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: Nov 10, 2023, 23:59
Lecturer: Dr. James Jackson

Your name: Nguyen Ngoc Phuc **Your student id:** 104180294

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	Thu 08:00
											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, Doubly-linked Nodes, Tutorial 9, 2022
3
4  #pragma once
5
6  template<typename T>
7  class DoublyLinkedList
8  {
9  private:
10
11     T fPayload;
12     DoublyLinkedList* fNext;
13     DoublyLinkedList* fPrevious;
14
15 public:
16
17     // l-value constructor
18     explicit DoublyLinkedList( const T& aPayload ) :
19         fPayload(aPayload),
20         fNext(this),
21         fPrevious(this)
22     {}
23
24     // r-value constructor
25     explicit DoublyLinkedList( T&& aPayload ) :
26         fPayload(std::move(aPayload)),
27         fNext(this),
28         fPrevious(this)
29     {}
30
31     DoublyLinkedList& push_front( DoublyLinkedList& aNode )
32     {
33         aNode.fNext = this;                    // make this the forward
              pointer of aNode
34
35         aNode.fPrevious = fPrevious;           // make this's backward
              pointer aNode's
36         fPrevious->fNext = &aNode;             // tie back to Node
37
38         fPrevious = &aNode;                    // this' backward pointer
              becomes aNode
39
40         return aNode;                          // last node inserted
41     }
42
43     DoublyLinkedList& push_back( DoublyLinkedList& aNode )
44     {
45         aNode.fPrevious = this;                // make this the backwards
              pointer of aNode
```

```
46
47          aNode.fNext = fNext;                    // make this's forward
              pointer aNode's
48          fNext->fPrevious = &aNode;              // tie back to Node
49
50          fNext = &aNode;                         // this' forward pointer
              becomes aNode
51
52          return aNode;                           // last node inserted
53      }
54
55      void isolate()
56      {
57          fPrevious->fNext = fNext;               // unlink previous
58          fNext->fPrevious = fPrevious;           // unlink next
59
60          fPrevious = this;                       // isolate this node
61          fNext = this;
62      }
63
64      void swap( DoublyLinkedList& aNode )
65      {
66          std::swap( fPayload, aNode.fPayload );  // exchange list elements
67      }
68
69      const T& operator*() const                  // dereference operator
70      {
71          return getPayload();
72      }
73
74      const T& getPayload() const
75      {
76          return fPayload;
77      }
78
79      const DoublyLinkedList& getNext() const
80      {
81          return *fNext;
82      }
83
84      const DoublyLinkedList& getPrevious() const
85      {
86          return *fPrevious;
87      }
88  };
89
```

```
 1
 2  // COS30008, Doubly-linked Nodes, Tutorial 9, 2022
 3
 4  #pragma once
 5
 6  #include "DoublyLinkedList.h"
 7
 8  template<typename T>
 9  class DoublyLinkedListIterator
10  {
11  private:
12      enum class States { BEFORE, DATA , AFTER };
13
14      using Node = DoublyLinkedList<T>;
15
16      const Node* fRoot;
17
18      States fState;
19      const Node* fCurrent;
20
21  public:
22
23      using Iterator = DoublyLinkedListIterator<T>;
24
25      DoublyLinkedListIterator( const Node* aRoot )
26      {
27          fRoot = aRoot;
28          fCurrent = fRoot;
29
30          if ( fCurrent != nullptr )
31          {
32              fState = States::DATA;
33          }
34          else
35          {
36              // empty doubly linked list of nodes
37              fState = States::AFTER;
38          }
39      }
40
41      const T& operator*() const          // dereference
42      {
43          return **fCurrent;
44      }
45
46      Iterator& operator++()              // prefix increment
47      {
48          switch ( fState )
49          {
```

```
50              case States::BEFORE:
51
52                  fCurrent = fRoot;   // set to first element
53
54                  if ( fCurrent == nullptr )
55                  {
56                      fState = States::AFTER;
57                  }
58                  else
59                  {
60                      fState = States::DATA;
61                  }
62
63                  break;
64
65              case States::DATA:
66
67                  // Is current previous of root (last element forward)?
68                  // Current cannot be nullptr as we are in state DATA.
69                  if ( fCurrent == &fRoot->getPrevious() )
70                  {
71                      // Yes, we are done
72                      fCurrent = nullptr;
73                      fState = States::AFTER;
74                  }
75                  else
76                  {
77                      // No, we can advance
78                      fCurrent = &fCurrent->getNext();
79                  }
80
81                  break;
82
83              default:
84
85                  break;
86          }
87
88          return *this;
89      }
90
91      Iterator operator++(int)            // postfix increment
92      {
93          Iterator temp = *this;
94
95          ++(*this);
96
97          return temp;
98      }
```

```
 99
100     Iterator& operator--()              // prefix decrement
101     {
102         switch ( fState )
103         {
104             case States::AFTER:
105
106                 fCurrent = fRoot;
107
108                 if ( fCurrent == nullptr )
109                 {
110                     fState = States::BEFORE;
111                 }
112                 else
113                 {
114                     fCurrent = &fCurrent->getPrevious();    // set to last ↵
                        element
115                     fState = States::DATA;
116                 }
117
118                 break;
119
120             case States::DATA:
121
122                 // Is current root (last element backwards)?
123                 // Current cannot be nullptr as we are in state DATA.
124
125                 if ( fCurrent == fRoot )
126                 {
127                     // Yes, we are done
128                     fCurrent = nullptr;
129                     fState = States::BEFORE;
130                 }
131                 else
132                 {
133                     // No, we can advance
134                     fCurrent = &fCurrent->getPrevious();
135                 }
136
137                 break;
138
139             default:
140
141                 break;
142         }
143
144         return *this;
145     }
146
```

```
147     Iterator operator--(int)            // postfix decrement
148     {
149         Iterator temp = *this;
150
151         --(*this);
152
153         return temp;
154     }
155
156     bool operator==( const Iterator& aOtherIter ) const
157     {
158         return
159             fRoot == aOtherIter.fRoot &&
160             fCurrent == aOtherIter.fCurrent &&
161             fState == aOtherIter.fState;
162     }
163
164     bool operator!=( const Iterator& aOtherIter ) const
165     {
166         return !(*this == aOtherIter);
167     }
168
169     Iterator begin() const
170     {
171         return ++(rend());
172     }
173
174     Iterator end() const
175     {
176         Iterator iter = *this;
177
178         iter.fCurrent = nullptr;
179         iter.fState = States::AFTER;
180
181         return iter;
182     }
183
184     Iterator rbegin() const
185     {
186         return --(end());
187     }
188
189     Iterator rend() const
190     {
191         Iterator iter = *this;
192
193         iter.fCurrent = nullptr;
194         iter.fState = States::BEFORE;
195
```

```
196         return iter;
197     }
198 };
199
```

```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() : fRoot(nullptr), fCount(0) {} // default constructor
    List(const List& aOtherList) : fRoot(nullptr), fCount(0) // copy constructor
    {
        *this = aOtherList;
    }

    List& operator=(const List& aOtherList) // assignment operator
    {
        if (&aOtherList != this)
        {
            this->~List();
            if (aOtherList.fRoot == nullptr)
            {
                fRoot = nullptr;
            }
            else
            {
                fRoot = nullptr;
                fCount = 0;
                for (auto& payload : aOtherList)
                {
                    push_back(payload);
                }
            }
        }
        return *this;
    }
    ~List() // destructor - frees all nodes
```

```
49      {
50          while (fRoot != nullptr)
51          {
52              if (fRoot != &fRoot->getPrevious())
53              {
54                  Node* lTemp = const_cast<Node*>(&fRoot->getPrevious());
55                  lTemp->isolate();
56                  delete lTemp;
57              }
58              else
59              {
60                  delete fRoot;
61                  break;
62              }
63          }
64      }
65
66      bool isEmpty() const // Is list empty?
67      {
68          return fRoot == nullptr;
69      }
70      size_t size() const // list size
71      {
72          return fCount;
73      }
74
75      void push_front(const T& aElement) // adds aElement at front
76      {
77          if (isEmpty())
78          {
79              fRoot = new Node(aElement);
80          }
81          else
82          {
83              Node* lNode = new Node(aElement);
84              fRoot->push_front(*lNode);
85              fRoot = lNode;
86          }
87          ++fCount;
88      }
89      void push_back(const T& aElement) // adds aElement at back
90      {
91          if (isEmpty())
92          {
93              fRoot = new Node(aElement);
94          }
95          else
96          {
97              Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
```

```
 98              lastNode->push_back(*new Node(aElement));
 99          }
100          ++fCount;
101      }
102      void remove(const T& aElement) // remove first match from list
103      {
104          Node* lNode = fRoot;
105          while (lNode != nullptr)
106          {
107              if (**lNode == aElement)
108              {
109                  break;
110              }
111              else
112              {
113                  if (lNode != &fRoot->getPrevious())
114                  {
115                      lNode = const_cast<Node*>(&lNode->getNext());
116                  }
117                  else
118                  {
119                      lNode = nullptr;
120                  }
121              }
122          }
123          if (lNode != nullptr)
124          {
125              if (fCount != 1)
126              {
127                  if (lNode == fRoot)
128                  {
129                      fRoot = const_cast<Node*>(&fRoot->getNext());
130                  }
131              }
132              else
133              {
134                  fRoot = nullptr;
135              }
136              lNode->isolate();
137              delete lNode;
138              fCount--;
139          }
140      }
141
142      const T& operator[](size_t aIndex) const // list indexer
143      {
144          if (aIndex >= size())
145          {
146              throw std::out_of_range("Index out of bounds");
```

```
147         }
148
149         Iterator lIterator = begin();
150         for (size_t i = 0; i < aIndex; i++)
151         {
152             ++lIterator;
153         }
154
155         return *lIterator;
156     }
157
158     Iterator begin() const // return a forward iterator
159     {
160         return Iterator(fRoot).begin();
161     }
162     Iterator end() const // return a forward end iterator
163     {
164         return Iterator(fRoot).end();
165     }
166     Iterator rbegin() const // return a backwards iterator
167     {
168         return Iterator(fRoot).rbegin();
169     }
170     Iterator rend() const // return a backwards end iterator
171     {
172         return Iterator(fRoot).rend();
173     }
174
175     // move features
176     List(List&& aOtherList) // move constructor
177     {
178         *this = std::move(aOtherList);
179     }
180     List& operator=(List&& aOtherList) // move assignment operator
181     {
182         if (&aOtherList != this)
183         {
184             this->~List();
185             if (aOtherList.fRoot == nullptr)
186             {
187                 fRoot = nullptr;
188             }
189             else
190             {
191                 fRoot = aOtherList.fRoot;
192                 fCount = aOtherList.fCount;
193                 aOtherList.fRoot = nullptr;
194                 aOtherList.fCount = 0;
195             }
```

```
196         }
197         return *this;
198     }
199     void push_front(T&& aElement) // adds aElement at front
200     {
201         if (isEmpty())
202         {
203             fRoot = new Node(std::move(aElement));
204         }
205         else
206         {
207             Node* lNode = new Node(std::move(aElement));
208             fRoot->push_front(*lNode);
209             fRoot = lNode;
210         }
211         ++fCount;
212     }
213     void push_back(T&& aElement) // adds aElement at back
214     {
215         if (isEmpty())
216         {
217             fRoot = new Node(aElement);
218         }
219         else
220         {
221             Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
222             lastNode->push_back(*new Node(aElement));
223         }
224         ++fCount;
225     }
226 };
```

```cpp

// COS30008, Problem Set 3, 2022

#include <iostream>
#include <string>
#include <stdexcept>

#include "ListPS3.h"

using namespace std;

#define P0
//#define P1
//#define P2
//#define P3
//#define P4
//#define P5

#ifdef P0

void testP0()
{
    cout << "Test basic setup:" << endl;

    List<string> lList;

    lList.remove( "P0" );
    lList.remove( string( "P0" ) );

    cout << "Complete" << endl;
}

#endif

#ifdef P1

void testP1()
{
    using StringList = List<string>;

    string s1( "AAAA" );
    string s2( "BBBB" );
    string s3( "CCCC" );
    string s4( "DDDD" );

    cout << "Test of problem 1:" << endl;

    StringList lList;

```

```
50      if ( !lList.isEmpty() )
51      {
52          cerr << "Error: Newly created list is not empty." << endl;
53      }
54
55      lList.push_front( s4 );
56      lList.push_front( s3 );
57      lList.push_front( s2 );
58      lList.push_front( s1 );
59
60      // iterate from the top
61      cout << "Top to bottom " << lList.size() << " elements:" << endl;
62      for ( const string& element : lList )
63      {
64        cout << element << endl;
65      }
66
67      // iterate from the end
68      cout << "Bottom to top " << lList.size() << " elements:" << endl;
69      for ( StringList::Iterator iter = lList.rbegin(); iter != iter.rend(); ↵
           iter-- )
70      {
71        cout << *iter << endl;
72      }
73
74      cout << "Completed" << endl;
75  }
76
77  #endif
78
79  #ifdef P2
80
81  void testP2()
82  {
83      using StringList = List<string>;
84
85      string s1( "AAAA" );
86      string s2( "BBBB" );
87      string s3( "CCCC" );
88      string s4( "DDDD" );
89      string s5( "EEEE" );
90      string s6( "FFFF" );
91
92      cout << "Test of problem 2:" << endl;
93
94      StringList lList;
95
96      lList.push_front( s4 );
97      lList.push_front( s3 );
```

```
 98      lList.push_front( s2 );
 99      lList.push_front( s1 );
100      lList.push_back( s5 );
101      lList.push_back( s6 );
102
103      // iterate from the top
104      cout << "Bottom to top " << lList.size() << " elements:" << endl;
105      for ( StringList::Iterator iter = lList.rbegin(); iter != iter.rend(); ↵
            iter-- )
106      {
107        cout << *iter << endl;
108      }
109
110      cout << "Completed" << endl;
111  }
112
113  #endif
114
115  #ifdef P3
116
117  void testP3()
118  {
119      using StringList = List<string>;
120
121      string s1( "AAAA" );
122      string s2( "BBBB" );
123      string s3( "CCCC" );
124      string s4( "DDDD" );
125      string s5( "EEEE" );
126      string s6( "FFFF" );
127
128      StringList lList;
129
130      lList.push_front( s4 );
131      lList.push_front( s3 );
132      lList.push_front( s2 );
133      lList.push_front( s1 );
134      lList.push_back( s5 );
135      lList.push_back( s6 );
136
137      cout << "Test of problem 3:" << endl;
138
139      try
140      {
141          cout << "Element at index 4: " << lList[4] << endl;
142          lList.remove( s5 );
143          cout << "Element at index 4: " << lList[4] << endl;
144
145          cout << "Element at index 6: ";
```

```
146         cout << lList[6] << endl;
147         cout << "Error: You should not see this text." << endl;
148     }
149     catch (out_of_range e)
150     {
151         cerr << "\nSuccessfully caught error: " << e.what() << endl;
152     }
153
154     cout << "Completed" << endl;
155 }
156
157 #endif
158
159 #ifdef P4
160
161 void testP4()
162 {
163     using StringList = List<string>;
164
165     string s1( "AAAA" );
166     string s2( "BBBB" );
167     string s3( "CCCC" );
168     string s4( "DDDD" );
169     string s5( "EEEE" );
170
171     List<string> lList;
172
173     cout << "Test of problem 4:" << endl;
174
175     lList.push_front( s4 );
176     lList.push_front( s3 );
177     lList.push_front( s2 );
178
179     List<string> copy( lList );
180
181     // iterate from the top
182     cout << "A - Top to bottom " << copy.size() << " elements:" << endl;
183
184     for ( const string& element : copy )
185     {
186         cout << element << endl;
187     }
188
189     // override list
190     lList = copy;
191
192     lList.push_front( s1 );
193     lList.push_back( s5 );
194
```

```
195     // iterate from the top
196     cout << "B - Bottom to top " << lList.size() << " elements:" << endl;
197
198     for ( auto iter = lList.rbegin(); iter != iter.rend(); iter-- )
199     {
200         cout << *iter << endl;
201     }
202
203     cout << "Completed" << endl;
204 }
205
206 #endif
207
208 #ifdef P5
209
210 void testP5()
211 {
212     using StringList = List<string>;
213
214     string s2( "CCCC" );
215
216     List<string> lList;
217
218     cout << "Test of problem 5:" << endl;
219
220     lList.push_front( string( "DDDD" ) );
221     lList.push_front( std::move(s2) );
222     lList.push_front( "BBBB" );
223
224     if ( s2.empty() )
225     {
226         cout << "Successfully performed move operation." << endl;
227     }
228     else
229     {
230         cerr << "Error: Move operation failed." << endl;
231     }
232
233     cout << "A - Top to bottom " << lList.size() << " elements:" << endl;
234
235     for ( const string& element : lList )
236     {
237         cout << element << endl;
238     }
239
240     List<string> move( std::move(lList) );
241
242     if ( lList.isEmpty() )
243     {
```

```
244         cout << "Successfully performed move operation." << endl;
245     }
246     else
247     {
248         cerr << "Error: Move operation failed." << endl;
249     }
250
251     // iterate from the top
252     cout << "B - Top to bottom " << move.size() << " elements:" << endl;
253
254     for ( const string& element : move )
255     {
256         cout << element << endl;
257     }
258
259     // override list
260     lList = std::move(move);
261
262     if ( move.isEmpty() )
263     {
264         cout << "Successfully performed move operation." << endl;
265     }
266     else
267     {
268         cerr << "Error: Move operation failed." << endl;
269     }
270
271     lList.push_front( "AAAA" );
272     lList.push_back( "EEEE" );
273
274     // iterate from the top
275     cout << "C - Bottom to top " << lList.size() << " elements:" << endl;
276
277     for ( auto iter = lList.rbegin(); iter != iter.rend(); iter-- )
278     {
279         cout << *iter << endl;
280     }
281
282     cout << "Completed" << endl;
283 }
284
285 #endif
286
287 int main()
288 {
289 #ifdef P0
290     testP0();
291 #endif
292
```

```
293 #ifdef P1
294     testP1();
295 #endif
296
297 #ifdef P2
298     testP2();
299 #endif
300
301 #ifdef P3
302     testP3();
303 #endif
304
305 #ifdef P4
306     testP4();
307 #endif
308
309 #ifdef P5
310     testP5();
311 #endif
312
313     return 0;
314 }
315
```