```
 1
 2  // COS30008, Final Exam, 2023
 3
 4  #pragma once
 5
 6  #include "DoublyLinkedList.h"
 7  #include "DoublyLinkedListIterator.h"
 8
 9  template<typename T>
10  class List
11  {
12  private:
13      using Node = typename DoublyLinkedList<T>::Node;
14
15      Node fHead;
16      Node fTail;
17      size_t fSize;
18
19  public:
20
21      using Iterator = DoublyLinkedListIterator<T>;
22
23      List() noexcept :
24          fSize(0)
25      {}
26
27      // Final Exam, 2023
28      ~List()
29      {
30          // Problem 1
31          Node lCurrent = fTail;
32          fTail.reset();
33          while (lCurrent) {
34              Node lPrevious = lCurrent->fPrevious.lock();
35              if (lPrevious) {
36                  lPrevious->fNext.reset();
37                  lCurrent.reset();
38              }
39
40              lCurrent = lPrevious;
41          }
42      }
43
44      List( const List& aOther ) :
45          List()
46      {
47          for ( auto& item : aOther )
48          {
49              push_back( item );
```

```
50          }
51      }
52
53      List operator=( const List& aOther )
54      {
55          if ( this != &aOther )
56          {
57              this->~List();
58
59              new (this) List( aOther );
60          }
61
62          return *this;
63      }
64
65      List( List&& aOther ) noexcept :
66          List()
67      {
68          swap( aOther );
69      }
70
71      List operator=( List&& aOther ) noexcept
72      {
73          if ( this != &aOther )
74          {
75              swap( aOther );
76          }
77
78          return *this;
79      }
80
81      void swap( List& aOther ) noexcept
82      {
83          std::swap( fHead, aOther.fHead );
84          std::swap( fTail, aOther.fTail );
85          std::swap( fSize, aOther.fSize );
86      }
87
88      size_t size() const noexcept
89      {
90          return fSize;
91      }
92
93      template<typename U>
94      void push_front( U&& aData )
95      {
96          Node lNode = DoublyLinkedList<T>::makeNode( std::forward<U>(aData) );
97
98          if ( !fHead )                                   // first element
```

```
 99          {
100              fTail = lNode;                          // set tail to first element
101          }
102          else
103          {
104              lNode->fNext = fHead;                   // new node becomes head
105              fHead->fPrevious = lNode;               // new node previous of head
106          }
107
108          fHead = lNode;                              // new head
109          fSize++;                                    // increment size
110      }
111
112      template<typename U>
113      void push_back( U&& aData )
114      {
115          Node lNode = DoublyLinkedList<T>::makeNode( std::forward<U>(aData) );
116
117          if ( !fTail )                               // first element
118          {
119              fHead = lNode;                          // set head to first element
120          }
121          else
122          {
123              lNode->fPrevious = fTail;               // new node becomes tail
124              fTail->fNext = lNode;                   // new node next of tail
125          }
126
127          fTail = lNode;                              // new tail
128          fSize++;                                    // increment size
129      }
130
131      void remove( const T& aElement ) noexcept
132      {
133          Node lNode = fHead;                         // start at first
134
135          while ( lNode )                             // Are there still nodes available?
136          {
137              if ( lNode->fData == aElement )         // Have we found the node?
138              {
139                  break;                              // stop the search
140              }
141
142              lNode = lNode->fNext;                   // move to next node
143          }
```

```
144
145         if ( lNode )                                // We have found a first
              matching node.
146         {
147             if ( fHead == lNode )                   // remove head
148             {
149                 fHead = lNode->fNext;               // make lNode's next head
150             }
151
152             if ( fTail == lNode )                   // remove tail
153             {
154                 fTail = lNode->fPrevious.lock();    // make lNode's previuos
                      tail, requires std::shared_ptr
155             }
156
157             lNode->isolate();                       // isolate node,
                  automatically freed
158             fSize--;                                // decrement count
159         }
160     }
161
162     const T& operator[]( size_t aIndex ) const
163     {
164         assert( aIndex < fSize );
165
166         Node lNode = fHead;
167
168         while ( aIndex-- )
169         {
170             lNode = lNode->fNext;
171         }
172
173         return lNode->fData;
174     }
175
176     Iterator begin() const noexcept
177     {
178         return Iterator( fHead, fTail );
179     }
180
181     Iterator end() const noexcept
182     {
183         return begin().end();
184     }
185
186     Iterator rbegin() const noexcept
187     {
188         return begin().rbegin();
189     }
```

```
190
191     Iterator rend() const noexcept
192     {
193         return begin().rend();
194     }
195 };
196
```