```
package cpsc331.assignment3;

import cpsc331.assignment3.Array;

/**
 *  Names: Lan Jin, Juwei Wang, Junrui Fang
 *  //UCID: 30078919, 30053278, 30041389
 * Provides a method to sort the given array.
 * <br />
 *
 * <p>
 *  <strong>Array Invariant:</strong>
 * </p>
 * <ol style="list-style-type: lower-alpha">
 * <li> this.LENGTH is an integer  </li>
 * <li> Heapsize is an integer  </li>
 * <li> i is an integer to check if the array sorted completely.
 * <li>
 * </ol>
 *
 */
public class ArrayUtils<T extends Comparable<T>> {


    public void sort(Array<T> A){

        int heapSize = 1;  // heap-size(A) = 1
        int i = 1; // integer i = 1

        while (i < A.length()){                    // while(i < A.length)
            //insert(...)
            insert(A.get(i),A,heapSize);          // insert (A[i])
            heapSize = heapSize + 1;              //update heapSize after insertion
            i = i + 1;                            // i ++
        }
        i = A.length() - 1;                       // i = A.length - 1

        while (i > 0){                            // while(i > 0)
            T largest = deleteMax(A, heapSize);   // int largest = DeleteMax()
            //System.out.println(largest);
            heapSize = heapSize - 1;              //update heapSize after deletion
            A.set(i,largest);                     // A[i] = largest
            i = i - 1;                            // i = i - 1
        }

    }

    /**
     * A method  to insert the elements to the heap:
     * @param key   The elements will be inserted
     * @param A     The elemets will be inserted in this array
     * @param heapSize  the sizes of the heap ( will change every time when the elements has been
inserted successfully)
     */
    public void insert(T key, Array<T> A, int heapSize){
        int x = heapSize;               //int x = heap-size(A)
        A.set(x,key);                   // A[x] = key
        bubbleup(x,A);                  // bubbleup(x)
    }

    /**
     * After the heap has been added, this method can ensure it is a completed tree and it is a max-heap:
     * @param x     The positions of elements in the heap
     * @param A     The elemets will be inserted in this array
     */
    public void bubbleup(int x, Array<T> A){
        if (!isroot(x) && (A.get(x).compareTo(A.get(parent(x))) > 0)){
            T temp = A.get(x);          // T tmp = value(x)
            A.set(x,A.get(parent(x)));  // set the value of x to be value(parent(x))
            A.set(parent(x), temp);     // set the value of parent(x) to be tmp
            bubbleup(parent(x),A);      // bubbleup(parent(x))
        }
    }

    /**
     * A method to check weather the leaf is a root. :
     * @param x     The positions of elements in the heap
     */
    public boolean isroot(int x){
        return x == 0; // index of root is 0, when heap is represented using an array.
    }


    /**
     * To get the parental position of the leaf :
     * @param x     The positions of elements in the heap
     */
    public int parent (int x){
        return (x-1)/2;  // index of parent of node x is (x-1)/2, when heap is represented using an array.
    }


    /**
     * A method  to delete the top element( The maximum number) in the heap:
     * @param A     The elemets will be inserted in this array
     * @param heapSize  the sizes of the heap ( will change every time when the elements has been deleted 
successfully)
     */
    public T deleteMax(Array<T> A, int heapSize){
        T v = A.get(heapSize-1);

        if ((heapSize-1) == 0){
            return v;
        } else{
            //System.out.println(A.get(0));
            T key = A.get(0);
            A.set(0,v);
            // heapsize is not a global data, we do not modified it in this method,
            //but bubbleDown needs a updated one, so we pass heapSize -1 as argument instead.
            bubbleDown(0, A, heapSize - 1);
            return key;
        }
    }



    /**
     * A method to ensure the heap is a completed tree and max-heap in deletion processes:
     * @param x     the position ( node) of the heap
     * @param A     The elemets will be inserted in this array
     * @param heapSize  the sizes of the heap ( will change every time when the elements has been deleted 
successfully)
     */
    public void bubbleDown(int x, Array<T> A, int heapSize){
        if (hasRight(x, heapSize)){  //node x has two children
            if (A.get(left(x)).compareTo(A.get(right(x))) >= 0){
                if (A.get(left(x)).compareTo(A.get(x)) > 0){
                    T t = A.get(left(x));            //T tep = value(left(x))
                    A.set(left(x), A.get(x));        //set the value of left(x) to be value(x)
                    A.set(x, t);                     // set the value of x to be tmp
                    bubbleDown(left(x), A, heapSize);  // bubbledown(left(x))
                }
            } else if (A.get(right(x)).compareTo(A.get(x)) > 0){
                    T t = A.get(right(x));           //T tep = value(right(x))
                    A.set(right(x), A.get(x));       //set the value of right(x) to be value(x)
                    A.set(x, t);                     // set the value of x to be tmp
                    bubbleDown(right(x), A, heapSize);  // bubbledown(right(x))
              }

        } else if (hasLeft(x, heapSize)){ //node has one child.
            if(A.get(left(x)).compareTo(A.get(x)) > 0){
                T t = A.get(left(x));                    //T tep = value(right(x))
                A.set(left(x), A.get(x));                //set the value of right(x) to be value(x)
                A.set(x, t);                             // set the value of x to be tmp
                bubbleDown(left(x), A, heapSize);        // bubbledown(right(x))
            }
        }
    }

    /**
     * To check weather the leaf has left child :
     * @param x     The positions of elements in the heap
     * @param heapSize      The sizes of the heap
     */
    public boolean hasLeft(int x, int  heapSize){
        return (2*x + 1) < heapSize;
    }

    /**
     * To check weather the leaf has right child :
     * @param x     The positions of elements in the heap
     * @param heapSize      The sizes of the heap
     */
    public boolean hasRight(int x, int heapSize){
        return (2*x + 2) < heapSize;
    }

    /**
     * A method to get the left child node. :
     * @param x     The positions of elements in the heap
     */
    public int left(int x){
        return (2*x + 1);         // index of left child of node x is (2*x + 1), when heap is represented
using an array.
    }

    /**
     * A method to get the right child node. :
     * @param x     The positions of elements in the heap
     */
    public int right(int x){
        return (2*x + 2);    // index of right child of node x is (2*x + 2), when heap is represented
using an array.
    }


}
```