

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3 /**
4  The Linked List Class is the framework for a series of ListNodes
5  that together function like a List. You must create a Linked List
6  containing values all of a certain type. Values can be added and removed
7  from a Linked list, among other functions. This Class also implements
8  Stack, Queue, and Iterable, giving it the functionality of all three
9  @author Zachary Keller
10  @version final
11 */
12 public class LinkedList<E> implements Stack<E>, Queue<E>, Iterable<E>
13 {
14     /**
15      A pointer to the first ListNode in the LinkedList
16     */
17     public ListNode<E> head;
18
19     /**
20      A pointer to the last ListNode in the LinkedList
21     */
22     public ListNode<E> tail;
23
24     /**
25      The length of the LinkedList, aka the number of ListNodes
26     */
27     private int size;
28
29     /**
30      Default Constructor, initializes head and tail pointers to null
31     */
32     public LinkedList()
33     {
34         head = null;
35         tail = null;
36         size = 0;
37     }
38
39     /**
40      Constructor that begins a new Linked List with an existing
41      Node as the head
42      @param h A node that will become the head of the new Linked List
43     */
44     public LinkedList(ListNode<E> h)
45     {
46         this();
47         add(h.getItem());
48     }
49
50     /**
51      Copy Constructor, makes a copy of an existing Linked List
52      @param list The Linked List to be copied
53     */
54     public LinkedList(LinkedList<E> list)
55     {
56         ListNode<E> node = list.head;
57         while (node != null)
58         {
59             add(node.getItem());
60             node = node.getNext();
61         }
62     }
63
64     /**
65      Returns the size of the Linked List
66      @return The size of the Linked List
67     */
68     public int size()
69     {
70         return size;
71     }
72
73     /**
74      adds a new item to a specific spot in the Linked List

```

```

75     @param index Where the item should be added
76     @param item the thing of type E being added
77     @return For a successful addition
78 */
79 public boolean add( int index, E item)
80 {
81     // Makes sure the index is within the size of the Linked List
82     if (index > size || index < 0 )
83     {
84         throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size);
85     }
86     // If the Linked List is empty
87     if (tail == null || index == size)
88     {
89         return add(item);
90     }
91     ListNode<E> holder = head;
92     int num = 0;
93     // Finds the List Node before the spot where the item is to be added
94     while (num < index - 1)
95     {
96         holder = holder.getNext();
97         num++;
98     }
99     if (index > 0)
100     {
101         ListNode<E> l = new ListNode<E>(item, holder.getNext());
102         holder.setNext(l);
103     }
104     else //basically if index is 0
105     {
106         ListNode<E> l = new ListNode<E>(item, head);
107         head = l;
108     }
109     size++;
110     return true;
111 }
112
113 /**
114  Creates and returns an iterator
115  @return The Iterator
116  */
117 public Iterator<E> iterator()
118 {
119     return new LinkedListIterator<E>(head);
120 }
121
122 /**
123  Adds an item to the end of the Linked List
124  @param item the thing to be added
125  @return If the addition was successful
126  */
127 public boolean add(E item)
128 {
129     ListNode<E> l = new ListNode<E>(item);
130     if (tail == null)
131     {
132         head = l;
133         tail = l;
134         size++;
135         return true;
136     }
137     tail.setNext(l);
138     tail = l;
139     size++;
140     return true;
141 }
142
143 /**
144  Removes the ListNode (and therefore item within the ListNode)
145  from a given index
146  @param index The place that will be removed
147  @return The value previously at the index
148  */

```

```

149 public E remove(int index)
150 {
151     // Makes sure a proper index was used
152     if (index > size || index < 0 )
153     {
154         throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
155     }
156     ListNode<E> node = head;
157     E returner;
158     // If you are just trying to remove the head
159     if (index == 0)
160     {
161         removeFirst();
162     }
163     int num = 0;
164     // Gets to the List Node before the one to be removed
165     while (num < index - 1)
166     {
167         node = node.getNext();
168         num++;
169     }
170     returner = node.getNext().getItem();
171     node.setNext(node.getNext().getNext());
172     // Case for if the tail is being removed
173     if (index == size - 1)
174     {
175         tail = node;
176     }
177     size -= 1;
178     return returner;
179 }
180
181 /**
182  Removes the first instance of a given value
183  @param item the desired value to be removed
184  @return For a successful removal
185  */
186 public boolean remove(E item)
187 {
188     if (!contains(item))
189         return false;
190     remove(indexOf(item));
191     return true;
192 }
193
194 /**
195  Checks to see if a given value is in the Linked List
196  @param object the item that is being checked for
197  @return Whether or not the item is contained within the Linked List
198  */
199 public boolean contains(E object)
200 {
201     return indexOf(object) != -1;
202 }
203
204 /**
205  Returns the index of the first instance of an object
206  @param object The item that is being checked for
207  @return The index of the object if it is in the List, -1 if it is not in the List
208  */
209 public int indexOf (E object)
210 {
211     ListNode<E> node = head;
212     int num = 0;
213     while (num < size)
214     {
215         if (object == null)
216         {
217             if (node.getItem() == null)
218                 return num;
219         }
220         else
221         {
222

```

```

223         if (object.equals(node.getItem()))
224             return num;
225     }
226     node = node.getNext();
227     num +=1;
228 }
229 return -1;
230 }
231
232 /**
233  * Empties the LinkedList
234  */
235 public void clear()
236 {
237     head = null;
238     tail = null;
239     size = 0;
240 }
241
242 /**
243  * Returns the Item at a given index
244  * @param index The spot to be gotten
245  * @return The Item at the desired spot
246  */
247 public E get(int index)
248 {
249     if (index > size || index < 0 )
250     {
251         throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size);
252     }
253     ListNode<E> node = head;
254     int num = 0;
255     while (num < index)
256     {
257         node = node.getNext();
258         num+=1;
259     }
260     return node.getItem();
261 }
262
263 /**
264  * Inserts an item at a given location regardless of what is already there
265  * @param o the item to be placed
266  * @param i The spot for the item
267  * @return The item that was previously in that spot
268  */
269 public E set(int i , E o)
270 {
271     add(i, o);
272     E holder = get(i+1);
273     remove(i + 1);
274     return holder;
275 }
276
277 /**
278  * Identifies whether the Linked List is empty; That is to say its size is 0
279  * @return Whether or not it is empty
280  */
281 public boolean isEmpty()
282 {
283     return (head == null);
284 }
285
286 /**
287  * Returns a string representation of the Linked List
288  * @return The string representation of the Linked List
289  */
290 public String toString()
291 {
292     ListNode<E> node = head;
293     String s = "";
294     while (node != null)
295     {
296         s += node.toString();

```

This seems pretty inefficient, even if $3n$ vs n is essentially the same.

```

297         s+= "\n";
298         node = node.getNext();
299     }
300     return s;
301
302 }
303
304 /**
305  Adds and item to the beginning of the linked List- resets the head
306  @param item The value to be pushed
307 */
308 public void push(E item)
309 {
310     addFirst(item);
311 }
312
313 /**
314  Removes and returns the head of the Linked List, adjusts accordingly
315  @return the value of the head / the first item
316 */
317 public E pop()
318 {
319     return removeFirst();
320 }
321
322 /**
323  Returns what is first in the Linked List, aka the head Node
324  BUT does not actually change anything
325  @return Head Node
326 */
327 public E peek()
328 {
329     return get(0);
330 }
331
332 /**
333  Adds an item to the end of the Linked List
334  @param item The thing being added ("offered")
335 */
336 public void offer(E item)
337 {
338     addLast(item);
339 }
340
341 /**
342  Removes and returns the head of the Linked List, adjusts accordingly
343  @return the value of the head / the first item
344 */
345 public E poll()
346 {
347     return removeFirst();
348 }
349
350 /**
351  adds an item to the beginning of the Linked List, adjusts the head accordingly
352  @param item the object being added to the List
353 */
354 public void addFirst(E item)
355 {
356     add(0, item);
357 }
358
359 /**
360  Adds an item to the end of the Linked List
361  @param item The object being added to the List
362 */
363 public void addLast(E item)
364 {
365     add(item);
366 }
367
368 /**
369  Removes and returns the first element in the Linked List
370  @return the item that was removed from the List

```

```

371 */
372 public E removeFirst()
373 {
374     E returner;
375     // Makes sure the Linked List is not empty
376     if (head == null)
377     {
378         throw new NoSuchElementException("Linked List is empty");
379     }
380     returner = head.getItem();
381     head = head.getNext();
382     size -= 1;
383     if (size == 0)
384         tail = null;
385     return returner;
386 }
387
388 /**
389     Removes the Last object in the Linked List, and returns it
390     @return The Item being removed
391 */
392 public E removeLast()
393 {
394     // Makes sure the Linked List is not empty
395     if (head == null)
396     {
397         throw new NoSuchElementException("Linked List is empty");
398     }
399     return remove(size - 1);
400 }
401 }
402
403

```

Great job. Works for all tests.
Grade: A+