

LinkedList.java

```
1  package edu.citadel.util;
2
3
4  import java.util.Iterator;
5  import java.util.NoSuchElementException;
6
7
8  /**
9   * This class implements a List by means of a linked data structure.
10  * A List (also known as a <i>sequence</i>) is an ordered collection.
11  * Elements in the list can be accessed by their integer index. The
12  * index of the first element in the list is zero.
13  */
14  public class LinkedList<E> implements Iterable<E>
15  {
16      private Node<E> first;    // reference to the first node
17      private Node<E> last;    // reference to the last node
18      private int size;        // number of elements in the list
19
20
21      /**
22       * A list node contains the data value and a link to the next
23       * node in the linked list.
24       */
25      private static class Node<E>
26      {
27          private E data;
28          private Node<E> next;
29
30
31          /**
32           * Construct a node with the specified data value and link.
33           */
34          public Node(E data, Node<E> next)
35          {
36              this.data = data;
37              this.next = next;
38          }
39
40
41          /**
42           * Construct a node with the given data value
43           */
44          public Node(E data)
45          {
46              this(data, null);
47          }
48      }
49
50
51      /**
52       * An iterator for this singly-linked list.
53       */
54      private static class LinkedListIterator<E> implements Iterator<E>
55      {
56          private Node<E> nextElement;
57
58      }
```

```
58
59     /**
60      * Construct an iterator initialized to the first element in the list.
61      */
62     public LinkedListIterator(Node<E> head)
63     {
64         nextElement = head;
65     }
66
67
68     /**
69      * Returns true if the iteration has more elements.
70      */
71     @Override
72     public boolean hasNext()
73     {
74         return nextElement != null;
75     }
76
77
78     /**
79      * Returns the next element in the list.
80      *
81      * @throws NoSuchElementException if the iteration has no next element.
82      */
83     @Override
84     public E next()
85     {
86         if(this.hasNext()) {
87             E rtnval = nextElement.data;
88             nextElement = nextElement.next;
89             return rtnval;
90         }
91         else throw new NoSuchElementException();
92     }
93
94     // Note: Do not have to implement other methods in interface
95     // Iterator since they have default implementations. The following
96     // is provided for versions of Java prior to version 8.
97
98     /**
99      * Remove operation is not supported by this iterator.
100     *
101     * @throws UnsupportedOperationException always.
102     */
103     @Override
104     public void remove()
105     {
106         throw new UnsupportedOperationException("remove");
107     }
108 }
109
110
111 /**
112  * Helper method: Checks that the specified index is between 0 and size - 1.
113  *
114  * @throws IndexOutOfBoundsException if the index is out of range
115  *      (<code>index < 0 || index >= size()</code>)
116  */
117 private void checkIndex(int index)
```

```
118     {
119         if (index < 0 || index >= size)
120             throw new IndexOutOfBoundsException(Integer.toString(index));
121     }
122
123
124     /**
125      * Helper method: Find the node at a specified index.
126      *
127      * @return a reference to the node at the specified index
128      *
129      * @throws IndexOutOfBoundsException if the index is out of range
130      *      (<tt>index &lt; 0 || index &gt;= size()</tt>)
131      */
132     private Node<E> getNode(int index)
133     {
134         checkIndex(index);
135         Node<E> node = first;
136
137         for (int i = 0; i < index; ++i)
138             node = node.next;
139
140         return node;
141     }
142
143
144     /**
145      * Constructs an empty list.
146      */
147     public LinkedList()
148     {
149         first = null;
150         last = null;
151         size = 0;
152     }
153
154
155     /**
156      * Appends the specified element to the end of the list.
157      */
158     public void add(E element)
159     {
160         if (isEmpty())
161         {
162             first = new Node<E>(element);
163             last = first;
164         }
165         else
166         {
167             last.next = new Node<E>(element);
168             last = last.next;
169         }
170
171         ++size;
172     }
173
174
175     /**
176      * Inserts the specified element at the specified position in the list.
177      *
```

```

178     * @throws IndexOutOfBoundsException if the index is out of range
179     *     (<tt>index < 0 || index > size()</tt>)
180     */
181     public void add(int index, E element)
182     {
183         Node<E> curr = first;
184         Node<E> newNode = new Node<>(element);
185
186         // If inserting at index 0, empty or not
187         if(index == 0) {
188             newNode.next = first;
189             first = newNode;
190             if(size == 0)
191                 last = newNode;
192         }
193
194         // If not empty and inserting in last index
195         else if(index == this.size) {
196             last.next = newNode;
197             last = newNode;
198         }
199
200         // If inserting anywhere else
201         else {
202             checkIndex(index);
203             curr = getNode(index - 1);
204             newNode.next = curr.next;
205             curr.next = newNode;
206         }
207         ++size;
208     }
209
210 /**
211  * Removes all of the elements from this list.
212  */
213     public void clear()
214     {
215         while (first != null)
216         {
217             Node<E> temp = first;
218             first = first.next;
219
220             temp.data = null;
221             temp.next = null;
222         }
223
224         last = null;
225         size = 0;
226     }
227
228 /**
229  * Returns the element at the specified position in this list.
230  *
231  * @throws IndexOutOfBoundsException if the index is out of range
232  *     (<tt>index < 0 || index >= size()</tt>)
233  */
234     public E get(int index)
235     {
236     }

```

```
238         // do not need explicit index check since getNode() does it for us
239         Node<E> node = getNode(index);
240         return node.data;
241     }
242
243
244     /**
245      * Replaces the element at the specified position in this list
246      * with the specified element.
247      *
248      * @returns The data value previously at index
249      * @throws IndexOutOfBoundsException if the index is out of range
250      *      (<tt>index < 0 || index >= size()</tt>)
251      */
252     public E set(int index, E newValue)
253     {
254         Node<E> curr = getNode(index);
255         E rtnval = curr.data;
256         curr.data = newValue;
257         return rtnval;
258     }
259
260
261     /**
262      * Returns the index of the first occurrence of the specified element
263      * in this list, or -1 if this list does not contain the element.
264      */
265     public int indexOf(Object obj)
266     {
267         int index = 0;
268
269         if (obj == null)
270         {
271             for (Node<E> node = first; node != null; node = node.next)
272             {
273                 if (node.data == null)
274                     return index;
275                 else
276                     index++;
277             }
278         }
279         else
280         {
281             for (Node<E> node = first; node != null; node = node.next)
282             {
283                 if (obj.equals(node.data))
284                     return index;
285                 else
286                     index++;
287             }
288         }
289
290         return -1;
291     }
292
293
294     /**
295      * Returns <tt>true</tt> if this list contains no elements.
296      */
297     public boolean isEmpty()
```

```
298     {
299         return this.size == 0;
300     }
301
302
303     /**
304      * Removes the element at the specified position in this list. Shifts
305      * any subsequent elements to the left (subtracts one from their indices).
306      *
307      * @returns the element previously at the specified position
308      *
309      * @throws IndexOutOfBoundsException if the index is out of range
310      *      (<tt>index < 0 || index >= size()</tt>)
311      */
312     public E remove(int index)
313     {
314         checkIndex(index);
315         E rtnval;
316         // If removing first item, empty or not
317         if(index == 0) {
318             rtnval = first.data;
319             if(size > 0)
320                 first = first.next;
321             else {
322                 first = null;
323                 last = null;
324             }
325         }
326         // If not empty and removing last item
327         else if(index == this.size-1) {
328             rtnval = last.data;
329             last = this.getNode(index-1);
330             last.next = null;
331         }
332         // If removing any other item
333         else {
334             Node<E> curr = getNode(index - 1);
335             rtnval = curr.next.data;
336             curr.next = curr.next.next;
337         }
338         --size;
339         return rtnval;
340     }
341
342
343     /**
344      * Returns the number of elements in this list.
345      */
346     public int size()
347     {
348         return this.size;
349     }
350
351
352     /**
353      * Returns an iterator over the elements in this list in proper sequence.
354      */
355     @Override
356     public Iterator<E> iterator()
357     {
```

```

358         return new LinkedListIterator<>(first);
359     }
360
361
362     /**
363      * Returns a string representation of this list.
364      */
365     @Override
366     public String toString()
367     {
368         String rtnval = "[";
369         if(size > 0) {
370             Node<E> curr = this.first;
371             while(curr != null) {
372                 if(curr.data == null) {
373                     rtnval += "null, ";
374                     curr = curr.next;
375                 }
376                 else {
377                     rtnval += curr.data.toString() + ", ";
378                     curr = curr.next;
379                 }
380             }
381             rtnval = rtnval.substring(0, rtnval.length()-2);
382         }
383         rtnval += "]";
384         return rtnval;
385     }
386
387
388     /**
389      * Compares the specified object with this list for equality. Returns true
390      * if and only if both lists contain the same elements in the same order.
391      */
392     @Override
393     @SuppressWarnings("rawtypes")
394     public boolean equals(Object obj)
395     {
396         if (obj == this)
397             return true;
398
399         if (!(obj instanceof LinkedList))
400             return false;
401
402         // cast obj to a linked list
403         LinkedList listObj = (LinkedList) obj;
404
405         // compare elements in order
406         Node<E> node1 = first;
407         Node    node2 = listObj.first;
408
409         while (node1 != null && node2 != null)
410         {
411             // check to see if data values are equal
412             if (node1.data == null)
413             {
414                 if (node2.data != null)
415                     return false;
416             }
417             else

```

```
418         {
419             if (!node1.data.equals(node2.data))
420                 return false;
421         }
422
423         node1 = node1.next;
424         node2 = node2.next;
425     }
426
427     return node1 == null && node2 == null;
428 }
429
430
431 /**
432  * Returns the hash code value for this list.
433  */
434 @Override
435 public int hashCode()
436 {
437     int hashCode = 1;
438     Node<E> node = first;
439
440     while (node != null)
441     {
442         E obj = node.data;
443         hashCode = 31*hashCode + (obj == null ? 0 : obj.hashCode());
444         node = node.next;
445     }
446
447     return hashCode;
448 }
449 }
450
```