

**CS310-01**

**PROGRAMMING ASSIGNMENT 1**

**02.22.18**

**QUANG TRINH**

OrderedArrayPriorityQueue.java

1. /\*
2. \* Program #1
3. \* Quang Trinh
4. \* cssc0759
5. \*/
7. **package** data\_structures;
9. **import** java.util.Iterator;
10. **import** java.util.NoSuchElementException;
12. **public** **class** OrderedArrayPriorityQueue<E **extends** Comparable<E>> **implements** PriorityQueue<E> {
13. **private** E[] storage;
14. **private** **int** currentSize;
15. **private** **int** maxSize;
16. **private** **long** modCounter;
18. **public** OrderedArrayPriorityQueue(**int** size) {
19. maxSize = size;
20. storage = (E[]) **new** Comparable[maxSize];
21. currentSize = 0;
22. modCounter = 0;
23. }
25. **public** OrderedArrayPriorityQueue() {
26. **this**(DEFAULT\_MAX\_CAPACITY);
27. }
29. **public** **boolean** insert(E object) {
30. **if** (isFull())
31. **return** **false**;
32. **int** where = findInsertionPoint(object, 0, currentSize - 1);
33. **for** (**int** i = currentSize - 1; i >= where; i--) // Shift right to make room for new item.
34. storage[i + 1] = storage[i];
35. storage[where] = object;
36. currentSize++;
37. modCounter++;
38. **return** **true**;
39. }
41. **public** E remove() {
42. **if** (isEmpty())
43. **return** **null**;
44. modCounter++;
45. **return** storage[--currentSize];
46. }
48. **public** **boolean** delete(E obj) {
49. **if** (!contains(obj))
50. **return** **false**;
51. **int** first = findInsertionPoint(obj, 0, currentSize - 1);
52. **int** last = lastPrioritizedIndex(obj, 0, currentSize - 1);
53. **int** unit = (last + 1) - first;
55. **if** (last != currentSize - 1) {
56. **for** (**int** i = first; i < currentSize - unit; i++)
57. storage[i] = storage[i + unit];
58. currentSize = currentSize - unit;
59. **return** **true**;
60. } **else** {
61. currentSize = currentSize - unit;
62. **return** **true**;
63. }
64. }
66. **public** E peek() {
67. **if** (isEmpty())
68. **return** **null**;
69. **return** storage[currentSize - 1];
70. }
72. **public** **boolean** contains(E obj) {
73. **if** (isEmpty())
74. **return** **false**;
75. **if** (obj.compareTo(storage[currentSize - 1]) < 0) // Best case complexity O(1)
76. **return** **false**;
77. **if** (indexOf(obj) != -1)
78. **return** **true**;
79. **return** **false**;
80. }
82. **public** **int** size() {
83. **return** currentSize;
84. }
86. **public** **void** clear() {
87. currentSize = 0;
88. modCounter = 0;
89. }
91. **public** **boolean** isEmpty() {
92. **return** currentSize == 0;
93. }
95. **public** **boolean** isFull() {
96. **return** currentSize == maxSize;
97. }
99. **public** Iterator<E> iterator() {
100. **return** **new** IteratorHelper();
101. }
103. /\*
104. \* Return current position of an item, if not found return -1.
105. \*/
106. **private** **int** indexOf(E obj) {
107. **int** lo = 0;
108. **int** hi = currentSize - 1;
109. **while** (lo <= hi) {
110. **int** mid = (lo + hi) / 2;
111. **if** (obj.compareTo(storage[mid]) == 0)
112. **return** mid;
113. **else** **if** (obj.compareTo(storage[mid]) < 0)
114. lo = mid + 1;
115. **else**
116. hi = mid + 1;
117. }
118. **return** -1; // Not found
119. }
121. /\*
122. \* Return position needed to be inserted.
123. \*/
124. **private** **int** findInsertionPoint(E obj, **int** lo, **int** hi) {
125. **if** (hi < lo)
126. **return** lo;
127. **int** mid = (lo + hi) / 2;
128. **if** (obj.compareTo(storage[mid]) >= 0)
129. **return** findInsertionPoint(obj, lo, mid - 1);
130. **return** findInsertionPoint(obj, mid + 1, hi);
131. }
133. /\*
134. \* Return position of the last Prioritized item.
135. \*/
136. **private** **int** lastPrioritizedIndex(E obj, **int** lo, **int** hi) {
137. **if** (hi < lo)
138. **return** hi;
139. **int** mid = (lo + hi) / 2;
140. **if** (obj.compareTo(storage[mid]) > 0)
141. **return** lastPrioritizedIndex(obj, lo, mid - 1);
142. **return** lastPrioritizedIndex(obj, mid + 1, hi);
143. }
145. /////////////////////////////////////////////
146. **class** IteratorHelper **implements** Iterator<E> {
147. **int** iterIndex;
148. **long** stateCheck;
150. **public** IteratorHelper() {
151. iterIndex = 0;
152. stateCheck = modCounter;
153. }
155. **public** **boolean** hasNext() {
156. **if** (stateCheck != modCounter)
157. **return** **false**; // ConcurrentModificationException
158. **return** iterIndex < currentSize;
159. }
161. **public** E next() {
162. **if** (!hasNext())
163. **throw** **new** NoSuchElementException();
164. **return** storage[iterIndex++];
165. }
167. **public** **void** remove() {
168. **throw** **new** UnsupportedOperationException();
169. }
170. }
171. //////////////////////////////////////////////////
172. }

UnorderedArrayPriorityQueue.java

1. /\*
2. \* Program #1
3. \* Quang Trinh
4. \* cssc0759
5. \*/
7. **package** data\_structures;
8. **import** java.util.Iterator;
9. **import** java.util.NoSuchElementException;
11. **public** **class** UnorderedArrayPriorityQueue<E **extends** Comparable<E>> **implements** PriorityQueue<E> {
12. **private** E [] storage;
13. **private** **int** currentSize;
14. **private** **int** maxSize;
15. **private** **long** modCounter;
17. **public** UnorderedArrayPriorityQueue(**int** size) {
18. maxSize = size;
19. storage = (E[]) **new** Comparable[maxSize];
20. currentSize = 0;
21. modCounter = 0;
22. }
24. **public** UnorderedArrayPriorityQueue() {
25. **this**(DEFAULT\_MAX\_CAPACITY);
26. }
28. **public** **boolean** insert(E object) {
29. **if**(isFull())
30. **return** **false**;
31. storage[currentSize++] = object;
32. modCounter++;
33. **return** **true**;
34. }
36. **public** E remove() {
37. **if**(isEmpty())
38. **return** **null**;
39. **int** index = highestPriorityIndex(storage);
40. E item = storage[index];
41. **for**(**int** i=index; i < currentSize-1; i++) // Remove the object and shift array left.
42. storage[i] = storage[i+1];
43. currentSize--;
44. modCounter++;
45. **return** item;
46. }
48. **public** **boolean** delete(E obj) {
49. **if**(!contains(obj))
50. **return** **false**;
51. **for**(**int** i=0; i < currentSize; i++) {
52. **if**(storage[i].compareTo(obj) == 0) {
53. storage[i] = storage[i+1];
54. i = 0;
55. currentSize--;
56. }
58. }
59. modCounter++;
60. **return** **true**;
61. }
63. **public** E peek() {
64. **if**(isEmpty())
65. **return** **null**;
66. **return** storage[highestPriorityIndex(storage)];
67. }
69. **public** **boolean** contains(E obj) {
70. **if**(isEmpty()) **return** **false**;
71. **for**(**int** i = 0; i < currentSize; ++i) {
72. **if**(obj.compareTo(storage[i]) == 0)
73. **return** **true**;
74. }
75. **return** **false**;
76. }
78. /\*
79. \* Return the index of the highest priority item in an array.
80. \*/
81. **private** **int** highestPriorityIndex(E[] array) {
82. **int** indexMax = 0;
83. **for**(**int** i = 1; i < currentSize; i++) {
84. **if**(array[indexMax].compareTo(array[i]) > 0 ) {
85. indexMax = i;
86. }
87. }
88. **return** indexMax;
89. }
91. **public** **int** size() {
92. **return** currentSize;
93. }
95. **public** **void** clear() {
96. currentSize = 0;
97. modCounter = 0;
98. }
100. **public** **boolean** isEmpty() {
101. **return** currentSize == 0;
102. }
104. **public** **boolean** isFull() {
105. **return** currentSize == maxSize;
106. }
108. **public** Iterator<E> iterator() {
109. **return** **new** IteratorHelper();
110. }
112. /////////////////////////////////////////////
113. **class** IteratorHelper **implements** Iterator<E> {
114. **int** iterIndex;
115. **long** stateCheck;
117. **public** IteratorHelper() {
118. iterIndex = 0;
119. stateCheck = modCounter;
120. }
122. **public** **boolean** hasNext() {
123. **if**(stateCheck != modCounter)
124. **return** **false**;   // ConcurrentModificationException
125. **return** iterIndex < currentSize;
126. }
128. **public** E next() {
129. **if**(!hasNext())
130. **throw** **new** NoSuchElementException();
131. **return** storage[iterIndex++];
132. }
134. **public** **void** remove() {
135. **throw** **new** UnsupportedOperationException();
136. }
137. }
138. //////////////////////////////////////////////////
139. }