A paper with text and numbers

Description automatically generated

**Problem 1**

//COS30008 - 104416957 - Avery Flannery

//ADT - List.h

#pragma once

#include "DoublyLinkedList.h"

#include "DoublyLinkedListIterator.h"

#include <utility> // for std::forward

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 (2)

// copy semantics

List(const List& aOther); // copy constructor (10)

List& operator=(const List& aOther); // copy assignment (14)

// move semantics

List(List&& aOther) noexcept; // move constructor (4)

List& operator=(List&& aOther) noexcept; // move assignment (8)

void swap(List& aOther) noexcept; // swap elements (9)

// basic operations

size\_t size() const noexcept { return fSize; } // list size (2)

template<typename U>

void push\_front(U&& aData); // add element at front (24)

template<typename U>

void push\_back(U&& aData); // add element at back (24)

void remove(const T& aElement) noexcept; // remove element (36)

const T& operator[](size\_t aIndex) const; // list indexer (14)

// iterator interface

Iterator begin() const noexcept { return Iterator(fHead, fTail).begin(); } // (4)

Iterator end() const noexcept { return Iterator(fHead, fTail).end(); } // (4)

Iterator rbegin() const noexcept { return Iterator(fHead, fTail).rbegin(); } // (4)

Iterator rend() const noexcept { return Iterator(fHead, fTail).rend(); } // (4)

};

// Implementation of push\_front

template<typename T>

template<typename U>

void List<T>::push\_front(U&& aData)

{

Node newNode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

if (fHead)

{

newNode->fNext = fHead;

fHead->fPrevious = newNode;

fHead = newNode;

}

else

{

fHead = fTail = newNode;

}

fSize++;

}

// Implementation of push\_back

template<typename T>

template<typename U>

void List<T>::push\_back(U&& aData)

{

Node newNode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

if (fTail)

{

newNode->fPrevious = fTail;

fTail->fNext = newNode;

fTail = newNode;

}

else

{

fHead = fTail = newNode;

}

fSize++;

}

// Implementation of remove

template<typename T>

void List<T>::remove(const T& aElement) noexcept

{

Node current = fHead;

while (current)

{

if (current->fData == aElement)

{

if (current == fHead)

{

fHead = current->fNext;

if (fHead)

{

fHead->fPrevious.reset();

}

else

{

fTail.reset();

}

}

else if (current == fTail)

{

fTail = current->fPrevious.lock();

if (fTail)

{

fTail->fNext.reset();

}

else

{

fHead.reset();

}

}

else

{

Node prevNode = current->fPrevious.lock();

Node nextNode = current->fNext;

if (prevNode)

{

prevNode->fNext = nextNode;

}

if (nextNode)

{

nextNode->fPrevious = prevNode;

}

}

current->isolate();

fSize--;

return;

}

current = current->fNext;

}

}

// Implementation of operator[]

template<typename T>

const T& List<T>::operator[](size\_t aIndex) const

{

assert(aIndex < fSize && "Index out of bounds");

Node current = fHead;

for (size\_t i = 0; i < aIndex; ++i)

{

current = current->fNext;

}

return current->fData;

}

**Problem 2**

// Copy constructor

template<typename T>

List<T>::List(const List& aOther) : fHead(nullptr), fTail(nullptr), fSize(0)

{

Node current = aOther.fHead;

while (current)

{

push\_back(current->fData);

current = current->fNext;

}

}

// Copy assignment operator

template<typename T>

List<T>& List<T>::operator=(const List& aOther)

{

if (this != &aOther)

{

List temp(aOther); // Reuse copy constructor

swap(temp);

}

return \*this;

}

// Swap function

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);

}

**Problem 3**

// Move constructor

template<typename T>

List<T>::List(List&& aOther) noexcept : fHead(nullptr), fTail(nullptr), fSize(0)

{

swap(aOther);

}

// Move assignment operator

template<typename T>

List<T>& List<T>::operator=(List&& aOther) noexcept

{

if (this != &aOther)

{

swap(aOther);

}

return \*this;

}

**Output**

**A screen shot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

A black screen with white text

Description automatically generated