第六章 二叉树

part 1: 二叉树基础

张史梁
slzhang.jdl@pku.edu.cn

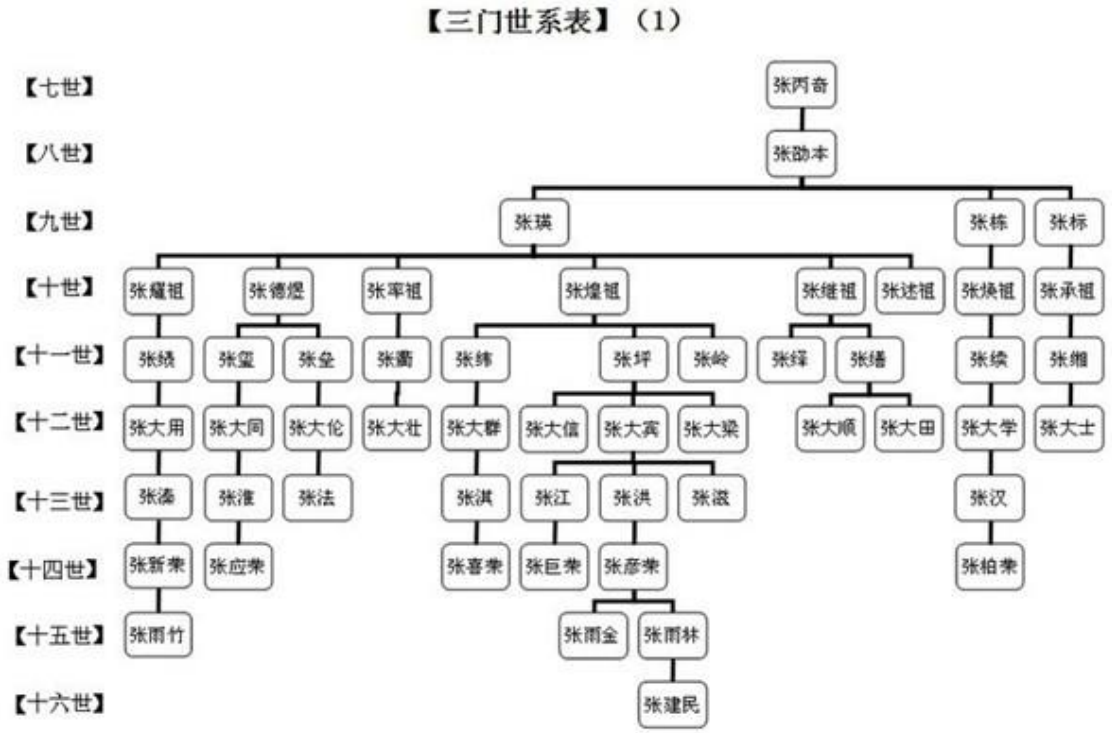
内容提要

- 二叉树基础
  - 树与二叉树的基本概念
  - 二叉树的存储结构
  - 二叉树的周游算法
  - 建立一个二叉树
- 二叉树的应用
  - 哈夫曼树
  - 二叉检索树/排序树
- 树与树林

线性结构 vs. 非线性结构

- 线性结构
  - 结构中的元素之间满足线性关系，每个内部结点（元素）都有且仅有一个前驱结点、一个后继结点
- 非线性结构
  - 至少存在一个数据元素，具有两个或两个以上的前驱或后继
    - 树形结构
    - 图结构

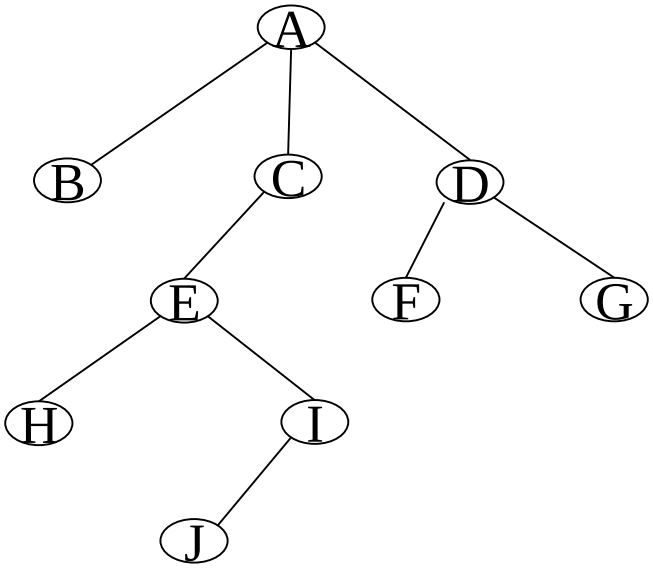
树的例子



- 政府部门：局---处---科---室
- 书籍目录：书---章----节----小节
- 物种分类：门---纲---类----科----目-----种

树的逻辑结构

- 包含n个结点的有穷集合K (n>0)，且在K上定义了一个关系 R，关系 R 满足以下条件：
  - 有且仅有一个结点$k_0 \in K$，它对于关系 R 来说没有前驱。结点k_0称作树的根；
  - 除结点k_0外K中的每个结点对于关系 R 来说都有且仅有一个前驱；
  - 除结点k_0外的任何结点$k \in K$，都存在一个结点序列$k_0,k_1,\ldots,k_s$，使得k_0就是树根，且$k_s=k$，其中有序对$<k_{i-1}, k_i> \in R\ (1 \le i \le s)$。这样的结点序列称为从根到结点k的一条路径。

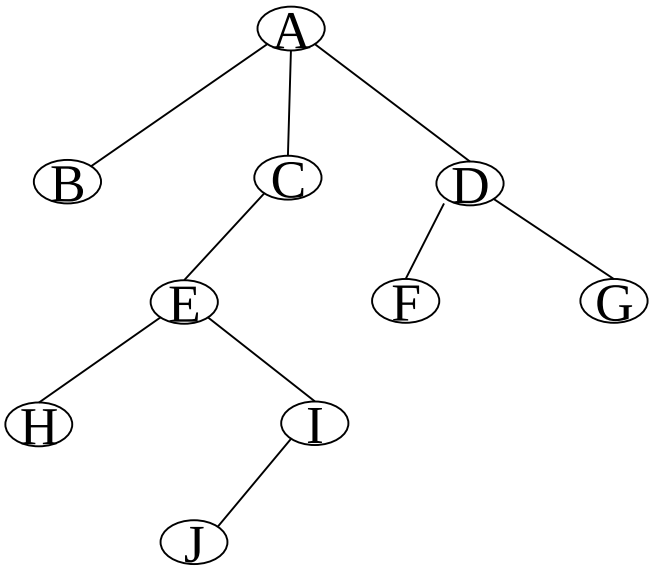


树的定义

- 树的递归定义：n(n≥0)个结点的有穷集合T，当T非空时满足：
  - 有且仅有一个特别标出的称为根的结点，
  - 除根外，其余结点分为m≥0个互不相交的非空集合$T_1,T_2,\ldots,T_m$，而且每个非空集合T_i又是一颗树，称为根结点的子树。

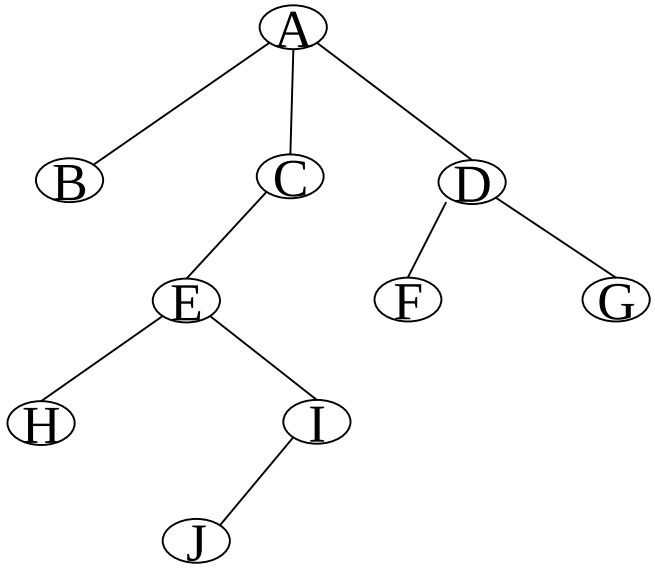
  T_1={B}, T_2={C, E, H, I, J},

  T_3={D, F, G}
- 允许不包括任何结点的树，把它称作空树。



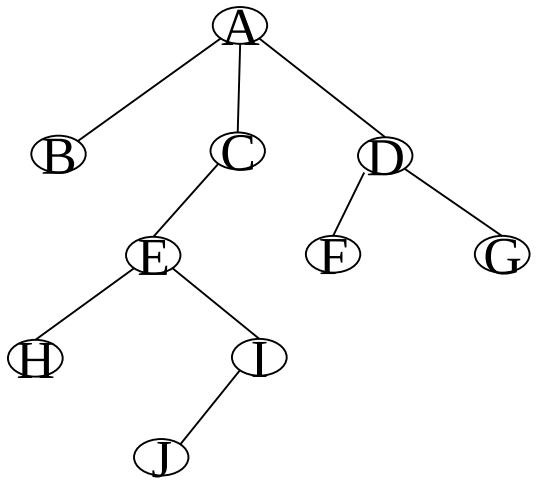
树的逻辑表示

- T = (N, R)
  - 结点集合 N = { A, B, C, D, E, F, G, H, I, J}
  - N上的关系R = { <A, B>, <A, C>, <A, D>, <C, E>,
    <D, F>, <D, G>, <E, H>, <E, I>,
    <I, J> }



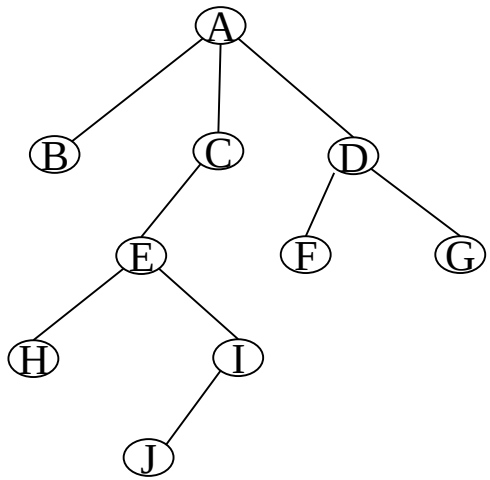
树的特点

- 在一棵树中，通常将一个结点定义为其子树的根结点的前驱结点，而子树的根结点就是它的后继结点。
- 根结点没有前驱结点，其它结点有唯一前驱
- 所有结点可以有零个或多个后继
- 树描述的是层次关系，数据元素之间存在一对多或多对一关系。



基本术语1

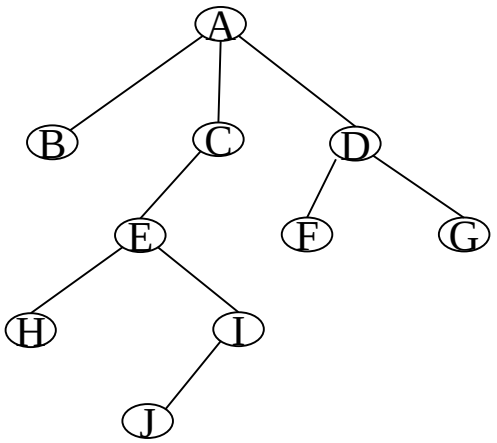
- 父结点、子结点、边
  - 若结点y是结点x的一棵子树的根，则x称作y的“父结点”（或父母）；
  - y称作x的“子结点”（或子女）；
  - 有序对<x, y>称作从x到y的“边”
  - 例如树t中，C是E的父结点，E是C的子结点，<C, E>是从C到E的边（它对应着图中的有向线段CE）
- 兄弟
  - 具有同一父结点彼此称作“兄弟”
  - 树t中B，C，D互为兄弟，F，G互为兄弟，
  - 注意，E和F并不是兄弟



树t

基本术语2

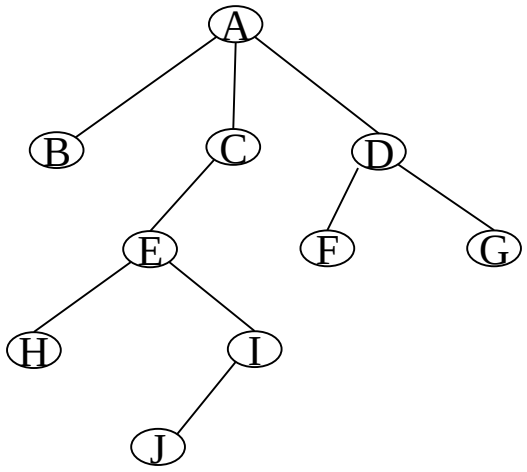
- 祖先、子孙
  - 若结点y在以结点x为根的一个子树中，且y≠x，则称x是y的“祖先”，y是x的“子孙”
  - 例如树t中，A是其它各结点的祖先；C是E，H，I，J的祖先。



- 路径、路径长度
  - 如果x是y的一个祖先，又有$x=x_0, x_1, \ldots, x_n=y$，满足$x_i$（i＝0，1，…，n-1）为$x_{i+1}$的父结点，则称$x_0$，$x_1$，…，$x_n$为从x到y的一条路径。
  - n称为这条路径的长度。路径中相邻的两个结点可以表示成一条边。
  - 例如树t中A，C，E，I，J是从A到J的一条路径，其长度为4。

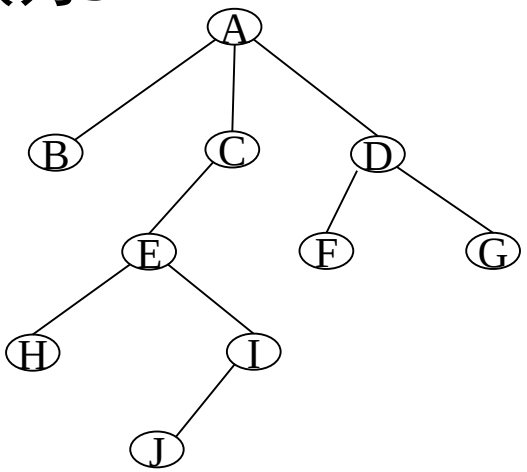
基本术语3

- 结点的层数、树的层数
  - 规定根的层数为0，其余“结点的层数”等于其父结点的层数加1。
  - 例如t中，0层的结点是A，1层的结点有B，C，D，4层的结点是J，树的层数是4。
  - 空树的层数为-1
- 树的深度或高度
  - 树中结点的最大层数称为“树的深度”或“树的高度”
  - 例如树t中，树的深度为4。



基本术语4

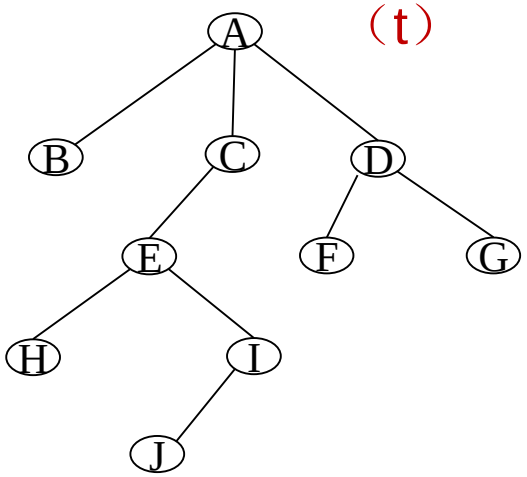
- 结点的度数、树的度数
  - 结点的子女个数叫作结点的“度数”。
  - 树中度数最大的结点的度数叫作“树的度数”
    - 例如t中A，C，E，J的度数分别为3，1，2，0；t的度数为3

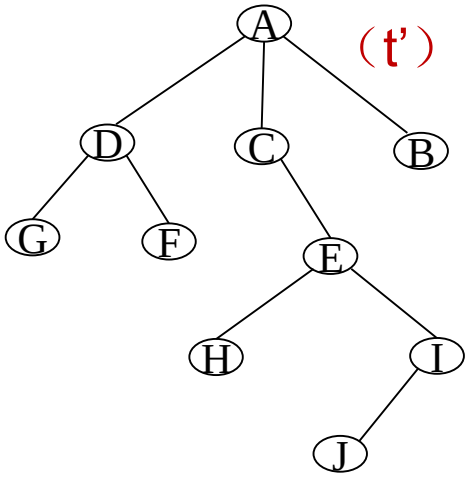


- 树叶、分支结点
  - 度数为0的结点称作“树叶”（又叫终端结点）
  - 度数大于0的结点称作“分支结点”或非终端结点
  - 例如树t中B，F，G，H，J都是树叶，其余结点都是分支结点
  - 注意，结点的度数为1时，虽然只有一个子女，也叫分支结点。这两个术语对于根结点也不例外

基本术语5

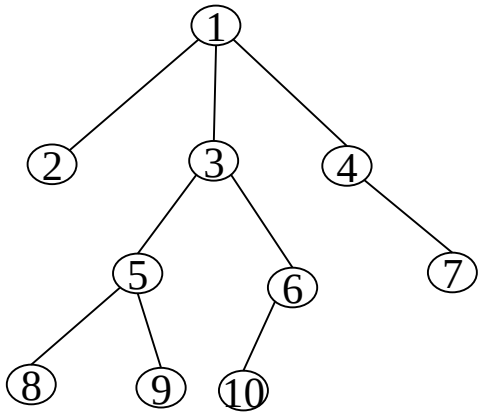
- 无序树、有序树
  - 对子树的次序不加区别的树叫作“无序树”。
  - 对子树之间的次序加以区别的树叫作“有序树”
  - 例如在右图中，按无序树的概念t和t’是同一棵树，按有序树的概念则是不同的树，本章讨论的树一般是有序树





基本术语6

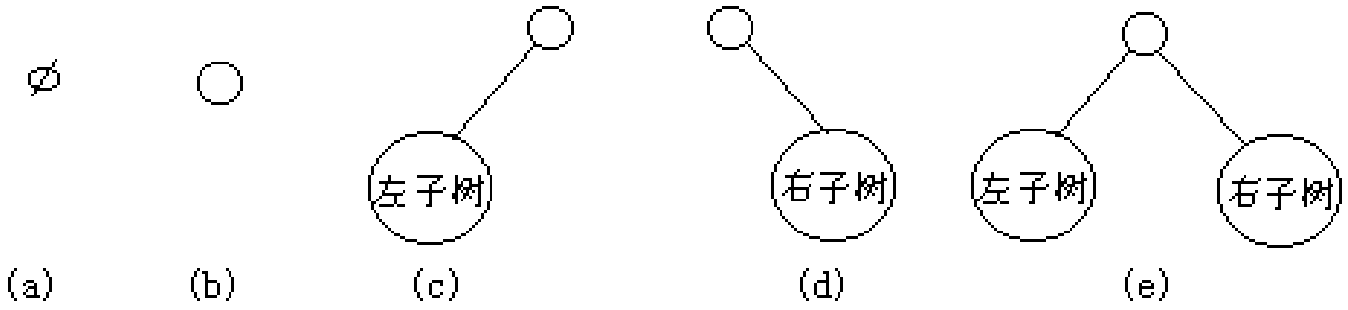
- 结点的次序
  - 在有序树中可以从左到右地规定结点的次序
  - 例如图中，结点2，3，4是从左到右排序的；可以说结点3是结点2右边的结点，是结点4左边的结点
  - 我们还可以说结点3的所有子女都在结点2及其子女的右边，而在结点4及其子女的左边
  - 按从左到右的顺序，可以把一个结点的最左边的结点简称“最左子结点”或“长子”，长子右边的结点称为“次子”
  - 注意：祖先与子孙之间不存在左右的概念。



二叉树

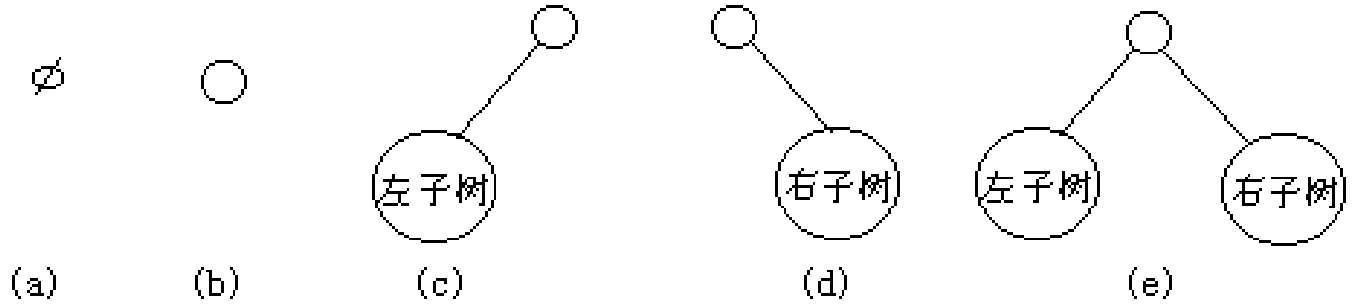
- 二叉树的定义：
  - 结点的有限集合，这个集合或者为空集，或者由一个根及两棵不相交的分别称作这个根的“左子树”和“右子树”的二叉树组成。

- 二叉树的五种基本形态



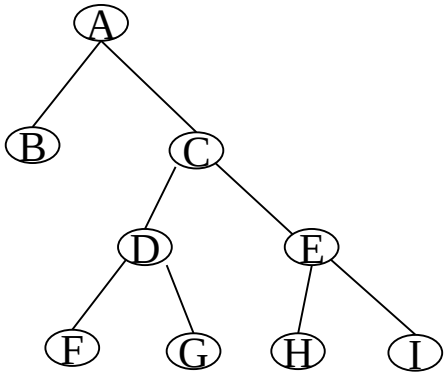
二叉树 vs 树

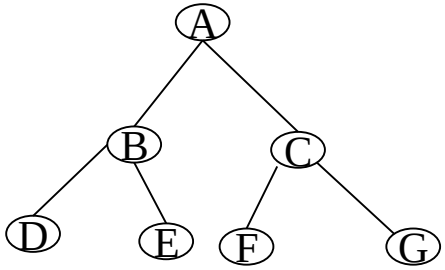
- 树和二叉树之间最主要的差别：
  - 二叉树中结点的子树要区分为左子树和右子树，即使在结点只有一棵子树的情况下也要明确指出该子树是左子树还是右子树
  - 譬如，（c）和（d）是两棵不同的二叉树，但作为树，它们是相同的



满二叉树（Full Binary Tree）

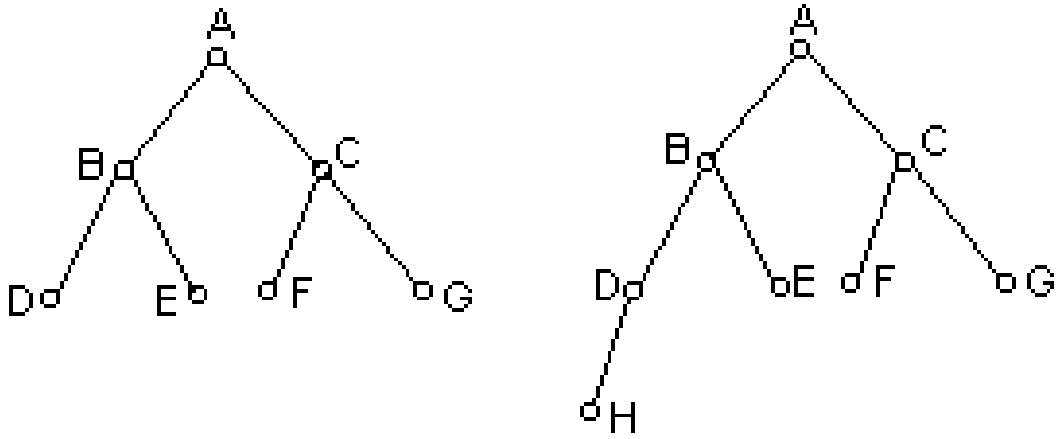
- 满二叉树：如果一棵二叉树的任何结点或者是树叶，或有两棵非空子树，则此二叉树称作“满二叉树”





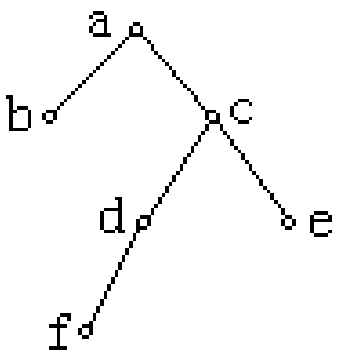
完全二叉树（Complete Binary Tree）

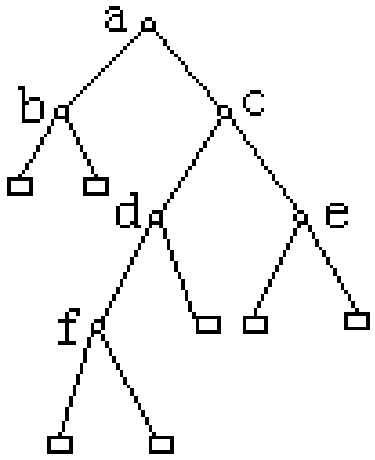
- 完全二叉树：如果一棵二叉树至多只有最下面的两层结点度数可以小于2，并且最下面一层的结点都集中在该层最左边的若干位置上



扩充二叉树（Extended Binary Tree）

- 扩充二叉树：把原二叉树的结点都变为度数为2的分支结点
- 如果原节点的度数为2，则不变
- 如果度数为1，则增加一个分支
- 度数为0（树叶）增加两个分支 ->满二叉树

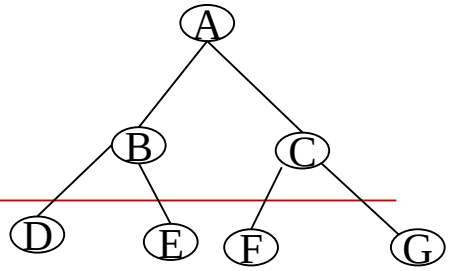




扩充二叉树（Extended Binary Tree）

- 在扩充的二叉树里，新增加的结点（树叶），称为外部结点；原有结点称为内部结点
  - “外部路径长度”E：在扩充的二叉树里从根到每个外部结点的路径长度之和
  - “内部路径长度”I：在扩充的二叉树里从根到每个内部结点的路径长度之和

二叉树的基本性质1



□ 性质1 在非空二叉树的第i层上至多有2^i个结点($i \geq 0$)

证明：用归纳法来证。

① i=0时，二叉树中只有一个根结点，显然$2^i = 2^0 = 1$是对的

② 现在假定对所有的j ($0 \leq j \leq i$)，命题成立，即第j层上至多有2^j个结点

③ 下面证明当j=i+1时，命题也成立

由归纳假设可知：第i层上至多有2^i个结点。又由于二叉树中每个结点的度至多为2，所以第i+1层上的最大结点个数是第i层上最大结点个数的2倍，即$2*2^i = 2^{i+1}$

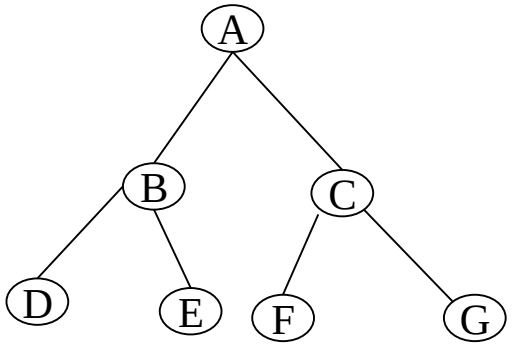
结论：故二叉树的第i层上至多有2^i个结点($i \geq 0$)

二叉树的基本性质2

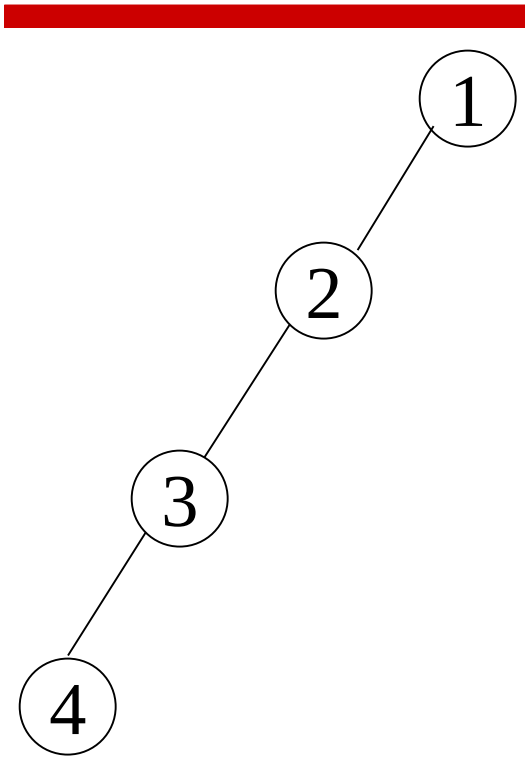
- **性质2　深度为k的二叉树中最多有　　个结点(k≥0)**

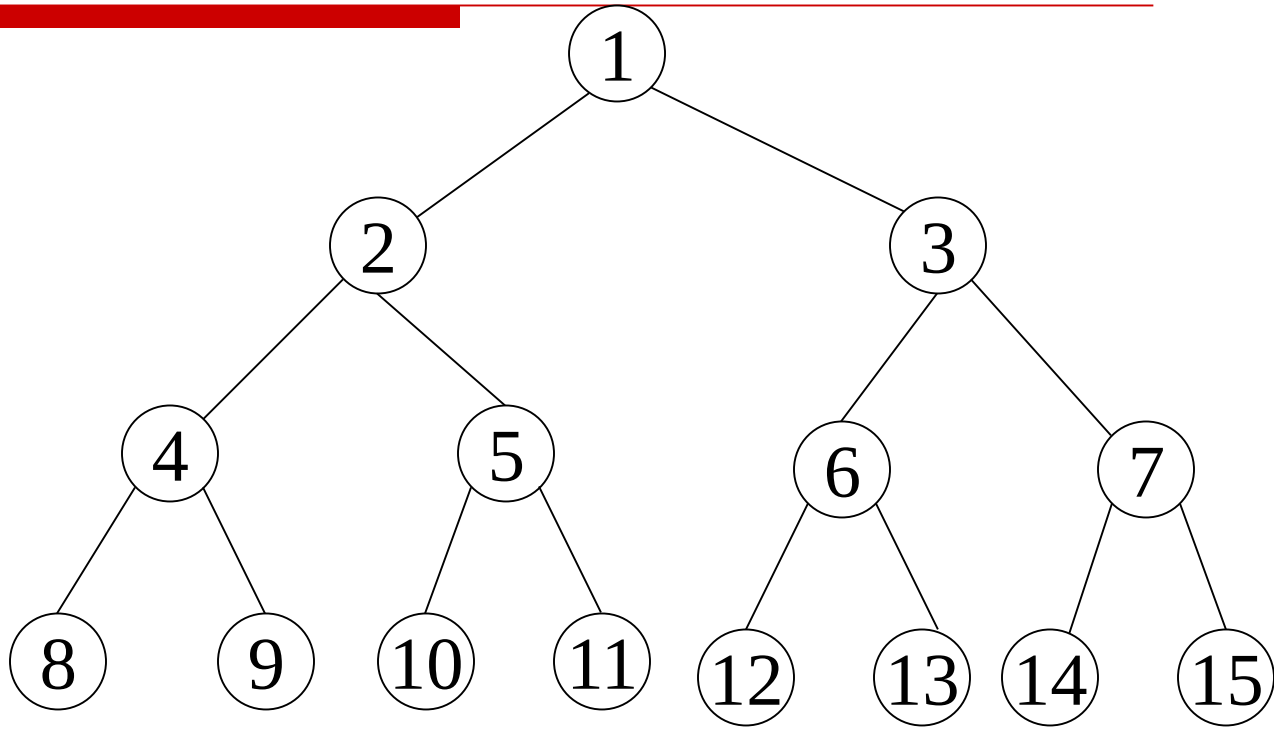
证明：假设第i层上的最大结点个数是m_i，由性质1可知，深度为k的二叉树中最大结点个数M为:

$$M = \sum_{i=0}^{k} m_i \le \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$



示例





结点最少的深度为3的二叉树：1+1+1+1

结点最多的深度为3的二叉树：$2^0+2^1+2^2+2^3=2^{(3+1)}-1$

高度为h的二叉树上只有度为0和2的结点，
那么二叉树结点数最少为多少？

二叉树的基本性质3

- **性质3　对于任何一棵非空的二叉树，如果叶结点个数为n_0，度为2的结点个数为n_2，则有**

证明：设一棵非空二叉树中有n个结点，度为1的结点个数为n_1，因为二叉树中所有结点的度均不大于2，所以

$$n = n_0 + n_1 + n_2 \qquad (1)$$

在二叉树中，除根结点外，其余每个结点都有且只有一个前驱，假设B为边（分支）的总数，则有

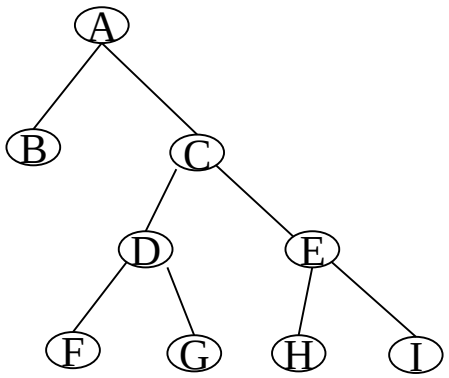
$$B = n - 1 \qquad (2)$$

又由于二叉树中的边都是由度为1和2的结点发出的，所以有

$$B = n_1 + 2n_2 \qquad (3)$$

综合(1)、(2)、(3)式可得

$$n_0 = n_2 + 1$$



二叉树的基本性质4

- **性质4. 具有n个结点的完全二叉树的深度k为**

证明：

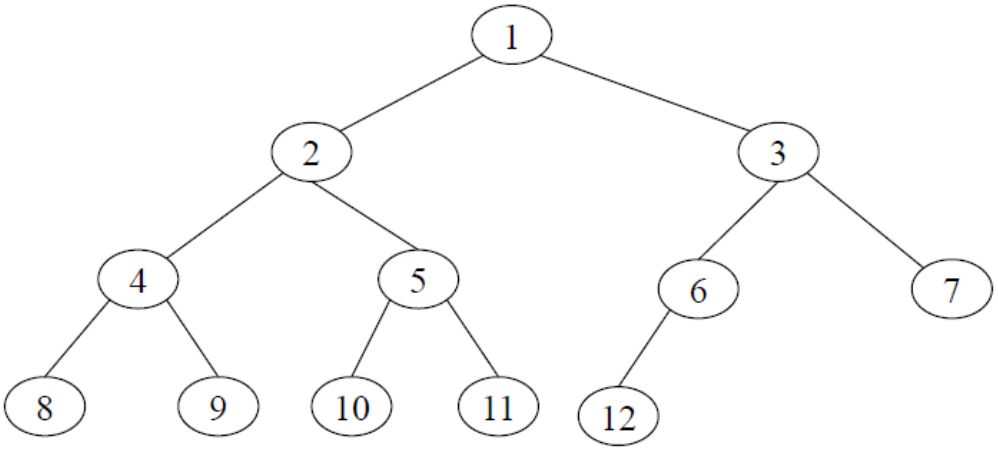
$$n = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} + m_k$$
$$= 2^k - 1 + m_k$$

$$2^k - 1 < n \leq 2^{k+1} - 1 \quad \text{（性质2）}$$
$$2^k \leq n < 2^{k+1}$$
$$k \leq \log_2 n < k+1$$
$$\therefore \quad k = \lfloor \log_2 n \rfloor$$

示例

完全二叉树层次序列反映出它的结构

1 2 3 4 5 6 7 8 9 10 11 12



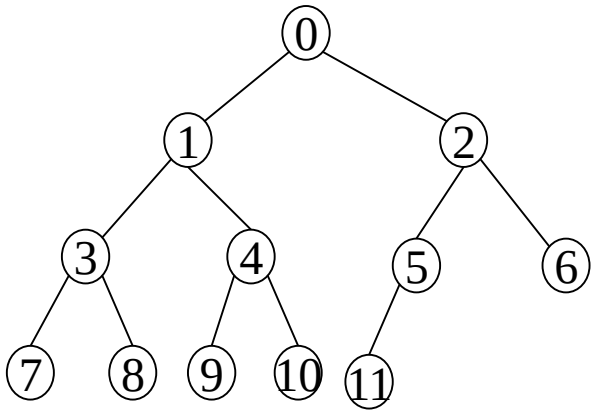
二叉树的基本性质5

- **性质5** **对于具有n个结点的完全二叉树，如果按照从上到下和从左到右的顺序对树中的所有结点从0开始进行编号，则对于任意的序号为i的结点，有：**

① 如果i＞0，则其父结点的序号为$\lfloor (i-1)/2 \rfloor$；
如果i=0，则其是根结点，它没有父结点。

② 如果2i+1≤n-1，则其左子女结点的序号为2i+1；
否则，其没有左子女结点。

③ 如果2i+2≤n-1，则其右子女结点的序号为2i+2；
否则，则其没有右子女结点。



二叉树的基本性质5---证明

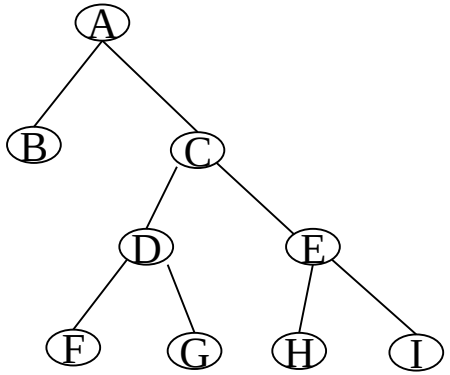
对于（2）和（3）当i=1时,若$2i+1 = 3 \le n-1$，左子女结点的序号为3。
$2i+2 = 4 \le n-1$，右子女结点的序号为4。

1. 假设对于序号为j的结点，命题成立。
2. 对于i=j+1，
   - 其左子女结点的序号等于j的右子女结点的序号加1，
     即 $2j + 2 + 1 = 2(j+1) + 1$
   - 其右子女结点的序号等于：2(j+1)+2。
3. 根据（2）和（3），可知i的父结点的序号为$\lfloor (i-1)/2 \rfloor$.

结论：完全二叉树的层次序列，反映了它的结构。

二叉树的基本性质6

性质6　在非空满二叉树中，叶节点的个数比分支节点的个数多



证明：

非空满二叉树的树叶数等于其分支结点数加1

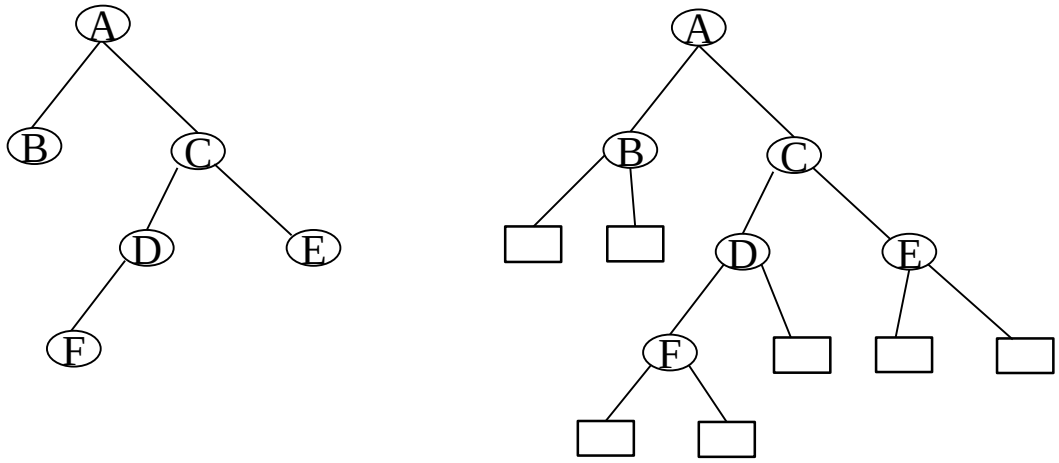
（性质3：对于任何一棵非空的二叉树，如果叶结点个数为n_0，度为2的结点个数为n_2，则有$n_0 = n_2 + 1$）。

二叉树的基本性质7

性质7　在扩充的二叉树里，新增加的外部结点的个数比原来的内部结点个数多

证明

- 对于任何一棵非空的二叉树，如果叶结点个数为n_0，度为2的结点个数为n_2，度为1的结点个数为n_1，原来的内部结点个数为$n= n_0 + n_1 + n_2$,，因为$n_0=n_2 + 1$，$n= n_1 + 2n_0-1$
- 新增加的外部结点为：　$n_1 + 2n_0$

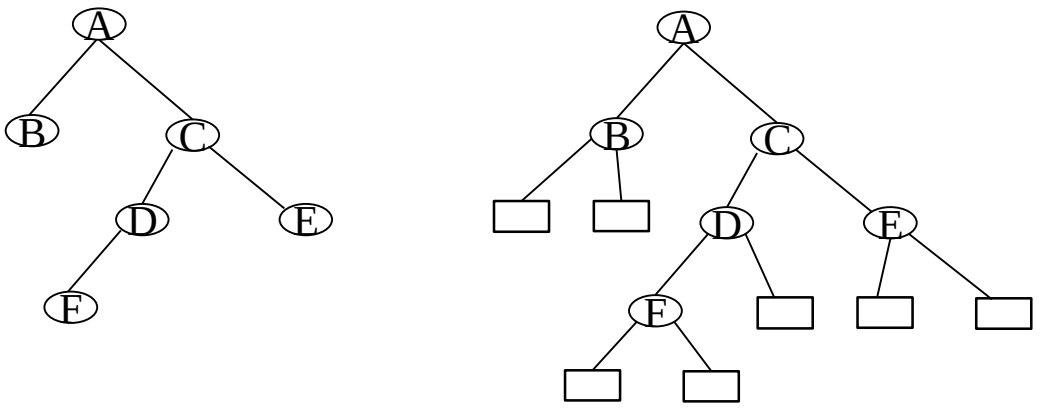


二叉树的基本性质8

- “外部路径长度”E:在扩充的二叉树里从根到每个外部结点的路径长度之和
- “内部路径长度”I:在扩充的二叉树里从根到每个内部结点的路径长度之和
- 性质8: 对任意扩充二叉树，E和I之间满足以下关系：E = I + 2n，其中n是内部结点个数。

E = 2 + 2 + 4 + 4 + 3 + 3 + 3 = 21

I = 0 + 1 + 1 + 2 + 2 + 3 = 9



证明

1. 归纳基础：当n=1时，I=0, E=2，此等式成立。
2. 归纳假设：设有n个内部结点的扩充二叉树，下式成立。

$$E_n = I_n + 2n \qquad (1)$$

3. 归纳推理：对于 n+1 个内部结点的扩充二叉树，去掉一个原来为树叶、路径长度为K的内部结点，内部路径长度变为：

$$I_n = I_{n+1} - K \qquad (2)$$

外部路径长度变为：

$$E_n = E_{n+1} - 2(K+1) + K = E_{n+1} - K - 2$$

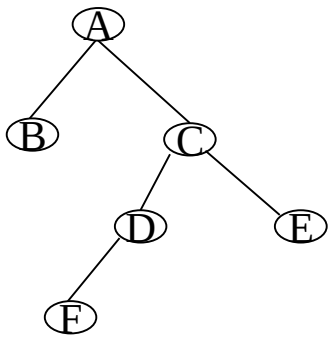
即：

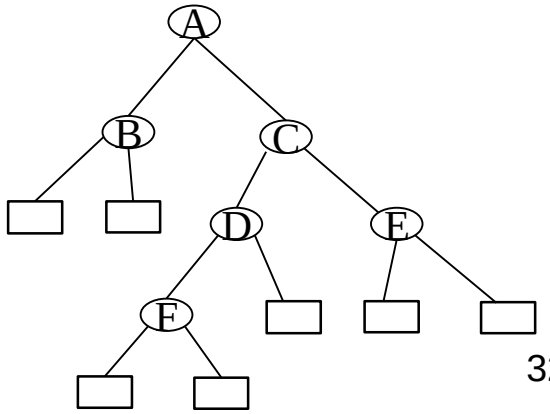
$$E_{n+1} = E_n + K + 2$$

$$E_{n+1} = \underline{(I_n + 2n)} + K + 2 = \underline{(I_{n+1} - K)} + 2n + K + 2$$

代入（1）　　代入（2）

$$= I_{n+1} + 2(n+1)$$





如何存储一个二叉树？

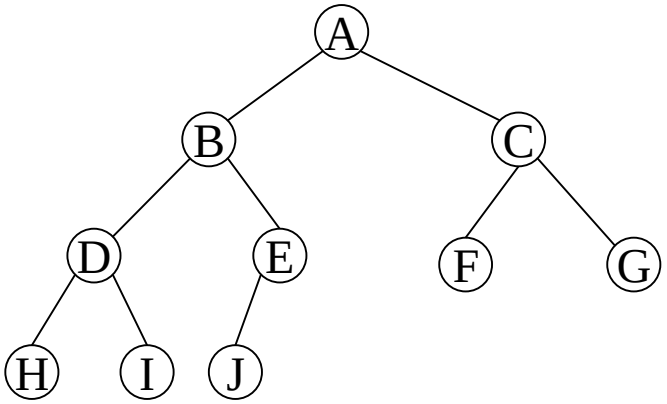
- 二叉树基础
  - 树与二叉树的基本概念
  - 二叉树的存储结构
  - 二叉树的周游算法
  - 建立一个二叉树
- 二叉树的应用
  - 哈夫曼树
  - 二叉检索树/排序树
- 树与树林

二叉树的实现

- 根据应用的不同以及二叉树本身的不同特点，二叉树可以采用不同的存储结构
  - 顺序表示
  - 链式表示

完全二叉树的顺序表示

- 用一组连续的存储单元来存放二叉树中的结点
  - 完全二叉树中结点的序号可以反映出结点之间的逻辑关系，按层次顺序将一棵有n个结点的完全二叉树的所有结点从0到n-1编号，就得到结点的一个线性序列。
  - 例：完全二叉树的顺序表示（根据二叉树性质5）



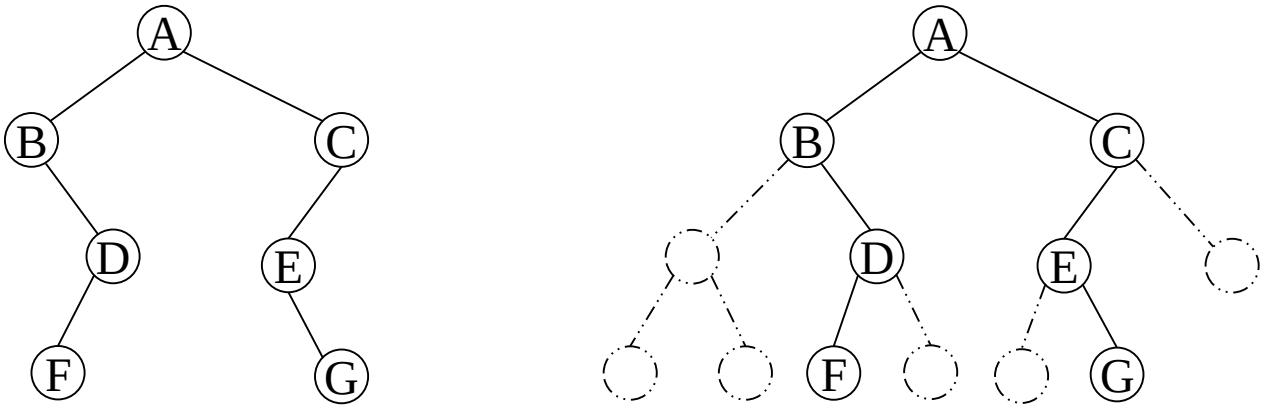
数组下标	A	B	C	D	E	F	G	H	I	J
	0	1	2	3	4	5	6	7	8	9

完全二叉树的顺序表示

- 完全二叉树结点的顺序存储
  - 所有结点按层次顺序依次存储在连续的存储单元中，
  - 根据一个结点的存储地址就可算出它的左右子女，父母的存储地址，如同存储了相应的指针一样。
- 顺序表示是存储完全二叉树的最简、最节省空间的存储方式
  - 完全二叉树的顺序存储，在存储结构上是线性的，但在逻辑结构上它仍然是二叉树型结构

一般二叉树的顺序表示

- 增加空结点来构造一棵完全二叉树,再以二叉树的方式存储
- 接近于完全二叉树的形态，需要增加的空结点数目不多，则也可采用顺序表示；



数组下标	0	1	2	3	4	5	6	7	8	9	10	11	12
	A	B	C	^	D	E	^	^	^	F	^	^	G

顺序表示的二叉树定义

采用顺序存储表示的二叉树定义如下:

```
#define  MAXNODE  100      /* 定义二叉树中结点的最大个数 */
struct SeqBTree                /* 顺序树类型定义 */
{
   DataType  nodelist[MAXNODE];
   int  n;          /* 改造成完全二叉树后，结点的个数 */
}
typedef  struct SeqBTree  *PSeqBTree;
                               /* 顺序树类型的指针类型 */
PSeqBTree  pabtree;
                              /* pabtree是指向顺序树的一个指针变量 */
```

二叉树的基本运算

1. 创建一棵空二叉树；
2. 判断某棵二叉树是否为空；
3. 求二叉树中的根结点；

   若为空二叉树，则返回一特殊值
4. 求二叉树中某个指定结点的父结点；

   当指定结点为根时，返回一特殊值
5. 求二叉树中某个指定结点的左子结点；

   当指定结点没有左子女时，返回一特殊值
6. 求二叉树中某个指定结点的右子结点；

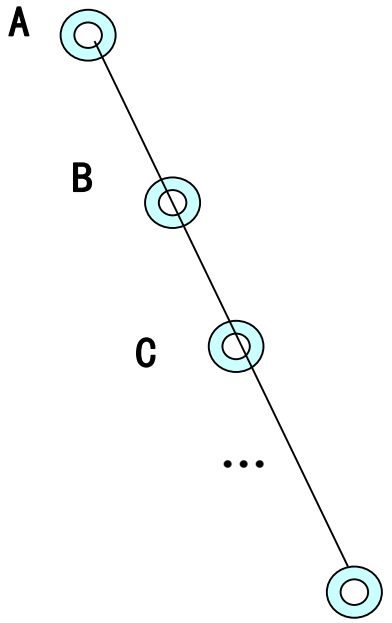
   当指定结点没有右子女时，返回一特殊值
7. 二叉树的周游，即按某种方式访问二叉树中的所有结点，并使每个结点恰好被访问一次。

二叉树上的基本运算

- root_seq(t)
  - 即结点为t->nodelist[0]
- parent_seq(t,p)
  - 即结点为t->nodelist[(p-1)/2]
- leftchild_seq(t,p)
  - 即结点为t->nodelist[2p+1]
- rightchild_seq(t,p)
  - 即结点为t->nodelist[2(p+1)]

一般二叉树的顺序存储的问题

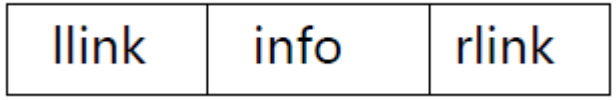
- 如果需要增加很多空结点才能将一棵一般的二叉树改造成完全二叉树，那么采用顺序表示就会造成空间的大量浪费，这时就不适合用顺序表示。
- 对于一种特殊情况，如果二叉树为右单支树[深度为k, 有k+1个结点]，则需要一个长度为2^{k+1}-1的一维数组（性质2），造成（2^{k+1}-1）-（k+1)个结点浪费。
  - 如果k=5, 则有57个结点空间浪费。



二叉树的链式表示

- 二叉树的各结点随机地存储在内存空间中，结点之间的逻辑关系用指针来链接。

- 二叉链表



  - 指针left 和right，分别指向结点的左孩子和右孩子

- 三叉链表
  - 指针left 和right，分别指向结点的左孩子和右孩子
  - 增加一个父指针

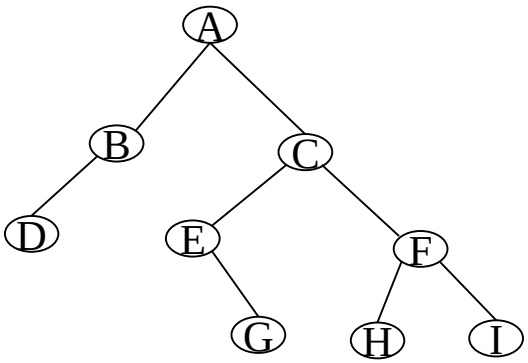
二叉树的链式表示

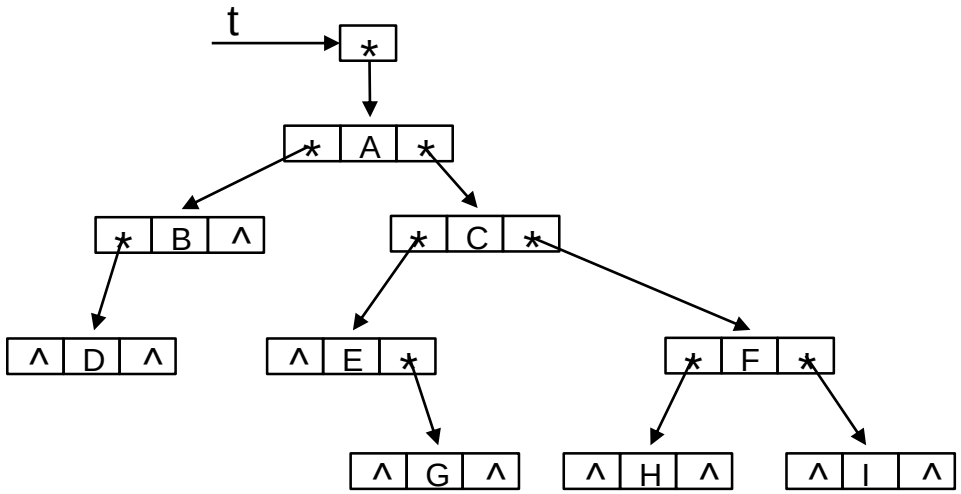
```
① struct BinTreeNode          /* 二叉树中结点 */
② typedef struct  BinTreeNode  *PBinTreeNode;

③ struct BinTreeNode
④ {    DataType  info;          /* 数据域 */
⑤        PBinTreeNode  llink;  /* 指向左子女 */
⑥        PBinTreeNode  rlink;  /* 指向右子女 */
⑦ };
⑧ typedef struct BinTreeNode *BinTree;      /*二叉树定义*/
⑨ typedef BinTree *PBinTRee;          /* 二叉树类型的指针类型 */
```

二叉树类型的指针变量

- PBinTree t; /* t是指向二叉树类型的指针变量 */
  - t实际上是一个指向二叉树根结点的指针的指针
  - 既指明根的位置又标识一棵二叉树





二叉树上的基本运算

BinTree t

- root_btree(t) ⟺ t
- leftChild_btree(p) ⟺ p->llink
- rightChild_btree(p) ⟺ p->rlink

- 如何访问一棵二叉树中所存储的所有数据？

二叉树上的周游算法

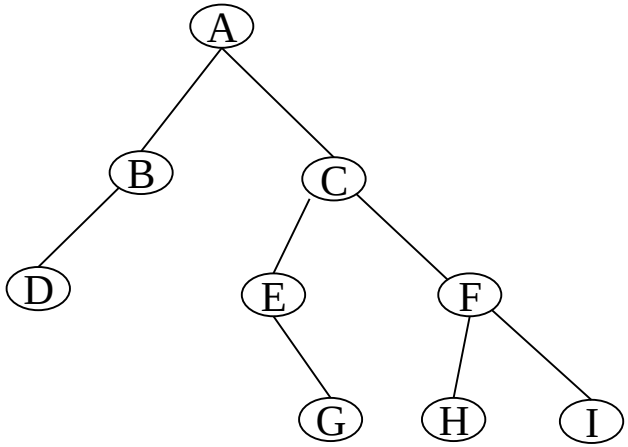
- 二叉树基础
  - 树与二叉树的基本概念
  - 二叉树的存储结构
  - 二叉树的周游算法
  - 建立一个二叉树
- 二叉树的应用
  - 哈夫曼树
  - 二叉检索树/排序树
- 树与树林

二叉树的周游概念

- 二叉树的周游是指按某种方式访问二叉树中的所有结点，使每个结点被访问一次且只被访问一次
  - 周游二叉树的过程实际上就是把二叉树的结点放入一个线性序列的过程，或者说是把二叉树进行线性化
- 周游的方式：
  - 深度优先周游二叉树（depth-first traversal）
    - **先根序；中根序；后根序**
  - 广度优先周游二叉树（breadth-first traversal）

先根周游

- 若以符号D、L、R分别表示访问根结点、周游根结点的左子树、周游根结点的右子树。



- 先根次序：DLR
  ① 访问根；
  ② 按先根次序周游左子树；
  ③ 按先根次序周游右子树。

- 将按先根次序对一棵二叉树周游得到的结果称为这棵二叉树的先根序列。

- 先根序列为：A B D C E G F H I

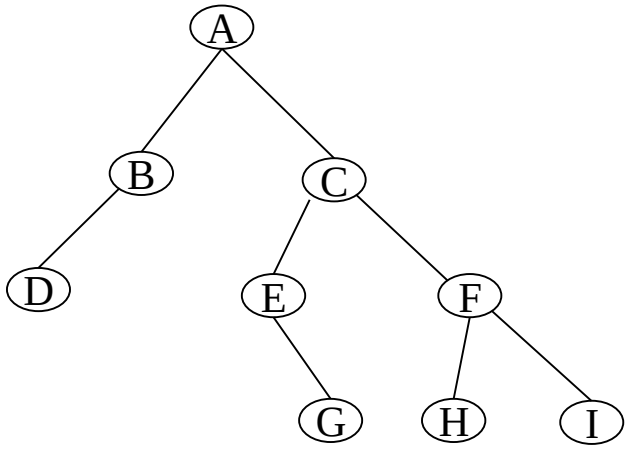
二叉树先根次序周游的递归算法

- 函数visit(BNode p)的作用是访问指针p所指向的结点

```
void preOrder(BNode p)
{
      if (p==NULL) return;
      visit(p);
    preOrder(leftchild(p));
    preOrder(rightchild(p));
}
```

后根周游

- 后根次序：LRD
  - ① 按后根次序周游左子树；
  - ② 按后根次序周游右子树；
  - ③ 访问根。



- 通常将按后根次序对一棵二叉树周游得到的结果称为这棵二叉树的后根序列。

- 后根序列是：D B G E H I F C A

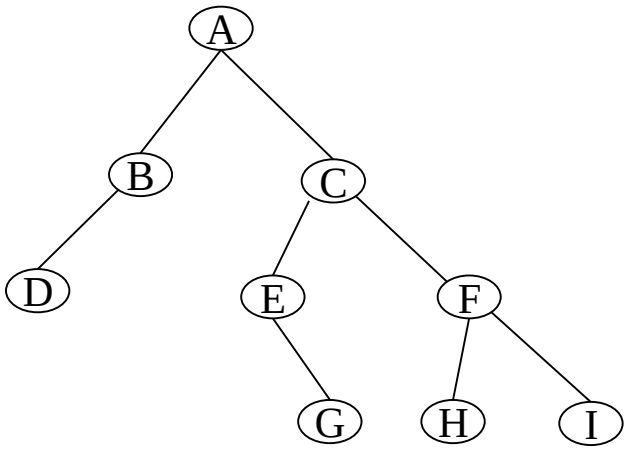
二叉树后根次序周游的递归算法

- 函数visit(BNode p)的作用是访问指针p所指向的结点

```
void postOrder(BNode p)
{
    if (p==NULL) return;
    postOrder(leftchild(p));
    postOrder(rightchild(p));
    visit(p);
}
```

中根周游

- 中根次序(对称序法)：LDR
  - ① 按对称序次序周游左子树；
  - ② 访问根；
  - ③ 按对称序次序周游右子树。



- 通常将按中根次序对一棵二叉树周游得到的结果称为这棵二叉树的中根序列。

- 中根(对称序)序列是： D B A E G C H F I

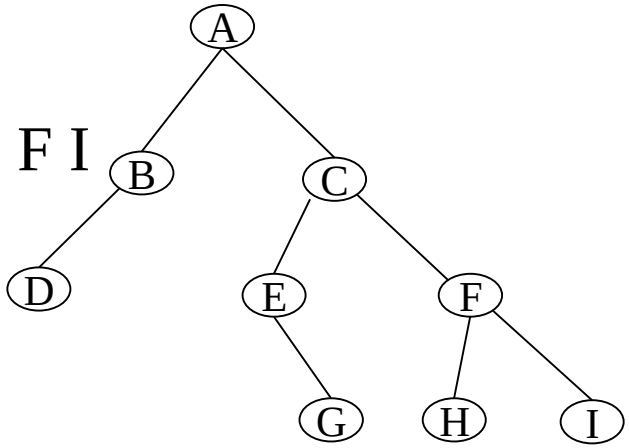
二叉树中根周游的递归算法

- 函数visit(BNode p)的作用是访问指针p所指向的结点

```
void inOrder(BNode p)
{
      if (p==NULL) return;
      inOrder(leftchild(p));
      visit(p);
    inOrder(rightchild(p));
}
```

二叉树结点的前驱与后继

- 先根序列是：A B D C E G F H I；
- 中根(对称序)序列是： D B A E G C H F I
- 后根序列是： D B G E H I F C A



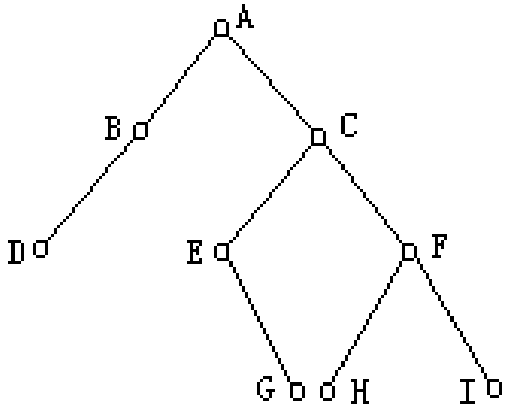
- 可定义在某次序下，某结点的前驱与后继
- 如：二叉树中的结点C，
- 它的先根前驱是D，先根后继是E，
- 对称序前驱是G，对称序后继是H

深度优先遍历算法的空间代价分析

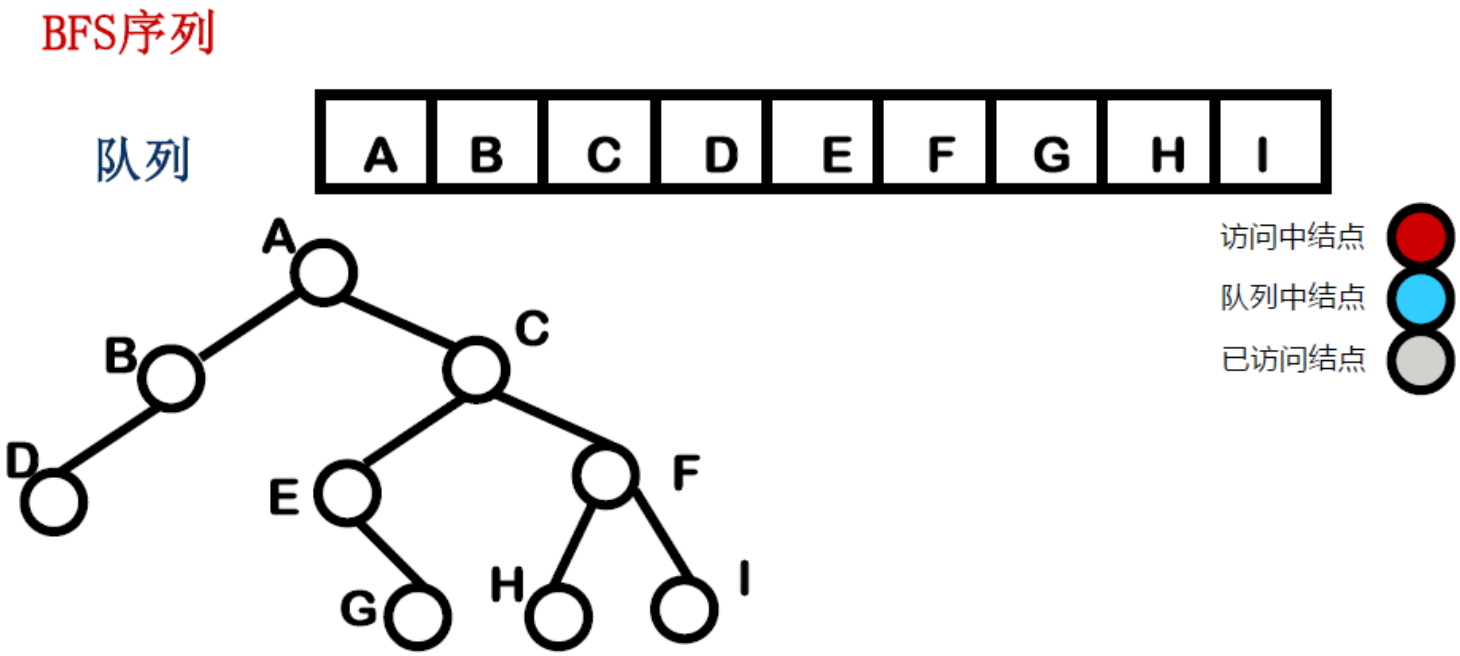
- 栈的深度与树的高度有关
  - 最好O(logn)
  - 最坏O(n)

宽度优先遍历二叉树

- 从二叉树的第0 层（根结点）开始，自上至下逐层遍历；在同一层中，按照从左到右的顺序对结点逐一访问。
- 例如：A B C D E F G H I



宽度优先遍历二叉树

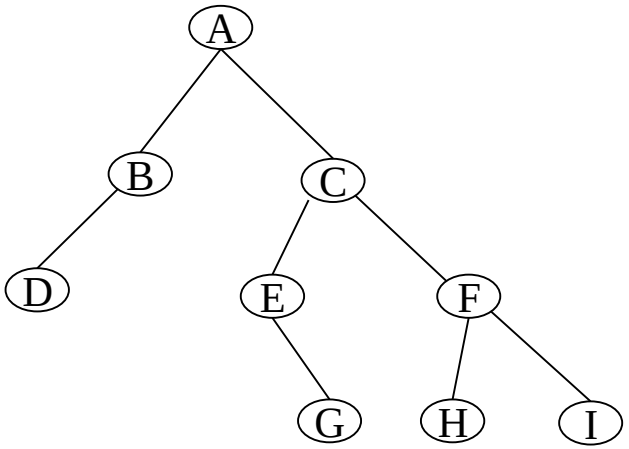


遍历二叉树的非递归算法

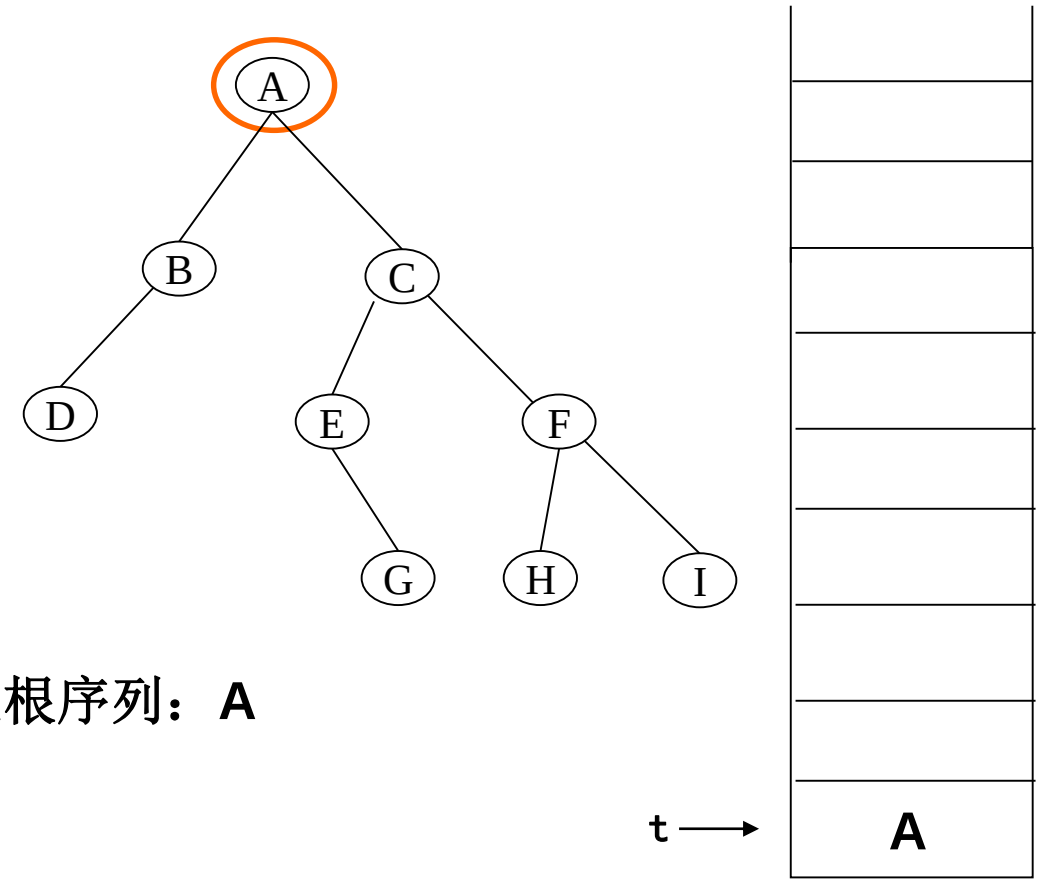
- 递归算法非常简洁——推荐使用
  - 当前的编译系统优化效率很不错了
- 特殊情况用栈模拟递归
  - 理解编译栈的工作原理
  - 理解深度优先遍历的回溯特点
  - 有些应用环境资源限制不适合递归

先根周游的非递归算法

- 基本思想：
  1. 从根结点开始，令变量p指向根结点；
  2. 若p不为空，
     - 访问当前结点p，并把p压入栈中；
     - 令p为当前结点的左子女结点。如此重复进行，直到p为空；
  3. 从栈中弹出栈顶元素赋给p，并令p为它当前指向结点的右子女结点，若当前指向结点的右子女结点为空，重复弹出，直到右子女结点不为空。
  4. 重复上述过程，当p为空时并且栈也为空时，周游结束。



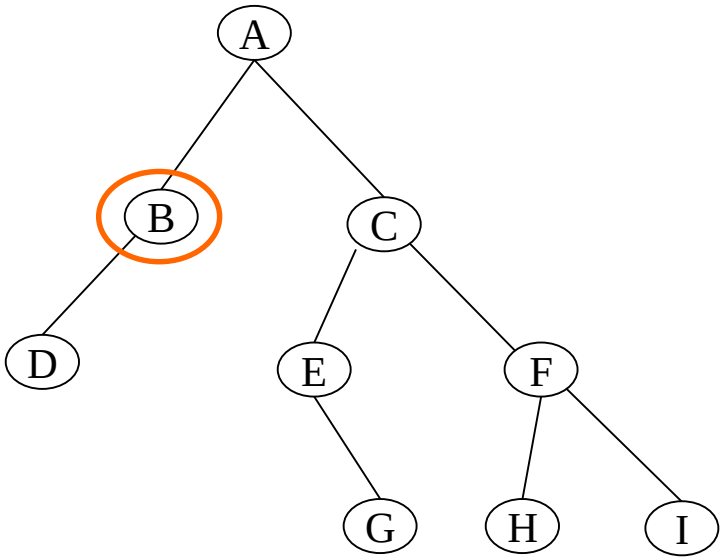
先根周游过程示意图



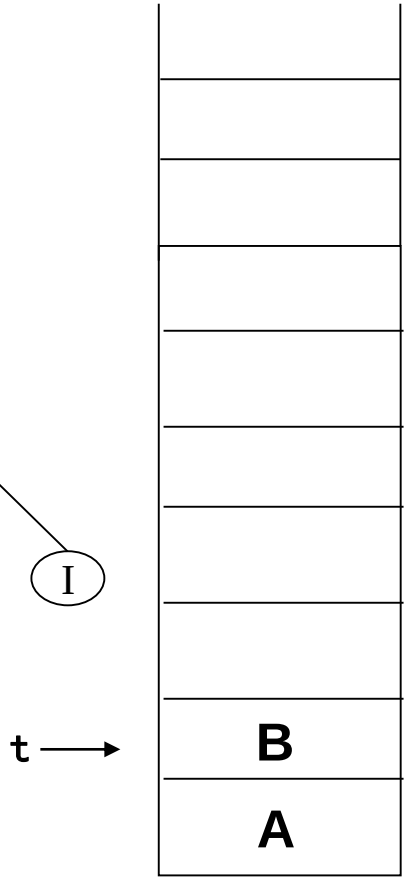
先根序列：A

```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

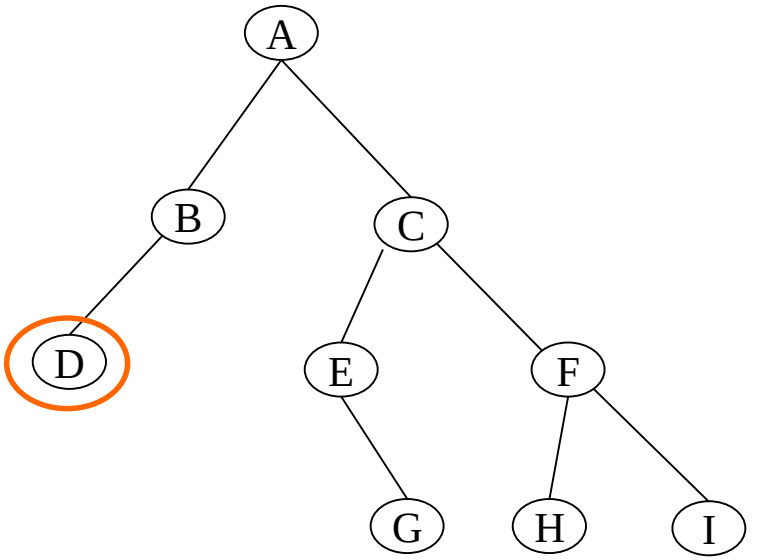


先根序列：**A B**

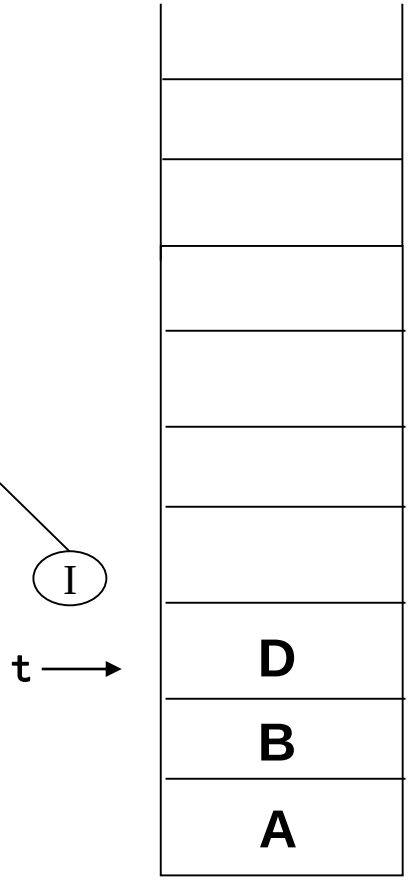



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

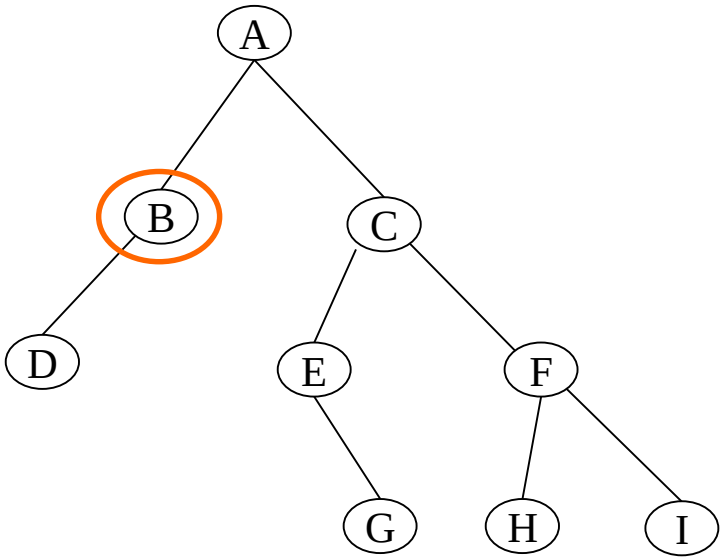


先根序列：**A B D**

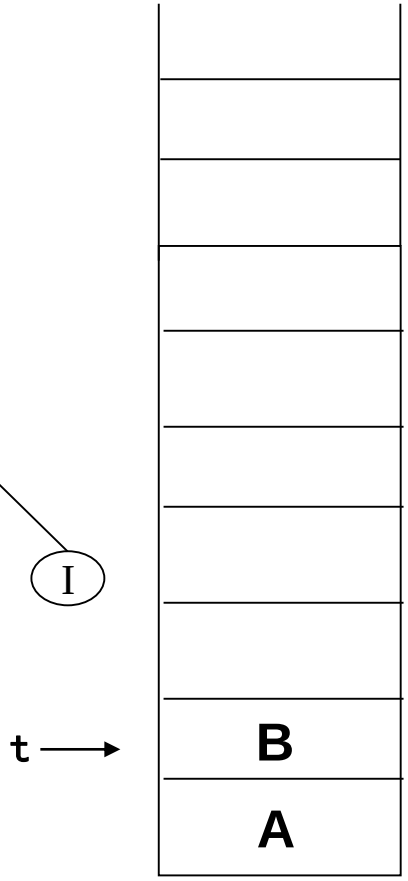



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

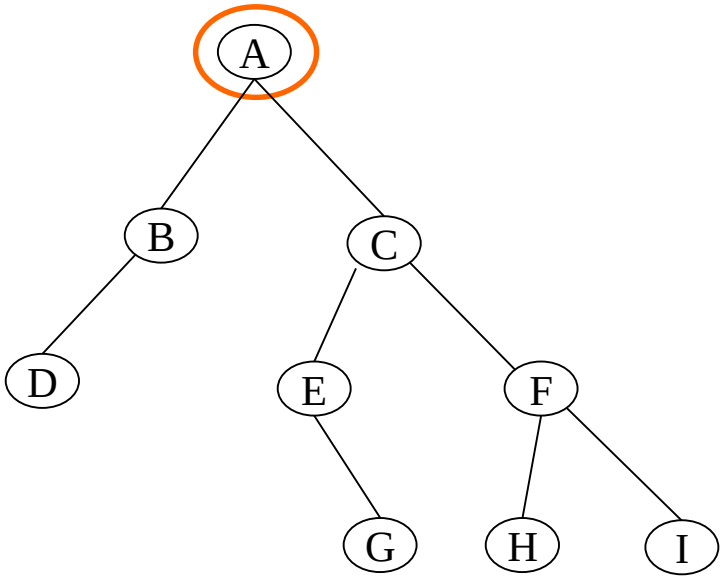


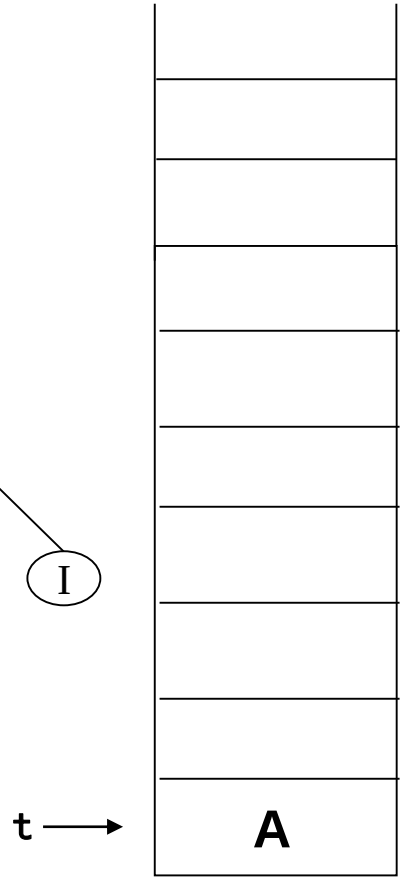
先根序列：**A B D**




```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

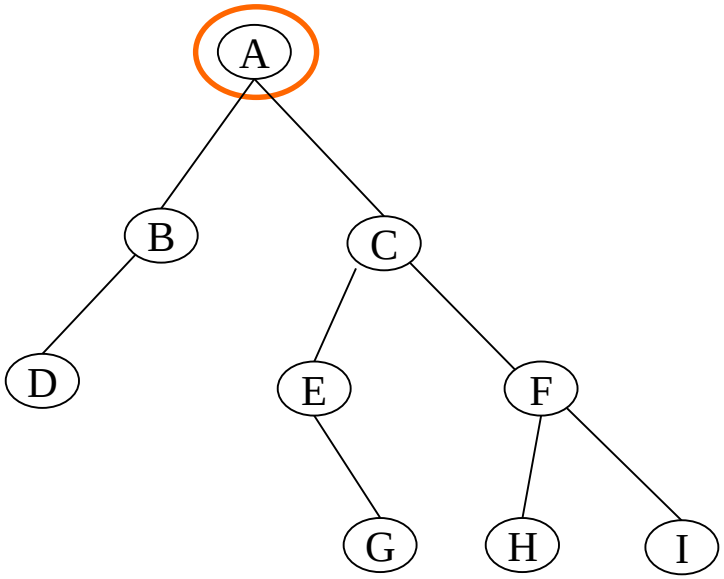




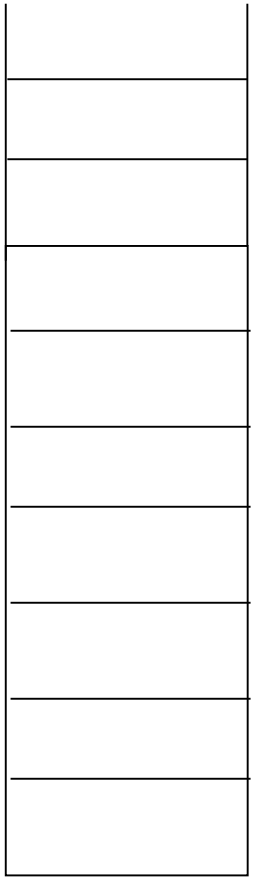
先根序列：**A B D**

```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图



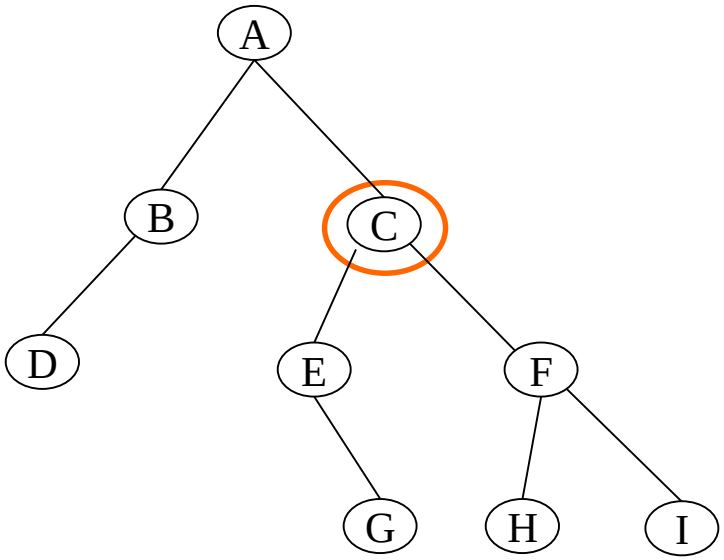
先根序列：**A B D**



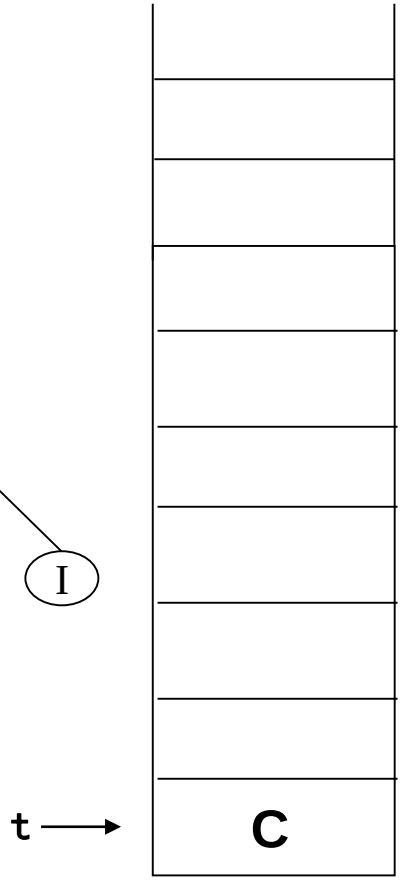
t ⟶

```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

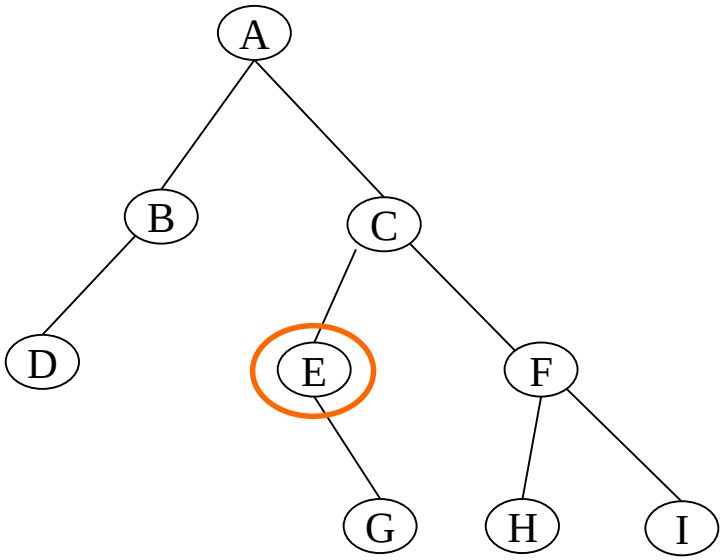


先根序列：**A B D C**

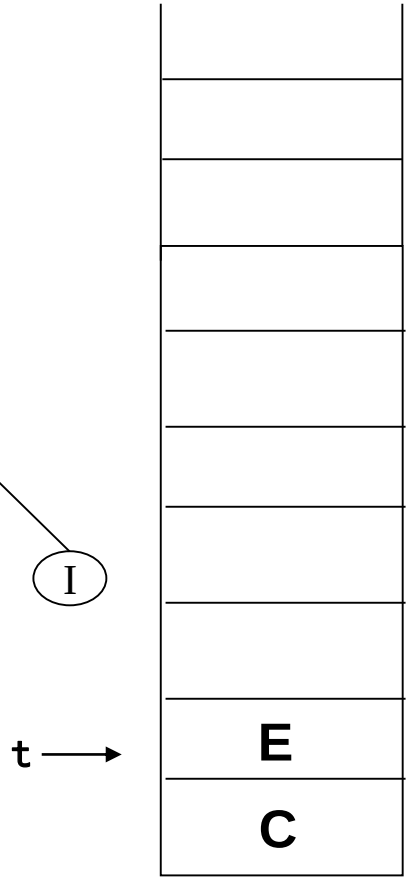



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

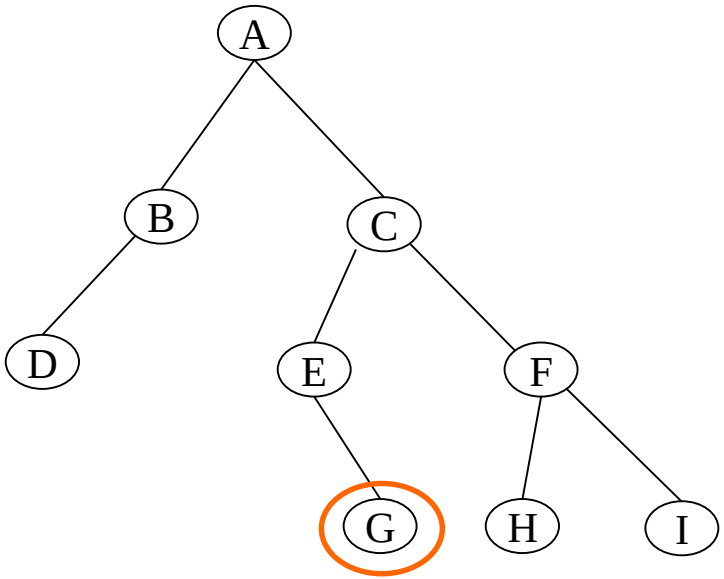


先根序列：**A B D C E**

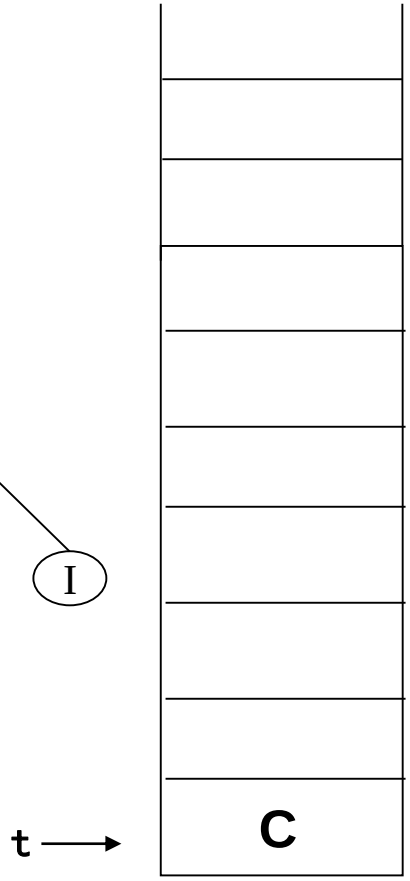



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

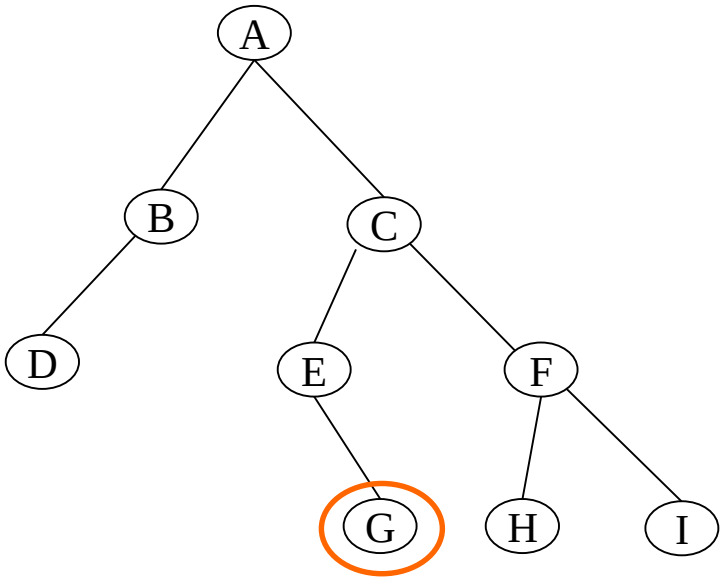


先根序列：**A B D C E**

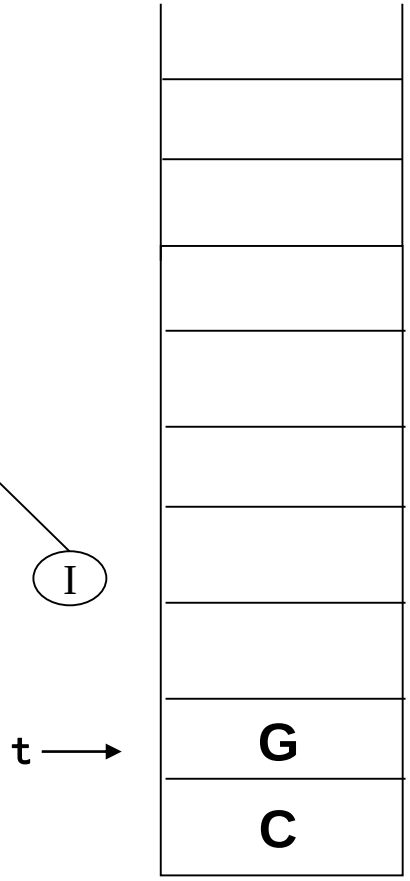



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

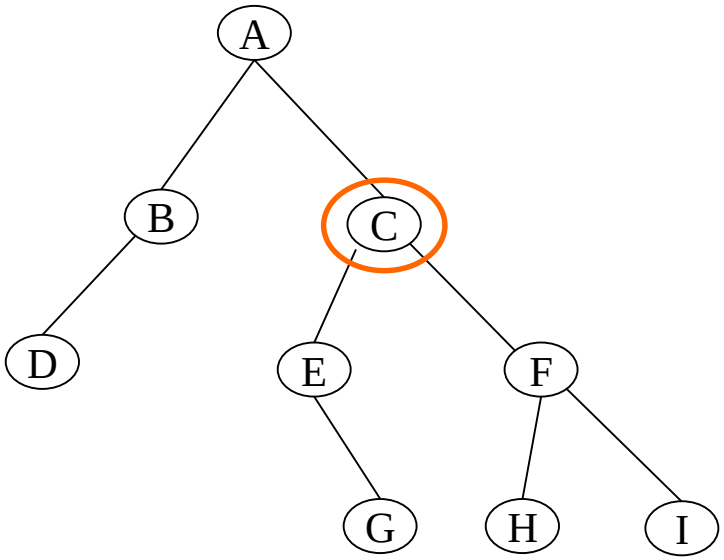


先根序列：**A B D C E G**

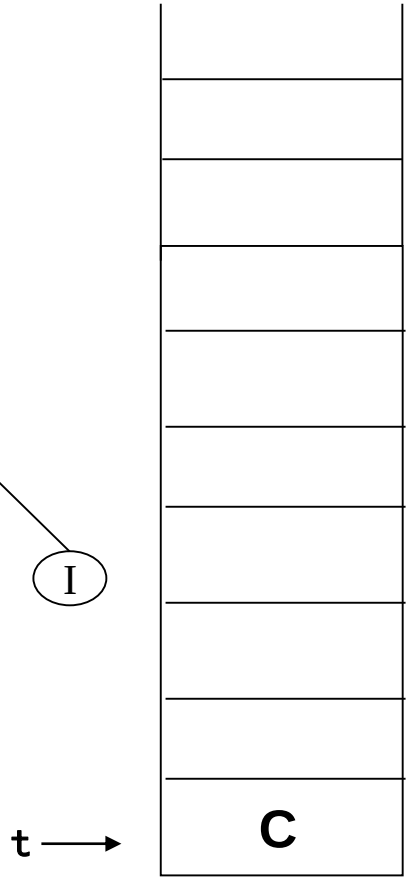



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

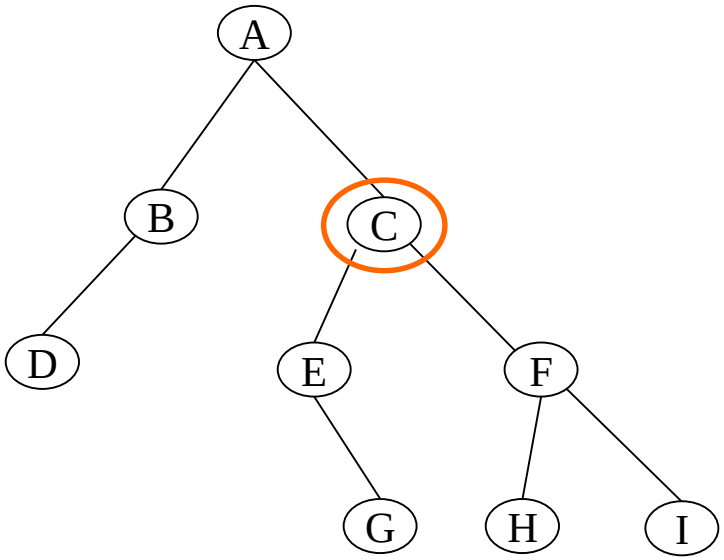


先根序列：**A B D C E G**

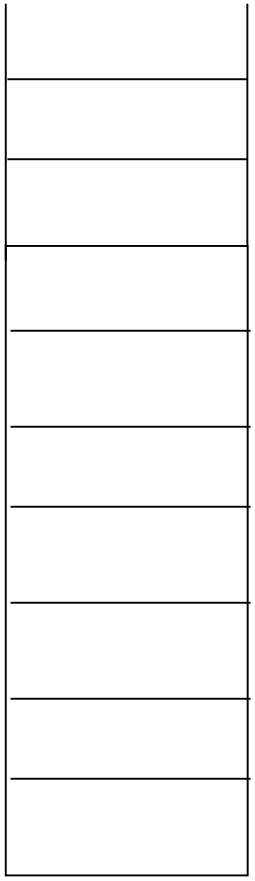



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图



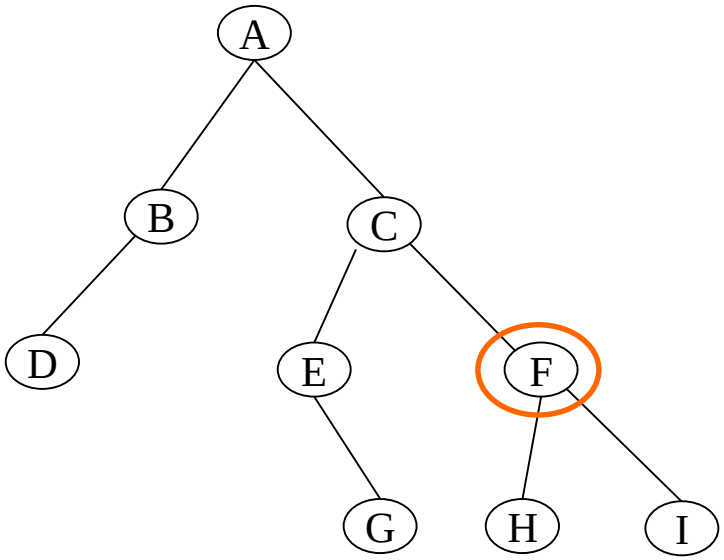
先根序列：**A B D C E G**



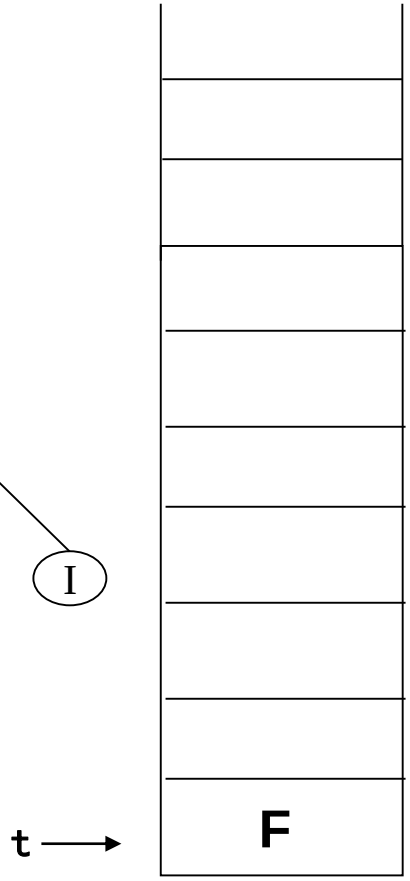
t ⟶

```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

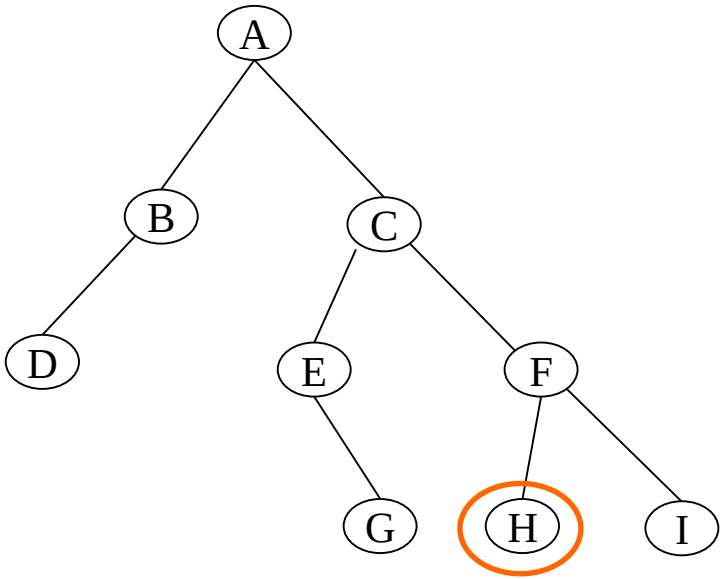


先根序列：**A B D C E G F**

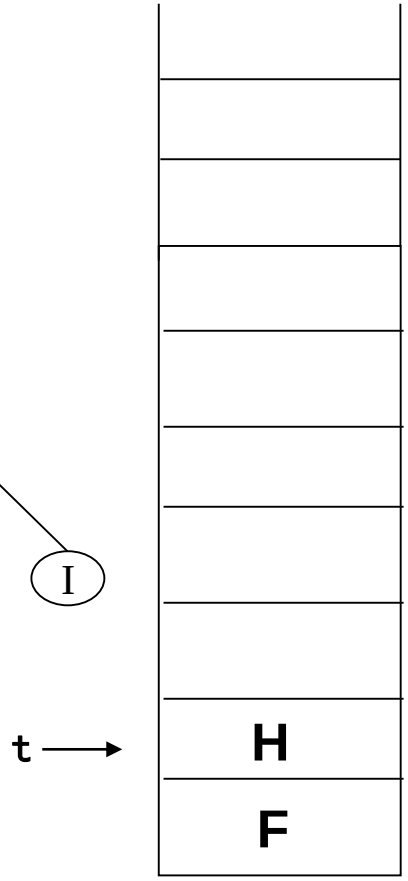



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

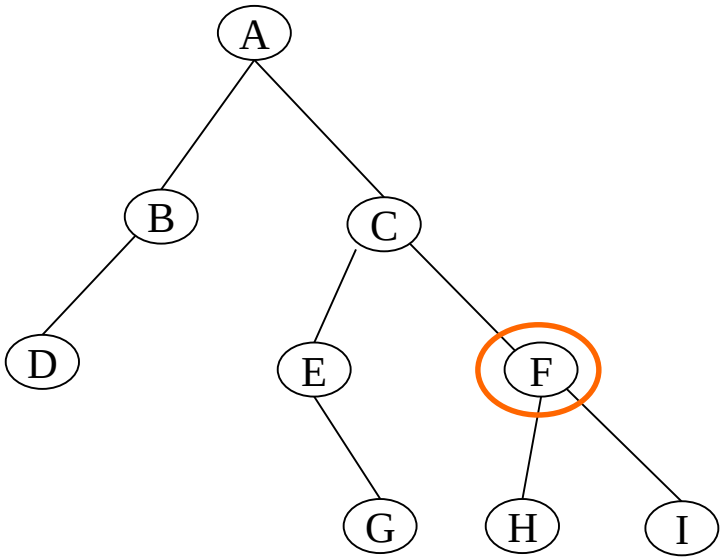


先根序列：**A B D C E G F H**

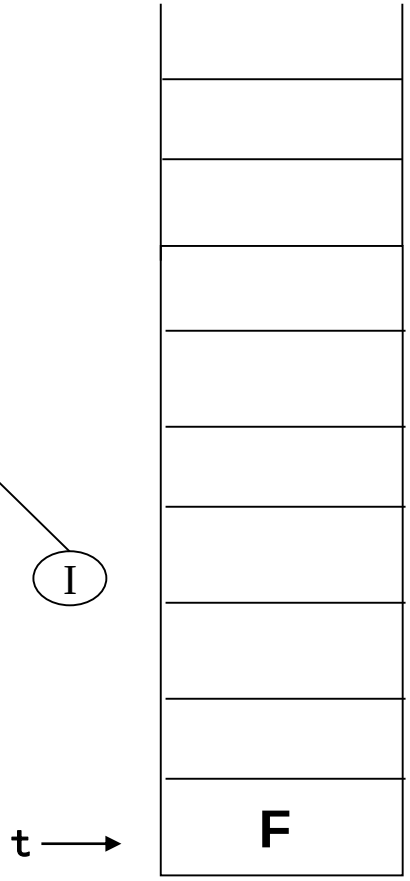



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

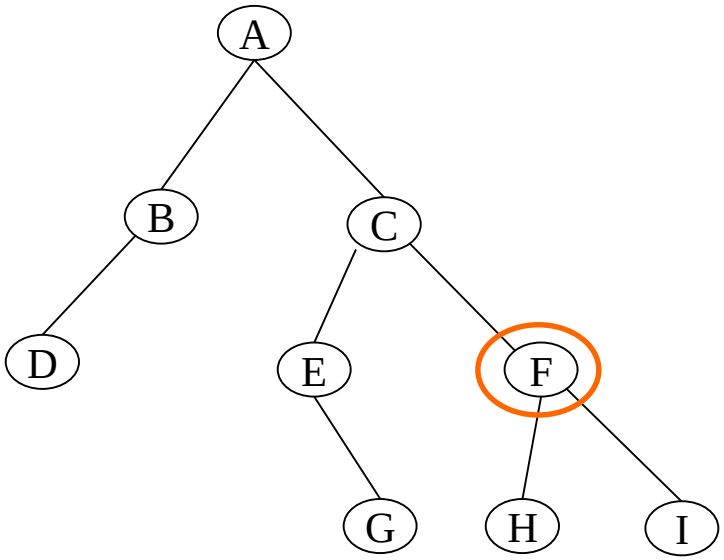


先根序列：**A B D C E G F H**

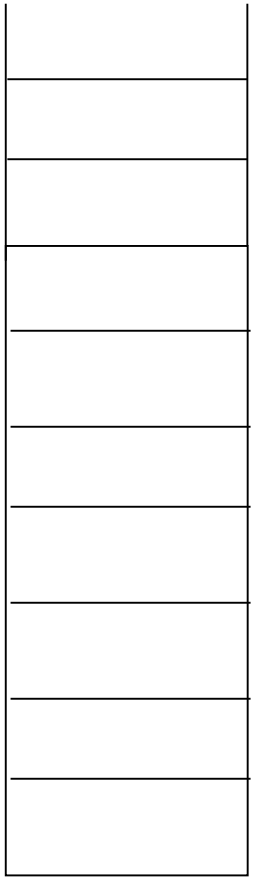



```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图



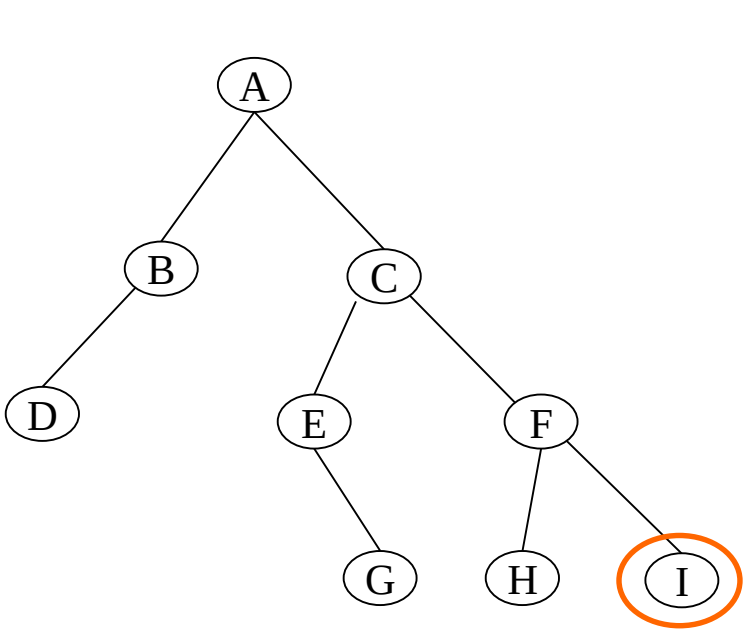
先根序列：**A B D C E G F H**



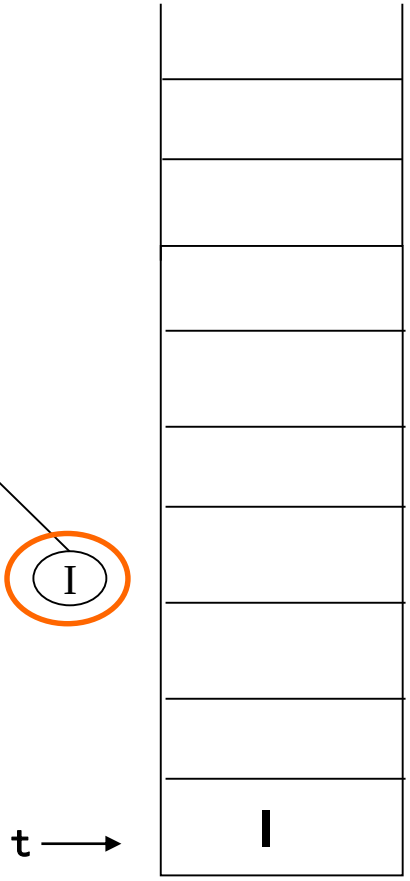
t ⟶

```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图

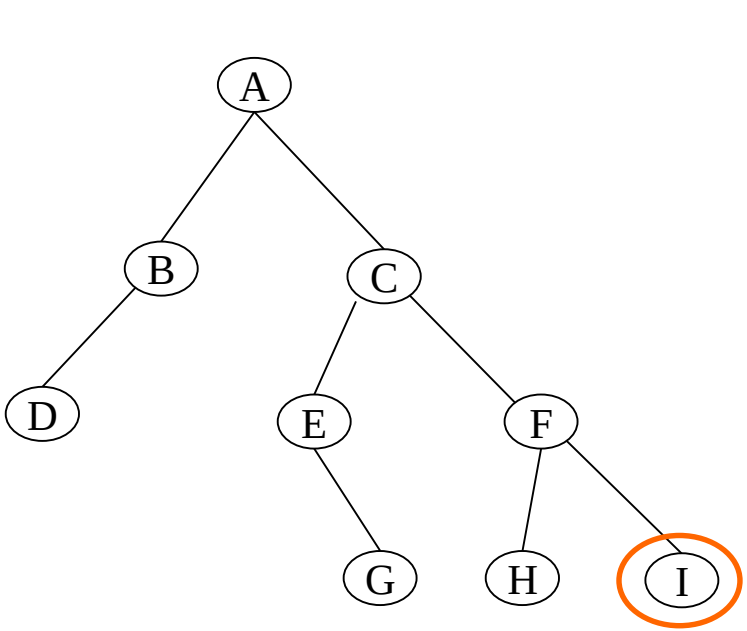


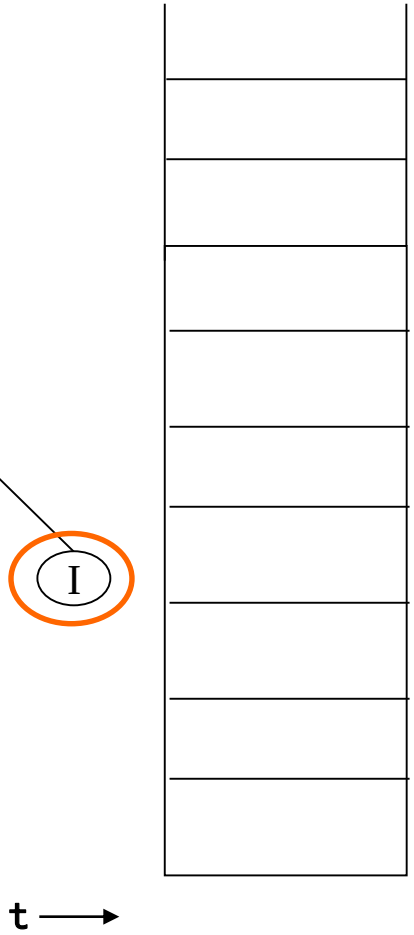
先根序列：**A B D C E G F H I**




```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游过程示意图



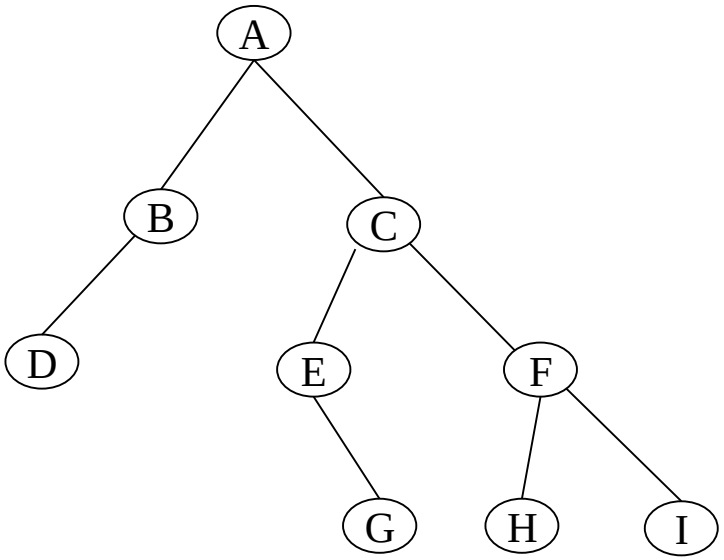


先根序列：**A B D C E G F H I**

```
Void npreOrder(Node p)
{
  Node c;
  Stack s;
  s = createEmptyStack();
  c = root(p);
  do
  {   while (c!=NULL)
      {
         visit(c);
         push(s, c);
         c = leftChild(c);
      }
      while ((c==NULL) &&
           (!isEmptyStack(s)))
      {
          c = rightChild (top(s));
         pop(s);
      }
   } while (c!=NULL);
}
```

先根周游的非递归算法（改进）

- 对于先根次序周游加进适当判断，可以减少进出栈的次数：
  - 访问一个结点之后，仅当该结点左、右子树都不空时才把结点的右子女推进栈。
  - 这样可以节省算法的时间与空间开销。



先根次序周游的非递归算法

- 算法思想：
  1. 从根结点开始，令变量p为根结点；
  2. 若p不为空，
     - 访问当前结点p，仅当该结点左、右子树都不空时才把结点的右子女推进栈；
     - 然后令p为当前结点的左子女结点，如此重复进行，直到当前结点的左子树为空。
  3. 若当前结点的右子树不为空，令p为当前结点的右子树，若当前结点的右子树为空，从栈中弹出栈顶元素赋给变量p。
  4. 然后，重复上述过程，当p为空时并且栈也为空时，周游结束。

教学网提供代码

中根次序周游的非递归算法

- 算法思想：
  - 与先根遍历基本类同，只是在沿左分支（左子树）向前搜索过程中将遇到的结点进栈，待遍历完左子树后，从栈顶退出结点并访问，然后再遍历右子树。

教学网提供代码

后根次序周游的非递归算法

- 算法思想：
  - 每个结点要到它们左、右子树都周游完时才得以访问
  - 在去周游它的左、右子树之前都需要进栈
  - 当结点出栈时，需要判断是从左子树回来（即刚周游完左子树，此时需要去周游右子树），还是从右子树回来（即刚周游完右子树，此时需要去访问这个结点)
  - 进栈的结点需要伴随着一个标记tag，tag定义为:

$$tag = \begin{cases} 1 & \text{表明周游它的左子树(第一次出栈,不能访问)} \\ 2 & \text{表明周游它的右子树(第二次出栈,可以访问)} \end{cases}$$

如何建立一个二叉树？

- 二叉树基础
  - 树与二叉树的基本概念
  - 二叉树的存储结构
  - 二叉树的周游算法
  - 建立一个二叉树
- 二叉树的应用
  - 哈夫曼树
  - 二叉检索树/排序树
- 树与树林

二叉树的周游

- 深度优先
  - 先根序列是：A B D C E G F H I；
  - 中根(对称序)序列是： D B A E G C H F I
  - 后根序列是： D B G E H I F C A
- 广度优先
  - ABCDEFGHI

二叉树周游运算的应用

- 利用二叉树的先根次序遍历结果和后根次序遍历结果是否能唯一的确定一棵二叉树？
  - （×）
- 利用二叉树的中根次序遍历结果和后根次序遍历结果是否能唯一的确定一棵二叉树？
  - （√）
  - why？？？

二叉树遍历的性质

- 性质1：已知二叉树的先序序列和中序序列，可以唯一确定一棵二叉树
  - 推论：已知二叉树的后序序列和中序序列，可以唯一确定一棵二叉树
- 性质2：已知二叉树的先序序列和后序序列，不能唯一确定一棵二叉树

重构树：已知前序和中序遍历序列

I. 确定树的根节点

- 树根是当前树中所有元素在前序序列中的第一个元素

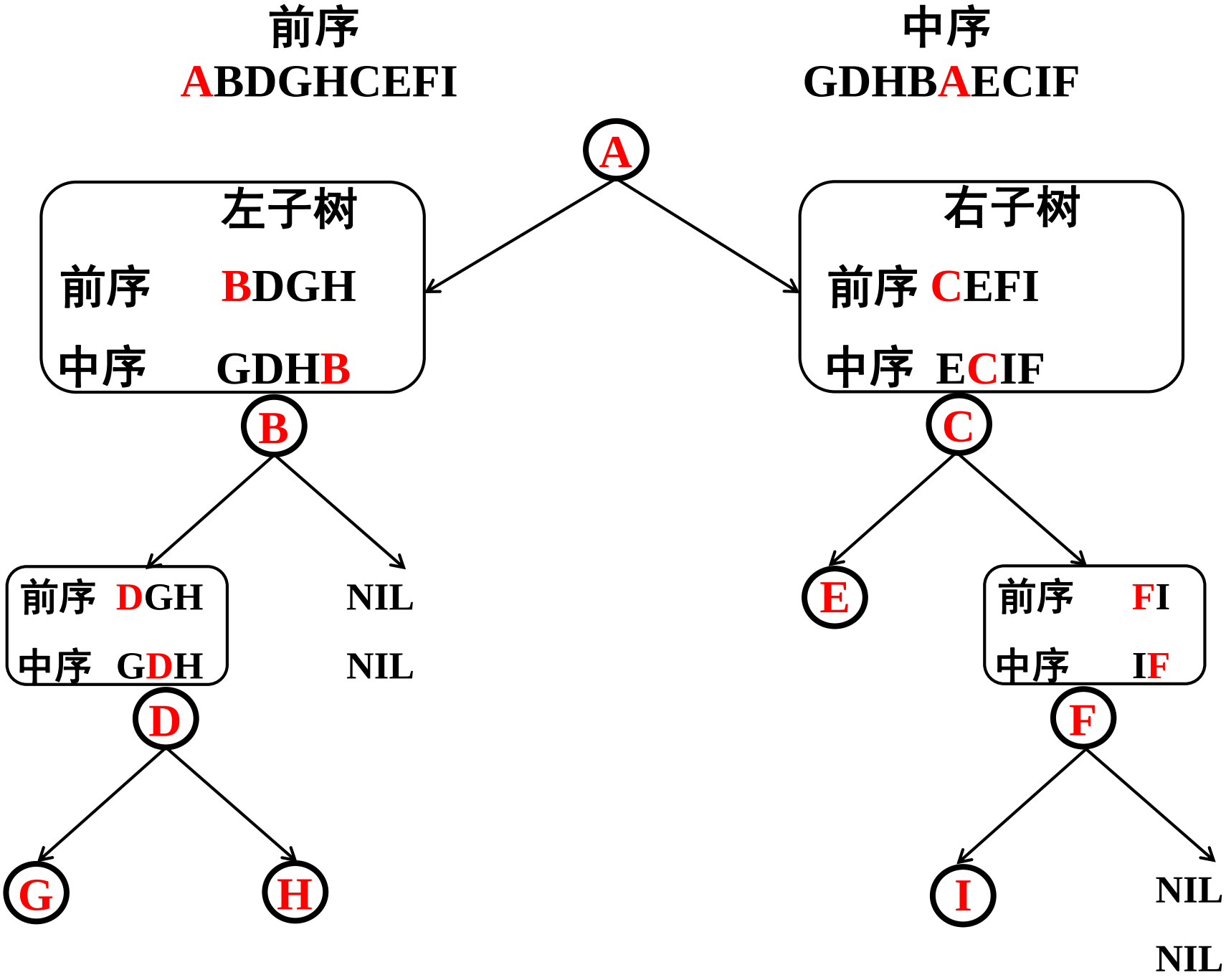
II. 求解树的子树

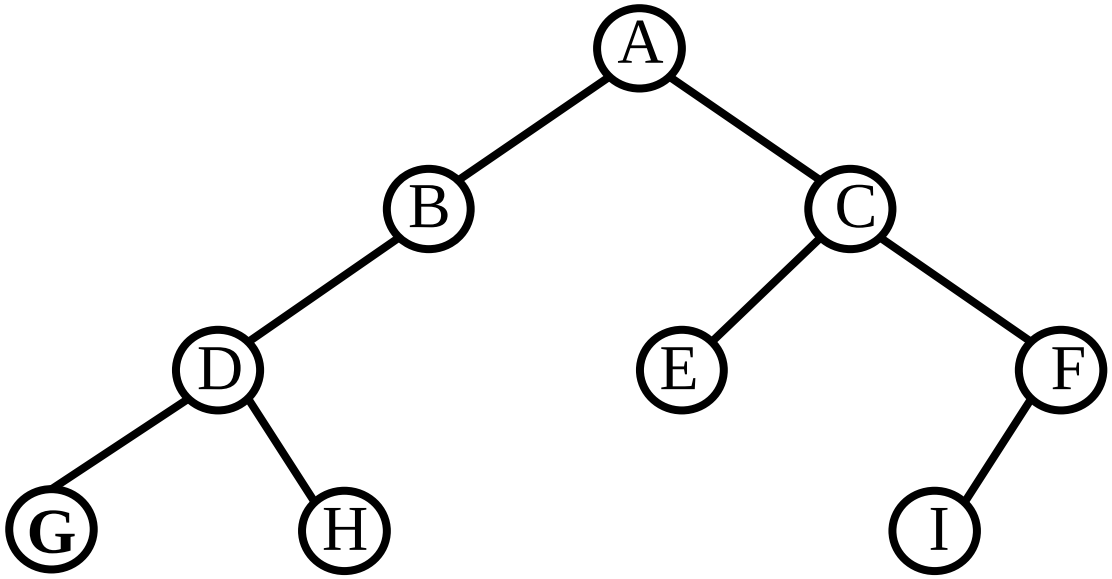
- 找出根节点在中序序列中的位置，根左边的所有元素就是左子树，根右边的所有元素就是右子树。
  - ✓ 若根节点左边或右边为空，则该方向子树为空
  - ✓ 若根节点左边和右边都为空，则根节点为叶节点

III. 递归求解树：将左、右子树分别看成一棵二叉树，重复上述步骤，直到所有节点完成定位。

举例

- (1)已知一棵二叉树的前序序列和中序序列分别为ABDGHCEFI和GDHBAECIF，请画出此二叉树。
- (2)已知一棵二叉树的中序序列和后序序列分别为BDCEAFHG和DECBHGFA，请画出此二叉树。
- (3)已知一棵二叉树的前序序列和后序序列分别为AB和BA，请画出这两棵不同的二叉树。

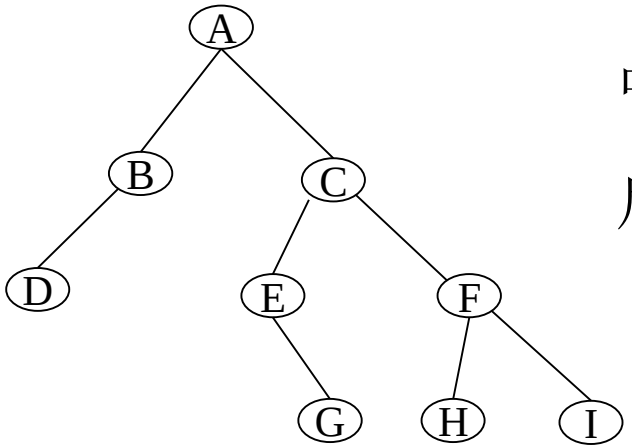

前序
ABDGHCEFI
中序
GDHBAECIF
A
左子树
前序 BDGH
中序 GDHB
右子树
前序 CEFI
中序 ECIF
B
C
前序 DGH
中序 GDH
NIL
NIL
E
前序 FI
中序 IF
D
F
G
H
I
NIL
NIL


A
B
C
D
E
F
G
H
I

思考

- 前序、中序、后序遍历结果中，所有叶结点都以相同的顺序出现吗？
- 前、中、后序哪几种结合可以恢复二叉树的结构？
  - 已知某二叉树的后根序列为{DBGEHIFCA },
    中根序列为{DBAEGCHFI}；
    则先根序列为_____。

示例



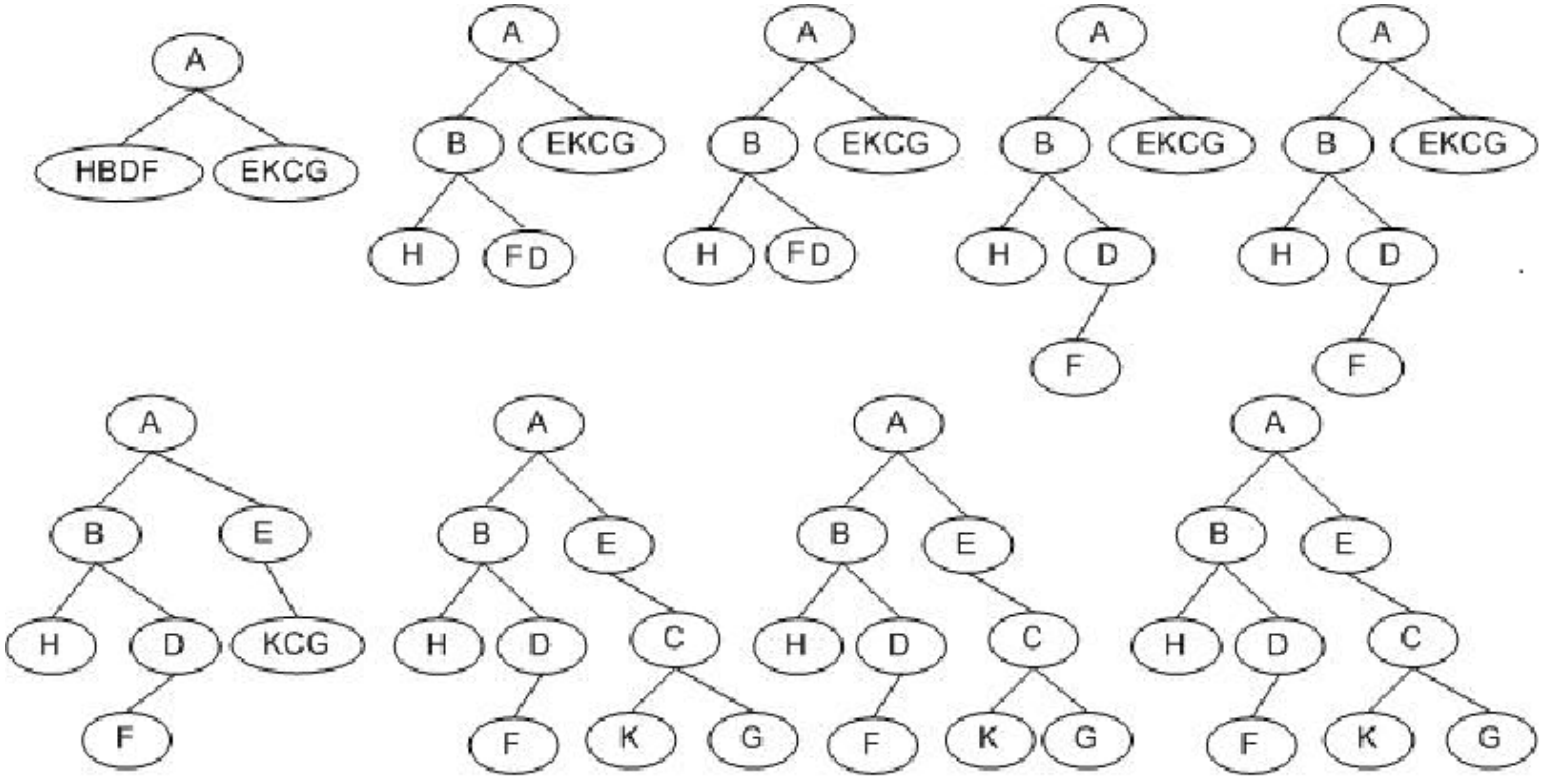
中根序列是： D B A E G C H F I

后根序列是： D B G E H I F C A

	A
DB	**EGCHFI**
DB	**GEHIFC**

示例

- 先根序列ABHDFECKG
- 中根序列HBFDAEKCG

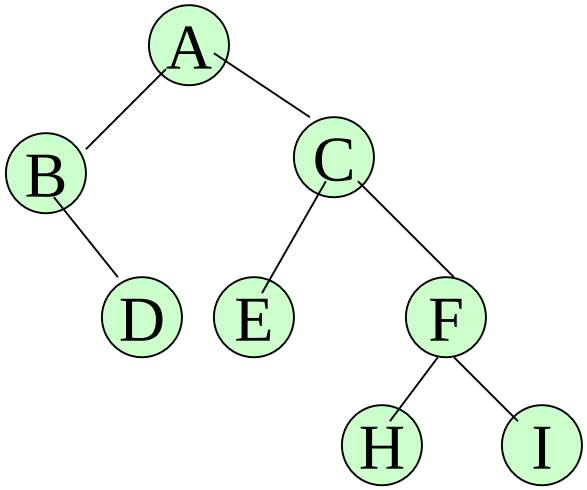


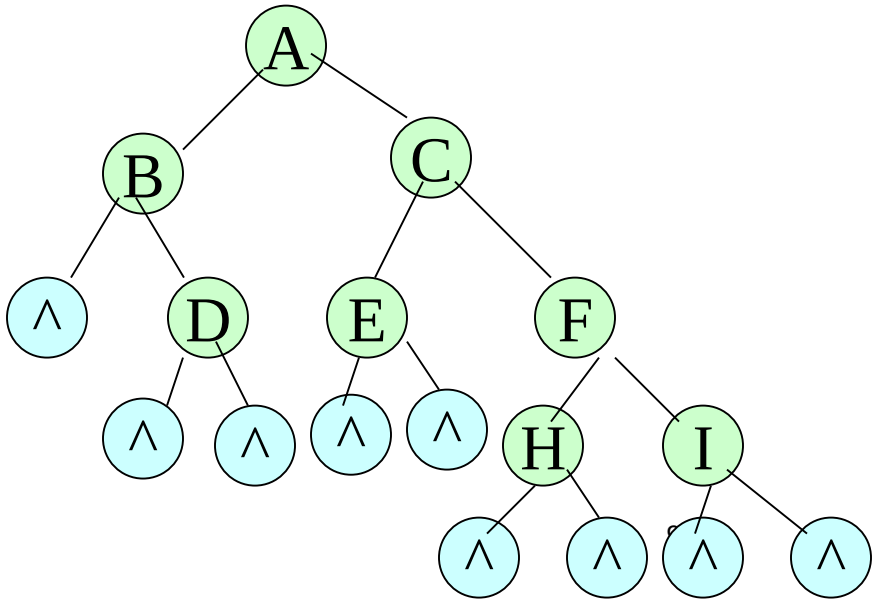
依据一个特殊数据序列建立二叉树

- 上述方法的困难：
  - 先根、中根、后根序列中任一个不能唯一确定一棵二叉树，所以不能直接用。
  - 中根序列和先根序列 或 中根序列和后根序列 可以唯一确定，但是不方便。
- 解决方法：读入一个特殊数据序列来建立二叉树
  - 将二叉树进行扩充，成为一棵扩充的二叉树，
  - 此时利用此扩充二叉树的先根/后根/层次序列可以唯一确定其原二叉树。

用扩充二叉树的先根列生成二叉树

- ABDCEFHI>>>>
  - @表示空节点
  - 设从A开始，A的左子树为B，B的左子树为空，因此D一定是B的右子结点
  - D有两个空子结点，说明C一定是A的右子结点





用扩充二叉树的先根列生成二叉树

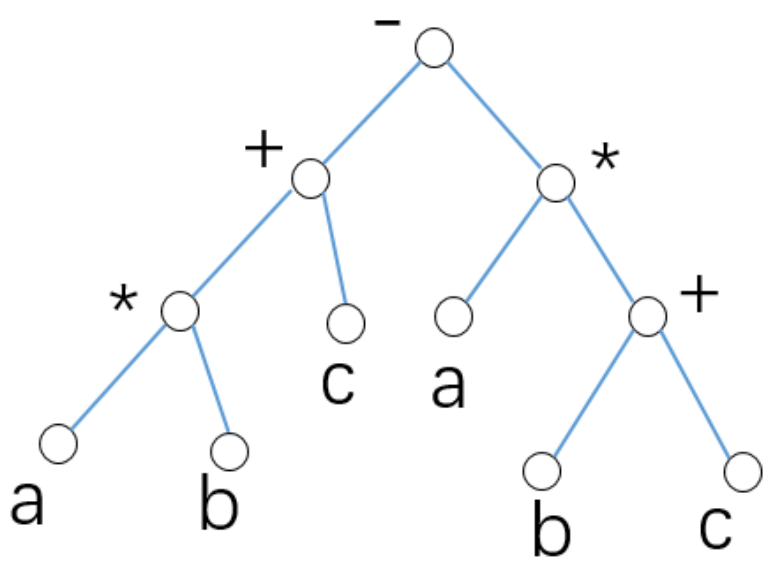
- 二叉树要先扩充为扩充的二叉树，然后再按先根周游，其中@表示为空结点

- 例：按先根序列建立二叉树
  - 读入A B D @ @ @ C E @ G @ @ F H @ @ I @ @，
  - 建立相应的llink-rlink表示法

教学网：用扩充的先根序列创建二叉树代码

表达式二叉树

- 前序（前缀）：-+ * a b c * a + b c
- 中序：a * b + c -a * (b + c)
- 后序（后缀）：a b * c + a b c + * -



打印中缀表达式

```
1.  void Expression (BinaryTreeNode<T>* root) {        // 打印中缀表达式
2.     int rootpriority, leftpriority, rightpriority;
3.     if (root != NULL)
4.     {
5.         rootpriority = priority(root);
6.         if (root->leftchild() != NULL)          leftpriority = priority(root-
    >leftchild());
7.         else  leftpriority = INFINITY;
8.         if (root->rightchild() != NULL)  rightpriority = priority(root->rightchild());
9.         else   rightpriority = INFINITY;
10.        if ((root->leftchild() != NULL) && (leftpriority != INFINITY)
                                          && (leftpriority < rootpriority))  // 左符小于父
11.            cout << "(";                              // 需要打印左括号
```

打印中缀表达式

```
12.         Expression(root->leftchild());          // 递归访问左子树
13.         if ((root->leftchild() != NULL) && (leftpriority != INFINITY)
14.                                                 && (leftpriority < rootpriority))
15.                     cout << ")";                                    // 打印左子树的右括号
16.         Visit(root->value());                                       // 打印运算符或操作数
17.         if ((root->rightchild() != NULL) && (rightpriority != INFINITY)
                                                    &&  (rightpriority <= rootpriority ))
            // 右符小或等于父
18.                     cout << "(";                                    // 需要打印括号
19.         Expression(root->rightchild());         // 递归访问右子树
20.         if ((root->rightchild() != NULL) && (rightpriority != INFINITY)
21.                                                 &&  (rightpriority <= rootpriority))
22.                     cout << ")";                                    // 打印右子树的右括号
23.     }
24. }
```

小结

- 二叉树的定义、性质；
- 二叉树的存储结构；
- 在熟悉上述内容的基础上，重点掌握二叉树的周游及生成算法。