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:	7 – Container Types & Iterators
Due date:	May 20, 2014, 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	14	
2	19	
3	14	
4	30	
Total	77	

Extension certification:

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

Signature of Convener:__________________________

Problem Set 7: Container Types & Iterators

Preliminaries

Review the solution of problem set 5, the `DoublyLinkedNode` template class developed in the tutorial in week 6, the lecture material regarding the construction of an abstract data type, and the solution of problem set 6 plus the copy control added in the tutorial in week 9.

Problem 1:

Using the template class `List` with proper copy control, implement the template class `DynamicStack` as specified below:

```
#pragma once

#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> lStack;

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

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

Result:

```
top: 6
top: 4
top: 3
size: 3
is empty: F
top: 1
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.

```
#pragma once

#include "DynamicStack.h"

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

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)
};
```

This problem requires some extra considerations. We cannot compare `DynamicStack` objects directly without destroying the stacks. This is not really a problem. We just demand that our stack iterator is being used consistently. That is, we do not mix stack iterators for different stacks. As a result, we only have to compare the respective stack sizes when defining **`operator==`** and **`operator!=`**. (We have used a similar approach when defining the `CharacterCounterIterator` and the `FibonacciIterator`).

What does it mean for a dynamic stack iterator to be positioned after the last element? The answer is straightforward. However, the solution must be consistent with the implementation of **`operator==`** and **`operator!=`**.

Complete the implementation of the template class `DynamicStackIterator`.

Test harness 2:

```
#include <string>

void test2()
{
  DynamicStack<string> lStack;

  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );
  string s5( "Five" );
  string s6( "Six" );

  lStack.push( s1 );
  lStack.push( s2 );
  lStack.push( s3 );
  lStack.push( s4 );
  lStack.push( s5 );
  lStack.push( s6 );

  cout << "Traverse elements" << endl;

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

Result:

```
Traverse elements
value: Six
value: Five
value: Four
value: Three
value: Two
value: One
```

Problem 3:

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

```
#pragma once

#include "List.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> lQueue;

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

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

  while ( !lQueue.isEmpty() )
  {
    cout << "value: " << lQueue.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.

```
#pragma once

#include "DynamicQueue.h"

template<class T>
class DynamicQueueIterator
{
private:
  DynamicQueue<T> fQueue;
  T fCurrentElement;
  bool fMustDequeue;

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 considerations. First, we cannot compare `Queue` objects directly without destroying the queues. This is not really a problem. We just demand that our queue iterator is being used consistently, that is, we do not mix queue iterators for different queues and inspect the respective queue sizes (compare Problem 2: `DynamicStackIterator`).

Second, rather than setting `fCurrentElement` each time we increment the iterator, we call the `dequeue` method only when the flag `fMustDequeue` is true inside the dereference **`operator*`**. Remember, adjacent calls of the deference **`operator*`** must yield the same element, if no increment has occurred in-between. However, elements never requested must be properly skipped. This is a subtle requirement for the proper functioning of the dynamic queue iterator.

Complete the implementation of the template class `DynamicQueueIterator`.

Test harness 4:

```
#include <string>

void test4()
{
  DynamicQueue<string> lQueue;

  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );
  string s5( "Five" );
  string s6( "Six" );

  lQueue.enqueue( s1 );
  lQueue.enqueue( s2 );
  lQueue.enqueue( s3 );
  lQueue.enqueue( s4 );
  lQueue.enqueue( s5 );
  lQueue.enqueue( s6 );

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

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

Result:

```
Traverse queue elements
value: One
value: Three
value: Five
```

Submission deadline: Tuesday, May 20, 2014, 10:30 a.m.

Submission procedure: on paper.