САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №6
по курсу «Алгоритмы и структуры данных»
Тема: Хеширование. Хеш-таблицы
Вариант 6

Выполнил:
Данилова А.В.
К3141 (номер группы)

Проверила:
Артамонова В.Е.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

1 задача. Множество

Реализуйте множество с операциями «добавление ключа», «удаление ключа», «проверка существования ключа».

Листинг кода:

```
from pathlib import Path
from lab_6.utils.utils import work


class HashTable:
    def __init__(self):
        self.table = {}

    def key_in_table(self, key):
        if key in self.table:
            return True
        return False

    def add_key(self, key, val=None):
        if not self.key_in_table(key):
            self.table[key] = val

    def del_key(self, key):
        if self.key_in_table(key):
            self.table.pop(key)


def main(n, *actions):
    ans = []
    arr = HashTable()
    for act, key in actions:
        match act:
            case 'A':
                arr.add_key(key)
            case 'D':
                arr.del_key(key)
            case '?':
                ans += 'Y' if arr.key_in_table(key) else 'N'
    return ans


if __name__ == "__main__":
    work(Path(__file__), main)
```

Класс HashTable

• __init__: Конструктор класса, который инициализирует пустую хеш-таблицу в виде словаря Python (self.table = {}).

• key_in_table: Проверяет, существует ли указанный key в хеш-таблице. Возвращает True, если ключ существует, и False в противном случае.

• add_key: добавляет новый ключ в таблицу с заданным значением val. Если ключ уже существует, он не добавляется.
• del_key: удаляет указанный key из хеш-таблицы, если он существует.

Функция main служит для обработки команд и взаимодействия с пользователем или другими частями системы.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 6
        TASK NUMBER: 1
                INPUT DATA: (8, ['A', 2], ['A', 5], ['A', 3], ['?', 2], ['?', 4], ['A', 2], ['D', 2], ['?', 2])
                OUTPUT DATA: ['Y', 'N', 'N']
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.5991286137060880 66e-06 секунд	30.96403375 МБ
Верхняя граница диапазона значений входных данных из текста задачи	1.345885128000099212 секунд	31.14453125 МБ

Вывод по задаче:
Хеш-таблица создается объединением хеш-функций с массивом.
Хеш-таблицы обеспечивают очень быстрое выполнение поиска, вставки и удаления.

3 задача. Снова сортировка

Редакционное расстояние между двумя строками – это минимальное количество операций (вставки, удаления и замены символов) для преобразования одной строки в другую. Это мера сходства двух строк. У редакционного расстояния есть применения, например, в вычислительной биологии, обработке текстов на естественном языке и проверке

орфографии. Ваша цель в этой задаче – вычислить расстояние редактирования между двумя строками.

Листинг кода:

```
from pathlib import Path
from lab_6.utils.utils import work
from collections import deque


class HashTableChaining:
    def __init__(self, m):
        self.m = m
        self.p = 1000000007
        self.x = 263
        self.table = [deque() for _ in range(m)]
        self.exist = set()

    def hash_func(self, s):
        h = 0
        for i in range(len(s)):
            h += ord(s[i]) * self.x ** i
        return (h % self.p) % self.m

    def add(self, s):
        if s in self.exist:
            return
        self.exist.add(s)
        key = self.hash_func(s)
        self.table[key].appendleft(s)

    def delete(self, s):
        if s not in self.exist:
            return
        self.exist.remove(s)
        index = self.hash_func(s)
        self.table[index].remove(s)

    def find(self, s):
        return 'yes' if s in self.exist else 'no'

    def check(self, i):
        if 0 <= i < self.m:
            return ' '.join(self.table[i])
        return ''


def main(m, n, *actions):
    arr = HashTableChaining(m)
    ans = []
    for act, arg in actions:
        match act:
            case 'add':
                arr.add(arg)
            case 'del':
                arr.delete(arg)
            case 'find':
                ans.append(arr.find(arg))
            case 'check':
                ans.append(arr.check(arg))
    return ans
```

```
if __name__ == "__main__":
    work(Path(__file__), main)
    a = HashTableChaining(5)
    print(a.hash_func('third'))
```

Класс HashTableChaining

• __init__: Конструктор класса, который инициализирует хеш-таблицу.
    • m: Размер хеш-таблицы.
    • p: Большое простое число, используемое для вычисления хеш-функции.
    • x: Основание для вычисления хеша (обычно выбирается как простое число).
    • table: Список, состоящий из m двусторонних очередей (deque), где каждая очередь будет хранить элементы, имеющие одинаковый хеш.
    • exist: Множество для быстрого хранения и проверки наличия строк.
• hash_func(self, s): Вычисляет хеш для строки s.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 6
        TASK NUMBER: 3
                INPUT DATA: (3, 12, ['check', 0], ['find', 'help'], ['add', 'help'], ['add', 'del'], ['add', 'add'], ['find', 'add'], ['find'
                OUTPUT DATA: ['', 'no', 'yes', 'yes', 'no', '', 'add help', '']
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	0.002114700007950887 секунд	32.921875 МБ
Верхняя граница диапазона значений входных данных из текста задачи	0.32775679999031126 секунд	35.69921875 МБ

Вывод по задаче:

Коллизии нежелательны. Хеш-функция должна свести количество коллизий к минимуму.

Дополнительные задачи

2 задача. Телефонная книга

В этой задаче ваша цель - реализовать простой менеджер телефонной книги. Он должен уметь обрабатывать следующие типы пользовательских запросов:
• add number name – это команда означает, что пользователь добавляет в телефонную книгу человека с именем name и номером телефона number. Если пользователь с таким номером уже существует, то ваш менеджер должен перезаписать соответствующее имя.
• del number – означает, что менеджер должен удалить человека с номером из телефонной книги. Если такого человека нет, то он должен просто игнорировать запрос.
• find number – означает, что пользователь ищет человека с номером телефона number. Менеджер должен ответить соответствующим именем или строкой «not found» (без кавычек), если такого человека в книге нет.

Листинг кода:

```
from pathlib import Path
from lab_6.utils.utils import work
from lab_6.task1.src.task1 import HashTable


class PhoneBook(HashTable):
    def add_key(self, phone, user):
        self.table[phone] = user

    def find_number(self, phone):
        if phone in self.table.keys():
            return self.table[phone]
        return "not found"


def main(n, *actions):
    ans = []
    book = PhoneBook()
    for act in actions:
        print(act)
        match act[0]:
            case 'add':
                book.add_key(act[1], act[2])
            case 'del':
                book.del_key(act[1])
            case 'find':
                ans.append(book.find_number(act[1]))
    return ans
```

```
if __name__ == "__main__":
    work(Path(__file__), main)
```

Этот код реализует простую телефонную книгу, основанную на хэш-таблице. Он определяет класс PhoneBook, который наследуется от класса HashTable. В классе PhoneBook реализованы методы для добавления записи (имя пользователя по номеру телефона) и поиска номера телефона. Метод add_key добавляет новую запись в телефонную книгу, а метод find_number ищет пользователя по номеру телефона и возвращает его имя или сообщение о том, что запись не найдена.
Функция main принимает количество действий и сами действия в виде переменного числа аргументов. Она создает экземпляр телефонной книги и обрабатывает каждое действие, используя конструкцию match для определения типа действия: добавление, удаление или поиск. В зависимости от действия выполняются соответствующие методы телефонной книги. Результаты поиска собираются в список, который затем возвращается.
В блоке, проверяющем, является ли текущий файл основным модулем, вызывается функция work, передавая ей путь к текущему файлу и функцию main.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 6
          TASK NUMBER: 2
                 INPUT DATA: (12, ['add', 911, 'police'], ['add', 76213, 'Mom'], ['add', 17239, 'Bob'], ['find', 76213], ['find', 910], ['find'
                 OUTPUT DATA: ['Mom', 'not found', 'police', 'not found', 'Mom', 'daddy']
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.1099997209385037e-05 секунд	30.236328125 МБ
Верхняя граница диапазона значений входных данных из текста задачи	1.0664576000126544 секунд	46.6484375 МБ

Вывод по задаче:

Хеш-таблицы хорошо подходят для моделирования отношений между объектами.

5 задача. Выборы в США

Как известно, в США президент выбирается не прямым голосованием, а путем двухуровневого голосования. Сначала проводятся выборы в каждом штате и определяется победитель выборов в данном штате. Затем проводятся государственные выборы: на этих выборах каждый штат имеет определенное число голосов — число выборщиков от этого штата. На практике, все выборщики от штата голосуют в соответствии с результатами голосования внутри штата, то есть на заключительной стадии выборов в голосовании участвуют штаты, имеющие различное число голосов. Вам известно за кого проголосовал каждый штат и сколько голосов было отдано данным штатом. Подведите итоги выборов: для каждого из участника голосования определите число отданных за него голосов.

Листинг кода:

```
from pathlib import Path
from collections import OrderedDict
from lab_6.utils.utils import work


def main(*actions):
    a = OrderedDict()
    for candidate, votes in actions:
        if candidate not in a:
            a[candidate] = int(votes)
        else:
            a[candidate] += int(votes)
    a = OrderedDict(sorted(a.items()))
    ans = [f'{pair[0]} {pair[1]}' for pair in list(a.items())]
    return ans


if __name__ == "__main__":
    work(Path(__file__), main)
```

Этот код реализует программу для обработки голосов за кандидатов, используя упорядоченный словарь (OrderedDict) для хранения и сортировки данных о кандидатах и их голосах.
В функции main принимается переменное количество аргументов, представляющих собой пары "кандидат - количество голосов". Программа проходит по этим парам и добавляет или обновляет количество голосов для каждого кандидата в словаре. Если кандидат уже есть в словаре, к его текущему количеству голосов добавляется новое значение. После

обработки всех голосов, словарь сортируется по именам кандидатов. Затем создается список строк, где каждая строка содержит имя кандидата и общее количество его голосов. Этот список возвращается как результат работы функции.
В блоке if __name__ == "__main__": происходит вызов функции work, которая, вероятно, служит для запуска программы с передачей ей текущего файла и функции main в качестве параметров. Это может быть полезно для тестирования или обработки входных данных. Таким образом, код позволяет собрать и отсортировать голоса за кандидатов, формируя итоговый список с результатами.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 6
            TASK NUMBER: 5
                    INPUT DATA: (['McCain', 10], ['McCain', 5], ['Obama', 9], ['Obama', 8], ['McCain', 1])
                    OUTPUT DATA: ['McCain 16', 'Obama 17']
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения	Память
Нижняя граница диапазона значений входных данных из текста задачи	1.6200006939470768e-05 секунд	0.00092315673828125 Б
Верхняя граница диапазона значений входных данных из текста задачи	0.945634567636943934 секунд	0.00011216345214375 Б

Вывод по задаче:
Хеш-таблицы хорошо подходят для обнаружения дубликатов.

Вывод

Основное назначение хеширования — проверка информации. Эта задача важна в огромном количестве случаев. Так как хеш — это уникальный код определенного набора данных, по нему можно понять, соответствует ли

информация ожидаемой. Поэтому программа может хранить хеши вместо образца данных для сравнения.