# 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:** Nguyen Minh Duy **Your student id:** 104974743

| 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, 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      } // remove 1st match
85
86      ////////////////////////////////////////////////////////////////
87      //// PS3
88      ////////////////////////////////////////////////////////////////
89
90      // P1
91
92      List() : fRoot(nullptr), fCount(0) {}
93                                      // default constructor
94      bool empty() const                                                  ↵
                                                                  // Is list   ↵
          empty?
95      {
96          return fRoot == nullptr;                                        ↵

97      }            // Is list empty?
98
99      size_t size() const {
100         return fCount;
101     }               // list size
102
103     // Thêm phần tử vào đầu danh sách
104     void push_front(const T& aElement) {
105         Node* newNode = new Node(aElement);
106
107         if (fRoot == nullptr) {
108             fRoot = newNode;
109         }
110         else
111         {
112             fRoot->push_front(*newNode); // trỏ về trước đó ( tạo nút )
113             fRoot = newNode; // dời nút
114         }
115         ++fCount;
116     }
117
118     Iterator begin() const
119     {
120         return Iterator(fRoot);
121     }       // return a forward iterator
122     Iterator end() const
```

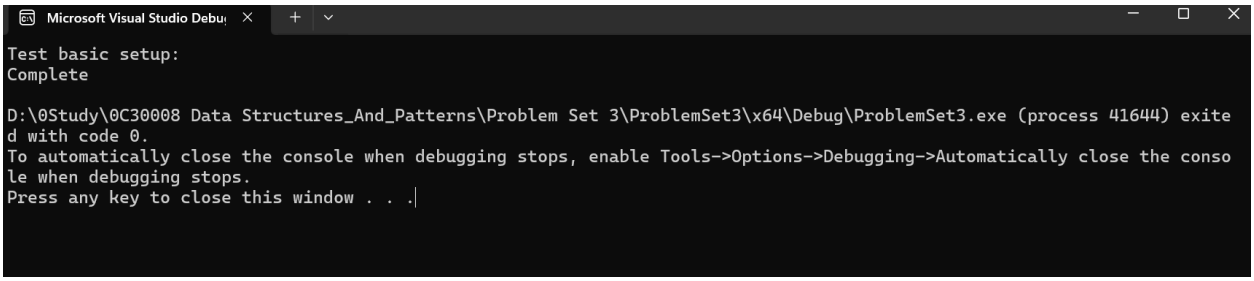
```
123     {
124         return begin().end();
125     }            // return a forward end iterator
126     Iterator rbegin() const
127     {
128         return begin().rbegin();
129     }                // return a backwards iterator
130     Iterator rend() const
131     {
132         return begin().rend();
133     }                    // return a backwards end iterator
134
135     // P2
136
137     void push_back(const T&                                                 ↪
          aElement)                                          // adds aElement  ↪
          at back
138     {
139         Node* newNode = new Node(aElement);
140
141         if (fRoot == nullptr)
142         {
143             fRoot = newNode;
144         }
145         else
146         {
147             fRoot->push_front(*newNode);
148         }
149
150         fCount++;                                                           ↪

151     }      // adds aElement at back
152
153     // P3
154
155     const T& operator[](size_t aIndex) const {
156         if (aIndex < fCount)                                                ↪

157         {
158             const Node* lNode = fRoot;
159             while (aIndex)
160             {
161                 // No
162                 aIndex--;                                                   ↪

163                 lNode = &lNode->getNext();                                  ↪

164             }
165             // Yes
```

```
166             return **lNode;

167         }
168         else
169         {
170             throw std::out_of_range("Index out of bounds.");

171         }
172     }   // list indexer
173
174     // P4
175
176     List(const List&
          aOtherList) :                                    // copy
          constructor
177         fRoot(nullptr),

178         fCount(0)

179     {
180         *this = aOtherList;
181     }
182
183     List& operator=(const List&
          aOtherList)                                // assignment operator
184     {
185         if (this != &aOtherList)

186         {
187             // delete
188             this->~List();

189
190             fRoot = nullptr;

191             fCount = 0;

192
193             // copy
194             for (const auto& e : aOtherList)
195             {
196                 push_back(e);
197             }

198         }
199
200         return *this;
201     }
202
```

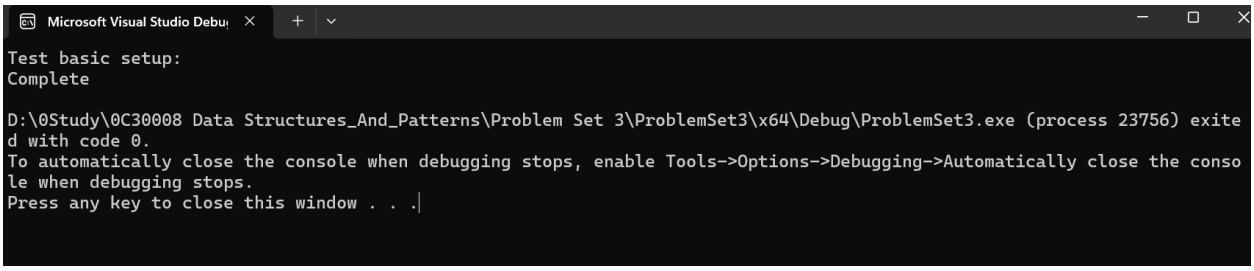
```
203     // P5
204
205     // move features
206
207     List(List&&
          aOtherList) :                                          //
          move constructor
208       fRoot(nullptr),

209       fCount(0)

210     {
211       *this = std::move(aOtherList);

212     }
213
214     List& operator=(List&&
          aOtherList)                                  // move
          assignment operator
215     {
216       if (this != &aOtherList)

217       {
218         // delete
219         this->~List();

220
221         // move (steal memory)
222         fRoot = aOtherList.fRoot;

223         fCount = aOtherList.fCount;

224         aOtherList.fRoot = nullptr;

225         aOtherList.fCount = 0;

226       }
227
228       return *this;

229     }
230
231     void push_front(T&& aElement)
232     {
233       Node* lNewElement = new Node(std::move(aElement));

234
235       if (fRoot == nullptr)
236       {
```

```
237             fRoot = lNewElement;
238         }
239         else
240         {
241             fRoot->push_front(*lNewElement);
242             fRoot = lNewElement;
243         }
244
245         fCount++;
246     }
247
248     void push_back(T&& aElement)
249     {
250         Node* lNewElement = new Node(std::move(aElement));

251
252         if (fRoot == nullptr)
253         {
254             fRoot = lNewElement;
255         }
256         else
257         {
258             fRoot->push_front(*lNewElement);
259         }
260
261         fCount++;
262     }
263
264 };
265
266
267
```

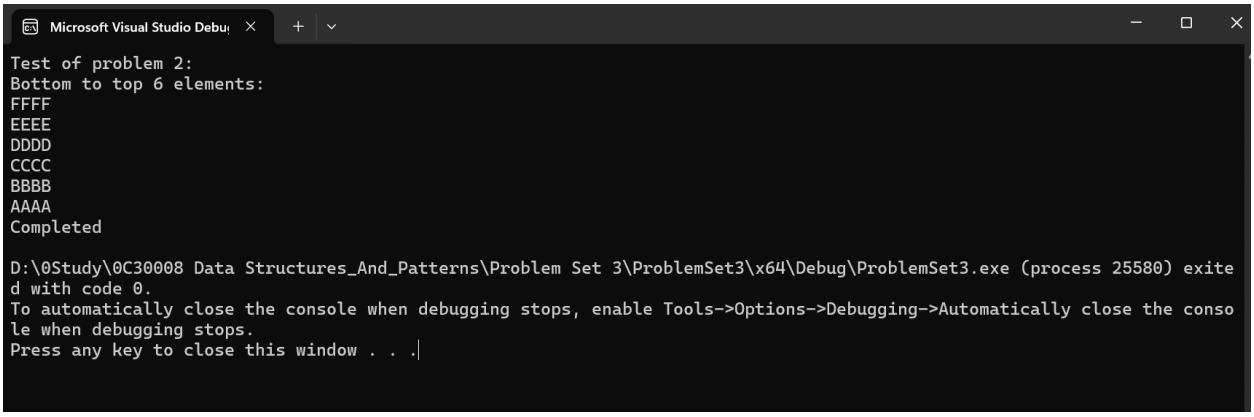
# Problem 0 Output:



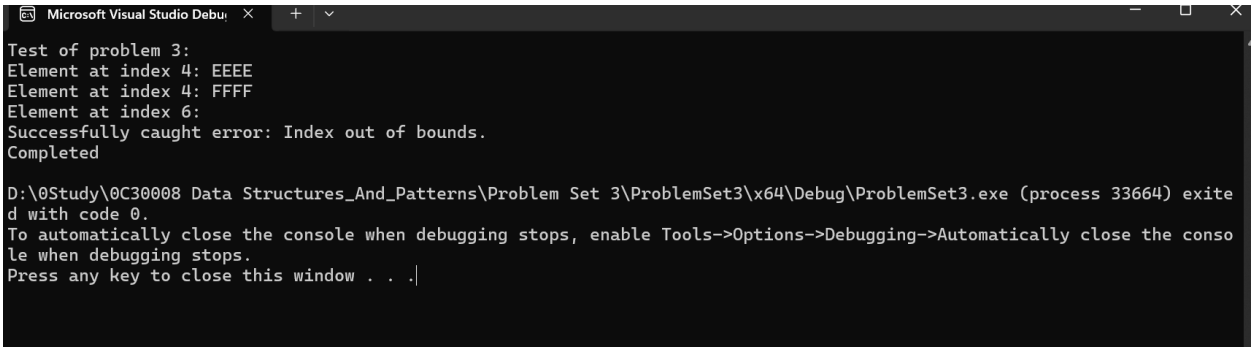

# Problem 1 Output:



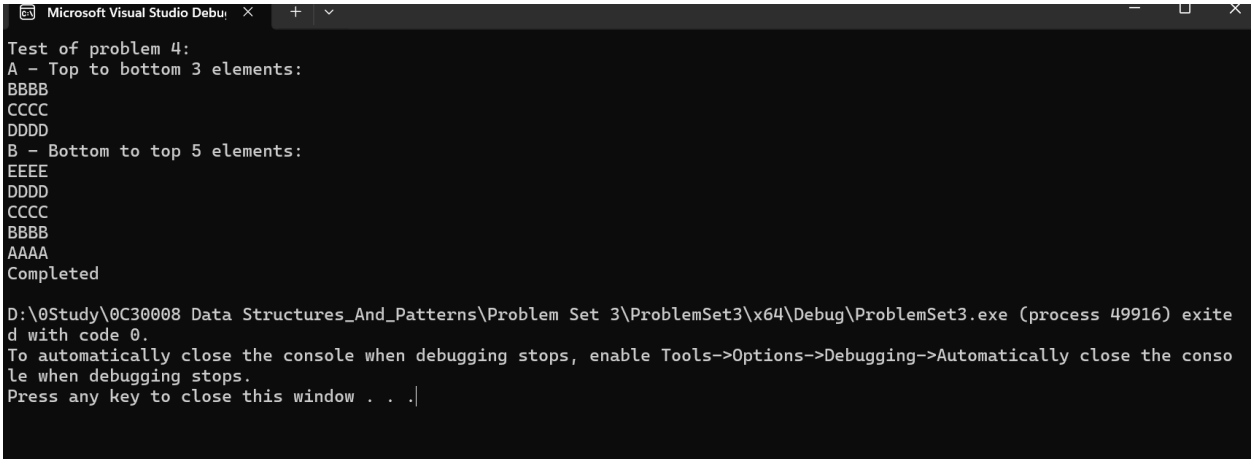

# Problem 2 Output:



# Problem 3 Output:




# Problem 4 Output:




# Problem 5 Output: