ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

*«*САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт машиностроения, материалов и транспорта

Кафедра “Мехатроника и роботостроение (при ЦНИИ РТК)”

**Курсовой проект**

по дисциплине «Объектно-ориентированное программирование”

R-Tree

|  |  |  |
| --- | --- | --- |
| Выполнил  Студент гр. 3331506/90401 | *(подпись)* | Лазарев М. Р. |
| Работу принял  доцент, к.т.н. | *(подпись)* | Ананьевский М. С. |

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_2022 г

Санкт-Петербург

2022

# Вступление

R-Tree - древовидная структура данных (дерево), предложенная в 1984 году Антонином Гуттманом. Оно подобна B-дереву, но используется для организации доступа к пространственным данным, то есть для индексации многомерной информации такой, например, как географические данные с двумерными координатами (широтой и долготой). Типичным запросом с использованием R-деревьев мог бы быть такой: «Найти все музеи в пределах 2 километров от моего текущего местоположения».

Эта структура данных разбивает многомерное пространство на множество иерархически вложенных и, возможно, пересекающихся, прямоугольников (для двумерного пространства). В случае трехмерного или многомерного пространства это будут прямоугольные параллелепипеды (кубоиды) или параллелотопы.

Алгоритмы вставки и удаления используют эти ограничивающие прямоугольники для обеспечения того, чтобы «близкорасположенные» объекты были помещены в одну листовую вершину. В частности, новый объект попадёт в ту листовую вершину, для которой потребуется наименьшее расширение её ограничивающего прямоугольника. Каждый элемент листовой вершины хранит два поля данных: способ идентификации данных, описывающих объект, (либо сами эти данные) и ограничивающий прямоугольник этого объекта.

Аналогично, алгоритмы поиска (например, пересечение, включение, окрестности) используют ограничивающие прямоугольники для принятия решения о необходимости поиска в дочерней вершине. Таким образом, большинство вершин никогда не затрагиваются в ходе поиска. Как и в случае с B-деревьями, это свойство R-деревьев обусловливает их применимость для баз данных, где вершины могут выгружаться на диск по мере необходимости.Для расщепления переполненных вершин могут применяться различные алгоритмы, что порождает деление R-деревьев на подтипы: квадратичные и линейные.

Изначально R-деревья не гарантировали хороших характеристик для наихудшего случая, хотя хорошо работали на реальных данных. Однако в 2004-м году был опубликован новый алгоритм, определяющий приоритетные R-деревья. Утверждается, что этот алгоритм эффективен, как и наиболее эффективные современные методы, и в то же время является оптимальным для наихудшего случая.

Изображение выглядит как текст

Автоматически созданное описаниеНа рисунке 1 представлен пример построения R-дерева.

Рисунок 1 - Пример R-дерева

R–дерево является сбалансированной древоидной структурой с минимальным и максимальынм количеством записей, выделямых под один узел. Максимальное количество записей в узел (и максимальное количество потомков у этого узла) будем называть b (от англ. – branch-factor). Минимальное число записей назовем m. Для корректоной работы должно выполняться условие:

Предельная сложность алгоритмов вставки, удаления и поиска зависит от высоты дерева, которая, в свою очередь, зависит от числа элементов N и среднего количества элементов в листе.

Из выражения для числа элементов в дереве N

можно получить O-натацию данных алгоритмов вида (для высоты дерева):

В таблице 1 представлены результаты измерений алгоритма insert. На рисунке 2 представлены точки и линия тренда эксперимента.

Таблица 1 – Результаты измерений алгоритма insert

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N, шт | 10 | 50 | 100 | 200 | 500 | 1000 | 2000 | 3000 | 5000 |
| t, с | 3.5∙10-6 | 5.9∙10-6 | 6.6∙10-6 | 5.4∙10-5 | 1∙10-5 | 1.5∙10-4 | 5.7∙10-5 | 1.2∙10-5 | 5.9∙10-5 |

Рисунок 2 – Зависимость затрат времени от N для insert

Как видно из эксперимента, средняя сложность алгоритма insert соответствуеит теоретическим расчетам, но фактически затраты времени колеблются и зависят от прочих факторов (например, дерево может требовать перестроения).

В таблице 2 представлены результаты замеров алгоритма search. На рисунке 3 можно увидеть график зависимости затрат времени от N для алгоритма search.

Таблица 2 – Результаты измерений для search

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N, шт | 10 | 50 | 100 | 200 | 500 | 1000 | 2000 | 3000 | 5000 |
| t, с | 2.4∙10-4 | 2.9∙10-4 | 3.7∙10-4 | 4.5∙10-4 | 8.2∙10-4 | 1.3∙10-3 | 2.2∙10-3 | 2.7∙10-3 | 3.4∙10-3 |

Рисунок 3 – Зависимость затрат времени от N для search

Как видно из практических результатов, поиск для данной реализации R-Tree соответствует ожидаемой логорифмической зависимости.

В таблице 3 представлены результаты замеров алгоритма remove. На рисунке 4 можно увидеть график зависимости затрат времени от N для алгоритма remove.

Таблица 3 - Результаты измерений для remove

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N, шт | 10 | 50 | 100 | 200 | 500 | 1000 | 2000 | 3000 | 5000 |
| t, с | 5.7∙10-5 | 2.5∙10-4 | 5.2∙10-4 | 4.0∙10-4 | 7.4∙10-4 | 1.1∙10-3 | 1.8∙10-3 | 2.4∙10-3 | 3.0∙10-3 |

Рисунок 4 – Зависимость затрат времени от N для remove

Как видно из практических результатов, удаление элементов для данной реализации R-Tree соответствует ожидаемой логорифмической зависимости.

Ниже представлен пример построения R-дерева на основе реализованного кода с m = 2 и b = 4. На рисунке 5 находится древоидное представление R-дерева. На рисунке 6 предаставлено разделние дерева на пространственные узлы.

std::vector<*int*> data = {0,1,2,3,4,5,6,7,8,9};

r\_tree a(2,4);

a.insert( &data.at(0), rectangle( -30,20, -15,10) );

a.insert( &data.at(1), rectangle( 90,70, 100,90) );

a.insert( &data.at(2), rectangle( 0,0, 12,30) );

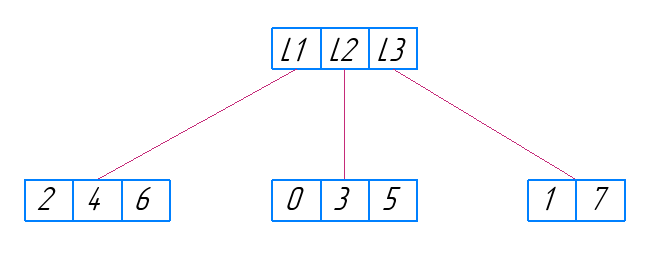
a.insert( &data.at(3), rectangle( -60,-60, -50,-50) );

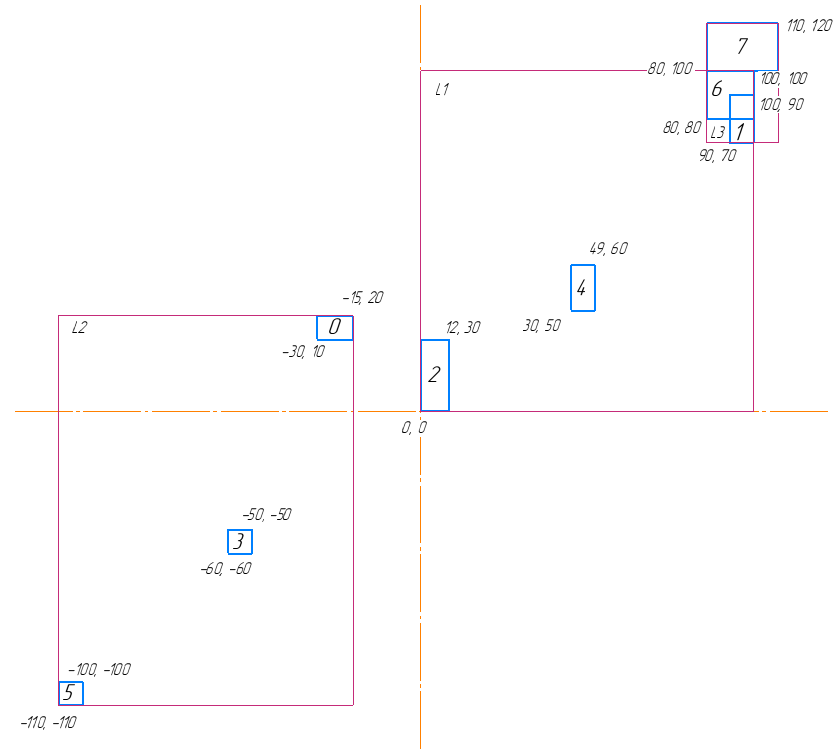
a.insert( &data.at(4), rectangle( 30,50, 49,60) );

a.insert( &data.at(5), rectangle( -100,-100, -110,-110) );

a.insert( &data.at(6), rectangle( 80,80, 100,100) );

a.insert( &data.at(7), rectangle( 80,100, 110,120) );

Рисунок 5 – Древоидное представление R-Tree

Рисунок 6 – Пространственное представление дерева

**Вывод**

На основе реализованного алгоритма можно видеть, что теоретические завимости проявляются и в данной реализации R-Tree. Такая структура хранения данных позволяет быстро работать с географическими объектами, но сам алгоритм может быть улучшен и уже достаточно давно можно анйти улучшенные версии R-Tree.

**Список литературы:**

1. R-TREES. A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING (1984) Antomn Guttman University of Cahforma Berkeley
2. Гулаков В. К., Трубаков А. О. [Многомерные структуры данных.](http://iipo.tu-bryansk.ru/index.php?id=book_mdstruct) 2010

**Приложение**

RTree.h

#ifndef **RTREE\_LIBRARY\_H**#define **RTREE\_LIBRARY\_H**#include **<string>**#include **<vector>**#include **<stack>  
  
class** r\_tree;  
**class** rectangle;  
**class** Irt\_node;  
**class** rt\_node;  
**class** leaf\_rt\_node;  
**struct** splitNodes;  
  
*//imt\_exception--------***class** imt\_exception : **public** std::exception {  
 std::string statement;  
**public**:  
 imt\_exception(**const** std::string statement) : statement(statement) {};  
  
 **const char** \*what() **const noexcept override** { **return** &(\*statement.begin()); }  
};  
  
*//point---------------***class** point {  
**public**:   
 **double** x;  
 **double** y;  
  
 point(**const double** x = 0, **const double** y = 0) : x(x), y(y) {};  
  
 point(**const** point &other) : x(other.x), y(other.y) {};  
  
 point(**const** point &&other) : x(other.x), y(other.y) {};  
  
 point &**operator**=(**const** point &other);  
  
 point &**operator**=(**const** point &&other);  
  
 **virtual** ~point() = **default**;  
  
 **bool operator**==(**const** point &other) **const**;  
 **bool operator**!=(**const** point &other) **const** { **return** !**operator**==(other); }  
  
**private**:  
 **bool** \_isInRectangle(**const** rectangle& rec);  
 **friend class** rectangle;  
};  
  
*//rectangle----------***class** rectangle {  
**public**:  
 point l\_point;  
 point r\_point;  
  
 rectangle(**const** point left\_point, **const** point right\_point) : l\_point(left\_point), r\_point(right\_point) {  
 \_normalize();  
 };  
  
 rectangle(**const double** frst\_x = 0, **const double** frst\_y = 0, **const double** sec\_x = 0, **const double** sec\_y = 0) :  
 rectangle(point(frst\_x, frst\_y), point(sec\_x, sec\_y)) {  
 \_normalize();  
 };  
  
 rectangle(**const** rectangle &other) : l\_point(other.l\_point), r\_point(other.r\_point) {  
 \_normalize();  
 };  
  
 rectangle(**const** rectangle &&other) : l\_point(other.l\_point), r\_point(other.r\_point) {  
 \_normalize();  
 };  
  
 **virtual** ~rectangle() = **default**;  
  
 **bool** areCrossedWith(**const** rectangle& other) **const**;  
  
 std::vector<point> getAllPoints() **const**;  
  
 rectangle getOverlapRecWith(**const** rectangle& other) **const**;  
  
 **double** getOverlapSizeWith(**const** rectangle& other) **const**;  
  
 **double** area() **const**;  
  
 **bool operator**==(**const** rectangle &other) **const**;  
  
 rectangle& **operator**=(**const** rectangle& other);  
  
 rectangle **operator**+(**const** rectangle& other);  
  
**private**:  
 **void** \_normalize();  
  
 rectangle \_tryExtendTo(**const** rectangle& other) **const**;  
  
 **void** \_extendTo(**const** rectangle& other);  
  
 **double** \_increasingArea(**const** rectangle& other) **const**;  
  
 **friend class** r\_tree;  
 **friend class** Irt\_node;  
 **friend class** leaf\_rt\_node;  
};  
  
*//Irt\_node---------------  
//Parrent class for nodes***class** Irt\_node {  
**protected**:  
 std::vector<rectangle> child\_mbr\_v;  
 rectangle mbr;  
  
**public**:  
 Irt\_node(**const** size\_t branch\_fcr) { child\_mbr\_v.reserve(branch\_fcr); }  
  
 Irt\_node(**const** Irt\_node &other) : child\_mbr\_v(other.child\_mbr\_v) {};  
  
 Irt\_node(**const** Irt\_node &&other) : child\_mbr\_v(other.child\_mbr\_v) {};  
  
 **virtual** ~Irt\_node() = **default**;  
  
 **bool operator**==(**const** Irt\_node &other) **const**;  
  
**protected**:  
 **bool** \_isLeaf() **const**;  
  
 **bool** \_isFull(**const** size\_t branch\_fctr) **const**;  
  
 **bool** \_isOverloaded(**const** size\_t branch\_fctr) **const**;  
  
 rectangle \_getMbr() **const**;  
  
 **void** \_updateMbr();  
  
 **bool** \_childsAreLeafs() **const**;  
  
 **double** \_getOverlappingSize(std::vector<Irt\_node\*> &nodes) **const**;  
  
 **double** \_wastedArea() **const**;  
  
 **friend class** r\_tree;  
 **friend class** rt\_node;  
};  
  
*//rt\_node---------------***class** rt\_node : **public** Irt\_node {  
 std::vector<Irt\_node\*> child\_v;  
  
**public**:  
 rt\_node(**const** size\_t branch\_fcr) : Irt\_node(branch\_fcr) { child\_v.reserve(branch\_fcr); }  
  
 rt\_node(**const** rt\_node &other) : Irt\_node(other), child\_v(other.child\_v) {};  
  
 rt\_node(**const** rt\_node &&other) : Irt\_node(other), child\_v(other.child\_v) {};  
  
 **void** \_insertSameAge(**const** rt\_node &other);  
  
 **void** \_eraseEntry(Irt\_node\* child);  
  
 **void** \_updateEntryMbr(Irt\_node\* child);  
  
 rt\_node& **operator**=(**const** rt\_node &other);  
  
 **virtual** ~rt\_node() = **default**;  
  
**private**:  
 **friend class** r\_tree;  
 **friend class** Irt\_node;  
};  
  
*//leaf\_rt\_node---------***class** leaf\_rt\_node : **public** Irt\_node {  
 std::vector<**void**\*> data\_v;  
  
**public**:  
 leaf\_rt\_node(**const** size\_t branch\_fcr) : Irt\_node(branch\_fcr) { data\_v.reserve(branch\_fcr); };  
  
 leaf\_rt\_node(leaf\_rt\_node &other) : Irt\_node(other), data\_v(other.data\_v) {};  
  
 leaf\_rt\_node(leaf\_rt\_node &&other) : Irt\_node(other), data\_v(other.data\_v) {};  
  
 **void** \_insertSameAge(**const** leaf\_rt\_node &other);  
  
 leaf\_rt\_node& **operator**=(**const** leaf\_rt\_node &other);  
  
 **virtual** ~leaf\_rt\_node() = **default**;  
  
**private**:  
   
 **friend class** r\_tree;  
 **friend class** Irt\_node;  
};  
  
  
  
*//r-tree------------***class** r\_tree {   
 Irt\_node \*root = **nullptr**;  
 size\_t height = 0;  
 **const** size\_t min\_child\_num;  
 **const** size\_t branch\_fctr;   
  
**public**:  
 r\_tree(**const** size\_t min\_child\_num = 2, **const** size\_t branch\_fctr = 4);  
  
 ~r\_tree() { clear(); }  
  
 **void** insert(**void** \***const** data, **const** rectangle insertingArea);  
  
 std::vector<**void**\*> search(**const** rectangle searchArea) **const**;  
  
 **void** remove(**void** \***const** data, **const** rectangle searchArea);  
  
 **void** clear();  
  
**private**:  
 Irt\_node\* \_chooseLeaf(Irt\_node\* curr\_I,  
 **void** \***const** data,  
 **const** rectangle &insertingArea,  
 std::stack<Irt\_node\*> &ancestry) **const**;  
  
 Irt\_node\* \_chooseSubTree(Irt\_node\* curr\_I, **const** rectangle &insertingArea) **const**;  
  
 splitNodes \_doInsert(Irt\_node\* leaf\_I, **void** \***const** data, **const** rectangle &insertingArea);  
  
 splitNodes \_split(Irt\_node\* node);  
  
 splitNodes \_splitLeaf(Irt\_node\* node);  
  
 splitNodes \_splitNode(Irt\_node\* node);  
  
 splitNodes \_adjustTree(Irt\_node\* node, splitNodes result, std::stack<Irt\_node\*> &ancestry, **bool** &rootWasSplited);  
  
 size\_t \_getSubtreeHeight(rt\_node\* localRoot) **const**;  
   
 **void** \_condenseTree(Irt\_node\* node, std::stack<Irt\_node\*> &ancestry, std::stack<Irt\_node\*> &orphanedSet);  
  
 **void** \_reinsertOrphanedSet(Irt\_node\* curr\_I);  
  
 **void** \_insertChildSplited(Irt\_node \*parrent, Irt\_node \*child);  
  
 **void** \_clearSplitNodes(splitNodes &sn) **const**;   
  
 std::pair<**int**, **int**> \_pickSeedsId(std::vector<rectangle> &rec\_v) **const**;  
  
 std::vector<leaf\_rt\_node\*> \_searchCrossedLeafs(**const** rectangle searchArea) **const**;  
  
 **bool** \_wasSplited(**const** splitNodes &result) **const**;  
  
 **int** \_pickNextId(Irt\_node\* group1, Irt\_node\* group2, std::vector<rectangle> &toAssert) **const**;  
};  
  
  
  
  
  
  
**struct** splitNodes   
{  
 splitNodes(Irt\_node\* first = **nullptr**, Irt\_node\* second = **nullptr**) :   
 first(first) , second(second) {};  
 Irt\_node\* first;  
 Irt\_node\* second;  
};  
  
  
#endif *//RTREE\_LIBRARY\_H*

RTree.cpp

#include **"RTree.h"**#include **<typeinfo>**#include **<functional>**#include **<iostream>**#include **<algorithm>**#include **<math.h>**#define **debug** ON  
  
*//point----------------*point &point::**operator**=(**const** point &other) {  
 x = other.x;  
 y = other.y;  
 **return** \***this**;  
}  
  
point &point::**operator**=(**const** point &&other) {  
 x = other.x;  
 y = other.y;  
 **return** \***this**;  
}  
  
**bool** point::\_isInRectangle(**const** rectangle &rec) {  
 **return** (x >= rec.l\_point.x && x <= rec.r\_point.x) && (y >= rec.l\_point.y && y <= rec.r\_point.y);  
}  
  
**bool** point::**operator**==(**const** point &other) **const** {  
 **return** (**this**->x == other.x && **this**->y == other.y);  
}  
  
*//rectangle-----------***void** rectangle::\_normalize() {  
 *//transform rectangle to form (left\_buttom - right\_top)   
 //normalize x* **if** (l\_point.x > r\_point.x) {  
 point temp(l\_point);  
 l\_point = r\_point;  
 r\_point = temp;  
 }  
  
 *//normalize y* **if** (r\_point.y < l\_point.y) {  
 **double** temp = l\_point.y;  
 l\_point.y = r\_point.y;  
 r\_point.y = temp;  
 }  
}  
  
std::vector<point> rectangle::getAllPoints() **const** {  
 *//return all 4 vertexes in order:  
 //left\_buttom, left\_top, right\_top, right\_buttom* std::vector<point> temp;  
 temp.push\_back(point(l\_point.x, l\_point.y));  
 temp.push\_back(point(l\_point.x, r\_point.y));  
 temp.push\_back(point(r\_point.x, r\_point.y));  
 temp.push\_back(point(r\_point.x, l\_point.y));  
  
 **return** temp;  
}  
  
**bool** rectangle::areCrossedWith(**const** rectangle &other) **const** {  
 *//checking each point for beeing into other rectangle* std::vector<point> this\_rec = getAllPoints();  
 std::vector<point> other\_rec = other.getAllPoints();  
 **bool** flag = **false**;  
 **for** (**auto** &el: other\_rec) {  
 flag = el.\_isInRectangle(\***this**);  
 **if** (flag) { **return** flag; }  
 }  
 **for** (**auto** &el: this\_rec) {  
 flag = el.\_isInRectangle(other);  
 **if** (flag) { **return** flag; }  
 }  
 **return** flag;  
}  
  
**double** rectangle::area() **const** {  
 **double** width = r\_point.x - l\_point.x;  
 **double** height = r\_point.y - l\_point.y;  
 **return** width \* height;  
}  
  
**double** rectangle::\_increasingArea(**const** rectangle& other) **const** {  
 *//returns area, which will be in case of   
 //including other rectangle* **double** first = area();  
 **double** second = \_tryExtendTo(other).area();  
 **return** (second - first);  
};  
  
rectangle rectangle::getOverlapRecWith(**const** rectangle &other) **const** {  
 *//returns the overlapping rectangle* point new\_left(std::max(l\_point.x, other.l\_point.x),  
 std::max(l\_point.y, other.l\_point.y));  
 point new\_right(std::min(r\_point.x, other.r\_point.x),  
 std::min(r\_point.y, other.r\_point.y));  
  
 **return** rectangle(new\_left, new\_right);  
}  
  
**double** rectangle::getOverlapSizeWith(**const** rectangle &other) **const** {  
 *//returns the overlapping size* **if** (!(**this**->areCrossedWith(other))) {  
 **return** 0;  
 }  
 rectangle overlap = getOverlapRecWith(other);  
 **double** width = overlap.r\_point.x - overlap.l\_point.x;  
 **double** height = overlap.r\_point.y - overlap.l\_point.y;  
  
 **return** width \* height;  
}  
  
rectangle rectangle::\_tryExtendTo(**const** rectangle &other) **const** {  
 *//return rectangle, which will be in case of  
 //extending to other* **double** new\_left\_X = std::min(**this**->l\_point.x, other.l\_point.x);  
 **double** new\_left\_Y = std::min(**this**->l\_point.y, other.l\_point.y);  
 **double** new\_right\_X = std::max(**this**->r\_point.x, other.r\_point.x);  
 **double** new\_right\_Y = std::max(**this**->r\_point.y, other.r\_point.y);  
 **return** rectangle(new\_left\_X, new\_left\_Y, new\_right\_X, new\_right\_Y);  
}  
  
**void** rectangle::\_extendTo(**const** rectangle &other) {  
 l\_point.x = std::min(**this**->l\_point.x, other.l\_point.x);  
 l\_point.y = std::min(**this**->l\_point.y, other.l\_point.y);  
 r\_point.x = std::max(**this**->r\_point.x, other.r\_point.x);  
 r\_point.y = std::max(**this**->r\_point.y, other.r\_point.y);  
}  
  
**bool** rectangle::**operator**==(**const** rectangle &other) **const** {  
 **return** (**this**->l\_point == other.l\_point && **this**->r\_point == other.r\_point);  
}  
  
rectangle& rectangle::**operator**=(**const** rectangle &other) {  
 l\_point = other.l\_point;  
 r\_point = other.r\_point;  
 **return** \***this**;  
}  
  
rectangle rectangle::**operator**+(**const** rectangle& other) {  
 rectangle toReturn = \***this**;  
 toReturn.\_extendTo(other);  
 **return** toReturn;  
}  
  
*//Irt\_node------------***bool** Irt\_node::\_isLeaf() **const** {  
 **return** (**typeid**(\***this**) == **typeid**(leaf\_rt\_node));  
}  
  
**bool** Irt\_node::\_isFull(**const** size\_t branch\_fctr) **const** {  
 **return** (**this**->child\_mbr\_v.size() == branch\_fctr);  
}  
  
**bool** Irt\_node::\_isOverloaded(**const** size\_t branch\_fctr) **const** {  
 **return** (**this**->child\_mbr\_v.size() > branch\_fctr);  
}  
  
rectangle Irt\_node::\_getMbr() **const** {  
 **if**( child\_mbr\_v.empty() ) {  
 **return** rectangle(0,0,0,0);  
 }  
 rectangle current = child\_mbr\_v.at(0);  
 **for** (size\_t i = 1; i < child\_mbr\_v.size(); ++i) {  
 current.\_extendTo(child\_mbr\_v.at(i));  
 }  
 **return** current;  
}  
  
**bool** Irt\_node::\_childsAreLeafs() **const** {  
 **const** rt\_node \*curr = **dynamic\_cast**<**const** rt\_node \*>(**this**);  
 **if** (curr->child\_v.size() == 0) {  
 **return false**;  
 } **else** {  
 **return** (curr->child\_v.at(0)->\_isLeaf());  
 }  
}  
  
**bool** Irt\_node::**operator**==(**const** Irt\_node &other) **const** {  
 **return** (**this**->\_getMbr() == other.\_getMbr());  
}  
  
**double** Irt\_node::\_getOverlappingSize(std::vector<Irt\_node \*> &nodes) **const** {  
 **double** overlappingSize = 0;  
 **for** (**auto** &node: nodes) {  
 **if** (**this** != node) {  
 overlappingSize += **this**->\_getMbr().getOverlapSizeWith(node->\_getMbr());  
 }  
 }  
 **return** overlappingSize;  
}  
  
**void** Irt\_node::\_updateMbr() {  
 mbr = \_getMbr();  
}  
  
**double** Irt\_node::\_wastedArea() **const** {  
 **double** toReturn = mbr.area();  
 **for** (**auto** &el : child\_mbr\_v) {  
 toReturn -= el.area();  
 }  
  
 **return** toReturn;  
}  
  
rt\_node& rt\_node::**operator**=(**const** rt\_node &other) {  
 child\_mbr\_v = other.child\_mbr\_v;  
 child\_v = other.child\_v;  
 mbr = other.mbr;  
  
 **return** \***this**;  
}  
  
**void** rt\_node::\_insertSameAge(**const** rt\_node &other) {  
 **for**(**int** c = 0; c < other.child\_v.size(); ++c){  
 child\_v.push\_back( other.child\_v.at(c) );  
 child\_mbr\_v.push\_back( other.child\_mbr\_v.at(c) );  
 }  
 \_updateMbr();  
}  
  
**void** rt\_node::\_eraseEntry(Irt\_node\* child) {  
 **int** index = 0;  
 **for**(**auto** &ch : child\_v) {  
 **if**(ch == child) {  
 child\_mbr\_v.erase( child\_mbr\_v.begin() + index );  
 child\_v.erase( child\_v.begin() + index );  
 }  
 ++index;  
 }  
 \_updateMbr();  
}  
  
**void** rt\_node::\_updateEntryMbr(Irt\_node\* child) {  
 **int** index = 0;  
 **for**(**auto**& ch : child\_v) {  
 **if**(ch == child) {  
 child\_mbr\_v.at(index) = child->mbr;  
 }  
 ++index;  
 }  
 \_updateMbr();  
}  
  
leaf\_rt\_node& leaf\_rt\_node::**operator**=(**const** leaf\_rt\_node &other) {  
 child\_mbr\_v = other.child\_mbr\_v;  
 data\_v = other.data\_v;  
 mbr = other.mbr;  
  
 **return** \***this**;  
}  
  
**void** leaf\_rt\_node::\_insertSameAge(**const** leaf\_rt\_node &other) {  
 **for**(**int** c = 0; c < other.data\_v.size(); ++c){  
 data\_v.push\_back( other.data\_v.at(c) );  
 child\_mbr\_v.push\_back( other.child\_mbr\_v.at(c) );  
 }  
 \_updateMbr();  
}  
  
*//r\_tree-----------*r\_tree::r\_tree(**const** size\_t min\_child\_num, **const** size\_t branch\_fctr) :  
 branch\_fctr(branch\_fctr), min\_child\_num(min\_child\_num) {  
   
 **if** (**this**->branch\_fctr / 2 < min\_child\_num) { **throw** imt\_exception(**"Uncorrect numbers of max and min entries."**); };  
}  
  
**void** r\_tree::remove(**void** \***const** data, **const** rectangle searchArea) {  
  
 *//isFound on enter must be false, stack should be empty* std::function< **void**( Irt\_node\*, **void** \***const**, **const** rectangle, Irt\_node\*\*, std::stack<Irt\_node\*>&, **bool**& ) > deepSearchRemove =  
 [&deepSearchRemove, **this**] ( Irt\_node\* curr\_I, **void** \***const** data, **const** rectangle searchArea, Irt\_node\*\* where, std::stack<Irt\_node\*> &ancestry, **bool** &isFound ) {  
  
 *//recursion base* **if**(curr\_I->\_isLeaf()) {  
 leaf\_rt\_node\* curr = **dynamic\_cast**<leaf\_rt\_node\*>(curr\_I);  
  
 **for**(**int** i = 0; i < curr->data\_v.size(); ++i){  
 **if**( curr->data\_v.at(i) == data ) {  
 isFound = **true**;  
 \*where = curr;  
 curr->data\_v.erase( curr->data\_v.begin() + i );  
 curr->child\_mbr\_v.erase( curr->child\_mbr\_v.begin() + i );  
 curr->\_updateMbr();  
 }  
 }  
  
 **return**;  
 }  
   
 **if**(!isFound) {  
 ancestry.push(curr\_I);  
 }  
  
 rt\_node\* curr = **dynamic\_cast**<rt\_node\*>(curr\_I);  
 **for**(**auto** &child : curr->child\_v) {  
 **if**( child->mbr.areCrossedWith(searchArea) ) {  
 deepSearchRemove(child, data, searchArea, where, ancestry, isFound);  
 }   
 }  
  
 **if**(!isFound) {  
 ancestry.pop();  
 }  
 };  
  
 Irt\_node\* where = **nullptr**;  
 std::stack<Irt\_node\*> ancestry;  
 **bool** isFound = **false**;  
 deepSearchRemove(root, data, searchArea, &where, ancestry, isFound);  
  
 std::stack<Irt\_node\*> orphanedSet;  
 \_condenseTree(where, ancestry, orphanedSet);  
  
 **if**( !root->\_isLeaf() && root->child\_mbr\_v.size() == 1 ) {  
 **if**(height != 0) {  
 --height;  
 }  
 Irt\_node\* toClear = root;  
 root = **dynamic\_cast**<rt\_node\*>(root)->child\_v.at(0);  
 **delete** toClear;  
 }   
  
 **if**( root->\_isLeaf() && root->child\_mbr\_v.empty() ) {  
 clear();  
 }  
}  
  
**void** r\_tree::clear() {  
 **if**(root->\_isLeaf()) {  
 **delete** root;  
 root = **nullptr**;  
 **return**;  
 }  
  
 *//labda deepSearch clearing* std::function< **void**(Irt\_node\*) > deepSearchClear = [&deepSearchClear, **this**] ( Irt\_node\* curr\_I ) {  
 *//recursion base* **if**(curr\_I->\_isLeaf()) {  
 **delete** curr\_I;  
 **return**;  
 }  
  
 rt\_node\* curr = **dynamic\_cast**<rt\_node\*>(curr\_I);  
 **for** (**auto** &el : curr->child\_v) {  
 deepSearchClear(el);  
 }  
 **delete** curr\_I;  
 **return**;  
 };  
  
 deepSearchClear(root);  
 height = 0;  
 root = **nullptr**;  
}  
  
**void** r\_tree::insert(**void** \***const** data, **const** rectangle insertingArea) {   
 **if**(root == **nullptr**) {  
 root = **new** leaf\_rt\_node(branch\_fctr);  
 }  
  
 std::stack<Irt\_node\*> ancestry;  
  
 Irt\_node\* where = \_chooseLeaf(root, data, insertingArea, ancestry);  
  
 splitNodes result = \_doInsert(where, data, insertingArea);  
  
 **bool** rootWasSplited = (\_wasSplited(result) && where == root) ? **true** : **false**;  
  
 result = \_adjustTree(where, result, ancestry, rootWasSplited);  
  
 **if**(rootWasSplited) {  
 ++height;  
 **delete** root;  
 root = **new** rt\_node(branch\_fctr);  
 \_insertChildSplited(root, result.first);  
 \_insertChildSplited(root, result.second);  
 }  
}  
  
splitNodes r\_tree::\_doInsert(Irt\_node\* leaf\_I, **void** \***const** data, **const** rectangle &insertingArea) {  
  
 leaf\_rt\_node\* leaf = **dynamic\_cast**<leaf\_rt\_node\*>(leaf\_I);  
  
 *//if node isn't full -> insert  
 //return null splitNodes* **if**(!leaf->\_isFull(branch\_fctr)) {  
 leaf->child\_mbr\_v.push\_back(insertingArea);  
 leaf->data\_v.push\_back(data);  
 splitNodes toReturn;  
 **return** toReturn;  
 }  
  
 *//do splitting  
 //return splited nodes* **else** {  
 leaf->child\_mbr\_v.push\_back(insertingArea);  
 leaf->data\_v.push\_back(data);  
 splitNodes toReturn = \_split(leaf\_I);  
 **return** toReturn;  
 }  
}  
  
  
Irt\_node \*r\_tree::\_chooseLeaf(Irt\_node\* curr\_I, **void** \***const** data, **const** rectangle &insertingArea, std::stack<Irt\_node\*> &ancestry) **const** {  
 *//recursion base* **if**(curr\_I->\_isLeaf()) {  
 **return** curr\_I;  
 }  
  
 ancestry.push(curr\_I);  
  
 curr\_I = \_chooseSubTree(curr\_I, insertingArea);  
  
 **return** \_chooseLeaf(curr\_I, data, insertingArea, ancestry);  
}  
  
Irt\_node\* r\_tree::\_chooseSubTree(Irt\_node\* curr\_I, **const** rectangle &insertingArea) **const** {  
 rt\_node\* curr = **dynamic\_cast**<rt\_node\*>(curr\_I);  
  
 **int** minIndex = 0;  
 **for**(**int** i = 0; i < curr->child\_v.size(); ++i) {  
 **if**( curr->child\_v.at(i)->mbr.\_increasingArea(insertingArea) < curr->child\_v.at(minIndex)->mbr.\_increasingArea(insertingArea) ) {  
 minIndex = i;  
 }  
  
 **if**( curr->child\_v.at(i)->mbr.\_increasingArea(insertingArea) == curr->child\_v.at(minIndex)->mbr.\_increasingArea(insertingArea) ) {  
 minIndex = ( curr->child\_v.at(i)->mbr.\_tryExtendTo(insertingArea).area() < curr->child\_v.at(minIndex)->mbr.\_tryExtendTo(insertingArea).area() ) ?  
 i : minIndex;  
 }  
 }  
  
 **return** curr->child\_v.at(minIndex);  
}  
  
splitNodes r\_tree::\_splitLeaf(Irt\_node\* nodeI) {  
 *//heap new nodes* Irt\_node\* nodeI\_1 = **new** leaf\_rt\_node(branch\_fctr);  
 Irt\_node\* nodeI\_2 = **new** leaf\_rt\_node(branch\_fctr);  
  
 *//cast list* leaf\_rt\_node\* leaf = **dynamic\_cast**<leaf\_rt\_node\*>(nodeI);  
 leaf\_rt\_node\* leaf\_1 = **dynamic\_cast**<leaf\_rt\_node\*>(nodeI\_1);  
 leaf\_rt\_node\* leaf\_2 = **dynamic\_cast**<leaf\_rt\_node\*>(nodeI\_2);  
  
 *//this vector must become 0* std::vector<rectangle> toAssert\_rec = leaf->child\_mbr\_v;  
 std::vector<**void**\*> toAssert\_data = leaf->data\_v;  
  
 *//find first enries for each of splited nodes* std::pair<**int**, **int**> firstEnrtyIndex = \_pickSeedsId(toAssert\_rec);  
 rectangle r1 = toAssert\_rec.at(firstEnrtyIndex.first);   
 rectangle r2 = toAssert\_rec.at(firstEnrtyIndex.second);  
 **void**\* d1 = toAssert\_data.at(firstEnrtyIndex.first);  
 **void**\* d2 = toAssert\_data.at(firstEnrtyIndex.second);  
  
 *//first elements insert* leaf\_1->child\_mbr\_v.push\_back(toAssert\_rec.at(firstEnrtyIndex.first));  
 leaf\_1->data\_v.push\_back(toAssert\_data.at(firstEnrtyIndex.first));  
 leaf\_2->child\_mbr\_v.push\_back(toAssert\_rec.at(firstEnrtyIndex.second));  
 leaf\_2->data\_v.push\_back(toAssert\_data.at(firstEnrtyIndex.second));  
  
 *//cleaf from toAssert\_rec* toAssert\_rec.erase( toAssert\_rec.begin() + firstEnrtyIndex.first );  
 toAssert\_rec.erase( toAssert\_rec.begin() + firstEnrtyIndex.second - ((firstEnrtyIndex.first < firstEnrtyIndex.second) ? 1 : 0) );  
  
 *//cleaf from toAssert\_data* toAssert\_data.erase( toAssert\_data.begin() + firstEnrtyIndex.first );  
 toAssert\_data.erase( toAssert\_data.begin() + firstEnrtyIndex.second - ((firstEnrtyIndex.first < firstEnrtyIndex.second) ? 1 : 0) );  
  
 **while**(!toAssert\_data.empty()) {  
 *//if list size is so small, that it could be insert in leaf\_1 -> insert in leaf\_1* **if**( ((toAssert\_data.size() + leaf\_1->data\_v.size() >= min\_child\_num) && (toAssert\_data.size() + leaf\_1->data\_v.size() <= branch\_fctr)) &&  
 leaf\_2->data\_v.size() >= min\_child\_num ) {  
 **int** size\_rest = toAssert\_data.size();  
 **for**(**int** i = 0; i < size\_rest; ++i) {  
 *//insert* leaf\_1->child\_mbr\_v.push\_back( toAssert\_rec.at(i) );  
 leaf\_1->data\_v.push\_back( toAssert\_data.at(i) );   
 }  
  
 toAssert\_rec.clear();  
 toAssert\_data.clear();  
 **return** splitNodes(nodeI\_1, nodeI\_2);  
 }  
  
 *//if list size is so small, that it could be insert in leaf\_2 -> insert in leaf\_2* **else if**( ((toAssert\_data.size() + leaf\_2->data\_v.size() >= min\_child\_num) && (toAssert\_data.size() + leaf\_2->data\_v.size() <= branch\_fctr)) &&  
 leaf\_1->data\_v.size() >= min\_child\_num ) {  
 **int** size\_rest = toAssert\_data.size();  
 **for**(**int** i = 0; i < size\_rest; ++i) {  
 *//insert* leaf\_2->child\_mbr\_v.push\_back( toAssert\_rec.at(i) );  
 leaf\_2->data\_v.push\_back( toAssert\_data.at(i) );   
 }  
  
 toAssert\_rec.clear();  
 toAssert\_data.clear();  
 **return** splitNodes(nodeI\_1, nodeI\_2);  
 }  
  
 *//insert next* **int** nextIndex = \_pickNextId(nodeI\_1, nodeI\_2, toAssert\_rec);  
 **double** inc\_1 = leaf\_1->mbr.\_increasingArea( toAssert\_rec.at(nextIndex) );  
 **double** inc\_2 = leaf\_2->mbr.\_increasingArea( toAssert\_rec.at(nextIndex) );  
 leaf\_rt\_node\* where = (inc\_1 < inc\_2) ? leaf\_1 : leaf\_2;  
 **if**( inc\_1 - inc\_2 == 0 ) {  
 where = (leaf\_1->mbr.area() < leaf\_2->mbr.area()) ? leaf\_1 : leaf\_2;  
 }  
   
 *//insert* where->child\_mbr\_v.push\_back( toAssert\_rec.at(nextIndex) );  
 where->data\_v.push\_back( toAssert\_data.at(nextIndex) );  
  
 *//clear* rectangle r = toAssert\_rec.at(nextIndex);  
 **void**\* d = toAssert\_data.at(nextIndex);  
 toAssert\_rec.erase( toAssert\_rec.begin() + nextIndex );  
 toAssert\_data.erase( toAssert\_data.begin() + nextIndex );  
 }  
  
 **return** splitNodes(nodeI\_1, nodeI\_2);  
}  
  
splitNodes r\_tree::\_splitNode(Irt\_node\* nodeI) {  
 *//heap new nodes* Irt\_node\* nodeI\_1 = **new** rt\_node(branch\_fctr);  
 Irt\_node\* nodeI\_2 = **new** rt\_node(branch\_fctr);  
  
 *//cast list* rt\_node\* leaf = **dynamic\_cast**<rt\_node\*>(nodeI);  
 rt\_node\* leaf\_1 = **dynamic\_cast**<rt\_node\*>(nodeI\_1);  
 rt\_node\* leaf\_2 = **dynamic\_cast**<rt\_node\*>(nodeI\_2);  
  
 *//these vectors must become 0* std::vector<rectangle> toAssert\_rec = leaf->child\_mbr\_v;  
 std::vector<Irt\_node\*> toAssert\_data = leaf->child\_v;  
  
 *//find first enries for each of splited nodes* std::pair<**int**, **int**> firstEnrtyIndex = \_pickSeedsId(toAssert\_rec);  
 rectangle r1 = toAssert\_rec.at(firstEnrtyIndex.first);   
 rectangle r2 = toAssert\_rec.at(firstEnrtyIndex.second);  
 Irt\_node\* d1 = toAssert\_data.at(firstEnrtyIndex.first);  
 Irt\_node\* d2 = toAssert\_data.at(firstEnrtyIndex.second);  
  
 *//first elements insert* leaf\_1->child\_mbr\_v.push\_back( toAssert\_rec.at(firstEnrtyIndex.first) );  
 leaf\_1->child\_v.push\_back( toAssert\_data.at(firstEnrtyIndex.first) );  
 leaf\_2->child\_mbr\_v.push\_back( toAssert\_rec.at(firstEnrtyIndex.second) );  
 leaf\_2->child\_v.push\_back( toAssert\_data.at(firstEnrtyIndex.second) );  
   
 *//cleaf from toAssert\_rec* toAssert\_rec.erase( toAssert\_rec.begin() + firstEnrtyIndex.first );  
 toAssert\_rec.erase( toAssert\_rec.begin() + firstEnrtyIndex.second - ((firstEnrtyIndex.first < firstEnrtyIndex.second) ? 1 : 0) );  
  
 *//cleaf from toAssert\_data* toAssert\_data.erase( toAssert\_data.begin() + firstEnrtyIndex.first );  
 toAssert\_data.erase( toAssert\_data.begin() + firstEnrtyIndex.second - ((firstEnrtyIndex.first < firstEnrtyIndex.second) ? 1 : 0) );  
  
 **while**(!toAssert\_data.empty()) {  
 *//if list size is so small, that it could be insert in leaf\_1 -> insert in leaf\_1* **if**( ((toAssert\_data.size() + leaf\_1->child\_v.size() >= min\_child\_num) && (toAssert\_data.size() + leaf\_1->child\_v.size() <= branch\_fctr)) &&  
 leaf\_2->child\_v.size() >= min\_child\_num ) {  
 **int** size\_rest = toAssert\_data.size();  
 **for**(**int** i = 0; i < size\_rest; ++i) {  
 *//insert* leaf\_1->child\_mbr\_v.push\_back( toAssert\_rec.at(i) );  
 leaf\_1->child\_v.push\_back( toAssert\_data.at(i) );   
 }  
  
 toAssert\_rec.clear();  
 toAssert\_data.clear();  
 **return** splitNodes(nodeI\_1, nodeI\_2);  
 }  
  
 *//if list size is so small, that it could be insert in leaf\_2 -> insert in leaf\_2* **else if**( ((toAssert\_data.size() + leaf\_2->child\_v.size() >= min\_child\_num) && (toAssert\_data.size() + leaf\_2->child\_v.size() <= branch\_fctr)) &&  
 leaf\_1->child\_v.size() >= min\_child\_num ) {  
 **int** size\_rest = toAssert\_data.size();  
 **for**(**int** i = 0; i < size\_rest; ++i) {  
 *//insert* leaf\_2->child\_mbr\_v.push\_back( toAssert\_rec.at(i) );  
 leaf\_2->child\_v.push\_back( toAssert\_data.at(i) );   
 }  
  
 toAssert\_rec.clear();  
 toAssert\_data.clear();  
 **return** splitNodes(nodeI\_1, nodeI\_2);  
 }  
  
 *//insert next* **int** nextIndex = \_pickNextId(nodeI\_1, nodeI\_2, toAssert\_rec);  
 **double** inc\_1 = leaf\_1->mbr.\_increasingArea( toAssert\_rec.at(nextIndex) );  
 **double** inc\_2 = leaf\_2->mbr.\_increasingArea( toAssert\_rec.at(nextIndex) );  
 rt\_node\* where = (inc\_1 < inc\_2) ? leaf\_1 : leaf\_2;  
 **if**( inc\_1 - inc\_2 == 0 ) {  
 where = (leaf\_1->mbr.area() < leaf\_2->mbr.area()) ? leaf\_1 : leaf\_2;  
 }  
   
 *//insert* where->child\_mbr\_v.push\_back( toAssert\_rec.at(nextIndex) );  
 where->child\_v.push\_back( toAssert\_data.at(nextIndex) );  
  
 *//clear* rectangle r = toAssert\_rec.at(nextIndex);  
 **void**\* d = toAssert\_data.at(nextIndex);  
 toAssert\_rec.erase( toAssert\_rec.begin() + nextIndex );  
 toAssert\_data.erase( toAssert\_data.begin() + nextIndex );  
 }  
  
 **return** splitNodes(nodeI\_1, nodeI\_2);  
}  
  
splitNodes r\_tree::\_split(Irt\_node\* nodeI) {  
 splitNodes toReturn = (nodeI->\_isLeaf()) ? \_splitLeaf(nodeI) : \_splitNode(nodeI);  
 toReturn.first->\_updateMbr();  
 toReturn.second->\_updateMbr();  
 **return** toReturn;  
}  
  
**bool** r\_tree::\_wasSplited(**const** splitNodes &result) **const** {  
 **bool** toReturn = (result.second == **nullptr**) ? **false** : **true**;  
 **return** toReturn;  
}  
  
std::pair<**int**, **int**> r\_tree::\_pickSeedsId(std::vector<rectangle> &rec\_v) **const** {  
 **int** i\_max = 0;  
 **int** j\_max = 0;  
 **double** maxWasteArea = 0;  
 **for**(**int** i = 0; i < rec\_v.size(); ++i) {  
 **for**(**int** j = 0; j < rec\_v.size(); ++j) {  
 **if**(i != j) {  
 rectangle paired = rec\_v.at(i) + rec\_v.at(j);  
 **double** wasteArea = paired.area() - rec\_v.at(i).area() - rec\_v.at(j).area() + rec\_v.at(i).getOverlapSizeWith(rec\_v.at(j));  
  
 **if**(wasteArea > maxWasteArea) {  
 maxWasteArea = wasteArea;  
 i\_max = i;  
 j\_max = j;  
 }  
 }  
 }   
 }  
  
 **if**(i\_max == j\_max) {  
 i\_max = 0;  
 j\_max = rec\_v.size() - 1;  
 }  
  
 **return** std::pair<**int**, **int**>(i\_max, j\_max);  
}  
  
**int** r\_tree::\_pickNextId(Irt\_node\* group1, Irt\_node\* group2, std::vector<rectangle> &toAssert) **const** {  
 *//d\_i the area increase required in the covering rectangle of Group i to include entire* **int** i\_max = 0;  
 **double** maxDiff = 0;  
 **for**(**int** i = 0; i < toAssert.size(); ++i) {  
 **double** areaAtFirst = group1->mbr.area();  
 **double** areaAtSecond = group1->mbr.\_tryExtendTo(toAssert.at(i)).area();  
 **double** d\_1 = areaAtSecond - areaAtFirst;  
  
 areaAtFirst = group2->mbr.area();  
 areaAtSecond = group2->mbr.\_tryExtendTo(toAssert.at(i)).area();  
 **double** d\_2 = areaAtSecond - areaAtFirst;  
  
 **double** diff = abs(d\_2 - d\_1);  
  
 **if**(diff > maxDiff) {  
 maxDiff = diff;  
 i\_max = i;  
 }  
 }  
  
 **return** i\_max;  
}  
  
splitNodes r\_tree::\_adjustTree(Irt\_node\* node, splitNodes result, std::stack<Irt\_node\*> &ancestry, **bool** &rootWasSplited) {  
 *//recursion base* **if**(node == root) {  
 node->\_updateMbr();  
 **return** result;  
 }  
  
 *//set nodes* Irt\_node\* N = node;  
 Irt\_node\* NN = result.second;  
  
 *//if theare was a splitting -> refactor N* **if**(\_wasSplited(result)) {  
 **if**(N->\_isLeaf()) {  
 leaf\_rt\_node\* NLeaf = **dynamic\_cast**<leaf\_rt\_node\*>(N);  
 leaf\_rt\_node\* resultLeafFirst = **dynamic\_cast**<leaf\_rt\_node\*>(result.first);  
  
 NLeaf->**operator**=(\*resultLeafFirst);  
 }   
 **else** {  
 rt\_node\* NNode = **dynamic\_cast**<rt\_node\*>(N);  
 rt\_node\* resultNodeFirst = **dynamic\_cast**<rt\_node\*>(result.first);  
  
 NNode->**operator**=(\*resultNodeFirst);  
 }  
 }  
 N->\_updateMbr();  
  
 *//get next parrent* rt\_node\* p = **dynamic\_cast**<rt\_node\*> ( ancestry.top() );  
 ancestry.pop();  
  
 *//get index of node in parrent* **int** index = 0;  
 **for**(**int** c = 0; c < p->child\_v.size(); ++c) {  
 index = ( p->child\_v.at(c) == node ) ? c : index;  
 }  
  
 *//update mbr of entire in parrent and parrent' mbr* p->child\_mbr\_v.at( index ) = N->mbr;  
 p->\_updateMbr();  
  
 *//if theare was splitting -> insert NN into p* **if**(\_wasSplited(result)) {  
 \_insertChildSplited(p, NN);  
 \_clearSplitNodes(result);  
  
 *//in case of overloading -> split* **if**( p->\_isOverloaded(branch\_fctr) ) {  
 result = \_split(p);  
 **if**(p == root) { rootWasSplited = **true**; }  
 }  
 }  
  
 **return** \_adjustTree(p, result, ancestry, rootWasSplited);  
}  
  
**void** r\_tree::\_insertChildSplited(Irt\_node \*parrent, Irt\_node \*child) {  
 **auto** a = **dynamic\_cast**<rt\_node\*>(parrent);  
  
 a->child\_v.push\_back( child );  
 a->child\_mbr\_v.push\_back( child->mbr );  
 a->\_updateMbr();  
}  
  
**void** r\_tree::\_clearSplitNodes(splitNodes &sn) **const** {  
 *//theare is a memory clearing only in first, cause the first one is copied,  
 //but the second adress must live in this tree* **delete** sn.first;  
 sn.first = **nullptr**;  
 sn.second = **nullptr**;  
}  
  
**void** r\_tree::\_condenseTree(Irt\_node\* node, std::stack<Irt\_node\*> &ancestry, std::stack<Irt\_node\*> &orphanedSet) {  
 *//recursion base* **if**(node == root) {  
 **while**( !orphanedSet.empty() ) {  
 Irt\_node\* curr\_I = orphanedSet.top();  
 orphanedSet.pop();  
 \_reinsertOrphanedSet(curr\_I);  
 }  
  
 **return**;  
 }  
  
 rt\_node\* p = **dynamic\_cast**<rt\_node\*>(ancestry.top());  
 ancestry.pop();  
  
 **if**(node->child\_mbr\_v.size() < min\_child\_num) {  
 orphanedSet.push(node);  
 p->\_eraseEntry(node);  
 } **else** {  
 p->\_updateEntryMbr(node);  
 }  
  
 \_condenseTree(p, ancestry, orphanedSet);  
  
}  
  
**void** r\_tree::\_reinsertOrphanedSet(Irt\_node\* curr\_I) {  
 *//recursion base* **if**(curr\_I->\_isLeaf()) {  
 leaf\_rt\_node\* curr = **dynamic\_cast**<leaf\_rt\_node\*>(curr\_I);  
 **int** index = 0;  
 **for**(**auto**& el : curr->data\_v) {  
 insert( curr->data\_v.at(index), curr->child\_mbr\_v.at(index) );  
 ++index;  
 }  
  
 **delete** curr;   
 **return**;  
 } **else** {  
 rt\_node\* curr = **dynamic\_cast**<rt\_node\*>(curr\_I);  
 **for**(**auto**& child : curr->child\_v) {  
 \_reinsertOrphanedSet(child);  
 }  
 **delete** curr;  
 }  
}  
  
std::vector<leaf\_rt\_node\*> r\_tree::\_searchCrossedLeafs(**const** rectangle searchArea) **const** {  
  
 std::function< **void**( Irt\_node\*, **const** rectangle, std::vector<leaf\_rt\_node\*>& ) > deepSearchLeaf =  
 [&deepSearchLeaf, **this**] ( Irt\_node\* curr\_I, **const** rectangle searchArea, std::vector<leaf\_rt\_node\*> &toReturn) {  
  
 *//recursion base* **if**(curr\_I->\_isLeaf()) {  
 leaf\_rt\_node\* curr = **dynamic\_cast**<leaf\_rt\_node\*>(curr\_I);  
 toReturn.push\_back(curr);  
  
 **return**;  
 }  
   
 rt\_node\* curr = **dynamic\_cast**<rt\_node\*>(curr\_I);  
 **for**(**auto** &child : curr->child\_v) {  
 **if**( child->mbr.areCrossedWith(searchArea) ) {  
 deepSearchLeaf(child, searchArea, toReturn);  
 }   
 }  
 };  
  
 std::vector<leaf\_rt\_node\*> toReturn;  
 deepSearchLeaf(root, searchArea, toReturn);  
  
 **return** toReturn;  
}  
  
std::vector<**void**\*> r\_tree::search(**const** rectangle searchArea) **const** {  
  
 std::vector<leaf\_rt\_node\*> crossedLeafs = \_searchCrossedLeafs(searchArea);  
 std::vector<**void**\*> toReturn;  
 **for**(**int** i = 0; i < crossedLeafs.size(); ++i){  
 **auto** childMbr = crossedLeafs.at(i)->child\_mbr\_v;  
 **auto** childData = crossedLeafs.at(i)->data\_v;  
  
 **for**(**int** k = 0; k < childData.size(); ++k) {  
 **if**(childMbr.at(k).areCrossedWith( searchArea )) {  
 toReturn.push\_back( childData.at(k) );  
 }  
 }  
 }  
  
 **return** toReturn;  
}  
  
size\_t r\_tree::\_getSubtreeHeight(rt\_node\* localRoot) **const** {  
 size\_t toReturn = 1;  
 Irt\_node\* next = localRoot->child\_v.at(0);  
 **while**( !next->\_isLeaf() ) {  
 next = **dynamic\_cast**<rt\_node\*>(next)->child\_v.at(0);  
 ++toReturn;  
 }  
  
 **return** toReturn;  
};