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 8

Выполнила:
Иванова А. Г.
К3141

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задание №6. Фибоначчи возвращается

6 задача. Фибоначчи возвращается

Вам дается последовательность чисел. Для каждого числа определите, является ли оно числом Фибоначчи. Напомним, что числа Фибоначчи определяются, например, так:

$$\begin{aligned} F_0 &= F_1 = 1 \\ F_i &= F_{i-1} + F_{i-2} \text{ для } i \geq 2. \end{aligned} \tag{1}$$

- **Формат ввода / входного файла (input.txt).** Первая строка содержит одно число N ($1 \leq N \leq 10^6$) - количество запросов. Следующие N строк содержат по одному целому числу. При этом соблюдаются следующие ограничения при проверке:
  1. Размер каждого числа не превосходит 5000 цифр в десятичном представлении.
  2. Размер входа не превышает 1 Мб.
- **Формат вывода / выходного файла (output.txt).** Для каждого числа, данного во входном файле, выведите «Yes», если оно является числом Фибоначчи, и «No» в противном случае.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 128 мб. ***Внимание:*** *есть вероятность превышения по памяти, т.к. сами по себе числа Фибоначчи большие. Делайте проверку на память!*

Решение:

```
from utils import read, write


def is_perfect_square(n):
    root = int(n ** 0.5)
    return root * root == n


def is_fibonacci(n):
    if is_perfect_square(5 * n**2 + 4) or is_perfect_square(5 * n**2 - 4):
        return 'Yes'
    return 'No'


def main():
    write(end='')
    data = [line[0] for line in read(type_convert=int)][1:]
```

```
    for line in data:
        write(is_fibonacci(line), to_end=True)


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

Функция is_perfect_square() проверяет, является ли число n точным квадратом. Она вычисляет целую часть корня числа и затем сравнивает результат возведения этого значения в квадрат с исходным числом. Функция использует свойство чисел Фибоначчи, которое гласит, что если n является числом Фибоначчи, то одно из выражений $5n^2 \pm 4$ должно быть точным квадратом. Таким образом, она вызывает функцию is_perfect_square для проверки двух условий и возвращает "Yes", если хотя бы одно из них выполняется, иначе возвращает "No".

Результат работы программы:
Входные данные:

```
8
1
2
3
4
5
6
7
8
```

Выходные данные:

```
Yes
Yes
Yes
No
Yes
No
No
Yes
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab6.task6.src.is_fibonacci import main, is_fibonacci


class TestFibonacci(unittest.TestCase):

    def test_should_recognize_fibonacci_numbers(self):
        # given
        numbers = [0, 1, 8, 21]
        expected_results = ['Yes', 'Yes', 'Yes', 'Yes']
```

```
        # when
        results = [is_fibonacci(n) for n in numbers]

        # then
        self.assertEqual(results, expected_results)

    def test_should_check_large_fibonacci_numbers(self):
        # given
        large_numbers = [144, 987, 1597, 2584]
        expected_results = ['Yes', 'Yes', 'Yes', 'Yes']

        # when
        results = [is_fibonacci(n) for n in large_numbers]

        # then
        self.assertEqual(results, expected_results)

    def test_should_check_non_fibonacci_numbers(self):
        # given
        non_fibonacci_numbers = [13, 30, 55, 89]
        expected_results = ['Yes', 'No', 'Yes', 'Yes']

        # when
        results = [is_fibonacci(n) for n in non_fibonacci_numbers]

        # then
        self.assertEqual(results, expected_results)

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 128

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:
В ходе решения данной задачи мы реализовали алгоритм, который проверяет каждое число на принадлежность к числам Фибоначчи и выводит результаты.

Дополнительные задачи

Задание №1. Множество

1 задача. Множество

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

- **Формат входного файла (input.txt).** В первой строке входного файла находится строго положительное целое число операций N, не превышающее $5 \cdot 10^5$. В каждой из последующих N строк находится одна из следующих операций:
  - A x – добавить элемент x в множество. Если элемент уже есть в множестве, то ничего делать не надо.
  - D x – удалить элемент x. Если элемента x нет, то ничего делать не надо.
  - ? x – если ключ x есть в множестве, выведите «Y», если нет, то выведите «N».

  Аргументы указанных выше операций – **целые числа**, не превышающие по модулю 10^{18}.
- **Формат выходного файла (output.txt).** Выведите последовательно результат выполнения всех операций «?». Следуйте формату выходного файла из примера.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

Решение:

```
from utils import read, write


def operations(data):
    result = []
    current_set = set()
    for line in data:
        operation, x = line
        if operation == 'A':
            current_set.add(x)
        elif operation == 'D':
            current_set.discard(x)
        elif operation == '?':
            if x in current_set:
                result.append('Y')
            else:
                result.append('N')
    return result
```

```
def main():
   write(end='')
   array = read(type_convert=str)
   result = operations(array)
   for line in result:
       write(line, to_end=True)


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

Алгоритм:

1) Создается пустой список result, который будет хранить результаты запросов, и пустое множество current_set, которое будет использоваться для хранения уникальных элементов.
2) Цикл проходит по каждому элементу списка data. Каждый элемент представляет собой кортеж из двух значений: операции (operation) и числа (x). Эти два значения распаковываются в переменные operation и x.
3) Операция 'A' добавляет элемент в множество
4) Операция 'D' удаляет элемент из множества
5) Операция '?' проверяет наличие элемента в множестве. та операция проверяет, содержится ли элемент x в множестве current_set. Если да, то в список result добавляется строка "Y", иначе — "N"

Результат работы программы:

Входные данные:

```
A 2
A 5
A 3
? 2
? 4
A 2
D 2
? 2
```

Выходные данные:

```
Y
N
N
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab6.task1.src.operations import main, operations
```

```
class TestOperations(unittest.TestCase):

    def test_should_add_and_query_element(self):
        # given
        data = [
            ('A', 1),
            ('?', 1),
            ('D', 1),
            ('?', 1)
        ]
        expected_result = ['Y', 'N']

        # when
        result = operations(data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_handle_multiple_elements(self):
        # given
        data = [
            ('A', 1),
            ('A', 2),
            ('A', 3),
            ('?', 1),
            ('?', 2),
            ('?', 3),
            ('D', 2),
            ('?', 2),
            ('?', 3)
        ]
        expected_result = ['Y', 'Y', 'Y', 'N', 'Y']

        # when
        result = operations(data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)
```

```
        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм, который позволяет эффективно обрабатывать команды добавления, удаления и проверки наличия элементов в множестве.

Задание №2. Телефонная книга

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.
- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится число N ($1 \leq N \leq 10^5$) - количество запросов. Далее следуют N строк, каждая из которых содержит один запрос в формате, описанном выше.

  Все номера телефонов состоят из десятичных цифр, в них нет нулей в начале номера, и каждый состоит не более чем из 7 цифр. Все имена представляют собой непустые строки из латинских букв, каждая из которых имеет длину не более 15. Гарантируется при проверке, что не будет человека с именем «not found».
- **Формат вывода / выходного файла (output.txt).** Выведите результат каждого поискового запроса `find` – имя, соответствующее номеру телефона, или «not found» (без кавычек), если в телефонной книге нет человека с таким номером телефона. Выведите по одному результату в каждой строке в том же порядке, как были заданы запросы типа `find` во входных данных.
- Ограничение по времени. 6 сек.
- Ограничение по памяти. 512 мб.

Решение:

```
from utils import write, read


def phone_book(data):
    book = {}
    results = []

    for command in data:
        if command[0] == "add":
            number, name = command[1], command[2]
            book[number] = name
```

```
        elif command[0] == "del":
            number = command[1]
            book.pop(number, None)
        elif command[0] == "find":
            number = command[1]
            results.append(book.get(number, "not found"))
    return results


def main():
    write(end='')
    array = read(type_convert=str)
    result = phone_book(array)
    for line in result:
        write(line, to_end=True)


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

Алгоритм:

1. Создаются две структуры данных:
    1) book: пустой словарь, который будет служить телефонной книгой. Ключи словаря будут номерами телефонов, а значения - именами владельцев.
    2) results: пустой список, куда будут добавляться результаты поиска.
2. Если первая часть команды равна "add", это означает, что нужно добавить новую запись в телефонную книгу. Вторая часть команды содержит номер телефона, третья — имя владельца. Номер телефона становится ключом в словаре book, а имя — значением.
3. Если первая часть команды равна "del", это означает, что нужно удалить запись из телефонной книги. Вторая часть команды содержит номер телефона, который необходимо удалить. Метод .pop() удаляет указанный ключ из словаря. Второй аргумент None указывает, что если ключа не существует, ошибка не возникает.
4. Если первая часть команды равна "find", это означает, что нужно найти имя владельца по номеру телефона. Вторая часть команды содержит номер телефона. Метод .get() ищет указанное значение в словаре. Если ключ найден, возвращается соответствующее значение; если ключ отсутствует, возвращается строка "not found". Результат поиска добавляется в список results.

Результат работы программы:

Входные данные:

```
add 911 police
add 76213 Mom
add 17239 Bob
find 76213
find 910
find 911
del 910
del 911
find 911
find 76213
add 76213 daddy
find 76213
```

Выходные данные:

```
Mom
not found
police
not found
Mom
daddy
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab6.task2.src.phone_book import main, phone_book


class TestPhoneBook(unittest.TestCase):

    def test_should_add_and_find_number(self):
        # given
        data = [
            ["add", "911", "police"],
            ["find", "911"]
        ]
        expected_result = ["police"]

        # when
        result = phone_book(data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_delete_and_find_number(self):
        # given
        data = [
            ["add", "1234567890", "John Doe"],
            ["del", "1234567890"],
            ["find", "1234567890"]
        ]
        expected_result = ["not found"]

        # when
        result = phone_book(data)
```

```
        # then
        self.assertEqual(result, expected_result)

    def test_should_handle_multiple_commands(self):
        # given
        data = [
            ["add", "911", "police"],
            ["add", "112", "ambulance"],
            ["find", "911"],
            ["find", "112"],
            ["del", "112"],
            ["find", "112"]
        ]
        expected_result = ["police", "ambulance", "not found"]

        # when
        result = phone_book(data)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_time_data(self):
        # given
        expected_time = 6

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 512

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:

В ходе решения данной задачи мы создали функцию phone_book, которая реализует простую телефонную книгу, позволяющую добавлять, удалять и искать записи по номерам телефонов.

Задание №4. Прошитый ассоциативный массив

4 задача. Прошитый ассоциативный массив

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.
- **Формат входного файла (input.txt).** В первой строке входного файла находится строго положительное целое число операций N, не превышающее $5 \cdot 10^5$. В каждой из последующих N строк находится одна из приведенных выше операций. Ключи и значения операций - строки из латинских букв длиной не менее одного и не более 20 символов.
- **Формат выходного файла (output.txt).** Выведите последовательно результат выполнения всех операций get, prev, next. Следуйте формату выходного файла из примера.
- Ограничение по времени. 4 сек.
- Ограничение по памяти. 256 мб.

Решение:

```
from utils import read, write
from collections import OrderedDict
```

```
def associative_array(commands):
    assoc_array = OrderedDict()
    results = []

    for parts in commands:
        command = parts[0]

        if command == "put":
            x, y = parts[1], parts[2]
            if x in assoc_array:
                assoc_array[x] = y
            else:
                assoc_array[x] = y

        elif command == "get":
            x = parts[1]
            results.append(assoc_array.get(x, "<none>"))

        elif command == "prev":
            x = parts[1]
            if x in assoc_array:
                keys = list(assoc_array.keys())
                idx = keys.index(x)
                if idx > 0:
                    results.append(assoc_array[keys[idx - 1]])
                else:
                    results.append("<none>")
            else:
                results.append("<none>")

        elif command == "next":
            x = parts[1]
            if x in assoc_array:
                keys = list(assoc_array.keys())
                idx = keys.index(x)
                if idx < len(keys) - 1:
                    results.append(assoc_array[keys[idx + 1]])
                else:
                    results.append("<none>")
            else:
                results.append("<none>")

        elif command == "delete":
            x = parts[1]
            assoc_array.pop(x, None)

    return results


def main():
    write(end='')
    array = read(type_convert=str)
    result = associative_array(array)
    for line in result:
        write(line, to_end=True)
```

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

Алгоритм:

1. Здесь создаются две структуры данных:
    1) assoc_array: упорядоченный словарь (OrderedDict), который используется для хранения ассоциативного массива.
    2) results: пустой список, в котором будут накапливаться результаты выполнения команд.
2. Команда "put" добавляет пару ключ-значение в ассоциативный массив. Если ключ уже существует, обновляется его значение, иначе пара добавляется в массив.
3. Команда "get" ищет значение по ключу. Если ключ найден, его значение добавляется в список results, иначе добавляется строка "<none>".
4. Команда "prev" ищет предыдущий элемент относительно указанного ключа. Если ключ существует и не является первым элементом, результат предыдущего элемента добавляется в results, иначе добавляется "<none>".
5. Команда "next" аналогична команде "prev", но ищет следующий элемент после указанного ключа. Если ключ существует и не является последним элементом, результат следующего элемента добавляется в results, иначе добавляется "<none>".
6. Команда "delete" удаляет указанную пару ключ-значение из ассоциативного массива. Если ключ не существует, ничего не происходит благодаря использованию метода .pop() с параметром по умолчанию None.

Результат работы программы:

Входные данные:

```
put zero a
put one b
put two c
put three d
put four e
get two
prev two
next two
delete one
delete three
get two
```

```
prev two
next two
next four
```

Выходные данные:

```
c
b
d
c
a
e
<none>
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab6.task4.src.associative_array import main, associative_array


class TestAssociativeArray(unittest.TestCase):

    def test_should_put_and_get_values(self):
        # given
        commands = [
            ["put", "key1", "value1"],
            ["get", "key1"],
            ["put", "key1", "new_value1"],
            ["get", "key1"]
        ]
        expected_results = ["value1", "new_value1"]

        # when
        results = associative_array(commands)

        # then
        self.assertEqual(results, expected_results)

    def test_should_find_previous_and_next_elements(self):
        # given
        commands = [
            ["put", "key1", "value1"],
            ["put", "key2", "value2"],
            ["put", "key3", "value3"],
            ["prev", "key2"],
            ["next", "key2"]
        ]
        expected_results = ["value1", "value3"]

        # when
        results = associative_array(commands)

        # then
        self.assertEqual(results, expected_results)

    def test_should_delete_key_and_check_its_absence(self):
        # given
```

```
        commands = [
            ["put", "key1", "value1"],
            ["put", "key2", "value2"],
            ["delete", "key1"],
            ["get", "key1"],
            ["get", "key2"]
        ]
        expected_results = ["<none>", "value2"]

        # when
        results = associative_array(commands)

        # then
        self.assertEqual(results, expected_results)

    def test_should_check_time_data(self):
        # given
        expected_time = 4

        # when
        time = time_data(main)

        # then
        self.assertLess(time, expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
        self.assertLess(current, expected_memory)
        self.assertLess(peak, expected_memory)


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

Вывод по задаче:

В ходе решения данной задачи мы реализовали функцию, которая предоставляет возможность выполнять различные операции над ассоциативным массивом, сохраняя порядок вставки ключей.

Вывод

В ходе выполнения лабораторной работы №6 мы изучили множества, словари, хеш-таблицы и хеш-функции и поработали с ними.