```java
/**
 * Implementation of MaxHeap<E>
 * @author ameliado
 */


public class MaxHeap<E> {
    E[] items;  // primitive array of items
    int size;   // size of heap

    /**
     * Creates a max heap from the given items
     * @param theItems - the given items
     */
    public MaxHeap(E[] theItems)
    {
        items = theItems;
        size = theItems.length;

        this.makeHeap();
    }

    /**
     * Pushes the item at the given index down the items array
     * @param i - the index where the item is
     */
    private void pushDown( int i )
    {
        while ( !isLeaf(i) ) {
            int leftIndex = left(i);
            int rightIndex = right(i);
            int largest = i;

            if (///leftIndex < size &&
                SortUtils.compare(items[largest],
  items[leftIndex]) < 0) {
                largest = leftIndex;
            }

```

```
39              if (rightIndex < size &&
40                  SortUtils.compare(items[largest],
  items[rightIndex]) < 0) {
41                  largest = rightIndex;
42              }
43
44              if (largest != i) {
45                  SortUtils.swap(items, i, largest);
46                  i = largest;
47              }
48              else {
49                  break;
50              }
51          }
52      }
53
54      /**
55       * Rearranges the array items so that it represents a max
  heap
56       */
57      private void makeHeap()
58      {
59          // i = 3 -> 0
60          // i = 4, 5 -> 1
61          // i = 5, 6 -> 2
62          // i = 7, 8 -> 3
63          // i / 2 - 1
64          for (int i = (size / 2) - 1; i >= 0; i--) {
65              pushDown(i);
66          }
67      }
68
69      /**
70       * Returns and removes the max value in the heap
71       * @return the max value in the heap
72       */
73      public E pop()
74      {
75          E maxVal = items[0];
```

```
 76
 77          items[0] = items[size - 1];
 78          size--;
 79
 80          pushDown(0);
 81
 82          return maxVal;
 83      }
 84
 85      /**
 86       * Determines if the item at the given index is a leaf
 87       * @param i - given index to check if the item is a leaf
 88       */
 89      private boolean isLeaf( int i )
 90      {
 91          return this.left(i) >= size;
 92
 93      }
 94
 95      /**
 96       * Returns the left child index
 97       * @param i - index of the item to find left child of
 98       * @return index of the left child of item at index i
 99       */
100      private int left( int i )
101      {
102          return 2 * i + 1;
103      }
104
105      /**
106       * Returns the right child index
107       * @param i - index of the item to find right child of
108       * @return index of the right child of item at index i
109       */
110      private int right( int i )
111      {
112          return 2 * i + 2;
113      }
114
```

```
115     public String toString()
116     {
117         return SortUtils.toString(items, size);
118     }
119
120 }
121
```