**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

отчет

**по лабораторной работе №2**

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

**Тема: «Самобалансирующиеся двоичные деревья поиска»**

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

Санкт-Петербург

2024

**Оглавление**

[**Введение3**](#_Введение)

1. [**Теоретическая часть4**](#_Теоретическая_часть)
   1. [Бинарное дерево поиска4](#_Бинарное_дерево_поиска)
   2. [AVL дерево7](#_AVL_дерево)
   3. [Красно-черное дерево10](#_Красно-черное_дерево)
2. [**Практическая часть15**](#_Сортировка_слиянием)
   1. [Зависимость высоты для бинарного дерева поиска15](#_Зависимость_высоты_для)
   2. [Зависимость высоты для AVL дерева17](#_Сортировка_выбором_1)
   3. [Зависимость высоты для красно-черного дерево](#_Сортировка_пузырьком_1)18
   4. [Реализация обхода деревьев 20](#_Реализация_обхода_деревьев)

[**Код** **программы23**](#_Код_программы)

[**Заключение36**](#_Заключение)

[**Ссылка на**](#_Ссылка_на_репозиторий)[**репозиторий GitHub36**](#_Ссылка_на_репозиторий)

# Введение

**Цель:** реализация бинарного, AVL и красно-черного деревьев поиска. Исследовать их зависимость высоты от количества ключей.

**Задачи:** для каждого вида дерева:

1. Реализовать функции вставки, удаления, поиска элемента. Для самобалансирующихся деревьев реализовать последующую балансировку.
2. Получить зависимость высоты дерева от количества ключей в этом дереве.
3. Сравнить практические результаты с теоретическими. Вывести итог на графики.
4. Реализовать обходы в глубину и ширину для каждого дерева.

# Теоретическая часть

# Бинарное дерево поиска

1. Сведения о дереве:

Бинарное дерево поиска – структура данных, которая имеет вид так называемого «дерева», в котором соблюдается сразу несколько условий:

* 1. Каждый узел имеет не более двух потомков – левый и правый.
  2. Значение всех узлов в левом поддереве меньше значения текущего узла.
  3. Значение всех узлов в правом поддереве больше значения текущего узла.

Каждое из этих условий формируют преимущества и недостатки бинарного дерева поиска, из которых можно выделить:

В среднем случае позволяет выполнять все действия (вставка, удаление, поиск) за логарифмическое время, а также не требует дополнительных действий для сортировки дерева по возрастанию или убыванию, поскольку по самому определению бинарного дерева поиска – это уже реализовано. Но при этом если дерево становится нестабильным, то все эти операции могут быть очень и очень неэффективными по сравнению с остальными структурами.

1. Описание алгоритмов удаления и добавления элементов:

В бинарное дерево поиска элемент добавляется следующим образом:

1. Проверяется наличие корня, если его нет, то на место корня встает элемент, который мы пытаемся добавить.
2. Если же корень уже есть, то значение ключа для элемента, который мы хотим добавить, сравнивается со значением ключа элемента, с которым мы его сравниваем. Если значение ключа элемента, который мы хотим добавить, больше значения текущего ключа, то наш элемент идет дальше по правому поддереву, если меньше, то по левому.
3. Так продолжается до тех пор, пока наш элемент не встанет на свое место.

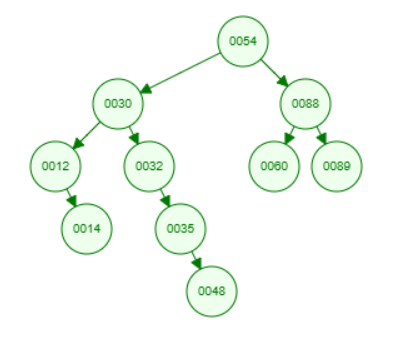
При удалении элемента из бинарного дерева поиска:

1. Для начала вызывается рекурсивная функция, которая проверяет, чему равно значение узла, если такой узел равен пустому значению – функция возвращает пустое значение.
2. Если же значение не нулевое, то для удаления узла, где проверяется, больше, меньше или равно значение текущего узла значению, которое мы хотим удалить.
3. Если значение больше или меньше, то происходит рекурсивный вызов этой же функции, но уже для правого или левого узла поддерева соответственно.
4. Если у узла нет детей – он просто удаляется, если же у него у него лишь один ребенок, то он просто заменяет узел, а если у него два ребенка, то мы ищем наименьший элемент в правом поддереве, заменяя данные текущего узла на данные минимального узла, рекурсивно удаляя минимальный узел.
5. Расчет зависимости высоты от количества ключей.

Так как бинарное дерево поиска не имеет никаких принципов балансировки, то и зависимость высоты будет очень сильно разниться, в зависимости от случая, начиная от лучшего, заканчивая худшим. Тогда рассчитаем это значение для среднего случая, поскольку для всех дальнейших деревьев лучший случай асимптотика зависимости высоты от количества ключей будет отличаться от худшего и среднего только константой.

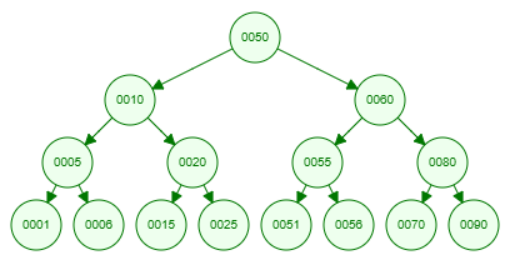
Тогда для среднего случая можем попытаться вывести формулу. Средний случай для нас означает то, что элементы в среднем располагаются практически равномерно, то есть лучший случай (полностью балансированное дерево), но с некоторой разницей в высотах поддеревьев. Тогда рассмотрим некоторый пример:

Изображение 1. Пример бинарного дерева поиска.



В таком случае мы можем заметить, что если мы продолжим добавлять элементы, то дерево продолжит делить на два каждое поддерево, но с некоторыми различиями. Тогда проведя некоторые аналитические действия, можем понять, что в лучшем случае высота дерева будет равна , поскольку дерево будет идеально структурировано, как на примере ниже:

Изображение 2. Пример идеального бинарного дерева поиска.



Проверим наше предположение, количество элементов , тогда по формуле . Получается наше предположение было верно. Тогда мы можем предположить, что в среднем случае наша высота будет равна так же , но с некоторой константой . Данную константу можно будет попытаться вывести с помощью практических значений позже. Итого верхняя оценка высота бинарного дерева поиска

# AVL дерево

1. Сведения о дереве:

AVL дерево – самобалансирующееся дерево, одно из возможных улучшений обыкновенного бинарного дерева поиска, который точно так же имеет свое свойство (свойства обычного бинарного дерева поиска сохраняются):

1. Значение разности высоты левого и правого поддерева на одном уровне должны по модулю быть меньше 2.

Именно это свойство формирует преимущества и недостатки AVL дерева, из которых можно выделить:

Во-первых, такое свойство позволяет с уверенностью сказать, что AVL дерево равномерно распределены по поддеревьям, так как высоты у поддеревьев будут различаться не более, чем на 1. Во-вторых, такая система обеспечивает более быстрый поиск, добавление и удаление элементов, поскольку даже в худшем случае высота дерева не может быть ненормированной, например, как в бинарном дереве поиска, когда дерево имеет только одно поддерево. Но также, такое дерево при достаточном количестве элементов требует больше времени на повороты и балансировку.

1. Описание алгоритмов удаления и добавления элементов:

В AVL дерево добавление элемента осуществляется таким же образом, как и в бинарном дереве поиска, поскольку сохраняет все его свойства. Но есть некоторое различие – повороты, которые применяются после добавления элемента и сравнения высоты поддеревьев, поэтому ниже будут рассмотрены не алгоритмы удаления и добавления, а алгоритмы поворотов, поскольку алгоритмы удаления и добавления были рассмотрены ранее:

* 1. Правый поворот:

Этот поворот применяется, если баланс узла превышает допустимое значение (+1), и дисбаланс возникает из-за добавления элемента в **левое поддерево левого ребенка.**

**Суть поворота:** левый ребенок проблемного узла становится новым корнем поддерева. Узел, который вызвал дисбаланс, перемещается вправо, а правое поддерево нового корня становится левым поддеревом старого корня. Этот поворот уменьшает высоту левого поддерева, восстанавливая баланс.

* 1. Левый поворот:

Этот поворот используется, если баланс узла становится меньше допустимого значения (-1), и дисбаланс возникает из-за добавления элемента в **правое поддерево правого ребенка**.

**Суть поворота:** правый ребенок проблемного узла становится новым корнем поддерева. Узел, который вызвал дисбаланс, перемещается влево, а левое поддерево нового корня становится правым поддеревом старого корня. Этот поворот уменьшает высоту правого поддерева, восстанавливая баланс.

* 1. Лево-правый поворот:

Этот поворот выполняется, если баланс узла превышает допустимое значение (+1), но дисбаланс вызван элементом, добавленным в **правое поддерево левого ребенка**.

**Суть поворота:** сначала выполняется левый поворот на левом ребенке проблемного узла, чтобы привести поддерево к форме, подходящей для правого поворота. Затем выполняется правый поворот на проблемном узле, восстанавливающий баланс.

* 1. Право-левый поворот:

Этот поворот применяется, если баланс узла становится меньше допустимого значения (-1), а дисбаланс вызван элементом, добавленным в **левое поддерево правого ребенка**.

**Суть поворота:** сначала выполняется правый поворот на правом ребенке проблемного узла, чтобы выровнять поддерево. Затем выполняется левый поворот на проблемном узле для окончательного восстановления баланса.

При удалении элемента из AVL дерева:

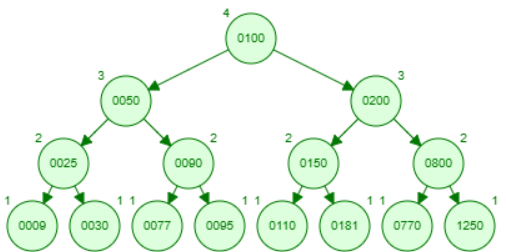
Точно также, как и при добавлении элемента – сначала выполняется базовая операция удаления для бинарного дерева поиска, после чего выполняется один из описанных выше поворотов, для каждой ситуации свой поворот.

1. Расчет верхней оценки высоты дерева:

Для расчета верхней оценки высоты AVL дерева можем использовать его основное свойство – разница левого и правого поддеревьев по модулю должна быть не больше единицы.

Тогда для среднего случая можем попытаться вывести формулу с учетом данного свойства. Несложно догадаться, что поскольку дерево должно быть сбалансированно, то разница между лучшим, худшим и средним случаями будет не больше единицы, а это дает нам понять, что высота во всех случаях будет практически одинакова. Тогда используя свойство AVL дерева, а также вычисления, полученные для бинарного дерева поиска, можем сказать, что дерево постоянно делится на два и по своему принципу, представляет из себя лучший случай бинарного дерева поиска. Таким образом верхняя оценка сложности высоты AVL дерева будет: .

Изображение 3. Пример AVL дерева.



# Красно-черное дерево

1. Сведения о дереве:

Красно-черное дерево – самобалансирующееся дерево, одно из возможных улучшений обыкновенного бинарного дерева поиска, который точно так же имеет свои свойства (свойства обычного бинарного дерева поиска сохраняются):

1. Каждый элемент имеет свой цвет – красный или черный.
2. Корневой элемент всегда имеет черный цвет.
3. У красного родителя не может быть красного ребенка.
4. Листовые узлы дерева (“NULL” элементы) – всегда черные.
5. На каждом пути от корня к любому листовому узлу должно быть одинаковое количество черных узлов.

Такие свойства формирует преимущества и недостатки красно-черного дерева, из которых можно выделить:

Во-первых, такие свойства дают преимущество красно-черному дереву перед AVL деревом, поскольку оно является чуть менее балансированным, следовательно, количество поворотов будет меньше, что требует меньше времени выполнения по сравнению с AVL. Но также, такое дерево может иметь более высокие поддеревья, что делает поиск необходимого ключа менее быстрым, чем в AVL.

1. Описание алгоритмов удаления и добавления элементов:

В красно-черное дерево, также как и в AVL добавление элемента осуществляется таким же образом, как и в бинарном дереве поиска, поскольку сохраняет все его свойства. Но есть некоторое различие – повороты и изменение цвета, которые применяются после добавления элемента и сравнения высоты поддеревьев. Поэтому ниже будут рассмотрены алгоритмы поворотов и перекраски цветов после вставки элемента:

* 1. Правый поворот:

**Используется**, если баланс узла нарушен и его левое поддерево слишком высокое.

Суть поворота: левый ребенок узла становится новым корнем поддерева. Узел, вызвавший дисбаланс, перемещается вправо, а правое поддерево нового корня становится левым поддеревом старого корня. Это уменьшает высоту левого поддерева и восстанавливает баланс.

1. Левый поворот:

**Используется**, если баланс узла нарушен и его правое поддерево слишком высокое.

Суть поворота: правый ребенок узла становится новым корнем поддерева. Узел, вызвавший дисбаланс, перемещается влево, а левое поддерево нового корня становится правым поддеревом старого корня. Это уменьшает высоту правого поддерева и восстанавливает баланс.

1. Лево-правый поворот:

**Используется,** если баланс узла нарушен из-за того, что элемент был добавлен в правое поддерево левого ребенка.

**Суть поворота:** сначала выполняется левый поворот на левом ребенке, чтобы привести поддерево к правильной форме. Затем выполняется правый поворот на родительском узле, восстанавливая баланс дерева.

1. Право-левый поворот:

Используется, если баланс узла нарушен из-за того, что элемент был добавлен в левое поддерево правого ребенка.

Суть поворота: сначала выполняется правый поворот на правом ребенке, чтобы выровнять поддерево. Затем выполняется левый поворот на родительском узле для окончательного восстановления баланса.

1. Перекраска элемента:

Используется, если нарушено одно из свойств дерева (корень не черный, красный узел имеет красного ребенка или каждый путь от корня до листа содержит разное количество черных узлов).

Суть перекраски: сначала идет проверка на то, является ли родитель вставленного элемента красным, если да – то требуется перекраска, потому что нарушается одно из свойств. После чего у нас имеется 4 варианта событий, до тех пор, пока родитель элемента существует, и его цвет равен красному.

Во-первых, если родитель элемента – левый ребенок, то если существует дядя и его цвет равен красному, то цвет родителя нашего элемента и цвет его дяди, перекрашиваются в черный, при этом, его дедушка перекрашивается в красный цвет и далее рассматриваться будет не наш элемент, а его дедушка.

Во-вторых, если у нашего элемента нет дяди, или он есть, но его цвет равен черному, то если текущий элемент – правый ребенок, то теперь рассматриваться будет не наш элемент, а его родитель, для которого выполняется левый поворот, после чего его родителю присваивается черный цвет, а дедушке – красный и происходит правый поворот для дедушки.

В-третьих, если родитель нашего элемента является правым ребенком, то если дядя существует и имеет красный цвет, то цвет дяди и родителя нашего элемента перекрашиваются в черный, а дедушка перекрашивается в красный, после чего рассматриваться будет не наш элемент, а его дедушка.

В-четвертых, если дяди не существует или его цвет черный, то идет проверка на то, является ли наш элемент левым ребенком, если да, то далее рассматриваться будет его родитель и применен правый поворот. После чего цвет родителя элемента перекрашивается в черный, а дедушка в черный и происходит левый поворот.

Таким образом, повороты используются только во втором и четвертом случаях. Во втором может быть либо лево-правый поворот, либо просто правый. В четвертом случае может быть либо право-левый поворот, либо просто левый.

При удалении элемента из бинарного дерева поиска:

Точно также, как и при добавлении элемента – сначала выполняется базовая операция удаления для бинарного дерева поиска, после чего выполняется один из описанных выше поворотов внутри реализации изменения цветов. Цвета при удалении изменяются по алгоритму, при этом он вызывается только если элемент, который мы поставили на место нашего удаленного элемента имеет черный цвет:

Во-первых, до тех пор, пока узел не равен корню и цвет узла не равен черному, мы проверяем, не является ли наш элемент левым ребенком, если это так, то если цвет брата равен красному, то мы то мы перекрашиваем его цвет в черный, а цвет родителя в красный и вызываем левый поворот, а так же после поворота снова сохраняем информацию о том, что брат – правый ребенок родителя.

Во-вторых, если у брата нашего элемента нет левого ребенка или цвет его левого ребенка равен черному, а также у него нет правого ребенка или у цвет его правого ребенка равен черному, то мы меняем цвет брата на красный, а вместо нашего элемента начинаем рассматривать родителя.

В-третьих, если ни одно из начальных условий выше не соблюдается, то если правого ребенка брата нашего элемента нет, или его цвет равен черному, то если у брата есть левый ребенок, цвет левого ребенка становится черным. После чего цвет нашего брата становится красным, выполняется правый поворот для брата и снова меняется наш брат. После выхода из проверки на наличие правого ребенка у брата или наличия черного цвета у правого ребенка брата выполняется смена цвета брата на родительский, а цвет родителей меняется на черный. Затем происходит проверка на то, есть ли у нашего брата правый ребенок, если это так – цвет этого ребенка становится черным. После всех этих действий выполняется левый поворот для родителя и рассматриваться далее будет не наш элемент, а корень дерева.

В-четвертых, в-пятых, и в-шестых, действия выполняются точно так же, как и в первых трех пунктах, но зеркально. Например, вместо левого поворота в первом случае, в четвертом случае будет делаться правый поворот, поскольку наш элемент будет не левым, а правым ребенком.

После всех действий цвет корня меняется на черный.

1. Расчет верхней оценки высоты дерева:

Также как и с AVL деревом – в лучшем случае красно-черное дерево является идеальным случаем для обыкновенного бинарного дерева поиска, то есть оно имеет сложность Однако, поскольку AVL дерево в силу своих свойств может иметь разницу в высотах поддеревьев до единицы, то оно является более балансированным. В свою очередь красно-черное дерево имеет другие свойства, которые хоть и делают его более балансированным, чем бинарное дерево поиска, но менее сбалансированным по сравнению с AVL деревом. Одно из таких свойств – количество черных элементов в одном поддереве не может быть более двух высот другого поддерева, откуда в худший случай для такого дерева добавляется константа, которая равна 2, поскольку высота может отличаться в два раза.

Однако для получения средней оценки, мы можем считать, что высоты пусть и отличаются, но не в 2 раза, как в худшем случае. Следовательно, верхняя оценка высоты дерева не будет превышать . Тогда можем предположить, что если у нас лишь какая-то часть будет немного больше, чем ее противолежащая часть, то можем сделать вывод, что средний случай лежит между лучшим и худшим, а то есть между константами . Тогда пусть итоговая сложность будет

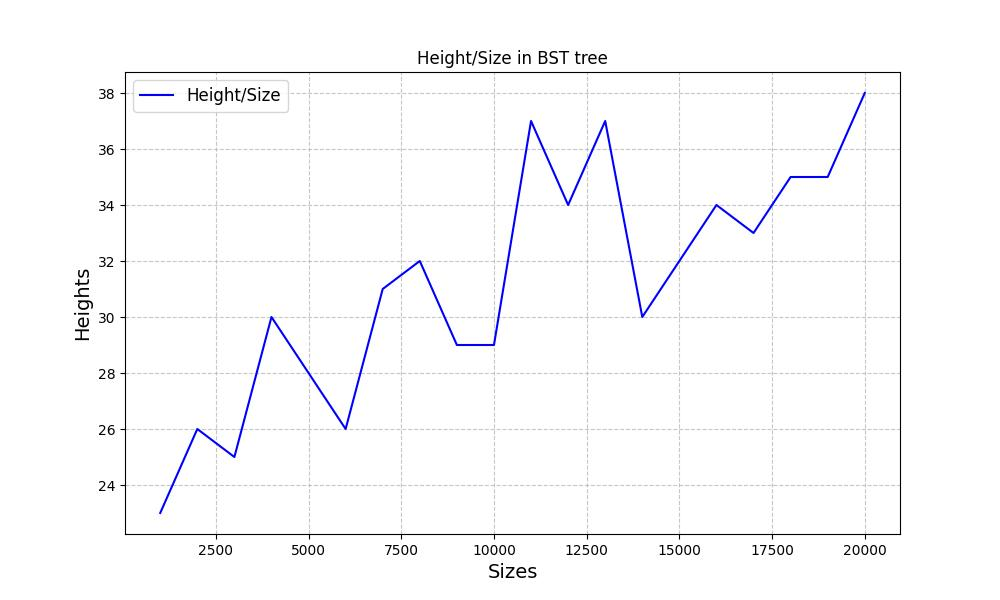
# Практическая часть.

# Зависимость высоты для бинарного дерева поиска.

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

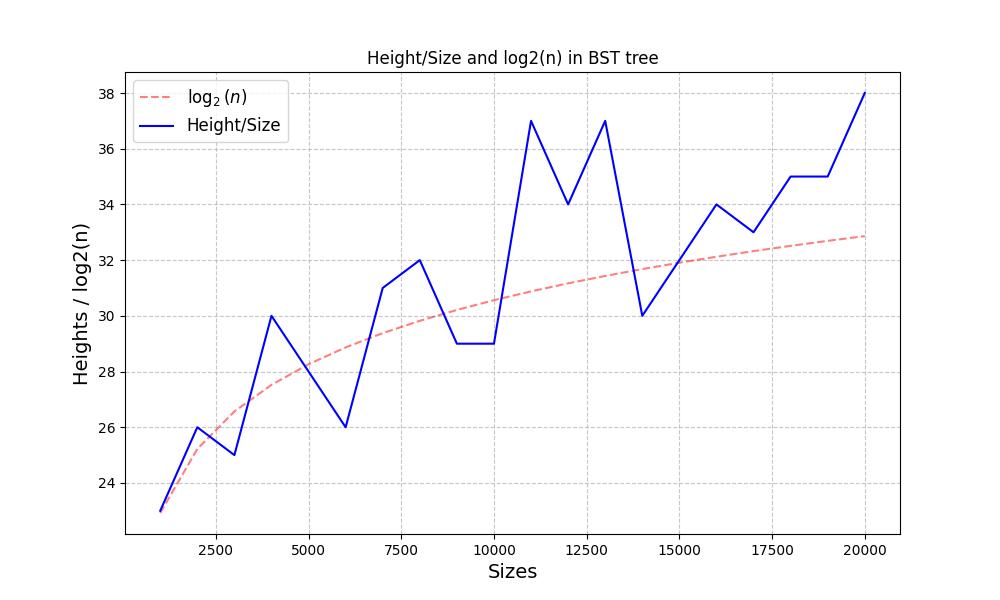
|  |  |  |
| --- | --- | --- |
| Номер | Количество ключей | Высота |
| 1 | 1000 | 23 |
| 2 | 2000 | 26 |
| 3 | 3000 | 25 |
| 4 | 4000 | 30 |
| 5 | 5000 | 28 |
| 6 | 6000 | 26 |
| 7 | 7000 | 31 |
| 8 | 8000 | 32 |
| 9 | 9000 | 29 |
| 10 | 10000 | 29 |
| 11 | 11000 | 37 |
| 12 | 12000 | 34 |
| 13 | 13000 | 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 имеет некую константу, что бы 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.



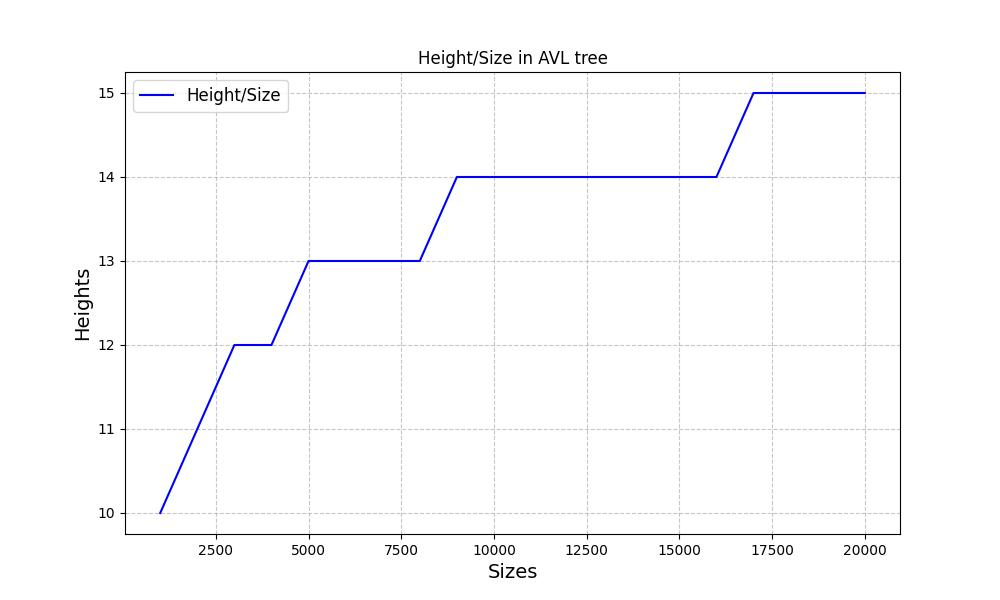
Можно заметить, что график не то, чтобы сходится с теоретическим предсказанием. Однако следует учитывать, что в теоретической части была получена верхняя оценка, а значит, что на особо больших значениях, график в среднем выровняется и частично совпадет с теоретическими значениями. Скачки же в моем графике 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

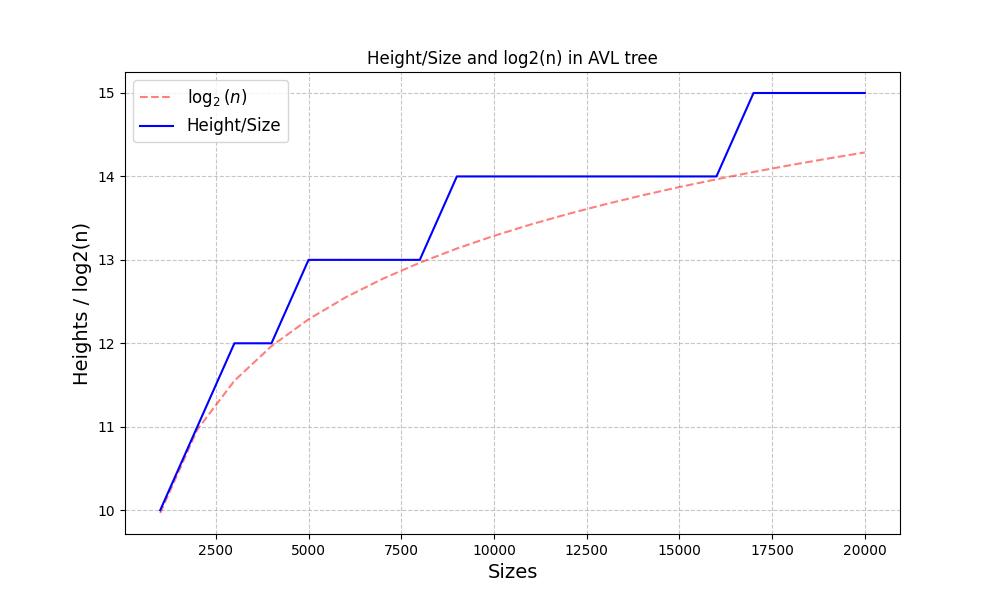
|  |  |  |
| --- | --- | --- |
| Номер | Количество ключей | Высота |
| 1 | 1000 | 10 |
| 2 | 2000 | 11 |
| 3 | 3000 | 12 |
| 4 | 4000 | 12 |
| 5 | 5000 | 13 |
| 6 | 6000 | 13 |
| 7 | 7000 | 13 |
| 8 | 8000 | 13 |
| 9 | 9000 | 14 |
| 10 | 10000 | 14 |
| 11 | 11000 | 14 |
| 12 | 12000 | 14 |
| 13 | 13000 | 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.



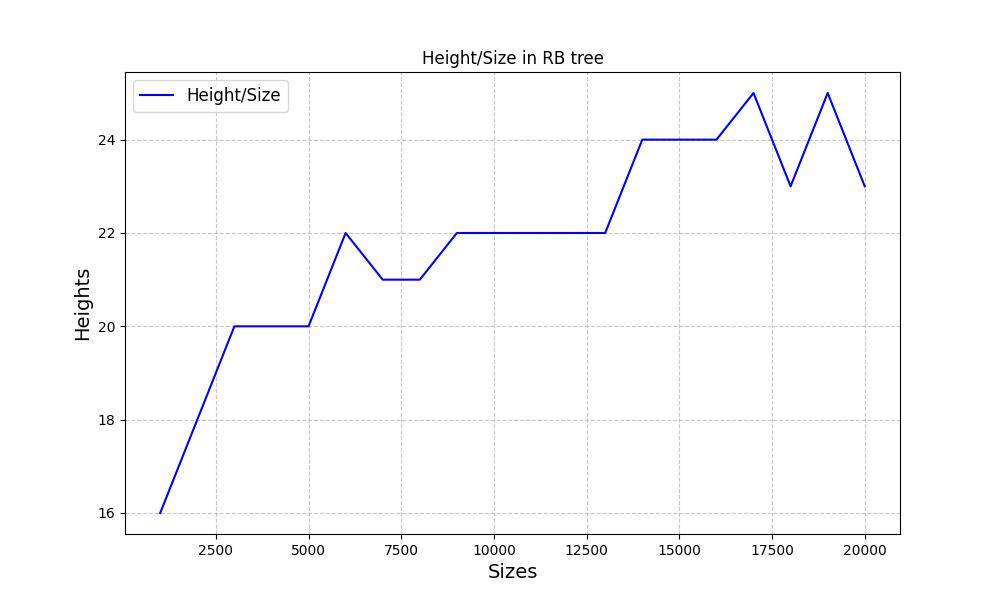
Как и было сказано ранее, график полностью совпал с ожидаемым, отсутствие скачков обусловлено свойствами AVL дерева.

# Зависимость высоты для красно-черного дерева.

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

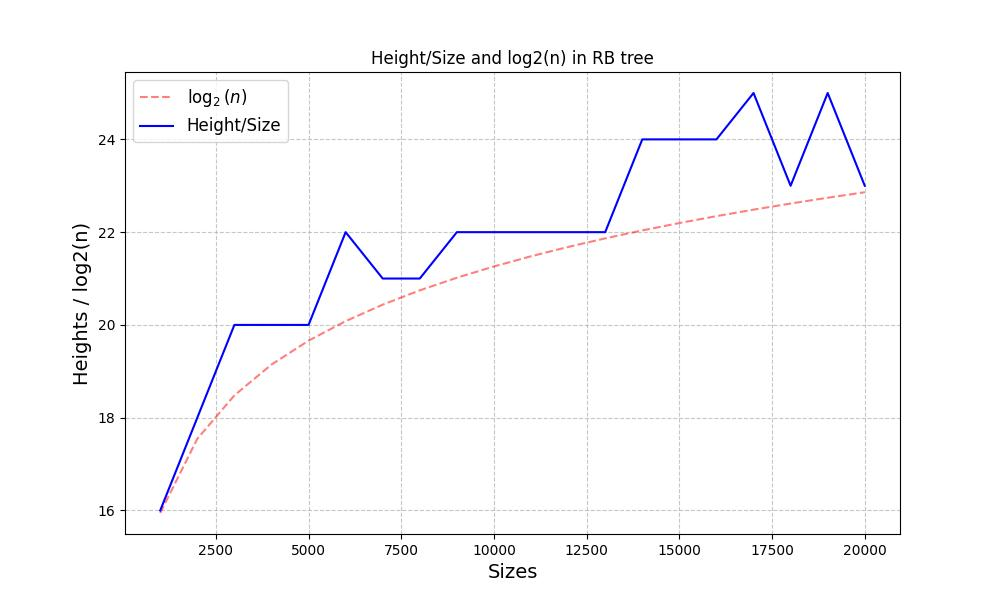
|  |  |  |
| --- | --- | --- |
| Номер | Количество ключей | Высота |
| 1 | 1000 | 16 |
| 2 | 2000 | 18 |
| 3 | 3000 | 20 |
| 4 | 4000 | 20 |
| 5 | 5000 | 20 |
| 6 | 6000 | 22 |
| 7 | 7000 | 21 |
| 8 | 8000 | 21 |
| 9 | 9000 | 22 |
| 10 | 10000 | 22 |
| 11 | 11000 | 22 |
| 12 | 12000 | 22 |
| 13 | 13000 | 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 логарифма был найден элементарным перебором, и он получился равен , это было сделано, поскольку как писалось ранее – верхняя оценка высоты дерева может разниться в зависимости от случая.

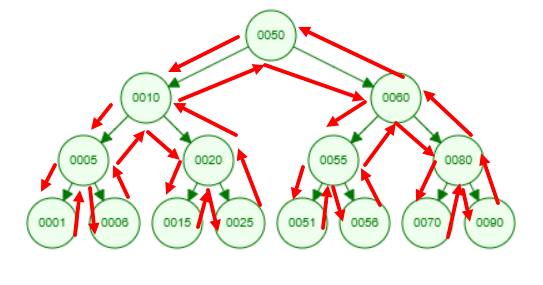
Однако подводя итоги по красно-черному дереву, можно сказать, что в целом теория вновь совпала с практикой.

# Реализация обхода деревьев

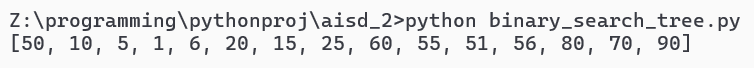
Реализация обхода была реализована только в бинарном дереве поиска, поскольку остальные виды деревьев просто наследовали данные методы.

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.

Изображение 10. Иллюстрация работы обхода в глубину.



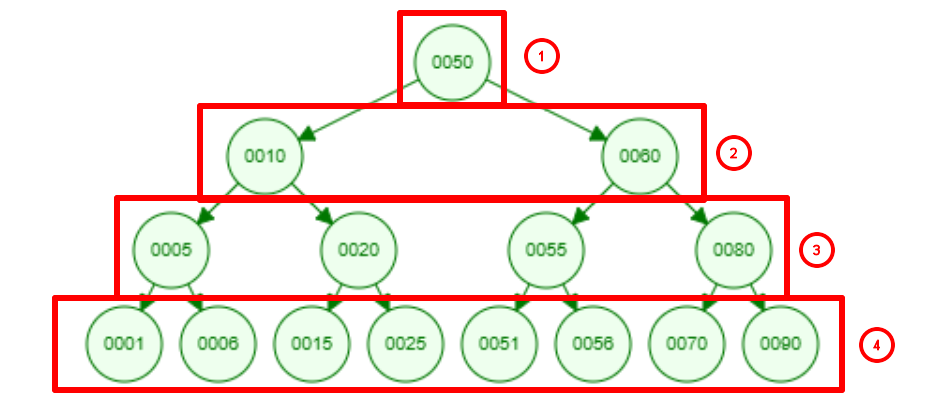
Изображение 11. Пример вывода для обхода в глубину.



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):

result = []

def \_dfs(node):

if node:

result.append(node.data)

\_dfs(node.left\_kid)

\_dfs(node.right\_kid)

\_dfs(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>