

**Quang Trinh**

**cssc0759**

**CS310-01**

**Program 2**

**03.15.18**

OrderedLinkedListPriorityQueue.java

1. /\*
2. \* Program #2
3. \* Quang Trinh
4. \* cssc0759
5. \*/
7. **package** data\_structures;
9. **import** java.util.ConcurrentModificationException;
10. **import** java.util.Iterator;
11. **import** java.util.NoSuchElementException;
13. **public** **class** OrderedLinkedListPriorityQueue<E **extends** Comparable<E>> **implements** PriorityQueue<E> {
14. //////////////////////////////////////
15. **private** **class** Node<T> {
16. T data;
17. Node<T> next;
19. **public** Node(T data) {
20. **this**.data = data;
21. next = **null**;
22. }
23. }
24. //////////////////////////////////////
26. **private** Node<E> head;
27. **private** **int** currentSize;
28. **private** **long** modCounter;
30. // Constructor
31. **public** OrderedLinkedListPriorityQueue() {
32. head = **null**;
33. currentSize = 0;
34. modCounter = 0;
35. }
37. // Insert in ascending order
38. **public** **boolean** insert(E object) {
39. Node<E> newNode = **new** Node<E>(object);
40. Node<E> prev = **null**, current = head;
42. **while**(current != **null** &&
43. ((Comparable<E>)object).compareTo(current.data) >= 0) {
44. prev = current;
45. current = current.next;
46. }
47. **if**(prev == **null**){               // insert at front
48. newNode.next = head;
49. head = newNode;
50. currentSize++;
51. } **else** **if**(current == **null**) {    // insert at last
52. prev.next = newNode;
53. currentSize++;
54. }
55. **else** {                          // insert in middle
56. newNode.next = current;
57. prev.next = newNode;
58. currentSize++;
59. }
60. modCounter++;
61. **return** **true**;
62. }
64. **public** E remove() {
65. E temp = **null**;
66. **if**(isEmpty())
67. **return** **null**;
68. // always remove first node
69. temp = head.data;
70. head = head.next;
71. currentSize--;
72. modCounter++;
73. **return** temp;
74. }
76. **public** **boolean** delete(E obj) {
77. Node<E> prev = **null**, current = head;
78. **if**(!contains(obj))
79. **return** **false**;
81. // Iterate entire list
82. **while**(current != **null**) {
83. // Remove from the front
84. **if**(prev == **null** && ((Comparable<E>)obj).compareTo(current.data) == 0) {
85. head = head.next;
86. current = current.next; // update pointer for next iteration
87. currentSize--;
88. }
89. // Found obj in the middle of list
90. **else** **if**( ((Comparable<E>)obj).compareTo(current.data) == 0) {
91. prev.next = current.next;
92. current = current.next;
93. currentSize--;
94. }
95. // If not found, keep moving
96. **else** {
97. prev = current;
98. current = current.next;
99. }
100. }
101. modCounter++;
102. **return** **true**;
103. }
105. **public** E peek() {
106. **if**(isEmpty())
107. **return** **null**;
108. **return** head.data;
109. }
111. **public** **boolean** contains(E obj) {
112. Node<E> current = head;
113. **if**(isEmpty())
114. **return** **false**;
115. **while**(current != **null**) {
116. **if**( ((Comparable<E>)obj).compareTo(current.data) == 0)
117. **return** **true**;
118. current = current.next;
119. }
120. **return** **false**;
121. }
123. **public** **int** size() {
124. **return** currentSize;
125. }
127. **public** **void** clear() {
128. head = **null**;
129. currentSize = 0;
130. modCounter = 0;
131. }
133. **public** **boolean** isEmpty() {
134. **return** (head == **null**);
135. }
137. **public** **boolean** isFull() {
138. **return** **false**;
139. }
141. **public** Iterator<E> iterator() {
142. **return** **new** IteratorHelper();
143. }
145. //////////////////////////////////////////////////////
146. **private** **class** IteratorHelper **implements** Iterator<E> {
147. **private** Node<E> current;
148. **private** **long** modCheck;
150. **public** IteratorHelper(){
151. current = head; // start at the front of list
152. modCheck = modCounter;
153. }
155. **public** **boolean** hasNext() {
156. **if**(modCheck != modCounter)
157. **throw** **new** ConcurrentModificationException();
158. **return** (current != **null**);
159. }
161. **public** E next() {
162. **if**(!hasNext())
163. **throw** **new** NoSuchElementException();
164. E temp = current.data;
165. current = current.next;
166. **return** temp;
167. }
169. **public** **void** remove() {
170. **throw** **new** UnsupportedOperationException();
171. }
172. }
173. ////////////////////////////////////////////////////
175. }

UnorderedLinkedListPriorityQueue.java

1. /\*
2. \* Program #2
3. \* Quang Trinh
4. \* cssc0759
5. \*/
7. **package** data\_structures;
9. **import** java.util.ConcurrentModificationException;
10. **import** java.util.Iterator;
11. **import** java.util.NoSuchElementException;
13. **public** **class** UnorderedLinkedListPriorityQueue<E **extends** Comparable<E>> **implements** PriorityQueue<E> {
14. //////////////////////////////////////
15. **private** **class** Node<T>{
16. T data;
17. Node<T> next;
19. **public** Node(T data) {
20. **this**.data = data;
21. next = **null**;
22. }
23. }
24. //////////////////////////////////////
26. **private** Node<E> head;
27. **private** **int** currentSize;
28. **private** **long** modCounter;
30. // Constructor
31. **public** UnorderedLinkedListPriorityQueue() {
32. head = **null**;
33. currentSize = 0;
34. modCounter = 0;
35. }
37. **public** **boolean** insert(E object) {
38. Node<E> newNode = **new** Node<E>(object);
39. newNode.next = head;
40. head = newNode;
41. currentSize++;
42. modCounter++;
43. **return** **true**;
44. }
46. **public** E remove() {
47. **if**(isEmpty())
48. **return** **null**;
49. Node<E> prevWhere, where, prev, current;
50. current = where = head;
51. prevWhere = prev = **null**;
53. // After the loop, "where" points to remove position,
54. // "prevWhere" points to the previous-remove-position
55. **while**(current != **null**) {
56. **if**(prev != **null** && ((Comparable<E>)(current.data)).compareTo(where.data) <= 0) {
57. prevWhere = prev;
58. where = current;
59. }
60. prev = current;
61. current = current.next;
62. }
63. // Remove position is first node of the list
64. **if**(prevWhere == **null**)
65. head = head.next;
66. **else**
67. prevWhere.next = where.next;
68. currentSize--;
69. modCounter++;
70. **return** where.data;
71. }
73. **public** **boolean** delete(E obj) {
74. Node<E> prev = **null**, current = head;
75. **if**(!contains(obj))
76. **return** **false**;
78. // Iterate entire list
79. **while**(current != **null**) {
80. // Remove from the front
81. **if**(prev == **null** && ((Comparable<E>)obj).compareTo(current.data) == 0) {
82. head = head.next;
83. current = current.next; // update pointer for next iteration
84. currentSize--;
85. }
86. // Found obj in the middle of list
87. **else** **if**( ((Comparable<E>)obj).compareTo(current.data) == 0) {
88. prev.next = current.next;
89. current = current.next;
90. currentSize--;
91. }
92. // If not found, keep moving
93. **else** {
94. prev = current;
95. current = current.next;
96. }
97. }
98. modCounter++;
99. **return** **true**;
100. }
102. **public** E peek() {
103. **if**(isEmpty())
104. **return** **null**;
106. Node<E> current, prev, where;
107. current = where = head;
108. prev = **null**;
109. **while**(current != **null**) {
110. **if**(prev != **null** && ((Comparable<E>)(current.data)).compareTo(where.data) <= 0) {
111. where = current;
112. }
113. prev = current;
114. current = current.next;
115. }
116. **return** where.data;
117. }
119. **public** **boolean** contains(E obj) {
120. Node<E> temp = head;
121. **if**(isEmpty())
122. **return** **false**;
123. **while**(temp != **null**) {
124. **if**( ((Comparable<E>)obj).compareTo(temp.data) == 0)
125. **return** **true**;
126. temp = temp.next;
127. }
128. **return** **false**;
129. }
131. **public** **int** size() {
132. **return** currentSize;
133. }
135. **public** **void** clear() {
136. head = **null**;
137. currentSize = 0;
138. modCounter = 0;
139. }
141. **public** **boolean** isEmpty() {
142. **return** (head == **null**);
143. }
145. **public** **boolean** isFull() {
146. **return** **false**;
147. }
149. **public** Iterator<E> iterator() {
150. **return** **new** IteratorHelper();
151. }
153. //////////////////////////////////////////////////////
154. **private** **class** IteratorHelper **implements** Iterator<E> {
155. **private** Node<E> current;
156. **private** **long** modCheck;
158. **public** IteratorHelper(){
159. current = head; // start at the front of list
160. modCheck = modCounter;
161. }
163. **public** **boolean** hasNext() {
164. **if**(modCheck != modCounter)
165. **throw** **new** ConcurrentModificationException();
166. **return** (current != **null**);
167. }
169. **public** E next() {
170. **if**(!hasNext())
171. **throw** **new** NoSuchElementException();
172. E temp = current.data;
173. current = current.next;
174. **return** temp;
175. }
177. **public** **void** remove() {
178. **throw** **new** UnsupportedOperationException();
179. }
180. }
181. //////////////////////////////////////////////////////
182. }