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 Thi Thanh Minh **Your student id:** 104169617

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
			X	X							

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.

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) We can use an array with degree (d) as length to store subtrees for each node, and a static sentinel node representing NIL. Starting with the root node, for each node we add exactly d children (with default value being NIL), and recursively repeat the process for each child node.

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

5b) An l-value reference (T&) is a variable name that refers to an object in a defined location in memory and can be used to modify the object. It's used for copying. An r-value reference (T&&) is just a temporary object that allows move semantics, e.g. "abc", int(9) or std::move(lValue).

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

5c) Abstract data types provide representation independence by hiding away their actual implementations, which means that they can be used as an interface for different implementations to work without knowing another's actual representation.

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

5d) We need to resolve the mutual dependence by using forward declaration for one of the classes, then define its full implementation after the other class, and put all specifications in the same file and include both headers so that the compiler can resolve dependencies.

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

5e) Copy Constructor: To allow copying of objects.
Copy Assignment Operator: To assign one object to another.
Equality Operator (operator==): To compare objects by value.
Move Constructor and Move Assignment Operator (optional but recommended for efficiency).

f. What is an object adapter? [6]

5f) An object adapter is a design pattern that converts the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

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

5g) The assignment operator copies another object to an existing one (overrides) that may not be empty or not, and these two objects may actually be the same. Whereas the copy constructor initializes a new object. We can guarantee safe operation by releasing old resources and checking the object addresses - if they are equal, it means that we are committing an accidental suicide!

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 key is at the root.
- Average case: O(log n) for a balanced binary search tree.
- Worst case: O(n) for a degenerate (unbalanced) tree.

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

5i) A reference data member is a member of a class that refers to another object. It must be initialized in the initializer list of the constructor (member initializers).

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

5j)

1. A for loop to calculate sum up to n-1 => O(n)
2. sum(n) = n * n(+1) / 2 => O(1)
3. n_k = sum(n) - sum(n-1) => O(1)

```cpp

// COS30008, Final Exam

#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;           // sentinel

    // 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
    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, may through a domain error
    const TTree& removeLeft() { return removeSubTree( 0 ); }
    const TTree& removeMiddle() { return removeSubTree( 1 ); }
    const TTree& removeRight() { return removeSubTree( 2 ); }

/////////////////////////////////////////////////////////////////////////
// Problem 1: TernaryTree Basic Infrastructure

private:

    // remove a subtree, may throw a domain error [22]
    const TTree& removeSubTree(size_t aSubtreeIndex)
    {
        if (fSubTrees[aSubtreeIndex] == &NIL)
        {
            throw std::domain_error("Empty subtree removed!");
        }

        TTree* lResult = fSubTrees[aSubtreeIndex];

        fSubTrees[aSubtreeIndex] = &NIL;

        return *lResult;
    }

    // add a subtree; must avoid memory leaks; may throw domain error [18]
    void addSubTree(size_t aSubtreeIndex, const TTree& aTTree)
    {
        if (aSubtreeIndex > 2)
        {
            throw std::out_of_range("Invalid subtree index!");
        }

        if (fSubTrees[aSubtreeIndex] != &NIL)
        {
            throw std::domain_error("Subtree is not empty!");
        }

        fSubTrees[aSubtreeIndex] = new TTree(aTTree);
    }

public:

    // TernaryTree l-value constructor [10]
    TernaryTree(const T& aKey) :
        TernaryTree()
    {
        fKey = aKey;
```

```cpp
    }

    // destructor (free sub-trees, must not free empty trees) [14]
    ~TernaryTree()
    {
        for (size_t i = 0; i < 3; i++)
        {
            if (fSubTrees[i] != &NIL)
            {
                delete fSubTrees[i];
            }
        }
    }

    // return key value, may throw domain_error if empty [2]
    const T& operator*() const
    {
        if (empty())
        {
            throw std::domain_error("Empty tree!");
        }

        return fKey;
    }

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

    // returns true if this ternary tree is a leaf [10]
    bool leaf() const
    {
        for (size_t i = 0; i < 3; i++)
        {
            if (fSubTrees[i] != &NIL)
            {
                return false;
            }
        }

        return true;
    }

    // return height of ternary tree, may throw domain_error if empty [48]
    size_t height() const
    {
        if (empty())
        {
            throw std::domain_error("Empty tree has no height!");
        }

        if (leaf())
        {
            return 0;
        }

        size_t lMaxHeight = 0;
        for (size_t i = 0; i < 3; i++)
        {
            if (fSubTrees[i] != &NIL)
            {
                lMaxHeight = std::max(lMaxHeight, fSubTrees[i]->height());
            }
        }

        return 1 + lMaxHeight;
    }

//////////////////////////////////////////////////////////////////////////
// Problem 2: TernaryTree Copy Semantics

    // copy constructor, must not copy empty ternary tree
    TernaryTree(const TTree& aOtherTTree) :
        TernaryTree()
    {
        *this = aOtherTTree;
    }

    // copy assignment operator, must not copy empty ternary tree
    // may throw a domain error on attempts to copy NIL
    TTree& operator=(const TTree& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw std::domain_error("Empty tree cannot be copied!");
        }

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

            fKey = *aOtherTTree;
            for (size_t i = 0; i < 3; i++)
            {
                fSubTrees[i] = aOtherTTree.fSubTrees[i]->clone();
            }
        }

        return *this;
    }


    // clone ternary tree, must not copy empty trees
    TSubTree clone() const
    {
        if (empty())
        {
            return const_cast<TSubTree>(this);
        }

        return new TTree(*this);
    }

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

    // TTree r-value constructor
    TernaryTree(T&& aKey) :
        TernaryTree()
    {
        fKey = std::move(aKey);
    }

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

    // move assignment operator, must not copy empty ternary tree
    TTree& operator=(TTree&& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw std::domain_error("Empty tree cannot be moved!");
        }

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

            fKey = std::move(*aOtherTTree);

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

        return *this;
    }

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

    // return ternary tree prefix iterator positioned at start
    Iterator begin() const
    {
        return Iterator(this);
    }

    // return ternary prefix iterator positioned at end
    Iterator end() const
    {
        return begin().end();
    }
};

template<typename T>
TernaryTree<T> TernaryTree<T>::NIL;
```

```cpp

// COS30008, Final Exam

#pragma once

#include "TernaryTree.h"

#include <stack>

template<typename T>
class TernaryTreePrefixIterator
{
private:
    using TTree = TernaryTree<T>;
    using TTreeNode = TTree*;
    using TTreeStack = std::stack<const TTree*>;

    const TTree* fTTree;                // ternary tree
    TTreeStack fStack;                  // traversal stack

public:

    using Iterator = TernaryTreePrefixIterator<T>;

    Iterator operator++(int)
    {
        Iterator old = *this;

        ++(*this);

        return old;
    }

    bool operator!=( const Iterator& aOtherIter ) const
    {
        return !(*this == aOtherIter);
    }

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

private:

    // push subtree of aNode [30]
    void push_subtrees(const TTree* aNode)
    {
        if (!aNode->getRight().empty())
        {
            fStack.push(&aNode->getRight());
        }
        if (!aNode->getMiddle().empty())
        {
            fStack.push(&aNode->getMiddle());
        }
        if (!aNode->getLeft().empty())
        {
            fStack.push(&aNode->getLeft());
        }
    }

public:

    // iterator constructor [12]
    TernaryTreePrefixIterator( const TTree* aTTree ) :
        fTTree(aTTree)
    {
        if (fTTree != &TTree::NIL)
        {
            fStack.push(fTTree);
        }
    }

    // iterator dereference [8]
    const T& operator*() const
    {
        return **fStack.top();
    }

    // prefix increment [12]
    Iterator& operator++()
    {
        if (!fStack.empty())
        {
            const TTree* current = fStack.top();
            fStack.pop();
            push_subtrees(current);
        }

        return *this;
    }

    // iterator equivalence [12]
    bool operator==(const Iterator& aOtherIter) const
    {
        return (fTTree == aOtherIter.fTTree)
            && (fStack == aOtherIter.fStack);
    }

    // auxiliaries [4,10]
    Iterator begin() const
    {
        return Iterator(fTTree);
    }

    Iterator end() const
    {
        Iterator lIter(fTTree);
        lIter.fStack = TTreeStack();
        return lIter;
    }
};

```