Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 3, List ADT
Due date: May 12, 2022, 14:30
Lecturer: Dr. Markus Lumpe

Your name: Dao Khanh Nga Thi **Your student id:** 104177393

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30
				X							

Marker's comments:

Problem	Marks	Obtained
1	48	
2	28	
3	26	
4	30	
5	42	
Total	174	

Extension certification:

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

Signature of Convener:_____

```
#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"
#include <stdexcept>

template<typename T>
class List
{
private:
	// auxiliary definition to simplify node usage
	using Node = DoublyLinkedList<T>;

	Node* fRoot; // the first element in the list
	size_t fCount; // number of elements in the list

public:
	// auxiliary definition to simplify iterator usage
	using Iterator = DoublyLinkedListIterator<T>;

	~List()
	{
		while (fRoot != nullptr)
		{
			if (fRoot != &fRoot->getPrevious())
			{
				Node* lTemp = const_cast<Node*>(&fRoot->getPrevious());
				lTemp->isolate();
				delete lTemp;
			}
			else
			{
				delete fRoot;
				break;
			}
		}
	}

	void remove(const T& aElement)
	{
		Node* fNode = fRoot;
		while (fNode != nullptr)
		{
			if (**fNode == aElement) // If the node contains the element
			{
				break;
			}
			if (fNode != &fRoot->getPrevious()) // If the node is not the
last node in the list
			{
				fNode = const_cast<Node*>(&fNode->getNext());
			}
			else
			{
				fNode = nullptr;
			}
		}
		if (fNode != nullptr)
```

```
        {
                if (fCount = 1) // If the list has only one node
                {
                        fRoot = nullptr;
                }
                else
                {
                        if (fNode == fRoot)
                        {
                                // Update the root pointer to the next node
                                fRoot = const_cast<Node*>(&fRoot->getNext());
                        }
                }
                fNode->isolate();
                delete fNode;
                fCount = fCount - 1;
        }
}

// Problem 1

List() : fRoot(nullptr), fCount(0) {} // default constructor

bool empty() const { return fRoot == nullptr; }

size_t size() const { return fCount; }

void push_front(const T& aElement)
{
        Node* fNode = new Node(aElement);

        if (!empty())
        {
                // Insert the new node at the front
                fRoot->push_front(*fNode);
                fRoot = fNode;

        }
        else
        {
                // The new node becomes the root
                fRoot = fNode;
        }
        fCount = fCount + 1;
}

Iterator begin() const { return Iterator(fRoot).begin(); }

Iterator end() const { return Iterator(fRoot).end(); }

Iterator rbegin() const { return Iterator(fRoot).rbegin(); }

Iterator rend() const { return Iterator(fRoot).rend(); }

// Problem 2

void push_back(const T& aElement)
{
```

```
		Node* fNode = new Node(aElement);
		if (!empty())
		{
			// Find the last node and insert the new node after it
			Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
			lastNode->push_back(*fNode);
		}
		else
		{
			fRoot = fNode;
		}
		fCount = fCount + 1;
	}

	// Problem 3

	const T& operator[](size_t aIndex) const
	{
		Iterator fIterator = Iterator(fRoot).begin();
		if (aIndex > size() - 1)
		{
			// Throw an out of range exception if the index is greater than
the size of the list
			throw std::out_of_range("Index out of bounds.");
		}

		for (size_t i = 0; i < aIndex; i++)
		{
			fIterator++;
		}
		return *fIterator;
	}

	// Problem 4

	List(const List& aOtherList) : fRoot(nullptr), fCount(0)
	{
		*this = aOtherList;
	}

	List& operator=(const List& aOtherList)
	{
		if (&aOtherList != this)
		{
			this->~List(); // Clear the current list by calling the
destructor
			if (aOtherList.fRoot != nullptr)
			{
				fRoot = nullptr;
				fCount = 0;
				for (auto& payload : aOtherList)
				{
					push_back(payload); // Iterate through each payload
in aOtherList and push it to the current list
				}
			}
			else
			{
```

```
                        fRoot = nullptr;
                }
        }
        return *this;
}

// Problem 5

List(List&& aOtherList) : fRoot(nullptr), fCount(0)
{
        *this = std::move(aOtherList);
}

List& operator=(List&& aOtherList)
{
        if (&aOtherList != this)
        {
                this->~List();
                if (aOtherList.fRoot == nullptr)
                {
                        fRoot = nullptr;
                }
                else
                {
                        // Transfer elements from aOtherList to the current list
                        fRoot = aOtherList.fRoot;
                        fCount = aOtherList.fCount;
                        // Clear aOtherList by resetting its root and count to
initial values
                        aOtherList.fRoot = nullptr;
                        aOtherList.fCount = 0;
                }
        }
        return *this;
}

void push_front(T&& aElement)
{
        Node* fNode = new Node(std::move(aElement));
        if (!empty())
        {
                fRoot->push_front(*fNode);
                fRoot = fNode; // If the list is not empty, update the links to
insert the new node at the front
        }
        else
        {
                fRoot = fNode;
        }
        fCount = fCount + 1;
}

void push_back(T&& aElement)
{
        if (!empty())
        {
                Node* lNode = const_cast<Node*>(&fRoot->getPrevious());
                lNode->push_back(*new Node(aElement));
```

```
                }
                else
                {
                        fRoot = new Node(aElement);
                }
                fCount = fCount + 1;
        }
};
```