10/26/15 01:54:20 /home/15504319/DSA120/DSAAssignment/connorLib/DSALinkedList.java

```
  1  /*****************************************************************************
  2  *    FILE: DSALinkedList.java
  3  *    AUTHOR: Connor Beardsmore - 15504319
  4  *    UNIT: DSA120 Assignment S2- 2015
  5  *    PURPOSE: General LinkedList class storing Objects
  6  *    LAST MOD: 20/10/15
  7  *  REQUIRES: java.util.Iterator
  8  *****************************************************************************/
  9  package connorLib;
 10  import java.util.Iterator;
 11
 12  public class DSALinkedList implements Iterable
 13  {
 14      //CLASSFIELDS
 15      private int numNodes;
 16      private DSALinkedList.DSAListNode head;
 17      private DSALinkedList.DSAListNode tail;
 18
 19  //---------------------------------------------------------------------------
 20               //PRIVATE CLASS, PROMOTES INFORMATION HIDING
 21               private class DSAListNode
 22              {
 23                  //Class Fields
 24                  public Object value;
 25                  public DSAListNode next;
 26                  public DSAListNode prev;
 27
 28              //-----------------------------------------------------------------
 29                  //CONSTRUCTOR Alternate
 30                  //IMPORT: inValue (Object)
 31
 32                  public DSAListNode( Object inValue )
 33                  {
 34                      value = inValue;
 35                      next = null;
 36                      prev = null;
 37                  }
 38
 39              }
 40              //END OF PRIVATE DSAListNode CLASS
 41
 42  //---------------------------------------------------------------------------
 43              //PRIVATE CLASS, PROMOTES INFORMATION HIDING
 44              private class DSALinkedListIterator implements Iterator
 45              {
 46                  private DSALinkedList.DSAListNode iterNext;
 47                  //-------------------------------------------------------------
 48                  public DSALinkedListIterator(DSALinkedList list)
 49                  {
 50                      iterNext = list.head;
 51                  }
 52                  //-------------------------------------------------------------
 53                  public boolean hasNext()
 54                  {
 55                      return (iterNext != null);
 56                  }
 57                  //-------------------------------------------------------------
 58                  public Object next()
 59                  {
 60                      Object value;
 61                      if ( iterNext == null )
 62                      {
 63                          value = null;

 64                      }
 65                      else
 66                      {
 67                          value = iterNext.value;
 68                          iterNext = iterNext.next;
 69                      }
 70                      return value;
 71                  }
 72                  //-------------------------------------------------------------
 73                  public void remove()
 74                  {
 75                      throw new UnsupportedOperationException("not supported");
 76                  }
 77              }
 78
 79  //---------------------------------------------------------------------------
 80      //CONSTRUCTOR Default
 81
 82      public DSALinkedList()
 83      {
 84          numNodes = 0;
```

```
85          head = null;
86          tail = null;
87      }
88  //--------------------------------------------------------------------------
89      //ACCESSOR getLength
90      //EXPORT: numNodes (int)
91
92      public int getLength()
93      {
94          return numNodes;
95      }
96  //--------------------------------------------------------------------------
97      //MUTATOR insertFirst
98      //IMPORT: newValue (Object)
99      //PURPOSE: Insert New Element Into Start Of List
100
101     public void insertFirst( Object newValue )
102     {
103         //Allocation New Node Using Node Constructor
104         DSAListNode newNode = new DSAListNode(newValue);
105
106         //If Empty, Set Tail To Node
107         if ( isEmpty() )
108         {
109             tail = newNode;
110         }
111         else
112         {
113             newNode.next = head;
114             head.prev = newNode;
115         }
116
117         //Set Head to new node regardless
118         head = newNode;
119         numNodes++;
120     }
121 //--------------------------------------------------------------------------
122     //MUTATOR insertLast
123     //IMPORT: newValue (Object)
124     //PURPOSE: Insert New Element Into End Of List
125
126     public void insertLast( Object newValue )
127     {
128         //Allocate New Node
129         DSAListNode newNode = new DSAListNode(newValue);
130
131         //If Empty, Set Tail To Node
132         if ( isEmpty() )
133         {
134             head = newNode;
135         }
136         else
137         {
138             newNode.prev = tail;
139             tail.next = newNode;
140         }
141
142         //Set Tail to new node regardless
143         tail = newNode;
144         numNodes++;
145     }
146 //--------------------------------------------------------------------------
147     //MUTATOR insertSorted
148     //IMPORT: newValue (Object)
149     //PURPOSE: Insert New Element Into Correct Place in list based on compareTo
150
151     public void insertSorted( Object newValue )
152     {
153         //Allocation New Node Using Node Constructor
154         DSAListNode newNode = new DSAListNode(newValue);
155         boolean done = false;
156         //If Empty, Set Tail To Node
157         if ( isEmpty() )
158         {
159             tail = newNode;
160             head = newNode;
161         }
162         //If Value is the largest item, add to end of list
163         else if ( ((Carton)(tail.value)).compareTo( (Carton)newValue ) <= 0 )
164         {
165             tail.next = newNode;
166             newNode.prev = tail;
167             tail = newNode;
168         }
169         //If the list is one element long, and value is the smallest item
170         else if ( head.next == null )
171         {
```

```
172             head = newNode;
173             head.next = tail;
174             tail.prev = newNode;
175         }
176         //Find correct place in list
177         else
178         {
179             DSAListNode currNode = head;
180             while ( (currNode != null) && (done == false) )
181             {
182                 //Iterate across list until item is no longer less than current
183                 Carton item = (Carton)currNode.value;
184                 if ( (item).compareTo(  (Carton)newValue  ) >= 0 )
185                 {
186                     if ( currNode == head )
187                     {
188                         head.prev = newNode;
189                         newNode.next = head;
190                         head = newNode;
191                     }
192 
193                     (currNode.prev).next = newNode;
194                     newNode.prev = currNode.prev;
195                     newNode.next = currNode;
196                     currNode.prev = newNode;
197                     done = true;
198                 }
199                 currNode = currNode.next;
200             }
201         }
202         numNodes++;
203     }
204 //---------------------------------------------------------------------------
205     //ACCESSOR isEmpty
206     //EXPORT: empty (boolean)
207     //PURPOSE: Check if List is Empty (No Elements)
208 
209     public boolean isEmpty()
210     {
211         return ( numNodes == 0 );
212     }
213 //---------------------------------------------------------------------------
214     //ACCESSOR peekFirst
215     //EXPORT: nodeValue (Object)
216     //PURPOSE: View Value Within 1st Element of List
217 
218     public Object peekFirst()
219     {
220         //Can't peek if the list is empty
221         if ( isEmpty() )
222         {
223             throw new IllegalStateException("Can't Peek Empty List");
224         }
225 
226         return head.value;
227     }
228 //---------------------------------------------------------------------------
229     //ACCESSOR peekLast
230     //EXPORT: nodeValue (Object)
231     //PURPOSE: View Value Within Last Element of List
232 
233     public Object peekLast()
234     {
235         //Can't peek if the list is empty
236         if ( isEmpty() )
237         {
238             throw new IllegalStateException("Can't Peek Empty List");
239         }
240 
241         return tail.value;
242     }
243 //---------------------------------------------------------------------------
244     //MUTATOR removeFirst
245     //EXPORT: nodeValue (Object)
246     //PURPOSE: Remove First Element From Start of List
247 
248     public Object removeFirst()
249     {
250         Object nodeValue = null;
251 
252         if ( isEmpty() )
253         {
254             throw new IllegalStateException("Can't Remove If List Empty");
255         }
256         //List is only one node
257         if ( head == tail )
258         {
```

```
259             tail = null;
260         }
261         //Covers all other situations
262         nodeValue = head.value;
263         head = head.next;
264 
265         numNodes--;
266         return nodeValue;
267     }
268 //--------------------------------------------------------------------------
269     //MUTATOR removeLast
270     //EXPORT: nodeValue (Object)
271     //PURPOSE: Remove Last Element From End of List
272 
273     public Object removeLast()
274     {
275         Object nodeValue = null;
276 
277         if ( isEmpty() )
278         {
279             throw new IllegalStateException("Can't Remove If List Empty");
280         }
281 
282         nodeValue = tail.value;
283 
284         //List is only one node
285         if ( head == tail )
286         {
287             head = null;
288             tail = null;
289         }
290         else
291         {
292             DSALinkedList.DSAListNode prevNode = head;
293             while ( prevNode.next != tail )
294             {
295                 prevNode = prevNode.next;
296             }
297 
298             tail = prevNode;
299             tail.next = null;
300         }
301 
302         numNodes--;
303         return nodeValue;
304     }
305 //--------------------------------------------------------------------------
306     //MUTATOR removeSpecific
307     //IMPORT: item (Object)
308     //PURPOSE: Remove node from list whose values matches memory address of item
309 
310    public void removeSpecific(Object item)
311     {
312         if ( isEmpty() )
313         {
314             throw new IllegalStateException("Can't Remove If List Empty");
315         }
316         if ( head == tail )
317         {
318             head = null;
319             tail = null;
320             numNodes--;
321         }
322         else if ( head.value == item )
323         {
324             removeFirst();
325         }
326         else if ( tail.value == item )
327         {
328             removeLast();
329         }
330         else
331         {
332             boolean done = false;
333             DSAListNode currNode = head.next;
334             while ( (currNode != null) && (done == false) )
335             {
336                 if ( currNode.value == item )
337                 {
338                     (currNode.prev).next = currNode.next;
339                     (currNode.next).prev = currNode.prev;
340                     done = true;
341                 }
342                 currNode = currNode.next;
343             }
344             numNodes--;
345             if ( done == false )
```

```
346             {
347                 throw new IllegalStateException("Item doesnt exist in list");
348             }
349         }
350     }
351 //-------------------------------------------------------------------------
352     //IMPERATIVE iterator
353     //EXPORT: New iterator
354     //PURPOSE: Wraps Iterator constructor method to allow outer classes to
355     //          construct an instance of the private inner class
356 
357     public Iterator iterator()
358     {
359         return new DSALinkedListIterator(this);
360     }
361 //-------------------------------------------------------------------------
362 }
```