**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: Исследование структур данных

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Еникеев А.А. |
| Преподаватель |  | Шалагинов И.В. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент: Еникеев А.А. | | |
| Группа: 3388 | | |
| Тема работы: Исследование структур данных | | |
| Исходные данные:  Требуется определить, какая структура данных подходит в контексте задания, реализовать выбранную структуру, написать код для визуализации структуры. | | |
| Разделы пояснительной записки: «Содержание», «Введение», «Формальная постановка задачи», «Выполнение работы», «Исследование», «Тестирование», «Заключение». | | |
| Предполагаемый объем пояснительной записки:  Не менее 15 страниц. | | |
| Дата выдачи задания: 06.11.2024 | | |
| Дата сдачи реферата: 08.12.2024 | | |
| Дата защиты реферата: 10.12.2024 | | |
| Студент гр. 3388 |  | Еникеев А.А. |
| Преподаватель |  | Шалагинов И.В. |

**Аннотация**

Данная курсовая работа посвящена исследованию и реализации структуры данных, подходящей для задачи проверки вершин в алгоритме Дейкстры. В рамках работы проведён анализ различных структур данных с точки зрения их временных характеристик, что позволило обосновать выбор оптимального решения для конкретной задачи. Основное внимание уделено обеспечению высокой скорости операций добавления и проверки элементов, что является критически важным для эффективной работы алгоритма Дейкстры.

**Summary**

This course work is devoted to the study and implementation of a data structure suitable for the task of checking vertices in Dijkstra's algorithm. As part of the work, an analysis of various data structures was carried out in terms of their temporal characteristics, which made it possible to justify the choice of the optimal solution for a specific task. The main attention is paid to ensuring high speed of operations for adding and verifying elements, which is critically important for the effective operation of Dijkstra's algorithm.

**Содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Формальная постановка задачи | 6 |
| 2. | Выполнение работы | 7 |
| 2.1 | Класс Node | 7 |
| 2.2 | Класс AVLTree | 7 |
| 2.3 | Класс HashTable | 8 |
| 2.4 | Класс Node для графа | 9 |
| 2.5 | Класс Graph | 10 |
| 2.6 | Описание алгоритма Дейкстры | 11 |
| 2.7 | Вспомогательные функции | 12 |
| 3. | Исследование | 14 |
| 3.1 | Задачи, которые должна решать структура данных | 14 |
| 3.2 | Особенности алгоритма Дейкстры | 14 |
| 3.3 | Выбранная структура данных | 15 |
| 3.4 | Алгоритм хэширования данных | 16 |
| 3.5 | Вычислительная сложность операций | 17 |
| 4. | Тестирование | 18 |

**введение**

Целью курсовой работы является исследование различных структур данных для решения задачи. Будет проведен анализ и выбрана подходящая структура данных. Помимо теоретического анализа, будет реализована выбранная структура с учетом особенностей задачи. Также будет разработан метод для визуализации полученной структуры.

**1. ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ**

## Вариант 1

Один из алгоритмов поиска кратчайших путей в графе - алгоритм Дейкстры. Основная его идея - улучшение пути через улучшающие вершины. Оно происходит при просмотре вершины и посещении смежных с нею не просмотренных вершин.

Вершина состоит из 3х полей: название, оценка длины пути до нее, предыдущая вершина.

При просмотре вершин они проверяются на то, были ли уже просмотрены до этого. А при рассмотрении новой вершины, она добавляется в структуру данных.

Для данной задачи скорость проверки и добавления вершины превалирует над затратами по памяти.

Решите, какая структура данных лучше подходит для решения задачи проверки вершины на то, была ли она уже просмотрена. Реализуйте выбранную структуру данных.

**2. Выполнение работы**

**2.1. Класс Node**

* Представляет узел AVL-дерева.
* Атрибуты:
  + val: строка, значение узла.
  + left и right: ссылки на левого и правого потомков соответственно.
  + height: высота узла, используется для вычисления балансировочного фактора.
* Конструктор позволяет задать значение узла, а также потомков (по умолчанию None).

**2.2. Класс AVLTree**

* Реализует AVL-дерево — самобалансирующееся двоичное дерево поиска.
* Атрибуты:
  + root: ссылка на корневой узел дерева.
  + Методы для работы с высотой и балансировкой:
  + get\_height(node): Возвращает высоту узла или 0, если узел отсутствует.
* get\_balance(node): Вычисляет балансировочный фактор узла.
* update\_height(node): Обновляет высоту узла на основе высот потомков.
* rotate\_left(x) и rotate\_right(y): Выполняют левый или правый поворот дерева для балансировки.
* balance(node): Балансирует поддерево, применяя соответствующие вращения.
* Методы вставки и удаления:
* insert(val): Вставляет новое значение в дерево.
* \_insert(node, val): Рекурсивная реализация вставки.
* delete(val): Удаляет значение из дерева.
* \_delete(node, key): Рекурсивная реализация удаления.
* \_get\_min\_value\_node(node): Находит узел с минимальным значением в поддереве.
* Обход и поиск:
  + in\_order\_traversal(): Возвращает список значений дерева в порядке возрастания.
  + \_in\_order\_traversal(node, result): Рекурсивный обход дерева.
  + \_\_contains\_\_(val): Проверяет, содержится ли значение в дереве.
  + \_contains(node, val): Рекурсивная реализация поиска.
* Служебные методы:
  + \_\_repr\_\_(): Возвращает строковое представление дерева в виде отсортированного списка.
  + get\_root(): Возвращает корень дерева.

## 2.3. Класс HashTable

* Реализует хэш-таблицу с использованием AVL-деревьев для разрешения коллизий.
* Основное предназначение: обеспечение эффективных операций вставки, удаления и проверки наличия элементов.
* Атрибуты:
* size: Размер таблицы (всегда простое число для уменьшения количества коллизий).
* Вычисляется с использованием функции find\_next\_prime для обеспечения устойчивого распределения хэшей.
* table: Список из объектов AVLTree или None, представляющий слоты хэш-таблицы.
* count: Количество элементов в таблице.
* load\_factor: Коэффициент загрузки, определяющий необходимость увеличения размера таблицы (по умолчанию 0.69)
* Работа с элементами:
  + add(key: str): Добавляет ключ в таблицу.
  + remove(key: str): Удаляет ключ из таблицы.
  + \_\_contains\_\_(key: str): Возвращает True, если ключ существует в таблице, иначе False.
* Изменение размера таблицы:
  + \_resize(): Увеличивает размер таблицы (до следующего простого числа, большего вдвое от текущего).
* Служебные методы
  + \_\_str\_\_(): Возвращает строковое представление таблицы.
  + remove\_all(): Удаляет все элементы из таблицы, сбрасывая её до начального состояния.

**2.4. Класс Node для графа**

* Класс представляет узел графа, необходимый для реализации алгоритма Дейкстры и других алгоритмов поиска путей.
* Атрибуты:
  + - * name: Имя узла (строковое значение), уникально идентифицирующее узел.
  + distance: Текущее расстояние от начального узла (по умолчанию бесконечность). Используется для оценки кратчайшего пути.
  + Previous: Ссылка на предыдущий узел в кратчайшем пути. Обновляется в процессе работы алгоритма Дейкстры.
* Методы:
  + \_\_lt\_\_(other: "Node"): Реализует операцию сравнения узлов по их расстоянию. Это позволяет использовать узлы в приоритетных очередях, где узлы с меньшими расстояниями обрабатываются первыми.

**2.5. Класс Graph**

* Класс реализует граф в виде списка смежности. Подходит для работы с алгоритмами на графах, включая алгоритм Дейкстры.
* Атрибуты:
  + - * adjacency\_list: Словарь, где ключи — это имена узлов (строки), а значения — словари, представляющие смежные узлы и веса рёбер. Формат: {узел: {сосед: вес, ...}, ...}.
* Методы:
* Добавление узлов:
  + - * add\_node(node\_name: str): Добавляет узел в граф, если его ещё нет.
* Добавление рёбер:
  + - * add\_edge(from\_node: str, to\_node: str, weight: float): Добавляет направленное ребро из одного узла в другой с заданным весом.

**2.6. Описание алгоритма Дейкстры**

* Алгоритм Дейкстры предназначен для поиска кратчайших путей от одного узла графа до всех остальных. Данный код реализует этот алгоритм, используя приоритетную очередь и хэш-таблицу для отслеживания посещённых узлов.
* Функция dijkstra(graph: Graph, start\_node\_name: str)
* Входные параметры:
  + - * graph — Граф, представленный объектом класса Graph. Содержит список смежности.
      * start\_node\_name — Имя начального узла, от которого начинается поиск кратчайших путей.
* Возвращаемое значение:
  + - * Словарь, где ключи — имена узлов, а значения — минимальные расстояния от начального узла до каждого из них.
* Алгоритм работы:

1. Инициализация:
   * + - Для каждого узла графа создаётся объект класса Node с бесконечным значением расстояния.
       - Расстояние начального узла (start\_node\_name) устанавливается в 0.
       - Используется хэш-таблица visited (объект класса HashTable) для хранения уже посещённых узлов.
       - Приоритетная очередь (priority\_queue) инициализируется начальным узлом с расстоянием 0.
2. Основной цикл:
   * + - Из приоритетной очереди извлекается узел с минимальным расстоянием (current\_node).
       - Если узел уже посещён (находится в visited), он пропускается.
       - Узел помечается как посещённый, добавляя его в хэш-таблицу visited.
3. Обновление расстояний:
   * + - Для каждого соседнего узла neighbor проверяется, не является ли он посещённым.
       - Рассчитывается новое расстояние до соседа как сумма расстояния до текущего узла и веса ребра между ними.
       - Если новое расстояние меньше текущего сохранённого значения, оно обновляется, а сосед добавляется в очередь с приоритетом.
4. Возврат результата:
   * + - После завершения обработки всех узлов функция возвращает словарь с кратчайшими расстояниями до всех узлов.

**2.7. Вспомогательные функции**

1. Проверка простого числа

* is\_prime(num: int): Функция проверяет, является ли заданное число простым.

1. Поиск следующего простого числа

* find\_next\_prime(start: int): Находит следующее простое число, начиная с заданного значения start. Использует заранее подготовленный список простых чисел \_\_primes для ускорения поиска. Используется для поиска размера хэш-таблицы при балансировке.

1. Визуализация хэш-таблицы

* visualize(ht, filename="hash\_table"): Генерирует графическое представление хэш-таблицы, включая содержимое её ячеек (AVL-деревьев). Использует библиотеку graphviz для создания диаграммы.

Пример визуализации хэш-таблицы см. в приложении A.

Исходный код программы см. в приложении B.

**3. Исследование**

* 1. **3.1. Задачи, которые должна решать структура данных**

## Для алгоритма Дейкстры структура данных, хранящая посещённые узлы, должна:

1. Эффективно проверять наличие элемента (узла) в множестве.
2. Эффективно добавлять новый элемент (посещение узла).
3. Гарантировать уникальность элементов.

В контексте задания эффективностью является скорость выполнения каждой из операций, память не имеет значения.

* 1. **3.2. Особенности алгоритма Дейкстры**

3.2.1. Посещённые вершины:

* Каждая вершина графа добавляется в множество только один раз.
* Это означает, что общее количество вставок в структуру данных равно **|V|**, где **|V|** - число вершин в графе.

3.2.2. Проверки на вхождение:

* Для каждой вершины алгоритм проверяет, посещена ли она (при обработке соседей текущей вершины).
* Каждое ребро **(v, u)** графа вызывает проверку, посещена ли вершина **u**. Таким образом, количество проверок пропорционально количеству рёбер **|E|**.

3.2.3. Оценим сравнительную частоту вставок и проверок:

Для оценки сравнительных частот вставок и проверок нужно рассмотреть отношение количества операций:

* Число вставок равно **|V|** (вершины)
* Число проверок пропорционально **|E|** (ребра)

Для связанных графов верно:

* **|E|** ≥ **|V|** - 1 (нижняя граница числа ребер)
* **|E|** ≤ **|V|2**  (верхняя граница числа ребер)
* Это означает, что в среднем количество проверок на вхождение значительно превосходит количество вставок:

Частота проверок = O(**|E|**), частота вставок = O(**|V|**) ⇒ **|E|** ≫**|V|** в реальных графах.

**3.3. Выбранная структура данных**

Комбинация HashTable + AVL-деревья в бакетах:

3.3.1. Эффективность проверки на вхождение:

* В среднем хэш-таблица обеспечивает O(n) для поиска бакета, где n — длина строки (хэш-таблица работает со строками).
* Внутри бакета элементы хранятся в виде AVL-дерева, где проверка выполняется за O(n\*logk), где k — число элементов в бакете, n — длина строки. Для хорошо сбалансированной хэш-таблицы количество коллизий невелико (стремиться к 0), поэтому итоговая сложность проверки —

O(n+ n\*logk) = O(n)

3.3.2. Эффективность вставки:

* Вставка выполняется аналогично проверке: O(n) в среднем и

O(n + n\*logk) в худшем случае. Поскольку число вставок в алгоритме Дейкстры невелико (**|V|** раз), этот метод решения коллизий хэш-таблицы (AVL-деревья в бакетах) оправдан и эффективен.

3.3.3. Оптимизация под характер работы Дейкстры:

* В алгоритме Дейкстры проверки на вхождение (O(**|E|**)) выполняются значительно чаще, чем вставки (O(**|V|**), подробнее п. 3.2.). Комбинация хэширования и AVL-деревьев минимизирует затраты на проверки, даже если бакеты становятся большими (из-за коллизий).

3.3.4. Обеспечение уникальности вершин:

* Для обеспечения корректной работы алгоритма Дейкстры вершины должны иметь уникальный идентификатор. В контексте задания таким идентификатором являются названия — строки.
* В реализованной структуре данных не поддерживается дублирование элементов.

**3.4. Алгоритм хэширования данных**

3.4.1. Требования к хэш-функции в контексте задачи:

* Обеспечивать равномерное распределение элементов по бакетам, чтобы минимизировать коллизии и ускорить операции (поиск, вставка).
* Быть быстрой, так как хэширование выполняется многократно (для каждой проверки или вставки).
* Обрабатывать строковые идентификаторы (названия вершин) эффективно.
* Иметь низкую вероятность коллизий, чтобы избежать снижения производительности AVL-деревьев внутри бакетов.

3.4.2. Почему **FNV-1a** соответствует этим требованиям:

* Простота и скорость: побитовые XOR и умножение на фиксированный простое число (16777619 для 32-битной версии). Это делает его очень быстрым, особенно при работе с короткими строками, что характерно для идентификаторов вершин.
* Хорошее распределение: FNV-1a обеспечивает равномерное распределение хэшей даже для похожих строк (например, "A" и "B"). Это важно для минимизации коллизий. Поскольку названия вершин обычно короткие и отличаются в малом числе символов, FNV-1a хорошо справляется с их хэшированием.
* Низкая вероятность коллизий: FNV-1a создаёт уникальные хэши для большинства входных данных, особенно строк. Хотя это не криптографическая хэш-функция, для задачи распределения данных по бакетам её устойчивости достаточно.
* Поддержка строковых данных: FNV-1a обрабатывает данные посимвольно, что делает его эффективным для строковых идентификаторов (названий вершин).
* Сложность алгоритма FNV-1a равна O(n), где n — длина строки.

**3.5. Вычислительная сложность операций**

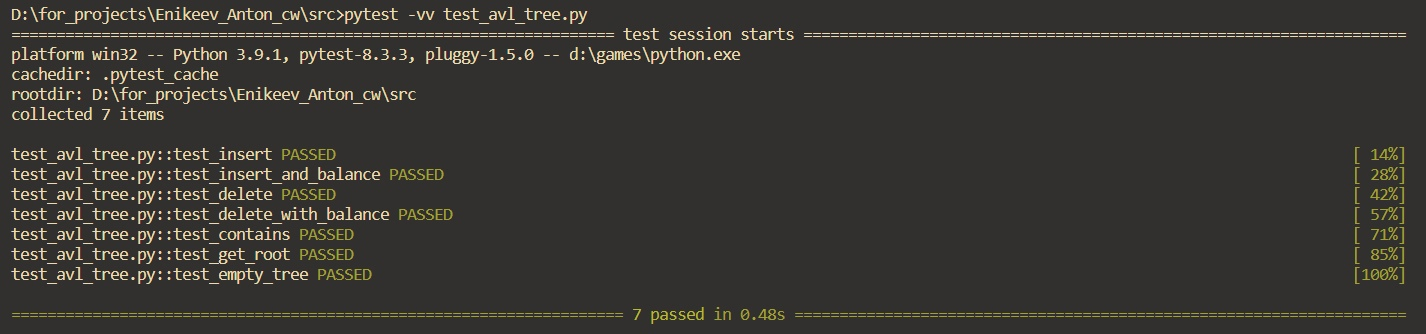
* FNV-1a имеет сложность O(n), где n — длина строки.
* Вставка и проверка вхождения в AVL-дерево: требует O(n\*logk), где k — число элементов в бакете.

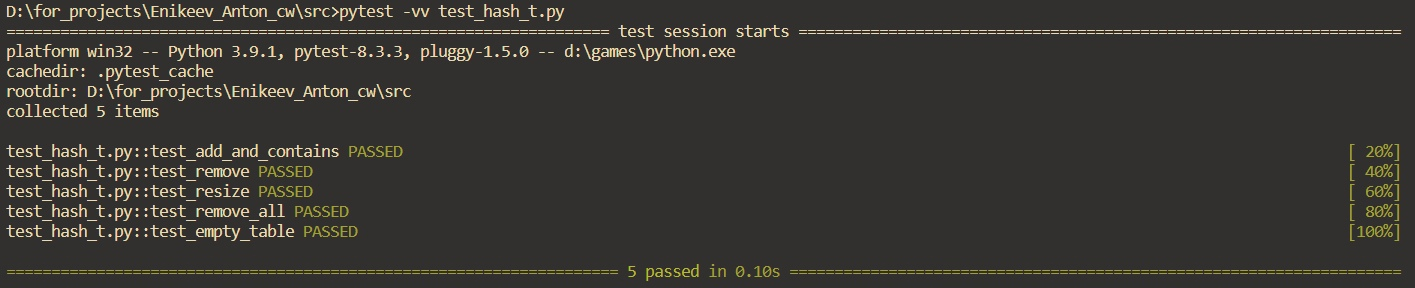
3.5.1. Итак, итоговая сложность вставки и поиска O(n + n\*logk), но на практике для 80% значений сложность составит O(n) из-за хорошего распределения, соответственно чем короче строка, тем быстрее будет выполнена операция. В худшем случае (если все строки попадают в один бакет), сложность может вырасти до O(n + n\*m\*logm), где m — общее количество строк в коллекции.

**4. ТЕСТИРОВАНИЕ**

## 4.1. Результаты тестирования класса AVLTree представлены на рис. 1.

4.2. Результаты тестирования класса HashTable представлены на рис. 2.

Рисунок 1 - Результаты тестирования класса AVLTree

Рисунок 2 - Результаты тестирования класса HashTable

**ЗАКЛЮЧЕНИЕ**

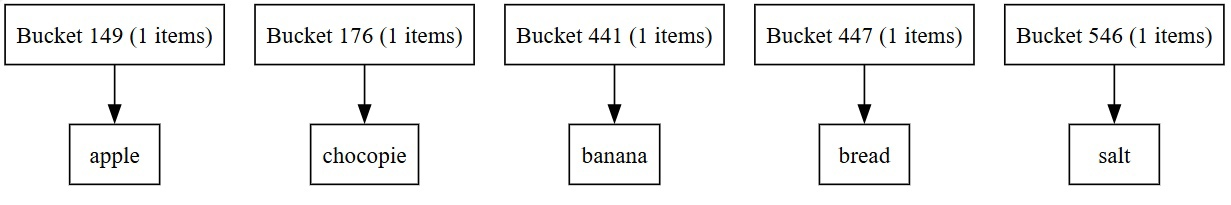
## В ходе выполнения курсовой работы была разработана и реализована структура данных на основе хэш-таблицы с использованием AVL-деревьев для разрешения коллизий. Для оптимизации работы таблицы была применена концепция динамического изменения размера, при которой размер таблицы всегда оставался простым числом. Это обеспечивало минимизацию числа коллизий и повышение эффективности распределения ключей.

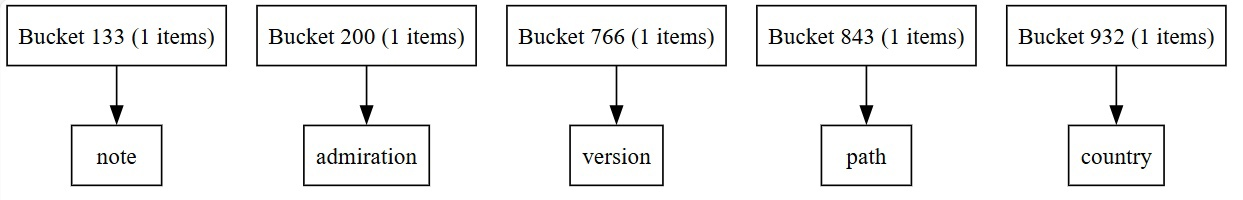
Алгоритм Дейкстры, интегрированный с разработанными структурами данных, позволил на практике оценить производительность хэш-таблицы и AVL-деревьев при решении задач поиска кратчайших путей. В рамках реализации были учтены ключевые аспекты, такие как обработка графов с произвольной структурой, предотвращение избыточного посещения узлов и корректное обновление приоритетов.

Дополнительно были разработаны вспомогательные функции для проверки простоты чисел, поиска ближайшего простого числа и визуализации структуры хэш-таблицы и AVL-деревьев. Визуализация позволила проанализировать распределение данных и структуру дерева, что подтвердило корректность работы алгоритмов и структуры данных.

# Приложение А ВИЗУАЛИЗАЦИЯ ХЭШ-ТАБЛИЦЫ

Примеры визуализации хэш-таблицы см. на рис. 3 и 4.

Рисунок 3 - Визуалзация хэш-таблицы (Пример 1)

Рисунок 4 - Визуализауия хэш-таблицы (Пример 2)

# Приложение B Исходный код программы

**Название файла: avl\_tree.py**

from typing import Optional, List

class Node:

def \_\_init\_\_(self, val: str, left: Optional["Node"] = None, right: Optional["Node"] = None):

"""

Узел дерева AVL.

:param val: Значение узла.

:param left: Левый потомок.

:param right: Правый потомок.

"""

self.val = val

self.left: Optional["Node"] = left

self.right: Optional["Node"] = right

self.height: int = 1 # Высота узла

class AVLTree:

def \_\_init\_\_(self) -> None:

"""

Инициализация пустого AVL-дерева.

"""

self.root: Optional[Node] = None

def get\_height(self, node: Optional[Node]) -> int:

"""

Получает высоту узла.

:param node: Узел.

:return: Высота узла или 0, если узел пустой.

"""

return 0 if node is None else node.height

def rotate\_right(self, y: Node) -> Node:

"""

Правый поворот дерева вокруг узла y.

:param y: Узел, вокруг которого выполняется поворот.

:return: Новый корень после поворота.

"""

x = y.left

T2 = x.right

# Выполняем вращение

x.right = y

y.left = T2

# Обновляем высоты

self.update\_height(y)

self.update\_height(x)

return x

def rotate\_left(self, x: Node) -> Node:

"""

Левый поворот дерева вокруг узла x.

:param x: Узел, вокруг которого выполняется поворот.

:return: Новый корень после поворота.

"""

y = x.right

T2 = y.left

# Выполняем вращение

y.left = x

x.right = T2

# Обновляем высоты

self.update\_height(x)

self.update\_height(y)

return y

def get\_balance(self, node: Optional[Node]) -> int:

"""

Вычисляет балансировочный фактор узла.

:param node: Узел.

:return: Балансировочный фактор (разница высот левого и правого поддерева).

"""

return 0 if node is None else self.get\_height(node.left) - self.get\_height(node.right)

def update\_height(self, node: Optional[Node]) -> None:

"""

Обновляет высоту узла.

:param node: Узел для обновления.

"""

if node:

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

def balance(self, node: Optional[Node]) -> Optional[Node]:

"""

Балансирует поддерево с корнем в заданном узле.

:param node: Узел.

:return: Сбалансированный узел.

"""

if node is None:

return node

balance\_factor = self.get\_balance(node)

# Левый поворот

if balance\_factor > 1 and self.get\_balance(node.left) >= 0:

return self.rotate\_right(node)

# Правый поворот

if balance\_factor < -1 and self.get\_balance(node.right) <= 0:

return self.rotate\_left(node)

# Лево-правый случай

if balance\_factor > 1 and self.get\_balance(node.left) < 0:

node.left = self.rotate\_left(node.left)

return self.rotate\_right(node)

# Право-левый случай

if balance\_factor < -1 and self.get\_balance(node.right) > 0:

node.right = self.rotate\_right(node.right)

return self.rotate\_left(node)

return node

def insert(self, val: str) -> None:

"""

Вставляет значение в дерево.

:param val: Значение для вставки.

"""

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

def \_insert(self, node: Optional[Node], val: str) -> Node:

"""

Рекурсивно вставляет значение в поддерево.

:param node: Текущий узел.

:param val: Значение для вставки.

:return: Корень поддерева после вставки.

"""

if node is None:

return Node(val)

if val < node.val:

node.left = self.\_insert(node.left, val)

elif val > node.val:

node.right = self.\_insert(node.right, val)

else:

return node # Дубликаты не вставляются

self.update\_height(node)

return self.balance(node)

def delete(self, val: str) -> None:

"""

Удаляет значение из дерева.

:param val: Значение для удаления.

"""

self.root = self.\_delete(self.root, val)

def \_delete(self, node: Optional[Node], key: str) -> Optional[Node]:

"""

Рекурсивно удаляет значение из поддерева.

:param node: Текущий узел.

:param key: Значение для удаления.

:return: Корень поддерева после удаления.

"""

if node is None:

return None

if key < node.val:

node.left = self.\_delete(node.left, key)

elif key > node.val:

node.right = self.\_delete(node.right, key)

else:

# Узел с одним потомком или без потомков

if node.left is None:

return node.right

elif node.right is None:

return node.left

# Узел с двумя потомками: получаем наименьший узел в правом поддереве

temp = self.\_get\_min\_value\_node(node.right)

node.val = temp.val

node.right = self.\_delete(node.right, temp.val)

self.update\_height(node)

return self.balance(node)

def \_get\_min\_value\_node(self, node: Node) -> Node:

"""

Находит узел с минимальным значением.

:param node: Корень поддерева.

:return: Узел с минимальным значением.

"""

current = node

while current.left is not None:

current = current.left

return current

def in\_order\_traversal(self) -> List[str]:

"""

Возвращает список значений дерева в порядке возрастания.

:return: Список значений.

"""

result: List[str] = []

self.\_in\_order\_traversal(self.root, result)

return result

def \_in\_order\_traversal(self, node: Optional[Node], result: List[str]) -> None:

"""

Рекурсивно выполняет обход в порядке возрастания.

:param node: Текущий узел.

:param result: Список для добавления значений.

"""

if node:

self.\_in\_order\_traversal(node.left, result)

result.append(node.val)

self.\_in\_order\_traversal(node.right, result)

def \_\_repr\_\_(self) -> str:

"""

Возвращает строковое представление

"""

return repr(self.in\_order\_traversal())

def \_\_contains\_\_(self, val: str) -> bool:

"""

Возвращает bool: строка содержиться в дереве

"""

return self.\_contains(self.root, val)

def \_contains(self, node: Optional[Node], val: str) -> bool:

"""

Рекурсивно обходит дерево и сравнивает строки

"""

if node is None:

return False

if val == node.val:

return True

if val < node.val:

return self.\_contains(node.left, val)

return self.\_contains(node.right, val)

def get\_root(self) -> Optional[Node]:

"""

Возвращает корень дерева

"""

return self.root

**Название файла: hash\_t.py**

from typing import Optional, List

from .hash\_func import \_FNV\_1a

from .avl\_tree import AVLTree

from .find\_next\_prime import find\_next\_prime

class HashTable:

def \_\_init\_\_(self, initial\_size: int = 1024):

"""

Инициализация хэш-таблицы.

:param initial\_size: Начальный размер таблицы.

"""

self.size: int = find\_next\_prime(initial\_size) # Размер таблицы (простое число)

self.table: List[Optional[AVLTree]] = [None] \* self.size

self.count: int = 0 # Количество элементов

self.load\_factor: float = 0.69 # Коэффициент загрузки

def \_resize(self) -> None:

"""

Увеличивает размер таблицы при превышении коэффициента загрузки.

"""

old\_table = self.table

self.size = find\_next\_prime(self.size \* 2)

self.table = [None] \* self.size

self.count = 0

for tree in old\_table:

if tree is not None:

keys = tree.in\_order\_traversal()

for key in keys:

self.add(key)

def add(self, key: str) -> None:

"""

Вставляет элемент в таблицу.

:param key: Ключ.

"""

if self.count / self.size > self.load\_factor:

self.\_resize()

index = \_FNV\_1a(key, self.size)

if self.table[index] is None:

self.table[index] = AVLTree()

tree = self.table[index]

tree.insert(key)

self.count += 1

def remove(self, key: str) -> None:

"""

Удаляет элемент по ключу.

:param key: Ключ.

"""

index = \_FNV\_1a(key, self.size)

if self.table[index] is not None:

tree = self.table[index]

if key in tree:

tree.delete(key)

self.count -= 1

# Если дерево пустое, освобождаем слот

if not tree.in\_order\_traversal():

self.table[index] = None

return

raise KeyError(f"'{key}' not found in HashTable")

def \_\_contains\_\_(self, key: str) -> bool:

"""

Проверяет наличие ключа в таблице.

:param key: Ключ.

:return: True, если ключ существует, иначе False.

"""

index = \_FNV\_1a(key, self.size)

if self.table[index] is not None:

tree = self.table[index]

return key in tree

return False

def \_\_str\_\_(self) -> str:

"""

Возвращает строковое представление хэш-таблицы.

:return: Строка с содержимым таблицы.

"""

result = '{' + ', '.join(str(x) for x in self.table if x is not None) + '}'

return result

def remove\_all(self):

self.size: int = find\_next\_prime(1024) # Размер таблицы (простое число)

self.table: List[Optional[AVLTree]] = [None] \* self.size

self.count: int = 0 # Количество элементов

**Название файла: \_\_init\_\_.py**

from .user\_cli import user\_cli

from .dijkstra\_algorithm import dijkstra

from .graph import Graph

from .hash\_t import HashTable

from .visualization\_table import visualize

**Название файла: dijkstra\_algorithm.py**

import heapq

from typing import Dict

from .hash\_t import HashTable

from .graph import Node, Graph

def dijkstra(graph: Graph, start\_node\_name: str) -> Dict[str, float]:

"""

Алгоритм Дейкстры для поиска кратчайших путей.

:param graph: Граф (объект класса Graph).

:param start\_node\_name: Начальный узел.

:return: Словарь с кратчайшими расстояниями до всех узлов.

"""

# Инициализация узлов

nodes: Dict[str, Node] = {name: Node(name) for name in graph.adjacency\_list}

nodes[start\_node\_name].distance = 0

visited = HashTable() # Для хранения посещённых узлов

priority\_queue: list[Node] = []

heapq.heappush(priority\_queue, nodes[start\_node\_name])

while priority\_queue:

current\_node = heapq.heappop(priority\_queue)

if current\_node.name in visited:

continue # Если узел уже посещён, пропускаем его

visited.add(current\_node.name)

# Обновляем расстояния до смежных узлов

for neighbor\_name, weight in graph.adjacency\_list[current\_node.name].items():

neighbor = nodes[neighbor\_name]

if neighbor\_name not in visited:

new\_distance = current\_node.distance + weight

if new\_distance < neighbor.distance:

neighbor.distance = new\_distance

neighbor.previous = current\_node

heapq.heappush(priority\_queue, neighbor)

# Возвращаем кратчайшие расстояния

return {node.name: node.distance for node in nodes.values()}

**Название файла: find\_next\_prime.py**

from math import isqrt

from bisect import bisect\_left

\_\_primes = [

3, 7, 11, 17, 23, 29, 37, 47, 59, 71, 89, 107, 131, 163, 197, 239, 293, 353, 431, 521, 631, 761, 919,

1103, 1327, 1597, 1931, 2333, 2801, 3371, 4049, 4861, 5839, 7013, 8419, 10103, 12143, 14591,

17519, 21023, 25229, 30293, 36353, 43627, 52361, 62851, 75431, 90523, 108631, 130363, 156437,

187751, 225307, 270371, 324449, 389357, 467237, 560689, 672827, 807403, 968897, 1162687, 1395263,

1674319, 2009191, 2411033, 2893249, 3471899, 4166287, 4999559, 5999471, 7199369, 12582917, 25165843, 50331653, 100663319, 201326611,

402653189, 805306457, 1610612741

]

def is\_prime(num: int) -> bool:

"""

Проверяет, является ли число простым.

:param num: Число для проверки.

:return: True, если число простое, иначе False.

"""

if num < 2:

return False

if num in (2, 3):

return True

if num % 2 == 0 or num % 3 == 0:

return False

limit = isqrt(num)

for i in range(5, limit + 1, 6):

if num % i == 0 or num % (i + 2) == 0:

return False

return True

def find\_next\_prime(start: int) -> int:

"""

Находит следующее простое число больше или равное start.

:param start: Начальное число.

:return: Следующее простое число.

"""

# Используем бинарный поиск для нахождения индекса

index = bisect\_left(\_\_primes, start)

# Если индекс в пределах списка, возвращаем найденное простое число

if index < len(\_\_primes):

return \_\_primes[index]

if start <= 2:

return 2

candidate = start if start % 2 != 0 else start + 1

while not is\_prime(candidate):

candidate += 2

return candidate

**Название файла: graph.py**

from typing import Optional, Dict

class Node:

def \_\_init\_\_(self, name: str):

"""

Узел графа.

:param name: Название узла.

"""

self.name: str = name

self.distance: float = float('inf') # Оценка длины пути (изначально бесконечность)

self.previous: Optional[Node] = None # Предыдущий узел на кратчайшем пути

def \_\_lt\_\_(self, other: "Node") -> bool:

"""

Сравнение узлов по оценке расстояния (для работы с приоритетной очередью).

:param other: Другой узел для сравнения.

:return: True, если расстояние текущего узла меньше.

"""

return self.distance < other.distance

class Graph:

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

"""

Граф, представленный в виде списка смежности.

"""

self.adjacency\_list: Dict[str, Dict[str, float]] = {}

def add\_node(self, node\_name: str) -> None:

"""

Добавить узел в граф.

:param node\_name: Название узла.

"""

if node\_name not in self.adjacency\_list:

self.adjacency\_list[node\_name] = {}

def add\_edge(self, from\_node: str, to\_node: str, weight: float) -> None:

"""

Добавить ребро в граф.

:param from\_node: Узел начала.

:param to\_node: Узел конца.

:param weight: Вес ребра.

"""

self.add\_node(from\_node)

self.add\_node(to\_node)

self.adjacency\_list[from\_node][to\_node] = weight

**Название файла: hash\_func.py**

def \_FNV\_1a(key, limit):

"""

Хэш-функция (FNV-1a)

"""

fnv\_prime = 16777619

hash\_value = 2166136261

for char in key:

hash\_value ^= ord(char) # XOR со значением символа

hash\_value = (hash\_value \* fnv\_prime) % limit

return hash\_value

**Название файла: user\_cli.py**

import sys

from .dijkstra\_algorithm import dijkstra

from .graph import Graph

from .hash\_t import HashTable

from .visualization\_table import visualize

def print\_help():

print("Программа для работы с алгоритмом Дейкстры")

print("Команды для работы с графом:")

print(" add\_vertex <name> - добавить вершину")

print(" add\_edge <from> <to> <distance> - добавить ребро с расстоянием")

print(" shortest\_path <from> <to> - найти кратчайший путь между двумя вершинами")

print(" all\_shortest\_paths - вывести все вершины и кратчайшие расстояния между ними")

print("Команды для работы с хеш-таблицей:")

print(" add <key> - добавить значение")

print(" find <key> - проверить наличие значения")

print(" delete <key> - удалить значение")

print(" visualize <filename> - сохранить визуализацию хеш-таблицы в файл")

print(" remove\_all - удалить все элементы хэш-таблицы")

print(" print\_table - вывести таблицу")

print("Общие команды:")

print(" help - вывести справку")

print(" exit - выйти из программы")

def user\_cli():

graph = Graph()

ht = HashTable()

print\_help()

while True:

command = input("Введите команду: ").strip().split()

if not command:

continue

if command[0] == "add\_vertex":

if len(command) != 2:

print("Ошибка: неверное количество аргументов.")

continue

vertex = command[1]

graph.add\_node(vertex)

elif command[0] == "add\_edge":

if len(command) != 4:

print("Ошибка: неверное количество аргументов.")

continue

from\_vertex, to\_vertex, distance = command[1], command[2], command[3]

try:

distance = float(distance)

except ValueError:

print("Ошибка: расстояние должно быть числом.")

continue

graph.add\_edge(from\_vertex, to\_vertex, distance)

elif command[0] == "shortest\_path":

if len(command) != 3:

print("Ошибка: неверное количество аргументов.")

continue

from\_vertex, to\_vertex = command[1], command[2]

if from\_vertex not in graph.adjacency\_list or to\_vertex not in graph.adjacency\_list:

print("Ошибка: одна или обе вершины не существуют.")

else:

distances = dijkstra(graph, from\_vertex)

distance = distances.get(to\_vertex, float('inf'))

if distance == float('inf'):

print(f"Нет пути от {from\_vertex} до {to\_vertex}.")

else:

print(f"Кратчайшее расстояние от {from\_vertex} до {to\_vertex}: {distance}")

elif command[0] == "all\_shortest\_paths":

for vertex in graph.adjacency\_list:

print(f"Кратчайшие пути от вершины {vertex}:")

distances = dijkstra(graph, from\_vertex)

for target, distance in distances.items():

if distance == float('inf'):

print(f" До {target}: нет пути")

else:

print(f" До {target}: {distance}")

print()

elif command[0] == "add":

if len(command) != 2:

print("Ошибка: неверное количество аргументов.")

continue

key = command[1]

ht.add(key)

print(f"Вставлено: {key}")

elif command[0] == "find":

if len(command) != 2:

print("Ошибка: неверное количество аргументов.")

continue

key = command[1]

if key in ht:

print(f"Найдено")

else:

print(f"Значение {key} не найдено.")

elif command[0] == "delete":

if len(command) != 2:

print("Ошибка: неверное количество аргументов.")

continue

key = command[1]

ht.remove(key)

print(f"Значение {key} удалено.")

elif command[0] == "visualize":

if ht.size == 0:

print("Ошибка: таблица пустая.")

continue

filename = "hash\_table"

if len(command) >= 2:

filename = command[1]

visualize(ht, filename)

elif command[0] == "remove\_all":

ht.remove\_all()

print(f"Все значения удалены")

elif command[0] == "print\_table":

print(ht)

elif command[0] == "help":

print\_help()

elif command[0] == "exit":

sys.exit()

else:

print("Неизвестная команда. Введите 'help' для справки.")

**Название файла: visualization\_table.py**

from .hash\_t import \*

import graphviz

def visualize(ht, filename="hash\_table"):

"""Визуализация хэш-таблицы с детализацией AVL-деревьев"""

dot = graphviz.Digraph(comment='Hash Table', node\_attr={'shape': 'box'})

for i, tree in enumerate(ht.table):

if tree is not None:

bucket\_name = f"Bucket {i}"

dot.node(bucket\_name, label=f"{bucket\_name} ({len(tree.in\_order\_traversal())} items)")

\_visualize\_avl\_tree(dot, tree.root, bucket\_name)

dot.format = 'svg' #!

dot.render(filename, view=True, cleanup=True)

def \_visualize\_avl\_tree(dot, node, parent\_name):

if node is not None:

node\_name = f"Node {node.val}"

dot.node(node\_name, label=str(node.val))

dot.edge(parent\_name, node\_name)

\_visualize\_avl\_tree(dot, node.left, node\_name)

\_visualize\_avl\_tree(dot, node.right, node\_name)

**Название файла: main.py**

import modules

if \_\_name\_\_ == '\_\_main\_\_':

modules.user\_cli()

**Название файла: test\_avl\_tree.py**

import pytest

from modules.avl\_tree import AVLTree

@pytest.fixture

def tree():

return AVLTree()

def test\_insert(tree):

tree.insert("50")

tree.insert("30")

tree.insert("70")

tree.insert("20")

tree.insert("40")

tree.insert("60")

tree.insert("80")

# Проверяем, что элементы вставлены корректно

assert tree.in\_order\_traversal() == ["20", "30", "40", "50", "60", "70", "80"]

def test\_insert\_and\_balance(tree):

# Вставляем элементы, которые требуют балансировки

tree.insert("30")

tree.insert("20")

tree.insert("10")

# После вставки дерево должно быть сбалансированным

assert tree.in\_order\_traversal() == ["10", "20", "30"]

def test\_delete(tree):

tree.insert("50")

tree.insert("30")

tree.insert("70")

tree.insert("20")

tree.insert("40")

tree.insert("60")

tree.insert("80")

tree.delete("20")

tree.delete("30")

tree.delete("50")

# Проверяем, что элементы удалены корректно

assert tree.in\_order\_traversal() == ["40", "60", "70", "80"]

def test\_delete\_with\_balance(tree):

tree.insert("10")

tree.insert("20")

tree.insert("30")

# Удаляем элементы и проверяем баланс после удаления

tree.delete("20")

tree.delete("10")

# Дерево должно быть сбалансировано

assert tree.in\_order\_traversal() == ["30"]

def test\_contains(tree):

tree.insert("10")

tree.insert("20")

tree.insert("30")

# Проверяем, что элемент содержится в дереве

assert "20" in tree

assert "40" not in tree

def test\_get\_root(tree):

tree.insert("10")

tree.insert("20")

tree.insert("30")

root = tree.get\_root()

assert root.val == "20" # Корень дерева должен быть 20

def test\_empty\_tree(tree):

# Проверяем пустое дерево

assert tree.in\_order\_traversal() == []

assert tree.get\_root() is None

assert "any\_value" not in tree

**Название файла: test\_hash\_t.py**

import pytest

from modules.hash\_t import HashTable

@pytest.fixture

def hash\_table():

return HashTable()

def test\_add\_and\_contains(hash\_table):

# Добавляем элементы в хэш-таблицу

hash\_table.add("apple")

hash\_table.add("banana")

hash\_table.add("cherry")

# Проверяем, что элементы присутствуют в хэш-таблице

assert "apple" in hash\_table

assert "banana" in hash\_table

assert "cherry" in hash\_table

def test\_remove(hash\_table):

# Добавляем элементы

hash\_table.add("apple")

hash\_table.add("banana")

hash\_table.add("cherry")

# Удаляем элемент и проверяем, что он удалён

hash\_table.remove("banana")

assert "banana" not in hash\_table

assert "apple" in hash\_table

assert "cherry" in hash\_table

# Проверяем исключение при удалении несуществующего элемента

with pytest.raises(KeyError):

hash\_table.remove("orange")

def test\_resize(hash\_table):

# Добавляем элементы для превышения коэффициента загрузки

for i in range(1000): # В зависимости от начального размера и коэффициента

hash\_table.add(f"key{i}")

# Проверяем, что элементы все добавлены, и таблица увеличилась

assert hash\_table.count == 1000

def test\_remove\_all(hash\_table):

# Добавляем элементы

hash\_table.add("apple")

hash\_table.add("banana")

# Удаляем все элементы

hash\_table.remove\_all()

# Проверяем, что таблица очищена

assert hash\_table.count == 0

assert "apple" not in hash\_table

assert "banana" not in hash\_table

def test\_empty\_table(hash\_table):

# Проверяем пустую хэш-таблицу

assert hash\_table.count == 0

assert "apple" not in hash\_table