**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1302 |  | Новиков Г.В. |
| Преподаватель |  | Родионова Е.А. |

Санкт-Петербург

2023

Постановка задачи

**Самобалансирующиеся двоичные деревья**

Реализовать двоичное дерево поиска, красно-черное дерево и АВЛ-дерево. Сравнить длины деревьев на случайном наборе входных данных, распределенных равномерно и экспоненциально. Сравнить временные затраты на балансировку для красно-черного и АВЛ-дерева. Отчет должен содержать графики.

Теория

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим-либо потомком), называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

Оценка сложности

n – кол-во узлов

h – высота дерева

**BinSearchTree:**

|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| find | O(h), ϴ(h) |
| get\_nodes\_breadth\_first | O(n), ϴ(n) |
| get\_nodes\_inorder | O(n), ϴ(n) |
| get\_nodes\_postorder | O(n), ϴ(n) |
| get\_nodes\_preorder | O(n), ϴ(n) |
| has\_node | O(h), ϴ(h) |
| height | O(n), ϴ(n) |
| max | O(h), ϴ(h) |
| min | O(h), ϴ(h) |
| next\_el | O(h), ϴ(h) |
| prev\_el | O(h), ϴ(h) |
| print\_tree | O(n), ϴ(n) |
| insert | O(h), ϴ(h) |

**RedBlackTree:**

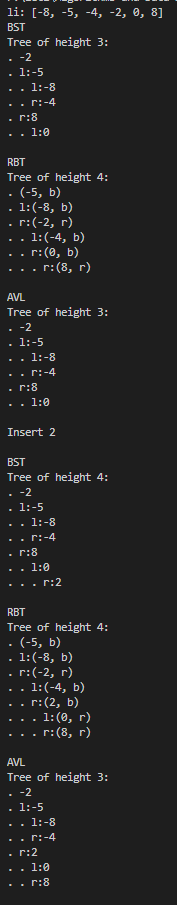
Наследует BinSearchTree

|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| left\_rotate | O(n), ϴ(h) |
| right\_rotate | O(n), ϴ(h) |
| get\_parent | O(n), ϴ(h) |
| insert | O(n), ϴ(h) |
| fixup | O(n), ϴ(h) |

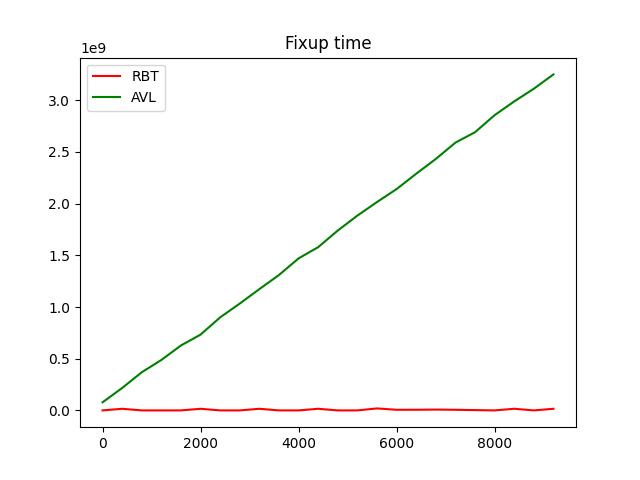
**AVLTree:**

|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| left\_rotate | O(n), ϴ(h) |
| right\_rotate | O(n), ϴ(h) |
| get\_parent | O(n), ϴ(h) |
| balance | O(n), ϴ(n) |
| insert | O(n), ϴ(n) |
| fixup | O(n), ϴ(n) |

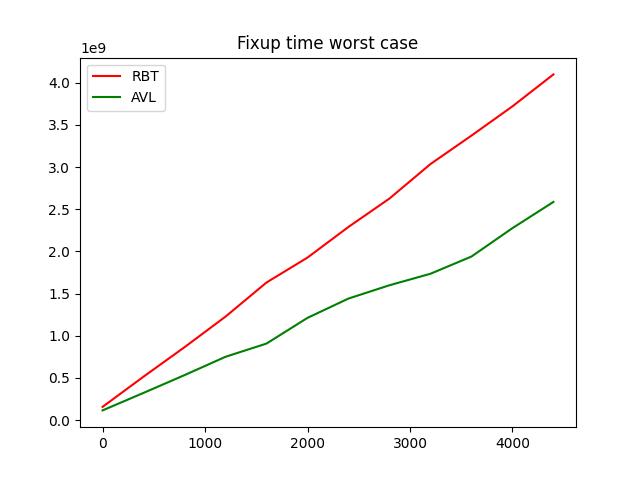
Пример работы



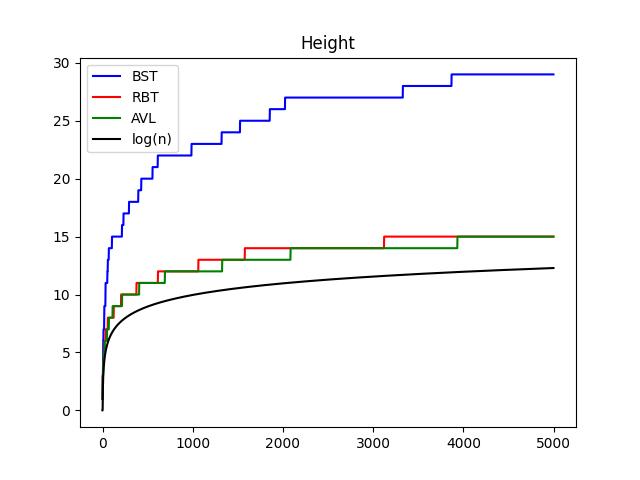
Графики



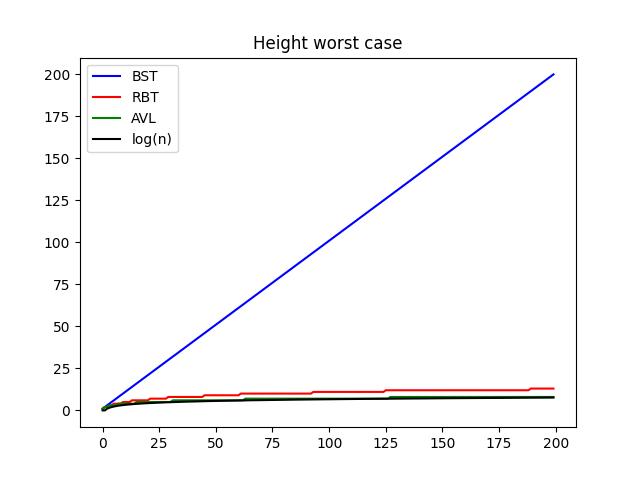
В среднем случае красно-черное дерево работает быстрее



В худшем случае асимптотика совпадает(O(n)). Красно-черное дерево немного быстрее.



Высота всех деревьев растет логарифмически, но высота двоичного дерево поиска несколько больше при всех n



В худшем случае высота двоичного дерева поиска растет линейно, высоты красно-черного и avl-дерева – логарифмически

*В худшем случае входные данные отсортированы (в данном случае по возрастанию) и вставляются всегда в самую правую ветвь.*

Листинг

**bin\_tree.py:**

from queue import Queue

from trees.node import Node

class BinTree:

def \_\_init\_\_(self):

self.root = None

# public:

def height(self, root: Node) -> int:

# O(n)

if root is None:

return 0

if root.left is None and root.right is None:

return 1

height\_l = self.height(root.left)

height\_r = self.height(root.right)

return max(height\_l, height\_r) + 1

def print\_tree(self, detail: bool = False) -> None:

# O(n)

print(f"Tree of height {self.height(self.root)}:")

self.\_\_print\_tree(self.root, detail=detail)

print()

def get\_nodes\_inorder(self) -> list:

# O(n)

return self.\_\_get\_nodes\_inorder(self.root)

def get\_nodes\_preorder(self) -> list:

# O(n)

return self.\_\_get\_nodes\_preorder(self.root)

def get\_nodes\_postorder(self) -> list:

# O(n)

return self.\_\_get\_nodes\_postorder(self.root)

def get\_nodes\_breadth\_first(self) -> list:

# O(n)

if self.root is None:

return []

nodes = []

queue = Queue()

current = self.root

queue.put(current)

while not queue.empty():

current = queue.get()

nodes.append(current)

if current.left:

queue.put(current.left)

if current.right:

queue.put(current.right)

return nodes

def insert(self, el: Node): ...

# private

def \_\_print\_tree(self, root: Node, level: int = 0, child\_letter = '', detail: bool = False) -> None:

# O(n)

if root is None:

return

child\_str = child\_letter + ':' if child\_letter != '' else ''

if detail:

print(". ", ". " \* (level - 1), child\_str, root, sep='')

else:

print(". ", ". " \* (level - 1), child\_str, root.value, sep='')

self.\_\_print\_tree(root.left, level + 1, 'l', detail)

self.\_\_print\_tree(root.right, level + 1, 'r', detail)

def \_\_get\_nodes\_inorder(self, root: Node) -> list:

# O(n)

if root is None:

return []

left = self.\_\_get\_nodes\_inorder(root.left)

right = self.\_\_get\_nodes\_inorder(root.right)

return left + [root] + right

def \_\_get\_nodes\_preorder(self, root: Node) -> list:

# O(n)

if root is None:

return []

left = self.\_\_get\_nodes\_preorder(root.left)

right = self.\_\_get\_nodes\_preorder(root.right)

return [root] + left + right

def \_\_get\_nodes\_postorder(self, root: Node) -> list:

# O(n)

if root is None:

return []

left = self.\_\_get\_nodes\_postorder(root.left)

right = self.\_\_get\_nodes\_postorder(root.right)

return left + right + [root]

**node.py:**

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

def \_\_repr\_\_(self):

return f"<Node({self.value}) at {hex(id(self))}>"

**rotation\_tree.py:**

from trees.bin\_tree import BinTree

from trees.node import Node

class RotationTree(BinTree):

def left\_rotate(self, node: Node) -> None:

# O(n)

if node.right is None:

return

right = node.right

node.right = right.left

right.left = node

parent = self.get\_parent(node) # O(n)

if parent is None:

self.root = right

elif parent.left is node:

parent.left = right

elif parent.right is node:

parent.right = right

def right\_rotate(self, node: Node) -> None:

# O(n)

if node.left is None:

return

left = node.left

node.left = left.right

left.right = node

parent = self.get\_parent(node) # O(h)

if parent is None:

self.root = left

elif parent.left is node:

parent.left = left

elif parent.right is node:

parent.right = left

def get\_parent(self, node: Node) -> Node:

# O(n)

if node is None:

return None

return self.\_\_get\_parent(node, self.root)

def \_\_get\_parent(self, node: Node, root: Node) -> Node:

# O(n)

if root == None:

return None

if root.right == node or root.left == node:

return root

if root.value < node.value:

return self.\_\_get\_parent(node, root.right)

elif root.value > node.value:

return self.\_\_get\_parent(node, root.left)

else:

left = self.\_\_get\_parent(node, root.left)

right = self.\_\_get\_parent(node, root.right)

if left:

return left

if right:

return right

**bin\_search\_tree.py:**

from trees.node import Node

from trees.bin\_tree import BinTree

class BinSearchTree(BinTree):

def \_\_init\_\_(self, elements: list):

elements.sort()

self.root = self.\_\_tree\_from\_list(elements, len(elements))

# public:

def min(self) -> Node:

# O(h)

return self.\_\_min(self.root)

def max(self) -> Node:

# O(h)

return self.\_\_max(self.root)

def find(self, value: int) -> Node:

# O(h)

return self.\_\_find(self.root, value)

def next\_el(self, node: Node) -> Node:

# O(h)

if node is None or not self.has\_node(self.root, node):

return None

if node.right is not None:

return self.\_\_min(node.right)

return self.\_\_get\_next\_in\_ancestors(self.root, node)

def prev\_el(self, node: Node) -> Node:

# O(h)

if node is None or not self.has\_node(self.root, node):

return None

if node.left is not None:

return self.\_\_max(node.left)

return self.\_\_get\_prev\_in\_ancestors(self.root, node)

def has\_node(self, root: Node, node: Node) -> bool:

# O(h)

if root is None:

return False

if node.value == root.value:

if node == root:

return True

return self.has\_node(root.left, node) or self.has\_node(root.right, node)

if node.value < root.value:

return self.has\_node(root.left, node)

if node.value > root.value:

return self.has\_node(root.right, node)

def insert(self, value: int):

node = Node(value=value)

parent = self.root

if parent is None:

self.root = node

return node

while True:

if parent.value > value:

if parent.left is not None:

parent = parent.left

else:

parent.left = node

break

else:

if parent.right is not None:

parent = parent.right

else:

parent.right = node

break

return node

# private:

def \_\_tree\_from\_list(self, elements: list, n: int) -> Node:

# elements must be sorted

# O(n)

if n == 1:

return Node(elements[0])

if n == 0:

return None

root = Node(elements[n // 2])

left = self.\_\_tree\_from\_list(elements[:n // 2], n // 2)

right = self.\_\_tree\_from\_list(elements[n // 2 + 1:], n // 2 - 1 + n % 2)

root.left = left

root.right = right

return root

def \_\_min(self, root: Node) -> Node:

# O(h)

if root is None or root.left is None:

return root

return self.\_\_min(root.left)

def \_\_max(self, root: Node) -> Node:

# O(h)

if root is None or root.right is None:

return root

return self.\_\_max(root.right)

def \_\_find(self, root: Node, value: int) -> Node:

# O(h)

if root is None:

return None

if value == root.value:

return root

if value < root.value:

return self.\_\_find(root.left, value)

if value > root.value:

return self.\_\_find(root.right, value)

def \_\_get\_next\_in\_ancestors(self, root: Node, node: Node) -> Node:

# O(h)

next\_el = None

while root and root.value != node.value:

if root.value > node.value:

next\_el = root

root = root.left

elif root.value < node.value:

root = root.right

return next\_el

def \_\_get\_prev\_in\_ancestors(self, root: Node, node: Node) -> Node:

# O(h)

next\_el = None

while root and root.value != node.value:

if root.value < node.value:

next\_el = root

root = root.right

elif root.value > node.value:

root = root.left

return next\_el

**red\_black\_node.py:**

from ..node import Node

class RedBlackNode(Node):

def \_\_init\_\_(self, value, is\_black: bool = False):

self.is\_black = is\_black

super().\_\_init\_\_(value)

def \_\_repr\_\_(self):

return f"({self.value}, {'b' if self.is\_black else 'r'})"

**red\_black\_tree.py:**

from trees.binary\_search\_tree.bin\_search\_tree import BinSearchTree

from trees.red\_black\_tree.red\_black\_node import RedBlackNode

from trees.rotation\_tree import RotationTree

class RedBlackTree(BinSearchTree, RotationTree):

def \_\_init\_\_(self, elements: list):

self.root = None

for el in elements:

self.insert(el)

def insert(self, value) -> None:

# O(n)

node = RedBlackNode(value=value, is\_black=False)

parent = self.root

if parent is None:

node.is\_black = True

self.root = node

return node

while True:

if parent.value > value:

if parent.left is not None:

parent = parent.left

else:

parent.left = node

break

else:

if parent.right is not None:

parent = parent.right

else:

parent.right = node

break

self.fixup(node) # O(n)

return node

def fixup(self, curr: RedBlackNode):

# O(n)

parent = self.get\_parent(curr) # O(n) ~Theta(h)

while parent is not None and not parent.is\_black:

if parent is None:

curr.is\_black = True

self.root = curr

return

grandparent = self.get\_parent(parent)

if grandparent is None:

parent.is\_black = True

return

if grandparent.left is parent:

uncle = grandparent.right

if grandparent.right is parent:

uncle = grandparent.left

if uncle is not None and not uncle.is\_black:

grandparent.is\_black = False

parent.is\_black = True

uncle.is\_black = True

curr = grandparent

else:

if grandparent.right is parent:

if parent.left is curr:

self.right\_rotate(parent)

curr, parent = parent, curr

self.left\_rotate(grandparent)

grandparent.is\_black = False

parent.is\_black = True

elif grandparent.left is parent:

if parent.right is curr:

self.left\_rotate(parent)

curr, parent = parent, curr

self.right\_rotate(grandparent)

grandparent.is\_black = False

parent.is\_black = True

parent = self.get\_parent(curr)

self.root.is\_black = True

**avl\_tree.py:**

from trees.node import Node

from trees.binary\_search\_tree.bin\_search\_tree import BinSearchTree

from trees.rotation\_tree import RotationTree

class AVLTree(BinSearchTree, RotationTree):

def balance(self, root: Node):

# O(n)

if root is None:

return 0

left = self.height(root.left)

right = self.height(root.right)

return left - right

def insert(self, value: int):

# O(n)

node = super().insert(value) # bst insert O(h)

self.fixup(node, self.root) # O(n)

return node

def fixup(self, node: Node, current: Node):

# O(n)

if current is None or current is node:

return

if self.has\_node(current.left, node): # O(n) ~Theta(h)

self.fixup(node, current.left)

else:

self.fixup(node, current.right)

if self.balance(current) == 2: # O(n)

if self.has\_node(current.left.left, node):

self.right\_rotate(current)

else:

self.left\_rotate(current.left)

self.right\_rotate(current)

elif self.balance(current) == -2: # O(n)

if self.has\_node(current.right.right, node):

self.left\_rotate(current)

else:

self.right\_rotate(current.right)

self.left\_rotate(current)

**main.py:**

from random import randrange

from trees.red\_black\_tree.red\_black\_tree import RedBlackTree

from trees.binary\_search\_tree.bin\_search\_tree import BinSearchTree

from trees.avl\_tree.avl\_tree import AVLTree

if \_\_name\_\_ == "\_\_main\_\_":

li = [randrange(-10, 10) for \_ in range(6)]

bst = BinSearchTree(li)

rbt = RedBlackTree(li)

avl = AVLTree(li)

print("li:", li)

detail = False

print("BST")

bst.print\_tree(detail=detail)

print("RBT")

rbt.print\_tree(detail=True)

print("AVL")

avl.print\_tree(detail=detail)

value = 2

print(f"Insert {value}\n")

bst.insert(value)

rbt.insert(value)

avl.insert(value)

print("BST")

bst.print\_tree(detail=detail)

print("RBT")

rbt.print\_tree(detail=True)

print("AVL")

avl.print\_tree(detail=detail)

**tests.py:**

import unittest

from trees.red\_black\_tree.red\_black\_tree import RedBlackTree

from trees.binary\_search\_tree.bin\_search\_tree import BinSearchTree

from trees.avl\_tree.avl\_tree import AVLTree

from random import randrange

class BSTTester(unittest.TestCase):

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.tree = None

super().\_\_init\_\_(\*args, \*\*kwargs)

def check\_tree(self, root):

if root is None:

return

self.assertTrue(root.left is None or root.left.value <= root.value)

self.assertTrue(root.right is None or root.right.value >= root.value)

self.check\_tree(root.left)

self.check\_tree(root.right)

def test\_bst\_insert(self):

self.li = [randrange(-100, 100) for \_ in range(100)]

self.tree = BinSearchTree([])

self.check\_tree(self.tree.root)

for el in self.li:

node = self.tree.insert(el)

self.check\_tree(self.tree.root)

self.assertTrue(self.tree.has\_node(self.tree.root, node))

class RBTTester(unittest.TestCase):

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.tree = None

super().\_\_init\_\_(\*args, \*\*kwargs)

def check\_tree(self, tree: RedBlackTree):

if tree.root is None:

return

self.assertTrue(tree.root.is\_black, f"Root is not black. Tree height is {self.tree.height(self.tree.root)}")

self.check\_subtree(tree.root)

def check\_subtree(self, node, b\_height: int = 1):

if node is None:

return b\_height

if not node.is\_black:

self.assertTrue(node.left is None or node.left.is\_black, f"Red node has red child. Tree height is {self.tree.height(self.tree.root)}")

self.assertTrue(node.right is None or node.right.is\_black, f"Red node has red child. Tree height is {self.tree.height(self.tree.root)}")

self.assertTrue(node.left is None or node.left.value <= node.value)

self.assertTrue(node.right is None or node.right.value >= node.value)

l\_b\_height = self.check\_subtree(node.left)

r\_b\_height = self.check\_subtree(node.right)

self.assertEqual(l\_b\_height, r\_b\_height, f"Black height is not equal in subtrees. Tree height is {self.tree.height(self.tree.root)}")

return l\_b\_height

def test\_rbt\_insert(self):

self.li = [randrange(-100, 100) for \_ in range(100)]

self.tree = RedBlackTree([])

self.check\_tree(self.tree)

for el in self.li:

node = self.tree.insert(el)

self.check\_tree(self.tree)

self.assertTrue(self.tree.has\_node(self.tree.root, node))

class AVLTester(unittest.TestCase):

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.tree = None

super().\_\_init\_\_(\*args, \*\*kwargs)

def check\_tree(self, root):

if root is None:

return

self.assertTrue(root.left is None or root.left.value <= root.value, f"Node: {root}")

self.assertTrue(root.right is None or root.right.value >= root.value, f"Node: {root}")

self.check\_tree(root.left)

self.check\_tree(root.right)

balance = self.tree.balance(root)

self.assertTrue(abs(balance) < 2, f"Balance is bigger than 1(node: {root}, {balance})")

def test\_avl\_insert(self):

self.li = [randrange(-100, 100) for \_ in range(100)]

self.tree = AVLTree([])

self.check\_tree(self.tree.root)

for el in self.li:

node = self.tree.insert(el)

self.check\_tree(self.tree.root)

self.assertTrue(self.tree.has\_node(self.tree.root, node))

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

**plots.py:**

from random import randrange

from math import log2

from time import time\_ns

from matplotlib import pyplot as plt

from trees.red\_black\_tree.red\_black\_tree import RedBlackTree

from trees.binary\_search\_tree.bin\_search\_tree import BinSearchTree

from trees.avl\_tree.avl\_tree import AVLTree

def get\_height\_distribution(tree, values: list) -> list:

height\_dist = []

for value in values:

tree.insert(value)

height\_dist.append(tree.height(tree.root))

return height\_dist

def get\_fixup\_time\_distribution(tree, values: list, sample\_frequency: int) -> list:

time\_dist = []

i = 0

st = time\_ns()

for value in values:

tree.insert(value)

if i == sample\_frequency:

time\_dist.append(time\_ns() - st)

st = time\_ns()

i = 0

i += 1

return [i \* sample\_frequency for i in range(len(time\_dist))], time\_dist

def show\_height\_plots(n: int):

x = list(range(n))

values = [randrange(-1000, 1000) for \_ in x]

# values = x # worst case - sorted list(all els go to the right)

bst = BinSearchTree([])

rbt = RedBlackTree([])

avl = AVLTree([])

bst\_height\_dist = get\_height\_distribution(bst, values)

rbt\_height\_dist = get\_height\_distribution(rbt, values)

avl\_height\_dist = get\_height\_distribution(avl, values)

plt.plot(x, bst\_height\_dist, color='b', label="BST")

plt.plot(x, rbt\_height\_dist, color='r', label="RBT")

plt.plot(x, avl\_height\_dist, color='g', label="AVL")

plt.plot(x, [0] + [log2(i) for i in x[1:]], color='black', label="log(n)")

plt.title("Height")

plt.legend()

plt.show()

def show\_fixup\_time\_plots(n):

x = list(range(n))

values = [randrange(-1000, 1000) for \_ in x]

# values = [0] \* len(x) # Worst case

rbt = RedBlackTree([])

avl = AVLTree([])

x, rbt\_time\_dist = get\_fixup\_time\_distribution(rbt, values, 400)

x, avl\_time\_dist = get\_fixup\_time\_distribution(avl, values, 400)

plt.plot(x, rbt\_time\_dist, color='r', label="RBT")

plt.plot(x, avl\_time\_dist, color='g', label="AVL")

# plt.plot(x, [0] + [log2(i) for i in x[1:]], color='black', label="log(n)")

plt.title("Fixup time")

plt.legend()

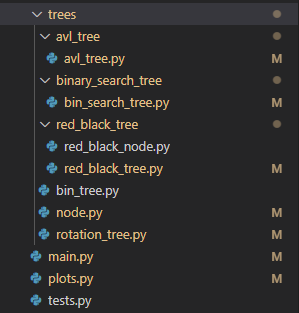
plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

show\_height\_plots(5000)

show\_fixup\_time\_plots(5000)

**Расположение файлов:**



Ссылка на код

[GitHub](https://github.com/Gregory-hub/leti/tree/master/Algorithms%20and%20Data%20Structures/sem2/lab1)