CSC 325 Adv Data Structures

Assignment #4

Red Black Trees

Your objective is to implement the main operations of a Red Black Tree from the lecture. To be clear, this entails creating a class for your tree nodes, then creating another class for your Red Black Tree.

Create a file called `Rbt.java`. Inside this file, put the following:

Create a class called **`RbtNode`** that has the following fields:

- Fields
  - data
    - Stores the information in the node. This should be a simple int and should be private.
  - color
    - This stores the color of the node, either red or black. This should be a single byte (not int) and should be private.
  - CL_RED
    - This needs to be a class variable (not instance variable) and should be marked as a public constant. It should be equal to the number 0 and must be used to represent a color of red for the node. The datatype should match that of the color field (i.e. a byte).
    - Use this by assigning the color field to CL_RED.
  - CL_BLACK
    - Same as CL_RED, yet this is used to show the node is black. The value of this constant should be 1.
  - left
    - Pointer to the left child. Should be private.
  - right
    - Pointer to the right child. Should be private.
  - parent
    - Pointer to the parent. Should be private
    - This will help out greatly for the rotations but you have to make sure you keep this up to date for all nodes if it changes. An operation may succeed once, but if a parent pointer isn't updated that should have been, the next operation may fail.
- Methods
  - Constructor
    - Only parameter is the data to put into the node. Should set all fields to a reasonable initial value.
  - Accessors
    - All private fields should be given an accessor

    - Mutators
        - All private fields should be given a mutator
    - Helper methods
        - You might find creating methods such as `getUncle`, `getGrandParent`, and others to be very helpful. Feel free to create whatever helper methods you need.
        - A rule of thumb is that if you find yourself doing something more than once to a node (for example, finding a node's uncle), then you really should make it a public helper method on the node. This will make your logic A LOT cleaner and easier to understand and will overall make this assignment easier.

Create a class (in the same file) called **`Rbt`** that has the following:

- Fields
    - public field called `root` that points to the root of the tree. This **must** be public for the visualization to work.
    - Any other fields you need
- Public methods
    - A default constructor (i.e. a constructor that doesn't take any parameters). This will initialize `root` to null.
    - `insert(value)`
        - Takes an int as input and inserts it into the logical place in the tree (using the same rules as a normal Binary Search Tree).
        - It will then check and solve any Red Black Tree violations. This can either be done in a separate function that is called from insert or done in the insert function itself. I recommend a separate function since then you can recursively call it (remember some cases require you to recursively call the function that detects and fixes violations).
    - `search(value)`
        - This method takes an int as input and returns true if the given value is in the tree, false otherwise. You don't have to return the location in the tree, just a boolean indicating whether the value is found.
        - If root is null, you should return false (since the value could not be found)
    - `min()`
        - Returns the smallest value in the tree.
        - If the root is null, return -1
    - `max()`
        - Returns the largest value in the tree.
        - If the root is null, return -1.
    - `size()`
        - Returns the number of nodes in the tree.
        - If the root is null, return 0.

- `inorder()`
  - Performs an inorder traversal.
  - Does NOT print anything
  - Instead, this returns a string with each node value (in order) with a single space between values. You are allowed to have a single leading and/or a single trailing space if you like.
  - You must use recursion here
  - Hint: making a local variable for the string you are creating might not be a good idea. Consider making a field for this instead
  - This function does NOT take any parameters, but feel free to create another function to help with the recursion.

Duplicate Values:
Do **not** worry about handling duplicate values. You can safely assume that I will not attempt to store the same value twice in your tree, so your tree does not have to handle this situation.

Main part:
Make a separate public class called **Main** and put this into a file called Main.java. Put your entry point inside this Main class. Make sure to put **Vis.class** and **Color.class** in the **same folder** as your .java files if you plan on running your code on the command line (more on this below).

Create an object of your Rbt class (call it whatever you like). You can now call whichever operations you want on your object. This will initialize your tree by giving it a set of nodes it will contain. When you are ready to see the visualize of your tree and test it in real-time, call

```
Vis.showTree(nameOfYourTreeObject);
```

Here is an example (this code would appear inside your main function in your Main class):

```
// instantiate and initialize our Rbt object
Rbt tree = new Rbt();

// load it up with some initial values
tree.insert(5);
tree.insert(15);
tree.insert(2);

// test it out
while (true) {
    Vis.showTree(tree);
    if (!Vis.showMenu(tree)) {
        break;
    }
}
```

This will allow you to take your Rbt object and show it on the screen (in beautiful ASCII art fashion). It will also display a menu wherein you can test out any operation you want. All the operations you must implement are able to be tested here. Typing quit() will exit the application (i.e. this means `Vis.showMenu(tree)` will return false and the loop above will exit.)

For submission:
Submit all .java files you have written by putting them into one zip file and submitting the zip. Do not submit any .class files.

Rubric:

#	ITEM	POINTS
1	insert method	15
2	search method	10
3	min method	5
4	max method	5
5	size method	3
6	inorder method	10
7	uses Vis correctly for visualization	2
	TOTAL	**50**

#	PENALTIES	POINTS
1	Doesn't compile	-50%
2	Doesn't execute once compiled (i.e. it crashes)	-25%
3	Late up to 1 day	-25%
4	Late up to 2 days	-50%
5	Late after 2 days	-100%