CSC265 F18: Assignment 2

Due: October 10, by midnight

Guidelines: (read fully!!)

- Your assignment solution must be submitted as a *typed* PDF document. Scanned handwritten solutions, solutions in any other format, or unreadable solutions will **not** be accepted or marked. You are encouraged to learn the LaTeX typesetting system and use it to type your solution. See the course website for LaTeX resources.
- Your submission should be no more than 7 pages long, in a single-column US Letter or A4 page format, using at least 9 pt font and 1 inch margins.
- To submit this assignment, use the MarkUs system, at URL `https://markus.teach.cs.toronto.edu/csc265-2018-09`
- This is a *group assignment*. This means that you can work on this assignment with *at most one other* student. You are *strongly encouraged* to work with a partner. Both partners in the group should work on and arrive at the solution together. Both partners receive the same mark on this assignment.
- Work on all problems together. For each problem, one of you should write the solution, and one should proof-read and revise it. The first page of your submission must list the *name*, *student ID*, and *UTOR email address* of both group members. It should also list, for each problem, which group member wrote the problem, and which group member proof-read and revised it.
- You **may not** consult any other resources except: your partner; your class notes; your textbook and assigned readings. *Consulting any other resource, or collaborating with students other than your group partner, is a violation of the academic integrity policy!*
- You may use any data structure, algorithm, or theorem previously studied in class, or in one of the prerequisites of this course, by just referring to it, and without describing it. This includes any data structure, algorithm, or theorem we covered in lecture, in a tutorial, or in any of the assigned readings. Be sure to give a *precise reference* for the data structure/algorithm/result you are using.
- Unless stated otherwise, you should justify all your answers using rigorous arguments. Your solution will be marked based both on its completeness and correctness, and also on the clarity and precision of your explanation.

Question 1. (16 marks)

An IMPORTANCE TREE is a variant of a Binary Search Tree (BST), in which each element, in addition to its key, is also given an importance number, which is an integer. A valid IMPORTANCE TREE must satisfy all the properties of a BST, and, moreover, the importance number of any tree node must be greater than or equal to the importance numbers of its children: we call this the *importance property*.

In the following questions you can assume that in an IMPORTANCE TREE T, every node u has fields $u.key$, $u.imp$, $u.parent$, $u.lchild$, $u.rchild$, storing, respectively, the key, the importance number, and pointers to the parent, left, and right child of u.

Part a. (7 marks)

An IMPORTANCE TREE T supports an INSERT(key, imp) operation, which inserts into T a new element with key set to key and importance set to imp. Describe how to implement INSERT so that its running time is $O(h)$ where h is the height of T. Specify your algorithm in clear and precise English (and, optionally, pseudocode), and justify its running time and also why it preserves the properties of an IMPORTANCE TREE.

HINT: First insert as in any BST. Then use left and right rotations to make sure that the importance property is satisfied.

[Solution]

As in the hint, first insert as in any BST. Let the new node be u. Then if the importance of the parent p of u is at least the importance of u, we are done. Otherwise, if p has lower importance than u, and u is a left child of p, perform a right rotation; if u is a right child of p, perform a left rotation. Finally, recurse on the new root of the subtree, going up the tree until the importance property is satisfied. The pseudocode is below.

INSERT(key, imp)

```
1  Insert a new node u with key key and importance imp using the BST Insert procedure.
2  while u is not the root of T
3      p = u.parent
4      if p.imp < u.imp and u == p.lchild
5          Perform a right rotation
6          u is the new root of the subtree
7      elseif p.imp < u.imp and u == p.rchild
8          Perform a left rotation
9          u is the new root of the subtree
10     else return
```

To prove the correctness of this procedure, we will show the invariant that at any point in the execution of the algorithm the importance property is satisfied in the subtree rooted at u, and, moreover, any ancestor of u (i.e. node on the path from u to the root) has importance greater than or equal to any node in the subtree rooted at u (except possibly u itself). This is true initially, because u is inserted as a leaf. For the inductive step we need to consider three cases: (i) we do a right rotation, (ii) we do a left rotation, and (iii) we do no rotations. In case (iii) the inductive step is trivial, and case (ii) is analogous to (i), so let's focus on (i). In case (i), the subtree at u changes as in Figure 1. Because we only do a rotation if $p.imp < u.imp$, and because the importance property was satisfied in the old subtree rooted at u by the induction hypothesis, we have that the importance property is satisfied at u. Moreover, it is also satisfied at p after the rotation, because by the induction hypothesis, the importance of any node in T_2 is at most that of p, and the same is true for any node in T_3 by the importance property in the original tree. This shows that the importance property is satisfied in the subtree rooted at u after the rotation. Moreover, the only new node in the subtree rooted at u is p, and by the importance property in the original tree T, its importance number is at most that of any ancestor of u. This finishes the proof of the inductive step.

When INSERT exits, we have that either the importance of the parent of u is at least that of u, or that u is the root of the tree. In either case, the invariant we just proved implies that the importance property holds

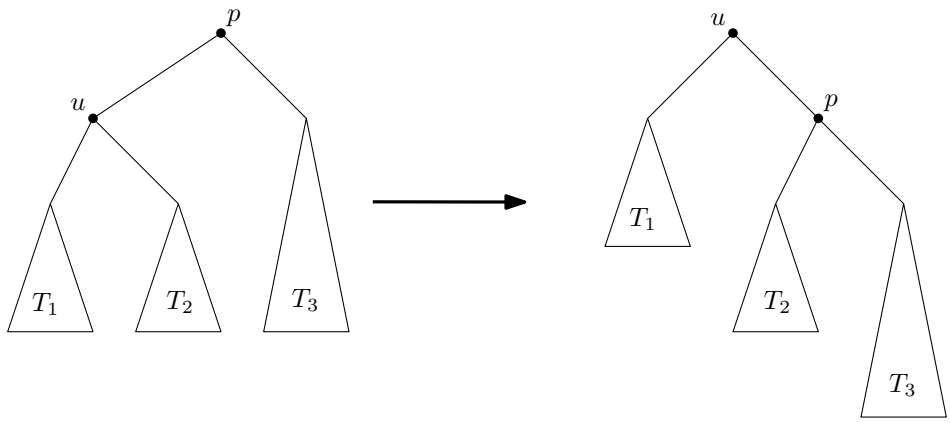

Figure 1: A right rotation at u.

everywhere.

To finish the proof of correctness, we need to show that the BST property is also satisfied after insertion. This is true by the correctness of the BST Insert procedure, and because rotations preserve the BST property.

The BST Insert procedure takes time $O(h)$, where h is the height of T. Then observe that every time we perform a rotation, the distance from u to the root of T decreases by 1. Therefore, we can perform at most $h+1$ rotations, and each rotation takes constant time. So, the total running time is $O(h)$.

Part b. (9 marks)

Suppose you are given two IMPORTANCE TREE data structures T_1 and T_2, so that every element in T_2 has key greater than that of every element of T_1. Describe a procedure MERGE(T_1, T_2) which returns a new IMPORTANCE TREE T containing the union of the elements from T_1 and T_2. The procedure should run in time $O(h_1+h_2)$, where h_1, h_2 are, respectively, the heights of T_1 and T_2. Specify your algorithm implementing MERGE in clear and precise English (and, optionally, pseudocode), and justify its running time and also why it returns a valid IMPORTANCE TREE T containing the elements of T_1 and T_2. You can assume that MERGE takes pointers to the roots of T_1 and T_2 as input, and returns a pointer to the new merged tree.

[Solution]

Find the minimum element of T_2. We do this by starting at the root of T_2 and following pointers to left children until we reach a node with no left child. Let the node we find this way be u: it follows from the definition of a BST that u contains the element with the minimum key, because it is the first node output in an in-order traversal of T_2. We remove u from T_2: if it has a right child v, we can just attach v to the parent of u. Then we make u the root of T, with T_1 as its left subtree, and T_2 as its right subtree. The BST property is satisfied because u contained the minimum element of T_2 and is bigger than any element of T_1 by assumption.

It remains to satisfy the importance property. We do this via rotations again. Let v and w be, respectively, the left and right child of u (i.e. the old roots of T_1 and T_2, respectively.) If $u.imp \geq \max\{v.imp, w.imp\}$, then we are done. Otherwise, we perform a right rotation if $v.imp \geq w.imp$, and a left rotation otherwise. Then we repeat this procedure at the new location of u (after the rotation), until the importance property becomes satisfied.

To show this procedure establishes the importance property we will prove a similar invariant to that in the first subquestion: at any point in the execution of the algorithm the importance property is satisfied in the left and right subtree of u, and, moreover, any ancestor of u (i.e. node on the path from u to the root) has importance greater than or equal to any node in the subtree rooted at u, including u itself. This is true initially, because the two subtrees under u are T_1 and T_2 and they both satisfy the importance property, and u is the root so it has no ancestors. Once again, to prove the inductive step, we can focus on the case of a right rotation: the left rotation is analogous, and if we do not do a rotation, then there is nothing to

prove. See Figure 2 for the result of the rotation. It follows immediately from the induction hypothesis that the importance property holds for the left and right subtrees at u. The only new ancestor of u after the rotation is v, and, since we performed a right rotation, the importance of v is at least that of both u and w. The importance of v is at least that of any node in T_{11} and T_{12} since they were subtrees below v before the rotation, and also at least that of any node in T_2, since w is the root of T_2 and the importance of v is at least that of w. This finishes the proof of the inductive step.

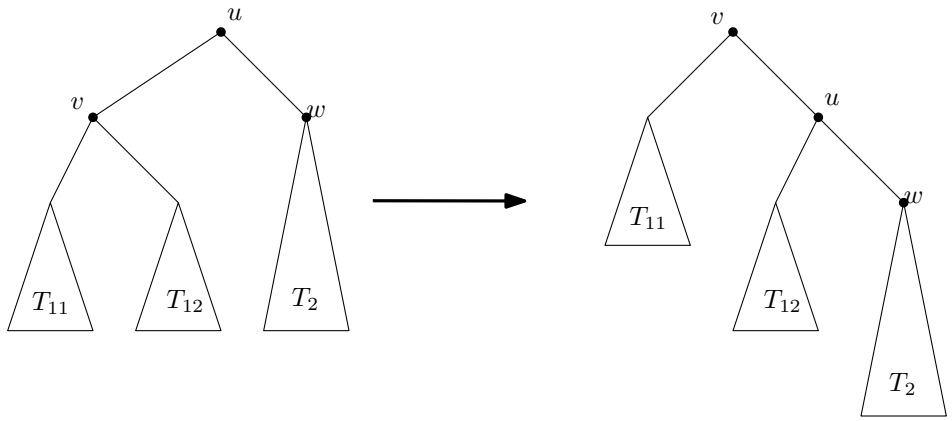

Figure 2: A right rotation making the left child of u its parent.

When the procedure above exits, we have that the importance of u is at least that of its children, or u is a leaf. In either case, we have that the importance property holds in the entire tree by the invariant we just proved.

To analyze the running time, observe first that finding the minimum of T_2 takes time $O(h_2)$. Then, let $h_L(u)$ be the height of the left subtree of u, and let $h_R(u)$ be the height of the right subtree of u. Initially $h_L(u) + h_R(u) = h_1 + h_2$. Each rotation decreases $h_L(u) + h_R(u)$ by at least one: in a right rotation $h_L(u)$ decreases by at least one, and in a left rotation $h_R(u)$ decreases by at least one. When $h_L(u) + h_R(u) = 0$ the node u has become a leaf and the algorithm quits. Therefore, the algorithm runs in worst-case time $O(h_1 + h_2)$.

Question 2. (12 marks)

Recall the Quicksort algorithm. In particular, we will use the QUICKSORT(A, p, r) algorithm described in Chapter 7 of the textbook. This algorithm takes an unsorted array $A[1\,..\,n]$ of integers, and two indexes p and r. At completion it has sorted the subarray $A[p\,..\,r]$. For completeness, the algorithm is given below. (This pseudocode is the same as in the textbook.)

```
QUICKSORT(A, p, r)
1  if p < r
2        q = PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Here, the PARTITION(A, p, r) procedure selects $x = A[r]$ as the ***pivot***. It rearranges the subarray $A[p\,..\,r]$ so that all its entries which have a key less than or equal to x are placed before x, and all entries which have a

key greater than x are placed after x. Then it returns q which is the new index of x after the rearrangement. See the book for more details.

For the purposes of this question, when QUICKSORT(A, r, r) is called, we consider $A[r]$ as the pivot, even though PARTITION is not called in this case. With this definition, you can check that every element of A becomes a pivot at some point during the execution of QUICKSORT$(A, 1, n)$.

Important Observation. Notice that the only time QUICKSORT(A, p, r) compares two entries of A is during the PARTITION procedure when every element of $A[p\,..\,r-1]$ is compared with the pivot $x = A[r]$.

Assume that all entries of A are distinct Let us list the the entries in the array A as $x_1, \ldots, x_n$, in the order in which they become pivots during an execution of QUICKSORT$(A, 1, n)$. Give x_i an importance score $n - i + 1$, so that the first element that becomes a pivot has score n, the next one $n - 1$ and so on. Let T be a IMPORTANCE TREE (as defined in the previous question) whose elements have $x_1, \ldots, x_n$ as keys, and the element with key x_i has importance score $n - i + 1$ as just defined.

Give an algorithm which, given as input *only* the IMPORTANCE TREE T defined above, computes in worst-case running time $O(n)$ the exact number of comparisons between entries of A performed by QUICKSORT$(A, 1, n)$. Describe your algorithm in clear and precise English, and also using pseudocode. Justify why it correctly computes the number of comparisons, and why it runs in time $O(n)$.

[Solution]

For any node u of T, let $level(u)$ be the distance from u to the root of T. We claim that the total number of comparisons performed by QUICKSORT$(A, 1, n)$ equals

$$\sum_{u \in T} level(u). \tag{1}$$

We show this by induction on the height of T. In the base case when T has height 0, the equality is obvious, as there are no comparisons performed and the sum (1) is 0. For the inductive step, let u be the root node of T, with importance n and key x. Then x is the first pivot, and the left subtree T_1 of u contains the entries of A less than x, and the right subtree T_2 contains the entries of A greater than x. The nodes of T_2 have importance numbers $1, \ldots, n - q$, in reverse order of the order in which they become pivots. Then, by the induction hypothesis, the number of times the entries in $A[q+1\,..\,n]$ (after A is modified by PARTITION) are compared is equal to the sum of their distances to the root of T_2. Since the root of T_2 is at distance 1 from u, this sum is

$$\sum_{v \in T_2} (level(v) - 1) = \sum_{v \in T_2} level(v) - (n - q). \tag{2}$$

Similarly, the nodes of T_1 have importance numbers $n-q+1, \ldots n-1$, in reverse order of the order in which they become pivots. Then T_1 is isomorphic to a IMPORTANCE TREE in which the nodes have importance numbers $1, \ldots, q-1$ in the same order. Again, by the induction hypothesis, the number of times the entries in $A[1\,..\,q-1]$ (after A is modified by PARTITION) are compared is equal to

$$\sum_{v \in T_1} (level(v) - 1) = \sum_{v \in T_1} level(v) - (q - 1). \tag{3}$$

Finally, we also have $n - 1$ comparisons between the pivot x and all other entries of A performed during the first call to PARTITION. Adding this to (2) plus (3) yields (1), and completes the proof.

The algorithm to compute (1) is given by the following pseudocode. Assume that NUMCOMPS takes a pointer to the root of T. It returns a pair of numbers: the sum in (1) and the number of nodes in T.

```
NumComps(u)
1  if u == Nil
2       return (0, 0)
3  else (L, n_L) = NumComps(u.lchild)
4       (R, n_R) = NumComps(u.rchild)
5       S = L + R + n_L + n_R
6       n = n_L + n_R + 1
7       return (S, n)
```

The correctness of this algorithm follows by a straightforward induction argument along the lines of our proof above. For the running time, notice that the algorithm is called at most once per node, and executes a constant number of operations.