```
 1
 2  // COS30008, Final Exam, 2024
 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      // Problem 1
28      ~List() noexcept {
29          Node lCurrent = fTail;
30          fTail.reset();
31
32          Node lPrevious;
33          while (lCurrent != nullptr) {
34              lPrevious = lCurrent->fPrevious.lock();
35              if (lPrevious) {
36                  lPrevious->fNext.reset();
37              }
38              else {
39                  fHead.reset();
40              }
41              lCurrent = lPrevious;
42          }
43      }
44
45
46      List(const List& aOther) :
47          List()
48      {
49          for (auto& item : aOther)
50          {
51              push_back(item);
52          }
53      }
```

```
54
55      List& operator=(const List& aOther)
56      {
57          if (this != &aOther)
58          {
59              this->~List();
60
61              new (this) List(aOther);
62          }
63
64          return *this;
65      }
66
67      List(List&& aOther) noexcept :
68          List()
69      {
70          swap(aOther);
71      }
72
73      List& operator=(List&& aOther) noexcept
74      {
75          if (this != &aOther)
76          {
77              swap(aOther);
78          }
79
80          return *this;
81      }
82
83      void swap(List& aOther) noexcept
84      {
85          std::swap(fHead, aOther.fHead);
86          std::swap(fTail, aOther.fTail);
87          std::swap(fSize, aOther.fSize);
88      }
89
90      size_t size() const noexcept
91      {
92          return fSize;
93      }
94
95      template<typename U>
96      void push_front(U&& aData)
97      {
98          Node lNode = DoublyLinkedList<T>::makeNode(std::forward<U>
              (aData));
99
100         if (!fHead)                                  // first element
101         {
102             fTail = lNode;                             // set tail to
                  first element
103         }
104         else
```

```
105         {
106             lNode->fNext = fHead;                    // new node becomes
                   head
107             fHead->fPrevious = lNode;                // new node
                  previous of head
108         }
109
110         fHead = lNode;                               // new head
111         fSize++;                                     // increment size
112     }
113
114     template<typename U>
115     void push_back(U&& aData)
116     {
117         Node lNode = DoublyLinkedList<T>::makeNode(std::forward<U>
              (aData));
118
119         if (!fTail)                                 // first element
120         {
121             fHead = lNode;                           // set head to
                  first element
122         }
123         else
124         {
125             lNode->fPrevious = fTail;                // new node becomes
                   tail
126             fTail->fNext = lNode;                    // new node next of
                   tail
127         }
128
129         fTail = lNode;                               // new tail
130         fSize++;                                     // increment size
131     }
132
133     void remove(const T& aElement) noexcept
134     {
135         Node lNode = fHead;                          // start at first
136
137         while (lNode)                               // Are there still
              nodes available?
138         {
139             if (lNode->fData == aElement)           // Have we found the
                  node?
140             {
141                 break;                               // stop the search
142             }
143
144             lNode = lNode->fNext;                    // move to next
                  node
145         }
146
147         if (lNode)                                  // We have found a
              first matching node.
```

```
148         {
149             if (fHead == lNode)                    // remove head
150             {
151                 fHead = lNode->fNext;               // make lNode's
                      next head
152             }
153
154             if (fTail == lNode)                    // remove tail
155             {
156                 fTail = lNode->fPrevious.lock();    // make lNode's
                      previuos tail, requires std::shared_ptr
157             }
158
159             lNode->isolate();                       // isolate node,
                  automatically freed
160             fSize--;                                // decrement count
161         }
162     }
163
164     const T& operator[](size_t aIndex) const
165     {
166         assert(aIndex < fSize);
167
168         Node lNode = fHead;
169
170         while (aIndex--)
171         {
172             lNode = lNode->fNext;
173         }
174
175         return lNode->fData;
176     }
177
178     Iterator begin() const noexcept
179     {
180         return Iterator(fHead, fTail);
181     }
182
183     Iterator end() const noexcept
184     {
185         return begin().end();
186     }
187
188     Iterator rbegin() const noexcept
189     {
190         return begin().rbegin();
191     }
192
193     Iterator rend() const noexcept
194     {
195         return begin().rend();
196     }
197 };
```