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: Luu Tuan Hoang **Your student id:** 104180391

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:_____

Selected files

3 printable files

BinarySearchTree.h
BinarySearchTreeIterator.h
BinaryTreeNode.h

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()
    {
        if (fRoot && !fRoot->empty())
        {
            delete fRoot;
        }
    }

    [[nodiscard]] bool empty() const
    {
        return fRoot->empty();
    }

    [[nodiscard]] size_t height() const
    {
        if (empty())
        {
            throw std::domain_error("Empty tree has no height.");
        }
        else
        {
            return fRoot->height();
        }
    }

    bool insert(const T& aKey)
    {
        if (empty())
        {
            fRoot = new BNode(aKey);
            return true;
```

```
        }
        else {
            return fRoot->insert(aKey);
        }
    }

    bool remove(const T& aKey)
    {
        if (empty()) {
            throw std::domain_error("Cannot remove in an empty tree.");
        }
        else if (fRoot->leaf() && fRoot->key == aKey) {
            fRoot = &BNode::NIL;
            return true;
        }
        else {
            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
    {
        Iterator iterator(*this);
        return iterator.begin();
    }

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

BinarySearchTreeIterator.h

```
#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)
    {
        if (!aNode->empty())
        {
```

```
            fStack.push(aNode);
            pushLeft(aNode->left);
        }
    }

public:

    using Iterator = BinarySearchTreeIterator<T>;

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

    const T& operator*() const
    {
        return fStack.top()->key;
    }

    Iterator& operator++()
    {
        while (!fStack.empty()) {
            BTreeNode lPopped = fStack.top();
            fStack.pop();
            pushLeft(lPopped->right);

            if (!fStack.empty()) {
                break;
            }
        }

        return *this;
    }


    Iterator operator++(int)
    {
        Iterator temp = std::exchange(*this, Iterator(*this));
        return temp;
    }

    bool operator==(const Iterator& aOtherIter) const
    {
        return (&fBSTree == &aOtherIter.fBSTree) && (fStack == aOtherIter.fStack);
    }

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

    Iterator begin() const
    {
        Iterator temp(*this);
        {
            temp.fStack = BTNStack();
            temp.pushLeft(temp.fBSTree.fRoot);
        }
        return temp;
```

```
    }

    Iterator end() const
    {
        Iterator temp(*this);

        if (!temp.fBSTree.empty())
        {
            temp.fStack = BTNStack();
        }
        return temp;
    }

};
```

BinaryTreeNode.h

```
#pragma once
#include <stdexcept>
#include <algorithm>
using namespace std;
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");
        }
        else {
            return (right->empty()) ? key : right->findMax();
        }
    }

    const T& findMin() const
    {
        if (empty())
        {
            throw std::domain_error("Empty tree encountered");
        }
        else {
            return (left->empty()) ? key : left->findMin();
        }
    }


    bool remove(const T& aKey, BTreeNode aParent)
    {
        BTreeNode current = this;
```

```
43          BTreeNode parent = aParent;
44
45          // Search for the node to be removed
46          while (!current->empty() && aKey != current->key)
47          {
48              parent = current;
49              current = (aKey < current->key) ? current->left : current->right;
50          }
51
52          // If the node is not found
53          if (current->empty())
54          {
55              return false;
56          }
57
58          if (!current->left->empty())
59          {
60              T replacementKey = std::move(current->left->findMax());
61              current->key = std::move(replacementKey);
62              current->left->remove(current->key, current);
63          }
64          else if (!current->right->empty())
65          {
66              T replacementKey = std::move(current->right->findMin());
67              current->key = std::move(replacementKey);
68              current->right->remove(current->key, current);
69          }
70          else // Node has no children
71          {
72              if (parent != &NIL)
73              {
74                  (parent->left == current) ? parent->left = &NIL : parent->right =
    &NIL;
75              }
76              delete current;
77          }
78
79          return true;
80      }
81
82      BinaryTreeNode(): key(T()), left(&NIL), right(&NIL){}
83
84      explicit BinaryTreeNode(const T& aKey): key(aKey), left(&NIL), right(&NIL){}
85
86      explicit BinaryTreeNode(T&& aKey): key(std::move(aKey)), left(&NIL), right(&
    NIL){}
87
88      ~BinaryTreeNode()
89      {
90          if (left && !left->empty())
91          {
92              delete left;
93          }
94
95          if (right && !right->empty())
96          {
97              delete right;
98          }
99      }
100
```

```
101     [[nodiscard]] bool empty() const {
102         return this == &NIL;
103     }
104 
105     [[nodiscard]] bool leaf() const {
106         return (left == &NIL) && (right == &NIL);
107     }
108 
109     [[nodiscard]] size_t height() const
110     {
111         if (empty())
112         {
113             throw std::domain_error("Empty tree encountered");
114         }
115         else if (leaf()) {
116             return 0;
117         }
118         else {
119             return 1 + std::max(left->empty() ? 0 : left->height(), right->empty()
    ? 0 : right->height());
120         }
121     }
122 
123     bool insert(const T& aKey) {
124         if (aKey == key || empty())
125             return false;
126         else if (aKey < key) {
127             if (!left->empty())
128                 return left->insert(aKey);
129             left = new BNode(aKey);
130             return true;
131         }
132         else {
133             if (!right->empty())
134                 return right->insert(aKey);
135             right = new BNode(aKey);
136             return true;
137         }
138     }
139 
140 };
141 template<typename T>
142 BinaryTreeNode<T> BinaryTreeNode<T>::NIL;
```