

Data Structures and Algorithms

**Lab 7: BST.h**

**The Scenario**

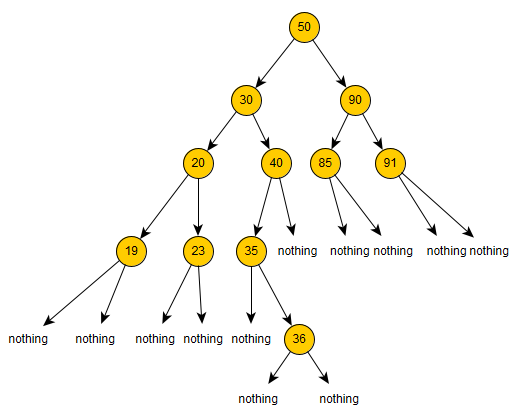
Throughout your time in this class, you’ve had many different roles. You’ve been a painter, a dragon slayer, and a successful programmer. You drive to your favorite mountain peak to reflect upon your time and look out across the land. You’ve searched for answers. If only there was a simple way to find the answers to all of life’s questions. And better yet, a way of indexing all of those answers. Something that makes the search not quite so lengthy. What better way than a **Binary Search Tree**?

A **Binary Search Tree (BST)** is, quite simply put, a **Binary Tree** that is also a **Search Tree**. To begin, let’s start with what a Binary Tree is.

A **Binary Tree** is the same as a Linked List with one major difference: each node has two subsequent nodes instead of one. In fact, if you should choose, you could call a Linked List a Unary Tree and that would be an accurate name. Having two subsequent nodes instead of one creates a sprawling structure that is similar in shape to a tree when drawn out. See the illustration later in this lab.

A **Search Tree** data structure is any Tree that is organized in a predictable way. In the illustration below, the BST is organized such that lower numbers are always on the left and larger are always on the right. This makes it incredibly easy to find values in the tree. Together, a Binary Tree and a Search Tree form a Binary Search Tree—i.e. a Binary Tree that is organized to make finding values much easier and use less processing power.

Below is an illustration of a Binary Search Tree:



As was mentioned, each node’s data determines its placement within the tree. Nodes to the left of the current node are all lower values than the current node while nodes to the right are all higher values. Further than this, no node within the entire left subtree of a current node will be greater than it’s value. This holds true for the right and being greater as well. Using this logic, try to find your way to the value “23” from the root. You will go left from 50 because 23 is lower than 50. The current node is now 30. 23 is less still so you will go left again to find yourself at a node with the value of 20. 23 is bigger than 20, so you will go right to the node with a value of 23. Hey, you found it! As you can see, this is much more efficient than a linked list or unorganized tree would be, where you would potentially have to examine every node.

Also worth noting, and the reason I included node 36, is that a BST may have nodes that don’t make it a “balanced” tree. Nodes may exist at the end of any arrow.

**What To Do…**

Open BST.h. There will be instructions written in the comments on what is expected. Below is the gist of each function and variable…

***Variables:***

**mRoot** The root of the tree. The “Starting Point”, so to speak.

**data** The data stored in each node.

**left** The node to the left. (i.e. a node of lesser value.)

**right** The node to the right. (i.e. a node of greater value.)

***Functions:***

**Node Constructor** Takes a data parameter and sets the node’s data equal to it. Left and Right are initialized appropriately as well.

**BST Constructor** Creates an empty tree. (i.e. a tree with absolutely no nodes.) Initializes variable(s) where needed.

**BST Destructor** Clears out all dynamic memory.

**BST Copy Constructor** Makes a deep copy of the tree passed in with the invoking object.

**Assignment Operator** Sets the invoking object equal to the object passed in. (Deep copy.)

**Copy (helper)** Should you choose to do your Assignment Operator and/or Copy Constructor recursively, this method will be responsible for the recursive calls to itself.

**Clear** Takes no parameters. Should you do this one iteratively, you won’t need the helper method. If you do it recursively, it will call the Clear helper function as well as do any tidying up that needs done afterwards. The resulting tree should be identical to one that has been created with the default constructor.

**Clear (helper)** Takes one parameter. Calls itself over and over to clear all nodes.

**Push** Takes one parameter: The value to add into a node and store somewhere in the tree. If you do it recursively, you’ll use the helper function below. Remember, it’s a search tree—you must traverse the tree based on the value passed in.

**Push (helper)** Takes two parameters. The value to add and the current node in the tree’s traversal. When the correct place is found, adds the node.

**Contains** Similar to Push. Simply searches for a value in the tree and returns true if it is found and false if it is not.

**Remove** Takes one parameter—the value to remove. Finds the value and removes it. This is possibly the most complex function you will write in DSA—there are 3 cases that must be looked at. The order they are handled will affect how you write your code. You must consider whether the node-to-remove is the left or right branch of its parent node as well as whether the node-to-remove has no children, a single left child, a single right child, or two children. A rough outline/some questions to guide you in the right direction of how to go about this function is here:

**Find whether the value to remove exists in the tree. If it does, proceed. If not, return false.**

**Case 0 children**

**Is it the left or right child of the parent?**

**Imagine removing node 23 in the figure above.**

**Case 1 child (left or right)**

**Is it the left or right child of the parent?**

**How about the node to remove’s child? Is it left or right?**

**Imagine removing node 40 in the figure above.**

**Case 2 children (both left and right)**

**The tree must be fixed in this case. i.e. we can’t simply remove the node—we have to swap the next-biggest value into the node we want to remove’s place. After finding the next-biggest node, the following code will be heavily based if not the same as the case 1 or case 0 code again—i.e. Is it the left or right child of the parent? Reassign the pointers as needed to do your swap and then reassign the pointers associated with your next-biggest’s old location. (Much of this will depend on if you go the recursive route or iterative route for this function, both of which are viable.)**

**Some more food for thought--How do we find the next-biggest value? How far away is that value from the node to remove? Does the mRoot case need handling here?**

**Hint: Draw the tree out several times (or print it out several times on paper) and practice “removing” specific nodes and redrawing the arrows (pointers) for each case to visualize it. Imagine you have to remove node 30 in the graph above, for example.**

**InOrder** Does an InOrder traversal of the tree and builds a string to return. Uses to\_string to convert each node’s data to a string.

**InOrder (helper)** Optional helper for if you decide to do InOrder recursively. Accepts the current node as a parameter as well as the string we’re building (So that we can add to it, of course!)

**Tips, Tricks, and Resources**

* Functions/Data Members available in the string classe can be found on the Cplusplus.com documentation:
  + <http://www.cplusplus.com/reference/string/string/>
* Draw trees! Maybe even photocopy your tree several times to draw on! Add nodes to your drawing, remove nodes, and test your functions out by drawing each line of code as it happens if needbe. This lab is much easier if you draw the tree out when you’re stuck. Once you understand exactly what needs to happen and it makes sense, the coding part, while lengthy, will be much more manageable. This is a difficult lab to attempt blindly.
* Many functions can make use of other functions in this lab.

**Plagiarism**

Plagiarism and Academic Dishonesty are considered a **very** serious offense in this class and can have a range of consequences including suspension, and in very serious cases, expulsion. If you either share your code or copy someone else’s code, you will be given a **0** on your lab and can face further disciplinary action.

In other words, don’t cheat please!