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 8, 2023, 09:00 – 12:00 AEST
Lecturer: Dr. Markus Lumpe

Your name: Sartaj Khan **Your student id:** 103509694

Check Tutorial	Tues 08:30	Tues 10:30	Tues 12:30 BA603	Tues 12:30 ATC627	Tues 14:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30
								✓			

Marker's comments:

Problem	Marks	Obtained
1	24	
2	88	
3	60	
4	60	
5	68	
Total	300	

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

```

// COS30008, Final Exam, 2023

#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)
    {}

    // Final Exam, 2023
    ~List()
    {
        // Problem 1
        Node lCurrent = fTail;
        fTail.reset();
        while (lCurrent) {
            Node lPrevious = lCurrent->fPrevious.lock();
            if (lPrevious) {
                lPrevious->fNext.reset();
                lCurrent.reset();
            }

            lCurrent = lPrevious;
        }
    }

    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();
    }
};

```

```cpp

// COS30008, Final Exam, 2023

#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)
    {}

    // Final Exam, 2023
    ~List()
    {
        // Problem 1
        Node lCurrent = fTail;
        fTail.reset();
        while (lCurrent) {
            Node lPrevious = lCurrent->fPrevious.lock();
            if (lPrevious) {
                lPrevious->fNext.reset();
                lCurrent.reset();
            }

            lCurrent = lPrevious;
        }
    }

    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();
    }
```

```cpp

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

```

```

// COS30008, Final Exam, 2023

#pragma once

#include <memory>
#include <cassert>
#include <algorithm>

template<typename T>
class TernaryTree
{
public:

    using Node = std::unique_ptr<TernaryTree>;

public:

    TernaryTree(const T& aKey = T{}) noexcept : fKey(aKey)

    {
        for (size_t i = 0; i < 3; i++) {
            fNodes[i] = nullptr;
        }
    }
    TernaryTree(T&& aKey) noexcept : fKey(std::move(aKey))
    {
        for (size_t i = 0; i < 3; i++) {
            fNodes[i] = nullptr;
        }
    }

    template<typename... Args>
    static Node makeNode(Args&&... args) {
        return std::make_unique<TernaryTree>(std::forward<Args>(args)...);
    }

    const T& operator*() const noexcept {
        return fKey;
    }

    TernaryTree& operator[](size_t aIndex) const {
        assert(aIndex < 3);
        return *fNodes[aIndex];
    }

    void add(size_t aIndex, Node& aNode) {
        assert(aIndex < 3);
        if (this) {
            if (!fNodes[aIndex]) {
                fNodes[aIndex] = std::move(aNode);
            }
        }
    }
    Node remove(size_t aIndex) {
        assert(aIndex < 3);
        if (this) {
            if (fNodes[aIndex]) {
                Node lRemoved = std::move(fNodes[aIndex]);
                return lRemoved;
            }
        }
    }

    bool leaf() const noexcept {
        if (this) {
            for (size_t i = 0; i < 3; i++) {
                if (fNodes[i] == nullptr)  {
                    continue;
                }
                else {
                    return false;
                }
            }
        }
        return true;
    }
    size_t height() const noexcept {
        if (leaf()) {
            return 0;
        }
        else {
            size_t lHeight = fNodes[0]->height();
            size_t mHeight = fNodes[1]->height();
            size_t rHeight = fNodes[2]->height();

            if (lHeight >= mHeight && lHeight >= rHeight) return lHeight+1;
            if (mHeight >= lHeight && mHeight >= rHeight) return mHeight+1;
            if (rHeight >= mHeight && rHeight >= lHeight) return rHeight+1;
        }
    }

private:

    T fKey;
    Node fNodes[3];
};

```

```cpp

// COS30008, Final Exam, 2023

#include "DSPString.h"

#include <cassert>
#include <algorithm>

DSPString::DSPString( const char* aContents )
{
    size_t lSize = 0;

    while ( aContents[lSize] )
    {
        lSize++;
    }

    fContents = new char[++lSize];

    for ( size_t i = 0; i < lSize; i++ )
    {
        fContents[i] = aContents[i];
    }
    fSize = lSize;
}

DSPString::~DSPString()
{
    delete[] fContents;

}

DSPString::DSPString( const DSPString& aOther ) :
    DSPString( aOther.fContents )
{}

DSPString& DSPString::operator=( const DSPString& aOther )
{
    if (!(*this == aOther)) {
        this->~DSPString();

        new (this) DSPString(aOther.fContents);
    }

    return *this;

}

DSPString::DSPString( DSPString&& aOther ) noexcept :
    DSPString( "\0" )
{
    std::swap(fContents, aOther.fContents);
    std::swap(fSize, aOther.fSize);
}

DSPString& DSPString::operator=( DSPString&& aOther ) noexcept
{
    if (!(this == &aOther)) {
        std::swap(fContents, aOther.fContents);
        std::swap(fSize, aOther.fSize);
    }
    return *this;
}

size_t DSPString::size() const noexcept
{
    return fSize;

}

char DSPString::operator[]( size_t aIndex ) const noexcept
{
    assert(aIndex < fSize);
    return fContents[aIndex];
}

bool DSPString::operator==( const DSPString& aOther ) const noexcept
{
    if ( size() == aOther.size() )
    {
        for ( size_t i = 0; i < size(); i++ )
        {
            if ( fContents[i] != aOther.fContents[i] )
            {
                return false;
            }
        }

        return true;
    }

    return false;
}

std::ostream& operator<<( std::ostream& aStream, const DSPString& aObject )
{
    return aStream << aObject.fContents;
}
```

```cpp

// COS30008, Final Exam, 2023

#include "DSPStringIterator.h"

DSPStringIterator::DSPStringIterator( const DSPString& aCollection ) :
    fCollection( std::make_shared<DSPString>( aCollection ) ),
    fIndex( 0 )
{}

char DSPStringIterator::operator*() const noexcept
{
    return (*fCollection.get())[fIndex];
}

DSPStringIterator& DSPStringIterator::operator++() noexcept
{
    fIndex++;
    return *this;
}

DSPStringIterator DSPStringIterator::operator++( int ) noexcept
{
    DSPStringIterator old = *this;

    ++(*this);

    return old;
}

DSPStringIterator& DSPStringIterator::operator--() noexcept
{
    fIndex--;
    return *this;
}

DSPStringIterator DSPStringIterator::operator--( int ) noexcept
{
    DSPStringIterator old = *this;

    --(*this);

    return old;
}

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

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

DSPStringIterator  DSPStringIterator::begin() const noexcept
{
    DSPStringIterator begin = *this;
    begin.fIndex = 0;
    return begin;
}

DSPStringIterator DSPStringIterator::end() const noexcept
{
    DSPStringIterator end = *this;
    end.fIndex = fCollection.get()->size()-1;
    return end;
}

DSPStringIterator DSPStringIterator::rbegin() const noexcept
{
    DSPStringIterator rBegin = *this;
    return rBegin.end();
}

DSPStringIterator DSPStringIterator::rend() const noexcept
{
    DSPStringIterator rEnd = *this;
    rEnd.fIndex = -1;
    return rEnd;
}

```

Problem 5 (68 marks)

Answer the following questions in one or two sentences:

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

5a)

Weak Pointers are smart pointers that do not reference count for an object. Weak pointers are used to break the circular dependency created by shared pointers.

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

5b)

We use assert statements to ensure the condition we are looking for is true. Otherwise, the function can cause undefined behavior or crash, so we abort the execution of the function using the assert statement.

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

5c)

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

5d)

Quick sort has complexity O(n^2) and merge sort O(nlogn). Quick sort is very fast for smaller arrays while merge sort is faster over increasing array sizes.

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

5e)

Practically, it is impossible to have an n-ary tree with all nodes having the same degree since this will lead to an infinite data structure. The notion of empty trees allow a brute-force way to define the definite degree to a tree and allow us to create an n-ary tree without an infinite structure. Whenever, a node does not have a child node, we replace it with an empty tree.

Problem 5 (68 marks)

Answer the following questions in one or two sentences:

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

5a) Weak Pointers are smart pointers that do not reference count for an object. Weak pointers are used to break the circular dependency created by shared pointers.

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

5b) We use assert statements to ensure the condition we are looking for is true. Otherwise, the function can cause undefined behavior or crash, so we abort the execution of the function using the assert statement.

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

5c)

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

5d) Quick sort has complexity O(n^2) and merge sort O(nlogn). Quick sort is very fast for smaller arrays while merge sort is faster over increasing array sizes.

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

5e) Practically, it is impossible to have an n-ary tree with all nodes having the same degree since this will lead to an infinite data structure. The notion of empty trees allow a brute-force way to define the definite degree to a tree and allow us to create an n-ary tree without an infinite structure. Whenever, a node does not have a child node, we replace it with an empty tree.

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

5f)

We use std::optional<>

g. What does amortized analysis show? (4)

5g)

Amortized analysis accounts for the worst case scenario for an algorithm or data structure. It averages the intensive operations and tells us the behavior for the worst case. The importance of this is, the intensive operations might be seldom used in the data structure so regarding the total behavior will lead to a wrong analysis of the behavior of the data structure. Averaging balances this act and gives us a proper notion - a weighted average which signifies the intensity and complexity of each statement of the data structure.

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

5h)

Load factors are used to alter the size of objects during runtime. It utilises a ratio of empty space to total space and uses this to decide whether the size needs to be increased or decreased. For example, a load factor of 1 means we should double the space for more objects to be stored whereas, a load factor of one-quarter means, we can halve the space.

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

5i)

Test whether each of the iterator attributes are the same and return the value by and-ing all of them.

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

5j)

When we have an interface, to tack the execution of actions, inputs and outputs we use state machines. This gives us a visual description of the functionality of the program and the flow of the program execution.

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

5f)

We use std::optional<>

g. What does amortized analysis show? (4)

5g)

Amortized analysis accounts for the worst case scenario for an algorithm or data structure. It averages the intensive operations and tells us the behavior for the worst case. The importance of this is, the intensive operations might be seldom used in the data structure so regarding the total behavior will lead to a wrong analysis of the behavior of the data structure. Averaging balances this act and gives us a proper notion - a weighted average which signifies the intensity and complexity of each statement of the data structure.

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

5h)

Load factors are used to alter the size of objects during runtime. It utilises a ratio of empty space to total space and uses this to decide whether the size needs to be increased or decreased. For example, a load factor of 1 means we should double the space for more objects to be stored whereas, a load factor of one-quarter means, we can halve the space.

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

5i)

Test whether each of the iterator attributes are the same and return the value by and-ing all of them.

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

5j)

When we have an interface, to tack the execution of actions, inputs and outputs we use state machines. This gives us a visual description of the functionality of the program and the flow of the program execution.

Problem 5 **(68 marks)**

Answer the following questions in one or two sentences:

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

5a) Weak Pointers are smart pointers that do not reference count for an object. Weak pointers are used to break the circular dependency created by shared pointers.

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

5b) We use assert statements to ensure the condition we are looking for is true. Otherwise, the function can cause undefined behavior or crash, so we abort the execution of the function using the assert statement.

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

5c)

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

5d) Quick sort has complexity O(n^2) and merge sort O(nlogn). Quick sort is very fast for smaller arrays while merge sort is faster over increasing array sizes.

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

5e) Practically, it is impossible to have an n-ary tree with all nodes having the same degree since this will lead to an infinite data structure. The notion of empty trees allow a brute-force way to define the definite degree to a tree and allow us to create an n-ary tree without an infinite structure. Whenever, a node does not have a child node, we replace it with an empty tree.

Problem 5 (68 marks)

Answer the following questions in one or two sentences:

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

5a)

Weak Pointers are smart pointers that do not reference count for an object. Weak pointers are used to break the circular dependency created by shared pointers.

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

5b)

We use assert statements to ensure the condition we are looking for is true. Otherwise, the function can cause undefined behavior or crash, so we abort the execution of the function using the assert statement.

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

5c)

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

5d)

Quick sort has complexity O(n^2) and merge sort O(nlogn). Quick sort is very fast for smaller arrays while merge sort is faster over increasing array sizes.

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

5e)

Practically, it is impossible to have an n-ary tree with all nodes having the same degree since this will lead to an infinite data structure. The notion of empty trees allow a brute-force way to define the definite degree to a tree and allow us to create an n-ary tree without an infinite structure. Whenever, a node does not have a child node, we replace it with an empty tree.

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

5f)

We use std::optional<>

g. What does amortized analysis show? (4)

5g)

Amortized analysis accounts for the worst case scenario for an algorithm or data structure. It averages the intensive operations and tells us the behavior for the worst case. The importance of this is, the intensive operations might be seldom used in the data structure so regarding the total behavior will lead to a wrong analysis of the behavior of the data structure. Averaging balances this act and gives us a proper notion - a weighted average which signifies the intensity and complexity of each statement of the data structure.

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

5h)

Load factors are used to alter the size of objects during runtime. It utilises a ratio of empty space to total space and uses this to decide whether the size needs to be increased or decreased. For example, a load factor of 1 means we should double the space for more objects to be stored whereas, a load factor of one-quarter means, we can halve the space.

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

5i)

Test whether each of the iterator attributes are the same and return the value by and-ing all of them.

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

5j)

When we have an interface, to tack the execution of actions, inputs and outputs we use state machines. This gives us a visual description of the functionality of the program and the flow of the program execution.

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

5f)

We use std::optional<>

g. What does amortized analysis show? (4)

5g)

Amortized analysis accounts for the worst case scenario for an algorithm or data structure. It averages the intensive operations and tells us the behavior for the worst case. The importance of this is, the intensive operations might be seldom used in the data structure so regarding the total behavior will lead to a wrong analysis of the behavior of the data structure. Averaging balances this act and gives us a proper notion - a weighted average which signifies the intensity and complexity of each statement of the data structure.

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

5h)

Load factors are used to alter the size of objects during runtime. It utilises a ratio of empty space to total space and uses this to decide whether the size needs to be increased or decreased. For example, a load factor of 1 means we should double the space for more objects to be stored whereas, a load factor of one-quarter means, we can halve the space.

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

5i)

Test whether each of the iterator attributes are the same and return the value by and-ing all of them.

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

5j)

When we have an interface, to tack the execution of actions, inputs and outputs we use state machines. This gives us a visual description of the functionality of the program and the flow of the program execution.