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: Gia Huy Nguyen **Your student id:** 103441107

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

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)

We can build a tree with all subtrees have the same degree when the subtrees have the same arity. We can use the empty set to represent empty trees.

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

1b)

Data members of an object that relate to another object are known as reference data members. We can initialize them by using pointers which hold the reference object's address or with C++ reference types.

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

1c)

Temporary object's lifetime can be extended using r-value references. I-value references pertain to already existent objects, therefore I-value references are non-temporary, except when declared const. Copy semantics use I-value references while r-value references are used in move semantics.

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

1d)

Object adapter is a wrapper around an existing object which allows it to be used in another interface without adding to it.

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

1e)

Key concept of an abstract data types:
- Hidden implementation details so anyone uses it just needs to know what operations could be performed on it.
- Abstract data types handle the implementation of data type's fuctions.
- An abstraction over a concrete data structure.

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

1f)

In C++, everything must be pre-defined before using, so one of the classes must have a forward declaration to define mutual dependent classes. When using a forward declaration in C++, we will start with existed things but its implementation will be given later.

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

1g)

C++ class-like objects are value types in default, thus there is no need to specify anything else in order to use them. However, it may be required to override the copy constructor and assignment to ensure that the values are replicated. If the copy constructor is manually generated, the matching move constructor is also required.

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

1h)

Constructors initialize newly-created objects while assignment operator override existing objects with a new value.The copy assignment will delete the objects that existed before the assignment, as well as fulfill the copy constructor's function. During self-assignment, assignment operators should not be enable an object to destroy itself. We can guarantee safe operation, C++ copy-and-swap idiom can be utilized to ensure that if a copy operation fails, the object's original state is retained.

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

1i)

For a binary tree, time complexity:
- Best case: O(1)
- Average case: O(n)
- Worst case: O(n)

Binary search tree, time complexity:
- Best case: O(1)
- Average case: O(logn)
- Worst case: O(n)

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 have the formular to calculate the sum of integers from 1 to n (n is integer): Sum = (n * (n+1)) / 2.

We compare the difference of two sums (1 to n-1 and 1 to n), the difference result is the value of missing number. This can be achieved in linear time by calculating the sum of intergers from 1 to n then substracting the sum of the set with a missing number.

Gia Huy Nguyen – 103441107

<TTree.h>

```
// 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 );

public:

    // TTree l-value constructor
    TTree( const T& aKey );

    // destructor (free sub-trees, must not free empty trees)
    ~TTree();

    // return key value, may throw domain_error if empty
    const T& operator*() const;

    // returns true if this TTree is empty
    bool empty() const;

    // returns true if this TTree is a leaf
    bool leaf() const;

// Problem 2: TTree Copy Semantics

    // copy constructor, must not copy empty TTree
```

```
    TTree( const TTree<T>& aOtherTTree );

    // copy assignment operator, must not copy empty TTree
    TTree<T>& operator=( const TTree<T>& aOtherTTree );

    // clone TTree, must not copy empty trees
    TTree<T>* clone() const;

// Problem 3: TTree Move Semantics

    // TTree r-value constructor
    TTree( T&& aKey );

    // move constructor, must not copy empty TTree
    TTree( TTree<T>&& aOtherTTree );

    // move assignment operator, must not copy empty TTree
    TTree<T>& operator=( TTree<T>&& aOtherTTree );

// Problem 4: TTree Postfix Iterator

    // return TTree iterator positioned at start
    Iterator begin() const;

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

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

// ------------------------------------------------------------------------
-- //

template<typename T>
const TTree<T> &TTree<T>::removeSubTree(TTree<T>** aBranch) {
    if ((*aBranch)->empty()) {
        throw std::domain_error("Remove empty subtree");
    }

    auto& removedSubTree(**aBranch);
    *aBranch = &NIL;
    return removedSubTree;
}
```

```
template<typename T>
TTree<T>::TTree(const T& aKey) : fKey(aKey), fLeft(&NIL), fMiddle(&NIL),
fRight(&NIL) {

}

template<typename T>
TTree<T>::~TTree() {
    if (!fLeft->empty()) {
        delete fLeft;
    }
    if (!fMiddle->empty()) {
        delete fMiddle;
        fMiddle = &NIL;
    }
    if (!fRight->empty()) {
        delete fRight;
        fRight = &NIL;
    }
}

template<typename T>
const T& TTree<T>::operator*() const {
    if (empty()) {
        throw std::domain_error("Empty TTree encountered. ");
    }
    return fKey;
}

template<typename T>
bool TTree<T>::empty() const {
    return this == &NIL;

}

template<typename T>
bool TTree<T>::leaf() const {
    return fLeft->empty() && fMiddle->empty() && fRight->empty();

}

template<typename T>
TTree<T>::TTree(const TTree& aOtherTTree) :fLeft(&NIL), fMiddle(&NIL),
fRight(&NIL) {
    if (aOtherTTree.empty()) {
```

```
        throw std::domain_error("COPYING NIL");
    }
    fKey = aOtherTTree.fKey;

    if (!aOtherTTree.fLeft->empty()) {
        fLeft = new TTree(*aOtherTTree.fLeft);
    }
    if (!aOtherTTree.fMiddle->empty()) {
        fMiddle = new TTree(*aOtherTTree.fMiddle);
    }
    if (!aOtherTTree.fRight->empty()) {
        fRight = new TTree(*aOtherTTree.fRight);
    }
}

template <typename T>
TTree<T>& TTree<T>::operator=(const TTree& aOtherTTree) {
    TTree copy(aOtherTTree);

    std::swap(fKey, copy.fKey);
    std::swap(fLeft, copy.fLeft);
    std::swap(fMiddle, copy.fMiddle);
    std::swap(fRight, copy.fRight);

    return *this;
}

template <typename T>
TTree<T>* TTree<T>::clone() const
{
    if (!empty()) {
        return new TTree(*this);
    }
    else {
        return &NIL;
    }
}

template <typename T>
TTree<T>::TTree(T&& aKey) :fKey(std::move(aKey)), fLeft(&NIL), fMiddle(&NIL),
fRight(&NIL) {

}

template <typename T>
```

```
TTree<T>::TTree(TTree&& aOtherTTree) :fLeft(aOtherTTree.fLeft),
fMiddle(aOtherTTree.fMiddle), fRight(aOtherTTree.fRight) {
    if (aOtherTTree.empty()) {
        throw std::domain_error("MOVING NIL.");
    }

    fKey = std::move(aOtherTTree.fKey);

    aOtherTTree.fLeft = &NIL;
    aOtherTTree.fMiddle = &NIL;
    aOtherTTree.fRight = &NIL;
}

template <typename T>
TTree<T>& TTree<T>::operator=(TTree&& aOtherTTree) {
    if (aOtherTTree.empty()) {
        throw std::domain_error("MOVING NIL.");
    }

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

    }

    return *this;
}
```