

**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:** \_\_\_\_\_ **Your student id:** \_\_\_\_\_

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.

## 1. TernaryTree.h

```
// COS30008, Final Exam, 2022
```

```
#pragma once
```

```
#include <stdexcept>
```

```
#include <algorithm>
```

```
using namespace std;
```

```
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 (aSubtreeIndex > 2)
        {
            throw out_of_range("Illegal subtree index");
        }

        while (fSubTrees[aSubtreeIndex] -> empty())
        {
            throw domain_error("Subtree is NIL");
        }

        const TTree& index = const_cast<TTree&>(*fSubTrees[aSubtreeIndex]);
        fSubTrees[aSubtreeIndex] = &NIL;
        return index;
    }

    // add a subtree; must avoid memory leaks; may throw domain error [18]
    void addSubTree(size_t aSubtreeIndex, const TTree& aTTree)
    {
        if (empty())
        {
            throw domain_error("Operation not supported");
        }

        if (aSubtreeIndex > 2)
        {
            throw out_of_range("Illegal subtree index");
        }

        for (; fSubTrees[aSubtreeIndex] -> empty(); )
        {
            fSubTrees[aSubtreeIndex] = const_cast<TTree*>(&aTTree);
            return;
        }

        throw domain_error("Subtree is not NIL");
    }

public:

    // TernaryTree l-value constructor [10]
    TernaryTree(const T& aKey) : fKey(aKey)
    {
        int i = 0;
        while (i < 3)
        {
            fSubTrees[i] = &NIL;
            i++;
        }
    }

```

```

}

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

// return key value, may throw domain_error if empty [2]
const T& operator*() const
{
    if (empty())
    {
        throw domain_error("Tree is empty");
    }
    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
{
    int i = 0;
    while (i < 3)
    {
        if (!fSubTrees[i]->empty())
        {
            return false;
        }
        i++;
    }
    return true;
}

// return height of ternary tree, may throw domain_error if empty [48]
size_t height() const
{
    if (empty())
    {
        throw domain_error("Operation not supported");
    }

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

```

```

    }

    size_t maxHeight = 0;

    for (int i = 0; i < 3; i++)
    {
        if (!fSubTrees[i]->empty())
        {
            size_t subtreeHeight = fSubTrees[i]->height();
            maxHeight = max(maxHeight, subtreeHeight);
        }
    }

    return maxHeight + 1;
}

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

// copy constructor, must not copy empty ternary tree
TernaryTree(const TTree& aOtherTTree)
{
    int i = 0;
    while (i < 3)
    {
        fSubTrees[i] = &NIL;
        i++;
    }

    *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 (this == &aOtherTTree) {
        return *this;
    }

    if (aOtherTTree.empty()) {
        throw domain_error("NIL as source not permitted.");
    }

    this->~TernaryTree();
    fKey = aOtherTTree.fKey;

    size_t i = 0;
    while (i < 3) {
        if (!aOtherTTree.fSubTrees[i]->empty()) {
            fSubTrees[i] = aOtherTTree.fSubTrees[i]->clone();
        }
        else {
            fSubTrees[i] = &NIL;
        }
        i++;
    }
}

```

```

        return *this;
    }

    // clone ternary tree, must not copy empty trees
    TSubTree clone() const
    {
        if (empty())
        {
            throw domain_error("NIL as source not permitted.");
        }
        return new TTree(*this);
    }

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

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

    // move constructor, must not copy empty ternary tree
    TernaryTree(TTree&& aOtherTTree)
    {
        int i = 0;
        while (i < 3)
        {
            fSubTrees[i] = &NIL;
            i++;
        }

        *this = std::move(aOtherTTree);
    }

    // move assignment operator, must not copy empty ternary tree
    TTree& operator=(TTree&& aOtherTTree)
    {
        if (this == &aOtherTTree) {
            return *this;
        }

        if (aOtherTTree.empty()) {
            throw std::domain_error("NIL as source not permitted.");
        }

        this->~TernaryTree();
        fKey = std::move(aOtherTTree.fKey);

        int i = 0;
        while (i < 3) {

```

```

        fSubTrees[i] = aOtherTTree.fSubTrees[i]->empty() ? &NIL :
const_cast<TSubTree>(&aOtherTTree.removeSubTree(i));
        i++;
    }

    return *this;
}

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

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

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

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

```

## 2. TernaryTreePrefixIterator.h

```

// COS30008, Final Exam, 2022

#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)
    {
        const TTree* subtrees[] = { &(*aNode).getRight(), &(*aNode).getMiddle(),
&(*aNode).getLeft() };

        for (const TTree* subtree : subtrees)
        {
            if (!subtree->empty())
            {
                fStack.push(const_cast<TTreeNode>(subtree));
            }
        }

        // for auxiliaries [4,10]
        void resetStackAndPushRoot()
        {
            fStack = TTreeStack();
            fStack.push(const_cast<TTreeNode>(fTTree));
        }

        void resetStack()
        {
            fStack = TTreeStack();
        }

public:
    // iterator constructor [12]
    TernaryTreePrefixIterator(const TTree* aTTree) : fTTree(aTTree), fStack()
    {
        while (!(*fTTree).empty())
        {
            fStack.push(const_cast<TTreeNode>(fTTree));
            break; // Exit the loop after pushing once
        }
    }

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

```



```

    }

    // prefix increment [12]
    Iterator& operator++()
    {
        if (!fStack.empty())
        {
            TTreeNode lPopped = const_cast<TTreeNode>(fStack.top());
            fStack.pop();
            push_subtrees(lPopped);
        }
        return *this;
    }

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

    // auxiliaries [4,10]
    Iterator begin() const
    {
        Iterator temp = *this;
        temp.resetStackAndPushRoot();
        return temp;
    }

    Iterator end() const
    {
        Iterator temp = *this;
        temp.resetStack();
        return temp;
    }
};

```

**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)**

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

**5b)**

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

**5c)**

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

**5d)**

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

**5e)**

f. What is an object adapter? [6]

5f)

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

5g)

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

5h)

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

5i)

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

5j)