**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 |  | Клепнёв Д.А. |
| Преподаватель |  | Глазунов С.А. |

Санкт-Петербург

2021

**Цель работы.**

Изучить такую структуру данных, как очередь с приоритетом. А также рассмотреть ее реализацию на примере двоичной мин-кучи/макс-кучи.

**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.**

Был реализован класс *Heap*, который представляет двоичную мин-кучу. Содержит метод *performing*(), который решает задачу. После пользовательского ввода создается список, в котором каждый элемент соответственно список с двумя значениями, первое – номер процессора, второе – время, за которое процессор перейдет к очередной задаче. Созданный список передается в конструктор экземпляра класса *Heap*. Метод *sift\_down()* выполняет просеивание вниз для соблюдения условия мин-кучи. Как происходит просеивание: ищет ‘ребенка’ с наименьшим временем процессора у вершины ‘родителя’, если время обоих ‘детей’ равны, берет процессор с наименьшим номером и сравнивает его со временем процессора вершины ‘родителя’. Метод *performing*() прибавляет к текущему времени процессора время на обработку поступившей задачи, а затем вызывает *sift\_down(0),* будет получаться, что в корне находится процессор с наименьшим временем, то есть тот процессор, который освободился от задачи быстрее всех.

**Тестирование.**

|  |  |  |
| --- | --- | --- |
| № теста | Входные данные | Выходные данные |
| 1 | 2 1  1 | 0 0 |
| 2 | 1 4  1 2 3 4 | 0 0  0 1  0 3  0 6 |
| 3 | 2 0 | - |
| 4 | 3 7  1 2 3 4 5 1 2 | 0 0  1 0  2 0  0 1  1 2  2 3  2 4 |
| 5 | 2 5  2 1 1 3 1 | 0 0  1 0  1 1  0 2  1 2 |

**Выводы.**

Изучил способы реализации очереди с приоритетом через реализацию двоичной мин-кучи.

приложение а

исходный код программы

class Heap:

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

self.heap = heap

self.size = len(heap)

self.result = []

def get\_left\_child(self, index):

return index \* 2 + 1

def get\_right\_child(self, index):

return index \* 2 + 2

def sift\_down(self, index):

current = index

left = self.get\_left\_child(index)

if left < self.size:

if self.heap[left][1] < self.heap[current][1]:

current = left

elif self.heap[left][1] == self.heap[current][1] and self.heap[left][0] < self.heap[current][0]:

current = left

right = self.get\_right\_child(index)

if right < self.size:

if self.heap[right][1] < self.heap[current][1]:

current = right

elif self.heap[right][1] == self.heap[current][1] and self.heap[right][0] < self.heap[current][0]:

current = right

if current != index:

self.heap[current], self.heap[index] = self.heap[index], self.heap[current]

self.sift\_down(current)

def performing(self, ilist):

for i in ilist:

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

self.heap[0][1] += i

self.sift\_down(0)

if \_\_name\_\_ == "\_\_main\_\_":

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

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

heap.performing(list(map(int, input().split())))

print(heap.result)

for current in heap.result:

print(current[0], current[1])