Swinburne University of Technology

School of Science, Computing and Engineering Technologies

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 3, List ADT
Due date: Monday, May 15, 2023, 10:30
Lecturer: Dr. Markus Lumpe

Your name: Ekrar Efaz **Your student id:** 103494172

Check Tutorial	Tues 08:30	Tues 10:30	Tues 12:30 BA603	Tues 12:30 ATC627	Tues 14:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30
									X		

Marker's comments:

Problem	Marks	Obtained
1	118	
2	24	
3	21	
Total	163	

Extension certification:

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

Signature of Convener: _____

```
// COS30008, Problem Set 3, 2023

#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"

template<typename T>
class List
{
private:
    using Node = typename DoublyLinkedList<T>::Node;

    Node fHead;  // first element
    Node fTail;  // last element
    size_t fSize; // number of elements

public:

    using Iterator = DoublyLinkedListIterator<T>;

    List() noexcept:
        fHead( nullptr ),
        fTail( nullptr ),
        fSize( 0 )
    {
    }
 // copy semantics
    List( const List& aOther ):
        fHead( nullptr ),
        fTail( nullptr ),
        fSize( 0 )
    {
        // iterate through the list
        for ( const T& lElement : aOther )
        {
            // add each element to the list
            push_back( lElement );
        }
    }

    List& operator=( const List& aOther ){
        if (this != &aOther)
        {
            List lTemp(aOther);
            swap(lTemp);
        }
        return *this;
    }

 // move semantics
    List( List&& aOther ) noexcept:
        List()
    {
        swap(aOther);
    }
    List& operator=( List&& aOther ) noexcept{
        if (this != &aOther)
        {
            swap(aOther);
        }
    }
    void swap( List& aOther ) noexcept{
        std::swap( fHead, aOther.fHead );
        std::swap( fTail, aOther.fTail );
        std::swap( fSize, aOther.fSize );
    }

 // basic operations
    size_t size() const noexcept{
        return fSize;
    }

    template<typename U>
    void push_front( U&& aData ){
```

```
    typename DoublyLinkedList<T>::Node lNewNode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

    // check if fHead is nullptr
    if ( fHead )
    {
        fHead->fPrevious = lNewNode;
    }
    // set the next node of the added node as the head
    lNewNode->fNext = fHead;

    // set the added node as the head
    fHead = lNewNode;

    // check if fTail is nullptr
    if ( !fTail )
    {
        fTail = fHead;
    }
    // increment the size of the list
    fSize++;
}

template<typename U>
void push_back( U&& aData ){
    typename DoublyLinkedList<T>::Node lNewNode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
    // link the current tail node to the added node
    if ( fTail )
    {
        fTail->fNext = lNewNode;
    }
    // set the previous node of the added node as the current tail node
    lNewNode->fPrevious = fTail;

    // set the added node as the tail
    fTail = lNewNode;

    // check if fHead is nullptr
    if ( !fHead )
    {
        fHead = fTail;
    }
    // increment the size of the list
    fSize++;
}

void remove( const T& aElement ) noexcept{
    Node lCurrentNode = fHead;
    // iterate through the list
    while ( lCurrentNode )
    {
        // check if the current node is the node to be removed
        if ( lCurrentNode->fData == aElement )
        {
            // check if the current node is the head
            if ( lCurrentNode == fHead )
            {
                // set the next node as the head
                fHead = lCurrentNode->fNext;
            }
            // check if the current node is the tail
            if ( lCurrentNode == fTail )
            {
                // set the previous node as the tail
                fTail = lCurrentNode->fPrevious.lock();
            }
            // isolate the current node
            lCurrentNode->isolate();
            // decrement the size of the list
            fSize--;
            // break the loop
            break;
        }
        // move to the next node
        lCurrentNode = lCurrentNode->fNext;
    }
```

```
    }

    const T& operator[]( size_t aIndex ) const{
        assert( 0 <= aIndex );
        assert( aIndex < fSize );
        Node lCurrentNode = fHead;
        // iterate through the list
        for ( size_t i = 0; i < aIndex; i++ )
        {
            // move to the next node
            lCurrentNode = lCurrentNode->fNext;
        }
        // return the data of the current node
        return lCurrentNode->fData;
    }

 // iterator interface
    Iterator begin() const noexcept {
        return Iterator(fHead, fTail).begin();
    }

    Iterator end() const noexcept{
        return Iterator(fHead,fTail).end();
    }

    Iterator rbegin() const noexcept{
        return Iterator(fHead, fTail).rbegin();
    }

    Iterator rend() const noexcept{
        return Iterator(fHead, fTail).rend();
    }

};
```