

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: _____ **Your student id:** _____

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.cpp
// Assignment4
//
// Created by Luan Nguyen on 22/5/2024.
//

#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

    void reset() noexcept;
    void clone(const List& aOther);
    void transfer(List&& aOther) noexcept;

public:
    using Iterator = DoublyLinkedListIterator<T>;

    List() noexcept; // default constructor
    ~List(); // destructor

    // 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; // list size

    template<typename U>
    void push_front(U&& aData); // add element at front

    template<typename U>
    void push_back(U&& aData); // add element at back

    void remove(const T& aElement) noexcept; // remove element

    const T& operator[](size_t aIndex) const; // list indexer

```

```

// iterator interface
Iterator begin() const noexcept;
Iterator end() const noexcept;
Iterator rbegin() const noexcept;
Iterator rend() const noexcept;
};

// Clears the list, releasing all nodes
template<typename T>
void List<T>::reset() noexcept {
    while (fHead) {
        Node temp = fHead;
        fHead = fHead->fNext;
        temp->isolate();
    }
    fTail = nullptr;
    fSize = 0;
}

// Copies elements from another list into this list
template<typename T>
void List<T>::clone(const List& aOther) {
    if (aOther.fSize == 0) {
        fHead = fTail = nullptr;
        fSize = 0;
        return;
    }

    fHead = DoublyLinkedList<T>::makeNode(aOther.fHead->fData);
    Node currNew = fHead;
    Node currOther = aOther.fHead->fNext;

    while (currOther) {
        Node newNode = DoublyLinkedList<T>::makeNode(currOther->fData);
        currNew->fNext = newNode;
        newNode->fPrevious = currNew;
        currNew = newNode;
        currOther = currOther->fNext;
    }

    fTail = currNew;
    fSize = aOther.fSize;
}

template<typename T>
void List<T>::transfer(List&& aOther) noexcept {
    if (this != &aOther) {
        // Release any resources currently held by this list
        reset();

        // Transfer ownership of resources from aOther to this list
    }
}

```

```

        fHead = aOther.fHead;
        fTail = aOther.fTail;
        fSize = aOther.fSize;

        // Reset aOther to an empty state
        aOther.fHead = nullptr;
        aOther.fTail = nullptr;
        aOther.fSize = 0;
    }
}

// Initializes an empty list
template<typename T>
List<T>::List() noexcept : fHead(nullptr), fTail(nullptr), fSize(0) {}

// Clears the list
template<typename T>
List<T>::~~List() {
    reset();
}

// Creates a new list as a copy of another list
template<typename T>
List<T>::List(const List& aOther) {
    Node curr = aOther.fHead;
    while (curr) {
        push_back(curr->fData);
        curr = curr->fNext;
    }
}

// Copies elements from another list into this list
template<typename T>
List<T>& List<T>::operator=(const List& aOther) {
    if (this != &aOther) {
        reset();
        clone(aOther);
    }
    return *this;
}

// Moves elements from another list into this list
template<typename T>
List<T>::List(List&& aOther) noexcept {
    transfer(std::move(aOther));
}

// Moves elements from another list into this list
template<typename T>
List<T>& List<T>::operator=(List&& aOther) noexcept {
    if (this != &aOther) {
        reset();
        transfer(std::move(aOther));
    }
}

```

```

        return *this;
    }

    // Swaps elements with another list
    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);
    }

    // Returns the number of elements in the list
    template<typename T>
    size_t List<T>::size() const noexcept {
        return fSize;
    }

    // Adds an element to the front of the list
    template<typename T>
    template<typename U>
    void List<T>::push_front(U&& aData) {
        Node newNode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

        if (!fHead) { // Check if the list is empty
            fHead = fTail = newNode;
        } else {
            newNode->fNext = fHead;
            fHead->fPrevious = newNode;
            fHead = newNode;
        }

        ++fSize;
    }

    // Adds an element to the back of the list
    template<typename T>
    template<typename U>
    void List<T>::push_back(U&& aData) {
        Node newNode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

        if (!fTail) { // Check if the list is empty
            fHead = fTail = newNode;
        } else {
            newNode->fPrevious = fTail;
            fTail->fNext = newNode;
            fTail = newNode;
        }

        ++fSize;
    }

    // Removes an element from the list
    template<typename T>
    void List<T>::remove(const T& aElement) noexcept {
        Node curr = fHead;

```

```

while (curr) {
    if (curr->fData == aElement) {
        curr->isolate();
        if (curr == fHead)
            fHead = curr->fNext;
        if (curr == fTail)
            fTail = curr->fPrevious.lock();
        --fSize;
        break;
    }
    curr = curr->fNext;
}

// Returns a reference to the element at the specified index
template<typename T>
const T& List<T>::operator[](size_t aIndex) const {
    if (aIndex >= fSize) {
        throw std::out_of_range("Index out of range");
    }

    Node curr = fHead;

    for (size_t i = 0; i < aIndex; ++i) {
        curr = curr->fNext;
    }

    return curr->fData;
}

// Returns an iterator to the beginning/end of the list
template<typename T>
typename List<T>::Iterator List<T>::begin() const noexcept {
    return Iterator(fHead, fTail).begin();
}

template<typename T>
typename List<T>::Iterator List<T>::end() const noexcept {
    return Iterator(fHead, fTail).end();
}

// Returns a reverse iterator to the beginning/end of the list
template<typename T>
typename List<T>::Iterator List<T>::rbegin() const noexcept {
    return Iterator(fTail, fHead).begin();
}

template<typename T>
typename List<T>::Iterator List<T>::rend() const noexcept {
    return Iterator(fTail, fHead).end();
}

```