Swinburne University of Technology

Faculty of Science, Engineering and Technology

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	10, List ADT (part A)
Lecturer:	Dr. Markus Lumpe

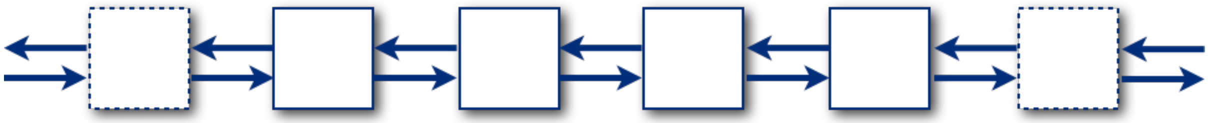

Figure 1: List as Abstract Data Type.

Review the solution of problem set 5, the `DoublyLinkedNode` template class developed in the tutorial 7, and the lecture material regarding the construction of an abstract data type.

Start with the header files provided on Canvas, as they have been fully tested.

Using the template classes `DoublyLinkedNode` and `DoublyLinkedNodeIterator`, implement the template class `List` as specified below:

```
#pragma once

#include "DoublyLinkedNode.h"
#include "DoublyLinkedNodeIterator.h"
#include <stdexcept>

template<class T>
class List
{
private:
  // auxiliary definition to simplify node usage
  typedef DoublyLinkedNode<T> Node;

  Node* fTop;        // the first element in the list
  Node* fLast;       // the last element in the list
  int fCount;        // number of elements in the list

public:
  // auxiliary definition to simplify iterator usage
  typedef DoublyLinkedNodeIterator<T> Iterator;

  List();                        // default constructor - creates empty list
  ~List();                       // destructor - frees all nodes

  bool isEmpty() const;          // Is list empty?
  int size() const;              // list size

  // adds a node initialized with aElement at front
  void push_front( const T& aElement );

  // adds a node initialized with aElement at back
  void push_back( const T& aElement );

  // removes node that matches aElement from list
  void remove( const T& aElement );

  const T& operator[]( size_t aIndex ) const;  // list indexer

  // returns an iterator for the nodes of the list
  Iterator begin() const;        // return a forward iterator
  Iterator end() const;          // return a forward end iterator
  Iterator rbegin() const;       // return a backwards iterator
  Iterator rend() const;         // return a backwards end iterator
};
```

The template class `List` defines an "object adapter" for `DoublyLinkedNode` objects (i.e., the list representation). Somebody else has already started with the implementation, but left the project unfinished. You find a header file for the incomplete `List` class on Canvas. This header file contains the specification of the template class List and the implementations for

- the destructor `~List()`
- the method `bool isEmpty() const`
- the method `int size() const`
- the method `void push_front( const T& aElement )`

You need to implement the remaining member functions. To facilitate this process, apply the following four-step approach:

1. Implement the default constructor and the iterator methods first. You can use the following test code for verification (you need to include `<string>` for the program to compile). The default constructor has to create an empty list. The iterator methods just return a corresponding value-based `DoublyLinkedNodeIterator` object properly initialized with the top list node.

```
#include <string>

using namespace std;

List<string> lList;

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

lList.push_front( s4 );
lList.push_front( s3 );
lList.push_front( s2 );
lList.push_front( s1 );

// iterate from the top
cout << "Top to bottom: " << lList.size() << " elements" << endl;
for ( const string& item : lList )
{
  cout << "A list element: " << item << endl;
}
```

   The result should look like this:

```
Top to bottom: 4 elements
A list element: AAAA
A list element: BBBB
A list element: CCCC
A list element: DDDD
```

2. Implement the method `push_back`, which is just a variant of method `push_front`.

```
string s5( "EEEE" );
string s6( "FFFF" );

lList.push_back( s5 );
lList.push_back( s6 );

// iterate from the top
cout << "Bottom to top: " << lList.size() << " elements" << endl;
for ( List<string>::Iterator iter = lList.rbegin();
                  iter != iter.rend(); iter-- )
{
      cout << "A list element: " << *iter << endl;
}
```

   The result should look like this:

```
Bottom to top: 6 elements
A list element: FFFF
A list element: EEEE
A list element: DDDD
```

```
A list element: CCCC
A list element: BBBB
A list element: AAAA
```

3. Implement the method `remove`. This method has to search for the node that matches `aElement`. If no such node exists, then the list remains unchanged. Otherwise, the corresponding node needs to be taken out of the list and its memory released. Please note that the identified node may coincide with the first or the last node in the list. These boundary conditions need to be addressed properly. Also, you need to use the cast operator `(Node*)` in this method to convert between `const Node*` and `Node*`. Remember that type `Node` is defined as `typedef DoublyLinkedNode<T> Node`.

```
lList.remove( s3 );
lList.remove( s1 );
lList.remove( s6 );

// iterate from the top
cout << "Top to bottom: " << lList.size() << " elements" << endl;
for ( List<string>::Iterator iter = lList.begin();
                  iter != iter.end(); iter++ )
{
      cout << "A list element: " << *iter << endl;
}
```

The result should look like this:

```
Top to bottom: 3 elements
A list element: BBBB
A list element: DDDD
A list element: EEEE
```

4. Implement **`operator[]`**. The indexer has to search for the element that corresponds to `aIndex`. Also, `aIndex` may be out of bounds. Hence the indexer has to throw a `range_error` exception. The implementation requires the cast operator `(Node*)` to convert between `const Node*` and `Node*`.

```
cout << "Element at index 2: " << lList[2] << endl;
```

The result should look like this:

```
Element at index 2: EEEE
```

Every student needs to complete this tutorial. Check with the tutor.