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: 4 – Lists, Iterators, and Design Patterns
Due date: **April 13, 2011, 10:30 a.m., on paper**
Lecturer: Dr. Markus Lumpe

Your name:_____

Marker's comments:

Problem	Marks	Obtained
1	25	
2	86	
Total	111	

Extension certification:

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

Signature of Convener:_____

DoubleLinkedNode.h

```
#ifndef DOUBLELINKEDNODE_H_
#define DOUBLELINKEDNODE_H_

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)
	{
		fValue = &aValue;
		fNext = &NIL;
		fPrevious = &NIL;
	}

	//Insert the given node after this node
	void insertNode(Node& aNode)
	{
		if(fNext != &NIL){
			this->fNext->fPrevious = &aNode;
			aNode.fNext = this->fNext;
		}
		this->fNext = &aNode;
		aNode.fPrevious = this;
	}

	void dropNode()
	{
		this->fPrevious->fNext = this->fNext;
		this->fNext->fPrevious = this->fPrevious;
	}

	const DataType& getValue() const
	{
		return *fValue;
	}

	Node& getNext() const
	{
		return *fNext;
	}

	Node& getPrevious() const
	{
		return *fPrevious;
	}
};
template<class DataType>
```

```
DoubleLinkedNode<DataType> DoubleLinkedNode<DataType>::NIL;
#endif /* DOUBLELINKEDNODE_H_ */
```

NodeIterator..h

```
#ifndef NODEITERATOR_H_
#define NODEITERATOR_H_

#include "DoubleLinkedNode.h"
#include <iostream>

template<class DataType>
class NodeIterator
{
private:
	enum IteratorStates {BEFORE, DATA, END};

	IteratorStates fState;
	bool fFirst;

	typedef DoubleLinkedNode<DataType> Node;

	Node& fLeftmost;
	Node& fRightmost;
	const Node* fCurrent;

	static Node& findBeginNode(Node& aList)
	{
		Node* temp = &aList;
		while(&temp->getPrevious() != &Node::NIL)
			temp = &temp->getPrevious();
		return *temp;
	}

	static Node& findEndNode(Node& aList)
	{
		Node* temp = &aList;
		while(&temp->getNext() != &Node::NIL)
			temp = &temp->getNext();
		return *temp;
	}

public:
	typedef NodeIterator<DataType> Iterator;

	NodeIterator(Node& aList): fLeftmost(findBeginNode(aList)), fRightmost(findEndNode(aList))
	{
		fCurrent = &aList;
		//Initialise state
		if(&fCurrent->getPrevious() == &Node::NIL)
			fState = BEFORE;
		else if(&fCurrent->getNext() == &Node::NIL)
			fState = END;
		else
			fState = DATA;
		fFirst = true;
	}

	const DataType& operator*() const //dereference
	{
		return fCurrent->getValue();
	}

	Iterator& operator++() //prefix increment
	{
		if((fState == BEFORE) && (fFirst == false)){
			fFirst = false;
			fState = DATA;
```

```
                return *this;
        }
        if(&fCurrent->getNext() == &Node::NIL){
                fState = END;
        }
        else{
                fFirst = false;
                fState = DATA;
                fCurrent = &fCurrent->getNext();
        }
        return *this;
}

Iterator operator++(int) //postfix increment
{
        if((fState == BEFORE) && (fFirst == false)){
                fFirst = false;
                fState = DATA;
                return *this;
        }
        if(&fCurrent->getNext() == &Node::NIL){
                fState = END;
        }
        else{
                fState = DATA;
                fFirst= false;
                Iterator temp = *this;
                fCurrent = &fCurrent->getNext();
                return temp;
        }
        return *this;
}

Iterator& operator--() //prefix decrement
{
        if(fState == END){
                fState = DATA;
                return *this;
        }
        if(&fCurrent->getPrevious() == &Node::NIL){
                fState = BEFORE;
        }
        else{
                fState = DATA;
                fCurrent = &fCurrent->getPrevious();
        }
        return *this;
}

Iterator operator--(int) //postfix decrement
{
        if(fState == END){
                fState = DATA;
                return *this;
        }
        else if(&fCurrent->getPrevious() == &Node::NIL){
                fState = BEFORE;
        }
        else{
                fState = DATA;
                Iterator temp = *this;
                fCurrent = &fCurrent->getPrevious();
                return temp;
        }
        return *this;
}
```

```
	bool operator==(const Iterator& aOtherIter) const
	{
		return ((fCurrent->getValue() == aOtherIter.fCurrent->getValue()) && (&fRightmost.getValue() ==
			&aOtherIter.fRightmost.getValue()) && (&fLeftmost.getValue() == &aOtherIter.fLeftmost.getValue())
			&& (fState == aOtherIter.fState));
	}

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

	Iterator begin() const
	{
		return Iterator(fLeftmost);
	}

	Iterator end() const
	{
		return Iterator(fRightmost);
	}
};

#endif /* NODEITERATOR_H_ */
```