**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 наиболее эффективной структуры данных для решения поставленной задачи**

Студент гр. 3384 _____ Пьянков М.Ф.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург

**2024**

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

Студент Пьянков М.Ф.

Группа 3384

Тема работы: Реализация наиболее эффективной структуры данных для решения поставленной задачи.

Оценка: Хорошо; Вариант 5

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: добавлять нового студента в систему регистрации, удалять студента из системы по его ID, обновлять информацию о студенте (имя, возраст, специальность), искать студента по его ID и получать всю информацию о нем, выводить всех студентов.

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

Дата выдачи задания: 06.11.2024

Дата сдачи реферата:08.12.2024

Дата защиты реферата: 09.12.2024

Студент гр. 3384 _____ Пьянков М. Ф.

Преподаватель _____ Шестопалов Р.П.

## АННОТАЦИЯ

Курсовая работа посвящена реализации наиболее эффективной структуры данных в рамках поставленного задания. Исходя из знаний, полученных на лекциях предлагается интуитивно выбрать лучшую структуру данных далее произвести исследование эффективности её работы, и по возможности сравнить с другими, обосновав таким образом свой выбор.

## SUMMARY

The course work is devoted to the implementation of the most efficient data structure within the framework of the assigned task. Based on the knowledge gained in the lectures, it is proposed to intuitively select the best data structure, then conduct a study of the effectiveness of its work, and, if possible, compare it with others, thus justifying your choice.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Цель работы:

Написать программу, решающую поставленную задачу наиболее эффективным способом.

Основные теоритические положения: в ходе лекций по данной дисциплине рассматривались следующие структуры данных с средней сложностью операций: вставка, поиск удаление (операции рассматриваются при применении в случайное место и соответственно условиям поставленного задания):

Массив: O(1), O(n), O(1)

Связные списки: O(n), O(n), O(n)

Развёрнутый список: O(√n), O(√n), O(√n)

Стек: O(n), O(n), O(n)

Очередь: O(n), O(n), O(n)

Очередь с приоритетом: O(n), O(n), O(n)

Двоичные кучи: O(log(n)), O(log(n)), O(log(n))

Бинарное дерево: O(log(n)), O(log(n)), O(log(n))

АВЛ-дерево: O(log(n)), O(log(n)), O(log(n))

КЧД-дерево: O(log(n)), O(log(n)), O(log(n))

Хэш-таблица: O(1), O(1), O(1)

Очевидно, что имеет смысл рассмотреть хэш-таблицу в качестве лучшей структуры данных. АВЛ и КЧД деревья могут подойти, но их сложность избыточна для решения задачи а архитектурные ограничения могут при определённых обстоятельствах мешать.

## ХОД ВЫПОЛНЕНИЯ РАБОТЫ

**Выбор структуры данных**

Заметим, что у структуры данных: хэш-таблица худшее время работы для вставки, поиска, удаления элементов: O(n), а лучшее O(1) соответственно. Такое решение подходит для выполнения поставленного задания.

**Хэш-таблица**

***Класс Student***

Этот класс необходим для более удобного представления данных в хэш-таблице. Поля: ***name*** – имя, ***age*** – возраст, ***specials*** – специальность. Методы: ***def __str__(self) -> str —*** представление в виде строки, ***__eq__(self, other) —*** оператор сравнения.

***Метод решения коллизий***

В качестве метода решения коллизий будем использовать метод цепочек. Примем за среднее значение максимальную длину цепочек в 3 элемента. Такое значение необходимо, чтобы адекватно оценивать заполненность хэш-таблицы. Проблема с решением коллизий также решается за счёт того, что можно хранить ключ, который хэшируется вместе с значением и проверять непосредственно ключи при возникновении коллизии.

***Метод __init__(self, max_size, max_load, max_chain_length)***

Конструктор хэш-таблицы. При создании также указывается текущий максимальный размер, ожидаемый размер цепочек и максимальная загруженность.

***Хэш-функция hash_function(self, key: str) -> int***

Была выбрана полиномиальная хэш функция, с простым числом p = 97, производится хэширование ID студента.

***Метод load_factor(self) -> float***

Метод считает текущую загруженность хэш-таблицы.

***Метод __re_balance__(self)***

Метод, осуществляющий ребалансировку хэш-таблицы, в случае, если после добавления нового элемента загруженность превысила лимит ***max_load*** (проверяется в ***load_factor(self)***). Происходит увеличение ***max_size*** в 2 раза, затем все элементы заново хэшируются и добавляются в уже увеличенную хэш-таблицу. Таким образом поддерживается работоспособность и эффективность структуры данных.

***Метод insert(self, student_id: str, student: Student)***

Метод, добавляющий значение по ключу с учётом метода цепочек. А также учитывающая уникальность ключа.

***Метод erase(self, student_id: str)***

Метод, удаляющий значение по ключу с учётом метода цепочек.

***Метод update(self, student_id: str, student: Student)***

Метод, обновляющий значение по ключу.

***Метод search(self, student_id: str)***

Метод, производящий поиск значения по ключу.

***Метод __str__(self) -> str***

Метод, возвращающий строковое представление хэш-таблицы.

**Интерфейс**

Для удобства и демонстрации работы внутри точки входа напишем CLI-подобный интерфейс для нашей программы, он будет содержать команды соответствующие всем методам структуры и обеспечивающие функционал программы.

Для подробностей см. ПРИЛОЖЕНИЕ А.

## ТЕСТИРОВАНИЕ И ПРИМЕРЫ РАБОТЫ

Тестирование проводилось на случайно сгенерированном наборе данных для каждого метода и было успешно пройдено. Замечание: сначала был протестирован метод search, результат его работы сравнивался непосредственно с значением, которое находилось в ячейке с индексом, равным хэшу данного ключа. Далее были протестированы методы: insert, erase, update, в качестве проверки выступал встроенный в python словарь dict() с результатами аналогичных функций которого сравнивались результаты методов хэш-таблицы (при помощи функции search, в работоспособность которой была подтверждена первым тестированием).

Примеры работы(см рис.2, рис. 3, рис. 4 и рис. 5):

```
Программа для администирования базы данных студентов, команды: [insert, search, erase, update, exit]
Сначала введите наименование команды, далее следуйте указаниям, появившимся в консоли
insert
Введите имя студента: Катя
Введите возраст студента: 21
Введите специальность студента: ПИ
Введите ID, по которому вы хотите добавить студента: 09
Добавление успешно выполнено
insert
Введите имя студента: Гена
Введите возраст студента: 19
Введите специальность студента: Правоведение
Введите ID, по которому вы хотите добавить студента: 25
Добавление успешно выполнено
insert
Введите имя студента: Никита
Введите возраст студента: 18
Введите специальность студента: ПМИ
Введите ID, по которому вы хотите добавить студента: 25
Студент с таким ID уже есть!
Добавление не было выполнено
insert
Введите имя студента: Никита
Введите возраст студента: 25
Введите специальность студента: ПМИ
Введите ID, по которому вы хотите добавить студента: 17
Добавление успешно выполнено
print
Вывод всей базы данных:
ID: 25 имя: Гена, возраст: 19, специальность: Правоведение
ID: 17 имя: Никита, возраст: 25, специальность: ПМИ
ID: 09 имя: Катя, возраст: 21, специальность: ПИ
```

Рисунок 2 — Добавление студента в базу данных и её вывод

```
print
Вывод всей базы данных:
ID: 25 имя: Гена, возраст: 19, специальность: Правоведение
ID: 17 имя: Никита, возраст: 25, специальность: ПМИ
ID: 09 имя: Катя, возраст: 21, специальность: ПИ
update
Введите ID, по которому вы хотите обновить данные студента: 25
Введите имя студента: Геннадий
Введите возраст студента: 20
Введите специальность студента: Физика
Обновление успешно выполнено
print
Вывод всей базы данных:
ID: 25 имя: Геннадий, возраст: 20, специальность: Физика
ID: 17 имя: Никита, возраст: 25, специальность: ПМИ
ID: 09 имя: Катя, возраст: 21, специальность: ПИ
```

Рисунок 3 — Обновление данных студента в базе данных

```
print
Вывод всей базы данных:
ID: 25 имя: Геннадий, возраст: 20, специальность: Физика
ID: 17 имя: Никита, возраст: 25, специальность: ПМИ
ID: 09 имя: Катя, возраст: 21, специальность: ПИ
erase
Введите ID, по которому вы хотите удалить студента: 17
Удаление успешно выполнено
print
Вывод всей базы данных:
ID: 25 имя: Геннадий, возраст: 20, специальность: Физика
ID: 09 имя: Катя, возраст: 21, специальность: ПИ
```

Рисунок 4 — Удаление студента из базы данных

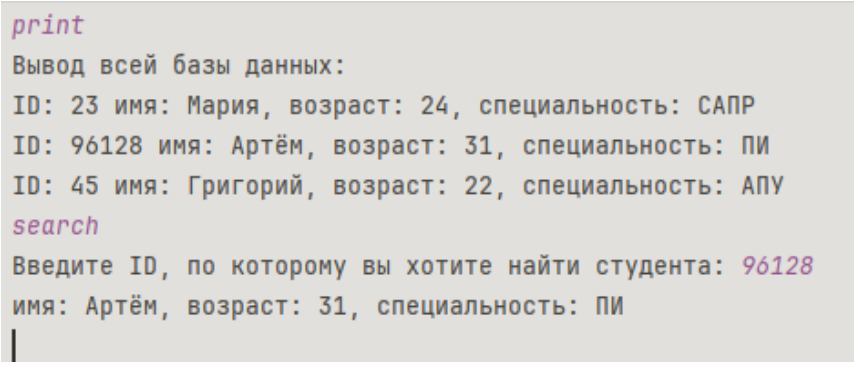


Рисунок 5 — Поиск студента в базе данных

Как видно из примеров, программа отработала корректно. Все условия курсовой работы удовлетворены.

Не рассматривались примеры с коллизией, но выбранная хэш-функция достаточно сложная, чтобы свести их к минимуму вследствие чего получить коллизии при работе с не слишком большим объёмом данных трудно. Однако даже в случае возникновения коллизий метод цепочек, а также непосредственное сравнение ключей решают эту проблему. Случаи с коллизией были протестированы в модуле с тестами.

# ИССЛЕДОВАНИЕ

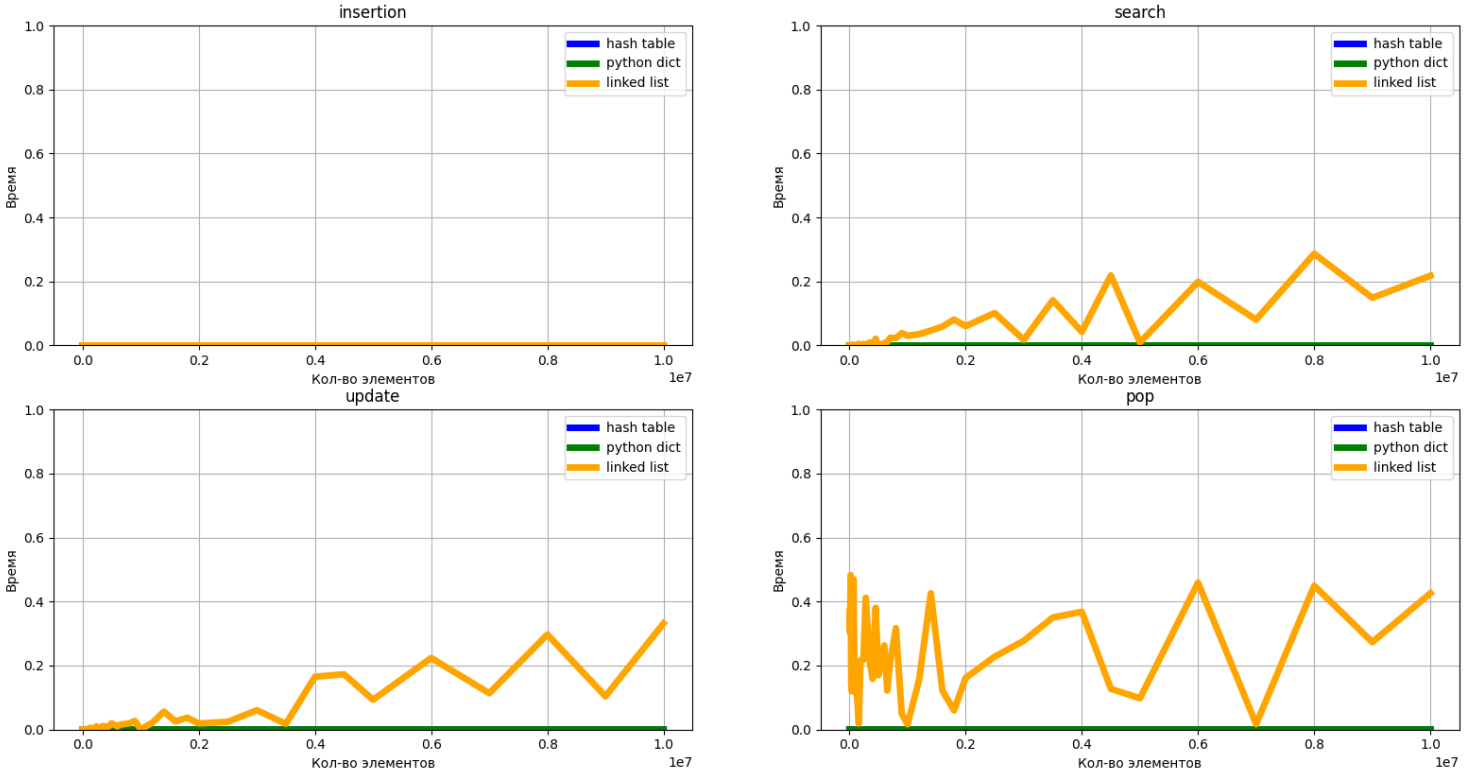


Рисунок 1 — исследование эффективности структур данных

Как можно заметить исходя из рис.1, сложность работы всех методов реализованной хэш-таблицы составляет O(1) для всех методов, что является лучшим случаем. Хэш-таблица эффективнее производила поиск, обновление и удаление элементов, чем связный список. Также видно, что реализованная хэш-таблица не уступает встроенному в python словарю dict(). Теоритические предположения совпали, хэш-таблица является лучше структурой данных для решения поставленной задачи.

Выбор метода решения коллизий и его эффективность: в рамках данной задачи лучше подойдёт метод цепочек, так как вставка в таком случае будет производится быстрее, чем при пробировании, так как новый элемент в худшем случае добавиться в конец какой-то цепочки. Двойное хэширование также может подойти, но при большой заполненности таблицы операции могут выполняться дольше, чем при методе цепочек.

Виды хэш-функций: от хэш-функции косвенно зависит эффективность структуры данных. Необходимо придумать такую функцию, чтобы возникало как можно меньше коллизий. В следствии этого была выбрана полиномиальная функция для хэширования ID студента. Единственный минус такого выбора это сложность расчёта O(m), где m – длина ID, но на практике ID не превышают длины 100-1000 символов, а то и меньше. Так как 100 символов это уже 10^99 вариантов, а 1000 символов — 10^999 соответственно, что достаточно много в рамках поставленной задачи.

Для точности исследования скорости работы подготовлены табл. 1, табл. 2, табл. 3:

| Структура данных | Вставка | Поиск | Обновление | Удаление |
|---|---|---|---|---|
| Хэш-таблица | 2.1000014385208488e-06 | 2.9000002541579306e-06 | 1.049999991664663e-05 | 2.4999972083605826e-06 |
| Python словарь | 4.00003045797348e-07 | 5.00003807246685e-07 | 1.2999953469261527e-06 | 1.8999999156221747e-06 |
| Связный список | 6.999980541877449e-07 | 0.022964999996474944 | 0.009630799999285955 | 0.051158700000087265 |

Таблица 1 — Скорость работы при 1000000 элементах

| Структура данных | Вставка | Поиск | Обновление | Удаление |
|---|---|---|---|---|
| Хэш-таблица | 2.1999949240125716e-06 | 3.0000010156072676e-06 | 1.550000160932541e-05 | 2.1000014385208488e-06 |
| Python словарь | 3.000022843480 11e-07 | 4.999965312890708e-07 | 1.4999968698248267e-06 | 1.7999991541728377e-06 |
| Связный список | 6.999980541877449e-07 | 0.0319524000005913 | 0.0711443999971 3795 | 0.4220647000038298 |

Таблица 2 — Скорость работы при 5000000 элементах

| Структура данных | Вставка | Поиск | Обновление | Удаление |
|---|---|---|---|---|
| Хэш-таблица | 3.4999975468963385e-06 | 4.8000001697800105e-06 | 8.9000022853724466e-06 | 2.7999994927085936e-06 |
| Python словарь | 1.2999953469261527e-06 | 1.0000003385357559e-06 | 1.1999945854768157e-06 | 2.1000014385208488e-06 |
| Связный список | 7.999988156370819e-07 | 0.1419225999981616 | 0.2898535999993328 | 0.4748089000058826 |

Таблица 3 — Скорость работы при 10000000 элементах

# ЗАКЛЮЧЕНИЕ

В результате написания курсовой работы была реализована структура данных — хэш-таблица с методом цепочек для разрешения коллизий и полиномиальной хэш-функцией. В процессе выполнения курсовой работы освоены новые знания, а материал, полученный на лекциях, закреплён на практике.

Поставленная задача выполнена, также были произведены тестирование и исследование скорости работы.

## СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Сведения о хэш-таблицах // URL: https://ru.wikipedia.org/wiki/Хеш-таблица

2. Сведения о хэш-таблицах и методах решения коллизий // URL: https://habr.com/ru/articles/509220/

3. Квадратичное пробирование // URL: https://en.wikipedia.org/wiki/Quadratic_probing

4. Хэширование // URL: https://aliev.me/runestone/SortSearch/Hashing.html

5. Объяснение работы хэш-таблицы // URL: https://habr.com/ru/articles/704724/

6. Презентация по хэш-таблицам из лекций // URL: https://docs.google.com/presentation/d/1mMW8VnFz-4he0PX2QGv49d9qlENoNDOETD1EZzWWMVI/edit#slide=id.g48597e4848_0_1

# ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main_cw.py

```
class Student:
    def __init__(self, full_name: str, age: int, special: str):
        self.full_name = full_name
        self.age = age
        self.special = special

    def __eq__(self, other):
        if other is None:
            return 0
                return self.age == other.age and self.full_name ==
other.full_name and self.special == other.special

    def __str__(self) -> str:
                return f'имя: {self.full_name}, возраст: {self.age},
специальность: {self.special}'


class HashTable:
                def __init__(self, max_size=997, max_load=0.75,
max_chain_length=3):
        self.max_size = max_size
        self.current_size = 0
        self.max_load = max_load
        self.max_chain_length = max_chain_length
        self.data = [[] for _ in range(self.max_size)]

    def load_factor(self) -> float:
        return self.current_size / self.max_size

    def hash_function(self, key: str) -> int:
        p = 97
        key_hash = 0
                key_hash += sum([ord(key[i]) * (p ** i) for i in
range(len(key))])
        return key_hash % self.max_size

    def __re_balance__(self):
        old_data = self.data
        self.max_size *= 2
        self.data = [[] for _ in range(self.max_size)]
        self.current_size = 0
        for chain in old_data:
            for student in chain:
                self.insert(student[0], student[1])

    def insert(self, student_id: str, student: Student):
        key_hash = self.hash_function(student_id)
        for i in self.data[key_hash]:
            if i[0] == student_id:
                print('Студент с таким ID уже есть!')
                return 0
```

```
        self.data[key_hash].append([student_id, student])
        self.current_size += 1

        if self.load_factor() > self.max_load:
            self.__re_balance__()
        return 1

    def erase(self, student_id: str):
        key_hash = self.hash_function(student_id)
        if self.data[key_hash] is []:
            print('Нет студента с таким ID!')
            return 0
        for i in range(len(self.data[key_hash])):
            if self.data[key_hash][i][0] == student_id:
                self.data[key_hash].pop(i)
                self.current_size -= 1
                return 1
        print('Нет студента с таким ID!')
        return 0

    def update(self, student_id: str, student: Student):
        key_hash = self.hash_function(student_id)
        if self.data[key_hash] is []:
            print('Нет студента с таким ID!')
            return 0
        for i in range(len(self.data[key_hash])):
            if self.data[key_hash][i][0] == student_id:
                self.data[key_hash][i][1] = student
                return 1
        print('Нет студента с таким ID!')
        return 0

    def search(self, student_id: str):
        key_hash = self.hash_function(student_id)
        if self.data[key_hash] is []:
            return None
        else:
            for i in range(len(self.data[key_hash])):
                if self.data[key_hash][i][0] == student_id:
                    return self.data[key_hash][i][1]
        return None

    def __str__(self) -> str:
        res = ''
        for i in range(self.max_size):
            for j in range(len(self.data[i])):
                        res += 'ID: ' + self.data[i][j][0] + ' ' +
self.data[i][j][1].__str__() + '\n'
        if res != '':
            return res[:-1]
        else:
            return 'Empty'


def student_input():
```

```
    name = input('Введите имя студента: ')
    age = input('Введите возраст студента: ')
    special = input('Введите специальность студента: ')
    while True:
        try:
            age = int(age)
            break
        except Exception:
            print('Ошибка, вы неверно ввели возраст студента!')
            age = input('Введите возраст студента: ')
    student = Student(name, age, special)
    return student


if __name__ == '__main__':
       print('Программа для администирования базы данных студентов,
команды: [insert, search, erase, update, exit]')
       print('Сначала введите наименование команды, далее следуйте
указаниям, появившимся в консоли')
    command = ''
       commands = ['insert', 'search', 'erase', 'update', 'exit',
'print']
    hashTable = HashTable()
    while command != 'exit':
        command = input()
        while not (command in commands):
            print('Вы ввели неверную команду, попробуйте снова!')
            command = input()

        if command == 'exit':
            exit()
        elif command == 'insert':
            student = student_input()
               ID = input('Введите ID, по которому вы хотите добавить
студента: ')
            flag = hashTable.insert(ID, student)
            if flag:
                print('Добавление успешно выполнено')
            else:
                print('Добавление не было выполнено')
        elif command == 'search':
                 ID = input('Введите ID, по которому вы хотите найти
студента: ')
            if hashTable.search(ID) == None:
                print('Нет студента с таким ID')
            else:
                print(hashTable.search(ID))
        elif command == 'erase':
               ID = input('Введите ID, по которому вы хотите удалить
студента: ')
            flag = hashTable.erase(ID)
            if flag:
                print('Удаление успешно выполнено')
            else:
                print('Удаление не было выполнено')
```

```
        elif command == 'update':
               ID = input('Введите ID, по которому вы хотите обновить
данные студента: ')
            student = student_input()
            flag = hashTable.update(ID, student)
            if flag:
                print('Обновление успешно выполнено')
            else:
                print('Обновление не было выполнено')
        elif command == 'print':
            print('Вывод всей базы данных:')
            print(hashTable)
```