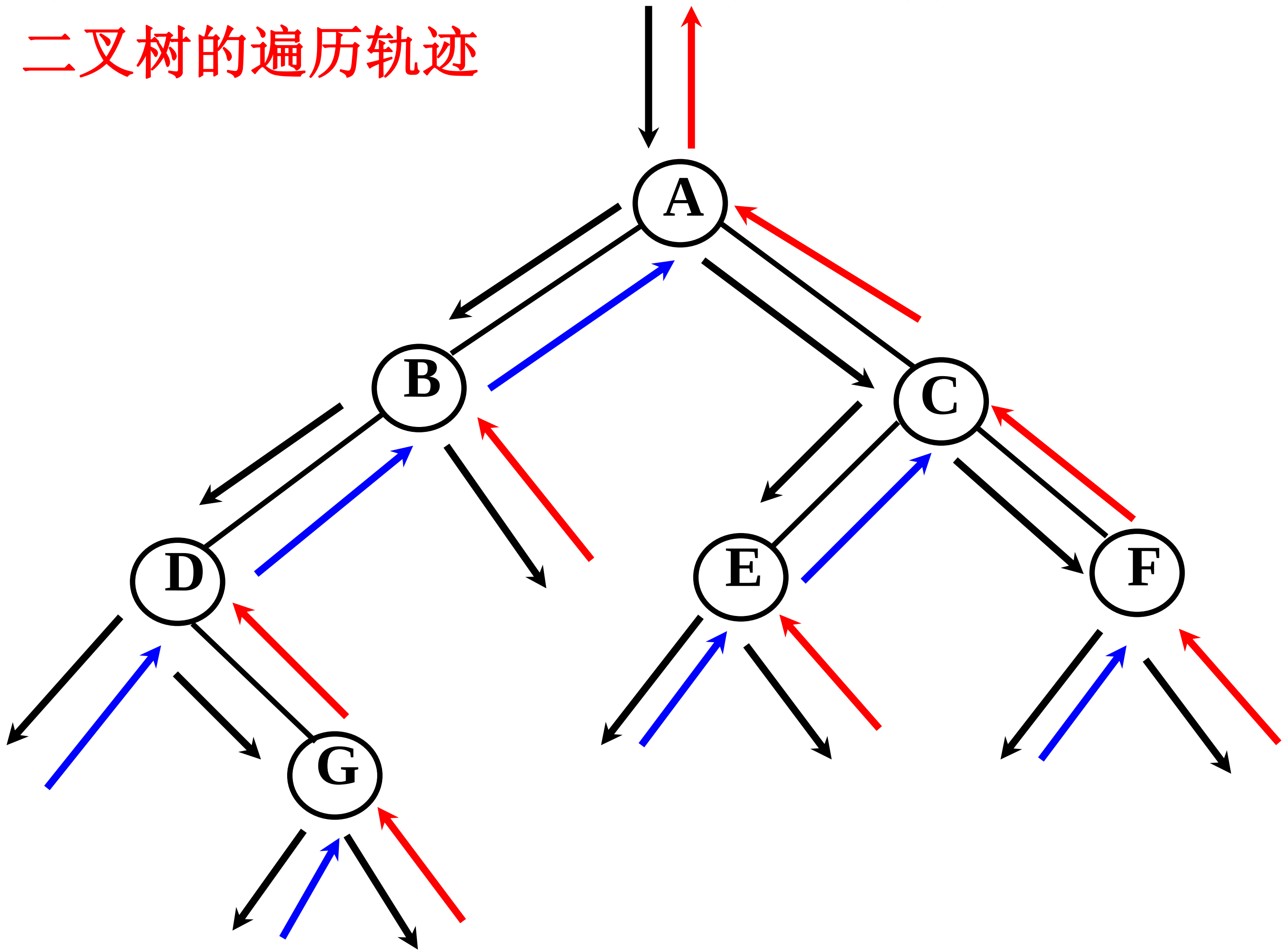
3.2 二叉树（Cont.）

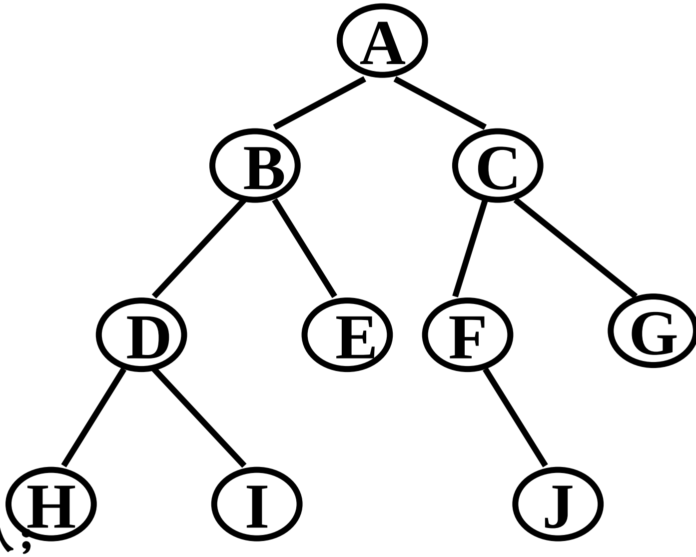
- 二叉树左右链存储结构下的非递归遍历算法
  - 二叉树的遍历轨迹



3.2 二叉树（Cont.）

层序遍历算法

- **基本思想：**按层次顺序遍历二叉树的原则是先被访问的结点的左、右儿子结点先被访问，因此，在遍历过程中需利用具有先进先出特性的队列结构
- **实现步骤：**



1. 队列Q初始化；
2. 如果二叉树非空，将根指针入队；
3. 循环直到队列Q为空

   3.1 q=队列Q的队头元素出队；

   3.2 访问结点q的数据域；

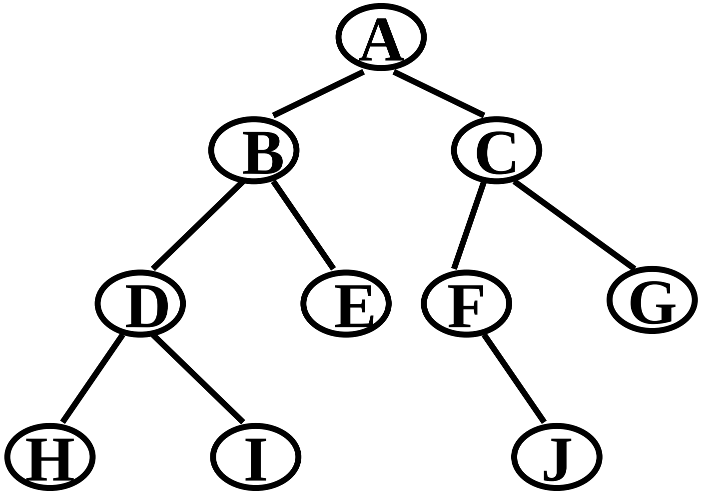
   3.3 若结点q存在左孩子，则将左孩子指针入队；

   3.4 若结点q存在右孩子，则将右孩子指针入队；

3.2 二叉树（Cont.）

■ 层序遍历算法

```
void LeverOrder (Btree root)
{   front=rear=0;  //采用顺序队列，并假定不会发生上溢
    if (root==Null) return;
        Q[++rear]=root;
    while (front!=rear) {
         q=Q[++front];
         cout<<q->data;
         if (q->lchild!=Null)  Q[++rear]=q->lchild;
         if (q->rchild!=Null)  Q[++rear]=q->rchild;
    }
}
```



3.2 二叉树（Cont.）

遍历算法应用举例

- 二叉树的遍历是二叉树各种操作和算法的基础，遍历算法中对每个结点的“访问”操作可以是对结点进行的各种处理
- 根据遍历算法的框架，适当修改访问操作的内容，可以派生出很多关于二叉树的应用算法。
- 因此，二叉树遍历算法是有关二叉树算法中最核心的算法.

计算二叉树结点个数的递归算法。

```
int Count ( Btree T )
{    if ( T == Null )   return 0;
     else  return 1 + Count ( T->lchild )
                    + Count ( T->rchild );
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
        datatype  data ;
} ;
typedef node  * Btree;
```

3.2 二叉树（Cont.）

```
void Count(node *root)
{   //n为全局量并已初始化为0
   if (root) {
       Count(root->lchild);
       n+ +;
       Count(root->rchild);
   }
}
```

```
void Destroy (Btree  T)
{
   if ( T != Null ) {
     Destroy ( T->lchild );
     Destroy ( T->rchild );
     delete T;
   }
}
```

删除二叉树的递归算法

- 求二叉树高度的递归算法

```
int Height (Btree T )
{  if ( T == Null ) return 0;
   else {int m = Height ( T->lchild );
         int n = Height ( T->rchild ) );
         return (m > n) ? (m+1) : (n+1);
        }
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
           datatype  data ;
} ;
typedef node  * Btree ;
```

3.2 二叉树（Cont.）

- 交换二叉树所有结点子树的算法.

```
void  Exchange ( Btree T )
{   Node *p = T, *tmp;
    if ( p != Null ) {
        temp = p->lchild;
        p->lchild = p->rchild;
        p->rchild = tmp;
        Exchange ( p->lchild );
        Exchange ( p->rchild );
    }
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
            datatype  data ;
} ;
typedef node  * Btree ;
```

3.2 二叉树（Cont.）

```
void Exchange ( Btree T )
{     struct node *p, *tmp;
      top = -1;      //采用顺序栈，并假定不会发生上溢
      if ( T != Null ) {
         s[++top] = T;
         while ( top !  =-1 ) {
             p = s[top--]; //栈中退出一个结点
             tmp = p->lchild//交换子女
             p->lchild = p->rchild;
             p->rchild = tmp;
             if ( p->lchild != Null )
                 s[++top] = p->lchild;
             if ( p->rchild != NULL )
                 s[++top] = p->rchild;
         }使用栈消去递归算法中的两个递归语句
     }
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
          datatype  data ;
} ;
typedef node  * Btree ;
```

3.2 二叉树（Cont.）

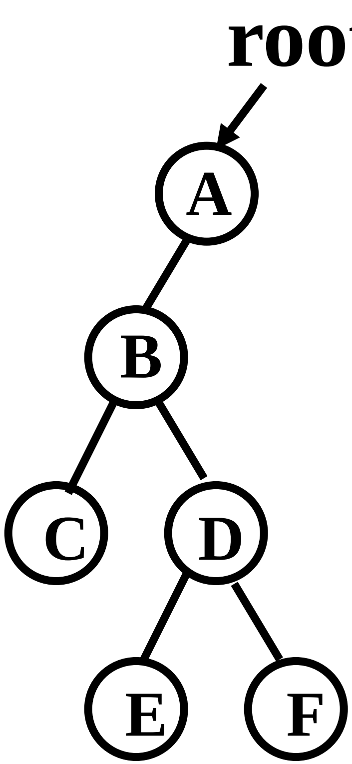
- 按先序次序打印二叉树中的叶子结点的算法.

```
void PreOrder(Btree T )
{
   if (T) {
      if (!T->lchild && !T->rchild)
            cout<<T->data;
      PreOrder(T->lchild);
      PreOrder(T->rchild);
   }
}
```

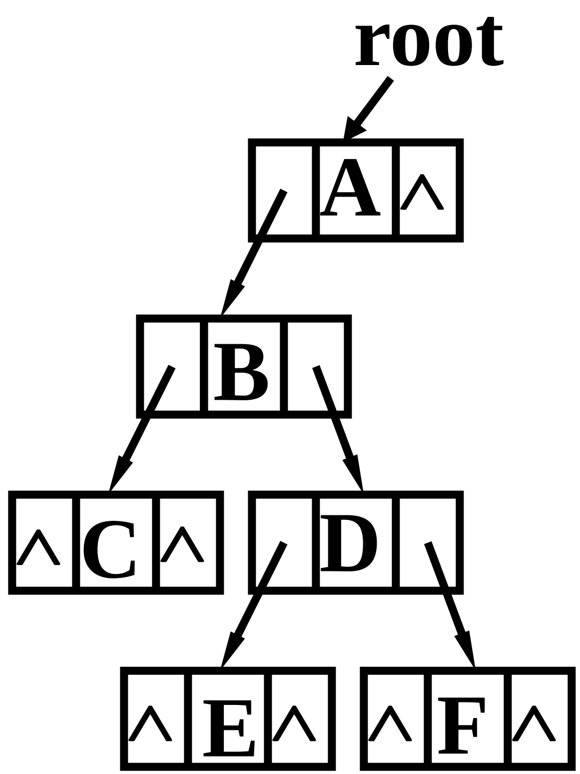
```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
          datatype  data ;
} ;
typedef node  * Btree ;
```

3.2 二叉树（Cont.）

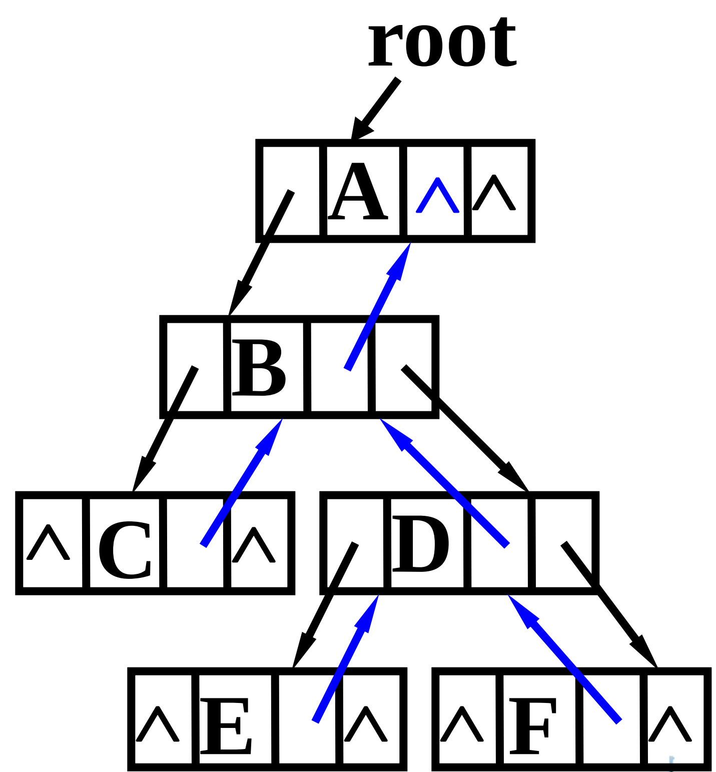
- 二叉树的其他链式存储结构----动态三叉链表
  - 在二叉链表的基础上增加了一个指向双亲的指针域。



二叉树



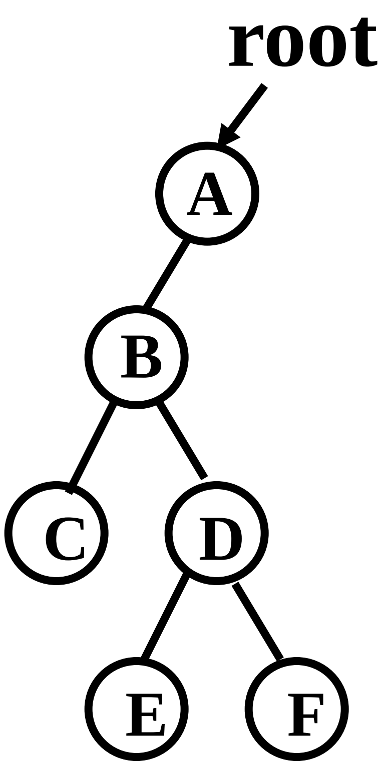
动态二叉链表



动态三叉链表

3.2 二叉树（Cont.）

- 二叉树的其他链式存储结构----静态二叉链表和三叉链表



二叉树

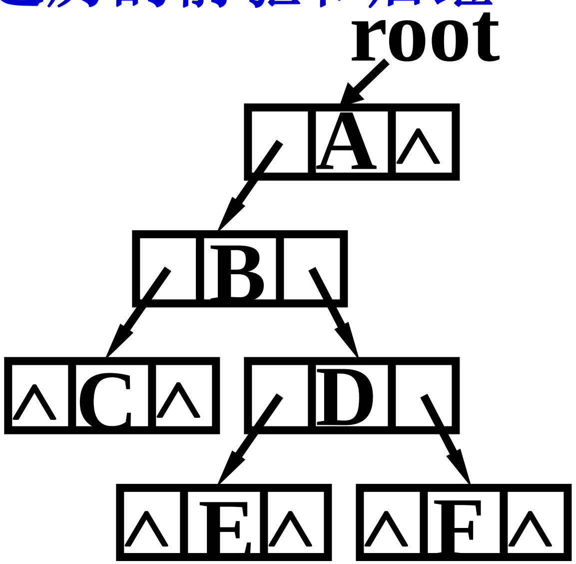
	data	parent	lchild	rchild
0	A	-1	1	-1
1	B	0	2	3
2	C	1	-1	-1
3	D	1	4	5
4	E	3	-1	-1
5	F	3	1	1

静态二叉链表和三叉链表

3.2 二叉树（Cont.）

二叉树的线索链表存储结构----线索二叉树

- 二叉链表的空间利用情况如何？
  - 在n（n≥1）个结点的二叉树左右链表示中，只有n-1个指向子树的指针，却有n+1个空指针域。
- 在二叉链表中如何找某个结点的某种遍历的前驱和后继？
  - 每次都要从根结点进行遍历
- 如何遍历二叉链表表示的二叉树？
  - 利用栈或队列，能否不用？
- 如何利用空指针域解决上述问题？



动态二叉链表

3.2 二叉树（Cont.）

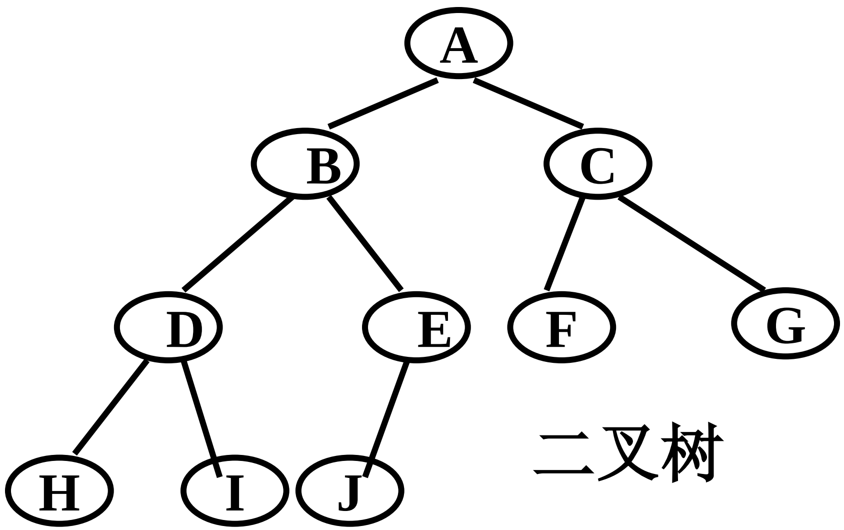
二叉树的线索链表存储结构----线索二叉树

- 若结点p有左孩子，则p->lchild指向其左孩子结点，否则令其指向其（先序、中序、后序、层序）前驱；
- 若结点p有右孩子，则p->rchild指向其右孩子结点，否则令其指向其（先序、中序、后序、层序）后继；

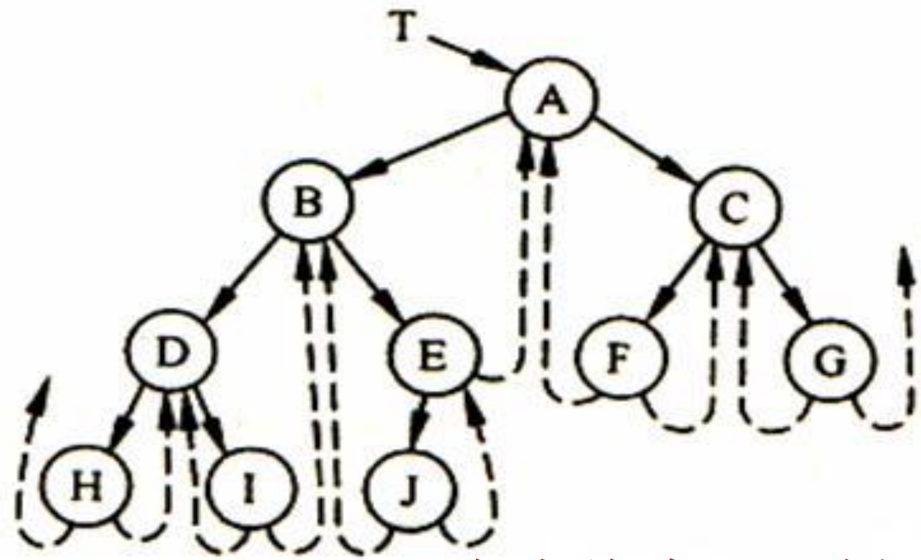
- 如何区分指针是指向其左/右孩子的指针还是指向某种遍历的前驱/后继？
  - 在每个结点中增加两个标志位，以区分该结点的的两个链域是指向其左/右孩子还是指向某种遍历的前驱/后继。

3.2 二叉树（Cont.）

二叉树的线索链表存储结构----线索二叉树



二叉树



中序线索二叉树

结点结构

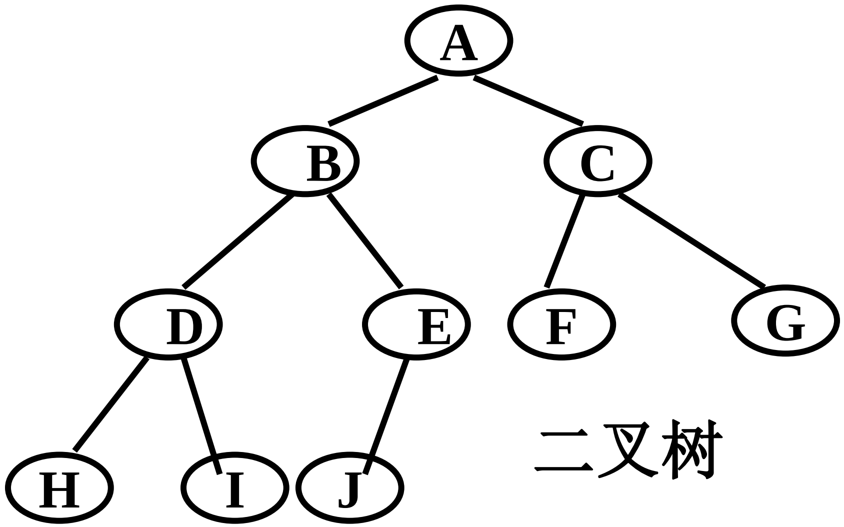
lchild	ltag	data	rchlid	rtag

p->ltag = { True p->lchild 指向左孩子; False p->lchild 指向（中序）前驱 }

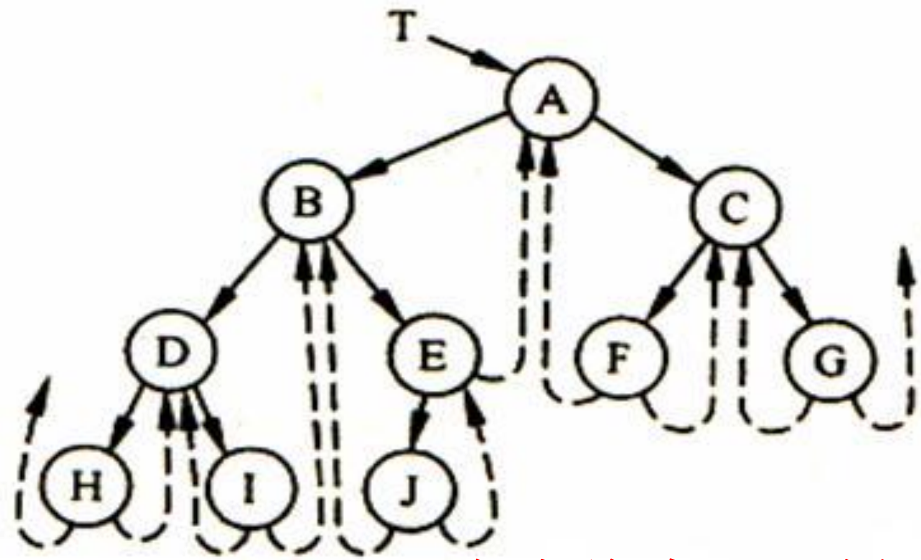
p->rtag = { True p->rchild 指向右孩子; False p->lchild 指向（中序）后继 }

3.2 二叉树（Cont.）

二叉树的线索链表存储结构----线索二叉树



二叉树

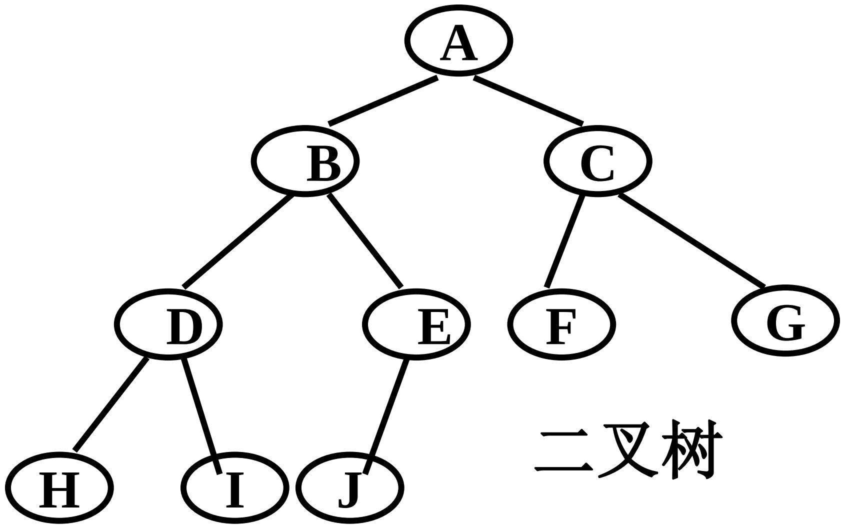


中序线索二叉树

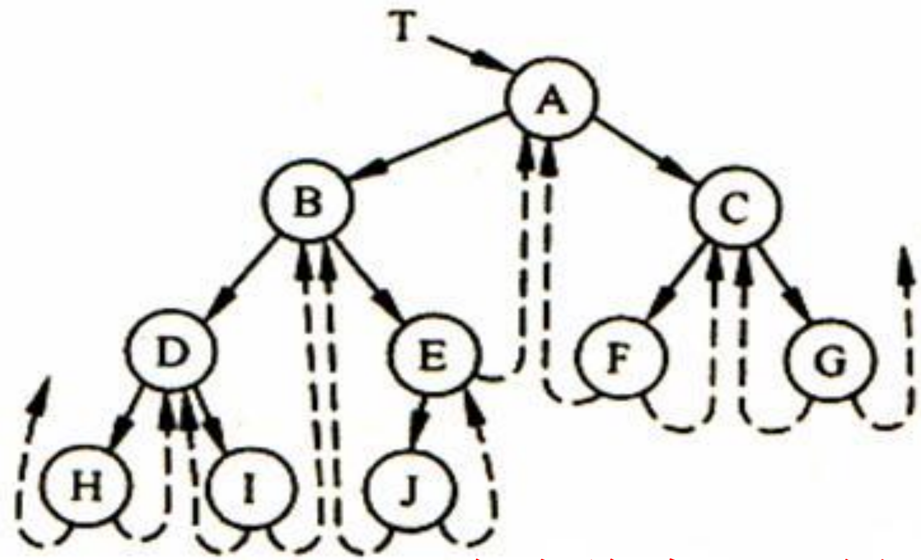
- **线索**：将结点的空指针域指向其前驱/后继的指针被称为**线索**；
- **线索化**：结点的空链域存放其前驱/后继的过程称为**线索化**；
- **线索二叉树**：线索化的二叉树称为**线索二叉树**。

3.2 二叉树（Cont.）

二叉树的线索链表存储结构----线索二叉树



二叉树



中序线索二叉树

- 二叉树的遍历方式有4种，故有4种意义下的前驱和后继，相应的有**4种线索二叉树**：

  (1) 先序线索二叉树；(2) 中序线索二叉树；

  (3) 后序线索二叉树；(4) 层序线索二叉树。

3.2 二叉树（Cont.）

- 线索链表的存储结构定义

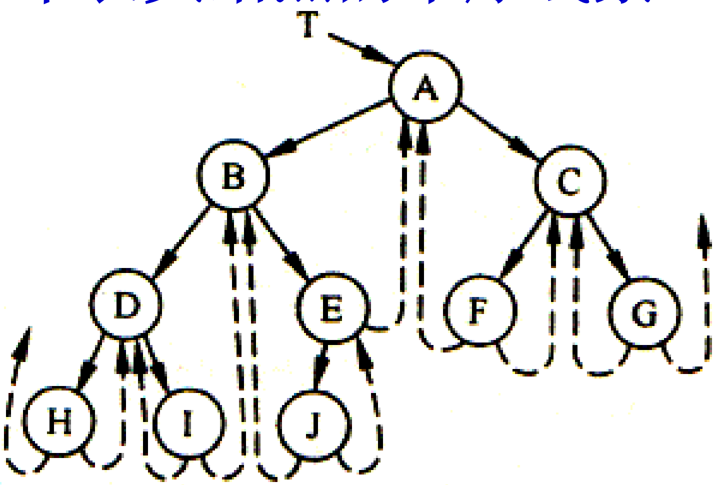
结点结构

lchild	ltag	data	rchlid	rtag

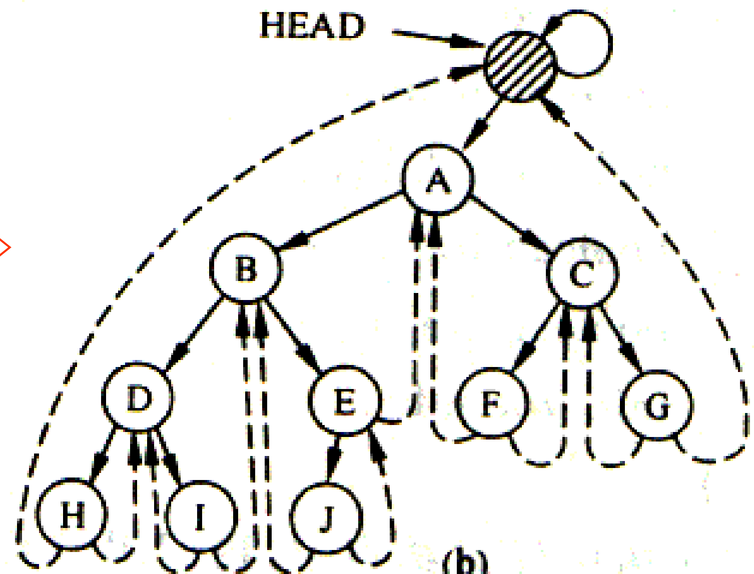
```
struct  node {
      datatype  data ;
      struct  node *lchild, *rchild;
                  bool   ltag, rtag;
};
typedef  struct node  *  ThTree;
```

3.2 二叉树（Cont.）

- 带表头结点的中序线索二叉树



(a)



(b)

lchild	ltag	data	rchlid	rtag

非空二叉树：

head->lchild = T；　（根）

head->ltag = True ;

head->rchild = head ;

head->rtag = True ;

空二叉树：

head->lchild = head ;

head->ltag = False ;

head->rchild = head ;

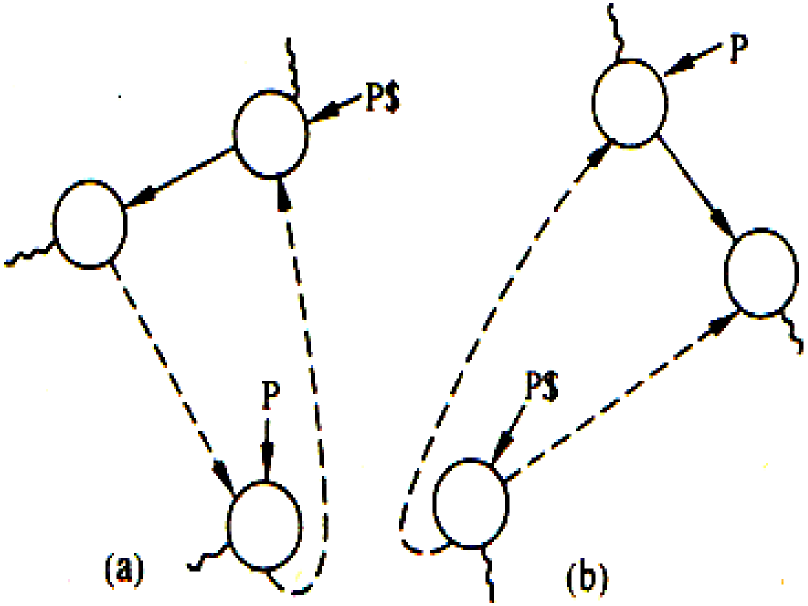
head->rtag = Ture ;

3.2 二叉树（Cont.）

线索二叉树的若干算法

- 算法1：在中序线索二叉树中求一个结点p的中序后继p$
- 分析:
  - (1)当p->rtag==False时，p->rchild 即为所求(线索)。
  - (2)当p->rtag==True时，p$为p 的右子树的最左结点。
- 算法实现:

```
ThTree  InNext( ThTree  p)
{ThTree  Q ;
    Q=p->rchild ;
    if (p->rtag = = True )
      while( Q->ltag = = True )
        Q = Q->lchild ;
    return ( Q ) ;
}
```

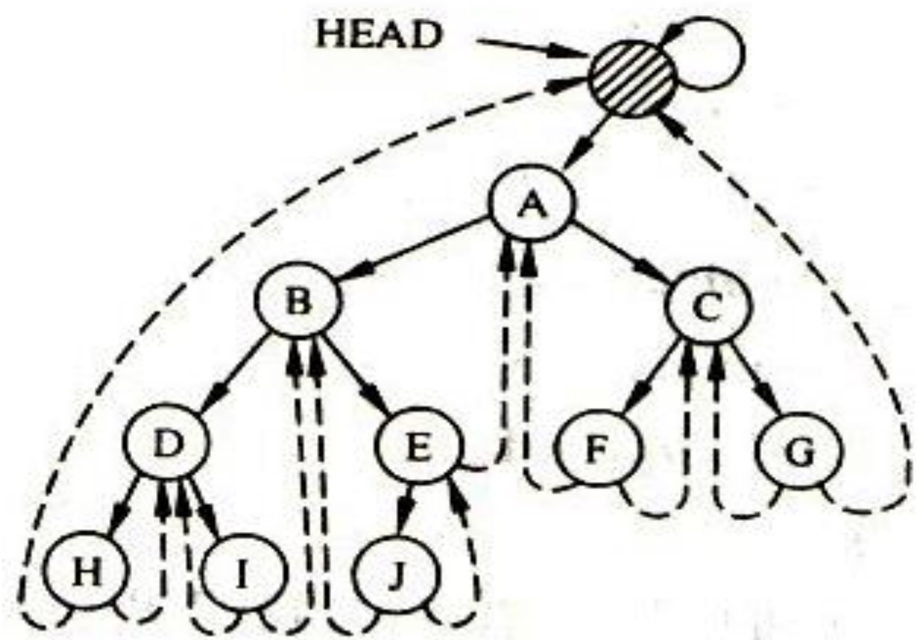


3.2 二叉树（Cont.）

- 算法2：利用InNext算法，中序遍历线索二叉树
- 算法实现：

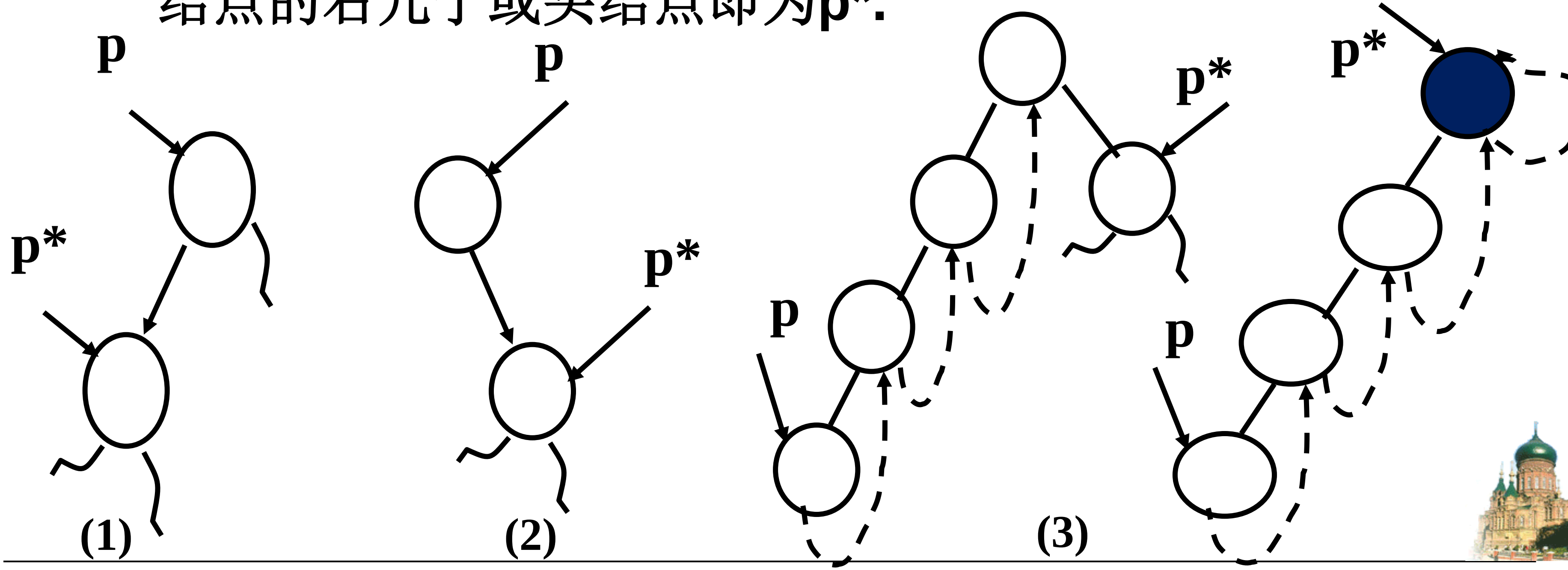
```
void ThInOrder(ThTree HEAD)
{   ThTree   tmp ;
    tmp = HEAD ;
    do {
        tmp = InNext ( tmp ) ;
        if ( tmp != HEAD )
            visit ( tmp -> data ) ;
    } while ( tmp != HEAD ) ;
}
```

```
head->lchild = T
head->rchild = head ;
head->ltag = True ;
head->rtag = True ;
```



3.2 二叉树（Cont.）

- 算法3：求中序线索二叉树中结点p 的先序顺序后继结点p*
- 分析：
  - (1) p 的左子树不空时，p 的左儿子p->lchild 即为 p*；
  - (2) p 的左子树空但右子树不空时，p 的p->rchild 为p*；
  - (3) p 的左右子树均空时，右线索序列中第一个有右孩子结点的右儿子或头结点即为p*.



3.2 二叉树（Cont.）

- 算法3：求中序线索二叉树中结点p 的先序顺序后继结点p*
- 算法实现：

```
THTREE   PreNext( ThTree  p)
{   ThTree   Q ;
     if (p->ltag = = True )
               Q=p->lchild ;
      else{ Q = p;
                while(Q->rtag = = False)
                     Q = Q->rchild ;
                Q = Q->rchild ;
        }  return ( Q ) ;
}
```

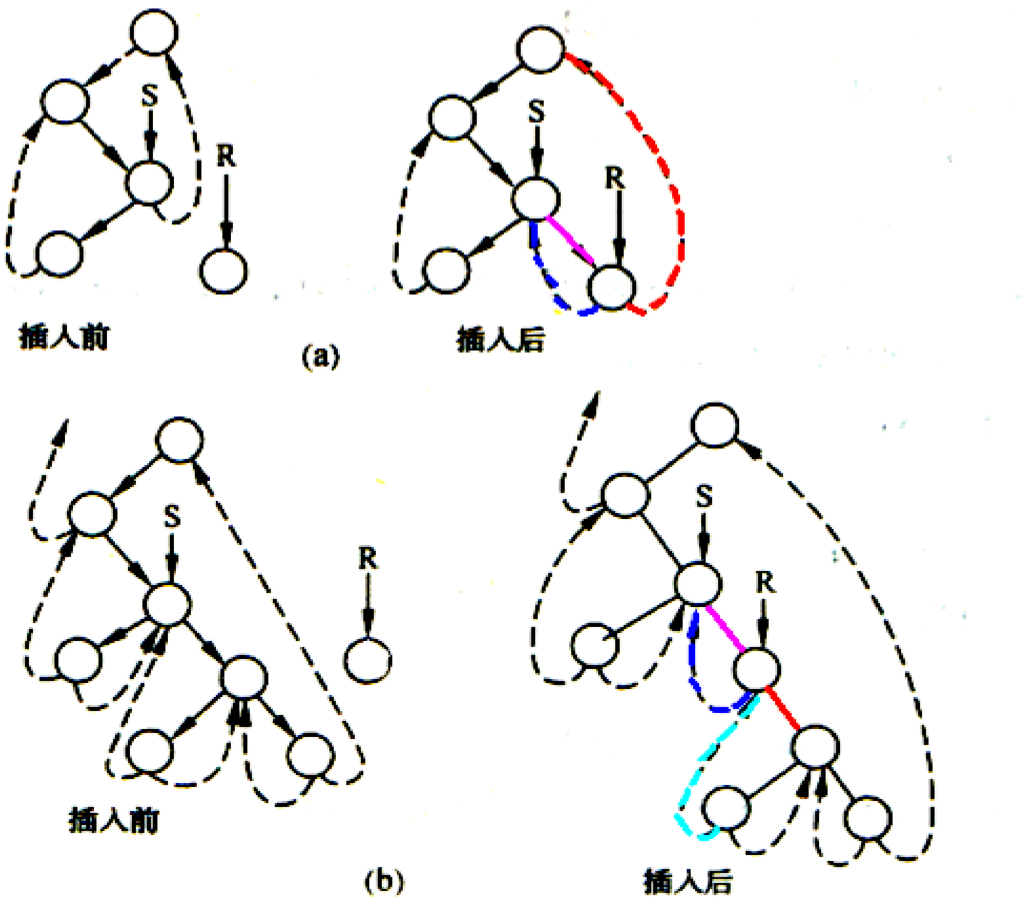
3.2 二叉树（Cont.）

线索二叉树的若干算法

- 算法4：中序线索二叉树的插入算法
- 分析：如将结点 R 插入作为结点 S 的右孩子结点。

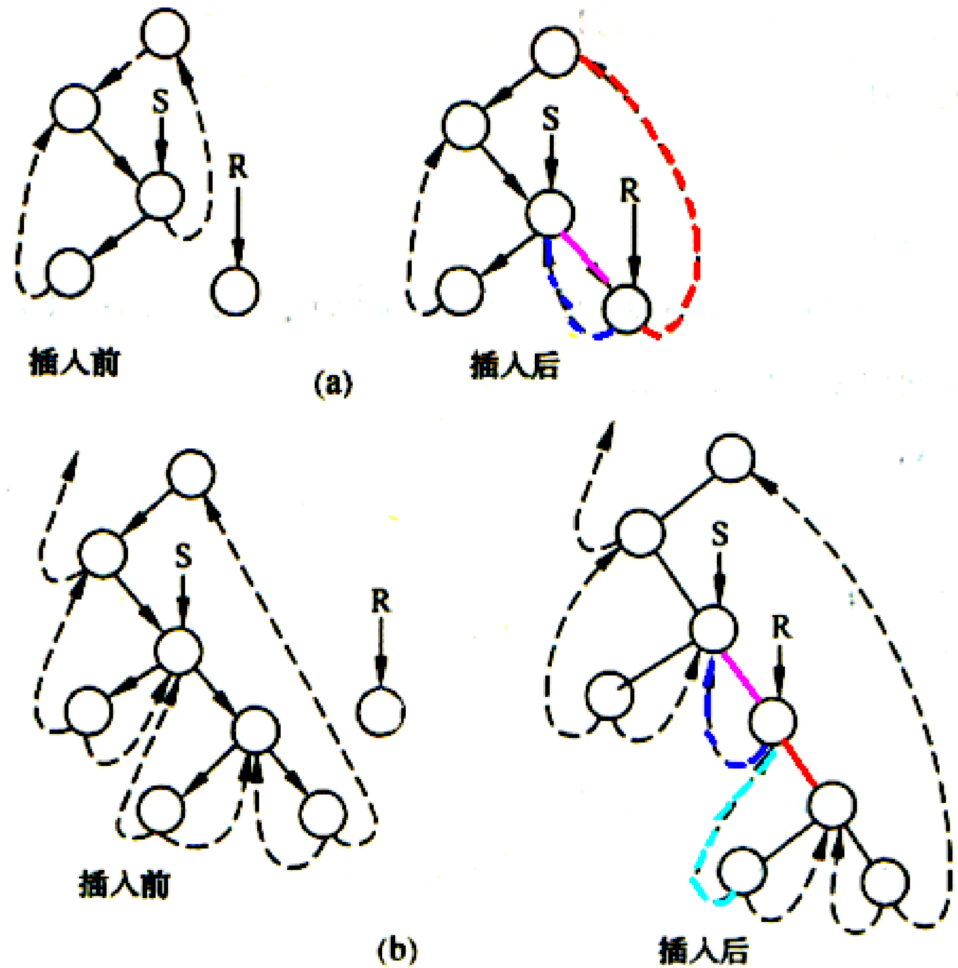
（1）若S的右子树为空，直接插入;

（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



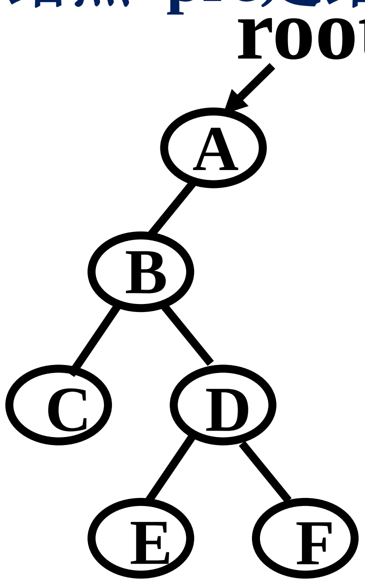
3.2 二叉树（Cont.）

```
void RInsert (ThTree S ,ThTree R)
{  ThTree   W ;
    R->rchild = S->rchild;
    R->rtag = S->rtag ;
    R->lchild = S ;//--
    R->ltag = False ;//--
    S->rchild = R ;
    S->rtag = True ;
    if (R->rtag==True) {
         w = InNext( R ) ;
         w->lchild = R ;  }
}
```

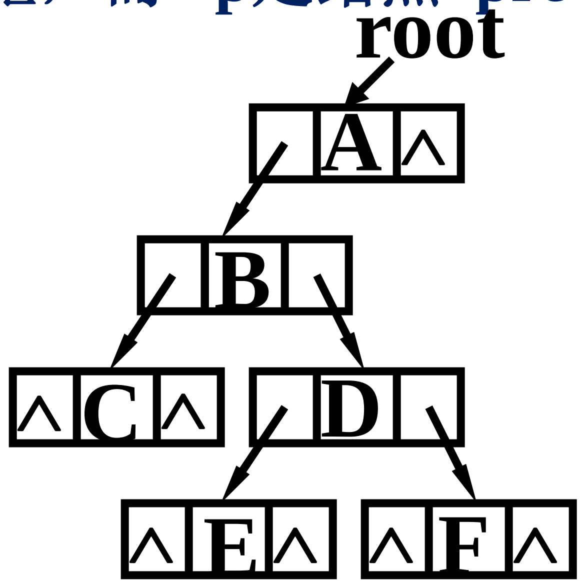


3.2 二叉树（Cont.）

- 算法5：二叉树的（中序）线索化算法--------递归算法
- 基本思想：
- 二叉树线索化，只要按某种次序遍历二叉树，在遍历过程中用线索取代空指针即可。为此，附设一个指针pre---始终指向刚刚访问过的结点，而指针p 指示当前正在访问的结点。显然，结点*pre是结点*p 的前驱，而 *p是结点*pre 的后继。



二叉树



动态二叉链表

3.2 二叉树（Cont.）

- 算法5：二叉树的（中序）线索化算法-------递归算法
- 实现步骤：

1 如果二叉链表root为空，则返回；否则，

2 对root的左子树建立线索；

3 对根结点root建立线索；

3.1 若root没有左孩子，则为root加上前驱线索;

3.2 若root没有右孩子，则将root右标志置为False；

3.3 若结点pre右标志为False，则为pre加上后继线索；

3.4 令pre指向刚刚访问的结点root;

4 对root的右子树建立线索。

3.2 二叉树（Cont.）

```
BTREE *pre=Null;  //全局量
void  InOrderTh(Btree *p) //将二叉树 p中序线索化
{   if( p ){ //p 非空时，当前访问的结点是 p
        InOrderTh( p->lchild ); //左子树线索化
        p->ltag=( p->lchild ) ? True : False; //左(右)孩子非空
        p->rtag=( p->rchild )? True : False; //时,标志1,否: 0
        if ( pre ) { //若*p 的前驱*pre 存在
            if ( pre->rtag ==False) //  *p的前驱右标志为线索
                pre->rchild=p; // 令 *pre 的右线索指向中序后继
            if ( p->ltag ==False) //   *p的左标志为线索
                p->lchild=pre;        //令 *p的左线索指向中序前驱
        }
        pre = p;   // 令pre 是下一个访问的中序前驱
        InOrderTh( p->rchild ); //右子树线索化
    }
}
```

3.2 二叉树（Cont.）

二叉树的复制

- 两株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.
- 相似二叉树:具有相同结构的二叉树为相似二叉树。
- 相似且对应结点包含相同信息的二叉树称为等价二叉树。
- 判断两株二叉树是否等价的算法:

```
struct  node  {
      struct  node  *lchild ;
      struct  node  *rchild ;
      datatype  data ;
};
typedef struct node  * Btree;
```

3.2 二叉树（Cont.）

```
int  Equal( Btree firstbt, Btree 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 */
```

3.2 二叉树（Cont.）

```
Btree  Copy( Btree  oldtree )
{  //二叉树的复制
   Btree   temp ;
   if ( oldtree != Null) {
        temp = new  Node ;
        temp -> data = oldtree->data ;
        temp -> lchild = Copy( oldtree->lchild ) ;
        temp -> rchild = Copy( oldtree->rchild ) ;
        return ( temp );
   }
   return ( Null ) ;
}  /* Copy */
```

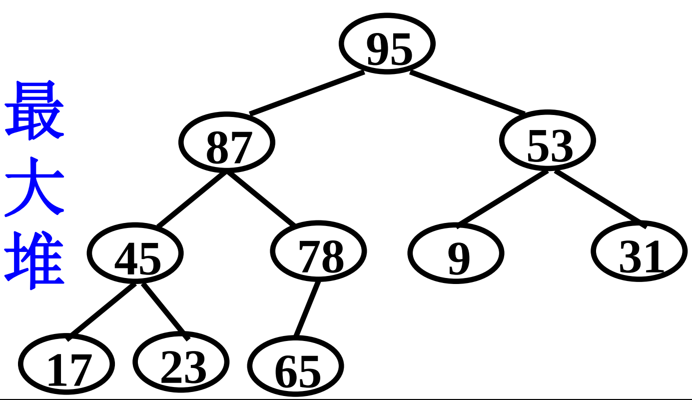
3.3 堆（Heap）

一、ADT堆

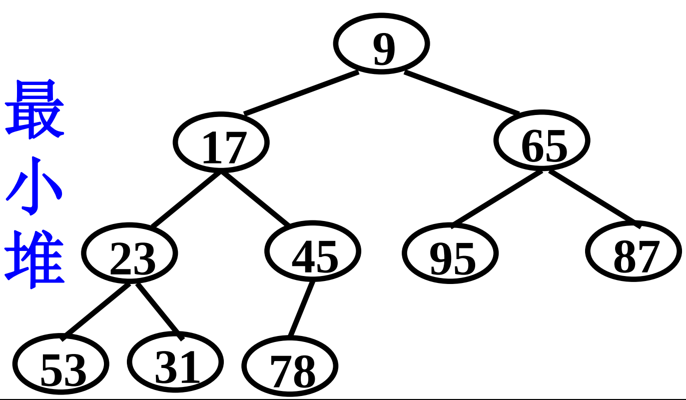
堆的定义

- 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（大顶堆、大根堆）。
- 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（小顶堆、小根堆）。
- 特点：根结点的元素是最大（小）的。。。。。。

最大堆



最小堆



3.3 堆（Heap）

一、ADT堆

ADT堆的基本操作

- **MaxHeap(maxsize)**：创建一个空堆，最多可容纳maxsize个元素
- **HeapFull(heap，n)**：判断堆是否为满。若堆中元素个数n达到最大容量maxsize，则返回TRUE，否则返回FALSE；
- **Insert(heap，item，n)**：插入一个元素。若堆不满，则将item 插入heap，否则不能插入；
- **HeapEmpty(heap，n)**：判断堆是否为空。若堆中元素个数n大于0，则返回TRUE，否则返回FALSE；
- **DeleteMax(heap，n)**：删除最大元素。若堆为不空，则返回堆中最大元素，并将其删除，否则，返回一个特定值，表明不能进行删除。

3.3 堆（Heap）

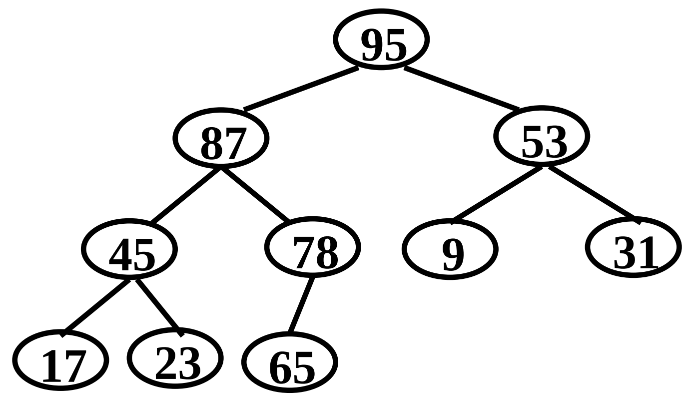
二、ADT堆的实现—最大堆的实现

ADT堆的存储结构

- 由于堆是一个完全二叉树，所以可以采用完全二叉树的数组表示。

堆的存储结构定义

```
#define Maxsize  200
typedef struct {
    int key;
    /* other fields*/
} ElemType;
typedef struct {
    ElemType data[Maxsize];
    int n;
} HEAP;
```



0	1	2	3	4	5	6	7	8	9	10				
	95	87	53	45	78	09	31	17	23	65				……

3.3 堆（Heap）

二、ADT堆的实现—最大堆的实现

堆的基本操作的实现

- ①创建空堆

```
void MaxHeap（HEAP heap）
{
        heap.n=0;
}
```

- ②判空

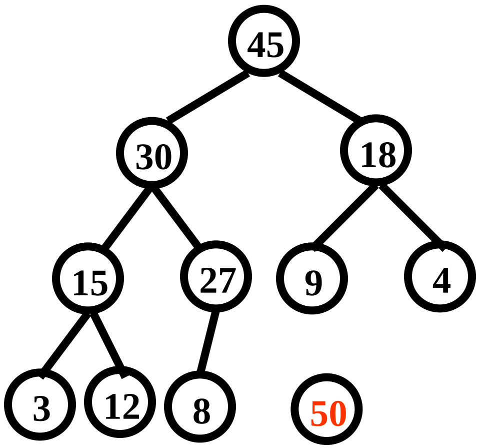
```
bool HeapEmpty（HEAP heap）
{
        return (!heap.n);
}
```

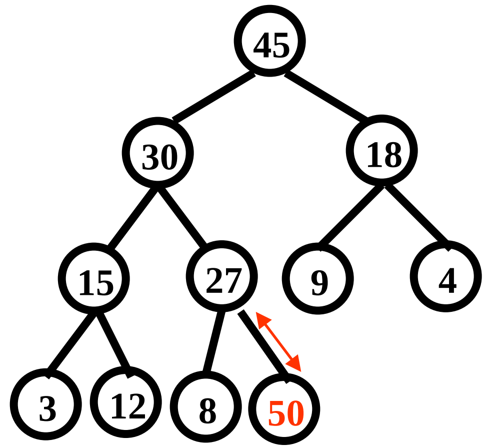
- ③判满

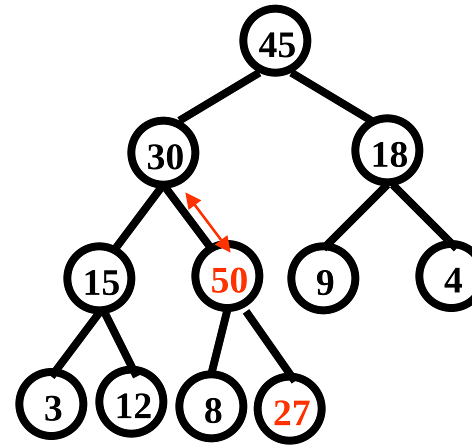
```
bool HeapFull（HEAP heap）
{
    return (heap.n==MaxSize);
}
```

```
#define Maxsize  200
  typedef struct {
      int key;
      /* other fields*/
  } ElemType;
  typedef struct {
      ElemType data[Maxsize];
      int n;
  } HEAP;
```

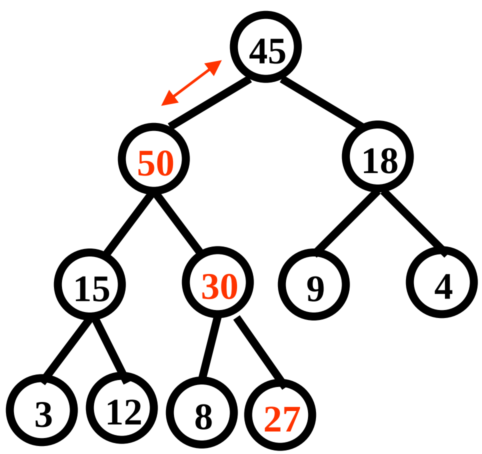
void Insert(HEAP& heap， ElemType elem)

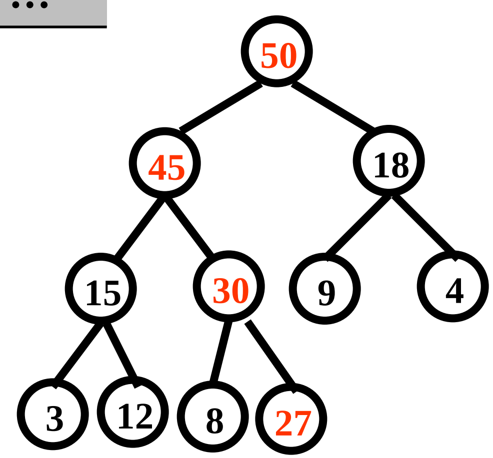






45	30	18	15	27	9	4	3	12	8	50	…





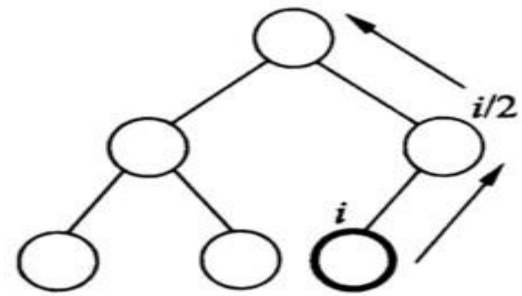
50	45	18	15	30	9	4	3	12	8	27	…

3.3 堆（Heap）

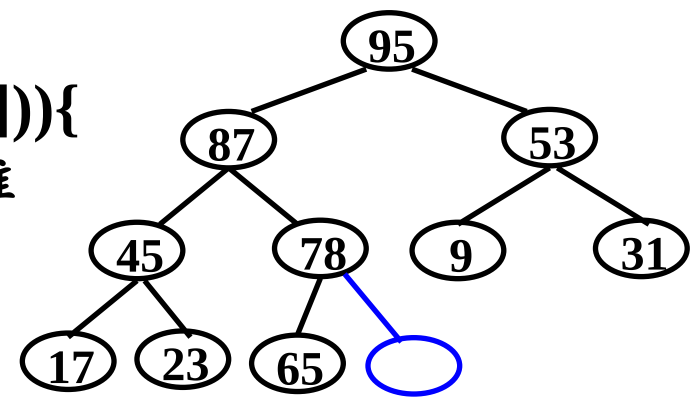
二、ADT堆的实现—最大堆的实现

堆的基本操作的实现

■ ④插入

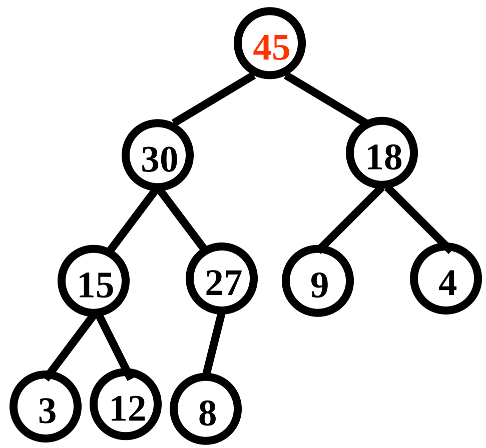


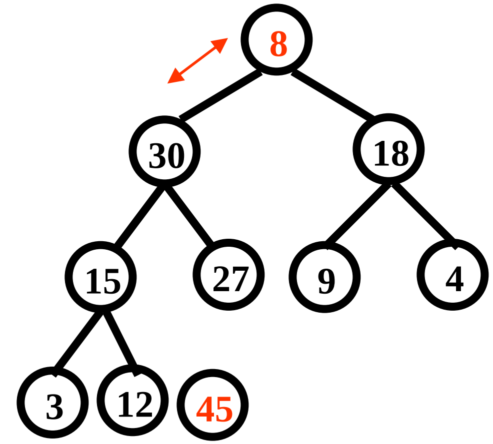

```
void Insert(HEAP& heap，ElemType elem)
{
    int i;
    if (!HeapFull(heap)){
        i=heap.n+1;
        while((i!=1)&&(elem >heap.data[i/2])){
            heap.data[i]=heap.data[i/2];//下推
            i/=2;
        }
    }
    heap.data[i]= elem;
    heap.n++
}//时间复杂性O（logn）
```



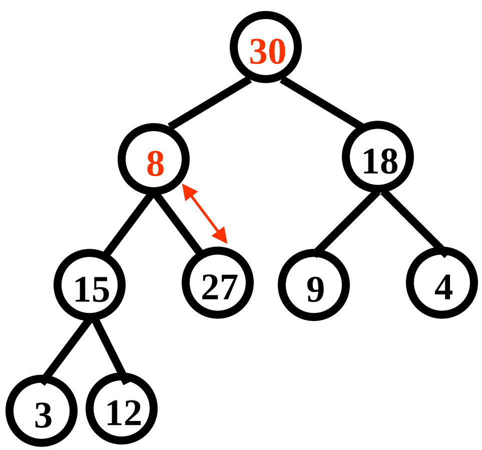
	95	87	53	45	78	09	31	17	23	65			
0	1	2	3	4	5	6	7	8	9	10	....		

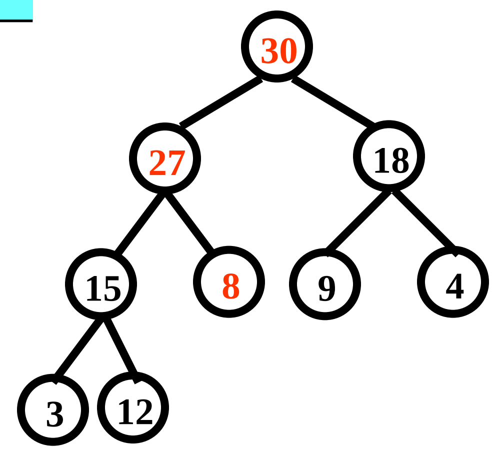
DeleteMax(HEAP heap)





45	30	18	15	27	9	4	3	12	8		…



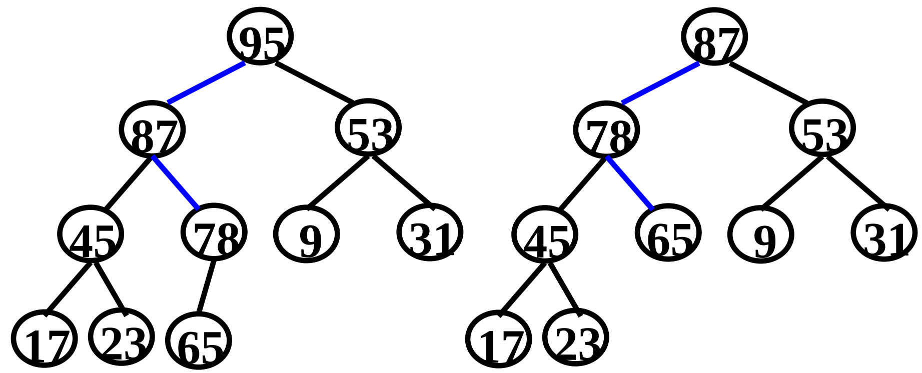


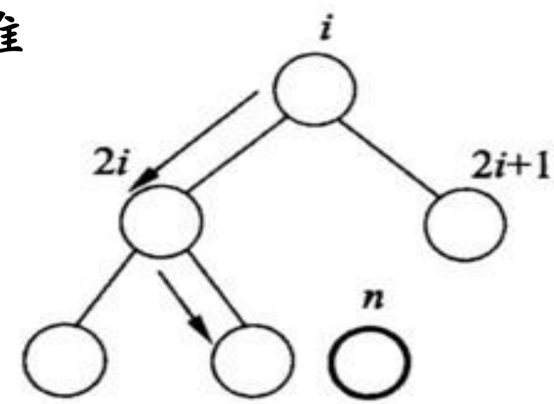
30	27	18	15	8	9	4	3	12	45		…

3.3 堆（Heap）

- ⑤删除最大元素

```
ElemType DeleteMax( HEAP &heap )
{   int parent=1, child=2;
    ElemType elem, tmp;
    if (!HeapEmpty(heap)){
       elem=heap.data[1];
       tmp=heap.data[heap.n--];
       while (child<=heap.n){
          if ((child< heap.n)&&
              (heap.data [child]<heap.data [child+1]) )
             child++; //找最大子结点（左右儿子的大者）
          if (tmp>= heap.data[child])  break;
          heap.data[parent]= heap.data[child];//上推
          parent=child;
          child*=2;
       }
       heap[parent]=tmp;
       return elem;
    }
} //时间复杂性O（logn）
```





经常使用堆来实现优先级队列（priority queue）。与第二章所讨论的队列不同的是，优先级队列只对最高（或最低）优先级的元素进行删除。但是在任何时候，都可以把任意优先级的元素插入到优先级队列。

操作系统中的进程管理是优先级队列的一个应用实例，系统中使用一个优先队列来管理进程。

每个进程有进程任务号和优先级两部分组成。当有多个进程排队时，优先级高的先操作。

练习题：设计一个程序模仿操作系统的进程管理问题，进程服务按优先级高的先服务，同优先级的先到先服务的管理原则。设文件task.dat中存放了仿真进程服务请求，其中第一列是进程任务号，第二列是进程的优先级。

1 30
2 20
3 40
4 20
5 0

算法：1）建立队列
2）建堆
3）循环出队，输出。