МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет електроніки та комп’ютерних технологій

**Звіт**

Про виконання лабораторної роботи №2

з дисципліни «Алгоритми та структури даних»

На тему «Структура даних Піраміди»

**Виконав**:

Малярчук О.В. ФЕП-24

**Перевірив**:

**Львів 2022**

**Мета роботи**: ознайомитись з структурою даних піраміди, зрозуміти її реалізацію, та навчитись реалізовувати їх в довільній мові програмування.

**Програмне забезпечення**: PyCharm.

**Теоретичні відомості:**

**• Операція ExtractMax(S) повертає елемент з найбільшим ключем, при цьому видаляючи його з множини S.**

**• Операція IncreaseKey(S, x, k) збільшує значення ключа, відповідного елементу x, шляхом заміни його значенням k. Передбачається, що величина k не менше поточного ключа елементу x.**

**Одна з галузей застосування незростаючих черг – планування завдань на комп’ютері, який разом використовується декількома користувачами. Черга дозволяє слідкувати за завданнями, які потребують виконання, та за їх пріоритетами. Якщо завдання перервано або завершило свою роботу, з черги за допомогою операції ExtractMax() обирається наступне завдання з найбільшим пріоритетом. В чергу в будь-який час можна додати нове завдання, використовуючи операцію Insert(). У неспадній черзі з пріоритетами підтримуються операції Insert(), Minimum(), ExtractMin() та DecreaseKey(). Черги такого типу можуть використовуватись в моделюванні систем, які керують подіями. В ролі елементів черги в такому випадку виступають змодельовані події, для кожної з яких зіставляється час початку, що відповідає ключу елемента. Елементи повинні моделюватись послідовно згідно часу подій, оскільки процес моделювання може виявити генерацію інших подій. Програма з моделювання обирає наступну подію за допомогою операції ExtractMin(). Коли ініціюються нові події, вони розміщуються в чергу за допомогою процедури Insert().**

**Пріоритетна черга можна реалізувати за допомогою піраміди. В кожному окремому випадку елементи черги з пріоритетами відповідають об’єктам, з якими працює відповідне програмний застосунок. Часто виникає необхідність виявити, який з об’єктів застосунку відповідає тому або іншому елементу черги, чи навпаки. Якщо черга з пріоритетами реалізується за допомогою піраміди, то в кожному елементі піраміди доводиться зберігати ідентифікатор відповідного об’єкту застосунку.**

**Зміст роботи**

* Створив файл Heap.py, де буду реалізувати весь функціонал, який вимагається в лабораторній роботі.
* Реалізував клас Heap, в якому буде виконаний наступний функціонал:
* Parent()
* Left()
* Right()
* BuildMaxHeap()
* BuildMinHeap()
* MaxHeapify()
* MinHeapify()
* MaxHeapInsert()
* MinHeapInsert()
* HeapIncreaseKey()
* HeapDecreaseKey()
* ExtractMax()
* ExtractMin()
* HeapMin()
* HeapMax()
* HeapSort()
* ShowHeap()

1. В файлі main.py прописав імпорт класу Heap з файлу Heap.py.

**Висновок:** на даній лабораторній роботі я ознайомився з найпоширенішими операціями для роботи з підамідами та пріоритетними чергами, розглянув алгоритми для створення бінарного дерева пошуку. Також, навчився реалізовувати функції для підамід та пріоритетних черг.

Додаток 1.

INDENTATION = 5

class Heap:

def \_\_init\_\_(self, InputList: list = None):

self.heap\_arr = InputList

self.heap\_len = len(InputList)

def BuildMaxHeap(self):

"""Creates Heap from given array"""

for i in range(self.heap\_len // 2, -1, -1):

self.SiftDownMax(index=i)

def BuildMinHeap(self):

"""Creates Heap from given array"""

for i in range(self.heap\_len // 2, -1, -1):

self.SiftDownMin(index=i)

def MaxHeapify(self, index):

"""Sift down element in MaxHeap"""

return self.SiftDownMax(index)

def MinHeapify(self, index):

"""Sift Down element in MinHeap"""

return self.SiftDownMin(index)

def MaxHeapInsert(self, data):

"""Insert element in MaxHeap"""

self.heap\_arr.append(data)

index = self.heap\_len

self.heap\_len += 1

self.SiftUpMax(index)

def MinHeapInsert(self, data):

"""Insert element in MinHeap"""

self.heap\_arr.append(data)

index = self.heap\_len

self.heap\_len += 1

self.SiftUpMin(index)

def \_Extract(self):

"""Swap First and Last element

:return Last Element"""

output = self.heap\_arr[0]

index = self.heap\_len - 1

self.heap\_arr[0], self.heap\_arr[index] = self.heap\_arr[index], self.heap\_arr[0]

self.heap\_arr.pop()

self.heap\_len -= 1

return output

def HeapIncreaseKey(self, index: int = 0, data: int = 0):

"""Change data in MaxHeap"""

try:

self.heap\_arr[index]

except IndexError:

return IndexError

if self.heap\_arr[index] < data:

self.heap\_arr[index] = data

self.SiftUpMax(index)

elif self.heap\_arr[index] > data:

return ValueError

def HeapDecreaseKey(self, index: int = 0, data: int = 0):

"""Change data in MinHeap"""

try:

self.heap\_arr[index]

except IndexError:

return IndexError

if self.heap\_arr[index] < data:

return ValueError

elif self.heap\_arr[index] > data:

self.heap\_arr[index] = data

self.SiftUpMin(index)

def ExtractMax(self):

"""Remove Max element from MaxHeap"""

output = self.\_Extract()

self.SiftDownMax(0)

return output

def ExtractMin(self):

"""Remove Min element from MinHeap"""

output = self.\_Extract()

self.SiftDownMin(0)

return output

def HeapMin(self):

"""Returns Min element from MinHeap"""

return self.heap\_arr[0]

def HeapMax(self):

"""Returns Max element from MaxHeap"""

return self.heap\_arr[0]

def \_HeapSortMax(self):

output = list()

for i in range(self.heap\_len - 1, -1, -1):

output.append(self.ExtractMax())

return output

def HeapSort(self, way\_of\_sorting: str = '>'):

"""Sort array using heap sort"""

output = self.\_HeapSortMax()

if way\_of\_sorting == '>':

return output

elif way\_of\_sorting == '<':

return output[::-1]

else:

return None

def SiftUpMax(self, index):

"""Sift up element for MaxHeap"""

parent = self.parent(index)

while self.HasParent(index) and self.heap\_arr[parent] < self.heap\_arr[index]:

self.heap\_arr[parent], self.heap\_arr[index] = self.heap\_arr[index], self.heap\_arr[parent]

index = parent

parent = self.parent(index)

return index

def SiftDownMax(self, index):

"""Sift down element for MaxHeap"""

max\_index = index

left\_child = self.left(index)

if self.HasLeftChild(index) and self.heap\_arr[left\_child] > self.heap\_arr[max\_index]:

max\_index = left\_child

right\_child = self.right(index)

if self.HasRightChild(index) and self.heap\_arr[right\_child] > self.heap\_arr[max\_index]:

max\_index = right\_child

if index != max\_index:

self.heap\_arr[index], self.heap\_arr[max\_index] = self.heap\_arr[max\_index], self.heap\_arr[index]

self.SiftDownMax(max\_index)

def SiftUpMin(self, index):

"""Sift up element for MinHeap"""

parent = self.parent(index)

while self.HasParent(index) and self.heap\_arr[parent] > self.heap\_arr[index]:

self.heap\_arr[parent], self.heap\_arr[index] = self.heap\_arr[index], self.heap\_arr[parent]

index = parent

parent = self.parent(index)

return index

def SiftDownMin(self, index):

"""Sift down element for MinHeap"""

min\_index = index

left\_child = self.left(index)

if self.HasLeftChild(index) and self.heap\_arr[left\_child] < self.heap\_arr[min\_index]:

min\_index = left\_child

right\_child = self.right(index)

if self.HasRightChild(index) and self.heap\_arr[right\_child] < self.heap\_arr[min\_index]:

min\_index = right\_child

if index != min\_index:

self.heap\_arr[index], self.heap\_arr[min\_index] = self.heap\_arr[min\_index], self.heap\_arr[index]

self.SiftDownMin(min\_index)

def ShowHeap(self):

print('-' \* 44)

self.\_ShowHeap()

print('-' \* 44)

def \_ShowHeap(self, index=0, d=0):

if index >= self.heap\_len:

return

l = self.left(index)

r = self.right(index)

self.\_ShowHeap(index=r, d=d + 1)

print(' ' \* INDENTATION \* d + str(self.heap\_arr[index]))

self.\_ShowHeap(index=l, d=d + 1)

@staticmethod

def parent(index: int):

return (index - 1) >> 1

@staticmethod

def left(index: int):

return (index << 1) + 1

@staticmethod

def right(index: int):

return (index << 1) + 2

def HasParent(self, index: int):

return self.heap\_len > self.parent(index) >= 0

def HasLeftChild(self, index: int):

return self.right(index) < self.heap\_len

def HasRightChild(self, index: int):

return self.right(index) < self.heap\_len