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 операции вставки в красно-черных деревьях

Студент гр. 9304 _____ Кузнецов Р.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ (КУРСОВОЙ ПРОЕКТ)

Студент Кузнецов Р.В.

Группа 9304

Тема работы: Исследование операции вставки в красно-черных деревьях

Исходные данные:

Содержание пояснительной записки:

- Аннотация
- Содержание
- Введение
- Формальная постановка задачи
- Описание структур данных и функций, описание алгоритма
- Тестирование
- Исследование
- Заключение
- Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 25 страниц.

Дата выдачи задания: 23.11.2020

Дата сдачи реферата: 26.12.2020

Дата защиты реферата: 27.12.2020

Студент _____ Кузнецов Р.В.

Преподаватель _____ Филатов А.Ю.

АННОТАЦИЯ

В данной курсовой работе производится исследование операции вставки в структуру данных Красно-черное дерево. Исследование производится с помощью тестов на сгенерированном множестве входных данных для лучшего, среднего и худшего случаев работы алгоритма. Результатами исследования являются числовые метрики, на основе которых формируются графики для сравнения с теоретическими оценками.

SUMMARY

In this course work, a study is made of the operation of insertion into the data structure named Red-black tree. The research is performed using tests on the generated set of input data for the best, average and worst cases algorithm performance. The research results are numerical metrics, on the basis of which graphs are formed for comparison with theoretical estimates.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: Изучить структуру данных Красно-черное дерево. Реализация и экспериментальное машинное исследование алгоритма вставки в Красно-черное дерево.

Задача: Реализовать программу, которая добавляет в Красно-черное дерево массив введенных пользователем данных и выводит его на экран.

1. ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Реализовать структуру данных Красно-черное дерево и провести исследование работы операций вставки в различных случаях чтобы проверить теоретическую оценку работы этой операции.

2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

2.1. Описание алгоритма

Алгоритм добавления элемента в Красно-черное дерево:

Спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения вставляемых данных и данных в текущем узле.

При достижении момента, когда нужно выбрать несуществующее поддерево – создаем его в этом месте с вставляемыми данными. Затем происходит балансировка дерева для сохранения следующих свойств дерева:

- Узел может быть либо красным, либо чёрным и имеет двух потомков;
- Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
- Все листья, не содержащие данных — чёрные.
- Оба потомка каждого красного узла — чёрные.
- Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Таким образом, балансировка может быть выполнена следующим образом:

1. Если текущий узел в корне дерева, он перекрашивается в черный цвет, балансировка завершается.
2. Если предок текущего узла черный, свойства не нарушаются, балансировка завершается.
3. Если родитель и дядя красные, то они перекрашиваются в черный, дедушка – в красный. Переходим к первому пункту.
4. Если родитель является красным, а дядя – черным, текущий узел – правый потомок, а родитель – левый потомок, поворачиваем дерево, относительно бывшего родителя переходим к пункту 5.

5. Если родитель красный, а дядя – черный, текущий узел – левый потомок, родитель – левый потомок, поворачиваем дерево относительно деда, балансировка дерева завершается.

2.2. Описание структур данных и методов

1. struct RBNode – структурная единица дерева. Исходный код в приложении А.
    a. Поля:
        i. RBNodeWptr<T>parent – слабый указатель на родителя.
        ii. RBNodeSptr<T>left, right – коллективный указатель на левое и правое поддерево.
        iii. T data – поле данных.
        iv. RBColor color – цвет узла.
2. class RBTree – класс Красно-черного дерева. Исходный код в приложении Б.
    a. Поля:
        i. RBNodeSptr<T> root – указатель на корень дерева
        ii. size_t treeSize – количество узлов в дереве.
        iii. int op_counter – счетчик операций.
    b. Методы:
        i. size_t size() – возвращает размер дерева.
        ii. int insert(T value) – метод вставки в дерево. Возвращает количество совершенных операций
        iii. void clear() – метод очищения дерева.
        iv. void outputSorted(ostream&) – выводит отсортированный массив в данный поток.
        v. void outputLayers(ostream&) – выводит послойно дерево в данный поток.
        vi. void RRight(RBNodeSptr<T>) – выполняет правый поворот дерева.

vii. void RLeft(RBNodeSptr<T>) – выполняет левый поворот дерева.
viii. void insert_1(RBNodeSptr<T>) – вспомогательная функция вставки. Нужна для балансировки дерева.
ix. void insert_2(RBNodeSptr<T>) – вспомогательная функция вставки. Нужна для балансировки дерева.

3. class RBTester – класс для проведения исследования. Исходный код в приложении В.
    a. Поля:
        i. RBTree<int>&tree – ссылка на красно-черное дерево.
    b. Методы
        i. void insertBest() – генерирует лучший случай для дерева и вставляет сгенерированный набор элементов в это дерево, выводя в файл insertBest.txt размер дерева, в который добавляется элемент, и количество совершенных операций.
        ii. void insertAverage() – генерирует средний случай для дерева и вставляет сгенерированный набор элементов в это дерево, выводя в файл insertAverage.txt размер дерева, в который добавляется элемент, и количество совершенных операций.
        iii. void insertWorst() – генерирует худший случай для дерева и вставляет сгенерированный набор элементов в это дерево, выводя в файл insertWorst.txt размер дерева, в который добавляется элемент, и количество совершенных операций.
4. Функции. Исходный код в приложении А.
    a. RBNodeSptr<T> grandparent(RBNodeSptr<T>) – возвращает деда узла.
    b. RBNodeSptr<T> uncle(RBNodeSptr<T>) – возвращает дядю узла.

3. ТЕСТИРОВАНИЕ

Входные данные: на вход программе подаются Красно-черные элементы в строке, разделенные пробелом.

Выходные данные: программа выводит в файл res.txt элементы в отсортированном порядке, а на экран – иерархическое представление красно-черного дерева с узлами, раскрашенными в соответствующий цвет.

Тестирование производится с помощью скрипта test.py, написанного на языке Python. Исходный код test.py представлен в приложении Е. Для удобства запуска был написан Makefile. Исходный код Makefile представлен в приложении Ж. Для запуска тестов, находящихся в папке Tests, необходимо написать make test. Данная команда скомпилирует версию программы для тестов, и проверит результат на валидность, после чего удалит созданные файлы. Результат запуска данной команды представлен на рисунке 1. Результаты тестирования представлены в таблице 1.

Таблица 1 – результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1	1 3 4 6 5 9	1 3 4 5 6 9	Случайный набор
2	-7 -6 -5 -4 -3 -2 -1 0	-7 -6 -5 -4 -3 -2 -1 0	Отрицательные значения
3	7 8 2 4	2 4 7 8	Нормальная работа программы
4	20 25 6 16 30 29 23 19 7 21 28 2 15 13 10 14 31 8 1 12 4 32 9 11 18 22 26 27 5 3 24 17	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	Перемешаны все числа от 1 до 32
5	1	1	Один элемент
6	-4 1 -2 -3 4 3 0 -5 -1 2 5	-5 -4 -3 -2 -1 0 1 2 3 4 5	Смешанные значения
7	-341 42 245 531 10 -124 952 -154 262 52 2345 14 351	-341 -154 -124 10 14 42 52 245 262 351 531 952 2345	Большие элементы
8	7 6 5 4 3 2 1 0 -1	-1 0 1 2 3 4 5 6 7	Элементы отсортированы
9	1 2 3	1 2 3	Элементы изначально отсортированы
10	8 7 6 5 4 3 2 1	1 2 3 4 5 6 7 8	Элементы обратно отсортированы

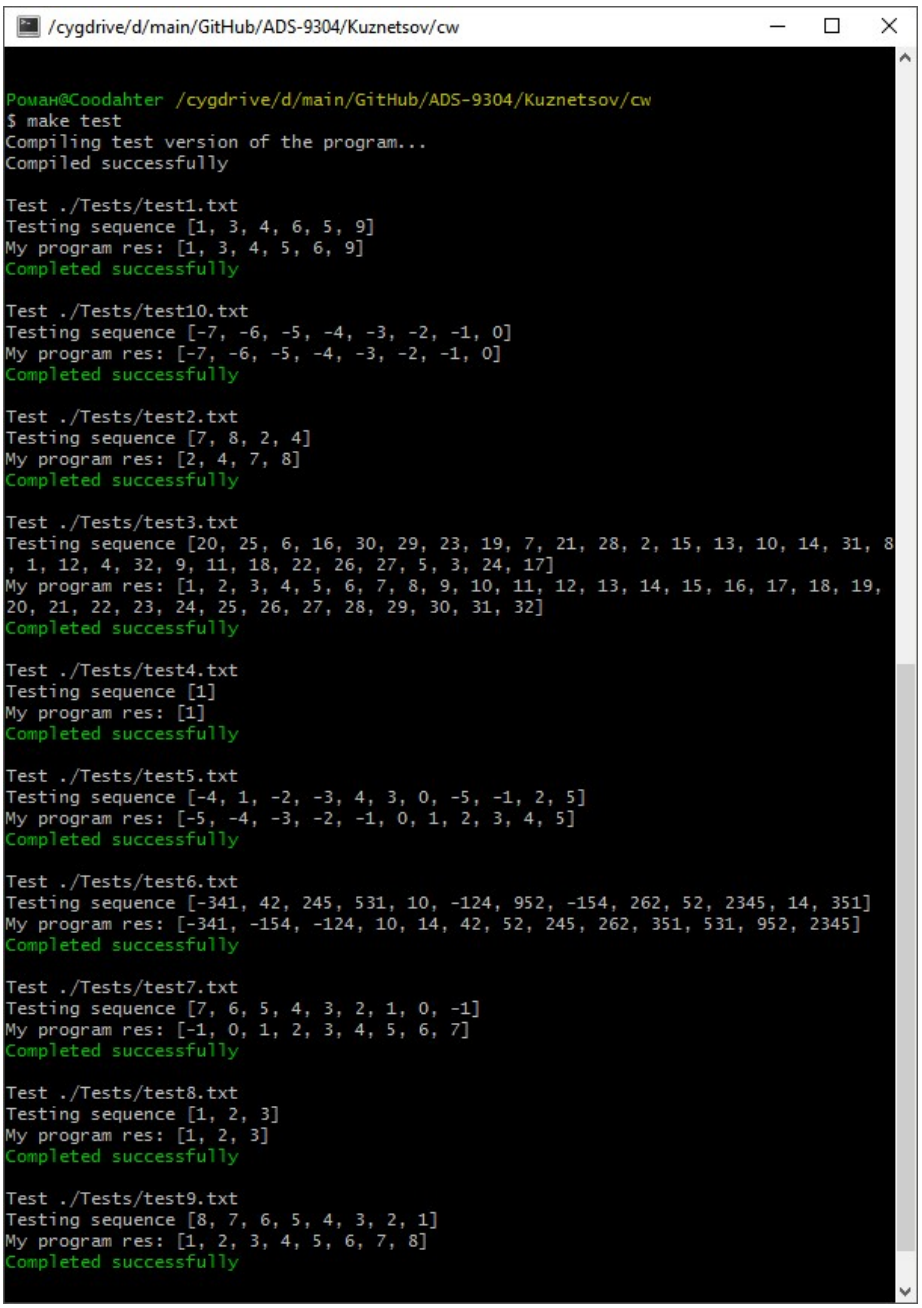

Рисунок 1 – пример запуска тестирующего скрипта.

4. ИССЛЕДОВАНИЕ

4.1. Исследование вставки в Красно-черное дерево

Рассмотрим операцию вставки в Красно-черное дерево. Для самой операции вставки нам потребуется O(h) операций, где h – высота дерева. Причем так как Красно-черное дерево гарантирует, что разница между самым коротким и самым длинным расстояниями от корня до листа не больше, чем в 2 раза (Для понимания этого достаточно рассмотреть 4 и 5 свойства дерева: так как оба потомка красного узла – черные, а любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов, путь может быть увеличен только за счет вставления красных узлов, после которых обязательно должен идти черный. Тогда кратчайший путь содержит B узлов и все они черные, а самый длинный путь состоит из 2B-1 последовательно черных и красных узлов), h = lb(n). После вставки элемента, Красно-черное дерево нужно сбалансировать, обеспечив сложность O(lb(n)) следующим вставкам. Операция балансировки занимает в худшем случае O(lb(n)) на основании предыдущих соображений и на том, что при балансировке выполняется не более одного поворота. Таким образом, сложность операции вставки составляет O(lb(n) + lb(n)) = O(lb(n)).

Асимптотика Красно-черного дерева представлена в таблице 2.

Таблица 2 – Асимптотика Красно-черного дерева

	В лучшем случае	В худшем случае
Расход памяти	O(n)	O(n)
Поиск	O(lb n)	O(lb n)
Вставка	O(lb n)	O(lb n)
Удаление	O(lb n)	O(lb n)

4.2. План исследования

План исследования состоит из следующих шагов:

1. Генерация входных данных с учетом различных особенностей распределения
2. Запуск функции вставки элемента в Красно-черное дерево с подсчетом количества операций
3. Анализ и интерпретация результатов

4.3. Генерация входных данных

Для генерации входных данных был создан класс RBTester, который с помощью функций (insertBest, insertWorst, insertAverage) генерирует входные данные соответственно трех типов: лучшей, худшей и средней последовательностей, и вставляет их в Красно-черное дерево, фиксируя количество совершенных операций при каждом размере дерева.

Лучшая последовательность построена таким образом, чтобы дерево не пришлось балансировать ни разу, а элементы просто послойно присоединяются к дереву. Для генерации такой последовательности было взято число (2^19), которое должно быть постоянным корнем дерева. Затем к нему прибавляется/вычитается 2^18. Это его левый и правые поддеревья. К этим двум числам вычитаются/прибавляются 2^17. Это левое поддерево левого поддерева, правое поддерево левого поддерева, левое поддерево правого поддерева и правое поддерево правого поддерева соответственно, и так до тех пор, пока не будут вставлены все числа от 1 до 2^20. Сложность вставки элемента в Красно-черное дерево в данном случае отражена на рисунке 2.

Рисунок 2 – Количество совершенных операций в лучшем случае

Генерация худшей последовательности для вставки тривиальна и заключается во вставке последовательности от 1 до 2^20 по возрастающей. В таком случае балансировать дерево придется чаще всего, так как каждый вставляемый элемент больше всех в дереве, и новый элемент вставляется на место правого поддерева правого поддерева правого поддерева… Сложность вставки в Красно-черное дерево в худшем случае представлена на рисунке 3.

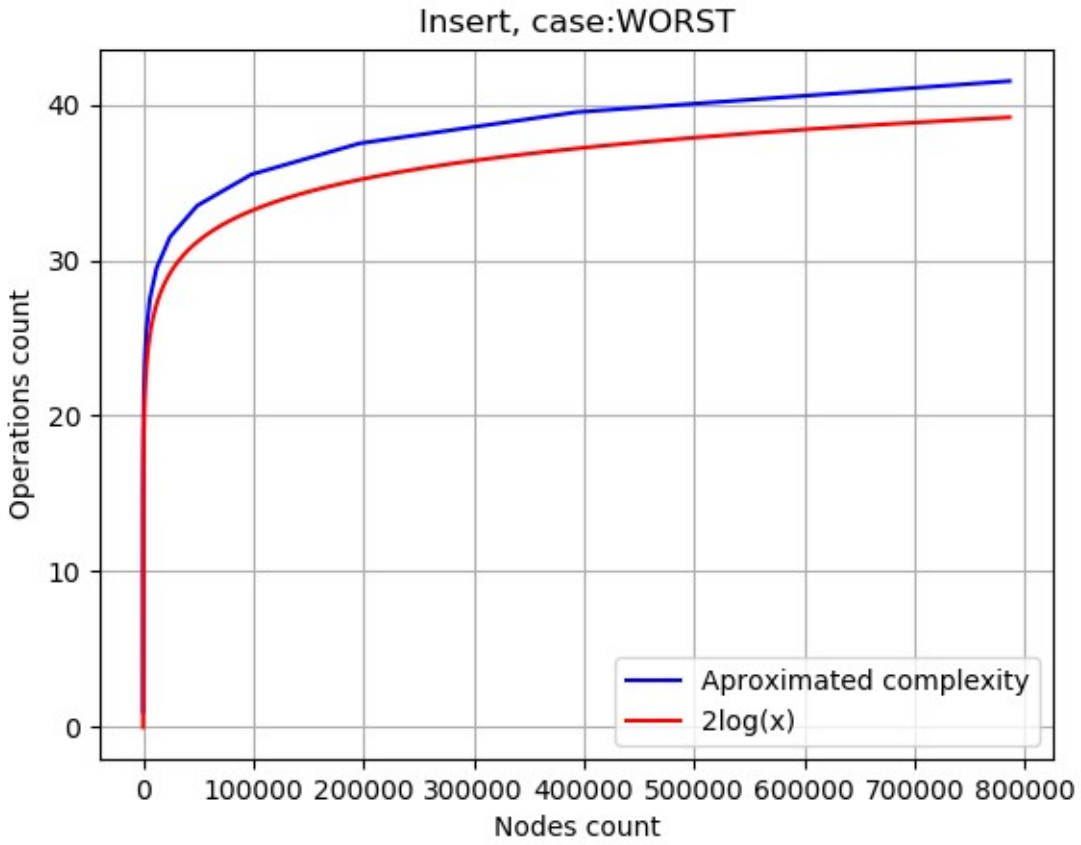

Рисунок 3 – Количество совершенных операций в худшем случае

Для наглядности, в этом случае красная линия отражает удвоенный бинарный логарифм от количества узлов. Как можно заметить, рассчитанная сложность больше ровно в два раза.

Генерация последовательности среднего случая для вставки в Красно-черное дерево заключается в создании вектора значений от 1 до 2^20 и последующим перемешиванием элементов. Перемешивание реализовано с помощью библиотеки random языка C++ и встроенной функции std::shuffle. Сложность вставки в Красно-черное дерево в среднем случае представлена на рисунке 4.

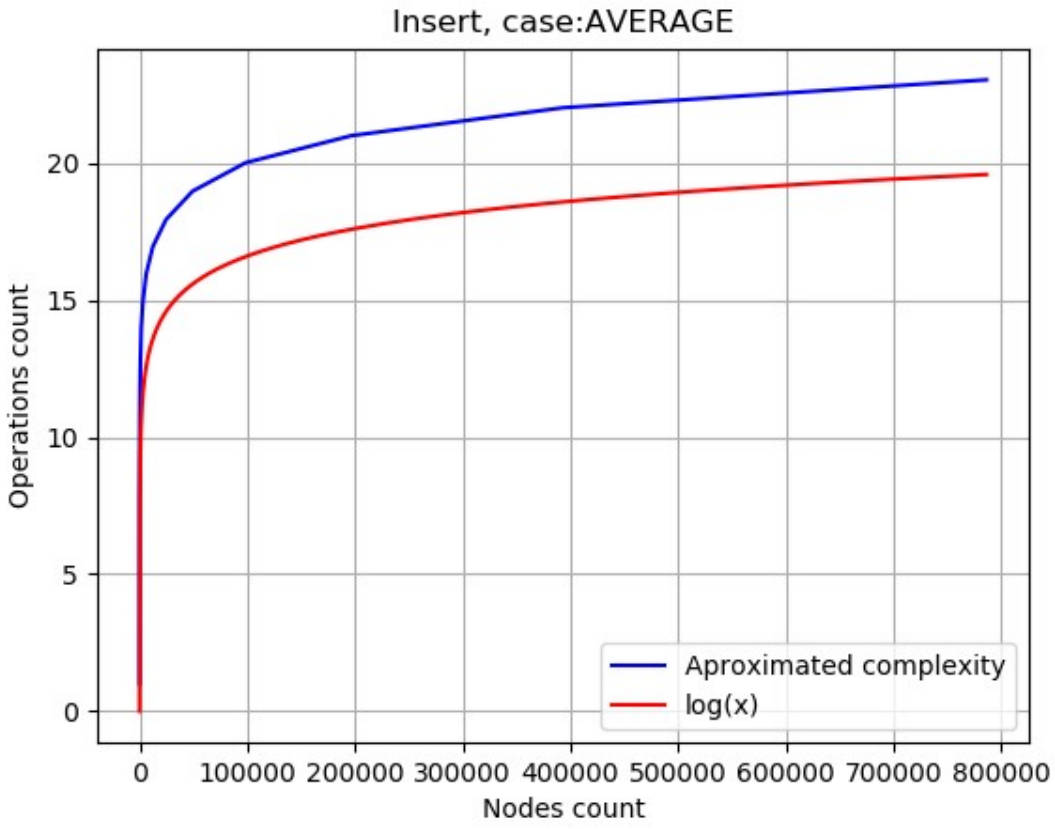

Рисунок 4 – Количество совершенных операций в среднем случае

4.4. Результаты исследования

На основе числовых метрик были построены графики. На всех графиках красной линией построен график бинарного логарифма от количества элементов в дереве, синей линией – график количества операций при вставке в дерево от количества элементов в дереве.

Визуализация поведения операций вставки продемонстрирована с помощью графиков, построенных программой, написанной на языке программирования Python (исходный код представлен в приложении Д).

Сравнив графики вставки в дерево в худшем случае и в лучшем, можем убедиться, что операция вставки всегда занимает lb(n) операций, а операция балансировки в худшем – lb(n) операций. Тогда полная операция вставки элемента в Красно-черное дерево требует O(lb(n)) + O(lb(n)) = O(lb(n))

операций, что подтверждает теоретическую оценку сложности операции вставки в Красно-черном дереве.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была реализована структура Красно-черного дерева, а также операция вставки для него. На основании числовых метрик были построены графики. Написан Makefile скрипт, позволяющий нарисовать графики зависимости количества операций для вставки от количества элементов в дереве для множеств данных с разными особенностями распределения при помощи одной команды.

Полученные практические результаты сравнили с теоретическими оценками и на основании сравнения была доказана теоретическая оценка асимптотики работы операции вставки в Красно-черное дерево.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ = Introduction to algorithms. — 2-е изд. — М.: Издательский дом «Вильямс», 2011. — С. 336-364. — ISBN 978-5-8459-0857-5.
2. Красно-черные деревья конспект лекций URL: http://lectures.stargeo.ru/alg/algorithms.htm#_Toc241931998
3. Курс kiev-clrs – Лекция 10. Красно-черные деревья URL: https://nord.org.ua/static/course/algo_2009/lecture10.pdf
4. Красно-черное дерево – Викиконспекты URL: https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД RBNODE

```
template <typename T>
struct RBNode;

template <typename T>
using RBNodeSptr = std::shared_ptr<RBNode<T>>;

template <typename T>
using RBNodeWptr = std::weak_ptr<RBNode<T>>;

template <typename T>
struct RBNode {
	RBNode() = default;
	RBNode(T data, RBColor color, RBNodeSptr<T> left = nullptr,
RBNodeSptr<T> right = nullptr) :
		data(data), color(color), left(left), right(right) {}
	RBNodeWptr<T> parent;
	RBNodeSptr<T> left, right;
	T data = NULL;
	RBColor color = RBColor::BLACK;
};

template <typename T>
RBNodeSptr<T> grandParent(RBNodeSptr<T> node) {
	if (node && !node->parent.expired())
		return ((node->parent).lock()->parent).lock();
	return nullptr;
}

template <typename T>
RBNodeSptr<T> uncle(RBNodeSptr<T> node) {
	RBNodeSptr<T> gp = grandParent(node);
	return !gp ? nullptr : (((node->parent).lock() == gp->left) ?
gp->right : gp->left);
}
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД RBTREE

```
template <typename T>
class RBTree {
#define op(x) x; op_counter++
public:
	RBTree() = default;
	RBTree(const std::initializer_list<T>& list);
	size_t size();
	int insert(T value);
	void clear();
	void outputSorted(std::ostream& out);
	void outputLayers(std::ostream& out);
private:
	void RRight(RBNodeSptr<T> node);
	void RLeft(RBNodeSptr<T> node);
	void insert_1(RBNodeSptr<T> node);
	void insert_2(RBNodeSptr<T> node);
	RBNodeSptr<T> root = nullptr;
	size_t treeSize = 0;
	int op_counter = 0;
};


template <typename T>
void RBTree<T>::RLeft(RBNodeSptr<T> node) {
	op();
	RBNodeSptr<T> pivot = node->right;
	pivot->parent = node->parent;
	if (pivot->parent.expired())
		root = pivot;
	if (!(node->parent.expired())) {
		if ((node->parent.lock())->left == node)
			(node->parent.lock())->left = pivot;
		else
			(node->parent.lock())->right = pivot;
	}
	node->right = pivot->left;
	if (pivot->left)
		pivot->left->parent = node;
	node->parent = pivot;
	pivot->left = node;
}

template <typename T>
void RBTree<T>::RRight(RBNodeSptr<T> node) {
	op();
	RBNodeSptr<T> pivot = node->left;
	pivot->parent = node->parent;
	if (pivot->parent.expired())
		root = pivot;
	if (!(node->parent.expired())) {
		if ((node->parent.lock())->left == node)
			(node->parent.lock())->left = pivot;
		else
			(node->parent.lock())->right = pivot;
```

```
	}
	node->left = pivot->right;
	if (pivot->right)
		pivot->right->parent = node;
	node->parent = pivot;
	pivot->right = node;
}

template <typename T>
RBTree<T>::RBTree(const std::initializer_list<T> & list) {
	for (const auto& i : list)
		insert(i);
}

template <typename T>
size_t RBTree<T>::size()
{
	return treeSize;
}

template <typename T>
void RBTree<T>::insert_1(RBNodeSptr<T> node) {
	op();
	if (node->parent.expired())
		node->color = RBColor::BLACK;
	else {
		if (node->parent.lock()->color == RBColor::BLACK)
			return;
		insert_2(node);
	}
}

template <typename T>
void RBTree<T>::insert_2(RBNodeSptr<T> node) {
	op();
	RBNodeSptr<T> U = uncle(node);
	if (U && U->color == RBColor::RED) {
		node->parent.lock()->color = RBColor::BLACK;
		U->color = RBColor::BLACK;
		RBNodeSptr<T> G = grandParent(node);
		G->color = RBColor::RED;
		insert_1(G);
	}
	else {
		RBNodeSptr<T> G = grandParent(node);
		if ((node == node->parent.lock()->right) && (node-
>parent.lock() == G->left)) {
			RLeft(node->parent.lock());
			node = node->left;
		}
		else if ((node == node->parent.lock()->left) && (node-
>parent.lock() == G->right)) {
			RRight(node->parent.lock());
			node = node->right;
		}
		G = grandParent(node);
		node->parent.lock()->color = RBColor::BLACK;
```

```
            G->color = RBColor::RED;
            if ((node == node->parent.lock()->left) && (node-
>parent.lock() == G->left))
                  RRight(G);
            else
                  RLeft(G);
      }
}

template <typename T>
int RBTree<T>::insert(T value) {
      RBNodeSptr<T> node = std::make_shared<RBNode<T>>(value,
RBColor::RED);
      treeSize++;
      if (treeSize == 1)
            root = node;
      else {
            RBNodeSptr<T> it = root, father = nullptr;
            while (it) {
                  op(father = it);
                  it = (it->data < node->data) ? it->right : it->left;
            }
            node->parent = father;
            if (father->data < node->data)
                  father->right = node;
            else
                  father->left = node;
      }
      insert_1(node);
      auto ops = op_counter;
      op_counter = 0;
      return ops;
}

template<typename T>
inline void RBTree<T>::clear()
{
      root.reset();
      treeSize = 0;
}

template <typename T>
void RBTree<T>::outputSorted(std::ostream & out) {
      auto pushToOut = [&out](const RBNodeSptr<T> & node, auto &&
pushToOut) {
            if (!node)
                  return;
            pushToOut(node->left, pushToOut);
            out << node->data << ' ';
            pushToOut(node->right, pushToOut);
      };
      pushToOut(root, pushToOut);
      out << '\n';
}

template <typename T>
void RBTree<T>::outputLayers(std::ostream & out) {
```

```
     auto showTree = [&out](RBNodeSptr<int> root, auto && showTree, int
p = 0, int s = 0) {
          auto showLine = [&out](const char* c, int p, int s) {
               for (int i = 0; i < p; i++) {
                    out << (s & 1 ? "|   " : "    ");
                    s /= 2;
               }
               out << c;
          };
          if (!root) return;
          out << (root->color == RBColor::RED ? "\033[31m" : "") <<
root->data << "\033[30m\n";
          if (root->left) {
               showLine("|\n", p, s);
               showLine("L: ", p, s);
               showTree(root->left, showTree, p + 1, s + ((root->right
== NULL ? 0 : 1) << p));
          }
          if (root->right) {
               showLine("|\n", p, s);
               showLine("R: ", p, s);
               showTree(root->right, showTree, p + 1, s);
          }
     };
     showTree(root, showTree);
     out << '\n'<<'\n';
}
```

ПРИЛОЖЕНИЕ В
ИСХОДНЫЙ КОД RBTESTER

```
#pragma once
#include <queue>
#include <algorithm>
#include <random>

#include "RBTree.h"

class RBTester {
public:
	RBTester(RBTree<int>& tree) :tree(tree) {}
	void insertBest();
	void insertAverage();
	void insertWorst();
private:
	RBTree<int>& tree;
};

void RBTester::insertBest() {
	tree.clear();
	std::queue<int>que;
	que.push(1<<19);
	que.push(0);
	int jmpLength = 1 << 18;
	std::ofstream f("insertBest.txt", std::fstream::out);
	while (jmpLength>0) {
		int cur = que.front();
		que.pop();
		if (cur == 0) {
			jmpLength >>= 1;
			que.push(0);
			continue;
		}
		f << tree.size() << ' ' << tree.insert(cur) << '\n';
		que.push(cur - jmpLength);
		que.push(cur + jmpLength);
	}
	while (!que.empty()) {
		int cur = que.front();
		que.pop();
		if (cur == 0)
			continue;
		else
			f << tree.size() << ' ' << tree.insert(cur) << '\n';
	}
	f << tree.size() << ' ' << tree.insert(1 << 20) << '\n';
}

void RBTester::insertAverage()
{
	tree.clear();
	std::vector<int>values;
	values.resize(1 << 20);
	for (int i = 0; i < 1 << 20; i++)
		values[i] = i+1;
```

```
	std::shuffle(values.begin(), values.end(),
std::mt19937(std::random_device()()));
	std::ofstream f("insertAverage.txt", std::fstream::out);
	for (int i = 0; i < 1 << 20; i++)
		f << tree.size() << ' ' << tree.insert(values[i]) << '\n';
}

void RBTester::insertWorst() {
	tree.clear();
	std::ofstream f("insertWorst.txt", std::fstream::out);
	for (int i = 1; i <= 1<<20; i++) {
		f << tree.size() << ' ' << tree.insert(i) << '\n';
	}
}
```

ПРИЛОЖЕНИЕ Г
ИСХОДНЫЙ КОД MAIN

```
#include <iostream>
#include <string>
#include <cstring>
#include <fstream>
#include <sstream>

#include "RBTree.h"
#include "RBTester.h"

int main(int argc, char** argv) {
#ifdef _WIN32
	std::system("color 80");
#endif
	RBTree<int> tree;
#ifdef GRAPH
	RBTester tester(tree);
	if(argc==2 && !std::strcmp(argv[1], "worst"))
		tester.insertWorst();
	else if (argc==2 && !std::strcmp(argv[1], "average"))
		tester.insertAverage();
	else if(argc==2 && !std::strcmp(argv[1], "best"))
		tester.insertBest();
	else {
		tester.insertWorst();
		tester.insertAverage();
		tester.insertBest();
	}
	return 0;
#else
	while (true) {
		std::string userInput;
		std::getline(std::cin, userInput);
		std::stringstream sstr(userInput);
		int value;
		while (sstr >> value)
			tree.insert(value);
#ifdef _DEBUG
		tree.outputLayers(std::cout);
#endif
#ifdef TEST
		std::ofstream file("res.txt", std::ofstream::out);
		tree.outputSorted(file);
		break;
#endif
	}
	return 0;
#endif
}
```

ПРИЛОЖЕНИЕ Д
ИСХОДНЫЙ КОД MAIN.PY

```
import sys
import matplotlib.pyplot as plt
import numpy as np
import random
import math as mt

fname = sys.argv[1]
f = open(fname, 'r')
case = "WORST" if fname=="insertWorst.txt" else \
          "BEST" if fname=="insertBest.txt" else \
          "AVERAGE"
nc = []
ops = []

for line in f:
    s = line.split()
    nc.append(int(s[0]))
    ops.append(int(s[1]))



graphLength = 3/4*2**20
totalcount = 2**20;
jumpLength = 1;
opsAvg = [ops[0]]
ncAvg = [nc[0]]
while jumpLength < totalcount:
     ncc = 0
     opc = 0
     for i in range(jumpLength, 2*jumpLength):
          ncc+=nc[i]
          opc+=ops[i]
     ncAvg.append(ncc/jumpLength)
     opsAvg.append(opc/jumpLength)
     jumpLength *=2

x = np.arange(1, graphLength, 0.1)
fstr = "2log(x)" if case=="WORST" else "log(x)"
lb = 2*np.log2(x) if case=="WORST" else np.log2(x)
plt.plot(ncAvg, opsAvg, "b", label="Aproximated complexity")
plt.plot(x, lb, "red", label=fstr)

plt.xlabel("Nodes count")
plt.ylabel("Operations count")
plt.title(f"Insert, case:{case}")
plt.grid()
plt.legend();
plt.show()
```

ПРИЛОЖЕНИЕ Е
ИСХОДНЫЙ КОД TEST.PY

```
myRes = list(map(int, input().strip().split(' ')))
inputSeq = list(map(int, input().strip().split(' ')))
print(f'Testing sequence {inputSeq}', f'My program res: {myRes}',
sep='\n')
trueRes = sorted(inputSeq)
if myRes == trueRes:
    print('\033[32mCompleted successfully\033[m\n')
else:
    print('\033[31mA failure occured\033[m\n', f'True program res:
{trueRes}', sep='\n')
```

ПРИЛОЖЕНИЕ Ж
ИСХОДНЫЙ КОД MAKEFILE

```
.PHONY: test clean worst average best

compile:
	@echo "Compiling program..."
	@g++ Source/main.cpp -D _DEBUG -o main
	@echo "Compiled successfully"

test:
	@echo "Compiling test version of the program..."
	@g++ Source/main.cpp -D TEST -o test
	@echo "Compiled successfully"; echo ''
	@for file in ./Tests/*; do \
	echo Test $${file}; \
	cp $${file} curTest.txt; \
	./test < curTest.txt && \
	cat res.txt curTest.txt > forCheck.txt; \
	python3.8 Source/test.py < forCheck.txt; \
	done
	@rm forCheck.txt res.txt curTest.txt test

graph:
	@echo "Compiling insert testing version of the program..."
	@g++ Source/main.cpp -D GRAPH -o graph

worst: graph
	@echo "Counting operations"
	@./graph worst
	@echo "Drawing worst case graph"
	@python Source/main.py insertWorst.txt

average: graph
	@echo "Counting operations"
	@./graph average
	@echo "Drawing average case graph"
	@python Source/main.py insertAverage.txt

best: graph
	@echo "Counting operations"
	@./graph best
	@echo "Drawing best case graph"
	@python Source/main.py insertBest.txt

clean:
	@rm main test graph res insertAverage.txt insertBest.txt
insertWorst.txt 2> /dev/null
```