**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: Работа со структурами данных

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Роднов И.С. |
| Преподаватель |  | Иванов Д. В. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**на курсовую работу**

Студент: Роднов И.С.

Группа: 3342

Тема работы: Работа со структурами данных

Вы являетесь администратором университета и отвечаете за регистрацию студентов. Каждый студент имеет следующие характеристики:

ID студента (целое число) — уникальный идентификатор студента.

Имя студента (строка).

Возраст (целое число).

Специальность (строка).

Вашей задачей является разработка программы для управления регистрацией студентов. Программа должна уметь:

1. Добавлять нового студента в систему регистрации.
2. Удалять студента из системы по его ID.
3. Обновлять информацию о студенте (имя, возраст, специальность).
4. Искать студента по его ID и получать всю информацию о нем.
5. Выводить всех студентов.

Для выполнения данной задачи реализуйте наиболее подходящую структуру из изученных вами.

Разделы пояснительный записки: «Содержание», «Введение», «Анализ», «Реализация», «Исследование», «Заключение», «Приложение А. Примеры работы программы», «Приложение Б. Исходный код программы».

Дата выдачи задания: 06.11.2024

Дата сдачи реферата: 05.12.2024

Дата защиты реферата: 05.12.2024

|  |  |  |
| --- | --- | --- |
| Студент |  | Роднов И.С. |
| Преподаватель |  | Иванов Д. В. |

**Аннотация**

Курсовая работа заключается в создании программы для управления регистрацией студентов на языке Python. Помимо этого, необходимо провести исследования и анализ для выбора оптимальной структуры данных под эту задачу. Каталог src хранит файлы исходного кода и тестов. Также приложены файлы .docx и .pdf отчётов с описанием. Для создания визуализации и тестов используется библиотека matplotlib.

Пример работы программы приведен в приложении А.

Исходный код программы приведен в приложении Б.

**ANNOTATION**

The coursework consists of creating a program for managing student registration in Python. In addition, it is necessary to conduct research and analysis to select the optimal data structure for this task. The src directory stores source code and test files. Also attached are .docx and .pdf files of reports with descriptions. The matplotlib library is used to create visualization and tests.

An example of the program's operation is given in Appendix A.

The source code of the program is given in Appendix B.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 8 |
| 1. | Исследование | 9 |
| 2. | Реализация | 13 |
| 2.1.  2.1.1  2.1.2  2.1.3 | Класс Student  Конструктор \_\_init\_\_(self, student\_id, name, age, specialty)  Метод update\_info(self, name=None, age=None, specialty=None)  Метод \_\_str\_\_(self) | 13  13  13  13 |
| 2.2.  2.2.1  2.2.2  2.2.3  2.2.4  2.2.5  2.2.6  2.2.7  2.2.8  2.2.9  2.2.10 | Класс HashTable  Конструктор \_\_init\_\_(self, size=11)  Метод \_hash1(self, student\_id)  Метод \_hash2(self, student\_id)  Метод \_probe(self, student\_id, step)  Метод \_resize(self, new\_size)  Метод add(self, student)  Метод remove(self, student\_id)  Метод find(self, student\_id)  Метод list\_all(self)  Метод \_next\_prime(self, n) | 13  13  14  14  14  14  14  15  15  15  15 |
| 2.3.  2.3.1  2.3.2  2.3.3  2.3.4  2.3.5  2.3.6  3. | Класс RegistrationSystem  Конструктор \_\_init\_\_(self)  Метод add\_student(self, student\_id, name, age, specialty)  Метод remove\_student(self, student\_id)  Метод update\_student(self, student\_id, name=None, age=None, specialty=None)  Метод remove\_student(self, student\_id)  Метод list\_all\_students(self)  Анализ | 15  15  16  16  16  16  16  17 |
|  | Заключение | 19 |
|  | Приложение А. ПРИМЕР РАБОТЫ ПРОГРАММЫ | 20 |
|  | Приложение Б. ИСХОДНЫЙ КОД | 21 |

Введение

Цель работы: написать программу на языке Python, которая реализует подходящую структуру данных для управления регистрацией студентов. Для этого требуется:

* Выявить лучший вариант из известных структур данных
* Реализовать функции добавления, удаления и поиска студентов
* Обеспечить хранение и доступ к данным студентов по их id
* Протестировать реализованную структуру

1. Исследование

Для обеспечения максимальной эффективности системы, было проведено исследование известных структур данных с целью выявления, наиболее подходящей для задачи.

* 1. Массив

Массив – структура данных, которая хранит набор однородных данных, доступ к которым осуществляется по индексам. Требуются выделения непрерывного участка памяти.

Сложность операций:

* Вставка (O(1) в конец, О(n) в начало и в середину)
* Удаление (O(1) из конца, О(n) из начала и из середины)
* Поиск (O(1) в начале, О(n) в конце и в середине)

Массив не подходит для нашей задачи, так как не предоставляет удобного способа сопоставления id студента и его данных. Для поиска требуется линейный перебор элементов, что приводит к временной сложности O(n). Это делает массив неэффективным при большом количестве данных, когда быстрый поиск и обработка данных являются ключевыми требованиями.

1.2 Связный список

Связный список — базовая динамическая структура данных в информатике, состоящая из узлов, содержащих данные и ссылки («связки») на следующий и/или предыдущий узел списка.

Сложность операций:

* Вставка (O(n) в конец, О(1) в начало, в середину O(1) или O(n))
* Удаление (O(n) из конца, О(1) из начала и O(n) из середины)
* Поиск (O(1) в начале, О(n) в конце и в середине)

Связной список не подходит к задаче по той же причине, что и массив: не предоставляет удобного способа сопоставления доменного имени и IP-адреса. Это действие будет выполнятся линейно.

1.3 Стек и очередь

Стек (stack) — это структура данных, работающая по принципу LIFO (Last In, First Out), где элементы добавляются и удаляются только с одного конца (верха стека).

Очередь — это структура данных, работающая по принципу FIFO (First In, First Out), где элементы добавляются в конец очереди, а удаляются из начала очереди.

По определениям становится понятно, что невозможно удалить элемент по конкретному ID.

1.4 Хеш-таблица

Хэш-таблица - структура данных, которая позволяет хранить пары (ключ, значение) и осуществлять доступ к элементу по ключу.

Сложность операций:

* Поиск (О(n) – худший случай, O(1) – лучший случай)
* Удаление (О(n) – худший случай, O(1) – лучший случай)
* Вставка (О(n) – худший случай, O(1) – лучший случай)

В хеш-таблице операции добавления и удаления выполняются за O(1), при правильном выборе метода разрешения коллизий и соответствующих хэш функций. Это особенно важно для систем, которые работают с большими объемами данных и требуют быстрой обработки изменений. Хеш-таблица обеспечивает быстрый доступ к данным по ключу с постоянной сложностью O(1), что означает, что независимо от количества данных время для выполнения операций поиска, вставки и удаления остается постоянным. Это делает её идеальной для добавления и удаления студентов по их ID.

1.5 Деревья

Все виды деревьев не подходят для решения задачи хранения студентов по их ID, так как они не обеспечивают прямого доступа к значениям по ключу. В деревьях поиск осуществляется через обход узлов, а это означает, что доступ к данным требует больше времени по сравнению с хеш-таблицей, где поиск производится напрямую через хеш-функцию. Однако, в сравнении с массивами и односвязным списком, деревья имеют более выгодное свойство — сортировку, что делает их более эффективными.

Сложность операций:

* Вставка O(log(n))
* Удаление O(log(n))
* Поиск O(log(n))

На основе вышеизложенного исследования можно сделать вывод, что хеш-таблица является наиболее подходящей структурой данных для управления системой регистрации студентов. Она сочетает в себе все необходимые свойства для эффективной реализации системы, включая скорость выполнения операций, простоту реализации и гибкость при увеличении объема данных. В качестве метода разрешения коллизий выбор делался из 4х вариантов. Открытая адресация – метод цепочек, этот вариант не подошел так как занимает много дополнительной памяти. Закрытая адресация – линейное пробирование, этот метод не был выбран, так как он имеет риски первичной кластеризации и сильного замедления работы программы при высокой загруженности таблицы. Квадратичное пробирование – аналогично линейному имеет высокую вероятность появления вторичной кластеризации, из-за чего скорость работы значительно снижается. Исходя из этих доводов, был сделан выбор в сторону двойного хэширования. При правильном выборе хэш функций, двойное хэширование максимально равномерно распределяет элементы по таблице, избегая кластеризаций, а также не занимает дополнительной памяти как метод цепочек, так же после исследований всех вариантов на практике, было выявлено что двойной хэширование работает наиболее быстро.

2. Реализация

2.1 Класс Student

Класс Student представляет собой модель студента с атрибутами:

* student\_id: уникальный идентификатор студента.
* name: имя студента.
* age: возраст студента.
* specialty: специальность студента.

2.1.1 Конструктор \_\_init\_\_(self, student\_id, name, age, specialty)

Конструктор инициализирует объект студента с указанными параметрами. Атрибуты сохраняются в специально отведённых слотах \_\_slots\_\_ для оптимизации памяти.

2.1.2 Метод update\_info(self, name=None, age=None, specialty=None)

Обновляет информацию о студенте. Если передан новый параметр (например, name), то обновляет соответствующее поле.

2.1.3 Метод \_\_str\_\_(self)

Возвращает строковое представление объекта студента в формате: ID: {student\_id}, Name: {name}, Age: {age}, Specialty: {specialty}

2.2 Класс HashTable

Класс HashTable реализует хэш-таблицу с двойным хэшированием для управления объектами Student.

2.2.1 Конструктор \_\_init\_\_(self, size=11)

Инициализирует хэш-таблицу:

* size: начальный размер таблицы (по умолчанию 11).
* table: массив для хранения элементов.
* count: текущее количество элементов в таблице

2.2.2 Метод \_hash1(self, student\_id)

Вычисляет первый хэш-индекс используя мультипликативное хеширование. В этой хэш функции задействуется константа золотого сечения, равная 0.6180339887. Этот способ хэширования помогает уменьшить количество коллизий и распределить значения по таблице наиболее равномерно. Для этого id студента умножается на эту константу, после чего из полученного значения вычитается оно же, округленное вниз до целого. После чего мы возвращаем в качестве хэша остаток от деления размера таблицы на эту разницу, все это предварительно округлив вниз до целого.

2.2.3 Метод \_hash2(self, student\_id)

Вычисляет второй хэш-индекс для разрешения коллизий. Им является ненулевое число равное 1 + (остаток от id студента деленного на размер таблицы – 1)

2.2.4 Метод \_probe(self, student\_id, step)

Выполняет двойное хэширование для разрешения коллизий и вычисления нового индекса. Им является остаток от деления суммы результата первой хэш функции и второй, умноженной на step.

2.2.5 Метод \_resize(self, new\_size)

Изменяет размер таблицы:

1. Создаёт новую таблицу увеличенного или уменьшенного размера.
2. Перемещает существующие элементы в новую таблицу, используя их новые хэш-индексы.
3. Размер таблицы увеличивается при заполненности > 70% или уменьшается при заполненности < 20%. Это делается для снижения вероятности коллизий при большой загрузке таблицы, и для освобождения лишней памяти при низкой загрузке.

2.2.6 Метод add(self, student)

Добавляет объект Student в таблицу:

1. Проверяет, превышает ли заполненность таблицы 70%. Если да, вызывает \_resize.
2. Использует двойное хэширование для вставки.
3. Если студент с таким student\_id уже существует, обновляет его данные.

2.2.7 Метод remove(self, student\_id)

Удаляет объект Student с указанным student\_id:

1. Использует двойное хэширование для поиска студента.
2. Если студент найден, помечает его как удалённого (DELETED).
3. Если заполненность таблицы падает ниже 20%, вызывает \_resize.

2.2.8 Метод find(self, student\_id)

Находит студента по его student\_id:

1. Использует двойное хэширование для поиска.
2. Возвращает строковое представление объекта Student или None, если студент не найден.

2.2.9 Метод list\_all(self)

Возвращает список всех студентов в таблице. Пропускает удалённые записи и пустые ячейки. Возвращает строку со списком студентов или сообщение "No students registered.".

2.2.10 Метод \_next\_prime(self, n)

Находит ближайшее простое число больше или равное n. Используется для определения размеров таблицы, что снижает вероятность коллизий.

2.3 Класс RegistrationSystem

Класс RegistrationSystem предоставляет интерфейс для управления студентами с помощью хэш-таблицы.

2.3.1 Конструктор \_\_init\_\_(self)

Создаёт объект системы регистрации с пустой хэш-таблицей (HashTable).

2.3.2 Метод add\_student(self, student\_id, name, age, specialty)

Добавляет нового студента:

1. Создаёт объект Student.
2. Добавляет его в хэш-таблицу с помощью метода add.
3. Возвращает сообщение о добавлении.

2.3.3 Метод remove\_student(self, student\_id)

Удаляет студента с указанным student\_id:

1. Использует метод remove хэш-таблицы.
2. Возвращает сообщение об успешном или неудачном удалении.

2.3.4 Метод update\_student(self, student\_id, name=None, age=None, specialty=None)

Обновляет данные студента:

1. Если студент найден, добавляет обновлённую запись в хэш-таблицу, заменяя старую.
2. Если студент не найден, добавляет новую запись.
3. Возвращает сообщение об обновлении.

2.3.5 Метод remove\_student(self, student\_id)

Ищет студента по student\_id:

1. Использует метод find хэш-таблицы.
2. Возвращает информацию о студенте или сообщение о его отсутствии.

2.3.6 Метод list\_all\_students(self)

Выводит список всех зарегистрированных студентов с помощью метода list\_all хэш-таблицы.

Разработанный программный код см. в приложении Б.

1. Анализ

В ходе тестирования был проведен замер времени выполнения операций с использованием хеш-таблицы для управления доменами и их IP-адресами. Были использованы различные объемы данных – (0, 50000, 100000, 150000, 200000, 250000, 300000) и (500000, 600000, 700000, 800000, 900000, 1000000 операций) и замерено время, необходимое для выполнения 4 ключевых операций: вставка, поиск по домену, поиск по IP и удаление.

Время выполнения операции вставки оставалось стабильно близким к O(1) для всех объемов данных. Даже при миллионе операций, время вставки не увеличивалось значительно, что подтверждает правильность работы хеш-таблицы и метода разрешения коллизий. Аналогично с остальными операциями.

Результаты анализов приведены на рисунке 1 и 2.

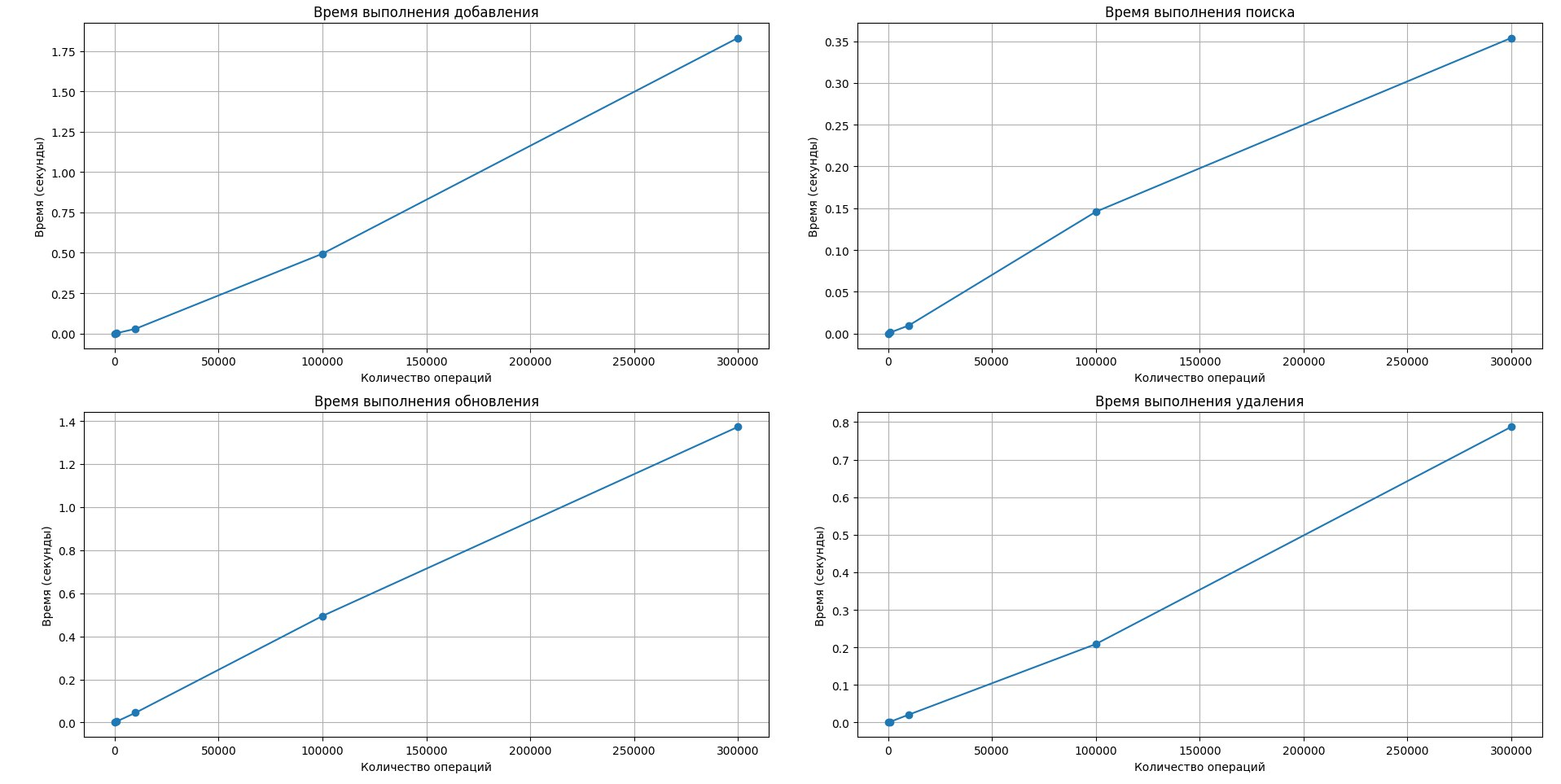


Рис 1 - Результаты анализа для размеров до 300 000 элементов.

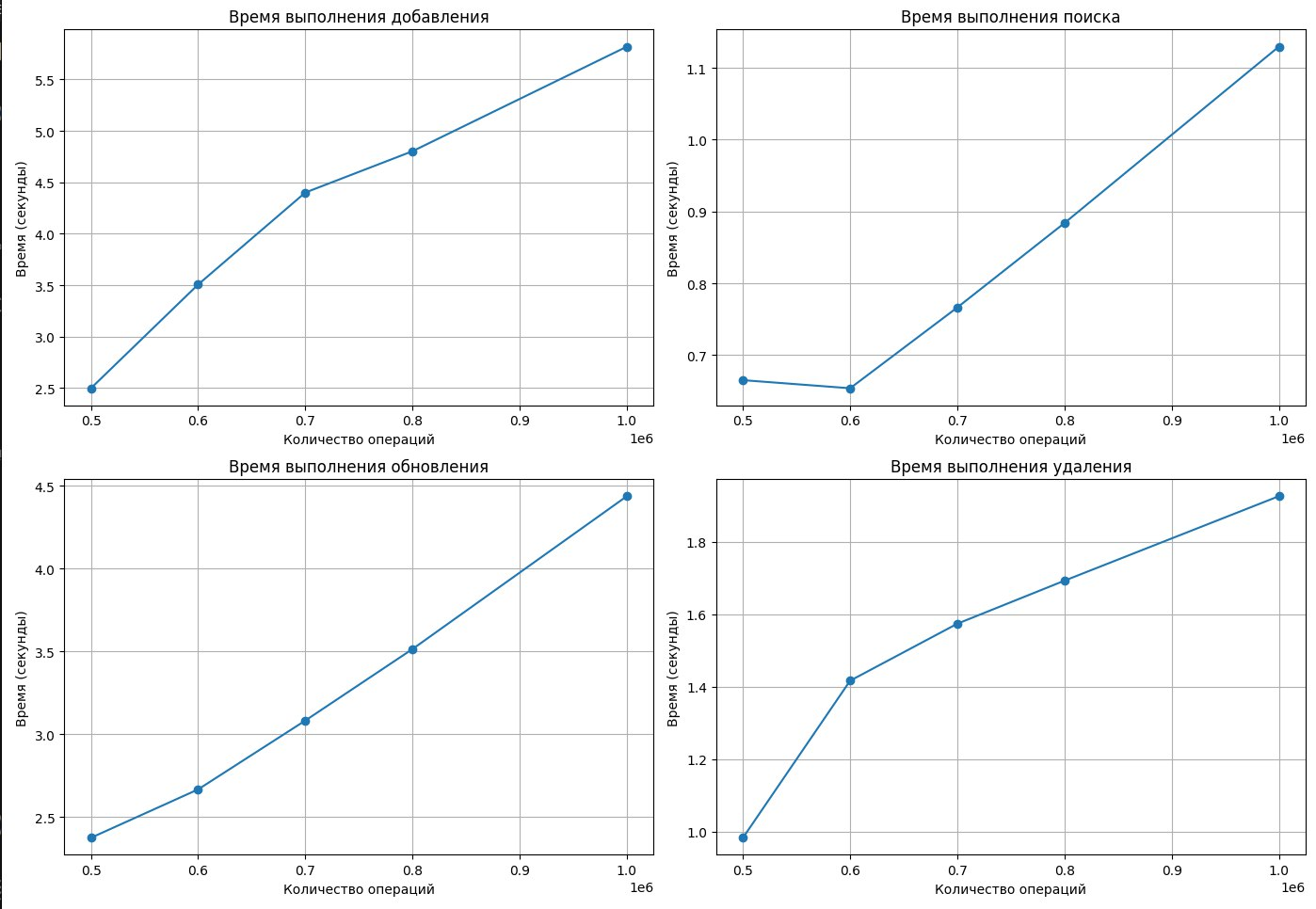


Рис. 2 – Результаты анализа для размеров до 1 000 000 элементов.

Заключение

Была разработана программа на языке Python, реализующая хеш-таблицу для управления системой регистрации студентов. В рамках данной реализации были созданы следующие функции: создание записей о студентах с указанием их ID, имени, возраста и специальности; поддержка добавления, удаления и обновления данных студентов; реализация механизма поиска студента по ID и вывода всех зарегистрированных студентов.

Для хеширования используется метод двойного хеширования, который обеспечивает эффективное разрешение коллизий. Программа автоматически масштабирует таблицу при добавлении или удалении записей, что позволяет сохранять высокую производительность. Выбранная структура идеально подходит для реализации задания, так как обеспечивает быстрое добавление, удаление, поиск и сортировку пользователей, что делает её эффективным инструментом для управления регистрацией студентов.

Приложение А

**ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ**

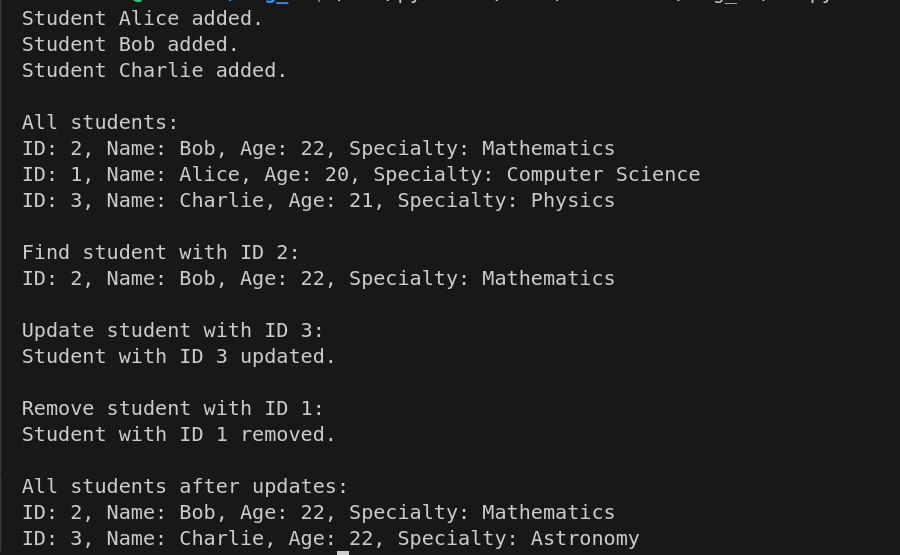
****

Рис. 3 – Результаты работы программы

Приложение Б

**Исходный код программы**

Файл main.py:

class Student:

    \_\_slots\_\_ = ['student\_id', 'name', 'age', 'specialty']

    def \_\_init\_\_(self, student\_id, name, age, specialty):

        """

        Initializes a Student object with the given student ID, name, age, and specialty.

        """

        self.student\_id = student\_id

        self.name = name

        self.age = age

        self.specialty = specialty

    def update\_info(self, name=None, age=None, specialty=None):

        """

        Updates the student's information (name, age, specialty) if the provided arguments are not None.

        """

        if name is not None:

            self.name = name

        if age is not None:

            self.age = age

        if specialty is not None:

            self.specialty = specialty

    def \_\_str\_\_(self):

        """

        Returns a string representation of the Student object, including the student ID, name, age, and specialty.

        """

        return f"ID: {self.student\_id}, Name: {self.name}, Age: {self.age}, Specialty: {self.specialty}"

class HashTable:

    DELETED = "<Deleted>"

    def \_\_init\_\_(self, size=11):

        """

        Initializes a hash table with a given size (default is 11).

        """

        self.size = size

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

        self.count = 0

    def \_hash1(self, student\_id):

        """

        Computes the first hash value based on the student's ID using the golden ratio.

        """

        A = 0.6180339887

        hash\_value = student\_id \* A

        fractional\_part = hash\_value - int(hash\_value)

        return int(self.size \* fractional\_part)

    def \_hash2(self, student\_id):

        """

        Computes the second hash value for use in double hashing.

        """

        return 1 + (student\_id % (self.size - 1))

    def \_probe(self, student\_id, step):

        """

        Computes the next index for probing during collision resolution using double hashing.

        """

        return (self.\_hash1(student\_id) + step \* self.\_hash2(student\_id)) % self.size

    def \_resize(self, new\_size):

        """

        Resizes the hash table to the given new size and rehashes all existing entries.

        """

        old\_table = self.table

        self.size = new\_size

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

        self.count = 0

        for entry in old\_table:

            if entry is not None and entry != self.DELETED:

                self.add(entry)

    def add(self, student):

        """

        Adds a student to the hash table. If the table is more than 70% full, it resizes before adding.

        """

        if self.count / self.size > 0.70:

            self.\_resize(self.\_next\_prime(self.size \* 2))

        step = 0

        index = self.\_hash1(student.student\_id)

        while self.table[index] is not None and self.table[index] != self.DELETED:

            if self.table[index].student\_id == student.student\_id:

                self.table[index].update\_info(student.name, student.age, student.specialty)

                return

            step += 1

            index = self.\_probe(student.student\_id, step)

        self.table[index] = student

        self.count += 1

    def remove(self, student\_id):

        """

        Removes a student from the hash table by student ID. If the table is less than 20% full, it resizes after removal.

        """

        step = 0

        index = self.\_hash1(student\_id)

        while self.table[index] is not None:

            if self.table[index] != self.DELETED and self.table[index].student\_id == student\_id:

                self.table[index] = self.DELETED

                self.count -= 1

                if self.count / self.size < 0.2:

                    self.\_resize(self.\_next\_prime(self.size // 2))

                return True

            step += 1

            index = self.\_probe(student\_id, step)

        return False

    def find(self, student\_id):

        """

        Searches for a student by ID and returns their information if found.

        """

        step = 0

        index = self.\_hash1(student\_id)

        while self.table[index] is not None:

            if self.table[index] != self.DELETED and self.table[index].student\_id == student\_id:

                return str(self.table[index])

            step += 1

            index = self.\_probe(student\_id, step)

        return None

    def list\_all(self):

        """

        Returns a string representation of all students in the hash table.

        """

        students = [str(student) for student in self.table if student is not None and student != self.DELETED]

        return "\n".join(students) if students else "No students registered."

    def \_next\_prime(self, n):

        """

        Finds the next prime number greater than or equal to 'n' for resizing the hash table.

        """

        def is\_prime(num):

            if num < 2:

                return False

            for i in range(2, int(num \*\* 0.5) + 1):

                if num % i == 0:

                    return False

            return True

        while not is\_prime(n):

            n += 1

        return n

class RegistrationSystem:

    def \_\_init\_\_(self):

        """

        Initializes the registration system with an empty hash table.

        """

        self.hash\_table = HashTable()

    def add\_student(self, student\_id, name, age, specialty):

        """

        Adds a new student to the registration system and returns a confirmation message.

        """

        student = Student(student\_id, name, age, specialty)

        self.hash\_table.add(student)

        return f"Student {name} added."

    def remove\_student(self, student\_id):

        """

        Removes a student from the registration system by student ID and returns a confirmation message.

        """

        if self.hash\_table.remove(student\_id):

            return f"Student with ID {student\_id} removed."

        else:

            return f"No student found with ID {student\_id}."

    def update\_student(self, student\_id, name=None, age=None, specialty=None):

        """

        Updates the information of an existing student by their ID.

        """

        if self.hash\_table.find(student\_id):

            self.hash\_table.add(Student(student\_id, name, age, specialty))

            return f"Student with ID {student\_id} updated."

        else:

            return f"Student with ID {student\_id} not found."

    def find\_student(self, student\_id):

        """

        Finds and returns the information of a student by their ID.

        """

        student\_info = self.hash\_table.find(student\_id)

        if student\_info:

            return student\_info

        else:

            return f"No student found with ID {student\_id}."

    def list\_all\_students(self):

        """

        Lists all the students currently registered in the system.

        """

        students = self.hash\_table.list\_all()

        print(students)

Файл tests.py:

from main import \*

def test\_registration\_system():

    system = RegistrationSystem()

    print("Добавление студентов:")

    assert system.add\_student(1, "Alice", 20, "Computer Science") == "Student Alice added."

    assert system.add\_student(2, "Bob", 22, "Mathematics") == "Student Bob added."

    assert system.add\_student(3, "Charlie", 21, "Physics") == "Student Charlie added."

    system.hash\_table.list\_all()

    print()

    assert system.find\_student(1) == "ID: 1, Name: Alice, Age: 20, Specialty: Computer Science"

    assert system.find\_student(2) == "ID: 2, Name: Bob, Age: 22, Specialty: Mathematics"

    assert system.find\_student(3) == "ID: 3, Name: Charlie, Age: 21, Specialty: Physics"

    print("Обновление информации о студенте:")

    assert system.update\_student(2, age=23, specialty="Applied Mathematics") == "Student with ID 2 updated."

    assert system.find\_student(2) == "ID: 2, Name: Bob, Age: 23, Specialty: Applied Mathematics"

    system.hash\_table.list\_all()

    print()

    print("Удаление студента:")

    assert system.remove\_student(1) == "Student with ID 1 removed."

    assert system.find\_student(1) == "No student found with ID 1."

    system.hash\_table.list\_all()

    print()

    print("Удаление несуществующего студента:")

    assert system.remove\_student(5) == "No student found with ID 5."

    print("Добавление новых студентов:")

    for i in range(4, 10):

        assert system.add\_student(i, f"Student{i}", 18 + i % 10, f"Specialty{i % 3}") == f"Student Student{i} added."

    system.hash\_table.list\_all()

    print()

    print("Проверка хэширования:")

    for i in range(4, 10):

        assert system.find\_student(i) == f"ID: {i}, Name: Student{i}, Age: {18 + i % 10}, Specialty: Specialty{i % 3}"

    print("Все тесты пройдены успешно!")

if \_\_name\_\_ == "\_\_main\_\_":

    test\_registration\_system()