CSC263: Problem Set 2

September 24, 2019

1 Problem 1

(a) Let Q be an array whose indices start from 1. Assume all keys in Q are distinct integers.

```
1 ExtractSecondLargest(Q):
2     if Q.length < 2 then
3         return None
4     else then
5         first = EXTRACT_MAX(Q)
6         second = EXTRACT_MAX(Q)
7         INSERT(Q, first)
8         return second
```

(b) Assume the length of array, Q, is greater than 2. Then we enter the else conditional on line 5 of the code. Line 5 of the code first extracts the element of Q with the largest key, and stores it in the variable $first$. Line 6 of the code, extracts the element with the largest key of the modified array, Q, and stores it in the variable *second*. We note here that *second* is the element of Q with the second largest key, when compared with the original Q. Note that at this point, Q is missing two elements with the largest keys, since they were extracted in lines 5 and 6. Then on line 7, the INSERT function, inserts the element with the largest key from the original Q back into Q. Thus, the resulting Q is only missing the element with the second largest key. In Line 8 of the code, the element with the second largest key is returned, thus it has been extracted from Q.

We note that the heap structure is preserved at every line because of the functions used to create $EXTRACTSECONDLARGEST(Q)$. First, we focus on the $EXTRACT\_MAX(Q)$ function. This function preserves the heap structure through the $MAX\_HEAPIFY(Q, i)$ function, where i is an index into the array. Once the maximal element has been extracted, the $MAX\_HEAPIFY(Q, i)$ function works by determining the largest of elements $Q[i]$, $Q[LEFT(i)]$, and $Q[RIGHT(i)]$, and the largest element's index is stored. If $Q[i]$ is the largest, then the subtree rooted at node i is already a max heap and so the function terminates. Otherwise, if one of the two children are larger then $Q[i]$ is swapped with the largest of its' children ($Q[LEFT(i)]$ or $Q[RIGHT(i)]$). This causes the original node at index i and its children to satisfy the max-heap property. However, the node indexed by one of the children now has the original value $Q[i]$ and the subtree now rooted at one of the children could possibly violate the max-heap property and so $MAX\_HEAPIFY$ is called recursively on that subtree. In this way, the heap structure is maintained.

Next, we focus on the $INSERT(Q, x)$ function, where x is an element that is to be inserted into the array, Q. The element is inserted at the very end of the array, i.e. a new leaf to the tree, but because of the $INCREASE\_KEY(Q, i, key)$ (where i is an index into the array) function call inside $INSERT(Q, x)$, the heap structure is maintained because $INCREASE\_KEY(Q, i, key)$ sets the key of the new node to its correct value and thus maintains the max-heap property.

In this way, at the end of each line of code that is executed in the else condition, the heap structure is maintained.

(c) Suppose Q has a length of $n > 2$. Then we skip lines 2 and 3 of the code which run in constant time and enter the else conditional. From lecture and the textbook, we know that $EXTRACT\_MAX(Q)$ and $INSERT(Q, x)$ run in $O(log(n))$ time (see pages 163 and 164 from CLRS). Thus, from the pseudo-code we can see that on lines 5 and 6 we run $EXTRACT\_MAX(Q)$ twice and on line 7 $INSERT(Q, x)$, thus we get a run-time of $clog(n)$, where c is some constant. Accounting for other lines of code like the return statement on line 8, we get $T(n) = clog(n) + d \epsilon O(log(n))$, for some constant d.