华中科技大学

数据结构

第9章　查找

主讲教师：周时阳

内容摘要

《数据结构》是计算机科学与技术类各专业的一门基础课。

本章主要介绍数据结构课程研究的问题背景、研究内容和范围。讨论了数据结构和算法的基本概念以及算法的评价。

关于线性结构、树型结构和图型结构等3类基本结构，将在后续各章陆续展开讨论它们的逻辑结构、逻辑结构上定义的运算、物理结构、逻辑结构与物理结构对应关系、运算的实现算法与效率分析。

重点讲解

小结

问题背景

“查找”是基于数据逻辑结构(D,R)定义的一种十分常见的运算。数学上，“查找” 是指问题：“$x \in D$?”，即查找是确定某个数据元素是否在数据元素集上的问题。

通常，$D=\{a_1,a_2,\ldots,a_i,\ldots,a_n\}$的每个数据元素$a_i$具有唯一标识的分量，被称为关键字，记为：$a_i.key$。

$$x \in D \Leftrightarrow x.key \in \{a_1.key,a_2.key,\ldots,a_i.key,\ldots,a_n.key\}$$

$$\Updownarrow$$

$$key \in \{key_1,key_2,\ldots,key_i,\ldots,key_n\}$$

$$\Updownarrow$$

$$k \in \{k_1,k_2,\ldots,k_i,\ldots,k_n\}\ (k,\ k_i \in I)$$

又因为key的值域与整数集I之间，可以建立一一对应关系， 问题$x \in D$ 可进一步等价转换。

查找问题：$k \in \{k_1, k_2, \ldots, k_i, \ldots, k_n\}$，对应的查找运算的一般描述形式为：Search(T,key)，即查找key是否在T中。

Search(T,key)返回值或函数值，通常采用下列3种定义形式之一。

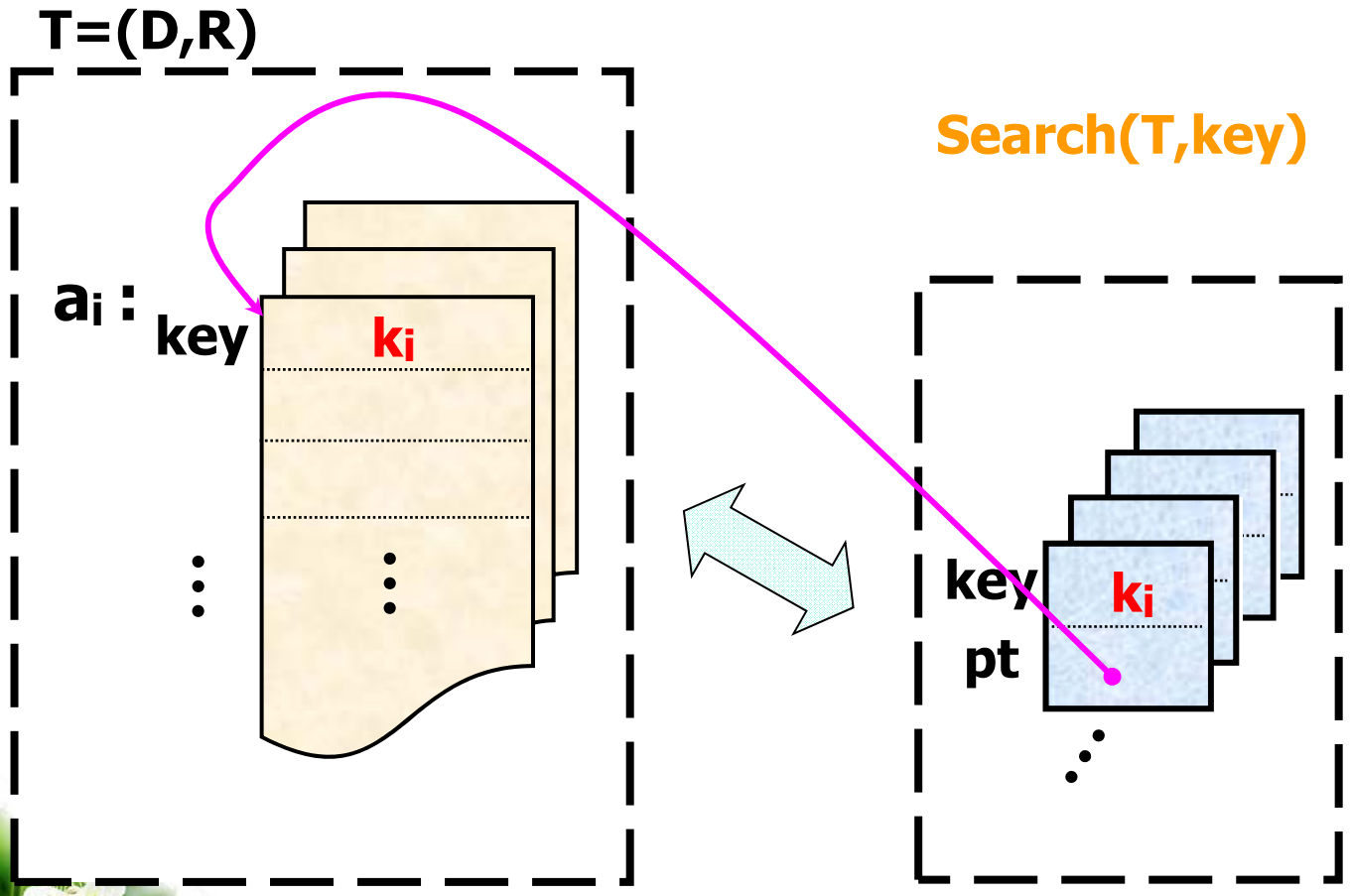
$$\text{Search}(T,key) = \begin{cases} \text{True} & (key \in T) \\ \text{False} & (key \notin T) \end{cases}$$

$$\text{Search}(T,key) = \begin{cases} i & (key \in T, k_i = key) \\ 0 & (key \notin T) \end{cases}$$

p为a_i之存储地址

$$\text{Search}(T,key) = \begin{cases} p & (key \in T, k_i = key) \\ \text{NULL} & (key \in T) \end{cases}$$

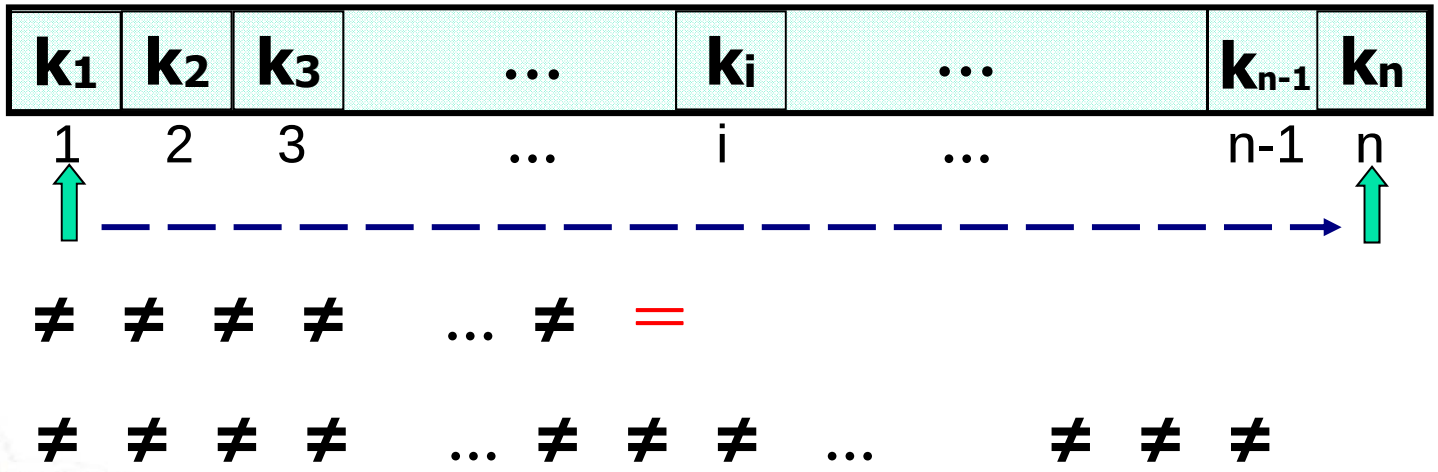
Search(T,key)函数值第3种定义形式之背景



9.1 静态查找表

9.1.1 顺序表的查找

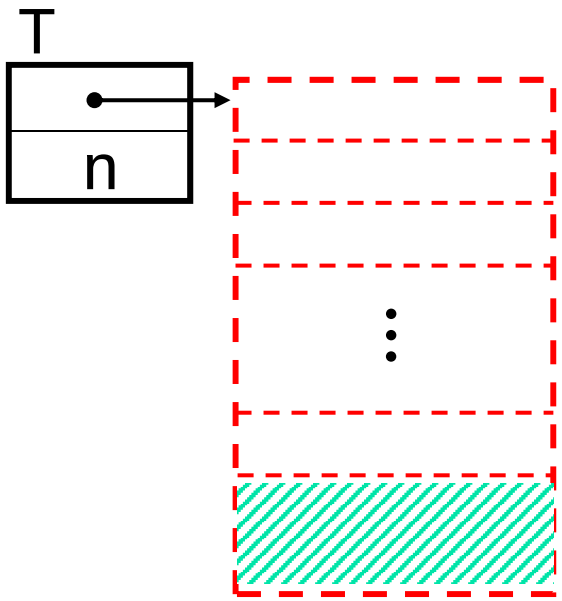
算法思想：依次与每个关键字逐个比较，如果与给定值相等，则查找成功，返回成功值；如果与所有关键字都不相等，则查找失败，返回失败值。



目录

```
Typedef struct{
    Elemtype * elem;
    int length;
} SSTable;

int Search(SSTable T,KeyType k){
     for(i=1; i<=T.length; i++)
        if(T.elem[i].key==k) return(i);
     return(0);
}
```



算法分析：规模为表长n，统计关键字之间的比较次数。

$T_{sb}(n)=1$

$T_{sw}(n)=n$

$T_{sa}(n)=(1+2+3+...+n)/n=(n+1)/2$

$T_f(n)=n$

定义：在表中查找给定值，需要与每个关键字次数的数学期望值，称为平均查找长度（**A**verage **S**earch **L**ength）。

成功平均查找长度：

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

（P_i和C_i分别为查找到第i个关键字概率和比较次数，n为表长）

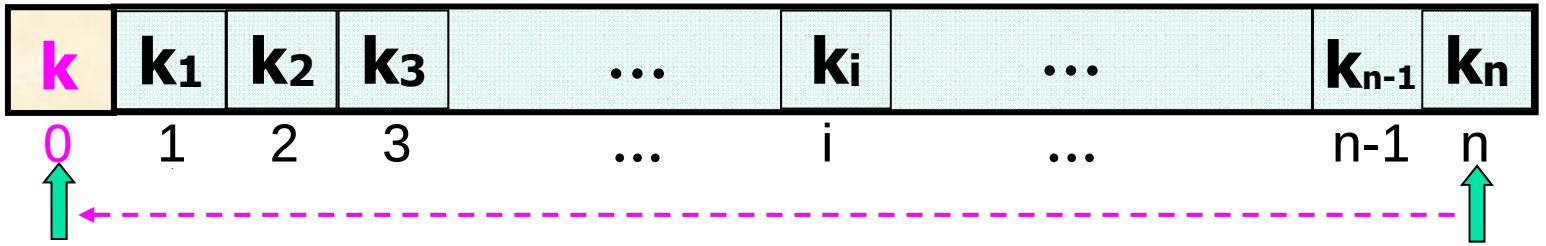
失败平均查找长度：

$$ASL=\sum_{i=1}^{m} Q_i C_i$$

（Q_i和C_i分别为第i种失败概率和关键字比较次数，m为失败情况）

注：表中之外的任意给定值，均为失败情况！

算法改进：哨兵技术！



算法分析：

$T_{sb}(n)=1$

$T_{sw}(n)=n$

$T_{sa}(n)=(n+1)/2$

$T_f(n)=n+1$

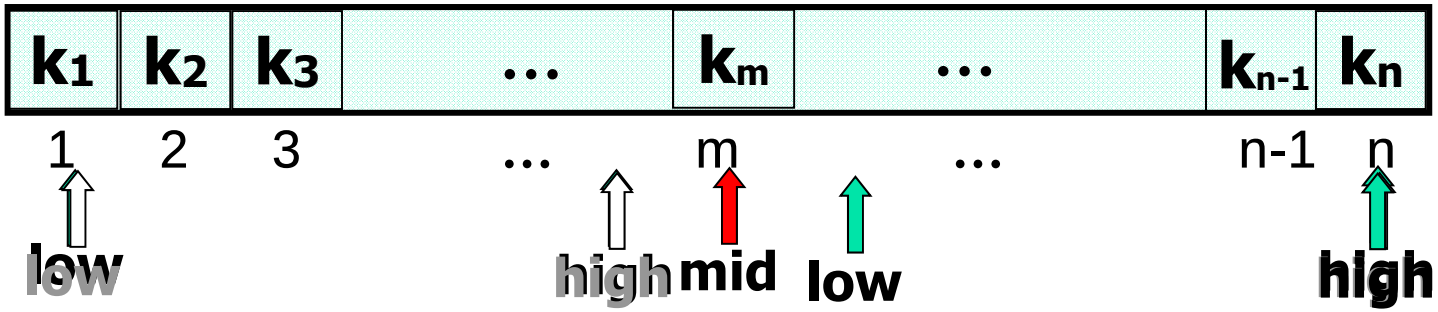
```
Typedef struct{
    Elemtype * elem;
    int length;
} SSTable;

int Search(SSTable T,KeyType k){
     T.elem[0]=k;
     for(i=T.length; T.elem[i].key !=k ; i--);
     return(i);
}
```

9.1.2 有序表的查找

(1) 折半查找法

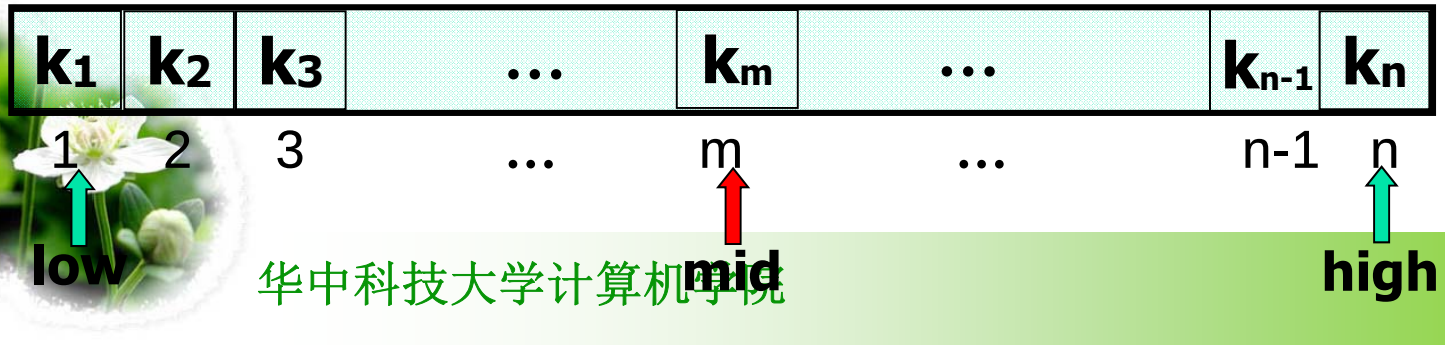
算法思想：与处于查找表中间位置关键字比较，如果等于给定值，则查找成功，返回成功值；如果大于给定值，在表的左部折半法查找；如果小于给定值，在表的右部折半法查找；仅当左部或右部为空时候，查找失败，返回失败值。



注：失败条件 —— low>high

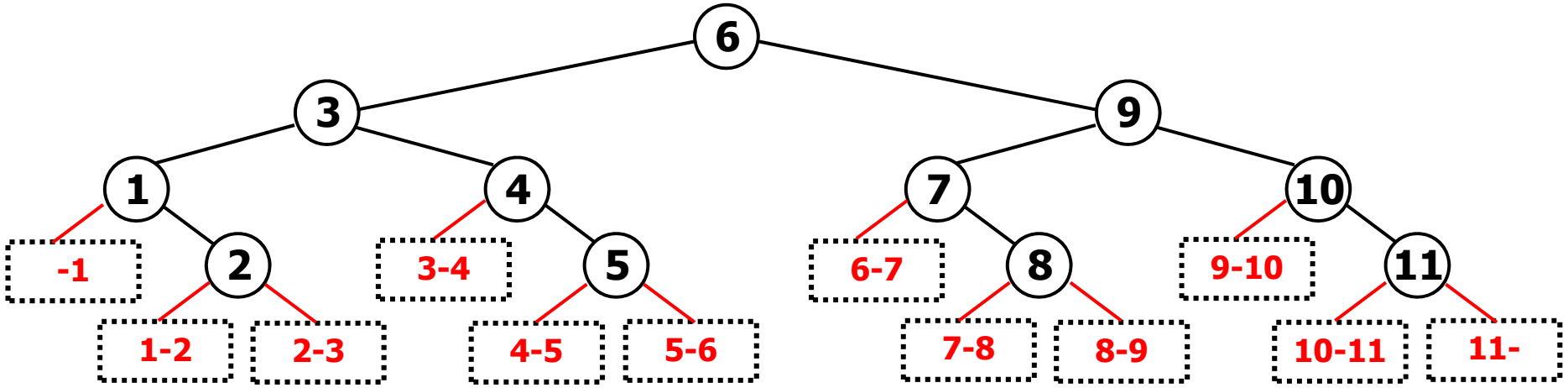
```
Typedef struct{
    Elemtype * elem;
    int length;
} SSTable;

int Search(SSTable T,KeyType k){
     low=1; high=T.length;
     while (low<=high){
         mid=(low+hig)/2;
         if(T.elem[mid]==k) return(mid);  //等于，成功！
             else if(T.elem[mid]>k) high=mid-1; //左部
                      else low=mid+1; //右部
     }
     return(0); //失败
}
```



算法分析：规模为表长**n**，统计关键字之间的比较次数。

$$ASL_{(S)} = \sum_{i=1}^{n} P_i C_i = \frac{1}{n} \sum_{i=1}^{h} j \cdot 2^{j-1} = \frac{n+1}{n} \log_2(n+1) - 1$$

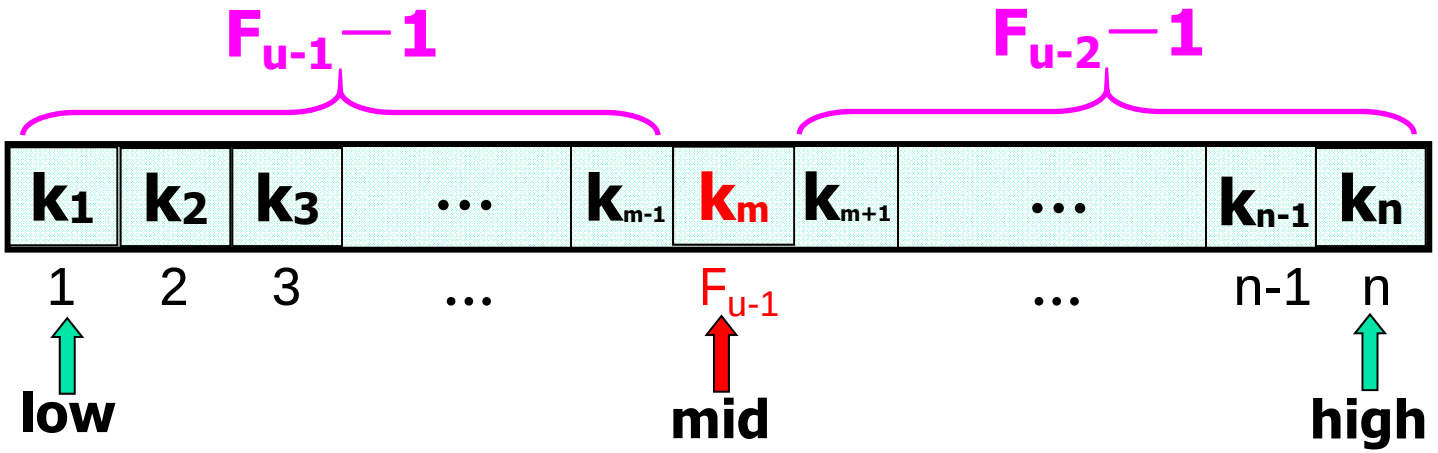


$ASL_{(S)}=(1\times1+2\times2+3\times4+4\times4)/11=33/11=3$

$ASL_{(f)}=(3\times4+4\times8)/12=44/12=11/3$

计算“中间位置”的其它方法

(2) 菲波那契序列法($n=F_u-1$)

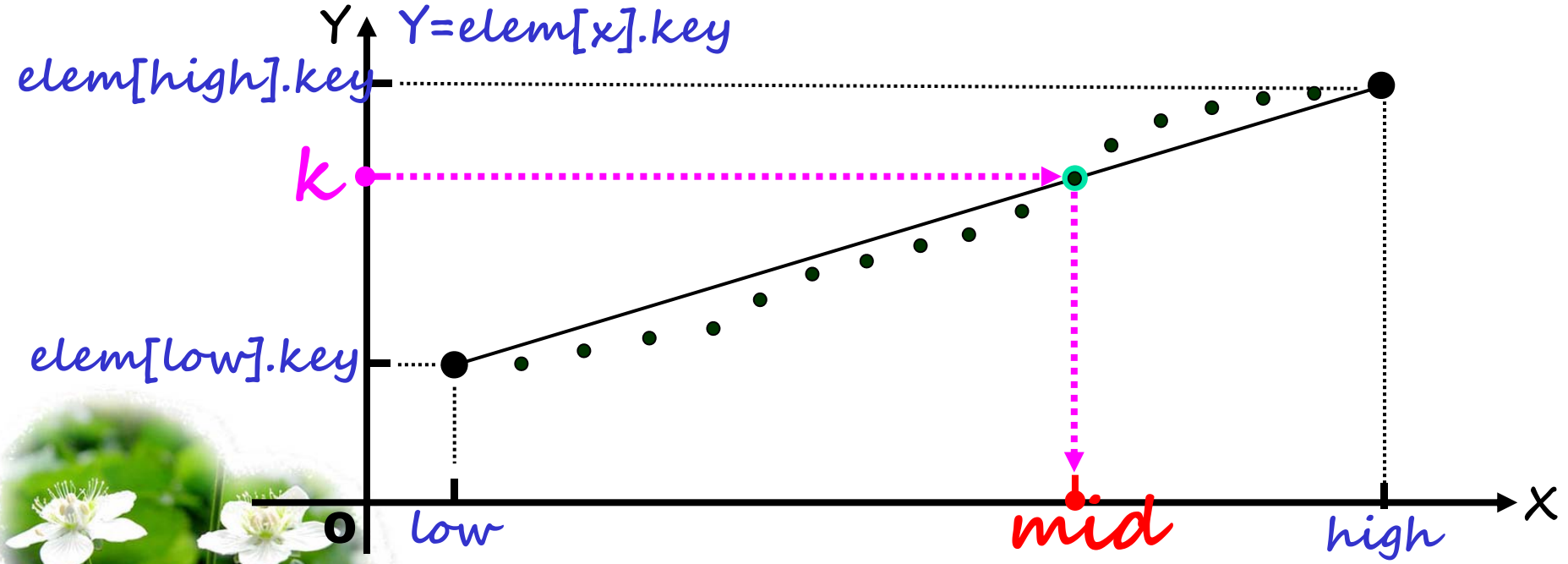


菲波那契序列：
$$F_u=\begin{cases}0 & (u=0)\\ 1 & (u=1)\\ F_{u-1}+F_{u-2} & (u>1)\end{cases}$$

计算“中间位置”的其它方法

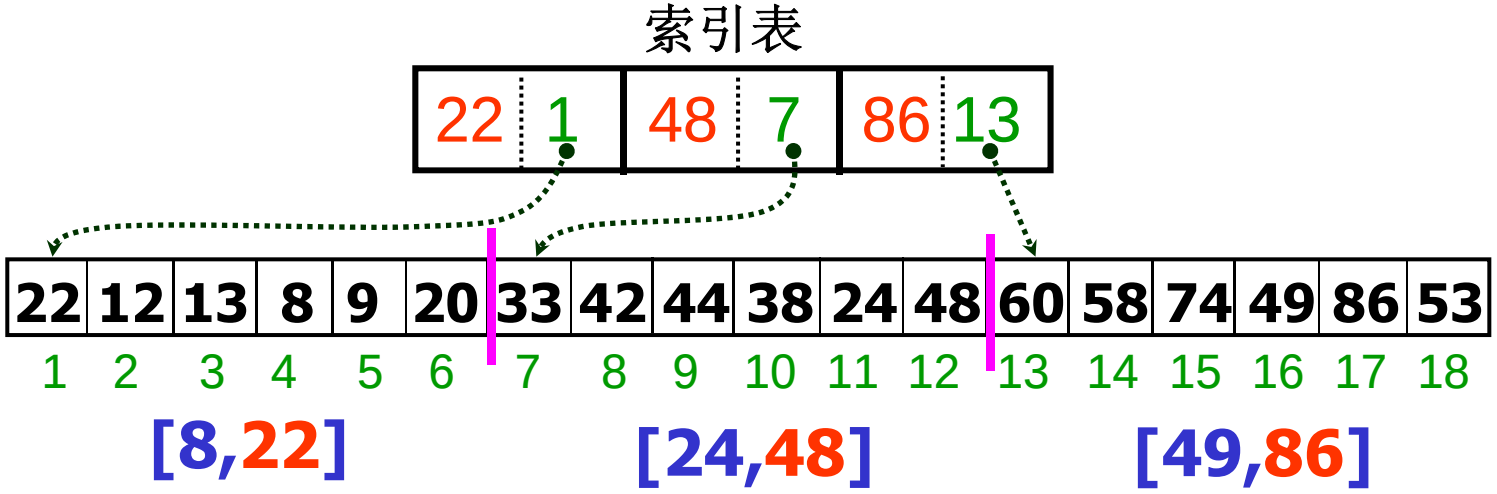
(3) 线性插值法(k为给定值)

$$mid = \frac{k - T.elem[low].key}{T.elem[high] - T.elem[low].key}(high - low + 1)$$



9.1.4 索引顺序表的查找

利用关键字序列的分段(块)有序性，建立分段(块)索引表。借助分段索引表，实现快速查找。这种方法称为分块查找法。



算法思想：

(1) 在分段索引表中“顺序”或“折半”查找给定值所在的块；

(2) 在(1)确定的块中，顺序查找给定值。

算法分析：假设在索引表上的平均查找长度为ASL_b，在查找表上的平均查找长度为ASL_w，则

$$ASL = ASL_b + ASL_w$$

一般情况下，不妨令查找表长为**n**，均匀划分为**b**块，每块含关键字**s=n/b**个；在索引表查找到每一块和块内查找到每个关键字，都是等概率的，即**1/b**和**1/s**。

■采用索引表上顺序查找

$$ASL = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

显然，当 $s=\sqrt{n}$ 时，ASL取最小值$\sqrt{n}+1$。

■采用索引表上折半查找

$$ASL \approx \log_2\left(\frac{n}{s}+1\right) + \frac{s}{2}$$

9.2 动态查找表

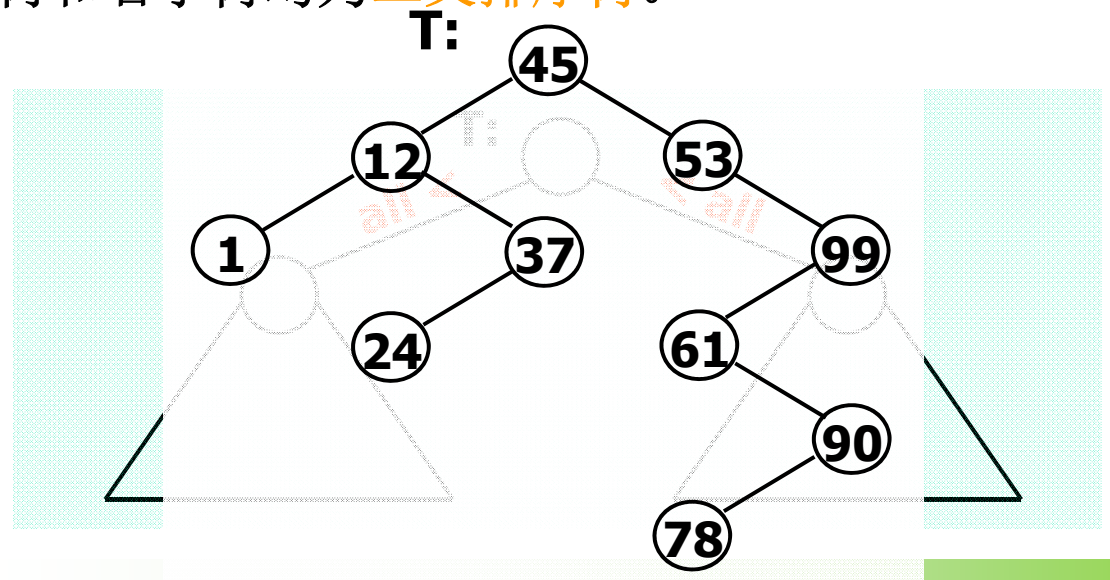
9.2.1 二叉排序树和平衡二叉树

二叉排序树(Binary Sort Tree)或是一棵空树，或满足下列性质的一棵非空的二叉树**T**：

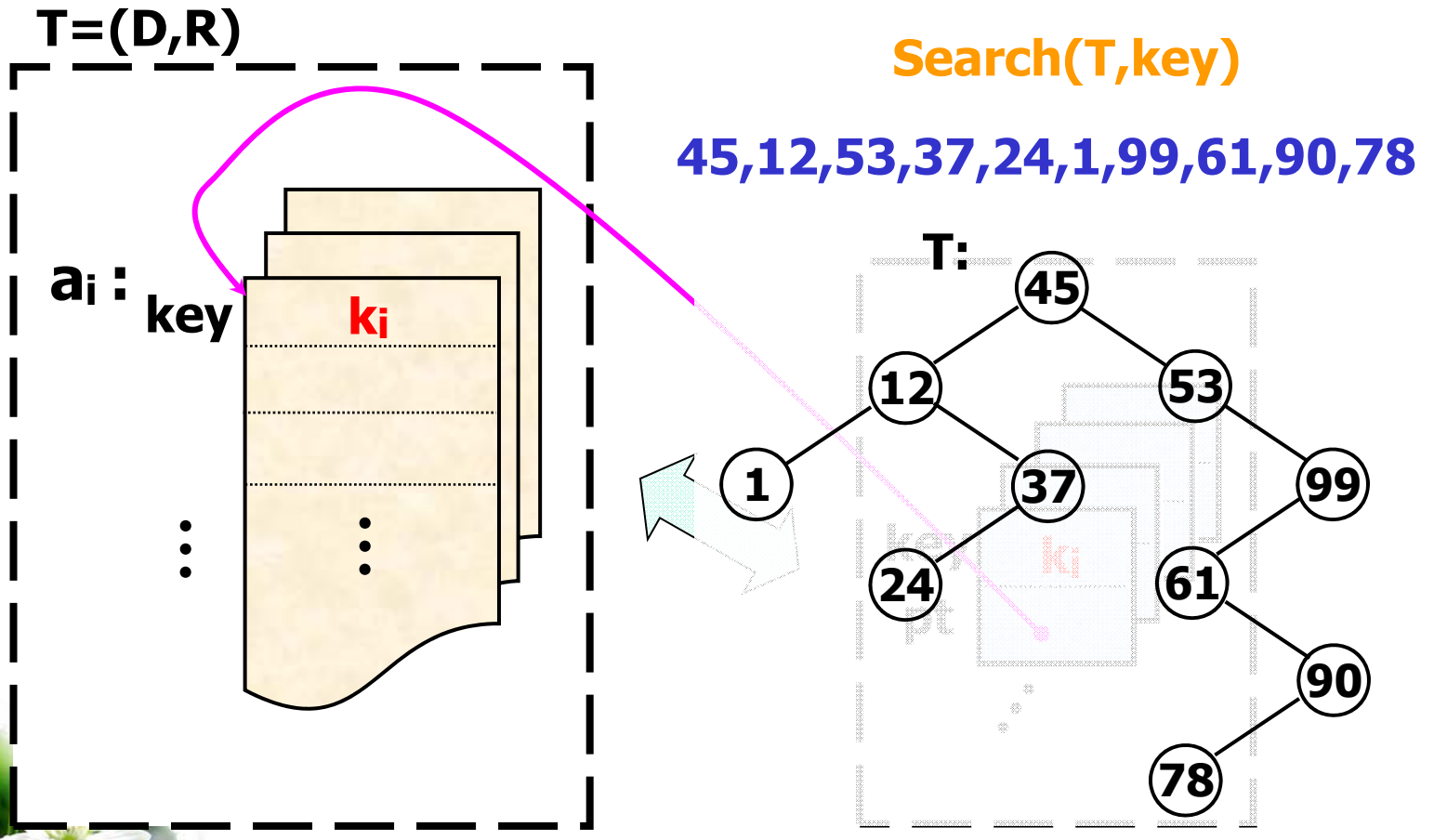
⑴如果**T**的左子树非空，则左子树所有结点值小于**T**的根值；

⑵如果**T**的右子树非空，则右子树所有结点值大于**T**的根值；

⑶**T**的左子树和右子树均为二叉排序树。



二叉排序树查找方案：基于**T=(D,R)**，创建对应的二叉排序树。

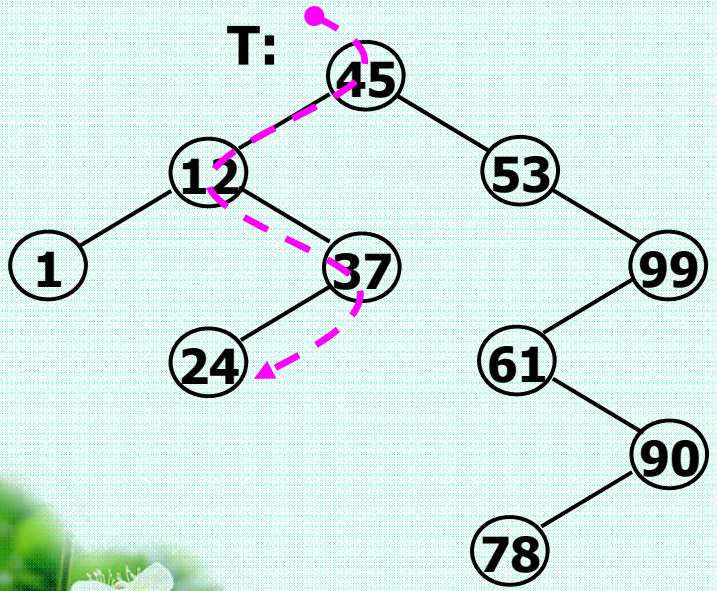


二叉排序树“查找”算法： Locate(T,k)

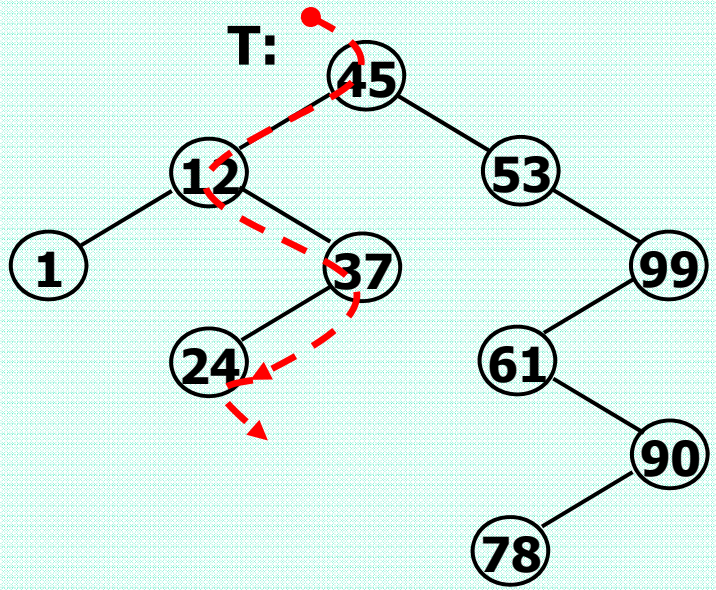
⑴如果排序树**T**为空树，返回“**false**”；

⑵如果排序树**T**之根值>待查找值**key**，则在**T**的左子树上递归查找；否则，在**T**的右子树上递归查找。

例1 Locate(T,24)

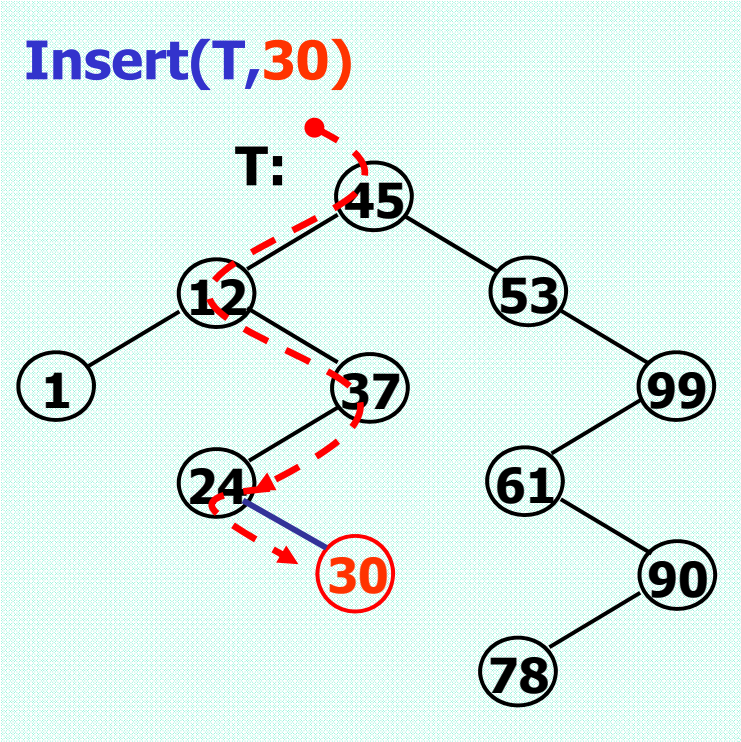


例2 Locate(T,30)



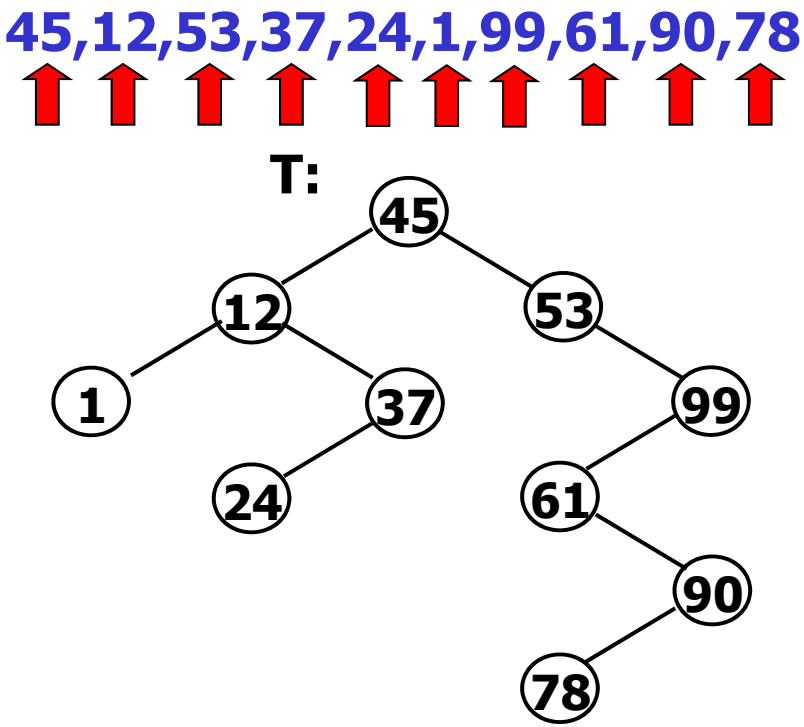
二叉排序树“插入”算法：Insert(&T,k)

根据关键字**k**，在排序树**T**上查找，并在失败处插入之。



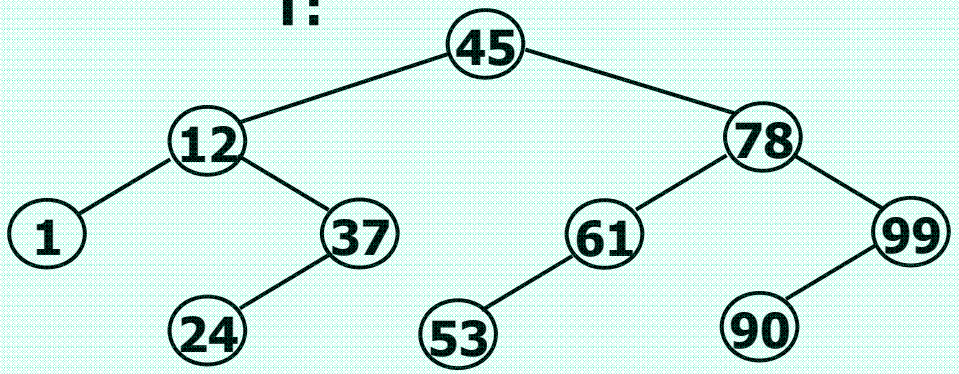
二叉排序树“创建”算法：Create(&T,definit)

根据DS=(D,R)的关键字序列，对每一个关键字k，逐个在排序树T上查找，并在失败处插入之。



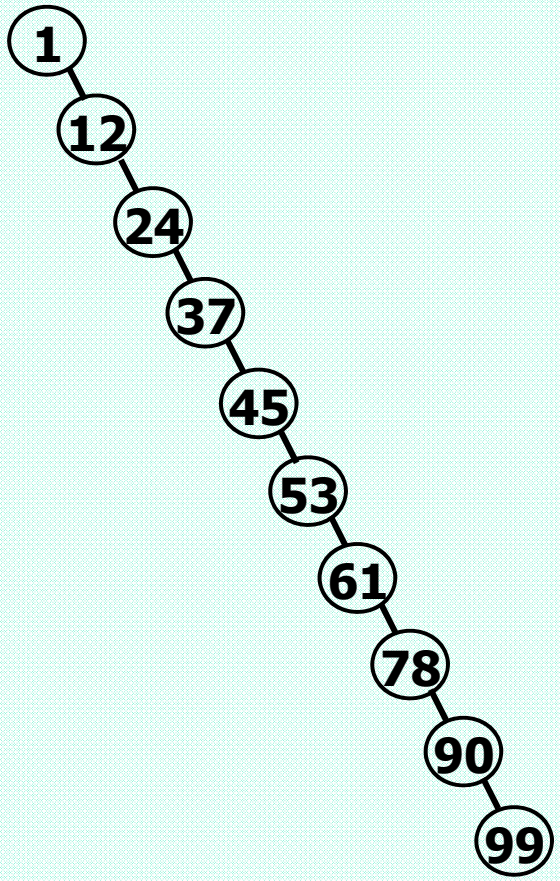
45,12,78,37,24,1,99,61,90,53

T:



1,12,24,37,45,53,61,78,90,99

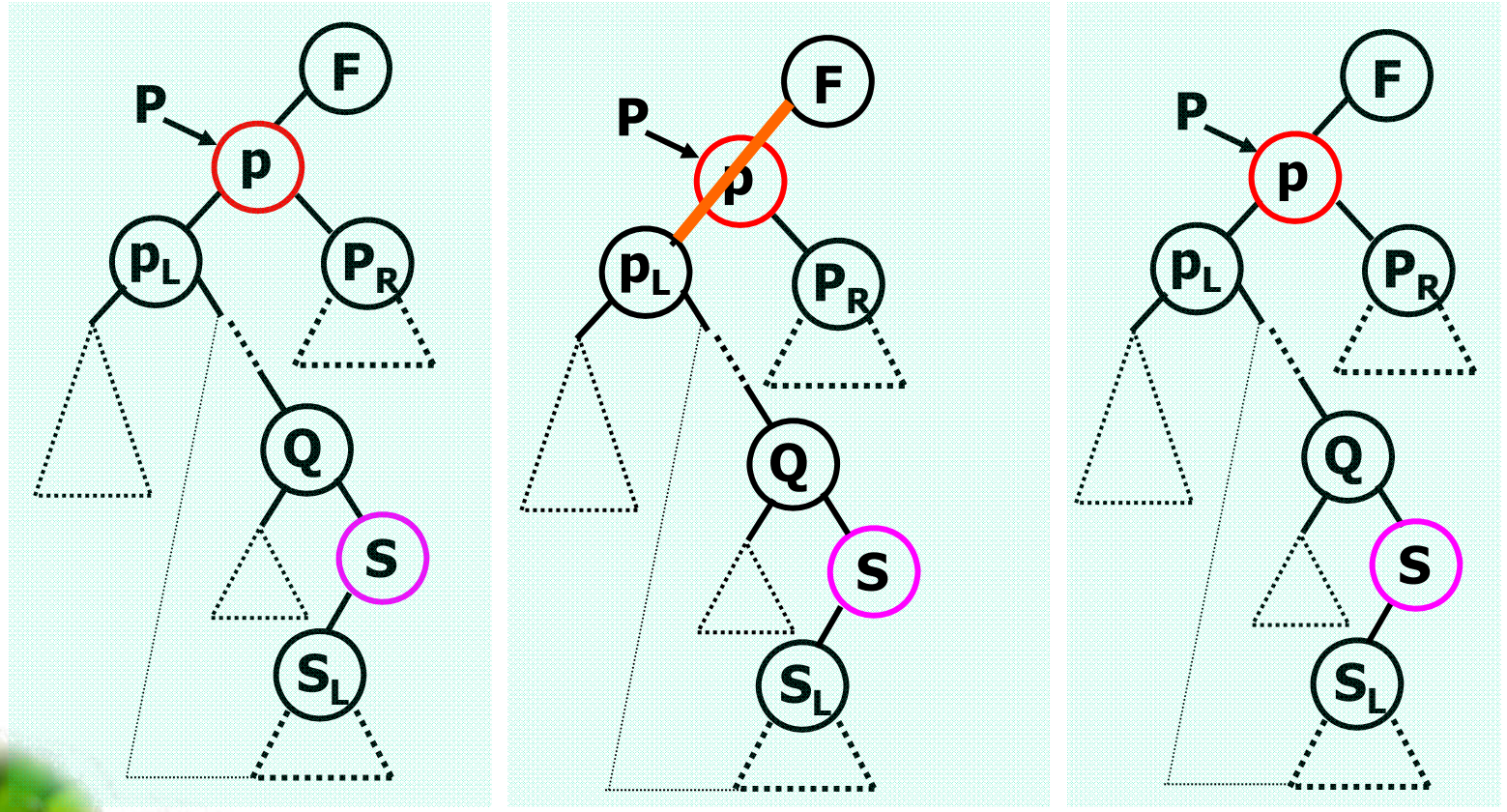
T:



显然，对于n个关键字的n!种排列，对应的二叉排序树的高度区间为：

$Log_2n \sim n$

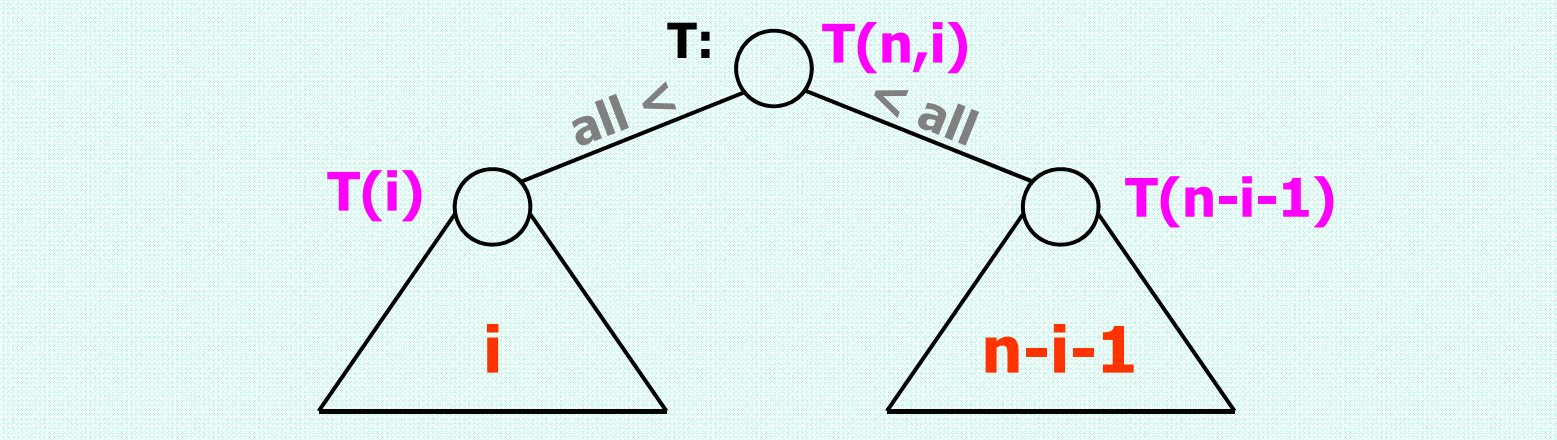
二叉排序树“删除”算法：假设**P**指向被删除的结点。



“查找”算法分析：规模为表长n，统计关键字之间的比较次数。

令 T(n)为在含有n个结点二叉排序树上平均查找长度，

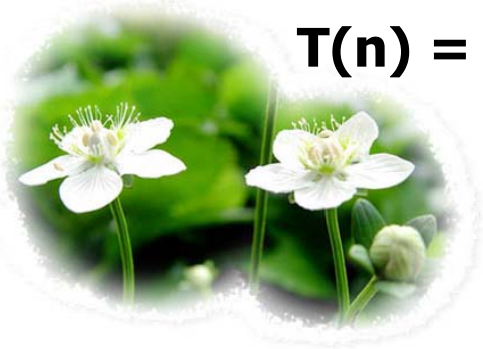
T(n,i)为在含有n个结点的、左子树含有i个结点的二叉排序树上平均查找长度



$$T(n,i) = \frac{1}{n}[1+i\times(T(i)+1)+(n-i-1)\times(T(n-i-1)+1)]$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}T(n,i) = 1+\frac{1}{n^2}\sum_{i=0}^{n-1} i\times T(i) \le c\ln n$$

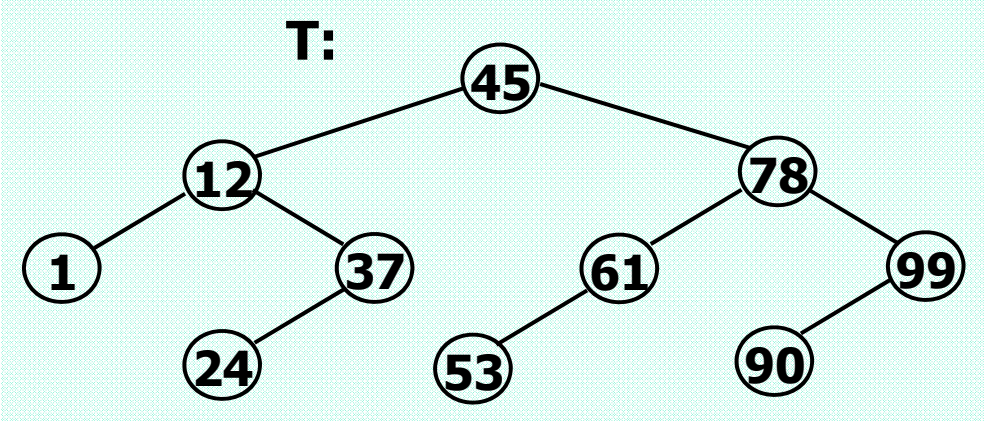
研究表明，这种情况出现的概率为46.5%。



课堂练习

假设关键字序列：**(45,12,78,37,24,1,99,61,90,53)**，试构造二叉排序树，并计算其平均查找长度**ASL**。

解：⑴ 二叉排序树构造如下（过程省略）：



⑵计算**ASL**如下：

$ASL_{成功} = (1\times1+2\times2+3\times4+4\times3)/10 = 2.9$

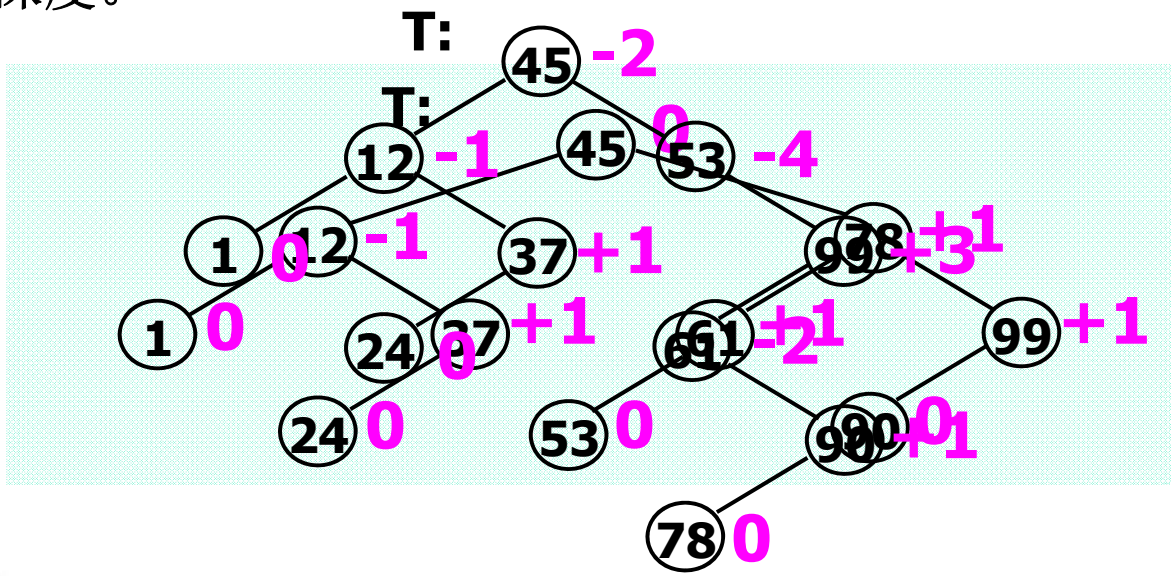
$ASL_{失败} = (3\times5+4\times6)/11 = 39/11\approx3.6$

平衡二叉树(Balancwed Binary Tree)或是一棵空树，或满足下列性质的一棵非空的二叉树**T**：

⑴ **T**的左子树和右子树的深度之差的绝对值不超过**1**；

⑵ **T**的左子树和右子树均为平衡二叉树。

二叉树**T**的结点平衡因子**BF**(Balance Factor)为左子树的深度减去右子树的深度。

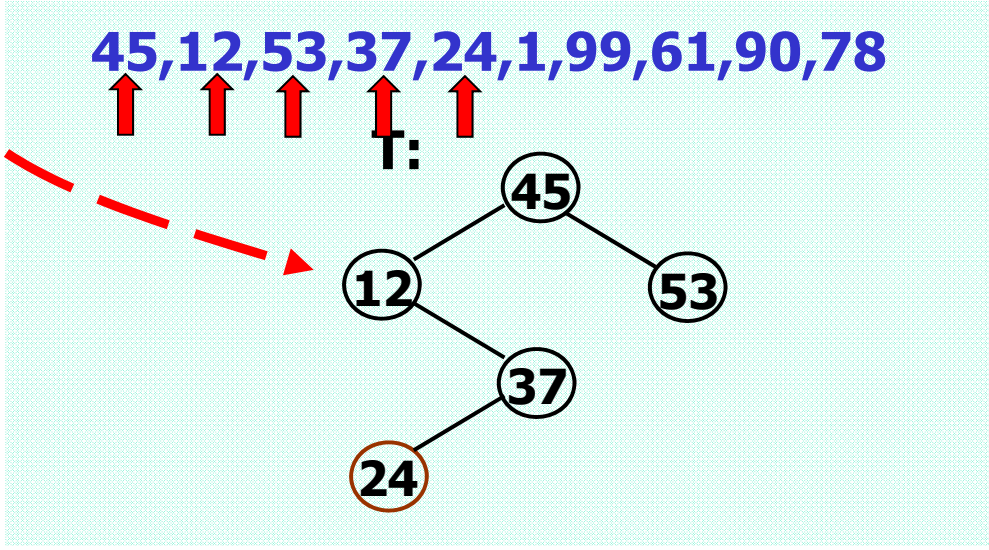


推论：二叉树**T**是平衡二叉树 **iff T**的每个结点**|BF|≤1**

平衡二叉排序树“创建”算法：Create(&T,definit)

根据**DS=(D,R)**的关键字序列，对每一个关键字**k**，逐个在平衡排序树**T**上查找，在失败处插入之；如果**T**失去平衡，对最小不平衡子树进行平衡化处理。

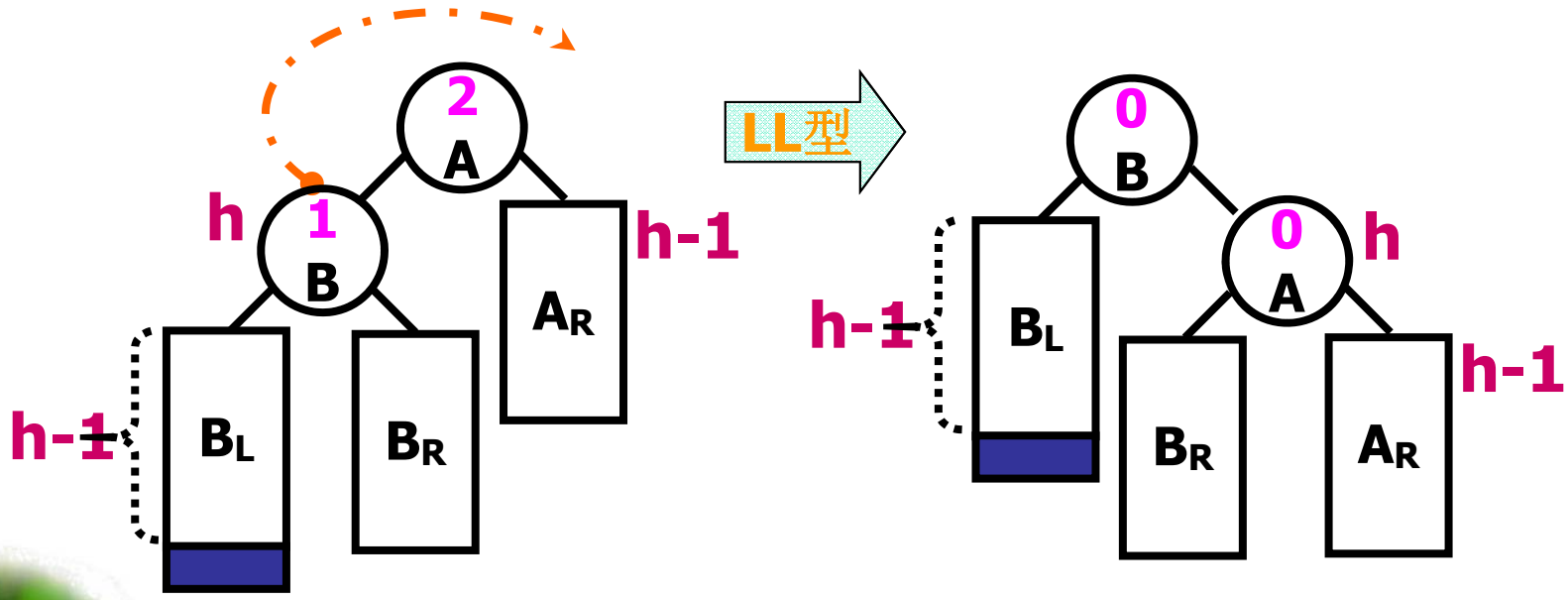
注释：最小不平衡子树是从插入结点**k**到二叉平衡树**T**的根之路径上，距离结点**k**最近的、平衡因子绝对值**>1**的结点为根的子树。



平衡化方法：

当平衡排序树**T**上插入失去平衡时，最小不平衡子树的平衡化处理如下：

(1)单向右旋：插入点是最小不平衡子树的左子树之左子树。

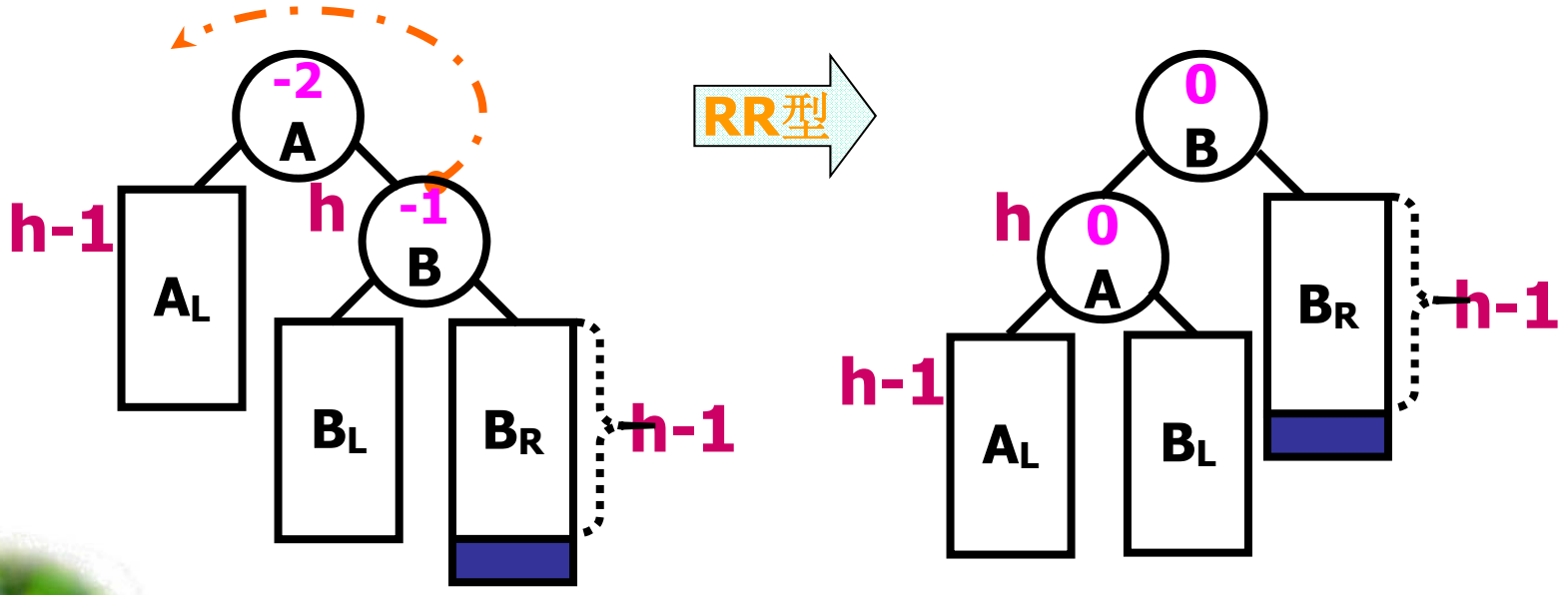


注：h ——最小不平衡子树插入前的左子树之深度

平衡化方法：

当平衡排序树T上插入失去平衡时，最小不平衡子树的平衡化处理如下：

⑵单向左旋：插入点是最小不平衡子树的右子树之右子树。

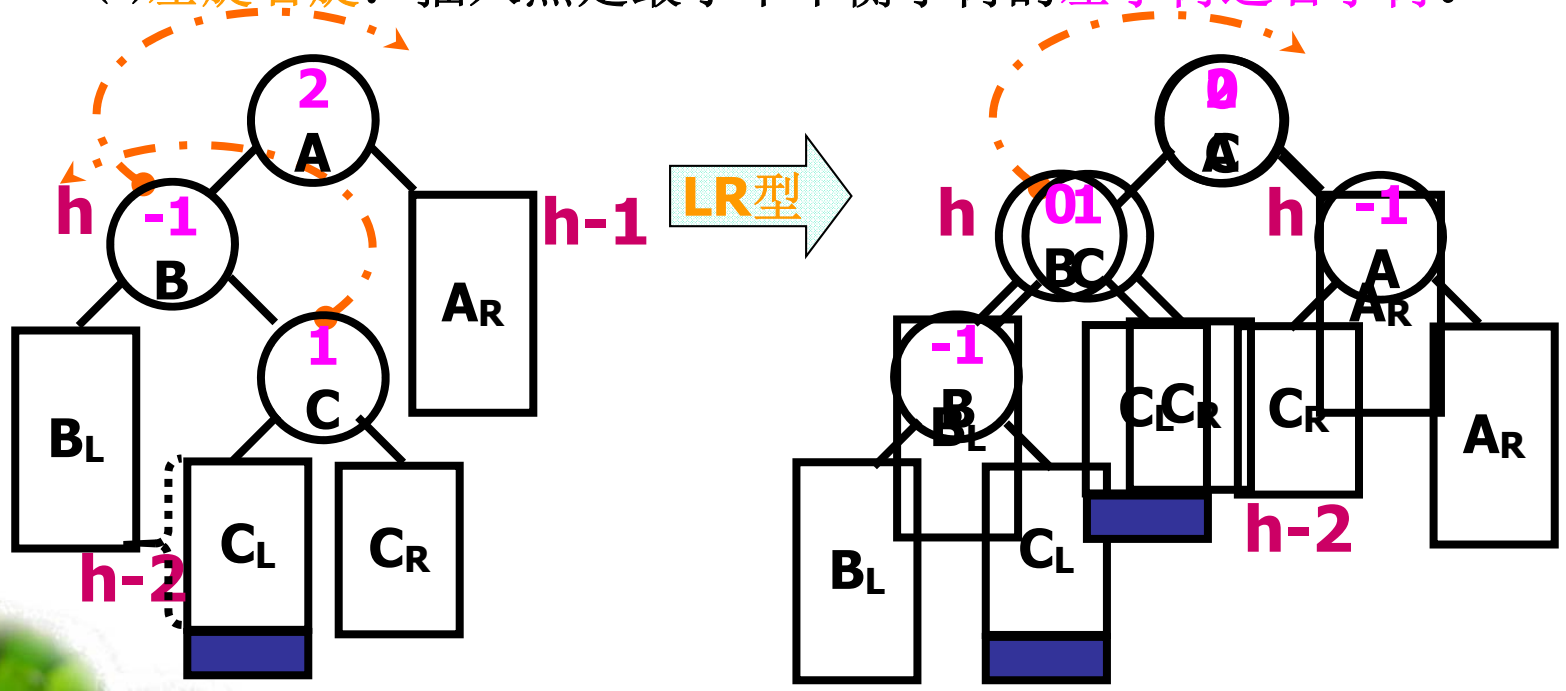


注：h ——最小不平衡子树插入前的右子树之深度

平衡化方法：

当平衡排序树T上插入失去平衡时，最小不平衡子树的平衡化处理如下：

(3)左旋右旋：插入点是最小不平衡子树的左子树之右子树。

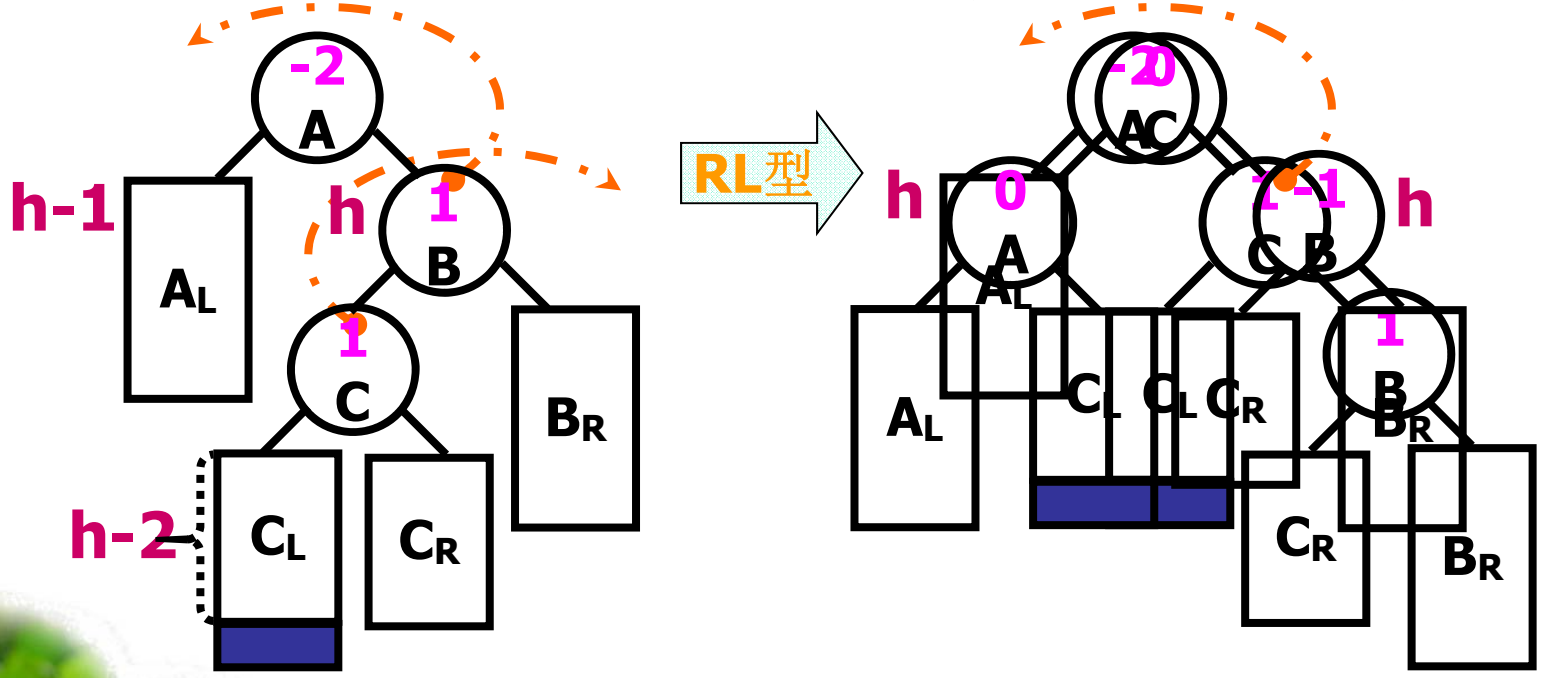


注：h ——最小不平衡子树插入前的左子树之深度

平衡化方法：

当平衡排序树T上插入失去平衡时，最小不平衡子树的平衡化处理如下：

(4)右旋左旋：插入点是最小不平衡子树的右子树之左子树。

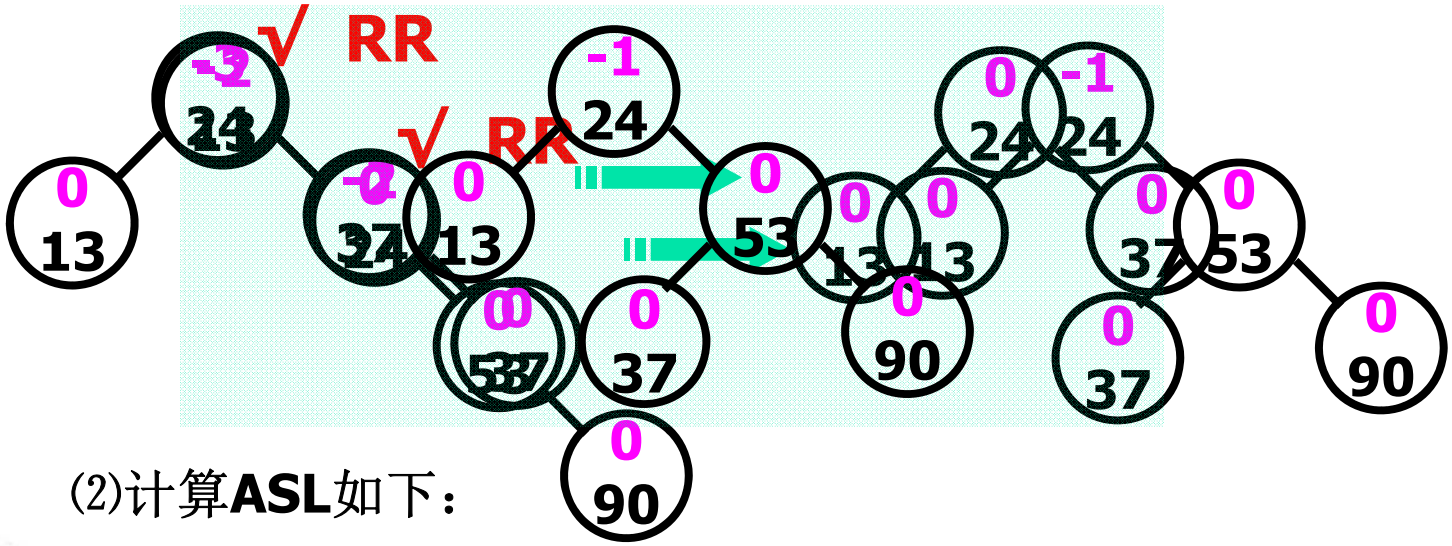


注：h ——最小不平衡子树插入前的右子树之深度

课堂练习

假设关键字序列：**(13,24,37,53, 90)**，试构造平衡的二叉排序树，并计算其平均查找长度**ASL**。

解：(1) 平衡二叉排序树构造如下：



(2)计算**ASL**如下：

$ASL_{成功} = (1\times1+2\times2+3\times2)/5 = 11/5\approx2.2$

$ASL_{失败} = (2\times2+3\times4)/6 = 8/3\approx2.7$

9.2.2 B-树和B+树

B-树的概念

构造B-树或B+树实现查找，这就是B树的查找方法。先讨论B-树及其查找问题， B+树及其查找问题与之极为类似。

一颗m阶的B-树，或为空树，或为满足下列特性的m叉树：

(1) 树中每个结点至多有m棵子树；

(2) 如果根结点不是叶子结点，则至少有两颗子树；

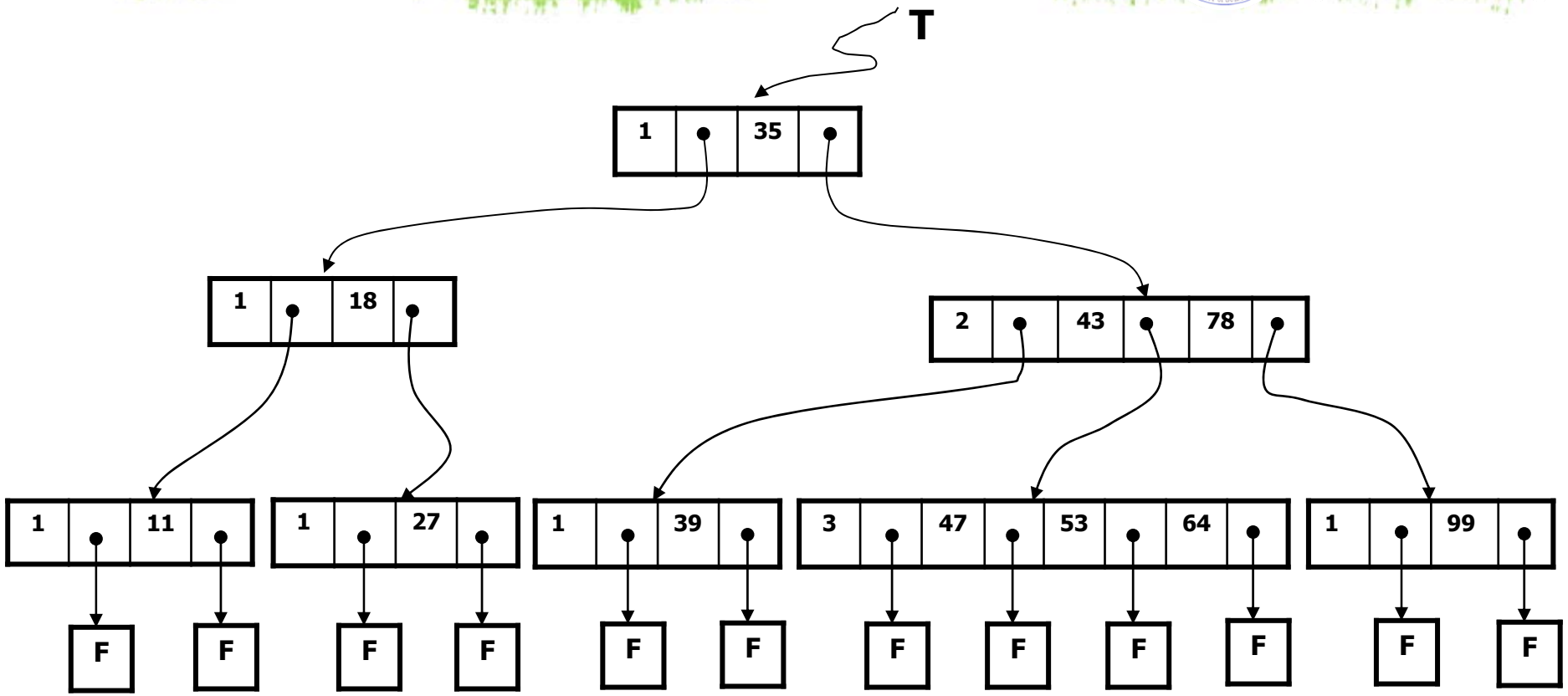
(3) 除了根之外的所有非终端结点至少有「m/2 」棵子树；

(4) 所有非终端结点中包含下列数据项目

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

其中，K_i为关键字$(i:1\sim n)$，且$K_i < K_{i+1}(i:1\sim n-1)$，$A_i(i:0\sim n)$为指向子树根结点的指针，$A_{i-1}$所指子树中的所有结点的关键字均小于$K_i()$，$A_n$所指子树中的所有结点的关键字均大于$K_n$，n(「m/2 」 -1~m-1)为关键字的个数。

(5) 所有叶子结点都在同一层上，不带信息。

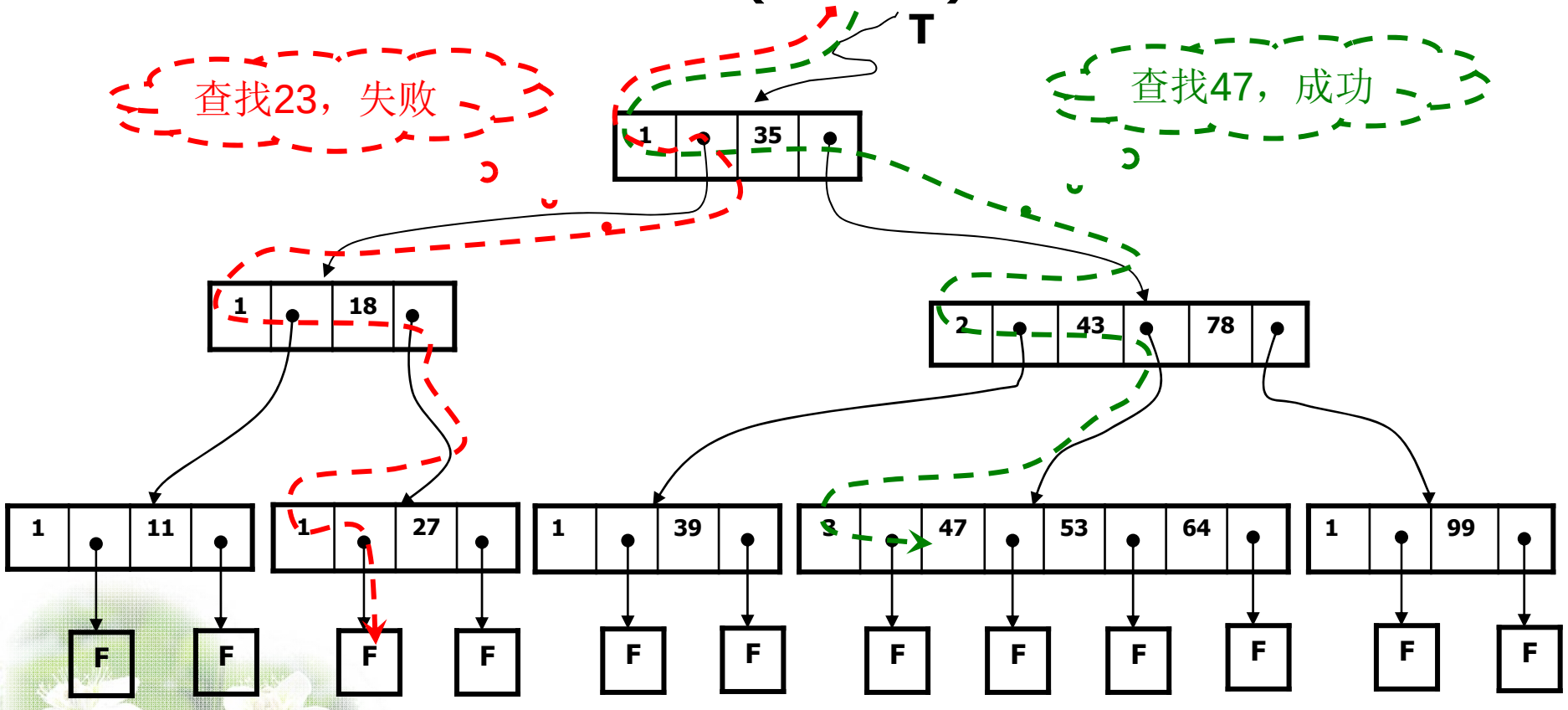


4阶B-树示意图

显然，B-树是一种平衡的多路查找树。除了根结点之外，每个非终端结点含有[「m/2 」~ m]棵子树，叶子结点是空指针指向的、并不存在实际结点，其含义是查找失败的结点。

B-树的查找算法思想

从根结点开始，先在当前结点中查找，如果找到，则查找成功；否则，在当前结点确定可能出现在子树后，从该子树根开始，重复上述过程，直到查找成功，或直到进入叶子结点(即空子树)表示查找失败。



4阶B-树示意图

m阶B-树查找算法的时间性能

查找时间花费为两个部分之和，第一部分是结点内的查找时间，第二部分是m阶B-树查找结点的时间。

第一部分：

结点内关键字的个数n在[「m/2」 -1 ~ m-1]范围内，所以，其花费最坏时间为O(m)或O(logm)。

第二部分：

假设m阶B-树含有N个关键字，树的高度为h。根据m阶B-树的定义，可得第h+1层的结点个数≥2(「m/2」)h-1, 第h+1层均为叶子结点，且结点个数为N+1。即：N+1≥2(「m/2」)h-1。

所以，$h \leq (\log_{\lceil m/2 \rceil}(N+1)/2)+1$，其花费最坏时间为O(logN)。

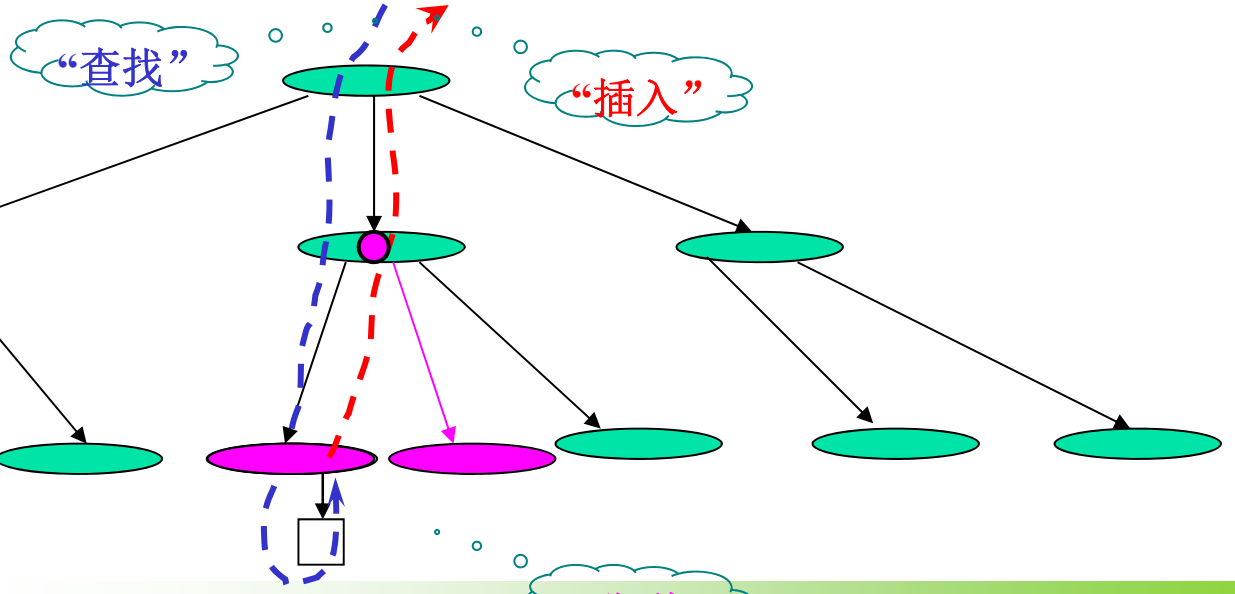
算法最坏复杂度T(N, m) = O(log(m×N))

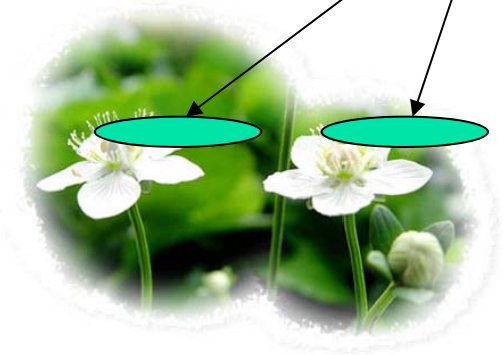
= O(logN)(m视为常量时)

m阶B-树插入算法思想

从根结点开始，查找待插入**key**值，直到叶子结点**(**即失败结点**)**；如果叶子之父结点的关键字个数＜**m**，将**key**插入到叶子之父结点中；如果叶子之父结点的关键字个数＝**m**，将产生结点的“分裂”处理，直到插入**key**后的父结点关键字个数≤**m**、或插入到根结点中为止。

结点的“分裂”处理是将插入key结点，取中间值左右两边的关键字分裂为两个结点，中间值插入其父结点。



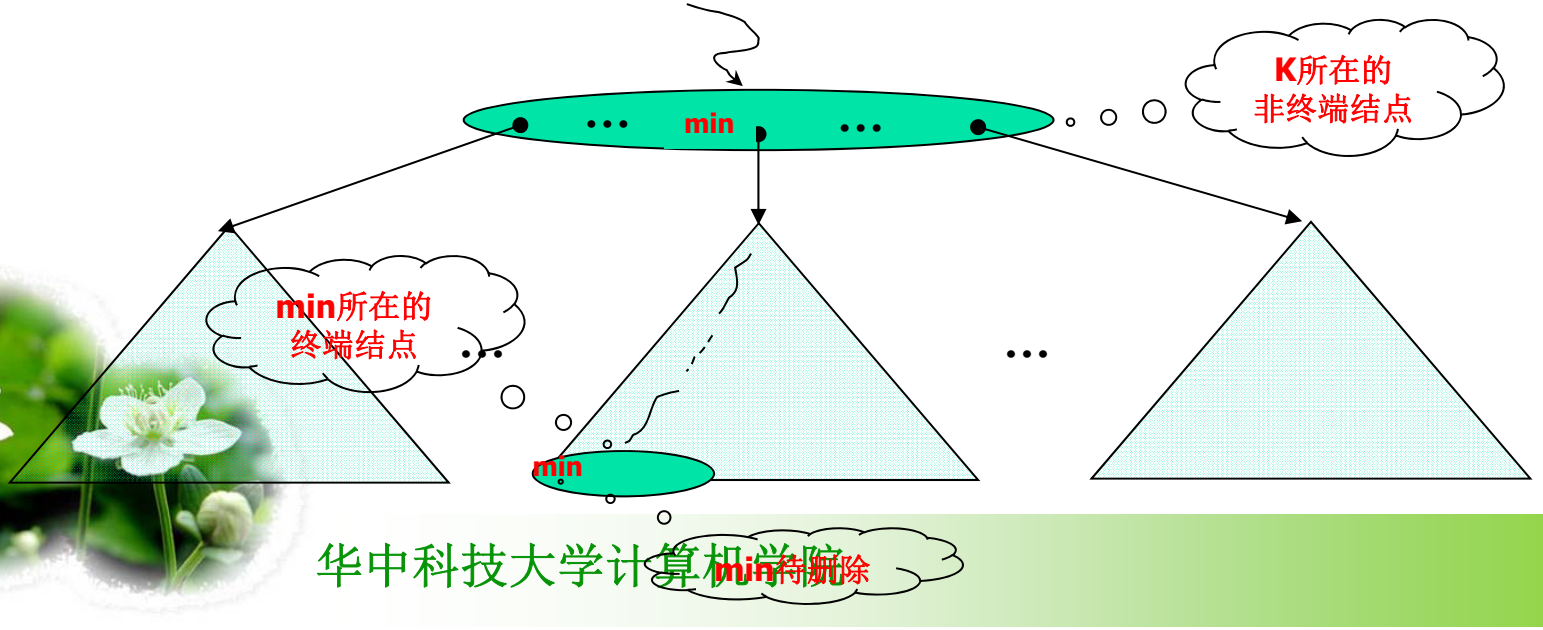


m阶B-树删除算法思想

总的过程是先查找待删除的关键字k所在的结点，之后在该结点“删除关键字k”。

“删除关键字k”分成关键字k所在的结点是非终端结点和终端结点两种情况处理。

对于非终端结点情况，处理的方法是用k后一个指针指向的子树中最小值min替换关键字k，再将min从所在结点中删除。这个最小值min实际上处于终端结点，相当于是对终端结点删除关键字的情况。



m阶B-树删除算法思想

对于终端结点情况，处理的方法是分4种情况分别处理。

① 终端结点关键字个数≥「m/2」

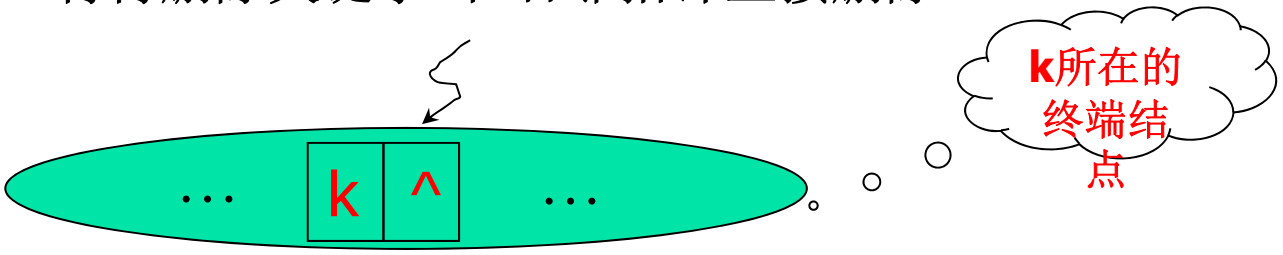
② 终端结点关键字个数＝「m/2」-1，右兄弟结点关键字个数＞「m/2」-1

③ 终端结点关键字个数＝「m/2」-1，左兄弟结点关键字个数＞「m/2」-1

④ 终端结点关键字个数＝「m/2」-1，左右兄弟结点关键字个数＝「m/2」-1

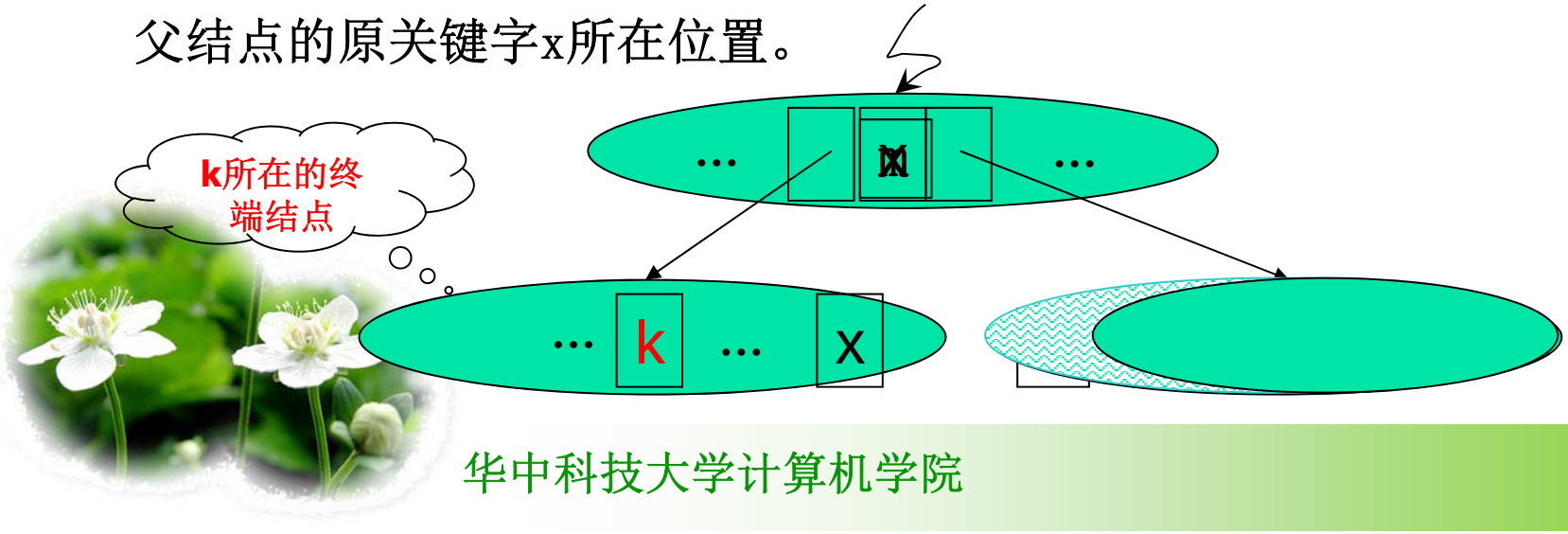
① 终端结点关键字个数≥「m/2」

处理方法：将待删除关键字k和右邻指针直接删除。



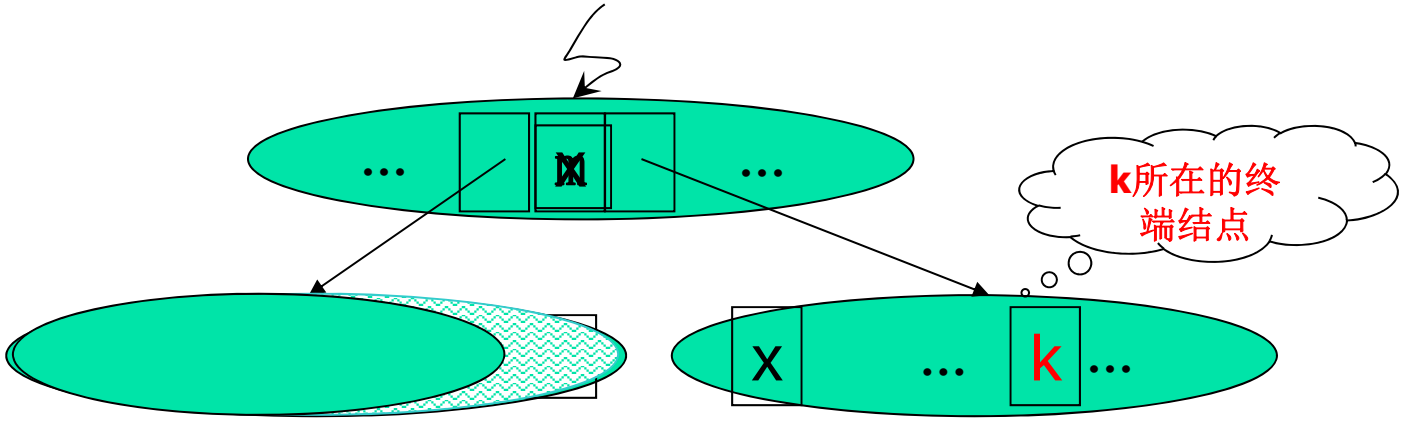
② 终端结点关键字个数＝「m/2」-1，右兄弟结点关键字个数＞「m/2」-1

处理方法：“删除”k，将右兄弟结点父指针左邻关键字x，“插入”到k之前所在结点中，将右兄弟结点中最小关键字m，“上移”至父结点的原关键字x所在位置。



③ 终端结点关键字个数＝「**m/2**」**-1** ，左兄弟结点关键字个数＞「**m/2**」**-1**

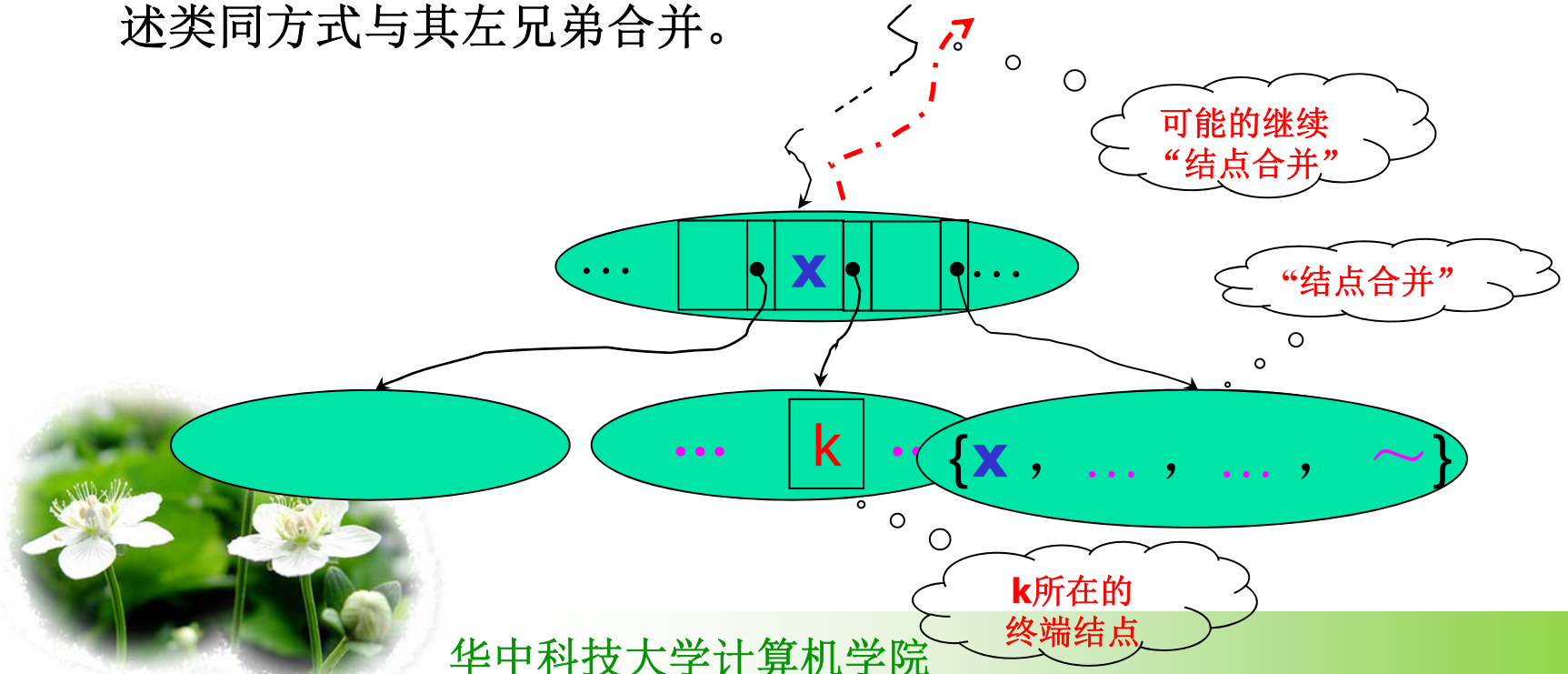
处理方法：“删除”k，将左兄弟结点父指针右邻关键字x，“插入”到k之前所在结点中，将左兄弟结点中最大关键字m，“上移”至父结点的原关键字x所在位置。



说明：情况②和情况③的处理方式极其相似，前者可以理解为“顺时针”方向；后者为“逆时针”方向。

④ 终端结点关键字个数＝「**m/2**」**-1**，左右兄弟结点关键字个数＝「**m/2**」**-1**

处理方法：如果右兄弟存在，则将待删除关键字k所在结点删除k后的数据，加上父结点指向右兄弟结点左邻关键字x，一起“合并”到右兄弟中 。合并后如果父结点的关键字个数小于了「m/2 」-1，则父结点依次做上述相同的“结点合并”，直到结点关键字个数不小于「m/2 」-1 为止，或直到根结点为止。如果右兄弟不存在，用上述类同方式与其左兄弟合并。



B+树的概念

一颗m阶的B+树，是B-树的变型树。一颗m阶的B+树与一颗m阶的B-树的差别在于：

⑴ 有n棵子树的中含有n个关键字；

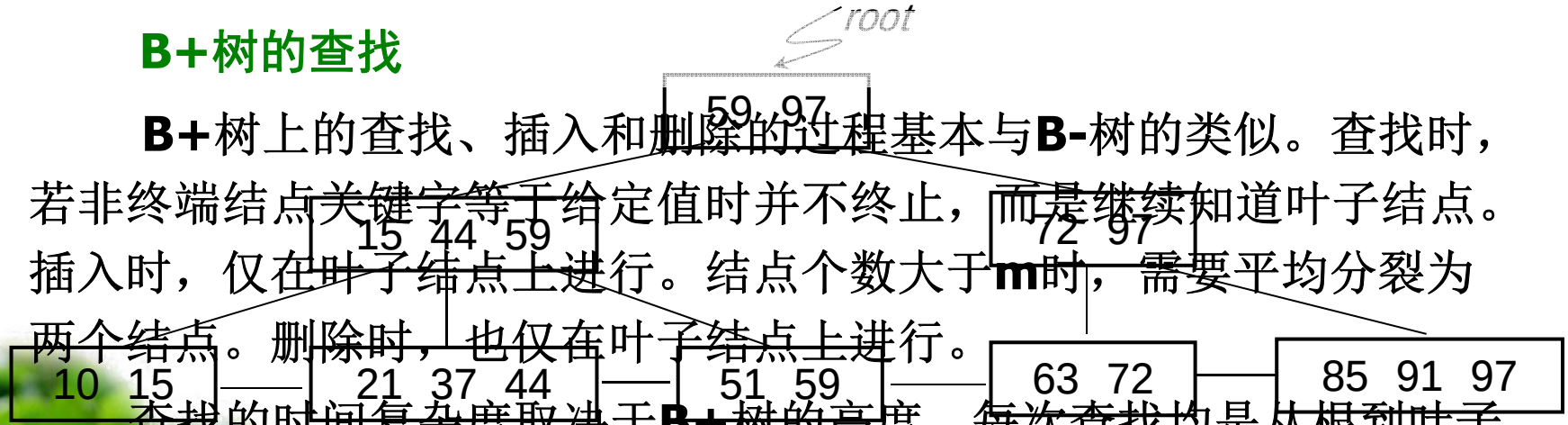
⑵ 所有叶子结点中包含全部关键字信息，及指向含有这些关键字记录的指针，且叶子结点依关键字大小顺序链接；

⑶ 所有非终端结点中仅含有其子树中的最大(或最小)关键字。

B+树的查找

B+树上的查找、插入和删除的过程基本与B-树的类似。查找时，若非终端结点关键字等于给定值时并不终止，而是继续知道叶子结点。插入时，仅在叶子结点上进行。结点个数大于m时，需要平均分裂为两个结点。删除时，也仅在叶子结点上进行。

查找的时间复杂度取决于B+树的高度，每次查找均是从根到叶子。



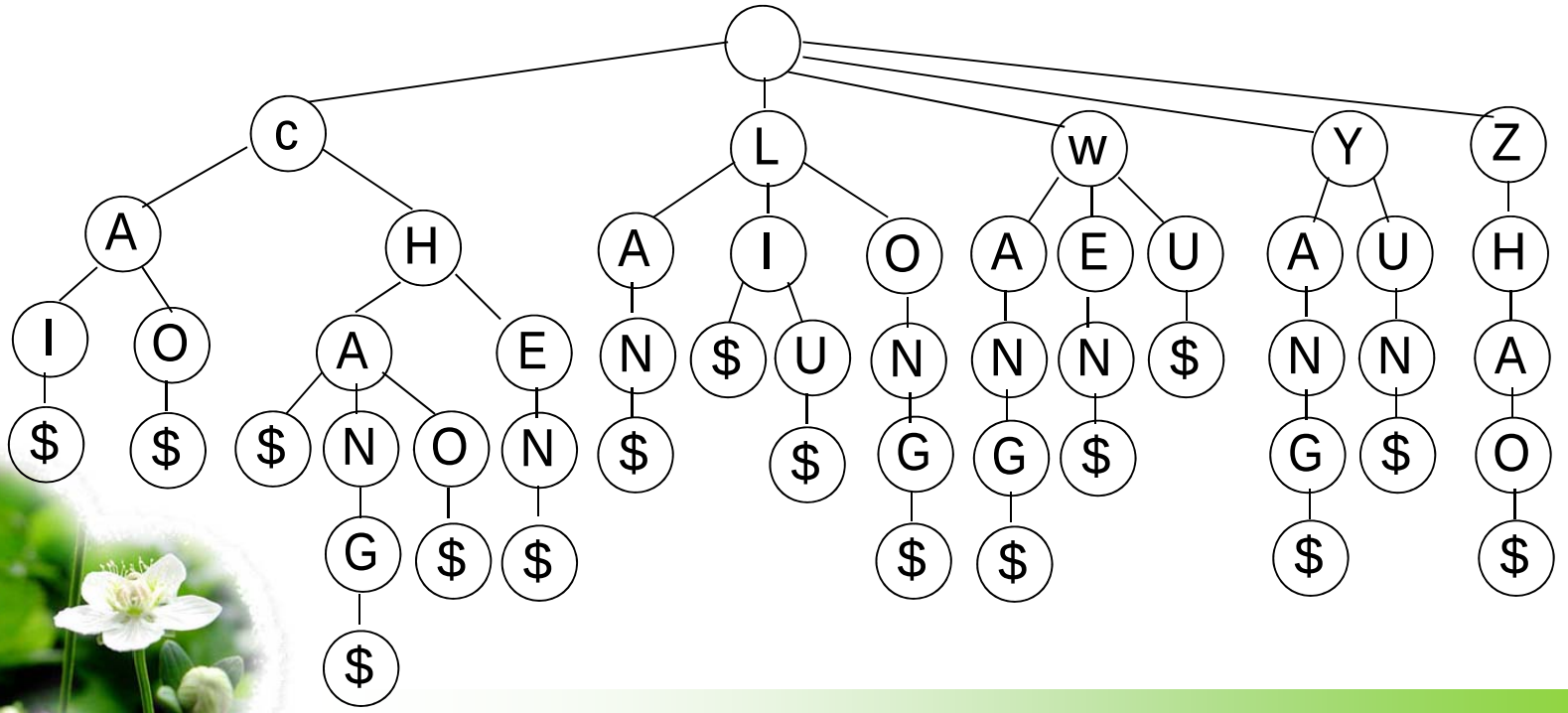
一颗3阶的B+树

9.2.3 键树

键树是一颗度数≥2的树，树中每个结点中仅含有一个关键字组成的符号。键树又称为数字查找树。

对于下列16个关键字的集合，构成一颗键树如下。其中，$表示串结束。

{CAI,CAO,LI,LAN,CHA,CHANG,WEN,CHAO,YUN,YANG,LONG,WANG,ZHAO,LIU,WU,CHEN}



键树有(1)“二叉链表”和(2)“多重链表”两种存储结构。

(1)“二叉链表”存储结构：即采用树的左孩子有兄弟的表示法，以二叉链表存储。

查到方法：假定结点结构中**first**指向孩子结点，**next**指向兄弟结点，查找关键字**k=d1d2…dn**的方法是从根出发，从**first**指针找到第一棵子树， 第一个字**d1**与子树根值比较，如果相等，继续顺**first**再比较下一个字符**di**；否则，沿**next**指针顺序查找；直到最后最后一个字符**dn**比较相等，则查找成功。

如果直到“空”（**$**）仍然比较不等，则查找失败。

时间效率分析：键树的最大度 **d** 取决于关键字的基数，高度**h**取决于关键字的位数。假定关键字每一位取每个字符是等概率，则查找每位的平均查找长度为 **(1+)/2** 。假定关键字位数相同是等概率，则查找关键字的平均查找长度为**h(1+)/2**。

(2)“多重链表” 存储结构：树的每个结点含有 **d** 个指针域。如果从键树中某结点到叶子路径上每个结点都仅有一个孩子，将该路径上的所有结点压缩为一个叶子结点，即该叶子结点中存储关键字。

此时，键树又称为**Trie**树。

查到方法：从根结点出发，沿和给定相应的指针逐层向下，直到叶子结点；如果叶子结点中的关键字与给定值比较相等，则查找成功；

如果叶子结点中的关键字与给定值比较不相等，或者分子结点中与给定值对应的指针为空，则查找失败。

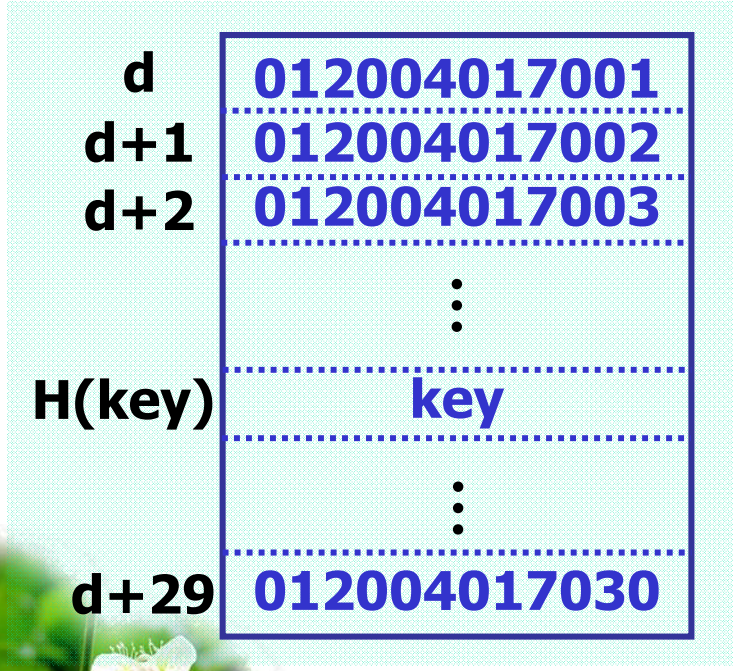
时间效率分析：查找过程是从根结点出发，走了一条从根到叶子的路径，其查找时间依赖于**Trie**树的深度。

说明：上述两种存储结构均具有各自的特点，如果键树的度较高时，采用**Trie**树较为适合。

9.3 哈希表

9.3.1 什么是哈希表

哈希表查找法思路是根据关键字计算其存储的地址，从而获得对应数据元素之存储地址。



$H(key)=d+key-key_0$

(key_0= 012004017001)

核心问题：

(1) 设计Hash函数

(2) 解决Hash冲突

具有相同的哈希地址的关键字具有相同的哈希地址。

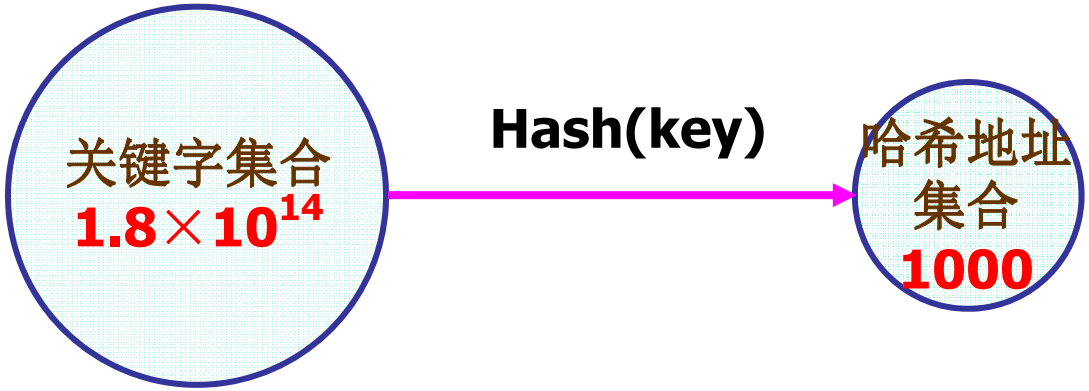
Hash冲突

同义词

Hash冲突原因：

⑴ **Hash**函数的定义域为关键字的理论取值范围，值域为哈希地址集合。

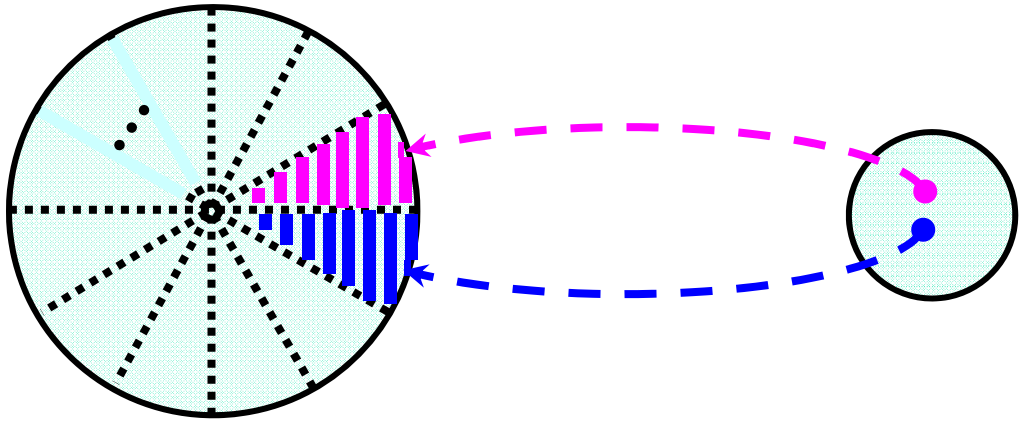
⑵ 通常，关键字的理论取值范围远远大于实际取值范围。



例如，C语言的8位长度标识符集合大小＝52×62^7，一个源程序实际出现的标识符数远远小于此数！

9.3.2 哈希函数的构造方法

如果对于关键字集合的每个关键字**key**，经哈希函数**H(key)**映射到哈希地址集合中的任何一个地址之概率均是相等的，则称哈希函数**H(key)**为均匀的(Uniform)哈希函数。



注：① 每类同义字的个数相等或大致相等！

② “均匀性”在于较少冲突。

(1) 直接定址法

$$H(key) = a \times key + b \quad (a \neq 0)$$

(2) 数字分析法

假定事先知道可能出现的关键字子集，分析这些关键字的每一位，选择其中“若干”“随机”位构成其哈希地址。

k_1:	8	1	3	4	6	5	3	2
k_2:	8	1	3	7	2	2	4	2
k_3:	8	1	3	8	7	4	2	2
k_4:	8	1	3	0	1	3	6	7
k_5:	8	1	3	2	2	8	1	7
k_6:	8	1	3	3	8	9	6	7
k_7:	8	1	3	5	4	1	5	7
k_8:	8	1	3	6	8	5	3	7
k_9:	8	1	3	1	9	3	5	5
	①	②	③	④	⑤	⑥	⑦	⑧

$H(k_3)=87$

提示：

✓随机性可计算

✓可结合叠加法

(3) 平方取中法

取关键字平方后的中间若干位为哈希地址之方法。

(4) 折叠法

将关键字分割成位数相同的若干个段，各段叠加求和为希地址之方法。

key: | p_1 | p_2 | p_3 | p_4 | p_5 | p_6 |

$$H(key) = p_1+p_2+p_3+p_4+p_5+p_6$$

或： $$H(key) = p_1+p_2'+p_3+p_4'+p_5+p_6'$$

(4) 余数法

取关键字被不大于哈希表长**m**的某个数**p**出后的余数为希地址之方法。

$$H(key) = key \ MOD \ p \quad (p \le m)$$

提示：**p**取为质数或不含小于**20**质因子的合数！

9.3.3 处理冲突的方法

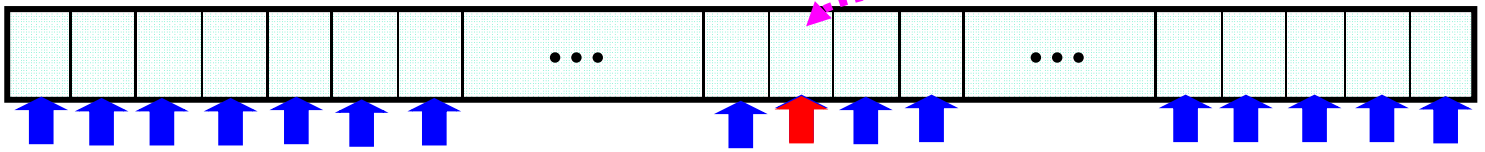
(1) 开放地址法

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

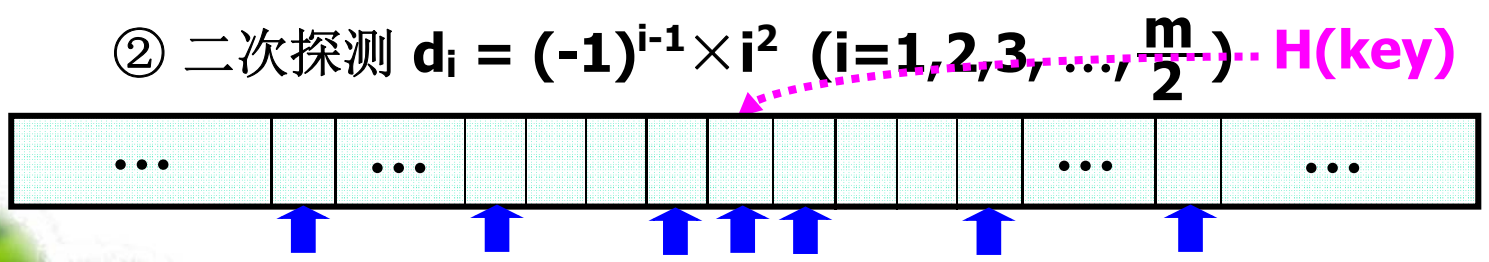
(H(key)为哈希函数，m为哈希表长，d_i为增量)

增量d_i取法如下：

① 线性探测 $d_i = i$ (i=1, 2, 3, … , m-1)



② 二次探测 $d_i = (-1)^{i-1} \times i^2$ $(i=1,2,3,\ldots,\frac{m}{2})$



③ 伪随机数探测

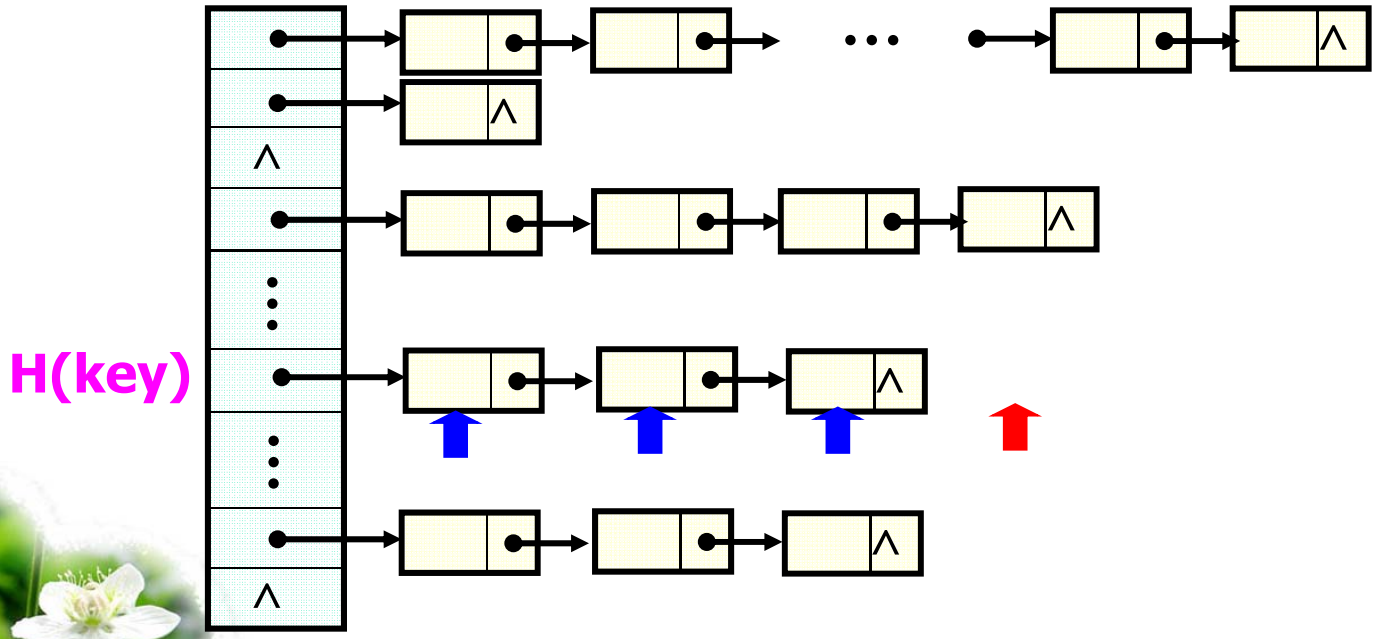
(2) 再哈希法

$$H_i(key) = RH_i(key) \quad (i=1,2,3,\ldots,k)$$

($RH_i(key)$为不同的哈希函数)

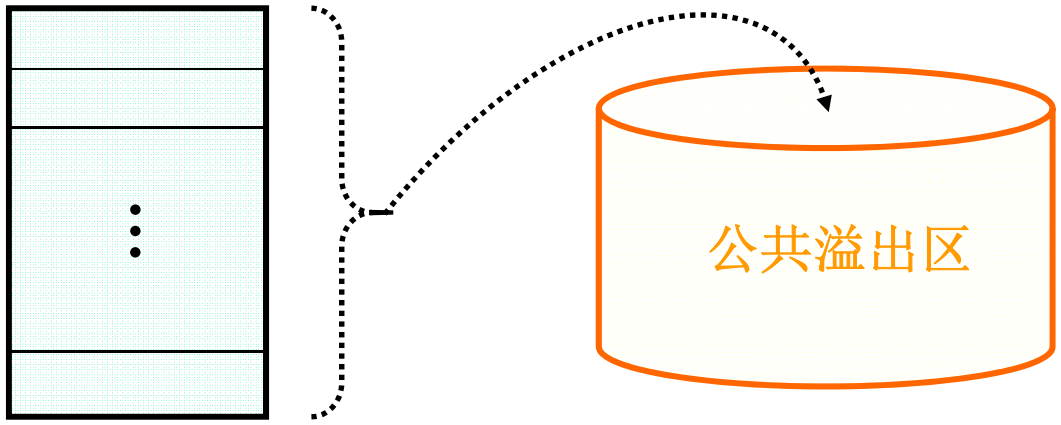
(3) 链地址法

将关键字的同义词存储在各自的单链表中。



(4) 公共溢出区法

当增加关键字到哈希表中遇到冲突时，将所有同义词统一存储到“公共溢出区”中。



Hash冲突解决
- (1) 开放地址法
- (2) 再哈希法
- (3) 链地址法
- (4) 公共溢出区法

9.3.4 哈希查找及其分析

哈希“查找”算法：

⑴ 在给定值**key**对应计算的哈希地址**H(key)**处查找；

⑵ 如果⑴查找不成功，根据处理冲突方法确定“下一个地址”处查找；

⑶ 重复⑵，直到查找成功、或遇到“结束标志”为止。⑤

说明：

① “下一个地址”和“结束标志”是由处理冲突方法决定的；

② 查找算法中既使用了“算术”运算，又使用了“比较”运算；

③ “插入”算法是在查找失败处增加新关键字；

④ “创建”算法可以循环调用“插入”运算实现。

课堂练习

设关键字序列(19,14,23,01,68,20,84,27,55,11,10,90)，哈希函数H(key)=key MOD 13，哈希表长16，并采用线性探索法处理冲突，试构造哈希表，并计算其平均查找长度ASL。

解：⑴ 构造哈希表如下：

key	19	14	23	01	68	20	84	27	55	11	10	90
H(key)	06	01	10	01	03	07	06	01	03	11	10	12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84		23	11	10	90		

⑵计算ASL如下：

$ASL_{成功}$ = (1+1+1+2+1+1+3+4+3+1+3+2)/12

$ASL_{失败}$ =

(0+8+7+6+5+4+3+2+1+0+4+3+2)/13

哈希表中填入的关键字个数与哈希表长之比，称为哈希表的装填因子。

线性探索法：

$$\text{ASL成功} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

$$\text{ASL失败} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

二次探索法：

$$\text{ASL成功} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

$$\text{ASL失败} \approx \frac{1}{1-\alpha}$$

链接地址法：

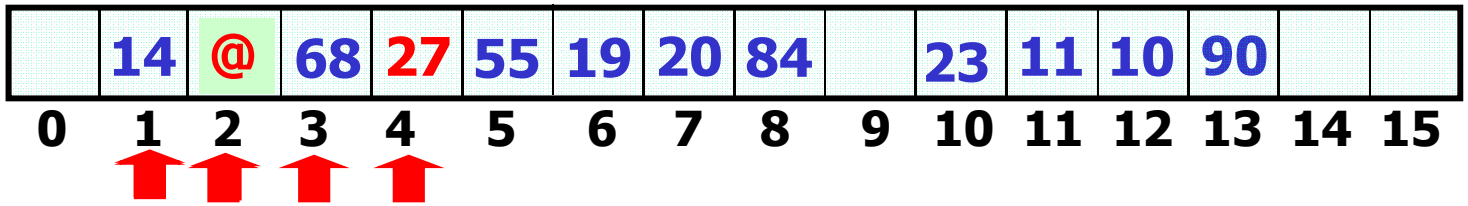
$$\text{ASL成功} \approx 1+\frac{\alpha}{2}$$

$$\text{ASL失败} \approx \alpha+e^{-\alpha}$$

特别说明：

(1) “删除”运算的实现是在查找成功处删除之。对于某些处理冲突的方法，删除实际上是填入“删除标志”，并且“查询”算法需要相应调整。

例如，删除01后，查找27。



(2) 特殊情况下，存在无处理冲突的哈希函数。

例如，**Pascal**保留字集的哈希函数如下：

$$H(key) = L + G(key[1]) + G(key[L])$$

其中，**L**是保留字长度，**key[i]**是保留字的第**i**个字符，**G(x)**字符转换称数字函数。

小结

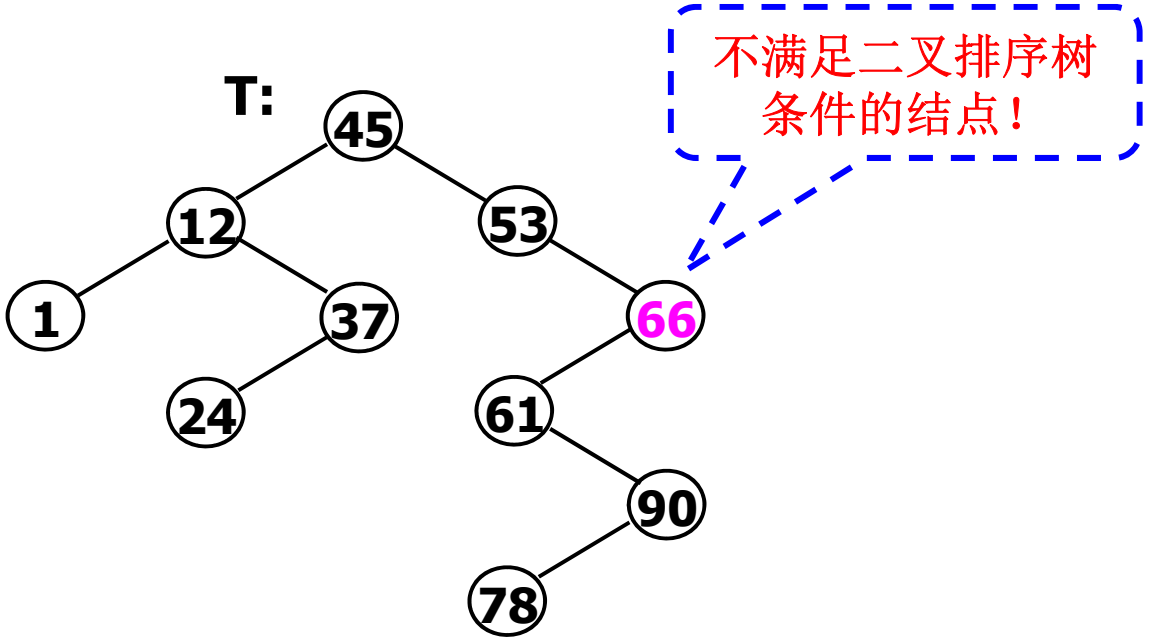
本章重点介绍了数据结构研究对象、内容和方法，并重点讨论了数据元素存储结构和算法效率估算方法。

数据逻辑结构基本类型划分为集合结构、线性结构、树形结构和图状结构。

研究的主要内容是（**1**）数据的逻辑结构、（**2**）逻辑结构上定义的运算、（**3**）数据的物理结构、（**4**）逻辑结构与物理结构的对应关系和（**5**）运算基于物理结构的实现算法及效率分析。

提出的基本概念是数据、数据元素、数据对象、逻辑结构、关系、物理结构、数据类型、算法和复杂度等。

重点掌握的内容是①基本概念；②数据元素存储结构；③算法效率估算方法。


T:
45
12
53
1
37
66
24
61
90
78
不满足二叉排序树
条件的结点！