Swinburne University Of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures & Patterns
Assignment number and title:	5 – Lists, List Iterator, and Design Patterns
Due date:	April 29, 2013, 10:30 a.m.
Lecturer:	Dr. Markus Lumpe

Your name:______________________________ **Your student id:**_________________

Check Tutorial	Fri 10:30	Fri 12:30	Fri 14:30

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 5: Lists, List Iterator, and Design Patterns

Problem 1:

Start with the `DoublyLinkedNode` template class developed in the tutorial in week 6. Define a bi-directional list iterator for doubly-linked lists that satisfies the following template class specification:

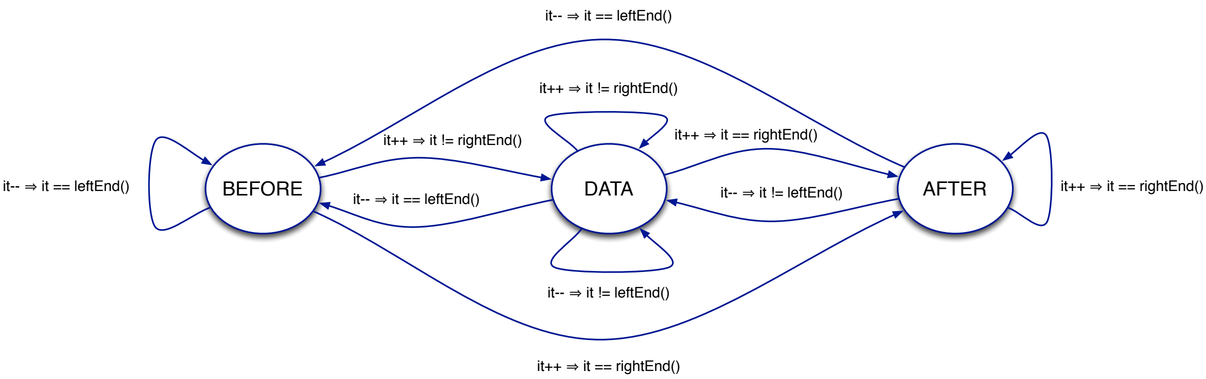
```
#pragma once

#include "DoublyLinkedNode.h"

template<class DataType>
class DoublyLinkedNodeIterator
{
private:
  enum IteratorStates { BEFORE, DATA , AFTER };

  IteratorStates fState;

  typedef DoublyLinkedNode<DataType> Node;

  const Node* fLeftmost;
  const Node* fRightmost;
  const Node* fCurrent;

public:
  typedef DoublyLinkedNodeIterator<DataType> Iterator;

  DoublyLinkedNodeIterator( const Node& aList );

  const DataType& operator*() const;        // dereference

  Iterator& operator++();                   // prefix increment
  Iterator operator++(int);                 // postfix increment
  Iterator& operator--();                   // prefix decrement
  Iterator operator--(int);                 // postfix decrement

  bool operator==( const Iterator& aOtherIter ) const;
  bool operator!=( const Iterator& aOtherIter ) const;

  Iterator leftEnd() const;
  Iterator first() const;
  Iterator last() const;
  Iterator rightEnd() const;
};
```

The bi-directional list iterator implements the standard operators for iterators: dereference to access the current element the iterator is positioned on, the increment operators advance the iterator to the next element, and the decrement operators take the iterator to the previous element. The list iterator also defines the equivalence predicates and the four factory methods: `leftEnd()`, `first()`, `last()`, and `rightEnd()`. The method `leftEnd()` returns a new list iterator positioned before the first element of the doubly-linked list, `first()` a new iterator positioned on the first element, `last()` a new iterator positioned on the last element, and `rightEnd()` returns a new iterator that is positioned after the last element of the doubly-linked list.

Implement the list iterator. Please note that the constructor of the list iterator has to properly set `fLeftmost`, `fRightmost`, and `fCurrent`. In particular, the constructor has to position the iterator on the first element of the list.

An iterator must not change the underlying collection. However, in the case of `DoublyLinkedNodeIterator` we need a special marker to denote, whether the iterator is "before" the first list element or "after" the last list element. Since we cannot change the underlying list, we need to add *state* to the iterator. Using the iterator state (i.e., `fState`) we can now clearly mark when the iterator is before the first element, within the first and the last element, or after the last element.

To guarantee to correct behavior of the `DoublyLinkedNodeIterator`, it must implement a *state machine* with three states: `BEFORE`, `DATA`, `AFTER`. The following state transition diagram illustrates, how `DoublyLinkedNodeIterator` works:



All increment and decrement operators have to test, whether the iterator is still positioned within the collection. In this case the current iterator is different from both `leftEnd()` and `rightEnd()`. If the iterator is positioned before the first element, then it is equivalent to `leftEnd()`. If the iterator is positioned past the last element, then it is equivalent to `rightEnd()`. Please note that the iterator can in one step become equivalent to `leftEnd()` or `rightEnd()`.

The labels in the state transition diagram describe the condition for state change. You will have to encode the corresponding tests as nested if-statements or a switch-statement. A given label has to be interpreted as follows. If the label reads

it-- ⇒ it == leftEnd()

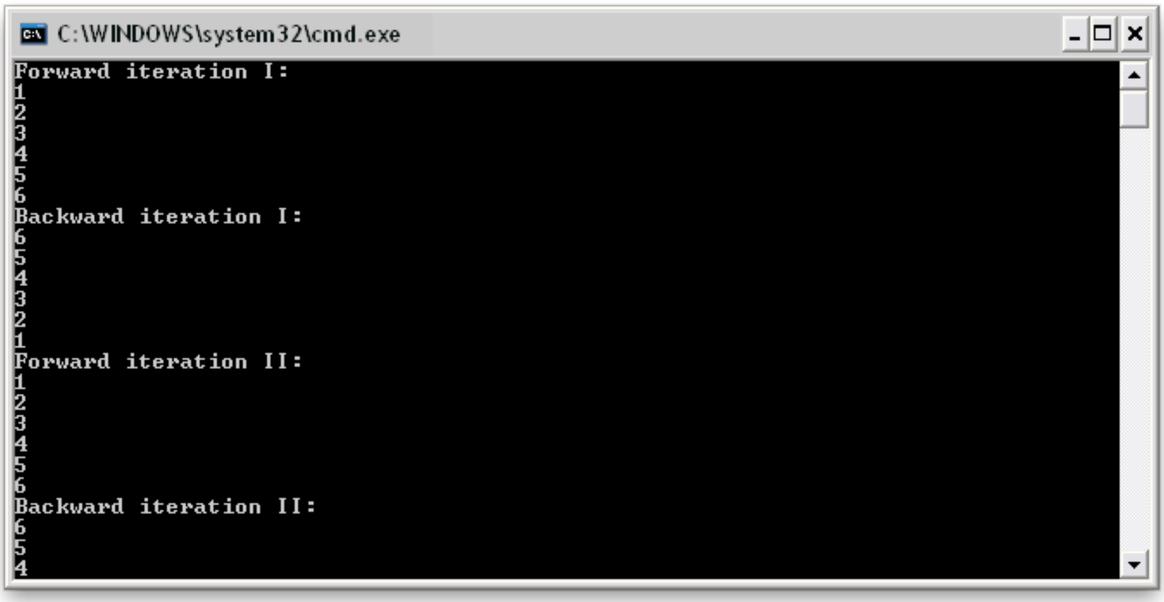
then it means that after performing a decrement operator on the iterator it, the iterator it is indistinguishable from the one produced by the method `leftEnd()`. In other words, if the implication it-- ⇒ it == leftEnd() is true, then the state machine has to switch state (here to `BEFORE`). Naturally, there are also cases where the state machine has to remain in a given state.

Test harness:

```
void testDoublyLinkedNodeIterator()
{
  typedef DoublyLinkedNode<int>::Node IntNode;

  IntNode n1( 1 );
  IntNode n2( 2 );
  IntNode n3( 3 );
  IntNode n4( 4 );
  IntNode n5( 5 );
  IntNode n6( 6 );

  n1.append( n6 );
  n1.append( n5 );
  n1.append( n4 );
  n1.append( n3 );
  n1.append( n2 );

  DoublyLinkedNodeIterator<int> iter( n1 );

  iter--;

  cout << "Forward iteration I:" << endl;
  for ( iter++; iter != iter.rightEnd(); iter++ )
    cout << *iter << endl;

  cout << "Backward iteration I:" << endl;
  for ( iter--; iter != iter.leftEnd(); iter-- )
    cout << *iter << endl;

  cout << "Forward iteration II:" << endl;
  for ( iter = iter.first(); iter != iter.rightEnd(); ++iter )
    cout << *iter << endl;

  cout << "Backward iteration II:" << endl;
  for ( iter = iter.last(); iter != iter.leftEnd(); --iter )
    cout << *iter << endl;
}
```

Result:

```
Forward iteration I:
1
2
3
4
5
6
Backward iteration I:
6
5
4
3
2
1
Forward iteration II:
1
2
3
4
5
6
Backward iteration II:
6
5
4
3
2
1
```



Submission deadline: Tuesday, April 29, 2014, 10:30 a.m.,

Submission procedure: on paper, code of class `DoublyLinkedNodeIterator`.