УНИВЕРСИТЕТ ИТМО

Факультет инфокоммуникационных технологий
Интеллектуальные системы в гуманитарной среде
Алгоритмы и структуры данных

Лабораторная работа №5 «Деревья. Пирамида, пирамидальная сортировка. Очередь с приоритетами.»

Студентка: Демша Евгения Сергеевна, К3143
Преподаватель: Харьковская Татьяна Александровна

Санкт-Петербург
10/09/2021

1 задача. Куча ли?

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива. Для этого должно выполнятся основное свойство неубывающей пирамиды, которое заключается в том, что для каждого 1 ≤ i ≤ n выполняются условия:
1. если 2i ≤ n, то ai ≤ a2i
2. если 2i + 1 ≤ n, то ai ≤ a2i+1.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

• Формат входного файла (input.txt). Первая строка входного файла содержит целое число n (1 ≤ n ≤ 10^6). Вторая строка содержит n целых чисел, по модулю не превосходящих 2 · 10^9

• Формат выходного файла (output.txt). Выведите «YES», если массив является неубывающей пирамидой, и «NO» в противном случае.

• Ограничение по времени. 2 сек.

• Ограничение по памяти. 256 мб.

Описание кода:

С помощью цикла проверяем, что каждый «ребенок» больше своего «родителя».

Исходный код:

```
with open("Input", "r") as fi:
    lines = fi.read().split("\n")
fo = open("Output", "w")
n = int(lines[0])
array = [int(i) for i in lines[1].split()]
print(array)

def check_heap(array, n):
    for i in range(n):
        if 2 * i + 1 < n:
            if array[2 * i + 1] < array[i] or array[2 * i + 2] < array[i]:
                return "No"
    return "Yes"

fo.write(str(check_heap(array, n)))
```

4 задача. Построение пирамиды

В этой задаче вы преобразуете массив целых чисел в пирамиду. Это важнейший шаг алгоритма сортировки под названием HeapSort. Гарантированное время работы в худшем случае составляет O(n log n), в отличие от среднего времени работы QuickSort, равного O(n log n). QuickSort обычно используется на практике, потому что обычно он быстрее, но HeapSort используется для внешней сортировки, когда вам нужно отсортировать огромные файлы, которые не помещаются в памяти вашего компьютера.

Первым шагом алгоритма HeapSort является создание пирамиды (heap) из массива, который вы хотите отсортировать. Ваша задача - реализовать этот первый шаг и преобразовать заданный массив целых чисел в пирамиду. Вы сделаете это, применив к массиву определенное количество перестановок (swaps). Перестановка - это операция, как вы помните, при которой элементы ai и aj массива меняются местами для некоторых i и j.

Вам нужно будет преобразовать массив в пирамиду, используя только O(n) перестановок. Обратите внимание, что в этой задаче вам нужно будет использовать min-heap вместо max-heap.

• Формат ввода или входного файла (input.txt). Первая строка содержит целое число n ($1 \le n \le 10^5$), вторая содержит n целых чисел ai входного массива, разделенных пробелом ($0 \le ai \le 10^9$, все ai - различны.)

• Формат выходного файла (output.txt). Первая строка ответа должна содержать целое число m - количество сделанных свопов. Число m должно удовлетворять условию $0 \le m \le 4n$. Следующие m строк должны содержать по 2 числа: индексы i и j сделанной перестановки двух элементов, индексы считаются с 0. После всех перестановок в нужном порядке массив должен стать пирамидой, то есть для каждого i при $0 \le i \le n-1$ должны выполняться условия:

1. если $2i + 1 \le n - 1$, то $ai < a2i+1$,

2. если $2i + 2 \le n - 1$, то $ai < a2i+2$.

Обратите внимание, что все элементы входного массива различны. Любая последовательность свопов, которая менее 4n и после которой входной массив становится корректной пирамидой, считается верной.

• Ограничение по времени. 3 сек.

• Ограничение по памяти. 512 мб.

Описание кода:

Двигаемся перебираем элементы списка с середины(в середине находится последний «родитель», который может быть больше своих «детей»), для каждого элемента восстанавливаем «неубываемость» кучи.

Исходный код:

```
with open("Input", "r") as fi:
    lines = fi.read().split("\n")
fo = open("Output", "w")
n = int(lines[0])
massive = [int(i) for i in lines[1].split()]
heap = []


def min_heapify(A, i):
    heap_size = len(A)
    _l = 2 * i + 1
    r = 2 * i + 2
    smallest = i
    if _l < heap_size and A[_l] < A[smallest]:
        smallest = _l
    if r < heap_size and A[r] < A[smallest]:
        smallest = r

    if smallest != i:
        A[i], A[smallest] = A[smallest], A[i]
        heap.append((i, smallest))
        min_heapify(A, smallest)


def build_min_heap(A):
    heap_size = len(A)
    for i in reversed(range(heap_size // 2)):
```

```
        min_heapify(A, i)


build_min_heap(massive)

fo.write(str(len(heap)))
fo.write("\n")
for h in heap:
    fo.write(str(h[0]))
    fo.write(" ")
    fo.write(str(h[1]))
    fo.write("\n")
```

5 задача. Планировщик заданий

В этой задаче вы создадите программу, которая параллельно обрабатывает список заданий. Во всех операционных системах, таких как Linux, MacOS или Windows, есть специальные программы, называемые планировщиками, которые делают именно это с программами на вашем компьютере.

У вас есть программа, которая распараллеливается и использует n независимых потоков для обработки заданного списка m заданий. Потоки берут задания в том порядке, в котором они указаны во входных данных. Если есть свободный поток, он немедленно берет следующее задание из списка. Если поток начал обработку задания, он не прерывается и не останавливается, пока не завершит обработку задания. Если несколько потоков одновременно пытаются взять задания из списка, поток с меньшим индексом берет задание. Для каждого задания вы точно знаете, сколько времени потребуется любому потоку, чтобы обработать это задание, и это время одинаково для всех потоков.

Вам необходимо определить для каждого задания, какой поток будет его обрабатывать и когда он начнет обработку.

• Формат ввода или входного файла (input.txt). Первая строка содержит целые числа n и m ($1 \le n \le 10^5$, $1 \le m \le 10^5$). Вторая строка содержит m целых чисел ti - время в секундах, которое требуется для выполнения i-ой задания любым потоком ($0 \le ti \le 10^9$). Все эти значения даны в том порядке, в котором они подаются на выполнение. Индексы потоков начинаются с 0.

• Формат выходного файла (output.txt). Выведите в точности m строк, причем i-ая строка (начиная с 0) должна содержать два целочисленных значения: индекс потока, который выполняет i-ое задание, и время в секундах, когда этот поток начал выполнять задание.

• Ограничение по времени. 6 сек.

• Ограничение по памяти. 512 мб.

Описание кода:

В кучу добавляются потоки, у которых есть два приоритета, главный – время завершения работы (сначала это 0), а второй номер потока, который важен, когда первые приоритеты разных потоков равны. Куча неубывает. Цикл проходится по списку задач, для каждой выбирает первый поток из кучи, увеличивает его время конца работы и кладет обратно в кучу, находя для него подходящее место. При каждом проходе цикла информация о номере потока и о начале выполнения задания (равно времени конца выполнения предыдущего) записывается в выходной файл.

Исходный код:

```
with open("Input", "r") as fi:
    lines = fi.read().split("\n")
fo = open("Output", "w")
n, m = [int(i) for i in lines[0].split()]
tasks = [int(i) for i in lines[1].split()]


class PriorityQueue:
    def __init__(self):
        self.heap_size = 0

    def min_heapify(self, A, i):
        _l = 2 * i + 1
        r = 2 * i + 2
        largest = i
        if _l < self.heap_size and A[_l][0] < A[largest][0]:
            largest = _l
        if _l < self.heap_size and A[_l][0] == A[largest][0] and A[_l][1] <
A[largest][1]:
            largest = _l
        if r < self.heap_size and A[r][0] < A[largest][0]:
            largest = r
        if r < self.heap_size and A[r][0] == A[largest][0] and A[r][1] <
A[largest][1]:
            largest = r

        if largest != i:
            A[i], A[largest] = A[largest], A[i]
            self.min_heapify(A, largest)

    def heap_extract_min(self, A):
        if self.heap_size < 1:
            return print("Ошибка: очередь пуста")
        max = A[0]
        A[0] = A[-1]
        A.pop(-1)
        self.heap_size -= 1
        self.min_heapify(A, 0)
        print(A)
        return max

    def find_parent(self, i):
        parent = i // 2
        if i % 2 == 0:
            parent -= 1
        return parent

    def heap_increase_key(self, A, i, key):
        if key[0] < A[i][0]:
            return print("Ошибка: новый ключ меньше текущего")
        A[i] = key
        parent = self.find_parent(i)
        while (i > 0 and A[parent][0] > A[i][0]) or (i > 0 and A[parent][0]
== A[i][0] and A[parent][1] > A[i][1]):
            A[i], A[parent] = A[parent], A[i]
            i = parent
            parent = self.find_parent(i)
        print(A)

    def max_heap_insert(self, A, key):
        A.append((float("-inf"),float("-inf")))
        self.heap_increase_key(A, self.heap_size, key)
        self.heap_size += 1
```

```
heap = []
a = PriorityQueue()
for flow in range(n):
    a.max heap insert(heap, (0, flow))


for t in tasks:
    fo.write(str(heap[0][1]))
    fo.write(" ")
    fo.write(str(heap[0][0]))
    fo.write("\n")
    temp = heap[0]
    a.heap_extract_min(heap)
    a.max_heap_insert(heap, (t + temp[0], temp[1]))
```

6 задача. Очередь с приоритетами

Реализуйте очередь с приоритетами. Ваша очередь должна поддерживать следующие операции: добавить элемент, извлечь минимальный элемент, уменьшить элемент, добавленный во время одной из операций.

• Формат входного файла (input.txt). В первой строке входного файла содержится число n (1 ≤ n ≤ 10^6) - число операций с очередью. Следующие n строк содержат описание операций с очередью, по одному описанию в строке. Операции могут быть следующими:

– A x – требуется добавить элемент x в очередь.

– X – требуется удалить из очереди минимальный элемент и вывести его в выходной файл. Если очередь пуста, в выходной файл требуется вывести звездочку «*».

– D x y – требуется заменить значение элемента, добавленного в очередь операцией A в строке входного файла номер x + 1, на y. Гарантируется, что в строке x + 1 действительно находится операция A, что этот элемент не был ранее удален операцией X, и что y меньше, чем предыдущее значение этого элемента. В очередь помещаются и извлекаются только целые числа, не превышающие по модулю 10^9

• Формат выходного файла (output.txt). Выведите последовательно результат выполнения всех операций X, по одному в каждой строке выходного файла. Если перед очередной операцией X очередь пуста, выведите вместо числа звездочку «*».

• Ограничение по времени. 2 сек.

• Ограничение по памяти. 256 мб.

Описание кода:

Реализован класс очередь с приоритетом.

Исходный код:

```
with open("Input", "r") as fi:
    lines = fi.read().split("\n")
fo = open("Output", "w")
n = int(lines[0])
lines.pop(0)
```

```
class PriorityQueue:
    def __init__(self):
        self.heap_size = 0

    def heap_minimum(self, A):
        return A[0]

    def find_parent(self, i):
        parent = i // 2
        if i % 2 == 0:
            parent -= 1
        return parent

    def heap_decrease_key(self, A, i, key):
        if key > A[i]:
            return print("Ошибка: новый ключ больше текущего")
        A[i] = key
        parent = self.find_parent(i)
        while i > 0 and A[parent] > A[i]:
            A[i], A[parent] = A[parent], A[i]
            i = parent
            parent = self.find_parent(i)

    def min_heap_insert(self, A, key):
        A.append(float("inf"))
        self.heap_decrease_key(A, self.heap_size, key)
        self.heap_size += 1

    def min_heapify(self, A, i):
        _l = 2 * i + 1
        r = 2 * i + 2
        smallest = i
        if _l < self.heap_size and A[_l] < A[smallest]:
            smallest = _l
        if r < self.heap_size and A[r] < A[smallest]:
            smallest = r

        if smallest != i:
            A[i], A[smallest] = A[smallest], A[i]
            self.min_heapify(A, smallest)

    def heap_extract_min(self, A):
        if self.heap_size < 1:
            return print("Ошибка: очередь пуста")
        max = A[0]
        A[0] = A[-1]
        A.pop(-1)
        self.heap_size -= 1
        self.min_heapify(A, 0)
        return max

l = []
a = PriorityQueue()

for action in lines:
    print(l)
    print(action)

    if action[0] == 'X':
        if l:
            fo.write(str(a.heap_extract_min(l)))
            fo.write("\n")
        else:
```

```
            fo.write('*')

    if action[0] == 'A':
        a.min_heap_insert(l, int(action[2]))

    if action[0] == 'D':
        x = int(action[2]) - 1
        if lines[x-1][0] == "A":
            if int(lines[x][2]) in l:
                i = l.index(int(lines[x][2]))
                a.heap_decrease_key(l, i, int(action[4]))
```

Вывод:

Я привыкла к пирамидам, научилась видеть их в массиве, восстанавливать их невозрастание или неубывание. Двоичная куча очень удобна для нахождения элемента с максимальным или минимальным приоритетом.