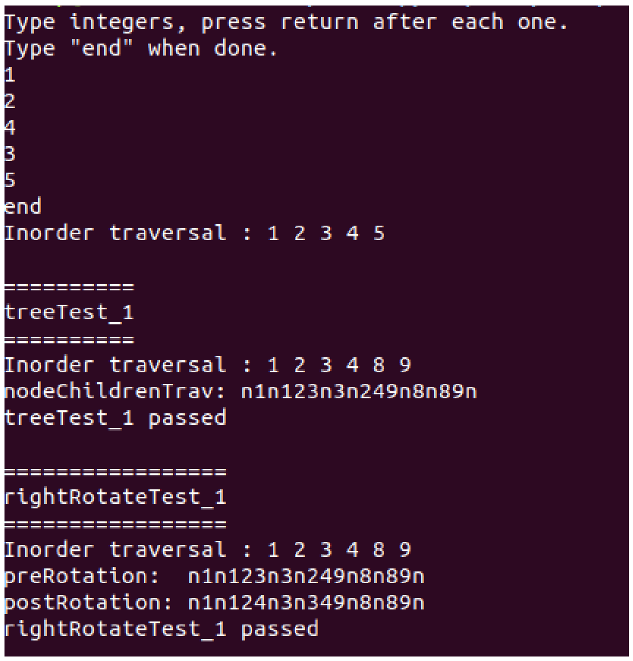
CSCI241, Winter 2020, Lab 5
Check Canvas for due date and time

Unlike the homework assignments for CSCI241, you are **encouraged** to work with your peers in completing the labs. However, each student must have their own submission; you cannot exchange files. If any of this is unclear, please ask for further clarification.

1 Overview and Goals

For this lab, you will extend the `BST` and `Node` classes that you wrote for lab 4, to also include a right rotation (refer to the lectures), as well as several unit tests. Unit tests are a formal approach for checking – programmatically – the correctness of your written code.

Screenshots of the behavior of the program are shown in Figure 1, along with schematics of the BST that is created, and the BST after a right rotation where the pivot is the root.



```
Type integers, press return after each one.
Type "end" when done.
1
2
4
3
5
end
Inorder traversal : 1 2 3 4 5

==========
treeTest_1
==========
Inorder traversal : 1 2 3 4 8 9
nodeChildrenTrav: n1n123n3n249n8n89n
treeTest_1 passed

=================
rightRotateTest_1
=================
Inorder traversal : 1 2 3 4 8 9
preRotation:  n1n123n3n249n8n89n
postRotation: n1n124n3n349n8n89n
rightRotateTest_1 passed
```

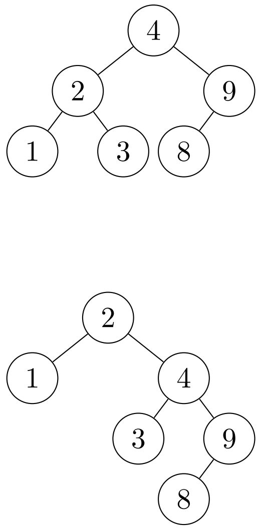


Figure 1: Example invocations (left) of `BST`, in which the user is prompted to input integer values, and a BST is built, and the in-order traversal is performed (this is the same as lab 4). Two test cases are invoked; one creates a tree by inserting 4, 2, 1, 3, 9 and 8 (right top), and the second performs a right rotation around the root node (value 4, shown right bottom). The `nodeChildrenTrav` is the in-order traversal of the tree, where for each node the left child's value, the node's value, and the right childn's value are printed. `n` in the output specifies *null*.

Like lab 4, the instructions for this lab are high-level. You are provided with the function signatures that you need to implement, but the details are left to you. Hint : Refer to the lecture slides for pseudocode – in many cases, almost "real" code – for right rotations.

2 Create a new branch

In your local git (log onto a lab linux computer, and using the terminal, navigate to the directory where you have your code for CSCI241 and where you cloned your git repo), create a new branch named *lab5* – remember to make a new branch from the *master* branch only; refer to lab 1 for a refresher – and then create a new directory, *lab5*. Do all of your work in your lab5 directory. Commit and push often. See previous labs for a refresher on how to check out a branch, how to commit, and how to push to a remote branch.

If you haven't finished lab 4, do that ideally before this lab, else finish it NOW

Copy over your code from your **completed** lab 4 into your lab 5 directory. Use the linux `cd` command to perform the copy task. Ask the TA for help if you have any questions.

3 Code Description, `Node` and `BST` classes.

For this lab, you will extend the two .java files, *Node.java* and *BST.java*. You will add several functions to both classes, and also include two unit cases.

3.1 Add to the `Node` class

Add just two functions, `getLeftChild()` and `getRightChild()`, to the `Node` class. Both should be public, return Node objects, and simply return the `left` or `right` nodes that are pointed to by the class's `left` and `right` class variables.

The `getLeftChild()` function is the following:

```
public Node getLeftChild(){

    return left;

}
```

Yes, that straight-forward. All that the function `getLeftChild` does is return the `Node` object that is referred to by the `left` class variable in the `Node` class.

3.2 Add to the BST class

The modifications that you must make to the **BST** class are a bit more significant. Remember that you should be making edits to the code that you COPIED from lab 4 to lab 5. Do NOT modify your lab 4 code.

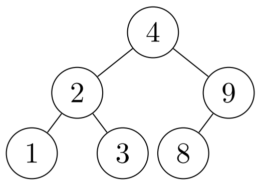
In addition to the already class variable `overallRoot`, add another class variable, and name it inOrderTraverseStr. Its declaration should be the following :

```
private static String inOrderTraverseStr = "";
```

In addition to the `getRoot()` function already in **BST**, also add a `updateRoot` function. The function `updateRoot` should be public, have a void return type, take a single argument of type `Node`, and reassigns the `overallRoot` class variable to refer to a different node.

3.3 The `inOrderTraverse` function

One of the goals of this lab is to write several unit tests, which is one of the many formal methods of testing software. Before you write your unit tests, we need something in addition to in-order traversal, because two trees, with the same nodes, and same in-order traversal, might have quite different tree structures.


(a)

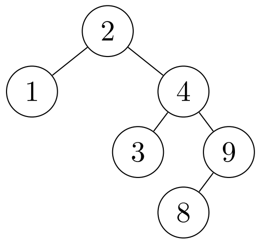

(b)

Figure 2: Both trees (a) and (b) have the same values and in-order traversals, but they have vastly different structures.

Consider the two trees in Figure 1, which are reproduced in Figure 2. Both of them are BSTs, have the identical values, and both of the in-order traversals of the trees are 1, 2, 3, 4, 8 and 9. This is despite the two trees being very different. The children of node 4 in tree (a) in Figure 1 are 2 and 9, while the children of node 4 of the tree in (b) are 3 and 9.

The motivation of the function `inOrderTraverse` is that it performs in-order traversal (the recursive way) as we've seen in lecture and in lab 4, but it prints not just the node's value, but first the value of the node's left child, then the value of the node, then the value of the node's right child.

The `inOrderTraverse` of trees (a) and (b) are therefore the following :

$$n1n123n3n249n8n89n$$

$$n1n124n3n349n8n89n$$

The printing of a tree's nodes, AND their children, is able to distinguish between two BSTs that have the same keys (values) but different structures.

There is one other important fact about trees (a) and (b) in Figure 2. Tree (b) is the result when tree (a) is right rotated where the pivot is the 4 node in tree (a). Refer to the lectures notes, and/or check online (one of the many tutorial resources) to refresh your memory of right rotation.

The skeleton of `inOrderTraverse` function is the following:

```
public static void inOrderTraverse(Node node){

        if (node == null){
            return;
        }
        // recursive call to left child

        // Get the "value" of the left child using the getLeftChild function
        // in Node. If the "value" of the left child is null, append "n" to
        // inOrderTraverseStr, else append the left node’s value.

        // Get the "value" of node that was passed as an argument, and
        // append the value to inOrderTraverseStr.

        // Get the "value" of the right child using the getRightChild
        // function in Node. If the "value" of the right child is null,
        // append "n" to inOrderTraverseStr, else append the right node’s
        // value.

        // recursive call to the right child
}
```

At this stage – before writing any more code! – it is a good idea to test the function `inOrderTraverse`. To do this, in the `main` function, after a BST is created (either manually or via the user’s input as you did for lab 4), you could write:

```
inOrderTraverseStr = "";
inOrderTraverse(tree.getRoot());
System.out.println("nodeChildrenTrav: " + inOrderTraverseStr);
```

4 Unit Testing

Unit Testing is one of several ways that software is tested. The term “unit” in unit testing refers to a small (some say the smallest) testable part of a piece of software. Generally, this refers to functions. The main idea is that rigorous testing is performed EACH time that a unit is modified or integrated into an existing piece of software. In this lab, you will create a new function, `rightRotate`, which implements the right rotation we’ve seen in class, and you will create your first unit tests.

4.1 The function `rightRotate`

Refer to the lecture notes, or one of many online resources, to refresh your understanding of a right rotation. Inside of your `BST` class, implement the `rightRotate` function, whose skeleton is the following :

```
public static Node rightRotate(Node pivot){

        return aNode;

}
```

where `aNode` is of type `Node` and it is the node that is elevated "up" during the right rotation. Hint : your function `rightRotate` should be about 10ish lines of code. You'll need to reason about reassigning left and right children class variables.

4.2 Unit tests

Now that you've written the function `rightRotate`, how might you test it? There are many ways. You can employ an army of print statements, or use a debugger and set break points, or ... whatever works for you. However, the idea of unit testing is that EACH function in a piece of code has a unit test case associated with it, and whenever ANY new code is added to a code base, then ALL of the unit test cases for ALL of the functions are run, to make sure that the addition of any new code didn't break any previously implemented lines of code.

There is nothing especially tricky or complicated with unit testing. Indeed, a unit test case is *just* another function, that performs some sort of pre-defined task, and the output of that task is compared to the expected (as determined using a human "expert") value/output of that task.

For example, a simple unit test for the `BST` class might create a BST t using the BST constructor, insert into that BST the values 4,2,1,3,9, and then 8, and then invoke `inOrderTraverse`. On paper, the traversal of t, where the left and right children are printed, would be the following:

$$n1n123n3n249n8n89n$$

A unit test might then compare the output (into the string variable) of the `inOrderTraverse` invocation of tree t against the physically (hard-coded) string $n1n123n3n249n8n89n$, and print to the screen FAIL or SUCCESS (or something similarly informative), indicating that the unit test has passed. For example, assuming a tree t has been made in the body of the function of the unit test, you could write :

```
inOrderTraverseStr = "";
inOrderTraverse(t.getRoot());
if (inOrderTraverseStr.equals("n1n123n3n249n8n89n")){
    System.out.println("treeTest_1 passed\n");
}else{
    System.out.println("treeTest_1 FAILED\n");
}
```

4.3 `treeTest_1()` and `rightRotateTest_1()` Test cases

Write two test cases (functions), in the `BST` class, and name them `treeTest_1` and `rightRotateTest_1`. Their signatures should be the following :

```
private static void treeTest_1(){

}

private static void rightRotateTest_1(){

}
```

The unit test `treeTest_1` should create a new tree, insert the nodes 4,2,1,3,9, and then 8 (in that order), and then invoke the `inOrderTraverse` function on that tree. The unit test should print to the screen an informative message, something along the lines of SUCCESS or FAILURE, depending on whether the `inOrderTraverse` function generated an inorder traversal string (with children) that is identical to the expected string.

The unit test `rightRotateTest_1` should do the following:

- Create a new BST tree
- Insert the values 4,2,1,3,9 and then 8 into the tree
- Print (using the functions from lab 4) the in order traversal of the tree
- Invoke the `inOrderTraverse` function
- Invoke the `rightRotate` function on the root of the just-created tree. This can be achieved via the following :

  ```
  tree.updateRoot(rightRotate(tree.getRoot()))
  ```

- Invoke the `inOrderTraverse` function (a second time)

Refer to the screen shot (Figure 1) to see what the output of your entire program should be.

5 Compiling and Running

As you did for lab 4, compile and run your program (which you should do incrementally), via the following:

```
javac BST.java
java BST
```

6 Pushing to your git

Add, commit, and push your .java files only, to your lab5 branch. **By convention, compiled code, nor object files, are NEVER (except in special, rare cases, and this is not one of them!) pushed to a git repo**.

```
git add BST.java
git add Node.java
git commit -m "Added this-and-that code"
git push origin lab5
```

Rubric

lab5 branch, which contains the lab5 directory, created. Class files nor anything other than .java files, are excluded	2
Node.java has been modified from Lab 4 to include the functions getLeftChild, getRightChild, and *BST.java* has been modified to include the string inOrderTraverseStr, as well as the functions updateRoot, rightRotate, and inOrderTraverse	5
The unit tests rightRotateTest_1 and treeTest_1, have been written, AND are invoked in the main method.	5
Program correct behavior, which includes all of the functions working as intended. At minimum, your program should be able to reproduce the screenshot shown in Figure 1.	30
The program compiles via `javac BST.java`, and upon invocation, exhibits the correct behavior.	5
The code is commented – including your name, date, brief description, etc. at the top of the .java file, variable names are appropriately named, and good coding practices are followed.	3
Total	60 points

To confirm that you've correctly pushed, open your browser and navigate to the gitlab for CS, and double check that all of the files, as well as branch and directory, are there.

Note : It is a good idea to NOT modify the git repo using the web interface. If you do add files, remove files, etc., via the web interface (insteada of via the command line), you'll need to `pull` the changes when logged in to your linux account.