CSC265 Assignment 4

Written by Kaiyang Wen, 1002123891, kai.wen@mail.utoronto.ca
Revised by Mohan Zhang, 1002748716, morgan.zhang@mail.utoronto.ca

Question 1.

(part a)

We are going to perform the following steps in sequence.

(step 1) Construct a complete binary tree rooted at **r**. The keys of the leaves, denoted **leaf.first.key** are the element in set A; the root of each pair of leaves (aka. height-1 internal node) stores the result of comparison, denote them **root.first.key** and **root.second.key** with the former at least the latter. For those roots, define **root.second.prototype** to be null.

(step 2) For each non-height-1 internal node of the binary tree, the root of each pair of subtrees stores the result of comparison between **left.first.key** and **right.first.key** into variables **root.first.key** and **root.second.key** with the former at least the latter.

The pointer to the root **x** of the subtree with **x.first.key** equals **root.first.key** will be stored into **root.second.versus**.

The pointer to the root **y** of the subtree with **y.first.key** equals **root.second.key** will be stored into **root.second.prototype**.

These pointers are the auxiliary variables.

After step 2, **r.first.key** is the maximum of set A. Now we want to find the second maximum.

(step 3) Run FindSecondLargest(r.second). The returned value will be the second maximum.

```
; A recursive function which finds the second largest.
FindSecondLargest(u):
    if u.prototype is null or u.versus.second.key < u.key:
        return u.key
    else:
        return FindSecondLargest(u.versus.second)
```

(claim) Step 1 through 3 have done n - 2 + $\log_2 n$ comparisons in total.

proof

In step 1 and 2, the total number of comparisons equals the number of internal nodes inside the complete binary tree rooted at r. So, there are total of n - 1 comparisons done in step 1, since the number of internal nodes of a complete binary tree equals the number of leaves minus 1.

In step 3, I claim that the maximum number of comparisons equals the number of recursive calls to FindSecondLargest. In the worst case scenario, the if-test inside the function fails, and we have to call FindSecondLargest(u) until u.prototype is null, which will happen when u is located inside a height-1 internal node. See the specification in step 1 about those height-1 internal nodes.

Each recursive call is analogous to ”jumping down” one step of the height, so there are total of $(\log_2 n)$ - 1 recursive calls.

Combining all results, we obtain the total number of comparisons equals n - 2 + $\log_2 n$.This finishes the proof.

(claim) Step 1 through step 3 are correct.

proof

Step 1 and 2 are trivial to prove, since picking the maximum over the set of maximums will guarantee the selected value will be the actual maximum of the union of two sets.

For proving step 3, consider an induction on the size of A, with the hypothesis that FindSecondLargest(u) on any subset of A having the shape of a complete binary tree (specified in the procedure) rooted at u, will return the second largest element of A.

When the size of A is 2, the auxiliary pointer will be null, and the set only has two elements, so we have the correct returned value.

Consider the size of A is more than 2:

When the current parameter u has a key greater than the ”versus.second” of u (something that’s smaller than the global maximum but does not have any direct/indirect comparison with u), then since that u.versus is at least as large as the subtrees rooted at u.versus.second, we can directly return u as the second maximum.

Otherwise, the recursion call guarantees a correct return value by induction hypothesis.

By the time complexity analysis above, we have shown that the procedure indeed terminates.

(part b)

Let M denotes the largest element in an array, and let m denotes the second largest.

We have an algorithm for an adversery whose output will be "<" or ">" with two indices of two elements in A[1 . . n] as input, while making sure the algorithm will perform at least n − 2 + $\lceil log_2 n \rceil$ comparisons.

```
// The algorithm will return either "P<Q" or "P>Q".
// The algorithm uses a global variable F, which is a forest (think of directed
// graph) and initially every element in A[1 . . n] is a 0-height-tree in F.
Adversary_Algorithm_AA (P, Q):
    if count(P.child)== count(Q.child):
        let A be a child of B
        return "P<Q"
    else if count(P.child)<count(Q.child):
        let P be a child of Q
        return "P<Q"
    else
        // Note that now count(P.child)>count(Q.child)
        let Q be a child of P
        return "P>Q"

// Trees in this forest has the property that the number of children is less
// than those of parent... call the number of children of each root RC
Init_forest( A[1 . . n]):
    create a forest F with every element in A[1 . . n] being a 0-height-tree.
```

Facts:

1. When there's only one tree in the forest F of adversary's algorithm, we have found M, since everything is less than this root.

2. To make all nodes in F being rooted at one common node (call it M) (we assume that M exists), any algorithm we use to find M and m must perform at least n-1 comparisons. (Imagine putting everything besides M as the children/descendents of M.)

3. To build a tree with RC=r at least 0 (for some arbitrary r), the only way is to to execute $Adversary_Algorithm_A$ on two trees having RC $\geq$ r-1. (Since the adversery will provide a result to detach all children from one root and attach those to another.)

4. The adversery's algorithm makes sure that every root of the tree has as many children as possible.

5. The parents will always have more number of children than that of their children. (It follows directly from the definition of such algorithm: "count(P.child)>count(Q.child) then let Q be a child of P".)

(claim) Any tree in F, whose RC=r, must contain at least 2^r nodes.

proof (by induction)

(base case)

When r = 0, there is only 1 node, which satisfies the claim.

When r = 1 , the only way to build this tree is to execute $Adversary_Algorithm_A$ on two trees having RC $\geq$ 0. So, the number of nodes is at least 2.

(induction-step)

Assume for RC = k, the tree contains at least 2^r nodes to be the induction hypothesis.

Then, for RC = k + 1, the only way to build such tree is to execute $Adversary_Algorithm_A$ on two trees having RC $\geq$ k. So, the number of nodes is at least $2^k + 2^k = 2^{k+1}$. This satisfies the claim, so it finishes the proof. From the former claim, we know that every tree whose children does not have multiple parents will have RC = $\lceil log_2 n \rceil$

Therefore, any algorithm which finds M and m, after performing all the necessary comparisons with the algorithm input specified by the adversary (which is the output of the adversary's algorithm), forest F will have only one tree (rooted at a single node), and whose node will have only one child (which is m).

Moreover, (for n $\geq$ 3) by the facts stated above, with any algorithm that will finally create one tree in F, the RC for that tree will be greater than 1.

So, the "final-version" tree in F will be created by performing $Adversary_Algorithm_A$ on roots' children, and by the claim, we know that this will take at least $\lceil log_2 n \rceil$-1 comparisons (in order to to remove multiple parents).

So, the total number of comparisons will be at least n-1 + $\lceil log_2 n \rceil$ -1.

Q.E.D..

Question 2.

(part a)

Assume the swap operation takes constant time. After the first if-block, consider H_a is rooted at r1 and H_b is rooted at r2.

(invariant) After the first if-block, if r1.left is not null, then the Union call in the else-block runs in O($l(H_a)$ + $l(H_b)$ - 1).

proof (by induction on the height of H_a)

(base case)

When the height of H_a is 1, then we are calling Union(r1.left, r2) in the else-block. Consider r1.left.key is less than every element in the left-most path of H_b... then for all recursive calls Union(r1.left, st), where st denotes the root of any subtrees of the left-most path of H_b, the test on the first if-test succeeds and the second if-test fails until it reaches a leaf of H_b. By definition of l, the number of recursion calls equals $l(H_b)$. Thus, in this case the recursive call runs in O($l(H_a)$ + $l(H_b)$ - 1).

For any other cases, at the k'th recursive call, where k $\leq l(H_b)$, the first if-test fails and the second if-test succeeds; a corresponding subtree will be connected to r1.left.left. No more operations will be performed afterwards. Therefore, the function still runs in O($l(H_a)$ + $l(H_b)$ - 1).

(induction step)

Assume that $l(H_u)$ equals $l(H_a)$ + 1. Notice that H_u is defined after the frist if-block.

Let the S-Heap H_v be rooted at r1.left, satisfying $l(H_v) = l(H_u) - 1 = l(H_u)$. By induction hypothesis, the sub-recursive call, Union(r1.left.left, r2) or Union(r2.left, r1.left), runs in O($l(H_v)$ + $l(H_b)$ - 1) = O($l(H_u)$ + $l(H_b)$ - 2).

Accumulating the recursive call at the else-block during the current call, we obtain that the Union call in the else-block runs in O($l(H_u)$ + $l(H_b)$ - 1).

Using the invariant and accumulating the current function call, we obtain that Union runs in O($l(H_u)$ + $l(H_b)$).

(part b)