```
"""
A Heap implemented by
mapping a tree onto an array (Python list) of the same size.
file: array_heap.py
language: python3
author: Matthew Fluet
author: James Heliotis
author: Arthur Nunes-Harwitt
author: Ben K Steele

new language feature: passing (and storing) functions as arguments.
"""

import copy

# Utility functions to map tree relations to array indices ###

def parent(i):
    """
       Return index of parent of node at index i.
    """
    return (i - 1)//2

def lChild(i):
    """
       Return index of left child of node at index i.
    """
    return 2*i + 1

def rChild(i):
    """
       Return index of right child of node at index i.
    """
    return 2*i + 2

############################################################################

class Heap(object):
    """
       A heap inside an array that may be bigger than the
       heapified section of said array
       SLOTS:
           array: the Python list object used to store the heap
           size: the number of array elements currently in the
                 heap. (size-1) is the index of the last element.
           compareFunc: A function to compare values in the heap.
                  For example, if compareFunc performs less-than,
                  then the heap will be a min-heap.
    """
    __slots__ = ('array', 'size', 'compareFunc')

    def __init__(self, maxSize, compareFunc):
        """
           Create an empty heap with capacity maxSize
           and comparison function compareFunc.
        """
        self.array = [None] * maxSize
        self.size = 0
        self.compareFunc = compareFunc
```

```
    def __str__(self):
        return str(self.size) + ": " + str(self.array)

def displayHeap(heap, startIndex=0, indent=""):
    """
       displayHeap : Heap * NatNum * String -> NoneType
       Display the heap as a tree with each child value indented
       from its parent value. Traverse the tree in preorder.
    """
    if startIndex < heap.size:
        print(indent + str(heap.array[startIndex]))
        displayHeap(heap, lChild(startIndex), indent + '    ')
        displayHeap(heap, rChild(startIndex), indent + '    ')

def siftUp(heap, startIndex):
    """
       siftUp : Heap * NatNum -> NoneType
       Move the value at startIndex up to its proper spot in
       the given heap. Assume that the value does not have
       to move down.
    """
    i = startIndex
    a = heap.array
    while i > 0 and not heap.compareFunc(a[parent(i)], a[i]):
        (a[parent(i)], a[i]) = (a[i], a[parent(i)])     # swap
        i = parent(i)

def _first_of_3(heap, index):
    """
    _first_of_3 : Heap * NatNum -> NatNum
    _first_of_3 is a private, utility function.
       Look at the values at:
       - index
       - the left child position of index, if in the heap
       - the right child position of index, if in the heap
       and return the index of the value that should come
       first, according to heap.compareFunc().
    """
    lt = lChild(index)
    rt = rChild(index)
    thisVal = heap.array[index]
    if rt < heap.size:        # If there are both left and right children
        lVal = heap.array[lt]
        rVal = heap.array[rt]
        if heap.compareFunc(lVal, thisVal)    \
        or heap.compareFunc(rVal, thisVal):
            if heap.compareFunc(lVal, rVal):
                return lt # The left child goes first
            else:
                return rt # The right child goes first
        else:
                return index # This one goes first
    elif lt < heap.size: # If there is only a left child
        lVal = heap.array[lt]
        if heap.compareFunc(lVal, thisVal):
            return lt # The left child goes first
        else:
            return index # This one goes first
    else: # There are no children
        return index
```

```
def siftDown(heap, startIndex):
    """
       siftDown : Heap * NatNum -> NoneType
       Move the value at startIndex down to its proper spot in
       the given heap. Assume that the value does not have
       to move up.
    """
    curIndex = startIndex
    a = heap.array
    swapIndex = _first_of_3(heap, curIndex)
    while (swapIndex != curIndex):
        (a[swapIndex], a[curIndex]) = (a[curIndex], a[swapIndex]) # swap
        curIndex = swapIndex
        swapIndex = _first_of_3(heap, curIndex)

def add(heap, newValue):
    """
       add : Heap * Comparable -> NoneType
       add inserts the element at the correct position in the heap.
    """
    if heap.size == len(heap.array):
        heap.array = heap.array + ([None] * len(heap.array))
    heap.array[heap.size] = newValue
    siftUp(heap, heap.size)
    heap.size = heap.size + 1

def removeMin(heap):
    """
       removeMin : Heap -> Comparable
       removeMin removes and returns the minimum element in the heap.
    """
    res = heap.array[0]
    heap.size = heap.size - 1
    heap.array[0] = heap.array[heap.size]
    heap.array[heap.size] = None
    siftDown(heap, 0)
    return res

def updateValue(heap, index, newValue):
    """
       Fix the heap after changing the value in one of its nodes.
    """
    oldValue = heap.array[index]
    heap.array[index] = newValue
    if heap.compareFunc(newValue, oldValue):
        siftUp(heap, index)
    else:
        siftDown(heap, index)

def top(heap):
    """
       top : Heap -> Comparable
       top returns a deep copy of the current 'top' of the heap
    """
    res = copy.deepcopy(heap.array[0])
    return res

def less(a, b):
    """
       less : Comparable * Comparable -> Boolean
       This ordering function returns True if the first value is smaller.
```

```
    """
    return a <= b

def greater(a, b):
    """
       greater : Comparable * Comparable -> Boolean
       This ordering function returns True if the first value is larger.
    """
    return a >= b

############################################################################


def testHeap( testData ):
    """
    testHeap : TupleOfComparable -> NoneType
    Create a min heap, fill it with the test data, and display it.
    """
    print( "testHeap(", testData, "):" )

    heap = Heap(len(testData), less)

    for i in range(len(testData)):
        add(heap, testData[i])
        if i % 2 == 0: print( i, "-th iteration's top:", top( heap ) )

    print("Heap size is now", heap.size)
    displayHeap(heap)
    print()

    # Perform some heap modifications. Tests updateValue().
    for (index, value) in ((1, 100), (4, -1)):
        print("Change value at position", index, "to", value)
        updateValue(heap, index, value)
        displayHeap(heap)
    print( "current top:", top( heap ) )

if __name__ == '__main__':

    testData = (1, 3, 5, 7, 9, 10, 8, 6, 4, 2, 0)     # Test data
    testHeap( testData )
```