# 查找表的概念

查找表：由同一类型的数据元素(或记录)构成的集合。对查找表进行的经常操作为：**查找、检索、增加、删除**。

**静态查找表**: 对查找表只进行前两种操作。

**动态查找表**：不仅限于前两种操作。

关键字：数据元素中某个数据项的值，用以标识一个数据元素，如果是唯一标识，则称为主关键字。

查找是否成功：根据给定的值，在查找表中确定一个其关键字等于给定值的元素，如果表中存在这样元素，则称查找成功，否则，不成功。

# 静态查找表

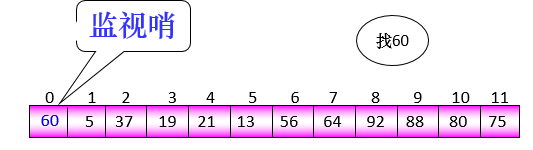
## 顺序查找

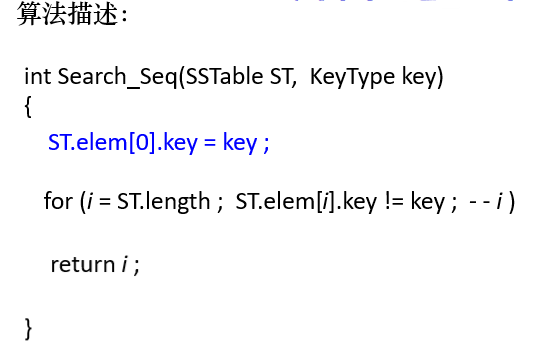
顺序查找：从表的一端开始，逐个进行记录的关键字和给定值的比较。

typedef struct {   
 ElemType \* elem;

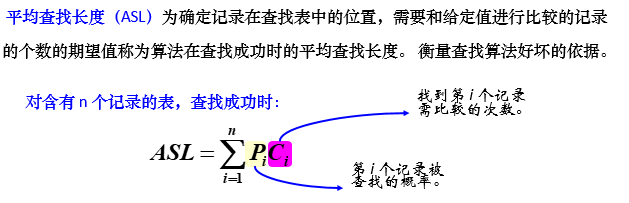
静态查找表的顺序存储结构

int length; // 表长度   
} SSTable;

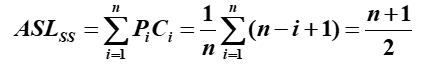




**设置监视哨的目的就是为了能够让每一次查找都能找到，若返回的是0，则代表查找不成功，若查找返回的结果不是0，则代表查找成功**



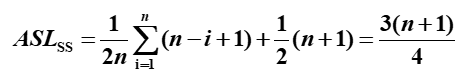
顺序查找的平均查找长度：



查找不成功时，关键字的比较次数总是 *n* + 1 次。

假设：查找成功与不成功的概率和可能性相同，每个记录被查找的概率相同。

平均查找长度（成功与不成功的平均查找长度之和）为



## 折半查找

int Search\_Bin ( SSTable ST, KeyType key )

{ low = 1 ; high = ST.length ; // 置区间初值

　 while (low <= high)

{

mid = (low + high) / 2 ;

　　 if (ST.elem[mid].key = key)

return mid ; // 找到待查元素

　　 else if (key < ST.elem[mid].key)

　　　 high = mid - 1; // 继续在前半区间进行查找

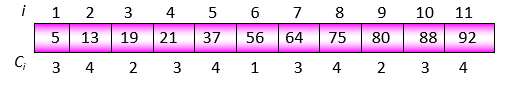
　　 else

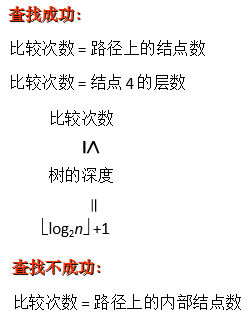
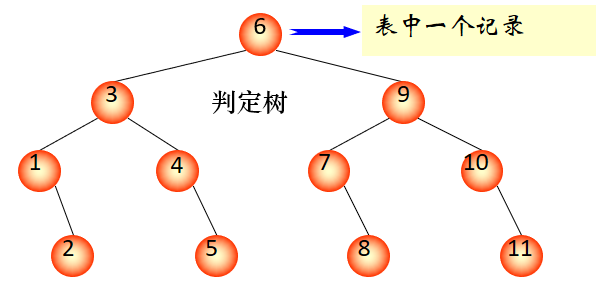
low = mid + 1; // 继续在后半区间进行查找

　 }

　return 0 ; // 顺序表中不存在待查元素

} // Search\_Bin





平均查找长度ASL（成功时）：



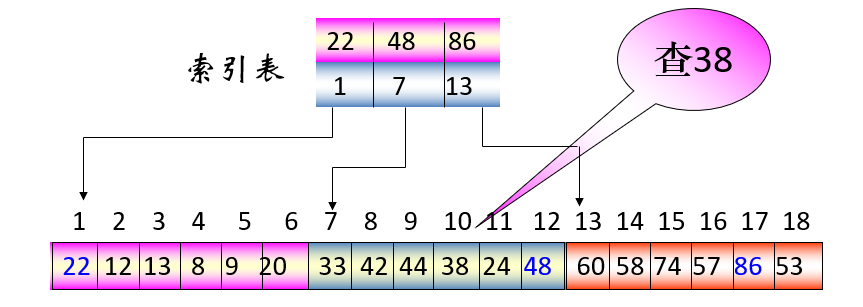
折半查找优点：效率比顺序查找高。

折半查找缺点：只适用于**有序表**，且限于**顺序存储结构。**

## 索引查找（分块查找）

条件：1、将表分成几块，且表或者有序，或者分块有序；（**若 *i* < *j*，则第 *j* 块中所有记录的关键字均大于第 *i* 块中的最大关键字**）

2、建立“索引表”（每个结点含有最大关键字域和指向本块第一个结点的指针，且按关键字有序）。



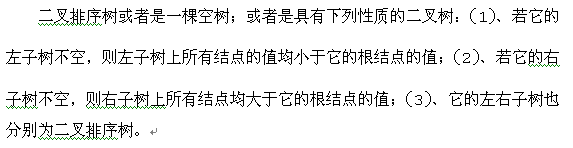
平均查找长度：**比顺序查好，比折半查差**

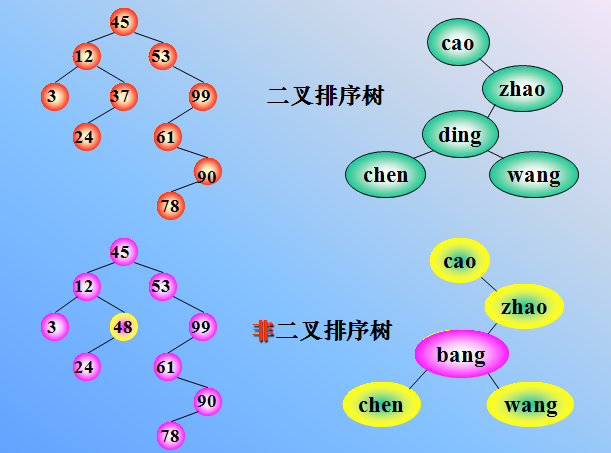
查找方法比较(静态查找表)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **顺序查找** | **折半查找** | **分块查找** |
| **ASL** | **最大** | **最小** | **中间** |
| **表结构** | **有序表、无序表** | **有序表** | **分块有序** |
| **存储结构** | **顺序表、线性链表** | **顺序表** | **顺序表、线性链表** |

# 动态查找表

## 二叉排序树





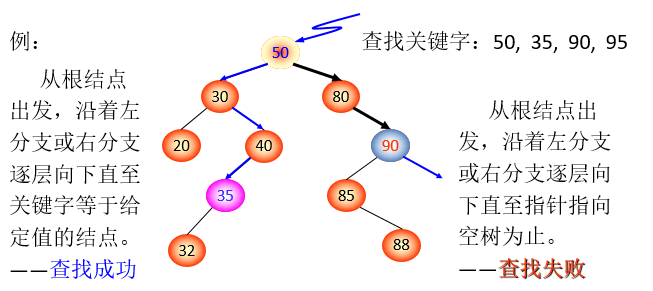
## 二叉排序树的查找过程

若二叉排序树为空，则查找不成功；否则

1) 若给定值等于根结点的关键字，则查找成功；

2) 若给定值小于根结点的关键字，则继续在左子树上进行查找；

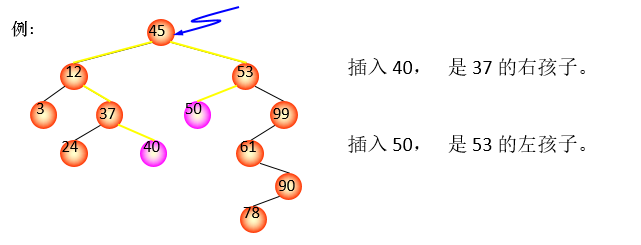
3) 若给定值大于根结点的关键字，则继续在右子树上进行查找。



## 二叉排序树的插入

1)、若二叉排序树为空树，则新插入的结点为**根结点**；

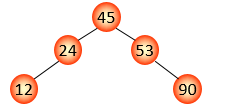
2)、若二叉排序树非空，则新插入的结点必为一个新的叶子结点，并且是**查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。**



## 二叉排序树生成

从空树出发，经过一系列的查找、插入操作 之后，可生成一棵二叉排序树。

例：设查找的关键字序列为 {45, 24, 53, 45, 12, 24, 90}，可生成二叉排序树如下：



**中序遍历二叉排序树可得到一个关键字的有序序列。**

## 二叉排序树的删除

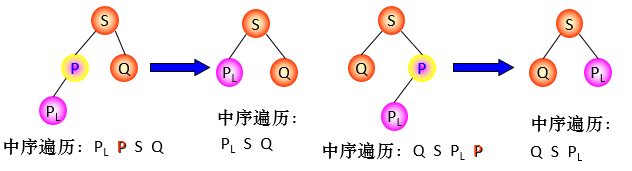
要求：在删除某个结点之后，仍然保持二叉排序树的特性。

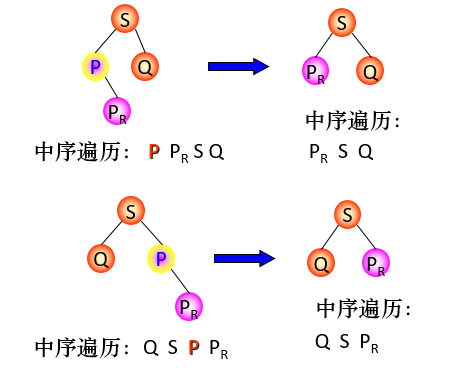
删除二叉排序树中的 \*p 结点，分三种情况讨论：

1)、\*p 为叶子结点。因删除叶子结点不破坏树的结构，故只

需修改 \*p 双亲 \*f 的指针：f -> lchild=NULL 或 f->rchild=NULL

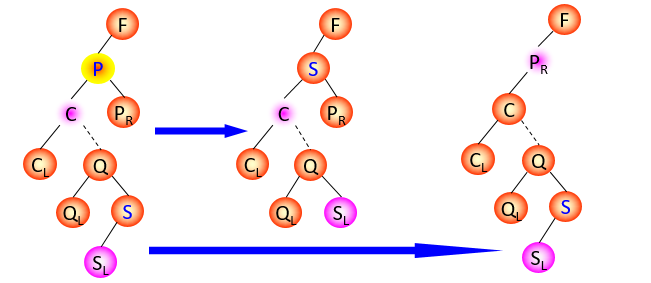
2)、\*p 只有左子树或右子树：\*p 只有左子树，用 \*p 的左孩子代替 \*p；\*p 只有右子树，用 \*p 的右孩子代替 \*p；



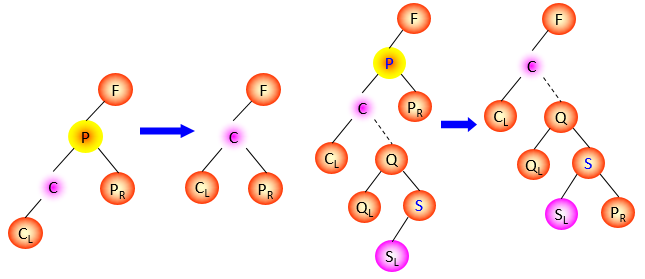


3)、**\*p 左、右子树均非空：**

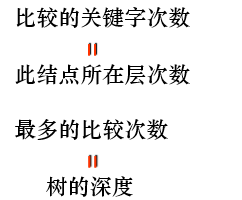
用 \*p 的直接前驱取代 \*p。 用 \*p 的直接后继取代 \*p。



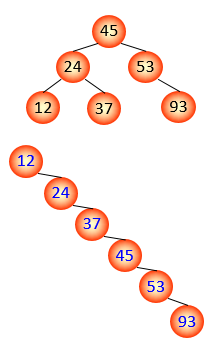
用 \*p 的左子树取代 \*p。



二叉排序树的查找分析



**含有 *n* 个结点的二叉排序树的平均查找长度和树的形态有关**



## 平衡二叉树

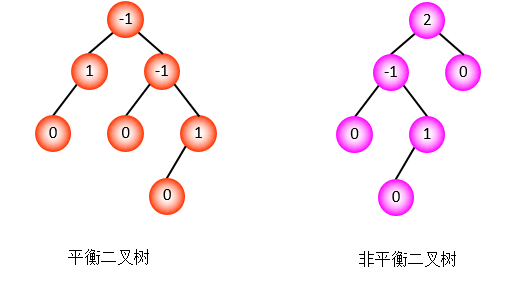
平衡二叉树又称 AVL 树，它是具有如下性质的二叉树：

* 左、右子树是平衡二叉树；
* **所有结点的左、右子树深度之差的绝对值≤ 1。**

为了方便起见，给每个结点附加一个数字 = **该结点左子树与右子树的深度差。**这个数字称为结点的**平衡因子**。这样，可以得到 AVL 树的其它性质（可以证明）：

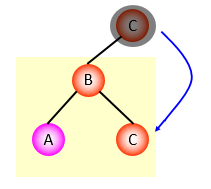
任一结点的平衡因子只能取：**-1、0 或 1**；如果树中任意一个结点的平衡因子的绝对值大于 1，则这棵二叉树就**失去平衡**。

判断下列二叉树是否 AVL 树？

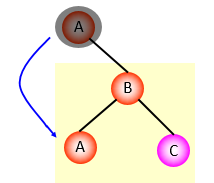


如果在一棵 AVL 树中插入一个新结点后造成失衡，则必须重新调整树的结构，使之恢复平衡。 我们称此调整平衡的过程为**平衡旋转**。

1) LL 平衡旋转：



2) RR 平衡旋转：

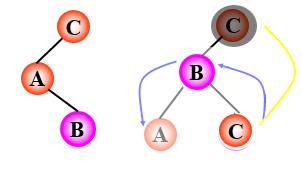


**以B为轴，对A做了一次单向右旋平衡旋转。**

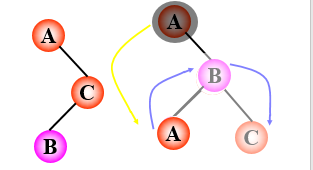
**以B为轴， 对A做了一次单向左旋平衡旋转。**



**3) LR 平衡旋转：**



**4) RL 平衡旋转：**



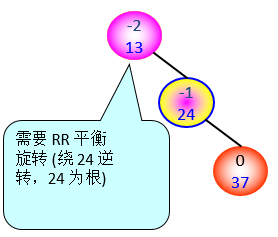
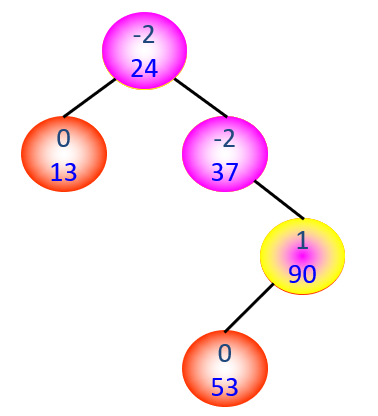
**对B做了一次逆时针旋转， 对A做了一次顺时针旋转。（先左后右）**

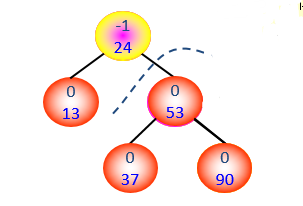


**对B做了一次顺时针旋转， 对A做了一次逆时针旋转。（先右后左）**



例：请将下面序列构成一棵平衡二叉排序树： (13, 24, 37, 90, 53)



**B-树的特点：平衡、多路、查找**