Boxiao Li 30069613
Chunyu Li 30056553
Sijia Yin 30049836

CPSC331 Assignment3

Date: June 17, 2019.

Group: Boxiao Li 30069613 \ Chunyu Li 30056553 \ Sijia Yin 30049836

1. See the Java file. The code provided here is copy from the Java file, which is handed in the D2L.

```
// Boxiao Li 30069613;
// Sijia Yin 30049836;
// Chunyu Li 30056553;

package cpsc331.assignment3;

import cpsc331.assignment3.Array;
import java.awt.image.*;

/*
*    Requirements (from the CPSC 331 – Assignment #3 .pdf):
*         The number of steps used to sort an Array with length n must be in
*   O(n log n) in the worst case – not just “most of the time.”
*         The algorithm must sort its input array in place, using O(log n)
*    additional storage – including the size of a call stack whenever a
*    recursive method is being used.
*/
public class ArrayUtils<T extends Comparable<T>>
{
     //-------------------------------------------------------------------------

          /*
          *    Precondition:
          *         A is an input array with length n>=1, storing elements from
          *         some ordered type T.
          *    Postcondition:
          *         The array A will not be changed.
          *         A new ordered array  with the same length as A, every time
recusion the max will be returned as
          *         output.
          *              a) the new ordered array is a reordering of A
          *              b) the new ordered array is sorted in an increasing order
          */

     public void sort(Array<T> A)
     {
          int i = A.length() - 1;//let i = A.length-1
          buildHeap(A);//call bulidHeap, which will return a binary max heap A
          while (i > 0) {
               T largest = deleteMax(A, i);//call deleteMax, and get the returned max
value into end of the array
               A.set(i, largest);
               i--;
          }
     }
          /*
          *    Bound Function
          *         f(i) = i
          *
          *    Loop Invariant
```

```
          *         a) A is an input array with length n>=1, storing elements
          *              from some ordered type T.
          *         b) The entries of A have been reordered but are otherwise
          *              unchanged
          *         c) i is an integer variable such that 0<=i<=A.length-1
          *         d) A is a binary max heap with size i+1
          *         e) if i<= A.length-2, then A[h]<=A[i+1] for every integer
          *               h such that 0<=h<=i
          *         f) A[h]<=A[h+1] for every integer h such that i+1<=h <=
          *              A.length-2
          */
     //-------------------------------------------------------------------------
          /*
          *    Precondition:
          *         A is an input array with length n>=1, storing elements from
          *         some ordered type T.
          *         A is the binary max heap.(index 0 is the largest number)
          *    Postcondition:
          *         The max number in the binary max heap will be deleted.
          */
     public T deleteMax(Array<T> A, int i)
     {

          T max = A.get(0);//get the binary max heap index 0, and store into T
          A.set(0, A.get(i));//let value of index 0 be the value of i
          bubbleDown(A, 0, i);//call bubbleDown
          return max;//return the max
     }
     //-------------------------------------------------------------------------
          /*
          *    Precondition:
          *         A is an input array with length n>=1, storing elements from
          *         some ordered type T.
          *    Postcondition:
          *         A will be a binary max heap.
          */
     public void buildHeap(Array<T> A)
     {
          if(A.length()>0){
               int i = ((A.length()-1) / 2);//i is the last internal node's index
          while( i >= 0){
               bubbleDown(A, i, A.length()-1);//call bubleDown with array A from
index i to A.length-1
               --i;
          }
          }
     }
          /*
          *    Bound Function
          *         f(i) = i
          *
          *    Loop Invariant
          *         a) i is an integer variable such that 0<=i<=A.length()-1)/2
          *         b) for all the integer k, i+1<= k<= A.length(), A still has
          *              the binary max heap order
          */
     //-------------------------------------------------------------------------
```

```
          /*
          *    Precondition:
          *         A is an input array with length n>=1, storing elements from
          *         some ordered type T.
          *    Postcondition:
          *         A will be a binary max heap.
          */
     // the number is from above is i.
     public void bubbleDown(Array<T> A, int n, int i)
     {//array A, i from buildHeap and A.length-1 are as input

          while (n * 2 + 1 <= i) {
               int toSwap = n * 2 + 1;//left(i)
               if (toSwap+1<=i && A.get(toSwap+1).compareTo(A.get(toSwap))>0)
               {//if there is a right child and right child is biger than left child.
                    ++toSwap;//right child
               }
               if (A.get(n).compareTo(A.get(toSwap)) > 0) {//n is biger than right
child
                    break;
               }
               T temp = A.get(n);//let temp be the n
               A.set(n, A.get(toSwap));//let value of n be the value of toSwap
               A.set(toSwap, temp);//let value of toSwap be the value of temp
               n = toSwap;
          }
     }
          /*
          *    Bound Function
          *         f(n, i) = i - (n*2 + 1)
          *
          *    Loop Invariant
          *         a) n is an integer variable such that 0<= n <= i
          *         b) toSwap is an integer variable such that 0<=toSwap<=i
          */

}
```

2. Our algorithm is a deleteMax algorithm operate with bubbleDown method.
The reason we do this is for the efficiency of this algorithm. The deleteMax method(in our code, it is called “findMax”) will execute O(n) times in the worst case. The bubbleDown method will be called at the end of each execution of the deleteMax method because the bubbleDown method will cost O(log n) times. Thus the total steps cost in the worst case is O(n log n) times. In large value case, O(n log n) is faster than the $O(n^2)$, thus we chose a deleteMax algorithm with bubbleDown methods in this assignment.