from doubly\_linked\_base import \_DoublyLinkedBase  
#these classes are from lecuture notes  
  
class PositionalList(\_DoublyLinkedBase):  
 *"""A sequential container of elements allowing positional access."""* # -------------------------- nested Position class --------------------------  
 class Position:  
 *"""An abstraction representing the location of a single element."""* def \_\_init\_\_(self, container, node):  
 *"""Constructor should not be invoked by user."""* self.\_container = container  
 self.\_node = node  
  
 def element(self):  
 *"""Return the element stored at this Position."""* return self.\_node.\_element  
  
 def \_\_eq\_\_(self, other):  
 *"""Return True if other is a Position representing the same location."""* return type(other) is type(self) and other.\_node is self.\_node  
  
 def \_\_ne\_\_(self, other):  
 *"""Return True if other does not represent the same location."""* return not (self == other) # opposite of \_\_eq\_\_  
  
 # ------------------------------- utility methods -------------------------------  
 def \_validate(self, p):  
 *"""Return position's node, or raise appropriate error if invalid."""* if not isinstance(p, self.Position):  
 raise TypeError('p must be proper Position type')  
 if p.\_container is not self:  
 raise ValueError('p does not belong to this container')  
 if p.\_node.\_next is None: # convention for deprecated nodes  
 raise ValueError('p is no longer valid')  
 return p.\_node  
  
 def \_make\_position(self, node):  
 *"""Return Position instance for given node (or None if sentinel)."""* if node is self.\_header or node is self.\_trailer:  
 return None # boundary violation  
 else:  
 return self.Position(self, node) # legitimate position  
  
 # ------------------------------- accessors -------------------------------  
 def first(self):  
 *"""Return the first Position in the list (or None if list is empty)."""* return self.\_make\_position(self.\_header.\_next)  
  
 def last(self):  
 *"""Return the last Position in the list (or None if list is empty)."""* return self.\_make\_position(self.\_trailer.\_prev)  
  
 def before(self, p):  
 *"""Return the Position just before Position p (or None if p is first)."""* node = self.\_validate(p)  
 return self.\_make\_position(node.\_prev)  
  
 def after(self, p):  
 *"""Return the Position just after Position p (or None if p is last)."""* node = self.\_validate(p)  
 return self.\_make\_position(node.\_next)  
  
 def \_\_iter\_\_(self):  
 *"""Generate a forward iteration of the elements of the list."""* cursor = self.first()  
 while cursor is not None:  
 yield cursor.element()  
 cursor = self.after(cursor)  
  
 # ------------------------------- mutators -------------------------------  
 # override inherited version to return Position, rather than Node  
 def \_insert\_between(self, e, predecessor, successor):  
 *"""Add element between existing nodes and return new Position."""* node = super().\_insert\_between(e, predecessor, successor)  
 return self.\_make\_position(node)  
  
 def add\_first(self, e):  
 *"""Insert element e at the front of the list and return new Position."""* return self.\_insert\_between(e, self.\_header, self.\_header.\_next)  
  
 def add\_last(self, e):  
 *"""Insert element e at the back of the list and return new Position."""* return self.\_insert\_between(e, self.\_trailer.\_prev, self.\_trailer)  
  
 def add\_before(self, p, e):  
 *"""Insert element e into list before Position p and return new Position."""* original = self.\_validate(p)  
 return self.\_insert\_between(e, original.\_prev, original)  
  
 def add\_after(self, p, e):  
 *"""Insert element e into list after Position p and return new Position."""* original = self.\_validate(p)  
 return self.\_insert\_between(e, original, original.\_next)  
  
 def delete(self, p):  
 *"""Remove and return the element at Position p."""* original = self.\_validate(p)  
 return self.\_delete\_node(original) # inherited method returns element  
  
 def replace(self, p, e):  
 *"""Replace the element at Position p with e.  
 Return the element formerly at Position p.  
 """* original = self.\_validate(p)  
 old\_value = original.\_element # temporarily store old element  
 original.\_element = e # replace with new element  
 return old\_value # return the old element value  
  
  
class PriorityQueueBase:  
 *"""Abstract base class for a priority queue."""* # ------------------------------ nested \_Item class ------------------------------  
 class \_Item:  
 *"""Lightweight composite to store priority queue items."""* \_\_slots\_\_ = '\_key', '\_value'  
  
 def \_\_init\_\_(self, k, v):  
 self.\_key = k  
 self.\_value = v  
  
 def \_\_lt\_\_(self, other):  
 return self.\_key < other.\_key # compare items based on their keys  
  
 def \_\_repr\_\_(self):  
 return '({0},{1})'.format(self.\_key, self.\_value)  
  
 # ------------------------------ public behaviors ------------------------------  
 def is\_empty(self): # concrete method assuming abstract len  
 *"""Return True if the priority queue is empty."""* return len(self) == 0  
  
 def \_\_len\_\_(self):  
 *"""Return the number of items in the priority queue."""* raise NotImplementedError('must be implemented by subclass')  
  
 def add(self, key, value):  
 *"""Add a key-value pair."""* raise NotImplementedError('must be implemented by subclass')  
  
 def min(self):  
 *"""Return but do not remove (k,v) tuple with minimum key.  
  
 Raise Empty exception if empty.  
 """* raise NotImplementedError('must be implemented by subclass')  
  
 def remove\_min(self):  
 *"""Remove and return (k,v) tuple with minimum key.  
  
 Raise Empty exception if empty.  
 """* raise NotImplementedError('must be implemented by subclass') # #  
  
  
class UnsortedPriorityQueue(PriorityQueueBase): # base class defines \_Item  
 *"""A min-oriented priority queue implemented with an unsorted list."""* # ----------------------------- nonpublic behavior -----------------------------  
 def \_find\_min(self):  
 *"""Return Position of item with minimum key."""* if self.is\_empty(): # is\_empty inherited from base class  
 raise Exception('Priority queue is empty')  
 small = self.\_data.first()  
 walk = self.\_data.after(small)  
 while walk is not None:  
 if walk.element() < small.element():  
 small = walk  
 walk = self.\_data.after(walk)  
 return small  
  
 # ------------------------------ public behaviors ------------------------------  
 def \_\_init\_\_(self):  
 *"""Create a new empty Priority Queue."""* self.\_data = PositionalList()  
  
 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."""* self.\_data.add\_last(self.\_Item(key, value))  
  
 def min(self):  
 *"""Return but do not remove (k,v) tuple with minimum key.  
  
 Raise Empty exception if empty.  
 """* p = self.\_find\_min()  
 item = p.element()  
 return (item.\_key, item.\_value)  
  
 def remove\_min(self):  
 *"""Remove and return (k,v) tuple with minimum key.  
  
 Raise Empty exception if empty.  
 """* p = self.\_find\_min()  
 item = self.\_data.delete(p)  
 return (item.\_key)  
  
 def \_\_iter\_\_(self):  
 *"""Generate iteration of the map's keys."""* for item in self.\_data:  
 yield item # yield the KEY  
  
  
def insertion(toSort):  
 sorted= UnsortedPriorityQueue()  
 for i in range(len(toSort)):  
 sorted.add(toSort.pop(), i) #puts the values from the input to a priority queue  
  
  
 for i in range(len(sorted)):  
 toSort.append(sorted.remove\_min()) #Removeing the min each time and puts it as the next slot  
  
 return toSort  
print("Enter the first value you want sorted")  
toSort = []  
answer = int(input())  
toSort.append(answer)  
while answer != "no": #can put in many numbers into list  
 print("Enter the next number in you list, or type 'no' to finish your list")  
 answer = input()  
 if answer != "no":  
 answer = int(answer)  
 else:  
 break  
 toSort.append(answer)  
print(insertion(toSort))