Assignment 2
CPSC 331

Guransh Mangat, 30061719
Daniel Contreras, 10080311
Steven Ferguson, 30037518

Problem 1. Describe how an algorithm for the “Treap Restoration after Insertion” problem could be used, along with an algorithm for insertion into a binary search tree, to obtain an algorithm for the “Insertion into This Treap” problem. Briefly explain why the algorithm you have described would be correct

Solution. The “insertion into a binary search tree” can be used along with “Insertion into Subtree” algorithm. Whenever “insertion into subtree”” algorithm is called with the preconditions satisfied, a node is included into the subtree with root x- either by replacing the value at the node storing k, or by adding a new node if no such node existed.The “insertion into the subtree” algorithm ensures that each element stored in a node in the left subtree of the subtree with root x is *less than* the element stored at x, and each element stored in a node in the right subtree of the subtree with root x is *greater than* the element stored at x.

Once “insertion into binary search tree” algorithm is executed, the elements E int the Treap T are stored satisfying property (b) of the definition of Treaps.

We then continue with executing “Treap Restoration after Insertion” algorithm with the preconditions satisfied.This ensured that he priority associated with each element of E, that is stored in Treap T, has not been changed. That is, if an element $e \in E$ and priority $p \in P$ were stored in the same node of T before this computation began, then e and p are also both stored in the same node of T after the computation ends.

Once “Treap Restoration after Insertion” algorithm is executed, the priorities P are stored in the Treap T satisfying property (c) of the definition of the Treaps. ■

Problem 2. Suppose that x is the left child of z, so that the subtree of T with root z is as shown in Figure 2. Show that if a right rotation at z is performed — resulting in subtree as shown in Figure 3 — then the precondition for the “Treap Restoration after Insertion” problem is still satisfied (using x as the input node, once again), the set of elements of E stored at nodes of T has not been changed, and the priority associated with each element of E stored in T has not been changed, either.

Solution. ■

Problem 3.

```
    private void restoreAfterInsertion(TreapNode x) {
        if (x.priority.compareTo(x.parent.priority()) == 1) {
            if (x.parent.left() != null && x.parent.left.equals(x)) {
                rightRotate(x.parent);
            } else {
                leftRotate(x.parent);
            }
            if (x.parent() != null) {
                restoreAfterInsertion(x);
            }
        }
```

Problem 4. Briefly describe how it can be shown that your algorithm is correct. Describe any proof techniques that are used in a proof of correctness along with loop invariants and bound functions for any loops in your code, bound functions for any recursive algorithm(s) you include, and describe how results that have already been established are used.

Solution. A proof that the restoreAfterDeletion algorithm is correct aims to establish the claims made by the preconditions and postconditions. The proof makes use of the strong form of mathematical induction on the depth of the subtree with the non-null node x as input. Furthermore, the depth of the subtree that has the node x as the root can be used as a bound function for this recursive algorithm. Two main cases will be considered in the proof,

1. when the priority of the node x is greater than the priority of its parents priority, and
2. the case where the priority of the node x is less than or equal to the priority of its parents priority.

In case 1, we further consider two sub-cases,

1. the node x is the left leaf node of its parent, in which case a right rotation is performed on the node x and its parent.
2. the node x is the right leaf node of its parent, in which case a left rotation is performed on the node x and its parent.

Naturally, we would induct on the depth that the node x travels to complete the restoration. And in case 2, the algorithm would simply halt as there is no restoration required. The proof would conclude in showing that when the execution of the algorithm halts, no additional data has been changed, namely the data stored in the sets E and P of the tree T and that T also satisfies all the Treap Properties.

■

Problem 5. Establish upper bounds for the number of steps executed by your algorithm and the storage space it requires — as functions of the size or depth of T or of a property of the input node x (which you should describe). Asymptotic notation can be used to answer this question — and your answer for this question should not be very long!

Solution. Since the bound function for our restoreAfterDeletion algorithm is the depth of the subtree that has the node x as its root, then the upper bound of the algorithm can be said to be linear in the depth of the subtree. Hence, the upper bound for the algorithm is $\mathcal{O}(n)$. ■

Problem 6. Show that if x has at most one child in this treap then the Treap properties are satisfied after the regular deletion operation ends — so that nothing more must be done to satisfy the post condition for the above problem in this case too.

Solution. Suppose the `delete` method is executed with the with the precondition for the Treap Deletion initially satisfied, so that `T` satisfies the Treap properties and `x` is the node storing a non-null input key of type `E` that is found successfully within the Treap. Since `x` is non-null, then `T` must have at least one node such that the root is non-null as well, and additionally, since we are assuming that key is indeed found, and `x` is the value that stores the input key, then `x` exists, and a NoSuchElementException would not be thrown in this case. The deleteFromSubtree method is then called, where the node to be deleted is returned, which we will call `z`. Finally, execution is returned to the `delete` method, whereby `restoreAfterDeletion` is performed on the node `z` that was returned from the delete operation. We have three cases:

Case 1: left and right node at `z` are both null:
If `z` has no children, and the parent of `z` is null, then `z` is the root node, in which case the root would be set to null. In this case, `T` is now an empty tree, and satisfies the Treap definition. If `z` is has a non-null parent and the value of `z` is less than the value of its parent, then the left node at `z`'s parent is set to null. Otherwise, if the value of `z` is greater than the value of its parent, then the right node at `z`'s parent is set to null. Since `z` was a leaf node, and each node in `T` is still an ordered pair (e, p) where $e \in E$ and $p \in P$, and the value of `z` was less than its parent, then `T` satisfies both (a) and (b) of the Treap definition. Finally, Since the priority of each node in the tree has not been modified, (c) is satisfied as well.

Case 2: Left child at `z` is null but right child is not
In this case, `z` has a null left node but a non-null right node. If `z`'s parent is null, then `z` is the root, the parent node of `z`'s rightmost child is set to null, and the root is updated to `z`'s rightmost child. Since `z` was the root, and its children satisfy property (b) of the Treap definition, then (b) is indeed satisfied after its rightmost child has been updated to the root. If `z` does have a parent, and the value at `z` is less than its parent, then the left node at `z`'s parent is set to the right node at `z`, otherwise the right node at `z`'s parent is set to the right node at `z`. Finally, the parent of `z`'s right node is set to `z`'s own parent. Since each node in `T` is still an ordered pair (e, p) where $e \in E$ and $p \in P$, and the value of `z` was less than its parent, then `T` satisfies both (a) and (b) of the Treap definition. Finally, Since the priority of each node in the tree has not been modified, (c) is satisfied as well.

Case 3: Left child at `z` is not null but right child is
In this case, `z` does not have a left child but has a right child. If the parent of `z` is null , then `z` is the root node, and the parent of `z`'s leftmost child is set to null, whereby the root of the tree is updated to `z`'s leftmost child. Since `z` was the root, and its children satisfy property (b) of the Treap definition, then (b) is indeed satisfied after its leftmost child has been updated to the root. If the parent at `z` is not null, and the value at `z` is less than its parent, then the left node at `z`'s parent is set to the current left child of `z`, otherwise if the value at `z` is greater than its parent, then the right node at `z`'s parent is set to the current left child of `z`. Finally, the parent of `z`'s left node is set to `z`'s own parent. Since each node in `T` is still an ordered pair (e, p) where $e \in E$ and $p \in P$, and the value of `z` was less than its parent, then `T` satisfies both (a) and (b) of the Treap definition. Finally, Since the priority of each node in the tree has not been modified, (c) is satisfied as well.

Since we are only evaluating if the node `x` being deleted has at most 1 child in the Treap, then we do not need to consider the case if `x` has both a non-null left and right child, as no rotation is required in this case. ■

Problem 7. Suppose that the precondition for the above property is satisfied, x has a non-null left child, the priority stored at the left child of x is greater than the priority stored at x, and the priority stored at the left child of x is greater than or equal to the priority stored at the right child of x, if this child exists (and is not null). Show that, if a right rotation is performed at x (so that the left child of x becomes the parent of x) then the precondition for the "Treap Restoration after Deletion" problem is satisfied (using the same node, x) once again.

Solution. Suppose the precondition for the "Treap Restoration after Deletion" problem is satisfied, and a right rotation is performed at `x` such that `x` has a non-null left child, and the priority of the left child at `x` is greater than or equal to the priority at the right child of `x`, if it exists. When `rightRotate` is called where `x` is passed as input, the leftmost child of `x` will be updated to the rightmost child of the leftmost child of `x`. If the rightmost child of `x`'s leftmost child is not null, then the parent of this node will be set to `x`. Finally, the parent of `x`'s leftmost child will be set to `x`'s current parent. If `x` does not have a parent, then the root of the tree will become `x`'s leftmost child. Otherwise, if `x` is the same node stored at the rightmost child of `x`'s parent, then the rightmost child of `x` is set to the leftmost child of `x`. If they are not the same node, then the leftmost child at `x`'s parent is set to the leftmost child at `x`. Finally, the rightmost child of `x`'s leftmost child is set to `x`, and the parent of `x` is set to the original leftmost child of `x`. ■

Problem 8.

```
private void restoreAfterDeletion(TreapNode x) {
    if (x.left == null) {
        if (x.right == null) {
            // case 1: leaf node or root
            if (x.parent == null) {
                root = null;
            } else {
                TreapNode parent = x.parent;
                if (x.element.compareTo(parent.element) == -1) {
                    parent.left = null;
                } else {
                    parent.right = null;
                }
            }
        } else {
            // case 2: left child is null but right child is not
            TreapNode rightChild = x.right;
```

```
                if (x.parent == null) {
                    rightChild.parent = null;
                    root = rightChild;
                } else {
                    TreapNode parent = x.parent;
                    if (x.element.compareTo(parent.element) == -1) {
                        parent.left = rightChild;
                    } else {
                        parent.right = rightChild;
                    }
                    rightChild.parent = parent;
                }
            }
        } else if (x.right == null) {
            // case 3: left child is not null but right child is
            TreapNode leftChild = x.left;
            if (x.parent == null) {
                leftChild.parent = null;
                root = leftChild;
            } else {
                TreapNode parent = x.parent;
                if (x.element.compareTo(parent.element) == -1) {
                    parent.left = leftChild;
                } else {
                    parent.right = leftChild;
                }
                leftChild.parent = parent;
            }
        } else {
            // case 4: Neither the left nor the right of x is null (need to rotate)
            if (x.left.priority.compareTo(x.right.priority) == 1) {
                rightRotate(x);
            } else {
                leftRotate(x);
            }
            restoreAfterDeletion(x);
        }
    }
```

Problem 9.

Solution. ■

Problem 10.

Solution. ■

Problem 11.