```
 1 import java.util.Arrays;
11
12 /**
13  * Program to test {@code siftDown} on int array.
14  *
15  * @mathdefinitions <pre>
16  * SUBTREE_IS_HEAP (
17  *   a: string of integer,
18  *   start: integer,
19  *   stop: integer,
20  *   r: binary relation on T
21  *  ) : boolean is
22  *  [the subtree of a (when a is interpreted as a complete binary tree) rooted
23  *   at index start and only through entry stop of a satisfies the heap
24  *   ordering property according to the relation r]
25  *
26  * SUBTREE_ARRAY_ENTRIES (
27  *   a: string of integer,
28  *   start: integer,
29  *   stop: integer
30  *  ) : finite multiset of T is
31  *  [the multiset of entries in a that belong to the subtree of a
32  *   (when a is interpreted as a complete binary tree) rooted at
33  *   index start and only through entry stop]
34  * </pre>
35  *
36  * @author Put your name here
37  *
38  */
39 public final class ArraySiftDownMain {
40
41     /**
42      * Private constructor so this utility class cannot be instantiated.
43      */
44     private ArraySiftDownMain() {
45     }
46
47     /**
48      * Number of junk entries at the end of the array.
49      */
50     private static final int JUNK_SIZE = 5;
51
52     /**
53      * Checks if the subtree of the given {@code array} rooted at the given
54      * {@code top} is a heap.
55      *
56      * @param array
57      *            the complete binary tree
58      * @param top
59      *            the index of the root of the "subtree"
60      * @param last
61      *            the index of the last entry in the heap
62      * @return true if the subtree of the given {@code array} rooted at the
63      *         given {@code top} is a heap; false otherwise
64      * @requires <pre>
65      * 0 <= top  and  last < |array.entries|  and
66      * [subtree rooted at {@code top} is a complete binary tree]
67      * </pre>
68      * @ensures isHeap = SUBTREE_IS_HEAP(heap, top, last, <=)
```

```
 69      */
 70     private static boolean isHeap(int[] array, int top, int last) {
 71         assert array != null : "Violation of: array is not null";
 72         assert 0 <= top : "Violation of: 0 <= top";
 73         assert last < array.length : "Violation of: last < |array|";
 74         /*
 75          * No need to check the other requires clause, because it must be true
 76          * when using the Array representation for a complete binary tree.
 77          */
 78         int left = 2 * top + 1;
 79         boolean isHeap = true;
 80         if (left <= last) { // there is non-empty left subtree
 81             isHeap = (array[top] <= array[left]) && isHeap(array, left, last);
 82             int right = left + 1;
 83             if (isHeap && (right <= last)) { // there is non-empty right subtree
 84                 isHeap = (array[top] <= array[right])
 85                         && isHeap(array, right, last);
 86             }
 87         }
 88         return isHeap;
 89     }
 90
 91     /**
 92      * Finds {@code item} in DOMAIN({@code m}) and, if such exists, adds 1 to
 93      * the value in {@code m} associated with key {@code item}; otherwise places
 94      * new key {@code item} in {@code m} with associated value 1.
 95      *
 96      * @param <K>
 97      *            the type of the map's key
 98      * @param item
 99      *            the item whose count is to be incremented
100      * @param m
101      *            the {@code Map} to be updated
102      * @aliases reference item
103      * @updates m
104      * @ensures <pre>
105      * if item is in DOMAIN(m) then
106      *   there exists count: integer ((item, count) is in #m
107      *     and m = (#m \ (item, count)) union {(item, count + 1)})
108      * else
109      *   m = #m union {(item, 1)}
110      * </pre>
111      */
112     private static <K> void incrementCountFor(K item, Map<K, Integer> m) {
113         assert item != null : "Violation of: item is not null";
114         assert m != null : "Violation of: m is not null";
115
116         if (m.hasKey(item)) {
117             Map.Pair<K, Integer> pair = m.remove(item);
118             m.add(pair.key(), pair.value() + 1);
119         } else {
120             m.add(item, 1);
121         }
122
123     }
124
125     /**
126      * Exchanges entries at indices {@code i} and {@code j} of {@code array}.
127      *
```

```
128      * @param array
129      *            the array whose entries are to be exchanged
130      * @param i
131      *            one index
132      * @param j
133      *            the other index
134      * @updates array
135      * @requires 0 <= i < |array| and 0 <= j < |array|
136      * @ensures array = [#array with entries at indices i and j exchanged]
137      */
138     private static void exchangeEntries(int[] array, int i, int j) {
139         assert array != null : "Violation of: array is not null";
140         assert 0 <= i : "Violation of: 0 <= i";
141         assert i < array.length : "Violation of: i < |array|";
142         assert 0 <= j : "Violation of: 0 <= j";
143         assert j < array.length : "Violation of: j < |array|";
144
145         if (i != j) {
146             int tmp = array[i];
147             array[i] = array[j];
148             array[j] = tmp;
149         }
150     }
151
152     /**
153      * Given an array that represents a complete binary tree and an index
154      * referring to the root of a subtree that would be a heap except for its
155      * root, sifts the root down to turn that whole subtree into a heap.
156      *
157      * @param array
158      *            the complete binary tree
159      * @param top
160      *            the index of the root of the "subtree"
161      * @param last
162      *            the index of the last entry in the heap
163      * @updates array
164      * @requires <pre>
165      * 0 <= top  and  last < |array.entries|  and
166      * SUBTREE_IS_HEAP(array, 2 * top + 1, last, <=)  and
167      * SUBTREE_IS_HEAP(array, 2 * top + 2, last, <=)
168      * [subtree rooted at {@code top} is a complete binary tree]
169      * </pre>
170      * @ensures <pre>
171      * SUBTREE_IS_HEAP(array, top, last, <=)  and
172      * perms(array, #array)  and
173      * SUBTREE_ARRAY_ENTRIES(array, top, last) =
174      *  SUBTREE_ARRAY_ENTRIES(#array, top, last)  and
175      * [the other entries in array are the same as in #array]
176      * </pre>
177      */
178     private static void siftDown(int[] array, int top, int last) {
179         assert array != null : "Violation of: array is not null";
180         assert 0 <= top : "Violation of: 0 <= top";
181         assert last < array.length : "Violation of: last < |array|";
182         assert isHeap(array, 2 * top + 1, last) : ""
183                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 1, last, <=)";
184         assert isHeap(array, 2 * top + 2, last) : ""
185                 + "Violation of: SUBTREE_IS_HEAP(array, 2 * top + 2, last, <=)";
186         /*
```

```
187          * No need to check the other requires clause, because it must be true
188          * when using the array representation for a complete binary tree.
189          */
190         int left = top * 2 + 1;
191         int right = top * 2 + 2;
192         int smaller = -1;
193
194         if (left <= last) {
195             if (right <= last) {
196                 if (array[left] > array[right]) {
197                     smaller = right;
198                 } else {
199                     smaller = left;
200                 }
201             } else {
202                 smaller = left;
203             }
204
205             if (array[top] > array[smaller]) {
206                 exchangeEntries(array, top, smaller);
207                 siftDown(array, smaller, last);
208             }
209         }
210
211     }
212
213     /**
214      * Main method.
215      *
216      * @param args
217      *            the command line arguments
218      */
219     public static void main(String[] args) {
220         SimpleReader in = new SimpleReader1L();
221         SimpleWriter out = new SimpleWriter1L();
222         /*
223          * Input array size from user
224          */
225         out.print("Enter (non-negative) heap size: ");
226         int heapSize = in.nextInteger();
227         /*
228          * Construct array as follows. Make its length be heapSize + JUNK_SIZE.
229          * The prefix of length heapSize of array represents a complete binary
230          * tree and contains pseudo-random integers in the range [JUNK_SIZE,
231          * heapSize + JUNK_SIZE). The suffix of length JUNK_SIZE is junk. These
232          * junk values are specifically set so as to count down from one less
233          * than JUNK_SIZE to 0.
234          *
235          * Also, build a Map<Integer, Integer> named original to represent the
236          * multiset of values present in the initial complete binary tree.
237          */
238         Map<Integer, Integer> original = new Map1L<>();
239         Random rnd = new Random1L();
240         int[] array = new int[heapSize + JUNK_SIZE];
241         for (int i = 0; i < heapSize; i++) {
242             int entry = JUNK_SIZE + ((int) (rnd.nextDouble() * heapSize));
243             array[i] = entry;
244             incrementCountFor(entry, original);
245         }
```

```
246         for (int i = heapSize; i < heapSize + JUNK_SIZE; i++) {
247             array[i] = heapSize + JUNK_SIZE - i - 1;
248         }
249         /*
250          * Output initial array
251          */
252         out.println("              initial array: " + Arrays.toString(array));
253         /*
254          * Heapify the heapSize-length prefix of array by repeatedly calling
255          * siftDown (this is an iterative implementation of heapify--it should
256          * start with i = heapSize / 2 - 1, but since it is intended to test
257          * siftDown, we call it on the leaves as well)
258          */
259         for (int i = heapSize - 1; i >= 0; i--) {
260             siftDown(array, i, heapSize - 1);
261         }
262         /*
263          * Make sure the heapSize-length prefix of array is now a heap
264          */
265         assert isHeap(array, 0, heapSize - 1) : ""
266                 + "Violation of: SUBTREE_IS_HEAP(array, 0, heapSize - 1, <=)";
267         /*
268          * Make sure the current multiset of values in the heapSize-length
269          * prefix of array is the same as the original
270          */
271         Map<Integer, Integer> current = original.newInstance();
272         for (int i = 0; i < heapSize; i++) {
273             incrementCountFor(array[i], current);
274         }
275         assert current.equals(original) : ""
276                 + "Method siftDown caused different values to be in the heap "
277                 + "than were in the original complete binary tree, "
278                 + "perhaps by failing to ignore the junk at "
279                 + "the far end of the array.";
280         /*
281          * Make sure the junk at the far end of array was not changed by
282          * siftDown
283          */
284         for (int i = heapSize; i < heapSize + JUNK_SIZE; i++) {
285             assert heapSize + JUNK_SIZE - i - 1 == array[i] : ""
286                     + "Method siftDown changed the junk at "
287                     + "the far end of the array: Expected "
288                     + (heapSize + JUNK_SIZE - i - 1) + " but was " + array[i];
289         }
290         /*
291          * If everything worked, output the array with a heapified prefix
292          */
293         out.println("array with heapified prefix: " + Arrays.toString(array));
294         /*
295          * Close streams
296          */
297         in.close();
298         out.close();
299     }
300
301 }
302
```