数据结构

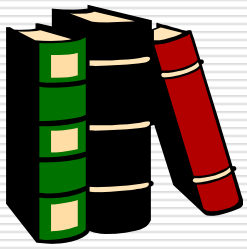
第七章　查找

人工智能学院

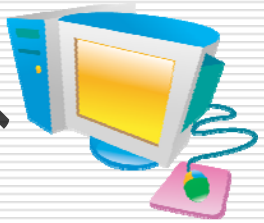
刘运

本章主要内容

- 7.1 查找的基本概念
- 7.2 线性表的查找
- 7.3 树表的查找
- 7.4 散列表的查找
- 7.5 小结

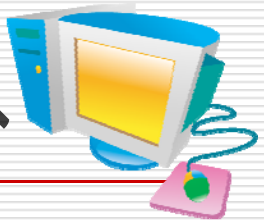


学习要点



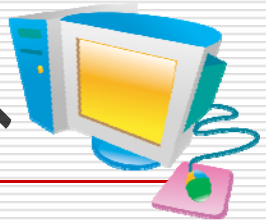
- 折半查找的过程及性能分析；
- 二叉排序树的构造及查找方法；
- 平衡二叉树的调整方法；
- 散列表的构造和查找方法；
- 各种查找技术的时间性能及对比。

7.1 查找的基本概念



- 查找表：是由同一类型的数据元素（或记录）构成的集合。
- 关键字：是数据元素（或记录）中某个数据项的值，用以标识（识别）一个数据元素（或记录）。

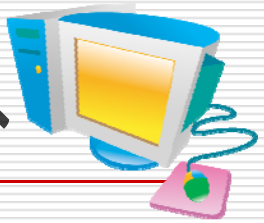
若此关键字可以识别唯一的一个记录，则称之为“主关键字”。若此关键字能识别若干记录，则称之为“次关键字”。



- 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；

若表中不存在关键字等于给定值的记录，则称“查找不成功”，查找结果：给出“空记录”或“空指针”。



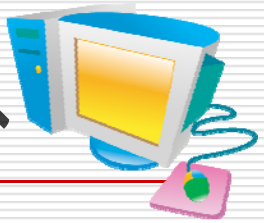
对查找表经常进行的操作：

（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）从查找表中删除某个数据元素。



- **静态查找表**

仅作查询和检索操作的查找表。

- **动态查找表**

在查找的同时对表做修改操作（如插入和删除）。

查找算法的评价指标

- 平均查找长度*ASL*(*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个数的期望值。

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

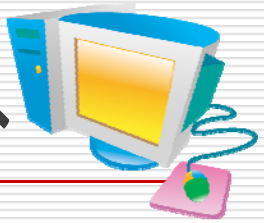
其中：

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，和给定值比较过的关键字的个数

7.2 线性表的查找



7.2.1 顺序查找（Sequential Search）

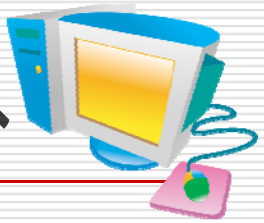
顺序查找既适用于线性表的顺序存储结构，也适用于线性表的链式存储结构。

顺序表存储结构

```
typedef  struct {
   ElemType  *R;   // 存储空间基址，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;
```



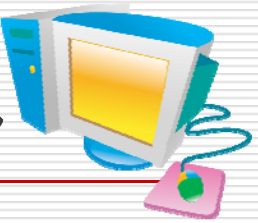
已知如下11个数据元素的线性表(关键字即为数据元素的值)：

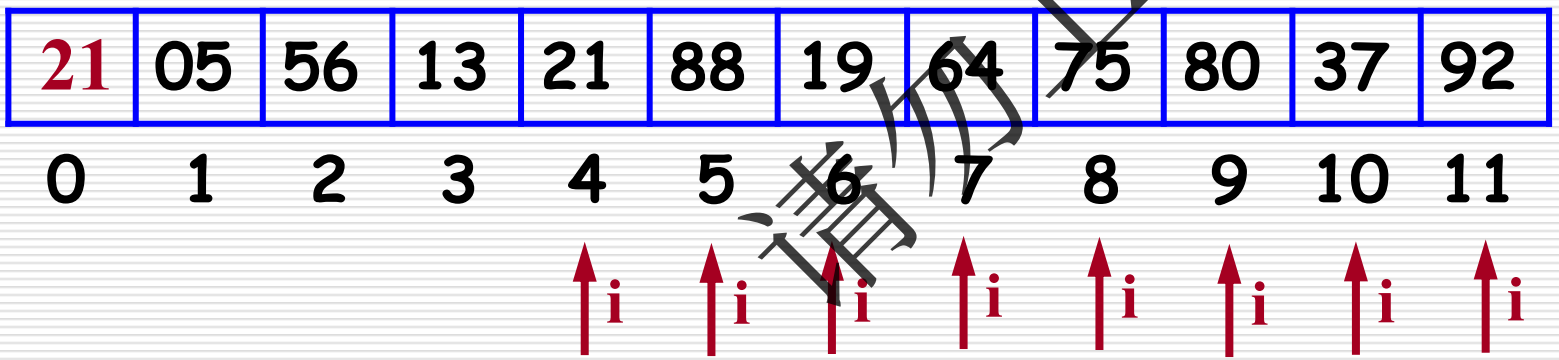
{ 05, 56, 13, 21, 88,19, 64, 75, 80, 37, 92 }

现要查找关键字为21和85的数据元素。

最简单的方法----
顺序查找！

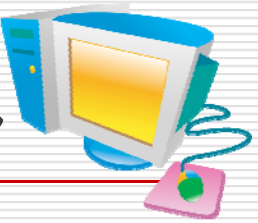
顺序查找key=21

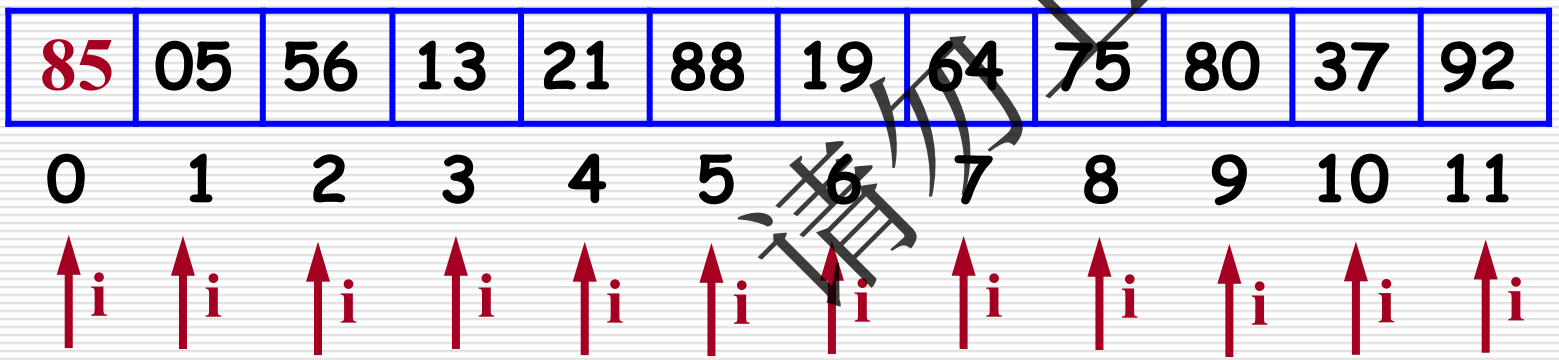




查找成功时的比较次数 $C_i = n-i+1$

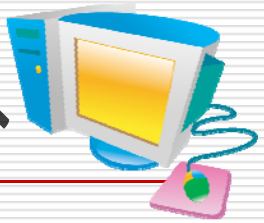
顺序查找key=85





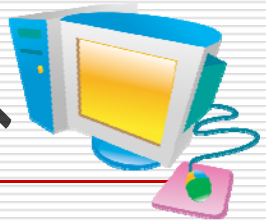
查找不成功时的比较次数 $n+1$

顺序查找算法：



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

顺序查找的时间性能分析



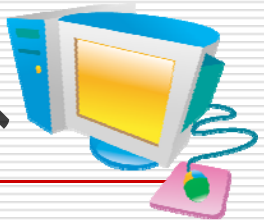
$$ASL = \sum_{i=1}^{n} P_i C_i$$

对顺序表而言，查找成功时 $C_i = n-i+1$

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：

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



顺序表查找的优缺点：

优：算法简单且适用面广，对表的结构无要求

缺：平均查找长度较大，表长较大时效率很低

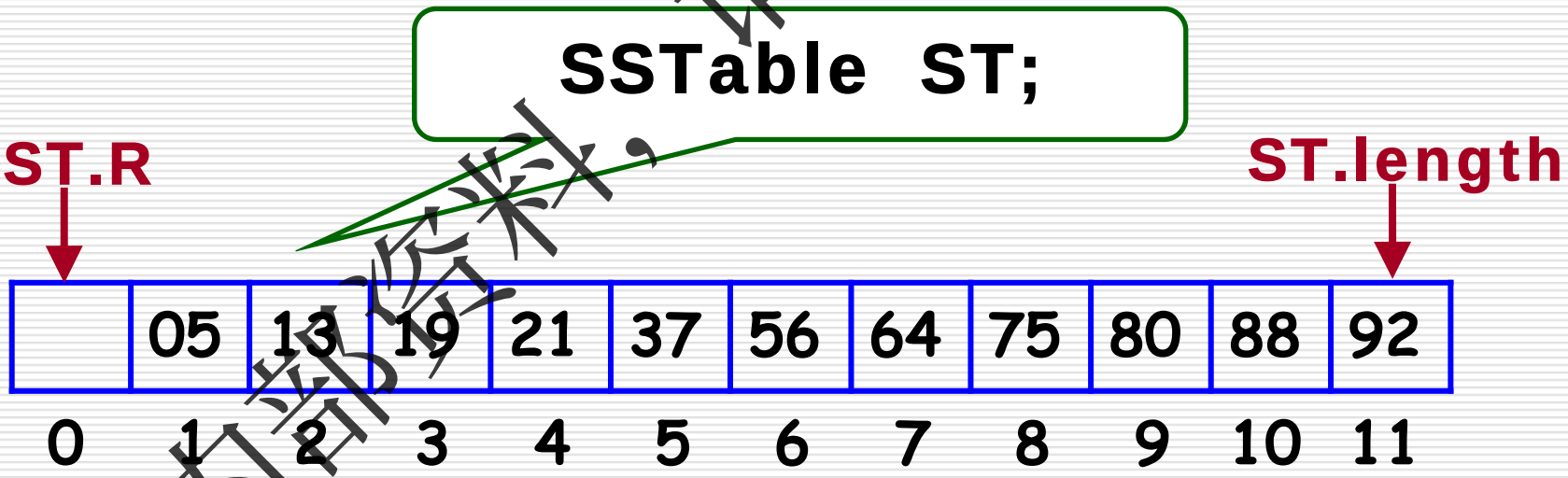
有没有更好的查找办法？

7.2.2 折半查找（Binary Search）

- 折半查找的条件

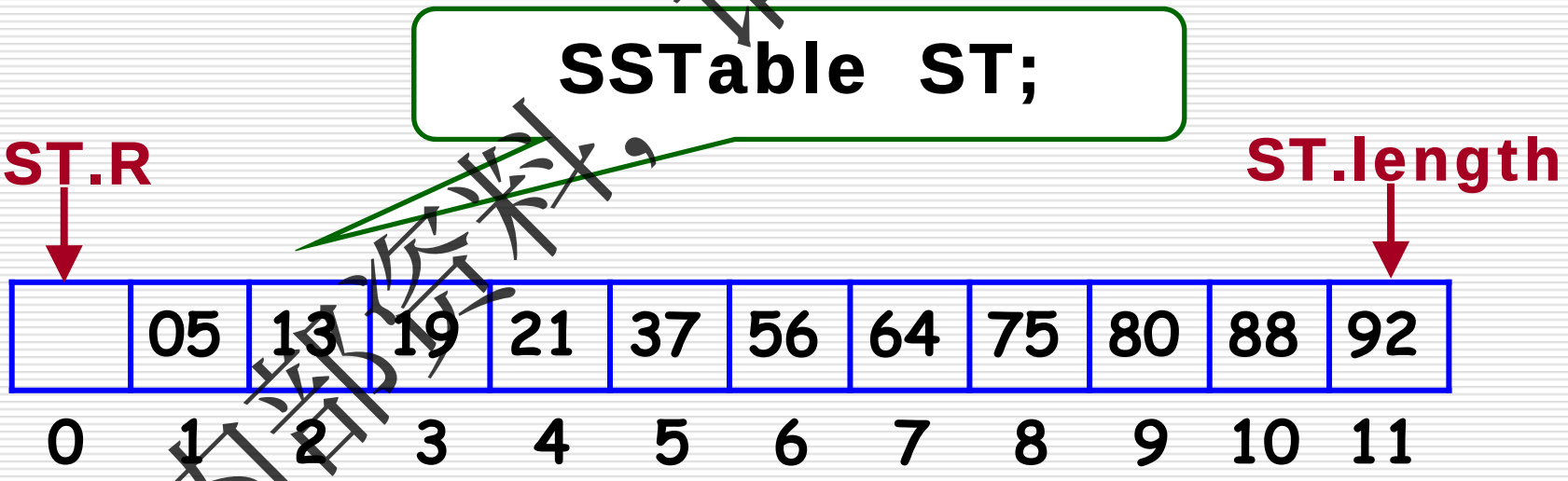
  (1) 表中元素按关键字有序排列；

  (2) 顺序存储。

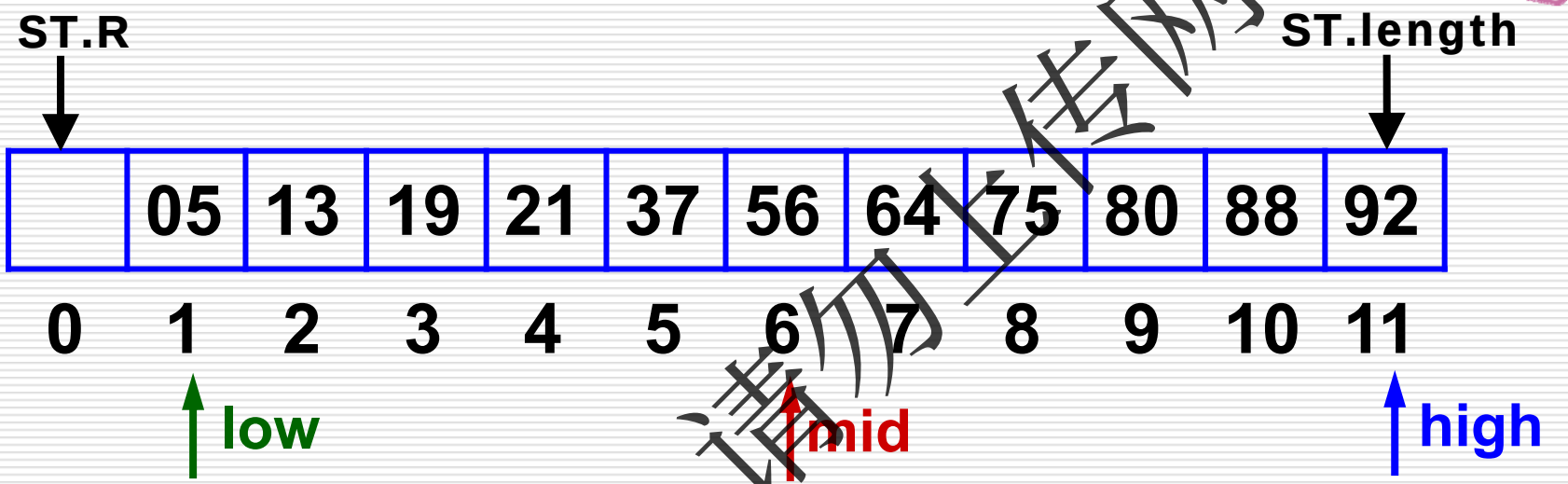


- 折半查找的思想

取中，比较



key=21的查找过程



low指示查找区间的下界；

high指示查找区间的上界；

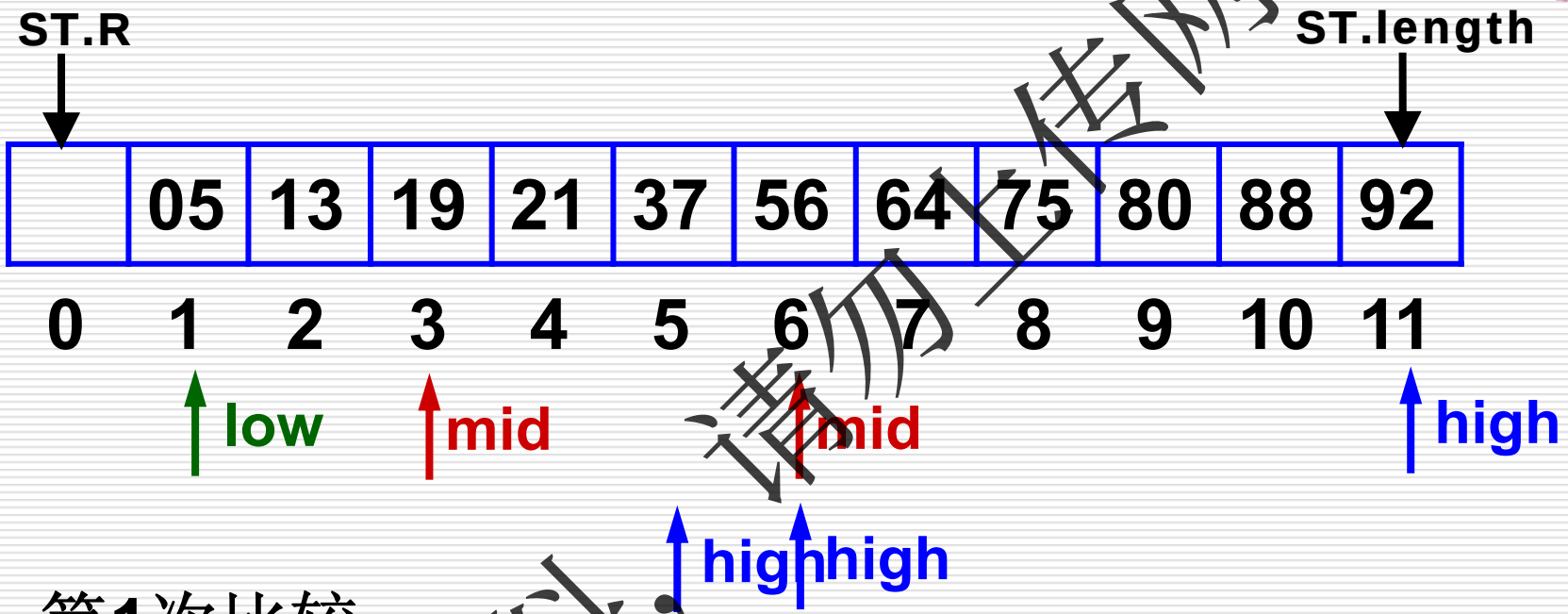
mid指示区间的中间位置，
mid=(low+high)/2

low=1

high=11

mid=(1+11)/2=6

key=21的查找过程



第1次比较：

key < ST.R[mid].key

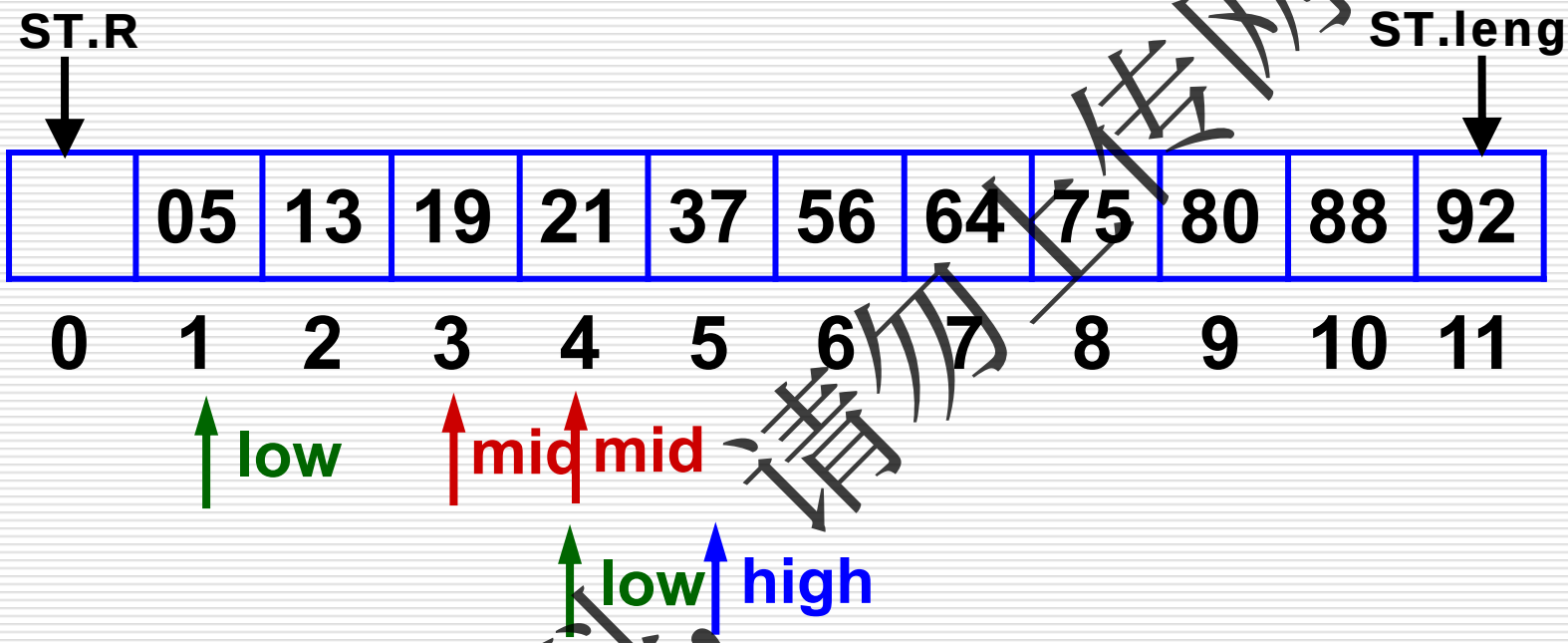
high=mid -1;

low=1

high=5

mid=(1+5)/2=3

key=21的查找过程



第2次比较：

key > ST.R[mid].key

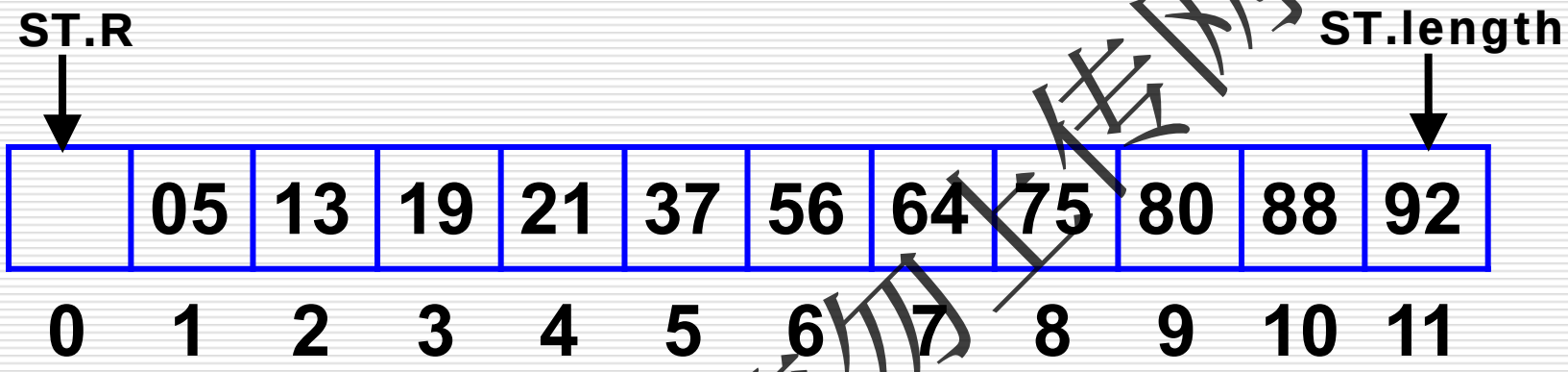
low=mid+1;

low=4

high=5

mid=(4+5)/2=4

key=21的查找过程



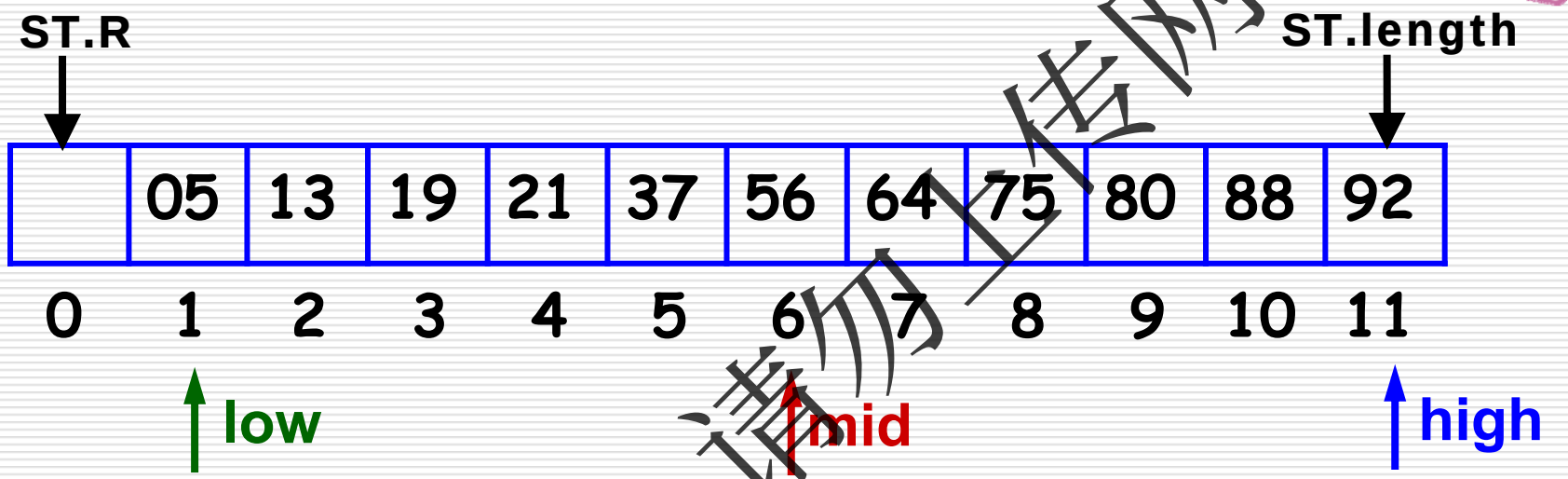
查找成功！

第3次比较：

key == ST.R[mid].key

return(mid);

key=85的查找过程



low指示查找区间的下界；

high指示查找区间的上界；

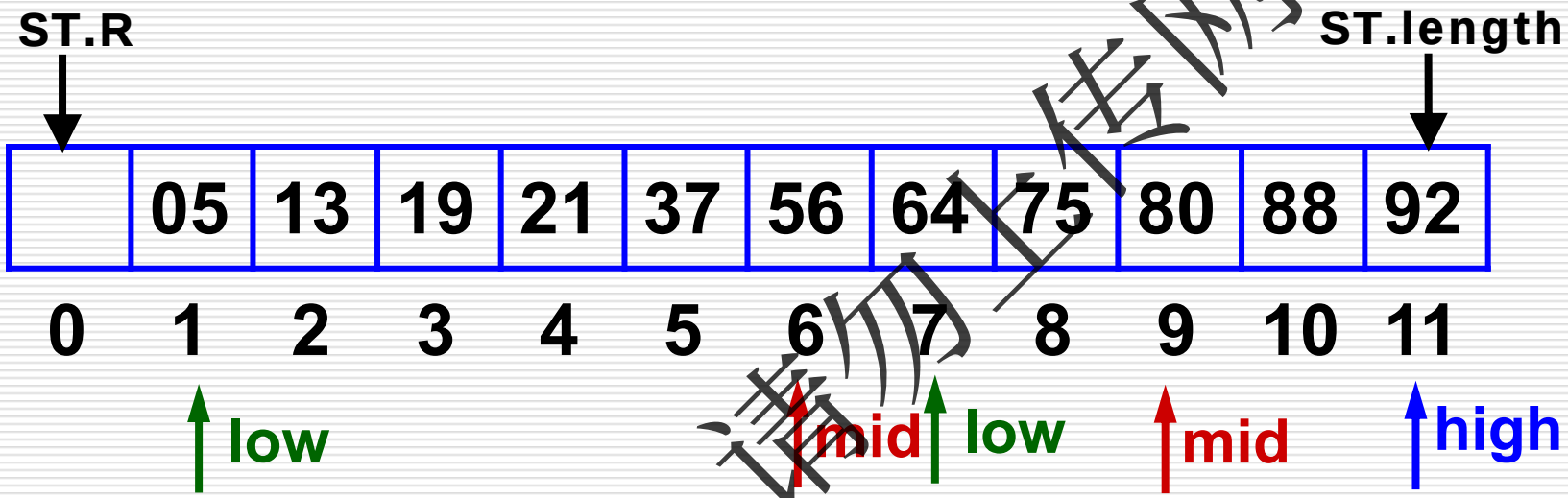
mid=(low+high)/2 。

low=1

high=11

mid=(1+11)/2=6

key=85的查找过程



第1次比较：

key > ST.R[mid].key

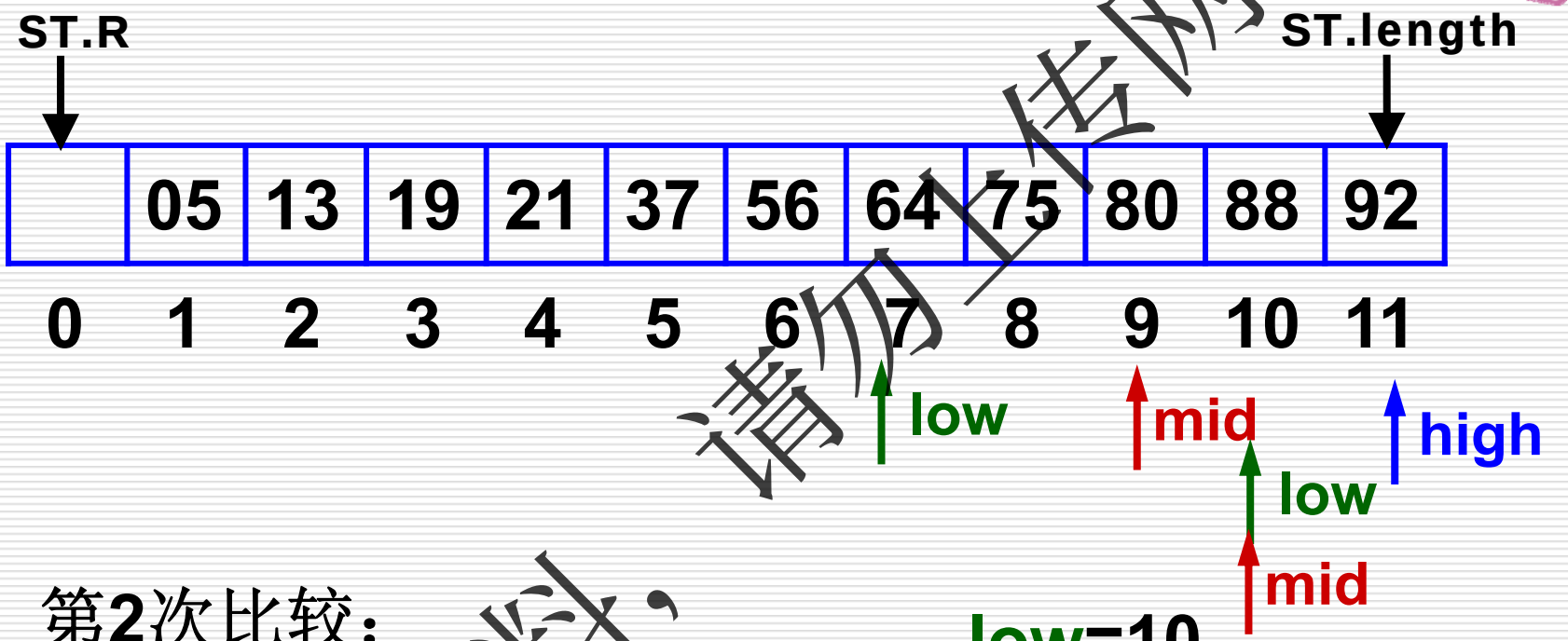
low=mid+1;

low=7

high=11

mid=(7+11)/2=9

key=85的查找过程



第2次比较：

key > ST.R[mid].key

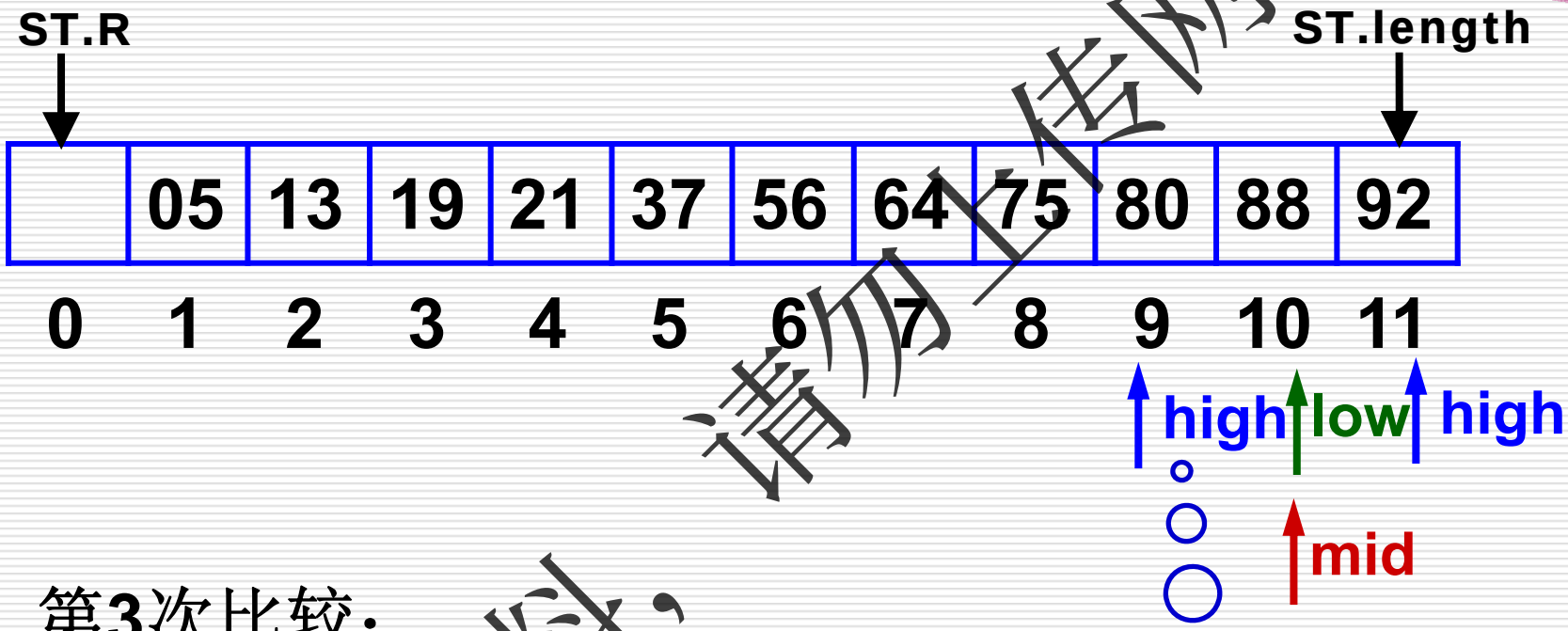
low=mid+1;

low=10

high=11

mid=(10+11)/2=10

key=85的查找过程



第3次比较：

key < ST.R[mid].key

high=mid-1;

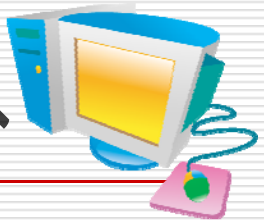
low>high

查找失败！

折半查找算法实现


```
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(key==ST.R[mid].key)
            return(mid);                            // 找到待查元素
      else if(key<ST.R[mid].key)
         high=mid-1;   //继续在当前区间的前半区间进行查找
          else
         low=mid+1;   //继续在当前区间的后半区间进行查找
   }
   return 0;                                        //顺序表中不存在待查元素
}// Search_Bin
```

思考：

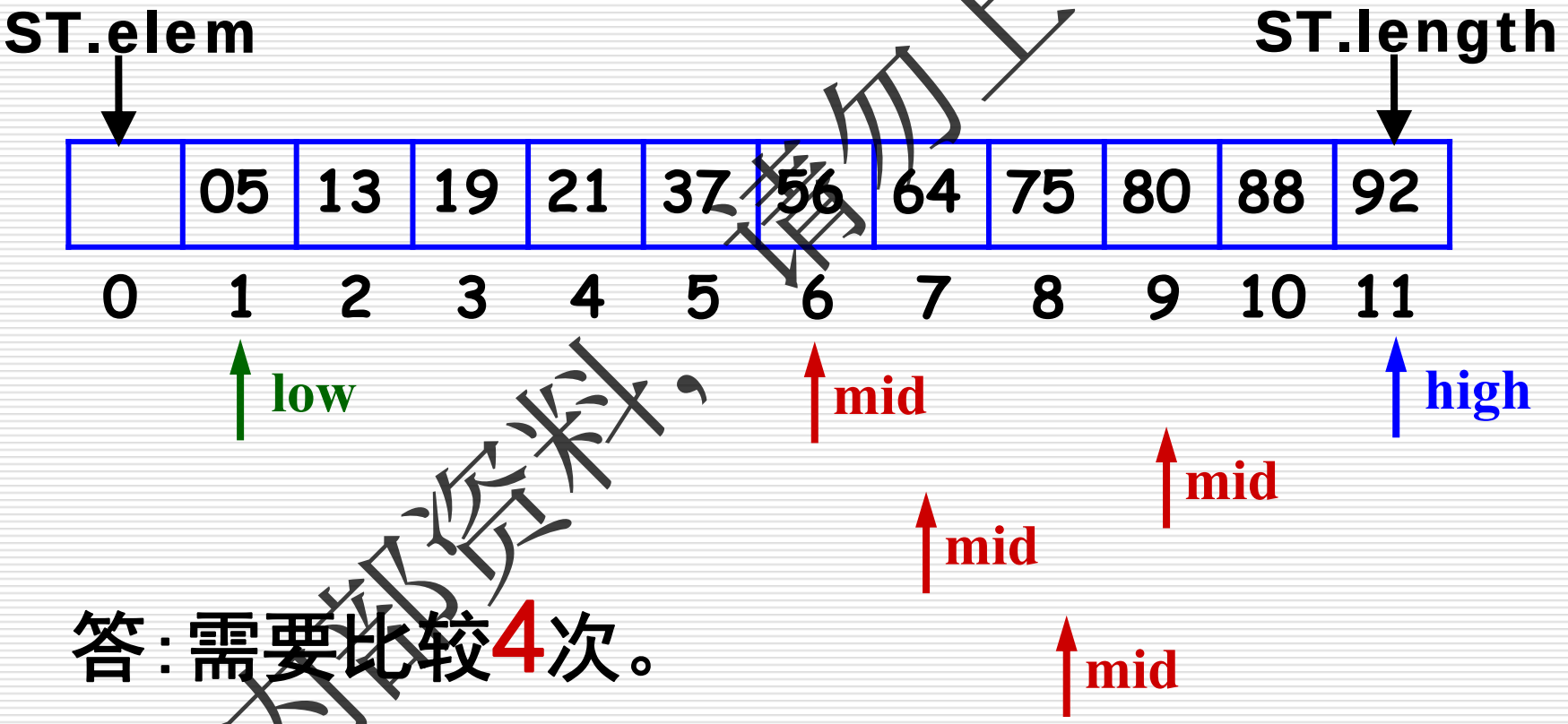


若采用链表（单链表或双向链表）作为查找表的存储结构，能否进行折半查找？

不能！

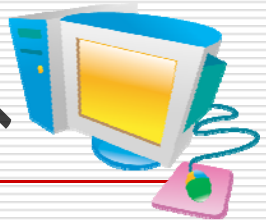
例1：

假设在一个长度为11的有序顺序表中进行折半查找，查找第8个元素需要比较多少次？



答：需要比较4次。

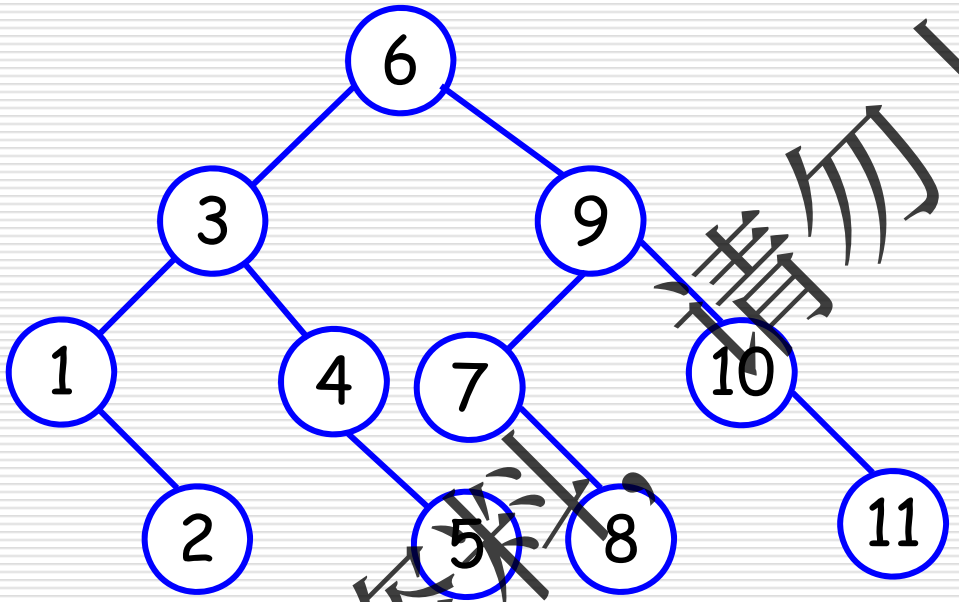
例2：



具有11个元素的有序表进行折半查找时，等概率情况下查找成功时的平均查找长度是多少？

- 判定树：折半查找的查找过程可以用一棵二叉树来描述。其中，树中的每个结点表示一个记录，结点中的值为该记录在表中的位置序号。把当前查找区间的中间位置作为根，左子表和右子表分别作为根的左子树和右子树，由此得到的二叉树称为折半查找的判定树。

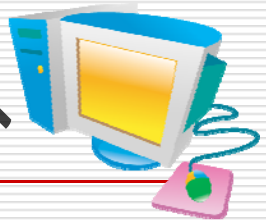
1	2	3	4	5	6	7	8	9	10	11
05	13	19	21	37	56	64	75	80	88	92



查找成功时比较的关键字个数恰为该结点在判定树中的层数。

$$ASL_{bs}=\sum_{i=1}^{n}P_iC_i=\frac{1}{11}(1+2*2+3*4+4*4)=\frac{33}{11}=3$$

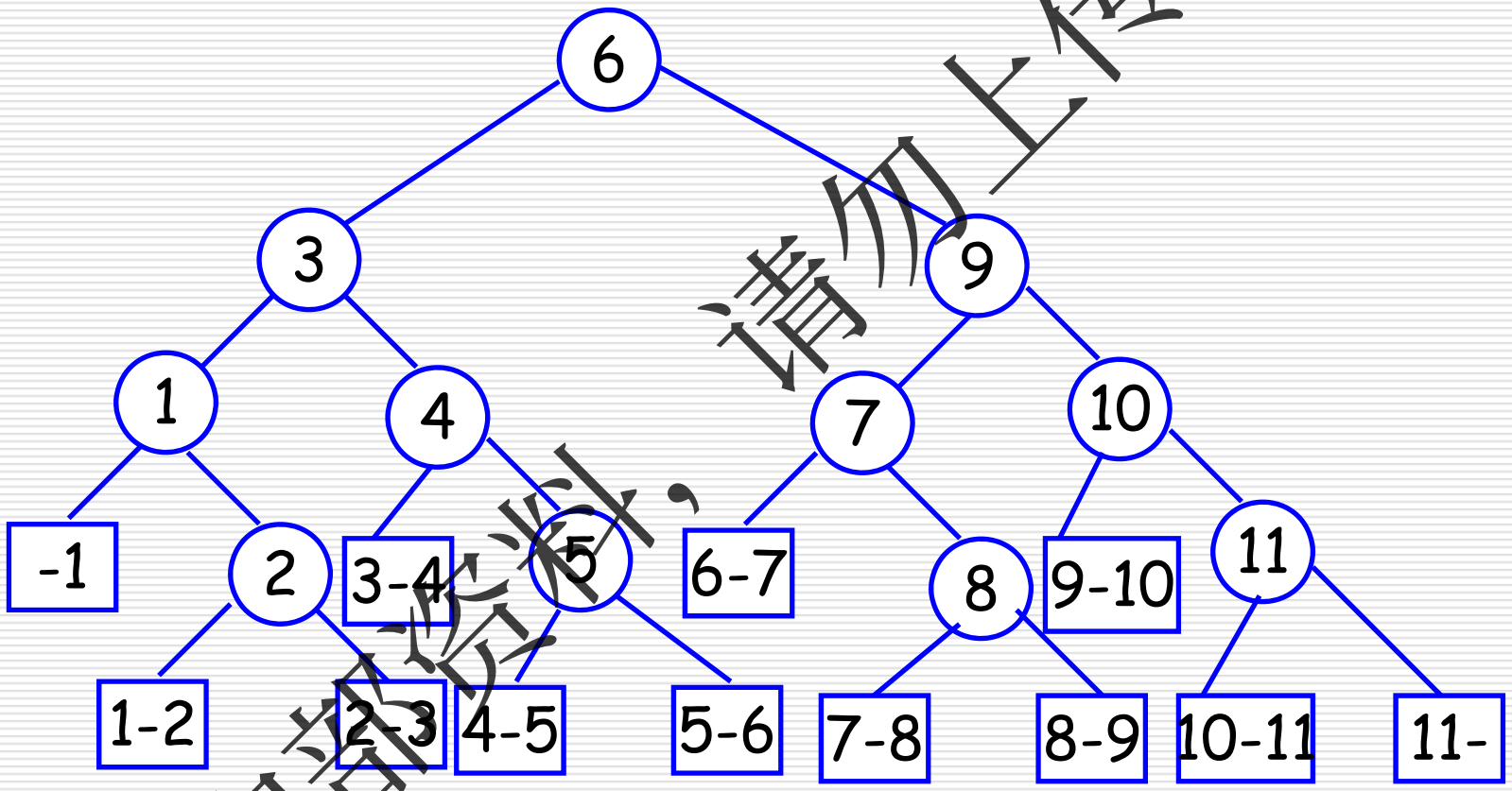
外部结点



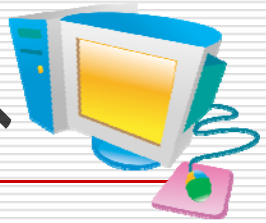
定义：在折半查找的判定树中的所有结点的空指针域上加上方形结点，称这些方形结点为外部结点（与之相对，称那些圆形结点为内部结点）。

折半查找时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，和给定值进行比较的关键字的个数等于路径上内部结点的个数。

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



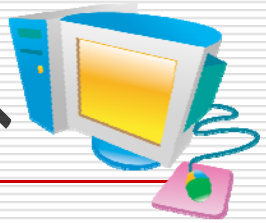
例3：（2010考研试题）



已知一个长度为16的顺序表L，其元素按关键字有序排列，若采用折半查找法查找一个不存在的元素，则比较次数最多的是（ C ）。

A. 4　　B. 5　　C. 6　　D. 7

折半查找性能分析



一般情况下，表长为 n 的折半查找的判定树的深度和含有 n 个结点的完全二叉树的深度相同。

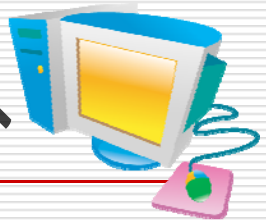
假设 $n=2^h-1$ 并且查找概率相等，则

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

在 n>50 时，可得近似结果

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

小结：顺序查找和折半查找性能对比



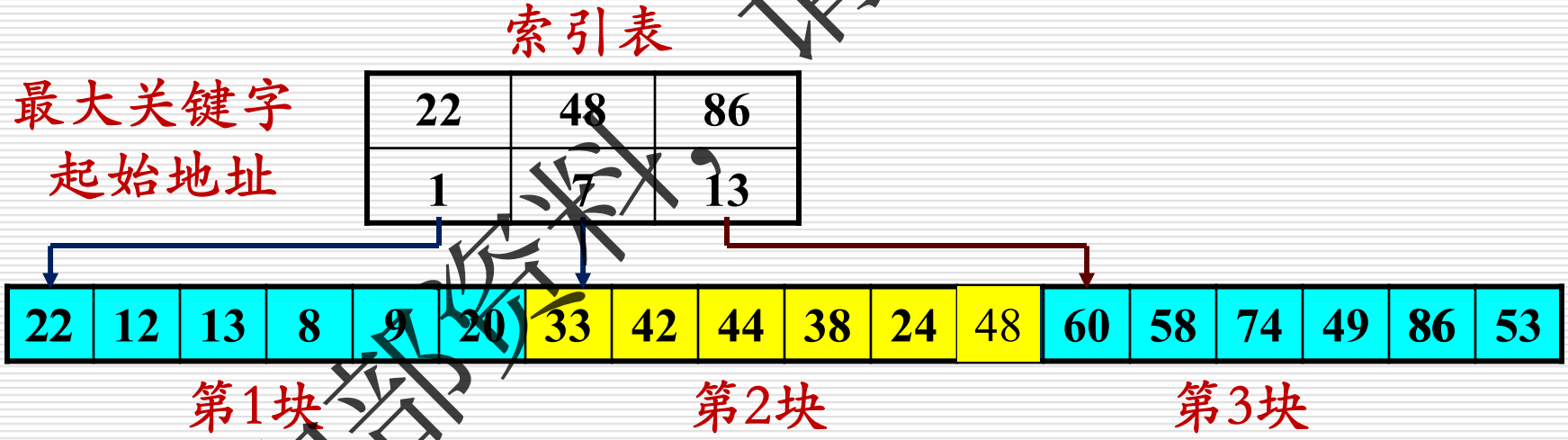
	顺序查找	折半查找
表的特性	无序	有序
存储结构	顺序或链式	顺序
插删操作	易于进行	需移动元素
ASL的值	大	小

7.2.3 分块查找（Blocking Search）

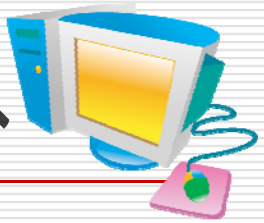
分块查找（块间有序，块内无序）

分块有序，即分成若干子表，要求每个子表中的数值都比后一块中数值小（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）。



索引顺序表的查找过程：



1）由索引确定记录所在区间；

2）在顺序表的某个区间内进行查找。

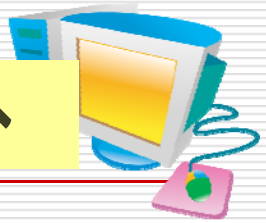
索引顺序查找的过程也是一个“缩小区间”的查找过程。

索引顺序查找的平均查找长度 =

查找“索引”的平均查找长度

+ 查找“顺序表”的平均查找长度

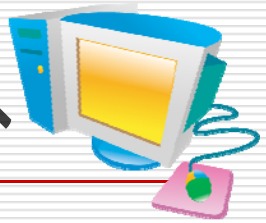
分块查找过程



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）。

分块查找性能分析



查找效率：$ASL=L_b+L_w$

对索引表查找的ASL

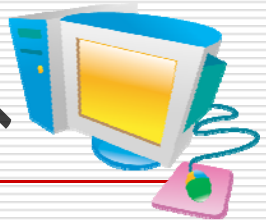
对块内查找的ASL

$$ASL_{bs} = \frac{b+1}{2} + \frac{s+1}{2}$$

S为每块内部的记录个数，n/s即块的数目

顺序查找确定所在块

分块查找性能分析



查找效率：$ASL=L_b+L_w$

对索引表查找的ASL

对块内查找的ASL

$$ASL_{bs} \cong \log_2(\frac{n}{s}+1)+\frac{s}{2}$$

S为每块内部的记录个数，n/s即块的数目

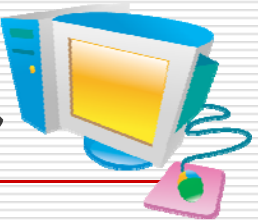
折半查找确定所在块

分块查找优缺点

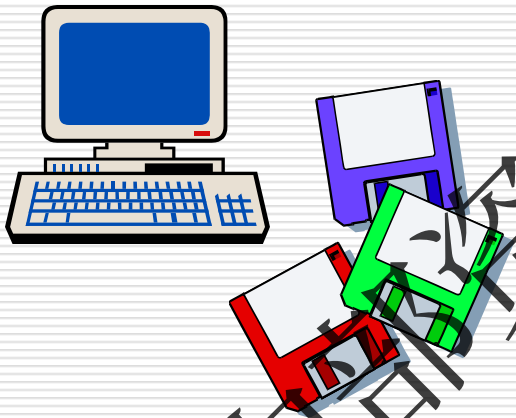
优点：插入和删除比较容易，无需进行大量移动。

缺点：要增加一个索引表的存储空间并对初始索引表进行排序运算。

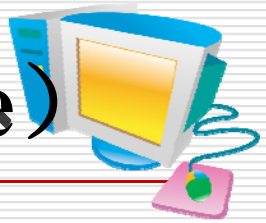
适用情况：如果线性表既要快速查找又经常动态变化，则可采用分块查找。



7.3 树表的查找



7.3.1二叉排序树（Binary Sort Tree）



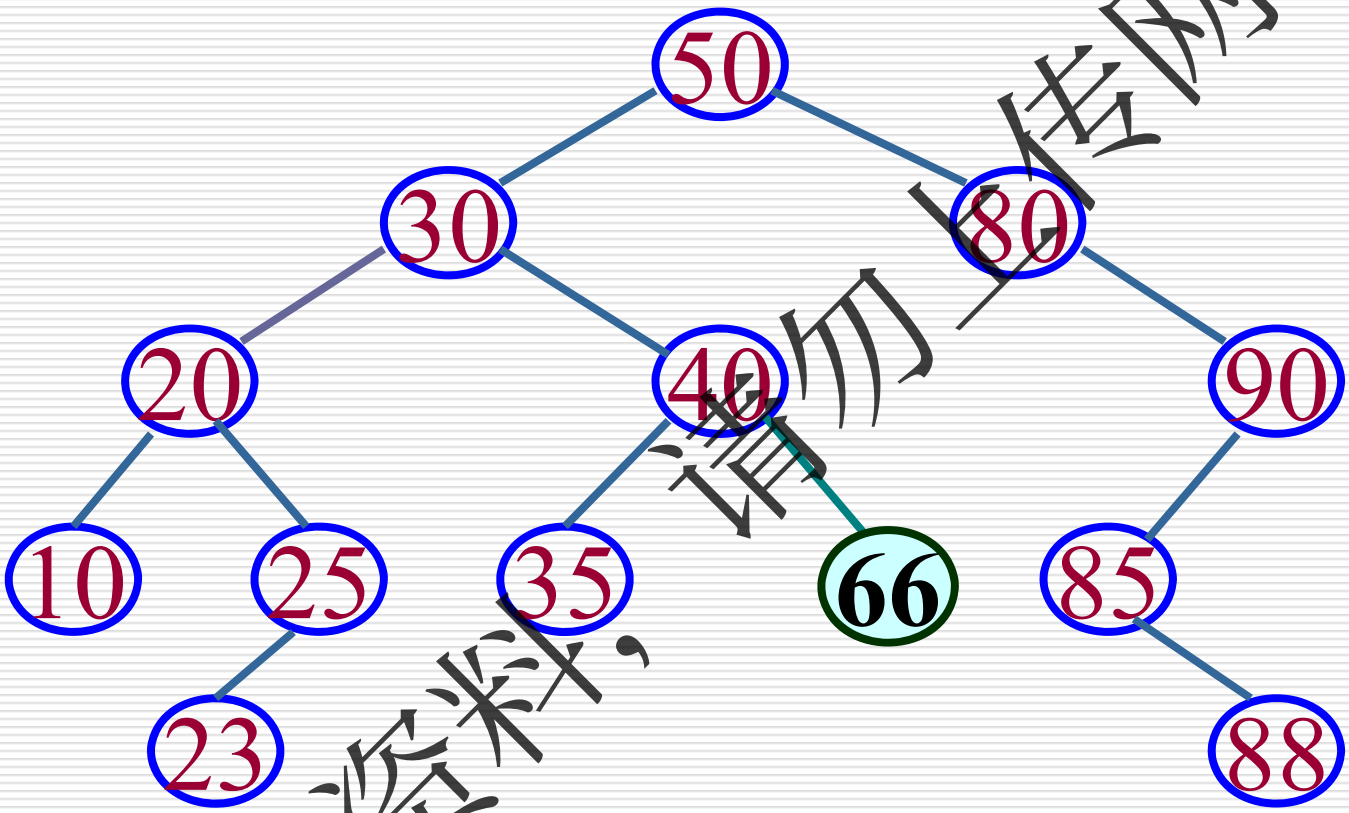
1. 定义：二叉排序树或者是一棵空树；或者是具有下列性质的二叉树：

（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；

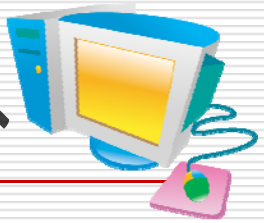
（3）它的左、右子树也都分别是二叉排序树。

例如：



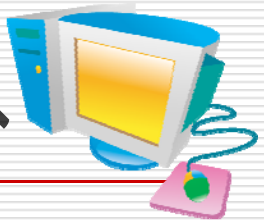
不是二叉排序树。

二叉排序树的存储结构：二叉链表



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

2. 二叉排序树的查找

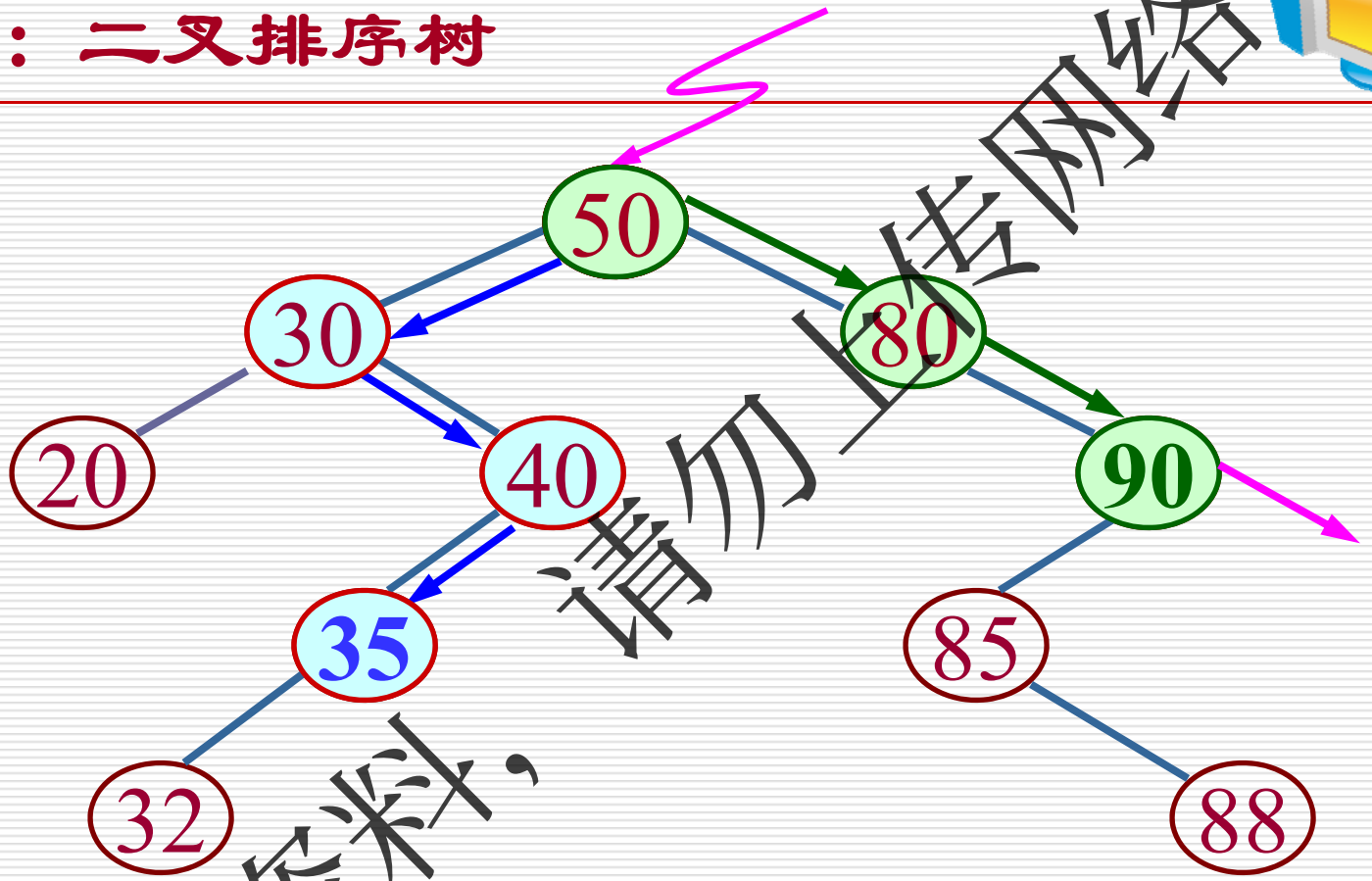


若二叉排序树为空，则查找不成功；

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。

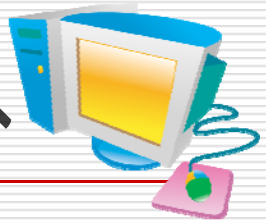
例如：二叉排序树



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；

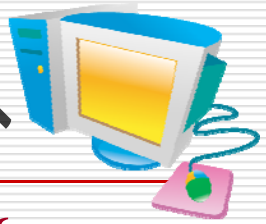
——查找成功

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。

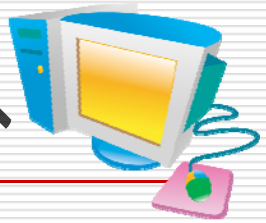
——查找不成功

算法7.4 二叉排序树的查找



```
BSTree SearchBST (BSTree T, KeyType key ) {
  // 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，
  // 则返回指向该数据元素的结点的指针，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));
                                        //在左子树中进行查找
else  return(SearchBST(T->rchile, key));
                                        //在右子树中进行查找
 } // SearchBST
```

3.二叉排序树的插入



算法步骤：

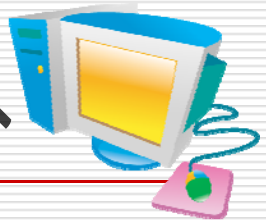
（1）若二叉排序树为空树，则待插入结点*S作为根结点插入到空树中；

（2）若二叉排序树非空，则将key与根结点的关键字T->data.key进行比较：

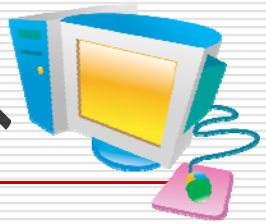
若key小于T->data.key，则将*S插入左子树；

若key大于T->data.key，则将*S插入右子树。

算法7.5 二叉排序树的插入

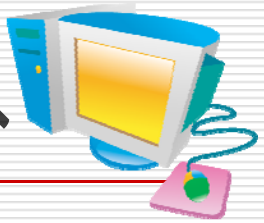


```
void Insert BST(BSTree &T, ElemType e )
{ // 当二叉排序树中不存在关键字等于 e.key 的 数据元素时，则插入该元素
  if(!T)
{ S= new BSTNode;              //生成新结点
   S->data = e;  //新结点的数据域置为*S
   S->lchild = S->rchild = NULL;  //新结点*S作为叶子结点
  T=S; // 把新结点*S 链接到已找到的插入位置
}
```



```
else   if (e.key<T->data.key)
        InsertBST(T->lchild ,e);     // 将*S插入左子树
else   if (e.key>T->data.key)
        InsertBST(T->rchild ,e); // 将*S插入右子树
}
```

4.二叉排序树的创建



算法步骤：

（1）若二叉排序T初始化为空树；

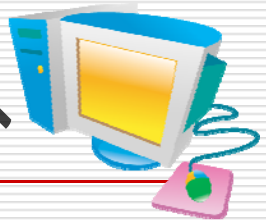
（2）读入一个关键字为key的结点；

（3）若读入的关键字不是结束标志，则循环执行以下操作：

将此结点插入二叉排序树T中；

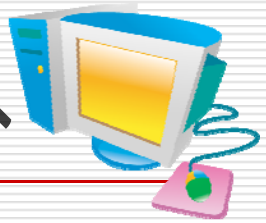
读入一个关键字为key的结点。

算法7.6 二叉排序树的创建



```
void  CreatBST(BSTree &T)
{//依次读入一个关键字为key的结点，将此结点插入二叉排序树T中
    T=NULL;  //将二叉排序树T初始化为空树
    cin>>e;
    while(e.key!=ENDFLAG)
    {  InsertBST(T,e);
       cin>>e;
    }
}
```

二叉排序树的删除算法

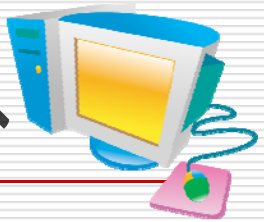


和插入相反，删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。

可分三种情况讨论：

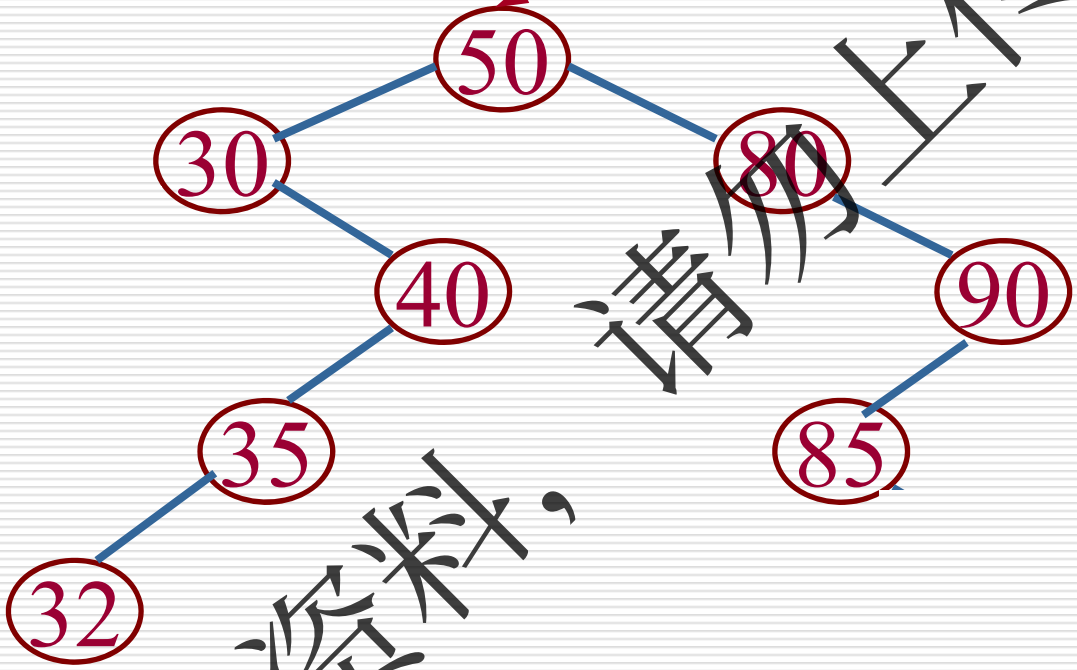
- 被删除的结点是叶子；
- 被删除的结点只有左子树或者只有右树；
- 被删除的结点既有左子树，也有右子树。

（1）被删除的结点是叶子结点



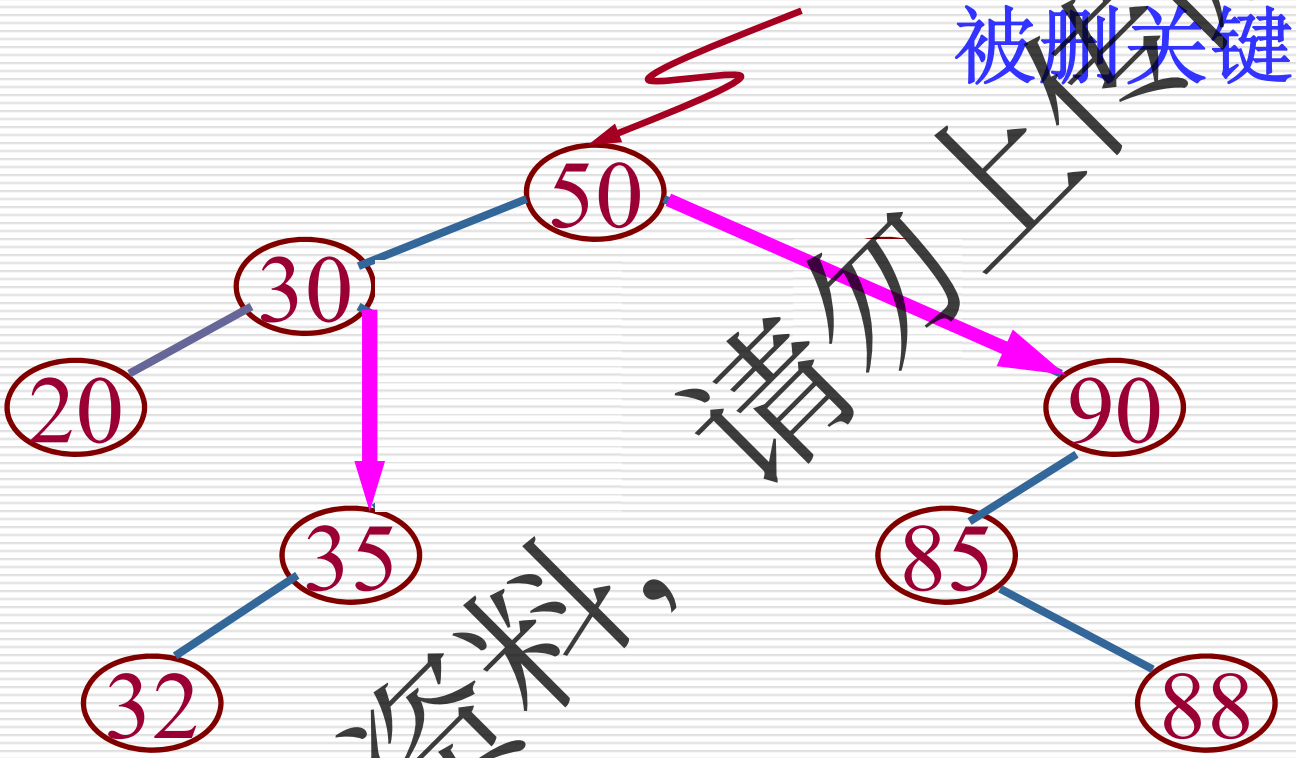
例如：

被删关键字 = **88**



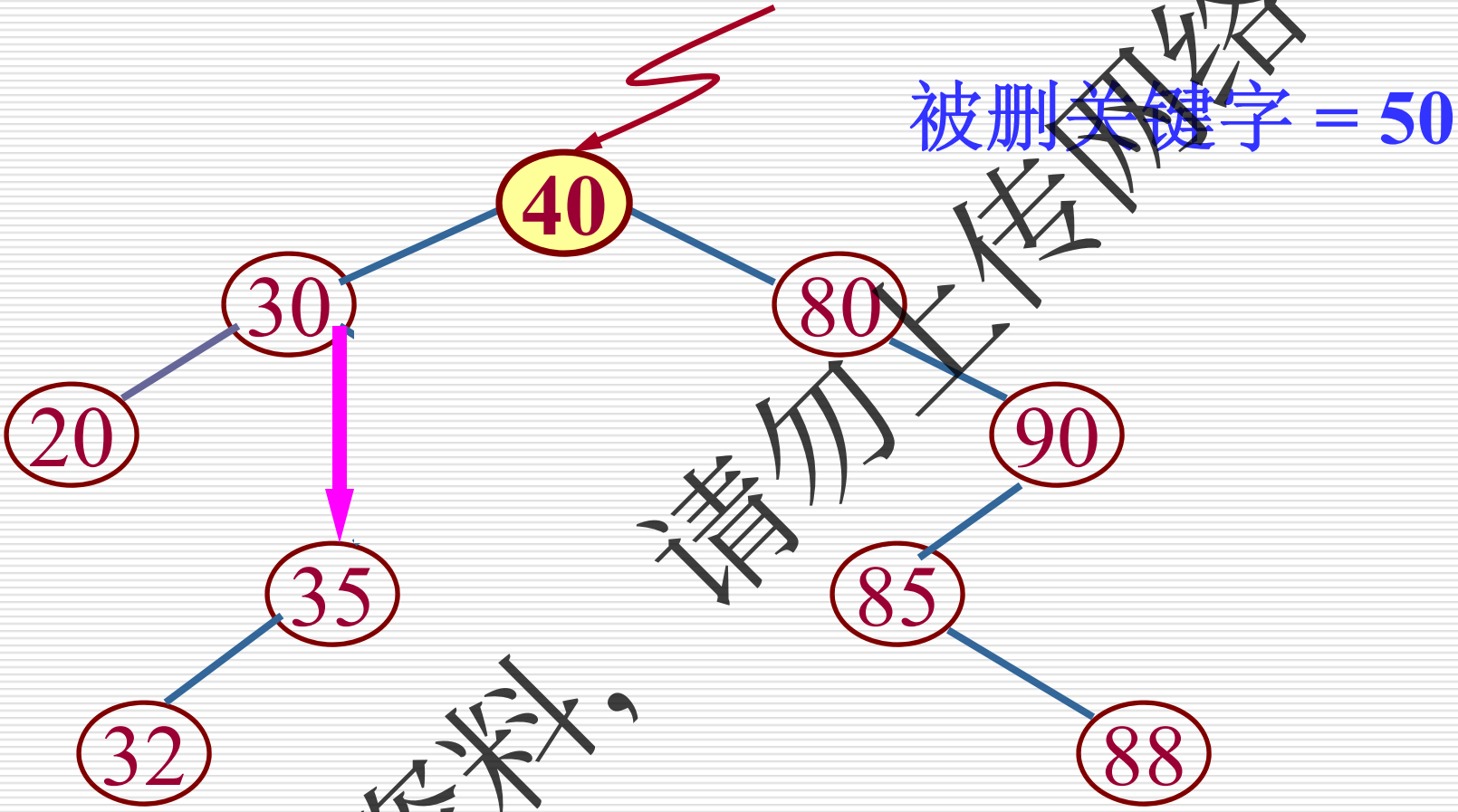
其双亲结点中相应指针域的值改为“空”

（2）被删结点只有左子树或只有右子树



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”。

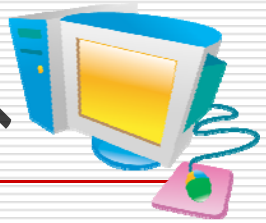
（3）被删除的结点既有左子树，也有右子树



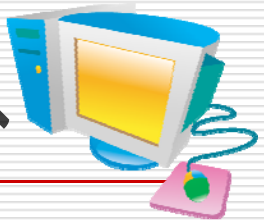
前驱结点　被删结点

以其前驱替代之，然后再删除该前驱结点

删除算法

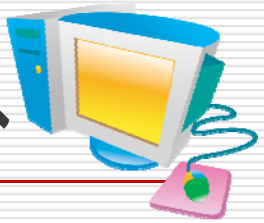


```
Status DeleteBST (BiTree &T, KeyType key) {
  // 若二叉排序树 T 中存在其关键字等于 key的
  // 数据元素，则删除该数据元素结点，并返回
  // 函数值 TRUE，否则返回函数值 FALSE
  if (!T)  return FALSE;
          // 不存在关键字等于key的数据元素
  else {   ……  ……  }
} // DeleteBST
```



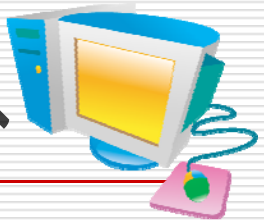
```
if ( key== T->data.key )
      {  Delete (T);   return TRUE; }
          // 找到关键字等于key的数据元素
else if (key< T->data.key )
      return DeleteBST ( T->lchild, key );
                  // 继续在左子树中进行查找
else
      return DeleteBST ( T->rchild, key );
    // 继续在右子树中进行查找
```

其中删除操作过程如下所描述：

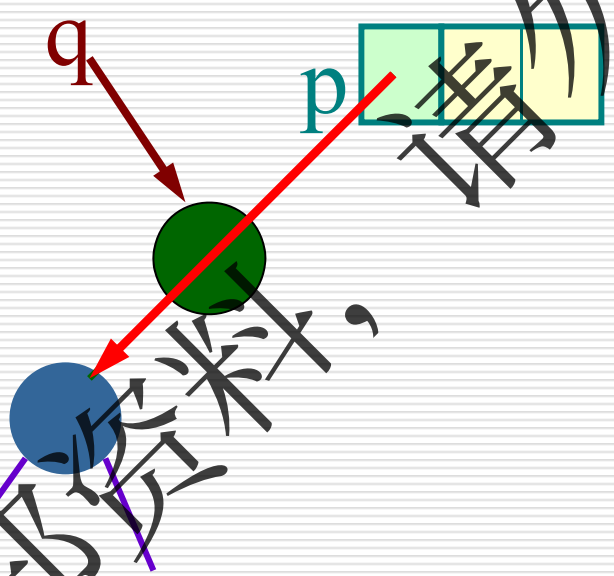


```
void Delete ( BiTree &p ){
  // 从二叉排序树中删除结点 p，
  // 并重接它的左子树或右子树
  if (!p->rchild) {… …  }
  else if (!p->lchild) {… …  }
  else { … … }
} // Delete
```

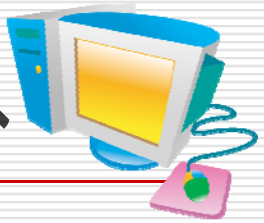
// 右子树为空树则只需重接它的左子树



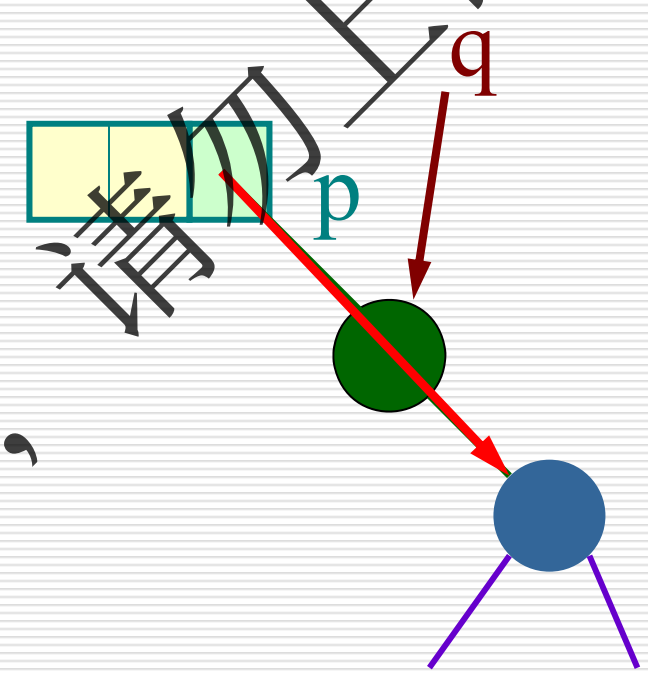
```
q = p;  p = p->lchild; delete q;
```



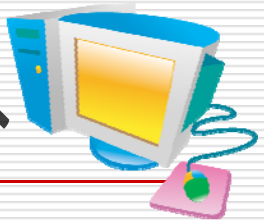
// 左子树为空树只需重接它的右子树



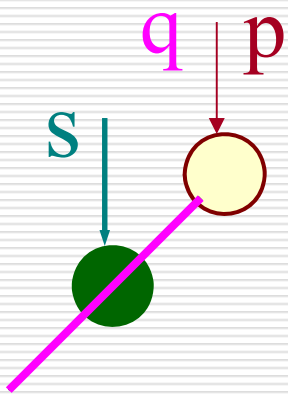
q = p; p = p->rchild; delete q;



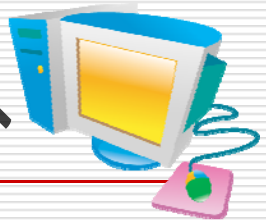
// 左右子树均不空



```
q = p;  s = p->lchild;
while (s->rchild) { q = s;  s = s->rchild; }
                    // s 指向被删结点的前驱
 p->data = s->data;
 if (q != p )  q->rchild = s->lchild;
 else  q->lchild = s->lchild;
                    //重接*q的左子树
 Delete s ;
```

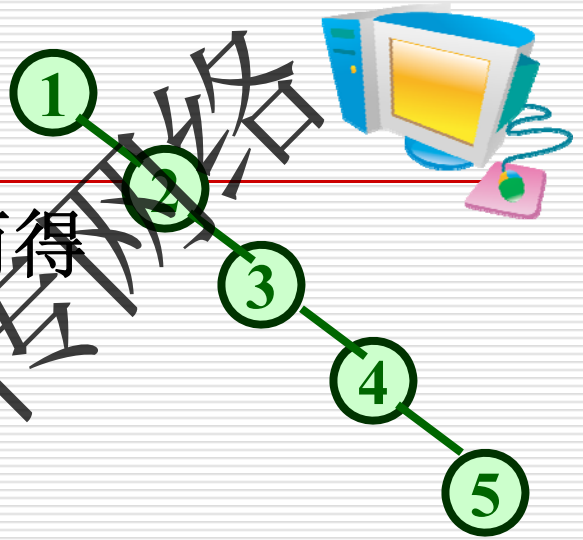


查找性能的分析



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。

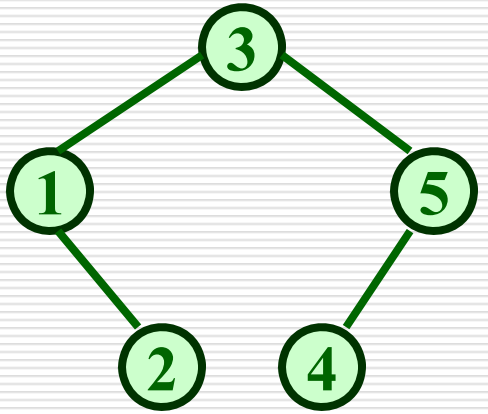
例如：



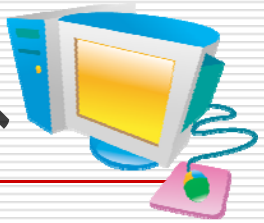
由关键字序列 1，2，3，4，5构造而得的二叉排序树，

ASL＝（1+2+3+4+5）/ 5 ＝ 3

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

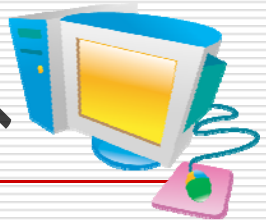


假设 n 个关键字可能出现的 $n!$ 种排列的可能性相同，则含 n 个关键字的二叉排序树的平均查找长度

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

在等概率查找的情况下，

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

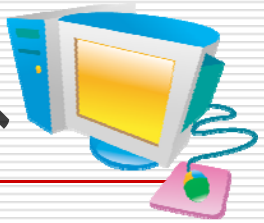


此递归方程有解：

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

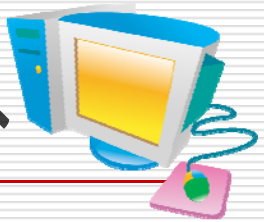
在随机情况下，二叉排序树的平均查找长度和 **logn** 是等数量级的。

7.3.2 平衡二叉树



- 何谓“平衡二叉树”?
  - 如何构造“平衡二叉树”
    - 平衡二叉树的查找性能分析

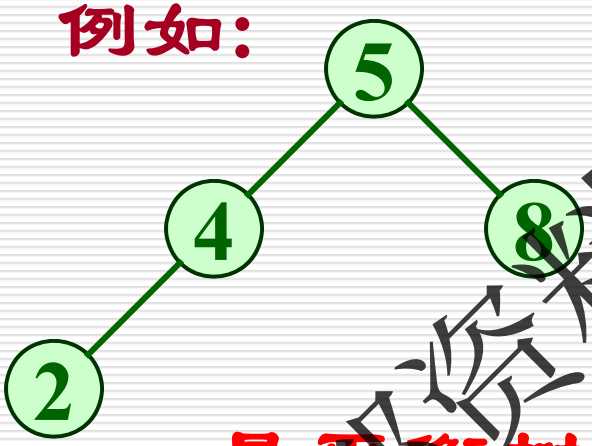
平衡二叉树特点为：



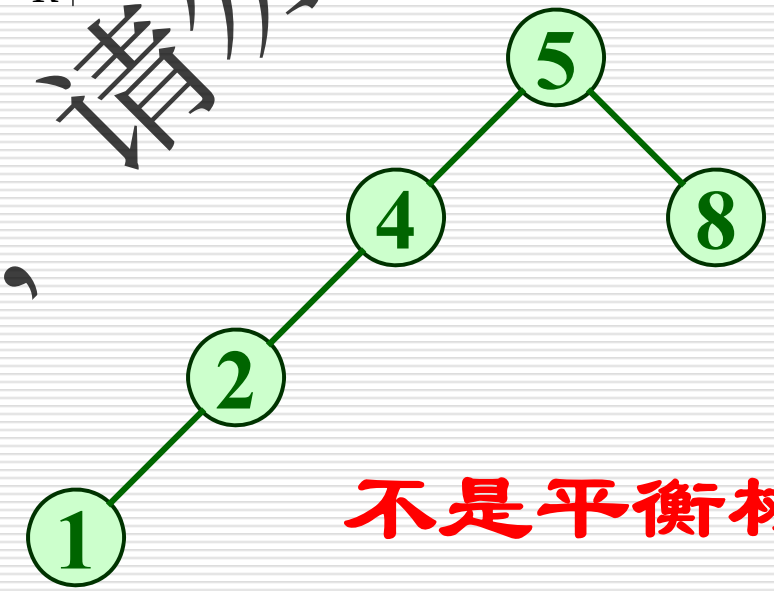
树中每个结点的左、右子树深度之差的绝对值不大于1。

$$|h_L - h_R| \le 1$$

例如：



是平衡树

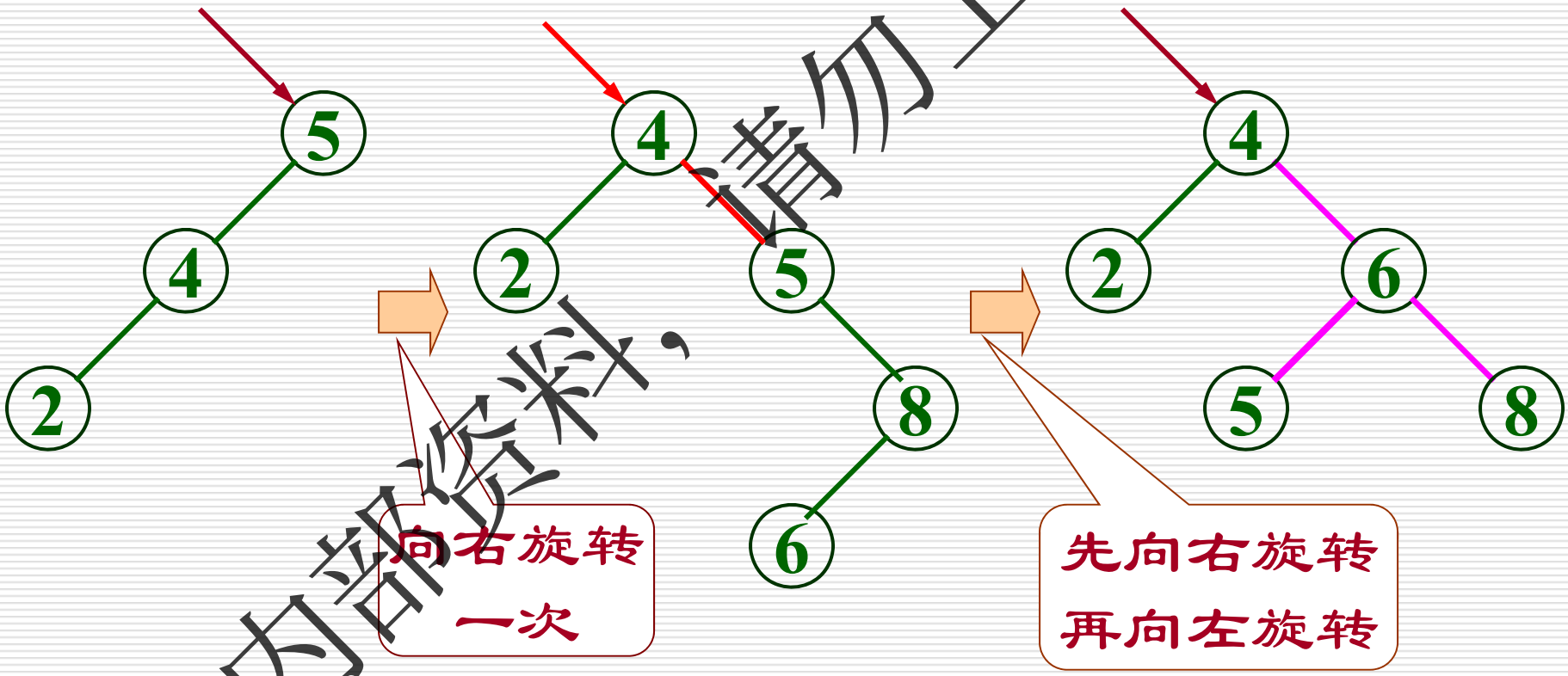


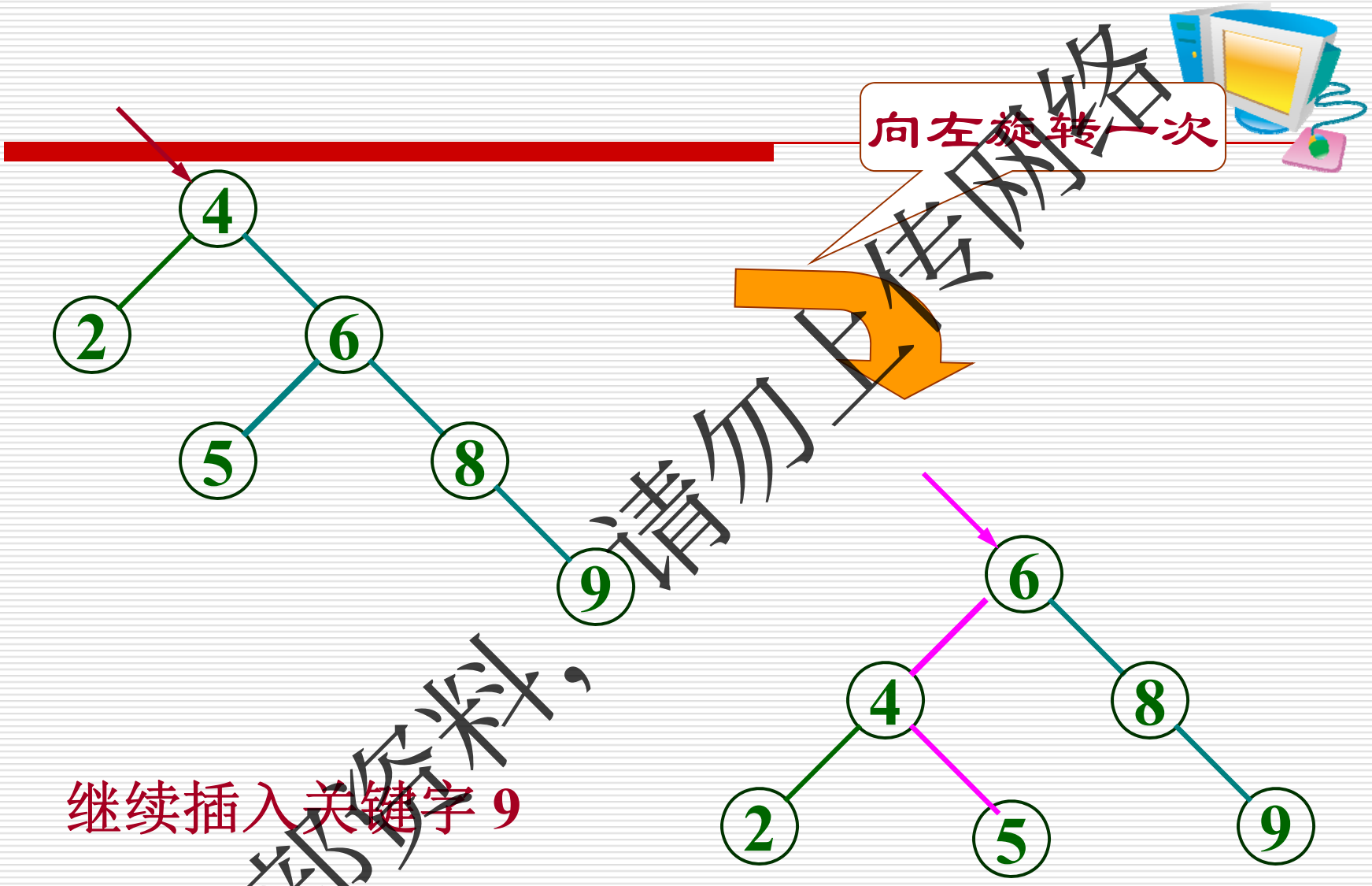
不是平衡树

构造平衡二叉（查找）树的方法

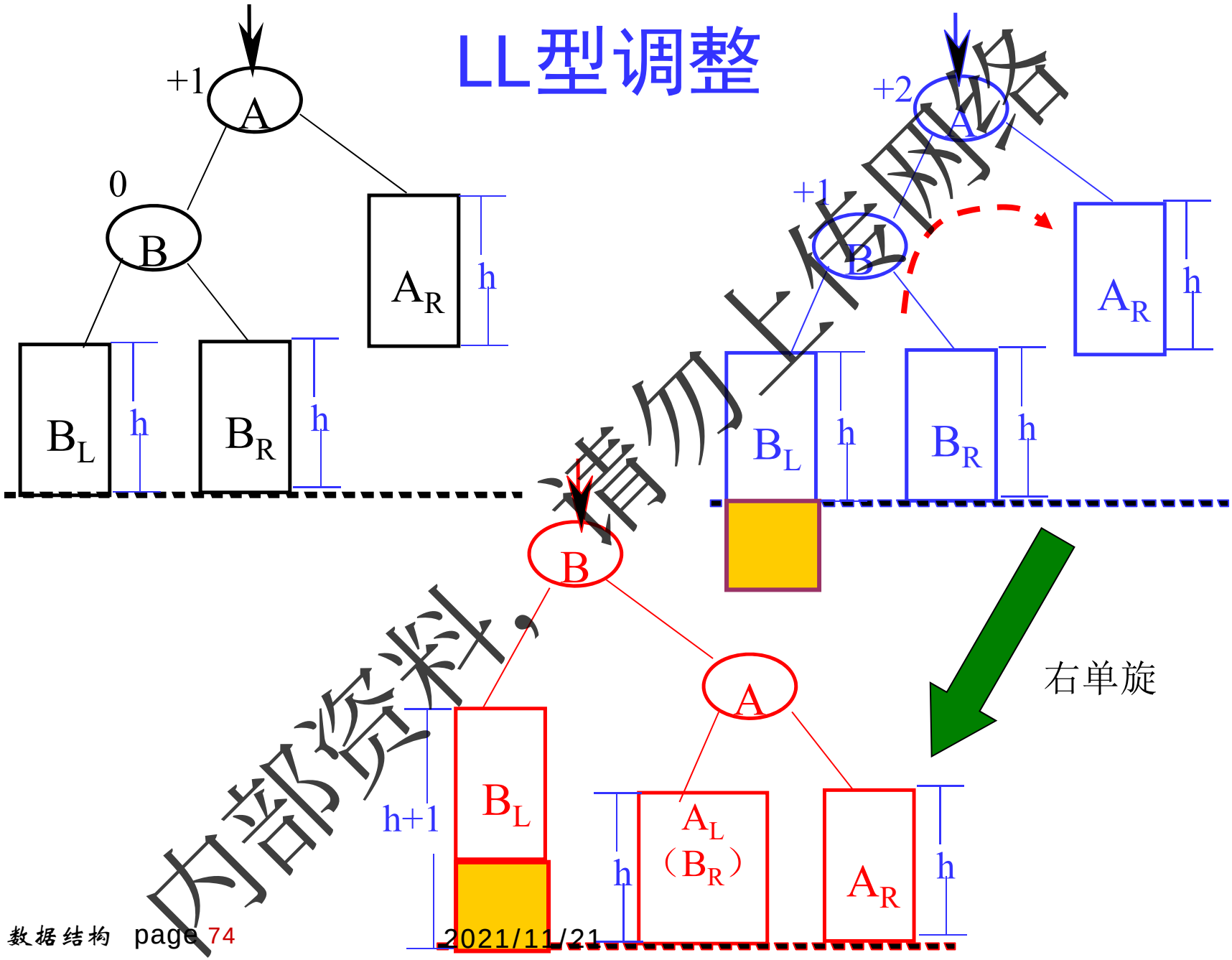
在插入过程中，采用平衡旋转技术。

例如:依次插入的关键字为5, 4, 2, 8, 6, 9

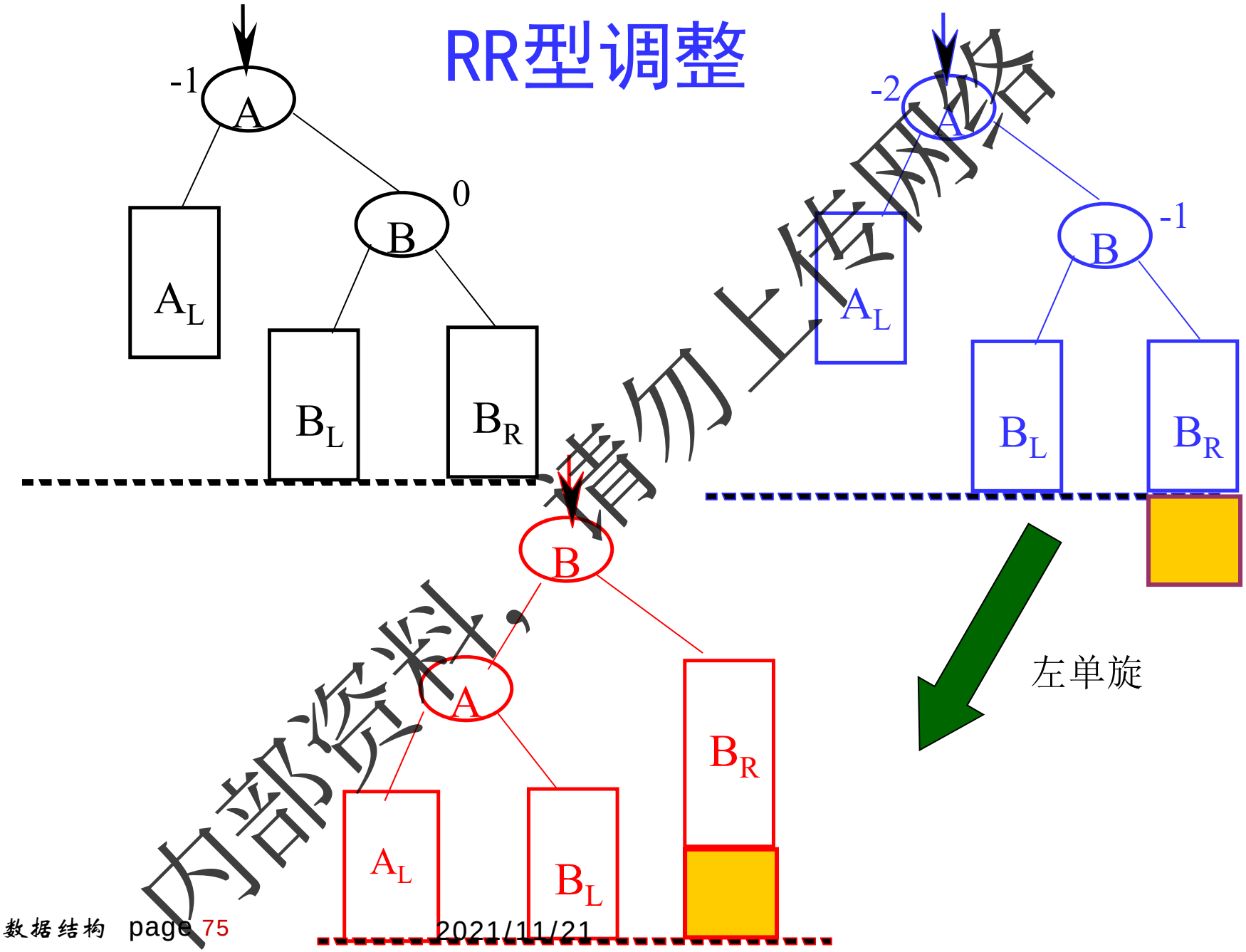




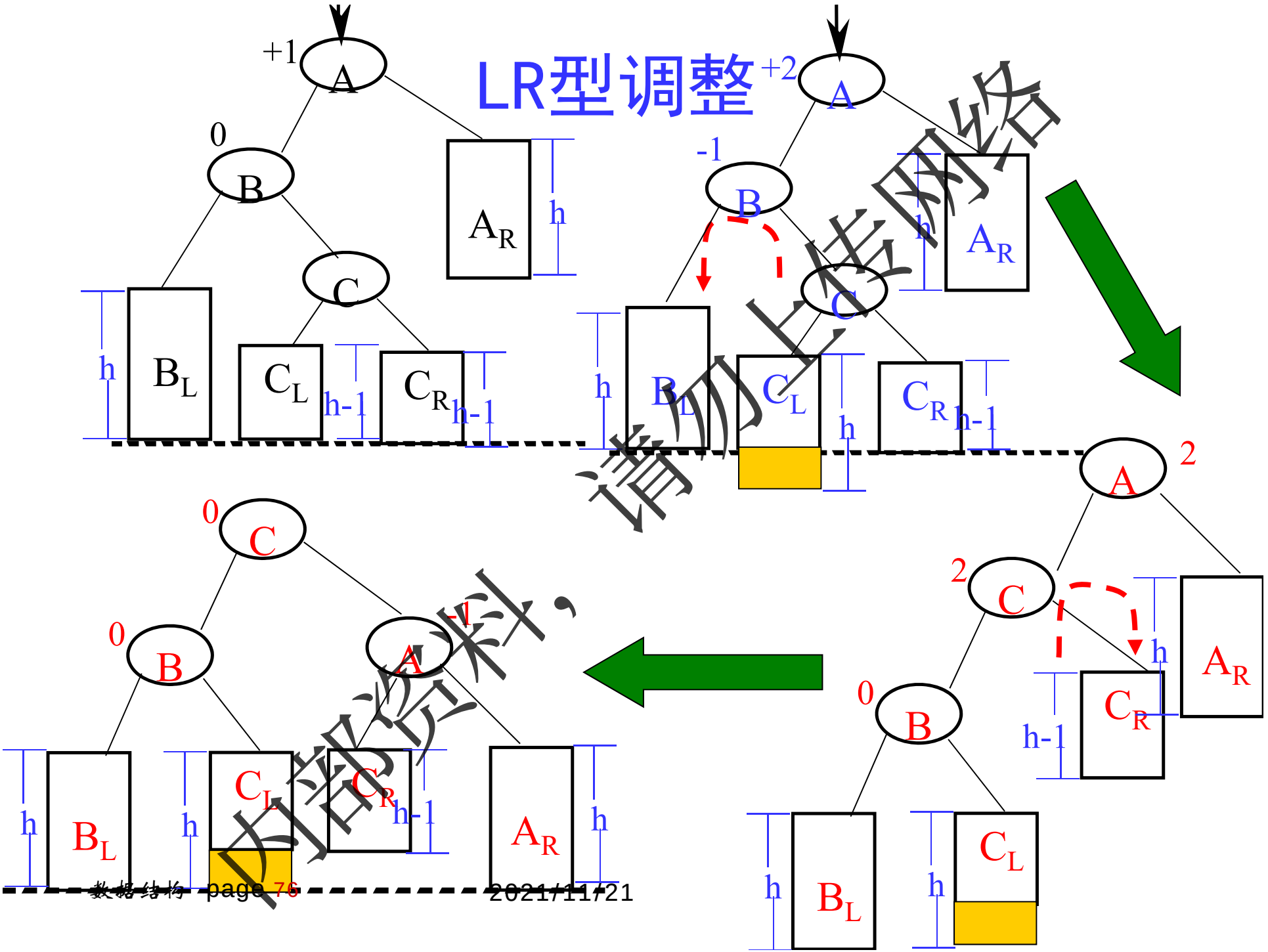
LL型调整



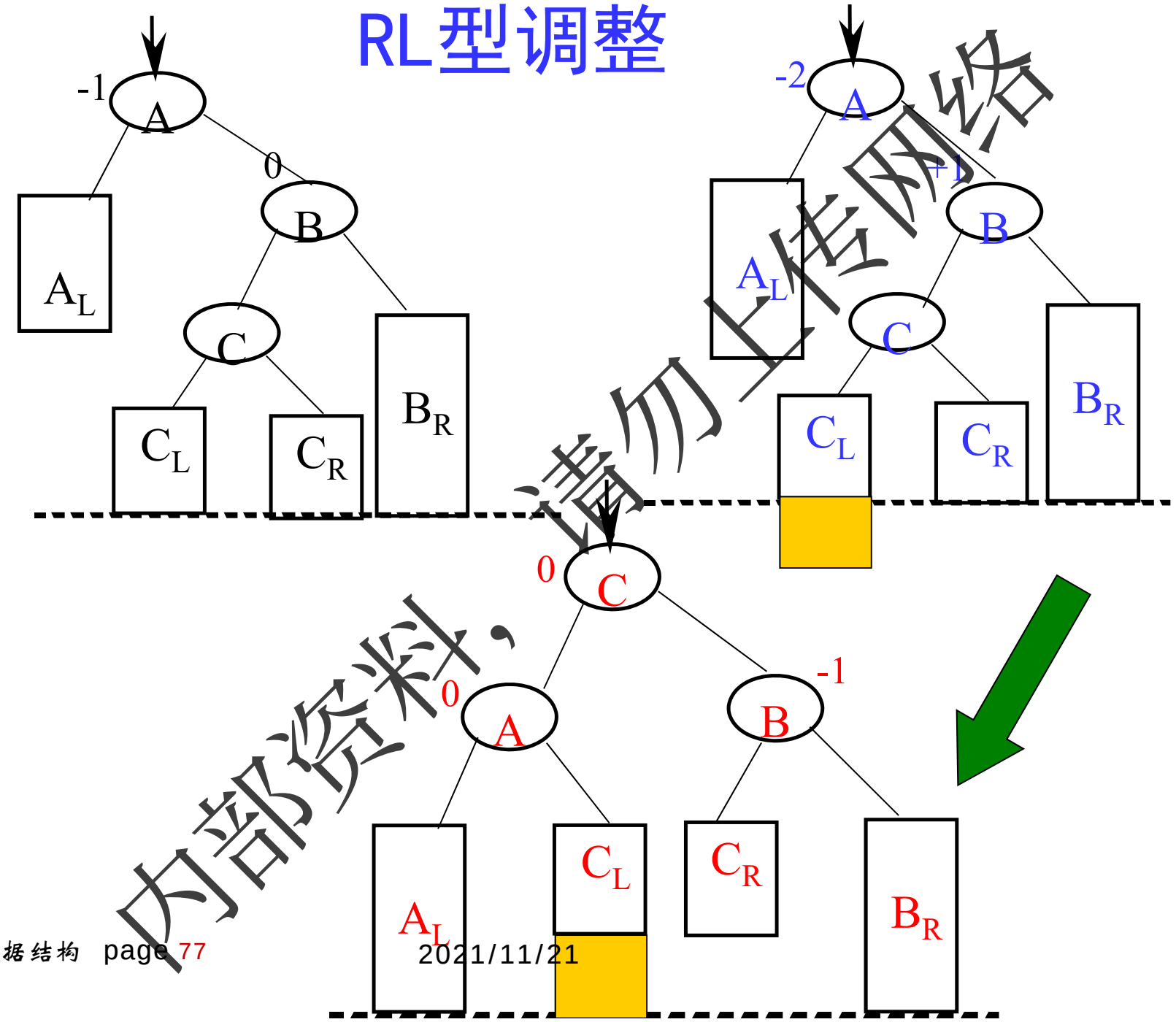
RR型调整



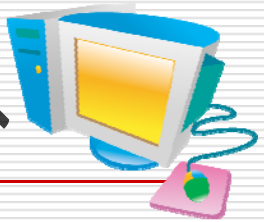
LR型调整



RL型调整



2010年数据结构考研真题



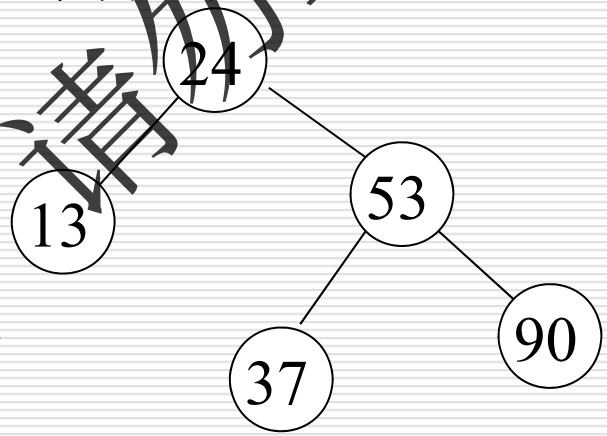
4、在下列所示的平衡二叉树中插入关键字48后得到一棵新平衡二叉树，在新平衡二叉树中，关键字37所在结点的左、右子结点中保存的关键字分别是（C）

A、13，48

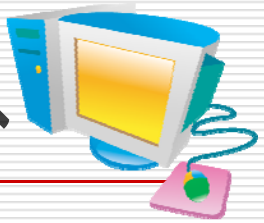
B、24，48

C、24，53

D、24，902



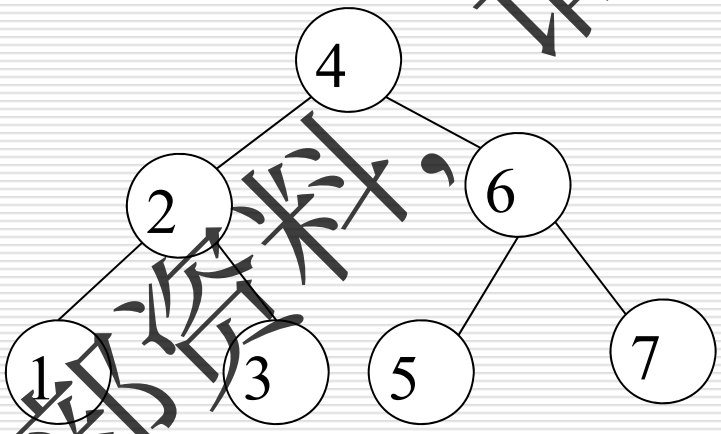
2013年数据结构考研真题



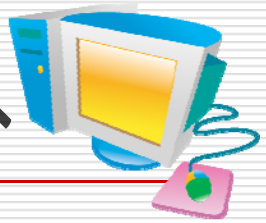
3. 若将关键字1，2，3，4，5，6，7依次插入到初始为空的平衡二叉树*T*中，则*T*中平衡因子为0的分支结点的个数是(D)

A.0　　B. 1　　C. 2　　D. 3

解答：



平衡二叉树的查找性能分析：



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 个关键字的平衡二叉树可能达到的最大深度是多少？

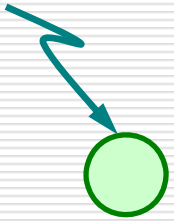
先看几个具体情况：

$n=0$

空树

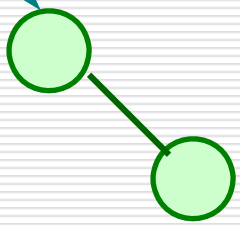
最大深度为 0

$n=1$



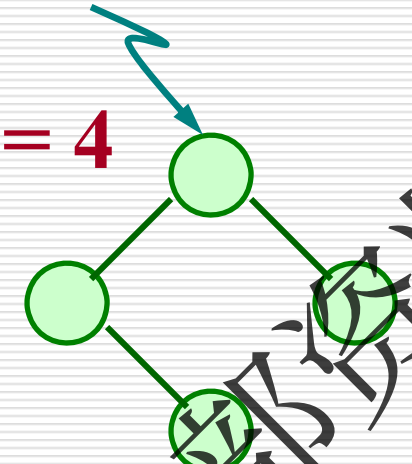
最大深度为 1

$n=2$



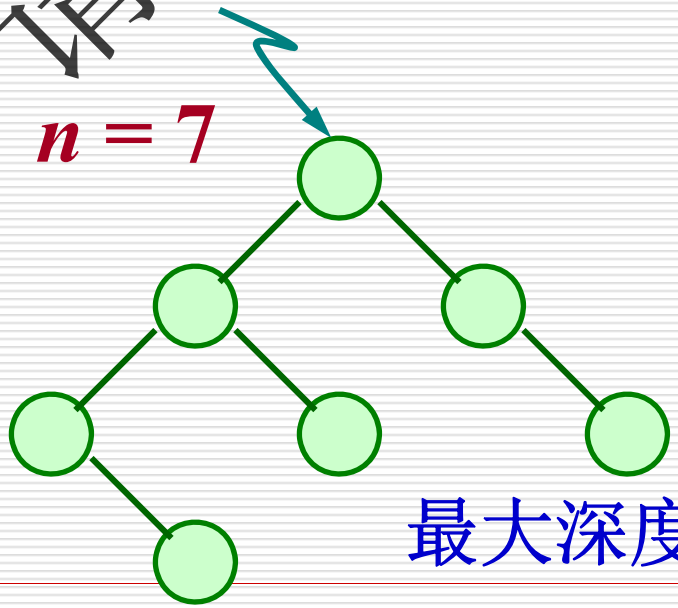
最大深度为 2

$n=4$

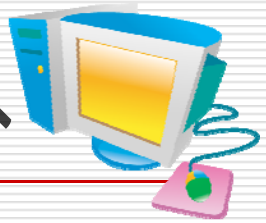


最大深度为 3

$n=7$



最大深度为 4



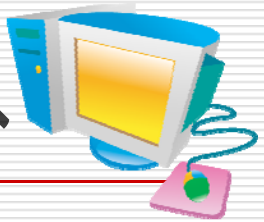
反过来，深度为 h 的二叉平衡树中所含结点的最小值 N_h 是多少？

$h=0$　　$N_0=0$　　$h=1$　　$N_1=1$

$h=2$　　$N_2=2$　　$h=3$　　$N_3=4$

一般情况下，　$N_h = N_{h-1} + N_{h-2} + 1$

利用归纳法可证得，　$N_h = F_{h+2} - 1$

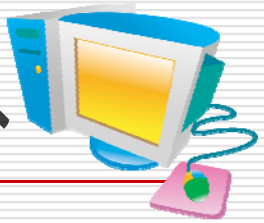


由此推得，深度为 h 的二叉平衡树中所含结点的最小值 $N_h = \varphi^{h+2}/5 - 1$

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个数和 $log(n)$ 相当。

7.3.3B-树

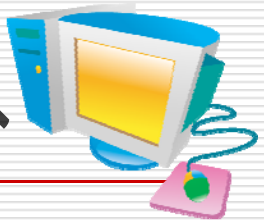


一棵B-树是一棵平衡的多路查找树，它或者是空树，或者是满足如下性质的m叉树：

（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；

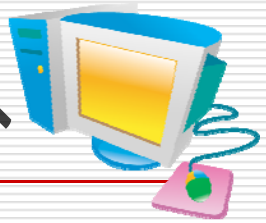
（3） 除根结点之外的所有非终端结点至少有⌈m/2⌉棵子树；



(4) 所有的非终端结点中包含下列信息数据

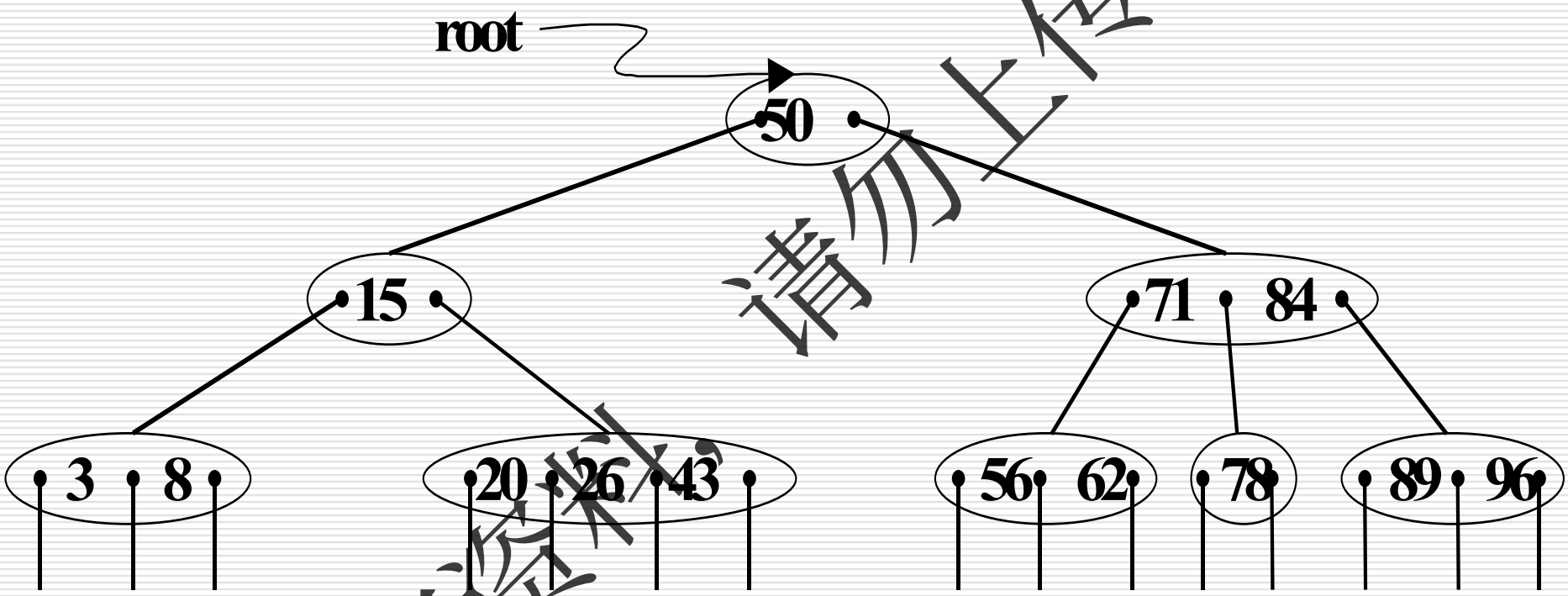
n	A_0	K_1	A_1	K_2	A_2	…	K_n	A_n

其中：**Ki (i=1，…，n)** 为关键字，且 **Ki<Ki+1(i=1，…，n-1)**；**Ai(0≤i≤n)**为指向子树根结点的指针，且指针**Ai-1**所指子树中的所有结点的关键字均小于**Ki (i=1，…，n)**，**Ai**所指子树中的所有关键字均大于**Ki**，**n**为关键字的个数(**m/2 -1<=n<=m-1**) 。

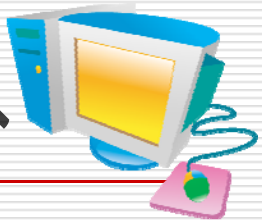


（5） 所有叶子结点出现在同一层上，并且不带信息，通常称为失败结点。失败结点为虚结点，在B-树中并不存在， 指向它们的指针为空指针。引入失败结点是为了便于分析B－树的查找性能。

B-树是一种平衡的多路查找树

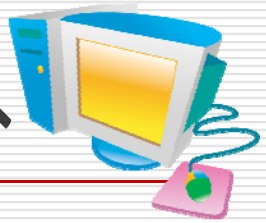


B-树结构的C语言描述如下：



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

2. 查找过程

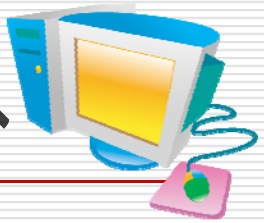


从根结点出发，沿指针搜索结点和在结点内进行顺序（或折半）查找 两个过程交叉进行。

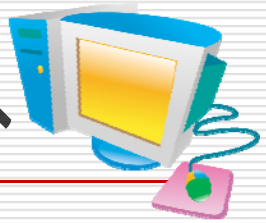
若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置；

若查找不成功，则返回插入位置。

B树的查找结果类型



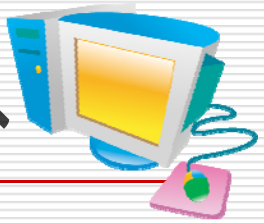
```
typedef struct {
  BTNode  *pt;     // 指向找到的结点的指针
  int  i;          // 1..m，在结点中的关键字序号
  int  tag;        // 标志查找成功(=1)或失败(=0)
} Result;          // 在B树的查找结果类型
```



```
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..keynum]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
```

3. 插入



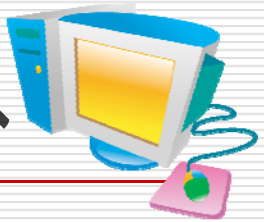
关键字插入的位置必定在最下层的非叶结点，有下列几种情况：

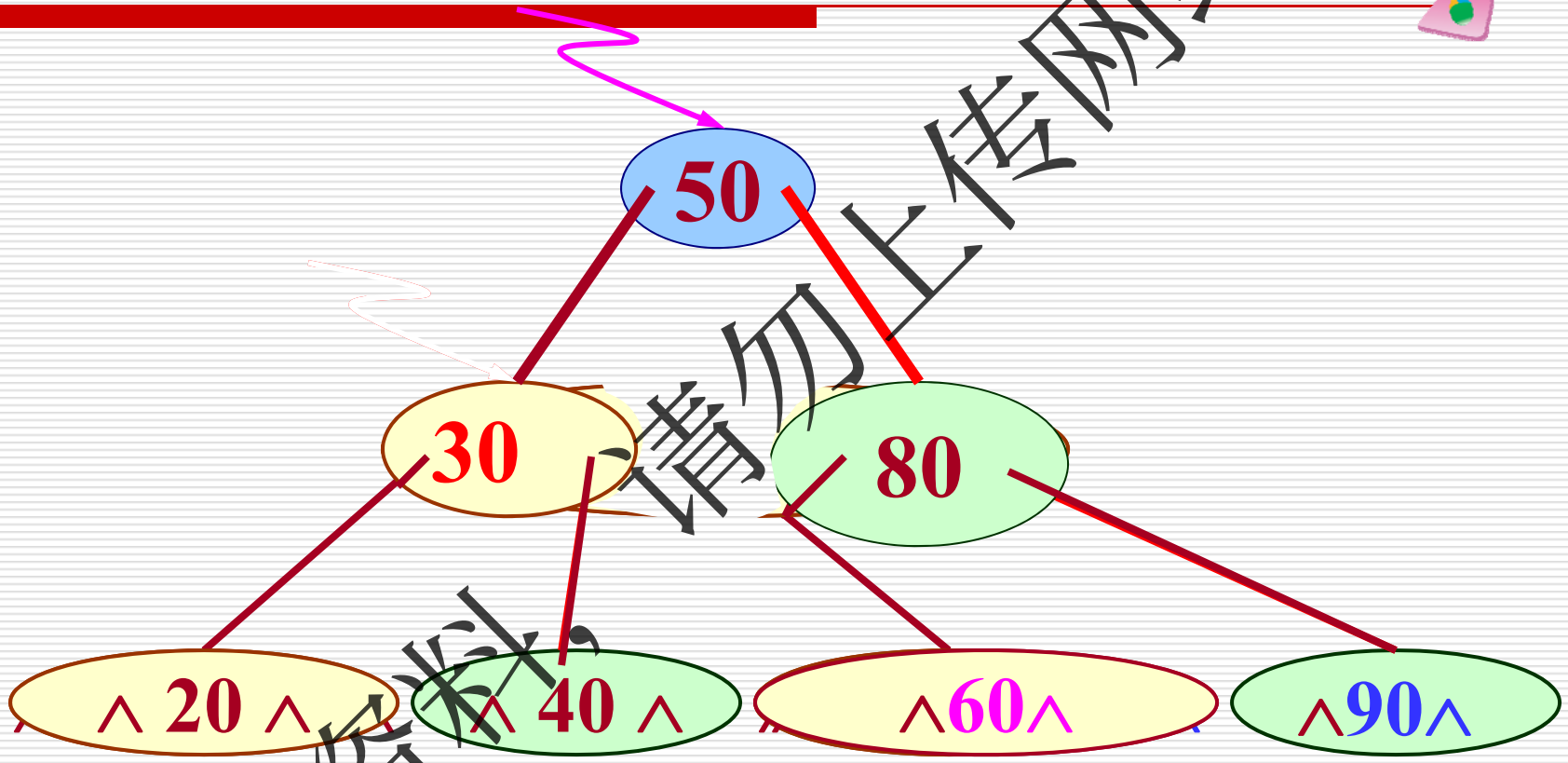
1）插入后，该结点的关键字个数n<m， 不修改指针；

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 =⌈m/2⌉，在原结点中保留（A0，K1，……，Ks-1，As-1）；新结点（As，Ks+1，……，Kn，An）；将（Ks，p）插入双亲结点；

3）若双亲为空，则建新的根结点。

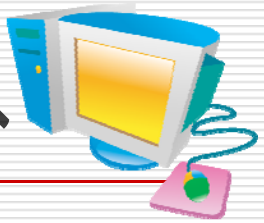
例如:下列为 3 阶B-树





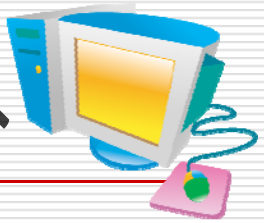
插入关键字= 60, 90, 30,

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”。

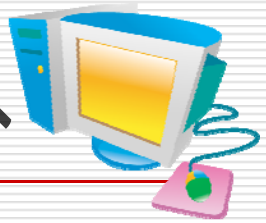
7.3.4 B^+树



1. 结构特点

- 每个叶子结点中含有 n 个关键字和 n 个指向记录的指针；并且，所有叶子结点彼此相链接构成一个有序链表，其头指针指向含最小关键字的结点；
- 每个非叶结点中的关键字 K_i 即为其相应指针 A_i 所指子树中关键字的最大值；
- 所有叶子结点都处在同一层次上，每个叶子结点中关键字的个数均介于 $\lceil m/2 \rceil$ 和 m 之间。

2．查找过程

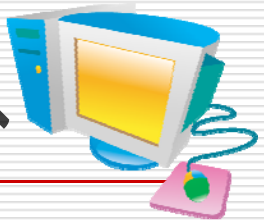


- 在 B^+ 树上，既可以进行缩小范围的查找，也可以进行顺序查找；

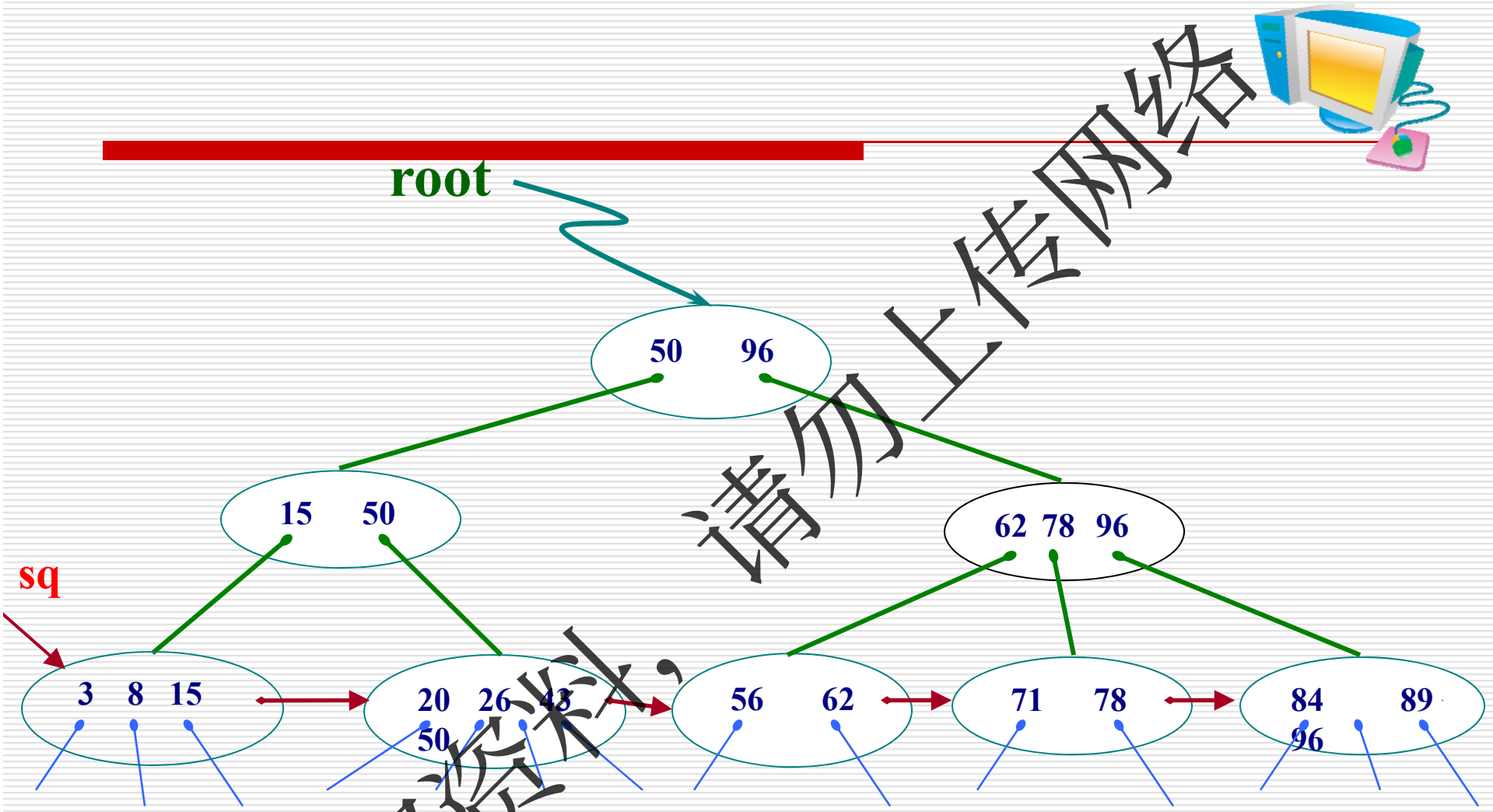
- 在进行缩小范围的查找时，不管成功与否，都必须查到叶子结点才能结束；

- 若在结点内查找时，给定值$\leqslant K_i$， 则应继续在 A_i 所指子树中进行查找；

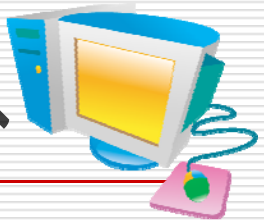
3．插入和删除的操作



类似于B-树进行，即必要时，也需要进行结点的“分裂”或“归并”。



7.4 散列表

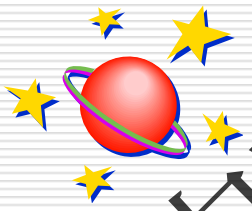


1. 散列表的基本概念

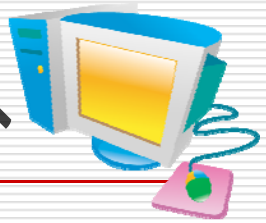
2.散列函数的构造方法

3. 处理冲突的方法

4.散列表的查找



7.4.1 散列表的基本概念

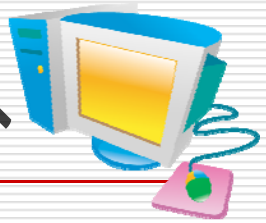


以上两节讨论的表示查找表的各种结构的共同特点：记录在表中的位置和它的关键字之间不存在一个确定的关系。

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个数。

例：

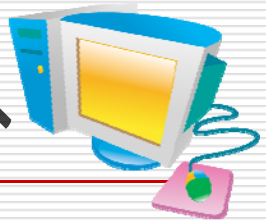


为每年招收的 1000 名新生建立一张查找表，其关键字为学号，其值的范围为 xx000～xx999 (前两位为年份)。

若以下标为000～999 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(key)** 作为关键字为 **key** 的记录在表中的位置，通常称这个函数 **f(key)** 为散列函数或者哈希函数。

例如： 对于如下 9 个关键字

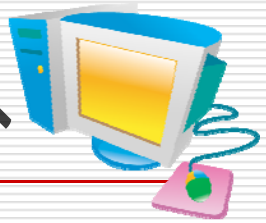
{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Du}

设 哈希函数 f(key) =⌊(Ord(第一个字母) -rd('A')+1)/2⌋

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Du		Han		Li		Qian	Sun		Wu	Ye	Zhao

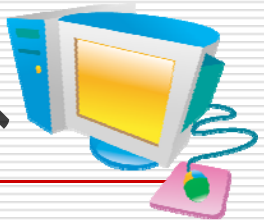
问题：若添加关键字 **Zhou**，怎么办？

从这个例子可见，



1) 哈希(Hash)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。

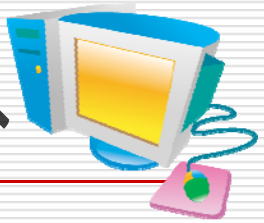
2) 由于哈希函数是一个压缩映象，因此，在一般情况下，很容易产生“冲突”现象，即：**key1≠key2，而 f(key1) = f(key2)。**



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。

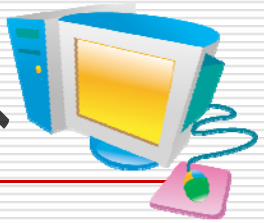
因此，在构造这种特殊的“查找表” 时，除了需要选择一个“好”（尽可能少产生冲突）的哈希函数之外；还需要找到一种“处理冲突” 的方法。

哈希表的定义：



根据设定的哈希函数 H(key) 和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。

2. 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） 数字分析法

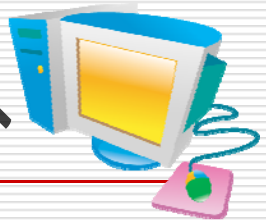
（2） 平方取中法

（3） 折叠法

（4） 除留余数法

若是非数字关键字，则需先对其进行数字化处理。

1. 数字分析法

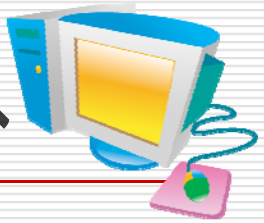


假设关键字集合中的每个关键字都是由 s 位数字组成 $(u_1, u_2, \ldots, u_s)$，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

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。

2. 平方取中法

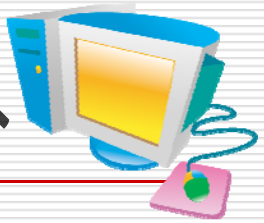


以关键字的平方值的中间几位作为存储地址。求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

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。

3. 折叠法

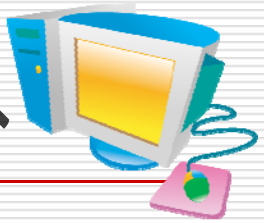


将关键字分割成若干部分，然后取它们的叠加和为哈希地址。有两种叠加处理的方法：移位叠加和间界叠加。

此方法适合于：

关键字的数字位数特别多。

4. 除留余数法



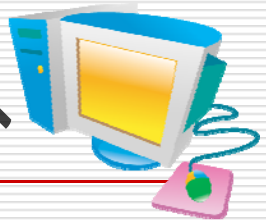
设定哈希函数为:

$$H(key) = key \ MOD \ p$$

其中， $p \le m$（表长） 并且

p 应为不大于 m 的素数或不含 20 以下的质因子

为什么要对 p 加限制？

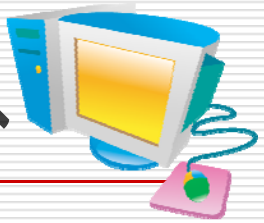


例如：

给定一组关键字为: 12, 39, 18, 24, 33, 21，若取 p=9, 则它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。

3.处理冲突的方法



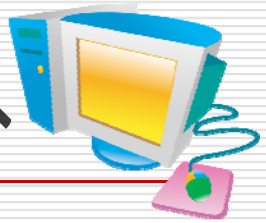
(1) 开放地址法（开地址法）

(2) 再哈希法（双哈希函数法）

(3) 链地址法（拉链法）

(4) 建立一个公共溢出区

(1) 开放定址法



为产生冲突的地址 H(key) 求得一个地址序列：

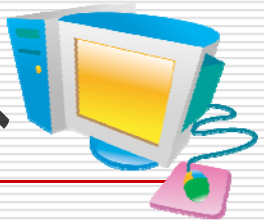
$H_0, H_1, H_2, \ldots, H_k \quad 1 \le k \le m-1$

其中：$H_0 = H(key)$

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

$i = 1, 2, \ldots, k$

对增量 d_i 有三种取法：



- 1）线性探测再散列

  $d_i = 1,2,3,\cdots,m-1$ （最简单的情况）

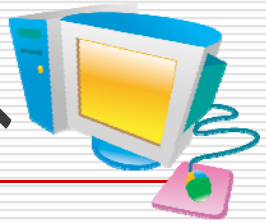
- 2）平方探测再散列

  $d_i = 1^2, -1^2, 2^2, -2^2, \cdots,$

- 3）随机探测再散列

  d_i 是一组伪随机数序列

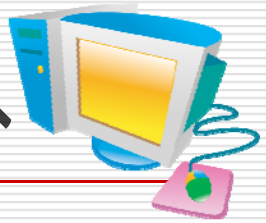
注意：增量 d_i 应具有“完备性”



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。

例如:



关键字集合 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key MOD 11 (表长=11)

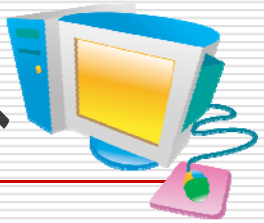
若采用线性探测再散列处理冲突

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

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

(2)再哈希法（双哈希函数法）



$$H_i=RH_i(key) \qquad i=1, 2, \ldots, k$$

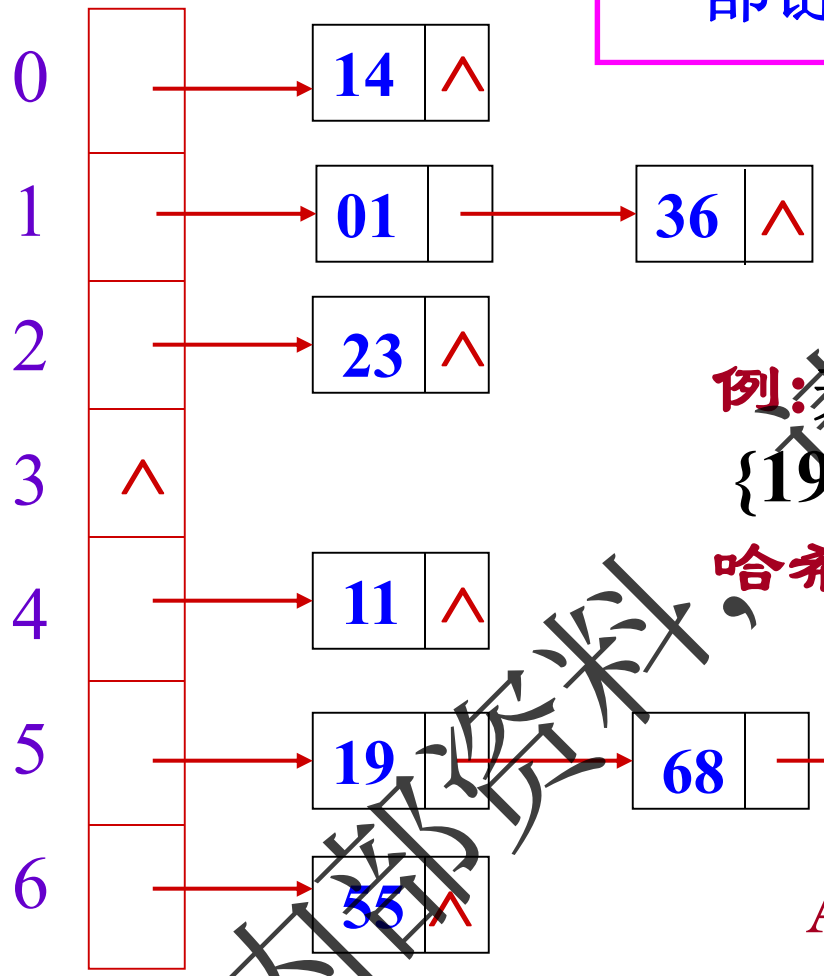
RH_i均是不同的哈希函数，当产生冲突时就计算另一个哈希函数，直到冲突不再发生。

优点：不易产生聚集；

缺点：增加了计算时间。

(3) 链地址法

将所有哈希地址相同的记录都链接在同一链表中。



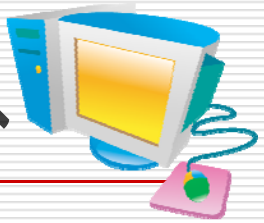
例:关键字集合

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

哈希函数： H(key)=key MOD 7

ASL=(6×1+2×2+3)/9=13/9

(4)建立一个公共溢出区

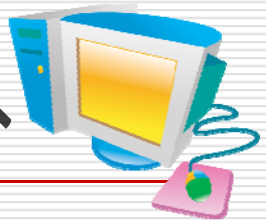


思路：

除设立哈希基本表外，另设立一个溢出向量表。

所有关键字和基本表中关键字为同义词的记录，不管它们由哈希函数得到的地址是什么，一旦发生冲突，都填入溢出表。

4. 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] = NULL** 则查找不成功

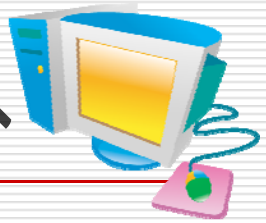
若 **r[i].key = K** 则查找成功

否则 “求下一地址 **Hi**”，直至

r[Hi] = NULL (查找不成功)

或 **r[Hi].key = K** (查找成功) 为止。

哈希表查找的分析

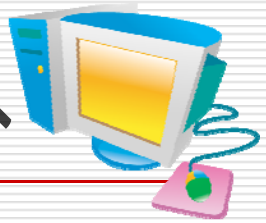


从查找过程得知，哈希表查找的平均查找长度实际上并不等于零。

决定哈希表查找的ASL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 α = n / m 值的大小（n—记录数，m—表的长度）。

查找成功时有下列结果：



线性探测再散列

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

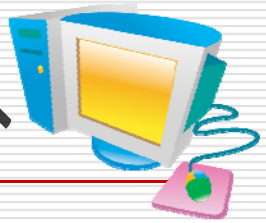
随机探测再散列

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

链地址法

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

结论：

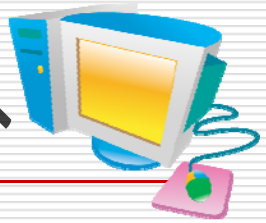


哈希表的平均查找长度是 α 的函数，而不是 n 的函数。

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 α，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

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(key)=key MOD 13,

哈希表长为m=16，

设每个记录的查找概率相等

关键字(19,14,23,1,68,20,84,27,55,11,10,79)

(1) 用线性探测再散列处理冲突，即Hi=(H(key)+di) MOD m

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	1	68	27	55	19	20	84	79	23	11	10			

H(19)=6

H(14)=1

H(23)=10

H(1)=1　冲突，H1=(1+1) MOD16=2

H(68)=3

H(20)=7

H(84)=6　冲突，H1=(6+1)MOD16=7

冲突，H2=(6+2)MOD16=8

H(27)=1　冲突，H1=(1+1)MOD16=2

冲突，H2=(1+2)MOD16=3

冲突，H3=(1+3)MOD16=4

H(55)=3　冲突，H1=(3+1)MOD16=4

冲突，H2=(3+2)MOD16=5

H(11)=11

H(10)=10　冲突，H1=(10+1)MOD16=11

冲突，H2=(10+2)MOD16=12

H(79)=1　冲突，H1=(1+1)MOD16=2

冲突，H2=(1+2)MOD16=3

冲突，H3=(1+3)MOD16=4

冲突，H4=(1+4)MOD16=5

冲突，H5=(1+5)MOD16=6

冲突，H6=(1+6)MOD16=7

冲突，H7=(1+7)MOD16=8

冲突，H8=(1+8)MOD16=9

ASL=(1*6+2+3*3+4*1+9*1)/12=2.5

关键字(19,14,23,1,68,20,84,27,55,11,10,79)

(2) 用二次探测再散列处理冲突，即Hi=(H(key)+di) MOD m

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	14	1	68	55	84	19	20		10	23	11		79		

H(19)=6

H(14)=1

H(23)=10

H(1)=1　冲突，H1=(1+1²) MOD16=2

H(68)=3

H(20)=7

H(84)=6　冲突，H1=(6+1²)MOD16=7

冲突，H2=(6－1²)MOD16=5

H(27)=1　冲突，H1=(1+1²)MOD16=2

冲突，H2=(1-1²)MOD16=0

H(55)=3　冲突，H1=(3+1²)MOD16=4

H(11)=11

H(10)=10　冲突，H1=(10+1²)MOD16=11

冲突，H2=(10 －1²)MOD16=9

H(79)=1　冲突，H1=(1 +1²)MOD16=2

冲突，H2=(1 － 1²)MOD16=0

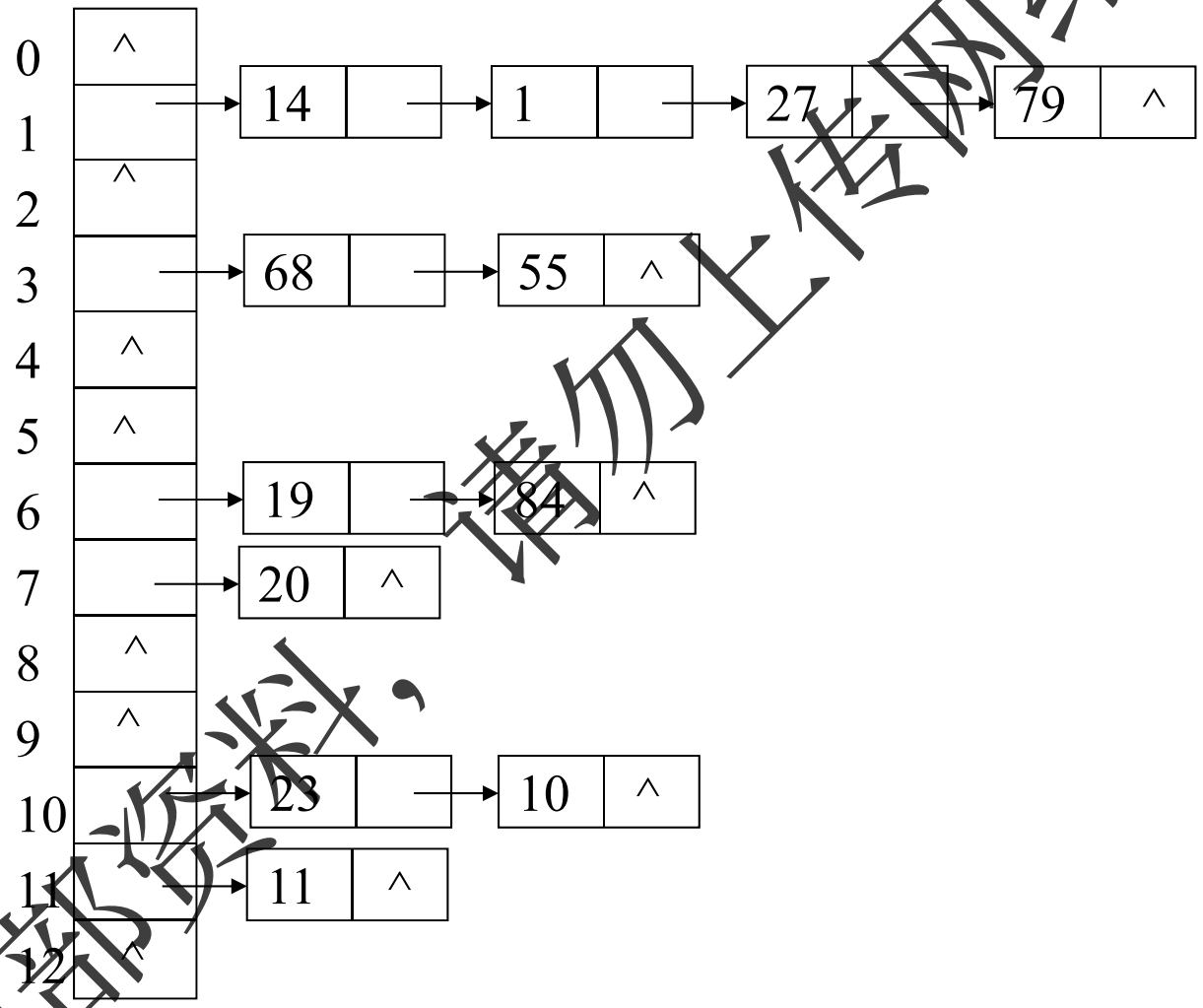
冲突，H3=(1+ 2²)MOD16=5

冲突，H4=(1 － 2²)MOD16=13

ASL=(1*6+2*2+3*3+5*1)/12=2

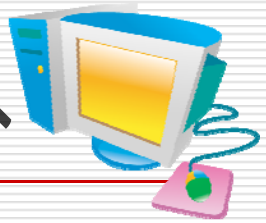
关键字(19,14,23,1,68,20,84,27,55,11,10,79)

(3) 用链地址法处理冲突



ASL=(1*6+2*4+3*1+4*1)/12=1.75

2010年计算机考研真题

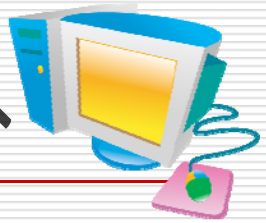


41.（10分）将关键字序列(7、8、30、11、18、9、14)散列存储到散列表中，散列表的存储空间是一个下标从0开始的一个一维数组散列,函数为： H(key)=(key×3)MOD T，处理冲突采用线性探测再散列法，要求装载因子为0.7.

问题：

(1)请画出所构造的散列表。

(2)分别计算等概率情况下，查找成功和查找不成功的平均查找长度。



解答：

（1）由装载因子0.7，数据总数为7，所以存储空间长度为L=7/0.7=10。所以构造的散列函数为：

$$H(key)=(key\times 3)MOD10$$

散列表为：

0	1	2	3	4	5	6	7	8	9
30	7	14	11	8	18		9		

（2）查找成功的ASL=（1+1+1+1+2+1+1）/7=8/7

查找不成功的ASL=（7+6+5+4+3+2+1+2+1+1）/10=3.2

7.5本章小结

1. 熟练掌握顺序表和有序表（折半查找）的查找算法及其性能分析方法；

2.熟练掌握二叉排序树的构造和查找算法及其性能分析方法；

3.熟练掌握二叉排序树的插入算法，掌握删除方法；

4.掌握平衡二叉树的定义

5.熟练掌握哈希函数（除留余数法）的构造

6.熟练掌握哈希函数解决冲突的方法及其特点

- 开放地址法（线性探测法、二次探测法）
- 链地址法
- 给定实例计算平均查找长度ASL，ASL依赖于装填因子α

第七章　结束

Thank you!