МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Очереди с приоритетом. Параллельная обработка

Студентка гр.1381 _____ Рымарь М.И.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2022

Цель работы.

Изучить такую структуру данных, как очереди. Реализовать очередь с приоритетом (min/max кучу) для решения задания.

Задание.

На вход программе подается число процессоров n и последовательность чисел t0 , . . . , tm−1 , где ti — время, необходимое на обработку i-й задачи.

Требуется для каждой задачи определить, какой процессор и в какое время начнёт её обрабатывать, предполагая, что каждая задача поступает на обработку первому освободившемуся процессору.

Примечание #1: в работе необходимо использовать очередь с приоритетом (т.е. min или max-кучу)

Примечание #2: в работе запрещено использовать библиотечные реализации алгоритмов и структур.

Формат входа

Первая строка входа содержит числа n и m. Вторая содержит числа t0 , . . . , tm−1 , где ti — время, необходимое на обработку i-й задачи. Считаем, что и процессоры, и задачи нумеруются с нуля.

Формат выхода

Выход должен содержать ровно m строк: i-я (считая с нуля) строка должна содержать номер процессора, который получит i-ю задачу на обработку, и время, когда это произойдёт.

Выполнение работы.

Для решения поставленной задачи было реализовано два класса Heap и Processor. Первый класс — это очередь с приоритетом. Реализован метод инициализации, метод insert() – добавление элемента в очередь, метод getParent() – получение индекса родителя, методы getLeft() и getRight() – получение индексов левого и правого ребёнка, соответственно. Также написано два метода, реализующие просеивание кучи вверх и вниз – siftUp() и siftDown(). Для

извлечения минимального элемента из кучи был создан метод extractMin(). Второй класс создан для хранения информации. Он имеет два атрибута, для доступа к которым написаны «геттеры» id() и time(). Реализован метод инициализации, увеличения времени, переопределён метод сравнения __lt__.

В функции main() происходит считывание данных, введённых пользователем. Далее создаётся экземпляр класса Heap, затем с помощью цикла экземпляры класса Processor добавляются в очередь с приоритетом. В списке result хранятся кортежи идентификатора процессора, который получит задачу на обработку и время, когда это произойдёт. Далее цикл for выводит результат работы программы в консоль.

Код программы из файла main.py представлен в приложении А.

Тестирование.

Для тестирования используется фреймворк unittest. В файле с тестами была написана функция main_tester(), реализующая те же функции, то и функция main() в файле main. Функция была написана для удобства тестирования. Тесты рассматривают несколько очередей с приоритетом. Результаты тестирования представлены на рисунке 1. Файл с тестированием *test.py* представлен в приложении А.

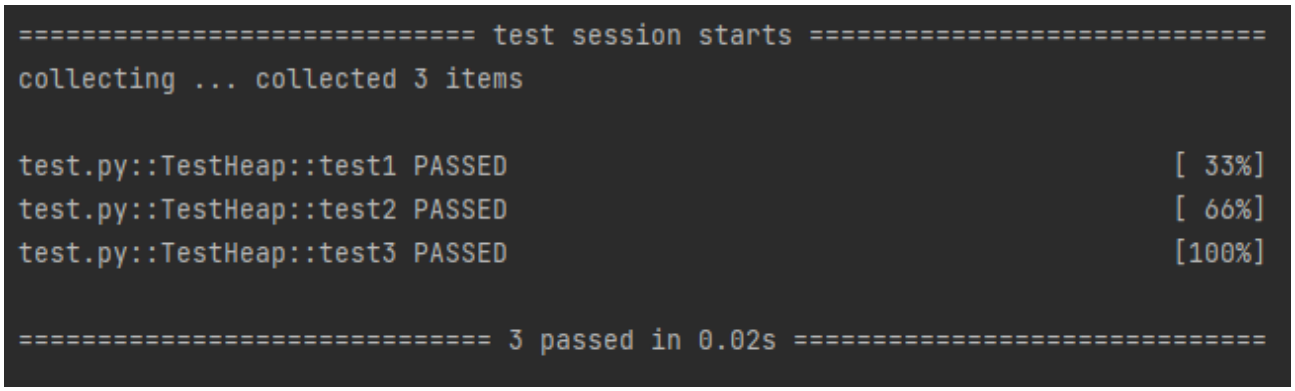

Рисунок 1 – Результаты тестирования

Выводы.

В ходе выполнения лабораторной работы была изучена такая структура данных, как очередь с приоритетом. Структура данных балы использована для решения поставленной задачи. С помощью юнит тестов была проверена корректность реализации.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММ

Название файла: *main.py*

```
class Heap:
    def __init__(self):
        self.__size = 0
        self.__MAX_SIZE = 10 ** 5
        self.__heap = [None] * self.__MAX_SIZE


    def getParent(self, index):
        if index == 0:
            return 0
        return (index - 1) // 2

    def getRight(self, index):
        return 2 * index + 2

    def getLeft(self, index):
        return 2 * index + 1

    def insert(self, elem):
        if self.__size == self.__MAX_SIZE:
            return -1
        self.__heap[self.__size] = elem
        self.siftUp(self.__size)
        self.__size = self.__size + 1

    def siftUp(self, index):
        if index < 0 or index >= self.__MAX_SIZE:
            raise IndexError("Incorrect index")

        parent = self.getParent(index)
        while index > 0 and self.__heap[index] < self.__heap[parent]:
            self.__heap[parent], self.__heap[index] = self.__heap[index],
self.__heap[parent]
            index = parent
            parent = self.getParent(index)

    def siftDown(self, index):
        if index < 0 or index >= self.__MAX_SIZE:
            raise IndexError("Incorrect index")

        left = self.getLeft(index)
        right = self.getRight(index)

        if left >= self.__size and right >= self.__size:
            return None

        if right >= self.__size:
            if self.__heap[left] < self.__heap[index]:
                minIndex = left
            else:
```

```
                minIndex = index

        else:
            if self.__heap[left] < self.__heap[right]:
                minIndex = left
            else:
                minIndex = right

            if self.__heap[minIndex] < self.__heap[index]:
                minIndex = minIndex
            else:
                minIndex = index

        if minIndex != index:
            self.__heap[minIndex], self.__heap[index] = self.__heap[index], self.__heap[minIndex]
            self.siftDown(minIndex)

    def extract_min(self):
        minElem = self.__heap[0]
        self.__heap[0], self.__heap[self.__size - 1] = self.__heap[self.__size - 1], None
        self.__size -= 1
        self.siftDown(0)
        return minElem


class Processor:
    def __init__(self, id, time=0):
        self.__id = id
        self.__time = time

    def time(self):
        return self.__time

    def id(self):
        return self.__id

    def increase(self, time):
        self.__time += time

    def __lt__(self, other):
        if self.time() == other.time():
            return self.id() < other.id()
        return self.time() < other.time()

    def __gt__(self, other):
        if self.time == other.time:
            return self.id > other.id
        return self.time > other.time


def main():
    n, m = map(int, input().split())
    time_array = list(map(int, input().split()))

    heap = Heap()
```

```
    for i in range(n):
        heap.insert(Processor(i))

    result = list()
    for time in time_array:
        min_element = heap.extract_min()
        if min_element:
            result.append((min_element.id(), min_element.time()))
            min_element.increase(time)
            heap.insert(min_element)

    for i in result:
        print(*i)

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

Название файла: *test.py*

```
import unittest
from main import *

def main_tester(data, other_data):
    n, m = map(int, data.split())
    time_array = list(map(int, other_data.split()))

    heap = Heap()
    for i in range(n):
        heap.insert(Processor(i))

    result = list()
    for time in time_array:
        min_element = heap.extract_min()
        if min_element:
            result.append((min_element.id(), min_element.time()))
            min_element.increase(time)
            heap.insert(min_element)

    return result

class TestHeap(unittest.TestCase):

    def test1(self):
        self.assertEqual(main_tester("2 5", "1 2 3 4 5"), [(0, 0), (1,
0), (0, 1), (1, 2), (0, 4)])

    def test2(self):
        self.assertEqual(main_tester("2 0", ""), [])

    def test3(self):
        self.assertEqual(main_tester("3 1", "10"), [(0, 0)])
```