CSC265 Homework 4

Question 1 (Mark: 11/11)

Part a. (Mark: 1/1)

Lemma. Given two disjoint perfect binary trees S_1 and S_2 of the same height, along with a node r that is disjoint from both trees, if we assign S_1 as the left subtree of r and S_2 as the right subtree, then the tree rooted at r forms a perfect binary tree.
Proof. By assigning S_1 and S_2 as subtrees to r, the depth of every leaf in both S_1 and S_2 increases by 1. Since their original depths were equal, they depth remain equal after this assignment. Each internal node in S_1 and S_2 initially had exactly two children, and now, with the addition of r, r also has two children. Consequently, all internal nodes in the tree rooted at r have exactly two children. This completes the proof. $\square$

By definition, a left-ordered tree is perfect if and only if the left subtree of the root is a perfect binary tree.

Suppose we have two perfect left-ordered trees with roots r_1 and r_2, along with left subtrees S_1 and S_2, respectively, all of equal height. Without loss of generality, let's assume $r_1.key \leq r_2.key$. After linking, S_1 and S_2 remain unchanged, preserving their structure as perfect binary trees. S_1 becomes the new left subtree of r_2, and S_2 becomes the right subtree of r_2. Consequently, the tree rooted at r_2 maintains its perfect binary tree property, as established by the lemma.

As r_2 becomes the new left child of r_1, the tree rooted at r_1 is a perfect left-ordered tree. Therefore, linking two perfect left-ordered trees of the same height results in a new perfect left-ordered tree. $\square$

Part b. (Mark: 4/4)
We will first outline the implementation of the Extract-Min operation.

- Step 1: Remove the root with the minimum key from the circular doubly linked list. Then we traversing down the root through the rightmost path and labeling the nodes on the path as $u_0, \dots, u_h$. On the backward traversal from u_h, for each u_i with $i > 0$, disconnect it from u_{i-1} and update its height to $h - i$. Subsequently, union u_i into the circular linked list. The result leaves us with u_0 which is the root. As u_1 was its left subtree and is now disconnected, we save the value of u_0, which will be returned at the end of the Extract-Min operation.

- Step 2: Initially, traverse the linked list to record the maximum height h among all trees. Initialize an array $A[0 \dots h]$ and an empty linked list L. Traverse the linked list again. For each root r, if $A[r.h]$ is empty, assign the pointer of r to $A[r.h]$. Otherwise, add a pair of pointers $(A[r.h], r)$ to the head of L, then set $A[r.h]$ to be empty.

- Step 3: Traverse through L. For each pair (r_1, r_2), remove them from the circular doubly linked list. Link r_1 and r_2 to form a new left-ordered tree, and union it into the circular doubly linked list. Subsequently, traverse through the circular doubly linked list while recording a pointer to the root with the minimum value. Finally, store this pointer in the data structure, and return the recorded value of u_0.

Now, we will provide justification for the correctness.

In step 1, according to the property of a left-ordered tree, the value of the root is smaller than all values stored in its left subtree. As u_0 contains the minimum value among all roots, it indeed holds the minimum value of the data structure.

In step 2, as per our implementation, the pointer of each root is either stored in A or L. For each height, since A can only hold one pointer, at most one tree of the same height will be left out from pairing.

In step 3, the correctness inherits from the linking and union operations. Moreover, this step preserves the correctness of the pointer that points to the root of the minimum value.

Finally, we calculate the worst-case complexity. Let n be the total number of elements, and ℓ be the number of left ordered trees in the data structure before Extract-Min is performed.

In step 1, accessing the root and removing it from the circular doubly linked list involves a constant number of pointer assignments, which requires constant time. Since each tree is perfectly left-ordered, its height h is constrained by $\log_2 n$. Therefore, traversing down and backward takes $O(\log n)$. For each node in the backward traverse, we execute a constant number of pointer assignments and variable assignments, along with one union operation, all taking constant time. Thus, this step has a worst-case time complexity of $O(\log n)$.

Since at most $\log_2 n$ unions are executed so far, the data structure now contains at most $\ell + \log_2 n$ trees.

In step 2, traversing through the list of trees takes $O(\ell + \log n)$. As h is bounded by $\log_2 n$, initializing A takes $O(\log n)$. Initializing L takes constant time. Traversing through the list again takes $O(\ell + \log n)$. For each tree, we perform a constant number of array accesses, taking constant time, and at most one linked list insertion at the head. Insertion at the head of the linked list involves only a constant number of pointer assignments, taking constant time. Therefore, this step takes worst-case $O(\ell + \log n)$.

In step 3, since no pointers in L are duplicated, the number of pairs is bounded by $\ell + \log_2 n$. For each pair, linking and taking the union takes constant time. Therefore, it takes a maximum of $O(\ell + \log n)$. As every two trees are replaced by one, the total number of trees is non-increasing, therefore still bounded by $\ell + \log_2 n$. Therefore, traversing the tree while keeping the pointer to the root of the minimum value takes at most $O(\ell + \log n)$. Adding things up, this step takes a worst-case $O(\ell + \log n)$.

By summing the worst-case complexities for all three steps, we conclude that our implementation of Extract-Min has a worst-case complexity of $O(\ell + \log n)$ as desired.

Part c. (Mark: 6/6)
Consider an arbitrary sequence of n operations on an initially empty LOF-heap.
Let D_i represent the LOF-heap after applying the i-th operation to D_{i-1}, where D_0 is empty.

Define $\Phi(D_i)$ as twice the number of left-ordered trees in the heap. Since D_0 is initially empty and the number of trees in the heap is never negative, we have $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i. Therefore the total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

If the i-th operation is an Insert, it introduces a new left-ordered tree to the heap. Therefore, $\Phi(D_i) - \Phi(D_{i-1}) = 2$. Assuming the actual cost c_i of Insert is 1, we obtain the amortized cost $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 3$, which is $O(1)$ as desired.

If the i-th operation is an Extract-Min, we analyze it in three phases progressively. Suppose ℓ is the number of left-ordered trees in the heap before the procedure is executed, then $\Phi(D_{i-1}) = 2\ell$.

- Step 1: Given that each left-ordered tree has a complete binary tree as its left child and an empty right child, its size is a power of 2. Since the number of items is bounded above by n, the maximum height of a left-ordered tree is less or equal to $\log_2 n$. Consequently, removing the root and splitting the tree leads to, at most, an increase in the total number of trees by $\log_2 n$. In an alternative scenario, removing the root of a left-ordered tree with a height of zero results in the deletion of a tree, consequently reducing the overall number of trees by 1.

These are the extreme cases, therefore the number of left-ordered trees after step 1 is $\ell' \in [\ell - 1, \ell + \log_2 n]$ which implies that the potential is $\Phi(D'_{i+1}) \in [2\ell - 2, 2\ell + 2\log_2 n]$

- Step 2: We form pairs maximally, ensuring that at most one tree of any given height is excluded from the pairs. As the total number of elements is less than n, there are at most $\log_2 n$ distinct heights. Thus, a maximum of $\log_2 n$ trees remains unpaired. Consequently, we have a minimum of $\dfrac{\ell' - \log_2 n}{2}$ pairs.

- Step 3: For each pair, the two trees are replaced by a new tree, leading to a decrease in the total number of trees by 1. Consequently, the count of left-ordered trees after step 3, denoted as ℓ'', falls within the range:

$$\ell'' \leq \ell' - \frac{\ell' - \log_2 n}{2} = \frac{\ell' + \log_2 n}{2} \leq \frac{(\ell + \log_2 n) + \log_2 n}{2} = \frac{\ell + 2\log_2 n}{2}$$

Therefore we have $\Phi(D_i) \leq \ell + 2\log_2 n \implies \Phi(D_i) - \Phi(D_{i-1}) \leq -\ell + 2\log_2 n$

By assuming the actual cost c_i of Extract-Min is $\ell + \log_2 n$, we have

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (\ell + \log_2 n) + (-\ell + 2\log_2 n) = 3\log_2 n$$

Therefore the amortized cost of Extract-Min is $O(\log n)$ as desired. □

Question 2 (Mark: 5/5)

Consider an arbitrary sequence of m consecutive AVL-Insert(x) operations.
Let D_i be the AVL tree after applying the i-th insertion to D_{i-1}, where D_0 is an empty tree.

Define $\Phi(D_i)$ to be the number of nodes in D_i that have a balance factor of zero. Thus, $\Phi(D_0) = 0$ since D_0 is empty, and $\Phi(D_i)$ is non-negative for all i. Therefore the total amortized cost of m operations with respect to Φ therefore represents an upper bound on the actual cost.

Let's consider the i-th insertion. Let t_i be the number of balance factor updates.

- Step 1: Create a new node with a balance factor of zero, thereby increasing the potential by 1.

- Step 2: If the newly created node is the root, terminate the process. Otherwise, proceed to its parent node.

- Step 3: Repeat the following:

  Begin by updating the balance factor, contributing 1 to t_i. Then, consider the following cases:

  Case a: If the balance factor is now zero, halt.

  Case b: Perform a single rotation, then halt. This contributes 2 to t_i.

  Case c: Perform a double rotation, then halt. This contributes 3 to t_i.

  Case d: If the current node is the root, halt.

  Case e: Otherwise, move to the current node's parent.

Now we analyze the amortized cost $\widehat{t_i}$ for the i-th insertion.

If the process terminates at step 2, then $\Phi(D_i) = \Phi(D_{i-1})+1$, with $t_i = 0$. This implies $\widehat{t_i} = t_i+\Phi(D_i)-\Phi(D_{i-1}) = 1$

Otherwise, let's assume that Step 3 is repeated n_i times. In other words, Case e is executed $n_i - 1$ times, then concluding with one of the cases a-d. As each iteration initially updates a balance factor, this contributes n_i to the actual cost. If Case e or d is executed after updating the balance factor, it implies that the update changed the current node's balance factor from 0 to ± 1 (since none of cases a-c are executed), resulting in a decrease in potential by 1. Therefore, the potential is decreased by $n_i - 1$ in the first $n_i - 1$ iterations.

Now, let's consider the last iteration: If the process concludes with case a, the balance factor is updated from ± 1 to 0, increasing the potential by 1. If the process concludes with case b [resp. c], then by the property of the rotation operation, two [resp. three] nodes that are non-zero are updated to be zero, increasing the potential by 2 [resp. 3]. If the process concludes with case d, then as explained in the preceding paragraph, the potential is decreased by 1.

By summing the contributions from the first $n_i - 1$ iterations and each of the last iteration cases, we obtain $t_i \leq n_i + 3$ and $\Phi(D_i) - \Phi(D_{i-1}) \leq 1 - (n_i - 1) + 3 = 5 - n_i$. Therefore $\widehat{t_i} = t_i + \Phi(D_i) - \Phi(D_{i-1}) \leq 8$

By definition of $\Phi(D_m)$, it is bounded above by the total number of nodes in D_m, which is m.

Thus we have $\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} \widehat{t_i} - \Phi(D_m) + \Phi(D_0) \leq 8m - m = 7m$, which implies $\sum_{i=1}^{m} t_i = O(m)$ as desired. $\square$