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: Nov 26, 2024, 11:00
Lecturer: Dr. Van Dai Pham

Your name: Nguyen Duc Chung **Your student id: 104972970**

Check Tutorial	Tues 08:00	Tues 13:00	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30
		X				

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.

TernaryTree.h:

```
// COS30008, Final Exam

#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)
   {

      if (fSubTrees[aSubtreeIndex] == &NIL)
      {
         throw std::domain_error("Subtree is NIL");
      }
      const TTree& subtree = *fSubTrees[aSubtreeIndex];
fSubTrees[aSubtreeIndex] = &NIL;
      return subtree;
   }

   // add a subtree; must avoid memory leaks; may throw domain error [18]
void addSubTree(size_t aSubtreeIndex, const TTree& aTTree)
   {
      if (fSubTrees[aSubtreeIndex] != &NIL)
      {
         throw std::domain_error("Subtree is not NIL");
      }
      fSubTrees[aSubtreeIndex] = new TTree(aTTree);
   }

public:

   // TernaryTree l-value constructor [10]
TernaryTree(const T& aKey)
   {        fKey = aKey;
fSubTrees[0] = &NIL;
fSubTrees[1] = &NIL;
fSubTrees[2] = &NIL;
   }

   // destructor (free sub-trees, must not free empty trees) [14]
   ~TernaryTree()
   {
      if (fSubTrees[0] != &NIL)
      {
         delete fSubTrees[0];
         fSubTrees[0] = &NIL;
      }
      if (fSubTrees[1] != &NIL)
      {
         delete fSubTrees[1];
         fSubTrees[1] = &NIL;
      }
      if (fSubTrees[2] != &NIL)
      {
         delete fSubTrees[2];
         fSubTrees[2] = &NIL;
      }
   }

   // return key value, may throw domain_error if empty [2]
const T& operator*() const
   {
      if (empty())
```

```
      {
         throw std::domain_error("Operation not supported");
      }
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
   {
      return fSubTrees[0] == &NIL && fSubTrees[1] == &NIL && fSubTrees[2] == &NIL;
   }

   // 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
height[3] = {};

      if (!fSubTrees[0]->empty())
      {
         height[0] = fSubTrees[0]->height();
      }
else        {
         height[0] = 0;
      }

      if (!fSubTrees[1]->empty())
      {
         height[1] = fSubTrees[1]->height();
      }
else        {
         height[1] = 0;
      }
      if (!fSubTrees[2]->empty())
      {
         height[2] = fSubTrees[2]->height();
      }
else        {
         height[2] = 0;
      }


      return std::max(std::max(height[0], height[1]), height[2]) + 1;
   }
```

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

      // copy constructor, must not copy empty ternary tree
   TernaryTree(const TTree& aOtherTTree) : fKey(aOtherTTree.fKey)




 {        size_t i =
0;        while (i <
3)
 {           if (aOtherTTree.fSubTrees[i] !=
&NIL)
          {
  fSubTrees[i] = new TTree(*aOtherTTree.fSubTrees[i]);
          }
else            {
             fSubTrees[i] = &NIL;
          }
          ++i;
       }

   }

   // 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;
       }
size_t i = 0;
while (i < 3)
       {
          fSubTrees[i] != &NIL
? delete fSubTrees[i]
             : void();

          fSubTrees[i += 1] = &NIL;
       }

       fKey = aOtherTTree.fKey;
i = 0;
       while (i < 3)
       {
          fSubTrees[i] = (aOtherTTree.fSubTrees[i] != &NIL)
             ? new TTree(*aOtherTTree.fSubTrees[i])
             : &NIL;
          ++i;
}        return
*this;

   }

   // clone ternary tree, must not copy empty trees
TSubTree clone() const
   {
      TSubTree newTree = new TTree(fKey);

      for (size_t i = 0; i < 3; i++)
```

```
      {
         if (fSubTrees[i] != &NIL)
         {
            newTree->fSubTrees[i] = fSubTrees[i]->clone();
         }
else
         {
            newTree->fSubTrees[i] = &NIL;
         }

         return newTree;
      }
   }


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

      // TTree r-value constructor
   TernaryTree(T&& aKey) : fKey(std::move(aKey))
   {
      fSubTrees[0] = &NIL;
fSubTrees[1] = &NIL;        fSubTrees[2]
= &NIL;
   }



   // move constructor, must not copy empty ternary tree
   TernaryTree(TTree&& aOtherTTree) : fKey(std::move(aOtherTTree.fKey))
   {
      for (size_t i = 0; i < 3; i++)
      {
         if (aOtherTTree.fSubTrees[i] != &NIL)
         {
            fSubTrees[i] = aOtherTTree.fSubTrees[i];
            aOtherTTree.fSubTrees[i] = &NIL;
         }
else           {
            fSubTrees[i] = &NIL;
         }
      }
      aOtherTTree.fKey = T();

   }

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

      if (this != &aOtherTTree)
      {            int i
= 0;           while (i
< 3)
         {
            if (fSubTrees[i] != &NIL)
```

```
            {                   delete
fSubTrees[i];
            }
fSubTrees[i] = &NIL;
            i++;
         }

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




    i = 0;
while (i < 3)
         {
            fSubTrees[i] = aOtherTTree.fSubTrees[i];
aOtherTTree.fSubTrees[i] = &NIL;
            i++;
         }
      }

      aOtherTTree.fKey = T();

       return
*this;
   }


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

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

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

};

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

TernaryTreePrefixIterator.h:

```
#pragma once

// COS30008, Final Exam

#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)
   {        if (!(aNode-
>getRight().empty()))
      {
         const TTree* rightSubtree = &aNode->getRight();
         fStack.push(const_cast<TTreeNode>(rightSubtree));
      }

      if (!(aNode->getMiddle().empty()))
      {
         const TTree* middleSubtree = &aNode->getMiddle();
         fStack.push(const_cast<TTreeNode>(middleSubtree));
      }

   }



public:

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

         if (!(fTTree->empty()))
         {
            const TTree* tempTree = fTTree;
            fStack.push(const_cast<TTreeNode>(tempTree));
         }
      }

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

   // prefix increment [12]
   Iterator& operator++()
   {
      TTreeNode poppedNode = const_cast<TTreeNode>(fStack.top());
fStack.pop();        if (poppedNode != nullptr)
      {
         push_subtrees(poppedNode);
      }
else
      {
         throw std::logic_error("Null node encountered while incrementing.");
      }
return *this;
   }

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

```
   {        if (fTTree !=
aOtherIter.fTTree)
      {
         return false;
      }        else        {            if (fStack.size()
!= aOtherIter.fStack.size())
         {
            return false;
         }
else            {
            return true;
         }
      }
   }
```

```
  // auxiliaries [4,10]
   Iterator begin() const
   {
      Iterator iteratorCopy = *this;

     if (!(iteratorCopy.fTTree->empty()))
      {
         iteratorCopy.fStack = TTreeStack();            const
TTree* root = iteratorCopy.fTTree;
iteratorCopy.fStack.push(const_cast<TTreeNode>(root));
      }

      return iteratorCopy;
   }

   Iterator end() const
   {
      Iterator iteratorCopy = *this;

      if (!iteratorCopy.fStack.empty())
      {
         iteratorCopy.fStack = TTreeStack();
      }

      return iteratorCopy;
   }
};
```

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)

We can make a tree where all the nodes have the same degree by giving every node the same number of child nodes, except for the leaf nodes which don't have any children.

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]

It's a kind of thing (or class) where what it does is all about the values it can have and the actions you can do with it

5c)

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

5d)

Only need to use forward declarations. We will inform the compiler about a specific class ve define that class, so the others will be able to use it.

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

5e)

Value-based need to copy constructor, assign it with assignment operator, and clean up it up (destructor).

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]

Best-case: is when you find the value right at the root of the tree, and it only takes O(1) time.

Average-case: The average time it takes to find something is almost the same as the worst time, and both are O(logN).

worse-case: when you find the value at the very bottom of the tree, and it takes O(log n) or depends on how tall the tree is.

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 ≤ k ≤ n, in linear time? [8]

Step 1 - using n * (n + 1) / 2 to get the total of numbers from 1 to n,

Step 2 - subtract the sum of the numbers given, the result is the missing one.

5j)