CPSC 331: Assignment 3

Matthew Allwright	Karim Beyk	Seth Campbell
30037812	30027342	10152719

December 5, 2018

Question 1:
Prove the following claim, and then prove the correctness of the `bubbleDown` algorithm.

Proof 1:

Claim:

If the precondition for the "MaxHeap Restoration after Deletion" problem is satisfied and the `bubbleDown` algorithm is executed (with the input node x, as described) then this execution of the algorithm eventually ends, and the postcondition for the "MaxHeap Restoration after Deletion" problem is satisfied on termination.

Proof:

Suppose that the precondition of the "Max Heap Restoration after Deletion" is satisfied, and the `bubbleDown` algorithm is executed. That is, suppose H is a binary tree with positive size whose nodes store non-null values from some ordered type T with the shape of the heap represented by using an array A, x is a non-null input node in H, and suppose for every node y in H except for x, if z is a child of y, then the value stored at y is greater than or equal to the value stored at z, and the value stored at y is greater than or equal to the value stored at any children of z.

This gives us 3 cases:

Case 1: x has no children

If x is a node with no children, then the tests at lines 1 and 13 will both fail, and the algorithm will end with the max heap property satisfied as a result of the lack of children. No values of nodes will have been modified.

Case 2: x has 1 child

If x is a node with one child, then the test at line 1 will fail, and the test at line 13 will be executed and pass. This is because by definition, if a node has one child in a heap, that child will be a left child.

If the left child of x is less than or equal to x, the algorithm halts with the max heap property satisfied, and no values of nodes have been modified.

If the left child of x is greater than x, the test at line 14 will pass, and the values of x and the left child of x will be swapped, but not modified, which can be seen by inspection of lines 15 through 17. As a result of the swap, the left child of x now has a value less than x, and the algorithm will recurse with a value that is smaller than the the original starting value, and at a higher (*i.e.* deeper) level. The cases presented are then applied recursively and thus the algorithm will eventually halt as explained.

Case 3: x has 2 children

If x is a node with two children, then the test at line 1 will pass.

If the left child of x is less than the value of the right child of x, and the right child of x is less than the value of x, then x is greater than both of its children, and the tests on lines 2 and 8 will both fail, and the algorithm will halt with the max heap property satisfied and no modifications to any node values.

If the left child of x is greater than or equal to the right child of x, the test at line 2 will pass.

If the left child of x is also greater than x, then the test at line 3 will pass, and the values of both x and the left child of x will be swapped, but not modified which can be seen by inspection of lines 4 through 6. As a result of the swap, the left child of x now has a value less than x, and the algorithm will recurse with a value that is smaller than the the original starting value, and at a higher (*i.e.* deeper) level. The cases presented are then applied recursively and thus the algorithm will eventually halt as explained.

If the right child is greater than the left child, then the test at line 3 will fail, and the test at line 8 will be executed.

If the right child of x is also greater than x, then the test at line 8 will pass, and the values of x and the right child of x will be swapped but not modified, which can be seen by inspection of lines 9 through 11. As a result of the swap, the right child of x now has a value less than x, and the algorithm will recurse with a value that is smaller than the the original starting value, and at a higher (*i.e.* deeper) level. The cases presented are then applied recursively and thus the algorithm will eventually halt as explained.

Thus we have established the partial correctness of this algorithm by proving that in every possible case that satisfies the precondition, the algorithm will eventually halt with the postcondition satisfied.

Proof 2:

Claim:

The `bubbleDown` algorithm correctly solves the "MaxHeap Restoration after Deletion" problem.

Proof:

In accordance with the proof of partial correctness above, it is clear by inspection of the code that no undocumented side-effects are present, and no undocumented changes to global data have been made.

Thus, this algorithm correctly solves the "MaxHeap Restoration after Deletion" problem.

Question 2:
Complete the `ArrayMaxHeap.java` program.

The program is completed and provided in the file `ArrayMaxHeap.java`.

Question 3:
Complete the `HeapSort.java` program.

The program is completed and provided in the file `HeapSort.java`.

Question 4:
Describe the changes you would need to make in order update a reference to the most recently insterted node as a new node is being added during an insert operation.

In the `deleteMin` algorithm, the `predeccessor` algorithm is used to navigate the tree and find the node before the current latest. This is done by moving up the tree until a certain "junction" is reached, where the node's sibling is selected, and then you move down the tree until the preceding node is reached. In this specific algorithm, a "junction" is defined to be where the current node is a *right* child. Because it is a *right* child, the parent must also have a *left* child. After swapping to this node, you follow the *right* child until it is a leaf. This leaf is then the predecessor.

For the `insert` algorithm, a variant of the `predecessor` algorithm, the `successorParent` algorithm, is used instead. This is used to find the node which shall become the parent of the new node. The difference from the `predecessor` algorithm is minimal. Instead of moving up until a *right* child is reached, you move up until a *left* child is reached. Then, on the way down, you swap directions again. This time, instead of following the *right* child, you follow the *left* child until it is a leaf.

Thus, the only difference in these algorithms is the direction of travel up and down the tree.

Question 5:
Complete the `TreeMinHeap.java` program.

The program is completed and provided in the file `TreeMinHeap.java`.