Swinburne University Of Technology

Faculty of Information and Communication Technologies

ASSIGNMENT COVER SHEET

Subject Code: HIT3303
Subject Title: Data Structures & Patterns
Assignment number and title: 3 – List ADT
Due date: **April 28, 2009, 02:30 p.m., on paper**
Lecturer: Dr. Markus Lumpe

Your name:_____

Marker's comments:

Problem	Marks	Obtained
1	86	
Total	86	

Extension certification:

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

Signature of Convener:_____

Problem Set 3: List ADT

Preliminaries

Review the solution of problem set 2.

Problem 1:

Consider the following C++ class specification

```
template<class ElementType>
class List
{
private:

  template<class DataType>
  class Node
  {
   …
  };

  template<class DataType>
  class NodeIterator
  {
   …
  };

  Node<ElementType>* fTop;
  Node<ElementType>* fLast;

public:
  typedef NodeIterator<ElementType> ListIterator;

  List();
  ~List();

  void Add( const ElementType& aElement );
  void AddFirst( const ElementType& aElement );
  bool Delete( const ElementType& aElement );
  void DeleteFirst();
  void DeleteLast();

  const ElementType& operator[]( int aIndex ) const;
  ListIterator GetIterator() const;
};
```

The specification defines an interface for the abstract data type `List`. `List` is a template class that is parameterized over the list element type `ElementType`. We wish list to support the following operations:

- Construct an empty list.
- Destruct a list, that is, release any allocated resources.

- Add an element at the end of a list.
- Add and element at the top of a list.
- Delete a given element from a list (return true, if the element was a member of the list).
- Delete the first element of a list.
- Delete the last element of a list.
- Provide an indexer to access elements of the list using array semantics.
- Provide a bi-directional iterator to traverse the elements of the list either in forward or backwards manner.

A particular feature of the definition of `List` is that we would like to hide the implementation details from clients of `List`. For this reason we declare the template class `Node` and `NodeIterator` in the private section of `List`. `Node` and `NodeIterator` are placeholders for the template classes defined in problem set 2. So, we just need to incorporate their definitions into the definition of the template class `List`. In fact, the template class List now constitutes an Adapter for `Node` and exposes the required functionality using class `Node` as underlying implementation representation. Furthermore, `GetIterator()` is a Factory method that returns an iterator that is an instance of `NodeIterator`.

There is one complication, however. The class `List` requires access to the data members of class `Node`. We need, therefore, to change the visibility `Node`'s data members from `private` to `public`.

Please note that when using template classes both the specification and the method implementations need to be available. For this reason, you should define all template classes in header files. It is recommended, but not required, to define all methods within the class declaration (this is also possible in C++ and resembles the way classes are defined in Java or C#).

To define the template class `List`, the specifications for classes `Node` and `NodeIterarator` need to be included into the region of the class `List`. You can either copy the original specification (not recommended) or use the `#include` directive to accomplish this task. The latter technique will be demonstrated in the labs.

Implement class `List`.

Test sample 1:

```
void TestList1()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );

  List<string> l;

  l.Add( s1 );
  l.Add( s2 );
  l.Add( s3 );
  l.Add( s4 );

  cout << "Forward:" << endl;

  for ( List<string>::ListIterator iter = l.GetIterator();
                                    iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }

  cout << "Backward:" << endl;

  for ( List<string>::ListIterator iter = l.GetIterator().end();
                                          --iter != iter.begin(); )
  {
    cout << *iter << endl;
  }
}
```

Result:

```
Forward:
One
Two
Three
Four
Backward:
Four
Three
Two
One
```

Test sample 2:

```
void TestList2()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  List<string> l;

  l.Add( s1 );
  l.Add( s2 );
  l.Add( s3 );

  cout << "Tree elements:" << endl;

  for ( List<string>::ListIterator iter = l.GetIterator();
                                   iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }

  l.Delete( s2 );

  cout << "Two elements:" << endl;

  for ( List<string>::ListIterator iter = l.GetIterator();
                                   iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }
}
```

Result:

```
Tree elements:
One
Two
Three
Two elements:
One
Three
```

Test sample 3:

```
void TestList3()
{
  List<int> l;

  l.Add( 1 );
  l.Add( 2 );
  l.Add( 3 );
  l.Add( 4 );
  l.Add( 5 );
  l.Add( 6 );

  cout << "Six elements:" << endl;

  for ( List<int>::ListIterator iter = l.GetIterator();
                                    iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }

  l.DeleteFirst();

  cout << "First Deleted:" << endl;

  for ( List<int>::ListIterator iter = l.GetIterator();
                                    iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }

  l.DeleteLast();

  cout << "Last Deleted:" << endl;

  for ( List<int>::ListIterator iter = l.GetIterator();
                                    iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }

  l.AddFirst( 0 );

  cout << "First Added:" << endl;

  for ( List<int>::ListIterator iter = l.GetIterator();
                                    iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }
}
```

Result:

```
Six elements:
1
2
3
4
5
6
First Deleted:
2
3
4
5
6
Last Deleted:
2
3
4
5
First Added:
0
2
3
4
5
```

Submission deadline: Tuesday, April 28, 2009, 2:30 p.m.

Submission procedure: on paper.