


МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

Работа допущена к защите

 Руководитель  
«26» 05 2021 г.

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Качество и тестирование программного обеспечения»

на тему: «Организация и проведение комплексного тестирования  
программного обеспечения для пополнения и поиска в структуре данных  
типа красно-чёрное дерево»

Студент  Беликов П.Г.

Шифр 180818

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Направленность (профиль) Промышленная разработка программного  
обеспечения

Группа 81ПГ

Руководитель  Ужаринский А.Ю.

Оценка: «отлично»

Дата 04.06.2021

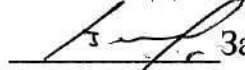
Орел 2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

УТВЕРЖДАЮ:

 Зав. кафедрой

«\_\_» \_\_\_\_\_ 20\_\_ г.

**ЗАДАНИЕ**  
на курсовой проект

по дисциплине «Качество и тестирование программного обеспечения»

Студент Беликов П.Г.

Шифр 180818

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Направленность (профиль) Промышленная разработка программного  
обеспечения

Группа 81ПГ

1 Тема курсового проекта

«Организация и проведение комплексного тестирования программного  
обеспечения для пополнения и поиска в структуре данных типа красно-  
чёрное дерево»

2 Срок сдачи студентом законченной работы «31» мая 2021

### 3 Исходные данные


Описание задачи, требования к разрабатываемому программному обеспечению.

### 4 Содержание курсового проекта

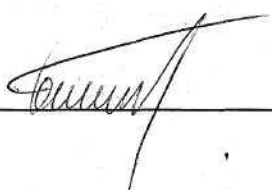
Проектирование и реализация программного обеспечения, тестирование программного обеспечения, оценка качества разработанного программного обеспечения.

### 5 Отчетный материал курсового проекта

Пояснительная записка курсового проекта.

Руководитель  Ужаринский А.Ю.

Задание принял к исполнению: «15» февраля 2021

Подпись студента 

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	5
1 ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ .....	6
1.1 Анализ и выбор методов представления красно-чёрного дерева.....	6
1.2 Проектирования алгоритмов .....	8
1.2.1 Алгоритмы левого и правого поворотов .....	8
1.2.2 Алгоритм балансировки .....	10
1.2.3 Алгоритм вставки.....	14
1.2.4 Алгоритм определения <i>i</i> -ой порядковой статистики .....	15
1.2.5 Алгоритм определения порядкового номера заданного элемента.....	16
1.3 Реализация программы .....	17
2 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.....	21
2.1 Организация процесса тестирования программного обеспечения.....	21
2.2 Тестирование элементов разработанного программного обеспечения .....	22
2.3 Тестирование интеграций модулей разработанного программного обеспечения .....	26
2.4 Тестирование правильности разработанного программного обеспечения .....	26
2.5 Системное тестирование разработанного программного обеспечения .....	29
3 ОЦЕНКА КАЧЕСТВА РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.....	31
3.1 Статическая оценка качества разработанного программного обеспечения .....	31

3.2	Динамическая оценка качества разработанного программного обеспечения .....	33
ЗАКЛЮЧЕНИЕ .....		35
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....		36
Приложение А .....		37
Приложение В.....		51

## **ВВЕДЕНИЕ**

В век технологий, ежедневно рождается множество новых программ или сервисов, которые призваны выполнить те или иные функции. На этом фоне становится очевидным, что пользователь может выбрать самое лучшее приложение или программное обеспечение, ведь выбор очень велик. Это делается обычно опытным путем, либо по рекомендации других пользователей. В связи с этим разработчики решили установить определенных стандарт качества ПО, который проверяется специальной процедурой, именуемой тестирование программного обеспечения. В процессе тестирования выявляются дефекты, которые впоследствии были бы неудобны пользователям и в результате ПО усовершенствуется, принося пользу всем.

Целью курсовой работы является организация и проведение комплексного тестирования программного обеспечения для пополнения и поиска в структуре данных типа красно-чёрное дерево.

Задачами курсовой работы, которые необходимо выполнить для достижения поставленной цели, являются:

1. Изучение предметной области.
2. Проектирование программного обеспечения.
3. Реализация программного обеспечения.
4. Тестирование программного обеспечения.
5. Оценка качества разработанного программного обеспечения.

# 1 ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

## 1.1 Анализ и выбор методов представления красно-чёрного дерева

Красно-чёрное дерево представляет собой бинарное дерево поиска с одним дополнительным битом цвета в каждом узле. Цвет узла может быть либо чёрным, либо красным. В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-чёрном дереве не отличается от другого по длине более чем в два раза, так что красно-чёрные деревья являются приближенно сбалансированными. [1]

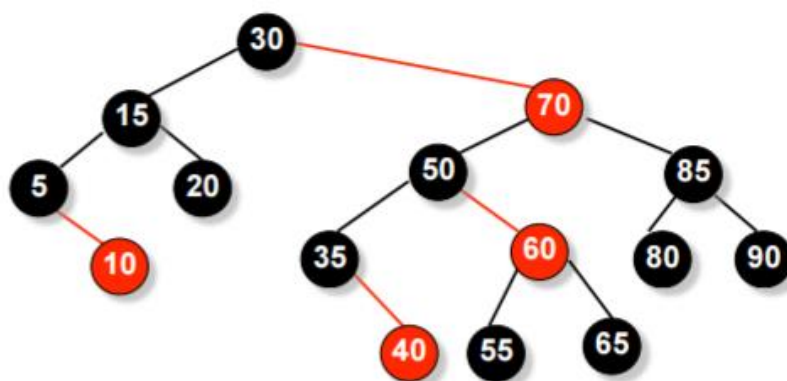


Рисунок 1 – Красно-чёрное дерево. [3]

Каждый узел дерева содержит атрибуты `key`, `color`, `left`, `right` и `parent`. Параметр `key` содержит значение узла, `color` – его цвет. Параметры `parent`, `left` и `right` содержат указатели на родительский узел, левый дочерний и правый дочерний соответственно. При отсутствии дочернего или родительского узла соответствующее значение принимает `None`.

```
class Node:
    def __init__(self, key):
        self.key = key
        self.red = True
        self.left = None
        self.right = None
        self.parent = None
```

Рисунок 2 – Узел красно-чёрного дерева (класс `Node`).

Бинарное дерево является красно-чёрным деревом, если оно удовлетворяет следующим свойствам:

1. Каждый узел является либо черным, либо красным.
2. Корень дерева является чёрным узлом.
3. У красного узла родительский узел всегда чёрный.
4. Все простые пути из любого узла до листьев содержат одинаковое количество чёрных узлов. [3]

Для реализации структуры данных был выбран язык Python. Сама структура данных будет представлена в виде двух классов: Node, RBTree.

Класс Node представляет собой описание узла дерева (Рисунок 2).

Класс RBTree является непосредственно деревом, и содержит одну переменную – указатель на экземпляр класса Node – корень дерева.

```
class RBTree:
    def __init__(self):
        self.root = None
```

Рисунок 3 – Класс RBTree.

Взаимодействие с деревом будет производиться при помощи методов класса RBTree.

Для решения задач определения *i*-й порядковой статистики и определения порядкового номера заданного элемента необходимо расширить имеющуюся структуру, добавив каждому узлу *x* атрибут *size*, который будет содержать количество узлов в поддереве с корнем *x* (включая сам *x*).

```
class Node:
    def __init__(self, key):
        self.key = key
        self.red = True
        self.left = None
        self.right = None
        self.parent = None
        self.size = 1
```

Рисунок 4 – Расширенный класс Node.



## 1.2 Проектирования алгоритмов

### 1.2.1 Алгоритмы левого и правого поворотов

Алгоритмы поворота представляют собой локальные операции в дереве поиска, сохраняющие свойства бинарного дерева поиска. [1]

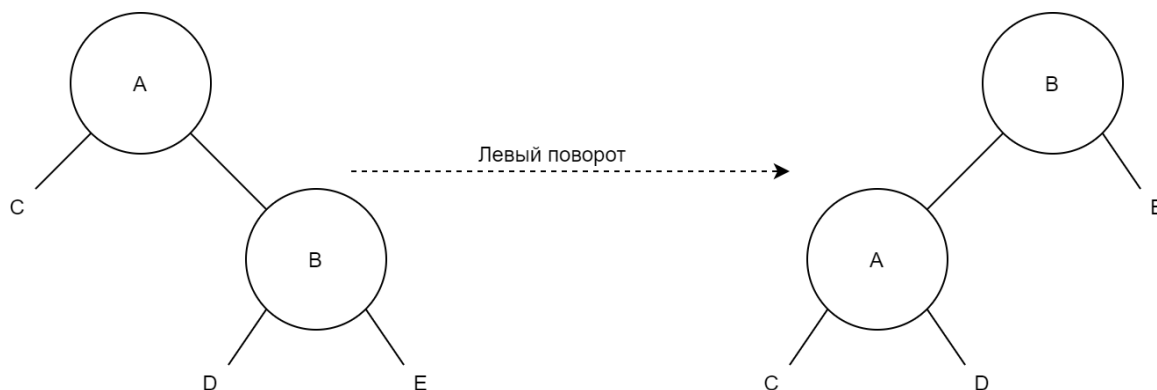


Рисунок 5 – Левый поворот в бинарном дереве поиска.

При выполнении левого поворота предполагается, что правый дочерний узел поворачиваемого узла не равен None (он существует).

При повороте изменяются только указатели и значение атрибута `size`, все остальные параметры остаются без изменений.

На Рисунке 5 представлен левый поворот в бинарном дереве. Он выполняется вокруг связи между A и B, делая B новым корнем поддерева, левым дочерним узлом которого становится A, а бывший левый потомок узла B – правым потомком A. После происходит пересчет размера поддеревьев узлов A и B. Размер поддерева B приравнивается к старому значению `size` узла A, а размер поддерева A рассчитывается как сумма размеров поддеревьев его новых правого и левого потомков, увеличенная на единицу. Реализация левого поворота представлена в Приложении А.

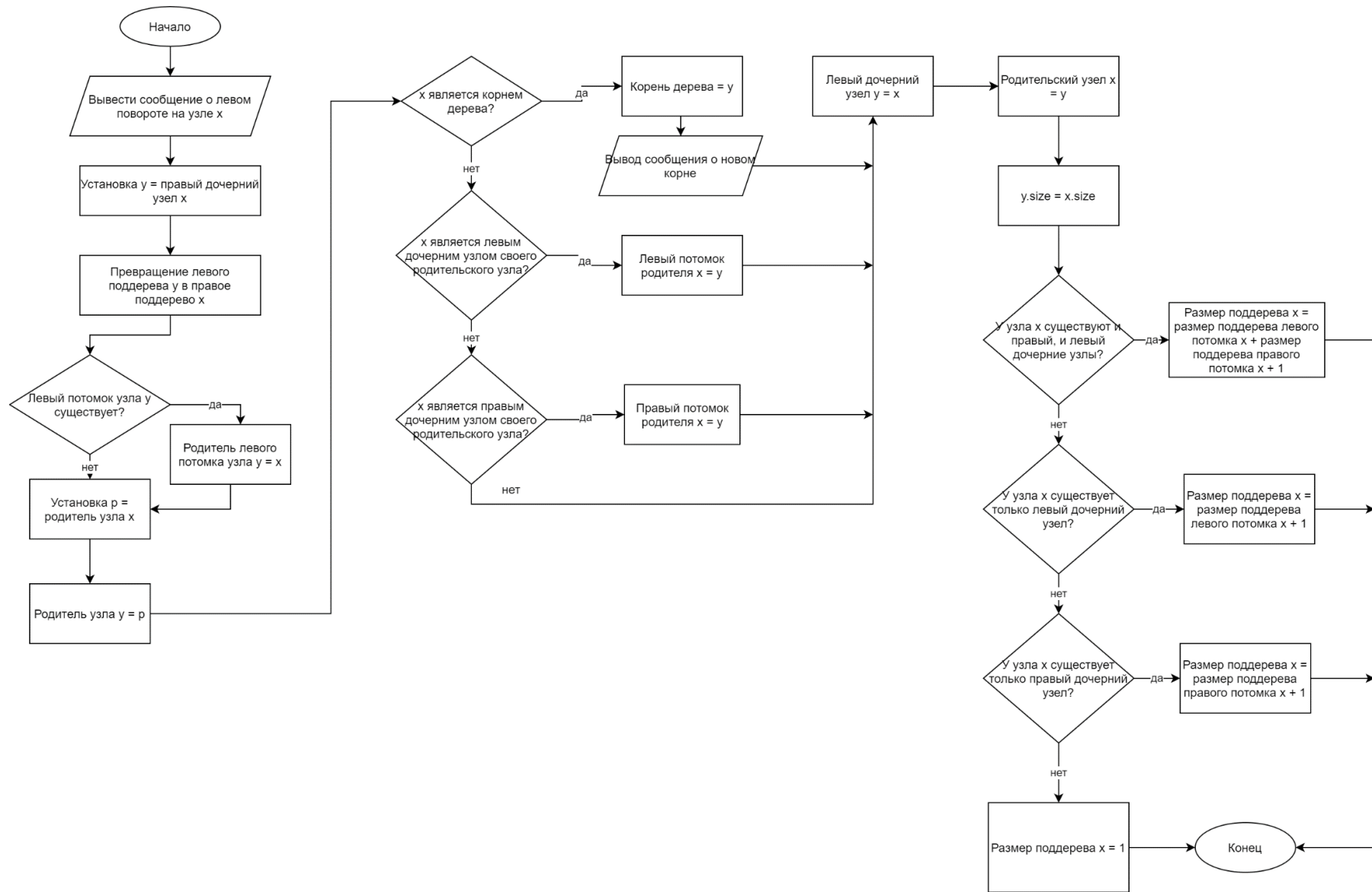


Рисунок 6 – Схема алгоритма левого поворота.

Алгоритм правого поворота симметричен алгоритму левого поворота.



Рисунок 7 – Правый поворот в бинарном дереве поиска.

При выполнении правого поворота предполагается, что левый дочерний узел поворачиваемого узла не равен None (он существует).

На Рисунке 7 представлен правый поворот в бинарном дереве. Он выполняется вокруг связи между В и А, делая А новым корнем поддерева, правым дочерним узлом которого становится В, а бывший правый потомок узла А – левым потомком В. После происходит пересчет размера поддеревьев узлов А и В. Размер поддерева А приравнивается к старому значению size узла В, а размер поддерева В рассчитывается как сумма размеров поддеревьев его новых правого и левого потомков, увеличенная на единицу. Реализация левого поворота представлена в Приложении А.

### 1.2.2 Алгоритм балансировки

Алгоритм балансировки приводит красно-чёрное дерево к виду, в котором все его свойства соблюдаются. Алгоритм рассматривает конкретный проблемный узел (для которого нарушено какое-либо свойство), и после восстановления его свойств двигается вверх по дереву, изменяя его.

Необходимость в балансировке возникает, когда у красного узла появляется красный дочерний угол. При балансировке красно-чёрных

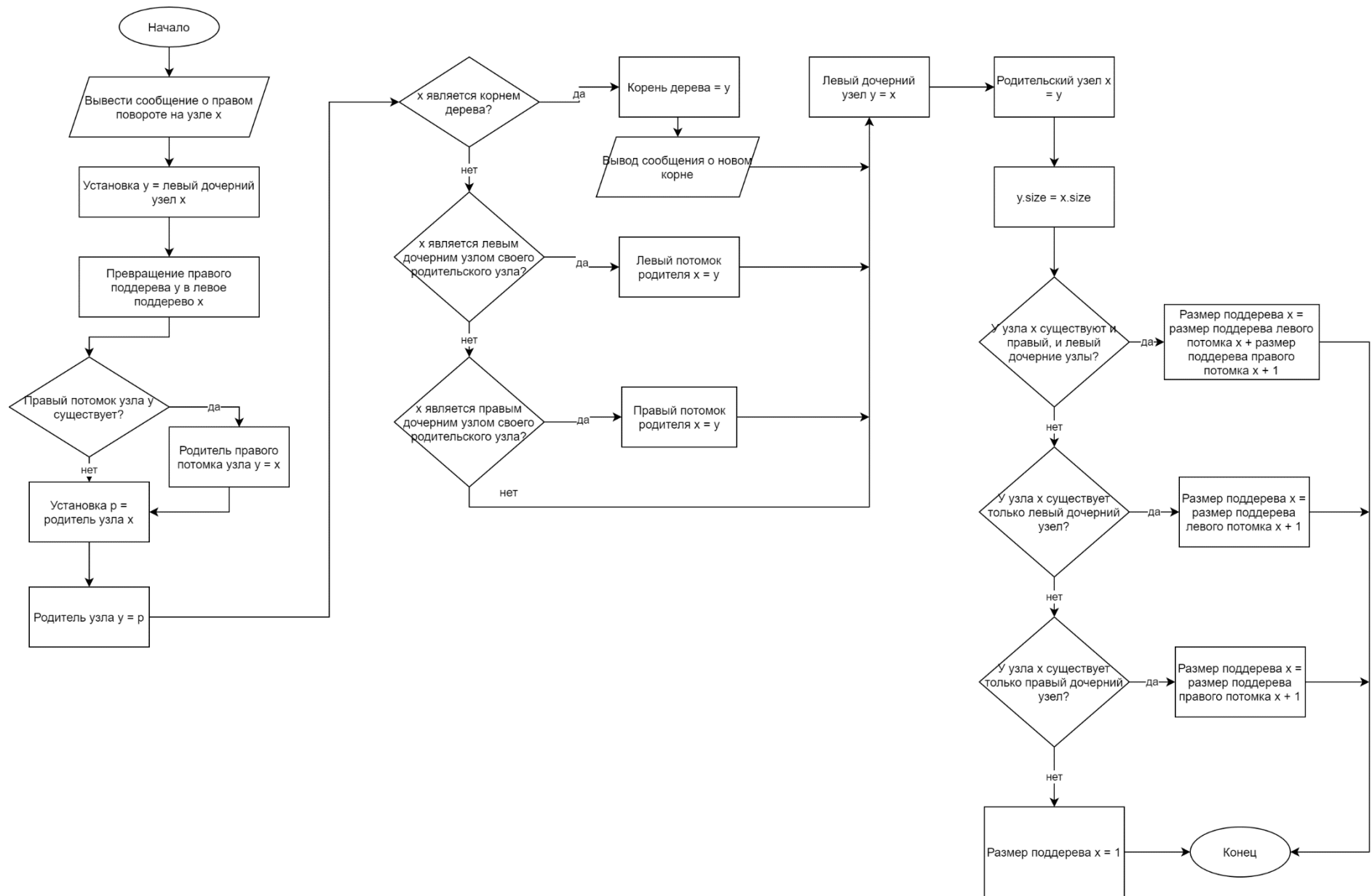


Рисунок 8 – Схема алгоритма правого поворота.

деревьев можно выделить три основных случая, при которых необходима балансировка:

1. “Дядя” z узла x красный.
2. “Дядя” z узла x чёрный (или отсутствует вовсе), и родительский узел x с родительским углом z находятся в разных сторонах.
3. “Дядя” z узла x чёрный (или отсутствует вовсе), и родительский узел x с родительским углом z находятся в одной стороне. [4]

В первом случае необходимо перекрасить родительский узел и “Дядю” в чёрный цвет, а цвет родительского узла родительского узла изменить на противоположный. Затем, если в алгоритме был изменен цвет корня, необходимо поменять его цвет на чёрный.

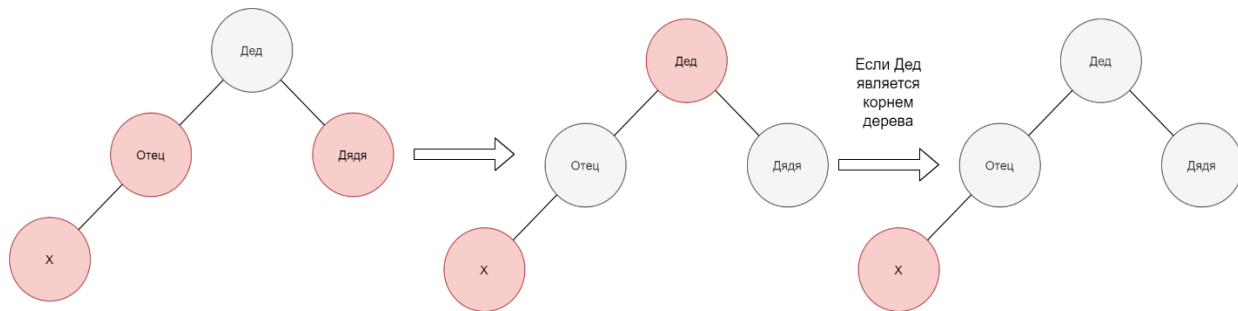


Рисунок 9 – Красный Дядя.

Во втором случае необходимо привести структуру к третьему случаю, когда Папа и Дед идут в одну сторону. Для этого нужно выполнить малый поворот по родительскому узлу x (отец).

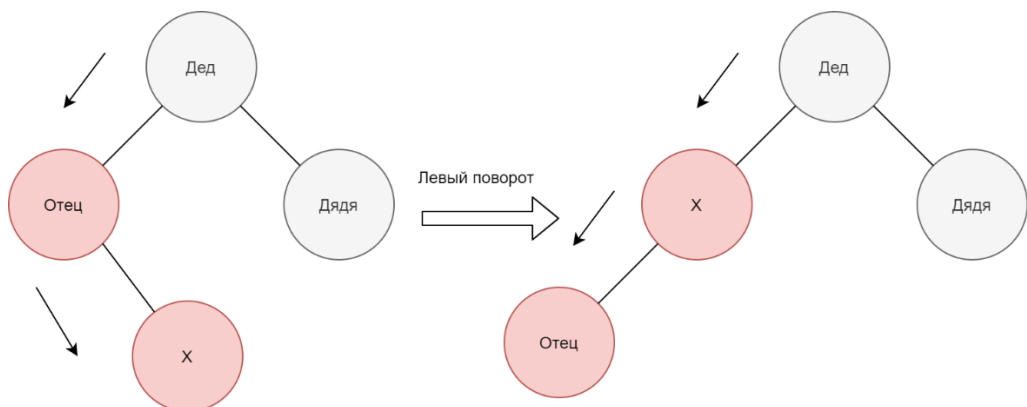


Рисунок 10 – Чёрный Дядя (лево).

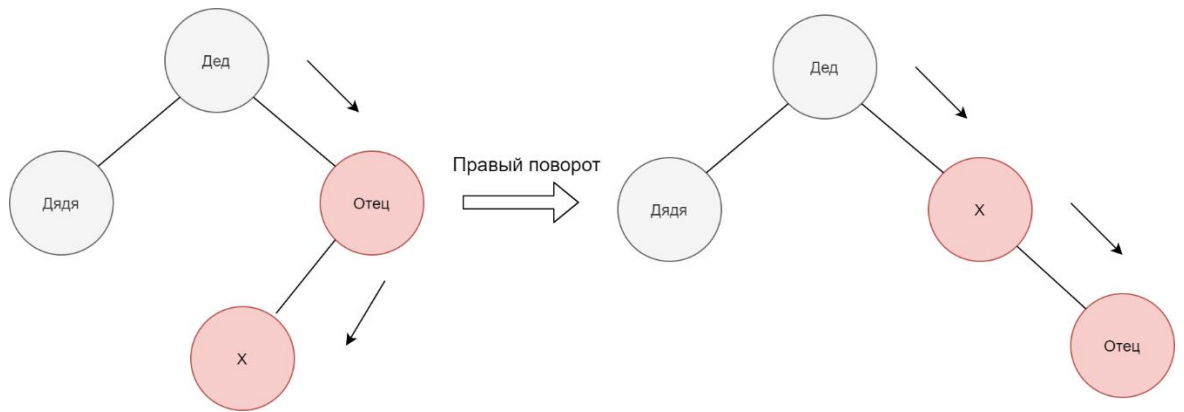


Рисунок 11 – Чёрный Дядя (право).

В третьем случае необходимо выполнить поворот деда в соответствующую сторону. После поворота поворачиваемый угол (дед) перекрашивается в красный, а родительский узел (отец) – в чёрный.

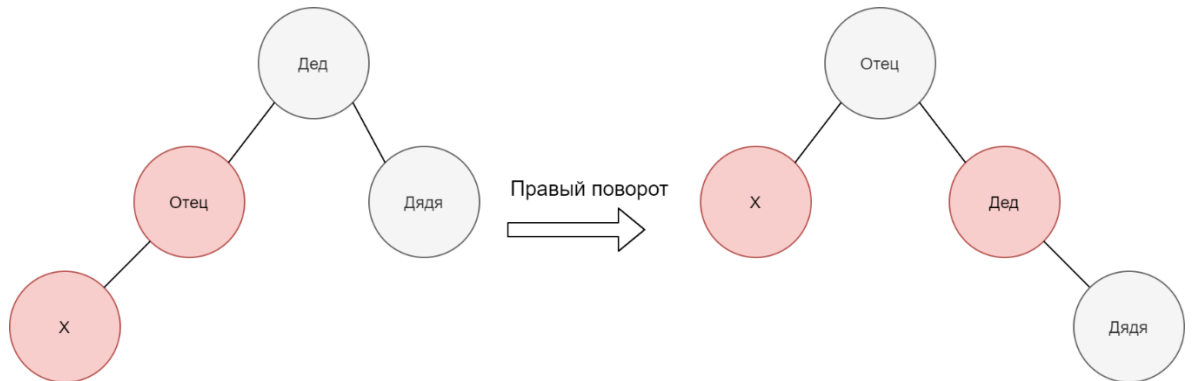


Рисунок 12 – Чёрный Дядя (лево).

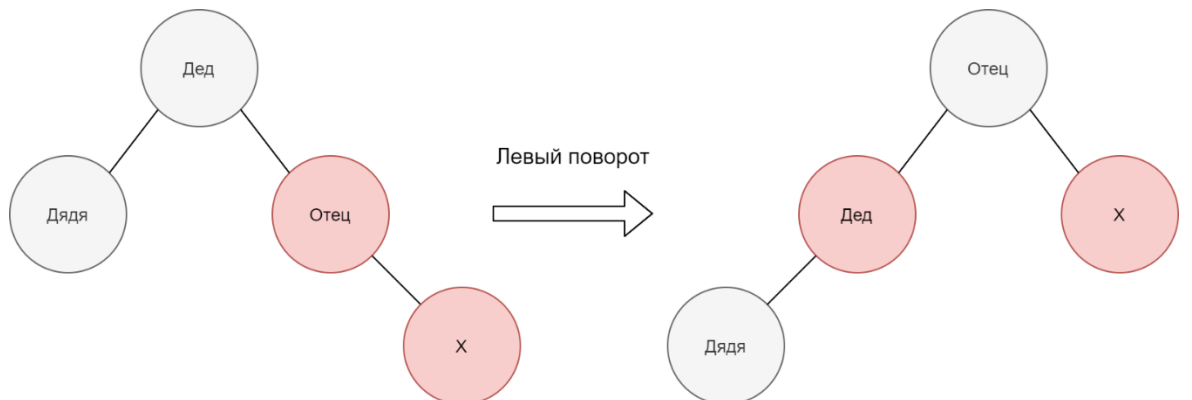


Рисунок 13 – Чёрный Дядя (право).

Балансировка позволяет красно-чёрному дереву сохранить свои свойства при каком-либо изменении структуры данных. Реализация алгоритма балансировки представлена в Приложении А.

### 1.2.3 Алгоритм вставки

Вставка в красно-черное дерево начинается со вставки элемента, как в обычном бинарном дереве поиска. Только здесь элементы вставляются в позиции NULL-листьев. Вставленный узел всегда окрашивается в красный цвет. [1]

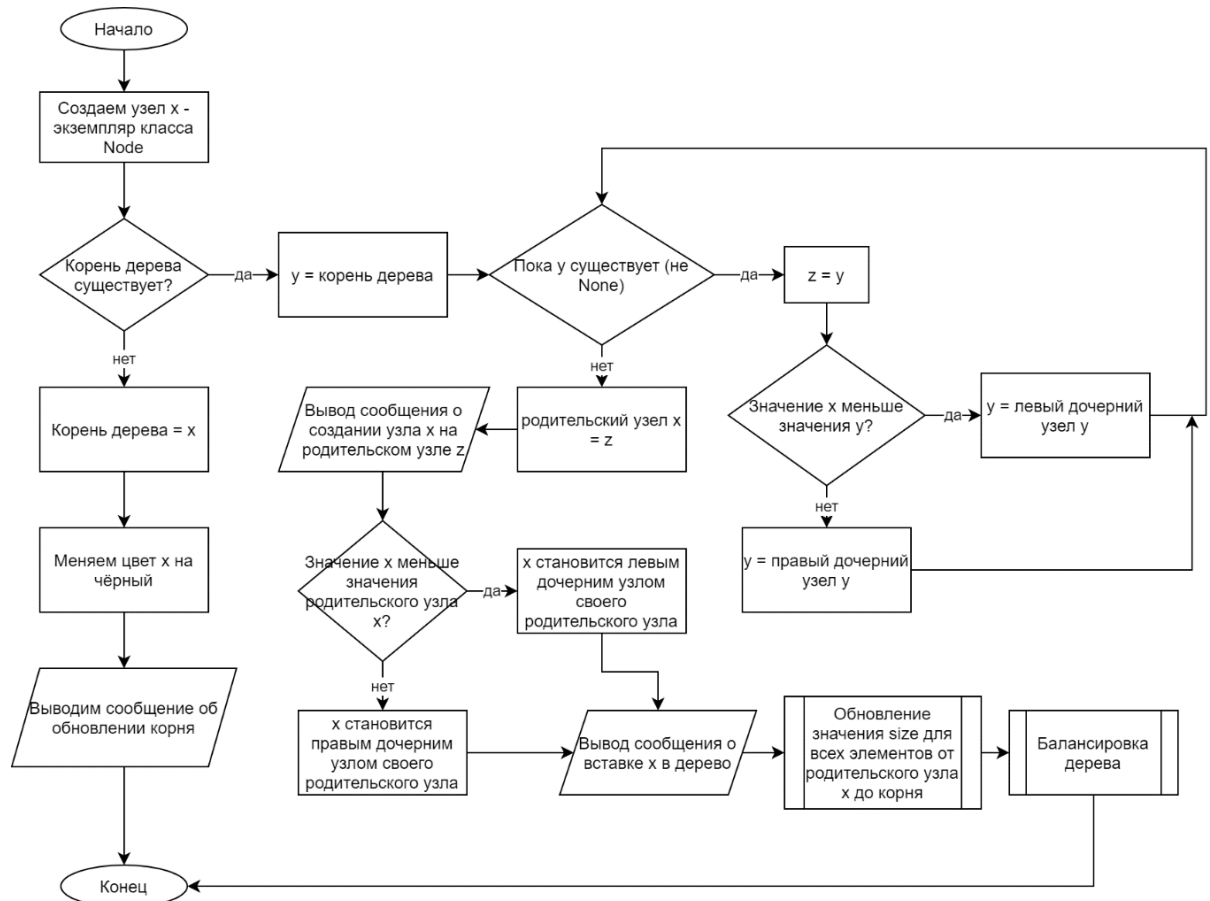


Рисунок 14 – Схема алгоритма вставки в красно-чёрное дерево.

При выполнении вставки нового узла в красно-чёрное дерево выполняется следующий алгоритм:

1. Создаём новый узел  $x$  – экземпляр класса `Node`. В качестве значения узла используется переданный аргумент, остальные поля устанавливаются по умолчанию.
2. Проверяем, существует ли корень дерева. Если существует, то переходим на шаг 6, иначе на шаг 3.
3. Узел  $x$  становится корнем дерева.

4. Меняем цвет  $x$  на чёрный.
  5. Выводим сообщение об обновлении корня. Переходим на шаг 20.
  6. Записываем корень дерева в переменную  $y$ .
  7. Если переменная  $y$  существует (не равна None), переходим на шаг 8, иначе на шаг 12.
  8. Записываем узел  $y$  в переменную  $z$ .
  9. Если значение узла  $x$  меньше значения  $y$ , то переходим на шаг 10, иначе на шаг 11.
  10. Помещаем в  $y$  левый дочерний узел  $y$ . Переходим на шаг 7.
  11. Помещаем в  $y$  правый дочерний узел  $y$ . Переходим на шаг 7.
  12.  $z$  становится родительским узлом узла  $x$ .
  13. Выводим сообщение о создании узла  $x$  с родительским узлом  $z$ .
  14. Если значение узла  $x$  меньше значения родительского узла  $x$ , то идем на шаг 15, иначе на шаг 16.
  15.  $x$  становится левым дочерним узлом своего родительского узла.
  16.  $x$  становится правым дочерним узлом своего родительского узла.
  17. Вывод сообщения о вставке  $x$  в дерево.
  18. Обновить значение `size` для узлов от родительского узла  $x$  до корня дерева.
  19. Выполнить балансировку дерева для сохранения красно-чёрных свойств.
- Завершить работу.

#### **1.2.4 Алгоритм определения $i$ -ой порядковой статистики**

Алгоритм определения  $i$ -ой порядковой статистики подразумевает поиск узла по заданному порядковому номеру в упорядоченном по возрастанию дереве.



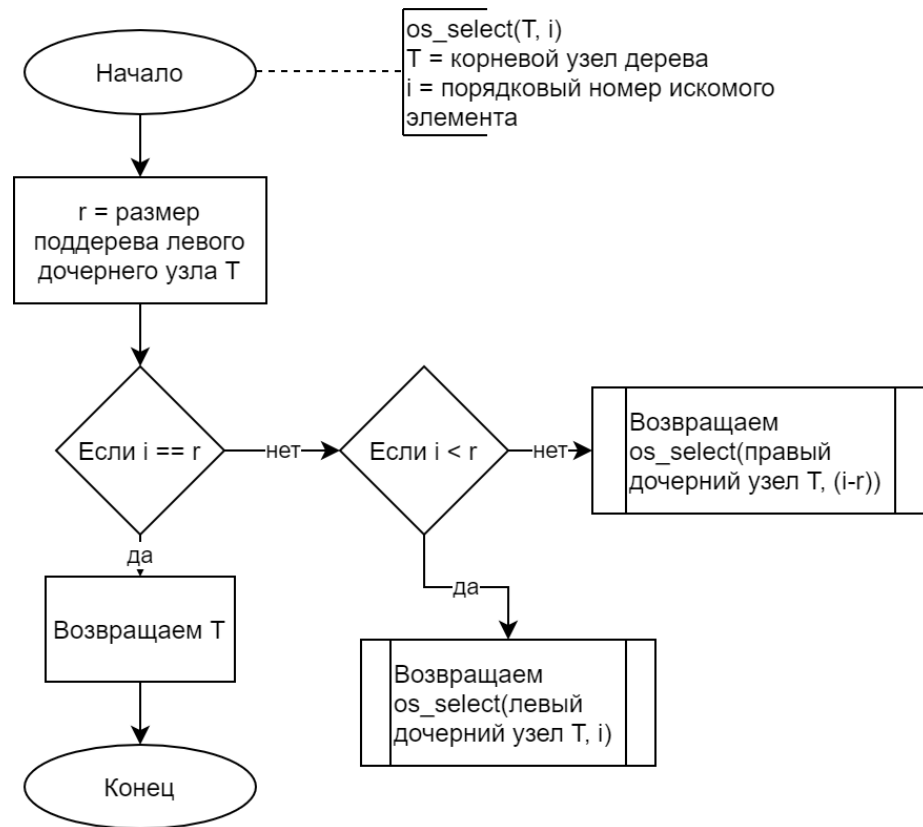


Рисунок 15 – Схема алгоритма поиска  $i$ -й порядковой статистики.

Функция данного алгоритма возвращает указатель на узел, содержащий  $i$ -ое в порядке возрастания значение в дереве. Реализация алгоритма поиска  $i$ -й порядковой статистики представлена в приложении А.

### 1.2.5 Алгоритм определения порядкового номера заданного элемента

Алгоритм определения порядкового номера заданного элемента должен по заданному указателю на узел дерева найти позицию данного узла при центрированном обходе дерева.

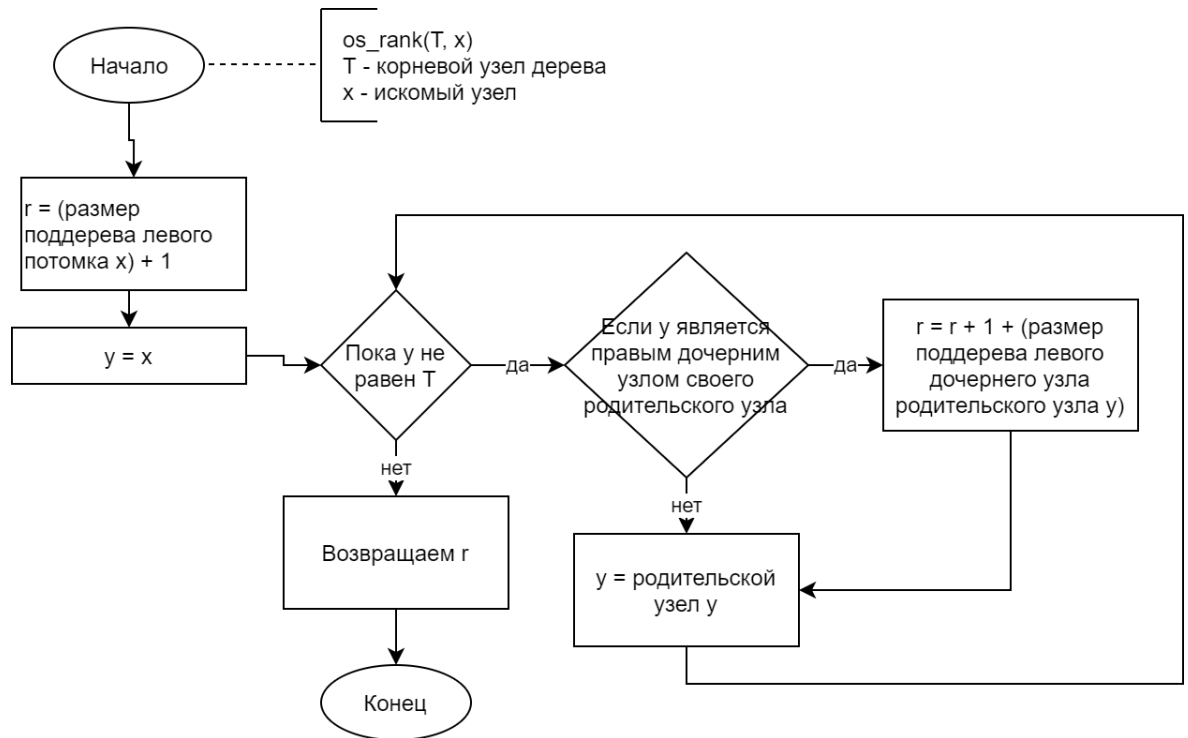


Рисунок 16 – Схема алгоритма определения порядкового номера заданного элемента.

### 1.3 Реализация программы

В программе представлены 2 класса, описывающие заданную структуру данных: `RBTree` и `Node`.

Класс `Node` представляет собой реализацию узла дерева со следующими полями:

1. `key` – содержит значение узла (`int`).
2. `red` – содержит информацию о цвете узла, `True` – красный узел, `False` – чёрный. По умолчанию `True`.
3. `left` – содержит указатель на левый дочерний узел, `None` если таковой не существует. По умолчанию `None`.
4. `right` – содержит указатель на правый дочерний узел, `None` если таковой не существует. По умолчанию `None`.

5. `parent` – содержит указатель на родительский узел, `None` если таковой не существует (только для корневого узла). По умолчанию `None`.
6. `size` – содержит размер поддерева узла. По умолчанию равен 1.

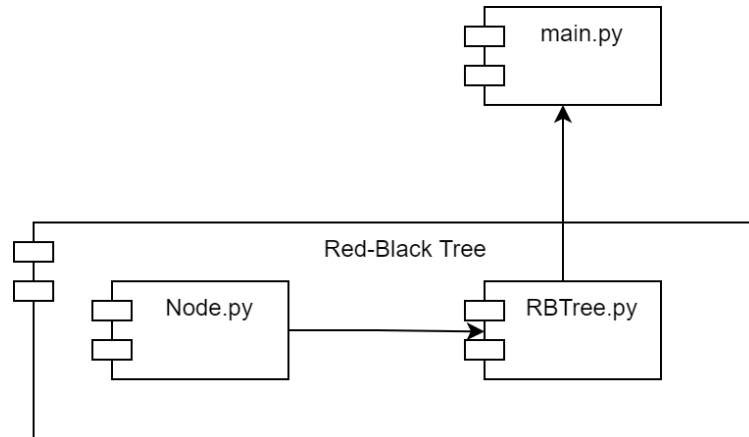


Рисунок 17 – Диаграмма компонентов.

Класс `RBTTree` представляет собой само дерево и имеет всего одно поле – указатель на корневой узел (`Node`). Именно класс `RBTTree` содержит все методы взаимодействия с деревом, такие как:

1. `left_rotate(x)` – выполняет левый поворот на узле `x`. Обновляет размер поддеревьев у задействованных в повороте узлов.
2. `right_rotate(x)` – выполняет правый поворот на узле `x`. Обновляет размер поддеревьев у задействованных в повороте узлов.
3. `print_root()` – выводит значение корня, его цвет и размер дерева.
4. `insert(x)` – создает экземпляр класса `Node` со значением `x` и выполняет его вставку в дерево.
5. `fix_tree(x)` – выполняет балансировку дерева, проверяя сохранение красно-чёрных свойств начиная с узла `x` и заканчивая корневым узлом дерева.
6. `fix_size(x)` – обновляет размер поддеревьев узлов начиная с `x` и заканчивая корневым узлом дерева.
7. `search(x)` – выполняет бинарный поиск узла `x` по дереву, возвращая его.
8. `os_select(root, i)` – выполняет поиск `i`-ого узла по возрастанию в дереве и возвращает его.

9. `os_rank(tree, x)` – выполняет поиск узла `x` в дереве `tree` возвращает его порядковую статистику.

При запуске, программа принимает команды пользователя до тех пор, пока тот не решит выйти. Всего пользователю доступны 4 команды:

1. Вставка в дерево нового узла.
2. Поиск узла по индексу.
3. Поиск индекса по узлу.
4. Выход.

Алгоритм работы программы представлен на Рисунке 18.

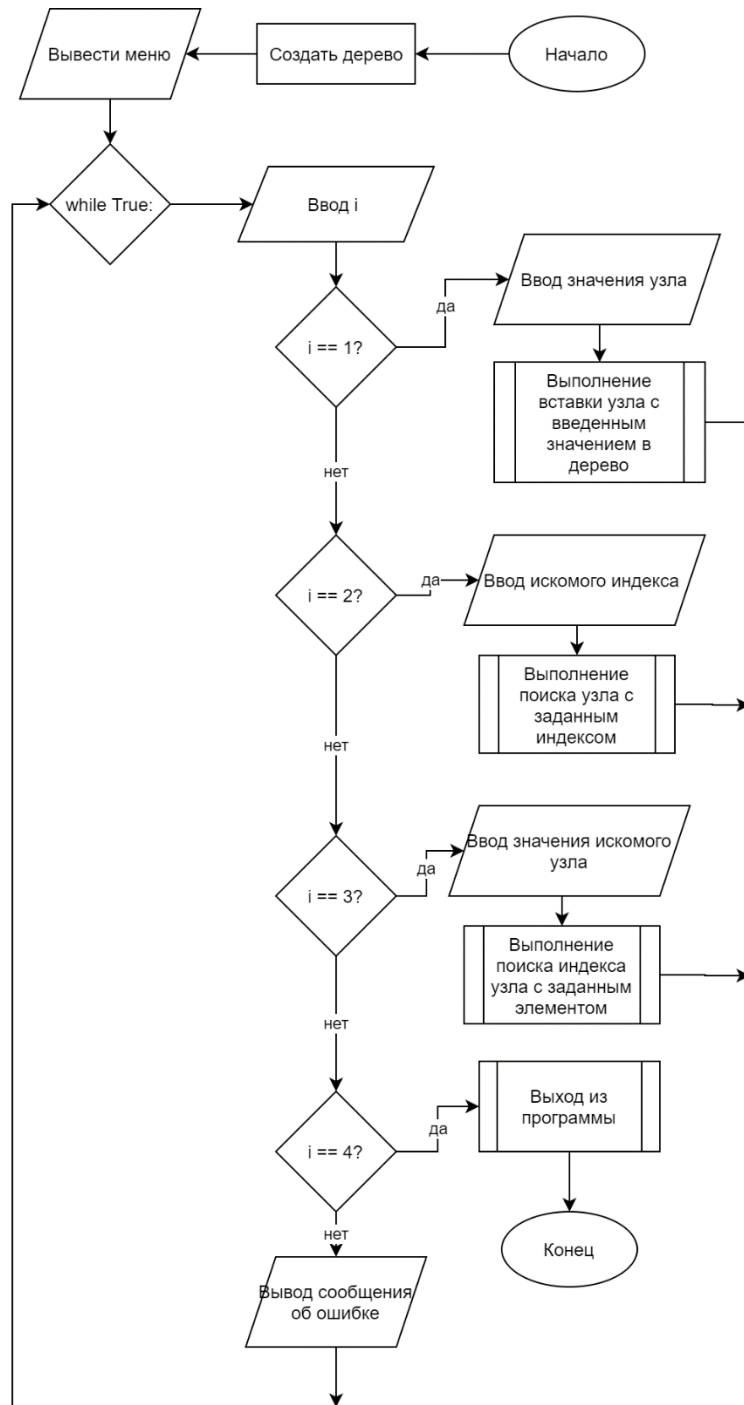


Рисунок 18 – Схема работы программы.

## **2 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

### **2.1 Организация процесса тестирования программного обеспечения**

Тестирование программного обеспечения – это оценка разрабатываемого программного продукта, чтобы проверить его возможности, способности и соответствие ожидаемым результатам. Различные наборы тест-кейсов и стратегий направлены на достижение общей цели – устранение багов и ошибок в коде, и обеспечение точной и оптимальной производительности программного обеспечения. [2]

Сегодня широко распространенными являются следующие методы тестирования:

1. Модульное тестирование.
2. Интеграционное тестирование.
3. Системное тестирование.

Модульное тестирование подразумевает тестирование на объектном уровне. Каждый модуль подвергается индивидуальной проверке. На данном этапе создаются тест-коды, которые проверяют, ведет ли программное обеспечение себя так, как задумывалось.

Интеграционное тестирование подразумевает тестирование собранных в единую систему модулей. На данном этапе тестируется в первую очередь интерфейс программного обеспечения. При интеграционном тестировании могут применяться две стратегии:

1. Нисходящее (следует архитектурному сооружению системы, а главный модуль тестируется отдельно).
2. Восходящее (осуществляется из нижней части потока управления).

Системное тестирование подразумевает тестирование системы в целом на наличие ошибок и багов. На данном этапе выполняется проверка

правильности сопряжения аппаратных и программных компонентов всей системы.

## 2.2 Тестирование элементов разработанного программного обеспечения

Тестирование элементов разработанного программного обеспечения производится с помощью структурного тестирования или тестирования «белого ящика». В ходе тестирования рассматриваются внутренние элементы программы и связи между ними.

Поскольку в разработанной программе реализовано большое число функций, то для структурного тестирования были выбраны функции, которые реализуют основную логику работы программы - «insert», «os\_select», «os\_rank».

Функция «insert» осуществляет вставку нового узла в красно-чёрное дерево. На рисунке 19 представлен потоковый граф для функции «insert».

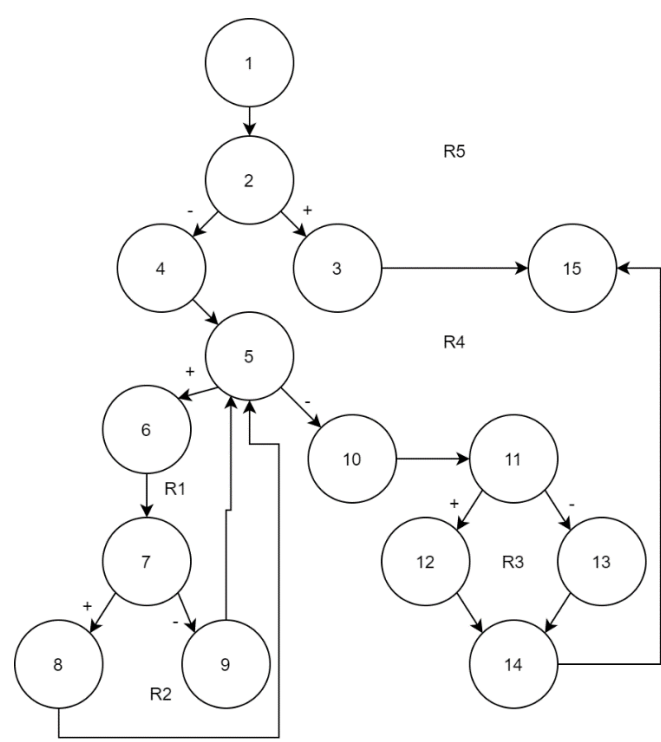


Рисунок 19 – Потокосый граф функции «insert».

Цикломатическая сложность данной функции равна 5. Определим возможные пути работы функции:

Путь 1: 1-2-3-15.

Путь 2: 1-2-4-5-6-7-8...5-10-11-12-14-15.

Путь 3: 1-2-4-5-6-7-8...5-10-11-13-14-15.

Путь 4: 1-2-4-5-6-7-9...5-10-11-12-14-15.

Путь 5: 1-2-4-5-6-7-9...5-10-11-13-14-15.

Тестовые варианты:

ТВ1. Исходные данные: tree = Null, key = 5.

Ожидаемый результат: ROOT IS 5, tree = {5}.

ТВ2. Исходные данные: tree = {5}, key = 3.

Ожидаемый результат: tree = {3, 5}.

ТВ3. Невозможно подобрать исходные данные.

ТВ4. Невозможно подобрать исходные данные.

ТВ5. Исходные данные: tree = {5}, key = 7.

Ожидаемый результат: tree = {5, 7}.

Функция «os\_select» выполняет поиск  $i$ -ого узла в дереве и возвращает его. На рисунке 20 представлен потоковый граф для функции «os\_select».

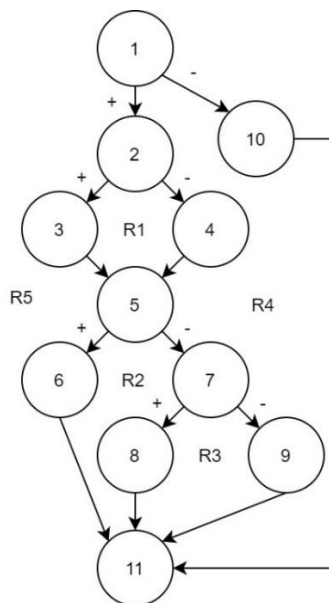


Рисунок 20 – Потоковый граф для функции «os\_select».



Цикломатическая сложность данной функции равна 5. Определим возможные варианты пути работы программы:

Путь 1: 1-10-11.

Путь 2: 1-2-3-5-6-11.

Путь 3: 1-2-4-5-6-11.

Путь 4: 1-2-4-5-7-8-11.

Путь 5: 1-2-4-5-7-9-11.

Тестовые варианты:

ТВ1. Исходные данные:  $tree = \{5, 6, 3\}$ ,  $i = 4$ .

Ожидаемый результат: ERROR! OUT OF RANGE!

ТВ2. Исходные данные:  $tree = \{5, 2\}$ ,  $i = 2$ .

Ожидаемый результат:  $key = 5$ .

ТВ3. Исходные данные:  $tree = \{5\}$ ,  $i = 1$ .

Ожидаемый результат:  $key = 5$ .

ТВ4. Невозможно подобрать исходные данные.

ТВ5. Исходные данные:  $tree = \{5, 8\}$ ,  $i = 2$ .

Ожидаемый результат:  $key = 8$ .

Функция «os\_rank» выполняет поиск заданного узла в дереве и возвращает его порядковую статистику. На рисунке 21 представлен потоковый граф для функции «os\_rank».

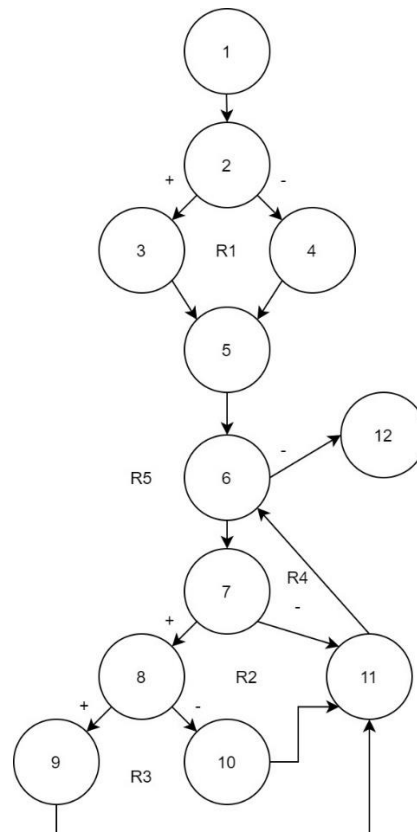


Рисунок 21 – Поточковый граф для функции «os\_rank».

Цикломатическая сложность данной функции равна 5. Определим возможные пути работы программы:

Путь 1: 1-2-3-5-6-12.

Путь 2: 1-2-4-5-6-12.

Путь 3: 1-2-3-5-6-7-8-9-11...6-12.

Путь 4: 1-2-3-5-6-7-8-10-11...6-12.

Путь 5: 1-2-3-5-6-7-11...6-12.

Тестовые варианты:

ТВ1. Исходные данные:  $tree = \{5, 2\}$ ,  $x = 5$ .

Ожидаемый результат:  $r = 2$ .

ТВ2. Исходные данные:  $tree = \{5, 10\}$ ,  $x = 5$ .

Ожидаемый результат:  $r = 1$ .

ТВ3. Исходные данные:  $tree = \{5, 10, 1, 8\}$ ,  $x = 10$ .

Ожидаемый результат:  $r = 4$ .

ТВ4. Невозможно подобрать исходные данные.

TB5. Исходные данные:  $tree = \{5, 2\}$ ,  $x = 5$ .

Ожидаемый результат:  $r = 2$ .

Листинг тестов приведен в Приложении В.

### **2.3 Тестирование интеграций модулей разработанного программного обеспечения**

Интеграционное тестирование – это тип тестирования, при котором программные модули объединяются логически и тестируются как группа. Как правило, программный продукт состоит из нескольких программных модулей, написанных разными программистами. Целью тестирования является выявление багов при взаимодействии между этими программными модулями и проверка обмена данными между этими модулями. [5]

В начале необходимо подключить «Node.py» к «RBTree.py». Эти файлы формируют модуль, описывающий красно-чёрное дерево. Затем «RBTree.py» подключается к «main.py», с помощью которого пользователь может взаимодействовать с программой. Помимо этого, к модулям «Node.py» и «RBTree.py» подключается сторонний модуль «termcolored».

Исходя из вышесказанного, первоначально проводилось тестирования модуля «Node.py». После подключения данного модуля к «RBTree.py» было произведено тестирование всех функций, реализованных в «RBTree.py». В самом конце аналогично тестировался «main.py».

В результате выполнения подключения модулей ошибок интеграции выявлено не было.

### **2.4 Тестирование правильности разработанного программного обеспечения**

Тестирование правильности разработанного программного обеспечения производится с помощью функционального тестирования или тестирования «чёрного ящика». В ходе тестирования рассматриваются возможные входные и выходные данные.

В данном тестировании будет использован метод диаграмм причин-следствий.

Определим причины:

- s1 – добавить узел.
- s2 – данные введены корректно.
- s3 – поиск ключа узла по индексу.
- s4 – поиск индекса узла по ключу.
- s5 – данные введены некорректно
- s6 – выход из программы.

Определим следствия:

- r1 – добавление узла в дерево.
- r2 – вывод искомого по индексу ключа узла.
- r3 – вывод искомого по ключу индекса узла.
- r4 – вывод сообщения об ошибке.
- r5 – корректное завершение работы программы.

Исходя из определенных причин и следствий, для реализованной программы был составлен граф. Граф причинно-следственных связей изображен на рисунке 22.

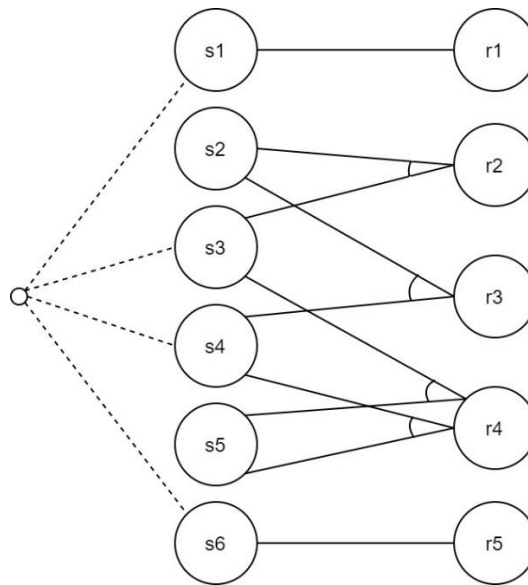


Рисунок 22 – Граф причинно-следственных связей.

Для графа причинно-следственных связей было составлена таблица решений:

	1	2	3	4	5	6
s1	1	0	0	0	0	0
s2	0	1	1	0	0	0
s3	0	1	0	1	0	0
s4	0	0	1	0	1	0
s5	0	0	0	1	1	0
s6	0	0	0	0	0	1
r1	1	0	0	0	0	0
r2	0	1	0	0	0	0
r3	0	0	1	0	0	0
r4	0	0	0	1	1	0
r5	0	0	0	0	0	1

Тестовые варианты:

ТВ1. Исходные данные: Пункт меню 1, node = 5.

Ожидаемый результат: tree = {5}, 5 is Root.

TB2. Исходные данные: Пункт меню 2,  $tree = \{1,3,5\}$ ,  $i = 2$ .

Ожидаемый результат: 3.

TB3. Исходные данные: Пункт меню 3,  $tree = \{1,3,5\}$ ,  $key = 5$ .

Ожидаемый результат: 3.

TB4. Исходные данные: Пункт меню 2,  $tree = \{1,3,5\}$ ,  $i = 5$ .

Ожидаемый результат: ERROR!

TB5. Исходные данные: Пункт меню 3,  $tree = \{1,3,5\}$ ,  $key = 6$ .

Ожидаемый результат: ERROR!

TB6. Исходные данные: Пункт меню 4.

Ожидаемый результат: корректное завершение работы программы.

## 2.5 Системное тестирование разработанного программного обеспечения

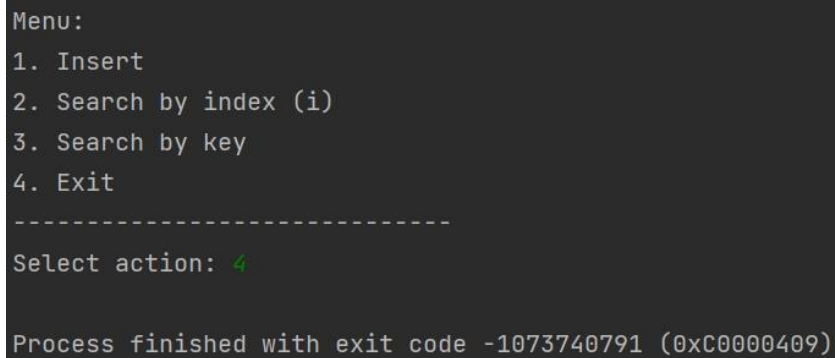
Системное тестирование программного обеспечение предполагает тестирование полной, интегрированной системы, с целью проверки соответствие системы исходным требованиям. Основной целью тестирования же является проверка функциональных и не функциональных требований к системе в целом.

```
Menu:
1. Insert
2. Search by index (i)
3. Search by key
4. Exit
-----
Select action:
```

Рисунок 23 – Главный экран программы.

Тестирование восстановления – это тип нефункционального тестирования, выполняемый для определения того, насколько быстро система сможет восстановиться после системного или аппаратного сбоя.

В разработанной программе предусмотрен вариант корректного завершения работы. Он вызывается при вводе пользователем значения «4».



```
Menu:
1. Insert
2. Search by index (i)
3. Search by key
4. Exit
-----
Select action: 4

Process finished with exit code -1073740791 (0xC0000409)
```

Рисунок 24 – Корректное завершение работы программы.

Однако, поскольку программа не сохраняет никаких данных в процессе своей работы, при некорректном завершении работы также не будет утеряно никаких данных.

При разработке программы не было задано никаких требований к безопасности данных, а также к самим данным. Из этого следует, что проведение тестирования безопасности проводить не нужно.

Таким образом, все виды системного тестирования пройдены.

### 3 ОЦЕНКА КАЧЕСТВА РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

#### 3.1 Статическая оценка качества разработанного программного обеспечения

Статическая оценка качества разработанного программного обеспечения путем расчёта метрик Холстеда.

Определим измеримые характеристики программы:

Таблица 1 – Операторы программы.

№	Оператор	main.py	RBTree.py	Node.py	Всего
1	=	5	105	13	123
2	==	4	19	2	25
3	+	0	13	4	17
6	print	9	31	2	42
8	if	1	23	2	26
9	else	1	14	1	16
10	elif	3	10	2	15
11	input	4	0	0	4
12	return	0	6	0	6
13	(	25	76	8	109
14	)	25	76	8	109
15	[	0	8	0	8
16	]	0	8	0	8
17	while	1	5	0	6
18	is	0	4	0	4
19	is not	0	3	0	3
20	continue	0	2	0	2



21	<	0	4	0	4
----	---	---	---	---	---

Таблица 2 – Операнды программы.

№	Операнд	main.py	RBTree.py	Node.py	Всего
1	1	1	14	4	15
2	2	1	0	0	1
3	3	1	0	0	1
4	4	1	0	0	1
5	first_tree	6	0	0	6
6	i	5	5	0	10
7	node	0	150	0	150
8	root	1	26	0	27
9	y	0	35	0	35
10	p	0	12	0	12
11	current_node	0	12	0	12
12	key	2	39	5	46
13	last_node	0	8	0	8
14	potential_parent	0	2	0	2
15	f	0	2	0	2
16	uncle	0	12	0	12
17	r	0	12	0	12
18	tree	0	3	0	3
19	red	0	30	2	32
20	left	0	36	6	42
21	right	0	30	6	36
22	parent	0	77	1	78
23	size	0	26	10	36
24	value	2	0	0	2

25	index	2	0	0	2
----	-------	---	---	---	---

На основе представленных выше таблиц, были вычислены следующие метрики:

n1 =	21	число уникальных операторов
n2 =	25	число уникальных операндов
N1 =	527	всего операторов
N2 =	583	всего операндов
n =	46	
n' =	52	словарь программы
N =	1110	длина программы
N' =	208.3351	теоретическая длина программы
V =	6131.154	объем программы
V' =	1187.602	потенциальный объем программы
L =	0.1937	уровень качества тестирования
L' =	0.004084	
EC =	750637.2	сложность понимания
D =	244.86	трудоемкость кодирования
T =	41702.06	время кодирования
I =	25.03943	информационное содержание программы
$\delta$ =	230.0378	уровень языка

Исходя из полученных метрик можно сделать вывод, что уровень реализации разработанной программы низок ввиду того, что потенциальный объем программы значительно меньше её реального объема.

### **3.2 Динамическая оценка качества разработанного программного обеспечения**

Динамическая оценка качества разработанного программного обеспечения предполагает проведение одного из следующих видов нагрузочного тестирования: тестирование производительности, стабильности, стрессовое тестирование или тестирование восстановления. Поскольку

тестирование восстановления было проведено ранее, выбор был сделан в пользу тестирования производительности.

Для проведения тестирования производительности было измерено время работы программы при разных величинах красно-чёрного дерева. Ввиду непостоянности времени поиска, было проведено 10 тестов с наполнением дерева и поиском по нему, после чего вычислено среднее значение. Результаты данного тестирования представлены на Рисунке 25.

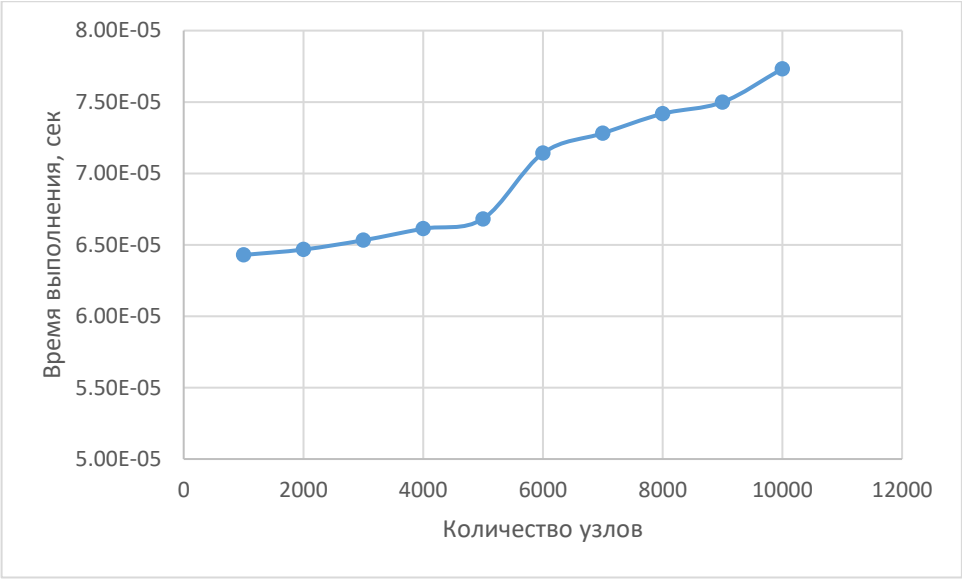


Рисунок 25 – График зависимости времени работы от количества узлов в красно-чёрном дереве.

В процессе проведения тестирования производительности критических ошибок не было выявлено.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсовой работы мы проанализировали и выбрали метод представления структуры данных типа красно-чёрное дерево, разработали алгоритмы программы и реализовали её на языке программирования высокого уровня Python. После было проведено комплексное тестирование разработанной программы и произведена оценка качества программного обеспечения. Таким образом, мы достигли поставленной цели.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Томас Кормен. Алгоритмы. Построение и анализ. – 3-е изд. Москва, 1324 с. – Текст: электронный. (дата обращения 01.05.2021)
2. Juice-Health. Методы тестирования программного обеспечения. [Электронный ресурс]. – Режим доступа: <http://juice-health.ru/program/software-testing/495-software-testing-methods> (дата обращения 05.05.2021)
3. Habr. Красно-черные деревья: кратко и ясно. [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/330644/> (дата обращения 01.05.2021)
4. Habr. Балансировка красно-черных деревьев – три случая [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/company/otus/blog/472040/> (дата обращения 03.05.2021)
5. Logoscon. Интеграционное тестирование – [Электронный ресурс]. – Режим доступа: [https://logrocon.ru/news/intgration\\_testing](https://logrocon.ru/news/intgration_testing) (дата обращения 08.05.2021)

## Листинг кода программы

## Файл main.py

```

from RBTree import *
global first_tree

first_tree = RBTree()

def main():
    first_tree = RBTree()
    print("Menu:")
    print("1. Insert")
    print("2. Search by index (i)")
    print("3. Search by key")
    print("4. Exit")
    print("-----")
    while True:
        i = int(input("Select action: "))
        if i == 1:
            key = int(input("Enter a key: "))
            first_tree.insert(key)
        elif i == 2:
            index = int(input("Enter searching index:
"))
            print("Key", ">>> ", end="")
            first_tree.os_select(first_tree.root, index))
        elif i == 3:

```

```

        value = int(input("Enter searching key: "))
        print("Index          >>          ",
first_tree.os_rank(first_tree, value))
        elif i == 4:
            os.abort()
        else:
            print("ERROR! Wrong value.")

main()

```

### **Файл Node.py**

```

import os
from termcolor import colored

class Node:
    def __init__(self, key):
        self.key = key
        self.red = True
        self.left = None
        self.right = None
        self.parent = None
        self.size = 1

    def print_node(self):
        if self.red:
            print("{0}|{1}".format(colored(self.key,
'red'), self.size))

```

```

        else:
            print("{0}|{1}".format(self.key,
self.size))

    def fix(self):
        if self.right != None and self.left != None:
            self.size = self.right.size + self.left.size
+ 1

        elif self.right == None and self.left != None:
            self.size = self.left.size + 1
        elif self.right != None and self.left == None:
            self.size = self.right.size + 1

```

### **Файл RBTtree.py**

```

from Node import *
from termcolor import colored

class RBTtree:
    def __init__(self):
        self.root = None

    def fix_size(self, node):
        while node != self.root:
            node.fix()
            node = node.parent
            self.fix_size(node)
        else:
            node.fix()

```



```

def print_root(self):
    print(self.root.key, self.root.size,
self.root.red)

def left_rotate(self, node):
    print("LEFT ROTATE ", node.key)
    y = node.right
    node.right = y.left
    if y.left != None:
        y.left.parent = node
    p = node.parent
    y.parent = p
    if node == self.root:
        self.root = y
        print("New root >> {}".format(y.key))
    elif node == p.left:
        p.left = y
    elif node == p.right:
        p.right = y

    y.left = node
    node.parent = y
    # size
    y.size = node.size
    if node.right != None and node.left != None:
        node.size = node.right.size + node.left.size
+ 1

    elif node.right == None and node.left != None:

```

```

        node.size = node.left.size + 1
    elif node.right != None and node.left == None:
        node.size = node.right.size + 1
    else:
        node.size = 1

def right_rotate(self, node):
    print("RIGHT ROTATE ", node.key)
    y = node.left
    node.left = y.right
    if y.right != None:
        y.right.parent = node
    p = node.parent
    y.parent = p
    if node == self.root:
        self.root = y
        print("New root >> {}".format(y.key))
    elif node == p.left:
        p.left = y
    elif node == p.right:
        p.right = y

    y.right = node
    node.parent = y
    # size
    y.size = node.size
    if node.right != None and node.left != None:
        node.size = node.right.size + node.left.size

```

```

elif node.right == None and node.left != None:
    node.size = node.left.size + 1
elif node.right != None and node.left == None:
    node.size = node.right.size + 1
else:
    node.size = 1

def search(self, key):
    current_node = self.root
    while current_node is not None and key !=
current_node.key:
        if key < current_node.key:
            current_node = current_node.left
        else:
            current_node = current_node.right
    #         print('Parent         {}         is
{}'.format(current_node.key, current_node.parent.key))
    #print(colored(current_node.key, 'red'))
    return current_node

def insert(self, key):
    node = Node(key)                                #1
    # Base Case - Nothing in the tree
    if self.root is None:                            #2
        node.red = False                             #3
        self.root = node                             #3
        print('ROOT                                IS          -
{}'.format(self.root.key))#3
    return                                           #15

```

```

43
last_node = self.root #4
while last_node is not None: #5
    potential_parent = last_node #6
    if node.key < last_node.key: #7
        last_node = last_node.left #8
    else: #9
        last_node = last_node.right #9
# Assign parents and siblings to the new node
node.parent = potential_parent #10
print('NODE KEY- {0} NODE PARENT - {1}
'.format(node.key,
node.parent.key))#10
    if node.key < node.parent.key: #11
        node.parent.left = node #12
        print('GO LEFT < NODE PARENT PARENT LEFT KEY
- ',
node.parent.key) #12
    else: #13
        node.parent.right = node #13
        print('GO RIGHT > NODE PARENT PARENT RIGHT
KEY - ',
node.parent.key) #13
node.left = None #14
node.right = None #14
f = node.parent #14
self.fix_size(f) #14
self.fix_tree(node) #14

```

```

def fix_tree(self, node):
    print('NODE      PARENT      RED      -
    {}'.format(node.parent.red))
    try:
        while node is not self.root and
node.parent.red is True:
            print('FIX>> NODE KEY - {} '
                  'NODE      PARENT      KEY      -      {}'.format(node.key, node.parent.key))
            if node.parent ==
node.parent.parent.left: # если отец является левым сыном
                try:
                    uncle = node.parent.parent.right
# то дядя - правый сын деда
                    print('[LEFT] UNCLE RED - {} '
                          'UNCLE      KEY      -      {}      PARENT
PARENT      KEY      -      {}'.format(uncle.red,      uncle.key,
node.parent.parent.key))
                    if uncle.red: # case 1 красный
дядя
                        node.parent.red = False
                        uncle.red = False
                        node.parent.parent.red =
True
                        node = node.parent.parent
                        if node != self.root:
                            print('NODE      RED      -      {}
UNCLE RED - {} PARENT RED - '
                                  '{}'.format(

```

```

        colored(node.red,
'red',
        colored(uncle.red,
'yellow',
        colored(node.parent.red, 'yellow',
attrs=['reverse', 'blink'])))
    else:
        print('NODE IS ROOT')
    else:
        if node ==
node.parent.right:
        # This is Case 2
        print('in TEST>>>>',
node.key)
        node = node.parent
        print('AFTER TEST>>>>',
node.key)
        self.left_rotate(node)
    # This is Case 3
    node.parent.red = False
    node.parent.parent.red =
True

```

```

self.right_rotate(node.parent.parent)

        except AttributeError:
            print("No uncle")
            if node == node.parent.right:
                # This is Case 2
                print('in          TEST>>>>',
node.key)

                node = node.parent
                print('AFTER          TEST>>>>',
node.key)

                self.left_rotate(node)
                # This is Case 3
                node.parent.red = False
                node.parent.parent.red = True

self.right_rotate(node.parent.parent)
        continue

    else:
        try:
            uncle = node.parent.parent.left
            print('[RIGHT] UNCLE RED - {} '
                  'UNCLE          KEY          -
{}'.format(uncle.red, uncle.key))

            if uncle.red:
                # Case 1
                node.parent.red = False

```

```

uncle.red = False
node.parent.parent.red =
True

node = node.parent.parent
if node != self.root:
    print('NODE RED - {}'.format(
        colored(node.red,
'yellow',
attrs=['reverse', 'blink'])),
        colored(uncle.red,
'yellow',
attrs=['reverse', 'blink'])))
    else:
        print('NODE IS ROOT')
else:
    if node ==
node.parent.left:
        # This is Case 2
        print('in TEST>>>>',
node.key)
        node = node.parent

```



```

node.key)

        print('AFTER TEST>>>>',

                self.right_rotate(node)
        # This is Case 3
        node.parent.red = False
        node.parent.parent.red      =

True

self.left_rotate(node.parent.parent)

        except AttributeError:
            print("No Uncle")
            if node == node.parent.left:
                # This is Case 2
                print('in          TEST>>>>',

node.key)

                node = node.parent
                print('AFTER      TEST>>>>',

node.key)

                self.right_rotate(node)
            # This is Case 3
            node.parent.red = False
            node.parent.parent.red = True

self.left_rotate(node.parent.parent)

        continue

        #self.root.red = False
    except AttributeError:

```

```

        print("\n\nTree BUILT")
        self.root.red = False

def os_select(self, root, i):
    try:
        if root.left != None:                #2
            r = root.left.size + 1           #3
        else:                                #4
            r = 1                             #4
        if i == r:                           #5
            return root.key                  #6
        elif i < r:                          #7
            return self.os_select(root.left, i)
#8
        else:                                #9
            return self.os_select(root.right, i -
r)#9
    except AttributeError:                  #10
        print("ERROR! OUT OF RANGE!")      #10

def os_rank(self, tree, x):
    node = tree.search(x)                  #
1
    if node.left != None:                  #
2
        r = node.left.size + 1            #
3
    else:                                  #
4

```

```

                                     r = 1                                     #
4
                                     y = node                             #
5
                                     while y != tree.root:                #
6
                                     if y == y.parent.right:              #
7
                                     if y.parent.left != None:            #
8
                                     r = r + y.parent.left.size + 1      #
9
                                     else:                                  #
10
                                     r = r + 1                             #
10
                                     y = y.parent                          #
11
                                     return r                               #
12
```

## Листинг файла test\_main.py

```
import unittest

import main
from RBTree import *
import Node

tree1 = RBTree()

class Test_select(unittest.TestCase):
    def test_TB1(self):
        tree1.root = None
        tree1.insert(5)
        tree1.insert(6)
        tree1.insert(3)
        self.assertEqual(tree1.os_select(tree1.root,
4), None)

    def test_TB2(self):
        tree1.root = None
        tree1.insert(5)
        tree1.insert(2)
        self.assertEqual(tree1.os_select(tree1.root,
2), 5)

    def test_TB3(self):
        tree1.root = None
```

```

        tree1.insert(5)
        self.assertEqual(tree1.os_select(tree1.root,
1), 5)

```

```

def test_TB5(self):
    tree1.root = None
    tree1.insert(5)
    tree1.insert(8)
    self.assertEqual(tree1.os_select(tree1.root,
2), 8)

```

```

class Test_rank(unittest.TestCase):
    def test_TB1(self):
        tree1.root = None
        tree1.insert(5)
        tree1.insert(2)
        self.assertEqual(tree1.os_rank(tree1, 5), 2)

    def test_TB2(self):
        tree1.root = None
        tree1.insert(5)
        tree1.insert(10)
        self.assertEqual(tree1.os_rank(tree1, 5), 1)

    def test_TB3(self):
        tree1.root = None
        tree1.insert(5)
        tree1.insert(10)

```

```
tree1.insert(1)
tree1.insert(8)
self.assertEqual(tree1.os_rank(tree1, 10), 4)

def test_TB5(self):
    tree1.root = None
    tree1.insert(5)
    tree1.insert(2)
    self.assertEqual(tree1.os_rank(tree1, 5), 2)
```