МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Исследование алгоритма БДП с рандомизацией на вставку и поиск элементов.

Студент гр. 9384 _____ Мосин К.К.

Преподаватель _____ Ефремов М.А.

Санкт-Петербург
2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Мосин К.К.

Группа 9384

Тема работы: **Исследование алгоритма БДП с рандомизацией на вставку и поиск элементов.**

Исходные данные:
Решение разрабатывалось с графической реализацией под фреймворком “Qt”. Для пользователя предоставлен дружеский интерфейс, в котором он может выбрать вероятность распределения(исход рандомизации - равномерно непрерывное, нормальное или несбалансированное)

Содержание пояснительной записки:
Перечисляются требуемые разделы пояснительной записки (обязательны разделы «Содержание», «Введение», «Заключение», «Список использованных источников»)

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания: 01.12.2020

Дата сдачи реферата: 28.12.2020

Дата защиты реферата: 28.12.2020

Студент		Мосин К.К.

Преподаватель		Ефремов М.А.

АННОТАЦИЯ

Для разработки решений использовался фреймворк “Qt”. У пользователя есть возможность выбрать вероятность распределения, влияющая на рандомизацию дерева. При написании кода создается структура данных, представляющее собой бинарное дерево, к которой добавляется метод вставки в корень. Этот метод позволяет с некоторой вероятностью повернуть влево или вправо дерево относительно данного узла. Пользователю будет представлен график со следующими статистическими данными: ось абсцисс представляет количество элементов в дереве, ось ординат необходима для подсчета необходимого времени для поиска самого глубокого элемента, ось аппликат - для подсчета базовых операций алгоритма.

SUMMARY

For the development of solutions, the "Qt" framework was used. The user has the ability to choose the probability of a distribution that affects the randomization of the tree. When you write the code, a data structure is created, which is a binary tree, to which an insert method is added to the root. This method allows, with some probability, to rotate the tree to the left or to the right relative to a given node. The user will be presented with a graph with the following statistical data: the abscissa axis represents the number of elements in the tree, the ordinate axis is needed to calculate the required time to find the deepest element, the applicate axis is used to calculate the basic operations of the algorithm.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: получить практические значения алгоритма и сравнить их с теоретическими.

Для распределения вероятности рандомизации были выбраны следующие три метода:

1. Равномерно непрерывное распределение
2. Нормальное(гауссовское) распределение
3. Несбалансированное(нечестное) распределение

Последнее является частным случаем и представляет собой стандартное бинарное дерево без рандомизации. За счет этого можно проанализировать насколько бинарное дерево поиска с рандомизацией(далее БДПсР) будет превосходить по работоспособности обычное бинарное дерево(далее БД).

1. ОПИСАНИЕ АЛГОРИТМА

1.1. Структура данных

Создается два класса Tree и Node. Первый хранит указатель на корень дерева, когда второй указывает на левые и правые узлы соответственно.

Необходимо хранить информацию о классе Tree для дальнейшей визуализации, поэтому создается класс SurfaceGraph, в котором определяются 3 массива для хранения БДПсР, так как для сравнения были выбраны 3 варианта распределения вероятностей - равномерно непрерывная, нормальная и несбалансированная. Графическая реализация достаточно трудоемкий процесс, поэтому деревья хранятся в классе, непосредственно визуализирующего информацию. Но для корректности статистики присутствует возможность регенерации дерева той или иной вероятности распределения.

1.2. Заполнение массива

Каждое дерево заполняется с pow(2, i + 1) * treeCount количеством элементов, где i >= 0, i <= treeCount и treeCount = 10, поэтому минимальный размер составляет 20 элементов, в то время как максимальный размер равен 10240 элементам.

2. ОПИСАНИЕ СТРУКТУРЫ ДАННЫХ И ФУНКЦИЙ

2.1. Класс Tree

Конструктор обнуляет все поля в списке инициализаций, деструктор высвобождает память, выделенное под корень дерева.

Функция insert представляет собой вставку очередного элемента.

Функция reset необходима для регенерации дерева, где высвобождается память, обнуляется число элементов и задается тип распределения вероятности.

Функции get_count, get_length, get_worst_time являются геттерами числа элементов дерева, длины и времени поиска самого глубокого элемента соответственно

Функции update_length и update_worst_time заполняют поля длины и времени для поиска самого глубокого элемента после создания дерева.

2.2. Класс Node

Типичный класс для работы с деревом. Имеется коструктор, принимающий на вход значение узла и создающий узел, и деструктор, удаляющий, если имеются, левые и правые узлы.

Функция insert - основная функция вставки. Если вводимое значение меньше текущего, то вставляется в левый узел, иначе в правый пока не дойдет до пустого указателся.

Функции get_lenght, get_worst_time, get_depth являются геттерами длинны, худшего времени поиска и глубины дерева соответственно.

Присутствуют приватные методы, такие как insert_root, необходимый для вставки элемента в корень, rotate_left_node и rotate_right_node, предназначенные для поворото вокруг узла после вставки в корень, а также get_weight, возвращающий текущие количество узлов, входящие в данный узел.

2.3. Класс SurfaceGraph

Основной класс, отвечающий за графическую реализацию. Создаются деревья с равномерно непрерывной, нормальной и несбалансированной

вероятностью распределения. В зависимости от выбранной в графической оболочке значения отображается то или иное дерево. При нажатии на кнопку генерации, БД перезаписывается или создается новое.

3. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

3.1. График

Основной атрибут формирования статистики - визуализированные числа.

По оси абсцисс располагается количество элементов в дереве, по оси ординат отображается время затраченное на поиск самого глубокого элемента в секундах, умноженных на 1000, а по оси аппликат - количество базовых операций при обходе всего дерева.

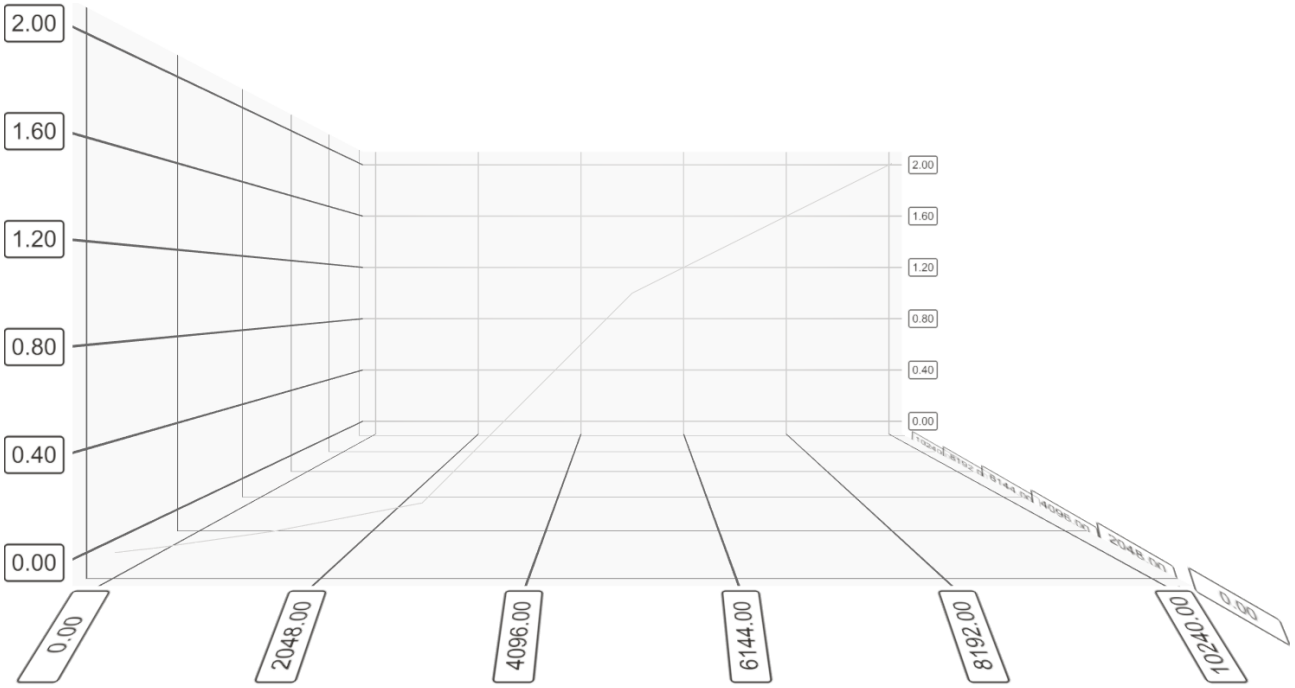
3.2. Взаимодействие с данными

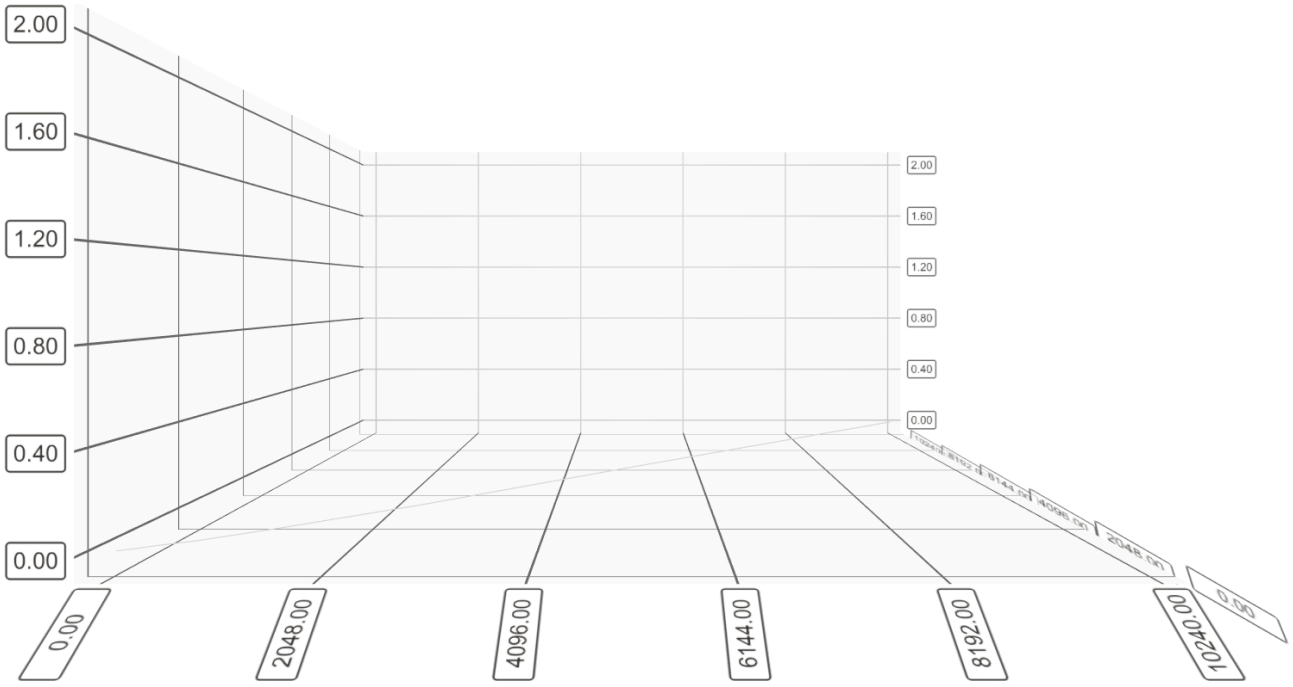
Для пользователя доступны 2 действия - отображение дерева с той или иной вероятностью распределения или регенерация уже имеющегося дерева для усреднения выданных значений. Кнопки с надписями uniform(непрерывное), normal(нормальное) и unbalanced(нечестное) отображают графическую сцену определенного дерева, а кнопка generate(генерация) перезаписывает дерево выбранной сцены.

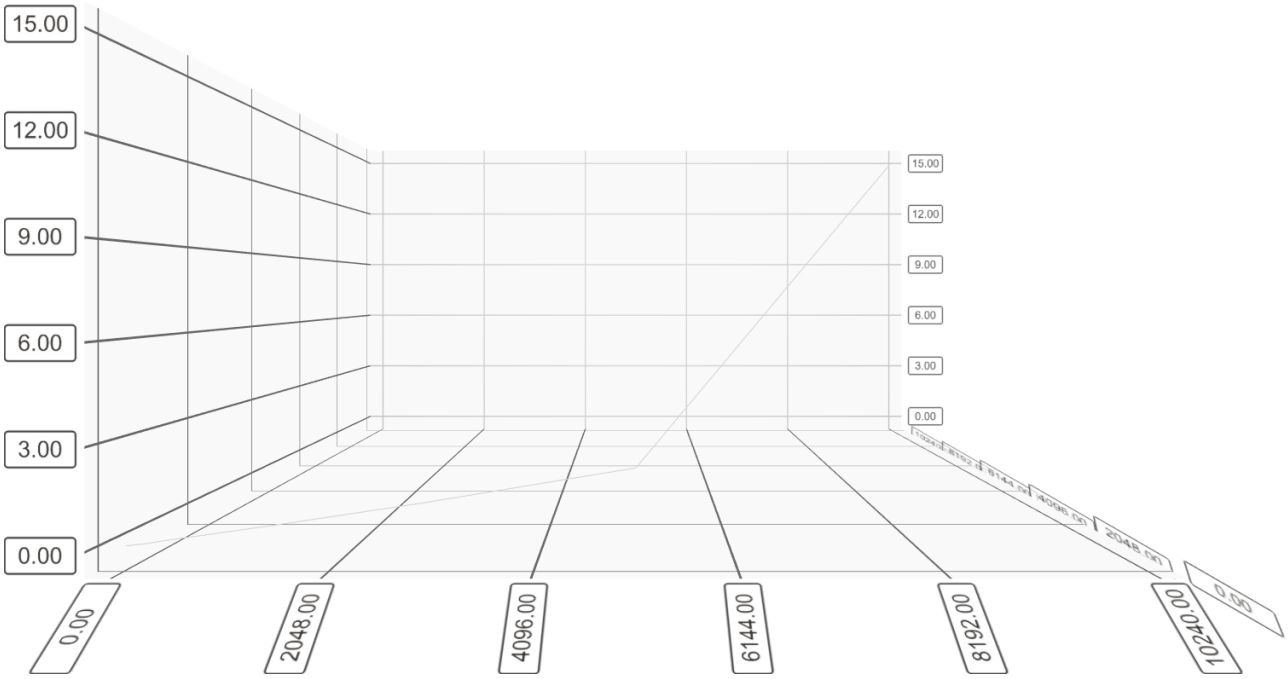
3. ТЕСТИРОВАНИЕ

Для каждой вероятности распределения смоделируем по 3 набора.

Uniform:

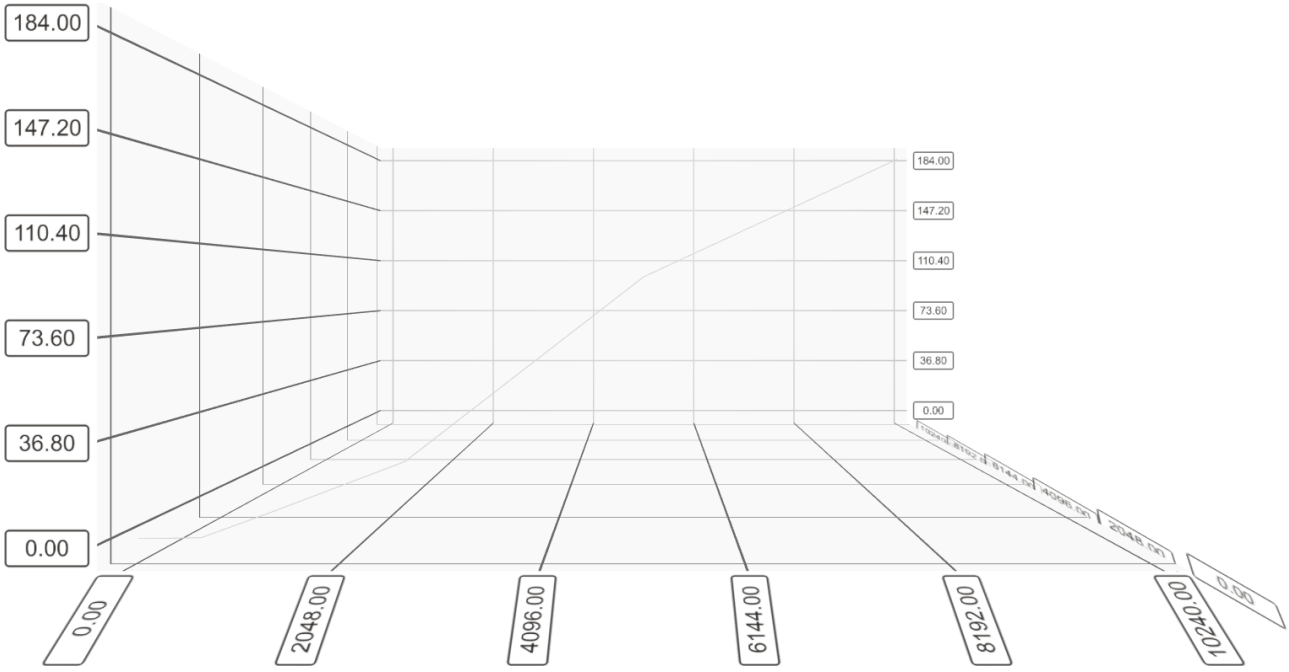


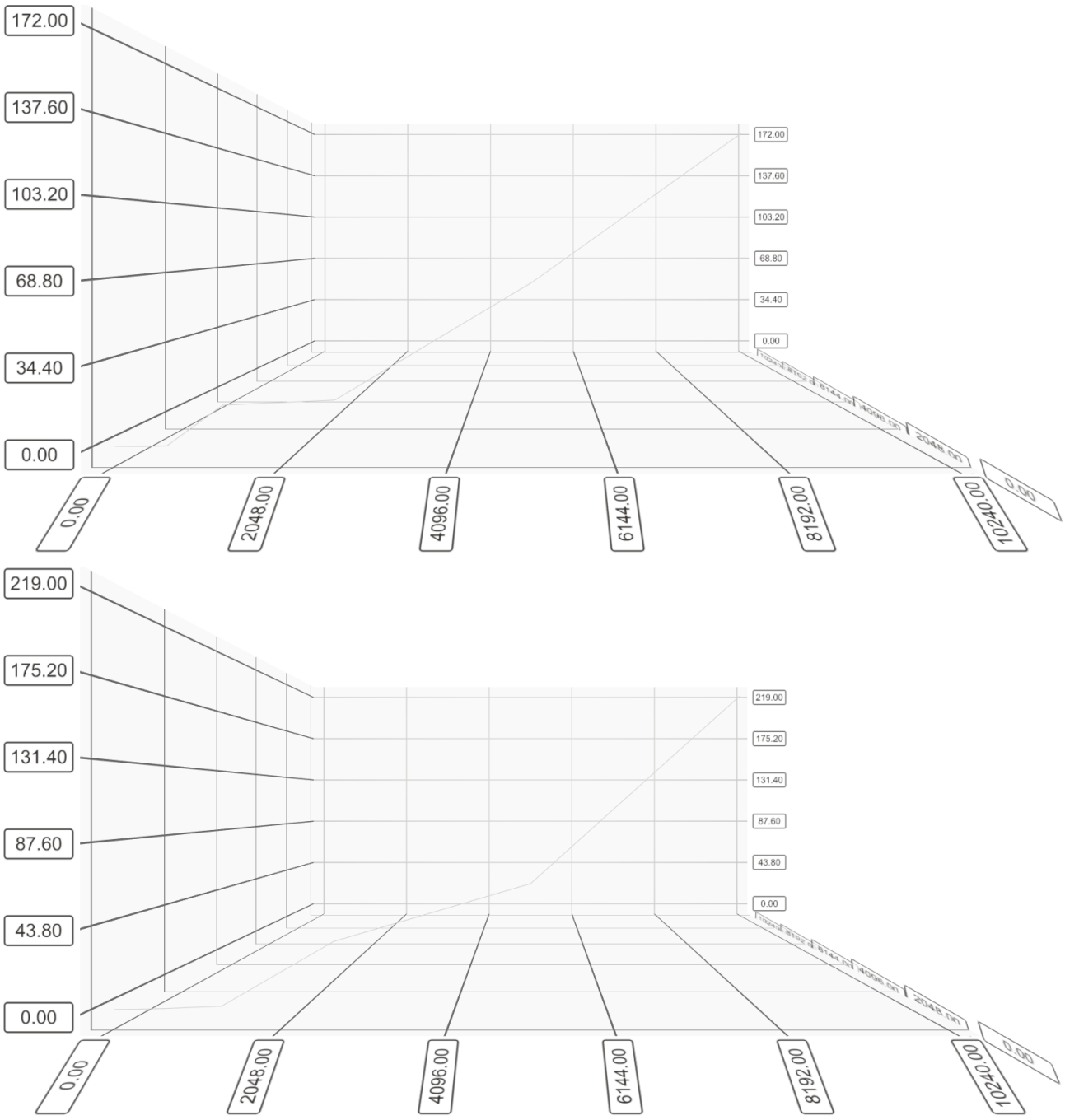




Итог: для 10240 элементов при равномерно непрерывном распределении на поиск самого глубоко элемента затрачивается в среднем по 0.005 секунд.

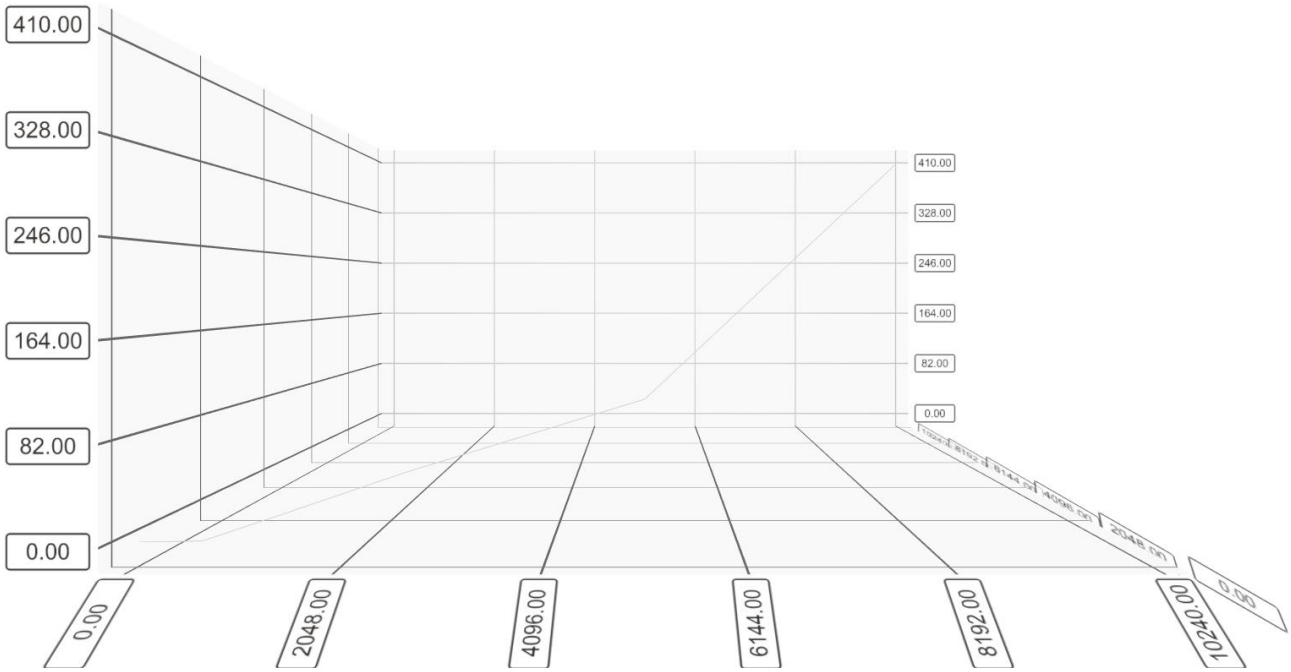
Normal

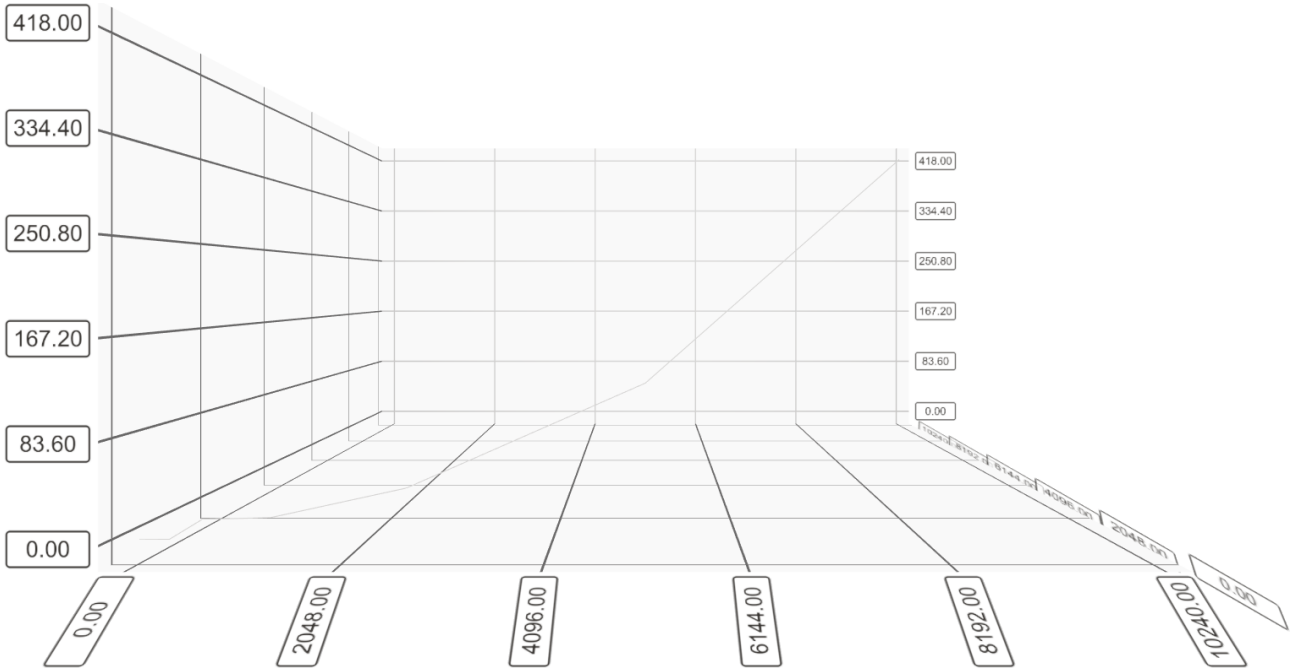


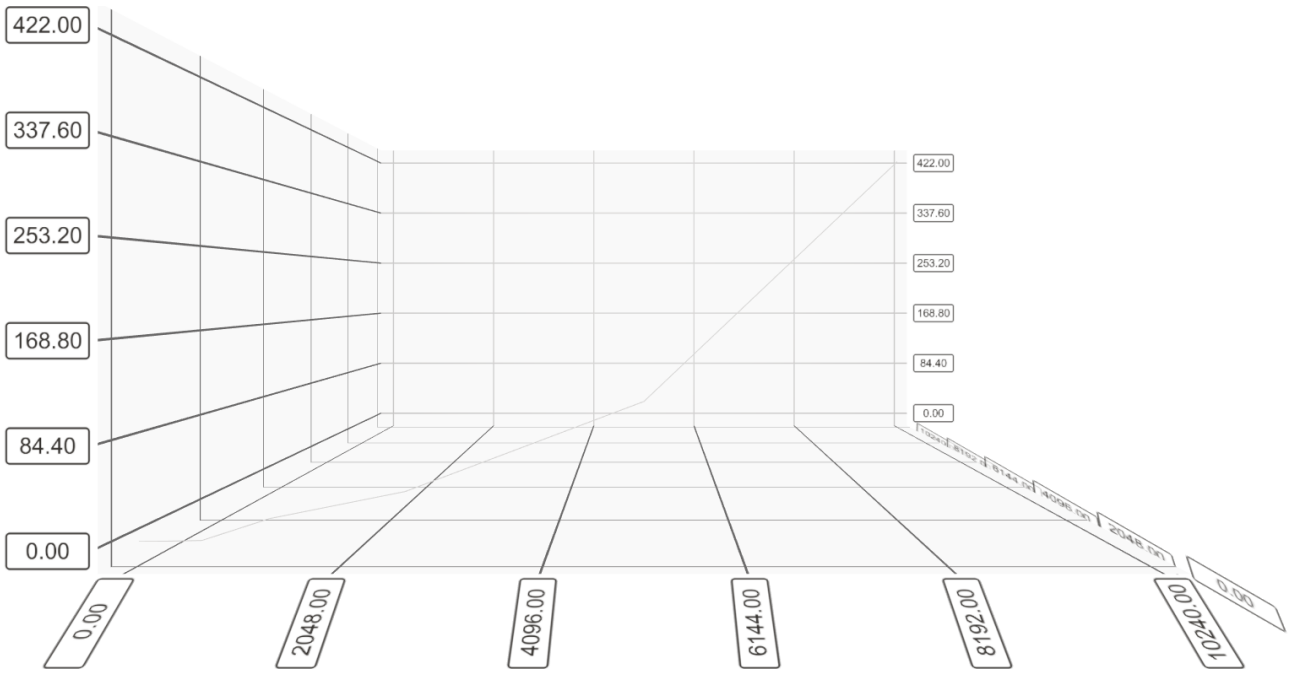


Итог: для 10240 элементов при нормальном распределении на поиск самого глубокого элемента затрачивается в среднем по 0.192 секунды.

Unbalanced


410.00
328.00
246.00
164.00
82.00
0.00
0.00
2048.00
4096.00
6144.00
8192.00
10240.00


418.00
334.40
250.80
167.20
83.60
0.00
0.00
2048.00
4096.00
6144.00
8192.00
10240.00



Итог: для 10240 элементов при нечестном распределении на поиск самого глубокого элемента затрачивается в среднем по 0.417 секунд.

3. ИССЛЕДОВАНИЕ

Для корректного исследования необходимо обращаться к количеству базовых операций, совершенным БД при поиске элемента. При добавлении в дерево элементов в возрастающем порядке, худший результат поиска составит примерно n^2 операций. БДПсР предполагают балансировку дерева, сводя максимальную глубину дерева до $\log_2(N) + 1$, что сведет худшее время поиска до $n\log_2 n$. БДПсР при нечестном распределении является вырожденным деревом и представляет собой обычное БДП. Практические результаты представлены в табл. 1.

Табл. 1

Тип распределения	Число элементов	Количество операций
Равномерная	40	238
	320	574
	1280	16006
Нормальная	40	440
	320	25920
	1280	410880
Нечестная	40	820
	320	51360
	1280	819840

Из табл. 1 при нечестном распределении для 1280 элементов $n^2 = 819840$, откуда $n = 905$, откуда $n\log_2 n = 8888$. На практике выходное значение равно 16000. При подсчете элементарных операций инкрементировалось значение только при переходе к следующему узлу, поэтому операций для $n\log_2 n$ должно быть чуть более чем в 2 раза больше. 8888*2*log2 ~ 12000. На основании результатов можно убедиться, что добавление рандомизации к бинарному дереву сокращает худшее время поиска элемента с $O(n^2)$ до $O(n\log_2 n)$.

ЗАКЛЮЧЕНИЕ

При подводе итогов следует акцентировать внимание, что бинарное дерево поиска с рандомизацией уменьшает высоту дерева вплоть до минимального значения $\log_2 N + 1$, где N количество элементов, что действительно сокращает худшее время поиска элемента, ведь глубина элементов уменьшается.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://doc.qt.io/qt-5/q3dsurface.html

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: node.h

```
#ifndef NODE_H
#define NODE_H

#include <random>

enum class Distribution{
   UNIFORM,
   NORMAL,
   UNBALANCED
};

class Node{
public:
   Node(int data);
   ~Node();
   Node* insert(Node* node, int data, Distribution type = Distribution::NORMAL);
   int get_length(Node *node, int lenght = 0);
   double get_worst_time(Node *node);
   int get_depth(Node *node);

private:
   Node* insert_root(Node* node, int data);
   Node* rotate_left_node(Node* node);
   Node* rotate_right_node(Node* node);
   int get_weight(Node* node);

private:
   int m_data;
```

```
    Node* left_node;
    Node* right_node;

    int weight;

    std::default_random_engine generator_for_normal;

};

#endif // NODE_H
```

Название файла: surfacegraph.h

```
#ifndef SURFACEGRAPH_H
#define SURFACEGRAPH_H

#include <QtWidgets>
#include <QtDataVisualization/Q3DSurface>

#include "tree.h"

using namespace QtDataVisualization;

class SurfaceGraph : public QObject{
    Q_OBJECT
public:
    SurfaceGraph(Q3DSurface *surface);
    ~SurfaceGraph();

    void enable_uniform_distribution(bool enable);
    void enable_normal_distributinon(bool enable);
    void enable_unbalanced_distribution(bool enable);
```

```
    void generate_new_tree();

private:
    void fill_uniform_proxy();
    void fill_normal_proxy();
    void fill_unbalanced_proxy();

private:
    Q3DSurface *m_graph;

    QSurfaceDataProxy *m_uniform_proxy;
    QSurface3DSeries *m_uniform_series;
    bool m_uniform_distribution;

    QSurfaceDataProxy *m_normal_proxy;
    QSurface3DSeries *m_normal_series;
    bool m_normal_distribution;

    QSurfaceDataProxy *m_unbalanced_proxy;
    QSurface3DSeries *m_unbalanced_series;
    bool m_unbalanced_distribution;

    Tree **m_tree_uniform;
    Tree **m_tree_normal;
    Tree **m_tree_unbalanced;
};

#endif // SURFACEGRAPH_H
```

Название файла: tree.h

```
#ifndef TREE_H
```

```
#define TREE_H

#include "node.h"

class Tree{
public:
    Tree();
    ~Tree();
    void insert(int data);
    void reset(Distribution type);
    void update_length();
    int get_count();
    int get_length();
    void update_worst_time();
    double get_worst_time();

private:
    Node *root;
    int count;
    int length;
    double worst_time;
    Distribution m_type;
};

#endif // TREE_H
```

Название файла: main.cpp

```
#include "surfacegraph.h"

#include <QApplication>
```

```
int main(int argc, char *argv[]){
    srand(time(NULL));

    QApplication a(argc, argv);

    Q3DSurface *graph = new Q3DSurface();
    QWidget *container = QWidget::createWindowContainer(graph);
    if (!graph->hasContext()) {
        QMessageBox msgBox;
        msgBox.setText("Couldn't initialize the OpenGL context.");
        msgBox.exec();
        return -1;
    }

    QSize screenSize = graph->screen()->size();
    container->setMinimumSize(QSize(screenSize.width() / 2, screenSize.height() /
1.6));
    container->setMaximumSize(screenSize);
    container->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    container->setFocusPolicy(Qt::StrongFocus);

    QWidget *widget = new QWidget;
    QHBoxLayout *hLayout = new QHBoxLayout(widget);
    QVBoxLayout *vLayout = new QVBoxLayout();
    hLayout->addWidget(container, 1);
    hLayout->addLayout(vLayout);
    vLayout->setAlignment(Qt::AlignTop);

    widget->setWindowTitle(QStringLiteral("cw"));

    QGroupBox *modelGroupBox = new QGroupBox(QStringLiteral("Model"));
```

```
QRadioButton *uniform_distribution = new QRadioButton(widget);
uniform_distribution->setText(QStringLiteral("uniform"));
uniform_distribution->setChecked(false);

QRadioButton *normal_distribution = new QRadioButton(widget);
normal_distribution->setText(QStringLiteral("normal"));
normal_distribution->setChecked(false);

QRadioButton *unbalanced_distribution = new QRadioButton(widget);
unbalanced_distribution->setText(QStringLiteral("unbalanced"));
unbalanced_distribution->setChecked(false);

QPushButton *generate = new QPushButton(widget);
generate->setText(QStringLiteral("generate new tree"));

QVBoxLayout *modelVBox = new QVBoxLayout;
modelVBox->addWidget(uniform_distribution);
modelVBox->addWidget(normal_distribution);
modelVBox->addWidget(unbalanced_distribution);
modelVBox->addWidget(generate);
modelGroupBox->setLayout(modelVBox);

vLayout->addWidget(modelGroupBox);

widget->show();

SurfaceGraph *modifier = new SurfaceGraph(graph);

QObject::connect(uniform_distribution, &QRadioButton::toggled,
                 modifier, &SurfaceGraph::enable_uniform_distribution);
```

```
    QObject::connect(normal_distribution, &QRadioButton::toggled,
                     modifier, &SurfaceGraph::enable_normal_distributinon);
    QObject::connect(unbalanced_distribution, &QRadioButton::toggled,
                     modifier, &SurfaceGraph::enable_unbalanced_distribution);
    QObject::connect(generate, &QPushButton::pressed,
                     modifier, &SurfaceGraph::generate_new_tree);

    return a.exec();
}
```

Название файла: node.cpp

```
#include "node.h"

Node::Node(int data) :m_data(data), left_node(nullptr), right_node(nullptr),
weight(1){}

Node::~Node(){
    if(left_node){
        delete left_node;
    }
    if(right_node){
        delete right_node;
    }
}

Node* Node::insert(Node *node, int data, Distribution type){
    if(!node){
        return new Node(data);
    }

    if(type == Distribution::UNIFORM && rand() % (node->weight + 1) == 0){
```

```
        return insert_root(node, data);
    }

    std::uniform_int_distribution<int> distribution_normal(0, node->weight);
    if(type == Distribution::NORMAL &&
distribution_normal(generator_for_normal)){
        return insert_root(node, data);
    }

    if(type == Distribution::UNBALANCED){
        return insert_root(node, data);
    }

    if(node->m_data > data){
        node->left_node = insert(node->left_node, data, type);
    }
    else{
        node->right_node = insert(node->right_node, data, type);
    }
    node->weight = get_weight(node->left_node) + get_weight(node->right_node) +
1;
    return node;
}

Node* Node::insert_root(Node* node, int data){
    if(!node){
        return new Node(data);
    }
    if(node->m_data > data){
        node->left_node = insert_root(node->left_node, data);
```

```
        node->weight = get_weight(node->left_node) + get_weight(node->right_node)
+ 1;
        return rotate_right_node(node);
    }
    else{
        node->right_node = insert_root(node->right_node, data);
        node->weight = get_weight(node->left_node) + get_weight(node->right_node)
+ 1;
        return rotate_left_node(node);
    }
}

int Node::get_length(Node *node, int length){
    if(node){
        length++;
        return length + get_length(node->left_node, length) + get_length(node-
>right_node, length);
    }
    return 0;
}

#include <ctime>

double Node::get_worst_time(Node *node){
    clock_t start = clock();
    while(node){
        if(node->get_depth(node->left_node) > node->get_depth(node->right_node)){
            node = node->left_node;
        }
        else{
            node = node->right_node;
```

```
        }
    }
    clock_t end = clock();
    return (double)(end - start) / CLOCKS_PER_SEC * 1000;
}

int Node::get_depth(Node *node){
    if(node){
        return std::max(get_depth(node->left_node), get_depth(node->right_node)) + 1;
    }
    else{
        return 0;
    }
}

Node* Node::rotate_left_node(Node* node){
    Node* ptr = node->right_node;
    if(!ptr){
        return node;
    }
    node->right_node = ptr->left_node;
    ptr->left_node = node;
    ptr->weight = node->weight;
    node->weight = get_weight(node->left_node) + get_weight(node->right_node) +
1;
    return ptr;
}

Node* Node::rotate_right_node(Node* node){
    Node* ptr = node->left_node;
    if(!ptr){
```

```
        return node;
    }
    node->left_node = ptr->right_node;
    ptr->right_node = node;
    ptr->weight = node->weight;
    node->weight = get_weight(node->left_node) + get_weight(node->right_node) +
1;
    return ptr;
}

int Node::get_weight(Node* node){
    if(node){
        return node->weight;
    }
    return 0;
}
```

Название файла: surfacegraph.cpp

```
#include "surfacegraph.h"

const int treeCount = 10;

SurfaceGraph::SurfaceGraph(Q3DSurface *surface) : m_graph(surface),
m_uniform_distribution(false), m_normal_distribution(false),
m_unbalanced_distribution(false){
    m_graph->axisX()->setRange(0.0f, 1.0f);
    m_graph->axisY()->setRange(0.0f, 1.0f);
    m_graph->axisZ()->setRange(0.0f, 1.0f);

    m_uniform_proxy = new QSurfaceDataProxy();
    m_uniform_series = new QSurface3DSeries(m_uniform_proxy);
```

```
    m_normal_proxy = new QSurfaceDataProxy();
    m_normal_series = new QSurface3DSeries(m_normal_proxy);

    m_unbalanced_proxy = new QSurfaceDataProxy();
    m_unbalanced_series = new QSurface3DSeries(m_unbalanced_proxy);

    m_tree_uniform = new Tree*[treeCount];
    m_tree_normal = new Tree*[treeCount];
    m_tree_unbalanced = new Tree*[treeCount];
    for(int i = 0; i < treeCount; i++){
        m_tree_uniform[i] = new Tree;
        m_tree_normal[i] = new Tree;
        m_tree_unbalanced[i] = new Tree;
    }
}

SurfaceGraph::~SurfaceGraph(){
    delete m_graph;
    for(int i = 0; i < treeCount; i++){
        delete m_tree_uniform[i];
        delete m_tree_normal[i];
        delete m_tree_unbalanced[i];
    }
    delete[] m_tree_uniform;
    delete[] m_tree_normal;
    delete[] m_tree_unbalanced;
}

void SurfaceGraph::fill_uniform_proxy(){
```

```
    double stepX = m_tree_uniform[treeCount - 1]->get_count() /
(double)(m_tree_uniform[treeCount - 1]->get_count() - 1);
    double stepZ = m_tree_uniform[treeCount - 1]->get_count() /
(double)(m_tree_uniform[treeCount - 1]->get_length() - 1);
    QSurfaceDataArray *dataArray = new QSurfaceDataArray;
    dataArray->reserve(treeCount);
    for (int i = 0 ; i < treeCount ; i++) {
        QSurfaceDataRow *newRow = new QSurfaceDataRow(treeCount);
        float x, y, z;
        if(m_tree_unbalanced[treeCount - 1]->get_count()){
            x = qMin((double)m_tree_unbalanced[treeCount - 1]->get_count(),
m_tree_uniform[i]->get_count() * stepX);
        }
        else{
            x = qMin((double)m_tree_uniform[treeCount - 1]->get_count(),
m_tree_uniform[i]->get_count() * stepX);
        }
        int index = 0;
        for(int j = 0; j < treeCount; j++){
            if(m_tree_unbalanced[treeCount - 1]->get_worst_time()){
                y = qMin(m_tree_unbalanced[treeCount - 1]->get_worst_time(),
m_tree_uniform[i]->get_worst_time());
            }
            else{
                y = qMin(m_tree_uniform[treeCount - 1]->get_worst_time(),
m_tree_uniform[i]->get_worst_time());
            }
            if(m_tree_unbalanced[treeCount - 1]->get_count()){
                z = qMin((double)m_tree_unbalanced[treeCount - 1]->get_count(),
m_tree_uniform[i]->get_length() * stepZ);
            }
```

```
            else{
                z = qMin((double)m_tree_uniform[treeCount - 1]->get_count(),
m_tree_uniform[i]->get_length() * stepZ);
            }
            (*newRow)[index++].setPosition(QVector3D(x, y, z));;
        }
        *dataArray << newRow;
    }
    m_uniform_proxy->resetArray(dataArray);
}

void SurfaceGraph::fill_normal_proxy(){
    double stepX = m_tree_normal[treeCount - 1]->get_count() /
(double)(m_tree_normal[treeCount - 1]->get_count() - 1);
    double stepZ = m_tree_normal[treeCount - 1]->get_count() /
(double)(m_tree_normal[treeCount - 1]->get_length() - 1);
    QSurfaceDataArray *dataArray = new QSurfaceDataArray;
    dataArray->reserve(treeCount);
    for (int i = 0 ; i < treeCount ; i++) {
        QSurfaceDataRow *newRow = new QSurfaceDataRow(treeCount);
        float x, y, z;
        if(m_tree_unbalanced[treeCount - 1]->get_count()){
            x = qMin((double)m_tree_unbalanced[treeCount - 1]->get_count(),
m_tree_normal[i]->get_count() * stepX);
        }
        else{
            x = qMin((double)m_tree_normal[treeCount - 1]->get_count(),
m_tree_normal[i]->get_count() * stepX);
        }
        int index = 0;
        for(int j = 0; j < treeCount; j++){
```

```
            if(m_tree_unbalanced[treeCount - 1]->get_worst_time()){
                y = qMin(m_tree_unbalanced[treeCount - 1]->get_worst_time(),
m_tree_normal[i]->get_worst_time());
            }
            else{
                y = qMin(m_tree_normal[treeCount - 1]->get_worst_time(),
m_tree_normal[i]->get_worst_time());
            }
            if(m_tree_unbalanced[treeCount - 1]->get_count()){
                z = qMin((double)m_tree_unbalanced[treeCount - 1]->get_count(),
m_tree_normal[i]->get_length() * stepZ);
            }
            else{
                z = qMin((double)m_tree_normal[treeCount - 1]->get_count(),
m_tree_normal[i]->get_length() * stepZ);
            }
            (*newRow)[index++].setPosition(QVector3D(x, y, z));;
        }
        *dataArray << newRow;
    }
    m_normal_proxy->resetArray(dataArray);
}

void SurfaceGraph::fill_unbalanced_proxy(){
    double stepX = m_tree_unbalanced[treeCount - 1]->get_count() /
(double)(m_tree_unbalanced[treeCount - 1]->get_count() - 1);
    double stepZ = m_tree_unbalanced[treeCount - 1]->get_count() /
(double)(m_tree_unbalanced[treeCount - 1]->get_length() - 1);
    QSurfaceDataArray *dataArray = new QSurfaceDataArray;
    dataArray->reserve(treeCount);
    for (int i = 0 ; i < treeCount ; i++) {
```

```
        QSurfaceDataRow *newRow = new QSurfaceDataRow(treeCount);
        float x = qMin((double)m_tree_unbalanced[treeCount - 1]->get_count(),
m_tree_unbalanced[i]->get_count() * stepX);
        int index = 0;
        for(int j = 0; j < treeCount; j++){
            float y = qMin(m_tree_unbalanced[treeCount - 1]->get_worst_time(),
m_tree_unbalanced[i]->get_worst_time());
            float z = qMin((double)m_tree_unbalanced[treeCount - 1]->get_count(),
m_tree_unbalanced[i]->get_length() * stepZ);
            (*newRow)[index++].setPosition(QVector3D(x, y, z));;
        }
        *dataArray << newRow;
    }
    m_unbalanced_proxy->resetArray(dataArray);
}

void SurfaceGraph::enable_uniform_distribution(bool enable){
    if(enable){
        m_uniform_distribution = true;
        m_normal_distribution = false;
        m_unbalanced_distribution = false;

        m_normal_series-
>setDrawMode(QSurface3DSeries::DrawSurfaceAndWireframe);
        m_normal_series->setFlatShadingEnabled(true);

        m_graph->removeSeries(m_normal_series);
        m_graph->removeSeries(m_unbalanced_series);
        m_graph->addSeries(m_uniform_series);

        if(m_tree_unbalanced[treeCount - 1]->get_count() > 1.0f){
```

```
            m_graph->axisX()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_count());
            m_graph->axisZ()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_count());
        }
        else if(m_tree_uniform[treeCount - 1]->get_count() > 1.0f){
            m_graph->axisX()->setRange(0.0f, m_tree_uniform[treeCount - 1]-
>get_count());
            m_graph->axisZ()->setRange(0.0f, m_tree_uniform[treeCount - 1]-
>get_count());
        }
        if(m_tree_unbalanced[treeCount - 1]->get_worst_time() > 1.0f){
            m_graph->axisY()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_worst_time());
        }
        else if(m_tree_uniform[treeCount - 1]->get_worst_time() > 1.0f){
            m_graph->axisY()->setRange(0.0f, m_tree_uniform[treeCount - 1]-
>get_worst_time());
        }
        m_graph->axisX()->setLabelAutoRotation(30);
        m_graph->axisY()->setLabelAutoRotation(90);
        m_graph->axisZ()->setLabelAutoRotation(30);
    }
}

#include <iostream>

void SurfaceGraph::enable_normal_distributinon(bool enable){
    if(enable){
        m_uniform_distribution = false;
        m_normal_distribution = true;
```

```
    m_unbalanced_distribution = false;

    m_normal_series-
>setDrawMode(QSurface3DSeries::DrawSurfaceAndWireframe);
    m_normal_series->setFlatShadingEnabled(true);

    m_graph->removeSeries(m_uniform_series);
    m_graph->removeSeries(m_unbalanced_series);
    m_graph->addSeries(m_normal_series);

    if(m_tree_unbalanced[treeCount - 1]->get_count() > 1.0f){
        m_graph->axisX()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_count());
        m_graph->axisZ()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_count());
    }
    else if(m_tree_normal[treeCount - 1]->get_count() > 1.0f){
        m_graph->axisX()->setRange(0.0f, m_tree_normal[treeCount - 1]-
>get_count());
        m_graph->axisZ()->setRange(0.0f, m_tree_normal[treeCount - 1]-
>get_count());
    }
    if(m_tree_unbalanced[treeCount - 1]->get_worst_time() > 1.0f){
        m_graph->axisY()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_worst_time());
    }
    else if(m_tree_normal[treeCount - 1]->get_worst_time() > 1.0f){
        m_graph->axisY()->setRange(0.0f, m_tree_normal[treeCount - 1]-
>get_worst_time());
    }
    m_graph->axisX()->setLabelAutoRotation(30);
```

```
        m_graph->axisY()->setLabelAutoRotation(90);
        m_graph->axisZ()->setLabelAutoRotation(30);
    }
}

void SurfaceGraph::enable_unbalanced_distribution(bool enable){
    if(enable){
        m_uniform_distribution = false;
        m_normal_distribution = false;
        m_unbalanced_distribution = true;

        m_normal_series-
>setDrawMode(QSurface3DSeries::DrawSurfaceAndWireframe);
        m_normal_series->setFlatShadingEnabled(true);

        m_graph->removeSeries(m_uniform_series);
        m_graph->removeSeries(m_normal_series);
        m_graph->addSeries(m_unbalanced_series);

        if(m_tree_unbalanced[treeCount - 1]->get_count() > 1.0f){
            m_graph->axisX()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_count());
            m_graph->axisZ()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_count());
        }
        if(m_tree_unbalanced[treeCount - 1]->get_worst_time() > 1.0f){
            m_graph->axisY()->setRange(0.0f, m_tree_unbalanced[treeCount - 1]-
>get_worst_time());
        }
        m_graph->axisX()->setLabelAutoRotation(30);
        m_graph->axisY()->setLabelAutoRotation(90);
```

```
        m_graph->axisZ()->setLabelAutoRotation(30);
    }
}

void SurfaceGraph::generate_new_tree(){
    if(m_uniform_distribution){
            for(int i = 0; i < treeCount; i++){
                m_tree_uniform[i]->reset(Distribution::UNIFORM);
                for(int j = 0; j < pow(2, i + 1) * treeCount; j++){
                    m_tree_uniform[i]->insert(j);
                }
                m_tree_uniform[i]->update_length();
                m_tree_uniform[i]->update_worst_time();
//                std::cout << m_tree_uniform[i]->get_count() << " " <<
m_tree_uniform[i]->get_length() << std::endl;
            }
            fill_uniform_proxy();
            enable_uniform_distribution(true);
        }
    else if(m_normal_distribution){
        for(int i = 0; i < treeCount; i++){
            m_tree_normal[i]->reset(Distribution::NORMAL);
            for(int j = 0; j < pow(2, i + 1) * treeCount; j++){
                m_tree_normal[i]->insert(j);
            }
            m_tree_normal[i]->update_length();
            m_tree_normal[i]->update_worst_time();
//            std::cout << m_tree_normal[i]->get_count() << " " << m_tree_normal[i]-
>get_length() << std::endl;
        }
        fill_normal_proxy();
```

```
        enable_normal_distributinon(true);
    }
    else if(m_unbalanced_distribution){
        for(int i = 0; i < treeCount; i++){
            m_tree_unbalanced[i]->reset(Distribution::UNBALANCED);
            for(int j = 0; j < pow(2, i + 1) * treeCount; j++){
                m_tree_unbalanced[i]->insert(j);
            }
            m_tree_unbalanced[i]->update_length();
            m_tree_unbalanced[i]->update_worst_time();
//            std::cout << m_tree_unbalanced[i]->get_count() << " " <<
m_tree_unbalanced[i]->get_length() << std::endl;
        }
        fill_unbalanced_proxy();
        enable_unbalanced_distribution(true);
    }
}
```

Название файла: tree.cpp

```
#include "tree.h"

Tree::Tree() : root(nullptr), count(0), length(0), worst_time(0) {}

Tree::~Tree(){
    if(root){
        delete root;
    }
}

void Tree::insert(int data){
    root = root->insert(root, data, m_type);
```

```
    count++;
}

void Tree::reset(Distribution type){
    if(root){
        delete root;
    }
    root = nullptr;
    count = 0;
    m_type = type;
}

void Tree::update_length(){
    length = root->get_length(root);
}

int Tree::get_count(){
    return count;
}

int Tree::get_length(){
    return length;
}

void Tree::update_worst_time(){
    worst_time = root->get_worst_time(root);
}

double Tree::get_worst_time(){
    return worst_time;
}
```