

```

1  /*
2  * Class name: DLList.h
3  * Class description: Serves as the apparatus for the node entries within a doubly linked list. Provides variables
4  * indicating the head and tail nodes, along with the total size of the list. The class is templated so that various
5  * data types can be used as doubly linked lists (such as int, double, float, char, etc.).
6  *
7  * Programmer: Chad Philip Johnson
8  * Date Created: Thursday, March 28th, 2013
9  * Last Date Modified: Sunday, April 14th, 2013
10 *
11 * Sources Used:
12 *     DLNode.h
13 *         - to create instances of individual nodes containing values of the supplied type within a doubly linked list
14 */
15
16 #include <iostream>
17
18 #include "DLNode.h"
19
20 using namespace std;
21
22 #ifndef DLLIST_H
23 #define DLLIST_H
24
25 template<typename T>
26 class DLList
27 {
28
29     public:
30
31     /***** constructor/destructor declarations *****/
32
33     /**
34      * Default constructor. Sets the size of the doubly linked list to zero.
35      */
36     DLList()
37     : uintCount( 0 ), pObjDLNodeHead( NULL ), pObjDLNodeTail( NULL )
38     {
39         /* empty */
40     }
41
42     /**
43      * Default destructor. Calls the clear() function to free memory associated with the current
44      * instance of the list.
45      */
46     ~DLList()
47     {
48         this->clear();
49     }
50

```

```

51  /***** overloaded operator definitions *****/
52
53  /**
54   * Friend overloaded insertion operator.  Displays all of the data members contained within each
55   * node of the doubly linked list.
56   * @param ostreamOut The associated output stream.
57   * @param objDLLListSrc The doubly linked list to have its node data members reported.
58   * @return The modified output stream containing reported values.
59   */
60  friend ostream& operator << ( ostream& ostreamOut, const DLLList<T>& objDLLListSrc )
61  {
62      if( objDLLListSrc.uintCount > 0 )
63      {
64          DLNode<T>* pObjDLNodeSelectedNode = objDLLListSrc.pobjDLNodeHead;
65
66          while( pObjDLNodeSelectedNode->getNext() != NULL )
67          {
68              ostreamOut << pObjDLNodeSelectedNode->getContents() << ",";
69              pObjDLNodeSelectedNode = pObjDLNodeSelectedNode->getNext();
70
71          }
72          ostreamOut << pObjDLNodeSelectedNode->getContents() << endl;
73      }
74
75      return ostreamOut;
76  }
77
78  /***** public function declarations *****/
79
80  /**
81   * Add a node to the front of the doubly linked list.
82   * @param contents The value to be assigned to the added node.
83   */
84  void pushFront( T contents )
85  {
86      if( uintCount > 0 )
87      {
88          DLNode<T>* pObjDLNodeSelectedNode = this->pobjDLNodeHead;
89
90          this->pobjDLNodeHead = new DLNode<T>( contents );
91          this->pobjDLNodeHead->setNext( pObjDLNodeSelectedNode );
92          pObjDLNodeSelectedNode->setPrevious( this->pobjDLNodeHead );
93      }
94      else
95      {
96          this->pobjDLNodeHead = new DLNode<T>( contents );
97          this->pobjDLNodeTail = this->pobjDLNodeHead;
98      }
99
100      uintCount++;

```

```

101     }
102
103     /**
104     * Add a node to the end of the doubly linked list.
105     * @param contents The value to be assigned to the added node.
106     */
107     void pushBack( T contents )
108     {
109         if( uintCount > 0 )
110         {
111             this->pobjDLNodeTail->setNext( new DLNode<T>( contents ) );
112             this->pobjDLNodeTail->getNext()->setPrevious( this->pobjDLNodeTail );
113             this->pobjDLNodeTail = this->pobjDLNodeTail->getNext();
114         }
115         else
116         {
117             this->pobjDLNodeTail = new DLNode<T>( contents );
118             this->pobjDLNodeHead = this->pobjDLNodeTail;
119         }
120
121         uintCount++;
122     }
123
124     /**
125     * Add a node into a sorted (ascending) order.
126     * @param contents The value to be assigned to the added node.
127     */
128     void insert( T contents )
129     {
130         if( uintCount > 0 )
131         {
132             DLNode<T>* pObjDLNodeSelectedNode = this->pobjDLNodeHead;
133             DLNode<T>* pObjDLNodeInsertionNode = NULL;
134
135             while( pObjDLNodeSelectedNode->getNext() != NULL )
136             {
137                 if( ( ( pObjDLNodeSelectedNode->getContents() < contents ) && ( pObjDLNodeSelectedNode->getNext()->
138                     getContents() > contents ) )
139                     || ( pObjDLNodeSelectedNode->getContents() == contents ) )
140                 {
141                     pObjDLNodeInsertionNode = new DLNode<T>( contents );
142
143                     pObjDLNodeInsertionNode->setNext( pObjDLNodeSelectedNode->getNext() );
144                     pObjDLNodeSelectedNode->getNext()->setPrevious( pObjDLNodeInsertionNode );
145
146                     pObjDLNodeInsertionNode->setPrevious( pObjDLNodeSelectedNode );
147                     pObjDLNodeSelectedNode->setNext( pObjDLNodeInsertionNode );
148
149                     uintCount++;
150                     return;

```

```

150         }
151         else
152         {
153             pObjDLNodeSelectedNode = pObjDLNodeSelectedNode->getNext();
154         }
155     }
156
157     if( pObjDLNodeSelectedNode->getContents() <= contents )
158     {
159         pushBack( contents );
160         return;
161     }
162     else if( pObjDLNodeSelectedNode->getContents() > contents )
163     {
164         pushFront( contents );
165         return;
166     }
167 }
168 else
169 {
170     pushFront( contents );
171     return;
172 }
173 }
174
175 /**
176  * Check to see if a value is contained within the doubly linked list.
177  * @param target The value to be checked.
178  * @return bool True if value exists in the list; false if not.
179  */
180 bool get( T target )
181 {
182     if( uintCount > 0 )
183     {
184         DLNode<T>* pObjDLNodeSelectedNode = pObjDLNodeHead;
185
186         while( pObjDLNodeSelectedNode->getNext() != NULL )
187         {
188             if( pObjDLNodeSelectedNode->getContents() == target )
189             {
190                 return true;
191             }
192             else
193             {
194                 pObjDLNodeSelectedNode = pObjDLNodeSelectedNode->getNext();
195             }
196         }
197
198         if( pObjDLNodeSelectedNode->getContents() == target )
199         {

```

```

200         return true;
201     }
202 }
203
204     return false;
205 }
206
207 /**
208  * Remove the node at the front of the list.
209  */
210 void popFront()
211 {
212     if( uintCount > 0 )
213     {
214         DLNode<T>* pObjDLNodeDeleteNode = this->pobjDLNodeHead;
215         if( pObjDLNodeDeleteNode->getNext() != NULL )
216         {
217             this->pobjDLNodeHead = pObjDLNodeDeleteNode->getNext();
218             pObjDLNodeHead->setPrevious( NULL );
219         }
220
221         delete pObjDLNodeDeleteNode;
222         uintCount--;
223     }
224     else
225     {
226         return;
227     }
228 }
229
230 /**
231  * Remove the node at the end of the list.
232  */
233 void popBack()
234 {
235     if( uintCount > 0 )
236     {
237         DLNode<T>* pObjDLNodeDeleteNode = this->pobjDLNodeTail;
238         if( pObjDLNodeDeleteNode->getPrevious() != NULL )
239         {
240             this->pobjDLNodeTail = pObjDLNodeDeleteNode->getPrevious();
241             pObjDLNodeTail->setNext( NULL );
242         }
243
244         delete pObjDLNodeDeleteNode;
245         uintCount--;
246     }
247     else
248     {
249         return;

```

```

250     }
251 }
252
253 /**
254  * Remove the first appearance of a value within the list, if it exists.
255  * @param target The value to be removed.
256  * @return bool True if successful; false if not.
257  */
258 bool removeFirst( T target )
259 {
260     if( uintCount > 0 )
261     {
262         DLNode<T>* pObjDLNodeSelectedNode = this->pobjDLNodeHead;
263         DLNode<T>* pObjDLNodeDeleteNode = NULL;
264
265         if( pObjDLNodeSelectedNode->getContents() == target )
266         {
267             this->pobjDLNodeHead = pObjDLNodeSelectedNode->getNext();
268
269             if( pObjDLNodeHead != NULL )
270             {
271                 this->pobjDLNodeHead->setPrevious( NULL );
272             }
273
274             pObjDLNodeDeleteNode = pObjDLNodeSelectedNode;
275         }
276         else
277         {
278             while( pObjDLNodeSelectedNode->getNext() != NULL )
279             {
280                 if( pObjDLNodeSelectedNode->getNext()->getContents() == target )
281                 {
282                     pObjDLNodeDeleteNode = pObjDLNodeSelectedNode->getNext();
283                     pObjDLNodeSelectedNode->setNext( pObjDLNodeDeleteNode->getNext() );
284
285                     if( pObjDLNodeDeleteNode->getNext() != NULL )
286                     {
287                         pObjDLNodeDeleteNode->getNext()->setPrevious( pObjDLNodeSelectedNode );
288                     }
289
290                     break;
291                 }
292                 else
293                 {
294                     pObjDLNodeSelectedNode = pObjDLNodeSelectedNode->getNext();
295                 }
296             }
297
298             if( pObjDLNodeDeleteNode == this->pobjDLNodeTail )
299             {

```

```

300         this->pobjDLNodeTail = pObjDLNodeDeleteNode->getPrevious();
301         this->pobjDLNodeTail->setNext( NULL );
302     }
303 }
304
305 if( pObjDLNodeDeleteNode != NULL )
306 {
307     delete pObjDLNodeDeleteNode;
308     uintCount--;
309     return true;
310 }
311 }
312
313 return false;
314 }
315
316 /**
317  * Remove all occurrences of a value within a list, provided at least one exists.
318  * @param target The value to be removed.
319  * @return bool True if value was removed; false if not.
320  */
321 bool removeAll( T target )
322 {
323     if( uintCount > 0 )
324     {
325         DLNode<T>* pObjDLNodeSelectedNode = this->pobjDLNodeHead;
326         DLNode<T>* pObjDLNodeDeleteNode = NULL;
327
328         while( pObjDLNodeSelectedNode->getContents() == target )
329         {
330             pObjDLNodeDeleteNode = pObjDLNodeSelectedNode;
331             this->pobjDLNodeHead = pObjDLNodeDeleteNode->getNext();
332
333             delete pObjDLNodeDeleteNode;
334             uintCount--;
335
336             pObjDLNodeSelectedNode = this->pobjDLNodeHead;
337             if( pObjDLNodeSelectedNode == NULL )
338             {
339                 this->pobjDLNodeTail = NULL;
340                 return true;
341             }
342         }
343
344         if( pObjDLNodeSelectedNode->getPrevious() != NULL )
345         {
346             this->pobjDLNodeHead->setPrevious( NULL );
347         }
348
349         pObjDLNodeSelectedNode = pObjDLNodeSelectedNode->getNext();

```

```

350
351     if( pObjDLNodeSelectedNode != NULL )
352     {
353         while( pObjDLNodeSelectedNode->getNext() != NULL )
354         {
355             if( pObjDLNodeSelectedNode->getContents() == target )
356             {
357                 pObjDLNodeDeleteNode = pObjDLNodeSelectedNode;
358
359                 pObjDLNodeDeleteNode->getPrevious()->setNext( pObjDLNodeDeleteNode->getNext() );
360                 pObjDLNodeDeleteNode->getNext()->setPrevious( pObjDLNodeDeleteNode->getPrevious() );
361
362                 pObjDLNodeSelectedNode = pObjDLNodeDeleteNode->getPrevious();
363
364                 delete pObjDLNodeDeleteNode;
365                 uintCount--;
366             }
367
368             pObjDLNodeSelectedNode = pObjDLNodeSelectedNode->getNext();
369         }
370
371         if( pObjDLNodeSelectedNode->getContents() == target )
372         {
373             pObjDLNodeDeleteNode = pObjDLNodeSelectedNode;
374             pObjDLNodeDeleteNode->getPrevious()->setNext( NULL );
375             this->pObjDLNodeTail = pObjDLNodeDeleteNode->getPrevious();
376
377             delete pObjDLNodeDeleteNode;
378             uintCount--;
379         }
380     }
381
382     if( pObjDLNodeDeleteNode != NULL )
383     {
384         return true;
385     }
386 }
387
388 return false;
389 }
390
391 /**
392  * Free all memory associated with this doubly linked list and reset the list to its default (empty) configuration.
393  */
394 void clear()
395 {
396     if( uintCount > 0 )
397     {
398         DLNode<T>* pObjDLNodeDeleteNode = this->pObjDLNodeHead;
399         DLNode<T>* pObjDLNodeSelectedNode = this->pObjDLNodeHead->getNext();

```



```

400
401     do
402     {
403         delete pobjDLNodeDeleteNode;
404
405         pobjDLNodeDeleteNode = pobjDLNodeSelectedNode;
406
407         if( pobjDLNodeDeleteNode != NULL )
408         {
409             pobjDLNodeSelectedNode = pobjDLNodeSelectedNode->getNext();
410         }
411     }
412     while( pobjDLNodeDeleteNode != NULL );
413
414     this->pobjDLNodeHead = NULL;
415     this->pobjDLNodeTail = NULL;
416     uintCount = 0;
417 }
418 }
419
420 /***** accessor/mutator function declarations *****/
421
422 /**
423  * Report the current size of the doubly linked list.
424  * @return unsigned int The current size of the list.
425  */
426 unsigned int getSize() const
427 {
428     return this->uintCount;
429 }
430
431 /**
432  * Report the data stored within the node at the front of the list.
433  * @return T The value stored within the node at the front of the list.
434  */
435 T getFront() const
436 {
437     if( uintCount == 0 )
438     {
439         throw "LIST EMPTY";
440     }
441     else
442     {
443         return this->pobjDLNodeHead->getContents();
444     }
445 }
446
447 /**
448  * Report the data stored within the node at the end of the list.
449  * @return T The value stored within the node at the end of the list.

```

```
450     */
451     T getBack() const
452     {
453         if( uintCount == 0 )
454         {
455             throw "LIST EMPTY";
456         }
457         else
458         {
459             return this->pobjDLNodeTail->getContents();
460         }
461     }
462
463     private:
464
465     /***** private variable declarations *****/
466
467     unsigned int uintCount;
468     DLNode<T>* pobjDLNodeHead;
469     DLNode<T>* pobjDLNodeTail;
470
471 };
472
473 #endif
474
```