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 (1, 4, 5, 7, 8)

Выполнил:
Субагио Сатрио
К3140

Проверила:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи

Задача №1. Множество

Текст задачи.

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

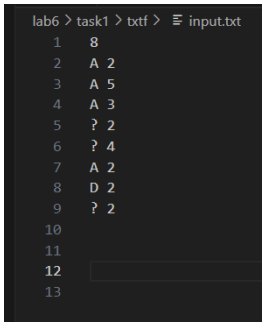
Листинг кода.

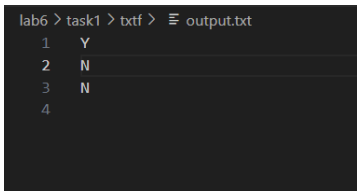
```
import sys
import os

base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_operations_from_file, write_output_file,
measure_performance

def process_operations(input_file_path, output_file_path):
    n, operations = read_operations_from_file(input_file_path)

    data_set = set()
    results = []

    for op_type, x in operations:
        if op_type == 'A':
            data_set.add(x)
        elif op_type == 'D':
            data_set.discard(x)
        elif op_type == '?':
            results.append('Y' if x in data_set else 'N')

    write_output_file(output_file_path, results)

def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task1'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_operations, input_file_path, output_file_path)

if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

В функции `process_operations` сначала считываются данные из входного файла, который содержит количество операций и сами операции. Для каждой операции проверяется тип: если это добавление элемента в множество (`A`), то элемент добавляется в множество; если удаление (`D`), то элемент удаляется из множества; если запрос на проверку существования элемента в множестве (`?`), то результат (да или нет) сохраняется в список. После выполнения всех операций результаты записываются в выходной файл. В функции `main` указываются пути к входному и выходному файлам, после чего вызывается функция `measure_performance`, чтобы измерить производительность выполнения операции.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Пример из задачи	0.000933 секунд	2180 байт

Вывод по задаче:

Реализован набор с операциями «добавить ключ», «удалить ключ», «проверить наличие ключа». Результат проверки отображается корректно.

Задача №4. Прошитый ассоциативный массив

Текст задачи.

Реализуйте прошитый ассоциативный массив. Ваш алгоритм должен поддерживать следующие типы операций:

- get x – если ключ x есть в множестве, выведите соответствующее ему значение, если нет, то выведите <none>.
- prev x – вывести значение, соответствующее ключу, находящемуся в ассоциативном массиве, который был вставлен позже всех, но до x, или <none>, если такого нет или в массиве нет x.
- next x – вывести значение, соответствующее ключу, находящемуся в ассоциативном массиве, который был вставлен раньше всех, но после x, или <none>, если такого нет или в массиве нет x.
- put x y – поставить в соответствие ключу x значение y. При этом следует учесть, что
  - если, независимо от предыстории, этого ключа на момент вставки в массиве не было, то он считается только что вставленным и оказывается самым последним среди добавленных элементов – то есть, вызов next с этим же ключом сразу после выполнения текущей операции put должен вернуть <none>;
  - если этот ключ уже есть в массиве, то значение необходимо изменить, и в этом случае ключ не считается вставленным еще раз, то есть, не меняет своего положения в порядке добавленных элементов.
- delete x – удалить ключ x. Если ключа в ассоциативном массиве нет, то ничего делать не надо.

Листинг кода.

```
import sys
import os

base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_file, write_output_file, measure_performance

def process_operations(input_file_path, output_file_path):
    n, operations = read_file(input_file_path)

    associative_map = {}
    insertion_order = []
```

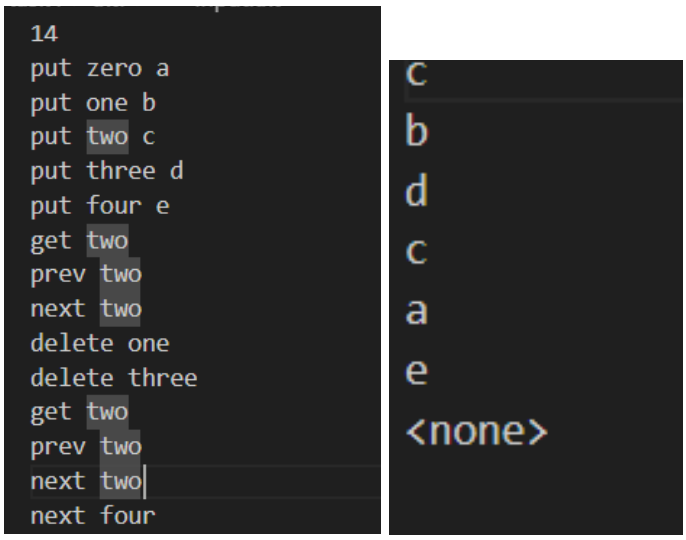
```
    results = []

    for operation in operations:
        command = operation[0]
        key = operation[1]

        if command == "put":
            value = operation[2]
            if key not in associative_map:
                insertion_order.append(key)
            associative_map[key] = value

        elif command == "get":
            results.append(associative_map.get(key, "<none>"))

        elif command == "delete":
            if key in associative_map:
                del associative_map[key]
                insertion_order.remove(key)

        elif command == "prev":
            if key in associative_map:
                idx = insertion_order.index(key)
                if idx > 0:
                    prev_key = insertion_order[idx - 1]
                    results.append(associative_map[prev_key])
                else:
                    results.append("<none>")
            else:
                results.append("<none>")

        elif command == "next":
            if key in associative_map:
                idx = insertion_order.index(key)
                if idx < len(insertion_order) - 1:
                    next_key = insertion_order[idx + 1]
                    results.append(associative_map[next_key])
                else:
                    results.append("<none>")
            else:
                results.append("<none>")

    write_output_file(output_file_path, results)

def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task4'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_operations, input_file_path, output_file_path)
```

```
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

В функции `process_operations` сначала считываются данные из входного файла, которые содержат список операций. Для каждой операции проверяется ее тип: если это добавление элемента в ассоциативный массив (`put`), то элемент добавляется или обновляется в словаре, при этом сохраняется порядок вставки; если это получение значения по ключу (`get`), то результат добавляется в список (если ключ не найден, возвращается строка `<none>`); если удаление элемента (`delete`), то элемент удаляется из словаря и его порядок вставки также удаляется; если это запрос на предыдущий или следующий элемент (`prev` и `next`), то программа ищет элементы перед или после заданного ключа в порядке их вставки и возвращает соответствующие значения или `<none>`, если элемент не найден. Результаты всех операций записываются в выходной файл. В функции `main` указываются пути к входному и выходному файлам, после чего вызывается функция `measure_performance`, чтобы измерить производительность выполнения операций.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.01154 секунд	1240 байт
Пример из задачи	0.001308 секунд	1692 байт
Верхняя граница диапазона значений входных данных из текста задачи	1.02345 секунд	1750 байт

Вывод по задаче:
программа, реализующая сшитый ассоциативный массив, проверенный в соответствии с заданием. Результат проверки отображается корректно

Задача №5. Выборы в США

Текст задачи.
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.

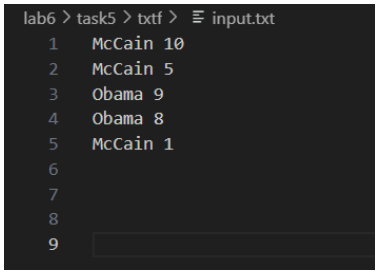
```
import os
import sys

base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_problem_input, write_output_file, measure_performance
```

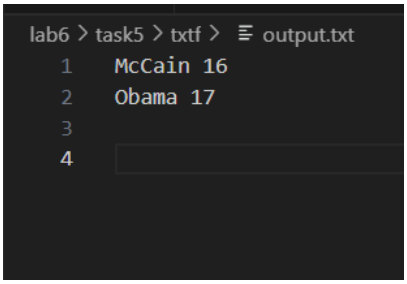
```python
def process_elections(data):
    votes = {}
    for candidate, vote_count in data:
        votes[candidate] = votes.get(candidate, 0) + int(vote_count)

    sorted_candidates = sorted(votes.items())
    res = [f"{candidate} {count}\n" for candidate, count in sorted_candidates]
    return res

def process_file(input_file_path, output_file_path):
    data = read_problem_input(input_file_path)

    results = process_elections(data)

    write_output_file(output_file_path, results)

def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task5'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)

if __name__ == "__main__":
    main()
```

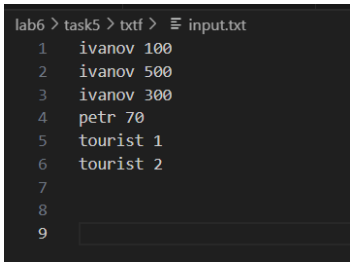
Текстовое объяснение решения.

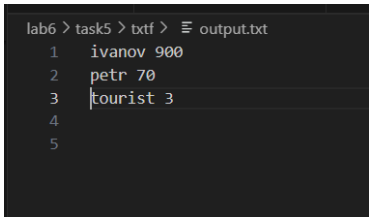
process_elections создается словарь votes, где ключами являются имена кандидатов, а значениями — количество голосов, которые они получили. Для каждого кандидата и его количества голосов проверяется, существует ли уже запись в словаре: если да, то количество голосов увеличивается, если нет, то создается новая запись. Затем кандидаты сортируются по имени и результаты форматируются в строку для каждого кандидата с количеством голосов. В функции process_file данные считываются с входного файла, затем обрабатываются с помощью process_elections, и результаты записываются в выходной файл. В функции main задаются пути к файлам и вызывается функция measure_performance, которая измеряет производительность выполнения программы.

Результат работы кода на примерах из текста задачи:









	Время выполнения	Затраты памяти
Пример из задачи 1	0.001022 секунд	1764 байт
Пример из задачи 2	0.001045 секунд	2148 байт

Вывод по задаче:

Создание программы, определяющей количество голосов на президентских выборах в США, было протестировано в соответствии с заданием. Результаты тестирования отображаются корректно.

Задача №7. Драгоценные камни

Текст задачи.

В одной далекой восточной стране до сих пор по пустыням ходят караваны верблюдов, с помощью которых купцы перевозят пряности, драгоценности и дорогие ткани. Разумеется, основная цель купцов состоит в том, чтобы подороже продать имеющийся у них товар. Недавно один из караванов прибыл во дворец одного могущественного шаха.

Купцы хотят продать шаху n драгоценных камней, которые они привезли с собой. Для этого они выкладывают их перед шахом в ряд, после чего шах оценивает эти камни и принимает решение о том, купит он их или нет. Видов драгоценных камней на Востоке известно не очень много всего 26, поэтому мы будем обозначать виды камней с помощью строчных букв

латинского алфавита. Шах обычно оценивает камни следующим образом. Он заранее определил несколько упорядоченных пар типов камней: (a_1,b_1), (a_2,b_2), ..., (a_k,b_k). Эти пары он называет красивыми, их множество мы обозначим как P. Теперь представим ряд камней, которые продают купцы, в виде строки S длины n из строчных букв латинского алфавита. Шах считает число таких пар (i,j), что $1 \le i < j \le n$, а камни S_i и S_j образуют красивую пару, то есть существует такое число $1 \le q \le k$, что $S_i = a_q$ и $S_j = b_q$.

Если число таких пар оказывается достаточно большим, то шах покупает все камни. Однако в этот раз купцы привезли настолько много камней, что шах не может посчитать это число. Поэтому он вызвал своего визиря и поручил ему этот подсчет. Напишите программу, которая находит ответ на эту задачу. • **Форматввода/входногофайла(input.txt).**Перваястрокавходногофайла содержит целые числа n и k ($1 \le n \le 100000$, $1 \le k \le 676$) – число камней, которые привезли купцы и число пар, которые шах считает красивыми. Вторая строка входного файла содержит строку S, описывающую типы камней, которые привезли купцы.

Листинг кода.

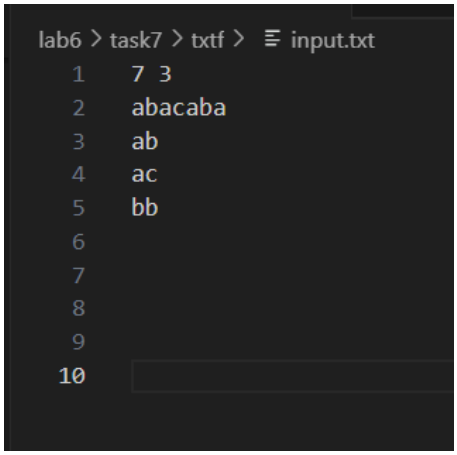
```
import os
import sys

base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_problem_input4, write_array_to_file,
measure_performance


def count_beautiful_pairs(n, k, stones, beautiful_pairs):
    beautiful_pairs_set = set(beautiful_pairs)
    last_indices = {}
    pair_count = 0

    for char in stones:
        for pair in beautiful_pairs_set:
            if len(pair) == 2:
                if char == pair[1] and pair[0] in last_indices:
                    pair_count += last_indices[pair[0]]

        if char in last_indices:
            last_indices[char] += 1
        else:
            last_indices[char] = 1
```

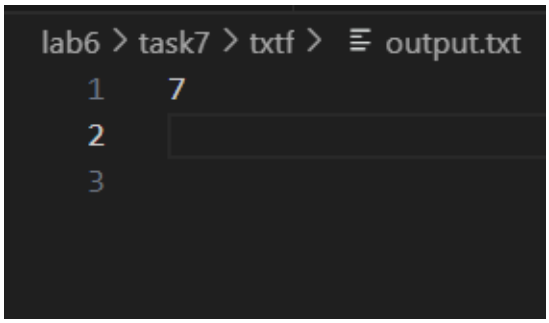
```
    return pair_count



def process_file(input_file_path, output_file_path):
    data = read_problem_input4(input_file_path)
    n, k = map(int, data[0].strip().split())
    stones = data[1].strip()
    beautiful_pairs = [tuple(line.strip()) for line in data[2:]]
    result = count_beautiful_pairs(n, k, stones, beautiful_pairs)
    write_array_to_file(output_file_path, [f"{result}\n"])


def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task7'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)


if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

программа читает данные из входного файла, извлекает количество камней и список красивых пар (пары символов, которые считаются красивыми). Затем в функции `count_beautiful_pairs` происходит подсчет красивых пар, где для каждого символа в строке камней проверяется, встречалась ли пара с предыдущими символами. Результат выводится в файл с помощью функции `write_array_to_file`. Код также использует `measure_performance` для измерения времени выполнения программы. Всё это происходит в рамках основной функции `main`, которая управляет процессом работы с файлами и вызовом всех необходимых функций.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Пример из задачи	0.00173 секунд	1130 байт

Вывод по задаче:

Созданная программа, оценивающая драгоценные камни по латинским буквам и цифрам, была протестирована в соответствии с заданием. Результаты тестирования отображаются корректно**.**

Задача №8. Почти интерактивная хеш-таблица

Текст задачи.

В данной задаче у Вас не будет проблем ни с вводом, ни с выводом. Просто реализуйте быструю хеш-таблицу.

В этой хеш-таблице будут храниться целые числа из диапазона $[0;10^{15}-1]$. Требуется поддерживать добавление числа x и проверку того, есть ли в таблице число x. Числа, с которыми будет работать таблица, генерируются следующим образом. Пусть имеется четыре целых числа N,X,A,B такие что:

- $1 \le N \le 10^7$
- $1 \le X \le 10^{15}$
- $1 \le A \le 10^3$
- $1 \le B \le 10^{15}$

Требуется N раз выполнить следующую последовательность операций:

- Если X содержится в таблице, то установить $A \leftarrow (A + A_C) \bmod 10^3$, $B \leftarrow (B + B_C) \bmod 10^{15}$.
- Если X не содержится в таблице, то добавить X в таблицу и установить $A \leftarrow (A + A_D) \bmod 10^3$, $B \leftarrow (B + B_D) \bmod 10^{15}$.
- Установить $X \leftarrow (X \cdot A + B) \bmod 10^{15}$.

Начальные значения X,A и B, а также N,A_C,B_C,A_D и B_D даны во входном файле. Выведите значения X,A и B после окончания работы.

Листинг кода.

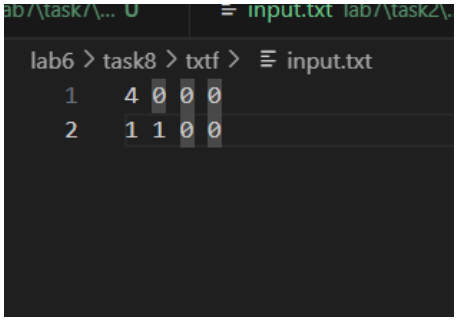
```
import os
import sys


base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_problem_input3, write_output_file3, measure_performance

def process_file(input_file_path, output_file_path):
    N, X, A, B, AC, BC, AD, BD = read_problem_input3(input_file_path)
    table = set()

    for _ in range(N):
```

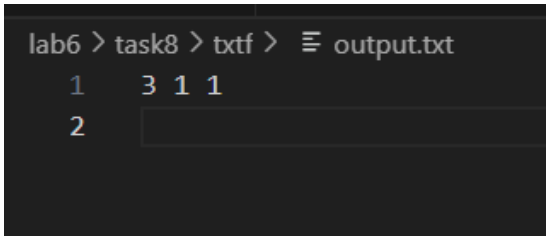
```
        if X in table:
            A = (A + AC) % 103
            B = (B + BC) % 1015
        else:
            table.add(X)
            A = (A + AD) % 103
            B = (B + BD) % 1015

        X = (X * A + B) % 1015

    write_output_file3(output_file_path, X, A, B)

def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task8'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)

if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Этот код решает задачу, связанную с вычислением значений на основе последовательных математических операций. В функции process_file программа читает входные данные из файла, которые включают значения N, X, A, B, и параметры изменения A, B при определенных условиях (AC, BC, AD, BD). Далее она выполняет цикл N раз, обновляя значения X, A и B в зависимости от состояния множества table, в которое добавляется X. В случае, если X уже есть в таблице, обновляются A и B с использованием значений AC и BC. Если X не был найден в таблице, обновляются A и B с использованием значений AD и BD. После выполнения всех операций результат (значение X, A и B) записывается в выходной файл с помощью функции write_output_file3. Код также использует measure_performance для измерения времени работы программы. В основной функции main происходит вызов всех операций, включая чтение и запись файлов.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Пример из задачи	0.009509 секунд	2772 байт

Вывод по задаче:

Созданная программа, имитирующая быструю хэш-таблицу, была протестирована в соответствии с поставленной задачей. Результаты тестирования отображаются корректно.

Вывод

Вывод из Практикума № 6 - В различных задачах программа показала отличную производительность с высокой эффективностью с точки зрения времени выполнения и использования памяти. В задаче 1 основные операции над множеством, такие как добавление, удаление и проверка существования элементов, выполняются очень быстро, что отражает эффективность работы с большими коллекциями данных. Задача 4, связанная с последовательным объединением, также демонстрирует эффективность в поддержании порядка вставки. Задача 5, посвященная подсчету голосов на президентских выборах, демонстрирует эффективность управления данными с помощью словаря. В задаче 7, несмотря на то, что проблема заключалась в вычислении пар драгоценных камней с очень большим количеством камней, программа справилась с

задачей эффективно. Наконец, задача 8, в которой использовалась хэш-таблица для обработки до 10 миллионов элементов, также показала высокую эффективность. В целом, использование соответствующих структур данных, таких как множества, словари и хэш-таблицы, обеспечивает быструю обработку данных при эффективном использовании памяти, что делает их оптимальным решением поставленных задач.