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: H M Asfaq Ahmed Shihab **Your student id:** 103849248

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:_____

```cpp
#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 constructor
    List(const List& aOther)
    {
        fSize = aOther.fSize;
        if (fSize == 0)
        {
            fHead = fTail = nullptr;
        }
        else
        {
            fHead =
             DoublyLinkedList<T>::makeNode(aOther.fHead->fData);
            Node currentSrc = aOther.fHead->fNext;
            Node currentDst = fHead;

            while (currentSrc)
            {
                currentDst->fNext =
                 DoublyLinkedList<T>::makeNode(currentSrc->fData);
                currentDst->fNext->fPrevious = currentDst;
                currentDst = currentDst->fNext;
                currentSrc = currentSrc->fNext;
            }

            fTail = currentDst;
        }
    }

    // Copy assignment operator
    List& operator=(const List& aOther)
```

```
{
    if (this != &aOther)
    {
        List temp(aOther);
        swap(temp);
    }
    return *this;
}

// Move constructor
List(List&& aOther) noexcept : fHead(std::move(aOther.fHead)),
 fTail(std::move(aOther.fTail)), fSize(aOther.fSize)
{
    aOther.fHead = nullptr;
    aOther.fTail = nullptr;
    aOther.fSize = 0;
}

// Move assignment operator
List& operator=(List&& aOther) noexcept
{
    if (this != &aOther)
    {
        clear();
        fHead = std::move(aOther.fHead);
        fTail = std::move(aOther.fTail);
        fSize = aOther.fSize;
        aOther.fHead = nullptr;
        aOther.fTail = nullptr;
        aOther.fSize = 0;
    }
    return *this;
}

void swap(List& aOther) noexcept
{
    std::swap(fHead, aOther.fHead);
    std::swap(fTail, aOther.fTail);
    std::swap(fSize, aOther.fSize);
}

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

// Add element at front
template<typename U>
void push_front(U&& aData)
{
    Node newNode =
     DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
    newNode->fNext = fHead;
```

```
    if (fHead)
    {
        fHead->fPrevious = newNode;
    }
    fHead = newNode;
    if (!fTail)
    {
        fTail = fHead;
    }
    ++fSize;
}

// Add element at back
template<typename U>
void push_back(U&& aData)
{
    Node newNode =
     DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
    newNode->fPrevious = fTail;
    if (fTail)
    {
        fTail->fNext = newNode;
    }
    fTail = newNode;
    if (!fHead)
    {
        fHead = fTail;
    }
    ++fSize;
}

// Remove element
void remove(const T& aElement) noexcept
{
    Node current = fHead;
    while (current)
    {
        if (current->fData == aElement)
        {
            if (current == fHead)
            {
                fHead = current->fNext;
            }
            if (current == fTail)
            {
                fTail = current->fPrevious.lock();
            }
            if (current->fPrevious.lock())
            {
                current->fPrevious.lock()->fNext = current->fNext;
            }
```

```
            if (current->fNext)
            {
                current->fNext->fPrevious = current->fPrevious;
            }
            current->isolate();
            --fSize;
            return;
        }
        current = current->fNext;
    }
}

// List indexer
const T& operator[](size_t aIndex) const
{
    if (aIndex >= fSize)
    {
        throw std::out_of_range("Index out of bounds");
    }
    Node current = fHead;
    for (size_t i = 0; i < aIndex; ++i)
    {
        current = current->fNext;
    }
    return current->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();
}

// Destructor
~List()
{
```

```
        clear();
    }

private:
    void clear()
    {
        while (fHead)
        {
            Node temp = fHead;
            fHead = fHead->fNext;
            temp->isolate();
        }
        fTail = nullptr;
        fSize = 0;
    }
};
```