Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	12, Binary Tree Traversal
Lecturer:	Dr. Markus Lumpe

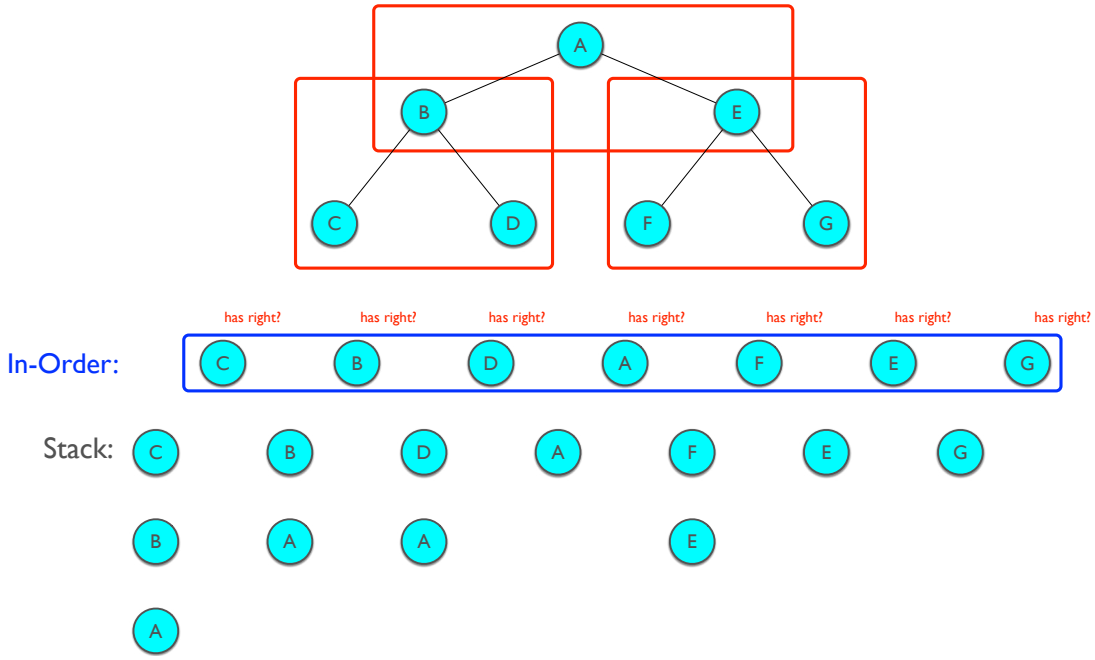

Figure 1: In-Order Binary Tree Traversal.

Binary Tree Traversal

Consider the template class `BTree` that we defined in tutorial 11. The aim of this tutorial is to define an iterator for binary trees that performs in-order traversal.

In order to perform in-order traversal, a suitable iterator implementation has to employ a traversal stack to control the sequence in which nodes are visited. In case of in-order traversal, we start with the left-most, outer-most node. That is, initially we populate the traversal stack with all left nodes originating and including the root node. Once a node has been visited, we remove it from the traversal stack and, if it has a right subtree, push all left nodes of the right subtree including the root of the right subtree onto the traversal stack. In general, the behavior of such an iterator is given by the following instructions:

- Initialization:
  - Push all left nodes, starting at the root node, onto stack.
- Element access:
  - Use top node on stack.
- Move forward:
  - Hold on to top node on stack and pop.
  - If top node has right node then push it onto stack.
  - Push all left nodes, originating from top's right node, onto stack.

In addition, though not strictly necessary for in-order traversal, we use a `Frontier` data type to record the traversal progress. The frontier abstraction is required, in general, for traversal tasks that have to explore multiple paths in a specific sequence. A frontier allows the same node to be analyzed as top node and update this node's traversal state. In case of in-order traversal we merely use the frontier to access the top node's right subtree. Retaining the `Frontier` abstraction in the implementation allows for future extensions and support of other traversal methods (e.g., post-order traversal).

Following the lecture, a suggested specification of a binary tree iterator supporting in-order traversal is given as follows:

```
#pragma once

#include "BTree.h"

#include <stack>
#include <memory>

template<typename T>
class DepthFirstTraversal
{
public:
  using BTree_ptr = const BTree<T>*;
  using Iterator = DepthFirstTraversal<T>;
  using Node = typename BTree<T>::Node;

  struct Frontier
  {
    bool mustExploreRight;
    BTree_ptr node;

    Frontier( BTree_ptr aNode = nullptr ) :
      mustExploreRight(true),
      node(aNode)
    {}

    bool operator==( const Frontier& aOther ) const noexcept
    {
      return
        mustExploreRight == aOther.mustExploreRight &&
        node == aOther.node;
    }
  };

  DepthFirstTraversal( const Node& aBtree );

  const T& operator*() const noexcept;

  Iterator& operator++();

  Iterator operator++(int)
  {
    Iterator old = *this;

    ++(*this);

    return old;
  }

  bool operator==( const Iterator& aOther ) const noexcept;

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

  Iterator begin() const noexcept;
  Iterator end() const noexcept;

private:
  BTree_ptr fRoot;
  std::stack<Frontier> fStack;

  void pushNode( BTree_ptr aNode ) noexcept;
};
```

Complete the implementation of the iterator.

The test driver in `Main.cpp` should produce the following output:

```
In-Order Traversal: 10 15 25 30 37 65
```

Can you explain the result? Does the output confirm to in-order traversal?