CSC265 W23: Assignment 4

Due: November 17, 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 6 pages long, in a single-column US Letter or A4 page format, using at least 10 pt font and 1 inch margins.
- To submit this assignment, use the MarkUs system, at URL `https://markus.teach.cs.toronto.edu/2023-09`
- 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. (11 marks)

In this question we define a type or mergeable (min-)heap data structure, and give an amortized analysis of its complexity.

The basic building block of the data structure is the left-ordered tree. A left-ordered tree is a binary tree whose nodes store keys of elements in the heap, and which satisfies the following properties

- the root of the tree has only a left child;
- for every node u of the tree, all nodes in the left subtree of u have key at least as large as the key of u.

With each node u of a left ordered tree we store its key in $u.key$, pointers to its left and right child, in, respectively $u.left$ and $u.right$, and also the height of u (i.e., distance to furthest leaf) in $u.h$.

A basic operation is linking two left-ordered trees T_1 and T_2, which produces a new left-ordered tree T. Let r_1 be the root of T_1, and r_2 the root of T_2. If $r_1.key \leq r_2.key$, then we make r_1 the root of T, r_2 becomes the left child of r_1, and the old left subtree S_1 of r_1 becomes the new right subtree of r_2. We also update the heights of r_1 and r_2. See Figure 1. The construction is symmetrical if $r_2.key \leq r_1.key$, with r_2 becoming the root of T, r_1 its left child, and the left subtree S_2 or r_2 becoming the new right subtree of r_1.

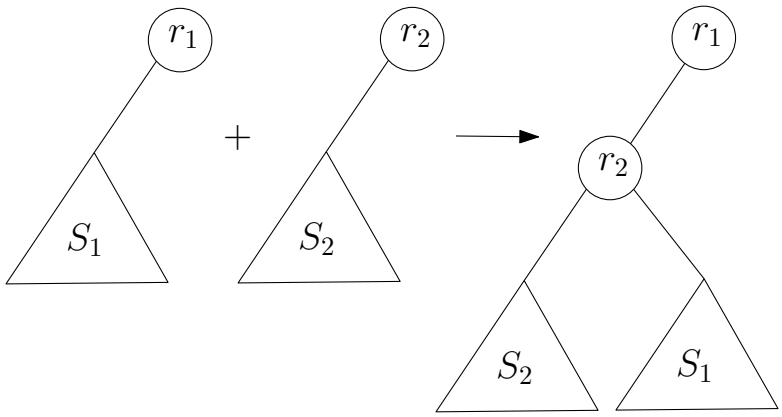

Figure 1: Linking two left-ordered trees

The LOF-heap data structure stores all the keys of its elements in a collection of disjoint left-ordered trees. Moreover, each of the trees in the collection is perfect, i.e., the left subtree of the root is a perfect binary tree (which means that every node except the leaves has exactly two children, *and all the leaves are at equal distance from the root*). The roots of the trees are linked in a circular doubly linked list, and the data structure also stores a pointer to the tree root with the smallest key.

To do a UNION of two LOF-heaps, we simply take the union of the two sets of trees by concatenating the linked lists, and update the pointer to the minimum. To do a INSERT we create a heap with one element and take its union with the old heap. The most complicated operation is EXTRACT-MIN, which works as follows:

1. Remove the root with the minimum key, and divide the remaining tree into left-ordered trees as follows. Let $u_0, \ldots, u_h$ be the nodes on the rightmost path from the root to a leaf, where u_0 is the root of the remaining tree, i.e., the left child of the removed old root. Each u_i, together with its left subtree forms a left ordered tree: see Figure 2. We disconnect these trees (and update the heights of their roots as necessary), and add them to the linked list of left-ordered trees in the data structure. In the process, we have destroyed the old tree with the removed minimum root, and it is no longer in the list.
2. Now create as many disjoint pairs $(T_1, T_1'), \ldots, (T_k, T_k')$ of left ordered trees as possible, so that in each pair T_i and T_i' have the same height. At most one tree of any given height should be left out from the pairs.

3. For each two trees in a pair T_i and T_i', link them into a new tree T_i'', and replace them with T_i'' in the list of left-ordered trees. Update the minimum key pointer after all the k link operations are done.

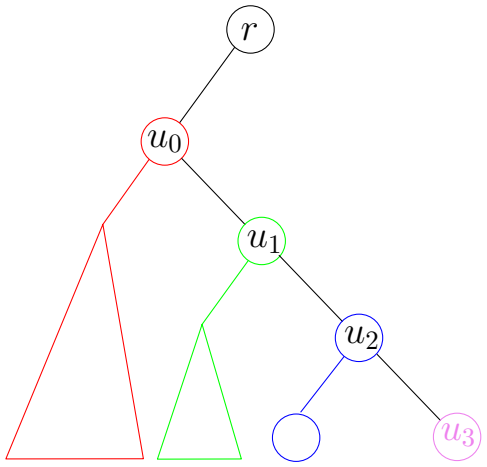

Figure 2: Dividing the tree whose root has minimum key into smaller left-ordered trees. In this case the tree initially has height 4, and height 3 after the root is removed. The four trees colored red, green, blue, and purple get disconnected and added to the list of left-ordered trees.

Part a. (1 marks)

Show that linking two perfect left-ordered trees of the same height gives a perfect left-ordered tree.

Part b. (4 marks)

Show that the EXTRACT-MIN procedure above can be implemented in worst-case running time $O(\ell + \log n)$ where n is the total number of elements, and ℓ is the number of left ordered trees in the data structure before EXTRACT-MIN is performed. Justify the correctness and running time of your implementation.

Part c. (6 marks)

Using the potential function method, show that in any sequence of n operations on an initially empty LOF-heap the amortized complexity of any INSERT is $O(1)$ and the amortized complexity of EXTRACT-MIN is $O(\log n)$. Clearly specify your potential function, prove that it is 0 for an empty heap, and always non-negative, and use it to bound the amortized complexity. As the actual cost of an INSERT, you can use 1, and as the actual cost of EXTRACT-MIN you can use $\log_2(n) + \ell$, where ℓ is the number of left-ordered trees in the data structure before the procedure is executed.

Question 2. (5 marks)

Recall that during an AVL-INSERT(x) operation in an AVL tree, we first insert the key x into the tree as we do in any binary search tree (BST). Then we go up the path from the new node to the root and update the balance factor fields until we either reach a node which becomes perfectly balanced after the insertion, or we have to do a single or a double rotation. (Review the notes on AVL trees.)

This second part of an AVL-INSERT(x) – updating balance factors and doing rotations – can, in the worst case, double the time it takes to insert a new node. Your goal in this question is to show that this is not true in an amortized sense.

Use the potential function method to show that in any sequence of m consecutive AVL-INSERT(x) operations, starting from an empty AVL tree, the total number of balance factor updates is bounded by $O(m)$. I.e. if t_i is the number of nodes whose balance factor is changed during the i-th consecutive insert (not counting the new node), show that $\sum_{i=1}^{m} t_i = O(m)$. Define the potential you are using, and show that it is 0 for an empty AVL tree, and is never negative. Derive the claimed bound on $\sum_{i=1}^{m} t_i$.