

**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:** \_\_\_\_\_ **Your student id:** \_\_\_\_\_

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

---

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, 2024
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             {
33                 // more than one element
34
35                 Node* lTemp = const_cast<Node*>(&fRoot->getPrevious()); // ➤
36                 // select last
37
38                 lTemp->isolate(); // ➤
39                 // remove from list
40
41                 delete lTemp; // ➤
42                 // free
43             }
44             else
45             {
46                 delete fRoot; // ➤
47                 // free last
48
49                 break; // ➤
50                 // stop loop
51             }
52         }
53     }
54 }
```

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

```
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 // default constructor
93 List() :
94     fRoot(nullptr),
95     fCount(0)
96 { }
97
98 // Is list empty?
99 bool empty() const
100 {
101     return fRoot == nullptr;
102 }
103
104 // list size
105 size_t size() const
106 {
107     return fCount;
108 }
109
110 // adds aElement at front
111 void push_front(const T& aElement)
112 {
113     // allocates a new list node on the heap
114     Node* lNode = new Node(aElement);
115
116     // If the list is not currently empty
117     // puts the new node in front of the existing root node
118     if (!empty())
119     {
120         fRoot->push_front(*lNode);
121     }
122
123     // sets new root node
124     fRoot = lNode;
125     // increments count
```

```
126         fCount++;
127     }
128
129     // return a forward iterator
130     Iterator begin() const
131     {
132         // Default iterator is at begin position
133         return Iterator(fRoot);
134     }
135
136     // return a forward end iterator
137     Iterator end() const
138     {
139         return Iterator(fRoot).end();
140     }
141
142     // return a backwards iterator
143     Iterator rbegin() const
144     {
145         return Iterator(fRoot).rbegin();
146     }
147
148     // return a backwards end iterator
149     Iterator rend() const
150     {
151         return Iterator(fRoot).rend();
152     }
153
154     // P2
155
156     // adds aElement at back
157     // push_back is similar to push_front in node order themselves,
158     // except push_back keeps the old root node (if exists) of the List object ➤
159     void push_back(const T& aElement)
160     {
161         // gets the root node address
162         Node* lRoot = fRoot;
163         // pushes the new node to the start of the list
164         push_front(aElement);
165
166         // If there exists a root node previously
167         // re-assigns the root node
168         if (lRoot != nullptr)
169         {
170             fRoot = lRoot;
171         }
172     }
173
```

```
174 // P3
175
176 // list indexer
177 const T& operator[](size_t aIndex) const
178 {
179     // throws exception if index is out of bounds
180     if (aIndex >= fCount)
181     {
182         throw std::out_of_range("Index out of bounds.");
183     }
184
185     // starts searching from the root node
186     // works with address so that
187     // neither the returned value go out of scope (if use Node)
188     // nor the node get changed when iterating through the list (if use Node&)
189     const Node* lNode = fRoot;
190
191     for ( ; aIndex > 0; aIndex--)
192     {
193         // gets the address of the next node
194         lNode = &(lNode->getNext());
195     }
196
197     // returns the payload of the desired node
198     return **lNode;
199 }
200
201 // P4
202
203 // copy constructor
204 List(const List& aOtherList) :
205     fRoot(nullptr),
206     fCount(0)
207 {
208     *this = aOtherList;
209 }
210
211 // assignment operator
212 List& operator=(const List& aOtherList)
213 {
214     // protection against accidental suicide
215     if (&aOtherList != this)
216     {
217         // releases all old resources
218         this->~List();
219         // copies the count value
220         fCount = aOtherList.fCount;
221     }
```

```
222         // appends each element of the other list to this list (deep copy)
223         for (const T& element : aOtherList)
224         {
225             push_back(element);
226         }
227     }
228
229     return *this;
230 }
231
232 // P5
233
234 // move constructor
235 List(List&& aOtherList) noexcept :
236     fRoot(nullptr),
237     fCount(0)
238 {
239     *this = std::move(aOtherList);
240 }
241
242 // move assignment operator
243 List& operator=(List&& aOtherList) noexcept
244 {
245     // protection against accidental suicide
246     if (&aOtherList != this)
247     {
248         // releases all old resources
249         this->~List();
250
251         // sets object members to the other list's members
252         // No need to use std::move() for primitive types and pointers
253         fRoot = aOtherList.fRoot;
254         fCount = aOtherList.fCount;
255
256         // empties the old list by resetting its members
257         aOtherList.fRoot = nullptr;
258         aOtherList.fCount = 0;
259     }
260
261     return *this;
262 }
263
264 // move push_front
265 void push_front(T&& aElement)
266 {
267     // calls Node's constructor for r-value
268     Node* lNode = new Node(std::move(aElement));
269 }
```

```
270         // The rest looks like the reference-based overload
271         if (!empty())
272         {
273             fRoot->push_front(*lNode);
274         }
275
276         fRoot = lNode;
277         fCount++;
278     }
279
280     // move push_back
281     void push_back(T&& aElement)
282     {
283         Node* lRoot = fRoot;
284         // calls the overload for r-value
285         push_front(std::move(aElement));
286
287         if (lRoot != nullptr)
288         {
289             fRoot = lRoot;
290         }
291     }
292 };
293
294
```