**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

отчет

**по курсовой работе**

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

**Тема: «Реализация нейронной сети для распознавания цветов»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3352 |  | Гультяев А.С. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

**Оглавление**

[**Введение 3**](#_Введение)

1. [**Теоретическая часть 4**](#_Теоретическая_часть)
   1. [Бинарное дерево поиска 4](#_Бинарное_дерево_поиска)
   2. [AVL дерево 7](#_AVL_дерево)
   3. [Красно-черное дерево 10](#_Красно-черное_дерево)
2. [**Практическая часть 15**](#_Сортировка_слиянием)
   1. [Зависимость высоты для бинарного дерева поиска 15](#_Зависимость_высоты_для)
   2. [Зависимость высоты для AVL дерева 17](#_Сортировка_выбором_1)
   3. [Зависимость высоты для красно-черного дерево](#_Сортировка_пузырьком_1) 18
   4. [Реализация обхода деревьев 20](#_Реализация_обхода_деревьев)

[**Код** **программы 23**](#_Код_программы)

[Заключение 3](#_Заключение)7

[**Ссылка на**](#_Ссылка_на_репозиторий)[**репозиторий GitHub 3**](#_Ссылка_на_репозиторий)**7**

# Введение

**Цель:** реализация сверточной нейронной сети, которая будет распознавать один из пяти видов цветков.

**Задачи:**

1. Изучить строение сетей.
2. Изучить различные виды нейронных сетей.
3. Изучить принципы подготовки изображений для подачи в сеть.
4. Изучить принципы обучения нейронных сетей для классификации изображений.
5. Рассмотреть методы улучшения сети.
6. Реализовать собственную нейронную сеть для предсказания классификации изображения.
7. Построить графики и модели эффективности обучения.

# Теоретическая часть

1.1. Базовая архитектура любой сети.

Каждая нейронная сеть (примеры различных сетей будут рассмотрены далее), имеет на начальном этапе общий вид — входной слой, какое-то количество скрытых слоев, выходной слой. Данное устройство помогает сети обрабатывать гигантское количество информации и чисел. По своему устройству нейронная сеть чем-то напоминает устройство мозга человека, а именно — наличием нейронов. Однако в отличие от человеческих нейронов, нейроны сети обрабатываются в сотни раз быстрее, поскольку компьютер может использовать параллельные вычисления для обработки большого количества данных.

Сами по себе нейроны в сети — некоторые математические функции или абстракции, которые принимают на вход данные (числовые значения), а затем обрабатывает их с использованием весов и функций активации. Пример нейрона приведен на изображении 1.

Изображение 1. Пример нейрона сети.

Тут картинка

Следующим составляющим нейронной сети будет являться слой сети. Слой сети, будь то скрытый, входной или выходной — тоже условная абстракция, которая содержит в себе как раз таки те самые нейроны. Вообще слои бывают разные видов — полносвязный, не полносвязный, сверточный, dropout, и многие многие другие. Однако большинство базовых сетей имеет обыкновенный полносвязный слой. Структура полносвязного слоя работает так, что каждый нейрон предыдущего слоя соединен с нейроном текущего, а также каждый нейрон текущего слоя соединен с нейроном следующего слоя. Пример полносвязного слоя приведен на изображении 2.

Изображение 2. Пример полносвязных слоев сети.

Тут картинка

Поскольку, как было сказано ранее, нейроны — некоторая функция, которая обрабатывается с помощью весов, стоит разобраться, что такое веса. Веса — это связи между каждым нейроном предыдущего слоя с нейроном текущего слоя. Веса имеют числовое значение и помогают работать нейроннной сети так же, как и человеческие нейроны, а именно — помогают какие-то нейронны активироваться сильнее, чем другие. Иными словами, веса служат некоторым инструментом, который помогает сети понять, какие нейроны должны быть более активны в том или ином случае. Примеры весов так же приведены на изображении 3.

Изображение 3. Пример весов нейронной сети.

Тут картинка

1.2. Виды нейронных сетей.

Всего видов нейронных сетей — бесчисленное множество. Например, для распознавания базовой задачи в обучении сетей MNIST (набор рукописых цифр от 0 до 9, размерами 28 на 28 пикселей) используется обыкновенная многослойная нейронная сеть (MLP — многослойный персептрон). Пример устройства такой сети ничем не отличается от базовой. Пример одной цифры из набора MNIST приведен на изображении 4.

Изображение 4. Пример цифры набора MNIST

Тут картинка

Регрессионная нейронная сеть — сеть используемая для сопоставления и анализа многих факторов, а также анализа прошлых значений и их зависимостей для того, что бы выдать какое-то итоговое значение. Например, такие сети используются для предсказания стоимости домов в зависимости от местности, города, положения в городе и стране, курсе валют и много другого. В свою очередь разница между работами данных сетей заключается в том, что регрессионная сеть выдает итоговое число, в то время как многослойная нейронная сеть выдает вероятность того или иного исхода. Структура такой сети зачастую проще, в силу того, что в ней содержится чуть меньше слоев, чем в многослойном персептроне. Пример работы такой сети приведен на изображении 5.

Изображение 5. Пример работы регрессионной нейронной сети.

Тут картинка

Классификационная нейронная сеть (или сверточная сеть) — сеть, которая используется для классификации изображений. Данная сеть имеет более сложную структуру чем два прошлых вида, вследствии того, что класификационной сети чаще всего подаются цветные изображения, в которых нужно распознать один из множества классов. На примере сети, которая будет строится в следующих пунктах, на вход подается изображение и сеть должна понять, что за цветок на ней изображен. Для распознавания этого сети необходимо выделять некоторые наиболее значимые детали изображение — такие как основание цветка, размеры лепестков, их наличие или отсутствие, а также многие другие факторы. Для этой задачи в классификационной сети используются определенные слои — сверточные (Convolution) и слои выделения наиболее важных параметров (MaxPool). Принципы работы каждого из этих слоев будут рассмотрены далее.

Помимо MLP, регрессионной и классификационной сетей существуют также генеративные сети, структура, принцип обучения и работы таких сетей зачастую сложнее и имеет иной подход, поэтому в данной работе такие сети не рассматриваются, поскольку они ничем не помогают в освоении материала для построения классификационной нейронной сети.

1.3. Методы улучшения работы сетей.

Для улучшения работы сети можно использовать множество вариантов. Допустим, если качество распознавания не так важно, как скорость обучения, то можно попытаться уменьшить размерность входного изображения, вследствии чего тензоры уменьшаться, и, следовательно, ускорится их машинная обработка. Однако, такой способ уменьшает качество работы сети, что не есть хорошо. В таком случае можно рассмотреть другие методы, один из них — использование батчей. Батч — некоторый пакет или набор изображений. Допустим, изначально планировалась обработка каждого изображения отдельно, в таком случае каждое изображение будет проходить через каждый слой, что займет гораздо больше времени, чем обработка некоторого количества изображений одновременно. Это особенно заметно на трехмерных изображениях для нашей задачи. Поскольку на вход планируются подаваться изображения размерами 128х128х3, то начальный размер тензор из одного изображения можем изобразить как (1, 128, 128, 3), в то время как батч из 16 изображений будет рамзерностью (16, 128, 128, 3). И вроде как ничего не поменялось, и кажется, что второй вариант будет обрабатываться в 16 раз дольше. Однако это далеко не так, поскольку существуют такие вещи, как **параллельная обработка** и **векторизация**. Когда сеть обрабатывает сразу несколько изображений (батч), большинство операций (например, матричные умножения) могут быть эффективно выполнены с использованием параллельных вычислений на графических процессорах (GPU) или других ускорителях. Это значительно сокращает время обработки по сравнению с поочередной обработкой каждого изображения. Однако, при сильно большом размере батча может забиться оперативная память, вследствии чего компьютер аварийно завершит работу и прекратится процесс обучения сети.

Также, помимо основных слоев сети, могут использоваться еще и другие виды, такие как BatchNormalization, DropOut.

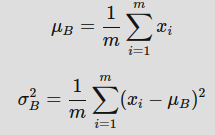
BatchNormalization — слой, который нормализует входы каждого из слоев в рамках каждого батча. То есть, для каждого слоя в сети вычисляется среднее значение и стандартное отклонение на всех входах по каждому признаку, в рамках батча. Данные значения далее используются для того, что бы нормализовать вхдоные данные слоя, то есть вычитается среднее значение и делится на стандартное отклонение. Далее, с помощью полей масштабирования (гамма) и смещения (бета), эти нормализованные данные масштабируются и сдвигаются, что бы сеть могла лучше обучаться. Пример работы показан на изображении 6.

Изображение 7. Пример работы BatchNormalization.

Тут картинка

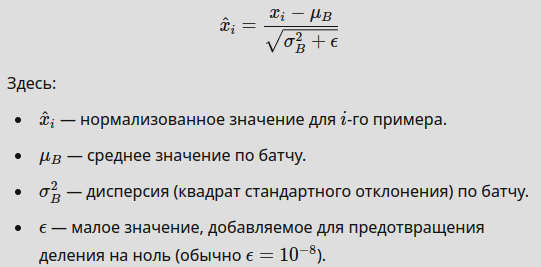
Рассмотрим поподробнее принцип работы BatchNormalization. Допустим, у нас есть набор входных данных x=[x1​,x2​,...,xm​], где m — это размер батча (количество примеров в батче), и xi​ — это входные значения для i-го примера. Тогда с**реднее значение** по батчу вычисляется по формуле в изображении 8.

Изображение 8. Формула для среднего значения и стандартного отклонения по батчу.



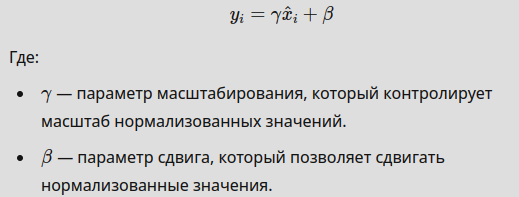
После вычисления среднего и стандартного отклонения данные нормализуются как на изображении 9.

Изображение 9. Формула для нормализации.



Для повышения гибкости нейронная сеть вводит два обучаемых параметра: **масштаб (gamma)** и **сдвиг (beta)**. После нормализации данные масштабируются и сдвигаются с использованием этих параметров, как показано на изображении 10.

Изображение 10. Масштабирование и сдвиг.



DropOut — слой, который отвечает за сброс некоторых случайных из нейронов. Это используется для того, что бы не возникало переобучения сети, при котором вся работа зависит только от малого множества. Это можно представить в виде множеств. Допустим, у нас есть некоторое множество всех нейронов слоя, тогда dropout выделит некоторое небольшое подмножество и обнулит его. Пример приведен на изображении 8.

Изображение 11. Пример DropOut на множестве.

Тут картинка

1.4. Архитектура исходной сверточной нейронной сети.

В случае нашей задачи, а именно распознавания и классификации цветов используется именно классификационная сеть (CNN).

Устройство такой сети отличается от MLP или регрессионной сетей. Как было сказано ранее, в CNN используются два дополнительных слоя — сверточный, и слой выделения наиболее важных параметров. Чаще всего эти два слоя следуют друг за другом, поскольку в комбинации они помогают ускорять обучение и сети и обращать внимание только на наиболее важные признаки. В нашем случае это может означать, что сеть просто не обращает внимание на фон изображения или какие-либо побочные объекты рядом, допустим пчел, траву.

В случае нашей сети было принято использовать структуру сети, указанную на изображении 12.

Изображение 12. Архитектура основной сети.

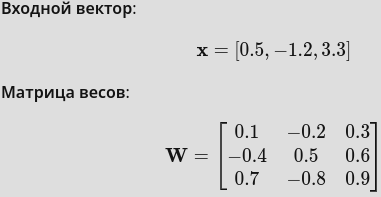
Тут картинка

1.5. Принцип работы исходной нейронной сети.

Поскольку архитектура сети уже выбрано, стоит разобрать принцип работы сети, а также принцип ее обучения.

Для начала стоит разбобраться с принципом работы каждого из слоев. Начнем с полносвязного слоя. Суть полносвязного слоя в том, что он перемножает входные значения на веса, после чего пропускает их через функцию активации. Виды функции активаций будут рассмотрены чуть позже. В общем случае формула для метода работы сети forward выглядит так: OutPut = x \* W + b, где OutPut — выходной тензор значений, x — входная матрица или скаляр, W — матрица весов, b — смещения. Однако, поскольку в нашем случае есть такой слой, как BatchNormalization, то смещения можно убрать, поскольку в этом слое они уже реализованы. Тогда итоговая формула будет иметь вид: OutPut = x \* W. Примером для работы такого слоя можно взять простейшую имитацию сети. Допустим, у нас есть некоторый входной тензор размером (1, 3). Тогда матрица весов будет иметь размерность (3, 3), поскольку имеется три входных признака и три нейрона. Изобразим данные тензоры со случайными значениями (изображение 13)

Изображение 13. Тензоры со случайными значениями.



После этого, можем применить исходную формулу и получим выходной тензор: [2.84, -3.34, 2.4]. Тогда, если представим, что данная сеть была бы сверточной, то на выходе она бы дала нам третье значение или класс. Поскольку его вероятность выше всего.

Следующим слоем для обозревания будет сверточный. Он является основной отличительной чертой классификационной сети. Суть его работы заключаетсся в том, что

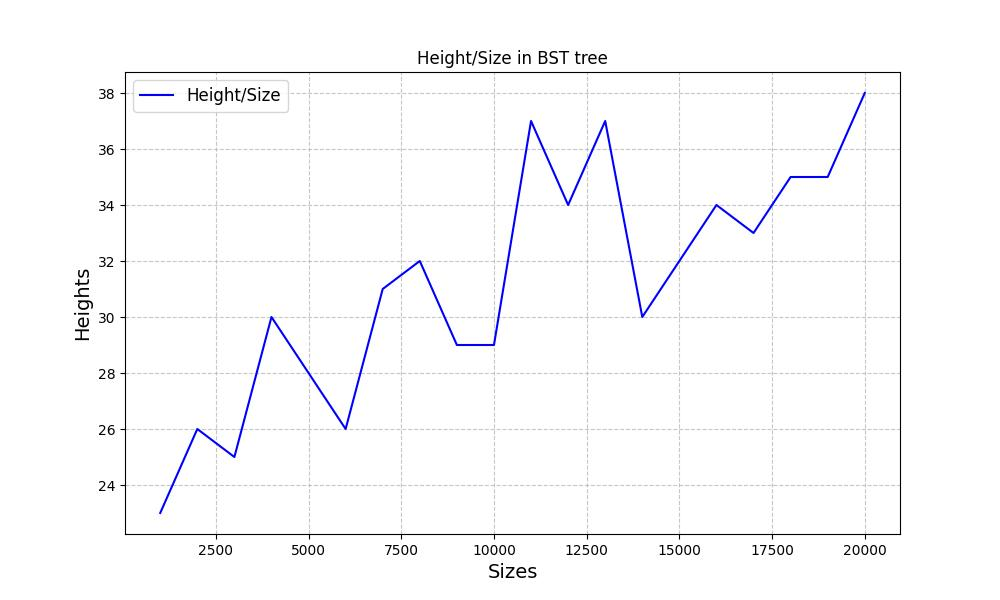
**Практическая часть.**

# Зависимость высоты для бинарного дерева поиска.

Таблица 1. Экспериментально полученные значения для зависимости высоты бинарного дерева поиска от количества ключей

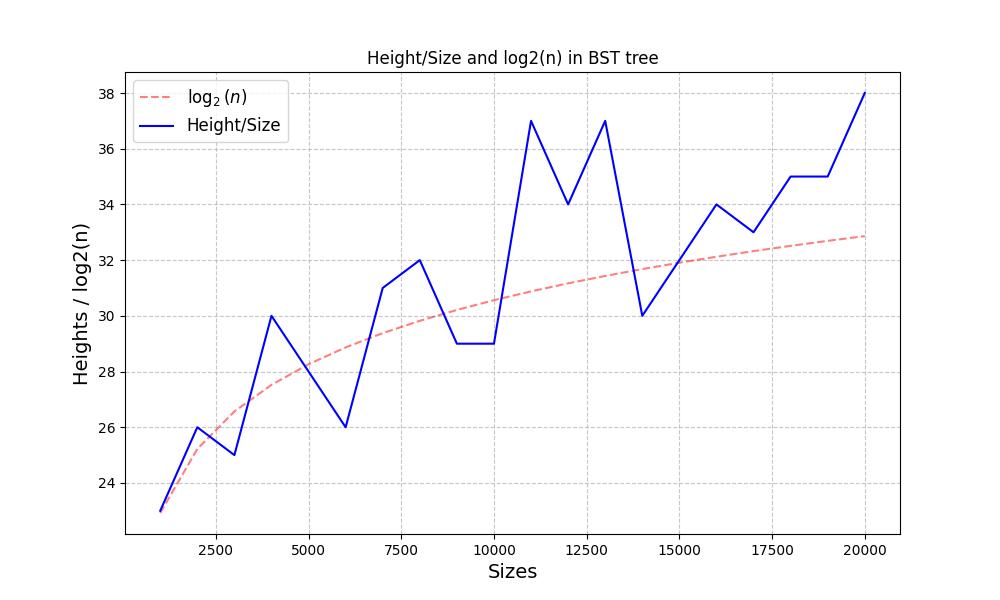
|  |  |  |
| --- | --- | --- |
| Номер | Количество ключей | Высота |
| 1 | 1000 | 23 |
| 2 | 2000 | 26 |
| 3 | 3000 | 25 |
| 4 | 4000 | 30 |
| 5 | 5000 | 28 |
| 6 | 6000 | 26 |
| 7 | 7000 | 31 |
| 8 | 8000 | 32 |
| 9 | 9000 | 29 |
| 10 | 10000 | 29 |
| 11 | 11000 | 37 |
| 12 | 12000 | 34 |
| 13 | 13000 | 37 |
| 14 | 14000 | 30 |
| 15 | 15000 | 32 |
| 16 | 16000 | 34 |
| 17 | 17000 | 33 |
| 18 | 18000 | 35 |
| 19 | 19000 | 35 |
| 20 | 20000 | 38 |

Изображение 4. График экспериментальных значений для бинарного дерева поиска.



Посмотрев на данный график, сразу тяжело будет сказать, являются ли данные значения хотя бы частично схожи с нашим теоретическим предсказанием Однако, для точности следует построить график логарифма для сравнения теории и практики. Но как было выявлено в теории, график логарифма для бинарного дерева поиска может иметь некую константу, чтобы точно совпадать с нашим графиком. В моем случае методом подбора этой самой константой было выбрано значение .

Изображение 5. График экспериментальных значений для бинарного дерева поиска с теоретическим графиком.



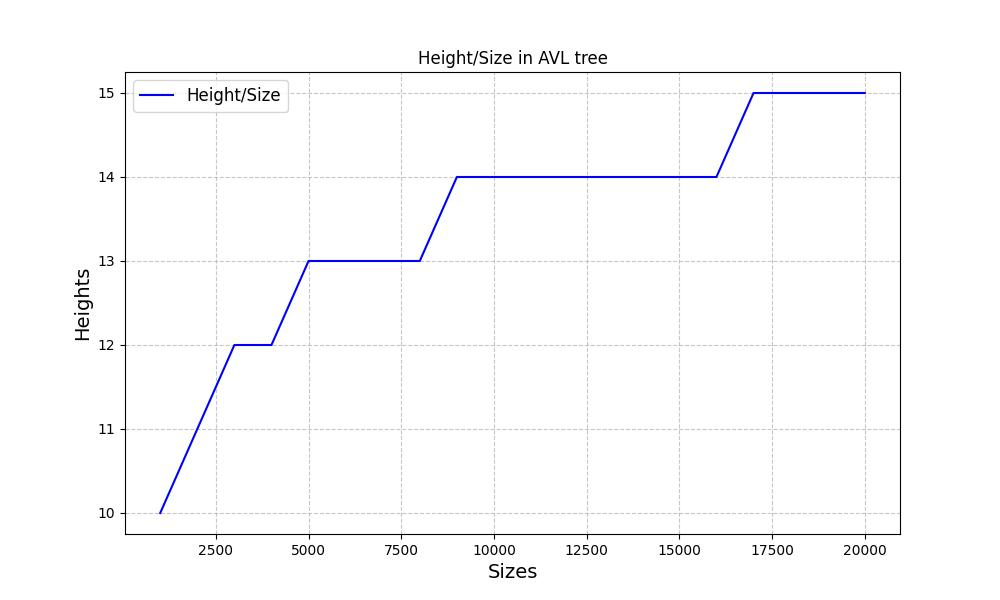
Можно заметить, что график не то, чтобы сходится с теоретическим предсказанием. Однако следует учитывать, что в теоретической части была получена асимптотическая оценка, а это значит, что на особо больших значениях, график в среднем выровняется и частично совпадет с теоретическими значениями. Скачки же в моем графике могут быть вызваны тем, что бинарное дерево поиска не является самобалансирующимся, а значит в некоторых случаях его значения могли скакать.

# Зависимость высоты для AVL дерева

Таблица 2. Экспериментально полученные значения для зависимости высоты AVL дерева от количества ключей

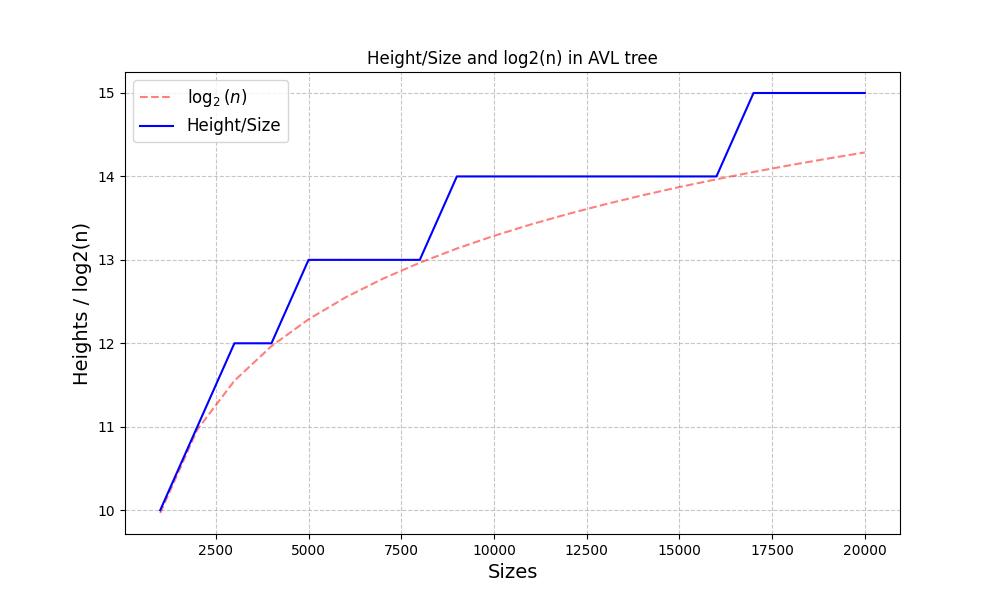
|  |  |  |
| --- | --- | --- |
| Номер | Количество ключей | Высота |
| 1 | 1000 | 10 |
| 2 | 2000 | 11 |
| 3 | 3000 | 12 |
| 4 | 4000 | 12 |
| 5 | 5000 | 13 |
| 6 | 6000 | 13 |
| 7 | 7000 | 13 |
| 8 | 8000 | 13 |
| 9 | 9000 | 14 |
| 10 | 10000 | 14 |
| 11 | 11000 | 14 |
| 12 | 12000 | 14 |
| 13 | 13000 | 14 |
| 14 | 14000 | 14 |
| 15 | 15000 | 14 |
| 16 | 16000 | 14 |
| 17 | 17000 | 15 |
| 18 | 18000 | 15 |
| 19 | 19000 | 15 |
| 20 | 20000 | 15 |

Изображение 6. График экспериментальных значений для AVL дерева.



Смотря на данный график зависимости высоты дерева от его количества элементов, можно предположительно сказать, что это похоже на теоретические значения, поскольку каждый следующий отрезок графика длиннее предыдущего, что характерно для графика логарифма, но для полной точности следует, как и в случае с бинарным деревом поиска, построить график логарифма и сравнить два графика.

Изображение 7. График экспериментальных значений для AVL дерева с теоретическим графиком.



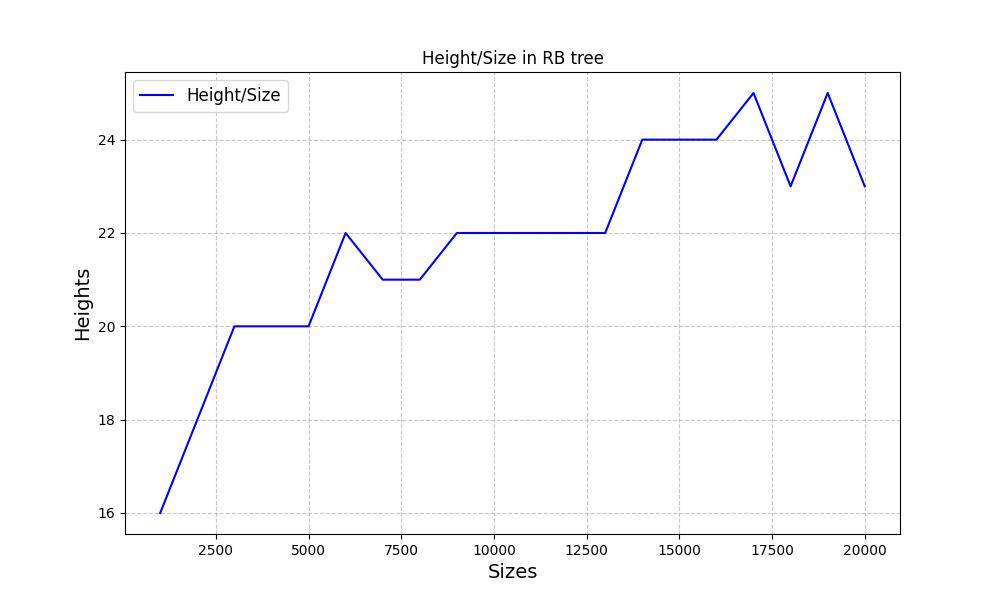
Как и было сказано ранее, график полностью совпал с ожидаемым, отсутствие скачков обусловлено свойствами AVL дерева.

# Зависимость высоты для красно-черного дерева.

Таблица 3. Экспериментально полученные значения для зависимости высоты красно-черного дерева от количества ключей

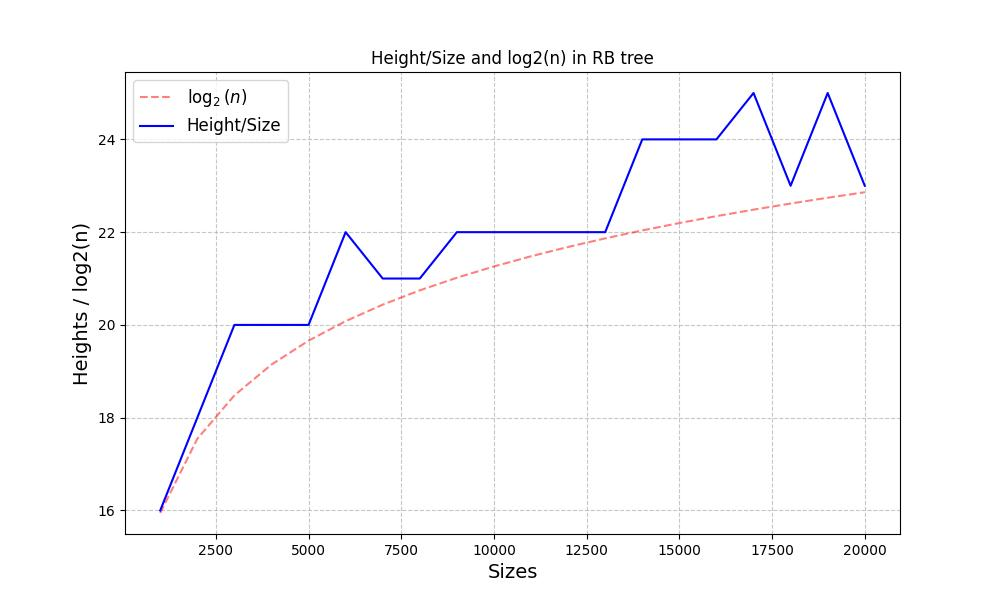
|  |  |  |
| --- | --- | --- |
| Номер | Количество ключей | Высота |
| 1 | 1000 | 16 |
| 2 | 2000 | 18 |
| 3 | 3000 | 20 |
| 4 | 4000 | 20 |
| 5 | 5000 | 20 |
| 6 | 6000 | 22 |
| 7 | 7000 | 21 |
| 8 | 8000 | 21 |
| 9 | 9000 | 22 |
| 10 | 10000 | 22 |
| 11 | 11000 | 22 |
| 12 | 12000 | 22 |
| 13 | 13000 | 22 |
| 14 | 14000 | 24 |
| 15 | 15000 | 24 |
| 16 | 16000 | 24 |
| 17 | 17000 | 25 |
| 18 | 18000 | 23 |
| 19 | 19000 | 25 |
| 20 | 20000 | 23 |

Изображение 8. График экспериментальных значений для красно-черного дерева.



Снова, просто взглянув на график, можно увидеть график логарифма, но с некоторыми несоответствиями на размерах 18000 и 20000, которые, скорее всего, обусловлены частичной балансировкой красно-черного дерева, которое подразумевает, что высота поддеревьев может отличаться в два раза, из-за чего и происходят такого рода скачки.

Изображение 9. График экспериментальных значений для красно-черного дерева с теоретическим графиком.



Как и в случае с бинарным деревом поиска, коэффициент для верхней оценки был найден элементарным перебором, и он получился равен , это было сделано, поскольку обычный график логарифма идет гораздо ниже нашего графика, однако для наглядности было принято немного приподнять график, дабы показать, что асимптотически он совпадает с предполагаемым значением.

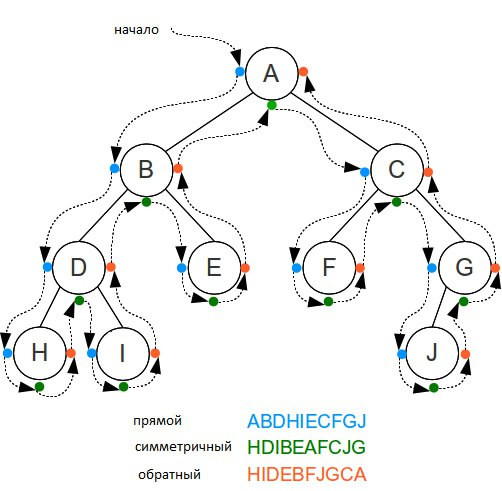
Однако подводя итоги по красно-черному дереву, можно сказать, что в целом теория вновь совпала с практикой.

# Реализация обхода деревьев

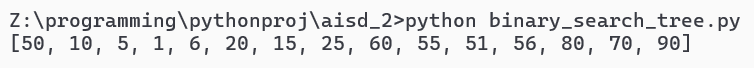
Реализация обхода была реализована только в бинарном дереве поиска, поскольку остальные виды деревьев просто наследовали данные методы.

Реализация обхода в глубину была сделана с использованием рекурсивных вызовов, начиная с корня и ступая, как бы вглубь дерева, то есть, как показано на иллюстрации ниже, мы начинаем обход с корня и идем сначала по левому поддереву, пока не достигнем края. После этого мы переходим постепенно к правой части, игнорируя элементы, которые уже были добавлены. Всего обходов в глубину есть 3 вида, pre-order, post-order, in-order. Все виды приведены на изображении 10.

Изображение 10. Иллюстрация работы обхода в глубину.



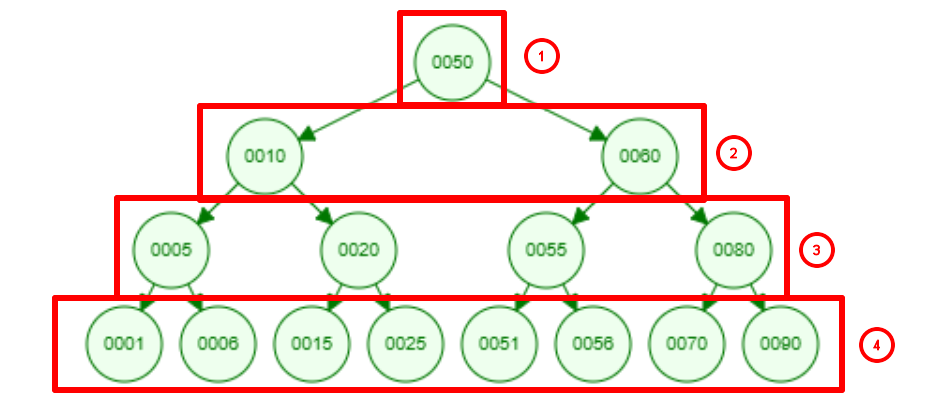
Изображение 11. Пример вывода для обхода в глубину.



Реализация обхода в ширину была реализована похожим образом, но для ее реализации использовалась структура данных deque для того, чтобы добавлять элементы не только в начало, но и в конец. Принцип работы основан на том, что мы заполняем нашу очередь начиная с корня. Далее извлекается левый элемент из очереди. Если у узла есть левый ребенок – сначала добавляется он, если есть правый – он добавляется после левого., так работает алгоритм начиная с узла и до самого низа дерева.

Как рассмотрено на иллюстрации ниже, сначала берем корень, после чего второй уровень дерева из чисел 10 и 60, далее 3 уровень из чисел 5, 20, 55, 80 и так далее до самого низа дерева.

Изображение 12. Иллюстрация работы обхода в ширину.



Изображение 13. Пример вывода для обхода в ширину.



# Код программы

binary\_search\_tree.py – файл с бинарным деревом поиска.

import os.path

import random

from collections import deque

from dop import Errors

class Node:

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

self.data = data

self.right\_kid = None

self.left\_kid = None

self.height = 1

self.color = "r"

self.parent = None

class BST:

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

self.root = Node(data)

self.size = 1

self.height = 1

def search(self, data, to\_print=False):

w\_node = self.root

if not data:

print(Errors.empty\_data())

return

while w\_node:

if data == w\_node.data:

if to\_print:

print(f"Searched: {data}")

return w\_node

w\_node = w\_node.right\_kid if w\_node.data < data else w\_node.left\_kid

print(Errors.not\_search(data))

return None

@staticmethod

def update\_height(node):

if not node:

return

stack = [node]

while stack:

current = stack[-1]

l\_height = current.left\_kid.height if current.left\_kid else 0

r\_height = current.right\_kid.height if current.right\_kid else 0

current.height = max(l\_height, r\_height) + 1

stack.pop()

if current.right\_kid:

stack.append(current.right\_kid)

if current.left\_kid:

stack.append(current.left\_kid)

def push(self, data):

w\_node = self.root

while True:

if data == w\_node.data:

print(Errors.equals\_elements(data))

return

if data > w\_node.data:

if not w\_node.right\_kid:

w\_node.right\_kid = Node(data)

break

w\_node = w\_node.right\_kid

else:

if not w\_node.left\_kid:

w\_node.left\_kid = Node(data)

break

w\_node = w\_node.left\_kid

self.size += 1

self.update\_height(self.root)

self.height = self.root.height

def remove(self, data):

node\_to\_remove = self.search(data)

if not node\_to\_remove:

return

self.root = self.\_remove(self.root, node\_to\_remove)

if self.root:

self.update\_height(self.root)

self.height = self.root.height

self.size -= 1

def \_remove(self, node, node\_to\_remove):

if not node:

return None

if node\_to\_remove.data < node.data:

node.left\_kid = self.\_remove(node.left\_kid, node\_to\_remove)

elif node\_to\_remove.data > node.data:

node.right\_kid = self.\_remove(node.right\_kid, node\_to\_remove)

else:

if not node.left\_kid and not node.right\_kid:

return None

if not node.left\_kid:

return node.right\_kid

elif not node.right\_kid:

return node.left\_kid

min\_node = self.\_find\_min(node.right\_kid)

node.data = min\_node.data

node.right\_kid = self.\_remove(node.right\_kid, min\_node)

self.update\_height(node)

return node

@staticmethod

def \_find\_min(node):

current = node

while current.left\_kid:

current = current.left\_kid

return current

def print(self):

def print\_in\_order(node, level=0, prefix="Root: "):

if node is not None:

print(" " \* (level \* 4) + prefix + str(node.data))

if node.left\_kid or node.right\_kid:

if node.left\_kid:

print\_in\_order(node.left\_kid, level + 1, "L--- ")

else:

print(" " \* ((level + 1) \* 4) + "L--- None")

if node.right\_kid:

print\_in\_order(node.right\_kid, level + 1, "R--- ")

else:

print(" " \* ((level + 1) \* 4) + "R--- None")

print\_in\_order(self.root)

def dfs(self):  
 def \_dfs\_in\_order(node):  
 if node:  
 \_dfs\_in\_order(node.left\_kid)  
 result.append(node.data)  
 \_dfs\_in\_order(node.right\_kid)  
  
 def \_dfs\_pre\_order(node):  
 if node:  
 result.append(node.data)  
 \_dfs\_pre\_order(node.left\_kid)  
 \_dfs\_pre\_order(node.right\_kid)  
  
 def \_dfs\_post\_order(node):  
 if node:  
 \_dfs\_post\_order(node.left\_kid)  
 \_dfs\_post\_order(node.right\_kid)  
 result.append(node.data)  
  
 result = []  
 \_dfs\_pre\_order(self.root)  
 return result  
  
def dfs\_in\_order(self):  
 result = []  
 def \_dfs\_in\_order(node):  
 if node:  
 \_dfs\_in\_order(node.left\_kid)  
 result.append(node.data)  
 \_dfs\_in\_order(node.right\_kid)  
  
 \_dfs\_in\_order(self.root)  
 return result  
  
def dfs\_post\_order(self):  
 result = []  
 def \_dfs\_post\_order(node):  
 if node:  
 \_dfs\_post\_order(node.left\_kid)  
 \_dfs\_post\_order(node.right\_kid)  
 result.append(node.data)  
  
 \_dfs\_post\_order(self.root)  
 return result  
  
def bfs(self):  
 result = []  
 if not self.root:  
 return result  
  
 queue = deque([self.root])  
  
 while queue:  
 node = queue.popleft()  
 result.append(node.data)  
  
 if node.left\_kid:  
 queue.append(node.left\_kid)  
 if node.right\_kid:  
 queue.append(node.right\_kid)  
  
 return result

def get\_dependence(rand, file, model, maximum=20001, stride=1000):

def \_generate\_arr(maxi):

if rand:

arr = random.sample(range(0, maxi), maxi)

else:

arr = list(range(maxi))

return arr

print(f"Random: {rand}, File: {file}, Model: {model.\_\_name\_\_}")

with open(os.path.join(r"TXT/", file), "w") as f:

for i in range(stride, maximum, stride):

arr = \_generate\_arr(i)

tree = model(arr[0])

for j in range(1, i):

tree.push(arr[j])

f.write(f"Size: {i}\tHeight: {tree.height}\n")

print(f"Size: {i}\tHeight: {tree.height}")

AVL\_tree.py – файл с AVL деревом:

from dop import Errors

from binary\_search\_tree import BST, Node, get\_dependence

class AVLT(BST):

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

super().\_\_init\_\_(data)

@staticmethod

def get\_height(node):

return node.height if node else 0

def get\_balance(self, node):

return self.get\_height(node.left\_kid) - self.get\_height(node.right\_kid) if node else 0

def r\_rotate(self, y):

if not y or not y.left\_kid:

return y

x = y.left\_kid

T = x.right\_kid

x.right\_kid = y

y.left\_kid = T

y.height = 1 + max(self.get\_height(y.left\_kid), self.get\_height(y.right\_kid))

x.height = 1 + max(self.get\_height(x.left\_kid), self.get\_height(x.right\_kid))

return x

def l\_rotate(self, x):

if not x or not x.right\_kid:

return x

y = x.right\_kid

T = y.left\_kid

y.left\_kid = x

x.right\_kid = T

x.height = 1 + max(self.get\_height(x.left\_kid), self.get\_height(x.right\_kid))

y.height = 1 + max(self.get\_height(y.left\_kid), self.get\_height(y.right\_kid))

return y

def balancing(self, node):

if not node:

return None

balance = self.get\_balance(node)

if balance > 1:

if self.get\_balance(node.left\_kid) < 0:

node.left\_kid = self.l\_rotate(node.left\_kid)

return self.r\_rotate(node)

if balance < -1:

if self.get\_balance(node.right\_kid) > 0:

node.right\_kid = self.r\_rotate(node.right\_kid)

return self.l\_rotate(node)

return node

def push(self, data):

def \_insert(node, data):

if not node:

return Node(data)

if node.data == data:

print(Errors.equals\_elements(data))

return node

if data < node.data:

node.left\_kid = \_insert(node.left\_kid, data)

elif data > node.data:

node.right\_kid = \_insert(node.right\_kid, data)

node.height = 1 + max(self.get\_height(node.left\_kid), self.get\_height(node.right\_kid))

return self.balancing(node)

self.root = \_insert(self.root, data)

self.size += 1

self.update\_height(self.root)

self.height = self.root.height

def \_remove(self, node, node\_to\_remove):

if not node:

return None

if node\_to\_remove.data < node.data:

node.left\_kid = self.\_remove(node.left\_kid, node\_to\_remove)

elif node\_to\_remove.data > node.data:

node.right\_kid = self.\_remove(node.right\_kid, node\_to\_remove)

else:

if not node.left\_kid and not node.right\_kid:

return None

elif not node.left\_kid:

return node.right\_kid

elif not node.right\_kid:

return node.left\_kid

min\_node = self.\_find\_min(node.right\_kid)

node.data = min\_node.data

node.right\_kid = self.\_remove(node.right\_kid, min\_node)

node.height = 1 + max(self.get\_height(node.left\_kid), self.get\_height(node.right\_kid))

return self.balancing(node)

def print(self):

def print\_in\_order(node, level=0, prefix="Root: "):

if node is not None:

print(" " \* (level \* 4) + prefix + str(node.data) + f" (H: {node.height})")

if node.left\_kid or node.right\_kid:

if node.left\_kid:

print\_in\_order(node.left\_kid, level + 1, "L--- ")

else:

print(" " \* ((level + 1) \* 4) + "L--- None")

if node.right\_kid:

print\_in\_order(node.right\_kid, level + 1, "R--- ")

else:

print(" " \* ((level + 1) \* 4) + "R--- None")

print\_in\_order(self.root)

def get\_tree\_balance(self):

def check\_balance(node):

if node:

balance = self.get\_balance(node)

print(f"Node {node.data} Balance: {balance}")

check\_balance(node.left\_kid)

check\_balance(node.right\_kid)

check\_balance(self.root)

RB\_tree.py – файл с красно-черным деревом:

from binary\_search\_tree import Node, BST, get\_dependence

from dop import Errors

class RBT(BST):

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

super().\_\_init\_\_(data)

self.root.color = "b"

def rotate\_left(self, node):

right\_kid = node.right\_kid

node.right\_kid = right\_kid.left\_kid

if right\_kid.left\_kid:

right\_kid.left\_kid.parent = node

right\_kid.parent = node.parent

if node.parent is None:

self.root = right\_kid

elif node == node.parent.left\_kid:

node.parent.left\_kid = right\_kid

else:

node.parent.right\_kid = right\_kid

right\_kid.left\_kid = node

node.parent = right\_kid

def rotate\_right(self, node):

left\_kid = node.left\_kid

node.left\_kid = left\_kid.right\_kid

if left\_kid.right\_kid:

left\_kid.right\_kid.parent = node

left\_kid.parent = node.parent

if node.parent is None:

self.root = left\_kid

elif node == node.parent.right\_kid:

node.parent.right\_kid = left\_kid

else:

node.parent.left\_kid = left\_kid

left\_kid.right\_kid = node

node.parent = left\_kid

def insert\_fix(self, node):

while node.parent and node.parent.color == "r":

if node.parent == node.parent.parent.left\_kid:

uncle = node.parent.parent.right\_kid

if uncle and uncle.color == "r":

node.parent.color = "b"

uncle.color = "b"

node.parent.parent.color = "r"

node = node.parent.parent

else:

if node == node.parent.right\_kid:

node = node.parent

self.rotate\_left(node)

node.parent.color = "b"

node.parent.parent.color = "r"

self.rotate\_right(node.parent.parent)

else:

uncle = node.parent.parent.left\_kid

if uncle and uncle.color == "r":

node.parent.color = "b"

uncle.color = "b"

node.parent.parent.color = "r"

node = node.parent.parent

else:

if node == node.parent.left\_kid:

node = node.parent

self.rotate\_right(node)

node.parent.color = "b"

node.parent.parent.color = "r"

self.rotate\_left(node.parent.parent)

self.root.color = "b"

def push(self, data):

w\_node = Node(data)

w\_node.color = "r"

parent = None

current = self.root

while current:

parent = current

if data < current.data:

current = current.left\_kid

elif data > current.data:

current = current.right\_kid

else:

print(Errors.equals\_elements(data))

return

w\_node.parent = parent

if parent is None:

parent = self.root

elif parent.data > data:

parent.left\_kid = w\_node

else:

parent.right\_kid = w\_node

self.size += 1

self.insert\_fix(w\_node)

self.update\_height(self.root)

self.height = self.root.height

def print(self):

def print\_in\_order(node, level=0, prefix="Root: "):

if node is not None:

print(" " \* (level \* 4) + prefix + str(node.data) + f" (С: {node.color})")

if node.left\_kid or node.right\_kid:

if node.left\_kid:

print\_in\_order(node.left\_kid, level + 1, "L--- ")

else:

print(" " \* ((level + 1) \* 4) + "L--- None")

if node.right\_kid:

print\_in\_order(node.right\_kid, level + 1, "R--- ")

else:

print(" " \* ((level + 1) \* 4) + "R--- None")

print\_in\_order(self.root)

def remove(self, data):

node = self.search(data)

if node is None:

return

if node.left\_kid and node.right\_kid:

successor = self.\_find\_min(node.right\_kid)

node.data = successor.data

node = successor

replacement = node.left\_kid if node.left\_kid else node.right\_kid

if replacement:

replacement.parent = node.parent

if node.parent is None:

self.root = replacement

elif node == node.parent.left\_kid:

node.parent.left\_kid = replacement

else:

node.parent.right\_kid = replacement

if node.color == "b":

self.\_remove\_fix(replacement)

elif node.parent is None:

self.root = None

else:

if node.color == "b":

self.\_remove\_fix(node)

if node.parent:

if node == node.parent.left\_kid:

node.parent.left\_kid = None

else:

node.parent.right\_kid = None

self.size -= 1

def \_remove\_fix(self, node):

if node is None:

return

while node != self.root and node.color == "b":

if node == node.parent.left\_kid:

sibling = node.parent.right\_kid

if sibling.color == "r":

sibling.color = "b"

node.parent.color = "r"

self.rotate\_left(node.parent)

sibling = node.parent.right\_kid

if (sibling.left\_kid is None or sibling.left\_kid.color == "b") and \

(sibling.right\_kid is None or sibling.right\_kid.color == "b"):

sibling.color = "r"

node = node.parent

else:

if sibling.right\_kid is None or sibling.right\_kid.color == "b":

if sibling.left\_kid:

sibling.left\_kid.color = "b"

sibling.color = "r"

self.rotate\_right(sibling)

sibling = node.parent.right\_kid

sibling.color = node.parent.color

node.parent.color = "b"

if sibling.right\_kid:

sibling.right\_kid.color = "b"

self.rotate\_left(node.parent)

node = self.root

else:

sibling = node.parent.left\_kid

if sibling.color == "r":

sibling.color = "b"

node.parent.color = "r"

self.rotate\_right(node.parent)

sibling = node.parent.left\_kid

if (sibling.left\_kid is None or sibling.left\_kid.color == "b") and \

(sibling.right\_kid is None or sibling.right\_kid.color == "b"):

sibling.color = "r"

node = node.parent

else:

if sibling.left\_kid is None or sibling.left\_kid.color == "b":

if sibling.right\_kid:

sibling.right\_kid.color = "b"

sibling.color = "r"

self.rotate\_left(sibling)

sibling = node.parent.left\_kid

sibling.color = node.parent.color

node.parent.color = "b"

if sibling.left\_kid:

sibling.left\_kid.color = "b"

self.rotate\_right(node.parent)

node = self.root

if node:

node.color = "b"

dop.py – файл с построением графиков и ошибками:

import matplotlib.pyplot as plt

import numpy as np

import os

class Color:

Reset = '\033[0m'

Red = '\033[91m'

Green = '\033[92m'

Yellow = '\033[93m'

Blue = '\033[94m'

class Errors:

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

pass

@staticmethod

def empty\_data():

return f"{Color.Red}Error: Cannot search for empty data in the tree!{Color.Reset}"

@staticmethod

def not\_search(data):

return f"{Color.Red}Error: Data not found in the tree! Data: {data} is not in the tree!{Color.Reset}"

@staticmethod

def equals\_elements(data):

return f"{Color.Red}Error: Duplicate element detected! Data: {data} cannot be added to the tree!{Color.Reset}"

def plot\_graphs(name\_out, name\_in):

sizes = []

heights = []

with open(name\_in, "r") as f:

for line in f:

line = line.strip()

if not line:

continue

parts = line.split("\t")

if len(parts) == 2:

size\_str, height\_str = parts

size = int(size\_str.split(":")[1].strip())

height = int(height\_str.split(":")[1].strip())

sizes.append(size)

heights.append(height)

print(name\_out)

sizes = np.array(sizes)

heights = np.array(heights)

# Рассчитываем логарифмическую функцию log2(n)

log2\_values = np.log2(sizes)

# Создаём директорию "Graphs", если она не существует

if not os.path.exists("Graphs"):

os.makedirs("Graphs")

for i in range(2):

# Строим график

plt.figure(figsize=(10, 6))

if i:

if name\_out == "BST":

plt.plot(sizes, 2.3 \* log2\_values, linestyle="--", color="r", label=r"$\log\_2(n)$",

alpha=0.5)

elif name\_out == "RB":

plt.plot(sizes, 1.6 \* log2\_values, linestyle="--", color="r", label=r"$\log\_2(n)$",

alpha=0.5)

else:

plt.plot(sizes, log2\_values, linestyle="--", color="r", label=r"$\log\_2(n)$",

alpha=0.5)

# Основной график

plt.plot(sizes, heights, marker="", linestyle="-", color="b", label="Height/Size")

if i:

plt.title(f"Height/Size and log2(n) in {name\_out} tree")

plt.ylabel("Heights / log2(n)", fontsize=14)

else:

plt.title(f"Height/Size in {name\_out} tree")

plt.ylabel("Heights", fontsize=14)

plt.xlabel("Sizes", fontsize=14)

plt.grid(True, linestyle="--", alpha=0.7)

plt.legend(fontsize=12)

# Сохраняем график в файл

plt.savefig(os.path.join("Graphs", name\_out + f"\_{i}.jpg"))

plt.close()

for filename in os.listdir("TXT"):

full\_path = os.path.join("TXT", filename)

plot\_graphs(filename.replace(".txt", ""), full\_path)

# Заключение

В ходе выполнения лабораторной работы были изучены различные виды деревьев. Их преимущества и недостатки. Был реализован обход дерева вглубь и вширь, произведен анализ зависимости высоты дерева от количества ключей в нем. Были изучены основные операции дерева, такие как поиск, удаление, вставка и вывод. В результате работы была продемонстрирована эффективность каждого из деревьев для своей задачи.

# Ссылка на репозиторий GitHub

<https://github.com/ZamniProg/aisd_lab_2>