# 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:**______________________________ **Your student id:**_________________

| 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 Set 4: Binary Search Trees & In-Order Traversal

## Problem 1

Implement template class `BinaryTreeNode` that defines a basic representation for binary search trees:

```
#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;
  const T& findMin() const;

  bool remove( const T& aKey, BNodeTree aParent );

  BinaryTreeNode();
  BinaryTreeNode( const T& aKey );
  BinaryTreeNode( T&& aKey );

  ~ BinaryTreeNode();

  bool empty() const;
  bool leaf() const;
  size_t height() const;

  bool insert( const T1& aKey );
};

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

Somebody has already started the project and implemented the `remove()` method and its two dependent functions `findMax()` and `findMin()`. You need to define the remaining features, including `insert()`.

The functions `leaf()` and `height()` define core tree primitives. The method `insert()` adds a node at the correct place in the binary tree structure, if this is possible.

Special care is required for the implementation of method `height()`. The empty tree has no height, hence `height()` must throw a `domain_error` exception in this case. For interior nodes, however, if a subtree of a given `BinaryTreeNode` is empty, then this subtree contributes a height of zero to the interior node. In other words, the smallest height of any non-empty `BinaryTreeNode` is zero. Remember, interior nodes add one to the maximum height of their subtrees.

The constructors and the destructor create and destroy nodes, respectively.

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver, which should produce the following output:

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

The test driver also implements **`operator<`**`()` for strings. This operator is required in `insert()` when we use strings as keys.

## Problem 2

Implement template class `BinarySearchTree` that defines the basic infrastructure for a binary search tree (we ignore proper copy semantics here):

```
#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();

  ~BinarySearchTree();

  bool empty() const;
  size_t height() const;

  bool insert( const T& aKey );
  bool remove( const T& aKey );

  // Problem 3 methods

  using Iterator = BinarySearchTreeIterator<T>;

  // Allow iterator to access private member variables
  friend class BinarySearchTreeIterator<T>;

  Iterator begin() const;
  Iterator end() const;
};
```

The template class `BinarySearchTree` defines an object adapter for binary tree nodes. The result is a simple binary search tree abstraction that provides support for adding and deleting nodes and a systematic traversal of the tree via a corresponding iterator.

The iterator `BinarySearchTreeIterator` is given as a forward declaration. The actual implementation is part of Problem 3. For `BinarySearchTree` you just have to add implementations, in `begin()` and `end()`, that yield a corresponding iterator. Compare with tutorial 9 and problem set 3, in which we used a similar approach to connect the container types their respective iterators. The test driver for this problem does not use iterators. Hence, the C++ compiler will not report any problems other than syntax errors.

The definition of `BinarySearchTree` contains a friend declaration. This allows instances of `BinarySearchTreeIterator` to access the private instance variables of objects of type `BinarySearchTree`. This is a design choice that avoids exposing the internal representation of `BinarySearchTree` to clients. In C++, we can use the friend mechanism. In other languages, like Java and C#, we can achieve the same by assigning package visibility to the member `fRoot` of class `BinarySearchTree`.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver, which should produce the following output:

```
Test Binary Search Tree:
Error: Empty tree has no height.
insert of 25 succeeded.
insert of 10 succeeded.
insert of 15 succeeded.
insert of 37 succeeded.
insert of 10 failed.
insert of 30 succeeded.
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.
```

## Problem 3

Implement a binary search tree iterator that performs in-order traversal. Iterators do not allow for a recursive search procedure. Instead we need to use a stack to record the nodes that need to be visited. The required process follows the depth-first search pattern, which first descents from the root along the left nodes to a leaf. Once the leftmost node has been visited, the iterator removes it from the stack, inspects its right node and descents, if necessary, to the leftmost node along the left nodes to a leaf again. The iterator stops if the traversal stack is empty.

A corresponding iterator is given by the following specification:

```
#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 );

public:

  using Iterator = BinarySearchTreeIterator<T>;

  BinarySearchTreeIterator( const BSTree& aBSTree );

  const T& operator*() const;
  Iterator& operator++();
  Iterator operator++(int);
  bool operator==( const Iterator& aOtherIter ) const;
  bool operator!=( const Iterator& aOtherIter ) const;

  Iterator begin() const;
  Iterator end() const;
 };
```

The class template `BinarySearchTreeIterator` implements a forward iterator. It uses `std::stack` as traversal stack. The template class `std::stack` implements the standard behavior of a stack. Please note that this iterator just maintains a reference to the binary search tree.

To streamline the implementation of `BinarySearchTreeIterator`, method `pushLeft()` performs the tree traversal along the left nodes. That is, while `aNode` is not empty, `pushLeft()` pushes it onto the traversal stack and sets `aNode` to its left node.

The methods `begin()` and `end()` have to return copies of the current iterator. Both methods require an update of the traversal stack. Avoid loops that pop elements. A simpler way is to use an assignment with the default (constructed) value of the traversal stack. That is, we can reset the traversal stack by assigning it an empty traversal stack.

You can use `#define P3` in `Main.cpp` to enable the corresponding test driver, which should produce the following output:

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

**Submission deadline: Thursday, May 26, 2023, 14:30.**

**Submission procedure:** PDF of printed code for `BinaryTreeNode`, `BinarySearchTree`, and `BinarySearchTreeIterator`.