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](http://wiki.kpi.ua/index.php/%D0%A2%D0%B5%D0%BE%D1%80%D1%96%D1%8F%20%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%96%D0%B2_(20202020))»

на тему

«Піраміди»

Виконав:

студент гр. ІС-91

Хмелiнiн Андрiй

Викладач:

[ст.вик. Халус О. А.](http://rozklad.kpi.ua/Schedules/ViewSchedule.aspx?v=318f7459-ddf0-47ee-b882-5650e756cf2a)

Київ – 2020

class MinHeap:

#Клас котрий реалізую власний тип данних Min-Heap

\_list = [] # Масив, котрий зберігає елементи піраміди

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

#Конструктор класу

pass

@property

def heap\_size(self):

#Властивість(getter), що повертає розмір піраміди

return len(self.\_list)

def add(self, item):

#Метод, що дозволяє додати новий елемент до піраміди та поставивши його у потрібне місце після цього

self.\_list.append(item) # додаємо елемент в кінець масиву

i = self.heap\_size - 1 # позначаємо його дочірнім

parent = int((i - 1) / 2) # знаходимо батьківський елемент для нього

# будемо обмінювати в циклі елементи,тим самим підіймати його по піраміді вгору, доки не знайдеться елемент, що буде меншим за елемент, що ми додали

while i > 0 and self.\_list[parent] > self.\_list[i]:

self.\_list[i], self.\_list[parent] = self.\_list[parent], self.\_list[i]

i = parent

parent = int((i - 1) / 2)

def heapify(self, i):

#Метод, що перевіряє піраміду на властивість неспадної піраміди та повертає цю властивість якщо вона була втрачена

# будемо опускатися вниз по піраміді, щоразу визначаючи найменший елемент та ставлячи його в правильну позицію доки властивість неспадної піраміди не буде виконана

while True:

left\_child = 2 \* i + 1 # індекс лівого дочірнього елемента

right\_child = 2 \* i + 2 # індекс правого дочірнього елемента

smallest\_child = i # індекс найменшого елемента

if left\_child < self.heap\_size and self.\_list[left\_child] < self.\_list[smallest\_child]:

smallest\_child = left\_child

if right\_child < self.heap\_size and self.\_list[right\_child] < self.\_list[smallest\_child]:

smallest\_child = right\_child

if smallest\_child == i:

break

self.\_list[i], self.\_list[smallest\_child] = self.\_list[smallest\_child], self.\_list[i]

i = smallest\_child

def get\_min(self, status\_delete=False):

#Метод, що повертає мінімальний елемент піраміди, при цьому його можна видалити з піраміди, передавши як аргумент status\_delete=True

if not status\_delete:

return self.\_list[0]

# якщо елемент необхідно видалити з піраміди:

# 1. обмінюємо місцями перший та останній елемент

# 2. видаляємо останній(мінімальний) елемент записавши результат в змінну

# 3. викликаємо метод heapify, що був описаний раніше в цьому класі

self.\_list[0], self.\_list[self.heap\_size-1] =\

self.\_list[self.heap\_size-1], self.\_list[0]

min\_item = self.\_list.pop(self.heap\_size-1)

self.heapify(0)

return min\_item

class MaxHeap:

#Клас котрий реалізує власний тип данних Max-Heap

\_list = [] # Масив, котрий зберігає елементи піраміди

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

#Конструктор класу

pass

@property

def heap\_size(self):

#Властивість(getter), що повертає розмір піраміди

return len(self.\_list)

def add(self, item):

#Метод, що дозволяє додати новий елемент до піраміди та поставивши його у потрібне місце після цього

self.\_list.append(item) # додаємо елемент в кінець масиву

i = self.heap\_size - 1 # позначаємо його дочірнім

parent = int((i - 1) / 2) # знаходимо батьківський елемент для нього

# будемо обмінювати в циклі елементи,тим самим підіймати його по піраміді вгору, доки не знайдеться елемент, що буде меншим за елемент, що ми додали

while i > 0 and self.\_list[parent] < self.\_list[i]:

self.\_list[i], self.\_list[parent] = self.\_list[parent], self.\_list[i]

i = parent

parent = int((i - 1) / 2)

def heapify(self, i):

#Метод, що перевіряє піраміду на властивість незростаючої піраміди та повертає цю властивість якщо вона була втрачена

# будемо опускатися вниз по піраміді, щоразу визначаючи найменший елемент та ставлячи його в правильну позицію доки властивість неспадної піраміди не буде виконана

while True:

left\_child = 2 \* i + 1 # індекс лівого дочірнього елемента

right\_child = 2 \* i + 2 # індекс правого дочірнього елемента

largest\_child = i # індекс найбільшого елемента

if left\_child < self.heap\_size and self.\_list[left\_child] > self.\_list[largest\_child]:

largest\_child = left\_child

if right\_child < self.heap\_size and self.\_list[right\_child] > self.\_list[largest\_child]:

largest\_child = right\_child

if largest\_child == i:

break

self.\_list[i], self.\_list[largest\_child] = self.\_list[largest\_child], self.\_list[i]

i = largest\_child

def get\_max(self, status\_delete=False):

#Метод, що повертає максимальний елемент піраміди, при цьому його можна видалити з піраміди, передавши як аргумент status\_delete=True

if not status\_delete:

return self.\_list[0]

# якщо елемент необхідно видалити з піраміди:

# 1) обмінюємо місцями перший та останній елемент

# 2) видаляємо останній(максимальний) елемент записавши результат в

# змінну

# 3) викликаємо метод heapify, що був описаний раніше в цьому класі

self.\_list[0], self.\_list[self.heap\_size - 1] = self.\_list[self.heap\_size - 1], self.\_list[0]

max\_item = self.\_list.pop(self.heap\_size - 1)

self.heapify(0)

return max\_item

def check\_count(low\_heap: MaxHeap, high\_heap: MinHeap):

#Функція, що перевіряє умову різниці елементів між пірамідами(різниця в к-сті елементів має бути не більше одного)

# якщо low\_heap більше на два елементи, то видаляємо з неї найбільший та вставляємо до high\_heap

if (low\_heap.heap\_size - high\_heap.heap\_size) == 2:

high\_heap.add(low\_heap.get\_max(status\_delete=True))

# якщо high\_heap більше на два елементи, то видаляємо з неї найменший та вставляємо до low\_heap

elif (high\_heap.heap\_size - low\_heap.heap\_size) == 2:

low\_heap.add(high\_heap.get\_min(status\_delete=True))

def get\_median(low\_heap: MaxHeap, high\_heap: MinHeap, count):

#Функція, що повертає одну чи дві медіани в залежності від к-сті переданих елементів"""

# якщо до програми було подано парну кількість елементів повертаємо дві медіани - найбільший з low\_heap та найменший з high\_heap

if count % 2 == 0:

return f"{low\_heap.get\_max()} {high\_heap.get\_min()}"

# інакше медіана одна і зберігається в піраміді в якій на один елемент більше

else:

if low\_heap.heap\_size > high\_heap.heap\_size:

return low\_heap.get\_max()

else:

return high\_heap.get\_min()

def program(file\_name,file\_name1):

#Основна функція, де відбувається зчитування з файлу, виконання поставлених задач та запис результату до файлу

# зчитуємо дані з файлу до масиву

with open(file\_name, 'r') as file:

data = (file.read()).split('\n')

data = [int(elem) for elem in data]

n, data = data[0], data[1:]

# ініціалізуємо low\_heap та high\_heap

low\_heap = MaxHeap()

high\_heap = MinHeap()

# первий елемент додаємо до low\_heap

low\_heap.add(data[0])

# зчитуємо дані з файлу до масиву

with open(file\_name1, 'w') as file:

# перша медіана буде перший елемент, що був доданий до low\_heap

file.write(str(low\_heap.get\_max()) + "\n")

# поступово опрацьовуємо всі вхідні елементи

for i in range(1, n):

num = data[i] # беремо новий елемент з масиву

# якщо елемент менший за максимальний елемент low\_heap, а отже у відсортованому масиві знаходився б у 1 половині то додаємо його до low\_heap

if num < low\_heap.get\_max():

low\_heap.add(num)

# інакше до high\_heap

else:

high\_heap.add(num)

# перевіряємо на різницю елементів між пірамідами

check\_count(low\_heap, high\_heap)

# записуємо отриману медіану до файлу

file.write(str(get\_median(low\_heap, high\_heap, i + 1)) + "\n")

program('Inputs/input\_01\_10.txt','Outputs/khmelinin\_output\_01\_10.txt')

program('Inputs/input\_02\_10.txt','Outputs/khmelinin\_output\_02\_10.txt')

program('Inputs/input\_03\_10.txt','Outputs/khmelinin\_output\_03\_10.txt')

program('Inputs/input\_04\_10.txt','Outputs/khmelinin\_output\_04\_10.txt')

program('Inputs/input\_05\_10.txt','Outputs/khmelinin\_output\_05\_10.txt')

program('Inputs/input\_06\_100.txt','Outputs/khmelinin\_output\_06\_100.txt')

program('Inputs/input\_07\_100.txt','Outputs/khmelinin\_output\_07\_100.txt')

program('Inputs/input\_08\_100.txt','Outputs/khmelinin\_output\_08\_100.txt')

program('Inputs/input\_09\_100.txt','Outputs/khmelinin\_output\_09\_100.txt')

program('Inputs/input\_10\_100.txt','Outputs/khmelinin\_output\_10\_100.txt')

program('Inputs/input\_11\_1000.txt','Outputs/khmelinin\_output\_11\_1000.txt')

program('Inputs/input\_12\_1000.txt','Outputs/khmelinin\_output\_12\_1000.txt')

program('Inputs/input\_13\_1000.txt','Outputs/khmelinin\_output\_13\_1000.txt')

program('Inputs/input\_14\_1000.txt','Outputs/khmelinin\_output\_14\_1000.txt')

program('Inputs/input\_15\_1000.txt','Outputs/khmelinin\_output\_15\_1000.txt')

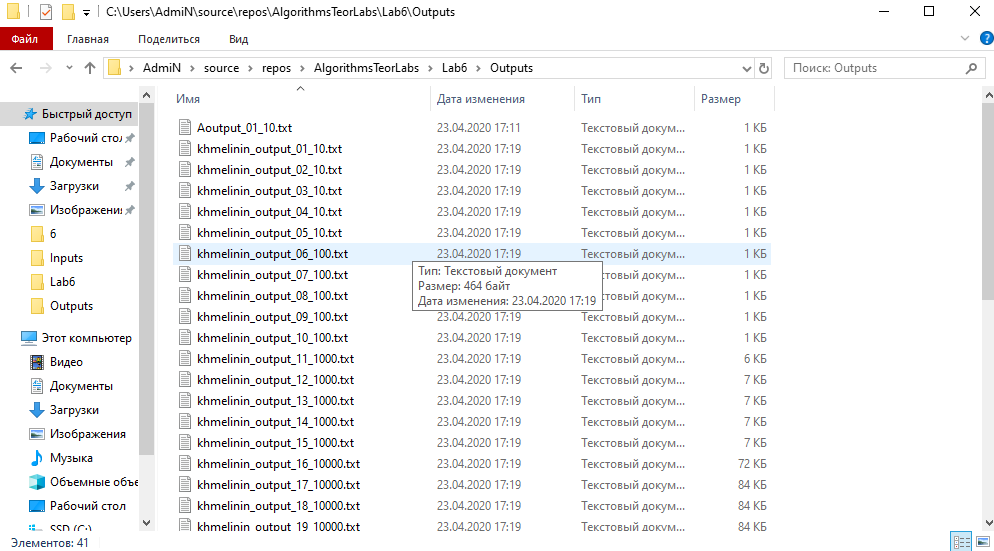
program('Inputs/input\_16\_10000.txt','Outputs/khmelinin\_output\_16\_10000.txt')

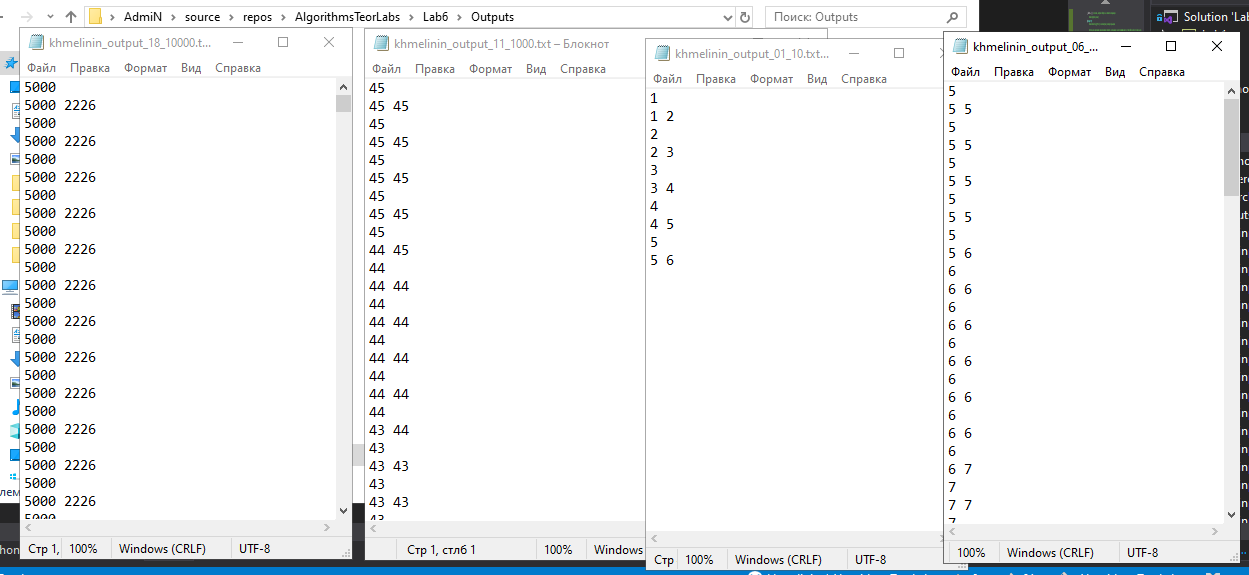
program('Inputs/input\_17\_10000.txt','Outputs/khmelinin\_output\_17\_10000.txt')

program('Inputs/input\_18\_10000.txt','Outputs/khmelinin\_output\_18\_10000.txt')

program('Inputs/input\_19\_10000.txt','Outputs/khmelinin\_output\_19\_10000.txt')

program('Inputs/input\_20\_10000.txt','Outputs/khmelinin\_output\_20\_10000.txt')





Висновок:

Я реалізував алгоритм HeapSort та виконав поставленне завдання. Розроблена програма зчитує вхідні дані з файлу заданого формату та записує дані у файл заданого формату.