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: Nguyen Duy Anh Tu **Your student id:** 104188405

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

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:
        using Node = DoublyLinkedList<T>;

        Node* fRoot;
        size_t fCount;

public:
        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* lNode = fRoot;
                while (lNode != nullptr)
                {
                        if (**lNode == aElement)
                        {
                                break;
                        }
                        if (lNode != &fRoot->getPrevious())
                        {
                                lNode = const_cast<Node*>(&lNode->getNext());
                        }
                        else
                        {
                                lNode = nullptr;
                        }
                }
                if (lNode != nullptr)
                {
                        if (fCount != 1)
                        {
                                if (lNode == fRoot)
                                {
                                        fRoot = const_cast<Node*>(&fRoot->getNext());
                                }
                        }
```

```
			else
			{
				fRoot = nullptr;
			}
			lNode->isolate();
			delete lNode;
			fCount--;
		}
	}


	List(): fRoot(nullptr), fCount(0) {}

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

	size_t size() const
	{
		return fCount;
	}

	void push_front(const T& aElement)
	{
		if (empty())
		{
			fRoot = new Node(aElement);
		}
		else
		{
			Node* lNode = new Node(aElement);
			fRoot->push_front(*lNode);
			fRoot = lNode;
		}
		++fCount;
	}

	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();
	}


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

```
        {
            fRoot = new Node(aElement);
        }
        else
        {
            Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
            lastNode->push_back(*new Node(aElement));
        }
        ++fCount;
    }


    const T& operator[](size_t aIndex) const
    {
        if (aIndex > size() - 1)
        {
            throw std::out_of_range("Index out of bounds");
        }
        Iterator lIterator = Iterator(fRoot).begin();
        for (size_t i = 0; i < aIndex; i++)
        {
            ++lIterator;
        }
        return *lIterator;
    }


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

    List& operator=(const List& aOtherList)
    {
        if (&aOtherList != this)
        {
            this->~List();
            if (aOtherList.fRoot == nullptr)
            {
                fRoot = nullptr;
            }
            else
            {
                fRoot = nullptr;
                fCount = 0;
                for (auto& payload : aOtherList)
                {
                    push_back(payload);
                }
            }
        }
        return *this;
    }


    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
            {
                fRoot = aOtherList.fRoot;
                fCount = aOtherList.fCount;
                aOtherList.fRoot = nullptr;
                aOtherList.fCount = 0;
            }
        }
        return *this;
    }

    void push_front(T&& aElement)
    {
        if (empty())
        {
            fRoot = new Node(std::move(aElement));
        }
        else
        {
            Node* lNode = new Node(std::move(aElement));
            fRoot->push_front(*lNode);
            fRoot = lNode;
        }
        ++fCount;
    }

    void push_back(T&& aElement)
    {
        if (empty())
        {
            fRoot = new Node(aElement);
        }
        else
        {
            Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
            lastNode->push_back(*new Node(aElement));
        }
        ++fCount;
    }
};
```