数据结构与算法

Data Structures and Algorithms

第三部分 树

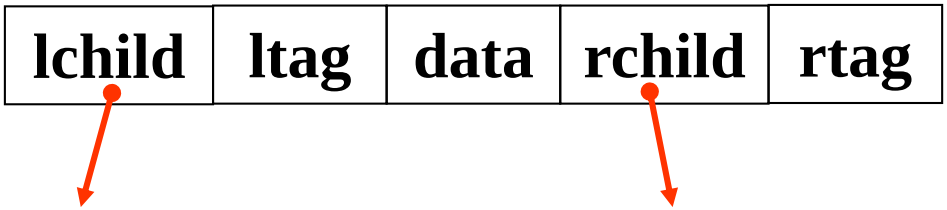
回顾：树--2

1. 二叉树的四种遍历方式

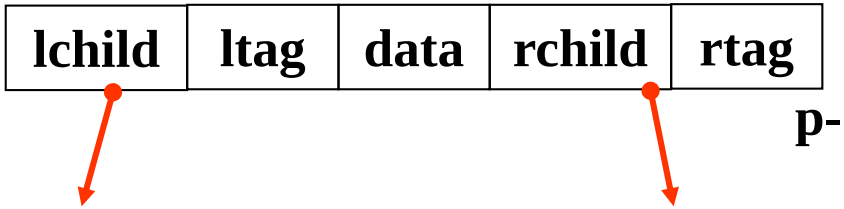
先根顺序遍历；中根顺序遍历；后根顺序遍历；层次遍历
递归过程 与 非递归过程

2. 二叉树的表示

（1）顺序存储（完全二叉树和一般二叉树）
（2）链式存储
-- 左右链表示方法
-- 三叉链表存储表示方法
-- 线索二叉树
中序线索二叉树（中序后继和中序前驱）

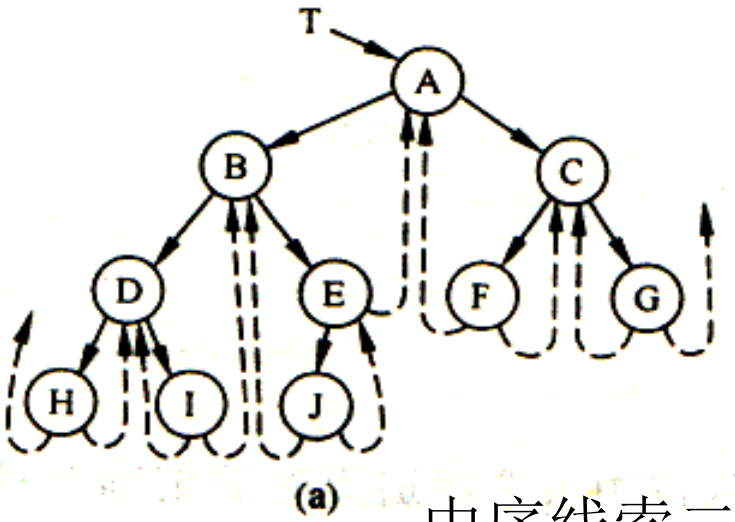


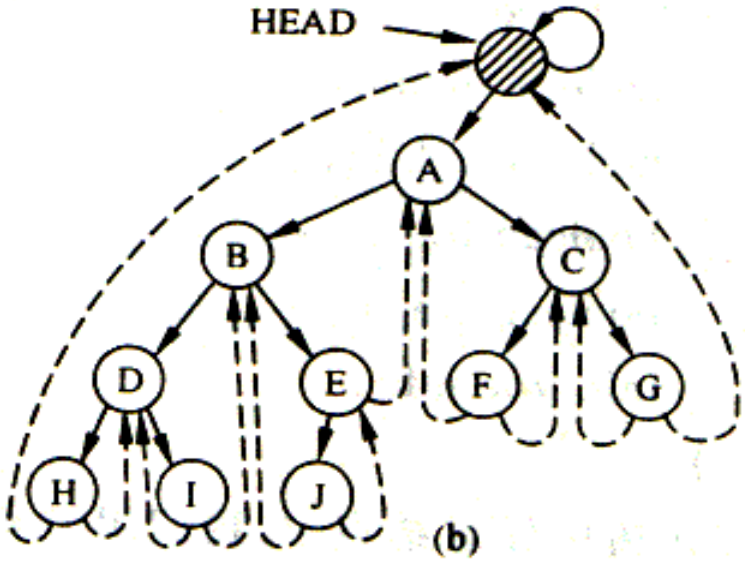
结点类型 LNode



$$p\text{->ltag} = \begin{cases} 1 & p\text{->lchild 指向左孩子} \\ 0 & p\text{->lchild 指向（中序）前驱} \end{cases}$$

$$p\text{->rtag} = \begin{cases} 1 & p\text{->rchild 指向右孩子} \\ 0 & p\text{->rchild 指向（中序）后继} \end{cases}$$





中序线索二叉树

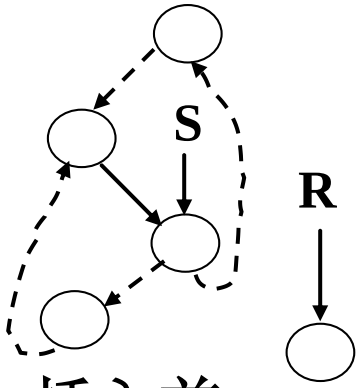
在二叉树中一般不讨论结点的插入与删除操作，原因是其插入与删除的操作与线性表相同，所不同的是需要详细说明操作的具体要求。

而在线索树中结点的插入与删除操作则不同，因为要同时考虑修正线索的操作。

例：将结点 R 插入作为结点 S 的右孩子结点。

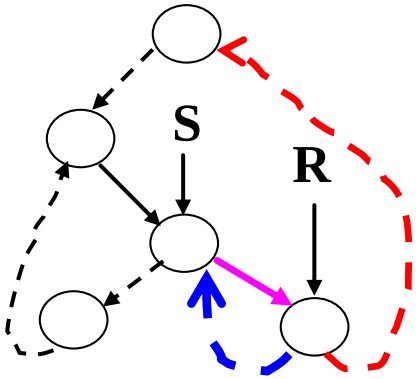
（1）若S的右子树为空，插入比较简单；

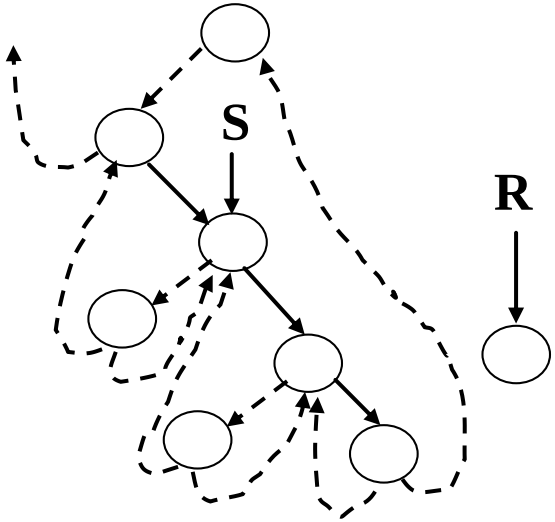
（2）若S的右子树非空，则 R 插入后，原来 S 的右子树作为 R 的右子树。



插入前
（S的右子树为空）

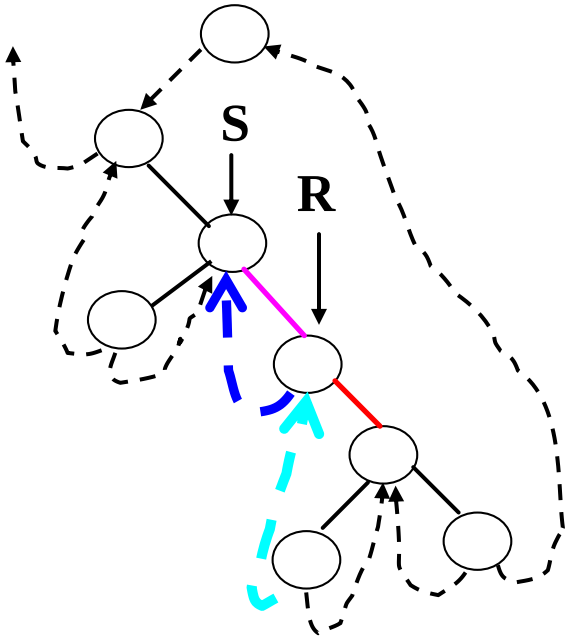
直接插入





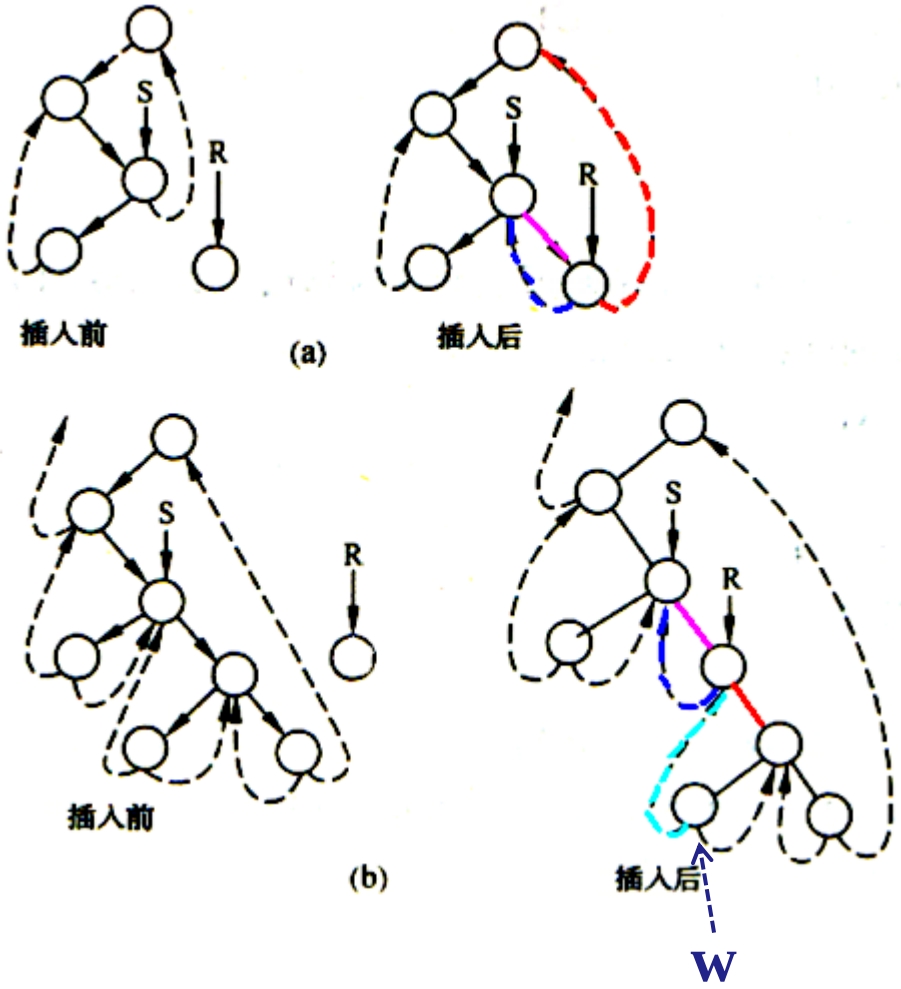
插入前
（右子树非空）

原S右子树
作为R的右子树



操作：

```
Void   RINSERT ( THTREE   S , THTREE   R )
  THTREE   w ;
  R->rchild = S->rchild ;
  R->rtag = S->rtag ;
  R->lchild = S ;
  R->ltag = 0 ;
  S->rchild = R ;
  S->rtag = 1 ;
  if ( R->rtag == 1 )
    {  w = InNext( R ) ;
       w->lchild = R ;  }
```



中序线索化：

1、将二叉树的空指针改为指向前驱或后继的线索；

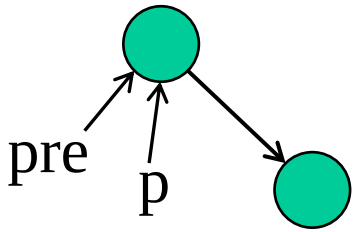
2、前驱或后继的信息只有在遍历时才能得到；

3、线索化：在遍历的过程中修改线索；

pre 始终指向刚刚访问过的节点；

p 指向当前访问过的节点，pre指向他的前驱；

线索化的实质是在遍历过程中，修改空指针的过程。



初始化过程

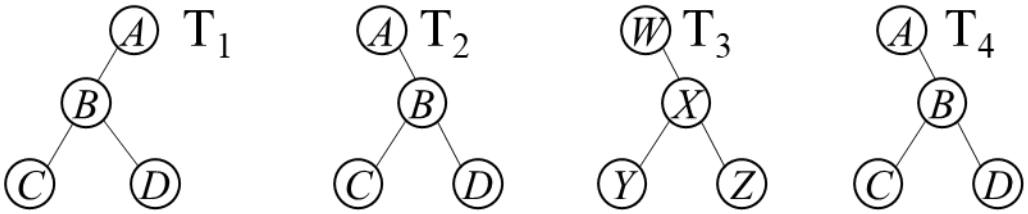
```
Status  InOrderThreading(THTREE  &Thrt,  THTREE  T)
{  if ( !(Thrt = (THTREE)malloc(Sizeof(LNode))))  exit(OVERFLOW); //头结点
   Thrt->ltag = 1 ; Thrt->rtag = 1 ;
   Thrt->rchild  = Thrt;                                  //右指针回指
   if ( !T ) { Thrt->lchild  = Thrt ;Thrt->ltag=0;}       //若二叉树空则左指针回指
   else {
          Thrt->lchild  = T ;     pre = Thrt ;            //初始pre指向头结点head
           InThreading( T ) ;                             //中序线索化
           pre->rchild  = Thrt;   pre->rtag  =  0 ;       //最后结点线索化
          Thrt->rchild=Thrt;  //  Thrt->rchild  = pre ;    //另外一种定义方式
        }
   return OK ;
}
```

线索化过程

```
Void  InThreading( THTREE p )
{   if( p )
    {   InThreading( p->lchild );                                  //左子树线索化
        if( ! p->lchild ) {   p->ltag  = 0 ;  p->lchild  =  pre; }  //无左子树，线索化前驱
        if( !pre->rchild ) {  pre->rtag = 0 ; pre->rchild = p; }   //无右子树，线索化后继
        pre = p;                                                    //保持pre 指向 p 的前驱
        InThreading( p->rchild );                                  //右子树线索化
    }
}
```

中序线索化算法

3.2.5 二叉树的复制



1、二叉树的相似与等价

两棵二叉树具有相同结构指：

“形状”相同

（1）它们都是空的；

（2）它们都是非空的，且左右子树分别具有相同结构。

【定义】具有相同结构的二叉树为相似二叉树。

【定义】相似且相应结点包含相同信息的二叉树称为等价二叉树。

判断两棵二叉树是否等价算法：

```
int   Equal( firstbt ,  secondbt )
BTREE  firstbt , secondbt ;
{  int  x ;
   x = 0 ;
   if ( IsEmpty(firstbt) && IsEmpty(secondbt) )
      x = 1 ;
   else if ( !IsEmpty( firstbt ) && ! IsEmpty( secondbt ) )
      if (  Data( firstbt ) == Data( secondbt )  )
         if ( Equal( Lchild( firstbt ) , Lchild( secondbt ) ) )
            x= Equal( Rchild( firstbt ) , Rchild( secondbt) )
   return( x ) ;
} /* Equal */
```

2、二叉树的复制（空间）

```
BTREE   Copy( BTREE   oldtree )
{
   BTREE   temp ;
   if ( oldtree != Null )
     {
        temp = New  Node ;
        temp -> lchild = Copy( oldtree->lchild ) ;
        temp -> rchild = Copy( oldtree->rchild ) ;
        temp -> data = oldtree->data ;//根结点
        return ( temp );
      }
   return ( Null ) ;
} /* Copy*/
```

3.3 堆（Heap）

如果一棵完全二叉树的任意一个非终端结点的元素都不小于其左儿子结点和右儿子结点（如果有的话）的元素，则称此完全二叉树为最大堆。

同样，如果一棵完全二叉树的任意一个非终端结点的元素都不大于其左儿子结点和右儿子结点（如果有的话）的元素，则称此完全二叉树为最小堆。

数据类型呢？

堆的特点：

最大堆的根结点中的元素在整个堆中是最大的；

最小堆的根结点中的元素在整个堆中是最小的。

（最大堆）操作：
1、MaxHeap(heap) 创建一个空堆
2、HeapFull(heap) 判断堆是否为满
3、Insert(heap,item) 插入一个元素
4、HeapEmpty(heap) 判断堆是否为空
5、DeleteMax(heap) 删除最大元素

（最大堆）类型定义

```
#define  Maxsize  200
Typedef  struct {
     int   key;
     /*  other fields  */
     }  ElementType ;
Typedef struct {
     ElementType elements[ MaxSize ];
     int n ;    /*当前元素个数计数器*/
     } HEAP;
```

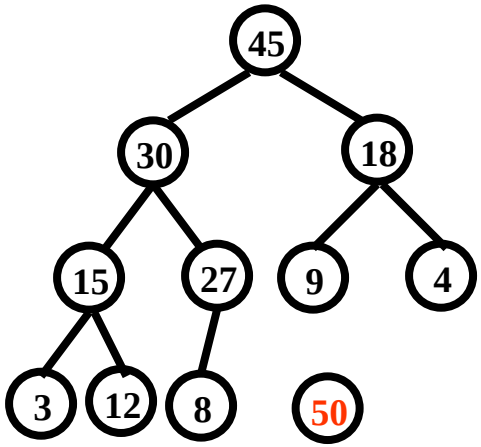
```
Void  MaxHeap ( Heap heap )
{
   heap.n = 0 ;
}
```

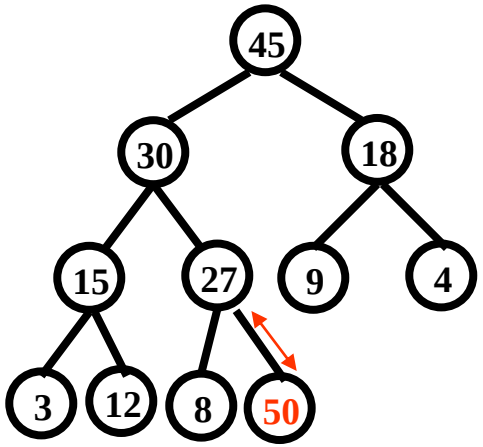
堆ADT操作

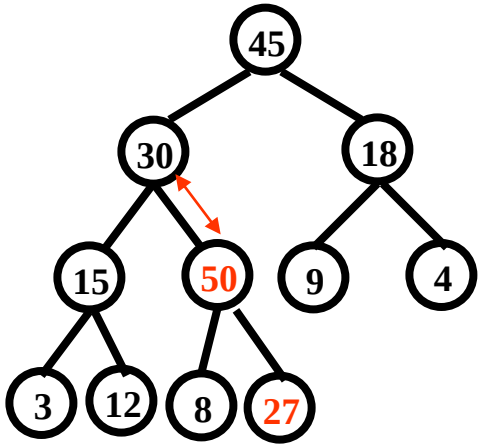
```
Bool HeapEmpty( HEAP heap )
{
   return( !heap.n );
}
```

```
Bool HeapFull ( HEAP heap )
{
   return( heap.n == MaxSize -1 );
}
```

Insert (HEAP heap , ElementType element)

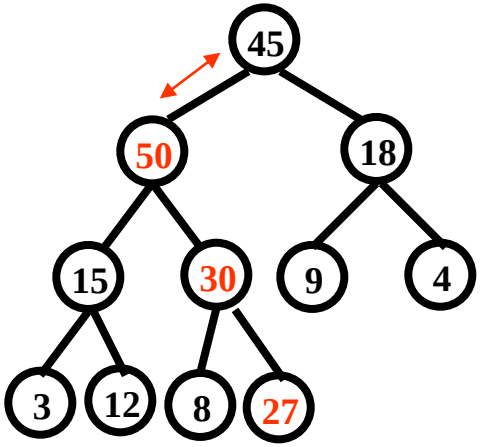


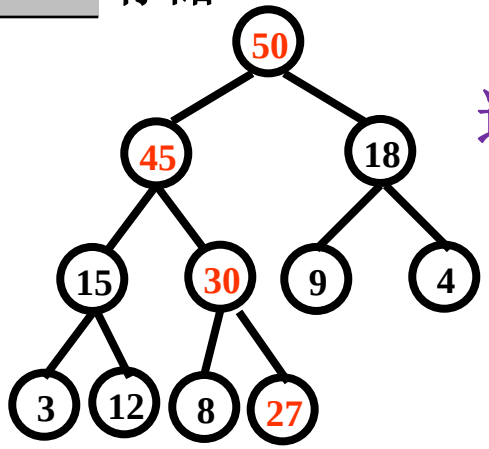




45	30	18	15	27	9	4	3	12	8	50	…

存储

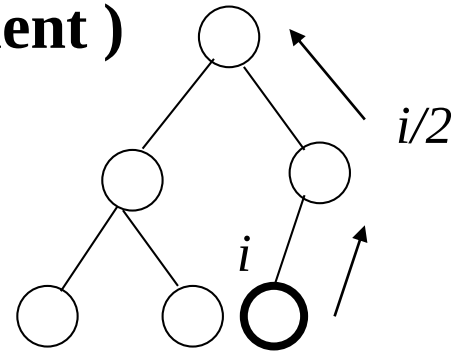




这样做的意义？

50	45	18	15	30	9	4	3	12	8	27	…

```
Void  Insert ( HEAP heap , ElementType  element )
{
    int  i ;
    if (!  HeapFull ( heap ) )
     {
       i = heap.n + 1 ;
      while ( ( i != 1 ) && (element > heap.element [ i/2 ] ))
      {
          heap.elements[i] = heap.elements[i/2] ;
           i/=2 ;
       }
      heap.elements[i] = element ;
    }
     heap.n++;
}
```

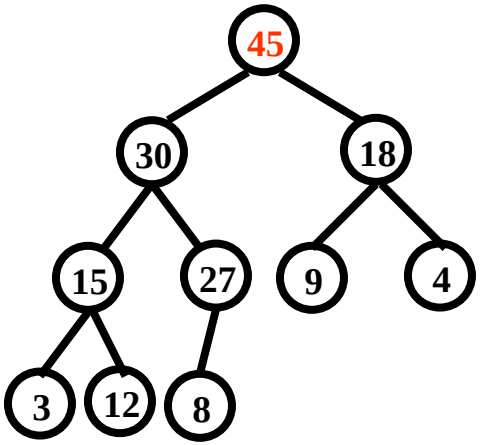


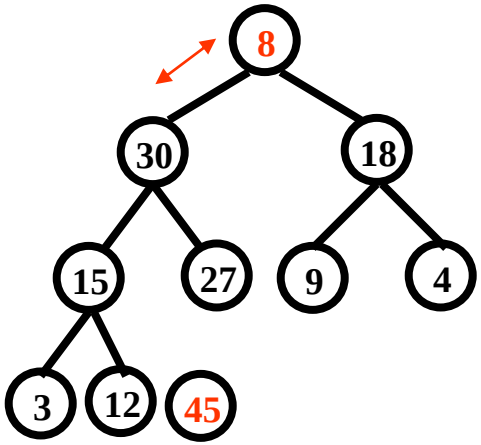
树的高度为 $\lceil \log(n+1) \rceil$

$T_n = O(\log n)$

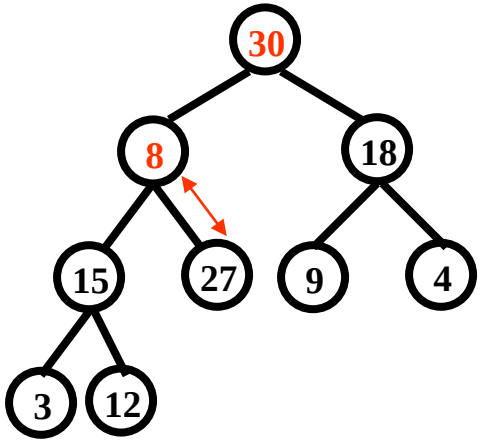
DeleteMax(HEAP heap)

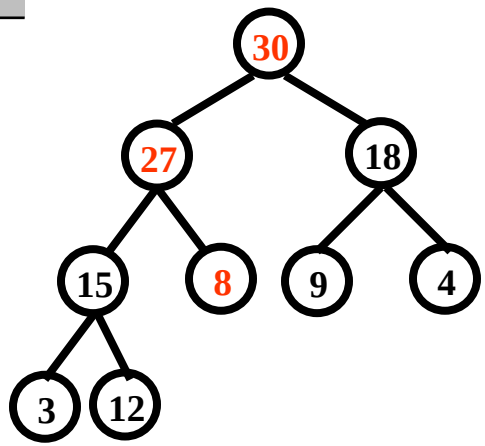
对于最大堆的删除，不能自己进行选择删除某一个结点，只能删除堆的根结点。





45	30	18	15	27	9	4	3	12	8		…

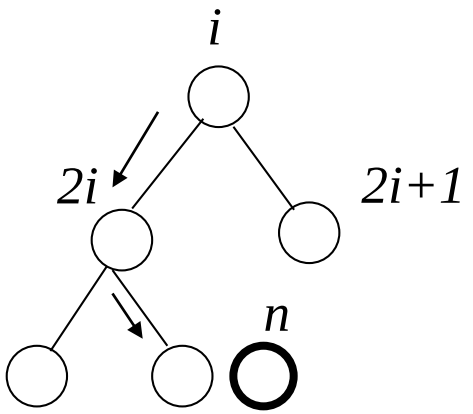




30	27	18	15	8	9	4	3	12	45		…

删除堆中的最大元素（根）

```
Void  DeleteMax( HEAP heap  )
{
   int  parent = 1 , child = 2 ;
   ElementType element , tmp ;
   if (!  HeapEmpty ( heap ) )
    {
      element = heap.elements[1] ;
      tmp = heap.elements[heap.n --] ;
      while ( child <= heap.n )
      {
          if (( child < heap.n ) &&
               ( heap.elements [ child ] < heap.elements[ child +1 ] ))
             child ++ ;
          if ( tmp >= heap.elements [ child ] )    break ;
          heap.elements [ parent ] = heap.elements[ child ] ;
          parent = child ;
          child *= 2 ;
      }
      heap.elements[ parent ] = tmp ;
      return element ;
    }
}
```

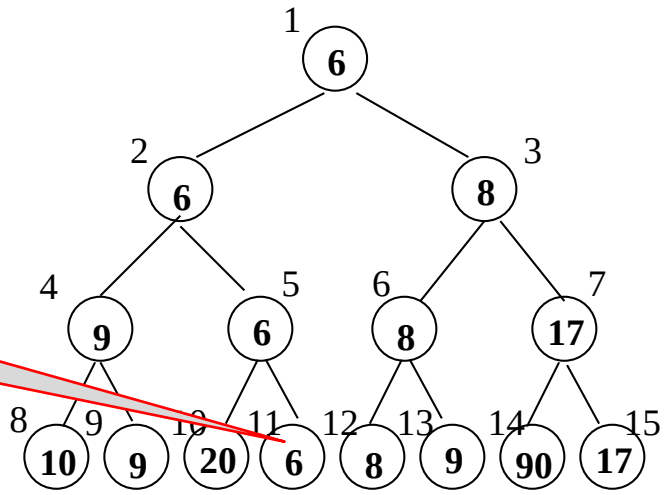


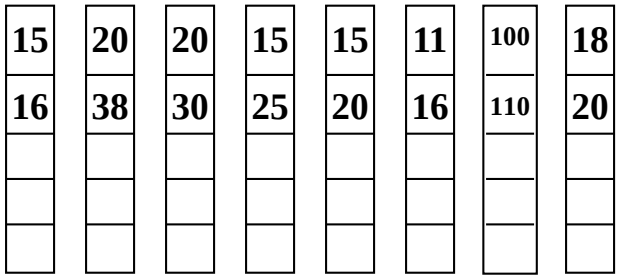
树的高度为 $\lceil \log(n+1) \rceil$

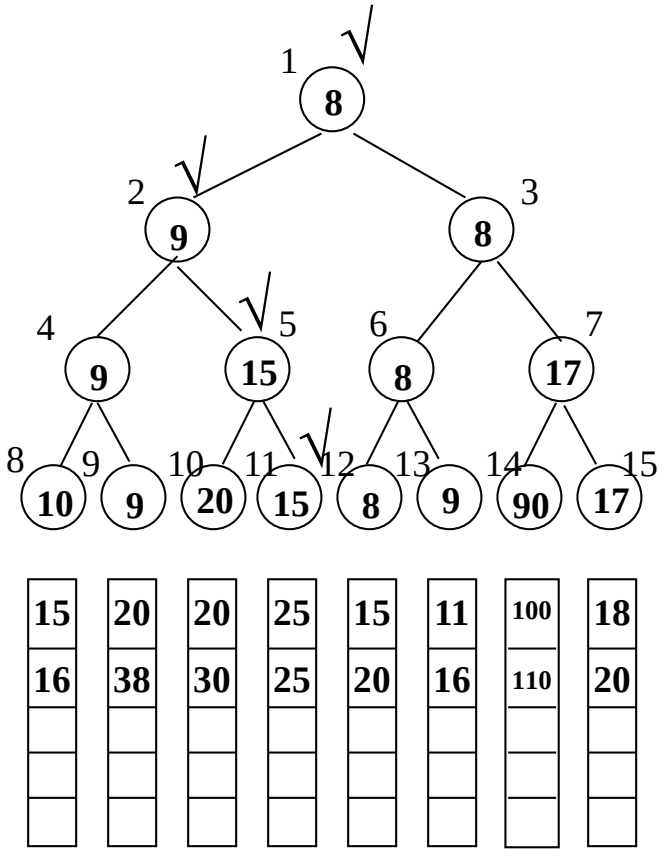
$T_n = O(\log n)$

3.4 选择树（Selection Tree）

一棵选择树是一棵二叉树，其中每一个结点都代表该结点两个儿子中的较小者。这样，树的根结点就表示树中最小元素的结点。

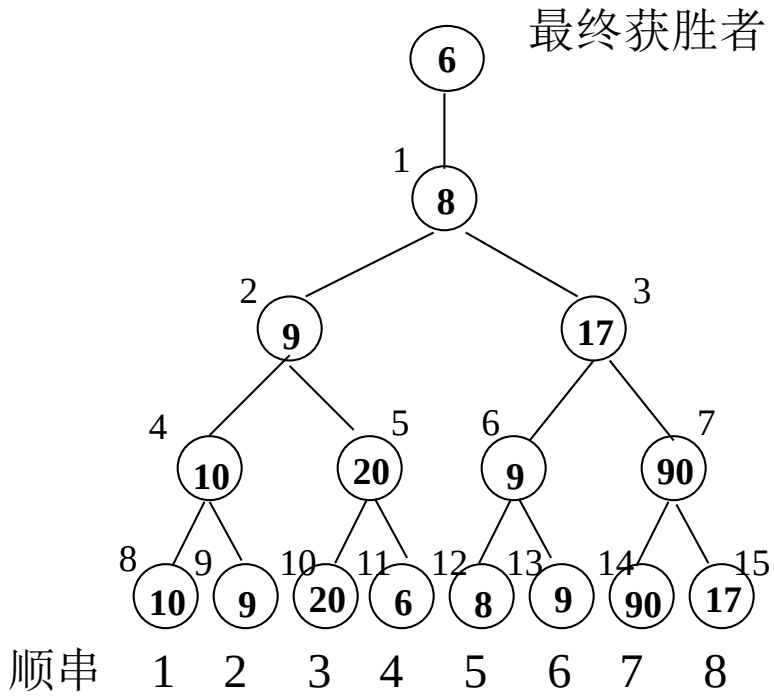




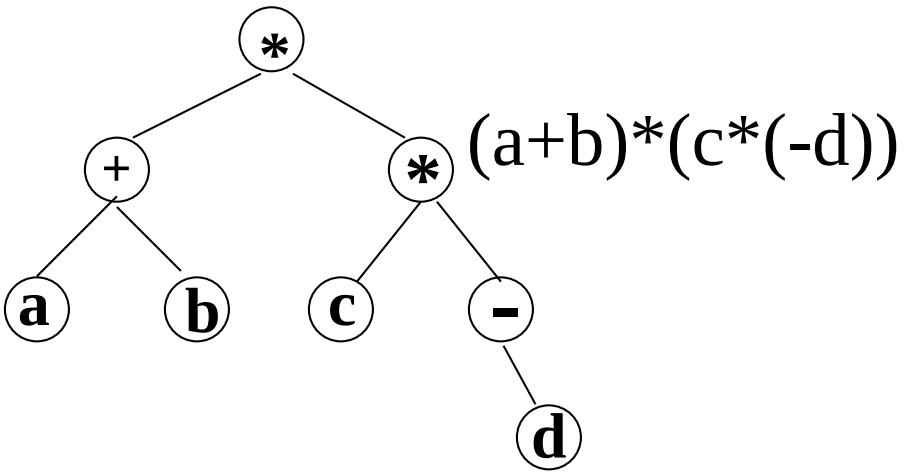


败者树

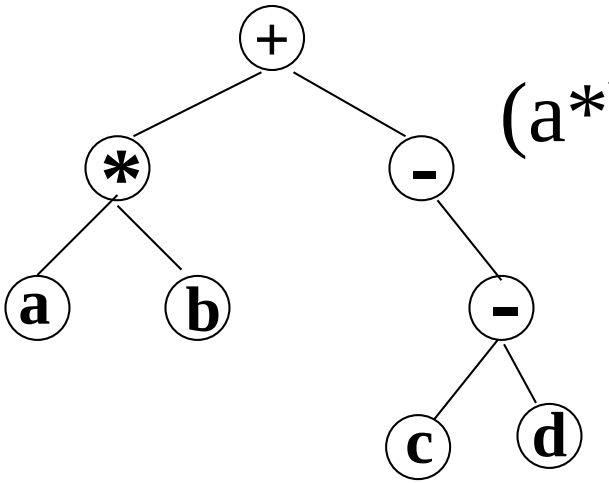
竞赛在兄弟结点之间进行结果放在父结点中。



思考题 如何通过设计算法，实现将给定的表达式树(二叉树) 转换成为等价的中缀表达式(通过括号反应出操作符的计算次序)并输出。例如，将以下两棵表达式树转化成前缀式子：



(a+b)*(c*(-d))



(a*b)+(-(c-d))

提示：中序遍历结果即为中缀表达式，采用中序遍历同时加上必要的括号，那么递归算法稍作修改即可。

除根和叶子节点外，遍历当前结点的**左子树前**加左括号；遍历当前结点的**右子树完**加上右括号。

3.5 树

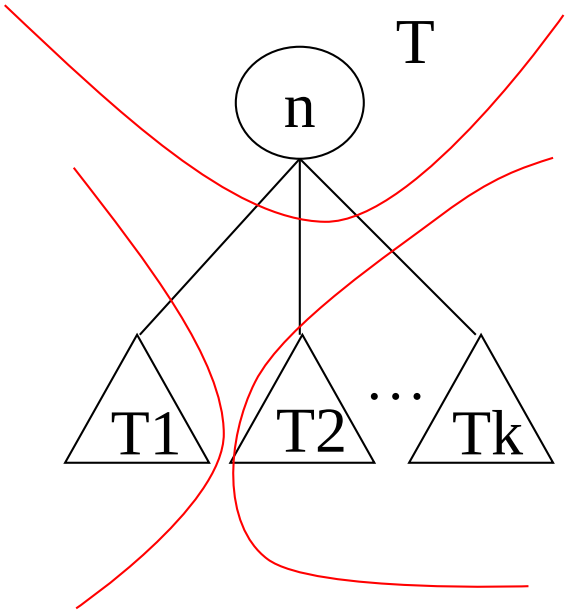
3.5.1 抽象数据型树

- **Parent(n , T)** 求结点 n 的双亲；
- **LeftMost-Child(n , T)** 返回结点 n 的最左儿子；
- **Right-Sibling(n , T)** 返回结点 n 的右兄弟；
- **Data(n , T)** 返回结点 n 的信息；
- **Create $_k$ (v , T_1 , T_2 , …… , T_k) , k = 1 , 2 , ……**

  建立data域v的根结点r , 有k棵子树T_1 , T_2 , …… , T_k

  且自左至右排列；返回r；
- **Root(T)** 返回树T的根结点。

树的三种遍历



- 先(根)序
  - 访问根结点；
  - 先根顺序遍历T_1；
  - 先根顺序遍历T_2；
  - …
  - 先根顺序遍历T_k；
- 中(根)序
  - 中根顺序遍历T_1；
  - 访问根结点；
  - 中根顺序遍历T_2；
  - …
  - 中根顺序遍历T_k；
- 后(根)序
  - 后根顺序遍历T_1；
  - 后根顺序遍历T_2；
  - …
  - 后根顺序遍历T_k；
  - 访问根结点；

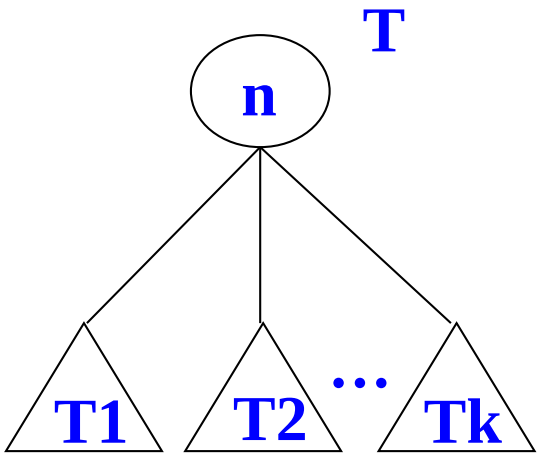
树的按层次遍历

□层次遍历

自上而下

自左到右

依次访问树中的每个结点



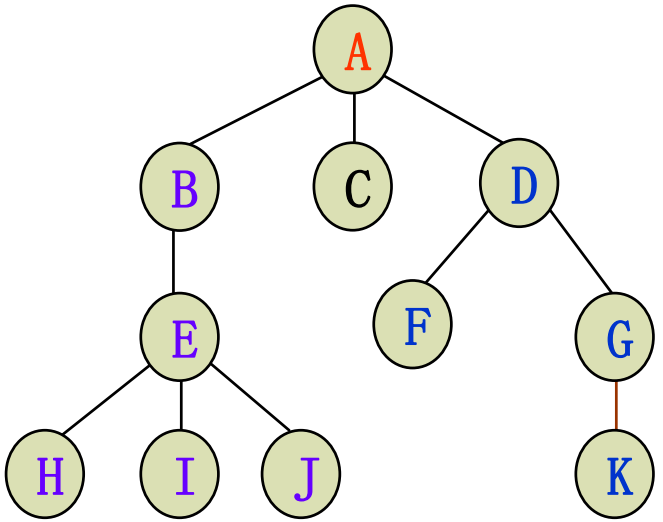
算法思想：

算法运用队列做辅助存储结构。其步骤为

（1）首先将树根入队列；

（2）出队一个结点便立即访问之，然后将它的非空的第一个孩子结点进队，同时将它的所有非空兄弟结点逐一进队；

（3）重复（2），这样便实现了树按层遍历。



【例3-22】假设树的类型为TREE，结点的类型为Node，数据项的类型为elementtype，用递归方法给出树的先根遍历如下：

```
Void    PreOrder( n , T )
Node  n ; TREE  T ;
{
    Node  c ;
    visit( Data( T ) ) ;
    c = LeftMost-Child( n , T ) ;
    while ( c != Null )
        {
            PreOrder( c , T ) ;
            c = Right-Sibling( c , T ) ;
         }
}
```

3.5.2　树的存储结构

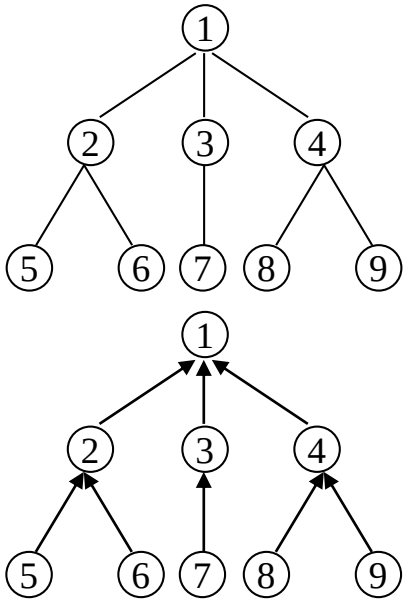
面向特定的操作，设计合适的存储结构

1、树的双亲表示法（~~数组实现方法~~）

树的结点依次编号为1，2，3，…，n；设数组A[i]

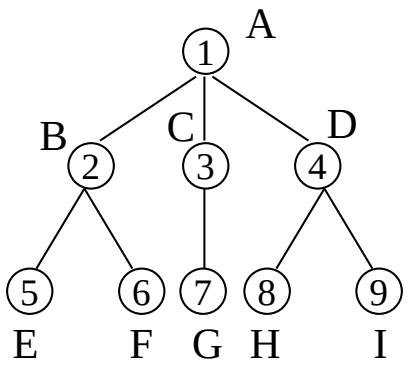
$$A[i] = \begin{cases} 0 & \text{若结点 i 是根} \\ j & \text{若结点 i 的父亲是 j} \end{cases}$$

A		0	1	1	1	2	2	3	4	4
	0	1	2	3	4	5	6	7	8	9



一般有：Parent(i) = A[i]

```
Struct  Node  {
        int  parent ;
        ElementType  data ;  } ;
Typdef  Node  TREE[11];
TREE  T ;
```



T

	0	1	2	3	4	5	6	7	8	9
parent	0	1	1	1	2	2	3	4	4	
data	A	B	C	D	E	F	G	H	I	

树的双亲表示法的改进方案

T[7].parent = 3 ;

T[7].data = G ;

```
Typedef   int   Node ;
Typedef  Node  TREE[MaxNodes] ;
```

```
Node   LeftMost-Child( n , T )
Node  n ;  TREE  T ;
{   Node  i ;
    for ( i = n + 1 ; i <= MaxNodes –1 ; i++ )
    if ( T[i] == n )
       return( i ) ;    i 为最左孩子
    return( 0 ) ;       n是叶子
}
```

算法

LeftMost-Child

如何找D的孩子呢?

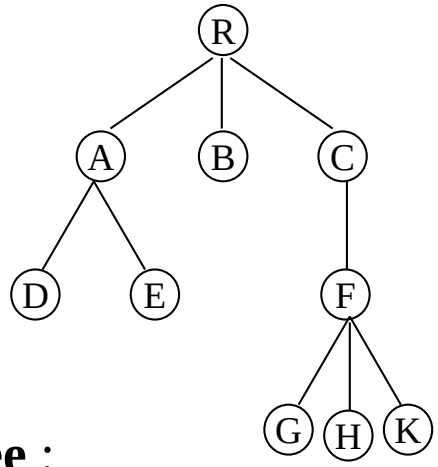
2、树的孩子表示法（~~邻接表表示法~~）

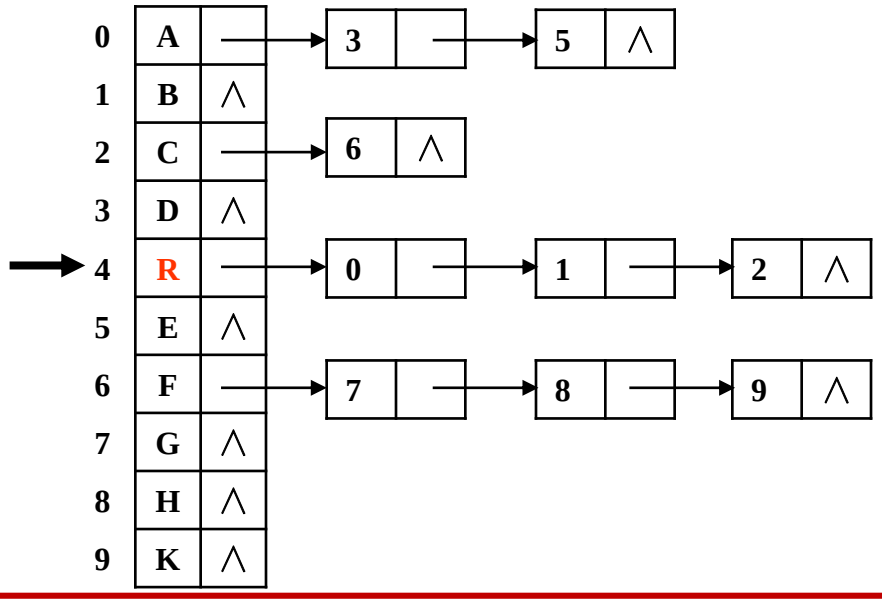
对树的每一个结点用线性链表存储它的孩子结点。

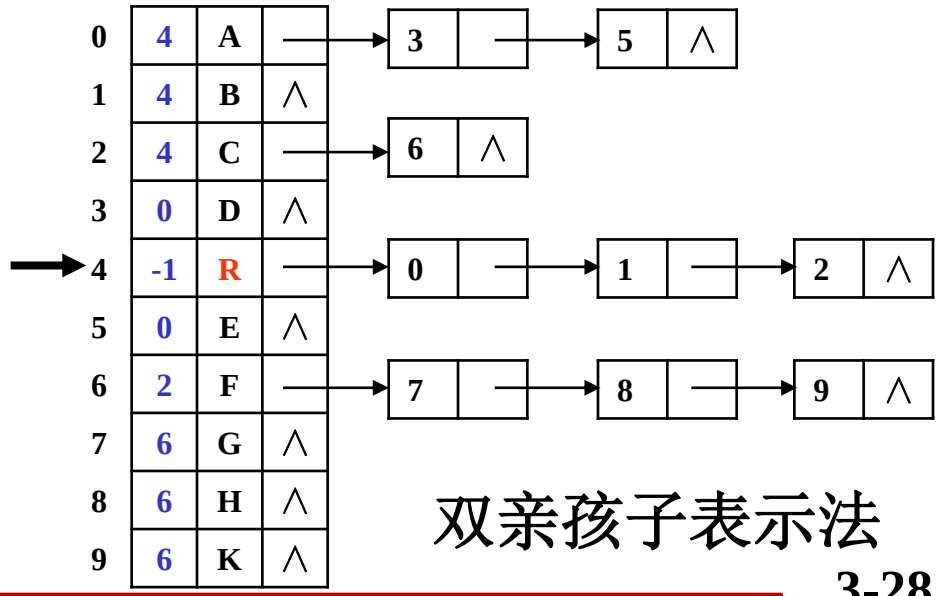
```
typedef  struct  CTNode  {
  int    child ;
  struct  CTNode  *next ; }   *ChildPtr ;
typedef  struct  {
   ElementType  data ;
   ChildPtr  firstchild ;      }    CTBox ;
typedef  struct  {
   CTBox  Nodes[MAX_TREE_SIZE] ;  int   n , r ; }  Ctree ;
```

child	next

data	firstchild

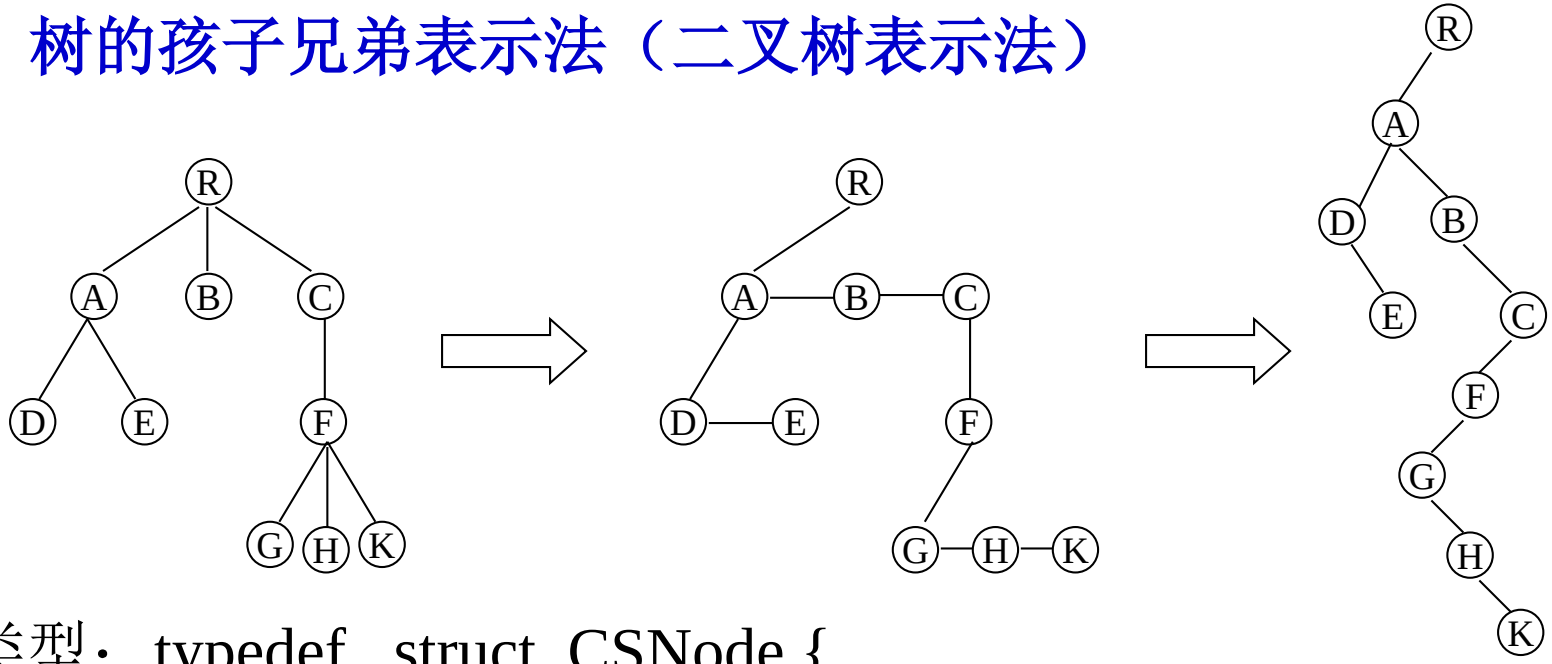






双亲孩子表示法

3、树的孩子兄弟表示法（二叉树表示法）

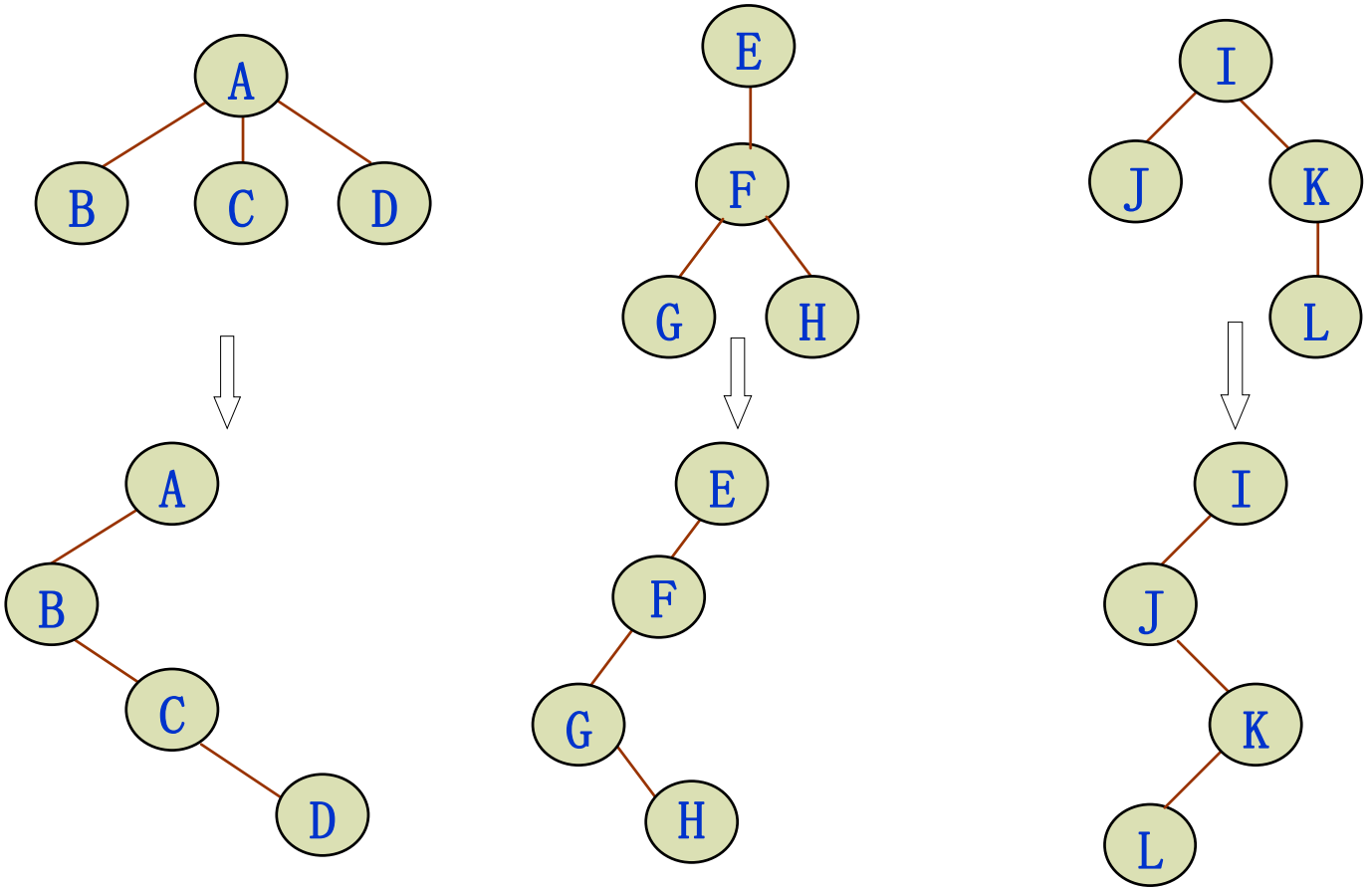


类型：typedef　struct　CSNode {
　　ElemType　　data;
　　struct　CSNode　*firstchild , *nextsibling ; } ;

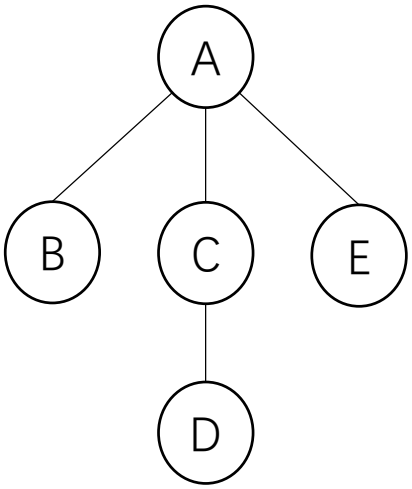
左孩子
右兄弟

遍历	树	二叉树
先序	RADEBCFGHK	RADEBCFGHK
中序	DAERBGFHKC	DEABGHKFCR
后序	DEABGHKFCR	EDKHGFCBAR

将树转换为二叉树



左右地址域分别指向该结点第一个孩子结点和右边第一个兄弟结点。



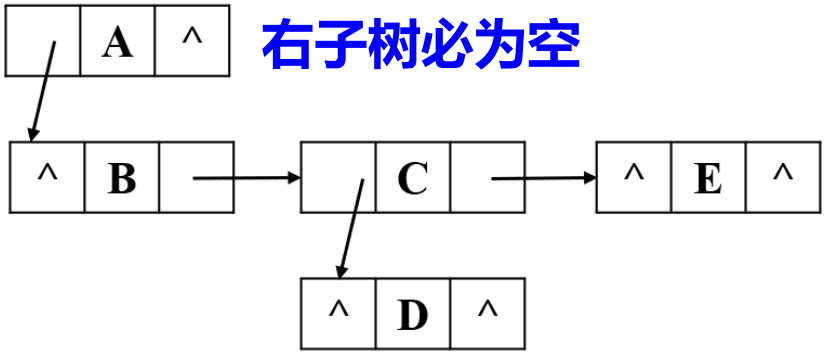

```
Struct {
    DataType data;
    int LeftChild;
    int RightSibling;
}CellSpace[MaxNodes];
```

CellSpace

0			
1			
2	Λ	D	Λ
3	Λ	E	Λ
4			
5	Λ	B	11
	•••	•••	•••
10	5	A	Λ
11	2	C	3
	•••	•••	•••
	Left Child	Data	Right Sibling

r

LeftMost RightSibling



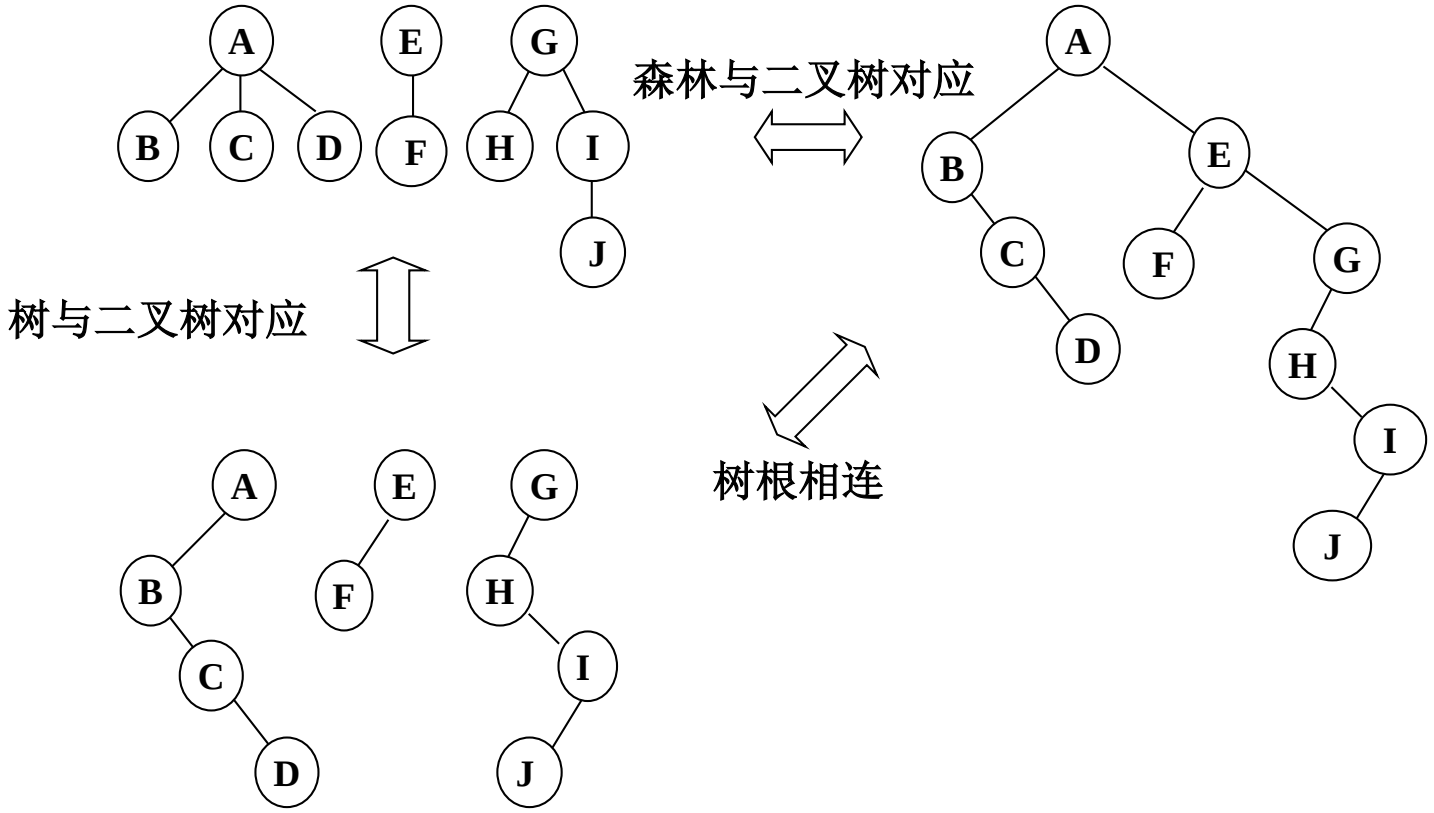
右子树必为空

3.6 森林与二叉树

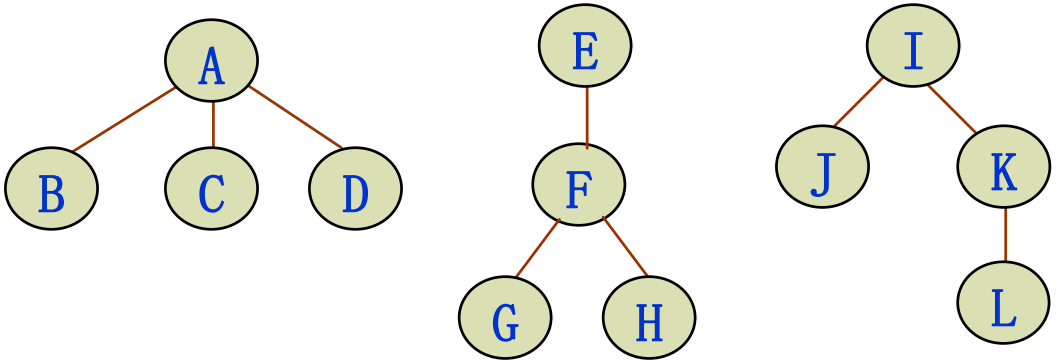
森林转换为二叉树：

（1）先将森林中每棵树转换成二叉树；

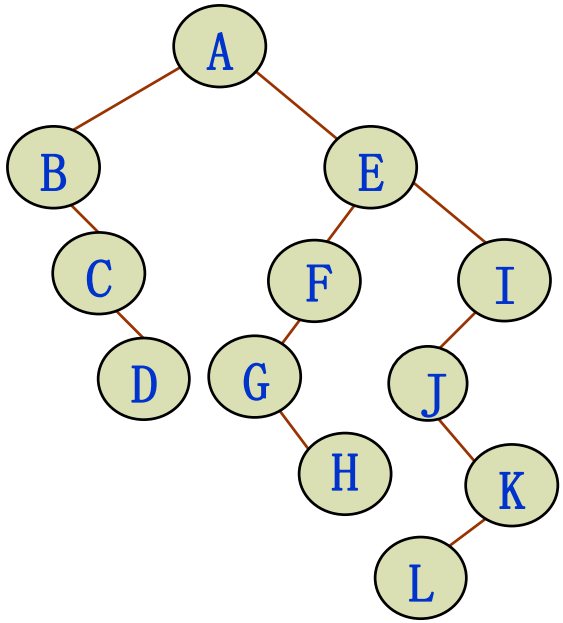
（2）二叉树的树根连接起来。



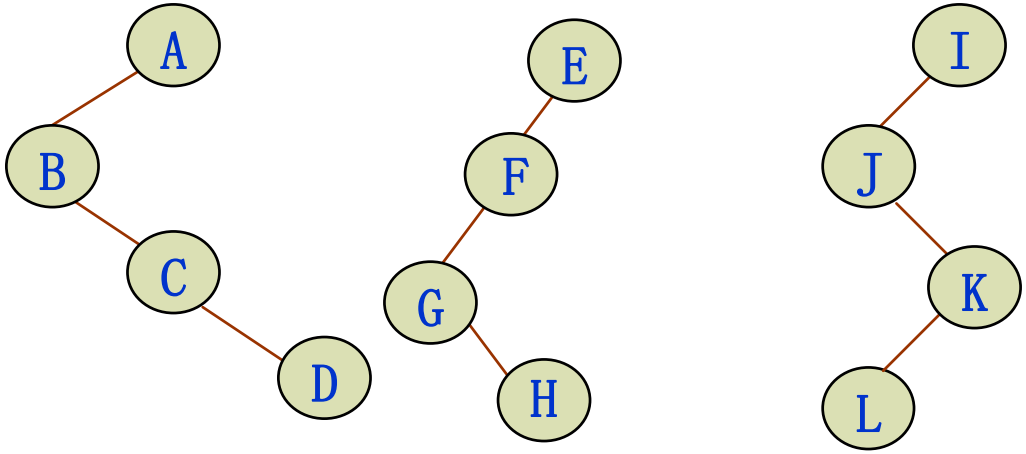
将森林转换为二叉树



包含3棵树的森林



森林对应的二叉树



每棵树对应的二叉树

中根遍历呢？

遍历森林与遍历二叉树的对应关系

遍历森林		遍历相应的二叉树	
先序	访问第一棵树的根； 先根顺序遍历第一棵树的全部子树； 先根顺序遍历其余全部树。	先序	访问树根； 先根顺序遍历左子树； 先根顺序遍历右子树。
中序	中序遍历森林中第一棵树的根节点的子树森林； 访问第一棵树的根节点； 中序遍历除去第一棵树之后剩余的树构成的森林。	中序	中根顺序遍历左子树； 访问树根； 中根顺序遍历右子树。

森林的中序遍历与二叉树的中序遍历方法的定义不同！！

遍历森林与遍历树的对应关系

遍历森林		遍历树	
先序	访问第一棵树的根； 先根顺序遍历第一棵树的全部子树； 先根顺序遍历其余全部树。	先序	访问根； 先根顺序遍历全部子树。
中序	中根顺序遍历第一棵树的根结点的全部子树； 访问第一棵树的根； 中根顺序遍历除去第一棵树之后剩余树构成的森林。	后序	后根顺序遍历全部子树； 访问根。

树、森林、二叉树的遍历对应关系		
树	**森林**	**对应的二叉树**
先序遍历	先序遍历	先序遍历
后序遍历	中序遍历	中序遍历
层次遍历		后序遍历

例题 将森林转换成对应的二叉树，若在二叉树中，结点u是结点v的父结点的父结点，则在原来森林中，u和v具备哪种关系（）
I 父子关系 II 兄弟关系 III u的父结点与v的父结点互为兄弟 I; II

例题 将森林F转换为对应的二叉树T，则F中叶子结点个数等于(C)
A. T中叶子结点个数 B. T中度为1的结点个数
C. T中左孩子指针为空的结点个数 D. T中右孩子指针为空的结点数

例题 若将一棵树T转化成对应的二叉树BT，则下列对BT的遍历中，其遍历序列与T的后根遍历序列相同的是(B)
A. 先根遍历 B. 中根遍历 C. 后根遍历 D. 层次遍历

例题 已知森林F及与之对应的二叉树T，若F的先根遍历序列是a,b,c,d,e,f，中根遍历序列是b,a,d,f,e,c，则T的后根遍历序列是(C)
A. b,a,d,e,f,c B. b,d,e,e,c,a C. b,f,e,d,c,a D. e,f,d,c,b,a F和T 先、中序对应

例题 若森林F有15条边、25个结点，则F包含树的个数是 (C)
A. 8 B. 9 C. 10 D. 11
N个结点的树有n-1条边