

## 第十二章 高级树结构

任课教员：张铭、赵海燕、冯梅萍、王腾蛟  
<http://db.pku.edu.cn/mzhang/DS/>  
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

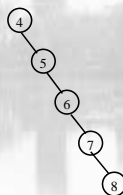
### 主要内容

- 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

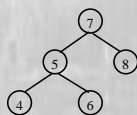
### 12.1Trie结构和Patricia树

- BST（二叉搜索树）不是一棵平衡的树，它的树结构与输入数据的顺序有很大的关系

输入顺序为 4、5、6、7、8



输入顺序为 7、5、4、6、8



### 对象空间( object space )分解

- BST是一种组织上基于对象空间( object space )分解的数据结构
  - 空间分解是存储在树中的对象（关键码值）决定
- 最简单的方法就是对对象（这里是关键码）的范围进行划分
  - 平均划分
  - 按照某种方式不均匀划分

### ■ 假设划分因子为2

- 每次仅把当前范围分为两部分（对应于二叉树）
  - 在进行插入的时候，只要是小于结点的键码的都在左子树中
  - 只要是大于结点的键码的都在右子树中

### 关键码空间（ key space ）分解

- 不依赖于键码的插入顺序
- 树的深度受到键码精度的影响
  - 最坏的情况下，深度等于存储键码所需要的位数
- 例如，如果键码是0到255之间的整数，那么键码的精度就是8个二进制位。
  - 如果有两个键码：10000010和10000011，它们的前面7位都是相同的
  - 所以直到第8次划分才能将这两个键码分开
  - 这样的搜索树深度也为8，但这是最坏的情况

- 基于关键码范围的分解
- 保证平衡吗？
  - 显然是不行的
  - 如果关键码的分布得很不平衡，将导致树的结构失衡
    - 一种极端的情况，导致所有的关键码都小于根结点，那么以该结点为根的子树的右子树将没有任何的元素

## Trie结构

- 基于关键码分解的数据结构，叫作Trie结构
  - “trie”这个词来源于“retrieval”
- 主要应用
  - 信息检索（information retrieval）
  - 用来存储英文字符串，尤其大规模的英文词典
    - 自然语言理解系统中经常用到

## Trie结构的适用情况

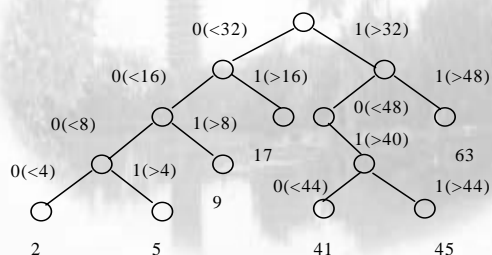
- Trie结构主要基于两个原则
  - 有一个固定的关键码集合
  - 对于结点的分层标记
- 如果所有的元素都可以使用数字或者字母等来标记，那么就可以考虑使用Trie结构
- 例如，元素可以用0-9的数字来标记
  - 在根结点的地方，它分出10个子结点，分别标记0-9
  - 然后每个子结点又可以分出10个结点
  - 如此下去直到所有的元素都能够被区分开

## Trie结构特点

- 与B+树一样，基于关键码空间分解的树结构，其内部结点仅作为占位符引导检索过程，数据纪录只存储在叶结点中
- Huffman编码树（Huffman coding tree）也是一种二叉Trie树

## Trie结构示意图

元素为 2、5、9、17、41、45、63

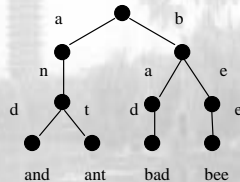


## Trie结构 应用：“字符树”

- 存储字典里面的单词
- 英文的单词是有26个字母组成的（简单起见，我们忽略大小写），
  - 英文字符树每一个内部结点都有26个子结点
  - 树的高度为最长字符串长度

## 英文字符树

存储单词 and、ant、bad、bee

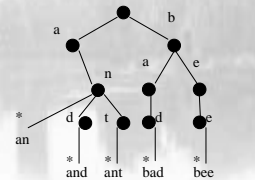


- 一棵子树代表具有相同前缀的关键词的集合。例如“an”子树代表具有相同前缀an-的关键词集合{and, ant}

## 字符树的改进

- 由于单词可能不等长，所以更好的存储是其内部结点不存储单词信息，只有叶结点才存储单词信息

存储单词 an、and、ant、bad、bee

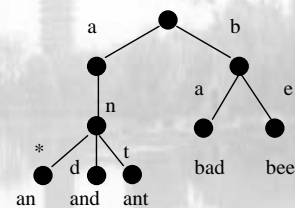


## 字符树的改进（续）

- 观察上图中的bad和bee两个单词，它们的父结点都只有一个儿子就是它们自己，也就是说，实际上它们在父结点的时候就已经被区分开了
- 因为我们真正想做的是要检索每一个单词，不需要在已经能够区分每个单词的情况下继续进行分裂结点的动作

## 字符树的改进（续）

存储单词 an、and、ant、bad、bee



## 字符树中的检索

- 首先用待查关键词的第一个字符与树林的各个根的字
- 符相比较
- 然后下一步的检索在前次比较相等的那棵树上进行其中，用待查关键词的第二个字符与选定的这棵树的根的各个子结点进行比较，
- 接着再沿着前次比较相等的分支进行进一步的检索
- ...
- 直到进行到某一层，该层所有结点的字符都与待查关键词相应位置的字符不同，这说明此关键词在树目录里没有出现；
- 若检索一直进行到树叶，那么就在树目录里找到了给定的关键词

## Trie字符树的缺陷

- Trie结构显然也不是平衡的
  - 在它存取英文单词的时候，显然t子树下的分支比z子树下的分支多很多（以t开头的单词比以z开头的单词多很多）
  - 而且，每次有26个分支因子使得树的结构过于庞大，给检索也带来了不便
- 字母在计算机中是以二进制ASCII码的形式存储的
  - 使用每个字母的二进制编码来代表这个字母
  - 关键词就只有编码0和1
  - 而不是a到z的26个字母
- 二叉Trie树

## Trie树的插入

### ■ Trie树的插入

- 首先根据插入纪录的关键码找到需要插入的结点位置
- 如果该结点是叶结点，那么就将其分裂出两个子结点，分别存储这个纪录和以前的那个纪录
- 如果是内部结点，则在那个分支上应该是空的，所以直接为该分支建立一个新的叶结点即可

## Trie树的删除

### ■ Trie树的删除

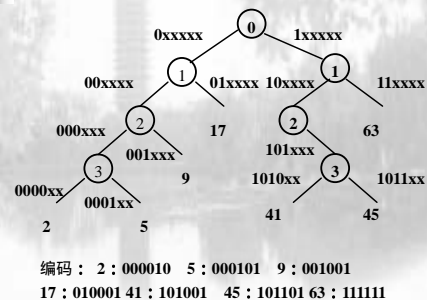
- 根据插入纪录的关键码找到需要删除的结点位置
- 如果一个被删除结点的父结点没有其他的孩子，那么就需要合并
- 否则只需要将此分支设置为空即可

## PATRICIA 结构

- 为了得到更加平衡的结构，D.Morrison发明了Trie结构的一种变体PATRICIA

- “Practical Algorithm To Retrieve Information Coded In Alphanumeric”
- 它不是对整个关键码大小范围的划分
- 而是根据关键码每一个二进制位的编码来划分
  - 因为每一位要么是0，要么是1，所以分支因子是2，
  - 生成的树是二叉树

## PATRICIA 结构图



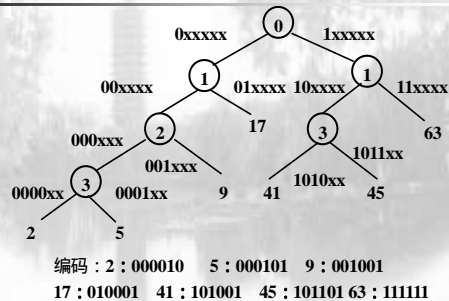
## PATRICIA 结构图（续）

- 上图因为最大的数是63，用6位二进制表示即可
- 每一个结点都有一个标号，表示它是在比较第几位，然后根据那一位是0还是1来划分左右两个子树
  - 标号为2的结点的右子树一定是编码形式为xx1xxx（x表示该位或0或1，标号为2说明比较第2位）
- 在图中检索5的话，5的编码为000101
  - 首先我们比较第0位，从而进入左子树
  - 然后在第1位仍然是0，还是进入左子树
  - 在第2位还是0，仍进入左子树
  - 第3位变成了1，从而进入右子树，就找到了位于叶结点的数字5

## PATRICIA 结构图改进

- 观察PATRICIA的图发现有与Trie图类似的情况
- 在区分2和5、41和45的时候，在第三个二进制位的比较是不能区别它们的
- 可以将它省略，得到一棵更为简洁的树

## PATRICIA 结构图改进 (续)



## PATRICIA的特点

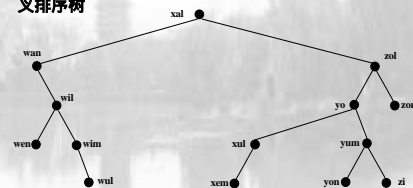
- 每个内部结点都代表一个位的比较，必然产生两个子结点
  - 所以它是一个满二叉树
- 进行一次检索，最多只需要关键码位数次的比较即可

## 12.2 二叉树结构的改进

- 平衡的二叉搜索树、伸展树和最佳二叉排序树，它们都是对二叉树的结构或者操作规则做部分的改进以使树保持在一种类似于平衡的状态，从而达到较好的效率

### 12.2.1 最佳二叉搜索树

- 根据前面章节的二叉树介绍我们知道对应于关键码集合  $K: K = \{xal, wan, wil, zol, yo, xul, yum, wen, wim, zi, yon, xem, wul, zom\}$  的二叉排序树



## 二叉搜索树的多样性

- 同一个关键码集合，其关键码插入二叉搜索树的次序不同，就构成不同的二叉搜索树。
- 包括  $n$  个关键码的集合中，关键码可以有  $n!$  种不同的排列法，因此可以构成  $n!$  个二叉搜索树 (其中有相同的)
- 可以用检索效率来衡量二叉搜索树

- 如果只计算不同的搜索树，则排列  $\{2, 1, 3\}$  的顺序插入关键码与按照排列  $\{2, 3, 1\}$  的顺序插入所构成的二叉搜索树完全相同
- 这种非前序排列的序列，总是可以找到与其相对应的一个合法前序排列
- 这  $n!$  种排列中，只有  $C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$
- 就是说  $n$  个结点可以构造  $\frac{1}{n+1} C_{2n}^n$  称为 Catalan 函数

## 扩充的二叉树

- 第4章讨论过扩充的二叉树的概念。扩充的二叉树是满二叉树，新增加的空树叶(以下称为外部结点)的个数等于原来二叉树的结点(以下称为内部结点)个数加1
- 在扩充的二叉搜索树里，关键码值最小的内部结点的左子女(外部结点)代表了值小于该内部结点的可能关键码的集合；关键码值最大的内部结点的右子女(外部结点)代表了值大于该内部结点的可能关键码的集合
- 除此之外，每个外部结点代表其值处于原来二叉搜索树的两个相邻结点的关键码值之间的可能关键码的集合

## 路径长度

- 外部路径长度E定义为从扩充的二叉树的根到每个外部结点的路径长度之和。
- 内部路径长度I定义为扩充的二叉树里从根到每个内部结点的路径长度之和

## 二叉搜索树里检索算法

- 在二叉搜索树里检索算法十分简单：首先用待查的关键码与二叉搜索树的根结点进行比较，若比较相等，则找到了要检索的关键码；
- 若比较不等，若待查关键码值小于根结点的关键码值，则下一次与根的左子树的根比较；否则与根的右子树的根比较。
- 如此递归地进行下去，直到某一次比较相等，检索成功；或一直比较到树叶都不相等，检索失败。

## 检索一个关键码的比较次数

- 在检索过程中，每进行一次比较，就进入下面一层。因此，对于成功的检索，比较的次数就是关键码所在的层数加1。对于不成功的检索，被检索的关键码属于哪个外部结点代表的可能关键码集合，比较次数就等于此外部结点的层数
- 在二叉搜索树里，检索一个关键码的平均比较次数为

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

## 参数意义

- 其中 $l_i$ ，是第 $i$ 个内部结点的层数，是第 $i$ 个外部结点的层数，
- $p_i$ 是检索第 $i$ 个内部结点所代表的关键码的频率
- $q_i$ 是被检索的关键码属于第 $i$ 个外部结点代表的可能关键码集合(即处于第 $i$ 个和第 $i+1$ 个内部结点之间)的频率。 $p_i$ ， $q_i$ 也叫做结点的权

$$W = \sum_{i=1}^n p_i + \sum_{i=0}^n q_i$$

## 最佳二叉搜索树定义

- $\frac{p_i}{W}$ 是检索第 $i$ 个内部结点所代表的关键码的概率， $\frac{q_i}{W}$ 是被检索的关键码属于第 $i$ 个外部结点代表的可能关键码集合的概率。
- 我们把检索中平均比较次数最小，也就是 $ASL(n)$ 最小的二叉搜索树称作最佳二叉搜索树

## 什么样的二叉搜索树是最佳的？

- 检索所有结点的概率都相等，即所有结点的权都相等：

$$\frac{p_1}{W} = \frac{p_2}{W} = \dots = \frac{p_n}{W} = \frac{q_0}{W} = \frac{q_1}{W} = \dots = \frac{q_n}{W} = \frac{1}{2n+1}$$

$$\begin{aligned} ASL(n) &= \frac{1}{2n+1} \left( \sum_{i=1}^n (l_i + 1) + \sum_{i=0}^n l'_i \right) \\ &= \frac{1}{2n+1} \left( \sum_{i=1}^n l_i + n + \sum_{i=0}^n l'_i \right) \\ &= \frac{1}{2n+1} (I + n + E) = \frac{2I + 3n}{2n+1} \end{aligned}$$

- 因此，要平均比较次数 $ASL(n)$ 最小，就是要内部路径长度 $I$ 最小

- 在一棵二叉树里，路径长度为0的结点仅有一个，路径长度为1的结点至多有两个，路径长度为2的结点至多有四个，等等。因此，有 $n$ 个结点的二叉树其内部路径长度 $I$ 至少等于序列：  
0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, ...的前 $n$ 项和。这个和写成：

$$\sum_{k=1}^n \lfloor \log_2 k \rfloor$$

- 可以证明： $\sum_{k=1}^n \lfloor \log_2 k \rfloor = (n+1) \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2$

- 这种二叉树的特点是只有最下面的两层结点的度数可以小于2，其它结点度数必须等于2。在所有结点的权相等的情况下，这样的二叉搜索树是最佳二叉搜索树，对它进行检索的平均比较次数为

$$ASL(n) = \frac{2(n+1) \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 4 + 3n}{2n+1} \quad (\text{公式(2.3)})$$

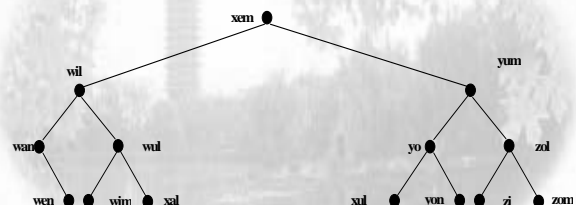
- 这个 $ASL(n)$ 是 $O(\log_2 n)$ 量级的

## 最佳二叉搜索树构造举例

- 首先将集合 $K$ 里的关键字排序 {wan, wen, wil, wim, wul, xal, xem, xul, yo, yon, yum, zi, zol, zom}
- 然后用二分法依次检索这些关键字，并把在检索中遇到的在二叉搜索树里还没有的关键字依次插入二叉搜索树中。
- 首先检索序列中的第一个关键字wan，用二分法检索wan的过程中会依次遇到关键字xem, wil, wan，这就是最先插入二叉搜索树的三个关键字。

## 最佳二叉搜索树构造举例

- 然后检索序列中的第二个关键字wen，用二分法检索wen的过程中会依次遇到关键字xem, wil, wan, wen，其中只有 wen是二叉搜索树中还没有的，因此第四个插入到二叉搜索树中的关键字是wen。
- 再检索序列中的第三个关键字wil，...，如此进行下去，直到所有的关键字都已插入到二叉搜索树中，这样可得到最佳二叉搜索树。



## 二叉搜索树的效率

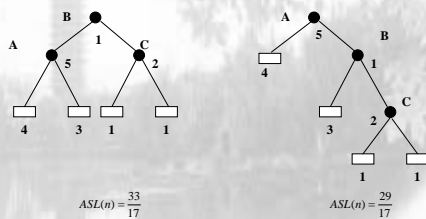
- 反过来，如果关键码按值递增的顺序依次插入到二叉搜索树中，则将得到退化为线性的二叉搜索树平均比较次数为 $O(n)$
- 按任意的顺序把关键码插入到二叉搜索树中，它的检索效率如何呢？平均比较次数是接近最坏的情况 $O(n)$ 呢，还是接近最好的情况 $O(\log_2 n)$ ？可以证明，对 $n!$ 种二叉搜索树进行平均，得到的平均检索次数仍是 $O(\log_2 n)$

## 检索各结点的概率不相等的情况

- 检索各结点的概率不相等的情况，即在具有不等权结点的二叉搜索树里进行检索。
- 现在的问题是给了一个排好序的关键码集合 $\{key_1, key_2, \dots, key_n\}$ ，和权的集合 $\{p_1, p_2, \dots, p_n, q_0, q_1, \dots, q_n\}$ ，要找使得 $ASL(n)$ 为最小的最佳二叉排序树，也就是要找使得

$$\sum_{i=1}^n p_i(l_i + 1) + \sum_{i=0}^n q_i l'_i$$

为最小，上式也称为二叉排序树的开销



## 最佳二叉搜索树的构造方法

- 最佳二叉搜索树有个特点：它的任何子树都是最佳二叉搜索树。
- 这一事实引导我们可以用这样的方法构造越来越大的最佳二叉搜索树：先构造包括一个结点的最佳二叉搜索树，再构造包括两个结点的最佳二叉搜索树，...，直到把所有的结点都包括进去。
- 后来构造的较大的最佳二叉搜索树用前面构造的较小的最佳二叉搜索树作为其子树

- 设包括关键码 $key_{i+1}, key_{i+2}, \dots, key_j$ 为内部结点 $(0 \leq i \leq j \leq n)$ ，结点的权为 $(q_i, p_{i+1}, q_{i+1}, \dots, p_j, q_j)$ 的最佳二叉搜索树为 $t(i, j)$ ，其根为 $r(i, j)$ ，其花费为 $C(i, j)$ ，其权的总和为 $W(i, j) = p_{i+1} + \dots + p_j + q_i + q_{i+1} + \dots + q_j$
- 第一步构造包括一个结点的最佳二叉搜索树，就是找 $t(0, 1), t(1, 2), \dots, t(n-1, n)$ 。

- 第二步构造包括两个结点的最佳二叉搜索树，就是找 $t(0, 2), t(1, 3), \dots, t(n-2, n)$
- 再构造包括三个，四个，...结点的最佳二叉搜索树，直到最后构造 $t(0, n)$
- 如何构造最佳二叉搜索树 $t(i, j)$ 呢？由最佳二叉搜索树的子树必是最佳二叉搜索树的原理可知：
  - $C(i, j) = W(i, j) + \min_{i < k \leq j} (C(i, k-1) + C(k, j))$



- 这个式子的意思是，用下列方法来找包括 $key_{i+1}, key_{i+2}, \dots, key_j$ 为内部结点的最佳二叉搜索树 $t(i, j)$ ：
- 分别用 $key_{i+1}, key_{i+2}, \dots, key_j$ 为根，考虑 $j-i$ 棵二叉搜索树。以 $key_k$ 为根的二叉搜索树其左子树包括 $key_{i+1}, \dots, key_{k-1}$ ，而包括这些关键字为内部结点的最佳二叉排序树 $t(i, k-1)$ 已在前面的步骤确定， $C(i, k-1)$ 已求出，而以 $key_k$ 为根的二叉搜索树其右子树包括 $key_{k+1}, key_{k+2}, \dots, key_j$ ，以这些关键字为内部结点的最佳二叉搜索树 $t(k, j)$ 也已在前面的步骤确定， $C(k, j)$ 已求出。

- 对于 $i < k$  找出使 $C(i, k-1) + C(k, j)$ 为最小的那个 $k_0$ ，以 $key_{k_0}$ 为根， $t(i, k_0-1)$ 为左子树， $t(k_0, j)$ 为右子树的那个二叉搜索树就是所要求的 $t(i, j)$ 。其花费  $C(i, j)$  等于其根的左子树的花费 $C(i, k_0-1)$ 加上右子树花费 $C(k_0, j)$ ，再加上结点的总的权 $W(i, j)$
- 每一步构造出一棵最佳二叉搜索树 $t(i, j)$ 就记下其根 $r(i, j)$ ，花费 $C(i, j)$ ，最后构造出 $t(0, n)$ ，整个最佳二叉搜索树的结构就明确了

## 不等权结点的最佳二叉搜索树算法

```
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];
    //求出权和w[i,j]
    for(int j=i+1; j<=n; j++)
        w[i][j]=w[i][j-1]+a[j]+b[j];
}
//确定一个结点的最佳二叉排序树
for(int j=1; j<=n; j++)
{
    c[j-1][j]=w[j-1][j];
    r[j-1][j]=j;
}
```

//确定d个结点的最佳二叉树

```
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;
}
}
```

## 构造过程

给出关键码集合

$\langle B, D, F, H \rangle$   
 $\langle key_1, key_2, key_3, key_4 \rangle$

及权的序列

$\langle 1, 5, 4, 3, 5, 4, 3, 2, 1 \rangle$   
 $\langle p_1, p_2, p_3, p_4, q_0, q_1, q_2, q_3, q_4 \rangle$

计算次数为

$$\sum_{j=1}^n \sum_{i=0}^{j-1} (j-i+1) \approx \frac{n^3}{6}$$

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

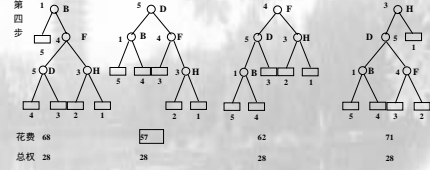
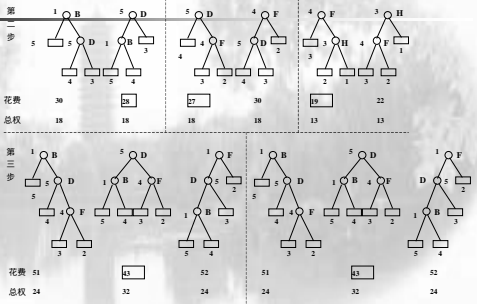
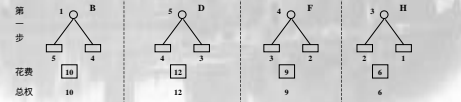
$r(i, j)$

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	13	6	
3	3	6			
4	1				

$W(i, j)$



## 动态规划 (dynamic programming)

- 动态规划方法将问题分解为若干个子问题，分别求解子问题的解，然后由这些子问题的解得到原问题的解。
- 动态规划算法的有效性依赖于问题本身所具有的两个重要性质：最优子结构性质和子问题重叠性质
  - 最优子结构是指问题的最优解包含其子问题的最优解
  - 重叠子问题是指在自顶向下的递归求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次

## 动态规划与分治策略

- 动态规划与分治策略有着明显的不同。
- 分治策略要求子问题之间相互独立，可以根据最有子结构直接用递归法求解
- 而动态规划方法所处理的子问题相互交叉。直接用递归法来求解具有重叠子问题性质的后一类问题时，常常要大量重复计算公共的子问题；而动态规划对每个子问题仅仅求解一次，并将结果填写到若干张表中，下次计算同一子问题时直接利用表中的结果。
- 动态规划的具体形式虽然多种多样，但它们一般都具有“填表”这一基本模式。

## 动态规划的应用

- 动态规划方法通常用于解决优化问题。这一类问题往往具有多个可行解，每一个解对应一个值。我们希望找到其中一个具有最优值（最大值或最小值）的解，称为最优解（可能存在多个最优解具有同一个最优值）。

## 动态规划算法的步骤

- 设计一个动态规划算法的步骤如下：
  - (1) 刻画最优解的结构特征
  - (2) 递归定义最优值
  - (3) 以自底向上的方式计算出最优值
  - (4) 根据计算过程的信息构造一个最优解
- 步骤(1) - (3)是动态规划方法求解最优值的基本步骤。如果需要求得最优解，则需要进一步记录计算过程的信息并执行步骤(4)。

## 动态规划算法求解例子

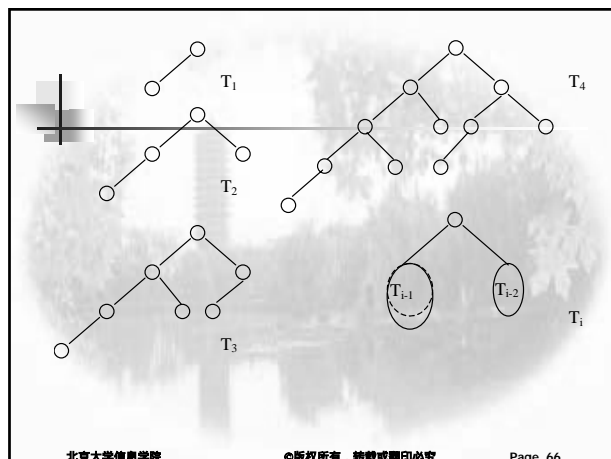
- 设计一个时间 $O(n^2)$ 的算法，找出由 $n$ 个数组成的序列的最长单调递增子序列。将这个 $O(n^2)$ 的算法的优化为 $O(n \log n)$ 的算法
- 设 $A$ 和 $B$ 是两个字符串。对字符串可以进行如下操作：
  - (1) 删除一个字符
  - (2) 插入一个字符
  - (3) 将一个字符替换为另一个字符
- 利用以上三种操作可以将字符串 $A$ 转换为字符串 $B$ 。我们称这种转换所需要的最少的字符串操作次数为字符串 $A$ 到 $B$ 的编辑距离 (Edit Distance)，记为。设计一个算法，对任给的两个字符串 $A$ 和 $B$ ，计算出它们的编辑距离

## 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 $O(n)$
- 怎样使得BST始终保持 $O(\log n)$ 级的平衡状态？
- Adelson-Velskii和Landis发明了AVL树
  - 一种平衡的二叉搜索树
  - 任何结点的左子树和右子树高度最多相差1

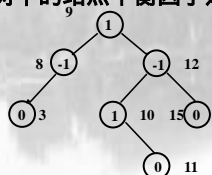
## AVL树的性质

- 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



## 平衡因子

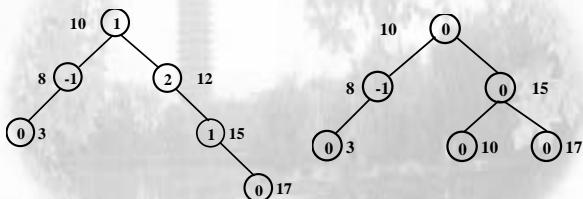
- 平衡因子，用 $bf(x)$ 来表示结点 $x$ 的平衡因子。它被定义为：  
 $bf(x) = x$ 的右子树的高度 -  $x$ 的左子树的高度
- 对于一个AVL树中的结点平衡因子之可能取值为0, 1和-1



## 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大致有三类
  - 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，而新结点又加到重的一边
    - 不平衡
    - “危急结点”

## 恢复平衡



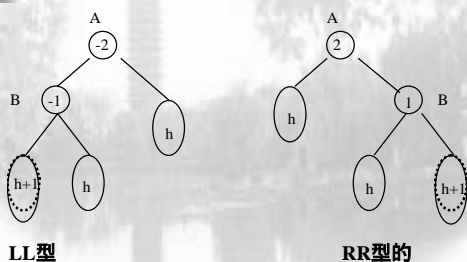
插入17后导致不平衡

重新调整为平衡结构

## 不平衡的情况

- 正常情况下，一个结点A的平衡因子只能是0, 1, -1，而在非正常情况有下面四种可能
  - LL型：导致不平衡的结点为A的左子树的左结点，这时A的平衡因子为-2
  - LR型：导致不平衡的结点为A的左子树的右结点，这时A的平衡因子为-2
  - RL型：导致不平衡的结点为A的右子树的左结点，这时A的平衡因子为2
  - RR型：导致不平衡的结点为A的右子树的右结点，这时A的平衡因子为2

## 不平衡的图示



LL型

RR型的

## 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

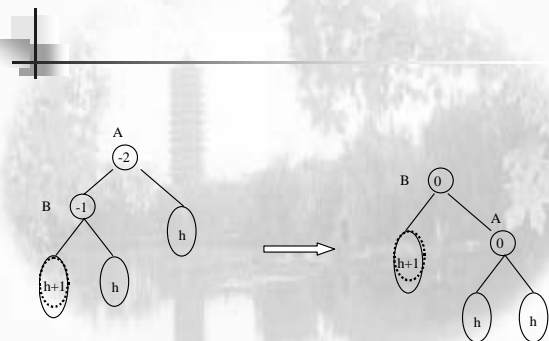
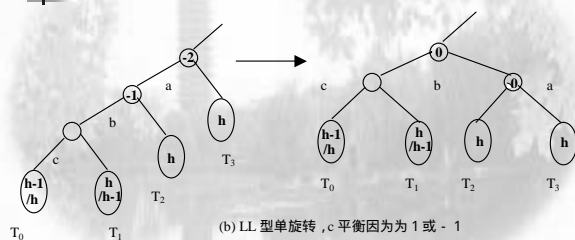
## 解决不平衡的方法——旋转

- 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 )

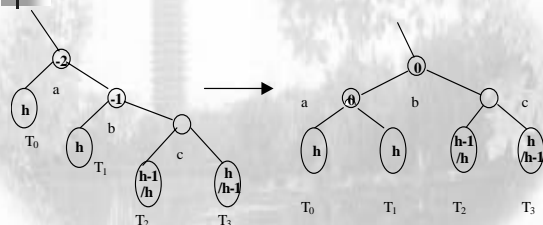
## 单旋转

- 如果加入一个新结点后需要对根为结点a的子树进行单旋转, 假设b为包含新加入结点的a的子女, c为包含新加入结点的b的子女
- 单旋转, 那么令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, 代替原b子树作为a的子树

### 单旋转示意图 (LL型)



### 单旋转示意图 (RR型)

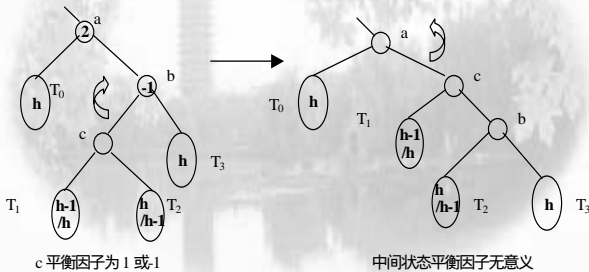


### RR型单旋转

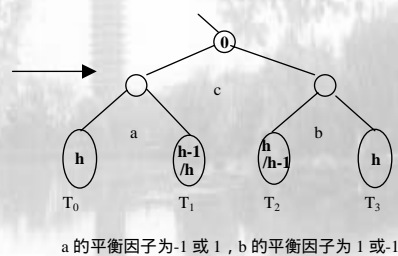
## 双旋转

- 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



## RL型双旋转第二步



## 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a的左子树 $T_0$ 
    - b的左子树 $T_1$
    - c的左子树 $T_2$ 和右子树 $T_3$ 
      - 加重的是c为根的子树，但是其结构其实没有变化
      - 因此，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
    - $T_0 a T_1 b T_2 c T_3$

## 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

```
template <class T> class avlNode//平衡二叉树结点类
{
public:
    avlNode(T val);//构造函数
    avlNode(T val,avlNode<T> *left,avlNode<T> *right,int bf);
    void release();//删除以当前结点为根的左右子树
    avlNode<T>* leftptr;//左右指针
    avlNode<T>* rightptr;
    int add(avlNode<T>* &p,T val);//插入一个值；返回新的avl树的根结点的指针
    void preorderview(avlNode<T>* current,int i=-1);//前序周游
    avlNode<T>* remove(T val,avlNode<T>* &waste,int &flag);// 根为当前结点的val
    结点
    avlNode<T>* findNodeValue(T val);//查找val结点
    T value;//码值
};
```

```
private:
    int bf;// 平衡因子
    avlNode<T>* removeLeftmostElement(avlNode<T>*
        &childptr,int &flag);//找最左的结点
    avlNode<T>* LL_singleRotation();//左子树LL失衡，返回调
        整后新树根
    avlNode<T>* RR_singleRotation();//右子树RR失衡，返回调
        整后新树根
    avlNode<T>* LR_doubleRotation();//左子树LR失衡，返回调
        整后新树根
    avlNode<T>* RL_doubleRotation();//右子树RL失衡，返回调
        整后新树根
};
```

```

template <class T> class avlTree//平衡二叉树类
{
public:
    avlTree();//构造函数
    avlTree(const avlTree<T> &source);
    ~avlTree();//析构函数
    void add(T value);
    void remove(T value);
    void deleteAllValue();
    void display();
    void display(avlNode<T>* found);
    avlNode<T>* findValue(T val);
private:
    avlNode<T> *root;
};

```

## 插入算法

```

template <class T> int avlNode<T>::add(avlNode<T>* &rp, T val)
//返回值表明以当前结点为根的树是否再插入之后增高
if (val<value)
    //左子树插入
    if (rp->leftptr==NULL)
        rp->leftptr=new avlNode<T>(val);
    else if (rp->leftptr->add(rp->leftptr, val)==0)
        return 0; //插入后子树没有增高
    if (rp->bf==1)
        //原来已经倾斜，左边失衡，需要做平衡处理
        if (rp->leftptr->bf<0) //插入在左侧，单旋转
            rp = LL_singleRotation();
        else rp = LR_doubleRotation(); //插入在右侧，双旋转
        return 0;
}

```

```

return --bf; // bf=(0, +1)的情况，不需要调整树，只要修改bf
}
else
{
    if (rp->rightptr==NULL)
        rp->right=new avlNode<T>(val);
    else if (rp->rightptr->add(rp->rightptr, val)==0)
        return 0; //插入后子树没有增高
    if (rp->bf==1)
        //原来已经倾斜，需要做平衡处理
        if (rp->rightptr->bf>0) //插入在右侧，单旋转
            rp = RR_singleRotation();
        else rp = RL_doubleRotation(); //插入点在右侧，双旋转
        return 0;
    }
    return ++bf; // bf=(0, -1)的情况，不需要调整树，只要修改bf
}
}

```

## 旋转的算法(LL)

```

template <class T> avlNode<T>* avlNode<T>::LL_singleRotation()
//以当前结点this(a)进行LL单旋转
avlNode<T>* b;
b=leftptr; //得到当前结点this(a)的左子树
leftptr=p->rightptr; //当前结点的左子树变为原子树的右子树
bf=0;
b->rightptr=this; //原子结点成为当前结点的父结点
if (b->bf==0) //如果是删除导致的旋转，则需要调整bf
    b->bf=1;
else b->bf=0;
return b; //返回新的子树根结点
}

```

## 旋转的算法(RR)

```

template <class T> avlNode<T>* avlNode<T>::RR_singleRotation()
//以当前结点this(a)进行RR单旋转
avlNode<T>* b;
b=rightptr; //得到当前结点右儿子
rightptr=b->leftptr; //当前结点的右儿子变为原其右儿子的左子女
bf=0;
b->leftptr=this; //原其右儿子变为当前结点的父亲
if (b->bf==0) //如果是删除导致的旋转，则需要调整bf
    b->bf=-1;
else b->bf=0;
return b;
}

```

## 旋转的算法(RL)

```

template <class T> avlNode<T>* avlNode<T>::RL_doubleRotation()
//以当前结点this(图11-24(d)中为a)进行LL单旋转
avlNode<T>* c,* b;
b=rightptr;
c=b->leftptr;
b->leftptr=c->rightptr; //第一次旋转c,b位置互换，c的右子树变为b的左子树
c->rightptr=b; b->bf=0;
if (c->bf==1) b->bf=-1; //因为c的右子树变为b的左子树
if (c->bf==1) bf=-1; //所以根据原来的高度可以知道现在的平衡因子
rightptr=c->leftptr; //第二次旋转c和当前结点a互换
c->leftptr=this; c->bf=0; //旋转完c平衡
return c;
}

```

## 旋转的算法(LR)

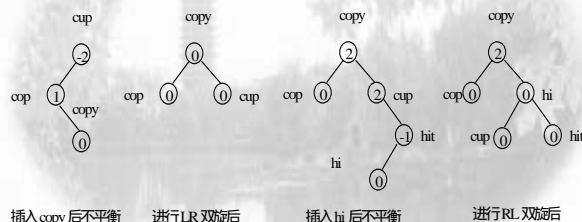
```
template <class T> avlNode<T>* avlNode<T>::LR_doubleRotation()
{//以当前结点this(a)进行LR双旋转
    avlNode<T> *c,*b;
    b=leftptr;
    c=b->rightptr;
    b->rightptr=c->leftptr; //第一次旋转b,c位置互换, c的左子树变为b
    的右子树
    c->leftptr=b;           bf=b->bf=0;
    if(c->bf==1) bf=1; //根据原来c的左子树的高度来判断当前
    if(c->bf==1) b->bf=-1; //b的平衡情况
    leftptr=c->rightptr; //第二次旋转c和当前结点互换
    c->rightptr=this;     c->bf=0;
    return c;
}
```

北京大学信息学院

©版权所有，转载或翻印必究

Page 91

## 插入单词：cup , cop , copy , hit , hi , his和 hia后得到的AVL树

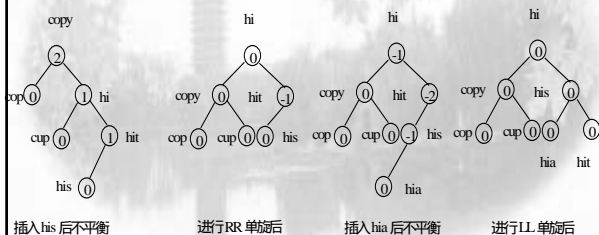


北京大学信息学院

©版权所有，转载或翻印必究

Page 92

## 插入单词：cup , cop , copy , hit , hi , his和 hia后得到的AVL树



北京大学信息学院

©版权所有，转载或翻印必究

Page 93

## AVL树结点的删除

- 删除是插入的逆操作。从删除结点的意义上来说，AVL树的删除操作与BST一样
- AVL树的删除是比较复杂过程，下面具体讨论一下删除的过程
- 由于情况较多，所以图示每种情况只列举了一种例子，其他都是类似的

北京大学信息学院

©版权所有，转载或翻印必究

Page 94

## AVL树结点的删除

具体删除过程请参考BST结点的删除

- 如果被删除结点a没有子结点
  - 直接删除a；
- 如果a有一个子结点
  - 用子结点的内容代替a的内容，然后删除子结点；
- 如果a有两个子结点
  - 那么则要找到a在中序周游下的前驱结点b（b的右子树必定为空）
  - 用b的内容代替a，并且删除结点b（如果b的左子树不空，则该左子树代替原来b的位置）。

北京大学信息学院

©版权所有，转载或翻印必究

Page 95

## AVL结点删除后的调整

- AVL树调整的需要
  - 删除结点后可能导致子树的高度以及平衡因子的变化
  - 沿着被删除结点到根结点的路径来调整这种变化
- 需要改动平衡因子
  - 则修改之；
- 如果发现不需要修改则不必继续向上回溯
  - 用一个布尔变量modified来记录之，其初值为TRUE
  - 当modified=FALSE时，回溯停止
- 有以下三种情况

北京大学信息学院

©版权所有，转载或翻印必究

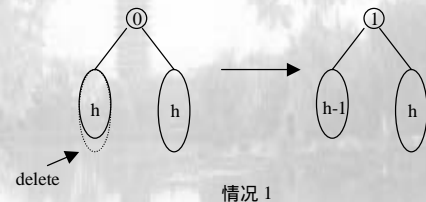
Page 96



## AVL树结点的删除过程2(续)

- 第一种情况当前结点a平衡因子为0
  - 如果它的左子树或者右子树被缩短，则它的平衡因子该为1或者-1
  - `modified=FALSE`
  - 因为这样的变化不会影响到上面的结点，调整可以结束

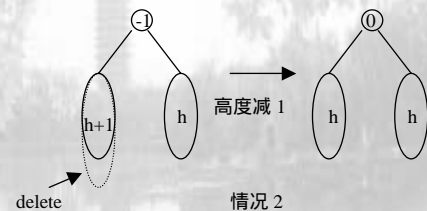
## AVL树结点的删除情况1图示



## AVL树结点的删除过程2(续)

- 第二种情况是当前结点a平衡因子不为0，但是其较高的子树被缩短
  - 则其平衡因子修改为0
  - `modified=TRUE`
  - 需要继续向上修改

## AVL树结点的删除情况2图示

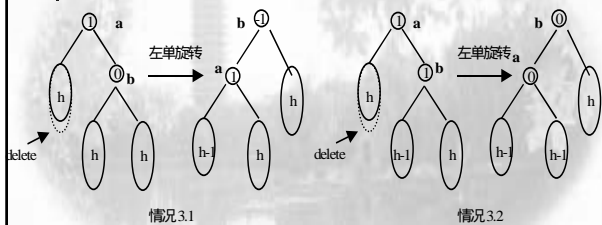


## AVL树结点的删除过程2(续)

- 第三种情况是当前结点a平衡因子不为0，且它的较低子树被缩短
- 结点a必然不再平衡，假设其较高子树的根结点为b，出现下面三种情况
  - 情况3.1：b的平衡因子为0
    - 单旋转
    - `modified=FALSE`
  - 情况3.2：b的平衡因子与a的平衡因子相同
    - 单旋转
    - 结点a、b平衡因子都变为0
    - `modified=TRUE`

- 情况3.3：b和a的平衡因子相反
  - 双旋转，先围绕b旋转，再围绕a旋转
  - 新的根结点平衡因子为0
  - 其他结点应做相应的处理
  - 并且`modified=TRUE`

## AVL树结点的删除情况3图示

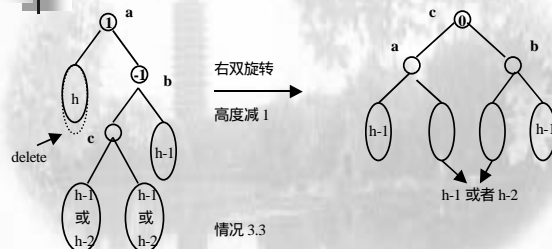


北京大学信息学院

©版权所有，转载或翻印必究

Page 103

## AVL树结点的删除情况3图示



北京大学信息学院

©版权所有，转载或翻印必究

Page 104

## 删除后的连续调整

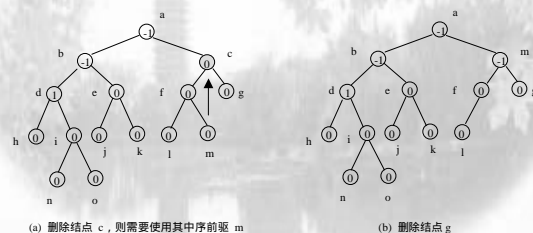
- 从被删除的结点向上查找其祖父结点
  - 然后开始单旋转或者双旋转操作
  - 旋转次数为 $O(\log n)$
- 连续调整
  - 调整可能会导致它祖先结点发生新的不平衡
  - 这样的调整操作要一直进行下去，可能传递到根结点为止

北京大学信息学院

©版权所有，转载或翻印必究

Page 105

## AVL树删除的例子

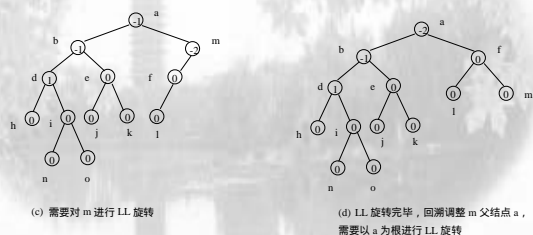


北京大学信息学院

©版权所有，转载或翻印必究

Page 106

## AVL树删除的例子

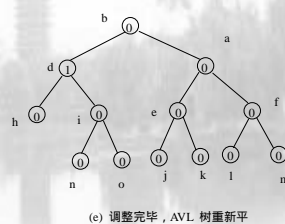


北京大学信息学院

©版权所有，转载或翻印必究

Page 107

## AVL树删除的例子



北京大学信息学院

©版权所有，转载或翻印必究

Page 108

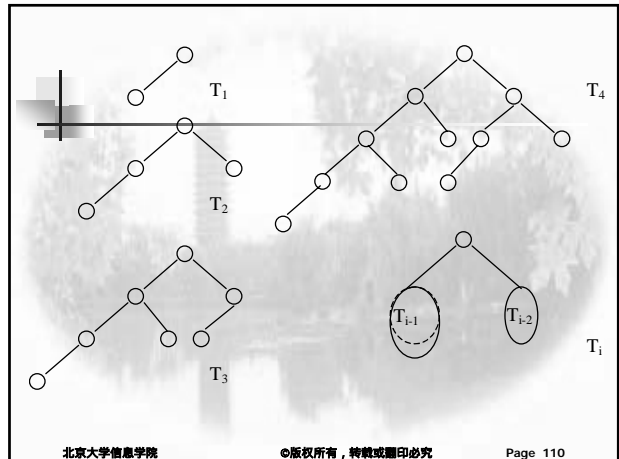
## AVL树的高度

- 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 $T_1, T_2, T_3, \dots$ 
    - 其中 $T_i$ 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, 每棵这样的树都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

北京大学信息学院

©版权所有, 转载或翻印必究

Page 109



北京大学信息学院

©版权所有, 转载或翻印必究

Page 110

## 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:
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(i) = F(i-1) + F(i-2)$

北京大学信息学院

©版权所有, 转载或翻印必究

Page 111

## 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

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

北京大学信息学院

©版权所有, 转载或翻印必究

Page 112

## 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  $\log X = \log_2 X / \log_2 2$  和  $\log_2 0.694$  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 $O(\log n)$

北京大学信息学院

©版权所有, 转载或翻印必究

Page 113

## AVL树的效率

- AVL树的检索、插入和删除效率都是 $O(\log_2 n)$ , 这是因为具有 $n$ 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。而对于较大的、存放在外存储器上的文件, 用二叉搜索树来组织索引就不合适了
- 在文件索引中大量使用每个结点包括多个关键字的B-树, 尤其是B+树

北京大学信息学院

©版权所有, 转载或翻印必究

Page 114

### 12.2.3 伸展树

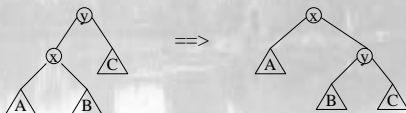
- 伸展树不是一个新数据结构，而只是改进BST性能的一组规则
  - 保证访问的总代价不高，达到最令人满意的性能
  - 不能保证最终树高平衡

### 展开( splaying )

- 每当访问一个结点x时(例如，当x被插入、删除或者是检索目标时)，伸展树就完成一次称为展开( splaying )的过程
  - 展开处理把结点x移到BST的根结点
  - 当删除结点x时，展开过程把结点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 )
  - 一次旋转通过调整结点x相对于其父结点和祖父结点的位置，把它移到树结构中的更高层

### 单旋转( single rotation )

- 当被访问结点是根结点的子女时，完成单旋转，它基本上在保持BST特性的情况下把结点x与它的父结点交换位置



### 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)
- 双旋转的结果是把结点x在树结构中向上移两层

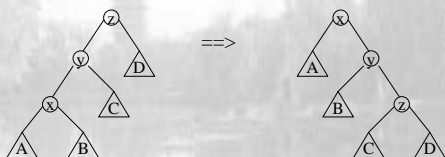
### 双旋转分为两类

- 一字形旋转( zigzag 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

- 当出现以下两种情况之一时，就会发生一字形旋转：
  1. 结点x是结点y的左子女，结点y是结点z的左子女。
  2. 结点x是结点y的右子女，结点y是结点z的右子女。

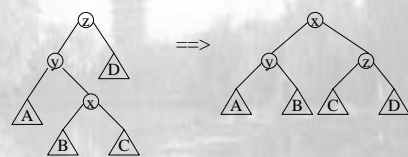
## 一字形旋转图示



## 之字形旋转

- 出现以下两种情况之一时，就会发生之字形旋转：
  1. 结点x是结点y的左子女，结点y是结点z的右子女。
  2. 结点x是结点y的右子女，结点y是结点z的左子女。

## 之字形旋转图示



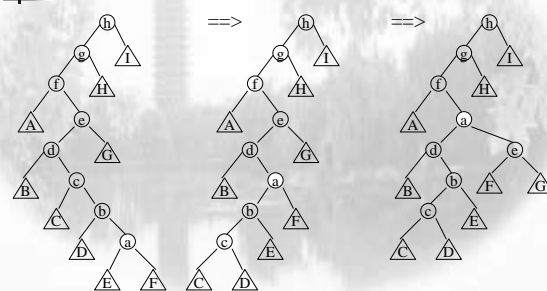
## 两种旋转的不同作用

- 之字形旋转
  - 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
  - 它只是把新访问的记录向根结点移动

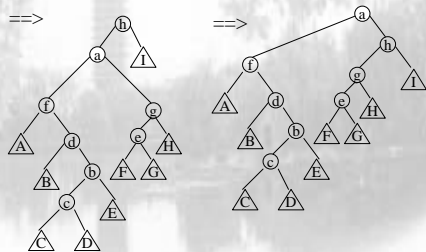
## 伸展树的调整过程

- 一系列双旋转
  - 直到结点x到达根结点或者根结点的子女。
- 如果结点x到达根结点的子女
  - 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
  - 都会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



## 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（包括）的频率相关
  - 能够进行更加动态的调整
- 而AVL树的结构与插入、删除或者是检索的频率无关
  - 只与插入、删除的顺序有关

## 伸展树的效率

- 对于一个包括 $n$ 个结点的伸展树，进行一组 $m$ 次操作（插入、删除、查找操作），当 $m \gg n$ 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 $O(\log n)$
- 证明伸展树确实能够保证 $O(m \log n)$ 时间超出了这本书的范围

## 12.3 空间数据结构

- 前面所讨论过的BST、AVL等搜索树都是针对一个一维的关键码进行的
- 多维数据不能简简单单的看作是多个一维数据的组合
  - 尽管做勉强也可以
- 所以我们下面介绍一些常用的多维数据结构

### 12.3.1 k-d树

- k-d树是早期发明的一种用于多维检索的树结构，它每一层都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

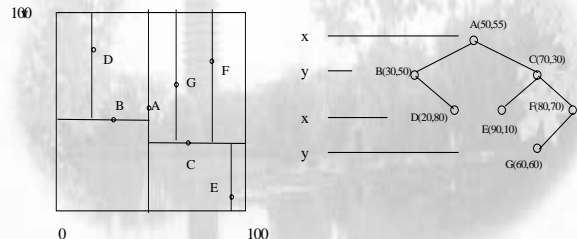
### 识别器( discriminator )

- 在每一层用来进行决策的关键码称为识别器( discriminator )
- 对于 $k$ 维关键码，在第 $i$ 层把识别器定义为 $i \bmod k$ 
  - 例如，对一个三维的关键码做检索，3个关键码 $(x, y, z)$ 标号分别为0、1、2
    - 第一层是 $0 \bmod 3 = 0$ ，所以使用关键码 $x$ ，
    - 第二层是 $1 \bmod 3 = 1$ ，所以使用关键码 $y$ .....

## 结点的分配

- 在结点分配的时候首先比较该层的识别器
  - 如果关键码小于识别器的值就放到左子树中
  - 否则放到右子树
- 然后在下一层使用新的识别器来判断每个结点的归属
- 识别器的值应该尽量使得被划分的结点大约一半落在左子树，另一半落在右子树

## K-D树示例



## K-D树的空间分解

- 上图是一个二维的k-d树，我们限制其取值范围为 $100 \times 100$ 之内
- k-d树的每个内部结点
  - 把当前的空间划分为两块
  - 交替地对两个维进行划分
- 根结点把空间划分成两部分
  - 其子结点进一步把空间划分成更小的部分
  - 子结点的划分线不会穿过根结点的划分线
- k-d树中的这些结点最终把空间分解为矩形
  - 这些矩形是结点可能落到的各子树范围

## K-D树的检索

- 交替地用识别器与各个维进行比较
- 不断地划分区间缩小范围
- 直到找到需要的点为止

- 例如，在上图检索点F(80, 70)
  - 在根结点A，先用80与A的X维比较
    - 80大于50，所以F应该在A的右子树，到达了结点C(70, 30)；
  - 比较F与C的Y维大小
    - 70大于30，所以F应该在C的右子树，然后到达结点F，即完成搜索
- 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树的搜索算法

```
KDNode * KDTree::KdFind(KDNode *n, Value val, int lev, int dimension)
{
    //n: 开始搜索的结点, val: 要搜索结点的多维值
    //lev: n处于的深度, dimension: k-d树的维数
    if (n == NULL) return NULL; //无法找到该结点
    else if (val == n->val) return n; //n既是搜索的结点
    else{
        int k = lev mod dimension; //求得识别器的编号
        if (ValofD(n->value, k) > ValofD(val, k))
            return KdFind(n->left, val, lev+1, dimension); //小于识别器，在左子树
        else
            return KdFind(n->right, val, lev+1, dimension); //大于识别器，在右子树
    }
}
```

## K-D树的删除

- 如果删除一个没有子结点的结点，不用做任何的调整
- 如果删除了一个有子结点的结点
  - 如果只有一个子结点并且子结点没有自己的子结点，那么就可以用它来取代被删除结点的位置
  - 而如果子结点还有自己的子结点，那么就需要进行选择

## K-D树的插入

- 首先也要使用上面的检索算法
  - 如果返回一个非NULL值，那么说明该结点已经在树中；
  - 否则，在NULL指针所在位置新建一个结点，存储插入的值。

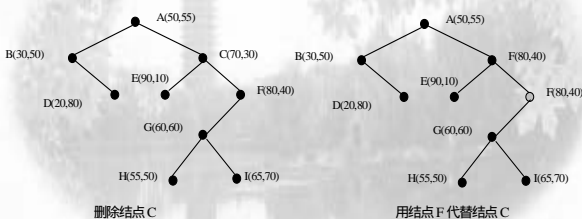
## K-D树的删除（结点的选择）

- 删除一个结点后需要从它的左右子树中选择一个合适的结点来替代它
  - 该结点最好能够保持原来的空间划分
    - 最好的选择是左子树中当前维中值最大的一个结点
    - 或者右子树中当前维值最小的一个结点

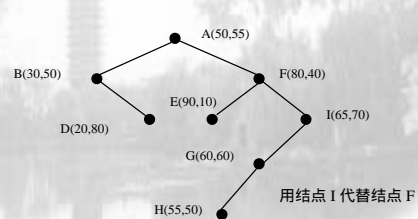
## K-D树的删除（重复调整）

- 删除过程也是一个循环的过程，因为选择一个结点代替当前删除结点将意味着那个结点在它原来的位置被删除，所以它的子结点也要进行相关的动作，直到没有结点移动为止

## K-D树的删除图示



## K-D树的删除图示





## 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

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

- 例如，在二维的情况下，可以查找X在一定范围的结点
  - 如果到达某个结点发现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了
    - 该结点的左子树都不用搜索了

## K-D树的范围查找

在上图中查找与 (50, 70) 距离小于20的点

- 根结点A (50, 55) 显然满足要求
- 进入左子树，发现点B (30, 50) 不在范围里面
  - 但是这并不是说左子树应该被抛弃
  - 70-50=20，所以结点B (30, 50) 的左子树应该抛弃，因为左子树的y<50。
  - 进入其右子树发现结点D (20, 80) 也不符合。
- 同样的道理，算法将进入根结点的右子树进行搜索

## K-D树的范围查找

查找与 (50, 70) 距离小于20的点 (续)

- 根的右结点C (70, 30) 显然不在其中
  - 纵坐标70-30>20
  - 不需要进入C的左子树，因为左子树的y<30
- 进入其右子树F，发现结点F (80, 70) 也不符合
  - 横坐标80-50>20，不需要进入F的右子树，因为右子树的x>80
  - 进入F的左子树，子结点G (60, 60) 复合要求

## 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

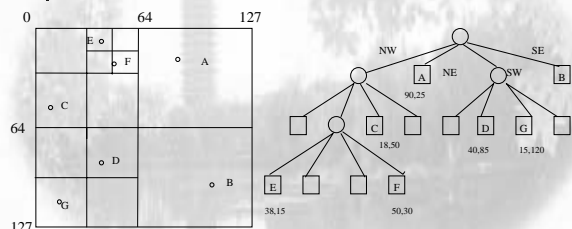
## 12.3.2 PR四分树

- 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
  - 每个内部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
- 完全四叉树

## PR四分树对空间的划分

- 每个内部结点将当前空间均等地划分为四个区域
  - 如果子区域包含的数据点数大于1，那么就  
把该区域继续均等地划分为四个区域
  - 依次类推，直到每个区域所包含的数据点不  
超过一个为止

## PR树的图示



## PR树的划分

- 上图所表示的PR四分树，其对象空间为  
128×128，并且其中包含点A、B、C、  
D、E、F和G
- 根结点的四个子结点把整个空间平分为  
四份大小为64×64的子空间
  - NW, NE, SW, SE
  - NW (包含三个数据点) 和SE(包含两个数  
据点) 需要进一步分裂

## PR树的检索

检索的过程与划分过程类似

- 例如，检索数据点D，其坐标为(40, 85)。
  - 在根结点，它属于SW子结点(范围为(0-64, 64-127))
  - 进入SW子树，其四个子树的范围是(0-32, 64-96)、(32-64, 64-96)、(0-32, 96-127)和(32-64, 96-127)，D应该位于下一个NE子树中
  - NE子树只有一个叶结点代表数据点D，所以返回D的值。
- 如果已经到达叶结点却没有找到该数据点，那么返回错误

## PR树的插入

首先通过检索确定其位置

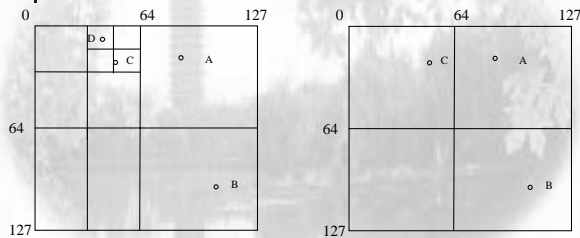
- 如果这个位置的叶结点没有包含其他的数据点
  - 那么我们就把记录插入这里；
- 如果这个叶结点中已经包含P了(或者一个具有P的坐标的记录)
  - 那么就报告记录重复；
- 如果叶结点已经包含另一条记录X
  - 那么就必须继续分解这个结点，直到已存在的记录X和P分别进入不同的结点为止

## PR树的删除

删除纪录

- 首先检索到P所在的结点N
  - 将结点N所包含的记录改为空
- 调整
  - 查看它周围的三个兄弟结点的子树
  - 如果只有一个兄弟记录(不可能少于一个记录，否则就没有必要分裂为四个子结点)
    - 把这四个结点的子树合并为一个结点N'，代替它们的父结点
  - 这种合并会产生连锁反应
    - 在上一层也可能需要相似的合并
    - 直到到达某一层，该层至少有两个记录分别包含在结点N'以及N'的某个兄弟结点子树中，合并结束

## PR树的删除产生的合并



## PR树的区域查询

检索以某个点为中心、半径为 $r$ 的范围内所有的点

- 根结点为空
  - 查询失败
- 如果根结点是包含一个数据记录的叶结点
  - 检查这个数据点的位置，确定它是否在范围内

- 如果根结点是一个内部结点
  - 在根结点确定包含该范围的子树
  - 然后进入符合这种条件的子树
  - 递归地继续这样的过程
    - 直到找不到这样的子树或者到达叶结点为止
    - 当到达叶结点时
      - 如果不包含任何的记录
        - 则直接返回；
      - 如果包含一个数据记录
        - 则检查这个数据记录是否在范围之内，如果在就返回该数据

## 12.3.3 R\*树

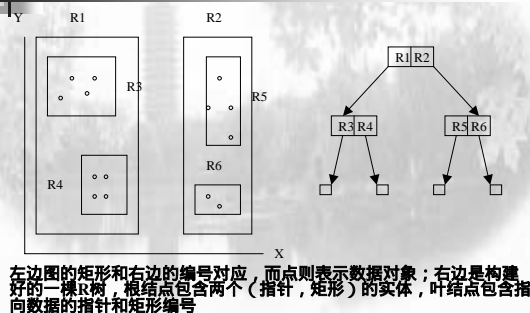
- R\*树是一种用于处理多维数据的数据结构
  - 用来访问二维或者更高维区域对象组成的空间数据
  - 1990年由N. Bechmann, H. Kriegel, R. Schneider 和 B. Seeger提出
- R\*树是对R树的结构改进

## R树

- R树的结构类似于B+树
  - 它可以用来存储多维的矩形
- 每个非叶结点都存储着(cp, rectangle)这种形式的实体
  - cp表示指向子结点的指针
  - rectangle表示这个结点所代表的矩形
    - 该矩形是包括所有子结点的最小矩形
- 其实R树可以用来存储多维数据，矩形只是二维的情况

- 每个叶结点一般存储形式为 (Oid, rectangle) 的实体
  - Oid一般表示一个存储在数据库里面的空间对象
  - 而rectangle表示能够包含这个空间对象的最小矩形

## R树的图示



北京大学信息学院

©版权所有，转载或翻印必究

Page 163

## 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： $2 \leq m \leq M/2$ 
  - 1. 根结点如果不是叶结点，那么应该至少有2个子结点；
  - 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

北京大学信息学院

©版权所有，转载或翻印必究

Page 164

## 矩形重叠覆盖

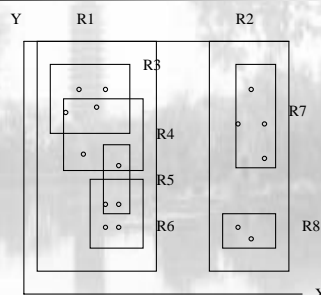
- R树的主要问题就是矩形重叠覆盖
  - R树要支持数据库中的各种复杂查询
  - 如果一个查询所需要的结果都在一个矩形区域里面那么速度将是最快的
- 很难找到一种合适的方法来确定单个矩形的大小
  - 各个矩形之间复杂的关系
  - 调整一个矩形的大小必然会影响到很多与它相关的其他矩形

北京大学信息学院

©版权所有，转载或翻印必究

Page 165

## R树中的矩阵重叠



北京大学信息学院

©版权所有，转载或翻印必究

Page 166

## 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

北京大学信息学院

©版权所有，转载或翻印必究

Page 167

## R树的动态插入算法

- 基本的思想
  - 先选择一个合适的结点
  - 如果该结点还有空位置就把数据的实体插入其中
  - 否则要分裂这个结点，将所有实体重新分配给分裂后的两个新结点
- 选择算法和分裂算法是整个插入的关键

北京大学信息学院

©版权所有，转载或翻印必究

Page 168

## R树的选择算法

### Choose Subtree:

1. 将根结点赋予N；
2. if N 是叶结点  
    返回 N；  
    else  
    (1) 选择N中子结点所代表矩形中只需要最小的扩充就可以包含插入数据的；  
    (2) 对选中子结点递归调用Choose Subtree；  
    返回第2步得到的结点。

## R树的分裂算法

### Split Node

//将包含M+1个实体的结点分裂为2个结点，并且重新分配实体

1. 调用Pick Seed方法选择每个新结点的第一个实体；
2. Repeat  
    调用Distribute Entry来给2个结点分配实体；  
    Until 实体已经被全部分配或者一个实体包含结点M-m
3. 如果还有实体剩余，全部分配给另一个结点。

## R树的分裂算法

- 对于一个已经满了的结点，即具有M个实体的结点
  - Split算法将原来M个实体和1新个实体分裂成m和M-m+1两组
  - 分配给两个新的结点

### ■ Pick Seed方法

- 在全部M+1个实体中，找到距离最远的两个实体作为两个组的第一个实体
- 这样分裂出来的两个结点之间的重叠覆盖将是最小

## R树的分裂算法(PickSeed)

### Pick Seed

1. 对于每一对实体E1, E2, 构建一个包含E1的矩形和E2的矩形的最小矩形R；
2.  $D = \text{面积}(R) - \text{面积}(E1\text{的矩形}) - \text{面积}(E2\text{矩形})$ ；  
    返回D值最小的实体对。

## R树的分裂算法(Distribute Entry )

### Distribute Entry

1. 调用Pick Next方法选择下一个需要分配的实体；
2. 计算为了包括该实体，每组实体需要扩大的面积，选择扩大较少的那个。如果两组所需要面积一样，那么依次按照优先级选择面积较小的一组，实体少的一组，两个组都加入。

## R树的分裂算法 (Pick Next)

### Pick Next

1. 对于每一个还没有被分组的实体，计算为了包括该实体，每组实体需要扩大的面积d1和d2。
  2. 选择|d1-d2|最大的实体
- Pick Next方法每次选择一个最容易区分属于那组的实体，这样的选择使得分组比较合理，也避免了组内实体互相重叠。

## R树的缺陷

R树的分裂策略会导致很坏的分裂出现

- 考虑一种比较极端的情况，如果有一个实体和另外n-1个距离很远，而那n-1个实体之间距离很近
  - 那么根据算法，我们选择这个实体与那n-1个实体中的一个作为Seed生成两个组G1和G2
  - 然后向这两个组加入实体，每次都是加入到G2中，直到G2中的实体到达M-m+1
  - 剩下的实体必然都加入到G1中，导致G1的面积迅速扩大
  - G1和G2之间的交迭非常严重。这样分裂出来的结构使得查询效率很低

## R树的改进——R\*树

- 在R树中的分裂主要考虑的是矩形面积的因素
- R\*树的插入算法，综合考虑
  - 矩形面积
  - 空白区域
  - 重叠

## 重叠 (overlap)

- 一个实体的重叠是其矩形与同结点其他实体矩形的面积之交的和：

$$overlap(E_k) = \sum_{i=1, i \neq k}^p area(E_k \text{ 'rect} \cap E_i \text{ 'rect})$$

p是该结点的实体数， $1 \leq k \leq p$

## R\*树的选择算法

- 与R树的选择算法相比，R\*算法主要增加了一个减少叶结点覆盖的步骤
  - 增加检索的效率
  - 但试验证明提高的并不是很明显

## R\*树的选择算法

Choose Subtree

1. 将根结点赋给N；
2. if N是叶结点  
Return N；  
if N的子结点指针指向叶结点（即N的子结点存储的实体是包含对象的矩形）
  - (1) 计算N中实体为包括新数据而将增加的重叠；
  - (2) 选择重叠增加最小的实体；

## R\*树的选择算法

- (3) 在最小重叠的实体中选择面积增加最小的实体；
  - if N的子结点并不是指向叶结点的
    - (1) 计算N中实体为包括新数据而将增加的面积
    - (2) 选择面积最小的实体
- 把选择的实体所指向的子结点赋予N，循环执行2

## R\*树的分裂策略

- 首先把待分裂结点的实体在各个维按照从小到大的顺序排列，再按照从大到小的顺序排列。
- 对每个排列进行下面的分裂：把所有M+1个实体在k处 ( $1 \leq k \leq M - 2m + 2$ ) 分成两组
  - 一组为 (m-1) + k个
  - 其他的放入另外一组
  - 然后给每种分裂策略评分

## R\*树的分裂评分

1. Area-value: 第一组所有的实体矩形面积和 - 第二组所有的实体矩形面积和；
2. Margin-value: 第一组所有的实体矩形边长和 - 第二组所有的实体矩形边长和；
3. Overlap-value: 第一组所有的实体矩形面积与第二组所有的实体矩形面积的交

- 这些方法也可以综合使用
  - 可以选择某个维某种排列方式中评分最低的
  - 也可以计算出来所有维上所有排列方式的评分最低的
  - 也可以 选择总和中评分最低的

## R\*的分裂算法

### Split Node

1. 调用ChooseSplitAxis来决定在哪一个维上进行分裂
2. 调用ChooseSplitIndex来选择最好的分裂方式 (找到合适的K)
3. 将实体分配到两个组中

## R\*的分裂算法 (ChooseSplitAxis)

- 对于每一维
  - 计算按照升序排列的Margin-value
  - 计算按照降序排列的Margin-value
- 选择所有维中Margin-value较小的作为分裂用的维

## R\*分裂算法 (ChooseSplitIndex)

- 在分裂的维上
- for( $k=1; k \leq M-2m+2; k++$ )
  - 计算分裂的Overlap-value的值，选择最小的Overlap-value的k值
  - 如果有相同的值出现，那么计算Area-value，选择产生最小值的k
- 实验证明当m的大小是M的2/5时，上面的算法得到最好的效率

## 删除和重插入

- R树和R\*树都是动态生成的，它们的树结构与插入数据的顺序相关
  - 在R树中可能存在不合适的结点，它们被过早地插入而影响了现在的效率
  - 每次的分裂也都是在当前状态下做出的，也许这样的分裂并不适合后来的情况
- 删除当前不合适的结点并且进行重新插入
  - 实验证明这种方法可以大大提高效率

## R\*树的 完整插入算法

### 插入算法 Insert

1. 调用Choose Subtree选择插入实体E的合适位置：结点N；
2. 如果N的实体数目小于M，插入E；  
如果N的实体数目等于M，调用OverflowTreatment(L)处理重新插入的情况  
其中L是N所在的层数；

## R\*树的 完整插入算法

3. 如果OverflowTreatment被调用，并且分裂已经进行，那么如果上一层也出现实体数目等于M的情况，传递OverflowTreatment；  
如果分裂是针对根结点的，那么需要新建一个根结点；
4. 调整在插入结点到根结点路径上所有的实体矩形，让它们是包含子结点矩形的最小矩形

## 溢出处理：OverflowTreatment

if 非根结点 and 这是在给定层次的插入过程中的第一次调用

调用reinsert方法；

else 调用split方法

## 重插入算法：Reinsert

- 计算结点N的M+1个实体矩形的中心和代表N的矩形（包含M个实体的矩形）的中心之间的距离
  - 按照距离的降序给实体排序
  - 删除前p个实体，调整N的矩形大小
  - 调用insert重新插入被删除的p个实体，插入可以按照递增或者递减的顺序
- 注释：p是一个常数，试验证明p取M的3/10最好



## 插入时的重构

- 每一层的第一次overflow的处理将导致p个实体被重新分配
- 它们如果再次被分配到同一个实体可能导致这个实体的分裂
- 它们也可能被分配到不同的实体，这也许还会导致分裂，但是大多数情况下不会出现分裂

## R\*的使用

- N. Bechmann, H. Kriegel, R. Schneider 和 B. Seeger对R\*树做了很多的试验
  - R\*树代价仅仅稍微高于R树
  - R\*树可以同时支持点和其他空间数据
- 所以R\*树应用非常广泛

## 树形结构的应用

- 树形结构是一种应用非常广泛的结构。除利用树形结构建立索引外（例如BST树、B+树），在许多算法中常常利用树形结构作为中间结构来表达问题空间，以求解问题、确定对策等。
- 这里将介绍树形结构的两个有趣应用——决策树和博弈树

## 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：

- 如果一个顾客年龄不超过25岁，并且在过去3年中出过两次或两次以上交通事故，则不接受他的申请。
- 如果一个顾客已婚或年龄超过25岁，并且在过去3年中出过两次以下交通事故，则向他收基本保险费。
- 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。

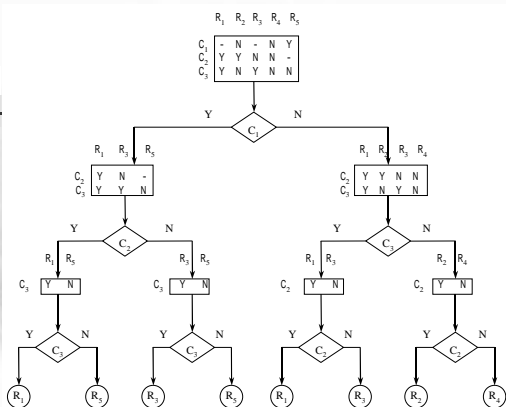
## 决策表

		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 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		

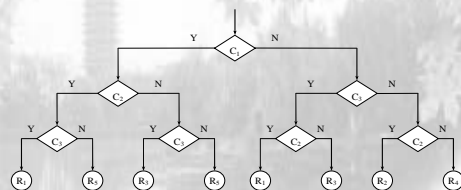
## 决策表转换成二叉树形式决策树

- 二叉决策树中的分支结点代表条件测试，测试结果确定下面的测试在哪棵子树上进行，二叉决策树的树叶代表选定的规则。
- 转换的方法是反复地把原来的问题分解为两个较小的子问题。令C是决策表中几个条件项构成的矩阵，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)

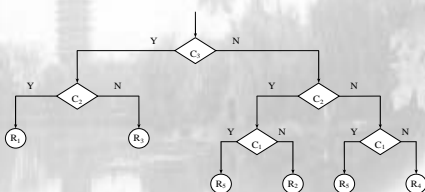
- 然后用条件测试 $C_i$ 作为根结点，并把 $C_Y$ 和 $C_N$ 作为根 $C_i$ 的两个子结点。
- 继续构造 $C_Y$ 和 $C_N$ 的决策子树，如此进行下去，直至 $C_Y$ 或 $C_N$ 中只有一行时就可作出最后的决定，用树叶代表规则



## 由上面的决策表最后得到的决策树



## 去掉多余的比较步骤



## 根据决策树写出解决决策问题的算法

\*\*\*\*\* 算法12-6 保险公司的决策算法 \*\*\*\*\*

struct { //顾客信息记录类型定义

```
char* name; //顾客姓名
bool married; //是否已婚
int age; //年龄
int accident_count; //交通事故次数
} CustomerRecord;
```

void Judge(CustomerRecord customer) //决策过程算法

```
{
    cout<<customer.name<<" "; //输出顾客姓名
    if (customer.accident_count>=2) //先判断C3
    {
```

```

    if (customer.age <= 25)                // C2
        cout << "R1";
    else cout << "R3";
}
else {
    if (customer.age <= 25)                // C2
    {
        if (customer.married)            // C1
            cout << "R5";
        else cout << "R2";
    }
}

```

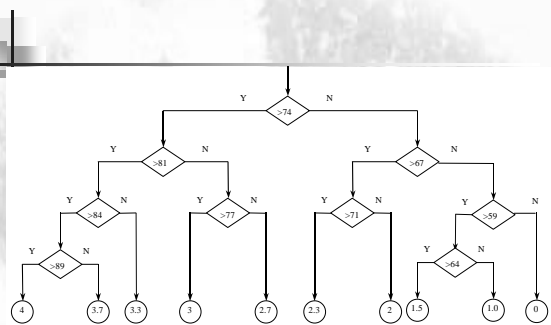
```

else {
    if (customer.married)                // C1
        cout << "R5";
    else cout << "R4";
}
}

```

## 决策树的另一个应用例子

- 在计算学生成绩点（简称为“GPA”）
- 北大的GPA转换算法为例：90-100分=4.0；85-89分=3.7；82-84分=3.3；78-81分=3.0；75-77分=2.7；72-74分=2.3；68-71分=2.0；64-67分=1.5；60-63分=1.0；60以下=0
- 首先将74作为一个分界点，判断是否“>75”，如果是，则以81为分界点，否则以67为分界点，类似地进行下去，直到得出最后的四分制GPA成绩



## 决策树应用于数据挖掘（Data Mining）

- 这里主要介绍在数据挖掘中如何基于决策树来进行分类
- 分类要解决的问题是为一个事件或对象归类。一般是在已有数据的基础上，用机器学习的方法训练出个分类函数或构造出一个分类模型，即我们通常所说的分类器(Classifier)。
- 该函数或模型能够把数据库中的数据纪录映射到给定类别中的某一个，从而可以应用于数据预测。

- 要构造分类器，需要有一个训练样本数据集作为输入。
- 训练集(Training set) 由一组数据库记录构成，每个记录是一个由有关字段值组成的特征向量，我们把这些字段称做属性(Attribute)，把用于分类的属性叫做标签(Label)，标签属性也就是训练集类别标记。

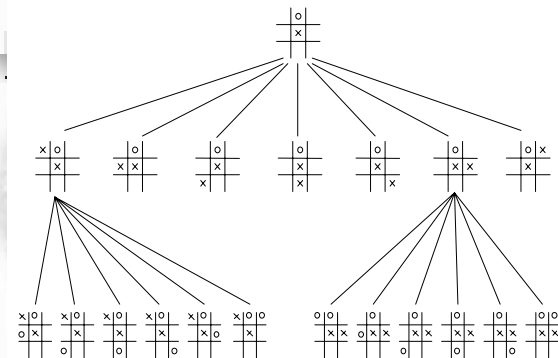
- 一个具体的样本的形式可以表示为  $(v_1, v_2, \dots, v_n; c)$ ，其中  $v_i$  表示字段值， $c$  表示类别。训练集是构造分类器的基础。标签属性的类型必须是离散的，且标签属性的可能值的数目越少越好（最好是两三个值）。标签值的数目越少，构造出来的分类器的错误率越低。


## 使用决策树分类的过程

- 将连续型属性值“离散化”
- 利用几个变量（每个变量对应一个问题）来判断所属的类别，最后每个叶子都对应一个类别。
- 在每个结点都会遇到一个问题，对每个结点上问题的不同回答导致不同的分支，最后会到达一个叶子结点。

## 12.4.2 博弈树

- 博弈树（game tree）最初主要应用在人对弈的棋类程序中
- 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。判定胜负的方法是：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。



- 上图描述了初始状态  作为树根，这时轮到甲放置棋子。甲放置完棋子后的所有可能的棋盘状态，都作为根的子结点出现在树的第一层。
- 第二层的结点是轮到乙放置棋子时的棋盘状态，乙从第一层的某个结点出发，选择某个方格放置棋子后的所有可能棋盘状态都作为该结点的子结点出现在第二层(由于版面限制，第二层的结点未全部画出)

- 上图就称为“博弈树”
- 这棵树的第二层并不是游戏终止时的棋盘状态。我们还可以画出第三层，第四层...的结点。但一般情况下，由于计算机存储器大小和运算速度的限制，不能把博弈树一直构造到游戏终止时的棋盘状态，而只能构造到一定深度。
- 上图是一棵深度为2的tic-tac-toe游戏博弈树

## 根据博弈树找到最佳行为

- 假设计算机作为游戏者甲，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，计算机面对上图博弈树的树根所代表的棋盘状态，有7种可能的下法，它当然愿意选择对它最有利，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 $E(x)$ ，此函数以棋盘状态 $x$ 为自变量，给出一个函数值，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。

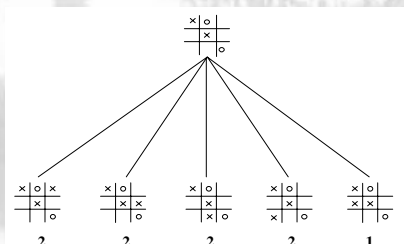
## 估值函数

- 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：

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

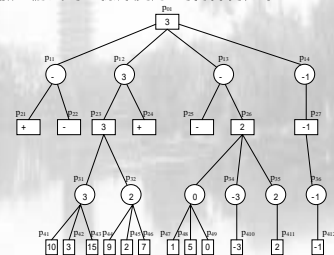
- 其中RCDC表示计算机还有可能占据的行、列、对角线数之和，RCDO表示对方还有可能占据的行、列、对角线之和。

## 带有估值的博弈树



## 博弈树的一般形式

- 在一个假想的二人游戏上（不局限于tic-tac-toe游戏）进行。假设该假想游戏的一棵深度为4的博弈树如下：



- 其中方形结点表示计算机面对的棋盘状态；圆形结点表示对方面对的棋盘状态；树叶结点中标出的是结点的 $E(x)$ 值
- 在此博弈树中，棋盘状态 $x$ 对计算机有利的程度 $V(x)$ 这样估算：设分支结点 $x$ 在博弈树中有 $d$ 个子结点 $c_1, c_2, \dots, c_d$ ，则 $V(x)$ 定义为

$$V(x) = \begin{cases} E(x), & \text{当 } x \text{ 是树叶} \\ \max_{1 \leq i \leq d} \{V(c_i)\}, & \text{当 } x \text{ 是方形的分支结点} \\ \min_{1 \leq i \leq d} \{V(c_i)\}, & \text{当 } x \text{ 是圆形的分支结点} \end{cases} \quad (\text{公式2.10})$$

- 定义 $V(x)$ 的理由很简单

- 如果游戏进行到方形结点代表的棋盘状态，则轮到计算机选择下一步的走法，因为计算机想获胜，它当然从所有可能的下一步状态中选择对于它最有利的状态，也就是 $V(c_i)$ 最大的状态
- 如果游戏进行到圆形结点代表的状态，则轮到对方选择下一步的走法，因为对方想获胜，它当然从所有可能的下一个状态中选择对计算机最不利的状态，也就是 $V(c_i)$ 最小的状态。
- 上图分支结点中给出的值就是用最大最小过程确定的

## V(x)的计算

- 下棋程序的关键部分是计算 $V(x)$ 值为了写出计算 $V(x)$ 的简单的递归程序，我们将 $V(x)$ 的定义改写成如下形式

$$V'(x) = \begin{cases} E(x), & \text{当 } x \text{ 是方形树叶} \\ -E(x), & \text{当 } x \text{ 是圆形树叶} \\ \max_{1 \leq i \leq d} \{-V'(c_i)\}, & \text{当 } x \text{ 是分支结点} \end{cases} \quad (\text{公式12.11})$$

- 与公式(12.10)对照可知：对于方形结点

$$V'(x) = V(x)$$

对于圆形结点

$$V'(x) = -V(x)$$

## 博弈树算法

- 计算 $V(x)$ 值和确定下一步走法的算法 `int ValueOf(node * x, int player, int depth, node * best)`，其中`node`是代表棋盘状态的博弈树结点的类型，例如，tic-tac-toe游戏的棋盘状态可以用包括9个元素的数组来表示，每个元素对应一个方格，可取值1、-1、0，表示对应的方格由计算机占据、对方占据、或尚未被占。

- 算法的输入参数`x`是当前出发点的棋盘状态；`player`指出该谁走下一步棋；`depth`指出希望产生深度为多少的博弈树。函数返回 $V(x)$ 的值，`best`指针指向应该选择的下一个棋盘状态。

- 由于算法太复杂这里就不列出请参考教科书

## $\alpha - \beta$ 剪裁

- 在上述算法中，实际上用的是穷举法的思想。由于是 $3 \times 3$ 的棋盘，因此每次放棋子之前，平均有5个空格位置可以下棋，如果博弈树深度为4，则需要搜索625（5的4次方）个位置。可以看出，需要搜索的位置数是呈指数级增长的。这将导致构造整棵博弈树的计算量大得令计算机无法承受。
- 事实上并不需要搜索所有的位置才能找到最佳下棋位置采用 $\alpha - \beta$ 剪裁的技术，从而大大减少计算量

## 剪裁 (prune)

- 一个最大层结点的  $\alpha$  值定义为该结点的最小可能值，如果已经确定一个最小层结点的  $\alpha$  值小于或等于其父母结点的  $\beta$  值，则可以停止产生这个最小层结点的其它子结点。按此规则终止结点的产生称作 剪裁 (prune)

## 剪裁 (prune)

- 一个最小层结点的  $\beta$  值定义为该结点的最大可能值，如果已经确定一个最大层结点的  $\beta$  值大于或等于其父母结点的  $\alpha$  值，则可以停止产生这个最大层结点的其余子结点。
- 剪裁和 剪裁的规则结合起称作  $\alpha - \beta$  剪裁。

- 把  $\alpha - \beta$  剪裁用于前面的假想图上，事实上整个以 $p_{26}$ 为根的子树都不必产生。这是因为当 $V(p_{12})$ 确定为3时， $p_{01}$ 的  $\alpha$  值成为3， $V(p_{25})$ 小于 $p_{01}$ 的  $\beta$  值，确定 $p_{13}$ 的值小于 $p_{01}$ 的  $\beta$  值，于是进行了 剪裁。
- 需要强调的是：结点的  $\alpha$  值和  $\beta$  值是动态的，它们依赖于当前哪些结点的值已被计算出来了。
- 要用  $\alpha - \beta$  剪裁的技术来改进算法`ValueOf`，重新叙述  $\alpha - \beta$  剪裁的规则如下：一个结点的 $B$ 值是该结点的最小可能值。对于任何结点 $x$ ，令 $B$ 是其父母结点的 $B$ 值，并且令 $d = -B$ ，则当 $x$ 的值已经确定了大于或等于 $d$ 时；就可以停止产生结点 $x$ 的其余子结点。

\*\*\*\*\*算法12-8 应用 - 剪裁技术的博弈树结点求值\*\*\*\*\*

```
const int MAX = 10000;      //代表+
const int MIN = -10000;     //代表-
const int SIZE = 3;
```

```
struct node{
    int grid[SIZE][SIZE];
};
int Valueofb(node * x, int player, int depth, int d, node * best)
{
    int value, Evalue;
    int count;
```

```
node * p;
node * bes;
Evalue=E(x);
if ((Evalue==MAX) || (Evalue==MIN) || (depth==0)) // x是代表
终局状态的结点或1=0
{
    for (int i=0;i<Size;i++) //best为空棋盘状态
        for (int j=0;j<Size;j++)
            best->grid[i][j]=0;
    if (player == 1)
        return Evalue; //返回E(x)
    else return (-Evalue); //返回-E(x)
}
```

```
p = generate(x,player,count); //产生下一步所有可能状态
value=MIN; //准备执行最大最小过程
best=p;
for (i=0;(i<count)&&(value<d);i++)
{
    val = Valueofb(p,-player,depth-1,-value,bes);
    if (value < -val)
    {
        value=-val;
        best=p;
    }
    p++;
}
```

## 学期总结

- 大家辛苦了！
  - 天道酬勤
- 纪律问题
  - 行政？旷课1/3以上取消资格
- 情商
  - 参与、合作、沟通
  - 纪律
- Thanks and Good Luck!