



数据结构与算法

第十二章 高级树结构

任课教员：张 铭

<http://db.pku.edu.cn/mzhang/DS/>

mzhang@db.pku.edu.cn

北京大学信息科学与技术学院

网络与信息系统研究所

©版权所有，转载或翻印必究

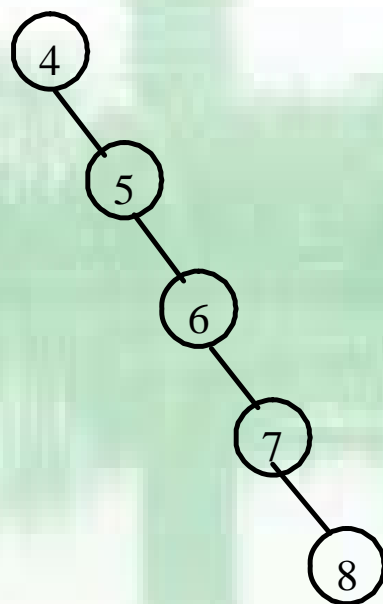


主要内容

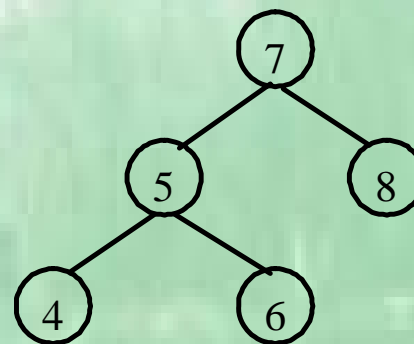
- 12.1 Trie和Patricia 结构
- 12.2 改进的BST
 - 最佳二叉搜索树
 - AVL树
 - 伸展树
- 12.3 空间树结构
- 12.4 决策树和博弈树

12.1 Trie结构和Patricia树

- 引子：BST（二叉搜索树）不是平衡的树
 - 树结构与输入数据的顺序有很大的关系



输入顺序：4、5、6、7、8



输入顺序：7、5、4、6、8

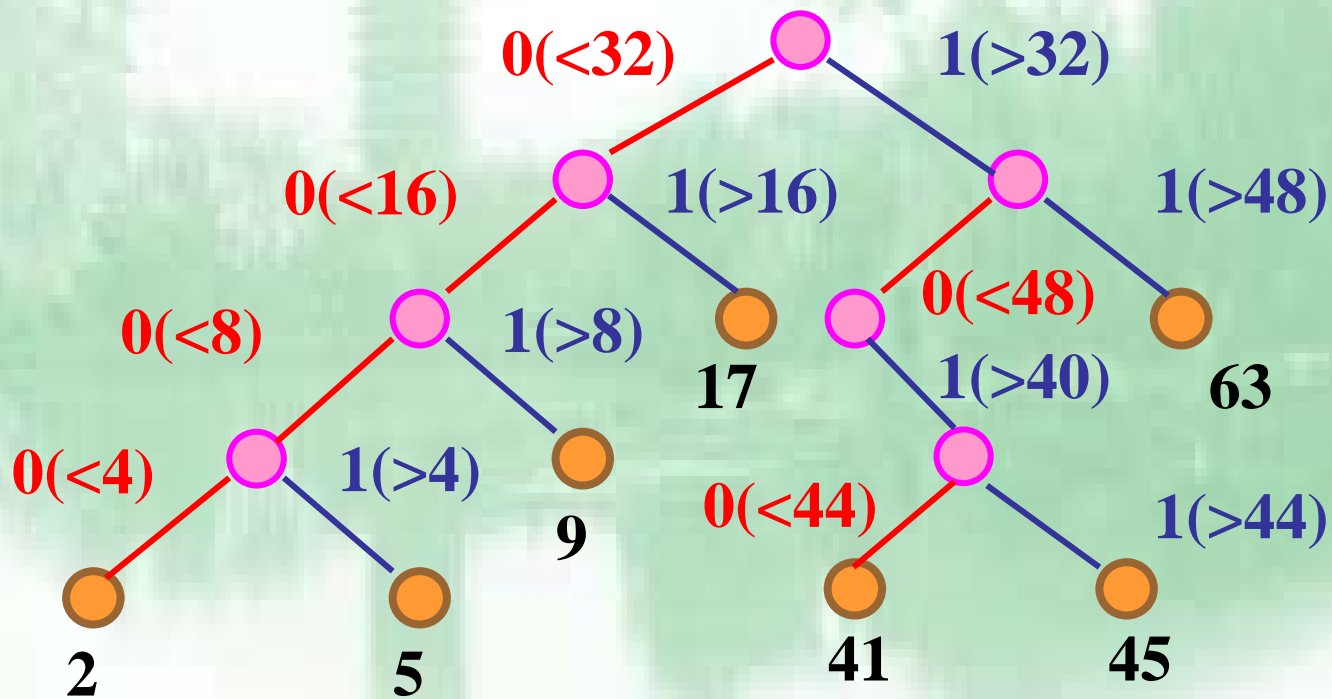


Trie结构

- 关键码对象空间分解
- “trie”这个词来源于“**retrieval**”
- 主要应用
 - 信息检索 (**information retrieval**)
 - 自然语言大规模的英文词典
- 二叉Trie树
 - 用每个字母的二进制编码来代表
 - 编码只有**0**和**1**

二叉Trie结构

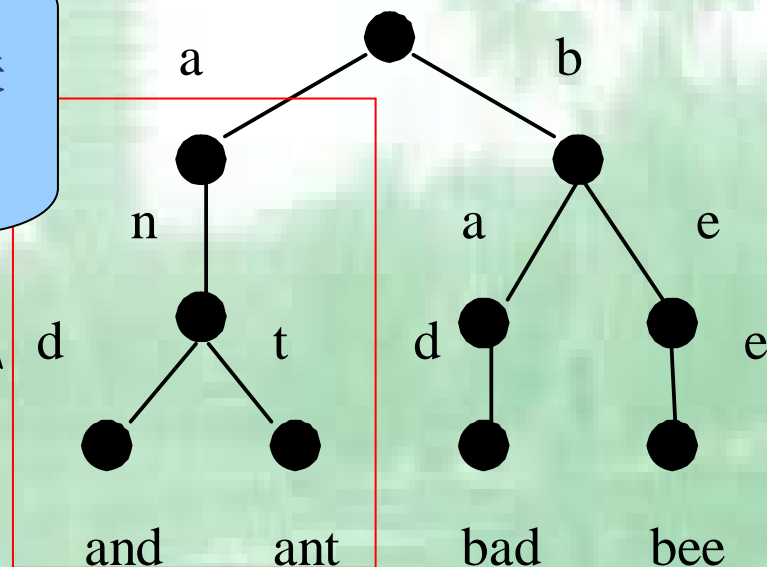
元素为2、5、9、17、41、45、63



英文字符树——26叉Trie

存单词 and、ant、bad、bee

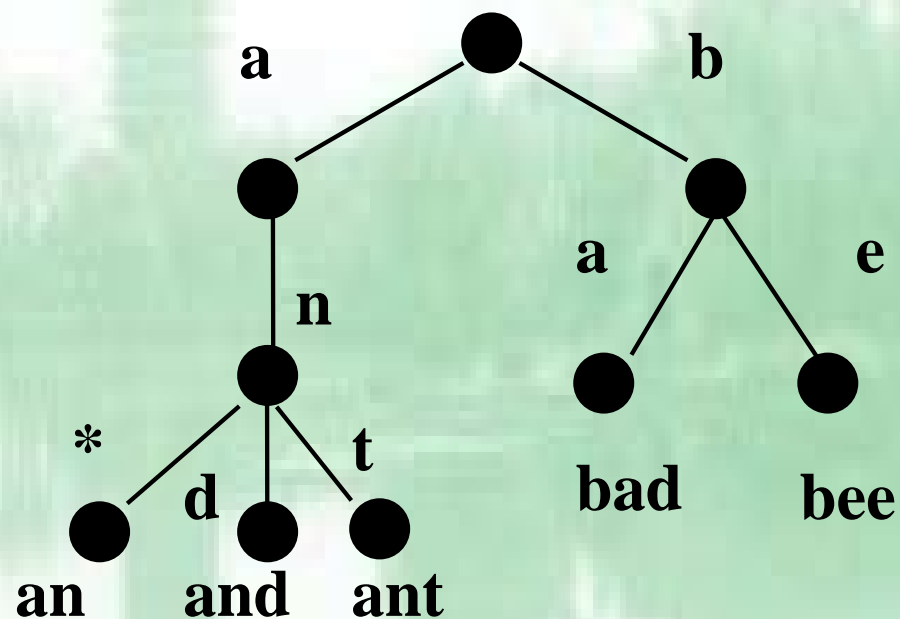
“an”子树代表具有相同前缀an-的关键码集合{and, ant}



- 一棵子树代表具有相同前缀的关键码的集合

字符树的改进（续）

存储单词 **an**、**and**、**ant**、**bad**、**bee**





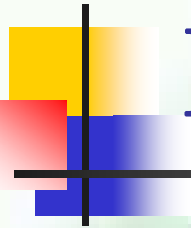
Trie字符树的特点

- **Trie结构也不是平衡的**
 - t子树下的分支比z子树下的多
 - **26个分支因子——庞大的26叉树**

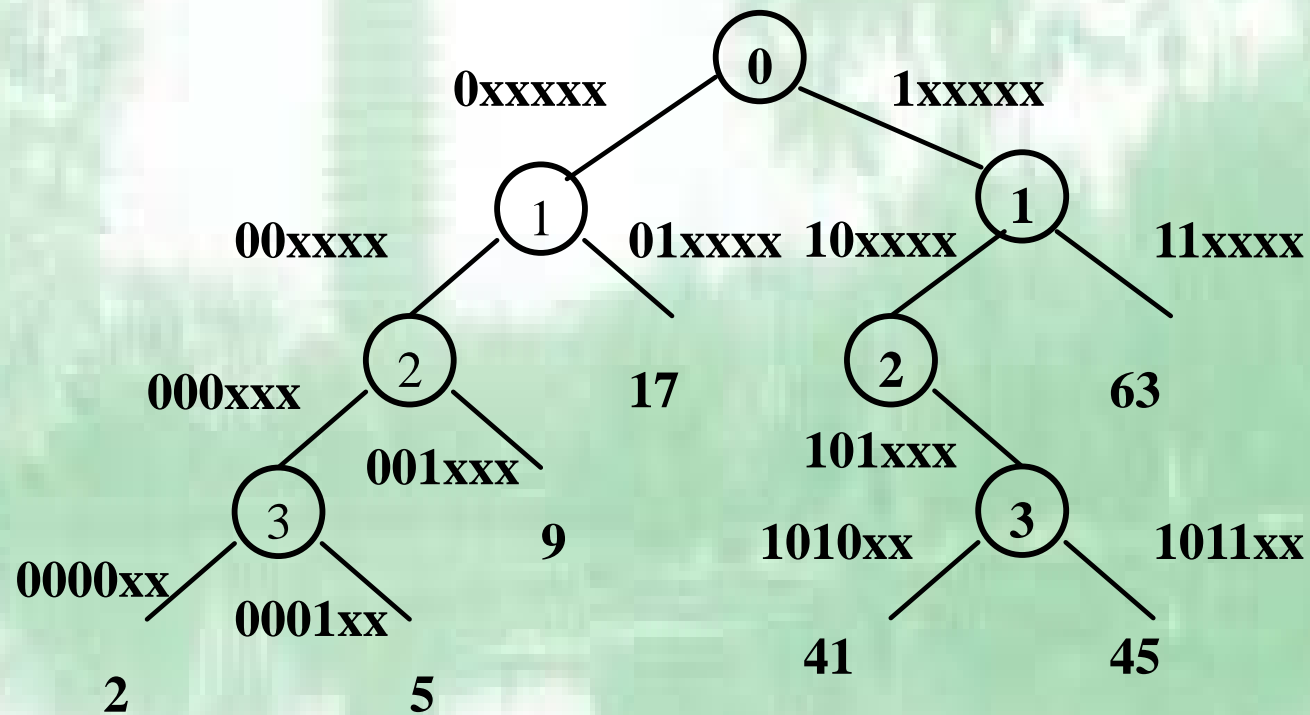


PATRICIA 结构

- **“Practical Algorithm To Retrieve Information Coded In Alphanumeric”**
- **D.Morrison发明的Trie结构变体**
 - 根据关键码二进制位的编码来划分
 - 二叉Trie树

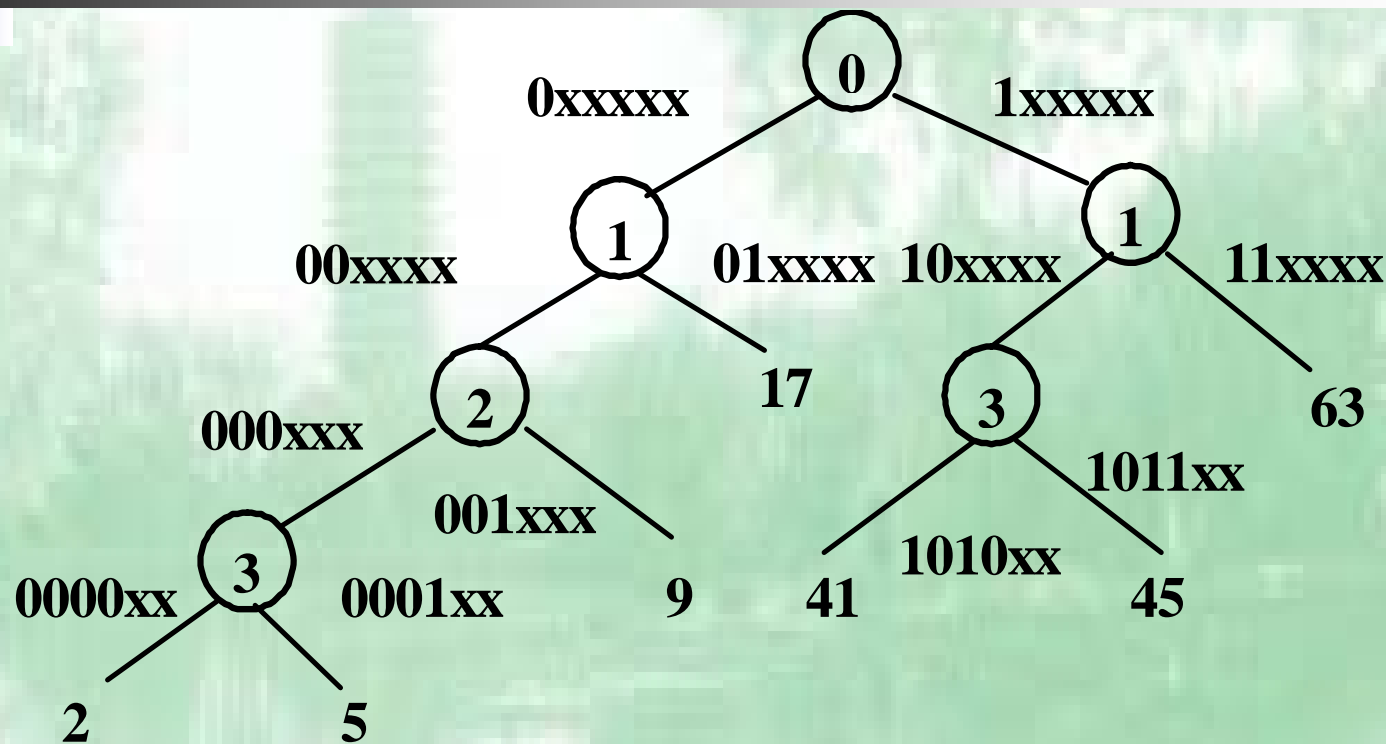


PATRICIA 结构图



编码: 2: 000010 5: 000101 9: 001001
17: 010001 41: 101001 45: 101101 63: 111111

压缩PATRICIA 结构



编码: 2: 000010 5: 000101 9: 001001
17: 010001 41: 101001 45: 101101 63: 111111



PATRICIA的特点

- 改进后的压缩PATRICIA树是满二叉树
 - 每个内部结点都代表一个位的比较
 - 必然产生两个子结点
- 一次检索不超过关键码的位个数



后缀树(Suffix Tree)

- 后缀树是表示一个字符串 **S** 所有后缀串的树
- 结点表示开始的字符(或压缩字符串)
- 边标注为子串——该字符串在原串中的起止位置
 - 边表示不同字符分支
 - 所有根到树叶结点的路径，可以表示串 **S** 的所有后缀串
- 通俗地说：
 - 一个字符串的所有后缀
 - 这些后缀组成**Trie**
 - 压缩**Trie**，得到字符串的后缀树



MALAYALAM\$ 的后缀

- M A L A Y A L A M
- A L A Y A L A M
- L A Y A L A M
- A Y A L A M
- Y A L A M
- A L A M
- L A M
- A M
- M
- \$

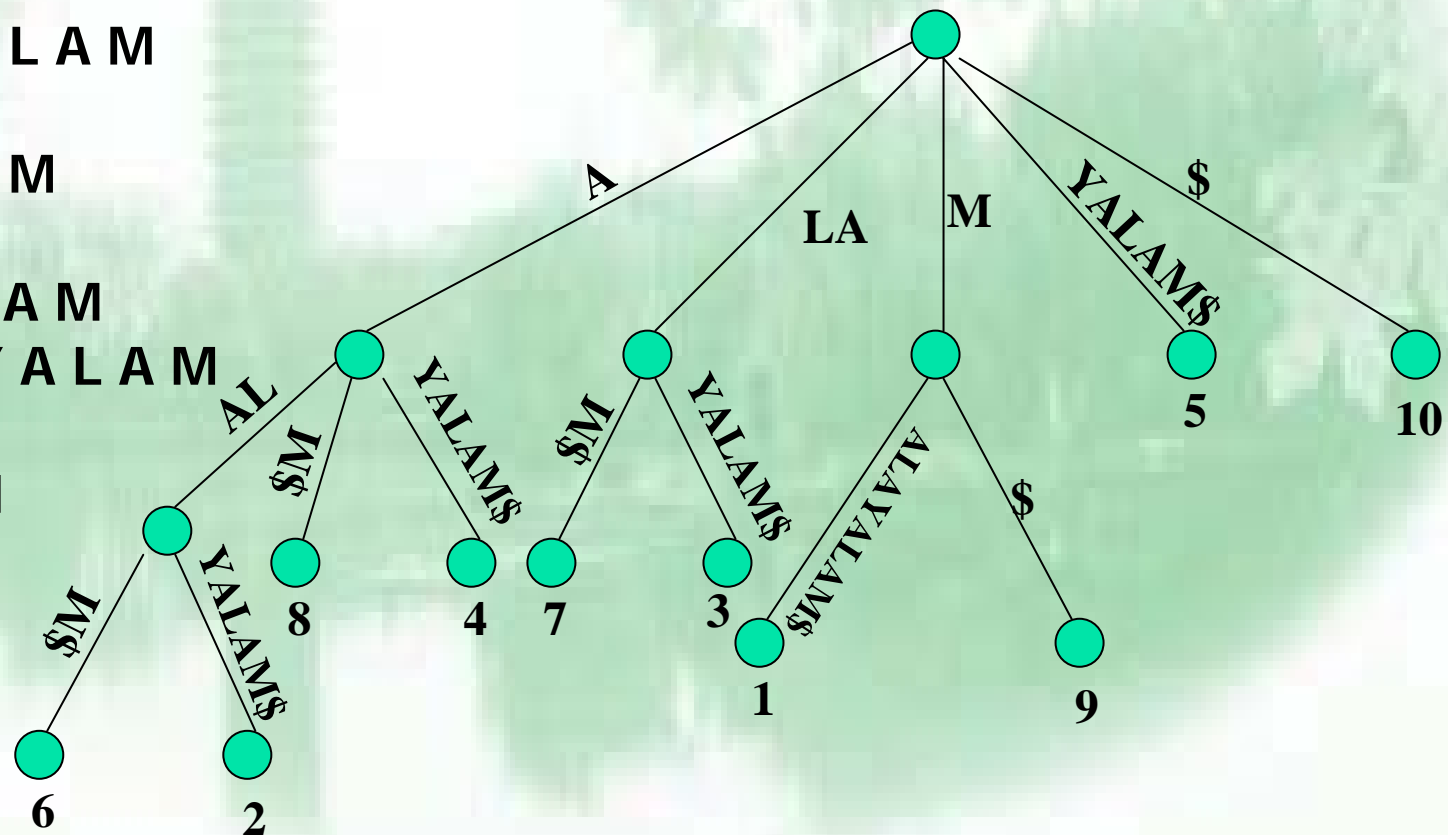
对后缀串排序

- A L A M
- A L A Y A L A M
- A M
- A Y A L A M
- L A M
- L A Y A L A M
- M A L A Y A L A M
- M
- Y A L A M
- \$

后缀树

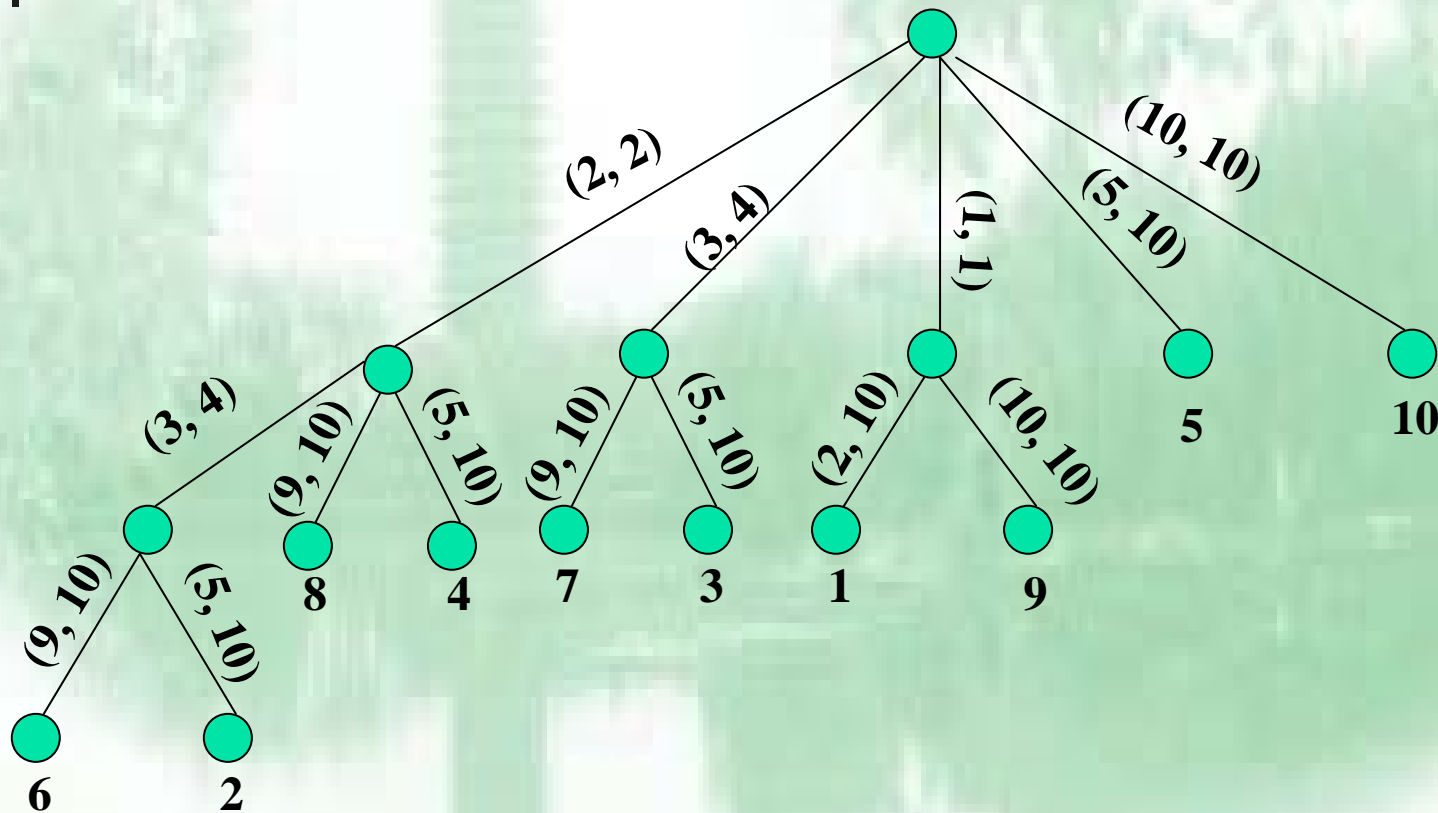
S = M A L A Y A L A M \$
1 2 3 4 5 6 7 8 9 10

- A L A M
- A L A Y A L A M
- A M
- A Y A L A M
- L A M
- L A Y A L A M
- M A L A Y A L A M
- M
- Y A L A M
- \$



边信息

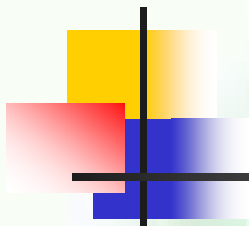
S = M A L A Y A L A M \$
1 2 3 4 5 6 7 8 9 10





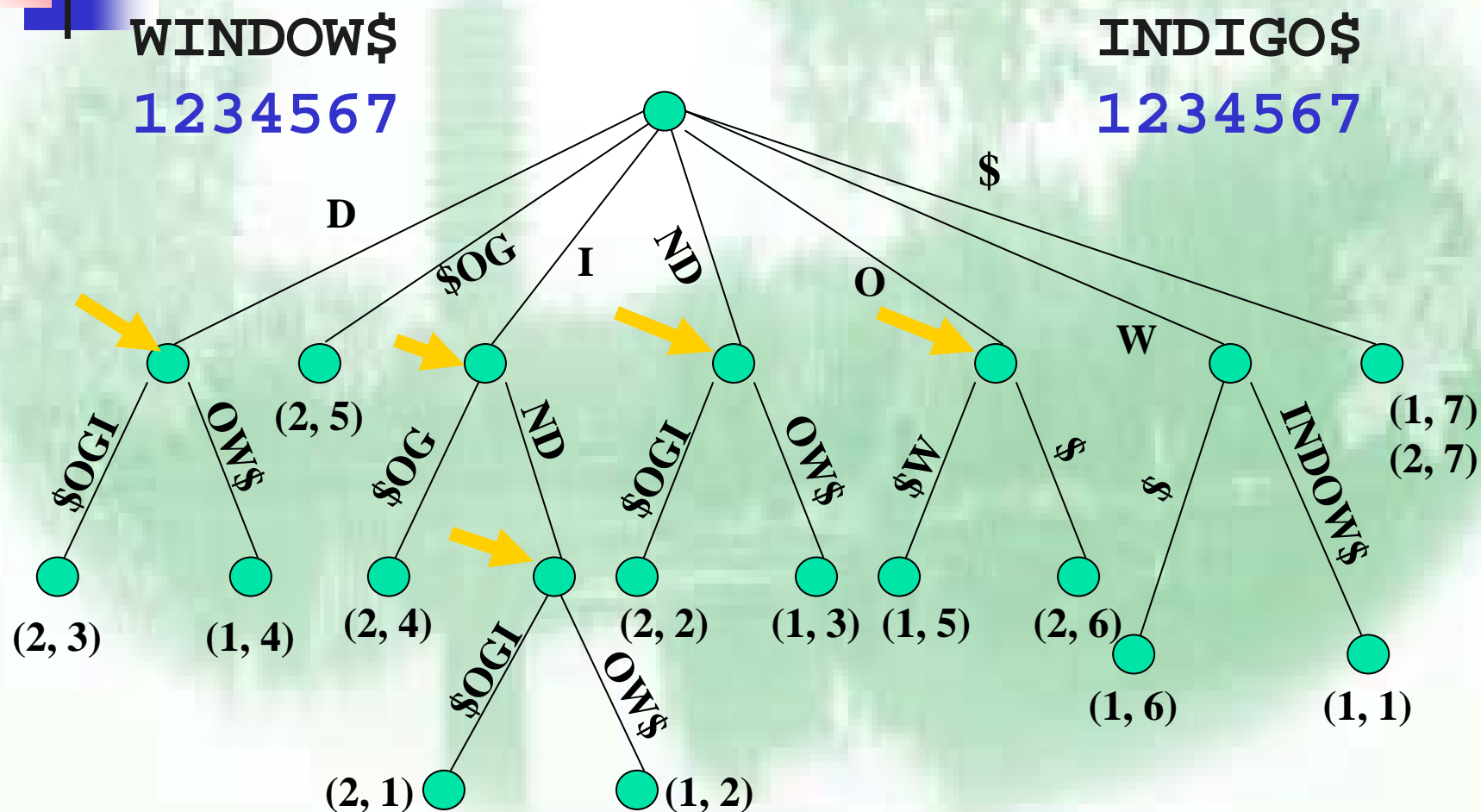
建树算法

- 对于长度为 n 的语料建立后缀树，直接的方法时间复杂度为 $O(n^2)$
- 1973年Weiner提出线性时间算法
- 1976年McCreigh提出更节约内存的算法
- 1995年Ukkonen提出线性时间建树算法



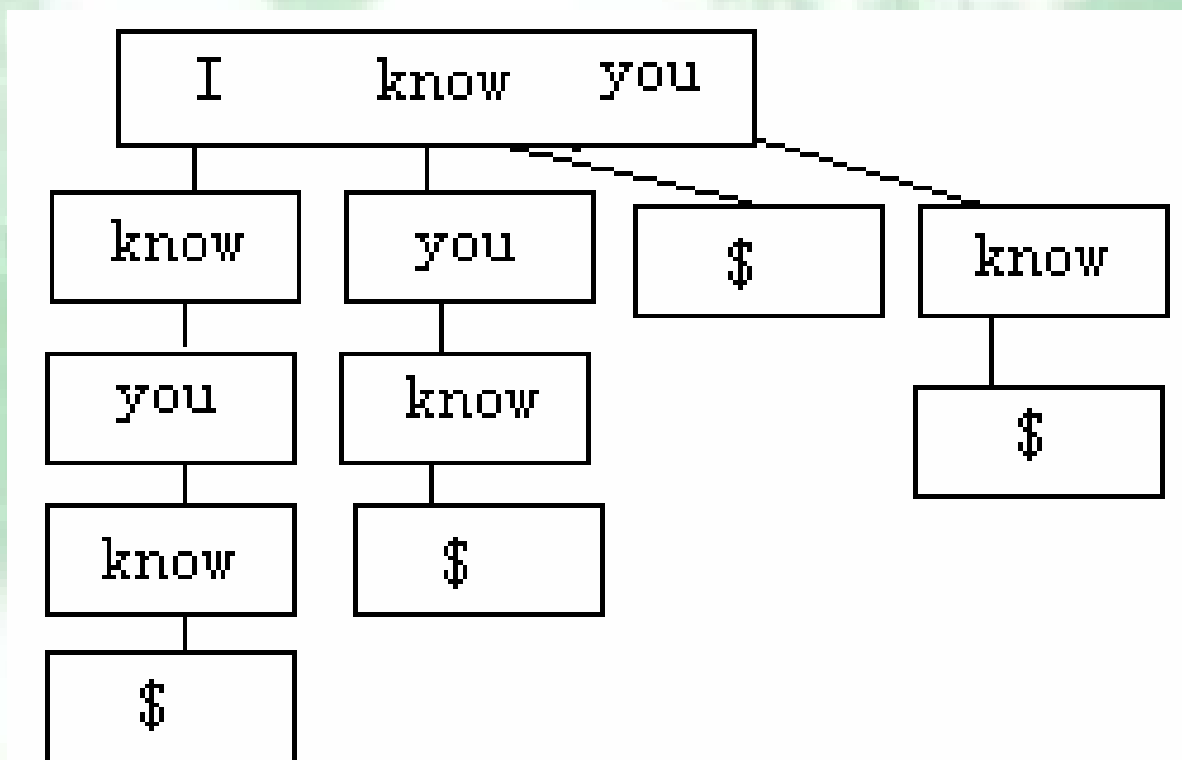
- 对于长度为 n 的字符串建立后缀树，直接的方法时间复杂度为 $O(n^2)$
- 1973年Weiner提出线性时间算法
- 1976年McCreigh提出更节约内存的算法
- 1995年Ukkonen提出线性时间建树算法

GST——通用后缀树(Generalized)



单词粒度的后缀树

- “I know you know \$ ”





后缀树的应用

- 查找字符串中的子串
- 统计**S**中出现**T**的次数
- 找出**S**中最长的重复子串
 - 出现了两次以上的子串
- 两个字符串的公共子串
- 最长共同前缀(**LCP**)
- 回文串



后缀树的应用

- 中文切词
- 关联分析
 - 发现经常共同出现的短语
- 频繁模式挖掘
- **STC** 聚类
- 基因/蛋白序列对比/分类
-



后缀数组

- 字符串**S**的后缀数组**SA**
 - 对**S**的所有后缀的指针排序
 - 即后缀树叶结点的字典序
- 后缀树**ST** = 后缀数组**SA** + **LCP**数组

数组 (Suffix Array)

M A L A Y A L A M \$
1 2 3 4 5 6 7 8 9 10

6	2	8	4	7	3	1	9	5	10
---	---	---	---	---	---	---	---	---	----

后缀数组

3	1	1	0	2	0	1	0	0	-
---	---	---	---	---	---	---	---	---	---

最长公共前缀数组

后缀6和2共享 "ALA"

后缀2和8共享 "A"

6	ALAM\$
2	ALAYALAM\$
8	AM\$
4	AYALAM\$
7	LAM\$
3	LAYALAM\$
1	MALAYALAM\$
9	M\$
5	YALAM\$
10	\$

LCP总是相邻的



代价分析

- 目标长 $n=|S|$, 模式长 $m=|P|$
- 空间代价
 - ST $20n$
 - SA $4n$
- 建数据结构时间代价
 - ST $O(n)$
 - SA $O(n \log n)$
- 查找子串时间代价
 - ST $O(m)$
 - SA $O(m \log n)$



后缀树小结

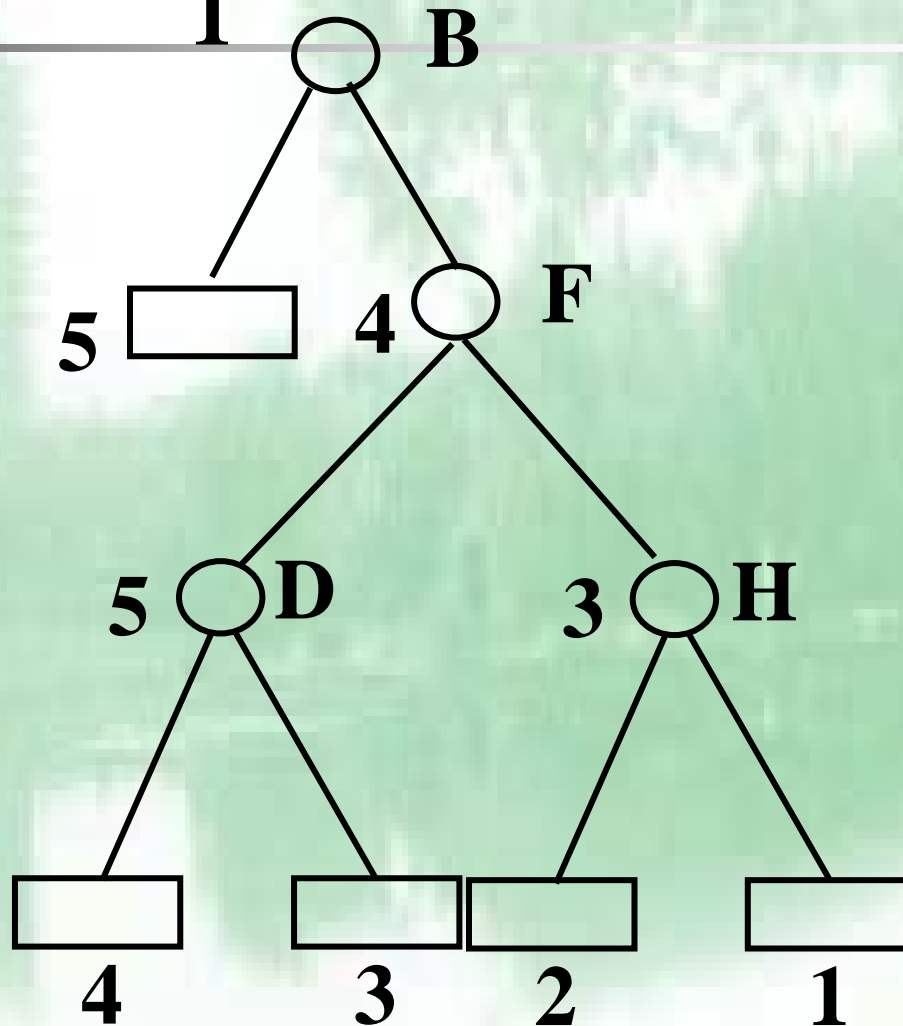
- 后缀树和后缀数组提供了很好的全索引结构
 - 适合于各种字符串算法
- 大量后缀树的变种
 - 尽力减少其空间消耗



12.2 二叉树结构的改进

- 最佳二叉排序树
- 平衡的二叉搜索树
- 伸展树

$$ASL(n) = \frac{1}{W} \left[\sum_{i=1}^n p_i (1_i + 1) + \sum_{i=0}^n q_i l'_i \right]$$





最佳二叉搜索树的动态规划

- 最佳子结构、重复子结构
 - 任何子树都是最佳二叉搜索树
- 动态规划过程
 - 第一步：构造包含1个结点的最佳二叉搜索树
 - 找 $t(0, 1)$, $t(1, 2)$, ..., $t(n-1, n)$
 - 第二步构造包含2个结点的最佳二叉搜索树
 - 找 $t(0, 2)$, $t(1, 3)$, ..., $t(n-2, n)$
 - 再构造包含3, 4, ...个结点的最佳二叉搜索树
 - 最后构造 $t(0, n)$



最佳二叉搜索树 $t(i, j)$

- 包含关键码 $\text{key}_{i+1}, \text{key}_{i+2}, \dots, \text{key}_j$ 为内部结点 ($0 \leq i \leq j \leq n$)
- 结点的权为 $(q_i, p_{i+1}, q_{i+1}, \dots, p_j, q_j)$,
- 根为 $r(i, j)$
- 开销为 $C(i, j)$, 即
$$\sum_{x=i+1}^j p_x (l_x + 1) + \sum_{x=i}^j q_x l'_x$$
- 权的总和为 $W(i, j) =$
$$p_{i+1} + \dots + p_j + q_i + q_{i+1} + \dots + q_j$$



- 以 key_k 为根

- 左子树包含 $\text{key}_{i+1}, \dots, \text{key}_{k-1}$

- $C(i, k-1)$

- 右子树包含 $\text{key}_{k+1}, \text{key}_{k+2}, \dots, \text{key}_j$

- $C(k, j)$ 已求出

- $C(i, j) =$

$$W(i, j) + \min_{i < k \leq j} (C(i, k-1) + C(k, j))$$

i \ j					
	0	1	2	3	4
0	0	1	2	2	2
1		0	2	2	3
2			0	3	3
3				0	4
4					0

$r(i, j)$

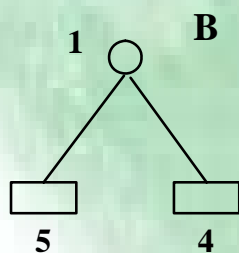
i \ j					
	0	1	2	3	4
0	0	10	28	43	57
1		0	12	27	40
2			0	9	19
3				0	6
4					0

$C(i, j)$

i \ j					
	0	1	2	3	4
0	5	10	18	21	28
1		4	12	18	22
2			3	9	3
3				3	6
4					1

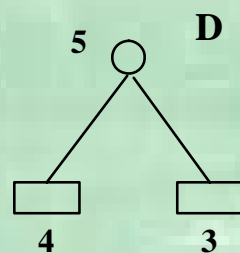
$W(i, j)$

第一步



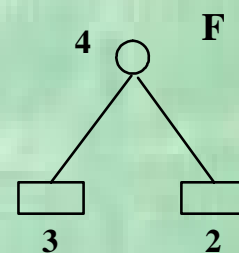
10

10



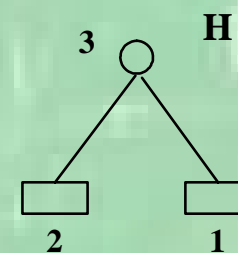
12

12



9

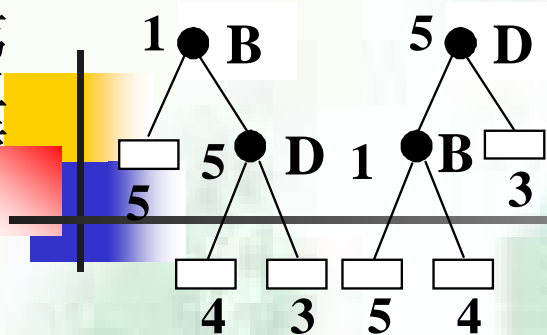
9



6

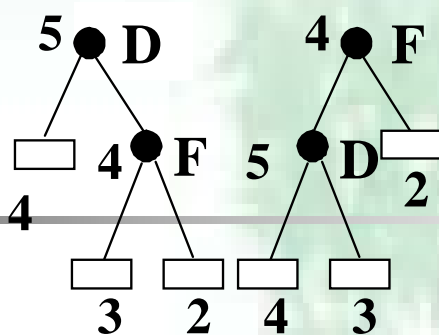
6

第二步



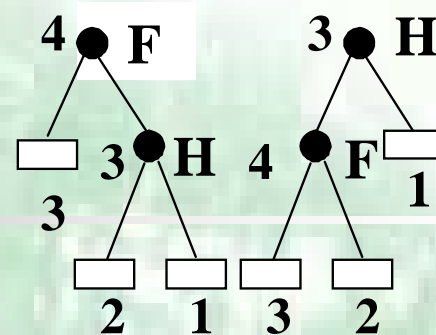
开销 30
总权 18

28
18



27
18

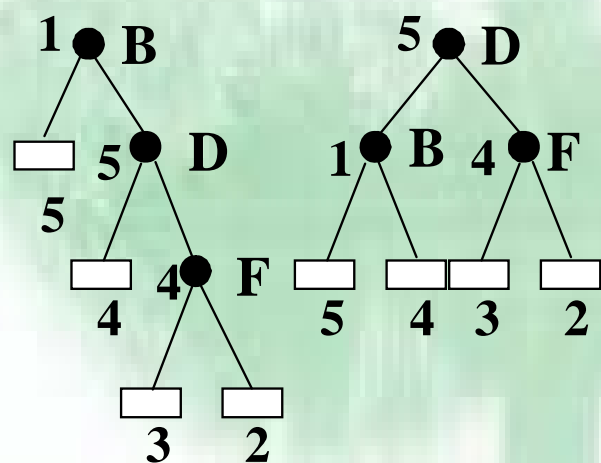
30
18



19
13

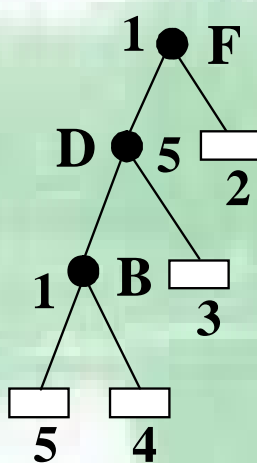
22
13

第三步

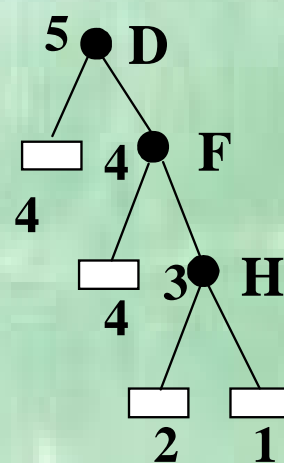


花费 51
总权 24

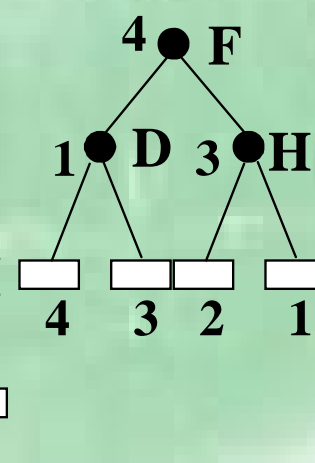
43
24



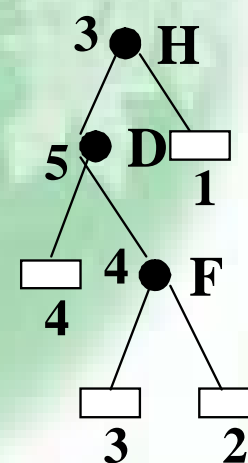
52
24



41
22

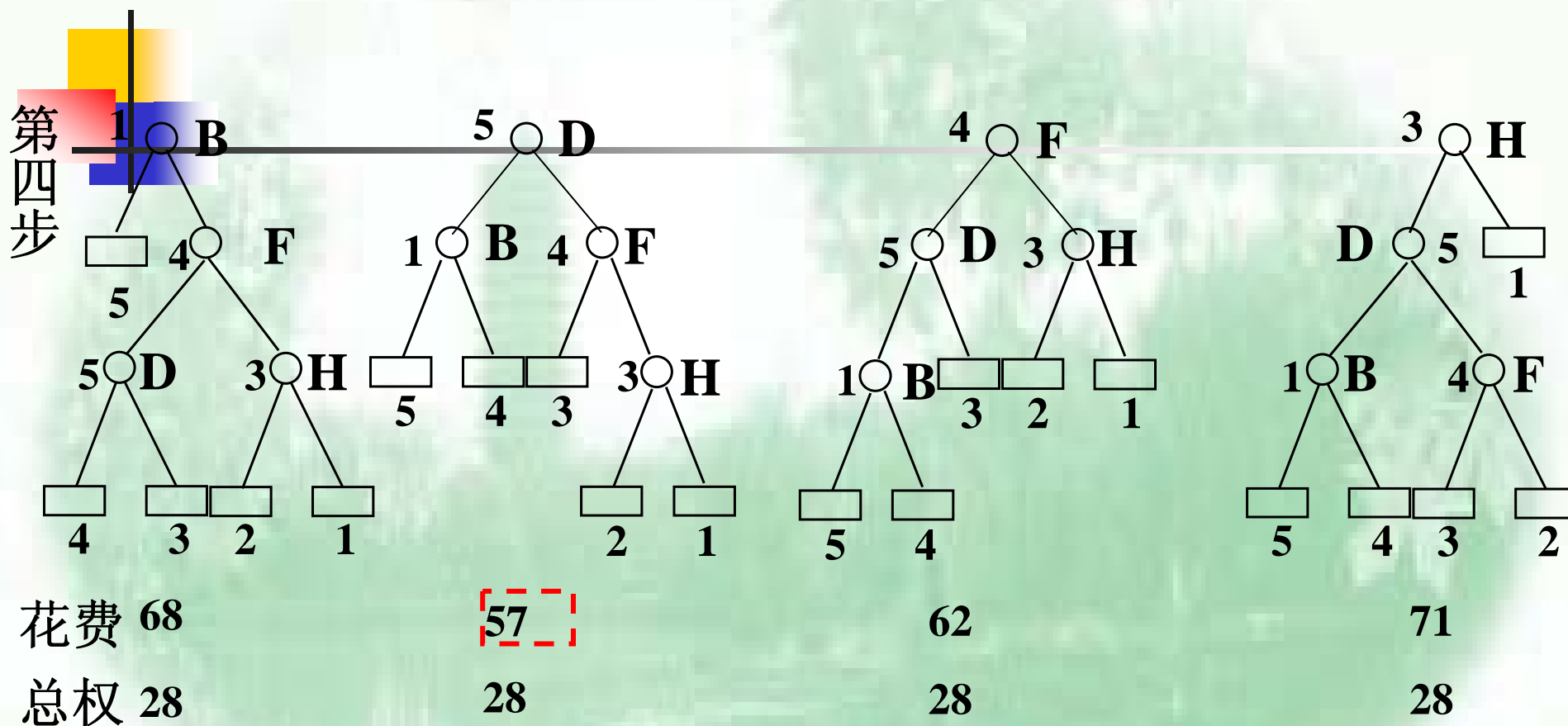


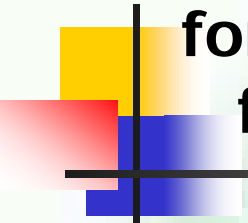
40
22



49
24

第四步

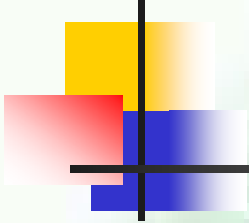




```

void OptimalBST(int a[], int b[], int n, int
    c[N+1][N+1], int r[N+1][N+1], int w[N+1][N+1])
{
    for(int i=0;i<=n;i++)
        for(int j=0;j<=n;j++) { // 初始化
            c[i][j]=0;
            r[i][j]=0;
            w[i][j]=0;
        }
    for (i = 0; i <=n; i++) {
        w[i][i] = b[i];
        for(int j=i+1;j<=n;j++) //求出权和w[i.j]
            w[i][j]=w[i][j-1]+a[j]+b[j];
    }
    for(int j=1;j<=n;j++) { //确定一个结点的BestBST
        c[j-1][j]=w[j-1][j];
        r[j-1][j]=j;
    }
}

```



```
int m,k0,k;  
for(int d=2;d<=n;d++) {  
    for(int j=d;j<=n;j++) {  
        i=j-d;  
        m=c[i+1][j];  
        k0=i+1;  
        for(k=i+2;k<=j;k++) {  
            if(c[i][k-1]+c[k][j]<m) {  
                m=c[i][k-1]+c[k][j];  
                k0=k;  
            }  
        }  
        c[i][j]=w[i][j]+m;    r[i][j]=k0;  
    }  
}  
}
```

12.2.2 平衡的二叉搜索树(AVL)

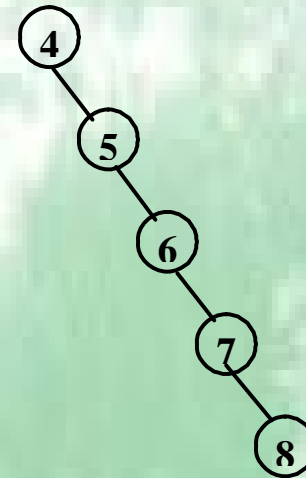
■ BST受输入顺序影响

- 最好 $O(\log n)$
- 最坏 $O(n)$

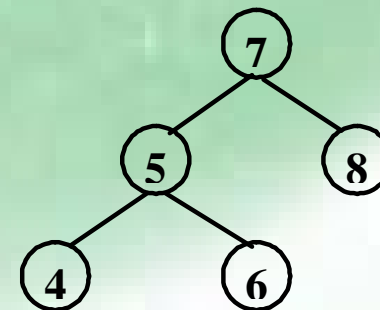
■ Adelson-Velskii 和 Landis发明了AVL树

- 平衡的二叉搜索树
- 始终保持 $O(\log n)$ 量级

输入顺序为 4、5、6、7、8



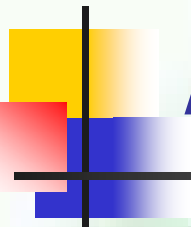
输入顺序为 7、5、4、6、8



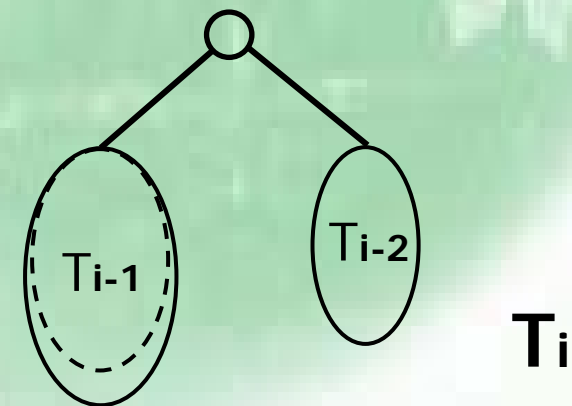
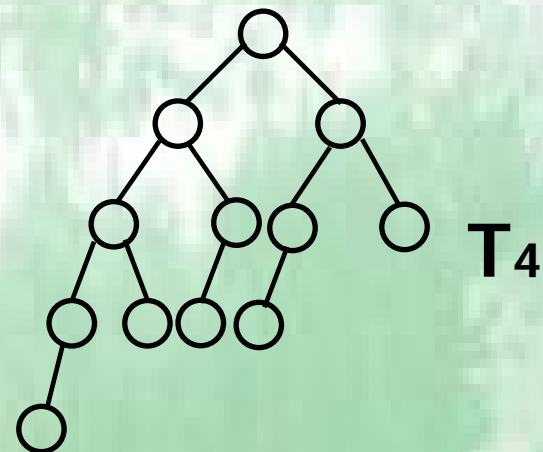
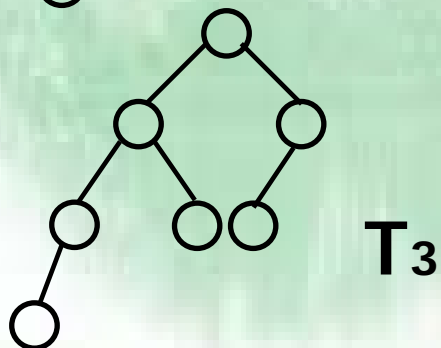
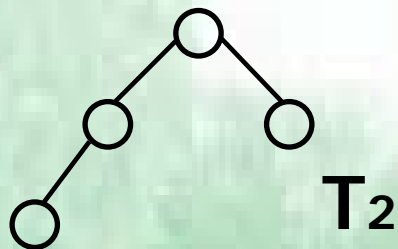
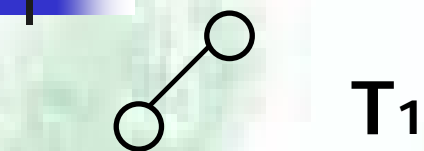


AVL树的性质

- 可以为空
- 具有 n 个结点的AVL树，高度为 $O(\log n)$
- 如果 T 是一棵AVL树
 - 那么它的左右子树 T_L 、 T_R 也是AVL树
 - 并且 $|h_L - h_R| \leq 1$
 - h_L 、 h_R 是它的左右子树的高度

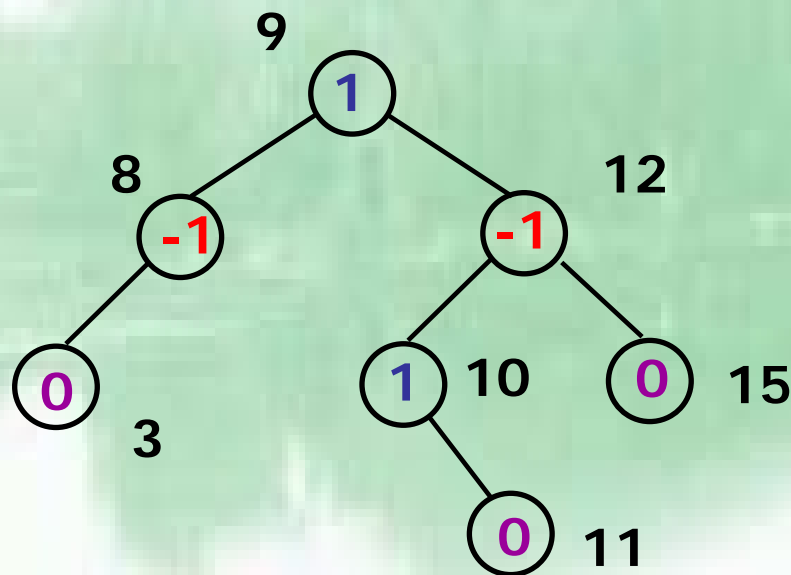


AVL树举例



平衡因子

- 平衡因子, $bf(x)$:
 - $bf(x) = x$ 的右子树的高度 - x 的左子树的高度
- 对于一个AVL树中的结点平衡因子之可能取值为0, 1和-1

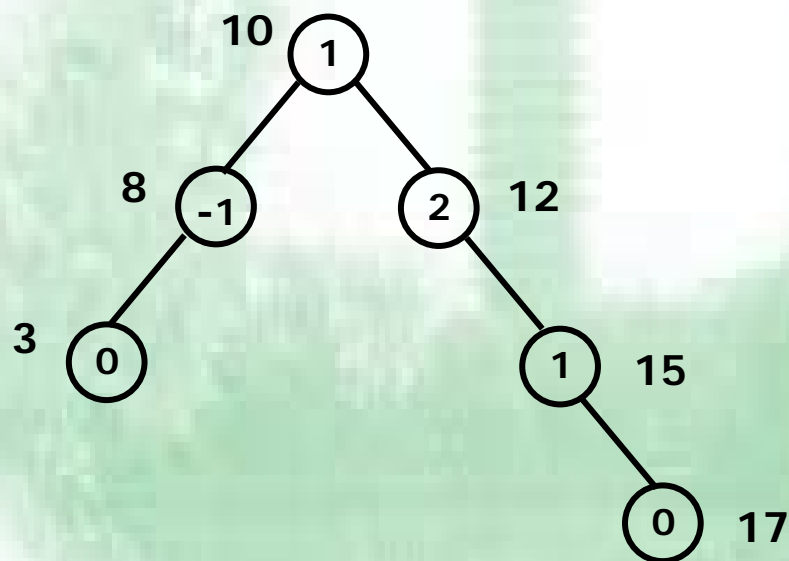




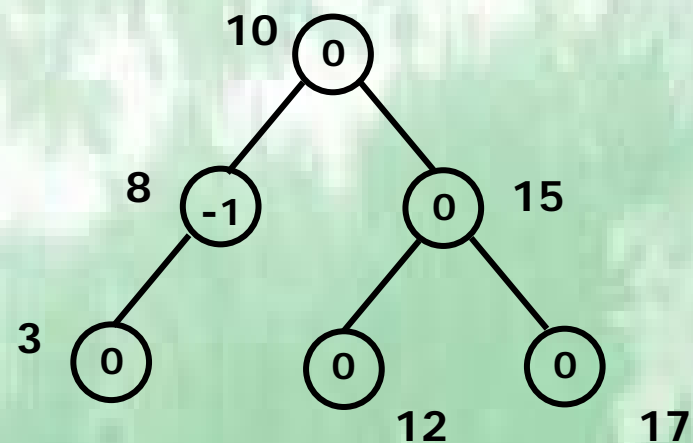
AVL树结点的插入

- 插入与**BST**一样
 - 新结点作叶结点
- 需要调整
- 相应子树的根结点变化
 - 结点原来是平衡的，现在成为左重或右重的
 - 修改相应前驱结点的平衡因子
 - 结点原来是某一边重的，而现在成为平衡的了
 - 树的高度未变，不修改
 - 结点原来就是左重或右重的，又加到重的一边
 - 不平衡
 - “危急结点”

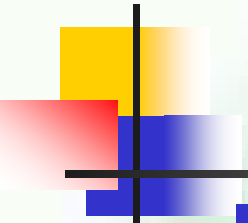
恢复平衡



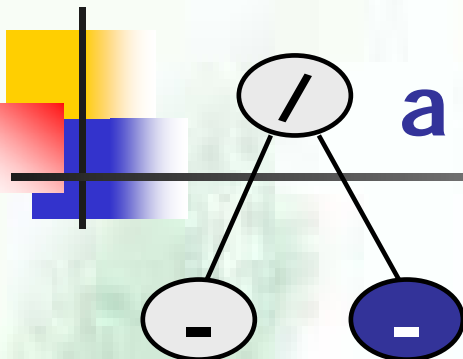
插入**17**后导致不平衡



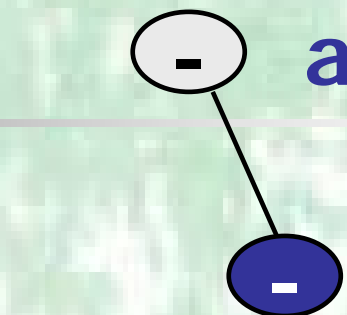
重新调整为平衡结构

- 
- 不平衡情况发生在插入新结点后
 - BST把新结点插入到叶结点
 - 假设a是离插入结点最近，且平衡因子绝对值不等于0的结点
 - 新插入的关键码为key的结点s要么在它的左子树中，要么在其右子树中
 - 假设插入在右边，原平衡因子
 - (1) $a \rightarrow bf = -1$
 - (2) $a \rightarrow bf = 0$
 - (3) $a \rightarrow bf = +1$

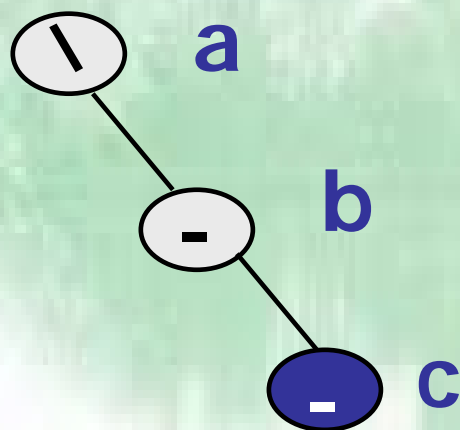
都插入在右边



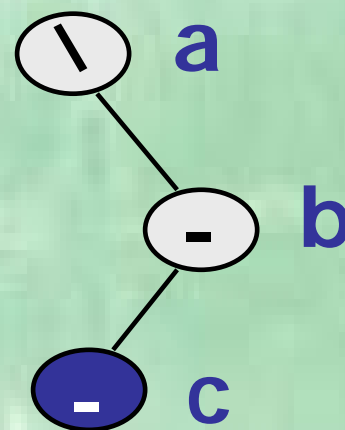
$$(1) a \rightarrow bf = -1 \Rightarrow 0$$



$$(2) a \rightarrow bf = 0 \Rightarrow +1$$

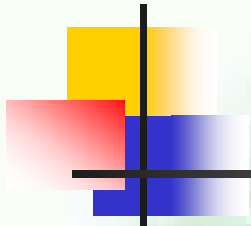


RR型



RL型

$$(3) a \rightarrow bf = +1 \Rightarrow +2$$



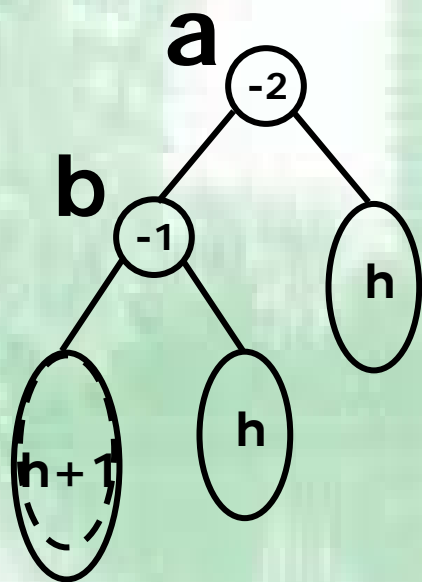
- 假设a离插入结点最近，且平衡因子绝对值不等于0
 - 新插入的结点s（关键码为key）要么在a的左子树中，要么在其右子树中
 - 假设在右边，因为从s（新插入结点）到a的除s和a以外的结点都要从原 $bf=0$ 变为 $|bf|=+1$ ，对于结点a
 - 1. $a \rightarrow bf = -1$ ，则 $a \rightarrow bf = 0$ ，以a为根的子树高度不变
 - 2. $a \rightarrow bf = 0$ ，则 $a \rightarrow bf = +1$ ，以a为根的子树树高改变
 - 由a的定义($a \rightarrow bf \neq 0$)，可知a是根
 - 3. $a \rightarrow bf = +1$ ，则 $a \rightarrow bf = +2$ ，**需要调整**



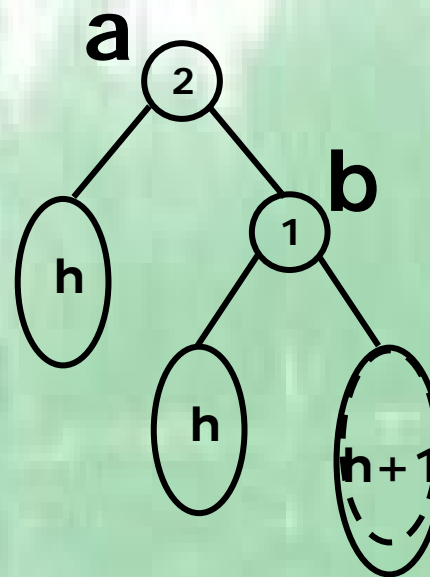
不平衡的情况

- AVL树任意结点A的平衡因子只能是**0, 1, -1**
- A本来左重, $A.bf == -1$, 插入一个结点导致A.bf变为-2
 - **LL型**: 插入到A的左子树的左子树
 - 左重+左重, A.bf变为-2
 - **LR型**: 插入到A的左子树的右子树
 - 左重+右重, A.bf变为-2
- 类似地, $A.bf == 1$, 插入新结点使得A.bf变为2
 - **RR型**: 导致不平衡的结点为A的右子树的右结点
 - **RL型**: 导致不平衡的结点为A的右子树的左结点

不平衡的图示



LL型



RR型



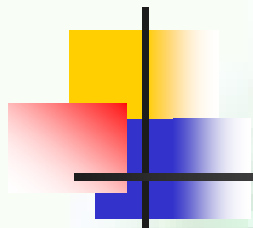
不平衡情况总结

- **LL型和RR型是对称的，LR型和RL型是对称的**
- **不平衡的结点一定在根结点与新加入结点之间的路径上**
- **它的平衡因子只能是2或者-2**
 - 如果是2，它在插入前的平衡因子是1
 - 如果是-2，它在插入前的平衡因子是-1



解决不平衡的方法——旋转

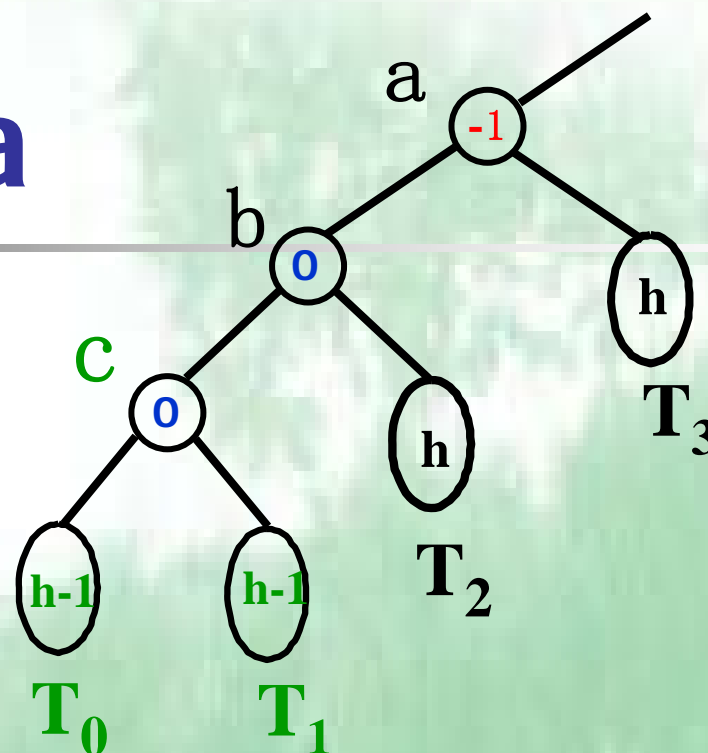
- 我们可以使用一系列称为旋转(**rotation**) 的局部操作解决这个问题
 - LL和RR的情况可以通过单旋转(**single rotation**)来解决
 - RL和LR的情况可以通过双旋转(**double rotation**)来解决
- 调整的整个过程称之为重构(**restructuring**)



单旋转

- 对根为结点a的子树进行单旋转
 - b为包含新加入结点的a的子结点
 - c为包含新加入结点的b的子结点
- 单旋转
 - 令b代替a成为新根，a和c作为其子结点
 - 原来c的子树保持不变
 - 原来b中c结点的兄弟子树，作为a的子树

危机结点a

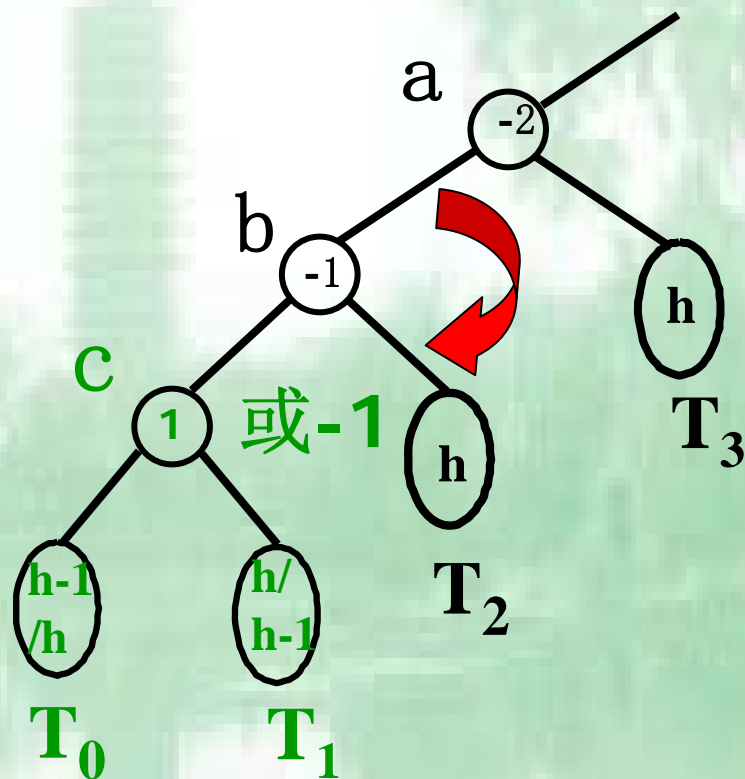


新结点插入到c的左或右子树

插入前，a是最靠近c的祖先路径上， $a.bf \neq 0$ 的结点。 a 子树高 $h+2$

访问顺序为a、b、c（编号，不是值大小）

LL型单旋转



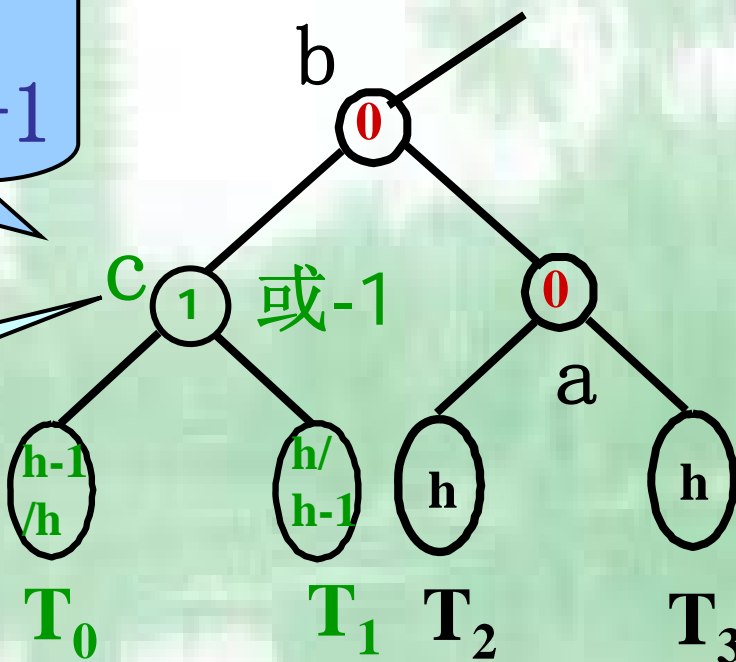
插入前
a子树高 $h+2$

插入后
a子树高 $h+3$

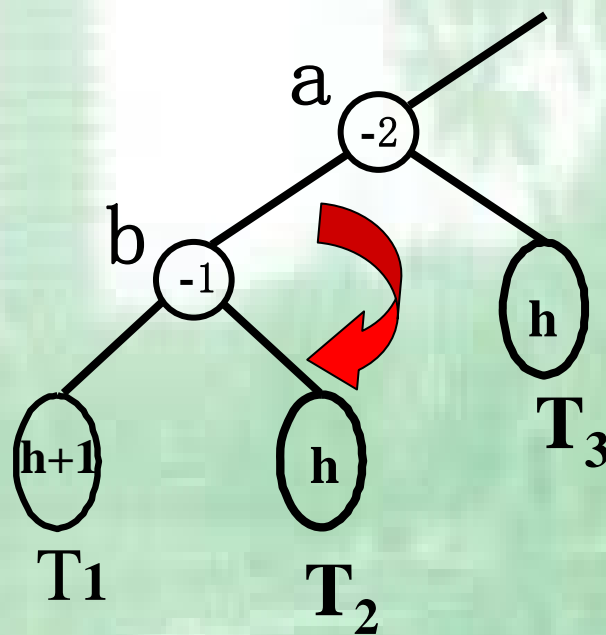
LL型单旋转

c. bf保持
原来的 1 或 -1

其实，不用
看结点C

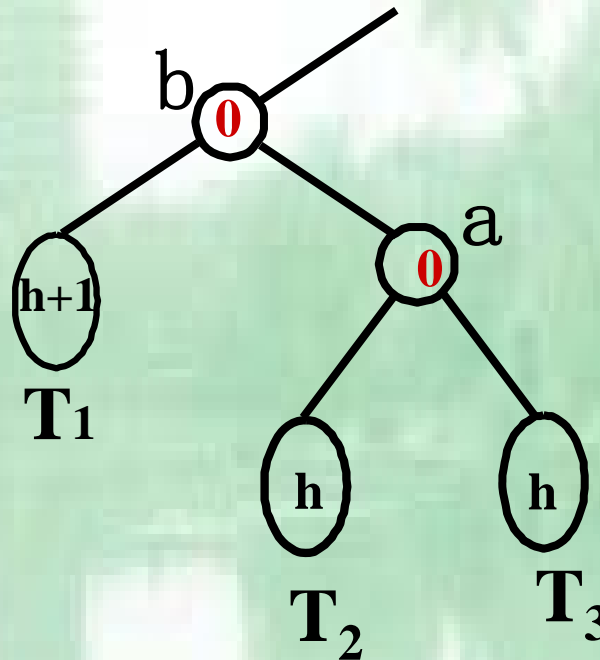


其实，不用看结点C

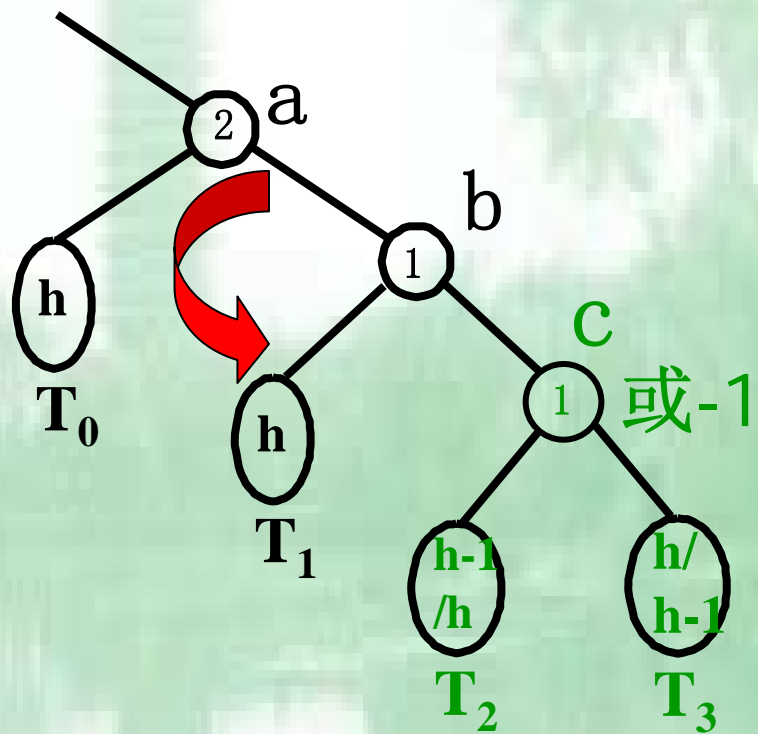


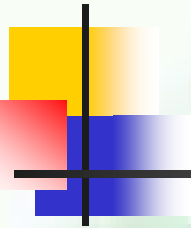


其实，不用看结点C

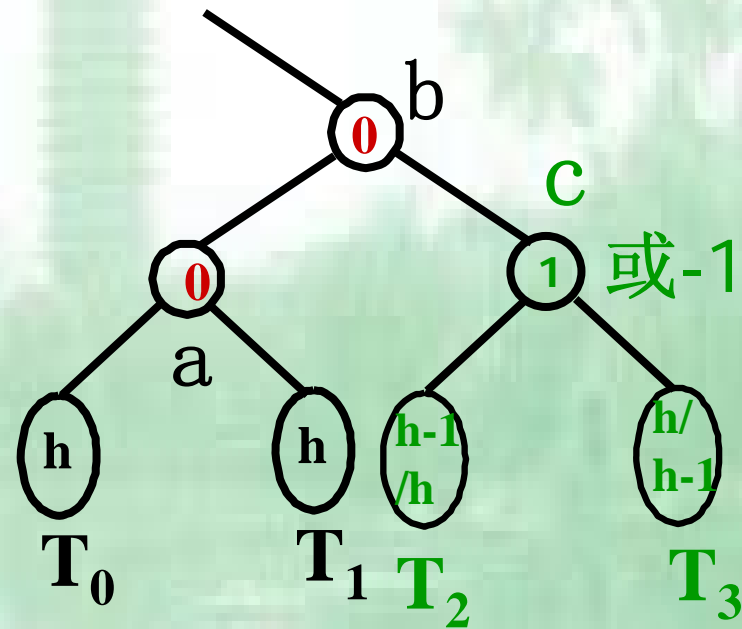


RR型单旋转





RR型单旋转





旋转运算的实质

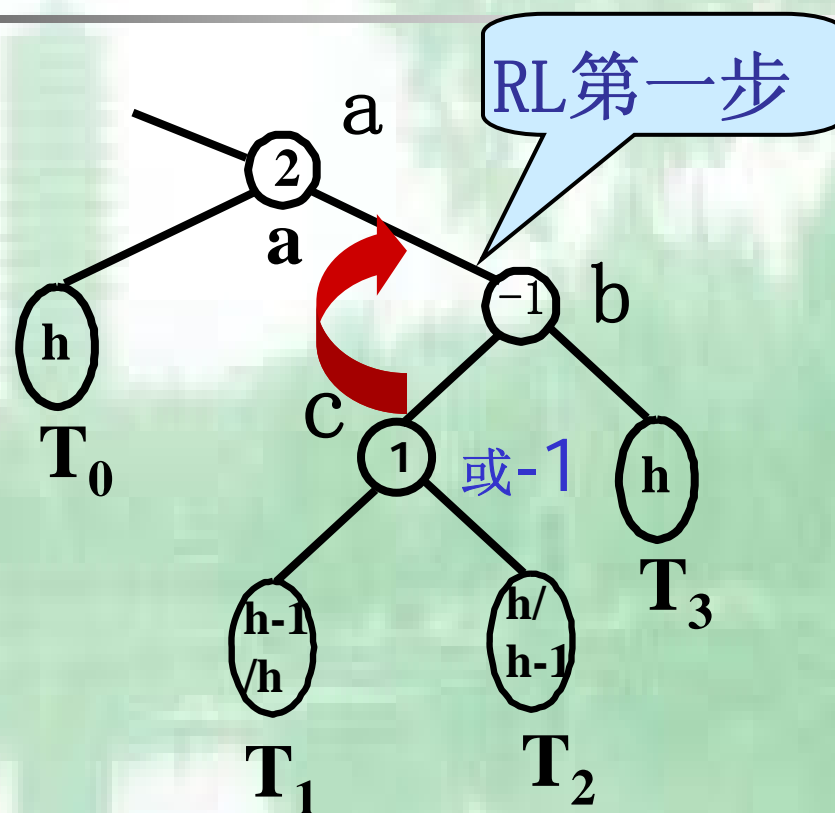
- 以RR型图示为例，总共有7个部分
 - 三个结点：**a**、**b**、**c**
 - 四棵子树**T₀**、**T₁**、**T₂**、**T₃**
 - 加重**c**为根的子树，但是其结构其实没有变化
 - **T₂**、**c**、**T₃**可以整体地看作**b**的右子树
- 目的：重新组成一个新的**AVL**结构
 - 平衡
 - 保留了中序周游的性质
 - **T₀ a T₁ b T₂ c T₃**



双旋转

- **RL**或者**LR**需要进行双旋转
 - 这两种情况是对称的
- 我们只讨论 **RL**的情况
 - **LR**是一样的

RL型双旋转第一步

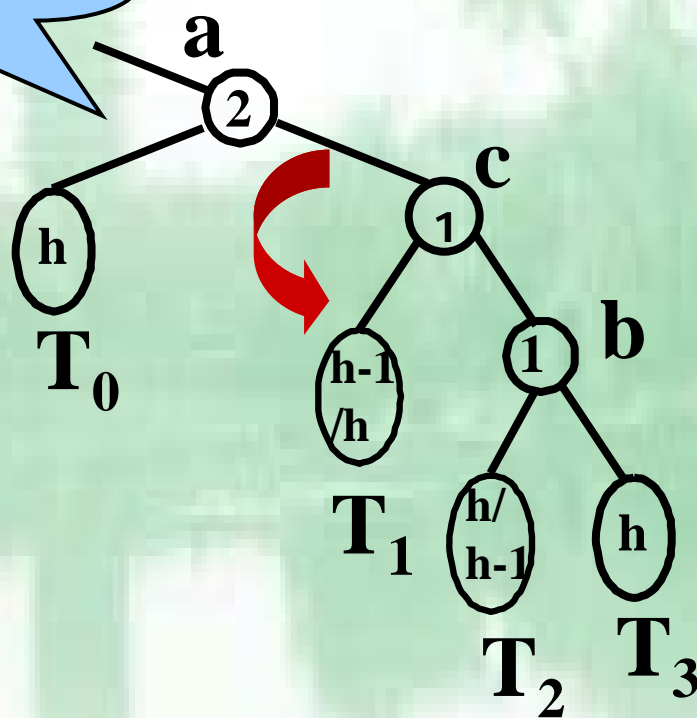


插入前
a子树高 $h+2$

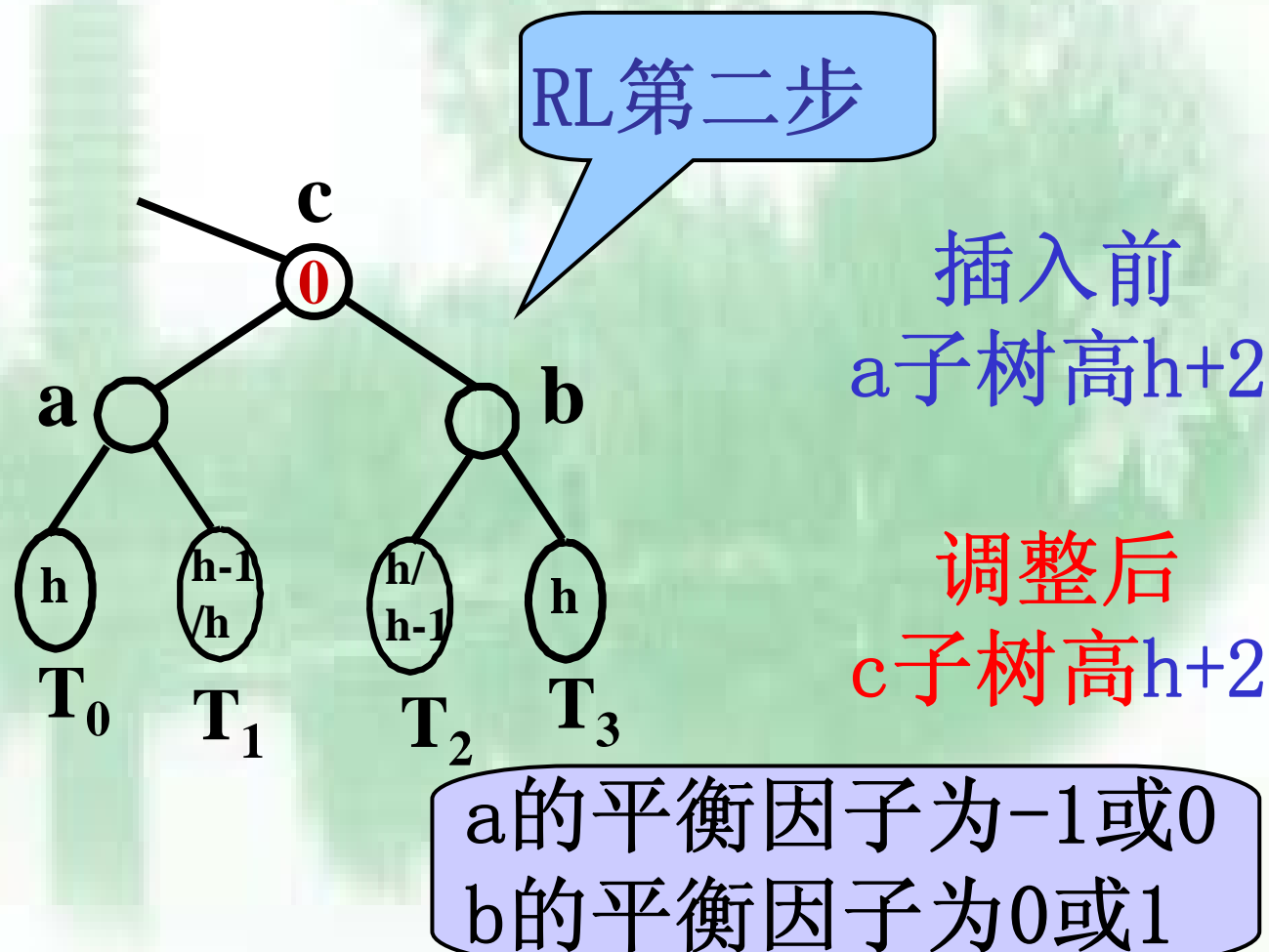
插入后
a子树高 $h+3$

RL型双旋转第一步

中间状态
平衡因子无意义



RL型双旋转第二步





旋转运算的实质（续）

把树做任何一种旋转（RR、RL、LL、LR）

- 新树保持了原来的中序周游顺序
- 旋转处理中仅需改变少数指针
- 而且新的子树高度为 $h+2$ ，保持插入前子树的高度不变
- 原来二叉树在a结点上面的其余部分（若还有的话）总是保持平衡的



AVL树结点的删除

- 删除是插入的逆操作
 - 删除操作与**BST**一样
 - **AVL**树结点删除后的调整比较复杂

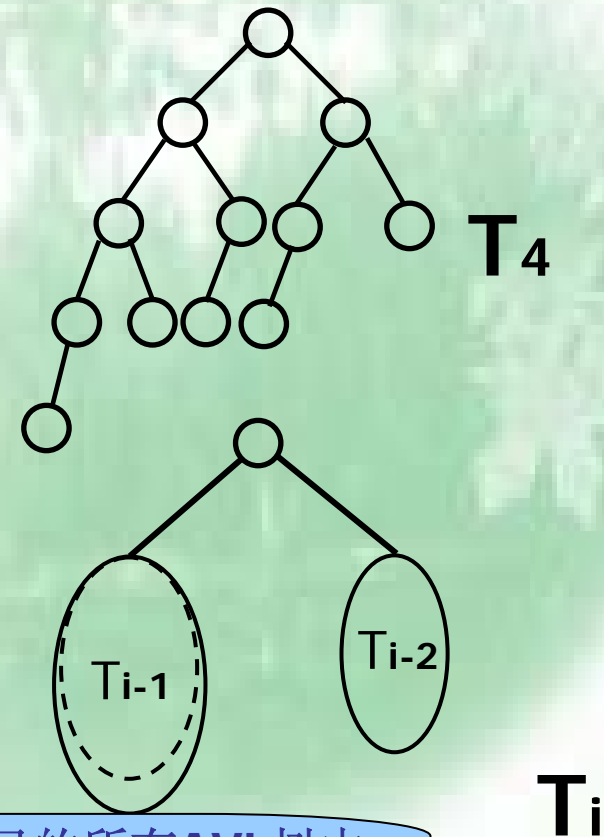
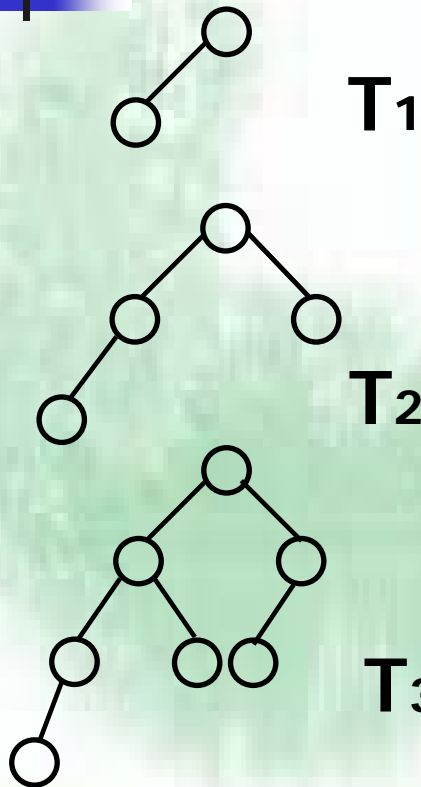


AVL树的高度

- 具有 n 个结点的AVL树的高度一定是 $O(\log n)$
 - n 个结点的AVL树的最大高度不超过 $K\log_2 n$
 - 这里 K 是一个小的常数
- 最接近于不平衡的AVL树
 - 构造一系列AVL树 T_1, T_2, T_3, \dots

T_i 的高度是 i

每棵具有高度 i 的其它AVL树都比 T_i 的结点个数多



或者说, T_i 是具有同样的结点数目的所有AVL树中最接近不平衡状态的, 删除一个结点都会不平衡



高度的证明（推理）

- 可看出有下列关系成立：
 - $t(1) = 2$
 - $t(2) = 4$
 - $t(i) = t(i-1) + t(i-2) + 1$
- 对于 $i > 2$ 此关系很类似于定义Fibonacci数的那些关系：
 - $F(0) = 0$
 - $F(1) = 1$
 - $F(i) = F(i-1) + F(i-2)$



高度的证明（推理续）

- 对于 $i > 1$ 仅检查序列的前几项就可有
 - $t(i) = F(i+3) - 1$
- **Fibonacci**数满足渐近公式

$$F(i) = \frac{1}{\sqrt{5}} \phi^i, \text{ 这里 } \phi = \frac{1 + \sqrt{5}}{2}$$

- 由此可得近似公式

$$t(i) \approx \frac{1}{\sqrt{5}} \phi^{i+3} - 1$$



高度的证明（结果）

- 解出高度*i*与结点个数*t(i)*的关系

$$\phi^{i+3} \approx \sqrt{5}(t(i) + 1)$$

$$i + 3 \approx \log_{\phi} \sqrt{5} + \log_{\phi} (t(i) + 1)$$

- 由换底公式 $\log_{\phi} X = \log_2 X / \log_2 \phi$ 和 $\log_2 \phi \approx 0.694$ 我们求出近似上限

$$i < \frac{3}{2} \log_2 (t(i) + 1) - 1$$

- $t(i) = n$

- 所以*n*个结点的AVL树的高度一定是 $O(\log n)$



AVL树的效率



- 检索、插入和删除效率都是 $O(\log_2 n)$
 - 具有 n 个结点的AVL树的高度一定是 $O(\log n)$
- AVL树适用于组织较小的、内存中的目录
- 存放在外存储器上的较大的文件
 - B树/B+树，尤其是B+树



12.2.3 伸展树

- 一种自组织数据结构
 - 数据随检索而调整位置
 - 汉字输入法的词表
- 伸展树不是一个新数据结构，而只是改进**BST**性能的一组规则
 - 保证访问的总代价不高，达到最令人满意的性能
 - 不能保证最终树高平衡



补充：自组织线性表

- 1) 计数方法
 - 为每一个记录保存一个访问计数，而且一直按照这个顺序维护计数。
 - 组织成按频率排序的线性表。
 - 缺点：保存访问计数需要空间，时间推移访问频率改变
- 2) 向前移动方法
 - 当找到一个记录时就把它放到线性表的前面，而把其他记录后退一个位置。
 - 适宜于链表实现，对访问频率的局部变化能够很好地响应



自组织线性表（续）

■ 3) 转置

- 把找到的记录与它在线性表中的前一个记录交换位置
- 适宜于顺序表和链表
- 随着时间的推移，最常使用的记录将移动到线性表的前面
 - 曾经被频繁访问但是以后不再使用的记录将会慢慢地落在后面

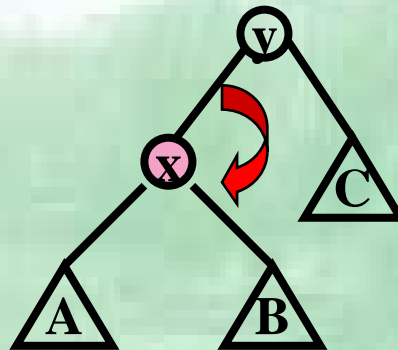


展开(**splaying**)

- 访问一次结点（例如结点 x ），完成一次称为展开的过程
 - x 被插入、检索时，把结点 x 移到BST的根结点
 - 删除结点 x 时，把结点 x 的父结点移到根结点
- 像在AVL树中一样，结点 x 的一次展开包括一组旋转(**rotation**)
 - 调整结点 x 、父结点、祖父结点的位置
 - 把 x 移到树结构中的更高层

单旋转(single rotation)

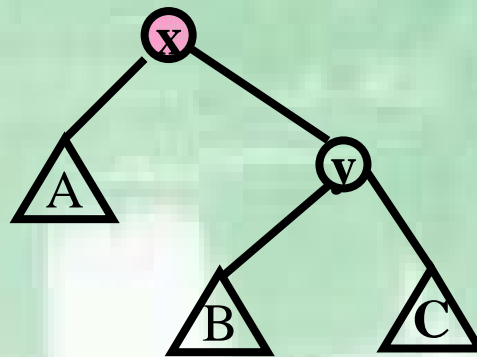
- x是根结点的直接子结点时
 - 把结点x与它的父结点交换位置
 - 保持BST特性



a、b为内部结点编号，不是值大小
A、B、C代表子树，有大小顺序

单旋转(single rotation)

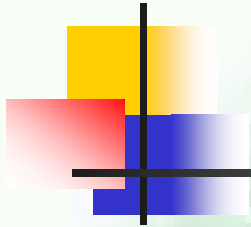
- x是根结点的直接子结点时
 - 把结点x与它的父结点交换位置
 - 保持BST特性





双旋转(double rotation)

- 双旋转涉及到
 - 结点 x
 - 结点 x 的父结点(称为 y)
 - 结点 x 的祖父结点(称为 z)
- 把结点 x 在树结构中向上移两层



- 双旋转分为两类
 - 一字形旋转(**zigzig rotation**)
 - 也称为同构调整 (**homogeneous configuration**) ;
 - 之字形旋转(**zigzag rotation**)
 - 也称为异构调整 (**heterogeneous configuration**)

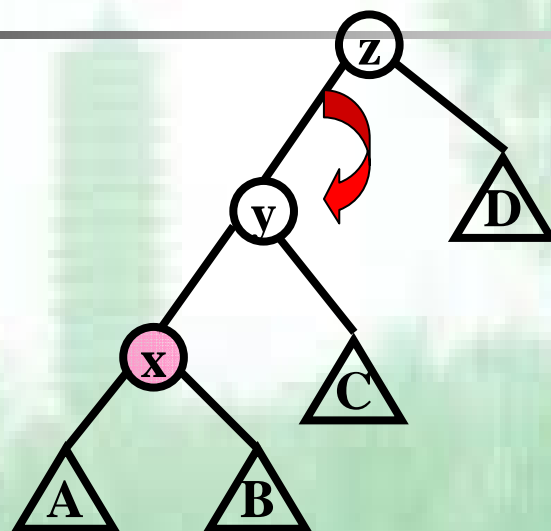


一字形旋转

- **x、y、z是一顺的**

- 1. 结点x是结点y的左，y是结点z的左子结点
- 2. 结点x是结点y的右，y是结点z的右子结点

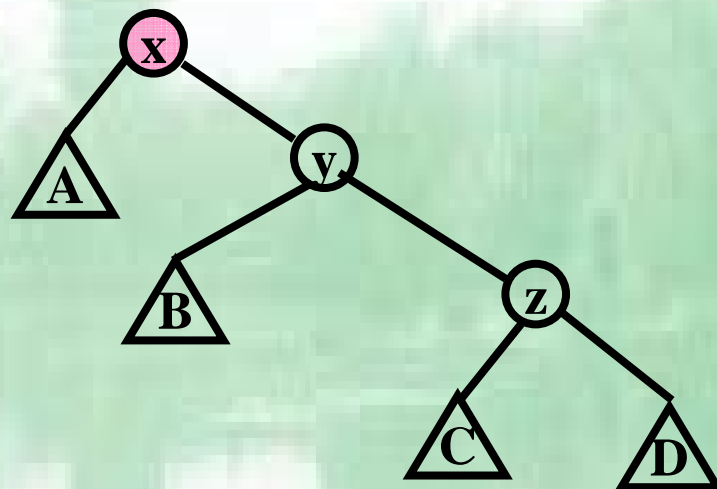
一字形旋转图示



结点x是y的左子结点
结点y是z的左子结点



一字形旋转图示



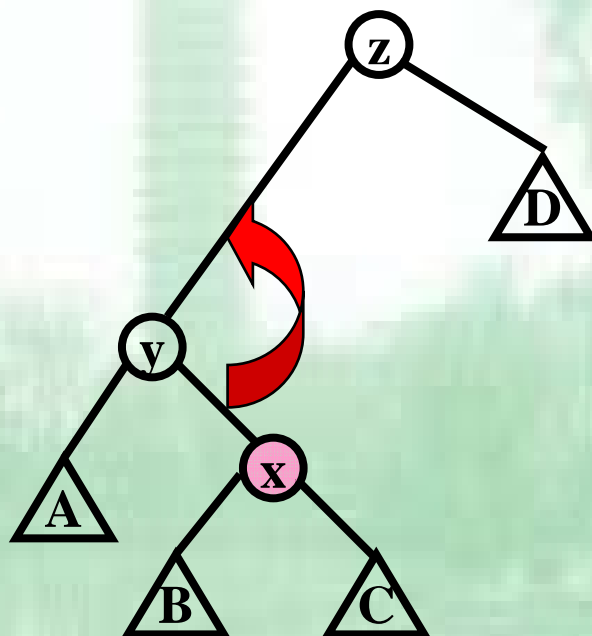


之字形旋转

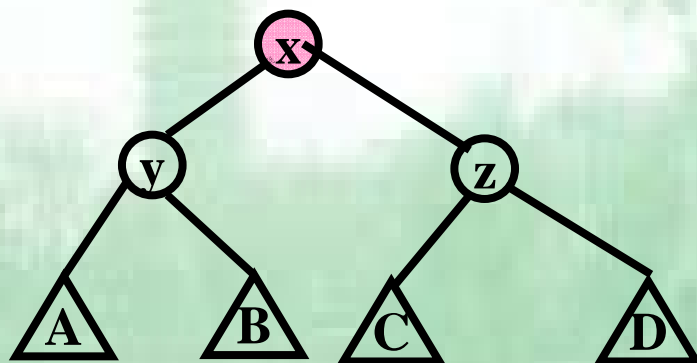
- **x、y、z**形成了“之字”扭结
 - 1. 结点**x**是结点**y**的左，**y**是结点**z**的右子结点
 - 2. 结点**x**是结点**y**的右，**y**是结点**z**的左子结点

之字形旋转图示

结点x是y的右子结点
结点y是z的左子结点



之字形旋转图示





两种旋转的不同作用

■ 之字形旋转

- 把新访问的记录向根结点移动
- 使子树结构的高度减1
- 趋向于使树结构更加平衡

■ 一字形提升

- 一般不会降低树结构的高度
- 只是把新访问的记录向根结点移动

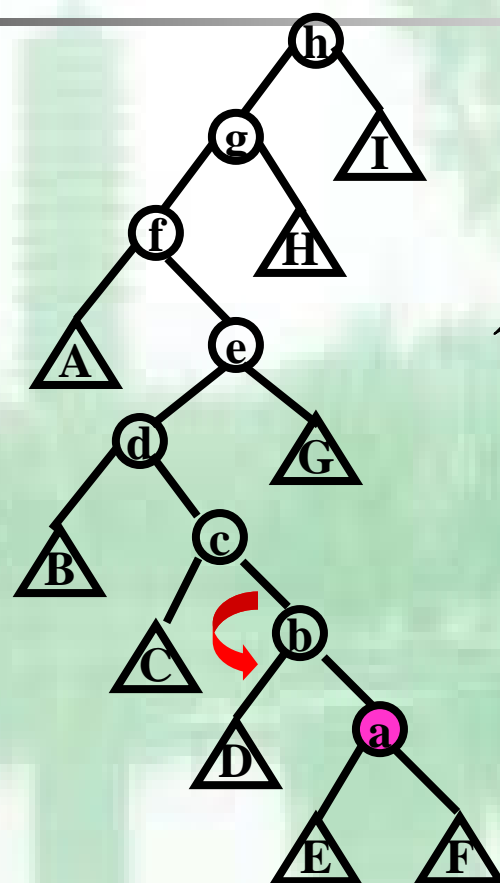


伸展树的调整过程



- 一系列双旋转
 - 直到结点 x 到达根结点或者根结点的子结点
- 如果结点 x 到达根结点的子结点
 - 进行一次单旋转使结点 x 成为根结点
- 这个过程趋向于使树结构重新平衡
 - 使访问最频繁的结点靠近树结构的根层
 - 从而减少访问代价

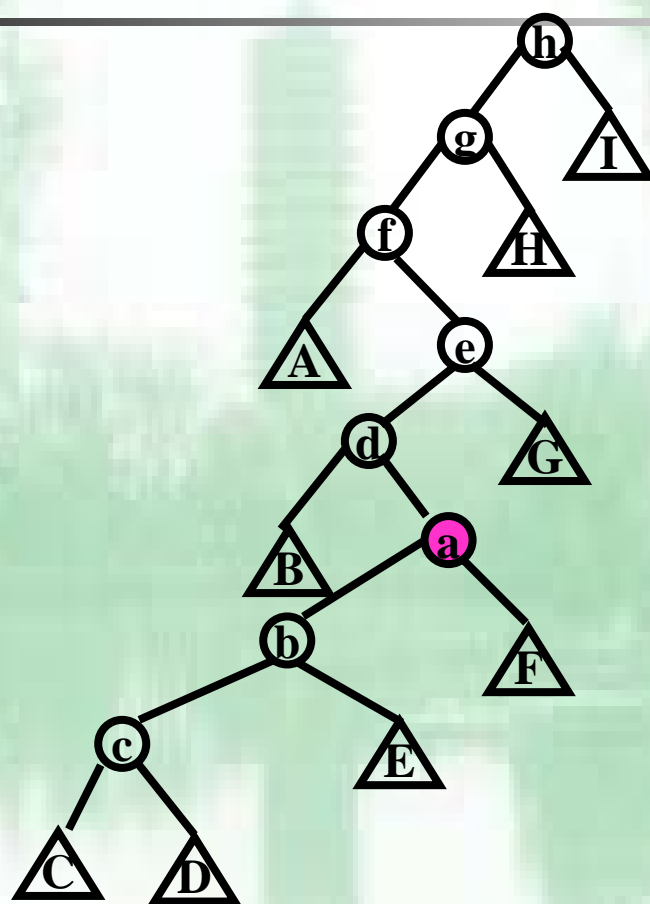
伸展树的调整过程



一字形旋转

(b, c) 左转
(a, b) 左转

伸展树的调整过程

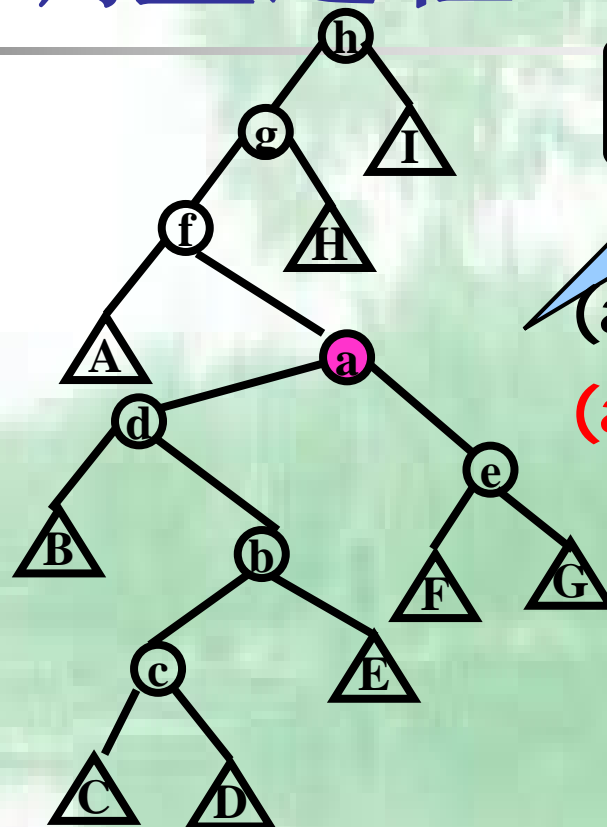


之字形旋转

(a, f) 左转

(a, g) 右转

伸展树的调整过程

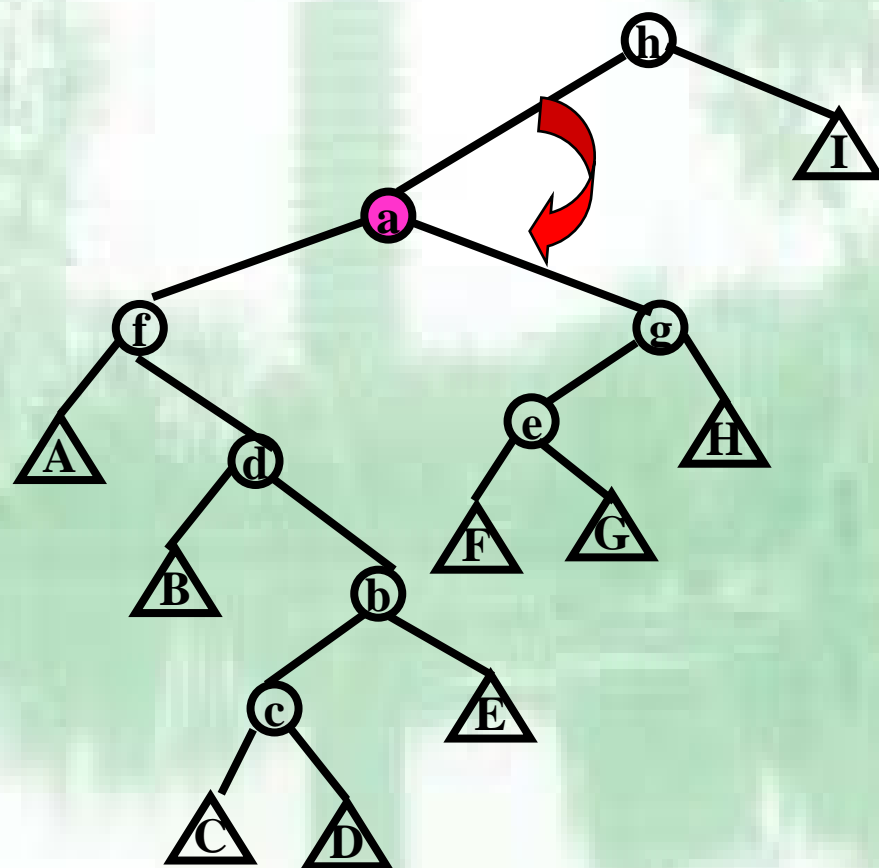


之字形旋转

(a, f) 左转

(a, g) 右转

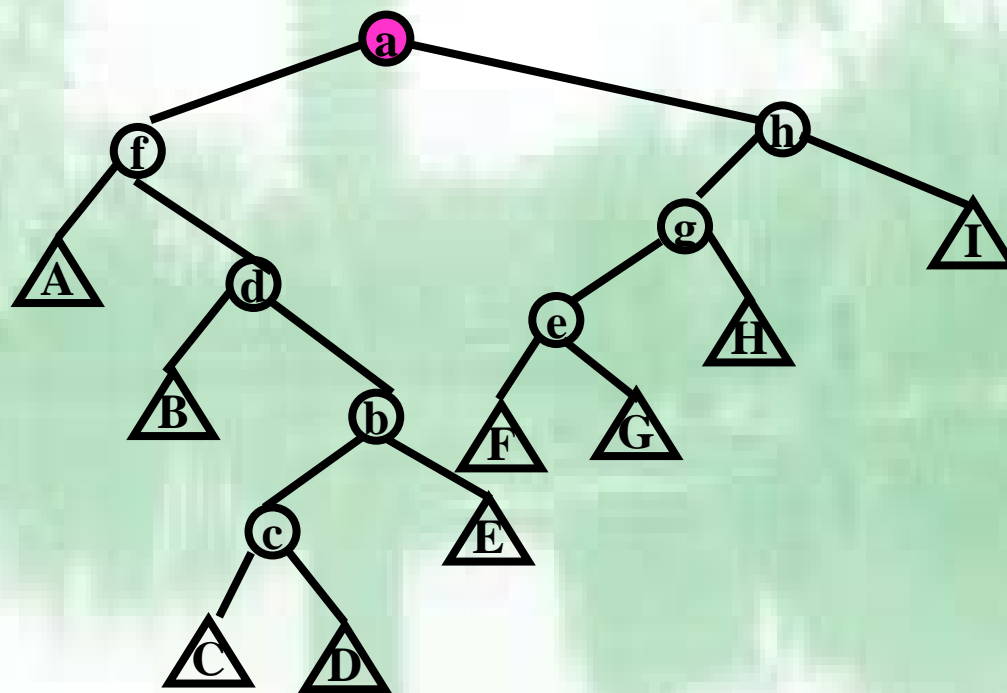
伸展树的调整过程



单旋转

(a, h) 右转

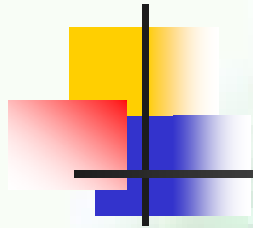
伸展树的调整过程





伸展树与AVL树的差别

- 伸展树与结点被访问的频率相关
 - 根据插入、删除、检索
 - 动态地调整
- 而AVL树的结构与访问频率无关
 - 只与插入、删除的**顺序**有关



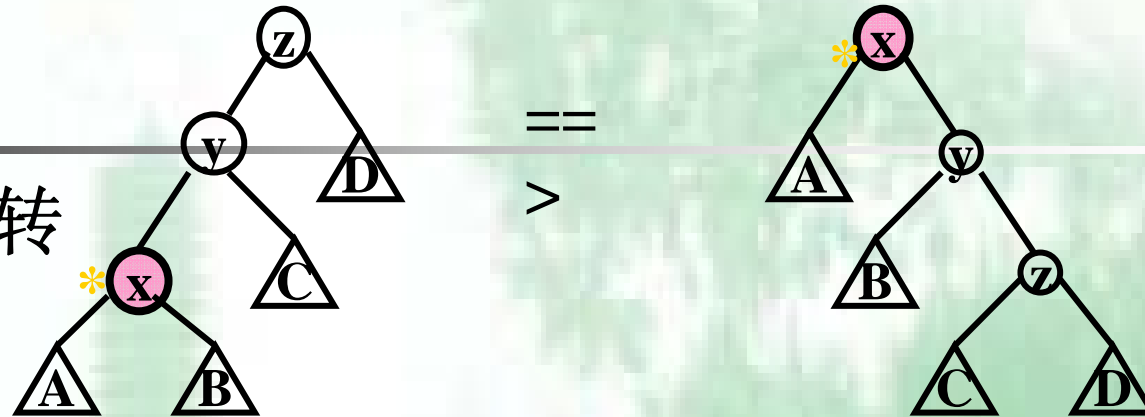
伸展树的效率

- n 个结点的伸展树
- 进行一组 m 次操作 (插入、删除、查找操作), 当 $m \geq n$ 时, 总代价是 $O(m \log n)$
 - 伸展树不能保证每一个单个操作是有效率的
 - 即每次访问操作的平均代价为 $O(\log n)$
- 不要求掌握证明方法

上机题12.5半伸展

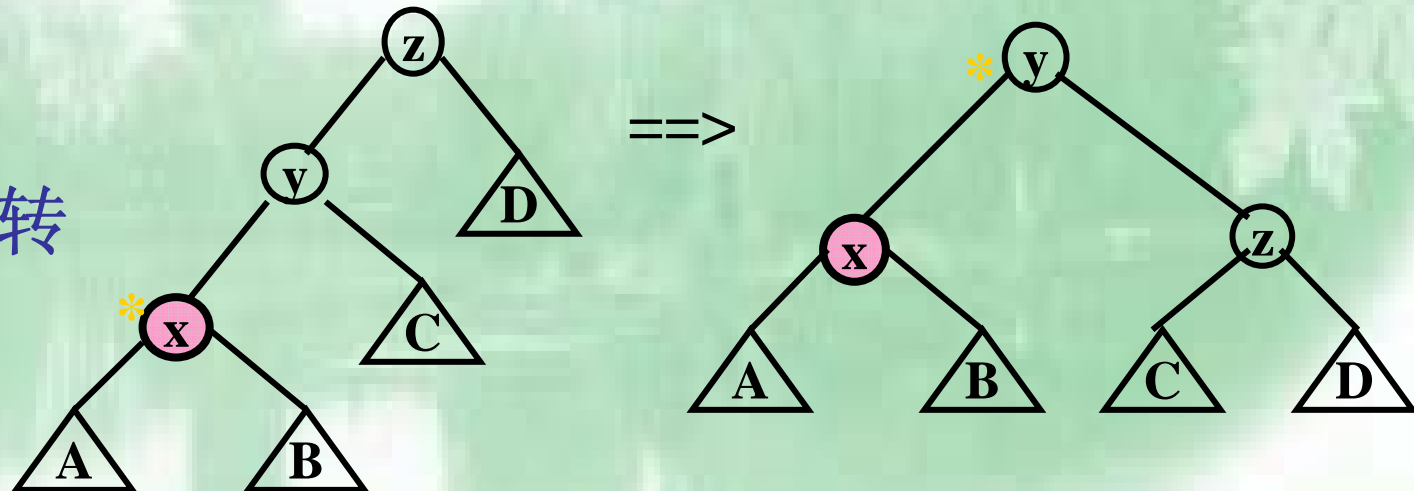
普通

一字旋转



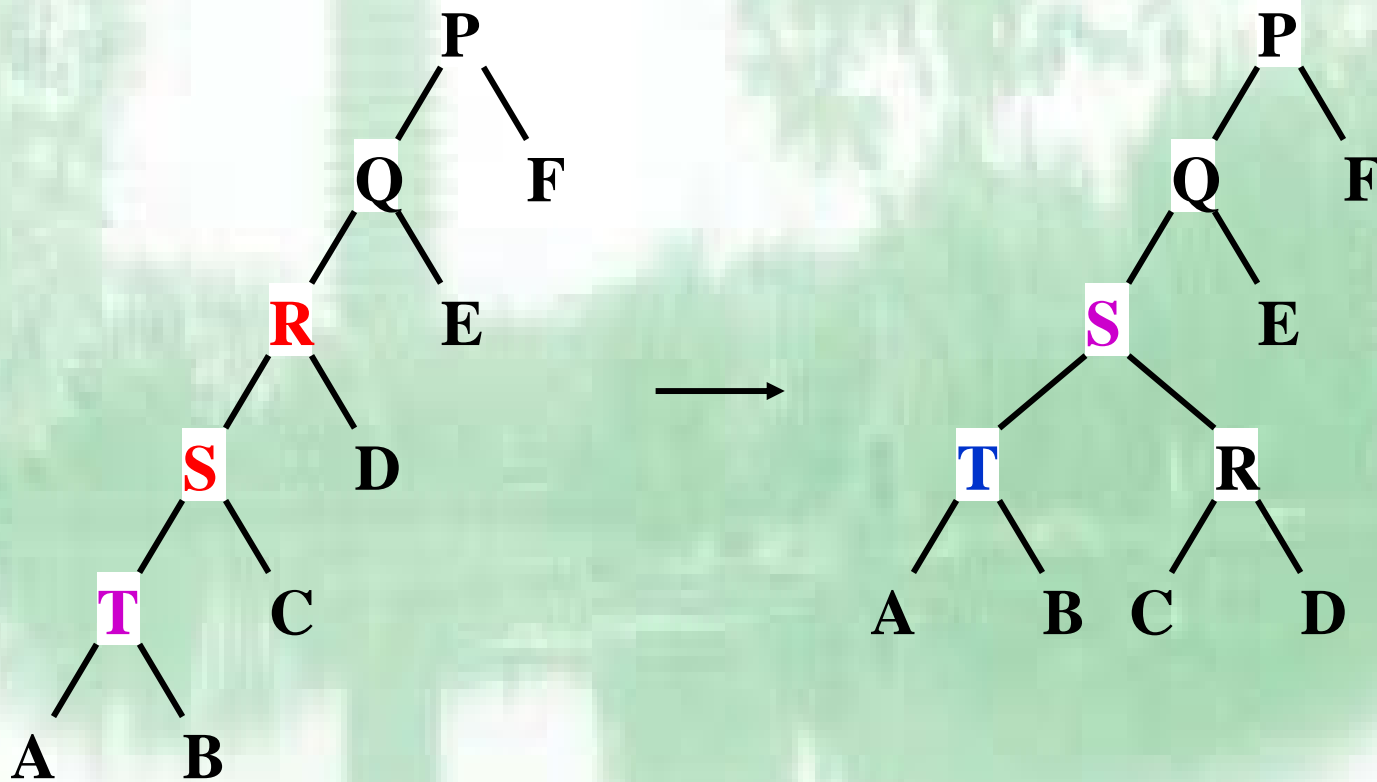
半伸展

一字旋转

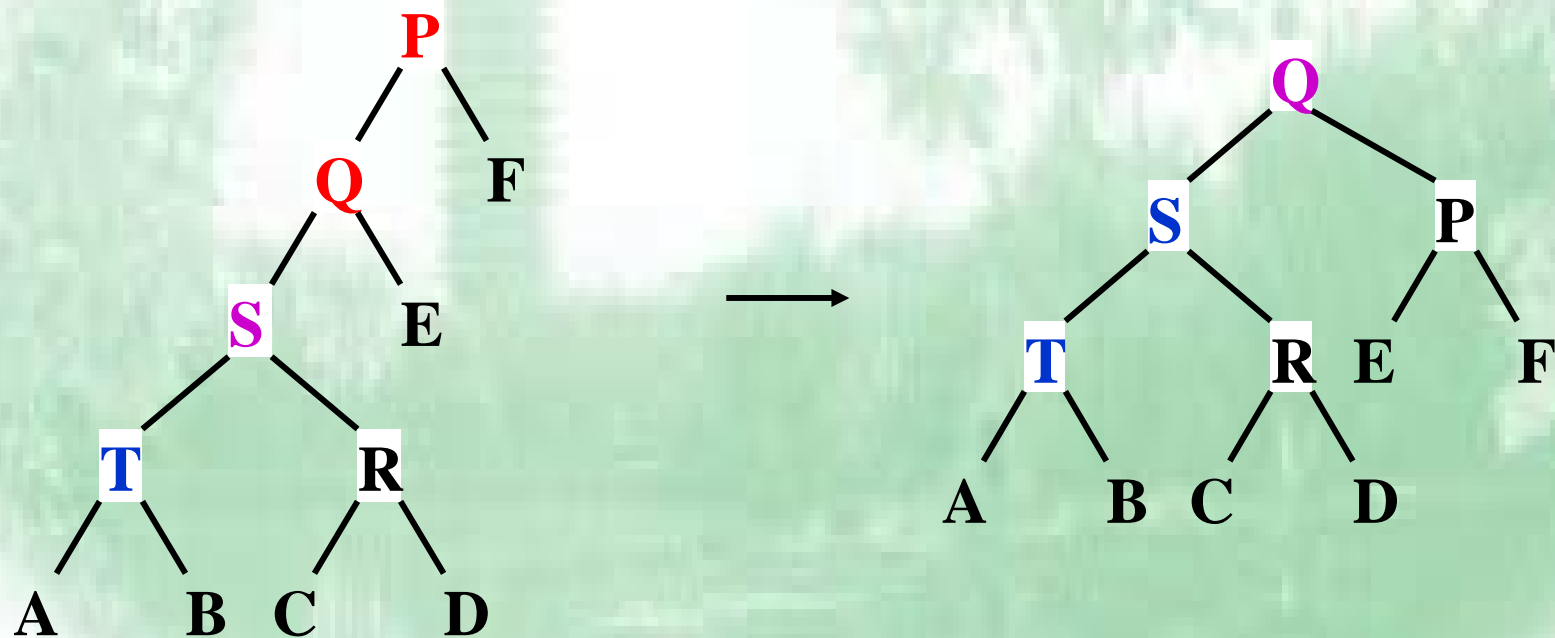


下一次旋转从y开始，而不从x开始

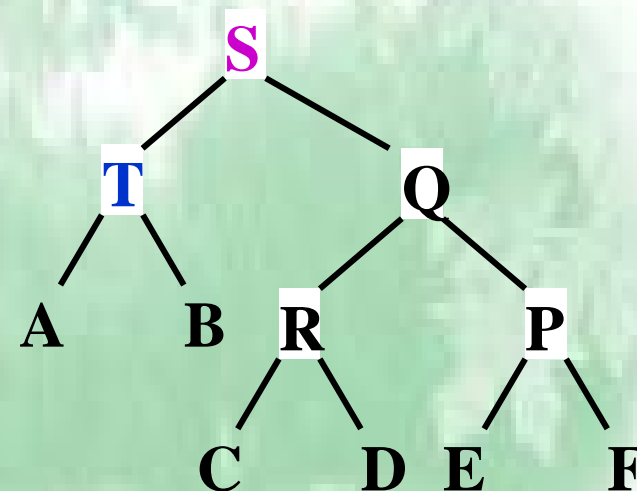
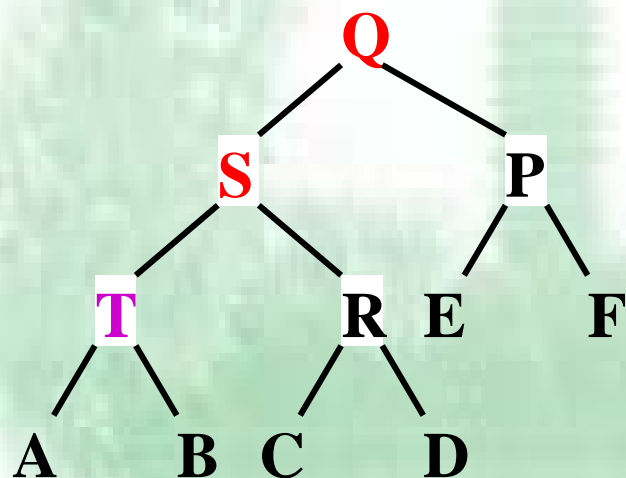
检索T



检索T(续)



第2次检索T





12.3 空间数据结构

- **BST、AVL**等都是针对一维关键码
 - 多维数据不能简单看作多个一维组合
- 常用的多维数据结构
 - **K-D树**
 - **PR四分树**
 - **R树/R*树**

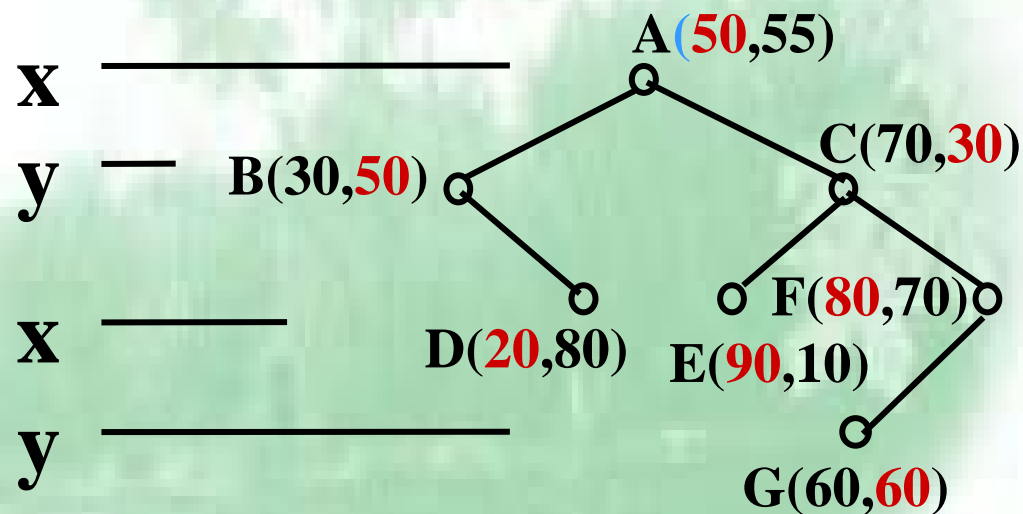
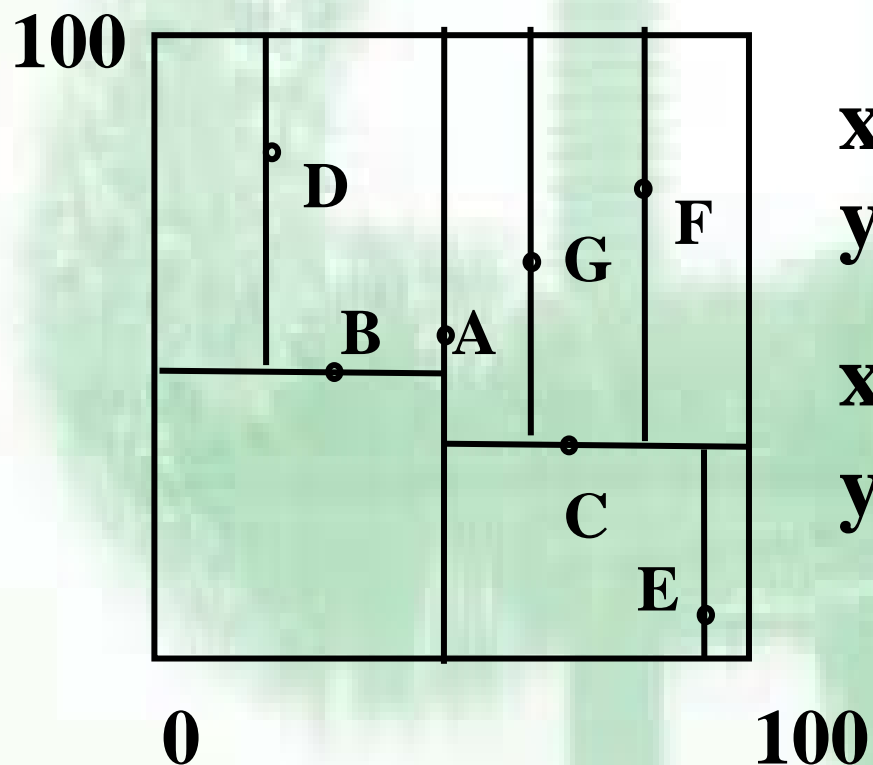


12.3.1 k-d树

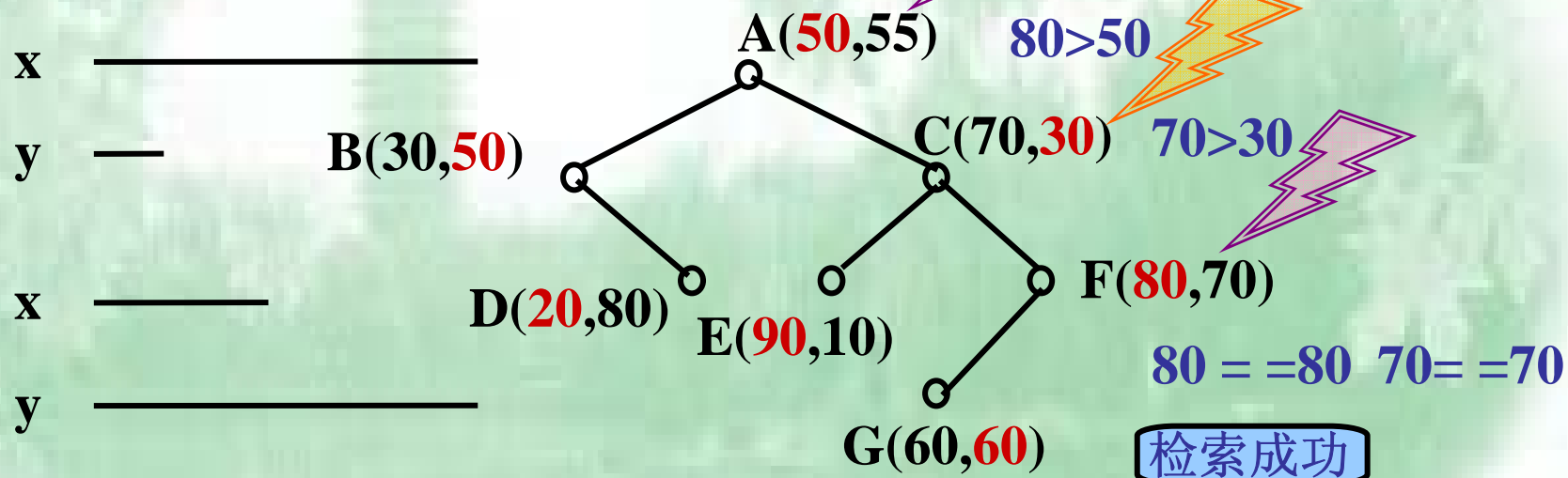
- k-d树每一层根据特定的关键码将对象空间分解为两个
 - 顶层结点按一个维划分
 - 第二层结点按照另一维进行划分
 - ...以此类推在各个维之间反复进行划分
 - 最终当一个结点中的点数少于给点的最大点数时，划分结束
- 交叉的“二维BST”

K-D树示例

A(50,55) B(30,55) C(70,30) D(20,80)
E(90,10) F(80,70) G(60,60)

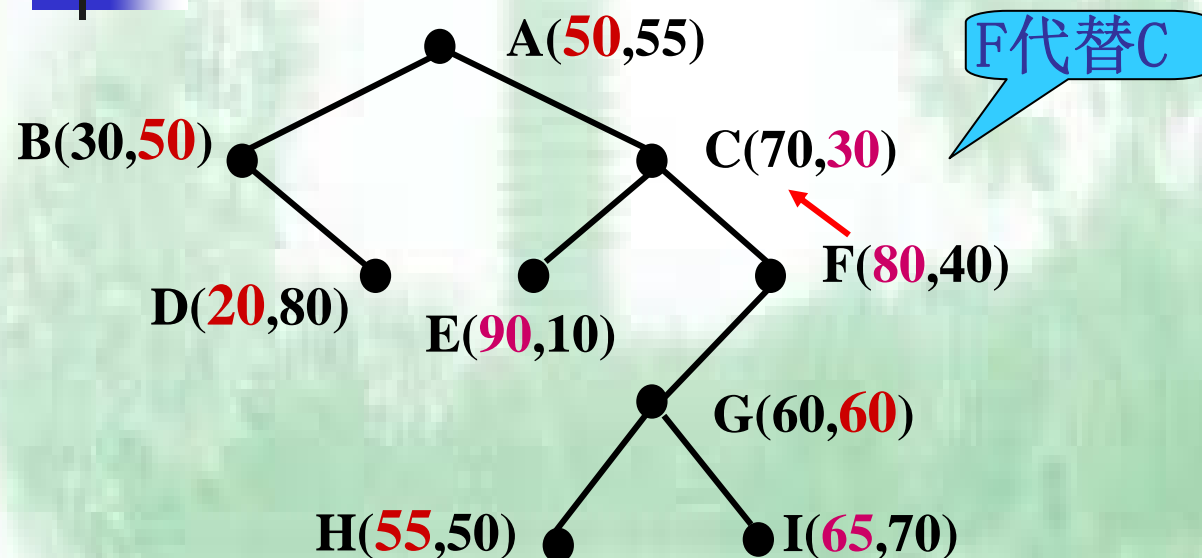


K-D树检索



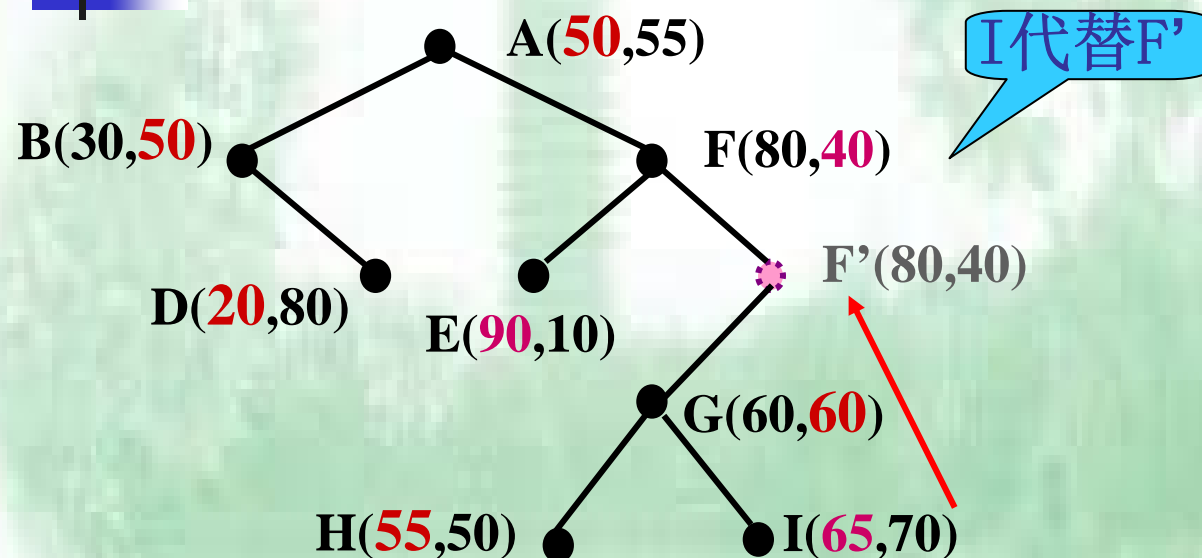
如果最后到达的结点指针是NULL，则检索结束，该结点不存在

K-D树的删除图示



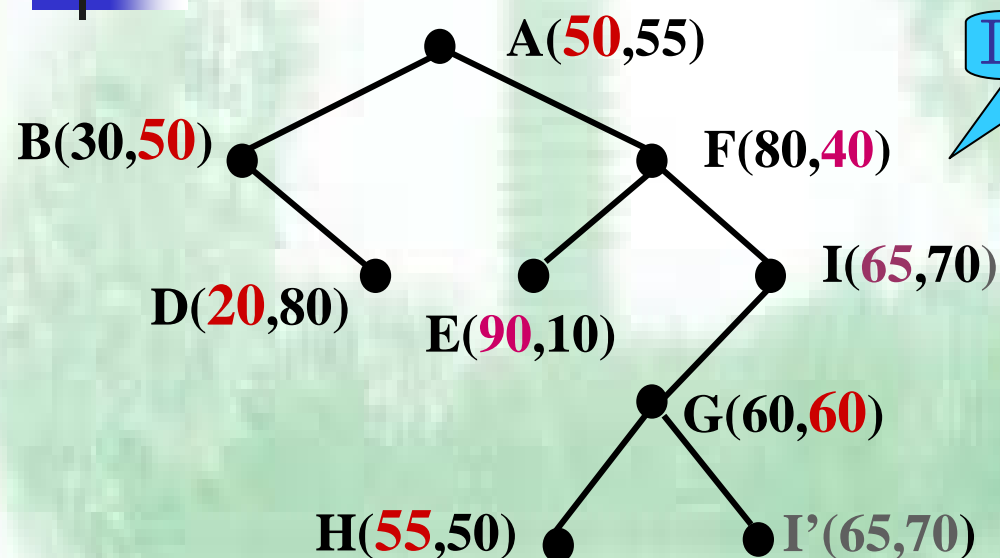
- 从当前维(y)来考虑替代结点
 - 左子树中当前维值最大的
 - 或右子树中当前维值最小的

K-D树的删除图示



- 从当前维(y)来考虑替代结点
 - 右子树中当前维值最小的

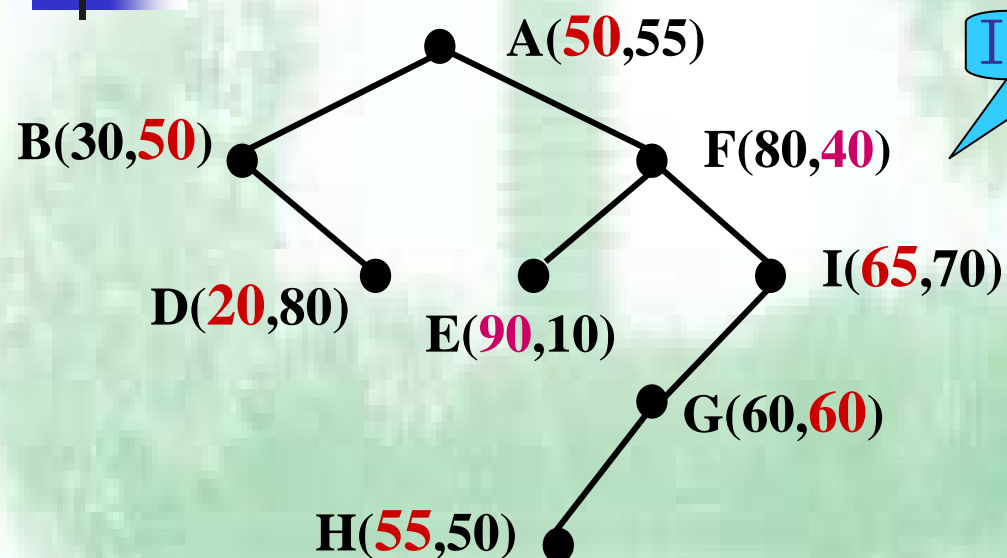
K-D树的删除图示



删除结点C

- 从当前维(y)来考虑替代结点
 - 右子树中当前维值最小的

K-D树的删除图示



删除结点C

- 从当前维(y)来考虑替代结点
 - 右子树中当前维值最小的



K-D树的范围查找

- 使用k-d树可以进行范围查找
 - 欧氏距离

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



查找一定范围的结点

- 识别器的关键码大于这个范围了
 - 那么该结点的右子树就不用搜索了
 - 因为它们必然都大于这个范围
- 同理，如果识别器的关键码小于这个范围
 - 该结点的左子树都不用搜索



K-D树的不足

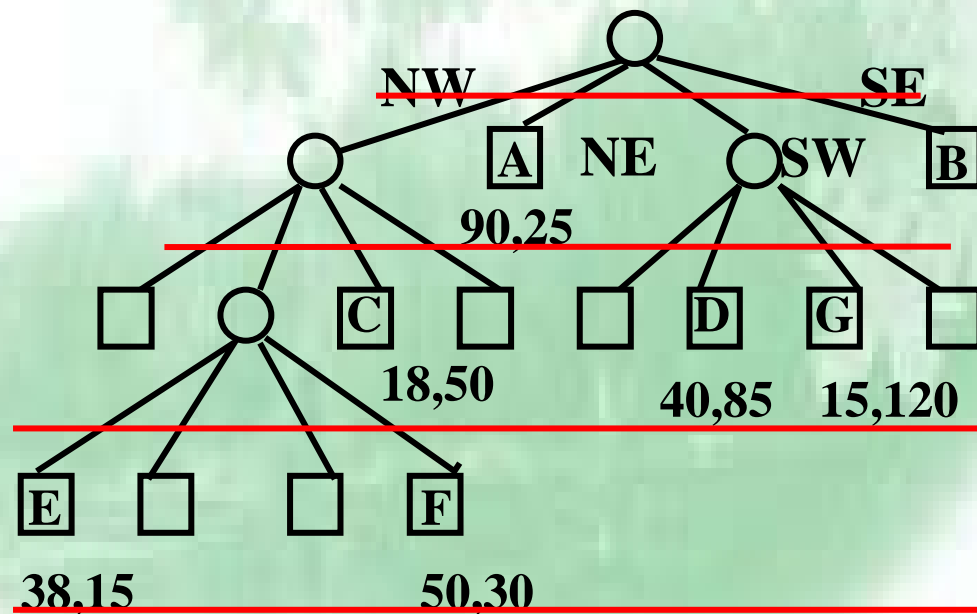
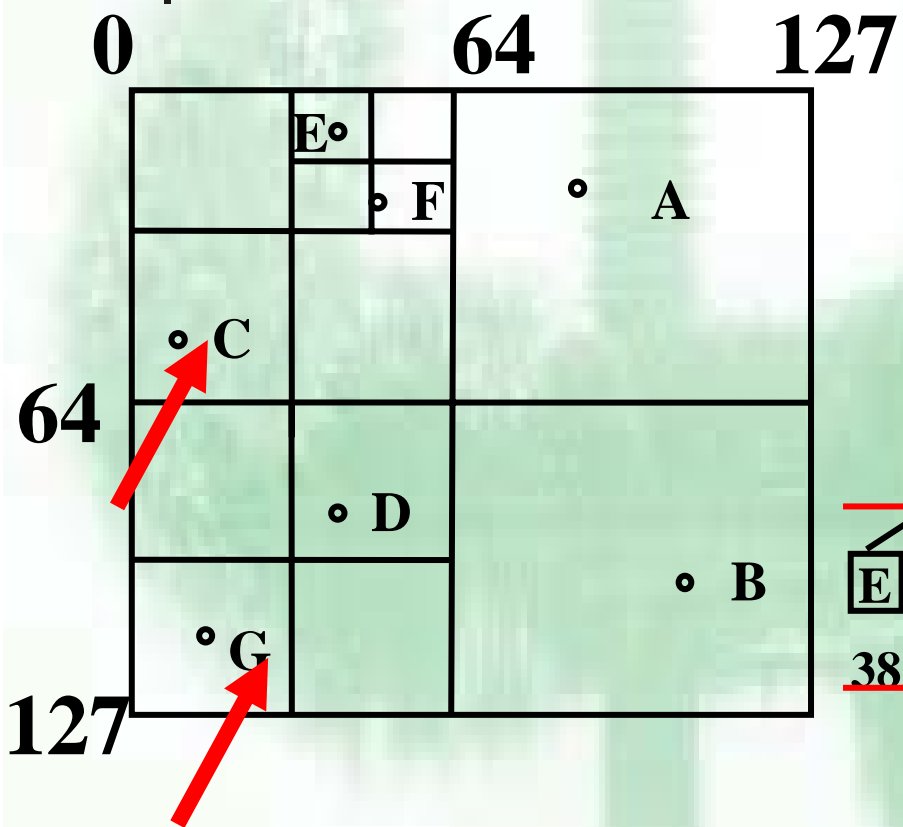
- 其结构与输入数据的顺序也是有关的
 - 有可能导致它每个子树的元素分配不均衡
- **Bentley和Friedman发明了adaptive k-d树**
 - 类似于BST
 - 所有的数据记录都存储在叶结点
 - 内部结点只是用来在各个维之间导航
 - 每一个识别器的选择不再依赖于输入的数据
 - 尽量选择让左右子树的记录数目相等的值

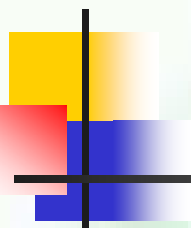


12.3.2 PR四分树

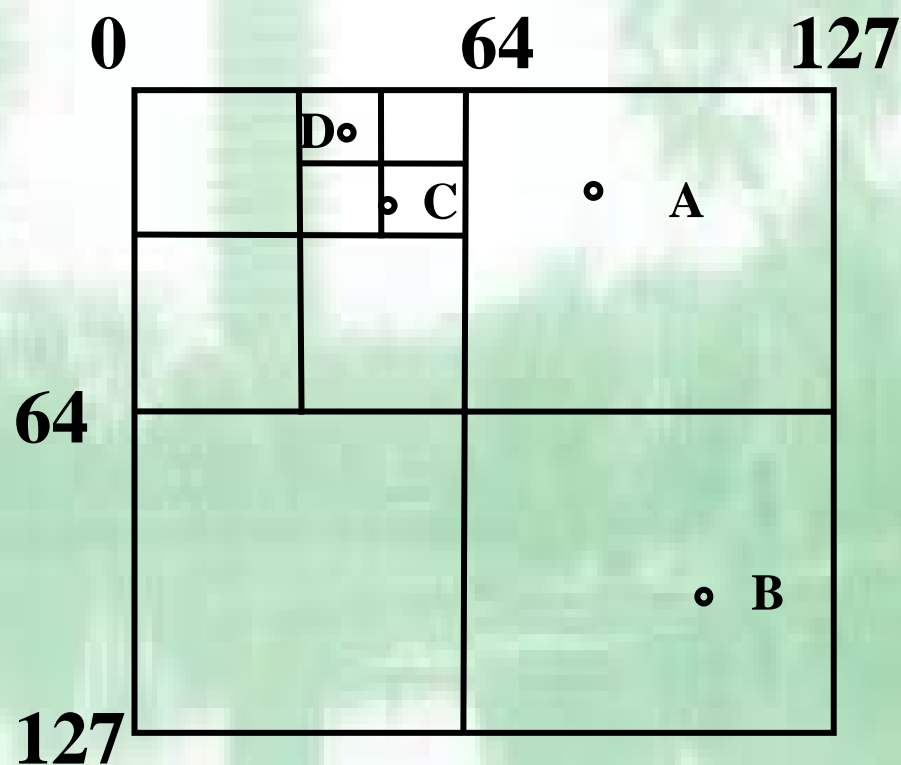
- 即点-区域四分树（**Point-Region Quadtree**）：
 - 每个内部结点都恰好有四个子结点
 - 将当前空间均等地划分为四个区域
 - NW（西北）、NE（东北）、SW（西南）和SE（东南）
- **PR四分树也是对对象空间的划分**
- **满四叉树**

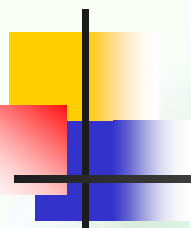
顶层空间平分为4份:
NW,NE,SW,SE



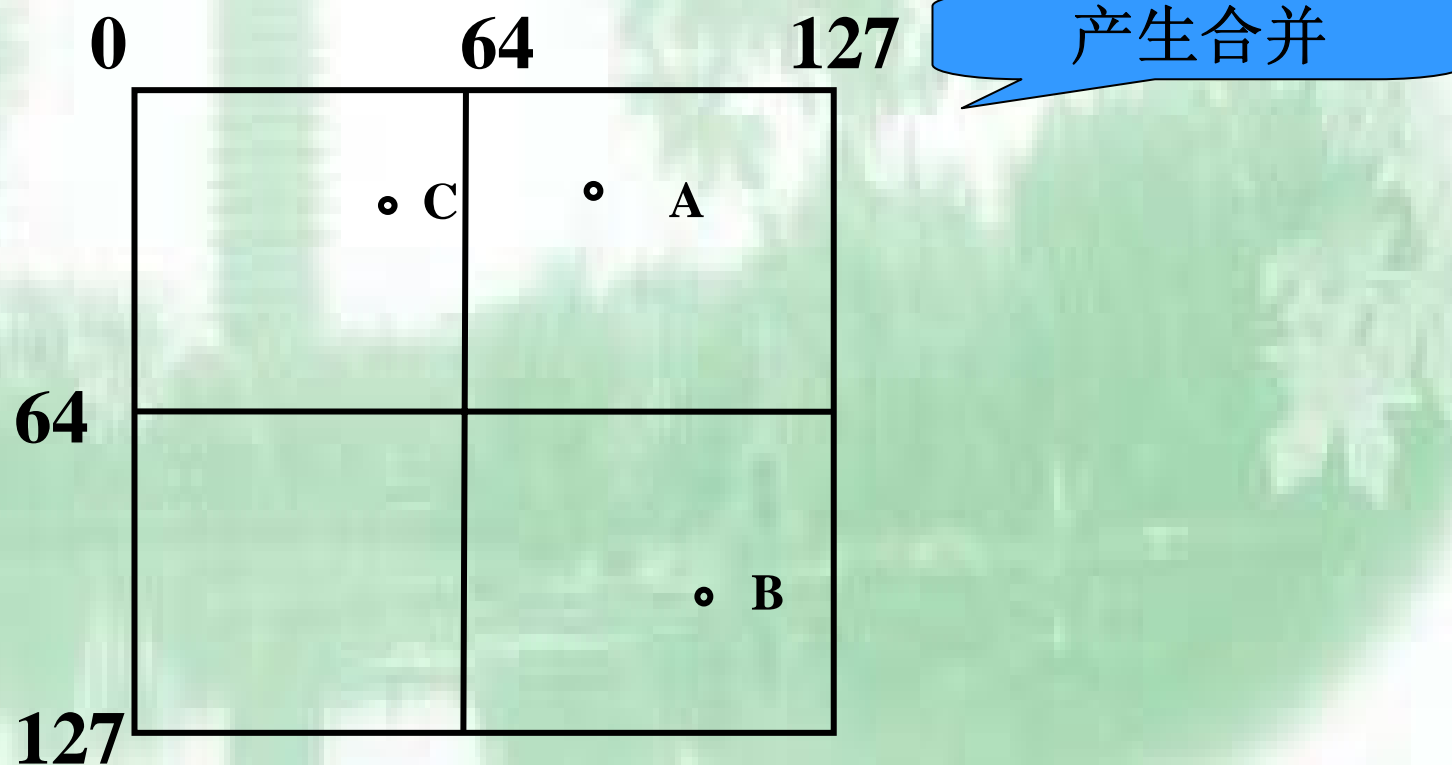


PR树的删除产生的合并





PR树的删除产生的合并

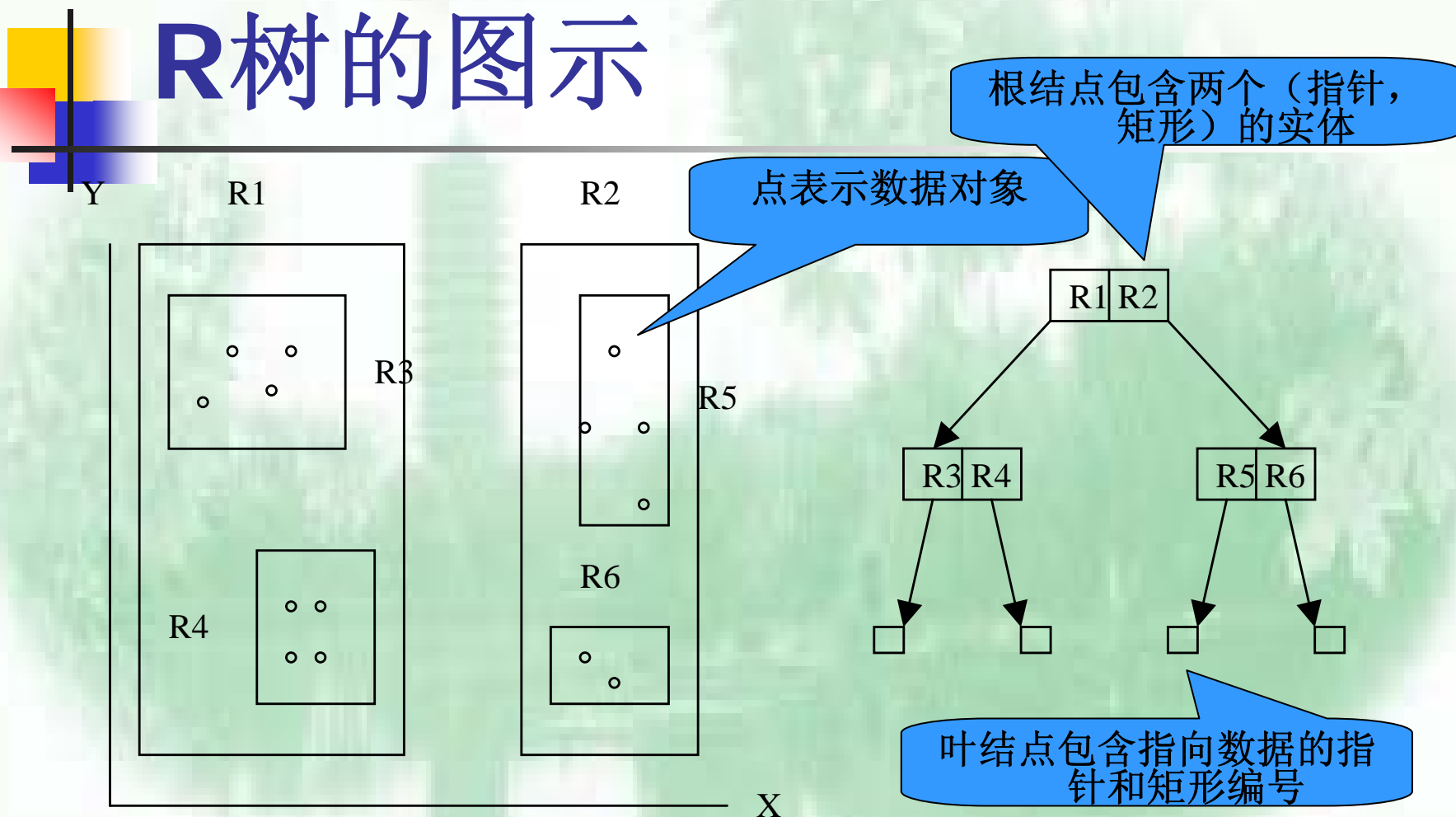




12.3.3 R*树

- **R*树**是一种用于处理多维数据的数据结构
 - 用来访问二维或者更高维区域对象组成的空间数据
 - 1990年由**N. Bechmann**, **H. Kriegel**, **R. Schneider**和 **B. Seeger**提出
- **R*树**是对**R树**的结构改进

R树的图示



左边图的矩形和右边的编号对应；右边是构建好的一棵R树



R树的性质

假设 m 是R树中一个结点所包含的最少实体个数，而 M 是R树中结点可以包含的最多实体个数

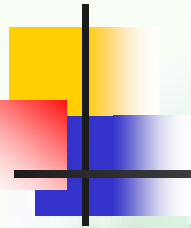
■ R树应该满足下面的条件：

- 1. 根结点如果不是叶结点，那么应该至少有2个子结点；
$$2 \leq m \leq M / 2$$
- 2. 每个既不是叶结点也不是根结点的内部结点所具有的子结点数在 m 至 M 之间；
- 3. 每个叶结点有 m 至 M 个实体，除非它是根结点
- 4. 所有的叶结点都处在同一层

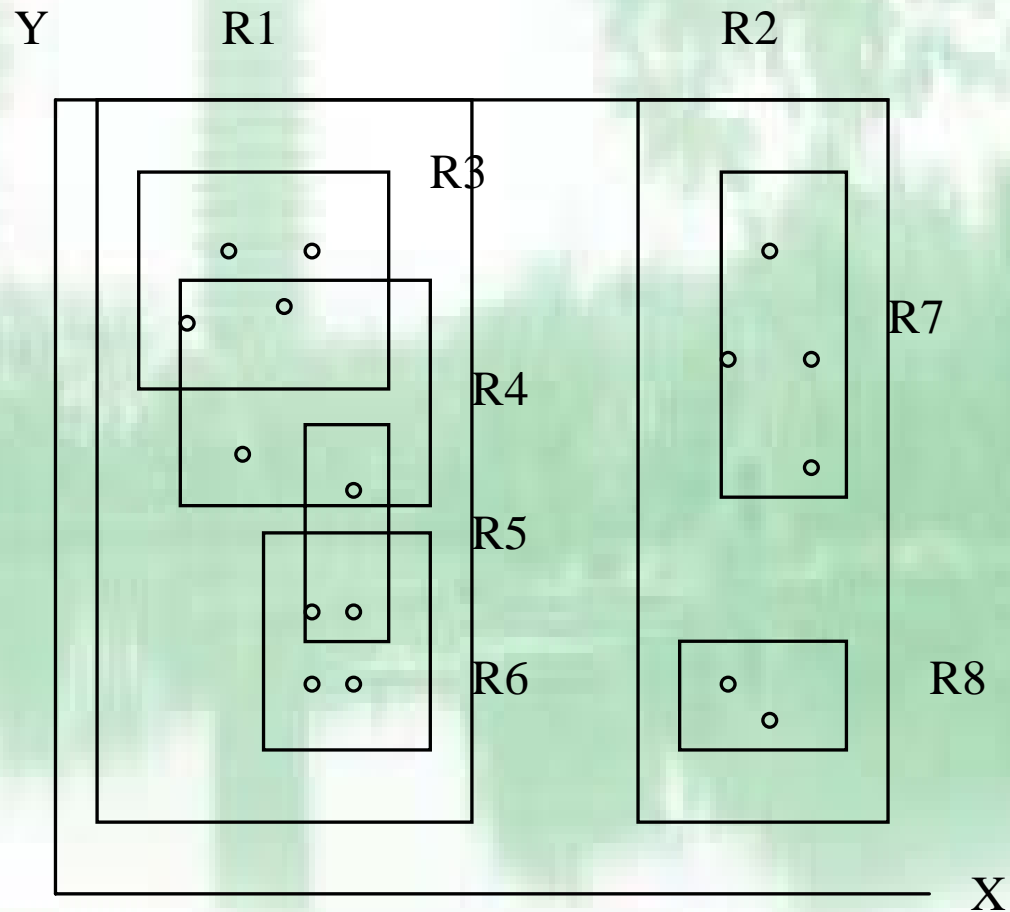


矩形重叠覆盖

- **R树的主要问题就是矩形重叠覆盖**
 - **R树要支持数据库中的各种复杂查询**
 - **如果一个查询所需要的结果都在一个矩形区域里面那么速度将是最快的**
- **很难找到一种合适的方法来确定单个矩形的大小**
 - **各个矩形之间复杂的关系**
 - **调整一个矩形的大小必然会影响到很多与它相关的其他矩形**



R树中的矩阵重叠





影响R树性能的因素

- 覆盖某个区域的矩形应该最小化
- 矩形之间的重叠应该最小化
 - 进行检索的时候能够减少需要搜索的路径
- 矩形的周长应该尽可能的小



树形结构的应用

应用非常广泛

- 建立索引（例如**BST**树、**B⁺**树）
- 问题建模
 - 决策树
 - 博弈树



12.4.1 决策树

- 决策问题就是要求根据一些给定条件来确定应采取什么决策行动
- 例如，在保险公司的业务中，当一个顾客申请汽车保险时，公司按如下规则根据顾客的条件决定是否接受他的申请，以及向他收多少保险费：

决策表

如果一个顾客年龄超过25岁，但在过去3年中出过两次或两次以上的交通事故；或年龄不超过25岁且未婚，但在过去3年中出过两次以下的交通事故，则除向他收基本保险费外，还要加收50%的附加保险费

		1	2	3	4	5
1	已婚	—	N	—	N	Y
2	年龄 ≤ 25 岁?	Y	Y	N	N	—
3	过去3年中 \geq 两次交通事故?	Y	N	Y	N	N
4	不接受	X				
5	收基本保险费		X	X	X	X
6	收附加保险费		X	X		



决策表转换成二叉决策树

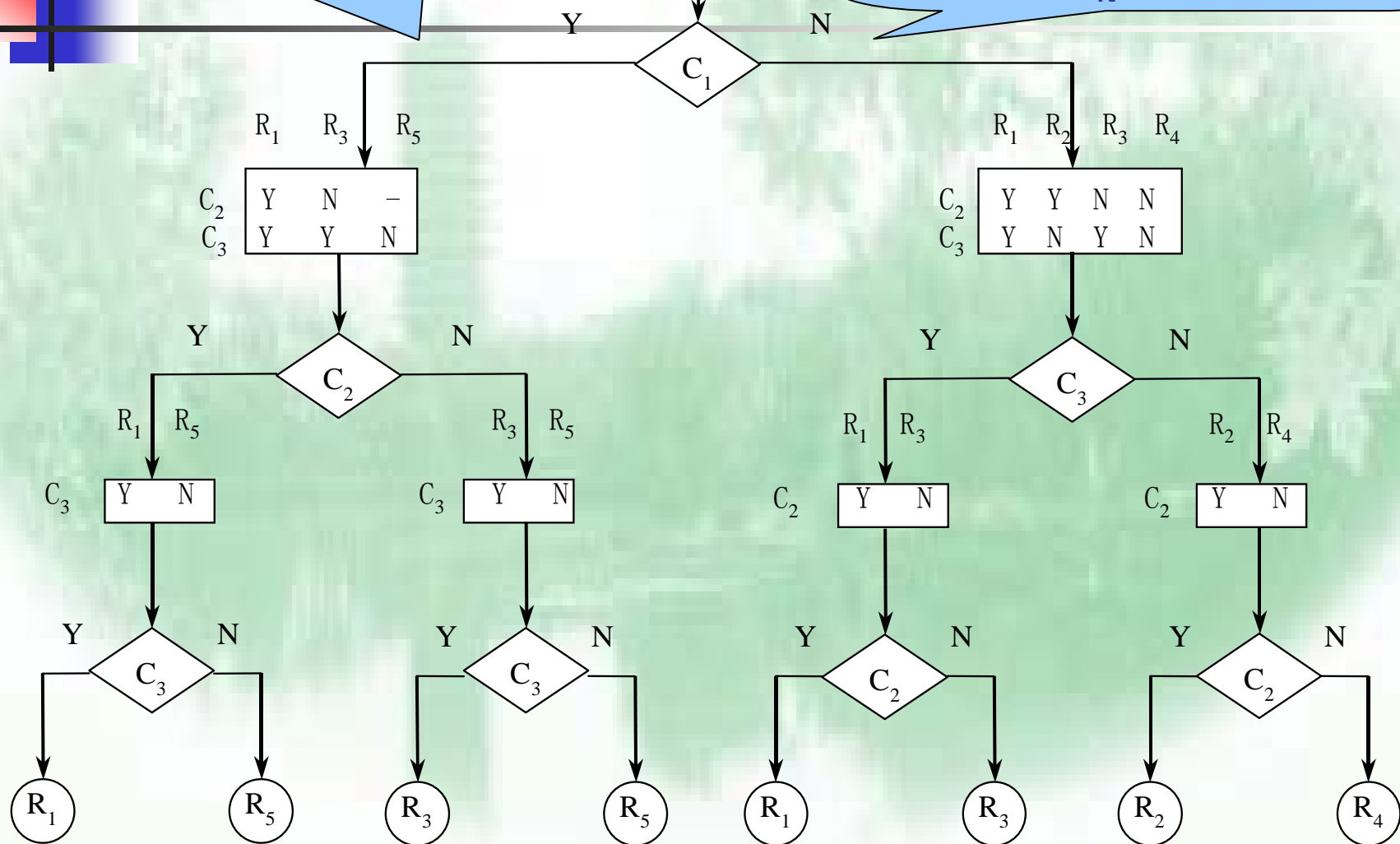
- 分支结点代表条件测试，树叶代表选定的规则
 - 令 C 是条件矩阵
- 反复把原来的问题分解为两个较小的子问题，直至 C_Y 或 C_N 中只有一行时就可作出最后的决定：选择某个条件项 C_i ，将 C 中满足条件
 - $C[i, j]='Y'$ 或 $C[i, j]='-'$ 的那些列 j 构成 C_Y (去掉第 i 行)
 - 将 C 中满足条件 $C[i, j]='N'$ 或 $C[i, j]='-'$ 的那些列构成 C_N (去掉第 i 行)
 - 测试 C_i 作为根结点，并把 C_Y 和 C_N 作为根 C_i 的两个子结点

将决策表变为决策树的过程

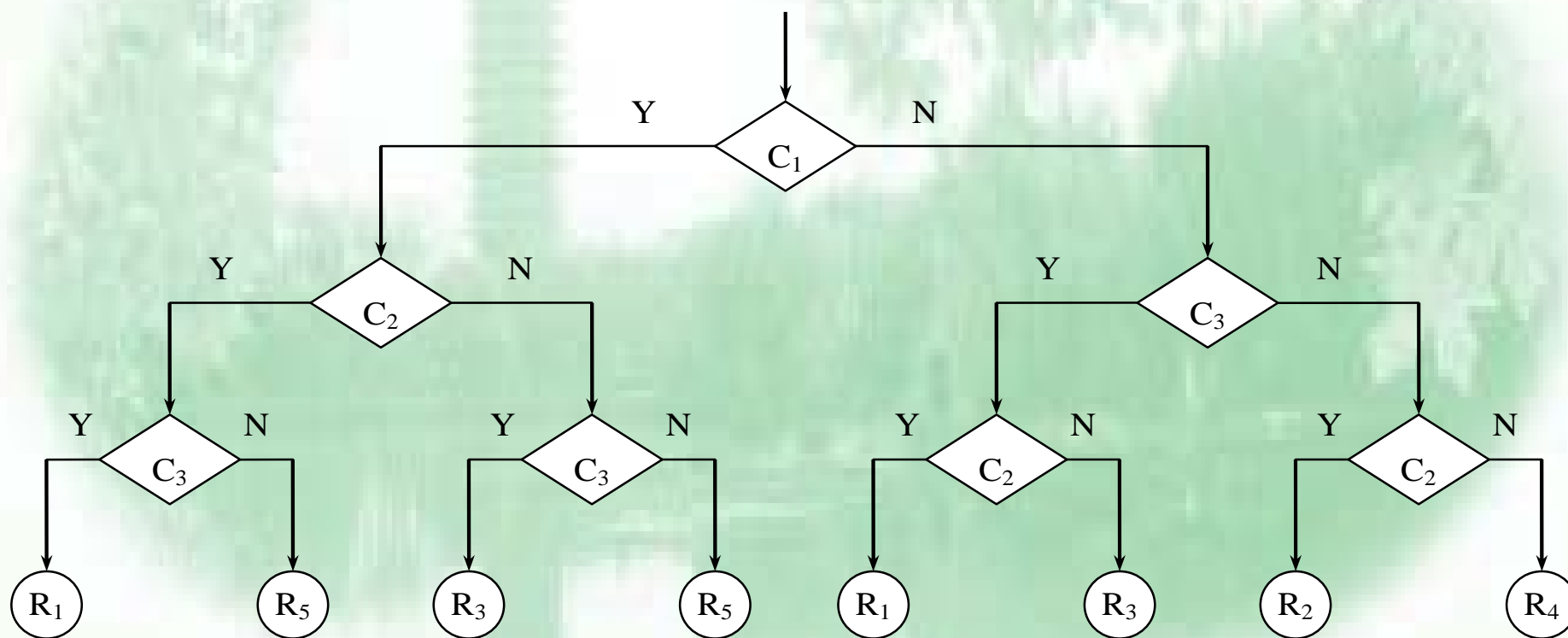
$C[i, j] = 'Y'$ 或 $C[i, j] = '-'$ 的那些列 j 构成 C_Y (去掉第 i 行)

	R_1	R_2	R_3	R_4	R_5
C_1	-	N	-	N	Y
C_2	Y	Y	N	N	-
C_3	Y	N	Y	N	N

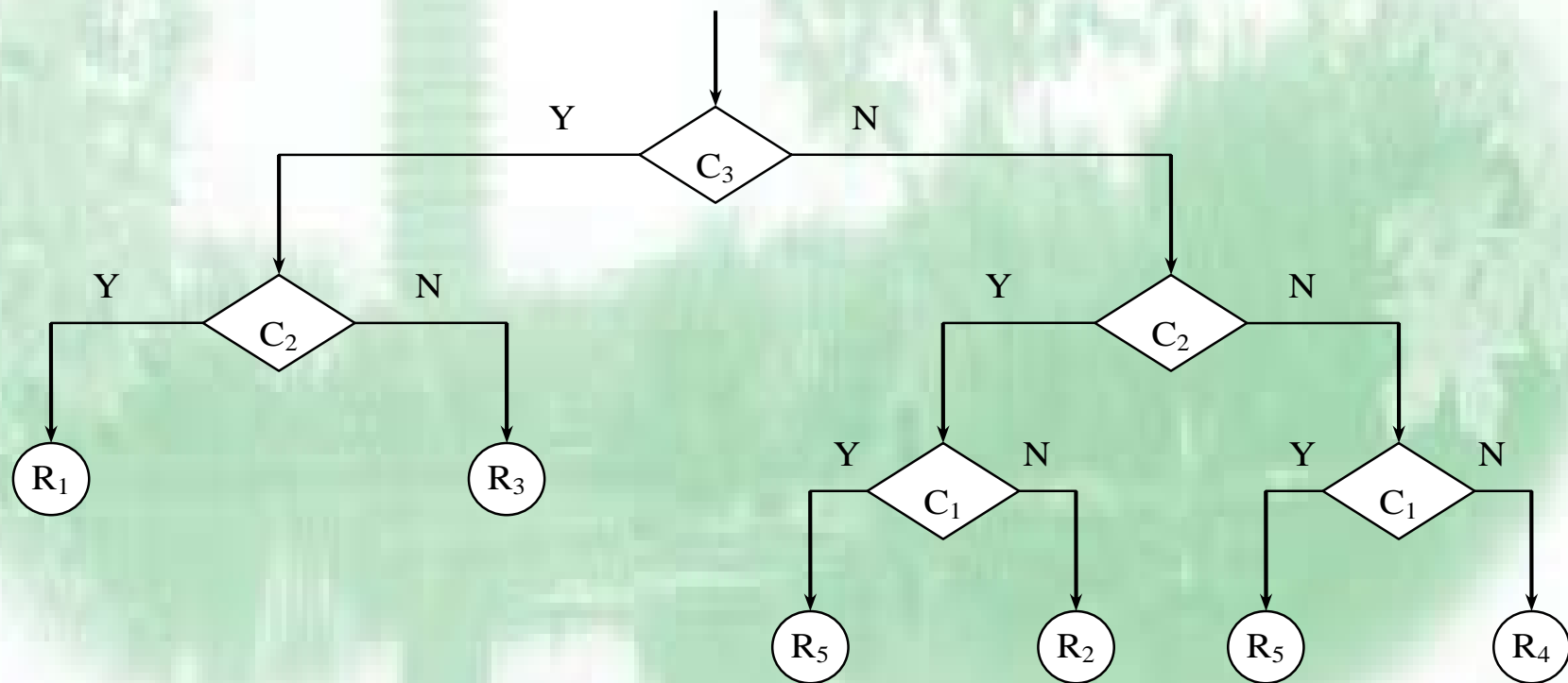
将 C 中满足条件 $C[i, j] = 'N'$ 或 $C[i, j] = '-'$ 的那些列构成 C_N (去掉第 i 行)



由决策表最后得到的决策树



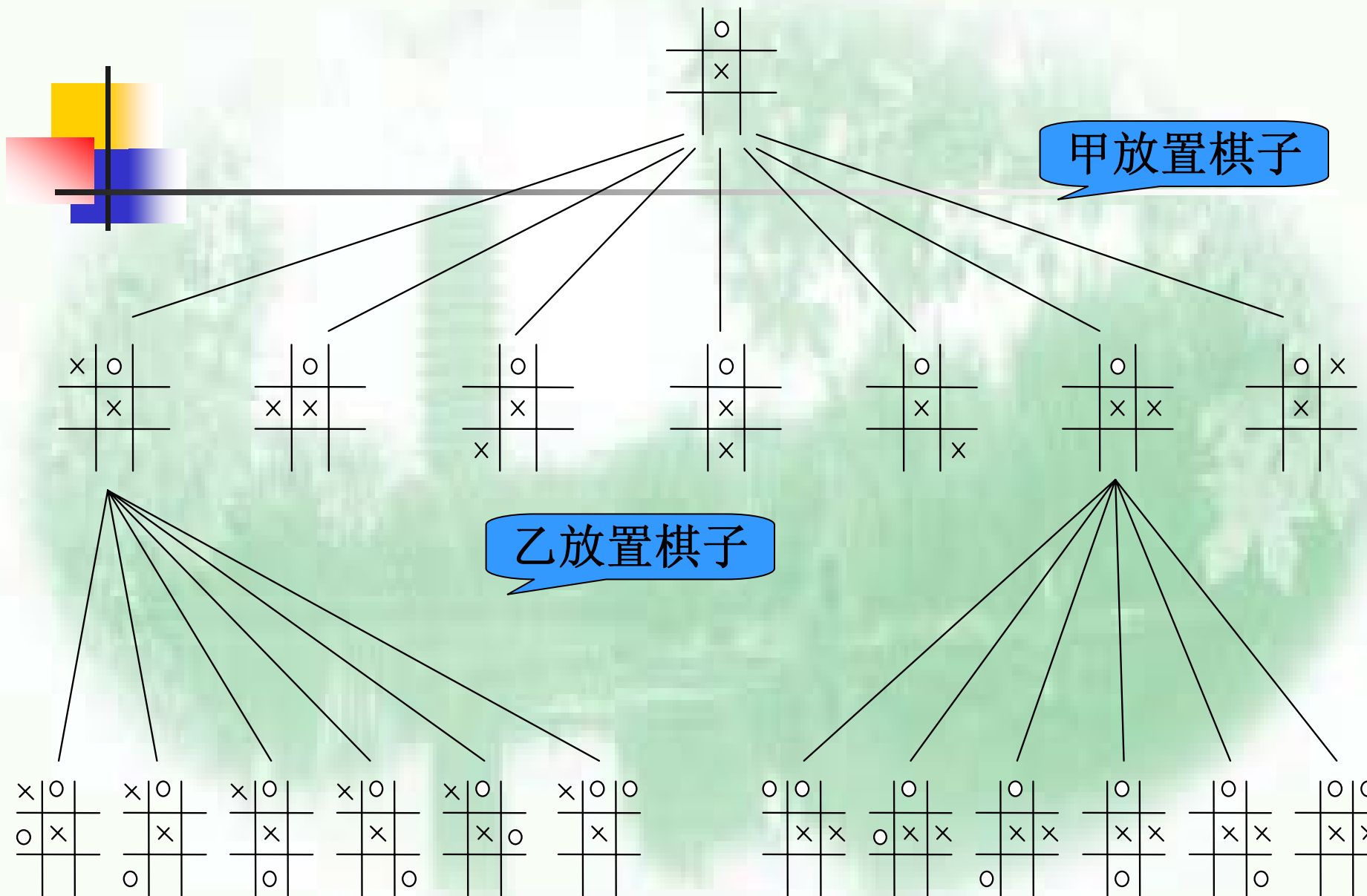
去掉多余的比较步骤





12.4.2 博弈树

- 博弈树（**game tree**）最初主要应用在人机对弈的棋类程序中
- **tic-tac-toe**的游戏规则
 - 从一个空的棋盘开始，甲乙二人轮流往棋盘上的空格子中放棋子
 - 若某一方者有三枚棋子连成一横行，或一竖列，或一对角线，则该方获胜
 - 若直至整个棋盘被占满还没有一方获胜，则为平局





根据博弈树找到最佳行为

- 假设计算机作为游戏者甲（树根），与他的对手游戏者乙进行tic-tac-toe游戏
- 下棋者总是希望赢棋的，选择对它最有利，也就是使它获胜的可能性最大的那种下法。
- 设计一个估值函数 $E(x)$
 - 此以棋盘状态 x 为自变量
 - 函数值代表此棋盘状态对计算机有利的程度
- 一个棋盘状态对计算机越有利，其 $E(x)$ 值越高
 - 计算机获胜的终局棋盘状态 $E(x)$ 取最高值



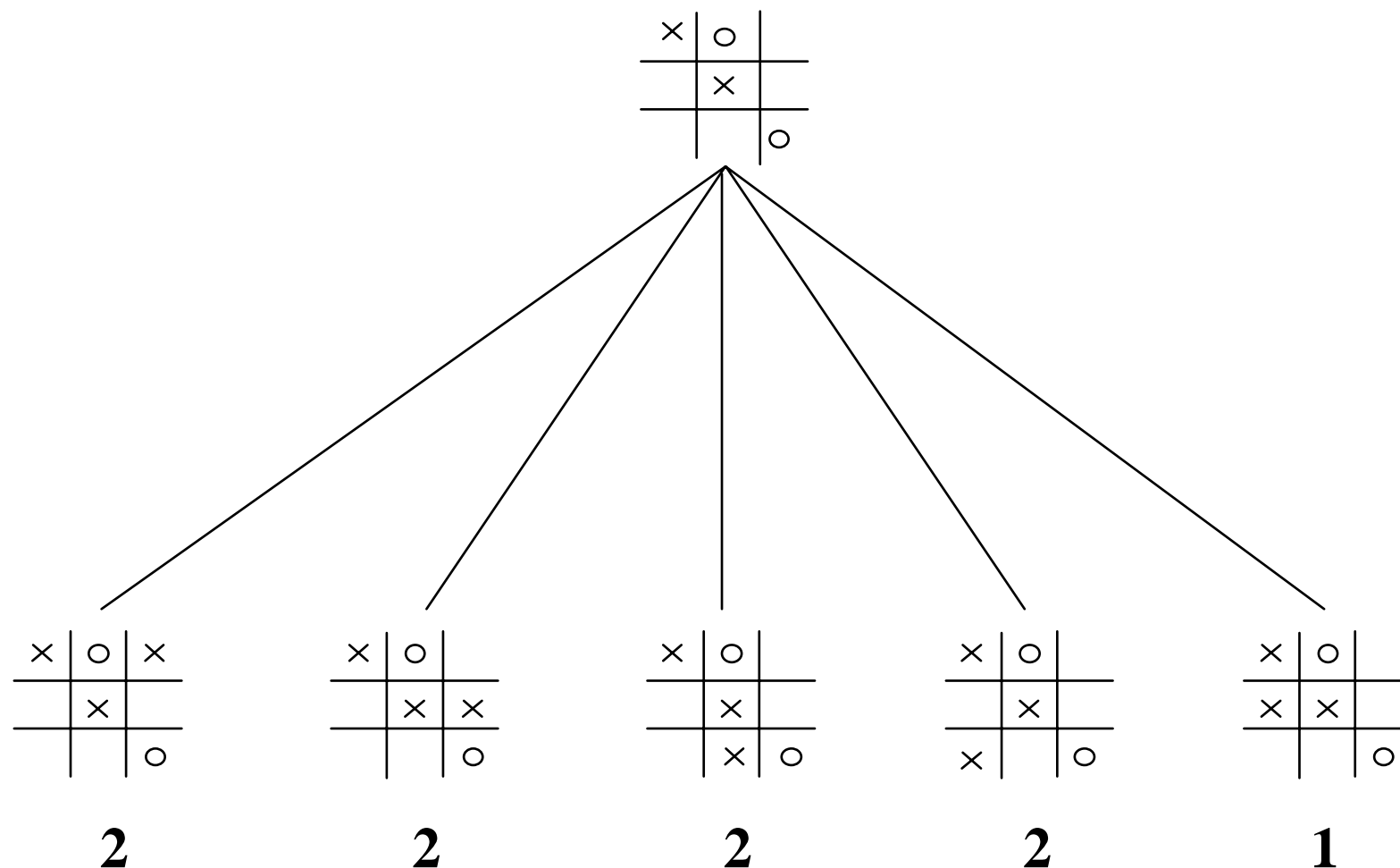
估值函数

- 估值函数的好坏是下棋程序是否成功的一个关键。
- 对于tic-tac-toe游戏我们可以确定这样的估值函数：

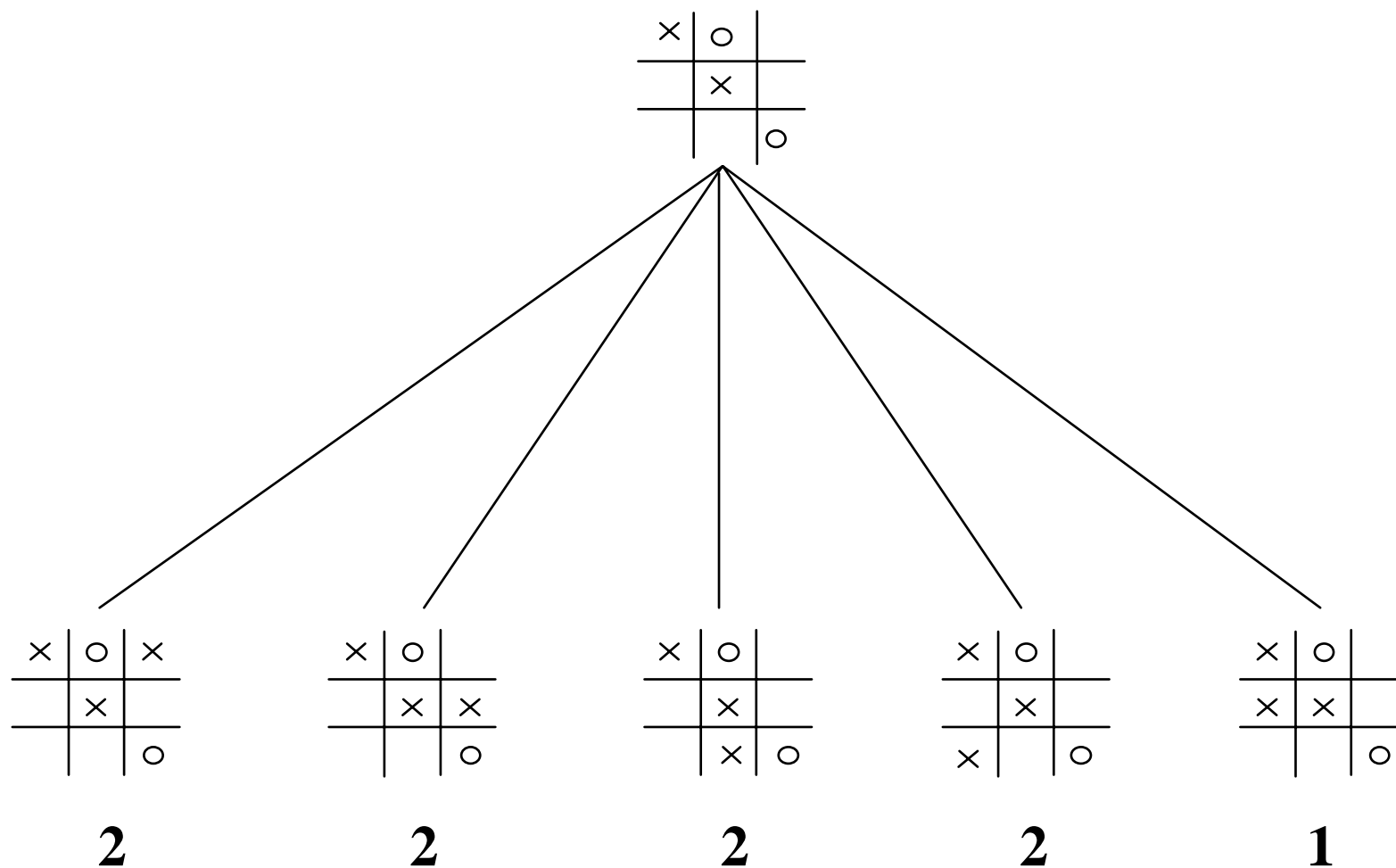
$$E(x) = \begin{cases} +\infty, & x \text{ 是计算机获胜的终局棋盘状态} \\ -\infty, & x \text{ 是对方获胜的终局棋盘状态} \\ RCDC - RCDO, & x \text{ 是其他棋盘状态} \end{cases} \quad (\text{公式2.9})$$

- 其中**RCDC**表示计算机还有可能占据的行、列、对角线数之和，**RCDO**表示对方还有可能占据的行、列、对角线之和。

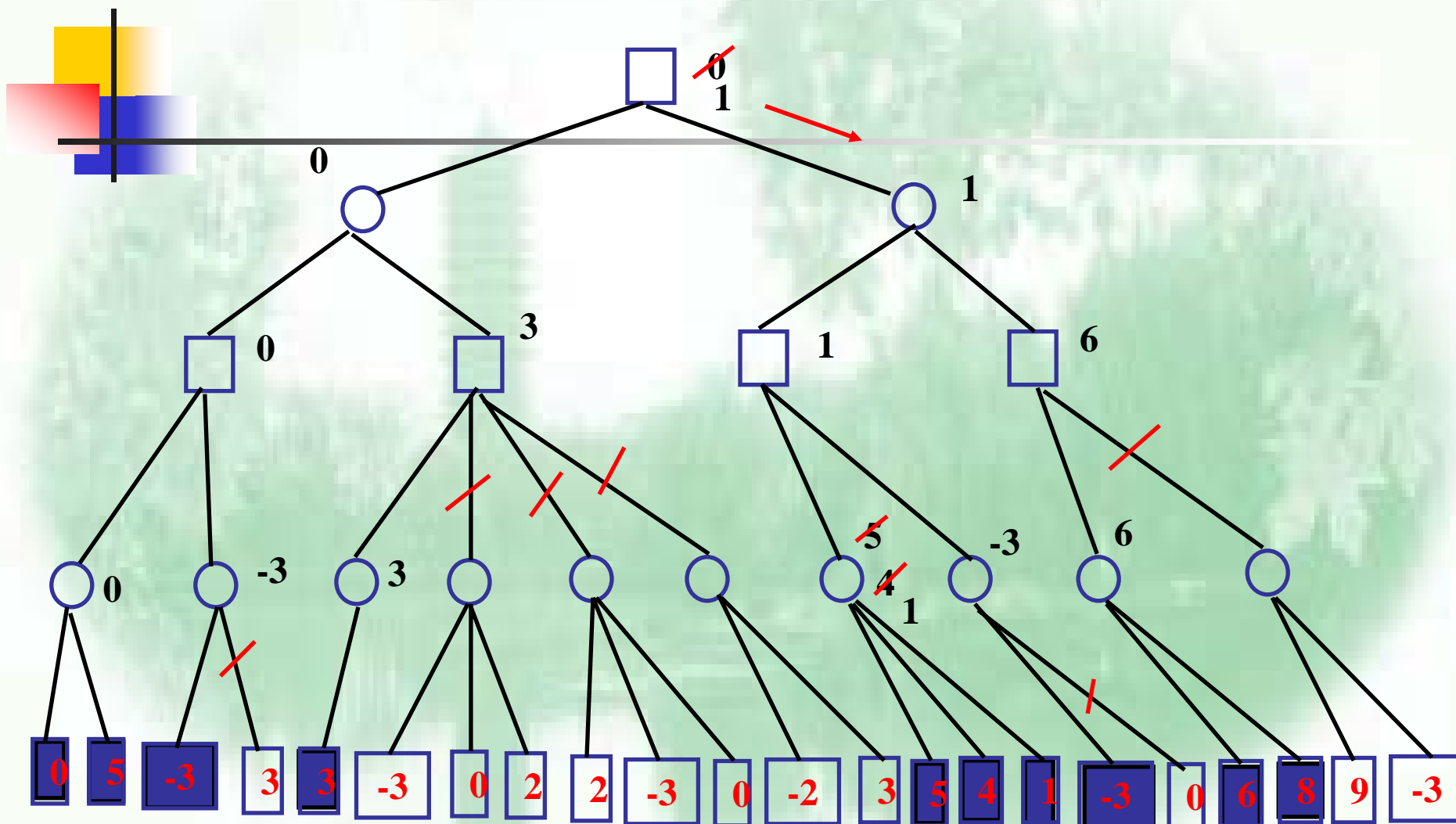
带有估值得博弈树



带有估值得博弈树



α - β 剪枝 (续)





主要内容

- 12.1 Trie和Patricia 结构
- 12.2 改进的BST
 - 最佳二叉搜索树
 - AVL树
 - 伸展树
- 12.3 空间树结构
- 12.4 决策树和博弈树



祝大家

圣诞快乐！

新年如意！

前程似锦！