**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.

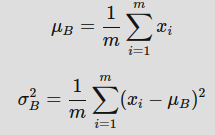
BatchNormalization — слой, который нормализует входы каждого из слоев в рамках каждого батча. То есть, для каждого слоя в сети вычисляется среднее значение и стандартное отклонение на всех входах по каждому признаку, в рамках батча. Данные значения далее используются для того, что бы нормализовать вхдоные 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

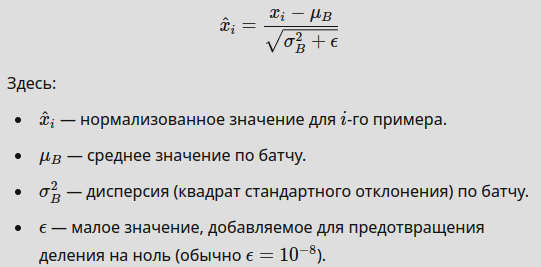
Рассмотрим поподробнее принцип работы BatchNormalization. Допустим, у нас есть набор входных данных x=[x1​,x2​,...,xm​], где m — это размер батча (количество примеров в батче), и xi​ — это входные значения для i-го примера. Тогда с**реднее значение** по батчу вычисляется по формуле в изображении 8.

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



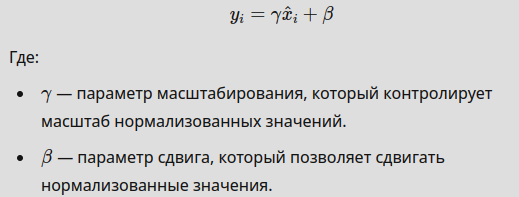
После вычисления среднего и стандартного отклонения данные нормализуются как на изображении 9.

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.

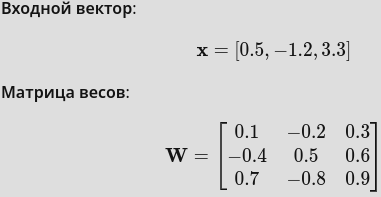
Тут картинка

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 заключается в том, что он принимает на вход некоторый тензор, после чего создает новый, путем матричного умножения, а именно, тензор фильтров доманожается на подтензоры входного тензора. Что бы было чуть понятнее, рассмотрим пример. Допустим, у нас имеется некоторый входной тензор размерностью (10, 10, 1). Тогда, нужно определиться с размером фильтров. Размер фильтра стоит выбирать опираясь на данную формулу: out = ((s — size) / stride) + 1, где out — выходная высота или ширина тензора, s — высота или ширина входного тензора, size — размер фильтра (его высота или ширина), stride — шаг, на который будет смещаться подтензор. Таким образом, если у нас имеется тот самый тензор размерами (10, 10, 1), где 10 — высота и ширина, а 1 — количество каналов, то попробуем использовать для этого фильтр размером (3, 3, 32), где 3 — высота и ширина фильтра, 32 — количество условных частиц, по которым фильтр будет обрабатывать и выделять некие черты изображения. Возьмем некоторый малый шаг, что бы не сильно уменьшить изображение, допустим 1. Тогда, итоговый размер выходного тензора будет: size = (10 — 3) / 1 + 1 = 8, получается, что итоговый размер (8, 8, 32). Для основных вычислений используется формула: Ti = Σ(ri\*ci), где Ti — это элемент итоговой подматрицы, ri — элемент по строке в подматрице, ci — элемент по колоне в фильтре. Пример того, как происходят вычисления приведен на изображении 14.

Изображение 14. Пример вычисления в CL.

Тут картинка

Еще один слой, который используется в данной сети — MaxPool. Суть данного слоя, как было написано ранее, заключается в том, что бы выделять наиболее значимые участки изображения, избегая ненужные. Тогда, рассмотрим пример работы такого слоя на изображении 15.

Изображение 15. Пример работы MaxPool слоя.

Тут картинка

Из изображения видно, что суть работы заключается в том, что у нас, как и в случае с CL имеется некоторая условная матрица, по которой будет идти работа. То есть, точно так же мы выбираем некоторый регион из входного изображения, при этом размер этой матрицы так же будет: region = (size — pool) / stride + 1, где region — размерность итогового региона, size — входной размер, pool — некоторый параметр (высота или ширина), stride — шаг, на который будет смещаться регион.

Двумя другими слоями будут BatchNormalization и dropout. По поводу них все было сказано и разобрано ранее.

После того, как стала понятна работа каждого слоя отдельно, стоит разобраться с таким параметром, как функция активации.

Функции активации — некоторые функции, которые принимают все тот же входной тензор, при этом обрабатывая его и приводя к более удобному для обучения и работы виду. Функций активации есть огромное множество, в данной работе использовались такие функции как softmax и ReLU. Softmax — функция активации, которая принимает значение и возвращает другое значение в пределах от 0 до 1, при этом всем, сумма всех нейронов (входных значений) будет равна 1. Данная функция чаще всего используется для выходных слоев в сверточных сетях, поскольку, так как сумма всех нейронов равна 1, то каждое из значений будет иметь некоторое процентное отношение. В таком случае возвращается вероятность в удобном виде, и сеть выдает то значение, которое имеет наибольшую вероятность. Формула данной функции, а также ее график приведены ниже, на изображении 16.

Изображение 16. Функция softmax.

Тут картинка

Другая функция активации — ReLU, используется для внутренних, скрытых слоев. Данная функция удобна тем, что если значение меньше 0, то функция возвращает ноль, если же значение больше, то функция выдает это же значение. Получается, что преимущество данной функции в том, что она обнуляет ненужные нейроны, оставляя только значимые, почти так же, как и dropout слой. Данная функция описывается такой формулой: F(x) = max(0, x). Так же график такой функции приведен на изображении 17.

Изображение 17. Функция ReLU.

Тут картинка

Резумируя все выше сказанное, картинка на входе превращается в тензор значений, после чего проходит в слой свертки, откуда выделяются наиболее важные параметры изображения, допустим лепестки. После этого, в MaxPool слое так же выделяются только наиболее полезные участки изображения. После каждого из слоев используется функция активации, что бы придать некоторую нелинейность сети. Затем, тензор переходит в плоносвязный слой, где перемножается на веса связи между слоями, после чего это повторяется n раз. И в конце сеть получает 5 чисел, обрабатывает их функцией softmax и имеет на выходе 5 значений, где каждое из полученных значений имеет вероятность от 0 до 99.(9) значений. И выбирается наибольшее из них, после чего выводится результат предсказания сети.

1.6. Принцип обучения сети.

Поскольку принцип работы самого по себе распознавания того, что изображено на картинке уже разобран, стоит понять, как именно обучается сеть. Для этого необходимо разобрать такие понятие: градиент, градиентный спуск, функция потерь, обратное распространение ошибки, скорость обучения.

Стоит начать с функции потерь, поскольку в алгоритме обучения первая идет она. Функций потерь, как и в случае с функциями активации, бесчисленное множество, где каждая подходит для своей задачи. В случае решения задачи распознавания изображения чаще всего используется функция кросс-энтропии. Формула такой функции выглядит как-то так: L = (x — y)2, где L — итоговая потеря, x — предсказания сети, y — реальные значения. Данная функция хороша тем, что она простая, удобная и понятная всем. Однако, данную функцию можно улучшить, просто используя принцип one hot, по которому вместо дробных чисел от 0.0 до 1.0 используется матрица, состоящая только из 0 и 1, где 1 — это реальное значение, а 0 — нет. Сама функция выглядит так: L = x — y, где L — потеря, x — предсказания сети, y — реальные значения. Однако x и y матрицы, поэтому и происходит матричная разность. Пример этой функции приведен на изображении 18.

Изображение 18. Пример работы функции потерь.

Тут картинка

После того, как вычисленна функция потерь, она передается обратно в сеть, начиная с конца и двигаясь к входному слою. Такой метод называется back propagation или обратное распространение. Для этого вычисляются градиенты весов в полносвязном слое, градиенты гаммы и беты в BatchNormalization слое и градиенты фильтров в сверточном слое.

Сами по себе, с математической точки зрения, градиенты высчитываются так: пусть f(x) = 7x2+3y, тогда ее градиентом grad(f(x))=(df/dx, df/dy) в некоторой точке M(x, y). Тогда для нашего примера, градиентом будет: grad(f(x)) = (14x, 3). Подставляя некоторую точку M(1, 0) получаем, что градиент в данной точке будет равен (14, 3). Градиент показывает путь, по которому функции растет с наибольшей скоростью.

Таким образом, градиент находит максимумы функции, однако, поскольку в сетях необходимо уменьшать ошибку, что бы точность росла, необходимо двигаться в другую сторону от градиента, что бы достичь минимума функции. Пример того, как выглядит градиентный спуск для заданной функции представлен на изображении 19.

Изображение 19. Пример градиентного спуска.

Тут картинка

Однако, в контексте машинного обучения используется немного иной подход к подсчету градиентов. В случае с нейронной сетью используется подход, при котором происходит матричное множение транспонированных входных данных, на градиент прошлого слоя. То есть формулу можно записать так: grad(W) = xT \* grad(O), где xT — транспонированная матрица с исходными значениями, grad(O) — некоторый градиент предыдущего слоя. То же самое происходит и с входными данными для того, что бы передать их следующему слою: grad(I) = grad(O) \* WT, где I — исходные данные, grad(O) — градиент предудщего слоя, WT — транспонированная матрица весов. Такие действия происходят в каждом из слоев сети для того, что бы уменьшать ошибку и корректировать веса связей между слоями.

Корректировка весов происходит с помощью нахождения градиентов и использованием такого параметра, как скорость обучения. Скорость обучения — некоторый небольшой параметр, который подсказывает сети, с какой скоростью ей изменять веса. В контексте математики — это то, на какое расстояние от начальной точки мы изменяем положение в пространстве. Очень важно выбрать средний параметр для этого параметра, поскольку сильно большой параметр может мешать достичь минимума из-за того, что постоянно перепрыгивает его. Сильно малый же параметр сильно замедляет обучение сети, а так же имеет возможность застрять в локальном минимуме и не достичь минимального значения всей функции. Пример этого приведен на изображении 20.

Изображение 20. Пример корректировки скорости обучения.

Тут картинка

Сама же по себе корректировка весов для обучения происходит путем умножения градиента весов на этот самый параметр скорости обучения, то есть итоговая формула имеет вид: W = r \* grad(W), где W — веса слоя, r — скорость обучения, grad(W) — градиенты весов.

Данное строение помогает сети корректировать связи между всеми слоями сети, используя итоговую потерю и двигаясь к началу сети корректируя веса.

**Практическая часть.**

2.1. Предобработка изображений.

Поскольку компьютер умеет работать только с числами, стоит понять, как преобразовать входное изображение. Так как сеть должна иметь фиксированное количество нейронов, то стоит преобразовать все изображения к одному общему размеру, чаще всего к некому квадрату, допустим, размерами 64х64 или 128х128. Поскольку известно, что изображение — набор пикселей размерностями от (0, 0, 0) до (255, 255, 255) в RGB формате, где (0, 0, 0) — черный цвет, а (255, 255, 255) — белый, то можно заметить, что если привести изображение к 3-х мерной матрице, где одна ось — высота, вторая — ширина, третья — количество каналов (в случае RGB — 3). Тогда мы получим некоторый тензор, в котором каждое значение варьируется от 0 до 255, однако, такой вид замедляет вычисления в сети, поэтому можно попробовать поделить каждое значение на 255, таким образом значения в тензоре будут варьироваться от 0.0 до 1.0, что упрощает работу компьютеру и практически убирает возможность переполнения в памяти.

# 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>