Swinburne University Of Technology

Faculty of Information and Communication Technologies

ASSIGNMENT COVER SHEET

Subject Code: HIT3303/8303
Subject Title: Data Structures & Patterns
Assignment number and title: 6 – Container Types & Iterators
Due date: **May 18, 2011, 10:30 a.m., on paper**
Lecturer: Dr. Markus Lumpe

Your name:____________________

Marker's comments:

Problem	Marks	Obtained
1	15	
2	30	
3	14	
4	35	
Total	94	

Extension certification:

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

Signature of Convener:________________________

Problem Set 5: Container Types & Iterators

Preliminaries

Review the solution of problem sets 4 & 5.

Bug Fix

The template class `DoubleLinkedNodeIterator` contains a bug that we need to fix before we can proceed with this assignment.

```
template<class T>
NodeIterator<T>::NodeIterator( const Node& aList )
{
  fLeftmost = &aList;

  // bug fix
  if ( fLeftmost != &Node::NIL )
  {
    while ( &fLeftmost->getPrevious() != &Node::NIL )
      fLeftmost = &fLeftmost->getPrevious();
  }

  fRightmost = & aList;

  // bug fix
  if ( fRightmost != &Node::NIL )
  {
    while ( &fRightmost->getNext() != &Node::NIL )
      fRightmost = &fRightmost->getNext();
  }

  // set current to leftmost element;
  fCurrent = &aList;

  // set state
  // bug fix
  fState = fCurrent != &Node::NIL ? DATA : END;
}
```

Problem 1:

Using the template class `List` defined in problem set 5, implement the template class `DynamicStack` as specified below:

```
#include "List.h"
#include <stdexcept>

template<class T>
class DynamicStack
{
private:
  List<T> fElements;

public:
  bool isEmpty() const;
  int size() const;
  void push( const T& aItem );
  void pop();
  const T& top() const;
};
```

That is, `DynamicStack` is a stack container type that can grow in size on demand.

Complete the implementation of the template class `DynamicStack`.

Test harness 1:

```
void test1()
{
  DynamicStack<int> iStack;

  iStack.push( 1 );
  iStack.push( 2 );
  iStack.push( 3 );
  iStack.push( 4 );
  iStack.push( 5 );
  iStack.push( 6 );

  cout << "top: " << iStack.top() << endl;
  iStack.pop();
  iStack.pop();
  cout << "top: " << iStack.top() << endl;
  iStack.pop();
  cout << "top: " << iStack.top() << endl;
  cout << "size: " << iStack.size() << endl;
  cout << "is empty: " << (iStack.isEmpty() ? "T" : "F" ) << endl;
  iStack.pop();
  iStack.pop();
  iStack.pop();
  cout << "is empty: " << (iStack.isEmpty() ? "T" : "F" ) << endl;
}
```

Result:

```
top: 6
top: 4
top: 3
size: 3
is empty: F
is empty: T
```

Problem 2:

Using the template class `DynamicStack`, define a `DynamicStackIterator` that is initialized with a `DynamicStack` and provides a sequential (forward) access to all elements contained in the stack.

```
#include "DynamicStack.h"

template<class T>
class DynamicStackIterator
{
private:
  DynamicStack<T> fStack;
  int fId;

  static int IteratorId;

public:
  DynamicStackIterator( const DynamicStack<T>& aStack );

  const T& operator*() const;                     // dereference
  DynamicStackIterator& operator++();             // prefix increment
  DynamicStackIterator operator++(int);           // postfix increment
  bool operator==( const DynamicStackIterator& aOtherIter ) const;
  bool operator!=( const DynamicStackIterator& aOtherIter ) const;

  DynamicStackIterator end() const; // new iterator (after last element)
};
```

However, this approach requires some extra infrastructure. In particular, we need to introduce a static member variable `IteratorId` as a counter for iterators. This counter will enable us to compare two iterators based on a unique numeric id. Moreover, you cannot compare the top-most element of an empty stack. You need to devise an equivalent measure in order to implement the equivalence operators for `DynamicStackIterator` objects.

Complete the implementation of the template class `DynamicStackIterator`.

Test harness 2:

```
void test2()
{
  DynamicStack<int> iStack;

  iStack.push( 1 );
  iStack.push( 2 );
  iStack.push( 3 );
  iStack.push( 4 );
  iStack.push( 5 );
  iStack.push( 6 );

  cout << "Traverse elements" << endl;

  for ( DynamicStackIterator<int> iter = DynamicStackIterator<int>( iStack );
             iter != iter.end(); iter++ )
  {
    cout << "value: " << *iter << endl;
  }
}
```

Result:

```
Traverse elements
value: 6
value: 5
value: 4
value: 3
value: 2
value: 1
```

Problem 3:

Using the template class `List` defined in problem set 5, implement the template class `DynamicQueue` as specified below:

```
#include "ListImpl.h"
#include <stdexcept>

template<class T>
class DynamicQueue
{
private:
  List<T> fElements;

public:
  bool isEmpty() const;
  int size() const;
  void enqueue( const T& aElement );
  const T& dequeue();
};
```

That is, `DynamicQueue` is a queue container type that can grow in size on demand.

Complete the implementation of the template class `DynamicQueue`.

Test harness 3:

```
void test3()
{
  DynamicQueue<int> iQueue;

  iQueue.enqueue( 1 );
  iQueue.enqueue( 2 );
  iQueue.enqueue( 3 );
  iQueue.enqueue( 4 );
  iQueue.enqueue( 5 );
  iQueue.enqueue( 6 );

  cout << "Queue elements:" << endl;

  while ( !iQueue.isEmpty() )
  {
    cout << "value: " << iQueue.dequeue() << endl;
  }
}
```

Result:

```
Queue elements:
value: 1
value: 2
value: 3
value: 4
value: 5
value: 6
```

Problem 4:

Using the template class `DynamicQueue`, define a `DynamicQueueIterator` that is initialized with a `DynamicQueue` and provides a sequential (forward) access to all elements contained in the queue.

```
#include "DynamicQueue.h"

template<class T>
class DynamicQueueIterator
{
private:
  DynamicQueue<T> fQueue;
  const T* fCurrentElement;
  bool fMustDequeue;
  int fId;

  static int IteratorId;

public:
  DynamicQueueIterator( const DynamicQueue<T>& aQueue );

  const T& operator*();                          // dereference
  DynamicQueueIterator& operator++();            // prefix increment
  DynamicQueueIterator operator++(int);          // postfix increment
  bool operator==( const DynamicQueueIterator& aOtherIter ) const;
  bool operator!=( const DynamicQueueIterator& aOtherIter ) const;

  DynamicQueueIterator end() const; // new iterator (after last element)
};
```

The `DynamicQueueIterator` requires some extra infrastructure. In particular, we need to introduce a static member variable `IteratorId` as a counter for iterators. This counter will enable us to compare two iterators based on a unique numeric id. Moreover, you cannot compare empty queues. You need to devise an equivalent measure in order to implement the equivalence operators for `DynamicQueueIterator` objects.

A particular difficulty is that repeated invocations of the dereference operator without interleaving increments have to return the same element. Since there is no peek operation for queues, you need to store the address of the current element in `fCurrentElement`. It is the dereference operator that updates this instance variable. The increment operators just signal to the iterator that the next deference has to dequeue an element.

Complete the implementation of the template class `DynamicQueueIterator`.

Test harness 4:

```
void test4()
{
  DynamicQueue<int> iQueue;

  iQueue.enqueue( 1 );
  iQueue.enqueue( 2 );
  iQueue.enqueue( 3 );
  iQueue.enqueue( 4 );
  iQueue.enqueue( 5 );
  iQueue.enqueue( 6 );

  cout << "Traverse queue elements" << endl;

  for ( DynamicQueueIterator<int> iter = DynamicQueueIterator<int>( iQueue );
            iter != iter.end(); iter++ )
  {
    cout << "value: " << *iter << endl;
  }
}
```

Result:

```
Traverse queue elements
value: 1
value: 2
value: 3
value: 4
value: 5
value: 6
```

Submission deadline: Wednesday, May 18, 2010, 10:30 a.m.

Submission procedure: on paper.