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.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>;

   // Default constructor
   List() noexcept : fHead(nullptr), fTail(nullptr), fSize(0) {}

   // Copy constructor
   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
   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) {
      swap(aOther);
   }
   return *this;
}

// Swap elements
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));
   if (!fHead) {
      fHead = fTail = newNode;
   }
   else {
      newNode->fNext = fHead;
      fHead->fPrevious = newNode;
      fHead = newNode;
   }
   ++fSize;
}

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

```
    }
    ++fSize;
}

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

// List indexer
const T& operator[](size_t aIndex) const {
    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();
    }
};
```