**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. 1384 |  | Усачева Д.В. |
| Преподаватель |  | Иванов Д.В. |
|  |  |  |

Санкт-Петербург 2022

**Цель работы.**

Знакомство с такой структурой данных, как очередь с приоритетом.

Применение очереди с приоритетом в задачах.

**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 .

**Выполнение работы.**

Для начала программа считывает количество процессоров, участвующих в обработке, и количество задач, которые необходимо обработать в переменные n и m соответственно. Список значений времени, необходимых для обработки каждой задачи, сохраняется в список work.

Был описан класс Processor, хранящий индекс процессора, время, необходимое для выполнения задачи, а также начало выполнения данной задачи. У класса были перегружены операторы сравнения, таким образом можно сравнить два объекта данного класса, что потребуется при выполнении операций просеивания в куче.

В классе Heap реализована очередь с приоритетом (мин-куча), хранящая процессор с наименьшим в хронологическом смысле временем выполнения задачи «вверху» кучи, и методы для работы с такой кучей.

Тестирование.

Чтобы удостовериться в правильности работы программы, она была протестирована на следующих случаях:

1. Задачи распределены таким образом, что один процессор выполняет маленькие задачи, а все остальные с самого начала нагружаются задачами с большим временем выполнения
2. Количество задач совпадает с количеством процессоров – т. е. каждому процессору достаётся по одной задаче
3. На несколько задач выделяется только один процессор
4. Подаются задачи с одинаковым временем выполнения и работают два процессора

**Выводы.**

По результатам лабораторной работы была изучена структура данных очередь с приоритетом. В разработанной программе была осуществлена работа с кучей, хранящей в себе объекты некоторого класса, путем

переопределения оператора сравнения

**ПРИЛОЖЕНИЕ 1**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: main.py

#python

class Heap:

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

self.size = size

self.heap = heap

@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 add\_wk(self, element):

proc = self.heap[0]

result = (proc.idx, proc.work)

proc.add(element)

self.sift\_down(0)

return result

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)

class Processor:

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

self.work = 0

self.idx = idx

def add(self, work):

self.work += work

def \_\_lt\_\_(self, other): # <

if self.work == other.work:

return self.idx < other.idx

else:

return self.work < other.work

def \_\_le\_\_(self, other): # ≤

if self.work == other.work:

return self.idx <= other.idx

else:

return self.work <= other.work

def \_\_eq\_\_(self, other): # ==

return self.work == other.work

def \_\_ne\_\_(self, other): # !=

return self.work != other.work

def \_\_gt\_\_(self, other): # >

if self.work == other.work:

return self.idx > other.idx

else:

return self.work > other.work

def \_\_ge\_\_(self, other): # ≥

if self.work == other.work:

return self.idx >= other.idx

else:

return self.work >= other.work

def solve(n, works):

ans=[]

heap\_list = [Processor(i) for i in range(n)]

heap = Heap(n, heap\_list)

for work in works:

result = heap.add\_wk(work)

ans.append(result)

return ans

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

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

works = list(map(int, input().split()))

for pair in solve(n, works):

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

Название файла: tests.py

from main import \*

def test\_all\_tasks\_to\_one\_processor():

N = 10

t = [1]

for i in range(N - 1):

t.append(1000)

for i in range(100):

t.append(1)

expectedAnswer = []

for i in range(N):

expectedAnswer.append((i, 0))

for i in range(100):

expectedAnswer.append((0, i + 1))

assert solve(N, t) == expectedAnswer

def test\_one\_task\_lots\_of\_processors():

N = 100

t = [34]

assert solve(N, t) == [(0, 0)]

def test\_one\_processor():

t = [i for i in range(15)]

answers = [(0, sum(t[:i])) for i in range(15)]

assert solve(1, t) == answers

def test\_two\_processors\_sequence():

t = [1] \* 10

N = 2

exAns = []

for i in range(len(t)):

exAns.append((i % 2, i // 2))

assert solve(N, t) == exAns