Lab 6 - Building a BST

Due Date: 5:00 p.m., March 23, 2015
All function interfaces are suggested naming and parameter guidelines. If you feel there is a better way, you are free to alter names, functions interfaces, etc, as long as you follow the lab and style guidelines. Output should match exactly unless otherwise stated.

The goal of Lab 6 is to start building a BST.

(Parts A must be completed in lab)

Part A: Creating a BST Class

- Write your own simple Binary Search Tree (BST) C++ class that includes the insert(), find(). We'll save remove for later. You BST must have the following public interface:
    - BSTree()
    - bool empty()
        - true if the tree is empty
        - false if it is not
    - bool insert(int val)
        - Returns true if the value was inserted
        - false if the value was already in the tree
    - bool find(int val)
        - true if the value is in the tree
        - false if the value is not in the tree
- Use one class for the entire BST, which contains a single pointer to a Node, which is the root of the tree.
- You'll also have a helper C++ class, Node, which contains an integer data member to store the data item, and three pointers to other Nodes, one for the left subtree, one for the right subtree, and one for the parent.
    - You are going to create the Node class as an internal class to the Tree. This MUST be the first thing in your BSTree class. You declare it just like a normal class, except it is kept as a private data member:

        -
          ```
          class BSTree{
                  private:
                          class Node{
                                  public:
                                          …
          ```
    - You can only access the node class within you BSTree class. To access it in external methods you can use the scope operator. ex. BSTree::Node
        - More info about nested classes

- You must complete your header file and empty implementations of your insert and find in lab. Make sure everything compiles before moving on.
    - Example:

      ```
      bool insert(int){
              return true;
      }
      ```

--END OF IN LAB REQUIRED WORK--

Part B: Implementing insert and find

- INSERT: To insert a new node, start at the root and move down left or right, following the appropriate pointers, until you get to the appropriate place at the bottom. Then create a new BSTree::Node and set the appropriate pointer to the new node.
- FIND: Find works the same as insert, expect you do not create a new node. Instead you return a boolean value when you find the value or a null branch.

Part C : Code Organization and Submission

- Required code organization:
    - lab6.cpp
    - BSTree.cpp/.h
    - makefile
        - executable should be called: lab6
            - *do not add a .exe extension*
    - readme
- Create a readme file following the format provided, and add it to your project folder
- While inside your lab 6 folder, create a zip archive with the following command
    - zip -r lab6 *
        - This creates an archive of all file and folders in the current directory called lab6.zip
        - **Do not zip the folder itself, only the files required for the lab**
- Upload the archive to Mimir under 'Lab 6'

Expected Interface and Test Output

- **Driver Test Commands**
    - Use the following driver code to test your BST
        - lab6.cpp

Grading Guidelines

- **Part A (3 points)**
  - Header and Implementation split into .h and .cpp (1 point)
  - Node is an internal class (2 points)
- **Part B (6 points)**
  - 1 point for each test
- **Part C (1 point)**
  - Follows formatting guidelines, requested project structure and naming conventions, contains written Readme, and submission does not include .o files or binary

Formatting Guidelines

- Stores all values in a named variable.
  - No Magic Numbers.
- Uses indentation to identify code blocks.
  - Every Code block should be indented from it's parent block to identify scope.
- No single letter or non-descriptive variable names
  - The only exception to this rule is 'i' in a for loop
- Separates code blocks and logical sections with whitespace
  - Optimize your code for the reader, not the writer
- Output is formatted with an explanation of the output values
  - Format your output so that someone who does not know what the program is supposed to do would know what the output meant
- Each method is preceded by a comment explaining what the method does
- Each significant code block is preceded by a comment explaining what the code block does.
  - A significant code block is more than 3 lines performing a single logical operation
- CONSTANTS are in all caps
- Only data types start with a capital letter
  - Classes, Enums, Structs, etc.
- Do not use the 'using namespace' declaration in a header (.h) file
- In general we will follow the Google C++ style guidelines. If you want more info, you can view them here: https://google.github.io/styleguide/cppguide.html