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: Dao Khanh Nga Thi **Your student id:** 104177393

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
				X							

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

```
// COS30008, Problem Set 4, Problem 1, 2022

#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." );
        }

        return right->empty() ? key : right->findMax();
    }

    const T& findMin() const
    {
        if ( empty() )
        {
            throw std::domain_error( "Empty tree encountered." );
        }

        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;
}

// PS4 starts here

BinaryTreeNode() : key(T()), left(&NIL), right(&NIL) {}
BinaryTreeNode(const T& aKey) : key(aKey), left(&NIL), right(&NIL) {}
BinaryTreeNode(T&& aKey) : key(move(aKey)), left(&NIL), right(&NIL) {}
~BinaryTreeNode() {
    deleteSubtree(left);
    deleteSubtree(right);
}

// Checks if this node is empty
bool empty() const
{
    return this == &NIL;
}

// Checks if this node is a leaf node
```

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

    // Returns the height of the tree rooted at this node
    size_t height() const
    {
        if (empty())
        {
            throw domain_error("Empty tree encountered");
        }
        if (leaf())
        {
            return 0;
        }
        const int left_height = left->empty() ? 1 : left->height() + 1;
        const int right_height = right->empty() ? 1 : right->height() + 1;
        return max(left_height, right_height);
    }

    // Inserts a new node with the given key into the tree rooted at this node
    bool insert(const T& aKey) {
        if (empty()) {
            return false; // Cannot insert into an empty tree.
        }

        BNode* currentNode = this;

        while (true) {

            if (aKey > currentNode->key) {

                if (currentNode->right->empty()) {
                    // If the right child is empty, insert a new node with the given
key.
                    currentNode->right = new BNode(aKey);
                    return true; // Insertion successful.
                }

                else {
                    // Move to the right subtree.
                    currentNode = currentNode->right;
                }
            }

            else if (aKey < currentNode->key) {

                if (currentNode->left->empty()) {
                    // If the left child is empty, insert a new node with the given
key.
                    currentNode->left = new BNode(aKey);
                    return true; // Insertion successful.
                }

                else {
                    // Move to the left subtree.
                    currentNode = currentNode->left;
```

```
                }
            }

            else {
                return false; // Duplicate key, insertion failed.
            }
        }
    }

    private:
        void deleteSubtree(BinaryTreeNode<T>*& node) {
            if (!node->empty()) {
                delete node;
                node = &NIL;
            }
        }
};

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

Problem 2: BinarySearchTree.h

```
#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()
	{
		while (!fRoot->empty())
		{
			delete fRoot;
		}
	}

	// Checks if the binary search tree is empty
	bool empty() const
	{
		return fRoot->empty();
	}

	// Returns the height of the binary search tree
```

```
size_t height() const
{
	while (empty())
	{
		throw domain_error("Empty tree has no height.");
	}
	return fRoot->height();
}

// Inserts a new key into the binary search tree
bool insert(const T& aKey)
{

	if (empty())
	{
		fRoot = new BNode(aKey);
		return true;
	}

	BTreeNode currentNode = fRoot;
	while (true)
	{

		if (aKey < currentNode->key)
		{

			if (currentNode->left == &BNode::NIL)
			{
				currentNode->left = new BNode(aKey);
				return true;
			}
			currentNode = currentNode->left;
		}

		else if (aKey > currentNode->key)
		{

			if (currentNode->right == &BNode::NIL)
			{
				currentNode->right = new BNode(aKey);
				return true;
			}
			currentNode = currentNode->right;
		}

		else
		{
			return false; // Key already exists in the tree
		}
	}
}

// Removes a key from the binary search tree
bool remove(const T& aKey)
{

	if (empty())
	{
```

```
                throw domain_error("Unable to remove from an empty tree.");
            }

            if (fRoot->leaf())
            {

                if (fRoot->key == aKey)
                {
                    fRoot = &BNode::NIL;
                    return true;
                }
                return false;
            }

            return fRoot->remove(aKey, &BNode::NIL);
        }

        // Problem 3 methods

        using Iterator = BinarySearchTreeIterator<T>;
        // Allow iterator to access private member variables
        friend class BinarySearchTreeIterator<T>;
        Iterator begin() const
        {
            return Iterator(*this).begin();
        }
        Iterator end() const
        {
            return Iterator(*this).end();
        }
};
```

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)
```

```cpp
	{
		while (!aNode->empty())
		{
			fStack.push(aNode);
			aNode = aNode->left;
		}
	}

public:

	using Iterator = BinarySearchTreeIterator<T>;

	BinarySearchTreeIterator(const BSTree& aBSTree) : fBSTree(aBSTree), fStack()
	{
		pushLeft(aBSTree.fRoot);
	}

	// Dereference operator to get the value at the current position
	const T& operator*() const
	{
		return fStack.top()->key;
	}

	// Pre-increment operator (++iter)
	Iterator& operator++()
	{
		BTreeNode lPop = fStack.top();
		fStack.pop();
		pushLeft(lPop->right);
		return *this;
	}

	// Post-increment operator (iter++)
	Iterator operator++(int)
	{
		// Create a temporary iterator and move it to the end of the tree
		Iterator temp = *this;

		while (!temp.fStack.empty())
		{
			BTreeNode currentNode = temp.fStack.top();
			temp.fStack.pop();

			BTreeNode rightNode = currentNode->right;
			while (rightNode->nonEmpty())
			{
				temp.fStack.push(rightNode);
				rightNode = rightNode->left;
			}
		}

		return temp;
	}

	// Check if two iterators are equal or not
	bool isEqual(const Iterator& aOtherIter) const
	{
		return &fBSTree == &aOtherIter.fBSTree && fStack == aOtherIter.fStack;
```

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

	// Return an iterator pointing to the beginning of the tree
	Iterator begin() const
	{
		Iterator temp = *this;
		temp.fStack = BTNStack();
		temp.pushLeft(temp.fBSTree.fRoot);
		return temp;
	}

	// Return an iterator pointing to the end of the tree
	Iterator end() const
	{
		Iterator temp = *this;
		temp.fStack = BTNStack();
		return temp;
	}
};
```