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:	5, Doubly-linked lists
Lecturer:	Dr. Markus Lumpe

Doubly-Linked List Nodes

Define a doubly-linked list that satisfies the following template class specification:

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

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

  DoublyLinkedNode()
  {
    fValue = DataType();
    fNext = &NIL;
    fPrevious = &NIL;
  }

public:
  static Node NIL;

  DoublyLinkedNode( const DataType& aValue );

  void prepend( Node& aNode );
  void append( Node& aNode );
  void remove();

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

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

The template class `DoublyLinkedNode` defines the structure of a doubly-linked list. It uses two pointers: `fNext` and `fPrevious` to connect adjacent list elements. The constructor takes a constant reference `aValue` as argument and returns a properly initialized list node in which both links are set to the address of the sentinel *NIL* – the empty list. The methods `getValue`, `getNext`, and `getPrevious` define simple read-only getter functions for the corresponding fields of a `DoublyLinkedNode` object.

The methods `prepend`, `append`, and `remove` provide then mechanisms to manipulate objects of class `DoublyLinkedNode`. The method `prepend` adds the argument `aNode` object into the list by making `aNode` the new `fPrevious` node of `this`. The method `append`, on the other hand, injects the argument `aNode` object into the list by making `aNode` the new `fNext` node of `this`. The method `remove` removes `this` from the list. That is, `remove` has to properly link the remaining list nodes adjacent to `this`.

There is, however, one complication. Template classes are "class blueprints" or, better, abstractions over classes. Before we can use template classes, we have to instantiate them.

But to work correctly, the instantiation process requires the complete implementation of the class (see lecture notes page 186ff). For this reason, when defining template classes, the implementation has to be included in the header file. There are two ways to accomplish this:

- Implement the member functions directly in the class specification (like it is done in Java or C#).
- Implement the member functions outside the class specification but within the same header file.

If you follow this scheme, working with templates is pretty straightforward.

Test harness 1:

```
void test1()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );

  typedef DoublyLinkedNode<string>::Node StringNode;

  StringNode n1( s1 );
  StringNode n2( s2 );
  StringNode n3( s3 );
  StringNode n4( s4 );

  cout << "Test append:" << endl;

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

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

  for ( const 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;
  }

  n2.remove();

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

  for ( const 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;
  }
}
```

Test harness 2:

```
void test2()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );

  typedef DoublyLinkedNode<string>::Node StringNode;

  StringNode n1( s1 );
  StringNode n2( s2 );
  StringNode n3( s3 );
  StringNode n4( s4 );

  cout << "Test prepend:" << endl;

  n4.prepend( n1 );
  n4.prepend( n2 );
  n4.prepend( n3 );

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

  for ( const 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;
  }

  n3.remove();

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

  for ( const 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;
  }
}
```

Main:

```
#include "DoublyLinkedNode.h"

#include <iostream>
#include <string>

using namespace std;

void test1()
{
      …
}

void test2()
{
      …
}

int main()
{
  test1();

  test2();

  return 0;
}
```

Results:

```
Test append:
The nodes:
(<NULL>,One,Two)
(One,Two,Three)
(Two,Three,Four)
(Three,Four,<NULL>)
The nodes:
(<NULL>,One,Three)
(One,Three,Four)
(Three,Four,<NULL>)

Test prepend:
The nodes:
(<NULL>,One,Two)
(One,Two,Three)
(Two,Three,Four)
(Three,Four,<NULL>)
The nodes:
(<NULL>,One,Two)
(One,Two,Four)
(Two,Four,<NULL>)
```