

# 第九章

## 查 找

## ※ 教学内容:

静态查找表(顺序表,有序表,索引顺序表); 动态查找表(二叉排序树,平衡二叉树,B-树)的建立和查找; 哈希表的建立, 查找及分析

## ※ 教学重点:

顺序查找, 折半查找和索引查找的方法; 二叉排序树的构造方法; 二叉平衡树的旋转平衡方法; B-树的建立过程; 哈希表的构造方法; 计算各种查找方法在等概率情况下查找成功时和失败时的平均查找长度

## ※ 教学难点:

二叉平衡树的旋转平衡方法

# 何谓查找表？

查找表是由同一类型的数据元素(或记录)构成的集合。

由于“集合”中的数据元素之间存在着松散的关系，因此查找表是一种应用灵便的结构。

# 对查找表经常进行的操作：

- 1) 查询某个“特定的”数据元素是否在查找表中；
- 2) 检索某个“特定的”数据元素的各种属性；
- 3) 在查找表中插入一个数据元素；
- 4) 从查找表中删去某个数据元素。

# 查找表可分为两类:

## ● 静态查找表

仅作**查询**和**检索**操作的查找表为静态查找表。

## ● 动态查找表

有时在查询之后,还需要将“查询”结果为“**不在查找表中**”的数据元素**插入到**查找表中;或者,从查找表中**删除**其“查询”结果为“**在查找表中**”的数据元素,此类表为动态查找表。

# 几个概念:

## ● 关键字

用来标识一个数据元素（或记录）的某个数据项的值，称为关键字。

## ● 主关键字

若此关键字可唯一地标识一个记录，则称此关键字是主关键字；

## ● 次关键字

反之，用以识别若干记录的关键字是次关键字。

## ● 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数据元素或(记录)。

若查找表中存在这样一个记录，则称“**查找成功**”。查找结果给出整个记录的信息，或指示该记录在查找表中的位置；

否则称“**查找不成功**”。查找结果给出“空记录”或“空指针”。

# 如何进行查找？

查找的方法取决于查找表的结构,即表中数据元素是依何种关系组织在一起的。

由于查找表中的数据元素之间不存在明显的组织规律, 因此不便于查找。

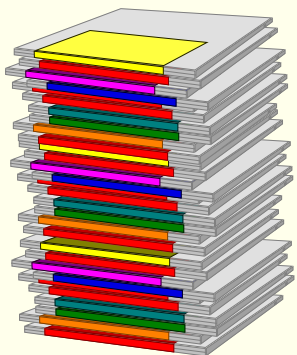
为了提高查找的效率, 需要在查找表中的元素之间人为地附加某种确定的关系, 换句话说, 用另外一种结构来表示查找表。



## 9.1 静态查找表

## 9.2 动态查找树表

## 9.3 哈希表



# 9.1

## 静态查找表

# 抽象数据类型静态查找表的定义:

## ADT StaticSearchTable {

**数据对象D:** D是具有相同特性的数据元素的集合。每个数据元素含有类型相同的**关键字**，可唯一标识数据元素。

**数据关系R:** 数据元素同属一个集合。

**基本操作P:** Create(&ST, n);  
Destroy(&ST);  
Search(ST, key);  
Traverse(ST, Visit());

} ADT StaticSearchTable

**Create(&ST, n);**

操作结果：构造一个含n个数据元素的静态查找表ST。

**Destroy(&ST);**

初始条件：静态查找表ST存在；

操作结果：销毁表ST。

# Search(ST, key);

**初始条件:** 静态查找表ST存在, key 为和查找表中元素的关键字类型相同的给定值;

**操作结果:** 若 ST 中存在其关键字等于 key 的数据元素, 则函数值为该元素的**值**或在表中的**位置**, 否则为“**空**”。

# Traverse(ST, Visit());

**初始条件：**静态查找表ST存在，Visit是对元素操作的应用函数；

**操作结果：**按某种次序对ST的每个元素调用函数Visit()一次且仅一次，一旦Visit()失败，则操作失败。

假设静态查找表的顺序存储结构为

```
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;
```

数据元素类型的定义为：

```
typedef struct {  
    keyType key;        // 关键字域  
    ... ..             // 其它属性域  
} ElemType ;
```

# 宏定义（对两个关键字的比较约定）

## //对数值型关键字

```
#define EQ(a,b) ((a) == (b))
```

```
#define LT(a,b) ((a) < (b))
```

```
#define LQ(a,b) ((a) <= (b))
```

## //对字符串型关键字

```
#define EQ(a,b) (! strcmp((a) , (b)))
```

```
#define LT(a,b) (strcmp((a) , (b)) < 0)
```

```
#define LQ(a,b) (strcmp((a) , (b)) <= 0)
```



## 一、顺序查找表

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

## 二、有序查找表

$$ASL_{bs} = \frac{1}{n} \sum_{i=1}^n C_i = \frac{1}{n} \left[ \sum_{j=1}^h j \times 2^{j-1} \right] = \frac{n+1}{n} \log_2(n+1) - 1$$

## 三、静态查找树表

## 四、索引顺序表



# 一、顺序查找表

以顺序表或线性链表  
表示静态查找表

# (1) 回顾顺序表的查找过程:

ST.elem

	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

ST.Length

$k$

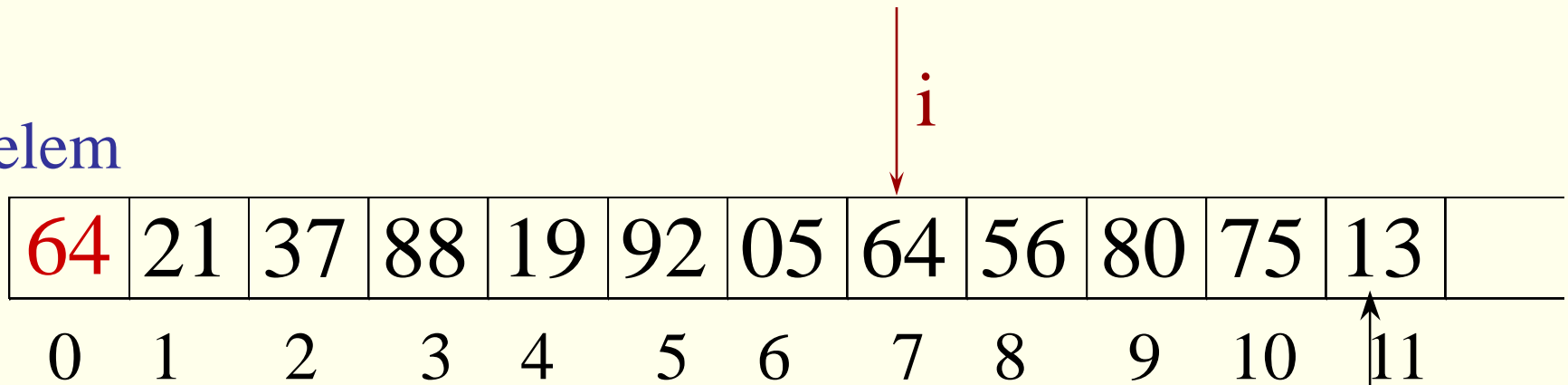
假设给定值  $e=64$ ,

要求  $ST.elem[k] = e$ , 问:  $k = ?$

## (2) 从前往后查的算法

```
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;  
} //location
```

ST.elem

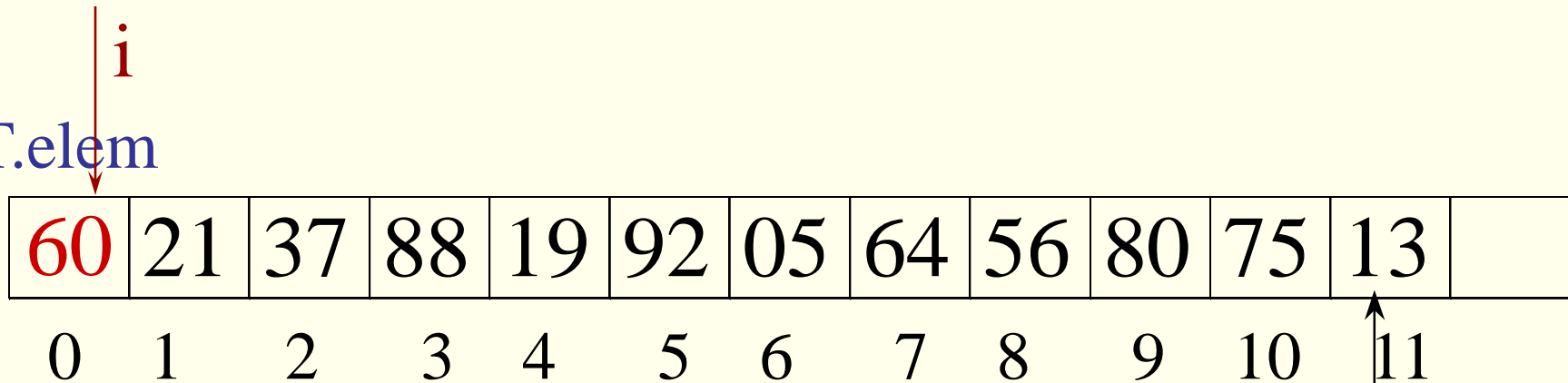


64	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=64

ST.Length

ST.elem



60	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=60

ST.Length

### (3) 从后往前查的算法

```
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”  
    for (i=ST.length; ST.elem[i].key!=key; --i);  
    // 从后往前找  
    return i;                       // 找不到时，i为0  
} // Search_Seq
```

从后往前查的算法中，查找之前，把第0个元素设为查找元素key。这样避免在查找过程中每一步都要检测整个表是否查找完毕。ST.elem[0]起到了监视哨的作用。

上述程序技巧上的改进，在查找表的长度 $>1000$ 时，查找所需的平均时间减少一半。

## (4) 分析顺序查找的时间性能

查找算法的平均查找长度

(Average Search Length)

为确定记录在表中的位置，需和给定值进行**比较**的关键字**次数**的**期望值**称为查找算法在**查找成功**时的**平均查找长度**。

**查找不成功**时的比较次数为 $n+1$ 。



对于含有n个数据元素的表, 查找成功时的平均查找长度为:

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

其中:  $P_i$  为查找第  $i$  个数据元素的概率, 且

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

$C_i$  为查到第  $i$  个数据元素时, 需要进行的比较次数.

对顺序表而言， $C_i$ 取决于所查元素所处的位置：

- 查找记录是 $A[n]$ 时，仅需比较一次；
- 查找记录是 $A[1]$ 时，要比较 $n$ 次；
- 查找记录是 $A[i]$ 时，要比较 $n-i+1$ 次。

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

在等概率查找的情况下， $P_i = \frac{1}{n}$

顺序表查找的平均查找长度为：

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

## 二、有序查找表

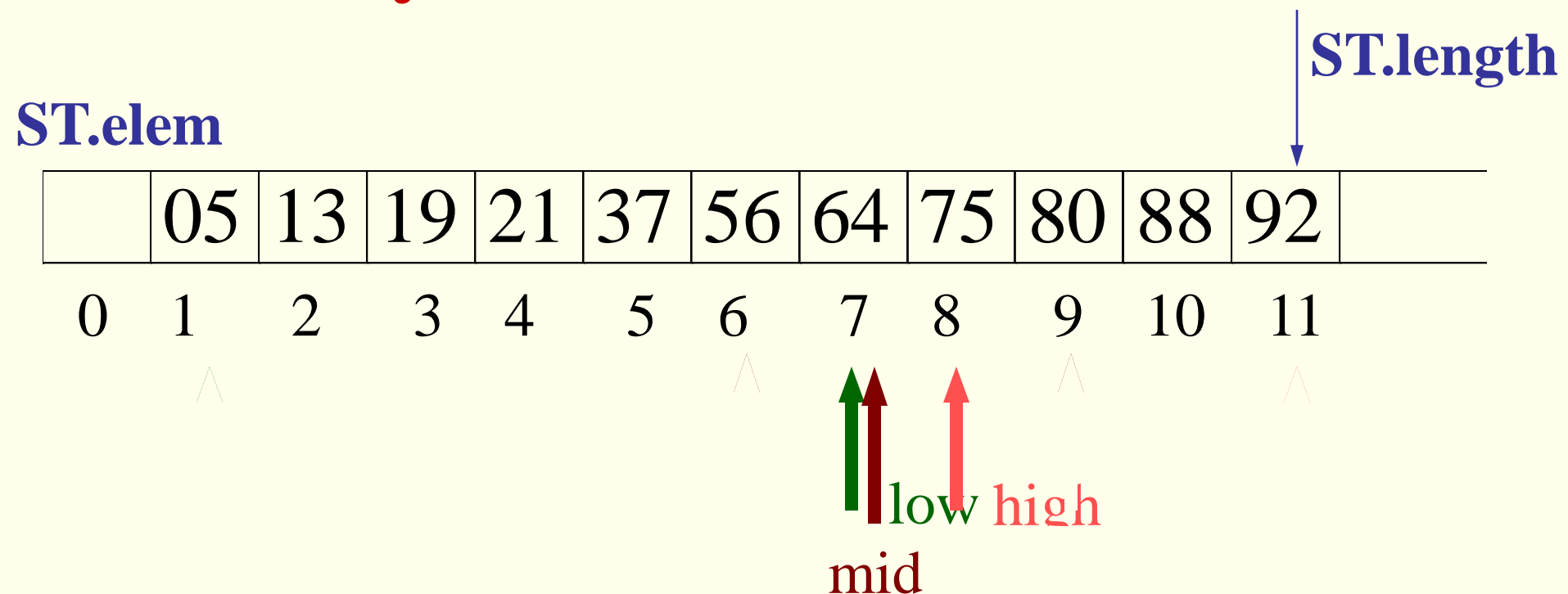
上述顺序查找表的查找算法简单，  
但平均查找长度较大，特别不适用于表  
长较大的查找表。

若以有序表表示静态查找表，则查  
找过程可以基于“折半”进行。

# (1) 折半查找(二分查找)的基本思想

- 如果查找表中的数据元素按关键字有序(假设递增有序),则在查找时不必逐个顺序比较,而采用跳跃的方式,
  - 先与“**中间位置**”的记录关键字值比较,若相等则查找成功;
  - 若给定值大于“中间位置”的关键字值,则在后半部继续进行折半查找;
  - 否则在前半部进行折半查找。
- 折半查找**仅适用于**以**顺序存贮**结构组织的**有序表**的查找。

例如: **key=64** 的查找过程如下:



**low** 指示查找区间的下界

**high** 指示查找区间的上界

**mid** =  $(low+high)/2$  (取整)

## (2) 折半查找算法

- 设待查元素所在区域的下界为low, 上界为hig, 则中间位置 $mid = (low + hig) \text{DIV} 2$ 
  - 若中间位置元素值等于给定值, 则查找成功;
  - 若中间位置元素值大于给定值, 则在区域[low, mid-1]进行折半查找
  - 若中间位置元素值小于给定值, 则在区域[mid+1, hig]内进行折半查找

# 折半查找算法

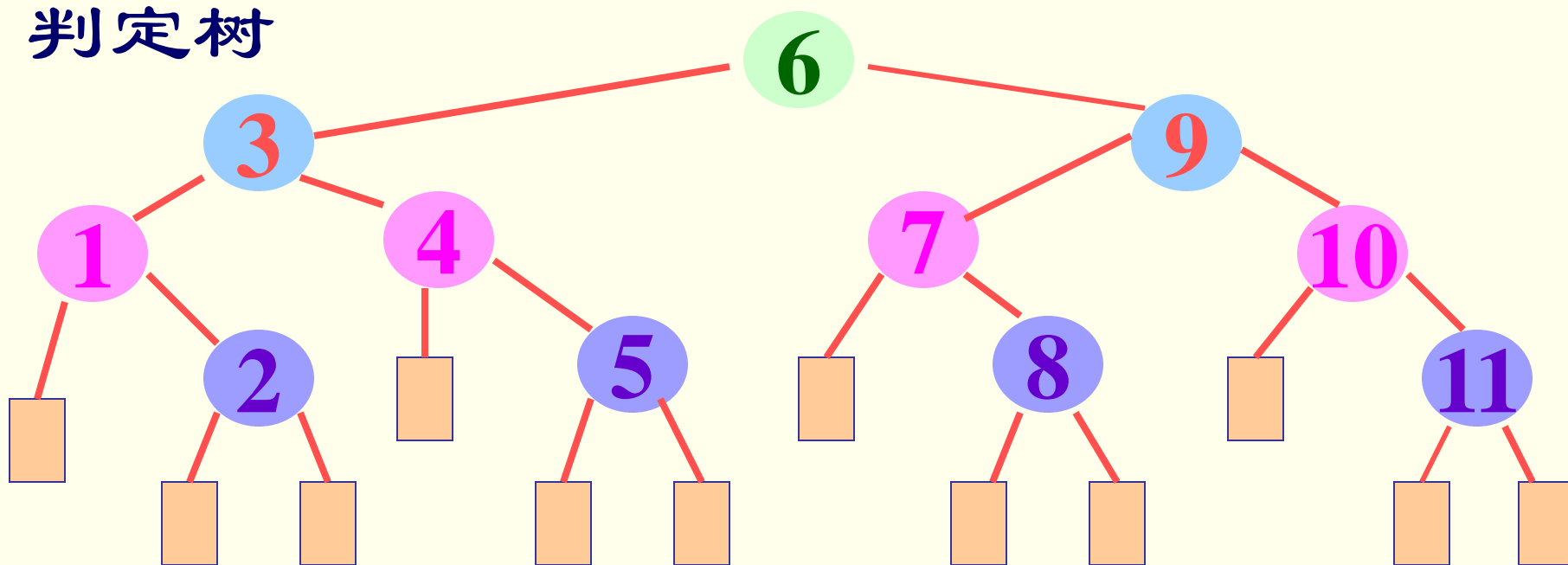
```
int Search_Bin ( SSTable ST, KeyType key ) {
    low = 1; high = ST.length;    // 置区间初值
    while (low <= high) {
        mid = (low + high) / 2;
        if ( EQ (key , ST.elem[mid].key) )
            return mid;            // 找到待查元素
        else if ( LT (key , ST.elem[mid].key) )
            high = 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
```

### (3)分析折半查找的时间性能 (ASL)

先看一个具体的情况，假设：n=11

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4

判定树





上述查找过程可用二叉树（又称为判定树）来描述。树中每个园结点（内部结点）代表表中的一个记录，园结点中的值代表该记录在表中的位置。方形结点（外部结点）是人为增加的。

查找成功时的比较次数不超过判定树的深度。比较次数等于该路径上内部结点的个数。

查找不成功的过程是走了从根结点到外部结点的路径（即树的深度）。比较次数等于该路径上结点个数。不超过  $\lfloor \log_2 n \rfloor + 1$

$n$ 个结点的判定树的深度与 $n$ 个结点的完全二叉树的深度相同。

假设  $n=2^h-1$  并且查找概率相等( $P_i=1/n$ ), 则判定树的深度为h的满二叉树。层次为h的结点有 $2^{h-1}$ 个。则平均查找长度ASL为:

$$ASL_{bs} = \frac{1}{n} \sum_{i=1}^n C_i = \frac{1}{n} \left[ \sum_{j=1}^h j \times 2^{j-1} \right] = \frac{n+1}{n} \log_2(n+1) - 1$$

在 $n>50$ 时, 可得近似结果

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

# 四、索引顺序表

索引顺序查找又称**分块查找**

## (1) 表中数据元素的特点

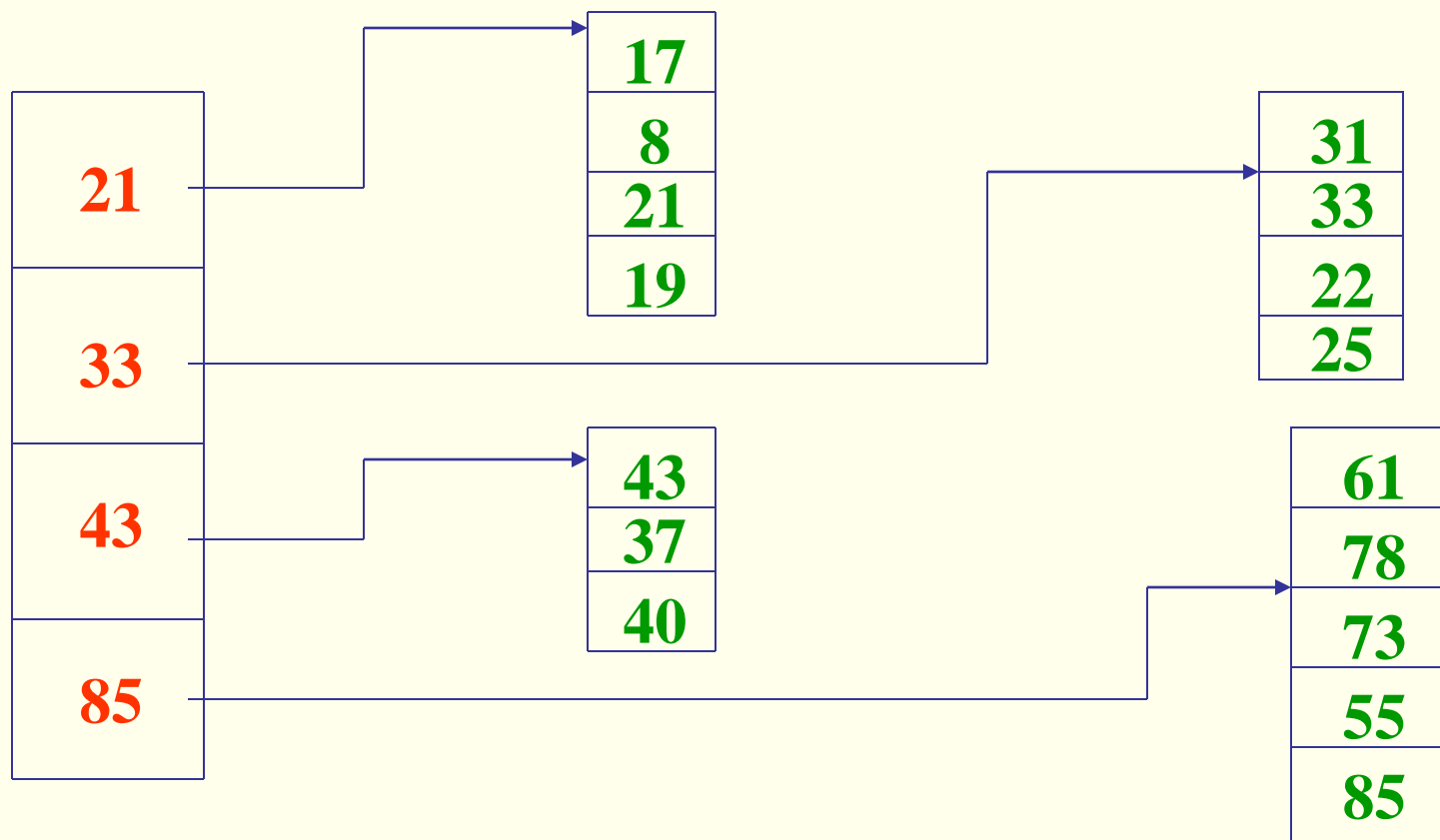
若把所有 $n$ 个数据元素分成 $m$ 块，第一块中任一元素的关键字都小于第二块中任一元素的关键字，第二块中任一元素的关键字都小于第三块中的任一元素的关键字...，第 $m-1$ 块中任一元素的关键字都小于第 $m$ 块中的任一元素的关键字，而**每一块中元素的关键字不一定是有序的。**

# 索引顺序查找示例

将17,8,21,19,31,33,22,25,43,37,40,61,78,73,55,85 分为4块:

[17,8,21,19], [31,33,22,25], [43,37,40], [61,78,73,55,85]

以每块中最大关键字作为该块所有元素的索引



## (2) 索引顺序查找算法

基本思想:

①先抽出各块中的最大关键字构成一个索引表;

②查找分两步进行:

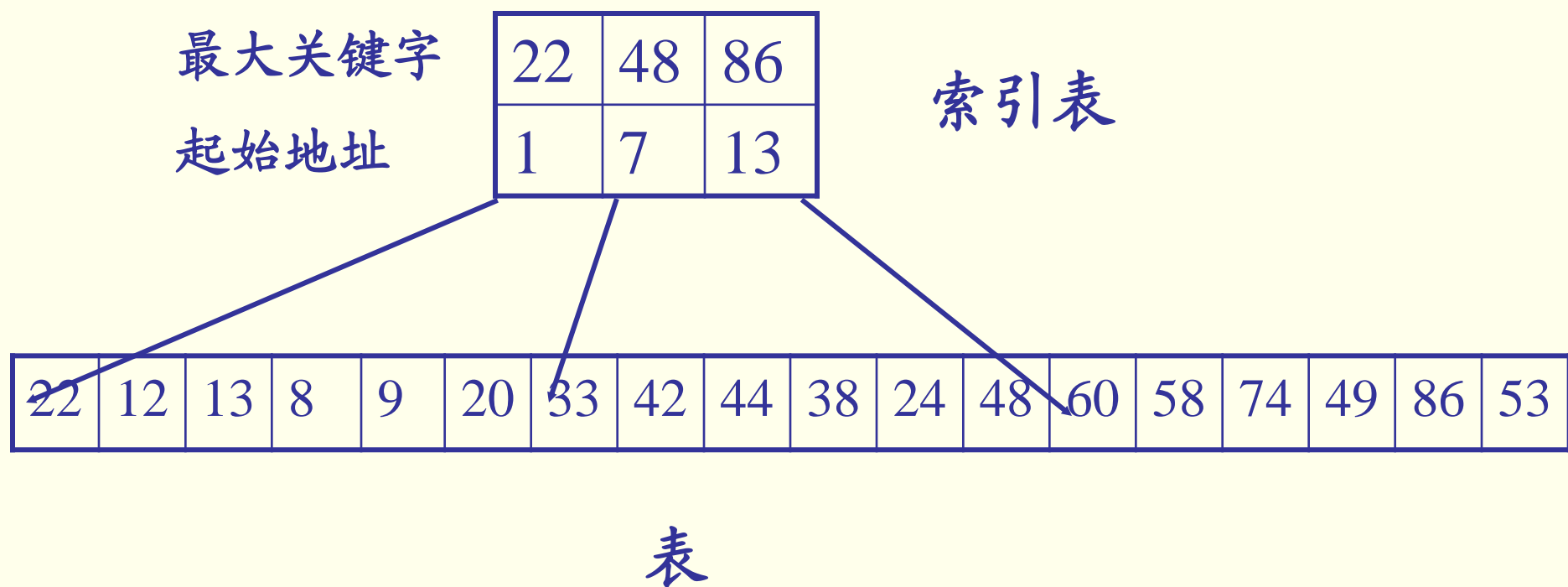
(a) 先对索引表进行折半查找或顺序查找, 确定待查记录在哪一块。

(b) 在已确定的那一块中进行顺序查找。

索引表按关键字有序，包含两项内容：

(1) 索引关键字——其值为该子表（块）内的最大关键字

(2) 指针项——指示该子表的第一个记录在表中的位置



# 索引顺序表的查找过程:

- 1) 由索引确定记录所在区间（块）；
- 2) 在顺序表的某个区间（块）内进行查找。

可见，

索引顺序查找的过程也是一个  
“缩小区间”的查找过程。

### (3) 索引顺序表查找性能分析

索引顺序查找的平均查找长度 =

查找“索引表”的平均查找长度 $L_b$

+ 查找“顺序子表”的平均查找长度 $L_w$



假设长度为  $n$  的表均匀地分成  $b$  块，每块含有  $s$  个记录，即  $b = \lceil n/s \rceil$ 。假定每条记录的查找概率相等，则每块查找的概率为  $1/b$ ，块中每个记录查找的概率为  $1/s$ ，采用顺序查找索引表+顺序查找被确定的块，其平均查找长度为：

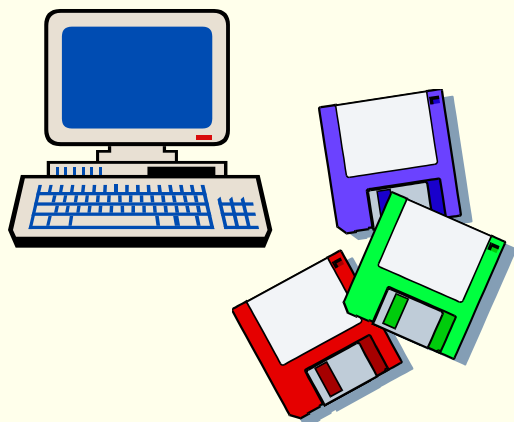
$$\begin{aligned} ASL &= L_b + L_w = (b+1)/2 + (s+1)/2 \\ &= (n/s + s)/2 + 1 \end{aligned}$$

由此可见，索引顺序表的平均查找长度与表长  $n$  和块中记录数  $s$  有关。

当  $s$  取  $\sqrt{n}$  时，ASL取最小值

# 9.2

## 动态查找树表



## 动态查找表的特点:

表结构本身是在查找过程中动态生成的，即对于给定值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。

抽象数据类型 **动态查找表** 的定义如下:

**ADT DynamicSearchTable {**

**数据对象D:** D是具有相同特性的数据元素的集合。  
每个数据元素含有类型相同的關鍵字，可唯一标识数据元素。

**数据关系R:** 数据元素同属一个集合。

## 基本操作P:

InitDSTable(&DT)

DestroyDSTable(&DT)

SearchDSTable(DT, key);

InsertDSTable(&DT, e);

DeleteDSTable(&DT, key);

TraverseDSTable(DT, Visit());

}ADT DynamicSearchTable

InitDSTable(&DT)

**操作结果：**构造一个空的动态查找表DT。

DestroyDSTable(&DT)

**初始条件：**动态查找表DT存在；

**操作结果：**销毁动态查找表DT。

## SearchDSTable(DT, key)

**初始条件：** 动态查找表DT存在，**key**为和关键字类型相同的给定值；

**操作结果：** 若DT中存在其关键字等于 **key** 的数据元素，则函数值为该元素的值或在表中的位置，否则为“空”。

## InsertDSTable(&DT, e)

**初始条件：** 动态查找表DT存在， **e** 为待插入的数据元素；

**操作结果：** 若DT中不存在其关键字等于 **e.key** 的数据元素，则插入 e 到DT。



## DeleteDSTable(&DT, key)

**初始条件:** 动态查找表DT存在, key为和关键字类型相同的给定值;

**操作结果:** 若DT中存在其关键字等于key的数据元素, 则删除之。

# TraverseDSTable(DT, Visit())

**初始条件:** 动态查找表DT存在, **Visit**是对结点操作的应用函数;

**操作结果:** 按某种次序对DT的每个结点调用函数 **Visit()** 一次且至多一次。一旦 **Visit()** 失败, 则操作失败。

一、二叉排序树（二叉查找树）

二、二叉平衡树

三、B - 树

四、B<sup>+</sup>树

五、键树



# 一、二叉排序树(二叉查找树)

1. 定义
2. 查找算法
3. 插入算法
4. 删除算法
5. 查找性能的分析



# 1. 二叉排序树(BST树)定义:

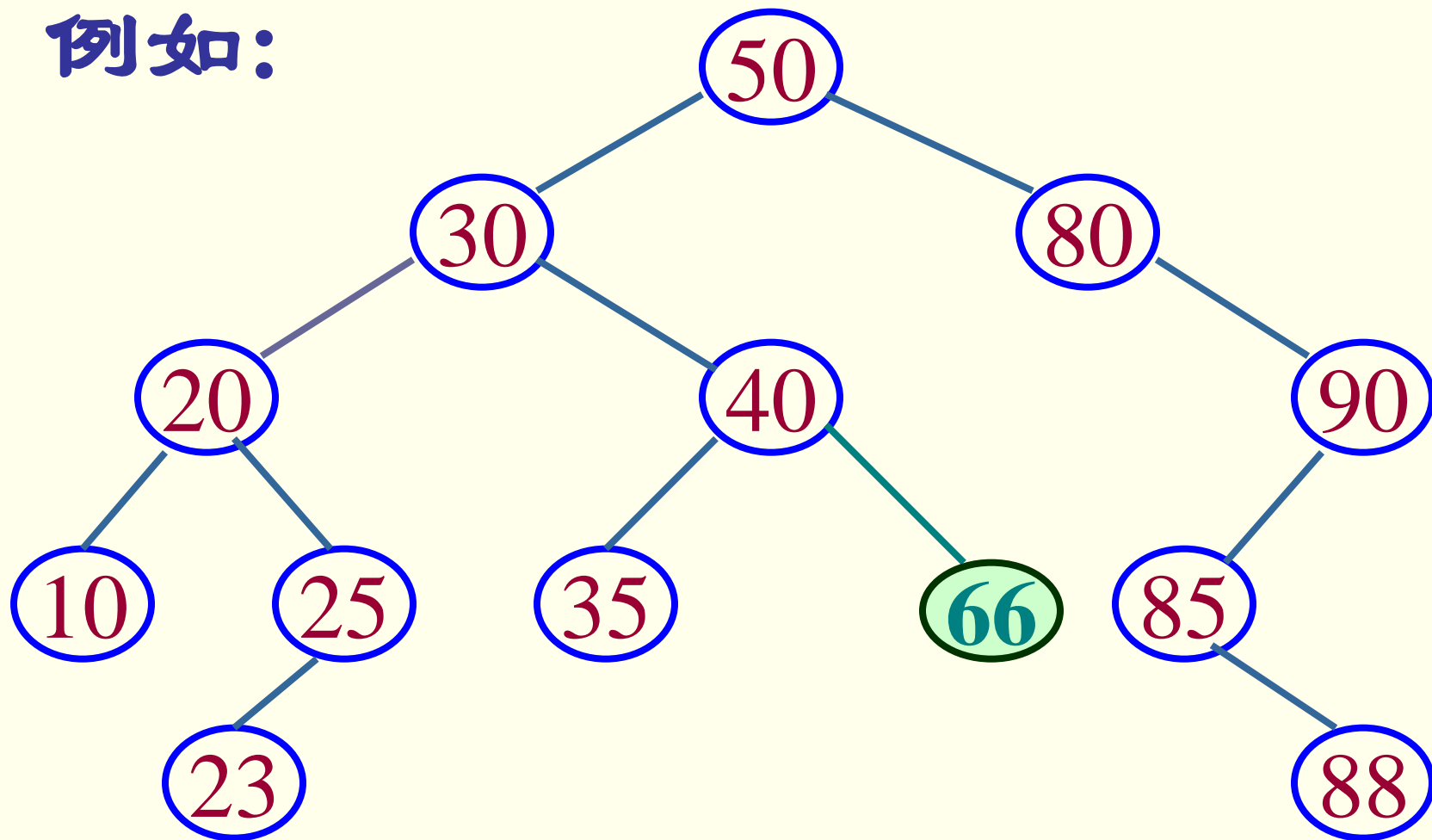
二叉排序树或者是一棵空树; 或者是具有如下特性的二叉树:

(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。

例如：



不是二叉排序树。

## 通常，取二叉链表作为二叉排序树的存储结构

```
typedef struct BiTNode { // 结点结构  
    TElemType      data;  
    struct BiTNode *lchild, *rchild;  
  
                                // 左右孩子指针  
  
} BiTNode, *BiTree;
```

## 2. 二叉排序树的查找算法:

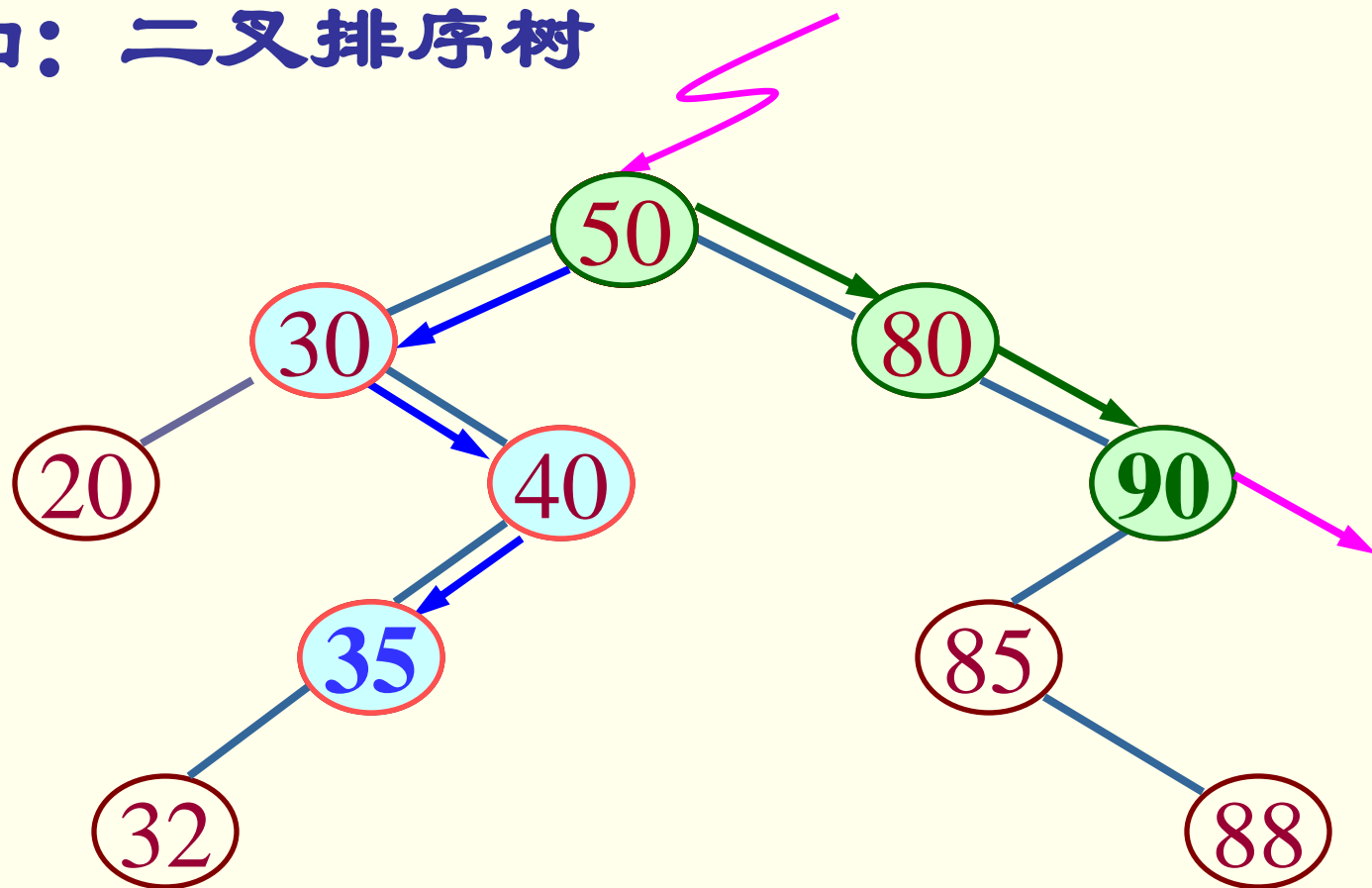
若二叉排序树**为空**，则查找不成功；

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**。



## 例如：二叉排序树



查找关键字

== 50 , 35 , 90 , 95 ,

从上述查找过程可见，

在查找过程中，生成了一条**查找路径**：

从根结点出发，沿着左分支或右分支逐层向下直至关键字等于给定值的结点；

——**查找成功**

或者

从根结点出发，沿着左分支或右分支逐层向下直至指针指向空树为止。

——**查找不成功**

## 算法描述如下:

```
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,  
    // 则返回指针 p 所指该数据元素的结点, 并返回  
    // 函数值为 TRUE; 否则表明查找不成功, 返回  
    // 指针 p所指查找路径上访问的最后一个结点,  
    // 并返回函数值为FALSE, 指针 f 指向当前访问  
    // 的结点的双亲, 其初始调用值为NULL  
    ... ..  
} // SearchBST
```

**if (!T)**

**{ p = f; return FALSE; }** // 查找不成功

**else if ( EQ(key, T->data.key) )**

**{ p = T; return TRUE; }** // 查找成功

**else if ( LT(key, T->data.key) )**

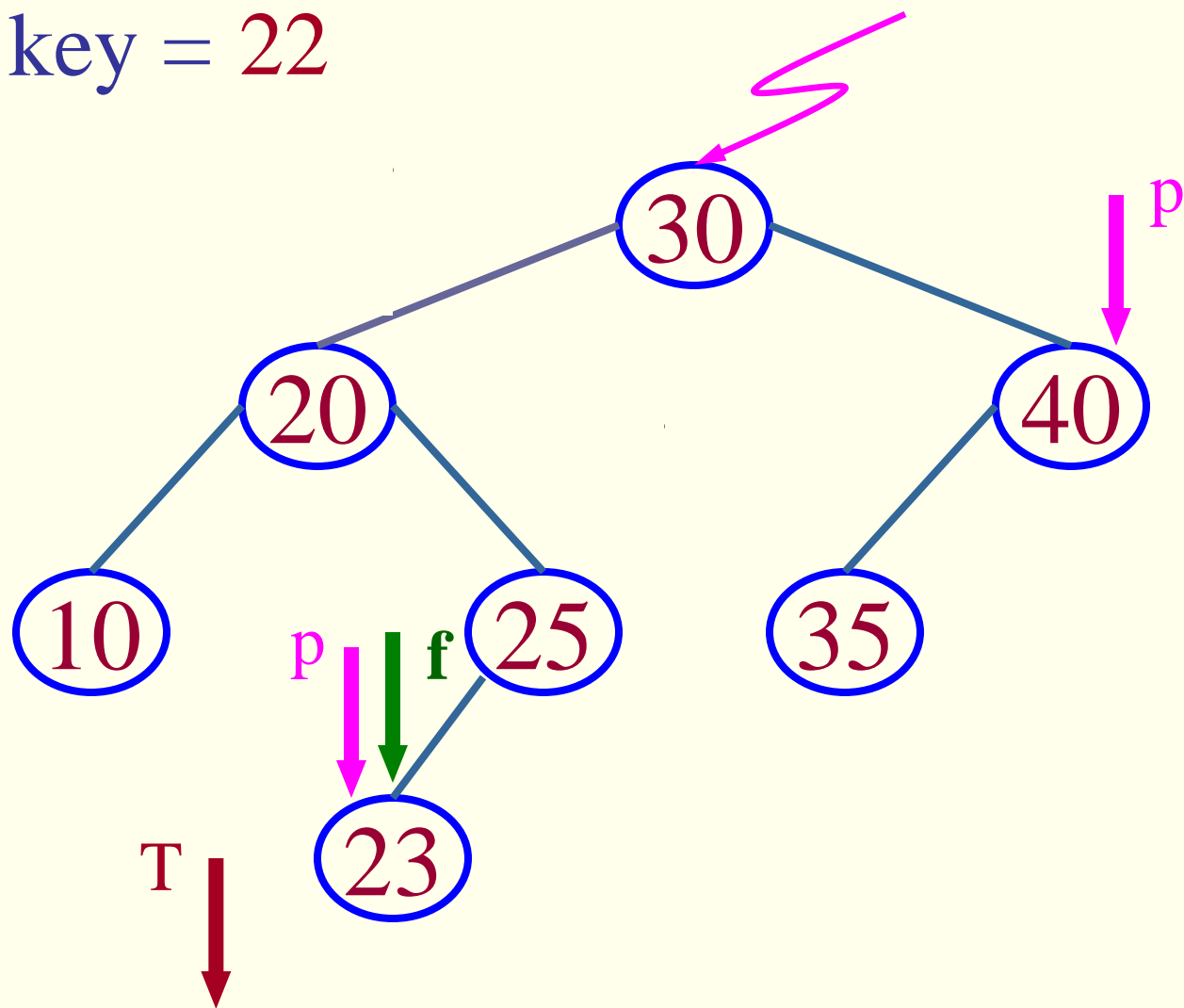
**SearchBST (T->lchild, key, T, p );**

// 在左子树中继续查找

**else SearchBST (T->rchild, key, T, p );**

// 在右子树中继续查找

设  $\text{key} = 22$



### 3. 二叉排序树的插入算法

根据动态查找表的定义，“插入”操作在**查找不成功**时才进行；

若二叉排序树为**空树**，则新插入的结点为**新的根结点**；否则，新插入的结点必为一个**新的叶子结点**，其插入位置由查找过程得到。

```

Status Insert BST(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，插入元素值为 e 的结点，并返
    // 回 TRUE; 否则，不进行插入并返回FALSE
    if (!SearchBST ( T, e.key, NULL, p ))
    {
        ...
    }
    else return FALSE;
} // Insert BST

```

**s = (BiTree) malloc (sizeof (BiTNode));**

*// 为新结点分配空间*

**s->data = e;**

**s->lchild = s->rchild = NULL;**

**if ( !p )** T = s; *// 插入 s 为新的根结点*

**else if ( LT(e.key, p->data.key) )**

**p->lchild = s;** *// 插入 \*s 为 \*p 的左孩子*

**else** p->rchild = s; *// 插入 \*s 为 \*p 的右孩子*

**return TRUE;** *// 插入成功*



## 4. 二叉排序树的删除算法

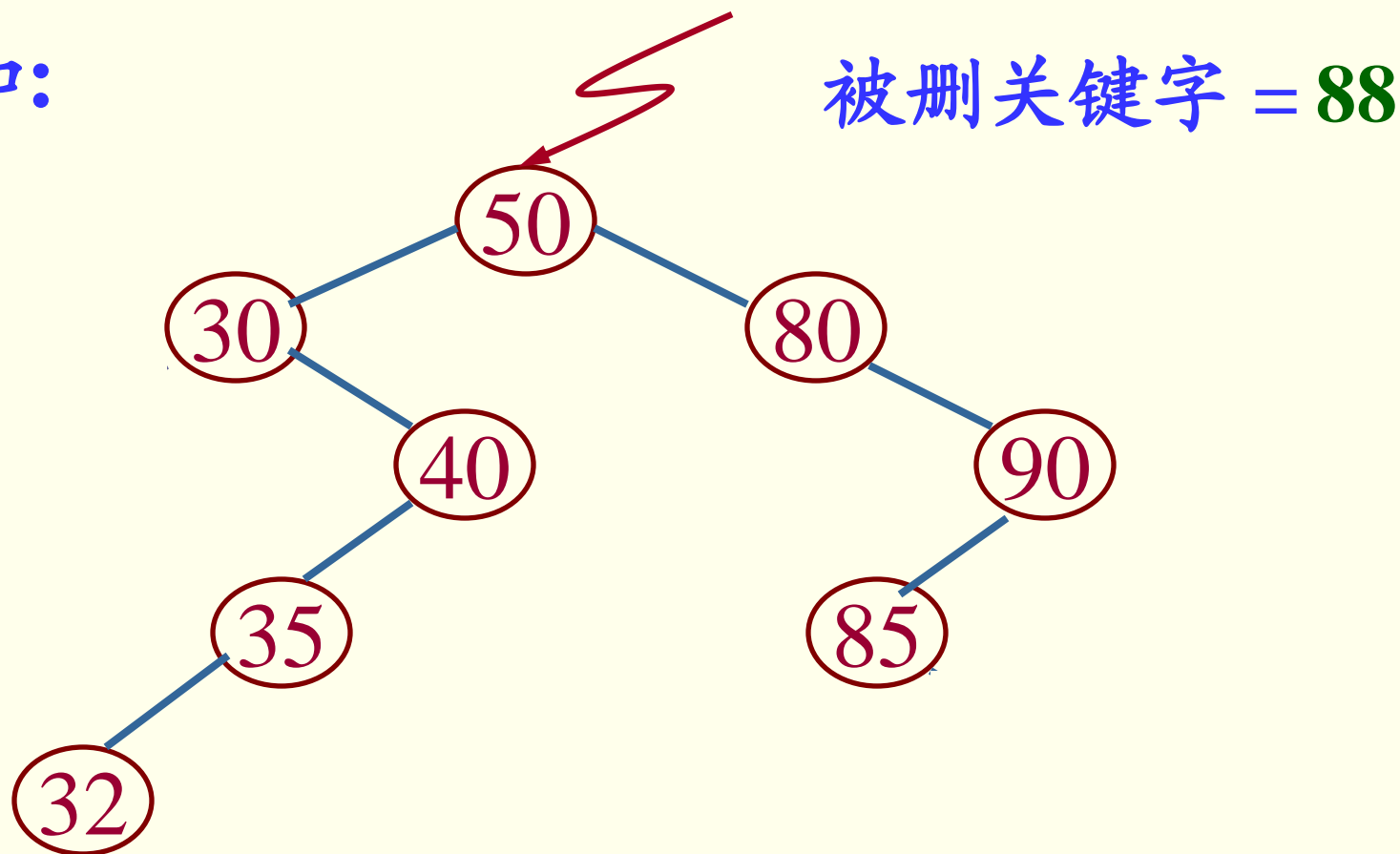
和插入相反，删除在**查找成功**之后进行，并且要求在删除二叉排序树上某个结点之后，**仍然保持二叉排序树的特性**。

可分**三种情况**讨论：

- (1) 被删除的结点是**叶子**；
- (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**。

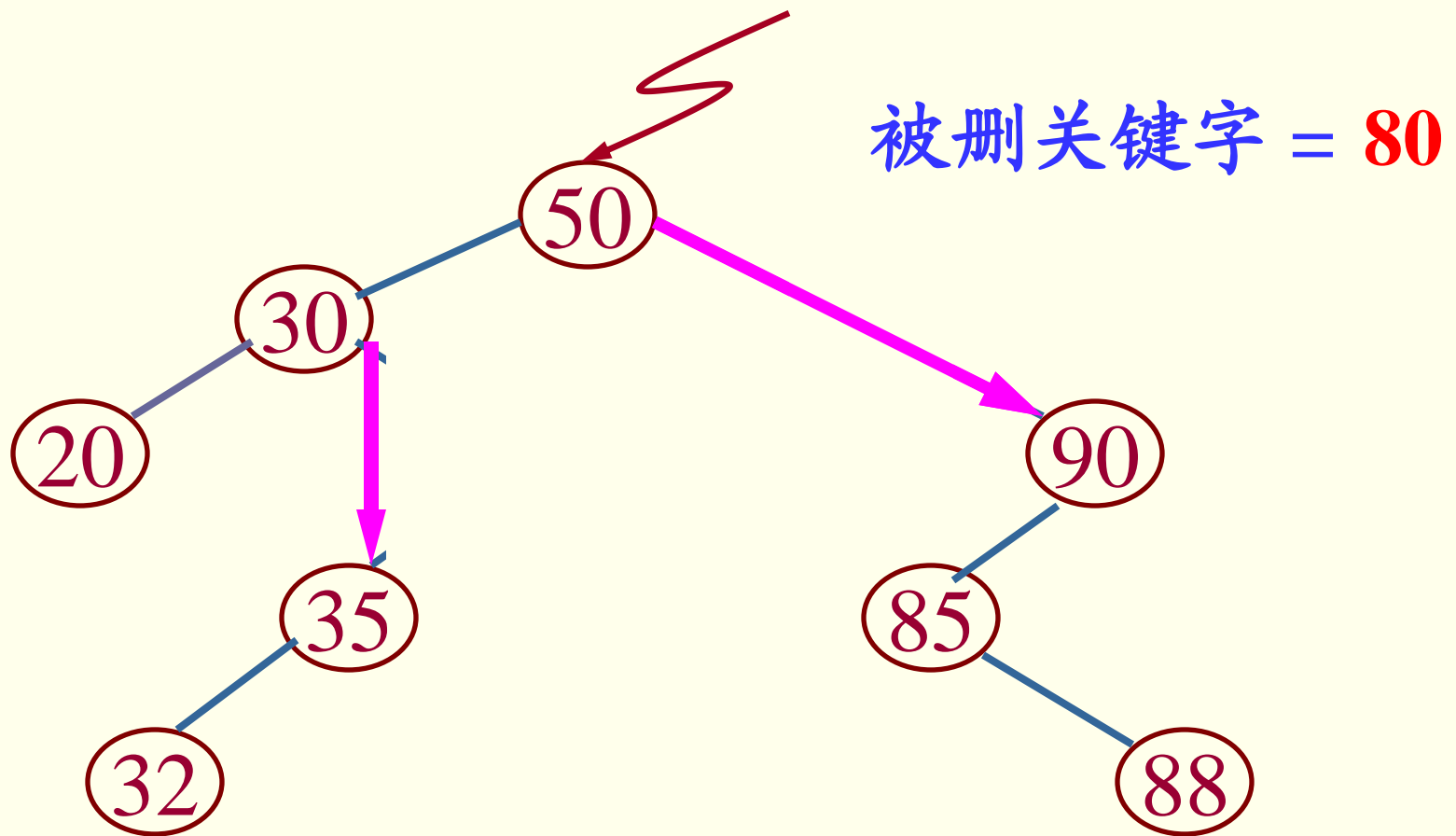
(1) 被删除的结点是叶子结点

例如:



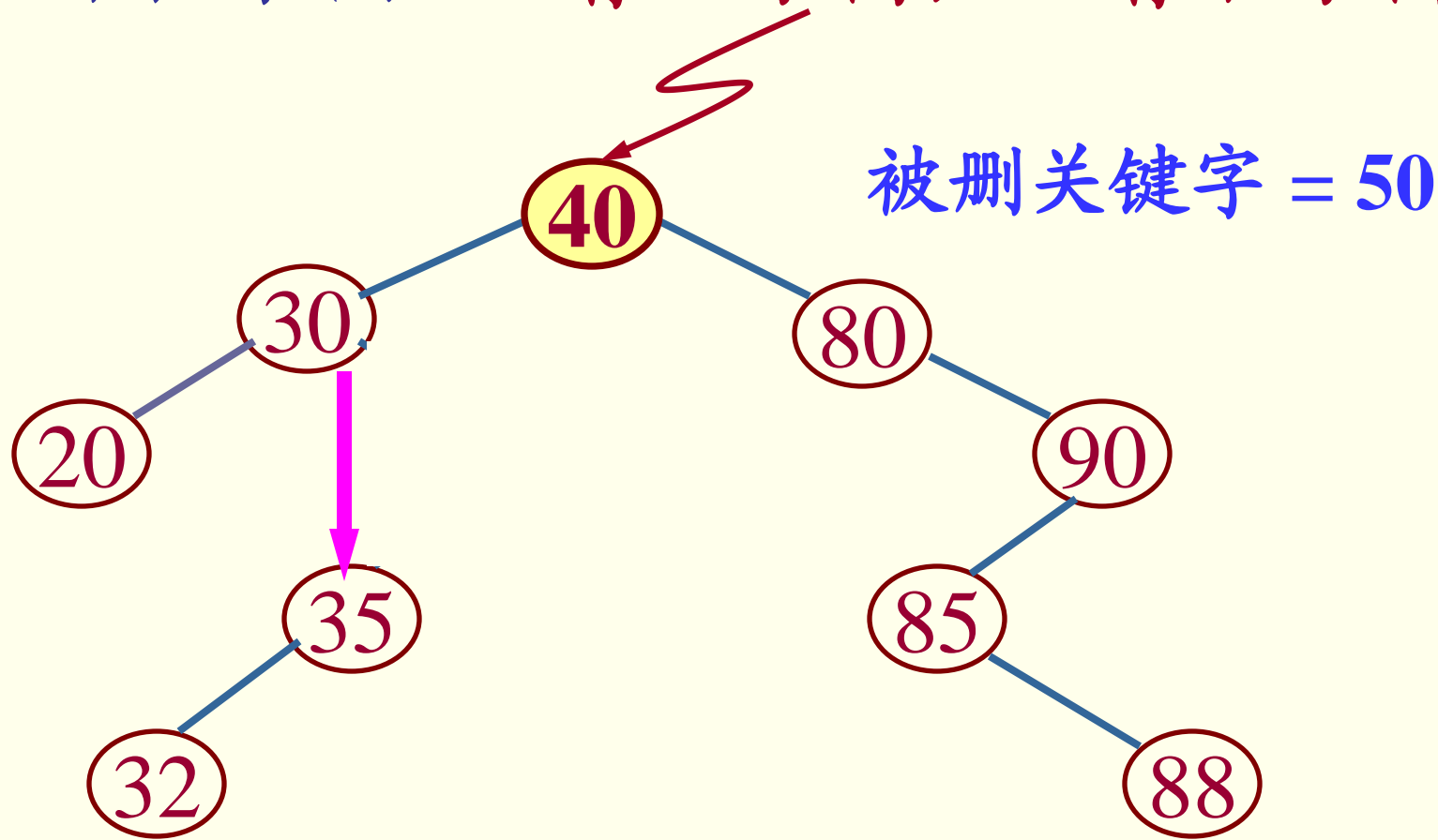
其双亲结点中相应指针域的值改为“空”

(2) 被删除的结点只有左子树或者只有右子树



其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。

### (3) 被删除的结点既有左子树，也有右子树



前驱结点      被删结点



以其中序前驱替代之，  
然后再删除该前驱结点

算法描述如下:

**Status DeleteBST (BiTree &T, KeyType key ) {**

**// 若二叉排序树 T 中存在其关键字等于 key 的**

**// 数据元素, 则删除该数据元素结点, 并返回**

**// 函数值 TRUE, 否则返回函数值 FALSE**

**if (!T) return FALSE;**

**// 不存在关键字等于key的数据元素**

**else {     ...     }**

**} // DeleteBST**

**if ( EQ (key, T->data.key) )**

**{ Delete (T); return TRUE; }**

// 找到关键字等于key的数据元素

**else if ( LT (key, T->data.key) )**

**DeleteBST ( T->lchild, key );**

// 继续在左子树中进行查找

**else DeleteBST ( T->rchild, key );**

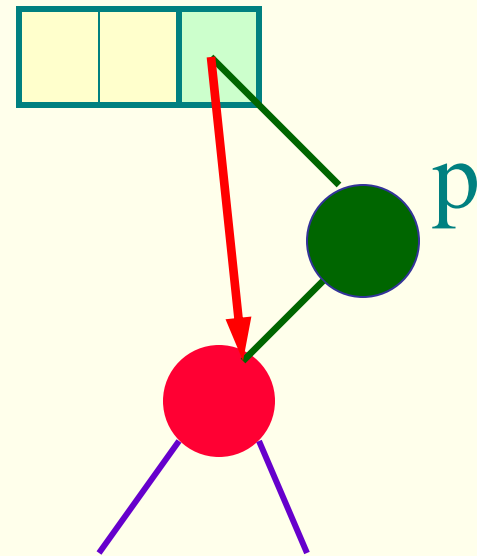
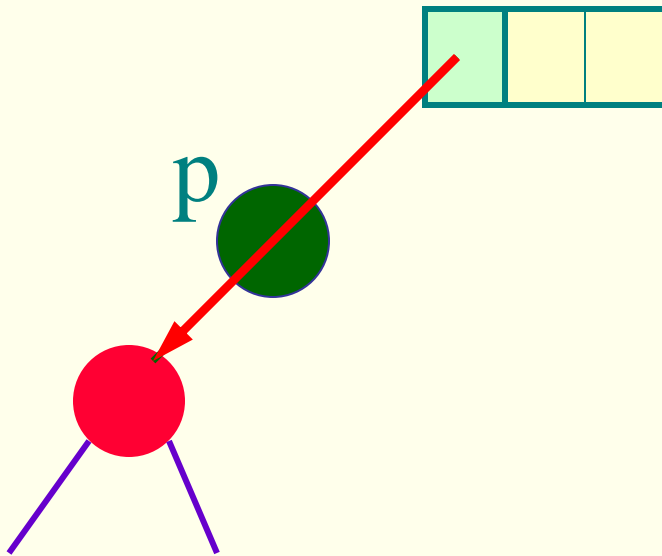
// 继续在右子树中进行查找

其中删除操作过程如下所描述:

```
void Delete ( BiTree &p ){  
    // 从二叉排序树中删除结点 p, 重接它左子树或右子树  
    if (!p->rchild) {q = p; p = p->lchild;  
                    free(q);}           // P只有左孩子  
    else if (!p->lchild) {q = p; p = p->rchild;  
                        free(q);}       // P只有右孩子  
    else { ... ... }                  // P有左、右孩子  
    return TRUE;  
} // Delete
```

// 右子树为空树则只需重接它的左子树

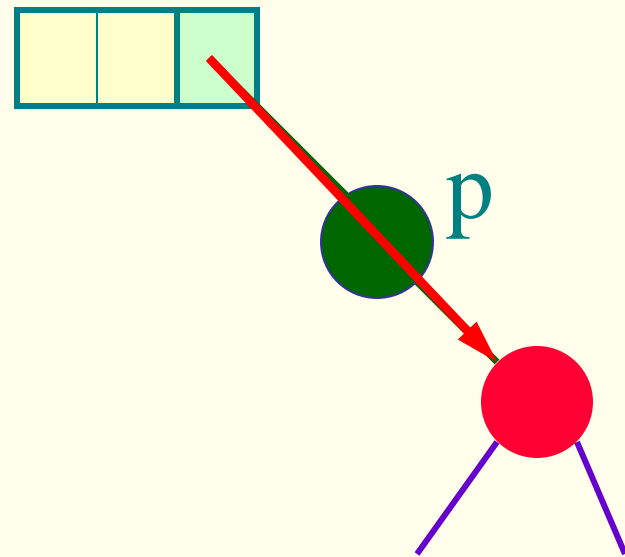
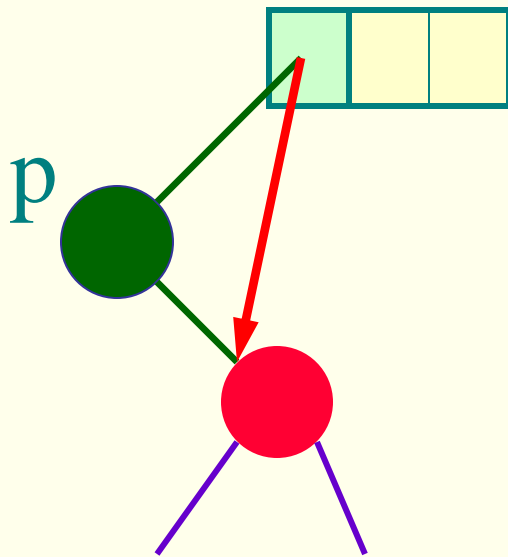
```
q = p; p = p->lchild; free(q);
```





// 左子树为空树只需重接它的右子树

$q = p;$   $p = p \rightarrow \text{rchild};$   $\text{free}(q);$



// 左右子树均不空

q = p; s = p->lchild;

**while** (s->rchild) { q = s; s = s->rchild; }

// 转左，然后向右到尽头， 指向被删结点P的直接前驱S

p->data = s->data; //直接前驱替代结点p

**if** (q != p ) q->rchild = s->lchild; //重接\*q的右子树

**else** q->lchild = s->lchild;

// q=p表明P的左孩子S没有右孩子，则p的左孩子S  
就是p的直接前驱。重接\*q的左子树

free(s);

# 结论:

(1) 一个无序序列 (10, 18, 3, 8, 12, 2, 7, 3) 可以通过构造一棵二叉排序树而变成一个有序序列, 构造树的过程即为对无序序列进行排序的过程;

(2) 每次插入的新结点都是二叉排序树的叶子结点, 在进行插入操作时, 不必移动其它结点, 仅需修改某个结点的指针由空变为非空即可。这就相当于在一个有序序列上插入一个元素而没有移动其它元素。这个特性告诉我们, 对于需要经常插入和删除记录的有序表采用二叉排序树结构更为合适。

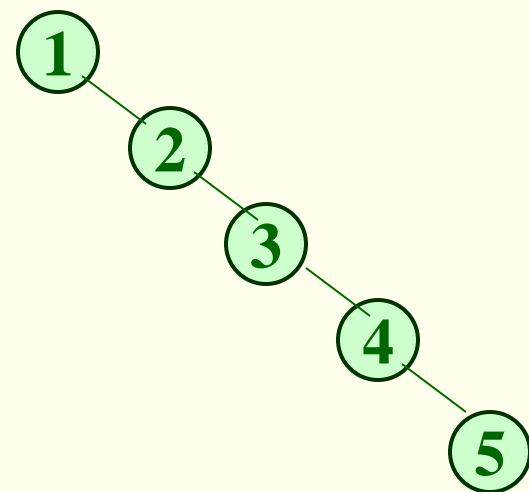
## 5. 查找性能的分析

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，显然，由值相同的  $n$  个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。

例如:

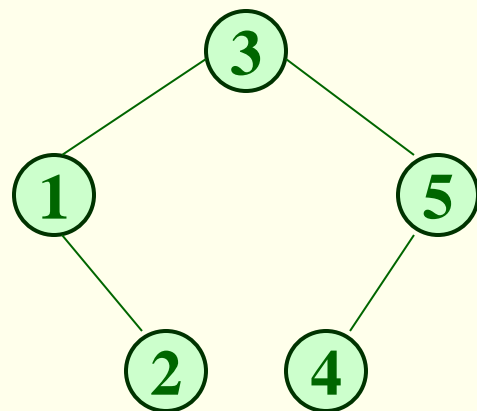
由关键字序列 1, 2, 3, 4, 5  
构造而得的二叉排序树,

$$\begin{aligned} \text{ASL} &= (1+2+3+4+5) / 5 \\ &= 3 \end{aligned}$$



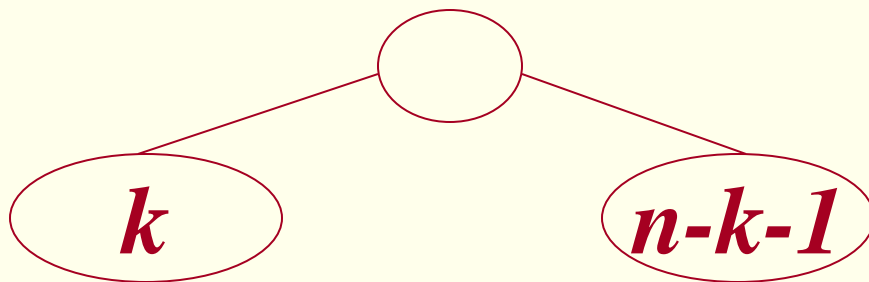
由关键字序列 3, 1, 2, 5, 4  
构造而得的二叉排序树,

$$\begin{aligned} \text{ASL} &= (1+2+3+2+3) / 5 \\ &= 2.2 \end{aligned}$$



## 下面讨论平均情况:

不失一般性, 假设长度为  $n$  的序列中有  $k$  个关键字小于第一个关键字, 则必有  $n-k-1$  个关键字大于第一个关键字, 由它构造的二叉排序树



的平均查找长度是  $n$  和  $k$  的函数

$$P(n, k) \quad (0 \leq k \leq n-1)。$$

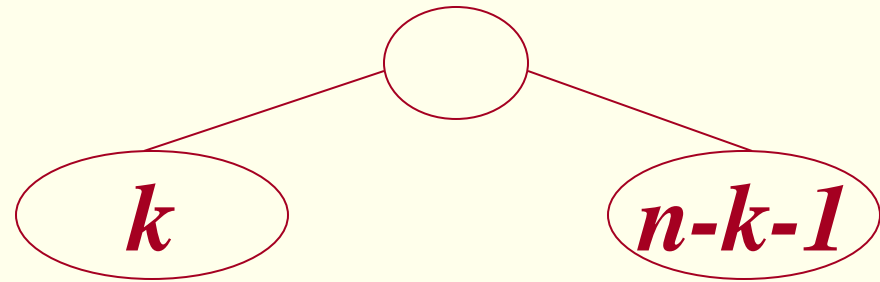
假设  $n$  个关键字中任何一个出现在第一个位置的可能性相同，则含  $n$  个关键字的二叉排序树的平均查找长度：

$$ASL = P(n) = \frac{1}{n} \sum_{k=0}^{n-1} P(n, k)$$

在等概率查找的情况下，

$$P(n, k) = \sum_{i=1}^n p_i C_i = \frac{1}{n} \sum_{i=1}^n C_i$$

$$\begin{aligned}
 P(n,k) &= \frac{1}{n} \sum_{i=1}^n C_i = \frac{1}{n} \left( C_{root} + \sum_L C_i + \sum_R C_i \right) \\
 &= \frac{1}{n} \left( 1 + k(P(k) + 1) + (n - k - 1)(P(n - k - 1) + 1) \right) \\
 &= 1 + \frac{1}{n} \left( k \times P(k) + (n - k - 1) \times P(n - k - 1) \right)
 \end{aligned}$$





由此

$$\begin{aligned} P(n) &= \frac{1}{n} \sum_{k=0}^{n-1} \left( 1 + \frac{1}{n} (k \times P(k) + (n-k-1) \times P(n-k-1)) \right) \\ &= 1 + \frac{2}{n^2} \sum_{k=1}^{n-1} (k \times P(k)) \end{aligned}$$

可类似于解差分方程，此递归方程有解：

$$P(n) = 2 \frac{n+1}{n} \log n + C$$

## 二、二叉平衡树

- 1、何谓“二叉平衡树”？
- 2、如何构造“二叉平衡树”？
- 3、二叉平衡树的查找性能分析

# 1、二叉平衡树概念

又称AVL树。它或者是一棵空树，或者是具有下列性质的二叉树：它的左子树或右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。

## (1) 平衡因子

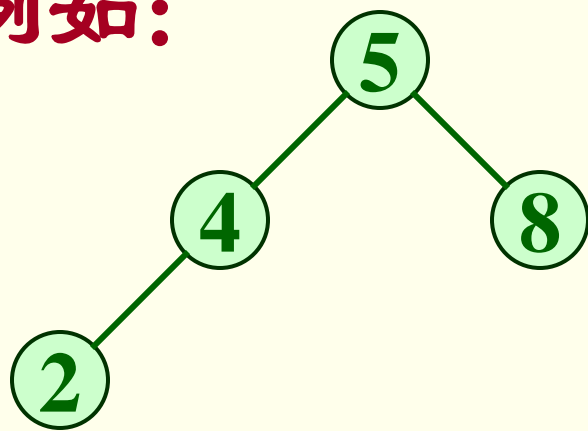
二叉树上任一结点的左子树深度减去右子树深度的差值，称为此结点的平衡因子。

## (2) 特点:

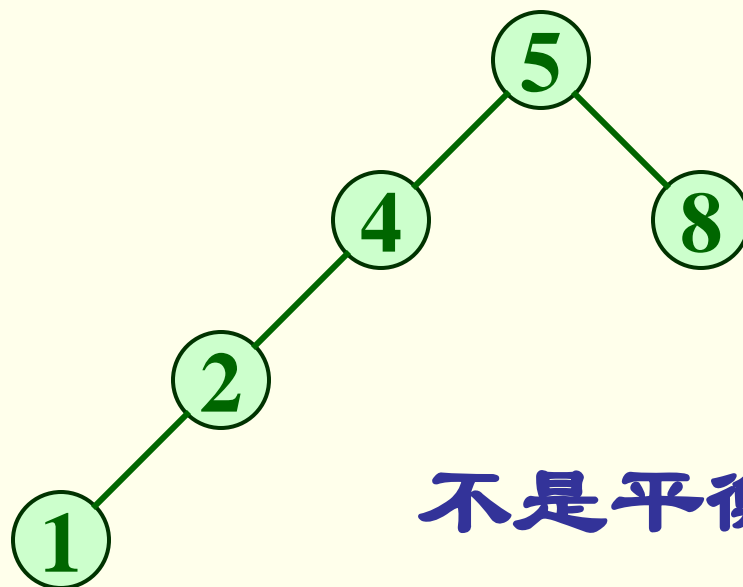
二叉平衡树是二叉查找树的另一种形式，其特点为：

树中每个结点的左、右子树深度之差的绝对值不大于1，即  $|h_L - h_R| \leq 1$ 。

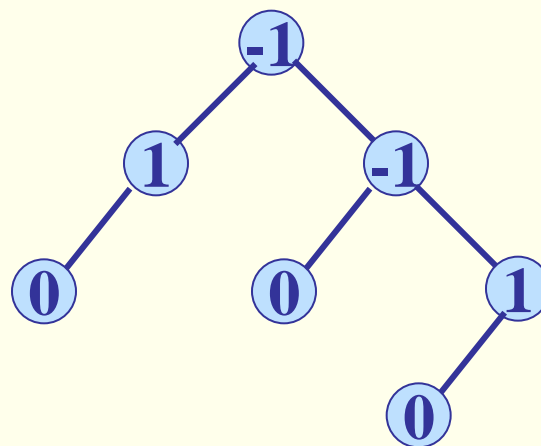
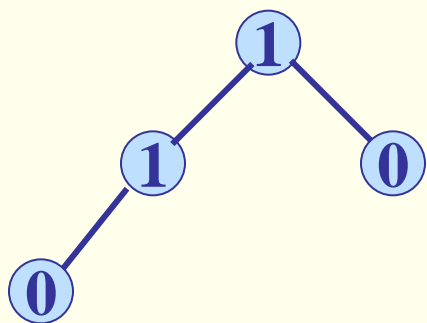
例如：



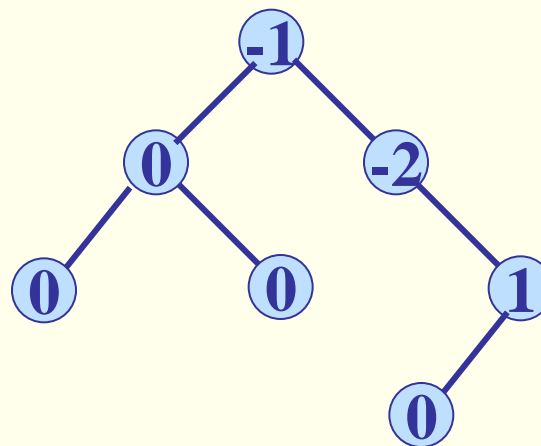
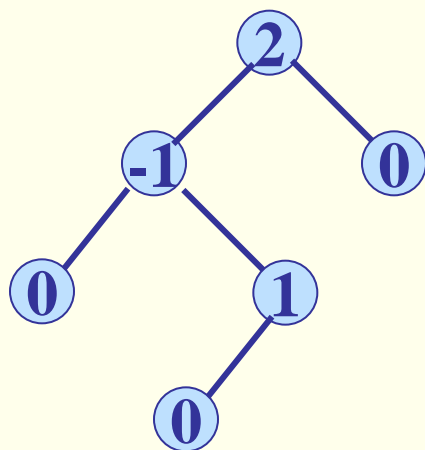
是平衡树



不是平衡树



平衡二叉树

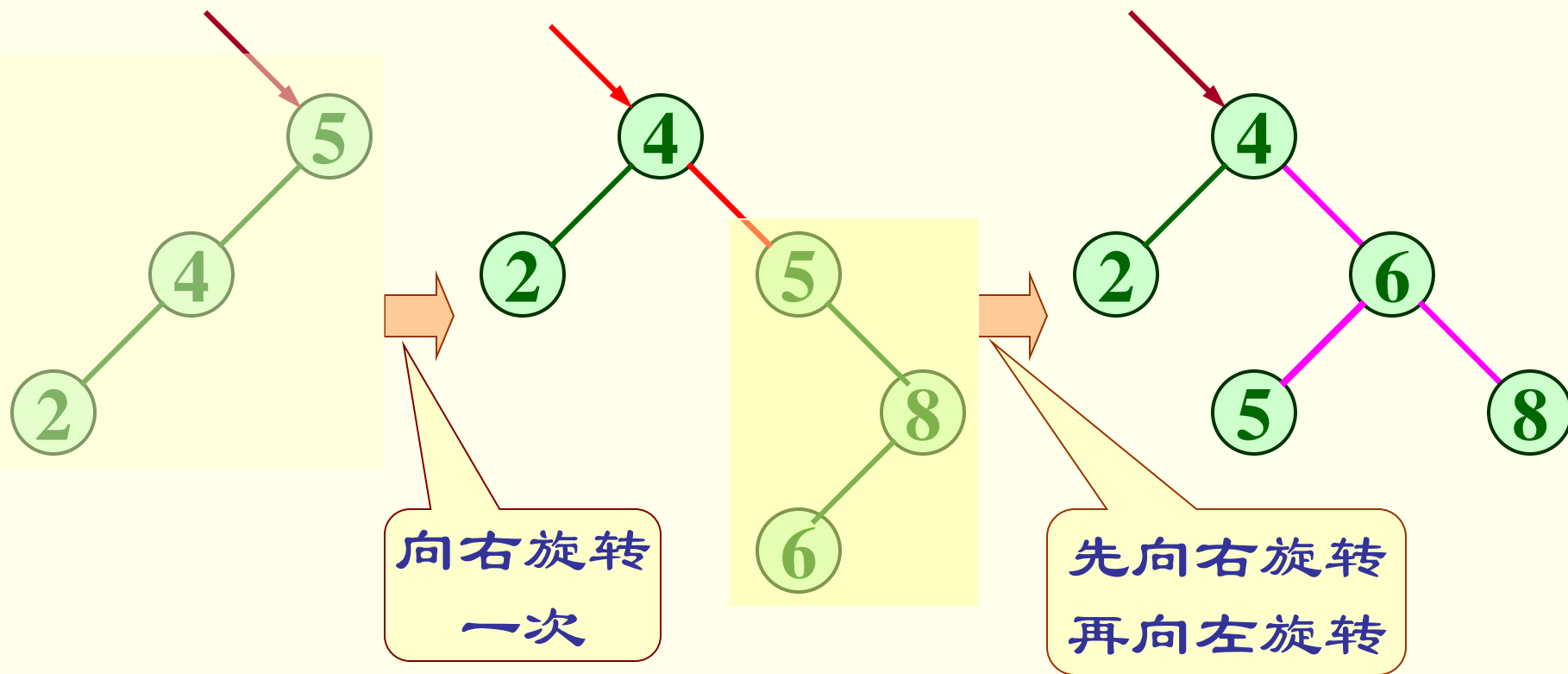


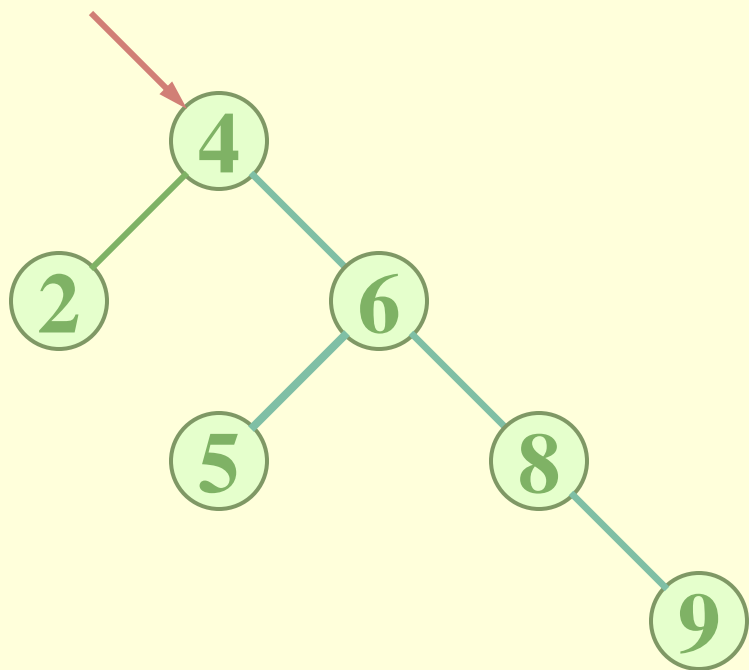
不平衡的二叉树

## 2、构造“二叉平衡树”

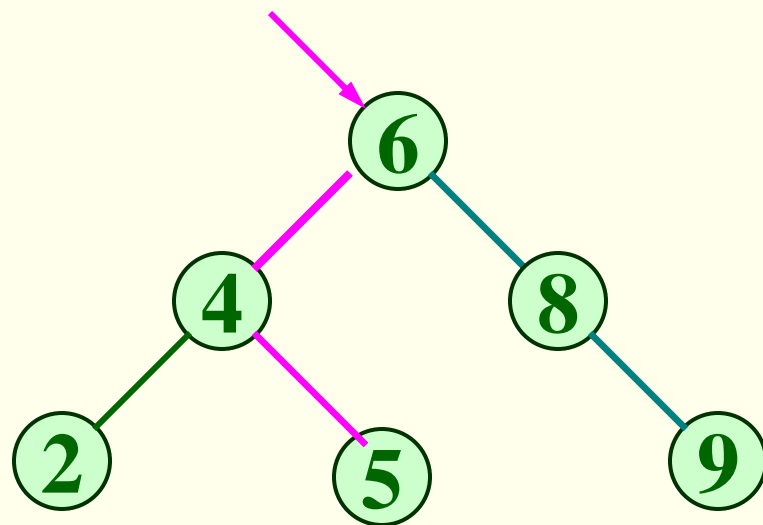
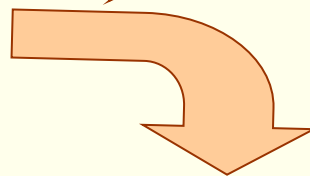
构造二叉平衡（查找）树的方法是：  
在插入过程中，采用平衡旋转技术。

例如：依次插入的关键字为5, 4, 2, 8, 6, 9





向左旋转一次



继续插入关键字 9

平衡旋转处理只对**最小不平衡子树**进行。

设a为插入结点而失去平衡的**最小子树根结点**的指针（即a是离插入结点最近，并且平衡因子绝对值超过1的祖先结点）。

**平衡旋转的规律如下：**

（1）**单向右旋转**平衡处理：在\*a的**左子树根结点的左子树**上插入结点。



(2) 单向左旋转平衡处理: 在\*a的右子树根结点的右子树上插入结点。

(3) 双向旋转--先左后右--平衡处理: 在\*a的左子树根结点的右子树上插入结点。

(4) 双向旋转--先右后左--平衡处理: 在\*a的右子树根结点的左子树上插入结点。

### 3、平衡树的查找性能分析:

在平衡树上进行查找的过程和二叉排序树相同，因此，查找过程中和给定值进行比较的关键字的个数不超过平衡树的深度。

问：含  $n$  个关键字的二叉平衡树可能达到的最大深度是多少？

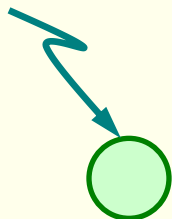
先看几个具体情况:

$n = 0$

空树

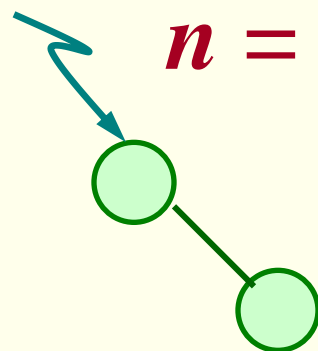
最大深度为 0

$n = 1$



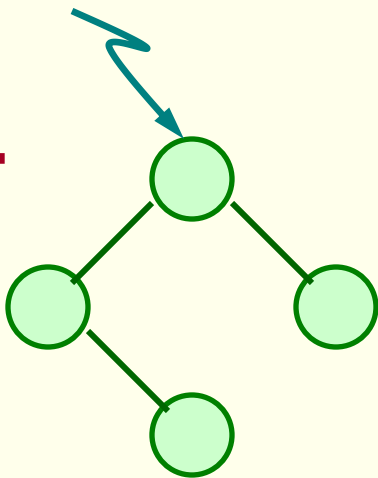
最大深度为 1

$n = 2$



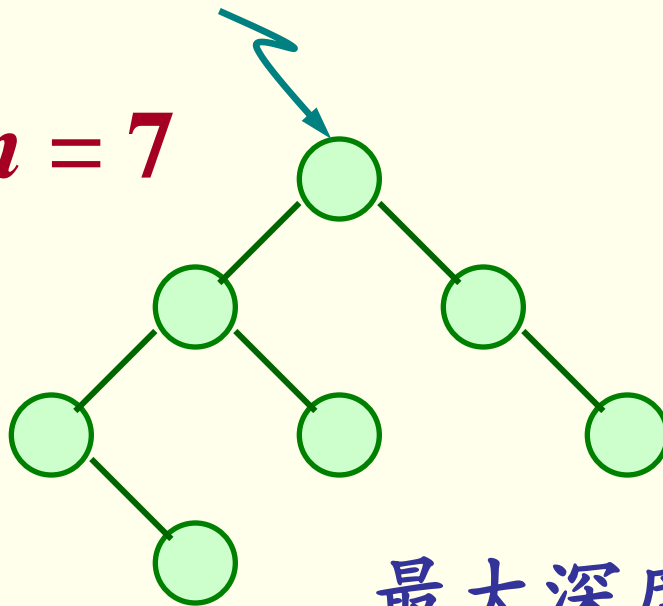
最大深度为 2

$n = 4$



最大深度为 3

$n = 7$



最大深度为 4

反过来问，深度为  $h$  的二叉平衡树中所含结点的最小值  $N_h$  是多少？

$$h = 0 \quad N_0 = 0 \quad h = 1 \quad N_1 = 1$$

$$h = 2 \quad N_2 = 2 \quad h = 3 \quad N_3 = 4$$

一般情况下 
$$N_h = N_{h-1} + N_{h-2} + 1$$

利用归纳法可证得 
$$N_h = F_{h+2} - 1$$

而 
$$F_h \approx \varphi^h / \sqrt{5}$$

其中 
$$\varphi = (1 + \sqrt{5}) / 2$$

由此推得，深度为  $h$  的二叉平衡树中所含结点的最小值  $N_h = \varphi^{h+2} / \sqrt{5} - 1$ 。

反之，含有  $n$  个结点的二叉平衡树能达到的最大深度  $h_n = \log_{\varphi}(\sqrt{5}(n+1)) - 2$ 。

因此，在二叉平衡树上进行查找时，查找过程中和给定值进行比较的关键字的次数和  $\log(n)$  相当。

## 三、 B - 树

1. 定义
2. 查找过程
3. 插入操作
4. 删除操作
5. 查找性能的分析

# 1. B-树的定义

B-树是一种平衡的多路查找树，它在文件系统中很有用。

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；  
(至少含1个关键字)
- ③ 除根之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树。  
(至少含 $\lceil m/2 \rceil - 1$ 个关键字)

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:  $K_i (i=1,2,\dots,n)$  为关键字, 且  $K_i < K_{i+1} (i=1,\dots,n-1)$ ;

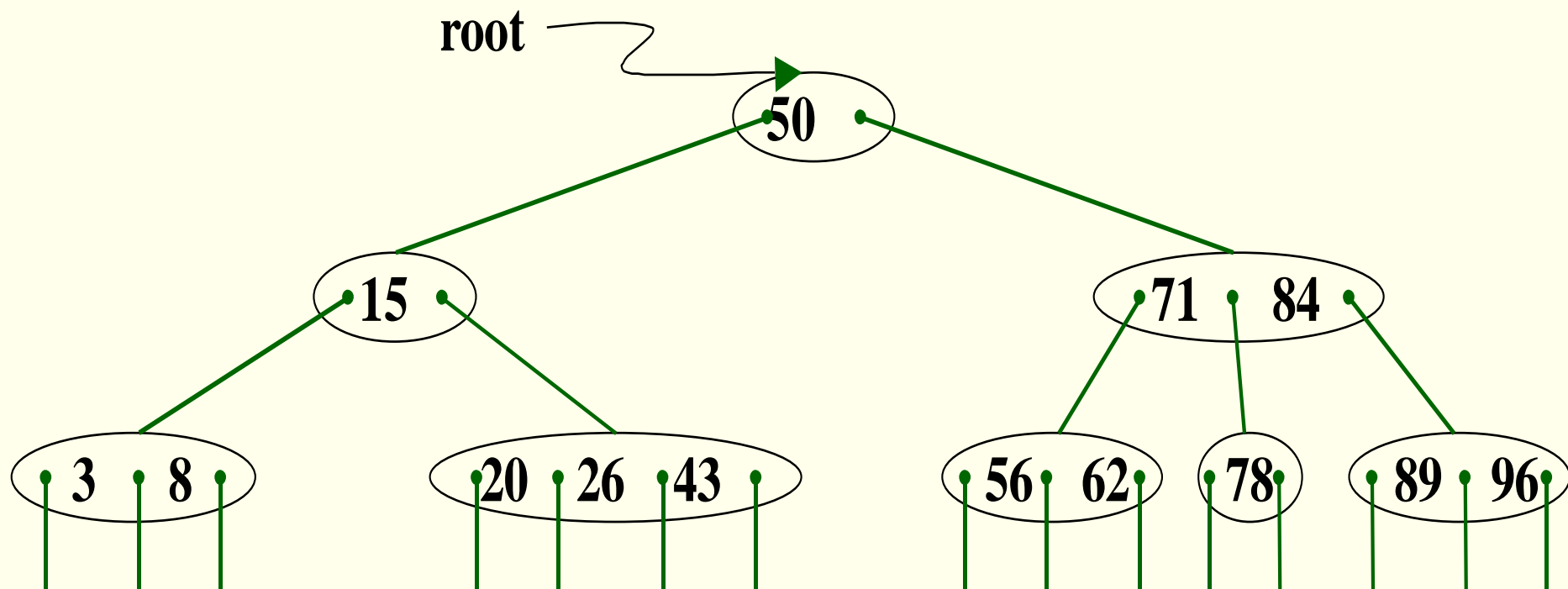
$A_i (i=0,\dots,n)$  为指向子树根结点的指针, 且指针  $A_{i-1}$  所指子树中所有结点的关键字均小于  $K_i (i=1,\dots,n)$ ,  $A_n$  所指子树中所有结点的关键字均大于  $K_n$ ,

$n (\lceil m/2 \rceil - 1 \leq n \leq m-1)$  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(可以看作是外部结点或查找失败的结点, 实际上这些结点不存在, 指向这些结点的指针为空)。



如图所示为一棵4阶的B-树，其深度为3。



在  $m$  阶的B-树上，每个非终端结点可能含有：

$n$  个关键字  $K_i$  ( $1 \leq i \leq n$ )  $n < m$

$n$  个指向记录的指针  $D_i$  ( $1 \leq i \leq n$ )

$n+1$  个指向子树的指针  $A_i$  ( $0 \leq i \leq n$ )

—— 多叉树的特性

- 非叶结点中的多个关键字均自小至大有序排列，即： $K_1 < K_2 < \dots < K_n$ ；
- $A_{i-1}$  所指子树上所有关键字均小于  $K_i$ ；
- $A_i$  所指子树上所有关键字均大于  $K_i$ ；

—— 查找树的特性

树中所有叶子结点均不带信息，且在树中的同一层次上；

根结点或为叶子结点，或至少含有两棵子树（至少含1个关键字）；

其余所有非叶子结点均至少含有 $\lceil m/2 \rceil$ 棵子树（至少含 $\lceil m/2 \rceil - 1$ 个关键字），至多含有  $m$  棵子树（最多含 $m-1$ 个关键字）；

—— 平衡树的特性

## B-树结构的C语言描述如下：

```
typedef struct BTreeNode {  
    int keynum;                // 结点中关键字个数，结点大小  
    struct BTreeNode *parent;  // 指向双亲结点的指针  
    KeyType key[m+1];          // 关键字向量（0号单元不用）  
    struct BTreeNode *ptr[m+1]; // 子树指针向量  
    Record *recptr[m+1];       // 记录指针向量(0号单元不用)  
} BTreeNode, *BTree;          // B-树结点和B-树的类型
```

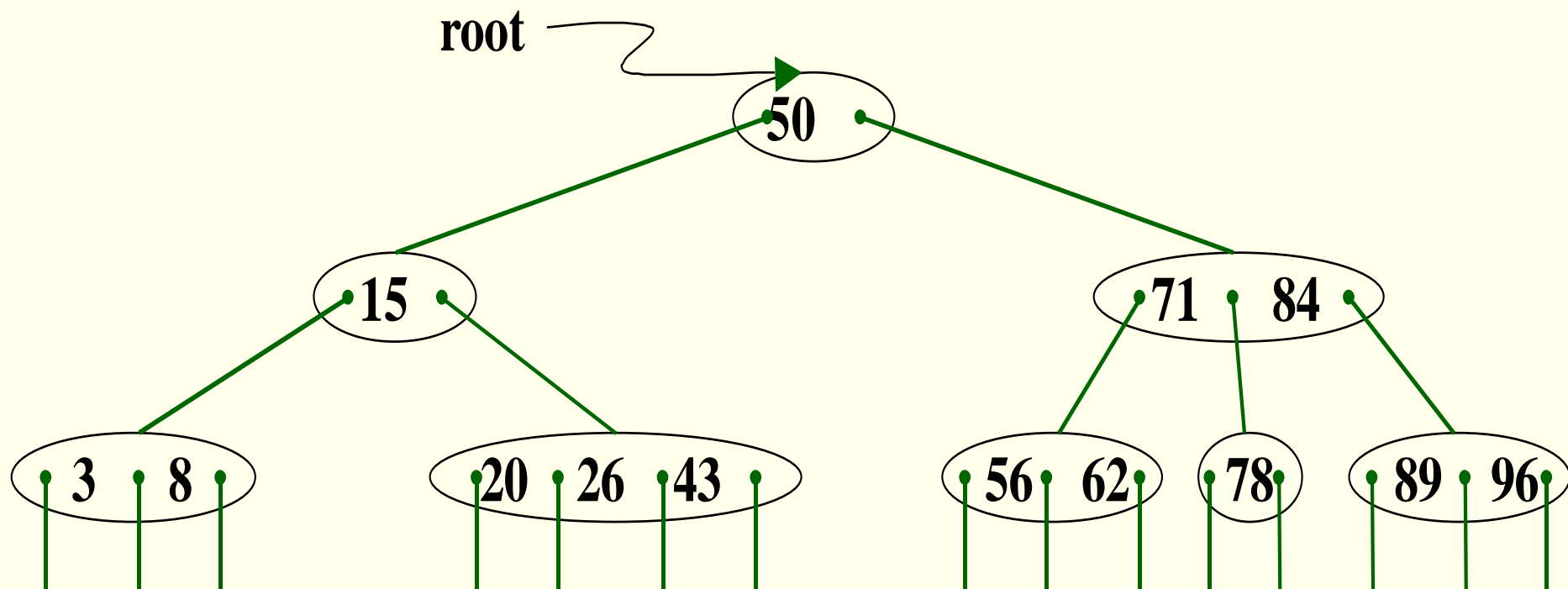
## 2.查找过程:

从根结点出发，沿指针搜索结点和在结点内进行顺序（或折半）查找两个过程交叉进行(查找包含两个操作：在B-树中找结点、在结点中找关键字)。

若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置；

若查找不成功，则返回插入位置。

如图所示为一棵4阶的B-树，其深度为3。



假设返回的是如下所述结构的记录:

```
typedef struct {  
    BTNode *pt; // 指向找到的结点的指针  
    int i;       // 1..m, 在结点中的关键字序号  
    int tag;     // 标志查找成功(=1)或失败(=0)  
} Result;       // 在B树的查找结果类型
```



**Result SearchBTree(BTree T, KeyType K) {**

**// 在m 阶的B-树 T 中查找关键字 K, 返回**

**// 查找结果 (pt, i, tag)。若查找成功, 则**

**// 特征值 tag=1, 指针 pt 所指结点中第 i 个**

**// 关键字等于 K; 否则特征值 tag=0, 等于**

**// K 的关键字应插入在指针 pt 所指结点**

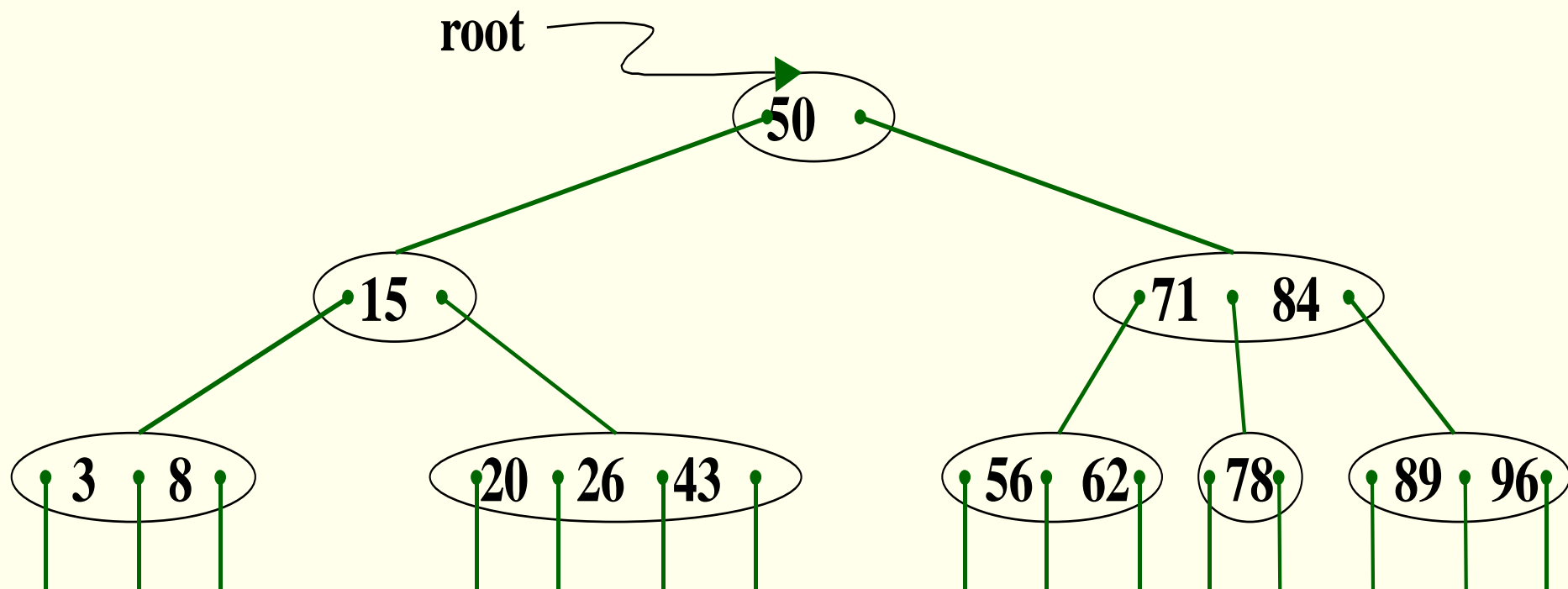
**// 中第 i 个关键字和第 i+1个关键字之间**

**... ..**

**} // SearchBTree**

```
p=T; q=NULL; found=FALSE; i=0;
while (p && !found) {
    n=p->keynum; i=Search(p, K);
    // 在p->key[1..n]中查找 i ,    p->key[i]<=K<p->key[i+1]
    if (i>0 && p->key[i]==K) found=TRUE;
    else { q=p; p=p->ptr[i]; } // q 指示 p 的双亲
}
if (found) return (p,i,1);        // 查找成功
else return (q,i,0);              // 查找不成功
```

如图所示为一棵4阶的B-树，其深度为3。



### 3. 插入

在查找不成功之后，需进行插入。  
显然，关键字插入的位置必定在最下层的非叶子结点，有下列几种情况：

- ✦ 1) 插入后，该结点的关键字个数 $n < m$ ，  
不修改指针；

✧ 2) 插入后，该结点的关键字个数  $n=m$ ，  
则需进行“结点分裂”，令  $s = \lceil m/2 \rceil$ ，  
在原结点中保留

$(A_0, K_1, \dots, K_{s-1}, A_{s-1})$ ；

建新结点

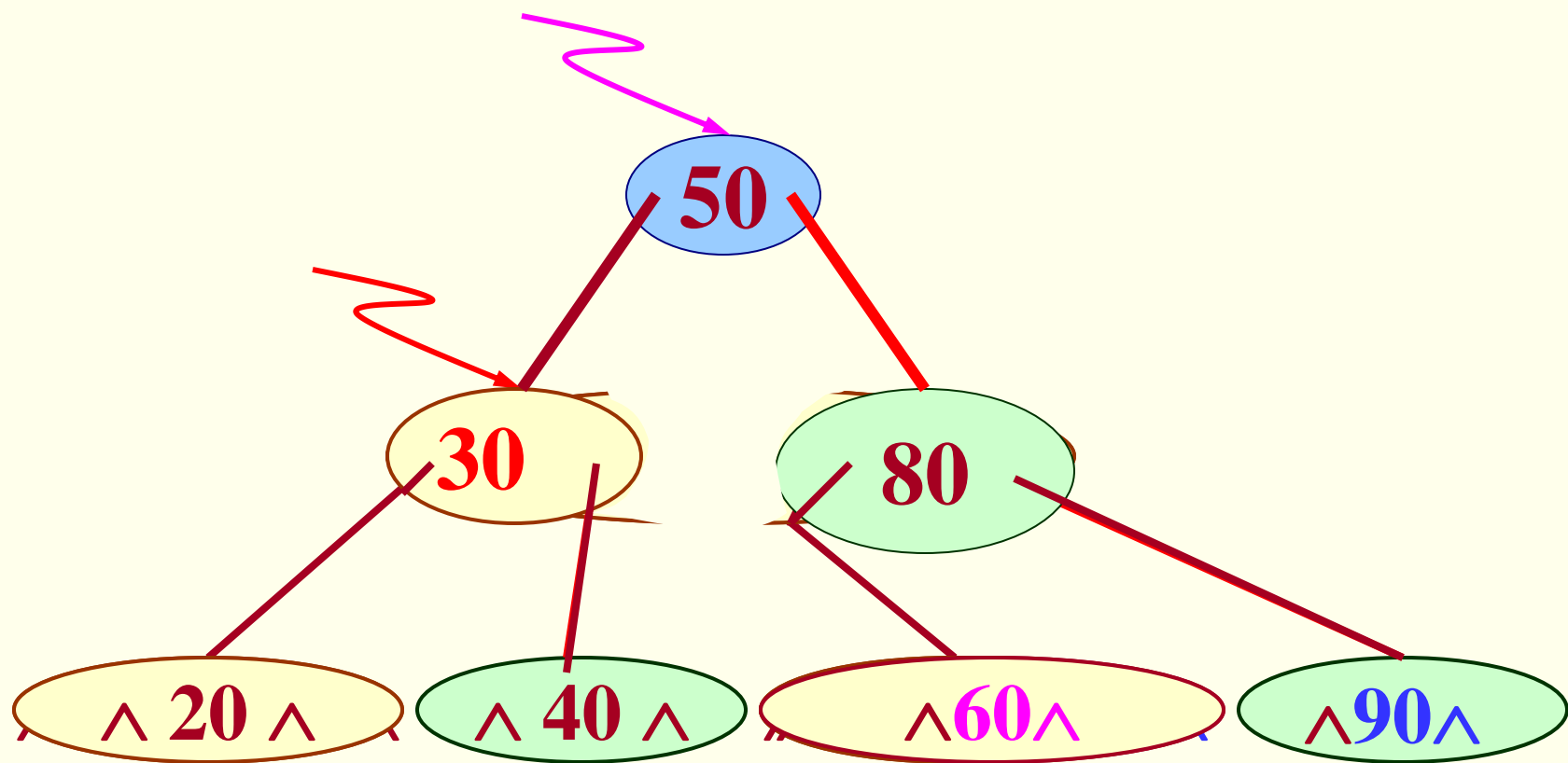
$(A_s, K_{s+1}, \dots, K_n, A_n)$ ；

将  $(K_s, p)$  插入双亲结点；

✧ 3) 若双亲为空，则建新的根结点。

( 算法见P244 )

例如:下列为 3 阶B-树



插入关键字 = 60, 90, 30,

## 4. 删除

和插入的考虑相反，首先必须找到待删关键字所在结点，并且要求删除之后，结点中关键字的个数不能小于 $\lceil m/2 \rceil - 1$ ，否则，要从其左(或右)兄弟结点“借调”关键字，若其左和右兄弟结点均无关键字可借(结点中只有最少量的关键字)，则必须进行结点的“合并”。

## 5. 查找性能的分析

在B-树中进行查找时，其查找时间花费在搜索结点（访问**外存**）和在结点内找关键字（访问**内存**）上，主要花费在搜索结点上。磁盘上查找结点的次数取决于B-树的深度。即主要取决于**B-树的深度**。

**问：**含  $N$  个关键字的  $m$  阶 B-树可能达到的最大深度  $H$  为多少？



反过来问：深度为H的m阶B-树中，  
至少含有多少个结点？

先推导每一层所含最少结点数：

第 1 层                      1 个

第 2 层                      2 个

第 3 层                       $2 \times \lceil m/2 \rceil$  个

第 4 层                       $2 \times (\lceil m/2 \rceil)^2$  个

... ..

第  $H+1$  层                       $2 \times (\lceil m/2 \rceil)^{H-1}$  个

假设  $m$  阶 B-树的深度为  $H+1$ ，由于第  $H+1$  层为叶子结点，而当前树中含有  $N$  个关键字，则叶子结点必为  $N+1$  个，由此可推得下列结果：

$$N+1 \geq 2(\lceil m/2 \rceil)^{H-1}$$

$$H-1 \leq \log_{\lceil m/2 \rceil}((N+1)/2)$$

$$H \leq \log_{\lceil m/2 \rceil}((N+1)/2) + 1$$

## 结论:

在含  $N$  个关键字的 B-树上进行一次查找，需访问的结点个数不超过

$$\log_{\lceil m/2 \rceil}((N+1)/2)+1$$

# 9.3 哈希表

- 一、哈希表是什么？
- 二、哈希函数的构造方法
- 三、处理冲突的方法
- 四、哈希表的查找

# 一、哈希表是什么？

以上两节讨论的查找表的各种结构的共同特点：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，

查找的过程为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次数。

顺序查找时，比较的结果为“=”与“<”两种结果。在折半查找、二叉排序树查找、B-树查找时，比较的结果为“=”、“<”、“>”三种结果。

用这类方法表示的查找表，其平均查找长度都不为零。

不同的表示方法，其差别仅在于：关键字和给定值进行比较的顺序不同。

对于频繁使用的查找表，希望

$$ASL = 0$$

只有一个办法：预先知道所查关键字在表中的位置。

即要求：记录在表中的位置和其关键字之间存在一种确定的对应关系 $f$ 。使得每个**关键字**和结构中的一个**惟一存储位置****相对应**。这种**对应关系 $f$** 为**哈希函数**，按这一思想建立的表称为**哈希表**。

**例如：**为每年招收的 1000 名新生建立一张查找表，其关键字为学号，其值的范围为  $xx000 \sim xx999$  (前两位为年份)。

若以下标为  $000 \sim 999$  的顺序表表示之。

则查找过程可以简单进行：取给定值（学号）的后三位，**不需要经过比较**便可直接从顺序表中找到待查关键字。



但是，对于动态查找表而言，

1) 表长不确定；

2) 在设计查找表时，只知道关键字所属范围，而不知道确切的关键字。

因此在一般情况下，需在关键字与记录在表中的存储位置之间建立一个函数关系，以  $f(\text{key})$  作为关键字为  $\text{key}$  的记录在表中的位置，通常称这个函数  $f(\text{key})$  为哈希函数。

# 哈希表：

在静态查找表和动态查找表中，查找的过程是将给定值依次和集合中各个关键字进行比较，

哈希表中关键字和其存储位置之间存在一种确定的对应关系 $f$ 。使得每个关键字和其存储位置相对应。这种对应关系 $f$ 为哈希函数，按这一思想建立的表称为哈希表。

例如：对于如下 9 个关键字

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dei}

26      17      19      12      23      3      8      25      4

设 哈希函数

$$f(\text{key}) = \lfloor (\text{Ord}(\text{第一个字母}) - \text{Ord}('A') + 1) / 2 \rfloor$$

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

	Chen Dei		Han		Li		Qian	Sun		Wu	Ye	Zhao
--	----------	--	-----	--	----	--	------	-----	--	----	----	------

**问题：**若添加关键字 Zhou , 怎么办？

能否找到另一个哈希函数？

# 从这个例子可见:

1) 哈希函数是一个**映象**，即：将**关键字的集合映射到某个地址集合上**，它的设置很灵活，只要这个**地址集合的大小不超出允许范围**即可；

2) 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，即： $\text{key1} \neq \text{key2}$ ，而  $f(\text{key1}) = f(\text{key2})$ 。

3) 很难找到一个不产生冲突的哈希函数。  
一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。

因此，在构造这种特殊的“查找表”时，除了需要选择一个“好”（尽可能少产生冲突）的哈希函数之外；还需要找到一种“处理冲突”的方法。

## 二、构造哈希函数的方法

对**数字**的关键字可有下列构造方法：

1. 直接定址法

4. 折叠法

2. 数字分析法

5. 除留余数法

3. 平方取中法

6. 随机数法

若是**非数字关键字**，则**需先**对其**进行数字化处理**。

# 1. 直接定址法



哈希函数为关键字的线性函数

$$H(\text{key}) = \text{key} \quad \text{或者}$$

$$H(\text{key}) = a \times \text{key} + b$$

此法仅适合于：

地址集合的大小 == 关键字集合的大小

不同的关键字不会发生冲突，但很少

用此方法。

## 2. 数字分析法



假设关键字集合中的每个关键字都是由  $s$  位数字组成  $(u_1, u_2, \dots, u_s)$ ，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。

此方法仅适合于：

能预先估计出全体关键字的每一位上各种数字出现的频度。



### 3. 平方取中法



以关键字的平方值的中间几位作为存储地址。求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

**此方法适合于：**

关键字中的每一位都有某些数字重复出现频度很高的现象。

## 4. 折叠法



将关键字分割成若干部分，然后取它们的叠加和为哈希地址。有两种叠加处理的方法：**移位叠加**(将分割后的每一部分的最低位对齐相加)和**间界叠加**(从一端向另一端沿分割符来回折叠，对齐相加)。

**此方法适合于：**

关键字的数字位数特别多。

## 5. 除留余数法



设定哈希函数为：

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

其中，  $p \leq m$  (表长) 并且

$p$  应为不大于  $m$  的素数

或是

不含 20 以下的质因子

为什么要对  $p$  加限制？

例如：

给定一组关键字为：12, 39, 18, 24, 33, 21,

若取  $p=9$ , 则他们对应的哈希函数值将为：

3, 3, 0, 6, 6, 3

可见，若  $p$  中含质因子 3，则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

## 6.随机数法



设定哈希函数为：

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

其中，**Random** 为伪随机函数

此方法适用于：

对长度不等的关键字构造哈希函数。

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。

### 三、处理冲突的方法

“冲突”是指由关键字得到的哈希地址上已有记录。

“处理冲突”的实际含义是：

为产生冲突的地址寻找下一个哈希地址。

1. 开放定址法

3. 再哈希法

2. 链地址法

4. 建立公共溢出区

# 1. 开放定址法

为产生冲突的地址  $H(\text{key})$  求得一个地址序列:

$$H_0, H_1, H_2, \dots, H_s \quad 1 \leq s \leq m-1$$

其中:  $H_0 = H(\text{key})$

$$H_i = ( H(\text{key}) + d_i ) \text{MOD } m$$

$$i=1, 2, \dots, s$$

$d_i$  为增量序列

$m$  为哈希表表长



对增量  $d_i$  有三种取法:

1) 线性探测再散列

$d_i = c \times i \quad i = 1, 2, 3, \dots, m-1$  最简单的情况  $c=1$

2) 平方探测再散列 (二次探测再散列)

$d_i = 1^2, -1^2, 2^2, -2^2, \dots, +k^2, -k^2$

3) 随机探测再散列

$d_i$  是一组伪随机数列 或者

$d_i = i \times H_2(key)$  (又称双散列函数探测)

例如：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数  $H(\text{key}) = \text{key} \bmod 11$  (表长=11)

若采用线性探测再散列处理冲突 (ASL=22/9)

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

若采用二次探测再散列处理冲突 (ASL=16/9)

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11
1	1	2	1	2	1	4		1		3

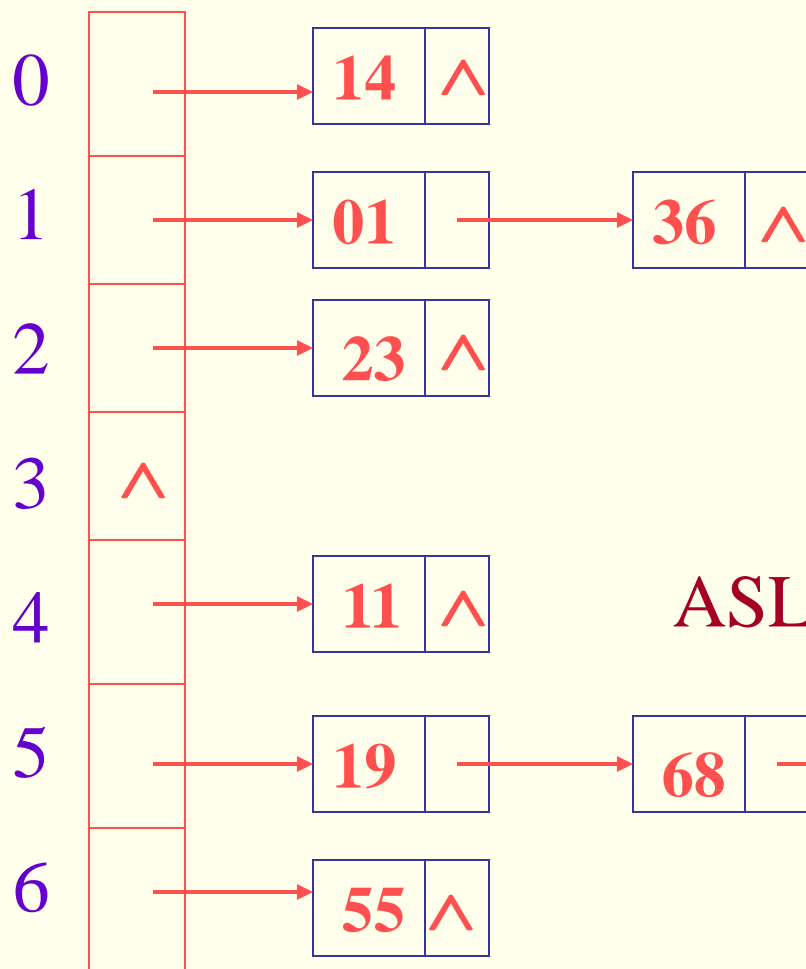
**注意：**增量  $d_i$  应具有“完备性”

即：产生的  $H_i$  均不相同，且所产生的  $s(m-1)$  个  $H_i$  值能覆盖哈希表中所有地址。则要求：

※ 平方探测时的表长  $m$  必为形如  $4j+3$  的素数（如：7, 11, 19, 23, ... 等）；

※ 随机探测时的  $m$  和  $d_i$  没有公因子。

## 2. 链地址法



例如:同前例的关键  
字{ 19, 01, 23, 14, 55,  
68, 11, 82, 36 }, 哈希  
函数为:

$$H(\text{key}) = \text{key} \text{ MOD } 7$$

$$\text{ASL} = (6 \times 1 + 2 \times 2 + 3) / 9 = 13 / 9$$

### 3、再哈希法

有多个不同的哈希函数，当产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生。这种方法不易产生聚集，但增加计算时间。

### 4、建立一个公共溢出区

设向量 $\text{HashTable}[0..m-1]$ 为基本表，存放记录。另设向量 $\text{OverTable}[0..v]$ 为溢出表。只要发生冲突，不管哈希地址是什么，都填入溢出表。

## 四、哈希表的查找

查找过程和造表过程一致。假设采用开放定址处理冲突，则**查找过程**为：

对于给定值  $K$ ， 计算哈希地址  $i = H(K)$

若  $r[i] = \text{NULL}$  则查找不成功

若  $r[i].\text{key} = K$  则查找成功

否则 “求下一地址  $H_i$ ”， 直至

$r[H_i] = \text{NULL}$  (查找不成功)

或  $r[H_i].\text{key} = K$  (查找成功) 为止。

//--- 开放定址哈希表的存储结构 ---

```
int hashsize[] = { 997, ... };
```

// 哈希表容量递增表，一个合适的素数序列

```
typedef struct {
```

```
    ElemType *elem;    // 数据元素存储基址,动态分配数组
```

```
    int count;          // 当前数据元素个数
```

```
    int sizeindex;      // hashsize[sizeindex]为当前容量
```

```
} HashTable;
```

```
#define SUCCESS 1
```

```
#define UNSUCCESS 0
```

```
#define DUPLICATE -1
```

```
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的记录
```

```
    p = Hash(K); // p为待查数据在表中的位置，求得哈希地址
```

```
    while ( H.elem[p].key != NULLKEY &&  
           !EQ(K, H.elem[p].key) )
```

```
        collision(p, ++c); // 求得下一探查地址 p, c冲突次数
```

```
    if (EQ(K, H.elem[p].key)) return SUCCESS;
```

```
        // 查找成功，返回待查数据元素位置 p
```

```
    else return UNSUCCESS;           // 查找不成功
```

```
} // SearchHash
```



```
Status InsertHash (HashTable &H, Elemtyp e){  
    c = 0;           // c用以计冲突次数，其初值置零  
  
    if ( HashSearch ( 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;  
    }  
    // 插入 e  
  
    else RecreateHashTable(H);      // 重建哈希表  
}  
// InsertHash
```

# 哈希表查找的分析：

从查找过程得知，哈希表查找的平均查找长度**实际上并不等于零**。平均查找长度ASL用给定值与关键字的比较次数作为度量。

## 决定哈希表ASL(或决定比较次数)的因素：

- 1) 选用的**哈希函数**；
- 2) 选用的**处理冲突的方法**；
- 3) 哈希表饱和的程度，**装载因子**

$\alpha = n/m$  值的**大小**（ $n$ —记录数， $m$ —表的长度）

一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论ASL时，可以不考虑它的因素。

因此，哈希表的ASL是处理冲突方法和装载因子的函数。相同的哈希函数，不同的处理冲突方法得到的哈希表不同，ASL也不同。

例如：前述例子

线性探测处理冲突时，  $ASL = 22/9$

二次探测处理冲突时，  $ASL = 16/9$

链地址法处理冲突时，  $ASL = 13/9$

可以证明：查找成功时有下列结果：

平均查找长度

**线性探测再散列**

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

**随机探测再散列**

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

**链地址法**

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

从以上结果可见：

哈希表的平均查找长度是  $\alpha$  的函数，而不是  $n$  的函数，依赖装填因子。 $\alpha$  越小，发生冲突的可能性越小。

这说明，用哈希表构造查找表时，可以选择一个适当的装填因子  $\alpha$ ，使得平均查找长度限定在某个范围内。

——这是哈希表所特有的特点。

# 本章学习要点

1. 顺序表和有序表的查找方法及其平均查找长度的计算方法。
2. 熟练掌握二叉排序树的构造和查找方法。
3. 掌握二叉平衡树的构造方法
4. 理解B-树的特点以及它们的建树和查找的过程。

5. 熟练掌握**哈希表**的构造方法，深刻理解哈希表与其它结构的表的实质性的差别。

6. 掌握按定义计算各种查找方法在等概率情况下查找成功时的平均查找长度。

# 作业： 9.9 9.14 9.19 9.20 9.21

9.9 已知如下所示长度为12的表

(Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec)

(1) 试按表中元素的顺序依次插入一棵初始为空的二叉排序树，画出插入完成后的二叉排序树，并求其在等概率的情况下查找成功的平均查找长度。

(2) 若对表中元素先进行排序构成有序表，求在等概率情况下对此有序表进行折半查找时查找成功的平均查找长度。

(3) 按表中元素顺序构造一棵平衡二叉排序树，并求其在等概率的情况下查找成功的平均查找长度。



9.14 试从空树开始，画出按以下次序向2-3树即3阶B-树中插入关键码的建树过程：20、30、50、52、60、68、70，如果此后删除50和68，画出每一步执行后2-3树的状态。

9.19 选取哈希函数 $H(k)=(3k) \text{ MOD } 11$ 。用开放定址法处理冲突， $d_i=i((7k) \text{ MOD } 10 +1)$  ( $i=1,2,3\dots$ )。试在0—10的散列地址空间中对关键字序列（22、41、53、46、30、13、01、67）构造哈希表，并求等概率情况下查找成功时的平均查找长度。

9.20 试为下列关键字建立一个装载因子不小于0.75的哈希表，并计算你所构造的哈希表的平均查找长度。

（ZHAO、QIAN、SUN、LI、ZHOU、WU、ZHANG、WANG、CHANG、CHAO、YANG、JIN）

9.21 在地址空间为0—16的散列区中，对以下关键字序列构造两个哈希表：

(Jan,Feb,Mar,Apr,May,June,July,Aug,Sep,Oct,Nov,Dec)

(1) 用线性探测开放定址法处理冲突

(2) 用链地址法处理

并分别求这两个哈希表在等概率情况下查找成功和不成功时的平均查找长度。设哈希函数为 $H(x) = \lfloor i/2 \rfloor$ ，其中 $i$ 为关键字中第一个字母在字母表中的序号。