Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Assignment number and title:	4, Binary Search Trees & In-Order Traversal
Due date:	May 26, 2022, 14:30
Lecturer:	Dr. Markus Lumpe

Your name:Nguyen Duc Chung

Your student id: 104972970

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	Obtained
1	94	
2	42	
3	8+86=94	
Total	230	

Extension certification:

This assignment has been given an extension and is now due on ____________________

Signature of Convener:____________________________

Problem 1 - BinaryTreeNode.h

```
#pragma once
#include <stdexcept>
#include <algorithm>

template<typename T>
struct BinaryTreeNode
{
    using BNode = BinaryTreeNode<T>;
    using BTreeNode = BNode*;

    T key;
    BTreeNode left;
    BTreeNode right;
    static BNode NIL;
   const T& findMax() const
   {
       if (empty())
       {
           throw std::domain_error("Empty tree encountered in findMax.");
       }
       return right->empty() ? key : right->findMax();
   }
const T& findMin() const
{
    if (empty())
    {
        throw std::domain_error("Empty tree encountered in findMin.");
    }
    return left->empty() ? key : left->findMin();
```

```
}
 bool remove(const T& aKey, BTreeNode aParent)
 {
    BTreeNode x = this;
    BTreeNode y = aParent;
    while (!x->empty())
    {
       if (aKey == x->key)
       {
          break;
       }
       y = x; // new parent
       x = aKey < x->key ? x->left : x->right;
    }
    if (x->empty())
    {
       return false; // delete failed
    }
    if (!x->left->empty())
    {
       const T& lKey = x->left->findMax(); // find max to left
       x->key = lKey;
       x->left->remove(lKey, x);
    }
    else
    {
       if (!x->right->empty())
       {
          const T& lKey = x->right->findMin(); // find min to right
          x->key = lKey;
          x->right->remove(lKey, x);
```

```
      }
      else
      {
         if (y != &NIL) // y can be NIL
         {
            if (y->left == x)
            {
               y->left = &NIL;
            }
            else
            {
               y->right = &NIL;
            }
         }
         delete x; // free deleted node
      }
   }
   return true;
}
BinaryTreeNode() : key(T()), left(&NIL), right(&NIL) {}
BinaryTreeNode(const T& aKey) : key(aKey), left(&NIL), right(&NIL) {}
BinaryTreeNode(T&& aKey) : key(std::move(aKey)), left(&NIL), right(&NIL) {}
~BinaryTreeNode()
{
   if (left != &NIL && !left->empty())
      delete left;

   if (right != &NIL && !right->empty())
      delete right;
}
  bool empty() const
```

```
{
    return this == &NIL;
}

bool leaf() const
{
    return !empty() && left->empty() && right->empty();
}

size_t height() const
{
    if (empty())
    {
        throw std::domain_error("Empty tree encountered in height.");
    }

    size_t leftHeight = left->empty() ? 0 : left->height() + 1;
    size_t rightHeight = right->empty() ? 0 : right->height() + 1;

    return std::max(leftHeight, rightHeight);
}

bool insert(const T& aKey)
{
    if (empty())
    {
        return false;
    }

    if (aKey < key)
    {
        if (left->empty())
        {
          left = new BNode(aKey);
          return true;
        }
        else
        {
          return left->insert(aKey);
        }
    }
    else if (key < aKey)
    {
        if (right->empty())
        {
          right = new BNode(aKey);
          return true;
        }
        else
        {
          return right->insert(aKey);
        }
    }
    return false; // Duplicate key
}
};

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

Problem 2 - BinarySearchTree.h:

```
// BinarySearchTree.h

// COS30008, Problem Set 4, Problem 2, 2022
#pragma once
#include "BinaryTreeNode.h"
#include <stdexcept>
// Problem 3 requirement
template<typename T>
class BinarySearchTreeIterator;
template<typename T>
class BinarySearchTree
{
private:
using BNode = BinaryTreeNode<T>;
using BTreeNode = BNode*;
BTreeNode fRoot;
public:
BinarySearchTree() : fRoot(&BNode::NIL) {}
~BinarySearchTree() {
if (!fRoot->empty())
{
delete fRoot;
}
}
bool empty() const {
return fRoot->empty();
}
size_t height() const {
if (empty())
```

```
{
throw std::domain_error("Empty tree has no height.");
}
return fRoot->height();
}
bool insert(const T& aKey) {
if (fRoot == &BNode::NIL)
{
fRoot = new BNode(aKey);
return true;
}
return fRoot->insert(aKey);
}
bool remove(const T& aKey) {
if (fRoot != &BNode::NIL)
{
if (fRoot->key == aKey)
{
// Handle removal of root
if (fRoot->left->empty() && fRoot->right->empty())
{
delete fRoot;
fRoot = &BNode::NIL;
}
else if (!fRoot->left->empty())
{
const T& lMax = fRoot->left->findMax();
fRoot->key = lMax;

fRoot->left->remove(lMax, fRoot);
}
else
{
const T& lMin = fRoot->right->findMin();
fRoot->key = lMin;

fRoot->right->remove(lMin, fRoot);
}
return true;
}
else
{
return fRoot->remove(aKey, nullptr);
}
}
return false;
}
// Problem 3 methods
using Iterator = BinarySearchTreeIterator<T>;
// Allow iterator to access private member variables
friend class BinarySearchTreeIterator<T>;
Iterator begin() const {
Iterator iter(*this);
return iter;
}
Iterator end() const {
Iterator iter(*this);
iter.clearStack();
return iter;
}
};
```

Problem 3 -BinarySearchTreeIterator.h:

```
// COS30008, Problem Set 4, Problem 3, 2022
#pragma once
#include "BinarySearchTree.h"
#include <stack>
template<typename T>
class BinarySearchTreeIterator
{
private:
using BSTree = BinarySearchTree<T>;
using BNode = BinaryTreeNode<T>;
using BTreeNode = BNode*;
using BTNStack = std::stack<BTreeNode>;

const BSTree& fBSTree; // binary search tree
BTNStack fStack; // DFS traversal stack

void pushLeft( BTreeNode aNode ) {
  while (!aNode->empty())
  {
    fStack.push(aNode);
    aNode = aNode->left;
  }
}
public:

using Iterator = BinarySearchTreeIterator<T>;

BinarySearchTreeIterator(const BSTree& aBSTree) : fBSTree(aBSTree) {
  pushLeft(fBSTree.fRoot);
}
const T& operator*() const {
  return fStack.top()->key;
}
Iterator& operator++() {

  BTreeNode currentNode = fStack.top();
  fStack.pop();

  if (!currentNode->right->empty())
  {
    pushLeft(currentNode->right);
  }
  return *this;
}
Iterator operator++(int) {

  Iterator temp = *this;
  ++(*this);
  return temp;
}
void clearStack()
{
  fStack = BTNStack(); // Assign an empty stack to clear it
}

bool operator==( const Iterator& aOtherIter ) const {

  return fStack == aOtherIter.fStack;
```

```
}
bool operator!=( const Iterator& aOtherIter ) const {

    return !(*this == aOtherIter);
}
Iterator begin() const {

    return Iterator(fBSTree);
}
Iterator end() const {

    Iterator temp(fBSTree);
    temp.fStack = BTNStack(); // Assign an empty stack to indicate the end
    return temp;
}
};
```