```cpp
#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"
#include <stdexcept>

template<typename T>
class List
{
private:
    // auxiliary definition to simplify node usage
    using Node = DoublyLinkedList<T>;

    Node* fRoot; // the first element in the list
    size_t fCount; // number of elements in the list

public:
    // auxiliary definition to simplify iterator usage
    using Iterator = DoublyLinkedListIterator<T>;

    ~List()
    {
        while (fRoot != nullptr)
        {
            if (fRoot != &fRoot->getPrevious())
            {
                Node* lTemp = const_cast<Node*>(&fRoot->getPrevious());
                lTemp->isolate();
                delete lTemp;
            }
            else
            {
                delete fRoot;
                break;
            }
        }
    } // destructor - frees all nodes

    void remove(const T& aElement)
    {
        Node* lNode = fRoot;
        while (lNode != nullptr)
        {
            if (**lNode == aElement)
            {
                break;
            }
            if (lNode != &fRoot->getPrevious())
            {
```

```
50                  lNode = const_cast<Node*>(&lNode->getNext());
51              }
52              else
53              {
54                  lNode = nullptr;
55              }
56          }
57          if (lNode != nullptr)
58          {
59              if (fCount != 1)
60              {
61                  if (lNode == fRoot)
62                  {
63                      fRoot = const_cast<Node*>(&fRoot->getNext());
64                  }
65              }
66              else
67              {
68                  fRoot = nullptr;
69              }
70              lNode->isolate();
71              delete lNode;
72              fCount--;
73          }
74      } // remove first match from list
75
76      // P1
77
78      List(): fRoot(nullptr), fCount(0) {} // default constructor
79
80      bool empty() const
81      {
82          return fRoot == nullptr;
83      } // Is list empty?
84
85      size_t size() const
86      {
87          return fCount;
88      } // list size
89
90      void push_front(const T& aElement)
91      {
92          if (empty())
93          {
94              fRoot = new Node(aElement);
95          }
96          else
97          {
98              Node* lNode = new Node(aElement);
```

```
 99              fRoot->push_front(*lNode);
100              fRoot = lNode;
101          }
102          ++fCount;
103      } // adds aElement at front
104
105      Iterator begin() const
106      {
107          return Iterator(fRoot).begin();
108      } // return a forward iterator
109
110      Iterator end() const
111      {
112          return Iterator(fRoot).end();
113      } // return a forward end iterator
114
115      Iterator rbegin() const
116      {
117          return Iterator(fRoot).rbegin();
118      } // return a backwards iterator
119
120      Iterator rend() const
121      {
122          return Iterator(fRoot).rend();
123      } // return a backwards end iterator
124
125      // P2
126
127      void push_back(const T& aElement)
128      {
129          if (empty())
130          {
131              fRoot = new Node(aElement);
132          }
133          else
134          {
135              Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
136              lastNode->push_back(*new Node(aElement));
137          }
138          ++fCount;
139      } // adds aElement at back
140
141      // P3
142
143      const T& operator[](size_t aIndex) const
144      {
145          if (aIndex > size() - 1)
146          {
147              throw std::out_of_range("Index out of bounds");
```

```
148         }
149         Iterator lIterator = Iterator(fRoot).begin();
150         for (size_t i = 0; i < aIndex; i++)
151         {
152             ++lIterator;
153         }
154         return *lIterator;
155     } // list indexer
156 
157     // P4
158 
159     List(const List& aOtherList): fRoot(nullptr), fCount(0)
160     {
161         *this = aOtherList;
162     } // copy constructor
163 
164     List& operator=(const List& aOtherList)
165     {
166         if (&aOtherList != this)
167         {
168             this->~List();
169             if (aOtherList.fRoot == nullptr)
170             {
171                 fRoot = nullptr;
172             }
173             else
174             {
175                 fRoot = nullptr;
176                 fCount = 0;
177                 for (auto& payload : aOtherList)
178                 {
179                     push_back(payload);
180                 }
181             }
182         }
183         return *this;
184     } // assignment operator
185 
186     // P5
187 
188     // move features
189     List(List&& aOtherList): fRoot(nullptr), fCount(0)
190     {
191         *this = std::move(aOtherList);
192     } // move constructor
193 
194     List& operator=(List&& aOtherList)
195     {
196         if (&aOtherList != this)
```

```
197         {
198             this->~List();
199             if (aOtherList.fRoot == nullptr)
200             {
201                 fRoot = nullptr;
202             }
203             else
204             {
205                 fRoot = aOtherList.fRoot;
206                 fCount = aOtherList.fCount;
207                 aOtherList.fRoot = nullptr;
208                 aOtherList.fCount = 0;
209             }
210         }
211         return *this;
212     } // move assignment operator
213 
214     void push_front(T&& aElement)
215     {
216         if (empty())
217         {
218             fRoot = new Node(std::move(aElement));
219         }
220         else
221         {
222             Node* lNode = new Node(std::move(aElement));
223             fRoot->push_front(*lNode);
224             fRoot = lNode;
225         }
226         ++fCount;
227     } // adds aElement at front
228 
229     void push_back(T&& aElement)
230     {
231         if (empty())
232         {
233             fRoot = new Node(aElement);
234         }
235         else
236         {
237             Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
238             lastNode->push_back(*new Node(aElement));
239         }
240         ++fCount;
241     } // adds aElement at back
242 };
243 
```