Swinburne University of Technology

Faculty of Science, Engineering and Technology

FINAL EXAM COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures & Patterns
Due date: June 3, 2021, 13:00
Lecturer: Dr. Markus Lumpe

Your name: Zoie Tad-y **Your student id:** 102884743

Check Tutorial	Wed 08:30	Wed 10:30	Wed 16:30	Thurs 08:30	Thurs 10:30	Thurs 14:30	Thurs 16:30	Fri 08:30	Fri 10:30	Fri 14:30
		X								

Marker's comments:

Problem	Marks	Time Estimate in minutes	Obtained
1	50	20	
2	54	15	
3	42	10	
4	60	15	
5	8+128=136	60	
Total	342	120	

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

Problem 1 (50 marks)

Answer the following questions in one or two sentences:

a. How can we construct a tree where all subtrees have the same degree? (4 marks)

1a) Predefine pointers to N number of nodes for each node created for a tree and make sure that the tree is strictly balanced. Degree being the number of child nodes for a specific node, setting a predefined definition that each node created will always point to either N number of nodes or 0 will ensure that each subtree will have have the same degree.

b. What are reference data members and how do we initialize them? (2 marks)

1b) Reference data members are "variables" that serve as an alias to an existing value stored in memory. We can intialise a reference member by using the type followed by "&" and the name of the variable reference, and equating them to a variable.

c. What is the difference between l-value and r-value references? (6 marks)

1c) L value references point towards or reference to a specific section in memory that may contain value, r-value references are simply references to value that is short lived and is not stored in memory. In most reference and even earlier early c++ discussion, r value references are simply referred to as the opposite of l value references. Typically l value references are in the left hand side of the assignment operator, and r value references are found in the right hand side of assignment operator. However l values may also appear on the right hand side of the when it is being assigned to another l value reference. R value references can never be modified or assigned with another value.

d. What is an object adapter? (6 marks)

1d) An Object adapter is a wrapper around existing objects that allows them to collaborate. Certain definitions of certain objects prevent them from working with the functionality of other objects, in the event that we need that objects functionality to work with another object we create adapters to transform or format one or two of the objects to make them seemingly compatible.

e. What is a key concept of an abstract data types? (4 marks)

1e) Abstract data types are data types that defined by their functionality, not strictly how they are constructed declaratively. Being abstract means that it exists as a thought or idea, and being an ADT is just that, for example two stacks that a programtically defined differently are still considered stacks as long as they adhere to the abstract definition of what a stack is and can do.

f. How do we define mutual dependent classes in C++? (4 marks)

1f)

We can use forward declaration, declare at least one of the classes in the definition of the other through references or pointers.

g. What must a value-based data type define in C++? (2 marks)

1g)

Value based data types must define copy constructors.

h. What is the difference between copy constructor and assignment operator and how do we guarantee safe operation? (8 marks)

1h)

Copy constructors creates new instances of the a given object with the same information that object has through assigning a spearate memory block for the new object, assignment operators assigns the entirety of one object to another another object without allocating new memory space. To ensure safe operation deleting the old instances, or freeing the memory space allocated from the old instance should be done to prevent memory leaks.

i. What is the best-case, average-case, and worse-case for a lookup in a binary tree? (6 marks)

1i)

Worst case is O(n) where each node is visited, best case is O(1) where the node being looked up is the root, and average case if O(log n) where approx half the nodes are visited in completing the look up.

j. You are given a set of 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 marks)

1j)

We can convert both sets, set one being the given n-1 numbers, and the other being the general set of n numbers, to its canonical form by sorting which is O(n log n) and then traverse both at the same time, when the values do not match the that means the value at that index on the general set of n numbers is the missing number. Linear traversal is O(n) so the entire process is O(nlogn) + O(n), drop O (nlogn) since O(n) is larger, hence O(n)

```
// COS30008, Final Exam, 2021

#pragma once

#include <stdexcept>

template<typename T>
class TTreePostfixIterator;

template<typename T>
class TTree
{
private:

    T fKey;
    TTree<T>* fLeft;
    TTree<T>* fMiddle;
    TTree<T>* fRight;

    TTree() : fKey(T())     // use default constructor to initialize fKey
    {
        fLeft = &NIL;       // loop-back: The sub-trees of a TTree object with
        fMiddle = &NIL;     //            no children point to NIL.
        fRight = &NIL;
    }

    void addSubTree( TTree<T>** aBranch, const TTree<T>& aTTree )
    {
        if ( !(*aBranch)->empty() )
        {
            delete *aBranch;
        }

        *aBranch = const_cast<TTree<T>*>(&aTTree);
    }

public:

    using Iterator = TTreePostfixIterator<T>;

    static TTree<T> NIL;    // sentinel

    // getters for subtrees
    const TTree<T>& getLeft() const { return *fLeft; }
    const TTree<T>& getMiddle() const { return *fMiddle; }
    const TTree<T>& getRight() const { return *fRight; }

    // add a subtree
    void addLeft( const TTree<T>& aTTree ) { addSubTree( &fLeft, aTTree ); }
    void addMiddle( const TTree<T>& aTTree ) { addSubTree( &fMiddle, aTTree ); }
    void addRight( const TTree<T>& aTTree ) { addSubTree( &fRight, aTTree ); }
```

```

    // remove a subtree, may through a domain error
    const TTree<T>& removeLeft() { return removeSubTree( &fLeft ); }
    const TTree<T>& removeMiddle() { return removeSubTree( &fMiddle ); }
    const TTree<T>& removeRight() { return removeSubTree( &fRight ); }

// Problem 1: TTree Basic Infrastructure

private:

    // remove a subtree, may through a domain error
    const TTree<T>& removeSubTree(TTree<T>** aBranch)
    {
        if (*aBranch == &NIL)
        {
            throw std::domain_error("Branch does not exist");
        }

        *aBranch = &NIL;
    }

public:

    // TTree l-value constructor
    TTree(const T& aKey) :fKey(aKey), fLeft(&NIL), fMiddle(&NIL), fRight(&NIL)
    {}

    // destructor (free sub-trees, must not free empty trees)
    ~TTree()
    {
        if (empty())
        {
            throw domain_error("Empty TTree");
        }
        delete fLeft;
        delete fMiddle;
        delete fRight;

    }


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

        return fKey;
    }

```

```
    // returns true if this TTree is empty
    bool empty() const
    {
        return this == &NIL;
    }

    // returns true if this TTree is a leaf
    bool leaf() const
    {
        return fLeft->empty() && fMiddle->empty() && fRight->empty();
    }

// Problem 2: TTree Copy Semantics

    // copy constructor, must not copy empty TTree
    TTree(const TTree<T>& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw domain_error("Empty Tree.");
        }

        if (!aOtherTTree.empty())
        {
            fKey = aOtherTTree.fKey;

            addRight(aOtherTTree.getRight());
            addMiddle(aOtherTTree.getMiddle());
            addLeft(aOtherTTree.getLeft());
        }
    }

    // copy assignment operator, must not copy empty TTree
    TTree<T>& operator=(const TTree<T>& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw domain_error("Empty Tree.");
        }

        if (this != &aOtherTTree)
        {
            delete fLeft;
            delete fMiddle;
            delete fRight;

            fKey = aOtherTTree.fKey;

            addRight(aOtherTTree.getRight());
            addMiddle(aOtherTTree.getMiddle());
            addLeft(aOtherTTree.getLeft());

```

```
        }
        return *this;
    }

    // clone TTree, must not copy empty trees
    TTree<T>* clone() const
    {
        if (!empty())
        {
            return new TTree(*this);
        }
    }

// Problem 3: TTree Move Semantics

    // TTree r-value constructor
    TTree(T&& aKey) :fKey(aKey), fLeft(&NIL), fMiddle(&NIL), fRight(&NIL)
    {}

    // move constructor, must not copy empty TTree
    TTree(TTree<T>&& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw domain_error("Empty Tree.");
        }

        if (this != &aOtherTTree)
        {
            fKey = aOtherTTree.fKey;
            fRight = std::move((aOtherTTree.getRight()));
            fMiddle = std::move((aOtherTTree.getMiddle()));
            fLeft = std::move((aOtherTTree.getLeft()));

        }
        return *this;
    }

    // move assignment operator, must not copy empty TTree
    TTree<T>& operator=(TTree<T>&& aOtherTTree)
    {
        if (aOtherTTree.empty())
        {
            throw domain_error("Empty Tree.");
        }

        if (this != &aOtherTTree)
        {
            delete fLeft;
            delete fMiddle;
            delete fRight;

```

```
            fKey = aOtherTTree.fKey;

            fRight = std::move((aOtherTTree.getRight()));
            fMiddle = std::move((aOtherTTree.getMiddle()));
            fLeft = std::move((aOtherTTree.getLeft()));

        }
        return *this;
    }

// Problem 4: TTree Postfix Iterator

    // return TTree iterator positioned at start
    Iterator begin() const
    {
        return Iterator(&this);
    }

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

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

```cpp

// COS30008, Final Exam, 2021

#pragma once

#include "TTree.h"

#include <stack>

template<typename S>
struct TTreeFrontier
{
    size_t stage;                           // frontier stages: 0, 1, 2
    const TTree<S>* node;                   // frontier TTree node

    TTreeFrontier( const TTree<S>* aNode ) :
        node(aNode),                        // TTree node
        stage(0)                            // 0 - start right
    {}
};

template<typename T>
class TTreePostfixIterator
{
private:
    const TTree<T>* fTTree;                 // 3-way tree
    std::stack<TTreeFrontier<T>> fStack;    // DFS traversal stack

    using Frontier = TTreeFrontier<T>;

    // push subtree starting with aNode
    void push_nodes(const TTree<T>* aNode)
    {
        fStack.push(aNode);
    }

 public:

    using Iterator = TTreePostfixIterator<T>;

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

        ++(*this);

        return old;
    }

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

```
    }

    // iterator constructor
    TTreePostfixIterator(const TTree<T>* aTTree)
    {
        fTTree = aTTree;

        constTTree<T>* lNode = fTTree;

        push_nodes(lNode);

        while (!lNode.empty())
        {

            if (!lNode.GetRight().empty())
                push_nodes(lNode.GetRight());
            if (!lNode.GetMiddle().empty())
                push_nodes(lNode.GetMiddle());
            if (!lNode.GetLeft().empty())
                push_nodes(lNode.GetLeft());

            lNode = lNode.GetLeft();
        }

    }

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

    // prefix increment
    Iterator& operator++()
    {
        TTree<T>* lNode = fStack.top().GetMiddle();

        fStack.pop();

        if (!lNode->empty())
        {
            fStack.push(lNode);

            lNode = lNode.GetMiddle();

            while (!lNode->empty())
            {
                fStack.push(lNode);
                lNode = lNode.GetMiddle();
            }
        }
        else
```

```
        {
            lNode = fStack.top().GetLeft();
            fStack.push(lNode);

            lNode = lNode.GetLeft();

            while (!lNode->empty())
            {
                fStack.push(lNode);
                lNode = lNode.GetLeft();
            }

        }
        return *this;
    }

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

    // auxiliaries
    Iterator begin() const
    {
        return Iterator(fTTree);
    }
    Iterator end() const
    {
        Iterator iter = *this;

        iter.fStack = std::stack<const TTree<T>*>();

        return iter;
    }
};
```