6.3遍历二叉树和线索二叉树

6.3.1 遍历二叉树

一、问题的提出

二、先左后右的遍历算法

三、算法的递归描述

四、中序遍历算法的非递归描述

五、遍历算法的应用举例

一、问题的提出

何谓二叉树的遍历？

顺着某一条搜索路径**巡访**二叉树中的结点，使得每个结点**均被访问一次**，而且**仅被访问一次。**

“**访问**”的含义可以很广，如：输出结点的信息等。

“遍历”是任何类型均有的操作，对线性结构而言，只有一条搜索路径(因为每个结点均只有一个后继)，故不需要另加讨论。而二叉树是非线性结构，每个结点有两个后继，则存在如何遍历即按什么样的搜索路径遍历的问题。

对“二叉树”而言，可以有三条搜索路径：

1．**先上后下**的按层次遍历；

2．**先左**（子树）**后右**（子树）的遍历；

3．**先右**（子树）**后左**（子树）的遍历。

二、先左后右的遍历算法

二叉树是由三个基本单元组成：根结点、左子树和右子树。假设以L、D、R分别表示遍历左子树、访问根结点、遍历右子树。若规定先左后右，则有三种情况：DLR、LDR、LRD，分别称之为**先**（根）序的遍历、**中**（根）序的遍历、**后**（根）序的遍历。

先（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）访问根结点；

（2）先序遍历左子树；（递归）

（3）先序遍历右子树。（递归）

中（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）中序遍历左子树；（递归）

（2）访问根结点；

（3）中序遍历右子树。（递归）

后（根）序的遍历算法：

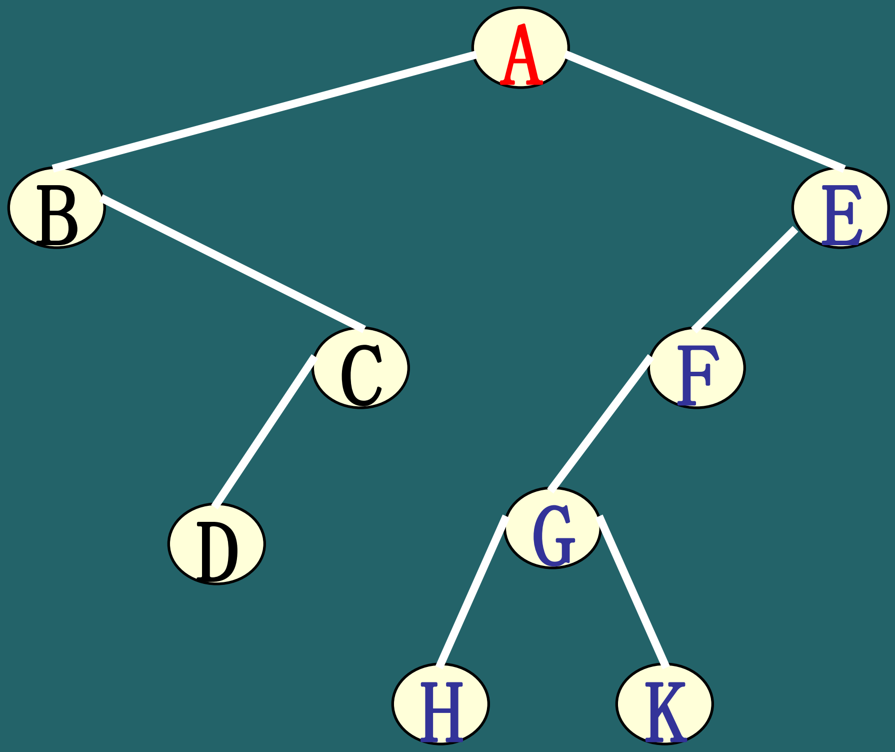
若二叉树为空树，则空操作；否则，

（1）后序遍历左子树；（递归）

（2）后序遍历右子树；（递归）

（3）访问根结点。

例如：对下面二叉树的各种遍历结果如下：



前序遍历结果：

A B C D E F G H K

中序遍历结果：

B D C A H G K F E

后序遍历结果：

D C B H K G F E A

```
Status PreOrderTraverse( BiTreee T, Status ( * Visit)(TElemType e) )
   {
 //采用二叉链表存储结构， Visit是对数据元素操作的应用函数，先序遍
   //历二叉树T的递归算法，对每个数据元素调用函数Visit。
 //调用实例： PreOrderTraverse( T, PrintElement) ;
if (T) {
  if (Visit(T->data ))
    if (PreOrderTraverse(T->lchild, Visit))
      if (PreOrderTraverse(T->rchild, Visit))     return OK;
   return ERROR ;
 } else  return OK ;
}// PreOrderTraverse
```

算法 6.1

最简单的Visit函数是：

```
Status PrintElement(TElemType e){
//输出元素e的值
      printf( e ) ;                    //实用时，加上格式串
      return OK ;
  }
```

从中序遍历递归算法执行过程中递归工作栈的状态可见：

（1）工作记录中包含两项：其一是递归调用的语句编号，其二是指向根结点的指针，则当栈顶记录中的指针非空时，应遍历左子树，即指向左子树根的指针进栈；

（2）若栈顶记录中的指针值为空，则应退至上一层，若是从左子树返回，则应访问当前层即栈顶记录中指针所指的根结点；

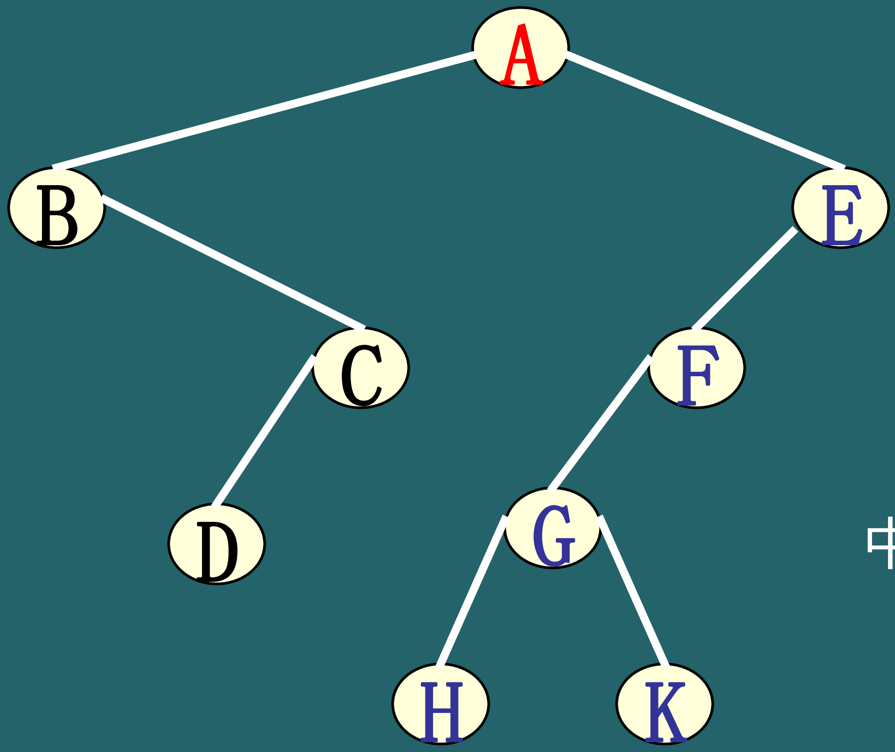
（3）若是从右子树返回，则表明当前层的遍历结束，应继续退栈。从另一角度看，这意味着遍历右子树时不再需要保存当前层的根指针，可直接修改栈顶记录中的指针即可。

由此得两个中序遍历二叉树的非递归算法如算法6.2 和 6.3所示。

```
Status InOrderTraverse( BiTreee T, Status ( * Visit)(TElemType e) ) {
  //采用二叉链表存储结构， Visit是对数据元素操作的应用函数。中序
  //遍历二叉树T的非递归算法，对每个数据元素调用函数Visit。
  InitStack(S)； Push(S，T)；  //根指针进栈
  while (! StackEmpty(S)) {
    while (GetTop(S,p)) && p) Push(S,p->lchild)；
        //向左走到尽头
    Pop(S,p)；     //空指针退栈
    if (!StackEmpty(S)) { //访问结点，向右一步
      Pop(S,p)； if( !Visit(p->data)) return ERROR；
      Push(S，p->rchild)；
     }//if
   }//While
  return OK；
}// InOrderTraverse
```

算法 6.2

例如：对下面二叉树的各种遍历结果如下：



中序遍历结果：

B D C A H G K F E

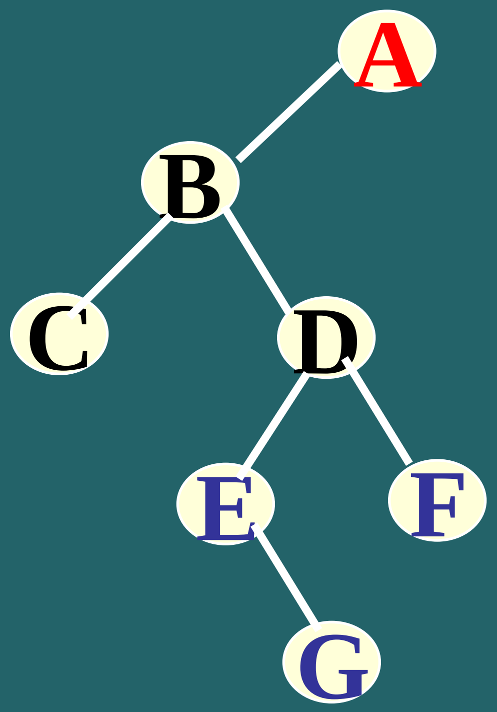
```
Status InOrderTraverse( BiTreee T, Status ( * Visit)(TElemType e) ) {
    //采用二叉链表存储结构， Visit是对数据元素操作的应用函数。中序
    //遍历二叉树T的非递归算法，对每个数据元素调 用函数Visit。
    InitStack(S)； p=T；
    while (p|| !StackEmpty(S)) {
        if (p) {Push(S，p)； p=p->lchild)；
       //根指针进栈，遍历左子树
        else{    //根指针退栈，访问根结点，遍历右子树
            Pop(S，p)；  if( !Visit(p->data)) return ERROR；
      p=p->rchild；
     }//else
     }//While
  return OK；
}// InOrderTraverse
```

算法 6.3

‘遍历’是二叉树各种操作的基础，也可在遍历过程中生成结点，建立二叉树的存储结构。例如：对下面所示二叉树，按下列次序顺序读入字符

A B C Ø Ø D E Ø G Ø Ø F Ø Ø Ø

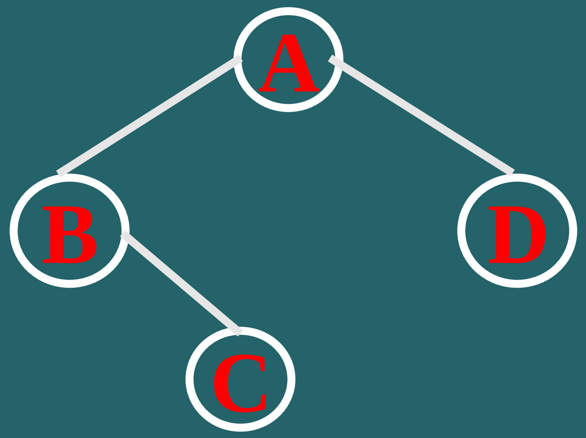
其中 Ø 表示空格字符，可建立相应的二叉链表。



例如:

空树 以字符“Ø”表示

只含一个根结点的二叉树 以字符串“A Ø Ø”表示



以下列字符串表示

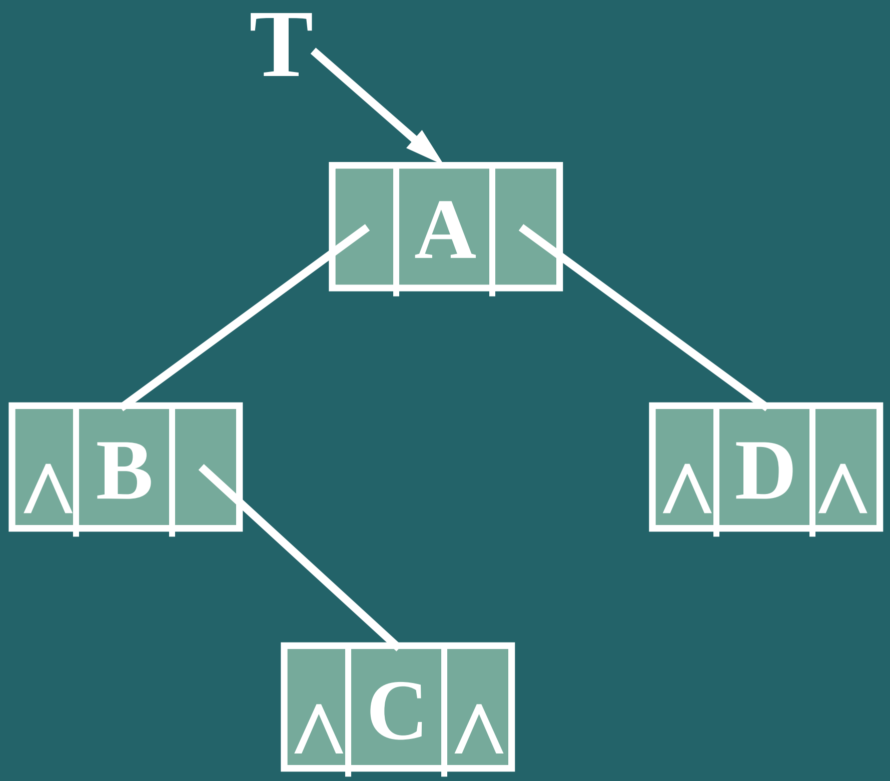
$$A(\overbrace{B(\underbrace{\varnothing}_{BL},\underbrace{C(\underbrace{\varnothing}_{CL}, \underbrace{\varnothing}_{CR})}_{BR})}^{AL},\overbrace{D(\underbrace{\varnothing}_{DL}, \underbrace{\varnothing}_{DR})}^{AR})$$

```
Status CreateBiTree( BiTreee &T ) {
    //按先序次序输入二叉树中结点的值（一个字符），空 字符表示空树
    //构造二叉链表表示的二叉树T。
    scanf(&ch)；
  if (ch== ‘ ’ ) T=NULL；
    else{
    if( !(T=(BiTNode * )malloc(sizeof(BiTNode))))  exit(OVERFLOW)；
    T->data=ch；                 //生成根结点
  CreateBiTree(T->lchild)；    //构造左子树
  CreateBiTree(T->rchild)；    //构造右子树
  }
  return OK；
}// CreateBiTree
```

算法 6.4

上述算法执行过程举例如下：

A B C D



由二叉树的先序和中序序列建二叉树

仅知一棵二叉树的先序序列 不能唯一确定该二叉树；

如果同时已知该二叉树的中序序列，则会如何呢？

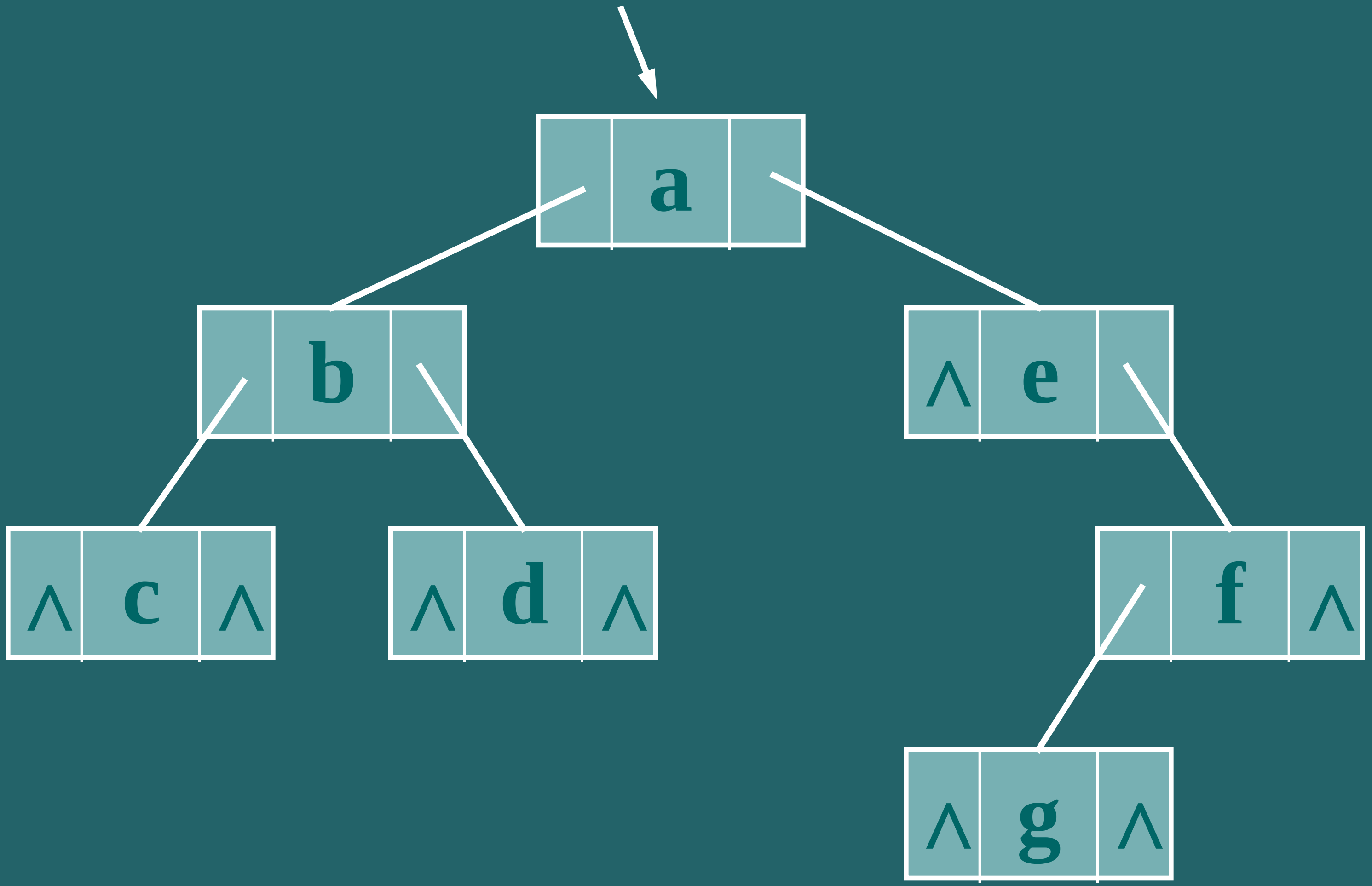
二叉树的先序序列　根　左子树　右子树

二叉树的中序序列　左子树　根　右子树

例如:

a b c d e f g 先序序列

c b d a e g f 中序序列



```
void CrtBT(BiTree& T, char pre[], char ino[],
                                    int ps, int is, int n ) {
  // 已知pre[ps..ps+n-1]为二叉树的先序序列，
  // ins[is..is+n-1]为二叉树的中序序列，本算
  // 法由此两个序列构造二叉链表
   if (n==0) T=NULL;
   else {
      k=Search(ino, pre[ps]); // 在中序序列中查询
 //根结点
      if (k== -1)  T=NULL;
      else {   … … }
   } //
} // CrtBT
```

```
T=(BiTNode*)malloc(sizeof(BiTNode));
T->data = pre[ps];
if (k==is)  T->Lchild = NULL;
else  CrtBT(T->Lchild, pre[], ino[],
                              ps+1, is, k-is );
if (k=is+n-1) T->Rchild = NULL;
else  CrtBT(T->Rchild, pre[], ino[],
                   ps+1+(k-is), k+1, n-(k-is)-1 );
```

三、算法的递归描述

1、二叉树的前序遍历

```
void Preorder (BiTree T,
              void( *visit)(TElemType& e))
{ // 先序遍历二叉树
  if (T) {
    visit(T->data);          // 访问结点
    Preorder(T->lchild, visit); //前序遍历左子树
    Preorder(T->rchild, visit);//前序遍历右子树
  }
}
```

2、二叉树的中序遍历

```
void Inorder (BiTree T,
                  void( *visit)(TElemType& e))
{ // 中序遍历二叉树
  if (T) {
    Inorder(T->lchild, visit); // 中序遍历左子树
    visit(T->data);            // 访问结点
    Inorder(T->rchild, visit);// 中序遍历右子树
  }
}
```

五、遍历算法的应用举例

1、统计二叉树中叶子结点的个数 (先序遍历)

2、求二叉树的深度(后序遍历)

1、统计二叉树中叶子结点的个数

算法基本思想:

先序（或中序或后序）遍历二叉树，在遍历过程中查找叶子结点，并计数。

由此，**需在遍历算法中增添一个“计数”的参数**，并将算法中“访问结点”的操作改为：**若是叶子，则计数器增1。**

```
void CountLeaf (BiTree T,  int& count){
//统计二叉树中叶子结点的个数，用count返回
  if ( T ) {
    if ((!T->lchild)&& (!T->rchild))
      count++;     // 对叶子结点计数
    CountLeaf( T->lchild, count);
    CountLeaf( T->rchild, count);
  } // if
} // CountLeaf
```

2、求二叉树的深度(后序遍历)

算法基本思想:

首先分析**二叉树的深度**和它的**左、右子树深度**之间的关系。

从二叉树深度的定义可知，**二叉树的深度应为其左、右子树深度的最大值加1**。由此，**需先分别求得左、右子树的深度**，算法中“访问结点”的操作为：**求得左、右子树深度的最大值，然后加 1 。**

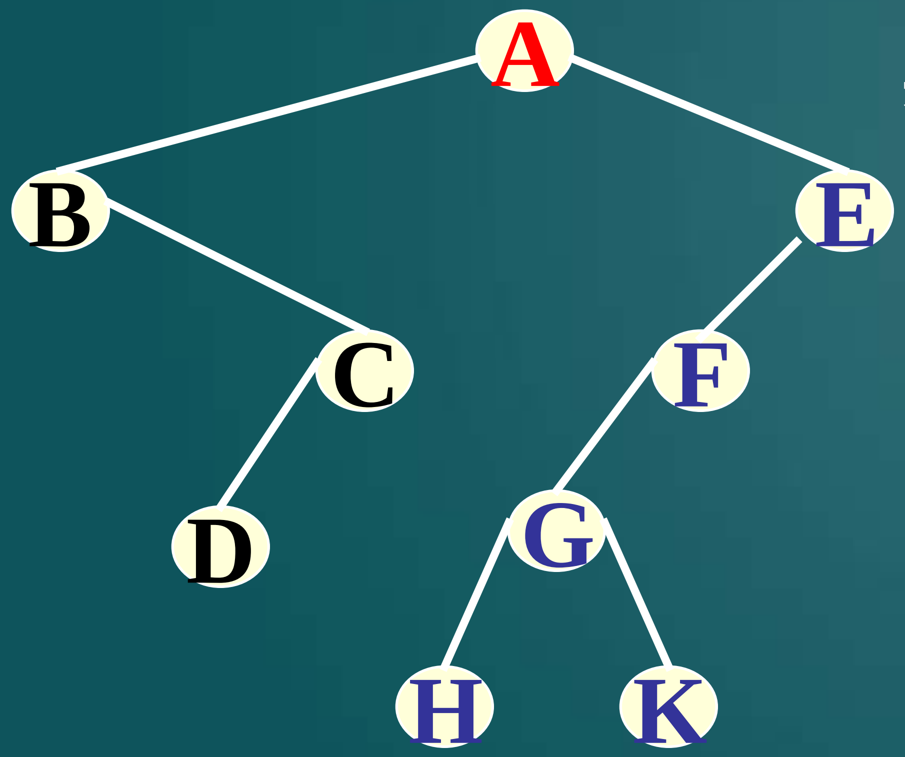
```
int Depth (BiTree T ){ // 返回二叉树的深度
   if ( !T )    depthval = 0;
   else   {
     depthLeft = Depth( T->lchild );
     depthRight= Depth( T->rchild );
     depthval = 1 + (depthLeft > depthRight  ?
                              depthLeft : depthRight);
   }
   return depthval;
}
```

6.3.2 线索二叉树

一、何谓线索二叉树？

遍历二叉树的结果是，求得结点的一个线性序列。

例如:



先序序列:

A B C D E F G H K

中序序列:

B D C A H G K F E

后序序列:

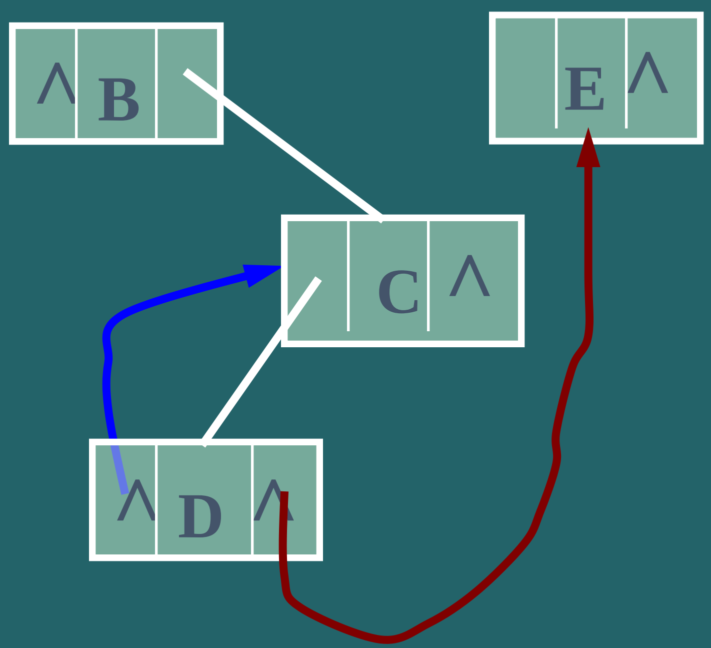
D C B H K G F E A

在以二叉链表作为存储结构时，只能找到结点的左、右结点，但是得不到结点在任一序列中的前驱和后继结点，这种信息只能在遍历的过程中才能得到。

如何保存这种在遍历过程中得到的信息呢？可以利用二叉链表中的n+1个空链来存放结点的前驱和后继结点的信息。

- 指向该线性序列中的“前驱”和 “后继” 的**指针**，称作 “**线索**”

例如：A B C D E F G H K



- 包含 “线索” 的存储结构，称作 “**线索链表**”

- 与其相应的二叉树，称作 “**线索二叉树**”

对线索链表中结点的约定：

在二叉链表的结点中**增加两个标志域**LTag 和Rtag，并作如下规定：

- **若该结点的左子树不空，**

  则Lchild域的指针指向其左子树，

  且左标志域 LTag的值为 0 “指针 Link” ；

  否则，Lchild域的指针指向其“前驱”，

  且左标志LTag的值为 1 “线索 Thread” 。

- **若该结点的右子树不空，**

  则rchild域的指针指向其右子树，

  　且右标志域RTag的值为 0 “指针 Link” ；

  否则，rchild域的指针指向其“后继” ，

  　且右标志RTag的值为 1 “线索 Thread” 。

如此定义的二叉树的存储结构称作“线索链表”。以某种次序遍历使其变为线索二叉树的过程叫做‘线索化’。

线索链表的结点结构：

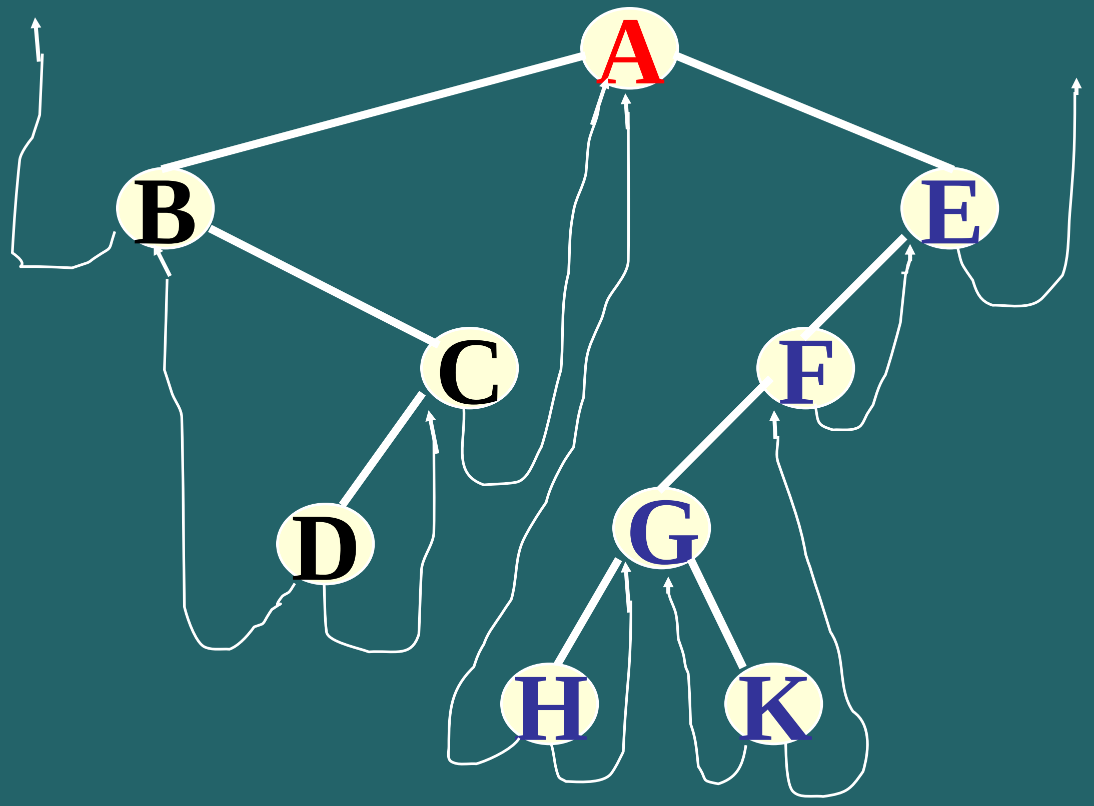
lchild	LTag	data	RTag	rchild

其中：

$$Ltag=\begin{cases}0 & \text{lchild域指示结点的左孩子}\\ 1 & \text{lchild域指示结点的前驱}\end{cases}$$

$$Rtag=\begin{cases}0 & \text{rchild域指示结点的右孩子}\\ 1 & \text{rchild域指示结点的后继}\end{cases}$$

例如，下图为一中序线索二叉树



线索链表的类型描述：

```
typedef enum { Link, Thread } PointerThr ;
  // Link=0：指针，Thread=1：线索
typedef struct BiThrNod {
   TElemType        data；
   struct BiThrNode  *lchild, *rchild;； // 左右指针
   PointerThr         LTag, RTag；   // 左右标志
} BiThrNode，*BiThrTree；
```

二、线索链表的遍历算法:

由于在线索链表中添加了遍历中得到的“前驱”和“后继”的信息，从而简化了遍历的算法。

```
for ( p = firstNode(T) ; p ; p = Succ(p) )
    Visit (p) ;
```

例如:

对中序线索化链表的遍历算法

※ 中序遍历的第一个结点 ？

左子树上处于“**最左下**”（没有左子树）的结点。

※ 在中序线索化链表中结点的后继 ？

若无右子树，**则为后继线索**所指结点；

否则为对其**右子树**进行中序**遍历**时访问的**第一个结点。**

具体算法见 算法 6.5

```
void InOrderTraverse_Thr(BiThrTree T,
                         void (*Visit)(TElemType e)) {
  p = T->lchild;       // p指向根结点
  while (p != T) {     // 空树或遍历结束时，p=T
    while (p->LTag==Link)  p = p->lchild;
    if (!Visit(p->data))  return error  //访问其左子树为空的结点
    while (p->RTag==Thread && p->rchild!=T) {
      p = p->rchild;  Visit(p->data);      // 访问后继结点
    }
    p = p->rchild;           // p进至其右子树根
  }
} // InOrderTraverse_Thr
```

算法 6.5

三、如何建立线索链表？

在中序遍历过程中修改结点的左、右指针域，以保存当前访问结点的“前驱”和“后继”信息。遍历过程中，附设指针pre 始终指向刚刚访问过的结点， 即，若指针 p 指向当前访问的结点，则pre 指向它的前驱。

```
Status InOrderThreading(BiThrTree &Thrt , BiThrTree T)
 { // 中序遍历二叉树T，并将其中序线索化，Thrt指向头结点
   if (!(Thrt = (BiThrTree)malloc(sizeof( BiThrNode))))
      exit (OVERFLOW) ;
   Thrt->LTag = Link ;  Thrt->RTag =Thread ;  //建头结点
   Thrt->rchild = Thrt ;      // 右指针回指
   if (!T)  Thrt->lchild = Thrt ;   //若二叉树空,则左指针回指
   else {  Thrt->lchild = T ;    pre = Thrt ;
           InThreading(T) ;    //中序遍历进行线索化
           pre->rchild = Thrt ;    // 最后一个结点线索化
           pre->RTag = Thread ;
          Thrt->rchild = pre ;     }
  return OK ;
} // InOrderThreading
```

算法 6.6

```
void InThreading(BiThrTree p) {
  if (p) {    // 对以p为根的非空二叉树进行线索化
    InThreading(p->lchild)；     // 左子树线索化
    if (!p->lchild)      // 建前驱线索
      { p->LTag = Thread；    p->lchild = pre; }
    if (!pre->rchild)   // 建后继线索
      { pre->RTag = Thread；   pre->rchild = p； }
    pre = p；            // 保持 pre 指向 p 的前驱
    InThreading(p->rchild)；     // 右子树线索化
  } // if
} // InThreading
```

算法 6.7