Submission

Binary Search Trees

Introduction

Today you will be creating and implementing a Binary Search Tree given just a strict specification for each component of the class.

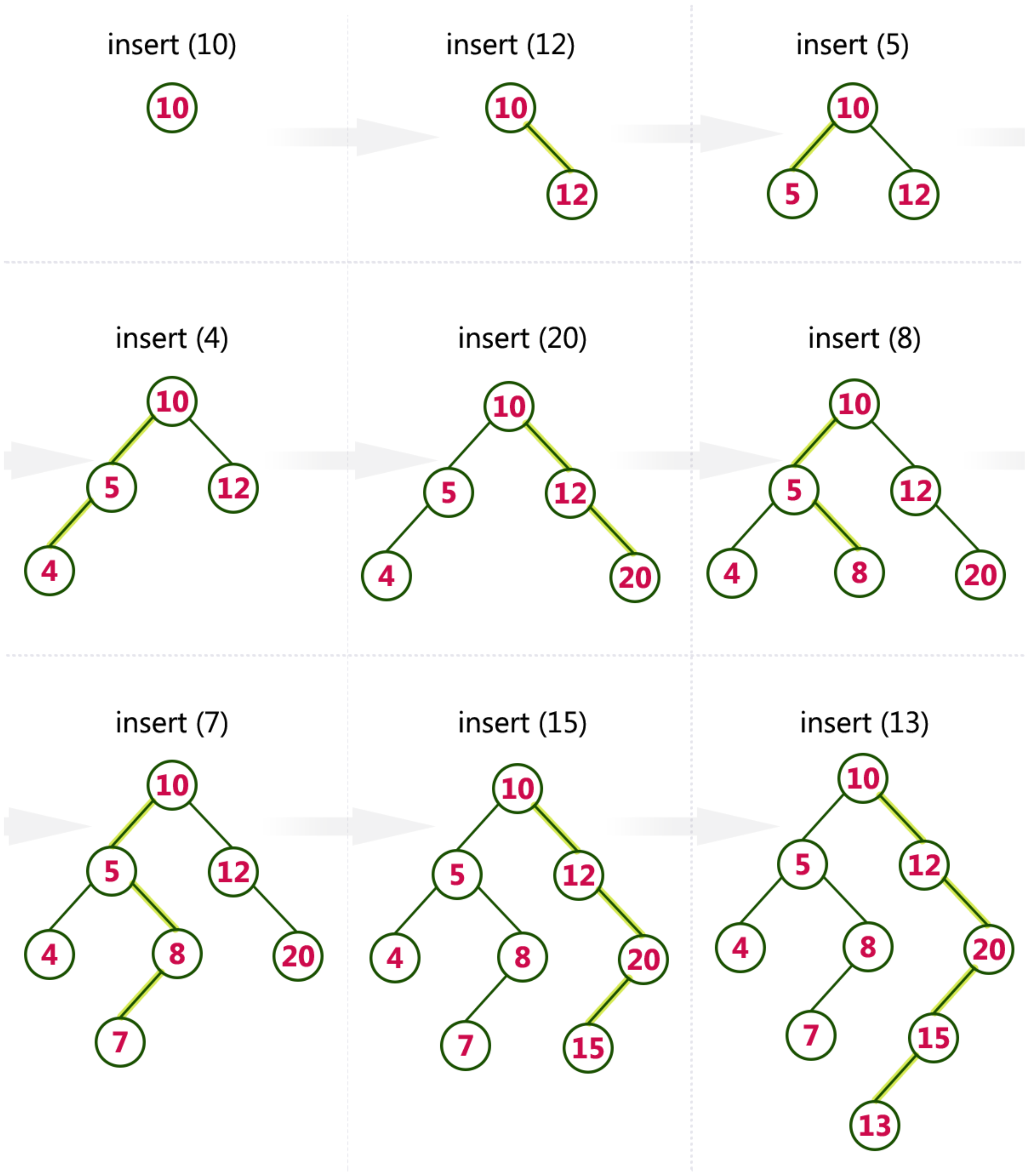
This may sound intimidating, but do not be alarmed, for as long as you take each part bit-by-bit you should have no trouble getting through at least the majority of the lab.

Binary Search Trees

A binary search tree is a data structure to store arbitrary "items" in a way that allows extremely rapid searches, insertion, and removal.

They rely on sortable "keys" which are used to store and look up a given node.

These "keys" and the way they are stored means that during a lookup, at each step, half of the current sub-tree is discarded from the search.



As you can see from the image above, a Binary Search Tree is ordered in the following way:

- A root node pointer which always points to the initial value
- Each node has a left pointer and a right pointer
    - The **left** pointer points to a node whose key is **less** than the parent node's key
    - The **right** pointer points to a node whose key is **greater** than the parent node's key

- This pattern continues throughout the BST

The resulting tree will have two subtrees stemming from the root node. The left-hand subtree will have key values which are all less than the root node's key. The right-hand subtree will have key values which are all greater than the root node's key.

Check out this [binary search tree visualization](http://btv.melezinek.cz/binary-search-tree.html)

The Starter Code

You are given `main.cpp` which contains the code to read the test files, perform the appends, perform the removes, and print the tree as required by the autograder.

Instructions

Be sure to read Sections 4.1 and 4.2 before starting

Your goal for this lab is to write a Binary Search Tree from scratch.

4.1 Class Contents

Aside from putting in the methods declarations, which will be explained in more detail below, you require the following to make your classes work properly:

In BSTNode:

- An integer for storing the node's data (private)
- A pointer to the node's left child (private)
- A pointer to the node's right child (private)
- A constructor that takes an integer to be stored in the data variable (public)
- An empty destructor (public)
- The BSTNode should also give the BSTree class permission to modify its private data members

In BSTree:

- A pointer to the root node (private)
- All of the methods specified in Section 4.2 below

4.2 Methods

Constructor and Destructor

The constructor should take no parameters and initialize the root variable with an appropriate value. In other words, it should create an empty tree.

The destructor should delete the entire tree, which is an operation best performed recursively.

However, destructors in C++ cannot be recursive.

The solution to this predicament is to use a private helper method, `destroy()`, which will be described in further detail below.

The destructor should just call `destroy()`, passing in the root node pointer.

> Note: A common theme in many of these methods is to have a public, non-recursive method which simply calls a private, recursive function on the root node.

> This is because a binary search tree can be traversed or searched easily by calling a recursive method on a node's left child, right child or both.

> In fact, due to a concept called **function overloading**, these methods can have the same name so long as they have different parameters.

It is recommended to work on these methods in order and test them before moving on to the next.

Insert

You should create two methods named `insert()`, one which takes an integer and returns nothing, and the other which takes a `BSTNode *` and an integer and returns a `BSTNode *`.

The first method is the public method, which the user will be able to call, and the second method is the private recursive method which actually inserts the node in the correct place in the tree.

Height

Again, create two methods named `height()`, one private and one public.

The public method should take no parameters, call the private method on the root, and return the resulting integer representing the height of the tree.

The private method should take a `BSTNode *` parameter and return an integer representing the height of the subtree rooted at the node passed in.

> Note: Calling the height method on an empty tree should return -1 as a tree with one node is of height 0. This is meant to distinguish between the two.

Traversals

Here you will need six methods in total:

1. Two \`preorder()\` methods: one private and one public
2. Two \`inorder()\` methods: one private and one public
3. Two \`postorder()\` methods: one private and one public

Each public method should take the line `std::ostream& os = std::cout` as a parameter and return `void`, simply calling the private counterpart with the root node pointer and the `ostream`.

Each private method should take a `BSTNode *` and a `std::ostream&` as parameters and return `void`, performing the traversal specified by its name.

The `ostream` is needed solely for testing, but you can also think of it as a generalization of printing.

Instead of writing data to STDOUT using `std::cout`, you will instead write to the `ostream` passed in as an argument.

Thus, when you visit each node, you should write its data to the `ostream` just as you would with `cout`, including the `std::endl`.

As for the parameter `std::ostream& os = std::cout`, this allows you to call the method without passing in an `ostream`, in which case the method will use `cout` by default.

Essentially, calling the method as `traversal()` will default to using `std::cout`, and outputting as `os << "anything"` will work exactly as you'd expect `std::cout << "anything"` to work.

Destroy

You should create a destroy function which is private and takes a node and returns void, the public version takes no arguments and invokes the private version.

It should destroy the entire subtree starting at a given node, recursively.

> Ensure that you only delete your current node after all subtrees have been freed.

> Think about which type of traversal would be best to implement this method.

Search

You should create a public search method which takes an integer and returns a boolean indicating whether or not the node was found.

It should search the tree recursively and return true if and only if the value is found, false otherwise.