Homework 2: CircularSinglyLinkedLists

Graded

Student

Vidit Dharmendra Pokharna

Total Points

94 / 100 pts

Autograder Score

99.0 / 100.0

Question 2

Feedback & Manual Grading

-5 / 0 pts

✔ **– 5 pts** Efficiency 1

[-1] removeLastOccurence - use .equals() instead of ==

[-5] Effiiciency - for removeLastOccurence, this method can be done in one pass by saving a "previous" node (the node right before the last occurence), so you can simply remove it after; calling removeAtIndex() makes this method do 2 passes of the LL

Great job! -Tomer \ (•□•) /

Autograder Results

Autograder Output

If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs

Submitted Files

CircularSinglyLinkedList.java

Download

```java
import java.util.NoSuchElementException;

/**
 * Your implementation of a CircularSinglyLinkedList without a tail pointer.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators:
 *
 * Resources:
 */
public class CircularSinglyLinkedList<T> {

    /*
     * Do not add new instance variables or modify existing ones.
     */
    private CircularSinglyLinkedListNode<T> head;
    private int size;

    /*
     * Do not add a constructor.
     */

    /**
     * Adds the data to the specified index.
     *
     * Must be O(1) for indices 0 and size and O(n) for all other cases.
     *
     * @param index the index at which to add the new data
     * @param data  the data to add at the specified index
     * @throws java.lang.IndexOutOfBoundsException if index < 0 or index > size
     * @throws java.lang.IllegalArgumentException  if data is null
     */
    public void addAtIndex(int index, T data) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("The index you have provided is outside the range of the array");
        } else if (data == null) {
            throw new IllegalArgumentException("The data provided does not have a value");
        } else if (index == 0) {
            addToFront(data);
        } else if (index == size) {
            addToBack(data);
```

```
46          } else {
47              CircularSinglyLinkedListNode<T> curr = head;
48              int indice = 0;
49              while (indice < index - 1) {
50                  curr = curr.getNext();
51                  indice++;
52              }
53              CircularSinglyLinkedListNode<T> newNode = new CircularSinglyLinkedListNode<T>(data);
54              newNode.setNext(curr.getNext());
55              curr.setNext(newNode);
56              size++;
57          }
58      }
59
60      /**
61       * Adds the data to the front of the list.
62       *
63       * Must be O(1).
64       *
65       * @param data the data to add to the front of the list
66       * @throws java.lang.IllegalArgumentException if data is null
67       */
68      public void addToFront(T data) {
69          if (data == null) {
70              throw new IllegalArgumentException("The data provided does not have a value");
71          } else if (head == null) {
72              CircularSinglyLinkedListNode<T> newNode = new CircularSinglyLinkedListNode<T>(data);
73              head = newNode;
74              head.setNext(head);
75              size++;
76          } else if (head.getNext() == null) {
77              CircularSinglyLinkedListNode<T> newNode = new CircularSinglyLinkedListNode<T>(null);
78              head.setNext(newNode);
79              head.getNext().setNext(head);
80              head.getNext().setData(head.getData());
81              head.setData(data);
82              size++;
83          } else {
84              CircularSinglyLinkedListNode<T> newNode = new CircularSinglyLinkedListNode<T>(null);
85              newNode.setNext(head.getNext());
86              head.setNext(newNode);
87              head.getNext().setData(head.getData());
88              head.setData(data);
89              size++;
90          }
91      }
92
93      /**
94       * Adds the data to the back of the list.
```

```
95      *
96      * Must be O(1).
97      *
98      * @param data the data to add to the back of the list
99      * @throws java.lang.IllegalArgumentException if data is null
100     */
101    public void addToBack(T data) {
102        if (data == null) {
103            throw new IllegalArgumentException("The data provided does not have a value");
104        } else {
105            addToFront(data);
106            head = head.getNext();
107        }
108    }
109
110    /**
111     * Removes and returns the data at the specified index.
112     *
113     * Must be O(1) for index 0 and O(n) for all other cases.
114     *
115     * @param index the index of the data to remove
116     * @return the data formerly located at the specified index
117     * @throws java.lang.IndexOutOfBoundsException if index < 0 or index >= size
118     */
119    public T removeAtIndex(int index) {
120        if (index < 0 || index >= size) {
121            throw new IndexOutOfBoundsException("The index you have provided is outside the range of
    the array");
122        } else if (index == 0) {
123            return removeFromFront();
124        } else {
125            CircularSinglyLinkedListNode<T> curr = head;
126            CircularSinglyLinkedListNode<T> remove = null;
127            int indice = 0;
128            while (indice < index - 1) {
129                curr = curr.getNext();
130                indice++;
131            }
132            remove = curr.getNext();
133            curr.setNext(curr.getNext().getNext());
134            size--;
135            return remove.getData();
136        }
137    }
138
139    /**
140     * Removes and returns the first data of the list.
141     *
142     * Must be O(1).
```

```
143     *
144     * @return the data formerly located at the front of the list
145     * @throws java.util.NoSuchElementException if the list is empty
146     */
147    public T removeFromFront() {
148        if (head == null) {
149            throw new NoSuchElementException("The list is empty so no element can be removed from the
    linked list");
150        } else if (size == 1) {
151            T remove = head.getData();
152            head = null;
153            size--;
154            return remove;
155        } else {
156            T remove = head.getData();
157            head.setData(head.getNext().getData());
158            head.setNext(head.getNext().getNext());
159            size--;
160            return remove;
161        }
162    }
163
164    /**
165     * Removes and returns the last data of the list.
166     *
167     * Must be O(n).
168     *
169     * @return the data formerly located at the back of the list
170     * @throws java.util.NoSuchElementException if the list is empty
171     */
172    public T removeFromBack() {
173        if (head == null) {
174            throw new NoSuchElementException("The list is empty so no element can be removed from the
    linked list");
175        } else if (size == 1) {
176            T remove = head.getData();
177            head = null;
178            size--;
179            return remove;
180        } else {
181            CircularSinglyLinkedListNode<T> curr = head;
182            CircularSinglyLinkedListNode<T> remove = null;
183            while (curr.getNext().getNext() != head) {
184                curr = curr.getNext();
185            }
186            remove = curr.getNext();
187            curr.setNext(head);
188            size--;
189            return remove.getData();
```

```
190         }
191     }
192 
193     /**
194      * Returns the data at the specified index.
195      *
196      * Should be O(1) for index 0 and O(n) for all other cases.
197      *
198      * @param index the index of the data to get
199      * @return the data stored at the index in the list
200      * @throws java.lang.IndexOutOfBoundsException if index < 0 or index >= size
201      */
202     public T get(int index) {
203         if (index < 0 || index >= size) {
204             throw new IndexOutOfBoundsException("The index you have provided is outside the range of
    the array");
205         } else {
206             CircularSinglyLinkedListNode<T> curr = head;
207             int indice = 0;
208             while (indice < index) {
209                 curr = curr.getNext();
210                 indice++;
211             }
212             return curr.getData();
213         }
214     }
215 
216     /**
217      * Returns whether or not the list is empty.
218      *
219      * Must be O(1).
220      *
221      * @return true if empty, false otherwise
222      */
223     public boolean isEmpty() {
224         return (head == null);
225     }
226 
227     /**
228      * Clears the list.
229      *
230      * Clears all data and resets the size.
231      *
232      * Must be O(1).
233      */
234     public void clear() {
235         head = null;
236         size = 0;
237     }
```

```
238
239     /**
240      * Removes and returns the last copy of the given data from the list.
241      *
242      * Do not return the same data that was passed in. Return the data that
243      * was stored in the list.
244      *
245      * Must be O(n).
246      *
247      * @param data the data to be removed from the list
248      * @return the data that was removed
249      * @throws java.lang.IllegalArgumentException if data is null
250      * @throws java.util.NoSuchElementException   if data is not found
251      */
252     public T removeLastOccurrence(T data) {
253         if (data == null) {
254             throw new IllegalArgumentException("The data provided does not have a value");
255         } else if (size == 0) {
256             throw new NoSuchElementException("Through a traversal of the linked list, the data was not
    found");
257         } else if (size == 1) {
258             if (head.getData() == data) {
259                 T remove = head.getData();
260                 head = null;
261                 size--;
262                 return remove;
263             } else {
264                 throw new NoSuchElementException("Through a traversal of the linked list, the data was not
    found");
265             }
266         } else {
267             if (head.getData() == data) {
268                 T remove = head.getData();
269                 head.setData(head.getNext().getData());
270                 head.setNext(head.getNext().getNext());
271                 size--;
272                 return remove;
273             }
274             CircularSinglyLinkedListNode<T> curr = head.getNext();
275             int index = 1;
276             int index1 = -1;
277             while (curr != head) {
278                 if (curr.getData() == data) {
279                     index1 = index;
280                 }
281                 curr = curr.getNext();
282                 index++;
283             }
284             if (index1 == -1) {
```

```
285             throw new NoSuchElementException("Through a traversal of the linked list, the data was not
    found");
286         } else {
287             return removeAtIndex(index1);
288         }
289       }
290   }
291
292   /**
293    * Returns an array representation of the linked list.
294    *
295    * Must be O(n) for all cases.
296    *
297    * @return the array of length size holding all of the data (not the
298    * nodes) in the list in the same order
299    */
300   public T[] toArray() {
301       Object[] array1 = new Object[size];
302       T[] array = (T[]) array1;
303       if (head == null) {
304           return array;
305       } else if (head.getNext() == null) {
306           array[0] = head.getData();
307           return array;
308       } else {
309           array[0] = head.getData();
310           CircularSinglyLinkedListNode<T> curr = head.getNext();
311           int index = 1;
312           while (curr != head) {
313               array[index] = curr.getData();
314               curr = curr.getNext();
315               index++;
316           }
317           return array;
318       }
319   }
320
321   /**
322    * Returns the head node of the list.
323    *
324    * For grading purposes only. You shouldn't need to use this method since
325    * you have direct access to the variable.
326    *
327    * @return the node at the head of the list
328    */
329   public CircularSinglyLinkedListNode<T> getHead() {
330       // DO NOT MODIFY!
331       return head;
332   }
```

```
333
334     /**
335      * Returns the size of the list.
336      *
337      * For grading purposes only. You shouldn't need to use this method since
338      * you have direct access to the variable.
339      *
340      * @return the size of the list
341      */
342     public int size() {
343         // DO NOT MODIFY!
344         return size;
345     }
346 }
347
```