Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Факультет Інформатики та Обчислювальної Техніки

Кафедра Автоматизованих Систем Обробки Інформації та Управління

Лабораторна робота № 6

з дисципліни «Теорія алгоритмів»

на тему:

**"** **Піраміди"**

Виконав:

студент гр. ІС-02

Плостак Ілля

Викладач:

Cт. Вик. Новікова П. А.

Київ – 2021

**1. Завдання**

Нехай заданий вхідний масив A = [x1, ..., xN]. Припустимо, що елементи масиву поступають на вхід програми послідовно: в кожний момент часу розглядається новий елемент xi. Необхідно для кожного i (від 1 до N) визначити медіану підмасиву A' = [x1, ..., xi], тобто медіану для масиву елементів, які були отримані програмою на даний момент часу. Необхідно розв’язати цю задачу, використовуючі структури даних пірамід і так, щоб кожна медіана визначалась за час O(log(i)).

**2. Програмний код:**

[main.py](https://github.com/feedblackg44/kpilabs2/blob/master/TA/Lab6/main.py):

from algorithms import GetMedian, AddElement  
from Heap import Heap  
  
lowHeap = Heap([])  
highHeap = Heap([])  
strOut = ""  
for i, line in enumerate(open("input.txt", "r")):  
 if i > 0:  
 AddElement(lowHeap, highHeap, int(line))  
 med = GetMedian(lowHeap, highHeap)  
 for i in range(len(med)):  
 strOut += str(med[i])  
 if i != len(med) - 1:  
 strOut += " "  
 strOut += "\n"  
  
out = open("output.txt", "w")  
out.write(strOut)  
out.close()

[algorithms.py](https://github.com/feedblackg44/kpilabs2/blob/master/TA/Lab6/algorithms.py):

from Heap import Heap  
  
def GetMedian(lowHeap, highHeap):  
 if len(lowHeap) > len(highHeap):  
 return [lowHeap.HeapMaximum()]  
 elif len(lowHeap) < len(highHeap):  
 return [highHeap.HeapMaximum()]  
 else:  
 return [lowHeap.HeapMaximum(), highHeap.HeapMaximum()]  
  
def AddElement(lowHeap, highHeap, element):  
 if len(lowHeap) == 0:  
 lowHeap.add(element)  
 elif len(highHeap) == 0:  
 if element < lowHeap.HeapMaximum():  
 highHeap.add(lowHeap.HeapExtractMax())  
 lowHeap.add(element)  
 else:  
 highHeap.add(element)  
 elif element < lowHeap.HeapMaximum():  
 lowHeap.ins(0, element)  
 lowHeap.MaxHeapify(0)  
 else:  
 highHeap.ins(0, element)  
 highHeap.MinHeapify(0)  
  
 if len(lowHeap) - len(highHeap) == 2:  
 highHeap.ins(0, lowHeap.HeapExtractMax())  
 elif len(highHeap) - len(lowHeap) == 2:  
 lowHeap.ins(0, highHeap.HeapExtractMax())

[Heap.py](https://github.com/feedblackg44/kpilabs2/blob/master/TA/Lab6/Heap.py):

class Heap:  
 arr = []  
 heap\_size = 0  
  
 def \_\_init\_\_(self, arr):  
 self.arr = arr  
 self.heap\_size = len(arr)  
  
 def MaxHeapify(self, i):  
 p = Heap.Left(i)  
 q = Heap.Right(i)  
 largest = i  
 if p < self.heap\_size and self[p] > self[largest]:  
 largest = p  
 if q < self.heap\_size and self[q] > self[largest]:  
 largest = q  
 if largest != i:  
 self.arr[i], self.arr[largest] = self.arr[largest], self.arr[i]  
 self.MaxHeapify(largest)  
  
 def MinHeapify(self, i):  
 p = Heap.Left(i)  
 q = Heap.Right(i)  
 lowest = i  
 if p < self.heap\_size and self[p] < self[lowest]:  
 lowest = p  
 if q < self.heap\_size and self[q] < self[lowest]:  
 lowest = q  
 if lowest != i:  
 self.arr[i], self.arr[lowest] = self.arr[lowest], self.arr[i]  
 self.MinHeapify(lowest)  
  
 def HeapMaximum(self):  
 return self.arr[0]  
 def HeapExtractMax(self):  
 if self.heap\_size < 1:  
 raise Exception("Heap is empty!")  
 self.heap\_size -= 1  
 return self.arr.pop(0)  
 @staticmethod  
 def Left(i):  
 return i << 1  
 @staticmethod  
 def Right(i):  
 return (i << 1) + 1  
 @staticmethod  
 def Parent(i):  
 return i >> 1  
 def \_\_getitem\_\_(self, key):  
 return self.arr[key]  
 def \_\_len\_\_(self):  
 return len(self.arr)  
 def \_\_str\_\_(self):  
 return str(self.arr)  
 def ins(self, key, obj):  
 self.arr.insert(key, obj)  
 self.heap\_size += 1  
 def add(self, obj):  
 self.arr.append(obj)  
 self.heap\_size += 1

**3. Приклад вхідного файлу, результат запуску програми та приклад вихідного файлу:**

Результат запуску програми для перевірки чи сортують розроблені алгоритми:



**4. Висновок:**

В цій лабораторній роботі ми вивчили алгоритм стійкого сортування (на основі Counting Sort) та Radix Sort. Стійкий алгоритм має складність Θ(n+k), де k це довжина тимчасового масиву (додаткова пам’ять), а загалом увесь Radix Sort має складність Θ(d(n+k)), де d це кількість розрядів чисел у масиві. Ця комбінація алгоритмів сильно скорочує час роботи порівняно зі звичайним Counting Sort. У Counting Sort розмір додаткової пам’яті завжди залежить від висхідного масиву та може бути дуже великим, а у цій комбінації k <=9 завжди. На жаль на відміну від алгоритмів які базуються на порівняннях цей алгоритм не працює для чисел менших за 0 тому що сортує за розрядами не враховуючи наявність мінусу.

Вивід у програмі здійснюється у консоль.