```
 1 package adt;
 2
 3 import java.io.Serializable;
 4 import java.util.Iterator;
 5
 6 /**
 7  * DoublyLinkedList.java - A class that implements the ADT linked list using a
 8  * doubly linked structure.
 9  *
10  * @author Chew Lip Sin
11  * @param <T> The type of elements stored in the linked list.
12  */
13 public class DoublyLinkedList<T> implements LinkedListInterface<T>, Serializable {
14
15     private Node firstNode;                     // reference to first node
16     private Node lastNode;                      // reference to last node
17     private int num;
18
19     /**
20      * Clears all elements from the linked list.
21      */
22     @Override
23     public void clear() {
24         firstNode = lastNode = null;    // set first and last node to empty
25         num = 0;                        // set number of size to 0
26     }
27
28     /**
29      * Adds a new element to the end of the linked list.
30      *
31      * @param newElement The element to be added.
32      * @return True if the element is added successfully.
33      */
34     @Override
35     public boolean add(T newElement) {
36         if (newElement != null) {
37             Node newNode = new Node(newElement);
38
39             if (isEmpty()) {
40                 firstNode = newNode;        // set new node if empty
41                 lastNode = newNode;
42             } else {
43                 newNode.prev = lastNode;    // arrange the node
44                 lastNode.next = newNode;
45                 lastNode = newNode;
46             }
```

```
47
48              num++;
49          }
50          return false;
51      }
52
53      /**
54       * Adds a new element at the specified index in the linked list.
55       *
56       * @param index The index at which the element should be added.
57       * @param newElement The element to be added.
58       * @return True if the element is added successfully.
59       */
60      @Override
61      public boolean add(int index, T newElement) {
62          if (newElement == null || !inAddRange(index)) {
63              return false;
64          } else {
65              Node newNode = new Node(newElement);
66              if (index == 0) {
67                  if (isEmpty()) {
68                      add(newElement);
69                      return true;        // Return because add function will num++
70                  } else {
71                      newNode.next = firstNode;
72                      firstNode.prev = newNode;
73                      firstNode = newNode;
74                  }
75              } else if (index == num) {
76                  lastNode.next = newNode;
77                  newNode.prev = lastNode;
78                  lastNode = newNode;
79              } else {
80                  Node nodeCurrent = travel(index);
81                  nodeCurrent.prev.next = newNode;
82                  newNode.prev = nodeCurrent.prev;
83                  newNode.next = nodeCurrent;
84                  nodeCurrent.prev = newNode;
85              }
86              num++;
87              return true;
88          }
89      }
90
91      /**
92       * Adds all the provided elements to the end of the linked list.
```

```
 93      *
 94      * @param newElements The elements to be added.
 95      * @return True if all elements are added successfully.
 96      */
 97     @Override
 98     public boolean addAll(T... newElements) {
 99         if (newElements != null) {
100             if (isElementsValid(newElements)) {
101                 for (T element : newElements) {
102                     add(element);
103                 }
104                 return true;
105             }
106         }
107         return false;
108     }
109 
110     /**
111      * Checks if the linked list contains the specified element.
112      *
113      * @param element The element to be checked.
114      * @return True if the element is found in the linked list.
115      */
116     @Override
117     public boolean contains(T element) {
118         if (element != null) {
119             return travel(element) != null;
120         }
121         return false;
122     }
123 
124     @Override
125     public T get(int index) {
126         T data = null;
127         if (inRange(index)) {
128             Node nodeCurrent = travel(index);
129             data = nodeCurrent.data;
130         }
131         return data;
132     }
133 
134     /**
135      * Retrieves the element at the specified index in the linked list.
136      *
137      * @param element The index of the element to retrieve.
138      * @return The element at the specified index, or null if index is out of
```

```
139      * bounds.
140      */
141     @Override
142     public int indexOf(T element) {
143         if (element != null) {
144             int index = 0;
145             for (Node nodeCurrent = firstNode; nodeCurrent != null
146                     && inRange(index); index++, nodeCurrent = nodeCurrent.next) {
147                 if (nodeCurrent.data.equals(element)) {
148                     return index;
149                 }
150             }
151         }
152         return -1;
153     }
154 
155     /**
156      * Checks if the linked list is empty.
157      *
158      * @return True if the linked list is empty.
159      */
160     @Override
161     public boolean isEmpty() {
162         return num == 0;
163     }
164 
165     /**
166      * Removes the first occurrence of the specified element from the linked
167      * list.
168      *
169      * @param element The element to be removed.
170      * @return True if the element is removed successfully.
171      */
172     @Override
173     public boolean remove(T element) {
174         if (element == null || isEmpty()) {
175             return false;
176         } else {
177             Node nodeCurrent = travel(element);
178             if (nodeCurrent != null) {
179                 remove(nodeCurrent);
180                 return true;
181             }
182             return false;
183         }
184     }
```

```
185
186     /**
187      * Removes the element at the specified index from the linked list.
188      *
189      * @param index The index of the element to be removed.
190      * @return True if the element is removed successfully.
191      */
192     @Override
193     public boolean remove(int index) {
194         if (isEmpty() || !inRange(index)) {
195             return false;
196         } else {
197             remove(travel(index));
198             return true;
199         }
200     }
201
202     /**
203      * Removes all occurrences of the specified elements from the linked list.
204      *
205      * @param elements The elements to be removed.
206      * @return True if all specified elements are removed successfully.
207      */
208     @Override
209     public boolean removeAll(T... elements) {
210         if (isEmpty() || !isElementsValid(elements)) {
211             return false;
212         } else {
213             for (T element : elements) {
214                 remove(element);
215             }
216             return true;
217         }
218     }
219
220     /**
221      * Replaces the element at the specified index with the new element.
222      *
223      * @param index The index of the element to be replaced.
224      * @param newElement The new element to be placed at the specified index.
225      * @return True if the replacement is successful.
226      */
227     @Override
228     public boolean set(int index, T newElement) {
229         if (isEmpty() || !inRange(index) || newElement == null) {
230             return false;
```

```
231         } else {
232             Node nodeCurrent = travel(index);
233             nodeCurrent.data = newElement;
234             return true;
235         }
236     }
237 
238     /**
239      * Returns the number of elements in the linked list.
240      *
241      * @return The number of elements in the linked list.
242      */
243     @Override
244     public int sizeOf() {
245         return num;
246     }
247 
248     /**
249      * Creates a new linked list containing elements that match the given
250      * condition.
251      *
252      * @param list The condition to filter the elements.
253      * @return A new linked list containing the filtered elements.
254      */
255     @Override
256     public LinkedListInterface where(WhereClause<T> list) {
257         LinkedListInterface<T> linkedList = new DoublyLinkedList<>();
258 
259         for (Node nodeCurrent = firstNode; nodeCurrent != null; nodeCurrent =
nodeCurrent.next) {
260             if (list.match(nodeCurrent.data)) {
261                 linkedList.add(nodeCurrent.data);
262             }
263         }
264         return linkedList;
265     }
266 
267     /**
268      * Orders the elements in the linked list according to the given condition.
269      *
270      * @param list The condition to order the elements.
271      */
272     @Override
273     public void orderBy(OrderClause<T> list) {
274         int endIndex = num - 1;
275         // Return true if bubble sort pass has changed
```

```
276         // Return false if end index reduced by 1 and continue until next sorting
277         while (bubbleSort(endIndex--, list)) {
278         }
279     }
280 
281     /**
282      * Returns the first element that matches the given condition.
283      *
284      * @param list The condition to search for the first element.
285      * @return The first element that matches the condition, or null if not
286      * found.
287      */
288     @Override
289     public T firstOrDefault(FirstOrDefaultClause<T> list) {
290         T data = null;
291         boolean found = false;
292         for (Node nodeCurrent = firstNode; nodeCurrent != null && !found; nodeCurrent =
nodeCurrent.next) {
293             if (list.match(nodeCurrent.data)) {
294                 data = nodeCurrent.data;
295 
296                 found = true;
297             }
298         }
299         return data;
300     }
301 
302     /**
303      * Returns an iterator over the elements in the linked list.
304      *
305      * @return An iterator over the elements in the linked list.
306      */
307     @Override
308     public Iterator<T> getIterator() {
309         return new DoublyLinkListIterator();
310     }
311 
312     /**
313      * Represents a node in the doubly linked list.
314      */
315     private class Node implements Serializable {
316 
317         private T data;
318         private Node next;
319         private Node prev;
320 
```

```
321         /**
322          * Constructs a new Node with the given data.
323          *
324          * @param data The data to be stored in the node.
325          */
326         private Node(T data) {
327             this.data = data;
328         }
329     }
330 
331     /**
332      * Sorts elements in the linked list using the bubble sort algorithm.
333      *
334      * @param endIndex The index up to which sorting is performed.
335      * @param list The condition for sorting the elements.
336      * @return True if any changes were made during the sorting process.
337      */
338     private boolean bubbleSort(int endIndex, OrderClause<T> list) {
339         int beginIndex = 0;
340         boolean hasChanges = false;
341         for (Node nodeCurrent = firstNode; beginIndex < endIndex; beginIndex++,
nodeCurrent = nodeCurrent.next) {
342             if (list.compare(nodeCurrent.data, nodeCurrent.next.data) ==
OrderClause.MOVE_BACKWARD) {
343                 T temp = nodeCurrent.data;
344                 nodeCurrent.data = nodeCurrent.next.data;
345                 nodeCurrent.next.data = temp;
346                 hasChanges = true;
347             }
348         }
349         return hasChanges;
350     }
351 
352     /**
353      * Removes the specified node from the linked list.
354      *
355      * @param nodeCurrent The node to be removed.
356      */
357     private void remove(Node nodeCurrent) {
358         if (nodeCurrent == firstNode && nodeCurrent == lastNode) {
359             firstNode = null;
360             lastNode = null;
361         } else if (nodeCurrent == firstNode) {
362             firstNode.next.prev = null;
363             firstNode = firstNode.next;
364         } else if (nodeCurrent == lastNode) {
```

```
365             lastNode.prev.next = null;
366             lastNode = lastNode.prev;
367         } else {
368             nodeCurrent.prev.next = nodeCurrent.next;
369             nodeCurrent.next.prev = nodeCurrent.prev;
370         }
371         num--;
372     }
373 
374     /**
375      * Checks if the specified elements are valid (not null).
376      *
377      * @param newElements The elements to be checked.
378      * @return True if all elements are valid, false otherwise.
379      */
380     private boolean isElementsValid(T... newElements) {
381         boolean valid = true;
382         for (int i = 0; i < newElements.length && valid; i++) {
383             if (newElements[i] == null) {
384                 valid = false;
385             }
386         }
387         return valid;
388     }
389 
390     /**
391      * Traverses the linked list to find the node containing the specified
392      * element.
393      *
394      * @param element The element to search for.
395      * @return The node containing the element, or null if not found.
396      */
397     private Node travel(T element) {
398         Node nodeCurrent = firstNode;
399         boolean arrive = false;
400 
401         while (nodeCurrent != null && !arrive) {
402             if (nodeCurrent.data.equals(element)) {
403                 arrive = true;
404             } else {
405                 nodeCurrent = nodeCurrent.next;
406             }
407         }
408         return nodeCurrent;
409     }
410 
```

```
411     /**
412      * Traverses the linked list to find the node at the specified destination
413      * index.
414      *
415      * @param dest The index of the destination node.
416      * @return The node at the specified index.
417      */
418     private Node travel(int dest) {
419         int dev = num / 2;
420         return dest < dev ? travelFromFirstTo(dest) : travelFromLastTo(dest);
421     }
422 
423     /**
424      * Traverses the linked list from the last node towards the specified
425      * destination index.
426      *
427      * @param dest The index of the destination node.
428      * @return The node at the specified index.
429      */
430     private Node travelFromLastTo(int dest) {
431         Node nodeCurrent = lastNode;
432         int begin = num - 1;
433 
434         while (begin != dest) {
435             nodeCurrent = nodeCurrent.prev;
436             begin--;
437         }
438         return nodeCurrent;
439     }
440 
441     /**
442      * Traverses the linked list from the first node towards the specified
443      * destination index.
444      *
445      * @param dest The index of the destination node.
446      * @return The node at the specified index.
447      */
448     private Node travelFromFirstTo(int dest) {
449         Node nodeCurrent = firstNode;
450         int begin = 0;
451 
452         while (begin != dest) {
453             nodeCurrent = nodeCurrent.next;
454             begin++;
455         }
456         return nodeCurrent;
```

```
457     }
458 
459     /**
460      * Checks if the specified index is within the valid range for adding an
461      * element.
462      *
463      * @param index The index to be checked.
464      * @return True if the index is within the valid range, false otherwise.
465      */
466     private boolean inAddRange(int index) {
467         return index >= 0 && index <= num;
468     }
469 
470     /**
471      * Checks if the specified index is within the valid range of the linked
472      * list.
473      *
474      * @param index The index to be checked.
475      * @return True if the index is within the valid range, false otherwise.
476      */
477     private boolean inRange(int index) {
478         return index >= 0 && index < num;
479     }
480 
481     /**
482      * Returns a string representation of the elements in the linked list.
483      *
484      * @return A string representation of the elements in the linked list.
485      */
486     @Override
487     public String toString() {
488         String str = "";
489         for (Node nodeCurrent = firstNode; nodeCurrent != null; nodeCurrent =
nodeCurrent.next) {
490             str += nodeCurrent.data + "\n";
491         }
492         return str;
493     }
494 
495     /**
496      * Implements an iterator for iterating through the linked list.
497      */
498     private class DoublyLinkListIterator implements Iterator<T> {
499 
500         Node nodeCurrent = firstNode;
501 
```

```
502         /**
503          * Checks if there is a next element in the linked list.
504          *
505          * @return True if there is a next element, false otherwise.
506          */
507         @Override
508         public boolean hasNext() {
509             return nodeCurrent != null;
510         }
511 
512         /**
513          * Retrieves the next element from the linked list.
514          *
515          * @return The next element.
516          */
517         @Override
518         public T next() {
519             T data = null;
520             if (hasNext()) {
521                 data = nodeCurrent.data;
522                 nodeCurrent = nodeCurrent.next;
523             }
524             return data;
525         }
526 
527     }
528 }
```