数据结构与算法实习

——空间数据结构

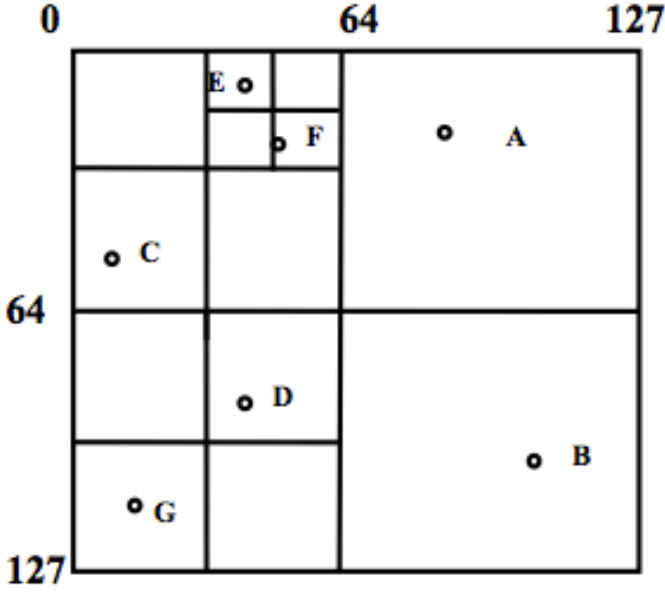
张路

北京大学信息科学技术学院

张铭 赵海燕 王腾蛟 宋国杰，《数据结构与算法实验教程》（国家十一五规划教材），高教社2011年1月

问题背景

- 数据库中，每个实体有许多属性
- 地理信息的范围查询
  - 经度、纬度
  - 经营类型



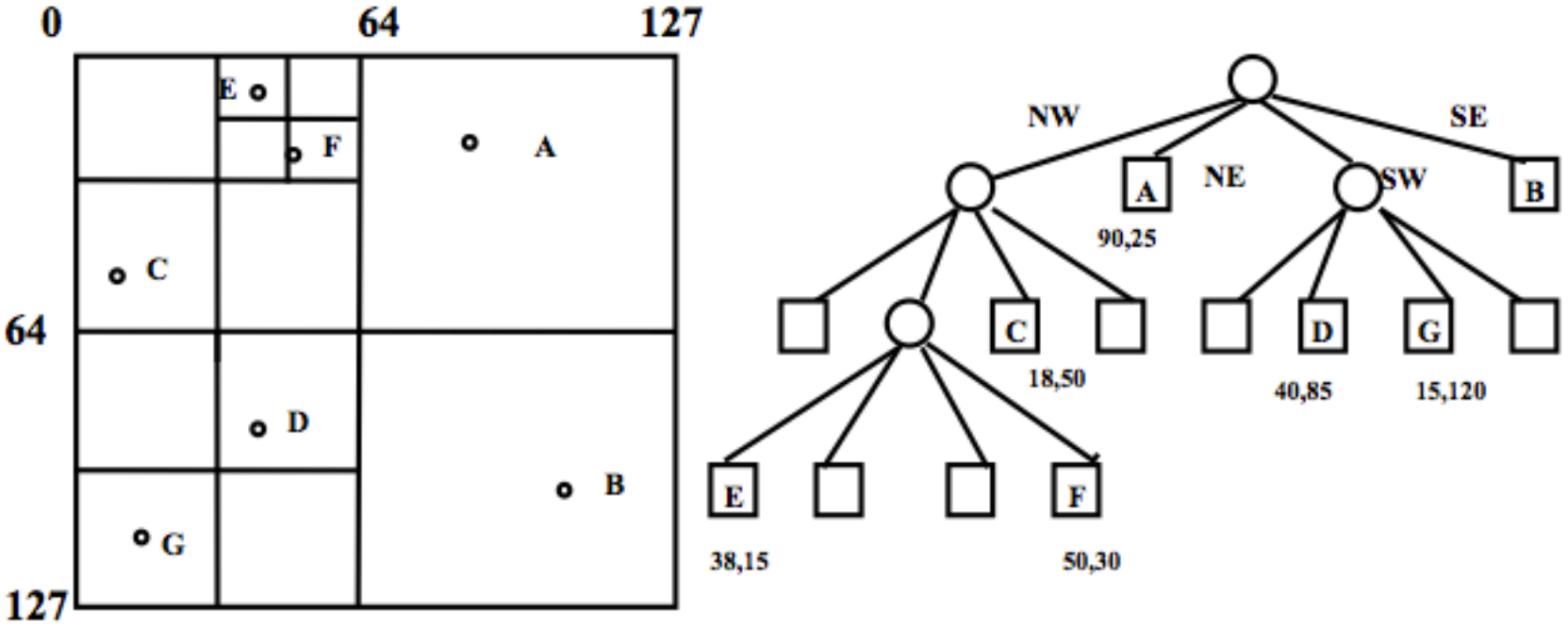
- 数据：N维空间中的一个点，(x1,x2,..xN)
- 区域：矩形(2维)，立方体(3维)等
- 数据在某个区域内：数据所对应的点在区域内部
- 目标点：查找操作中要查找的数据

空间数据结构

- PR 树
- K-D 树
- R 树

PR树(四分树)

- **定义：**点-区域四分树（Point-Region Quadtree）
- **空间划分：**NW(西北)、NE(东北)、SW(西南)和SE(东南)
- **检索等基本操作**



定义

- 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
- 完全四叉树

PR树的空间划分

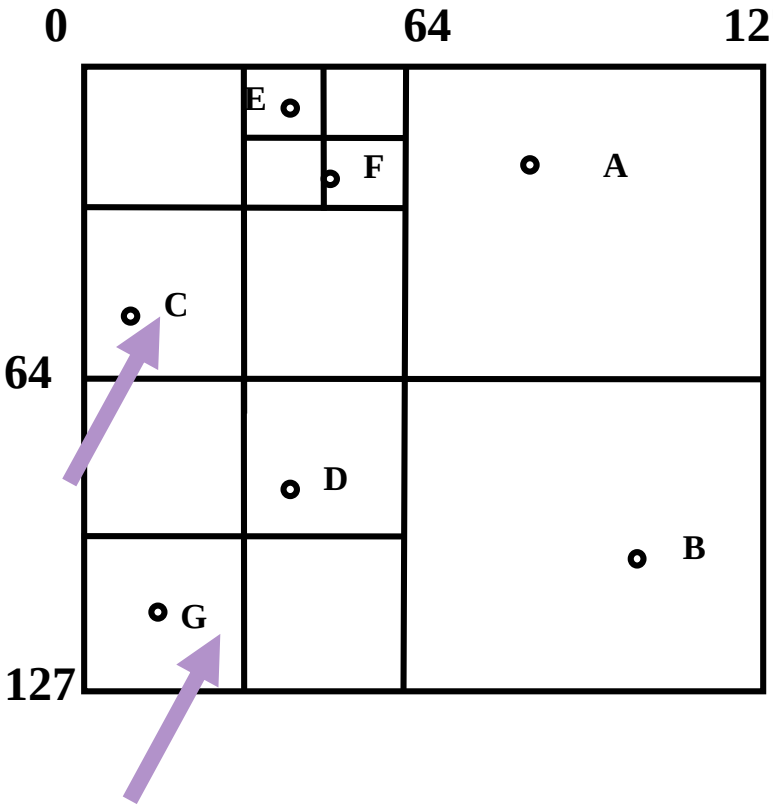
- 每个内部结点将当前空间均等地划分为四个区域
  - 如果子区域包含的数据点数大于1，那么就把该区域继续均等地划分为四个区域
  - 依次类推，直到每个区域所包含的数据点不超过一个为止

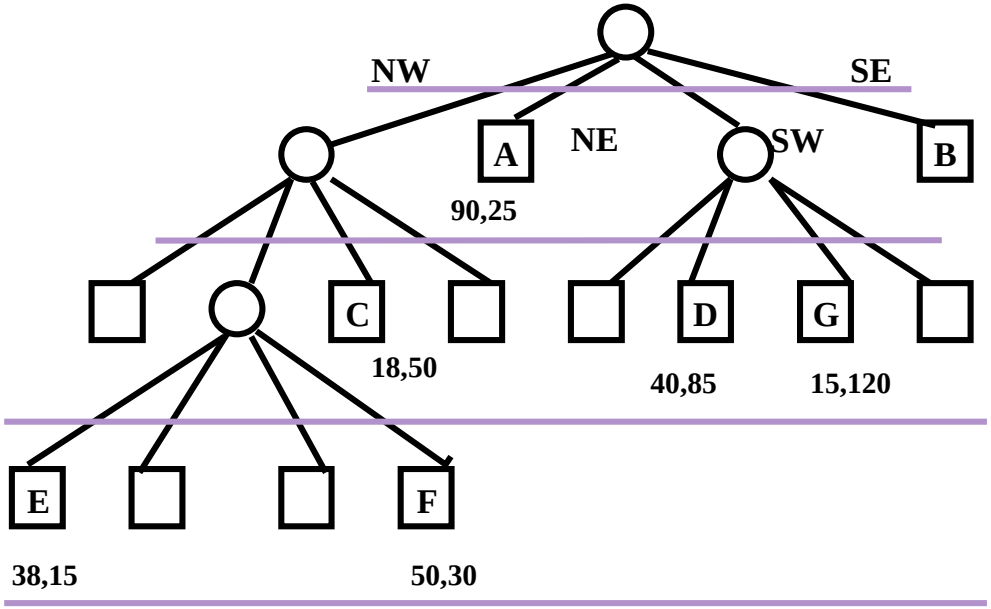
PR树的图示

对象空间为128×128， 点A、B、C、D、E、F和G

顶层空间平分为4份: NW,NE,SW,SE

NW,SE (包含多个数据点)进一步分裂





PR树的空间划分

- 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

基本操作

- 插入
- 删除
- 区域查询

PR树的插入

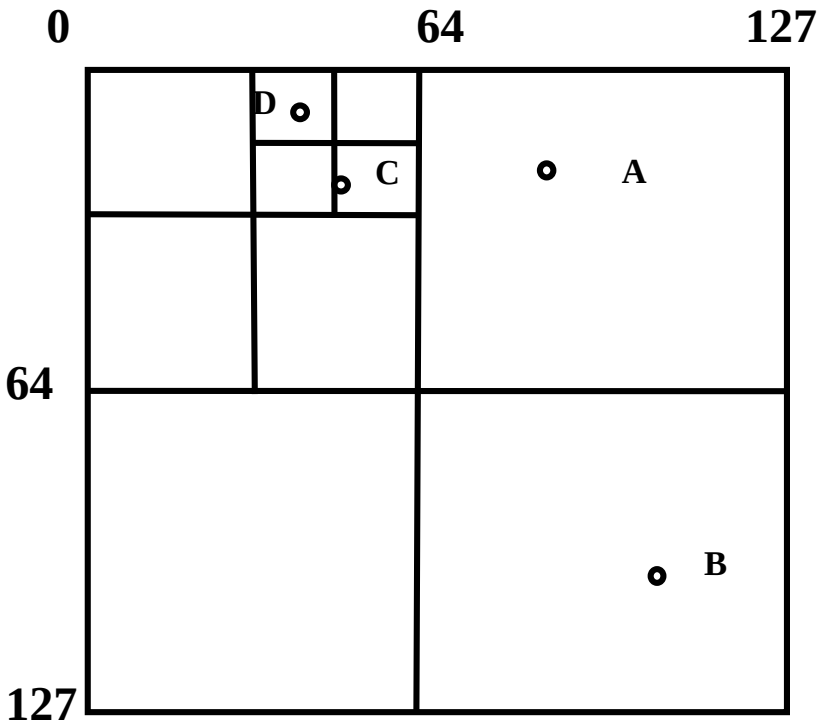
首先通过检索确定其位置

- 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（删除D）



删除结点D

合并空间

再合并

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

PR树的区域查询

- 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

K-D树

- 定义
- 构造
- 检索
- 基本操作

定义

- K-D树是特殊二叉树，用于多维检索
- 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定的最大点数时，划分结束

构造

- 空间分解
- 识别器
- 结点的分配

kd树的构建

1.确定被分割的维度：对于所有数据，统计它们在各个维度上的方差，选取方差最大的那个维度进行分割(数据点分散得比较开)

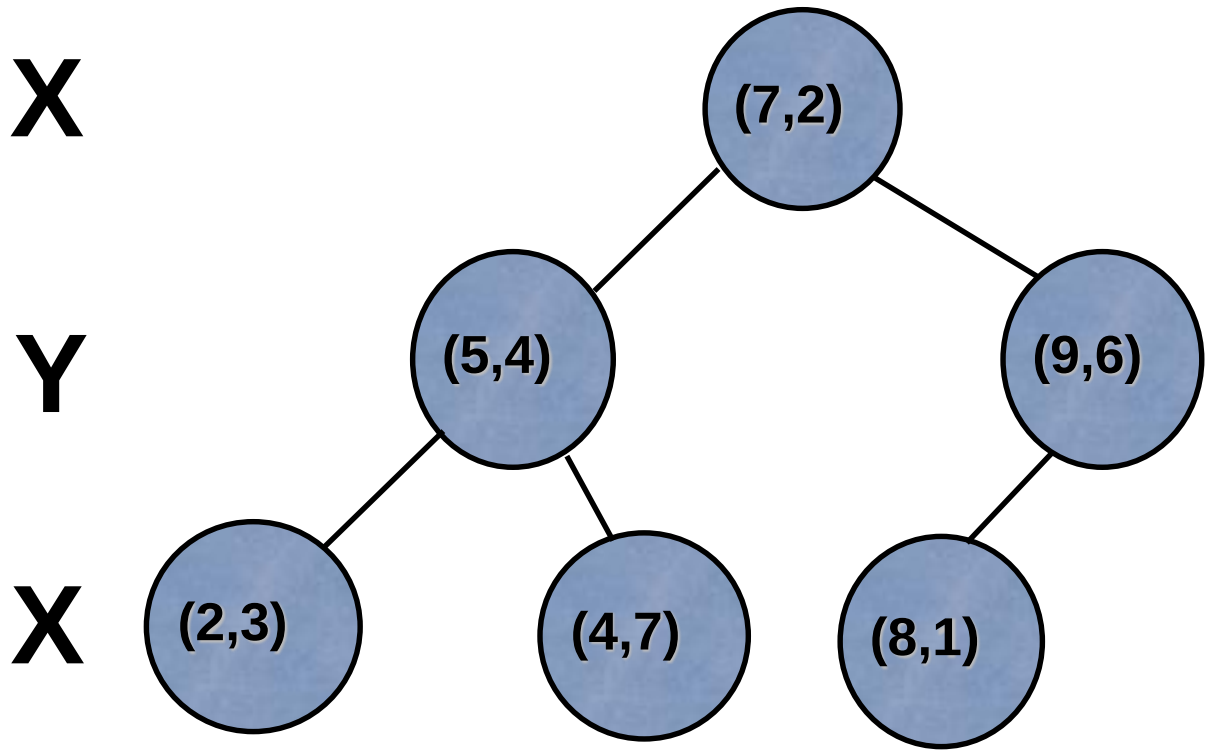
2.对数据点按照刚才确定的维度的值进行排序。记位于正中间的数据为x，那么当前结点的信息就是x，所有小于x(只看切割的维度)的数据被分配到左子树，大于或等于x的数据被分配到右子树。{递归进行1,2操作}

时间复杂度O(k*n*logn)

kd树的构建

据点{(2,3),(5,4),(9,6),(4,7),(8,1),(7,2)}

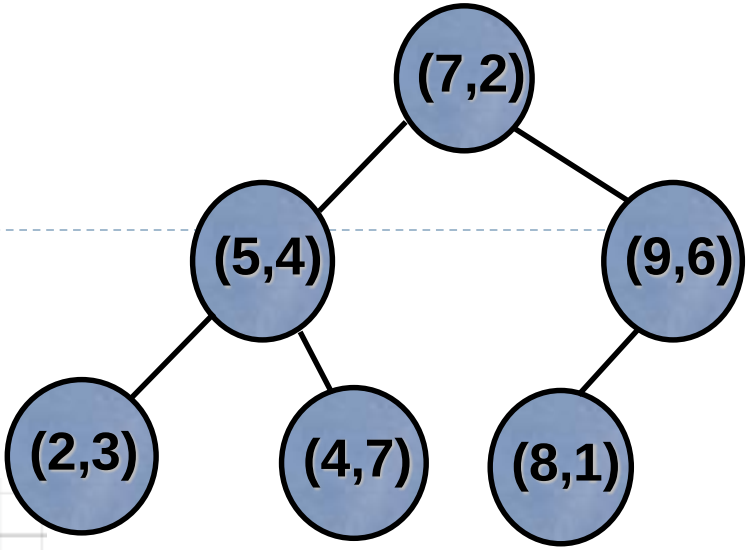
1. 6个数据在x,y维度上的方差分别为39,28.63，选x
2. 根据x维上的值进行排序，中间点为(7,2)

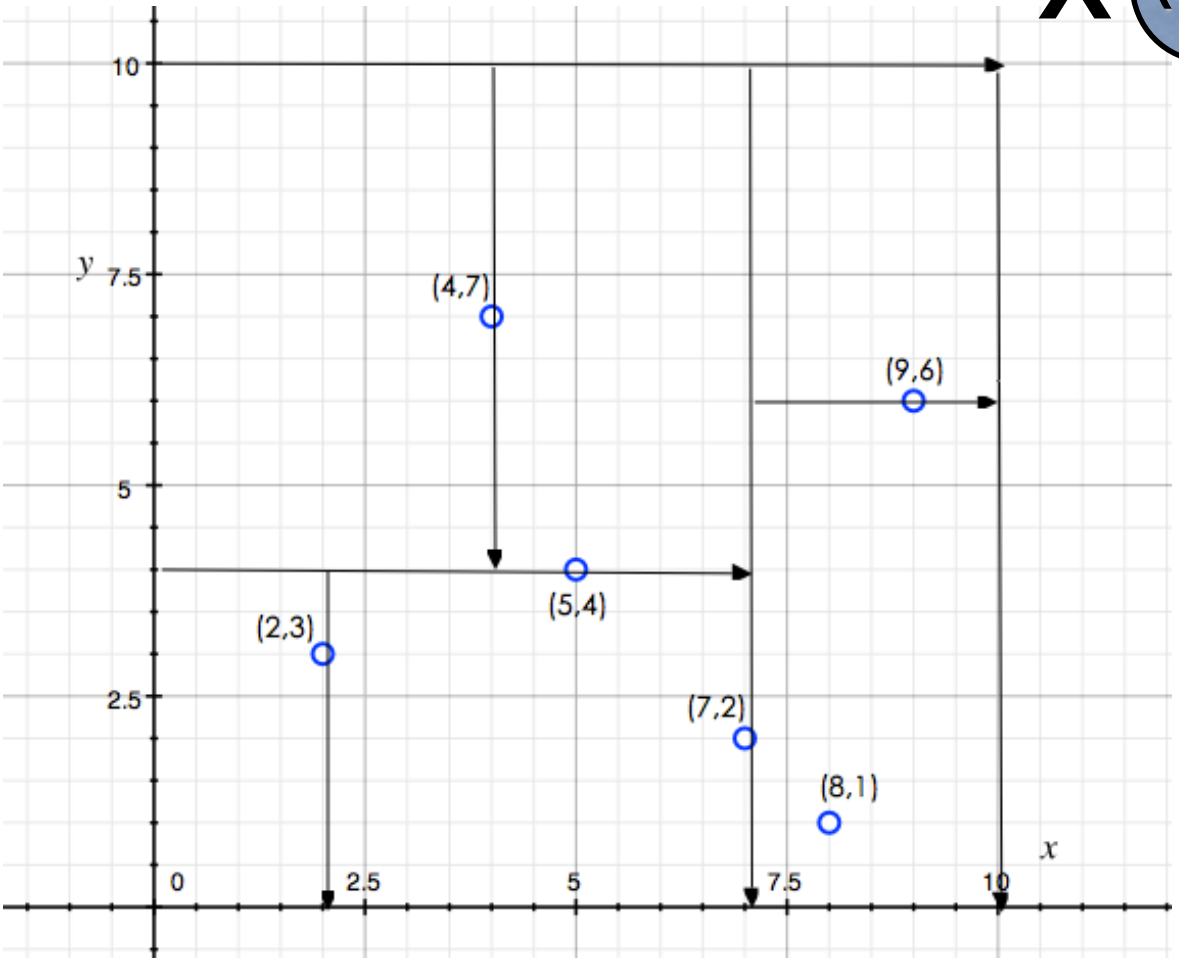


X

Y

X





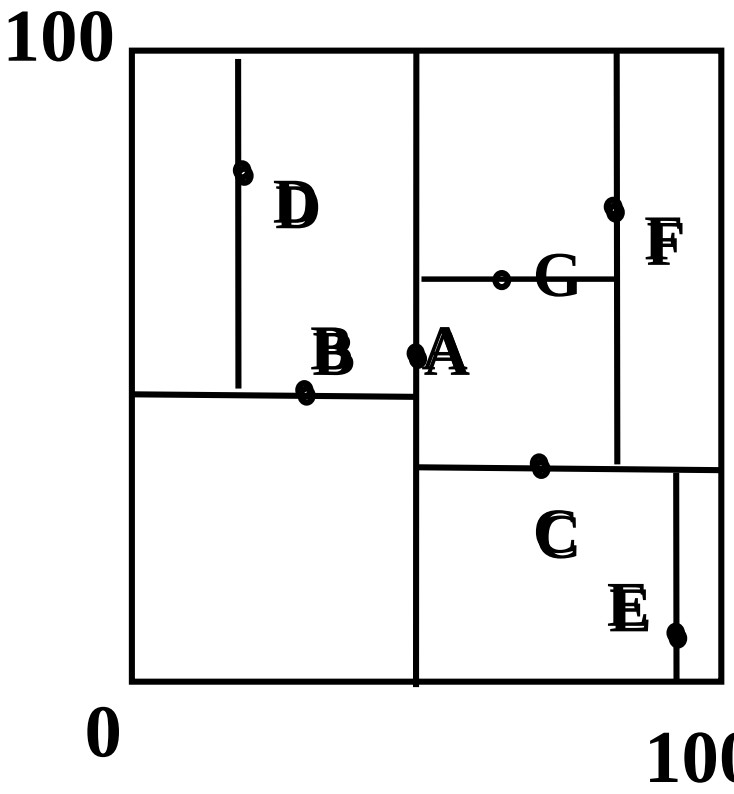
K-D树示例

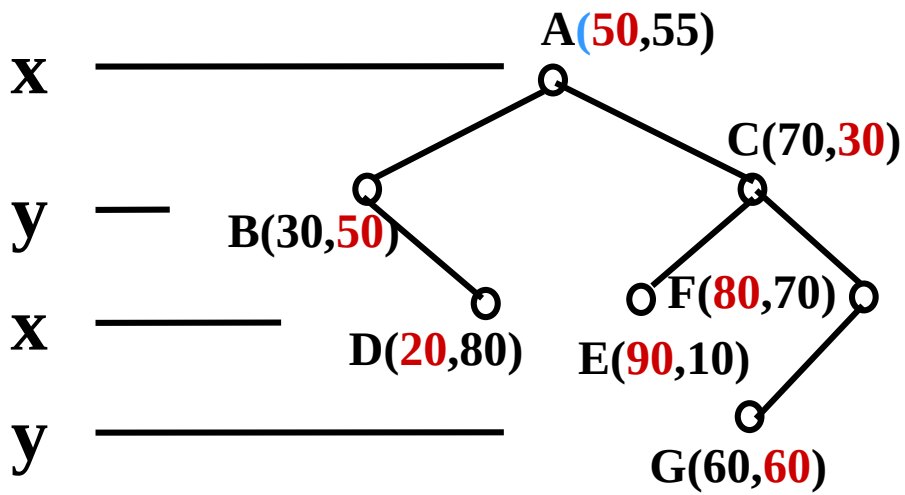
x: A(50,55)

y: B(30,55) C(70,30)

x: D(20,80) E(90,10) F(80,70)

y: G(60,60)





- 二维的k-d树，限制其取值范围为100×100之内

K-D树的空间分解

- 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

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，标号分别为0, 1, 2…k，在第 I 层把识别器的标号定义为 i mod 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0层是 0 mod 3=0，所以使用关键码 x
    - 第1层是 1 mod 3=1，所以使用关键码 y
    - 第2层是 2 mod 3=2，所以使用关键码 z

结点的分配

- 在结点分配的时候先比较该层的识别器
  - 若关键码小于识别器的值就放入左子树
  - 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

建KD树

Algorithm BUILDKDTREE($P, depth$)

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** $depth$ is even
4. **then** Split P with a vertical line ℓ through the median x-coordinate into P_1 (left of or on ℓ) and P_2 (right of ℓ)
5. **else** Split P with a horizontal line ℓ through the median y-coordinate into P_1 (below or on ℓ) and P_2 (above ℓ)
6. $v_{\text{left}} \leftarrow$ BUILDKDTREE($P_1, depth + 1$)
7. $v_{\text{right}} \leftarrow$ BUILDKDTREE($P_2, depth + 1$)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

创建KD树效率分析

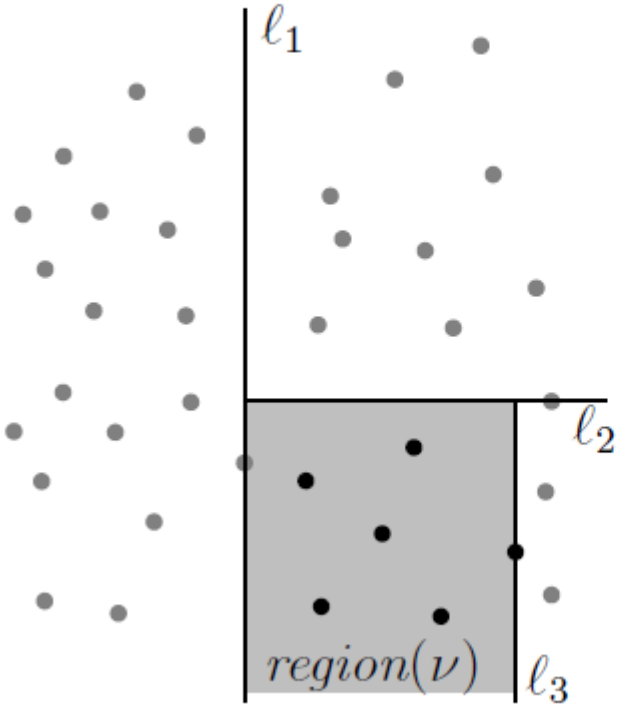
- 求中位数：O(n)

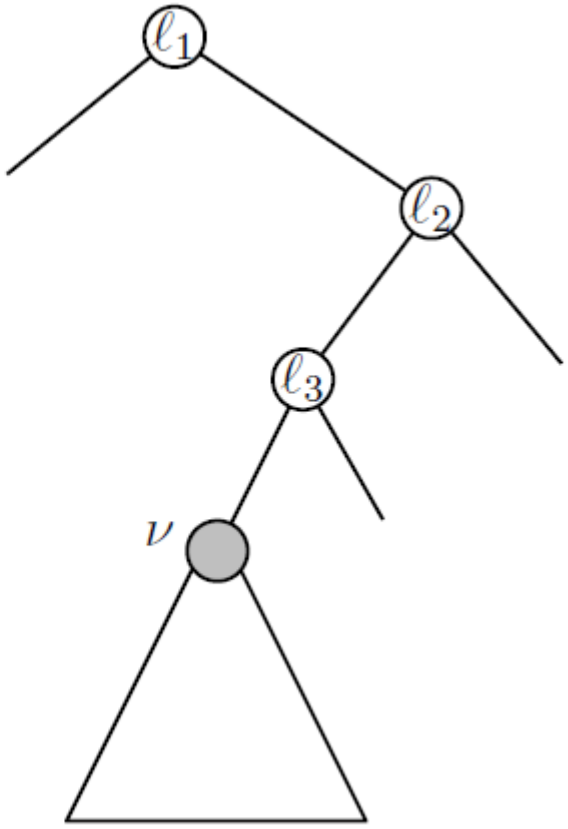
- T(1) = O(1)
- T(n) = 2T(n/2)+O(n)

总时间：O(nlogn)

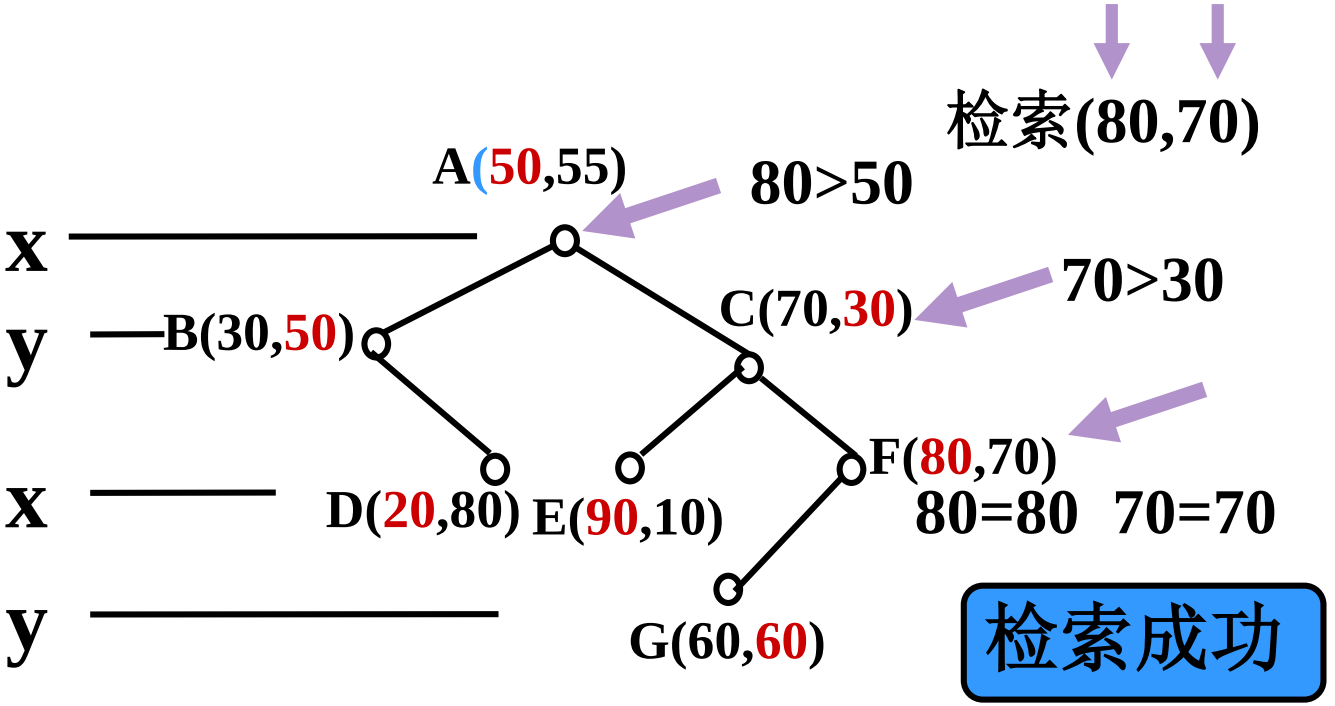
K-D树的检索

- 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





K-D树的检索



- 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树的检索算法

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

基本操作

- K-D树的插入
- K-D树的删除
- K-D树的范围查找

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

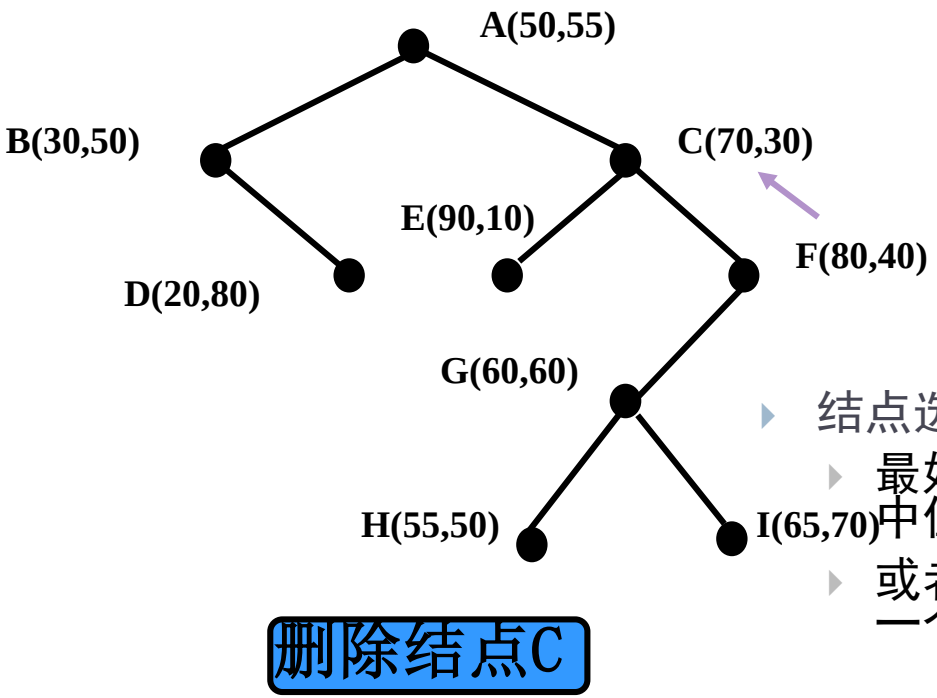
- 如果删除一个没有子结点的结点，不用做任何的调整
- 如果删除了一个有子结点的结点
  - 如果只有一个子结点并且子结点没有自己的子结点，那么就可以用它来取代被删除结点的位置
  - 而如果子结点还有自己的子结点，那么就需要进行选择

K-D树的删除（结点的选择）

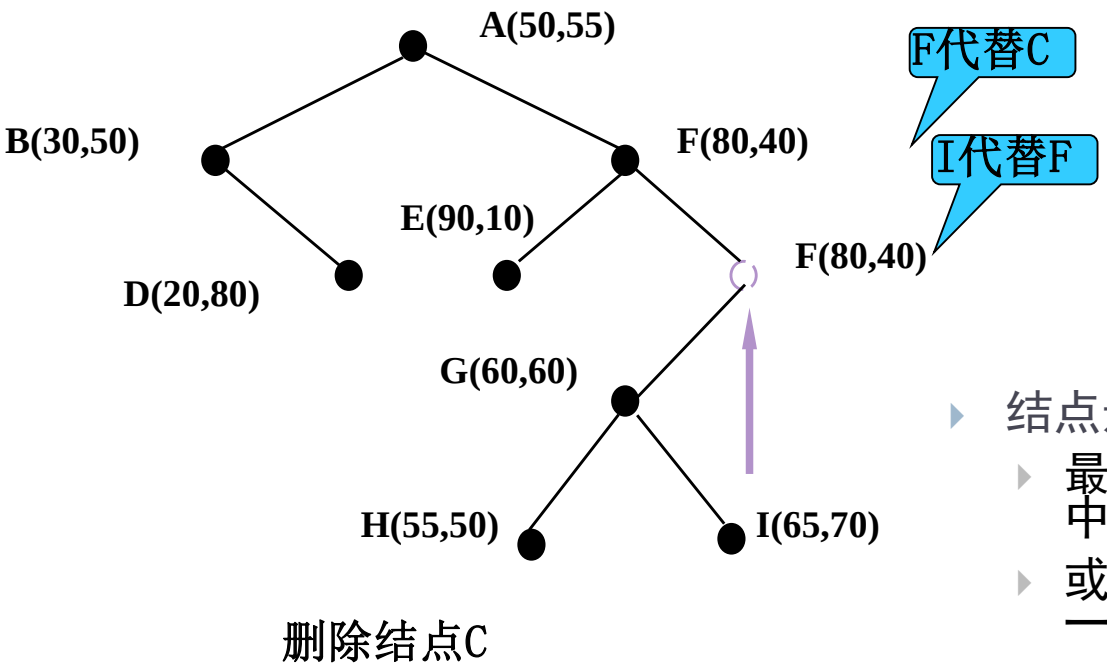
- 删除一个结点后需要从它的左右子树中选择一个合适的结点来替代它
  - 该结点最好能够保持原来的空间划分
    - 最好的选择是左子树中当前维中值最大的一个结点
    - 或者右子树中当前维值最小的一个结点

K-D树的删除（重复调整）

- 删除过程也是一个迭代的过程
  - 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
  - 它的子结点也要进行相关的动作
  - 直到没有结点移动为止

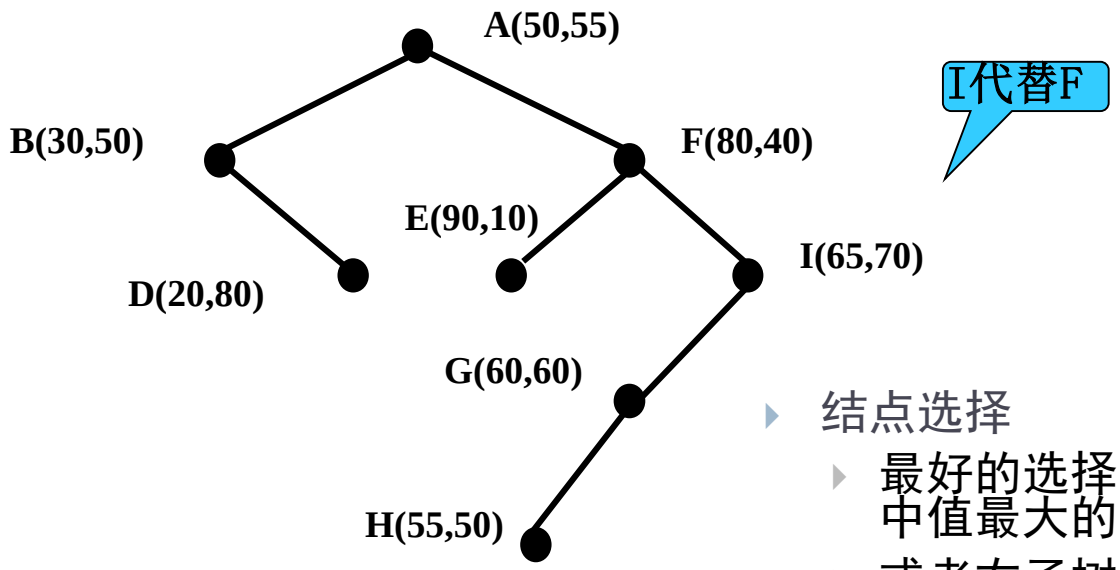


- 结点选择
  - 最好的选择是左子树中当前维中值最大的一个结点
  - 或者右子树中当前维值最小的一个结点



删除结点C

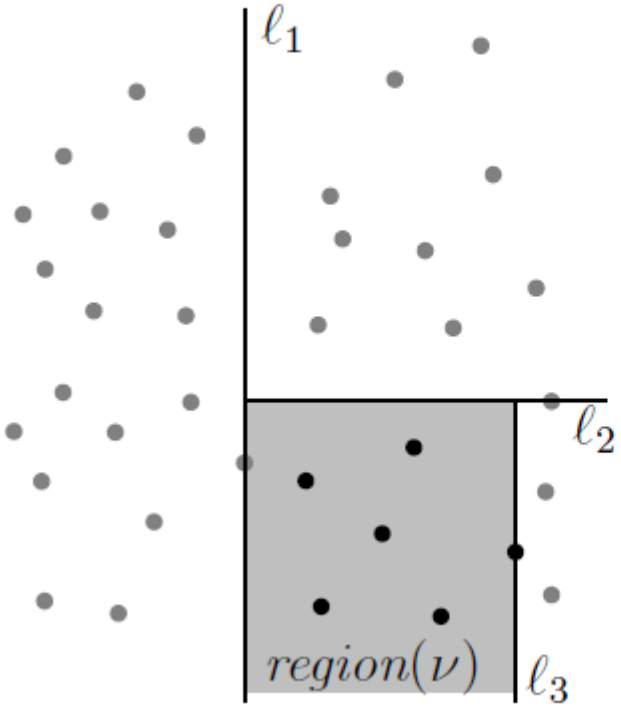
- 结点选择
  - 最好的选择是左子树中当前维中值最大的一个结点
  - 或者右子树中当前维值最小的一个结点

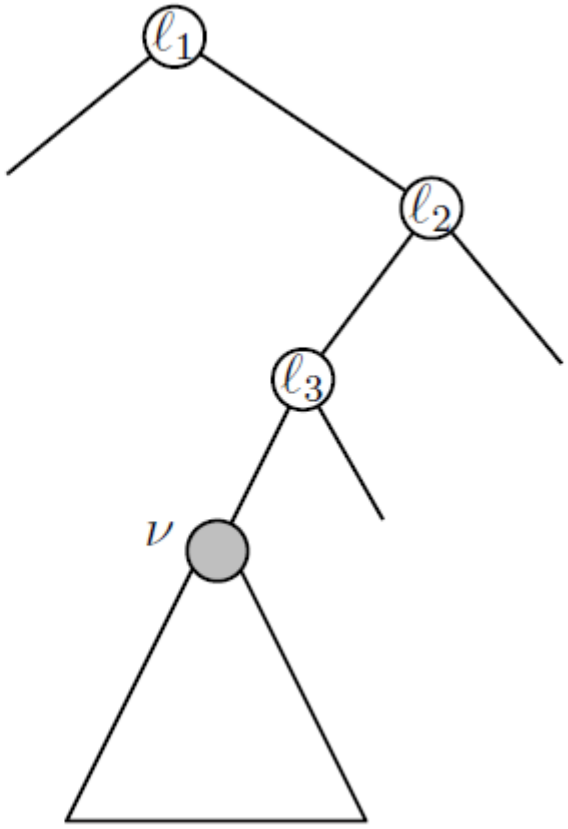


- 结点选择
  - 最好的选择是左子树中当前维中值最大的一个结点
  - 或者右子树中当前维值最小的一个结点

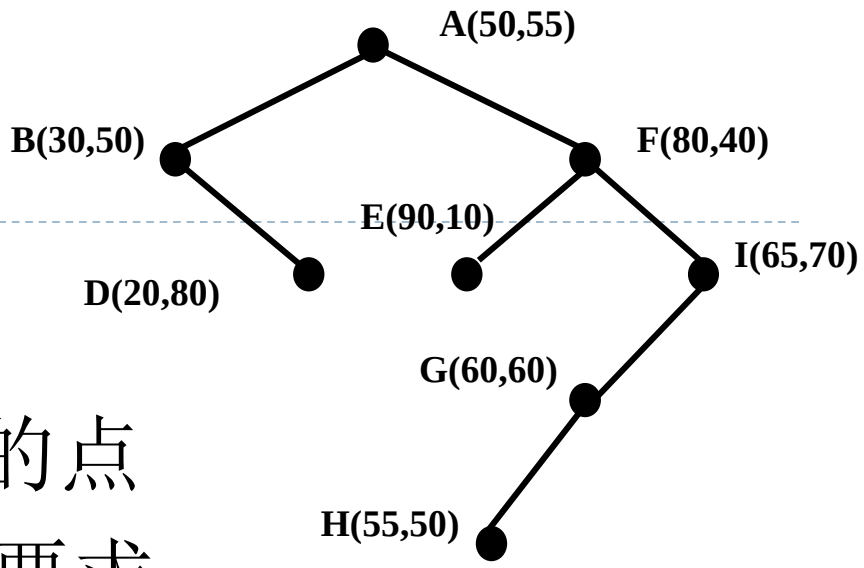
K-D树的范围查找

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





K-D树的范围查找



与（50，70）距离小于20的点

- 根结点A（50，55）满足要求
- 进入左子树，点B（30，50）不在范围内
  - 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）的左边子树应该抛弃，因为左边子树的y<50。
    - 进入其右子树发现结点D（20，80）也不符合。
- 类似地，进入根结点的右子树进行搜索

kd树查找最近点

给定k维空间上的点x，查找数据集中离x最近的点

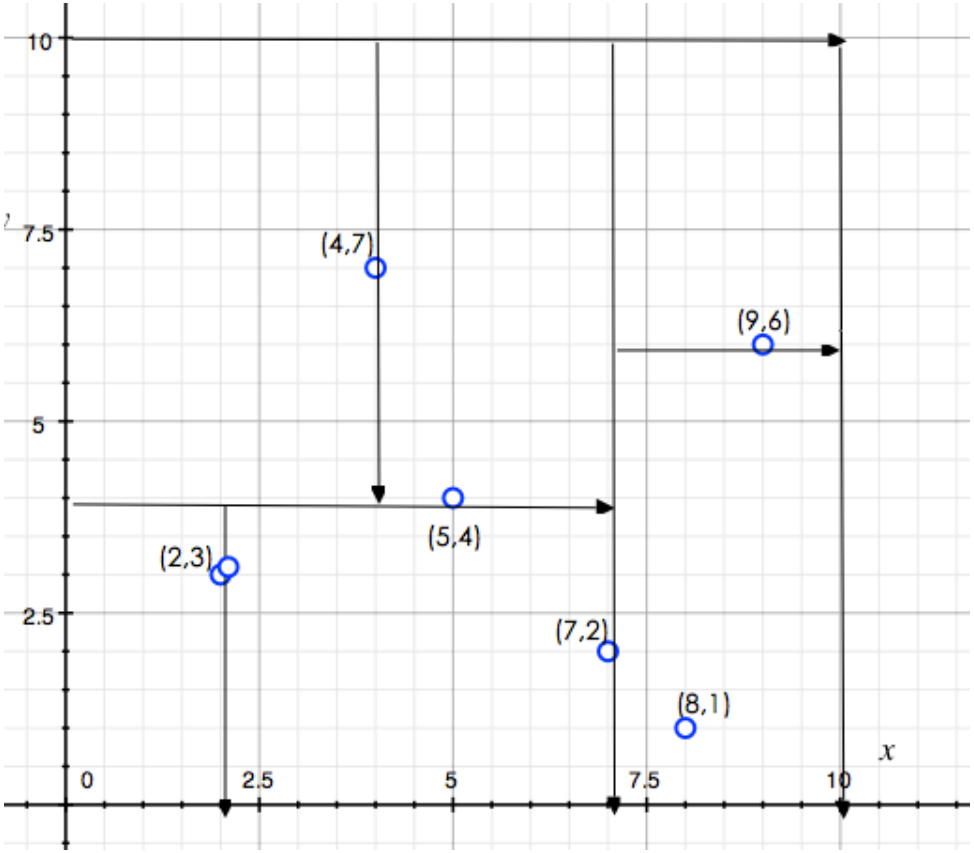
通过二叉搜索，顺着搜索路径一直走到叶子结点。这样可以找到一个近似的最近点，即叶子到根之间的某个点

还有一些潜在点(可能是最近点)没有遍历。这些潜在点一定是在以查询点为圆心,查询点到近似最近点的距离为半径的圆内

回溯时，如果圆跨过了当前点所在的半平面，那么就查询另外那棵子树

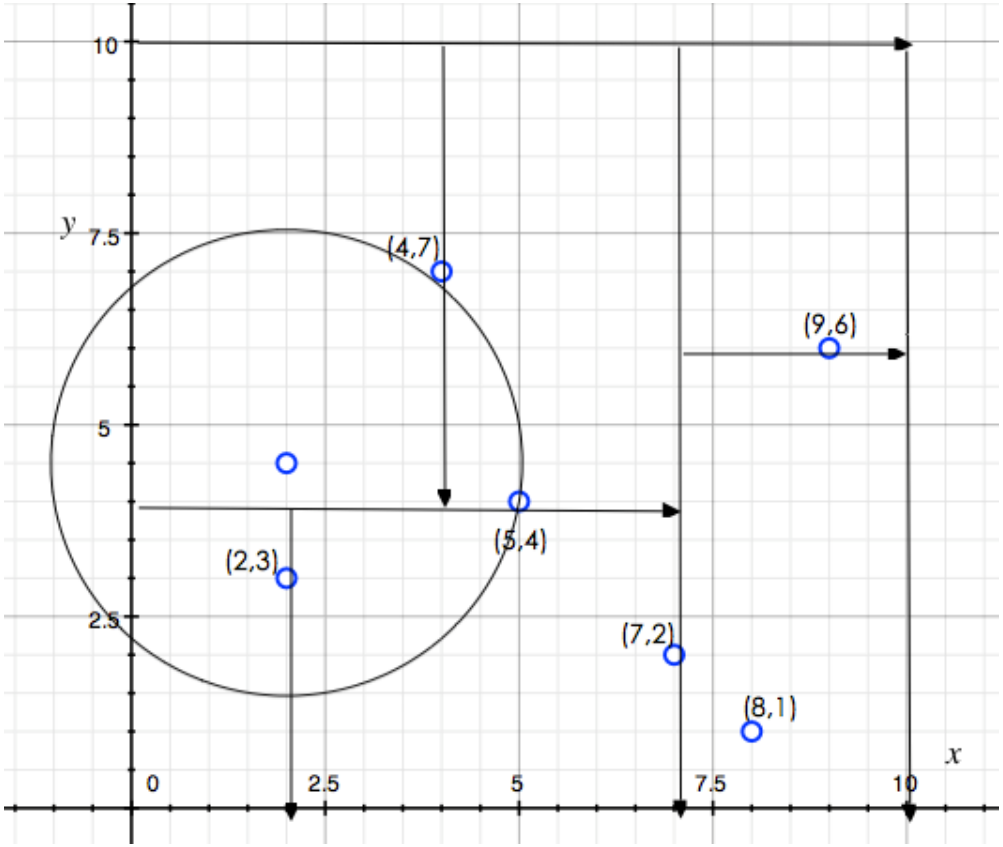
kd树的查找

查询(2.1,3.1)。从根一直到叶子结点，搜索路径为<(7,2),(5,4),(2,3)>,(2,3)是近似邻近点，距离为0.1414



kd树的查找

查询点(2,4.5)，搜索路径<(7,2),(5,4),(4,7)>，近似最近点为(5,4)。回溯时发现跨过了y = 4，那么需要进入(5,4)的左子树进行查找。找到(2,3)



kd树的查找伪代码

```
find(Node A, point x) {  //x给定点, A是kd树的某个结点
    dist = Distance(x, A.x); // A.x是数据集中的某个点
    update(ans, dist, A.x);//ans是最终答案，更新ans的值
    if (compare(A, x)==Left ) // compare只比较当前维度
        find(A.left, x);
    else find(A.right, x); // 查找x所属的区域
    // 判断以x为圆心ans为半径的圆是否穿越A.x所在半平面
    if cross(circle(x, ans), A.x)
        find(anotherSubtree(x,A,x), x);//查找另外一棵子树
}
```

kd树查找优化

是否跨过半平面取决于当前近似最近点

回溯后回到原来结点的父亲，这种方法太盲目?

回溯后回到离目标点最近的某个结点？

对查询路径进行排序？

BBF算法(Best-Bin-First)

基于在kd树查找

对于每个结点定义优先级，即离目标点的距离

维护一个优先队列，优先队列里面存的是查询路径上结点对应的相反结点。比如：搜索左子树，就把这一层的右子树存进队列

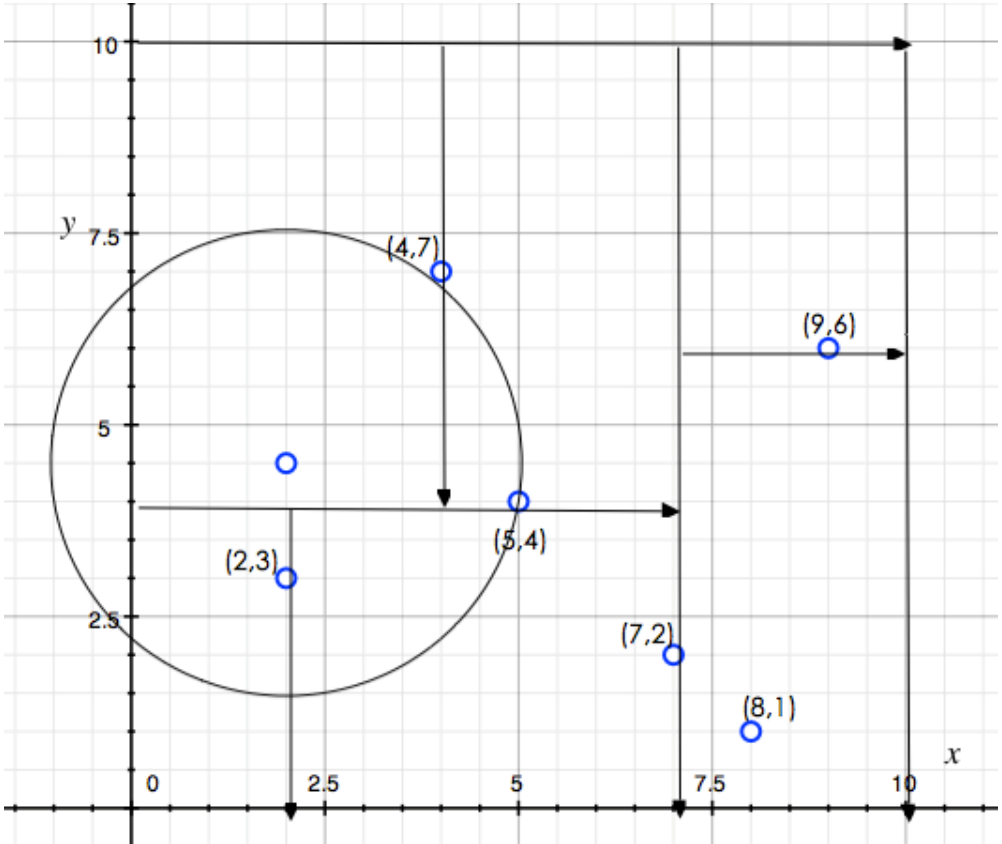
每次回溯时，从优先队列中选取一个优先级最高的结点。从那个结点继续进行查找操作。优先队列为空就停止

BBF算法：以查询(2,4.5)为例

1.将(7,2)加入优先队列

2.从优先队列中取出(7,2)，由于(2,4.5)位于(7,2)的左子树，所以检索左子树，并把右子树(相反结点)(9,6)加入优先队列。一直检索到叶子结点(4,7),此时优先队列为{(2,3),(9,6)}。

3.从优先队列中取出(2,3)



M个最近点的查找

查找M个最近的邻居

先看一个简化版的问题：给定n个整数，求出其中最小的m个数

简化版问题

维护m个元素的最大堆。(算法结束后，堆中元素就是最小的m个数)

扫描这n个数。对于当前数x，如果最大堆的元素个数小于m，那么直接把x添加进堆中。否则，比较x与堆顶元素的大小。如果x更小，就删除堆顶元素，把x加进堆中。时间复杂度O(n*logk)

这个思想能不能用到BBF算法的查询中呢？

M个最近点的查找

在原来优先队列的基础上，多维护一个M个元素的最大堆(M个近似最近点)

原来的回溯过程是，设优先队列优先级最高的元素为x。如果目标点和当前近似最近点构成的圆跨过x的半平面，那么就继续查找。

回溯过程中，如果堆的元素小于M，就继续查找。等于M，就判断目标点和堆顶元素构成的圆是否跨过x的半平面。

SIFT算法中的应用

sift是图片匹配算法

物体辨识、机器人地图感知与导航、影像缝合、3D模型建立、手势辨识、影像追踪和动作比对

在求出sift特征向量后，需要对某个关键点，求出欧式距离最近的两个关键点

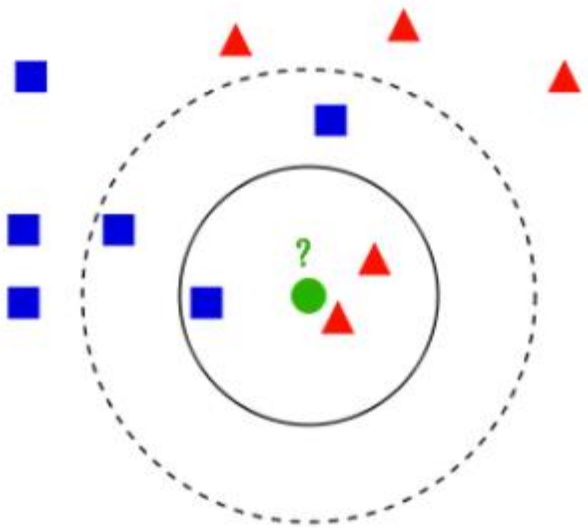
KNN算法

K-Nearest Neighbor algorithm，机器学习中的一个比较简单的分类算法

给定一个训练数据集，对新的输入实例，在训练集中找到与该实例最近的K个实例，那么新的实例属于K个实例中最多的那个类

KNN算法

有两种不同的样本数据(两类)，用蓝色正方形和红色三角形表示。绿色的圆表示待分类的数据(新实例)，确定绿色的圆是属于蓝色正方形还是红色三角形



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，**根结点是包含所有点的超矩形**
  - 所有的真实数据记录都存储在叶结点
  - 内部结点是分割，用于各维之间导航（同KD树）
  - 每一个识别器的选择不再依赖于输入的数据
    - 尽量让左右子树的记录数目相等（平衡的思想）

R树

- 引入
- 构造
- 性质
- 矩形重叠覆盖
- 插入
- 删除

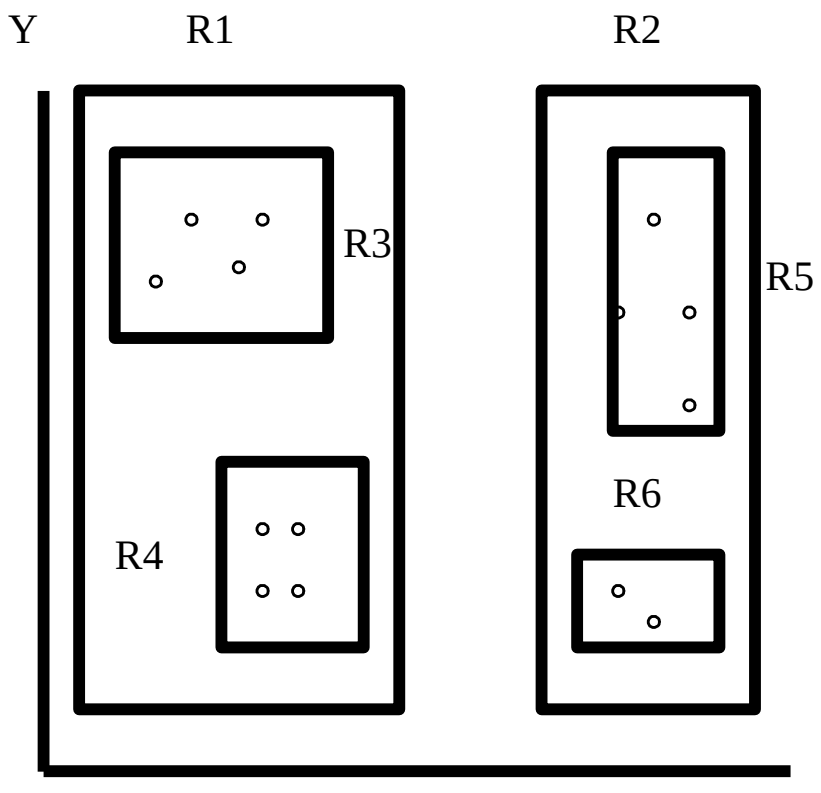
引入

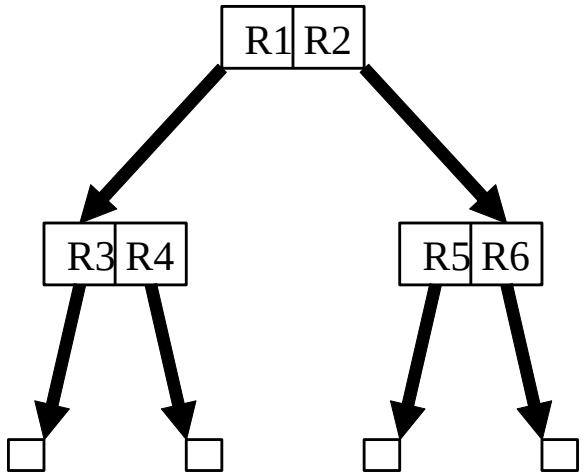
- R树（Rectangle）用来存储高维数据的平衡树
  - 采用了B树分割空间的思想
  - 在添加、删除操作时采用合并、分解结点的方法，保证树的平衡性

构造

- R树运用了空间分割的理念
- R树采用了一种称为MBR(Minimal Bounding Rectangle)的方法，译作“最小边界矩形”
- 从叶子结点开始用矩形（rectangle）将空间框起来，结点越往上，框住的空间就越大，以此对空间进行分割。

R树的图示





左边图的矩形和右边的编号对应，而点则表示数据对象；右边是构建好的一棵**R**树，根结点包含两个（指针，矩形）的实体，叶结点包含指向数据的指针和矩形编号

例子图示

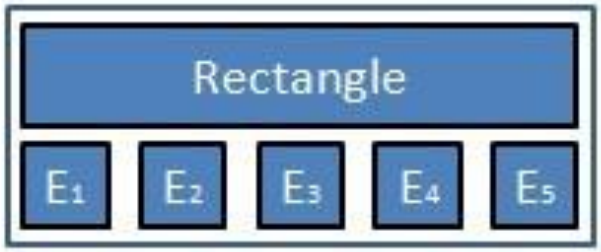

Tianxin
Huashengliao
Huaqiaobu
Cross Rd
Yongcun
DFM Garden
Tianrun Rd
Huangpu W Ave
Xiancun
Xinqingcun
Jinsui Rd
Meihuacun
Wuyangcun
Liedecun
Sanmuwei
Wenge
Yumin New Village
Shipai Village

查询特定的数据

- 以餐厅为例，假设我要查询广州市天河区天河城附近一公里的所有餐厅地址怎么办？
- 打开地图（也就是整个R树），先选择国内还是国外（也就是根结点）。
- 然后选择华南地区（对应第一层结点），选择广州市（对应第二层结点），再选择天河区（对应第三层结点），最后选择天河城所在的那个区域（对应叶子结点，存放有最小矩形），遍历所有在此区域内的结点，看是否满足我们的要求即可。

叶子结点

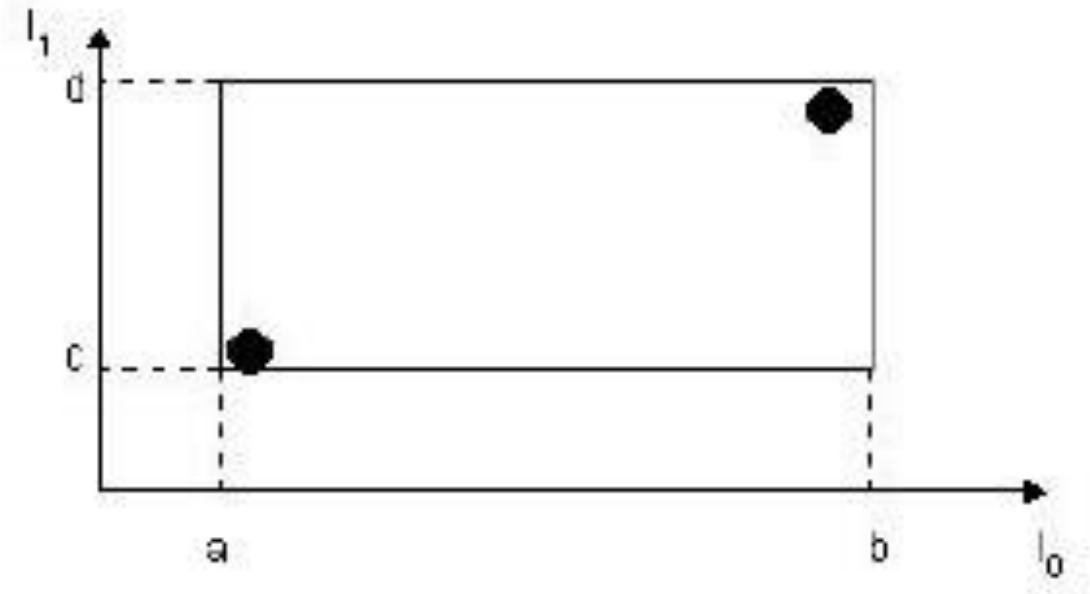
- 叶子结点所保存的数据形式为：(I, tuple-identifier)。
  - tuple-identifier表示的是一个存放于数据库中的tuple，也就是一条记录，它是n维的。
  - I是一个n维空间的矩形，并可以恰好框住这个叶子结点中所有记录代表的n维空间中的点。I=(I0,I1,...,In-1)。
  - 其结构如下图所示：



R树中结点的存储结构。E代表Entry，即指向孩子结点的条目

叶子结点

- 下图描述的就是在二维空间中的叶子结点所要存储的信息:



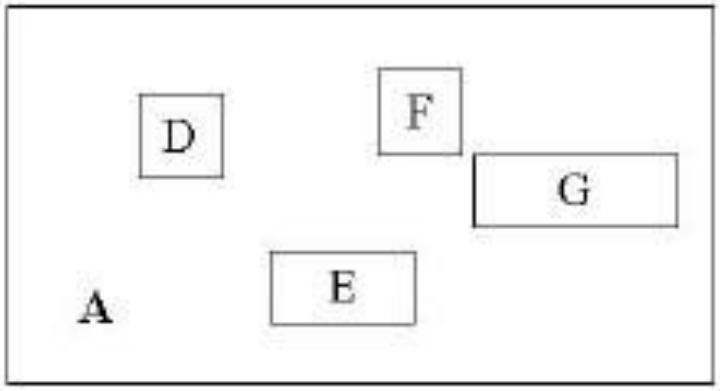
- I所代表的就是图中的矩形，其范围是a<=I0<=b，c<=I1<=d。有两个tuple-identifier，在图中即表示为那两个点。

非叶子结点

- 非叶子结点的结构其实与叶子结点非常类似。
- R树的非叶子结点存放的数据结构为：(I, child-pointer)。
- child-pointer是指向孩子结点的指针。
- I是覆盖所有孩子结点对应矩形的矩形。

非叶子结点

- 图示：



- D,E,F,G为孩子结点所对应的矩形。A为能够覆盖这些矩形的更大的矩形。这个A就是这个非叶子结点所对应的矩形。无论是叶子结点还是非叶子结点，它们都对应着一个矩形。树形结构上层的结点所对应的矩形能够完全覆盖它的孩子结点所对应的矩形。根结点也唯一对应一个矩形，而这个矩形是可以覆盖所有我们拥有的数据信息在空间中代表的点的。

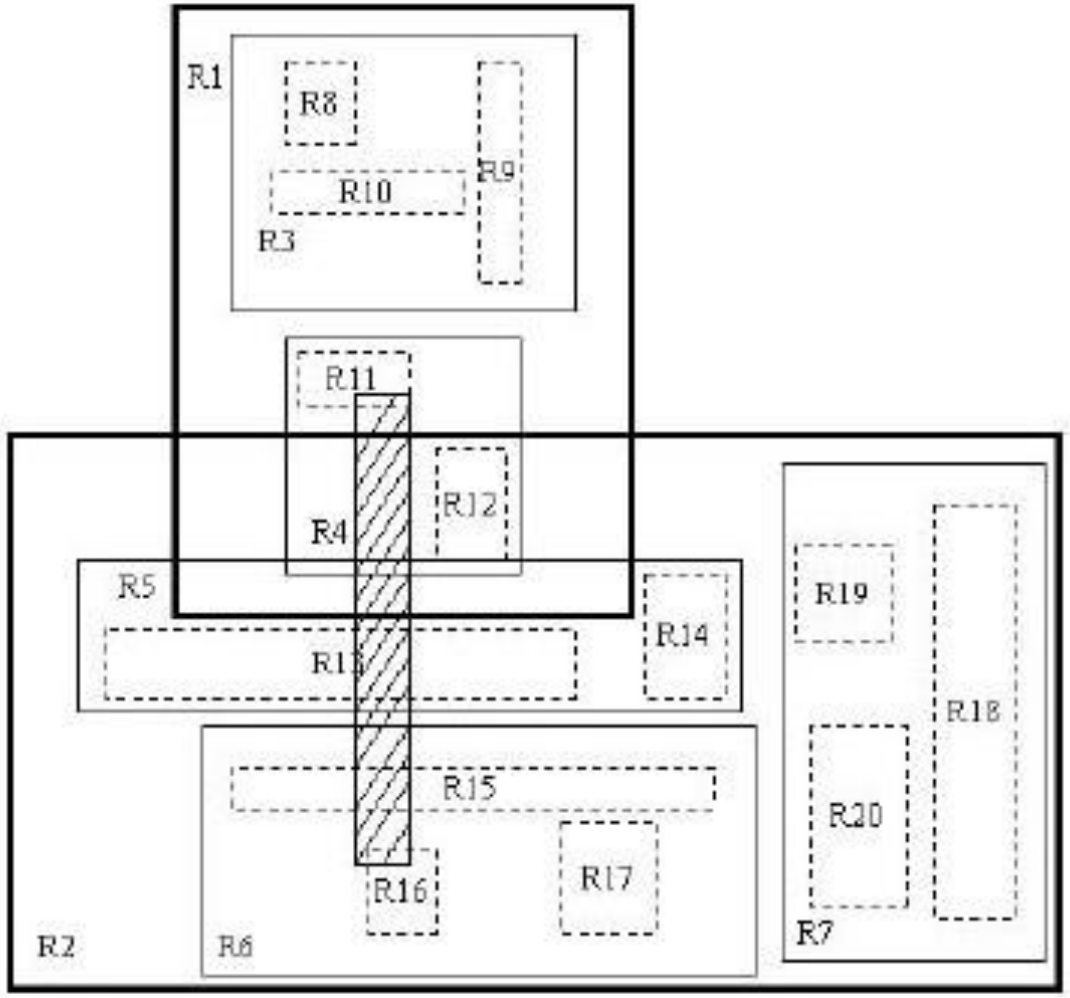
R树的性质

- R树应该满足下面的条件：
  - 1.除根结点之外，所有叶子结点包含有m至M个记录索引（条目）。作为根结点的叶子结点所具有的记录个数可以少于m。通常，m=M/2。
  - 2.对于所有在叶子中存储的记录（条目），I是最小的可以在空间中完全覆盖这些记录所代表的点的矩形（注意：此处所说的“矩形”是可以扩展到高维空间的）。
  - 3.每一个非叶子结点拥有m至M个孩子结点，除非它是根结点。
  - 4.对于在非叶子结点上的每一个条目，i是最小的可以在空间上完全覆盖这些条目所代表的店的矩形（同性质2）。
  - 5.所有叶子结点都位于同一层，因此R树为平衡树。

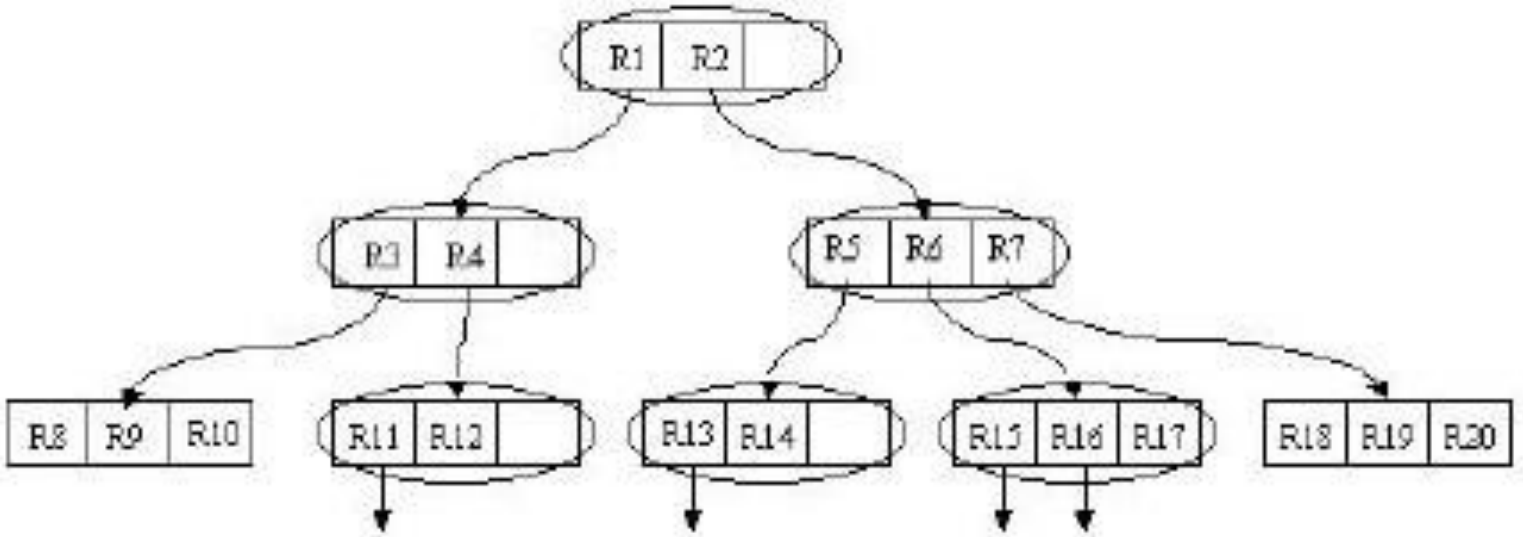
R树的搜索

- 输入的是一个搜索矩形
- 它返回的结果是所有符合查找信息的记录条目
- 描述：假设T为一棵R树的根结点，查找所有搜索矩形S覆盖的记录条目。
- S1:[查找子树] 如果T是非叶子结点，如果T所对应的矩形与S有重合，那么检查所有T中存储的条目，对于所有这些条目，使用搜索操作作用在每一个条目所指向的子树的根结点上（即T结点的孩子结点）。
- S2:[查找叶子结点] 如果T是叶子结点，如果T所对应的矩形与S有重合，那么直接检查S所指向的所有记录条目。返回符合条件的记录。

R树的搜索(例)

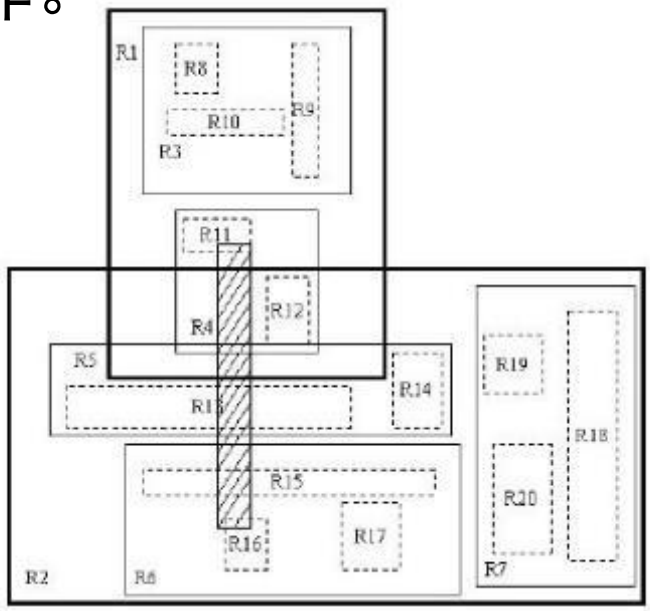

R1
R8
R9
R10
R3
R11
R12
R4
R5
R14
R19
R13
R18
R15
R20
R16
R17
R2
R6
R7

R树的搜索(例)


R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
R20

R树的搜索(例)

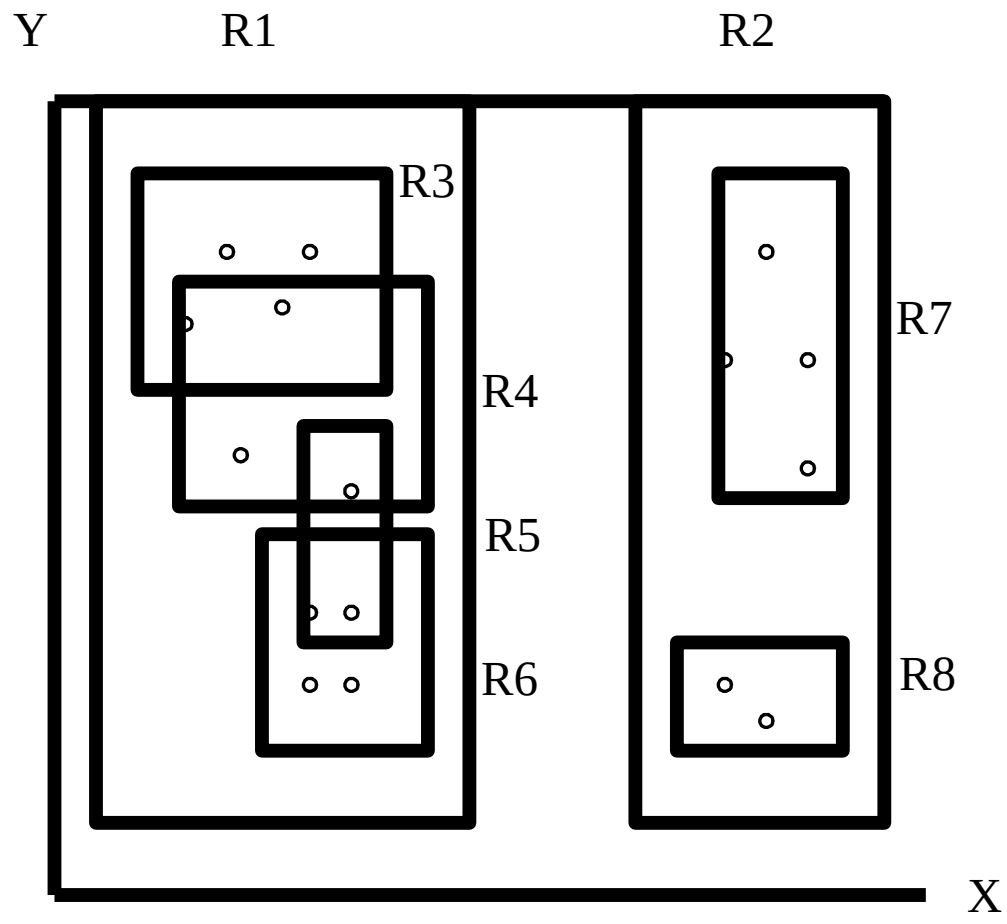
- 阴影部分所对应的矩形为搜索矩形。它与根结点对应的最大的矩形（未画出）有重叠。这样将搜索操作作用在其两个子树上。两个子树对应的矩形分别为R1与R2。搜索R1，发现与R1中的R4矩形有重叠，继续搜索R4。最终在R4所包含的R11与R12两个矩形中查找是否有符合条件的记录。搜索R2的过程同样如此。很显然，该算法进行的是一个迭代操作。



R树矩形重叠覆盖

- 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

矩形重叠图示



R树矩形重叠覆盖

- 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

插入

- R树的插入操作同B树的插入操作类似。
- 当新的数据记录需要被添加入叶子结点时，若叶子结点溢出，那么我们需要对叶子结点进行分裂操作。
- 显然，叶子结点的插入操作会比搜索操作要复杂。插入操作需要一些辅助方法才能够完成。

Function： Insert

- 描述：将新的记录条目E插入给定的R树中。
- I1：[为新记录找到合适插入的叶子结点] 开始ChooseLeaf方法选择叶子结点L以放置记录E。
- I2：[添加新记录至叶子结点] 如果L有足够的空间来放置新的记录条目，则向L中添加E。如果没有足够的空间，则进行SplitNode方法以获得两个结点L与LL，这两个结点包含了所有原来叶子结点L中的条目与新条目E。
- I3：[将变换向上传递] 开始对结点L进行AdjustTree操作，如果进行了分裂操作，那么同时需要对LL进行AdjustTree操作。
- I4：[对树进行增高操作] 如果结点分裂，且该分裂向上传播导致了根结点的分裂，那么需要创建一个新的根结点，并且让它的两个孩子结点分别为原来那个根结点分裂后的两个结点。

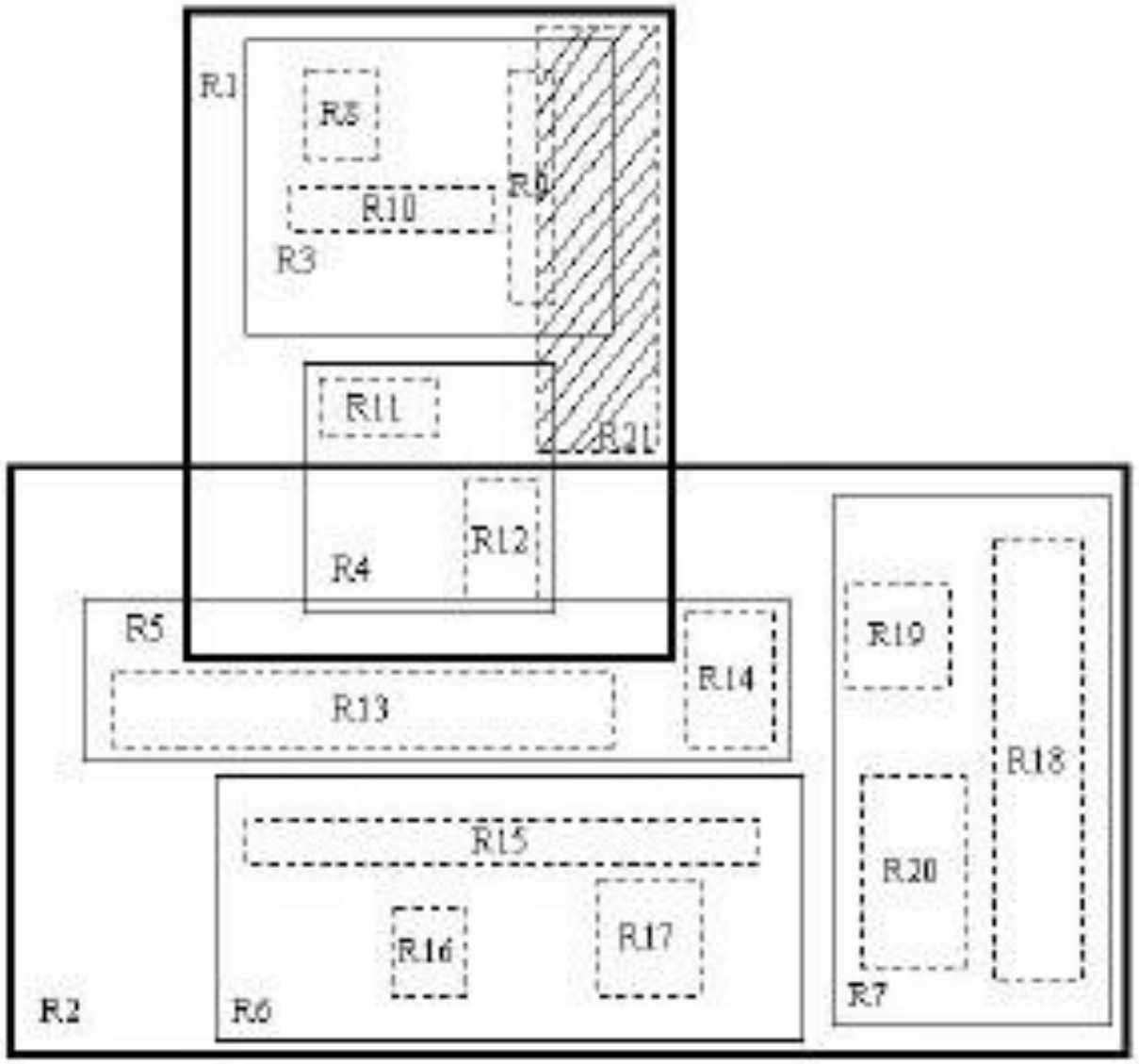
Function： ChooseLeaf

- 描述：选择叶子结点以放置新条目E。
- CL1：[Initialize] 设置N为根结点。
- CL2：[叶子结点的检查] 如果N为叶子结点，则直接返回N。
- CL3：[选择子树] 如果N不是叶子结点，则遍历N中的结点，找出添加E.I时扩张最小的结点，并把该结点定义为F。如果有多个这样的结点，那么选择面积最小的结点。
- CL4：[下降至叶子结点] 将N设为F，从CL2开始重复操作。

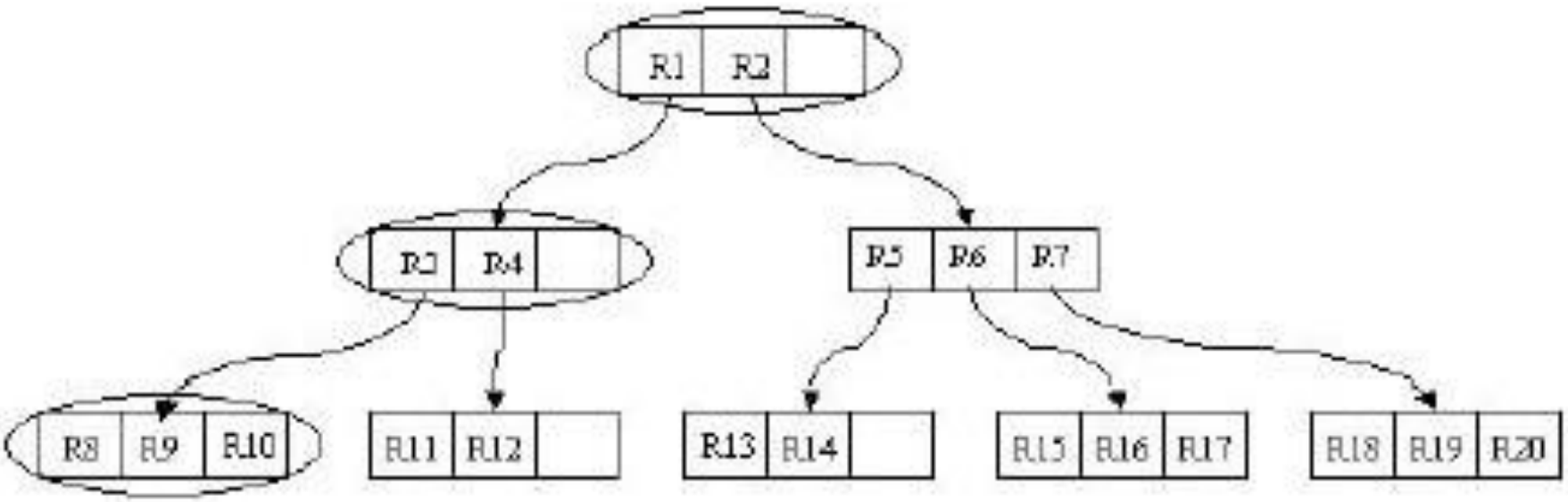
Function： AdjustTree

- 描述：叶子结点的改变向上传递至根结点以改变各个矩阵。在传递变换的过程中可能会产生结点的分裂。
- AT1：[初始化] 将N设为L。
- AT2：[检验是否完成] 如果N为根结点，则停止操作。
- AT3：[调整父结点条目的最小边界矩形] 设P为N的父结点，EN为指向在父结点P中指向N的条目。调整EN.I以保证所有在N中的矩形都被恰好包围。
- AT4：[向上传递结点分裂] 如果N有一个刚刚被分裂产生的结点NN，则创建一个指向NN的条目ENN。如果P有空间来存放ENN，则将ENN添加到P中。如果没有，则对P进行SplitNode操作以得到P和PP。
- AT5：[升高至下一级] 如果N等于L且发生了分裂，则把NN置为PP。从AT2开始重复操作。

插入（例）

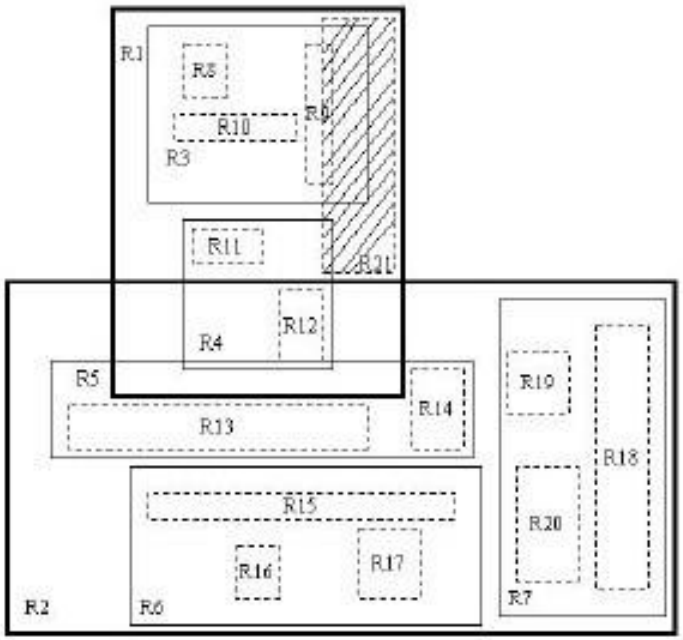

R1
R8
R9
R10
R3
R11
R21
R12
R4
R5
R13
R14
R19
R18
R20
R15
R16
R17
R2
R6
R7

插入（例）

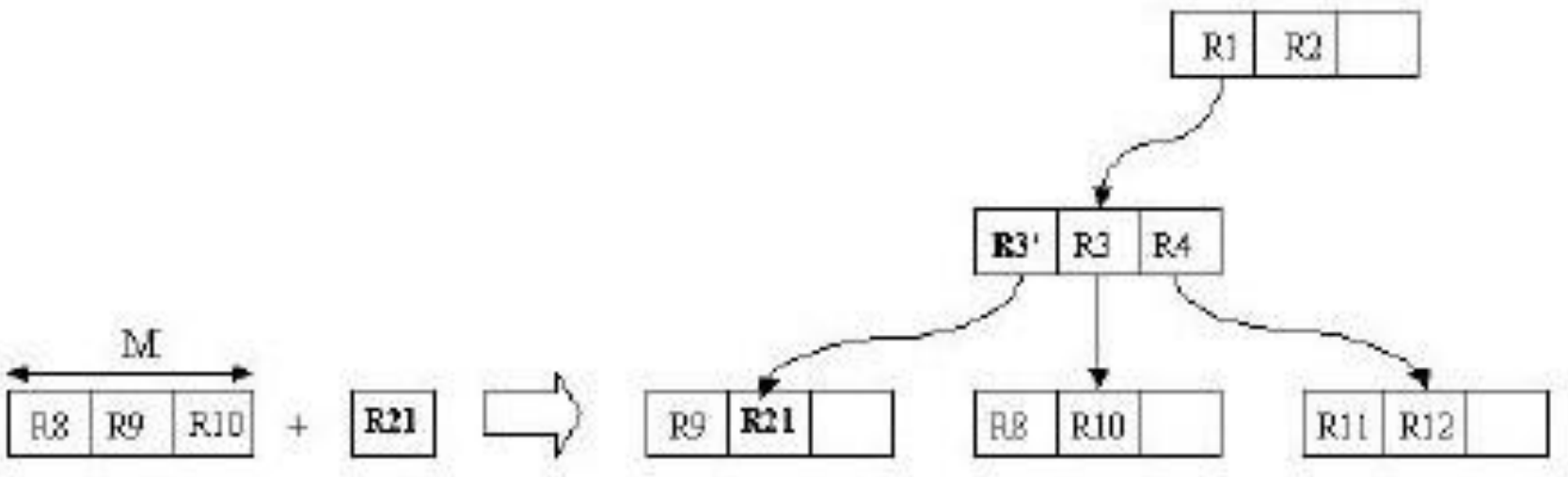

R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
R20

插入（例）

- 若插入R21这个矩形。首先进行ChooseLeaf操作。在根结点中有两个条目，分别为R1，R2。其实R1已经完全覆盖了R21，显然我们选择R1插入。然后进行下一级。相比于R4，向R3中添加R21会更合适，因为R3覆盖R21所需增大的面积相对较小。这样就在R8，R9，R10所在的叶子结点中插入R21。由于叶子结点没有足够空间，则要进行分裂操作。



插入（例）



删除

- R树的删除操作与B树的删除操作会有所不同，不过同B树一样，会涉及到压缩等操作。R树的删除同样是比较复杂的，需要用到一些辅助函数来完成整个操作。

Function： Delete

- 描述：将一条记录E从指定的R树中删除。
- D1：[找到含有记录的叶子结点] 使用FindLeaf方法找到包含有记录E的叶子结点L。如果搜索失败，则直接终止。
- D2：[删除记录] 将E从L中删除。
- D3：[传递记录] 对L使用CondenseTree操作
- D4：[缩减树] 当经过以上调整后，如果根结点只包含有一个孩子结点，则将这个唯一的孩子结点设为根结点。

Function： FindLeaf

- 描述：根结点为T，期望找到包含有记录E的叶子结点。
- FL1：[搜索子树] 如果T不是叶子结点，则检查每一条T中的条目F，找出与E所对应的矩形相重合的F（不必完全覆盖）。对于所有满足条件的F，对其指向的孩子结点进行FindLeaf操作，直到寻找到E或者所有条目均以被检查过。
- FL2：[搜索叶子结点以找到记录] 如果T是叶子结点，那么检查每一个条目是否有E存在，如果有则返回T。

Function： CondenseTree

- 描述：L为包含有被删除条目的叶子结点。如果L的条目数过少（小于要求的最小值m），则必须将该叶子结点L从树中删除。经过这一删除操作，L中的剩余条目必须重新插入树中。此操作将一直重复直至到达根结点。同样，调整在此修改树的过程所经过的路径上的所有结点对应的矩形大小。
- CT1：[初始化] 令N为L。初始化一个用于存储被删除结点包含的条目的链表Q。
- CT2：[找到父条目] 如果N为根结点，那么直接跳转至CT6。否则令P为N 的父结点，令EN为P结点中存储的指向N的条目。

Function： CondenseTree

- CT3：[删除下溢结点] 如果N含有条目数少于m，则从P中删除EN，并把结点N中的条目添加入链表Q中。
- CT4：[调整覆盖矩形] 如果N没有被删除，则调整EN.I使得其对应矩形能够恰好覆盖N中的所有条目所对应的矩形。
- CT5：[向上一层结点进行操作] 令N等于P，从CT2开始重复操作。
- CT6：[重新插入孤立的条目] 所有在Q中的结点中的条目需要被重新插入。原来属于叶子结点的条目可以使用Insert操作进行重新插入，而那些属于非叶子结点的条目必须插入删除之前所在层的结点，以确保它们所指向的子树还处于相同的层。

删除

- R树删除记录过程中的CondenseTree操作是不同于B树的。我们知道，B树删除过程中，如果出现结点的记录数少于半满（即下溢）的情况，则直接把这些记录与其他叶子的记录“融合”，也就是说两个相邻结点合并。然而R树却是直接重新插入。

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