Min-Heap

When it comes to the implementation of binary heap, there are different ways to approach it. There are max-heaps and min-heaps and I have chosen to implement min-heap in java. In a min-heap the value of the node i is greater than or equal to the value of the parent node. With this in mind storing our heap in an array and sorting the array in non-decreasing order is how we will be approaching this implementation. I will also be discussing the design choices I decided to make and why they were chosen.

When setting up our min-heap as an array we can find the parent, left child, and right child for a given index. However, to begin we first need to create our instance variables for our empty heap list, where we will be storing our final min-heap array. We will also need a variable for size and capacity to ensure our process is as efficient as possible. As mentioned, before we would need to include methods to support finding the parent, left child, and right child for a given index, which ofc would not work for the root as it has no parent. We also want to include an insert() method which would be used to add a new value to our min-heap. Aswell as a extract\_min() method to extract the minimum value of our heap or in other words our ‘root’. To incorporate both, insert() and extract\_min() we will also need to introduce two more methods that will allow our heap to remain correct as we make changes throughout our code. We will also need to incorporate a swap() method to make sure our nodes are placed correctly when inserting a new value at the end of our array. Similarly, we will also need a method, which we will call ‘heapify’, which ensures our heap remains a valid min-heap when we call our extract\_min() method. Finally, we would want to display our final array which will be done through our display() method.

Creating our parent, left child and right child methods were simple to create all it required was a simple equation for each. Since, our min-heap will be stored in an array we would access elements through their index. To determine the parent of the node we would have our index value minus 1 and then divided by 2 ((i -1) / 2)). For our left child, we would do our index times 2 and then add 1 (2 \* i + 1). For our right child, we would do our index times 2 and then add 2 (2 \* i + 2). Moving onto our insert() method it was a bit more challenging but understanding how it worked before coding it was extremely crucial and helpful. Through this method we would insert a value to the end of our array and increment the size of our array. We will also want to iterate through the array by calling upon our swap method which reorders our array to maintain the min-heap and checks if our new inserted value is not smaller than its parent and if it is then it continues to swap places and update its current index with that of its parent. However, when looking at our extract\_min() and heapify() methods this is where I faced some challenges.

At first, I wasn’t sure exactly how to approach this concept in code, but I knew what I had to do in pseudo code. For our extract\_min() method, we would need to first find the minimum value which would be at index 0 of our heap and remove this item and decrement the size of our heap by 1 because we are removing an element and then we would have to check each node to ensure it still holds our min-heap property. This is when I decided to do some research and found it was easiest to move the last element of the heap to the root position at index 0 which essentially removes the root element. After doing so this is when I also found that we would need to incorporate our heapify() method within the extract\_min() method to ensure we reorganize our heap to ensure it remains a min-heap. This was perhaps the most difficult part because I had to find the different things to compare to ensure our min-heap remained valid. I once again had to resort to further research and making sure the step I was taking were correct. Once we remove the minimum element we would essentially need to do the same thing that we did with insert() and swap() method in which we would have to iterate through our array to assign the correct left and right child to the new root and so on. When looking at examples I found that the best way to achieve this would have to be through creating a variable called smallest that is called up on a given index and is compared to its left and right child. If the value of smallest is smaller than the left and right child it remains the same, however if it is larger than it is swapped with the smallest child, and we continue this recursively. The implementation was rocky at first but eventually I was able to get it to work and ensure that the sub-tree rooted at the smallest child satisfied the heap property.

In conclusion, the implementation of a min-heap in java was mostly straight forward when implementing the simpler methods, although when I ran into problems, I did need to do some extra research on how to solve them. When it came to the implementation what really helped me was visualizing it before hand and thinking about all the methods and variables I would need. Not only this, but also testing each method once created to make sure it was doing what I needed to and would test it with usually 3 numbers at a time. Overall, this assignment allowed me to explore topics much deeper such as priority queue, heap data structure, and my coding skills.