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 23

Выполнил: Хабиби Я.

К3140

Проверил: Афанасьев А. В.

Санкт-Петербург 2024 г.

# Содержание отчета

[**Содержание отчета** **2**](#_Toc25893)

[**Задачи по варианту** **3**](#_Toc25894)

[Задание №6. Фибоначчи возвращается 3](#_Toc25895)

[**Дополнительные задачи** **7**](#_Toc25896)

[Задание №1. Множество 7](#_Toc25897)

[Задание №2. Телефонная книга 11](#_Toc25898)

[Задание №4. Прошитый ассоциативный массив 16](#_Toc25899)

[**Вывод** **22**](#_Toc25900)

**Задачи №3**

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

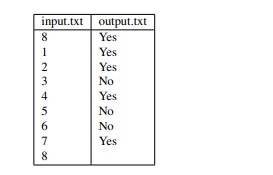
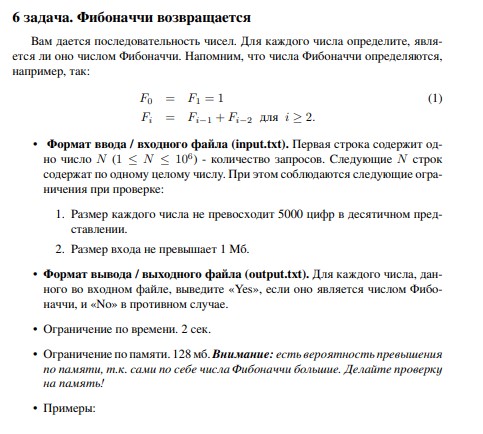
Решение

class HashTable:  
 def \_\_init\_\_(self, bucket\_count):  
 self.bucket\_count = bucket\_count  
 self.table = [[] for \_ in range(bucket\_count)]  
  
 def \_hash\_function(self, string):  
 *"""  
 Полиномиальная хеш-функция.  
 """* p = 1000000007  
 x = 263  
 hash\_value = 0  
 for i, c in enumerate(string):  
 hash\_value += ord(c) \* (x \*\* i)  
 hash\_value %= p  
 return hash\_value % self.bucket\_count  
  
 def add(self, string):  
 *"""  
 Добавление строки в хеш-таблицу.  
 """* index = self.\_hash\_function(string)  
 if string not in self.table[index]:  
 self.table[index].insert(0, string)  
  
 def delete(self, string):  
 *"""  
 Удаление строки из хеш-таблицы.  
 """* index = self.\_hash\_function(string)  
 if string in self.table[index]:  
 self.table[index].remove(string)  
  
 def find(self, string):  
 *"""  
 Проверка наличия строки в хеш-таблице.  
 """* index = self.\_hash\_function(string)  
 return "yes" if string in self.table[index] else "no"  
  
 def check(self, i):  
 *"""  
 Возвращает содержимое сегмента хеш-таблицы в обратном порядке добавления.  
 """* if 0 <= i < self.bucket\_count:  
 return " ".join(self.table[i][::-1]) # Возвращаем список в обратном порядке  
 return ""  
  
  
def process\_hash\_table(data):  
 *"""  
 Обработка запросов для хеш-таблицы.  
 """* bucket\_count = int(data[0])  
 queries = data[1:]  
  
 # Создаём хеш-таблицу  
 hash\_table = HashTable(bucket\_count)  
 results = []  
  
 # Обрабатываем запросы  
 for query in queries:  
 parts = query.split()  
 command = parts[0]  
  
 if command == "add":  
 hash\_table.add(parts[1])  
 elif command == "del":  
 hash\_table.delete(parts[1])  
 elif command == "find":  
 results.append(hash\_table.find(parts[1]))  
 elif command == "check":  
 results.append(hash\_table.check(int(parts[1])))  
  
 return results

Тесты:

class TestHashChains(unittest.TestCase):  
  
  
  
 def test\_add\_find\_and\_delete(self):  
 # given  
 input\_data = [  
 "3", # Количество сегментов  
 "add hello",  
 "add world",  
 "find hello",  
 "find world",  
 "del hello",  
 "find hello"  
 ]  
 expected = ["yes", "yes", "no"]  
  
 # when  
 result, elapsed\_time, peak\_memory = measure\_performance(process\_hash\_table, input\_data)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 7, "Execution time exceeded 7 seconds")  
 self.assertLessEqual(peak\_memory, 512, "Memory usage exceeded 512 MB")  
  
 def test\_multiple\_operations(self):  
 # given  
 input\_data = [  
 "4", # Количество сегментов  
 "add test",  
 "add test",  
 "find test",  
 "del test",  
 "find test",  
 "check 1"  
 ]  
 expected = ["yes", "no", ""]  
  
 # when  
 result, elapsed\_time, peak\_memory = measure\_performance(process\_hash\_table, input\_data)  
  
 # then  
 self.assertEqual(result, expected)  
 self.assertLessEqual(elapsed\_time, 7, "Execution time exceeded 7 seconds")  
 self.assertLessEqual(peak\_memory, 512, "Memory usage exceeded 512 MB")

## Задание №6. Фибоначчи возвращается



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 5n2±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): |

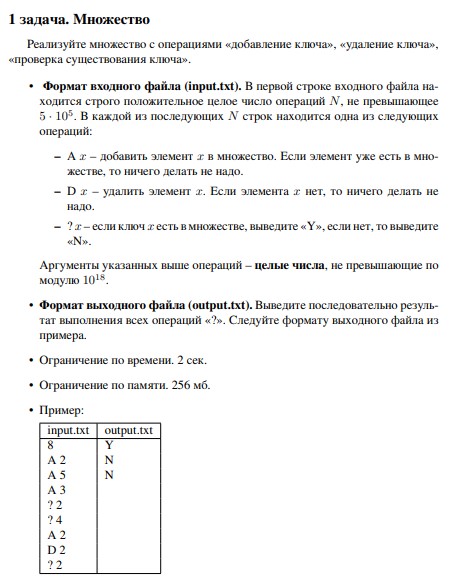
|  |
| --- |
| # given |
| numbers = [0, 1, 8, 21] |
| expected\_results = ['Yes', 'Yes', 'Yes', 'Yes'] |
|  |
| # when |
| results = [is\_fibonacci(n) for n in numbers] |
|  |
| # then |
| self.assertEqual(results, expected\_results) |
|  |
| def test\_should\_check\_large\_fibonacci\_numbers(self): |
| # 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



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 |

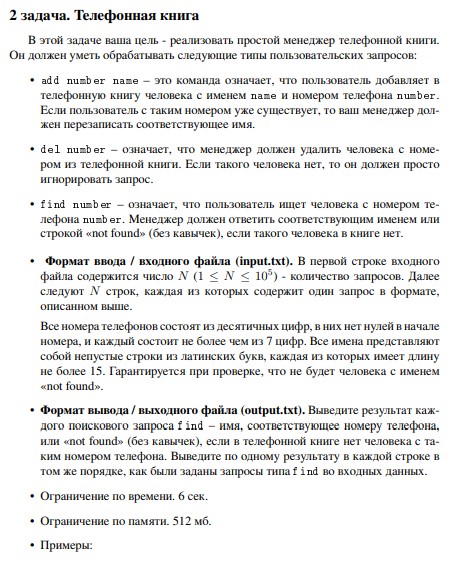
Тесты:

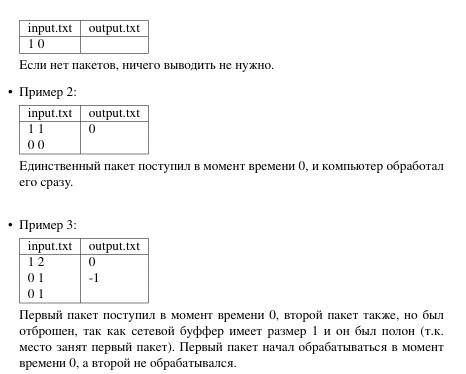
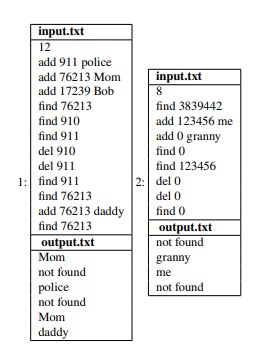
|  |
| --- |
| import unittest |
| from utils import memory\_data, time\_data |
| from lab6.task1.src.operations import main, operations |
|  |
| class TestOperations(unittest.TestCase): |
|  |
| def test\_should\_add\_and\_query\_element(self): |
| # given |
| data = [ |
| ('A', 1), |
| ('?', 1), |
| ('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





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 |

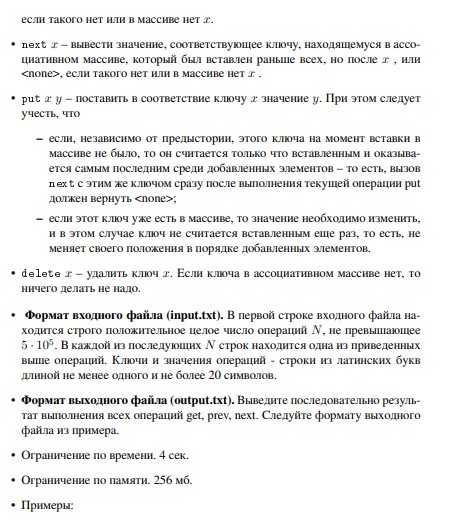
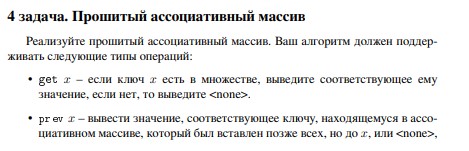
Тесты:

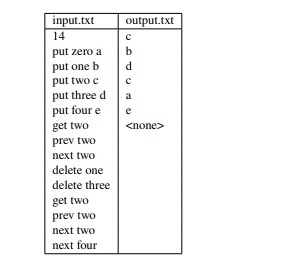
|  |
| --- |
| import unittest |
| from utils import memory\_data, time\_data |
| from lab6.task2.src.phone\_book import main, phone\_book |
|  |
| class TestPhoneBook(unittest.TestCase): |
|  |
| def test\_should\_add\_and\_find\_number(self): |
| # given |
| data = [ |
| ["add", "911", "police"], |
| ["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





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.