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

Кафедра программной инженерии

Работа допущена к защите

______________Руководитель

«____»_____________20___г.

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

на тему: «Пополнение структур данных: динамические порядковые статистики»

Студент _________________ Беликов П.Г.

Шифр 180818

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 81ПГ

Руководитель __________________ Артёмов А.В.

Оценка: «________________» Дата ______________

Орел 2020

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

Кафедра программной инженерии

УТВЕРЖДАЮ:

____________Зав. кафедрой

«___» _____________20__г.

ЗАДАНИЕ
на курсовую работу

по дисциплине «Алгоритмы и структуры данных»

Студент Беликов П.Г. Шифр 180818

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 81ПГ

1 Тема курсовой работы

«Пополнение структур данных: динамические порядковые статистики»

2 Срок сдачи студентом законченной работы «02» декабря 2020

3 Исходные данные

Данные представлены красно – чёрным деревом.

4 Содержание курсовой работы

Анализ и выбор методов представления красно – черного дерева

Проектирование алгоритмов вставки, балансировки, определения i-ой порядковой статистики и определения порядкового номера заданного элемента

Проектирование и реализация программы

5 Отчетный материал курсовой работы

Пояснительная записка курсовой работы; приложение

Руководитель ________________________ Артёмов А.В.

Задание принял к исполнению: «02» октября 2020

Подпись студента___________________

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
1 АНАЛИЗ И ВЫБОР МЕТОДОВ ПРЕДСТАВЛЕНИЯ КРАСНО – ЧЁРНОГО ДЕРЕВА 6
2 ПРОЕКТИРОВАНИЕ АЛОГРИТМОВ 8
2.1 Алгоритмы левого и правого поворотов 8
2.2 Алгоритм балансировки 10
2.3 Алгоритм вставки 14
2.4 Алгоритм определения i-ой порядковой статистики 15
2.5 Алгоритм определения порядкового номера заданного элемента 16
3 РЕАЛИЗАЦИЯ ПРОГРАММЫ 18
ЗАКЛЮЧЕНИЕ 21
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 22
Приложение А 23

ВВЕДЕНИЕ

Целью курсовой работы является решение задач определения i-й порядковой статистики и определение порядкового номера заданного элемента. Данные представлены красно-чёрным деревом.

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

1. Анализ и выбор методов представления красно-чёрного дерева.
2. Разработка алгоритма пополнения структуры данных.
3. Разработка алгоритма определения i-й порядковой статистики.
4. Разработка алгоритма определения порядкового номера заданного элемента.
5. Реализация разработанных алгоритмов и программы.

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]

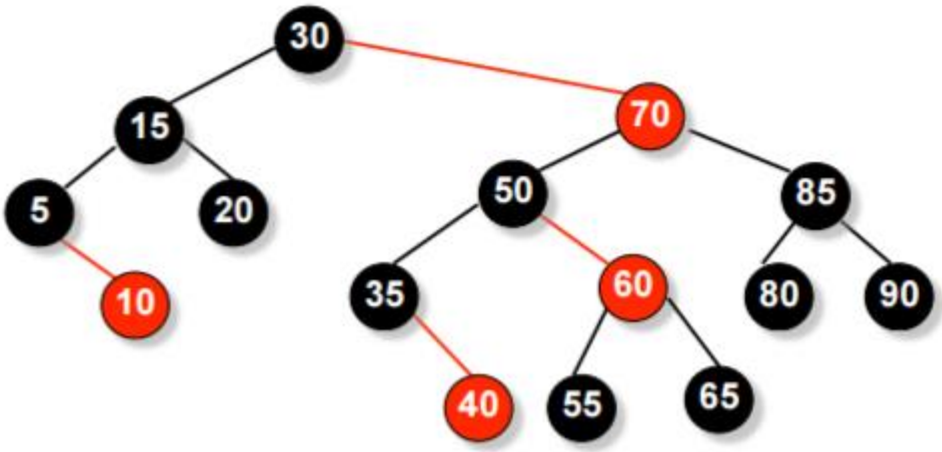

Рисунок 1 – Красно-чёрное дерево. [3]

Каждый узел дерева содержит атрибуты key, color, left, right и parent. Параметр key содержит значение узла, color – его цвет. Параметры parent, left и right содержать 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.

2 ПРОЕКТИРОВАНИЕ АЛОГРИТМОВ

2.1 Алгоритмы левого и правого поворотов

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

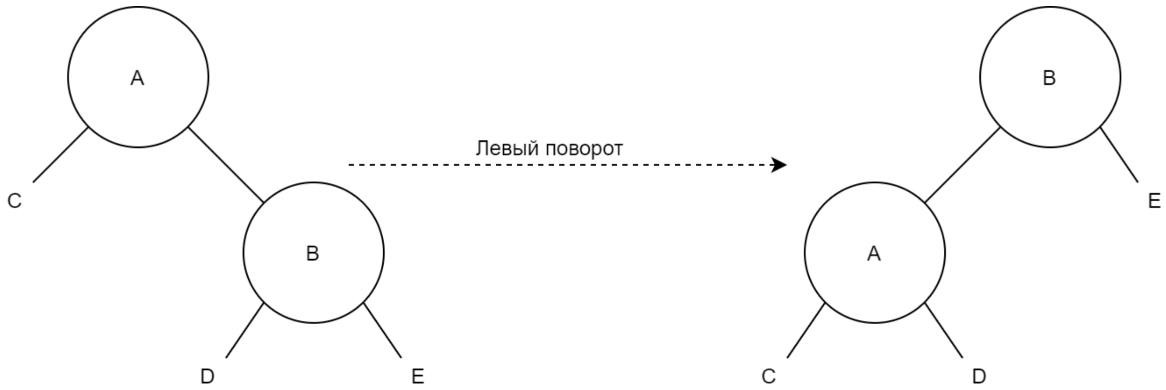

Рисунок 5 – Левый поворот в бинарном дереве поиска.

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.

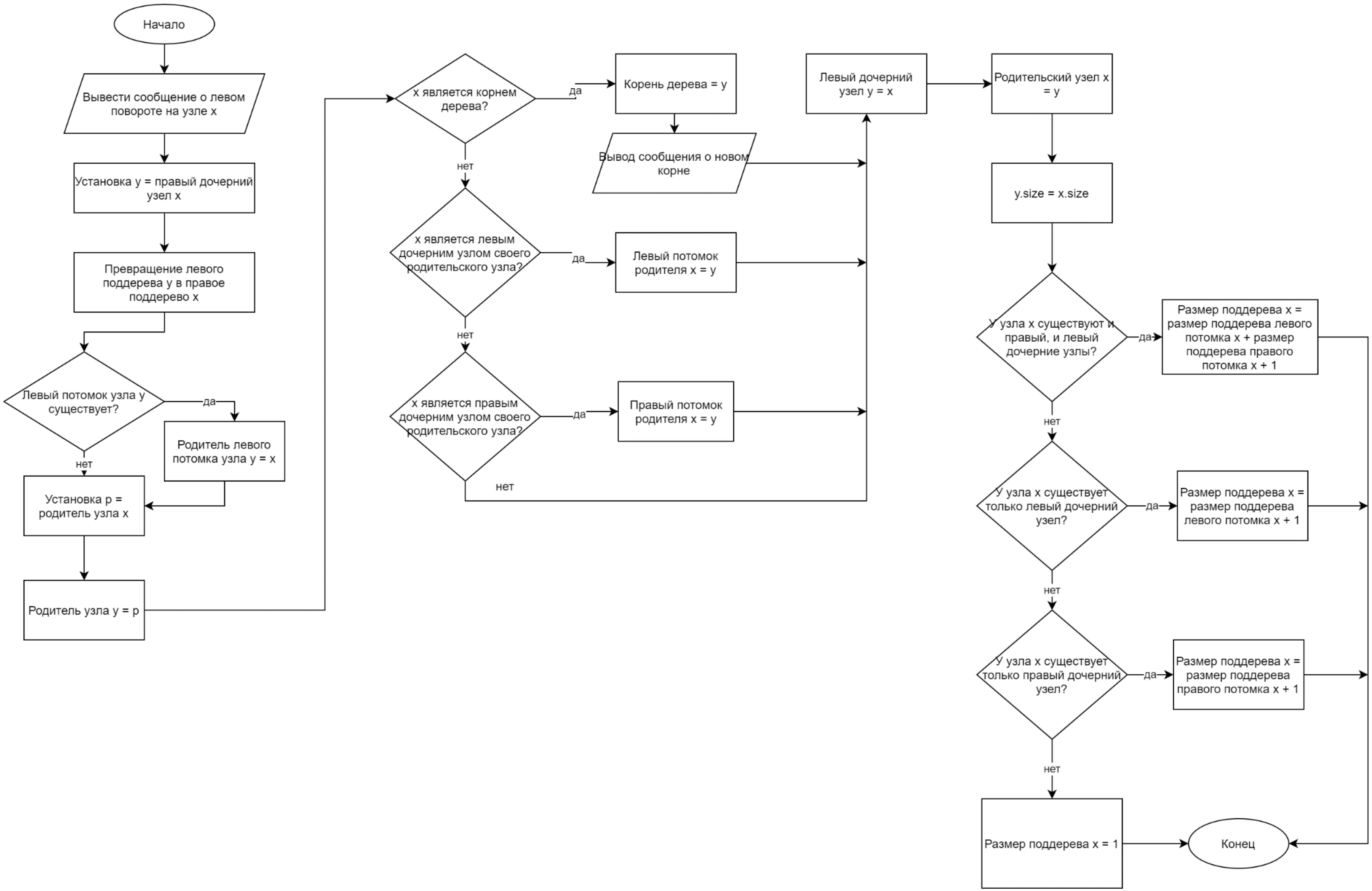

Рисунок 6 – Схема алгоритма левого поворота.

Алгоритм правого поворота симметричен алгоритму левого поворота.

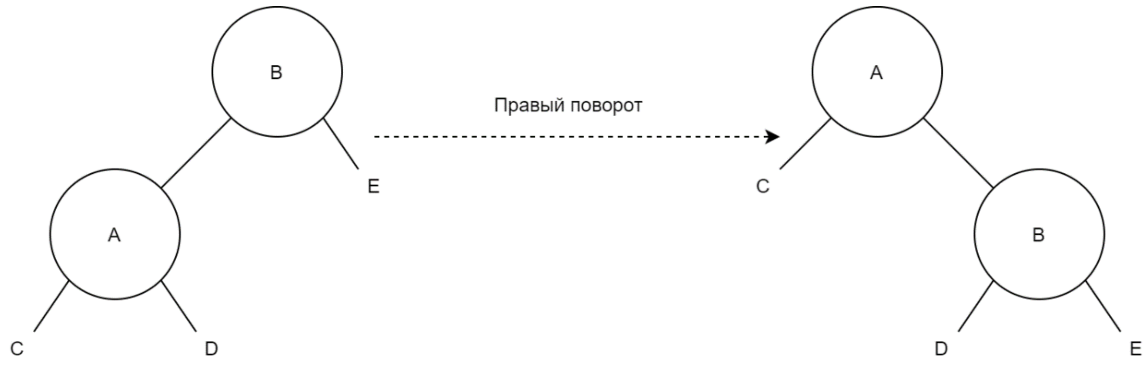

Рисунок 7 – Правый поворот в бинарном дереве поиска.

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 B и A, делая A новым корнем поддерева, правым дочерним узлом которого становится B, а бывший правый потомок узла A – левым потомком B. После происходит пересчет размера поддеревьев узлов A и B. Размер поддерева A приравнивается к старому значению size узла B, а размер поддерева B рассчитывается как сумма размеров поддеревьев его новых правого и левого потомков, увеличенная на единицу. Реализация левого поворота представлена в Приложении А.

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

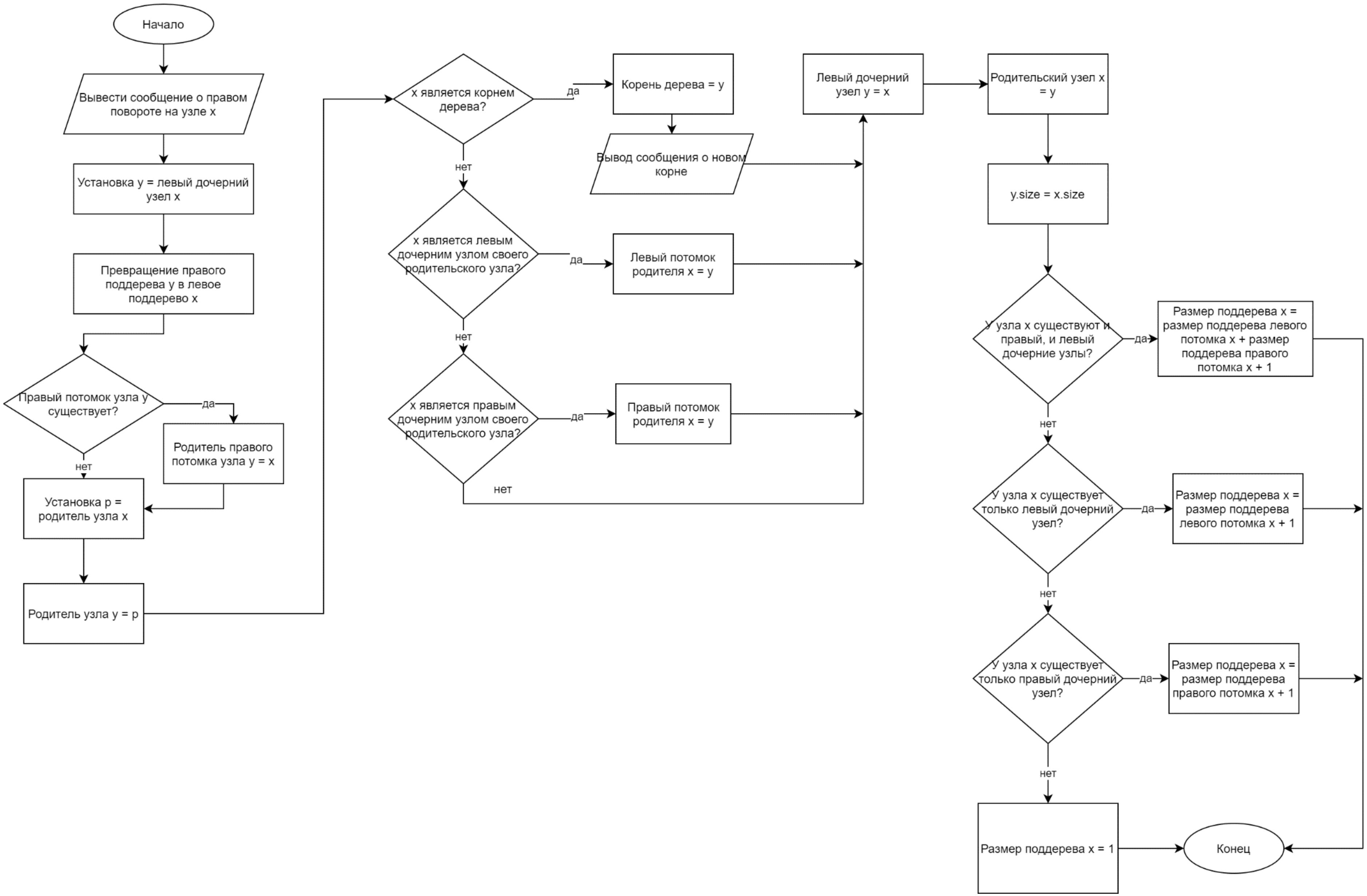

Рисунок 8 – Схема алгоритма правого поворота.

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.

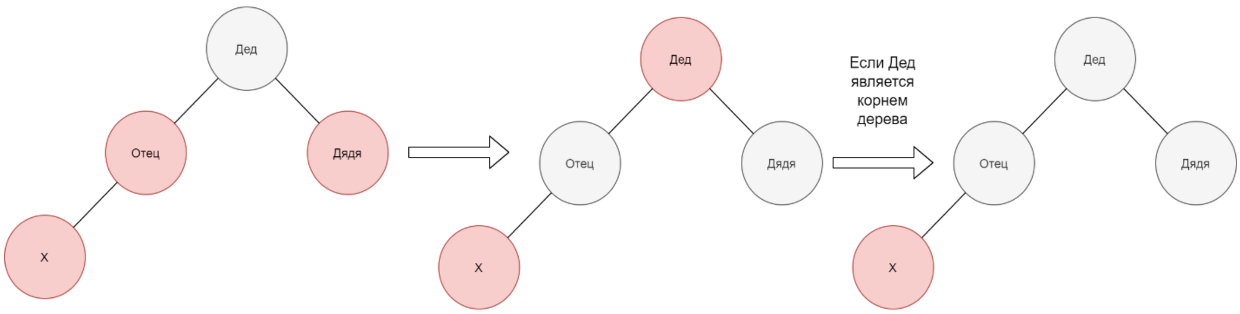

Рисунок 9 – Красный Дядя.

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).

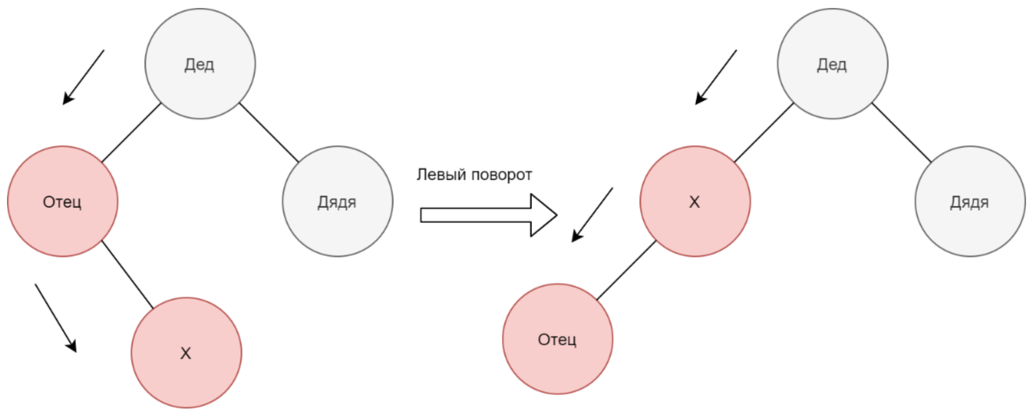

Рисунок 10 – Чёрный Дядя (лево).

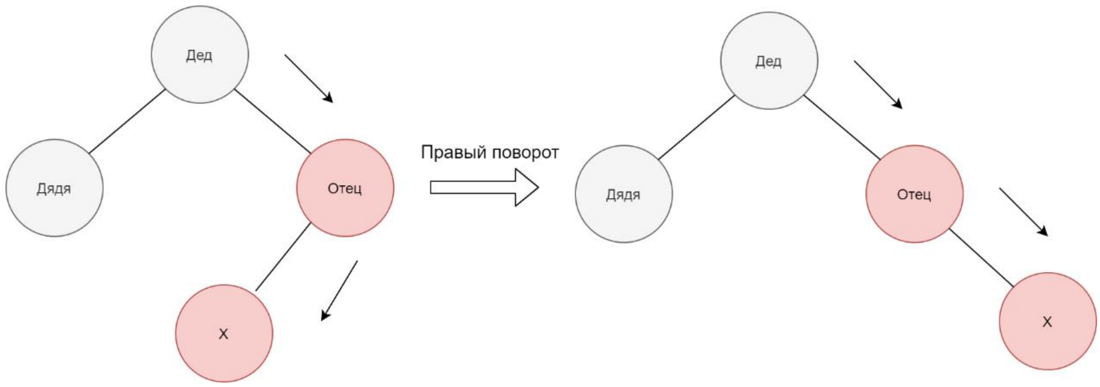

Рисунок 11 – Чёрный Дядя (право).

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

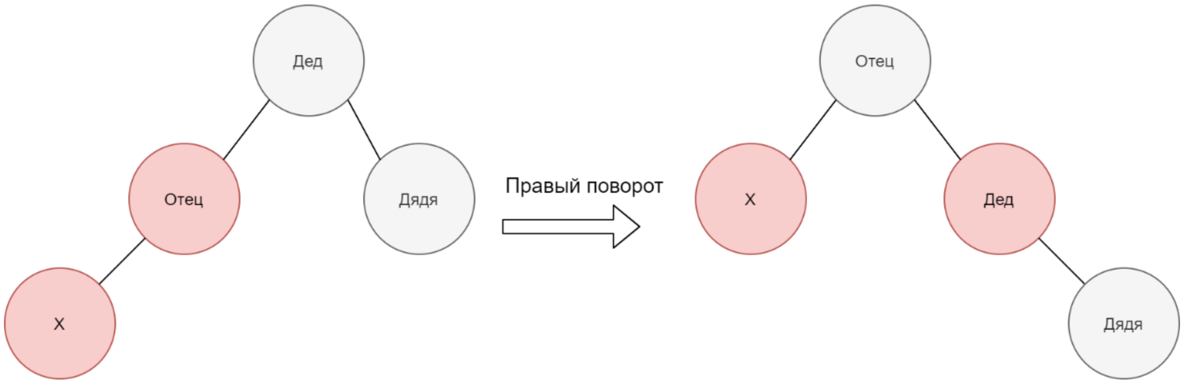

Рисунок 12 – Чёрный Дядя (лево).

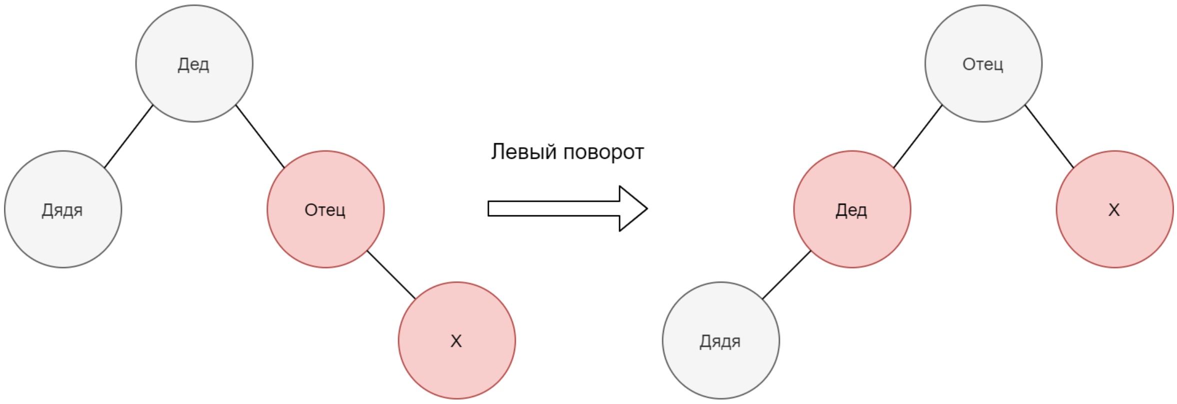

Рисунок 13 – Чёрный Дядя (право).

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

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]

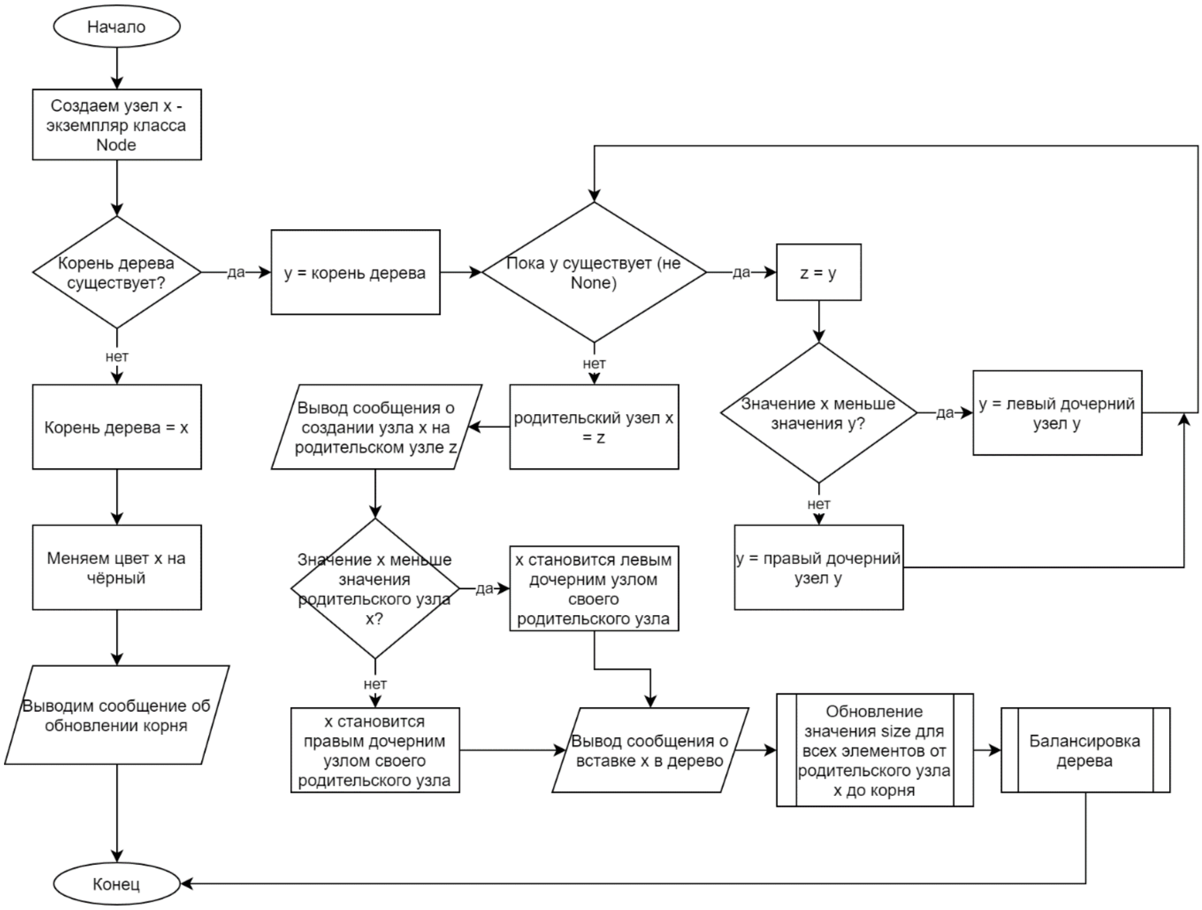

Рисунок 14 – Схема алгоритма вставки в красно-чёрное дерево.

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.

20. Завершить работу.

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.

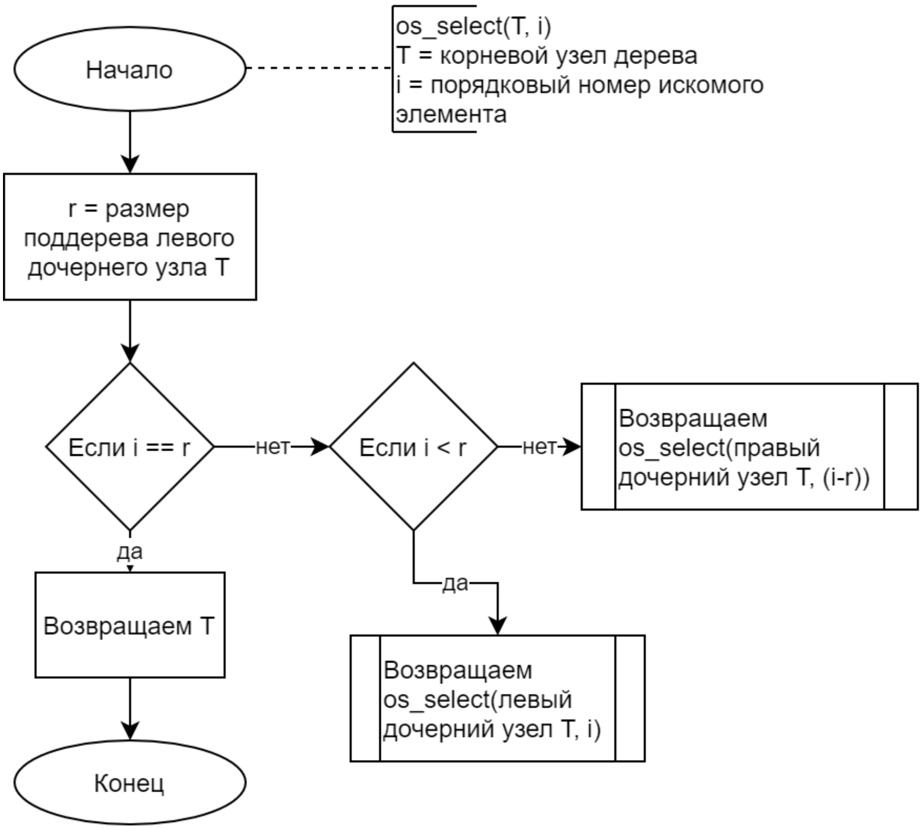

Рисунок 15 – Схема алгоритма поиска i-й порядковой статистики.

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.

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.

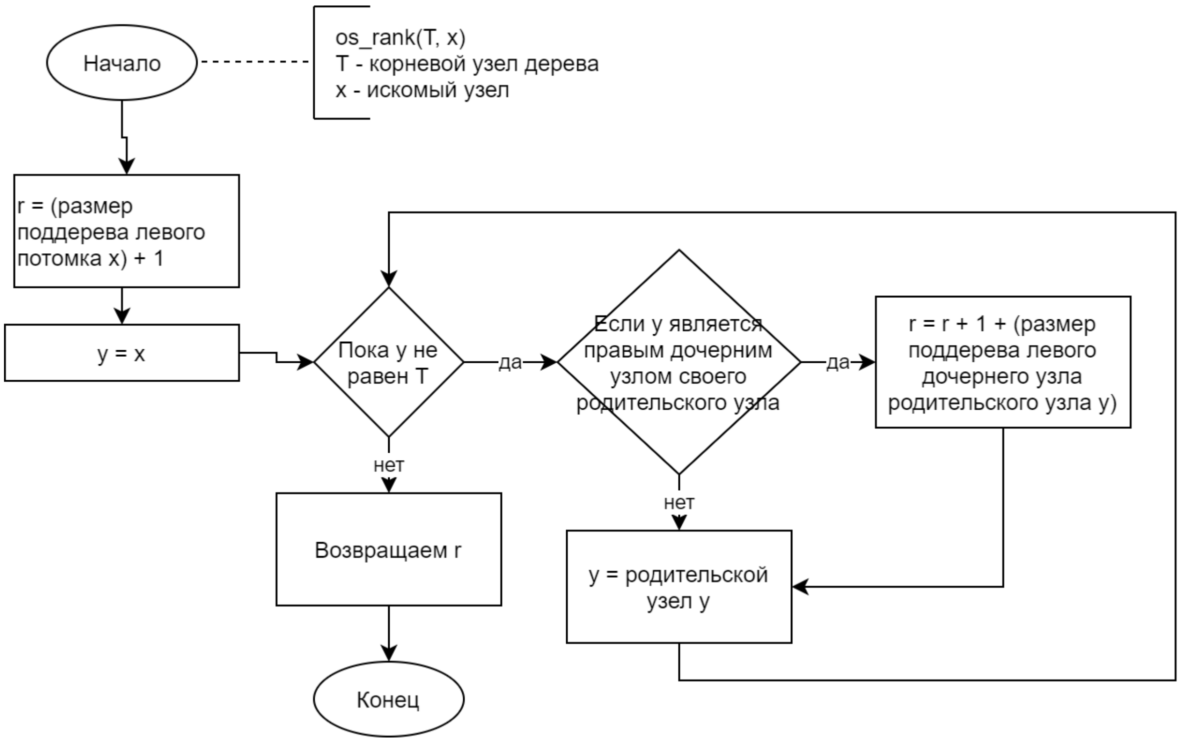

Рисунок 16 – Схема алгоритма определения порядкового номера заданного элемента.

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, Fasle – 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.

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

1. left_rotate(x) – выполняет левый поворот на узле x. Обновляет размер поддеревьев у задействованный в повороте узлов.
2. right_rotate(x) – выполняет правый поворот на узле x. Обновляет размер поддеревьев у задействованный 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 Рисунке17.

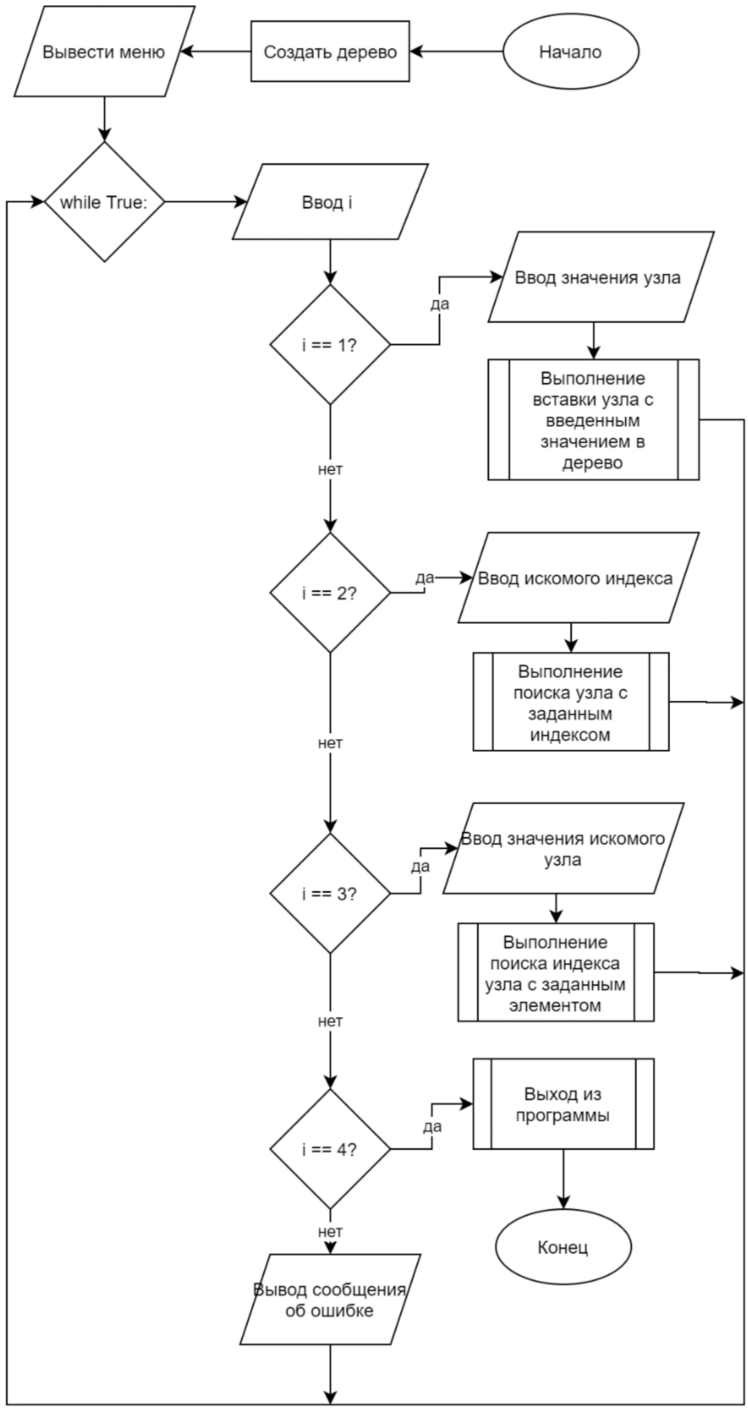

Рисунок 17 – Схема работы программы.

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. 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 25.11.2020)

2. IBM. Красно-черные деревья. [Электронный ресурс]. – Режим доступа: https://www.ibm.com/developerworks/ru/library/l-data_structures_09/index.html (дата обращения 25.11.2020)

3. Habr. Красно-черные деревья: коротко и ясно. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/330644/ (дата обращения 23.11.2020)

4. Habr. Балансировка красно-черных деревьев – три случая [Электронный ресурс]. – Режим доступа: https://habr.com/ru/company/otus/blog/472040/ (дата обращения 28.11.2020)

Приложение А

Листинг кода программы

main.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


class RBTree:
    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 >> {0}".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 >> {0}".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 + 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 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)
    # Base Case - Nothing in the tree
    if self.root is None:
        node.red = False
        self.root = node
        print('ROOT IS - {0}'.format(self.root.key))
        return
    last_node = self.root
    while last_node is not None:
        potential_parent = last_node
        if node.key < last_node.key:
            last_node = last_node.left
        else:
            last_node = last_node.right
    # Assign parents and siblings to the new node
    node.parent = potential_parent
    print('NODE KEY- {0} NODE PARENT - {1} '.format(node.key,
                                      node.parent.key))
    if node.key < node.parent.key:
        node.parent.left = node
        print('GO LEFT < NODE PARENT PARENT LEFT KEY - ',
              node.parent.key)
```

```
        else:
            node.parent.right = node
            print('GO RIGHT > NODE PARENT PARENT RIGHT KEY - ',
                  node.parent.key)
        node.left = None
        node.right = None
        f = node.parent
        self.fix_size(f)
        self.fix_tree(node)

    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',
                                            attrs=['reverse', 'blink']),
                                    colored(uncle.red, 'yellow',
                                            attrs=['reverse', 'blink']),
                                    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 - {} UNCLE RED - {} PARENT RED - '
                  '{}'.format(
                colored(node.red, 'red',
                        attrs=['reverse', 'blink']),
                colored(uncle.red, 'yellow',
                        attrs=['reverse', 'blink']),
                colored(node.parent.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
            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)

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:
            r = root.left.size + 1
        else:
            r = 1
        if i == r:
            return root.key
        elif i < r:
            return self.os_select(root.left, i)
        else:
            return self.os_select(root.right, i - r)
    except AttributeError:
        print("ERROR! OUT OF RANGE!")

def os_rank(self, tree, x):
    node = tree.search(x)
    if node.left != None:
        r = node.left.size + 1
    else:
        r = 1
    y = node
    while y != tree.root:
```

```
            if y == y.parent.right and y.parent.left != None:
                r = r + y.parent.left.size + 1
            elif y == y.parent.right and y.parent.left == None:
                r = r + 1
            y = y.parent
        return r

    # def real_delete_node(self, key): #trash
    #     current_node = self.search(key)
    #     if current_node is None:
    #         return
    #     if current_node.parent is None:
    #         if current_node == self.root:
    #             self.root = None
    #         return
    #     if current_node.parent.left == current_node:
    #         current_node.parent = None
    #     else:
    #         current_node.parent = None


def test_lr():
    first_tree = RBTree()
    first_tree.insert(11)
    first_tree.insert(2)
    first_tree.insert(14)
    first_tree.insert(15)
    first_tree.insert(1)
    first_tree.insert(7)
    first_tree.insert(5)
    first_tree.insert(8)
    first_tree.insert(4)

    first_tree.print_root()
    first_tree.root.left.print_node()
```

```
    first_tree.root.right.print_node()
    first_tree.search(2).right.print_node()
    first_tree.search(2).left.print_node()
    first_tree.search(14).right.print_node()
    first_tree.search(11).right.print_node()
    first_tree.search(11).left.print_node()
    first_tree.search(5).left.print_node()

def test_rr():
    first_tree = RBTree()
    first_tree.insert(11)
    first_tree.insert(2)
    first_tree.insert(20)
    first_tree.insert(1)
    first_tree.insert(16)
    first_tree.insert(22)
    first_tree.insert(15)
    first_tree.insert(17)
    first_tree.insert(18)

    first_tree.print_root()
    first_tree.root.left.print_node()
    first_tree.root.right.print_node()
    first_tree.search(11).right.print_node()
    first_tree.search(11).left.print_node()
    first_tree.search(20).right.print_node()
    first_tree.search(20).left.print_node()
    first_tree.search(2).left.print_node()
    first_tree.search(17).right.print_node()

def test_rot():
    first_tree = RBTree()
    first_tree.insert(16)
    #first_tree.insert(17)
    #first_tree.search(17).red = False
```

```
    first_tree.insert(11)
    first_tree.insert(8)
    #first_tree.insert(8)

    first_tree.print_root()
    first_tree.root.left.print_node()
    first_tree.root.right.print_node()

def test_rr_size():
    first_tree = RBTree()
    first_tree.insert(11)
    first_tree.insert(2)
    first_tree.insert(20)
    first_tree.insert(1)
    first_tree.insert(16)
    first_tree.insert(22)
    first_tree.insert(15)
    first_tree.insert(17)
    first_tree.insert(18)

    print("We FIND >> ", first_tree.os_select(first_tree.root, 6))

def test_rr_size2():
    first_tree = RBTree()
    first_tree.insert(11)
    first_tree.insert(2)
    first_tree.insert(20)
    first_tree.insert(1)
    first_tree.insert(16)
    first_tree.insert(22)
    first_tree.insert(15)
    first_tree.insert(17)
    first_tree.insert(18)

    print("We FIND >> ", first_tree.os_rank(first_tree, 17))
```

```
#test_rr_size2()

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 >> ", 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()
```