R*Tree 算法实现

目录

实验内容

本实验实现了 R 树和 R*树的空间索引结构。它的优点在于支持多维区域查询，与 kd 树相比，它同时可以动态的增加删除数据，使用更为方便。R 树是利用多个维中的区间来表示数据对象的，它是一种高度平衡的树，其组成结构与 B 树十分类似，而 B 树只能进行一维数据查询，R 树可以对任意多维数据建立查询结构并提供查询功能，并且可以动态插入和删除节点。R 树由中间节点和叶节点组成，实际数据对象的最小外接矩形存储在叶节点中，中间节点通过聚集其低层节点的外接矩形形成，包含所有这些外接矩形。而在操作上则是基于在每一个内部节点中封闭矩形的面积的启发式优化。而 R*树则是 R 树的一个改进，在构造算法上，R*树不仅考虑了索引空间的“面积”，而且还考虑了索引空间的重叠。该方法对结点的插入、分裂算法进行了改进，并采用“强制重新插入”的方法使树的结构得到优化。

本次实验，我们不仅实现了 R 树和 R*树，并提供了其对应的数据接口。在指定数据的矩形区域的前提下，用户可以通过继承 RSTNode 类，自行指定所插入数据类型，包括点、矩形等等，并且，所插入的数据可以是任意维度，这就是所谓的多维空间搜索。同时支持动态的增加、删除数据，提高操作上的便利性。

算法原理

R 树算法原理

R 树基本结构

R 树是高度平衡的树，在其叶节点的索引记录中包含指向数据对象的指针。对于 R 树的叶子节点，其包含索引记录的条目，基本形式为(I, 元组标示符)，其中元祖标示符指向对应的数据，而 I 则是一个包含在一个边界盒内的空间对象的 n 维矩形，表示为：

$$I = (I_0, I_1, ...I_{n-1})$$

其中 n 是指维度的个数，I_i 是一个封闭的有界区间[a, b]，用来描述在维度 i 上的空间对象的范围，I_i 可以有一个或两个边界都是无穷大的，表面对象是无穷大的。R 树的非叶子节点则包含了形如(I, child-pointer)的条目，这里 child-pointer 是一个低级节点在 R 树中的地址，而 I 覆盖了所有低级节点条目中的矩形。简单来讲，即每个节点包含了多个子节点或数据（当节点为叶子时），而节点中又包含了多维矩形 I 表示所有子节点或数据的最小包围矩形。

简单的一个 R 树结构如下图：

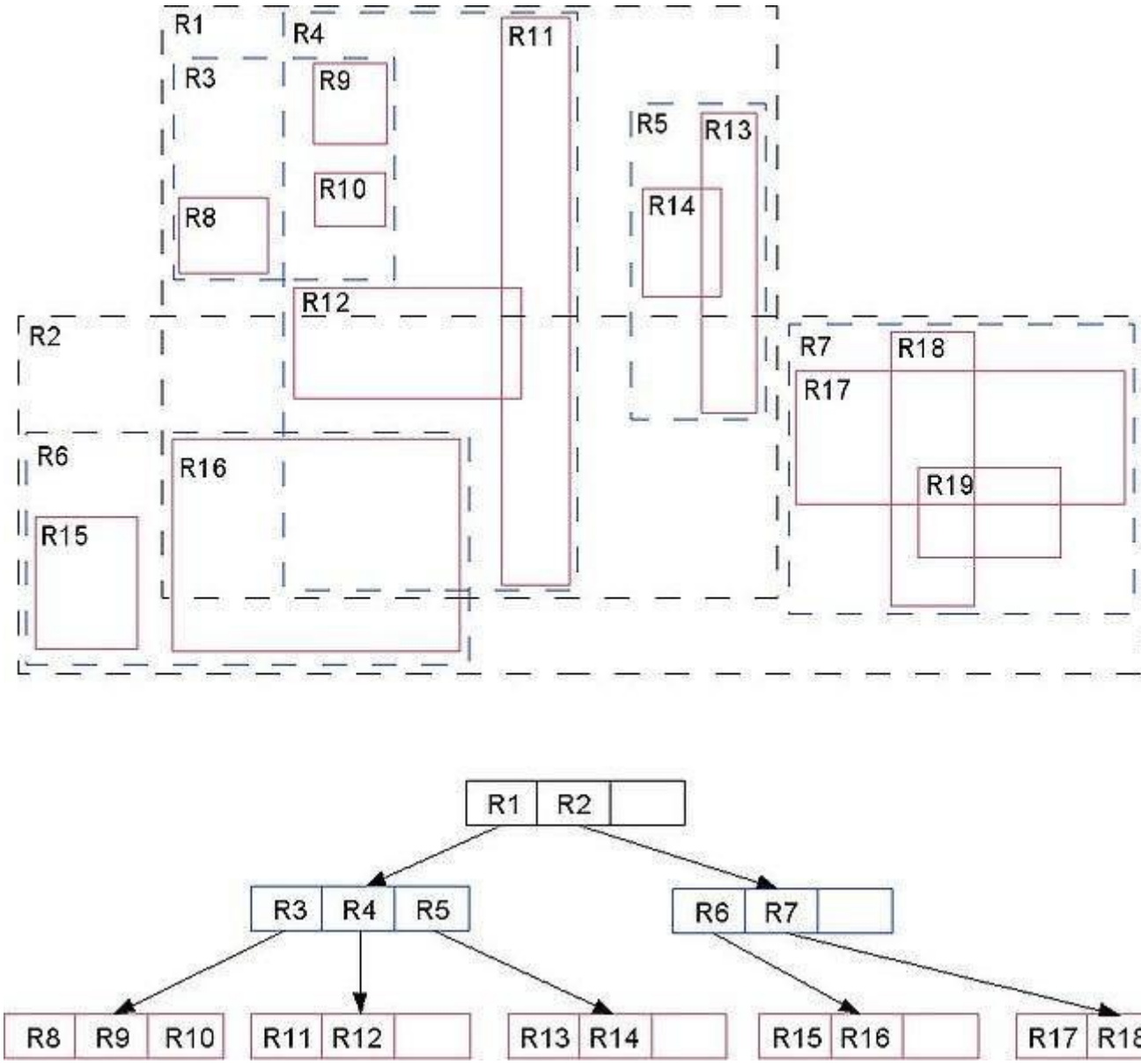


图 1

R 树性质

R 树有两个重要属性：M 和 m。其中 M 表示一个节点中条目的最大数量，而 m 小于等于 M/2，表示一个节点中条目的最小数量。一个 R 树则具有下列性质：

1、每个叶节点若不是根节点，则包含 m 至 M 个索引记录。

2、叶节点中的每一个索引记录(I, 元组标示符)，I 是在空间上包含 n 维数据对象的最小的矩形，该对象是由相应的元组给出的。

3、每个非叶节点若不是根节点则有 m 至 M 个子节点。

4、对于一个非叶节点中的条目(I, child-pointer)，I 是在空间上包含在子节点中的矩形的最小矩形。

5、根节点若果不是一个叶子，则至少有两个子节点。

6、所有的叶子都位于同一层上

图 1 给出了 R 树的结构，并举例说明了其矩形间的包含及覆盖关系。

含有 N 挑索引记录的 R 树的高度最大是$|\log_2 m^N|-1$，这是因为每个节点的分支至少是 m。所以节点的最大数目就是$|N/m|+|N/m^2|+...+1$，出了根节点外，所有节点都是 m 时，空间的利用是最差的。节点含有多于 m 个条目时，将降低树的高度，提高空间利用率。如果节点有多于 3 个或 4 个条目的话，树就会变得很宽，几乎所有的空间多用于包含索引记录的叶节点。参数 m 可以调整，取不同的值。

R 树算法

这里用 EI 表示索引条目 E 的矩形，用 EP 表示元组标识符或 child-pointer。

1、搜索算法 Search

给定一棵 R 树，其根节点是 T，输入参数为需要搜索的矩形 S，找出其矩形覆盖 S 的所有索引记录。

- [搜索子树]如果 T 不是叶子，检查每一个条目 E，判断是否 EI 与 S 相交。对于所有相交的条目，在由 EP 指向的子树的根节点上调用 Search
- [搜索叶节点]如果 T 是一个叶子，检查所有 EI 判断是否覆盖 S，若是，则 E 就是一个呵护要求的记录

2、插入算法 Insert

把一个新的索引条目 E 插入一个 R 树中。

- [找到新纪录的位置]调用 Choose Leaf 选择一个叶节点 L 存放 E。
- [把记录加入到叶节点中]如果 L 有空间存放额外的条目，加入 E；否则调用 Split Node 以获得包含 E 及所有原来 L 的条目 L 及 LL。
- [向上传递变化]在 L 上调用 Adjust Tree，若完成了分裂，则也调整了 LL。
- [把树变高]如果节点分裂导致根节点的分裂，则生成一个新的根节点，其子节点为两个已有的节点

3、算法 Choose Leaf

选择一个叶节点来存放一个新的索引条目 E。

- [初始化]设 N 为根节点。
- [检查叶子]如果 N 是叶子，返回 N。
- [选择子树]如果 N 不是叶子，设 F 为 N 中条目，它的矩形 FI 需要至少放大到包含了 EI。通过选择有最小区域的矩形的条目来重新连接。
- [向下进行直至到达一个叶子]设 N 为由 FP 指向的一个子节点，并从第 2 步处重复此过程。

4、算法 Adjust Tree

从一个叶节点 L 点上升到根，调整覆盖的矩形，需要则传递分裂。

- [初始化]令 N=L，如果 L 前面分裂过，设 NN 为所得的第二个节点。
- [检查是否完成]如果 N 为根，则停止。
- [调整在父条目中覆盖的矩形]令 P 作为 N 的父节点，令 EN 作为 N 在 P 中的条目，调整 ENI，使其紧密地围住 N 中全部的条目矩形。
- [向上传递节点分裂]如果 N 有一个伙伴——从早先分裂中得到的 NN，则生成一个新的条目 ENN，并用 ENNP 指向 NN，而 ENNI 围住所有 NN 中的

矩形。若还有空间，则把 ENN 加入 P 中。否则调用 Split Node 来生成 P、PP、ENN 及 P 中的所有条目。

- [移动到下一层]令 N=P，如果出现一个分裂令 NN=PP，从第 2 步重复。

5、算法 Delete

从 R 树中删除索引记录 E。

- [找到包含记录的节点]调用 Find Leaf 找到包含 E 的叶节点 L，若果没有找到记录则停止。
- [删除记录]从 L 中删除 E。
- [传递变化]调用 Condense Tree，经过 L。
- [降低书]如果根节点在经过树的调整之后仅有一个子节点，将这个子节点作为新的根节点。

6、算法 Find Leaf

给定一棵 R 树，其根节点为 T，找出所有包含索引条目 E 的叶节点。

- [搜索子树]如果 T 不是叶子，检查 T 中的每个条目 F，判断是否 FI 覆盖了 EI。针对每个这样的条目，在由 FP 指向的树中的根节点上调用 Find Leaf，直至 E 被找到或者所有的条目都被检查过。
- [搜索记录的叶节点]如果 T 是一个叶子，检查每个条目，看其是否与 E 匹配，若找到 E 则返回 T。

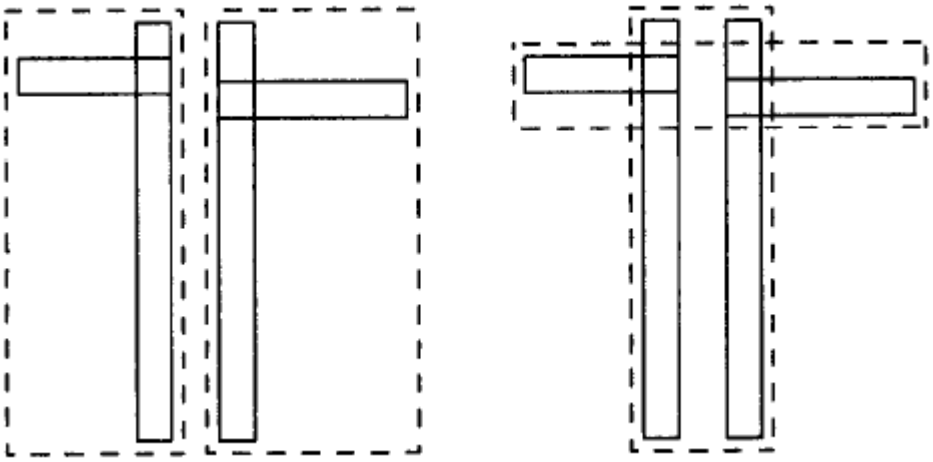
7、算法 Condense Tree

给定一个叶节点，其中已删除一个条目，如果它有很少的条目，则消除节点，并转移其条目。若有需要，则向上传递的节点消除。调整到达根的路径上所有覆盖的矩形，如果可能，则使他们变得更小。

- [开始]令 N=L，设 Q 为一组消除的节点，置为空。
- [找到父条目]如果 N 为根，转移到第 6 步，否则令 P 为 N 的父节点，令 EN 为 N 在 P 中的条目。
- [消除不饱和的节点]若 N 中有少于 m 个条目，从 P 中删除 EN，把 N 加入到 Q 中。
- [调整覆盖的矩形]若 N 未被消除，调整 ENI，使其包含 N 中的所有条目。
- [向上移动一层]令 N=P，从第 2 步重复进行。
- [重新插入孤立的条目]重新插入所有 Q 中节点的条目。将消除的叶节点中的条目重新插入到树的叶节点上，插入算法见算法 Insert。但在高一层的节点上条目必须放置在树的高层上，这样它们独立的子树上的叶子可以在同一层上，就像主树上的叶子一样。

8、节点分裂

为了在一个以包含 M 个条目的已满的节点中加入一个新的条目，把 M+1 个条目的集合分成两个节点是必须的。这个分配应按下述方法进行，尽量使两个新节点在接下来的搜索检查中不同时出现。因为访问一个节点取决于其覆盖的矩形所覆盖的搜索面积。两个覆盖矩形的总面积在一个分裂之后应为最小。图 2 举例说明了这点。不良分裂的覆盖矩形面积要比最佳情况下的面积大的多。



较差分裂方式　　较好分裂方式

图 2

9、二次方代价算法

这个算法是为了找到一个最小面积的分裂，但无法保证一定会找到这个最小面积。代价是 M 的二次方并且与维数成线性关系。算法首先从 M+1 个条目中选出两个条目作为两个新组中的第一个成员，选择这两个条目的方法是若两个条目放在同一组内将浪费的面积最大，即覆盖了两个条目的矩形面积减去两个条目的面积最大的。剩余的条目每次分配就是使两个差别最大的那个条目。

10、算法 Quadratic Split

把 M+1 个索引条目分成两组。

- [从每个组中取第一个条目]调用算法 Pick Seeds 选出两个条目作为两组中的第一个成员，分配到组中。
- [检查是否结束]若所有的条目都被分配，则停止；若一组中条目很少，则剩余的条目必须分配到这组中，来保证其条目数量达到最小值 m，分配之后停止。
- [选择待分配的条目]调用算法 Pick Next 选择下一个待分配的条目。把它加入到所覆盖矩形在进行最小扩展就可容纳它的组中。将条目加入到哪个组中的方案为：首先考虑面积较小的组，其次是条目较少的组，最后是其他条件。从第 2 步重复。

11、算法 Pick Seeds

选择两个条目作为组中的新成员。

- [计算两个条目的面积对应值]对每一对条目 E1 及 E2 组成一个包括 E1I 及 E2I 的矩形 J，计算 d=area(J)-area(E1I)-are(E2I)。
- [选择浪费最大的对]选择 d 最大的一对条目返回。

12、算法 Pick Next

从余下的条目中选出一个防盗组中。

- [判断把一个条目放入一个组中的代价]对还未放入组中的每个条目 E，计算 d1=第一组包含 EI 后覆盖矩形增加的买年纪，类似地计算第二组 d2。

- [找到对于每个组的最佳条目]选择 d1 与 d2 最大差别的条目。

R*树算法原理

R*树是对 R 树的改进，在稍微增加建树时间的基础上，大大提高了查询效率。其主要在以下三个部分对 R 树进行了改进。

Choose Subtree 算法

为了解决选择一个合适的插入路径的问题，R 树的描述中仅考虑面积参数。在 R*树的描述中将对面积、边缘及覆盖组合起来考虑，其中一个条目的覆盖定义如下：

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

下面是具有最佳查询性能的算法。

Choose Subtree 算法：

- **令 N 为根节点。**
- **if N 是一个叶子。**

  return N

  else

  if N 中子节点指针不指向叶子，[确定最小面积耗费]选择 N 中的条目，该条目需要最小的面积扩展以包含新数据的矩形，选择其矩形需要的最小区域扩展的条目来继续下一步查询。

  then 选择条目具有最小面积的矩形。

  if N 中的孩子的指针不指向叶子，[确定最小覆盖代价]选择 N 中的条目，该条目需要最小的覆盖扩展以包含新数据的矩形，选择其矩形需要的最小覆盖扩展来继续查询数据。
- **设 N 为所选择出的条目的子指针所指向的子节点，转到第 2 步重复执行**

对于选择最佳非叶节点，R*树的算法并不优于 Guttman 的原始算法。然而对于叶节点，使用覆盖最小扩展在性能方面则会更好一些。

在该算法中，决定覆盖的 CPU 的运算量是条目数目的平方量级，因为对于每个条目，与节点中其他条目的覆盖均是需要计算的，但对于大容量的节点，可以减少必须进行计算的条目数量。可以这样认为，对于距离较远的矩形，产生最小覆盖的可能性极小，因此为了降低 CPU 的运算量，这部分算法可以修改成如下算法：

[确定最近似的最小覆盖算法]

- **将 N 中的矩形排序，按其需要包含新数据的矩形面积扩展量的升序排列。**
- **设 A 代表排序后前 p 个条目，仅对 A 中的条目进行处理，来代替处理 N 中的所有条目，选择其矩形需要最小覆盖扩展的条目来进行查询。如果出现相等情况，则选择扩展矩形较小的节点继续查询。**

在二维的情况下，把 p 设为 32 通常就不会降低查询操作的性能。

R*树的分裂

算法 Split

输入：要分裂的节点的指针

输出：指向分裂后两个节点的指针

Step.1 调用 ChooseSplitAxis 算法选择要分裂的坐标轴（维度）splitAxis

Step.2 在选择好的维度上调用 ChooseSplitIndex 算法，算法结束后返回两个值，

bool minOrMax:标识最终的分裂是在哪个排序结果上

int splitIndex:将某个排序进行分裂的分界点，对与由 minOrMax 所确定的

算法 ChooseSplitAxis

输入：要分裂的节点

输出：要分裂的维度，范围为 0～d-1，其中 d 为 R 树中数据的维度。

Step.1 将当前维度置为 0，将最佳分裂维度设置为 0，转到 Step.2

Step.2 如果当前维度已经超过 d-1，返回当前的最佳分裂维度.

否则，在当前维度上对分裂节点的所有子节点做两组排序

第一组排序根据当前维度区间的下届，从小到大

第二组排序根据当前维度区间的上界，从小到大

对于两组排序中的任何一组，我们都可以将其分裂两个部分，第一个部分包括前 m-1+k 个子节点，另外一部分 M+2-m-k 个子节点。其中 k 的范围是 1 到 M-2×m+2。

对于两组排序，都可以按以上方案分裂成两个节点，S1 和 S2，对于全部两个排序下的全部分裂方案，都计算 S1 和 S2 的 margin 之和 margin(S1)+margin(S2)，并将其累加。

margin 指的是各个维度的区间长度之和。

对于当前维度，我们都可以得到一个 margin 的总和，如果该总和与以前维度所计算出来的总和相比更小，则将当前维度设置为最佳分裂维度。

将当前维度加 1，重复执行 Step2.

算法 ChooseSplitIndex

输入：要分裂的节点和最佳分裂维度

输出：最佳分裂排序，最佳分裂点

算法过程：

在最佳分裂维度上，对分裂节点的所有子节点做两组排序

第一组排序根据当前维度区间的下届，从小到大

第二组排序根据当前维度区间的上界，从小到大

对于两组排序中的任何一组，我们都可以将其分裂两个部分，第一个部分包括前 m-1+k 个子节点，另外一部分 M+2-m-k 个子节点。其中 k 的范围是 1 到 M-2×m+2。

对两组排序，都可以按以上方案分裂成两个节点 S1 和 S2，计算 S1 和 S2 区域的重叠部分的面积（如果是 2 维的话）。

对于所有的分裂方案，选择重叠面积最小的分裂方案，如果发生重叠面积相同的情况，则选择 S1 和 S2 面积之和(如果是 2 维的话)最小的一个，如果两个分裂方案面积之和也出现相 同情况，可以选择两种方案中的任何一个。

对于最佳分裂方案，返回其所在的排序分组以及分裂点。

强迫重插入

在分配条目入节点时，R 树和 R*树都是不确定的，既不同的插入会建成不同的树。因为这个原因，R 树受制于它的旧条目。数据矩形插入在早期的结构生成时会产生固定的矩形，其不能保证当前情况下好的查询操作。一个对有限制性的固定矩形的调整是在一个分裂过程中执行的，但这一过程效率不高，因此我们需要一个更有力且少限制性的调整方法。

为了得到一个有力的调整，R*树在常规插入时，进行强迫重插入操作。下列的算法即是基于树的同层插入记录的常规插入及删除算法，它与原始的 Guttman 算法很相似，但这只是一个简单草案。

1、 Insert Data 算法

- **调用 insert 作为参数，从叶子层开始，插入一个新的数据矩形。**

2、 Insert 算法

- **调用 Choose Subtree，以层作为参数，找到一个合适的点 N，插入新条目 E。**
- **如果 N 中有少于 M 条条目，把 E 插入 N，如果 N 中有 M 条条目，以 N 层作为参数，调用 Overflow Treatment[重插入或分割操作]。**
- **如果 Overflow Treatment 被调用，而且执行了一次分裂，如果需要，把 Overflow Treatment 向上传递，如果 Overflow Treatment 导致根节点的分裂，就重建一个根节点，树高度加一。**
- **调节所有在插入路径上被覆盖的矩形，这一，它们的矩形就是最小的封闭它们孩子们的矩形。**

3、 Overflow Treatment 算法

- **在插入一个数据矩形的过程中，如果不是根节点层，且是在给定层的第一次调用该算法，则调用 Re Insert，否则调用 Split 算法。**

4、 Re Insert 算法

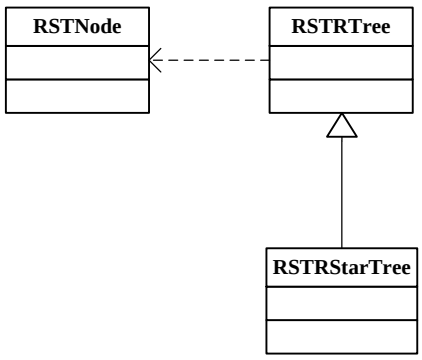
- **对于一个节点 N 中的所有 M+1 条条目，分别计算它们矩形中心到 N 的矩形中心的距离。**
- **按距离降序排列这些节点。**
- **去掉 N 中的前 p 条条目，调整 N 的矩形大小。**
- **按第 2 步的顺序，以距离最大值开始调用 Insert 算法来重新插入 P 条条目。（这里同样可以按距离最小值插入，删除掉 P 个距离最小的节点，然后从小到大插入）**

可以总结如下：

1) 强迫重插入在相邻节点间改变条目，因此降低了重叠。
2) 作为一方面的影响，提高了存储利用率。
3) 对于更多的调整，分裂较少。
4) 一个节点的远离中心的矩形被重新插入，节点矩形区域会减小，更加利于查询。

程序框架

类图



数据结构

区间结构

```
struct RSTInter
{
public:
    double min;// 区间下界
    double max;// 区间上界
}
```

多维区间集合

```
typedef std::vector<RSTInter> RSTRange;// 表示多维区间，为区间的集合，可以用于表示高维矩形
```

节点定义

```
class RSTNode
{
public:
    int type;// 节点类型，包括叶子节点、非叶子节点和数据节点
    RSTNode* parent;// 父节点指针
    RSTRange range;// 记录矩形区域
    int childNum;// 子节点个数
    RSTNode** childSet;// 子节点指针，为指针动态数组
}
```

节点集合

```
typedef vector<RSTNode*> RSTNodeSet; // 节点集合，用于记录一系列节点
```

点定义

```
typedef vector<double> RSTCPoint;// 点定义，用于记录多维点
```

***R* 树结构**

```
class RSTRTree
{
public:
    int height;// 树的高度，从跟开始到叶子节点的高度，只有一个根时高度为
    int dim;// 树的维度
    RSTNode* Root;// 树的根节点

    int m;// 子节点的最小数目
    int M;// 子节点的最大数目
}
```

R*树结构

```
class RSTRStarTree : public RSTRTree// 继承了 R 树结构
{
}
```

文件说明

主要的算法文件包括以下一些文件：

1、 RSTDefine.h，RSTBasisFunction.cpp，定义了一些基本数据结构，包括节点结构、区间结构等，实现了节点和区间的基本操作。同时也包括其他的一些常量值、集合的定义和相关方法的实现。

2、 RSTRTree.h，RSTRTree.cpp，定义了 R 树和 R*树的算法实现，包括树结构定义和具体的算法实现。

3、 RSTTestData.h，RSTTestData.cpp，定义了一些用于测试的数据结构，包括二维点、矩形及多维点、矩形。

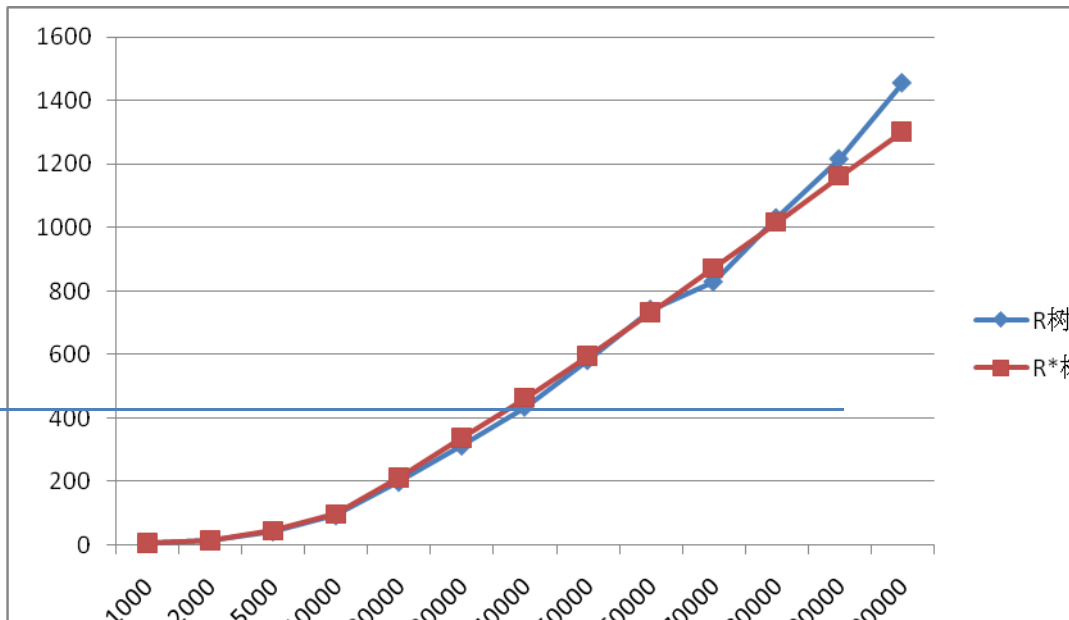
实验结果

建树时间比较

下面给出了 m 与 M 分别取不同值时的建树时间，其中数据均为二维点形式。

表 1 m=2，M=4 时 R 树与 R*树建树时间比较

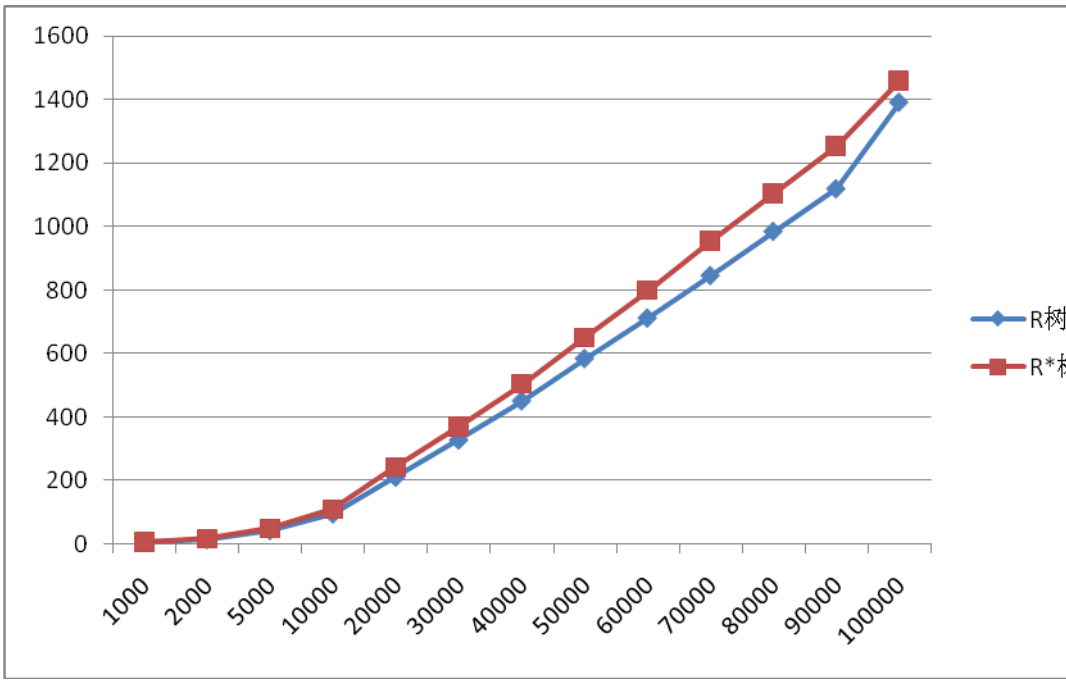
	R 树 (ms)	R*树 (ms)
1000	6.69771	7.83333
2000	15.3371	16.8487
5000	43.3573	46.3984
10000	93.5264	99.454



20000	199.157	214.922
30000	313.626	338.701
40000	435.331	465.45
50000	582.49	597.271
60000	742.974	733.777
70000	830.35	873.199
80000	1030.94	1017.13
90000	1216.56	1162.56
100000	1455.96	1302.14

表 2 m=4，M=8 时 R 树与 R*树建树时间比较

	R 树 (ms)	R*树 (ms)
1000	7.05016	8.4434
2000	15.6818	18.534
5000	43.8957	51.5164
10000	96.1912	112.405
20000	211.291	244.551
30000	328.643	370.751
40000	451.479	506.056
50000	584.496	652.28
60000	711.931	800.231
70000	845.493	954.702
80000	984.106	1104.65



90000	1118.18	1252.91
100000	1389.89	1460.06

表 3 m=10，M=20 时 R 树与 R*树建树时间比较

	R 树 (ms)	R*树 (ms)
1000	9.29371	18.3244
2000	21.5018	41.019
5000	61.2638	115.863
10000	135.278	242.296
20000	289.849	505.993
30000	462.038	787.036
40000	638.364	1073.86
50000	848.286	1374
60000	1006.24	1689.08
70000	1202.11	1978.7
80000	1387.98	2286.31
90000	1588.58	2575.39
100000	1793.46	2886.9

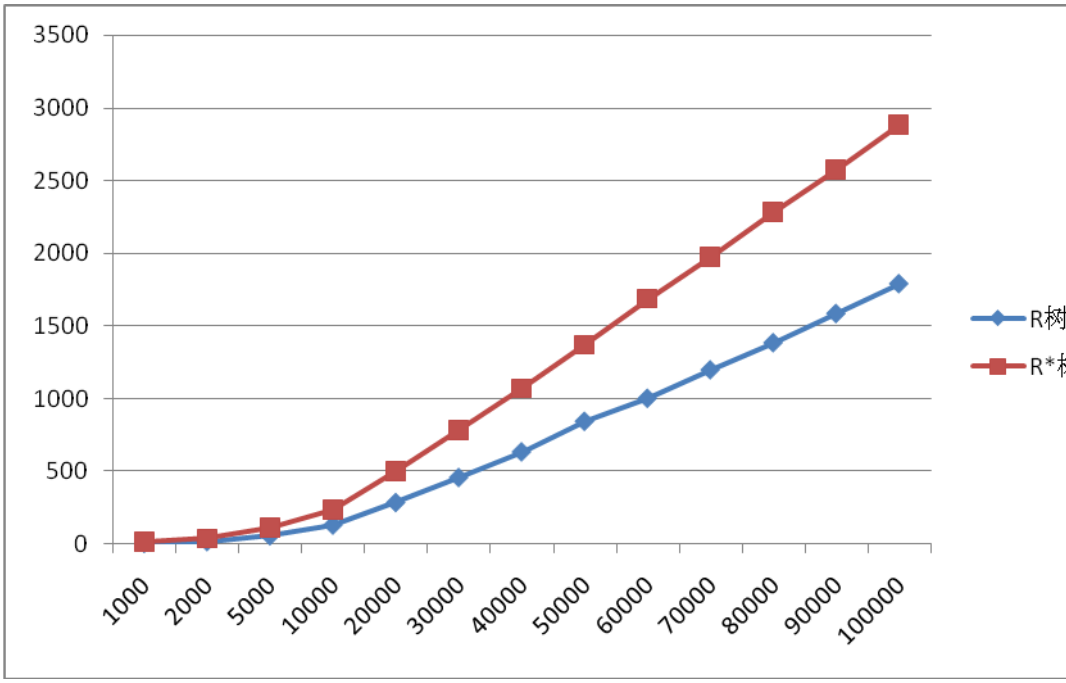
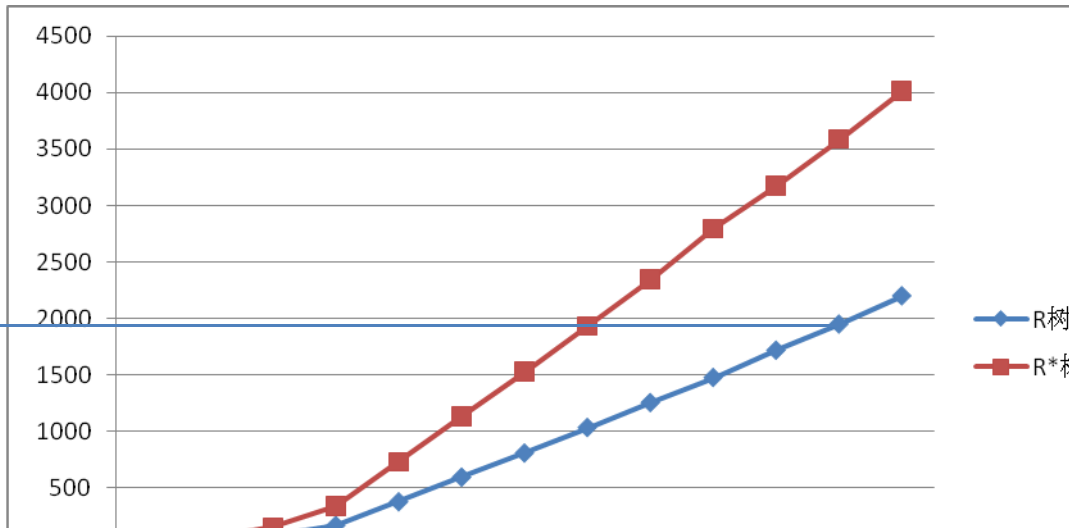


表 4 m=12，M=30 时 R 树与 R*树建树时间比较

	R 树 (ms)	R*树 (ms)
1000	12.5704	23.7222
2000	26.9295	54.4829
5000	77.039	156.8



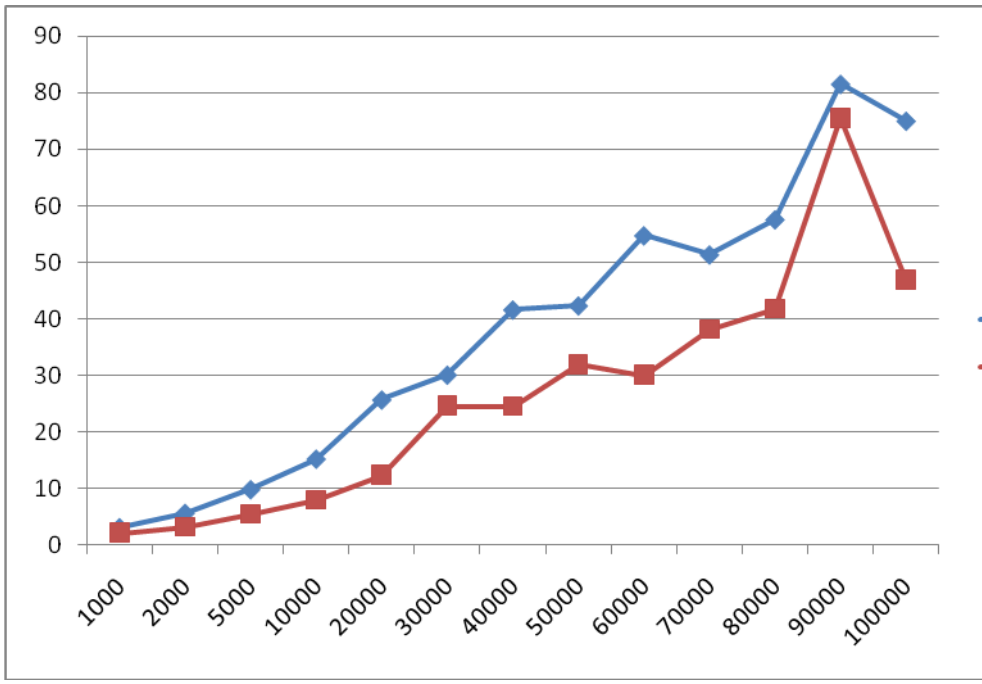
10000	173.8	342.036
20000	381.795	734.517
30000	599.956	1137.8
40000	812.804	1535.91
50000	1038.28	1937.43
60000	1258.98	2349.78
70000	1480.74	2798.74
80000	1721.78	3178.42
90000	1952.91	3590.32
100000	2202.45	4017.48

查询时间比较

下面给出了 m 和 M 取不同值时的 R 树和 R*树的查询时间比较。数据为二维的点。在给定点数的情况下，每次进行十次随机查询，表格中的数据为这十次查询时间的平均值。

表 5 m=2，M=4 时 R 树与 R*树查询时间比较

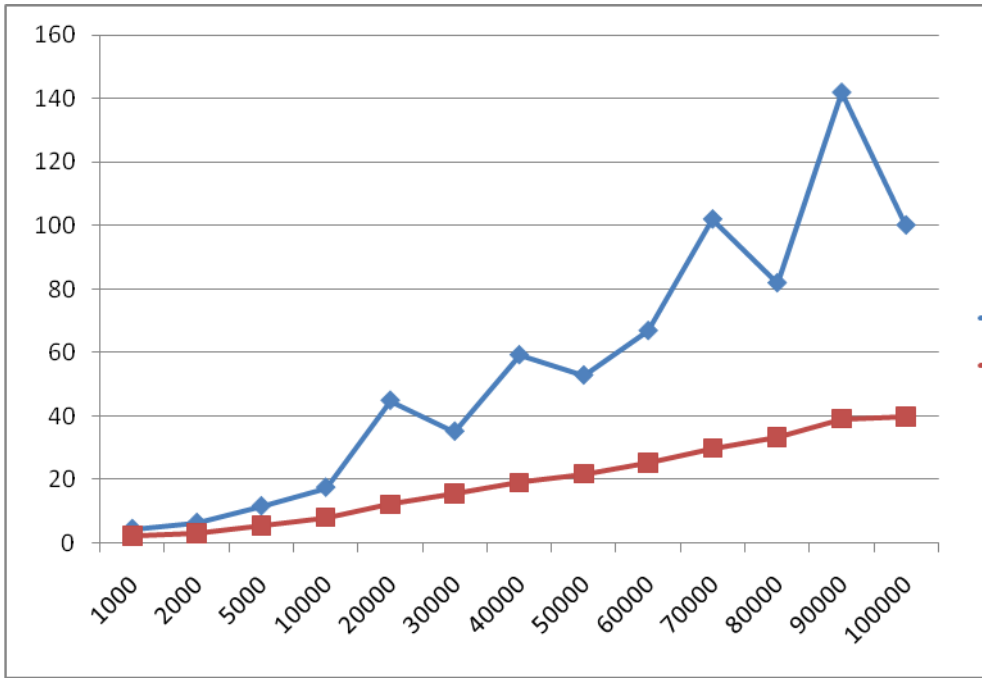
	R 树(μs)	R*树(μs)	R 树时间/R*树时间
1000	3.180	2.12	66.78%
2000	5.71	3.18	55.77%
5000	9.88	5.42	54.83%
10000	15.2	7.95	52.02%
20000	25.7	12.3	47.90%
30000	30.1	24.6	81.65%
400	41.6	24.5	58.87



00	589	272	62%
500	42.3	31.9	75.44
00	727	669	22%
600	54.7	30.0	54.86
00	681	489	57%
700	51.3	38.1	74.18
00	836	213	96%
800	57.5	41.7	72.52
00	985	709	08%
900	81.5	75.5	92.65
00	348	499	98%
100	75.0	46.9	62.57
000	183	442	69%

表 6 m=4，M=8 时 R 树与 R*树查询时间比较

	R 树 (μs)	**R* 树 (μs)**	**R 树时间 /R*树时间**
100	4.61	2.20	47.90
0	233	939	18%
200	6.53	3.14	48.09
0	103	132	84%
500	11.7	5.43	46.17
0	709	567	87%
100	17.6	8.06	45.76
00	121	045	65%
200	45.0	12.2	27.11
00	036	006	02%
300	35.2	15.6	44.52
00	587	987	43%
400	59.3	19.0	32.14
00	114	641	23%
500	52.9	21.7	41.10
00	205	509	11%
600	66.9	25.3	37.88
00	138	486	24%
700	101.	29.8	29.31
00	977	922	26%
800	81.9	33.3	40.64
00	403	026	26%
900	141.	39.2	27.64
00	829	074	41%
100	100.	39.8	39.79



000	115	379	23%

表 7 m=10，M=20 时 R 树与 R*树查询时间比较

	R 树 (μs)	R* 树 (μs)	R 树时间 /R*树时间
1000	6.05398	2.78081	45.93 36%
2000	8.50702	4.34846	51.11 62%
5000	14.0938	7.53976	53.49 69%
10000	25.6896	11.2814	43.91 44%
20000	35.3417	15.3871	43.53 81%
30000	42.2973	18.4817	43.69 48%
40000	48.6164	25.4482	52.34 49%
50000	62.3192	24.8921	39.94 29%
60000	191.243	42.1867	22.05 92%
70000	83.6408	32.9463	39.39 02%
80000	115.555	52.8656	45.74 94%
90000	97.4421	38.6812	39.69 66%
100000	103.204	45.5524	44.13 81%

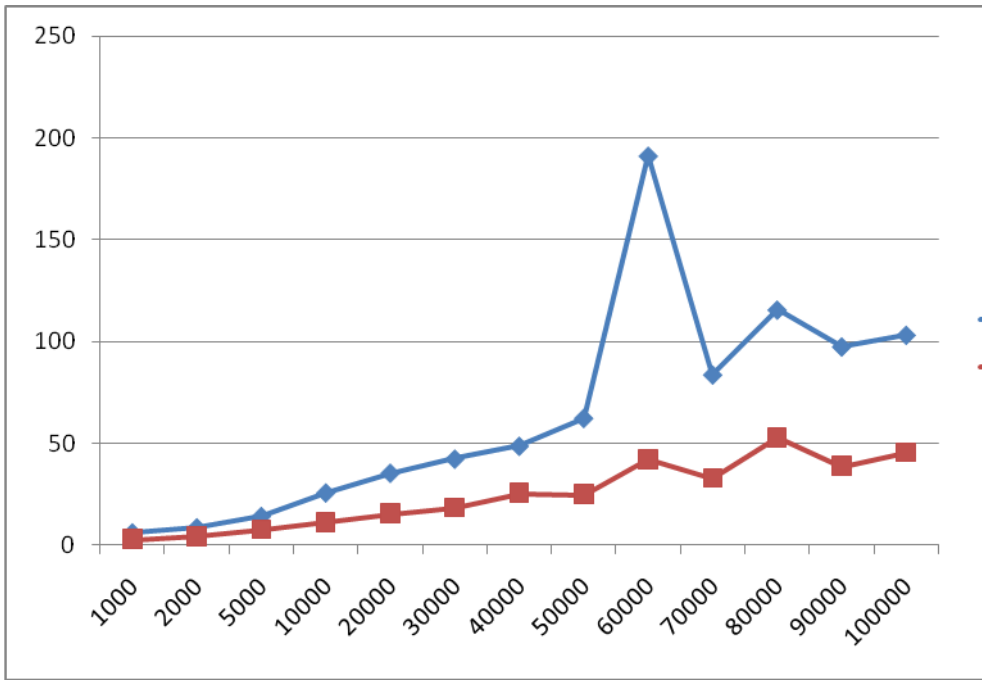
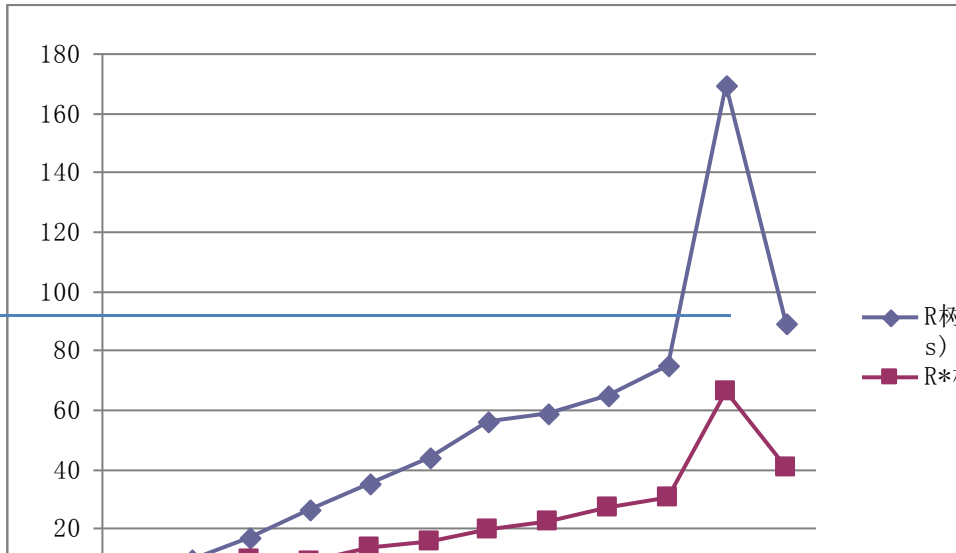


表 8 m=12，M=30 时 R 树与 R*树查询时间比较

	R 树 (μs)	R*树 (μs)	R 树时间 /R*树时间
1000	5.51727	2.72252	49.345 5%
2000	9.40701	3.91463	41.614 %



5000	16.6946	9.43709	56.5277%
10000	26.4103	8.83523	33.4538%
20000	34.9172	13.5134	38.7012%
30000	43.7916	15.7201	35.8975%
40000	55.7035	19.6915	35.3506%
50000	58.9305	22.2614	37.7757%
60000	64.5187	27.2327	42.209%
70000	74.5303	30.1764	40.4887%
80000	169.253	65.7575	38.8516%
90000	89.12	40.5601	45.5118%
100000	100.978	46.7032	46.2508%

显示说明

我们的 R 树和 R*树理论上可以应用于任意维度的数据，但是多维数据的可视化比较抽象和难以看懂，这里只实现了对二维数据的三维显示。

目前支持显示的数据类型有二维点和二维矩形，如果用户想显示自己定义的二维数据，需要继承 RSTNode 这个类，并实现几个必要的函数，具体可以参考“RSTTestData.h”中 RSTPoint2D 类的写法。

与显示相关的类主要有两个：一个是 MyGLShow3D 类，封装了最底层的 OpenGL 视图控制操作，这个类与数据无关，可以移植到其他希望通过 MFC 框架实现 OpenGL 三维显示的工程中。另一个类是 Tree2DShow3D 类，这个类封装了显示二维 R 和 R*树所需的辅助变量和操作，它是在 R 树的上层建立起来的，不会对 R 树的实现和应用产生任何影响，如果用户希望单独使用 R 树，可以将 R 树部分的代码单独分离出来使用。

总结

本次实验主要进行了一种多维查找结构 R-Tree 以及其重要变种 R*Tree 的实现以及测试。R-Tree 不仅在计算几何上有所应用，在空间数据库索引和 GIS 系统中也有着广泛的应用，一方面，该结构吸收了 B 树的优点，另一方面，有弥补了 k-d 树和 Range 树静态建树而不能动态插入和删除节点的不足。

本次实验针对二维点数据和二维矩形进行了 R 树和 R*树的演示，演示内容包括树的直观表示，查找过程的进行等等，并提供手动加入二维点数据或矩形数据以及删除数据等功能，为 R 树的用户提供了 R 树的直观表示，加深其对 R 树的理解和使用。对于任意维度的数据，我们规范了 R 树的使用接口，实现了数据处理与用户数据实现相分离。

另外对 R 树和 R*树的性能我们在实验中也做了充分测试，实验结果表明虽然 R*的插入过程和节点分裂过程更加复杂，但其查询效率明显高于 R 树。

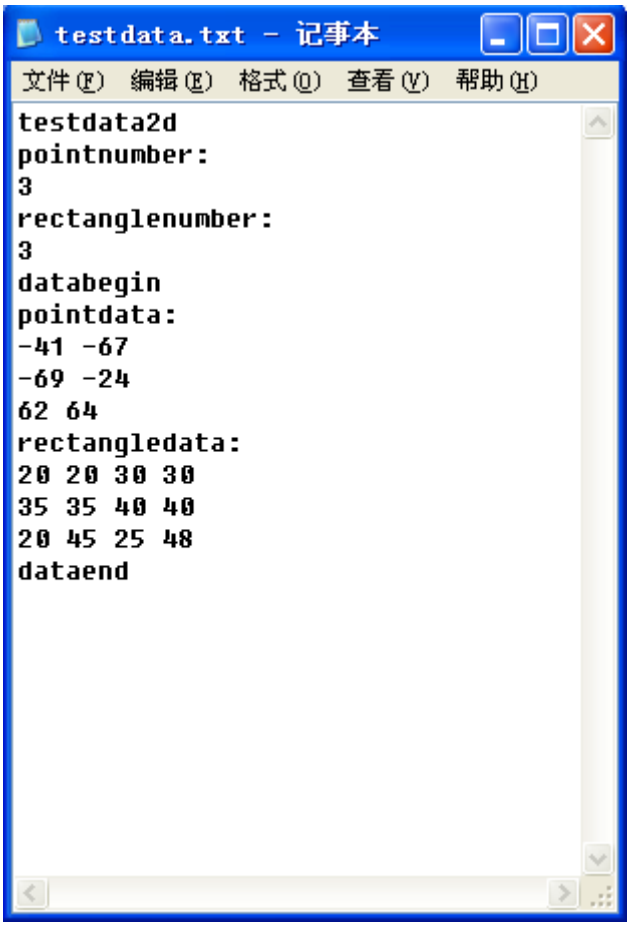
本实验将演示程序和 R 树的定义接口整合在了同一个工程中，如果单独使用请将 R 树的实现部分单独分离出来进行使用。

参考文献

- A Guttman ‘R-trees a dynamic mdex structure for spatial searching’, Proc ACM SIGMOD Int Conf on Management of Data, 47-57, 1984.
- BeckmannN ., KriegelH .-P.,S chneideRr , andS eegeBr . 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

附录

输入数据格式



读入数据的例子

文件首行是必须关键字：“testdata2d”。如果首行没有这个关键字，程序会报告“格式错误”。

接下来是各个数据的个数，目前只支持两种数据类型，一种是 2 维点，另一种是 2 维矩形。例如上图，输入 3 个点，和 3 个矩形，首先写“pointnumber：”，然后换行，然后写对应的数目“3”。换行，写“rectanglenumber：”，换行，写对应的数目“3”。可以调换点和矩形的输入顺序，也可以只写其中一种。

接下来由关键字“databegin”标记数据段开始。

接下来由关键字“pointdata”或“rectangledata”来标记数据的类型，数据类型后接坐标值。同一个数据的坐标值之间以“空格”分隔，不同的数据的坐标之间以“换行”分隔。

对于 2 维点，第一个数是 x 坐标值，第二个数是 y 坐标值。

对于 2 维矩形，第一个数是矩形左下角的 x 坐标，第二数是左下角的 y 坐标，第三个点是右上角的 x 坐标，第四个点是右上角的 y 坐标。如果输入的坐标顺序有错，那么程序计算出的结果将不正确。

最后由关键字“dataend”标记数据段结束。

输入数据的拓展名是“.txt”，这样做是为了查看和编辑数据方便，在手工编辑数据的时候有两点非常需要注意：1，前面声明数据的数目和后面真实的数据数目要一致，比如前面声明有 3 个点，那么后面必须有 3 个点的坐标，否则程序可能出现错误。2，尽量不要有空行，否则程序也可能出现错误。