COS30008 Semester 1, 2024 Dr. Markus Lumpe

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: Avery Flannery **Your student id:** 104416957

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:___________________________

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>
```

```cpp
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;
}
```

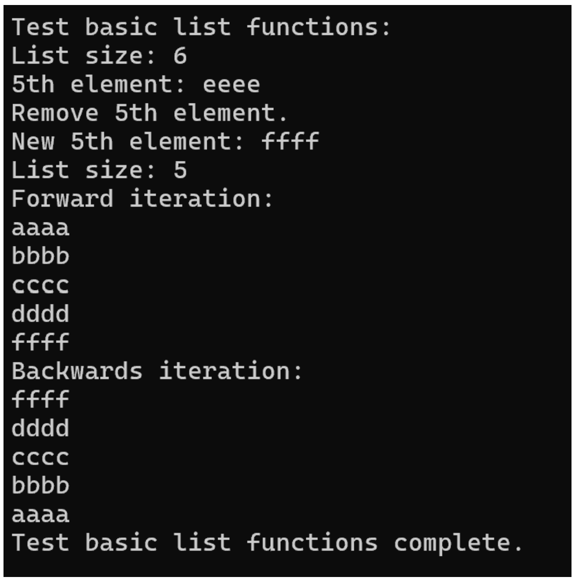
Problem 2

```
// 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);
}
```

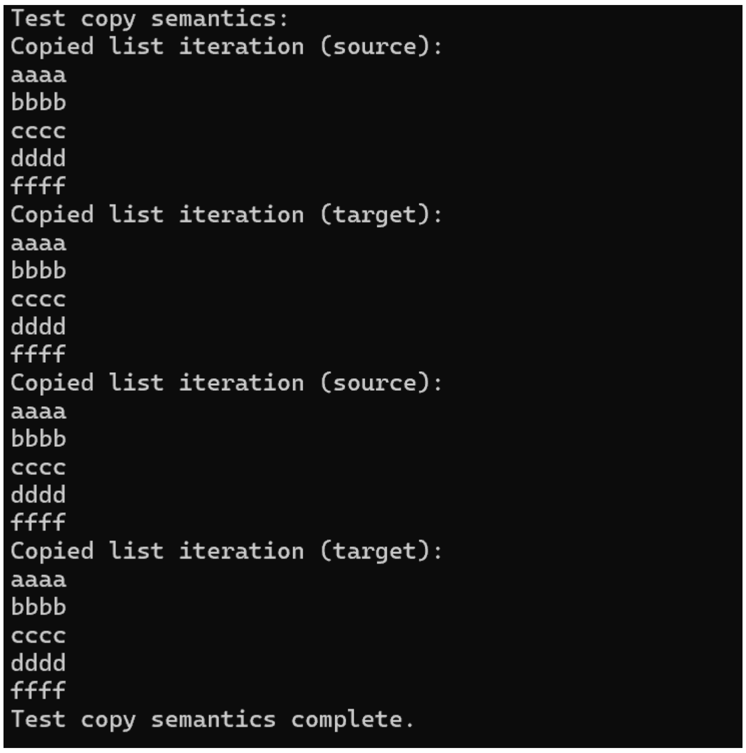
Problem 3

```
// 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;
}
```

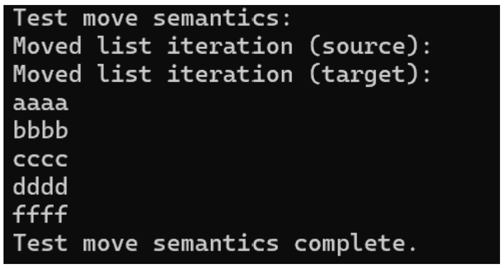
Output



```
Test basic list functions:
List size: 6
5th element: eeee
Remove 5th element.
New 5th element: ffff
List size: 5
Forward iteration:
aaaa
bbbb
cccc
dddd
ffff
Backwards iteration:
ffff
dddd
cccc
bbbb
aaaa
Test basic list functions complete.
```



```
Test copy semantics:
Copied list iteration (source):
aaaa
bbbb
cccc
dddd
ffff
Copied list iteration (target):
aaaa
bbbb
cccc
dddd
ffff
Copied list iteration (source):
aaaa
bbbb
cccc
dddd
ffff
Copied list iteration (target):
aaaa
bbbb
cccc
dddd
ffff
Test copy semantics complete.
```



```
Test move semantics:
Moved list iteration (source):
Moved list iteration (target):
aaaa
bbbb
cccc
dddd
ffff
Test move semantics complete.
```