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. 1304 Шаврин А.П.

Преподаватель Глазунов С.А.

Санкт-Петербург

2022

Цель работы.

Изучить работу очереди с приоритетом и параллельной обработки данных. Реализовать очередь с приоритетом при помощи мин-кучи.

Задание.

Параллельная обработка**.**

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

Первой строкой добавьте #python или #c++, чтобы проверяющая система знала, каким языком вы пользуетесь.

Выполнение работы.

1. Изначально было решено реализовать мин-кучу для решения задачи параллельной обработки данных.

Был создан класс *MinHeap*. Для него был написан конструктор, принимающий на вход список процессоров и устанавливающий поля класса в соответствующие ему значения. Одно из полей класса – список *result*, реализовано для проверок с помощью системы *pytest*.

В классе были написаны два статических метода *getLeftChildIndex* и *getRightChildIndex*, получающие на вход индекс родителя и возвращающие индекс соответствующего названию ребенка.

Реализован метод *getResult*, не принимающий аргументов и возвращающий значение поля класса *result*. Данный метод используется только для проверок с помощью системы *pytest*.

После был создан метод *siftDown*, принимающий на вход индекс корня и просеивающий его вниз по куче, что бы обеспечивалась целостность мин-кучи (значение в любом родителе не больше, чем значение его детей).

В конце был написан метод *parallelProcessing*, принимающий на вход список задач и осуществляющий их параллельную обработку. Для каждой задачи на экран выводится номер процессора, который ее обрабатывает и время, через которое он приступит к ее обработке. Затем время работы текущего процессора увеличивается на величину, равную времени процессора. После вызывается метод *siftDown(0)*, для обеспечения целостности мин-кучи с уже изменившимися значениями узлов.

2. Затем была реализована основная логика работы программы.

Сначала было реализовано считывание параметров *n* и *m*, которые затем передаются в функцию *main*

В функции *main* создаются:

- список процессоров, элементы которого – пара значений (первое - время, в которое процессор перейдет к следующей задаче, второе – номер процессора)
- список задач
- объекта класса *MinHeap* с передачей в конструктор вычисленного списка процессоров.

Затем вызывается метод *parallelProcessing*, через объект класса *MinHeap*, в который передается вычисленный ранее список задач.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

Была изучена работа очереди с приоритетом и параллельная обработка данных. Была написана программа, которая выполняет параллельную обработку данных при помощи мин-кучи.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: MinHeap.py

```
#python

class MinHeap:
    def __init__(self, processors):
        self.data = processors
        self.size = len(processors)
        self.result = []

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

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

    def getResult(self):
        return self.result

    def siftDown(self, index):
        min_index = index
        left_child_index = self.getLeftChildIndex(index)
        right_child_index = self.getRightChildIndex(index)

        if left_child_index < self.size and self.data[left_child_index] <
self.data[min_index]:
            min_index = left_child_index

        if right_child_index < self.size and self.data[right_child_index]
< self.data[min_index]:
            min_index = right_child_index

        if index != min_index:
            self.data[index], self.data[min_index] = self.data[min_index],
self.data[index]
            self.siftDown(min_index)

    def parallelProcessing(self, tasks):
        if (self.data != []):
            for cur_task in tasks:
                cur_time = self.data[0][0]
                processor = self.data[0][1]
                print(processor, cur_time)
                self.result.append([processor, cur_time])
                self.data[0][0] += cur_task
                self.siftDown(0)


def main(n, m):
    processors = [[0, i] for i in range(n)]
```

```
    if m <= 0:
        tasks = []
    else:
        tasks = list(map(int, input().split(' ')))

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)


if __name__ == "__main__":
    n, m = map(int, input().split())
    main(n, m)
```

Название файла: pytests.py

```
from MinHeap import MinHeap


def test_mv():
    processors = [[0, 0], [0, 1]]
    tasks = [1, 2, 3, 4]
    correct_answer = [[0, 0], [1, 0], [0, 1], [1, 2]]

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer


def test_one_task():
    processors = [[0, 0], [0, 1]]
    tasks = [1]
    correct_answer = [[0, 0]]

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer


def test_zero_tasks():
    processors = [[0, 0], [0, 1]]
    tasks = []
    correct_answer = []

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer

def test_negative_tasks():
    processors = [[0, 0], [0, 1]]
    tasks = []
    correct_answer = []

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)
```

```
    assert min_heap.getResult() == correct_answer

def test_same_tasks():
    processors = [[0, 0], [0, 1]]
    tasks = [1, 1, 1, 1, 1]
    correct_answer = [[0, 0], [1, 0], [0, 1], [1, 1], [0, 2]]

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer


def test_many_processors():
    processors = [[0, 0], [0, 1], [0, 2], [0, 3]]
    tasks = [1, 2, 3, 4, 5, 6, 7]
    correct_answer = [[0, 0], [1, 0], [2, 0], [3, 0], [0, 1], [1, 2], [2,
3]]

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer

def test_zero_processors():
    processors = []
    tasks = [1, 2, 3]
    correct_answer = []

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer

def test_negative_number_of_processors():
    processors = []
    tasks = [1, 2, 3]
    correct_answer = []

    min_heap = MinHeap(processors)
    min_heap.parallelProcessing(tasks)

    assert min_heap.getResult() == correct_answer
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные
1.	2 4 1 2 3 4	0 0 1 0 0 1 1 2
2.	2 1 1	0 0
3.	2 0	
4.	3 -3	
5.	2 5 1 1 1 1 1	0 0 1 0 0 1 1 1 0 2
6.	4 7 1 2 3 4 5 6 7	0 0 1 0 2 0 3 0 0 1 1 2 2 3
7.	0 2 3 4	
8.	-1 2 1 2	