```
import java.util.Iterator;
   import java.util.NoSuchElementException;
2
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
       Stack, Queue, and Iterable, giving it the functionality of all three
8
       @author Zachary Keller
10
       @version final
11
12
   public class LinkedList<E> implements Stack<E>, Queue<E>, Iterable<E>
13
   {
14
           A pointer to the first ListNode in the LinkedList
15
16
17
       public ListNode<E> head;
18
19
           A pointer to the last ListNode in the LinkedList
20
21
22
       public ListNode<E> tail;
23
24
           The length of the LinkedList, aka the number of ListNodes
25
26
27
       private int size;
28
29
30
           Default Constructor, initializes head and tail pointers to null
31
       public LinkedList()
32
33
       {
34
           head = null;
           tail = null;
35
           size = 0;
36
37
       }
38
39
           Constructor that begins a new Linked List with an existing
40
           Node as the head
41
           @param h A node that will become the head of the new Linked List
42
43
       public LinkedList(ListNode<E> h)
44
45
       {
46
           this();
47
           add(h.getItem());
48
       }
49
50
           Copy Constructor, makes a copy of an existing Linked List
51
52
           Oparam list The Linked List to be copied
53
       public LinkedList(LinkedList<E> list)
54
55
       {
           ListNode<E> node = list.head;
56
57
           while (node != null)
58
           {
59
                add(node.getItem());
60
                node = node.getNext();
61
           }
       }
62
63
       /**
64
           Returns the size of the Linked List
65
           @return The size of the Linked List
66
67
       public int size()
68
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
             @param item the thing of type E being added
 76
             @return For a successful addition
 77
 78
 79
        public boolean add( int index, E item)
 80
        {
 81
             // Makes sure the index is within the size of the Linked List
             if (index > size || index < 0 )</pre>
 82
 83
             {
                 throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
 84
 85
 86
             // If the Linked List is empty
             if (tail == null || index == size)
 87
 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
            while (num < index - 1)</pre>
 94
 95
 96
                 holder = holder.getNext();
 97
                 num+=1;
 98
             if (index > 0)
 99
100
101
                 ListNode<E> 1 = new ListNode<E>(item, holder.getNext());
102
                 holder.setNext(1);
103
104
             else //basically if index is 0
105
             {
                 ListNode<E> 1 = new ListNode<E>(item, head);
106
107
                 head = 1;
108
109
             size+=1;
110
             return true;
111
        }
112
        /**
113
114
        Creates and returns an iterator
        @return The Iterator
115
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
             @param item the thing to be added
124
             @return If the addition was successful
125
126
127
        public boolean add(E item)
128
        {
129
             ListNode<E> 1 = new ListNode<E>(item);
             if (tail == null)
130
131
             {
132
                 head = 1;
                 tail = 1;
133
134
                 size+=1;
135
                 return true;
136
137
             tail.setNext(1);
138
             tail = 1;
             size+=1;
139
140
             return true;
141
        }
142
143
        /**
144
             Removes the ListNode (and therefore item within the ListNode)
             from a given index
145
146
             Oparam 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 )</pre>
153
             {
                 throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
154
155
             ListNode<E> node = head;
156
             E returner;
157
158
             // If you are just trying to remove the head
             if (index == 0)
159
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)</pre>
166
167
                 node = node.getNext();
                 num+=1;
168
169
             returner = node.getNext().getItem();
170
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
             size -= 1;
177
178
             return returner;
179
        }
180
181
        /**
182
             Removes the first instance of a given value
183
             Oparam item the desired value to be removed
184
             @return For a successful removal
185
        public boolean remove(E item)
186
187
188
             if (! contains(item))
189
                 return false;
190
             remove(indexOf(item));
191
             return true;
192
193
        }
194
195
196
             Checks to see if a given value is in the Linked List
197
             Oparam object the item that is being checked for
             Greturn Whether or not the item is contained within the Linked List
198
199
200
        public boolean contains(E object)
201
        {
202
             return indexOf(object) != -1;
203
        }
204
205
        /**
206
             Returns the index of the first instance of an object
207
             @param object The item that is being checked for
208
             @return The index of the object if it is in the List, -1 if it is not in the List
209
        public int indexOf (E object)
210
211
        {
212
             ListNode<E> node = head;
213
             int num = 0;
214
             while (num < size)</pre>
215
             {
                 if (object == null)
216
217
218
                       if (node.getItem() == null)
219
                          return num;
220
                 }
221
                 else
222
                 {
```

```
223
                      if (object.equals(node.getItem()))
224
                          return num;
225
                 node = node.getNext();
226
227
                 num +=1;
228
229
             return -1;
230
        }
231
         /**
232
             Empties the LinkedList
233
234
235
        public void clear()
236
237
             head = null;
238
             tail = null;
239
             size = 0;
240
        }
241
242
             Returns the Item at a given index
243
244
             Oparam index The spot to be gotten
245
             @return The Item at the desired spot
246
247
        public E get(int index)
248
             if (index > size || index < 0 )</pre>
249
250
             {
                 throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
251
252
253
             ListNode<E> node = head;
             int num = 0;
254
             while (num < index)</pre>
256
257
                 node = node.getNext();
258
                 num+=1;
259
             return node.getItem();
260
261
        }
262
263
        Inserts an item at a given location regardless of what is already there
264
265
        @param o the item to be placed
        @param i The spot for the item
266
267
        @return The item that was previously in that spot
268
        public E set(int i , E o)
269
270
                                        This seems pretty inefficient,
271
             add(i, o);
             E 	ext{ holder} = get(i+1);
                                        even if 3n vs n is essentially
272
             remove(i + 1);
273
                                        the same.
274
             return holder;
275
        }
276
277
        Identifies whether the Linked List is empty; That is to say its size is 0
278
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
        public String toString()
290
291
292
             ListNode<E> node = head;
             String s = "";
293
             while (node != null)
294
295
                 s += node.toString();
296
```

```
297
                  s+= "\n";
                 node = node.getNext();
298
299
300
             return s;
301
302
        }
303
304
305
             Adds and item to the beginning of the linked List- resets the head
306
             Oparam item The value to be pushed
307
308
        public void push(E item)
309
        {
             addFirst(item);
310
311
        }
312
313
             Removes and returns the head of the Linked List, adjusts accordingly {\tt @return} the value of the head / the first item
314
315
316
317
        public E pop()
318
         {
319
             return removeFirst();
320
        }
321
322
             Returns what is first in the Linked List, aka the head Node
323
324
             BUT does not actually change anything
             @return Head Node
325
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
        {
             addLast(item);
338
339
        }
340
341
             Removes and returns the head of the Linked List, adjusts accordingly
342
             @return the value of the head / the first item
343
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
             Oparam item the object being added to the List
352
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
             @param item The object being added to the List
361
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
             @return the item that was removed from the List
370
```

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

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