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: Ekrar Efaz **Your student id:** 103494172

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
									X		

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.

Problem 1 (24 marks)

Consider the template class `List` that we implemented in Problem Set 3. It defines an "object adapter" for `DoublyLinkedList` objects, that is, template class `List` uses as representation objects of type `DoublyLinkedList`.

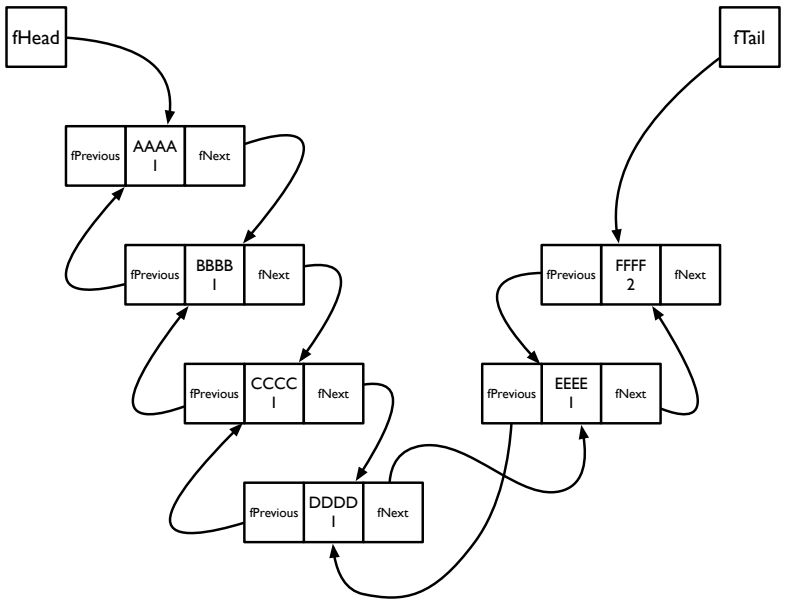

Figure 1: The List structure with reference counts (under the payload label).

The actual list elements are anchored at two ends: `fHead` – pointing at the first element in the list and `fTail` – pointing at the last element in the list. The member variables `fHead` and `fTail` are smart pointers (i.e., `std::shared_ptr<DoublyLinkedList<T>>`, where type `T` refers to the payload type of list elements). This also applies to all individual list nodes.

Using smart pointers freed us from defining explicit memory management, or so we thought. We let C++ synthesize the destructor and it worked as expected. When objects of type `List` went out of scope, then the destructor freed all list elements. This, however, only works for small lists. If the number of list elements exceeds approx. 600 elements on Windows (and Visual Studio), then suddenly the destructor fails with a stack overflow. This is a result of the default implementation of the destructor.

In general, objects in C++ are destroyed in reverse order of creation. When a `List` object is being destroyed, first the smart pointer `fTail` is freed. If the list is not empty, then the last element in the list has a reference count of 2. Freeing `fTail` reduces the reference count to 1. The element cannot be destroyed yet. It has either a previous node or `fHead` is pointing at it.

The List destructor moves to `fHead` now. Freeing `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 free the first element, we first 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 can produce a stack overflow on large lists if the depth of the call sequence exceeds the operational stack size.

We need an iterative bottom up technique to free list elements properly. We start at `fTail` and assign `fTail` to a temporary variable, say `lCurrent`. Then we immediately reset `fTail`, which corresponds to the first cut in Figure 2. This reduces the reference count of the last list element to 1, if it exists. Please note that calling `reset()` on a shared pointer always works. Now, while `lCurrent` is not a null pointer, we follow the previous link and reset its next node (cuts 2 to 6), or reset `fHead` (cut 7) if there is no previous node. The first list node does not have a previous node. At the end of the loop `lCurrent` becomes the previous of `lCurrent` triggering the deletion of the current list node (its reference count has gone down to 0). Remember, the previous node is defined as a weak smart pointer. We need to turn it to a shared pointer first before we can perform any operations on it.

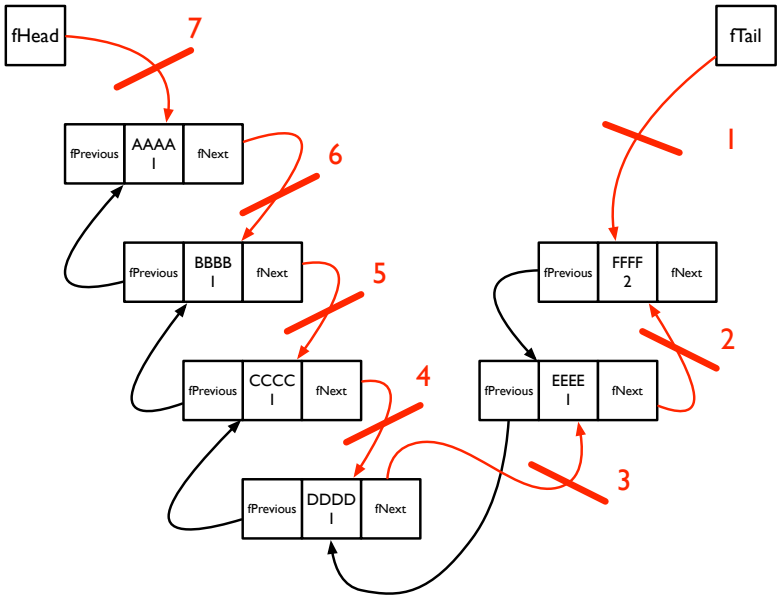

Figure 2: Freeing list elements by cutting the next links iteratively from `fTail` to `fHead`.

Using this approach, the destructor of `List` never generates a stack overflow. We use the C++ scoping rules to free the list elements iteratively in a bottom up fashion, that is, from the last to the first element.

The test code

```
void testP1()
{
  List<std::string> lList;

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

  lList.push_back( "DDDD" );
  lList.push_front( "CCCC" );
  lList.push_back( "EEEE" );
  lList.push_front( "BBBB" );
  lList.push_back( "FFFF" );
  lList.push_front( "AAAA" );

  for ( auto& item : lList )
  {
    std::cout << item << std::endl;
  }

  std::cout << "End of scope, free lList:" << std::endl;
}
```

should produce the following output without errors:

```
Test list Destructor:
AAAA
BBBB
CCCC
DDDD
EEEE
FFFF
End of scope, free lList:
Delete Node FFFF
Delete Node EEEE
Delete Node DDDD
Delete Node CCCC
Delete Node BBBB
Delete Node AAAA
```

The output "`Delete Node XXXX`" is generated by the destructor of `DoublyLinkedList`:

```
~DoublyLinkedList()
{
  std::cout << "Delete Node " << fData << std::endl;
}
```

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 extend the code in `List.h`:

```
~List()
{
```

1)

```
// Final Exam, 2023
~List()
   {
      Node lCurrent = fTail;
      fTail.reset(); // reduce reference count of last element

      while (lCurrent)
      {
         auto prevNode = lCurrent->fPrevious.lock(); // get shared_ptr to previous
node
         if (prevNode)
         {
            prevNode->fNext.reset(); // cut the link to current node
         }
         else
         {
            fHead.reset(); // if there is no previous node, reset fHead
         }

         lCurrent = prevNode; // move to the previous node
      }
   }
```

```
}
```

Problem 2 (88 marks)

We wish to define a generic 3-ary tree in C++. We shall call this data type `TernaryTree`. A 3-ary tree has a key `fKey` and three subtrees, or nodes. Following the principles underlying the definition of general trees, a 3-ary tree is a finite set of nodes and it is either

- an empty set, or
- a set that consists of a root and exactly 3 distinct 3-ary subtrees.

In this problem, we are solely interested in the basic infrastructure of 3-ary trees. We ignore copy and move semantics.

A possible specification for `TernaryTree`, a template class, is given below:

```
#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;        // 3
  TernaryTree( T&& aKey ) noexcept;                   // 4

  template<typename... Args>
  static Node makeNode( Args&&... args );             // 7

  const T& operator*() const noexcept;                // 2

  TernaryTree& operator[]( size_t aIndex ) const;     // 12

  void add( size_t aIndex, Node& aNode );             // 12
  Node remove( size_t aIndex );                       // 12

  bool leaf() const noexcept;                         // 14
  size_t height() const noexcept;                     // 22

private:

  T fKey;
  Node fNodes[3];
};
```

The test code

```
void testP2()
{
  using TTree = TernaryTree<std::string>;
  using TTreeNode = typename TTree::Node;

  std::cout << "Test TTree:" << std::endl;

  std::string lB( "B" );

  TTreeNode nA = TTree::makeNode( "A" );
  TTreeNode nB = TTree::makeNode( lB );
  TTreeNode nC = TTree::makeNode( std::string( "C" ) );
  TTreeNode nD = TTree::makeNode( "D" );
```

```
  nC->add( 2, nD );
  nB->add( 0, nC );
  nA->add( 1, nB );

  if ( !nC )
  {
    std::cout << "Ownership control sound." << std::endl;
  }
  else
  {
    std::cout << "Ownership control broken." << std::endl;
  }

  TTreeNode& root = nA;

  std::cout << "root:\t\t" << **root << std::endl;
  std::cout << "root[1]:\t" << *(*root)[1] << std::endl;
  std::cout << "root[1][0]:\t" << *(*root)[1][0] << std::endl;
  std::cout << "root[1][0][2]:\t" << *(*root)[1][0][2] << std::endl;

  std::cout << "height of root: " << root->height() << std::endl;
  (*root)[1][0].remove( 2 );
  std::cout << "height of root: " << root->height() << std::endl;

  std::cout << "TTree test complete." << std::endl;
}
```

should produce the following output without errors:

```
Test TTree:
Ownership control sound.
root:           A
root[1]:        B
root[1][0]:     C
root[1][0][2]:  D
height of root: 3
height of root: 2
TTree test complete.
```

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

2)

```
// 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),
    fNodes{nullptr, nullptr, nullptr}
    {}

    TernaryTree( T&& aKey ) noexcept:
    fKey (std::move(aKey)),
    fNodes{nullptr, nullptr, nullptr}
    {}

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

    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 && aNode); // check if valid index and node
        fNodes[aIndex] = std::move(aNode);
    }
    Node remove( size_t aIndex ){
        assert( aIndex < 3 );
        return std::move(fNodes[aIndex]);
    }
```

2)

```
// check above box
```

Problem 3 (4+9+3+12+4+9+9+6+4=60 marks)

We wish to define our own simple string data type in C++. We shall call this data type `DSPString`. We further wish to implement `DSPString` as a standard C++ class with a private instance variable and public methods. Somebody else has already started with the implementation, but left the project unfinished. Complete the following code fragments. In particular, make sure that both the class definition and the methods are correctly defined. Internally, a `DSPString` is an array of characters terminated by the zero character (i.e., `'\0'`). Consequently, the storage size for a `DSPString` is the length of the string plus one for the terminator.

Use the provided code fragments to infer the missing declarations for given function implementations. You can use the space below, or extend the files `DSPString.h` and `DSPString.cpp` provided on Canvas. You can only use the features defined directly in `DSPString` or included from the header files `ostream`, `algorithm`, and `cassert`. The use of features defined for C-strings is not permitted.

Header DSPString.h:

```
#pragma once

#include <ostream>

class DSPString
{
private:
```

3a)

```
char* fContents;
```

```
public:

  DSPString( const char* aContents = "\0" );

  ~DSPString();

  // copy semantics
  DSPString( const DSPString& aOther );
  DSPString& operator=( const DSPString& aOther );

  // move semantics
  DSPString( DSPString&& aOther ) noexcept;
  DSPString& operator=( DSPString&& aOther ) noexcept;
```

3b)

```
size_t size() const noexcept;
char operator[]( size_t aIndex ) const noexcept;
```

```
  bool operator==( const DSPString& aOther ) const noexcept;

  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const DSPString& aObject );
};
```

Compilation unit DSPString.cpp:

```
#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];
  }
}

DSPString::~DSPString()
{
```

3c)

```
delete[] fContents;  //3C
```

```
}

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

DSPString& DSPString::operator=( const DSPString& aOther )
{
```

3d)

```
if( this != &aOther ){
     new (this) DSPString(aOther);
  }
  return *this;
```

```
}

DSPString::DSPString( DSPString&& aOther ) noexcept :
       DSPString( "\0" )
{
```

3e)

```
std::swap( fContents, aOther.fContents );
```

```
}
```

```
DSPString& DSPString::operator=( DSPString&& aOther ) noexcept
{
```

3f)

```
if( this != &aOther ){
      std::swap( fContents, aOther.fContents );
   }
   return *this;
```

```
}

size_t DSPString::size() const noexcept
{
```

3g)

```
return strlen( fContents );
```

```
}

char DSPString::operator[]( size_t aIndex ) const noexcept
{
```

3h)

```
assert( aIndex < size() );
 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& aOStream, const DSPString& aObject )
{
```

3i)

```
aOStream << aObject.fContents;
return aOStream;
```

```
}
```

The test code

```
void testP3()
{
  std::cout << "Test DSPString:" << std::endl;

  DSPString lS1;
  DSPString lS2( "Problem 3" );

  std::cout << "S1: \'" << lS1 << "\'" << std::endl;
  std::cout << "S2: \'" << lS2 << "\'" << std::endl;

  DSPString lS3 = lS2;
  lS1 = lS2;

  std::cout << "S1: \'" << lS1 << "\'" << std::endl;
  std::cout << "S3: \'" << lS3 << "\'" << std::endl;

  DSPString lS4 = std::move( lS2 );
  lS1 = std::move( lS3 );

  std::cout << "S1: \'" << lS1 << "\'" << std::endl;
  std::cout << "S4: \'" << lS4 << "\'" << std::endl;

  if ( !(lS3 == lS4) || lS4 == lS2 )
  {
    std::cout << "There is something wrong." << std::endl;
  }
  else
  {
    std::cout << "String comparison correct." << std::endl;
  }

  std::cout << "DSPString test complete." << std::endl;
}
```

should produce the following output without errors:

```
Test DSPString:
S1: ''
S2: 'Problem 3'
S1: 'Problem 3'
S3: 'Problem 3'
S1: 'Problem 3'
S4: 'Problem 3'
String comparison correct.
DSPString test complete.
```

Problem 4: (6+5+5+8+7+10+11+8=60 marks)

Assume a correct implementation of class `DSPString`. We now wish to define a bi-directional iterator for objects of class `DSPString`. In order to provide support for both, forward and backward iteration, we use the postfix increment and decrement operators. As per convention, we also have to provide an operator to access the element at the current iterator position and the required equivalence predicates.

The most important aspect of the bi-directional `DSPStringIterator` 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()`. We use the following specification for the bi-directional iterator:

```
#pragma once

#include <memory>

#include "DSPString.h"

class DSPStringIterator
{
private:
  std::shared_ptr<DSPString> fCollection;
  int fIndex;

public:

  DSPStringIterator( const DSPString& aCollection );

  char operator*() const noexcept;

  DSPStringIterator& operator++() noexcept;
  DSPStringIterator operator++( int ) noexcept;

  DSPStringIterator& operator--() noexcept;
  DSPStringIterator operator--( int ) noexcept;

  bool operator==( const DSPStringIterator& aOther ) const noexcept;
  bool operator!=( const DSPStringIterator& aOther ) const noexcept;

  DSPStringIterator begin() const noexcept;
  DSPStringIterator end() const noexcept;
  DSPStringIterator rbegin() const noexcept;
  DSPStringIterator rend() const noexcept;
};
```

We are using a smart pointer to store the underlying collection. This allows for controlled sharing and proper testing of the collection. Note that all iterator methods must be defined in a type safe manner. Suitable typecasts are required at times.

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

```
#include "DSPStringIterator.h"

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

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

4a)

```
  assert(fIndex < fCollection->size());
  return (*fCollection)[fIndex];
}

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

4b)

```
  assert(fIndex < fCollection->size());
  fIndex++;
  return *this;
}

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

  ++(*this);

  return old;
}

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

4c)

```
  assert( ! (fIndex < 0));
  fIndex--;
  return *this;
}

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

  --(*this);

  return old;
}

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

4d)

```
  return fIndex == aOther.fIndex && fCollection == aOther.fCollection;
}

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

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

4e)

```
  DSPStringIterator copyIterator = *this;
  copyIterator.fIndex = 0;

  return copyIterator;
}
```

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

4f)

```
  DSPStringIterator copyIterator = *this;
  copyIterator.fIndex = fCollection->size();

  return copyIterator;
}
```

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

4g)

```
  DSPStringIterator copyIterator = *this;
  copyIterator.fIndex = fCollection->size() - 1;

  return copyIterator;
}
```

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

4h)

```
  DSPStringIterator copyIterator = *this;
  copyIterator.fIndex = -1;

  return copyIterator;
}
```

The test code

```
void testP4()
{
  std::cout << "Test DSPString iterator:" << std::endl;

  DSPString lString( "Glenelg" );

  std::cout << "Forward iteration: ";

  for ( char c : DSPStringIterator( lString ) )
  {
    std::cout << c;
  }

  std::cout << std::endl;
```

```
  std::cout << "Backwards iteration: ";

  DSPStringIterator iter = DSPStringIterator( lString ).rbegin();

  for ( ; iter != iter.rend(); --iter )
  {
    std::cout << *iter;
  }

  std::cout << std::endl;

  std::cout << "DSPString complete test complete." << std::endl;
}
```

should produce the following output without errors:

```
Test DSPString iterator:
Forward iteration: Glenelg
Backwards iteration: glenelG
DSPString complete test complete.
```

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)

A weak pointer is a specialized type of pointer that offers a reference to an object controlled by a shared pointer, without increasing the reference count. Its main purpose is to observe or access objects without extending their lifetimes. It doesn't keep the object alive.

In C++, weak pointers are utilized to address circular dependencies, observer patterns, and safeguard against dangling pointers. They enable the deallocation of objects in the presence of circular references, provide non-owning access to objects for observation or retrieval, and ensure secure access to objects when passing pointers between different components or threads.

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

5b)

Using Assertions for Input Validation we can guarantee preconditions for operations in C++.

Assertions are statements incorporated into code to validate specific conditions at designated locations. They serve the purpose of verifying assumptions regarding the program's state or inputs. By employing assertions, programmers can ensure that anticipated conditions are met before executing operations. In the event of an assertion failure where the condition is false, an error or exception is triggered, signaling a violation of the expected preconditions.

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

5c)

In the realm of C++, the canonical methods are a set of member functions that establish a standardized interface for governing the lifespan and behavior of objects. These methods, typically incorporated within classes, adhere to a prescribed naming convention and encompass the following:

Default Constructor: This constructor initializes an object with default values. It becomes active when an object is instantiated without any arguments.

Copy Constructor: With this constructor, a new object is generated by replicating the state of an existing object. It enables the creation of an object that mirrors the state of another.

Copy Assignment Operator: This operator facilitates the assignment of one object's state to another object of the same type. It is utilized when an already initialized object necessitates the assignment of a fresh value.

Destructor: The destructor undertakes the responsibility of releasing resources allocated by an object upon its termination or explicit deletion. Its invocation is automatic when an object is destroyed.

Move Constructor: By means of this constructor, the ownership of resources is efficiently transferred from one object to another. It comes into play when an rvalue is utilized to initialize a novel object.

Move Assignment Operator: Similar to the move constructor, this operator serves the purpose of assigning one object's state to another. It enables the efficient transfer of resources.

By integrating these canonical methods into a class, a coherent and predictable interface is established for object construction, copy operations, destruction, and resource management.

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

5d)

There are advantages and disadvantages of both the algorithms and the best algorithm only can be determined with specific use case. The comparison can be made based on the Time Complexity, Space Complexity and the size of input data.

Quick Sort:

Quick Sort is usually faster than Merge Sort for average cases. It has a medium time complexity of O(n log n).
Quick Sort uses less memory than Merge Sort because it can modify the input array directly.
Quick Sort is often faster for smaller or randomly ordered datasets.

Merge Sort:

Merge Sort always guarantees a time complexity of O(n log n), regardless of the input order.
Merge Sort maintains the relative order of equal elements, making it stable.
Merge Sort performs well on larger datasets or datasets with a more balanced structure.

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

5e)

An empty tree, also called a null tree or a tree with no nodes, has several important uses:

Base Case: It provides a stopping point for recursive algorithms on trees. Without it, the algorithms would run indefinitely.
Initialization: It serves as a starting point for creating and setting up a tree structure. Nodes can be added gradually to build the tree.
Placeholder: It represents the absence of elements or a valid tree structure when needed.
Memory Efficiency: An empty tree requires less memory compared to a non-empty tree, which can be beneficial in memory-constrained situations.
Standardization: Including an empty tree as a valid state establishes a consistent representation for trees across different algorithms and implementations.

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

5f)

In modern C++, when we encounter a scenario where we need to return a value that may or may not exist, we utilize the "optional" type as an abstraction. The "optional" type serves as a wrapper that represents either a valid value or no value at all. Its purpose is to handle situations where a function may or may not be capable of producing a result.

The Syntax is as follows:

```
std::optional<Type> Name();
```

g. What does amortized analysis show? (4)

5g)

Amortized analysis is a technique that looks at the average cost of a series of operations, even if some individual operations may be more expensive. It helps us understand how an algorithm or data structure performs over multiple operations, taking into account the balance between high-cost and low-cost operations. This analysis gives us a better understanding of the overall resource usage and performance of the algorithm or data structure.

Amortized analysis helps to understand the behavior of an algorithm or data structure in scenarios where the cost of some operations may be high, but they are offset by other operations with lower costs. It aims to provide a more accurate and comprehensive understanding of the resource utilization and performance characteristics of an algorithm.

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 Factor Factors:

Expansion Load Factor: This is the load factor at which the data structure is considered full, and an expansion of the allocated memory is triggered. It is typically set below 1 to allow for efficient insertion and minimize collisions or performance degradation.

Contraction Load Factor: This is the load factor at which the data structure is considered too empty, and a contraction of the allocated memory is triggered to save space. The contraction load factor is typically set higher than the expansion load factor to avoid frequent resizing and unnecessary memory operations.

Threshold and Aim:

The expansion threshold represents the point at which the number of elements in a data structure reaches a specific limit, indicating that the structure is approaching its capacity. If the number of elements exceeds this threshold, the data structure's capacity is expanded to accommodate additional elements. The expansion threshold can be set as a fixed number or as a percentage of the current capacity.

The contraction threshold represents the point at which the number of elements in a data structure falls below a certain limit, indicating that the structure is becoming relatively empty. If the number of elements falls below this threshold, the data structure's capacity is reduced to conserve memory. Similar to the expansion threshold, the contraction threshold can be defined as a fixed number or as a percentage of the current capacity.

Load Factor Example for Stack ADT:

To manage the changes in size, we utilize a load factor approach. When the stack is initially empty, the load factor is 0, indicating that there is one free slot available. If the stack becomes full, with a load factor of 1, we double its size to accommodate more elements. This results in a stack with a load factor of 1/2.

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

5i)

To check the equivalence of iterators we have to make sure the following requirements are fullfilled:

1. Same Container Comparison is being done.
2. The iterators being compared are at the same position.
3. The values referred to by the iterators should also be equal.
4. The iterators are of the same type (i.e. forward-iterator, backward-iterator etc)

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

5j)

We implement a state machine, when we want to represent and manage systems that have distinct states and transitions between those states.

For Example we used state machine in our implementation of DoublyLinkedList Iterator.

Our iterator had for forward and backward iterators and the doubly-linked list represented a complex enough data structure for which we couldn't manage the state in any other way. Our program had 3 states so that we could implement the forward and backward iterator begin,end,rbegin and rend functions with proper placements and determine if the iterator has finished or not based on its states.