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:_____

DynamicStack.h

```
#ifndef DYNAMICSTACK_H_
#define DYNAMICSTACK_H_

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

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

public:
	bool isEmpty() const
	{
		return fElements.isEmpty();
	}

	int size() const
	{
		return fElements.size();
	}

	void push(const T& aItem)
	{
		fElements.addFirst(aItem);
	}

	void pop()
	{
		fElements.dropFirst();
	}

	const T& top() const
	{
		return fElements[0];
	}
};

#endif /* DYNAMICSTACK_H_ */
```

DynamicStackIterator.h

```
#ifndef DYNAMICSTACKITERATOR_H_
#define DYNAMICSTACKITERATOR_H_

#include "DynamicStack.h"

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

	static int IteratorId;

public:
	DynamicStackIterator(const DynamicStack<T>& aStack)
	{
		fStack = aStack;
		IteratorId++;
		fId = IteratorId;
	}

	const T& operator*() const // dereference
	{
		return fStack.top();
	}

	DynamicStackIterator& operator++() // prefix increment
	{
		fStack.pop();
		return *this;
	}

	DynamicStackIterator operator++(int) // postfix increment
	{
		DynamicStackIterator tempIter = *this;
		fStack.pop();
		return tempIter;
	}

	bool operator==(const DynamicStackIterator& aOtherIter) const
	{
		if(fStack.isEmpty() && aOtherIter.fStack.isEmpty())
			return (fId == aOtherIter.fId);
		else if(fStack.size() != aOtherIter.fStack.size())
			return false;
		else
			return ((fId == aOtherIter.fId) && (fStack.top() == aOtherIter.fStack.top()));
	}

	bool operator!=(const DynamicStackIterator& aOtherIter) const
	{
		return !(*this == aOtherIter);
	}

	DynamicStackIterator end() const // new iterator (after last element)
	{
		DynamicStackIterator returnIter = *this;

		while(!returnIter.fStack.isEmpty()){
			returnIter++;
		}
		return returnIter;
```

```
        }
};

template<class T>
int DynamicStackIterator<T>::IteratorId = 0;

#endif /* DYNAMICSTACKITERATOR_H_ */
```

DynamicQueue.h

```
#ifndef DYNAMICQUEUE_H_
#define DYNAMICQUEUE_H_

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

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

public:
	bool isEmpty() const
	{
		return fElements.isEmpty();
	}

	int size() const
	{
		return fElements.size();
	}

	void enqueue(const T& aElement)
	{
		fElements.addFirst(aElement);
	}

	const T& dequeue()
	{
		if(size() > 0){
			T temp = fElements[fElements.size() - 1];
			fElements.dropLast();
			return temp;
		}
		else{
			throw std::out_of_range("Trying to dequeue an empty queue!");
		}
	}
};

#endif /* DYNAMICQUEUE_H_ */
```

DynamicQueueIterator.h

```
#ifndef DYNAMICQUEUEITERATOR_H_
#define DYNAMICQUEUEITERATOR_H_

#include "DynamicQueue.h"

#include <iostream>

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)
	{
		fQueue = aQueue;
		IteratorId++;
		fId = IteratorId;
		fMustDequeue = true;
	}

	const T& operator*() // dereference
	{
		if(fMustDequeue){
			fCurrentElement = &fQueue.dequeue();
			fMustDequeue = false;
			return *fCurrentElement;
		}
		else{
			return *fCurrentElement;
		}
	}

	DynamicQueueIterator& operator++() // prefix increment
	{
		fMustDequeue = true;
	}

	DynamicQueueIterator operator++(int) // postfix increment
	{
		DynamicQueueIterator tempIter = *this;
		fMustDequeue = true;
		return tempIter;
	}

	bool operator==(const DynamicQueueIterator& aOtherIter) const
	{
		if(fQueue.isEmpty() && aOtherIter.fQueue.isEmpty()){
			return (fId == aOtherIter.fId);
		}
		else if(fQueue.size() != aOtherIter.fQueue.size()){
			return false;
		}
		else{
			return (fId == aOtherIter.fId) && (*fCurrentElement == *aOtherIter.fCurrentElement) &&
			(fMustDequeue == aOtherIter.fMustDequeue);
		}
```

```
	}

	bool operator!=(const DynamicQueueIterator& aOtherIter) const
	{
		return !(*this == aOtherIter);
	}

	DynamicQueueIterator end() const //new iterator (after last element)
	{
		DynamicQueueIterator returnIter = *this;
		while(!returnIter.fQueue.isEmpty()){
			returnIter.fQueue.dequeue();
		}
		return returnIter;
	}
};

template<class T>
int DynamicQueueIterator<T>::IteratorId = 0;

#endif /* DYNAMICQUEUEITERATOR_H_ */
```