```

# HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
# Path    : uebung04/ml/aufgabe02
# Version: Tue Oct 10 09:56:57 CEST 2023

import sys
import random
from queue import PriorityQueue as PythonPQ
from uebung04.ml.aufgabe02.priority_queue_adv import PriorityQueueADV
from uebung04.ml.aufgabe02.priority_queue import PriorityQueue


def stress_test():
  print("\nStress-Test: ... ", end = "")
  NUMBER_OF_TESTS = 20000
  LENGTH_RANGE = 10
  DATA_RANGE = 10
  i = 0
  while i < NUMBER_OF_TESTS:
    length = random.choice(range(1, LENGTH_RANGE))
    randoms = []
    j = 0
    while j < length:
      randoms.append(int(random.uniform(0, DATA_RANGE)))
      j += 1
    ourPQ = PriorityQueue(length)
    pythonPQ = PythonPQ(length)
    for r in randoms:
      ourPQ.insert(r, "Value_" + str(r))
      pythonPQ.put(r)
    j = 0
    while j < length:
      if ourPQ.size() != pythonPQ.qsize():
        print("ERROR: wrong size!")
        print("randoms: " + str(randoms))
        sys.exit(1)
      if ourPQ.remove_min().get_key() != pythonPQ.get():
        print("ERROR: wrong removeMin()!")
        print("randoms: " + str(randoms))
        sys.exit(2)
      j += 1
    if ourPQ.remove_min() != None:
      print("ERROR: removeMin() != None")
      print("randoms: " + str(randoms))
      sys.exit(3)
    i += 1
  print("o.k.")



```

```

if __name__ == ’__main__’:

  pq = PriorityQueue(7)
  # pq = PriorityQueueADV(7, "Uebung 4:PQ", 2, 2)

  print("insert()’s: ")
  pq.print()
  pq.insert(4, "D")
  pq.print()
  pq.insert(5, "E")
  pq.print()
  pq.insert(3, "C")
  pq.print()
  pq.insert(2, "B")
  pq.print()
  pq.insert(1, "A")
  pq.print()
  print("\nmin(): " + str(pq.min()))
  while pq.size() > 1:
    print("remove_min(): " + str(pq.remove_min()))
    pq.print()

  stress_test()


""" Session-Log::

insert()’s:
[None, None, None, None, None, None, None, None]
[None, [(4,D),1], None, None, None, None, None, None]
[None, [(4,D),1], [(5,E),2], None, None, None, None, None]
[None, [(3,C),1], [(5,E),2], [(4,D),3], None, None, None, None]
[None, [(2,B),1], [(3,C),2], [(4,D),3], [(5,E),4], None, None, None]
[None, [(1,A),1], [(2,B),2], [(4,D),3], [(5,E),4], [(3,C),5], None, None]

min(): (1,A)
remove_min(): (1,A)
[None, [(2,B),1], [(3,C),2], [(4,D),3], [(5,E),4], None, None, None]
remove_min(): (2,B)
[None, [(3,C),1], [(5,E),2], [(4,D),3], None, None, None, None]
remove_min(): (3,C)
[None, [(4,D),1], [(5,E),2], None, None, None, None, None]
remove_min(): (4,D)
[None, [(5,E),1], None, None, None, None, None, None]

Stress-Test: ... o.k.

"""
```

```

# HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
# Path    : uebung04/ml/aufgabe02
# Version: Tue Oct 10 09:56:57 CEST 2023

import functools
from uebung04.ml.aufgabe02.full_priority_queue_exception import FullPriorityQueueException


class PriorityQueue:
  """ A heap-based (array-implementation) Priority-Queue with fixed length. """

  @functools.total_ordering
  class _PQEntry:

    def __init__(self, key, value):
      self._key = key
      self._value = value

    def get_key(self):
      return self._key

    def get_value(self):
      return self._value

    def __lt__(self, other):
      if other == None:
        return False
      return self._key < other._key

    def __eq__(self, other):
      if other == None:
        return False
      return self._key == other._key

    def __str__(self):
      return "(" + str(self._key) + "," + str(self._value) +")"


  def __init__(self, max_size):
    self._heap_array = [None] * (max_size+1)
    self._last = 0 # Points to the last element in the heap.

  def insert(self, key, value):
    if self._last == (len(self._heap_array) - 1):
      raise FullPriorityQueueException("Maximum reached: " + str(len(self._heap_array)))
    self._last += 1
    e = PriorityQueue._PQEntry(key, value)
    self._heap_array[self._last] = e
    self._upheap(self._last)
    return e

  def min(self):
    return self._heap_array[1]

  def remove_min(self):
    if self._last == 0:
      return None
    result = self._heap_array[1]
    self._heap_array[1] = self._heap_array[self._last]
    self._heap_array[self._last] = None
    self._last -= 1
    self._downheap(1)
    return result

  def is_empty(self):
    return self.size() == 0


```

```

  def size(self):
    return self._last

  def print(self):
    print(self.__str__())

  def __str__(self):
    string = "["
    for i in range(len(self._heap_array)):
      entry= self._heap_array[i]
      if entry != None:
        string += "[" + str(entry) + "," + str(i) + "]"
      else:
        string += "None"
      if i < len(self._heap_array)-1:
        string += ", "
    string += "]"
    return string

  def _swap(self, parent_index, child_index):
    """ Swaps a child-node with its parent-node.

    parentIndex Index of the parent-node.

    childIndex Index of the child-node.
    """
    tmp = self._heap_array[parent_index]
    self._heap_array[parent_index] = self._heap_array[child_index]
    self._heap_array[child_index] = tmp

  def _upheap(self, current_index):
    if current_index == 1:
      return  # no further _upheap-swaps possible
    parent = current_index // 2
    if self._heap_array[parent] > self._heap_array[current_index]:
      self._swap(parent, current_index)
      self._upheap(parent)

  def _downheap(self, current_index):
    left_child_index = current_index * 2
    right_child_index = left_child_index + 1
    left_child_is_smaller  = self._check_for_potential_swap(current_index, left_child_index)
    right_child_is_smaller = self._check_for_potential_swap(current_index, right_child_index)
    if left_child_is_smaller and right_child_is_smaller:
      if self._heap_array[left_child_index] <= self._heap_array[right_child_index]:
        self._swap_and_downheap(current_index, left_child_index)
      else:
        self._swap_and_downheap(current_index, right_child_index)
    elif left_child_is_smaller:
      self._swap_and_downheap(current_index, left_child_index)
    elif right_child_is_smaller:
      self._swap_and_downheap(current_index, right_child_index)

  def _check_for_potential_swap(self, parent, child):
    return (child <= self._last) and (self._heap_array[parent] > self._heap_array[child])

  def _swap_and_downheap(self, parent, child):
    self._swap(parent, child)
    self._downheap(child)

```

10.10.2023 9:56:57 **full_priority_queue_exception.py** Page 1/1

```

# HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
# Path   : uebung04/ml/aufgabe02
# Version: Tue Oct 10 09:56:57 CEST 2023

class FullPriorityQueueException(Exception):

  def __init__(self, err):
    super().__init__(err)


```