字 典 与 检 索

基本概念

- 字典：字典是元素的有穷集合，其中每个元素由两部分组成，分别称为元素的“关键码”和“属性”
  - 例：英汉字典，其中的每个词条是一个元素，被解释的英文单词可以看作是该元素的关键码，词条中对单词的解释是元素的属性
- 特点：
  - 字典中的两个元素能够根据其关键码进行比较，对字典元素的存取、检索也是以关键码为依据进行的
  - 在字典中最主要的运算是检索

基本概念(续)

- 检索：给定一个值key，在字典中找出关键码等于key的元素
  - 如果找到，则检索成功
  - 否则检索失败
  - 为了便于字典的维护，有时也要考虑在字典中插入和删除元素的操作
- 字典的种类
  - **静态字典**：字典一经建立就基本固定不变，主要的操作就是字典元素的检索
    - 为静态字典选择存储方法主要需考虑**检索效率**及**检索运算的简单性**
  - **动态字典**：经常需要改动的字典
    - 对于动态字典，存储方法的选择不仅要考虑**检索效率**，还要考虑字典元素的**插入、删除运算是否简便**

基本概念(续)

- 衡量一个检索算法效率的主要标准
  - 检索过程中和关键码的平均比较次数，即平均检索长度ASL(Average Search Length)，定义为：

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

    - n是字典中元素的个数
    - p_i是查找第i个元素的概率，若不特别声明，一般认为每个元素的检索概率相等，即$p_i=1/n$，c_i是找到第i个元素的比较次数
  - 算法的空间开销，以及算法是否易于理解等因素

基本概念(续)

- 假设:
  - 字典元素类型相同，且关键码为数值类型(例如正整数)，因此，可以将字典元素按关键码排序

顺序表示

- 字典的顺序表示定义如下：

```
typedef  int          KeyType;
typedef  int          DataType;
typedef struct
{  KeyType key;                /* 字典的关键码字段*/
   DataType other;             /* 字典的属性字段*/
}DicElement;
typedef struct
{  DicElement element[MAXNUM];
   int n;      /* n<MAXNUM，为字典中元素的个数 */
} SeqDictionary;
```

*为了描述简单，将KeyType和DataType定义为int类型

顺序表示(续)

- 顺序检索
  - 基本思想
  - 从字典的一端开始顺序扫描，将字典中元素的关键码和给定值比较，如果相等，则检索成功;
  - 当扫描结束时，还未找到关键码等于给定值的元素，则检索失败

顺序表示(续)

- 顺序检索算法
- 算法结束时返回检索成功或失败信息
  - 若检索成功，则参数position指向找到的元素位置
  - 否则，position指向应插入元素的位置

- 顺序检索优点:

  算法简单且适应面广。无论字典中元素是否有序均可使用
- 顺序检索缺点:

  平均检索长度较大，特别是当n很大时，检索效率较低

顺序表示(续)

- 算法分析
  - 若找到的是第一个元素element[0]，则比较次数$c_1=1$;
  - 若找到的是第i个元素element[i-1]，则比较次数为$c_i=i$；因此，顺序检索的平均检索长度为

$ASL=1\times P_1+2\times P_2+\ldots+n\times P_n$

  - 假设每个元素的检索概率相等，即$P_i=1/n$，则平均检索长度为：

$$ASL=\sum_{i=1}^{n}p_ic_i=\sum_{i=1}^{n}i/n=(n+1)/2$$

  - 因此，成功检索的平均比较次数约为字典长度的一半，若字典中不存在关键码为key的元素，则需进行n次比较。顺序检索的平均检索时间为ASL=O(n)

顺序表示(续)

- 二分法检索(二分检索)-是一种效率较高的检索方法
  - 要求：字典元素按关键码排序
  - 基本思想：
    - 设字典有序地存放在数组中
      - ✓ 首先将字典中间位置上元素的关键码和给定值key比较，如果相等，则检索成功;
      - ✓ 否则，若大于key，则在字典前半部分中继续进行二分法检索，否则在字典后半部分中继续进行二分法检索
  - 经过一次比较就缩小一半的查找区间，如此进行下去，直到检索成功或检索失败

顺序表示(续)

例题：被检索字典中元素的关键码序列为：

05，10，18，25，27，32，41，51，68，73，99

检索关键码为25和78的元素

- 检索关键码为25的元素

[05 10 18 25 27 32 41 51 68 73 99]

↑

[05 10 18 25 27] 32 41 51 68 73 99

↑

05 10 18 [25 27] 32 41 51 68 73 99

↑

经过三次比较后检索成功

顺序表示(续)

- 检索关键码为78的元素

[05 10 18 25 27 32 41 51 68 73 99]

↑

05 10 18 25 27 32 [41 51 68 73 99]

↑

05 10 18 25 27 32 41 51 68 [73 99]

↑

05 10 18 25 27 32 41 51 68 73 [99]

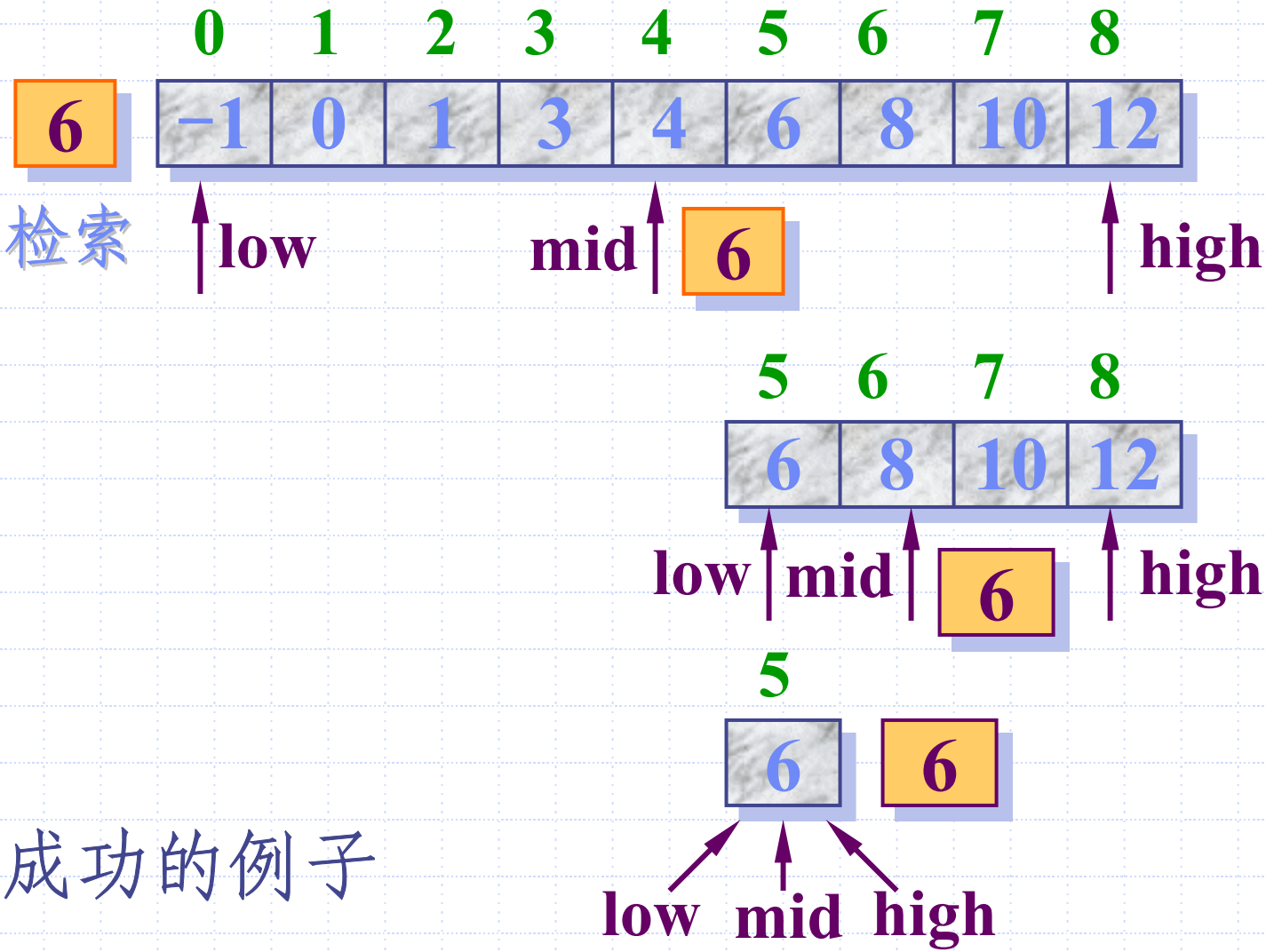
↑

- 经过四次比较后检索失败

基于有序顺序表的二分检索

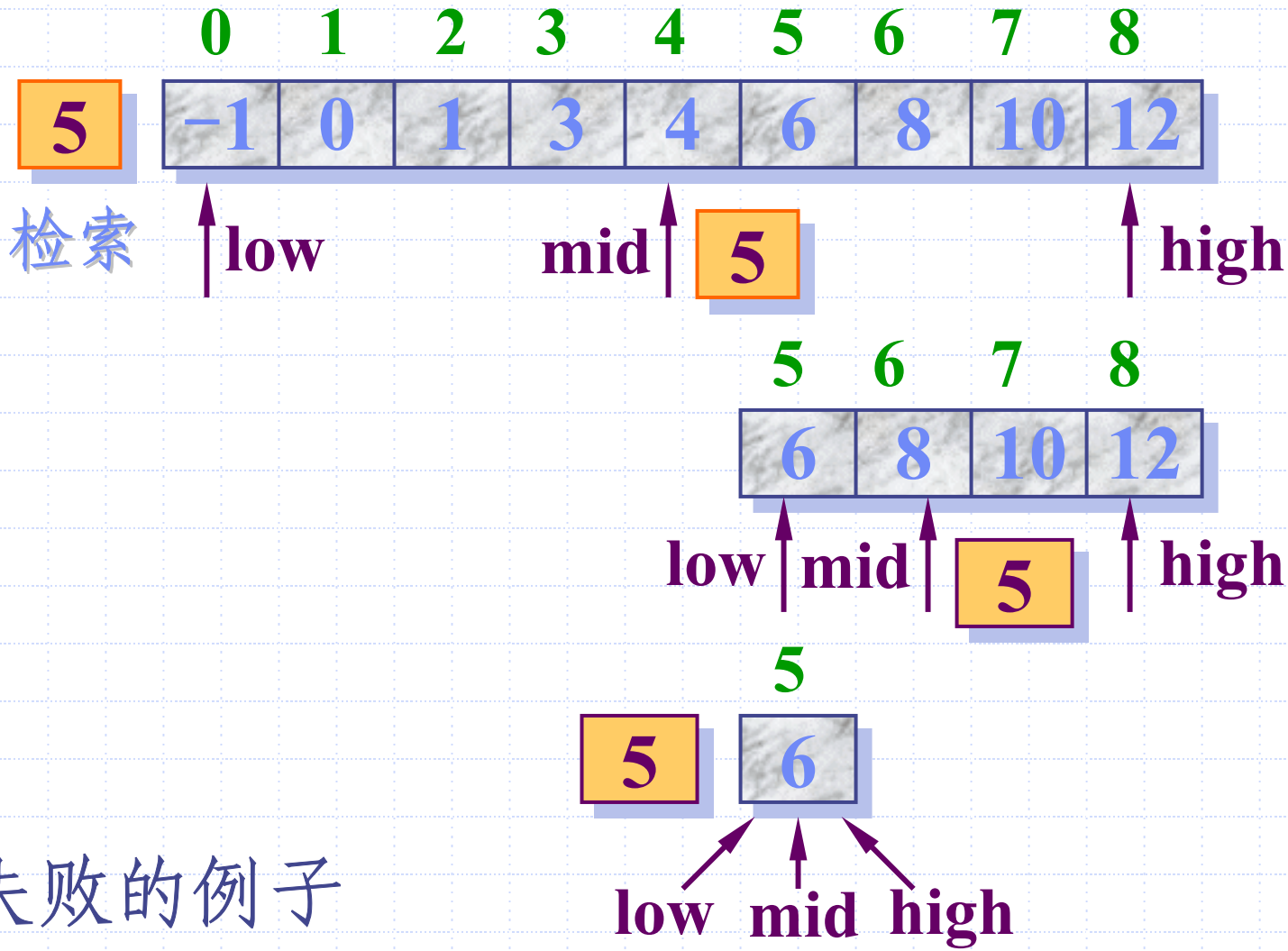
- 设*n*个对象存放在一个有序顺序表中，并按其关键码小到大排好了序。
- 二分检索时，先求位于检索区间正中的对象的下标用其关键码与给定值x比较:
  - Element[mid].key == x，检索成功;
  - Element[mid].key > x，把检索区间缩小到表的前半部分，继续二分检索;
  - Element[mid].key < x，把检索区间缩小到表的后半部分，继续二分检索。
- 如果检索区间已缩小到一个对象，仍未找到想要检索的对象，则检索失败。

基于有序顺序表的二分检索



检索成功的例子

基于有序顺序表的二分检索



检索失败的例子

顺序表示(续)

- 二分法检索的算法
  - 分别用low和high表示当前查找区间的下界和上界
- 二分法检索的优点

  比较次数少，检索速度快
- 二分法检索缺点

  要将字典按关键码排序，且只适用于顺序存储结构

顺序表示(续)

- 算法分析
  - 二分法检索每经过一次比较就将检索范围缩小一半
  - 二分法检索的最大检索长度为$\lceil \log_2(n+1) \rceil$
    ($\lceil X \rceil$表示取不小于x的最大整数)

散列表示

- 最大优点是，执行元素的插入、删除、检索运算平均仅要求常数的时间

基本概念

- 散列表：用散列法表示的字典称为“散列表”，散列法又称为“杂凑法”或“关键码—地址转换法”
- 散列法：
  - 设给定一个字典元素，其关键码为key;
  - 将key看成自变量，按一个确定的散列函数h计算出h(key);
  - 根据该值再进行字典元素的插入和删除操作

基本概念(续)

- 碰撞：如果两个不相等的关键码key1和key2，用散列函数h得到相同的散列地址(即h(key1)=h(key2))，则这种现象称为“碰撞”
  - 发生碰撞的两个(或多个)关键码称为同义词(相对于函数h而言)，发生碰撞后要用另一个函数I再进行计算I(h(key))，如此反复直至不发生碰撞
  - 要提高处理效率就应尽量减少碰撞
- 基本区域：h(key)的值域所对应的地址空间称为基本区域
- 发生碰撞时，同义词可以存放在基本区域中未被占用的单元，也可以放到基本区域以外另开辟的区域(称溢出区)

基本概念(续)

- 负载因子:

$$\alpha = \frac{\text{散列表中结点数目}}{\text{基本区域能容纳的结点数}}$$

  - 当 $\alpha > 1$ 时碰撞不可避免
  - 负载因子的大小要取得适当，既要减少碰撞，又不浪费存贮空间

基本概念(续)

- 散列表建表过程
  - 将key看成自变量，按一个确定的散列函数h计算出h(key)，如果以该值为地址的存贮空间未被占用，则将元素存入该单元;
  - 如果存贮单元已经存放了其它的元素，则用另一函数I计算出I(h(key))，若以该值为地址的单元也被占用，则再用I函数计算，直到找到空的单元将元素存入为止

基本概念(续)

- 散列表检索过程
  - 首先计算h(key)，以该值为地址在地址空间中查找，如果该地址空间未被占用，则说明检索失败;
  - 否则用该元素的关键码与要查找的key比较，如果相等则检索成功，否则用函数I计算出I(h(key));
  - 如此反复直到求出的某地址空间未被占用(检索失败)或者地址中存放的元素的关键码与key相等(检索成功)为止

散列函数

- 要提高检索的效率，散列函数的选取非常关键
- 选取散列函数的原则：
  - 根据字典元素关键码集合的特性选取散列函数，使计算出的地址尽可能均匀地分布在地址空间中
  - 为了提高关键码到地址的转换速度，散列函数应该尽量简单

散列函数(续)

- 常用的散列函数

  - 数字分析法
    - 方法：对关键码的各位进行分析，丢掉分布不均匀的位留下均匀的位作为地址。（关键码位数比存贮区的地址码位数多）
    - 缺点：散列函数依赖于关键码集合

散列函数(续)

- 例:对下列关键码集合进行关键码到地址的转换。关键码是9位的，地址是3位的，需要经过数字分析丢掉6位。

key	h(key)
000319426	326
000718309	709
000629443	643
000758615	715
000919697	997
000310329	329

- 分析这6个关键码，前三位都是0，不均匀，应该丢掉；第五位4个1，不均匀，第六、七位也不太均匀，都应该丢掉，因此留下第四、八、九位作为地址

散列函数(续)

除余法

- 方法：选择一个适当的正整数P，用P去除关键码，余数作为散列地址，即h(key)=key%P
  - P的取值：关键是选取适当的P
  - 如果P为偶数，则总是把奇数的关键码转换为奇数地址，把偶数的关键码转换为偶数地址
  - 如果P是关键码的基数的幂次，则相当于选择关键码的最后几位数字作为地址，这两种情况都不好
  - 一般P为小于基本区域长度m的最大素数比较好

散列函数(续)

- 例如：

  m=8,16,32,64,128,256,512,1024

  p=7,13,31,61,127,251,503,1019

- 除余法地址计算公式非常简单
- 除余法是一种最常用的散列函数

散列函数(续)

- 折叠法
  - 适应情况：如果关键码的位数比地址位数多，且各位分布较均匀，不适于用数字分析法，则可以考虑折叠法
  - 方法：将关键码从某些地方断开，分为几部分，其中一部分的长度等于地址码的长度，再加上其余部分，舍弃进位

散列函数(续)

例：关键码为key=582422241，要求转换为4位的地址码。

58 | 2422 | 241

→ └──┘ ←

移位折叠相加

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

h1(key)=1064

58 | 2422 | 241

→ └──┘ →

移位相加

```
     5 8
   2 4 1
   2 4 2 2
-----------
   2 7 2 1
```

h2(key)=2721

散列函数(续)

- 中平方法
  - 方法：先求出关键码的平方，然后取中间几位作为地址。
    - 例如关键码key=4731，4731^2=22382361

      如果地址长度为3位，则可以取第三位到第五位作为散列地址，即为h(4731)=382

散列函数(续)

基数转换法

- 方法：把关键码看作是另一个进制的表示，然后再转换成原来进制的数，用数字分析法取其中几位作为散列地址。一般转换基数大于原基数，且两个基数最好互素。
- 例如key=$(236075)_{10}$是十进制数，把它看作十三进制数$(236075)_{13}$，再转换成十进制数：

$(236075)_{13}=2\times13^5+3\times13^4+6\times13^3+7\times13+5$

$=(841547)_{10}$

- 通过数字分析选择第二位到第五位，则h(236075)=4154

碰撞处理－开地址法

- 开地址法
  - 方法：
  - 当碰撞发生时，用某种方法在基本区域内形成一个探查序列，沿着探查序列逐个单元查找，直到找到该元素或碰到一个未被占用的地址为止
  - 若插入元素，则碰到空的地址单元就存放要插入的同义词。若检索元素，则碰到空的地址单元说明表中没有待查的元素
  - 形成探查序列的方法有多种，下面介绍两种探查方法

碰撞处理－开地址法(续)

- 线性探查法
- 方法:
  - 将基本存贮区看作一个循环表。若在地址为d(d=h(key))的单元发生碰撞，则依次探查下述地址单元：

    d+1, d+2, ..., m-1, 0, 1, ..., d-1 (m为基本存贮区的长度)直到找到一个空单元或查找到关键码为key的元素为止
  - 如果从单元d开始探查，查找一遍后，又回到地址d，则表示基本存贮区已经溢出

碰撞处理－开地址法(续)

- 例：已知关键码集合K={18，73，10，05，68，99，27，41，51，32，25}，用线性探查法解决碰撞

设散列表用数组element表示，大小为m(m=13)，散列函数为h(key)=key%13。

首先计算散列地址d=key%13，若地址未被占用，则插入新结点；否则进行线性探查

h(18)=5，　h(73)=8，　h(10)=10，

h(05)=5，　h(68)=3，　h(99)=8

h(27)=1，　h(41)=2，　h(51)=12，

h(32)=6，　h(25)=12

碰撞处理－开地址法(续)

- 插入18,73,10时，地址都未被占用，可以直接存放;
- 插入05时，其地址与18的地址发生碰撞，进行线性探查，由于element[6]为空单元，可以插入;
- 68的地址为3，element[3]是空单元，直接插入;
- 99的地址为8，与73的地址相冲突，线性探查后放入element[9];
- 27，41，51的地址都未被占用，可以直接存放;
- 32的地址为6，element[6]已经被5占用，线性探查后存入element[7]
- 25的地址为12，与51发生冲突，线性探查后存入element[0]，存放后的散列表为：

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	25	27	41	68		18	5	32	73	99	10		51

碰撞处理－开地址法(续)

- 在上例中，h(32)=6，h(5)=5，32和5不是同义词，但是在解决05与18的碰撞时，05已经存入element[6]，使得在插入32时，32与05不冲突的两个非同义词之间发生了碰撞
- 用线性探查法解决碰撞时，两个同义词子表结合在一起的现象称为“堆积”，即不是同义词的结点处在同一个探查序列中，增加了探查序列的长度
- 为了减少堆积的产生，可以改进线性探查方法，使探查顺序跳跃式地散列在表中

碰撞处理－开地址法(续)

- 线性探查法解决碰撞的散列表的检索和插入算法
- 注：设检索或插入元素的关键码为key，散列函数为h(key)，m为基本区域长度

碰撞处理－开地址法(续)

- 双散列函数法
- 双散列函数法中选用两个散列函数h_1和h_2
  - h_1以关键码为自变量，产生一个0到m-1之间的数作为地址。
  - h_2以关键码为自变量，产生一个1到m-1之间的并和m互素的数作为对地址的补偿，一般m取素数。
    - 例如两个散列函数可以为h1(key)=key%m和h2(key)=key%(m-2)+1。
    - 如果d=h1(key)发生碰撞，则再计算h2(key)，得到探查序列为：(d+h2(key))%m, (d+2h2(key))%m, (d+3h2(key))%m, …
  - 直到找到未被占用的地址插入同义词为止。

碰撞处理－开地址法(续)

- 例：已知关键码集合K={18，73，10，05，68，99，27，41，51，32，25}，用双散列函数法解决碰撞

设散列表用数组element表示，大小为m(m=13)，h1、h2函数分别为：

h1(key)=key%13,

h2(key)=key%(m-2)+1=key%11+1

首先计算地址d=h1(key)，若地址未被占用，则插入新结点；否则用h2函数计算地址，直到找到未被占用的地址。

h1(18)=5, h1(73)=8, h1(10)=10,
h1(05)=5, h1(68)=3, h1(99)=8

h1(27)=1, h1(41)=2, h1(51)=12,
h1(32)=6, h1(25)=12

碰撞处理－开地址法(续)

- 05与18冲突，用h2函数对地址进行补偿，h2(05)=5%11+1=6，得到散列地址为(d+6)%m=(5+6)%13=11，是空地址单元可以直接插入
- 同样可以得到99的插入地址为9，25的地址为7。存放后的散列表为：

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		27	41	68		18	32	25	73	99	10	5	51

碰撞处理－开地址法(续)

- 用开地址法构造散列表，删除结点时，不能简单地将要删除的结点空间置为空，因为各种开地址法中，空地址单元是检索失败的依据，删除一个结点会影响其它结点的检索，因此只能在被删除结点上做标记，不能真正删除。这样，做了许多删除后，形式上散列表是满的，但实际上却很空，浪费了存贮空间。
- 改进的方法是，在插入结点时利用做了标记的地址空间

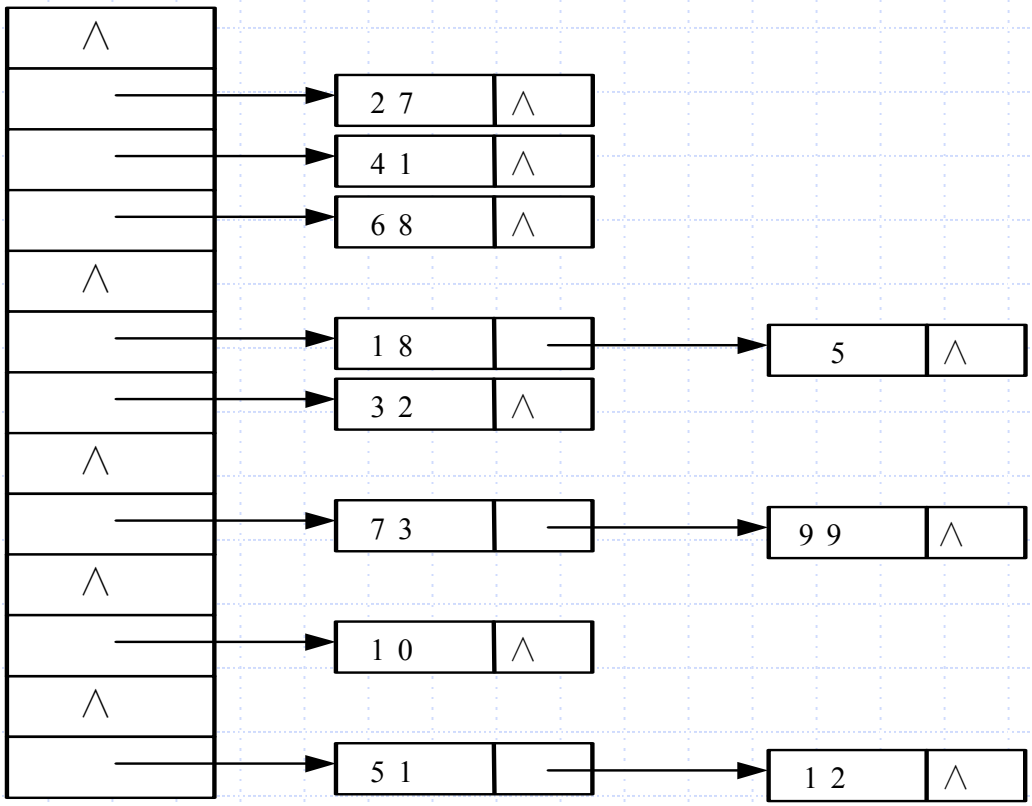
碰撞处理－拉链法

- 方法：
  - 碰撞发生时建立一个链表，若n个关键码映象到基本区域的m个存贮单元上，则最多可以建立m个链表
  - 例如已知关键码集合K={18，73，10，05，68，99，27，41，51，32，25}，用拉链法解决碰撞。
  - 设散列函数为h(key)=key%13，插入新结点时，将结点插入到链表中

碰撞处理－拉链法(续)

h(18)=5, h(73)=8, h(10)=10, h(05)=5, h(68)=3, h(99)=8, h(27)=1, h(41)=2, h(51)=12, h(32)=6, h(25)=12

得到的散列表为：

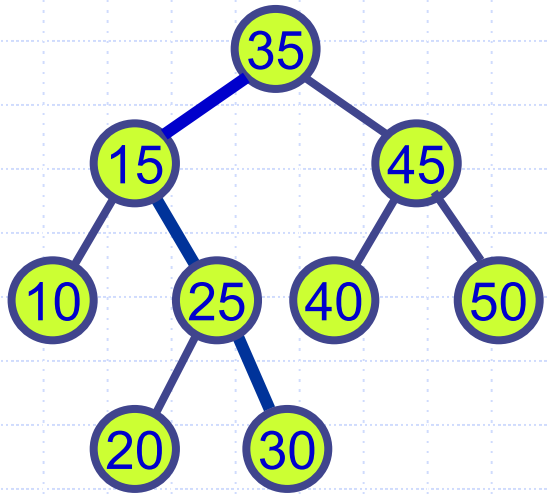


碰撞处理－拉链法(续)

- 优点:
  - 设n个关键码通过散列函数对应到m个单元中，每个同义词子表的平均长度为n/m，检索的速度较快;
  - 由于各链表上的结点空间是动态申请的，因此更适应于造表前无法确定表长的情况;
  - 拉链法不会造成堆积现象，结点的删除操作较方便。
- 缺点:
  - 负载因子较大时，拉链法比开地址法占用的空间多，但负载因子越大，开地址法所需的检索长度越长

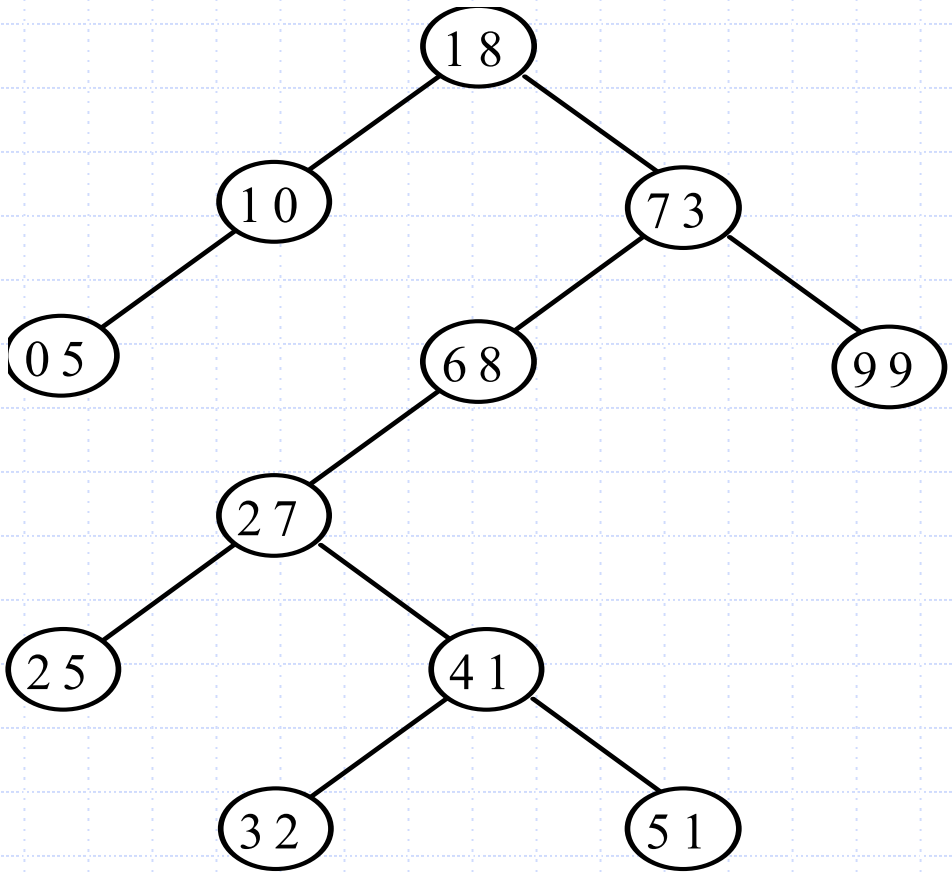
二叉树表示

- 基本概念
- 二叉排序树具有以下性质：
  - 如果任一结点的左子树非空，则左子树中的所有结点的关键码都小于根结点的关键码;
  - 如果任一结点的右子树非空，则右子树中的所有结点的关键码都大于根结点的关键码。
    - 注意：若从根结点到某个叶结点有一条路径，路径左边的结点的关键码不一定小于路径上的结点的关键码。
- 用二叉排序树表示字典，树中的一个结点对应于字典中的一个元素。



二叉树表示(续)

- 例如关键码集合
  K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}，得到的一棵二叉排序树如图所示



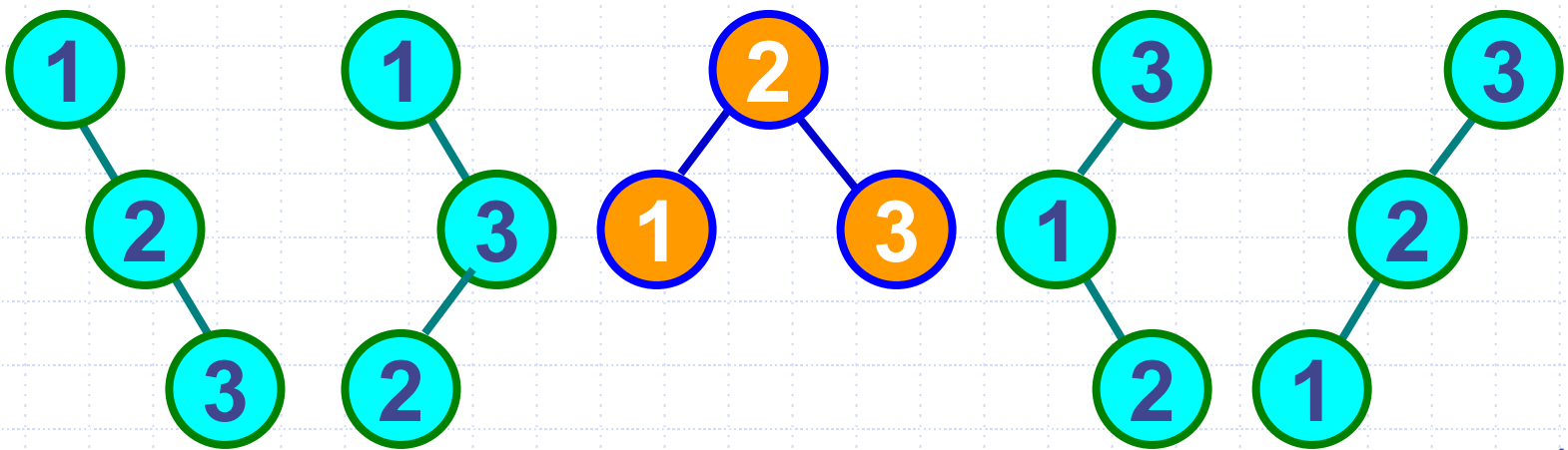
二叉树表示(续)

n 个结点的二叉排序树的数目

【例】3 个结点的二叉排序树

$$\frac{1}{3+1}C_{2*3}^{3}=\frac{1}{4}*\frac{6*5*4}{3*2*1}=5$$

{123} {132} {213} {312} {321}



二叉树表示(续)

- 二叉排序树采用llink-rlink法表示，其存储结构定义如下：

```
struct BinNode;
typedef struct BinNode * PBinNode;
struct BinNode
{  KeyType key;                          /* 结点的关键码字段 */
   DataType other;                       /* 结点的属性字段 */
   PBinNode llink, rlink;                /* 二叉树的左、右指针 */
};
typedef struct BinNode * BinTree;
typedef BinTree * PBinTree;
```

二叉树表示(续)

- 二叉排序树的检索
  - 方法: 二叉排序树中查找某个结点和二分法检索相似，也是逐步缩小检索范围的过程，比线性检索的效率高得多
  - 二叉排序树的检索
    - 设在二叉排序树上查找关键码为key的结点，算法结束时返回检索成功或失败的标志
    - 检索成功时，由参数position指向查找到的结点；检索失败时，position指向应插入的子树的根结点

在二叉排序树上进行检索，是一个从根结点开始，沿某一个分支逐层向下进行比较判等的过程。它可以是一个递归的过程。

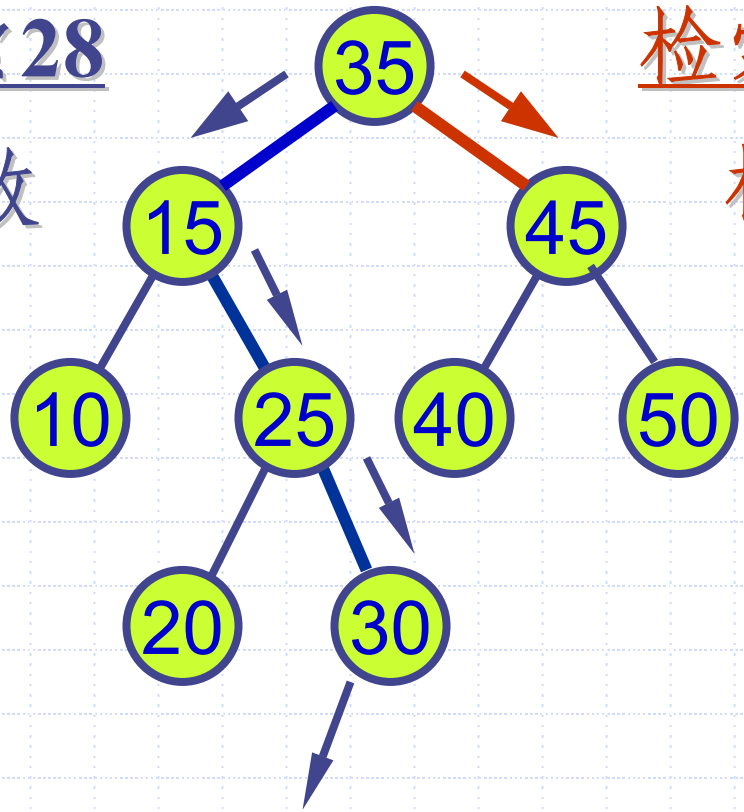
二叉树表示(续)

- 假设想要在二叉排序树中检索关键码为x的元素，检索过程从根结点开始。
- 如果根指针为NULL，则检索不成功；否则用给定值x与根结点的关键码进行比较：
  - 如果给定值等于根结点的关键码，则检索成功。
  - 如果给定值小于根结点的关键码，则继续递归检索根结点的左子树;
  - 否则。递归检索根结点的右子树。

二叉树表示(续)

检索28

检索失败



检索45

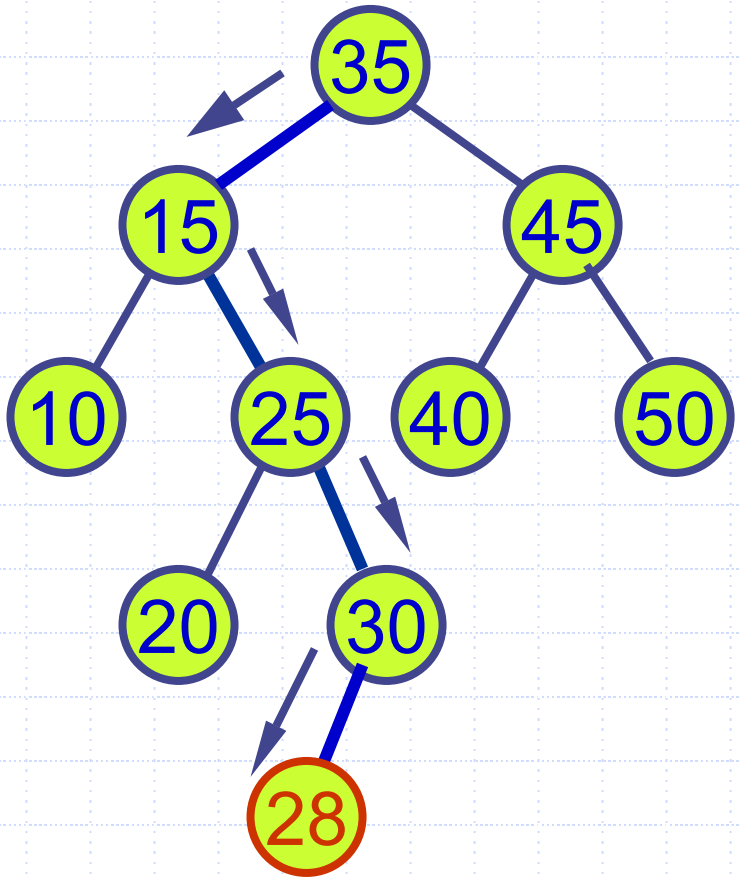
检索成功

二叉树表示(续)

- 二叉排序树的插入和构造
  - 在二叉排序树中插入新结点，要保证插入后仍满足二叉排序树的定义
  - 插入新结点的方法是：
    - 如果二叉排序树为空，则新结点作为根结点
    - 如果二叉排序树非空，则将新结点的关键码与根结点的关键码比较，若小于根结点的关键码，则将新结点插入到根结点的左子树中；否则，插入到右子树中
  - 子树中的插入过程和树中的插入过程相同，如此进行下去，直到找到该结点，或者直到新结点成为叶子结点为止

二叉树表示(续)

- 每次结点的插入，都要从根结点出发检索插入位置，然后把新结点作为叶结点插入。



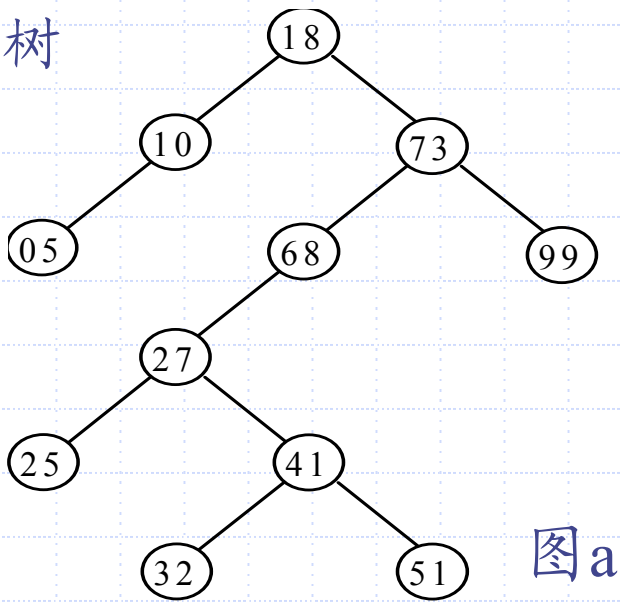
插入新结点**28**

二叉树表示(续)

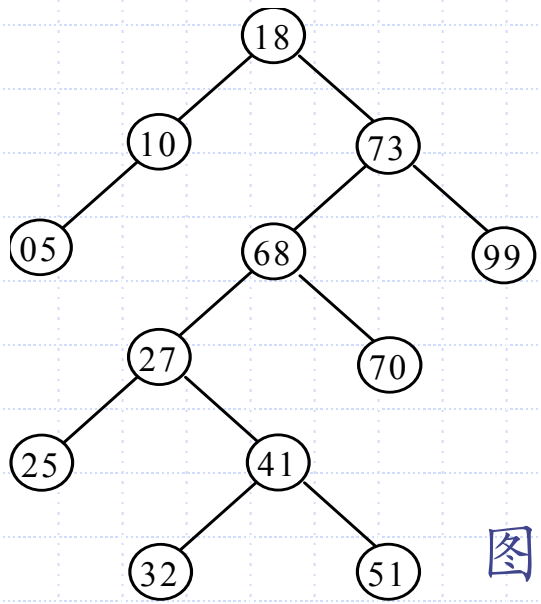
- 为了向二叉排序树中插入一个新元素，必须先检查这个元素是否在树中已经存在。
- 在插入之前，先使用检索算法在树中检查要插入元素有还是没有。
  - 检索成功：树中已有这个元素，不再插入。
  - 检索不成功：树中原来没有关键码等于给定值的结点，把新元素加到检索操作停止的地方。

二叉树表示(续)

- 在图a所示的二叉排序树中插入关键码为70的结点。插入后的二叉排序树如图b所示，插入过程如下
  - 插入时二叉排序树非空，70与根结点18相比，70>18，应插入18的右子树
  - 18的右子树非空，70<73，应插入73的左子树
  - 73的左子树非空，70>68，应插入68的右子树
  - 68的右子树为空，因此，70作为68的右子女插入二叉排序树



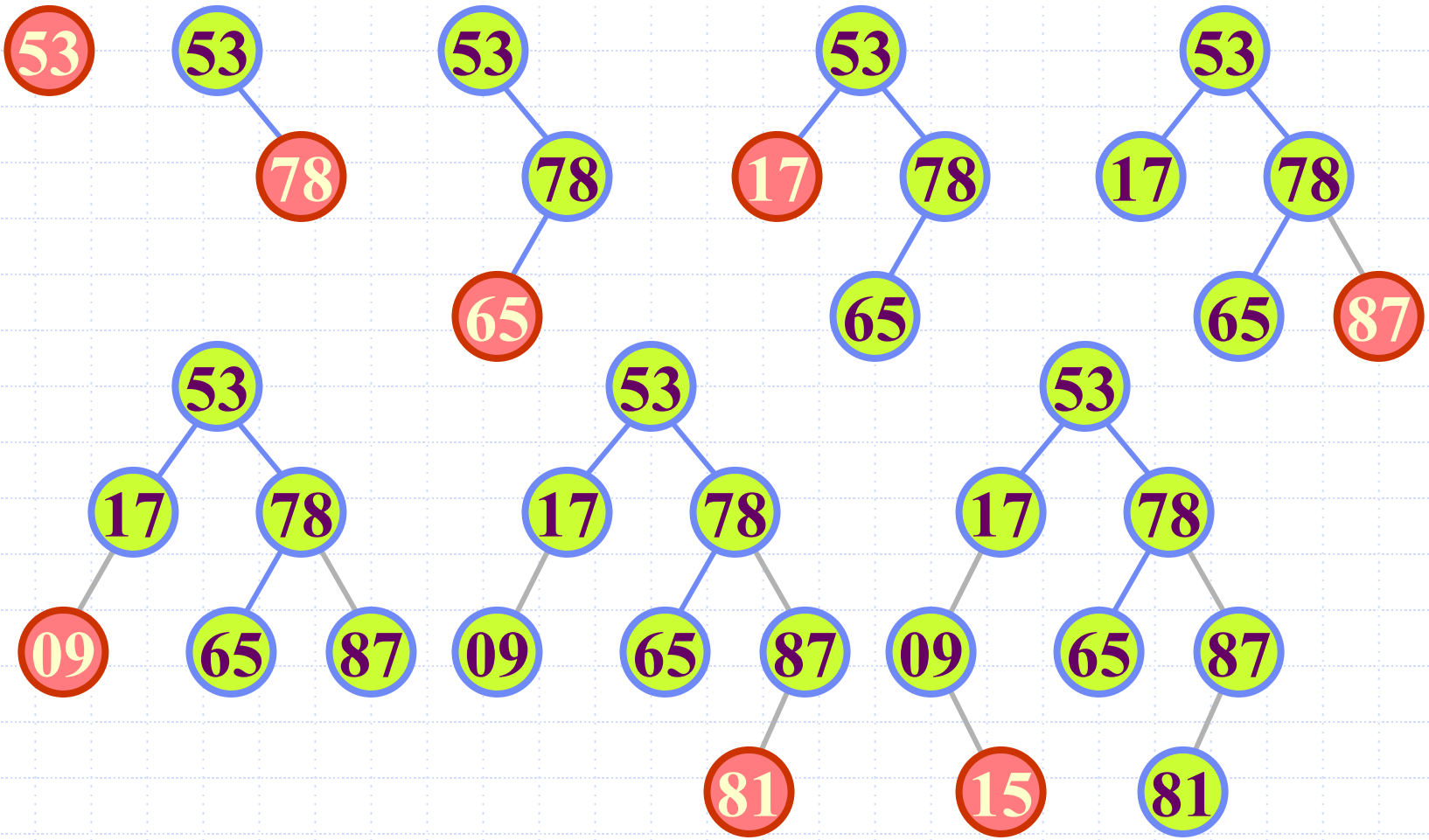
图a



图b

输入数据，建立二叉排序树的过程

输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }

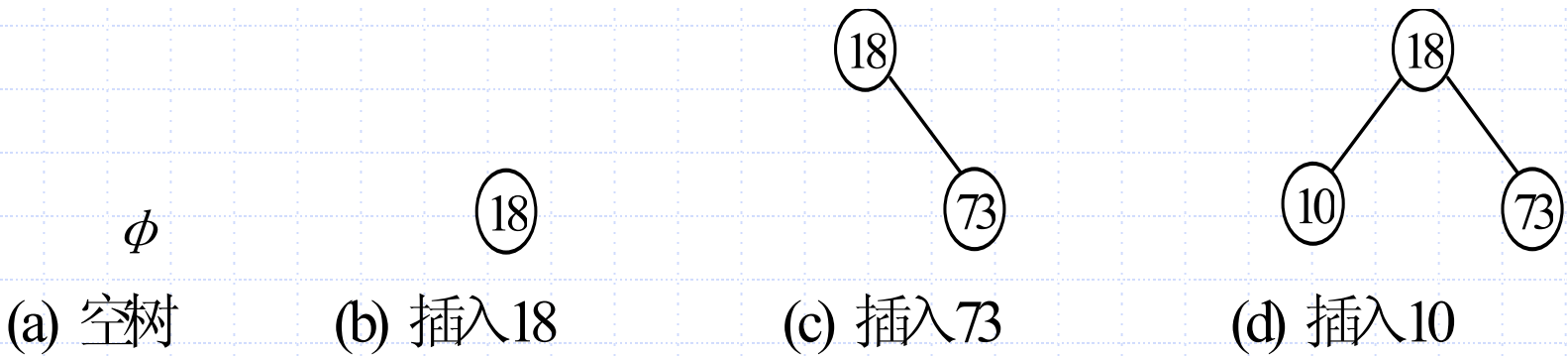


二叉树表示(续)

- 二叉排序树中插入结点的算法

二叉树表示(续)

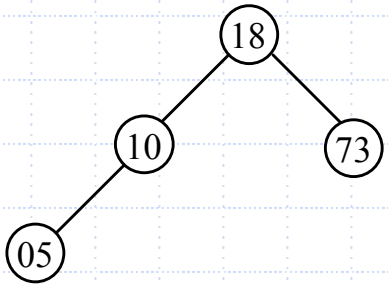
例子: 关键码集合为
K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}，写出二叉排序树的构造过程



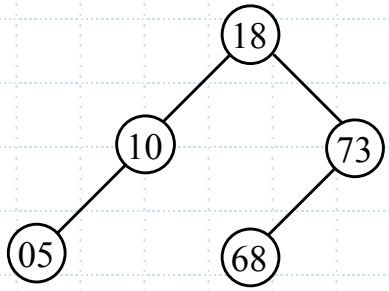
ϕ

(a) 空树　(b) 插入18　(c) 插入73　(d) 插入10

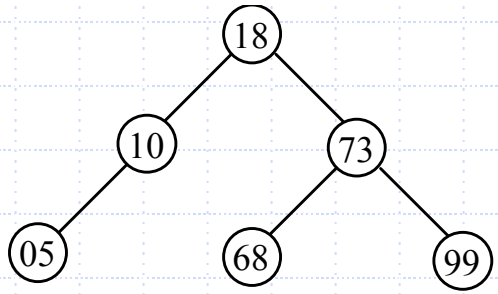
二叉树表示(续)



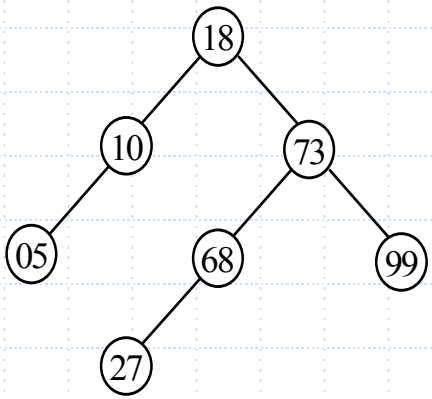
(e) 插入 05



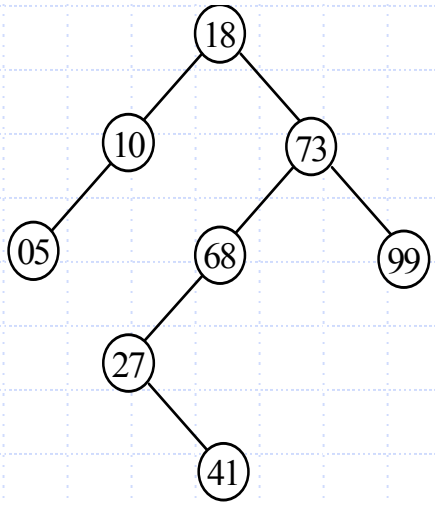
(f) 插入 68



(g) 插入 99

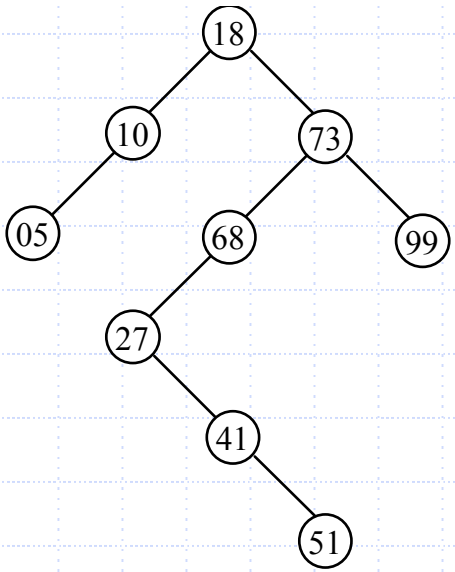


(h) 插入27

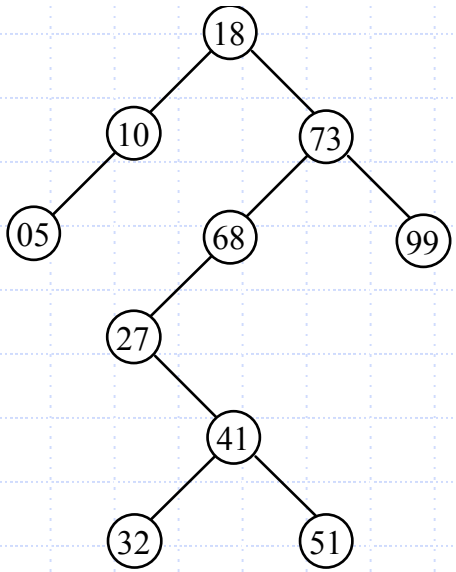


(i) 插入41

二叉树表示(续)

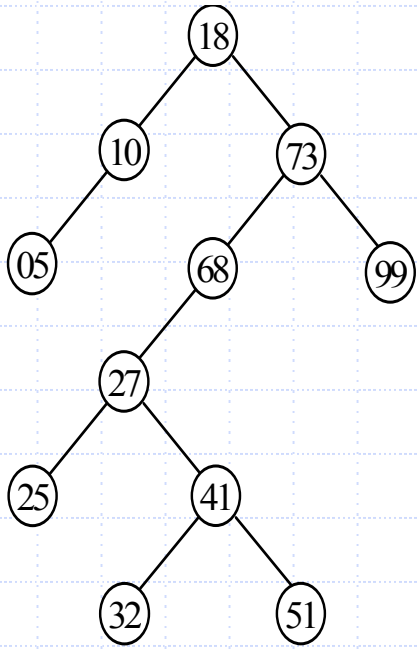


(j) 插入 51



(k) 插入 32

二叉树表示(续)



(l) 插入25

二叉树表示(续)

二叉排序树的构造算法

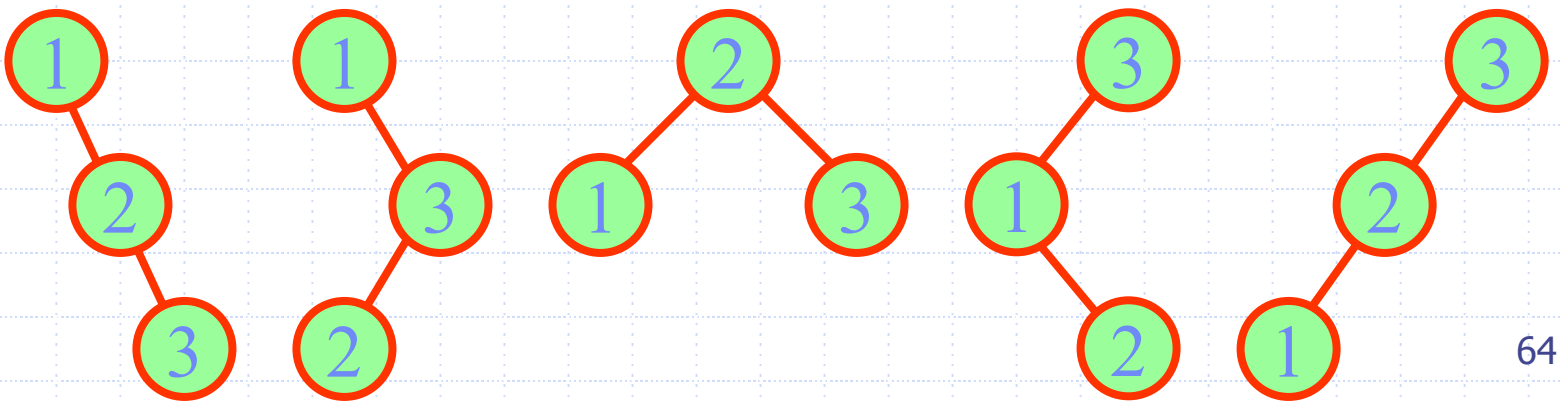
```
void creatTree(PBinTree ptree, SeqDictionary dic)
    {
      int i;
       for(i=0;i<dic.n;i++)
         insertNode(ptree,dic.element[i].key);
         /* 将新结点插入树中 */
    }
```

二叉树表示(续)

同样3个数据{1, 2, 3}，输入顺序不同，建立起来的二叉排序树的形态也不同。这直接影响到二叉排序树的搜索性能。

如果输入序列选得不好，会建立起一棵单支树，使得二叉排序树的高度达到最大。

{1, 2, 3}　{1, 3, 2}　{2, 1, 3}　{3, 1, 2}　{3, 2, 1}

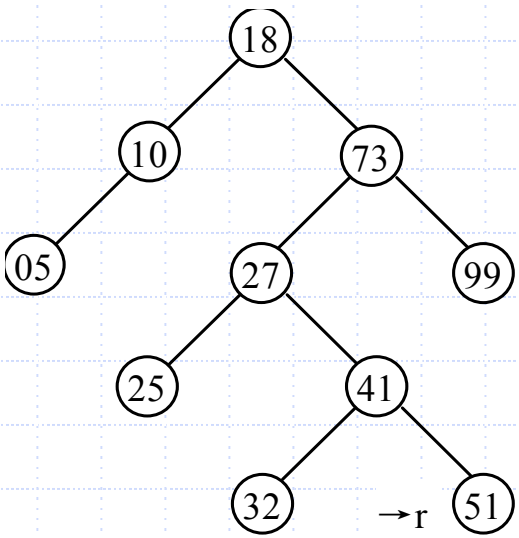


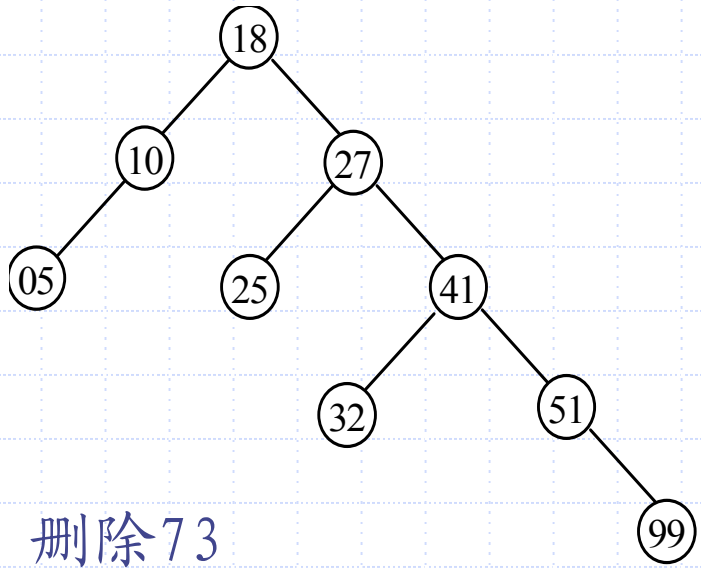
二叉树表示(续)

- 二叉排序树的删除
  - 在二叉排序树中删除一个指定结点，删除结点后的二叉树必须满足二叉排序树的性质
  - 下面给出两种删除方法

二叉树表示(续)

- 第一种方法：
  - 如果被删除结点p没有左子树，则用p的右子女代替p即可
  - 否则，在p的左子树中，对称序周游找出最后一个结点r（r一定无右子女），将r的右指针指向p的右子女，用p的左子女代替p结点



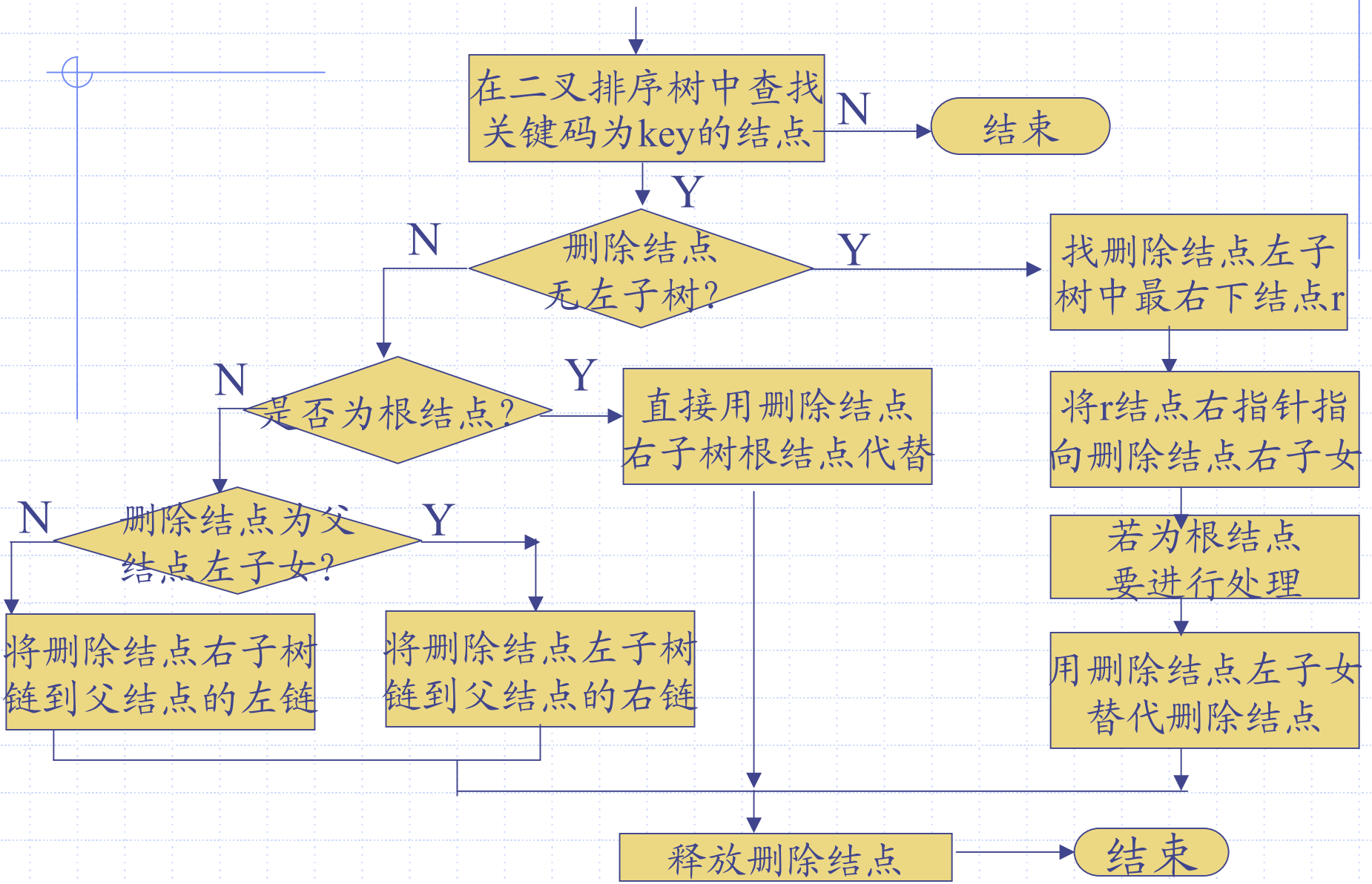


删除73

二叉树表示(续)

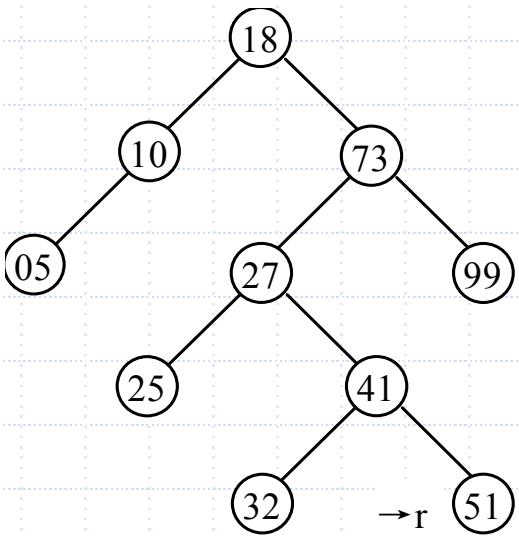
- 说明:
  - p的右子树中结点的关键码都大于p结点的关键码
  - 在p的左子树中按对称序周游找出的最后一个结点r，r在对称序列中紧排在p结点的前面
  - 因此，用r的右指针指向p的右子树，可以保证二叉排序树的性质
- 二叉排序树的删除算法

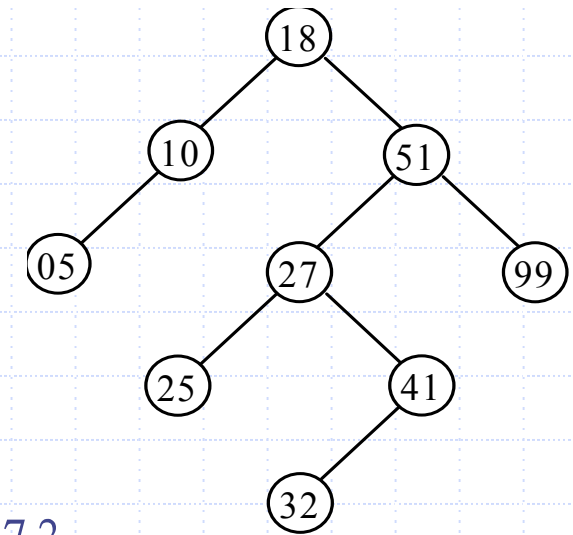
二叉排序树的删除算法


在二叉排序树中查找关键码为key的结点
N
结束
Y
删除结点无左子树?
N
Y
找删除结点左子树中最右下结点r
是否为根结点?
N
Y
直接用删除结点右子树根结点代替
将r结点右指针指向删除结点右子女
删除结点为父结点左子女?
N
Y
若为根结点要进行处理
将删除结点右子树链到父结点的左链
将删除结点左子树链到父结点的右链
用删除结点左子女替代删除结点
释放删除结点
结束

二叉树表示(续)

- 第二种方法：
  - 同第一种方法的(1)(如果被删除结点p没有左子树，则用p的右子女代替p即可)
  - 同第一中方法找到r结点后，将r删除(r一定无右子女，用r的左子女代替r结点即可)
  - 用r结点代替被删除的结点即可





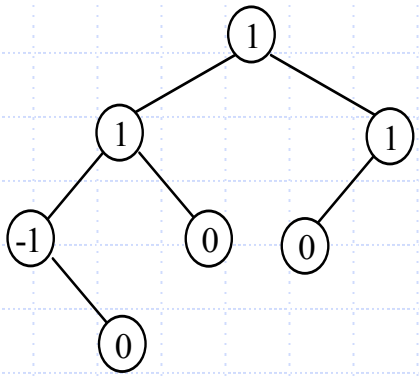
删除73

二叉树表示(续)

- 说明
  - 因为r在对称序列中紧排在p结点之前，所以用r代替p结点不会改变二叉排序树的性质

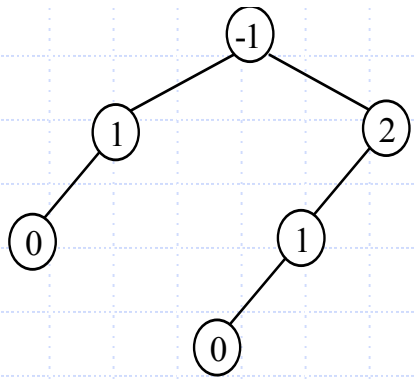
AVL树表示（续）

- 基本概念
  - 平衡二叉排序树
  - 每个结点左、右子树深度之差的绝对值不超过1
  - 结点左、右子树深度之差定义为该结点的平衡因子，平衡二叉排序树中每个结点的平衡因子只能是1、0或－1



(a)

平衡二叉排序树



(b)

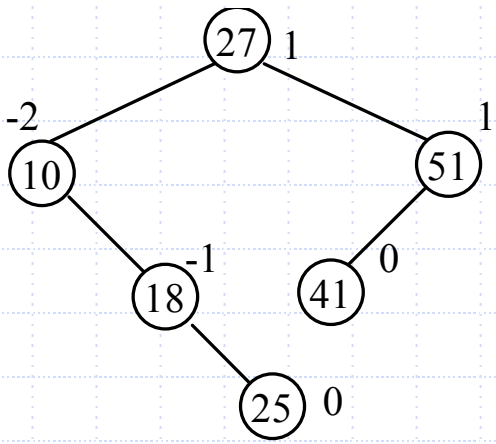
非平衡二叉排序树

平衡二叉排序树的运算

- 插入:
  - 在平衡二叉排序树中插入新结点时，如果新结点插入后不影响其父结点为根的子树深度，则不会破坏整个二叉排序树的平衡;
  - 反之，若父结点为根的子树深度增加了，则会引起一连串的反映
    - 在其祖先的某一层上不再影响子树的深度，则整个二叉排序树仍然是平衡的;
    - 在其祖先的某一层上破坏了平衡的要求，使整个二叉排序树不再是AVL树

平衡二叉排序树的运算(续)

- 失去平衡后的处理方法
  - 首先找出最小不平衡子树，在保证排序树性质的前提下，调整最小不平衡子树中各结点的连接关系，以达到新的平衡
  - 最小不平衡子树指离插入结点最近，且以平衡因子绝对值大于1的结点为根的子树
- 例在插入25后，图中以10为根结点的子树是最小不平衡子树

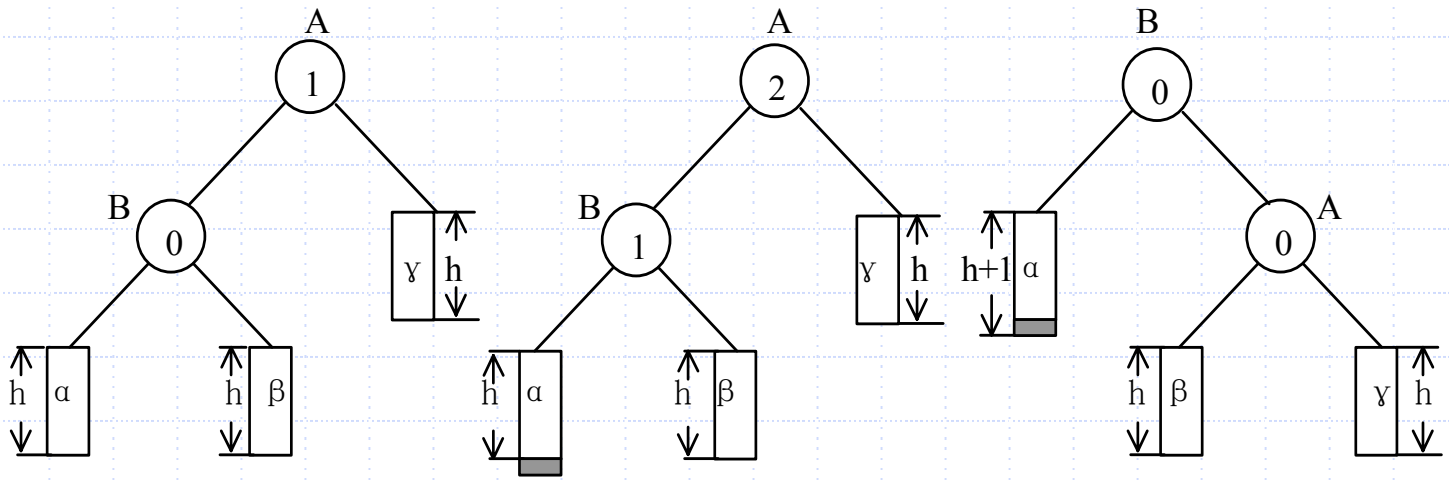


平衡二叉排序树的运算(续)

- 假设最小不平衡子树的根结点为A，调整子树的操作可归纳为以下四种情况
  - LL型调整
  - LR型调整
  - RL型调整
  - RR型调整

平衡二叉排序树的运算(续)

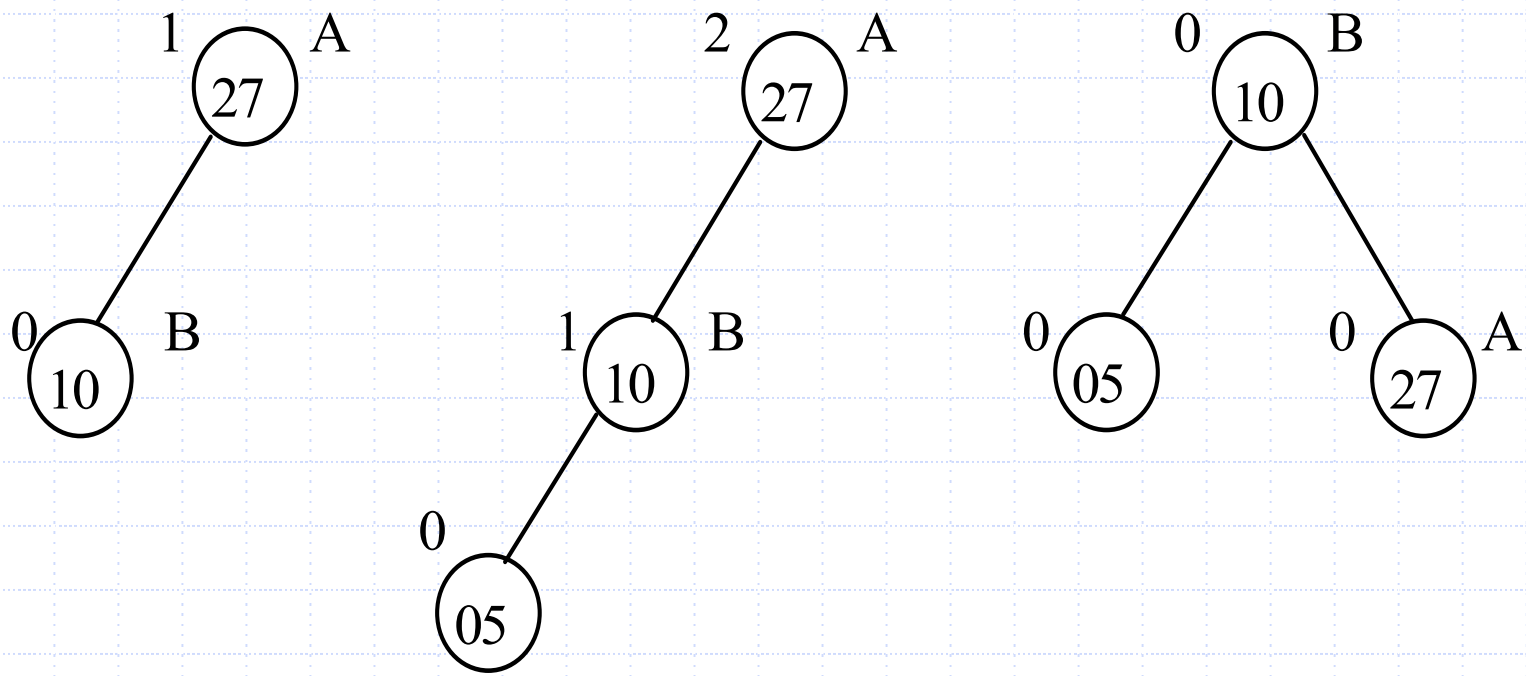
- LL型调整
  - **破坏平衡的原因**：由于在A的左子女(L)的左子树(L)中入结点，使A的平衡因子由1变为2而失去平衡，调整过程如图所示。
  - 图中长方框表示子树，α、β、γ子树的深度都是h(h⩾0；若h=0，则都是空树)。带阴影的小框表示要插入的结点。原来结点A和B的平衡因子分别为1和0，B的左子树α上插入一个新结点，使以A为根的子树成为最小不平衡子树。



平衡二叉排序树的运算(续)

- 调整规则是
  - 将A的左子女B提升为新二叉树的根
  - 原来的根A连同其右子树γ向右下旋转成为B的右子树
  - B的原右子树β作为A的左子树。如同使用了代数中的结合律：(αBβ)A(γ)= (α)B(βAγ)
    - (αBβ)A(γ)是以A为根的二叉树的对称序列，其左、右括号内是A的左、右子树的对称序列
    - (α)B(βAγ)是以B为根的二叉树的对称序列；B的左子树是α，右子树是βAγ，右子树的根是A
  - 调整后仍保持二叉排序树的性质

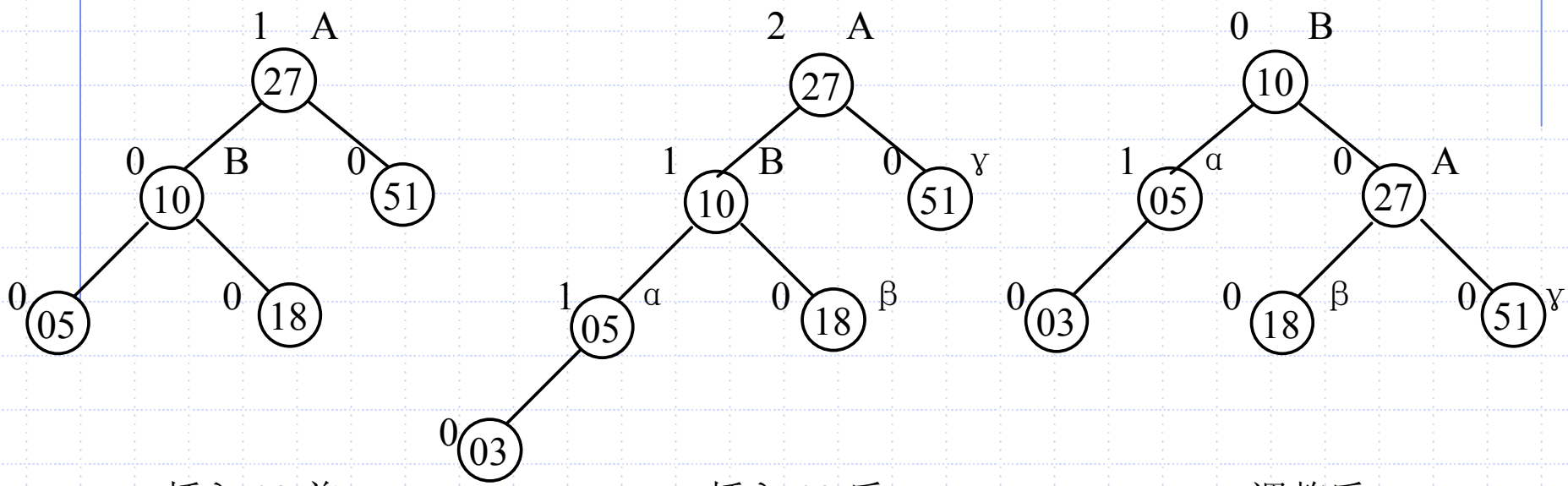
平衡二叉排序树的运算(续)



输入05前　　输入05后　　调整后

(a) α、β和γ都是空树的调整

平衡二叉排序树的运算(续)

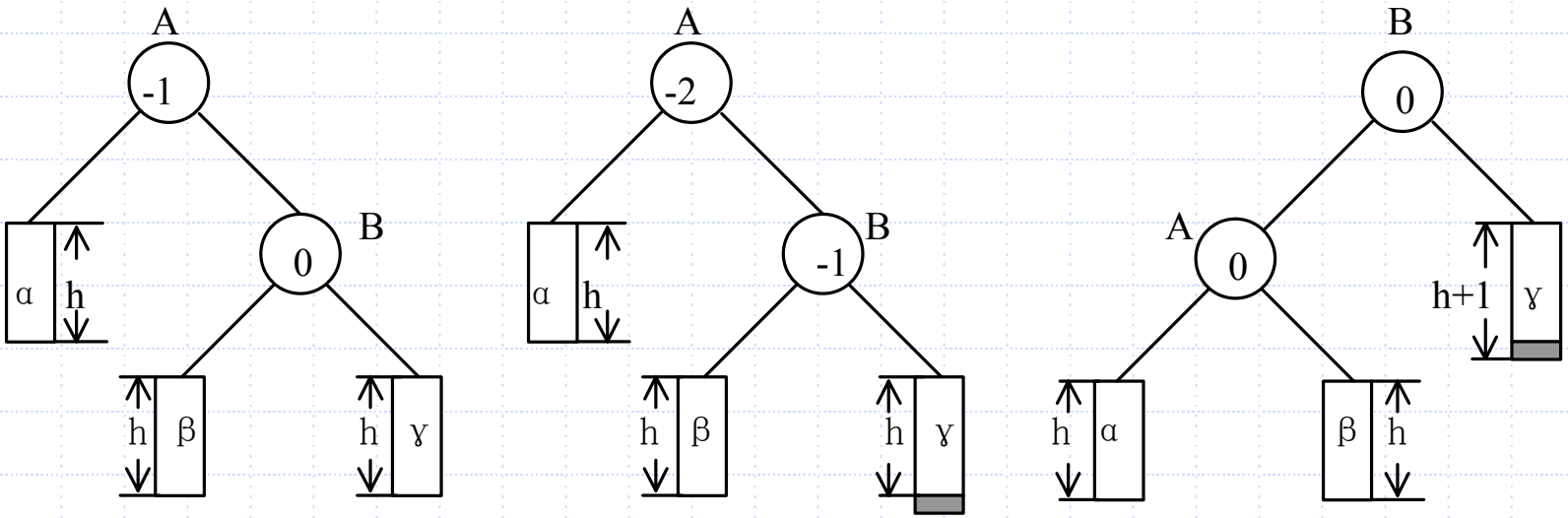


插入 03 前　　插入 03 后　　调整后

(a) α 、 β 和 γ 都是非空树的调整

平衡二叉排序树的运算(续)

- RR型调整
  - 破坏平衡的原因：由于在A的右子女(R)的右子树(R)中插入结点，使A的平衡因子由－1变为－2而失去平衡
  - 调整过程如图所示

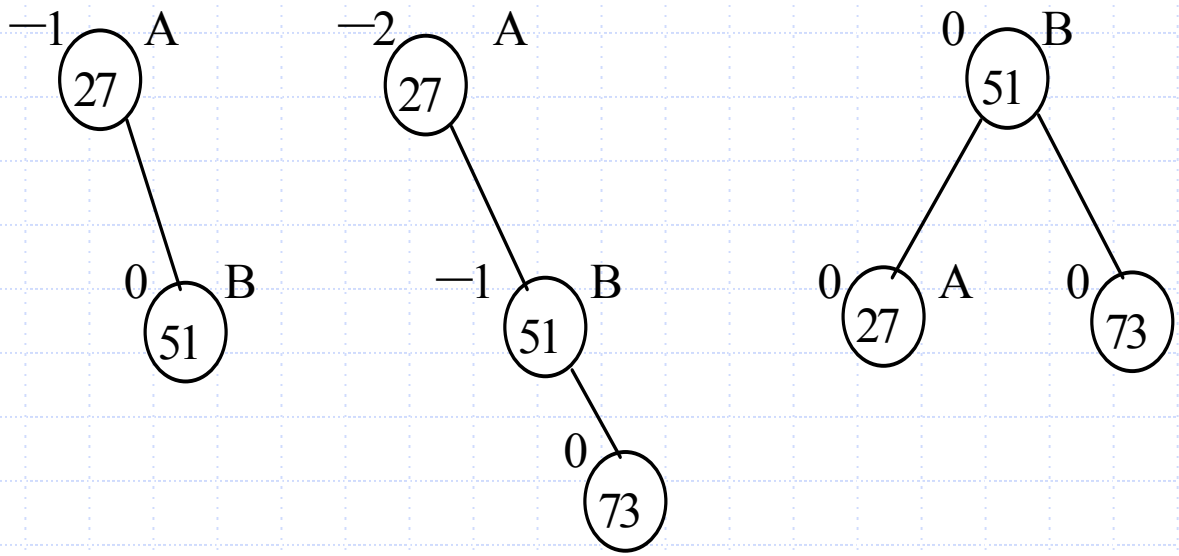


平衡二叉排序树的运算(续)

- RR型调整规则:
  - 与LL型的对称
  - 将A的右子女B提升为新二叉树的根
  - 原来的根A连同其左子树向左下旋转成为B的左子树；B的原左子树作为A的右子树
  - 如同代数中的结合律：

    $(\alpha)A(\beta B\gamma)=(\alpha A\beta)B(\gamma)$

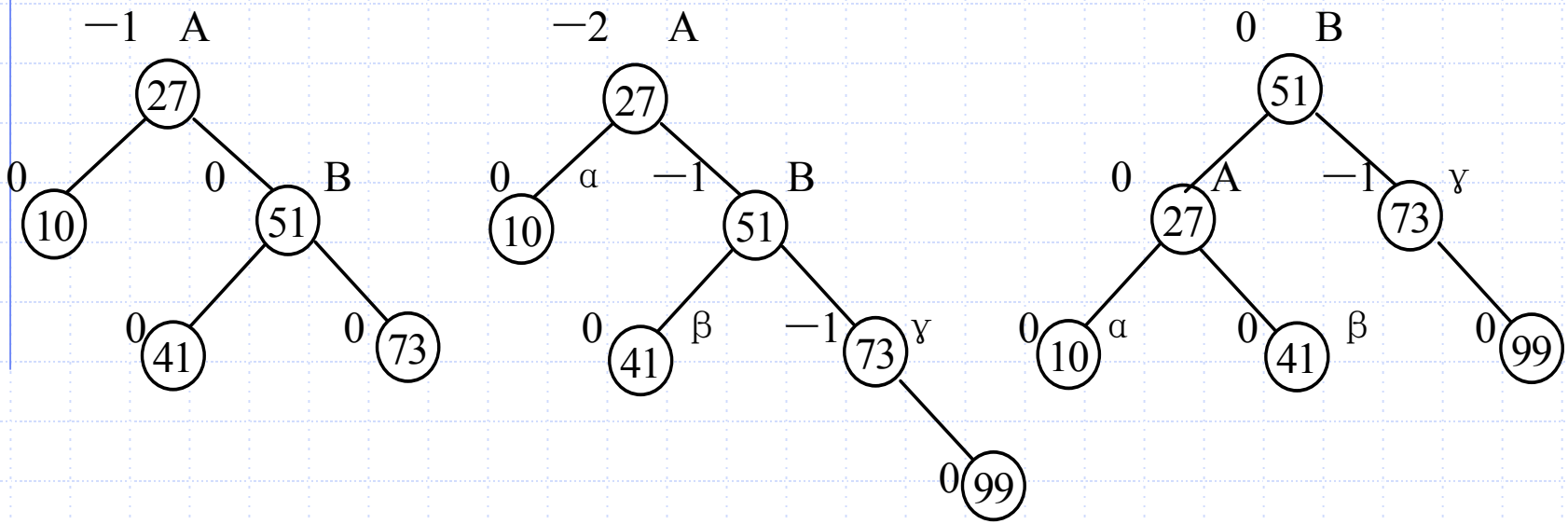
平衡二叉排序树的运算(续)



输入73前　　输入73后　　调整后

(a) α、β和γ都是空树的调整

平衡二叉排序树的运算(续)



插入 99 前　　插入 99 后　　调整后

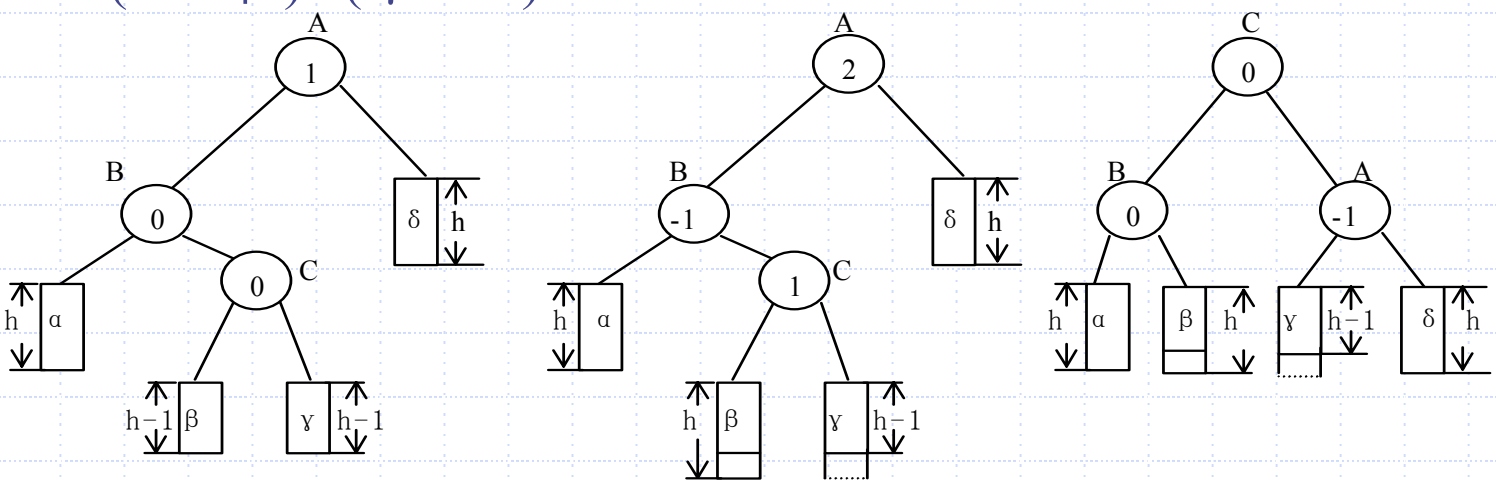
(b) α、β 和 γ 都是非空树的调整

平衡二叉排序树的运算(续)

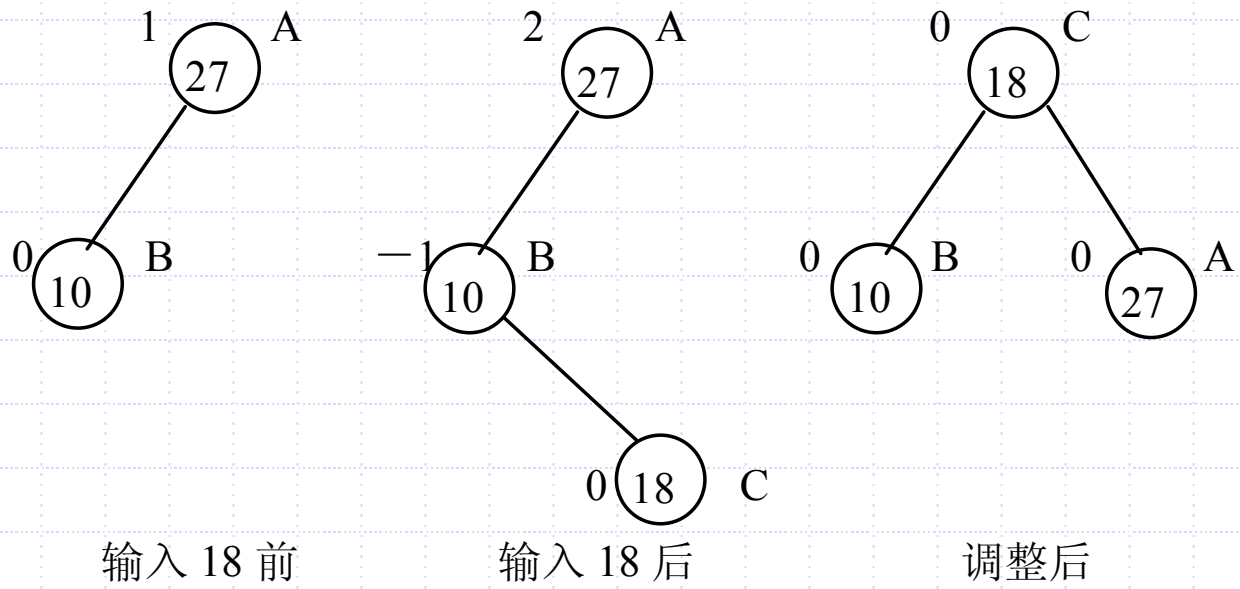
- LR型调整
  - 破坏平衡的原因:
  - 由于在A的左子女(L)的右子树(R)中插入结点，使A的平衡因子由1变为2而失去平衡
    - 若h=0，则α、β、γ、δ全为空树，C就是新插入的结点，记为LR(0)
    - 若h>0，则新结点可能插在C的左子树中，也可能插在C的右子树中，分别记为LR(L)和LR(R)

平衡二叉排序树的运算(续)

- 三种情况的调整规则相同：
  - 设C为A的左子女的右子女，将A的孙子结点C提升为新二叉树的根
  - 原C的父结点B连同其左子树α向左下旋转成为新根C的左子树，原C的左子树β成为B的右子树
  - 原根A连同其右子树δ向右下旋转成为新根C的右子树，原C的右子树γ成为A的左子树
  - 用代数中的结合律说明这种操作：((α)B(βCγ))A(δ)=(αBβ)C(γAδ)



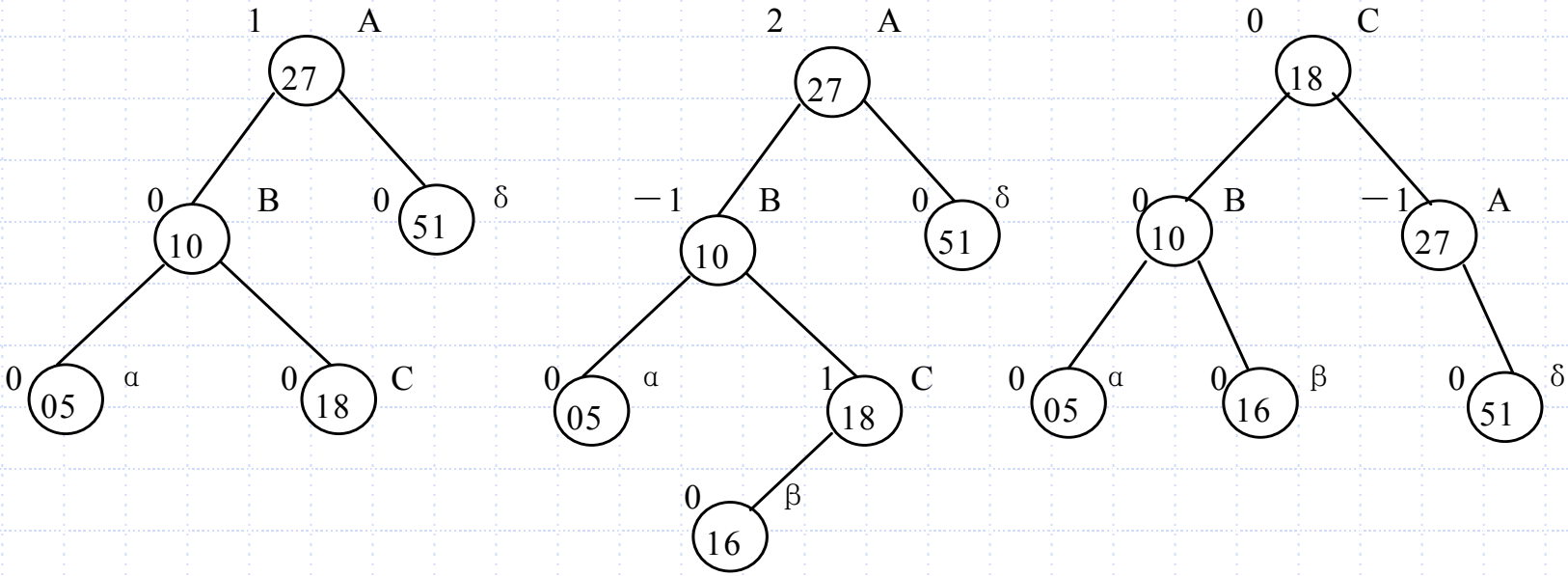
平衡二叉排序树的运算(续)



输入 18 前　　输入 18 后　　调整后

(a) LR(0)型调整

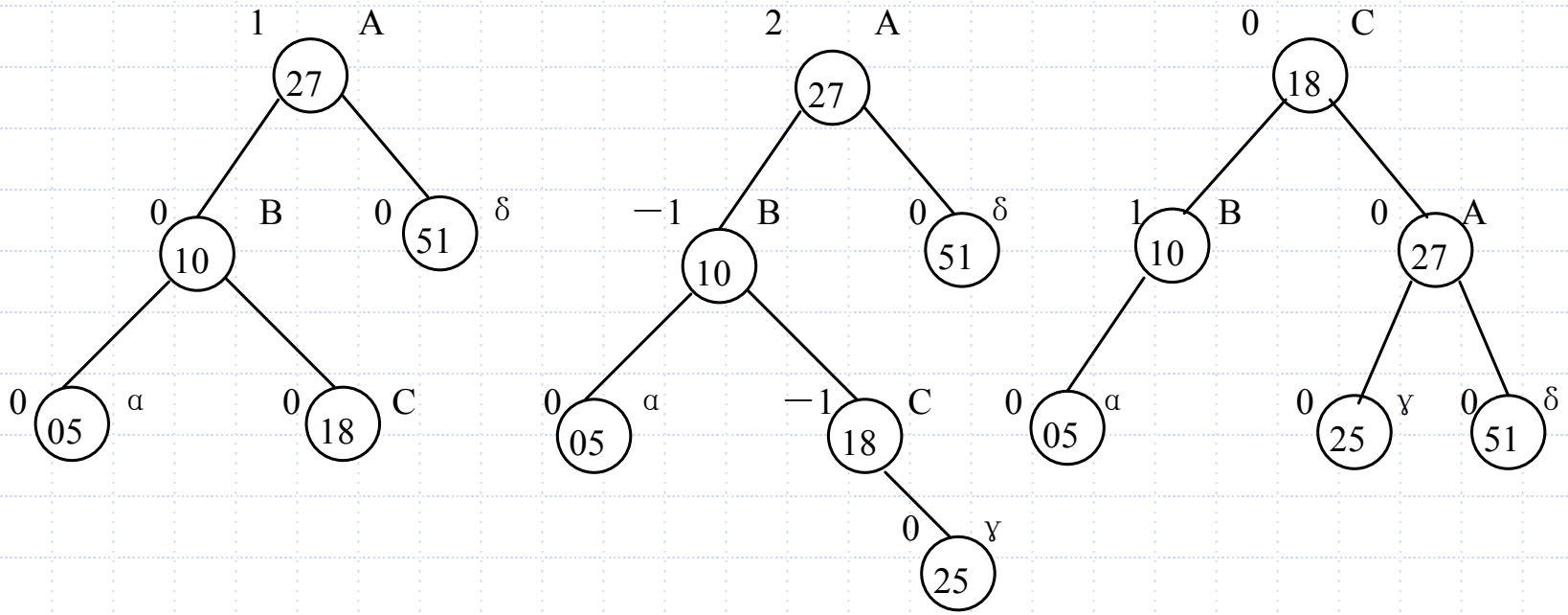
平衡二叉排序树的运算(续)



输入 16 前　　输入 16 后　　调整后

(b) LR(L)型调整

平衡二叉排序树的运算(续)

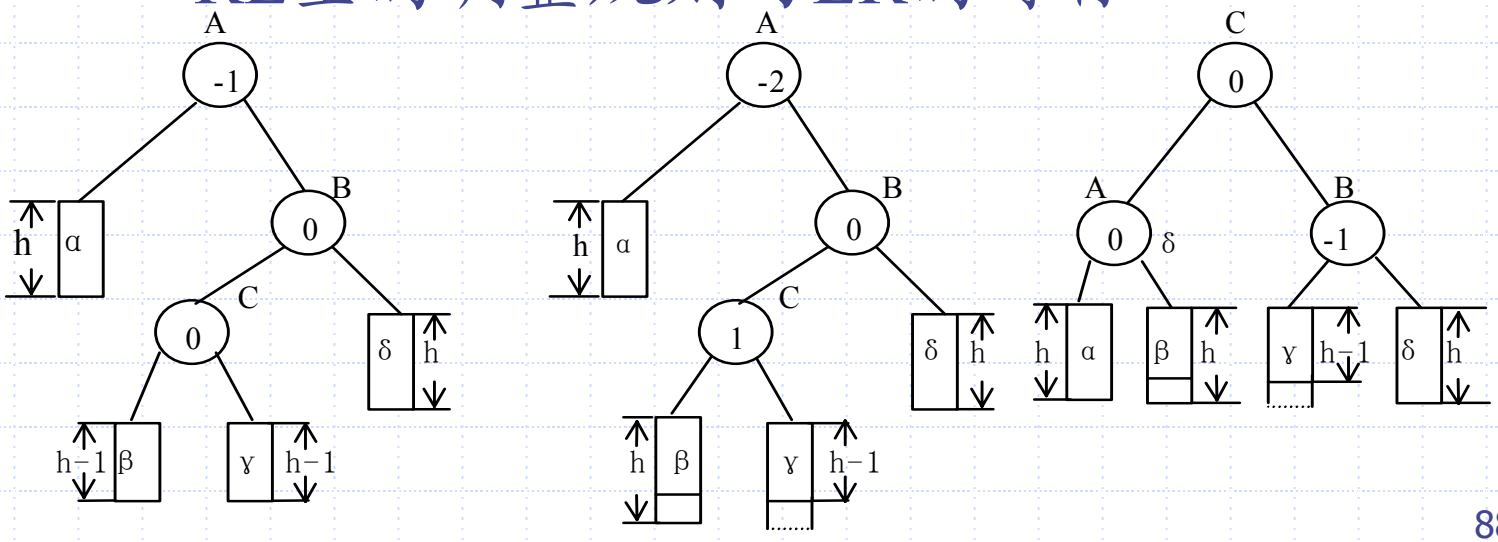


输入 25 前　　输入 25 后　　调整后

(c) LR(R)型调整

平衡二叉排序树的运算(续)

- RL型调整
  - 破坏平衡的原因:
    - 由于在A的右子女(R)的左子树(L)中插入结点，使A的平衡因子由－1变为－2而失去平衡
    - RL型的调整规则与LR的对称

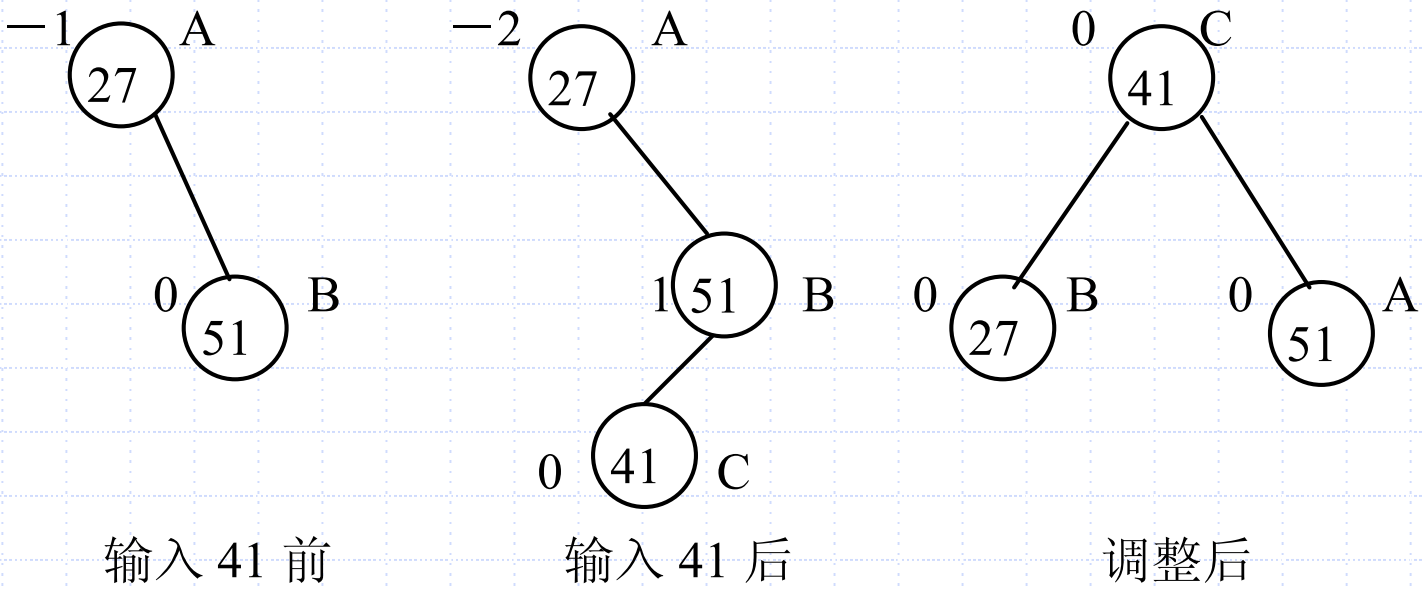


平衡二叉排序树的运算(续)

- 调整规则
  - 设C为A的右子女的左子女，将A的孙子结点C提升为新二叉树的根，原来C的父结点B连同其右子树δ向右下旋转成为新根C的右子树，C的原右子树γ成为B的左子树
  - 原来的根A连同其左子树α向左下旋转成为新根C的左子树，原来C的左子树β成为A的右子树
  - 用代数中结合律说明：

    (α)A((βCγ)B(δ))=(αAβ)C(γBδ)

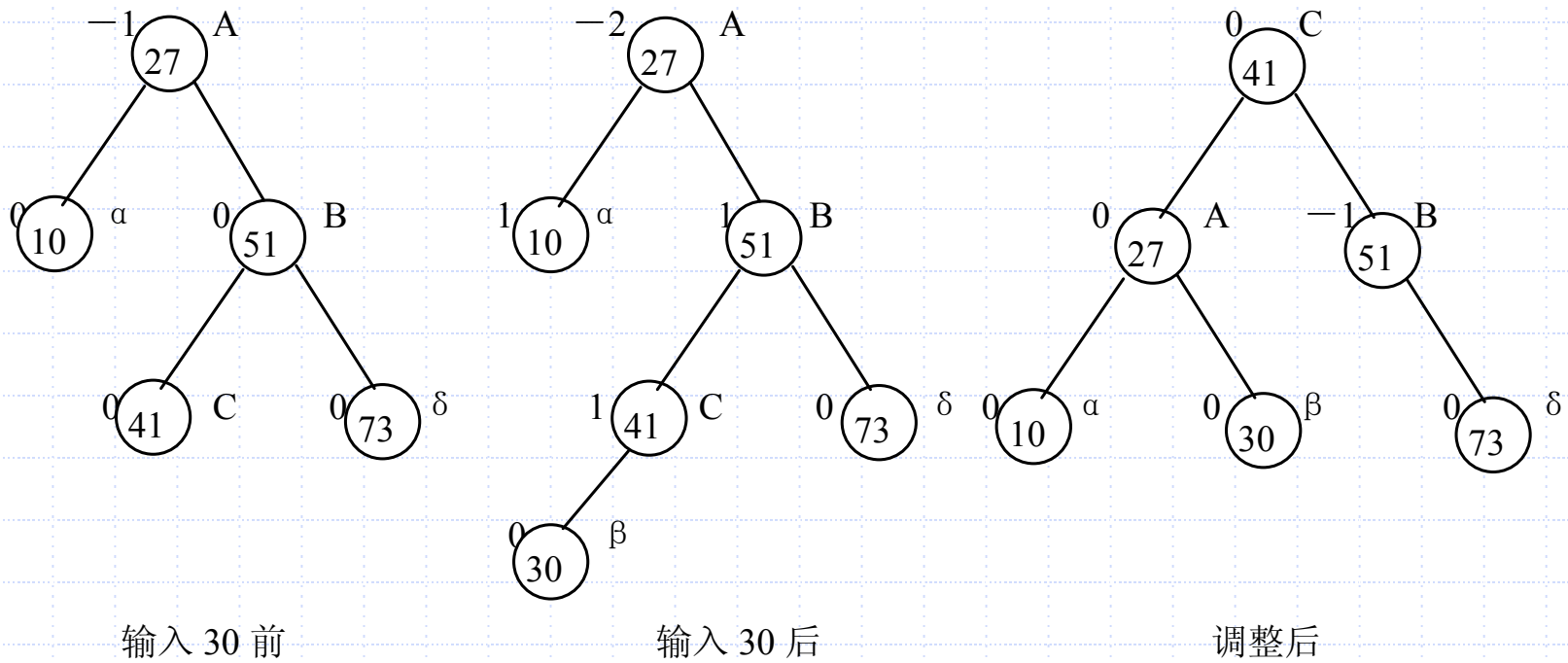
平衡二叉排序树的运算(续)



输入 41 前　　输入 41 后　　调整后

(a) RL(0)型调整

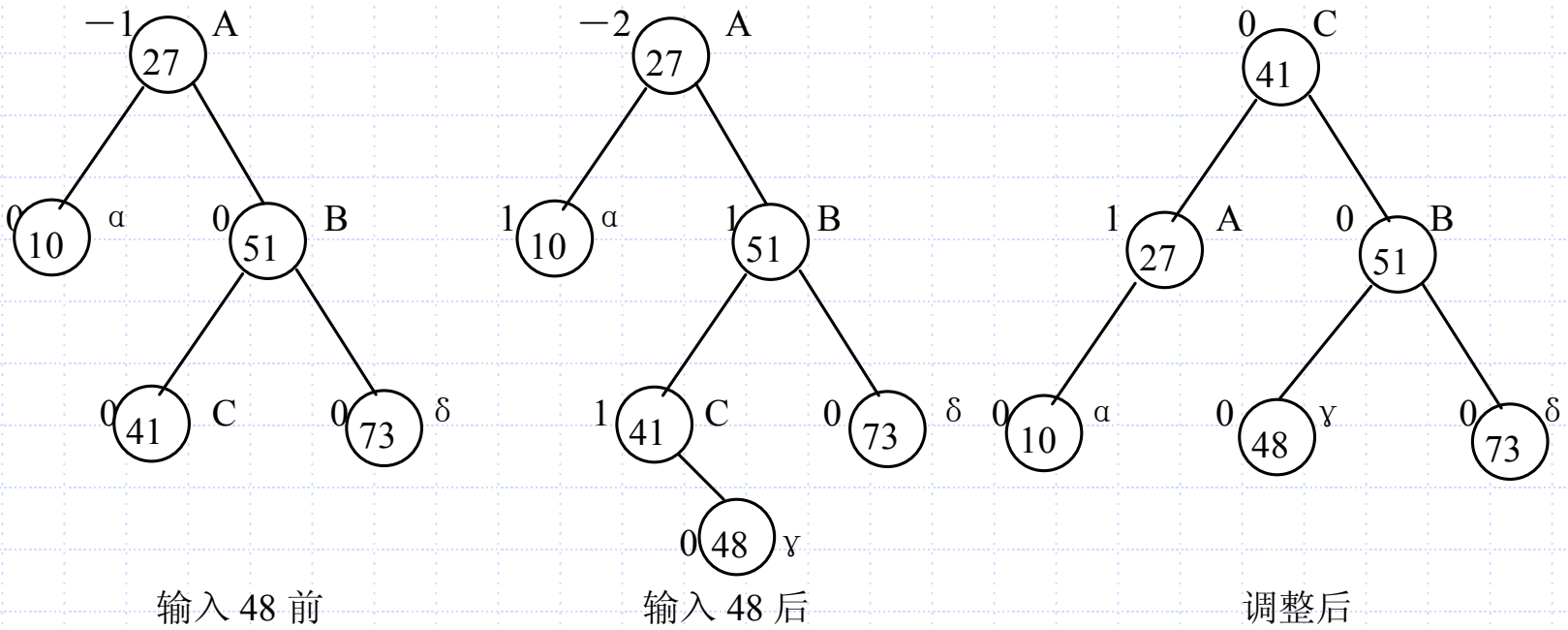
平衡二叉排序树的运算(续)



输入 30 前　　输入 30 后　　调整后

(b) RL(L)型调整

平衡二叉排序树的运算(续)



输入 48 前　　输入 48 后　　调整后

(c) RL(R)型调整

平衡二叉排序树的运算(续)

- 说明
  - 上述所有的调整操作中，A为根的最小不平衡子树的深度在插入结点之前和调整之后相同，对A为根的子树之外的其它结点的平衡性无影响，调整后二叉排序树成为平衡二叉排序树

平衡二叉排序树的运算(续)

- AVL树的结点插入算法：
  - 设在平衡二叉排序树中插入关键码为key的结点，二叉树采用llink-rlink法表示，结构中增加一个字段bf存储结点的平衡因子。算法包括以下几个关键部分：
    - 新结点插入后，若二叉树失去了平衡，则要找到最小不平衡子树
      - 算法中在寻找新结点的插入位置时，始终令A指向离插入位置最近、且平衡因子不为零的结点，如果这样的结点不存在，则A指向根结点；若新结点插入后使AVL树失去平衡，则*A是最小不平衡子树的根，parentA指向*A的父结点。

平衡二叉排序树的运算(续)

- 新结点插入后，要修改一些结点的平衡因子
  - 新结点插入后，会改变*A到新结点路径上各结点的平衡因子，该路径上只有*A的平衡因子不为零，其余结点平衡因子都为零。
  - 因此，从*A的子女*B开始，顺序扫描路径上的结点*p，若新结点插在*p的左子树中，则*p的平衡因子由0变为1；否则，新结点插在*p的右子树中，*p的平衡因子由0变为－1。

平衡二叉排序树的运算(续)

- 判别以*A为根的子树是否失去了平衡
  - 若原*A的左、右子树等高，即原来的平衡因子为0，则新结点插入后，*A的左或右子树高度加1，则插入后不会失去平衡;
  - 若原*A的平衡因子为1或－1，而新结点插在高度较小的子树中，则*A的左、右子树变为等高，令*A的平衡因子为0即可;
  - 若新结点插在高度较高的子树中，则会使AVL树失去平衡，以*A为根的子树即为最小不平衡子树，可根据新结点的插入位置判断进行哪种调整，使插入结点后的树成为新的AVL树

B树表示

- B树的引入
  - 二叉排序树适用于组织较小的、内存中能容纳的字典。对于较大的必须存放在外存贮器上的字典，用二叉排序树组织索引就不合适。
  - 外存文件索引中大量使用的是每个结点包括多个关键码的B树和B+树

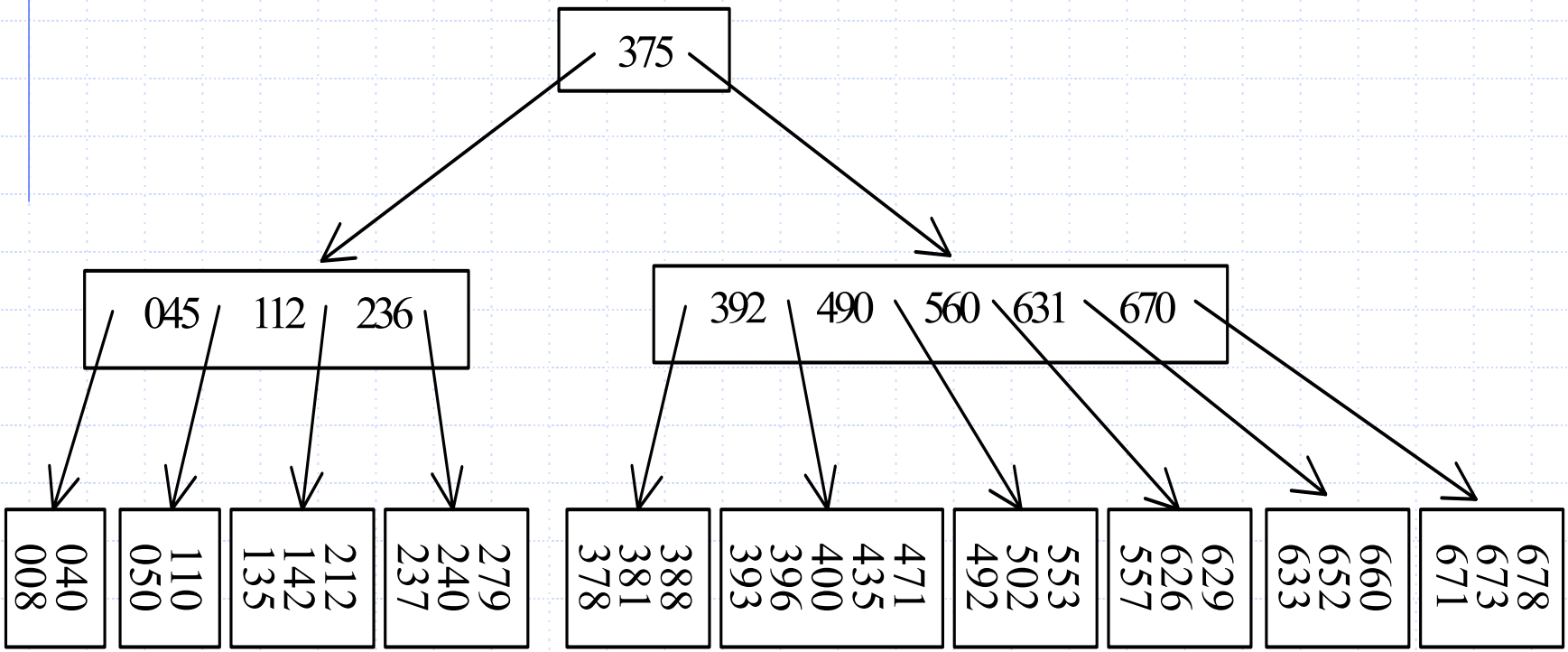
B树表示

- B树的定义
- 一棵m阶的B树满足下列条件：
  - 每个结点至多有m棵子树
  - 除根结点外，其它每个分支至少有⌈m/2⌉棵子树
  - 根结点至少有两棵子树(除非B树只有一个结点)
  - 所有叶结点在同一层上
  - 有j个孩子的非叶结点恰好有j-1个关键码，关键码按递增次序排列。结点中包含的信息为：$(p_0,k_1,p_1,k_2,p_2,\ldots,k_{j-1},p_{j-1})$

    其中，k_i为关键码，且满足$k_i<k_{i+1}$；p_i为指向子树根结点的指针

B树表示

- 如图所示的是一棵6阶的B树



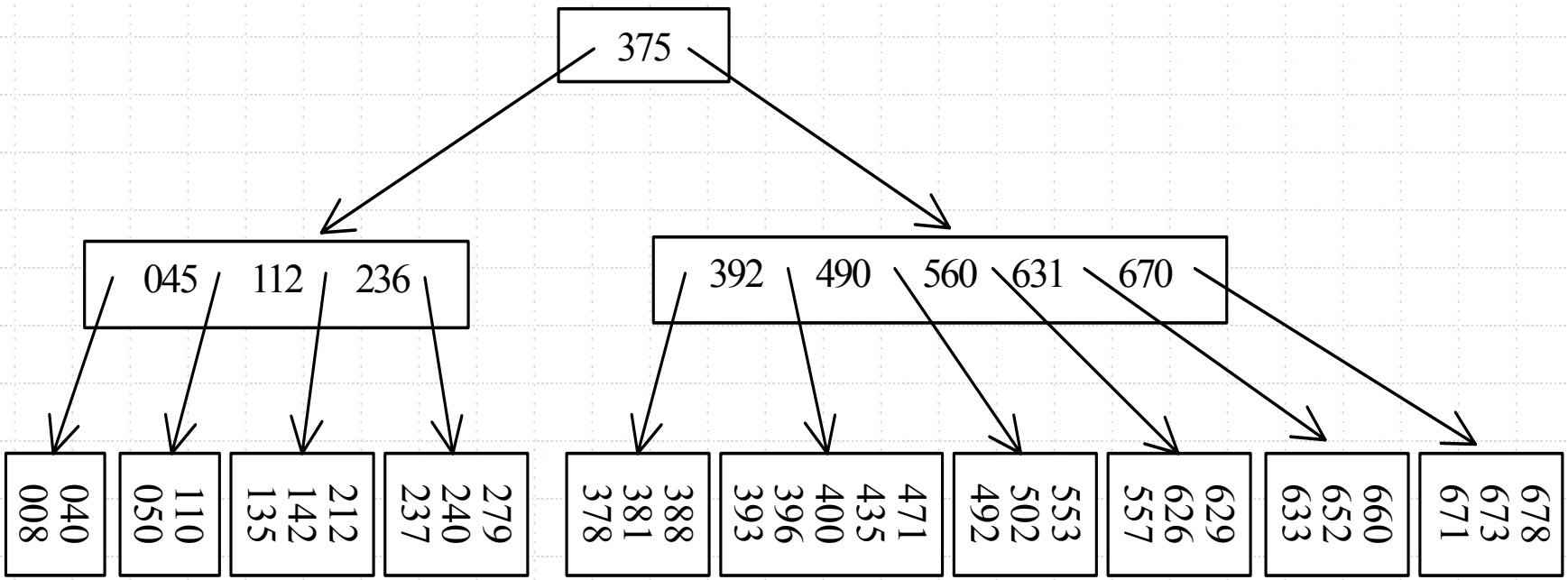
B树的运算

B树的检索

- B树上的检索过程与二叉排序树的相似
- 根据要查找的关键码key，在根结点的关键码集合中进行顺序或二分法检索，若key=k_i，则检索成功；
- 否则，key一定在某k_i和k_{i+1}之间，取指针p_i所指结点继续查找，重复上述检索过程，直到检索成功，或指针p_i为空，检索失败。

B树的运算(续)

- 在下图的B树中检索关键码为100的结点
  - 首先从根结点开始，100<375，进入p0所指的子树
  - p0所指的结点包含三个关键码(045,112,236)，045<100<112，进入p1所指的子树
  - 在结点中检索，050<100<110，而此时p1=NULL，说明此B树中不存在关键码100，检索失败



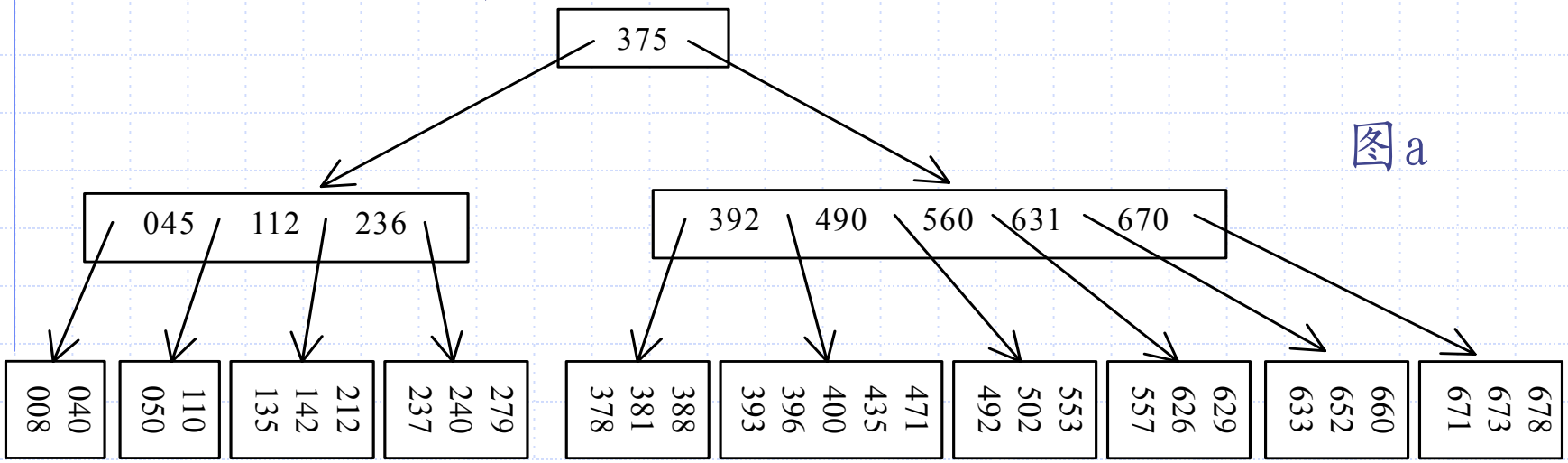
B树的运算(续)

B树的插入

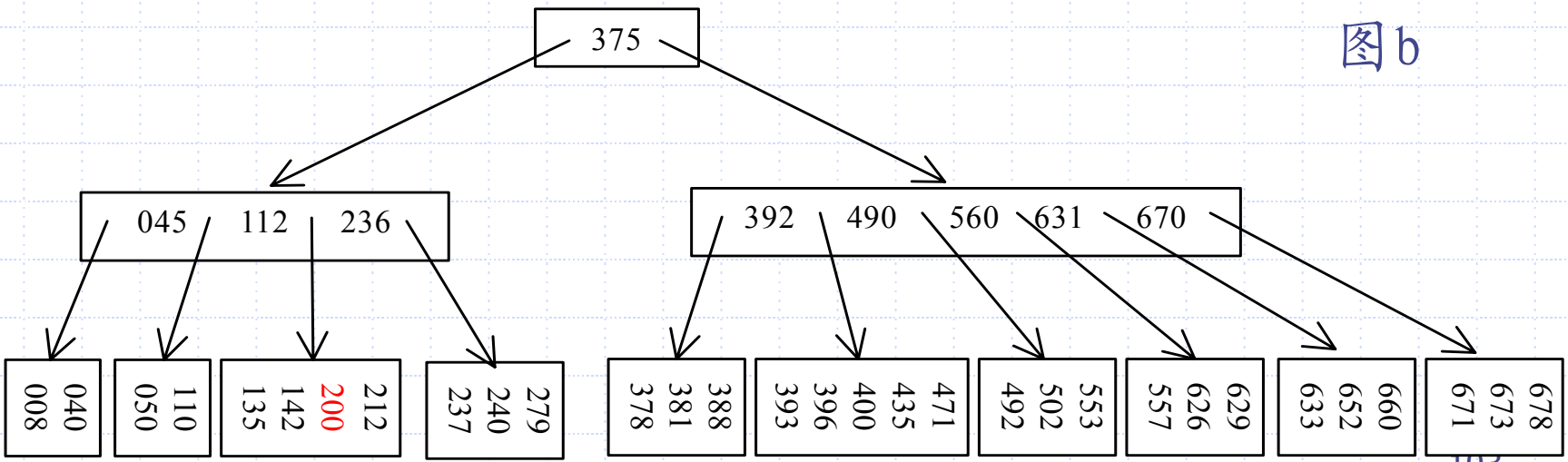
- 深度为h的m阶B树，结点一般是插在第h层首先检索到第h层，确定关键码应插入的结点
  - 若该结点中关键码个数小于m-1，则直接插入即可

B树的运算(续)

- 如在图a的B树中插入关键码200，插入后的B树如图b所示



图a



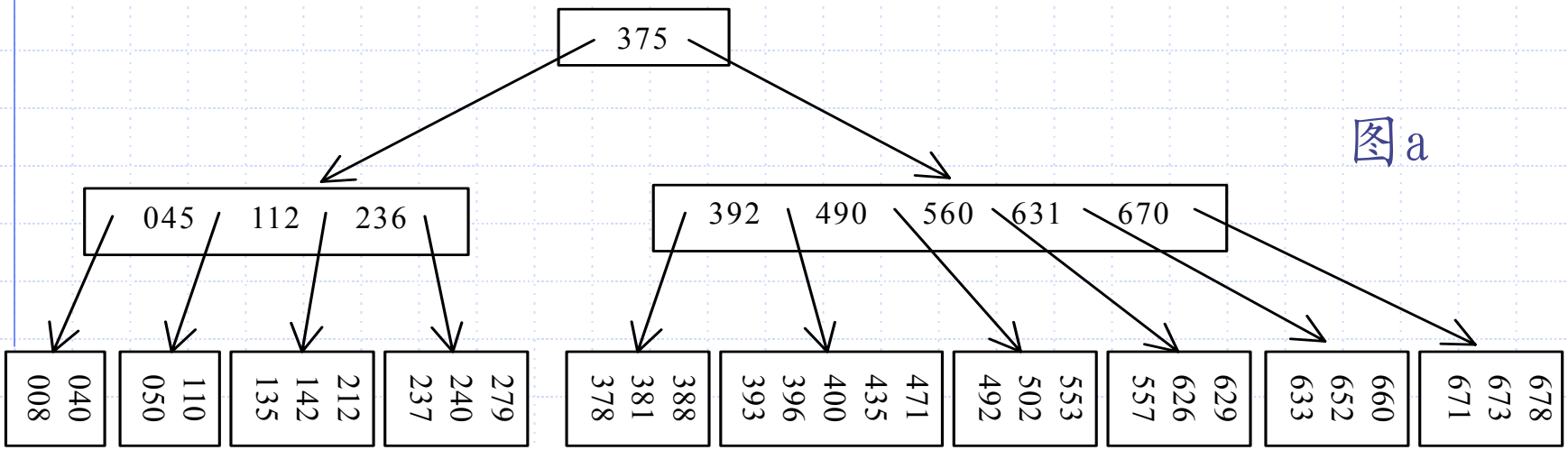
图b

B树的运算(续)

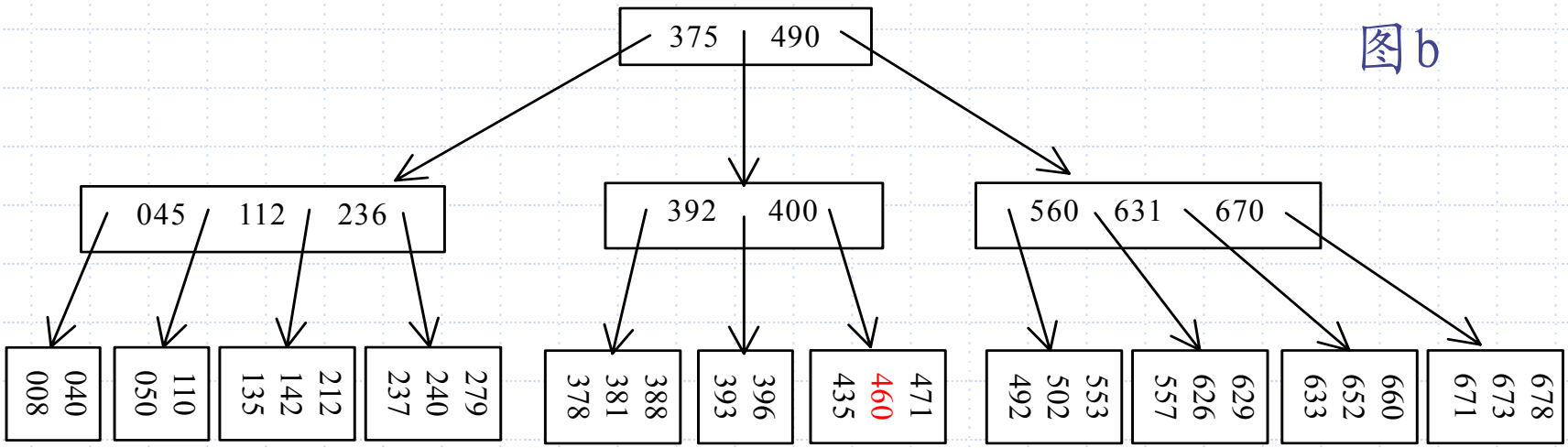
- 若该结点中关键码个数等于m-1，则将引起结点的分裂
  - 以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点中
  - 若父结点中关键码数也为m-1，则需要再次分裂
  - 最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层

B树的运算(续)

- 例:在图a中插入460后的B树如图b



图a



图b

B树的运算(续)

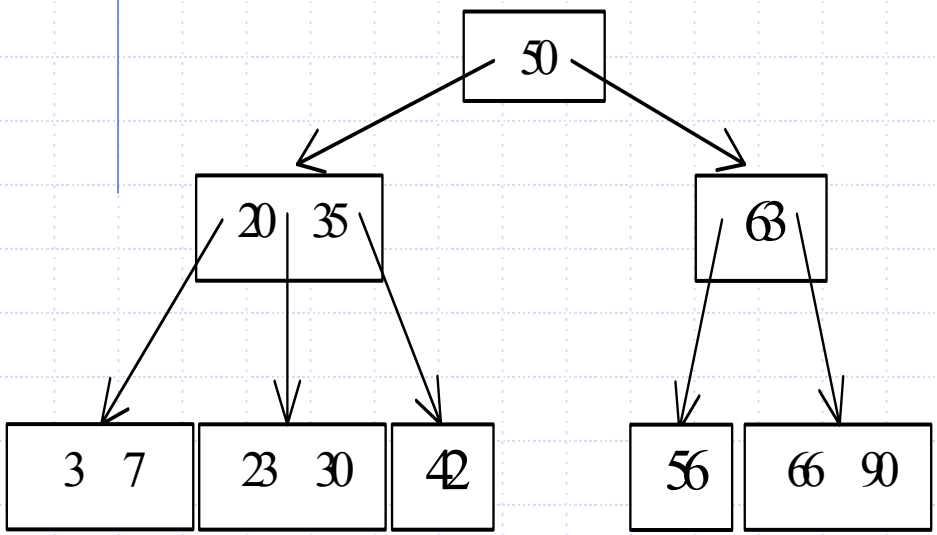
B树的删除

- 在深度为h的m阶B树中删除一个关键码，首先检索到该关键码所在的结点，然后根据不同情况进行删除

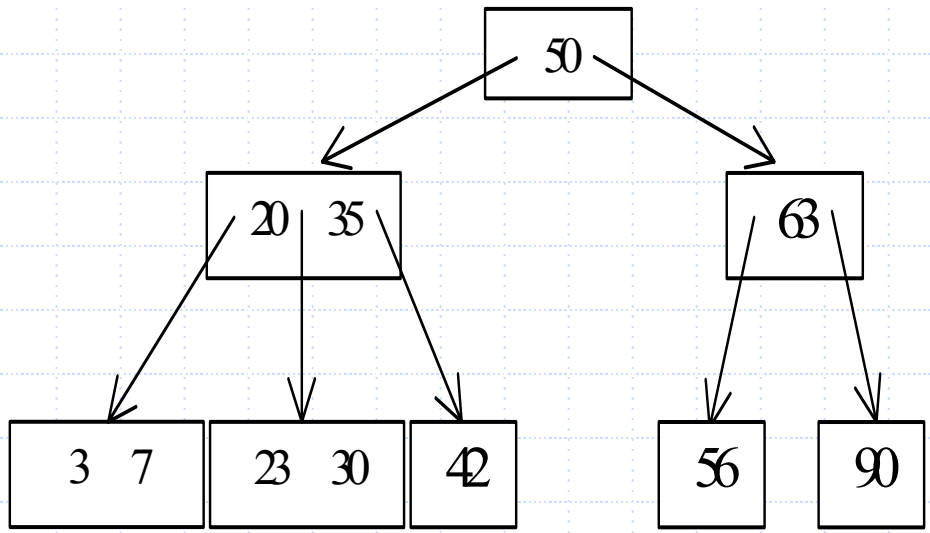
- 若结点在第h层，且结点中关键码个数大于$\lceil m/2 \rceil - 1$，则直接删除即可

B树的运算(续)

- 如在图 (a)的3阶B树中删除关键码66，删除后的B树如图 (b)所示



图a



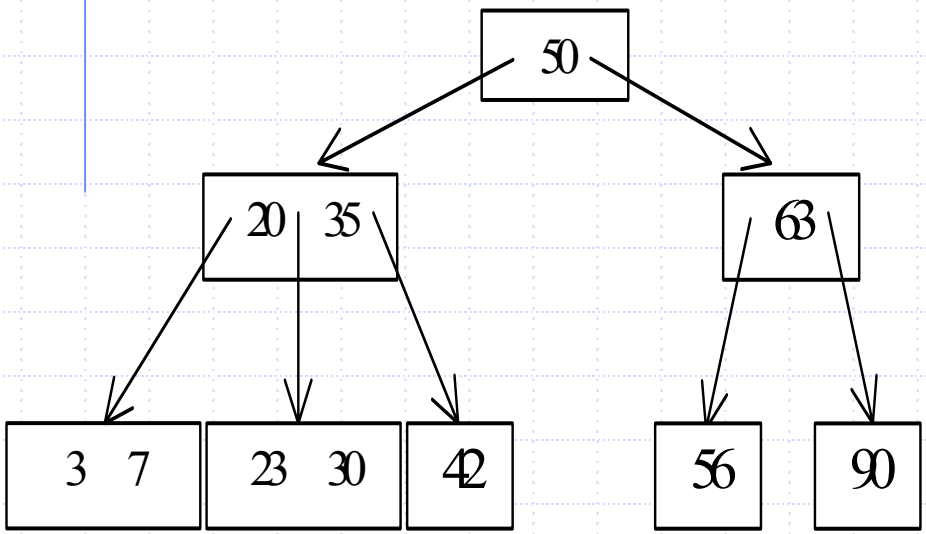
图b

B树的运算(续)

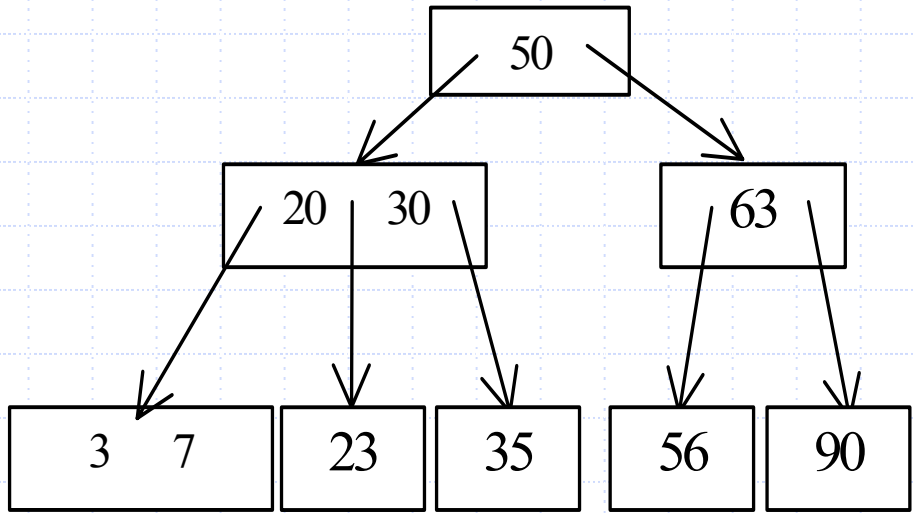
- 若结点在第h层，且关键码个数等于⌈m/2⌉-1 ，结点左(或右)兄弟结点的关键码个数大于⌈m/2⌉-1
  - 则把左(或右)兄弟结点中最大(或最小)的关键码移到父结点中，并将父结点中大于(或小于)其值的关键码移到被删除关键码所在的结点中

B树的运算(续)

- 如在图(b)的B树中删除关键码42，删除后的B树如图(c)所示



图b



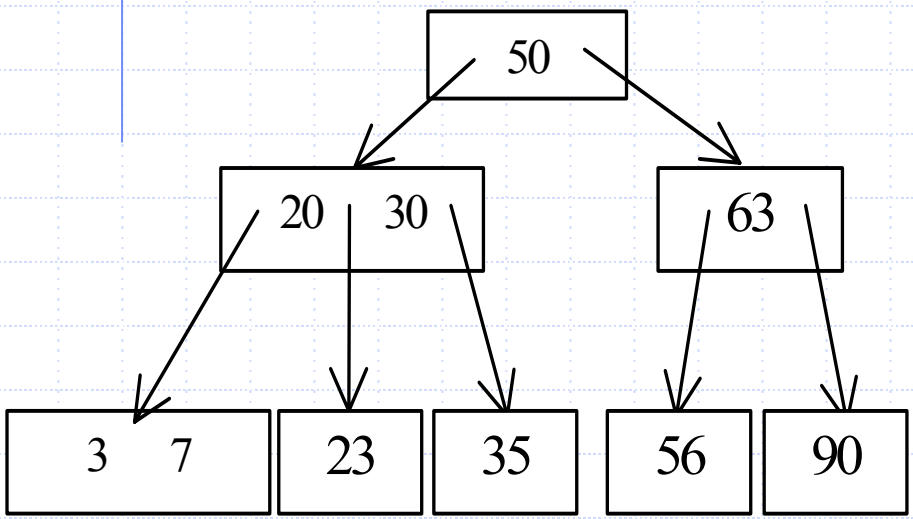
图c

B树的运算(续)

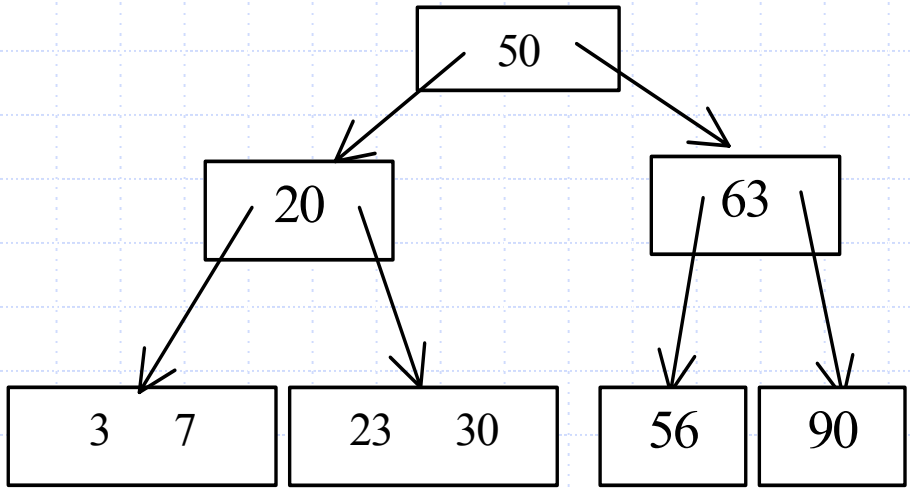
- 若结点在第h层，且关键码个数及结点左、右兄弟中的关键码个数都等于$\lceil m/2 \rceil - 1$ ，则需要合并该结点、其兄弟结点及父结点中的一个关键码
  - 假设其有右兄弟，且父结点中由指针p_i指向其右兄弟，则将该结点剩下的关键码加上其父结点中k_i以及p_i所指结点中的关键码合并，从父结点中删除k_i
  - 由于删除k_i，可能引起父结点进行同样的调整，这种调整可能一直延续到根结点，而使整个B树减少一层

B树的运算(续)

- 如在图 (c)的B树中删除关键码35，删除后的B树如图 (d)所示



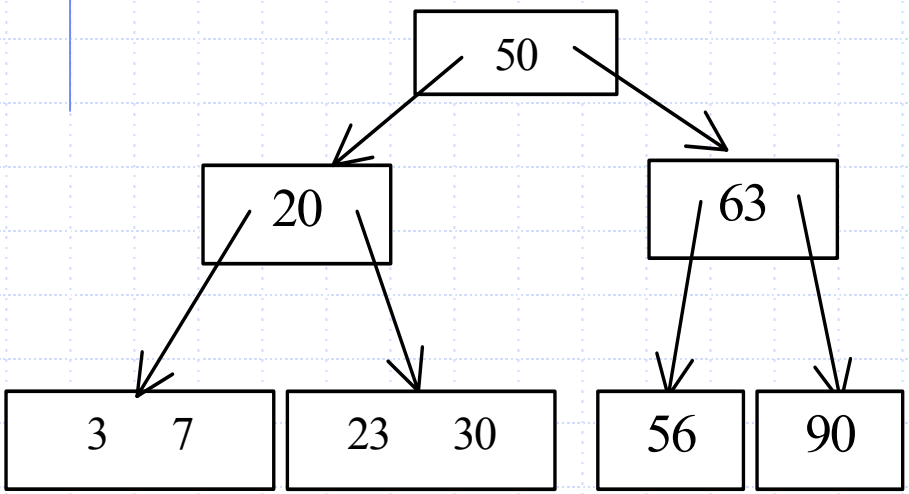
图c



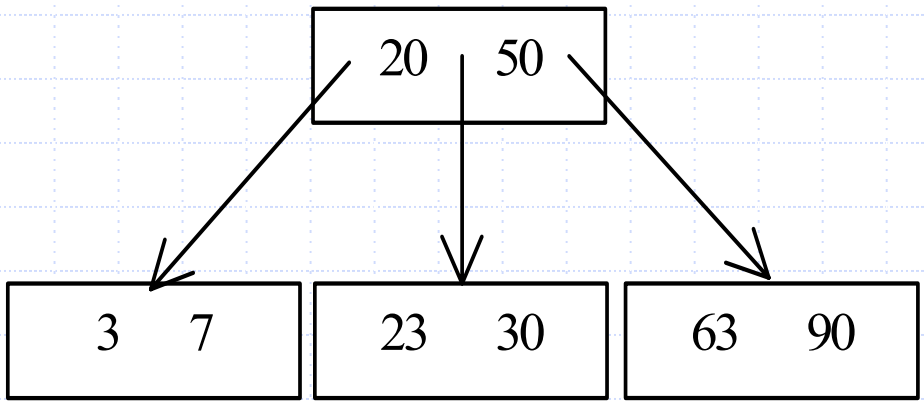
图d

B树的运算(续)

- 以及在图 (d)的B树中删除关键码56后的B树如图 (e)所示



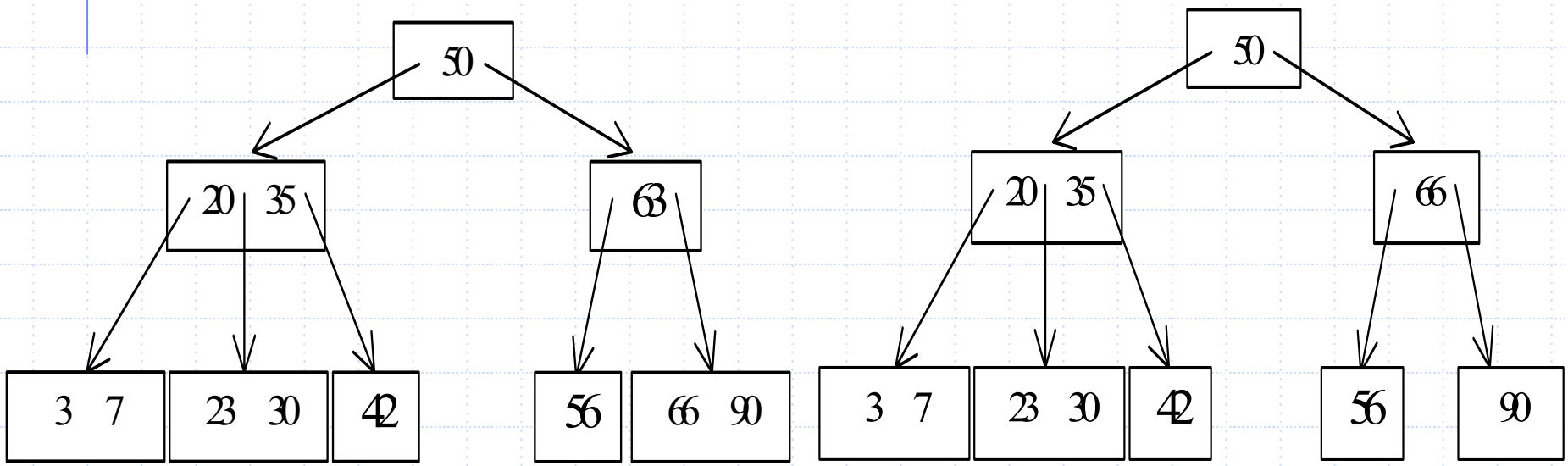
图d



图e

B树的运算(续)

- 若结点的层数小于h，设删除的关键码为结点中的第i个关键码k_i，则用p_i所指的结点中的最小关键码k代替k_i，然后再删除关键码k



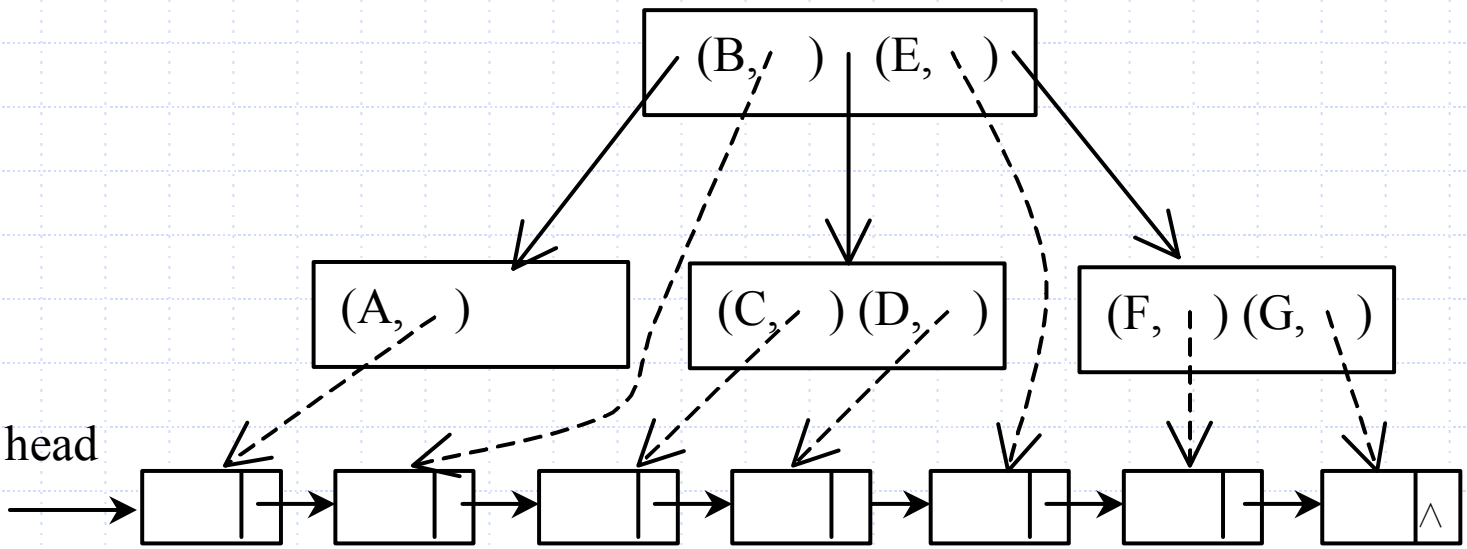
删除63

B树的应用

- B树在索引文件中的应用
- 为了给出一种用B树来组织索引文件的方法，做如下规定：
  - B树的每个结点存放在外存储器的一个页块上
  - B树中任何结点内的一个关键码实际上是一个索引项。即由一个关键码k和指针p所组成的二元组(k, p)。p是指向主文件页块(或主文件记录)的指针，它在B树中没有画出，但总是伴随着关键码隐式存在。
- 这样，整个B树构成了文件的索引

B树的应用

- 图是一个由3阶B树作稀疏索引的索引顺序文件
  - 图中给出了B树的每个关键码所代表的索引项，用虚线表示每个索引项中指向主文件页块的指针
  - 注意，这些索引项分布在B树的各个层次上，而不仅仅在树叶那一层



作业

- 书面
  - 画出用第二种方法删除二叉排序树结点的程序流程图
  - 求由结点13，24，37，90，53，5，2，12，65，9所构成的平衡二叉树(要求按照所给序列的次序依次生成)
  - 画出AVL树的结点插入算法的流程图
- 上机
  - 实现二分查找法
  - 采用双散列法解决冲突，实现散列表的检索与插入（可采用书P168例子的函数）