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: 4, List ADT
Due date: Friday, May 24, 2024, 10:30
Lecturer: Dr. Markus Lumpe

Your name: Vu Duc Tran **Your student id:** 104175614

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:_____

List.h

```
#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) {}   //
default constructor

    // Copy semantics
    List(const List& aOther);                       // copy constructor
    List& operator=(const List& aOther);            // copy assignment

    // Move semantics
    List(List&& aOther) noexcept;                   // move constructor
    List& operator=(List&& aOther) noexcept;        // move assignment
    void swap(List& aOther) noexcept;               // swap elements

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

    template<typename U>
    void push_front(U&& aData) {
        Node newEle = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
        newEle->fNext = fHead;
        if (fHead) {
            fHead->fPrevious = newEle;
        }
        fHead = newEle;
        if (!fTail) {
            fTail = newEle;
        }
        ++fSize;
    }

    template<typename U>
    void push_back(U&& aData) {
        Node newEle = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
        newEle->fPrevious = fTail;
        if (fTail) {
            fTail->fNext = newEle;
        }
        fTail = newEle;
        if (!fHead) {
```

```
            fHead = newEle;
        }
        ++fSize;
    }

    void remove(const T& aElement) noexcept {
        Node now = fHead;
        while (now) {
            if (now->fData == aElement) {
                if (now == fHead) {
                    fHead = now->fNext;
                    if (fHead) {
                        fHead->fPrevious.reset();
                    }
                } else {
                    now->fPrevious.lock()->fNext = now->fNext;
                }
                if (now == fTail) {
                    fTail = now->fPrevious.lock();
                    if (fTail) {
                        fTail->fNext.reset();
                    }
                } else {
                    now->fNext->fPrevious = now->fPrevious;
                }
                now->isolate();
                --fSize;
                return;
            }
            now = now->fNext;
        }
    }

    const T& operator[](size_t aIndex) const {
        Node now = fHead;
        for (size_t i = 0; i < aIndex; ++i) {
            now = now->fNext;
        }
        return now->fData;
    }

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

    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();
```

```
    }
};

template<typename T>
List<T>::List(const List& aOther) : fHead(nullptr), fTail(nullptr), fSize(0)
{
    for (const auto& item : aOther) {
        push_back(item);
    }
}

template<typename T>
List<T>& List<T>::operator=(const List& aOther) {
    if (this != &aOther) {
        List temp(aOther);
        swap(temp);
    }
    return *this;
}

template<typename T>
List<T>::List(List&& aOther) noexcept : fHead(nullptr), fTail(nullptr),
fSize(0) {
    swap(aOther);
}

template<typename T>
List<T>& List<T>::operator=(List&& aOther) noexcept {
    if (this != &aOther) {
        swap(aOther);
    }
    return *this;
}

template<typename T>
void List<T>::swap(List& aOther) noexcept {
    std::swap(fHead, aOther.fHead);
    std::swap(fTail, aOther.fTail);
    std::swap(fSize, aOther.fSize);
}
```