第 3 讲 基于树表的查找——教学讲义

**基于树的查找法**是将待查表组织成特定树的形式并在树结构上实现查找的方法，故又称为**树表式查找法**，主要包括**二叉排序树、平衡二叉树和 B_树**等。

## 一、 二叉排序树

### 1、 二叉排序树定义与描述

二叉排序树又称为二叉查找树，它是一种特殊的二叉树。

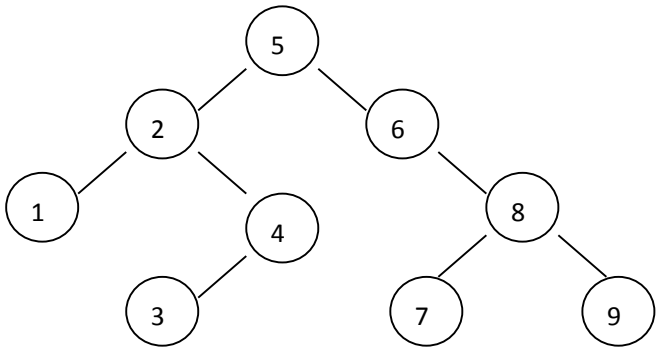
其定义为：二叉树排序树或者是一棵空树，或者是具有如下性质的二叉树：

（1）若它的左子树非空，则左子树上所有结点的值均小于根结点的值；

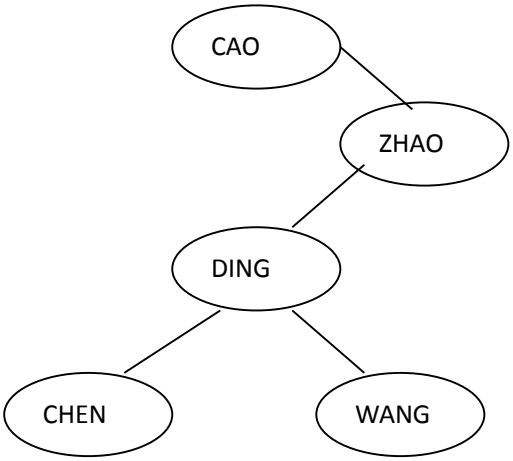
（2）若它的右子树非空，则右子树上所有结点的值均大于（或大于等于）根结点的值；

（3）它的左右子树也分别为二叉排序树。

这是一个递归定义。注意只要结点之间具有可比性即可，如下图 (a)中的数字值间的比较，图 (b)中是用单词字符的 ASCII 码间的比较。




(a)二叉排序树示例 1




(b) 二叉排序树示例 2(根据字符 ASCII 码的大小)

二叉排序树的存储结构同二叉树，使用二叉链表作为存储结构。

其结点结构描述说明如下：

```
typedef struct    node
{ KeyType    key ; /*关键字的值*/
  struct node    *lchild,*rchild;/*左右指针*/
```

}BSTNode, *BSTree;

**2．二叉排序树的创建**

假若给定一个元素序列，我们可以利用逐个插入结点算法创建一棵二叉排序树。因此，实现建立二叉排序树包括创建树与插入结点两个算法。

**[算法思想]：**

首先，将二叉树序树初始化为一棵空树，然后逐个读入元素，每读入一个元素，就建立一个新的结点，并插入到当前已生成的二叉排序树中，即通过多次调用二叉排序树的插入新结点的算法实现，注意插入时比较结点的顺序始终是从二叉排序树的根结点开始。

**①二叉排序树的插入**

已知一个关键字值为 key 的结点 s，若将其插入到二叉排序树中，只要保证插入后仍符合二叉排序树的定义即可。

**[算法思想]：**

1） 若二叉排序树是空树，则 key 成为二叉排序树的根；

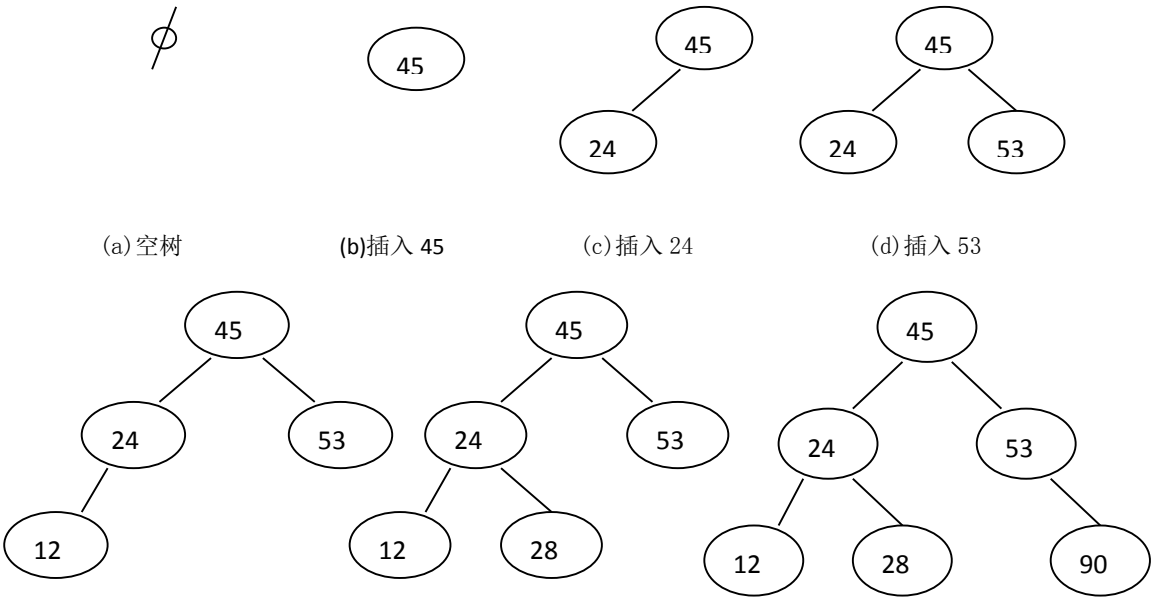
2） 若二叉排序树非空，则将 key 与二叉排序树的根进行比较：

a) 如果 key 的值等于根结点的值，则停止插入；

b) 如果 key 的值小于根结点的值，则将 key 插入左子树；

c) 如果 key 的值大于根结点的值，则将 key 插入右子树。

例如，设关键字的输入顺序为：45，24 ，53，12，28，90，按上述算法生成的二叉排序树的过程如下图所示。




(a)空树 (b)插入 45 (c)插入 24 (d)插入 53

(e)插入 12 (f)插入 28 (g)插入 90

图二叉排序树的建立过程

**[算法描述]：**

```
void InsertBST(BSTree *bst, KeyType key)
/*若在二叉排序树 bst 中不存在关键字等于 key 的元素，插入该元素*/
{ BiTree s;
```

```
  if (*bst==NULL)/*递归结束条件*/
      {
       s=(BSTree)malloc(sizeof(BSTNode));/*申请新的结点 s*/
       s-> key=key;
       s->lchild=NULL;    s->rchild=NULL;
       *bst=s;
      }
  else    if (key < (*bst)->key)
  InsertBST(&((*bst)->lchild), key);/*将 s 插入左子树*/
  else    if (key > (*bst)->key)
  InsertBST(&((*bst)->rchild), key); /*将 s 插入右子树*/
}
```

【二叉排序树插入的递归算法】

可以看出，二叉排序树的插入，即插入每一个结点都是作为一个叶子结点，将其插到二叉排序树的合适位置，插入时不需要移动元素，不涉及树的整体改动。

**[算法分析]：**二叉排序树的插入算法的时间复杂度仍为$O(\log_2 n)$，具体的分析可留作练习。

**②创建二叉排序树**

**[算法描述]：**

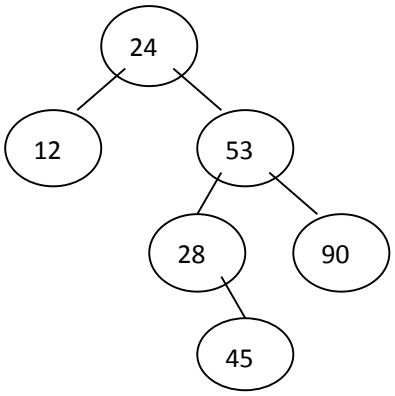
```
void    CreateBST(BSTree    *bst)
/*从键盘输入元素的值，创建相应的二叉排序树*/
{ KeyType key;
  *bst=NULL;
  scanf("%d", &key);
  while (key!=ENDKEY)     /*EDNKEY 为自定义常量*/
     {
       InsertBST(bst, key);          /*在二叉排序树 bst 中插入结点 key*/
       scanf("%d", &key);
     }
}
```

【算法 创建二叉排序树】

**[算法分析]：**

假设共有 n 个元素，要插入 n 个结点需要 n 次插入操作，而插入一个结点的算法时间复杂度为 $O(\log_2 n)$，因此创建二叉排序树的算法时间复杂度为 $O(n\log_2 n)$。

如果输入顺序为 24，53，90，12，28，45，则生成的二叉排序树如下图所示：



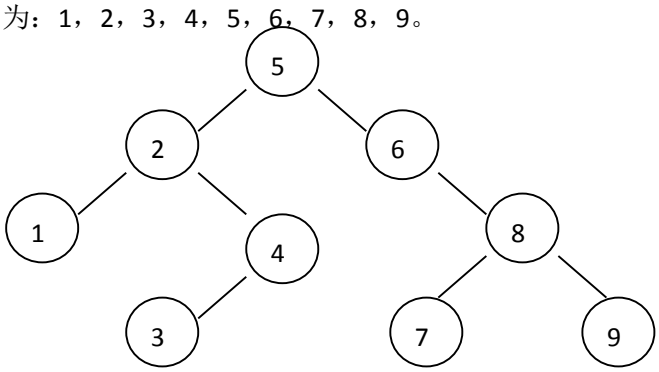

图：输入顺序不同所建立的不同二叉排序树

注意：这个输入序列的值与前一个输入序列具有同样元素值，只是输入顺序不同，所创建的二叉排序树的形态不同。

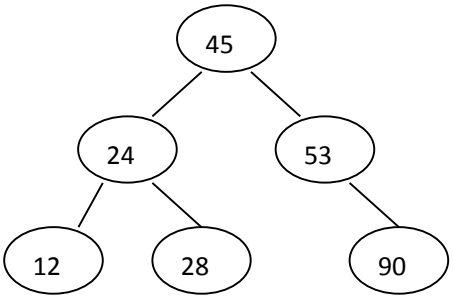
### 3．二叉排序树的查找

**二叉排序树的特性：**根据二叉排序树的定义（左子树小于根结点，右子树大于根结点），根据二叉树中序遍历的定义（先中序遍历左子树，访问根结点，再中序遍历右子树），可以得出二叉排序树的一个重要性质，中序遍历一个二叉排序树，可以得到一个递增有序序列。

图 8.4 所示的二叉树就是一棵二叉排序树，若中序遍历下图的二叉排序树，则可得到一个递增有序序列为：1，2，3，4，5，6，7，8，9。




若中序遍历下图的二叉排序树，均可得到一个递增有序序列为：12，24，28，45，53，90。



**二叉排序树查找的实现方法：**

因为二叉排序树可看作是一个有序表，所以在二叉排序树上进行查找，和折半查找类似，也是一个逐步缩小查找范围的过程。

**[算法思想]：**

首先将待查关键字 k 与根结点关键字 t 进行比较，如果：

1） k= t：则返回根结点地址；

2） k<t：则进一步查左子树；

3） k>t：则进一步查右子树。

**[实现算法]：**

根据二叉排序树的定义，在二叉排序树结构上查找可以用递归与非递归两种实现算法。

1．二叉排序树查找的递归算法

```
BSTree    SearchBST(BSTree bst, KeyType key)
/*在根指针 bst 所指二叉排序树中，递归查找某关键字等于 key 的元素，若查找成功，返回指向该元素结点指针，否则返回空指针*/
{
  if (!bst) return NULL;
  else if (bst->key == key) return bst;/*查找成功*/
  else
    if (bst->key > key)
    return SearchBST(bst->lchild, key);/*在左子树继续查找*/
  else
     return SearchBST(bst->rchild, key);/*在右子树继续查找*/
}
```

**【二叉排序树查找的递归算法】**

2．二叉排序树查找的非递归算法

根据二叉排序树定义，其查找也可以用循环方式直接实现。

二叉排序树的非递归查找过程如下：

```
BSTree    SearchBST(BSTree bst, KeyType key)
/*在根指针 bst 所指二叉排序树 bst 上，查找关键字等于 key 的结点，若查找成功，返回指向该元素结点指针，否则返回空指针*/
{ BSTree q;
  q=bst;
  while(q)
    {if (q->key == key)    return q;    /*查找成功*/
     if (q->key > key)    q=q->lchild;    /*在左子树中查找*/
     else    q=q->rchild;    /*在右子树中查找*/
     }
  return NULL; /*查找失败*/
}/*SearchBST*/
```

**【二叉排序树查找的非递归算法】**

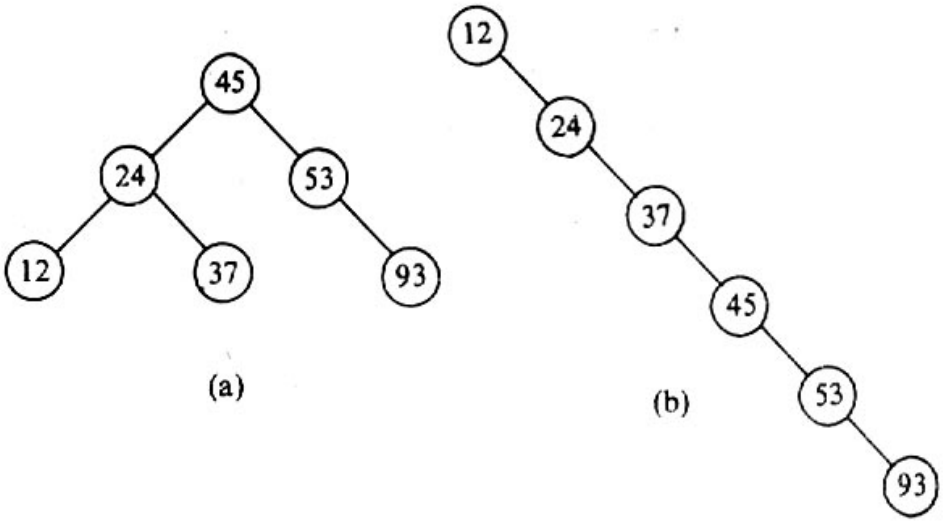
**[算法分析]：**

显然，在二叉排序树上进行查找，若查找成功，则是从根结点出发走了一条从根结点到

待查结点的路径。若查找不成功，则是从根结点出发走了一条从根到某个叶子结点的路径。因此，二叉排序树的查找与折半查找过程类似，在二叉排序树中查找一个记录时，其比较次数不超过树的深度。但是，对长度为 n 的有序表而言，折半查找对应的判定树是唯一的，而含有 n 个结点的二叉排序树却是不唯一的，因为对于同一个关键字集合，关键字插入的先后次序不同，所构成的二叉排序树的形态和深度也不同。而二叉排序树的平均查找长度 ASL 与二叉排序树的形态有关，二叉排序树的各分支越均衡，树的深度浅，其平均查找长度 ASL 越小。例如，下图为两棵二叉排序树，它们对应同一元素集合，但排列顺序不同，分别是：（45，24，53，12，37，93）和（12，24，37，45，53，93）。假设每个元素的查找概率相等，则它们的平均查找长度分别是：

ASL=1/6 （1+2+2+3+3+3）=14/6

ASL=1/6 （1+2+3+4+5+6）=21/6




(a)关键字序列为{45，24，53，12，37，93}的二叉排序树

(b)关键字序列为{12，24，37，45，53，93}的单支树

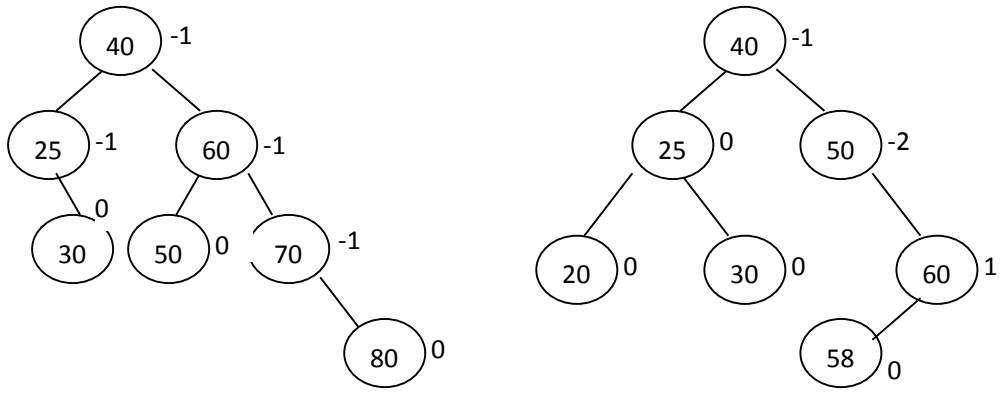
图：二叉查找树的不同形态

由此可见，在二叉排序树上进行查找时的平均查找长度和二叉排序树的形态有关。在最坏情况下，二叉排序树是通过把一个有序表的 n 个结点一次插入生成的，由此得到二叉排序树蜕化为一棵深度为 n 的单支树，它的平均查找长度和单链表上的顺序查找相同，也是（n+1）/2。在最好情况下，二叉排序树在生成过程中，树的形态比较均匀，最终得到的是一棵形态与折半查找的判定树相似的二叉排序树，此时它的平均查找长度大约是 $\log_2 n$。若考虑把 n 个结点，按各种可能的次序插入到二叉排序树中，则有 n！棵二叉排序树（其中有的形态相同），可以证明，对这些二叉排序树的查找长度进行平均，得到的平均查找长度仍然是 $O(\log_2 n)$。就平均性能而言，二叉排序树上的查找和折半查找相差不大，并且二叉排序树上的插入和删除结点十分方便，无需移动大量结点。因此，对于需要经常做插入、删除、查找运算的表，宜采用二叉排序树结构。人们也常常将二叉排序树称为二叉查找树。

**二、平衡二叉排序树**

**平衡二叉排序树**又称为 AVL 树。一棵平衡二叉排序树或者是空树，或者是具有下列性质

的二叉排序树：（1）左子树与右子树的高度之差的绝对值小于等于 1；（2）左子树和右子树也是平衡二叉排序树。引入平衡二叉排序树的目的，是为了提高查找效率，其平均查找长度为$O(\log_2 n)$。在下面的描述中，需要用到结点的**平衡因子**（ **balance factor** ）这一概念，其定义为：结点的左子树深度与右子树深度之差。显然，对一棵平衡二叉排序树而言，其所有结点的平衡因子只能是-1、 0、或 1。当我们在一个平衡二叉排序树上插入一个结点时，有可能导致失衡，即出现绝对值大于 1 的平衡因子，如 2、-2。图 8.9 中给出了一棵平衡二叉排序树和一棵失去平衡的二叉排序树。

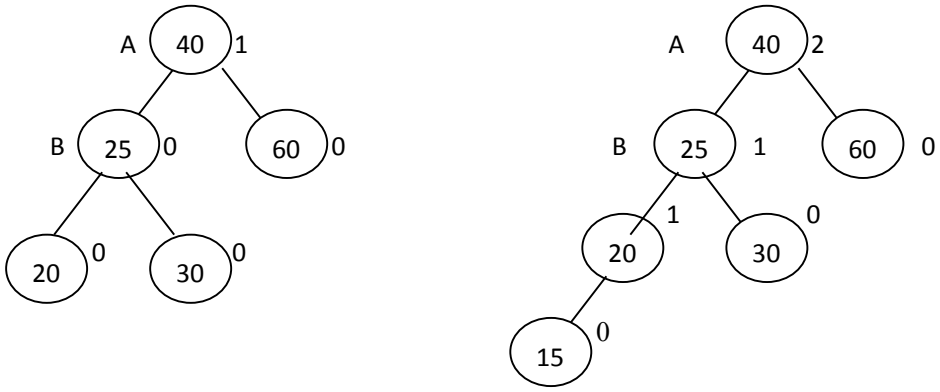



(a) 一棵平衡二叉排序树　　(b) 一棵失去平衡的二叉排序树

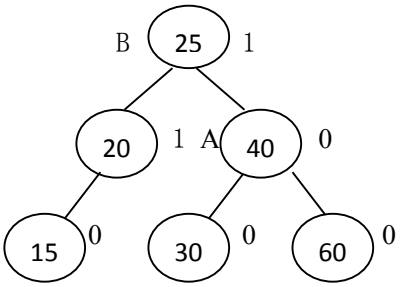
图 8.9 平衡与不平衡的二叉排序树

下面通过几个实例，直观说明失衡情况以及相应的调整方法。

**例 1** 一棵平衡二叉排序树如下图（a）所示。在 A 的左子树的左子树上插入 15 后，导致失衡，如下图（b）所示。为恢复平衡并保持二叉排序树特性，可将 A 改为 B 的右子，B 原来的右子，改为 A 的左子，如下图（c）所示。这相当于以 B 为轴，对 A 做了一次顺时针旋转。



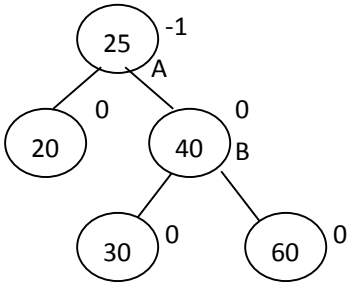

（a）一棵平衡二叉排序树　　（b）插入 15 后失去平衡



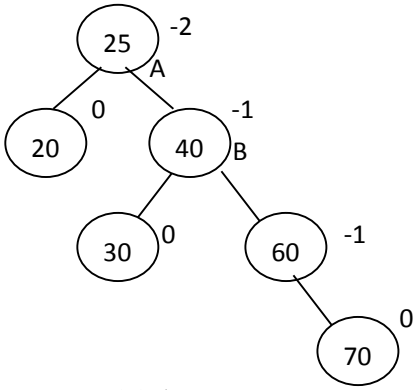

（c）调整后的二叉树

图：不平衡二叉树的调整

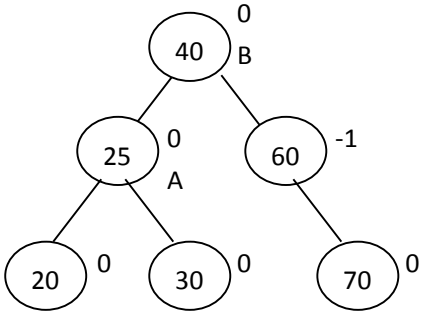
**例 2** 已知一棵平衡二叉排序树如下图（a）所示。在 A 的右子树 B 的右子树上插入 70 后，导致失衡，如下图（b）所示。为恢复平衡并保持二叉排序树特性，可将 A 改为 B 的左子，B 原来的左子，改为 A 的右子，如下图（c）所示。这相当于以 B 为轴，对 A 做了一次逆时针旋转。




（a）一棵平衡二叉排序树




（b）插入 70

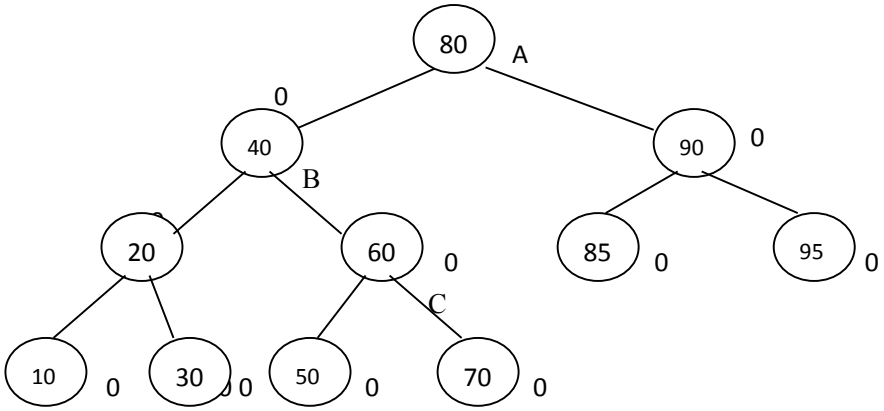



（c）调整后的二叉树

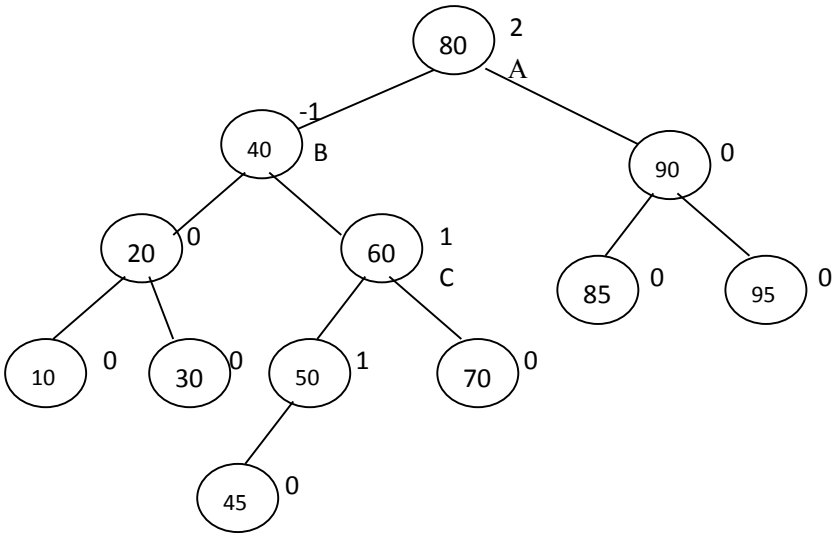
图不平衡二叉树的调整

**例 3** 已知一棵平衡二叉排序树如下图（a）所示。在 A 的左子树 B 的右子树上插入 45 后，导致失衡，如下图（b）所示。为恢复平衡并保持二叉排序树特性，可首先将 B 改为 C 的左子，而 C 原来的左子，改为 B 的右子；然后将 A 改为 C 的右子， C 原来的右子，改为 A 的左子，如下图（c）所示。这相当于对 B 做了一次逆时针旋转，对 A 做了一次顺时针旋转。

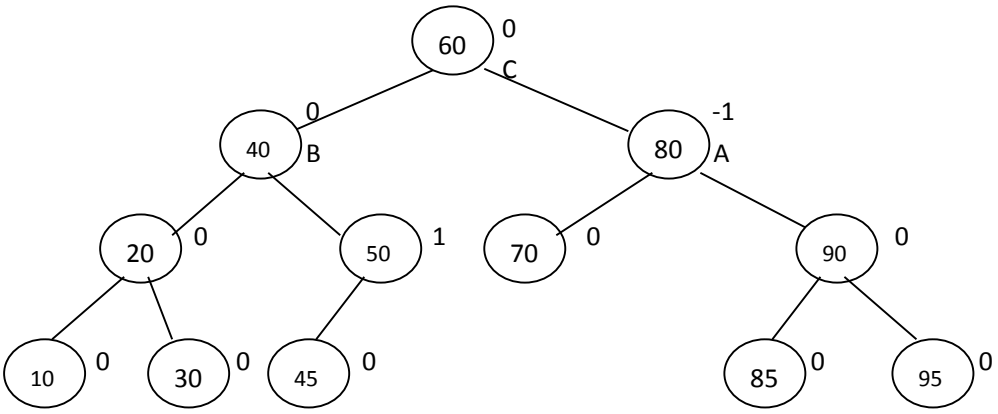
1




（a） 一棵平衡二叉排序树



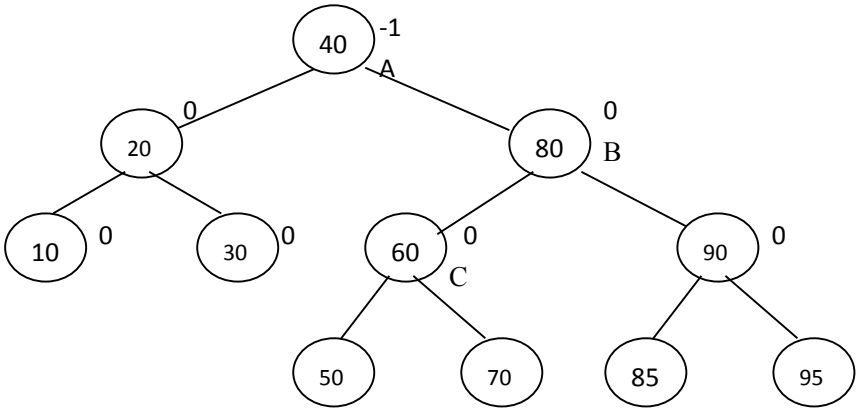

（b）插入 45




（c）调整后的二叉树

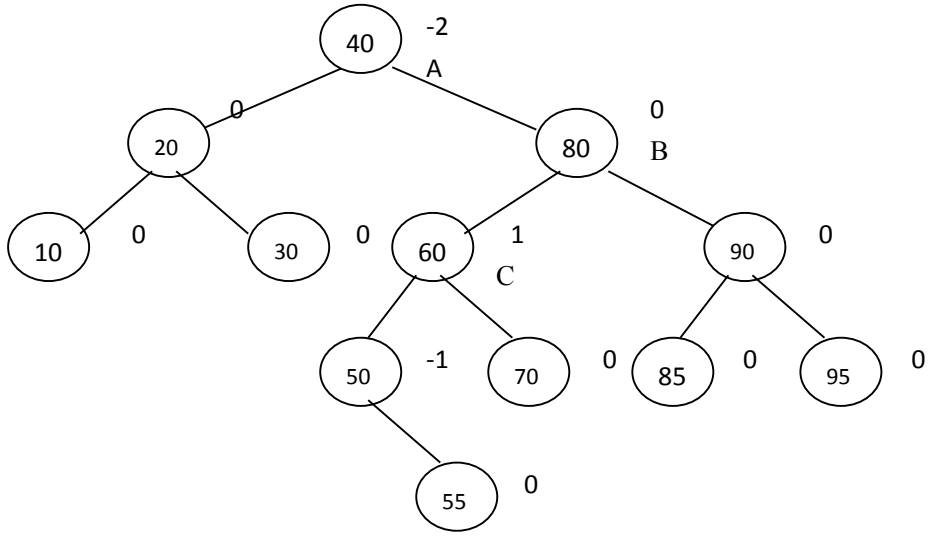
图：不平衡二叉树的调整

**例 4** 已知一棵平衡二叉排序树如下图（a）所示。在 A 的右子树的左子树上插入 55 后，导致失衡，如下图（b）所示。为恢复平衡并保持二叉排序树特性，可首先将 B 改为 C 的右子，而 C 原来的右子，改为 B 的左子；然后将 A 改为 C 的左子， C 原来的左子，改为 A 的右子，如下图（c）所示。这相当于对 B 做了一次顺时针旋转，对 A 做了一次逆时针旋转。

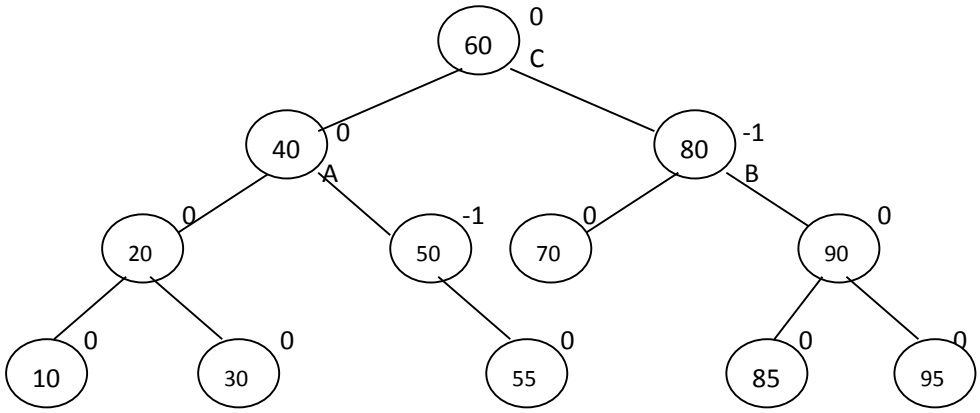


0　　0　　0　　0

（a）　一棵平衡二叉排序树




（b）插入 55




（3）调整后的二叉树

图：不平衡二叉树平衡过程

一般情况下，只有新插入结点的祖先结点的平衡因子受影响，即以这些祖先结点为根的子树有可能失衡。下层的祖先结点恢复平衡，将使上层的祖先结点恢复平衡，因此应该调整最下面的失衡子树。因为平衡因子为 0 的祖先不可能失衡，所以从新插入结点开始向上，遇到的第一个其平衡因子不等于 0 的祖先结点为第一个可能失衡的结点，如果失衡，则应调整以该结点为根的子树。失衡的情况不同，调整的方法也不同。

。

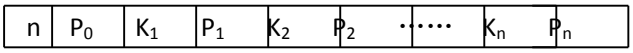
三、B_树

**1． m 路查找树**

与二叉排序树类似，可以定义一种“m 叉排序树”，通常称为 m 路查找树。

一棵 m 路查找树，或者是一棵空树，或者是满足如下性质的树：

1） 结点最多有 m 棵子树，m—1 个关键字，其结构如下：




其中 n 为关键字个数，Pi（0≤i≤n）为指向子树根结点的指针，$K_i$（1≤i≤n）为关键字，

2） $K_i<K_{i+1}$，1≤i≤n-1

3） 子树 $P_i$ 中的所有关键字均大于 $K_i$、小于 $K_{i+1}$，1≤i≤n-1

4） 子树 $P_0$ 中的关键字均小于 $K_1$，而子树 $P_n$ 中的所有关键字均大于 $K_n$

5） 子树 $P_i$ 也是 m 路查找树，0≤i≤n。

从上述定义可以看出，对任一关键字 $K_i$ 而言，$P_{i-1}$ 相当于其“左子树”，$P_i$ 相当于其“右子树”，1≤i≤n 。

下图所示为一棵 3 路查找树，其查找过程与二叉排序树的查找过程类似。如果要查找 35，首先找到根结点 A，因为 35 介于 20 和 40 之间，因而找到结点 C，又因为 35 大于 30，所以找到结点 E，最后在 E 中找到 35。

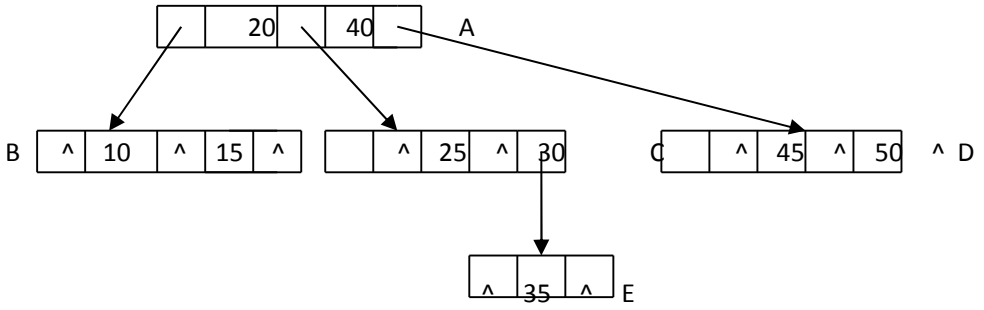



图　3 路查找树

显然，如果 m 路查找树为平衡树时，其查找性能会更好。下面要讨论的 B_树便是一种平衡的 m 路查找树。

## 2. B_树

一棵 B_树是一棵平衡的 m 路查找树，它或者是空树，或者是满足如下性质的树：

（1）树中每个结点最多有 m 棵子树；

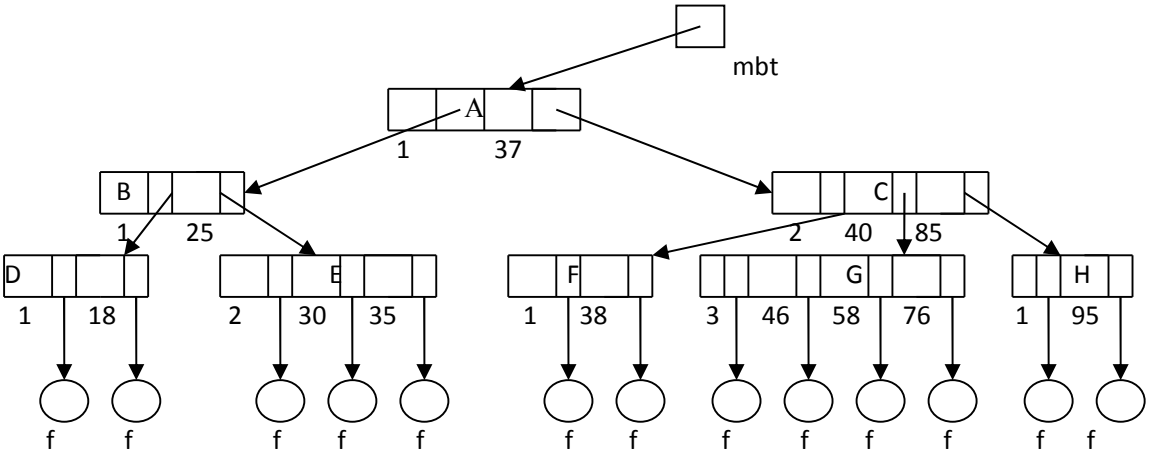
（2）根结点至少有两棵子树；

（3）除根结点之外的所有非叶结点至少有$\lceil m/2 \rceil$棵子树；

（4）所有叶结点出现在同一层上，并且不含信息，通常称为失败结点。失败结点为虚结点，在 B_树中并不存在，指向它们的指针为空指针。引入失败结点是为了便于分析 B_树的查找性能。

下图所示为一棵 4 阶 B_树，其查找过程与 m 路查找树相同。

例如，查找 58 的过程如下：




图一棵 4 阶 B_树

首先由根指针 mbt 找到根结点 A，因为 58>37，所以找到结点 C，又因为 40<58<85，所以找到结点 G，最后在结点 G 中找到 58。如果要查找 32，首先由根指针 mbt 找到根结点 A，因为 32<37，所以找到结点 B，又因为 32>25，所以找到结点 E，因为 30<32<35，所以最后找到失败结点 f ，表示 32 不存在，查找失败。

在具体实现时，采用如下结点结构：

| parent | n | $K_1$ | $K_2$ | … | $K_n$ | $P_0$ | $P_1$ | … | $P_n$ |
|---|---|---|---|---|---|---|---|---|---|

其中 n 、$K_i$、 $P_i$ 的含义以及使用方法与前面 m 路查找树相同，parent 为指向双亲结点的指针。

```
#define    m    <阶数>
typedef   int    Boolean;
typedef   struct Mbtnode
                    {
                       struct Mbtnode    *parent ;
                       int    keynum ;
                       KeyType    key[m+1] ;
                       struct Mbtnode    *ptr[m+1] ;
                    }   Mbtnode, *Mbtree;
```

```
Boolean    srch_mbtree (Mbtree    mbt,    KeyType    k,    Mbtree    *np,    int *pos)
/*在根为 mbt 的 B_树中查找关键字 k，如果查找成功，则将所在结点地址放入 np，将结点内位置序号放入 pos，并返回 true；否则，将 k 应被插入的结点地址放入 np，将结点内应插位置序号放入 pos，并返回 false*/
{
      p = mbt; fp = NULL; found = false; i = 0;
      while (p != NULL && !found)
            {
            i = search (p, k);
                  if (i>0 && p->key[i] == k)    found = true;
                  else    { fp = p;    p = p->ptr[i]; }
            }
```

```
    if (found)    { *np = p; *pos = i ; return true ;}
       else    {*np = fp; *pos = i; return false ;}
}
```

【在 B_树中查找关键字为 k 的元素】

```
    int    search (Mbtree    mbt,    KeyType    key )
    /*在 mbt 指向的结点中，寻找小于等于 key 的最大关键字序号*/
    {
          n = mbt->keynum ;
          i = 1 ;
          while (i <= n && mbt->key[i] <= key )    i ++;
          return (i – 1)       /*  返回小于等于 key 的最大关键字序号 ， 为 0  时表示应到
                                   最左分支找，越界时表示应到最右分支找  */
    }
```

【寻找小于等于关键字 k 的关键字序号】