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:	Wednesday, 13th November 2024, 23:59
Lecturer:	Ms. Siti Hawa

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:__________________________

Problem Set 4: List ADT

Review the template classes `DoublyLinkedList` and `DoublyLinkedListIterator` developed in tutorial 11. In addition, it might be beneficial to review also the lecture material regarding the construction of an abstract data type and linked lists.

Start with the header files provided on Canvas, as they have been fully tested.

Using the template classes `DoublyLinkedList` and `DoublyLinkedListIterator`, implement the template class `List` as specified below:

```
#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;                              // 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;                 // 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;              //                         (4)
  Iterator end() const noexcept;                //                         (4)
  Iterator rbegin() const noexcept;             //                         (4)
  Iterator rend() const noexcept;               //                         (4)
};
```

The template class `List` defines an "object adapter" for `DoublyLinkedList` objects (i.e., the list representation). There are three parts to the implementation: the basic list features, the copy semantics, and the move semantics.

Problem 1

Implement the basic list features.

The default list is empty. The method `size()` just returns the number of list elements.

The create new list elements you need to use `DoublyLinkedNode`'s `makeNode()` method. For `push_front()` and `push_back()` to work properly, we need to use perfect forwarding when calling `makeNode()`. In addition, you need to insert the new element properly into the doubly-linked chain and update `fHead` and `fTail` (the list endpoints). Finally, adding an element to the list increases its size.

`Remove()` deletes the first match from the list. If the list does not contain the element in question, the list remains unchanged. The element to be removed, once found, must be isolated. In addition, it may be necessary to adjust `fHead` and `fTail` when the removed element is the first and last element, respectively. Please note, the removed element is automatically destructed when it goes out of scope.

The indexer has to return the corresponding element, if the index is within bounds. The first element in the list has index 0. The last element has index `size()` – 1.

The iterator methods return the corresponding iterators. See tutorial 10 for details of how to obtain the corresponding iterators.

To test your implementation of the basic features, uncomment **`#define`** `P1` and compile your solution.

The test driver should produce the following 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.
```

Problem 2

Implement the copy semantics.

To test your implementation of the basic features, uncomment **`#define`** `P2` and compile your solution.

The test driver should produce the following output:

```
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.
```

Problem 3

Implement the move semantics.

To test your implementation of the basic features, uncomment **`#define`** `P3` and compile your solution.

The test driver should produce the following output:

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