

# 第9章 查找



# 第9章 查找

---

## 9.0 基本概念

## 9.1 静态查找表

## 9.2 动态查找表

## 9.3 哈希表

# 基本概念

- **查找：**就是在数据集合中寻找满足某种条件的数据对象。
- **查找表：**是由同一类型的数据元素(或记录)组成的数据集合。
- 查找的结果通常有两种可能：
  - ◆ **查找成功**，即找到满足条件的数据对象。
  - ◆ **查找不成功**，或查找失败。作为结果，报告一些信息，如失败标志、失败位置等。

# 基本概念

- **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。
- **主关键字**：可唯一地标识一个数据元素的关键字。
- **次关键字**：用以识别若干记录的关键字。

使用基于主关键字的查找，查找结果应是唯一的。

- **静态查找表**（p214）——只查找，不改变集合内的数据元素。
- **动态查找表**（p214）——既查找，又改变（增减）集合内的数据元素。

# 基本概念

名称	代码	涨跌幅	最新价	涨跌额	买入/卖出价	成交量(手)
中国石油	sh601857	-0.47%	12.68	-0.06	12.68/12.69	391306
工商银行	sh601398	-2.31%	4.66	-0.11	4.65/4.66	442737
中国银行	sh601988	-1.43%	3.45	-0.05	3.45/3.46	194203
招商银行	sh600036	-1.63%	14.52	-0.24	14.52/14.54	385271
交通银行	sh601328	-1.29%	6.10	-0.08	6.09/6.10	347937
中信证券	sh600030	-2.69%	15.22	-0.42	15.22/15.23	597025
中国石化	sh600028	-1.16%	9.38	-0.11	9.37/9.38	538895
中国人寿	sh601626	-0.16%	25.63	-0.04	25.61/25.63	66666
中国平安	sh601318	+1.28%	63.29	+0.80	63.29/63.30	153700
宝钢股份	sh600019	-1.77%	7.21	-0.13	7.21/7.22	211077
中国远洋	sh601919	-2.35%	11.24	-0.27	11.22/11.24	156162
万科A	sz000002	-1.85%	9.01	-0.17	9.00/9.01	542249

①数据元素(记录)

②数据项(字段)

③主关键字

④主关键码

⑤次关键字

# 基本概念

---

**讨论1：**查找的过程是怎样的？

给定一个值 $K$ ，在含有 $n$ 个记录的文件中进行搜索，寻找一个关键字值等于 $K$ 的记录，如找到则输出该记录，否则输出查找不成功的信息。

# 基本概念

讨论2：对查找表常用的操作有哪些？

- ❖ 查询某个“特定的”数据元素是否在表中；
- ❖ 查询某个“特定的”数据元素的各种属性；
- ❖ 在查找表中插入一元素；
- ❖ 从查找表中删除一元素。

“特定的” =  
关键字

# 基本概念

---

讨论3：有哪些查找方法？

查找方法取决于表中数据的排列方式；

针对静态查找表和动态查找表的查找方法也有所不同。



例如查字典



# 基本概念

---

## 讨论4：如何评估查找方法的优劣？

明确：查找的过程就是将给定的K值与文件中各记录的关键字项进行比较的过程。

在查找过程中关键字的平均比较次数或平均读写磁盘次数(只适合于外部查找)称为平均查找长度ASL。



Average Search Length

# 基本概念

## 讨论4：如何评估查找方法的优劣？

$$ASL = \sum_{i=1}^n P_i \cdot C_i$$

统计意义上的数学期望值

其中：

$n$ 是文件记录个数；

$P_i$ 是查找第 $i$ 个记录的查找概率（通常取等概率,即 $P_i = 1/n$ ）；

$C_i$ 是找到第 $i$ 个记录时所经历的比较次数。

# 基本概念

---

## 讨论4：如何评估查找方法的优劣？

物理意义：假设每一元素被查找的概率相同，  
则查找每一元素所需的比较次数之总和再取平均，即为ASL。

显然，ASL值越小，时间效率越高。

# 9.1 静态查找表

---

在静态查找表中，数据对象存放于数组中，利用数组元素的下标作为数据对象的存放地址。查找算法根据给定值 $x$ ，在数组中进行查找。直到找到 $x$ 在数组中的存放位置或可确定在数组中找不到 $x$ 为止。

# 9.1 静态查找表

---

针对静态查找表的查找算法主要有：

- 一、顺序查找（线性查找）
- 二、折半查找（二分或对分查找）（重点与考点）
- 三、分块查找（索引顺序查找）（第三种物理结构）

# 9.1 静态查找表

---

静态查找表的定义（抽象数据类型）

ADT StaticSearchTable {

数据对象D:

数据关系R:

基本操作P:

} ADT StaticSearchTable

参见教材P216

## 9.1.1 顺序表查找（又称线性查找）

顺序表查找：用逐一比较的办法顺序查找关键字，这显然是最直接的办法。

见下页之例或教材P216

❖ 对顺序结构如何线性查找？

❖ 对单链表结构如何线性查找？

从头指针始 “顺藤摸瓜”

❖ 对非线性树结构如何顺序查找？

借助各种遍历操作！

## 9.1.1 顺序表查找（又称线性查找）

### 1 顺序表的存储结构(动态数组)

```
typedef struct {  
    ElemType *elem;  
    //表基址，0号单元留空。表容量为全部元素  
    int length; //表长，即表中数据元素个数  
}SSTable;
```



## 9.1.1 顺序表查找（又称线性查找）

### 2 算法的实现

```
int Search_Seq( SSTable ST , KeyType key ){  
    int i;  
    for( i=1; i<= ST.length; i++ ){  
        if(ST.elem[i]==key)  
            return i;  
    }  
    return 0;  
} // Search_Seq
```

# 9.1.1 顺序表查找（又称线性查找）

## 2 算法的实现

技巧：把待查关键字key存入表头或表尾（俗称“哨兵”），这样可以加快执行速度。

例：若将待查找的特定值key存入顺序表的首部（如0号单元），并从后向前逐个比较,就不必担心“查不到”。

# 9.1.1 顺序表查找（又称线性查找）

## 2 算法的实现

```
int Search_Seq( SSTable ST, KeyType key ){
```

```
    ST.elem[0].key = key;
```

```
    for( i=1; i<=n; i++ )  
        //在顺序表ST中，查找关键字与key相同的元素；若成功，  
        //返回其位置信息，否则返回0
```

```
        return i; //设立哨兵，可免去查找过程中每一步都要检测是否查
```

```
    } // Search_Seq 找完毕。
```

// 不要用 `for (i=1; i<=n; i++)`

或 `for (i=n; i>0; --i)`

// 找不到=不

找到=成功，返回值1正好代表所找元素的位置。

# 9.1.1 顺序表查找（又称线性查找）

## 4 算法分析讨论

讨论1：查找失败怎么办？

——返回特殊标志，例如返回空记录或空指针。前例中设立了“哨兵”，就是将关键字送入末地址 `ST.elem[0].key` 使之结束，并返回 `i=0`。

讨论2：查找效率怎样计算？

——用平均查找长度ASL衡量。

## 9.1.1 顺序表查找（又称线性查找）

### 4 算法分析讨论

讨论3：如何计算ASL？

分析：

查找第1个元素所需的比较次数为1；

查找第2个元素所需的比较次数为2

.....

查找第n个元素所需的比较次数为n；

未考虑查找不成功的情况：查找哨兵所需的比较次数为n+1

总计全部比较次数为： $1 + 2 + \cdots + n = (1+n)n/2$

若求某一个元素的平均查找次数，还应当除以n（等概率），

即： $ASL = (1+n) / 2$ ，时间效率为  $O(n)$

这是查找成功的情况

## 9.1.1 顺序表查找（又称线性查找）

### 4 算法分析讨论

设查找第  $i$  个元素的概率为  $p_i$ ，查找到第  $i$  个元素所需比较次数为  $c_i$ ，则查找成功的平均查找长度：

$$ASL_{succ} = \sum_{i=1}^n p_i \cdot c_i \quad \left( \sum_{i=1}^n p_i = 1 \right)$$

在顺序查找情形， $c_i = n - i + 1$ ， $i = 1, \dots, n$ ，因此：

$$ASL_{succ} = \sum_{i=1}^n p_i \cdot (n - i + 1)$$

在等概率情形， $p_i = 1/n$ ， $i = 0, 1, \dots, n-1$ 。

$$ASL_{succ} = \sum_{i=1}^n \frac{1}{n} (n - i + 1) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

## 9.1.1 顺序表查找（又称线性查找）

---

### 5 顺序查找的特点：

优点：算法简单，且对顺序结构或链表结构均适用。

缺点：ASL 太大，时间效率太低。

## 9.1.2 折半查找（二分查找）

这是一种容易想到的查找方法。

先给数据排序（例如按升序排好），形成有序表，然后再将key与正中元素相比，若key小，则缩小至左半部内查找；再取其中值比较，每次缩小1/2的范围，直到查找成功或失败为止。



## 9.1.2 折半查找（二分查找）

### 1、存储结构

- ❖ 对顺序表结构如何编程实现折半查找算法？  
——见下页之例，或见教材（P219）
- ❖ 对单链表结构如何折半查找？  
——无法实现！因全部元素的定位只能从头指针head开始
- ❖ 对非线性(树)结构如何折半查找？  
——可借助二叉排序树来查找（属动态查找表形式）。

## 9.1.2 折半查找（二分查找）

**折半查找举例：**已知如下11个元素的有序表：

(05 13 19 21 37 56 64 75 80 88 92)，请查找关键字为21和85的数据元素。

Low指向待查元素所在区间的下界

mid指向待查元素所在区间的中间位置

high指向待查元素所在区间的上界

解：

① 先设定3个辅助标志: low, high, mid,  
显然有:  $\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$

## 9.1.2 折半查找（二分查找）

### ② 运算步骤：

- (1)  $low = 1, high = 11$  ,故 $mid = 6$  , 待查范围是  $[1, 11]$ ;
- (2) 若  $ST.elem[mid].key < key$  , 说明  $key \in [mid+1, high]$  ,  
则令:  $low = mid+1$ ; 重算  $mid = \lfloor (low+high)/2 \rfloor$  ; .
- (3) 若  $ST.elem[mid].key > key$  , 说明  $key \in [low, mid-1]$  ,  
则令:  $high = mid-1$ ; 重算  $mid$  ;
- (4) 若  $ST.elem[mid].key = key$  , 说明查找成功, 元素序号= $mid$ ;

## 9.1.2 折半查找（二分查找）

---

② 运算步骤：

结束条件：

(1) 查找成功： $ST.elem[mid].key = key$

(2) 查找不成功： $high < low$ （意即区间长度小于0）

## 9.1.2 折半查找（二分查找）

```
int Search_Bin(SSTable ST, KeyType key) {  
    int low=1, high=ST.length, mid, count=0;  
    while(low<=high) {  
        mid=(low+high)/2;  
        if(ST.elem[mid].key== key)  
            return mid; //找到了  
        if(ST.elem[mid].key> key)  
            high=mid-1; //在前半部分找  
        else  
            low=mid+1; //在后半部分找  
    }  
    return -1;  
}
```

# 折半查找

关键代码

a相当于ST.elem  
n相当于ST.length

```
int Binary_Search(int *a,int n,int key)
```

```
{
```

```
    int low,high,mid;
```

```
    low=1;
```

```
    high=n;
```

```
    while(low<=high)
```

```
    {
```

```
        mid=(low+high)/2;
```

```
        if (key<a[mid])
```

```
            high=mid-1;
```

```
        else if (key>a[mid])
```

```
            low=mid+1;
```

```
        else
```

```
            return mid;
```

```
    }
```

```
    return 0;
```

```
}
```

数据变化



n=10, key=62, low=1, high=10

mid=(1+10)/=5

# 折半查找

关键代码

a相当于ST.elem  
n相当于ST.length

```
int Binary_Search(int *a,int n,int key)
{
    int low,high,mid;
    low=1;
    high=n;
    while(low<=high)
    {
        mid=(low+high)/2;
        if (key<a[mid])
            high=mid-1;
        else if (key>a[mid])
            low=mid+1;
        else
            return mid;
    }
    return 0;
}
```

数据变化



n=10, key=62, high=10

mid=5  
a[mid]=47  
low=6

# 折半查找

关键代码

a相当于ST.elem  
n相当于ST.length

```
int Binary_Search(int *a,int n,int key)
{
    int low,high,mid;
    low=1;
    high=n;
    while(low<=high)
    {
        mid=(low+high)/2;
        if (key<a[mid])
            high=mid-1;
        else if (key>a[mid])
            low=mid+1;
        else
            return mid;
    }
    return 0;
}
```

数据变化



n=10, key=62, low=6

mid=(6+10)/2=8  
a[mid]=73  
high=7



## 9.1.2 折半查找（二分查找）

### 问题讨论

讨论1：若关键字不在表中，怎样得知并及时停止查找？

——典型标志是：当查找范围的上界 $<$ 下界时停止查找。

讨论2：如何计算二分查找的时间效率（ASL）？

## 9.1.2 折半查找（二分查找）

### 推导过程

经1次比较就查找成功的元素有1个 ( $2^0$ )，即中间值；

经2次比较就查找成功的元素有2个 ( $2^1$ )，即1/4处和3/4处；

3次比较就查找成功的元素有4个 ( $2^2$ )，即1/8, 3/8, 5/8, 7/8处

4次比较就查找成功的元素有8个 ( $2^3$ )，即1/16处, 3/16处……

……

则第m次比较时查找成功的元素应该有 ( $2^{m-1}$ ) 个。

注：为方便起见，假设表中全部n个元素刚好是  $2^m-1$  个，此时暂不讨论第m次比较后还有剩余元素的情况。

## 9.1.2 折半查找（二分查找）

### 推导过程

全部比较总次数为  $1 \times 2^0 + 2 \times 2^1 + 3 \times 2^2 + 4 \times 2^3 \cdots +$

$$m \times 2^{m-1} = \sum_{j=1}^m j \cdot 2^{j-1}$$

## 9.1.2 折半查找（二分查找）

### 推导过程

请注意：ASL的含义是“平均每个数据的查找时间”，  
而前式是n个数据查找时间的总和，所以：

$$ASL = \frac{1}{n} \sum_{j=1}^m j \cdot 2^{j-1} = \left[ \frac{n+1}{n} \log_2(n+1) - 1 \right] \approx \log_2 n$$

（详细推导过程见教材P221的附录1）

## 9.1.2 折半查找（二分查找）

### 练习题

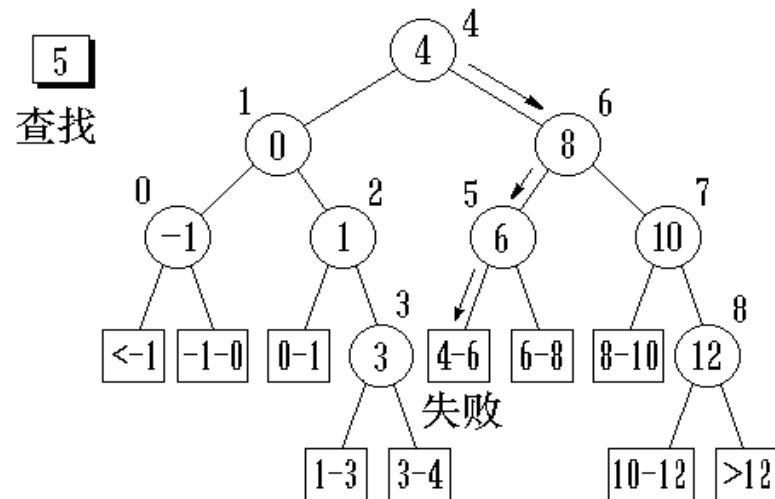
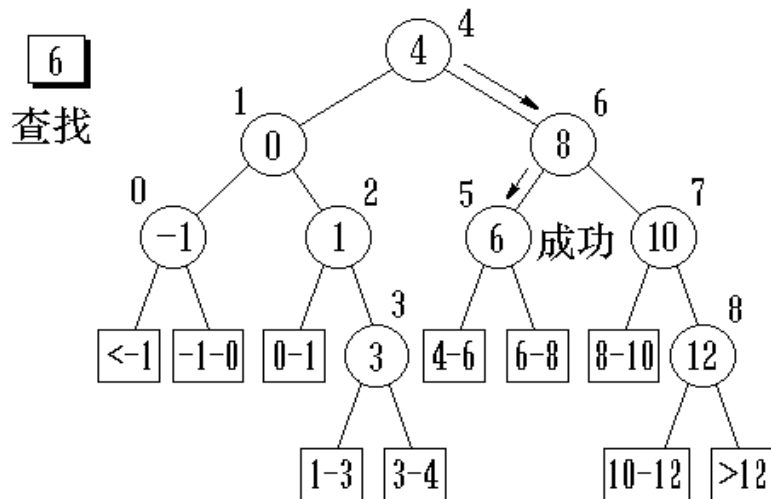
课堂练习1（多选题）使用折半查找算法时，要求被查文件：

A. 采用链式存储结构    B. 记录的长度 $\leq 12$

☒ C. 采用顺序存储结构    ☒ D. 记录按关键字递增有序

课堂练习2 在有序线性表a[20]上进行折半查找，  
则平均查找长度为 3.5。

# 再用二分查找树来分析ASL(参见教材P220)



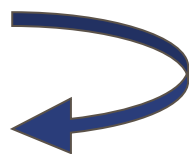
- 找到有序表中任一记录的过程就是：走了一条从根结点到与该记录对应结点的路径。
- 比较的关键字个数为：该结点在判定树上的层次数。
- 查找成功时比较的关键字个数最多不超过树的深度  $d$ ：  

$$d = \lfloor \log_2 n \rfloor + 1$$
- 查找不成功的过程就是：走了一条从根结点到外部结点的路径。查找不成功最多比较次数也为  $d$

## 9.1.2 折半查找（二分查找）

### 顺序查找其它改进算法

讨论：对顺序查找除了折半改进法外，还有无其他改进算法？有，例如斐波那契查找、插值查找、分块查找法。



分块查找（索引顺序查找）

## 9.1.3 分块查找（索引顺序查找）

思路：先让数据**分块有序**，即分成若干子表，要求每个子表中的数据元素值都比后一块中的数值小（但子表内部未必有序）。然后将各子表中的最大关键字构成一个**索引表**，表中还要包含每个子表的起始地址（即头指针）。

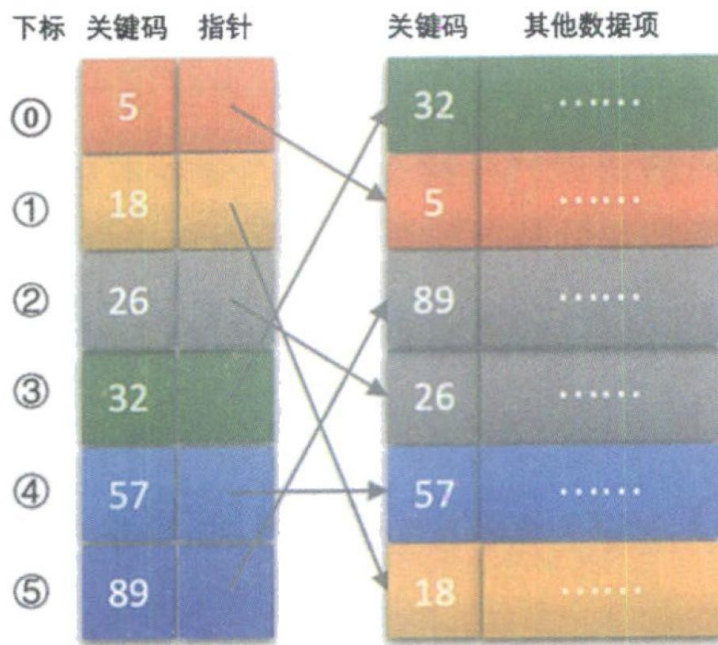
例：

22	12	13	8	9	20	33	42	44	38	24	48	60	58	74	49	86	53
----	----	----	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

特点：块间有序，块内无序



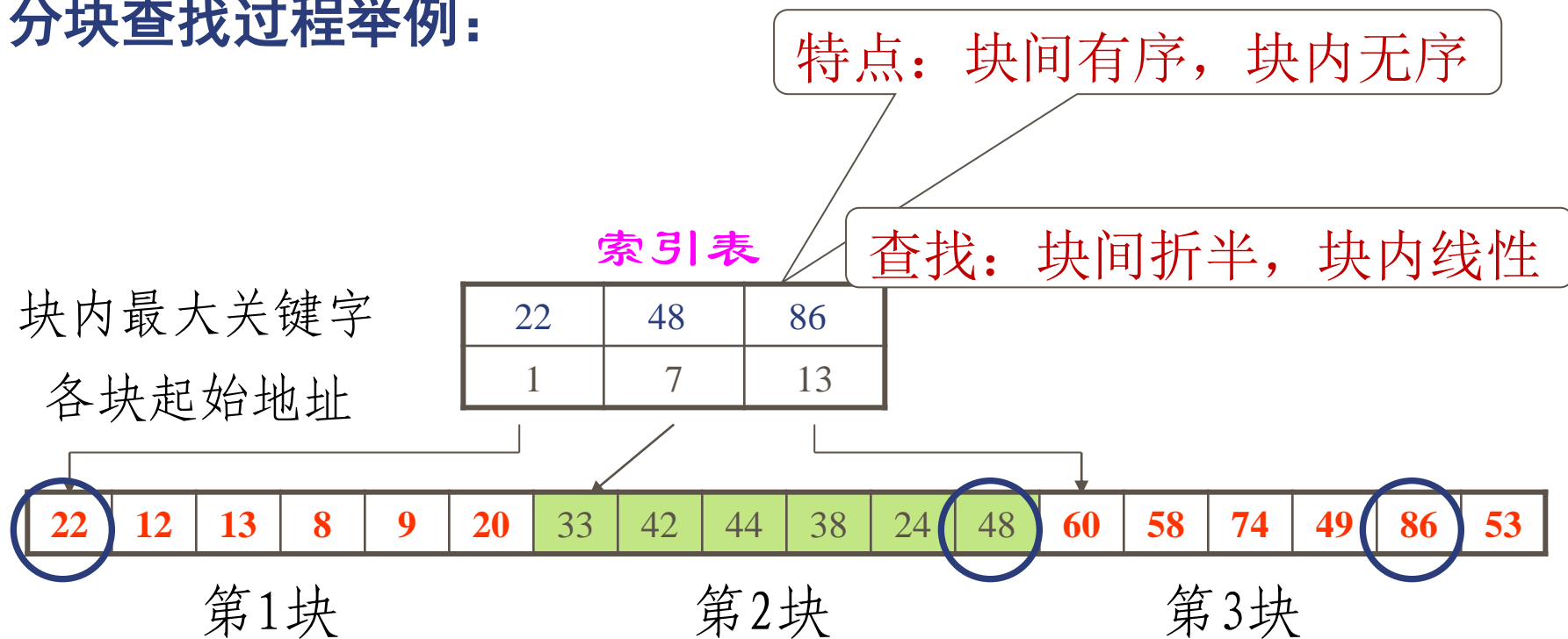
## 9.1.3 分块查找（索引顺序查找）



稠密索引：将数据集中的每个记录对应一个索引项（索引项按关键码有序排列的）

## 9.1.3 分块查找（索引顺序查找）

分块查找过程举例：



## 9.1.3 分块查找（索引顺序查找）

- 1) 分块有序（升序或降序）  
——第i块中的最大（小）值小（大）于第i+1块中的最（大）小值
- 2) 查找
  - (1) 先查索引表——折半查找
  - (2) 再查顺序表——顺序查找块间有序，块内无序

```
typedef struct
    { int key;}  Elemtyp;
typedef struct {
    Elemtyp *elem;
    int length;
} SSTable;
# define BLOCK_NUM 3
```

## 9.1.3 分块查找（索引顺序查找）

查找步骤分两步进行：

- ① 对索引表使用折半查找法（因为索引表是有序表）；
- ② 确定了待查关键字所在的子表后，在子表内采用顺序查找法（因为各子表内部是无序表）；

## 9.1.3 分块查找（索引顺序查找）

```
Typedef struct {
    int Maxkey;
    int next; } BLKNode, IT[BLOCK_Num];
int Search_Bin(SSTable ST, int key) { //折半查找
    int i, p, low, high, mid;
    printf( "Create index table, Input Maxkey& next\n" )
    for(i=1; i<=BLOCK_NUM; i++)
        scanf( "%d, %d" , &IT[i].Maxkey, &IT[i].next);
    if(key>IT[BLOCK_NUM].Maxkey) return(0);
    low = 1; high=BLOCK_NUM;
    while (low < =high){
        mid = (low+high)/2;
        if (IT[mid].Maxkey>=key) high=mid-1;
        else low=mid+1;
    }
```

## 9.1.3 分块查找（索引顺序查找）

```
i=IT[low].next;
ST.elem[0].key = key;
if(low!=BLOCK_NUM) p=IT[low+1].next;
else p=ST.length+1;
while(ST.elem[i%p].key != key ) i++ ;
return( i%p );
}
```

性能分析：

$$\begin{aligned} ASL(b|s) &= L_b + L_w = (b+1)/2 + (s+1)/2 \\ &= (b+s+2)/2 = (n/s+s)/2+1 \end{aligned}$$

b —— 分块的个数  $b=n/s$

s —— 每块中的记录个数

n —— 记录的总个数

$L_b$  —— 确定所在的块

$L_w$  —— 块内查找

## 9.1.3 分块查找（索引顺序查找）

例如：当 $n=9$ ， $s=3$ 时

分块法的 $ASL_{bs}=3.5$

折半法的 $ASL \approx \log_2 n = 3.1$

顺序法的 $ASL = (1+n)/2 = 5$

但折半法要预先全排序，仍需时间。

## 9.2 动态查找表

**特点：**表结构在查找过程中动态生成。

**要求：**对于给定值key, 若表中存在其关键字等于key的记录，则查找成功返回；否则插入关键字等于key的记录。

典型的动态表——二叉排序树



## 9.2 动态查找表

---

表结构本身是在查找过程中动态生成的。

基本操作：

`InitDSTable(&DT);` //构造一个空的动态查找表DT

`DestroyDSTable(&DT);` //销毁表

`SearchDSTable(DT, key);` //查找关键字为key的数据元素

`InsertDSTable(&DT, e);`

`DeleteDSTable(&DT, key);`

`TraverseDSTable(DT, visit());` //遍历查找表

## 9.2 动态查找表

典型的动态表——二叉排序树

- 一、二叉排序树的定义
- 二、二叉排序树的插入与删除
- 三、二叉排序树的查找分析
- 四、平衡二叉树(难点)

## 9.2.1 二叉排序树的定义

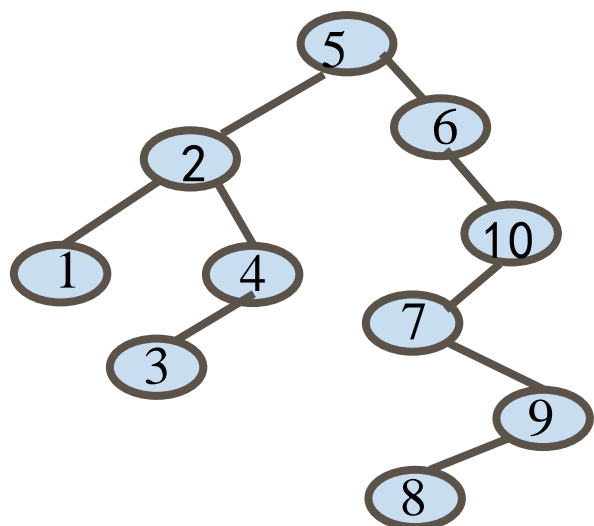
---

或是一棵空树；或者是具有如下性质的非空二叉树：

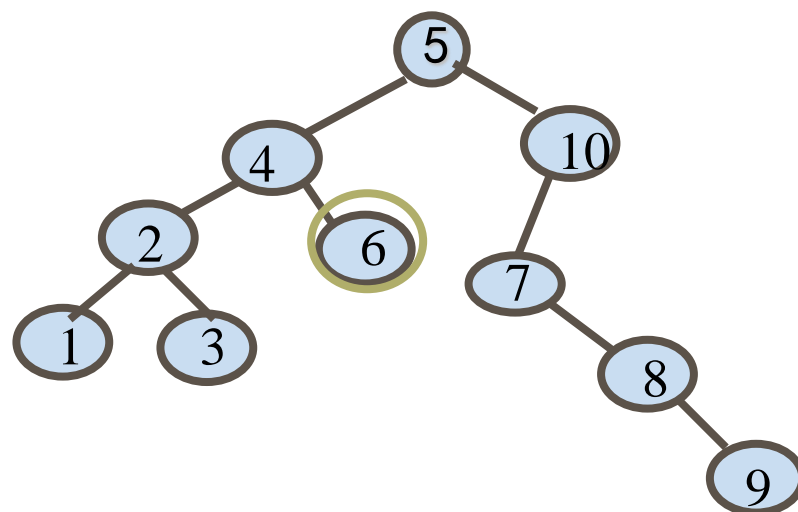
- (1) 左子树的所有结点均小于根的值；
- (2) 右子树的所有结点均大于根的值；
- (3) 它的左右子树也分别为二叉排序树。

## 9.2.1 二叉排序树的定义

例：下列2种图形中，哪个不是二叉排序树？



(a)



(b)

想一想：对它中序遍历之后是什么效果？

## 9.2.1 二叉排序树的定义

---

```
typedef struct { int key; } ElemType;

typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild,*rchild;
} BiTNode, * BiTree;
```

## 9.2.1 二叉排序树的定义

### 构造过程:

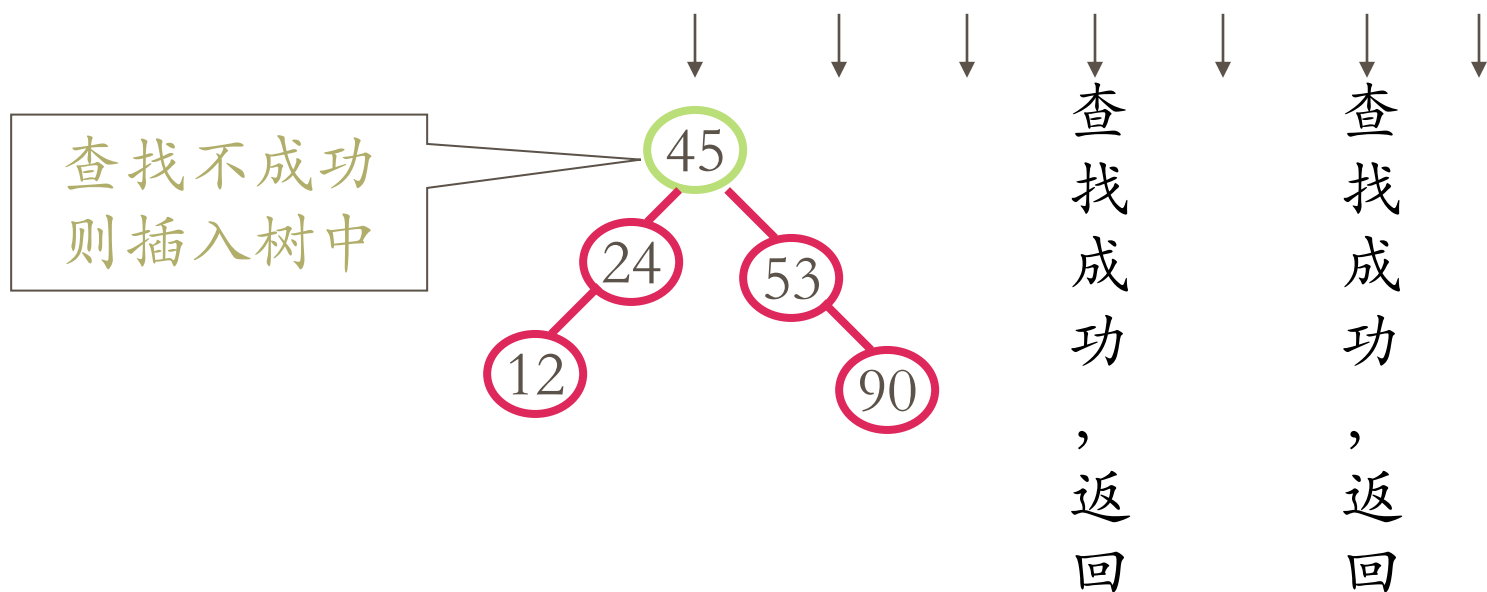
从空树出发, 依次插入 $R_1 \sim R_n$ 各数据值:

(1) 如果二叉排序树是空树, 则插入结点就是二叉排序树的根结点;

(2) 如果二叉排序树是非空的, 则插入值与跟结点比较, 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。

## 9.2.1 二叉排序树的定义

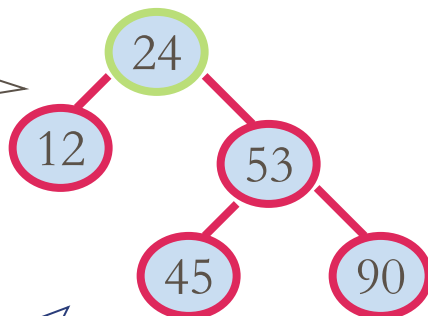
例：输入待查找的关键字序列= (45, 24, 53, 45, 12, 24, 90)



## 9.2.2、二叉排序树的插入与删除

如果改变输入顺序为 (24, 53, 45, 45, 12, 24, 90) ,

则生成的二叉排序树形态不同



这种既查找又插入的过程称为动态查找

↓  
查找成功，返回

↓  
查找成功，返回



## 9.2.2、二叉排序树的插入与删除

将线性表构造成二叉排序树的优点

- ① 查找过程与顺序结构有序表中的折半查找相似，查找效率高；
- ② 中序遍历此二叉树，将会得到一个关键字的有序序列（即实现了排序运算）；
- ③ 如果查找不成功，能够方便地将被查元素插入到二叉树的叶子结点上，而且插入或删除时只需修改指针而不需移动元素。

## 9.2.2、二叉排序树的插入与删除

---

### 1 二叉排序树的查找&插入算法如何实现？

思路：查找不成功，生成一个新结点s，插入到二叉排序树中；查找成功则返回。

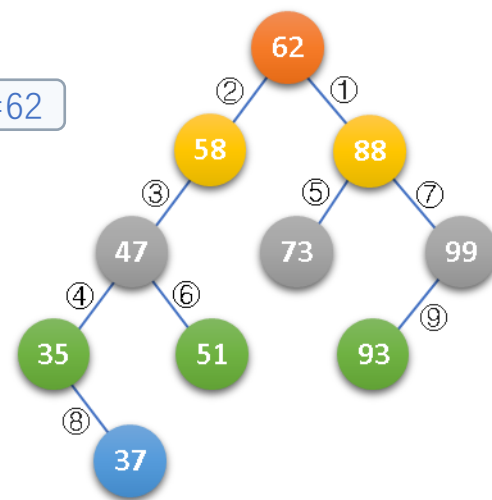
# 二叉树排序查找

关键代码

```
Status SearchBST(BiTree T, int key, BiTree f, BiTree &p)
{
    if (!T)
    {
        p = f;
        return FALSE;
    }
    else if (key==T->data)
    {
        p = T;
        return TRUE;
    }
    else if (key<T->data)
        return SearchBST(T->lchild, key, T, p);
    else
        return SearchBST(T->rchild, key, T, p);
}
```

key=93, T->data=62

数据变化

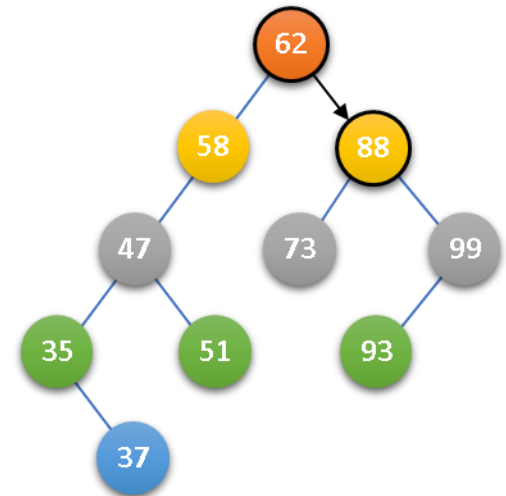


# 二叉树排序查找

关键代码

```
Status SearchBST(BiTree T, int key, BiTree f, BiTree &p)
{
    if (!T)
    {
        p = f;
        return FALSE;
    }
    else if (key==T->data)
    {
        p = T;
        return TRUE;
    }
    else if (key<T->data)
        return SearchBST(T->lchild, key, T, p);
    else
        return SearchBST(T->rchild, key, T, p);
}
```

数据变化



key=93, T->data=62

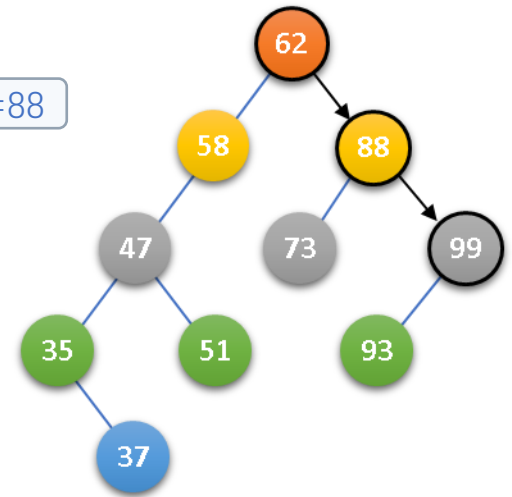
# 二叉树排序查找

关键代码

```
Status SearchBST(BiTree T, int key, BiTree f, BiTree &p)
{
    if (!T)
    {
        p = f;
        return FALSE;
    }
    else if (key==T->data)
    {
        p = T;
        return TRUE;
    }
    else if (key<T->data)
        return SearchBST(T->lchild, key, T, p);
    else
        return SearchBST(T->rchild, key, T, p);
}
```

key=93, T->data=88

数据变化



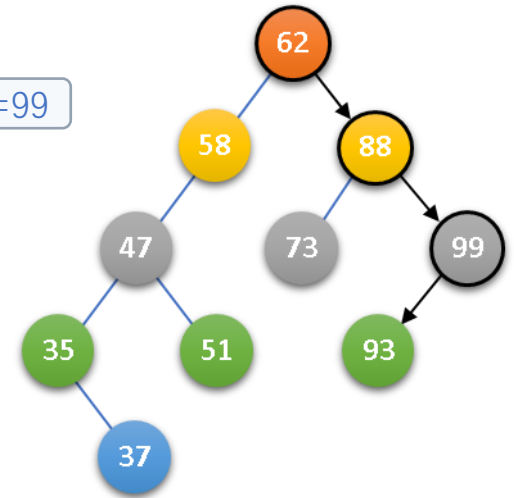
# 二叉树排序查找

关键代码

```
Status SearchBST(BiTree T, int key, BiTree f, BiTree &p)
{
    if (!T)
    {
        p = f;
        return FALSE;
    }
    else if (key==T->data)
    {
        p = T;
        return TRUE;
    }
    else if (key<T->data)
        return SearchBST(T->lchild, key, T, p);
    else
        return SearchBST(T->rchild, key, T, p);
}
```

key=93, T->data=99

数据变化

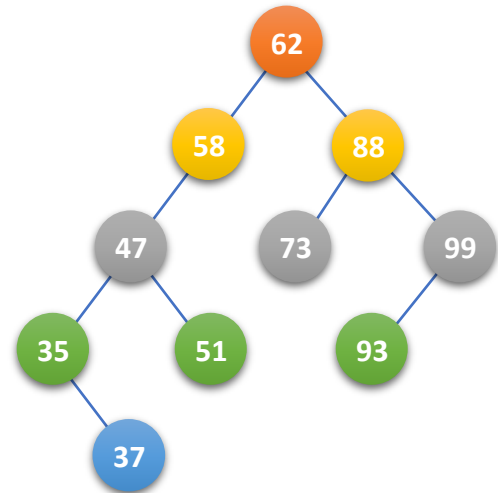


## 9.2.2、二叉排序树的插入与删除

```
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->dat.key)
            q->lchild = S;
        else
            q->rchild = S;
    }
    return;
}
```

数据变化

数据插入： 62,88,58,47,35,73,51,99,37,93



## 9.2.2、二叉排序树的插入与删除

输入一组数据元素的序列，构造二叉排序树的算法

```
void Creat_BST( BiTree &T ) {  
    int x; BiTree S; T=NULL;  
    while ( scanf ( "%d" ,&x), x!=0 ) {  
        S = (BiTNode *) malloc(sizeof(BiTNode));  
        S->data.key = x;  
        S->lchild = NULL;  
        S->rchild = NULL;  
        Insert_BST( T, S );  
    }  
    return;  
}
```



## 9.2.2、二叉排序树的插入与删除

---

### 2 二叉排序树的删除操作如何实现？

对于二叉排序树，删除树上一个结点相当于删除有序序列中的一个记录，要求删除后仍需保持二叉排序树的特性。

## 9.2.2、二叉排序树的插入与删除

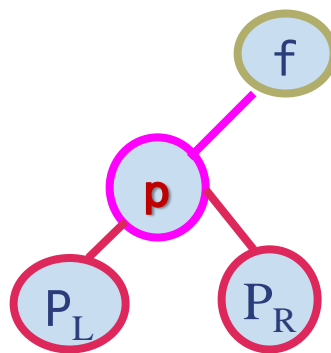
### 2 二叉排序树的删除操作如何实现？

如何删除一个结点？

假设： $*p$ 表示被删结点的指针； $P_L$ 和 $P_R$ 分别表示 $*p$ 的左、右孩子指针；

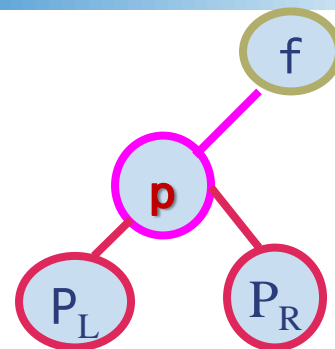
$*f$ 表示 $*p$ 的双亲结点指针；并假定 $*p$ 是 $*f$ 的左孩子；

则可能有三种情况：



## 9.2.2、二叉排序树的插入与删除

### 2 二叉排序树的删除操作如何实现？



\*p为叶子：删除此结点时，直接修改\*f指针域即可；

\*p只有一棵子树（或左或右）：令 $P_L$ 或 $P_R$ 为\*f的左孩子即可；

\*p有两棵子树：情况最复杂 →

## 9.2.2、二叉排序树的插入与删除

难点：\*p有两棵子树时，如何进行删除操作？

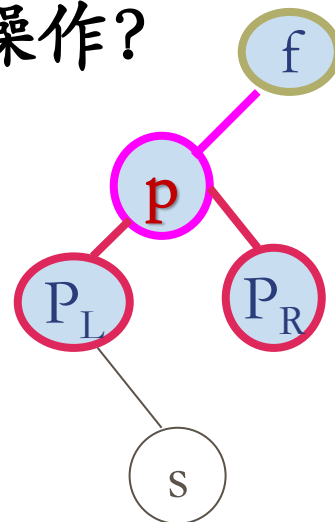
分析：

设删除前的中序遍历序列为：

$\cdots P_L s_L s p P_R f \cdots$  //显然p的直接前驱是s

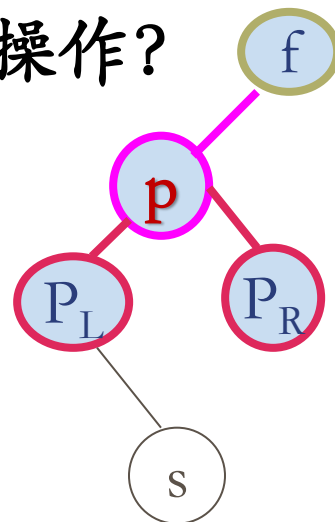
//s是p左子树最右下方的结点

希望删除p后，其它元素的相对位置不变。



## 9.2.2、二叉排序树的插入与删除

难点：\*p有两棵子树时，如何进行删除操作？



分析：

有两种解决方法：

法1：令p的左子树为 f的左子树，p的右子树接为s的右子树； // 即  $f_L = P_L$ ；  $S_R = P_R$ ；

法2：直接令s代替p，s的左子树接为  $P_L$   
// s为p左子树最右下方的结点