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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

ТЕМА: КРАСНО-ЧЕРНОЕ ДЕРЕВО

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2023

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студентка Соц Е.А.

Группа 2384

Тема работы: Красно-черное дерево - вставка

Вариант 13

Исходные данные:

Реализовать красно-черное дерево и провести исследование операции вставки

Содержание пояснительной записки:

«Содержание», «Введение», «Теоретическое описание красно-черного дерева», «Реализация красно-черного дерева и операции вставки», «Исследование красно-черного дерева», «Заключение», «Список использованных источников», «Приложение А. Исходный код программы»

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 13.12.2023

Дата сдачи реферата: 23.12.2023

Дата защиты реферата: 23.12.2023

Студентка _____ Соц Е.А.

Преподаватель _____ Иванов Д.В.

АННОТАЦИЯ

Реализация и анализ производительности красно-черного дерева на Python. Исследование операции вставки, тестирование структуры в общем, классическая теория.

SUMMARY

Implementation and performance analysis of red-black tree in Python. Investigation of insertion operation, structure testing in general, classical theory.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель: реализовать красно-черное дерево на языке Python. Провести исследование операции вставки.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Создать класс красно-черного дерева и разработать нужные методы
2. Провести тесты работоспособности структуры
3. Провести исследование операции вставки, предоставив их анализ с графиками

1. ТЕОРЕТИЧЕСКОЕ ОПИСАНИЕ КРАСНО-ЧЕРНОГО ДЕРЕВА

1.1 Общая теория

Красно-черное дерево – это двоичное дерево поиска, в котором баланс осуществляется на основе «цвета» узла дерева, который принимает только два значения: «красный» и «черный». Все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются черными.

Красно-черные деревья обладают некоторыми особенностями:

1. Узел может быть либо красным, либо черным и имеет двух потомков
2. Корень дерева всегда черный
3. Все листья, не содержащие данных, черные
4. Если узел красный, то оба его потомка должны быть черными
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего, и дерево примерно сбалансировано.

Алгоритм поиска в красно-черном дереве начинается с корня дерева и движется вниз по дереву, сравнивая значение ключа с значением ключа в узле. Если значение ключа меньше, то поиск продолжается в левом поддереве, если больше – в правом поддереве. Этот процесс повторяется до тех пор, пока не будет найден узел с искомым ключом или достигнут листовой узел, что означает, что искомое значение отсутствует в дереве.

Алгоритм вставки начинается с корня дерева и движется вниз по дереву, сравнивая значение ключа с значением ключа в узле. Если значение ключа меньше, то вставка продолжается в левом поддереве, если больше – в правом. Этот процесс повторяется до тех пор, пока не будет найден узел, в который можно вставить новый узел. Новый узел добавляется как листовое дерево, затем дерево балансируется с помощью серии поворотов и перекрасок узлов.

Алгоритм удаления в красно-черном дереве начинается с поиска узла для удаления. Если у удаляемого узла есть два потомка, то его заменяет узел с наименьшим значением в правом поддереве или наибольшим в левом поддереве. Затем узел для удаления удаляется, и дерево балансируется с помощью серии поворотов и перекрасок узлов.

1.2 Временная и пространственная сложность операций

Рассмотрим временную сложность основных операций и случаи, в которых реализуется каждая сложность.

1) Вставка: временная сложность составляет O(log n) в худшем случае, когда дерево уже сбалансировано и может потребовать до 2 поворотов. Однако, в лучшем случае, когда вставка происходит на самом нижнем уровне дерева, вставка может выполняться за константное время O(1).

2) Удаление: временная сложность O(log n) в худшем случае, O(1) в лучшем, рассуждения такие же, как и в операции вставки.

3) Поиск: временная сложность составляет O(log n) в лучшем и худшем случаях, где n – количество узлов в дереве. Это обусловлено тем, что красно-черные деревья являются сбалансированными, и в худшем случае высота дерева не превышает 2 log n.

Пространственная сложность каждой из перечисленных операций – O(1), так как ни одна из операций не требует дополнительной памяти.

2. РЕАЛИЗАЦИЯ КРАСНО-ЧЕРНОГО ДЕРЕВА

2.1 Красно-черное дерево

Для корректной реализации красно-черного дерева было создано два класса: *Node* и *RBTree*.

Класс *Node* представляет собой узел в красно-черном дереве. Каждый узел имеет значение *val*, родительский узел *parent*, левый и правый потомки (*left* и *right*), а также цвет color. При создании нового узла его цвет по умолчанию устанавливается в 1(красный).

Класс *RBTree* представляет собой красно-черное дерево. Он содержит методы для вставки узлов, балансировки дерева после вставки, а также для выполнения левых и правых поворотов.

2.2 Методы RBTree и пример взаимодействия

1) Конструктор *__init__(self)* инициализирует красно-черное дерево. Он создает специальный узел *NULL*, который используется как заглушка для отсутствующих узлов в дереве. Этот узел имеет значение 0, цвет 0 (черный) и не имеет потомков. Затем он устанавливает корнем дерева узел *NULL*.

2) *insertNode(self, key)*

Вставляет новый узел в красно-черное дерево. Он начинает с корня дерева и движется вниз по дереву, сравнивая значение ключа с значением ключа в узле. Если значение ключа меньше, то вставка продолжается в левом поддереве, иначе – в правом. Этот процесс повторяется до тех пор, пока не будет найден узел, в который можно вставить новый узел.

3) *fixInsert(self, inserted)*

Восстанавливает свойства красно-черного дерева после вставки нового узла. Цикл продолжается до тех пор, пока родитель вставленного узла красный. Это обусловлено свойством красно-черного дерева, которое гласит, что красный узел не может иметь красного родителя.

Дядя – это узел, который является братом родителя вставленного узла. Если родитель вставленного узла находится справа от дедушки, то дядя –

левый потом дедушки, и обратно, если родитель вставленного узла слева от дедушки, то дядя – правый потомок дедушки.

Если родительский элемент вставленного узла является левым дочерним элементом, проверяется цвет дяди. Если дядя также красный, перекрашивается родитель и дядя в черный, дедушка – в красный, а вставленный узел перемещается на один уровень выше.

Если дядя черный, то выполняется вращение, чтобы сбалансировать дерево. Если вставленный узел является левым дочерним узлом, выполняется вращение вправо. Затем перекрашивается родитель и дедушка и выполняется поворот дедушки влево.

Если родительский элемент вставленного узла является правым дочерним элементом, выполняются аналогичные операции, описанные выше, а направления вращений меняются на противоположные.

Если после этих операций вставленный узел становится корневым, цикл прерывается.

После всего цикла гарантируем, что цвет корня – черный.

4) *leftRotate(self, pivot)*

Осуществляет левый поворот вокруг узла, который передается в качестве аргумента (*pivot),* по следующему алгоритму:

Устанавливает *child* как правый ребенок *pivot*, обновляет правого ребенка *pivot* на левого ребенка *child*. Если у *child* есть левый ребенок, обновляет родителя левого ребенка *child* на *pivot*. Если родитель *pivot* равен *None*, это означает, что *pivot* был корнем дерева, поэтому *child* становится новым корнем. Если *pivot* был левым ребенком своего родителя, *child* становится левым ребенком родителя *pivot*. В противном случае делает *child* правым ребенком *pivot*. Устанавливает левого ребенка *child* на *pivot*. И, наконец, обновляет родителя *pivot* на *child*.

5) *rightRotate(self, pivot)*

Осуществляет правый поворот вокруг узла, который передается в качестве аргумента (*pivot),* по следующему алгоритму:

Устанавливает *child* как левый ребенок *pivot*, обновляет левого ребенка *pivot* на правого ребенка *child*. Если у *child* есть правый ребенок, обновляет родителя правого ребенка *child* на *pivot*. Если родитель *pivot* равен *None*, это означает, что *pivot* был корнем дерева, поэтому *child* становится новым корнем. Если *pivot* был правым ребенком своего родителя, *child* становится правым ребенком родителя *pivot*. В противном случае делает *child* левым ребенком *pivot*. Устанавливает правого ребенка *child* на *pivot*. И, наконец, обновляет родителя *pivot* на *child*.

6) *printTree(self, root, offset)*

Печатает красно-черное дерево: если корень не пустой, то метод вызывается рекурсивно для правого поддерева с увеличенным значением отступа, а затем – для левого поддерева с тем же значением отступа.

7) *in_order(self, root, node_list)*

Обходит дерево в порядке «лево-корень-право». Если значение корня не равно нулю или у корня есть ребенок, то метод вызывается рекурсивно для левого поддерева. После того, как левое поддерево полностью обработано, метод добавляет значение корня в список. Затем метод вызывается рекурсивно для правого поддерева. Когда все узлы дерева обработаны, возвращается список, который содержит значения всех узлов в порядке обхода. В целом, для реализации вставки, метод не нужен, но его удобно использовать при тестировании, так как на выходе получаем упорядоченный список всех узлов дерева.

Пример работы с классом:

```
tree = RBTree()
tree.insertNode(23)
tree.insertNode(45)
tree.insertNode(33)
tree.printTree(tree.root, 0)
```

Создается красно-черное дерево, вставляются три элемента, затем дерево печатается.

Подробнее ознакомиться с исходным кодом и тестами структуры, сделанными при помощи библиотеки Pytest, можно в приложении А.

3. ИССЛЕДОВАНИЕ КРАСНО-ЧЕРНОГО ДЕРЕВА (ВСТАВКА)

Для исследования операции вставки были проведены тесты на разных размерах красно-черного дерева (10, 100, 1000, 10000, 100000 элементов). Замеры исследования представлены в табл.1.

Количество элементов в дереве	Время, сек
10	2,13E-06
100	2,92E-06
1000	3,26E-06
10000	4,54E-06
100000	5,15E-06

Таблица 1 – Измерения исследования

Результаты измерений представлены на рис.1. По вертикальной оси отложено время в секундах, по горизонтальной - количество элементов в красно-черном дереве.

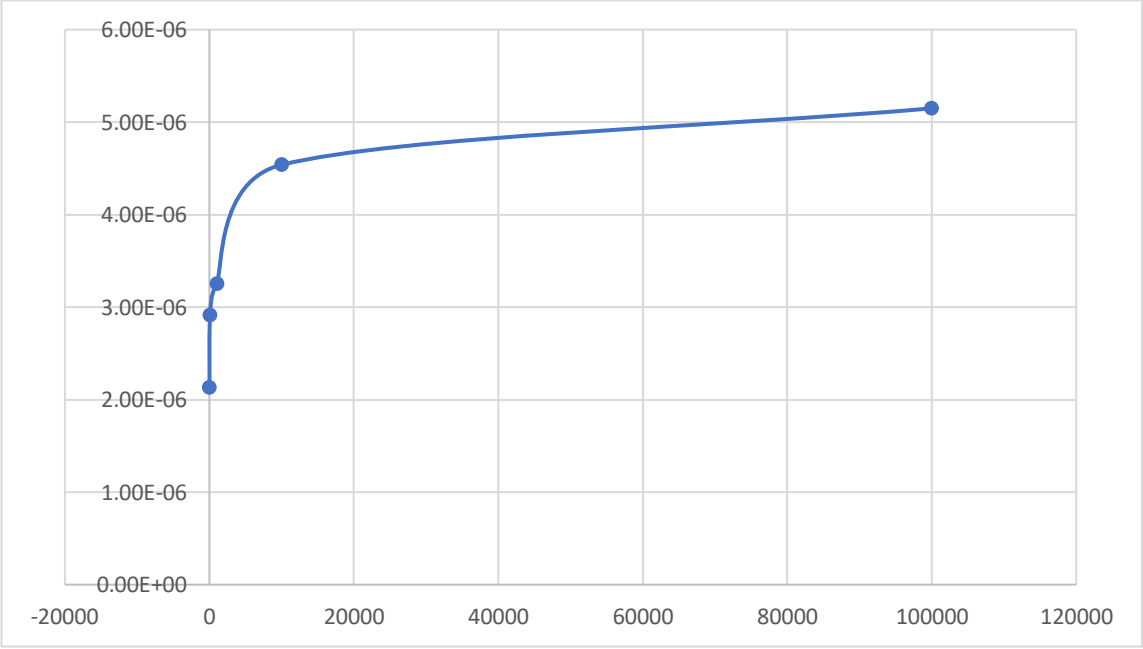

Рисунок 1 - График исследования

По графику видно, что временная сложность вставки в дерево составляет O(log n): время, необходимое для вставки элемента, увеличивается логарифмически по мере увеличения размера дерева. Это обусловлено тем, что красно-черные деревья являются сбалансированными, и в худшем случае высота дерева не превышает 2 log n. Таким образом, практические данные подтверждают теорию.

Вставка в красно-черное дерево может занять больше времени, чем вставка в AVL-дерево, из-за большей высоты красно-черного дерева. Однако, в отличие от AVL-деревьев, красно-черные не требуют хранения дополнительной информации о балансе в узлах, что делает их более экономичными с точки зрения использования памяти.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы реализована структура красно-черного дерева и операция вставки элемента.

Работоспособность структуры и операции вставки протестирована при помощи тестов.

Проведено исследование вставки, приведены и проанализированы его результаты, которые подтверждают теоретические данные.

Одним из главных преимуществ красно-черных деревьев является то, что они автоматически балансируются при вставках и удалениях. Это обеспечивает низкую высоту дерева и эффективность операции поиска, вставки и удаления.

К преимуществам еще можно отнести логарифмическую временную сложность, так как это делает красно-черные деревья очень эффективными для больших объемов данных.

К минусам красно-черных деревьев можно отнести сложность реализации: нужно управлять цветами узлов и выполнять дополнительные операции для поддержания всех свойств.

Также к минусам можно отнести потребление памяти: в отличие от некоторых других структур данных, таких как хэш-таблицы или сбалансированные бинарные деревья, красно-черные деревья могут потребовать больше памяти, так как каждый узел хранит дополнительную информацию о своем цвете и ссылках на потомков.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Красно-черное дерево [Wikipedia.org]. URL: https://en.wikipedia.org/wiki/Red–black_tree (Дата обращения: 20.12.2023)

2. Красно-черное дерево, механизм вставки. URL: https://javarush.com/groups/posts/4165-krasno-chjernoe-derevo-svoystva-principih-organizacii-mekhanizm-vstavki (Дата обращения: 20.12.2023)

3. Вставка элемента [ИТМО Вики]. URL: https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево (Дата обращения 21.12.2023)

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: RBTree.py

```
class Node():
    def __init__(self, val):
        self.val = val
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1 #new Node is red node


class RBTree():
    def __init__(self):
        self.NULL = Node(0)
        self.NULL.color = 0
        self.NULL.left = None
        self.NULL.right = None
        self.root = self.NULL

    def insertNode(self, key):
        node = Node(key)
        node.val = key
        node.parent = None
        node.left = self.NULL
        node.right = self.NULL
        node.color = 1

        curNode = self.root
        parentNode = None

        #find position for insert new node
        while curNode != self.NULL:
            parentNode = curNode
            if node.val < curNode.val:
                curNode = curNode.left
            else:
                curNode = curNode.right

        node.parent = parentNode
        if parentNode == None:
            self.root = node
        elif node.val < parentNode.val:
            parentNode.left = node
        else:
            parentNode.right = node

        if node.parent == None:
            node.color = 0
            return

        if node.parent.parent == None:
            return
```

```
        self.fixInsert(node)

    def fixInsert(self, inserted):
        while inserted.parent.color == 1:
            if inserted.parent == inserted.parent.parent.right:
                uncle = inserted.parent.parent.left
                if uncle.color == 1:
                    uncle.color = 0
                    inserted.parent.color = 0
                    inserted.parent.parent.color = 1
                    inserted = inserted.parent.parent
                else:
                    if inserted == inserted.parent.left:
                        inserted = inserted.parent
                        self.rightRotate(inserted)
                    inserted.parent.color = 0
                    inserted.parent.parent.color = 1
                    self.leftRotate(inserted.parent.parent)
            else:
                uncle = inserted.parent.parent.right
                if uncle.color == 1:
                    uncle.color = 0
                    inserted.parent.color = 0
                    inserted.parent.parent.color = 1
                    inserted = inserted.parent.parent
                else:
                    if inserted == inserted.parent.right:
                        inserted = inserted.parent
                        self.leftRotate(inserted)
                    inserted.parent.color = 0
                    inserted.parent.parent.color = 1
                    self.rightRotate(inserted.parent.parent)
            if inserted == self.root:
                break
        self.root.color = 0

    def leftRotate(self, pivot):
        child = pivot.right
        pivot.right = child.left
        if child.left != self.NULL:
            child.left.parent = pivot

        child.parent = pivot.parent
        if pivot.parent == None:
            self.root = child
        elif pivot == pivot.parent.left:
            pivot.parent.left = child
        else:
            pivot.parent.right = child
        child.left = pivot
        pivot.parent = child

    def rightRotate(self, pivot):
        child = pivot.left
        pivot.left = child.right
        if child.right != self.NULL:
```

```
                child.right.parent = pivot

            child.parent = pivot.parent
            if pivot.parent == None:
                self.root = child
            elif pivot == pivot.parent.right:
                pivot.parent.right = child
            else:
                pivot.parent.left = child
            child.right = pivot
            pivot.parent = child

        def printTree(self, root, offset):
            if root:
                self.printTree(root.right, offset + 7)
                print(' ' * offset, end='')
                print(root.val, "R" if root.color == 1 else "B")
                self.printTree(root.left, offset + 7)

        def in_order(self, root, node_list):
            if root.val != 0 or root.left or root.right:
                self.in_order(root.left, node_list)
                node_list.append(root.val)
                self.in_order(root.right, node_list)

            return node_list
```

Название файла: main.py

```
from modules import RBTree
from random import randint


if __name__ == "__main__":
    random_numbers = [randint(0, 100) for _ in range(int(input('AMOUNT
OF ELEMENTS (>2): ')))]
    tree = RBTree()
    for i in random_numbers:
        tree.insertNode(i)
    print(f'START TREE:\n{"-" * 100}')
    tree.printTree(tree.root, 0)
    num_to_insert = randint(0, 100)
    print(f'{"-" * 100}\nAFTER INSERT {num_to_insert}:\n{"-" * 100}')
    tree.insertNode(num_to_insert)
    tree.printTree(tree.root, 0)
```

Название файла: tests.py

```
from modules import RBTree
from random import randint


def test_insert_random():
    data = [randint(0, 100) for _ in range(10000)]
    tree = RBTree()
    for i in data:
        tree.insertNode(i)
```

```
    assert tree.in_order(tree.root, []) == sorted(data)


def test_insert():
    tree = RBTree()
    for i in range(10):
        tree.insertNode(i)
    assert tree.in_order(tree.root, []) == [0, 1, 2, 3, 4, 5, 6, 7, 8,
9]
    assert tree.root.val == 3
```