# 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.

## 3-ary Trees and Prefix Traversal

We wish to define a generic 3-ary tree in C++. We shall call this data type `TernaryTree`. Our 3-ary tree has a payload key `fKey` and an array `fSubTrees` to store 3-ary subtrees. Following the principles underlying the definition of general trees, a 3-ary tree is a finite set of nodes and it is either

- an empty set, or
- a set that consists of a root and exactly 3 distinct 3-ary subtrees.

Somebody has already started with the implementation and created the header file `TernaryTree.h`, but left the project unfinished (see Canvas).

```
#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 );

  // add a subtree; must avoid memory leaks; may throw domain error [18]
  void addSubTree( size_t aSubtreeIndex, const TTree& aTTree );

public:

  // TernaryTree l-value constructor [10]
  TernaryTree( const T& aKey );

  // destructor (free sub-trees, must not free empty trees) [14]
  ~TernaryTree();

  // return key value, may throw domain_error if empty [6]
  const T& operator*() const;

  // returns true if this ternary tree is empty [4]
  bool empty() const;

  // returns true if this ternary tree is a leaf [10]
  bool leaf() const;

  // return height of ternary tree, may throw domain_error if empty [48]
  size_t height() const;

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

  // copy constructor, must not copy empty ternary tree [10]
  TernaryTree( const TTree& aOtherTTree );

  // copy assignment operator, must not copy empty ternary tree
  // may throw a domain error on attempts to copy NIL [36]
  TTree& operator=( const TTree& aOtherTTree );

  // clone ternary tree, must not copy empty trees [10]
  TSubTree clone() const;

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

  // TTree r-value constructor [12]
  TernaryTree( T&& aKey );

  // move constructor, must not copy empty ternary tree [12]
  TernaryTree( TTree&& aOtherTTree );

  // move assignment operator, must not copy empty ternary tree [36]
  TTree& operator=( TTree&& aOtherTTree );

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

  // return ternary tree prefix iterator positioned at start [4]
  Iterator begin() const;

  // return ternary prefix iterator positioned at end [6]
  Iterator end() const;
};

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

There are actual two template classes here: `TernaryTree<T>` and `TernaryTreePrefixIterator<T>`. The two template classes occur mutually dependent. However, as long as we do not use the iterator elements template class `TernaryTree<T>` can be safely implemented. The C++ compiler ignores unimplemented features that are not used.

The implementation of `TernaryTree<T>` is defined in three stages: basic infrastructure, copy control and, move semantics.

Once these stages are completed, we can focus our attention on the prefix iterator part.

You may always assume that previous steps produced a correct implementation. However, it does not guarantee that you can thoroughly test all steps.

## Problem 1 (132 marks)

Implement the basic `TernaryTree<T>` infrastructure:

- `const TTree<T>& removeSubTree( size_t aSubtreeIndex );`
- `void addSubTree( size_t aSubtreeIndex, const TTree& aTTree );`
- `TernaryTree( const T& aKey );`
- `~TernaryTree();`
- `const T& operator*() const;`
- `bool empty() const;`
- `bool leaf() const;`
- `size_t height() const;`

Use the available information to implement these features. The method `removeSubTree()` has to guarantee that empty trees are not removed. In this case, `removeSubTree()` has to throw a domain error. If the subtree can be removed, then a constant reference to it must be returned. In addition, the pointer of the subtree being removed must be set the address of `NIL` to indicate that this branch is now empty.

The method `addSubTree()` adds a new 3-ary subtree at index `aSubtreeIndex`. The index must be valid and the slot in the array `fSubTrees` must be available (i.e., set to the address of `NIL`). If any error occurs, the method `addSubTree()` must throw a corresponding exception.

To create `TernaryTree<T>` objects, we need to define its constructor, and the destructor releases the memory associated with `TernaryTree<T>` objects. The empty tree must not be deleted. It is unique and system-created.

In addition, there are four service functions: `operator*()`, `empty()`, `leaf()`, and `height()` that return the payload of a `TernaryTree<T>` object, test whether the current `TernaryTree<T>` object is the empty tree, check whether the current `TernaryTree<T>` object is a leaf node, and return the height of the current `TernaryTree<T>` object, respectively. Please note that the height of an empty tree is undefined.

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
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.
```

No other outputs or errors should occur.

## Problem 2 (56 marks)

Implement copy control for `TernaryTree<T>`:

- `TernaryTree( const TTree& aOtherTTree );`
- `TTree& operator=( const TTree& aOtherTTree );`
- `TSubTree clone() const;`

Use the available information to implement these features.

The copy control must not create copies of empty trees. If an empty tree is encountered in the copy constructor or assignment operator, then a domain error must be thrown. The method `clone()` can easily prevent copies of empty trees by returning the `this` object. You may need to apply suitable casts where necessary to make the implementation sound.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
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.
```

No other outputs or errors should occur. If the destructor and the elements of copy control work, then, at the end, when objects `root` and `copy` go out of scope, they are properly destroyed. No memory errors should occur.

## Problem 3 (60 marks)

Implement move semantics for `TTree<T>`:

- `TernaryTree( T&& aKey );`
- `TernaryTree( TTree&& aOtherTTree );`
- `TTree& operator=( TTree&& aOtherTTree );`

Use the available information to implement these features.

Move semantics avoids copying data when possible. We achieve move semantics by "stealing" the memory associated with the objects being moved. Move semantics uses r-value references. If you use an l-value as an argument to a move operation, then we should find that l-value empty after the move operation. We can use this feature to test out implementation.

You can use `#define P3` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
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.
```

No other outputs or errors should occur. When objects `root` and `copy` go out of scope, they are properly destroyed. No memory errors should occur.

## Problem 4 (98 marks)

We now wish to add a prefix iterator to `TernaryTree<T>`. Recall pre-order traversal, as shown in class:

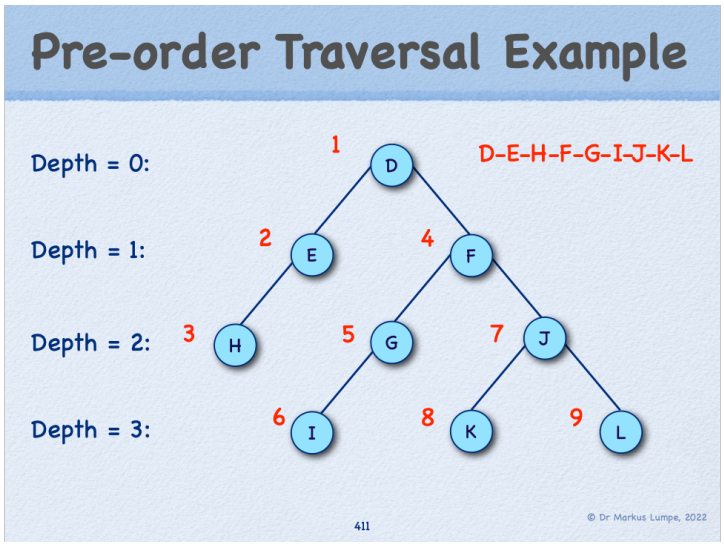


*Figure 1: Pre-order Tree Traversal.*

We wish to extend the basic pre-order traversal to a 3-ary tree. That is, when performing pre-order traversal, we

1. Visit the key,
2. Visit the left subtree,
3. Visit the middle subtree, and
4. Visit the right subtree.

This process is most easily realized via a recursive traversal procedure. However, we seek to implement it using a prefix iterator. That is, we need to use a stack to record the tree nodes that still need to be visited.

Initially, the stack contains only the root of the tree. This allows the iterator to access the key of the root node. When the iterator advances, the top node has to be removed from the stack and all its non-empty subtrees have to be pushed unto the stack from right to left. That is, once the top node has been popped, we need to push first the right subtree, next the middle subtree, and finally the left subtree, if they are not empty. Using this approach, we obtain pre-order traversal for 3-ary trees. To facilitate this task, we can use the method `push_subtrees()` that pushes all non-empty subtrees of the tree node argument unto the stack from right to left.

A suitable solution for a prefix iterator is given below:

```
#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 TTreeNode>;

  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 );

public:

  // iterator constructor [12]
  TernaryTreePrefixIterator( const TTree* aTTree );

  // iterator dereference [8]
  const T& operator*() const;

  // prefix increment [12]
  Iterator& operator++();

  // iterator equivalence [12]
  bool operator==( const Iterator& aOtherIter ) const;

  // auxiliaries [4,10]
  Iterator begin() const;
  Iterator end() const;
};
```

Template class `TernaryTreePrefixIterator<T>` defines a standard forward iterator. To facilitate its implementation there is a private member function `push_subtrees()`. This function takes a pointer to a constant `TernaryTree<T>` object and pushes the corresponding subtrees from right to left unto the traversal stack, if there are any.

The prefix increment always removes the top element from the stack. Next, the subtrees of the top element have to be pushed onto the traversal stack from right to left. This yields the required pre-order traversal of a 3-ary tree.

The other iterator methods are defined in the usual way. The constructor has to set up the initial stack. That is, the root node must be pushed onto the traversal stack, if it is not empty.

The equivalence operator tests the addresses of the trees and the respective stack sizes. The iterator is at the end, if the traversal stack is empty.

To complete the solution, you need to implement the iterator methods for class `TernaryTree<T>`. They are used to map for-range loops to plain for loops in C++. The compiler will report "undefined symbol" if these methods have not been implemented.

You can use `#define P4` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

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

No other outputs or errors should occur.

## 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)**