```java
/**
   *  Program 3
   *  This is a Prioity Queue Structure using a Binary Min Heap Implementation.
   *  All elements start at index 1 for improved reability and math operations.
   *  CS310-01
   *  4/10/2019
   *  @author  Karl Parks cssc1506
   */

package data_structures;

import java.util.ConcurrentModificationException;
import java.util.Iterator;

public class BinaryHeapPriorityQueue<E extends Comparable<E>>
implements PriorityQueue<E> {
  private int sequenceNumber, size, modCount, capacity;
  private Wrapper<E>[] binaryHeap;

  @SuppressWarnings("hiding")
  protected class Wrapper<E> implements Comparable<Wrapper<E>> {
    int num; //each element will have a priority assigned, this enables stability
    E data; //object data variable

    public Wrapper (E d){
      num = sequenceNumber++; //sequenceNumber will always increase
      data = d;
    }

    @SuppressWarnings("unchecked")
    public int compareTo(Wrapper<E> o){
      if(((Comparable<E>)data).compareTo(o.data) == 0) {
        //System.out.println("They are equal so compares priority next");
        return (int) (num - o.num);
      }
      //System.out.println("They are not equal... but which is bigger?");
      return ((Comparable<E>)data).compareTo(o.data);
    }
  }
```

```
39
40    public BinaryHeapPriorityQueue() {
41      this(DEFAULT_MAX_CAPACITY);
42    }
43
44    @SuppressWarnings("unchecked")
45    public BinaryHeapPriorityQueue(int maxSize) {
46      this.binaryHeap = new Wrapper[maxSize+1]; //maxSize + 1 is
      •   required for starting at index 1
47      capacity = maxSize;
48    }
49
50    public void trickleUp(int n) {
51      //System.out.println("trickleUp");
52      //parent is n/2 (n >> 1) of binary heap... current is n,
53        //initially, n = size... which is good to start... but we need
      •   to do this recursively to trickle up the entire tree
54      if ((binaryHeap[(n >> 1)]).compareTo((binaryHeap[n])) > 0) {
55        //System.out.println("Parent: " + binaryHeap[n/2].data + " > "
      •   + binaryHeap[n].data + " (the Child)");
56        swap(n); //simply swap function
57        if (n > 3) { //checks if already checked root node
58          trickleUp((n >> 1)); //recursive call
59        }
60      }
61    }
62
63    public void trickleDown(int index) {
64      //used riggins text
65      //System.out.println("trickleDown");
66      int current = index; //my heap starts at 1 instead of 0
67      int child = getNextChild(current);
68      while (child != -1 &&
      •   (binaryHeap[current].compareTo(binaryHeap[child]) < 0) &&
      •   (binaryHeap[child].compareTo(binaryHeap[size]) < 0)) {
69        //System.out.println("Child Index: binaryHeap[" + child + "]
      •   Child Value: " + binaryHeap[child].data);
70        binaryHeap[current] = binaryHeap[child];
71        current = child;
72        child = getNextChild(current);
73      }
74      binaryHeap[current] = binaryHeap[size];
75
76      //trickleDown explanation in pseudocode:
```

```
    //trickleDown explanation in pseudocode:
    //first index now equals data of last index
    //from "parent", look at left child, right child, determine which is smallest (min-heap)
    //check if smaller child is less than parent
    //if true -> swap smaller child and parent
    //do it again
    //stop when no more children exist or children are larger than parent
    //some conditions to avoid are no child, single child, double child cases
  }

  public int getNextChild(int current) {
    //used riggins text
    //this method checks which child is smaller, or if there even is a child
    int left = (current << 1); //array starts at 1 instead of 0 for easy math
    int right = left+1;
    if (right < size) { //checks for two children
      if(binaryHeap[left].compareTo(binaryHeap[right]) < 0 ) {
        return left;
      }
      return right;
    }
    if (left < size) {
      return left;
    }
    return -1; //no children
  }

  public void swap(int i) {
    //simple swap method
    Wrapper<E> tmp = binaryHeap[(i >> 1)];
    binaryHeap[(i >> 1)] = binaryHeap[i];
    binaryHeap[i] = tmp;
  }

  @Override
  public boolean insert(E object) {
    if (isFull()) { //quick check if array is full
      return false;
    }
```

```
115       Wrapper<E> newObj = new Wrapper<E>(object);
116       binaryHeap[size+1] = newObj; //add new wrapper object to array
117       size++; //increse size
118       if (size > 1) {
119         trickleUp(size); //trickleUp and reheapify
120       }
121       modCount++; //we modified something so we increase the
          modification number
122       return true;
123     }
124
125     @Override
126     public E remove() {
127       //simple to insert
128       if (isEmpty()) { //quick check if array is empty
129         return null;
130       }
131       E tmp = binaryHeap[1].data;
132       trickleDown(1);
133       size--;
134       modCount++;
135       return tmp;
136     }
137
138     @Override
139     public boolean delete(E obj) {
140       boolean found = false;
141       //search array
142       for (int i = 1; i <= size; i++) { //repeat if necessary
143         if (obj.compareTo(binaryHeap[i].data) == 0) {     //if found,
          remove and trickle down from that index
144           //System.out.println("Found Something");
145           trickleDown(i);
146           size--;
147           modCount++;
148           found = true; //return true if something was found
149           delete(obj); //recursive call, necessary because of changing
              size after trickleDown
150         }
151       }
152       //return true if something was found
153       return found;
154     }
155
```

```
156     @Override
157     public E peek() {
158       if (isEmpty()) { //quick check if array is empty
159         return null;
160       }
161       return binaryHeap[1].data;
162     }
163
164     @Override
165     public boolean contains(E obj) {
166       boolean found = false;
167       //loops through array
168       for (int i = 1; i <= size; i++) { //repeat if necessary
169         if (obj.compareTo(binaryHeap[i].data) == 0) {
170           //System.out.println("Found Something");
171           found = true; //return true if something was found
172         }
173       }
174       return found;
175     }
176
177     @Override
178     public int size() {
179       return size;
180     }
181
182     @Override
183     public void clear() {
184       modCount++; //clearing is a modification
185       size = 0; //sets size to 0 but doesn't change anything else. This
  •       affects peek.
186     }
187
188     @Override
189     public boolean isEmpty() {
190       return size == 0;
191     }
192
193     @Override
194     public boolean isFull() {
195       return (size >= capacity);
196     }
197
198     public void debugger() {
```

```
199       //my debugger method for printing things without the iterator
200       System.out.println("\n--- Debugger Method ---");
201       System.out.println("size:           " + size);
202       System.out.println("sequenceNumber: " + sequenceNumber);
203       for (int i = 1; i <= size; i++) {
204         System.out.println("idx: " + i + "\tnum: " + binaryHeap[i].num
          + "\tdata: " + binaryHeap[i].data);
205       }
206       System.out.println("");
207     }
208
209     public void printSort() {
210       //if you would like a sorted array
211       //this method uses remove in a normal for-loop with the original
          size
212       Wrapper<E>[] duplicate = binaryHeap.clone();
213       int startSize = size;
214       System.out.print("\nSorted: \t");
215       for (int i = 0; i < startSize; i++) {
216         System.out.print(remove() + " ");
217       }
218       size = startSize;
219       binaryHeap = duplicate;
220     }
221
222     @Override
223     //this iterator only returns an iterator of the objects in the PQ,
          in no particular order
224     //if you wanted a sorted array printed, use the above printSort
          method
225     public Iterator<E> iterator() { //iterator helper method for
          enhanced for-loop
226       return new IteratorHelper();
227     }
228
229     private class IteratorHelper implements Iterator<E> {
230       private int count, expectedMod;
231
232       public IteratorHelper() {
233         expectedMod = modCount; //checks for modifications
234         count = 0;
235       }
236
237       public boolean hasNext() {
```

```
    public boolean hasNext() {
      return count != size; //checks if at end of list
    }

    public E next() {
      if (modCount != expectedMod) { //modification error throw here
            throw new ConcurrentModificationException("Cannot modify
            list during enhanced for-loop");
      }
      count++;
      return binaryHeap[count].data;
    }

    public void remove() {
```