第九章　查　找

王　勇
计算机/软件学院　大数据分析与信息安全团队
21#518　电　话 13604889411
Email：　wangyongcs@hrbeu.edu.cn

哈尔滨工程大学
Harbin Engineering University

本章内容

1. 基本概念与术语
2. 静态查找表
3. 动态查找表
4. 哈希表
5. 本章小结

本章说明

- 学习目标
  - 理解“查找表”的结构特点以及各种表示方法的适用范围；
  - 熟练掌握以顺序表或有序表表示静态查找表时的查找方法；
  - 掌握描述查找过程的判定树的构造方法，以及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
  - 熟练掌握二叉排序树的构造和查找方法；
  - 理解掌握二叉平衡树的构造过程；

本章说明

- 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；

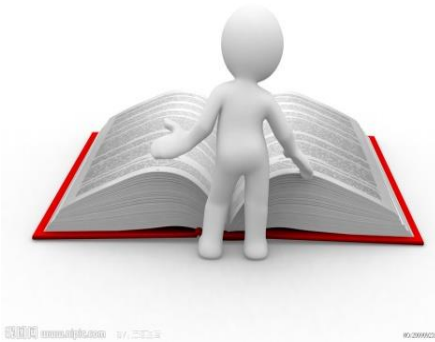
◆ **重点和难点**

重点在于理解查找表的概念及其各种表示方法的特点和适用场合。

◆ **知识点**

顺序表、有序表、索引顺序表、判定树、二叉排序树、二叉平衡树、哈希表

本章内容



1. 基本概念与术语
2. 静态查找表
3. 动态查找表
4. 哈希表
5. 本章小结

基本概念

- 查找——也叫检索，是根据给定的某个值，在表中确定一个关键字等于给定值的记录或数据元素
- 关键字——是数据元素中某个数据项的值，它可以标识一个数据元素或记录
- 查找方法评价
  - 查找速度
  - 占用存储空间多少
  - 算法本身复杂程度
  - 平均查找长度ASL(Average Search Length)：为确定记录在表中的位置，需和给定值进行比较的关键字的个数的期望值;等概率条件下，总体考虑比较次数;

对含有n个数据元素的表，$ASL = \sum_{i=1}^{n} p_i c_i$

其中：p_i为查找表中第i个元素的概率，$\sum_{i=1}^{n} p_i = 1$

c_i为找到表中第i个元素所需比较次数

- 查找表——是由同一类型的数据元素(或记录)构成的集合，是查找操作的对象。

由于“集合”中的数据元素之间仅仅存在着松散的关系，因此查找表是一种应用灵便的数据结构。

- 对查找表经常进行的操作：
  - 查询某个“特定的”数据元素是否在查找表中；
  - 检索某个“特定的”数据元素的各种属性；
  - 在查找表中插入一个数据元素；
  - 从查找表中删去某个数据元素。

基本概念

- 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

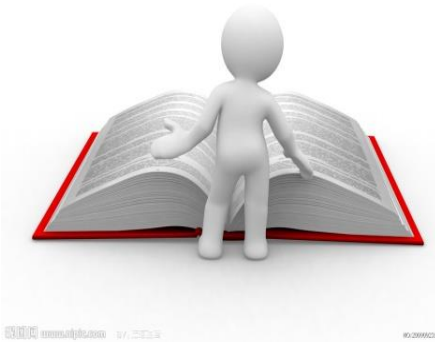
如何进行查找？

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， 可能用另外一种结构来表示查找表。

本章内容



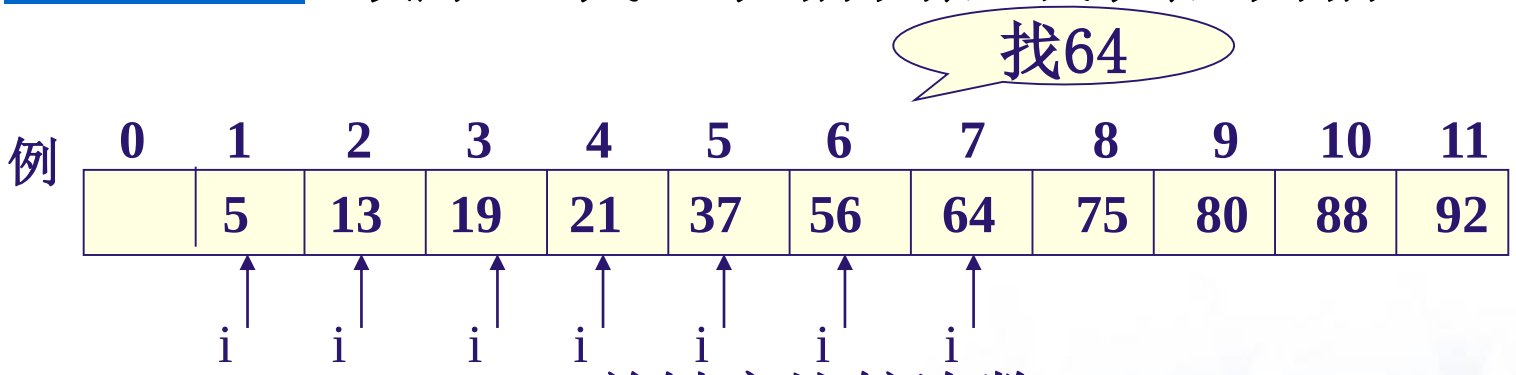
1. 基本概念与术语
2. 静态查找表
3. 动态查找表
4. 哈希表
5. 本章小结

静态表的查找

- 顺序查找
- 二分查找
- 分块查找
- 方法比较

静态表的查找

- 查找过程：从表的一端开始逐个进行记录的关键字和给定值的比较
- 算法描述 顺序查找，从前向后或从后向前



关键字比较次数：

查找第n个元素： n

查找第n-1个元素： n-1

……

查找第1个元素： 1

查找第i个元素： i

查找失败： n+1

比较次数=7×2

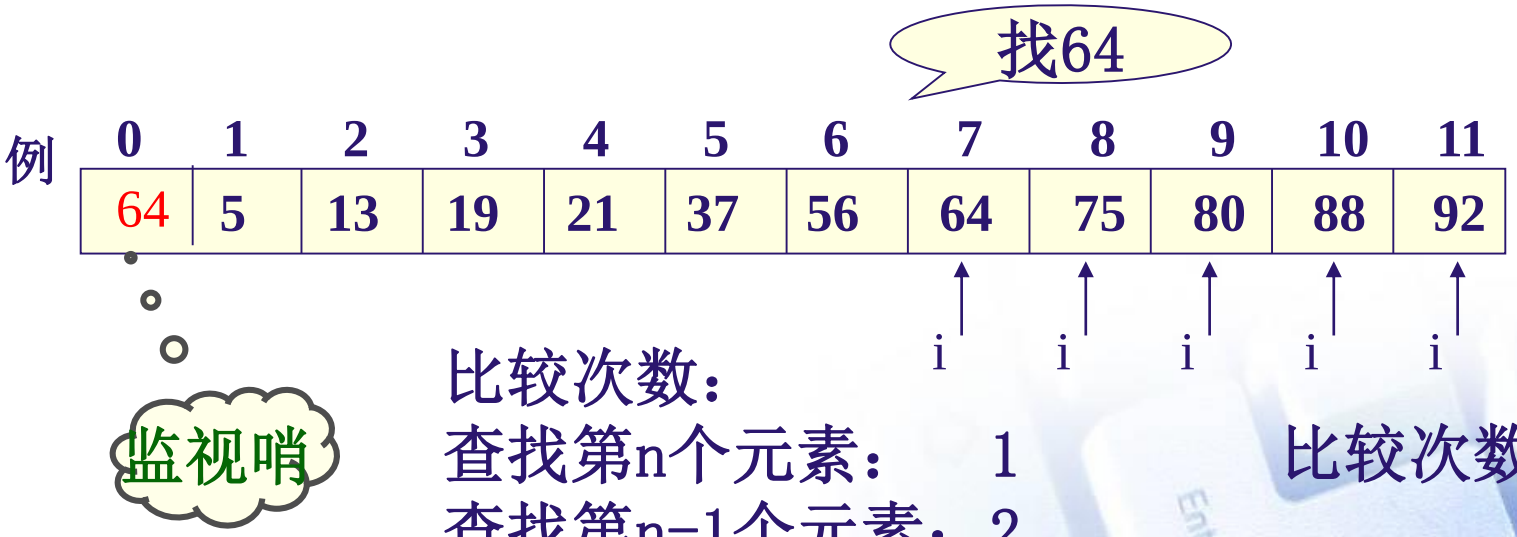
静态表的查找

```
typedef struct //-----静态查找表的顺序存储结构-----
{    ElemType *elem;
     int          length;
}SSTable;//静态查找表

int Search_Seq(SSTable ST, KeyType key)  //从前向后找
{   //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
    for ( i=1;!EQ(ST.elem[i].key, key)&&i<=ST.length;  ++i);      //从前向后找
    if i>ST.length
        return 0;
    else
        return i;
}//Search_Seq                                                    //找不到时，i为0

int Search_Seq(SSTable ST, KeyType key)  //从后向前找
{   //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
    for ( i=ST.length;!EQ(ST.elem[i].key, key)&&i>0; --i);      //从前向后找
    return i;
}//Search_Seq          //找不到时，i为0
```

静态表的查找

- 查找过程：从表的一端开始逐个进行记录的关键字和给定值的比较
- 算法描述

sf9.1



比较次数：

查找第n个元素： 1

查找第n-1个元素： 2

……

查找第1个元素： n

查找第i个元素： n+1-i

查找失败： n+1

比较次数=5

静态表的查找

```
/-----静态查找表的顺序存储结构-----
typedef struct
{    ElemType *elem;
      int          length;
}SSTable;

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;
    for ( i=ST.length; !EQ(ST.elem[i].key, key); --i);      //从后向前找
    return i;
}//Search_Seq                                                  //找不到时，i为0
```

- 顺序查找方法的ASL

对含有n个记录的表，$ASL = \sum_{i=1}^{n} p_i c_i$

设表中每个元素的查找概率相等$p_i = \frac{1}{n}$

则$ASL = \sum_{i=1}^{n} p_i c_i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$

静态表的查找

- 查找思想：每次将待查记录所在区间缩小一半
- 适用条件：采用顺序存储结构的有序表
- 算法描述
  - 设表长为n，low、high和mid分别指向待查元素所在区间的上界、下界和中点，key为给定值
  - 初始时，令low=1，high=n，mid=⌊(low+high)/2⌋
  - 让key与mid指向的记录比较
    - 若key= =r[mid].key，查找成功
    - 若key<r[mid].key，则high=mid-1
    - 若key>r[mid].key，则low=mid+1
  - 重复上述操作，直至low>high时，查找失败

- 算法实现


sf9.2

例


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

静态表的查找


找70
例
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
high
mid

静态表的查找

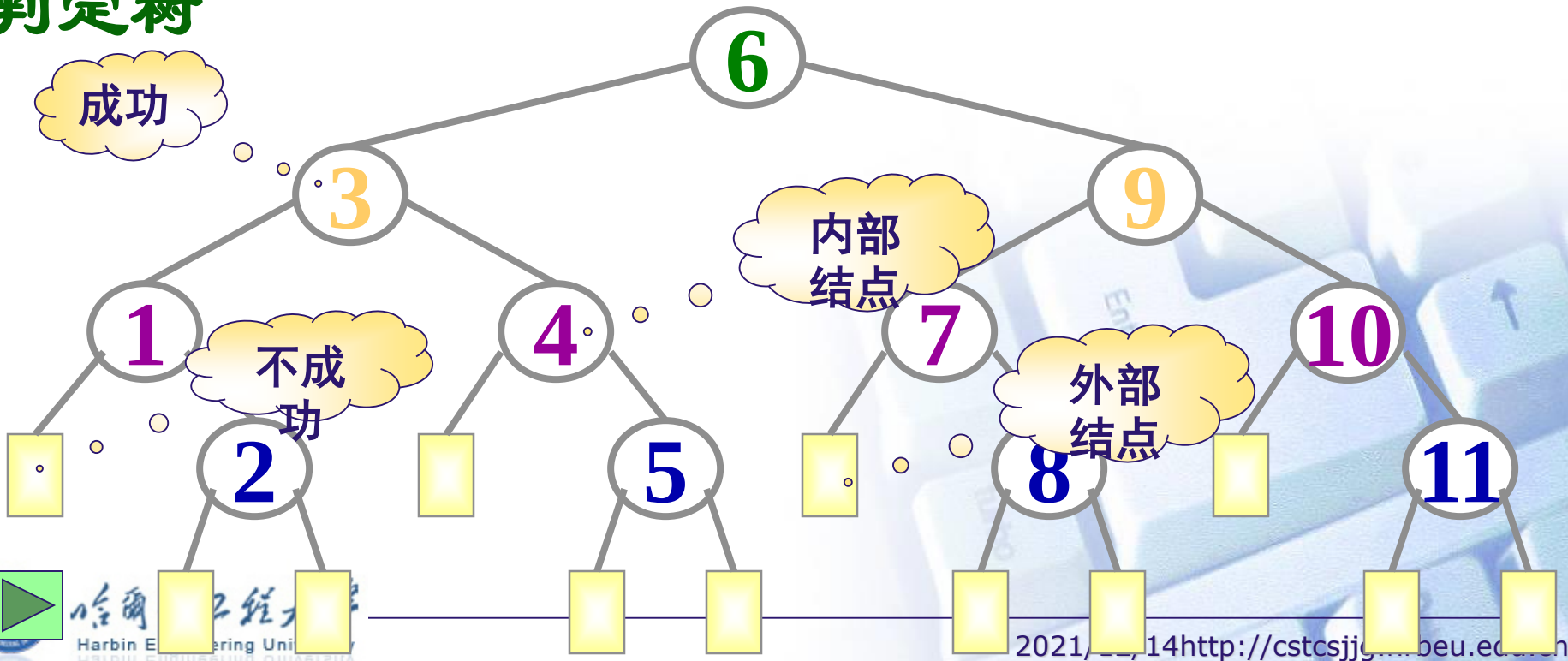
```
int Search_Bin(SSTable ST, KeyType key)
{   //在有序表ST中折半查找其关键字等于key的数据元素。若找到，则函    //数值为该元素在表中的位置，否则为0
  low=1; high=ST.length;          //置区间初值
  while (low<=high)
      { mid=(low+high)/2;
        if (EQ(key, ST.elem[mid].key)) return mid;  //找到待查元素
        else if ( LT(key, ST.elem[mid].key)) high=mid-1;//在前半区查找
             else low=mid+1;                                  //在前后区查找
      }
    return 0;   //查找成功返回mid，不成功为0
}//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



- 算法评价
  - 判定树：描述查找过程的二叉树
  - 有n个结点的判定树的深度为$\lfloor \log_2 n \rfloor + 1$
  - 折半查找法在查找过程中进行的比较次数最多不超过其判定树的深度
  - 折半查找的ASL

设表长$n = 2^h - 1, h = \log_2(n+1)$,即判定树是深度为$h$的满二叉树

设表中每个记录的查找概率相等$p_i = \frac{1}{n}$

则：$ASL = \sum_{i=1}^{n} p_i c_i = \frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1}$

$$= \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

◆分块（索引顺序表）查找的思想

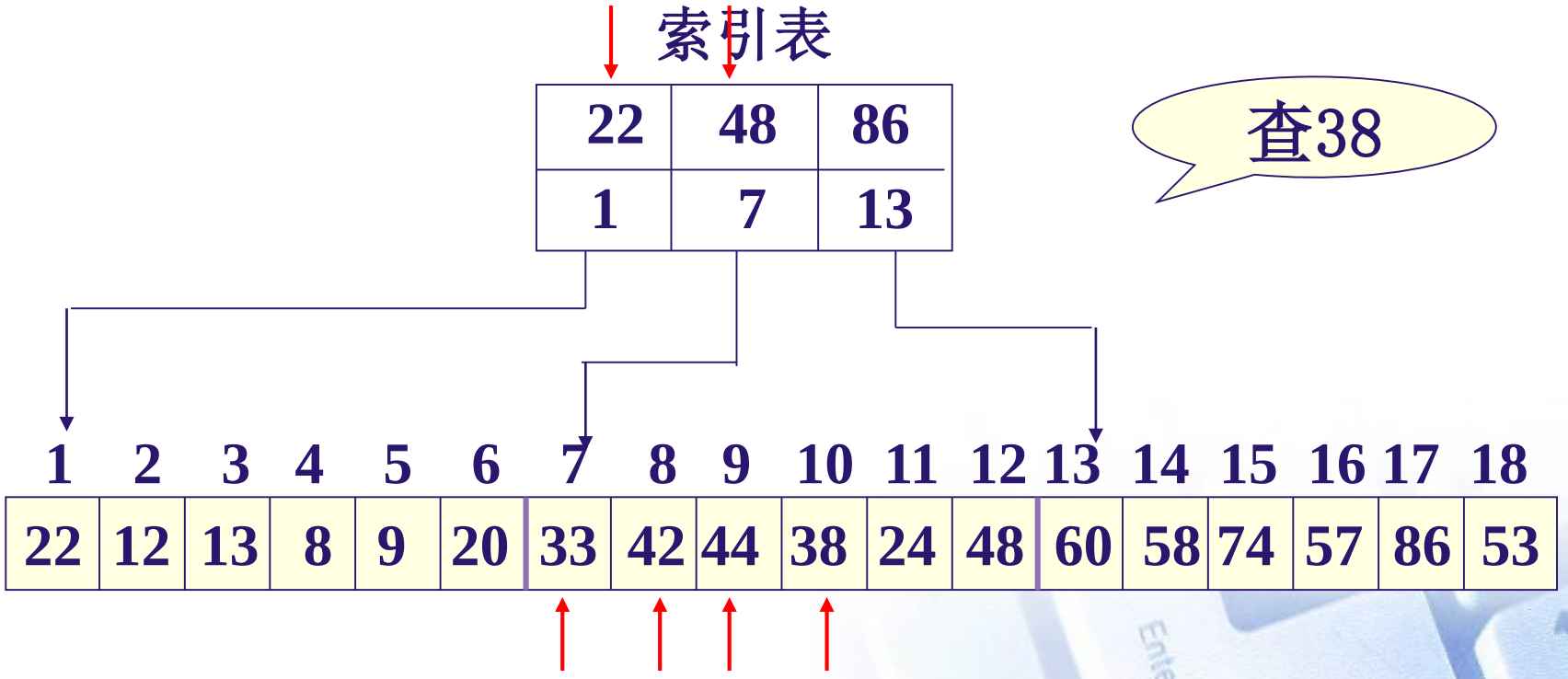
- 将待查找表(长度=n）按关键字等长的分为若干个子表即块，R_1，R_2, …, R_b
- 块内的元素（s个记录）无序
- 块之间有序，即$R_i.key <= R_{i+1}.key$
- 索引表中的每个元素含有各块的最高关键字和各块中第一个元素的地址（下标）
- 索引表按关键字有序排列

查找时，对索引表可以顺序查找或二分法查找，在块内只能顺序查找。

静态表的查找

- 查找过程：将表分成几块，块内无序，块间有序；先确定待查记录所在块，再在块内查找
- 适用条件：分块有序表
- 算法实现
  - 用数组存放待查记录，每个数据元素至少含有关键字域
  - 建立索引表，每个索引表结点含有最大关键字域和指向本块第一个结点的指针

静态表的查找


索引表
22 48 86
1 7 13
查38
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
22 12 13 8 9 20 33 42 44 38 24 48 60 58 74 57 86 53

分块查找方法评价

$ASL_{bs} = L_b + L_w$

其中：L_b——查找索引表确定所在块的平均查找长度

L_w——在块中查找元素的平均查找长度

若将表长为n的表平均分成b块（$b=\lceil n/s \rceil$），每块含s个记录，并设表中每个记录的查找概率相等，则：

(1)用顺序查找确定所在块：$ASL_{bs} = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i$

$$= \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

(2)用折半查找确定所在块：$ASL_{bs} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$

查找方法的比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表 无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

综合前面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(log2n)	O(n)	O(n)
有序线性链表	O(n)	O(1)	O(1)

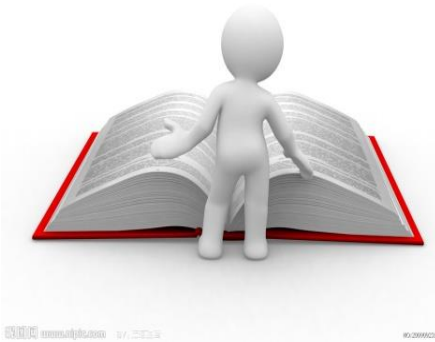
可得如下结论：

1）从查找性能看，最好情况能达$O(\log_2 n)$，此时要求表有序；

2）从插入和删除的性能看，最好

而动态查找表的特点：表结构是在查找过程中动态产生，找到等于关键字key的记录，则查找成功，否则插入该关键字

本章内容



1. 基本概念与术语
2. 静态查找表
3. 动态查找表
4. 哈希表
5. 本章小结

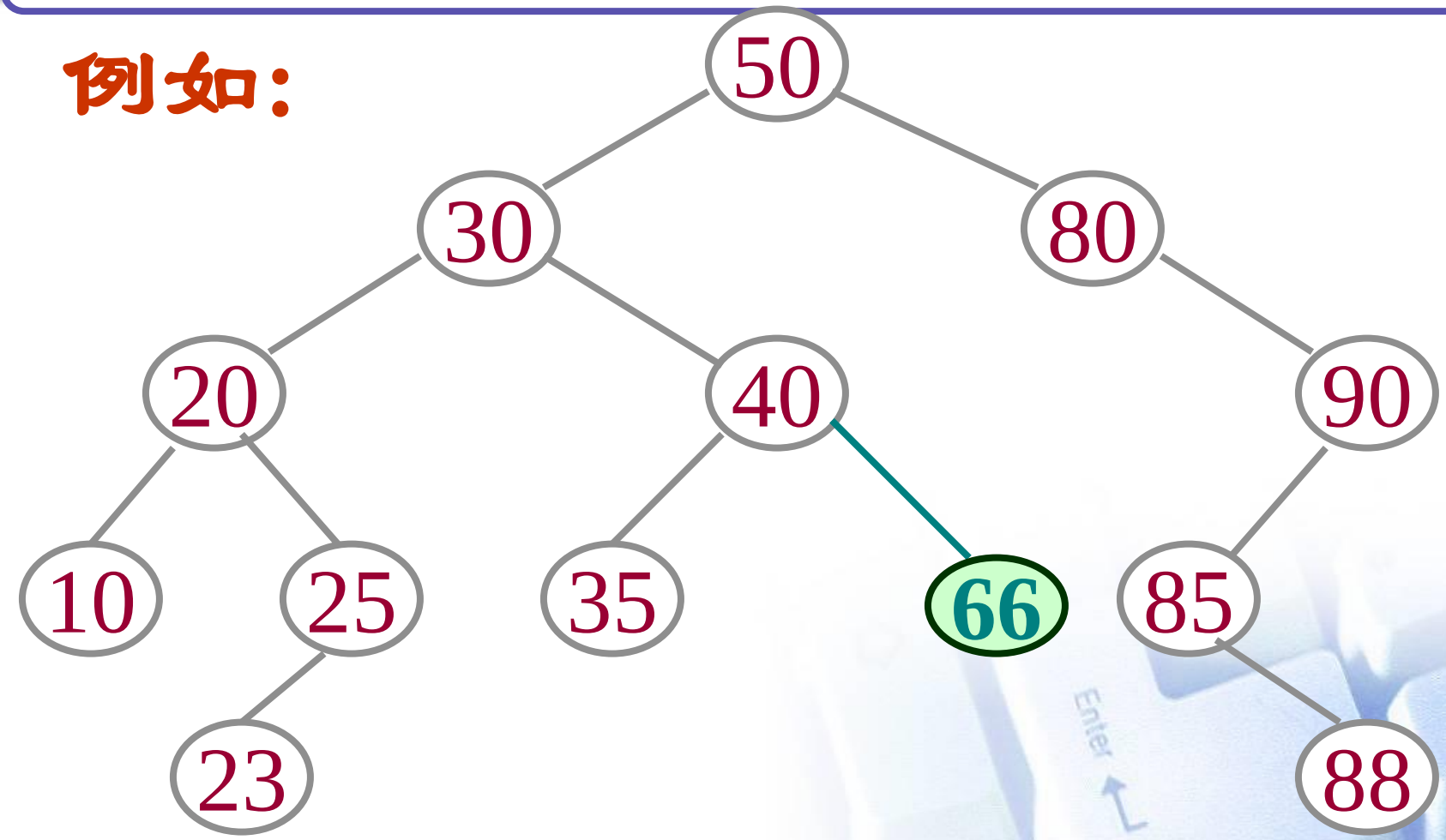
- 二叉排序树
- 平衡二叉树
- B-树
- B+树

◆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：

- 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；
- 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；
- 它的左、右子树也都分别是二叉排序树。

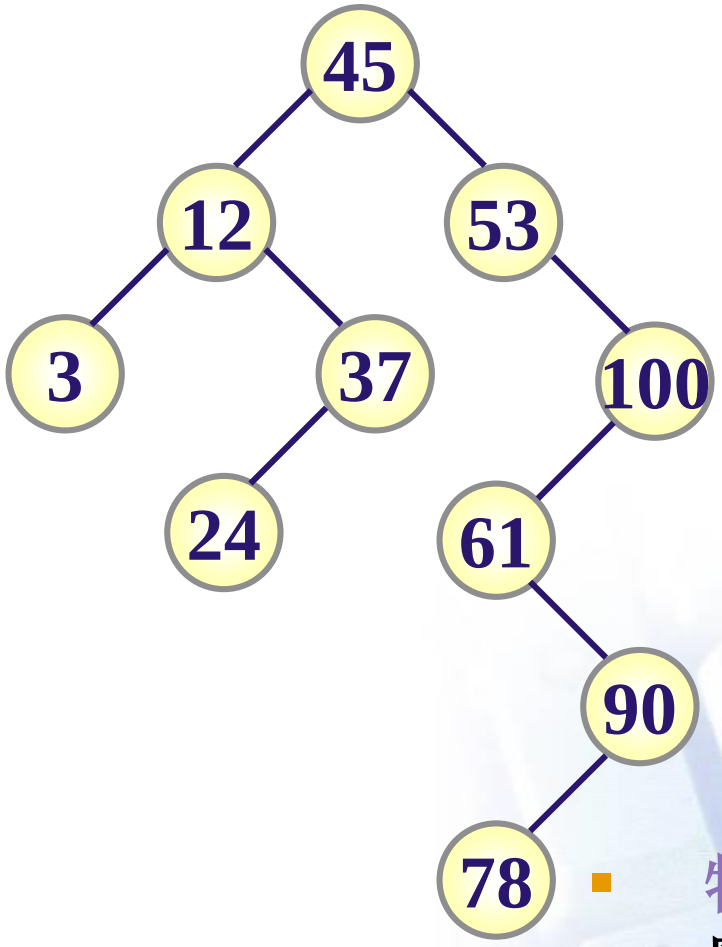
例如：



不是二叉排序树。

动态查找表

- 实例 key=(45,12,53,3,37,100,24,61,90,78)



- 特点：中序遍历二叉排序树所得结点序列即为有序序列

- 二叉排序树的查找算法
  - 假定二叉排序树的根结点指针为root，给定的关键字值为K，则查找算法可描述为：

(1)置初值：T=root;

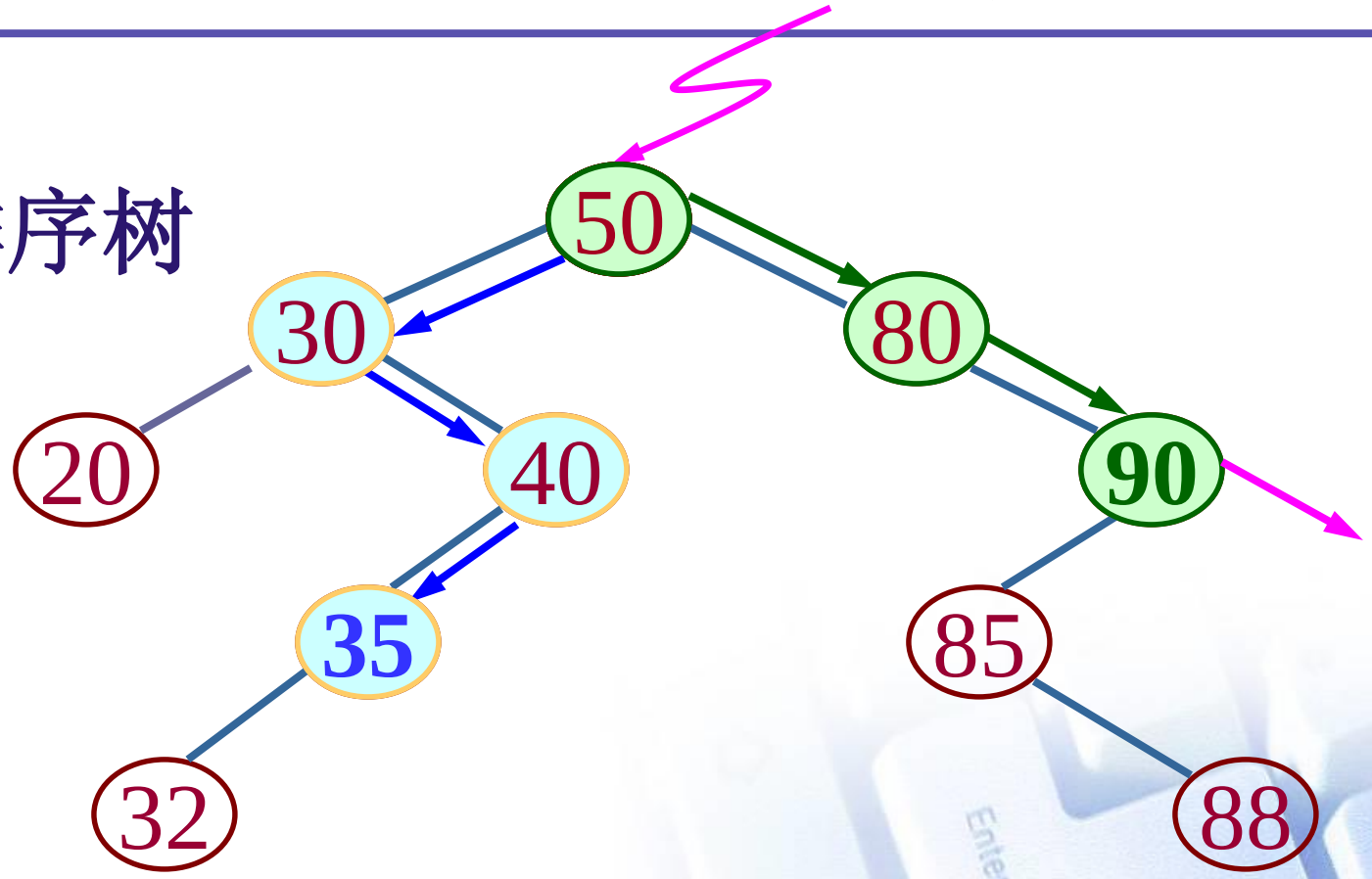
(2)如果K=T->key，则查找成功，算法结束；

(3)否则

如果K＜T->key，而且T的左子树非空，则将T的左子树根送T，转步骤(2)；否则（左子树为空），查找失败，结束算法；

否则，如果K＞T->key，而且T的右子树非空，则将T的右子树根送T，转步骤(2)；否则（右子树为空），查找失败，算法结束。

例如：
二叉排序树



查找关键字

== 50 , 35 , 90 , 95 ,

从根结点出发，沿着左分支或右分支递归进行查询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递归进行查询直至子树为空树止。

——查找不成功

❑算法实现

设 key = 22

递归：

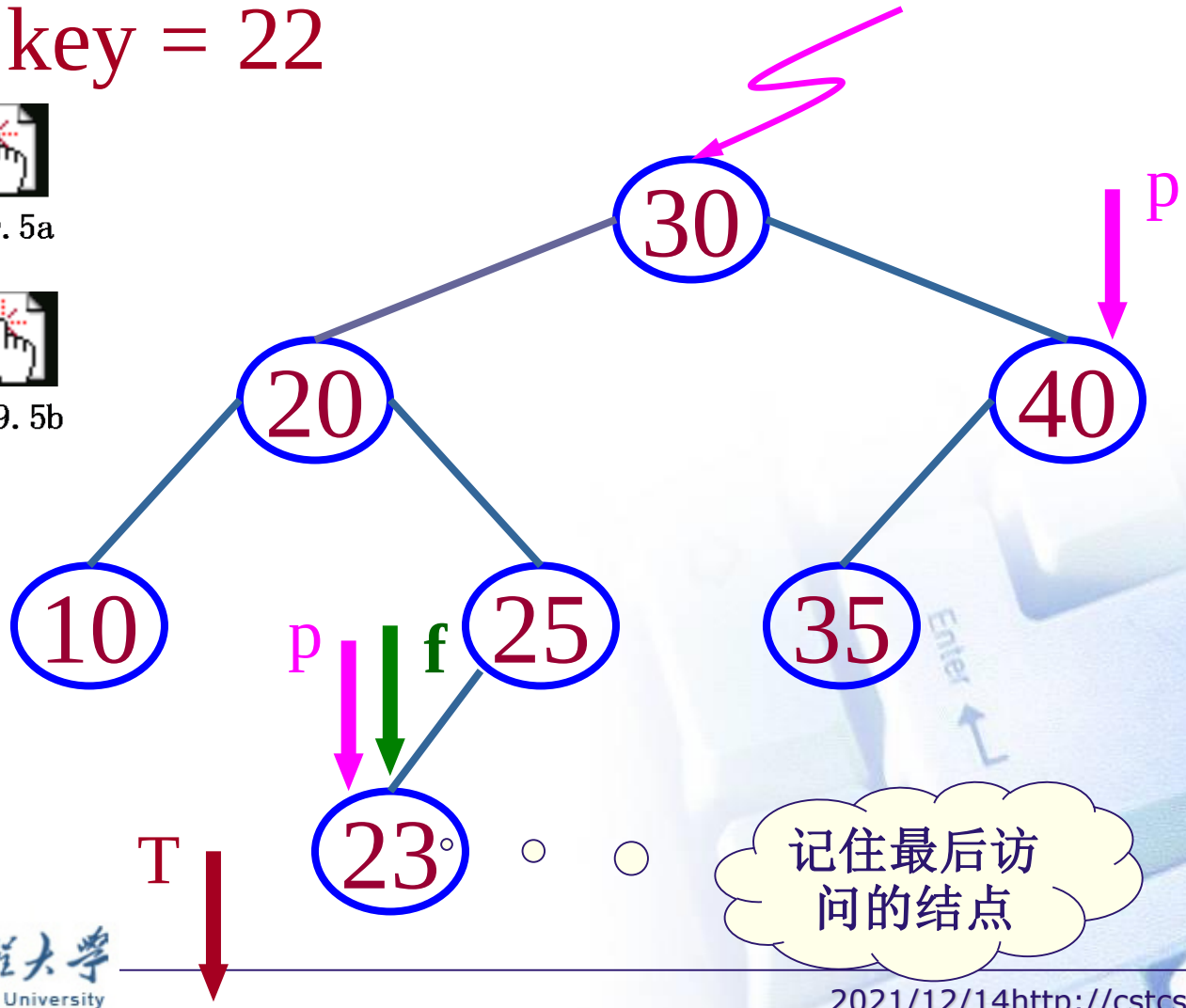
sf9.5a

插入需要：

sf9.5b

非递归：

sf9.5c



◆ 二叉排序树的插入

- 思想：插入的结点一定是一个新添加的叶子结点，且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子
- 调用查找过程SearchBST (root，k，null，p)；
- 若查找不成功，即函数返回FALSE，p指向最后访问的结点时做：

  ① 动态生成一具有关键字值为K的新结点s；

  ② 若p为NULL（空树），则root=s；

  ③ 若K＜p->key，则p->lchild=s；

  ④ 若K＞p->key，则p->rchild=s；
- 算法结束

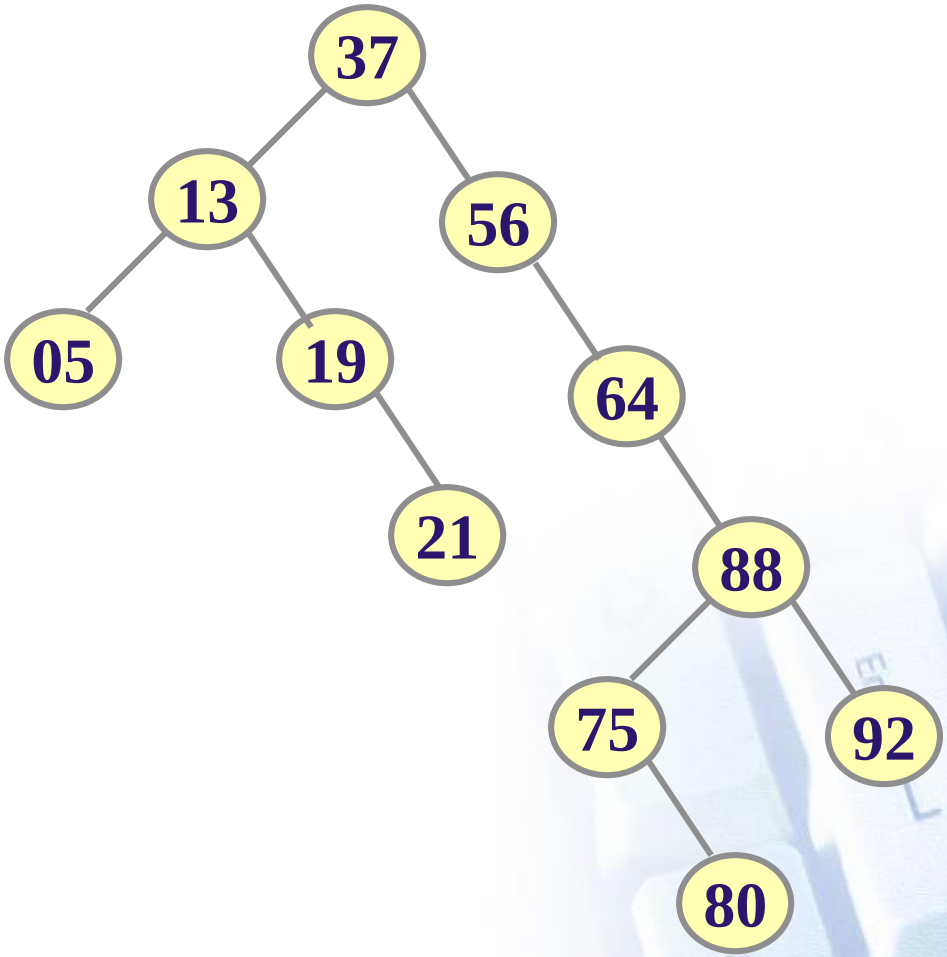
生成二叉排序树的过程就是一个查找、插入的过程

插入算法

sf9.6

动态查找表

例： 37，13，05，19，21，56，64，88，75，92，80



◆二叉排序树的删除

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

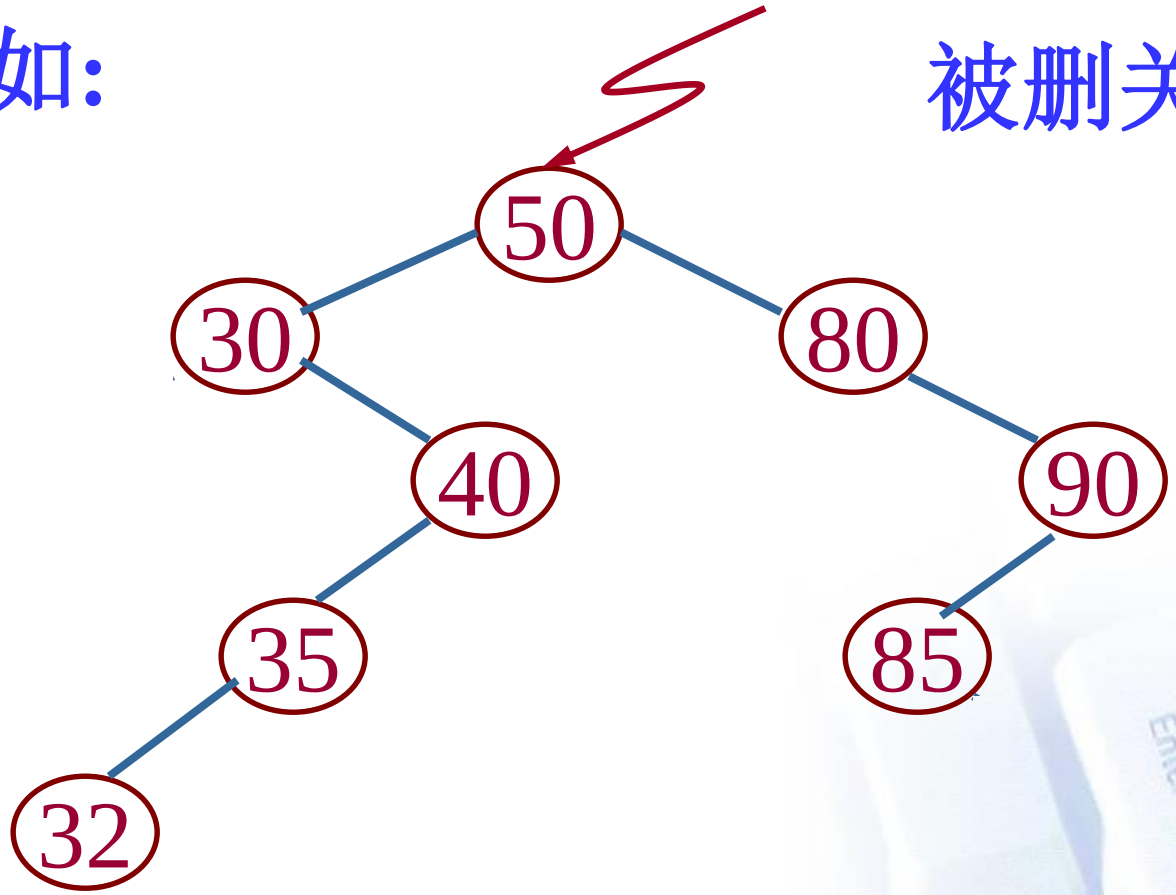
（2）被删除的结点只有左子树或者只有右子树

（3）被删除的结点既有左子树，也有右子树

被删除的结点是叶子结点

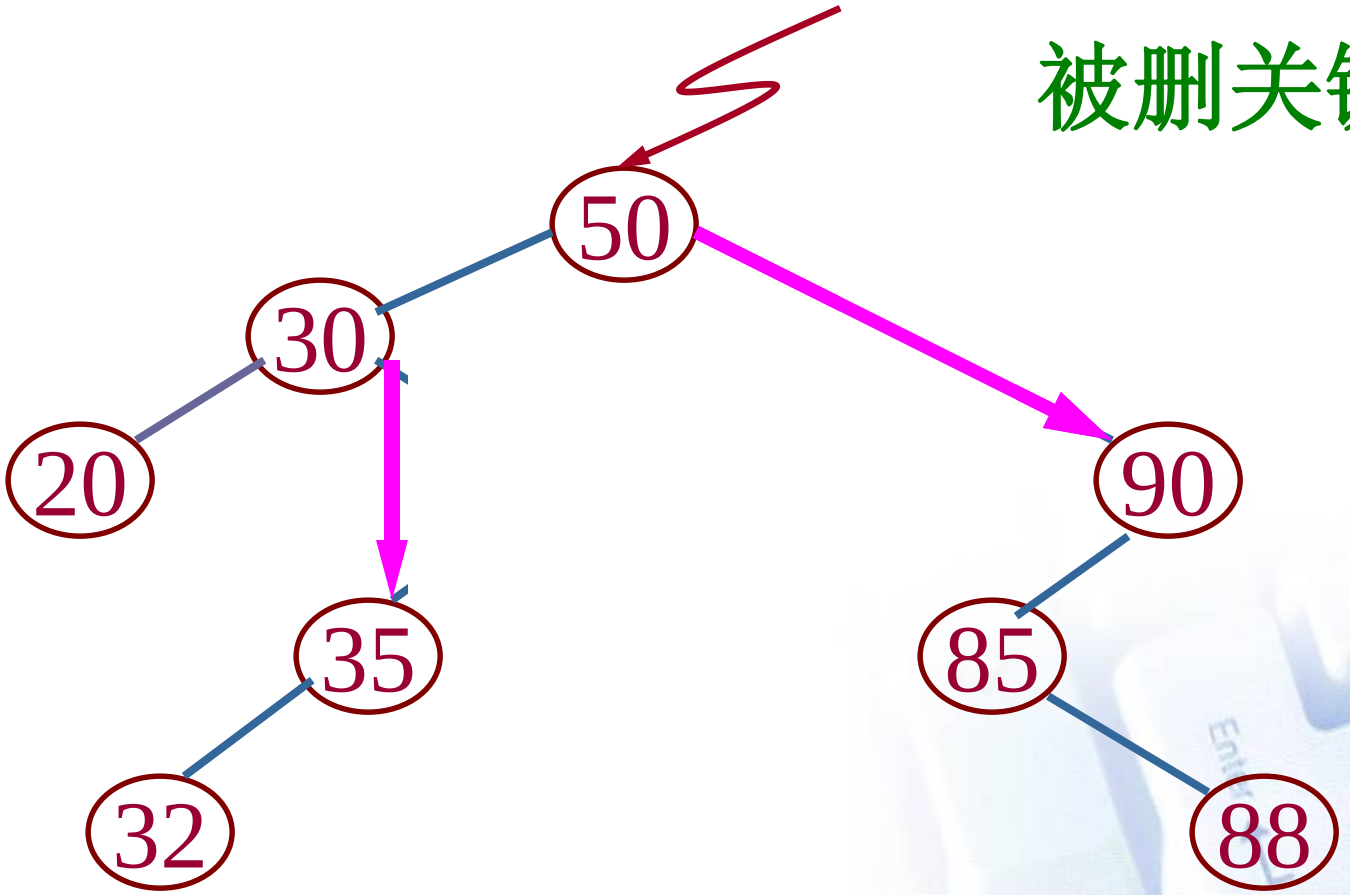
例如：

被删关键字 = **88**



其双亲结点中相应指针域的值改为“空”

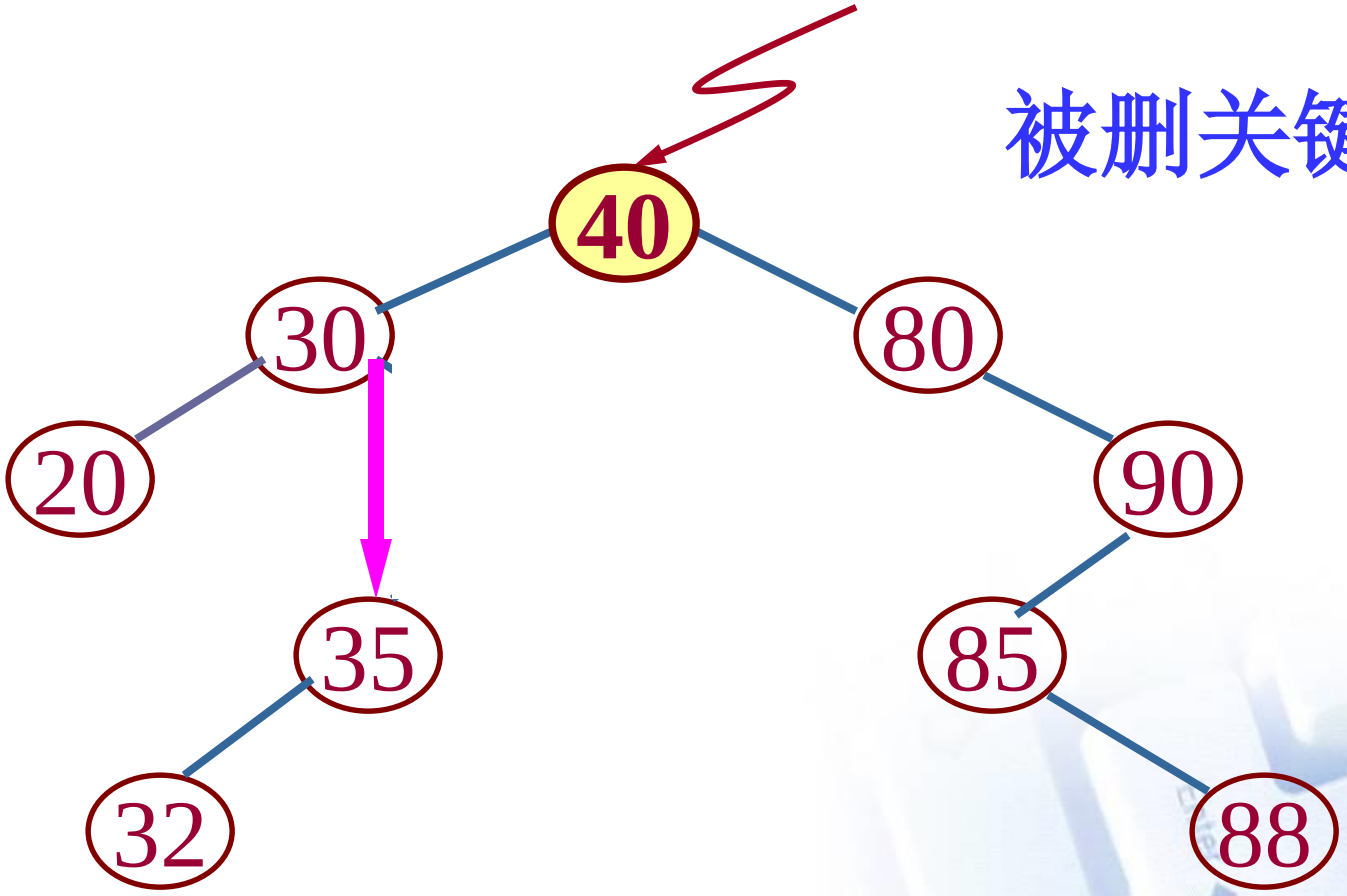
被删除的结点只有左子树或只有右子树



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”。

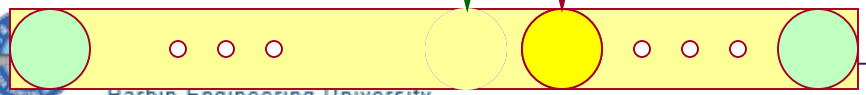
被删除的结点既有左子树，也有右子树



被删关键字 = 50

前驱结点

被删结点

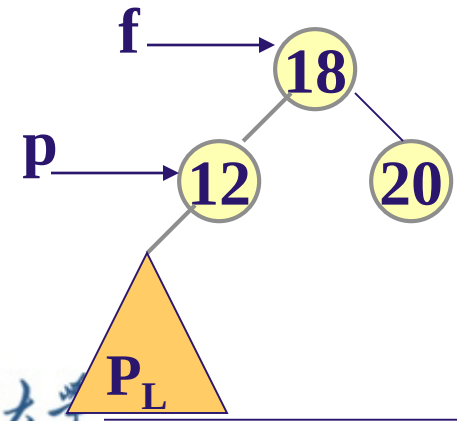


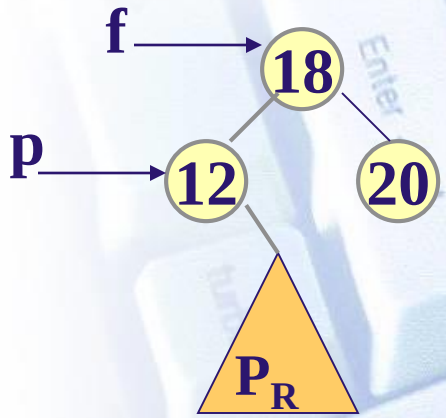
以其前驱替代之，然后再删除该前驱结点

设要删除结点*p，*f为其双亲结点，且*p为*f的左孩子，则：

1）若*p结点为叶子结点，即其P_L和P_R均为空，则修改其双亲指向该结点的指针为空

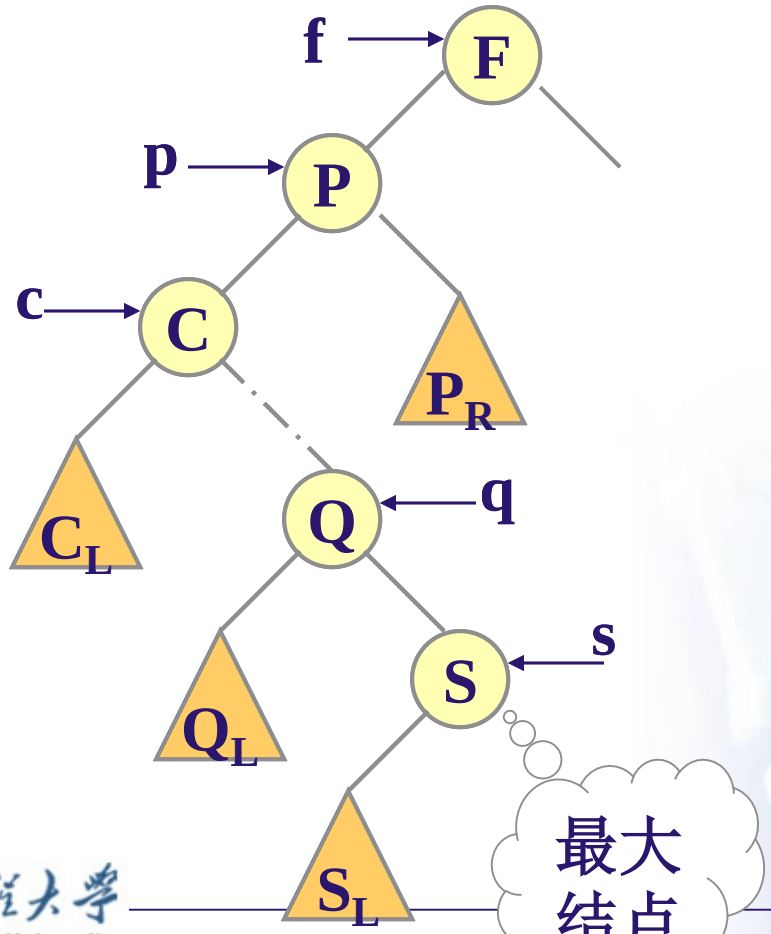
2）若*p结点只有P_L或只有P_R，则P_L或P_R为其双亲*f的左子树

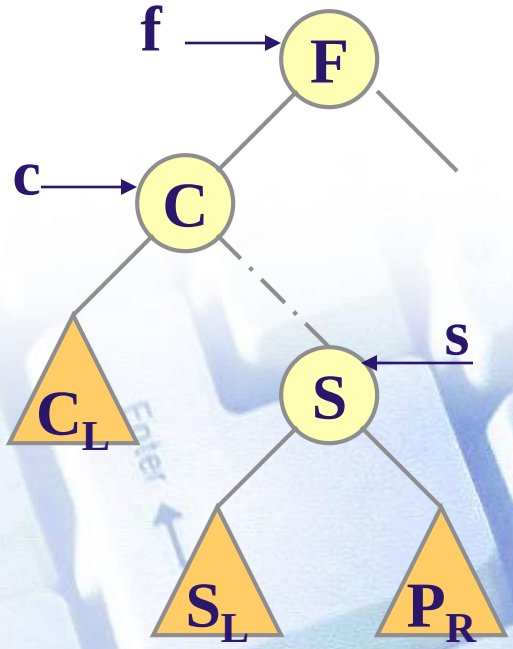




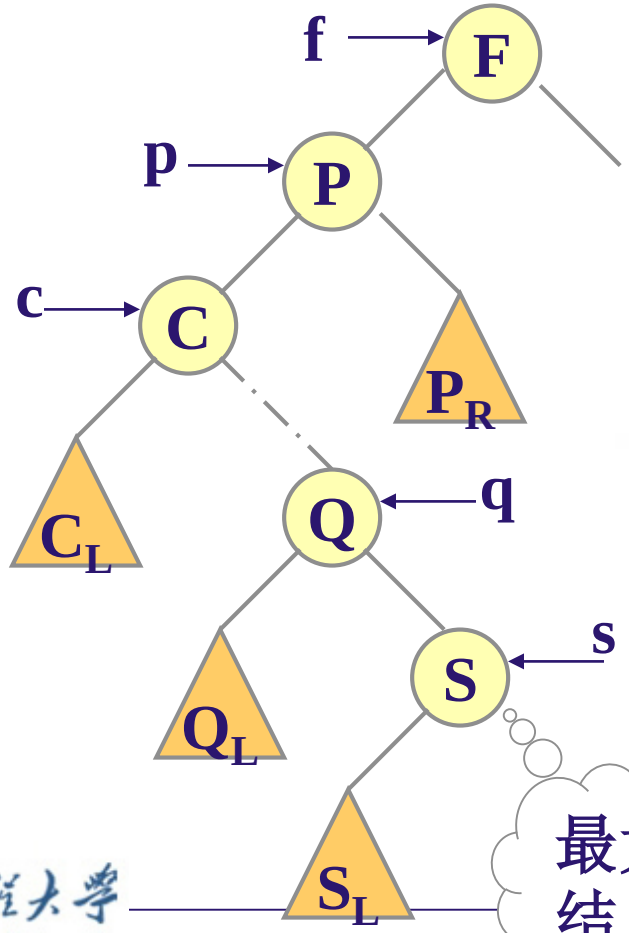
3）若*p结点左右子树均不为空，两种做法：

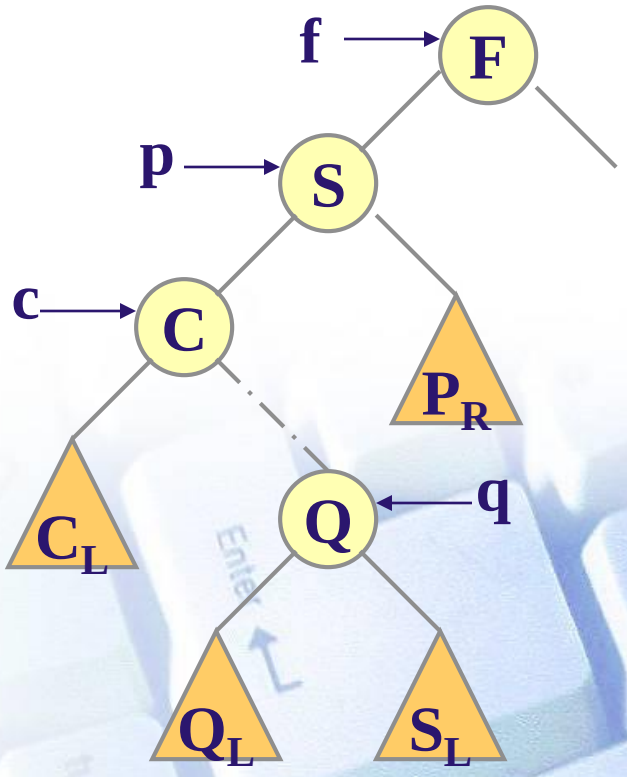
a. 让*p的左子树为*f的左子树，*p的右子树为*s的右子树；





b. 让*p的直接前驱（或直接后继）替代*p，然后再从二叉排序树中删去它的直接前驱（或直接后继）





二叉排序树的删除算法

- 二叉排序树的查找分析
  - 若查找成功，则是从根结点出发走了一条从根到某个叶子的路径
  - 与关键字比较次数不超过该二叉树的深度。深度为i的结点，查找成功时所需比较次数为i。因此，对于深度为d的二叉排序树，若设第i层有n_i个结点(1≤i≤d)，则在等概率的情况下，其平均查找长度为

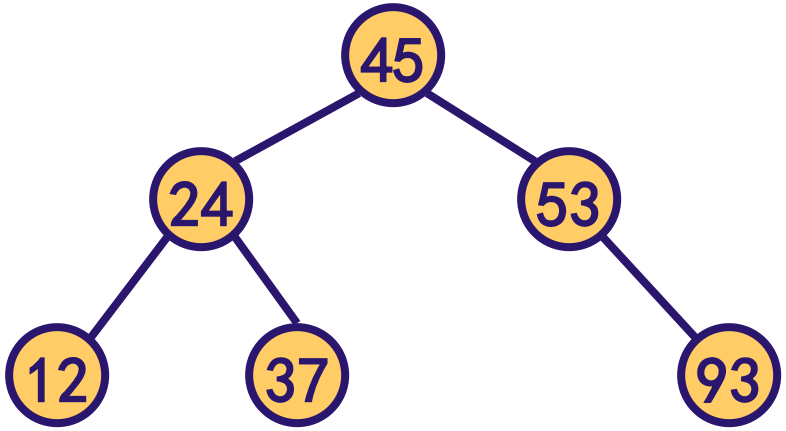
$$ASL = \frac{1}{n}\sum_{i=1}^{d} i \times n_i$$

其中，$n=1+n_2+\ldots+n_d$为二叉树的结点数。

动态查找表

- n个结点的二叉排序树不唯一，由关键字插入的先后次序决定。
- 二叉排序树的平均查找长度与树的形态（深度）有关。

关键字序列(45, 24, 53, 12, 37, 93)构成二叉排序树。



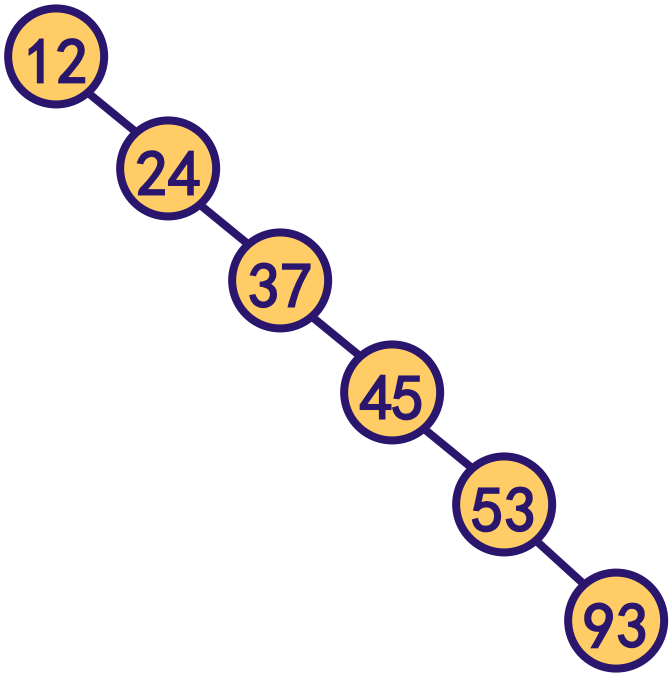
在查找概率相等的情况下，

$$ASL = \frac{1}{6}(1+2\times2+3\times3) = \frac{14}{6}$$

动态查找表

- n个结点的二叉排序树不唯一，由关键字插入的先后次序决定。
- 二叉排序树的平均查找长度与树的形态（深度）有关。

关键字序列(12, 24, 37, 45, 53, 93)构成二叉排序树。



在查找概率相等的情况下，

$$ASL = \frac{1}{6}(1+2+3+4+5+6) = \frac{21}{6}$$

动态查找表

- 关键字输入序列
  - 最差的情况
    - 二叉排序树为单支树，深度为n
  - 最好的情况
    - 树的形状比较匀称，二叉排序树和折半查找判定树相同，深度为$\lfloor \log_2 n \rfloor + 1$
  - 平均性
    - 二叉排序树上的查找和折半查找相差不大，并且二叉排序树上的插入和删除结点十分方便，无须移动大量结点

※结论：需要经常做插入、删除和查找运算的表，选择二叉排序树结构较好

由此，二叉排序树也称为二叉查找树

❖ 能否用一个更简单的方法来判别给定的二叉树是否是二叉排序（查找）树呢？

若对二叉树进行“中序遍历”，得到的是一个有序序列，则为二叉查找树；否则，不是

❖二叉查找树实质上是一个“有序表”

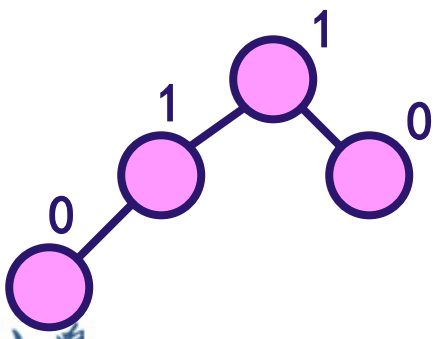
问题：单支树查找效率低，怎么办？

思路：在动态生成二叉排序树的过程中，进行平衡化处理，使之成为平衡二叉树

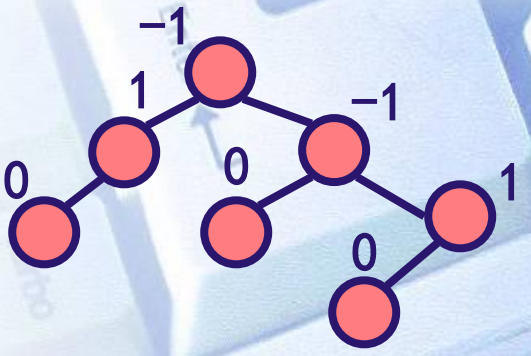
注意：平衡二叉树与关键字序列无关

- 二叉排序树
- 平衡二叉树
- B-树
- B+树

- 平衡二叉排序树（AVL树）
  - 结点的平衡因子
    - 结点的左子树的深度减去它的右子树的深度。
  - 定义
    - 它或是空树，或具有下列特性：
      - 左子树和右子树都是平衡二叉树
      - 左右子树深度之差的绝对值不大于1。
    - 即，所有结点的平衡因子的绝对值不超过1的二叉树。



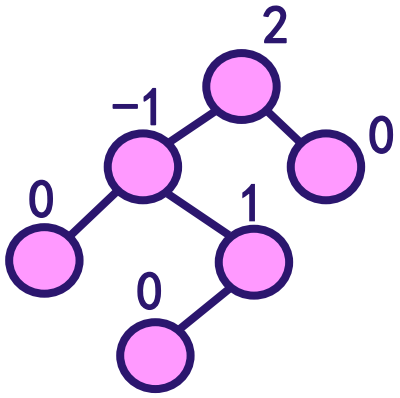
平衡二叉树



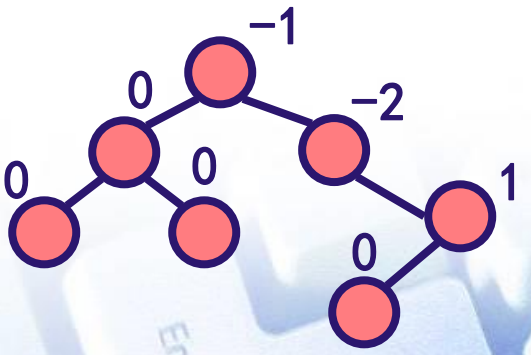
平衡二叉树

◆平衡二叉排序树

➢即，所有结点的平衡因子的绝对值不超过1的二叉树。



非平衡二叉树



非平衡二叉树

- 动态平衡技术构造平衡二叉树
  - 基本思想
    - 从空树起（空树是平衡树），每插入一个关键字检查一次二叉排序树是否失去平衡
    - 如果是因插入结点而破坏了树的平衡性，则找出其中最小不平衡子树
    - 对它进行“平衡旋转”处理。在保持排序树特性的前提下，调整最小不平衡子树中各结点之间的连接关系，以达到新的平衡

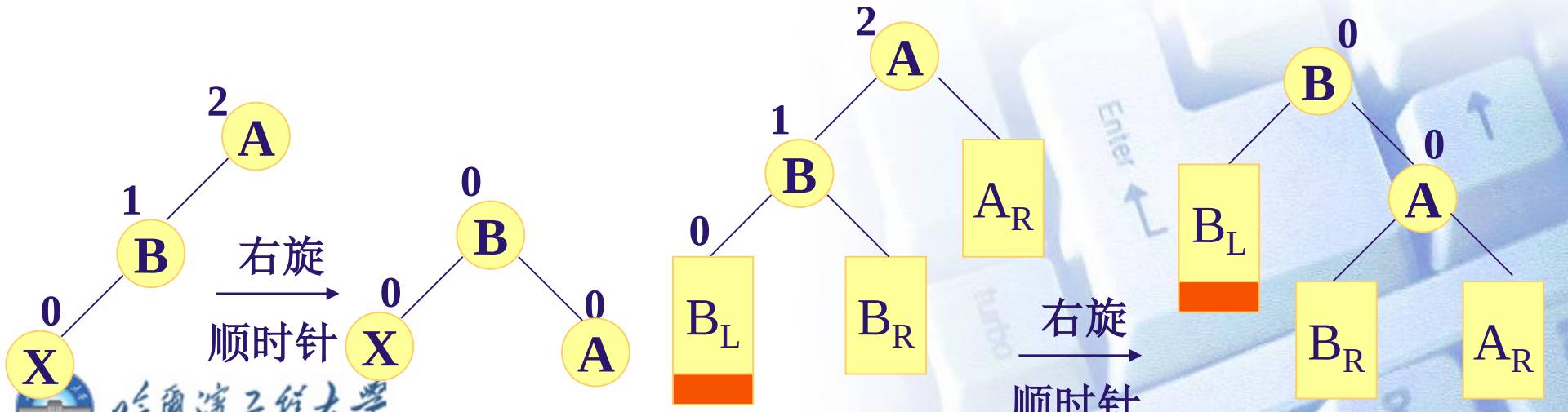
为什么只需要对"最小不平衡子树"进行旋转处理？

因为经过旋转处理之后的子树的深度和插入新的关键字之前的子树深度相同，因而不因插入而改变其所有祖先结点的平衡因子的值。

- 平衡处理
- 设二叉排序树的最小不平衡子树的根结点为A，则调整该子树的规律可归纳为下列四种情况：

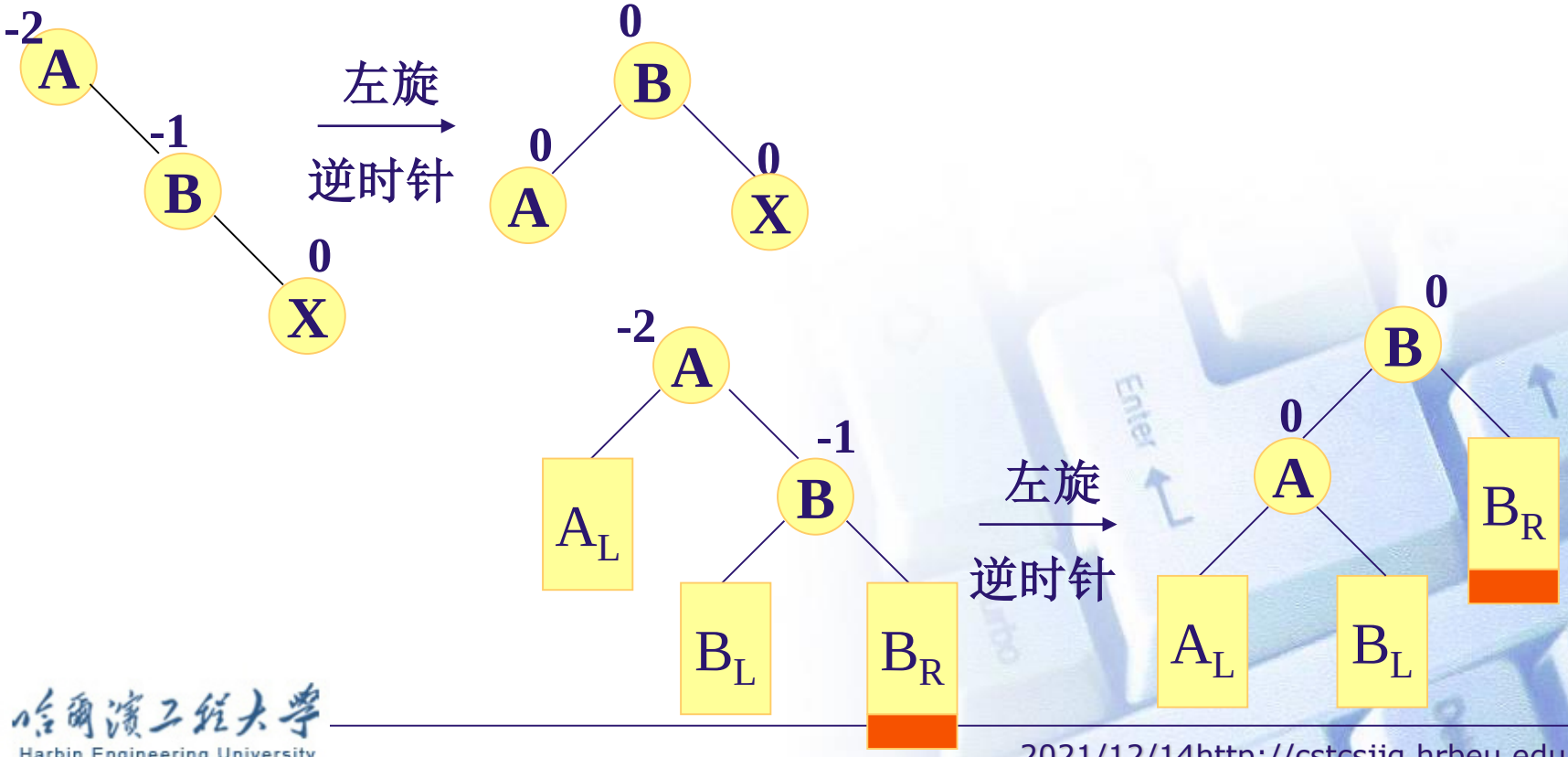
（1）LL型：新结点X插在A的左孩子B的左子树里，A的平衡因子由1增至2。

- 调整方法:以B为轴心，将A结点从B的右上方转到B的右下侧，使A成为B的右孩子



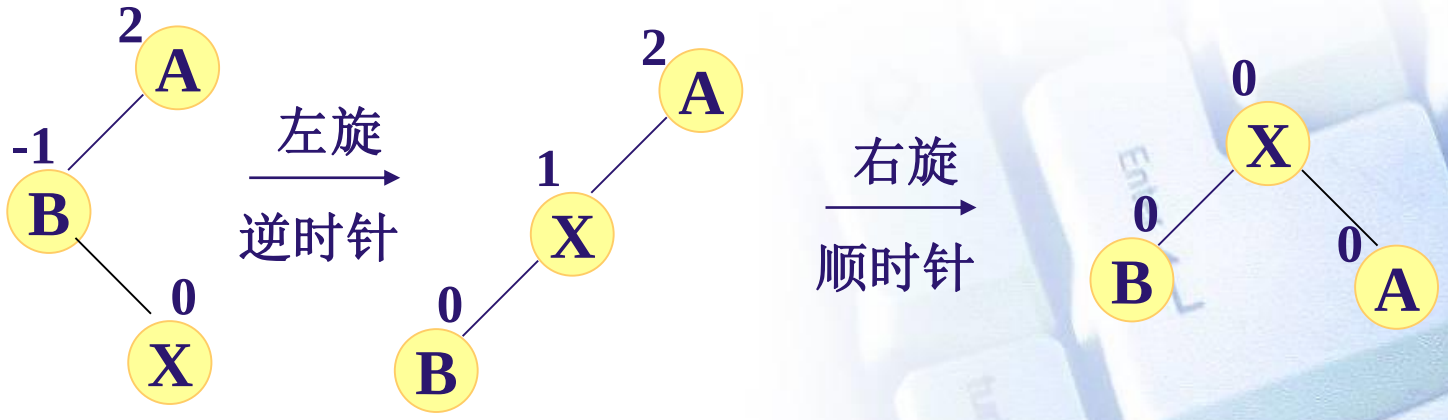
(2)RR型：新结点X插在A的右孩子B的右子树里，A的平衡因子由-1增至-2。

➢ 调整方法：以B为轴心，将A结点从B的左上方转到B的左下侧，使A成为B的左孩子。

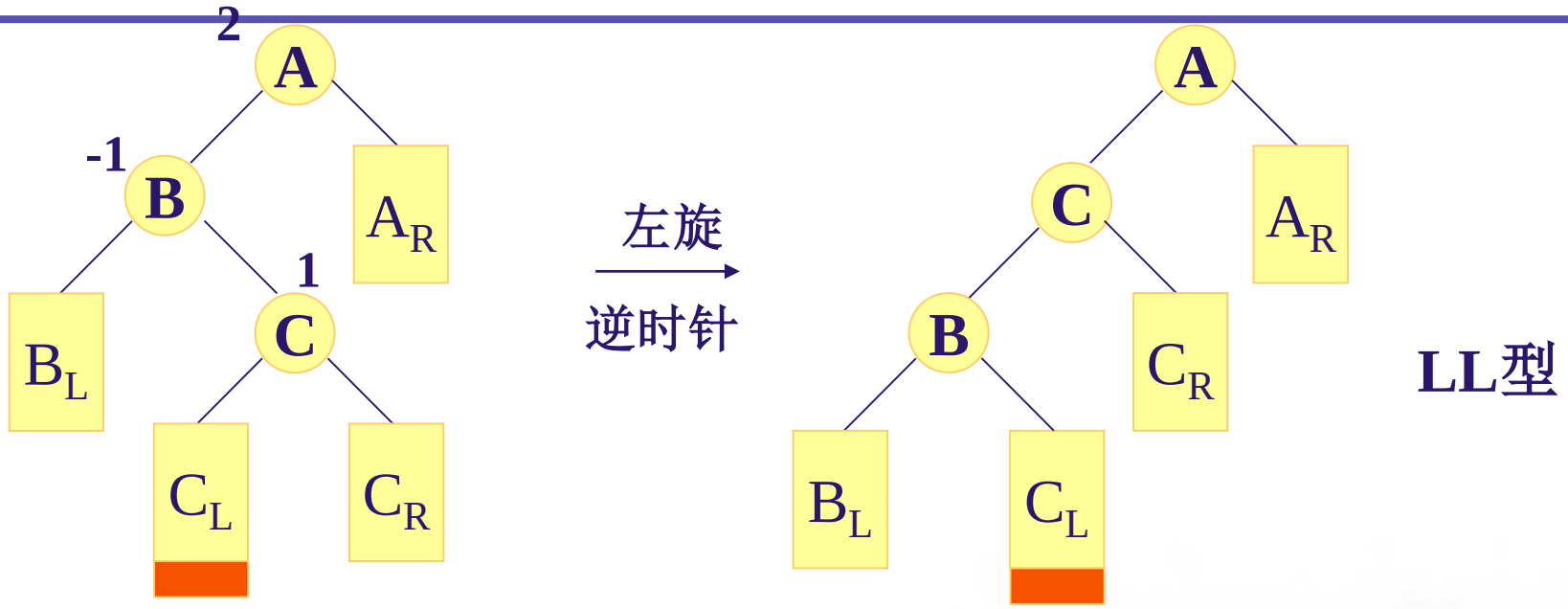


(3)LR型：新结点X插在A的左孩子B的右子树里，A的平衡因子由1增至2。

- 调整方法分为两步进行：第一步以X为轴心，将B从X的左上方转到X的左下侧，使B成为X的左孩子，X成为A的左孩子。第二步跟LL型一样处理(应以X为轴心)

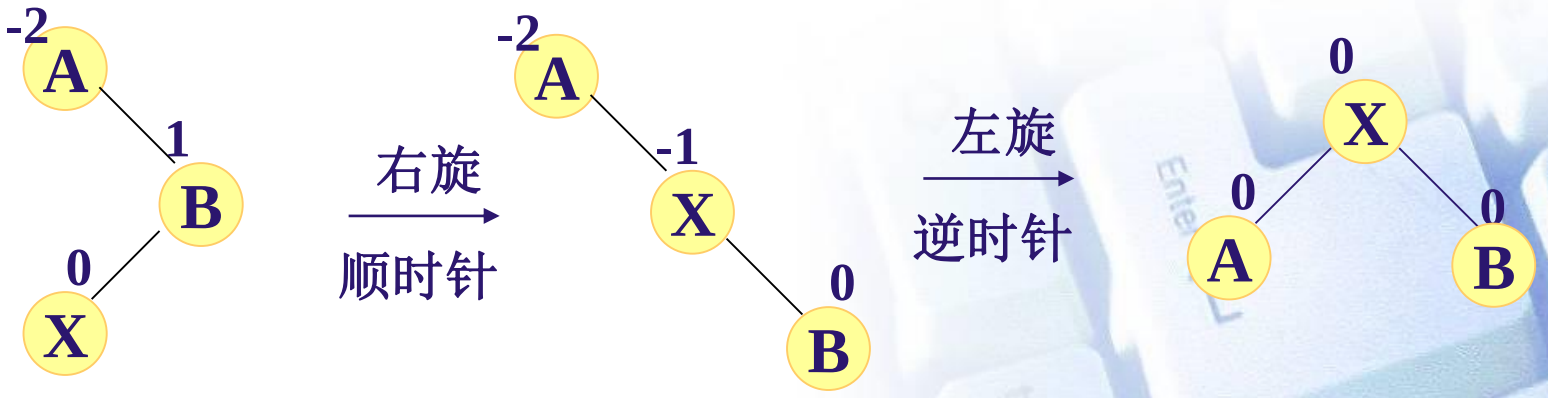


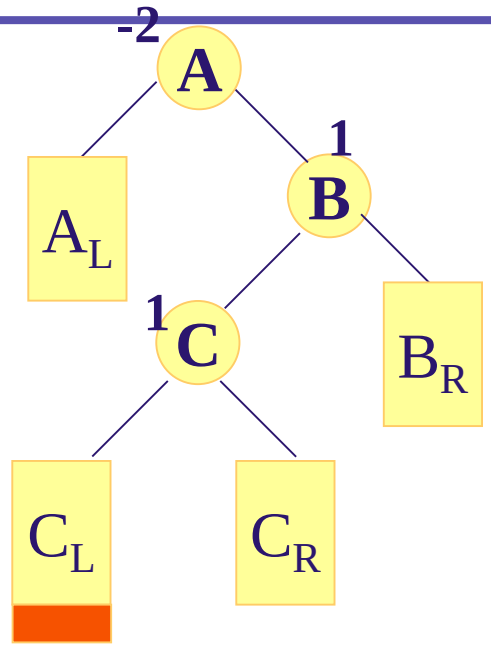
动态查找表


2
A
-1
B
1
C
A_R
B_L
C_L
C_R
左旋
逆时针
A
C
B
A_R
C_R
B_L
C_L
LL型
LR型
右旋
顺时针
0
C
0
B
-1
A
B_L
C_L
C_R
A_R

(4)RL型：新结点X插在A的右孩子B的左子树里，A的平衡因子由-1增至-2。

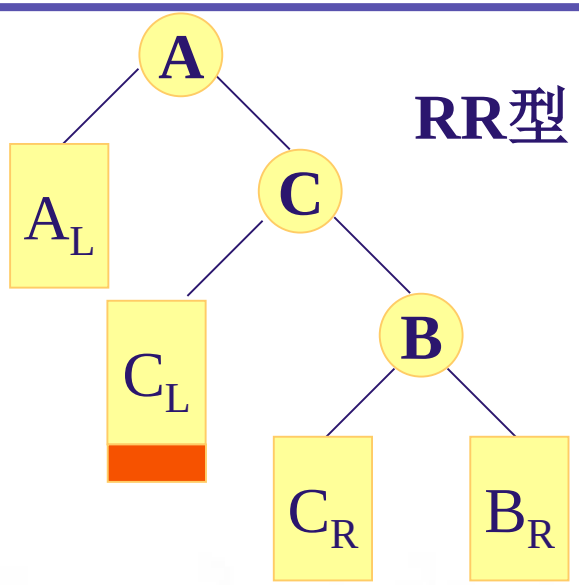
- 调整方法分为两步进行：第一步以X为轴心，将B从X的右上方转到X的右下侧，使B成为X的右孩子，X成为A的右孩子。第二步跟RR型一样处理(应以X为轴心)。





右旋

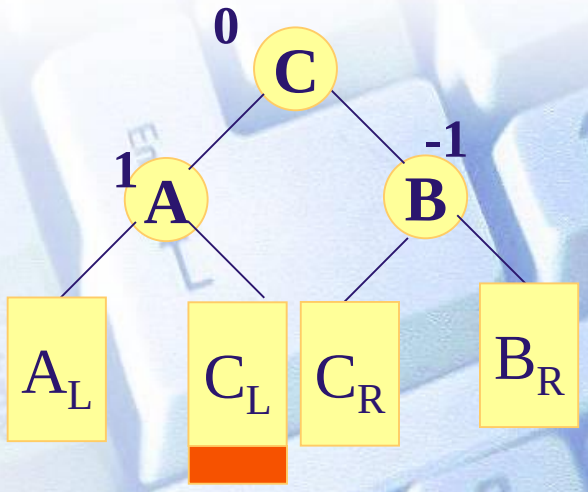
顺时针



RL型

左旋

逆时针



- 实际的插入情况可能要复杂
- 例，设一组记录的关键字按以下次序进行插入：4、5、7、2、1、3、6构造二叉平衡树的过程

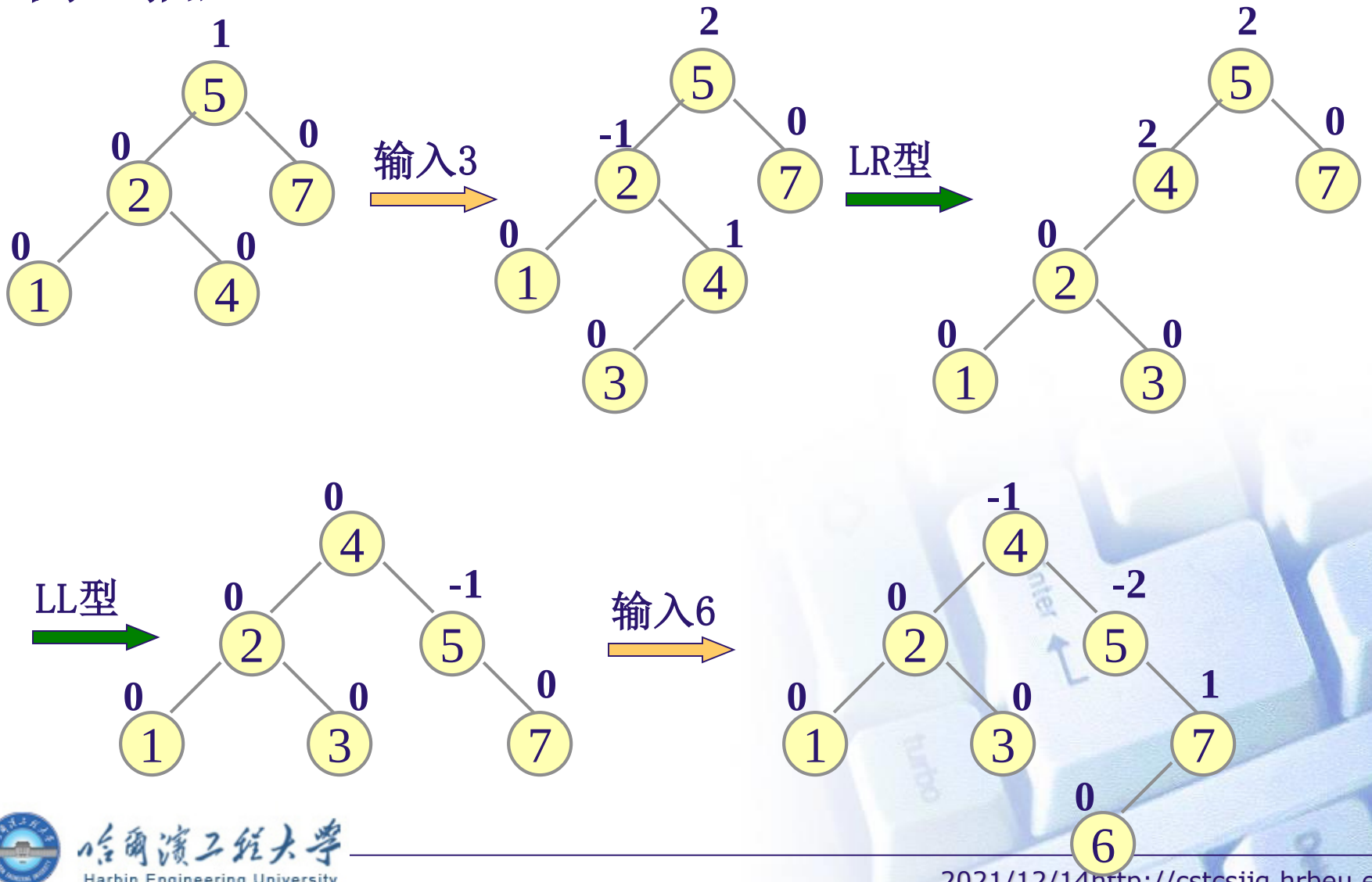
- 当插入关键字为3的结点后，由于离结点3最近的平衡因子为2的祖先是根结点5，因此第一次旋转应以结点4为轴心，把结点2从结点4的左上方转到左下侧，从而结点5的左孩子是结点4，结点4的左孩子是结点2，原结点4的左孩子变成了结点2的右孩子。第二步再以结点4为轴心，按LL类型进行转换

例：插入（4、5、7、2、1、3、6）


输入4
0
4
输入5
-1
4
0
5
输入7
-2
4
-1
5
0
7
RR型
0
5
0
4
0
7
输入2
1
5
1
4
0
7
0
2
输入1
2
5
2
4
0
7
1
2
0
1
LL型
1
5
0
2
0
7
0
1
0
4

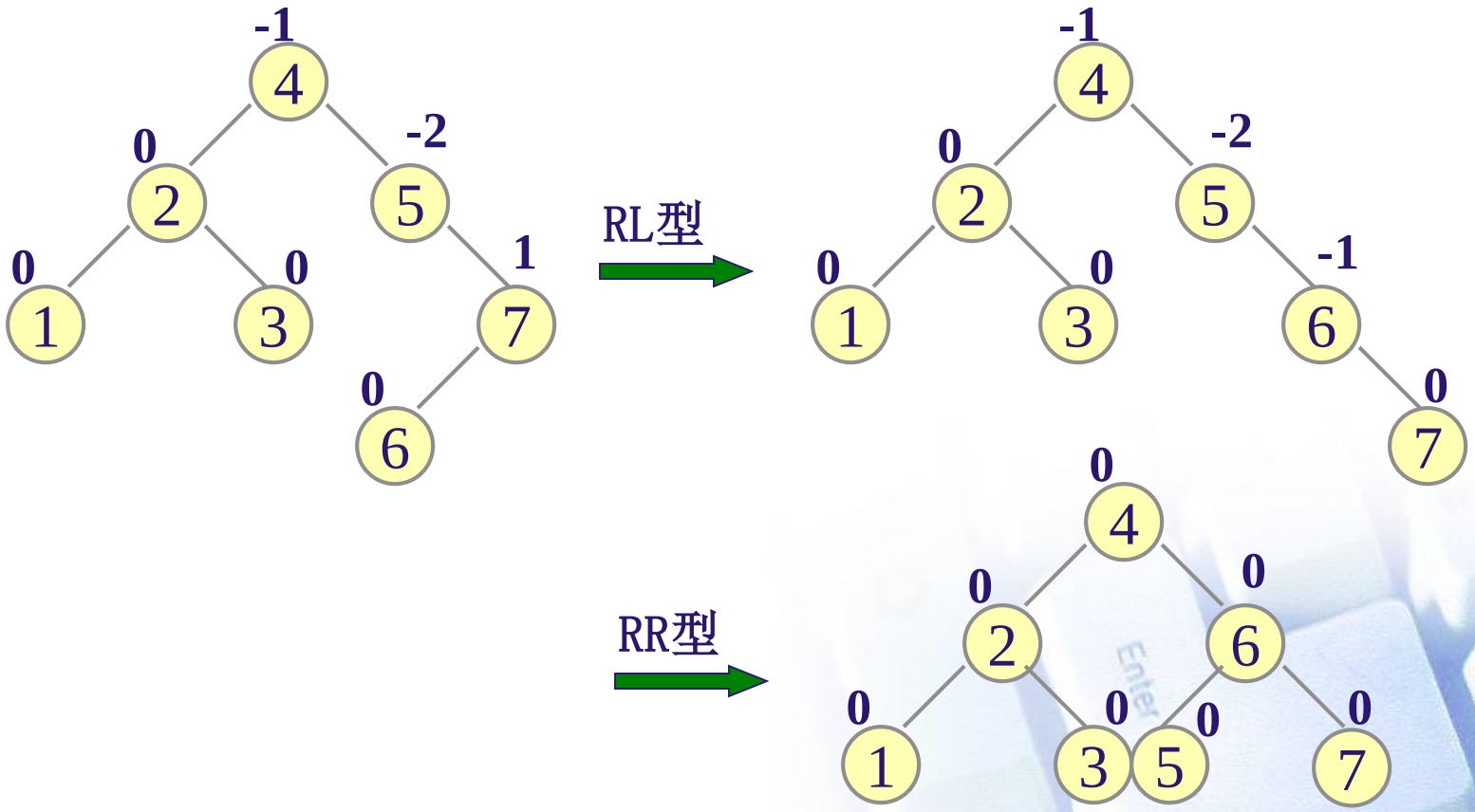
动态查找表

例：插入（4、5、7、2、1、3、6）



动态查找表

例：插入（4、5、7、2、1、3、6）



- 平衡二叉树查找性能分析

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。

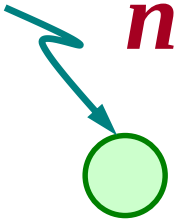
思考：含 *n* 个关键字的二叉平衡树可能达到的最大深度是多少？

先看几个具体情况：

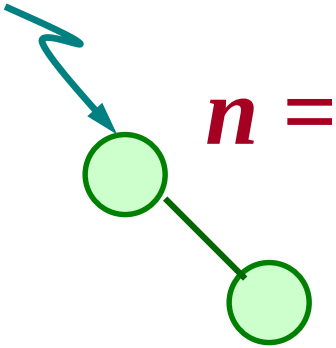
$n = 0$

空树

$n = 1$



$n = 2$

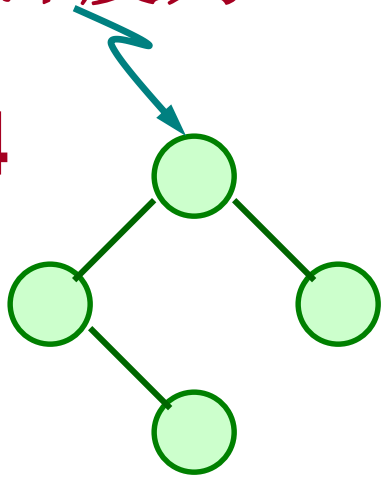


最大深度为 0

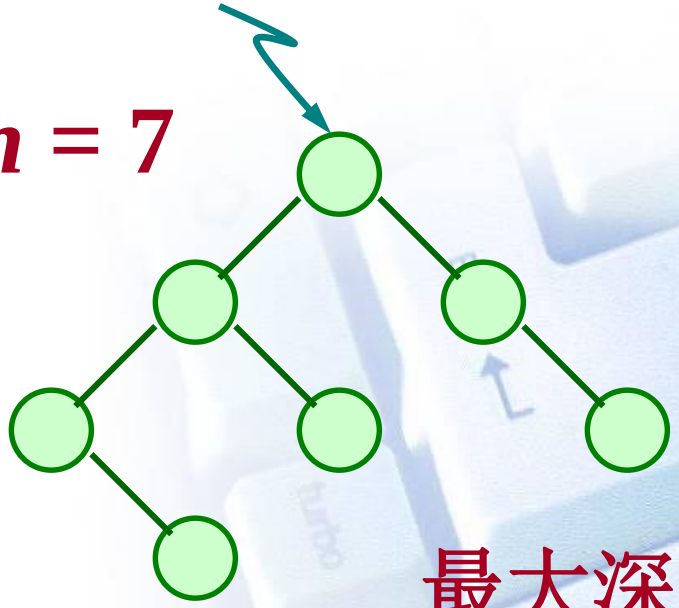
最大深度为 1

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？

$h = 0$　　$N_0 = 0$　　$h = 1$　　$N_1 = 1$

$h = 2$　　$N_2 = 2$　　$h = 3$　　$N_3 = 4$

一般情况下，　$N_h = N_{h-1} + N_{h-2} + 1$

利用归纳法可证得，　$N_h = F_{h+2} - 1$

由此推得，深度为 *h* 的二叉平衡树中所含结点的最小值

$$N_h = \varphi^{h+2} / \sqrt{5} - 1$$

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。

- **B-树定义**

一棵m阶的B-树，或为空树，或为满[illegible]m叉树：

(1)树中每个结点最多有m棵子树；

(2)除根结点和叶结点外，其他每[illegible]⌈m/2⌉棵子树；（即关键字数至少⌈m/2⌉-1）。

(3)根结点至少有两棵子树(惟一例外的是只包含一个根结点的B-树)；

(4)所有的叶结点在同一层，叶结点不包含任何关键字信息；

(5)有n+1个孩子的非叶结点恰好包含n个关键字。

(n, A_0, K_1, A_1, K_2, A_2, …, K_n, A_n)

关键字数 <=m-1

关键字数 >=⌈m/2⌉-1

➢B-树中每个结点中的关键字从小到大排

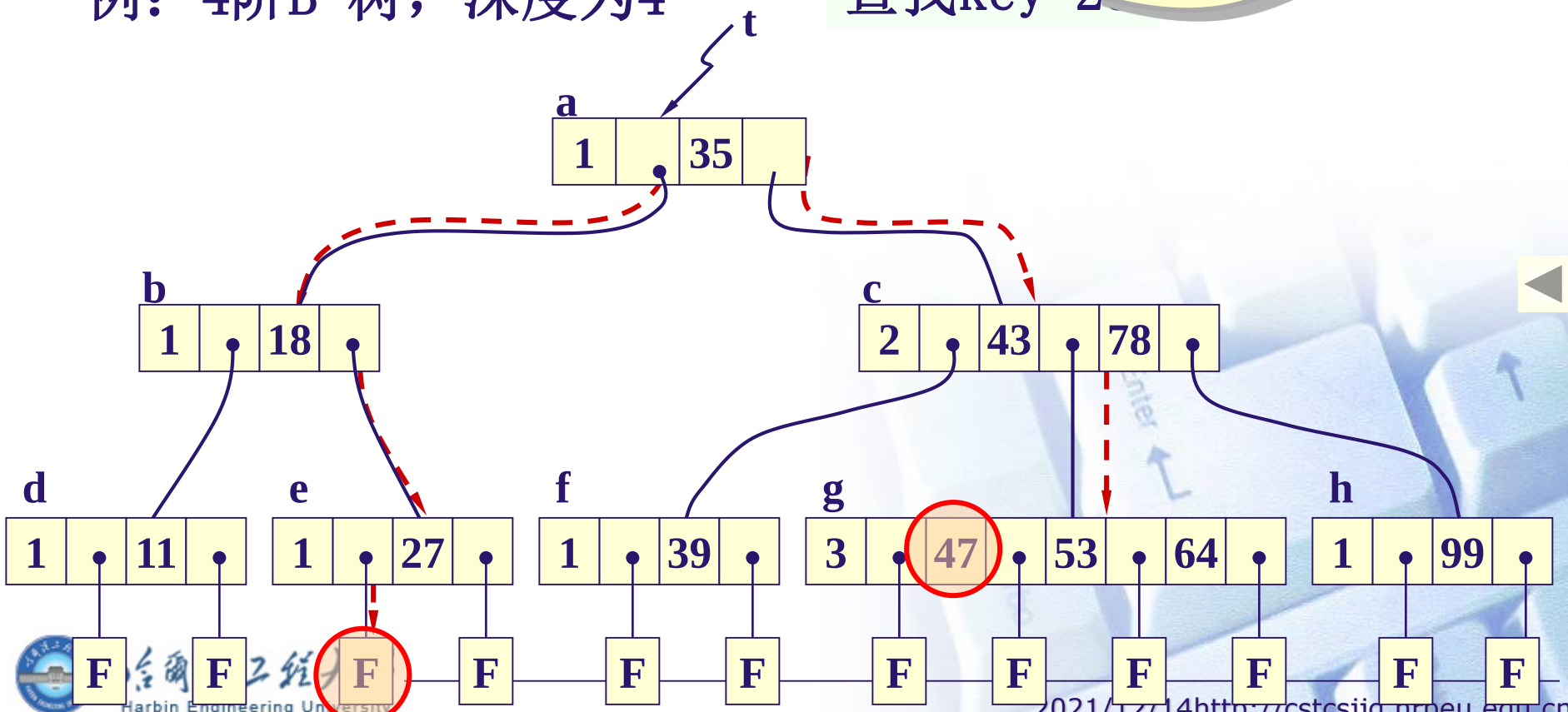
➢叶结点不包含关键字可看成外部结点，指针为空

➢叶结点总数=树中关键字总数+1

例：4阶B-树，深度为4　　查找key=2

思考

4阶B-树的最少关键字和最多关键字数？



- B-树操作
  - B-树查找

（1）把根结点取来，在根结点所包含的关键字K_1，…，K_n中查找给定的关键字(关键字个数不多时，可用顺序查找；个数较多时，可用二分查找)

（2）若找到等于给定值的关键字，则查找成功。否则，一定可以确定要查找的关键字是在某个K_i和K_{i+1}之间(因为在结点内部的关键字是排序的)

（3）于是，取A_i所指向的结点继续查找。

（4）重复（2）（3），直到找到，或指针A_i为空时，查找失败。

- **B-树操作**
  - B-树查找
    - B-树的查找包含两种基本操作
      - 在B-树中找结点（在磁盘上找，找到结点块后，调入内存）
      - 在结点中找关键字（结点块调入内存后，在内存中找）
      - 内存的查找速度远远高于磁盘的查找速度
    - B-树的查找效率取决于在磁盘上查找的次数，即待查关键字所在结点在B-树的层次数。也就是N个关键字的m阶B-树的最大深度是多少？（按最坏情况计算）

B-树查找算法　　动态查找表

➢类型定义


sf9. 13

```
#define m 3
Typedef struct BTNode
{   int                 keynum;          //结点中的关键字个数
    struct BTNode  *parent;              //指向双亲结点
    KeyType          key[m+1];           //关键字向量，0单元不用
    struct BTNode  *ptr[m+1];            //子树指针向量
    Record          *recptr[m+1]; //记录指针向量， 0单元不用
}BTNode, *Btree;                         //B树结点和B树的类型
Typedef struct
{  BTNode  *pt;        //指向找到的结点
    int          i;          //1..m，在结点中的关键字序号
    int          tag;       //1：查找成功；0：查找失败
}Result;                    //B-树查找结果的类型
```

- **B-树查找分析**

若一棵L+1层的m阶B⁻树包含n个关键字，查找失败的关键字会有N+1种情况，而B⁻树叶结点表示树中并不存在的外部结点，正好对应N+1种查找失败的情况。因此，B⁻树有N+1个树叶，树叶都在第L+1层。第一层为根，至少一个结点，根至少有两个孩子，即第二层至少有两个结点。

除根和树叶外，其他结点至少有$\lceil m/2 \rceil$个孩子。因此，第三层至少有$2\times\lceil m/2 \rceil$个结点，第四层至少有$2\times(\lceil m/2 \rceil)^2$个结点……第L+1层至少有$2\times(\lceil m/2 \rceil)^{L-1}$个结点，于是有：

$$n+1\geq 2\times(\lceil m/2 \rceil)^{L-1}$$

在含有n个关键字的B⁻树上进行查找时，从根结点到关键字所在结点的路径上，涉及的结点数不超过L层次数。即：

$$L \leq \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)+1$$

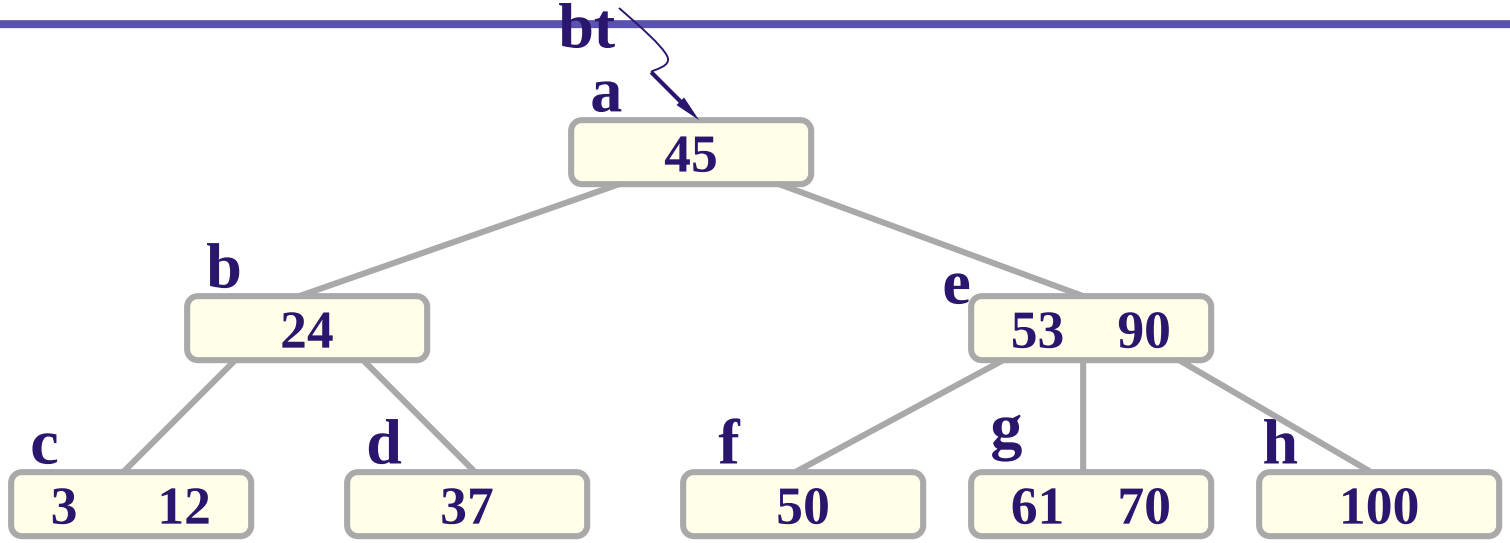
这意味着若m=199， n=1、999、998，则L至多等于4，而一次查找最多进行L次存取。因此，这个公式保证了B⁻树的查找效率是相当高的。

B-树插入

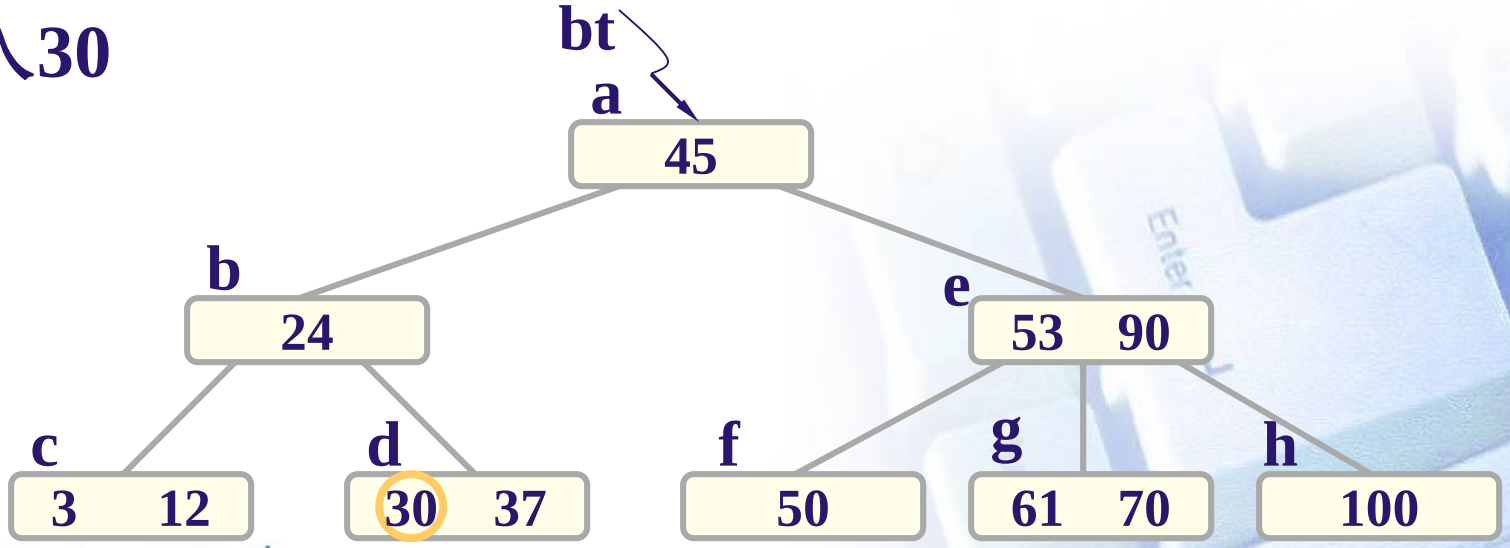
基本思想

- B-树结点中的关键字个数必须≥⌈m/2⌉-1
- 每次插入一个关键字是在最底层（L层）的某个非终端结点添加一个关键字，若该结点的关键字个数不超过m-1，则插入完成；否则，产生结点“分裂”。
- B-树是从空树开始，逐个插入关键字而生成的
- 例如，下图的3阶B-树，深度为4

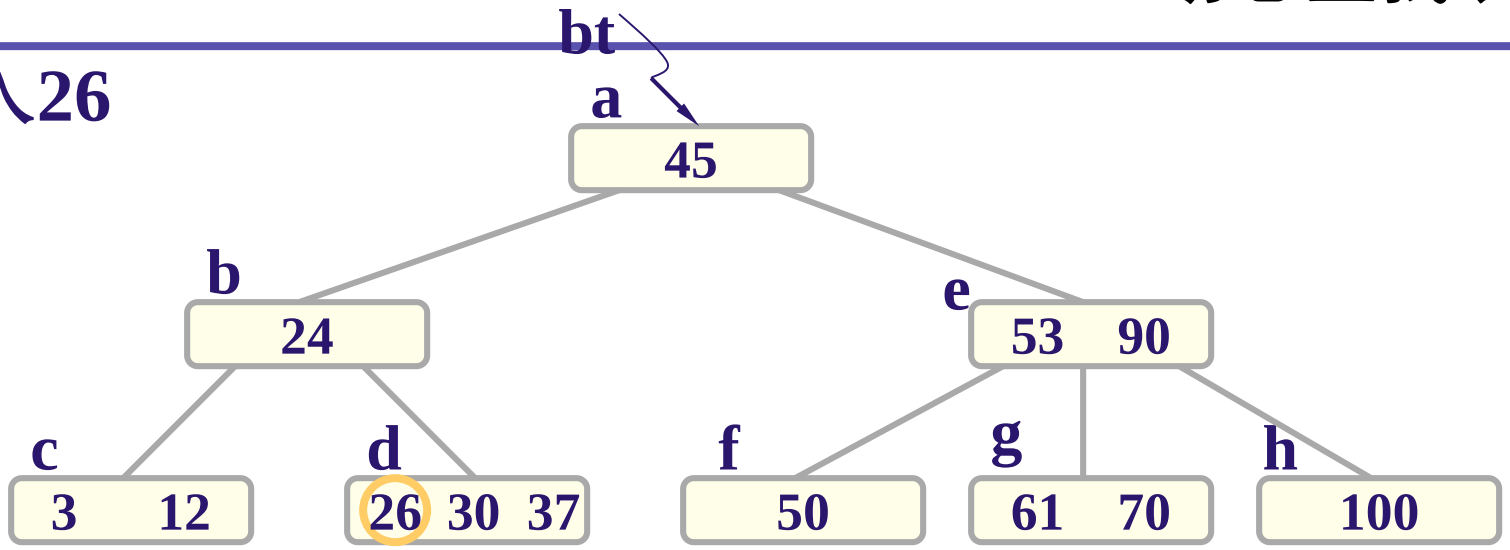
动态查找表


bt
a
45
b
24
e
53 90
c
3 12
d
37
f
50
g
61 70
h
100

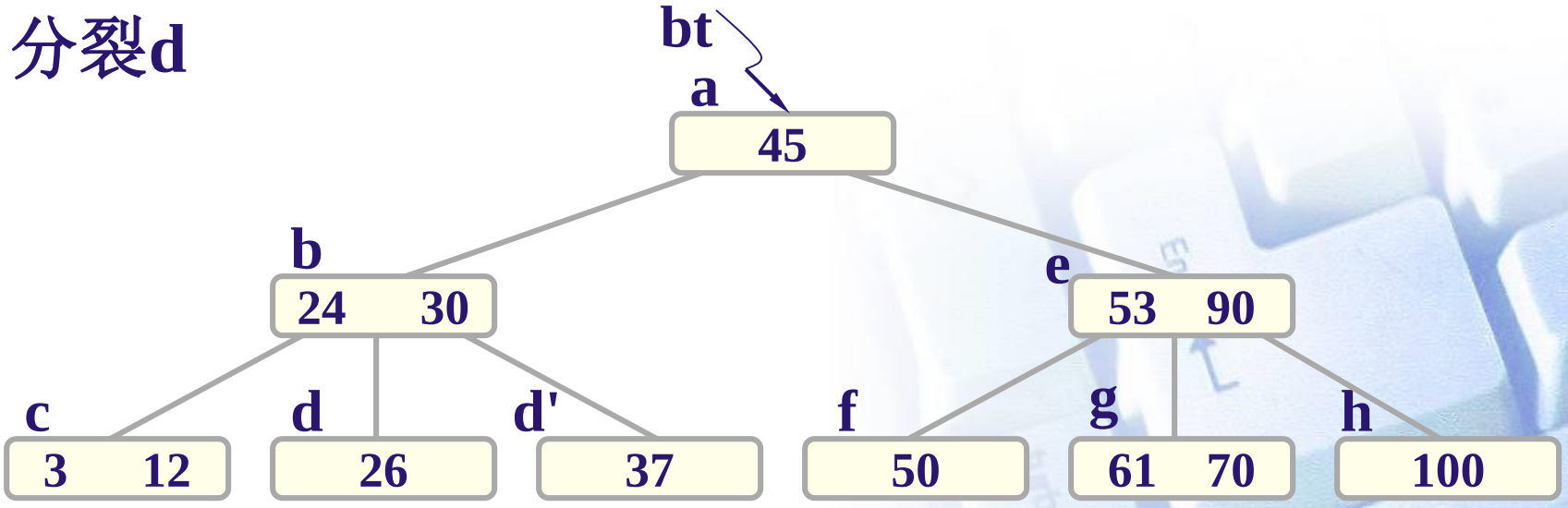
插入30


bt
a
45
b
24
e
53 90
c
3 12
d
30 37
f
50
g
61 70
h
100

插入26

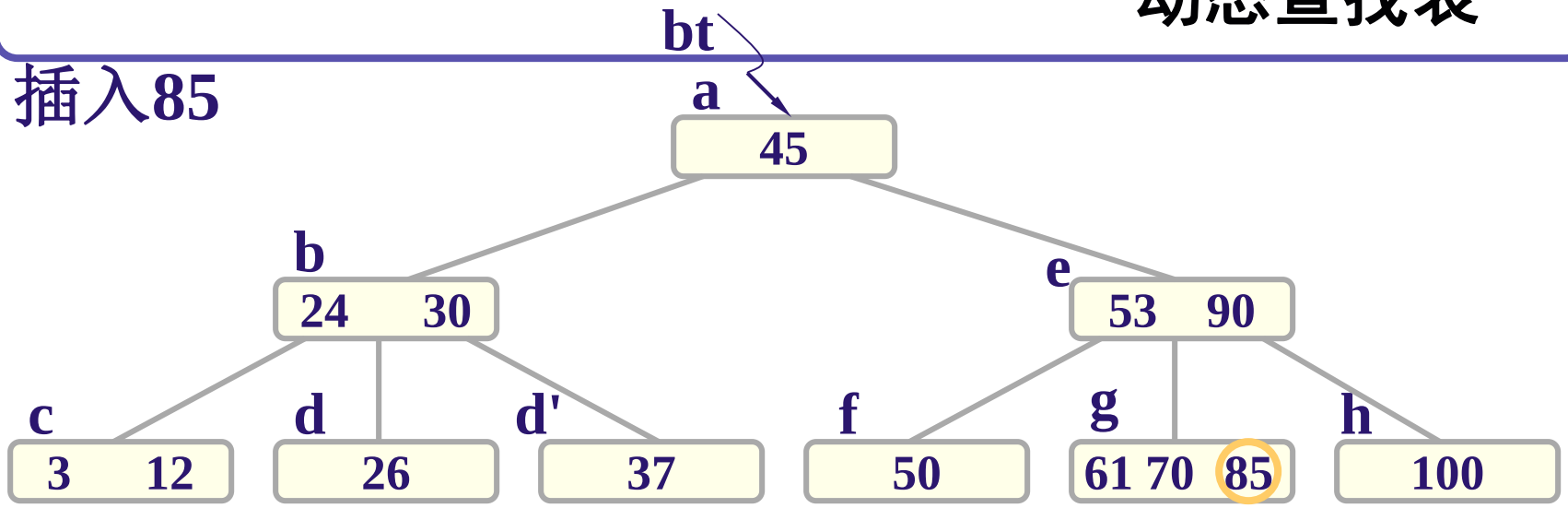

bt
a
45
b
24
e
53 90
c
3 12
d
26 30 37
f
50
g
61 70
h
100

分裂d

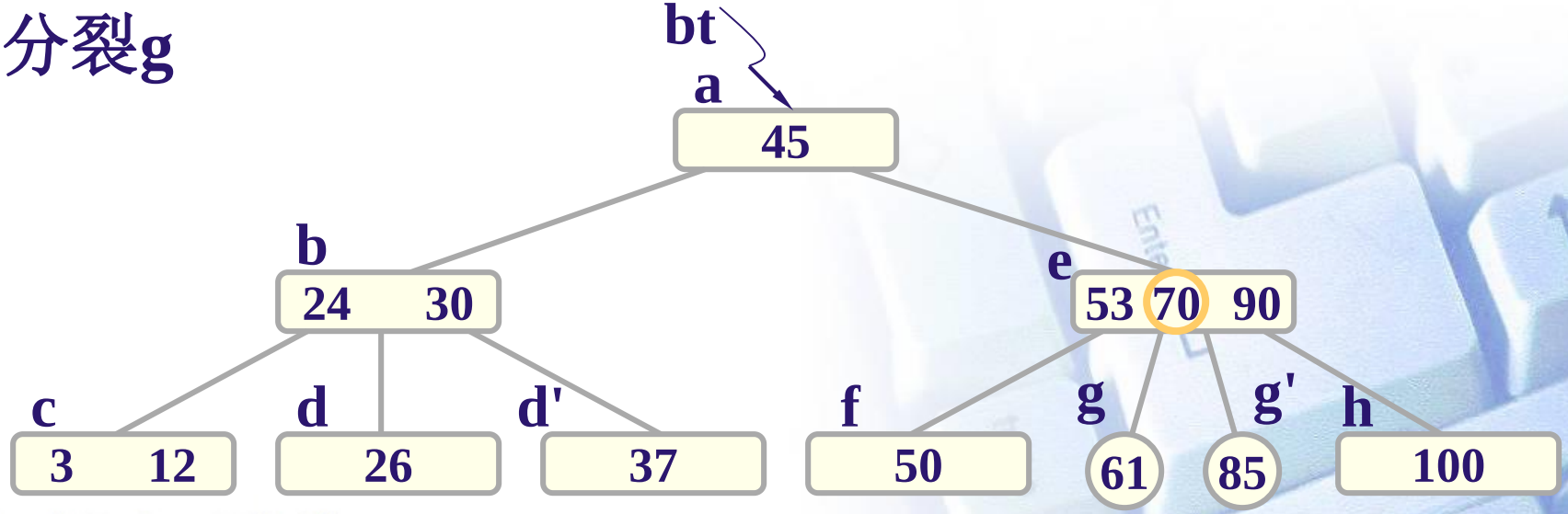

bt
a
45
b
24 30
e
53 90
c
3 12
d
26
d'
37
f
50
g
61 70
h
100

动态查找表

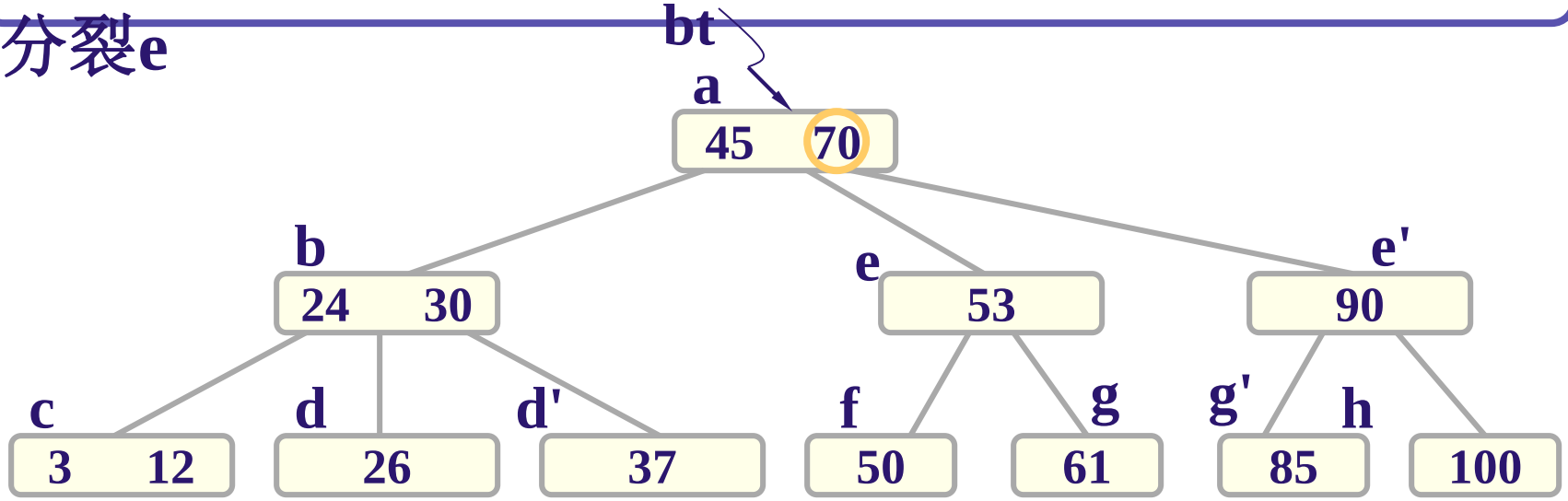
插入85



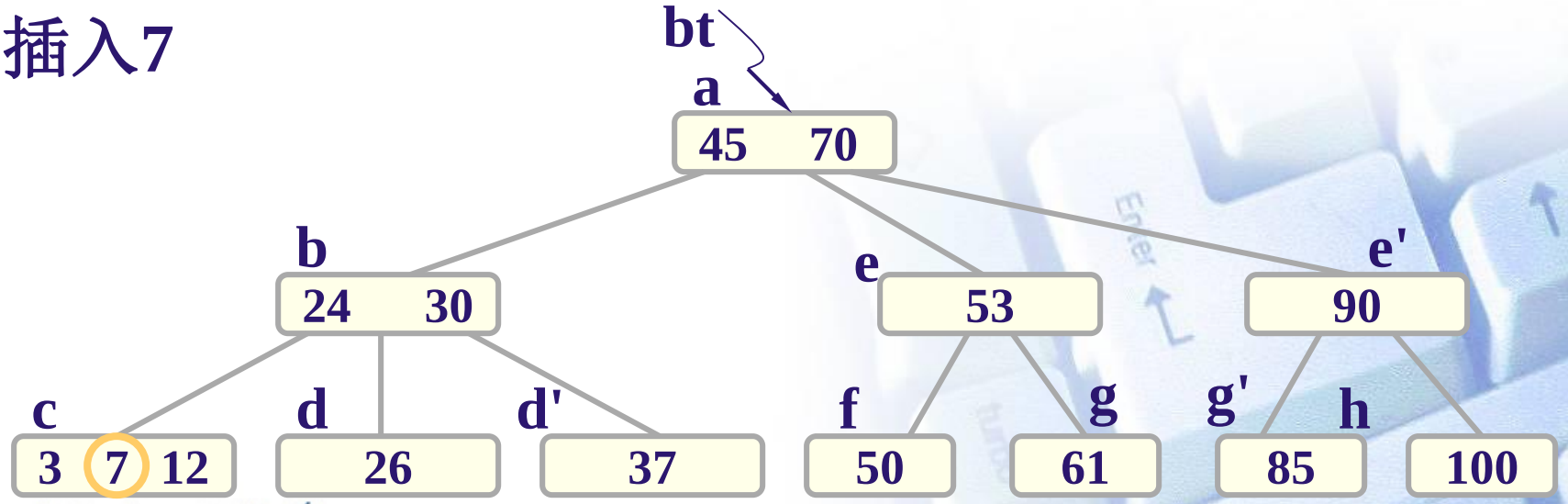
分裂g



分裂e

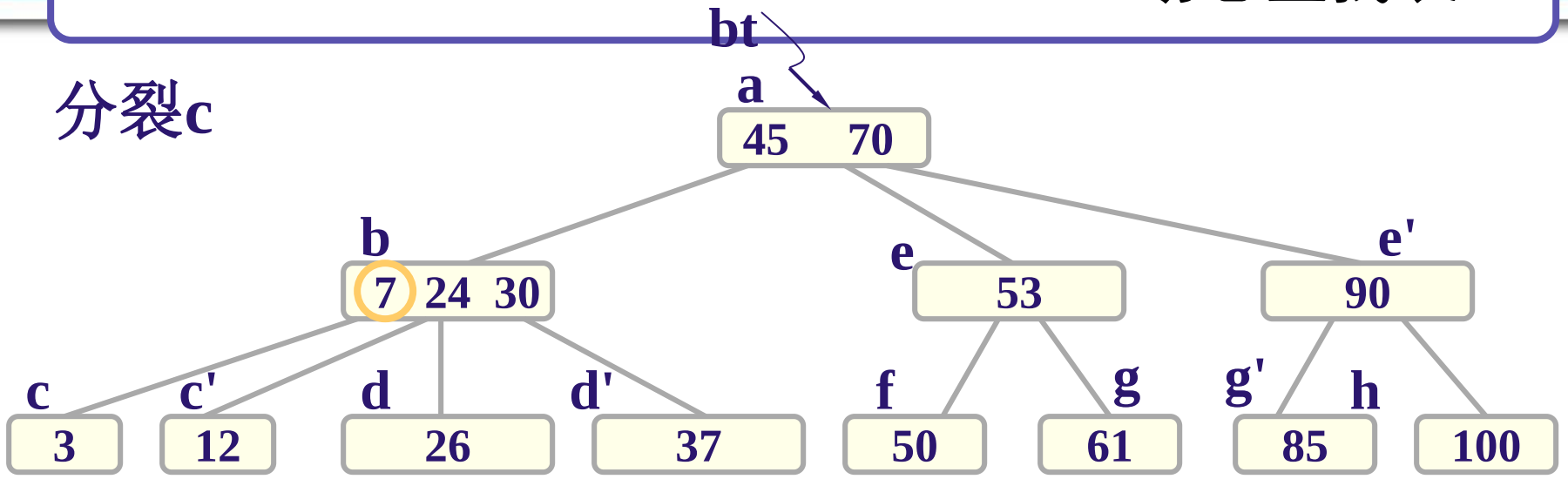

bt
a
45 70
b
24 30
e
53
e'
90
c
3 12
d
26
d'
37
f
50
g
61
g'
85
h
100

插入7

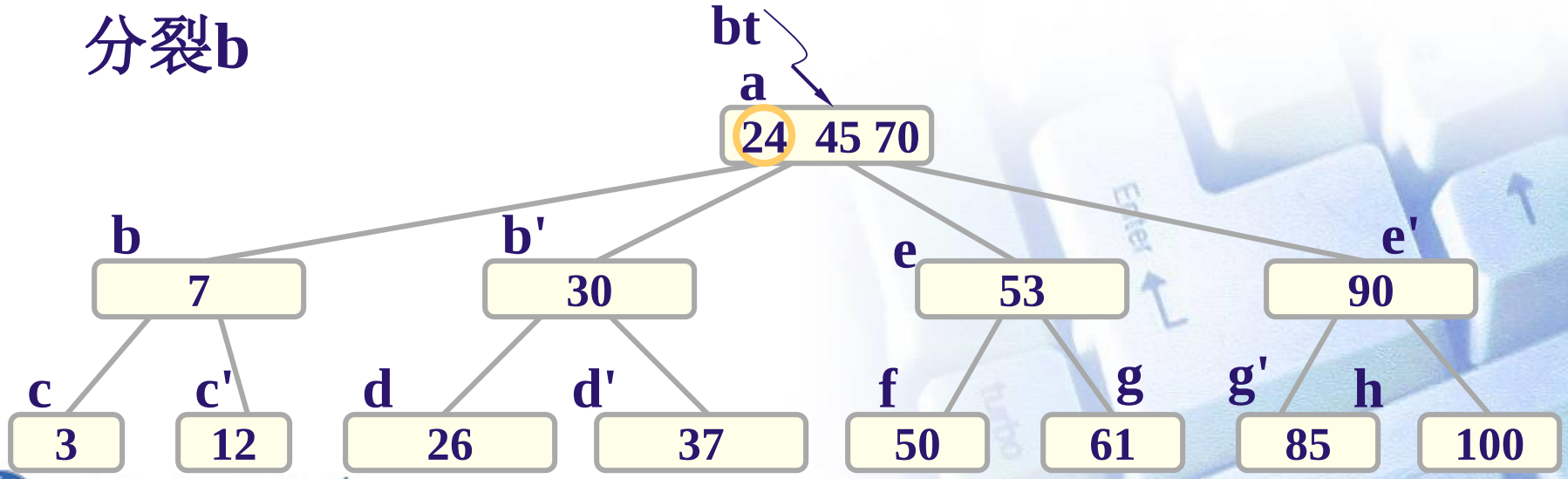

bt
a
45 70
b
24 30
e
53
e'
90
c
3 7 12
d
26
d'
37
f
50
g
61
g'
85
h
100

动态查找表

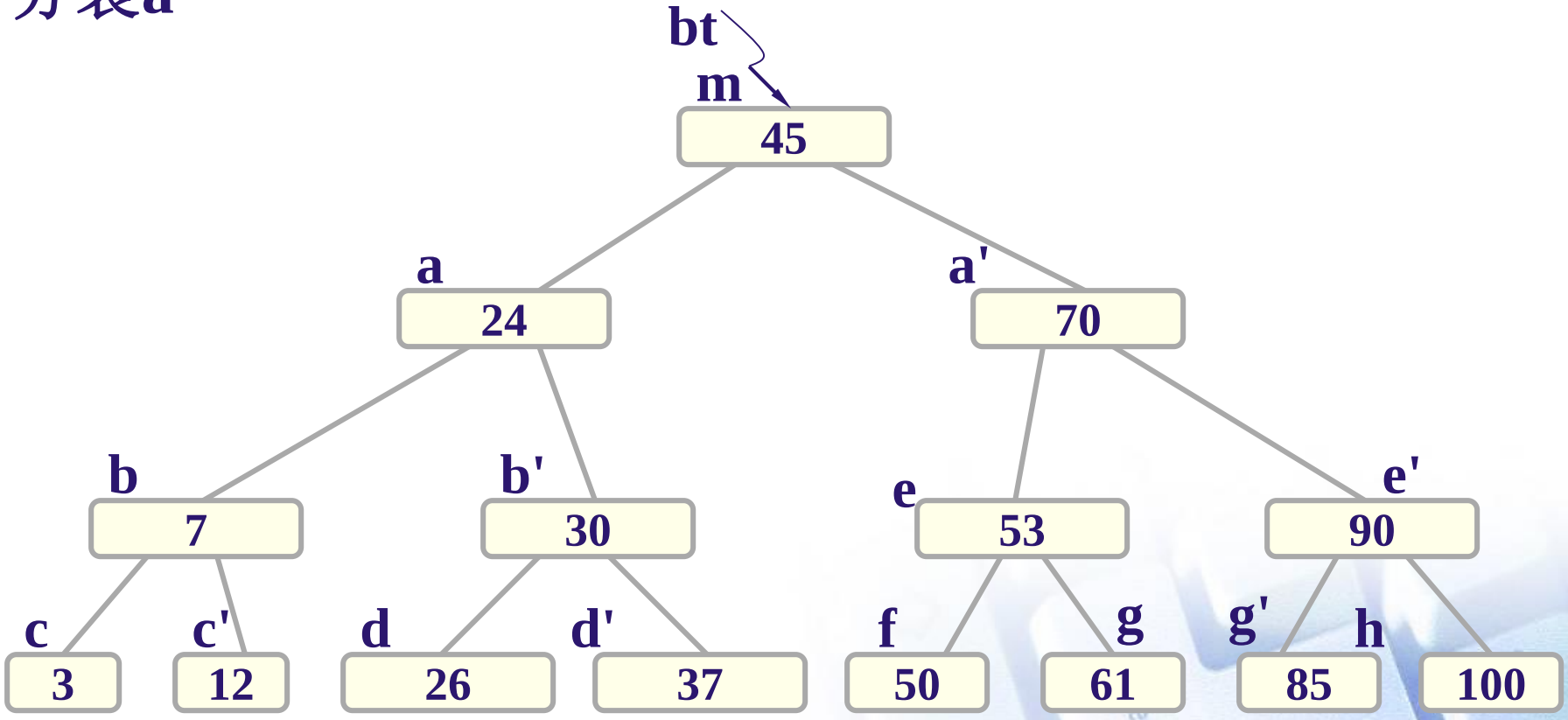
分裂c



分裂b



分裂a



- **B-树删除**

（1）找到该关键字所在结点

（2）若该结点为最下层非终端结点，且关键字数≥⌈m/2⌉，则删除之，转（4），否则，进行合并操作。

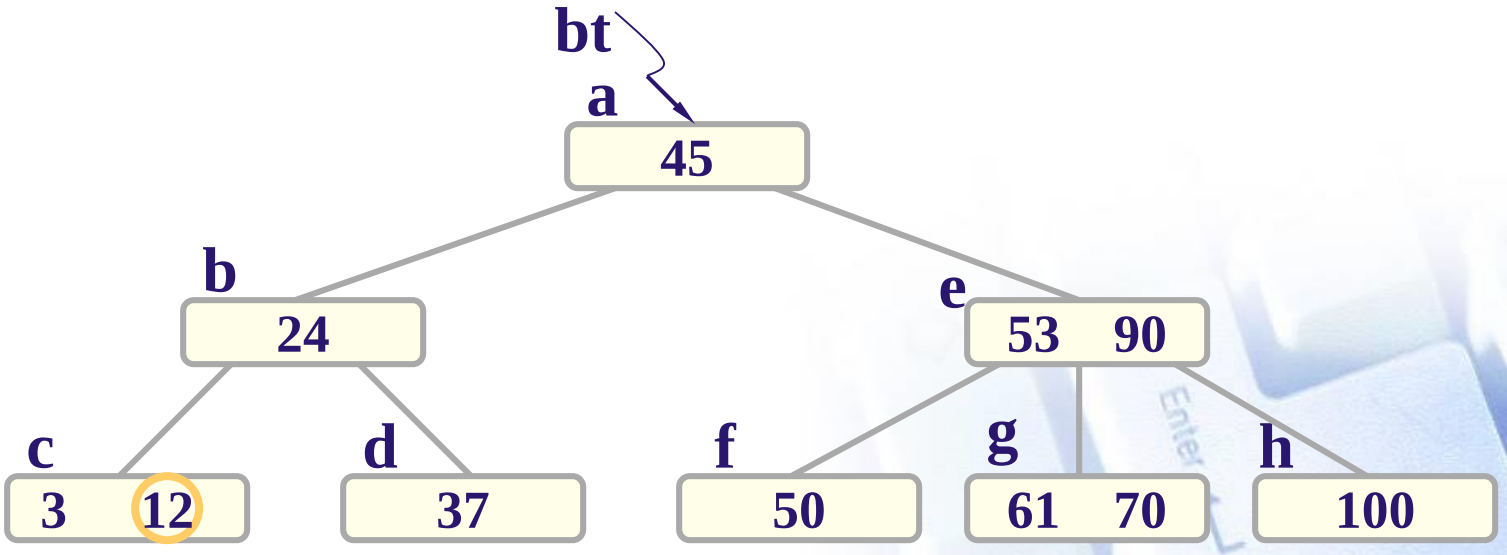
（3）若该结点为非终端结点中的Ki，则以指针Ai所指子树中的最小关键字Y代替Ki，然后在相应结点中删去Y。

（4）结束

最下层非终端结点中的关键字的情况，有下列3种：

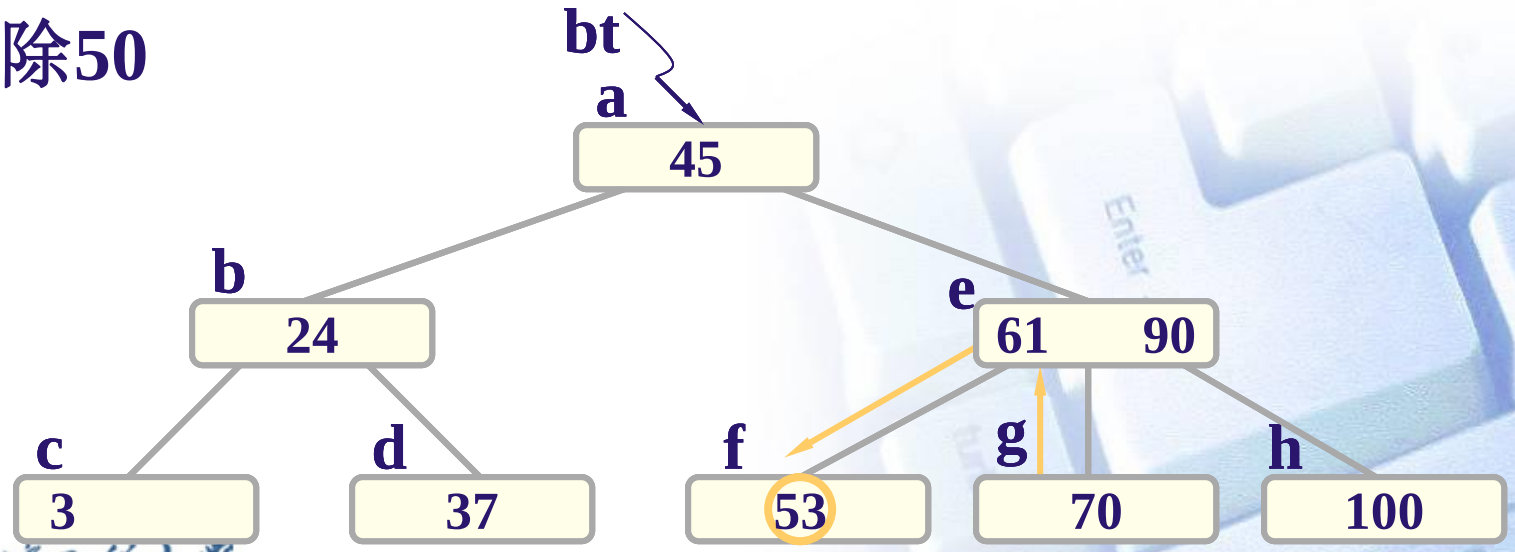
➢被删Ki所在结点中的关键字数 ≥⌈m/2⌉，删去该Ki和相应的指针Ai

删除12



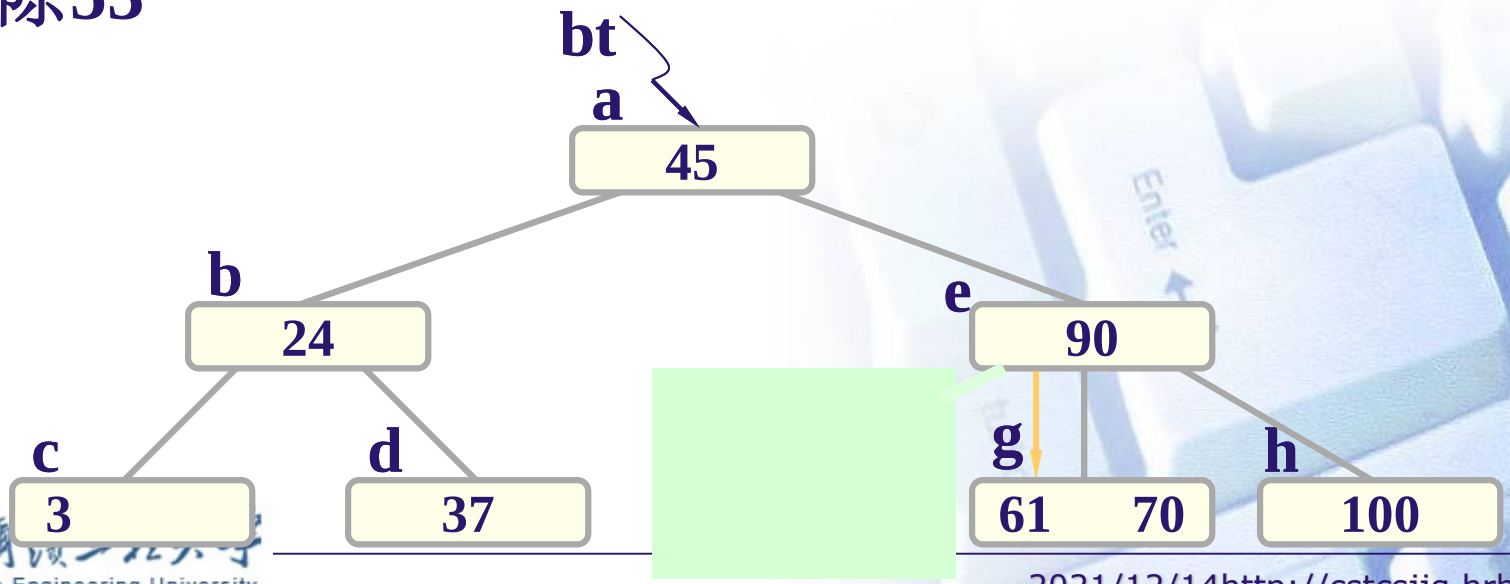
- 被删Ki所在结点中的关键字数=⌈m/2⌉-1，而与该结点相邻的右(或左)兄弟结点中的关键字数>⌈m/2⌉-1
  - 将其兄弟结点中最小（或最大）的关键字上移至双亲结点中
  - 将双亲结点中小于（或大于）且紧靠该上移关键字的关键字下移至被删Ki所在结点中

删除50

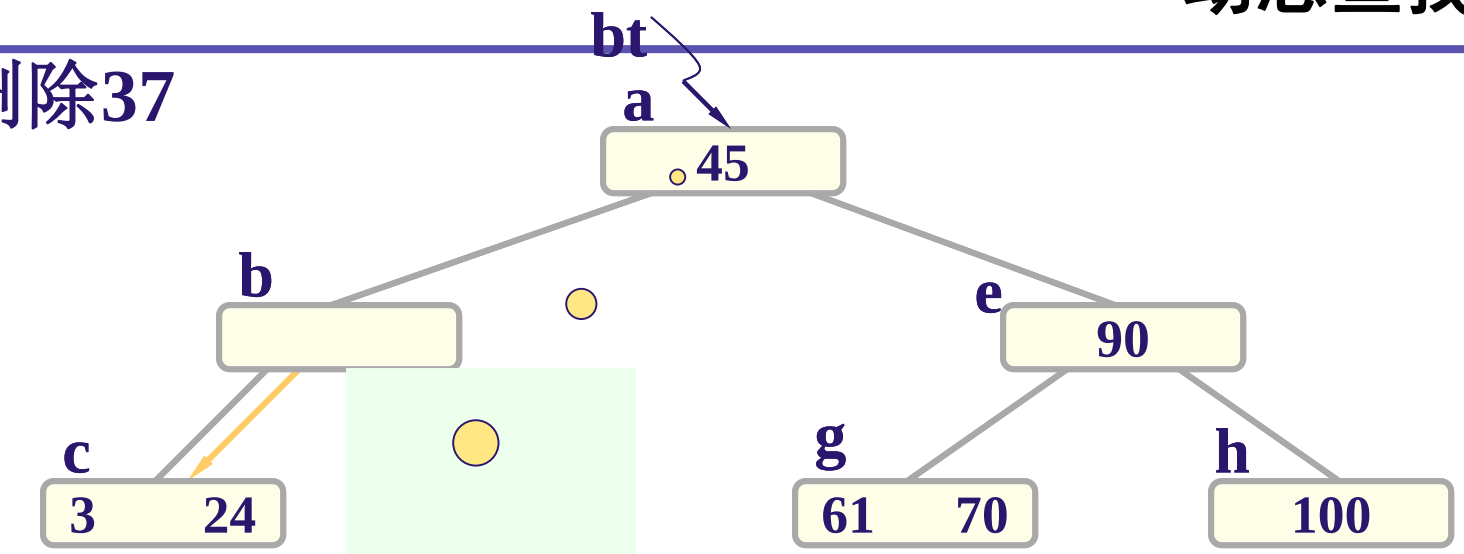


- 被删Kj所在结点和其相邻的兄弟结点中的关键字数均=⌈m/2⌉-1，若该结点有右兄弟，且其地址由双亲结点中的指针Ai所指，则删去Kj后，它所在结点中剩余的关键字和指针，加上双亲结点中的关键字Ki一起，合并到Ai所指的兄弟结点中（若无右兄弟，则合并到左兄弟中）。

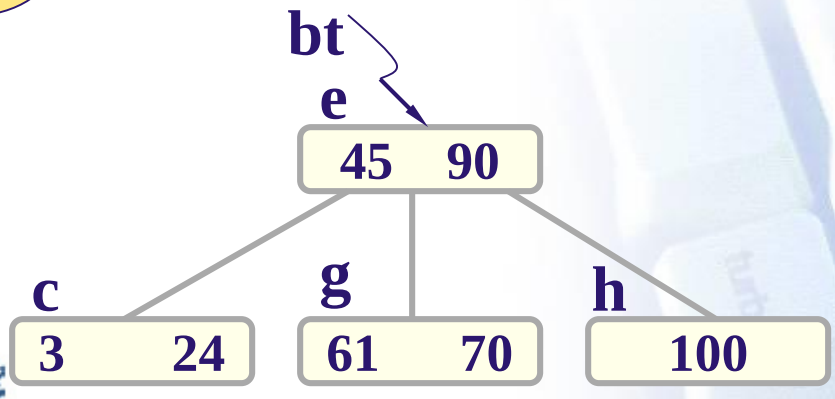
删除53



删除37



因b<⌈m/2⌉-1,
让其双亲Ki
与右兄弟合
并



◆B+树定义

B^+树是应文件系统所需而出的一种B^-树的变形树。一棵m阶的B^+树和m阶的B^-树的差异在于：

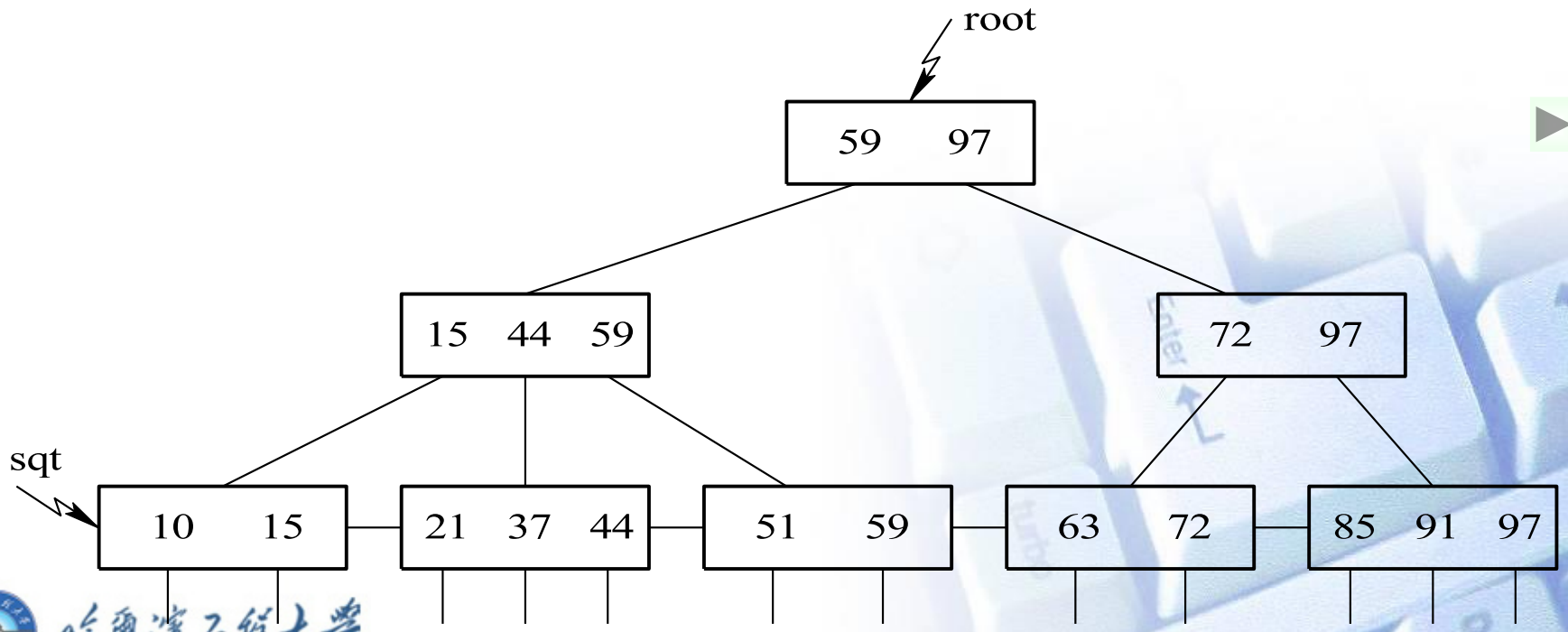
(1) 有n棵子树的结点中含有n个关键字；

(2) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接；

(3) 所有的非终端结点可以看成是索引部分，结点中仅含有其子树(根结点)中的最大(小)关键字。

动态查找表

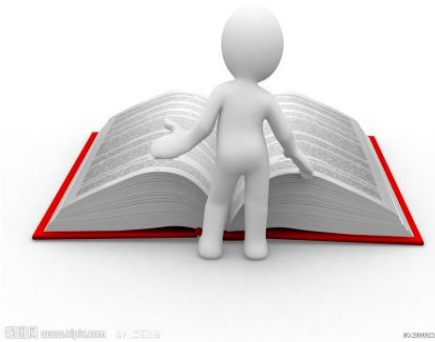
例如，图所示为一棵3阶B+树，通常在B+树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶子结点。因此，可以对B+树进行两种查找运算：一种是从最小关键字起顺序查找；另一种是从根结点开始，进行随机查找。



动态查找表

- 在B^+树上操作过程基本上与B^-树类似
- 查找时，若非终端结点上的关键字等于给定值，并不终止，而是继续向下直到叶子结点。
  - 不管查找成功与否，每次查找都是走了一条从根结点到叶子结点的路径。
  - 性能也等价于在关键字全集做一次二分查找；
- B+的特性：
  - 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；
  - 不可能在非叶子结点命中；
  - 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；
  - 更适合文件索引系统；

本章内容



1. 基本概念与术语
2. 静态查找表
3. 动态查找表
4. 哈希表
5. 本章小结

- 哈 希 表
  - 哈希函数构造
    - 处理冲突方法
      - 查找效率分析

哈希表

- 哈希表（散列表、杂凑表）
  - 基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；理想情况下，不经过比较，一次存取就能得到所查元素
  - 定义
    - 哈希函数（散列函数）——在记录的关键字与记录的存储地址之间建立的一种对应关系
      - ☞哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象
      - ☞哈希函数可写成：addr（a_i）=H（K_i）
        - a_i是表中的一个元素
        - Addr（a_i）是a_i的存储地址
        - K_i是a_i的关键字

哈希表

- 哈希表——应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表
- 哈希查找——又叫散列查找，利用哈希函数进行查找的过程

例　30个地区的各民族人口统计表

编号	地区名	总人口	汉族	回族.....
1	北京			
2	上海			
⋮	⋮			

以编号作关键字，构造哈希函数
H(key)=key
H(1)=1
H(2)=2

取地区名称第一个拼音字母的序号作哈希函数
H(Beijing)=2
H(Shanghai)=19
H(Shenyang)=19 冲突

从例子可见：

哈希函数只是一种映象，所以哈希函数的设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。

因为哈希的作用就是由表中记录的关键字来确定该记录在表中的地址。

- 哈希函数的构造方法
  - 直接定址法
    - 构造：取关键字或关键字的某个线性函数作哈希地址，即H(key)=key

      或　　H(key)=a·key+b
    - 特点
      - ☞所得地址集合与关键字集合大小相等，不会发生冲突
      - ☞实际中能用这种哈希函数的情况很少，其中a不能为零；

哈希表

- 数字分析法
  - 构造：对关键字进行分析，取关键字的若干位或其组合，作为哈希地址
  - 适于：关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况

例　有80个记录，关键字为8位十进制数，哈希地址为2位十进制数

①②	③	④	⑤	⑥	⑦	⑧
8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

分析：①只取8
②只取1
③只取3、4
⑧只取2、7、5
④⑤⑥⑦数字分布近乎随机
所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

哈希表

- 平方取中法
  - 构造：取关键字平方后，中间几位作哈希地址
  - 适于不知道全部关键字情况，存储整型数字
- 折叠法
  - 构造：将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）做哈希地址
  - 种类
    - ☞移位叠加：将分割后的几部分低位对齐相加
    - ☞间界叠加：从一端沿分割界来回折叠，然后对齐相加
  - 适于关键字位数很多，且每一位上数字分布大致均匀情况

例 关键字为 ：0442205864，哈希地址位数为4

```
  5 8 6 4
  4 2 2 0
      0 4
-----------
  1 0 0 8 8
```

移位叠加

H(key)=0088

```
  5 8 6 4
  0 2
```

一般情况下，可以选p为质数，若给定表长度m=12，则p应选11

- 除留余数法
  - 构造：取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，即

$$H(key)=key \ \ MOD \ \ p，\ p\le m$$

  - 特点
    - 简单、常用，可与上述几种方法结合使用
    - p的选取很重要；p选的不好，容易产生同义词----地址冲突

哈希表

- 随机数法
  - 构造：取关键字的随机函数值作哈希地址，即H(key)=random(key)
  - 适于关键字长度不等的情况
- 选取哈希函数考虑以下因素：
  - 计算哈希函数所需时间
  - 关键字长度
  - 哈希表长度（哈希地址范围）
  - 关键字分布情况
  - 记录的查找频率

哈希表

- 处理冲突的方法
  - 开放定址法
    - 方法：当冲突发生时，形成一个探查序列；沿此序列逐个地址探查，直到找到一个空位置（开放的地址），将发生冲突的记录放到该地址中，即

      $Hi=(H(key)+di)MOD\ m,\ i=1,2,\ldots\ldots k(k\leq m-1)$

      其中：H(key)——哈希函数

      m——哈希表表长，不是P

      di——增量序列
    - 分类
      - ☞线性探测再散列：di=1,2,3,……m-1
      - ☞二次探测再散列：$di=1^2,-1^2,2^2,-2^2,3^2,\ldots\ldots\pm k^2(k\leq m/2)$
      - ☞伪随机探测再散列：di=伪随机数序列

例：表长为11的哈希表中已填有关键字为17，60，29的记录，H(key)=key MOD 11，现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中。

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

(1) H(38)=38 MOD 11=5 冲突
H_1=(5+1) MOD 11=6 冲突
H_2=(5+2) MOD 11=7 冲突
H_3=(5+3) MOD 11=8 不冲突

(2) H(38)=38 MOD 11=5 冲突
H_1=(5+1²) MOD 11=6 冲突
H_2=(5-1²) MOD 11=4 不冲突

(3) H(38)=38 MOD 11=5 冲突
设伪随机数序列为9，…，则：
H_1=(5+9) MOD 11=3 不冲突

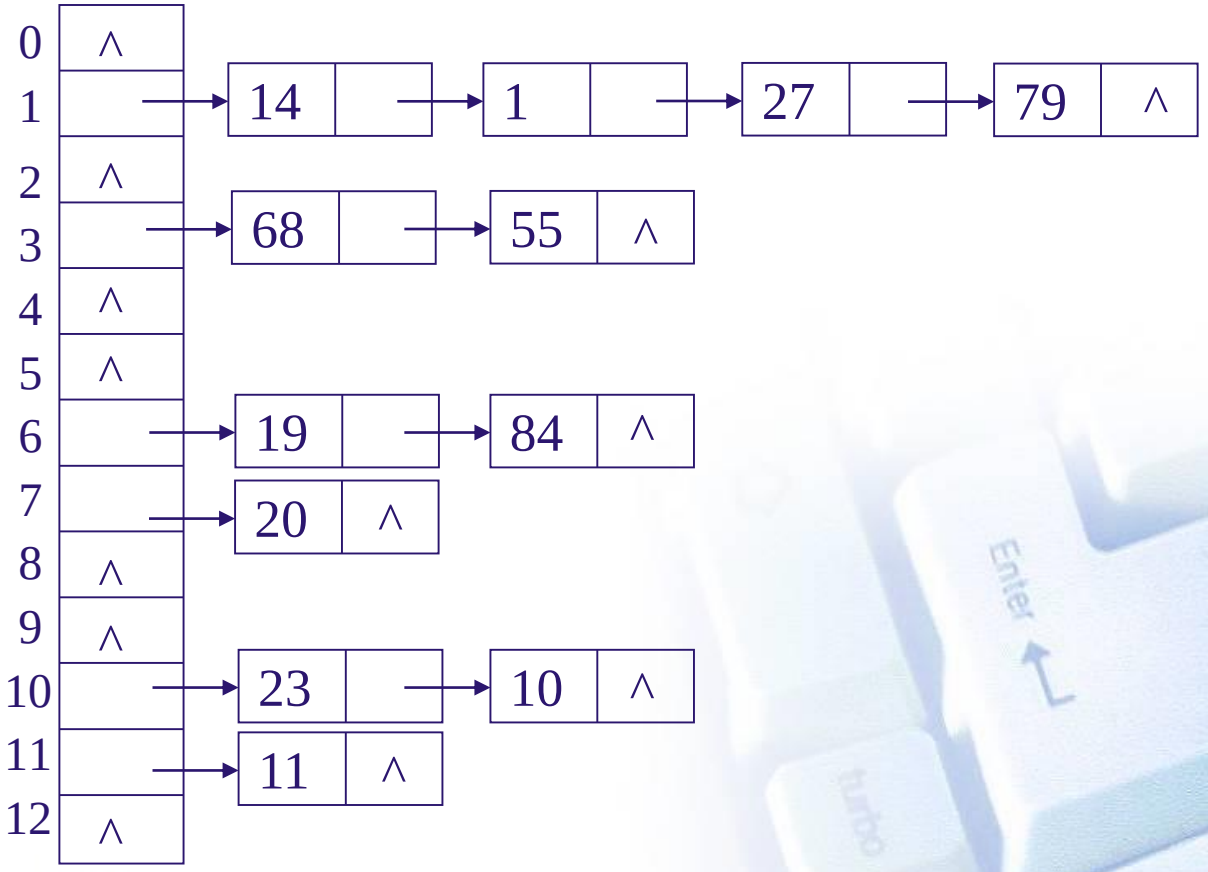
- 再哈希法
  - 方法：构造若干个哈希函数，当发生冲突时，计算下一个哈希地址，即：
    $H_i=Rh_i(key)$　$i=1,2,\ldots\ldots k$

    其中：Rh_i——不同的哈希函数
  - 特点：计算时间增加
- 链地址法
  - 方法：将所有关键字为同义词的记录存储在一个单链表中，并用一维数组存放头指针

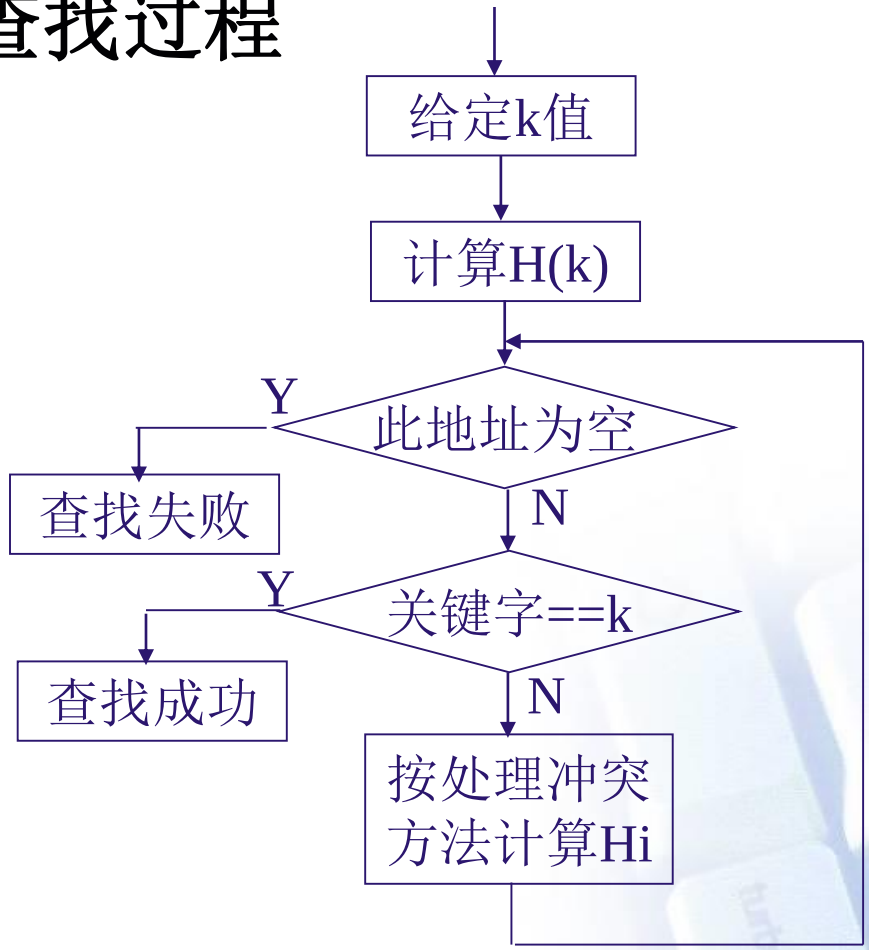
例 已知一组关键字

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

哈希函数为：H(key)=key MOD 13,用链地址法处理冲突



- 哈希查找过程及分析
  - 哈希查找过程



哈希表

- 哈希查找分析
  - 哈希查找过程仍是一个给定值与关键字进行比较的过程，但是比较次数有限
  - 评价哈希查找效率仍要用ASL
  - 哈希查找过程与给定值进行比较的关键字的个数取决于：
    - ☞哈希函数
    - ☞处理冲突的方法
    - ☞哈希表的装填因子α=表中填入的记录数/哈希表长度，即：表长度=记录数个数/装填因子

哈希表

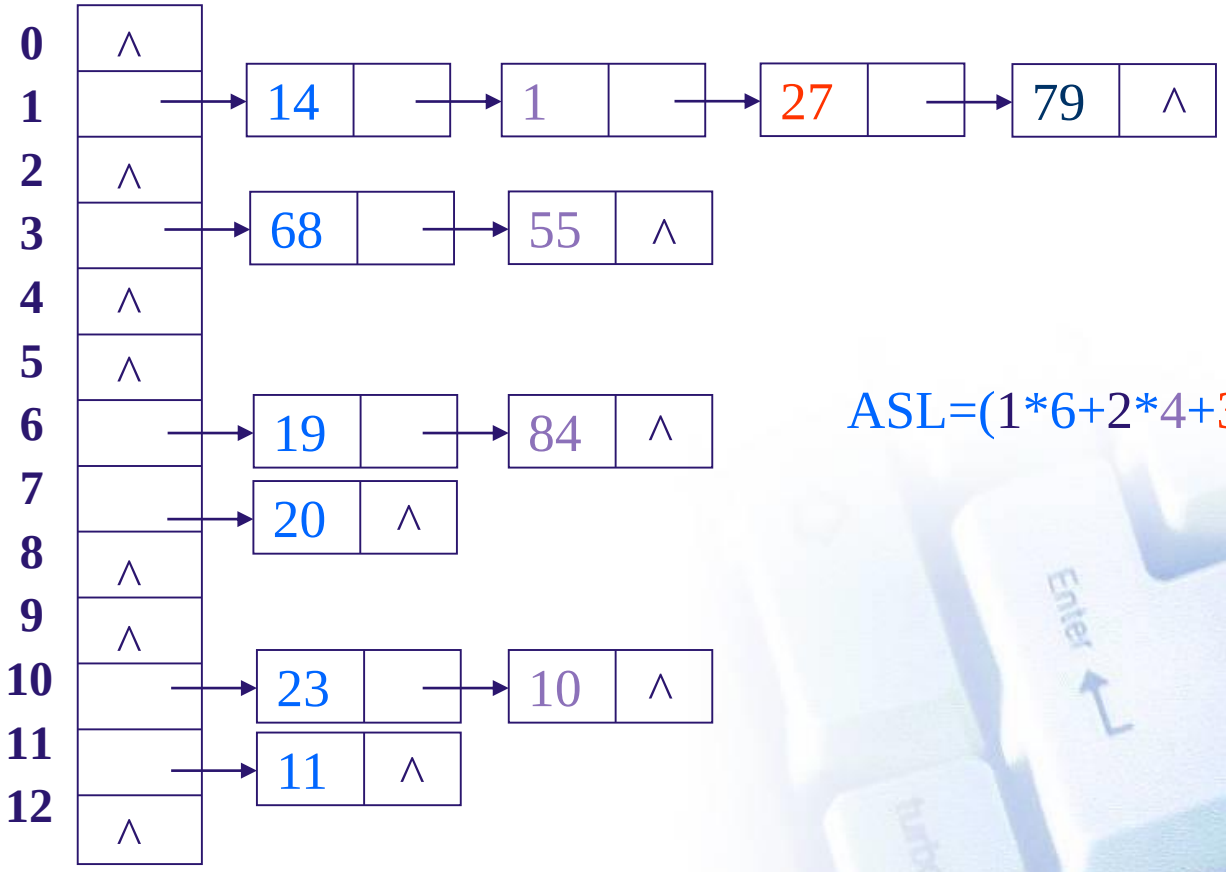
例　已知一组关键字(19,14,23,1,68,20,84,27,55,11,10,79)
哈希函数为：H(key)=key MOD 13, 哈希表长为m=16，
设每个记录的查找概率相等
(1) 用线性探测再散列处理冲突，即Hi=(H(key)+di) MOD m

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

H(19)=6
H(14)=1
H(23)=10
H(1)=1　冲突，H1=(1+1) MOD16=2
H(68)=3
H(20)=7
H(84)=6　冲突，H1=(6+1)MOD16=7
　　冲突，H2=(6+2)MOD16=8
H(27)=1　冲突，H1=(1+1)MOD16=2
　　冲突，H2=(1+2)MOD16=3
　　冲突，H3=(1+3)MOD16=4
H(55)=3　冲突，H1=(3+1)MOD16=4
　　冲突，H2=(3+2)MOD16=5
H(11)=11
H(10)=10　冲突，H1=(10+1)MOD16=11
　　冲突，H2=(10+2)MOD16=12
H(79)=1　冲突，H1=(1+1)MOD16=2
　　冲突，H2=(1+2)MOD16=3
　　冲突，H3=(1+3)MOD16=4
　　冲突，H4=(1+4)MOD16=5
　　冲突，H5=(1+5)MOD16=6
　　冲突，H6=(1+6)MOD16=7
　　冲突，H7=(1+7)MOD16=8
　　冲突，H8=(1+8)MOD16=9
ASL=(1*6+2+3*3+4+9)/12=2.5

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

(2) 用链地址法处理冲突



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

- 哈希查找算法实现
  - 用线性探测再散列法处理冲突
    - ☞实现
      - 查找过程：同前，利用动态生成过程
      - 删除：只能作标记，不能真正删除
      - 插入：遇到空位置或有删除标记的位置就可以插入

本章小结

1、本章讨论查找表的各种表示方法以及查找效率的衡量标准-平均查找长度。

2、在本章中介绍了查找表的三类存储表示方法：顺序表、树表和哈希表。

这里的顺序表指的是顺序存储结构，包括有序表和索引顺序表，因此主要用于表示静态查找表，树表包括静态查找树、二叉（排序）查找树和二叉平衡树，树表和哈希表主要用于表示动态查找表。

3、重点掌握查找表的ASL、二叉排序树构造及查找方法、平衡二叉树的构造方法。

4、重点掌握哈希表的构造及冲突处理方法。

下课休息一会！