

# C Chapter 3 Trees



- About Trees
- Definition of binary Trees
- Implementation of binary Trees
- Trees and binary Trees
- Applications of binary Trees
- Binary search Tree
- AVL Trees
- B-Trees

## 3.1 Definition of Trees



### 3.1.1 Definition of Trees

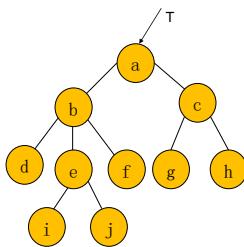
A Tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node  $r$ , called the **root**, and zero or more nonempty (**sub**) trees  $T_1, T_2, \dots, T_k$ , each of whose roots are connected by a directed edge from  $r$ .

没有空子树

### Expression of Trees

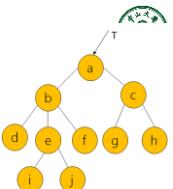
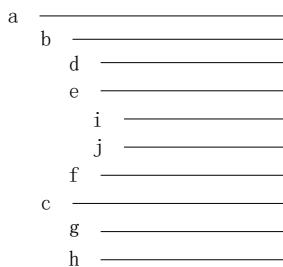


#### >Type of tree

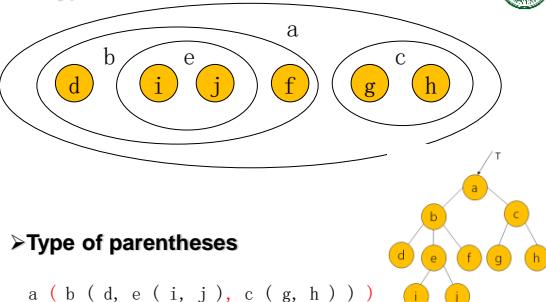


一对多的关系，反映了结点之间的层次关系。

#### Type of concave



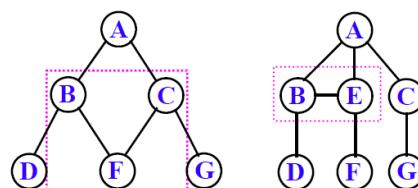
#### Type of set



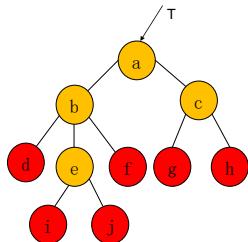
#### Type of parentheses

a ( b ( d, e ( i, j ), c ( g, h ) ) )

非树的结构



结点的度 (degree of node): 结点的子树个数



结点a的度为2  
结点b的度为3  
结点d的度为0

树的度: 树中结点度的最大值

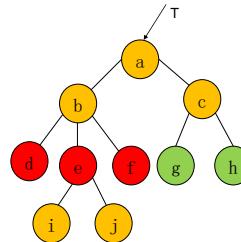
分支 (branch node) 结点: 度不为0的结点

叶 (leaf node) 结点: 度为0的结点

子女 (child node) 结点: 某结点子树的根结点

双亲 (parent node) 结点: 某个结点是其子树之根的双亲

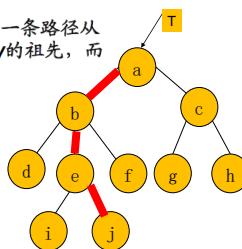
兄弟 (sibling) 结点: 具有同一双亲的所有结点



结点d,结点e和  
结点f互为兄弟

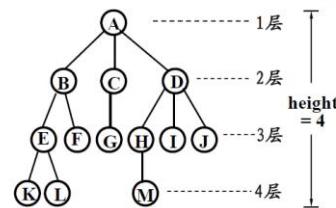
路(径)和路(径) 长度: 如果树的结点序列  $n_1, n_2, \dots, n_k$  有如下关系: 结点  $n_i$  是  $n_{i+1}$  的双亲 ( $1 \leq i < k$ ) , 则把  $n_1, n_2, \dots, n_k$  称为一条由  $n_1$  至  $n_k$  的路(径); 路径上经过的边的个数称为路径长度。

祖先、子孙: 在树中, 如果有一条路径从结点  $x$  到结点  $y$ , 那么  $x$  就称为  $y$  的祖先, 而  $y$  称为  $x$  的子孙。



结点的层数: 根结点的层数为1; 对其余任何结点, 若某结点在第  $k$  层, 则其孩子结点在第  $k+1$  层。

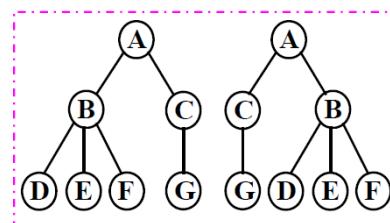
树的深度: 树中所有结点的最大层数, 也称高度。



- 有序树 子树的次序不能互换
- 无序树 子树的次序可以互换
- 森林 互不相交的树的集合



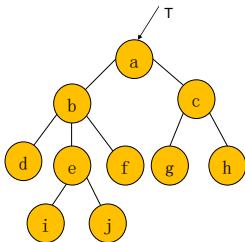
### 3.1.2 Operations of Trees



1. 求指定结点所在树的根结点
2. 求指定结点的双亲结点
3. 求指定结点的某一孩子结点
4. 求指定结点的最右边兄弟结点
5. 将一棵树插入到另一树的指定结点下作为它的子树
6. 删除指定结点的某一子树
7. 树的遍历



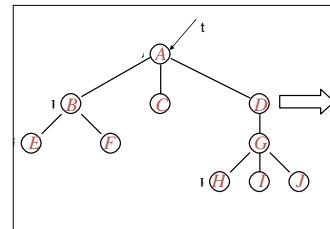
### 3.1.3 存储设计



数据元素的值  
父子关系  
兄弟关系  
子孙关系

#### 1. (Parents) 双亲表示法

树的双亲表示法主要描述的是结点与双亲的关系。



下标	info	paren
0	A	1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	3
7	H	6
8	I	6
9	J	6

#### 2. (Child) 孩子表示法

孩子表示法主要描述的是结点与孩子的关系。由于每个结点的孩子个数不定，所以利用链式存储结构更加适宜。



T	0   A	→ 2 → 1 → 4
	1   C	→ ^
	2   B	→ 3 → 5 ^
	3   E	→ ^
	4   D	→ 6 ^
	5   F	→ ^
	6   G	→ 7 → 8 → 9 ^
	7   H	→ ^
	8   I	→ ^
	9   J	→ ^

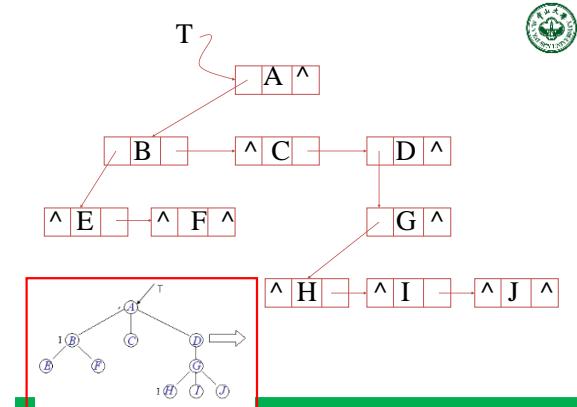


#### 3. (Child and brother) 孩子兄弟表示法

孩子兄弟表示法也是一种链式存储结构。它通过描述每个结点的一个孩子和兄弟信息来反映结点之间的层次关系，其结点结构为：



其中，firstchild为指向该结点第一个孩子的指针，nxtsibling为指向该结点的下一个兄弟，item是数据元素内容。



## 3.2 二叉树 (Binary Trees)



### 3.2.1 Definition

二叉树一个是 $n (n \geq 0)$ 个结点的有限集合，该集合或者为空（称为空二叉树）；或者是由一个根结点和两棵互不相交的、分别称为左子树和右子树的二叉树组成。

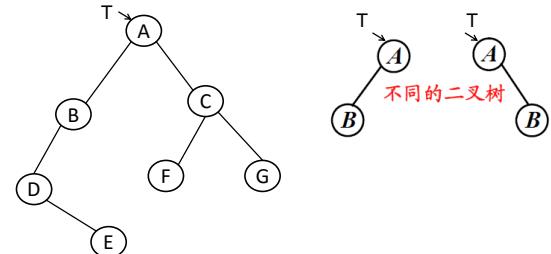
**结构特点：**

每个结点最多只有两棵子树，即结点的度不大于2。

子树有左右之别，子树的次序(位置)不能颠倒。

即使某结点只有一棵子树，也有左右之分。

一棵二叉树



一棵二叉树的基本形态



$\Phi$

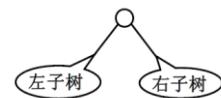
空二叉树

○

只有一个根结点

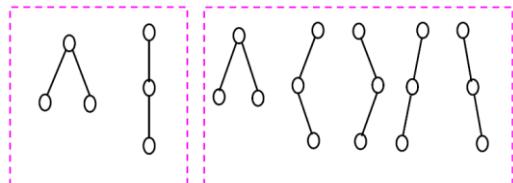


根结点只有左子树



根结点同时有左右子树

◆ 具有3个结点的树和二叉树的不同构的形态：



树的不同构形态

二叉树的不同构形态

特殊的二叉树----满二叉树

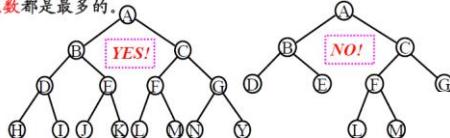
◆ 定义：高度为K且有 $2^k - 1$ 个结点的二叉树称为满二叉树。

◆ 结构特点：

- 分支结点都有两棵子树

- 叶子结点都在最后一层

◆ 满二叉树在相同高度的二叉树中，结点数、分支结点数和叶结点数都是最多的。

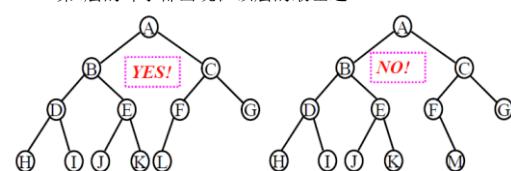


特殊的二叉树----完全二叉树

◆ 定义：满足下列性质的二叉树（假设树的高度为k）称为完全二叉树。

- ✓ 所有叶子结点都出现在第k层或k-1层

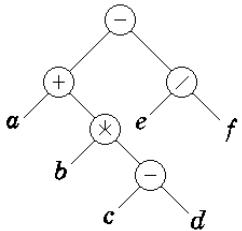
- ✓ 第k层的叶子都出现在该层的最左边



### 3.2.2 Properties of binary Trees



**Property 1** 若二叉树的层次从1开始，则在二叉树的第*i*层最多有 $2^{i-1}$ 个结点。 $(i \geq 1)$



**Property 2** 高度为*k*的二叉树最多有 $2^k - 1$ 个结点。

$$(k \geq 1)$$

证明：仅当每一层都含有最大结点数时，二叉树的结点数最多，利用性质1可得二叉树的结点数至多为：

$$2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{k-1} = 2^k - 1$$

**Property 3** 对任何一棵二叉树，如果其叶结点个数为*n*<sub>0</sub>，度为2的非叶结点个数为*n*<sub>2</sub>，则有

$$n_0 = n_2 + 1$$

证明：

1、结点总数为度为0的结点加上度为1的结点再加上度为2的结点：

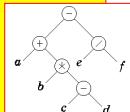
$$n = n_0 + n_1 + n_2$$

2、另一方面，二叉树中一度结点有一个孩子，二度结点有两个孩子，根结点不是任何结点的孩子，因此，结点总数为：

$$n = n_1 + 2n_2 + 1$$

3、两式相减，得到：

$$n_0 = n_2 + 1$$



**Property 5** 如果将一棵有*n*个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号1, 2, ..., *n*-1, *n*，然后按此结点编号将树中各结点顺序地存放于一个一维数组中，并简称编号为*i*的结点为结点*i* ( $1 \leq i \leq n$ )。则有以下关系：

- 若*i* = 1，则*i* 无双亲
- 若*i* > 1，则*i* 的双亲为*i*/2
- 若 $2^i \leq n$ ，则*i* 的左子女为 $2^i$ ；否则，*i* 无左子女，必定是叶结点，二叉树中 $i > \lfloor n/2 \rfloor$ 的结点必定是叶结点
- 若 $2^i+1 \leq n$ ，则*i* 的右子女为 $2^i+1$ ；否则，*i* 无右子女
- 若*i* 为奇数，且*i* 不为1，则其左兄弟为*i*-1，否则无左兄弟；若*i* 为偶数，且小于 *n*，则其右兄弟为*i*+1，否则无右兄弟
- *i* 所在层次为 $\lfloor \log_2(i) \rfloor + 1$



**Property 4** 具有*n*个结点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor + 1$

证明：设完全二叉树的高度为*h*，则有

$$2^{h-1} - 1 < n \leq 2^h - 1 \quad 2^{h-1} \leq n < 2^h$$

$$\text{取对数 } h - 1 < \log_2(n) < h$$



### 3.2.3 存储设计



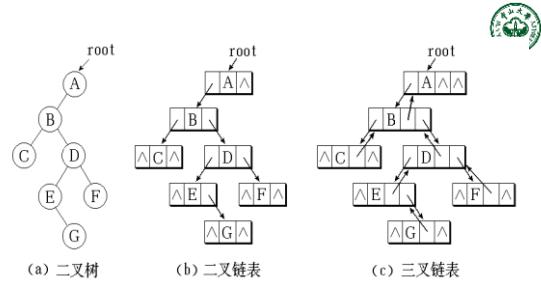
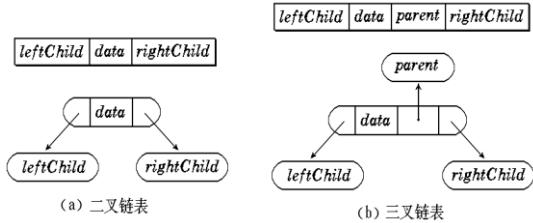
二叉树也可以采用两种存储方式：顺序存储结构和链式存储结构。

#### 1、Array

类似于树的顺序存储表示：父亲数组表示。



## 2、Link



## Cases of linked Trees



```

typedef int datatype;
typedef struct node
{ datatype data;
  struct node *lchild,*rchild;
} bitree;

```

## 3.3 Travel of Binary Trees



### 二叉树的遍历:

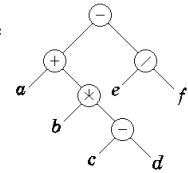
根据某种规则，按照一定的顺序访问树中的每一个结点，使得每个结点被访问且仅被访问一次。这个过程称为二叉树的遍历

按照子树和根的访问顺序，可以分为：

**中序遍历**

**前序遍历**

**后序遍历**



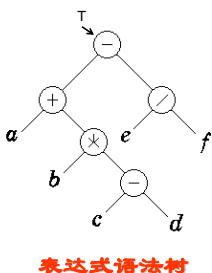
### 3.3.1 中序遍历

中序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 中序遍历左子树 (L)；
  - 访问根结点 (V)；
  - 中序遍历右子树 (R)。

遍历结果

$a + b * c - d - e / f$



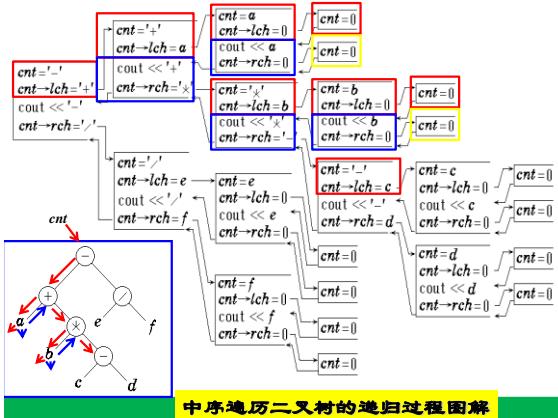
### Algorithm:

```

INORDER(bitree *t)
{
  if (t)
  {
    INORDER(t->lchild);
    printf("%c\n", t->data);
    INORDER(t->rchild);
  }
}

```



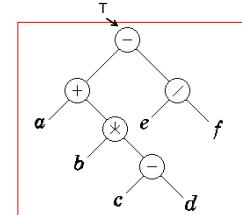


### 3.3.2 前序遍历

前序遍历二叉树算法的框架是

- 若二叉树为空，则空操作；
- 否则

- 访问根结点 (V);
- 前序遍历左子树 (L);
- 前序遍历右子树 (R)。



遍历结果

$- + a * b - c d / e f$

### 3.3.3 后序遍历

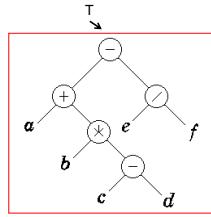


后序遍历二叉树算法的框架是

- 若二叉树为空，则空操作；
- 否则
  - 后序遍历左子树 (L);
  - 后序遍历右子树 (R);
  - 访问根结点 (V)。

遍历结果

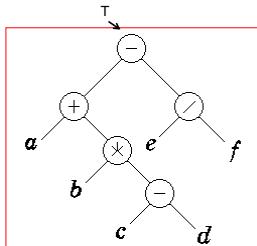
$a b c d - * + e f / -$



### 3.3.4 Applications



#### 1、在二叉树中查找具有给定值的结点



### 3.3.4 Applications



#### 2、按前序遍历次序打印二叉树中的叶子结点

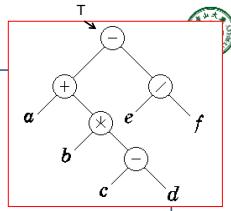


```
Pre_ORDER(bitree *t)
{
    if (t)
    {
        if (t->lchild==NULL & t->rchild==NULL)
            printf("%c\n", t->data);
        Pre_INORDER(t->lchild);
        Pre_INORDER(t->rchild);
    }
}
```

```
biter findnode(biter *t, datatype x)
{
    if (t == NULL) return(NULL);
    else if (t->data == x) return(t);
    else return( findnode(t->lchild) ||
                findnode(t->rchild) )
}
```

### 3、求二叉树的高度

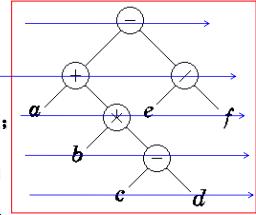
```
void depth(biter *t)
{
    int dep1, dep2;
    if (t == NULL) return(0);
    else
    {
        dep1 = depth(t->lchild);
        dep2 = depth(t->rchild);
        if (dep1 > dep2) return(dep1 + 1);
        else return(dep2 + 1); printf("(");
    }
}
```



### 4、层序遍历

层序遍历二叉树算法的框架是

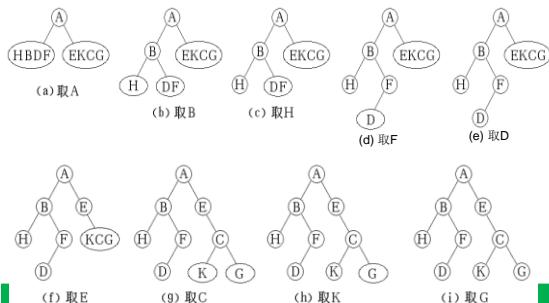
- 若二叉树为空，则空操作；
- 否则，如队列不空，循环：
- 根结点入队，并作为当前结点；  
    将当前结点的左右孩子入队；  
    做出队操作，队首元素作为当前结点
- 最后，出队序列就是层序遍历序列



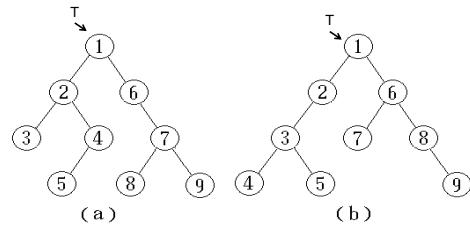
遍历结果  
- + / a \* e f b - c d

### 5、由二叉树的前序序列和中序序列可唯一地确定一棵二叉树。

例：前序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }



如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



前序序列： 1, 2, 3, 4, 5, 6, 7, 8, 9

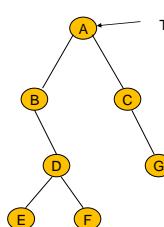
中序序列 a: 3, 2, 5, 4, 1, 6, 8, 7, 9

中序序列 b: 4, 3, 5, 2, 1, 7, 6, 8, 9

### 3.4 线索(thread)二叉树

当以二叉链表作为存储结构时，只能找到结点的左右孩子的信息，而不能确定结点在任一序列下的前驱与后继信息，这种信息只有在遍历的动态过程中才能得到。

中序遍历序列：BEDFACG



线索：二叉树中某种遍历下结点前驱（后继）的信息。

线索二叉树：加上线索的二叉树称之为线索二叉树。

线索二叉树的存储设计：

- 增加指针表示线索
- 增加标志域，利用原二叉树中的空指针表示线索。（多少个？）

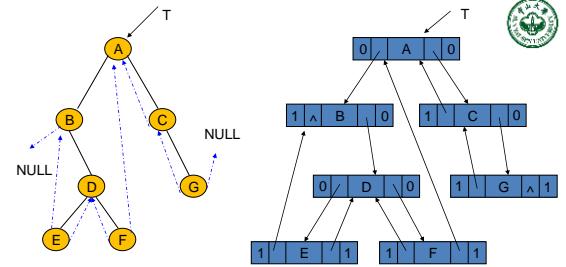
其中：

ltag= {	0 lchild 域指示结点的左孩子
1	lchild 域指示结点的前驱
rtag= {	0 rchild 域指示结点的右孩子
1	rchild 域指示结点的后驱

二叉树的二叉线索存储表示：

```
Typedef enum{Link, Thread} PointerTag;
Link= 0:指针, Thread= 1:线索
Typedef struct BiThrNode{
    TelemType data;
    struct BiTreeNode *lchild, *rchild; PointerTag
    LTag, Rtag;
}BiTreeNode, *BiThrTree;
```

lchild	ltag	data	rtag	rchild
--------	------	------	------	--------



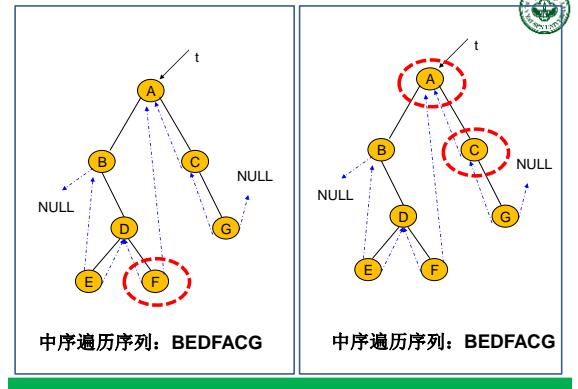
中序遍历序列：BEDFACG

### 线索化二叉树

- 对每个正在访问的结点t建立前驱线索和后继线索
- 而t的后继线索还没有出现

- 对每个正在访问的结点t进行前驱线索的修改
- 对pre (t的前驱结点) 的后继线索进行修改

中序遍历序列：BEDFACG



中序遍历序列：BEDFACG

中序遍历序列：BEDFACG

### 线索化二叉树

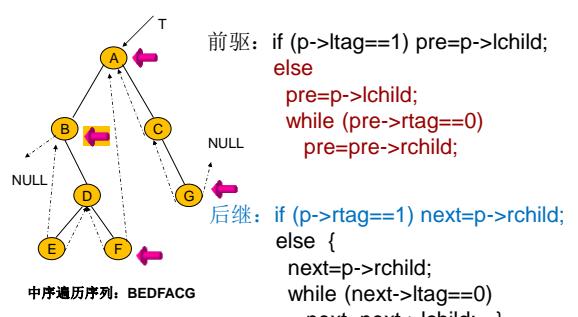
初始时, pre为空

```
INORDERThreading(btreetree *t)
{ if (t)
    { INORDERThreading (t->lchild);
      if (t->ltag==1) t->lchild=pre;
      if (pre!=NULL)&&(pre->rtag==1) NULL
          pre->rchild=t;
      pre=t;
      INORDERThreading (t->rchild);
    }
}
```

中序遍历序列：BEDFACG



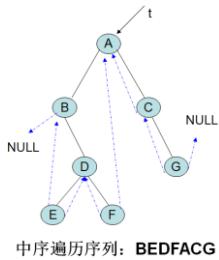
### 中序线索二叉树上前驱后继的查找



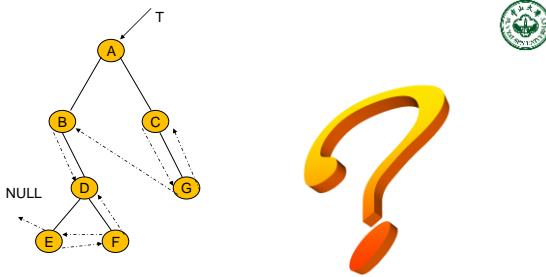
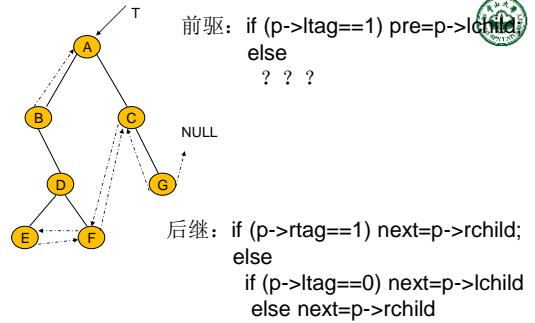
前驱: if (p->ltag==1) pre=p->lchild;  
else  
pre=p->lchild;  
while (pre->rtag==0)  
pre=pre->rchild;

后继: if (p->rtag==1) next=p->rchild;  
else {  
next=p->rchild;  
while (next->ltag==0)  
next=next->lchild; }

## 中序线索二叉树的遍历



```
Inorder(bitree T)
{ //查找二叉树的第一个结点
    p=;
    while (p->ltag==0) p=p->lchild;
    visit(p);
    //查找结点的后继
    while (p->rchild!=NULL)
    {
        if (p->rtag==1)
            { p=p->rchild; visit(p); }
        else { next=p->rchild;
                while (next->ltag==0)
                    next=next->lchild;
                visit(next); p=next; }
    }
}
```



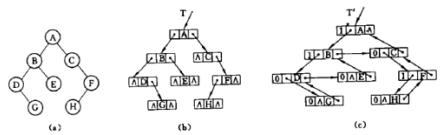
## 一种新的二叉树存储设计



二叉链表的优点是: 结构简单、直观, 基本操作易于实现;

缺点是:

- (1) 空间利用率低。含  $n$  个结点的二叉树共有  $2n$  个指针字段, 其中的  $n+1$  个字段的值为零;
- (2) 寻找给定结点的父/兄结点, 遍历二叉树等基本操作的时、空开销大。



设  $T$  是一棵二叉树(采用左、右链方式表示),

$T$  是  $T$  采用本文方法存储表示的二叉树。指针  $P$  指向二叉树中的结点。

若二叉树  $T$  为空, 则令  $T$  为空, 否则定义  $T'$  如下:

若结点  $p$  是  $T$  的根结点, 则在  $T'$  中令  $p->right = null$

若结点  $p$  是  $T$  的叶结点, 则在  $T'$  中令  $p->child = null$

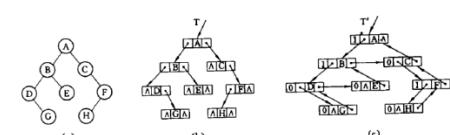
若结点  $p$  有且仅有一个子结点  $q$ , 则在  $T'$  中令  $p->child = q$ ,  $q->right = p$ ,

若结点  $p$  即有左子结点  $q$ , 又有右子结点  $r$ , 则在  $T'$  中令

$p->child = q$ ,  $q->right = r$ ,  $r->right = p$ ;

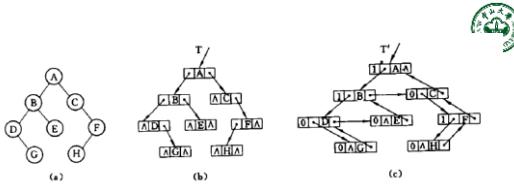
若结点  $p$  有左子结点  $q$ , 则在  $T'$  中令

$p->tag = 1$ , 否则  $p->tag = 0$ 。



容易验证  $T'$  具有如下重要的基本性质:

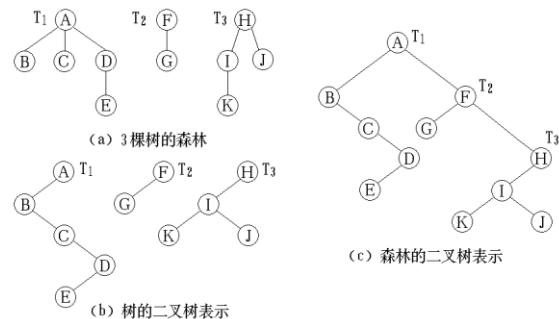
- (1)  $p$  是二叉树的根结点, 当且仅当  $p->right = null$
- (2)  $p$  是二叉树的叶结点, 当且仅当  $p->child = null$
- (3)  $p$  在是二叉树有左子结点, 当且仅当  $p->tag = 1$ , 进一步  $p$  的左子结点是  $p->child$
- (4)  $p$  在是二叉树有右子结点, 当且仅当  $(p->tag = 1, \text{且 } p->child->right \neq p)$  或  $(p->tag = 0, \text{且 } p->child->right \neq null)$   
进一步若  $p->tag = 1$  则  $p$  的右子结点是  $p->child->right$ , 否则是  $p->child$
- (5)  $p$  有父结点, 当且仅当  $p->right \neq null$



利用性质（1）或（2）可判断结点是否是根或叶结点。利用性质（3）或（4）可判断结点是否有左子树。

或右子女，并可找出孩子结点。利用性质（5）可判断结点是否有父结点，可找出其父结点并能断定。该结点是其父结点的左或右子女。

### 3.5 森林与二叉树的转换



#### (1) 森林转化成二叉树的规则

- ① 若  $F$  为空，即  $n = 0$ ，则对应的二叉树  $B$  为空二叉树。
- ② 若  $F$  不空，则对应二叉树  $B$  的 **root(B)** 是  $F$  中第一棵树  $T_1$  的 **root(T<sub>1</sub>)**；其 **左子树为 B (T<sub>11</sub>, T<sub>12</sub>, ..., T<sub>1m</sub>)**，其中， $T_{11}, T_{12}, \dots, T_{1m}$  是 **root(T<sub>1</sub>)** 的子树；其 **右子树为 B (T<sub>2</sub>, T<sub>3</sub>, ..., T<sub>n</sub>)**，其中， $T_2, T_3, \dots, T_n$  是除  $T_1$  外其它树构成的森林。

#### (2) 二叉树转换为森林的规则

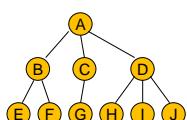
- ① 如果  $B$  为空，则对应的森林  $F$  也为空。
- ② 如果  $B$  非空，则  $F$  中第一棵树  $T_1$  的根为 **root**； $T_1$  的根的子树森林  $\{T_{11}, T_{12}, \dots, T_{1m}\}$  是由 **root** 的左子树 **LB** 转换而来， $F$  中除了  $T_1$  之外其余的树组成的森林  $\{T_2, T_3, \dots, T_n\}$  是由 **root** 的右子树 **RB** 转换而成的森林。

#### 树的遍历

##### 前序遍历

若树非空，则

- 1、访问根结点
- 2、依次前序遍历树的各子树



##### 遍历序列：

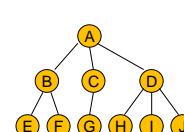
A, B, E, F, C, G, D, H, I, J



##### 后序遍历

若树非空，则

- 1、依次后序遍历树的各子树
- 2、访问根结点



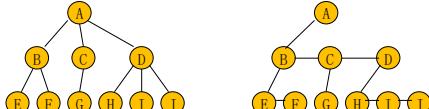
##### 遍历序列：

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



注意：

前序遍历一棵树等价于前序遍历该树对应的二叉树  
后序遍历一棵树等价于中序遍历该树对应的二叉树



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



### 3.6 哈夫曼树(Huffman)及其应用

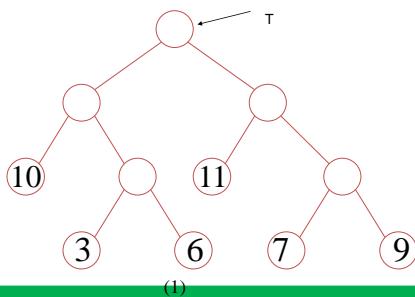


在二叉树中，一个结点到另一个结点之间的分支构成这两个结点之间的路径。树的路径长度是从树根到每片叶子结点的路径长度之和。

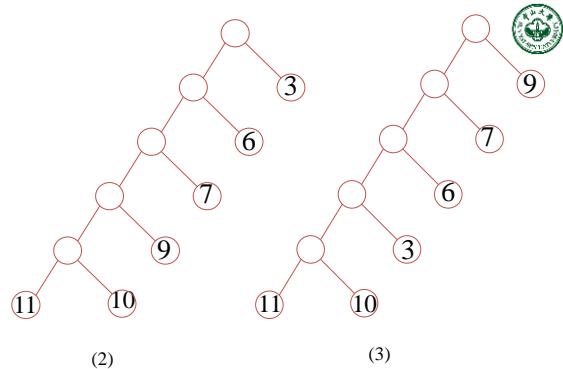
树的带权路径长度可以用公式表示为：

$$WPL = \sum_{k=1}^n w_k l_k$$

权值、树形与带权的路径长度之间的关系。  
假设有6个权值分别为{3, 6, 9, 10, 7, 11}，以这6个权值作为叶子结点的权值可以构造出下面三棵二叉树。



(1)



(2)

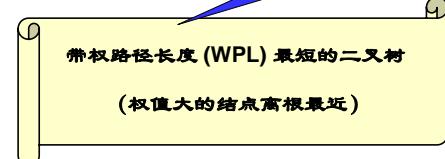
(3)

- 这三棵二叉树的带权路径长度分别为：
- $WPL_1 = 10*2 + 11*2 + 3*3 + 6*3 + 7*3 + 9*3 = 117$
- $WPL_2 = 3*1 + 6*2 + 7*3 + 9*4 + 10*5 + 11*5 = 177$
- $WPL_3 = 9*1 + 7*2 + 6*3 + 3*4 + 10*5 + 11*5 = 158$



哈夫曼树：

目标



因为构造这种树的算法是由哈夫曼于 1952 年提出的，

所以被称为哈夫曼树，相应的算法称为哈夫曼算法。



构造哈夫曼树的过程：

- (1) 将给定的  $n$  个权值  $\{w_1, w_2, \dots, w_n\}$  作为  $n$  个根结点的权值构造一个具有  $n$  棵二叉树的森林  $\{T_1, T_2, \dots, T_n\}$ , 其中每棵二叉树只有一个根结点;
- (2) 在森林中选取两棵根结点权值最小的二叉树作为左右子树构造一棵新二叉树, 新二叉树的根结点权值为这两棵二叉树根的权值之和;
- (3) 在森林中, 将上面选择的这两棵根权值最小的二叉树从森林中删除, 并将刚刚构造的二叉树加入到森林中;
- (4) 重复上面 (2) 和 (3), 直到森林中只有一棵二叉树为止。这棵二叉树就是哈夫曼树。

假设有一组权值  $\{5, 29, 7, 8, 14, 23, 3, 11\}$ , 下面我们将利用这组权值演示构造哈夫曼树的过程。

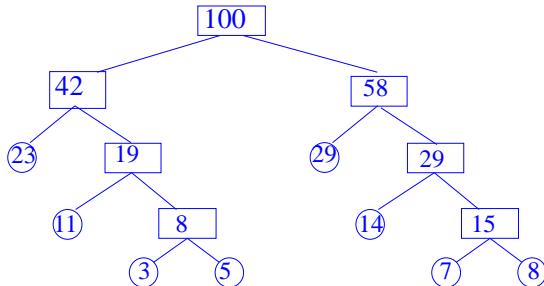
第一步：以这8个权值作为根结点的权值构造具有8棵树的森林

⑤ ②9 ⑦ ⑧ ⑭ ⑬ ③ ⑪



## 哈夫曼树的特点

树中没有分支为一的结点;  
 $m$  片叶子的哈夫曼树具有  $2m-1$  个结点  
 哈夫曼树的形态不唯一  
 权值较大的结点离根较近,  
 权值较小的结点离根较远。



$$WPL = (23+29)*2 + (11+14)*3 + (3+5+7+8)*4 = 271$$

## Huffman 算法描述

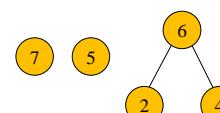


HT

	结点	双亲	左孩子	右孩子
1	7	0	0	0
2	5	0	0	0
3	2	0	0	0
4	4	0	0	0
5				
6				
7				

构造森林

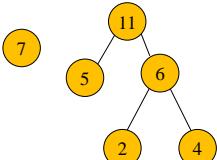
```
For (i=1;i<=m;i++)
{
    HT[i].data=w[i];
    HT[i].Parent=0;
    HT[i].Lchild=0;
    HT[i].Rchild=0;
}
```



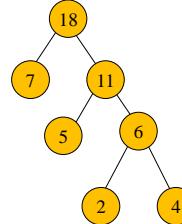
构造树中的结点

```
HT
For (i=m+1;i<=2*m-1;i++)
{
    Select(HT,i-1,s2);
    HT[s1].Parent=i;
    HT[s2].Parent=i;
    HT[i].Rchild=s1;
    HT[i].Lchild=s2;
    HT[i].Parent=0;
    HT[i].data=HT[s1].data+
    HT[s2].data;
}
```

	结点	双亲	左孩子	右孩子
1	7	0	0	0
2	5	0	0	0
3	2	5	0	0
4	4	5	0	0
5	6	0	3	4
6				
7				



HT	结点	双亲	左孩子	右孩子
1	7	0	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
6	11	0	2	5
7				



HT	结点	双亲	左孩子	右孩子
1	7	7	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
6	11	7	2	5
7	18	0	1	6

数组中最后一个元素是根结点  
数组前m个元素是叶子

## 霍夫曼编码



主要用途是实现数据压缩。

设给出一段报文：

CAST CAST SAT AT A TASA

字符集合是 { C, A, S, T }，各个字符出现的频度(次数)是  $W = \{ 2, 7, 4, 5 \}$ 。

若给每个字符以等长编码

A : 00 T : 10 C : 01 S : 11  
CAST CAST SAT AT A TASA  
01001110 01001110 110010 0010 10001100

则总编码长度为  $(2+7+4+5) * 2 = 36$ 。

若给每个字符以不等长编码

A : 0 T : 10 C : 110 S : 111

CAST CAST SAT AT A TASA

110011110 110011110 111010 010 0 1001110

它的总编码长度： $7*1+5*2+(2+4)*3 = 35$ 。比等长编码的情形要短。

若编码为：

A : 0 T : 01 C : 110 S : 011

CAST CAST SAT AT A TASA

110001101110001101011001001001000110

它的总编码长度： $7*1+5*2+(2+4)*3 = 35$ 。

## 前缀编码



- 一个编码集合中，任何一个字符的编码都不是另外一个字符编码的前缀，这种编码叫作前缀编码
- 这种前缀特性保证了代码串被反编码时，不会有多种可能。例如，
  - 对于上面 8 个字符，编码为 Z(111100), K(111101), F(11111), C(1110), U(100), D(101), L(110), E(0)。
  - 这是一种前缀编码，对于代码“000110”，可以翻译出唯一的字符串“EEEL”

**最优前缀码：**使得平均编码长度最小的前缀编码称为最优的前缀码。

对于给定的字符集及其每个字符出现的概率（使用频度），求该字符集的**最优的前缀性编码—哈夫曼编码问题**。

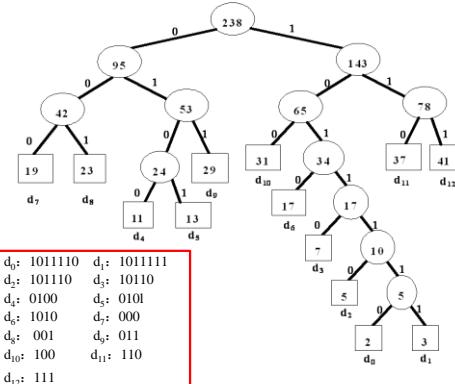


## 用哈夫曼算法求字符集最优前缀编码

使字符集中的每个字符对应一棵只有叶结点的二叉树，叶的权值为对应字符的使用频率---初始化；

利用Huffman算法来构造一棵Huffman树---构造算法；

对Huffman树上的每个结点，左分枝标以0，右分枝标以1（或者相反），则从根到叶的路上的0、1序列就是相应字符的编码---哈夫曼编码。



假设有一个电文字符集中有8个字符，每个字符的使用频率分别为 $\{0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11\}$ ，现以此为例设计哈夫曼编码。

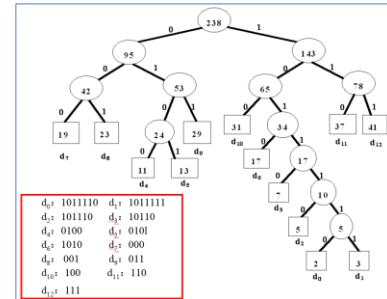
0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11

0.08, 0.29, 0.07, 0.08, 0.14, 0.23, 0.11

0.15, 0.29, 0.08, 0.14, 0.23, 0.11

0.15, 0.29, 0.19, 0.14, 0.23

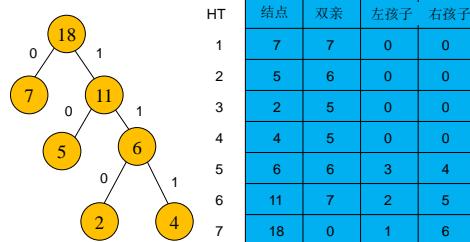
## 哈夫曼编码算法的实现



为每片叶子进行编码，其编码为从根到叶子的路径上的0或1。



## 哈夫曼编码算法的实现



结点1->双亲为结点7，是其左孩子，因此编码为0，结点7为根，结束编码。（0）

结点3->双亲为结点5，是其左孩子，因此编码为0。

结点5->双亲为结点6，是其右孩子，因此编码为1。

结点6->双亲为结点7，是其右孩子，因此编码为1，结点7为根，结束编码。（110）

## 哈夫曼编码算法的实现

```

void CharSetHuffmanEncoding( HuffmanTree T, HuffmanCode H ) {
    /*根据Huffman树T求Huffman编码表 H*/
    int c, p, i;           /*c 和p 分别指示T 中孩子和双亲的位置 */
    char cd[n+1];          /*临时存放编码 */
    int start;              /*指示编码在cd 中的位置 */
    cd[n]=0';               /*编码结束符 */
    for( i=0; i<n; i++ ){ /*依次求叶子T[i]的编码 */
        H[i].ch=getchar(); /*读入叶子T[i]对应的字符 */
        start=i;            /*编码起始位置的初值 */
        c = i;               /*从叶子T[i]开始上溯 */
        while( (p=T[c].parent)>=0){ /*直到上溯到T[c]是树根位置 */
            cd[i]=start=(T[p].lchild==c)? '0': '1';
            /*若T[c]是T[p]的左孩子，则生成代码0，否则生成代码1*/
            c=p;               /*继续上溯 */
        }
        strcpy(H[i].bits,&cd[start]); /*复制编码为串于编码表H*/
    }
}

```





## 3.7 二叉查找树



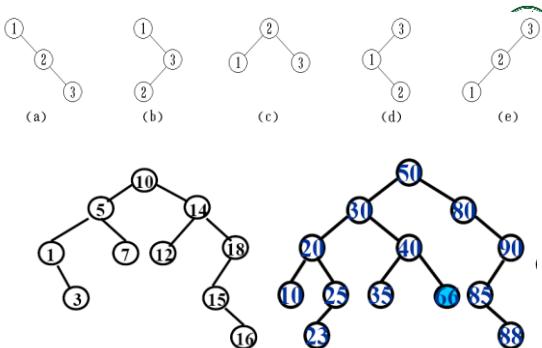
### ( Binary Search Trees )

#### 定义

二叉查找树（二叉排序树）或者是一棵空树，或者是具有下列性质的二叉树：

- 每个结点都有一个作为查找依据的关键字(key)，所有结点的关键字互不相同。
- 左子树(若非空)上所有结点的关键字都小于根结点的关键字。
- 右子树(若非空)上所有结点的关键字都大于根结点的关键字。
- **左子树和右子树也是二叉查找树。**

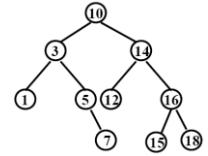
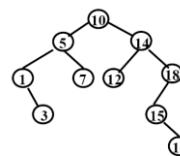
**译码：**依次读入文件的二进制码，在Huffman树中从根结点出发，若读入0，则走左支，否则，走右支，一旦到达某叶结点时便译出相应的字符。然后重新从根出发继续译码，直到文件结束。



几个二叉查找树的例子

#### 二叉查找树的结构特点：

- 任意一个结点的关键字，都大于(小于)其左(右)子树中任意结点的关键字，因此各结点的关键字互不相同
- 按中序遍历二叉查找树所得的中序序列是一个递增的有序序列，因此，二叉查找树可以把无序序列变为有序序列。
- 同一个数据集合，可按关键字表示成不同的二叉查找树，即同一数据集合的二叉查找树不唯一；但中序序列相同。



#### 二叉查找树的数据类型描述



```

typedef struct { int key; } ElemtType;
typedef struct BiTNode
{
    ElemtType data;
    struct BiTNode *lchild,*rchild;
} BiTNode, * BiTree;
  
```

#### (1) 二叉查找树的查找操作：

在T中查找关键字为k的记录如下：

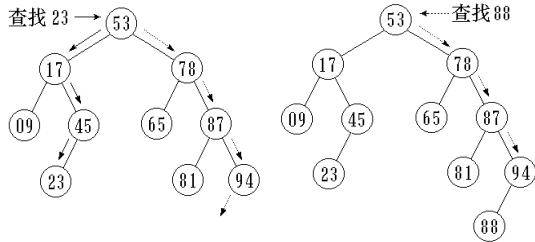
若T = Null，则查找失败；否则，

k == T->data.key，则查找成功；否则，

k < T->data.key，则**递归地**在T的左子树查找k；否则

k > T->data.key，则**递归地**在T的右子树查找k。

## 二叉树的查找过程



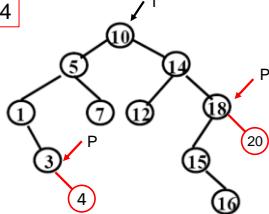
## 二叉树的查找算法

```
BSTNode * SearchBST(keytype k, BST T)
{ BSTNode * p = T;
if (p == Null || k == p->data.key) // 递归终止条件
return p;
if (k < p->data.key)
return (SearchBST(k, p->lchild)); // 查找左子树
else
return (SearchBST(k, p->rchild)); // 查找右子树
}
```

## (2) 二叉查找树的插入算法

若二叉排序树为空树，则新插入的结点为根结点；  
否则，新插入的结点必为一个新的叶结点。

插入 20, 4



## (2) 二叉查找树的插入算法

将指针 S 所指向的结点插入到根结点指针为 T 的二叉查找树中  
void Insert\_BST(BiTree &T, BiTree S)

```
{ BiTree p, q;
if (!T) T=S;
else { p=T;
while (p)
{ q=p;
if (S->data.key < p->data.key) p=p->lchild;
else p=p->rchild;
}
if (S->data.key < q->data.key) q->lchild=S;
else q->rchild=S;
}
return; }
```

时间复杂性  $O(\log n)$

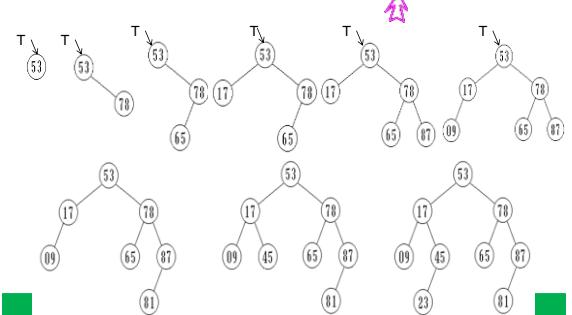
## (3) 二叉查找树上的构建

### 1) 构造过程：

- 从空树出发，依次插入  $R_1 \sim R_n$  各数据值：
- (1) 如果二叉查找树是空树，则插入结点就是二叉查找树的根结点；
  - (2) 如果二叉查找树是非空的，则插入值与根结点比较，若小于根结点的值，就插入到左子树中去；否则插入到右子树中。

## 输入数据，建立二叉查找树的过程

输入数据序列 { 53, 78, 65, 17, 87, 09, 81, 45, 23 }

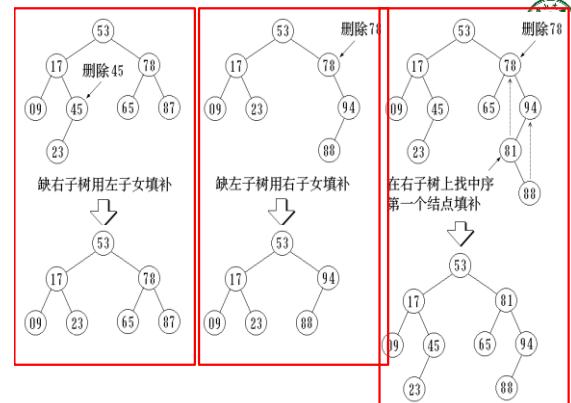


## (4) 二叉查找树的删除



- 删除二叉查找树的叶子结点
- 删除二叉查找树的根结点的算法

- (1) 根结点有左右子树的情况下，选择根结点的左子树中的最大结点为新的根结点；或者是右子树中的最小结点为新的根结点；
- (2) 如果根结点没有左子树，则以右子树的根结点作为新的根结点；
- (3) 如果根结点没有右子树，则以左子树的根结点作为新的根结点



### 删除二叉查找树中结点的算法描述



——寻找被删除的结点

```
int Delete_BST( BiTree &T,int key)
{ BiTree p,f;
  p=T; f=NULL;
  while(p)
    { if (p->data.key == key) { delNode ( T, p, f ) ; return(1); }
      else if (p->data.key > key) { f=p; p=p->lchild; }
      else { f=p; p=p->rchild; }
    }
  return(0)
}
```

### 删除二叉查找树中结点的算法 —— 删除找到的结点

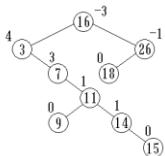


```
void delNode ( BiTree &T, BiTree p, BiTree f )
{ BiTree s, q ;
  int tag ; tag=0;
  if (!p->lchild) s=p->rchild;
  else if (!p->rchild) s=p->lchild;
  else{ q=p; s=p->lchild;
    while(s->rchild) { q=s; s=s->rchild; } //左子树中最大的
    p->data=s->data;
    if (q==p) q->lchild=s->lchild;
    else q->rchild=s->lchild;
    free(s);
    tag=1; }
  if (!tag){ if (!f) T=s;;
    else if (f->lchild==p) f->lchild=s;
    else f->rchild=s;
    free(p);
    return;
  }
}
```

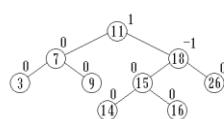
## 3.8 AVL树 高度平衡的二叉查找树



一棵AVL树或者是空树，或者是具有下列性质的二叉查找树：它的左子树和右子树都是AVL树，且左子树和右子树的高度之差的绝对值不超过1。



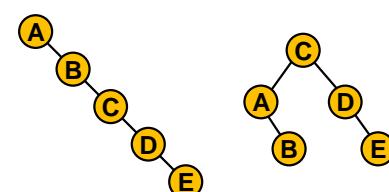
高度不平衡的二叉排序树



高度平衡的二叉查找树

➤ 二叉查找树的高度越小，平均查找长度越小。

➤  $n$  个结点的二叉查找树的高度最大为  $n-1$ ，最小为  $\lfloor \log_2 n \rfloor$ 。



## 结点的平衡因子balance (balance factor)

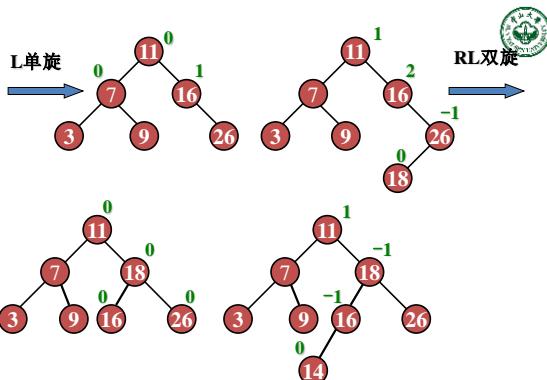
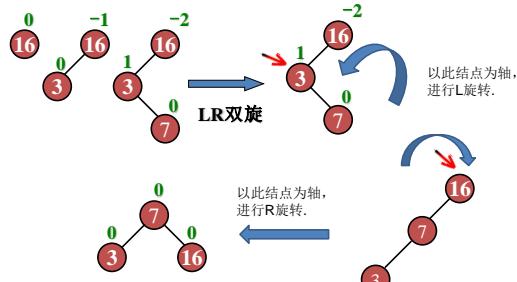
- 每个结点附加一个数字，给出该结点右子树的高度减去左子树的高度所得的高度差。这个数字即为结点的平衡因子**balance**。
- 根据AVL树的定义，任一结点的平衡因子只能取 **-1, 0 和 1**。
- 如果一个结点的平衡因子的绝对值大于1，则这棵二叉查找树就失去了平衡，不再是AVL树。
- 如果一棵二叉查找树是高度平衡的，它就成为AVL树。如果它有  $n$  个结点，其高度可保持在  $O(\log_2 n)$ ，平均查找长度也可保持在  $O(\log_2 n)$ 。



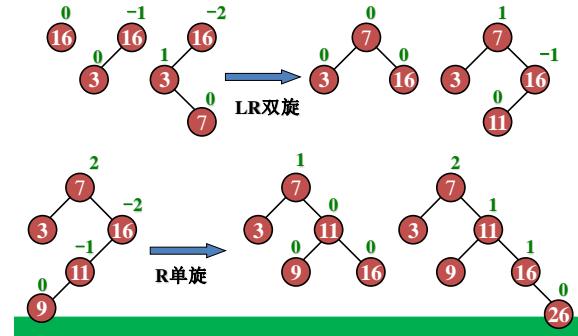
## AVL树的插入

- 在向一棵本来是高度平衡的AVL树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值  $|balance| > 1$ ，则出现了不平衡，需要做平衡化处理。
- 算法从一棵空树开始，通过输入一系列对象的关键字，逐步建立AVL树。在插入新结点时使用了前面所给的算法进行平衡旋转。

例，输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 } 插入和调整过程如下。



例，输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 } 插入和调整过程如下。



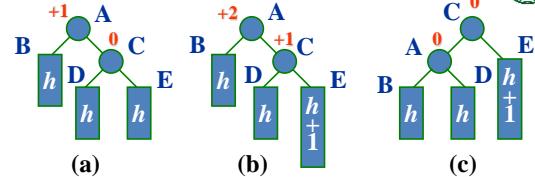
从空树开始的建树过程

### 平衡化旋转

- 如果在一棵平衡的二叉查找树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 每插入一个新结点时，AVL树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子(左、右子树的高度差)。



#### 左单旋转

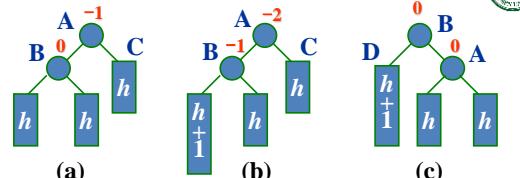


- 如果在子树E中插入一个新结点，该子树高度增1导致结点A的平衡因子变成+2，出现不平衡。
- 沿插入路径检查三个结点A、C和E。它们处于一条方向为“\”的直线上，需要做左单旋转。
- 以结点C为旋转轴，让结点A反时针旋转。

```
void L_Rotate(BSTree &p){
    //左单旋转的算法
    BSTree rc;
    rc=p->rchild;
    p->rchild=rc->lchild;
    rc->lchild=p; p=rc;
}
```



#### 右单旋转



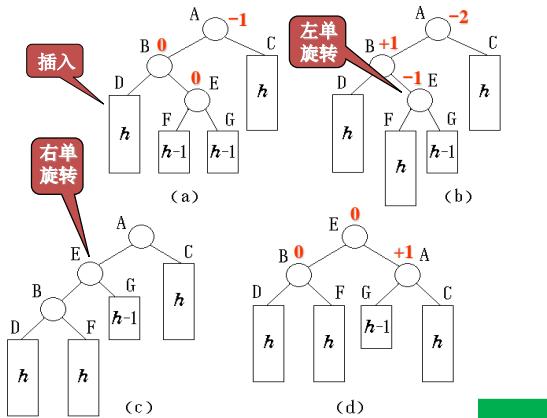
- 在左子树D上插入新结点使其高度增1，导致结点A的平衡因子增到-2，造成了不平衡。
- 为使树恢复平衡，从A沿插入路径连续取3个结点A、B和D，它们处于一条方向为“/”的直线上，需要做右单旋转。
- 以结点B为旋转轴，将结点A顺时针旋转。

```
void R_Rotate(BSTree &p){
    //右单旋转的算法
    BSTree lc;
    lc=p->lchild;
    p->lchild=lc->rchild;
    lc->rchild=p; p=lc;
}
```



#### 先左后右双旋转 (LR)

- 在子树F或G中插入新结点，该子树的高度增1。结点A的平衡因子变为-2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、B和E，它们位于一条形如“(”的折线上，因此需要进行先左后右的双旋转。
- 首先以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置，做左单旋转。
- 再以结点E为旋转轴，将结点A顺时针旋转，做右单旋转。使之平衡化。



void LeftBalance(BSTree &T){ //左平衡化的算法

```

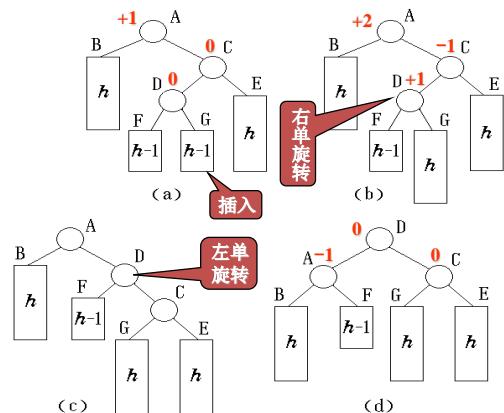
    BSTree lc,rd; lc=T->lchild;
    switch(lc->bf){
        case LH: T->bf = lc->bf = EH;
            R_Rotate(T); break;
        case RH: rd=lc->rchild;
            switch(rd->bf){
                case LH: T->bf=RH; lc->bf=EH; break;
                case EH: T->bf=lc->bf=EH; break;
                case RH: T->bf = EH; lc->bf=LH; break;
            }
            rd->bf=EH;
            L_Rotate(T->lchild);
            R_Rotate(T);
    }
}

```

### 先右后左双旋转 (RL)



- 右左双旋转是左右双旋转的镜像。
- 在子树F或G中插入新结点，该子树高度增1。结点A的平衡因子变为2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、C和D，它们位于一条形如“Y”的折线上，需要进行先右后左的双旋转。
- 首先做右单旋转：以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。
- 再做左单旋转：以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。



## 分析



根据平衡树的定义，平衡树上所有结点的平衡因子的绝对值都不超过1。在插入结点之后，若查找树上某个结点的平衡因子的绝对值大于1，则说明出现不平衡。同时，失去平衡的最小子树的根结点必然离插入结点最近，其平衡因子的绝对值在插入之前大于零。

## 分析



为此，需要做到以下几点：

- 在查找S结点的插入位置过程中，记录与S结点最近，且平衡因子不等于零的结点a。
- 修改自a到S的路径上所有结点的平衡因子值。
- 判断树是否出现不平衡，即a的平衡因子是否大于1。
- 若出现不平衡，进行平衡调整。



### Status InsertAVL (BSTree & T, ElemtType s)

```

{ if (T==NULL) T=s;
else {
    f=NULL; //记录a的双亲
    a=T; //记录在查找中平衡因子不为零的结点
    p=T; //进行比较的结点
    q=NULL; //记录p的双亲
    While (p!=NULL) //查找结点
    { if (p->bf!=0) { a=p; f=q; }
      q=p;
      if (s->data<p->data) p=p->Lchild;
      else p=p->Rchild;
    }
}

```

```

if (s->data<q->data) q->Lchild=s;
else q->Rchild=s; //将s插入
//修改a至s路径上结点的平衡因子
if (s->data<a->data)
{ p=a->Lchild; b=p; d=1; }
else { p=a->Rchild; b=p; d=-1; }
while (p!=s)
if (s->data<p->data)
{ p->bf=1; p=p->Lchild; }
else { p->bf=-1; p=p->Rchild; }
if (a->bf==0) a->bf=d;
else if (a->bf+d==0) a->bf=0;
else {
}

```

```

else { if (d==1)
        if (b->bf==1) LL 旋转
        else LR旋转
        else if (b->bf==-1) RR旋转
        else RL旋转
    //修改a的双亲
    case
    f=NULL:   T=b;
    f->Lchild=a: f->Lchild=b;
    f->Rchild=a: f->Rchild=b;
    end of case
}
}

```

### 3.9 动态查找结构B-树

- ◆ 当查找表的大小超过内存容量时，由于必须从磁盘等辅助存储设备上去读取这些查找树结构中的结点，每次只能根据需要读取一个结点，因此，AVL树性能就不是很高。
- ◆ 在AVL树在结点高度上采用相对平衡的策略，使其平均性能接近于BST的最好情况下的性能。
- ◆ 如果保持查找树在高度上的绝对平衡，而允许查找树结点的子树个数（分支个数）在一定范围内变化，能否获得很好的查找性能呢？
- ◆ 基于这样的想法，1970年，R. Bayer设计了许多在高度上保持绝对平衡，而在宽度上保持相对平衡的查找结构
- ◆ 如B-树及其各种变形结构，这些查找结构不再是二叉结构，而是m-路查找树（ $m$ -way search tree），且以其子树保持等高为基本性质，在实际中都有着广泛的应用。

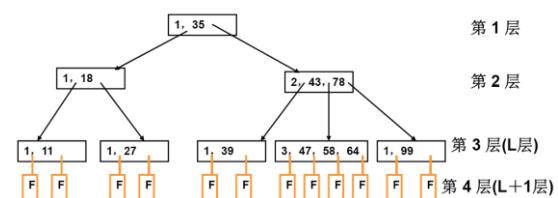
### (1) $m$ -阶查找树

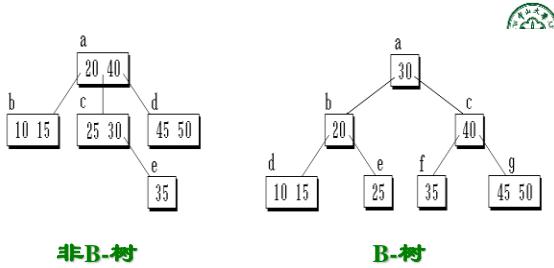
- 一棵  $m$  阶B-树是一棵  $m$  路查找树，它或者是空树，或者是满足下列性质的树：
  - 树中每个结点至多有 $m$ 棵子树；
  - 根结点至少有 2 棵子树；
  - 除根结点以外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树；
  - 所有非终端结点中包含下列信息数据  $(n, A_0, K_1, A_1, K_2, A_2, \dots, K_n, A_n)$ ，其中：  $K_i (i=1, \dots, n)$  为关键字，且  $K_i < K_{i+1}$ ， $A_i (i=0, \dots, n)$  为指向子树根结点的指针， $n$  为关键字的个数
  - 所有的叶子结点（失败结点）都位于同一层。

事实上，每个结点中还应包含指向每个关键字的记录的指针。



### 4 阶B-树





## (2) B-树的查找

- B-树的查找过程是一个顺指针查找结点和在结点的关键字进行查找交叉进行的过程。因此，**B-树的查找时间与B-树的阶数 $m$ 和B-树的高度 $h$ 直接有关，必须加以权衡。**
- 在B-树上进行查找，查找成功所需的时间取决于关键码所在的层次，查找不成功所需的时间取决于树的高度。如果我们定义B-树的高度 $h$ 为失败结点所在的层次，需要了解树的高度 $h$ 与树中的关键码个数 $N$ 之间的关系。



### m值的选择

- 如果提高B-树的阶数 $m$ ，可以减少树的高度，从而减少读入结点的次数，因而可减少读磁盘的次数。
- 事实上， $m$ 受到内存可使用空间的限制。当 $m$ 很大超出内存工作区容量时，结点不能一次读入到内存，增加了读盘次数，也增加了结点内查找的难度。

## (3) B-树的插入

- B-树是从空树起，逐个插入关键码而生成的。
- 在B-树，每个非失败结点的关键码个数都在 $[\lceil m/2 \rceil - 1, m - 1]$ 之间。
- 插入在某个叶结点开始。如果在关键码插入后结点中的关键码个数超出了上界 $m - 1$ ，则结点需要“分裂”，否则可以直接插入。



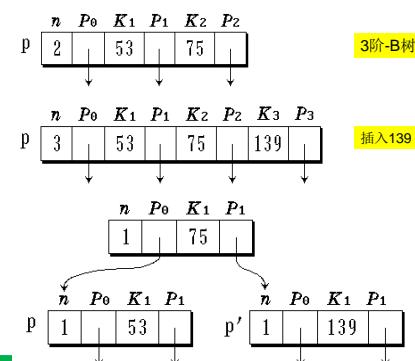
实现结点“分裂”的原则是：设结点A中已经有 $m-1$ 个关键码，当再插入一个关键码后结点中的状态为

$$(m, A_0, K_1, A_1, K_2, A_2, \dots, K_m, A_m)$$

其中  $K_i < K_{i+1}$ ,  $1 \leq i < m$

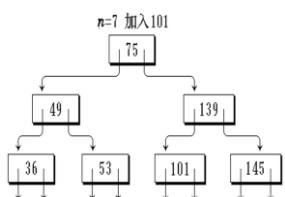
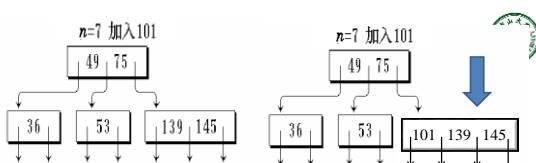
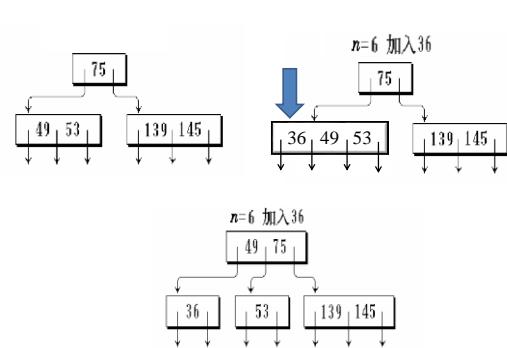
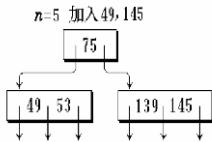
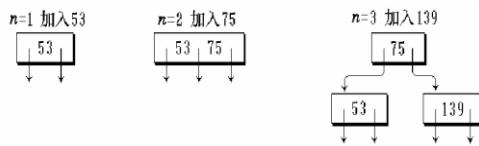
- 这时必须把结点p分裂成两个结点p和q，它们包含的信息分别为：
- 结点 p：  
 $(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \dots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$
- 结点 q：  
 $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \dots, K_m, A_m)$
- 位于中间的关键码 $K_{\lceil m/2 \rceil}$ 与指向新结点q的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$ ，插入到这两个结点的双亲结点中去。

### 结点“分裂”的示例





### 示例：从空树开始逐个加入关键码建立3阶B-树



若设B-树的高度为 $h$ , 那么在自顶向下查找到叶结点的过程中需要进行 $h$ 次读盘。



- 在插入新关键码时，需要自底向上分裂结点，最坏情况下从被插关键码所在叶结点到根的路径上的所有结点都要分裂。

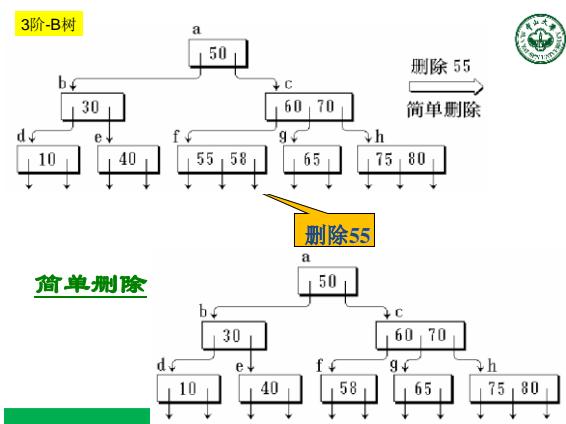
### (4) B-树的删除



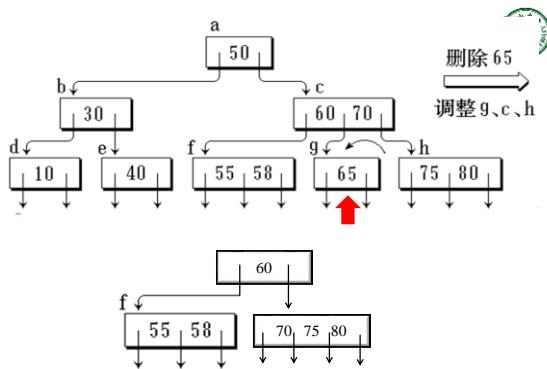
- 在B-树上删除一个关键码时，首先需要找到这个关键码所在的结点，从中删去这个关键码。若该结点不是叶结点，且被删关键码为 $K_i$ ,  $1 \leq i \leq n$ ，则在删去该关键码之后，应以该结点 $A_i$ 所指示子树中的最小关键码 $x$ 来代替被删关键码 $K_i$ 所在的位置，然后在 $x$ 所在的叶结点中删除 $x$ 。

#### • 在叶结点上的删除有4种情况。

- 被删关键码所在叶结点同时又是根结点且删除前该结点中关键码个数 $n \geq 2$ ，则直接删去该关键码。

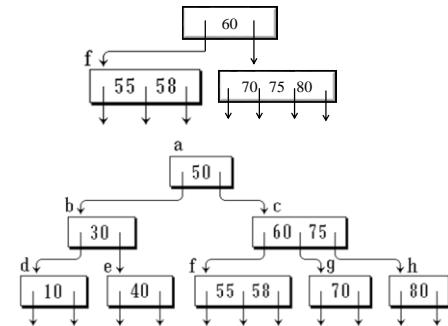


- ① 被删关键码所在叶结点不是根结点且删除前该结点中关键码个数  $n \geq \lceil m/2 \rceil$ , 则直接删去该关键码并将修改后的结点写回磁盘, 删除结束。
- ② 被删关键码所在叶结点删除前关键码个数  $n = \lceil m/2 \rceil - 1$ , 若这时与该结点相邻的右兄弟(或左兄弟)结点的关键码个数  $n \geq \lceil m/2 \rceil$ , 则可按以下步骤调整该结点、右兄弟(或左兄弟)结点以及其双亲结点, 以达到新的平衡。
- 将双亲结点中刚刚大于(或小于)该被删关键码的关键码  $K_i$  ( $1 \leq i \leq n$ ) 下移;
  - 将右兄弟(或左兄弟)结点中的最小(或最大)关键码上移到双亲结点的  $K_i$  位置;



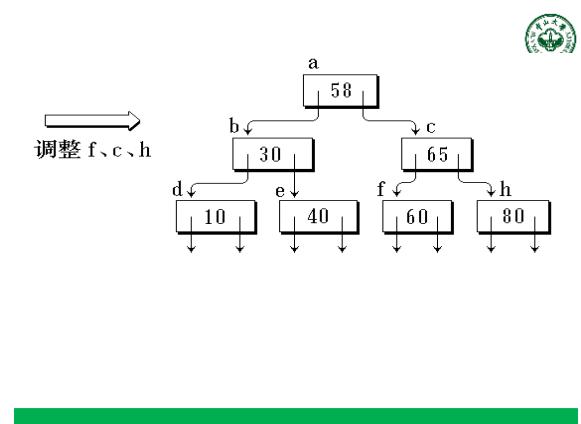
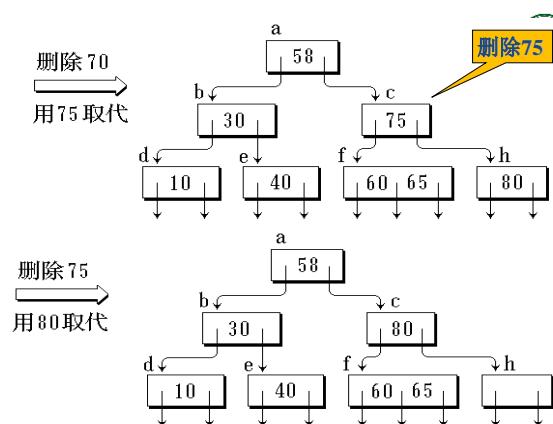
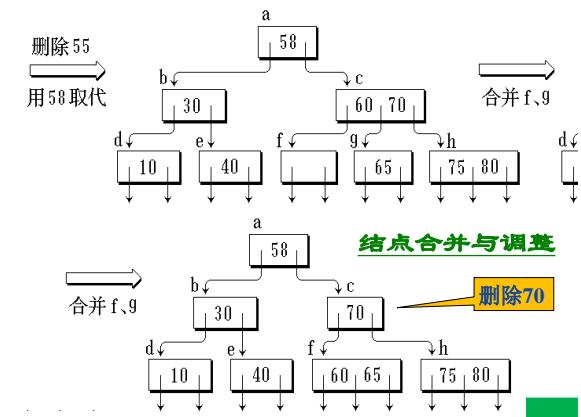
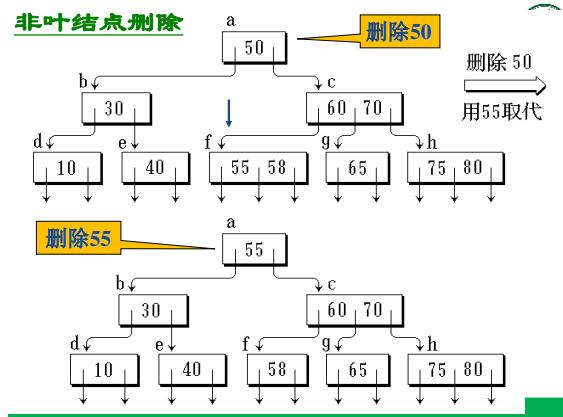
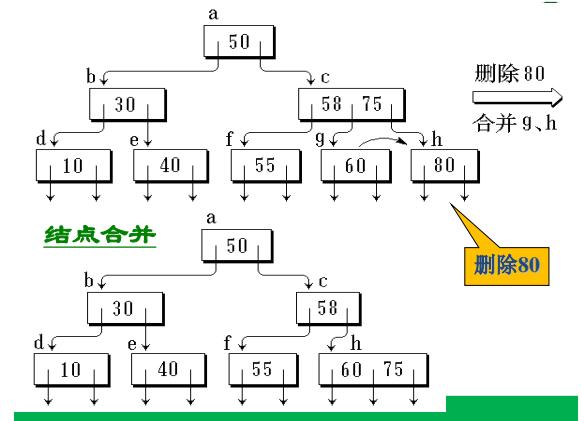
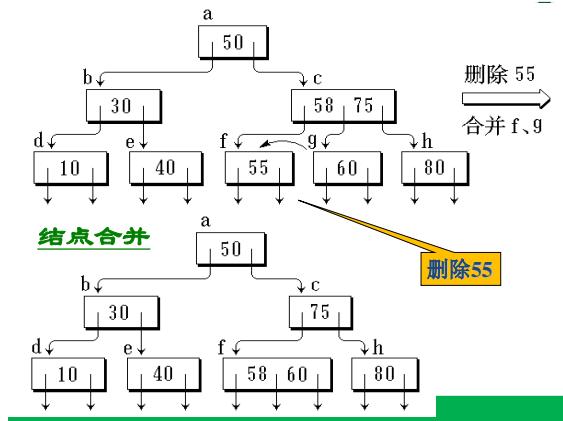
- 将右兄弟(或左兄弟)结点中的最左(或最右)子树指针平移到被删关键码所在结点中最后(或最前)子树指针位置;
  - 在右兄弟(或左兄弟)结点中, 将被移走的关键码和指针位置用剩余的关键码和指针填补、调整。再将结点中的关键码个数减1。
- ③ 被删关键码所在叶结点删除前关键码个数  $n = \lceil m/2 \rceil - 1$ , 若这时与该结点相邻的右兄弟(或左兄弟)结点的关键码个数  $n = \lceil m/2 \rceil - 1$ , 则必须按以下步骤合并这两个结点。

### 结点联合调整

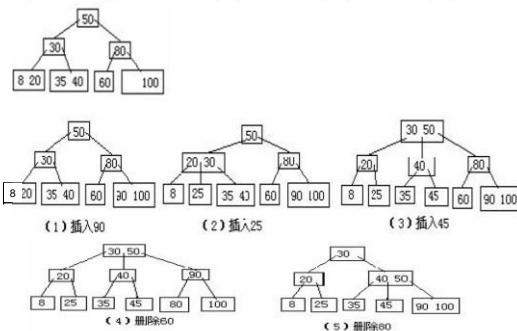


- 将双亲结点  $p$  中相应关键码下移到选定保留的结点中。若要合并  $p$  中的子树指针  $A_i$  与  $A_{i+1}$  所指的结点, 且保留  $A_i$  所指结点, 则把  $p$  中的关键码  $K_{i+1}$  下移到  $A_i$  所指的结点中。
- 把  $p$  中子树指针  $A_{i+1}$  所指结点中的全部指针和关键码都照搬到  $A_i$  所指结点的后面。删去  $A_{i+1}$  所指的结点。
- 在结点  $p$  中用后面剩余的关键码和指针填补关键码  $K_{i+1}$  和指针  $A_{i+1}$ 。
- 修改结点  $p$  和选定保留结点的关键码个数。
- 在合并结点的过程中, 双亲结点中的关键码个数减少了。

- 若双亲结点是根结点且结点关键码个数减到 0, 则该双亲结点应从树上删去, 合并后保留的结点成为新的根结点; 否则将双亲结点与合并后保留的结点都写回磁盘, 删除处理结束。
- 若双亲结点不是根结点, 且关键码个数减到  $\lceil m/2 \rceil - 2$ , 又要与它自己的兄弟结点合并, 重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。



例.对下面的3阶B-树，依次执行下列操作，画出各步操作的结果。  
 1) 插入90 (2) 插入25 (3) 插入45 (4) 删除60 (5) 删除80



## 四、B+树



B树只有利于单个关键字查找，而在数据库等许多应用中常常也需要范围查询。

为了满足这种需求，需要改进B树。

B+树就是这种改进的结果，也是现实中最常用的一种B树变种。

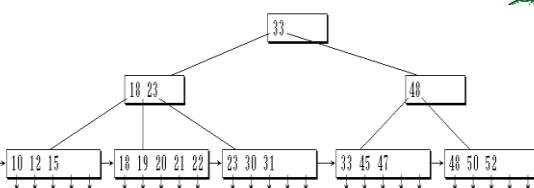
B+树与B树的最大区别是：B+树的元素只存放在叶结点中。

不是叶的结点又称为中间结点。中间结点也存放关键字，但这些关键字只起引导查找的“路标”作用。

- 一棵m阶B+树可以定义如下：

- 树中每个非叶结点最多有  $m$  棵子树；
- 根结点（非叶结点）至少有 2 棵子树。除根结点外，其它的非叶结点至少有  $\lceil m/2 \rceil$  棵子树；有  $n$  棵子树的非叶结点有  $n-1$  个关键码。
- 所有的叶结点都处于同一层次上，包含了全部关键码及指向相应数据对象存放地址的指针，且叶结点本身按关键码从小到大顺序链接；

- 每个叶结点中的子树棵数  $n$  可以多于  $m$ ，可以少于  $m$ ，视关键码字节数及对象地址指针字节数而定。若设结点可容纳最大关键码数为  $m_1$ ，则指向对象的地址指针也有  $m_1$  个。
- 结点中的子树棵数  $n$  应满足  $n \in [\lceil m/2 \rceil, m_1]$ 。
- 若根结点同时又是叶结点，则结点格式同叶结点。
- 所有的非叶结点可以看成是索引部分，结点中关键码  $K_i$  与指向子树的指针  $A_i$  构成对子树（即下一层索引块）的索引项  $(K_i, A_i)$ ， $K_i$  是子树中最小的关键码。特别地，子树指针  $A_0$  所指子树上所有关键码均小于  $K_1$ 。结点格式同B-树。



- 叶结点中存放的是对实际数据对象的索引。
- 在B+树中有两个头指针：一个指向B+树的根结点，一个指向关键码最小的叶结点。可对B+树进行两种查找运算：一种是循叶结点链顺序查找，另一种是从根结点开始，进行自顶向下，直至叶结点的随机查找。

B+树的查找与B-树的查找类似，但是也有不同。

由于与记录有关的信息存放在叶结点中，查找时若在上层已找到待查的关键码，并不停止，而是继续沿指针向下一直查到叶结点层的关键码。

B+树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。上面两种方式结合起来，使得B+树非常适合范围检索。





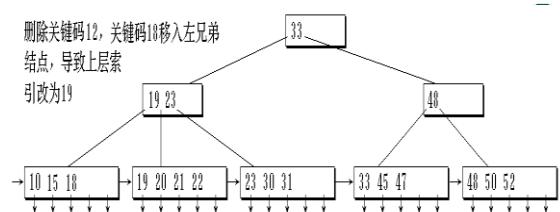
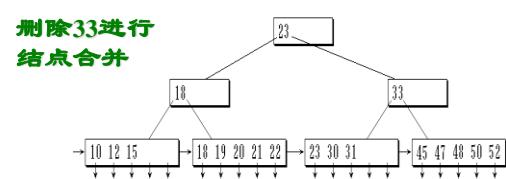
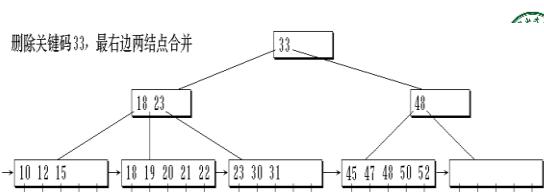
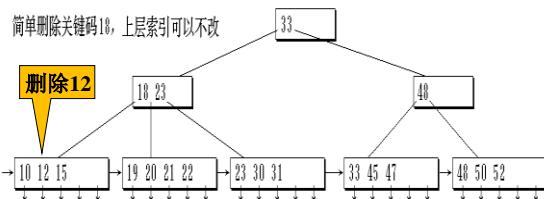
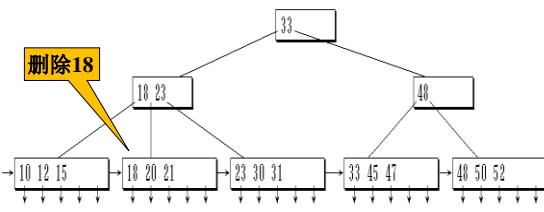
B+树的插入与B-树的插入过程类似。

- B+树的插入仅在叶结点上进行。每插入一个关键码-指针索引项后都要判断结点中的子树棵数是否超出范围。当插入后结点中的子树棵数  $n > m_1$  时，需要将叶结点分裂为两个结点，它们的关键码分别为  $\lceil (m_1+1)/2 \rceil$  和  $\lfloor (m_1+1)/2 \rfloor$ 。并且它们的双亲结点中应同时包含这两个结点的最小关键码和结点地址。此后，问题归于在非叶结点中的插入了。



- B+树的删除仅在叶结点上进行。当在叶结点上删除一个关键码-指针索引项后，结点中的子树棵数仍然不少于  $\lceil m_1/2 \rceil$ ，这属于简单删除，其上层索引可以不改变。
- 如果删除的关键码是该结点的最小关键码，但因在其上层的副本只是起了一个引导查找的“分界关键码”的作用，所以上层的副本仍然可以保留。
- 如果在叶结点中删除一个关键码-指针索引项后，该结点中的子树棵数  $n$  小于结点子树棵数的下限  $\lceil m_1/2 \rceil$ ，必须做结点的调整或合并工作。

### 二、B+树的删除



- 如果右兄弟结点的子树棵数已达到下限  $\lceil m_1/2 \rceil$ ，没有多余的关键码可以移入被删关键码所在的结点，这时必须进行两个结点的合并。将右兄弟结点中的所有关键码-指针索引项移入被删关键码所在结点，再将右兄弟结点删去。