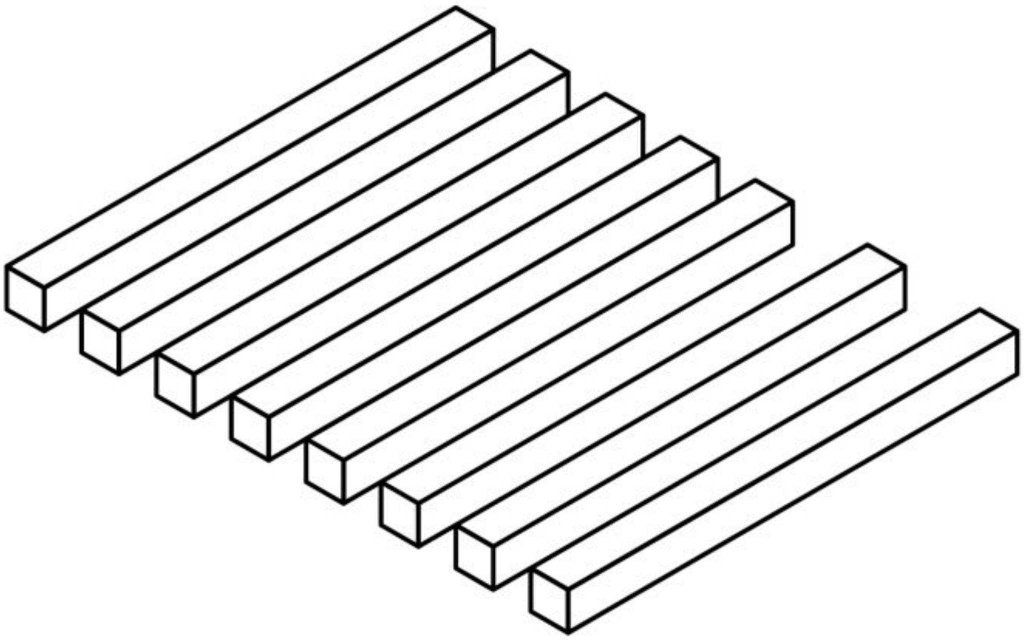
Swinburne University of Technology

School of Science, Computing and Emerging Technologies

LABORATORY COVER SHEET

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

If you think it's simple, then you have misunderstood the problem.
Bjarne Stroustrup



Problem 1

We have studied the construction of general n-ary trees in the lecture last week. This tutorial aims to define a template class `BTree` that implements the basic infrastructure of binary trees, including full copy control and recursive tree traversal. Some details may differ from those presented in the lecture.

The lecture material covers most of the implementation details. However, some of the features (i.e., methods) defined for the template class `NTree` need adjustment to achieve a suitable binary tree implementation. Merely creating a type alias for `NTree`, where `N equals 2`, is insufficient as it does not provide an abstraction that is conceptually close enough to the hierarchical data structure of a binary tree.

```
#pragma once

#include <cstddef>
#include <memory>
#include <cassert>
#include <iostream>

#include "Visitors.h"

template<typename T>
class BTree
{
public:
  using node = std::unique_ptr<BTree>;

  BTree( const T& aKey = T{} ) noexcept;
  BTree( T&& aKey ) noexcept;

  ~BTree();

  template<typename... Args>
  static node makeNode( Args&&... args );

  BTree( const BTree& aOther );
  BTree& operator=( const BTree& aOther );

  BTree( BTree&& aOther ) noexcept;
  BTree& operator=( BTree&& aOther ) noexcept;
  void swap( BTree& aOther ) noexcept;

  const T& operator*() const noexcept;

  bool hasLeft() const noexcept;
  BTree& left() const noexcept;

  bool hasRight() const noexcept;
  BTree& right() const noexcept;

  void attachLeft( node& aNode );
  void attachRight( node& aNode );

  node detachLeft();
  node detachRight();

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

  const T& findMax() const noexcept;
  const T& findMin() const noexcept;

  void doDepthFirstSearch( const TreeVisitor<T>& aVisitor ) const noexcept;

private:

  T fKey;
  node fLeft;
  node fRight;
};
```

Implement the template class `BTree`.

Some elements have already been defined. The `doDepthFirstSearch()` method implements depth-first search. Unlike the approach shown in the lecture, if a given tree lacks a left or right subtree, the `doDepthFirstSearch()` method calls `emitEmty()` on the visitor. The default implementation is empty, but a subclass can override this behavior.

For the methods `findMin()` and `findMax()` find an iterative solution. These methods were presented in the lecture in a tail-recursive form. Every tail recursion can be converted to a while loop. The solution requires the use of raw pointers to constant `BTree` objects.

The class `BTree` defines a destructor. This is not strictly necessary. However, in this case, we use it to print a notification that a given `BTree` node is about to be destroyed.

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

```
Test BTree functionality.
Height of n25: 2
Min of n25: 10
Max of n25: 65
Max of left of n25: 15
Min of right of n25: 30
Pre-Order Traversal: 25 10 15 37 30 65
In-Order Traversal: 10 15 25 30 37 65
Post-Order Traversal: 15 10 30 65 37 25
Test BTree functionality complete.
Deleting 25
Deleting 37
Deleting 65
Deleting 30
Deleting 10
Deleting 15
```

Problem 2

Consider the following template class TreeDecorator:

```
#pragma once

#include "Visitors.h"

template<typename T>
class TreeDecorator : public TreeVisitor<T>
{
public:

  void emitEmpty() const noexcept override;

  // override pre-order behavior
  void preVisit( const T& aKey ) const noexcept override;

  // override post-order behavior
  void postVisit( const T& aKey ) const noexcept override;
};
```

The template class TreeDecorator is a visitor used for depth-first search traversal. A TreeDecorator visitor overrides the emitEmpty() function to create a printable representation of an empty tree. Most importantly, a TreeDecorator visitor performs both pre-order and post-order traversal. The preVisit() method generates the opening sequence for a linear tree representation, while the postVisit() method produces the closing sequence. The TreeDecorator visitor is an abstraction that generates the fully parenthesized version of an expression, specifically a linear representation of a binary tree.

The solution may produce a leading space.

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

```
Test TreeDecorator functionality.
 { 25 { 10 {} { 15 {} {} } } { 37 { 30 {} {} } { 65 {} {} } } }
Test TreeDecorator functionality complete.
Deleting 25
Deleting 37
Deleting 65
Deleting 30
Deleting 10
Deleting 15
```