Student: John Rizzo

Course: CS590-A Algorithms

Instructor: Dr. William Hendrix

Due Date: February 27, 2025

Description: Homework 3 Algorithms

Problem 1

What new field(s) does the data structure need?

The new solution requires that the root node is augmented to store the minimum value, such as in node.minval.

Problem 2

Give pseudo code for the min operation for the BST.

```
def BST.min():
    node = root
    if node != None:
        return node.minval
```

Problem 3

Give pseudo code for the insert operation. Reference pseudo code for the insert method appears below.

```
def BST.insert(newvalue, root):
    node = root
    while node != None:
        if node.value <= newvalue:
            if root.left == None:
                root.left = node
            node = node.left
            if root.minval > node.value:
                root.minval = node.value
        else:
            if root.right = None:
                node = node.right
            node = node.right
```

Problem 4

```
def BST.delete(node):
    if node.left != None and node.right != None:
        swapnode = right
        while swapnode.left != None:
            swapnode = swapnode.left
        Swap node's parent and children links with swapnode
        if node is self.root:
            root = swapnode
    if node.left == None and node.right == None:
        if node == self.root:
            root = None
        else:
            node.parent.left = None
            node.parent.right = None
    else:
        # Node must have one child
        if node == self.root:
            Set root to be node's child
        else:
            set node.parent's child to be node's child
        Set node's child's parent to be node.parent
        Find the minimum value from the root node
        set the root's min to be the minimum value
```

Problem 5

Give pseudo code for an efficient algorithm for the *top k* search problem. In top-k search, you are given an array of n integers and must return the k largest integers, where k is generally much smaller than n. Acceptable algorithms might be $O(n + klgn)$ or $O(nlgk)$, but not $O(nk)$ or $O(nlgn)$. *Hint* use an appropriate data structure!

Given the following heap data structure, we can implement a top-k search algorithm, which is $O(n + klgn)$.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, element):
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)

    def max(self):
        if not self.heap:
            return None
        return self.heap[0]

    def extract_max(self):
        if len(self.heap) == 0:
            return None
        if len(self.heap) == 1: return self.heap.pop()
        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return root

    def _heapify_up(self, index):
        parent_index = (index - 1) // 2
        if index > 0 and self.heap[index] > self.heap[parent_index]:
            self.heap[index], self.heap[parent_index] =
                self.heap[parent_index], self.heap[index]
            self._heapify_up(parent_index)

    def _heapify_down(self, index):
        largest = index
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2

        if left_child_index < len(self.heap) and
            self.heap[left_child_index] > self.heap[largest]:
                largest = left_child_index

        if right_child_index < len(self.heap) and
            self.heap[right_child_index] > self.heap[largest]:
                largest = right_child_index

        if largest != index:
            self.heap[index], self.heap[largest] =
                self.heap[largest], self.heap[index]
            self._heapify_down(largest)

def topk(arr):
    heap = 0
    result = 0

    for i = 0 to n:
        heap.insert(arr[i])

    for i = 1 to (k+1):
        max = heap.max()
        heap.delete(max)
        result.insert(max)

    return result
```