几种非常有用的、有代表性的二叉树

6.线索二叉树* （Threaded Binary Tree）

7.二叉查找树 （ Binary Search Tree, BST）

8.平衡二叉树或AVL树 (Balanced Binary Tree)

9.堆 (Heap)

10.哈夫曼树 (Huffman Tree)

问题5.1：词频统计

在“线性表”一章中给出了两种方案：

- 基于顺序表（数组）
  - 优点：单词查找效率高（可用折半查找）
  - 缺点：
    1. 单词表长度需事先定义，易造成空间浪费或不足
    2. 插入操作效率低（需要移动大量数据）
- 基于链表
  - 优点：空间动态管理（随问题规模动态改变），程序具有可扩展性；单词插入不需要移动数据，效率高
  - 缺点：单词查找效率低（每次要从头开始查找）

有没有一种方法能兼顾两者的优点？

7 二叉查找树(二叉搜索树、二叉排序树)

7.1 基本概念

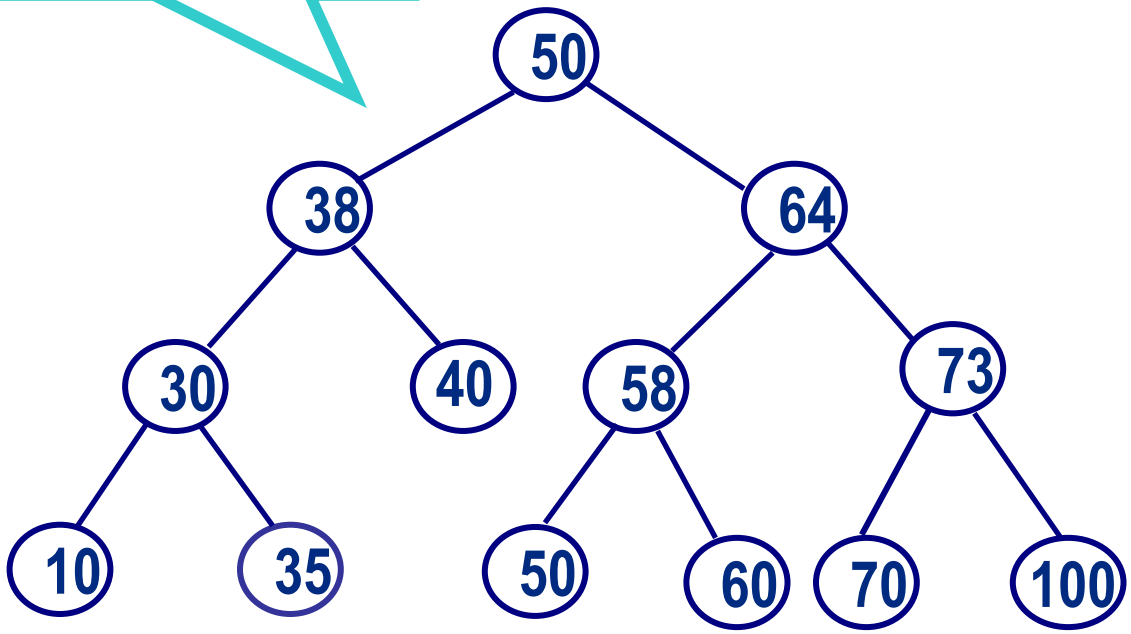
二叉查找树或者为空二叉树，或者为具有以下性质的二叉树：

- 若根结点的左子树不空，则左子树上所有结点的值都小于根结点的值；
- 若根结点的右子树不空，则右子树上所有结点的值都大于或等于根结点的值；

每一棵子树分别也是二叉查找树。

递归定义

一棵二叉查找树



中序序列:

10, 30, 35, 38, 40, 50, 50, 58, 60, 64, 70, 73, 100

7.2 二叉查找树的查找

1. 查找过程

若二叉查找树为空，则查找失败，查找结束。

若二叉查找树非空，则将被查找元素与二叉排序树的根结点的值进行比较，

- 若等于根结点的值，则查找成功，结束；
- 若小于根结点的值，则到根结点的左子树中重复上述查找过程；
- 若大于根结点的值，则到根结点的右子树中重复上述查找过程；

直到查找成功或者失败。

递归过程

约定

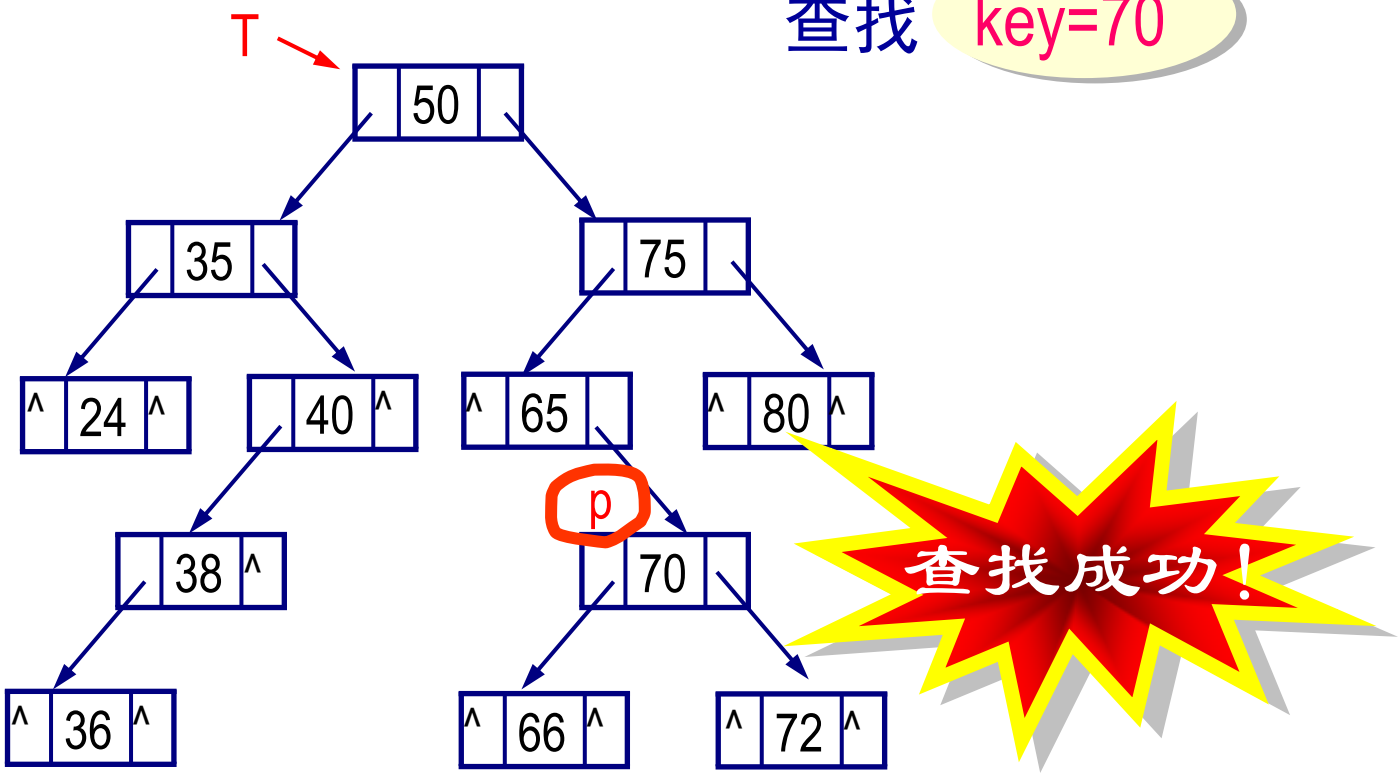
若查找成功，返回被查找元素所在结点的地址；

若查找失败，返回NULL。

二叉树采用二叉链式存储结构

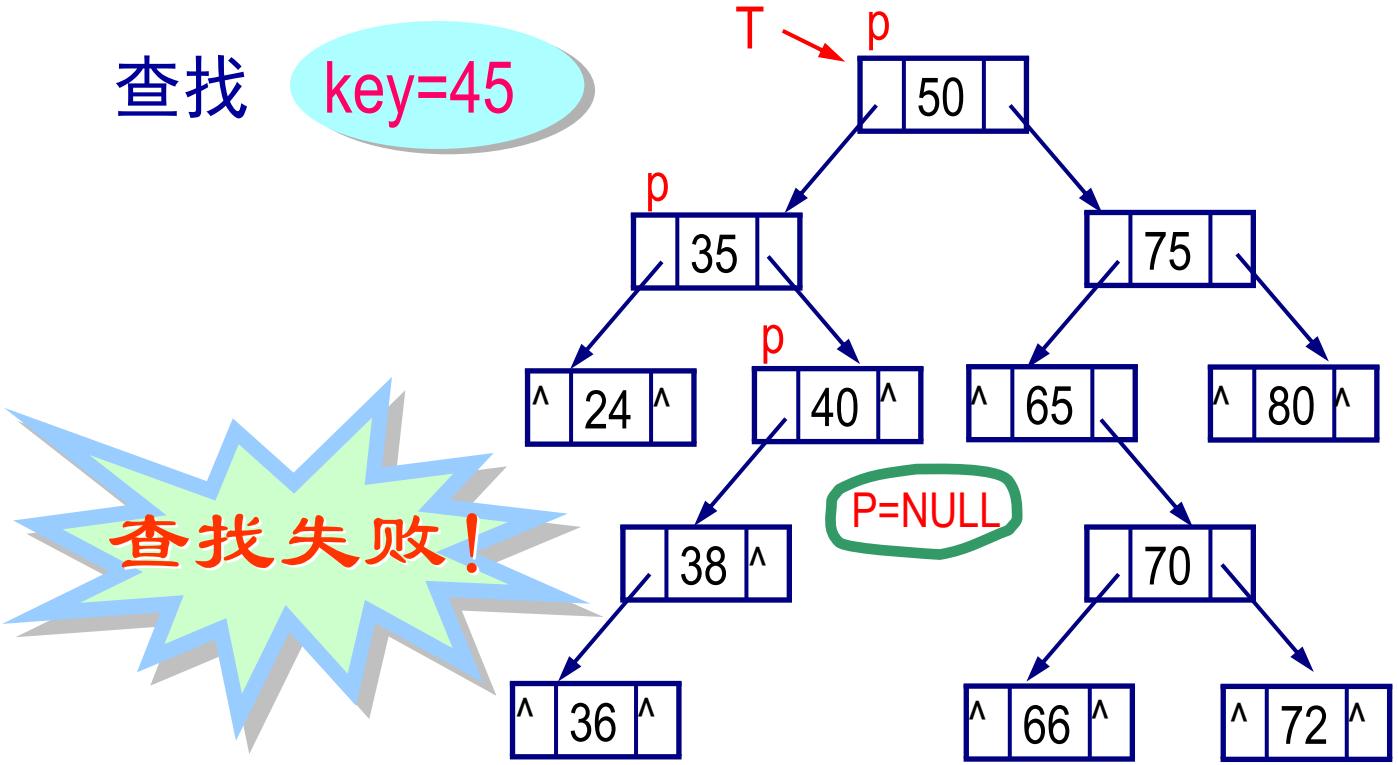
例

查找 key=70



例

查找 key=45



查找失败！

无论查找成功或失败，搜索“轨迹”限定在一条路径上

==>查找的效率与二叉查找树的深度有关。

2. 查找算法


```
BTNodeptr  searchBST( BTNodeptr t, Datatype key ){
    if(t==NULL)
        return NULL;       /* 查找失败 */

    if(key == t->data)     /* 查找成功 */
        return t;

    if(key > t->data)      /* 查找T的右子树 */
        return searchBST(t->rchild, key);
    else                   /* 查找T的左子树 */
        return searchBST(t->lchild, key);
}
```

2. 查找算法

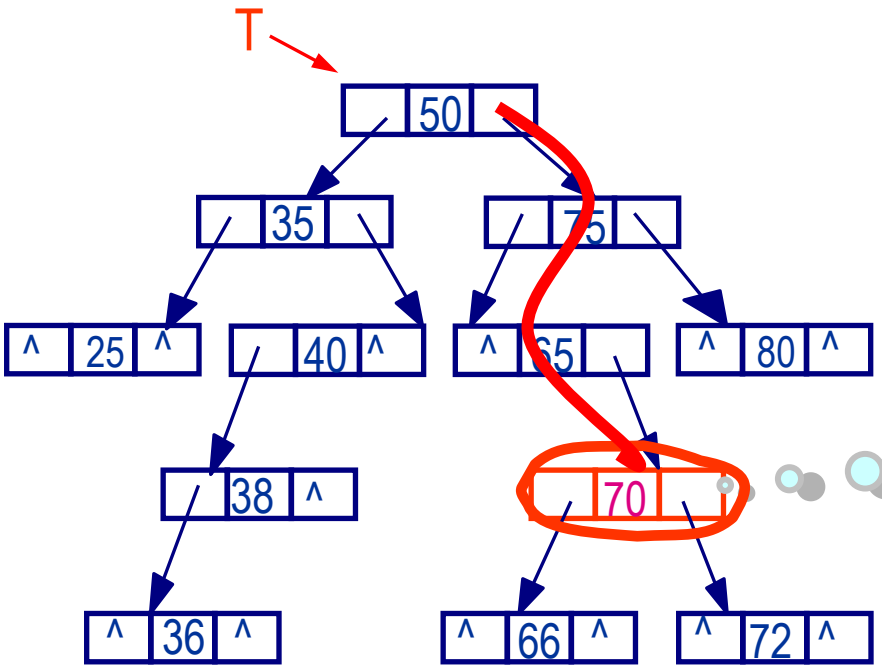
非递归算法

```
BTNodeptr  searchBST(BTNodeptr t, Datatype  key){
    BTNodeptr  p=t;
    while(p!=NULL){
        if(key == p->data)
            return p;          /* 查找成功 */
        if(key > p->data)
            p=p->rchild;       /* 将p移到右子树的根结点 */
        else
            p=p->lchild;       /* 将p移到左子树的根结点 */
     }
     return NULL;              /* 查找失败 */
}
```

例

已知二叉查找树采用二叉链表存储结构，根结点地址为T，请写一非递归算法，打印数据信息为item的结点的所有祖先结点。
（设该结点存在祖先结点）

从根结点到该结点的所有分支上经过的结点



祖先结点:
50, 75, 65

该查找过程同样限定在一条路径上，无需回溯！

```
void  searchBST(BTNodeptr t, Datatype item){
    BTNodeptr  p=t;
    while(p!=NULL){
        if(item == p->data)
            break;                  /* 查找结束 */
        printf("%d ",p->data);  // 最好先记录，最后一起输出
        if(item > p->data)
            p=p->rchild;          /* 将p 移到右子树的根结点 */
        else
            p=p->lchild;          /* 将p 移到左子树的根结点 */
    }
}
```

3. 查找效率

平均查找长度ASL —— 确定一个元素在树中位置所需要进行的元素间的比较次数的期望值(平均值)。

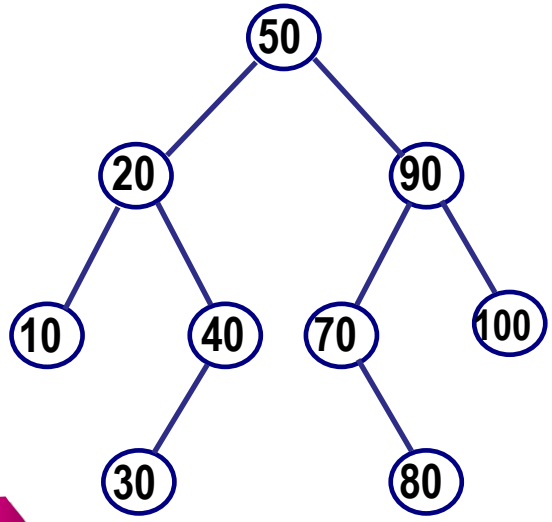
$$ASL=\sum_{i=1}^{n} p_i c_i$$

n　二叉树中结点的总数；

p_i 表示查找第i个元素的概率；

c_i 表示查找第i个元素需要进行的元素之间的比较次数。

注： 查找成功的比较次数等于该结点所处的层次数



例

ASL=(1×1+2×2+3×4+4×2)/9
=(1+4+12+8)/9
=25/9

7.3 二叉查找树的建立 (*逐点插入法*)

设K=(k_1, k_2, k_3, … , k_n)为具有n个数据元素的序列。从序列的第一个元素开始，，依次取序列中的元素，每取一个元素k_i，按照下述原则将k_i插入到二叉树中：

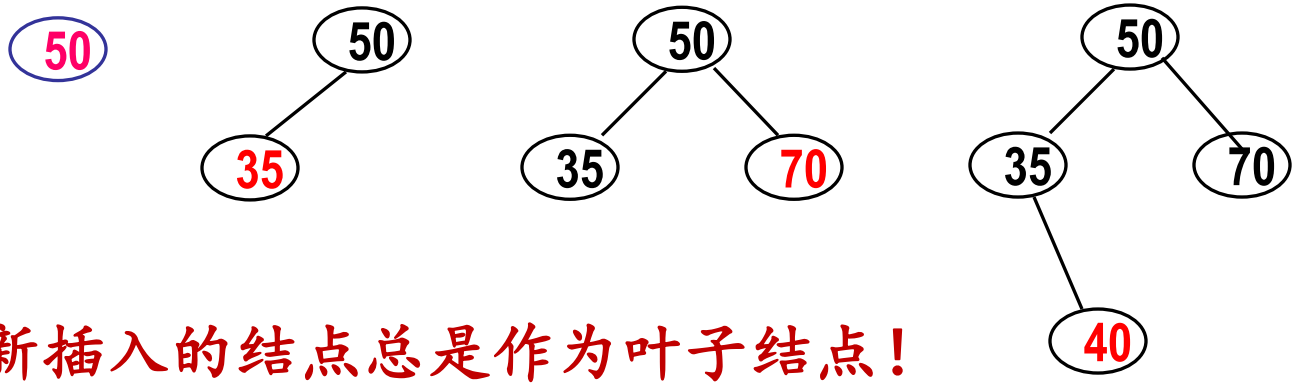
1. 若**二叉树**为空， 则k_i作为该二叉树的根结点；
2. 若**二叉树**非空，则将k_i与该二叉树的根结点的值进行比较，
   ——若k_i小于根结点的值，则将k_i插入到根结点的左子树中；
   ——否则，将k_i插入到根结点的右子树中。

递归

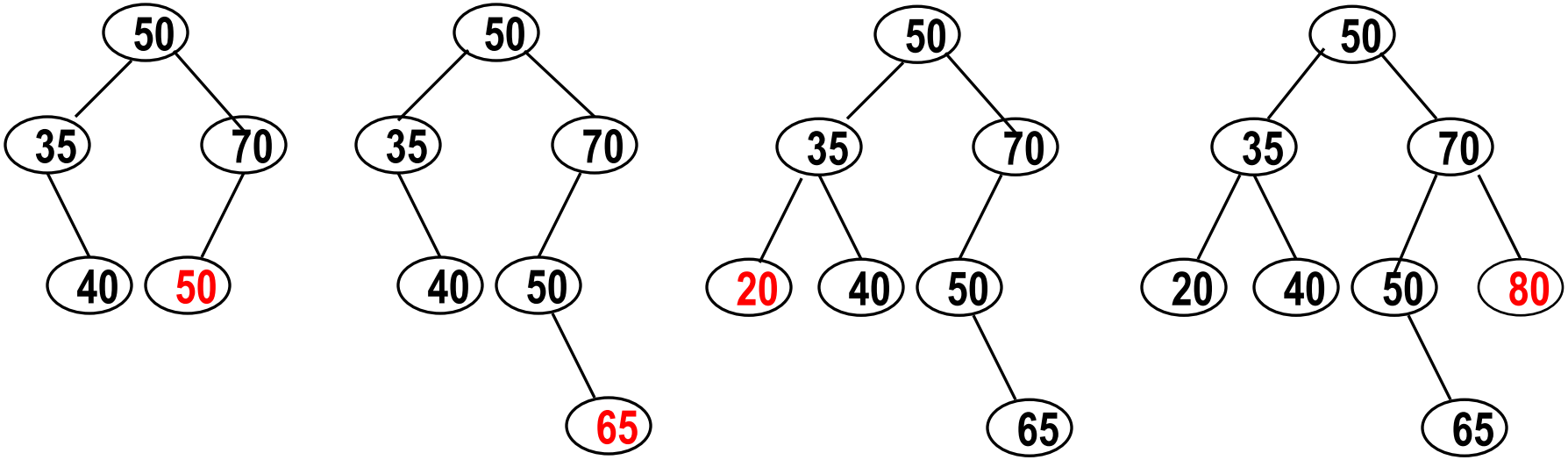
将k_i插入到左子树或者右子树中仍然遵循上述原则。

例 K = (50, 35, 70, 40, 50, 65, 20, 80)

空



注意：新插入的结点总是作为叶子结点！



```
/*
 *将一个数据元素item插入到二叉排序树中。
 */
BTNodeptr insertBST(BTNodeptr p, Datatype item)
{
    if (p == NULL) {
        p = (BTNodeptr)malloc(sizeof(BTNode));
        p->data = item;
        p->lchild = p->rchild = NULL;
    }else if (item < p->data) {
        p->lchild = insertBST(p->lchild, item);
    }else if (item > p->data) {
        p->rchild = insertBST(p->rchild, item);
    }else
        //do something; //如树中存在该元素
    return p;
}
```

```
#include <stdio.h>
typedef int Datatype;
struct node {
    Datatype data;
    struct node* lchild, * rchild;
};
typedef struct node BTNode, * BTNodeptr;
BTNodeptr insertBST(BTNodeptr p,
                    Datatype item);

int main() {
    int i, item;
    BTNodeptr  root = NULL;
    for (i = 0; i < 10; i++) {
        scanf("%d", &item);
        root = insertBST(root, item);
    }
    return 0;
}
```

非递归算法

功能　将一个数据元素item插入到根指针为Root的二叉排序树中。

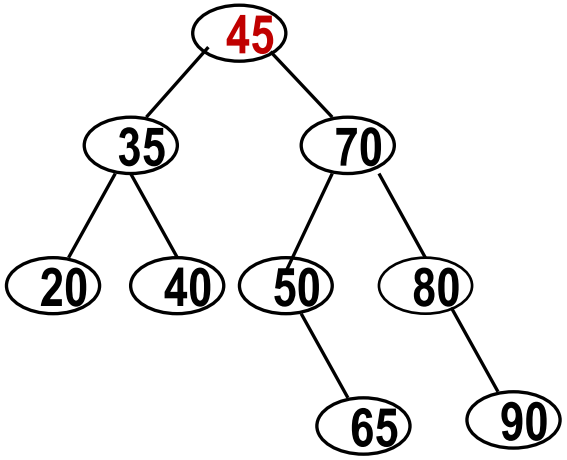
```
//Root为根结点指针，全局变量
BTNodeptr  Root=NULL;

void  insertBST( Typedata item){
    BTNodeptr p, q;
    p=(BTNodeptr)malloc(sizeof(BTNode));
    p->data=item;
    p->lchild=NULL;
    p->rchild=NULL;

    if(Root==NULL){
        Root=p;
        return;
    }
    q=Root;
    while(1)
        /* 比较值的大小 */
        /* 小于向左，大于向右 */
        if(item<q->data) {
            if(q->lchild==NULL){
                q->lchild=p;
                break;
            }else
                q=q->lchild;
        }else if (item>q->data) {
            if(q->rchild==NULL){
                q->rchild=p;
                break;
            }else
                q=q->rchild;
        } else { /* do-something */ }
    } // end of while
}
```

建立一个新结点

插入结点

二叉查找树与二分查找判定树

K = (45, 35, 70, 40, 50, 65, 20, 80, 90)



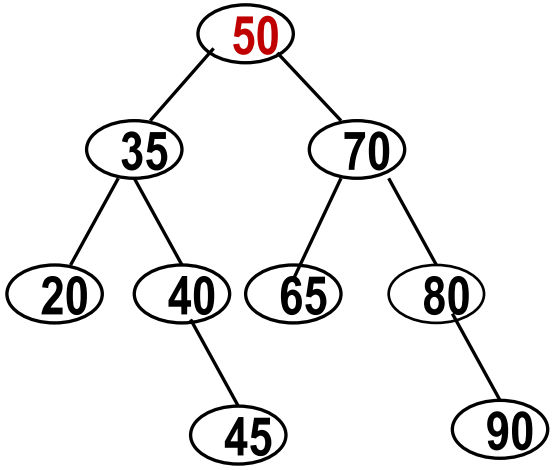
二叉查找树

二叉查找树以无序序列构造：首个元素为树根

二叉查找树的左右子树可能很不均衡

中序序列： 20, 35, 40, 45, **50**, 65, 70, 80, 90

描述二分查找过程的“判定树”

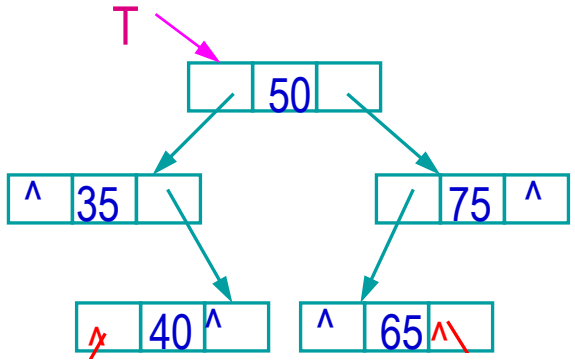


判定树以有序序列“构造”：中间元素为树根

判定树的左右子树相对均衡，其形状**主要**受**结点个数**的影响……

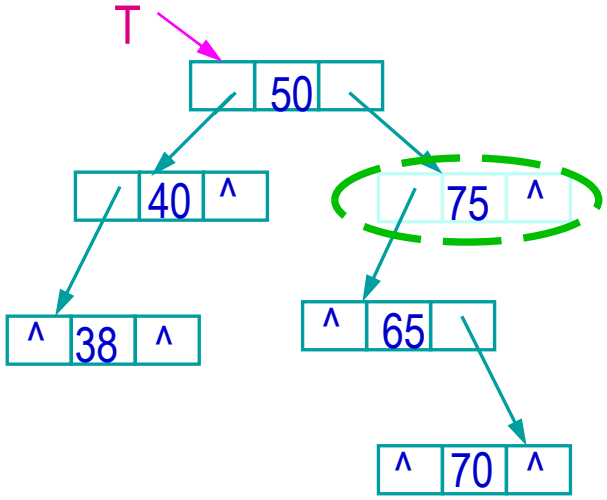
7.3 二叉查找树的删除（删除一结点）

1. 被删除结点为叶结点，
则直接删除。

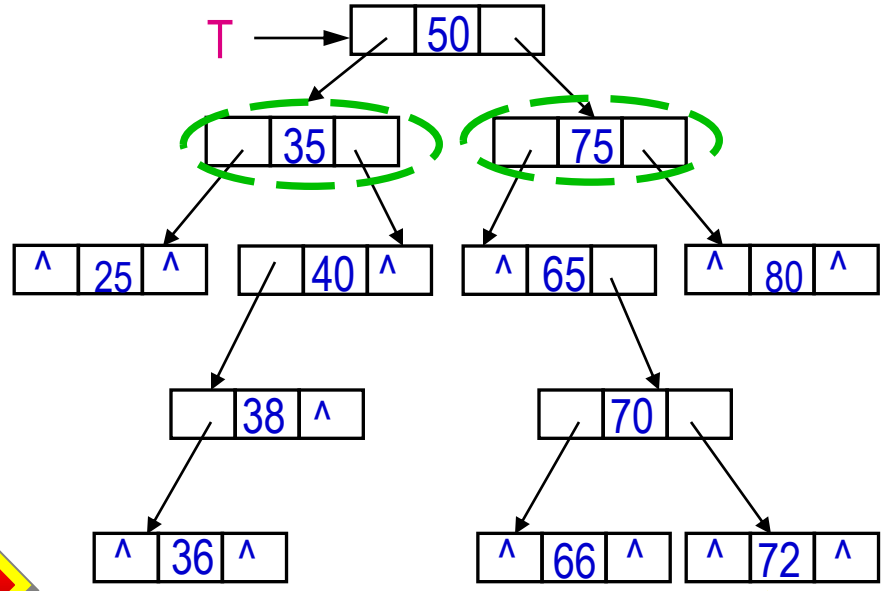


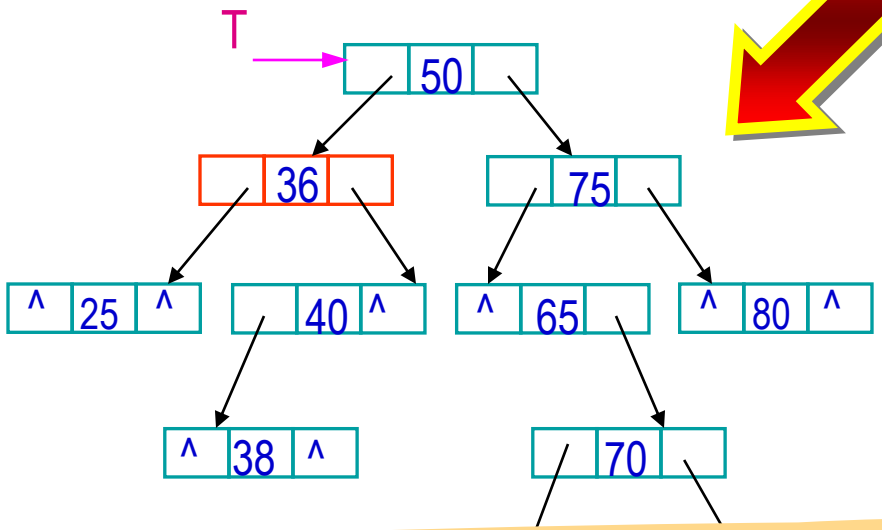
2. 被删除结点无左子树，
则用右子树的根结点取代被删除结点。

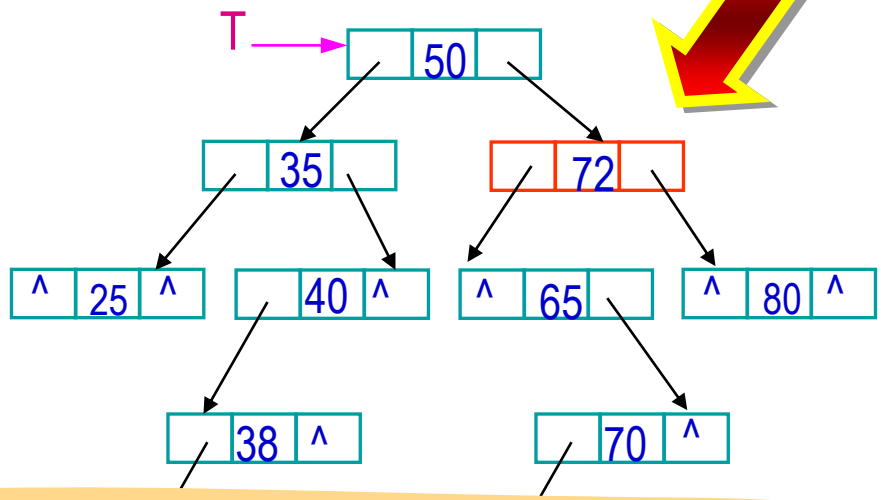
3. 被删除结点无右子树，
则用左子树的根结点取代被删除结点。



4. **被删除结点的左、右子树都存在，**则用被删除结点的右子树中值最小的结点（或被删除结点的左子树中值最大的结点）取代被删除结点。







延伸阅读*：请同学自学有关二叉查找树结点删除算法的C实现

关于二叉查找树

对于输入序列**无序**、且需要频繁进行**查找**、**插入**和**删除**操作的**动态表**（如词频统计中的单词表），如何选取数据结构和算法即能兼顾插入和删除效率，又能较**高效率地实现查找**？

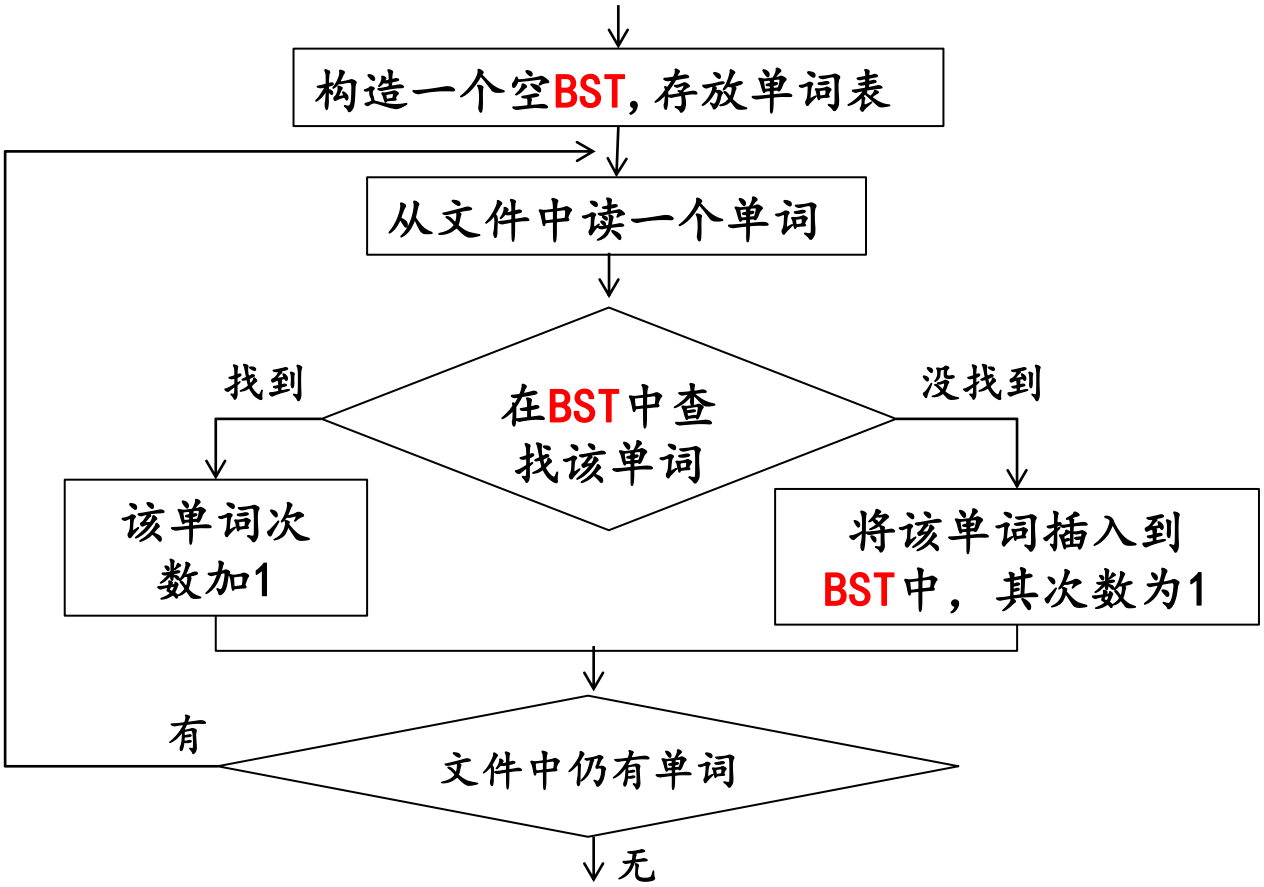
有序顺序表(数组)	有序链表	二叉查找树
✓ 有序顺序存储的数据能够采用如折半查找等算法，**查找效率高** ✓有序顺序存储数据由于据需要移动数据，**插入和删除操作效率低** ✓顺序存储需要事先分配空间，**空间利用率低，同时存在溢出风险**	✓ 有序链表存储，数据**插入和删除操作效率高** ✓链式存储能够动态分配空间，**空间利用率高** ✓链表存储的数据查找必须从链头开始，数据**查找效率低**	✓ 数据**插入和删除操作效率高** ✓能够动态分配空间，**空间利用率高** ✓数据**查找效率较高**（理想情况下查找效率为$O(\log_2 n)$）

关于二叉查找树*

- 从二叉查找树的定义与构造方式可见，其查找某个元素的效率要比采用顺序比较（如链表）的效率要高得多。
- 二叉查找树特别适合于数据量大、且无序的数据，如单词词频统计（单词索引）等。

问题5.1：词频统计(二叉查找树)

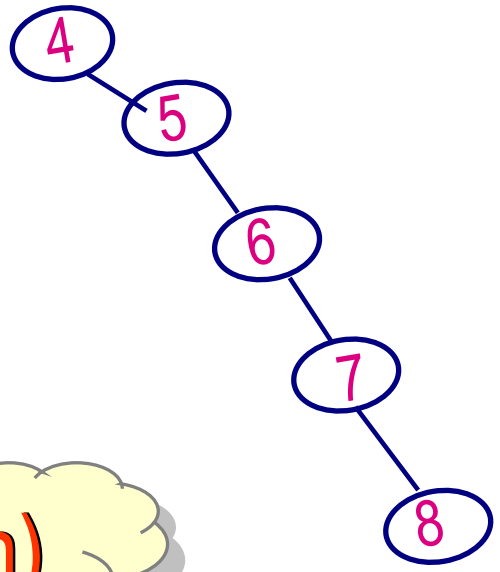
- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：本问题算法基本上只有**查找**和**插入**操作。



二叉查找树的缺陷：树的形态无法预料、随意性大。得到的可能是一个不平衡的树，即树的深度差很大。

——失去了利用二叉树组织数据所带来的好处。

例，逐点插入：4，5，6，7，8



查找时间

$O(\log_2 n)$

比较理想的情况

$O(n)$

当出现“退化二叉树”时

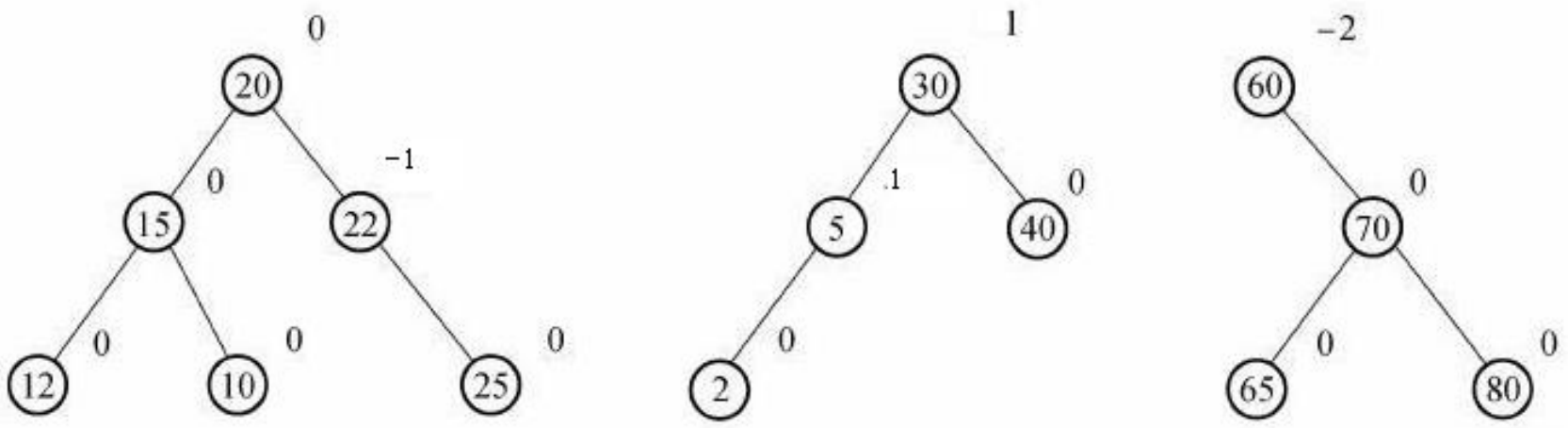
如果被插入的元素序列是随机序列，或者序列的长度较小，采用“逐点插入法”建立二叉查找树可以接受。

如果建立的二叉查找树中出现结点子树的深度之差较大时（即产生不平衡），就有必要采用其他方法建立二叉查找树，即建立所谓“平衡二叉树”。

8.平衡二叉树 (Adelson-Velskii and Landis, AVL)

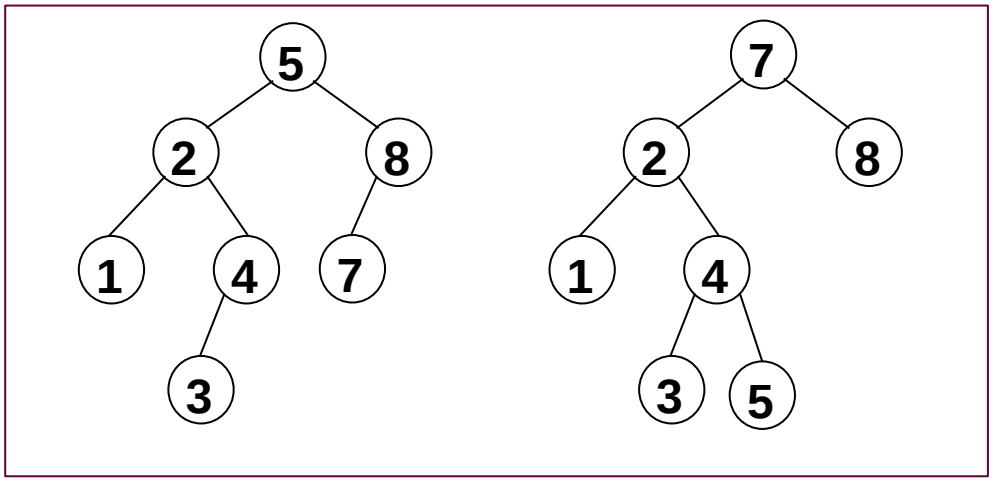
平衡二叉树又称AVL树。它或者是一棵空树,或者是具有下列性质的二叉树： 它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。

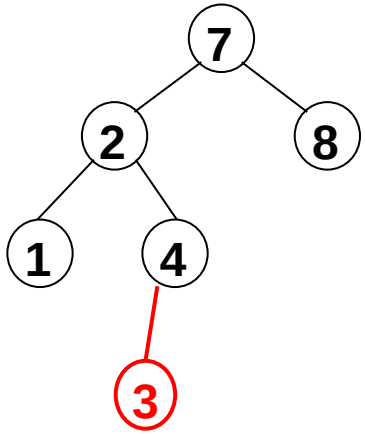
若定义二叉树结点的平衡因子为该结点左子树深度减去右子树深度，则平衡二叉树上所有结点的平衡因子只可能是-1、0和1。



8.平衡二叉树 (Adelson-Velskii and Landis, AVL)

- 平衡因子与结点的值和插入的次序有关。



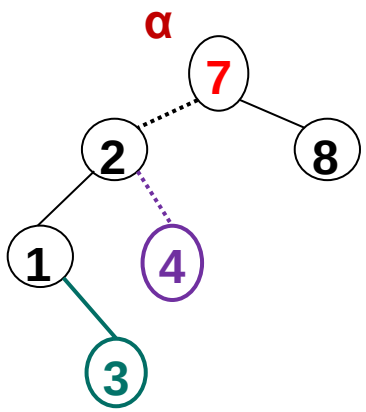


两棵查找树，其中左图的树是平衡树

- 插入结点时可能影响该结点祖先结点的平衡因子
- 在插入和删除结点的同时对二叉树的形态（结构）进行必要的调整，使之重新处于平衡状态。

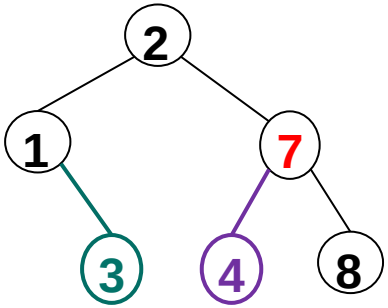
如何再平衡*

- 必须被平衡的结点（其平衡因子为2或-2）称为α，分为四种情况：
  - α左儿子的左子树进行插入操作
  - α左儿子的右子树进行插入操作
  - α右儿子的左子树进行插入操作
  - α右儿子的右子树进行插入操作



- 如何调整：旋转

延伸阅读*：请同学自学有关平衡二叉树构造算法的C实现



✓ 频繁插入/删除带来的旋转操作可能使AVL性能下降
→ 放宽平衡程度的要求(红黑树*)
✓ 更一般的多路查找树是B树，见后续章节“查找”。

几种非常有用的、有代表性的二叉树

线索二叉树* (Threaded Binary Tree)

二叉查找树 (Binary Search Tree, BST)

平衡二叉树或AVL树 (Balanced Binary Tree)

堆 (Heap)

哈夫曼树 (Huffman Tree)

9.堆(heap)

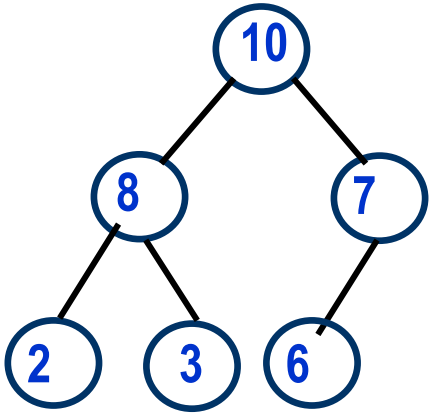
9.1 堆的基本性质

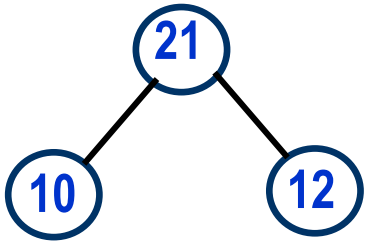
堆是一种特殊类型的完全二叉树，具有以下两个性质：

（1）每个节点的值大于（或小于）等于其每个子节点的值；

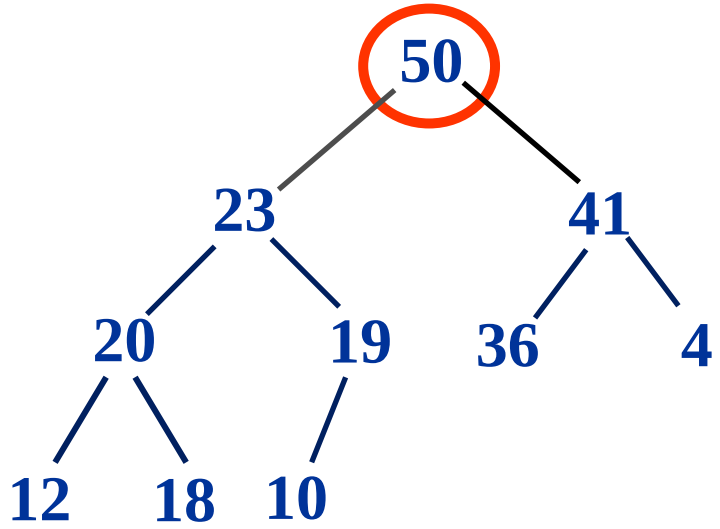
（2）该树完全平衡，其最后一层的叶子都处于最左侧的位置。

满足上面两个性质的是大顶堆max heap（或小顶堆min heap）。





大顶堆



序列第一个元素或者二叉树的根结点的值最大

序列形式

50 23 41 20 19 36 4 12 18

由于堆是一个完全平衡树，故可实现为数组：

heap[i] ≧heap[2*i]

heap[i] ≧heap[2*i+1]，　　i=1,2,3,…n

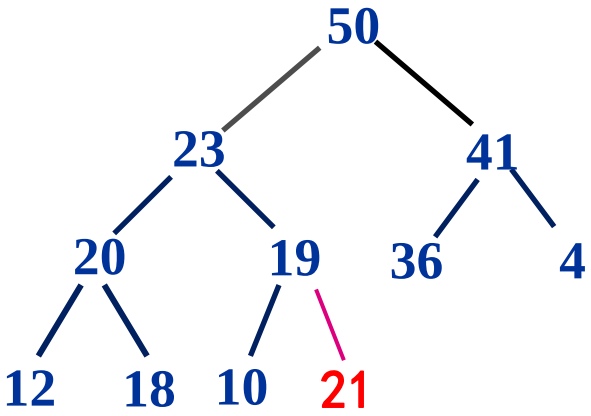
在堆（数组）的头取数据，在尾插入数据

9.2 堆的基本操作

堆的基本操作：insert(插入)和delete(删除)

类似堆栈和队列，堆也是操作受约束的数据结构

- 插入（尾部）
- 删除（头部/根）



插入算法

heapInsert(e)

将e放在堆的末尾;

while e 不是根 && e > parent(e)

e 与其父节点交换


```
#define HEAPSIZE    1000          //堆元素个数上限
ElemType  Heap[HEAPSIZE];         //用一位数组实现堆
int  Hnum;                        //当前堆中元素个数
```

```
void  heapInsert(ElemType e, ElemType heap[ ]){
    int i;
    if(isFull(heap))
        Error("Heap is full");

    for(i= Hnum++; i!=0&&(e > heap[(i-1)/2]) ; i=(i-1) / 2)
        heap[i] = heap[(i-1)/2];  //上滤
    heap[i] = e;
}
```

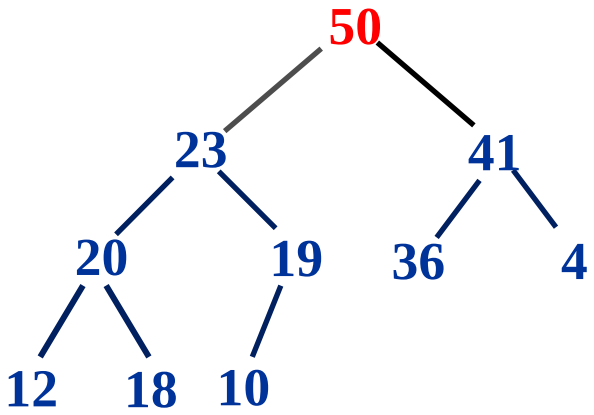
算法复杂度为$O(\log_2 n)$

9.2 堆的基本操作(续)

堆的基本操作：insert(插入)和delete(删除)

删除算法

取堆顶元素，并从堆中删除。



heapDelete() //取堆顶（树根）元素

从根节点提取元素；

将最后一个叶节点中的元素放到根的位置；

删除最后一个叶节点；

```
//下滤
p = 根节点 ;
while p不是叶节点 && p<它的任何子节点
      p与其较大的子节点交换
```

```
ElemType  heapDelete(ElemType heap[ ]){
    int i=0, j; //i指向父结点；j指向i的孩子结点
    ElemType  cur, last;
    if(isEmpty(heap))
        Error("Heap is empty");
    cur = heap[0]; last = heap[--Hnum];
    while( i*2+1 < Hum) { //下滤
        j = i*2+1;
        if(j != Hum-1 && heap[j]  < heap[j+1]) //j指向较大的孩子
            j++;
        if( last < heap[j])
            heap[i] = heap[j];
        else
            break;
         i = j;
     }
     heap[i] = last;
     return cur;
}
```

算法复杂度为$O(\log_2 n)$

9.3 堆的构造

堆的构造有两种方法：自顶向下(John Williams提出)和自底向上(Robert Floyd提出)

自顶向下

从空堆开始，依次向堆中添加(**heapinsert**函数)元素

自底向上

先建立无序的二叉树，然后从底层开始构造较小的堆，然后再重复构造较大的堆

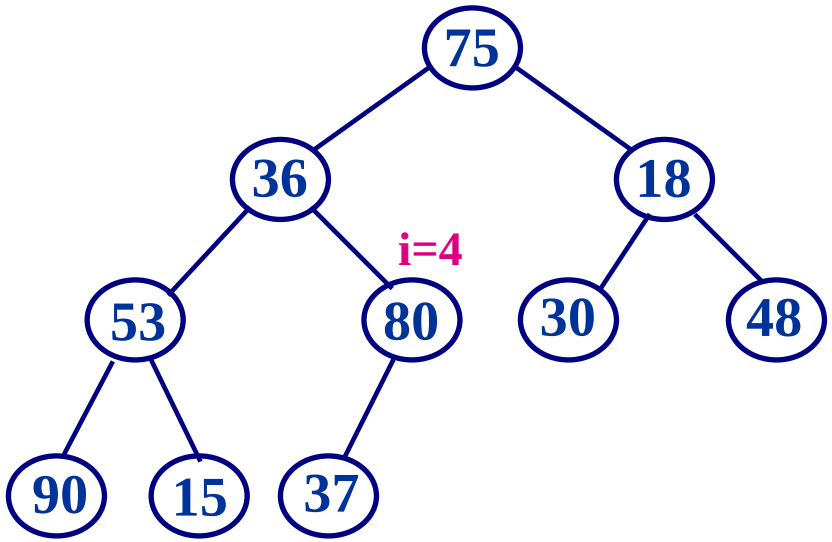
自底向上调整法

从二叉树的最后那个分支结点(编号为 i=⌊n/2-1⌋)开始，依次将编号为i的结点为根的二叉树**转换**为一个堆；每转换一棵子树，做一次i减1，直到将i=0的结点为根的二叉树转换为堆。

注意：以此方法，每次调整时当前子树仅根结点不满足堆条件

例

75 36 18 53 80 30 48 90 15 37




```
//自底向上调整核心代码
for(i=n/2-1; i>=0; i--)
    adjust(heap, i, n);
```

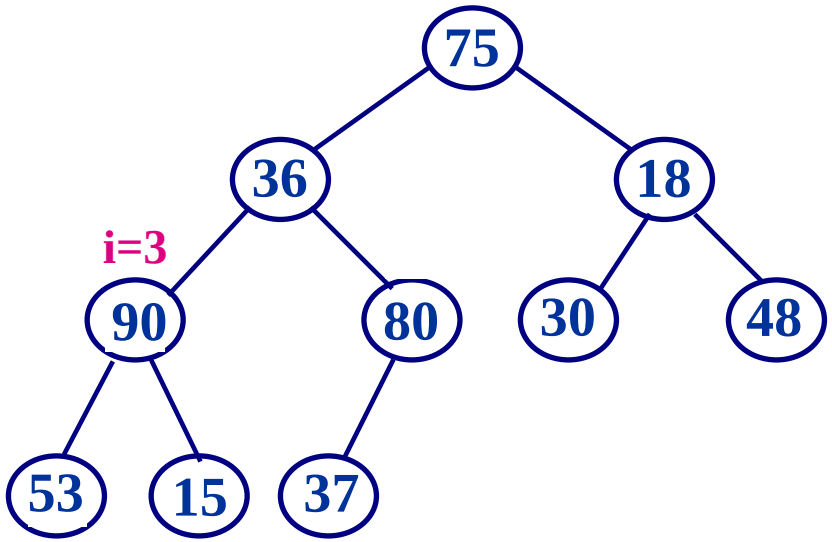
自底向上调整法

从二叉树的最后那个分支结点(编号为 $i=\lfloor n/2-1 \rfloor$)开始，依次将编号为i的结点为根的二叉树**转换**为一个堆；每转换一棵子树，做一次i减1，直到将i=0的结点为根的二叉树转换为堆。

注意：以此方法，每次调整时当前子树仅根结点不满足堆条件

例

75 36 18 53 80 30 48 90 15 37




```
//自底向上调整核心代码
for(i=n/2-1; i>=0; i--)
    adjust(heap, i, n);
```

自底向上调整法

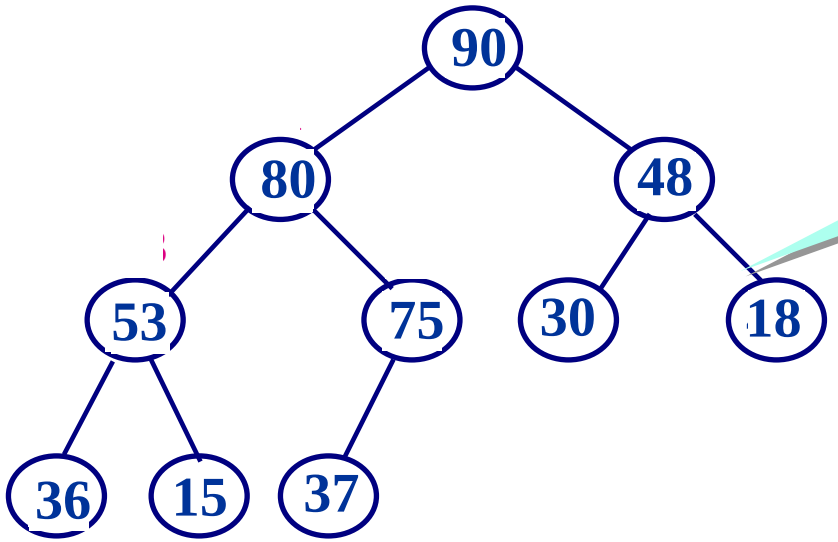
从二叉树的最后那个分支结点(编号为 i=⌊n/2-1⌋)开始，依次将编号为i的结点为根的二叉树**转换**为一个堆；每转换一棵子树，做一次i减1，直到将i=0的结点为根的二叉树转换为堆。

注意：以此方法，每次调整时当前子树仅根结点不满足堆条件

例

75	36	18	53	80	30	48	90	15	37
90	80	48	53	75	30	18	36	15	37

i=0



初始堆积

```
//自底向上调整核心代码
for(i=n/2-1; i>=0; i--)
    adjust(heap, i, n);
```

堆调整子算法

若一棵树仅根结点i不满足堆条件，则该函数可将其调整为一个堆。
向下调整结点i的位置，使得其祖先结点值 都比其大。

```
/*调整一棵子树：i为子树根的编号，n为二叉树的结点数目*/
void  adjust(keytype k[ ], int i, int n){
      int j;
      keytype  temp=k[i];
      j=2*i+1;
      while(j<n){
          if(j+1<n && k[j]<k[j+1])  //选出大的孩子
              j++;
          if(temp<k[j]) {  //下滤
              k[(j-1)/2]=k[j];
              j=2*j+1;
          }
          else break;
       }
       k[(j-1)/2]=temp;
}
```

建初始堆积

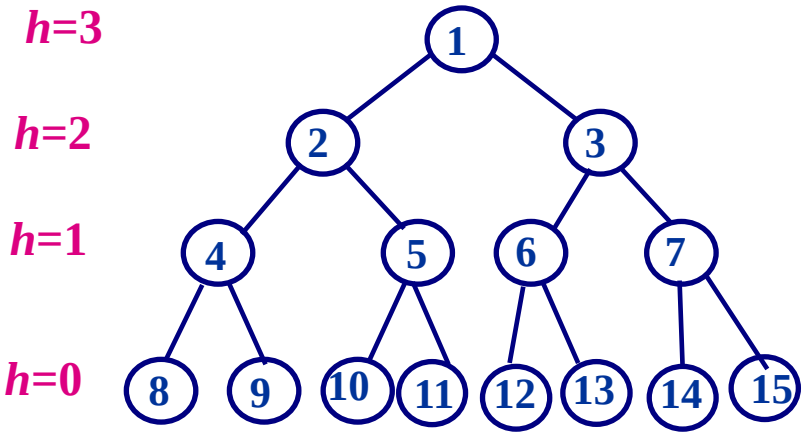
```
for(i=n/2-1;i>=0;i– –)
        adjust(K,i,n);
```

第h层节点数 $\leq \left\lceil 2^{(\lg n-h)-1} \right\rceil = \left\lceil \frac{2^{\lg n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$

时间复杂度= $\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right)$

$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$

$= O(n)$



堆的典型应用

优先队列（Priority queue）：与传统队列不同的是下一个服务对象是队列中优先级最高的元素。优先队列常用的实现方式是用堆，其最大好处是管理元素的效率高(O($\log_2$ N))。

优先队列是计算机中常用的一种数据结构，如操作系统中进程调度就是基于优先队例。

堆排序（Heap sort）：一种基于堆的高效排序算法，时间复杂度为O(n$\log_2$n)。

回顾：选择(select)排序法

核心思想

第i趟排序从未排序子序列(原始序列的后n−i+1个元素)中 **选择** 一个值最小的元素，将其置于子序列的最前面。

选择	35	97	38	27	65	13	80	75
交换	13	97	38	27	65	35	80	75
选择	13	97	38	27	65	35	80	75
交换	13	27	38	97	65	35	80	75

9.4 堆(Heap)排序法

由John Williams提出

选择排序时，从无序子序列中挑选最大/最小元素的代价较高，倘若利用堆技术选择元素，则可以显著提高算法性能。

堆排序的核心思想

第i趟排序将未排序元素（序列的前n−i+1个元素组成的子序列）**转换**为一个**堆积**，然后将堆的第一个元素与堆的最后那个元素交换位置。

是选择排序的堆优化！

堆排序步骤

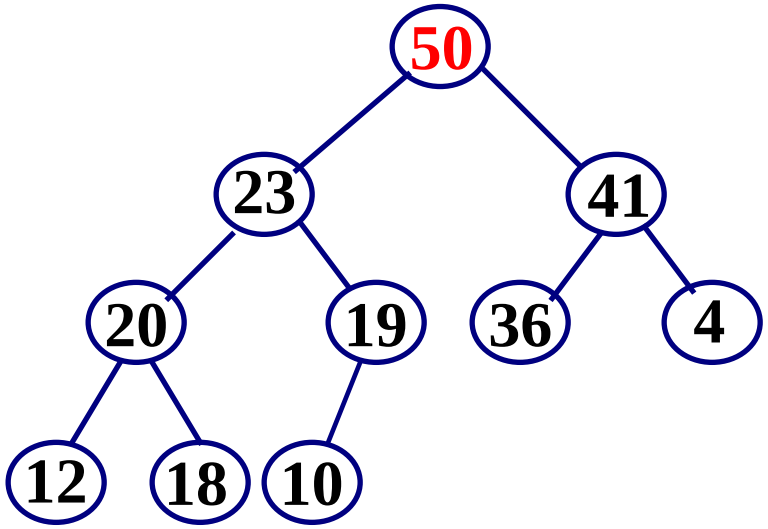
注意：堆可以存储在顺序表中

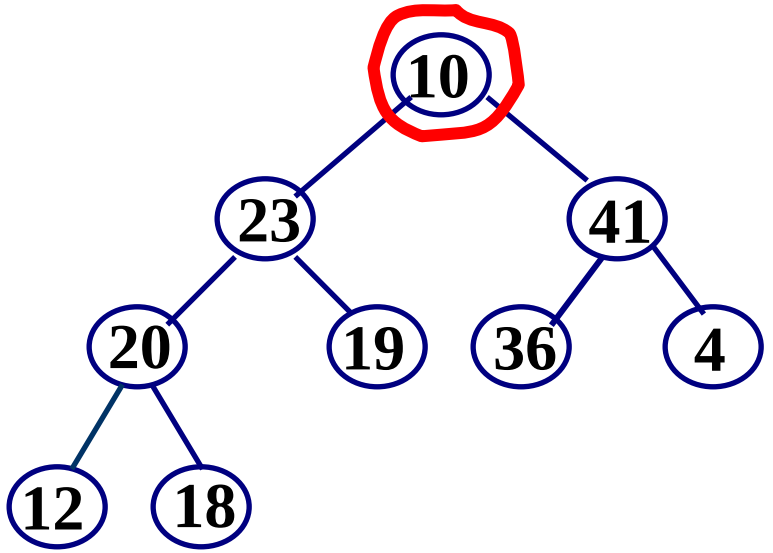
1. 建初始堆积：将原始序列转换为第一个堆。

2. 将堆的第一个元素与堆积的最后那个元素交换位置。(即“选出”最大值元素并交换位置)

3. 将剩余元素（即“去掉”最大值元素后）组成的子序列重新调整为堆。

4. 重复上述过程的第2至第3步n–1次。





下滤：将根结点往叶结点方向移

… …

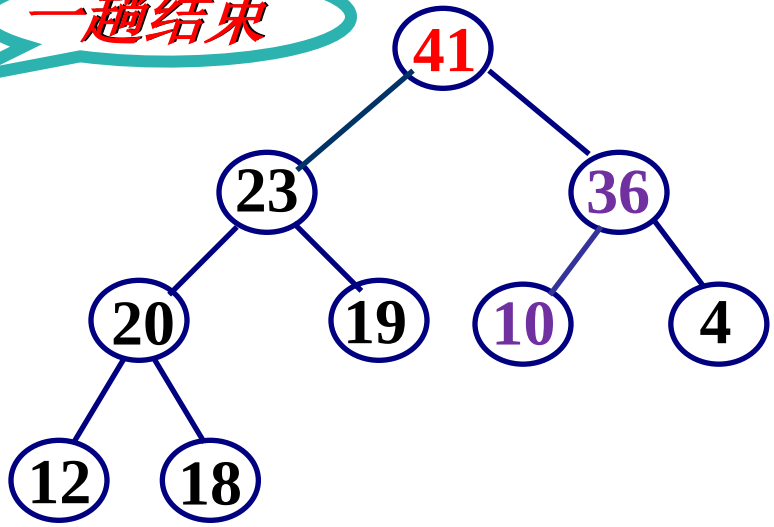
50 23 41 20 19 36 4 12 18 10
10 23 41 20 19 36 4 12 18 50

41 23 36 20 19 10 4 12 18 50
18 23 36 20 19 10 4 12 41 50

36 23 18 20 19 10 4 12 41 50
12 23 18 20 19 10 4 36 41 50

… …

一趟结束



堆排序完整过程示例

N-1趟，每趟代价为logn

原始	75	36	18	53	80	30	48	90	15	37
第1趟	37	80	48	53	75	30	18	36	15	90
第2趟	15	75	48	53	37	30	18	36	80	90
第3趟	15	53	48	36	37	30	18	75	80	90
第4趟	18	37	48	36	15	30	53	75	80	90
第5趟	18	37	30	36	15	48	53	75	80	90
第6趟	15	36	30	18	37	48	53	75	80	90
第7趟	15	18	30	36	37	48	53	75	80	90
第8趟	15	18	30	36	37	48	53	75	80	90
第9趟	15	18	30	36	37	48	53	75	80	90

堆排序算法

```
void heapSort(keytype k[ ],int n){
    int i,
    keytype  temp;
    for(i=n/2-1;i>=0;i--)
        adjust(k,i,n);
    for(i=n-1;i>=1;i--){
        temp=k[i];
        k[i]=k[0];
        k[0]=temp;
        adjust(k,0,i);
    }
}
```

建初始堆积

具体排序

n−1趟排序

$O(n\log_2 n)$

heap排序是 不稳定的 。

堆排序算法分析

稳定性:	不稳定
时间代价:	$O(nlog_2n)$
空间代价:	O(1) 注：此处仅考虑辅助空间

注：可看作是选择排序的优化（提高选择的效率）

几种非常有用的、有代表性的二叉树

线索二叉树* （Threaded Binary Tree）

二叉查找树 （ Binary Search Tree, BST）

平衡二叉树或AVL树 (Balanced Binary Tree)

堆 (Heap)

哈夫曼树 (Huffman Tree)

字符编码

例

要传送的文字

'ABACCDA'

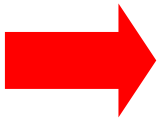
如何找到高效且正确的编码方式呢?

一种错误的不等长编码

例如令 **A=0　B=00　C=1　D=01**

要传送的文字

'ABACCDA'



由二进制代码组成的电文

0000 11010

AAAA　ABA　BB

缺点： 编码表示不惟一，致使电文难以/无法翻译。

要求： 任意一个字符的编码不能是另一个字符的编码的前缀。

字符编码

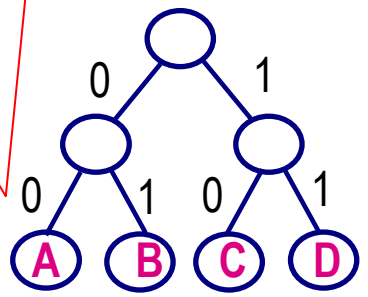
左分支表示字符0，右分支表示字符1。从根结点到叶结点的路径上分支字符组成的字符串作为该叶结点字符的编码。

例

要传送的文字

'ABACCDA'

正确的编码应该让字符位于叶子结点



一种正确的编码方案 (但不够精简)

例如：**A=00　B=01　C=10　D=11**

要传送的文字

'ABACCDA'

由二进制代码组成的电文(14bits)

00010010101100

不够精简

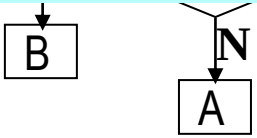
如何得到使电文总长最短的编码？

再考察一个实例

假设输入10000个同学的成绩

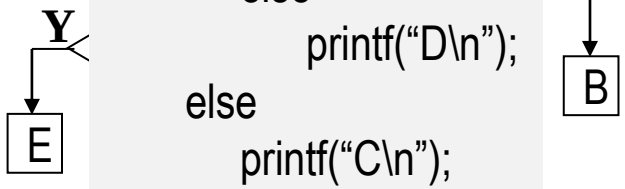
等级	E	D	C	B	A
分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10

```
if(a<60)
    printf("E\n");
else if(a<70)
    printf("D\n");
else if(a<80)
    printf("C\n");
else if(a<90)
    printf("B\n");
else
    printf("A\n");
```



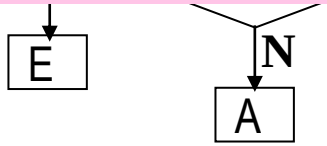
比较31500次

```
if(a<80)
    if(a<70)
        if(a<60)
            printf("E\n");
        else
            printf("D\n");
    else
        printf("C\n");
else
    if(a<90)
        printf("B\n");
    else
        printf("A\n");
```



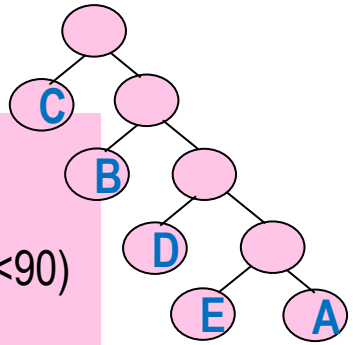
比较22000次

```
if(70<=a && a<80)
    printf("C\n");
else if(80<=a && a<90)
    printf("B\n");
else if(60<=a && a<70)
    printf("D\n");
else if(a<60)
    printf("E\n");
else
    printf("A\n");
```



比较20500次

二叉树



字符编码：

要传送的文字

'ABACCDA'

成绩分级：

等级	E	D	C	B	A
分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10

共性特征：

1. 可以表示成二叉树，且“对象”作为叶子结点
2. 不同“对象”出现次数不一样——权重不一样

10 哈夫曼（Huffman）树及其应用

10.1 哈夫曼树的基本概念

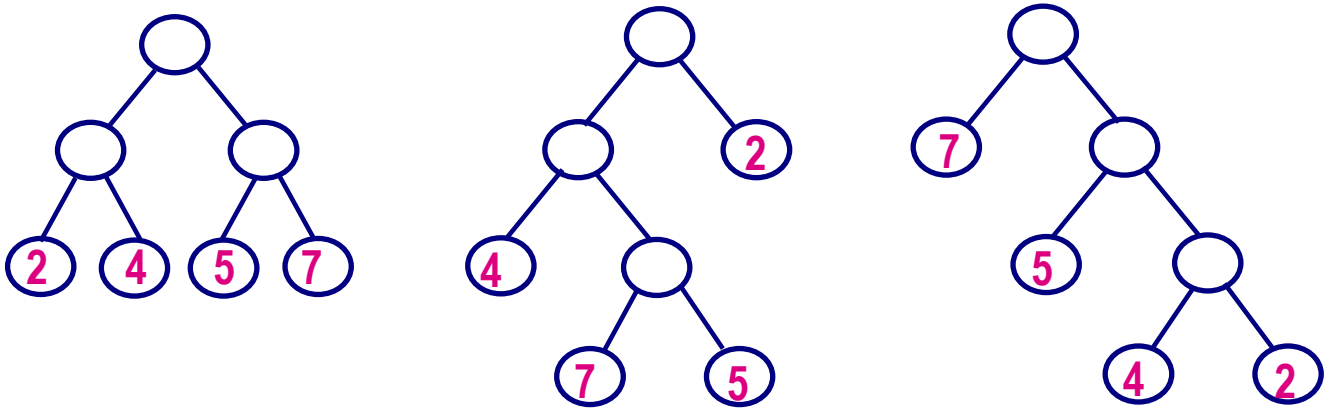
1. 树的带权路径长度

若给二叉树的每个叶结点 i 赋予一个权值，则该二叉树的带权路径长度定义为 $WPL=\sum w_i l_i$

根结点到所有结点的带权路径长度之和。

其中，w_i为第i个叶结点的权值，l_i为第i个叶结点的路径长度。

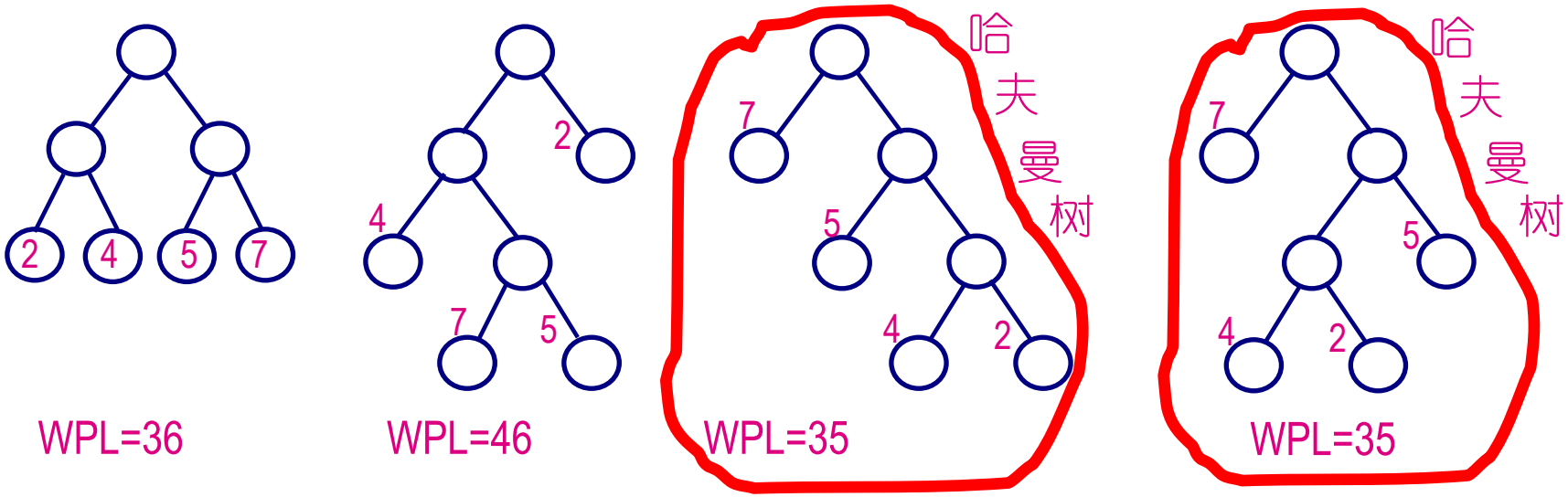
例 对于上述字符集{A,B,C,D}对应的一组权值{ 7, 5, 2, 4 }，可以有如下多个带权二叉树



WPL=36　　WPL=46　　WPL=35

2.哈夫曼树的定义

给定一组权值，构造出的具有最小带权路径长度的二叉树称为 哈夫曼树



例，一组权值 { 7, 5, 2, 4 }

3. 哈夫曼树的特点

（1）**权值越大的叶结点离根结点越近，权值越小的叶结点离根结点越远**；(这样的二叉树WPL最小)

（2）无度为1的结点；

（3）哈夫曼树不是惟一的。

10.2 哈夫曼树的构造

核心思想

1. 对于给定的权值$W=\{w_1,w_2,\ldots,w_m\}$，构造出树林$F=\{T_1,T_2,\ldots,T_m\}$，其中，$T_i(1\le i\le m)$为左、右子树为空，且根结点(叶结点)的权值为w_i的二叉树。
2. 将F中根结点权值最小的两棵二叉树合并成为一棵新的二叉树，即把这两棵二叉树分别作为新的二叉树的左、右子树，并令新的二叉树的根结点权值为这两棵二叉树的根结点的权值之和，将新的二叉树加入F的同时从F中删除这两棵二叉树。
3. 重复步骤2，直到F中只有一棵二叉树。

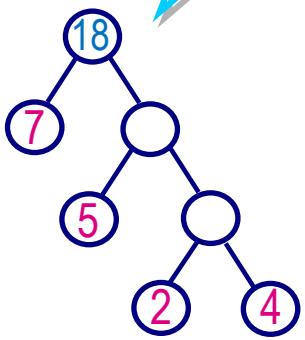
例 W={7, 5, 2, 4}

F={ 7 5 2 4 }

F={ 7 5 6 } （6 下有 2、4）

F={ 7 11 } （11 下有 5、18）

哈夫曼树



思考 设计何种数据结构？

在信息和网络时代，数据的压缩解压是一个非常重要的技术，现有的数据压缩和解压技术很多是基于Huffman的研究之上发展而来。

10.3 哈夫曼编码：一种熵编码方式

例

要传送的文字

'ABACCDA'

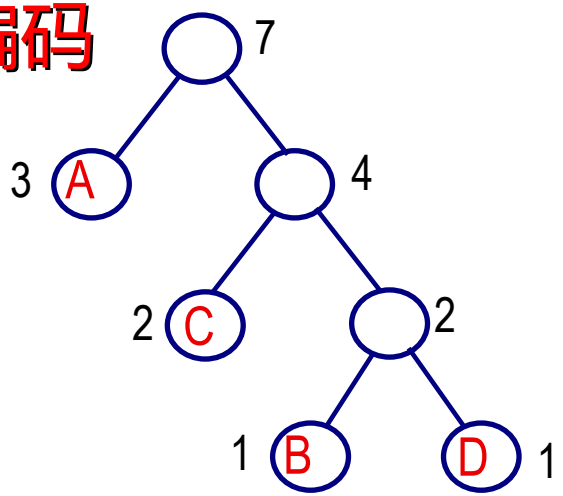
字符集：A B C D

出现次数：3 1 2 1

利用二叉树设计二进制前缀编码

左分支表示字符0

右分支表示字符1



A: 0　　B: 110　　C: 10　　D: 111

要传送的文字

'ABACCDA'

由二进制代码组成的电文(13bits)

0110010101110

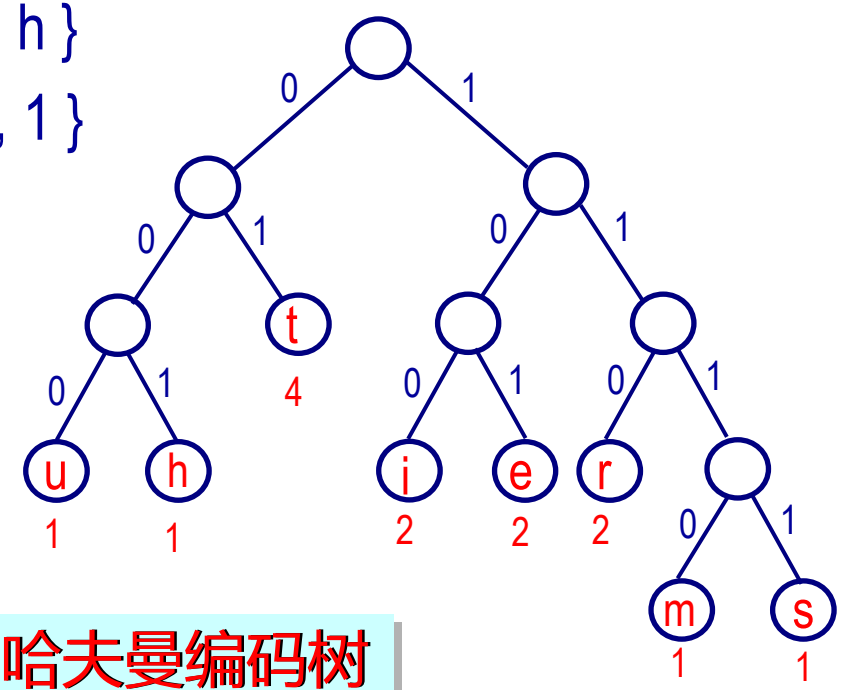
练习 电文：time tries truth 大家来构造一下这棵哈夫曼树？

字符集: { t, i, m, e, r, s, u, h }

字符出现次数: { 4, 2, 1, 2, 2, 1, 1, 1 }

哈夫曼编码

t:	01
i:	100
m:	1110
e:	101
r:	110
s:	1111
u:	000
h:	001



哈夫曼编码树

01 100 1110 101 01 110 100 101 1111 01 110 000 01 001

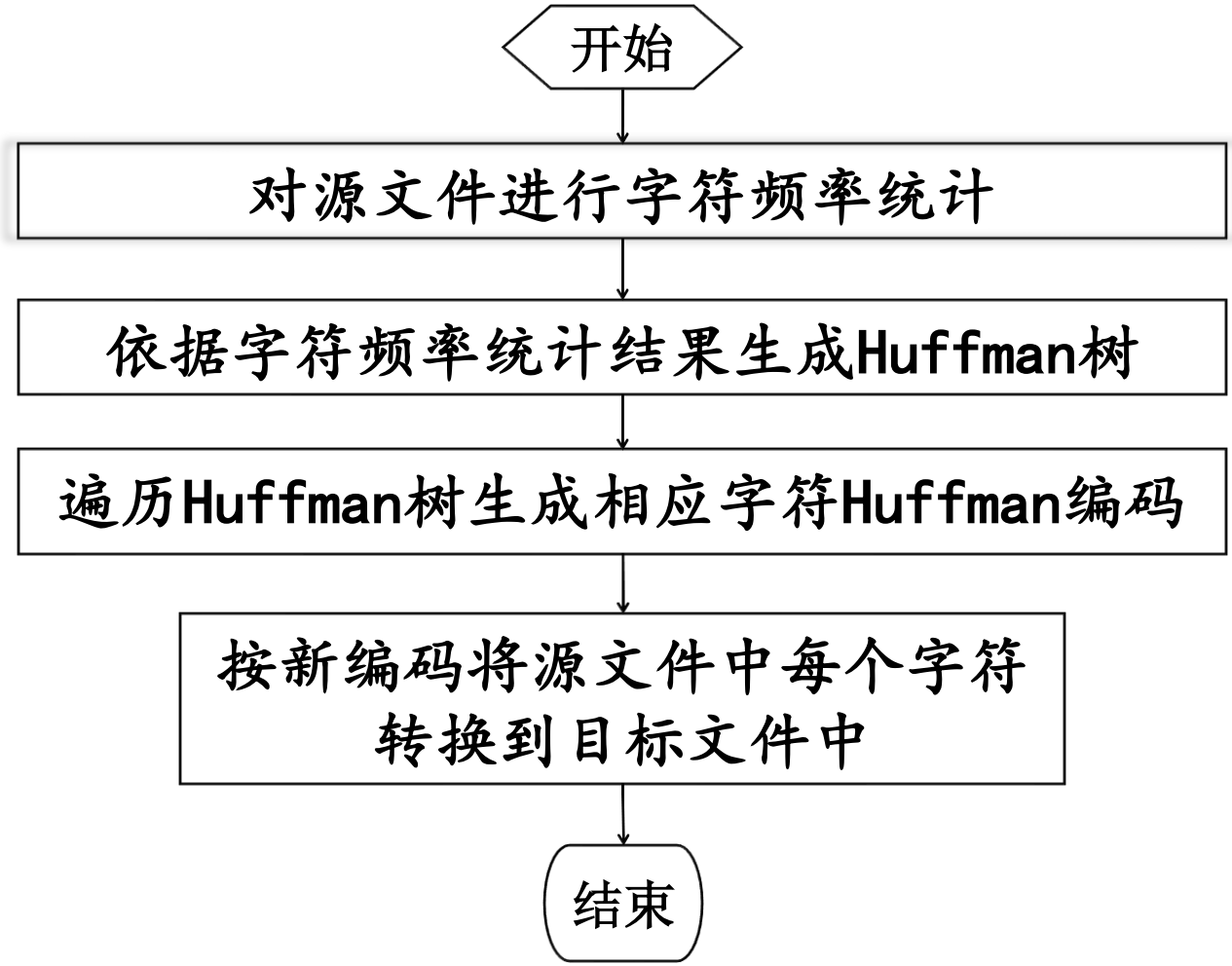
该电文以本Huffman编码传输时传输长度为：40 位bits（5字节）

而该电文以ASCII编码传输时传输长度为：14 字节(byte)

问题5.2：文件压缩-Huffman编码

- 编写一个程序采用Huffman编码实现对一个文件的压缩。
- 要求：首先读取文件，对文件中出现的每个字符进行字符频率统计，然后根据频率采用Huffman方法对每个字符进行编码，最后根据新字符编码表输出文件。

问题5.2：问题分析（算法流程）



字符频率表

树结构 + 链表

字符—编码表

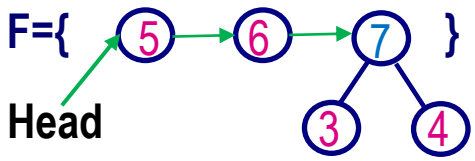
问题5.2：数据结构与算法设计 – 数据结构

```
#define MAXSIZE  32
struct cc {//字符及出现次数
    char c;
    int count;
} ;
struct cc ccount[128];
```

字符频率统计：

```
while( (c=fgetc(fp)) != EOF){
    ccount[c].c=c; ccount[c] .count++;
}
```

```
struct tnode { //Huffman树结点 + 链表结点
    struct cc ccount; //字符及出现次数
    struct tnode *left,*right;  //树的左右节点指针
    struct tnode *next;  //一个有序链表的节点指针
} ;
struct tnode *Head=NULL; //有序链表的头节点，也是最终Huffman树的根节点
```




```
char HCode[128][MAXSIZE]; //字符的Huffman编码，Hcode[0]为文件结束符的编码
                          //例如：Hcode['a']表示字符a的Huffman编码串。
```

问题5.2：思考

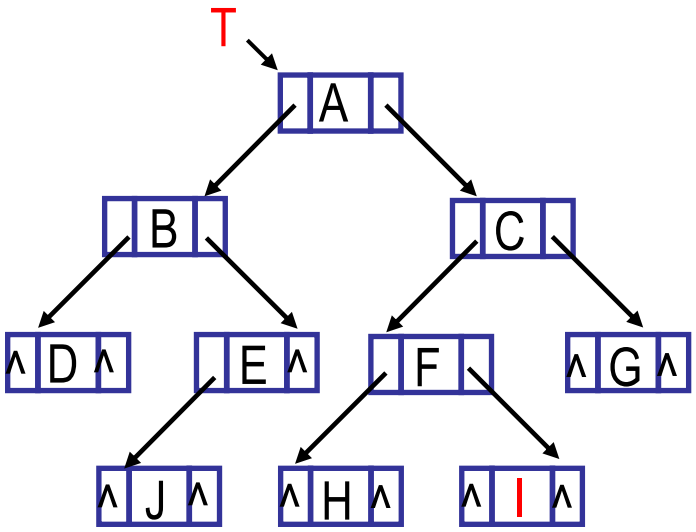
- 如果给定了字符的Huffman编码和一个用该编码压缩的文件，请解压该文件。（问题5.2的逆问题）。

本节结束！

6 线索二叉树*

6.1基本概念

如何在存储结构中
方便地知道结点的
直接前驱或后继



对于具有n个结点的二叉树，二叉链表中空的指针域数目为

n+1

中序序列：

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

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

指针域总数　已用指针域数

6.2 什么是线索二叉树

利用二叉链表中空的指针域指出结点在某种遍历序列中的直接前驱或直接后继。指向前驱和后继的指针称为**线索**，加了线索的二叉树称为**线索二叉树**。

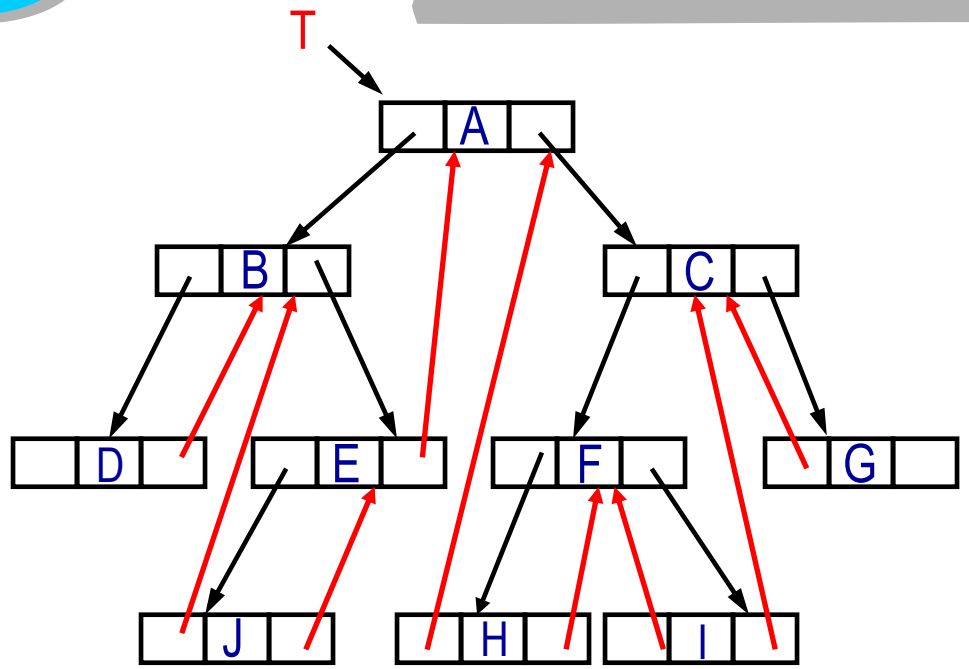
6.3 线索二叉树的构造

利用链结点的**空的左指针域**存放该结点的直接前驱的地址，**空的右指针域**存放该结点直接后继的地址；而非空的指针域仍然存放结点的左孩子或右孩子的地址。

以中序线索二叉树为例

中序序列:

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



中序线索二叉树

如何区分指针和线索

指针与线索的区分方法之一

lbit	lchild	data	rchild	rbit

约定

$$p\text{->lbit} = \begin{cases} 0 & \text{表示p->lchild为指向直接前驱的线索} \\ 1 & \text{表示p->lchild为指向左孩子的指针} \end{cases}$$

$$p\text{->rbit} = \begin{cases} 0 & \text{表示p->rchild为指向直接后继的线索} \\ 1 & \text{表示p->rchild为指向右孩子的指针} \end{cases}$$

指针与线索的区分方法之二

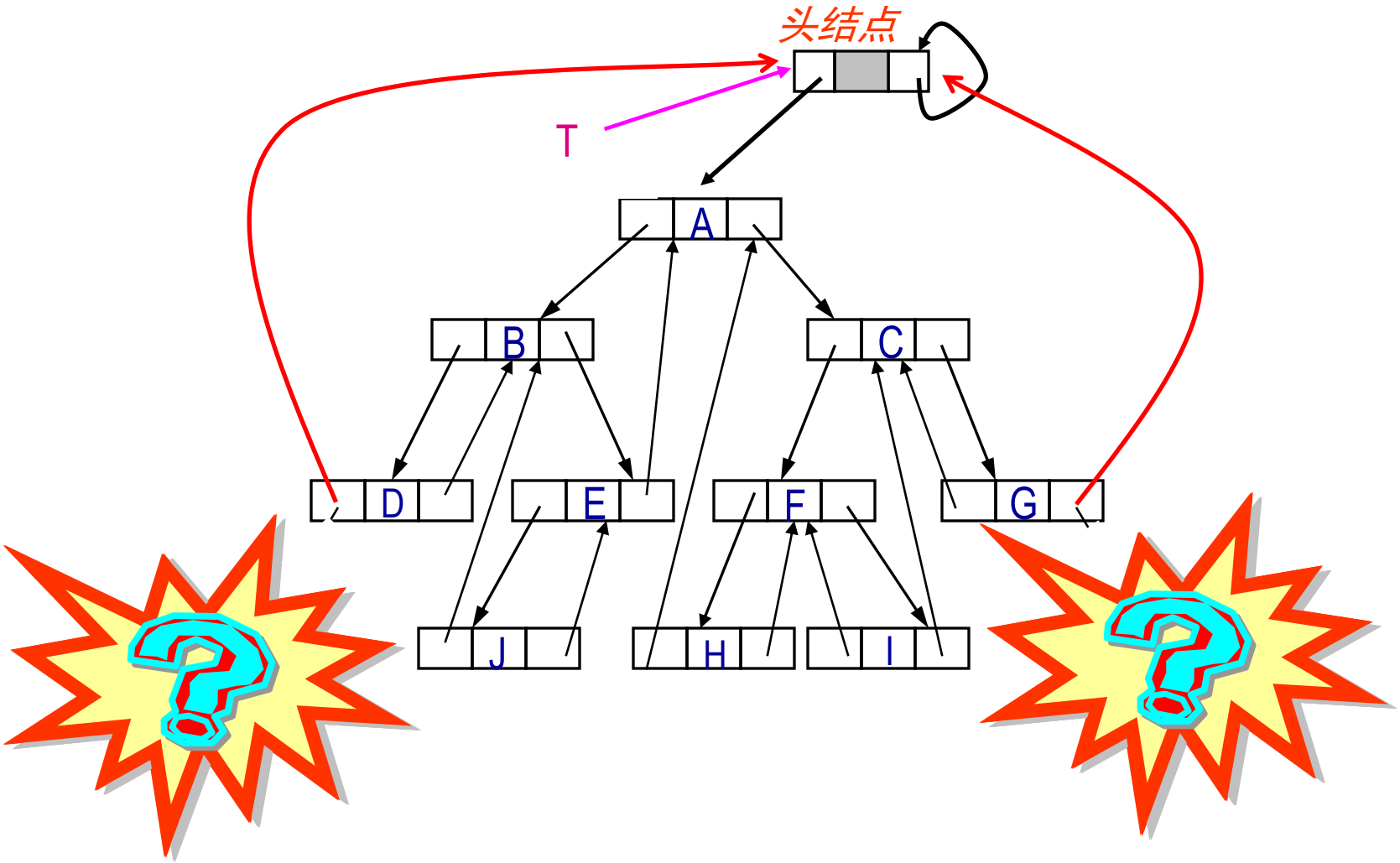
不改变链结点的构造，而是在作为线索的地址前加一个负号，即“负地址”表示线索，“正地址”表示指针。

lbit	lchild	data	rchild	rbit

线索二叉树链结点类型定义为：

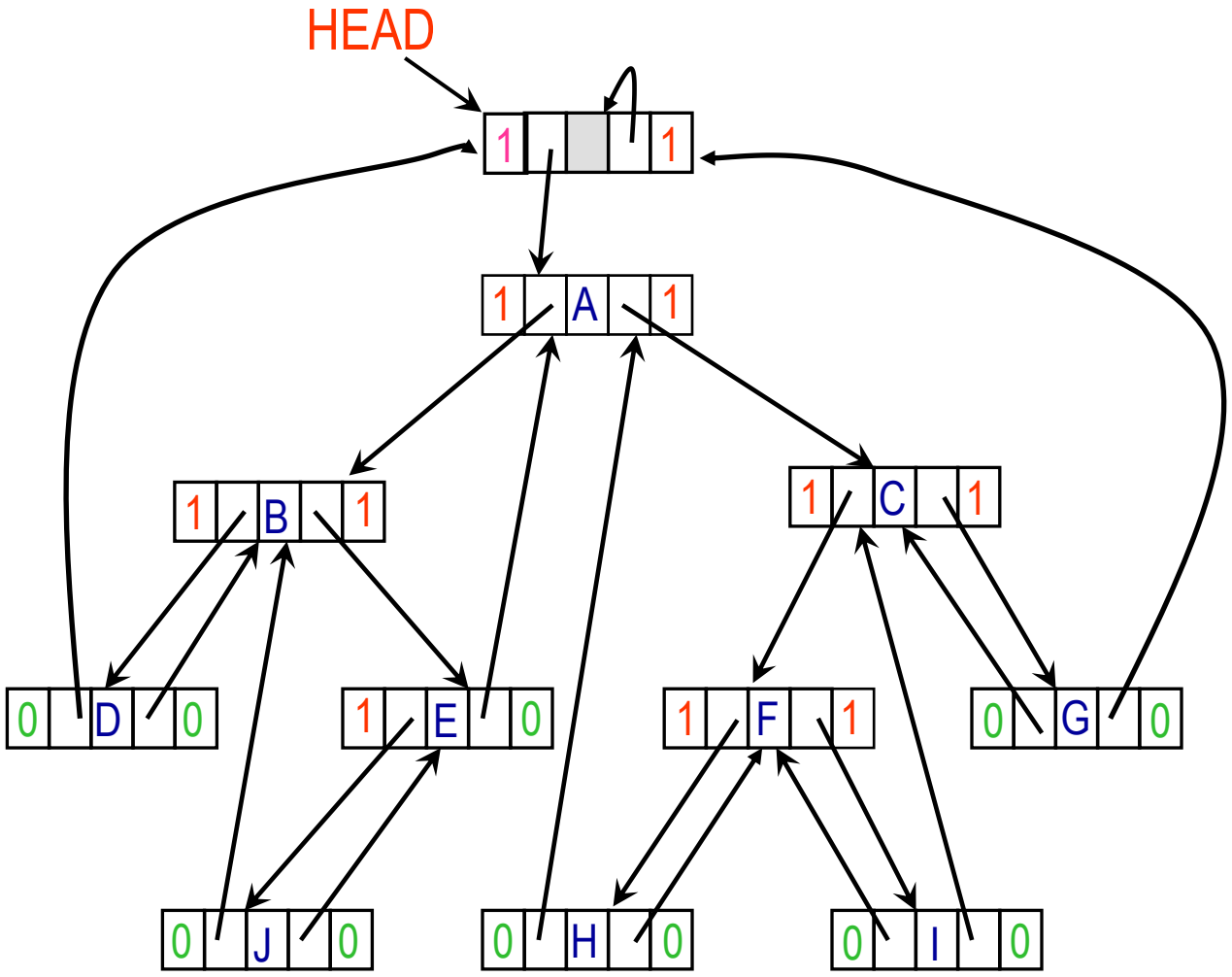
```
struct  node {
        Datatype   data;
        struct  node   *lchild,  *rchild;
        char  lbit, rbit;
} ;
typedef struct node TBTNode;
typedef struct node  *TBTNodeptr;
```

中序线索二叉树



中序序列　D, B, J, E, A, H, F, I, C, G

一棵完整的中序线索二叉树



6.4 线索二叉树的应用

中序序列:

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

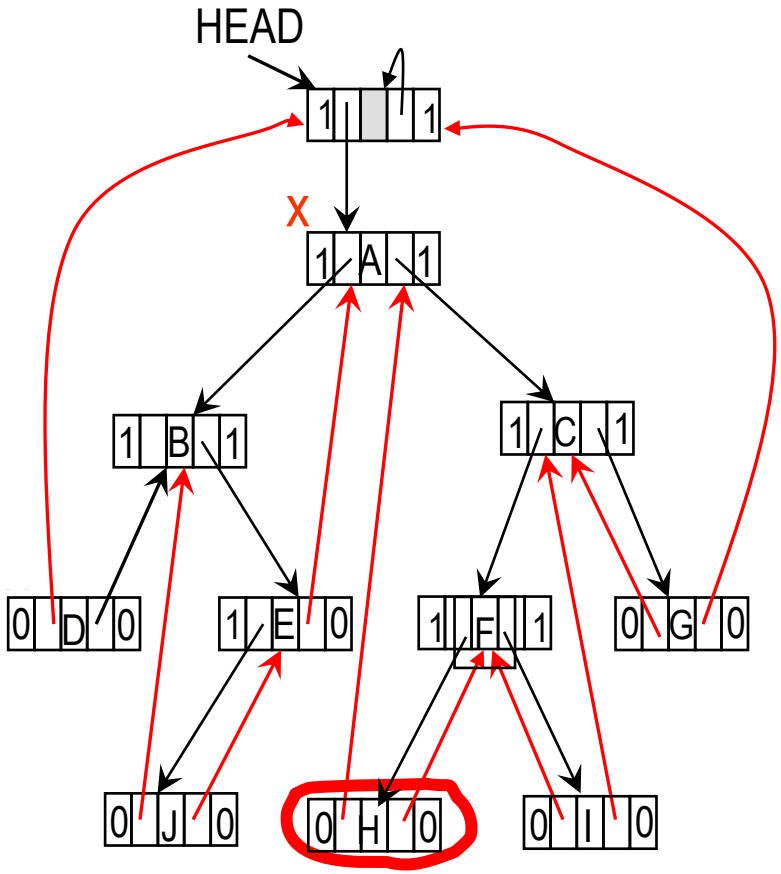
在中序线索二叉树中确定地址为x的结点的直接后继。

规律

(1) 当x->rbit=0时,x->rchild指出的结点就是x的直接后继结点。

(2) 当x->rbit=1时，沿着x的右子树的根的左子树方向查找,直到某结点的 lchild 域为线索时,此结点就是x结点直接后继结点。

该结点的lbit=0

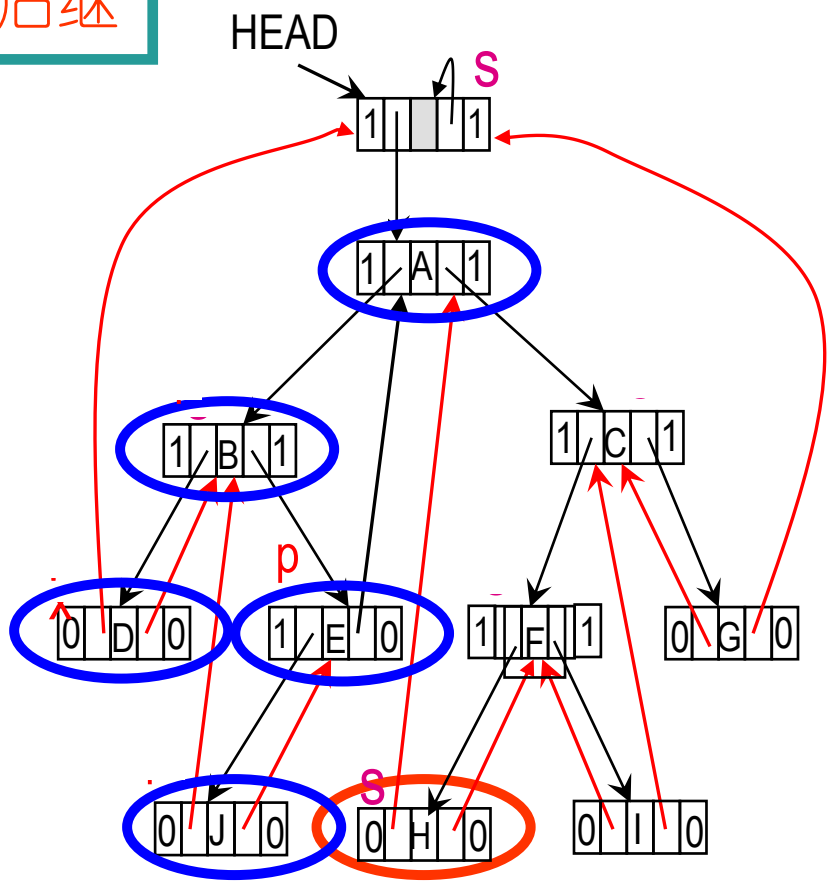


```
TBTNodeptr insucc(TBTNodeptr x)
{
    TBTNodeptr s;
    s=x->rchild;
    if(x->rbit==1)
        while (s->lbit==1)
            s=s->lchild;
    return(s);
}
```

确定X的直接后继

中序序列：

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




```
void torder(TBTNodeptr head)
{
    TBTNodeptr  p=head;
    while(1){
        p=insucc(p);
        if(p==head)
            break;
        VISIT(p);
    }
}
```

中序遍历

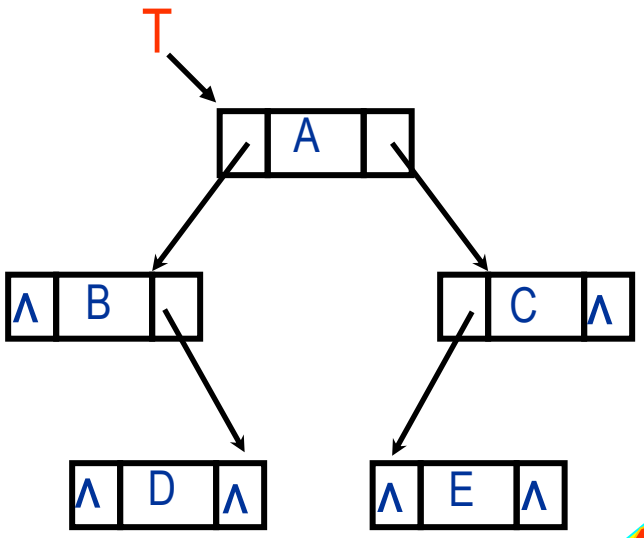
非递归

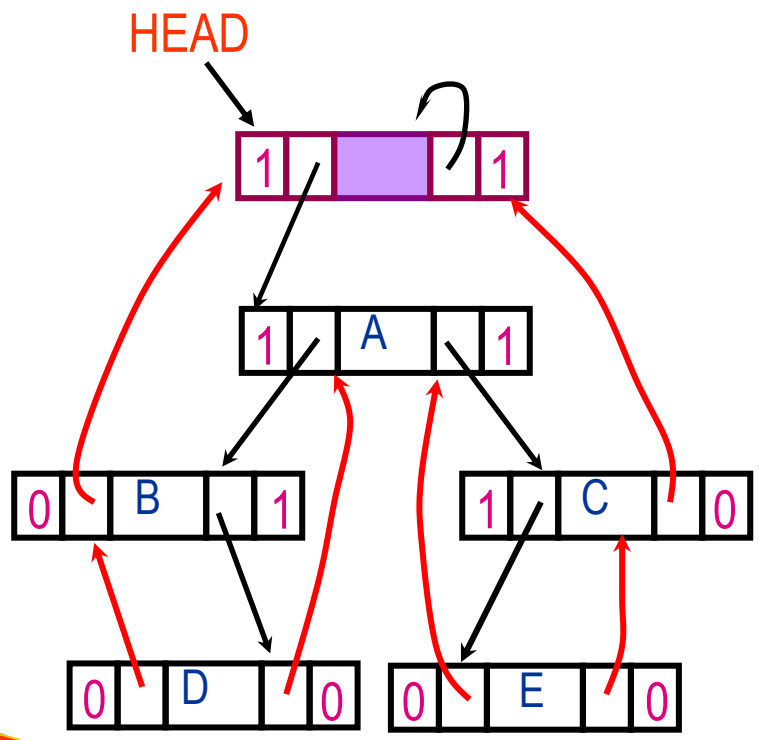
这是一种不使用栈的深度优先遍历非递归算法。

6.5 线索二叉树的建立

以中序线索二叉树为例

中序序列：B,D,A,E,C

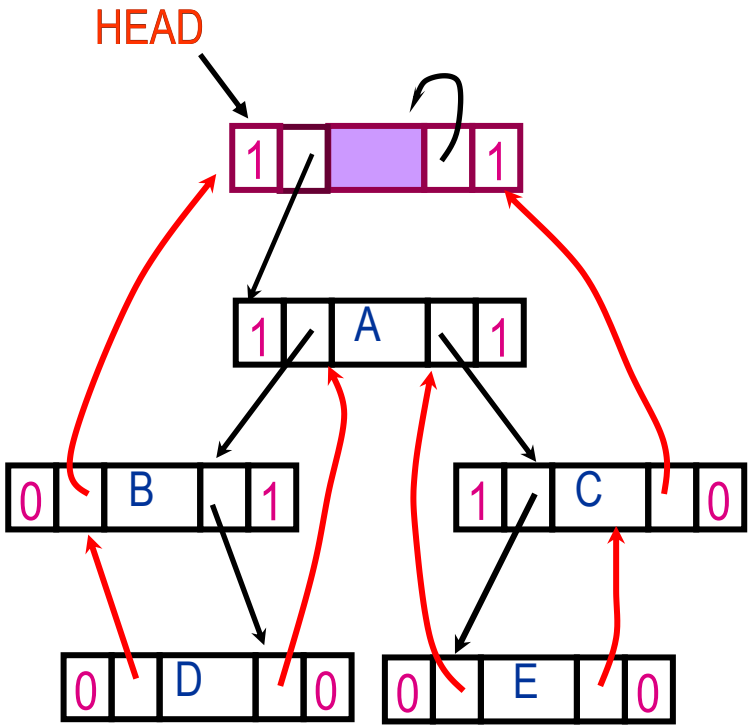




中序线索二叉树

在遍历过程中建立线索

prior: 指向前一次访问结点
p: 指向当前访问结点

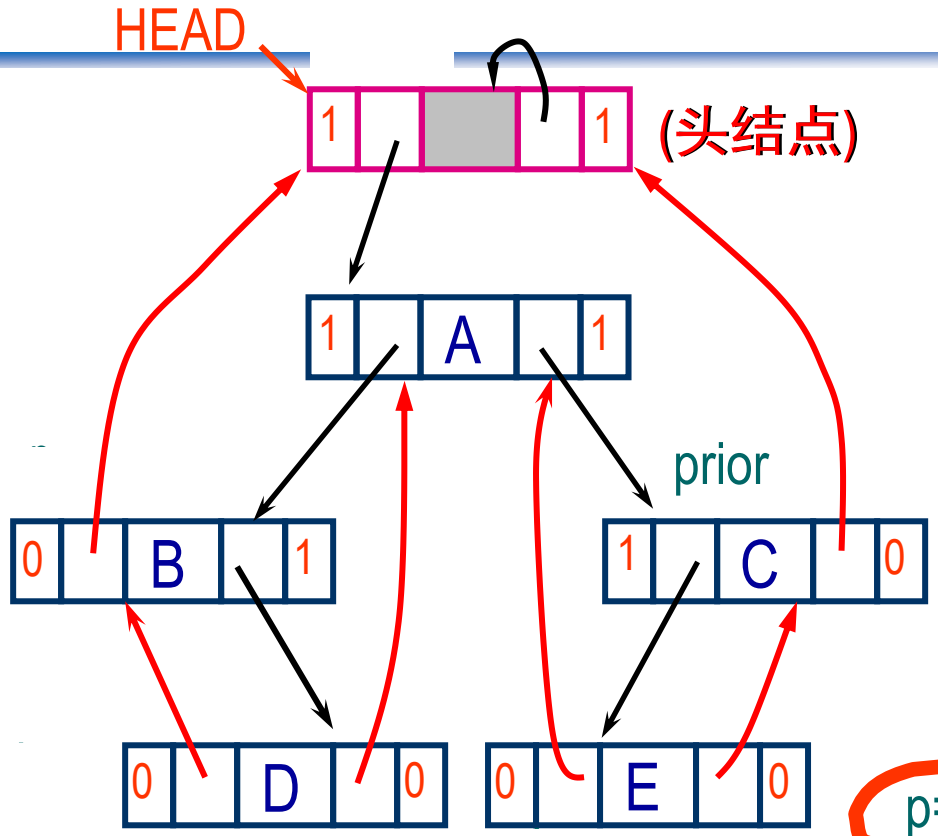


中序序列：B,D,A,E,C

建立线索的规律

- 若当前访问的结点p的**左指针域**为空，则它指向prior指的结点；
- 若prior所指结点的**右指针域**为空，则它指向当前访问的结点p。

中序线索二叉树



规律

- 若当前访问的结点的左指针域为空，则左指针域指向prior指的结点,同时置lbit为0, 否则, 置lbit为1；
- 若prior所指结点的右指针域为空, 则右指针域指向当前访问的结点, 同时置rbit为0, 否则, 置rbit为1。

p=NULL 遍历结束

prior: 指向前一次访问结点
p: 指向当前访问结点

中序序列：B,D,A,E,C

注意

遍历结束时，将prior->rchild
指向头结点，并置prior->rbit为0

建立线索

算法

```
/* 中序遍历进行中序线索化*/
/* piror是一个全局变量，初始时，piror指向树head结点*/
void inThreading(TBTNodeptr p)
{
   if(p != NULL) {
      inThreading(p->lchild)          //递归左子树线索化
      if(p->lchild == NULL) {         //没有左孩子
         p->lbit = 0;                 //前驱线索
         p->lchild = prior;           //左孩子指针指向前驱
      }
      else p->lbit = 1;
      if( prior->rchild == NULL) { //前驱没有右孩子
         prior->rbit = 0;             //后继线索
         prior->rchild = p;           //前驱右孩子指向后继
      }
      else p->rbit = 1;
      prior = p;                      //保持prior指向p的前驱
      inThreading(p->rchild);         //递归右子树线索化
   }
}
```

规律

- 若当前访问的结点的左指针域为空，则左指针域指向prior指的结点，同时置lbit为0，否则，置lbit为1(创建结点时缺省设计为1)；
- 若prior所指结点的右指针域为空，则右指针域指向当前访问的结点，同时置rbit为0，否则，置rbit为1（创建结点时缺省设计为1）。

该算法除了加粗部分外，与中序遍历树递归算法几乎完全一样。

```
TBTNodeptr  piror;
TBTNodeptr  threading(TBTNodeptr  root)
{
    TBTNodeptr  head;
    head = (TBTNodeptr)malloc(sizeof(TBTNode));
    head->lchild = root; head->rchild = head; head->lbit = head->rbit=1;
    piror = head;
    inThreading(root);
    piror->rchild = head; piror->rbit = 0;
    return head;
}
```

```
#define NodeNum   100                    /* 定义二叉树中结点最大数目*/ */
TBTNodeptr  inthread(TBTNodeptr t)
{  TBTNodeptr  head, p=t, prior, stack[NodeNum];
   int top=-1;
   head=(TBTNoteptr)malloc(sizeof(TBNode));  /* 申请线索二叉树的头结点空间 */
   head->lchild=t;   head->rchild=head;  head->lbit=1;
   prior=head;                          /* 假设中序序列的第1个结点的 “前驱” 为头结点 */
   do{
       for(; p!=NULL; p=p->lchild) /* p移到左孩子结点 */
            stack[++top]=p;          /* p指结点的地址进栈
       p=stack[top--];               /* 退栈 */
       if(p->lchild==NULL){          /* 若当前访问结点的左孩子为空 */
            p->lchild=prior;         /* 当前访问结点的左指针域指向前一次访问结点 */
            p->lbit=0;               /* 当前访问结点的左标志域置0(表示地址为线索) */
       }else
            p->lbit=1;               /* 当前访问结点的左标志域置1(表示地址为指针) */
       if(prior->rchild==NULL){      /* 若前一次访问的结点的右孩子为空 */
            prior->rchild=p;         /* 前一次访问结点的右指针域指向当前访问结点 */
            prior->rbit=0;           /* 前一次访问结点的右标志域置0(表示地址为线索) */
       }else
            prior->rbit=1;           /* 前一次访问结点的右标志域置1(表示地址为指针) */
       prior=p;                      /* 记录当前访问的结点的地址 */
       p=p->rchild;                  /* p移到右孩子结点 */
    }while(!(p==NULL && top==-1));
    prior->rchild=head;              /* 设中序序列的最后结点的后继为头结点 */
    prior->rbit=0;                   /* prior指结点的右标志域置0(表示地址为线索) */
    return head;                     /* 返回线索二叉树的头结点指针 */
}
```

建立线索

访问一个结点

非递归算法

二叉树线索化的好处：

其实线索化二叉树等于将一棵二叉树转变成了一个双向链表，这为二叉树结点的插入、删除和查找带来了方便。

在实际问题中，如果所用的二叉树需要经常遍历或查找结点时需要访问结点的前驱和后继，则采用线索二叉树结构是一个很好的选择。

将二叉树线索化可以实现不用栈的树深度优先遍历算法。