Swinburne University of Technology

School of Science, Computing and Engineering Technologies

FINAL EXAM COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures & Patterns
Due date:	June 14, 2024, 12:00 AEST
Lecturer:	Dr. Markus Lumpe

Your name: Quang Vinh Le_____ **Your student id:**104097488_____

Marker's comments:

Problem	Marks	Obtained
1	34	
2	130	
3	114	
4	68	
Total	346	

This test requires approx. 2 hours and accounts for 50% of your overall mark.

Problem Set 1 (List.h)

// COS30008, Final Exam, 2024

```
#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"

template<typename T>
class List
{
private:
    using Node = typename DoublyLinkedList<T>::Node;

    Node fHead;
    Node fTail;
    size_t fSize;

public:

    using Iterator = DoublyLinkedListIterator<T>;

    List() noexcept :
        fSize(0)
    {}

    // Problem 1
    ~List() noexcept
    {
        Node lCurrent = fTail;

        while (lCurrent)
        {
            Node lPrevious = lCurrent->fPrevious.lock();
            lCurrent->fPrevious.reset();
            lCurrent->fNext.reset();
            lCurrent = lPrevious;
        }
        fHead.reset();
        fTail.reset();
    }

    List( const List& aOther ) :
        List()
    {
        for ( auto& item : aOther )
```

```
    {
        push_back( item );
    }
}

List& operator=( const List& aOther )
{
    if ( this != &aOther )
    {
        this->~List();

        new (this) List( aOther );
    }

    return *this;
}

List( List&& aOther ) noexcept :
    List()
{
    swap( aOther );
}

List& operator=( List&& aOther ) noexcept
{
    if ( this != &aOther )
    {
        swap( aOther );
    }

    return *this;
}

void swap( List& aOther ) noexcept
{
    std::swap( fHead, aOther.fHead );
    std::swap( fTail, aOther.fTail );
    std::swap( fSize, aOther.fSize );
}

size_t size() const noexcept
{
    return fSize;
}
```

```
template<typename U>
void push_front( U&& aData )
{
   Node lNode = DoublyLinkedList<T>::makeNode( std::forward<U>(aData) );

   if ( !fHead )                          // first element
   {
      fTail = lNode;                      // set tail to first element
   }
   else
   {
      lNode->fNext = fHead;               // new node becomes head
      fHead->fPrevious = lNode;           // new node previous of head
   }

   fHead = lNode;                         // new head
   fSize++;                               // increment size
}

template<typename U>
void push_back( U&& aData )
{
   Node lNode = DoublyLinkedList<T>::makeNode( std::forward<U>(aData) );

   if ( !fTail )                          // first element
   {
      fHead = lNode;                      // set head to first element
   }
   else
   {
      lNode->fPrevious = fTail;           // new node becomes tail
      fTail->fNext = lNode;               // new node next of tail
   }

   fTail = lNode;                         // new tail
   fSize++;                               // increment size
}

void remove( const T& aElement ) noexcept
{
   Node lNode = fHead;                    // start at first

   while ( lNode )                        // Are there still nodes available?
```

```
    {
        if ( lNode->fData == aElement )         // Have we found the node?
        {
            break;                              // stop the search
        }

        lNode = lNode->fNext;                   // move to next node
    }

    if ( lNode )                                // We have found a first matching node.
    {
        if ( fHead == lNode )                   // remove head
        {
            fHead = lNode->fNext;               // make lNode's next head
        }

        if ( fTail == lNode )                   // remove tail
        {
            fTail = lNode->fPrevious.lock();    // make lNode's previuos tail, requires std::shared_ptr
        }

        lNode->isolate();                       // isolate node, automatically freed
        fSize--;                                // decrement count
    }
}

const T& operator[]( size_t aIndex ) const
{
    assert( aIndex < fSize );

    Node lNode = fHead;

    while ( aIndex-- )
    {
        lNode = lNode->fNext;
    }

    return lNode->fData;
}

Iterator begin() const noexcept
{
    return Iterator( fHead, fTail );
}
```

```
    Iterator end() const noexcept
    {
        return begin().end();
    }

    Iterator rbegin() const noexcept
    {
        return begin().rbegin();
    }

    Iterator rend() const noexcept
    {
        return begin().rend();
    }
};
```

Problem Set 2 (DynamicQueue.h)

```
#pragma once
#include <optional>
#include <cassert>
#include <iostream>

template<typename T>
class DynamicQueue {
private:
    T* fElements;
    size_t fFirstIndex;
    size_t fLastIndex;
    size_t fCurrentSize;
    size_t fCapacity;


    void resize(size_t aNewSize) {
        if (aNewSize == 0) {
            throw std::runtime_error("New size cannot be 0.");
        }
        T* newElements = new T[aNewSize];
        for (size_t i = 0; i < fCurrentSize; ++i) {
            newElements[i] = fElements[(fFirstIndex + i) % fCapacity];
        }
        delete[] fElements;
        fElements = newElements;
        fFirstIndex = 0;
```

```
        fLastIndex = fCurrentSize;
        fCapacity = aNewSize;
    }

    void ensure_capacity() {
        if (fCurrentSize == fCapacity) {
            resize(fCapacity * 2);
        }
    }

    void adjust_capacity() {
        if (fCurrentSize <= fCapacity / 4 && fCapacity > 1) {
            resize(fCapacity / 2);
        }
    }

public:
    DynamicQueue() : fElements(new T[1]), fFirstIndex(0), fLastIndex(0), fCurrentSize(0),
fCapacity(1) {}

    ~DynamicQueue() { delete[] fElements; }

    DynamicQueue(const DynamicQueue&) = delete;
    DynamicQueue& operator=(const DynamicQueue&) = delete;

    std::optional<T> top() const noexcept {
        if (fCurrentSize == 0) {
            return std::nullopt;
        }
        else {
            return fElements[fFirstIndex];
        }
    }

    void enqueue(const T& aValue) {
        ensure_capacity();
        fElements[fLastIndex] = aValue;
        fLastIndex = (fLastIndex + 1) % fCapacity;
        ++fCurrentSize;
    }

    void dequeue() {
        assert(fCurrentSize > 0);
        fFirstIndex = (fFirstIndex + 1) % fCapacity;
```

```
        --fCurrentSize;
        adjust_capacity();
    }
};
```

Problem Set 3 (PalindromeStringIterator.cpp)

```
#include "PalindromeStringIterator.h"
#include <iostream>
#include <cctype>

void PalindromeStringIterator::moveToNextIndex() {
    while (++fIndex < static_cast<int>(fString.length())) {
        if (std::isalpha(fString[fIndex])) {
            break;
        }
    }
}

void PalindromeStringIterator::moveToPreviousIndex() {
    while (--fIndex >= 0) {
        if (std::isalpha(fString[fIndex])) {
            break;
        }
    }
}

PalindromeStringIterator::PalindromeStringIterator(const std::string& aString)
    : fString(aString), fIndex(-1) {
    moveToNextIndex();
}

char PalindromeStringIterator::operator*() const noexcept {
    return std::toupper(fString[fIndex]);
}

PalindromeStringIterator& PalindromeStringIterator::operator++() noexcept {
    moveToNextIndex();
    return *this;
}

PalindromeStringIterator PalindromeStringIterator::operator++(int) noexcept {
    PalindromeStringIterator old = *this;
    ++(*this);
    return old;
```

```
}

PalindromeStringIterator& PalindromeStringIterator::operator--() noexcept {
    moveToPreviousIndex();
    return *this;
}

PalindromeStringIterator PalindromeStringIterator::operator--(int) noexcept {
    PalindromeStringIterator old = *this;
    --(*this);
    return old;
}

bool PalindromeStringIterator::operator==(const PalindromeStringIterator& aOther) const noexcept
{
    return fIndex == aOther.fIndex;
}

bool PalindromeStringIterator::operator!=(const PalindromeStringIterator& aOther) const noexcept {
    return !(*this == aOther);
}

PalindromeStringIterator PalindromeStringIterator::begin() const noexcept {
    PalindromeStringIterator temp = *this;
    temp.fIndex = -1;
    temp.moveToNextIndex();
    return temp;
}

PalindromeStringIterator PalindromeStringIterator::end() const noexcept {
    PalindromeStringIterator temp = *this;
    temp.fIndex = static_cast<int>(fString.length());
    return temp;
}

PalindromeStringIterator PalindromeStringIterator::rbegin() const noexcept {
    PalindromeStringIterator temp = *this;
    temp.fIndex = static_cast<int>(fString.length());
    temp.moveToPreviousIndex();
    return temp;
}

PalindromeStringIterator PalindromeStringIterator::rend() const noexcept {
    PalindromeStringIterator temp = *this;
```

```
    temp.fIndex = -1;
    return temp;
}
```

Problem Set 4

Answer the following questions in one or two sentences:

a. What is a weak pointer and when do we use it? (8)

4a) A weak pointer in C++ is designed to refer to an object without claiming ownership, helping to prevent circular dependencies and memory leaks in situations where multiple owners share an object.

b. How do we guarantee preconditions for operations in C++? (2)

4b) Preconditions for functions in C++ can be enforced using assert for development checks or by validating conditions and throwing exceptions if necessary.

c. What are the canonical methods in C++? (12)

4c) The fundamental special member functions in C++ include the default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator, and destructor.

d. Is Quick Sort strictly better than Merge Sort? Justify. (8)

4d) Quick Sort usually has better performance on average but has a potential worst-case time complexity of O(n^2), while Merge Sort consistently performs in O(n log n) time and is stable, making it better for certain datasets.

e. What is the purpose of an empty tree? Justify. (8)

4e) An empty tree acts as a placeholder or null object for a tree with no nodes, which simplifies the logic of tree operations and provides a clear base case for recursive algorithms.

f. Which modern C++ abstraction do we use when we need to return a value that does not exist? (2)

4f) The std::optional type in modern C++ is used to represent values that might not exist, avoiding the need for pointers or special sentinel values.

g. What does amortized analysis show? (4)

4g) Amortized analysis provides the average time per operation over a sequence, ensuring that even if some operations are costly, the overall time complexity remains manageable.

h. What is a load factor and what are the recommended factors, thresholds, and aims for expansion and contraction of dynamic memory? (12)

4h) The load factor, which is the ratio of elements to buckets in a hash table, typically should be between 0.7 and 0.75. Expansion occurs when this ratio is too high, and contraction happens when it is significantly low.

i. What is required to test the equivalence of iterators? (4)

4i) To check if iterators are equivalent, they must belong to the same container and either point to the same element or both be at the container's end.

j. When do we need to implement a state machine? (8)

4j) A state machine is necessary when an object has multiple states and its behavior changes based on the current state, which is common in applications like parsers, protocol handlers, and user interfaces.