**R树的操作**

这一部分也许是编程者最关注的问题了。这么高效的数据结构该如何去实现呢？这便是这一节需要阐述的问题。

**搜索**

R树的搜索操作很简单，跟B树上的搜索十分相似。它返回的结果是所有符合查找信息的记录条目。而输入是什么？就我个人的理解，输入不仅仅是一个范围了，它更可以看成是一个空间中的矩形。也就是说，我们输入的是一个搜索矩形。

先给出伪代码：

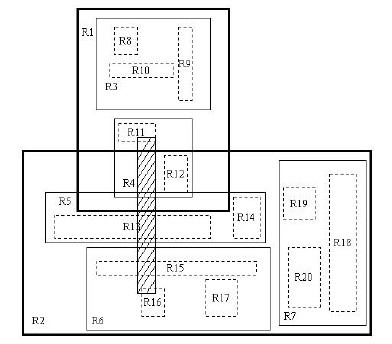
**Function：Search**

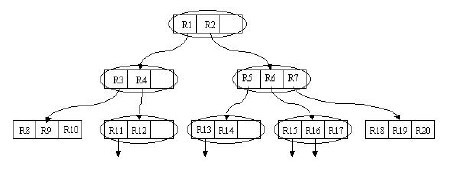
描述：假设T为一棵R树的根结点，查找所有搜索矩形S覆盖的记录条目。

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Search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。

我们通过下图来理解这个Search操作。





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Search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。

**插入**

      R树的插入操作也同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。

来看一下伪代码：

**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。

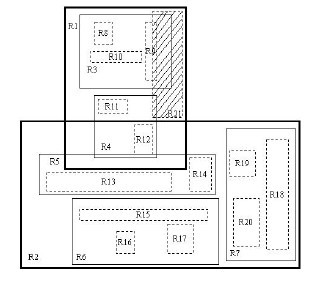
AT2：[检验是否完成] 如果N为根结点，则停止操作。

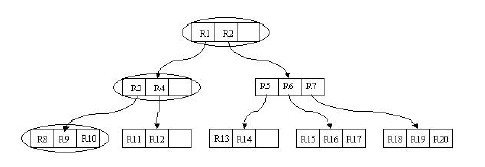
AT3：[调整父结点条目的最小边界矩形] 设P为N的父节点，EN为指向在父节点P中指向N的条目。调整EN.I以保证所有在N中的矩形都被恰好包围。

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。

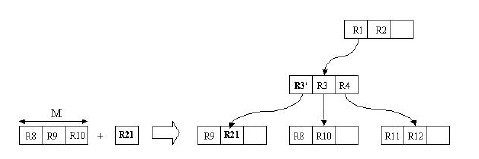
同样，我们用图来更加直观的理解这个插入操作。





    我们来通过图分析一下插入操作。现在我们需要插入R21这个矩形。开始时我们进行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，而若向R2中添加R21，则会使R2.I增大很多。显然我们选择R1插入。然后进行下一级的操作。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B8，B9，B10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。

    插入操作如下图所示：



这个插入操作其实类似于第一节中B树的插入操作，这里不再具体介绍，不过想必看过上面的伪代码大家应该也清楚了。

**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。相信读者看完以下的伪代码之后会有所体会。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。

伪代码如下：

**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。

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。

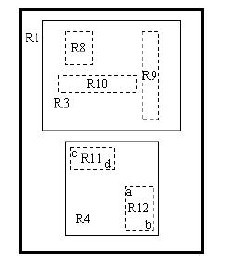
CT4：[调整覆盖矩形] 如果N没有被删除，则调整EN.I使得其对应矩形能够恰好覆盖N中的所有条目所对应的矩形。

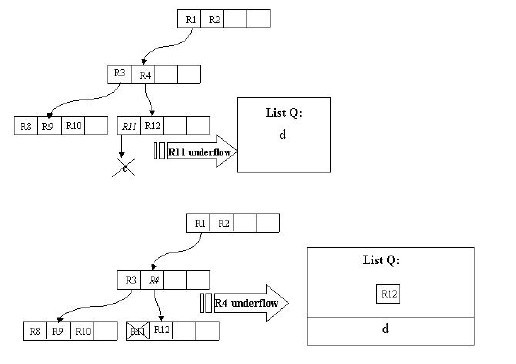
CT5：[向上一层结点进行操作] 令N等于P，从CT2开始重复操作。

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。

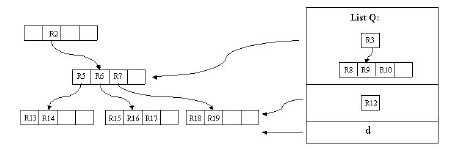
R树删除记录过程中的CondenseTree操作是不同于B树的。我们知道，B树删除过程中，如果出现结点的记录数少于半满（即下溢）的情况，则直接把这些记录与其他叶子的记录“融合”，也就是说两个相邻结点合并。然而R树却是直接重新插入。

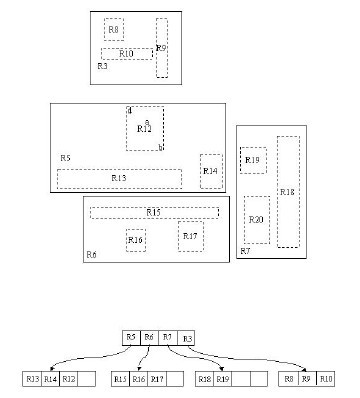
同样，我们用图直观的说明这个操作。





假设结点最大条目数为4，最小条目数为2。在这张图中，我们的目标是删除记录c。首先使用FindLeaf操作找到c所处在的叶子结点的位置——R11。当c从R11删除时，R11就只有一条记录了，少于最小条目数2，出现下溢，此时要调用CondenseTree操作。这样，c被删除，R11剩余的条目——指向记录d的指针——被插入链表Q。然后向更高一层的结点进行此操作。这样R12会被插入链表中。原理是一样的，在这里就不再赘述。





有一点需要解释的是，我们发现这个删除操作向上传递之后，根结点的条目R1也被插入了Q中，这样根结点只剩下了R2。别着急，重新插入操作会有效的解决这个问题。我们插入R3，R12，d至它原来所处的层。这样，我们发现根结点只有一个条目了，此时根据Inert中的操作，我们把这个根结点删除，它的孩子结点，即R5，R6，R7，R3所在的结点被置为根结点。至此，删除操作结束。