只需在前序遍历非递归程序的基础上，将访问结点语句调到出栈语句后即可。

非递归的遍历算法

```
/*中序遍历非递归算法*/
void lNr(BinaryTreeNode *t)
{
    BinaryTreeNode *stack[100];
    int top=0;              // “数组 + top” 模拟栈的实现
    stack[0]=t;             // root 入栈
    do
    {
        while(stack[top]!=NULL)    // 左子节点依次入栈
            stack[++top]=stack[top]->LeftChild;
        if(top>0)
        {       // 此时到达树的最左下子节点
            printf("%c", stack[--top]->data); // 访问当前节点
            stack[top]=stack[top]->RightChild; // 当前节点的右子
    节点入栈
        }
    }while(top>0 || stack[top]!=NULL);
}
```

非递归的遍历算法

```
/*后序遍历非递归算法*/
void lrN(BinaryTreeNode *t)
{  BinaryTreeNode *stack[100];
   unsigned tag[100];
   unsigned top=0;
   stack[0]=t; // root 入栈
   tag[0]=0;   // tag==1表示当前节点的子节点均已访问过
   do
   {    while(stack[top]!=NULL) // 左子节点依次入栈
        {       stack[++top]=stack[top]->LeftChild;
                tag[top]=0;
        } // 此时 stack[top]==NULL，实际有效的位置是 top-1
        while(tag[top-1]==1) // 如果当前节点的子节点均已入栈
                printf("%c", stack[--top]->data);
        if(top>0)
        {       stack[top]=stack[top-1]->RightChild;
                tag[top-1]=1; // 两个子节点均已入栈，标记tag
                tag[top]=0;
        }
   }while(top!=0);
}
```

为什么右节点入栈后，即可标记已访问tag？

线索化二叉树

- 二叉树是非线性结构，树中的结点可有多个(≤2)后继。但若按某种方式遍历，遍历结果序列就变为线性结构，每个结点就有了唯一的前趋与后继。有时需知道二叉树结点在某种遍历方式下的前趋与后继。这就是线索化的目的
- 一棵具有n个结点的二指针式二叉树，有n+1个空链域(总链域数为2n)。这些空链域有时可用来存放一些有用的信息。

线索化二叉树

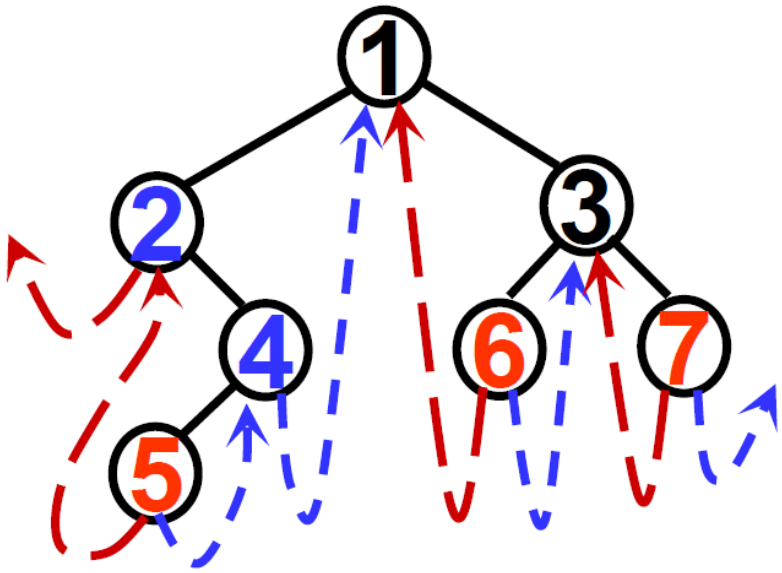
- 用空链域存放结点在某种遍历方式下的前趋或后继指针：
  - 对任一结点，用空左链域存放它的前驱的指针，而用空右链存放它的后继的指针；
  - 若某链域不空，则不存储这种前驱与后继指针。
  - 优点：既保持了原树的结构，又利用空链表示了部分前驱与后继关系。
- 称这种使树中结点空链域存放前驱或后继信息的过程为线索化，被线索化了的树为线索树。

线索化二叉树

- 树的遍历方式有四种，故有四种意义下的前趋与后继。相应有四种线索化方法和线索树：
  - 前序线索化/树
  - 中序线索化/树
  - 后序线索化/树
  - 层序线索化/树
- 按某种次序将二叉树线索化，只要按该次序遍历二叉树，在遍历过程中用线索取代空指针。

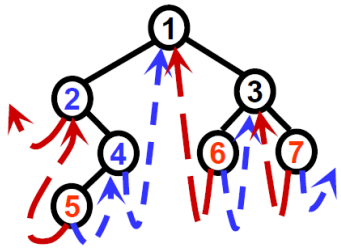
线索化二叉树

- 假设已经对一颗树进行中序线索化，如何遍历该树？
  - 如何找到当前节点的前驱
  - 如何找到当前节点的后继



线索化二叉树

```
/*在中序(对称序)线索化树中寻找指定结点p的中序后继*/
void Inordernext(ThreadedBTNode *p,ThreadedBTNode **q)
//将*p的中序后继的指针放入*q
{
   if(p->rtag==1)                    //p 没有右子节点
        *q=p->RightChild;            //右指针指向了后继
   else
   {
        *q=p->RightChild;            //到达右子树
        while((*q)->ltag==0)         // 右子树的最左下子节点
              *q=(*q)->LeftChild;
   }
}
```



线索化二叉树

```
/*对中序(对称序)线索化树进行中序遍历*/
void ThreadedInTravel(ThreadedBTNode *root)
{
    if(root!=NULL) {
        p = root;
        while(p->ltag==0)       // 寻找最左下子节点
            p=p->LeftChild;
        do {
            printf("%c ", p->data);
            if(p->rtag==1)  // if 没有右子节点(但指向后继)
                p=p->RightChild; //到达后继
            else {              // 有右子节点
                p=p->RightChild;  //到达右子树
                //寻找右子树的最左下子节点，作为后继节点
                while(p->ltag==0)
                    p=p->LeftChild;
            }
        }while(p!=NULL); // 当p没有到达中序尾结点
    }
}
```

线索化二叉树

```
/*中序(对称序)线索化二叉树的递归实现*/
typedef ThreadedBTNode * pNode;          // pNode表示一个指针
void InorderThreaded(pNode root, pNode & pre)
//对以root为根的子树进行中序线索化
//pre指向当前节点的中序前驱
{   if(root!=NULL) {
        InorderThreaded(root->LeftChild, pre); // 处理左子树, 处理完之后: pre
    指向左子树的尾节点（最后一个访问的节点）

        if(pre!=NULL && pre->rtag==1)
        {   pre->RightChild=root;  } //左子树尾节点的后继是当前节点root
        if (root->LeftChild==NULL) { //当前节点的前驱是左子树的尾节点
                root->ltag=1;
                root->LeftChild = last;   }

        //在下一层设置 pre->rtag=1, 返回后被上面的if语句使用
        if (root->RightChild==NULL)   root->rtag=1;

        pre=root; // 当前节点是待访问右子树的前驱
        InorderThreaded(root->RightChild, pre);   } }
```