**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФГБОУ ВО  
**«БРЯНСКИЙ ГОСУДАРСТВЕНННЫЙ   
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**Кафедра** «Информатика и программное обеспечение»

**«У Т В Е Р Ж Д А Ю»**  
Зав. кафедрой «И и ПО», к.т.н., доцент  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Подвесовский А.Г.  
«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2022г.

**КУРСОВАЯ РАБОТА**

3D R\*-дерево

всего листов 31

Выполнил студент гр. О-20-МОА-тп-Б

зач. кн. №\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Шумилов Г. Р.

«\_\_\_» май 2025 г.

Руководитель

\_\_\_\_\_\_\_\_\_\_\_ Трубаков А. О.

«\_\_\_» май 2025 г.

Брянск 2025

**СОДЕРЖАНИЕ**

[Введение 4](#_Toc106910457)

[1. Общее описание исследуемой предметной области 5](#_Toc106910458)

[1.1. Аналитическая часть 5](#_Toc106910459)

[1.1.1. Общее описание R\*-дерева 5](#_Toc106910460)

[1.1.2. Некоторые определения 6](#_Toc106910461)

[1.1.3. Эффективность 6](#_Toc106910462)

[1.2. Описание алгоритмов 6](#_Toc106910463)

[1.2.1. Алгоритм вставки 6](#_Toc106910464)

[1.2.2. Алгоритм выбора узла для вставки. 6](#_Toc106910465)

[1.2.3. Алгоритм расщепления узла 7](#_Toc106910466)

[1.2.4. Алгоритм сортировки дочерних элементов по оси. 7](#_Toc106910467)

[1.2.5. Алгоритм слияния узлов 10](#_Toc106910468)

[1.2.6. Алгоритмы поиска. 11](#_Toc106910469)

[1.2.7. Алгоритм поиска ближайшего соседа. 11](#_Toc106910470)

[1.2.8. Алгоритм удаления. 11](#_Toc106910471)

[2. Конструкторская часть 12](#_Toc106910472)

[2.1. Общее описание 12](#_Toc106910473)

[2.2. Геометрические классы. 12](#_Toc106910474)

[2.2.1. Точка 12](#_Toc106910475)

[2.2.2. Параллелепипед. 13](#_Toc106910476)

[2.3. Логические классы 15](#_Toc106910477)

[2.3.1. Узел. 15](#_Toc106910478)

[2.3.2. Дерево 16](#_Toc106910479)

[3. Тестирование 18](#_Toc106910480)

[3.1. Первый набор данных. 18](#_Toc106910481)

[3.1.1. Построение 18](#_Toc106910482)

[3.1.2. Поиск 21](#_Toc106910483)

[3.1.3. Удаление 22](#_Toc106910484)

[3.2. Второй набор данных 25](#_Toc106910485)

[3.2.1. Построение 25](#_Toc106910486)

[3.2.2. Поиск 26](#_Toc106910487)

[3.2.3. Удаление 28](#_Toc106910488)

[Заключение 30](#_Toc106910489)

[Список литературы 31](#_Toc106910490)

# Введение

Многомерные структуры данных позволяют хранить нетривиальную информацию в больших количествах, при этом их методы доступа дают быстрый доступ к необходимой информации.

В условиях непрерывно растущей компьютеризации во всех сферах жизнедеятельности, стал вопрос о хранении информации об объектах на цифровых картах местности.

Если хранить данные линейно, то из-за огромного количества объектов на такой карте, доступ к ним будет медленный и неэффективный.

Подобную проблему в классической ситуации решали такие структуры данных, как деревья. Однако классические деревья не подходят для хранения информации об объекте на карте, то есть на плоскости.

Для решения этой проблемы в 1984 году Антонином Гуттманом было предложена новая структура данных – R-дерево. Однако, несмотря на плюсы этой структуры, были выявлены недостатки, а именно, смежные узлы слишком много пересекали друг друга, из-за чего поиск в таком дереве был замедлен.

В 1990 году Норберт Бекманн, Ханс-Петер Кригель , Ральф Шнайдер и Бернхард Зигер предложили свою модификацию R-дерева, которая давала лучшие результаты при поиске узлов. Данную модификацию назвали R\*-дерево.

Обычно такие деревья строят для объектов на плоскости, однако можно предположить, что в будущем будут нужны подобные структуры данных для полноценных трёхмерных карт, например, для навигации в космосе.

Целью данной работы является построение алгоритмов создания данной структуры, а также исследование её при разных входных данных.

# 1. Общее описание исследуемой предметной области

## 1.1. Аналитическая часть

### 1.1.1. Общее описание R\*-дерева

Данные в R-деревьях организованы в страницы, которые могут иметь переменное количество записей (до некоторого заранее определенного максимума и обычно выше минимального заполнения). Каждая запись в нелистовом узле хранит две части данных: способ идентификации дочернего узла и ограничивающую рамку всех записей в этом дочернем узле. Конечные узлы хранят данные, необходимые для каждого дочернего элемента, часто точку или ограничивающую рамку, представляющую дочерний элемент, и внешний идентификатор для дочернего элемента. Для точечных данных конечными записями могут быть только сами точки. Для данных многоугольников (которые часто требуют хранения больших многоугольников) обычная настройка - хранить только MBR (минимальный ограничивающий прямоугольник) многоугольника вместе с уникальным идентификатором в дереве.

R \* -деревья представляют собой вариант R-деревьев, используемых для индексации пространственной информации. R \* -деревья имеют немного более высокую стоимость строительства, чем стандартные R-деревья, так как данные могут нуждаться в повторной вставке; но результирующее дерево обычно будет иметь лучшую производительность запроса. Как и стандартное R-дерево, оно может хранить как точечные, так и пространственные данные.

3D R\*-деревья практически ничем не отличаются от обычных R\*-деревьев на плоскости, с той лишь разницей, что их минимальным ограничивающим регионом служит не прямоугольник, а прямоугольный параллелепипед.

### 1.1.2. Некоторые определения

Конечный узел – узел, хранящий только листья. Корень может быть конечным узлом.

Лист – узел, не имеющий потомков, т.е. непосредственно объект на карте.

MBP (minimal bounding parallelepiped) – минимальный ограничивающий параллелепипед.

### 1.1.3. Эффективность

Расход памяти в среднем и в наихудшем случае одинаковый – O(n).

Эффективность поиска и вставки – O(log n) (обращений к узлам дерева).

## 1.2. Описание алгоритмов

### 1.2.1. Алгоритм вставки

1. Выбрать узел N для вставки.

2. Добавить в узел N новый узел, расширить родительский узел при необходимости.

3. Если после вставки в узле N дочерних узлов стало слишком много, то необходимо вызвать процедуру расщепления узла N.

### 1.2.2. Алгоритм выбора узла для вставки.

Начать этот алгоритм следует с назначения очередного узла N корнем дерева.

1. Рассмотреть узел N, расширить при необходимости.

2. Если узел N является конечным, то вернуть данный узел.

3. В противном случае, выбрать потомка, который при расширении для вставки даст новый MBP, дающий минимальный объём пересечений с другими дочерними MBP в узле N.

4. Если таких узлов несколько, выбрать тот, что при увеличении даст наименьшую разницу в объёме.

5. Если и таких узлов несколько, выбрать узел с минимальным объёмом.

6. Этот дочерний узел назначить узлом N, применить к нему данный алгоритм, начиная с пункта 1.

### 1.2.3. Алгоритм расщепления узла

1. Проверить количество дочерних узлов в данном узле.

2. Отсортировать дочерние элементы по осям Ox, Oy, Oz.

3. Сформировать три пары новых узлов в соответствии с сортировками.

4. Сравнить эти пары между собой, выбрать вариант, который даёт наименьшее взаимное пересечение MBP-ов.

5. Если таких вариантов несколько, то выбрать тот, что даёт наименьший суммарный объём.

6. Всех потомков перенести в соответствующие новые два узла.

7. Взять предка старого узла N. Добавить в него новые узлы, удалить старый.

8. Если данный узел является корнем дерева, то создаём новый корень, в котором будут лежать новые два узла.

9. Выполнить алгоритм расщепления, для узла-предка с первого пункта.

### 1.2.4. Алгоритм сортировки дочерних элементов по оси.

Для наглядности лучше представить изображения (рисунок 1).

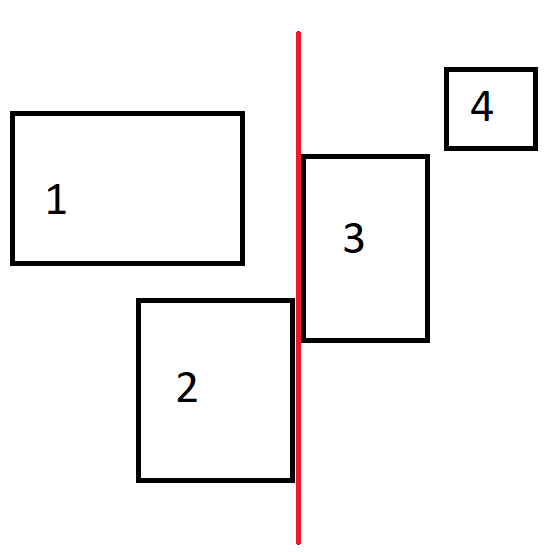


Рисунок 1 – демонстрация сортировки узлов по оси Ox и примерное разбиение

Следует пронумеровать узлы по оси по «левой» и «правой» границам. После чего «левейшие» узлы отнести в «левую» половину, а остальные – в «правую» половину.

Однако надо иметь ввиду, что объекты – не точки, и имеют свою ширину. Именно для этого необходимо учитывать разные нумерации – по левой и правой границам.

На рисунке 2 изображена нумерация узлов по «левой» (т.е. та, что по координатам является меньшей на оси) границе, на данном примере её можно также назвать «нижней».

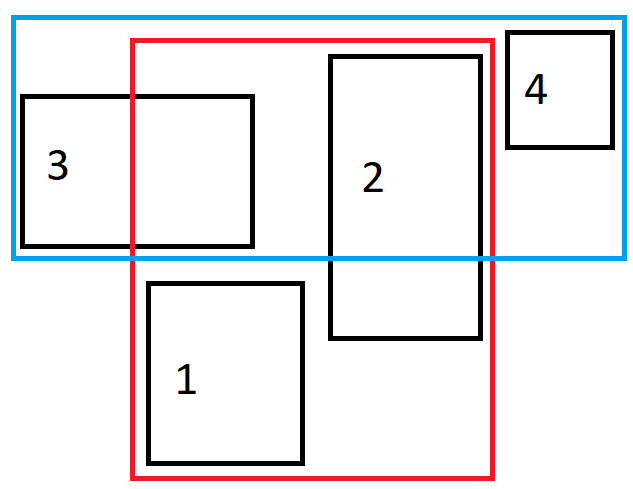


Рисунок 2 – демонстрация сортировки узлов по оси Oy по «левой» границе

Можно заметить, что такое разбиение узла не оптимальное. Однако, можно разбить эту же конфигурацию по той же оси но по «правой» границе (рисунок 3).

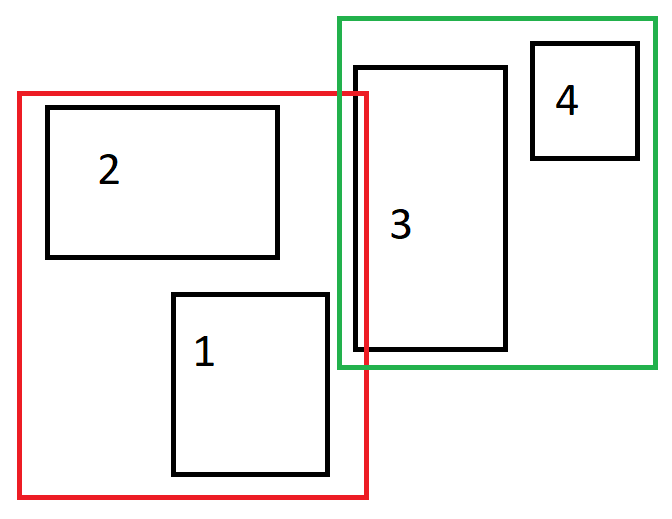


Рисунок 3 – демонстрация сортировки узлов по оси Oy по «правой» границе

Очевидно, что такое разбиение гораздо выгоднее. Это несложно подтверждается алгоритмами.

### 1.2.5. Алгоритм слияния узлов

1. Обратиться к предку N, установить его как узел M.

2. Отсортировать дочерние элементы узла M по осям Ox, Oy, Oz.

3. Выбрать из остальных дочерних узлов M ближайший, который в результате слияния c N даст наименьший объём пересечений.

4. Если подобных узлов несколько, выбрать тот, что при поглощении узла N, даст наименьшее увеличение.

5. Если таких узлов несколько, то выбрать узел, который будет с наименьшим объёмом при поглощении.

6. Установить M как N, выполнить алгоритм с 1 пункта, если в N будет недостаточно узлов.

7. Если N – корень, то ничего не делать.

### 1.2.6. Алгоритмы поиска.

Алгоритмы поиска в точке и в области достаточно тривиальны. Они представляют из себя рекурсивные алгоритмы, где, начиная с корня, идёт поиск дочерних узлов, пересекающихся с точкой, либо областью. Повторять до тех пор, пока не будут найдены все подходящие узлы.

Алгоритм поиска конкретного узла аналогичен поиску по области, с той лишь разницей, что область должна быть равна MBP искомого узла.

### 1.2.7. Алгоритм поиска ближайшего соседа.

1. Выбрать точку, от которой будет вестись поиск.

2. Установить корень дерева как узел N.

3. Если узел N является листом, или все его дочерние узлы были рассмотрены, то добавить его в список, вернуться к его предку и установить его узлом N.

4. Если узел N содержит нерассмотренные узлы, то рассчитать дистанцию до каждого дочернего узла N, выбрать ближайший, и рассмотреть его по данному алгоритму с пункта 3.

### 1.2.8. Алгоритм удаления.

Алгоритм удаления достаточно тривиален. Сначала следует вызвать алгоритм поиска удаляемого узла, удалить этот узел, если есть, после чего, если в предке удалённого узла стало слишком мало узлов, то выполнить алгоритм слияния.

# 2. Конструкторская часть

## 2.1. Общее описание

Для реализации данной структуры данных, было решено отдельно создать два класса – точка и параллелепипед.

Точка представлена как вектор координат, что получилось крайне удачно при реализации алгоритмов сортировки по осям. Кроме всего прочего, были переопределены операции сравнения, которые работали по принципу «если все координаты соответствуют данной операции сравнения, значит данная операция вернёт true».

Параллелепипед представлен как вектор, состоящий из двух точек, которые являются крайними точками его диагонали. Кроме базовых методов доступа и прочего, данный класс умеет вычислять пересечение двух параллелепипедов, содержание одного параллелепипеда в другом и т.д.

Узел дерева содержит в себе параллелепипед, указатель на данные, строчку его описания, вектор потомков, указатель на предка.

Дерево в себе содержит единственный указатель на корень дерева. Причём в конструкторе корень сразу же инициализируется, т.к. удалить корень в данном дереве – нельзя.

Реализация алгоритмов, описанных в первой главе, разнесена между классами узла и дерева.

Так как общий исходный код получился слишком громоздким, то в приложениях будут размещены лишь реализации самых важных алгоритмов.

## 2.2. Геометрические классы.

### 2.2.1. Точка

Интерфейс точки представлен в листинге 1.

*Листинг 1*

#pragma once

#include <iostream>

#include <vector>

class Point

{

public:

Point();

Point(int x, int y, int z);

Point(Point const& point);

virtual ~Point();

int getX() const;

int getY() const;

int getZ() const;

void setX(int x);

void setY(int y);

void setZ(int z);

int get(int axis) const;

void set(int coordinate, int axis);

friend std::ostream& operator << (std::ostream& out, Point const& point);

friend std::istream& operator >> (std::istream& in, Point& point);

Point operator = (Point const& point);

bool operator == (Point const& point) const;

bool operator != (Point const& point) const;

bool operator < (Point const& point) const;

bool operator <= (Point const& point) const;

bool operator > (Point const& point) const;

bool operator >= (Point const& point) const;

Point operator + (Point const& point) const;

Point operator - (Point const& point) const;

protected:

protected:

std::vector<int> coordinates\_;

};

### 2.2.2. Параллелепипед.

Интерфейс параллелепипеда представлен в листинге 2.

*Листинг 2*

#pragma once

#include "Point.h"

#include <vector>

class Parallelepiped

{

public:

Parallelepiped();

Parallelepiped(int Ox, int Oy, int Oz);

Parallelepiped(Point location, int Ox, int Oy, int Oz);

Parallelepiped(Point location, Point end);

Parallelepiped(Parallelepiped const& parallelepiped);

~Parallelepiped();

// Возвращает "рабочие" координаты узла

Point getLocation() const;

void setLocation(Point point);

// Возвращает приблизительный центр узла

Point location() const;

Point getEnd() const;

void setEnd(Point point);

int get(int i, int axis) const;

void set(int coordinate, int i, int axis);

int getX() const;

int getY() const;

int getZ() const;

void setX(int x);

void setY(int y);

void setZ(int z);

int getX2() const;

int getY2() const;

int getZ2() const;

void setX2(int x);

void setY2(int y);

void setZ2(int z);

int getOx() const;

int getOy() const;

int getOz() const;

void setOx(int Ox);

void setOy(int Oy);

void setOz(int Oz);

friend std::ostream& operator << (std::ostream& out, Parallelepiped const& parallelepiped);

friend std::istream& operator >> (std::istream& in, Parallelepiped& parallelepiped);

Parallelepiped operator = (Parallelepiped const& parallelepiped);

bool operator == (Parallelepiped const& parallelepiped) const;

bool operator != (Parallelepiped const& parallelepiped) const;

bool operator < (Parallelepiped const& parallelepiped) const;

bool operator <= (Parallelepiped const& parallelepiped) const;

bool operator > (Parallelepiped const& parallelepiped) const;

bool operator >= (Parallelepiped const& parallelepiped) const;

int volume() const;

int area() const;

// Возвращает true, если паралеллипипед содержит точку

bool contains(Point const& point) const;

// Возвращает true, если паралеллипипед полность содержит в себе другой паралеллипипед

bool contains(Parallelepiped const& parallelepiped) const;

// Возвращает true, если паралеллипипеды пересекаются друг другом

bool overlapping(Parallelepiped const& parallelepiped) const;

// Возвращает паралелипипед, являющийся расширенным паралеллипипедом, чтобы вместить в себя аргумент

Parallelepiped extend(Parallelepiped const& parallelepiped) const;

// Возвращает паралелипипед, являющийся пересечением this и аргумента

Parallelepiped intersection(Parallelepiped const& parallelepiped) const;

private:

private:

std::vector<Point> point\_;

};

## 2.3. Логические классы

### 2.3.1. Узел.

Интерфейс узла дерева представлен в листинге 3.

*Листинг 3*

#pragma once

#include "Parallelepiped.h"

#include <iostream>

#include <string>

#include <map>

class Node

{

friend class TDRtree;

public:

Node();

Node(std::string description);

Node(Parallelepiped const& MBP);

Node(std::string description, Parallelepiped const& MBP);

Node(std::string description, void\* data, Parallelepiped const& MBP);

Node(Node const& node);

~Node();

void add(Node\* newObject);

void del(Node\* object);

std::ostream& show(std::ostream& out, std::string tab) const;

friend std::ostream& operator << (std::ostream& out, Node const& node);

friend std::ostream& operator << (std::ostream& out, Node const\* node);

int distance(Point const& point) const;

private:

// Данный метод "обрывает" все связи с потомками и предком, не удаляя их!

void breakBounds();

// Возвращает массив индексов элементов, отсоритрованных по заданной оси

std::vector<int> sortByAxis(int axis, int border) const;

// Возвращает два узла, полученных в результате деления узла по заданной оси

std::pair<Node\*, Node\*> splitByAxis(int axis) const;

std::pair<Node\*, Node\*> mergeByAxis(Node\* node, int axis) const;

std::pair<Node\*, Node\*> mergeNodeByAxis(std::vector<int> numeration, Node\* node) const;

std::pair<Node\*, Node\*> chooseOptimalMergin(std::pair<Node\*, Node\*> left, std::pair<Node\*, Node\*> right, Node\* node) const;

int findChildIndex(std::vector<int> numeration, Node\* child) const;

void reduce();

bool isLeaf() const;

bool childrenAreLeafs() const;

Node\* chooseChild(Node\* newObject);

private:

Parallelepiped MBP\_;

std::string description\_;

void\* data\_;

std::vector<Node\*> children\_;

Node\* parent\_;

};

### 2.3.2. Дерево

Интерфейс дерева представлен в листинге 4.

*Листинг 4*

#pragma once

#define DEBUG

#include "Node.h"

#include <stack>

#define CHILDREN\_MAX\_COUNT 5

#define CHILDREN\_MIN\_COUNT 2

class TDRtree

{

friend class Node;

private:

public:

TDRtree();

~TDRtree();

// Добавляет новый объект в дерево

void add(std::string description, void\* data, Parallelepiped const& MBP);

// Возвращает массив узлов, которые содержат данную точку

std::vector<Node\*> pointSearch(Point const& point) const;

// Возвращает массив узлов, которые перекрываются данной областью

std::vector<Node\*> rangeSearch(Parallelepiped const& parallelepiped) const;

// Поиск ближайших соседей

std::vector<Node\*> closestNeighbours(Point const& point, int max) const;

// Поиск конкретного узла по названию и региону

Node\* find(std::string description, Parallelepiped const& MBP) const;

// Удаление заданного узла со всеми потомками

void del(std::string description, Parallelepiped const& MBP);

friend std::ostream& operator << (std::ostream& out, TDRtree const&);

private:

void replace(Node\* old, Node\* n1, Node\* n2);

Node\* choose(Node\* newObject);

Node\* chooseNode(Node\* node, Node\* newObject);

Node\* cluster(Node\* node, std::string description, Parallelepiped const& MBP) const;

void cluster(Point const& point, Node\* node, std::stack<Node\*>\* s) const;

void cluster(Parallelepiped const& parallelepiped, Node\* node, std::stack<Node\*>\* s) const;

void cluster(Point const& point, Node\* node, std::vector<Node\*>\* nodes, int\* max, int\* count) const;

// Производит разбиение узла на два других узла

void split(Node\* node);

// Производит оптимальное сращивание узла с другим узлом

void merge(Node\* node);

void normalize(Node\* node);

void del(Node\* node);

void redirection(Node\* was, Node\* now);

private:

Node\* root\_;

#ifdef DEBUG

int N\_;

public:

static int operations;

#endif // DEBUG

};

# 3. Тестирование

Для упрощения процесса тестирования были написаны вспомогательные функции, в которых автоматически строятся разные варианты дерева, после чего можно исследовать ту или иную конфигурацию.

Также в код были добавлены сегменты, которые активны при определённом макросе DEBUG, необходимые для тестирования и отладки дерева. Они показывают количество листьев в дереве и количество обращений к любым узлам дерева за одну операцию.

## 3.1. Первый набор данных.

### 3.1.1. Построение

Первый набор данных представляет собой набор хаотичных объектов в трёхмерном пространстве, служит скорее для того, чтобы определить правильность построения дерева. Этот набор добавляется в дерево функцией, представленной в листинге 5.

*Листинг 5*

void firstSet(TDRtree& tree)

{

tree.add("A", nullptr, Parallelepiped(Point(12, 25, 10), 10, 10, 5));

tree.add("B", nullptr, Parallelepiped(Point(-5, 5, 5), 3, 3, 3));

tree.add("C", nullptr, Parallelepiped(Point(-40, -30, -50), 15, 10, 8));

tree.add("D", nullptr, Parallelepiped(Point(50, 60, 50), 20, 30, 15));

tree.add("E", nullptr, Parallelepiped(Point(80, -50, -5), 17, 8, 21));

tree.add("F", nullptr, Parallelepiped(Point(25, 50, 10), 5, 5, 10));

tree.add("G", nullptr, Parallelepiped(Point(0, -40, 0), 20, 15, 25));

tree.add("H", nullptr, Parallelepiped(Point(50, 8, 0), 5, 5, 10));

tree.add("I", nullptr, Parallelepiped(Point(20, 5, -6), 11, 7, 12));

tree.add("J", nullptr, Parallelepiped(Point(30, 30, 40), 10, 15, 10));

std::cout << tree << "\n\n";

}

Результат построения такой конфигурации представлен на рисунке 4.

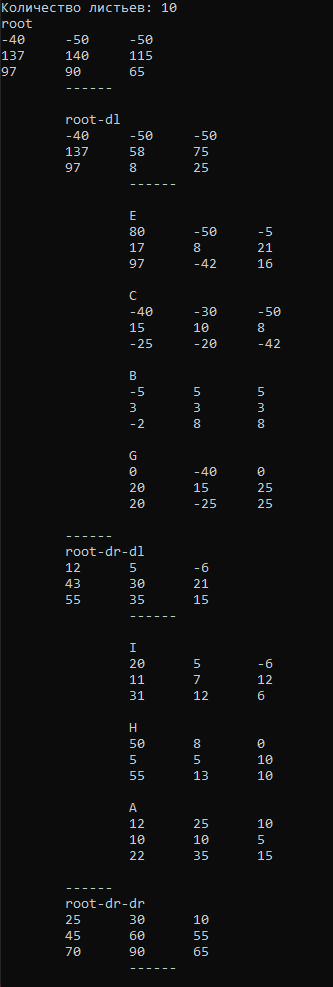


Рисунок 4 – вывод на экран первой тестовой конфигурации

К сожалению, весь результат выполнения не поместился в один скриншот. Однако, данное представление данных хоть и достаточно информативно, но всё ещё не наглядно для человека. Поэтому, ввиду малого количества объектов, была построена в программе трёхмерного моделирования SketchUp модель, согласно этим данным (рисунок 5)

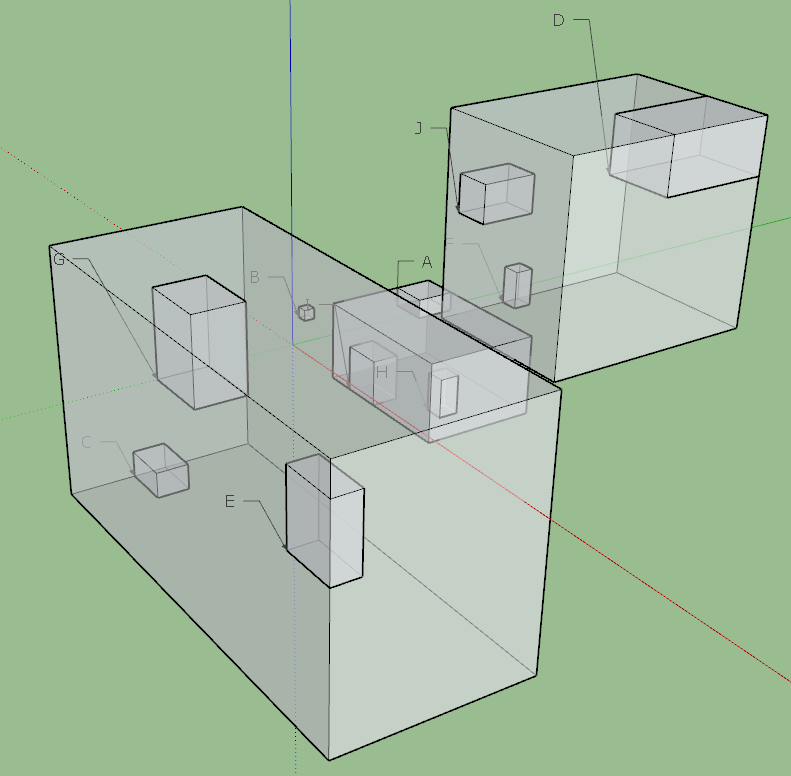


Рисунок 5 – модель первой тестовой конфигурации в SketchUp

Чтобы не перегружать данный рисунок, корневой узел был спрятан, однако он точно содержит в себе три конечных узла, которые отчётливо видно на рисунке 5.

### 3.1.2. Поиск

Для тестирования был вызван метод поиска ближайших соседей, его результаты можно увидеть на рисунке 6.

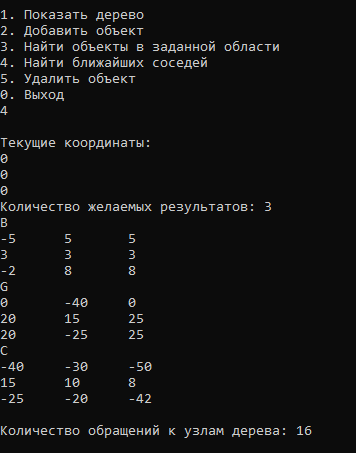


Рисунок 6 – результаты поиска ближайших соседей в первой тестовой конфигурации

Подобным образом был протестирован метод поиска по области (рисунок 7).

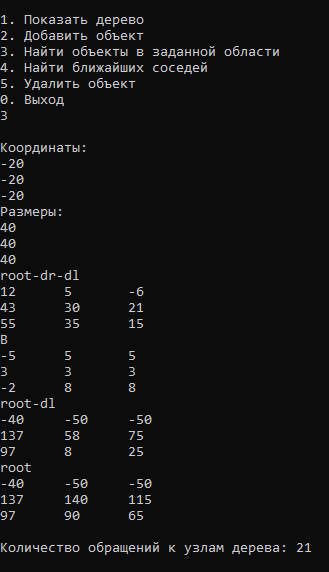


Рисунок 7 – результаты поиска в заданной области в первой тестовой конфигурации

Можно заметить, что в данной конфигурации, количество обращений к узлам гораздо больше, чем есть самих листьев в дереве. Это не отвечает критерию эффективности поиска O(log n). Однако надо учесть, что данная конфигурация мала, и такая сложная структура избыточна для десятка элементов.

### 3.1.3. Удаление

Результат удаления узла C представлен на рисунке 8.

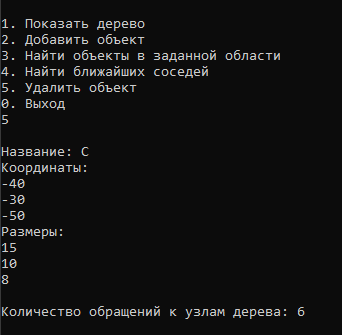


Рисунок 8 – результаты удаления узла «С» в первой тестовой конфигурации

Для полноты тестирования были удалены ещё два узла-соседа узла «C» - «B» и «E», после чего дерево имело следующий вид (рисунок 9, 10).

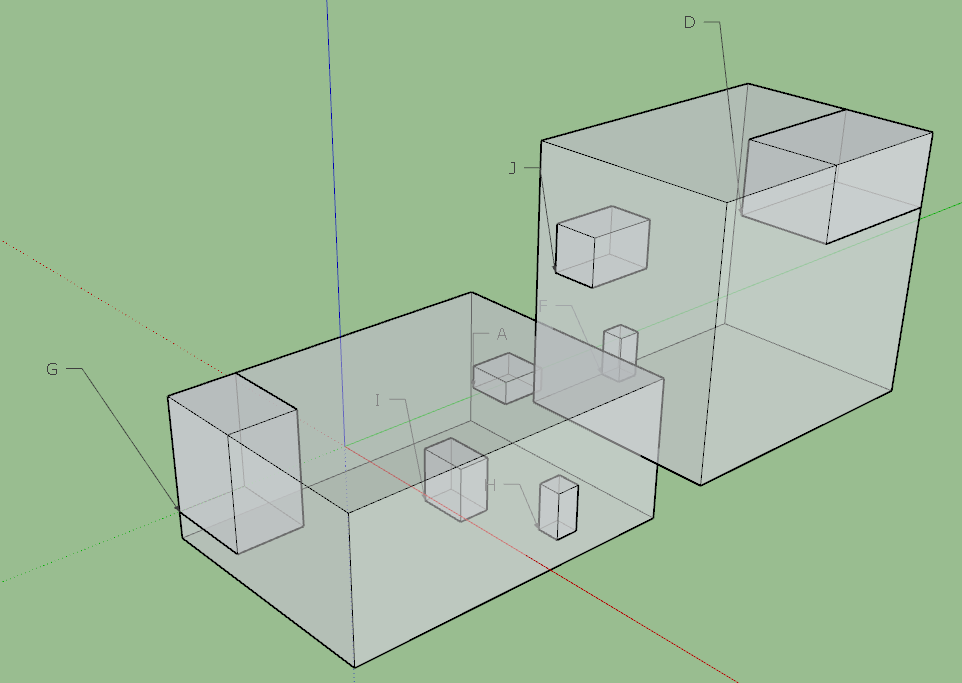


Рисунок 9 - результат удаления узлов в первой тестовой конфигурации

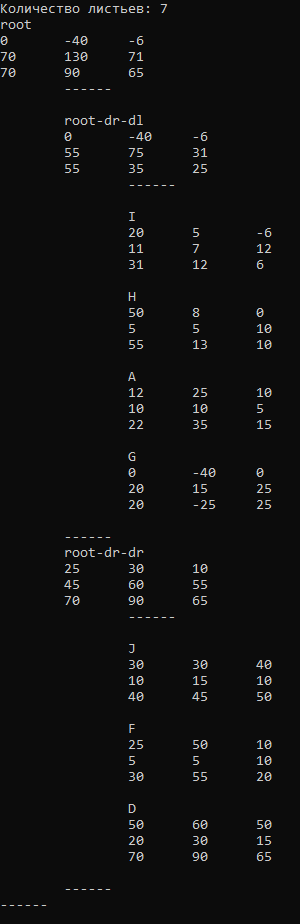


Рисунок 10 – результат удаления узлов в первой тестовой конфигурации

## 3.2. Второй набор данных

### 3.2.1. Построение

Для исследования работы дерева, был сформирован второй набор данных. Как и в первом случае, он формируется автоматизировано в специальной функции (листинг 6).

*Листинг 5*

void secondSet(TDRtree& tree, int n)

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

for (int k = 0; k < n; k++)

{

std::string s = std::to\_string(i) + "-" + std::to\_string(j) + "-" + std::to\_string(k);

tree.add(s, nullptr, Parallelepiped(Point(i \* 100, j \* 100, k \* 100), 30, 30, 30));

}

}

}

std::cout << tree << "\n\n";

}

В данном случае строятся, пусть и упорядоченно, но массово узлы, расположенные в форме «куба». Параметр функции n – ребро этого «куба».

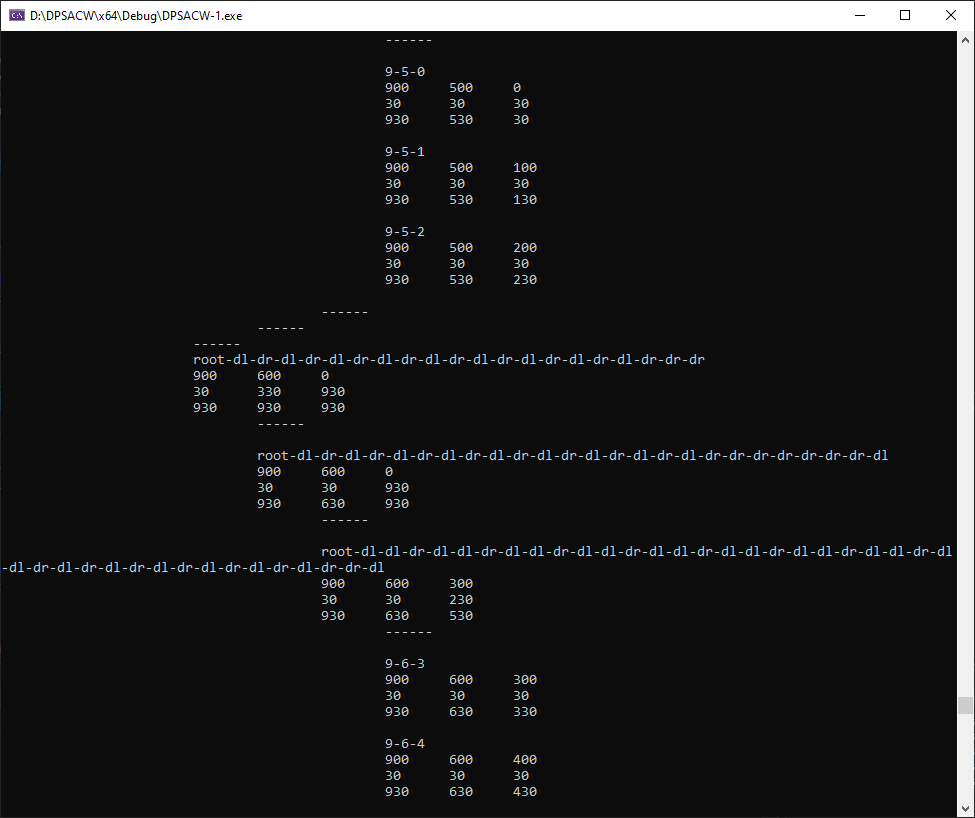


Рисунок 11 – сегмент результата построения дерева по второму набору даных

Данное дерево строилось в пределах 2-3 секунд, однако вывод на экран занял более 10 секунд.

### 3.2.2. Поиск

Данная конфигурация интересна своим количеством элементов, здесь уже можно судить об эффективности алгоритмов поиска (рисунок 12-13):

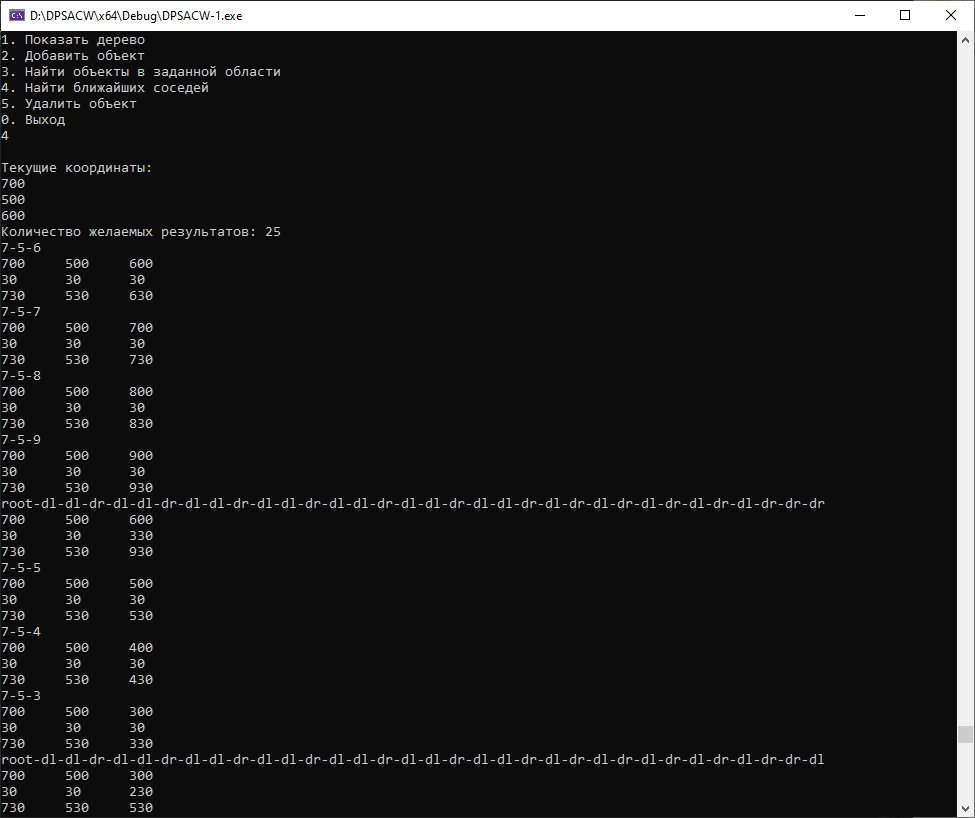


Рисунок 12 – результат поиска в заданной области в во втором наборе тестовых данных

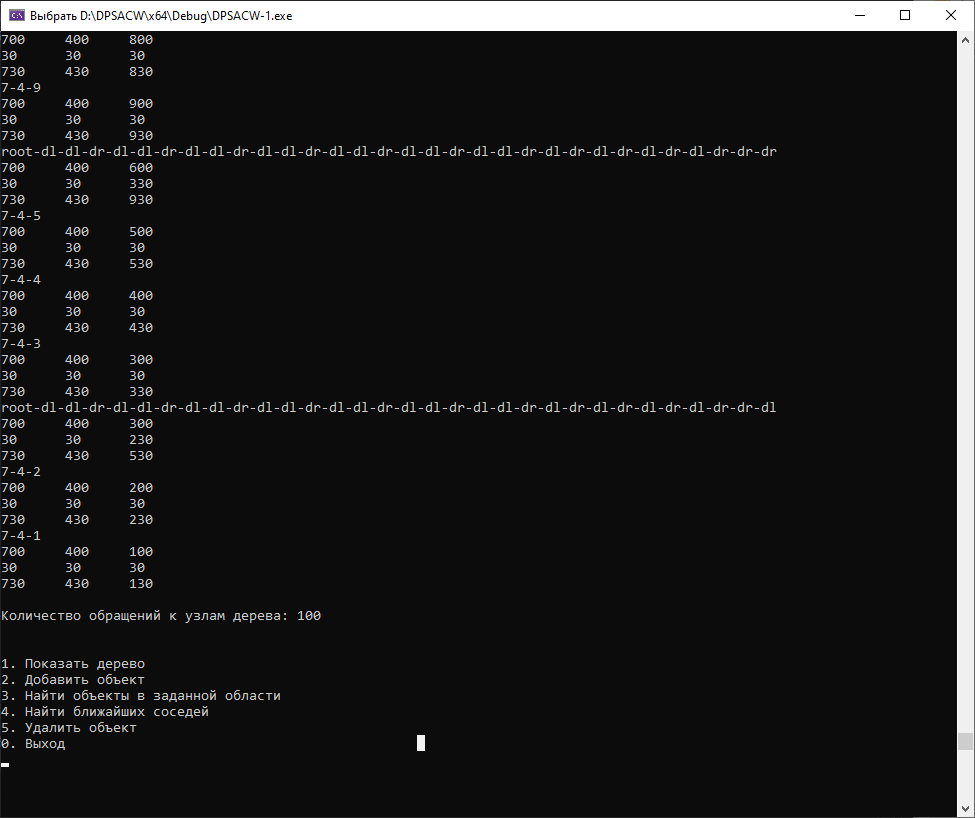


Рисунок 13 – результат поиска в заданной области в во втором наборе тестовых данных

Такое соотношение обращений к узлам дерева и количеству листьев уже гораздо ближе к O(log n). Также следует учесть то, что отладочная переменная, считающая обращения к узлам дерева, кроме листьев считает промежуточные узлы тоже.

Чтобы ещё больше убедиться в эффективности данной структуры данных, был проведён поиск ближайшего соседа (рисунок 14):

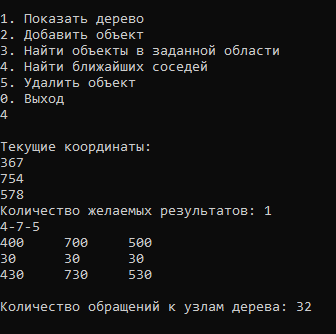


Рисунок 14 – результат поиска ближайшего соседа в во втором наборе тестовых данных

Данная скорость ещё больше похожа на логарифмическую. А значит, что данная структура данных достаточно эффективна в поиске.

### 3.2.3. Удаление

Для демонстрации удаления, при котором произойдёт слияние узлов, были удалены узлы 9-9-6, 9-9-7, 9-9-8 (рисунок 15):

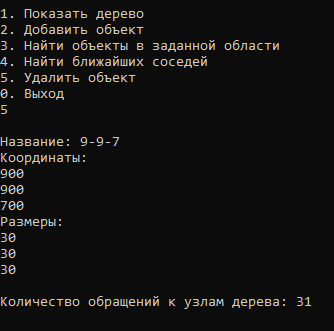


Рисунок 15 – количество обращений к узлам при удалении и слиянии

Резюмируя, можно сказать, что эффективность доступа в такой структуре данных весьма неплохая.

# Заключение

В данной работе было изучена многомерная структура данных 3D R\*-дерево. Были выявлены и рассмотрены его отличительные особенности, алгоритмы построения и доступа к его узлам.

Данная структура данных была реализована на языке C++.

Кроме того, была проанализирована эффективность алгоритма при использовании на различных входных данных и доказана эмпирически его эффективность.

# Список литературы

1. Скворцов А. В: Глобальные алгоритмы построения r-деревьев // 1999, с. 79 - 83.
2. Guttmann A. R-trees: A Dynamic Index Structure For Spatial Searching // Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, p. 47-57.
3. Beckmann N., Kriegel H., Schneider R., Seeger B. The R\* tree: An Efficient and Robust Access Method for Points and Rectangles // ACM, 1990, p. 322-331.
4. Kriegel H., Schiwietz M., Schneider R. Seeger B. Performance comparison of point and spatial access methods // Proc. Symp. On the Design and Implementation of Large Spatial Databases, Santa Barbara, 1989, Lecture Notes in Computer Science.