Daniel Oliveros A02093272
March 20, 2018

Assignment 4: Data Structure Design

Problem #1 k-th element smaller than x

1. **Algorithm Description:** This algorithm works by making a decision as to whether or not we should check the children of a node based on the value of that node. If a node is bigger than or equal to x, we can assume its children are bigger than x, so we don't care about checking them. If a node is smaller than x, we can check its children and see if they are also smaller than it. We first set up a queue to hold the root node of our tree, and then run a loop that adds nodes to the queue if they are smaller than x. This loop runs until we've either run out of nodes in the queue or our count of nodes smaller than x is of size k. Based on the value of count once the loop is done and whether or not the queue is empty, we can determine if we should return True or False.

2. **Pseudocode:**

```
import heapq
import queue

def problem1(heap, x, k):
    q = queue.Queue(len(heap))
    q.put(0) # add first element to queue
    count = 0
    while(q.empty()!=True and count<k): #run until no more elements are
        ↪ found or the # of elements found is the same as k
        val = q.get()
        if(heap[val]<x):#element is smaller than x, increase count
            count+=1
            #check if the index for children is valid and add them to the
                ↪ queue
            if((2*val)+1<=len(heap)-1):
                q.put((2*val)+1)
            if((2*val)+2<=len(heap)-1):
                q.put((2*val)+2)
    if(q.empty() and count<k):
        return False
    else:
        return True

#testing code
nums = [1, 45, 12, 7, 30, 41]
heapq.heapify(nums)
print(nums)
print(problem1(nums, 46, 6))
```

3. **Correctness:** This program only increases the value of count when the node visited has a value smaller than the x given, this means that we optimize the process of finding if the k-th smallest number is smaller than x by determining if there are k elements smaller than x in the first place. If we determine there are less than k elements smaller than x, which happens when our queue is emptied without count reaching the value of k, then we know x must be smaller than the k-th smallest number in our heap. We never check the children of nodes that are greater than x, since these children are also guaranteed to be greater than x.

4. **Time Analysis:** The while loop runs until either the queue has been emptied, or count has increased to the point where it is equal to k. The operations performed within this loop are all of constant time, making it so the total runtime is based solely on the number of iterations the loop runs for. The worst case scenario that can happen is we are given a k equal to the height, and on each check we travel down in a way that adds a node that is smaller than x, and a node that is

greater than x. In this scenario, we end up checking 2*k nodes, which is still $O(k)$. For this function then, we can determine our total runtime is $O(k)$

Problem #2 rank of x

1. **Algorithm Description:** This algorithm works by performing an operation similar to search. We augment our tree so that each node stores a variable named size, which refers to the number of nodes in the subtree that given node is a root of. When we perform the search, we determine whether or not the nodes at the left of our node being analyzed are bound to be smaller than x, if so, we add the variable node.left.size to our total count for rank.
   Augmenting the tree in this way does not increase the runtime of Search, Insert, or Delete. This is due to the fact that the operations that need to be performed to keep this variable updated can be performed in constant time when Inserting and Deleting, which means the total runtime of these operations isn't affected significantly.

2. **Pseudocode:**

```
import queue

def problem2(tree, x):
    q = queue.Queue(tree.size())
    q.put(tree.root())
    rank = 1
    while (q.empty() != True):#a new node is added to this queue as we go
        ↪ down into the tree, only stopping once the node added has no
        ↪ children
        node = q.get()
        if(node.key<x):#if node.key < x all the nodes in its left subtree
            ↪ are smaller than x, add those and look to its right
            rank += 1
            if(node.left is not None):
                rank+= node.left.size
            if(node.right is not None):
                q.put(node.right)
        elif(node.key == x):#if node.key == k all nodes to its left are
            ↪ smaller than x
            if(node.left is not None):
                rank+= node.left.size
        else:#if node.key > x, none of the nodes in its right side are
            ↪ smaller than x, only need to look to its left side
            if(node.left is not None):
                q.put(node.left)
    return rank
```

3. **Correctness:** This algorithm does not increase the value of rank unless it is clear that the left subtree of a given node is populated only by elements smaller than x. This is because all nodes in the left subtree of a node must be smaller than the subroot, so by knowing the value of that subroot we can assume that these nodes hold smaller values. In the case of Figure 1 from the assignment description, when we perform the operation rank(21) this is what happens. First, we look at the root node, since its value is greater than 21 we know for a fact the right subtree has no nodes smaller than 21, so we look at the subtree to its left. For the second subtree, with a root of 16, 16¡21, we add one to rank since the root is smaller than 21, and we add the value of the size variable of the left subtree, which is 2, rank now has a value of 4, finally, we move onto the right subtree from this node. In this subtree, with a root of 20, the same happens as in the previous scenario, we add one to rank for the root and one from the left subtree, rank has a value of 6 now and we move onto the subtree to its right. For the subtree with 23 as its root, since 23¿21 and there are no nodes to the left of 23, no new node is added to our queue and it's now empty. We finish the loop with rank = 6.

4. **Time Analysis:** This algorithm runs until the queue is empty, and only has the potential of adding one element to the queue per iteration. this means that it goes down into the tree until it either finds

a node with no children or a node with the value of x, in the worst case scenario, it goes until it reaches the bottom-most node. This means that it's worst case scenario is $O(h)$, where h is the height of the tree.

Problem #3 Range print in order

1. **Algorithm Description:** This algorithm traverses the tree down from the root visiting the paths following the in-order concept of first checking the left side of a tree, then printing the node itself, and then its right side. This algorithm avoids checking subtrees that are bound to contain elements outside of the range we want to work with. This means that the algorithm traverses the tree until finding nodes within the range we want, and then traverses the area within those boundaries from smallest to greatest until reaching the upper bound.

2. **Pseudocode:**

```
def rangeTraverse(node, xl, xr):
    if node is None:#Base case, node does not exist
        return
    if node.key < xl:#value is smaller than left limit, only right side is
    ↪  worth checking
        rangeTraverse(node.right, xl, xr)
    if node.key > xr:#value is greater than right limit, only left side is
    ↪  worth checking
        rangeTraverse(node.left, xl, xr)
    else:# if the value is in the range, print its structure in order
        rangeTraverse(node.left, xl, xr)
        print(node.key)
        rangeTraverse(node.right, xl, xr)

def problem3(tree, xl, xr):
    if tree is not None:
        rangeTraverse(tree.root, xl, xr)
```

3. **Correctness:** This algorithm avoids visiting regions of our tree that are outside of the range we want to work with, visits each node only once, and prints the nodes it finds within the range in order, meaning that it is always expected to work correctly. It bases its assumptions of which subtrees are not worth checking based on the knowledge that no nodes to the left of a given node have a value greater than it, and no node to its right has a value smaller than it.

4. **Time Analysis:** This algorithm travels down the tree until finding nodes within the range we want to work with, this means that in the case where no nodes are within this range it travels down a distance of h, in the other extreme case, where every node is within this range, this algorithm visits every single node in the tree. This means that the algorithm has a complexity dependent on the tree we're working with and the number of nodes that can be found within the range $[x_l, x_r]$. The total runtime for this algorithm is, therefore, $O(h + k)$

Problem #4 Sum of nodes in range

1. **Algorithm Description:** This algorithm requires us to augment the tree so that every node stores a sum of the entire subtree it is a root of. This augmentation does not increase the total runtime of Insert, Delete, or search. These functions can easily update these variables when they are completed. This algorithm goes down the paths of both boundaries of our range and decrease the value of the total sum based on which subtrees we know are not part of our range.

2. **Pseudocode:**

```
import queue
def problem4(tree, xl, xr):
    q = queue.Queue(2*tree.size)
    q.put(tree.root)
    rangeSum = tree.root.sum
    while(q.empty != True): #decrease the sum of subtrees we're not
        ↪ interested in on the left
        node = q.get()
        if(node.key < xl):
            rangeSum -= node.key
            if(node.left is not None):
                rangeSum -= node.left.sum
            if(node.right is not None):
                q.put(node.right)
        elif(node.key == xl):
            if(node.left is not None):
                rangeSum -= node.left.sum
        else:
            if(node.left is not None):
                q.put(node.left)
    q.put(tree.root)
    while(q.empty != True): #decrease the sum of subtrees we're not
        ↪ interested in on the right
        node = q.get()
        if(node.key > xl):
            rangeSum -= node.key
            if(node.right is not None):
                rangeSum -= node.right.sum
            if(node.left is not None):
                q.put(node.left)
        elif(node.key == xr):
            if(node.right is not None):
                rangeSum -= node.right.sum
        else:
            if(node.right is not None):
                q.put(node.right)
    return rangeSum
```

3. **Correctness:** This algorithm determines which parts of the tree aren't in our range and reduces the total sum we're working with by removing the sum of these parts. It applies a principle similar to the principle of Inclusion and Exclusion by constantly checking which parts of the tree we want and which ones we don't

4. **Time Analysis:** Each of the while loops runs until their given queues are emptied. Each loop runs down through the tree until it either finds a node with the value it's looking for, or it reaches a node with no children. In the worst case scenario, each loop will run until it reaches the deepest node in the tree, in which case its runtime is simply $O(h)$. The total runtime of this algorithm is simply $O(2h) = O(h)$

People I worked with for this assignment:

- Erik Davtyan