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 _____ Лазарев М. Р.
(подпись)

Работу принял
доцент, к.т.н. _____ Ананьевский М. С.
(подпись)

«____» _____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.

На рисунке 1 представлен пример построения R-дерева.

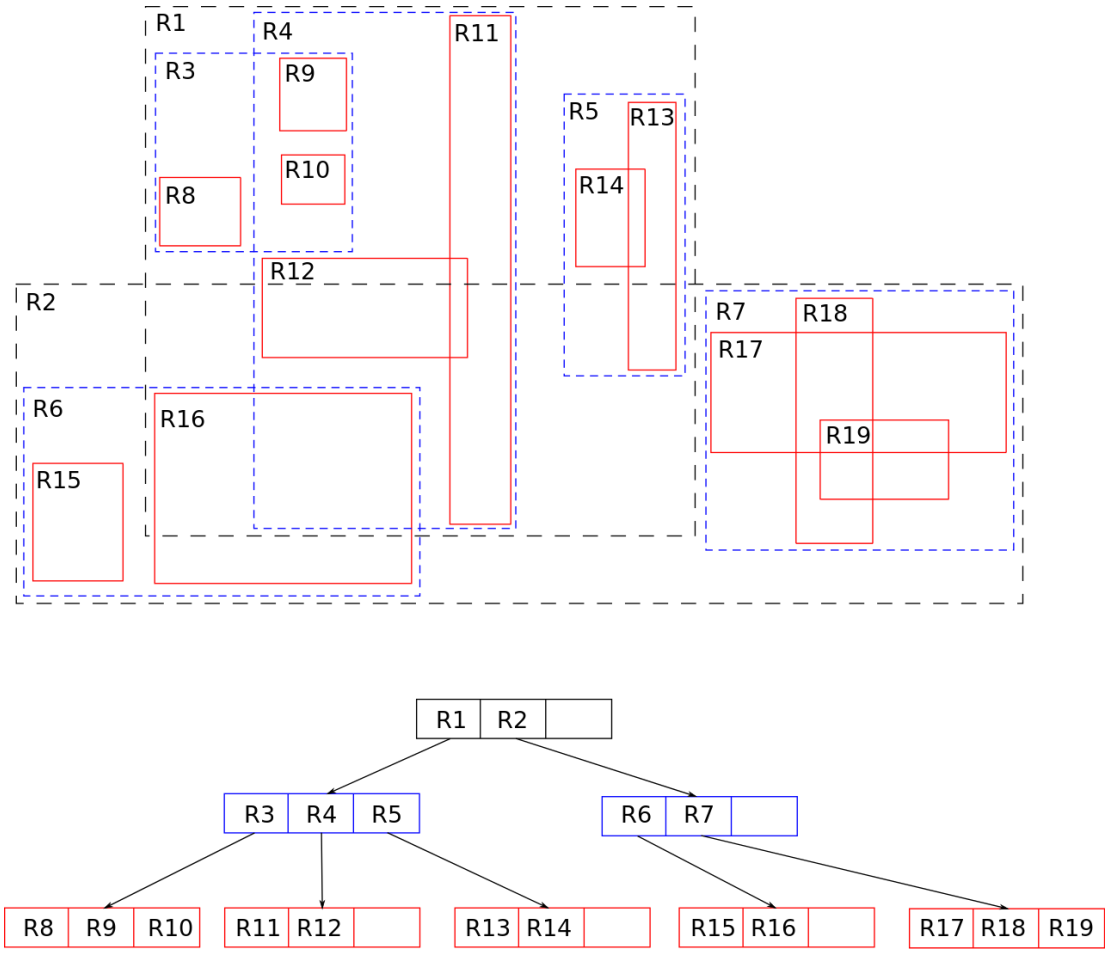

Рисунок 1 - Пример R-дерева

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:

$$\frac{b}{2} \geq m$$

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

$$N = n^{h+1}$$

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

$$O(logN)$$

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, c	$3.5 \cdot 10^{-6}$	$5.9 \cdot 10^{-6}$	$6.6 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$5.7 \cdot 10^{-5}$	$1.2 \cdot 10^{-5}$	$5.9 \cdot 10^{-5}$

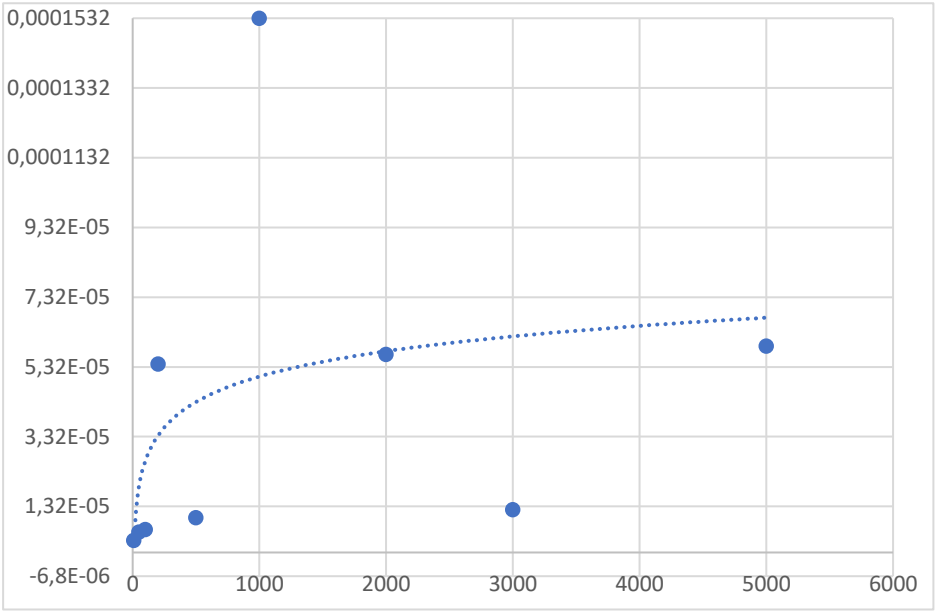

Рисунок 2 – Зависимость затрат времени от N для insert

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, c	$2.4 \cdot 10^{-4}$	$2.9 \cdot 10^{-4}$	$3.7 \cdot 10^{-4}$	$4.5 \cdot 10^{-4}$	$8.2 \cdot 10^{-4}$	$1.3 \cdot 10^{-3}$	$2.2 \cdot 10^{-3}$	$2.7 \cdot 10^{-3}$	$3.4 \cdot 10^{-3}$

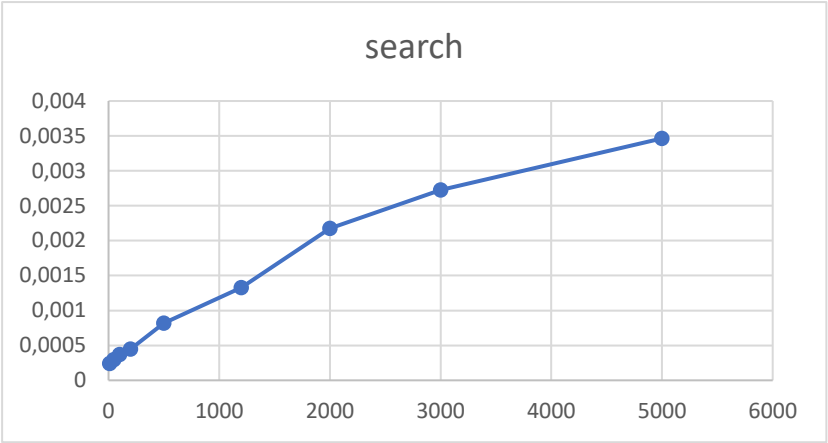

Рисунок 3 – Зависимость затрат времени от N для search

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, c	$5.7 \cdot 10^{-5}$	$2.5 \cdot 10^{-4}$	$5.2 \cdot 10^{-4}$	$4.0 \cdot 10^{-4}$	$7.4 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$	$1.8 \cdot 10^{-3}$	$2.4 \cdot 10^{-3}$	$3.0 \cdot 10^{-3}$

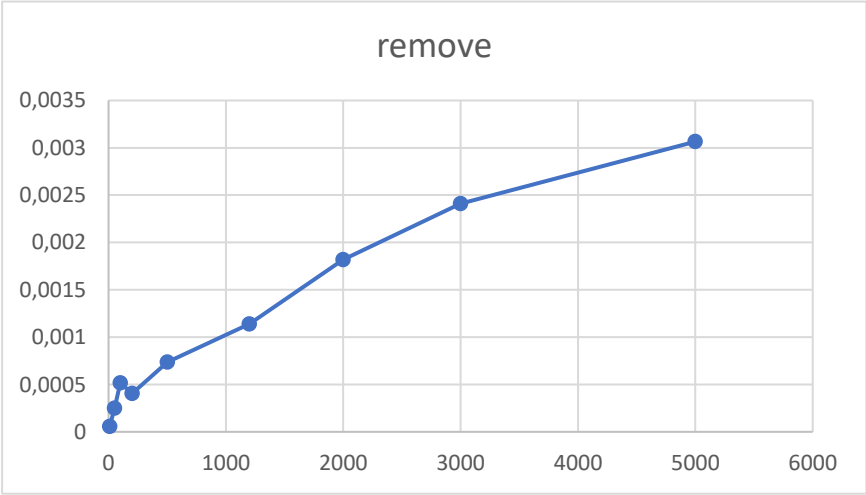

Рисунок 4 – Зависимость затрат времени от N для remove

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) );
```

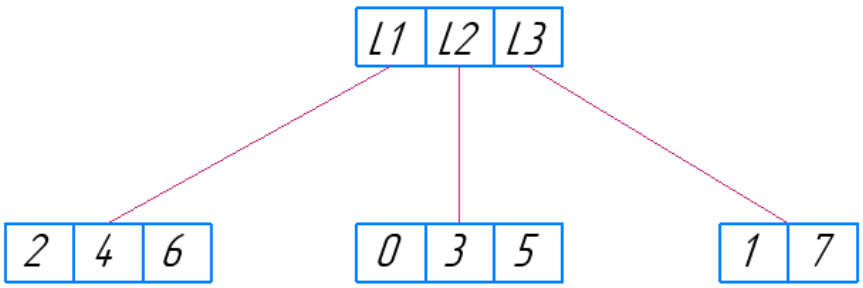

Рисунок 5 – Древоидное представление R-Tree

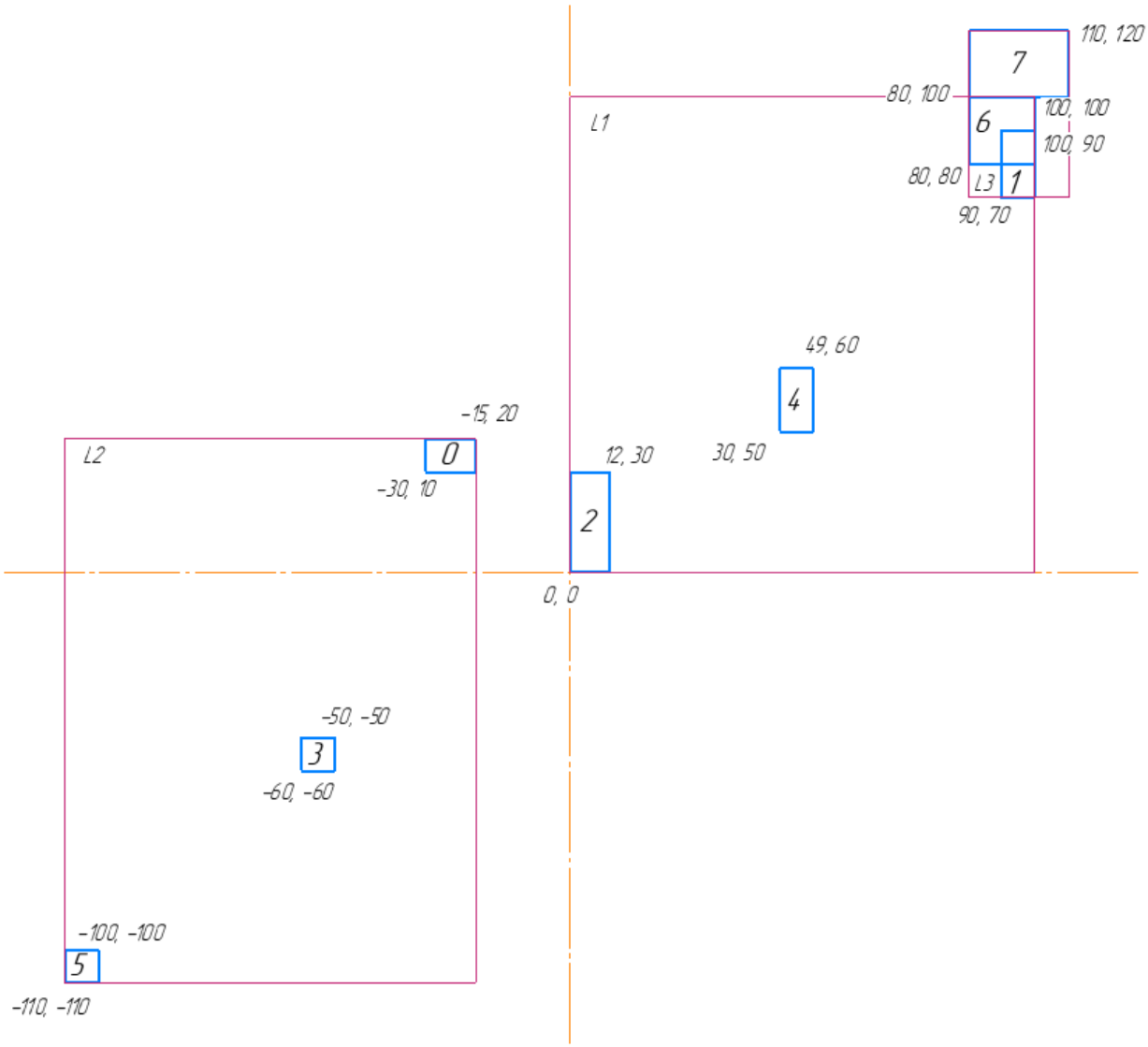

Рисунок 6 – Пространственное представление дерева

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. 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;
}
```

```cpp
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;
};
```