

Swinburne University of Technology
Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 3, List ADT
Due date: May 12, 2022, 14:30
Lecturer: Dr. Markus Lumpe

Your name: Jamie Kozminska **Your student id:** 101114436

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30
					X						

Marker's comments:

Problem	Marks	Obtained
1	48	
2	28	
3	26	
4	30	
5	42	
Total	174	

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener: _____

```
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() // ➤
26     {
27         destructor - frees all nodes
28
29         while ( fRoot != nullptr )
30         {
31             if ( fRoot != &fRoot->getPrevious() ) // ➤
32                 more than one element
33             {
34                 Node* lTemp = const_cast<Node*>(&fRoot->getPrevious()); // ➤
35                 select last
36
37                 lTemp->isolate(); // ➤
38                 remove from list
39                 delete lTemp; // ➤
40                 free
41             }
42             else
43             {
44                 delete fRoot; // ➤
45                 free last
46                 break; // ➤
47                 stop loop
48             }
49         }
50     }
51 }
52 }
```

```
43
44     void remove( const T& aElement )           //  ↗
45         remove first match from list
46     {
47         Node* lNode = fRoot;                   //  ↗
48         start at first
49
50         while ( lNode != nullptr )             //  Are ↗
51             there still nodes available?
52         {
53             if ( **lNode == aElement )         //  ↗
54                 Have we found the node?
55             {
56                 break;                         //  ↗
57                 stop the search
58             }
59
60             if ( lNode != &fRoot->getPrevious() ) //  not ↗
61                 reached last
62             {
63                 lNode = const_cast<Node*>(&lNode->getNext()); //  go ↗
64                 to next
65             }
66             else
67             {
68                 lNode = nullptr;               //  ↗
69                 stop search
70             }
71         }
72
73         // At this point we have either reached the end or found the node.
74         if ( lNode != nullptr )                 //  We ↗
75             have found the node.
76         {
77             if ( fCount != 1 )                 //  not ↗
78                 the last element
79             {
80                 if ( lNode == fRoot )
81                 {
82                     fRoot = const_cast<Node*>(&fRoot->getNext()); //  ↗
83                     make next root
84                 }
85             }
86             else
87             {
88                 fRoot = nullptr;              //  ↗
89                 list becomes empty
90             }
91         }
92     }
```

```
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 = 
209             aOtherList.begin())
210         {
211             auto val = *iter;
212             this->push_back(std::move(val));
213             aOtherList.remove(*iter);
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
```