list = []

from priority\_queue\_base import PriorityQueueBase

############################################# QUESTION 1 #############################################

# Classes from in Class examples

class HeapPriorityQueue(PriorityQueueBase): # base class defines \_Item

"""A min-oriented priority queue implemented with a binary heap."""

# ------------------------------ nonpublic behaviors ------------------------------

def \_parent(self, j):

return (j - 1) // 2

def \_left(self, j):

return 2 \* j + 1

def \_right(self, j):

return 2 \* j + 2

def \_has\_left(self, j):

return self.\_left(j) < len(self.\_data) # index beyond end of list?

def \_has\_right(self, j):

return self.\_right(j) < len(self.\_data) # index beyond end of list?

def \_swap(self, i, j):

"""Swap the elements at indices i and j of array."""

self.\_data[i], self.\_data[j] = self.\_data[j], self.\_data[i]

# Change sign to greater than

def \_upheap(self, j):

parent = self.\_parent(j)

if j > 0 and self.\_data[j] > self.\_data[parent]:

self.\_swap(j, parent)

self.\_upheap(parent) # recur at position of parent

# Change sign to greater than

def \_downheap(self, j):

if self.\_has\_left(j):

left = self.\_left(j)

small\_child = left # although right may be smaller

if self.\_has\_right(j):

right = self.\_right(j)

if self.\_data[right] > self.\_data[left]:

small\_child = right

if self.\_data[small\_child] > self.\_data[j]:

self.\_swap(j, small\_child)

self.\_downheap(small\_child) # recur at position of small child

# ------------------------------ public behaviors ------------------------------

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

"""Create a new empty Priority Queue."""

self.\_data = []

def \_\_len\_\_(self):

"""Return the number of items in the priority queue."""

return len(self.\_data)

def add(self, key, value):

"""Add a key-value pair to the priority queue."""

self.\_data.append(self.\_Item(key, value))

self.\_upheap(len(self.\_data) - 1) # upheap newly added position

def min(self):

"""Return but do not remove (k,v) tuple with minimum key.

Raise Empty exception if empty.

"""

if self.is\_empty():

raise Exception('Priority queue is empty.')

item = self.\_data[0]

return (item.\_key, item.\_value)

#removes the max

def remove\_max(self):

if self.is\_empty():

raise Exception('Priority queue is empty.')

self.\_swap(0, len(self.\_data) - 1) # put minimum item at the end

item = self.\_data.pop() # and remove it from the list;

self.\_downheap(0) # then fix new root

return (item.\_key, item.\_value)

def \_\_iter\_\_(self):

"""Generate iteration of the map's keys."""

for item in self.\_data:

yield item # yield the KEY

#in place sort, derived from in class example

def InPlaceHeapSort(self, sorted, a, b):

if a >= b:

return

pivot = sorted[b]

left = a

right = b - 1

while left <= right:

while left <= right and sorted[left] < pivot:

left += 1

while left <= right and pivot < sorted[right]:

right -= 1

if left <= right:

sorted[left], sorted[right] = sorted[right], sorted[left]

left, right = left + 1, right - 1

sorted[left], sorted[b] = sorted[b], sorted[left]

# recursive calls

self.InPlaceHeapSort(sorted, a, left - 1)

self.InPlaceHeapSort(sorted, left + 1, b)

############################################# QUESTION 2 #############################################

# Classes from in Class examples, tweaked for question 2

class BinHeap(): # base class defines \_Item

"""A min-oriented priority queue implemented with a binary heap."""

# ------------------------------ nonpublic behaviors ------------------------------

def \_parent(self, j):

return (j - 1) // 2

def \_left(self, j):

return 2 \* j + 1

def \_right(self, j):

return 2 \* j + 2

def \_has\_left(self, j):

return self.\_left(j) < len(self.\_data) # index beyond end of list?

def \_has\_right(self, j):

return self.\_right(j) < len(self.\_data) # index beyond end of list?

def \_swap(self, i, j):

"""Swap the elements at indices i and j of array."""

self.\_data[i], self.\_data[j] = self.\_data[j], self.\_data[i]

# Change sign to greater than

def upheap(self, j):

parent = self.\_parent(j)

if j > 0 and self.\_data[j] < self.\_data[parent]:

self.\_swap(j, parent)

self.upheap(parent) # recur at position of parent

# Change sign to greater than

def downheap(self, j):

if self.\_has\_left(j):

left = self.\_left(j)

small\_child = left # although right may be smaller

if self.\_has\_right(j):

right = self.\_right(j)

if self.\_data[right] < self.\_data[left]:

small\_child = right

if self.\_data[small\_child] < self.\_data[j]:

self.\_swap(j, small\_child)

self.downheap(small\_child) # recur at position of small child

# ------------------------------ public behaviors ------------------------------

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

"""Create a new empty Priority Queue."""

self.\_data = []

def size(self):

"""Return the number of items in the priority queue."""

return len(self.\_data)

def insert(self, value):

"""Add a key-value pair to the priority queue."""

self.\_data.append(value)

self.upheap(len(self.\_data) - 1) # upheap newly added position

def find\_min(self):

"""Return but do not remove (k,v) tuple with minimum key.

Raise Empty exception if empty.

"""

if self.is\_empty():

raise Exception('Priority queue is empty.')

item = self.\_data[0]

return (item)

def del\_min(self):

"""Remove and return (k,v) tuple with minimum key.

Raise Empty exception if empty.

"""

if self.\_data.\_\_len\_\_()==0:

raise Exception('Priority queue is empty.')

self.\_swap(0, len(self.\_data) - 1) # put minimum item at the end

item = self.\_data.pop() # and remove it from the list;

self.downheap(0) # then fix new root

return (item)

def \_\_iter\_\_(self):

"""Generate iteration of the keys."""

for item in self.\_data:

yield item # yield the KEY

#is empty function

def is\_empty(self):

return(self.\_data.\_\_len\_\_()==0)

#added function for buiding a heap from a list

def build\_heap(self, L):

new = BinHeap()

for i in range(len(L)):

new.insert(L[i])

print("New Heap From A New List Is")

i=new.\_data.\_\_len\_\_()

for j in range(i):

print(new.del\_min(), end=" ")

print()

############################################# QUESTION 1 #############################################

print("Question 1:")

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

H = HeapPriorityQueue() # Create HeapPriorityQueue[]

H.add(9, '9')

H.add(7, '7')

H.add(5, '5')

H.add(2, '2')

H.add(6, '6')

H.add(4, '4')

for i in range(len(H)):

item = H.\_data[i]

list.append(item.\_key)

print("Keys Before In Place Sort:")

print(list)

print("Keys After In Place Sort:")

H.InPlaceHeapSort(list, 0, list.\_\_len\_\_() - 1)

print(list)

print()

print('\*\* Print Using Remove\_Max \*\*')

for i in range(len(H)):

print(H.remove\_max())

############################################# QUESTION 2 #############################################

print()

print()

print("Question 2:")

#class object

Bh = BinHeap()

#insert items

Bh.insert(7)

Bh.insert(4)

Bh.insert(9)

Bh.insert(3)

Bh.insert(8)

Bh.insert(13)

i = Bh.size()

print("First Created Heap Is:")

print("The size of below array is: ", Bh.size())

print("Is the array empty?:", Bh.is\_empty())

for j in range(i):

print(Bh.del\_min(), end=" ")

print()

print()

list2 = [9,7,5,2,6,4]

#build heap from list 2

Bh.build\_heap(list2)