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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Очереди с приоритетом. Параллельная обработка.

Студент гр. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург

2021

Цель работы.

Изучить структуру данных – очередь с приоритетом. Научиться с её помощью реализовывать сортировку и другие задач.

Задание.

Параллельная обработка. Python

На вход программе подается число процессоров n и последовательность чисел t_0 , . . . , t_{m-1} , где t_i — 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: в работе запрещено использовать библиотечные реализации алгоритмов и структур

Формат входа

Первая строка входа содержит числа n и m. Вторая содержит числа t_0 , . . . , t_{m-1} , где t_i — 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.

Ограничения

$1 \le n \le 10\ 5$; $1 \le m \le 10\ 5$; $0 \le t\ i \le 10\ 9$

Пример:

Вход:

2 5

1 2 3 4 5

Выход:

0 0

1 0

0 1

1 2

0 4

Выполнение работы.

Программа выполнена на языке программирования Python.

Реализована мин-куча (очередь с приоритетом, элементы кучи расположены по возрастанию от первого к последнему) на основе массива (*class Heap).*

Элемент массива (кучи) являет собой кортеж (tuple) из двух элементов, где первый – индекс процессора, а второй – время, пройденное с начала обработки задач процессорами.

Класс кучи содержит следующие методы:

__init__(self, n); - конструктор класса, в нём инициализируются поля (максимальный размер массива, количество элементов в нём на текущий момент и сам массив кучи).

get_parent(index); - метод, возвращающий индекс родителя элемента, индекс которого передаётся методу в качестве аргумента.

get_left_child(index), get_right_child(index); - методы, возвращающие индекс «ребенка» элемента (правого, левого), индекс которого передаётся методу в качестве аргумента.

insert(self, element); - добавляет элемент в кучу, записывая его в конец и вызывая метод просеивания элемента наверх очереди.

extract_min(self); - метод, который достаёт из кучи верхний (минимальный) элемент.

Элемент считается минимальным, если его второй элемент пары (пара – кортеж из двух элементов, ячейка массива кучи) является наименьшим. Если такое значение не единственное, сравнение выполняется по первому элементу пары.

sift_up(self, index); - метод просеивания элемента кучи вверх.

sift_down(self, index); - метод просеивания элемента кучи вниз.

В функции *main* считываются из потока ввода необходимые данные, создаётся объект класса *Heap* (создаётся куча). В циклах *for* происходит начальное заполнение кучи и дальнейшая работа с ней (вызов методов в цикле).

Остальные функции предназначены для тестирования.

Тестирование.

Таблица 1. Результат тестирования.

№	Входные данные	Результат	Комментарий
1	2 5 1 2 3 4 5	0 0 1 0 0 1 1 2 0 4	Верно
2	4 20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	0 0 1 0 2 0 3 0 0 1 1 1 2 1 3 1 0 2 1 2 2 2 3 2 0 3 1 3	Верно

		2 3 3 3 0 4 1 4 2 4 3 4	
3	1 1 0	0 0	Верно
4	3 6 100 100 100 100 100 100	0 0 1 0 2 0 0 100 1 100 2 100	Верно
5	2 5 50 40 30 20 10	0 0 1 0 1 40 0 50 0 70	Верно

Вывод.

Была изучена структура данных – очередь с приоритетом. Получены навыки реализации с её помощью сортировки и иных задач.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММ

Файл lb3.py

```
class Heap:
    def __init__(self, n):
        self.length = n
        self.heap = [None] * self.length
        self.size = 0

    @staticmethod
    def get_parent(index):
        return (index - 1) // 2

    @staticmethod
    def get_left_child(index):
        return 2 * index + 1

    @staticmethod
    def get_right_child(index):
        return 2 * index + 2

    def insert(self, element):
        if self.size == self.length:
            return -1
        self.heap[self.size] = element
        self.sift_up(self.size)
        self.size += 1

    def extract_min(self):
        max_element = self.heap[0]
        self.heap[0], self.heap[self.size-1] = self.heap[self.size-
1], None
        self.size -= 1
        self.sift_down(0)
        return max_element

    def sift_up(self, index):
        parent = self.get_parent(index)
```

```
            while index > 0 and (self.heap[parent][1] >
self.heap[index][1]\
                    or self.heap[parent][1] == self.heap[index][1] and
self.heap[parent][0] > self.heap[index][0]):
                self.heap[parent], self.heap[index] = self.heap[index],
self.heap[parent]
                index = parent
                parent = self.get_parent(index)


        def sift_down(self, index):
            left = self.get_left_child(index)
            right = self.get_right_child(index)
            if left >= self.size and right >= self.size:
                return
            if right >= self.size:
                if self.heap[left][1] < self.heap[index][1]:
                    max_index = left
                else:
                    if self.heap[left][1] == self.heap[index][1]:
                        max_index = left if self.heap[left][0] <
self.heap[index][0] else index
                    else:
                        max_index = index
            else:
                if self.heap[left][1] < self.heap[right][1]:
                    max_index = left
                else:
                    if self.heap[left][1] == self.heap[right][1]:
                        max_index = left if self.heap[left][0] <
self.heap[right][0] else right
                    else:
                        max_index = right
                if self.heap[max_index][1] < self.heap[index][1]:
                    max_index = max_index
                else:
                    if self.heap[max_index][1] == self.heap[index][1]:
                        max_index = max_index if self.heap[max_index][0]
< self.heap[index][0] else index
```

```
                    else:
                        max_index = index
            if max_index != index:
                self.heap[max_index], self.heap[index] = self.heap[index], self.heap[max_index]
                self.sift_down(max_index)


        def __str__(self):
            return str(self.heap)

    def main():
        n, m = map(int, input().split())
        min_heap = Heap(n)
        for index_proc in range(n):
            min_heap.insert((index_proc, 0))
           # print(min_heap)
        for time in input().split():
            min_element = min_heap.extract_min()
            print(*min_element)
            min_heap.insert((min_element[0], min_element[1] + int(time)))
           # print(min_heap)

    def pattern_test_for_heap(n, m, array, answer):
        result = []
        min_heap = Heap(n)
        for index_proc in range(n):
            min_heap.insert((index_proc, 0))
        for time in array:
            min_element = min_heap.extract_min()
            print(*min_element)
            result.append(min_element)
            min_heap.insert((min_element[0], min_element[1] + int(time)))
        assert result == answer, "\nTest: {}\nGot: {}\nExpected: {}" .format(array, result, answer)


    def test_for_heap():
        test_n = 4
```

```
        test_m = 20
        test_array = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1]
        answer = [(0, 0), (1, 0), (2, 0), (3, 0),(0, 1), (1, 1), (2, 1),
(3, 1),
                  (0, 2), (1, 2), (2, 2), (3, 2),(0, 3), (1, 3), (2, 3),
(3, 3),
                  (0, 4), (1, 4), (2, 4), (3, 4)]
        pattern_test_for_heap(test_n, test_m, test_array, answer)

        test_n = 1
        test_m = 1
        test_array = [0]
        answer = [(0, 0)]
        pattern_test_for_heap(test_n, test_m, test_array, answer)

        test_n = 0
        test_m = 0
        test_array = []
        answer = []
        pattern_test_for_heap(test_n, test_m, test_array, answer)

        test_n = 3
        test_m = 6
        test_array = [100, 100, 100, 100, 100, 100]
        answer = [(0, 0), (1, 0), (2, 0), (0, 100), (1, 100), (2, 100)]
        pattern_test_for_heap(test_n, test_m, test_array, answer)

        test_n = 2
        test_m = 5
        test_array = [50, 40, 30, 20, 10]
        answer = [(0, 0), (1, 0), (1, 40), (0, 100), (1, 100), (2, 100)]
        pattern_test_for_heap(test_n, test_m, test_array, answer)

    main()

    test_for_heap()

    print("All tests were successfully passed")
```