```
 1
 2  // COS30008, List, Problem Set 3, 2022
 3
 4  #pragma once
 5
 6  #include "DoublyLinkedList.h"
 7  #include "DoublyLinkedListIterator.h"
 8
 9  #include <stdexcept>
10
11  template<typename T>
12  class List
13  {
14  private:
15      // auxiliary definition to simplify node usage
16      using Node = DoublyLinkedList<T>;
17
18      Node* fRoot;    // the first element in the list
19      size_t fCount;  // number of elements in the list
20
21  public:
22      // auxiliary definition to simplify iterator usage
23      using Iterator = DoublyLinkedListIterator<T>;
24
25      ~List()                                                          //    ↵
          destructor - frees all nodes
26      {
27          while ( fRoot != nullptr )
28          {
29              if ( fRoot != &fRoot->getPrevious() )                    //    ↵
                  more than one element
30              {
31                  Node* lTemp = const_cast<Node*>(&fRoot->getPrevious()); //    ↵
                      select last
32
33                  lTemp->isolate();                                    //    ↵
                      remove from list
34                  delete lTemp;                                        //    ↵
                      free
35              }
36              else
37              {
38                  delete fRoot;                                        //    ↵
                      free last
39                  break;                                               //    ↵
                      stop loop
40              }
41          }
42      }
```

```
43
44      void remove( const T& aElement )                                    //
          remove first match from list
45      {
46          Node* lNode = fRoot;                                            //
              start at first
47
48          while ( lNode != nullptr )                                      // Are
              there still nodes available?
49          {
50              if ( **lNode == aElement )                                  //
                  Have we found the node?
51              {
52                  break;                                                  //
                      stop the search
53              }
54
55              if ( lNode != &fRoot->getPrevious() )                       // not
                  reached last
56              {
57                  lNode = const_cast<Node*>(&lNode->getNext());           // go
                      to next
58              }
59              else
60              {
61                  lNode = nullptr;                                        //
                      stop search
62              }
63          }
64
65          // At this point we have either reached the end or found the node.
66          if ( lNode != nullptr )                                         // We
              have found the node.
67          {
68              if ( fCount != 1 )                                          // not
                  the last element
69              {
70                  if ( lNode == fRoot )
71                  {
72                      fRoot = const_cast<Node*>(&fRoot->getNext());       //
                        make next root
73                  }
74              }
75              else
76              {
77                  fRoot = nullptr;                                        //
                      list becomes empty
78              }
79
```

```
 80              lNode->isolate();                                       //  ↵
                   isolate node
 81              delete lNode;                                           //  ↵
                   release node's memory
 82              fCount--;                                               //  ↵
                   decrement count
 83          }
 84      }
 85
 86      ////////////////////////////////////////////////////////////////
 87      //// PS3
 88      ////////////////////////////////////////////////////////////////
 89
 90      // P1
 91
 92      List() // default constructor
 93      {
 94          fCount = 0;
 95      }
 96
 97      bool empty() const
 98      {
 99          return fCount == 0;
100      }
101
102      // Is list empty?
103      size_t size() const
104      {
105          return fCount;
106      };                      // list size
107
108      void push_front(const T& aElement)
109      {
110          Node* node = new Node(aElement);
111
112          if (!empty()) {
113              fRoot->push_front(*node);
114          }
115
116          fRoot = node;
117          fCount++;
118      }
119
120      // return a forward iterator
121      Iterator begin() const {
122          return Iterator(fRoot).begin();
123      }
124      // return a forward end iterator
125      Iterator end() const {
```

```
126         return Iterator(fRoot).end();
127     }
128     // return a backwards iterator
129     Iterator rbegin() const {
130         return Iterator(fRoot).rbegin();
131     }
132     // return a backwards end iterator
133     Iterator rend() const {
134         return Iterator(fRoot).rend();
135     }
136     // P2
137     //
138      // adds aElement at back
139     void push_back(const T& aElement) {
140 
141         Node* node = new Node(aElement);
142 
143 
144         if (!empty()) {
145             fRoot->push_front(*node);
146         }
147         else {
148             fRoot = node;
149         }
150 
151 
152         fCount++;
153     }
154 
155     // P3
156 
157     const T& operator[](size_t aIndex) const
158     {
159         if (aIndex >= fCount || aIndex < 0)
160         {
161             throw std::out_of_range("Index out of bounds.");
162         }
163 
164         int currentIndex = 0;
165         Iterator iter = begin();
166 
167         for (int i = 0; i != aIndex; i++)
168         {
169             iter++;
170         }
171 
172         return *iter;
173     }
174 
```

```
175     // P4
176
177     // copy constructor
178     List(const List& aOtherList)
179     {
180         for (const T& element : aOtherList)
181         {
182             this->push_back(element);
183         }
184     };
185
186     // assignment operator
187     List& operator=(const List& aOtherList) {
188
189         if (&aOtherList != this) {
190
191             this->~List();
192             this->fCount = 0;
193
194             for (const T& element : aOtherList)
195             {
196                 this->push_back(element);
197             }
198         }
199
200         return *this;
201     }
202
203     // P5
204
205     // move constructor
206     List(List&& aOtherList) {
207
208         for (auto iter = aOtherList.begin(); iter != iter.end(); iter =
              aOtherList.begin())
209         {
210             auto val = *iter;
211             this->push_back(std::move(val));
212             aOtherList.remove(*iter);
213         }
214     }
215
216     // move assignment operator
217     List& operator=(List&& aOtherList) {
218         if (&aOtherList != this) {
219
220             this->~List();
221             this->fCount = 0;
222
```

```
223             for (auto iter = aOtherList.begin(); iter != iter.end(); iter =
                  aOtherList.begin())
224             {
225                 auto val = *iter;
226                 this->push_back(std::move(val));
227                 aOtherList.remove(*iter);
228             }
229         }
230
231         return *this;
232     }
233
234     // move push_front
235     void push_front(T&& aElement) {
236
237         Node* node = new Node(std::move(aElement));
238
239         if (!empty()) {
240             fRoot->push_front(*node);
241         }
242
243         fRoot = node;
244         fCount++;
245     }
246
247     // move push_back
248     void push_back(T&& aElement) {
249         Node* node = new Node(std::move(aElement));
250
251         if (!empty()) {
252             fRoot->push_front(*node);
253         }
254         else {
255             fRoot = node;
256         }
257
258         fCount++;
259     }
260 };
261
262
263
```