

# 第九章            查    找

## 9. 1    静态查找表

## 9. 2    动态查找表

## 9. 3    哈希表

# 一、查找表的基本概念

**查找表：** 是由**同一类型的数据元素（或记录）**构成的集合。  
在查找表中数据元素除了同属一个集合，不考虑元素之间其它关系.

学号	姓名	专业	年龄
94801	王洪	计算机	17
94802	孙文	计算机	18
94803	谢军	计算机	18
94804	李辉	计算机	20
94805	沈祥福	计算机	25
94806	余斌	计算机	19

## 二 查找的有关概念

### 1 静态查找表、动态查找表

查找表最基本的操作是查找，常用的查找操作：

- (1) 查询某个“特定”元素是否在表中；
- (2) 检索某个“特定”元素的各种属性；
- (3) 查找某个“特定”元素是否在表中，若不在，则将该元素插入表中；
- (4) 从查找表中删除某个“特定”元素；

---

若对查找表只有 (1)、(2) 两种查找，此类查找表称为**静态查找表**；

若查找表还提供 (3)、(4) 两种查找操作，在查找过程中同时插入表中不存在的数据元素或从查找表中删除已存在的某个数据元素，此类表为**动态查找表**。

## 2 记录、关键字

**记录：**由若干数据项构成的数据元素称为记录.

**关键字：**数据元素（或记录）中某个数据项.

例：学生管理系统，若想按姓名查找，可将姓名作为关键字，若想按学号查找，可将学号作为关键字。

学号	姓名	专业	年龄
94801	王洪	计算机	17
94802	孙文	计算机	18
94803	谢军	计算机	18
94804	李辉	计算机	20
94805	沈祥福	计算机	25
94806	余斌	计算机	19

说明：

①记录的任何数据项都可用为关键字；

②若此关键字的值能唯一标识记录，则称该关键字为主关键字，否则次关键字；

例：学号是主关键字，姓名是次关键字

学号	姓名	专业	年龄
94801	王洪	计算机	17
94802	孙文	计算机	18
94803	谢军	计算机	18
94804	李辉	计算机	20
94805	沈祥福	计算机	25
94806	余斌	计算机	19

### 3. 查找

根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素。若表中存在这样的—个记录，则称**查找是成功的**，此时查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置；若表中不存在关键字等于给定值的记录，则称**查找不成功**。

学号	姓名	专业	年龄
94801	王洪	计算机	17
94802	孙文	计算机	18
94803	谢军	计算机	18
94804	李辉	计算机	20
94805	沈祥福	计算机	25
94806	余斌	计算机	19

### 三 查找表的组织与查找

如何在查找表中查找？取决于如何组织查找表

例1： 查找索引号为TP312. 12的图书

- 1) 若图书馆中的图书无序地摆放，则只能顺序查找；
- 2) 若图书按索引号顺序摆放，则可先找到TP类图书，再找到TP3

例2 在词典中查找单词 are

- 1) 先找到a打头的单词，再在a打头的单词中找
- 2) 若词典中单词无序排列，则只能顺序查找；

本章介绍：

**静态查找表**的组织方法与查找

**动态查找表**的组织方法与查找

**哈希表**的组织方法与查找

查找表的数据元素类型定义为：

```
typedef struct {  
    KeyType key;    //关键字域  
    ...;           //其它域  
} SElemType;
```



## 9.1 静态查找表

只提供如下两种查找的查找表，称为静态查找表

- (1) 查询某个“特定”元素是否在表中；
- (2) 检索某个“特定”元素的各种属性；

对静态查找表介绍三种组织与查找方法

顺序表及查找

有序表及查找

索引顺序表及查找

## 一、顺序表的查找

**查找表组织：**查找表用线性表表示。即将查找表的记录排成一个记录序列。L1=(45, 53, 12, 3, 37, 24, 100, 61, 90, 78)

假设静态查找表用顺序表存储，其类型定义如下：

```
typedef struct {  
    ElemType *elem; // 数据元素存储空间基址，建表时  
                    // 按实际长度分配，0号单元留空  
    int      length; // 表的长度  
} SSTable;
```

## 顺序查找的查找过程：

从表中最后一个记录 (或第一个记录) 开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则**查找成功**，找到所查记录；反之，**查找不成功**。

找64

例

0	1	2	3	4	5	6	7	8	9	10	11
	5	43	19	21	37	56	64	75	80	88	22
							↑	↑	↑	↑	↑

## 回顾顺序表的查找算法：

```
int location( SqList L, ElemType &e, Status (*compare)(ElemType, ElemType))
{
    k = 1;
    p = L.elem;
    while ( k<=L.length && !(*compare)(*p++,e))
        k++;
    if ( k<= L.length) return k;
    else return 0;
}
```

— 上述算法中的基本操作是：**compare**，为了避免“出界”，同时需作 **k<=L.length** 的判断。

为此，改写顺序表的查找算法，算法中附设监视哨，以避免循环时每一步都要判别是否数组出界。

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

```
// 在顺序表ST中顺序查找其关键字等于key的数据元素。若找到，则函数  
值为该元素在表中的位置，否则为0。
```

```
    ST.elem[0].key = key;    // “哨兵”
```

```
    i=ST.length;
```

```
    while (ST.elem[i].key!=key)    i--;    // 从后往前找
```

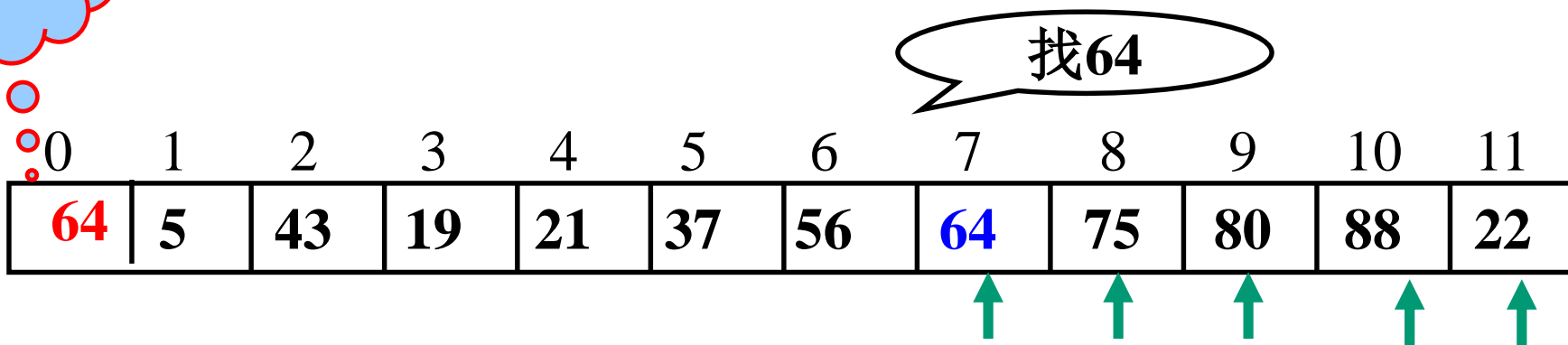
```
    return i;    // 找不到时，i为0
```

```
}
```

监视哨

找64

0	1	2	3	4	5	6	7	8	9	10	11
64	5	43	19	21	37	56	64	75	80	88	22



比较次数:

查找第 $n$ 个元素: 1

比较次数=5

查找第 $n-1$ 个元素: 2

.....

查找第1个元素:  $n$

查找第 $i$ 个元素:  $n+1-i$

查找失败:  $n+1$

## 分析顺序查找的性能

**定义：**查找算法的**平均查找长度** (Average Search Length) 为确定记录在查找表中的位置，**需和给定值进行比较的关键字个数的期望值**

对于含有n个记录的表，查找成功时的平均查找长度为

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

其中：**n** 为表长， **$P_i$**  为查找表中第i个记录的概率，且  $\sum_{i=1}^n P_i = 1$

**$C_i$** 为找到该记录时，曾和给定值比较过的关键字的个数。

对顺序表而言,  $C_i = n-i+1$

$$ASL = nP_1 + (n-1)P_2 + \dots + 2P_{n-1} + P_n$$

在等概率查找的情况下:  $P_i = 1/n$

顺序表查找的平均查找长度为:

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

**顺序查找的优点**是查找算法简单, 既适用于顺序表, 也适用于单链表, 同时对表中元素的排列次序无要求。

**缺点**是平均查找长度较大, 特别不适用于表长较大的查找表。



## 二、有序表的查找

**有序表**：若查找表中的记录按关键字有序，则称为有序表。

以有序表表示静态查找表时，Search函数可用折半查找来实现。

折半查找法（也称为二分查找法）：

1) **基本思想**：

将给定值与查找范围中间位置的记录关键字比较：

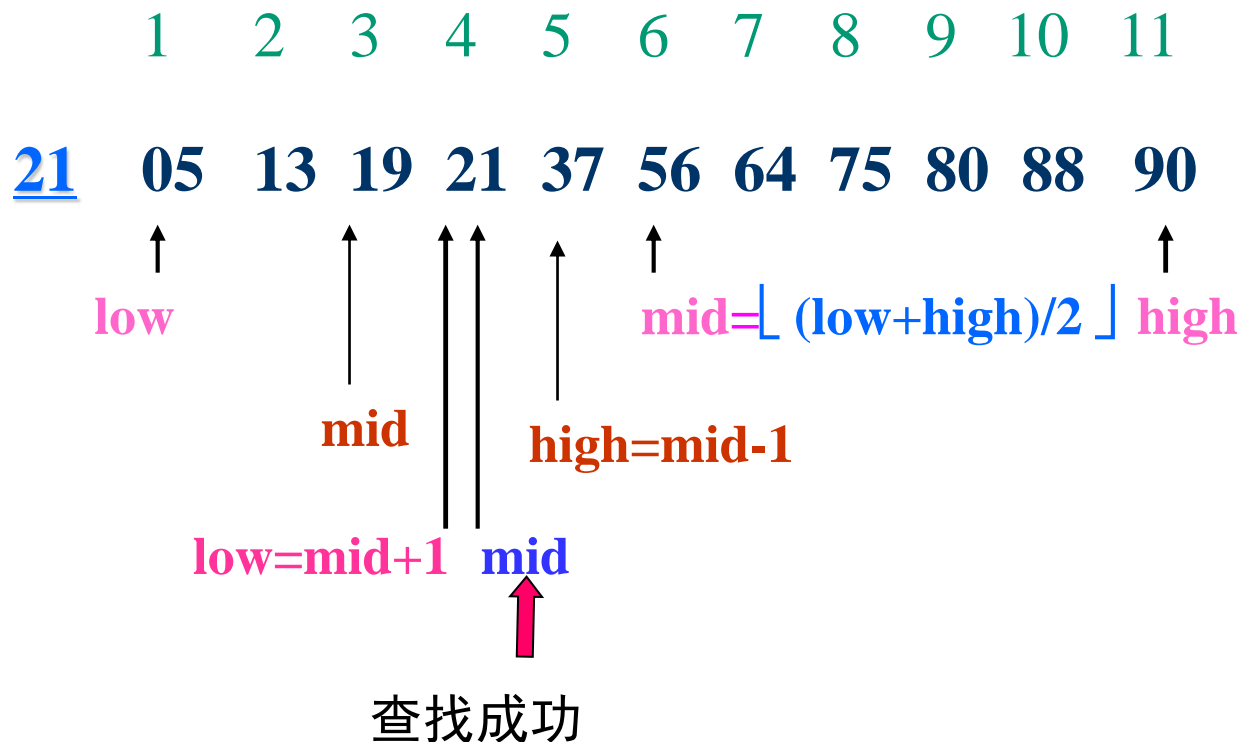
< ：继续在前半个表中用折半查找法查找

= ：查找成功，返回记录位置

> ：继续在后半个表中中用折半查找法查找

例1 L2=(05 13 19 21 37 56 64 75 80 88 90),

查找 Key=21的记录



例2 L2=(**05 13 19 21 37 56 64 75 80 88 90**),

查找 Key=20的记录

1 2 3 4 5 6 7 8 9 10 11

**05 13 19 21 37 56 64 75 80 88 90**

## 算法实现:

- ☆ 设表长为 $n$ ,  $low$ 、 $high$ 和 $mid$ 分别指向待查元素所在区间的下界、上界和中点,  $key$ 为给定值
- ☆ 初始时, 令 $low=1$ ,  $high=n$ ,  $mid=\lfloor (low+high)/2 \rfloor$
- ☆ 让 $key$ 与 $mid$ 指向的记录比较
  - 若 $key==ST.elem[mid].key$ , 查找成功
  - 若 $key<ST.elem[mid].key$ , 则 $high=mid-1$
  - 若 $key>ST.elem[mid].key$ , 则 $low=mid+1$
- ☆ 重复上述操作, 直至 $low>high$ 时, 查找失败

折半查找的查找算法:

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

```
    // 在有序表ST中折半查找其关键字等于key的数据元素。
```

```
    // 若找到，则函数值为该元素在表中的位置，否则为0。
```

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

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

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

```
    if (key == ST.elem[mid].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
```

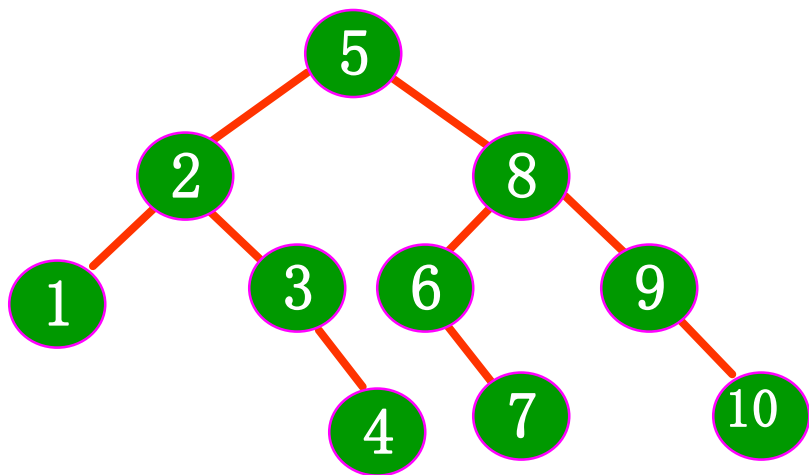
```
}
```

## 折半查找的性能分析

折半查找法查找过程可用判定树描述

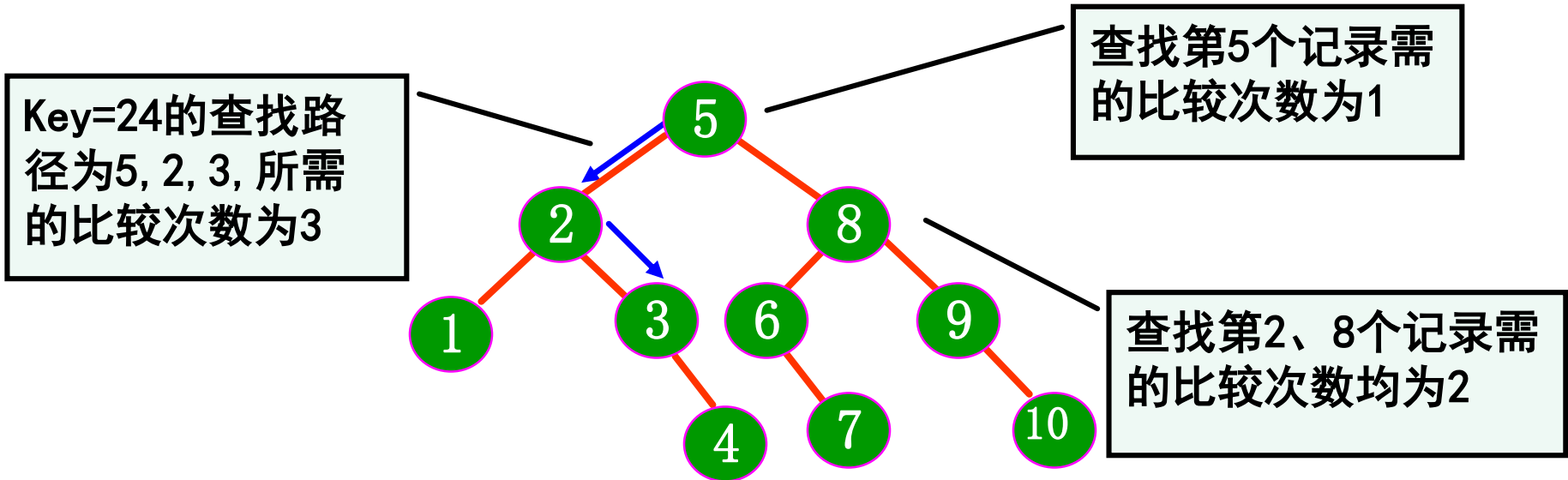
1 2 3 4 5 6 7 8 9 10

**例** L2=( 3, 12, 24, 37, 45, 53, 61, 78, 90, 100 ), 查找过程可用如下判定树描述:



树中每个结点表示表中一个记录，结点中的值为该记录在表中的位置，通常称这个描述查找过程的二叉树为**判定树**。

1 2 3 4 5 6 7 8 9 10  
L2=( 3, 12, 24, 37, 45, 53, 61, 78, 90, 100 )



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相应的结点的路径，和给定值进行比较的关键字个数恰为该结点在判定树上的层次数。

设查找每一个记录的概率相同，即均为1/10，

平均查找长度ASL = 在查找过程中与给定值比较的关键字个数的数学期望值  $\sum P_i C_i = (1 + 2 \times 2 + 4 \times 3 + 3 \times 4) / 10$

- 折半查找法效率比顺序查找高;平均查找长度

$$ASL_{bs} = \log_2 (n+1) - 1$$

- 要求表中的记录按关键字有序;
- 表中的记录要用顺序存储结构(对线性链表无法进行折半查找)

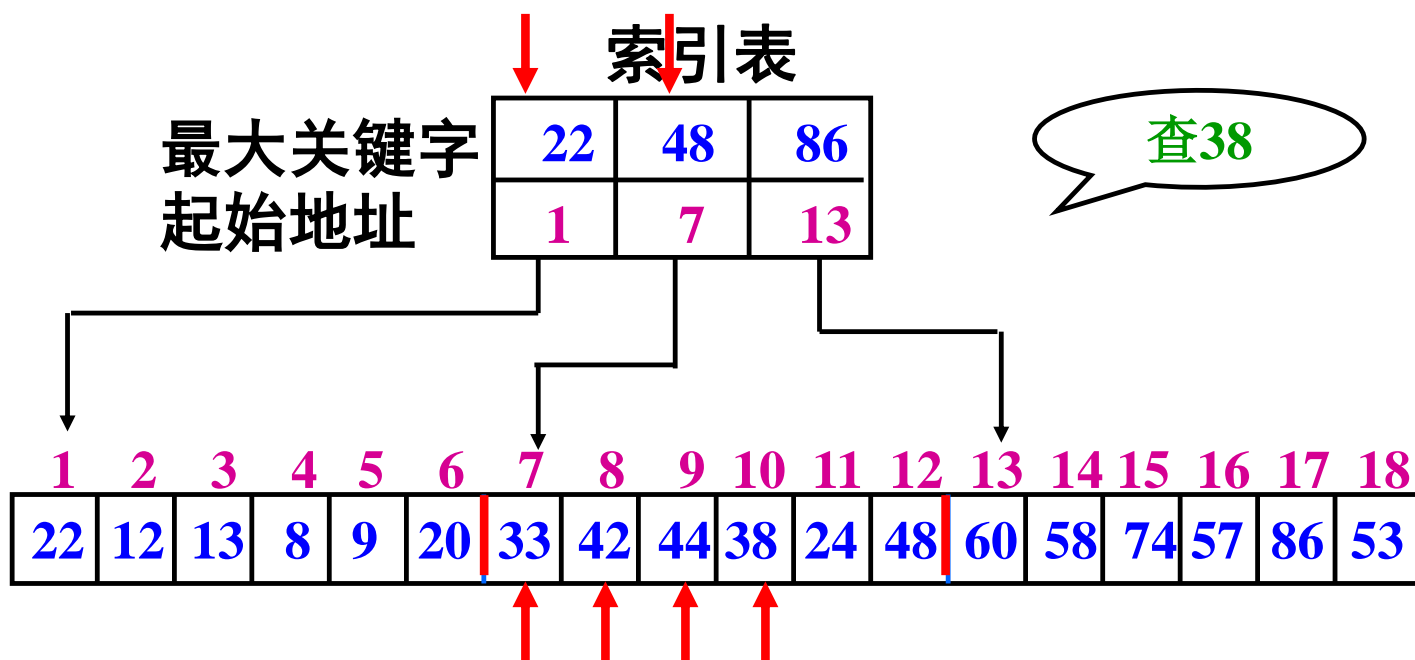
例:在有1000个记录的查找表查找,顺序查找法平均要比较500次,折半查找法平均要比较9次,可见折半查找法效率比顺序查找高。



### 三、索引顺序表的查找

若以索引顺序表表示静态查找表，则Search函数可用分块查找来实现。

**分块查找**又称**索引顺序查找**，在此查找法中，除表本身以外，尚需建立一个“索引表”。**索引顺序表 = 索引表 + 顺序表**



**分块查找的基本思想：**首先查找索引表(因为索引表是有序表，故可采用二分查找或顺序查找)，以确定待查的结点在哪一块；然后在已确定的那一块中进行顺序查找。

分块查找算法是折半查找和顺序查找的简单合成，算法效率介于顺序查找和二分查找之间。

索引顺序表的平均查找长度为在索引表中进行查找的平均查找长度和在顺序表中进行查找的平均查找长度之和。

### 查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

# 静态查找表三种组织与查找方法

顺序表

有序表

索引顺序表

顺序查找

二分查找

分块查找

## 9.2 动态查找表

如果应用问题涉及的数据量很大，而且数据经常发生变化，如图书馆经常购进图书，每购进新书，需将新书记录插入图书表，对这类表除了提供(1)、(2)两种查找外，还要提供动态查找功能：

- (3) 查找某个“特定”元素是否在表中，若不在，则将该元素插入表中；
- (4) 查找某个“特定”元素是否在表中，若在，从表中删除该元素；



线性表(无序)：顺序查找效率低；

有序表：二分查找法需用顺序存储结构存储有序表，  
插入删除操作要移动元素；

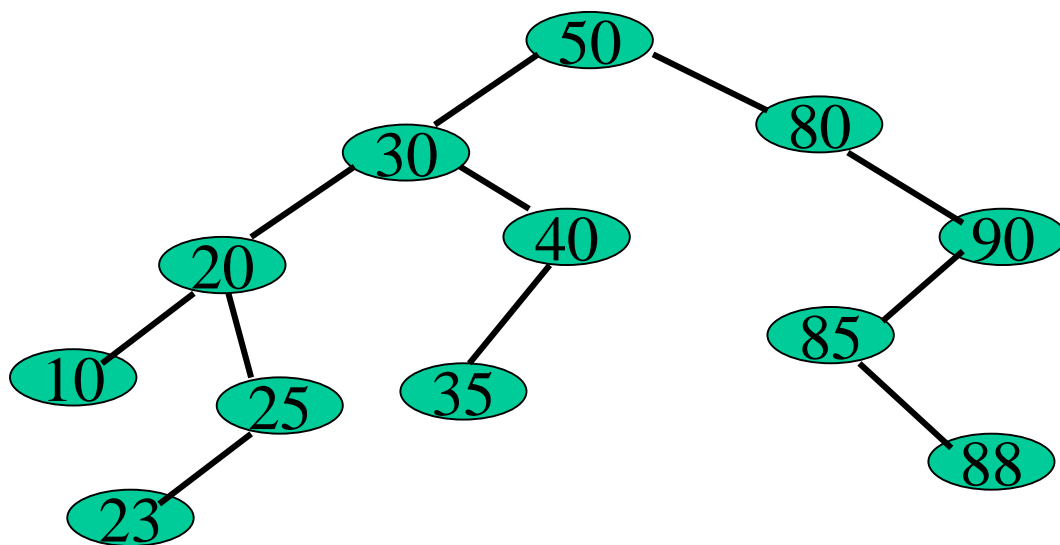
本节介绍动态查找表的一种组织形式—— 二叉排序树

# 一、二叉排序树（二叉查找树）

## 1. 定义：

二叉排序树或者是一棵空树；或者是具有如下特性的二叉树：

- (1) 若它的左子树不空，则左子树上所有结点的值均小于根结点的值；
- (2) 若它的右子树不空，则右子树上所有结点的值均大于根结点的值；
- (3) 它的左、右子树也都分别是二叉排序树。



按中序遍历二叉排序树所得到的中序序列是一个递增有序序列。

在下面讨论二叉排序树的操作中，使用二叉链表作为存储结构。二叉排序树的二叉链表存储表示如下：

```
typedef struct
{   KeyType key;           //关键字项
    InfoType otherinfo;    //其他数据项
}ElemType;                //每个结点的数据域的类型

typedef struct BSTNode
{   ElemType data;
    struct BSTNode *lchild,*rchild;
}BSTNode,*BSTree;
```

## 2 二叉排序树的查找

在二叉排序树中查找关键字为key的记录.

### 基本思想

若二叉排序树为空树，**查找失败**，返回null或 0；

否则，将key与根结点的关键字比较：

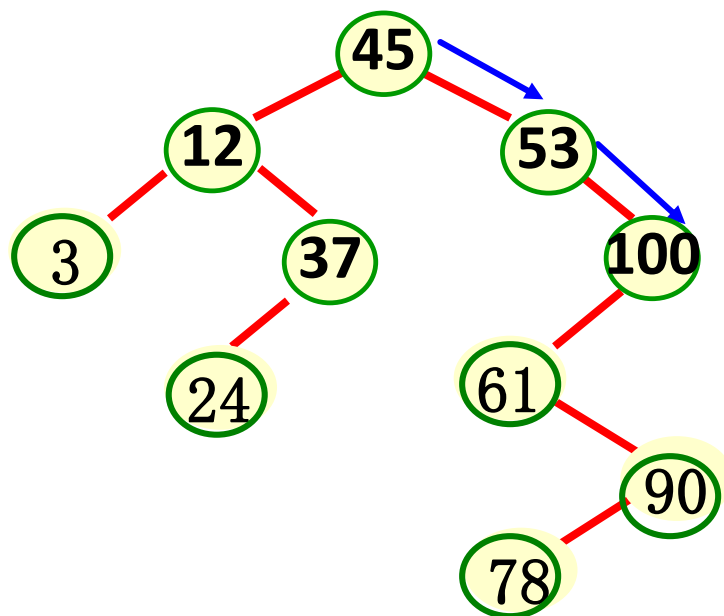
若key = 根结点的关键字，**查找成功**，返回该记录在二叉排序树中的位置；

若key < 根结点的关键字，继续在**左子树**中查找；

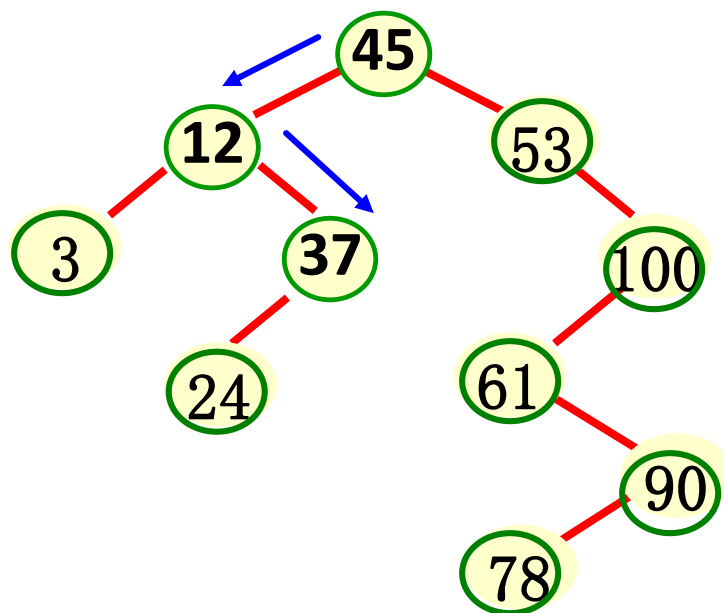
若key > 根结点的关键字，继续在**右子树**中查找；



例如在如下二叉排序树中查找关键字为100的记录(树中结点内的数均为记录的关键字)。



又如在如下二叉排序树中查找关键字为40的记录。



## 查找过程算法描述如下：

```
BiTree SearchBST(BiTree T, KeyType key) {
```

```
    /*在根指针T所指二叉排序树中递归地查找关键字等于key的记录，若查找成功，则返回指向该数据元素结点的指针，否则返回空指针*/
```

```
    if ((!T) || key == T->data.key) return(T); //查找结束
```

```
    else if (key < T->data.key)
```

```
        return SearchBST(T->lchild, key);
```

```
            //在左子树中继续查找
```

```
    else
```

```
        return SearchBST(T->rchild, key); //在右子树中继续查找
```

```
}
```

### 三. 二叉排序树的插入、删除

二叉排序树是动态查找表的组织形式, 动态查找表要提供插入、删除数据元素的操作。

二叉排序树是作为动态查找表的组织形式, 目的是获得较高的查找效率并且能方便地进行插入删除操作。因此在二叉排序树中插入、删除结点的原则是: 插入、删除结点后仍是二叉排序树。

## 1) 二叉排序树的插入

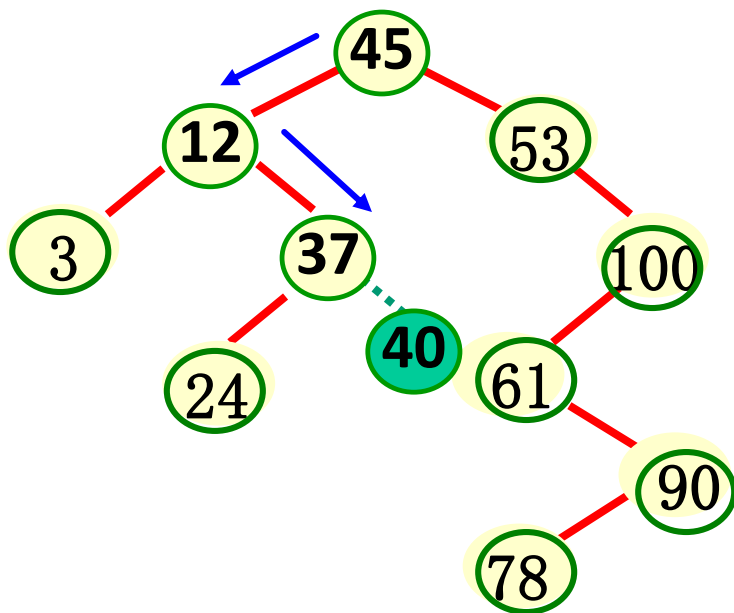
从查找的过程容易得出**插入的算法**：

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**叶子结点**，并且是**查找**

**不成功时，查找路径上最后一个结点的左孩子或右孩子。**

例如在如下二叉排序树中查找关键字为40的记录。



## 课堂作业：

1.已知一表为（25, 18, 46, 2, 53, 39, 32, 4, 74, 67, 60, 11），按表中顺序依次插入初始为空的二叉排序树，要求：

(1)画出建立的二叉排序树。

(2)求出在等概率情况下查找成功的平均查找长度。

## 原查找算法:

```
BiTree SearchBST(BiTree T, KeyType key) {
```

```
    /*在根指针T所指二叉排序树中递归地查找关键字等于key的记录，  
    若查找成功，则返回指向该数据元素结点的指针，否则返回空指针*/
```

```
    if( (!T)||key==T->data. Key ) return(T); //查找结束
```

```
    else if (key<T->data. key)
```

```
        return SearchBST(T->lchild, key); //在左子树中继续查找
```

```
    else
```

```
        return SearchBST(T->rchild, key); //在右子树中继续查找
```

```
}
```

## 新的查找算法(能在查找不成功时返回插入位置):

**Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree &p)**

{//在根指针T所指二叉排序树中递归地查找其关键字等于key的数据元素，

若查找成功，则指针p指向该数据元素结点，并返回TRUE，

否则指针p指向查找路径上访问的最后一个结点并返回FALSE，指针f指向T的双亲，  
其初始调用值为NULL

**if (!T) { p=f; return FALSE }; //查找不成功**

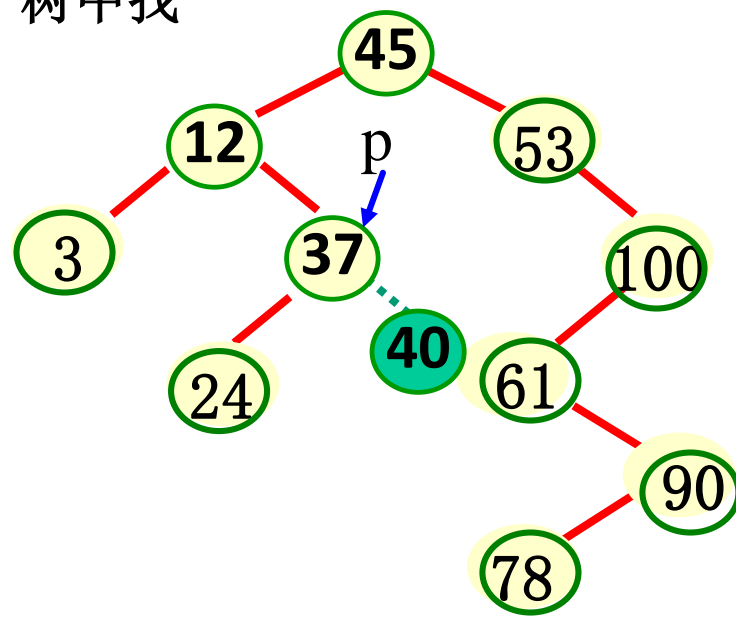
**else if (key==T->data.key){ p=T; return TRUE; } //查找成功**

**else if (key<T->data.key)**

**searchBST(T->lchild, key, T, p); //在左子树中继续查找**

**else searchBST(T->rchild, key, T, p); //右子树中找**

**}**



查找40



# 插入算法

```
Status insertBST(BiTree &T, ElemType e){
```

```
//当二叉排序树中不存在关键字等于e.key的数据元素时，插入e并返回TRUE，  
    否则返回FALSE
```

```
if (!searchBST(T, e.key, NULL, p)) {
```

```
    s=(BiTree)malloc(sizeof(BiTNode)); //建立新结点
```

```
    s->data=e;  s->lchild=s->rchild=NULL;
```

```
    if (!p) T=s;  //被插结点*s为新的根结点
```

```
    else if (e.key<p->data.key) p->lchild=s; //被插结点*s为左孩子
```

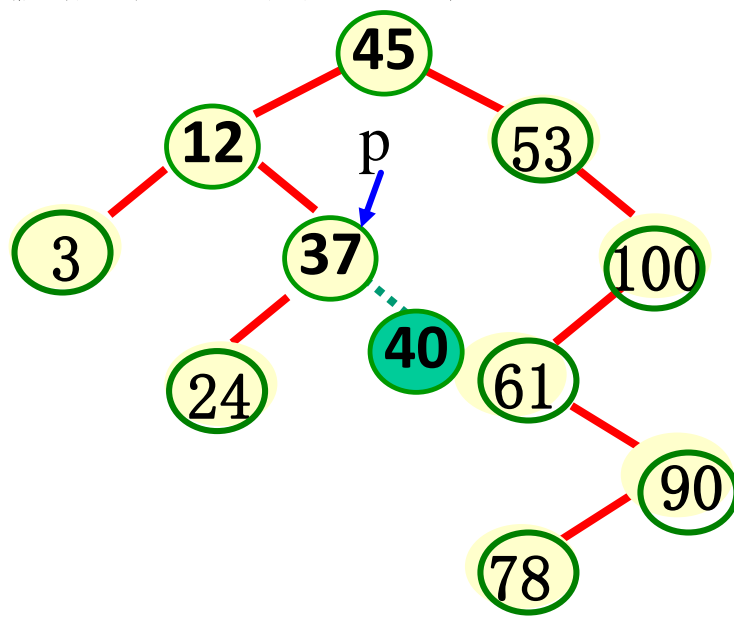
```
    else p->rchild=s; //被插结点*s为右孩子
```

```
    return TRUE;
```

```
}
```

```
else return FALSE;
```

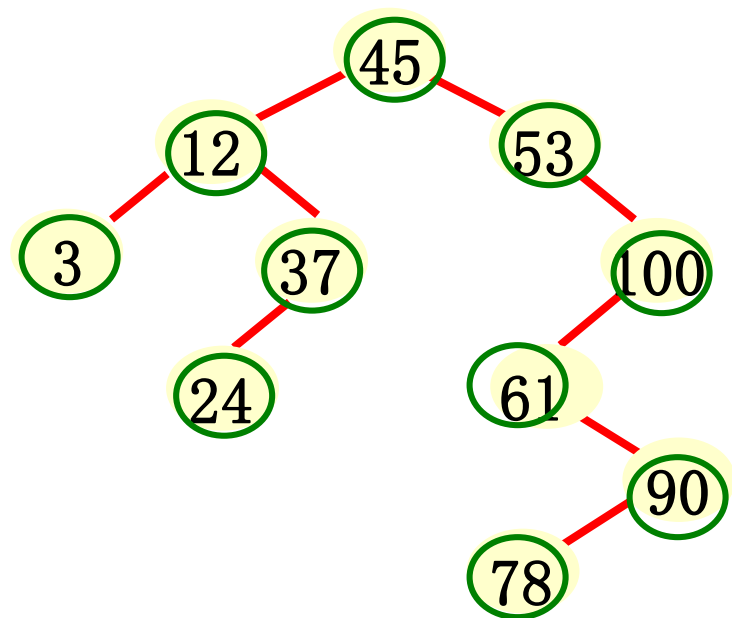
```
}
```



## 2) 二叉排序树的删除

在二叉排序树中删除结点的原则是：**删除结点后仍是二叉排序树。**

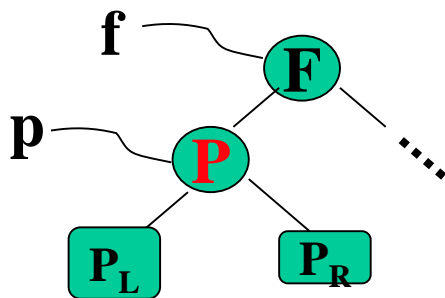
在二叉排序树中删除一个结点相当于在有序序列删除一个结点，**只要删除结点后仍保持二叉排序树的特性。**



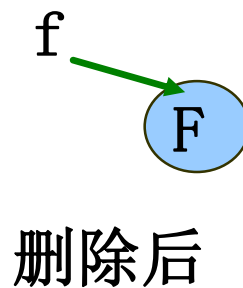
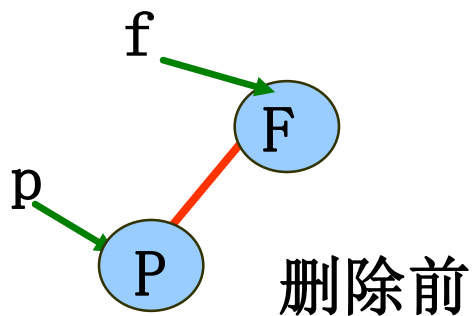
可分**三种情况**讨论：

- (1) 被删除的结点是**叶子结点**；
- (2) 被删除的结点**只有左子树**或者**只有右子树**；
- (3) 被删除的结点**既有左子树，也有右子树**。

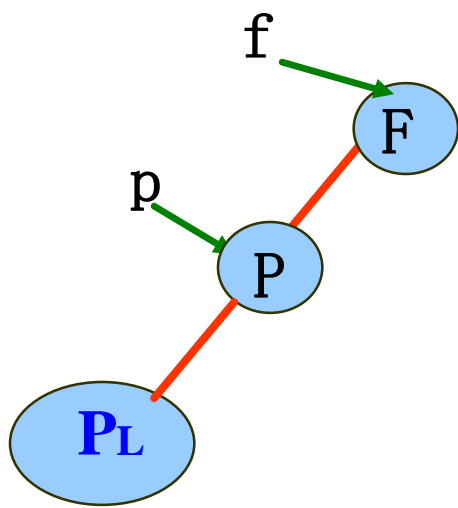
设在二叉排序树被删除结点是P（p是指向被删除结点P的指针），其双亲结点为F（f是指向被删除结点双亲F的指针），不失一般性，设P为F的左孩子，PL和PR分别表示\*p结点的左子树和右子树。分三种情况讨论：



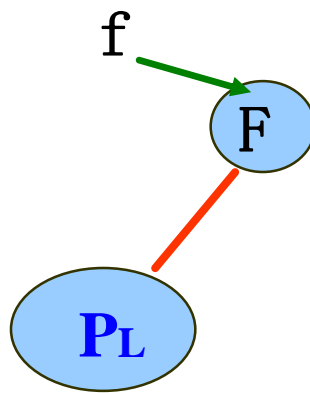
(1) 若P结点为叶子结点，即 $P_L$ 和 $P_R$ 均为空树。由于删去叶子结点不破坏整棵树的结构，则只需修改其双亲结点的指针即可。即将 $f \rightarrow lchild$  赋值为null；



(2) 若P结点只有左子树 $P_L$ 或者只有右子树 $P_R$ ，此时只要令 $P_L$ 或 $P_R$ 直接成为其双亲结点F的左子树即可。



删除前

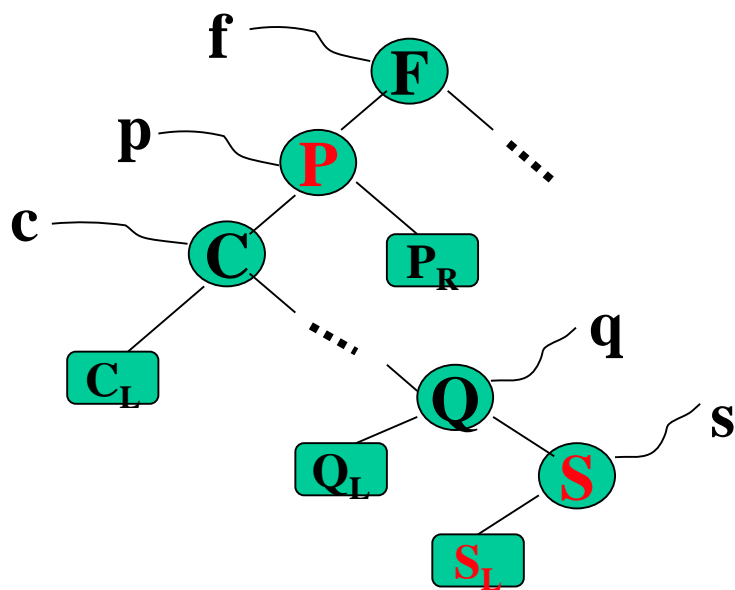


删除后

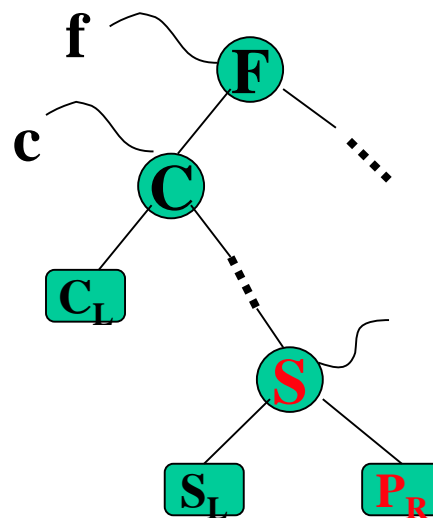
### (3)若P结点的左子树和右子树均不空

如图a，二叉排序树的中序序列为 $(\dots C_L C \dots Q_L Q S_L S P P_R F \dots)$ ，S是P的直接前趋，S是P的左子树的最右下结点，S至少没有右子树。在删除P后，为保持中序序列中其它元素之间的相对位置不变，可有两种作法：

**方法一：**令P的左子树为F的左子树，而P的右子树为S的右子树。结果如图b所示。

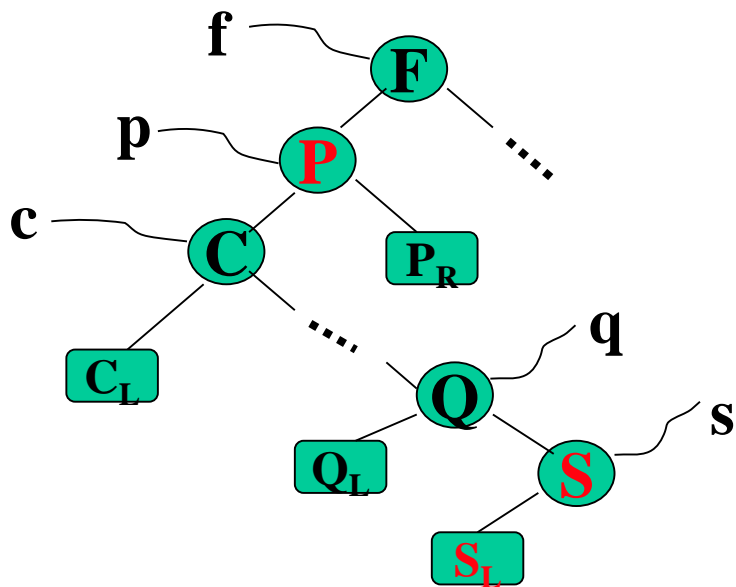


a. 删除P前

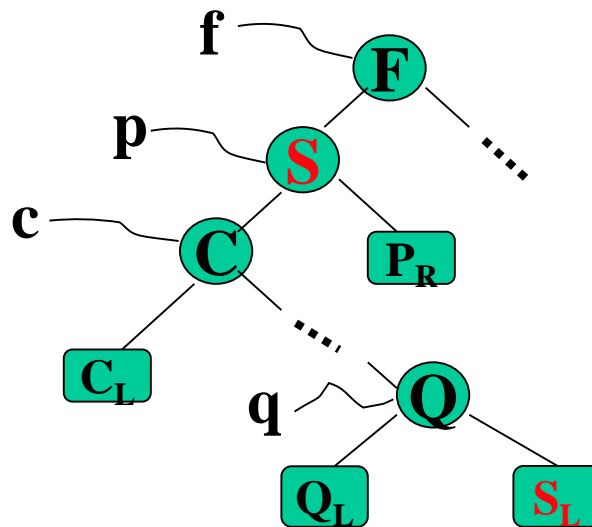


b. 删除P后(方法一)

**方法二：** 用P的直接前趋S代替P，然后从二叉排序树删除P直接前趋S，结果如图c所示。



a. 删除P前

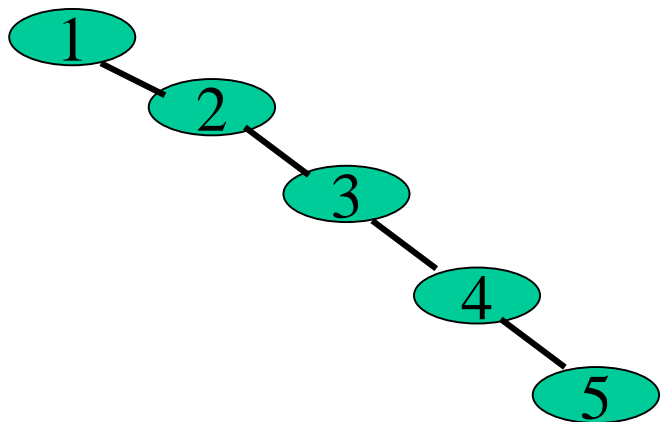


c. 删除P后(方法二)

## 5. 查找性能的分析

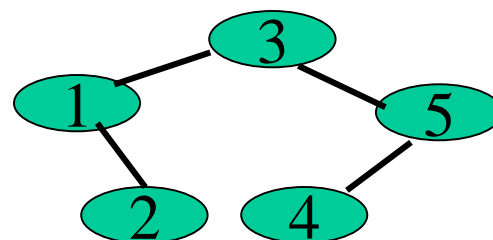
对于每一棵特定的二叉排序树，均可按照平均查找长度的定义来求它的ASL值。例如：

a. 由关键字序列 (1, 2, 3, 4, 5)  
构造而得的二叉排序树：



$$ASL = (1+2+3+4+5) / 5 = 3$$

b. 由关键字序列 (3, 1, 2, 5, 4)  
构造而得的二叉排序树：



$$ASL = (1+2+3+2+3) / 5 = 2.2$$

显然，由值相同的 $n$ 个关键字构造所得的不同形态的二叉排序树的平均查找长度的值不同，甚至可能差别很大。

**最坏情况：**二叉排序树是通过把一个有序表的 $n$ 个结点依次插入而生成的，此时所得的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顺序查找相同。亦是  $(n+1) / 2$ 。

**最好情况：**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  $\log_2 n$ 。

为获得较高的查找效率，需要构造“平衡”的二叉排序树。



## 二、平衡二叉树

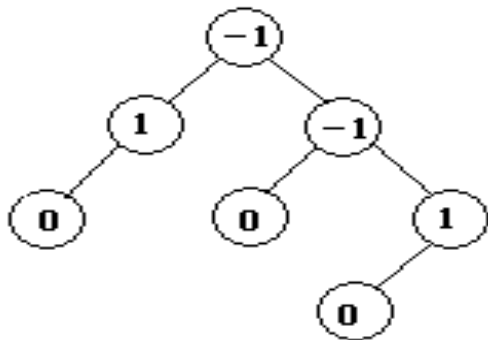
这是另一种形式的二叉查找树，其特点为：

左、右子树深度之差的绝对值不大于1，

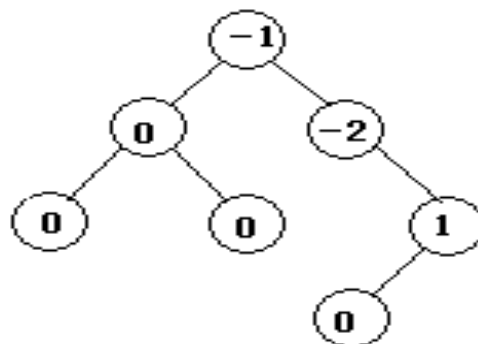
称有这种特性的二叉树为平衡树。

结点的平衡因子BF (Balance Factor)：

该结点的左子树的深度减去它的右子树的深度。



(a) 平衡二叉树；



(b) 不平衡的二叉树

课堂作业:

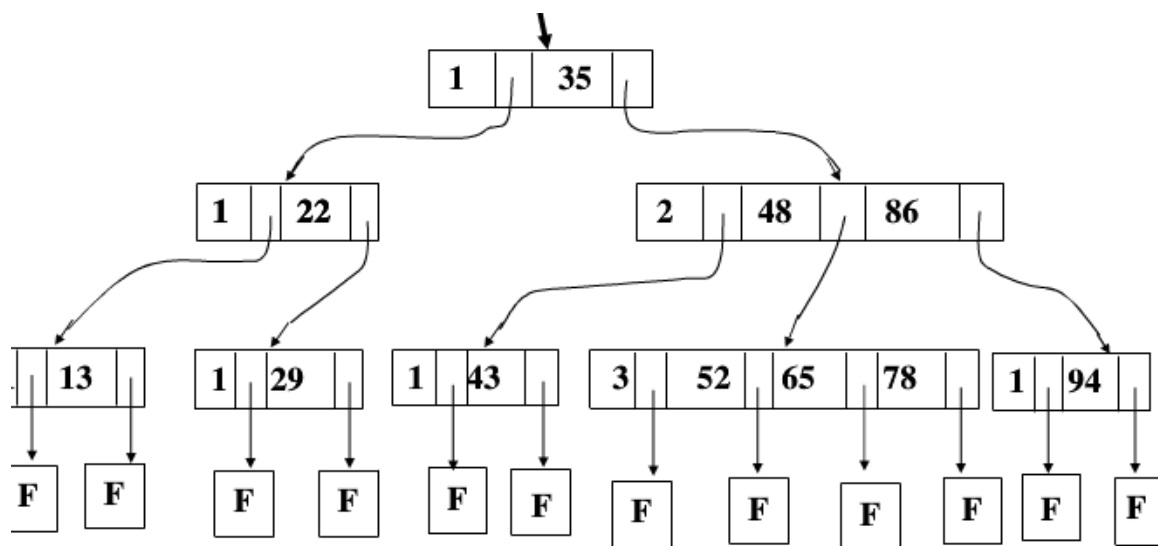
一棵深度为6的平衡二叉树, 其每个非终端结点的平衡因子均为1, 则该树共有 \_\_\_\_ 个结点。

## • B-树

一棵**m**阶的**B-树**，或为空树，或为满足下列特性的**m**叉树：

- 1) 树中每个结点至多有**m**棵子树；
- 2) 若根结点不是叶子结点，则至少有**两**棵子树；
- 3) 除根之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树；
- 4) 所有的非终端结点中包含下列信息数据

$(n, A_0, K_1, A_1, K_2, A_2, \dots, K_n, A_n)$



一棵4阶的B-树

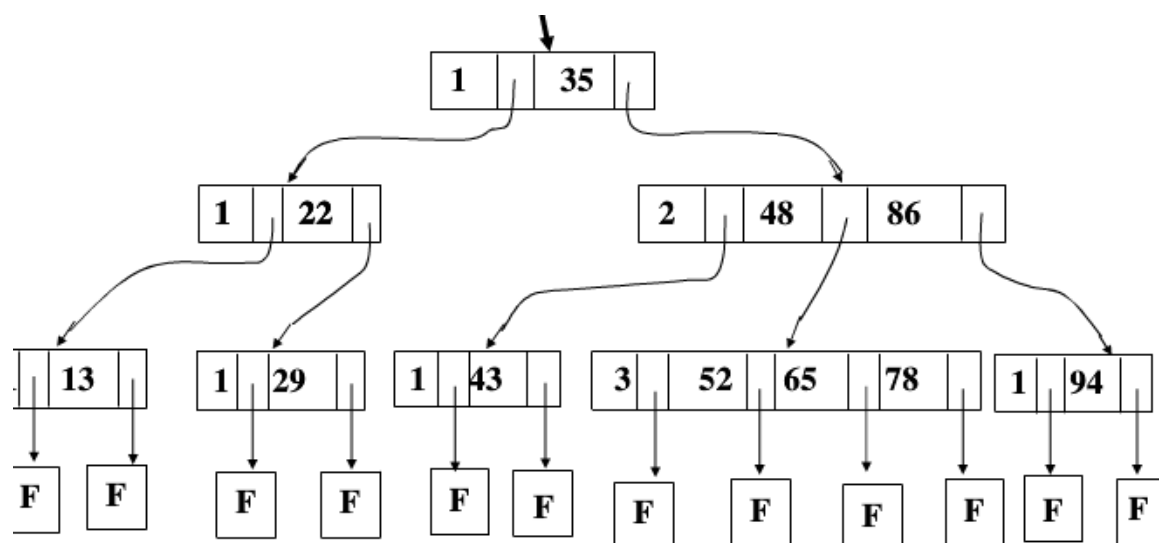
4) 所有的非终端结点中包含下列信息数据

$(n, A_0, K_1, A_1, K_2, A_2, \dots, K_n, A_n)$

其中:  $K_i (i=1, \dots, n)$  为关键字, 且  $K_i < K_{i+1} (i=1, \dots, n-1)$ ;

$A_i (i=0, \dots, n)$  为指向子树根结点的指针, 且指针  $A_{i-1}$  所指子树中所有结点的关键字均小于  $K_i (i=1, \dots, n)$ ,  $A_n$  所指子树中所有结点的关键字均大于  $K_n$ ,  $n (\lceil m/2 \rceil - 1 \leq n \leq m-1)$  为关键字的个数 (或  $n+1$  为子树个数)。

5) 所有的叶子结点都出现在同一层次上, 并且不带信息, 仅表示查找失败。



一棵4阶的B-树

# 静态查找表三种组织与查找方法

顺序表

有序表

索引顺序表

顺序查找

二分查找

分块查找

# 顺序查找

例

0	1	2	3	4	5	6	7	8	9	10	11
50	5	43	19	21	37	56	64	75	80	88	22

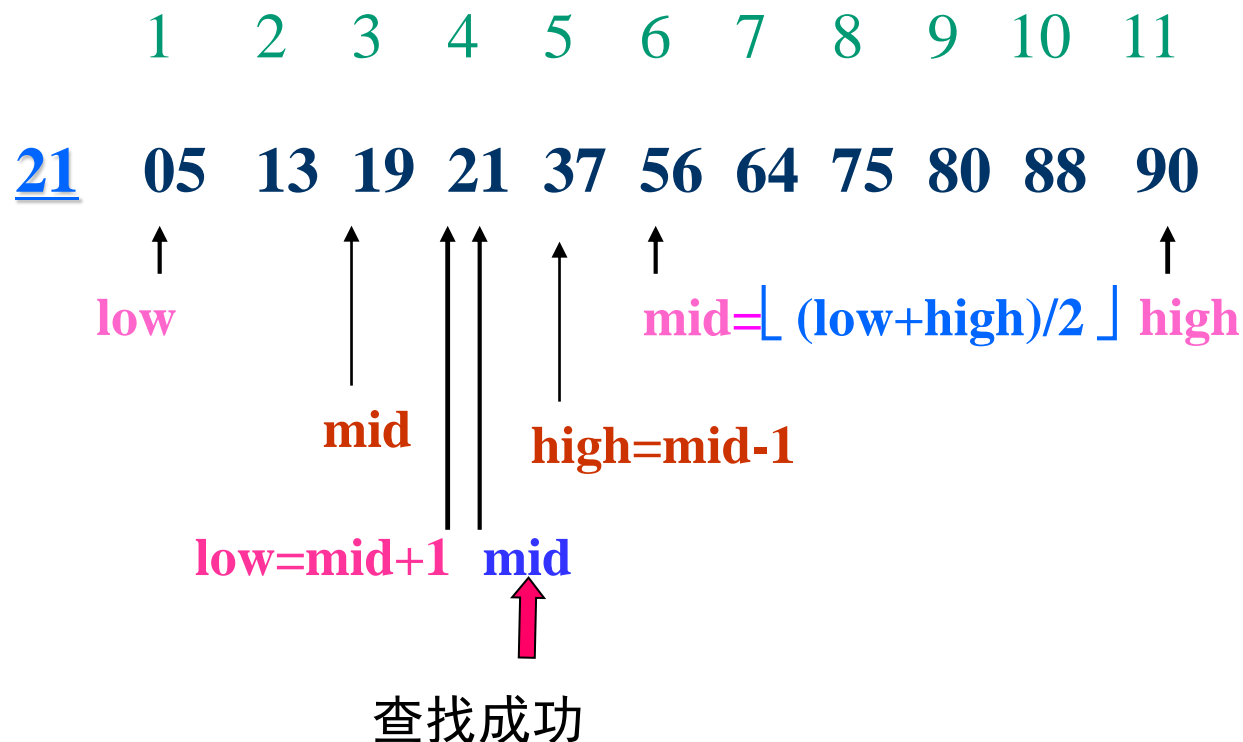
找50

监视哨

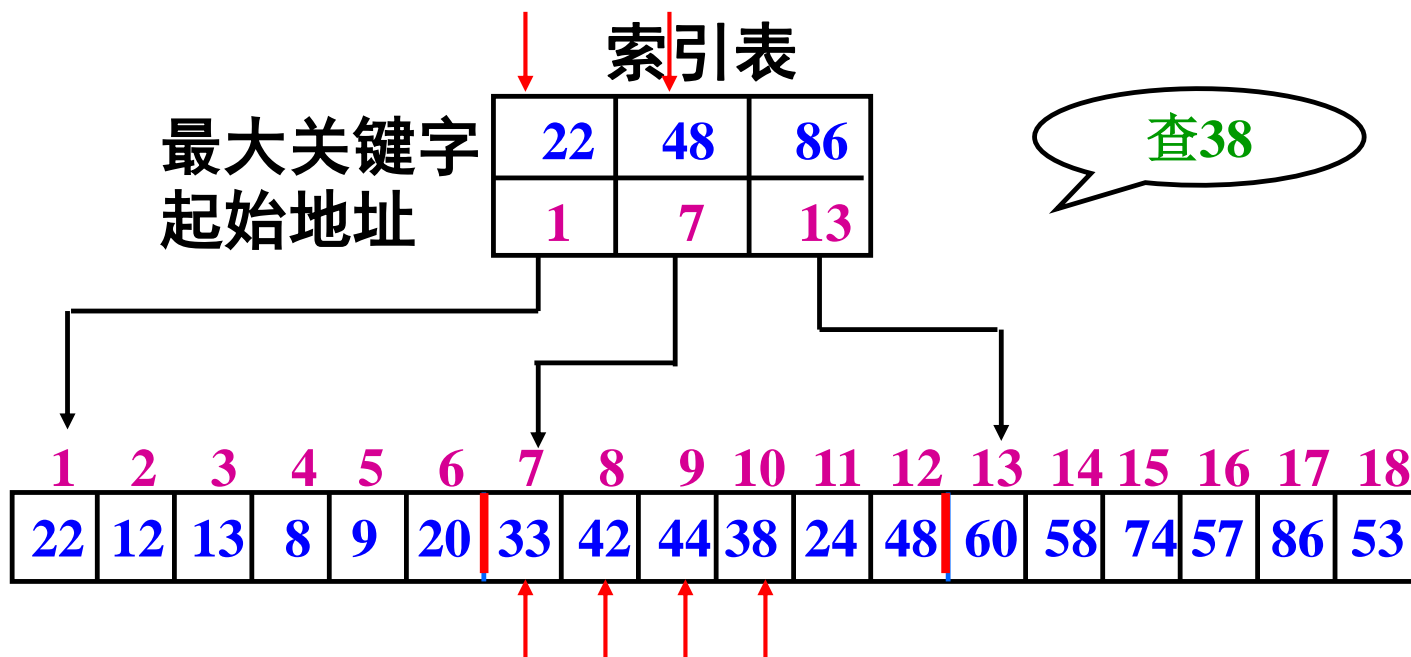
## 二分查找

L2=(05 13 19 21 37 56 64 75 80 88 90),

查找 Key=21的记录



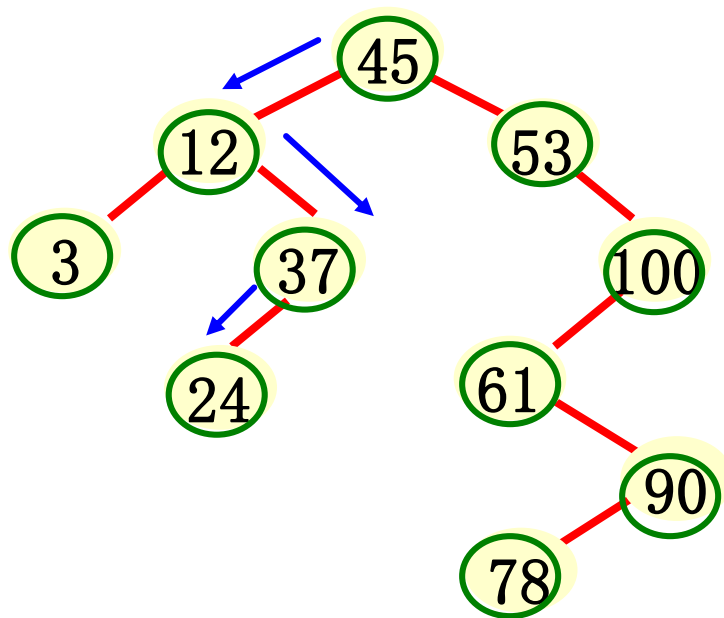
# 分块查找





## 动态查找表 二叉排序树的查找

例 在如下二叉排序树中查找关键字为24的记录



## 9.3

## 哈希表

上两节介绍的查找表的查找方法共同特点：

记录在表中的位置和它的关键字之间不存在一个确定的关系。

查找的过程：给定值依次和关键字集合中各个关键字进行比较。

查找的效率：取决于和给定值进行比较的关键字个数。

查找表不同表示方法的差别仅在于：

和给定值进行比较的关键字的顺序不同。

理想的方法：

不需要比较，根据给定值能直接定位记录的存储位置。

显然，要想根据给定值能直接定位记录的存储位置。需要在记录的存储位置与该记录的关键字之间建立一种确定的对应关系，使每个记录的关键字与一个存储位置相对应。

例     1949-2000年某地区人口统计表

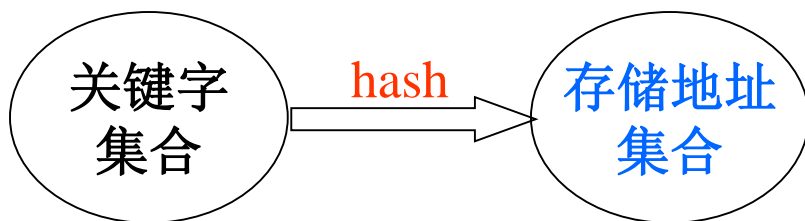
	1	2	3		51	52
年份	1949	1950	1951		1999	2000
人数	2000	2100	2200		4400	4420

各年度人口统计记录的存储位置= 年份-1948

### 9.3.1 什么是哈希表：

◆ **哈希函数**：用来定义记录的关键字与记录存储位置的对应关系的函数，其自变量是记录的关键字，函数值是记录存储位置。

◆ **哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象**



◆ **哈希地址**：由哈希函数求出的记录存储位置称为哈希地址。

◆ **哈希表**：也叫散列表，是基于哈希函数建立的一种查找表。

例 1949-2000年某地区人口统计表

关键字：年份

哈希函数： $H(\text{KEY}) = \text{KEY} - 1948$

哈希表

	1	2	3		51	52
年份	1949	1950	1951		1999	2000
人数	2000	2100	2200		4400	4420

哈希函数只是一种映象，所以哈希函数的设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。由于哈希函数是一个压缩映象，因此，在一般情况下，很容易产生“冲突”现象。

**冲突：**  $key1 \neq key2$ ，但  $H(key1) = H(key2)$  的现象。

**同义词：** 具有相同函数值的两个关键字，叫该哈希函数的~

例 某年全国各城市人口统计表

关键字：城市名

哈希函数：  $H(key) = key$  的第一个字母在字母表中的序号

同义词							
key	北京	天津	石家庄	保定	吉林	哈尔滨	长春
H(key)	2	20	19	2	10	8	3

构造哈希表要解决两方面的问题：

- 1) 构造“好”的哈希函数
- 2) 选择一种处理冲突的方法。

### 9.3.2 哈希函数的构造方法

“好”的哈希函数的条件：

简单：易于计算。

均匀：减少冲突。

若对于关键字集合中的任一个关键字，经哈希函数映射到地址集合中任何一个地址的概率是相等的，则称此类哈希函数为均匀的。

## 1. 直接定址法

哈希函数为关键字或关键字的线性函数

$$H(\text{key}) = \text{key} \quad \text{或者} \quad H(\text{key}) = a * \text{key} + b$$

其中，a和b为常数。

特点：

▲地址集合的大小 = 关键字集合的大小，不会发生冲突。

▲实际中能用这种哈希函数的情况很少。

例如：以学生学号为关键字的成绩表，1号学生的记录位置在第一条，10号学生的记录位置在第10条...



## 2. 数字分析法

假设关键字集合中的**每个关键字都是由n位数字组成** ( $k_1, k_2, \dots, k_n$ ), 分析关键字集中的全体, 并从中提取**分布均匀的若干位**或它们的组合作为地址。

有学生的生日数据如下:

年 . 月 . 日  
98 . 10 . 03  
98 . 11 . 23  
99 . 03 . 02  
99 . 07 . 12  
97 . 08 . 21  
99 . 02 . 15  
...

**分析:** 第一位, 第二位, 第三位重复的可能性大, 取这三位造成冲突的机会增加;  
**结论:** 尽量不取前三位, 取后三位比较好。

### 3. 平方取中法

取关键字平方后中间几位作哈希地址。

若关键字的每一位都有某些数字重复出现频度很高的现象，则先求关键字的平方值，以通过“平方”扩大差别，同时平方值的中间几位受到整个关键字中各位的影响；

记录	关键字	(关键字) <sup>2</sup>	哈希地址
A	0100	00 <u>10</u> 000	010
I	1100	12 <u>10</u> 000	210
J	1200	14 <u>40</u> 000	440
I0	1160	13 <u>70</u> 400	370
P1	2061	43 <u>10</u> 541	310
P2	2062	43 <u>14</u> 704	314
Q1	2161	47 <u>34</u> 741	734
Q2	2162	47 <u>41</u> 304	741

## 4. 折叠法

若关键字的位数特别多，则可将其分割成几部分，然后取它们的叠加和为哈希地址。可有：移位叠加和间界叠加两种处理方法。

## 5. 除留余数法

取关键字被某个不大于哈希表表长 $m$ 的数 $p$ 除后所得余数为哈希地址。

$$H(\text{key}) = \text{key} \text{ MOD } p \quad p \leq m$$

关键问题是: 如何选取  $p$  ?

可以选 $P$ 为质数或不包含小于20的质因数的合数。

## 6. 随机数法

选择一个随机函数，取关键字的随机函数值为它的哈希地址，即

$$H(\text{key}) = \text{Random}(\text{key})$$

实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是使产生冲突的可能性降到尽可能地小。

### 9.3.3 解决冲突的方法：

当不同记录由哈希函数确定的存储位置相同，发生冲突，这时需用某种方法解决冲突。

常用的处理冲突的方法有下列几种：

1. 开放定址法
2. 链地址法

## 1. 开放定址法

$$H_i = ( H(\text{key}) + d_i ) \text{ MOD } m \quad i=1, 2, \dots, k \ (k \leq m-1)$$

其中：  $H(\text{key})$ ——哈希函数

$m$ ——哈希表表长

$d_i$ ——增量序列

◆分类：

●线性探测再散列：  $d_i = 1, 2, 3, \dots, m-1$

●二次探测再散列：  $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, \pm k^2 (k \leq m/2)$

●伪随机探测再散列：  $d_i = \text{伪随机数序列}$

**例：** 设哈希表HT表长为11，哈希函数  $H(\text{key}) = \text{key} \bmod 11$ ，  
试用开放定址法中线性探测再散列解决冲突

$$H_i(\text{key}) = (H(\text{KEY}) + d_i) \bmod 11 \quad (d_1=1, d_2=2, d_3=3, \dots)$$

试对下列关键字序列 (19, 13, 33, 02, 16, 29, 24) 构造  
哈希表HT。

$$H(19) = 19 \bmod 11 = 8 \quad H(13) = 13 \bmod 11 = 2 \quad H(33) = 33 \bmod 11 = 0$$

$$H(02) = 02 \bmod 11 = 2 \quad \text{冲突} \quad H_1(02) = (H(02) + d_1) \bmod 11 = (2+1) \bmod 11 = 3$$

$$H(16) = 16 \bmod 11 = 5 \quad H(29) = 29 \bmod 11 = 7$$

$$H(24) = 24 \bmod 11 = 2 \quad \text{冲突}$$

$$H_1(24) = (H(24) + d_1) \bmod 11 = (2+1) \bmod 11 = 3 \quad \text{冲突}$$

$$H_2(24) = (H(24) + d_2) \bmod 11 = (2+2) \bmod 11 = 4$$

0	1	2	3	4	5	6	7	8	9	10
33		13	02	24	16		29	19		

哈希表建好了！

## 1. 开放定址法

$$H_i = ( H(\text{key}) + d_i ) \text{ MOD } m \quad i=1, 2, \dots, k \ (k \leq m-1)$$

其中：  $H(\text{key})$ ——哈希函数

$m$ ——哈希表表长

$d_i$ ——增量序列

◆分类：

●线性探测再散列：  $d_i = 1, 2, 3, \dots, m-1$

●二次探测再散列：  $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, \pm k^2 (k \leq m/2)$

●伪随机探测再散列：  $d_i = \text{伪随机数序列}$



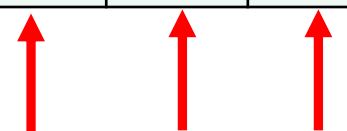
例如：哈希表的长度为11，哈希函数为 $H(\text{key}) = \text{key} \% 11$ ，假设表中已填有关键字分别为17、60、29的记录，如图所示。现有第四个记录，其关键字为38，由哈希函数得到哈希地址为5，产生冲突。

0	1	2	3	4	5	6	7	8	9	10
					60	17	29			

a. 插入前

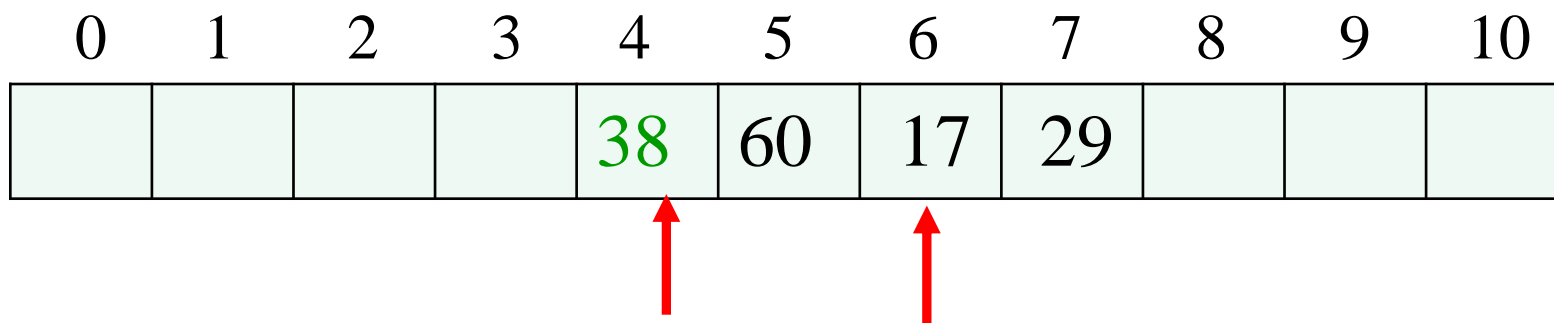
若用线性探测再散列处理：

0	1	2	3	4	5	6	7	8	9	10
					60	17	29	38		



若用二次探测再散列处理:

0	1	2	3	4	5	6	7	8	9	10
				38	60	17	29			



The diagram shows a hash table with 11 slots, indexed from 0 to 10. The values stored are: slot 4: 38, slot 5: 60, slot 6: 17, slot 7: 29. Red arrows point to slots 4 and 6, indicating a secondary probe for collision resolution.

若用伪随机探测再散列处理:

假设产生的伪随机数为9, 则计算下一个哈希地址为  $(5+9)\%11=3$ , 所以38填入序号为3的位置。

0	1	2	3	4	5	6	7	8	9	10
			38		60	17	29			

课堂练习题：

设有一组关键字(19, 01, 25, 12, 24, 38, 84, 27)，采用哈希函数： $H(\text{key}) = \text{key} \bmod 13$ ，采用开放定址法的二次探测再散列方法解决冲突，试在0-16的散列地址空间中对该关键字序列构造哈希表。

课堂练习题：

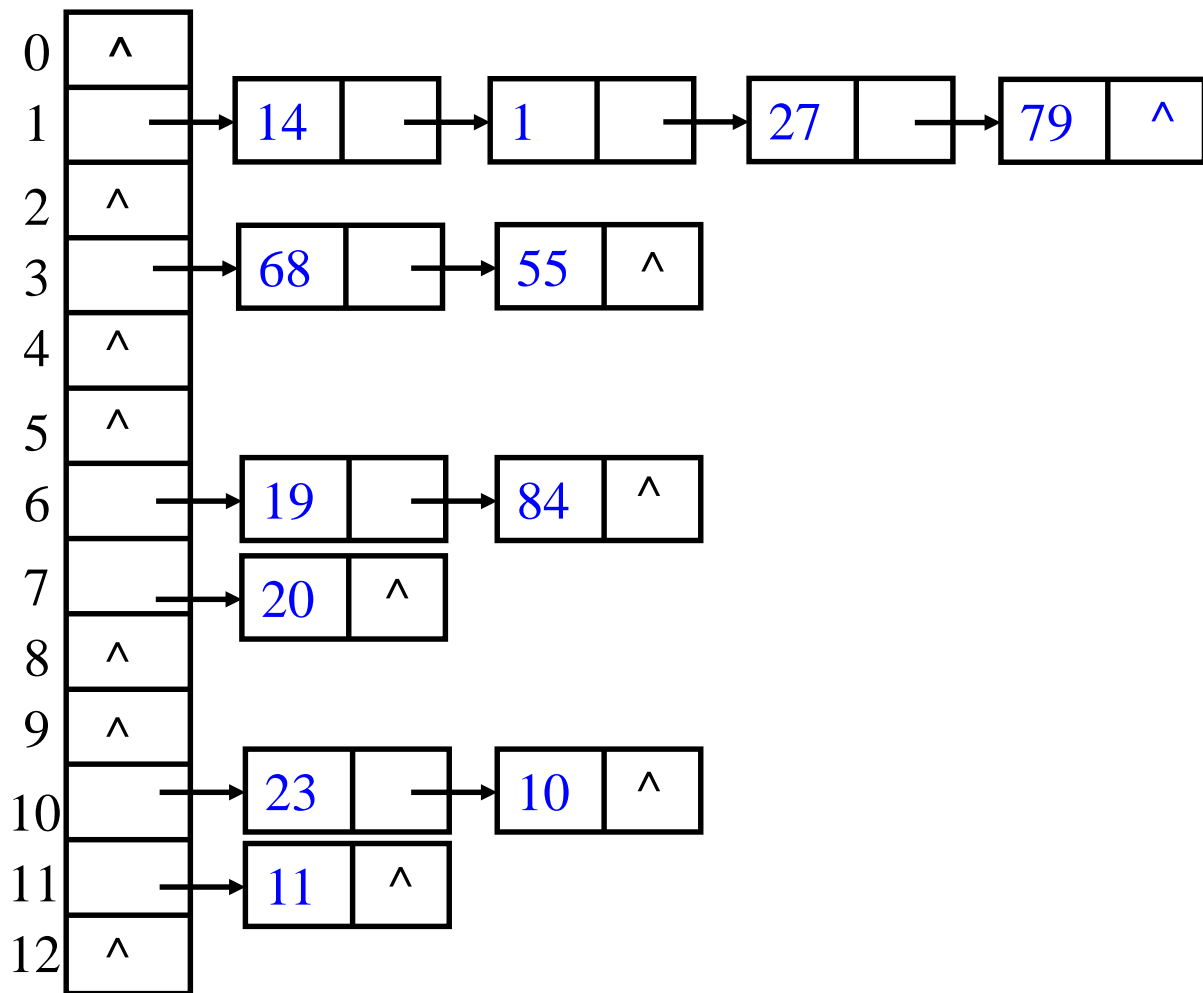
设有一组关键字(19, 01, 25, 12, 24, 38, 84, 27)，采用哈希函数： $H(\text{key}) = \text{key} \bmod 13$ ，采用开放定址法的二次探测再散列方法解决冲突，试在0-16的散列地址空间中对该关键字序列构造哈希表。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

## 2、链地址法

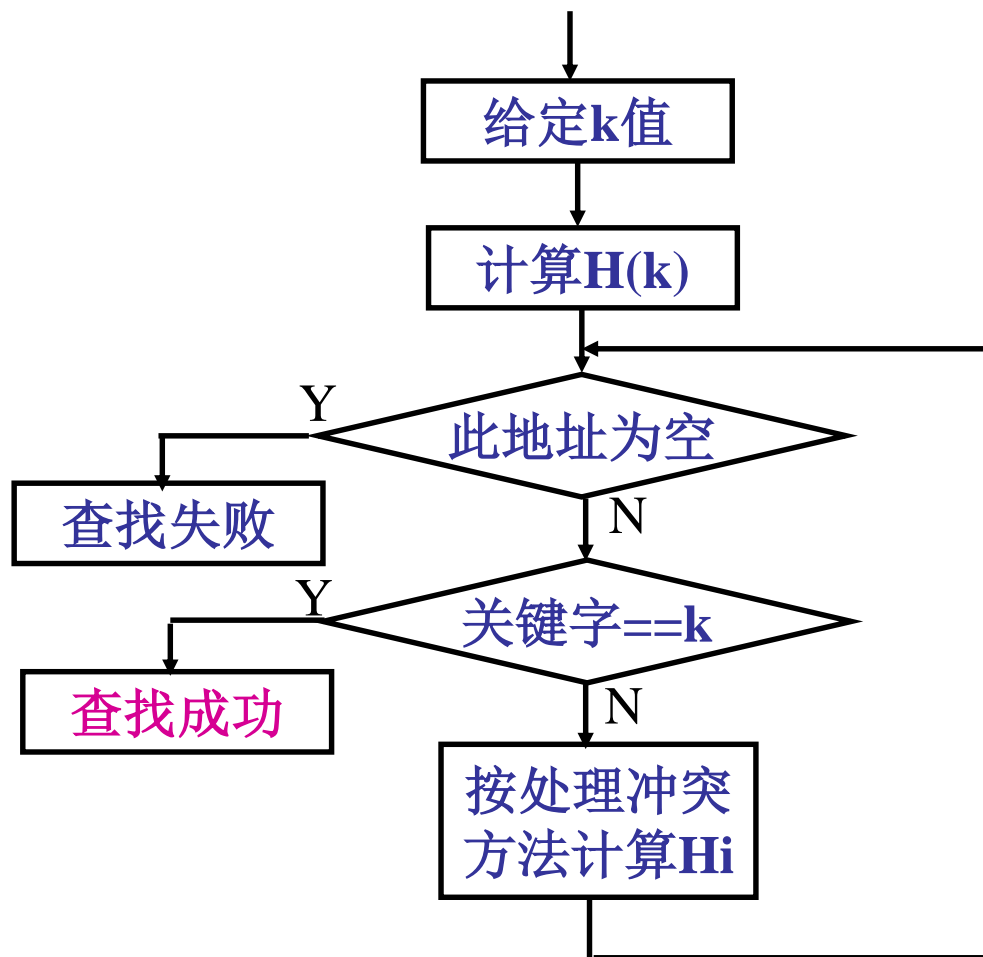
**链地址法的基本思想是：**将所有哈希地址相同的记录都链接在同一单链表中。有 $m$ 个哈希地址就有 $m$ 个单链表，同时用数组 $HT[0 \cdots m-1]$ 存放各个链表的头指针，凡是哈希地址为 $i$ 的记录都以结点方式插入到以 $HT[i]$ 为头结点的单链表中。

例 已知一组关键字 (19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)  
哈希函数为:  $H(\text{key}) = \text{key} \text{ MOD } 13$ ,  
用链地址法处理冲突



### 9.3.4 哈希表的查找

在哈希表上进行查找的过程和哈希造表的过程基本一致。



例：给出哈希表HT，哈希函数  $H(\text{key}) = \text{key} \bmod 11$ ，解决冲突方法：  
：开放地址法中线性探测再散列  $H_i(\text{key}) = (H(\text{KEY}) + d_i) \bmod 11$   
( $d_1=1, d_2=2, d_3=3, \dots$ )，试查找关键字19、02。

### 查找关键字19

0	1	2	3	4	5	6	7	8	9	10
33		13	02	24	16		29	19		

用哈希函数求19 对应的哈希地址：  
 $H(19) = 19 \bmod 11 = 8$

将HT[8]与19比较  
相等，查找成功



# 查找关键字02

用哈希函数求02 对应的哈希地址：  
 $H(02)=02 \bmod 11=2$

用解决冲突方法为02求下一个“地址”  
 $H_1(02)=(H(02)+d_1) \bmod 11=3$

0	1	2	3	4	5	6	7	8	9	10
33		13	02	24	16		29	19		

将HT[2]与02比较不相等

将HT[3]与02比较相等, 查找成功

```
Status SearchHash (HashTable H, KeyType K, int &p, int &c) {  
/*在开放定址哈希表H中查找关键码为K的元素， 若查找成功，以 p指示待查  
数据元素在表中位置， 并返回SUCCESS； 否则，以p 指示插入位置， 并返回  
UNSUCCESS, c用以计冲突次数，其初值置零，供建表插入时参考*/  
    p = Hash(K); // 求得哈希地址  
    while ( H.elem[p].key != NULLKEY && K != H.elem[p].key)  
        // 该位置中填有记录并且关键字不相等  
        collision(p, ++c); // 求得下一探查地址p  
    if (K== H.elem[p].key) return SUCCESS;  
        // 查找成功， p返回待查数据元素位置  
    else return UNSUCCESS;  
        // 查找不成功， p返回的是插入位置  
}
```

```
Status InsertHash (HashTable &H, Elemtype e){  
    c = 0;  
    if ( SearchHash ( H, e.key, p, c ) == SUCCESS )  
        return DUPLICATE; // 表中已有与 e 有相同关键字的元素  
    else if ( c < hashsize[H.sizeindex]/2 ) {  
        // 冲突次数 c 未达到上限, (阈值 c 可调)  
        H.elem[p]=e; ++H.count; return OK;  
    }  
    else { RecreateHashTable(H);  
        return UNSUCCESS; } // 重建哈希表  
}
```

## 哈希表的查找的性能分析:

哈希表的**装填因子**定义为  $\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$

$\alpha$  标志着哈希表的**装满程度**，直观地看， $\alpha$  越小，发生冲突的可能性就越小；反之， $\alpha$  越大，发生冲突的可能性就越大。

**线性探测再散列**的哈希表查找成功时的平均查找长度为

$$S_{nl} \approx \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)$$

**二次探测再散列**的哈希表查找成功时的平均查找长度为

$$S_{nr} \approx -\frac{1}{\alpha} \ln(1 - \alpha)$$

**链地址法**处理冲突的哈希表查找成功时的平均查找长度为

$$S_{nc} \approx 1 + \frac{\alpha}{2}$$

## 一、基本内容

静态查找表：顺序查找、二分查找和分块查找，

动态查找表：二叉排序树和平衡二叉树，

哈希查找、哈希函数的构造方法和处理冲突的方法。

## 二、学习要点

1. 了解顺序查找、二分查找和分块查找的概念，二叉排序树和平衡二叉树、哈希查找等的概念。

2、理解顺序查找、二分查找和分块查找算法，二叉排序树的性质。

3. 掌握哈希函数的构造方法和处理冲突的方法，。

4、会用哈希函数、开放地址法或链地址法建立散列表。

# 第九章 习 题一

p55

1、 9.2

2、 9.9 (1). (2)

设有一组关键字(19, 01, 23, 14, 55, 20, 84, 27, 68, 11, 10, 77), 采用哈希函数:  $H(key) = key \text{ Mod } 13$ , 采用开放定址法的二次探测再散列方法解决冲突, 试在0-18的散列地址空间中对该关键字序列构造哈希表。

[illegible]

1、给定11个数据元素的有序表(2, 3, 10, 15, 20, 25, 28, 29, 30, 35, 40)，采用折半查找，试问：

1) 若查找给定值为20的元素，将依此与表中哪些元素比较？

2) 假设查找表中每个元素的概率相同，求查找成功时的平均查找长度。



2.已知一表为（25, 18, 46, 2, 53, 39, 32, 4, 74, 67, 60, 11）,按表中顺序依次插入初始为空的二叉排序树，要求：

(1)画出建立的二叉排序树。

(2)求出在等概率情况下查找成功的平均查找长度。

3.一棵深度为6的平衡二叉树，其每个非终端结点的平衡因子均为1,则该树共有 \_\_\_\_个结点。