**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.**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 1304 |  | Чернякова В.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2022

## Цель работы.

Освоить работу с очередью, а именно очередь с приоритетом. Разработать класс минимальной двоичной кучи для реализации очереди с приоритетом.

## Задание.

**Параллельная обработка.**

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.

**Ограничения**

1 ≤ n ≤ 10 5 ; 1 ≤ m ≤ 105 ; 0 ≤ t i ≤ 109 .

**Пример:**

**Вход:**

2 5

1 2 3 4 5

**Выход:**

0 0

1 0

0 1

1 2

0 4

## Выполнение работы.

На вход программе с помощью функции *input()* подается строка, содержащая натуральное число *n* – количество процессоров и *m* – количество задач, которые надо обработать. С помощью функции *int()* в введённой строке преобразуются соответствующие строковые числа в обычные. Далее с помощью функции *input().split()* в список *arr* считываются значения времени, необходимые для обработки *i*-той задачи. Список *time* формируется преобразованием значений, хранящихся в *arr*, с помощью функции *int().* Создается *heap* – объект класса *Heap(n).* Для решения поставленной задачи лабораторной работы вызывается метод *parallel\_process(time).* Результат работы метода выводится на экран.

**Класс Heap и его методы.**

Конструктор класса *def \_\_init\_\_(self, n)* – создание экземпляра класса. Поле *MAX\_SIZE* отвечает за максимальный размер списка и равен *n*, так как количество процессоров вводится с клавиатуры и больше него их не будет. Поле *heap* создает список из кортежей, то есть элементы кучи представляются в виде списка: по умолчанию на первой позиции кортежа *0*, на второй *i for i in range(n)*, то есть номер процессора. Поле *size* – размер списка.

Далее объявляются статические методы поиска родителя, левого и правого ребенка. *@staticmethod* используется, так как *self* не требуются, и они могут быть вызваны, когда объект не создан.

Индекс родителя: *def get\_parent(index): return (index - 1) // 2.*

Индекс левого ребенка: *def get\_left\_child(index): return 2 \* index + 1.*

Индекс правого ребенка: *def get\_right\_child(index): return 2 \* index + 2.*

Выполнение поставленной задачи лабораторной работы происходит следующим образом: печатаем вершину кучи, добавляем время задачи ко времени вершины и просеиваем данную вершину вниз.

Метод *sift\_down(self, index)* – просеивание вниз. В переменные *left* и *right* записываются индексы левого и правого ребенка соответственно с помощью методов *get\_left\_child(index)* и *get\_right\_child(index).* Осуществляем проверку, имеются ли дети вообще: *left >= self.size and right >= self.size*. Действия, если нет правого ребенка, а только левый: для просеивания вниз минимального элемента, сравниваем значение левого ребенка и значение родителя соответственно *min\_index = left if self.heap[left] < self.heap[index] else index*. Если есть оба ребенка, то в начале проверяем, какое ребенок меньше *min\_index = left if self.heap[left] < self.heap[right] else right*. Затем сравниваем выбранного ребенка с родителем, меньше ли он: *min\_index = min\_index if self.heap[min\_index] < self.heap[index] else index*. Если *min\_index != index*, то нужно поменять их местами: s*elf.heap[min\_index], self.heap[index] = self.heap[index], self.heap[min\_index]*. Так как свойство минимальной кучи могло нарушиться, то для *min\_index* осуществляем также просеивание вниз *self.sift\_down(min\_index).*

Метод *def parallel\_process(self, times)* – осуществляющий параллельную обработку. Метод принимает в качестве аргумента список, содержащий время выполнения соответствующей *i*-той задачи. Создается с помощью *list()* список *answer*, куда будут помещаться ответы. С помощью цикла *for* проходим по *elem* из *times.* В ответ добавляются поля класса *self.heap[0][1]* и *self.heap[0][0],* где хранятся номер процессора, который сейчас будет обрабатывать задачу, и время, когда задача пришла этому процессору. Далее к *heap[0][0]* добавляется время обрабатываемой задачи, так как следующая придет только после того, как пройдет столько времени. И этот процессор просеивается вниз *self.sift\_down(0).* Метод возвращает *answer*.

Разработанный программный код см. в приложении А.

## Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 2 5  1 2 3 4 5 | 0 0  1 0  0 1  1 2  0 4 | Проверка работы алгоритма для стандартного случая. |
|  | 4 7  10 10 10 10 10 10 10 | 0 0  1 0  2 0  3 0  0 10  1 10  2 10 | Проверка работы алгоритма для задач с одинаковым временем. |
|  | 1 3  12 3 6 | 0 0  0 12  0 15 | Проверка работы алгоритма для одного процессора. |
|  | 4 5  100 9 1 23 45 | 0 0  1 0  2 0  3 0  2 1 | Проверка работы алгоритма для случая, когда процессоров на один меньше => поиск первого освободившегося процессора. |
|  | 3 2  23 3 | 0 0  1 0 | Проверка работы алгоритма при избытке процессоров. |

## Выводы.

Был изучен принцип работы с очередью с приоритетом. На основе полученных знаний создана программа, в которой реализован класс минимальной кучи, позволяющий работать с очередью с приоритетом. Написано тестирование для программного кода, проверяющее его корректность.

# Приложение А Исходный код программы

Название файла: main.py

class Heap:

def \_\_init\_\_(self, n):

self.MAX\_SIZE = n

self.heap = [[0, i] for i in range(n)]

self.size = n

@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.MAX\_SIZE:

return -1

self.heap[self.size] = element

self.sift\_up(self.size)

self.size += 1

def extract\_max(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] < self.heap[index]:

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:

min\_index = left if self.heap[left] < self.heap[index] else index

else:

min\_index = left if self.heap[left] < self.heap[right] else right

min\_index = min\_index if self.heap[min\_index] < self.heap[index] else index

if min\_index != index:

self.heap[min\_index], self.heap[index] = self.heap[index], self.heap[min\_index]

self.sift\_down(min\_index)

def parallel\_process(self, times):

answer = list()

for elem in times:

answer.append([self.heap[0][1], self.heap[0][0]])

self.heap[0][0] += elem

self.sift\_down(0)

return answer

if \_\_name\_\_ == '\_\_main\_\_':

n, m = map(int, input().split())

arr = input().split()

time = [int(i) for i in arr]

heap = Heap(n)

result = heap.parallel\_process(time)

for answ in result:

print(\*answ)

Название файла: test.py

from main import Heap

import pytest

def test1():

heap = Heap(2)

time = [1, 2, 3, 4, 5]

assert heap.parallel\_process(time) == [[0, 0], [1, 0], [0, 1], [1, 2], [0, 4]]

def test2():

heap = Heap(4)

time = [10, 10, 10, 10, 10, 10, 10]

assert heap.parallel\_process(time) == [[0, 0], [1, 0], [2, 0], [3, 0], [0, 10], [1, 10], [2, 10]]

def test3():

heap = Heap(1)

time = [12, 3, 6]

assert heap.parallel\_process(time) == [[0, 0], [0, 12], [0, 15]]

def test4():

heap = Heap(4)

time = [100, 9, 1, 23, 45]

assert heap.parallel\_process(time) == [[0, 0], [1, 0], [2, 0], [3, 0], [2, 1]]

def test5():

heap = Heap(3)

time = [23, 3]

assert heap.parallel\_process(time) == [[0, 0], [1, 0]]