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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Реализация структуры данных для конкретной ситуации

Студент гр. 3384 _____ Рудаков А.Л.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург

2024

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Рудаков А.Л.

Группа 3384

Тема работы: Реализация структуры данных для конкретной ситуации

Исходные данные:
Выбрать оптимально-подходящую структуру данных для конкретной ситуации. Провести исследование и сравнение с другими структурами данных. Реализовать выбранную структуру данных.

Содержание пояснительной записки:
«Содержание», «Введение», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:
Не менее 20 страниц.

Дата выдачи задания: 06.11.2024

Дата сдачи реферата: 24.11.2024

Дата защиты реферата: 25.11.2024

Студент _____ Рудаков А.Л.

Преподаватель _____ Шестопалов Р.П.

АННОТАЦИЯ

Вариант 8.

Программа представляет собой рейтинговую систему, которая хранит набранные игроками очки. При использовании можно узнать, был ли текущий результат в рейтинговой таблице, узнать ближайший результат, узнать все результаты, выше данного, а также добавить свой результат в рейтинг.

SUMMARY

Option 8.

The program is a rating system that stores the points scored by the players. When using it, you can find out if the current result was in the rating table, find out the nearest result, find out all the results above this one, and also add your result to the rating.

СОДЕРЖАНИЕ

	Введение	5
1.	Исследование	6
2.	Реализация	9
2.1.	Класс элемента дерева	9
2.2.	Вспомогательные функции	9
2.3.	Функции поворотов	9
2.4.	Функция добавления элемента	10
2.5.	Функция поиска	11
2.6.	Функции вывода	12
3.	Тестирование	13
3.1.	Вставка	13
3.2.	Поиск	15
3.3.	Вывод элементов с результатом, больших данного	16
	Заключение	20
	Список использованных источников	21
	Приложение А. КОД	22

ВВЕДЕНИЕ

Цель работы: Выбрать оптимальную для ситуации структуру данных и реализовать её.

Задачи: Для достижения поставленной цели требуется провести исследование различных структур данных и выбрать лучшую для данной задачи. После необходимо реализовать выбранную структуру на языке Python.

1. ИССЛЕДОВАНИЕ

Основные задачи структуры данных для заданной ситуации:

- Хранение рейтинговой системы
- Поиск результата с таким же количеством очков
- Поиск ближайшего по количеству очков результата
- Поиск всех результатов с количеством очков больше данного
- Вывод всех результатов с количество очков больше данного в порядке возрастания
- Добавление своего результата в рейтинг (очень редко)

В данных задачах чаще всего используется поиск, поэтому хорошим вариантом будет структура данных, в которой поиск быстрее остальных. Также важен порядок хранения данных, так как требуется вывод значений выше данного в отсортированном виде.

Сразу убираются из вариантов массив и односвязный и двусвязный списки, так как в них поиск элемента по значению выполняется за O(n). Кроме того, в задаче есть добавление элементов, хоть оно и редкое. В массиве вставка элементов осуществляется за O(n), из-за чего от точно не подходит. В списке вставка осуществляется за O(1), но в то же время требуется хранить данные в упорядоченном виде, вследствие чего после каждой вставки требуется сортировка списка, что в среднем случае занимает O(nlog(n)), а в худшем O(n^2).

Вариант хэш-таблицы подходит для поиска и вставки, так как в среднем случае это будет занимать O(1), но в худшем случае O(n). При этом хэш-таблица не может хранить данные в отсортированном виде, что становится большой проблемой, так как в таком случае вывод значений в отсортированном виде будет занимать O(n).

Варианты деревьев:

- Бинарное дерево

- Бинарное дерево поиска
- Красно-черное дерево
- АВЛ дерево

Несбалансированные деревья, такие как бинарное дерево и бинарное дерево поиска, не подойдут для данной задачи, так как хоть и в среднем случае операция поиска имеет логарифмическую сложность, но в худшем случае сложность может деградировать до линейной сложности. Операция вставки элемента имеет такую же проблему.

Сбалансированные деревья являются хорошим вариантом для данной задачи, так как операции поиска и вставки в них выполняются за O(log(n)) вне зависимости от того, лучший, средний или худший случай. Хоть они и имеют дополнительные расходы на поддержание сбалансированности, сбалансированные деревья все равно остаются быстрее чем несбалансированные.

Оба сбалансированных дерева подходят для данной задачи, так как имеют примерно одинаковые временные сложности, но все же между ними имеются различия.

В АВЛ-дереве поиск выполняется быстрее, чем в красно-черном дереве, так как АВЛ-дерево поддерживает более строгий баланс, из-за чего его высота всегда логарифмически пропорциональна количеству узлов в дереве, в то время, как в КЧД более слабая гарантия баланса, что приводит к большей высоте. Кроме того, для вывода значений больше данного лучше подойдет баланс АВЛ-дерева, так как большие значения всегда находятся в правых поддеревьях. В это же время в красно-черных деревьях из-за менее строгого баланса правые поддеревья могу содержать как большие, так и меньшие значения, из-за чего для вывода больших значений придется исследовать больше поддеревьев, что делает процесс более медленным.

С другой стороны, вставка выполняется быстрее в КЧД, так как при каждой операции вставки АВЛ-дереву приходится обновлять коэффициенты

баланса, в то время как в красно-черном дереве перекрашивание и повороты выполняются только при необходимости.

Так как в задании прямо указано, что в структуре требуется частый поиск и очень редкая вставка, то для ситуации оптимально подойдет АВЛ-дерево.

2. РЕАЛИЗАЦИЯ

2.1. Класс элемента дерева

Класс *Score* представляет собой элемент дерева и хранит в себе поля: *score* – количество очков, names – список никнеймов игроков с данным количество очков, *height* – высота элемента, *right* и *left* – правый и левый элементы соответственно.

Кроме того, реализован метод *__str__* для корректного преобразования в строковый вид, используемый при выводе списка результатов, больших поданного.

Также реализован метод *out_str(),* для корректного преобразования в строковый вид, используемый при обходе дерева в функции *post_order(...)*.

2.2. Вспомогательные функции

Функция *height(root_score)* возвращает высоту дерева, с вершиной в поданном узле.

Функция *calc_height(root_score)* вычисляет высоту дерева, с вершиной в поданном узле исходя из высот правого и левого поддерева, используя функцию *height(...)*.

Функция *check_balance(root_score)* возвращает коэффициент баланса для правого и левого узла дерева, с вершиной в поданном узле, используя для вычислений функцию *height(...)*. Если возвращается 0, 1 или -1 – перебалансировка не нужна, если 2 – требуется левый поворот, если -2 – требуется правый поворот.

Функция *check_right_bigger(root_score)* возвращает *True,* если высота правого поддерева дерева, с вершиной в поданном узле, больше или равна высоте левого поддерева дерева, с вершиной в поданном узле, *False* в противном случае. При сравнении используется функция *height(...)*.

Функция *check_left_bigger(root_score)* возвращает *True,* если высота левого поддерева дерева, с вершиной в поданном узле, больше или равна

высоте правого поддерева дерева, с вершиной в поданном узле, *False* в противном случае. При сравнении используется функция *height(...)*.

2.3. Функции поворотов

Функция *left_rotate(root_score)* выполняет левый поворот дерева, с вершиной в поданном узле. Новой вершиной становится правый узел *root_score*, правое поддерево *root_score* становится равно левому поддереву новой вершины, дерево, с вершиной в *root_score*, становится левым поддеревом новой вершины. После проделанных операций у *root_score* и новой вершины обновляются высоты благодаря *calc_height(...)*, после чего возвращается новая вершина.

Функция *right_rotate(root_score)* выполняет правый поворот дерева, с вершиной в поданном узле. Новой вершиной становится левый узел *root_score*, левое поддерево *root_score* становится равно правому поддереву новой вершины, дерево, с вершиной в *root_score*, становится левым поддеревом новой вершины. После проделанных операций у *root_score* и новой вершины обновляются высоты благодаря *calc_height(...)*, после чего возвращается новая вершина.

2.4. Функция добавления элемента

Функция *insert(new_score, new_name, root_score)* вставляет новые значения в АВЛ-дерево. Внутри происходит рекурсивный обход дерева до момента, пока не найдется место, куда можно вставить элемент. Таких случаев может быть 2: найдено незанятое место (когда *root_score == None*), тогда создается новый объект класса *Score*, с поданными значениями, или же найден элемент дерева с таким же количеством очков, тогда в поле *names* данного элемента добавляется поданный никнейм. При рекурсивном обходе дерева текущий путь выбирается путем сравнения поданного количества очков и количества очков текущего элемента дерева, если вставляемое количество очков меньше, то вставка продолжается в левое поддерево, иначе в правое.

После нахождения места вставки происходит рекурсивный подъем вверх. На каждом этапе рассчитывается новая высота текущего элемента

благодаря функции *calc_height(root_score)* и проверяется на необходимость балансировки при помощи *check_balance(root_score)*.

Если необходим левый поворот, то при помощи *check_right_bigger(root_score.right)* проверяется является ли правое поддерево правого элемента текущего элемента больше или равно левому поддереву правого элемента текущего элемента. Если да, то выполняется малый левый поворот, используя функцию *left_rotate(root_score)*, если нет, выполняется большой левый поворот, путем правого поворота поддерева с вершиной в правом элементе текущего элемента, благодаря *right_rotate(root_score.right)*, после чего левый поворот дерева, с вершиной в текущем элементе, при помощи *left_rotate(root_score)*.

Если необходим правый поворот, то при помощи *not check_left_bigger(root_score.left)* проверяется, является ли правое поддерево левого элемента текущего элемента меньше или равно левому поддереву левого элемента текущего элемента. Если да, то выполняется малый правый поворот, используя функцию *right_rotate(root_score)*, если нет, выполняется большой правый поворот, путем левого поворота поддерева с вершиной в левом элементе текущего элемента, благодаря *left_rotate(root_score.left)*, после чего правый поворот дерева, с вершиной в текущем элементе, при помощи *right_rotate(root_score)*.

2.5. Функция поиска

Функция *find(root_score, find_score, min_sub)* возвращает элемент с поданным значением или, если такого нет в дереве, ближайший по значению элемент. Внутри происходит рекурсивный спуск по дереву в направлении к поданному значению. Вначале считается разница значения текущего элемента с поданным значением, если она меньше, чем была до этого, то разница изменяется на новую. Если разница равна 0, то сразу возвращается текущий элемент, и разница, равная 0.

Если значение текущего элемента больше поданного значения, проверяется левый элемент. Если его нет, то возвращается текущий элемент и

текущая разница, это будет ближайший больший элемент. Если левый элемент существует, то вызывается *find(root_score.left, find_score, min_sub)*, после чего по модулю сравнивается посчитанная разность и найденная разность, если посчитанная меньше, то возвращается найденный элемент и найденная разница, иначе возвращается текущий элемент и текущая разница.

Если значение текущего элемента меньше поданного значения, проверяется правый элемент. Если его нет, то возвращается текущий элемент и текущая разница, это будет ближайший меньший элемент. Если правый элемент существует, то вызывается *find(root_score.right, find_score, min_sub)*, после чего по модулю сравнивается посчитанная разность и найденная разность, если посчитанная меньше, то возвращается найденный элемент и найденная разница, иначе возвращается текущий элемент и текущая разница.

2.6. Функции вывода

Функция *print_biggest(root_score, min_score)* выводит все результаты больше данного в порядке возрастания. Внутри используется рекурсивный алгоритм обхода *InOrder*. Вначале, если существует левый элемент текущего элемента и при этом текущее значение больше поданного, то вызывается *print_biggest(root_score.left, min_score)*. После этого, если текущее значение больше поданного, то элемент выводится на экран. Далее, если существует правый элемент текущего элемента, то вызывается *print_biggest(root_score.right, min_score)*.

Функция *post_order(root_score)* выводит каждый элемент дерева в виде: *[left: x center: x right: x]*, проходя по дереву *PostOrder* обходом.

3. ТЕСТИРОВАНИЕ

3.1. Вставка.

Вставка элементов [1, ‘A’], [2, ‘B’], после чего вставка элемента [3, ‘C’], после которая вызывает малый левый поворот, показана на рис. 1. Наглядное представление в виде дерева показано на рис. 2.

```
h: 1 left: x                                center: Score: 2, Names: B              right: x
h: 2 left: x                                center: Score: 1, Names: A              right: Score: 2, Names: B
-------------------------------------------------------------------------------------------------------------
h: 1 left: x                                center: Score: 1, Names: A              right: x
h: 1 left: x                                center: Score: 3, Names: C              right: x
h: 2 left: Score: 1, Names: A               center: Score: 2, Names: B              right: Score: 3, Names: C
```

Рисунок 1 – Малый левый поворот

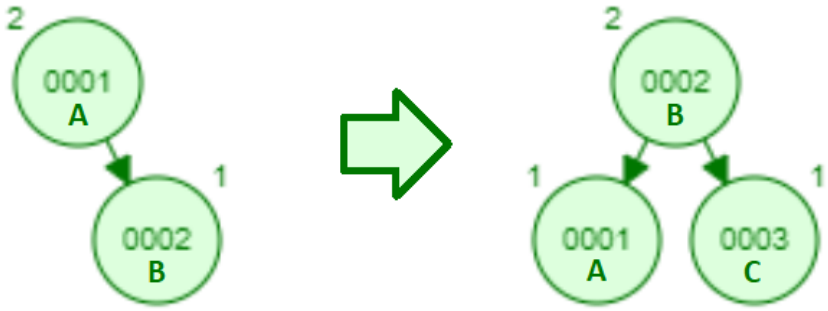

Рисунок 2 – Наглядное представление малого левого поворота

Вставка элементов [3, ‘A’], [2, ‘B’], после чего вставка элемента [1, ‘C’], которая вызывает малый правый поворот, показана на рис. 3. Наглядное представление в виде дерева показано на рис. 4.

```
h: 1 left: x                                center: Score: 2, Names: B              right: x
h: 2 left: Score: 2, Names: B               center: Score: 3, Names: A              right: x
-------------------------------------------------------------------------------------------------------------
h: 1 left: x                                center: Score: 1, Names: C              right: x
h: 1 left: x                                center: Score: 3, Names: A              right: x
h: 2 left: Score: 1, Names: C               center: Score: 2, Names: B              right: Score: 3, Names: A
```

Рисунок 3 – Малый правый поворот

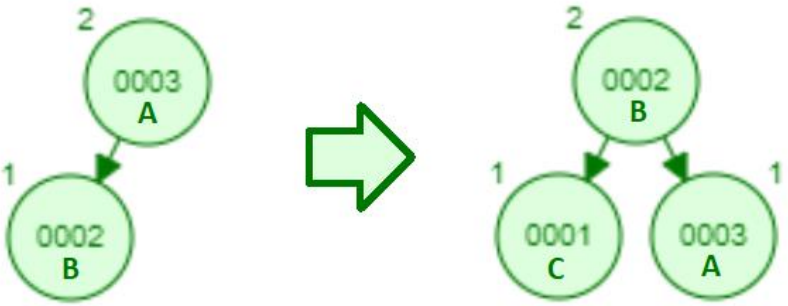

Рисунок 4 – Наглядное представление малого правого поворота

Вставка элементов [2, ‘A’], [5, ‘B’], [1, ‘C’], [6, ‘D’], [4, ‘E’], после чего вставка элемента [3, ‘F’], которая вызывает большой левый поворот, показана на рис. 5. Наглядное представление в виде дерева показано на рис. 6.

```
h: 1 left: x                          center: Score: 1, Names: C          right: x
h: 1 left: x                          center: Score: 4, Names: E          right: x
h: 1 left: x                          center: Score: 6, Names: D          right: x
h: 2 left: Score: 4, Names: E         center: Score: 5, Names: B          right: Score: 6, Names: D
h: 3 left: Score: 1, Names: C         center: Score: 2, Names: A          right: Score: 5, Names: B
-------------------------------------------------------------------------------------------------------
h: 1 left: x                          center: Score: 1, Names: C          right: x
h: 1 left: x                          center: Score: 3, Names: F          right: x
h: 2 left: Score: 1, Names: C         center: Score: 2, Names: A          right: Score: 3, Names: F
h: 1 left: x                          center: Score: 6, Names: D          right: x
h: 2 left: x                          center: Score: 5, Names: B          right: Score: 6, Names: D
h: 3 left: Score: 2, Names: A         center: Score: 4, Names: E          right: Score: 5, Names: B
```

Рисунок 5 – Большой левый поворот

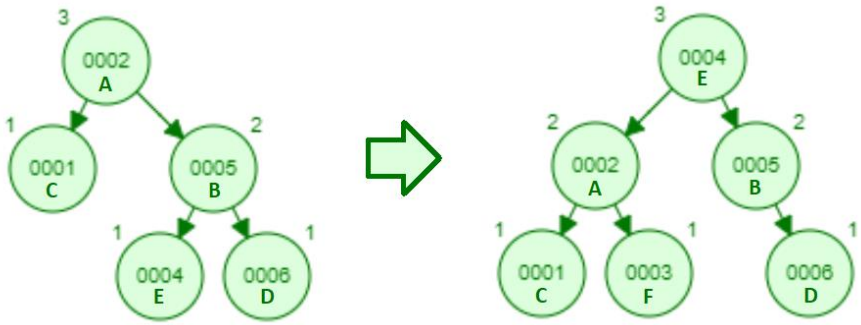

Рисунок 6 – Наглядное представление большого левого поворота

Вставка элементов [6, ‘A’], [5, ‘B’], [2, ‘C’], [1, ‘D’], [3, ‘E’], после чего вставка элемента [4, ‘F’], которая вызывает большой правый поворот, показана на рис. 7. Наглядное представление в виде дерева показано на рис. 8.

```
h: 1 left: x                          center: Score: 1, Names: D          right: x
h: 1 left: x                          center: Score: 3, Names: E          right: x
h: 2 left: Score: 1, Names: D         center: Score: 2, Names: C          right: Score: 3, Names: E
h: 1 left: x                          center: Score: 6, Names: A          right: x
h: 3 left: Score: 2, Names: C         center: Score: 5, Names: B          right: Score: 6, Names: A
-------------------------------------------------------------------------------------------------------
h: 1 left: x                          center: Score: 1, Names: D          right: x
h: 2 left: Score: 1, Names: D         center: Score: 2, Names: C          right: x
h: 1 left: x                          center: Score: 4, Names: F          right: x
h: 1 left: x                          center: Score: 6, Names: A          right: x
h: 2 left: Score: 4, Names: F         center: Score: 5, Names: B          right: Score: 6, Names: A
h: 3 left: Score: 2, Names: C         center: Score: 3, Names: E          right: Score: 5, Names: B
```

Рисунок 7 – Большой правый поворот

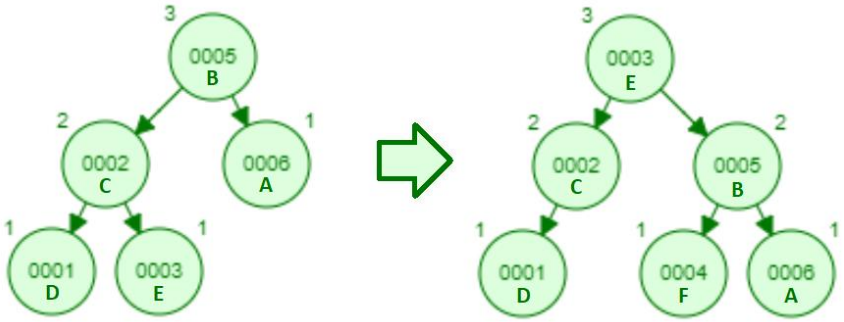

Рисунок 8 – Наглядное представление большого правого поворота

Вставка элементов [1, ‘A’], [2, ‘B’], [3, ‘C’], после чего вставка элемента [1, ‘F’], которая добавляет имя к элементу 1, показана на рис. 9. Наглядное представление в виде дерева показано на рис. 10.

```
h: 1 left: x                          center: Score: 1, Names: A          right: x
h: 1 left: x                          center: Score: 3, Names: C          right: x
h: 2 left: Score: 1, Names: A         center: Score: 2, Names: B          right: Score: 3, Names: C
-------------------------------------------------------------------------------------------------------
h: 1 left: x                          center: Score: 1, Names: A, F       right: x
h: 1 left: x                          center: Score: 3, Names: C          right: x
h: 2 left: Score: 1, Names: A, F      center: Score: 2, Names: B          right: Score: 3, Names: C
```

Рисунок 9 – Большой левый поворот

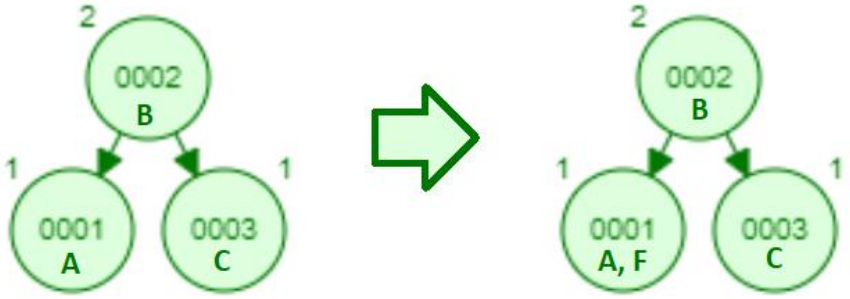

Рисунок 10 – Наглядное представление большого левого поворота

3.2. Поиск.

Вставка элементов [2, ‘A’], [4, ‘B’], [1, ‘C’], [8, ‘D’], после чего поиск элемента с количеством очков равным 1. Результат показан на рис. 11.

```
h: 1 left: x                      center: Score: 1, Names: C     right: x
h: 1 left: x                      center: Score: 8, Names: D     right: x
h: 2 left: x                      center: Score: 4, Names: B     right: Score: 8, Names: D
h: 3 left: Score: 1, Names: C     center: Score: 2, Names: A     right: Score: 4, Names: B

Score: 1, Name: C Sub: 0
```

Рисунок 11 – Поиск элемента, находящегося в дереве

Вставка элементов [2, ‘A’], [4, ‘B’], [1, ‘C’], [8, ‘D’], после чего поиск элемента с количеством очков равным 5. Результат показан на рис. 12.

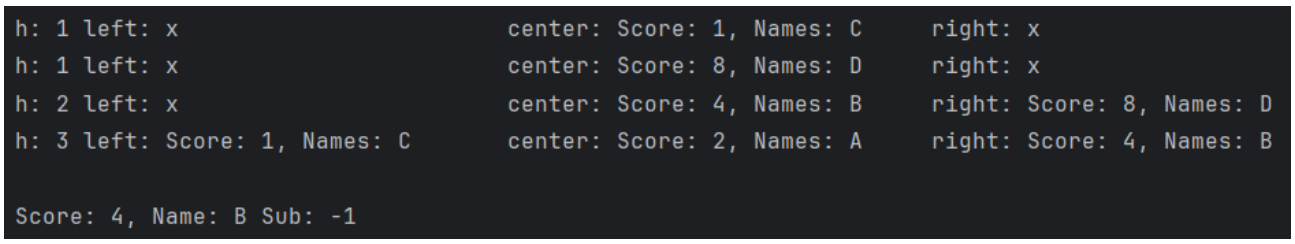

Рисунок 12 – Поиск элемента, не находящегося в дереве, больше ближайшего

Вставка элементов [2, ‘A’], [4, ‘B’], [1, ‘C’], [8, ‘D’], после чего поиск элемента с количеством очков равным 7. Результат показан на рис. 13.

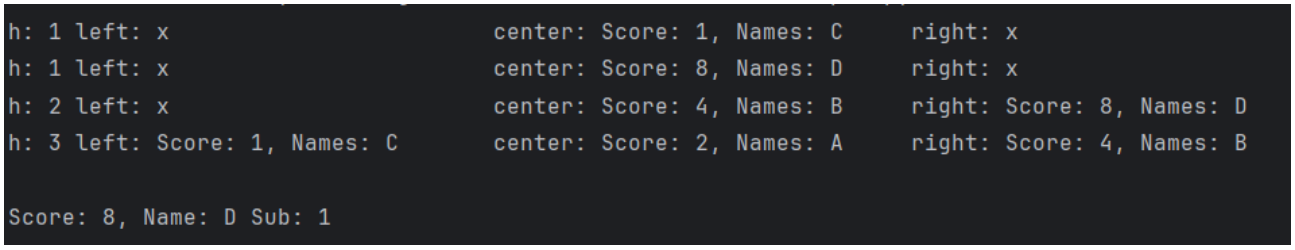

Рисунок 13 – Поиск элемента, не находящегося в дереве, меньше ближайшего

Вставка элементов [2, ‘A’], [4, ‘B’], [1, ‘C’], [8, ‘D’], после чего поиск элемента с количеством очков равным 3. В данном случае будет найден результат с количеством очков равным 4, так как результат 3 находится ровно между 2 и 4, в таком случае будет выдаваться больший результат. Результат показан на рис. 14.

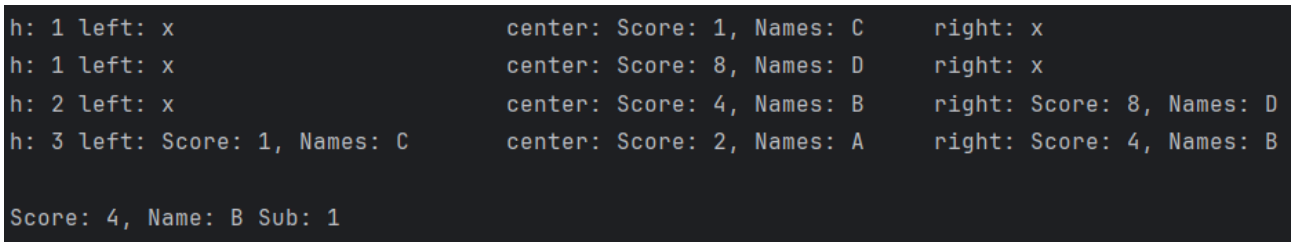

Рисунок 14 – Поиск элемента, не находящегося в дереве, находящегося ровно между двумя элементами

3.3. Вывод элементов с результатом, больше заданного.

Следующие тесты будут проводится на дереве, показанном на рис. 15.

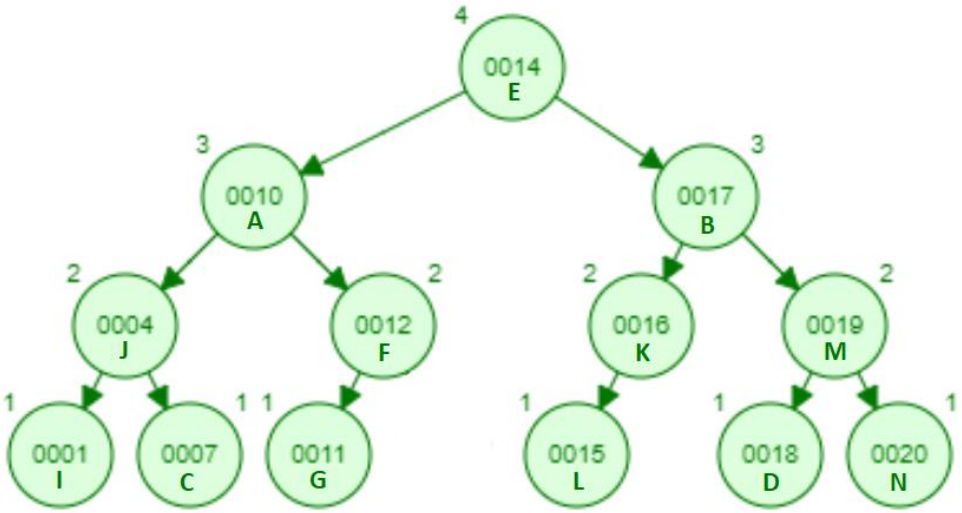

Рисунок 15 – АВЛ-дерево для тестов

Вывод значений, больших 16 показан на рис. 16.

```
Score: 17, Name: B
Score: 18, Name: D
Score: 19, Name: M
Score: 20, Name: N
```

Рисунок 16 – Значения, большие 16

Вывод значений, больших 6 показан на рис. 17.

```
Score: 7, Name: C
Score: 10, Name: A
Score: 11, Name: G
Score: 12, Name: F
Score: 14, Name: E
Score: 15, Name: L
Score: 16, Name: K
Score: 17, Name: B
Score: 18, Name: D
Score: 19, Name: M
Score: 20, Name: N
```

Рисунок 17 – Значения, большие 6

В следующем тесте будет использоваться дерево, изображенное на рис. 18.

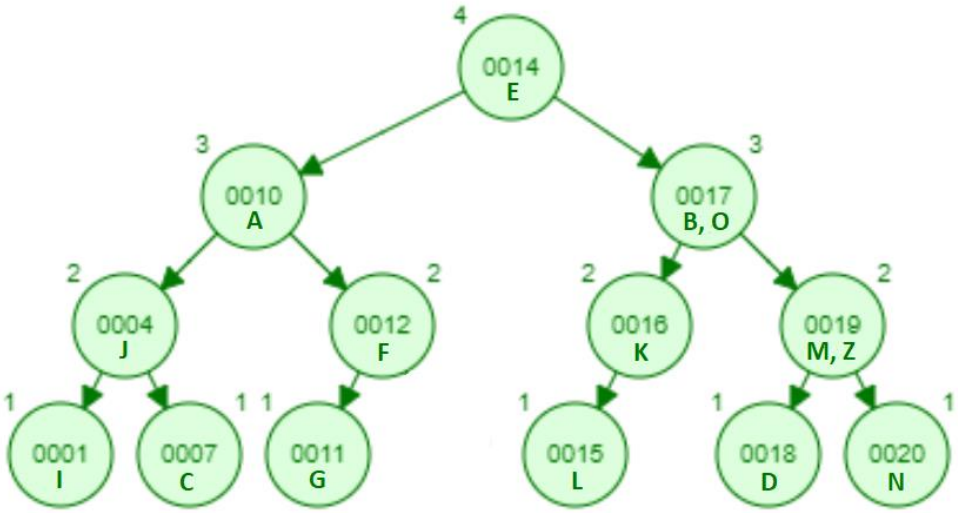

Рисунок 18 – Дерево для тестов

Вывод значений, больших 14 показан на рис. 19.

```
Score: 15, Name: L
Score: 16, Name: K
Score: 17, Name: B
Score: 17, Name: O
Score: 18, Name: D
Score: 19, Name: M
Score: 19, Name: Z
Score: 20, Name: N
```

Рисунок 19 – Значения, большие 14

Время вставки n-ого количества элементов представлено на рис. 20.

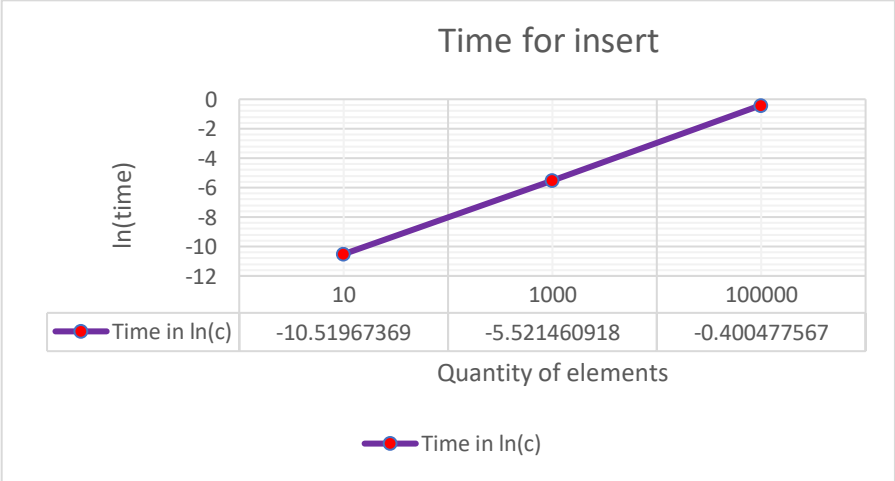

Рисунок 20 – Время вставки

Время поиска элементов в разных частях данных на количестве данных, равных 10 элементам представлено на рис. 21.

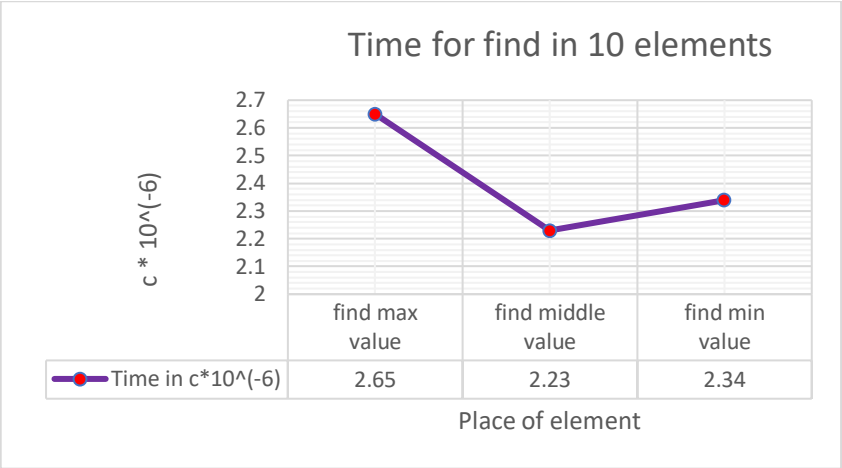

Рисунок 21 – Время поиска на объеме данных в 10 элементов

Время поиска элементов в разных частях данных на количестве данных, равных 1000 элементам представлено на рис. 22.

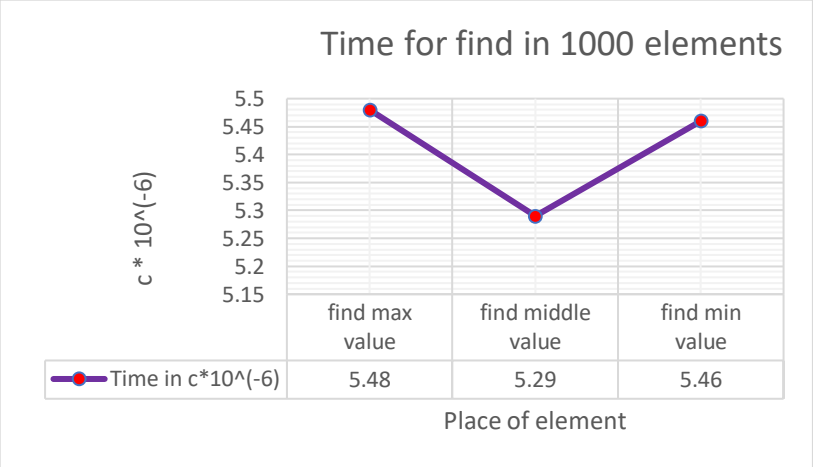

Рисунок 22 – Время поиска на объеме данных в 1000 элементов

Время поиска элементов в разных частях данных на количестве данных, равных 100000 элементам представлено на рис. 23.

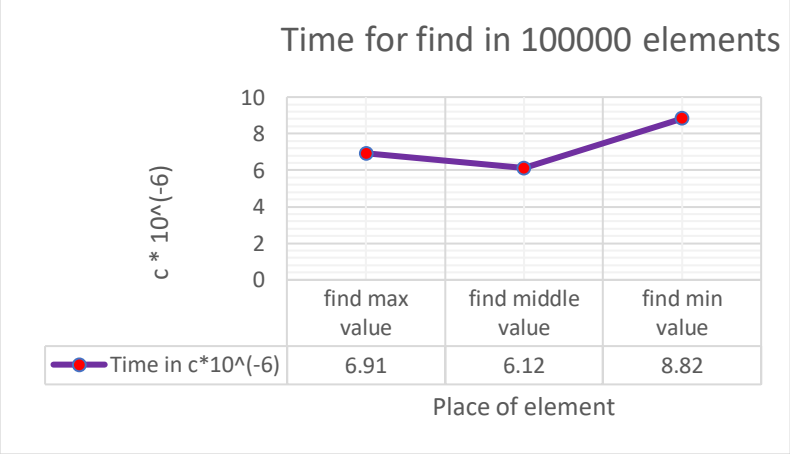

Рисунок 23 – Время поиска на объеме данных в 100000 элементов

ЗАКЛЮЧЕНИЕ

В ходе работы проведено исследование, в результате которого выяснено, что для заданной ситуации лучшим вариантом структуры данных будет АВЛ-дерево. Кроме того, данная структура данных была реализована на языке Python.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Базовые сведения о структуре АВЛ-дерева // URL: https://blog.skillfactory.ru/glossary/avl-derevo/

2. Электронный сервис для визуализации АВЛ-дерева // URL: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

ПРИЛОЖЕНИЕ А.
КОД

Название файла: ETurbo.py

```
from typing import Union
from random import randint
from itertools import product
import time

MAX_SUB = 100000
N_MAX = 100000

class Score:
    def __init__(self, score: int, names: list, right = None, left =
None):
        self.score: int = score
        self.names: list = names
        self.right: Union[Score, None] = right
        self.left: Union[Score, None] = left
        self.height: int = 1

    def __str__(self):
        out = ''
        for name in self.names:
            out += f'Score: {self.score}, Name: {name}\n'
        return out[:-1]
    # def __str__(self):
    #       return  f'Score: {self.score}, Names: {', '.join(map(str,
self.names))}'

def left_rotate(root_score: Union[Score, None]) -> Score:
    new_root = root_score.right
    temp = new_root.left
    new_root.left = root_score
    root_score.right = temp

    root_score.height = calc_height(root_score)
    new_root.height = calc_height(new_root)
    return new_root

def right_rotate(root_score: Union[Score, None]) -> Score:
    new_root = root_score.left
    temp_root = new_root.right
    new_root.right = root_score
    root_score.left = temp_root

    root_score.height = calc_height(root_score)
    new_root.height = calc_height(new_root)
    return new_root

def height(root_score: Union[Score, None]) -> int:
    if root_score is None:
        return 0
    return root_score.height

def calc_height(root_score: Union[Score, None]) -> int:
    return 1 + max(height(root_score.right), height(root_score.left))
```

```
def check_balance(root_score: Union[Score, None]) -> int:
    return height(root_score.right) - height(root_score.left)

def check_right_bigger(root_score: Union[Score, None]) -> bool:
    if height(root_score.right) >= height(root_score.left):
        return True
    return False

def check_left_bigger(root_score: Union[Score, None]) -> bool:
    if height(root_score.right) <= height(root_score.left):
        return True
    return False

def insert(new_score: int, new_name: str, root_score: Union[Score,
None]) -> Score:
    if root_score is None:
        return Score(new_score, [new_name])
    if root_score.score > new_score:
        root_score.left = insert(new_score, new_name, root_score.left)
    elif root_score.score < new_score:
        root_score.right = insert(new_score, new_name,
root_score.right)
    else:
        root_score.names.append(new_name)

    root_score.height = calc_height(root_score)
    balance_flag = check_balance(root_score)

    if balance_flag == 2:
        if check_right_bigger(root_score.right):
            root_score = left_rotate(root_score)
        else:
            root_score.right = right_rotate(root_score.right)
            root_score = left_rotate(root_score)
    elif balance_flag == -2:
        if check_left_bigger(root_score.left):
            root_score = right_rotate(root_score)
        else:
            root_score.left = left_rotate(root_score.left)
            root_score = right_rotate(root_score)

    return root_score

def find(root_score: Union[Score, None], find_score: int, min_sub =
MAX_SUB) -> (Score, int):
    if root_score.score == find_score:
        return root_score, 0

    current_sub = root_score.score - find_score                # check
current
    if abs(current_sub) <= abs(min_sub):
        min_sub = current_sub

    if root_score.score > find_score:
        if root_score.left is None:
            return root_score, min_sub
```

```
        left_root, left_sub = find(root_score.left, find_score, min_sub)
#check left
        current_sub = left_root.score - find_score
        if abs(current_sub) <= abs(left_sub):
            return left_root, current_sub

        return root_score, min_sub

    if root_score.score < find_score:
        if root_score.right is None:
            return root_score, min_sub

        right_root,  right_sub  =  find(root_score.right,  find_score,
min_sub)  # check right
        current_sub = right_root.score - find_score
        if abs(current_sub) <= abs(right_sub):
            return right_root, current_sub

        return root_score, min_sub

def print_biggest(root_score: Union[Score, None], min_score: int) ->
None:
    if root_score.left and root_score.score > min_score:
        print_biggest(root_score.left, min_score)
    if root_score.score > min_score:
        print(root_score)
    if root_score.right:
        print_biggest(root_score.right, min_score)

def post_order(root_score: Union[Score, None]) -> None:
    left, right = 'left: x', 'right: x'
    center = 'center: ' + str(root_score)
    if root_score.left:
        left = left[:-1] + str(root_score.left)
        post_order(root_score.left)
    if root_score.right:
        right = right[:-1] + str(root_score.right)
        post_order(root_score.right)
    print(f'h: {root_score.height}   {left} {center} {right}')

root_score = None
for i in range(N_MAX):
    score = randint(1, N_MAX)
    val = randint(1,4)
    name  =  ''.join(map(str,  list(product('ABCD',  repeat  =  randint(1,
val)))[randint(0, val - 1)]))
    root_score = insert(score, name, root_score)

class game:
    def __init__(self, root_score: Union[Score, None], current_score:
int = 0):
        self.__root_score: Union[Score, None] = root_score
        self.__current_score: int = current_score

    def start_game(self):
        self.__input_answer()
```

```
    def __random_value(self):
        self.__current_score = randint(1, N_MAX)

    def __input_value(self):
        value = input('- Please enter your score: ')
        flag = True
        for char in value:
            if not char.isdigit():
                flag = False
                break
        if flag:
            self.__current_score = int(value)
        else:
            print('Incorrect input. Please try again.')
            self.__input_value()

    def __new_game(self):
        print('Start game!')
        for i in range(3):
            time.sleep(0.8)
            print('.', end = '')
        print('\nGame over!')
        self.__input_value()
        print(f'You have scored {self.__current_score} points')
        start = time.time()
        near_score, sub = find(self.__root_score, self.__current_score)
        end = time.time()
        print(f'----------[{start-end} c.]--------------')
        if sub == 0:
            print('The same result for the players:')
        else:
            print('Wow! There was no such result before')
            print('The nearest result for the players:')
        print(near_score)
        self.__input_answer(False)

    def __show_biggest(self):
        print('A list of results larger than yours:')
        print_biggest(self.__root_score, self.__current_score)

    def __add_result(self):
        name = input('Enter your nickname: ')
        self.__root_score    =    insert(self.__current_score,    name,
self.__root_score)
        print('Your result has been added to the list')

    def __input_answer(self, is_first = True):
        if not is_first:
            print('- To see all the results above yours, enter [y]\n-
To add your result, enter [s]')
        print('- To create the game, enter [c]\n- To turn off the
electricity, enter [q]')
        answer = input()
        if answer == 'c':
            self.__new_game()
        elif answer == 'y' and not is_first:
```

```
            self.__show_biggest()
            self.__input_answer(False)
        elif answer == 's' and not is_first:
            self.__add_result()
            self.__input_answer(True)
        elif answer != 'q':
            print('Incorrect answer. Please try again')
            self.__input_answer(is_first)

new_game = game(root_score)
new_game.start_game()
```