插入部分:传入参数为要插入的数据的值以及它的包围盒。

```
void Insert(leafdata,bound){
    if (没有根节点) {
            创建一个根节点空间并将其的值赋值为传入的参数值;
    }
    else {
        执行 insertInternal((leaf, bound), root);
    }
    ++树的大小
}
```

InsertInternal:插入功能的主体部分,传入的参数为 leaf(要插入的节点), node(当前的节点), firstInsert (是否为第一次插入)

```
Node * InsertInternal(Leaf * leaf, Node * node, bool firstInsert = true)
    {
        将当前节点拉伸至覆盖要插入的节点
        if (当前节点为叶结点) {
            直接插入
        }
        else {
            Node* temp_node = InsertInternal(leaf, ChooseSubtree(node, &leaf->bound),
firesInsert);
            //递归调用 InsertInternal 并接收向上溢出的节点
            if (!temp_node) {
                return NULL;
            }
            向当前节点中插入向上溢出的节点;
        }
        if (当前节点的大小大于最大节点大小的值) {
            return OverflowTreatment(node, firstInsert);
            //进行溢出处理并向上返回上溢的节点
        }
        return NULL;
    }
```

ChooseSubtree: 插入路径选择算法, 传入参数为当前节点以及所要查询的边界矩形。返回值为"几乎最小重叠矩形"。取前 32 个索引项。

```
Node* ChooseSubtree(Node *node,const BoundingBox* bound){
    if(当前节点的孩子节点指向叶结点){
        if (当前节点的孩子节点的个数大于 32) {
            对前 32 项根据增长面积进行排序;
            return  前 32 项中重叠面积最小的项;
        }
        else{
            return  所有项中的最小重叠面积项;
```

```
        }
    }
    else{
        return 所有项中增长面积最小的项;
    }
}
```

OverflowTreatment：插入溢出处理,传入参数为 level 当前溢出节点，以及是否为第一次插入（第一次调用 OverflowTreatment）

```
Node* OverflowTreatment(Node* level,bool firstInsert){
    if (当前节点不为根节点且不是第一次插入) {
            Reinsert(level);
            //对当前节点进行重插入;
            return NULL;
        }

        Node* splitIntem = Split(level);
        //得到分裂出来的节点
        if (当前节点为根节点) {
            创建一个新的根节点，并将分裂后的节点赋值给新的根节点，拉伸新的根节点的包围盒大小;
                return NULL;
        }
        return splitItem;
}
```

Reinsert:重插入，对溢出节点进行重插入

```
{
    定义 p;
    p = 判断节点的大小乘上 RTREE_REINSERT_P 是否大于 0？p*RTREE_REINSERT_P:1;
    //RTREE_REINSERT_P 优化重插入的算法，如果上面的判断条件返回 1 则证明 R*Tree 的约束节点大小空间很大，可以
    //选择多个节点进行重插入优化存储空间
    对前 M + 1 - p 项根据溢出节点的包围盒的中心距离排序;
    保存删除的后 p 项后将溢出节点的后 p 项删除;
    对后 p 项节点执行插入
}
```

Split：节点分裂函数，传入将分裂的节点，返回分裂后后半段的节点的值

主要分为 Split，ChooseSplitAxis 和 ChooseSplitIndex 三个部分

```
Node *Split(Node *node){
    定义一个 newnode 记录分裂出去的部分
        定义    split_axis ， split_margin ， split_edge ， split_index
        //分别代表分裂的维度，分裂后两个包围盒的长度和，排序的根据（上下界），分裂节点的位置
        for (从 0 到 k 维) {
            定义 dist_area, dist_overlap，dist_edge，  dist_index ;
```

```
            //分别代表当前维度下最优情况下的：面积和，重叠面积，排序方式，分裂节点位置
            for (维度下界到上界) {
                分别根据上下界排序;
                计算分裂后的两个包围盒的长度，面积以及重叠面积;
                    if (重叠面积小于 dist_overlap 或（等于 dist_overlap 并且面积小于 dist——area）) {
                        更新维护 dist_area, dist_overlap，dist_edge， dist_index 的值;
                    }
                }
            }
            if (当前维度下的包围盒的长度< split_margin) {
                更新维护 split_axis ， split_margin ， split_edge ， split_index
            }
        }
        根据得到的最优 split_edge 和 dist_index 将节点分为两个部分
        return 分出的部分;
}
```

删除部分：

```
Remove:传入参数为两个删除类别判别器
void Remove( const Acceptor &accept, LeafRemover leafRemover){
    定义一个接收重插入 items 的容器
    if(当前树为空){
        return;
    }
    RemoveFunctor<Acceptor, LeafRemover> remove(accept, leafRemover, &itemsToReinsert, &m_size);
    remove(m_root,true);
    //执行删除的仿函数
    if(重插入的 items 不为空){
        执行插入函数
    }
}

RemoveFunctor：伪函数
RemoveFunctor(当前节点，是否为根节点)
if(判别器判断为正){
    if(该节点拥有叶子节点){
        将对应的叶子节点删除;
    }
    else{
        对子节点递归调用该函数
```

```
    }
    if(不是根节点){
        if(该节点为空){
            删除该节点
        }
        else if(该节点的 size 小于最小 size){
            递归查找要重插入的节点;
        }
    }
    else if(该节点的子节点为空){
        hasLeaves = true;
        重置包围盒大小
    }
}
```

查询部分:

```
QueryFunctor:传入判别器仿函数和访问仿函数
{
    if(该节点拥有叶子){
        判别该叶子是否为想要访问的叶子并执行访问操作;
    }
    else{
        递归调用该函数
    }
}
```