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: Ruffin Remad **Your student id:** 103840173

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 1 (34 marks)

Consider the following template class `List`.

```
#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

  // Problem 1
  ~List() noexcept                                     // destructor
  {}

  // copy semantics
  List( const List& aOther );                          // copy constructor
  List& operator=( const List& aOther );               // copy assignment

  // move semantics
  List( List&& aOther ) noexcept;                      // move constructor
  List& operator=( List&& aOther ) noexcept;           // move assignment
  void swap( List& aOther ) noexcept;                  // swap elements

  // basic operations
  size_t size() const noexcept;                        // list size

  template<typename U>
  void push_front( U&& aData );                        // add element at front

  template<typename U>
  void push_back( U&& aData );                         // add element at back

  void remove( const T& aElement ) noexcept;           // remove element

  const T& operator[]( size_t aIndex ) const;          // list indexer

  // iterator interface
  Iterator begin() const noexcept;
  Iterator end() const noexcept;
  Iterator rbegin() const noexcept;
  Iterator rend() const noexcept;
};
```

Template class `List` defines an abstract data type lists that is identical to the one defined in Problem Set 4, except that here `List` explicitly defines destructor `~List()`.

The destructor is marked **`noexcept`** to mean that the destructor must not fail. If the destructor tries to exit with an exception, then the program has to terminate. We cannot recover from a bad design error.

The body of the destructor is currently empty. That is, it defines the default behavior when a `List` object goes out of scope. In general, objects in C++ are destroyed in reverse order of

creation. So, even though the body of the destructor is empty, the C++ compiler creates code that first frees `fTail` and then `fHead`. Consider the following test function:

```
void runP1()
{
  using CardinalList = List<size_t>;

  std::cout << "Test List Destructor:" << std::endl;

  CardinalList lList;

  for (size_t i = 1; i <= 1000; i++)
  {
    lList.push_back(i);
  }

  std::cout << "List with " << lList.size()
            << " elements goes out of scope." << std::endl;
}
```

Function `runP1()` creates a list of unsigned integers with 1,000 elements, 1 .. 1,000. The output on Windows looks like the following

```
Test List Destructor:
List with 1000 elements goes out of scope.
Node [1] destroyed.
Node [2] destroyed.
Node [3] destroyed.
Node [4] destroyed.
Node [5] destroyed.
Node [6] destroyed.
…
Node [526] destroyed.
Node [527] destroyed.
Node [
```

The list elements are freed from the top. However, around element 528 a stack overflow exception occurs and the application terminates.

The stack overflow occurs while destroying `fHead`. Destroying `fHead` reduces the reference count of the first list element to 0. It can now be freed. However, this causes a recursive call sequence along the next links. In order to complete the destruction of the first element, we need to delete the second element, if it exists. The reference count of the second element becomes 0, so it can be deleted. Deleting the second element requires the destruction of the third element, if it exists. The reference count of the third element becomes 0, so it can be deleted. Deleting the third element requires the destruction of the fourth element, if it exists, and so on. This process stops at the last element. In order words, default destruction of list elements follows a recursive top-down approach. It works for small lists, but it can produce a stack overflow on large lists if the depth of the call sequence exceeds the operational stack size. (You may not experience a stack overflow exception on macOS, as macOS application have a much larger stack memory than Windows applications. But the issue is the same.)

Hence, we cannot rely on the default C++ implementation. A possible solution is an iterative approach that destroys the list elements along the previous links starting from `fTail`:

- Assign `fTail` to a temporary shared pointer, say `lCurrent`.
- Reset `fTail`.
- While `lCurrent` is not a null pointer, assign the node linked via previous link to a shared pointer `lPrevious`.
- Reset `lPrevious`'s next node, or reset `fHead` if `lPrevious` is a null pointer.
- At the end of the loop, assign `lPrevious` to `lCurrent`, which triggers the deletion of the current list node.

Running function `runP1()` should create the following output

```
Test List Destructor:
List with 1000 elements goes out of scope.
Node [1000] destroyed.
Node [999] destroyed.
Node [998] destroyed.
Node [997] destroyed.
Node [996] destroyed.
Node [995] destroyed.
…
Node [3] destroyed.
Node [2] destroyed.
Node [1] destroyed.
1 test(s) completed.
```

The complete source code of `DoublyLinkedList`, `DoublyLinkedListIterator`, and `List` (except the destructor) is available on Canvas.

Implement the destructor `~List()`. You can use the space below, or complete the code in `List.h`:

```
~List() noexcept
{
```

1)

```
    Node lCurrent = fTail;
    fTail.reset();

    Node lPrevious;
    while (lCurrent != nullptr) {
      lPrevious = lCurrent->fPrevious.lock();
      if (lPrevious) {
        lPrevious->fNext.reset();
      }
      else {
        fHead.reset();
      }
      lCurrent = lPrevious;
    }
```

```
}
```

Problem 2 (130 marks)

When studying amortized analysis, we investigated a dynamic `Stack` class and showed that the average amortized cost of insertion and deletion is O(1).

In this problem, you define a dynamic `Queue` class in which the average amortized cost of insertion and deletion is also O(1).

A queue is a linear data structure, where elements are inserted at one end and elements are deleted at the other end. This leads to the familiar *first-in-first-out* policy. The standard methods for queue manipulation are `enqueue()` – to insert an element at the end and `dequeue()`– to remove the first element. In addition, we use `top()` to access the first element of the queue.

There are many ways to implement a queue. Here, the focus is only on the basic behavior of a queue and an implementation that yields O(1) for the average amortized cost of insertion and deletion. The specification of `DynamicQueue` is as follows

```
#pragma once

#include <optional>
#include <cassert>

template<typename T>
class DynamicQueue
{
private:
  T* fElements;                         // queue elements
  size_t fFirstIndex;                   // first element in queue
  size_t fLastIndex;                    // next free slot
  size_t fCurrentSize;                  // size of queue

  void resize(size_t aNewSize);                              // 50
  void ensure_capacity();                                    // 12
  void adjust_capacity();                                    // 16

public:
  DynamicQueue();                                            // 12
  ~DynamicQueue();                                           // 4

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

  std::optional<T> top() const noexcept;                     // 18
  void enqueue(const T& aValue);                             // 8
  void dequeue();                                            // 10
};
```

`DynamicQueue` objects are not copyable. The copy semantics is disabled.

Objects of class `DynamicQueue` maintain four instance variables: `fElements` – a raw pointer to heap memory for queue elements, `fFirstIndex` – an index to the first element in the queue, `fLastIndex` – an index to the next free entry in the queue, and `fCurrentSize` – the current size of the queue.

There are two major parts in the queue implementation: the standard queue operations plus construction and destructions of `DynamicQueue` objects and the resize logic. To enqueue an item into the queue, it must have sufficient space. Here, we use a naïve approach. If `fLastIndex` equals the size of the queue, then the queue must be resized using a suitable new size. Similarly, after dequeuing an item from the queue, the queue's size may have to be adjusted. In this case, the difference between `fLastIndex` and `fFirstIndex` serves as indicator if resizing to a new suitable size is required. Please note that resize always creates

a queue state in which `fFirstIndex` equals zero and `fLastIndex` is set to the next available slot in the queue. The following two figures illustrate what should happened when a resize is required.

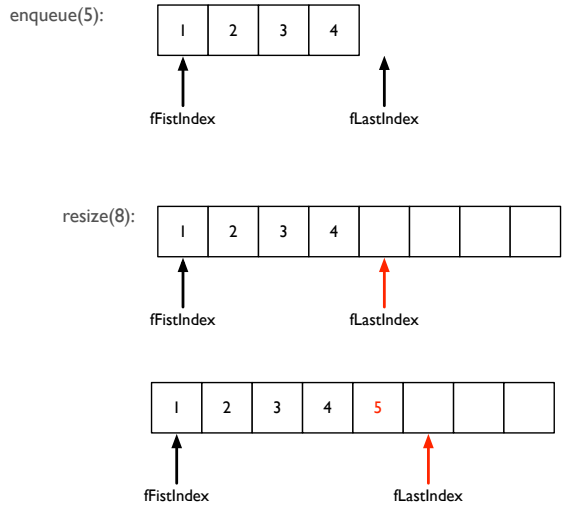

Figure 1: Enqueue with resize.

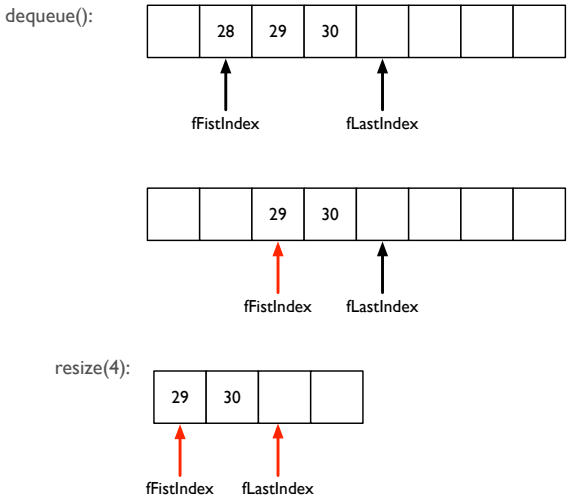

Figure 2: Dequeue with resize.

Consider the following test function:

```
void runP2()
{
    DynamicQueue<int> lQueue;

    std::cout << "Enqueue elements:" << std::endl;

    for (int i = 1; i <= 30; i++)
    {
        lQueue.enqueue(i);
        std::cout << i << std::endl;
    }

    std::cout << "Dequeue elements:" << std::endl;

    std::optional<int> lValue = lQueue.top();

    while ( lValue )
    {
        std::cout << *lValue << std::endl;
        lQueue.dequeue();
        lValue = lQueue.top();
    }

    if ( lValue )
        std::cout << "failure" << std::endl;
    else
        std::cout << "success" << std::endl;
}
```

Function `runP2()` creates a queue, enqueues 30 items, and dequeues all items. When implemented correctly, the dynamic queue `lQueue` must increase its size five times while enqueuing elements to reach a final queue size of 32. When dequeuing elements, `lQueue` must resize its size also five times and finish with a size of 1.

Implement template class `DynamicQueue`. You can use the space below, or extend the code in `DynamicQueue.h` provided on Canvas.

```
#pragma once

#include <optional>
#include <cassert>

template<typename T>
class DynamicQueue
{
private:
  T* fElements;                 // queue elements
  size_t fFirstIndex;           // first element in queue
  size_t fLastIndex;            // next free slot
  size_t fCurrentSize;          // size of queue

  void resize(size_t aNewSize)
  {
```

2a)

```
    T* lNewElements = new T[aNewSize];
    size_t j = 0;
    for (size_t i = fFirstIndex; i < fLastIndex; ++i, ++j) {
      lNewElements[j] = std::move(fElements[i]);
    }
    delete[] fElements;
    fElements = lNewElements;
    fFirstIndex = 0;
    fLastIndex = j;
    fCurrentSize = aNewSize;
```

```
  }

  void ensure_capacity()
  {
```

2b)

```
    if (fLastIndex >= fCurrentSize) {
      resize(fCurrentSize * 2);
    }
```

```
  }
```

```
void adjust_capacity()
{
```

2c)

```
    if ((fLastIndex - fFirstIndex) <= fCurrentSize / 4 && fCurrentSize > 1) {
        resize(fCurrentSize / 2);
    }
```

```
}

public:
```

2d)

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

```
~DynamicQueue()
{
```

2e)

```
delete[] fElements;
```

```
}

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

std::optional<T> top() const noexcept
{
```

2f)

```
    if (fFirstIndex == fLastIndex) {
        return std::nullopt;
    }
    return fElements[fFirstIndex];
```

```
}
```

```
    void enqueue(const T& aValue)
    {
```

2g)

```
        ensure_capacity();
        fElements[fLastIndex++] = aValue;
```

```
    }

    void dequeue()
    {
```

2h)

```
        if (fFirstIndex < fLastIndex) {
            ++fFirstIndex;
            adjust_capacity();
        }
```

```
    }
};
```

Problem 3: (114 marks)

Iterators provide a method for systematic traversal of a collection. We have seen numerous examples of iterators in this semester.

In this problem, you define an iterator over strings that allows us to check if the source string is a palindrome. A palindrome is a word, number, phrase, or any sequence of symbols that reads the same forward as backward. In this problem, you check a phrase written by the Scottish poet Alastair Reid:

> "*T. Eliot, top bard, notes putrid tang emanating, is sad; I'd assign it a name:*
>
> *gnat dirt upset on drab pot toilet.*"

At the first glance it is difficult to say, whether this phrase is a palindrome. To test this hypothesis, you implement `PalindromeStringIterator` in this problem. The following test function defines the test:

```
void runP3()
{
  // Palindrome by Scottish poet Alastair Reid
  std::string lTest =
      "T. Eliot, top bard, notes putrid tang emanating, is sad;\
       I'd assign it a name: gnat dirt upset on drab pot toilet.";

  std::cout << "Test palindrome iterators:" << std::endl;

  PalindromeStringIterator lIter(lTest);
  PalindromeStringIterator lForward = lIter.begin();
  PalindromeStringIterator lBackward = lIter.rbegin();

  while (lForward != lForward.end() && lBackward != lBackward.rend())
  {
    if (*lForward != *lBackward)
    {
      break;
    }

    ++lForward;
    lBackward--;
  }

  if (lForward == lForward.end() && lBackward == lBackward.rend())
  {
    std::cout << "String is a palindrome." << std::endl;
  }
  else
  {
    std::cout << "Oops. Something went wrong." << std::endl;
  }
}
```

If `PalindromeStringIterator` is correctly defined, then function `runP3()` produces the following output:

```
Test palindrome iterators:
String is a palindrome.
```

Alastair Reid's phrase is a palindrome.

For `functionP3()` to work, the iterator has to travers the whole source string and only return letters in upper case. All other characters, that is, punctuation, spaces, and digits are skipped. In addition, traversing the source string simultaneously forward and backward must end at the same instant. That is, the corresponding iterators (`lForward` and `lBackward`) must reach their respective end position at the same time.

A suggested specification for `PalindromStringIterator` is given below:

```
#pragma once

#include <string>
#include <cctype>

class PalindromeStringIterator
{
private:
  std::string fString;
  int fIndex;

  void moveToNextIndex();                                                    // 18
  void moveToPreviousIndex();                                                // 14

public:
  PalindromeStringIterator(const std::string& aString);                      // 6

  char operator*() const noexcept;                                           // 6

  PalindromeStringIterator& operator++() noexcept;                           // 4
  PalindromeStringIterator operator++(int) noexcept;

  PalindromeStringIterator& operator--() noexcept;                           // 4
  PalindromeStringIterator operator--(int) noexcept;

  bool operator==(const PalindromeStringIterator& aOther) const noexcept;    // 10
  bool operator!=(const PalindromeStringIterator& aOther) const noexcept;

  PalindromeStringIterator begin() const noexcept;                           // 12
  PalindromeStringIterator end() const noexcept;                             // 14
  PalindromeStringIterator rbegin() const noexcept;                          // 16
  PalindromeStringIterator rend() const noexcept;                            // 10
};
```

`PalindromeStringIterator` is a bi-directional iterator for `std::string` objects. In order to provide support for both, forward and backward iteration, defines increment and decrement operators, respectively. As per convention, the iterator also provides an operator to access the element at the current iterator position and the required equivalence predicates. Please note that the deference operator has to return only upper case letters. Use the corresponding `cctype` function for this purpose.

The increment and decrement operators cannot just move along the corresponding direction. Every time the iterator moves into a new position, it has to be positioned at an alphabetic character. The methods `moveToNextIndex()` and `moveToPreviousIndex()` perform the corresponding tasks. The method `moveToNextIndex()` uses a while-loop to increment the iterator index if the iterator index is not at the forward end position. The loop stops, if the iterator is positioned at an alphabetic character. Use the corresponding `cctype` function here.

Method `moveToPreviousIndex()` works similarly, but it has to decrement the iterator index and stops when the iterator index is at the backward end position.

Please note that you need to use static casts to **`int`** in order to use the iterator index in combination with the string length (which is of type `size_t`). In addition, the forward direction starts with the iterator index positioned before the first character, whereas the backward direction positions the iterator index initially after the last character.

The most important aspect of the bi-directional `PalindromeStringIterator` iterator is a set of auxiliary methods that yield new iterators (i.e., copies of the original iterator), which are positioned on (a) the first element, (b) the last element, (c) before the first element to the left, and (d) past the last element to the right. We can capture these four scenarios by

the factory methods `begin()`, `rbegin()`, `rend()`, and `end()`.

You can use the space below, or extend the file `PalindromeStringIterator.cpp` provided on Canvas.

```
#include "PalindromeStringIterator.h"

void PalindromeStringIterator::moveToNextIndex()
{
```

3a)

```
   while (fIndex < static_cast<int>(fString.length()) && !std::isalpha(fString[fIndex]))
   {
      ++fIndex;
   }
```

```
}

void PalindromeStringIterator::moveToPreviousIndex()
{
```

3b)

```
   while (fIndex >= 0 && !std::isalpha(fString[fIndex]))
   {
      --fIndex;
   }
```

```
}
```

3c)

```
PalindromeStringIterator::PalindromeStringIterator(const std::string& aString)
 : fString(aString), fIndex(0)
{
   moveToNextIndex();
}
```

```
char PalindromeStringIterator::operator*() const noexcept
{
```

3d)

```
  return std::toupper(fString[fIndex]);
```

```
}

PalindromeStringIterator& PalindromeStringIterator::operator++() noexcept
{
```

3e)

```
  ++fIndex;
  moveToNextIndex();
  return *this;
```

```
}

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

  ++(*this);

  return old;
}

PalindromeStringIterator& PalindromeStringIterator::operator--() noexcept
{
```

3f)

```
  --fIndex;
  moveToPreviousIndex();
  return *this;
```

```
}

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

  --(*this);

  return old;
}

bool PalindromeStringIterator::operator==(const PalindromeStringIterator& aOther)
 const noexcept
{
```

3g)

```
  return fIndex == aOther.fIndex;
```

```
}
```

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

PalindromeStringIterator PalindromeStringIterator::begin() const noexcept
{
```

3h)

```
  PalindromeStringIterator iter(*this);
  iter.fIndex = 0;
  iter.moveToNextIndex();
  return iter;
}

PalindromeStringIterator PalindromeStringIterator::end() const noexcept
{
```

3i)

```
  PalindromeStringIterator iter(*this);
  iter.fIndex = fString.length();
  return iter;
}

PalindromeStringIterator PalindromeStringIterator::rbegin() const noexcept
{
```

3j)

```
  PalindromeStringIterator iter(*this);
  iter.fIndex = static_cast<int>(fString.length()) - 1;
  iter.moveToPreviousIndex();
  return iter;
}

PalindromeStringIterator PalindromeStringIterator::rend() const noexcept
{
```

3k)

```
  PalindromeStringIterator iter(*this);
  iter.fIndex = -1;
  return iter;
}
```

Problem 4 (68 marks)

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 is a smart pointer that does not increase the life of the object it refers to. It is used to watch an object without assuming ownership of it, preventing difficulties such as circular references, which occur when two or more objects have strong relationships to one other and cannot be deleted.

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

4b)

We can guarantee preconditions for operations in C++ by using assertions (assert), exceptions, or error codes to validate input values and object states before executing the operation.

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

4c)

The canonical methods in C++ are the copy constructor, copy assignment operator, move constructor, move assignment operator, and destructor. These methods are essential for managing the lifetime and ownership of objects, especially when dealing with resources.

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

4d)

No, Quick Sort is not strictly better than Merge Sort. Quick Sort is often faster in practice, but its worst-case time complexity is O(n^2), while Merge Sort consistently has O(n log n) time complexity.

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

4e)

The purpose of an empty tree is to serve as a base case for recursive tree algorithms. It simplifies the implementation of operations like insertion, deletion, and traversal by providing a well-defined starting point and termination condition.

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

4f)

The modern C++ abstraction used to return a value that does not exist is std::optional. It allows you to represent the presence or absence of a value, avoiding the need for special values like null pointers.

g. What does amortized analysis show? (4)

4g)

Amortized analysis shows the average cost of a sequence of operations over time. It helps analyze algorithms where individual operations might have varying costs, but the overall average cost 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 in a hash table is the number of elements divided by the capacity; recommended factors are 0.5 to 0.7, with expansion at 0.75 and contraction at 0.25 to maintain performance and memory efficiency.

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

4i)

To test the equivalence of iterators, we need to compare their underlying positions or the elements they point to. This can be done using the equality operator (==) or by comparing the memory addresses of the elements they reference.

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

4j)

We need to implement a state machine when modeling systems or processes that have distinct states and transitions between those states based on events or inputs. State machines provide a structured way to represent and manage complex behavior.