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:	5 – ADTs
Due date:	**May 11, 2011, 10:30 a.m., on paper**
Lecturer:	Dr. Markus Lumpe

Your name:____________________

Marker's comments:

Problem	Marks	Obtained
1	10	
2	121	
Total	131	

Extension certification:

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

Signature of Convener:__________________________

Problem Set 5: ADTs

Preliminaries

Review the solution of problem set 4(a).

Problem 1:

In problem set 4(a), we defined the template class DoubleLinkedNode, which offered a method insertNode to insert a node as the fNext node of this node and a method dropNode to remove this node from the list. This worked nicely in problem set 4(a). But there is one problem: we cannot make a node the predecessor of this, which is a prerequisite for the data type List. In order to rectify this problem, we have to alter the template class DoubleLinkedNode to contain also a method prependNode, which makes the argument aNode the fPrevious of this.

```
template<class DataType>
class DoubleLinkedNode
{
public:
  typedef DoubleLinkedNode<DataType> Node;

private:
  const DataType* fValue;
  Node* fNext;
  Node* fPrevious;

  DoubleLinkedNode(): fValue((const DataType*)0)
  {
    fNext = (Node*)0;
    fPrevious = (Node*)0;
  }

public:
  static Node NIL;

  DoubleLinkedNode( const DataType& aValue );

  void insertNode( Node& aNode );
  void prependNode( Node& aNode );
  void dropNode();

  const DataType& getValue() const;
  Node& getNext() const;
  Node& getPrevious() const;
};

template<class DataType>
DoubleLinkedNode<DataType> DoubleLinkedNode<DataType>:: NIL;
```

Create the new version of DoubleLinkedNode.

Test harness 1:

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

  typedef DoubleLinkedNode<string>::Node StringNode;

  StringNode n1( s1 );
  StringNode n2( s2 );
  StringNode n3( s3 );

  n3.prependNode( n2 );
  n2.prependNode( n1 );

  cout << "Three elements:" << endl;

  for ( StringNode* pn = &n1; pn != &StringNode::NIL; pn = &pn->getNext() )
  {
    cout << "(";
    if ( &pn->getPrevious() != &StringNode::NIL )
      cout << pn->getPrevious().getValue();
    else
      cout << "<NULL>";

    cout << "," << pn->getValue()<< ",";

    if ( &pn->getNext() != &StringNode::NIL )
      cout << pn->getNext().getValue();
    else
      cout << "<NULL>";

    cout << ")" << endl;
  }

  n1.getNext().dropNode();

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

  for ( StringNode* pn = &n1; pn != &StringNode::NIL; pn = &pn->getNext() )
  {
    cout << "(";
    if ( &pn->getPrevious() != &StringNode::NIL )
      cout << pn->getPrevious().getValue();
    else
      cout << "<NULL>";

    cout << "," << pn->getValue() << ",";

    if ( &pn->getNext() != &StringNode::NIL )
      cout << pn->getNext().getValue();
    else
      cout << "<NULL>";

    cout << ")" << endl;
  }
}
```

Result:

```
Three elements:
(<NULL>,One,Two)
(One,Two,Three)
(Two,Three,<NULL>)
Two elements:
(<NULL>,One,Three)
(One,Three,<NULL>)
```

Problem 2:

Using the new template class `DoubleLinkedNode` and the `NodeIterator` template class from problem set 4(a), implement the template class `List` as specified below:

```
#include "DoubleLinkedNode.h"
#include "DoubleLinkedNodeIterator.h"
#include <stdexcept>

template<class T>
class List
{
private:
  typedef DoubleLinkedNode<T>       Value;
  typedef DoubleLinkedNode<T>*      ListImpl;

  ListImpl fTop;                                 // leftmost element
  ListImpl fLast;                                // rightmost element
  int fCount;                                    // number of nodes

public:
  typedef NodeIterator<T> ListIterator;

  List();                                        // List constructor
  List( const List& aOtherList );                // List copy constructor
  ~List();                                       // List destructor

  List& operator=( const List& aOtherList );     // List assignment operator

  bool isEmpty() const;                          // empty list predicate
  int size() const;                              // get number of nodes
  void add( const T& aElement );                 // add element at end
  void addFirst( const T& aElement );            // add element at top
  bool drop( const T& aElement );                // delete matching element
  void dropFirst();                              // delete first node
  void dropLast();                               // delete last node

  const T& operator[]( int aIndex ) const;       // List indexer
  ListIterator begin() const;                    // List iterator
  ListIterator end() const;                      // List iterator
};
```

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

- Construct an empty list.
- Destruct a list, that is, release any allocated resources.
- A copy constructor and an associate assignment operator
- 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, through `begin()` and `end()`, to traverse the elements of the list either in forward or backwards manner.

The template class `List` constitutes an Adapter for `DoubleLinkedNode` and exposes the required functionality using class `DoubleLinkedNode` as underlying implementation representation. Furthermore, `begin()` and `end()` are Factory methods that return an iterator, which is an instance of `NodeIterator`.

Note: the indexer has to throw an `out_of_range` exception if the given index is out of bounds.

Test harness 2a:

```
void testList2A()
{
  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.begin(); iter != iter.end(); iter++ )
  {
    cout << *iter << endl;
  }

  cout << "Backward:" << endl;

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

Result:

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

Test harness 2b:

```
void testList2B()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  List<string> l;

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

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

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

  l.drop( s2 );

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

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

Result:

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

Test harness 2c:

```
void testList2C()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  List<string> l;

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

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

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

  l.dropFirst();

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

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

Result:

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

Test harness 2d:

```
void testList2D()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  List<string> l;

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

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

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

  l.dropLast();

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

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

Result:

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

Test harness 2e:

```
void testList2E()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  List<string> l;

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

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

  l.dropFirst();

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

  l.dropLast();
  l.dropFirst();
  l.add( s2 );

  cout << "One element:" << endl;

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

Result:

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

Test harness 2f:

```
void testList2F()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  List<string> l;

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

  cout << "To:" << endl;

  for ( int i = 0; i < l.size(); i++ )
  {
    cout << l[i] << endl;
  }

  cout << "Down:" << endl;

  for ( int i = l.size() - 1; i >= 0; i-- )
  {
    cout << l[i] << endl;
  }
}
```

Result:

```
To:
One
Two
Three
Four
Down:
Four
Three
Two
One
```

Test harness 2g:

```
void testList2G()
{
  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;

  List<string> cp = l;

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

  l = cp;

  cout << "Backward:" << endl;

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

Result:

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

Submission deadline: Wednesday, May 11, 2011, 10:30 a.m.

Submission procedure: on paper.