

前面我们学习了三种基本数据结构：

■ 线性表

- 一般线性表（数据元素、关系、操作均无限制）
- 特殊：栈和队列（操作有限制：两端插入删除）
- 特殊：字符串（数据元素有限制带来操作对象变化）
- 特殊：数组（参与关系有限制：参与多个关系）

■ 树和二叉树结构

■ 图结构

讲授和学习的方法：

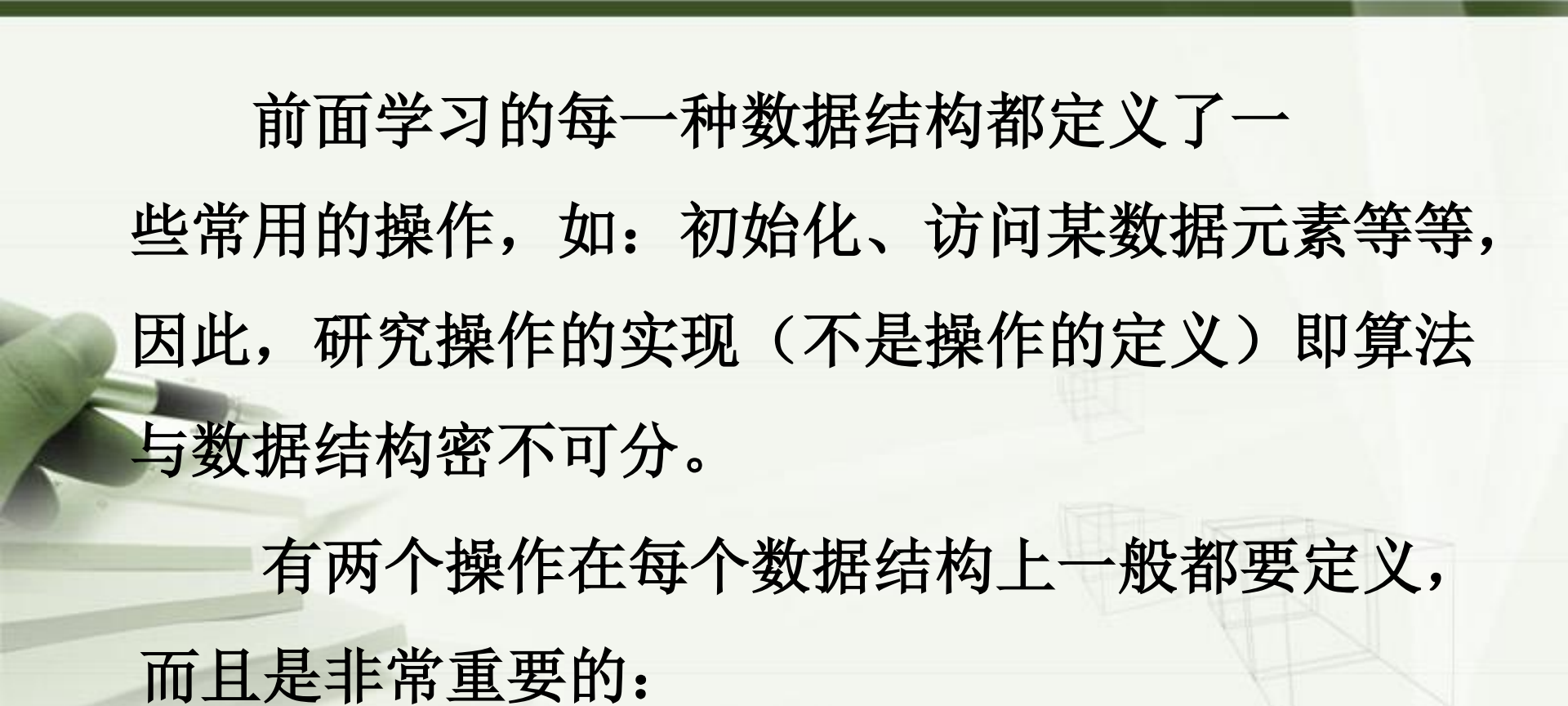
■ ADT的定义：

- 数据结构的逻辑特性；
- 数据结构上定义的操作；

■ ADT的实现：

- 逻辑结构的实现（存储结构）
- 操作的实现（算法）；

■ ADT应用举例：



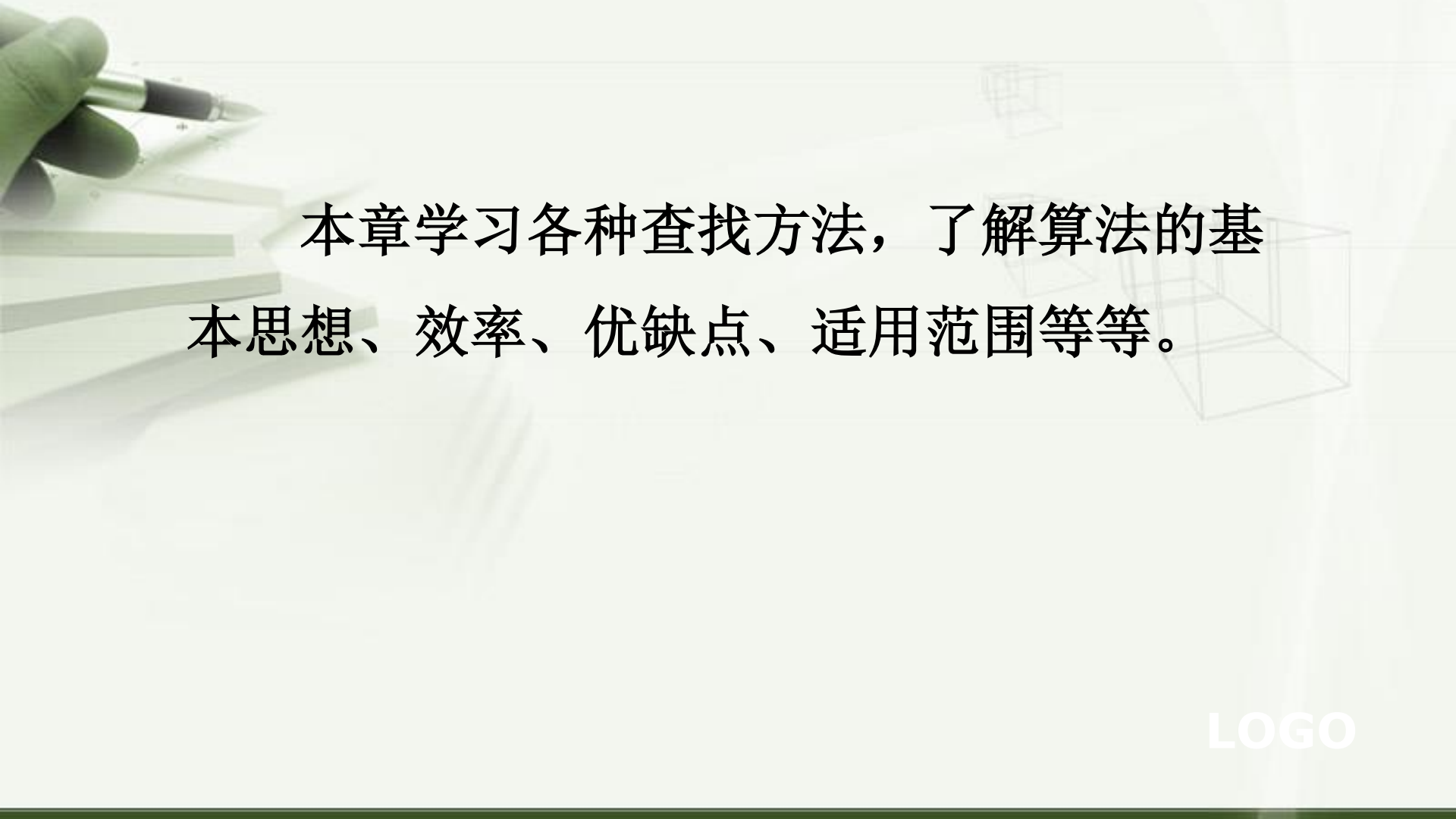
前面学习的每一种数据结构都定义了一些常用的操作，如：初始化、访问某数据元素等等，因此，研究操作的实现（不是操作的定义）即算法与数据结构密不可分。

有两个操作在每个数据结构上一般都要定义，而且是非常重要的：

- 确定元素的位置——搜索（查找）；
- 将元素按某种顺序排列——排序（分类）

学习和掌握操作实现（算法）的基本思想、效率、优缺点

第七章 搜索（查找）



本章学习各种查找方法，了解算法的基本思想、效率、优缺点、适用范围等等。

7.1 基本概念

1. **查找**：在某一数据集合中查找数据元素是否存在，若存在，返回其位置，否则，返回失败信息。

2. **查找表**：被查找数据元素的集合，如果是逻辑结构上定义的操作，那么位置就是逻辑位置；如果有了存储结构，那么位置应该就是物理位置。

因此，查找表一般是“数据结构+存储结构”

3. 查找表的种类

静态查找表：数据集合在查找前后不变(没有插入、删除)；

动态查找表：数据集合在查找后会改变(有插入、删除)；

7.1 基本概念

4. **关键码**： 可以唯一地标识一个数据元素的数据项（属性）。

- 基于关键码的查找： 结果唯一；
- 基于一般属性的查找： 结果可能不唯一；

5. **查找方法**：

查找表不同，查找方法就会不同。有很多不同的查找方法。

7.1 基本概念

6. 查找算法的评价:

空间: 占用的辅助空间少;

时间: 时间少。查找的基本操作是比较, 因此时间主要体现为比较次数。

查找成功:

最大比较次数——MSL(Maximum Search Length)

平均比较次数——ASL(Average Search Length)

查找失败:

最大比较次数——MSL

平均比较次数——ASL

7.2 静态表的查找

7.2.1 静态查找表是顺序或链式存储的线性表 ——顺序查找

1. 查找表的要求：线性表
2. 查找方法：不断找前驱（或后继）即可
3. 特点：
 - 思想简单，对查找表要求少，适应面广；
 - 比较次数较大 $O(n)$ ——成功、失败

注：

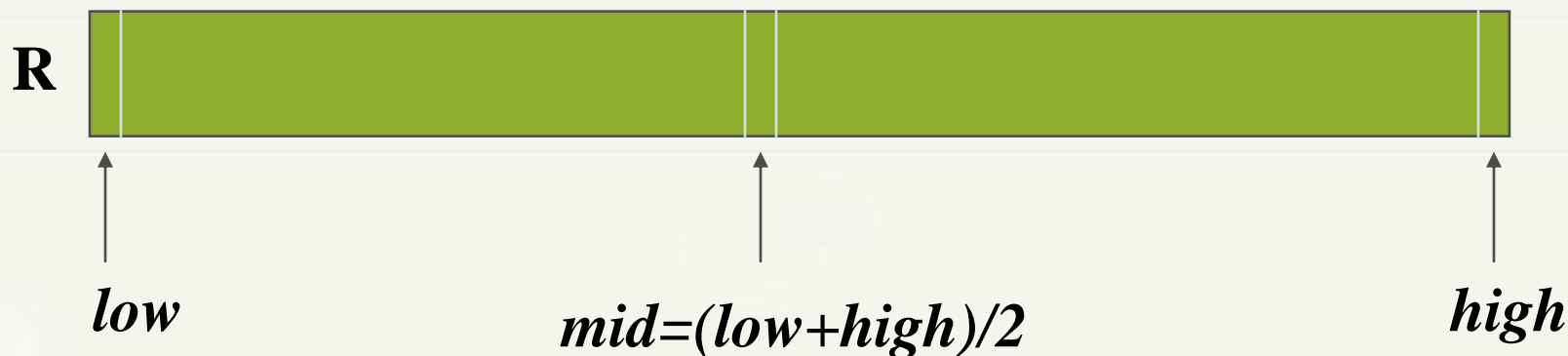
- (1) 查找概率不相等时应该如何提高查找效率？
- (2) 表有序，可以提高顺序查找效率吗？

7.2 静态表的查找

7.2.2 静态查找表是顺序存储的、有序的线性表 ——折半查找（Fibonacci查找、插值查找）

1. 查找表的要求：顺序存储、有序的线性表

2. 查找方法：



$x = R[mid]$	OK!!
$x < R[mid]$	$low \sim mid - 1$
$x > R[mid]$	$mid + 1 \sim high$

7.2 静态表的查找

7.2.2 静态查找表是顺序存储的、有序的线性表 ——折半查找（Fibonacci查找、插值查找）

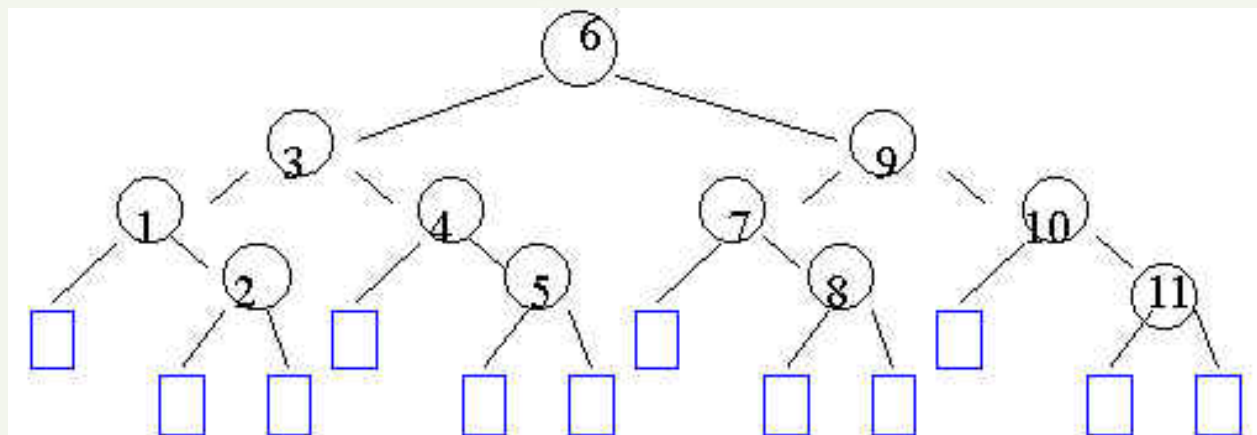
3. 特点：

对查找表要求多；

比较次数较少 $O(\log_2 n)$ ；

折半查找的过程可以用一棵二叉树表示，称之为“折半查找的判定树”。

例如：折半查找在 $n=11$ 时的判定树如下：



7.2 静态表的查找

7.2.3 静态查找表是二叉树

——静态二叉排序树查找

1. 查找表的要求：二叉排序树

2. 查找方法：

X与根比较：

相等： OK！

X<根： 在左子树上找

X>根： 在右子树上找

3. 特点：

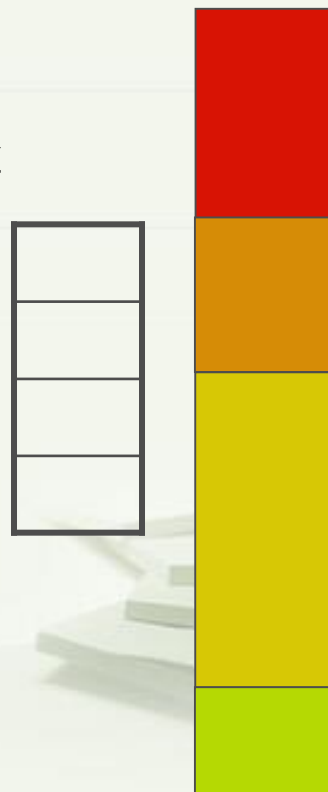
类似折半，比较次数最大是树的深度；
等概率时，深度为 $\log_2 n$ 二叉排序树最好；
不等概率时，概率高的应该靠近根。

7.2 静态表的查找

7.2.4 静态查找表是分块索引表

——分块查找(索引顺序查找)

1. 查找表的要求：顺序存储、分块有序；即：
每一块中的结点不必有序，但块与块之间必须"按块有序"；
2. 查找方法：
 - 先选取各块中的最大关键字构成一个索引表
 - 折半方法确定被查找元素可能所在的块；
 - 在块中采用顺序查找，确定元素是否存在；
3. 特点：
 - 要建立索引表；
 - 效率介于折半和顺序之间；



7.3 动态表的查找

查找表本身是在查找过程中动态生成的，即对于给定值key，若查找表中存在其关键码等于key的元素，则查找成功返回，否则插入关键码等于key的元素。

动态查找表可以是线性表，也可以是树（二叉排序树）。最常用的动态查找表是：二叉排序树。

查找方法：

若二叉排序树为空，则查找不成功，在二叉排序树中插入该元素；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。

7.3 动态表的查找

特点:

查找效率与构造出的二叉排序树的深度有关;

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:

$$ASL = \sum_{i=1}^n p_i \times c_i$$

其中:

p_i 是查找第*i*个元素的概率, 通常假设为等概率;

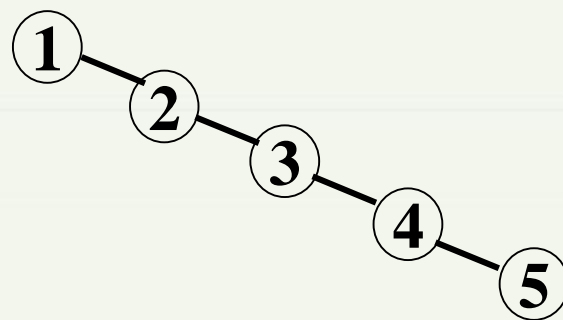
c_i 是查找第*i*个元素的比较次数, 这里 $c_i = h_i$

7.3 动态表的查找

显然，由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：

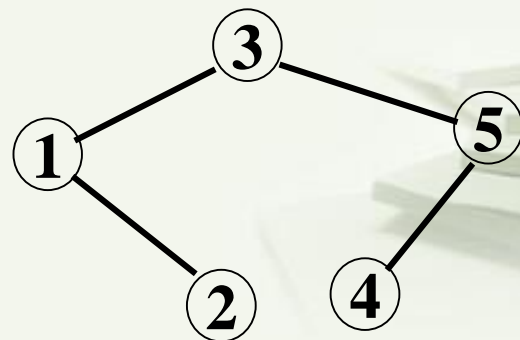
由关键字序列1, 2, 3, 4, 5
构造而得的二叉排序树：

$$ASL = (1+2+3+4+5) / 5 = 3$$



由关键字序列3, 1, 2, 5, 4
构造而得的二叉排序树：

$$ASL = (1+2+3+2+3) / 5 = 2.2$$

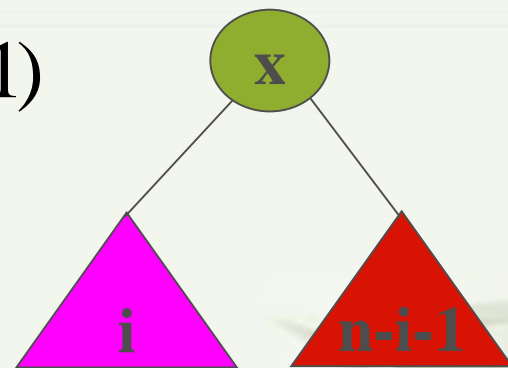


7.3 动态表的查找

通过前面的学习知道：对于同一个关键码集合，因为关键码插入的顺序不同，可以得到不同的二叉排序树。

对于有 n 个关键码的集合，其关键码共有 $n!$ 种不同的排列。那么可以构成多少棵不同的二叉排序树？

$$SP(n) = \sum_{i=0}^{n-1} SP(i) \times SP(n-i-1)$$
$$SP(n) = \frac{1}{n+1} C_{2n}^n$$



所有这些二叉排序树中，最高 n ，最低 $\log_2 n$ 。

对于一个数据集合，如何构造一棵查找效率高的二叉排序树？

7.3 动态表的查找

7.3.1 平衡二叉树

■ 平衡二叉树的定义

1. 定义：一棵二叉排序树是平衡的，当且仅当每个结点的左右子树的高度至多相差为1。由G.M.Adelson_Velskii和E.M.Landis给出的定义——AVL树。

递归定义：

（1）空树是二叉排序树；

（2）它的左右子树都是二叉排序树，且左右子树的高度最多相差为1；

2. 平衡因子：

左子树的高度—右子树的高度，即

$$BF(t) = H_l - H_r$$

平衡二叉树中，对任意结点：BF=1、0、-1

7.3 动态表的查找

7.3.1 平衡二叉树

■ 平衡二叉树的定义

3. 平衡二叉树的特点：

其深度和 $\log_2 n$ 同数量级，即 A V L 树的平均查找长度为 $O(\log_2 n)$ ；

4. AVL树的构造和调整过程：

（1）基本原则：

按照二叉排序树的构造方法，构造过程中判断是否为平衡二叉树（平衡因子），是，则继续构造；否则，按一定的原则（保持是二叉排序树和平衡）将其调整为平衡，然后继续。

7.3 动态表的查找

7.3.1 平衡二叉树

■ 平衡二叉树的定义

(2) 插入过程中的调整原则:

二叉排序树在插入前平衡, 插入一个结点后如果失去平衡, 则至少有一个结点的平衡因子变为 $+2$ 或 -2 。

若平衡因子 $=+2$, 则左分支高于右分支;

若平衡因子 $=-2$, 则右分支高于左分支;

分为4种情况, 分别进行调整:

LL型: 在左分支的左子树上插入后, 失去平衡, $BF=2$

LR型: 在左分支的右上子树插入后, 失去平衡, $BF=2$

RR型: 在右分支的右子树上插入后, 失去平衡, $BF=-2$

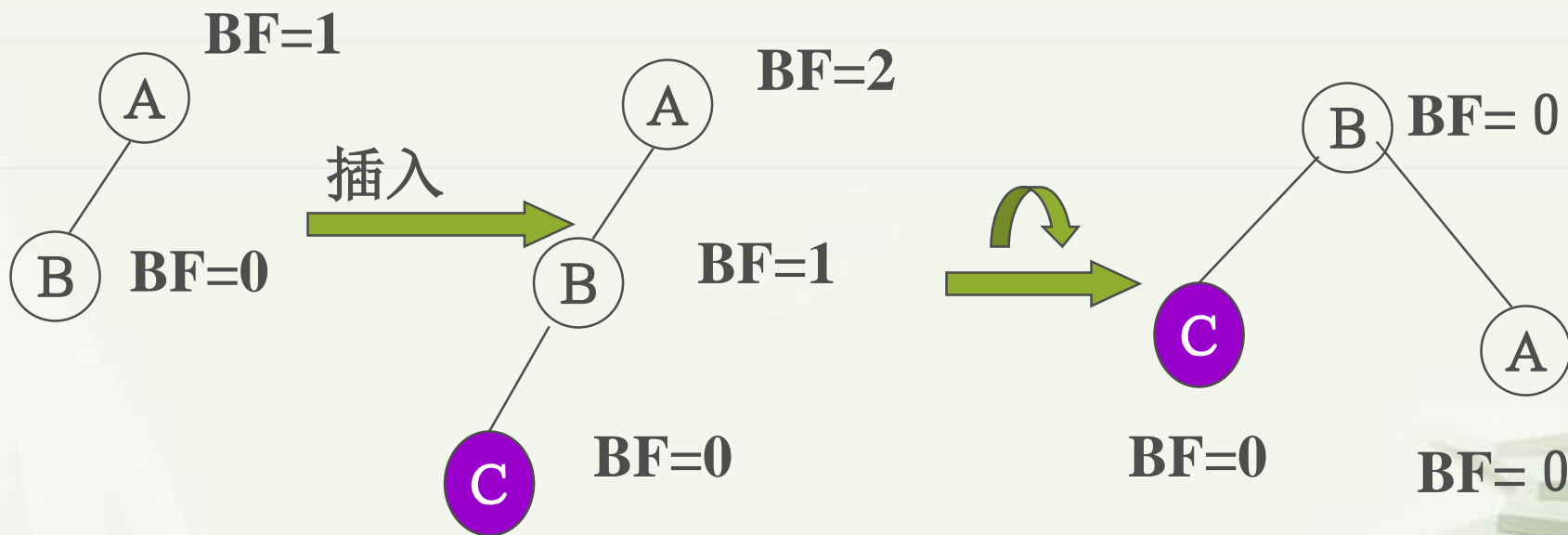
RL型: 在右分支的左子树上插入后, 失去平衡, $BF=-2$

7.3 动态表的查找

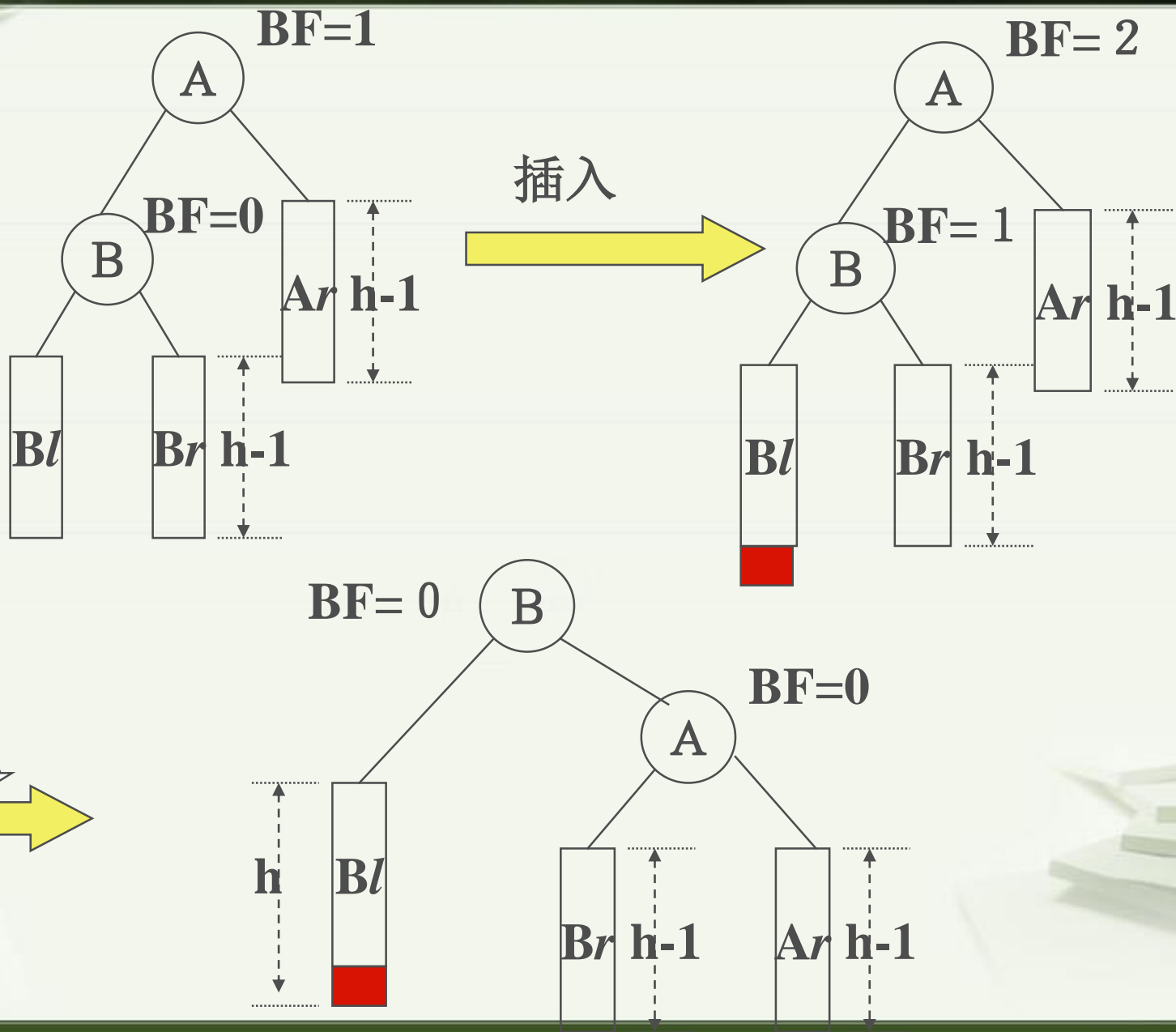
7.3.1 平衡二叉树

LL型:

特殊地:



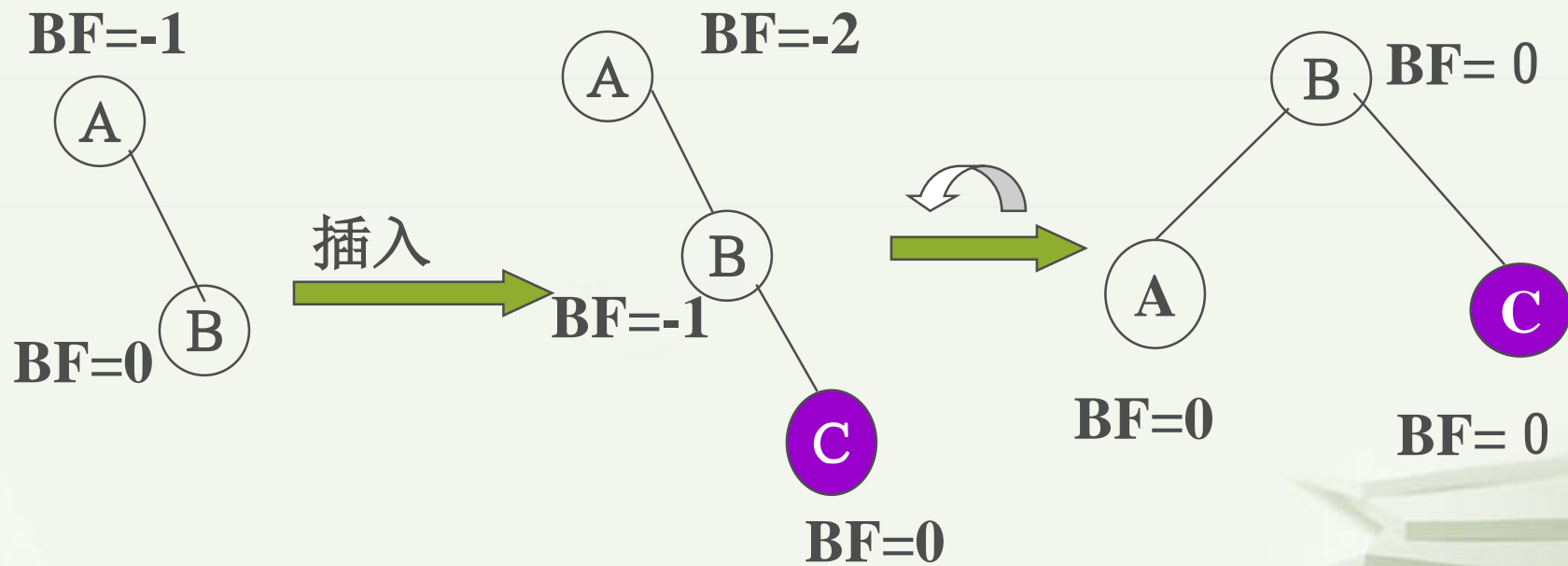
7.3 动态表的查找



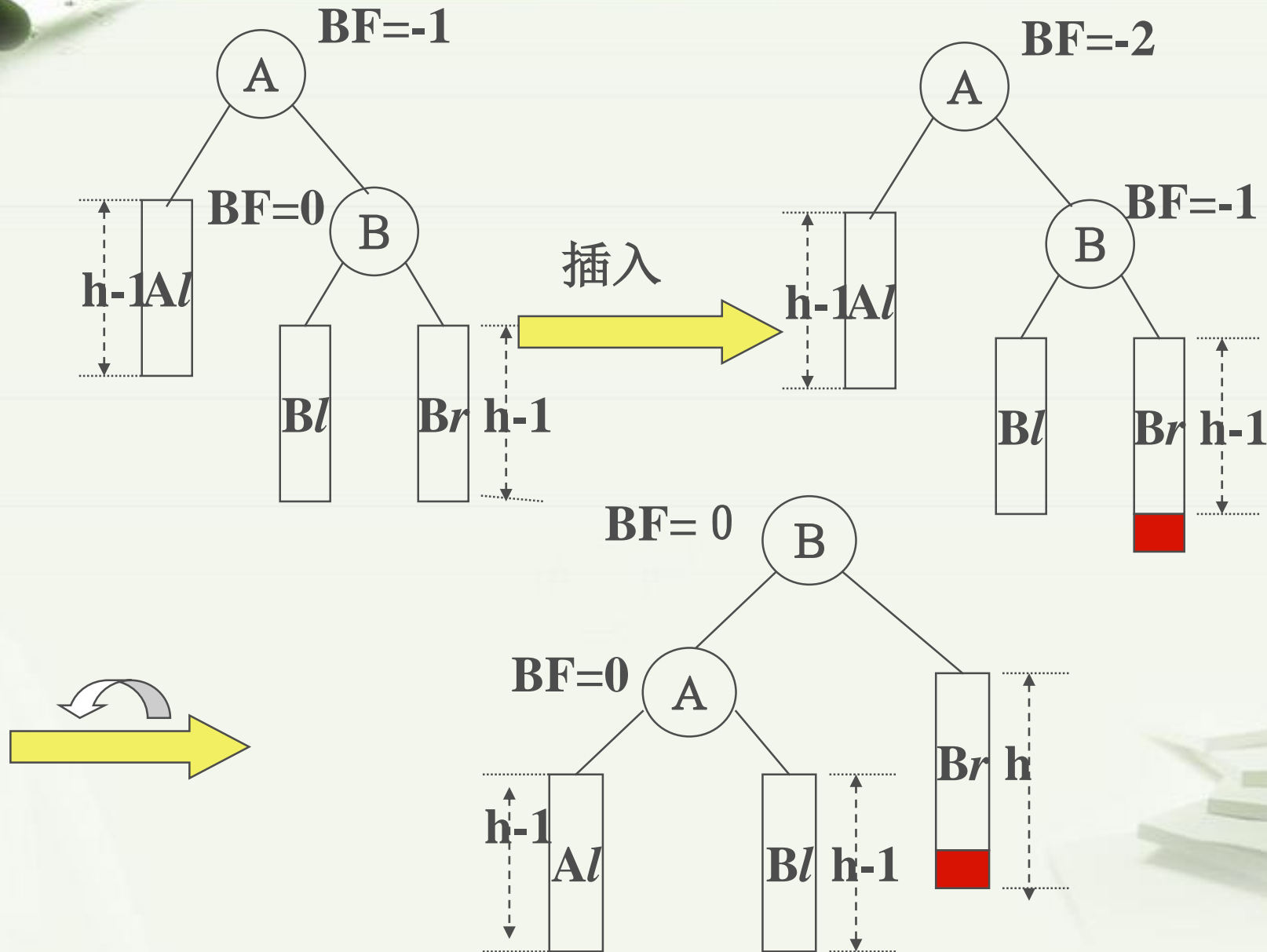
7.3 动态表的查找

RR型:

特殊地:



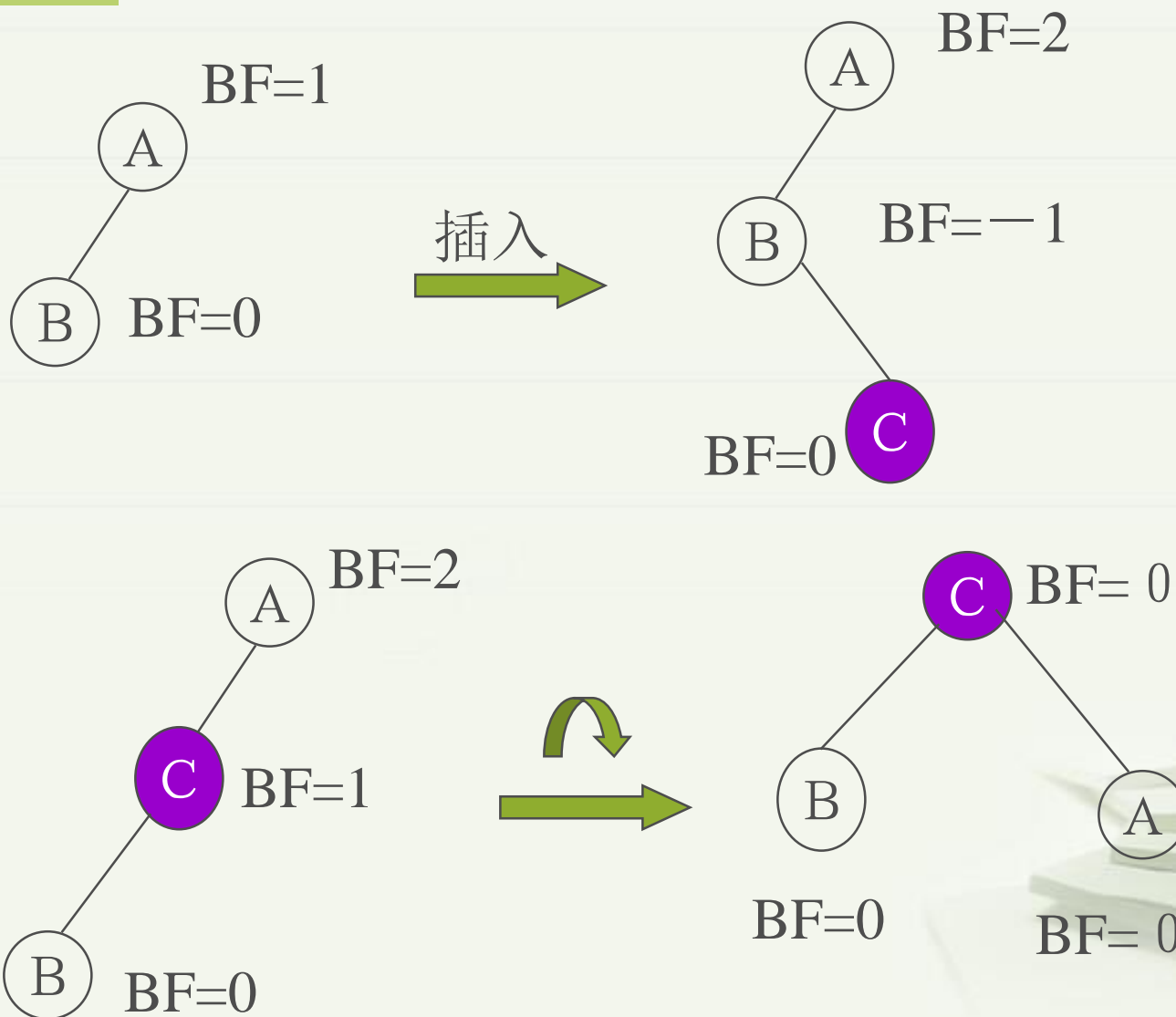
7.3 动态表的查找



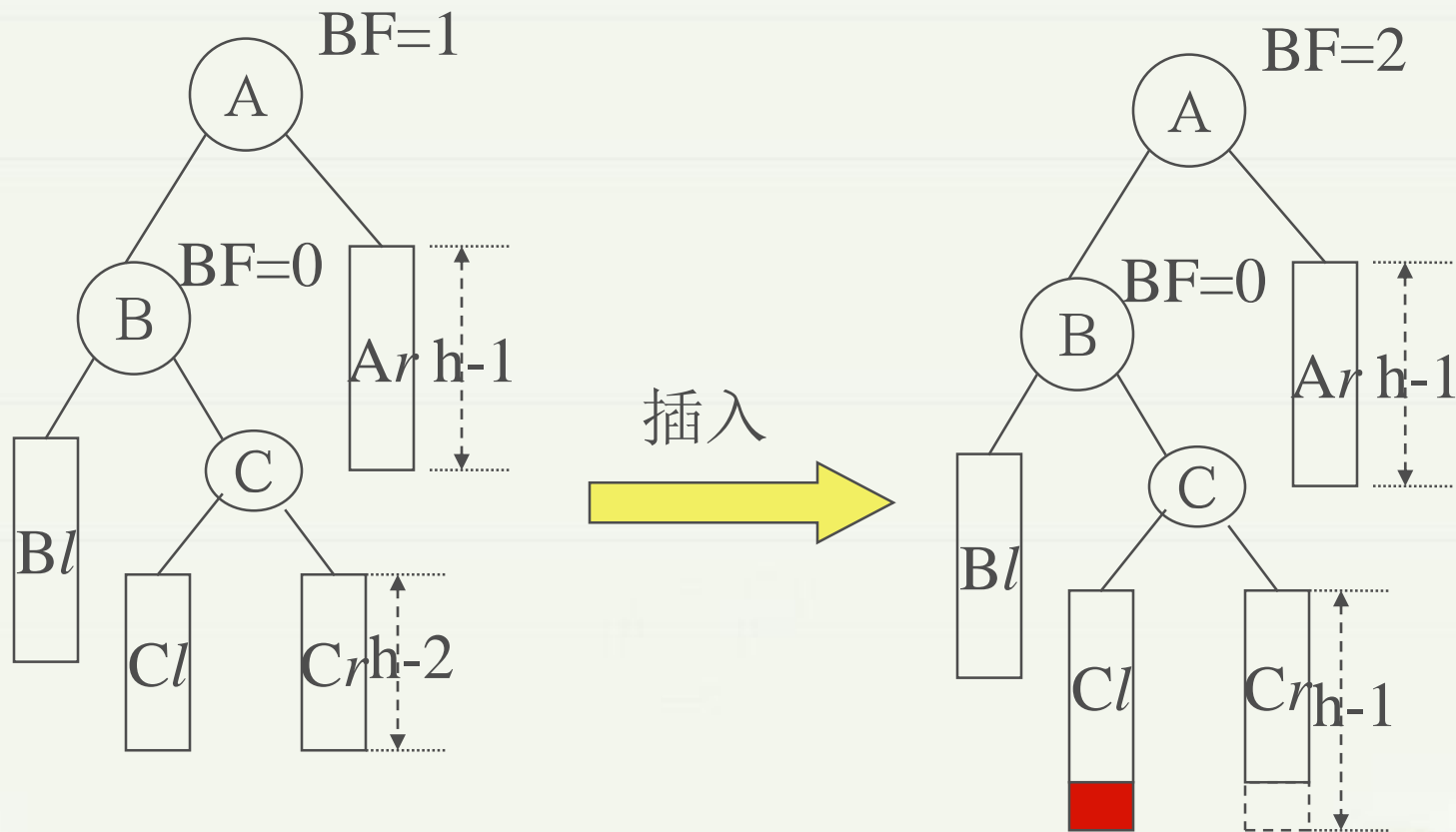
7.3 动态表的查找

L R 型:

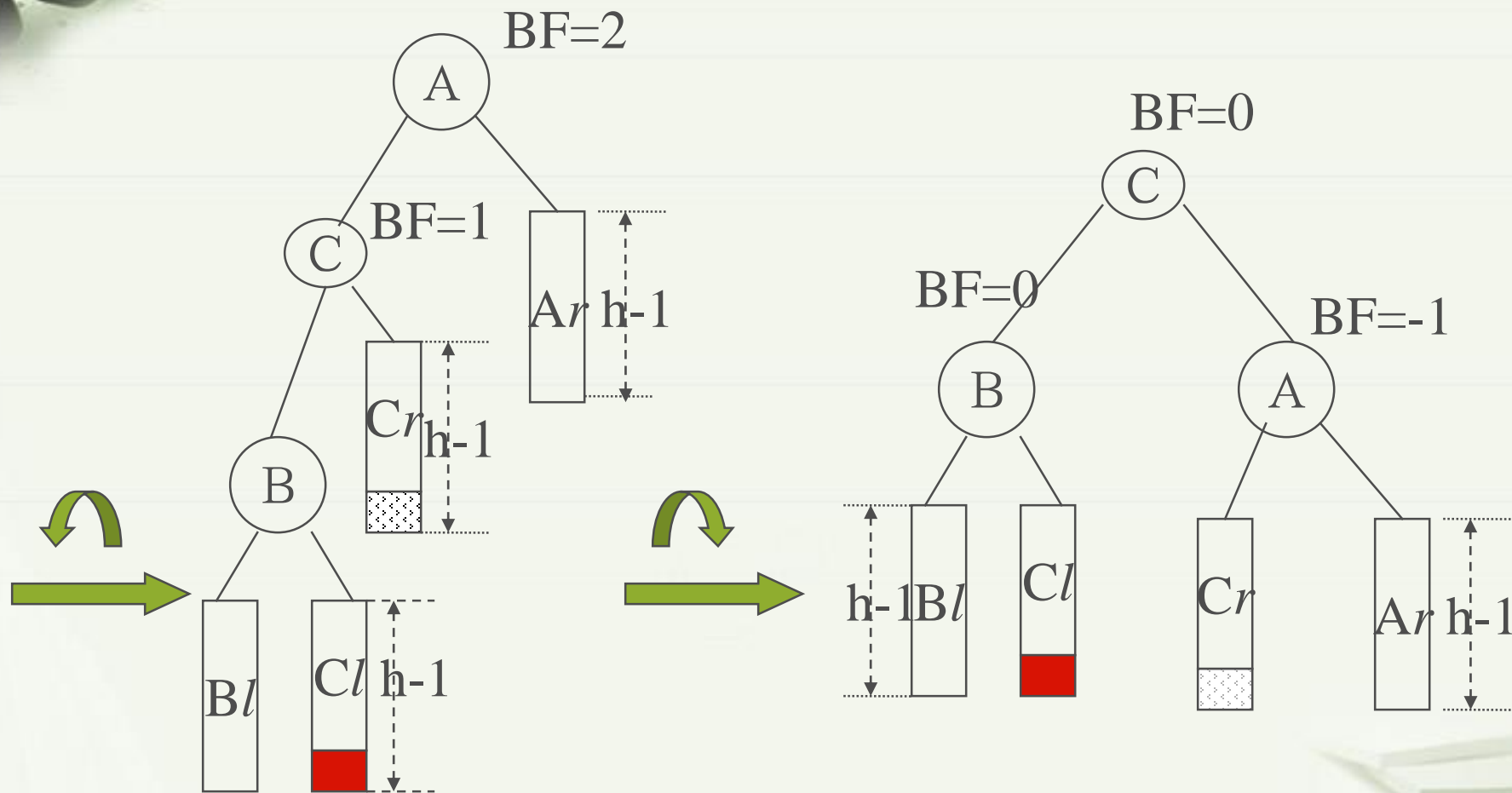
特殊地:



7.3 动态表的查找



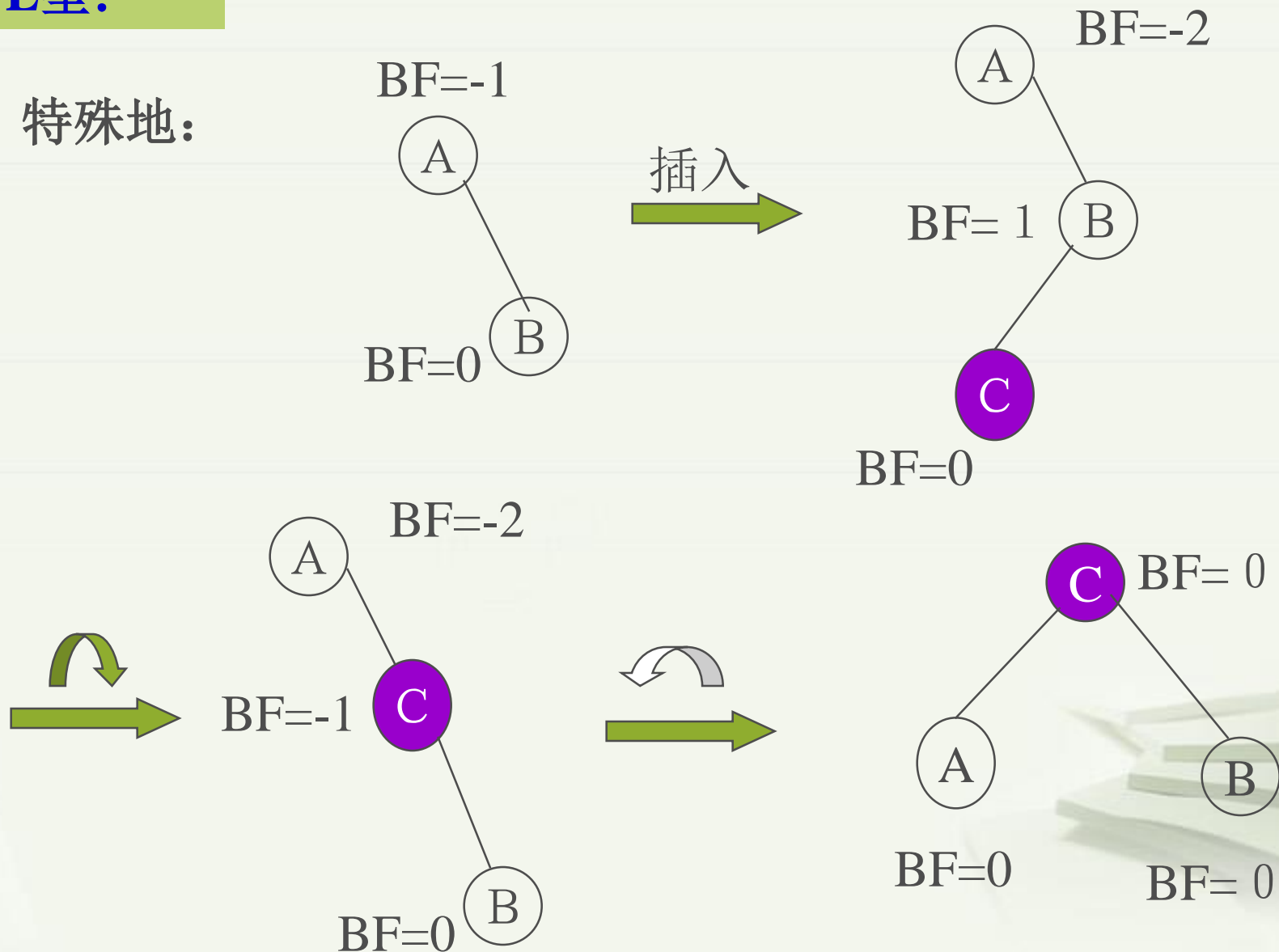
7.3 动态表的查找



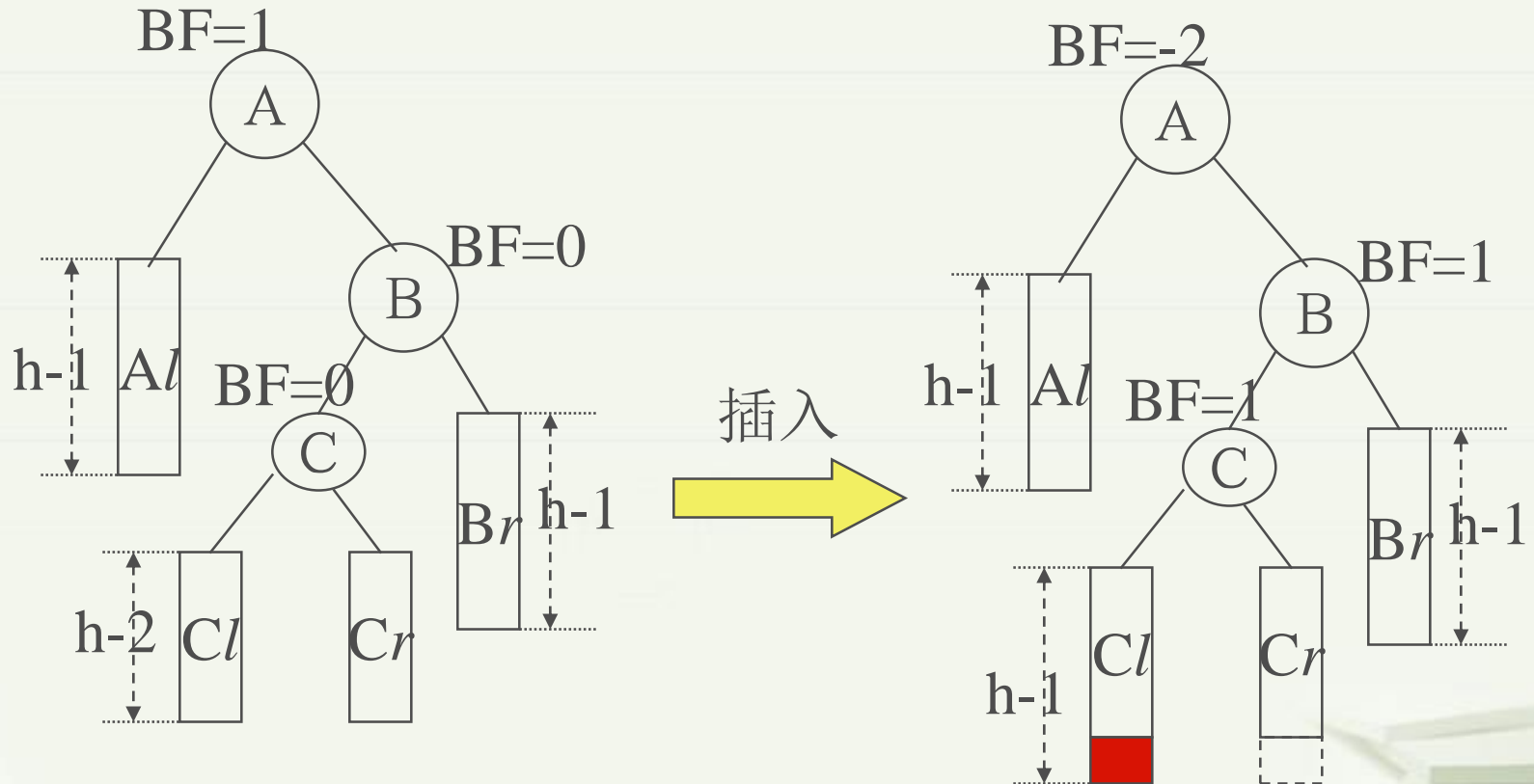
7.3 动态表的查找

RL型:

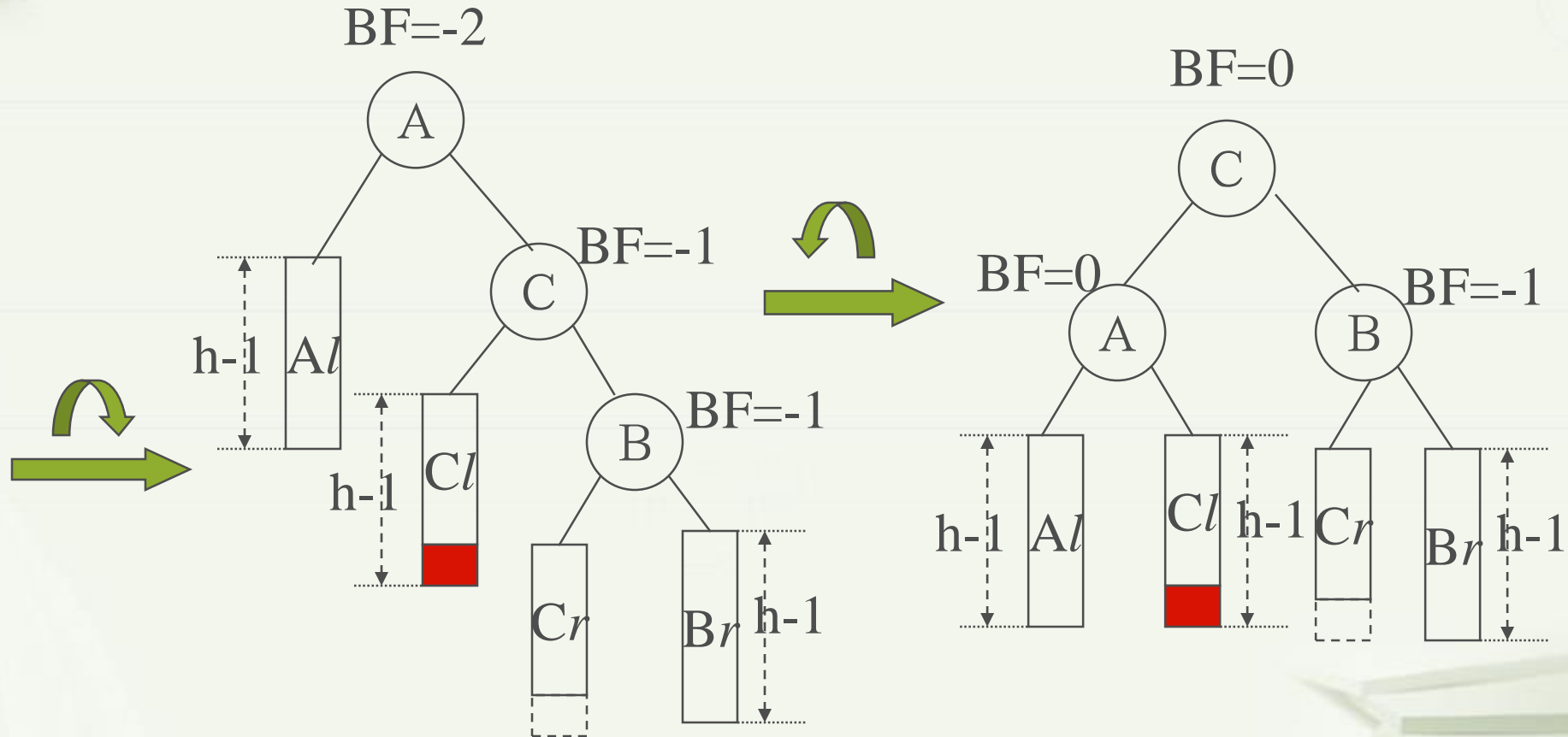
特殊地:



7.3 动态表的查找



7.3 动态表的查找



7.3 动态表的查找

■ 平衡二叉树上的查找

1. 查找过程：同静态二叉排序树一样！

2. 效率分析：

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。

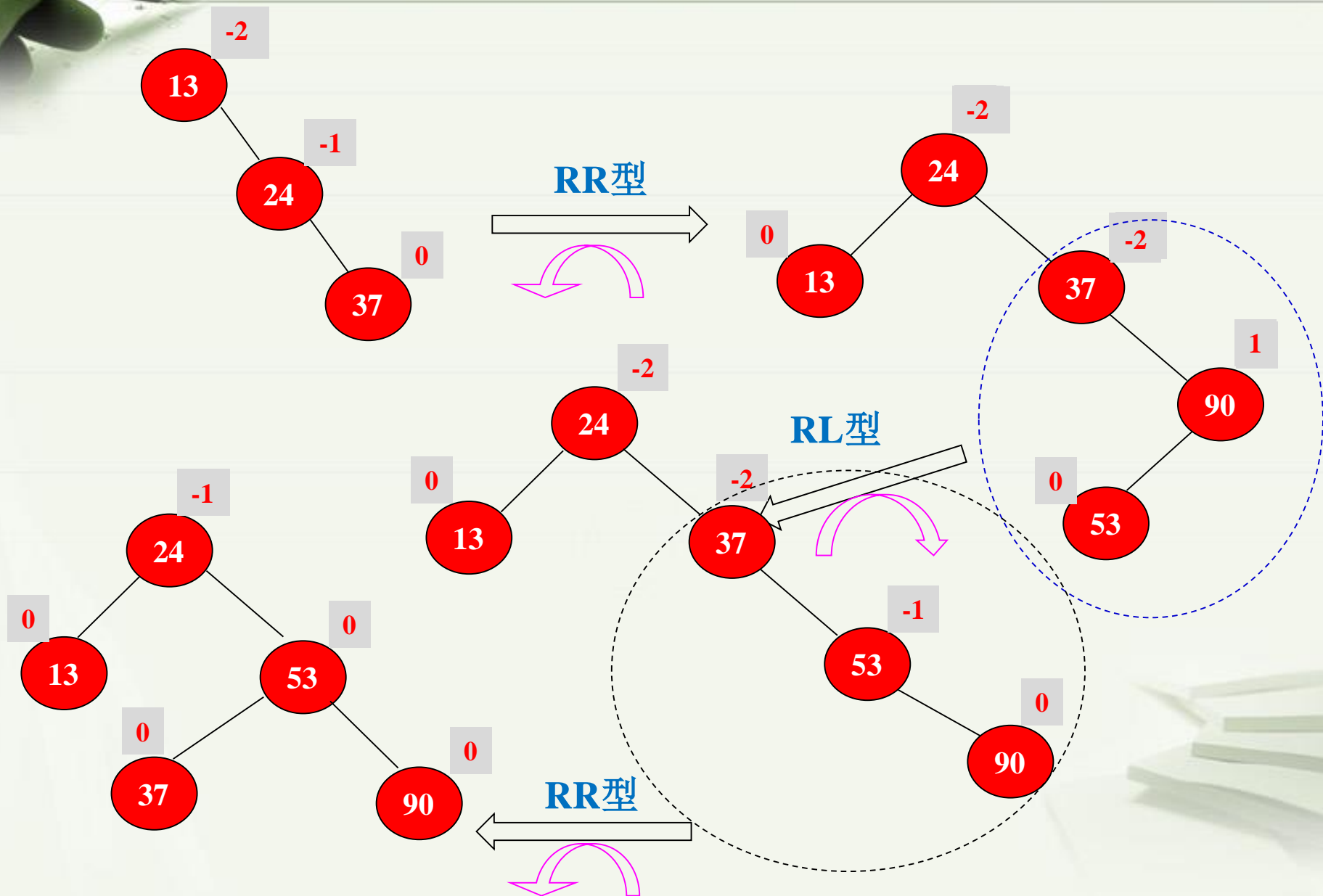
假设深度为 h 的二叉平衡树上所含结点数最小值为 N_h ，则显然 $N_h = N_{h-1} + N_{h-2} + 1$

由此可以推导出： $h \approx \log(n)$

因此，在平衡树上进行查找的时间复杂度为 $O(\log(n))$

举例：有下列元素，构造平衡二叉树
13, 24, 37, 90, 53

7.3 动态表的查找



7.4 高级搜索树

7.4.1 伸展树

通过“旋转”把刚访问的元素调整到根（伸展），使下一次访问效率更高！

- . 逐层展开
- . 双侧展开

7.4.2 B-树

多级存储的组织，各级存储之间速度差异巨大，I/O瓶颈

7.4.3 红黑树

通过定义红黑两种结点，并给出约束条件，控制树的高度，从而达到较好的查找效率。

7.5 HASH 查找

7.5.1 哈希技术概述

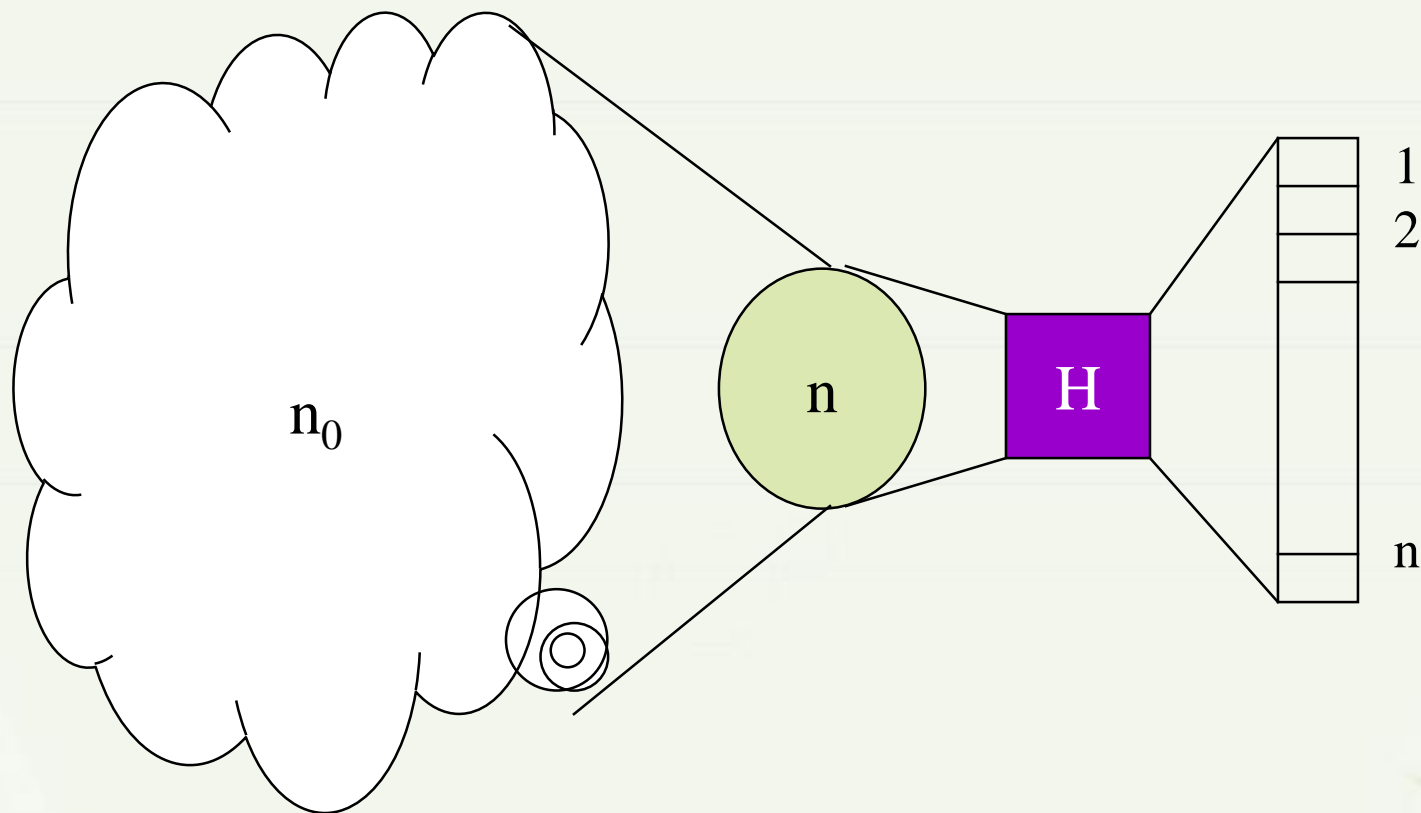
1. 哈希技术的提出背景:

我们前面介绍的各种查找方法，其基本操作是“比较”即通过比较得到元素的位置。那么，有没有不用“比较”的查找方法？

如果有一个函数，它能够根据要查找的关键字直接计算出要查找元素的地址，该地址的内容为空，则查找的元素不存在，否则，查找成功。显然，这样的查找不需要比较！

7.5 HASH 查找

7.5.1 哈希技术概述



7.5 HASH 查找

7.5.1 哈希技术概述

2. HASH类问题的描述:

假设问题可能用到的关键字集合为 U , $|U|=n_0$, 即该集合的元素个数为 n_0 , 而一个问题实际用到的关键字集合为 S , $|S|=n$, $n \ll n_0$, 且 S 是取自 U 的任意一个子集合, 表示为 $\{R_1, R_2, R_3, \dots, R_n\}$, 其关键字集合为 $\{K_1, K_2, \dots, K_n\}$; T 是解决问题需要的连续空间, 它由 m 个存储单元组成, 表示为 $T[0..m-1]$, $m \geq n$ 。

有一个函数 H , 其定义域是 $key \in U$, 值域是 $i \in 0..m-1$, 它将关键字映射到存储空间地址上;

$$H(key)=i \quad key \in U, i \in 0..m-1$$

7.5 HASH 查找

7.5.1 哈希技术概述

3. 有关基本概念：

- (1) **HASH函数**：将元素按照关键字映射出存储空间地址的函数，记作： $H(\text{key})$
- (2) **HASH地址**：由HASH函数计算出的数据元素的存储地址；
- (3) **HASH表**：存储元素的连续地址空间T；
- (4) **HASH造表**：利用HASH函数将元素存储到HASH表中的过程；
- (5) **HASH表的填充度（装填因子）**：

$$\alpha = \frac{\text{表中填入的元素个数}}{\text{HASH表的长度}}$$

7.5 HASH 查找

7.5.1 哈希技术概述

- (6) 冲突：由于HASH函数的定义域是U，而S是U的任意一个小子集，映射地址空间是由S的大小确定的，因此，对于不同的关键字可能得到相同的HASH地址，即：
若 $\text{key1} \neq \text{key2}$ ，而 $H(\text{key1}) = H(\text{key2})$ ，则称为冲突。
- (7) 同义词：若 $H(\text{key1}) = H(\text{key2})$ ，则key1和key2互称为同义词。

4. HASH技术的关键：

- ♠ HASH函数的构造
- ♠ 解决冲突的方法

7.5 HASH 查找

7.5.2 哈希函数的构造方法

函数设计目标:使通过哈希函数得到的 n 个数据元素的哈希地址尽可能均匀地分布在 m 个连续内存单元上，同时使计算过程尽可能简单以达到尽可能高的时间效率。

常用的哈希函数构造方法有：

- 1.除法取余
- 2.直接定址法
- 3.数字分析法
- 4.折叠法
- 5.平方取中

7.5 HASH 查找

7.5.2 哈希函数的构造方法

除留余法:

$H(\text{key}) = \text{key} \text{ MOD } P$ P 是不大于 m 的素数,
 m 是HASH表的长度;

特点: 这是一种最简单、最常用的方法。

- * P 的选择很重要, 选择不好会产生同义词;
- * P 一般取位素数或不包含小于20的质数的合数;

7.5 HASH 查找

7.5.3 冲突的解决方法

解决冲突的策略分为两类：

- (1) 闭散列方法(Closed Hashing):同义词放在HASH表中的其他位置; (Open addressing,又称为开地址法)
- (2) 开散列方法(Open Hashing):同义词放在HASH表之外的空间中; (Separate chaining,又称为拉链法)

1. 开放定址法: (闭散列)

发生冲突后, 按一定的原则寻找新的地址:

$$H_i = (H(\text{key}) + d_i) \text{ MOD } m \quad i=1, 2, \dots, k$$

d_i 为增量, m 为HASH表的长度;

d_i 的取法:

线性探测: $d_i = 1, 2, 3, \dots, m-1$

二次探测: $d_i = 1^2, -1^2, 2^2, -2^2, \dots, \pm k^2$

7.5 HASH 查找

7.5.3 冲突的解决方法

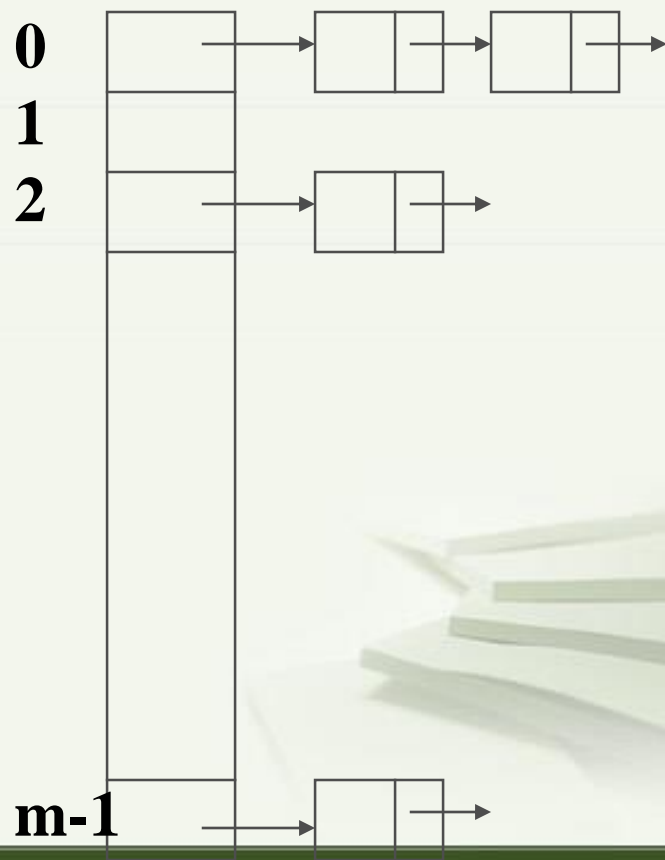
2. 再造HASH法：（闭散列）

$$H_i = RH_i(\text{key})$$

用一系列HASH函数计算地址

3. 链地址法：（开散列）

将同义词存放在同一个链表中；

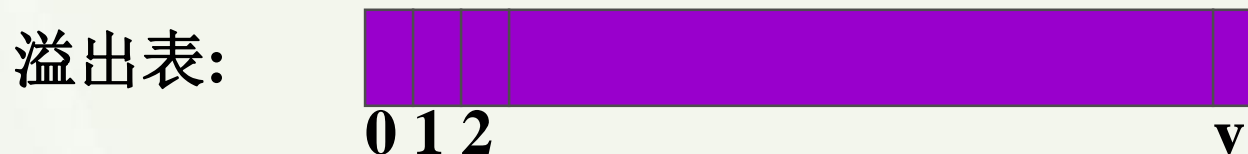
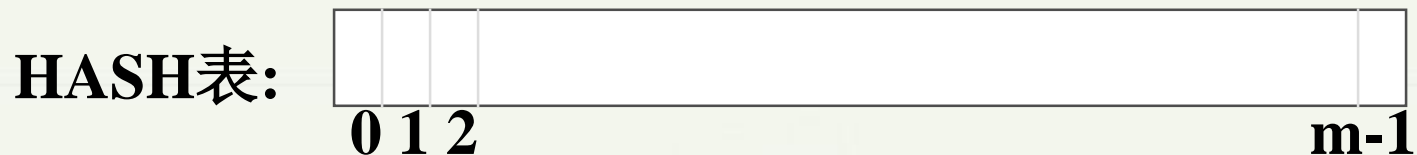


7.5 HASH 查找

7.5.3 冲突的解决方法

4. 建立公共溢出区法:

除了HASH表外，开辟一个公共溢出区，一旦冲突，将同义词放入公共溢出区；



7.5 HASH 查找

7.5.4 哈希查找

1. 哈希查找的方法

给定关键字 k :

- (1) 用给定的HASH函数计算出 k 的HASH地址; $i=H(\text{key})$
- (2) 若该地址为空, 则查找失败;
否则, 若 $T[i]=k$, 则查找成功, 返回地址;
否则, 按HASH造表时解决冲突的方法计算出新的地址;
- (3) 重复 (2) 直到查找成功或失败。

2. 性能分析

若没有冲突: $O(1)$;

有冲突, 性能与下列因素有关:

- HASH函数
- 解决冲突的方法
- 装填因子

7.5 HASH 查找

7.5.4 哈希查找

已知有12个元素：19,14,23,01,68,20,84,27,55,11,10,79

HASH函数为： $H(\text{key}) = \text{key} \text{ MOD } 13$ ，HASH表长度为16，构造HASH表，进行HASH查找。

(1) 线性探测法解决冲突 $H_i = (H(\text{key}) + d_i) \text{ MOD } 15$
 $d_i = 1, 2, 3, \dots, m-1$

关键字： 19 14 23 01 68 20 84 27 55 11 10 79

地 址： 6 1 10 1 3 7 6 1 3 11 10 1

HASH表：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			
	79	79	79	79	79	79	79	79		10	10				

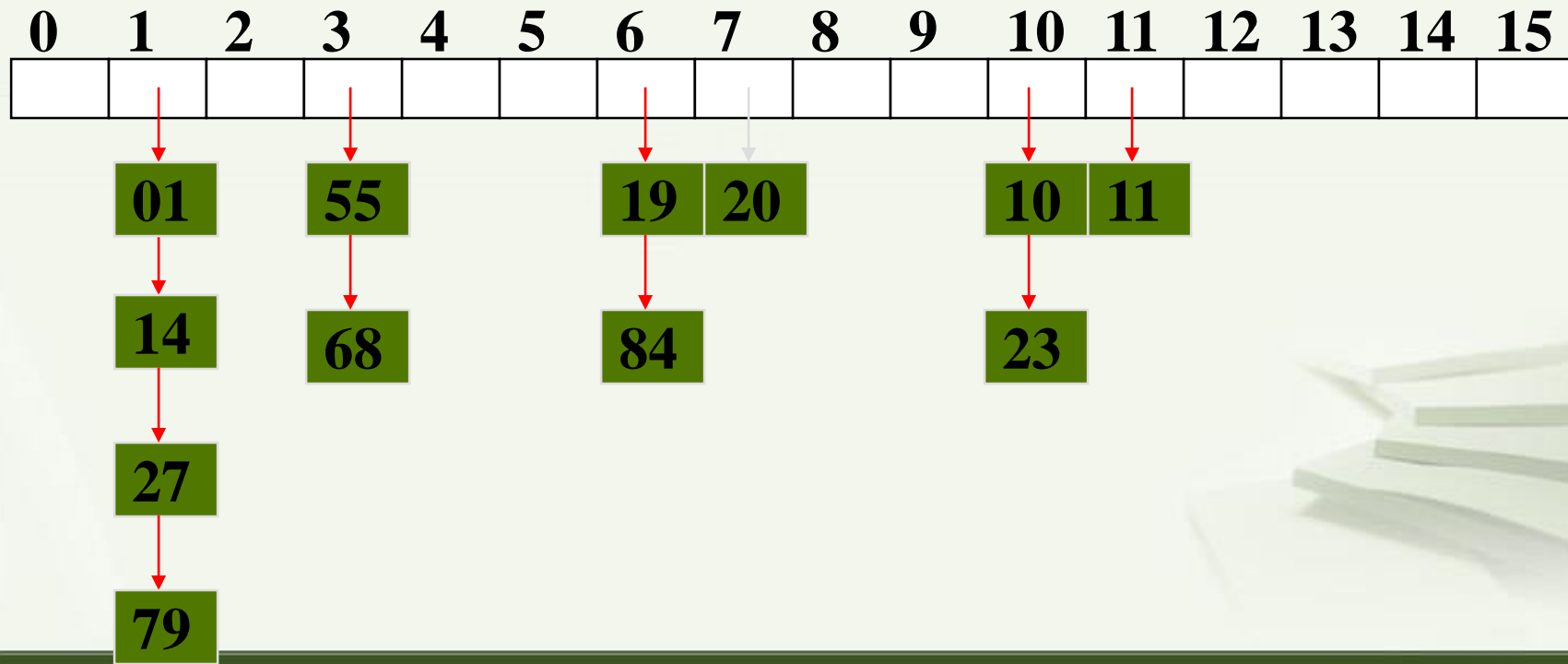
7.5 HASH 查找

(2) 链地址法解决冲突

关键字: 19 14 23 01 68 20 84 27 55 11 10 79

地 址: 6 1 10 1 3 7 6 1 3 11 10 1

HASH表:





7.5 HASH 查找

7.5.4 哈希查找

查找元素 27： 需要比较多少次？

查找元素 9： 需要比较多少次？





END



The word "END" is rendered in a large, bold, 3D font with a yellow-to-orange gradient. It is positioned in the center of a white, slightly textured surface. In the top left corner, a hand is visible holding a pen, appearing to have just finished writing. In the bottom right corner, there is a stack of papers or documents, slightly out of focus.