

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，则称之为谓 “**主关键字**”。

若此关键字能识别**若干**记录，则称之为谓 “**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。

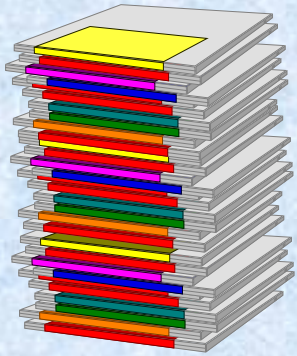
为了提高查找的效率，需要在查找表中的元素之间人为地附加某种确定的关系，换句话说，**用另外一种结构来表示查找表。**

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。

基本操作 P:

Create(&ST, n);

Destroy(&ST);

Search(ST, key);

Traverse(ST, Visit());

} ADT StaticSearchTable

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



一、顺序查找表

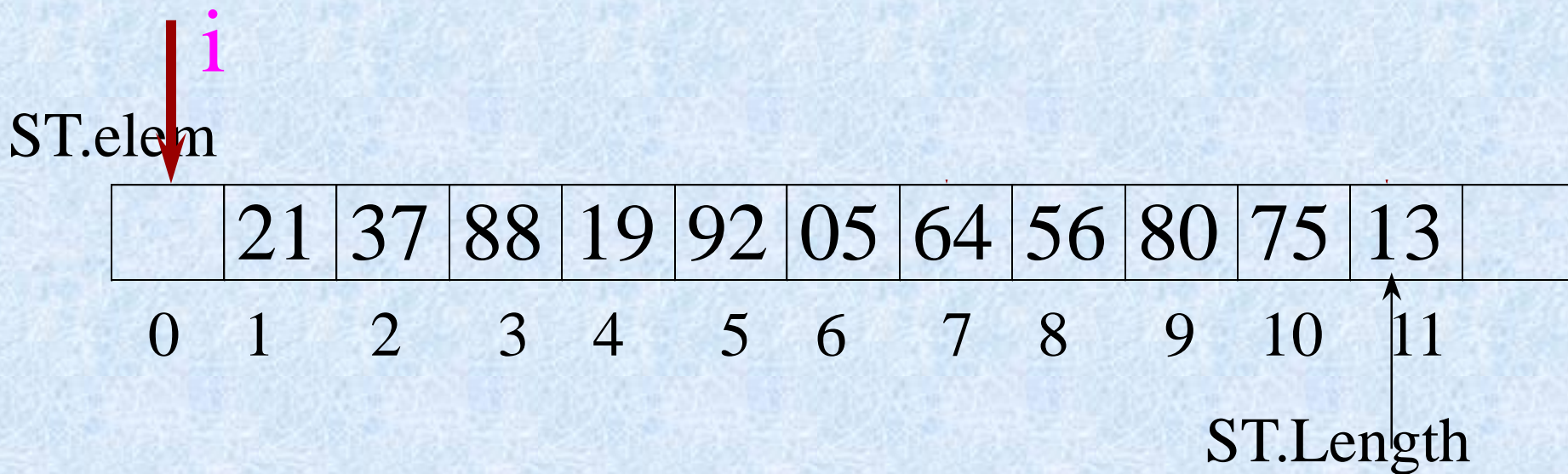
二、有序查找表

三、索引顺序表



一、顺序查找表

如果数组中关键字的排列是无序的，只能进行逐个查找。可采用顺查或逆查。



假设给定值 $e = 66$

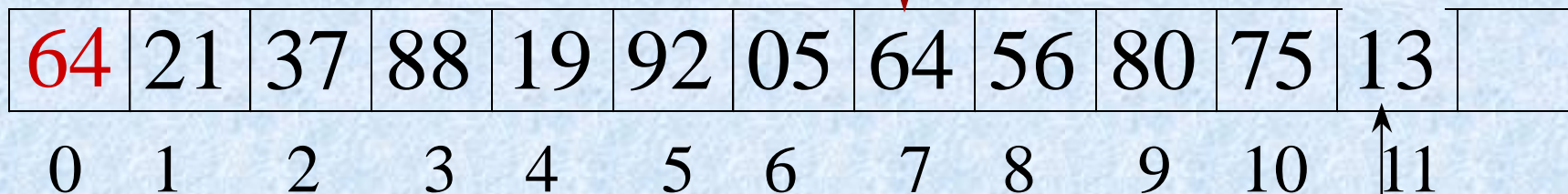
要求 $ST.elem[i] = e$, 问: $i = ?$

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

改进:

在ST.elem[0]处设立监视哨, 可以减少查找时间

ST.elem

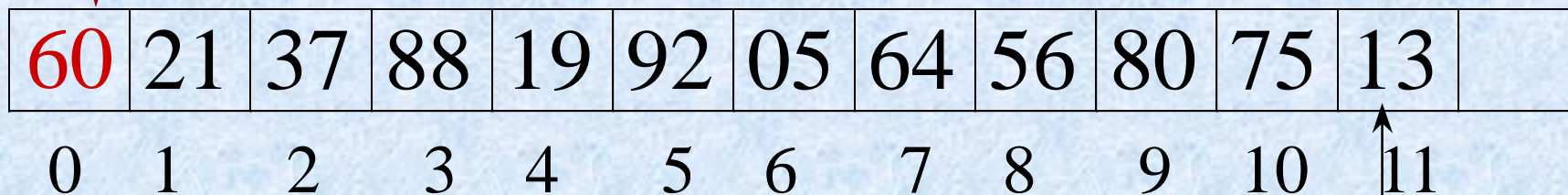


64	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

kval = 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	

kval = 60

ST.Length

```
int Search_Seq(SSTable ST,  
                KeyType kval) {  
    // 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 = kval;    // 设置 “哨兵”  
    for (i=ST.length; ST.elem[i].key!=kval; --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**

(Average Search Length)

为确定记录在查找表中的位置，需和给定值
进行比较的关键字个数的平均值

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

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, $C_i = n-i+1$

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

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:

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

查找不成功时的比较次数: $n+1$

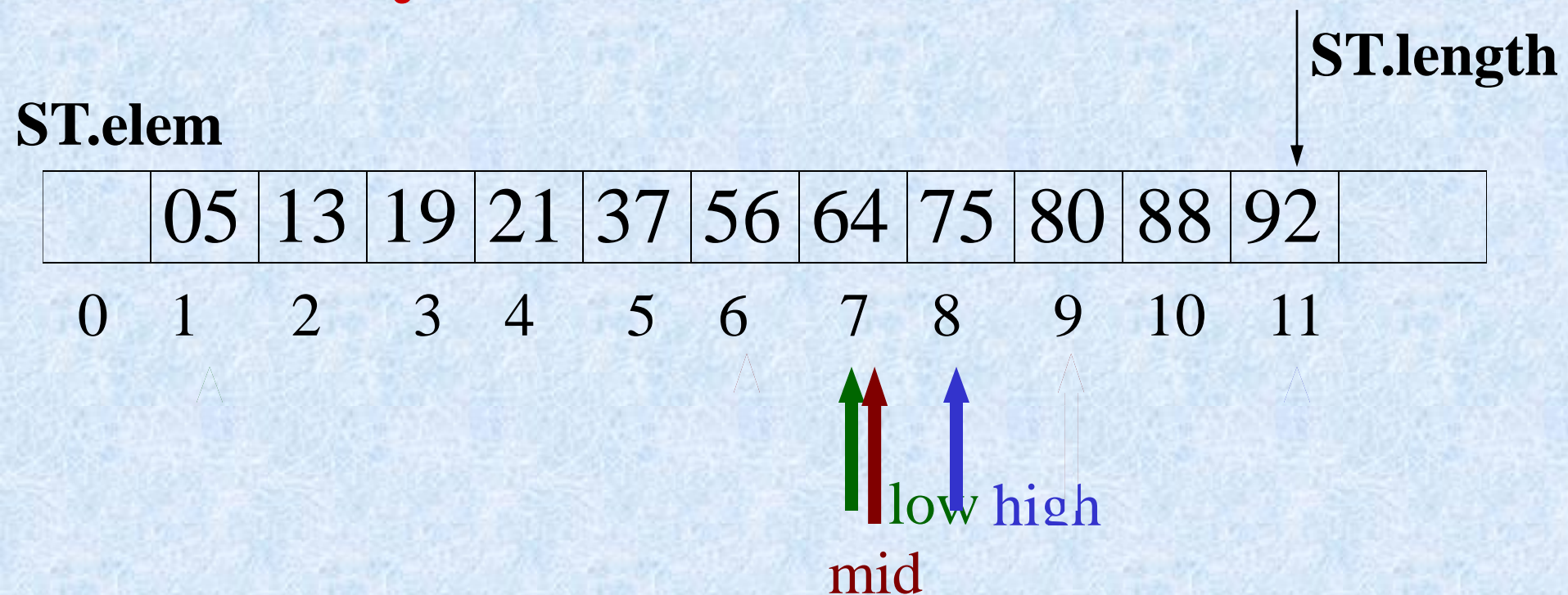
二、有序查找表

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。

有序表：表中的记录按关键字**有序排列**。

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 = $(\text{low} + \text{high}) / 2$ 。

折半查找算法：

假设有一序列存在数组 r 中，指针 low 和 $high$ 分别指示查找范围的下界和上界， mid 指示区间的中间位置，即

$$mid = (low + high) / 2$$

首先令 $r[mid].key$ 与给定值 k 相比较，
如果

(1) $r[mid].key = k$ ，则查找成功。（退出）

(2) $k < r[mid].key$ ，

则在 $[low, mid-1]$ 范围内查找。

(3) $k > r[mid].key$ ，

则在 $[mid+1, high]$ 范围内查找

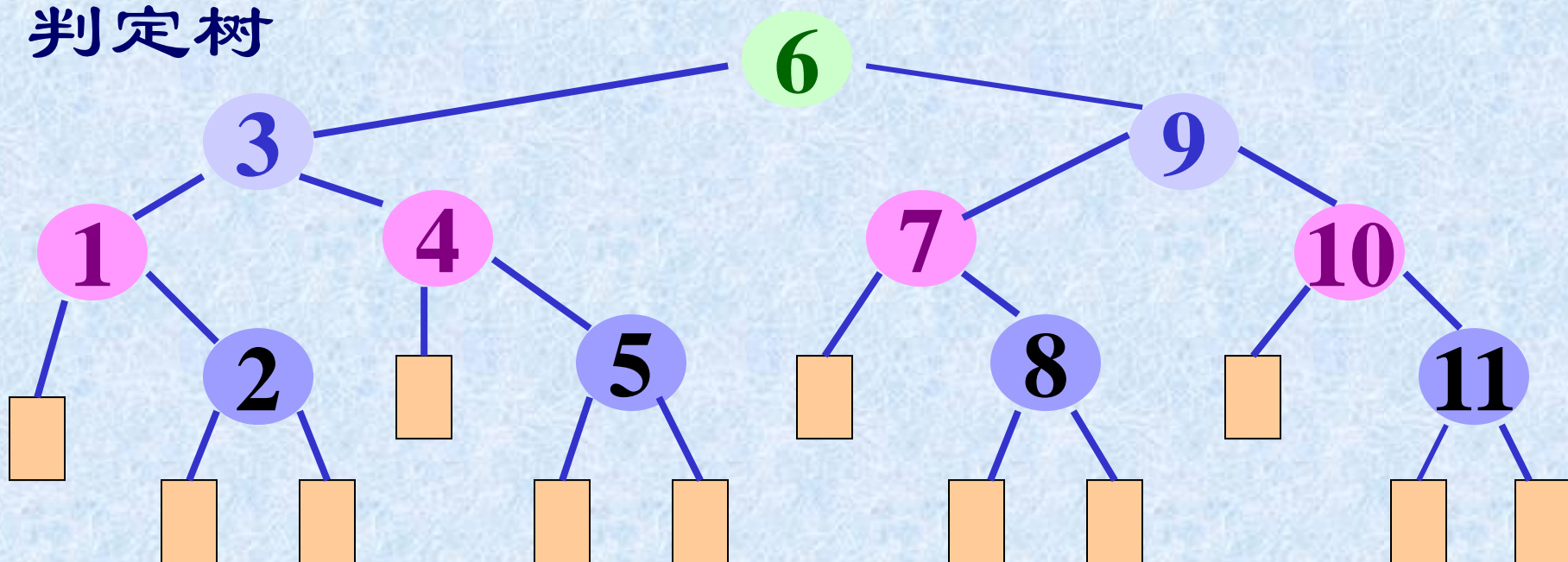

```
int Search_Bin ( SSTable ST, KeyType kval ) {  
    low = 1; high = ST.length;    // 置区间初值  
    while (low <= high) {  
        mid = (low + high) / 2;  
        if (kval == ST.elem[mid].key )  
            return mid;    // 找到待查元素  
        else if ( kval < ST.elem[mid].key )  
            high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


分析折半查找的平均查找长度

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



$$ASL_{bs} = (1*1 + 2*2 + 3*4 + 4*4) / 11 = 33 / 11 = 3$$

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

查找成功的比较次数至多为 $\lfloor \log_2 n \rfloor + 1$ 。

查找不成功时的比较次数也不超过 $\lfloor \log_2 n \rfloor + 1$ 。

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

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

三、索引顺序表

索引顺序查找也称**分块查找**——顺序查找的改进条件：

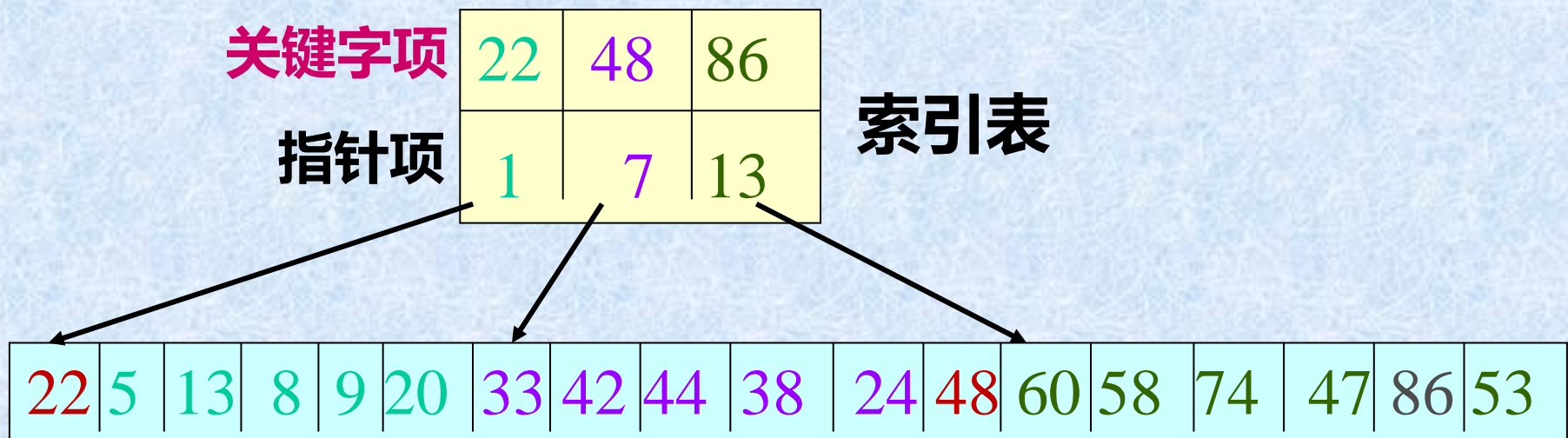
把线性表顺序划分为若干个子表(块)后满足：

- (1) 子表之间递增(或递减)有序——后一子表的每一项大于前一子表的所有项；
- (2) 块内元素可以无序。

表结构的建立：

- ① 把线性表均匀划分为若干个子表(块)，使子表之间有序；
- ② 建立**索引表**(有序表)，结点结构：

{ **关键字项** — 每个子表元素的最大值
指针项 -- 子表中第一个元素在线性表中的位置。



n — 表长 (18)

s — 块长 (6)

b — 块数 (3)

若不能均匀划分线性表，则索引表中应增加子表长度信息。

查找过程分两步进行：

(1) 确定待查记录所在的块。

(2) 在相应的块内查找

$$ASL_{bs} = L_b + L_s$$

索引表的平均查找长度

块内的平均查找长度

如整个表长为 n ，均匀地分成 b 块，每块含有 s 个记录，即 $b = \lceil n/s \rceil$ 。

假设表中每个记录的查找概率相等，则

- (1) 每块的查找概率是 $1/b$ 。
- (2) 块内每个记录的查找概率为 $1/s$ 。

若都采用顺序查找，则

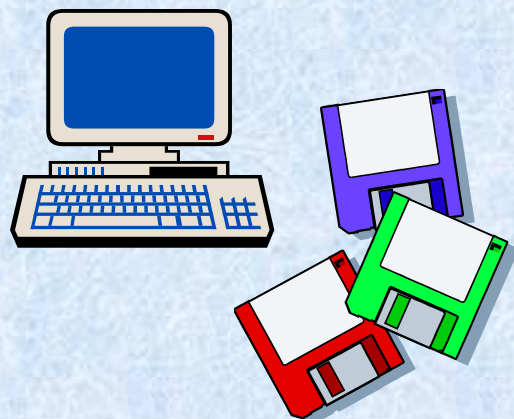
$$\begin{aligned} \text{ASL}_{bs} &= L_b + L_s \\ &= \frac{1}{b} \sum_{j=1}^b j + \frac{1}{s} \sum_{i=1}^s i \\ &= (b+1)/2 + (s+1)/2 \\ &= (b+s)/2 + 1 \\ &= \frac{1}{2} \left(\frac{n}{s} + s \right) + 1 \end{aligned}$$

当s取 \sqrt{n} 时， ASL_{bs} 取最小值。

如果n=400, 则b=20, s=20时，查找性能最佳。

9.2

动态查找树表



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	$O(\log n)$	$O(n)$	$O(n)$
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 $O(\log n)$ ，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。

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

next

一、二叉排序树（二叉查找树）

二、二叉平衡树



一、二叉排序树

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

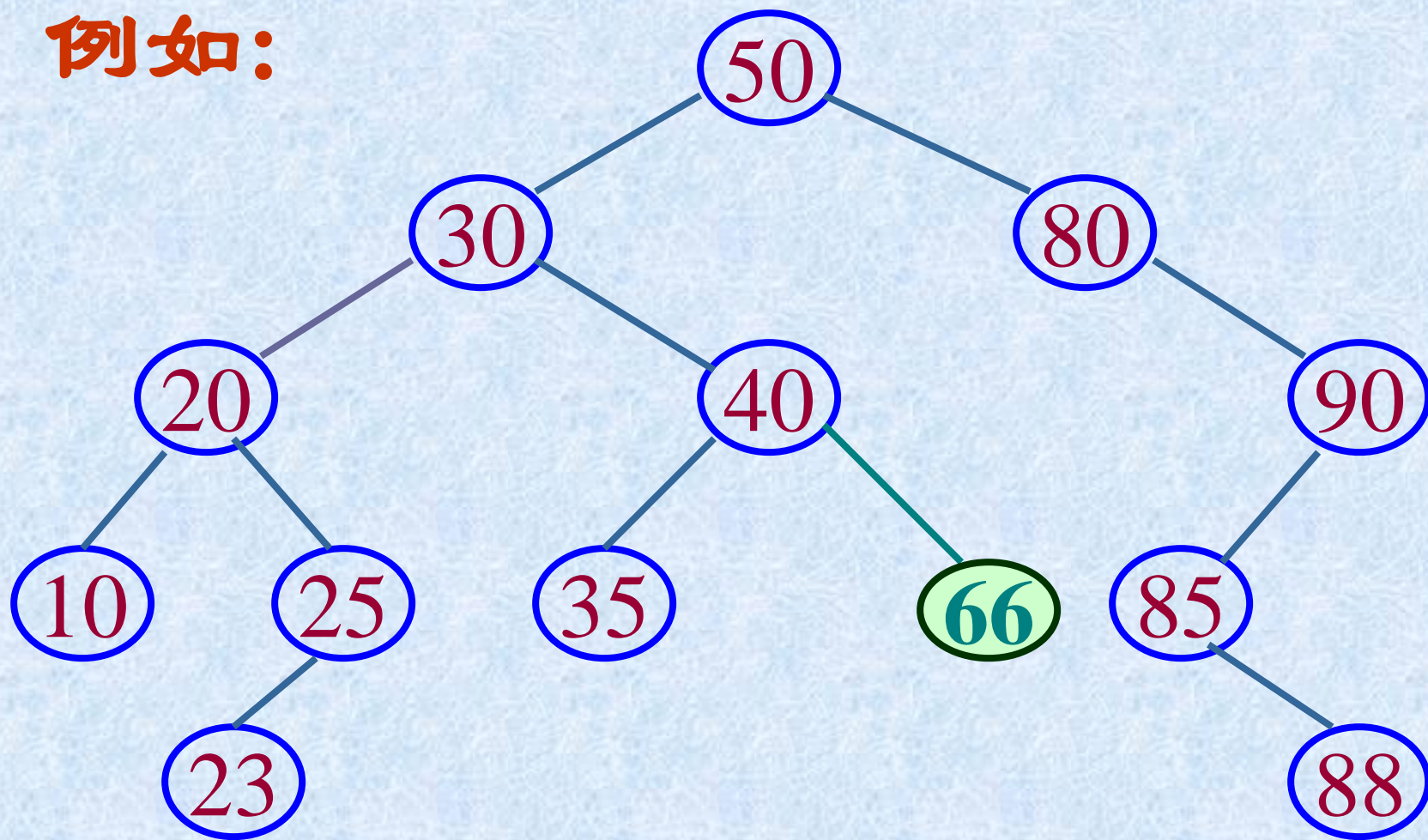


1. 定义:

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。

对二叉排序树进行中序遍历，得到一个有序序列

通常，取二叉链表作为
二叉排序树的存储结构

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

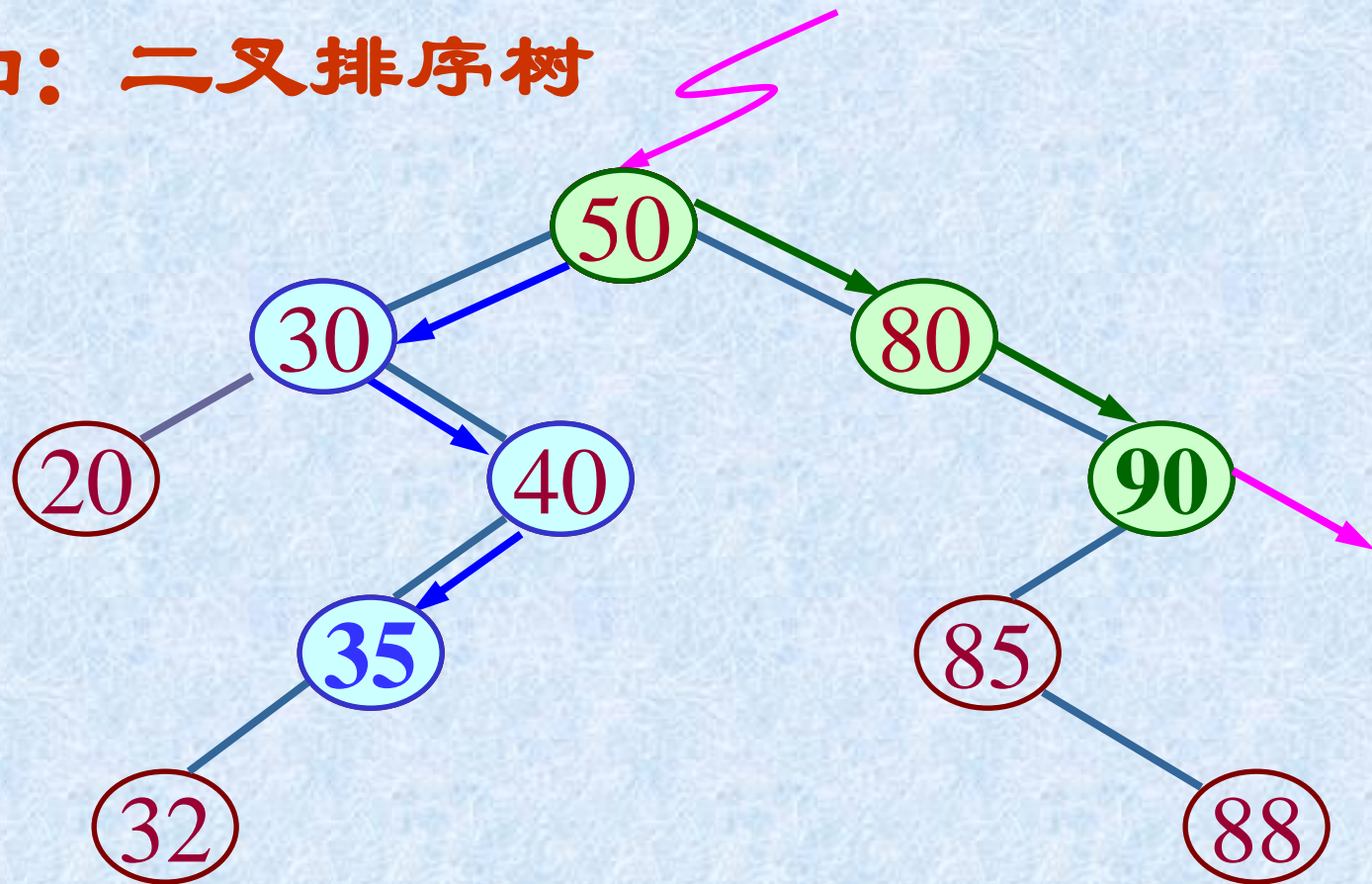


2. 二叉排序树的查找算法：

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。**

——查找失败

算法描述如下:

```
Status SearchBST (BiTree T, KeyType kval,  
                  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 kval 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  
    // 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(kval, T->data.key))

{ p = T; return TRUE; } // 查找成功

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

return SearchBST (T->lchild, kval, T, p);

// 在左子树中继续查找

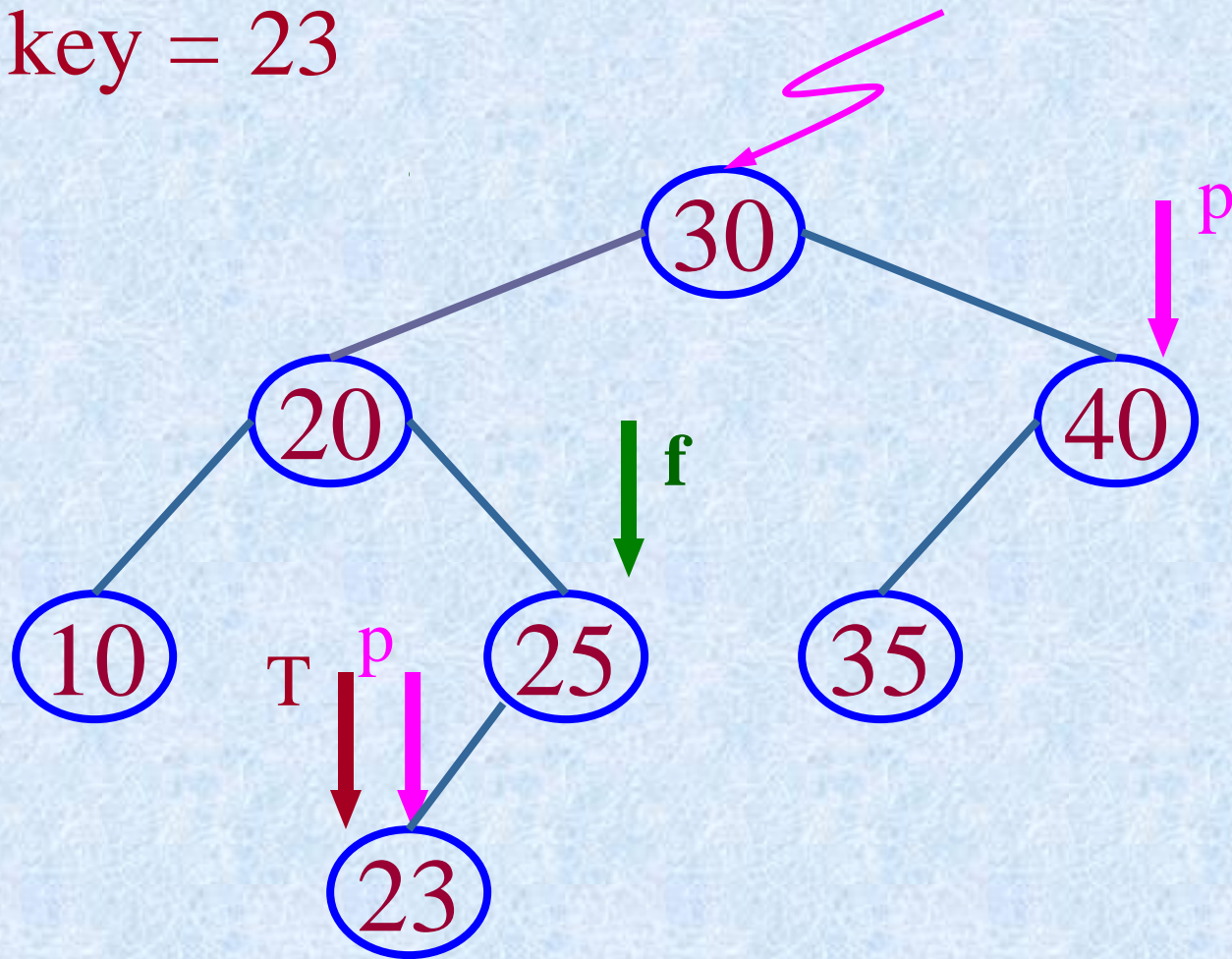
else

return SearchBST (T->rchild, kval, T, p);

// 在右子树中继续查找



设 $\text{key} = 23$



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 InsertBST(BiTree &T, ElemType e)

{

**// 当二叉排序树中不存在关键字等于 e.key 的
// 数据元素时，插入元素值为 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 = new 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
```



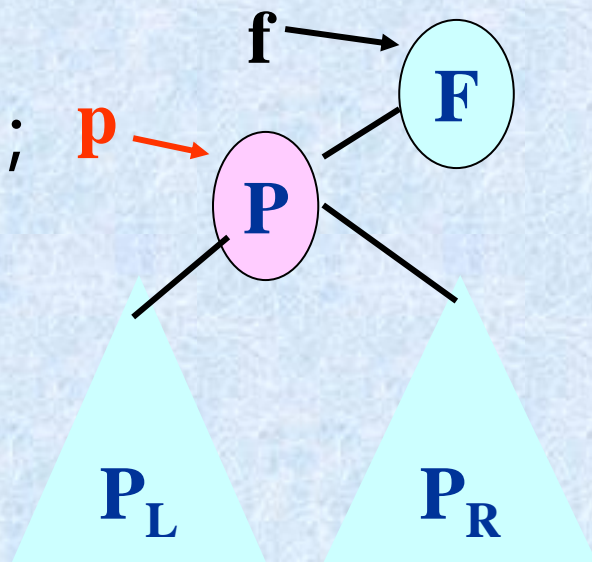
4. 二叉排序树的删除算法

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。**

设被删除结点为P，由指针p指向；

P的双亲结点为F，由指针f指向；

可分三种情况讨论：

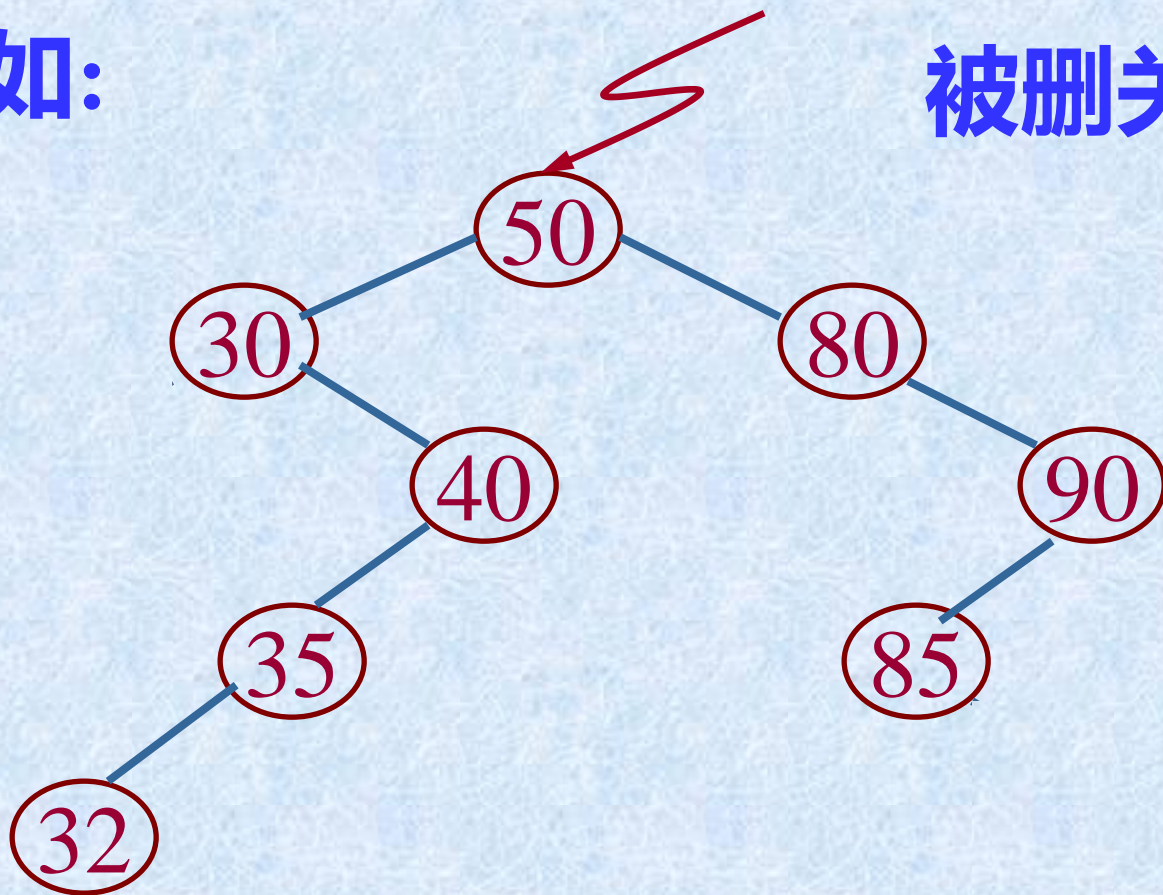


- (1) 被删除的结点是**叶子**；
- (2) 被删除的结点**只有左子树**或者**只有右子树**；
- (3) 被删除的结点**既有左子树，也有右子树**。

(1) 被删除的结点是**叶子结点**

例如:

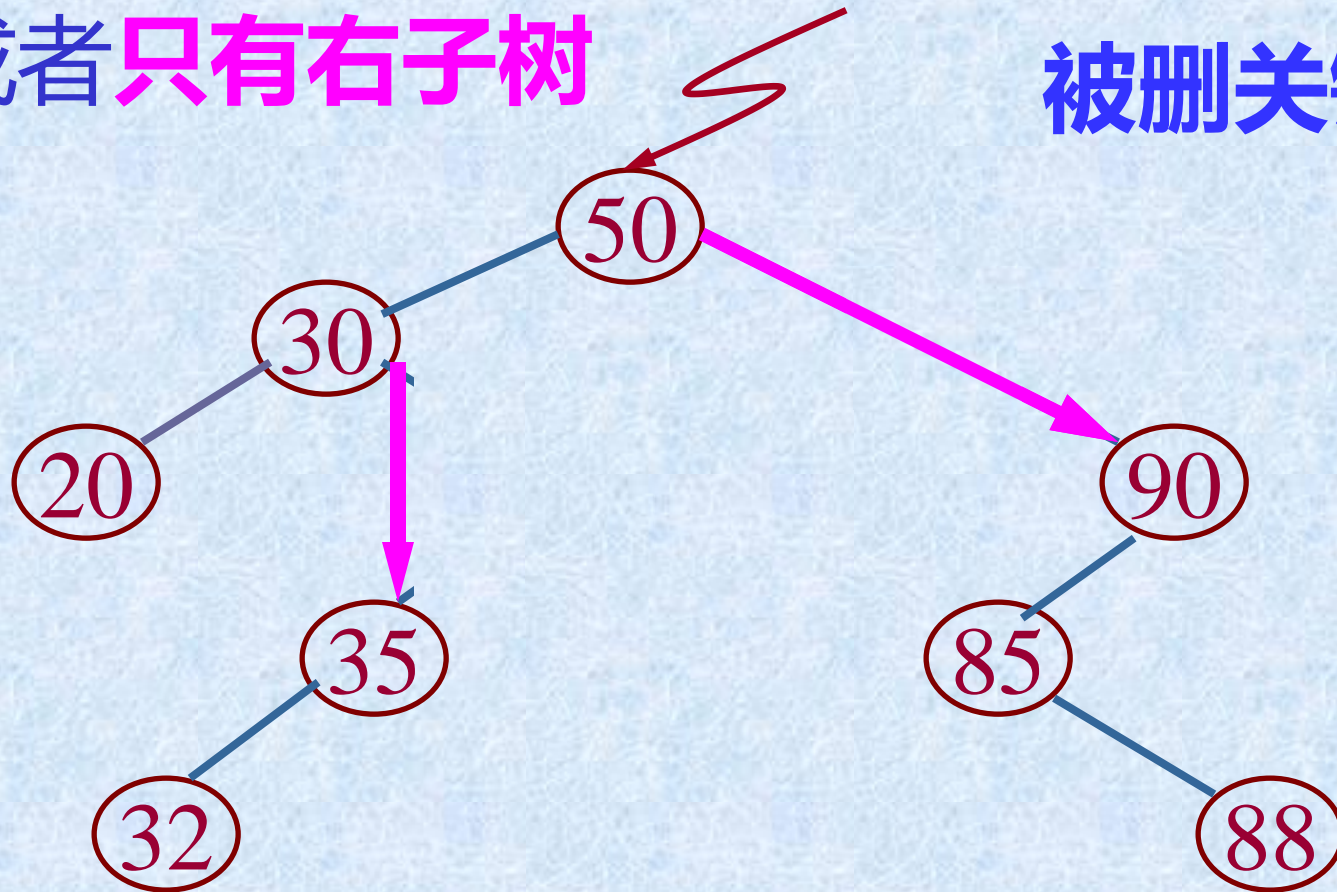
被删关键字 = 88



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**

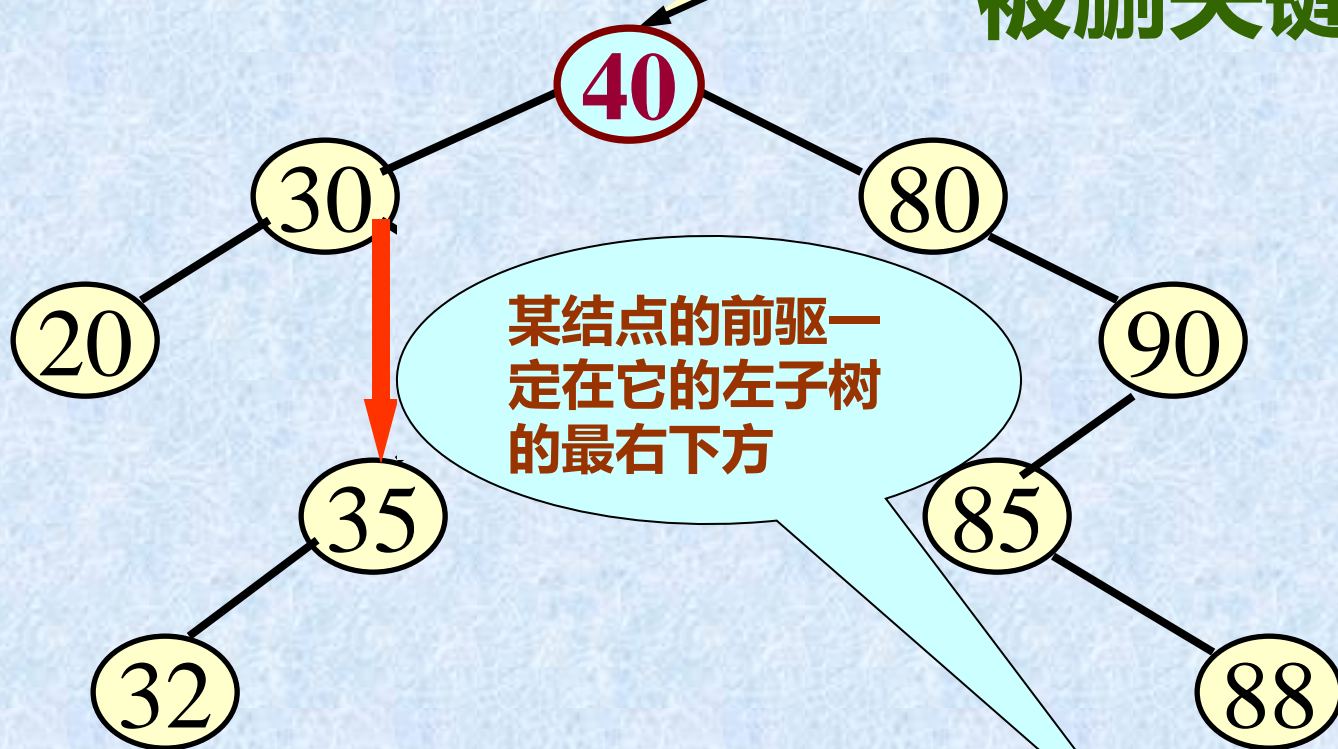
被删关键字 = 80



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

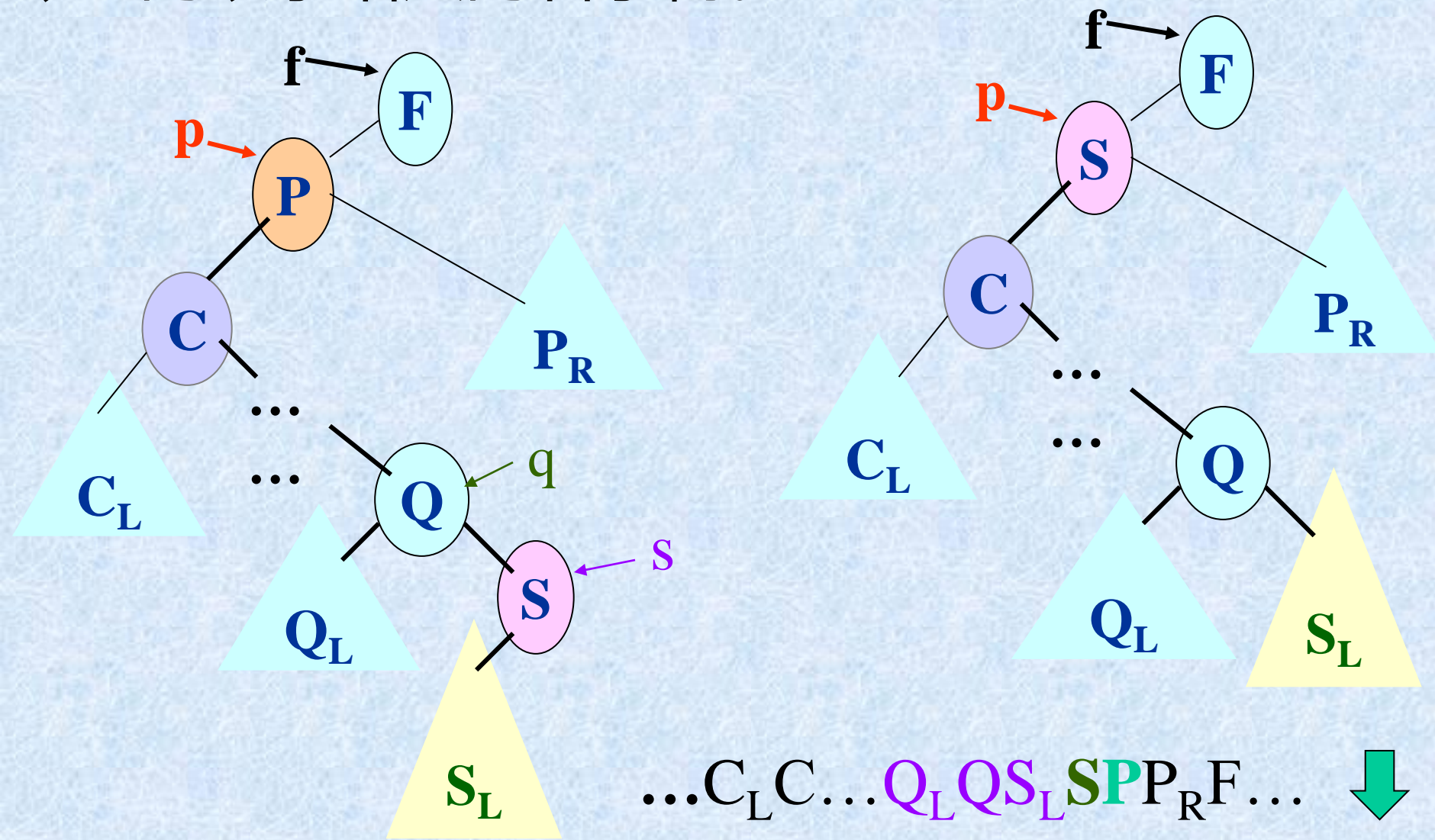


前驱结点 被删结点

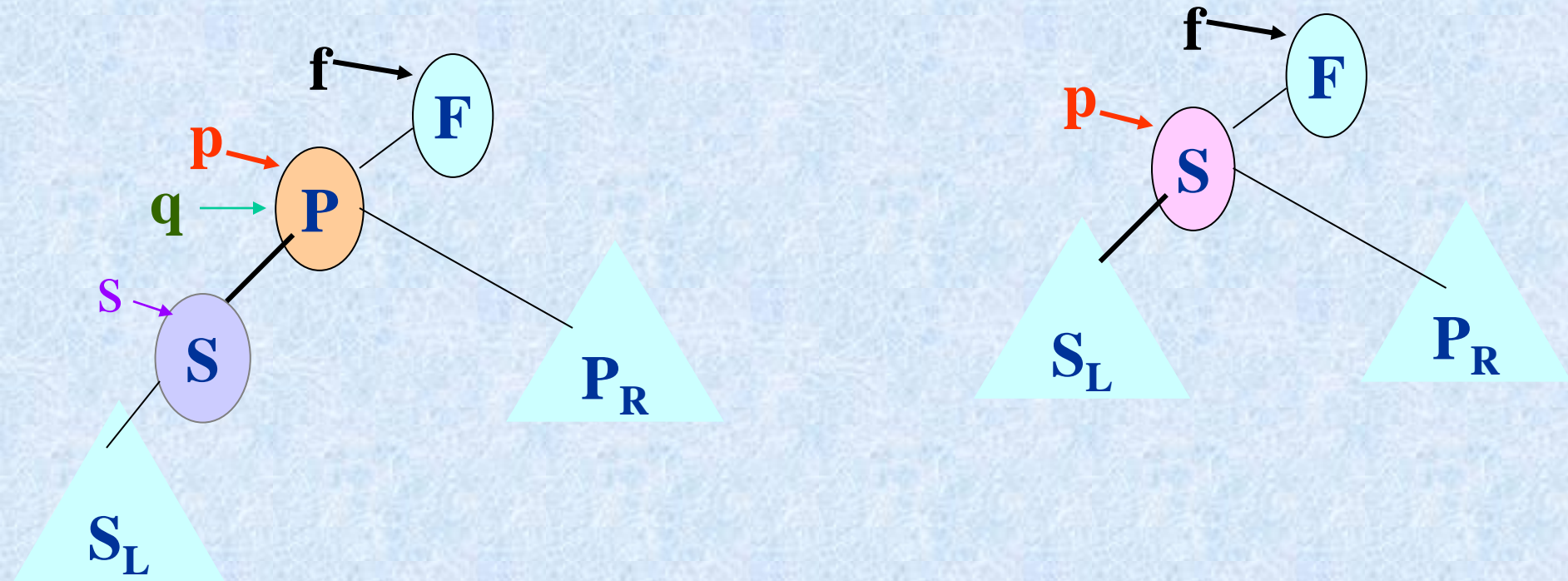


以其前驱替代之，然后再删除该前驱结点

(1) 设S结点是P结点的直接前驱，用S结点取代P结点，然后再从二叉树中删除S结点，并将 S_L 作为S的双亲结点的右子树。



(2) S结点是P结点的直接前驱，用S结点取代P结点，然后再从二叉树中删除S结点，并将 S_L 作为S的双亲结点的左子树。



... S_L **S** P P_R F ...



算法描述如下：

```
Status DeleteBST (BiTree &T, KeyType kval ) {  
    // 若二叉排序树 T 中存在其关键字等于 kval 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kval的数据元素  
    else { ... ... }  
} // DeleteBST
```




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



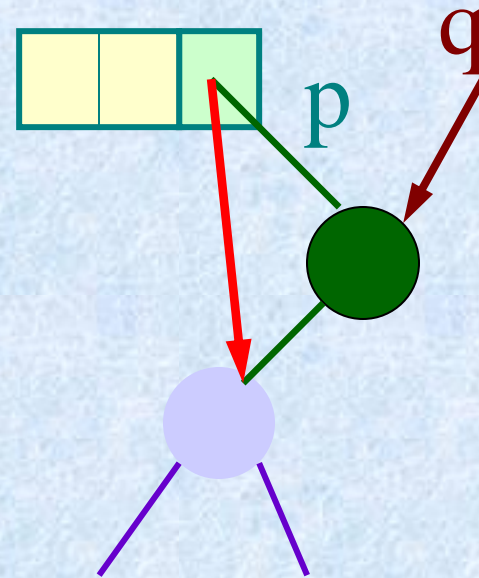
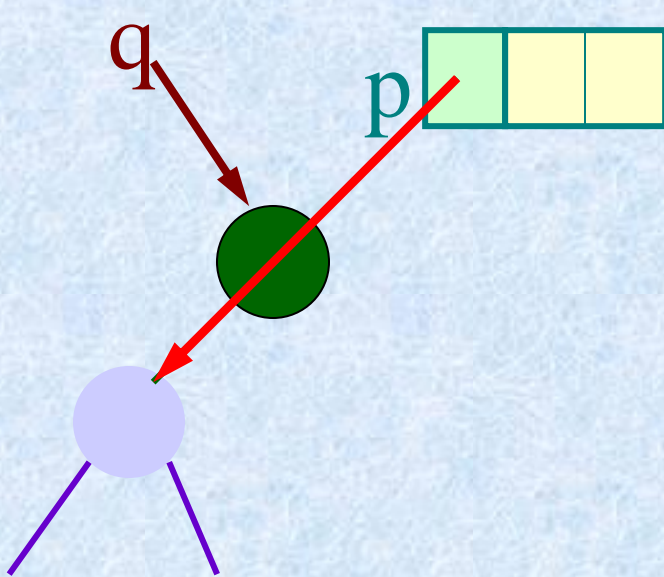
其中删除操作过程如下所描述：

```
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
```



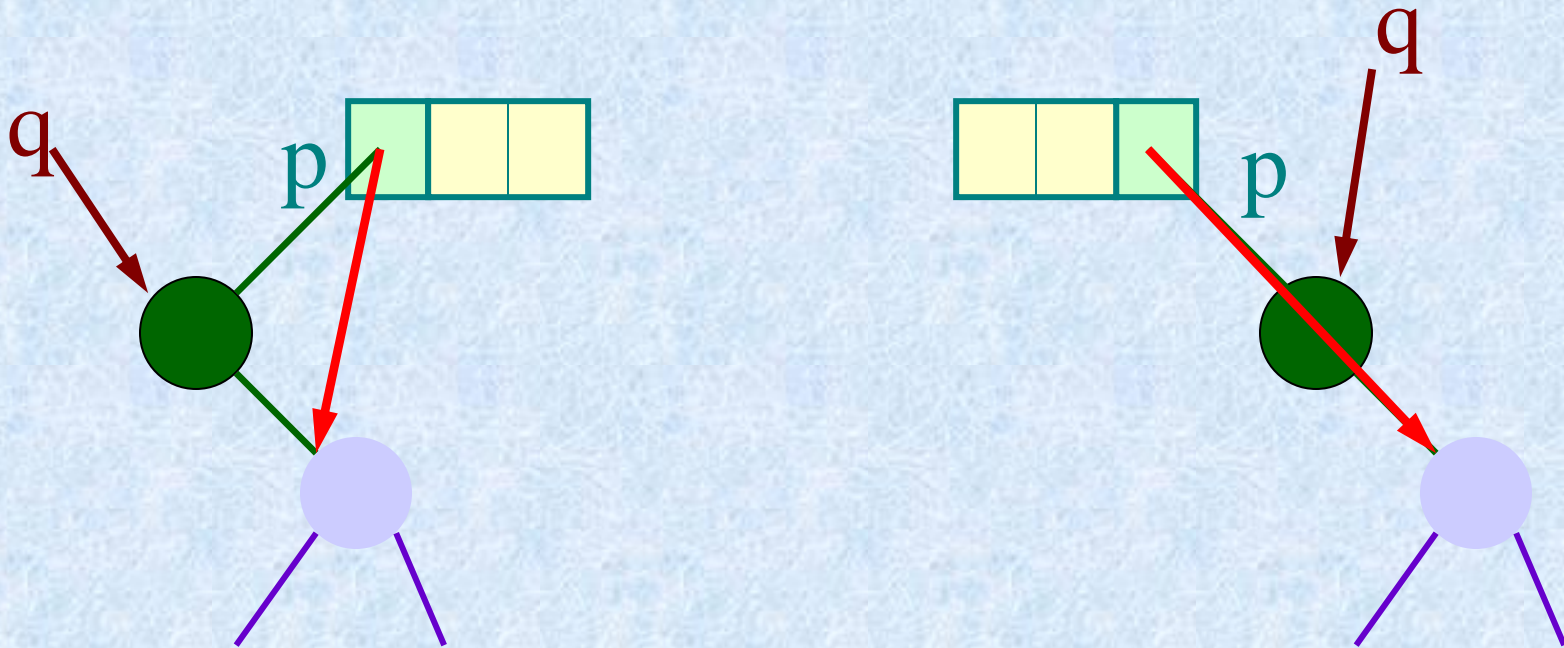
// 右子树为空树则只需重接它的左子树

$q = p$; $p = p \rightarrow \text{lchild}$; **delete**(q);



// 左子树为空树只需重接它的右子树

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



// 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, q指向s的父节点

p->data = s->data;

if (q != p) q->rchild = s->lchild;

// 重接*q的右子树

else q->lchild = s->lchild;

// 重接*q的左子树

delete(s);

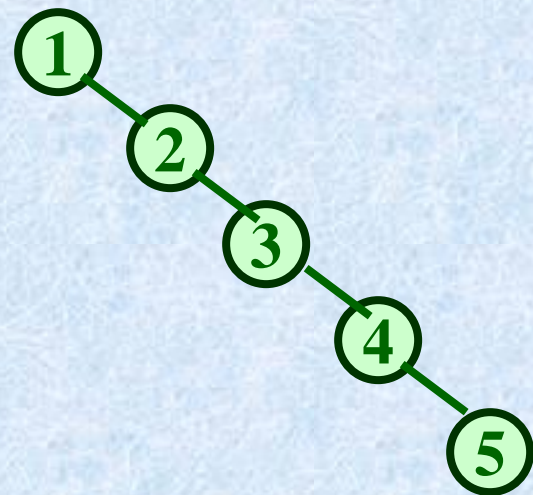
5. 查找性能的分析

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：

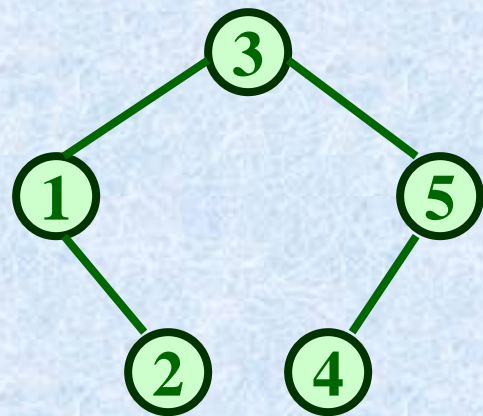
由关键字序列 **1, 2, 3, 4, 5**
构造而得的二叉排序树，

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



由关键字序列 **3, 1, 2, 5, 4**
构造而得的二叉排序树

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



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,

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

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

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

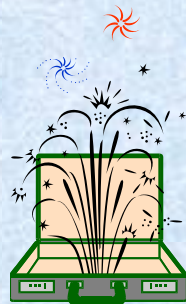
$$= 1 + \frac{1}{n} \left(k \times P(k) + (n - k - 1) \times P(n - k - 1) \right)$$

由此

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

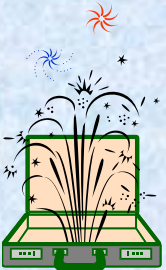
可类似于解差分方程，此递归方程有解：

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



二、二叉平衡树

- 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

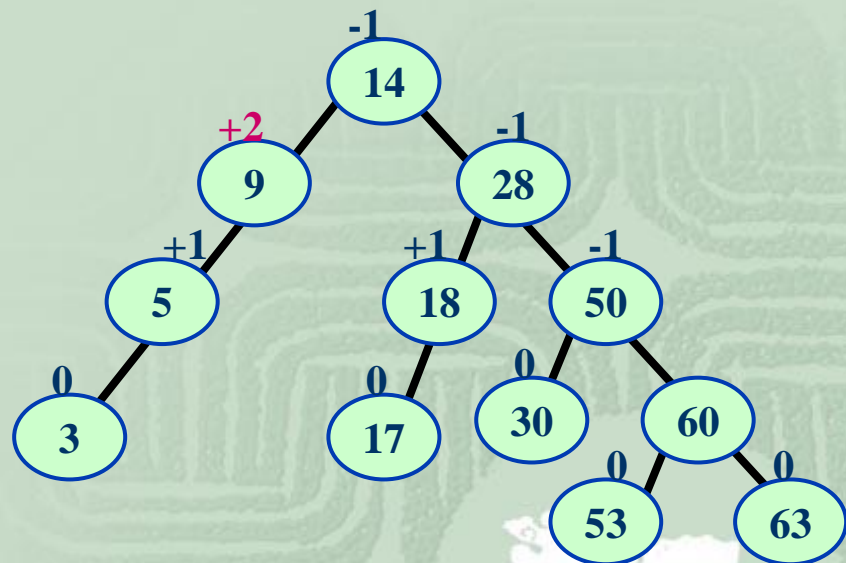
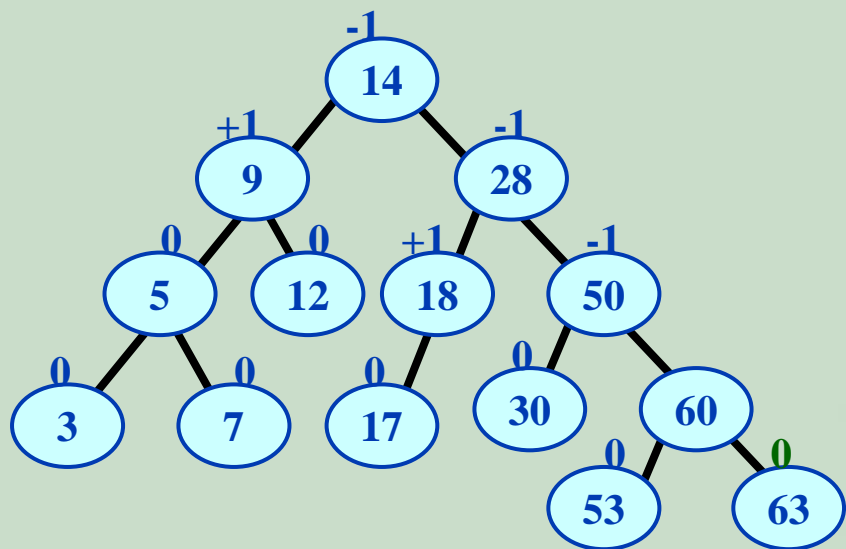


二叉平衡树：对于每个结点，

$| \text{左子树的深度} - \text{右子树的深度} | \leq 1$

结点的平衡因子 = 左子树的深度 - 右子树的深度

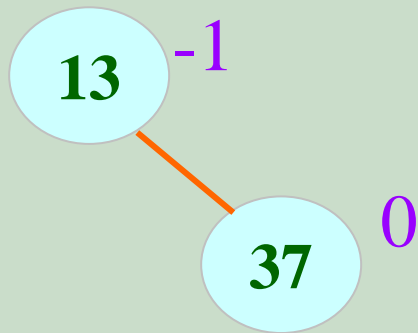
AVL树中所有结点的平衡因子只有三种值：-1, 0, 1



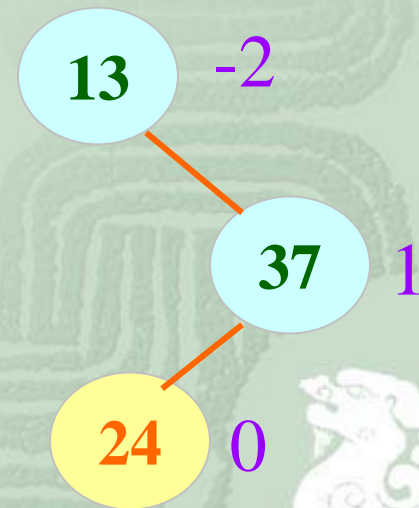
希望任何序列构成的二叉排序树都是AVL树，因为含有n个节点的AVL树的深度和logn是同数量级的，因此它的平均查找长度和logn同数量级。

如何检测二叉树是否失去平衡?

当插入一个新结点时，重新计算从新插入的结点到根结点的路径上所包含结点的平衡因子 (bf)，如果某一结点的 bf 的绝对值超过 1，则二叉树失去平衡。



平衡

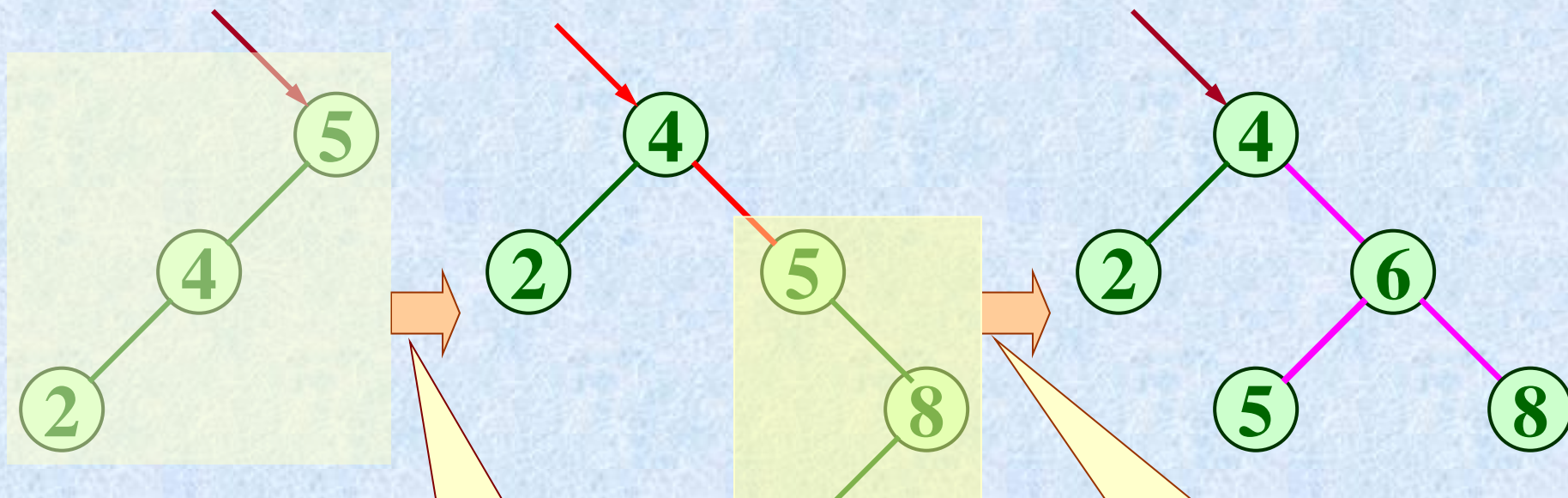


失去平衡

如何使失衡的二叉树达到平衡?

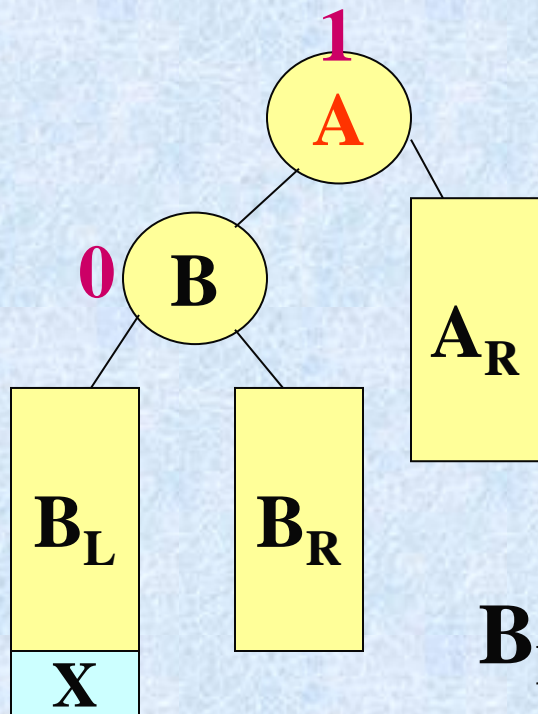
在插入过程中，采用平衡旋转技术。

例如:依次插入的关键字为5, 4, 2, 8, 6, 9

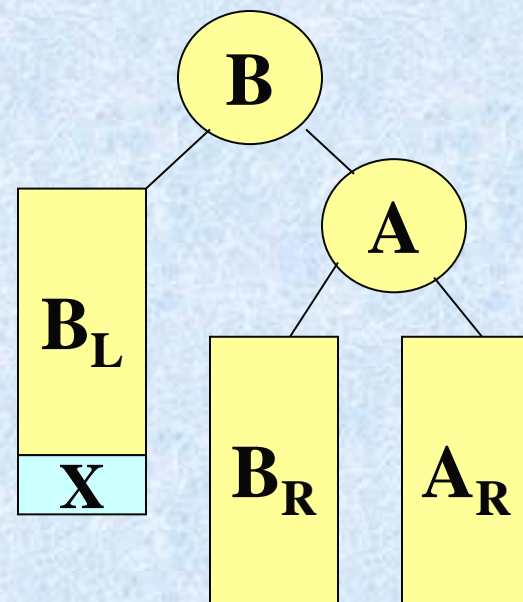


四种调整类型:

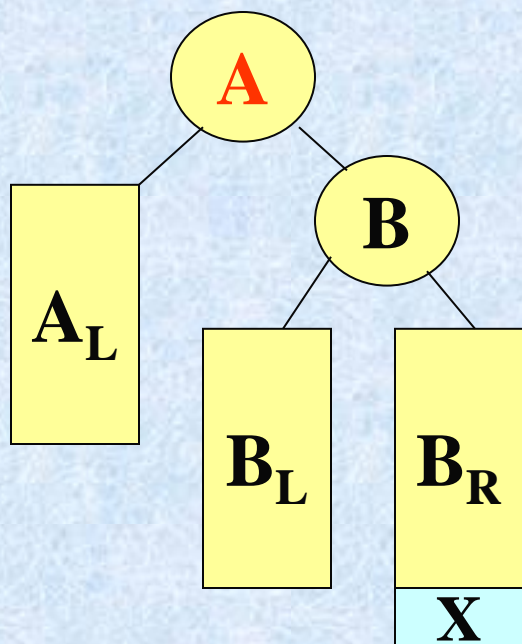
设 x 为插入结点， A 结点为离插入结点 x 最近的一个失衡结点:



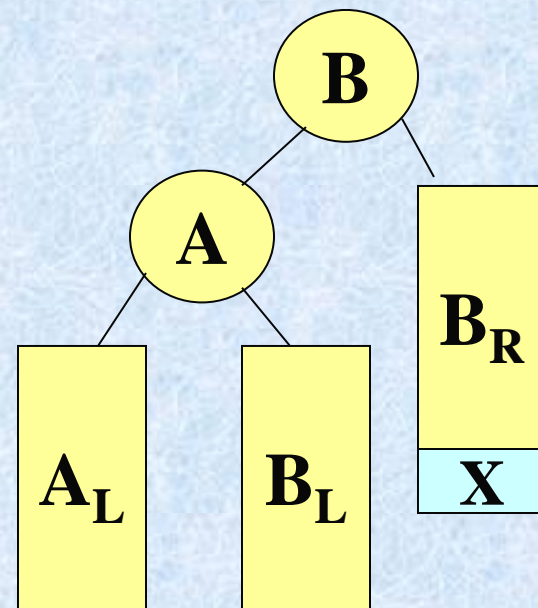
LL型



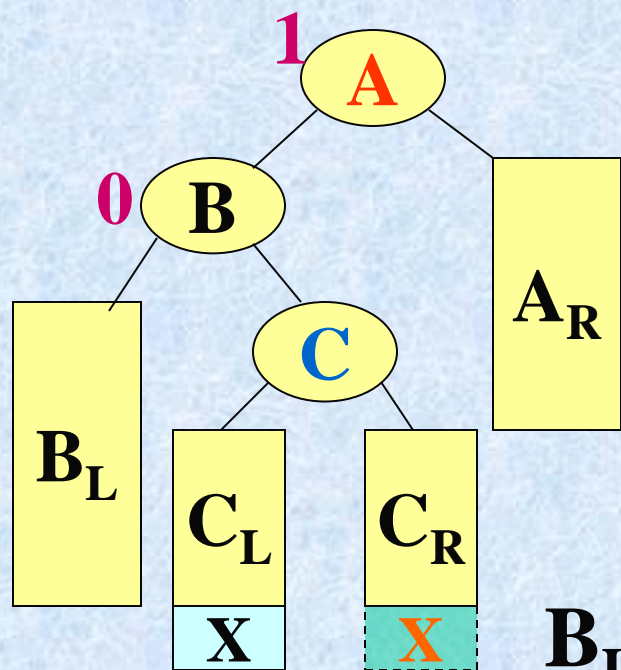
$B_L < \mathbf{B} < B_R < \mathbf{A} < A_R$



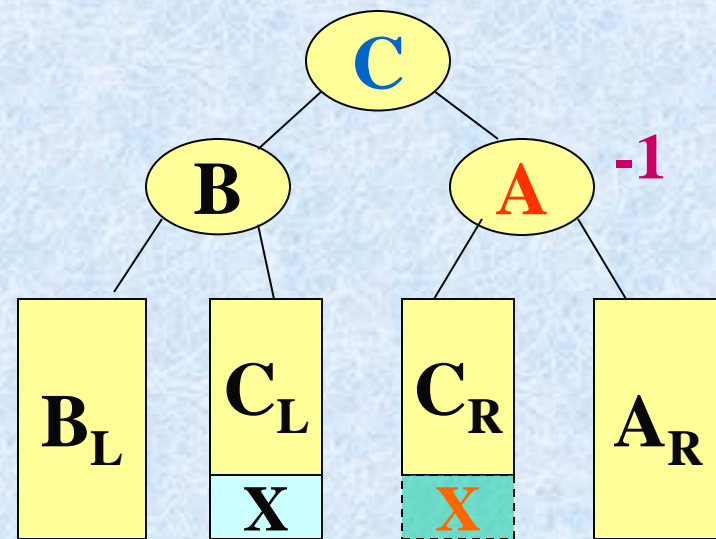
RR型



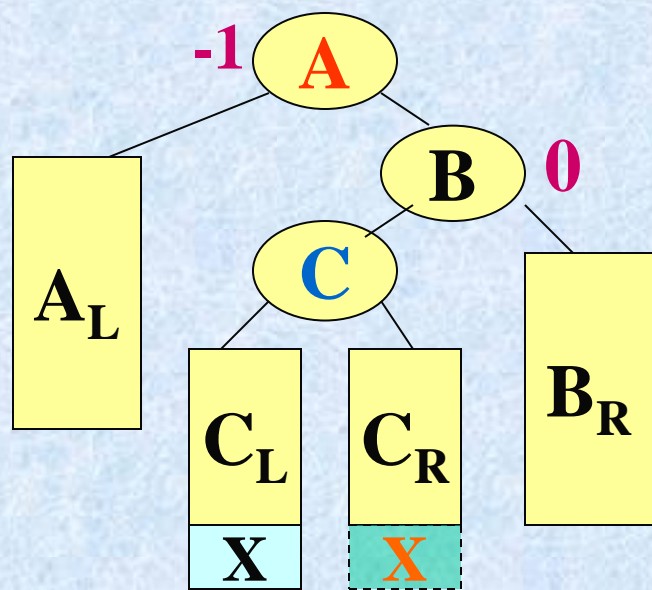
$A_L < \mathbf{A} < B_L < \mathbf{B} < B_R$



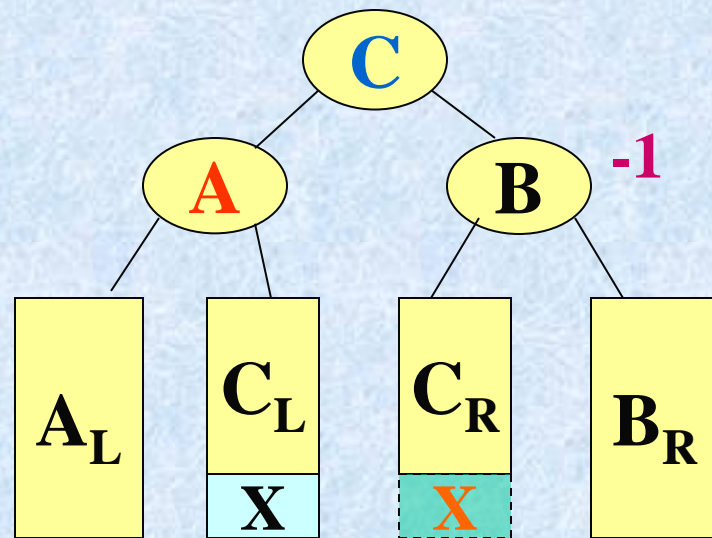
LR型



$B_L < B < C_L < C < C_R < A < A_R$



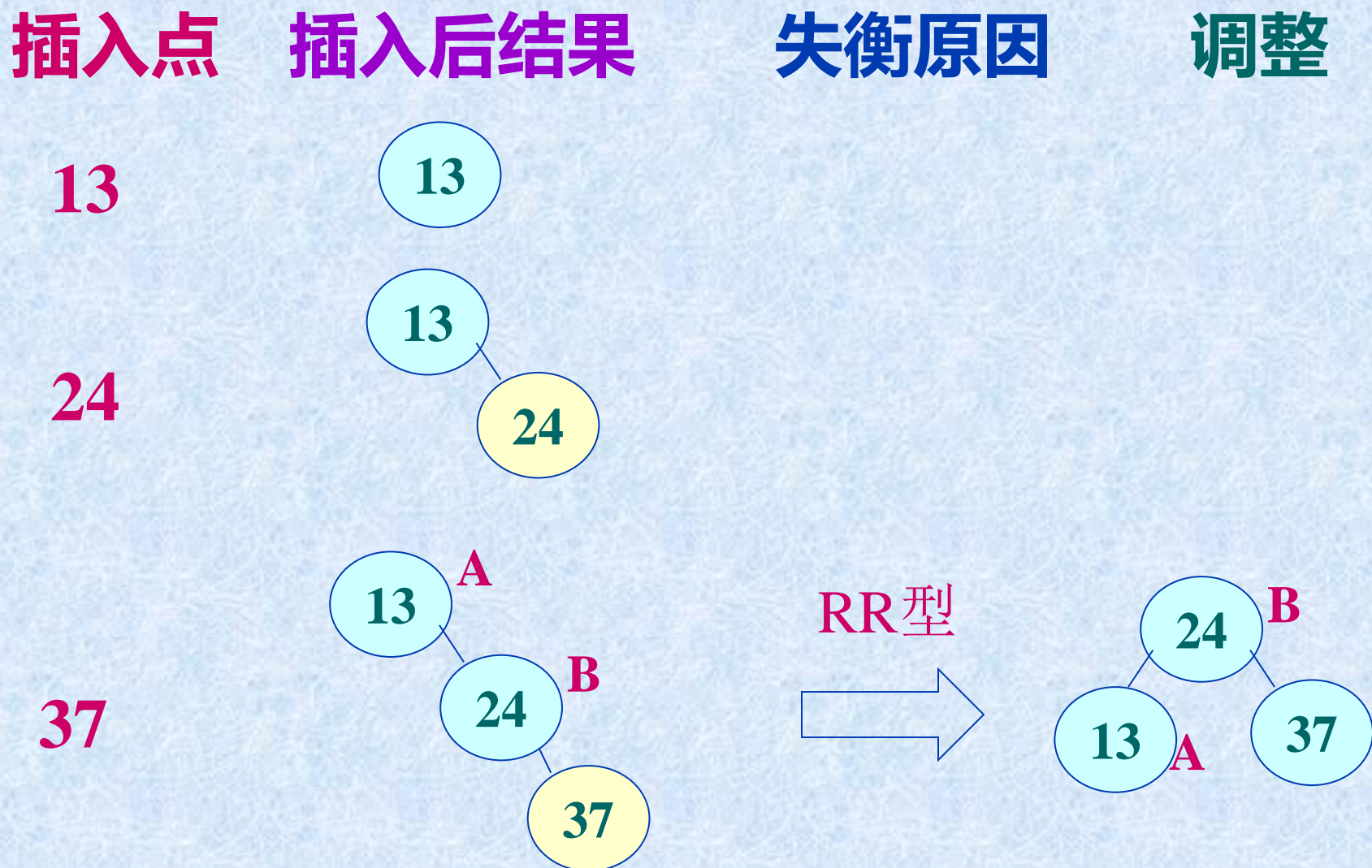
RL型



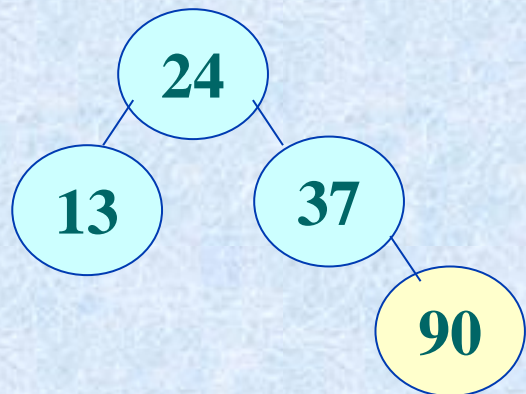
$A_L < A < C_L < C < C_R < B < B_R$

例：按下列关键字的次序生成一棵AVL树。

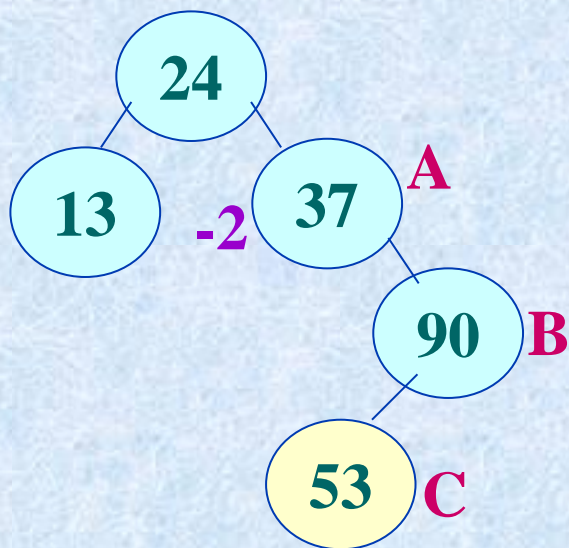
{13, 24, 37, 90, 53, 28, 98}



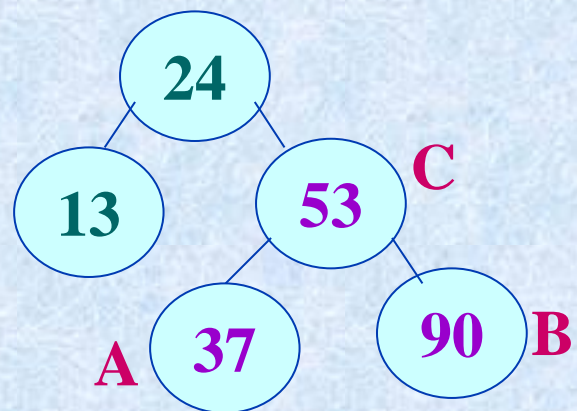
90

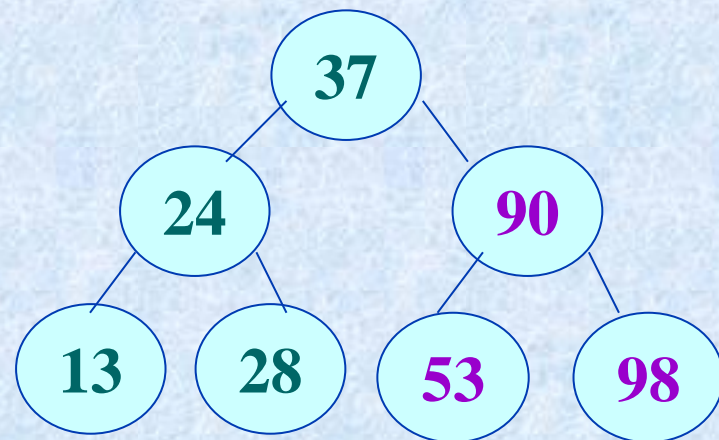
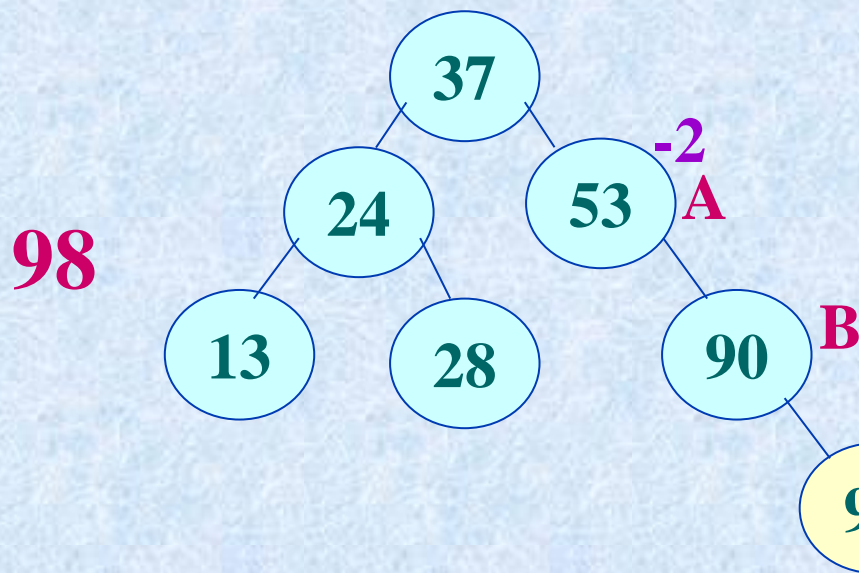
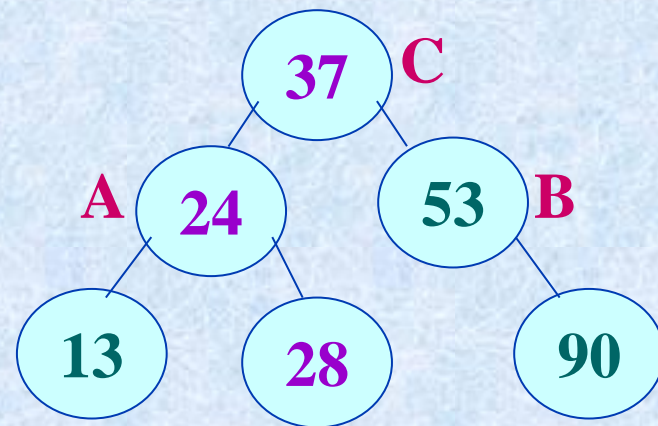
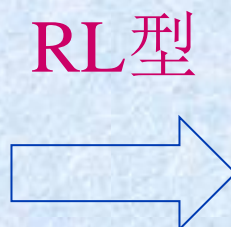
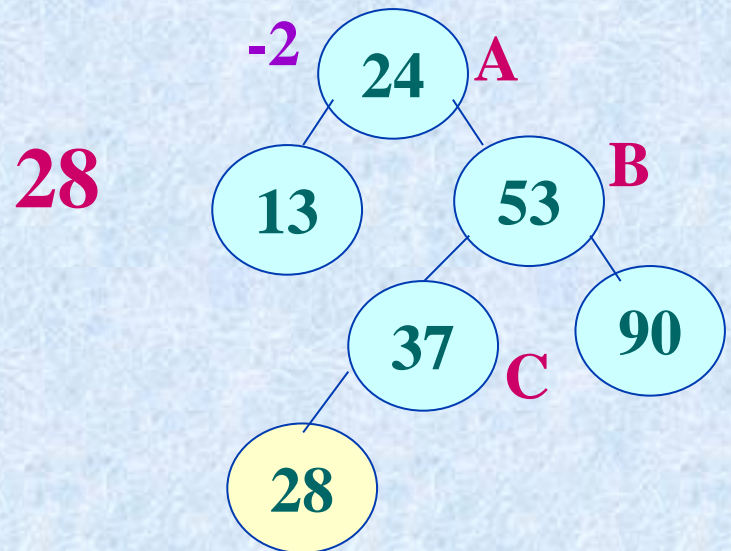


53



RL型





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。

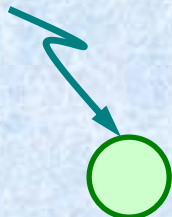
问：含 n 个关键字的二叉平衡树可能达到的最大深度是多少？

先看几个具体情况:

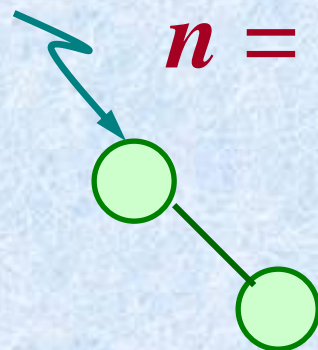
$n = 0$

空树

$n = 1$



$n = 2$

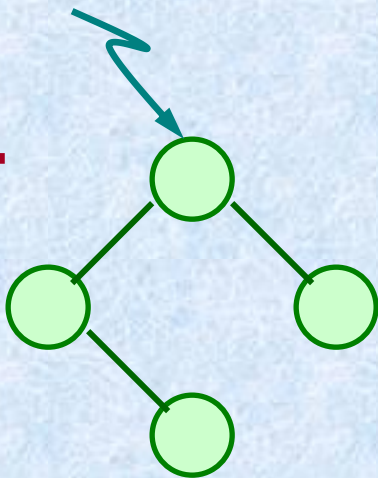


最大深度为 0

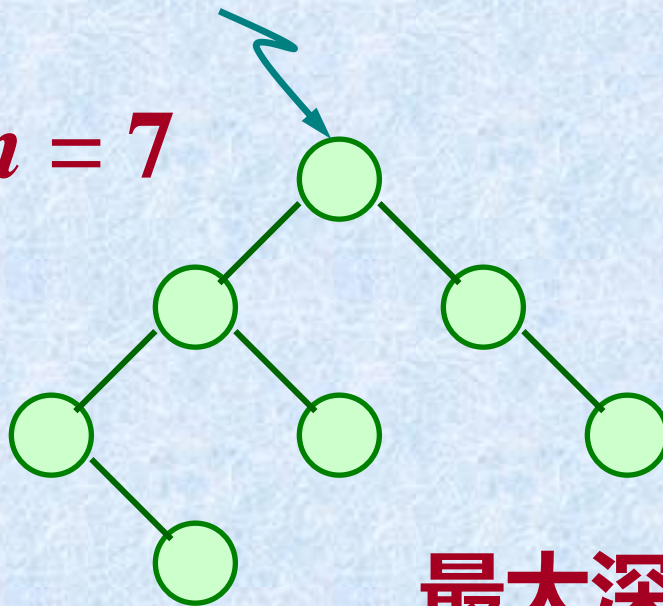
最大深度为 1

最大深度为 2

$n = 4$



$n = 7$



最大深度为 3

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，
$$N_h = F_{h+2} - 1$$

$$F_h = \varphi / \sqrt{5}, \text{ 其中 } \varphi = (1 + \sqrt{5}) / 2$$

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

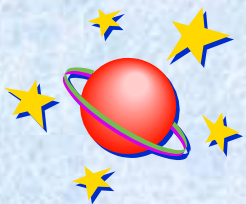
一、哈希表是什么？

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

五、哈希表的删除操作



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，

查找的过程是把给定值依次与关键字集合中各个**关键字**进行**比较**，

查找的效率取决于与给定值**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（学生信息表），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 $0 \sim 999$ 的顺序表存储学生信息。

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

例如：对于如下 9 个关键字

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

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(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**不超出存储允许范围**即可；
- 2) 由于哈希函数通常是一个**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，
使冲突尽可能少地发生。

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** $H(\text{key})$ 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

构造哈希函数的目标：

- ✦ 哈希地址尽可能均匀分布——均匀性好；
- ✦ 哈希地址计算尽量简单。

对**数字型**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

$$H(\text{key}) = a \times \text{key} + b, \quad \text{其中} a, b \text{ 为常数}$$

例：1949年后出生的人口调查表，关键字是年份

年份	1949	1950	1951	...
人数

$$H(\text{key}) = \text{key} + (-1948)$$

此法不会发生冲突，但是仅适合于：

地址集合的大小 等于 关键字集合的大小 

5. 除留余数法

当关键字key为整数时，用关键字除以一个整数p 所得余数作为哈希地址。

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

其中， $p \leq m$ (表长) 并且

p 应为不大于 m 的质数

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。



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 $H(\text{key})$ 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： $H_0 = H(\text{key})$

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

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

m 为哈希表表长

对增量 d_i 有两种取法:

1) 线性探测再散列

$$d_i = 1, 2, 3, \dots,$$

2) 平方探测再散列

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

例如：关键字集合

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

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

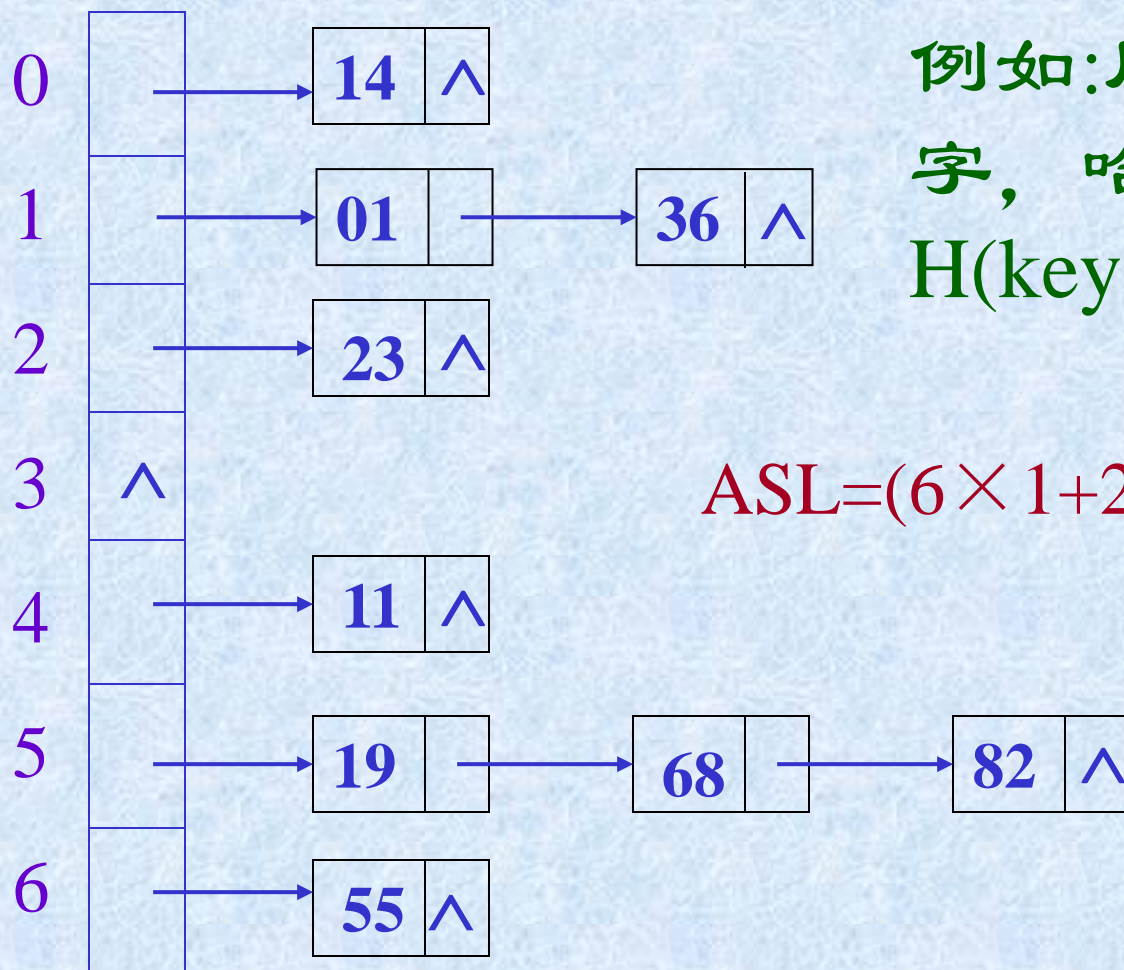
若采用线性探测再散列处理冲突

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
1	1	2	1	2	1	4		1		3

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

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



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 $r[i] = \text{NULL}$ **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, ... }; //容量递增表, 一个合适
typedef struct {                //的素数序列
    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中查找关键字为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, Elemtyp 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, (阈值 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， **装填因子**
 $\alpha = n/m$ 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， $ASL = 22/9$

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：

线性探测再散列

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

二次探测再散列

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

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。**



小结

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，会对不平衡的二叉排序中进行平衡旋转操作。
3. 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查找表的实质性差别。
4. 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。

练习

1. 已知长度为12的表

{Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}

- (a)依次插入构造二叉排序树，画出每步树的结构
- (b)依次插入构造二叉平衡树，画出每步树的结构
- (c)计算上述两种情况在等概率查找下查找成功的ASL

- ## 2. 设散列表为H，表长为13，散列函数为 $H(\text{key}) = \text{key} \bmod 13$ ，采用线性再散列。设插入关键字为{2, 8, 31, 20, 19, 18, 53, 27}。画出插入这8个关键字后的散列表再计算等概率查找下查找成功时的ASL