# 第九章 查找表

# 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。

- **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。

# 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，则称之谓“**主关键字**”。

若此关键字能识别**若干**记录，则称之谓“**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。**

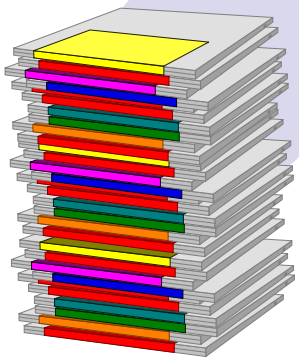
由于查找表中的数据元素之间不存在明显的组织规律，因此不便于查找。

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

# 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。

# 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  , TElemType ;
```

# 一、顺序查找表

## 二、有序查找表

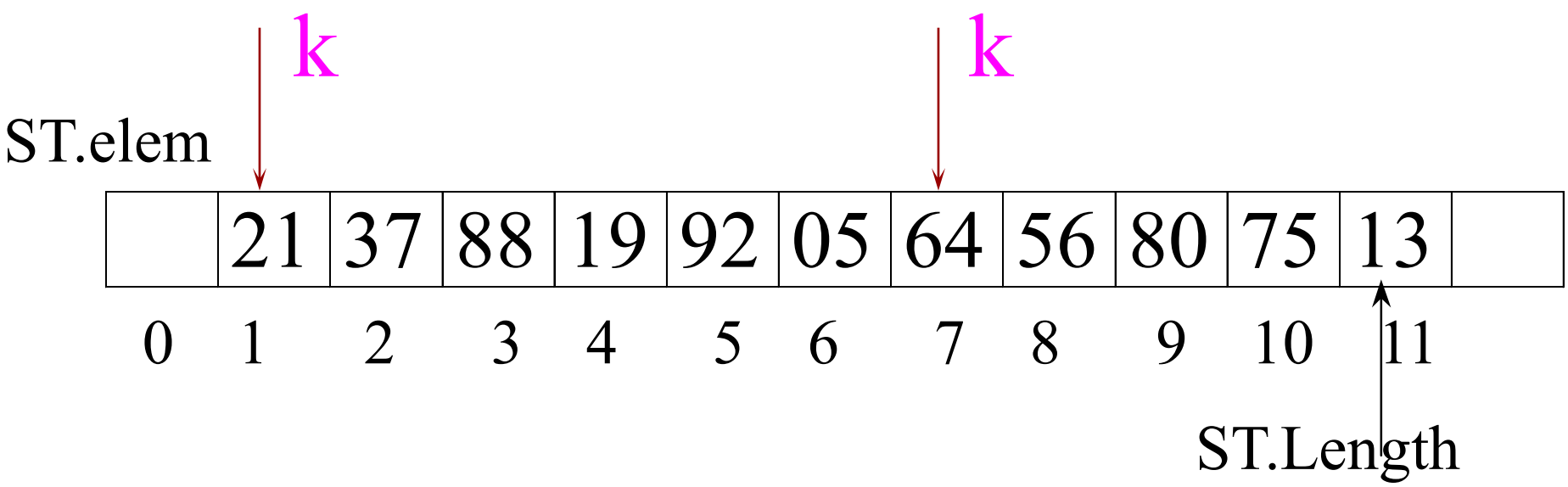
## 三、静态查找树表

## 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

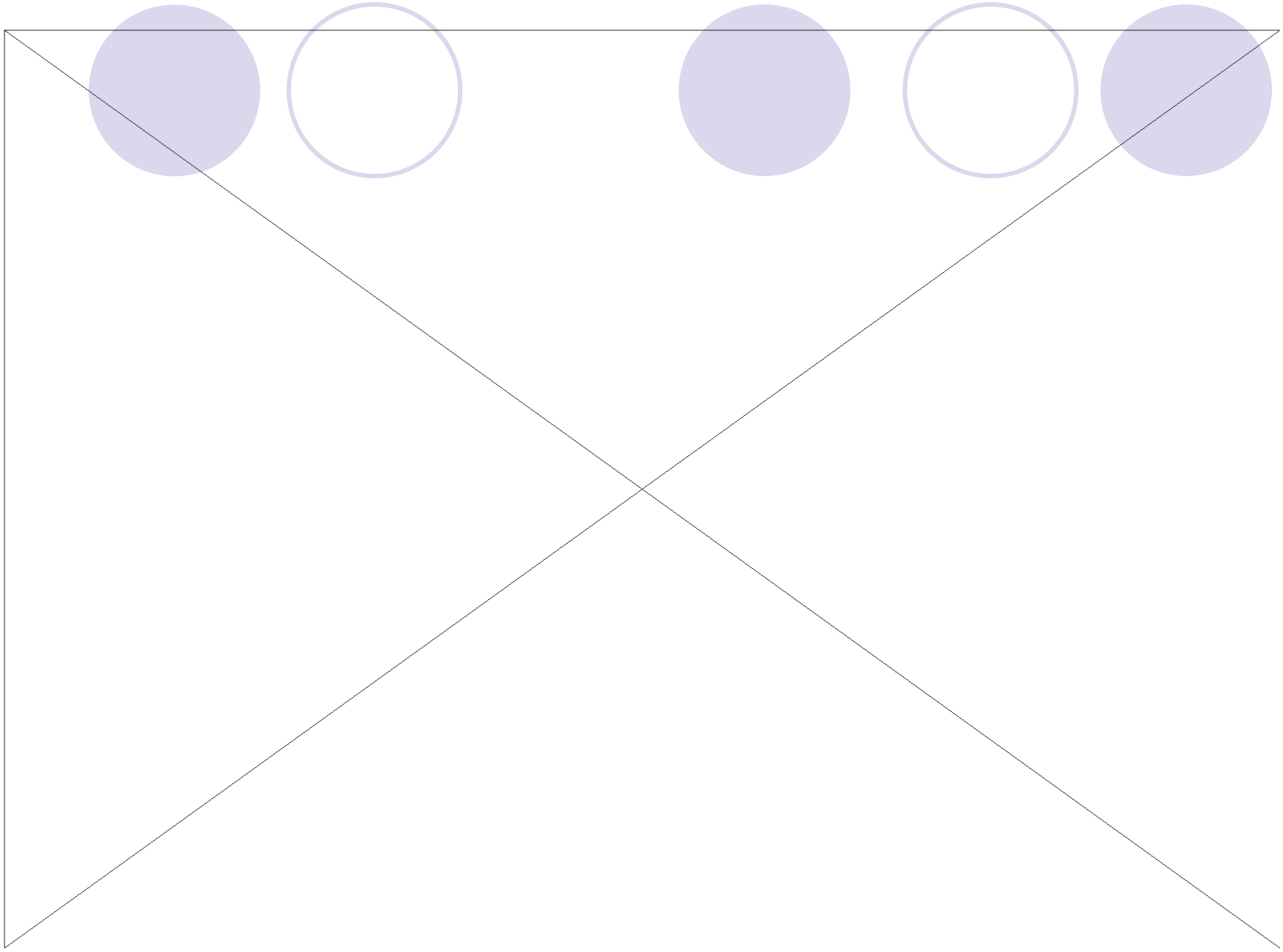
# 回顾顺序表的查找过程:




假设给定值 e=64,

要求 ST.elem[k] = e, 问 : k = ?

```
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
```



```
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
```

# 分析顺序查找的时间性能

**定义：** 查找算法的**平均查找长度**

(**A**verage **S**earch **L**ength)

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

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

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，$C_i = n-i+1$

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

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：

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

在不等概率查找的情况下，$ASL_{ss}$ 在

$$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$$

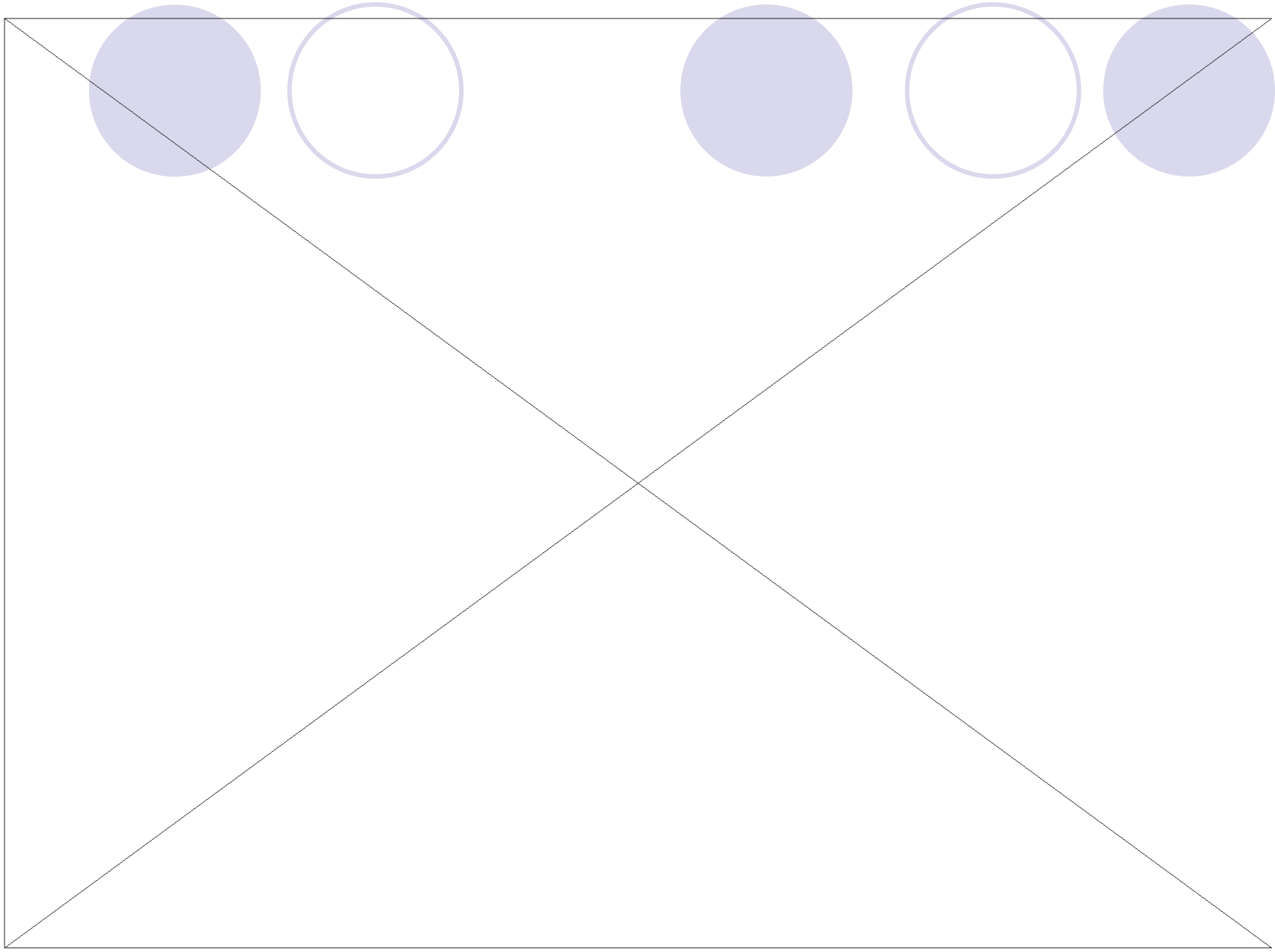
时取极小值

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。

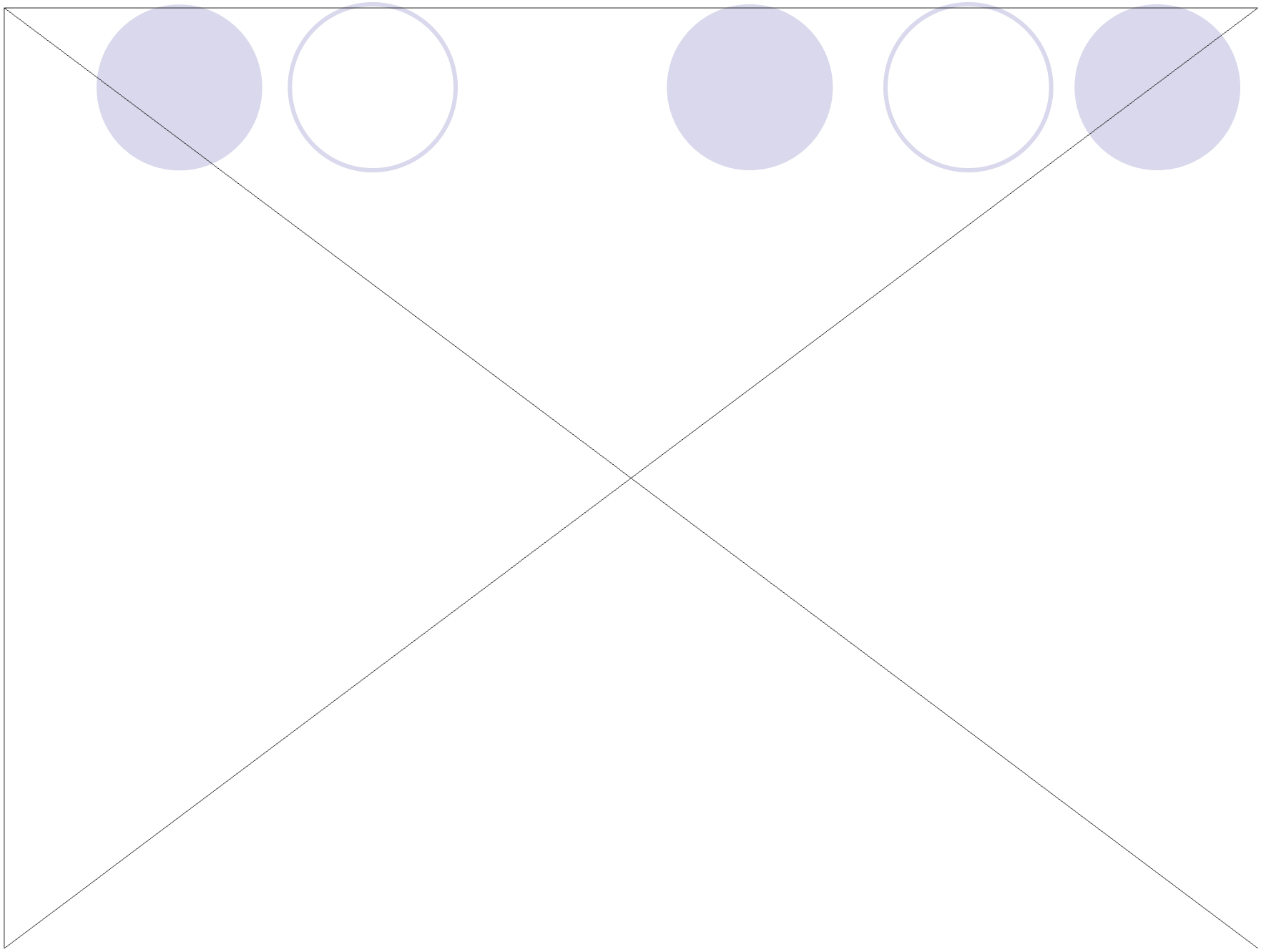
# 二、有序查找表

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。



```
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
```



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

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

题1：采用顺序查找方法查找长度为n的线性表时，不成功情况下平均比较次数为_____。

A.n　　B.n/2　　C. (n+1)/2　　D.(n-1)/2

题2：有一个有序表为{1,3,9,12,32,41,45,62,75,77,82,95,99}，当采用折半查找关键字为82的元素时，___次比较后查找成功。

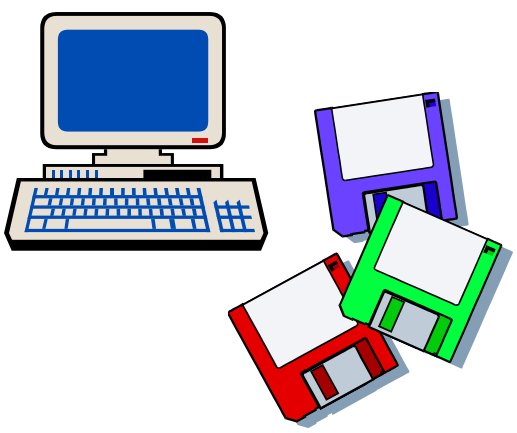
A.1　　B. 2　　C.4　　D.8

题3：设有100个元素的有序表，用折半查找时，不成功时最大的比较次数是_____。

A. 25　　B. 50　　C.10　　D. 7

# 9.2

# 动态查找树表



# 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:

InitDSTable(&DT)

DestroyDSTable(&DT)

SearchDSTable(DT, key);

InsertDSTable(&DT, e);

DeleteDSTable(&T, key);

TraverseDSTable(DT, Visit());

**}ADT DynamicSearchTable**

# 综合上一节讨论的几种查找表的特性:

| | 查找 | 插入 | 删除 |
|---|---|---|---|
| 无序顺序表 | O(n) | O(1) | O(n) |
| 无序线性链表 | O(n) | **O(1)** | **O(1)** |
| 有序顺序表 | **O(logn)** | O(n) | O(n) |
| 有序线性链表 | O(n) | **O(1)** | **O(1)** |

## 可得如下结论:

1) 从查找性能看，最好情况能达O(logn)，此时要求表有序;

2) 从插入和删除的性能看，最好情况能达O(1)，此时要求存储结构是链表。

# 一、二叉排序树( 二叉查找树)

## 二、二叉平衡树

三、B - 树

四、$B^+$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. 定义:

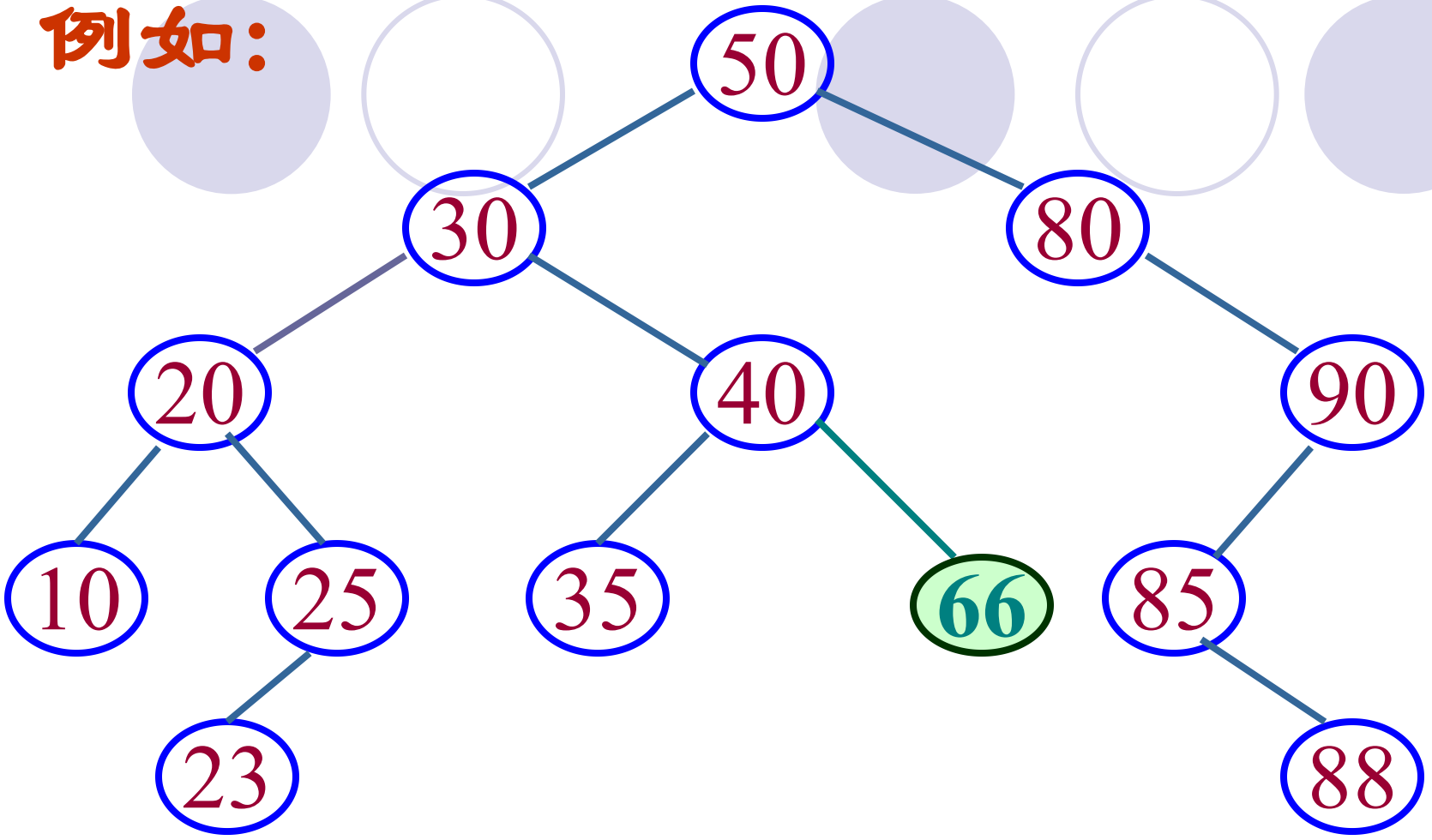
二叉排序树或者是一棵空树；或者是具有如下特性的二叉树:

（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;

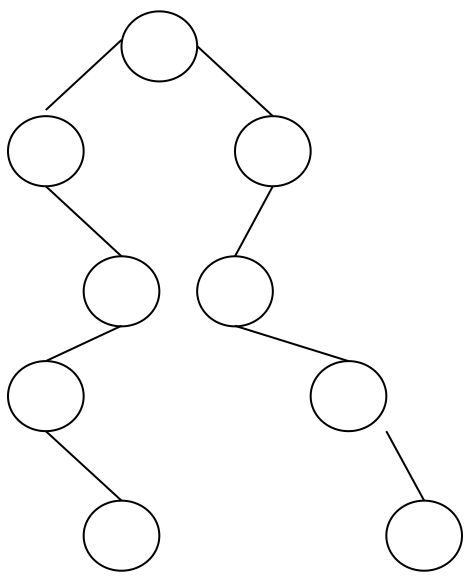
（3）它的左、右子树也都分别是二叉排序树。

例如：




不是二叉排序树。

例：一棵二叉排序树结构如下,各结点的值从小到大依次为1-9，请标出各结点的值。



# 通常，取二叉链表作为二叉排序树的存储结构

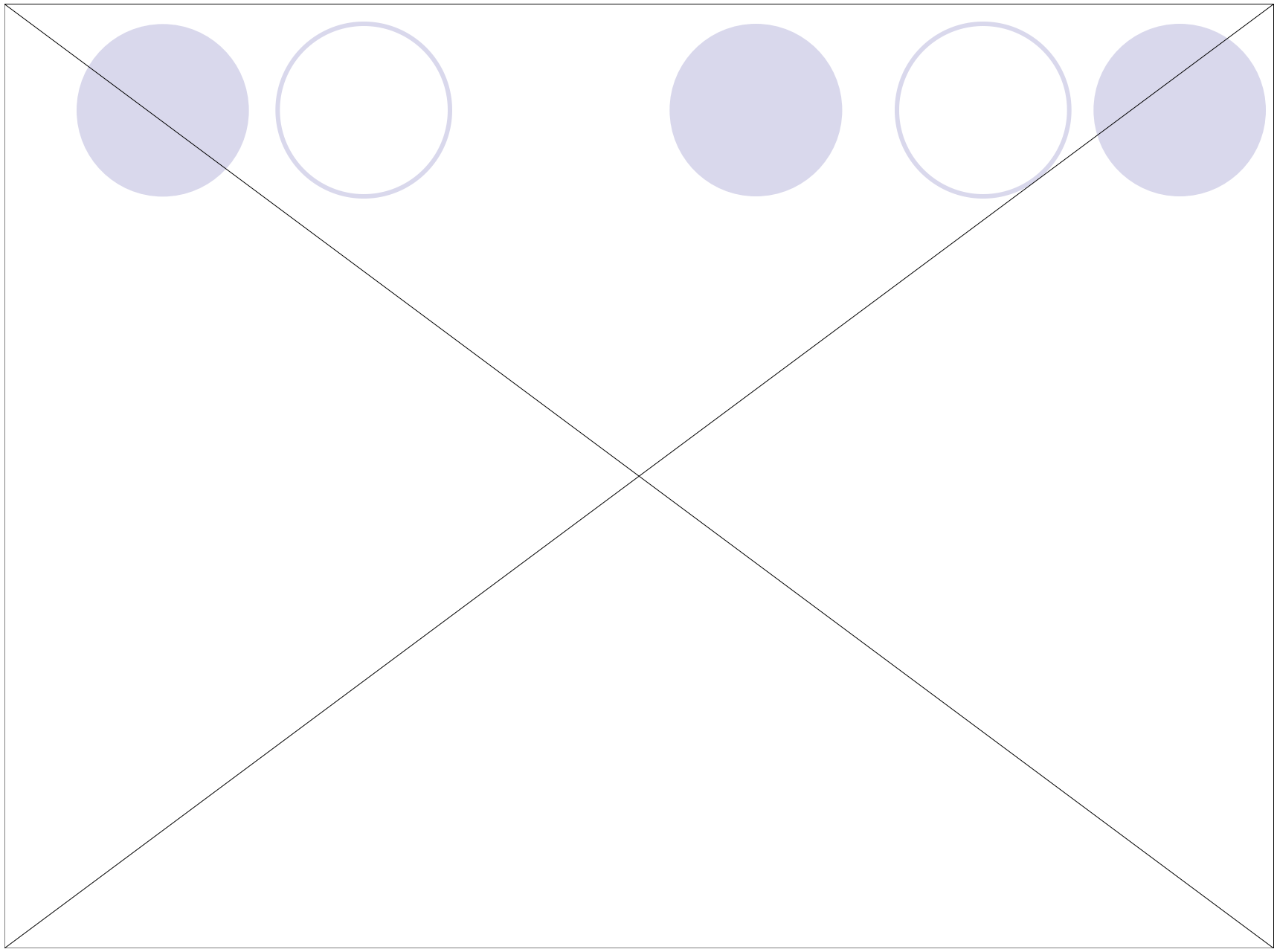
```
typedef struct BiTNode { // 结点结构
    TElemType      data;
    struct BiTNode  *lchild, *rchild;
                                  // 左右孩子指针
} BiTNode, *BiTree;
```

## 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**。



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。

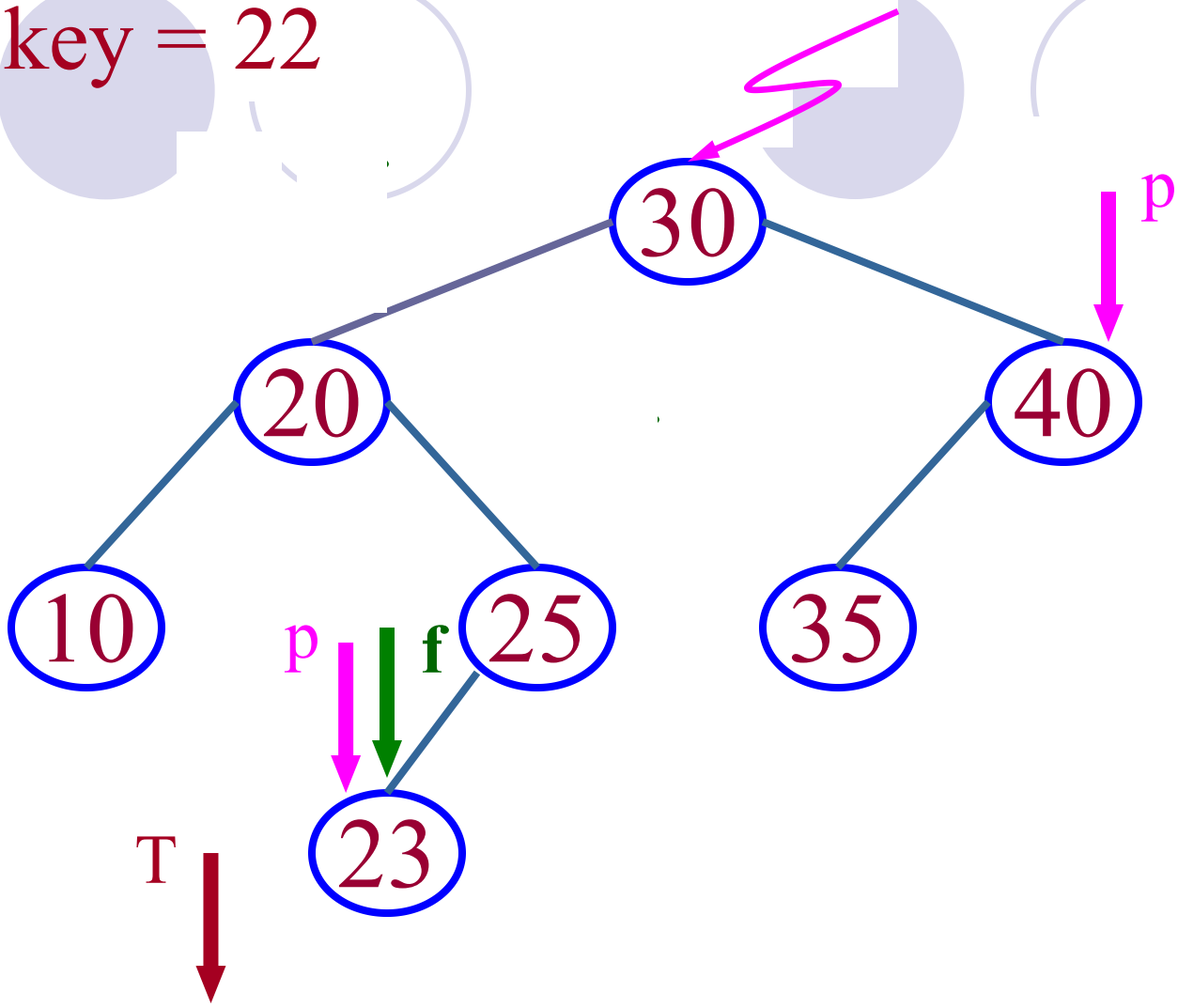
——查找不成功

## 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 指向该数据元素的结点，并返回
  // 函数值为 TRUE；否则表明查找不成功，返回
  // 指针 p 指向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
```

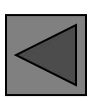
设 key = 22



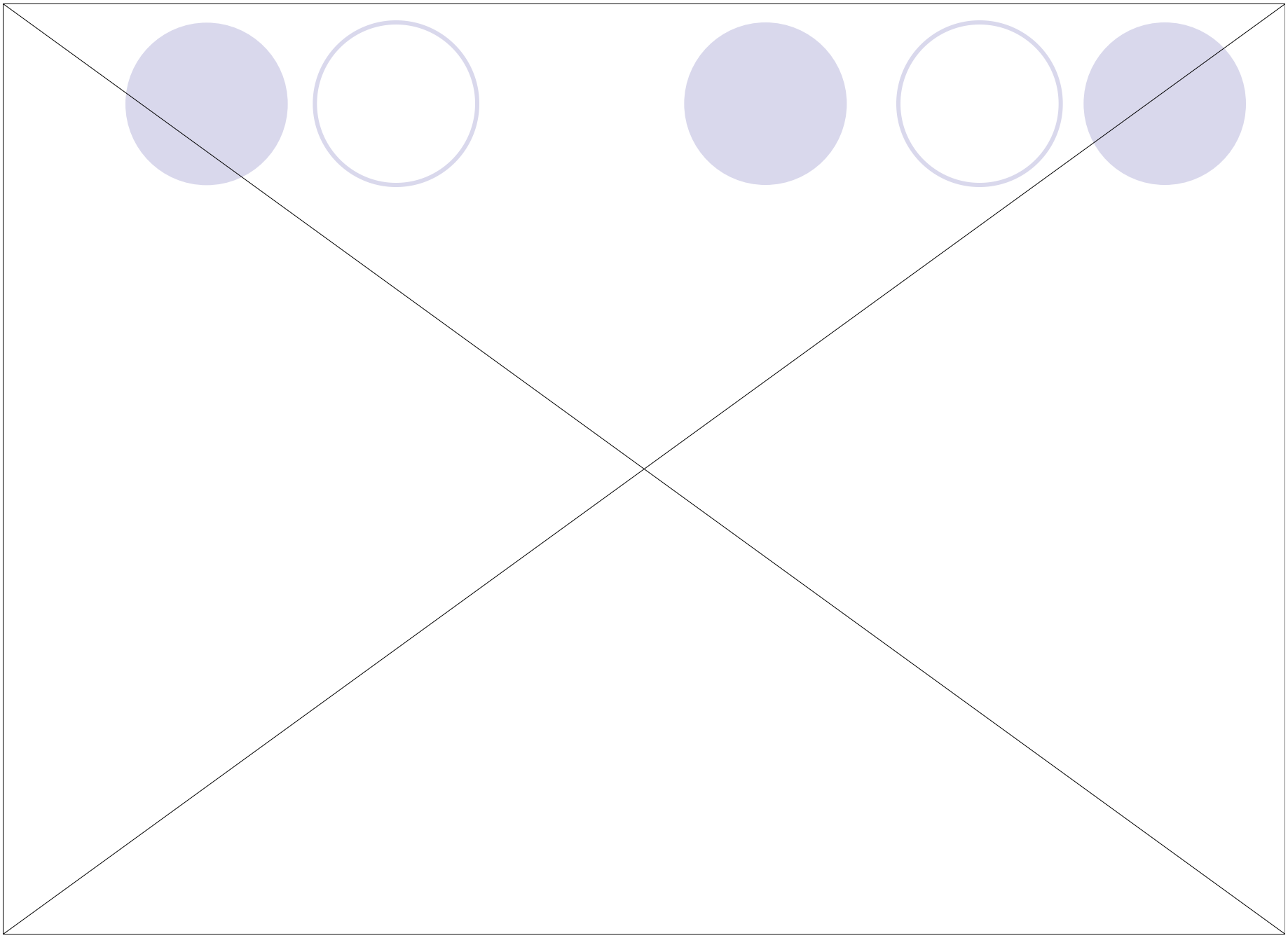
# 3. 二叉排序树的插入算法

- 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
```

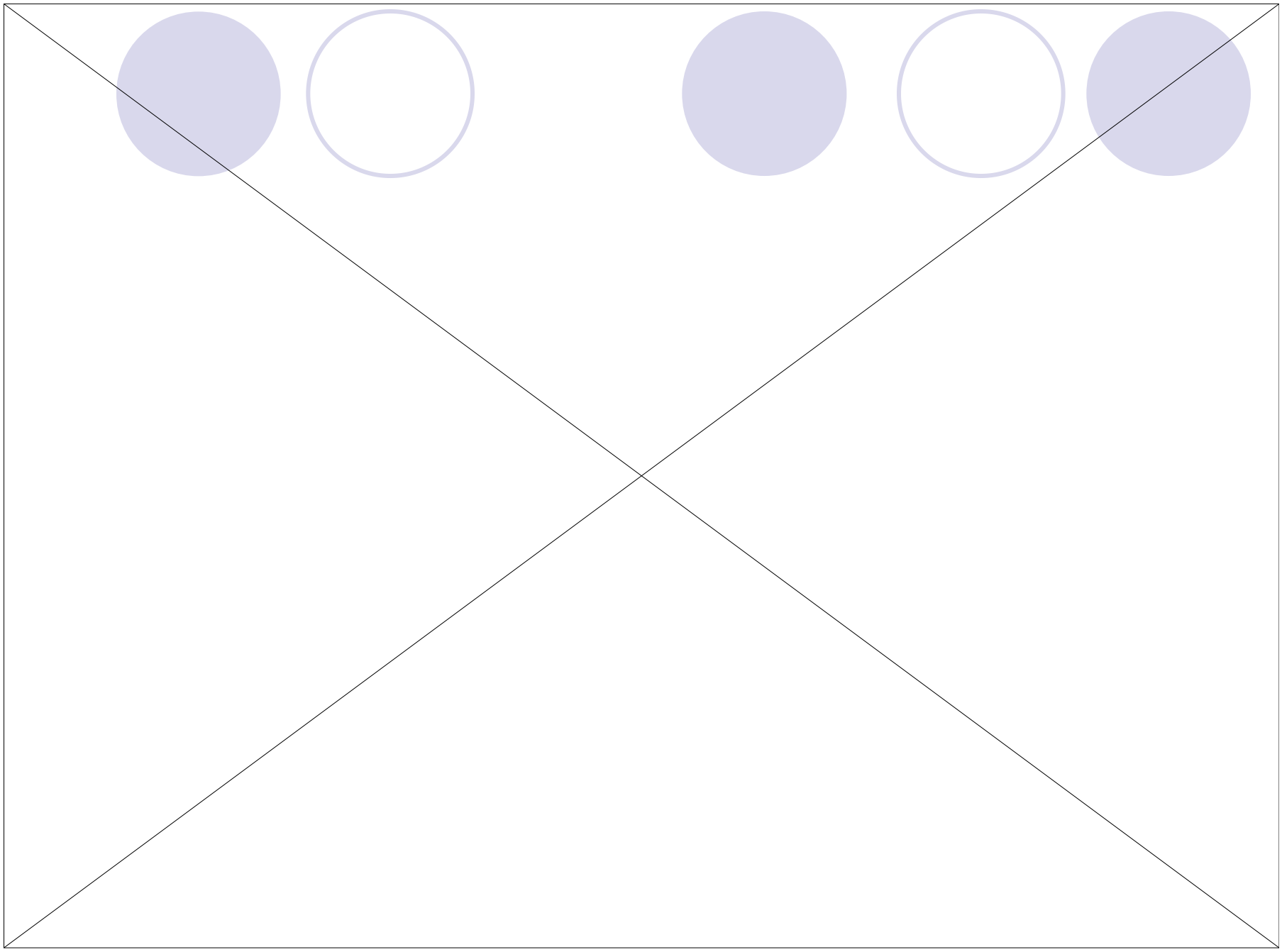


# 4. 二叉排序树的删除算法

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:

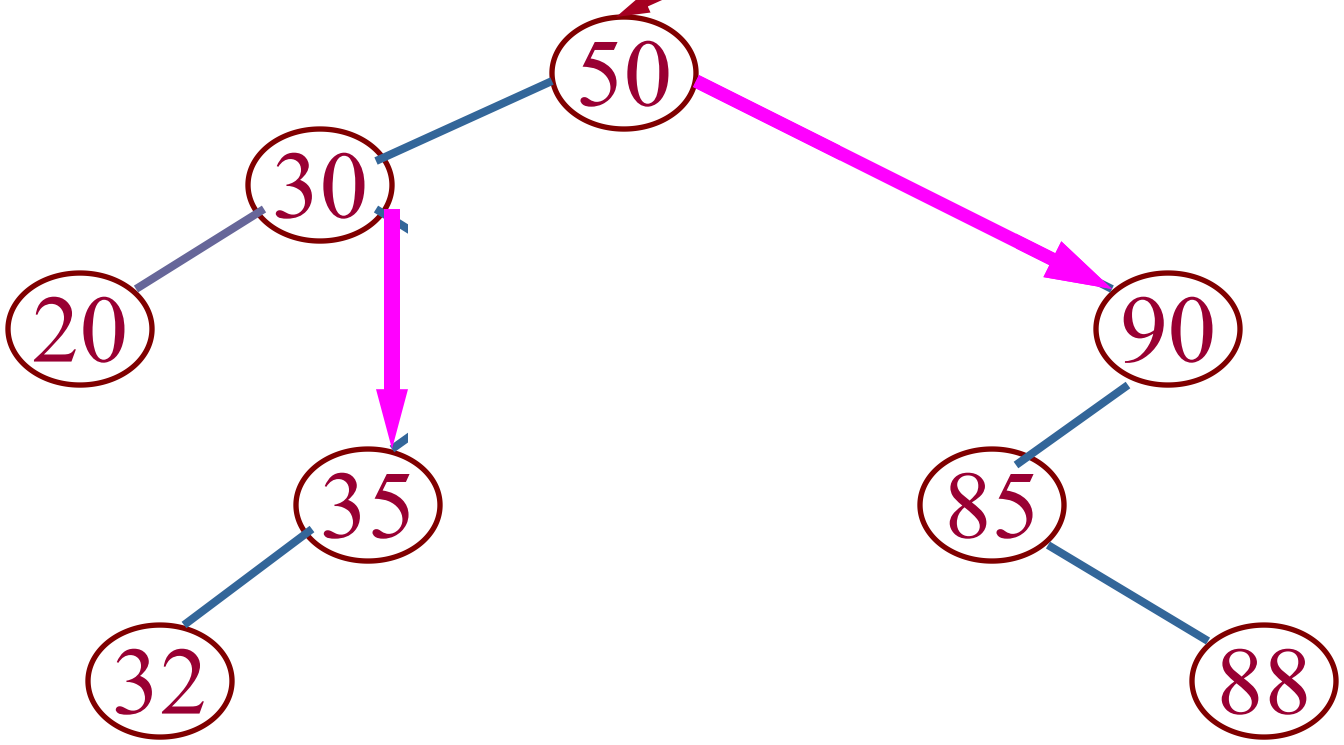
- ( 1) 被删除的结点是叶子;
- ( 2) 被删除的结点只有左子树或者只有右子树;
- ( 3) 被删除的结点既有左子树，也有右子树。



（ 2）被删除的结点只有左子树

或者只有右子树

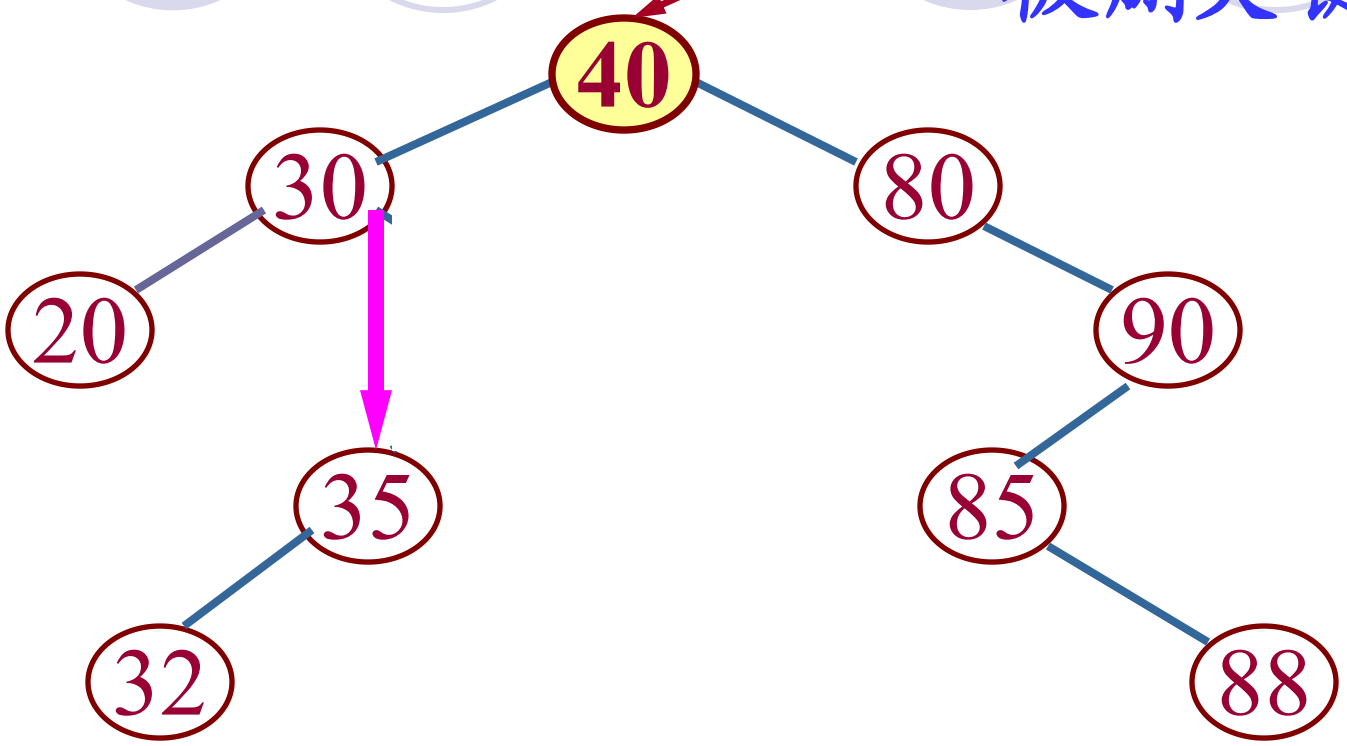
被删关键字 = 80

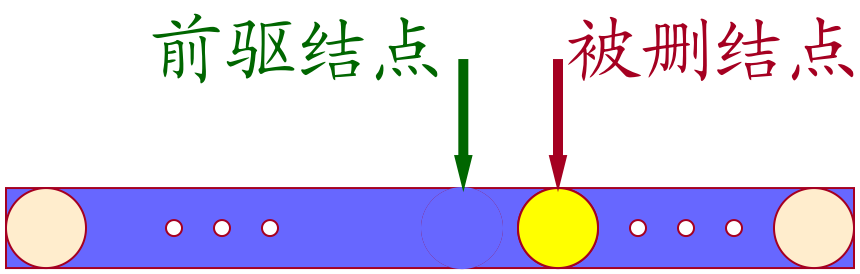



其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。

(3) 被删除的结点既有左子树，也有右子树

被删关键字 = 50

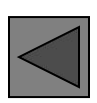



以其前驱替代之，然后再删除该前驱结点

## 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
```



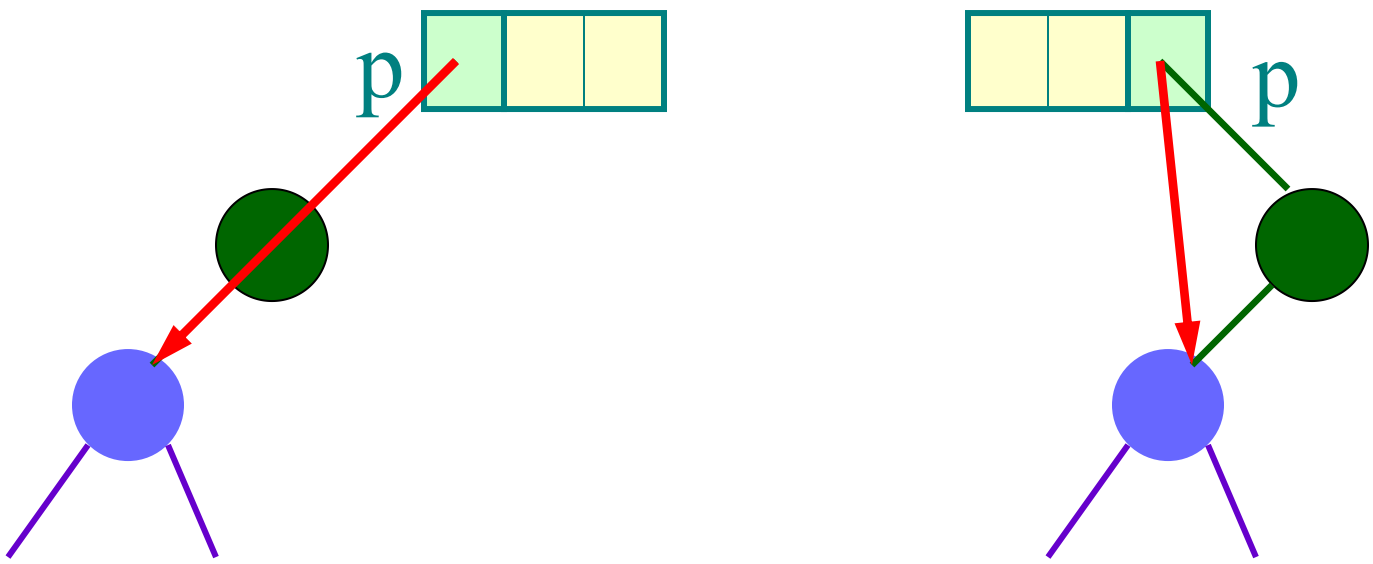
```
if ( EQ (key, T->data.key) )
    {  Delete (T);   return TRUE;  }
        // 找到关键字等于key的数据元素
else if ( LT (key, T->data.key) )
        DeleteBST ( T->lchild, key );
            // 继续在左子树中 进行查找
else   DeleteBST ( T->rchild, key );
            // 继续在右子树中 进行查找
```

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,
  // 并重接它的左子树或右子树
  if (!p->rchild) { … … }
  else if (!p->lchild) { … … }
  else { … … }
} // Delete
```

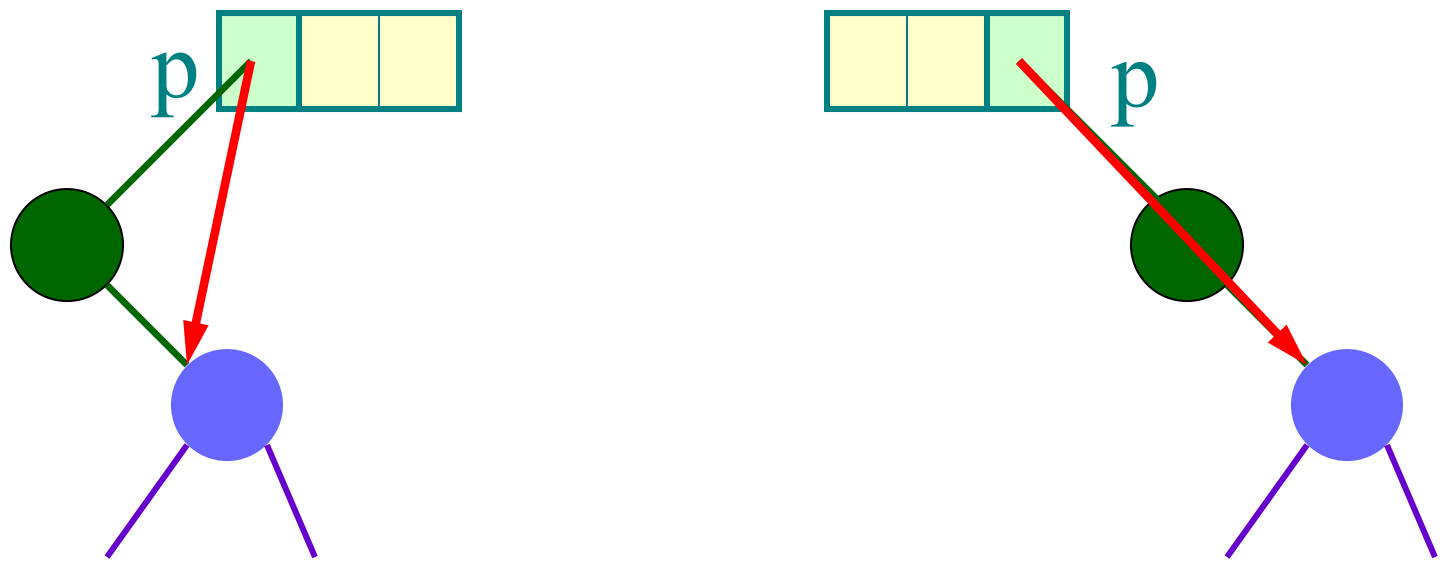
// 右子树为空树则只需重接它的左子树

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



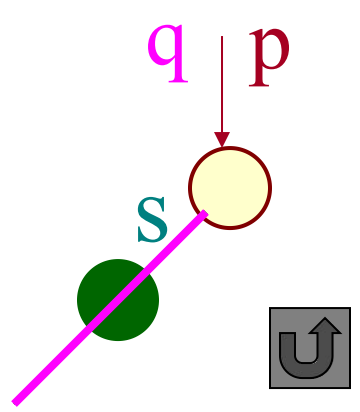
// 左子树为空树只需重接它的右子树

q = p;  p = p->rchild;  free(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
free(s);
```

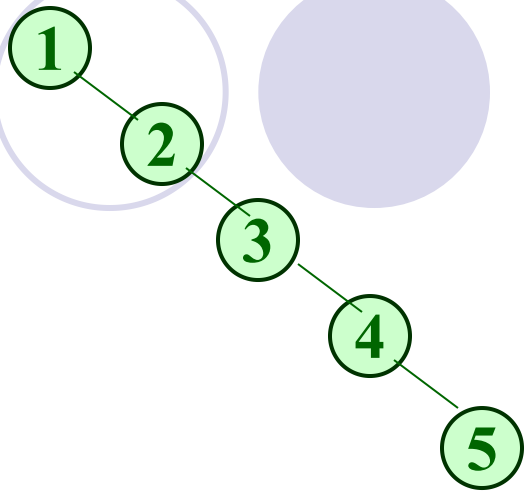


# 5. 查找性能的分析

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:**

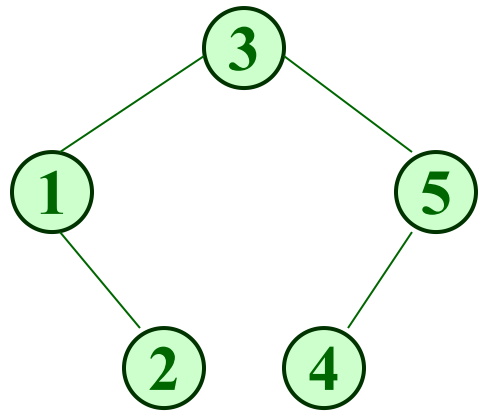
由关键字序列 **1, 2, 3, 4, 5**
构造而得的二叉排序树,




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

题4：分别以下列序列构造二叉排序树，与用其它三个序列所构造的结果不同的是_____ _.

A. （100，80， 90， 60， 120，110，130）

B.（100，120，110，130，80， 60， 90）

C.（100，60， 80， 90， 120，110，130）

D. (100，80， 60， 90， 120，130，110)

题5：如果按关键码值递增的顺序依次将n关键码值插入到二叉排序树中，则对这样的二叉排序树检索时，平均比较次数为_____。

题6. 在含有27个节点的二叉排序树上，查找关键字为35的节点，则一次比较的关键字有可能是_____。

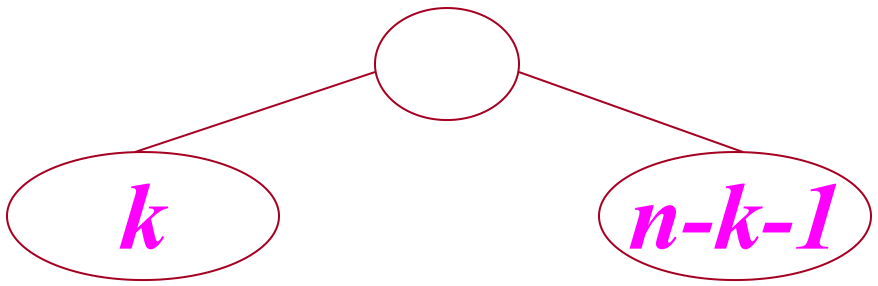
A.28, 36, 18, 46, 35　B. 18, 36, 28, 46, 35

C. 46, 28, 18, 36, 35　D. 46, 36, 18, 26, 35

题7. 用数据集合{10, 7, 18, 3, 8, 15, 16}按一定的次序插入构造一个排序二叉树，求对该排序二叉树进行中序遍历得到的序列。

## 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

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

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，

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

$$P(n,k) = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\left( C_{root} + \sum_{L} C_i + \sum_{R} C_i \right)$$

$$= \frac{1}{n}\left(1 + k(P(k)+1) + (n-k-1)(P(n-k-1)+1)\right)$$

$$= 1 + \frac{1}{n}\left(k \times P(k) + (n-k-1) \times P(n-k-1)\right)$$

由此

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

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

可类似于解差分方程，此递归方程有解:

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

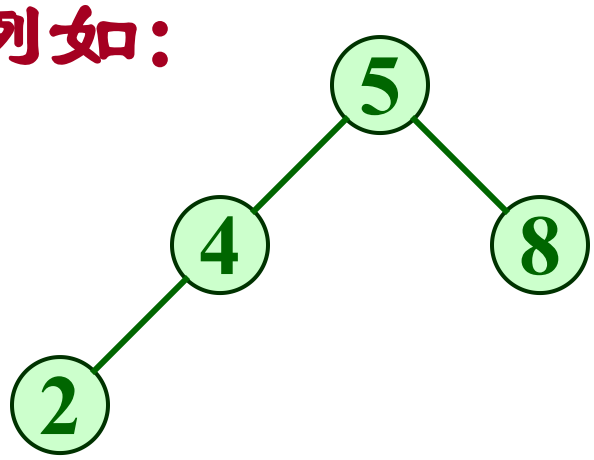
# 二、二叉平衡树

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

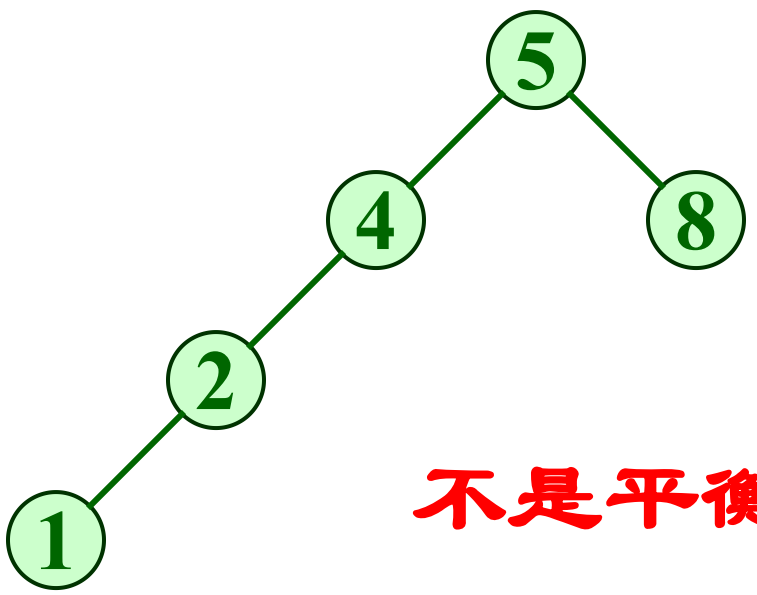
二叉平衡树是二叉查找树的另一种形式，其特点为：

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

例如：




是平衡树

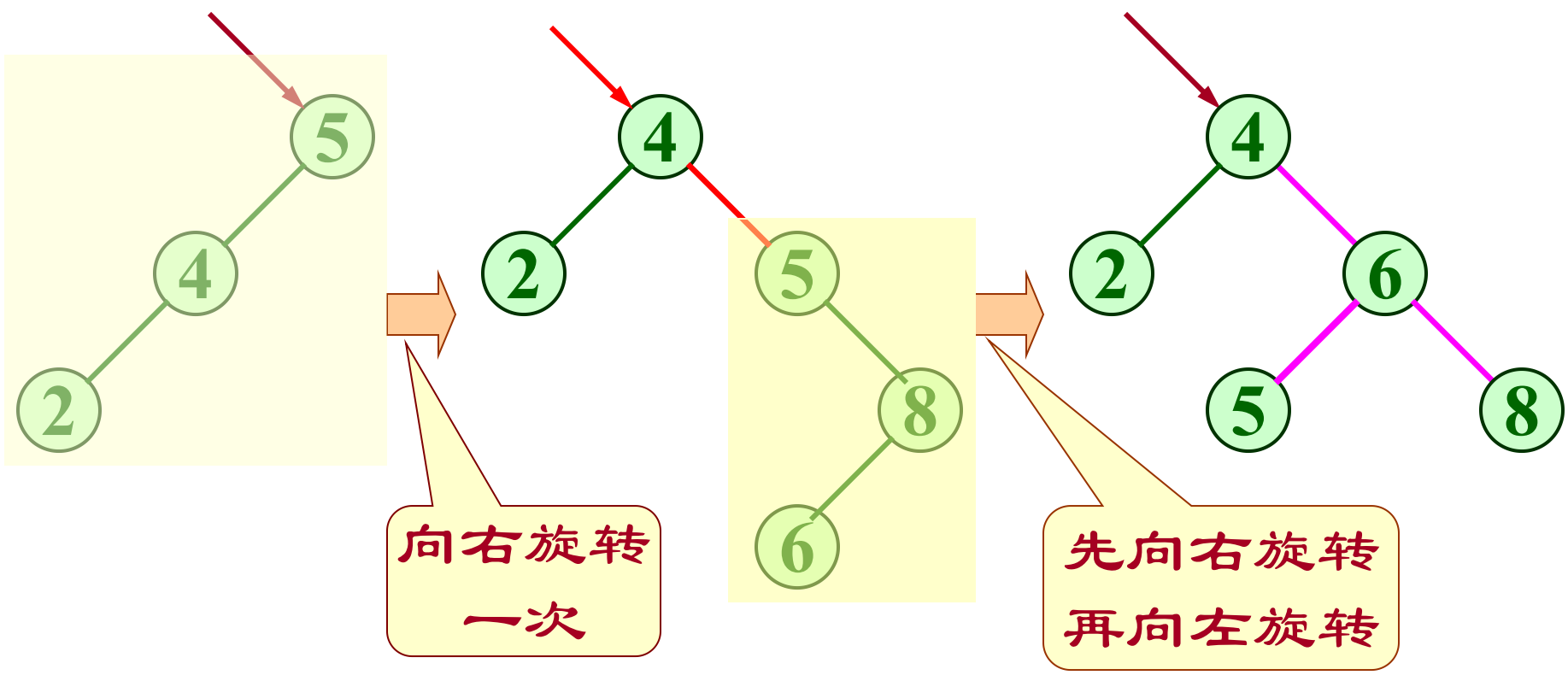


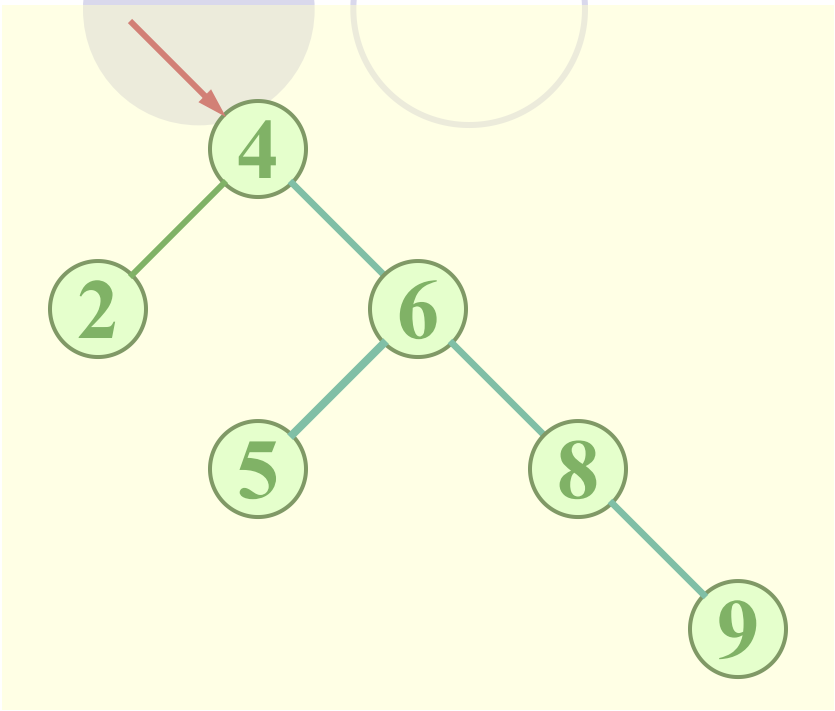

不是平衡树

# 构造二叉平衡（查找）树的方法是:

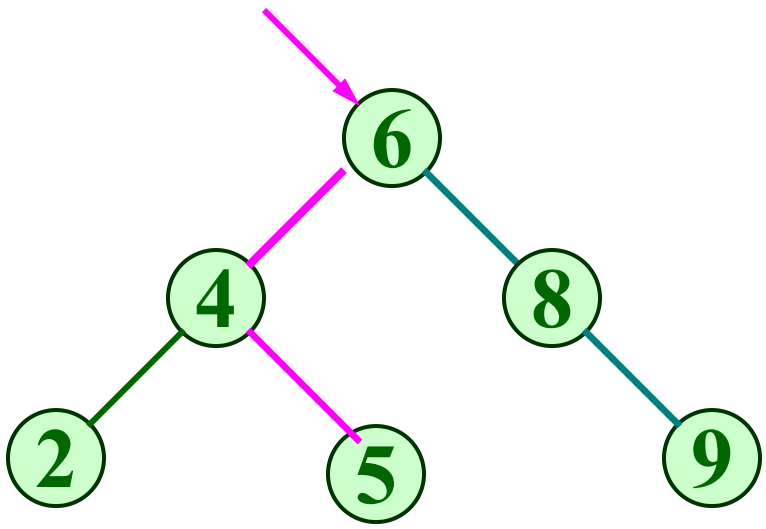
## 在插入过程中，采用平衡旋转技术。

例如:依次插入的关键字为5, 4, 2, 8, 6, 9



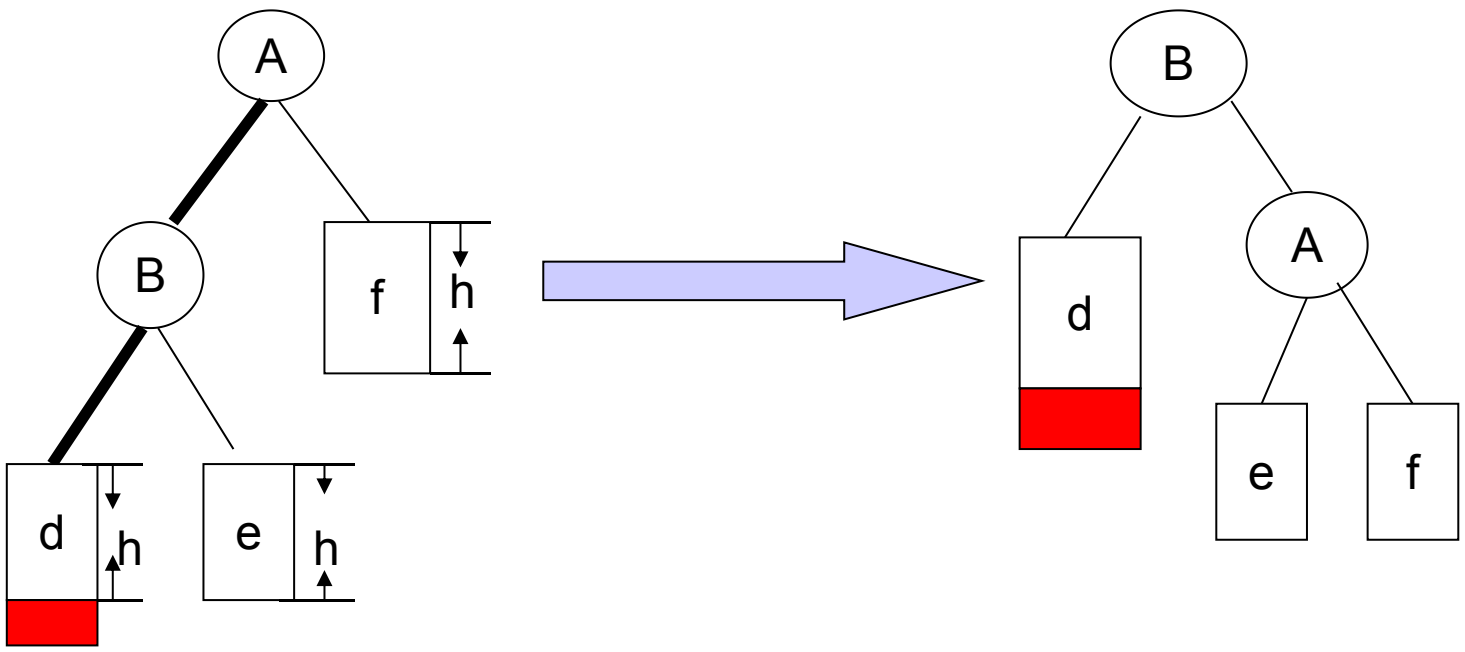



向左旋转一次

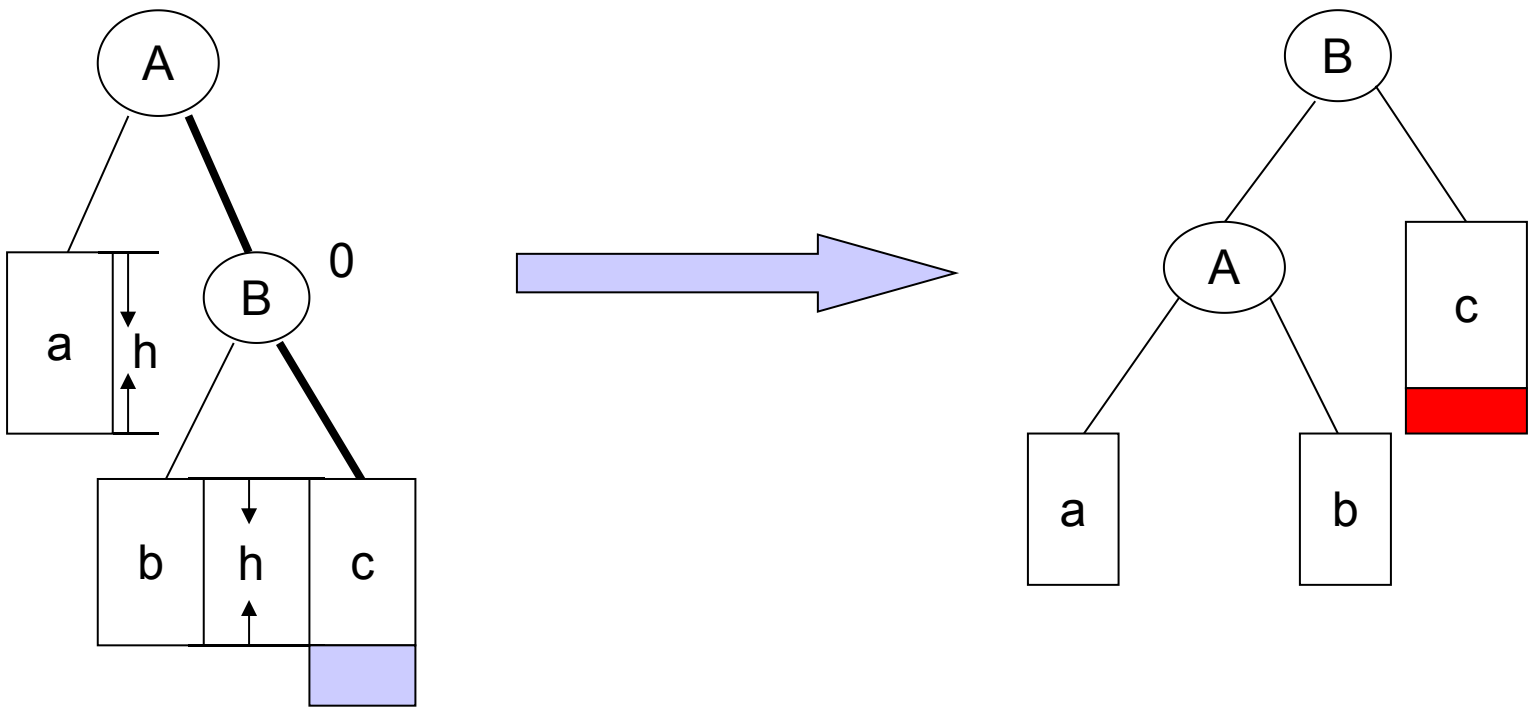



继续插入关键字 9

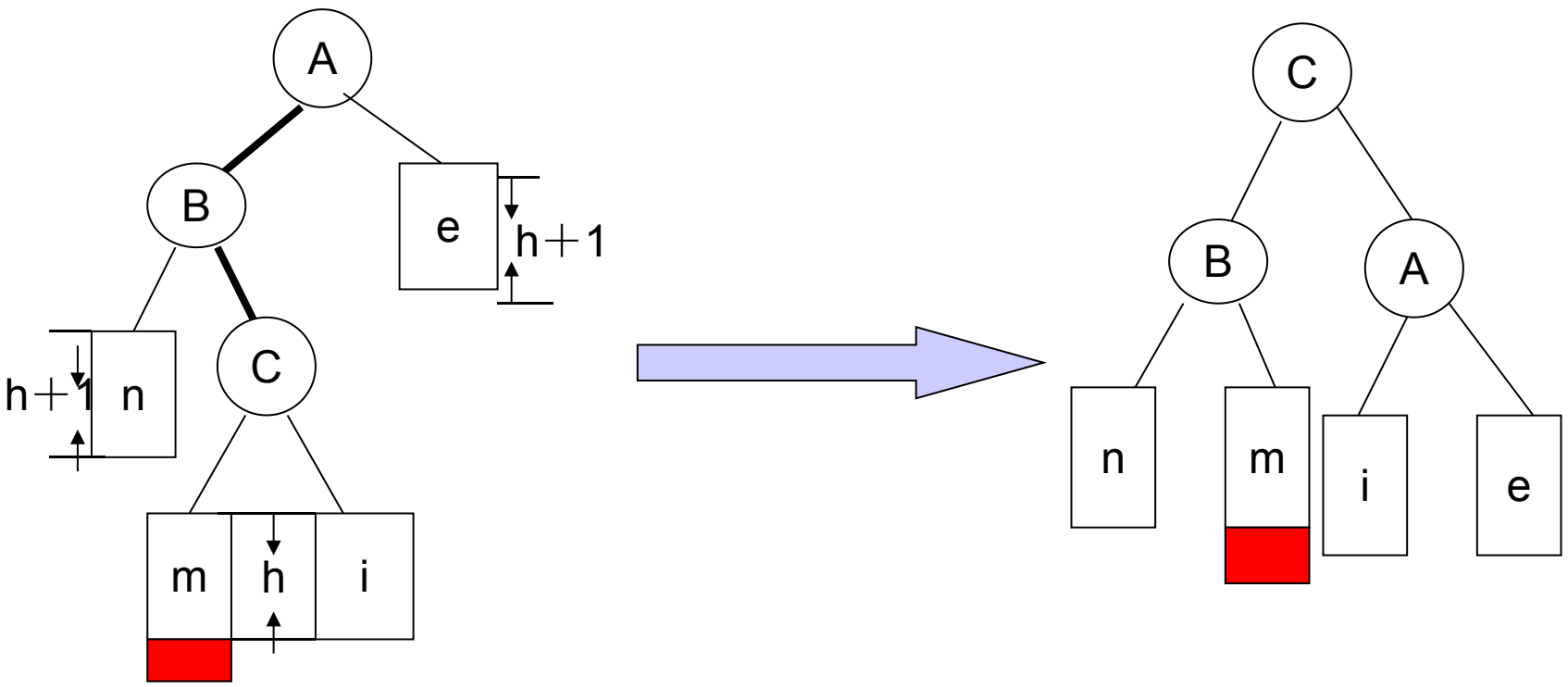
# LL调整



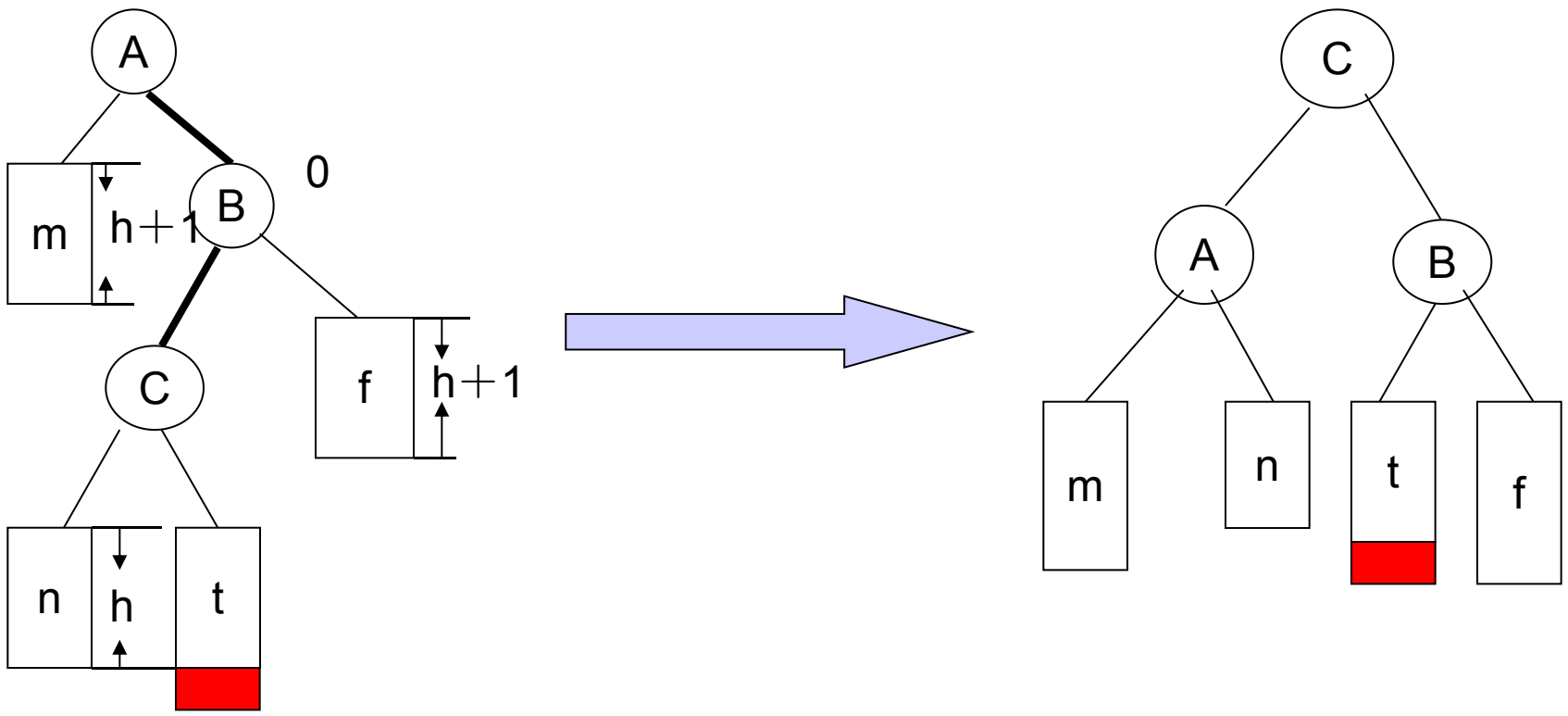
# RR调整



# LR调整

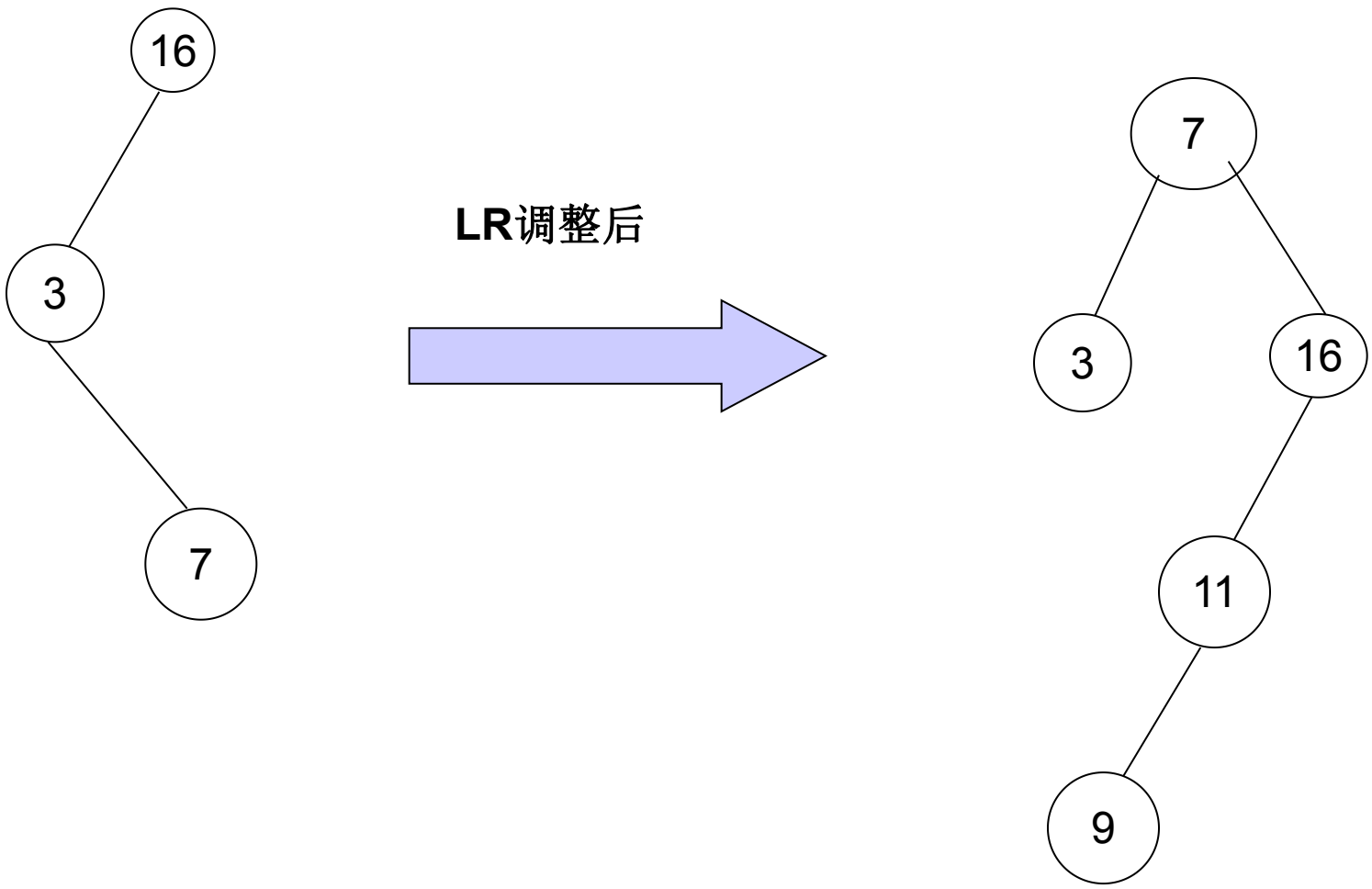


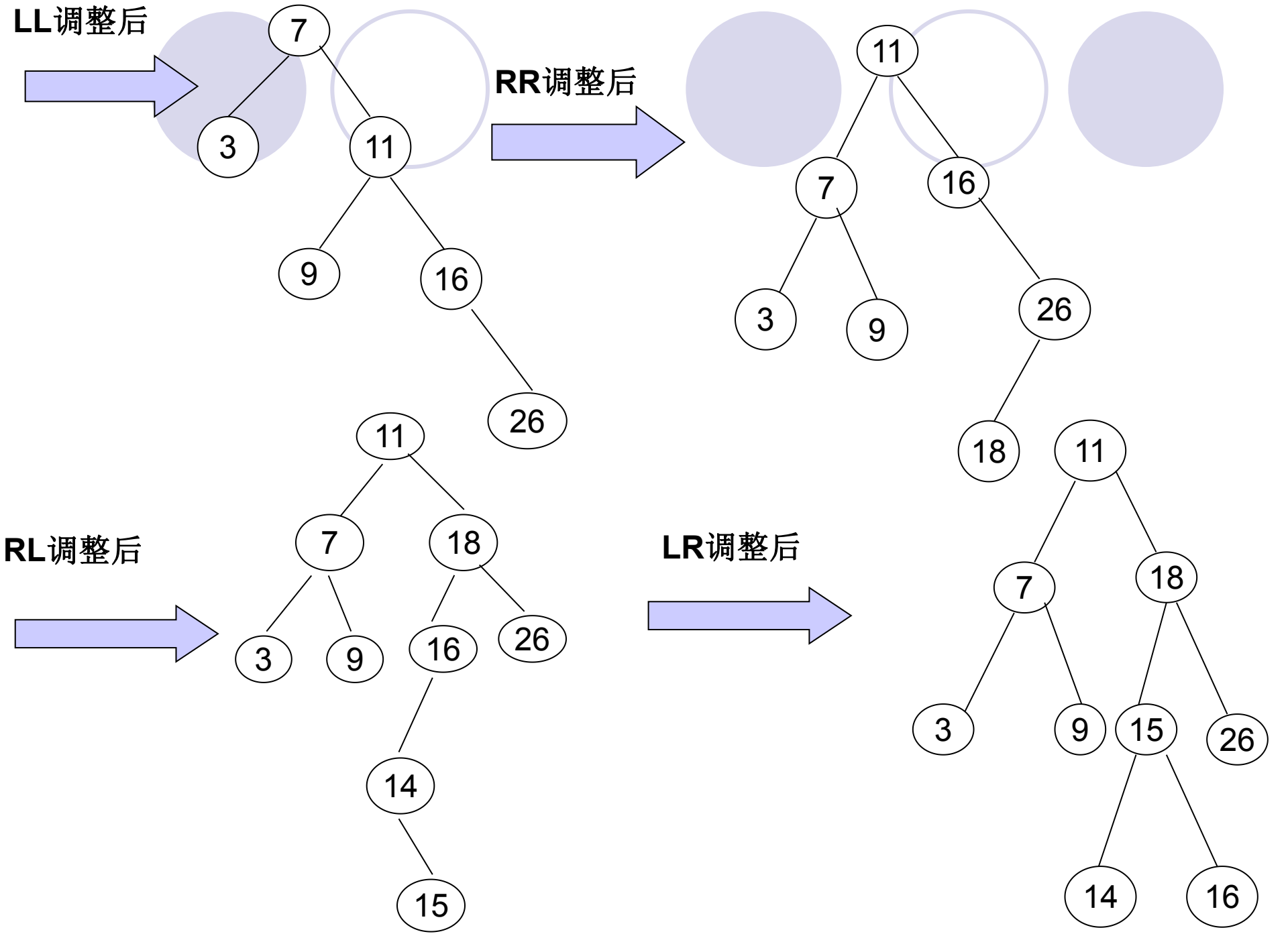
# RL调整



题8. 输入关键字序列{16,3,7,11,9,26,18,14,15},给出构造一棵AVL树的步骤。

{16, 3, 7, 11, 9, 26, 18, 14, 15}




LL调整后
7
3
11
9
16
26
RR调整后
11
7
16
3
9
26
18
RL调整后
11
7
18
3
9
16
26
14
15
LR调整后
11
7
18
3
9
15
26
14
16

# 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？

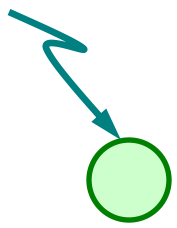
先看几个具体情况：

$n=0$

空树

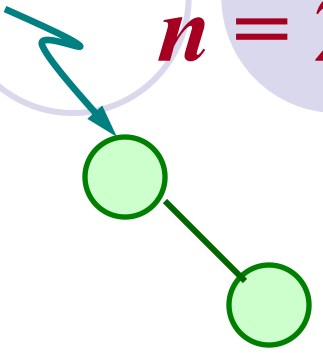
最大深度为 0

$n=1$



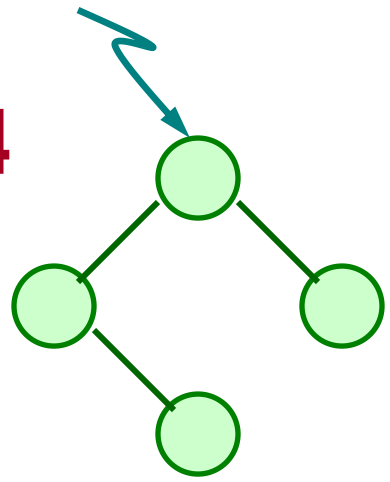
最大深度为 1

$n=2$



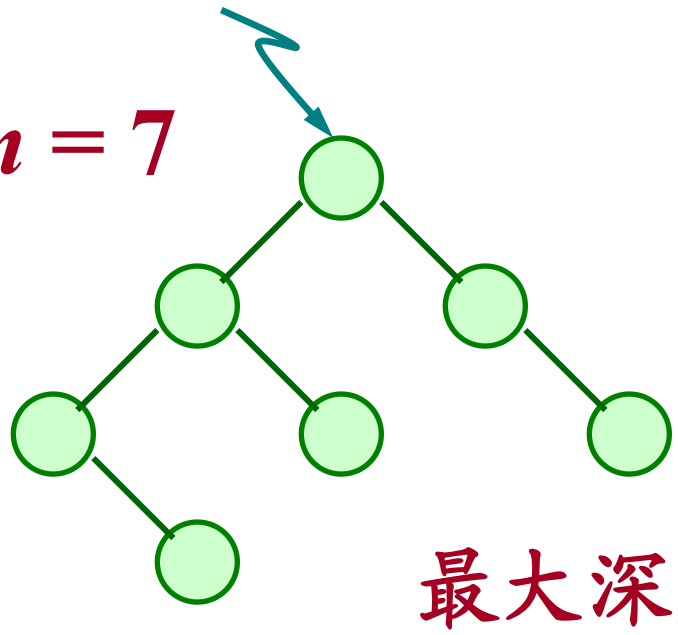
最大深度为 2

$n=4$



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？

$h=0$　　$N_0=0$　　$h=1$　　$N_1=1$

$h=2$　　$N_2=2$　　$h=3$　　$N_3=4$

一般情况下　　$N_h = N_{h-1} + N_{h-2} + 1$

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。

# 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**。

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：记录在表中位置和其关键字之间存在一种确定的关系。

**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～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) 为哈希函数。

例如：对于如下 9 个关键字

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

设哈希函数 f(key) =

⌊(Ord(第一个字母) -Ord('A')+1)/2⌋

| 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: 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**”。

# 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
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
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

$H(key) = key$　　或者

$H(key) = a \times key + b$

此法仅适合于：

**地址集合的大小 ＝＝关键字集合的大小**

# 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 $(u_1, u_2, \ldots, u_s)$，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**。

# 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**和**间界叠加**。

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：

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

其中，　$p \le m$（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?

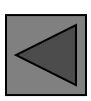
**例如:**

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：

$$H(key) = Random(key)$$

其中，Random为伪随机函数

通常，此方法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

“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**

**2. 链地址法**

# 1. 开放定址法

为产生冲突的地址 H(key) 求得一个

地址序列:

$H_0, H_1, H_2, \ldots, H_s \quad 1 \le s \le m-1$

其中： $H_0 = H(key)$

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

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

对增量 $d_i$ 有三种取法：

- 1) 线性探测再散列

  $d_i = c \times i$ 最简单的情况 $c=1$

- 2) 二次探测再散列

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

- 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)

## 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：

※ 二次探测时的表长 ***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. 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。


例如:同前例的关键字，哈希函数为

H(key)=key MOD 7

ASL=(6×1+2×2+3)/9=13/9

## 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] = 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。

```
//--- 开放定址哈希表的存储结构 ---
int  hashsize[] = { 997, ... };
typedef struct {
  ElemType  *elem;
  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);       // 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
    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, Elemtype e){
    c = 0;
    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，（ 阀值 c 可调）
                H.elem[p] = e;  ++H.count;  return OK;
    }           // 查找不成功时， 返回 p为插入位置
    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。

## 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**

$\alpha$ =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。**

例如：前述例子

线性探测处理冲突时，　ASL = 22/9

双散列探测处理冲突时，ASL = 14/9

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：

**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 **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。

- 有关键字集合K={15，22，50，13，20，36，28，48，35，31，41，18}采用散列存取，散列表为HT[0：14]。设散列函数H（K）=K MOD 13，解决冲突采用开发地址法中的二次探测再散列的方法。试将k值填入HT表中，并把查找每个关键字所需的比较次数M填入表中，并计算出查找成功时的平均查找长度。

| I | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | 13 | | | | | | | | | | | | | | |
| M | | | | | | | | | | | | | | | |

**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.** 静态查找树的构造方法和查找算法，理解静态查找树和折半查找的关系。

**3.** 熟练掌握**二叉排序树**的构造和查找方法。

**4.** 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。

**5.** 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

- 1.画出对长度为 10 的有序表进行折半查找的判定树，并求其等概时查找成功的平均查找长度。

- 2.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，请画出插入完成之后的二叉排序树，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，求在等概率的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.选取哈希函数 H(k)=(3k) MOD 11。用二次探测再散列法处理冲突。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)造哈希表，并求等概情况下查找成功时的平均查找长度。