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 7, 2022, 18:00
Lecturer:	Dr. Markus Lumpe

Your name: Nguyen Minh Duy **Your student id:** 104974743

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30

Marker's comments:

Problem	Marks	Time Estimate in minutes	Obtained
1	132	30	
2	56	10	
3	60	15	
4	10+88=98	45	
5	50	20	
Total	396	120	

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

```
// COS30008, Final Exam
// Nguyen Minh Duy - 104974743
// Implementation of a generic TernaryTree class supporting prefix iteration, copy, and move semantics.

#pragma once

#include <stdexcept>
#include <algorithm>

template<typename T>
class TernaryTreePrefixIterator;

template<typename T>
class TernaryTree
{
public:

    using TTree = TernaryTree<T>;
    using TSubTree = TTree*;

private:

    T fKey;
    TSubTree fSubTrees[3];

    // private default constructor used for declaration of NIL
    TernaryTree() :
        fKey(T())
    {
        for (size_t i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL;
        }
    }

public:

    using Iterator = TernaryTreePrefixIterator<T>;

    static TTree NIL;

    // Getters for subtrees
    const TTree& getLeft() const { return *fSubTrees[0]; }
    const TTree& getMiddle() const { return *fSubTrees[1]; }
    const TTree& getRight() const { return *fSubTrees[2]; }

    // Add a subtree to the left, middle, or right position
    void addLeft(const TTree& aTTree) { addSubTree(0, aTTree); }
    void addMiddle(const TTree& aTTree) { addSubTree(1, aTTree); }
    void addRight(const TTree& aTTree) { addSubTree(2, aTTree); }

    // Remove a subtree from the left, middle, or right position
    const TTree& removeLeft() { return removeSubTree(0); }
    const TTree& removeMiddle() { return removeSubTree(1); }
    const TTree& removeRight() { return removeSubTree(2); }

    /////////////////////////////////////////////////////////////////////////
    // Private helper functions for managing subtrees
    // Problem 1: TernaryTree Basic Infrastructure

private:
    // remove a subtree, may throw a domain error [22]
    // Remove a subtree; checks for valid index and throws errors if conditions are violated
    const TTree& removeSubTree(size_t aSubtreeIndex)
    {
        if (aSubtreeIndex >= 3) // Check for valid subtree index
        {
            throw std::out_of_range("Illegal subtree index");
        }

        if (fSubTrees[aSubtreeIndex]->empty()) // Check if the subtree is NIL
        {
            throw std::domain_error("Subtree is NIL");
        }

        const TTree& Outcome = *fSubTrees[aSubtreeIndex]; // Save subtree for return
        fSubTrees[aSubtreeIndex] = &NIL;                // Set the subtree to NIL

        return Outcome;                                  // Return the removed subtree
    }

    // add a subtree; must avoid memory leaks; may throw domain error [18]
    // Add a subtree; ensures no memory leaks and validates subtree conditions
    void addSubTree(size_t aSubtreeIndex, const TTree& aTTree)
    {
        if (aSubtreeIndex >= 3) // Check for valid subtree index
        {
            throw std::out_of_range("Illegal subtree index");
        }

```

```cpp
        if (!fSubTrees[aSubtreeIndex]->empty()) // Ensure the position is currently NIL
        {
            throw std::domain_error("Subtree is not NIL");
        }

        fSubTrees[aSubtreeIndex] = const_cast<TSubTree>(&aTTree); // Add the subtree
    }

    /////////////////////////////////////////////////////////////////////////
    // Public constructors, destructor, and utility methods

public:
    // TernaryTree l-value constructor [10]
    // Constructor for l-value keys
    TernaryTree(const T& aKey) :
        fKey(aKey) // Initialize the key
    {
        for (size_t i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL; // Initialize subtrees to NIL
        }
    }

    // destructor (free sub-trees, must not free empty trees) [14]
    // Destructor: Frees all non-NIL subtrees
    ~TernaryTree()
    {
        for (size_t i = 0; i < 3; i++)
        {
            if (!fSubTrees[i]->empty()) // Only delete non-empty subtrees
            {
                delete fSubTrees[i];
            }
        }
    }

    // return key value, may throw domain_error if empty [2]
    // Access the key value; throws error if the tree is empty
    const T& operator*() const
    {
        if (empty())
        {
            throw std::domain_error("NIL payload access");
        }
        return fKey;
    }

    // returns true if this ternary tree is empty [4]
    // Check if the tree is empty
    bool empty() const
    {
        return this == &NIL;
    }

    // returns true if this ternary tree is a leaf [10]
    // Check if the tree is a leaf (all subtrees are NIL)
    bool leaf() const
    {
        return fSubTrees[0] == &NIL &&
            fSubTrees[1] == &NIL &&
            fSubTrees[2] == &NIL;
    }

    // return height of ternary tree, may throw domain_error if empty [48]
    // Compute the height of the tree; throws error if the tree is empty
    size_t height() const
    {
        if (empty())
        {
            throw std::domain_error("Operation not supported");
        }

        // leaf
        if (leaf())
        {
            return 0;
        }

        // need variables
        size_t lLeft = 0;
        size_t lMiddle = 0;
        size_t lRight = 0;

        // left
        if (!fSubTrees[0]->empty())
        {
            lLeft = fSubTrees[0]->height();
        }

        // middle
        if (!fSubTrees[1]->empty())
        {
            lMiddle = fSubTrees[1]->height();
        }

        // right
        if (!fSubTrees[2]->empty())
        {
            lRight = fSubTrees[2]->height();
        }

        return std::max(lLeft, std::max(lMiddle, lRight)) + 1;
    }

    /////////////////////////////////////////////////////////////////////////
    // Problem 2: TernaryTree Copy Semantics
    // Copy and move semantics

    // copy constructor, must not copy empty ternary tree
        // Copy constructor: Avoids copying NIL trees
    TernaryTree(const TTree& aOtherTTree) :
        TernaryTree()
    {
        *this = aOtherTTree; // Delegate to copy assignment
    }

    // copy assignment operator, must not copy empty ternary tree
    // may throw a domain error on attempts to copy NIL
    // Copy assignment: Ensures proper handling of NIL and non-NIL trees
    TTree& operator=(const TTree& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw std::domain_error("NIL as source not permitted.");
        }

        if (this != &aOtherTTree)
        {
            // free this
            this->~TernaryTree();

            fKey = aOtherTTree.fKey;

            // just use clone
            fSubTrees[0] = aOtherTTree.getLeft().clone();
            fSubTrees[1] = aOtherTTree.getMiddle().clone();
            fSubTrees[2] = aOtherTTree.getRight().clone();
        }

        return *this;
    }

    // clone ternary tree, must not copy empty trees
```

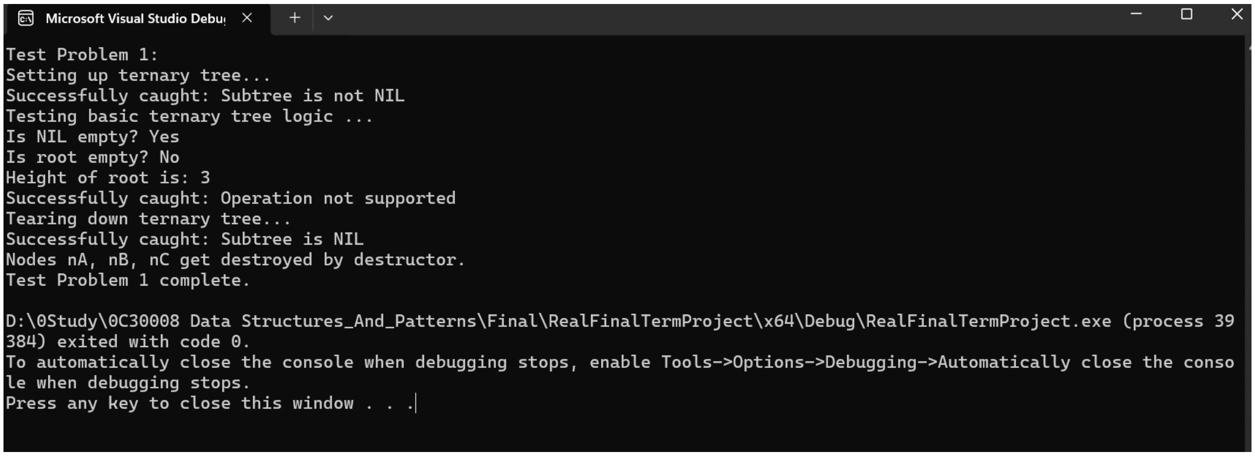
```cpp
    // Clone method: Creates a new tree copy or returns the current object if NIL
    TSubTree clone() const
    {
        if (empty())
        {
            // const cast required (remove const)
            return const_cast<TSubTree>(this);
        }
        else
        {
            return new TTree(*this);
        }
    }

    /////////////////////////////////////////////////////////////////////////
    // Problem 3: TernaryTree Move Semantics
    // Move constructor

    // TTree r-value constructor
    TernaryTree(T&& aKey) :
        fKey(std::move(aKey))
    {
        for (size_t i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL;
        }
    }

    // move constructor, must not copy empty ternary tree
    TernaryTree(TTree&& aOtherTTree) :
        /* just use default private default constructor */
        TernaryTree()
    {
        // use assignent operator
        *this = std::move(aOtherTTree);
    }


    // move assignment operator, must not copy empty ternary tree
    // Move assignment
    TTree& operator=(TTree&& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw std::domain_error("NIL as source not permitted.");
        }

        if (this != &aOtherTTree)
        {
            // free this
            this->~TernaryTree();

            // swap preparation
            fKey = T();

            for (size_t i = 0; i < 3; i++)
            {
                fSubTrees[i] = &NIL;
            }

            std::swap(fKey, aOtherTTree.fKey);
            std::swap(fSubTrees[0], aOtherTTree.fSubTrees[0]);
            std::swap(fSubTrees[1], aOtherTTree.fSubTrees[1]);
            std::swap(fSubTrees[2], aOtherTTree.fSubTrees[2]);
        }

        return *this;
    }

    /////////////////////////////////////////////////////////////////////////
    // Iteration support

        // Prefix iterator positioned at the start of the tree
    Iterator begin() const
    {
        return Iterator(this);
    }

    // Prefix iterator positioned at the end of the tree
    Iterator end() const
    {
        return begin().end();
    }
};

// Definition of the NIL sentinel
template<typename T>
TernaryTree<T> TernaryTree<T>::NIL;

```

```cpp
#pragma once

// COS30008, Final Exam
// Nguyen Minh Duy - 104974743

#include "TernaryTree.h"

#include <stack>

template<typename T>
class TernaryTreePrefixIterator
{
private:
    using TTree = TernaryTree<T>;            // Alias for the ternary tree type
    using TTreeNode = TTree*;                // Alias for a pointer to a tree node
    using TTreeStack = std::stack<const TTree*>; // Alias for a stack of tree pointers

    const TTree* fTTree;                     // Pointer to the ternary tree being iterated
    TTreeStack fStack;                       // Stack used for managing the traversal

public:

    using Iterator = TernaryTreePrefixIterator<T>; // Alias for the iterator type

    // Postfix increment operator
    Iterator operator++(int)
    {
        Iterator old = *this;  // Save the current state
        ++(*this);             // Perform prefix increment
        return old;            // Return the state before increment
    }

    // Inequality comparison operator
    bool operator!=(const Iterator& aOtherIter) const
    {
        return !(*this == aOtherIter); // Use equality to determine inequality
    }

    //////////////////////////////////////////////////////////////////////////
    // // Problem 4: TernaryTree Prefix Iterator

```

```cpp
private:

    // Pushes the subtrees (left, middle, right) of a given node onto the stack
    void push_subtrees(const TTree* aNode)
    {
        if (!aNode->getRight().empty()) // Check if the right subtree exists
        {
            fStack.push(&aNode->getRight()); // Push the right subtree
        }

        if (!aNode->getMiddle().empty()) // Check if the middle subtree exists
        {
            fStack.push(&aNode->getMiddle()); // Push the middle subtree
        }

        if (!aNode->getLeft().empty()) // Check if the left subtree exists
        {
            fStack.push(&aNode->getLeft()); // Push the left subtree
        }
    }

public:

    // iterator constructor
    TernaryTreePrefixIterator(const TTree* aTTree) :
        fTTree(aTTree) // Initialize the ternary tree pointer
    {
        if (!fTTree->empty()) // If the tree is not empty
        {
            fStack.push(fTTree); // Push the root of the tree onto the stack
        }
    }

    // iterator dereference
    const T& operator*() const
    {
        return **fStack.top(); // Return the value of the node on top of the stack
    }

    // prefix increment
    Iterator& operator++()
    {
        const TTree* lTop = fStack.top(); // Get the current node (top of the stack)
        fStack.pop();                    // Remove the current node from the stack
        push_subtrees(lTop);             // Push its subtrees onto the stack
        return *this;                    // Return the updated iterator
    }

    // iterator equivalence
    bool operator==(const Iterator& aOtherIter) const
    {
        return
            fTTree == aOtherIter.fTTree &&           // Check if they are iterating the same tree
            fStack.size() == aOtherIter.fStack.size(); // Check if their stacks have the same size
    }

    // auxiliaries
    Iterator begin() const
    {
        return TernaryTreePrefixIterator(fTTree); // Create a new iterator starting at the root
    }

    // Returns an iterator representing the end of the traversal
    Iterator end() const
    {
        Iterator Result = *this;       // Copy the current iterator
        Result.fStack = TTreeStack(); // Clear the stack to represent the end
        return Result;                // Return the end iterator
    }
};

```

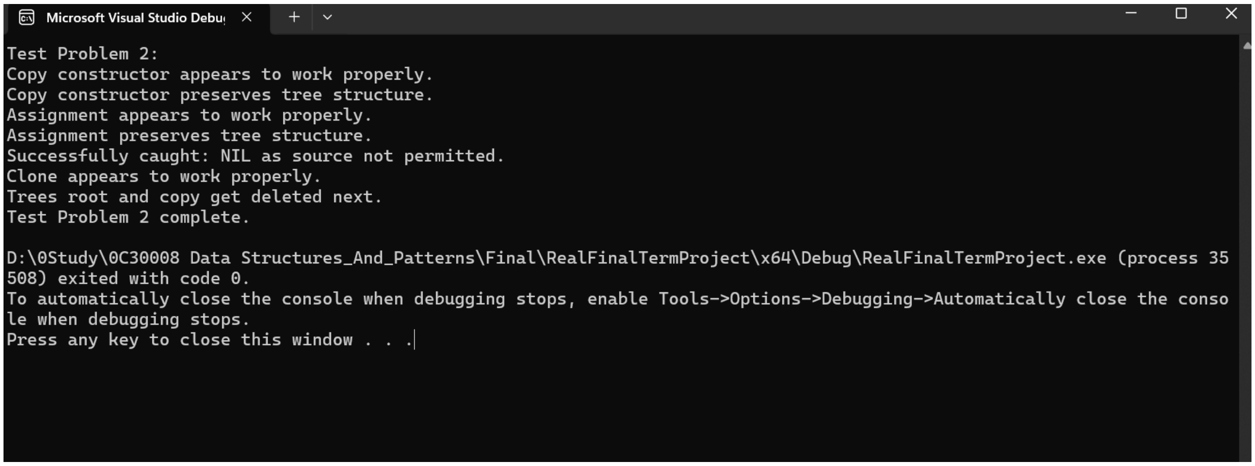
Output P1 – P4

Problem 1



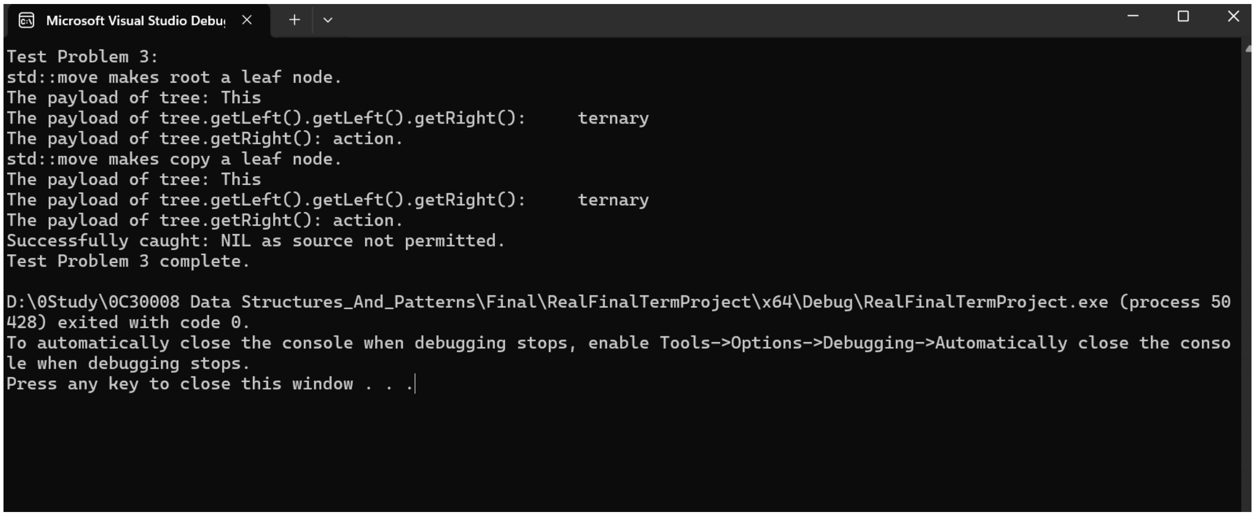
```
Test Problem 1:
Setting up ternary tree...
Successfully caught: Subtree is not NIL
Testing basic ternary tree logic ...
Is NIL empty? Yes
Is root empty? No
Height of root is: 3
Successfully caught: Operation not supported
Tearing down ternary tree...
Successfully caught: Subtree is NIL
Nodes nA, nB, nC get destroyed by destructor.
Test Problem 1 complete.

D:\0Study\0C30008 Data Structures_And_Patterns\Final\RealFinalTermProject\x64\Debug\RealFinalTermProject.exe (process 39
384) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Problem 2



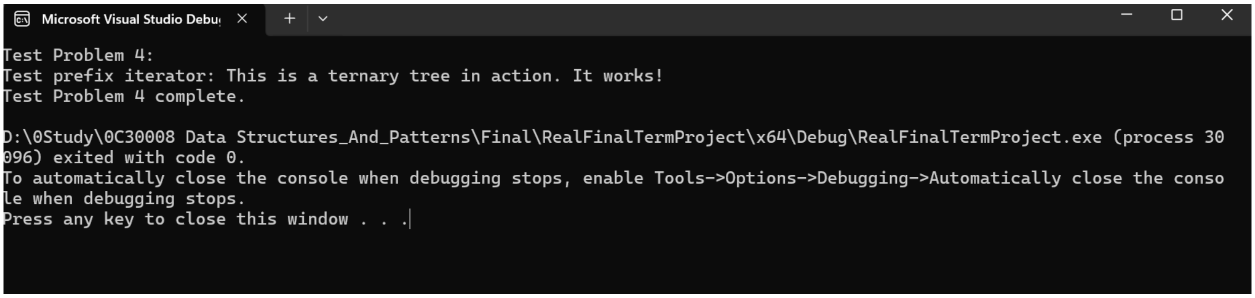
```
Test Problem 2:
Copy constructor appears to work properly.
Copy constructor preserves tree structure.
Assignment appears to work properly.
Assignment preserves tree structure.
Successfully caught: NIL as source not permitted.
Clone appears to work properly.
Trees root and copy get deleted next.
Test Problem 2 complete.

D:\0Study\0C30008 Data Structures_And_Patterns\Final\RealFinalTermProject\x64\Debug\RealFinalTermProject.exe (process 35
508) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Problem 3



```
Test Problem 3:
std::move makes root a leaf node.
The payload of tree: This
The payload of tree.getLeft().getLeft().getRight():     ternary
The payload of tree.getRight(): action.
std::move makes copy a leaf node.
The payload of tree: This
The payload of tree.getLeft().getLeft().getRight():     ternary
The payload of tree.getRight(): action.
Successfully caught: NIL as source not permitted.
Test Problem 3 complete.

D:\0Study\0C30008 Data Structures_And_Patterns\Final\RealFinalTermProject\x64\Debug\RealFinalTermProject.exe (process 50
428) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Problem 4



```
Test Problem 4:
Test prefix iterator: This is a ternary tree in action. It works!
Test Problem 4 complete.

D:\0Study\0C30008 Data Structures_And_Patterns\Final\RealFinalTermProject\x64\Debug\RealFinalTermProject.exe (process 30
096) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Problem 5 (50 marks)

Answer the following questions in one or two sentences:

a. How can we construct a tree where all nodes have the same degree? [4]

5a)

To construct a tree where all nodes have the same degree, we introduce a special abstraction called an empty tree. This serves as a sentinel, often referred to as NIL. The NIL object acts as a placeholder for missing children in the tree, ensuring every node has a consistent degree. By utilizing this abstraction, the tree structure remains uniform regardless of the data or branching patterns.

b. What is the difference between l-value and r-value references? [6]

5b)

L-value references refer to objects that persist beyond a single expression. They avoid unnecessary copies and allow direct access to existing memory locations, making manipulation of existing objects efficient. They are primarily used when modifications or bindings to named objects are needed.
R-value references are used for temporary objects that do not persist beyond a single expression. They enable move semantics, allowing efficient transfer of resources (like memory or file handles) from temporary objects, thereby optimizing resource management and avoiding unnecessary duplications.

c. What is a key concept of an abstract data types? [4]

5c)

The key concept of an abstract data type is representation transparency. This means that multiple different implementations can fulfill the same interface, ensuring abstraction and flexibility. Users interact with the ADT solely through its interface without being concerned about its internal workings or data structure, enabling modular and reusable code.

d. How do we define mutual dependent classes in C++? [4]

5d)

To handle mutual dependencies between classes in C++, we use a combination of:
1. Forward declaration to break dependency cycles. This allows one class to reference another without including its full definition upfront.
2. Including headers of all mutually dependent classes in the implementation files, ensuring that the compiler has full information when needed.

e. What must a value-based data type define in C++? [2]

5e)

A value-based data type must define proper copy semantics. This involves implementing a copy constructor and copy assignment operator to ensure that objects can be duplicated safely and efficiently without issues like shallow copies, which may lead to unexpected behaviors or resource conflicts.

f. What is an object adapter? [6]

5f)

An object adapter is a structural design pattern. It defines a wrapper class that encapsulates a delegate object, known as the adaptee, to adapt its interface to match the client's expectations. This pattern ensures seamless interaction by mapping client operations to the corresponding operations provided by the adaptee. It’s commonly used to bridge incompatible interfaces in object-oriented systems.

g. What is the difference between copy constructor and assignment operator and how do we guarantee safe operation? [8]

5g)

The copy constructor initializes a new object using an existing object as a source.
The assignment operator overwrites an already existing object with the contents of another.
To ensure safe operations:
1. Properly manage memory by freeing resources in the target object before assigning new values.
2. Implement safeguards against self-assignment to prevent accidental data corruption or resource deallocation.

h. What is the best-case, average-case, and worse-case for a lookup in a binary tree? [6]

5h)

Best-case: O(1), when the desired value is located at the root.
Average-case: O(log n), typical for balanced binary search trees (BSTs), A balanced tree where each step reduces the search space.
Worst-case: O(n), occurs when the tree degenerates into a linked list.

i. What are reference data members and how do we initialize them? [2]

5i)

Reference data members act as aliases to existing objects. They cannot be null or re-bound once initialized. Initialization is performed using the member initializer list in the constructor of the containing class. This ensures that references are properly assigned during object creation.

j. You are given n-1 numbers out of n numbers. How do we find the missing number n_k, $1 \le k \le n$, in linear time? [8]

5j)

To find the missing number:
1. Calculate the sum of the n−1 numbers, v, in O(n) time using a simple for-loop.
2. Compute the expected sum of n numbers as sum(n) = n(n+1)/2, which is O(1).
3. The missing number nk is obtained by subtracting v from sum(n):
nk = sum(n) − v
This algorithm ensures linear time complexity, combining efficiency with clarity.