**Module on the Heap Data Structure**

Kimberly Lluz

1. **Abstract**

This module aims to enhance understanding on Heap data structures by implementing a simple Java program which allows users to create a heap using 0 or n input items. Users of this program may then insert and or delete nodes to or from the heap and the program shows the resulting heap. The specifications of this application were chosen to allow us to spend more time on computational thinking while figuring out and communicating well-annotated, structured, and mathematically correct solutions to quantitative problems.

1. **Introduction**

A heap is a partially ordered complete binary tree. To say that a heap is partially ordered is to say that there is some relationship between the value of a node and the values of its children. In a max-heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree. In the other hand, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree in a min-heap. This module aims to implement the heaps and their operations using Java.

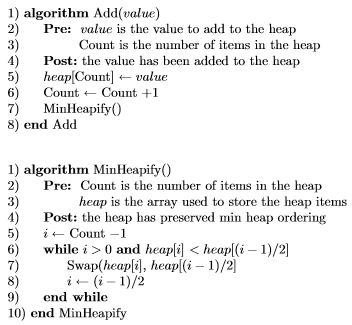
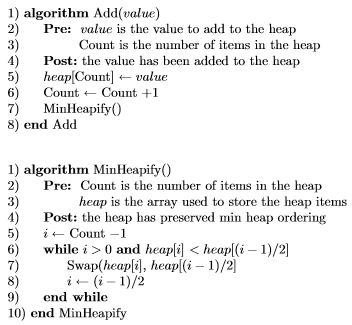
1. **Backgrounder**

A heap is a partially sorted binary tree. Although a heap is not completely in order, it conforms to a sorting principle: every node has a value less or more either of its children. Additionally, a heap is a complete tree. A complete tree is one in which there are no gaps between leaves. For instance, a tree with a root node that has only one child must have its child as the left node. More precisely, a complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks.

A binary heap is a data structure which is designed to ensure the minimum element can always be accessed in constant time and removed in log(n) time. Elements can also be added in log(n) time. A binary heap is typically represented as array. We can calculate the left child of index (n) by doing (2n + 1), and the right child with (2n + 2). The parent of any node is ((n-1)/2). This will make sense if you draw out a binary tree and label the nodes left-to-right, row-by-row 0, 1, 2, 3, etc. You will see the node left of 0 is 1 (2\*0 + 1). The node left of 1 is 3 (2\*1 + 1), and the node right of 1 is 4 (2\*1 + 2), etc.

Insertion

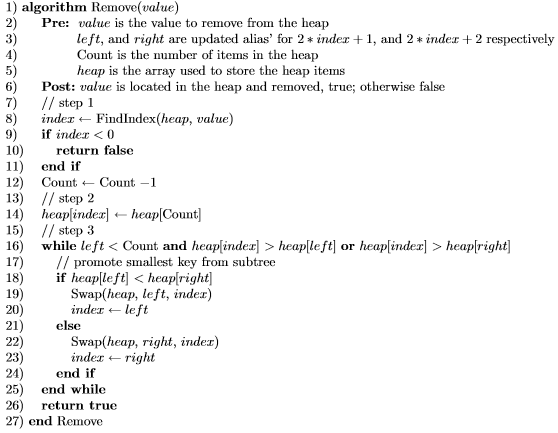
Designing an algorithm for heap insertion is simple, but we must ensure that heap order is preserved after each insertion. Generally this is a post-insertion operation. Inserting a value into the next free slot in an array is simple: we just need to keep track of the next free index in the array as a counter, and increment it after each insertion. Inserting our value into the heap is the ﬁrst part of the algorithm; the second is validating heap order. In the case of min-heap ordering this requires us to swap the values of a parent and its child if the value of the child is < the value of its parent. We must do this for each subtree containing the value we just inserted.

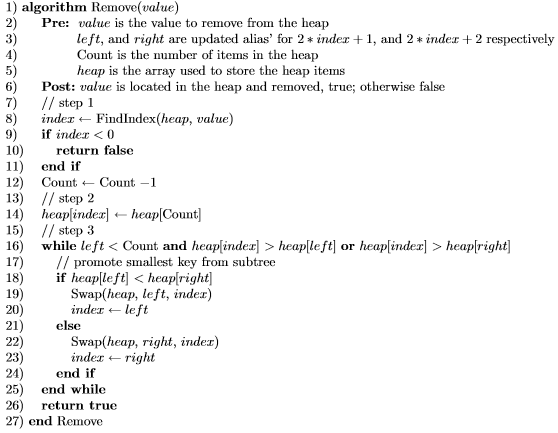


The design of the MaxHeapify algorithm is very similar to that of the MinHeapify algorithm, the only diﬀerence is that the < operator in the second condition of entering the while loop is changed to >.

Deletion

Just as for insertion, deleting an item involves ensuring that heap ordering is preserved. The algorithm for deletion has three steps: (1.) Find the index of the value to delete   
(2.) Put the last value in the heap at the index location of the item to delete  
(3.) Verify heap ordering for each subtree which used to include the value



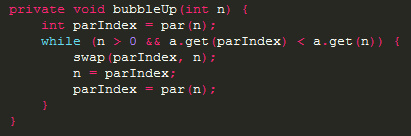


1. **Report proper**

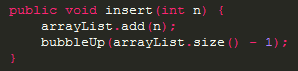
Most heap implementations require a backing array which is dynamically sized. However, using this would need to detect when it's full, create a new one, copy elements across, and to downsize when it has been emptied a lot. This is error-prone logic and deferring to an array list for it saves time and testing. In this program, an array list is used to implement heaps.

The next method we will implement is insert. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely violate the heap structure property. However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent. If the newly added item is less than its parent, the item will then be swapped with its parent.

When an item is percolated up, the heap property is restored between the newly added item and the parent. The heap property is preserved for any siblings. If the newly added item is very small, it needs to be swapped up to another level. In fact, it needs to keep swapping until we get to the top of the tree. Figure 1 shows the bubbleUp method, which percolates a new item as far up in the tree as it needs to go to maintain the heap property.

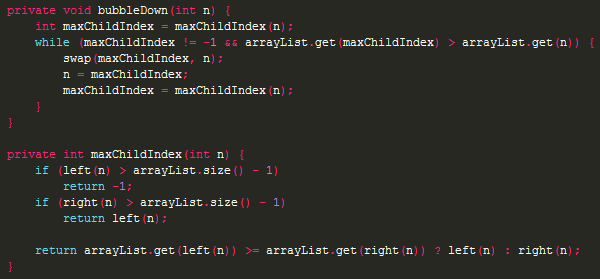
  
**Figure 1.** Code snippet of the bubbleUp() method

The insert method is presented in Figure 2. Most of the work in the insert method is really done by bubbleUp. Once a new item is appended to the tree, bubbleUp takes over and positions the new item properly.

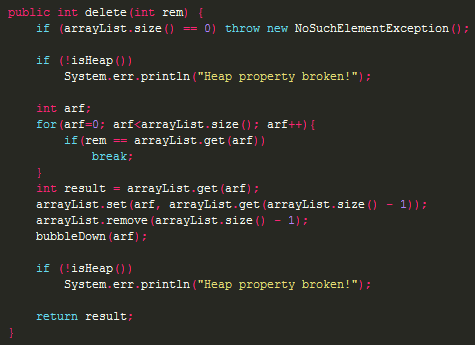
 **Figure 2.** Code snippet of the insert() method

With the insert method properly defined, we can now look at the delete method. To delete a particular node in the tree, the tree must be traversed first to find the index of the node to be removed. Then, restoring full compliance with the heap structure and heap order properties after the node has been removed is done. We can restore our heap in two steps. First, we will restore the root item by taking the last item in the list and moving it to the root position. Moving the last item maintains our heap structure property. However, we have probably destroyed the heap order property of our binary heap. Second, we will restore the heap order property by pushing the new root node down the tree to its proper position.

In order to maintain the heap order property, all we need to do is swap the root with its smallest or largest child less than the root. After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children. The code for percolating a node down the tree is found in the bubbleDown and maxChildIndex (minChildIndex for min heap) methods in Figure 3.

  
**Figure 3.** Code snippet of bubbleDown() and maxChildIndex() methods

The code for the delete operation is in Figure 4. Note that once again the hard work is handled by a helper function, in this case bubbleDown.

 **Figure 4.** Code snippet of delete() method

1. **Result and Discussion**

The heap abides progressively to a strategy during the invocation of the insertion and, deletion algorithms. The cost of such a policy is that upon each insertion and deletion, we invoke algorithms that have logarithmic run time complexities. While the cost of maintaining the strategy might not seem overly expensive, it does still come at a price. By using an array list to implement a heap, we have minimized and avoided the impact of dynamic array resizing.

1. **Conclusion**

Heaps are important because they allow you to search and select the element with the highest priority at any given time. They allow you to search for the element with the highest priority in O(1) time. But, searching for any other element in the heap is O(n) because the nodes across the heap aren’t ordered like they would be in a binary search tree. In addition, heaps have good deletion time. For instance, if you want to remove the value at the top of a heap, it will take O(logn) time. However, heaps are inherently unsuited for removing duplicate keys, and are inherently suited for storing data where duplicate keys exist.

 A heap is a data structure that maintains data semi ordered. Thus, it is a good tradeoff between the costs of maintaining a complete order and the cost of searching through random chaos. This attribute is used on many algorithms, such as selection, ordering, or classification.

1. **References**

[1] Adamchik, V. (2009). *Binary Heaps.* Retrieved from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html

[2] Allain, A. *What is a Heap*. Retrieved from

<http://www.cprogramming.com/tutorial/computersciencetheory/heap.html>

[3] Barnet, G. (2008). *Data Structures and Algorithms*. Retrieved from http://dotnetslackers.com/

[4] Hughes, J. *The Heap*. Retrieved from http://www.biostat.umn.edu/~johnh/book/06.pdf

[5] Humphreys, J. (2015, April). *Binary Heap Based on Array List.*

Retrieved from http://www.codescream.com/ContentDisplay?targetContent=HeapArrayList

[6] Miller, B. *Binary Heap Implementation*. Retrieved from

http://interactivepython.org/courselib/static/pythonds/Trees/BinaryHeapImplementation.html