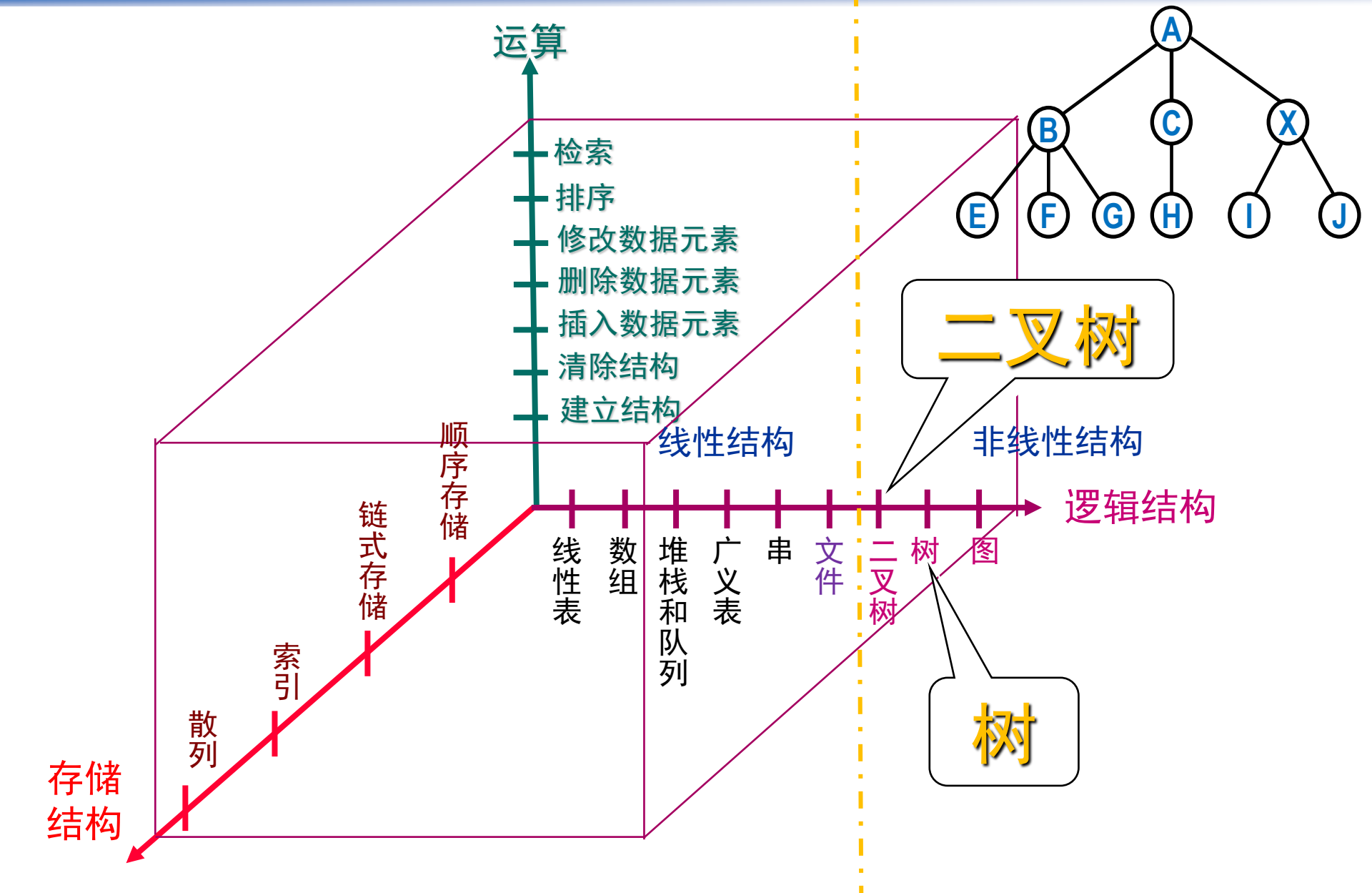
数据结构与程序设计
(Data Structure and Programming)

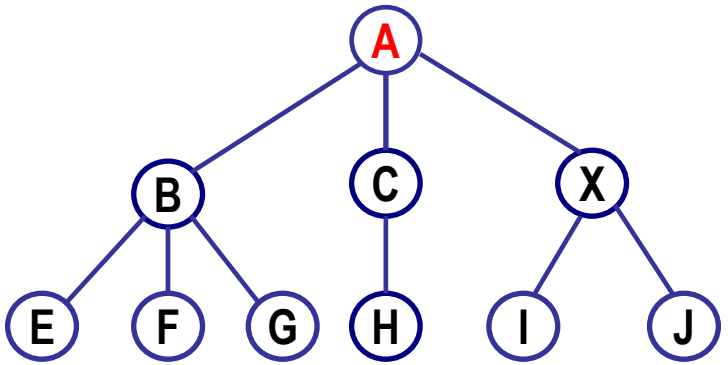
树

Tree

北航计算机学院 林学练

数据结构的基本问题空间





线性结构	树结构
第一个数据元素 (无前驱)	根结点(无前驱)
最后一个数据元素 (无后继)	叶子结点(无后继)
其它数据元素(一个前驱、 一个后继)	树中其它结点(一个前驱、 多个后继)

本章内容

1 树的基本概念
2 树的存储结构
3 二叉树
4 二叉树的存储结构
5 二叉树的遍历
6 线索二叉树*
7 二叉查找树
8 平衡二叉树
9 堆
10 哈夫曼(Huffman)树

1 树的基本概念

1.1 树的定义

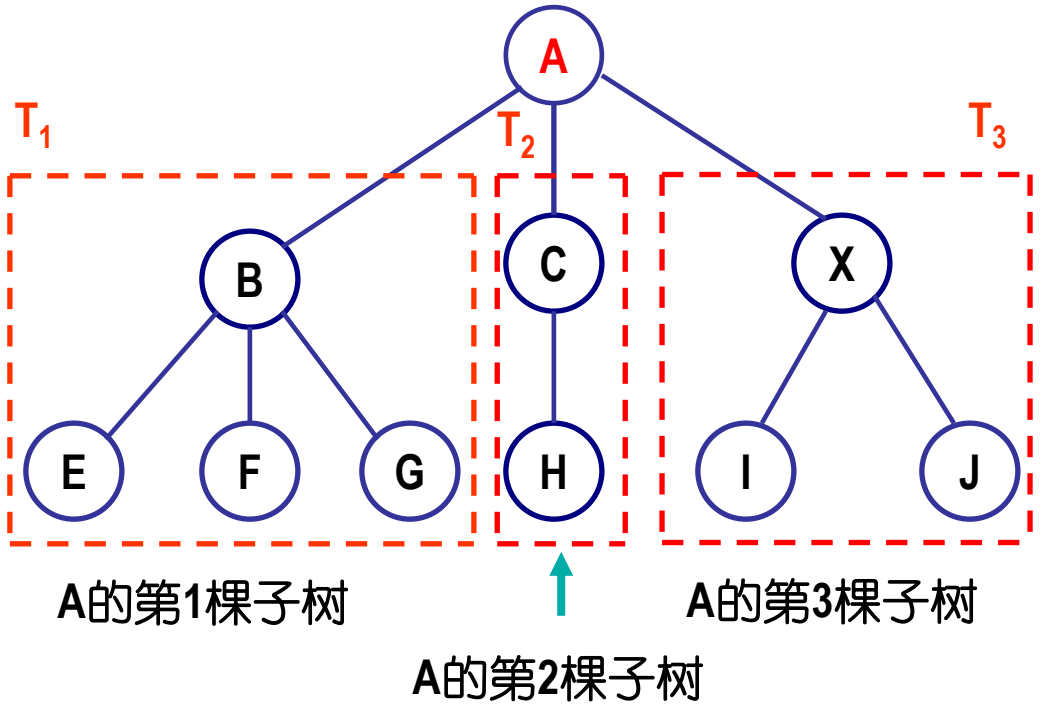
树是由$n \geq 0$个结点组成的有穷集合（不妨用符号D表示）以及结点之间关系组成的集合构成的结构，记为T（当n=0时，称T为空树），满足：

在任何一棵非空的树中，有一个特殊的结点$t \in D$，称之为该树的根结点；其余结点$D-\{t\}$被分割成$m>0$个不相交的子集$D_1, D_2, \ldots, D_m$，其中，每一个子集D_i分别构成一棵树，称之为t的子树 。

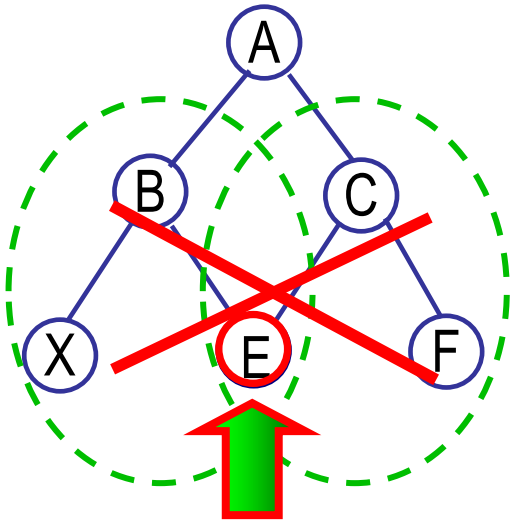
递归定义

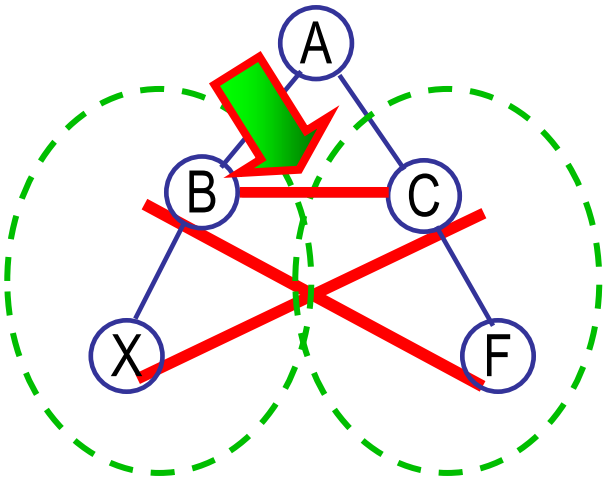
结点集合

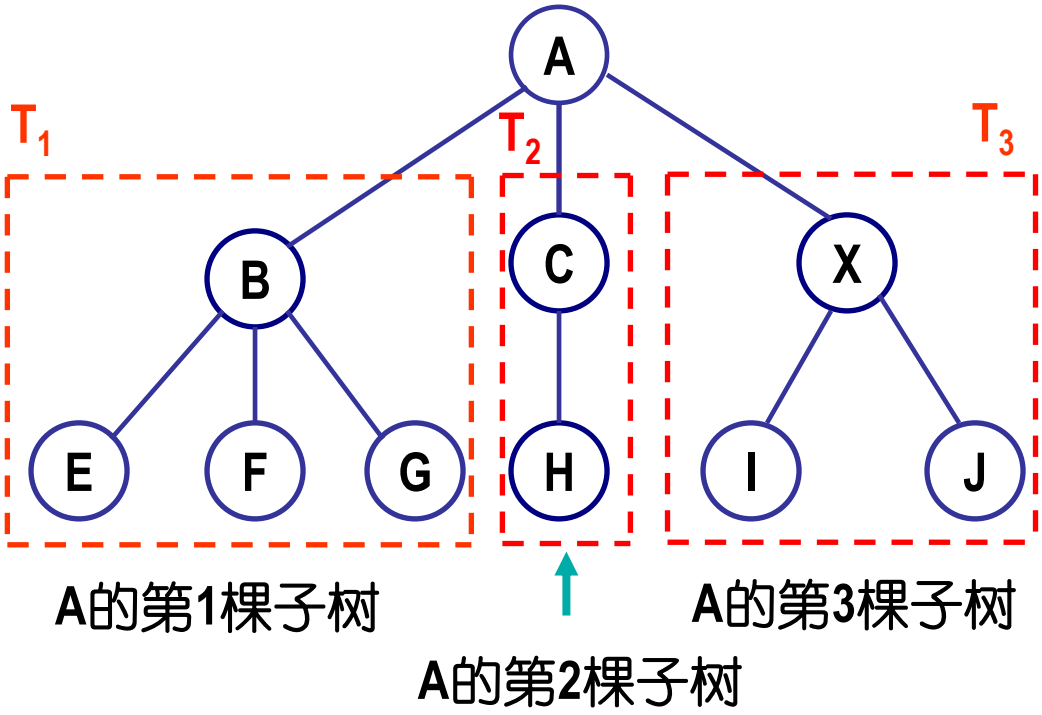
D={ A,B,C,E,
F,G,H,I,
J,X }



不相交的子集







1.2 树(逻辑上)的特点

1.有且仅有一个结点没有前驱结点，该结点为树的**根结点**；

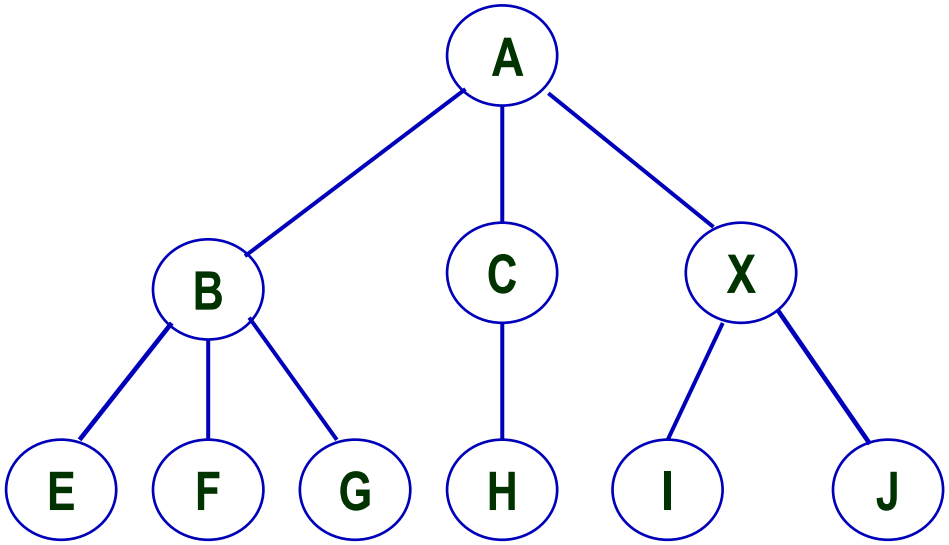
2.除了根结点外，每个结点有且仅有**一个直接前驱结点**；

3.包括根结点在内，每个结点**可以**有**多个后继结点**。

1.3 树的逻辑表示方法

1. 树形表示法

借助自然界中一棵倒置的树的形状来表示数据元素之间层次关系的方法。



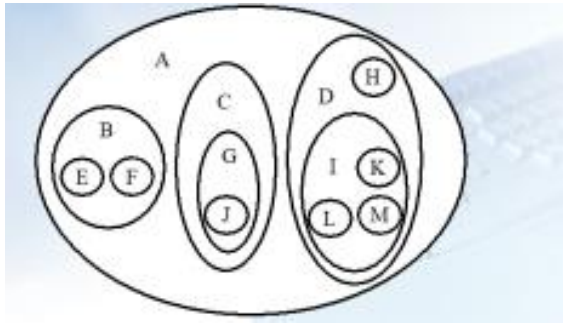
1.3 树的逻辑表示方法

1. 树形表示法

2*. 文氏图表示法

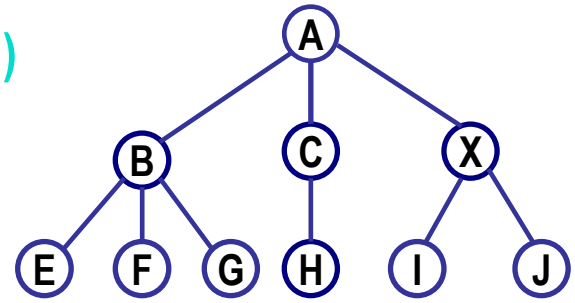
3*. 凹入表示法

4*. 嵌套括号法(广义表表示法)



A(B(E, F, G), C(H), X(I, J))

第1棵子树　第2棵子树　第3棵子树



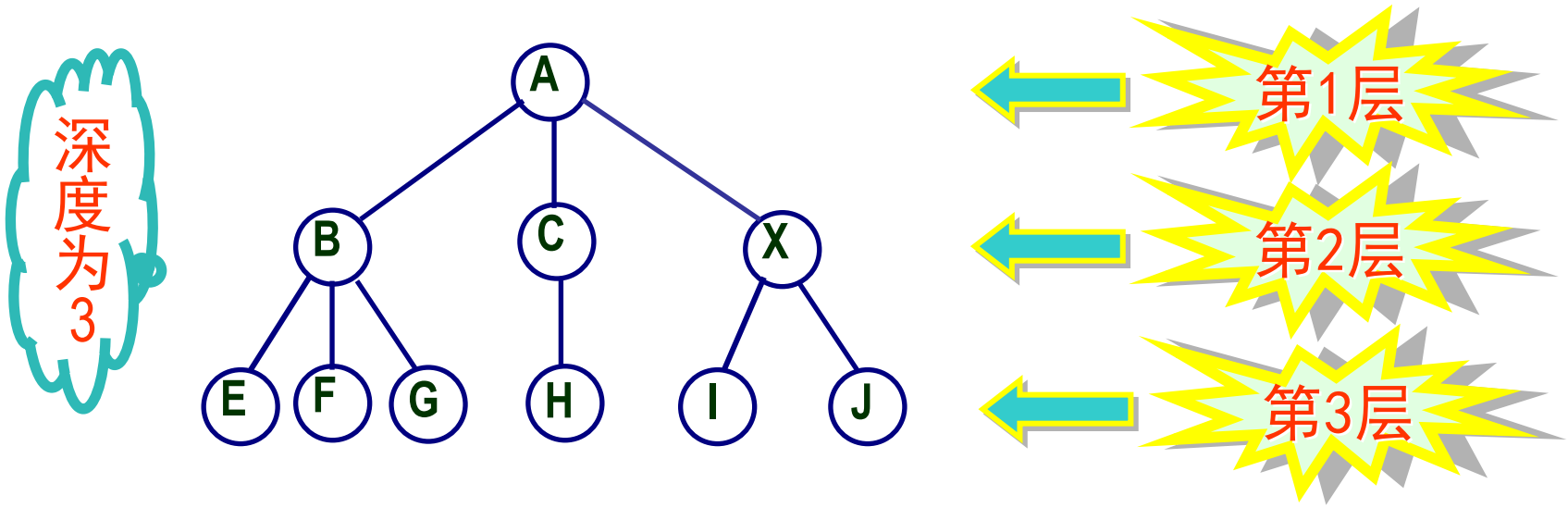
一个长度为n≥0 的**广义表**LS定义为：

$$LS = (a_1, a_2, \ldots\ldots, a_{n-1}, a_n)$$

其中，a_i为表中元素。a_i可以是原子元素，也可以是一个子表。

1.4 基本名词术语

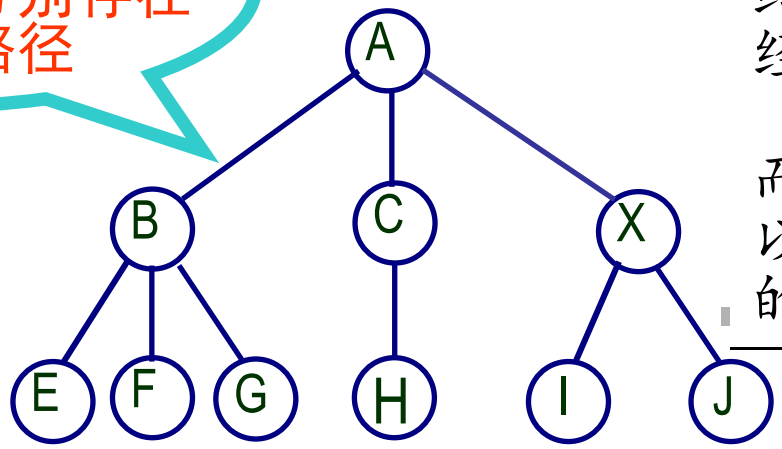
1. 结点的度：该结点拥有的子树的数目。
2. 树的度：树中结点度的最大值。
3. 叶结点：度为0 的结点。(终端结点)
4. 分支结点：度非0 的结点。(非终端结点)
5. 树的层次：根结点为第1层，若某结点在第i 层，则其孩子结点(若存在)为第i+1层。
6. 树的深度：树中结点所处的最大层次数。(高度)



7. 路径：对于树中任意两个结点d_1和d_j，若在树中存在一个结点序列$d_1, d_2, \cdots d_i, \cdots, d_j$，使得$d_i$是$d_{i+1}$的双亲$(1 \leqslant i < j)$，则称该结点序列是从$d_1$到$d_j$的一条路径。路径的长度为$j-1$。

8. 祖先与子孙：若树中结点d到d_s存在一条路径，则称d是d_s的祖先，d_s是d的子孙。

从根结点到树中其余结点均分别存在一条唯一路径



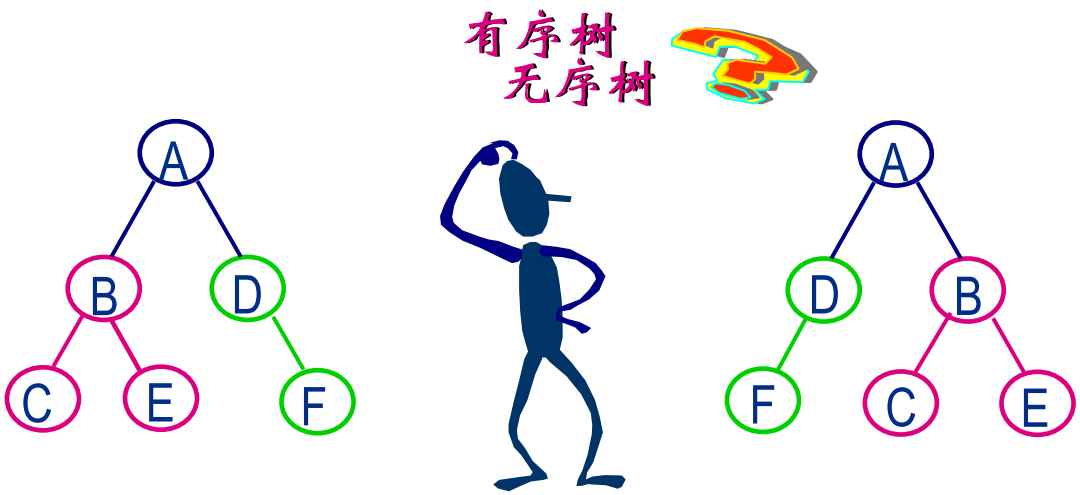
一个结点的祖先是从根结点到该结点路径上所经过的所有结点；

而一个结点的子孙则是以该结点为根的子树上的所有其他结点。

结点的子树的根称为该结点的**孩子**(child)，相应地，该结点称为孩子结点的**父结点**(或双亲，parent)。同一个双亲的孩子之间互称**兄弟**。

9. 树林(森林)：m≥0 棵不相交的树组成的树的集合。

10. 树的有序性：若树中结点的子树的相对位置不能随意改变，则称该树为**有序树**，否则称该树为**无序树**。



2 树的存储结构

- 顺序存储结构
- 链式存储结构（居多）

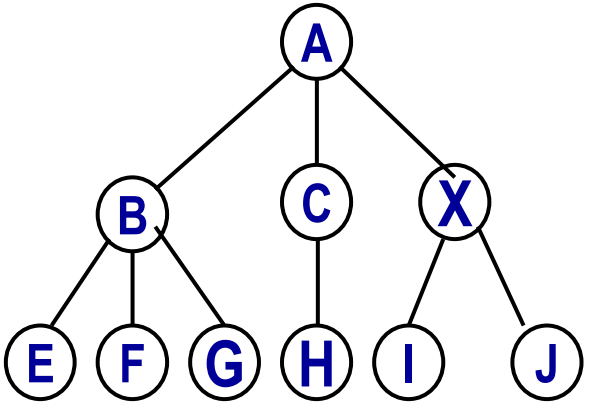
主要取决于要对树进行何种操作

无论采用何种存储结构，需要存储的信息有：

- 结点本身的数据信息；
- 结点之间存在的关系（分支）。

2.1 多重链表结构

1. 定长结点的多重链表结构

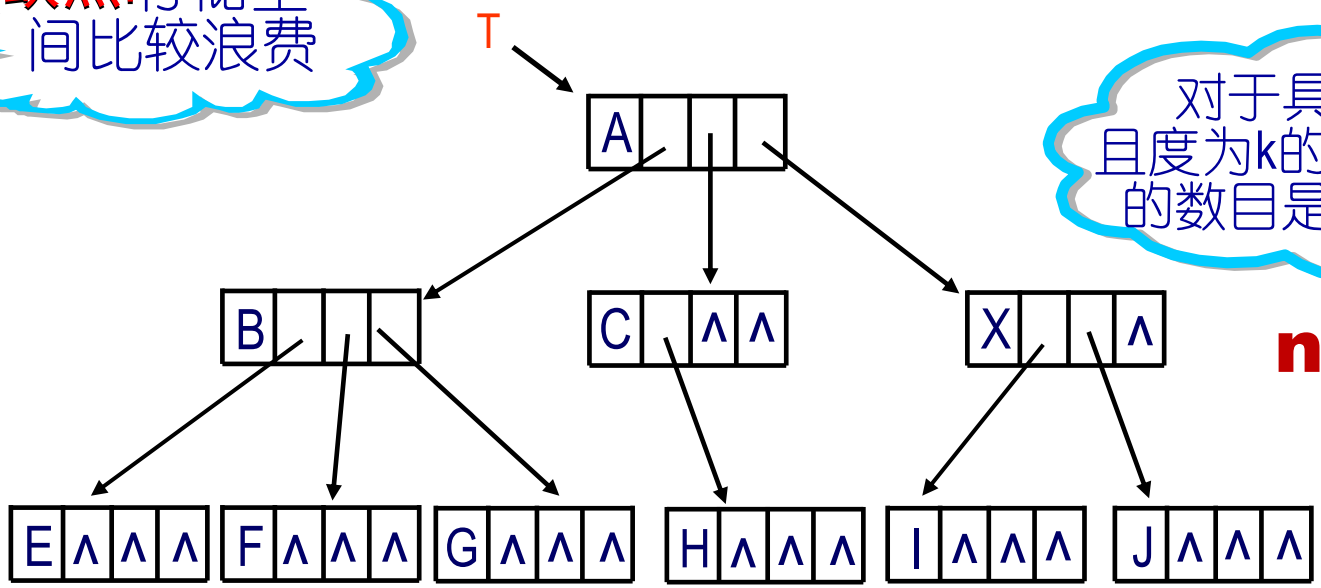


链结点的构造

data	$child_1$	$child_2$	……	$child_k$

k为树的度

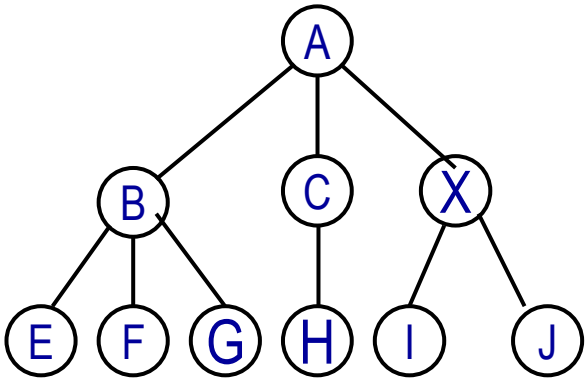
缺点:存储空间比较浪费



对于具有n个结点且度为k的树，空指针域的数目是多少？

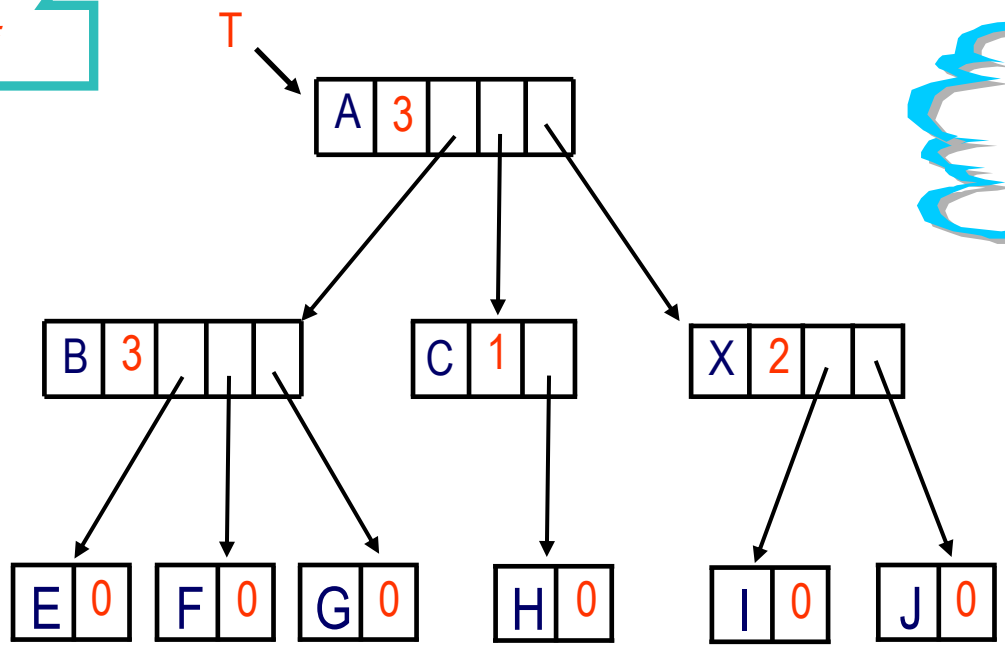
n(k-1) + 1

2. 不定长结点的多重链表结构



链结点的构造

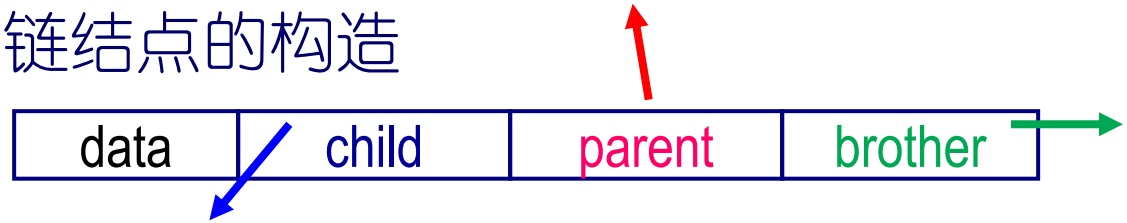
结点的度

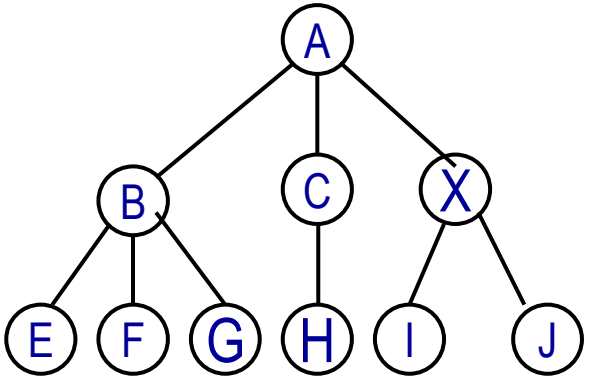


缺点:对树的操作相对不方便。

2.2 三重链表结构

链结点的构造



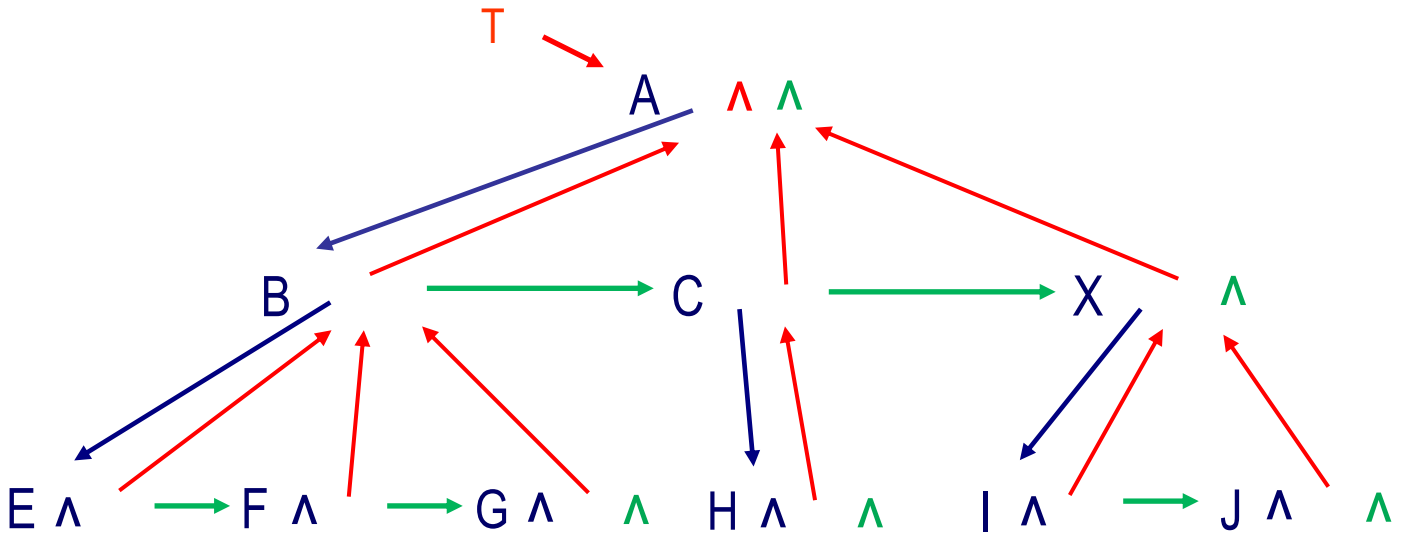


其中,data 为数据域;

child 为指针域,指向该结点的第1个孩子结点;

parent 为指针域,指向该结点的双亲结点;

brother 为指针域,指向右边第一个兄弟结点。



3 二叉树

3.1 二叉树的定义

二叉树是n≥0个结点的有穷集合D与D上关系的集合R构成的结构，记为T。当n=0时，称T为空二叉树；否则，它为包含了一个根结点以及两棵不相交的、分别称之为左子树与右子树的二叉树。

递归定义

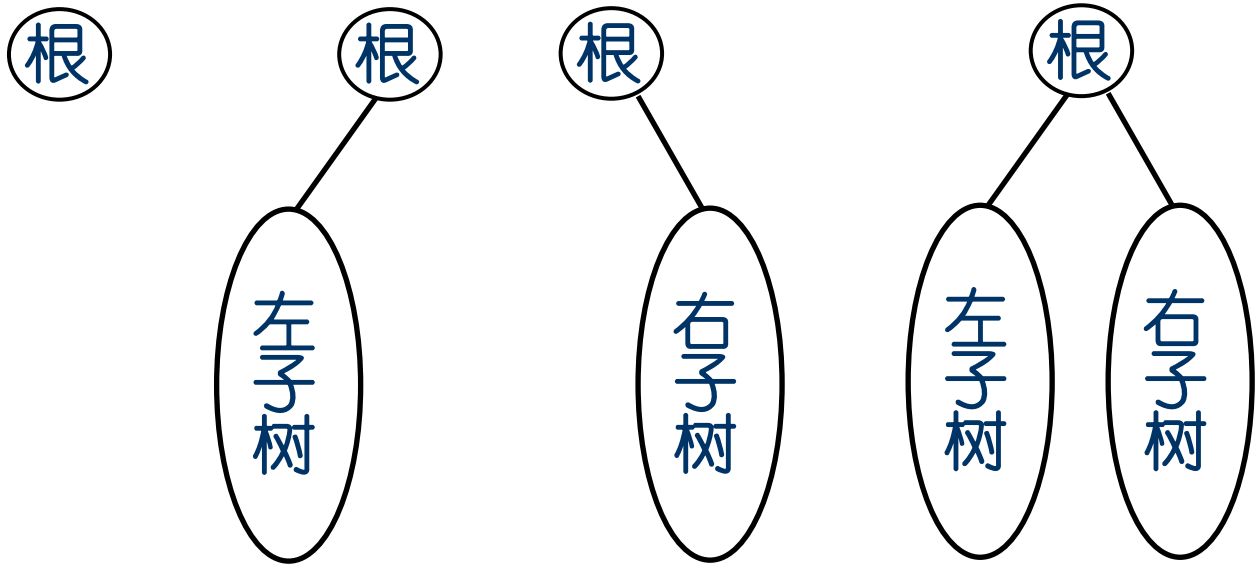
[思考]二叉树为何重要*

- 一些重要的算法或方法对应到二叉树
  如二分查找、哈夫曼编码
- 结构简单，便于管理和计算。
  如每层的数量最多为2^n，容易映射到线性表

二叉树的基本形态

5种！

(空)

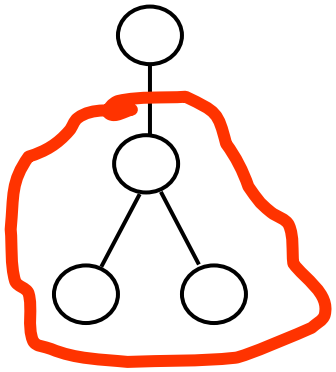


思考 下面的说法正确与否

✗1) 度为2 的 树 是二叉树。

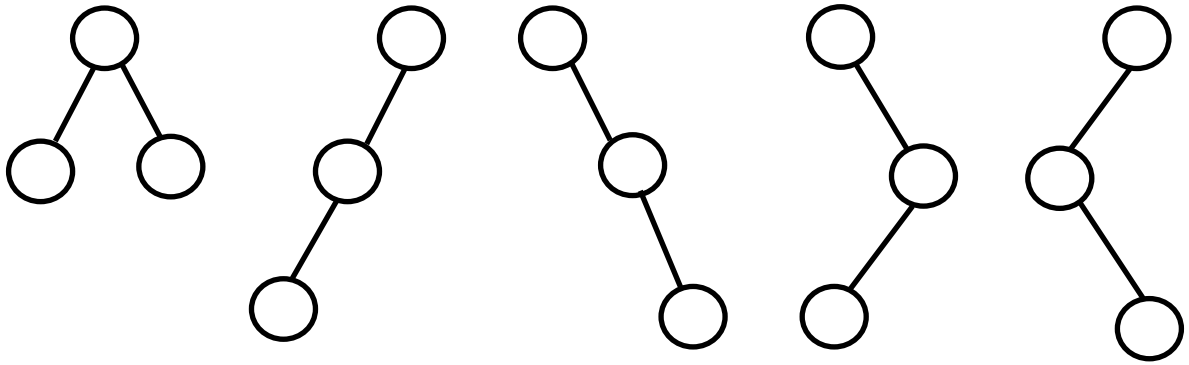
✗2) 度为2 的有序树 是二叉树。

没有区分左右子树



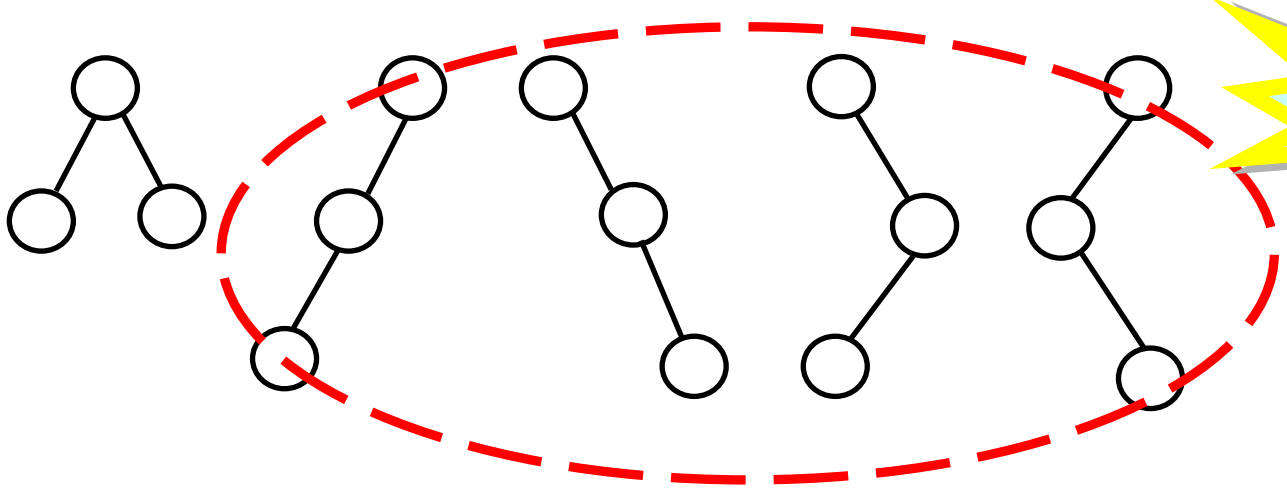
结论:子树有严格的左、右之分且度≤2的树是二叉树。

3）具有三个结点的二叉树可以有多少种形态?



5种

✗ 4）具有三个结点的树可以有 **5** 种形态

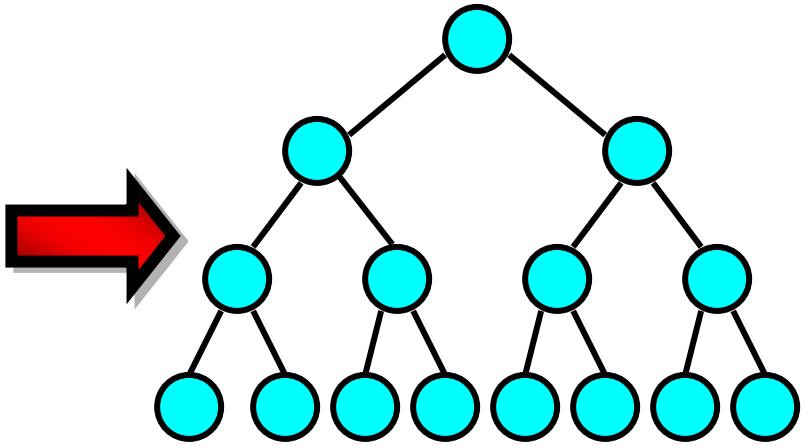


1种

3.2 两种特殊形态的二叉树

1. 满二叉树 (full binary tree)

若一棵二叉树中的结点，或者为叶结点，或者具有两棵非空子树，并且叶结点都集中在二叉树的最下面一层. 这样的二叉树为满二叉树。

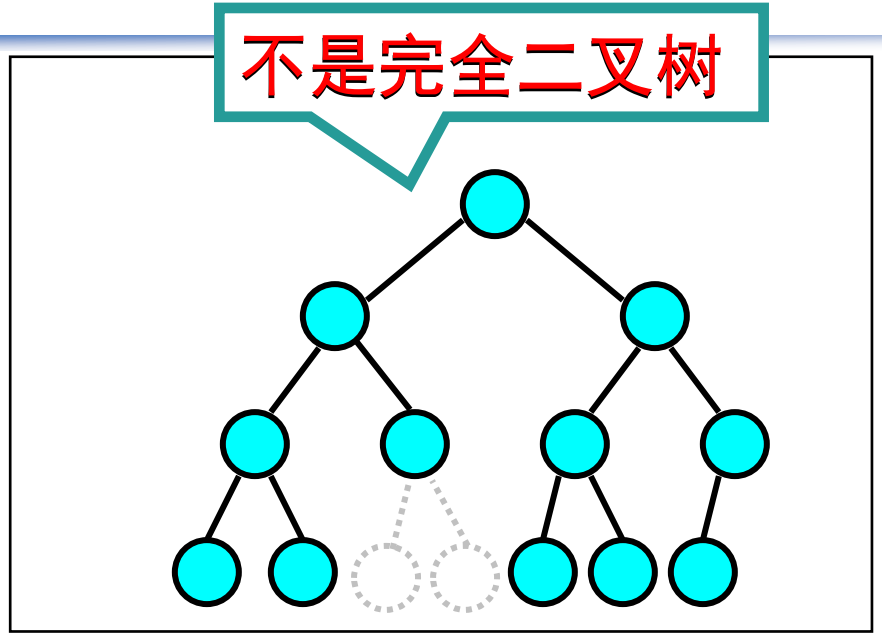


3.2 两种特殊形态的二叉树

1. 满二叉树

full binary tree

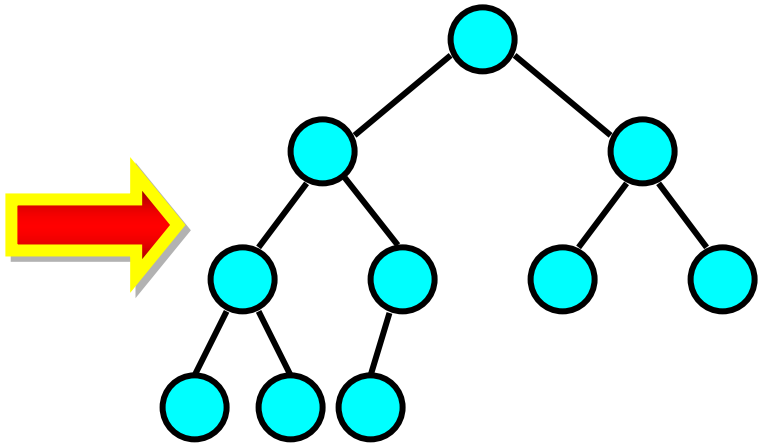
若一棵二叉树中的结点，或者为叶结点， 或者具有两棵非空子树，并且叶结点都集中在二叉树的最下面一层. 这样的二叉树为满二叉树。



2. 完全二叉树

complete binary tree

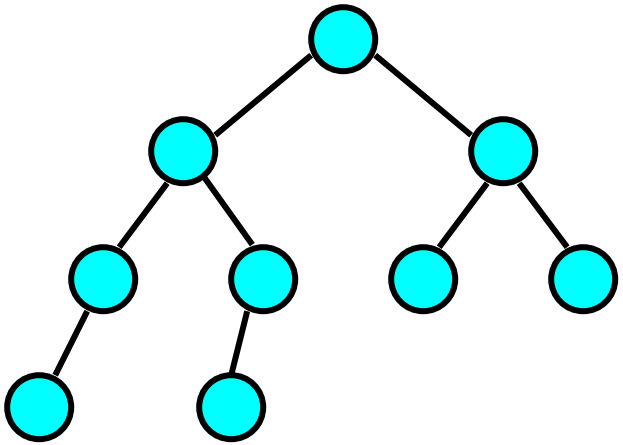
若一棵二叉树中只有最下面两层的结点的度可以小于2，并且最下面一层的结点(叶结点)都依次排列在该层从左至右的位置上. 这样的二叉树为完全二叉树。



3.3 二叉树的性质

1. 具有**n**个结点的非空二叉树共有 **n-1** 个分支。

证明：除了根结点以外，每个结点有且仅有一个双亲结点，即每个结点与其双亲结点之间仅有一个分支存在， 因此，具有n个结点的非空二叉树的分支总数为n–1。



n=9

2. 若非空二叉树有n_0个叶结点，有n_2个度为2的结点，则 $n_0=n_2+1$

证明:

设该二叉树有n_1个度为1的结点,结点总数为n,有

$n=n_0+n_1+n_2$ --------(1)

设二叉树的分支数目为B，根据性质1，有

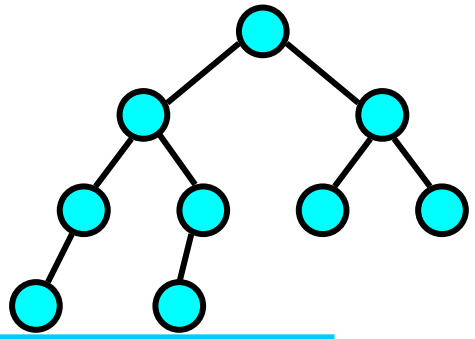
$B=n-1$ -------- (2)

这些分支来自度于为1的结点与度为2结点，即

$B=n_1+2n_2$ ---------(3)

联列关系(1),(2)与(3),可得到

$n_0=n_2+1$



推论 $n_0=n_2+2n_3+3n_4+ \ \ldots \ +(m-1)n_m+1$

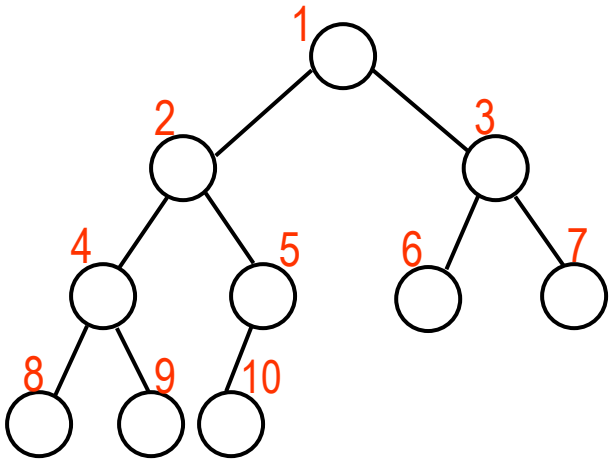
3. 若对具有n个结点的**完全二叉树**按照层次从上到下，每层从左到右的顺序进行编号，则编号为i的结点具有以下性质：

(1) 当i=1，则编号为i的结点为二叉树的根结点；
若i>1，则编号为i 的结点的双亲的编号为⌊i/2⌋；

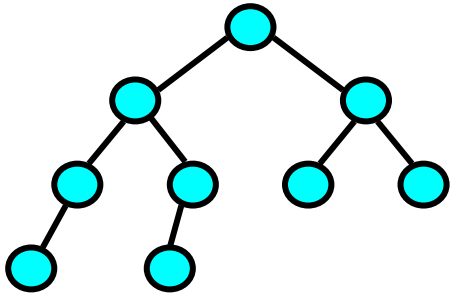
(2) 若2i>n，则编号为i 的结点无左子树；
若2i≤n，则编号为i 的结点的左孩子的编号为2i；

(3) 若2i+1>n，则编号为i 的结点无右子树；
若2i+1≤n，则编号为i 的结点的右孩子的编号为2i+1。

n=10



4. 非空二叉树的第i 层最多有2^{i-1}个结点(i≥1)。



证明(采用归纳法)

(1) 当i=1时,结论显然正确。非空二叉树的第1层有且仅有一个结点,即树的根结点.

(2) 假设对于第j层($1 \le j \le i-1$)结论也正确,即第j层最多有2^{j-1}个结点.

(3) 由定义可知, 二叉树中每个结点最多只能有两个孩子结点。若第i–1层的每个结点都有两棵非空子树,则第i层的结点数目达到最大.而第i–1层最多有2^{i-2}个结点已由假设证明,于是, 应有 $2 \times 2^{i-2} = 2^{i-1}$

5. 深度为h 的非空二叉树最多有2^h-1个结点。

证明:

由性质2可知,若深度为h的二叉树的每一层的结点数目都达到各自所在层的最大值,则二叉树的结点总数一定达到最大。即有

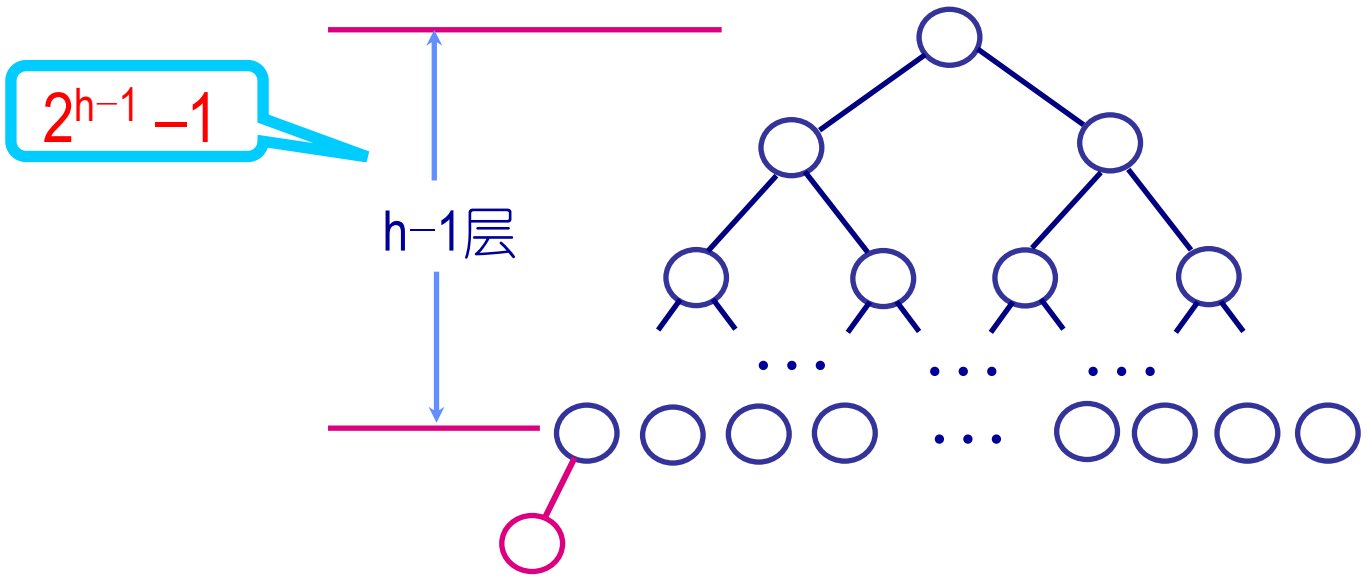
$$2^0+2^1+2^2+\cdots+2^{i-1}+\cdots+2^{h-1}=2^h-1$$

结论

深度为h 且具有2^h-1个结点的二叉树为满二叉树。

思 考

深度为h 的完全二叉树至少有多少个结点



6. 具有n个结点的非空完全二叉树的深度为 $h=\lfloor \log_2 n \rfloor+1$.

证明:

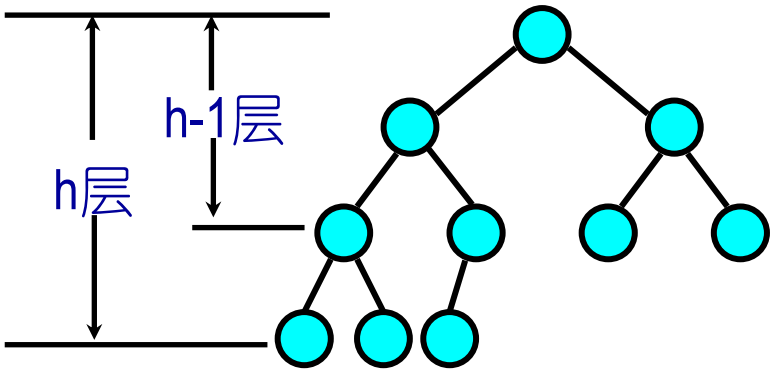
设深度为h，

则： $2^{h-1}-1 < n <= 2^h-1$

即： $2^{h-1} <= n < 2^h$

即： $h-1 <= \log_2 n < h$

因此： $h=\lfloor \log_2 n \rfloor+1$



练习

若一二叉树中有20个叶子结点，则其度为2的结点___个。

若一棵二叉树有1001个结点,且无度为1的结点,则叶结点的个数为___

A)498　B)499　C)500　D)501

一个具有767个结点的完全二叉树，其叶子结点个数为___

A)383　B)384　C)385　D)386

若一棵深度为6的完全二叉树的第6层有3个叶结点,则该二叉树共有叶结点的个数为___

A)17　B)18　C)19　D)20

3.4 二叉树的基本操作

若x是二叉树的根结点，或二叉树中不存在结点x,则返回“空”

1. INITIAL(T) 初始(创建)一棵二叉树。
2. ROOT(T)或ROOT(x) 求二叉树T的根结点，或求结点x所在二叉树的根结点。
3. PARENT(T,x) 求二叉树T中结点x的双亲结点。
4. LCHILD(T,x)或RCHILD(T,x) 分别求二叉树T中结点x的左孩子结点或右孩子结点。
5. LDELETE(T,x)或RDELETE(T,x) 分别删除二叉树T中以结点x为根的左子树或右子树。

遍历

重点

6. TRAVERSE(T) 按照某种次序(或原则)依次访问二叉树T中各个结点，得到由该二叉树的所有结点组成的序列。
7. LAYER(T,x) 求二叉树中结点x所处的层次。
8. DEPTH(T) 求二叉树T的深度。
9. DESTROY(T) 销毁一棵二叉树。

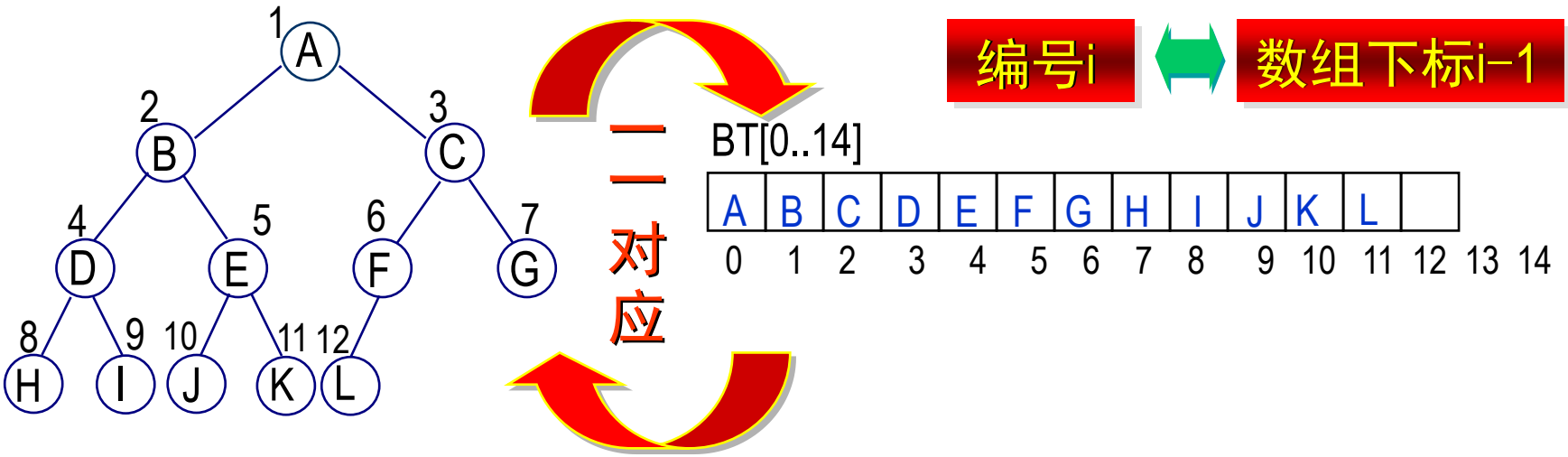
……

4 二叉树的存储结构

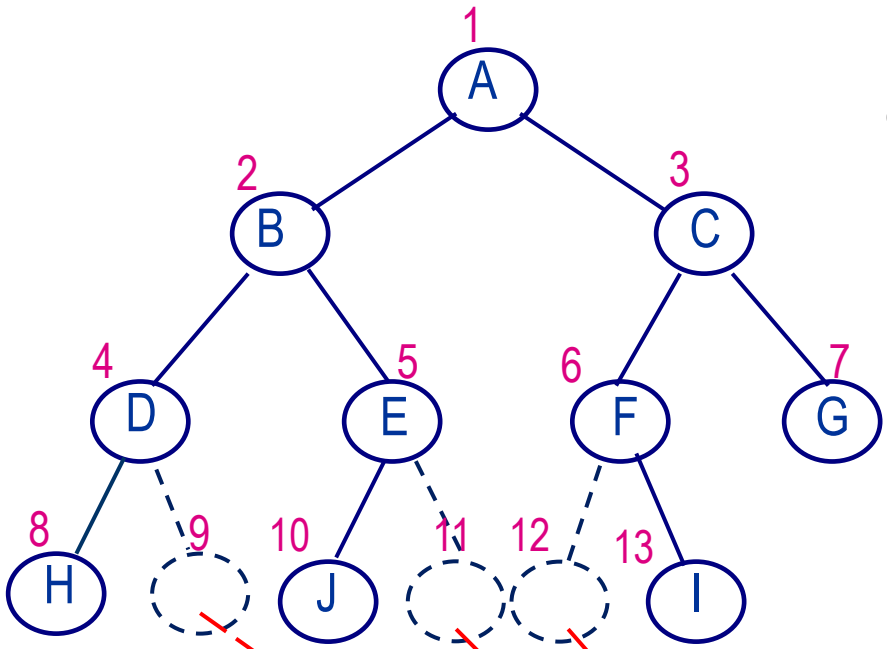
4. 1. 二叉树的顺序存储结构

1. 完全二叉树的顺序存储结构

根据完全二叉树的性质5，对于深度为h的完全二叉树，将树中所有结点的数据信息按照编号的顺序依次存储到一维数组BT[0..2^h-2]中，由于编号与数组下标一一对应，该数组就是该完全二叉树的顺序存储结构。



2. 一般二叉树的顺序存储结构



“完全二叉树”

BT[0..14]

	A	B	C	D	E	F	G	H		J			I		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

结论（对于一般二叉树）

对于一般二叉树，只须在二叉树中“添加”一些实际上二叉树中并不存在的“虚结点”（可以认为这些结点的数据信息为空），使其在形式上成为一棵“完全二叉树”，然后按照完全二叉树的顺序存储结构的构造方法将所有结点的数据信息依次存放于数组BT[0..2^h -2]中。

【练习题】当一棵有n个结点的二叉树按层次从上到下，同层次从左到右将数据存放在一维数组 A[1...n]中时，数组中第i个结点的左孩子为【 】。

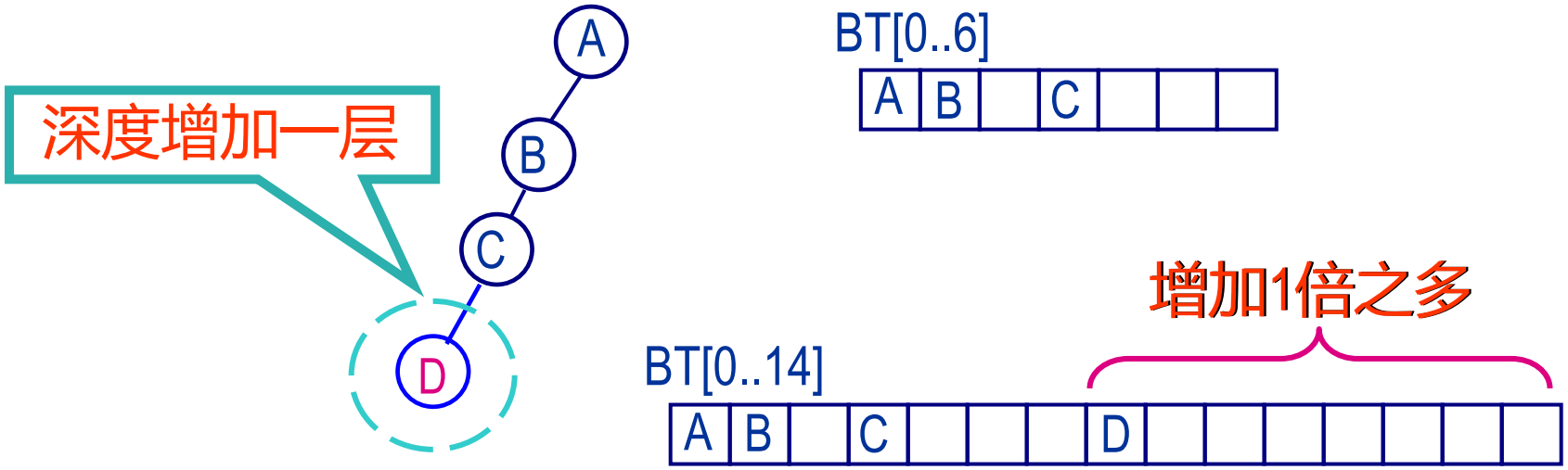
A．A[2i](2i<=n)　　B．A[2i+1](2i+1<= n)

C．A[i/2]　　D．无法确定

1.顺序存储结构比较适合满二叉树，或者接近于满二叉树的完全二叉树，对于一些称为“退化二叉树”的二叉树，顺序存储结构的空间开销浪费的缺点表现比较突出。

2.顺序存储结构便于结点的检索（由双亲查子、由子查双亲）。

3.顺序存储结构需事先分配存储空间，对于动态数据容易溢出。



4.2 二叉树的链式存储结构

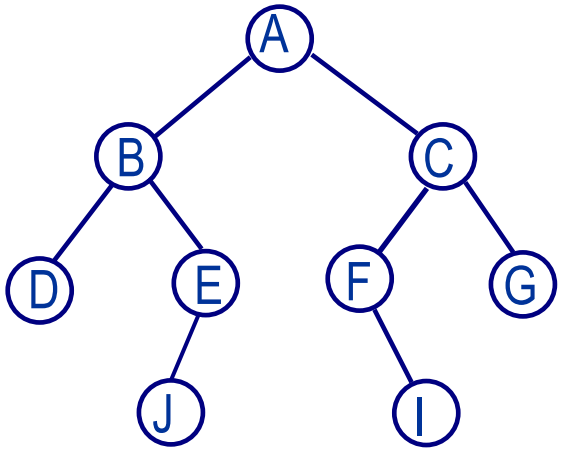
1. 二叉链表

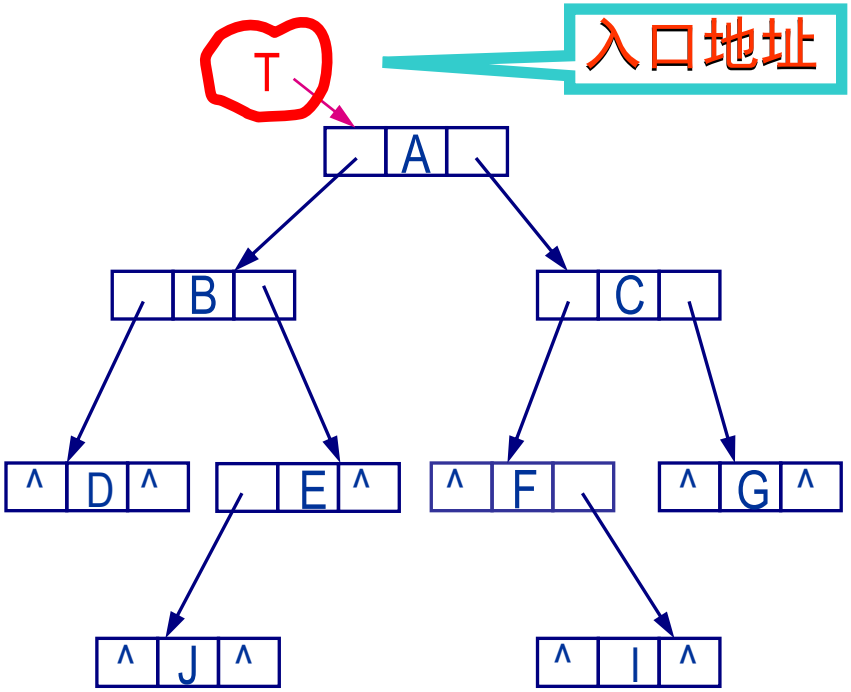
链结点的构造为

lchild	data	rchild

其中, data 为数据域；

lchild 与rchild 分别为指向左、右子树的指针域。



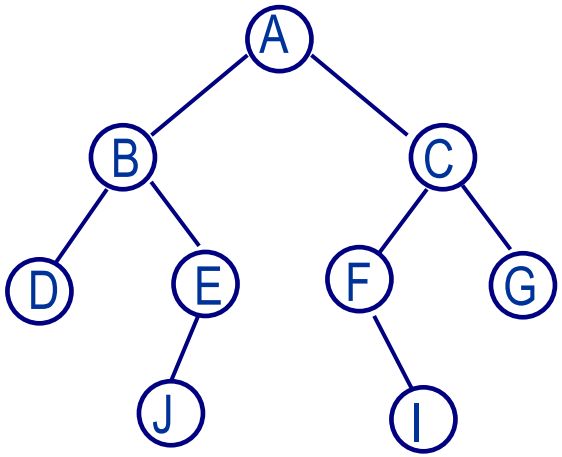


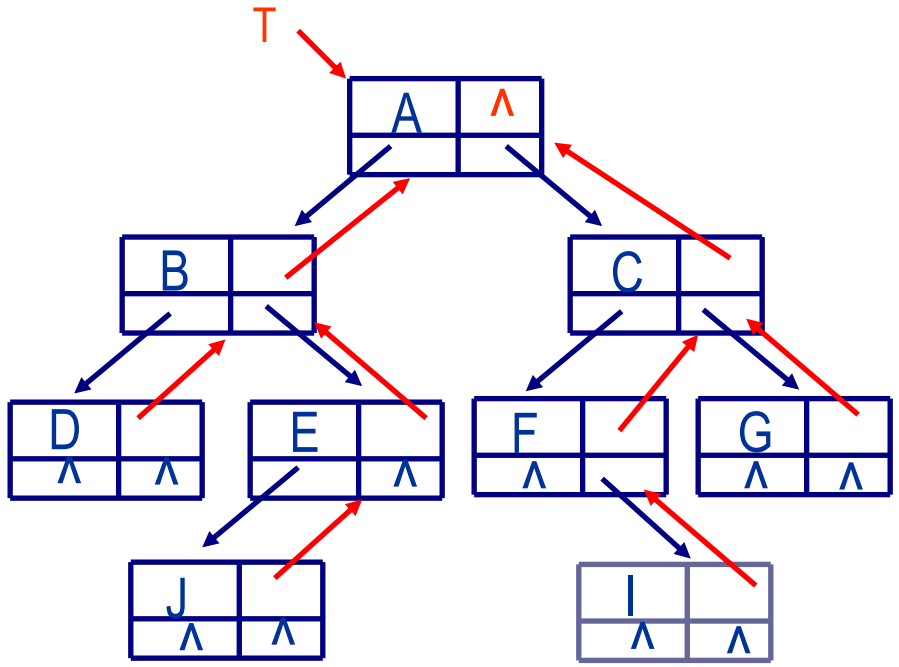
2. 三叉链表

链结点的构造为

data	parent
lchild	rchild

其中, data 为数据域， parent为指向双亲结点的指针；
lchild 与rchild 分别为指向左、右孩子结点的指针。





二叉链表的结点类型定义

```
struct  node{
        Datatype   data;
        struct  node   *lchild,  *rchild;
};
typedef struct node  BTNode;
typedef struct node  *BTNodeptr;
```

```
BTNodeptr  T, p, q;
```

二叉树应用：表达式（expression tree）树

表达式树是一种特殊类型的树，其叶结点是操作数（operand），而其它结点为操作符（operator）：

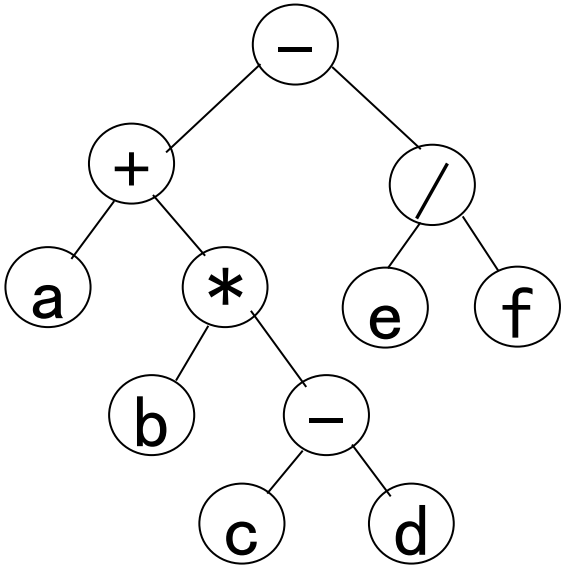
（1）由于操作符一般都是双目的，该树通常是一棵二叉树；

（2）对于单目操作符（如++），其只有一个子结点。

对于表达式：

a + b * (c – d) – e / f

其对应的表达树如右图所示。



表达式树的最大好处是表达式没有括号，计算时也不用考虑运算符优先级。主要应用：(1)编译器用来处理程序中的表达式

二叉树应用：表达式（expression tree）树

【编程题3】从标准输入中读入一个整数算术运算表达式，如 24 / (1 + 2 + 36 / 6 / 2 - 2) * (12 / 2 / 2)= ，计算表达式结果，并输出。要求：

1、表达式运算符只有+、-、*、/，表达式末尾的=字符表示表达式输入结束，表达式中可能会出现空格；

2、表达式中会出现圆括号，括号可能嵌套，不会出现错误的表达式；

3、出现除号/时，以整数相除进行运算，结果仍为整数，例如：5/3结果应为1。

4、要求采用表达式树来实现表达式计算。

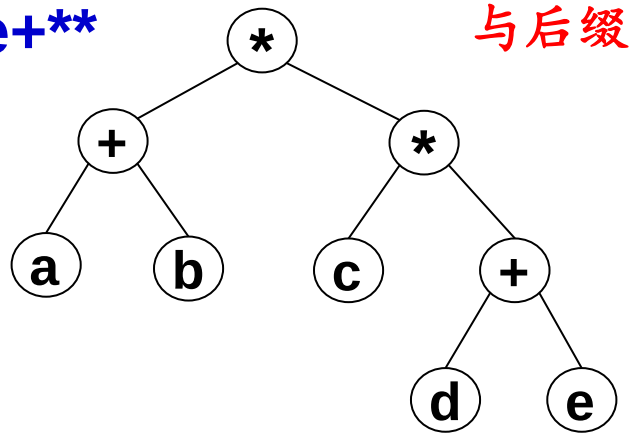
【编程题3】一种构造表达式树的思路（分两步处理）

1. **由中缀表达式转成后缀表达式**（参见《堆栈》章节中的由中缀表达式生成后缀表达式）
2. **由后缀表达式生成表达式树**（参见后缀表达式计算）：

从左至右从后缀表达式中读入一个符号：

- 若是操作数，则建立一个单节点树并将指向它的指针压入栈中；
- 若是运算符，就从栈中弹出指向两棵树T_1和T_2的指针（T_1先弹出）并形成一棵新树，树根为该运算符，它的左、右子树分别指向T_2和T_1，然后将新树的指针压入栈中。

ab+cde+**



与后缀表达式计算的处理逻辑类似

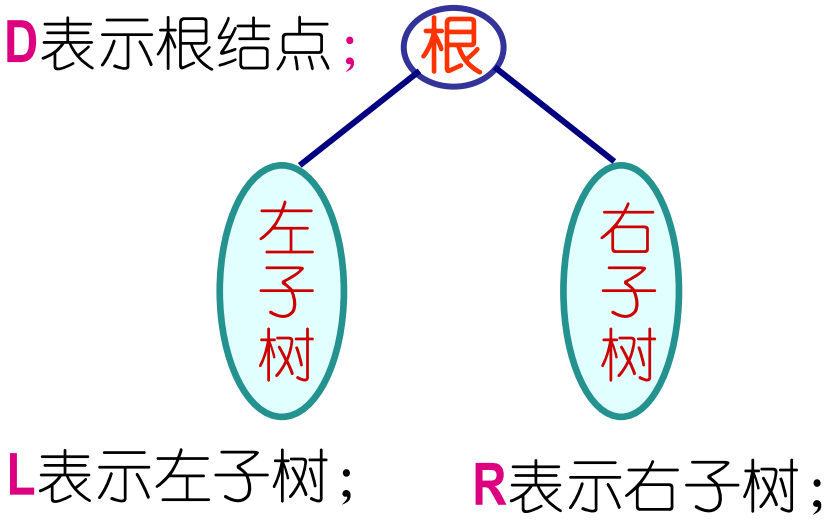
构造表达式树的过程中可以同步进行计算，并将计算结果保存在子树的根结点。

5 二叉树的遍历

5.1 基本概念

按照一定的顺序(原则)对二叉树中每一个结点都访问一次(仅访问一次), 得到一个由该二叉树的所有结点组成的序列,这一过程称为二叉树的遍历

- 输出所有结点的信息;
- 找出或统计满足条件的结点;
- 求二叉树的深度;
- 求指定结点所在的层次;
  求指定结点的所有祖先结点
  ……



广度优先遍历方法:
按层次遍历

深度优先遍历方法:
1.前序遍历 DLR
2.中序遍历 LDR
3.后序遍历 LRD

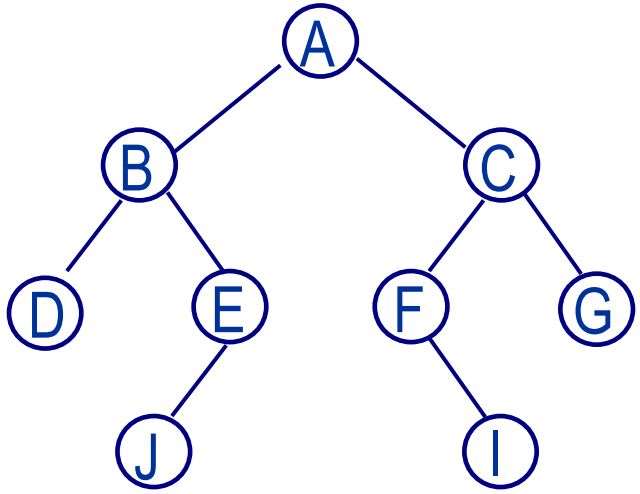
5.2 按层次遍历

原则

若被遍历的二叉树非空，则按照层次从上到下，每一层从左到右依次访问结点。

按层次遍历序列:

A B C D E F G J I



层次遍历为一种广度优先算法（BFS）。

```
#define NodeNum  100
void  layerorder(BTNodeptr t){
    BTNodeptr queue[NodeNum], p;
    int front, rear;
    if(t!=NULL){
        queue[0]=t;
        front=-1;
        rear=0;
        while(front<rear){    /* 若队列不空 */
            p=queue[++front];
            VISIT(p);         /* 访问p指结点 */
            if(p->lchild!=NULL)   /* 若左孩子非空
                queue[++rear]=p->lchild;
            if(p->rchild!=NULL)   /* 若右孩子非空
                queue[++rear]=p->rchild;
        }
    }
}
```

顺序队列

```
//封装了队列操作
void  layerorder(BTNodeptr t){
    BTNodeptr  p;
    if(t!=NULL){
        enQueue(t);
        while(!isEmpty()){
            p=deQueue();
            VISIT(p);
            if(p->lchild!=NULL)
                enQueue(p->lchild);
            if(p->rchild!=NULL)
                enQueue(p->rchild);
        }
    }
}
```

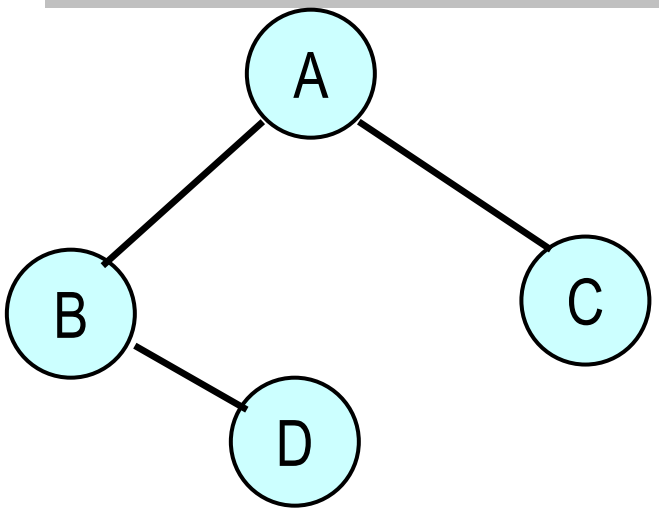
对二叉树进行层次遍历的时间复杂度均为O(n)

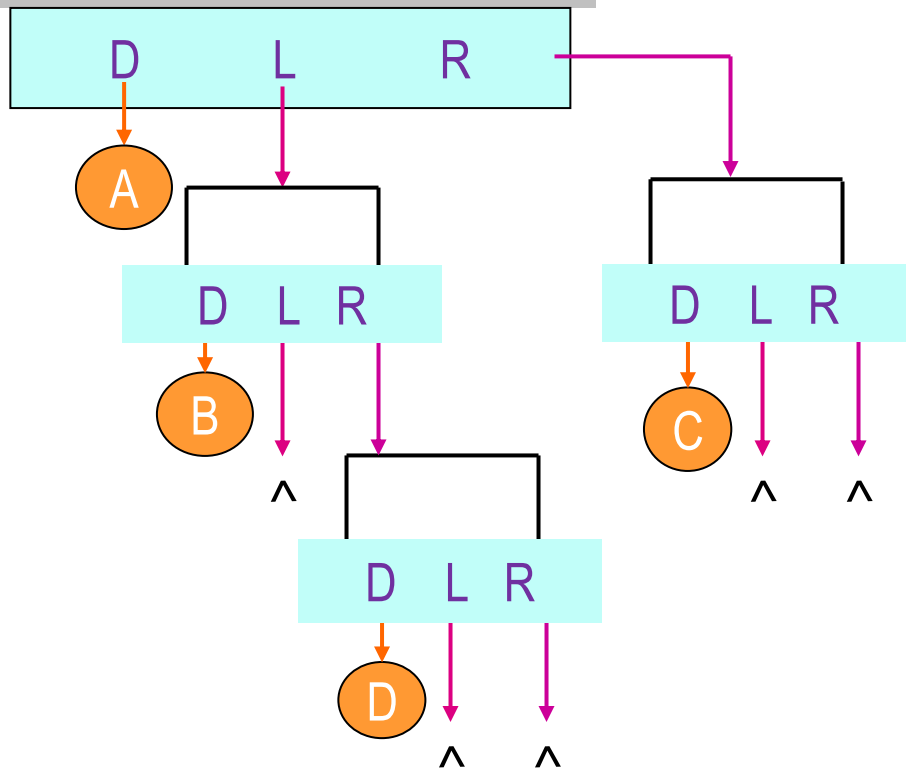
5.3 前序遍历

原则若被遍历的二叉树非空，则

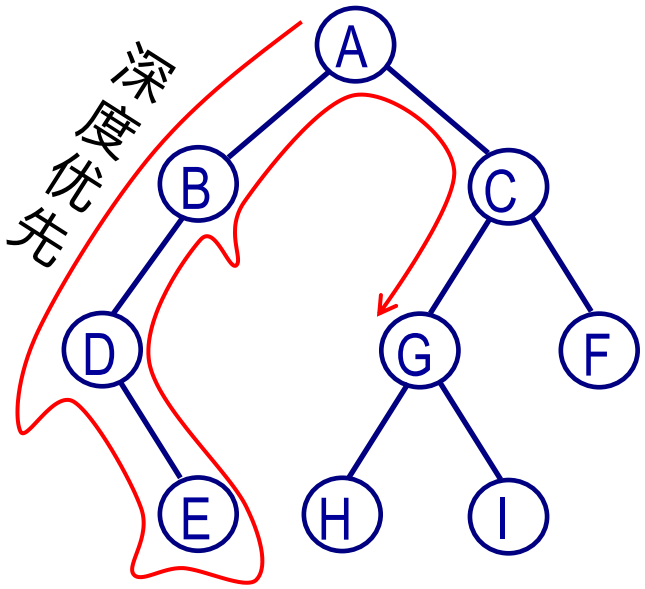
1. 访问根结点;
2. **以前序遍历原则**遍历根结点的左子树;
3. **以前序遍历原则**遍历根结点的右子树。

递归





前序遍历序列：A B D C



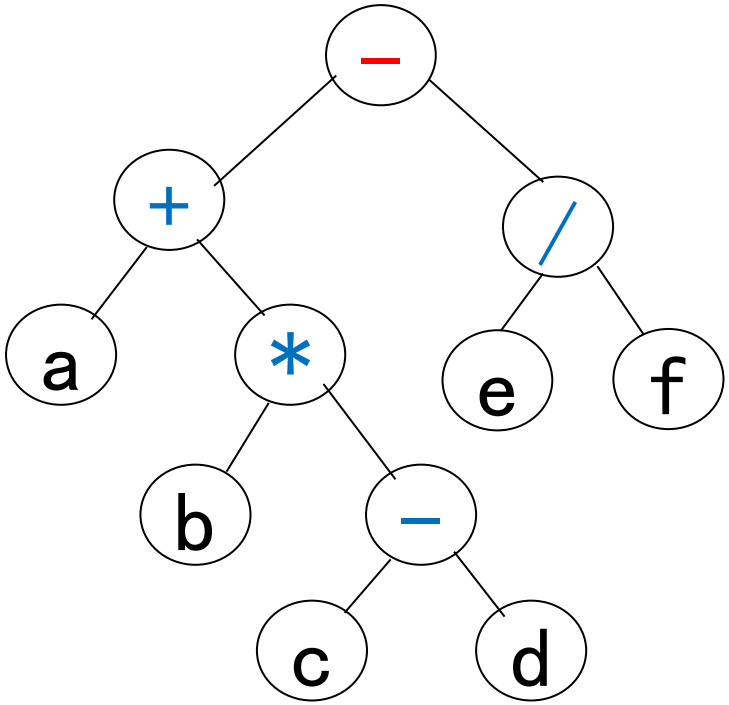
递归算法

```
void  preorder(BTNodeptr t){
    if(t!=NULL){
        VISIT(t);    /* 访问t指向结点 */
        preorder(t->lchild);
        preorder(t->rchild);
    }
}
```

前序序列: A B D E C G H I F

练习 表达式树的前序遍历

(a+b*(c-d)) – e/f



前序遍历：- + a * b – c d / e f

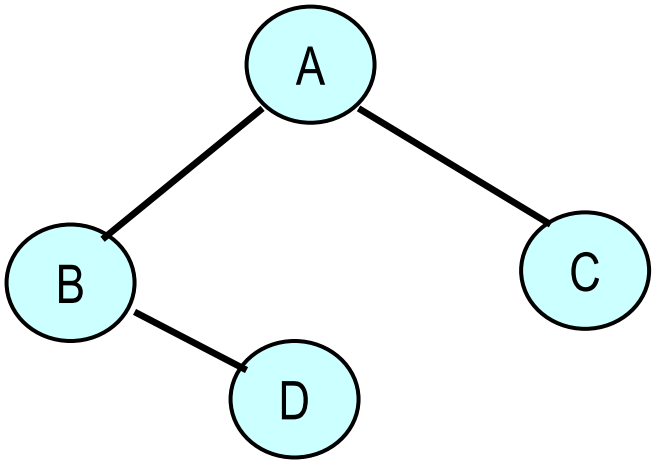
前缀表达式

5.4 中序遍历

递归

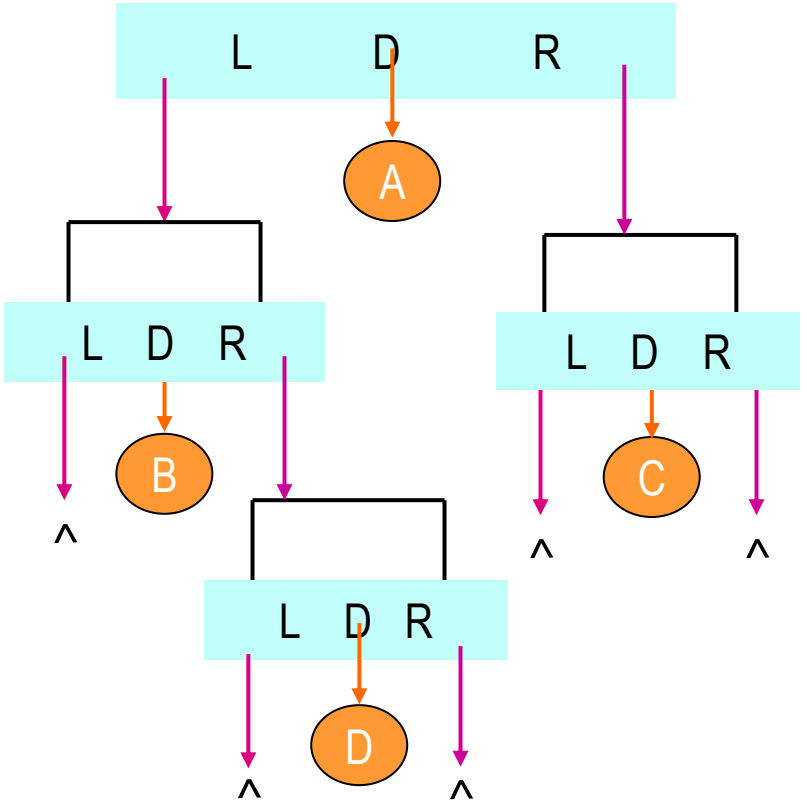
原则 若被遍历的二叉树非空，则

1. 以中序遍历原则遍历根结点的左子树；
2. 访问根结点；
3. 以中序遍历原则遍历根结点的右子树。

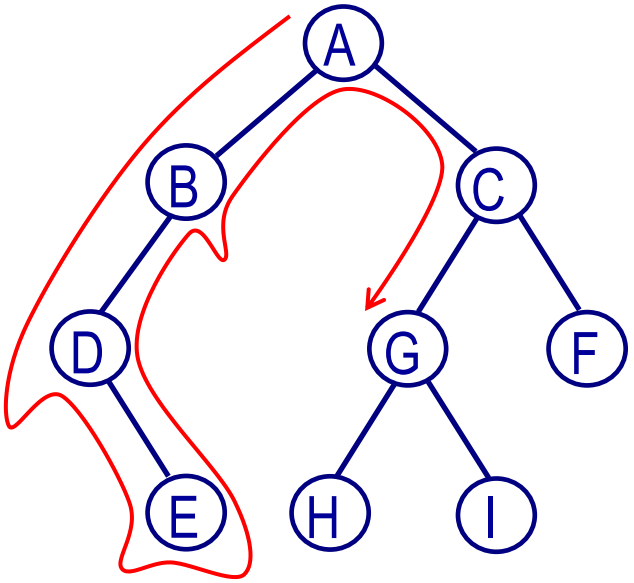


中序遍历序列：

B D A C



递归算法



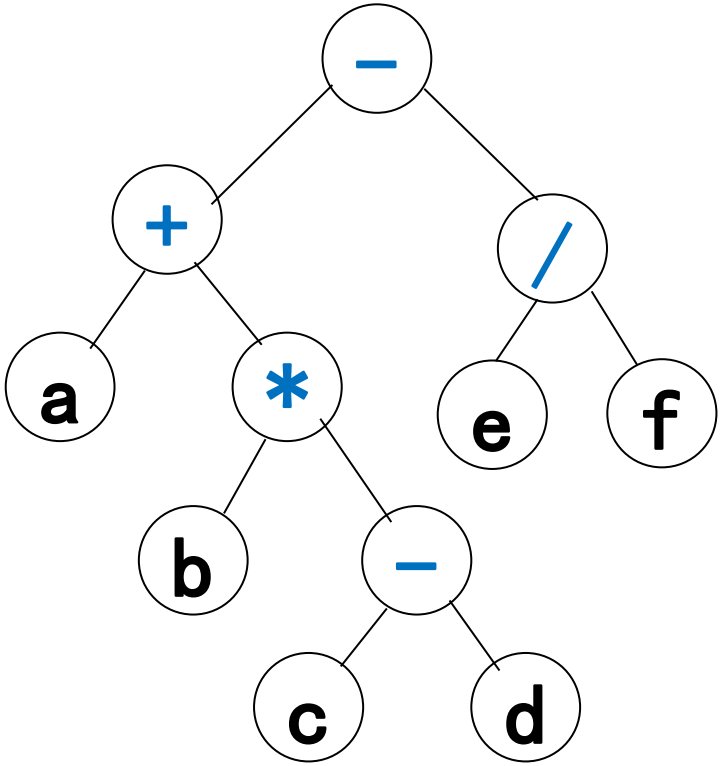

```
void  inorder(BTNodeptr t){
      if(t!=NULL){
            inorder(t->lchild);
            VISIT(t);  /* 访问T指结点 */
            inorder(t->rchild);
      }
}
```

前序序列: A B D E C G H I F

中序序列: D E B A H G I C F

练习 表达式树的中序遍历

(a+b*(c-d)) – e/f



前序遍历：- + a * b – c d / e f

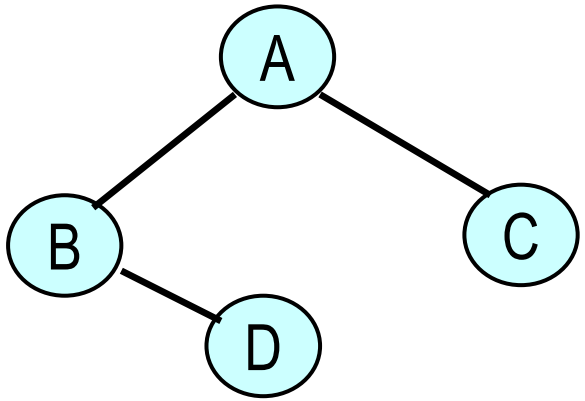
中序遍历：a + b * c – d – e / f

5.5 后序遍历

递归

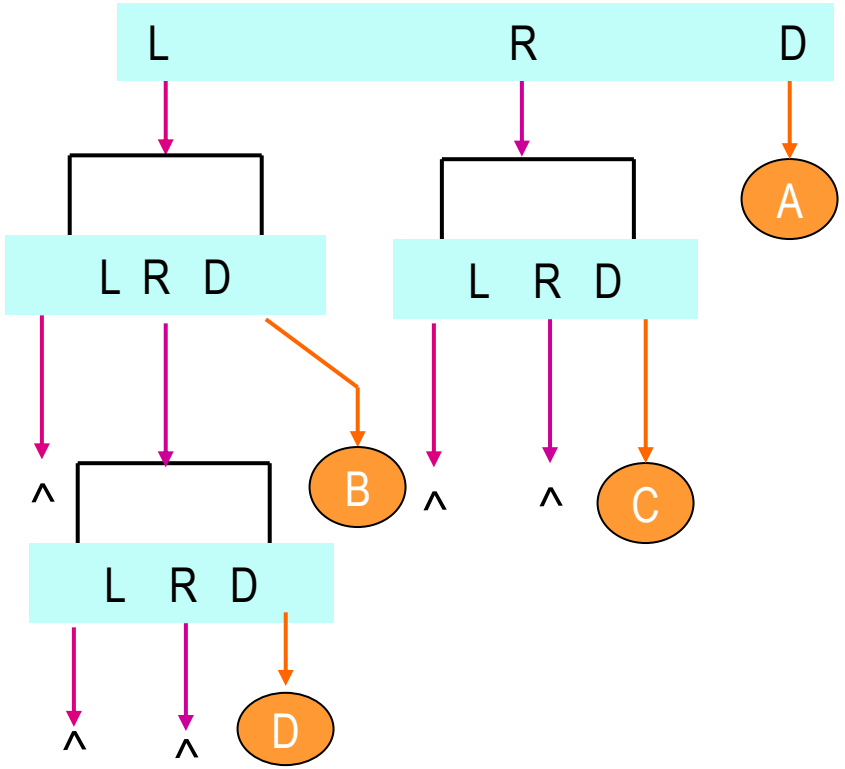
若被遍历的二叉树非空，则

1. **以后序遍历原则**遍历根结点的左子树；
2. **以后序遍历原则**遍历根结点的右子树；
3. 访问根结点。

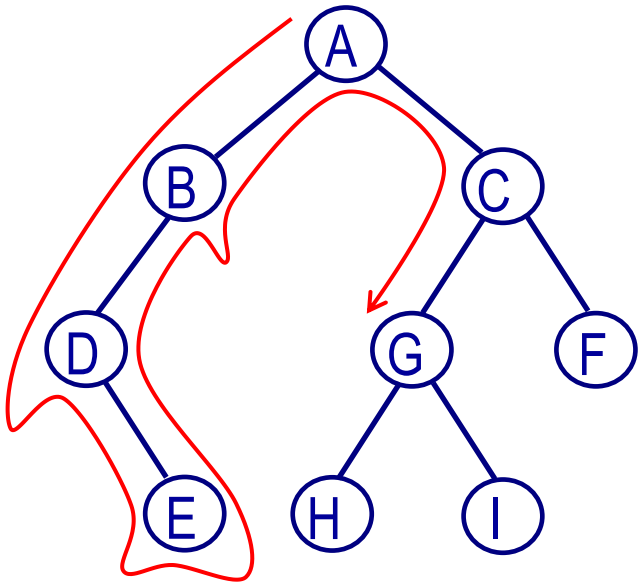


后序遍历序列：

D B C A



递归算法




```
void  postorder(BTNodeptr t){
    if(t!=NULL){
        postorder(t->lchild);
        postorder(t->rchild);
        VISIT(t);  /* 访问t指结点 */
    }
}
```

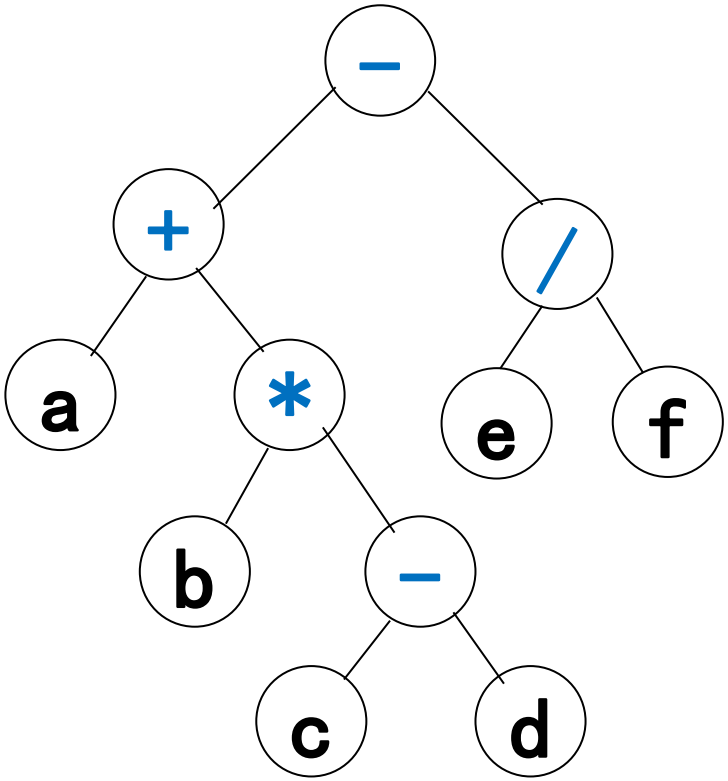
前序序列: A B D E C G H I F

中序序列: D E B A H G I C F

后序序列: E D B H I G F C A

练习 表达式树的后序遍历

(a+b*(c-d)) – e/f



前序遍历：-+a*b–cd/ef

中序遍历：a+b*c–d–e/f

后序遍历：abcd-*+ef/-

后缀表达式

前序遍历

若被遍历的二叉树非空，则
1.访问根结点;
2.以前序遍历原则遍历根结点的左子树;
3.以前序遍历原则遍历根结点的右子树。

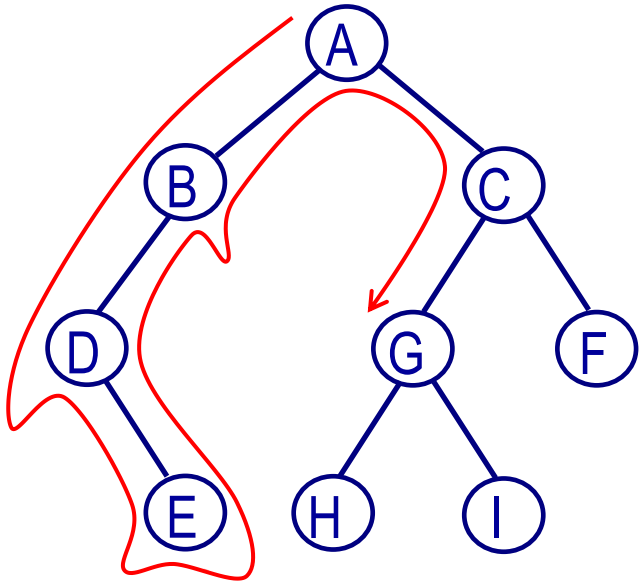
中序遍历

若被遍历的二叉树非空，则
1.以中序遍历原则遍历根结点的左子树;
2.访问根结点;
3.以中序遍历原则遍历根结点的右子树。

后序遍历

若被遍历的二叉树非空，则
1.以后序遍历原则遍历根结点的右子树
2.以后序遍历原则遍历根结点的左子树;
3.访问根结点。

前序、中序及后序遍历实质为深度优先算法（DFS）



前序遍历

```
void  preorder(BTNodeptr t){
    if(t!=NULL){
        VISIT(t);      /* 访问t指向结点 */
        preorder(t->lchild);
        preorder(t->rchild);
    }
}
```

中序遍历

```
void  inorder(BTNodeptr t){
    if(t!=NULL){
        inorder(t->lchild);
        VISIT(T);      /* 访问T指结点 */
        inorder(t->rchild);
    }
}
```

后序遍历

```
void  postorder(BTNodeptr t){
    if(t!=NULL){
        postorder(t->lchild);
        postorder(t->rchild);
        VISIT(T);      /* 访问T指结点 */
    }
}
```

附：递归问题的非递归算法的设计*

1. 递归算法的优点

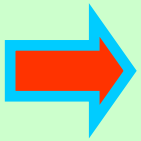
(1) 问题的数学模型或算法设计方法本身就是递归的，采用递归算法来描述它们非常自然；

(2) 算法的描述直观，结构清晰、简洁；算法的正确性证明比非递归算法容易。

例如

斐波那契数列的计算

$$\begin{cases} F_0=F_1=1; \\ F_i=F_{i-1}+F_{i-2}, \quad i>1 \end{cases}$$



```
int F(int n){
    if(n<=1)
        return 1;
    else
        return F(n−1)+F(n−2);
}
```

附：递归问题的非递归算法的设计*

1. 递归算法的优点

(1) 问题的数学模型或算法设计方法本身就是递归的，采用递归算法来描述它们非常自然；

(2) 算法的描述直观，结构清晰、简洁；算法的正确性证明比非递归算法容易。

结论：谨慎使用递归，因为它的简洁可能会掩盖它的低效率。

2. 递归算法的不足

(1) 算法的执行时间与空间开销往往比非递归算法要大，当问题规模较大时尤为明显；

(2) 对算法进行优化比较困难；

(3) 分析和跟踪算法的执行过程比较麻烦；

(4) 算法语言不具有递归功能时，算法无法描述。

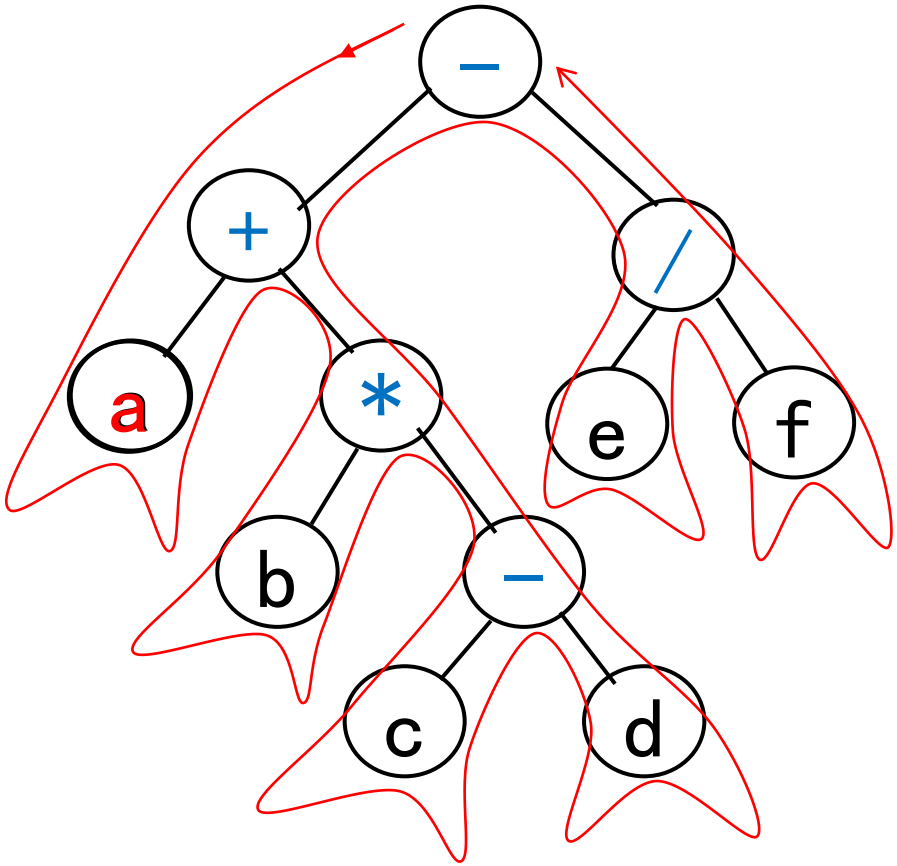
5.6 深度优先遍历的非递归算法

非递归算法如何设计

```
void  inorder( BTNodeptr t ){
    if(t!=NULL){
        inorder(t->lchild);
        VISIT(t);
        inorder(t->rchild);
    }
}
```

以中序遍历为例

5.6 深度优先遍历的非递归算法



（中序）遍历过程分析：

1. 沿左子树（左孩子）前进，直至左子树为空；
2. 从左子树回退到父结点，访问该结点

表达式：**(a+b*(c-d)) – e/f**

STACK[0..M−1] -- 保存遍历过程中结点的地址；
top -- 栈顶指针，初始为**-1**；
p -- 为遍历过程中使用的指针变量，初始时指向根结点。

中序遍历非递归算法步骤：

p=p->lchild;

1. 若p 指向的结点非空，则将p指的结点的地址进栈，然后，将p 指向左子树的根；

2. 若p指向的结点为空，则从堆栈中退出栈顶元素送p，访问该结点，然后，将p 指向右子树的根；

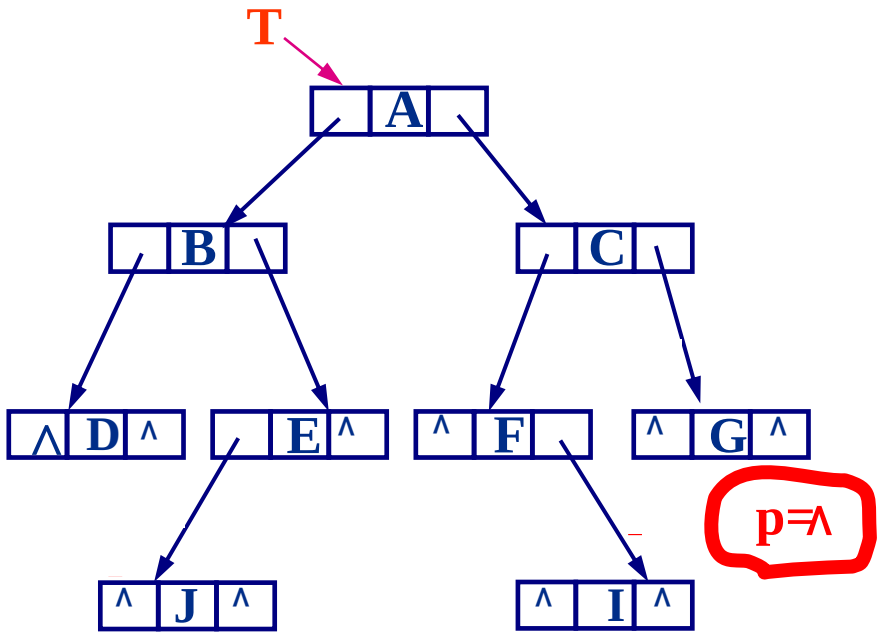
重复上述过程，直到p为空，并且堆栈也为空。

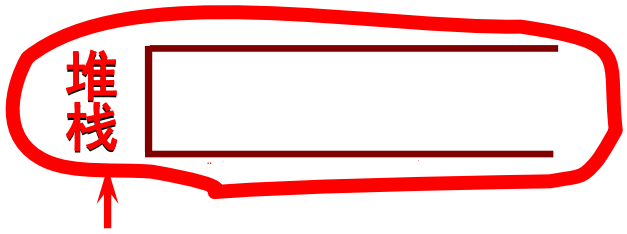
p=p->rchild;

中序遍历的非递归算法

中序序列：
D, B, J, E, A, F, I, C, G

```
void  inOrder(BTNodeptr t){
    BTNodeptr stack[M], p=t;
    int top=−1;
    if(t!=NULL)
       do {
            for(; p!=NULL; p=p−>lchild)
                stack[++top]=p;
            p=stack[top– –]; //回退
            VISIT(p);
            p=p−>rchild;
     }while(!(p==NULL && top==−1));
}
```



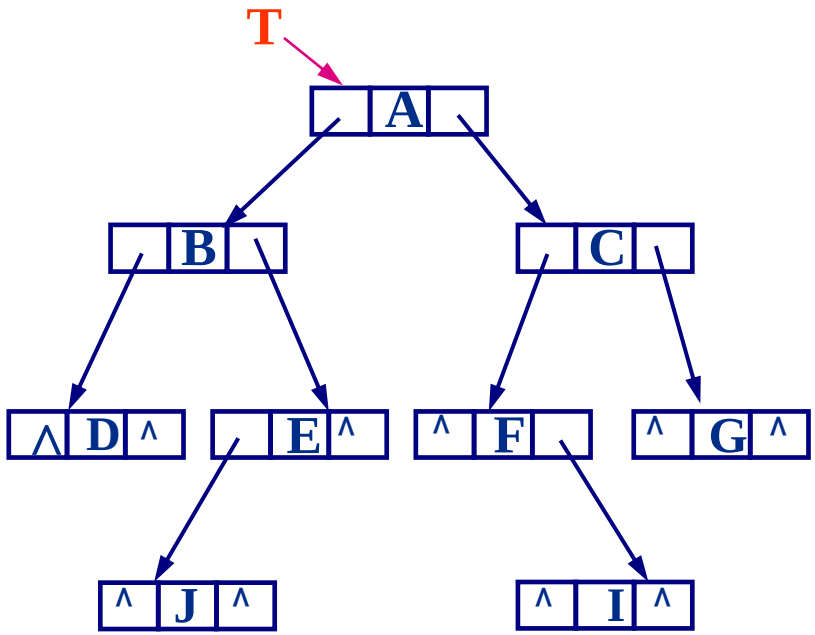


D B J E A F I C G

前序遍历的非递归算法

```
void  preOrder(BTNodeptr t){
    BTNodeptr stack[M], p=t;
    int top=−1;
    if(t!=NULL)
        do {
            for(; p!=NULL; p=p−>lchild)
                stack[++top]=p;
            p=stack[top– –]; //回退
            VISIT(p);
            p=p−>rchild;
        }while(!(p==NULL && top==−1));
}
```

前序遍历

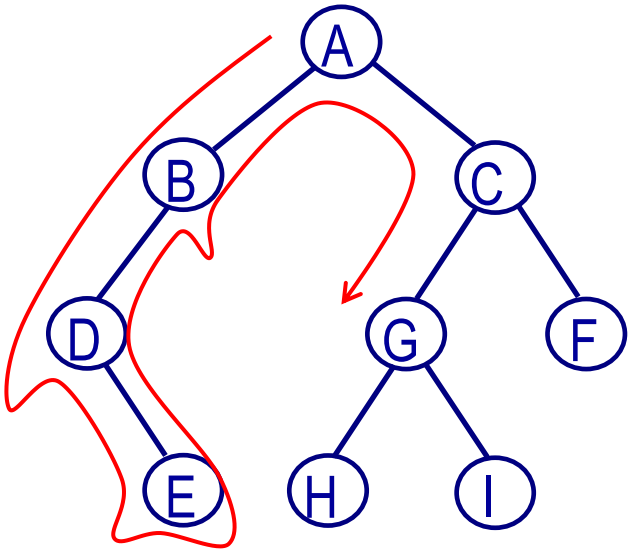


后序遍历的非递归算法略有不同，结点需进出栈2次，第2次出栈时访问该结点。

对于一个普通二叉树，如何获得一个节点的祖先序列呢？

原则上可结合不同的深度遍历法（递归或非递归）

- 若采用非递归遍历，则后序遍历的栈里保存了祖先序列
- 若采用递归遍历，则可以额外构造栈记录祖先序列



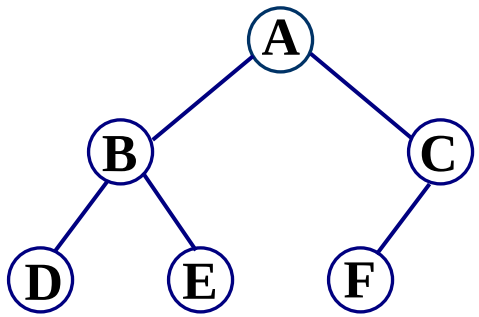
基于前序遍历（递归）

```
void  preorder(BTNodeptr t, Datatype item){
    if(t!=NULL){
        push(t);
        if(item == t->data)
            {弹出栈中所有元素；结束}
        preorder(t->lchild);
        preorder(t->rchild);
        pop();
    }
}
```

5.7 顺序存储下的遍历

已知具有n个结点的完全二叉树采用顺序存储结构，结点的数据信息依次存放于一维数组BT[0..n-1]中，写出中序遍历二叉树的非递归算法。

完全二叉树



中序序列: D B E A F C

BT[0..6]

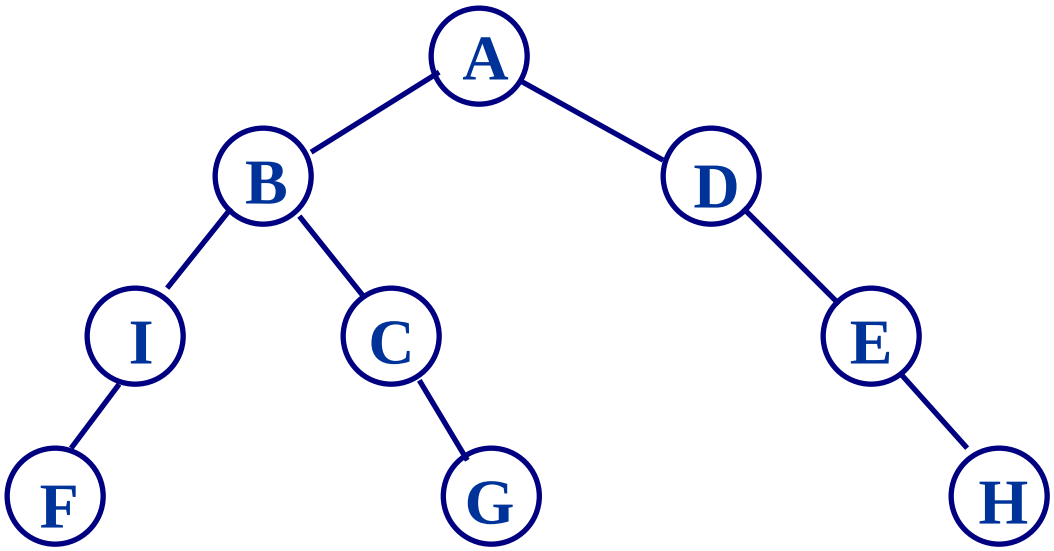
0	1	2	3	4	5	6
A	B	C	D	E	F	

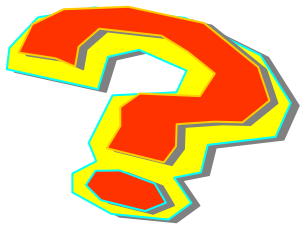
例4

一般二叉树

BT[0..14]

A	B	D	I	C		E	F			G				H





前序序列：A B I F C G D E H

中序序列：F I B C G A D E H

后序序列：F I G C B H E D A

对于一般二叉树，只须在二叉树中“添加”一些实际上二叉树中并不存在的“虚结点”（可以认为这些结点的数据信息为空），使其在形式上成为一棵“完全二叉树”，然后按照完全二叉树的顺序存储结构的构造方法将所有结点的数据信息依次存放于数组BT[0..2^h -2]中。

【易错题】当一棵有n个结点的二叉树按层次从上到下，同层次从左到右将数据存放在一维数组 A[1...n]中时，数组中第i个结点的左孩子为【 】。

A．A[2i](2i<=n)　　B．A[2i+1](2i+1<= n)

C．A[i/2]　　D．无法确定

二叉链表=>顺序表

STACK[0..M-1] -- 保存遍历过程中结点的位置(下标);
top -- 栈顶指针, 初始为**-1**;
i -- 为遍历过程中使用的位置变量，初始指向根结点。

i=0, BT[0]

1. 若 i 指向的结点非空，则将i 进栈，然后，将i指向左子树的根;

i=2*i+1;

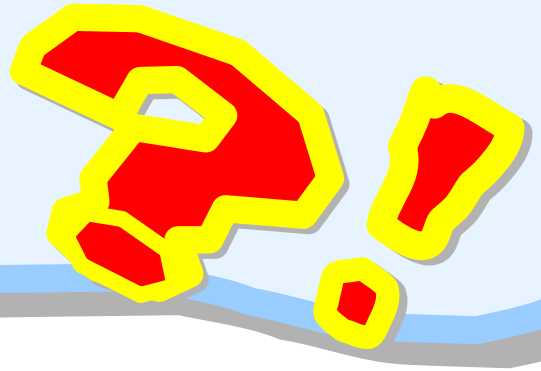
2. 若 i 指向的结点为空，则从堆栈中退出栈顶元素送 i，访问该结点, 然后, 将i指向右子树的根;

i=2*i+2;

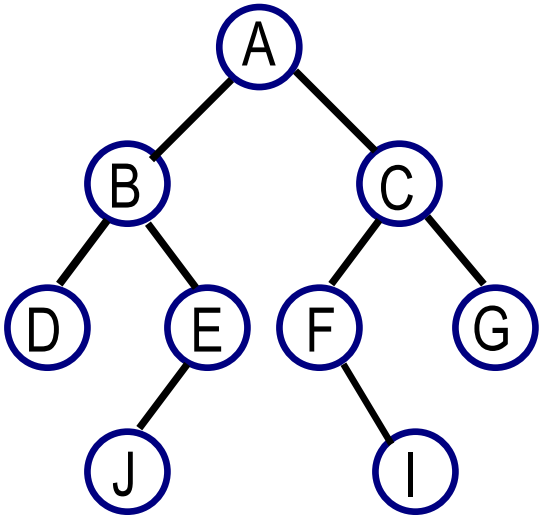
重复上述过程, 直到i指结点不存在, 并且栈空。

```
#define MaxSize  100
void inorder(Datatype bt[],int n){
    int stack[MaxSize],i,top=-1;
    i=0;
    if(n>=0){
       do{
          while(i<n){
             stack[++top]=i;      /* bt[i]的位置i进栈*/
             i=i*2+1;             /* 找到i的左孩子的位置 */
          }
          i=STACK[top--];         /* 退栈*/
          VISIT(bt[i]);           /* 访问结点bt[i] */
          i=i*2+2;                /* 找到i的右孩子的位置*/
       }while(!(i>n-1 && top==-1));
    }
}
```

若已知二叉树的前序序列与中序序列，如何求二叉树的后序序列



5.8 由遍历序列恢复二叉树



前序序列:

A, B, D, E, J, C, F, I, G

中序序列:

D, B, J, E, A, F, I, C, G

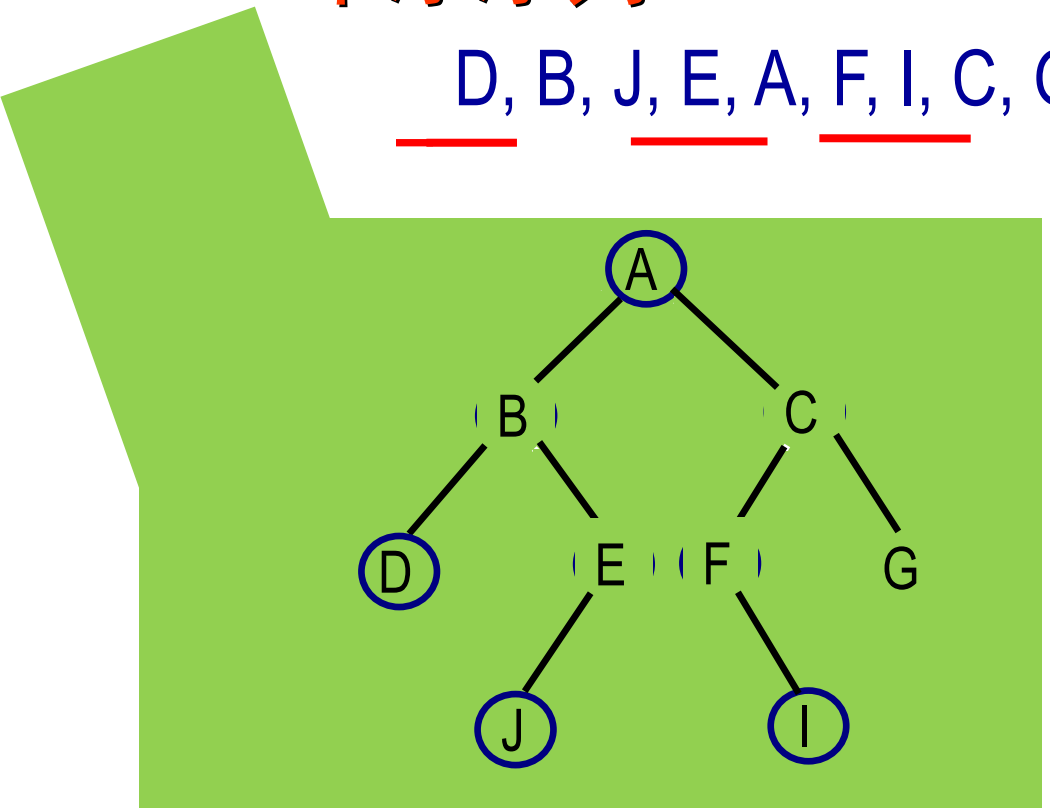
后序序列？

前序序列:

A, B, D, E, J, C, F, I, G

中序序列:

D, B, J, E, A, F, I, C, G



后序序列:

D, J, E, B, I, F, G, C, A

已知前序序列和中序序列，恢复二叉树：

在前序序列中确定根；
到中序序列中分左右。

已知中序序列和后序序列，恢复二叉树：

在后序序列中确定根；
到中序序列中分左右。

思考

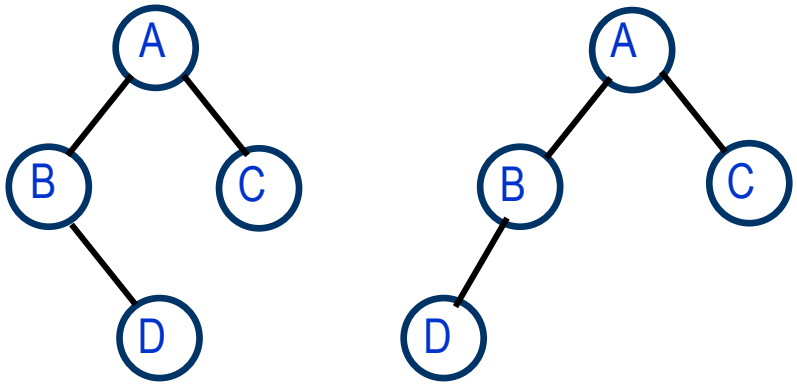
能否利用遍历序列恢复二叉树

利用前序序列和中序序列恢复二叉树 ✓

利用中序序列和后序序列恢复二叉树 ✓

利用前序序列和后序序列恢复二叉树 ✗

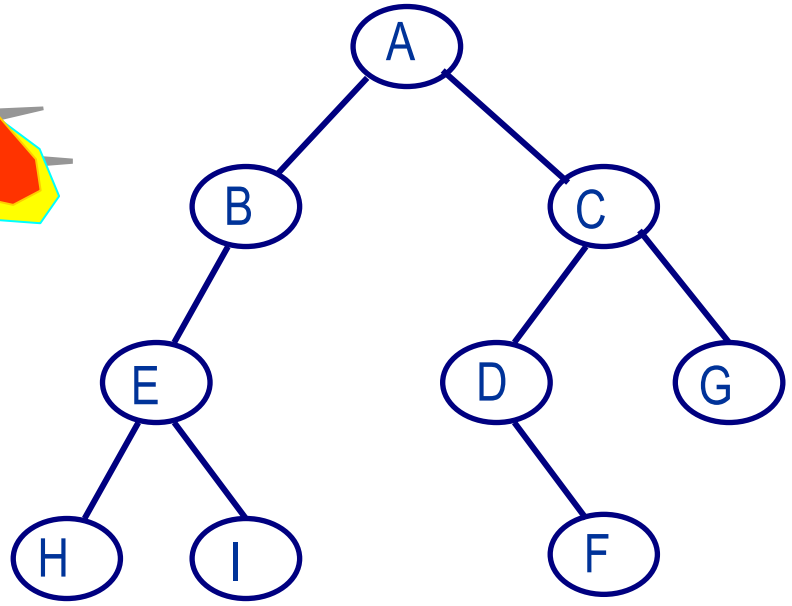
例



前序序列：A,B,D,C

后序序列：D,B,C,A

练习



前序序列:

A B E H I C D F G

中序序列 :

H E I B A D F C G

后序序列:

H I E B F D G C A

5.8 二叉树的典型操作（遍历的应用）

- 检查二叉树是否为空
- 在二叉树中查找给定元素
- 在二叉树中插入一个元素
- 从二叉树中删除一个元素
- 遍历二叉树
- 获得二叉树的节点数目
- 获得二叉树叶节点的数目
- **获得二叉树的高度**
- **拷贝二叉树**
- **删除二叉树**

基于二叉树的遍历
可以是深度或广度遍历
可以是递归或非递归

注：1. 由于在二叉树中插入及删除元素与树的构造方式有关，将结合特定树（二叉查找树）来介绍。
2. 查找元素与遍历树算法相似。

二叉树的高度（递归实现）

```
int max(x,y)
{ if((x >y)  return x; else return y; }

int  heightTree(BTNodeptr  p) {
    if(p == NULL)
        return 0;
    else
        return 1 + max(heightTree(p->lchild),
                       heightTree(p->rcihld));
 }
```

哪种遍历？

二叉树的删除（递归实现）

```
void  destoryTree(BTNodeptr  p){
    if(p != NULL){
        destoryTree(p->lchild);
        destoryTree(p->rchild);
        free(p);
        p = NULL;
    }
}
```

哪种遍历？

二叉树的拷贝（递归实现）

```
BTNodeptr  copyTree(BTNodeptr  src){
    BTNodeptr obj;
    if(src == NULL)
        obj = NULL;
    else {
        obj = (BTNodeptr) malloc(sizeof(BTNode));
        obj->data = src->data;
        obj->lchild = copyTree(src->lchild);
        obj->rchild = copyTree(src->rchild);
    }
    return obj;
}
```

哪种遍历？