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: Tran Quoc Dung **Your student id:** 103803891

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.

Final Exam 2022

File: TernaryTree.h

```
#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);
    }

    /////////////////STARTING WITH PROBLEMS
    // Problem 1: TernaryTree Basic Infrastructure

private:

    // remove a subtree, may throw a domain error [22]
    const TTree& removeSubTree(size_t aSubtreeIndex)
    {
        if (fSubTrees[aSubtreeIndex]->empty()) throw domain_error("Subtree is empty");
        if (aSubtreeIndex >= 3) throw out_of_range("Illegal subtree index");
        const TTree& lSubtreeIndex = const_cast<TTree&>(*fSubTrees[aSubtreeIndex]);
        fSubTrees[aSubtreeIndex] = &NIL;
        return lSubtreeIndex;
    }

    // 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("Empty tree");
        if (aSubtreeIndex >= 3) throw out_of_range("Illegal index");
        if (!fSubTrees[aSubtreeIndex]->empty()) throw domain_error("Non-empty subtree present");
        fSubTrees[aSubtreeIndex] = const_cast<TTree*>(&aTTree);
    }

public:

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

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

    // return height of ternary tree, may throw domain_error if empty [48]
    size_t height() const
    {
        if (empty()) throw domain_error("Empty tree encountered");
        if (leaf()) return 0;
        size_t lHeight[3] = {};

        for (size_t i = 0; i < 3; i++) lHeight[i] = fSubTrees[i]->empty() ? 0 : fSubTrees[i]->height();
        return *std::max_element(lHeight, lHeight + 3) + 1;
    }

    // Problem 2: TernaryTree Copy Semantics

        // copy constructor, must not copy empty ternary tree
    TernaryTree(const TTree& aOtherTTree)
    {
        for (size_t i = 0; i < 3; i++) fSubTrees[i] = &NIL;
        *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)
            if (!aOtherTTree.empty())
            {
                this->~TernaryTree();
                fKey = aOtherTTree.fKey;
                for (size_t i = 0; i < 3; i++)
                    if (!aOtherTTree.fSubTrees[i]->empty()) fSubTrees[i] = aOtherTTree.fSubTrees[i]-
>clone();
                    else fSubTrees[i] = &NIL;
            }
            else throw domain_error("Cannot copy NIL");
        return *this;
    }

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

    // Problem 3: TernaryTree Move Semantics

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

    // move constructor, must not copy empty ternary tree
    TernaryTree(TTree&& aOtherTTree)
    {
        for (size_t i = 0; i < 3; i++) fSubTrees[i] = &NIL;
        *this = move(aOtherTTree);
    };
    // move assignment operator, must not copy empty ternary tree
    TTree& operator=(TTree&& aOtherTTree)
    {
        if (this != &aOtherTTree)
            if (!aOtherTTree.empty())
            {
                this->~TernaryTree();
                fKey = move(aOtherTTree.fKey);
                for (size_t i = 0; i < 3; i++)
                    if (!aOtherTTree.fSubTrees[i]->empty()) fSubTrees[i] =
const_cast<TSubTree>(&aOtherTTree.removeSubTree(i));
                    else fSubTrees[i] = &NIL;
            }
            else throw domain_error("Cannot move NIL");
    }

    // 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;
```

File: TernaryTreePrefixIterator.h

```
#pragma once

#include "TernaryTree.h"

#include <stack>

using namespace std;

template<typename T>
class TernaryTreePrefixIterator
{
private:
   using TTree = TernaryTree<T>;
   using TTreeNode = TTree*;
   using TTreeStack = 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
   void push_subtrees(const TTree* aNode)
   {
      if (!(*aNode).getRight().empty())
fStack.push(const_cast<TTreeNode>(&(*aNode).getRight()));
      if (!(*aNode).getMiddle().empty())
fStack.push(const_cast<TTreeNode>(&(*aNode).getMiddle()));
```

```
    if (!(*aNode).getLeft().empty()) fStack.push(const_cast<TTreeNode>(&(*aNode).getLeft()));
  }

public:

  // iterator constructor
  TernaryTreePrefixIterator(const TTree* aTTree) :fTTree(aTTree), fStack()
  {
    if (!(*fTTree).empty())
      fStack.push(const_cast<TTreeNode>(fTTree));
  }

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

  // prefix increment
  Iterator& operator++()
  {

    TTreeNode lPopped = const_cast<TTreeNode>(fStack.top());
    fStack.pop();
    push_subtrees(lPopped);
    return *this;
  }

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

  // auxiliaries
  Iterator begin() const
  {
    Iterator lTmp = *this;
    lTmp.fStack = TTreeStack();
    lTmp.fStack.push(const_cast<TTreeNode>(lTmp.fTTree));
    return lTmp;
  }
  Iterator end() const
  {
    Iterator lTmp = *this;
    lTmp.fStack = TTreeStack();
    return lTmp;
  }
};
```

File: Main.cpp

```
#include <iostream>
#include <string>
#include <stdexcept>

using namespace std;

#define P1
#define P2
#define P3
#define P4

// Test keys
string s1("This");
string s2("is");
string s3("a");
string s4("ternary");
string s5("tree");
string s6("in");
string s7("action.");
string s8("It");
string s9("works!");

#ifdef P1

#include "TernaryTree.h"

void runP1()
{
   cout << "Test Problem 1:" << endl;

   using S3Tree = TernaryTree<string>;

   cout << "Setting up ternary tree..." << endl;

   S3Tree root(s1);
   S3Tree* nA = new S3Tree(s2);
   S3Tree* nB = new S3Tree(s5);
   S3Tree* nC = new S3Tree(s7);
   S3Tree nAA(s3);
   S3Tree nAAC(s4);
   S3Tree nBB(s6);
   S3Tree nCB(s8);
   S3Tree nCC(s9);

   nAA.addRight(nAAC);
   nA->addLeft(nAA);

   nB->addMiddle(nBB);
```

```
nC->addMiddle(nCB);
nC->addRight(nCC);

root.addLeft(*nA);
root.addMiddle(*nB);
root.addRight(*nC);

try
{
   root.addRight(*nC);

   cerr << "Error: Non-empty subtree overridden." << endl;
}
catch (std::domain_error e)
{
   cout << "Successfully caught: " << e.what() << endl;
}

cout << "Testing basic ternary tree logic ..." << endl;

cout << "Is NIL empty? " << (S3Tree::NIL.empty() ? "Yes" : "No") << endl;
cout << "Is root empty? " << (root.empty() ? "Yes" : "No") << endl;

try
{
   cout << "Height of root is: " << root.height() << endl;

   S3Tree::NIL.height();

   cerr << "Error: NIL has no height." << endl;
}
catch (std::domain_error e)
{
   cout << "Successfully caught: " << e.what() << endl;
}

cout << "Tearing down ternary tree..." << endl;

nC->removeRight();
nC->removeMiddle();
nB->removeMiddle();
nAA.removeRight();
nA->removeLeft();

try
{
   nA->removeLeft();

   cerr << "Error: Empty subtree removed." << endl;
}
catch (std::domain_error e)
```

```
    {
        cout << "Successfully caught: " << e.what() << endl;
    }

    cout << "Nodes nA, nB, nC get destroyed by destructor." << endl;

    cout << "Test Problem 1 complete." << endl;
}

#endif

#ifdef P2

#include "TernaryTree.h"

void runP2()
{
    cout << "Test Problem 2:" << endl;

    using S3Tree = TernaryTree<string>;

    S3Tree root(s1);
    S3Tree* nA = new S3Tree(s2);
    S3Tree* nB = new S3Tree(s5);
    S3Tree* nC = new S3Tree(s7);
    S3Tree* nAA = new S3Tree(s3);
    S3Tree* nAAC = new S3Tree(s4);
    S3Tree* nBB = new S3Tree(s6);
    S3Tree* nCB = new S3Tree(s8);
    S3Tree* nCC = new S3Tree(s9);

    nAA->addRight(*nAAC);
    nA->addLeft(*nAA);

    nB->addMiddle(*nBB);

    nC->addMiddle(*nCB);
    nC->addRight(*nCC);

    root.addLeft(*nA);
    root.addMiddle(*nB);
    root.addRight(*nC);

    S3Tree copy = root;

    const S3Tree* lLeft;
    const S3Tree* lRight;

    lLeft = &copy.getLeft().getLeft().getRight();
    lRight = &root.getLeft().getLeft().getRight();
```

```
if (lLeft == lRight)
{
   cerr << "Error: Shallow copy detected." << endl;
}
else
{
   cout << "Copy constructor appears to work properly." << endl;
}

lLeft = &copy.getMiddle().getLeft();
lRight = &root.getMiddle().getRight();

if (lLeft != lRight)
{
   cerr << "Error: Copy does not preserve tree structure." << endl;
}
else
{
   if (!lLeft->empty())
   {
      cerr << "Error: NIL not preserved." << endl;
   }
   else
   {
      cout << "Copy constructor preserves tree structure." << endl;
   }
}

root = copy;

lLeft = &copy.getLeft().getLeft().getRight();
lRight = &root.getLeft().getLeft().getRight();

if (lLeft == lRight)
{
   cerr << "Error: Shallow copy detected." << endl;
}
else
{
   cout << "Assignment appears to work properly." << endl;
}

lLeft = &copy.getMiddle().getLeft();
lRight = &root.getMiddle().getRight();

if (lLeft != lRight)
{
   cerr << "Error: Assignment does not preserve tree structure." << endl;
}
else
{
        if (!lLeft->empty())
        {
            cerr << "Error: NIL not preserved." << endl;
        }
        else
        {
            cout << "Assignment preserves tree structure." << endl;
        }
    }

    try
    {
        root = S3Tree::NIL;

        cerr << "Error: Copy of NIL! You should not see this message." << endl;
    }
    catch (domain_error e)
    {
        cout << "Successfully caught: " << e.what() << endl;
    }

    S3Tree* clone = root.clone();

    lLeft = &clone->getLeft().getLeft().getRight();
    lRight = &root.getLeft().getLeft().getRight();

    if (lLeft == lRight)
    {
        cerr << "Error: Shallow copy detected." << endl;
    }
    else
    {
        cout << "Clone appears to work properly." << endl;
    }

    delete clone;

    cout << "Trees root and copy get deleted next." << endl;
    cout << "Test Problem 2 complete." << endl;
}

#endif

#ifdef P3

#include "TernaryTree.h"

void runP3()
{
    cout << "Test Problem 3:" << endl;
```

```
using S3Tree = TernaryTree<string>;

S3Tree root(string("This"));
S3Tree* nA = new S3Tree(s2);
S3Tree* nB = new S3Tree(s5);
S3Tree* nC = new S3Tree(s7);
S3Tree* nAA = new S3Tree(s3);
S3Tree* nAAC = new S3Tree(s4);
S3Tree* nBB = new S3Tree(s6);
S3Tree* nCB = new S3Tree("It");
S3Tree* nCC = new S3Tree(s9);

nAA->addRight(*nAAC);
nA->addLeft(*nAA);

nB->addMiddle(*nBB);

nC->addMiddle(*nCB);
nC->addRight(*nCC);

root.addLeft(*nA);
root.addMiddle(*nB);
root.addRight(*nC);

S3Tree copy = std::move(root);

if (root.leaf())
{
   cout << "std::move makes root a leaf node." << endl;
}
else
{
   cerr << "Error: You should not see this message as root must become a leaf node." << endl;
}

cout << "The payload of tree: " << *copy << endl;
cout << "The payload of tree.getLeft().getLeft().getRight():\t" <<
*copy.getLeft().getLeft().getRight() << endl;
cout << "The payload of tree.getRight():\t" << *copy.getRight() << endl;

root = std::move(copy);

if (copy.leaf())
{
   cout << "std::move makes copy a leaf node." << endl;
}
else
{
   cerr << "Error: You should not see this message as copy must become a leaf node." << endl;
}
```

```
    cout << "The payload of tree: " << *root << endl;
    cout << "The payload of tree.getLeft().getLeft().getRight():\t" <<
*root.getLeft().getLeft().getRight() << endl;
    cout << "The payload of tree.getRight():\t" << *root.getRight() << endl;

    try
    {
        root = std::move(S3Tree::NIL);

        cerr << "Error: Move of NIL! You should not see this message." << endl;
    }
    catch (domain_error e)
    {
        cout << "Successfully caught: " << e.what() << endl;
    }

    cout << "Test Problem 3 complete." << endl;
}

#endif

#ifdef P4

#include "TernaryTreePrefixIterator.h"

void runP4()
{
    cout << "Test Problem 4:" << endl;

    using S3Tree = TernaryTree<string>;

    S3Tree root(s1);
    S3Tree* nA = new S3Tree(s2);
    S3Tree* nB = new S3Tree(s5);
    S3Tree* nC = new S3Tree(s7);
    S3Tree* nAA = new S3Tree(s3);
    S3Tree* nAAC = new S3Tree(s4);
    S3Tree* nBB = new S3Tree(s6);
    S3Tree* nCB = new S3Tree(s8);
    S3Tree* nCC = new S3Tree(s9);

    nAA->addRight(*nAAC);
    nA->addLeft(*nAA);

    nB->addMiddle(*nBB);

    nC->addMiddle(*nCB);
    nC->addRight(*nCC);

    root.addLeft(*nA);
    root.addMiddle(*nB);
    root.addRight(*nC);

    cout << "Test prefix iterator:";

    for (const string& k : root)
    {
        cout << ' ' << k;
    }

    cout << endl;

    cout << "Test Problem 4 complete." << endl;
}

#endif

int main()
{
#ifdef P1
    runP1();
    cout << "\n" << endl;
#endif

#ifdef P2
    runP2();
    cout << "\n" << endl;
#endif

#ifdef P3
    runP3();
    cout << "\n" << endl;
#endif

#ifdef P4
    runP4();
    cout << "\n" << endl;
#endif

    return 0;
}
```

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)

By using a root pointer to point to the same degree, we will be able to construct a tree with all nodes have the same degree.

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

5b)

"l-value" illustrate the location in memory of an object and uses &, while "r-value" evaluate the value stored in memory and uses &&.

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

5c)

Abstract data types mainly hide the detail information, showing only necessary data to users. To elaborate, it only mentions what to execute (function... etc.) but not how those executions process step-by-step.

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

5d)

Mutual dependant classes are classes that relies on each other functionality, can be recognized when the code in one class mention the other classes.
For instance, two class A and B and in class A if we see a variable crated based on class B, those classes are mutually dependant.

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

5e)

A value-based data type element directly contains a data value within its own memory space, must always specify what type it is.

f. What is an object adapter? [6]

5f)

The object adaptor allows items to collaborate with one another, especially when those two are incompatible by taking calls from other objects and convert them into another format that other objects can comprehend.

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

5g)

We use the copy constructor when an existing object is copied into a new object, and the assignment operator is used when we assign a new value from another existing object to an initialized object. Ensuring the object is empty or not is quite necessary for safe operation.

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

5h)

Best-case (Lower Bound): O(1), when it searches for the first element
Worst-case (Upper Bound): 0(n), when it searches for the last element, and
we will have to traverse n elements
Average-case: O(log n), in an array of size n takes on average n/2

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

5i)

Reference data members is kind of an object with a specific data type. In order to initialize reference data members, we initialize the object's "l-value" or "r-value", using &(reference) and =(asign) sign.

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)

On this condition, by using the algorithm of calculating all the numbers in an array (such as n*(n+1)/2 for the first n number), we can calculate the sum of all the first n natural numbers. Then subtract it with the sum of all n-1 numbers above, and we will be able to find the missing number.

Time complexity: O(n)