Assignment 4

Due date: Sunday, November 17, 2019 at 11:59pm

Individual assignment. No group work allowed.
Weight: 10% of the final grade.

Implement the following:

- implement a delete algorithm for a binary search tree **(20 points)**
  - a BST class that has all other methods already implemented is attached
  - you should implement the incomplete method called remove(self, value)
  - your implementation should perform the deletion in O(lg n) runtime

- implement your own heap insert algorithm in a function called my_heappush(heap,item) **(20 points)**
  - your implementation should work exactly the same as heapq.heappush(heap, item)
  - you will need to implement a sift_up algorithm for this
  - the runtime of your implementation should be O(lg n)

- implement your own heap deletion algorithm in a function called my_heappop(heap) **(20 points)**
  - your implementation should work exactly the same as heapq.heappop(heap)
  - you will need to implement a sift_down algorithm for this
  - the run-time of your implementation should be O(lg n)

- implement your own heapify algorithm in a function called my_heapify(heap) **(20 points)**
  - your implementation should work exactly the same as heapq.heapify(heap)
  - your implementation must run in O(n) time, and be in-place, for full credit
  - for partial credit you can implement an O(n lg n) algorithm, and/or not in-place algorithm. **(-5 points)**

- implement your own heapsort algorithm in a function called my_heapsort(a) **(20 points)**
  - the algorithm must run in O(n lg n) time
  - the algorithm must be in-place
  - the algorithm should use your my_heapify() function
  - if you did not succeed in implementing my_heapify(), you can use heapq.heapify(). **(-5 points)**

You can find documentation on the Python heapq module here: https://docs.python.org/3.7/library/heapq.html

Below you can also find testing code for you heap algorithms.
Please use it to test your implementations, but do not modify the testing algorithms.

Submit your solution as an .ipynb file to D2L before the due date.

Incomplete BST class:

```
# helper function
```

```
def print_tree(n, indent = 0):
  if n is None:
    print( " " * indent, "X")
    return
  print_tree(n.right, indent + 4)
  print( " " * indent, n.val)
  print_tree(n.left, indent + 4)

# helper function
def inorder(n):
  if not n: return
  yield from inorder(n.left)
  yield n.val
  yield from inorder(n.right)

class BST:

  class Node:
    def __init__(self, val):
      self.val = val
      self.left = None
      self.right = None

  def __init__(self):
    self.root = None

  def insert(self, val):
    n = self.Node(val)
    if self.root is None:
      self.root = n
      return
    r = self.root
    while r is not None:
      if val < r.val:
        if r.left is None:
          r.left = n
          return
        else:
          r = r.left
      else:
        if r.right is None:
          r.right = n
          return
        else:
          r = r.right

  def __iter__(self):
    yield from inorder(self.root)

  def print(self):
    print("------- Tree -----------")
    if self.root is None:
      print("Empty tree")
    else:
      print_tree(self.root)
    print("------------------------")

  def remove(self, val):
    # not implemented
    pass
```

Testing code for your heap algorithms:

```
import heapq

def my_heappush(heap,value):
  # re-implement this !!!
  heapq.heappush(heap,value)

def my_heappop(heap):
  # re-implement this !!!
  return heapq.heappop(heap)

def my_heapify(heap):
  # re-implement this !!!
  heapq.heapify(heap)

def my_heapsort(heap):
  # re-implement this !!!
  heap.sort()

#
# you can use the tests below to verify your implementations are correct
# but do not modify the tests below

def is_heap(heap):
  def parent(i): return (i-1) // 2
  def left(i): return 2 * i + 1
  def right(i): return 2 * i + 2
  for i in range(0, parent(len(heap)-1)):
    if left(i) < len(heap) and heap[i] > heap[left(i)]: return False
    if right(i) < len(heap) and heap[i] > heap[right(i)]: return False
  return True

import random
def test_my_heappush(n):
  print("Testing my_heappush:")
  a = [random.randint(0,100) for x in range(n)]
  print("  [I] inserting:", a)
  heap = []
  for i in a:
    my_heappush(heap,i)
  print("  [I] heap:", heap)
  if sorted(heap) != sorted(a):
    print("  [E] incorrect elements in heap after inserting:")
  if not is_heap(heap):
    print("  [E] not a heap after inserting")

def test_my_heappop(n):
  print("Testing my_heappop:")
  a = [random.randint(0,100) for x in range(n)]
  heapq.heapify(a)
  print("  [I] testing pop on heap:", a)
  heapcopy = a[:]
  true_min = heapq.heappop(heapcopy)
  my_min = my_heappop(a)
  print("  [I] popped item:", my_min)
  print("  [I] resulting heap:", a)
  if true_min != my_min:
    print("  [I] my_heappop returned", my_min)
    print("  [I] but real min is", true_min)
  if sorted(heapcopy) != sorted(a):
    print("  [E] incorrect elements in heap after popping")
  if not is_heap(a):
    print("  [E] not a heap after popping")

def test_my_heapify(n):
  print("Testing my_heapify:")
  a = [random.randint(0,100) for x in range(n)]
  print("  [I] input array:", a)
  heapcopy = a[:]
  heapq.heapify(heapcopy)
  my_heapify(a)
  print("  [I] heap:", a)
  if sorted(heapcopy) != sorted(a):
    print("  [E] incorrect elements in heap after heapify")
  if not is_heap(a):
    print("  [E] not a heap after heapify")

def test_my_heapsort(n):
  print("Testing my_heapsort")
  a = [random.randint(0,100) for x in range(n)]
  copy = a[:]
  my_heapsort(a)
  print("  [I] input array:", copy)
  print("  [I] sorted output:", a)
  if sorted(a) != sorted(copy):
    print("  [E] incorrect elements in sorted array")
  if a != sorted(a):
    print("  [E] not sorted")

def test_all(n):
  test_my_heappush(n)
  test_my_heappop(n)
  test_my_heapify(n)
  test_my_heapsort(n)

test_all(9)
```