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: 5 – ADTs
Due date: **May 11, 2011, 10:30 a.m., on paper**
Lecturer: Dr. Markus Lumpe

Your name:_____

Marker's comments:

Problem	Marks	Obtained
1	10	
2	121	
Total	131	

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;
	}

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

	void dropNode()
	{
		fPrevious->fNext = fNext;
		fNext->fPrevious = 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_ */
```

List.h

```
#ifndef LIST_H_
#define LIST_H_

#include "DoubleLinkedNode.h"
#include "NodeIterator.h"
#include <stdexcept>

template<class T>
class List
{
private:
	typedef DoubleLinkedNode<T> Value;
	typedef DoubleLinkedNode<T>* ListImpl;

	ListImpl fTop; //leftmost element
	ListImpl fLast; //rightmost element
	int fCount; //number of nodes

public:
	typedef NodeIterator<T> ListIterator;

	List()//list constructor
	{
		fTop = &Value::NIL;
		fLast = &Value::NIL;
		fCount = 0;
	}

	List(const List& aOtherList) //list copy constructor
	{
		fTop = new Value(aOtherList.fTop->getValue());
		fLast = fTop;
		for(int i = 1; i < aOtherList.size(); i++){
			ListImpl newNode = new Value(aOtherList[i]);
			fLast->insertNode(*newNode);
			fLast = newNode;
		}
		fCount = aOtherList.fCount;
	}

	~List() //list deconstructor
	{
		for(ListImpl curr = fTop; curr != &Value::NIL; curr = &curr->getNext()){
			curr->dropNode();
		}
	}

	List& operator=(const List& aOtherList) //list assignment operator
	{
		//delete old nodes
		for(ListImpl curr = fTop; curr != &Value::NIL; curr = &curr->getNext()){
			curr->dropNode();
			delete curr;
		}
		//get new nodes
		fTop = new Value(aOtherList.fTop->getValue());
		fLast = fTop;
		for(int i = 1; i < aOtherList.size(); i++){
			ListImpl newNode = new Value(aOtherList[i]);
			fLast->insertNode(*newNode);
			fLast = newNode;
		}
		fCount = aOtherList.fCount;
```

```
		return *this;
}

bool isEmpty() const //empty list predicate
{
		if(fCount == 0)
			return true;
		else
			return false;
}

int size() const //get number of nodes
{
		return fCount;
}

void add(const T& aElement) //add element at end
{
		ListImpl newNode = new Value(aElement);
		if(fCount == 0){
			fTop = newNode;
			fLast = newNode;
		}
		else{
			fLast->insertNode(*newNode);
			fLast = newNode;
		}
		fCount++;
}

void addFirst(const T& aElement)//add element at top
{
		ListImpl newNode = new Value(aElement);
		if(fCount == 0){
			fTop = newNode;
			fLast = newNode;
		}
		else{
			fTop->prependNode(*newNode);
			fTop = newNode;
		}
		fCount++;
}

bool drop(const T& aElement) //delete matching element
{
		if(fCount > 0){
			fCount--;
			for(ListImpl curr = fTop; curr != fLast; curr = &curr->getNext()){
				if(curr->getValue() == aElement){
					curr->dropNode();
					delete curr;
					return true;
				}
			}
		}
		return false;
}

void dropFirst() //delete first node
{
		if(fCount > 0){
			ListImpl temp = fTop;
			fTop = &fTop->getNext();
			temp->dropNode();
```

```
                        delete temp;
                        fCount--;
                }
        }

        void dropLast() //delete last node
        {
                if(fCount > 0){
                        ListImpl temp = fLast;
                        fLast = &fLast->getPrevious();
                        temp->dropNode();
                        delete temp;
                        fCount--;
                }
        }

        const T& operator[](int aIndex) const //list indexer
        {
                ListImpl curr = fTop;
                if(aIndex >= size()){
                        throw std::out_of_range("Out of range index!");
                }
                else{
                        for(int i = 0; i < aIndex; i++)
                                curr = &curr->getNext();
                }
                return curr->getValue();
        }

        ListIterator begin() const //list iterator
        {
                ListIterator returnIter(*fTop);
                returnIter = returnIter.begin();
                returnIter++;
                return returnIter;
        }

        ListIterator end() const //list iterator
        {
                ListIterator returnIter(*fLast);
                returnIter = returnIter.end();
                return returnIter;
        }
};

#endif /* LIST_H_ */
```

NodeIterator.h

```
#ifndef NODEITERATOR_H_
#define NODEITERATOR_H_
template<class DataType>
class NodeIterator
{
private:
	enum IteratorStates { BEFORE, DATA , END };

	IteratorStates fState;

	typedef DoubleLinkedNode<DataType> Node;

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

public:
	typedef NodeIterator<DataType> Iterator;

	NodeIterator(const Node& aList)
	{
		//Set leftmost
		fLeftmost = &aList;
		while (&fLeftmost->getPrevious() != &Node::NIL)
			fLeftmost = &fLeftmost->getPrevious();

		//Set rightmost
		fRightmost = &aList;
		while (&fRightmost->getNext() != &Node::NIL)
			fRightmost = &fRightmost->getNext();

	  //start iterator at leftmost element
	  fCurrent = fLeftmost;

	  //set state
	  if(fCurrent == &Node::NIL)
		fState = END;
	  else
		fState = DATA;
	}

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

	Iterator& operator++()
	{
		if(fState == BEFORE){
			fCurrent = fLeftmost;
			if(fCurrent == &Node::NIL)
				fState = END;
			else
				fState = DATA;
		}
		else if(fState == DATA){
			fCurrent = &fCurrent->getNext();
			if(fCurrent == &Node::NIL)
				fState = END;
		}
	  return *this;
	}
```

```
		Iterator operator++(int)
		{
			Iterator returnIter = *this;
			++(*this);

			return returnIter;
		}

		Iterator& operator--()
		{
			if(fState == END){
				fCurrent = fRightmost;
				if(fCurrent == &Node::NIL)
					fState = BEFORE;
				else
					fState = DATA;
			}
			else if(fState == DATA){
				fCurrent = &fCurrent->getPrevious();
				if(fCurrent == &Node::NIL)
					fState = BEFORE;
			}

		 return *this;
		}

		NodeIterator operator--(int)
		{
			Iterator returnIter = *this;
			--(*this);

			return returnIter;
		}

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

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

		Iterator begin()
		{
			Iterator returnIter = *this;
			returnIter.fCurrent = &Node::NIL;
			returnIter.fState = BEFORE;

			return returnIter;
		}

		Iterator end()
		{
			Iterator returnIter = *this;
			returnIter.fCurrent = &Node::NIL;
			returnIter.fState = END;

			return returnIter;
		}
};

#endif /* NODEITERATOR_H_ */
```