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: Sunday, November 19, 2023, 23:59 (VN Time)

Lecturer: Dr. Van Dai PHAM

Your name: Le Minh Kha **Your student id:** 104179506

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Thursday 10:00 (Innovation Lab)
							x

Marker's comments:

Problem	Marks	Obtained
1	94	94
2	42	36
3	8+86=94	94
Total	230	224

Extension certification:

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

BinaryTreeNode.h

```
// COS30008, Problem Set 4, Problem 1, 2022
#pragma once
#include <stdexcept>
#include <algorithm>
#include<iostream>
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()), right(&NIL), left(&NIL) {};
    // copy constructor
    BinaryTreeNode(const T& aKey) :key(aKey), left(&NIL), right(&NIL) { std::cout
<< "copied"<<std::endl; }

    // move constructor
    BinaryTreeNode(T&& aKey) :key(std::move(aKey)), left(&NIL), right(&NIL)
{ std::cout << "moved" << std::endl; }

    //destructor
    ~BinaryTreeNode()
    {
        if (!left->empty()) delete left;
        if (!right->empty()) delete right;
    }

    bool empty() const
    {
        return this == &NIL;
    }

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

    }

    size_t height() const
    {
        if (empty()) throw std::domain_error("Empty Tree encountered");

        // Tree height == maximum of the heights of its (left and right
subtrees) + 1.
```

```
            return std::max(left != &NIL ? left->height() + 1 : 0, right != &NIL ?
right->height() + 1 : 0);
      }

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

            // check duplication
            if (aKey == key) return false;

            //compare, inserts the key into the left subtree if less than the
current key && vice versa
            BinaryTreeNode*& node = aKey < key ? left : right;
            // if nothing to compare, create new node with that key
            if (node->empty()) node = new BNode(aKey);
            else return node->insert(aKey);

            return true;
      }


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

BinaryTreeSearch.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()
    {
        if (!fRoot->empty()) delete fRoot;
```

```
    }

    bool empty() const
    {                                   // -6: fRoot == &BNode::NIL
        return fRoot->empty();
    }

    size_t height() const
    {
        // invoke the #height# method in Node class
        return empty() ? throw domain_error("No height") : fRoot->height();
    }

    bool insert(const T& aKey)
    {
        // same as the method in Node but we dont need to compare which side to
insert
        // invoke the #insert# method we created on Node class on the root node.
        return empty() ? (fRoot = new BNode(aKey), true) : fRoot->insert(aKey);
    }
    bool remove(const T& aKey)
    {
        // case empty && root is leaf node
        return empty() ? throw domain_error("Cannot move empty tree") : fRoot-
>leaf() ?

        // compare the key, if true -> invoke #remove# onto fRoot
        (fRoot->key != aKey ? false : (fRoot = &BNode::NIL, true)) : fRoot-
>remove(aKey, &BNode::NIL);
    }



    using Iterator = BinarySearchTreeIterator<T>;


    friend class BinarySearchTreeIterator<T>;

    Iterator begin() const { return Iterator(*this).begin(); }
    Iterator end() const { return Iterator(*this).end(); }
};
```

BinaryTreeSearchIterator.h

```
#pragma once

#include "BinaryTreeSearch.h"

#include <stack>
#include <queue>


template<typename T>
class BinarySearchTreeIterator
{
```

```
private:

    // I decided to implement BFS as well cause I am super excited about this topic
    // I even have a git repo about using BFS and DFS to solve eight puzzle
    //  https://github.com/SimplyLMK/BFS-and-DFS-in-eight-puzzle.git
    // BFS and DFS has same logic, queue structure makes all the difference.
    // in myBFS implementation I just changed the code to use a queue struct
    using BSTree = BinarySearchTree<T>;
    using BNode = BinaryTreeNode<T>;
    using BTreeNode = BNode*;
    using BTNStack = std::stack<BTreeNode>;
    // the structure of BFS is a queue if you think about it..
    using BTNQueue = std::queue<BTreeNode>;

    const BSTree& fBSTree;
    BTNStack fStack;
    BTNQueue fQueue;

    // say we have a tree structure with R as Root, its children is A and D
    // children of A is B and C. A is left-est node.


    // >| B | // R is init as root, and pushLeft(R) to the stack
    //  | A | // A is R's left-est node so pushLeft into stack
    //  | R | // B is A's left-est node, pushLeft B which is a leaf node

    //  |   | // operator* to print out B
    // >| A | // operator++ move iterator to (A), pop B and pushLeft nothing
    //  | R | // oper* to print A


    //  |   | // oper++ move iter to (C), pop A and pushLeft(C) leaf on stack
    // >| C | // * print out C
    //  | R | // ++ move iter to (R) pop C, pushleft nothing onto Stack

    //  |   |// * print R
    //  |   |// ++ pop C, pushLeft the right node D onto the Stack
    // >| R |

    //  |   | // when Stack is finally empty
    //  |   | // operator== return true when compare with Iterator end()
    //  |   | // indicate complete -> output:  B, A, C, R




    // push the node into the stack, ready to be pop
    // in this implemntation, we are going from left to right
    // but the right to left is valid as well
    void pushLeft(BTreeNode aNode)
    {
        if (!aNode->empty())
        {
            fStack.push(aNode);
            pushLeft(aNode->left);
        }
    }
```

```
public:

    using Iterator = BinarySearchTreeIterator<T>;

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

    // Dereference operator
    const T& operator*() const
    {
        // return key of the top node in the stack
        return fStack.top()->key;

    }

    //Moves the iterator to the next node in the tree
    Iterator& operator++()
    {

        BTreeNode lPopped = fStack.top();
        fStack.pop(); // pops the node at the top
        pushLeft(lPopped->right); //push that node into right subtree to the stack
        return *this;
    }
    Iterator operator++(int) // return copy of ite before incremented
    {
        Iterator temp = *this;
        ++(*this);
        return temp;
    }

    // if equal -> iterating in the same tree && pointing to same node.
    bool operator==(const Iterator& aOtherIter) const { return &fBSTree ==
&aOtherIter.fBSTree && fStack == aOtherIter.fStack; }

    // opposite of ==
    bool operator!=(const Iterator& aOtherIter) const { return !(*this ==
aOtherIter); }

    Iterator begin() const
    {
        // Returns an iterator pointing to the leftmost node of the tree.

        Iterator temp = *this;
        temp.fStack = BTNStack();
        temp.pushLeft(temp.fBSTree.fRoot);
        return temp;
    }
    Iterator end() const
    {
        // Returns an iterator pointing to the NIL node.
        Iterator temp = *this;
        temp.fStack = BTNStack();
        return temp;
    }
```

```
    // BFS implementation, how it works is completely the same logic

    /*
    void enqueueLeft(BTreeNode aNode)
    {
        if (!aNode->empty())
        {
            fQueue.push(aNode);
            enqueueLeft(aNode->left);
            enqueueLeft(aNode->right);
        }
    }

    const T& operator*() const
   {
       return fQueue.front()->key;
   }
   Iterator& operator++()
   {
       fQueue.pop();
       return *this;
   }
   Iterator operator++(int)
   {
       Iterator temp = *this;
       ++(*this);
       return temp;
   }
   bool operator==(const Iterator& aOtherIter) const { return &fBSTree ==
&aOtherIter.fBSTree && fQueue == aOtherIter.fQueue; }

   bool operator!=(const Iterator& aOtherIter) const { return !(*this ==
aOtherIter); }

   Iterator begin() const
   {
       Iterator temp = *this;
       temp.fQueue = BTNQueue();
       temp.enqueueLeft(temp.fBSTree.fRoot);
       return temp;
   }
   Iterator end() const
   {
       Iterator temp = *this;
       temp.fQueue = BTNQueue();
       return temp;
   }
    */

};
```

Console output:

Microsoft Visual Studio Debug Console

```
Test BinaryTreeNode:
lRoot is NIL; insert failed successfully.
Determining height of NIL.
Successfuly caught domain error: Empty Tree encountered
Insert of 25 as root.
moved
Successfully applied move constructor.
copied
Insert of 10 succeeded.
copied
Insert of 15 succeeded.
copied
Insert of 37 succeeded.
Insert of 10 failed (duplicate key).
copied
Insert of 30 succeeded.
copied
Insert of 65 succeeded.
Height of tree: 2
Delete binary tree
Test BinaryTreeNode completed.


Test Binary Search Tree:
Error: No height
copied
insert of 25 succeeded.
copied
insert of 10 succeeded.
copied
insert of 15 succeeded.
copied
insert of 37 succeeded.
insert of 10 failed.
copied
insert of 30 succeeded.
copied
insert of 65 succeeded.
Height of tree: 2
Delete binary search tree now.
remove of 25 succeeded.
remove of 10 succeeded.
remove of 15 succeeded.
remove of 37 succeeded.
remove of 10 failed.
remove of 30 succeeded.
remove of 65 succeeded.
Test Binary Search Tree completed.


Test Binary Search Tree Iterator DFS:
copied
copied
copied
copied
copied
copied
copied
DFS: 8 10 15 25 30 37 65
Test Binary Search Tree Iterator DFS completed.
```