Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	11, Linked Lists
Lecturer:	Dr. Markus Lumpe

Lab 11: Linked Lists

In this tutorial, we experiment with doubly linked lists and their corresponding iterator.

The test driver (i.e., main.cpp) uses `P1` and `P2` as variables to enable or disable the test associated with the corresponding problem. To enable a test, uncomment the respective **`#define`** line. For example, to test problem 2 only, enable **`#define`** `P2`:

```
// #define P1
#define P2
```

In Visual Studio, the code blocks enclosed in **`#ifdef`** `PX …` **`#endif`** are grayed out if the corresponding test is disabled. The preprocessor definition **`#ifdef`** `PX …` **`#endif`** enables conditional compilation, while Xcode does not use this color-coding scheme.

Problem 1

Define a doubly linked list that satisfies the following template class specification:

```
#pragma once

#include <memory>

template<typename T>
struct DoublyLinkedList
{
  using node = std::shared_ptr<DoublyLinkedList>;
  using node_w = std::weak_ptr<DoublyLinkedList>;

  T data;
  node next;
  node_w previous;

  DoublyLinkedList( const T& aData ) noexcept : data(aData) {}

  DoublyLinkedList( T&& aData ) noexcept : data(std::move(aData)) {}

  void link( node& aPrevious, node& aNext ) noexcept; // link adjacent nodes
  void isolate() noexcept;                            // unlink node

  // factory method for list nodes
  template<typename... Args>
  static node makeNode( Args&&... args )
  {
    // make_share<T, Args...>
    return std::make_shared<DoublyLinkedList>( std::forward<Args>(args)... );
  }
};
```

To guide your implementation, refer to the lecture slides.

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver, which should produce the following output:

```
Test DoublyLinkedList:
Traverse links in the forward direction:
Value: DDDD
Value: EEEE
Value: FFFF
Value: AAAA
Value: BBBB
Value: CCCC
Traverse links in the backward direction:
Value: DDDD
Value: CCCC
Value: BBBB
Value: AAAA
Value: FFFF
Value: EEEE
Traverse links in the forward direction (Four <==> Six):
Value: FFFF
Value: EEEE
Value: DDDD
Value: AAAA
Value: BBBB
Value: CCCC
Traverse links in the forward direction (isolate Three):
Value: FFFF
```

```
Value: EEEE
Value: DDDD
Value: AAAA
Value: BBBB
Test complete.
```

Problem 2

Start with the template class `DoublyLinkedList`. Define a bi-directional list iterator for doubly linked lists that satisfies the following template class specification:

```
#pragma once

#include <cassert>

#include "DoublyLinkedList.h"

template<typename T>
class DoublyLinkedListIterator
{
public:
  using iterator = DoublyLinkedListIterator<T>;
  using node = DoublyLinkedList<T>::node;
  using difference_type = std::ptrdiff_t;      // concept weakly_incrementable
  using value_type = T;                        // concept indirectly_readable

  enum class States { BEFORE, DATA, AFTER };

  DoublyLinkedListIterator() noexcept : fState(States::AFTER) {}
  DoublyLinkedListIterator( const node& aHead, const node& aTail ) noexcept;

  const T& operator*() const noexcept;
  iterator& operator++() noexcept;                // prefix
  iterator operator++(int) noexcept;              // postfix
  iterator& operator--() noexcept;                // prefix
  iterator operator--(int) noexcept;              // postfix

  bool operator==( const iterator& aOther ) const noexcept;

  iterator begin() const noexcept;
  iterator end() const noexcept;
  iterator rbegin() const noexcept;
  iterator rend() const noexcept;

private:
  node fHead;
  node fTail;
  node fPosition;
  States fState;
};

template<typename T>
concept BasicBidirectionalIterator =
std::bidirectional_iterator<T> &&
requires( T i, T j )
{
    { i.begin() } -> std::same_as<T>;
    { i.end() } -> std::same_as<T>;
};

static_assert( BasicBidirectionalIterator<DoublyLinkedListIterator<int>>);
```

The doubly-linked list iterator implements a standard bidirectional iterator. The dereference operator provides access to the current element to which the iterator is positioned. The increment operators advance the iterator to the next element, while the decrement operators move it to the previous element. Additionally, the doubly-linked list iterator defines an equivalence predicate and four factory methods: `begin()`, `end()`, `rbegin()`, and `rend()`. The method `begin()` returns a new iterator positioned at the first element, while `end()` returns a new iterator that is placed after the last element. The `rbegin() method` returns a new iterator set at the final element, and the `rend() method` returns a new list iterator positioned before the first element of the doubly-linked list.

To guarantee the correct behavior of the `DoublyLinkedListIterator`, it must implement a *state machine* with three states: `BEFORE`, `DATA`, `AFTER`. Think of the iterator as an analog clock. The start of the list is noon. The iterator can freely move around the clock in either direction. However, it must not go past noon. This position marks the end of a forward or backwards iteration.

The respective ends of the doubly linked list are `fHead` and `fTail` in the iterator. The ends of the doubly linked list are not connected (i.e., `fPrevious` of the `fHead` is **`nullptr`** and `fNext` of the `fTail` is **`nullptr`**). Together with the state of the iterator, we can hence always determine whether there are more elements to be visited or if the iterator has reached the end. The direction (i.e., increment or decrement) tells us whether we have moved past the last element or whether we have moved before the first element in the doubly linked list.

The constructor for the iterator has to test a crucial precondition. The head and the tail must either point to a proper list node or be **`nullptr`**.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver, which should produce the following output (sorting in increasing order and counting the number of exchanges):

```
Test DoublyLinkedListIterator:
Forward iteration:
AAAA
BBBB
CCCC
DDDD
EEEE
FFFF
Backwards iteration:
FFFF
EEEE
DDDD
CCCC
BBBB
AAAA
Test complete.
```

There should be no memory leaks. The smart pointers handle memory management automatically when a list object goes out of scope.

Please note, however, that the doubly linked list uses a weak smart pointer. To access the corresponding handle (i.e., the pointer to the object stored on the heap), the weak smart pointer must be locked first to obtain a shared smart pointer.