CISC220H Project 1

Project due November 10.

You may not help other people debug their programs, except in the very limited way described on the class website. You may not use any code written by others, whether they are students or not. The TA and the instructor are available for office hours, and by appointment if you can't make hours because of a class conflict.

Overview

Each group has a different subject area, but in general all groups will need to implement a balanced binary tree and some form of graphical rendering for their final submission. Individual project descriptions will be given out shortly, but we will get started with implementation of an AVL tree.

Assignment

1. **Part 1** In order to balance a binary tree (specifically using the AVL algorithm), we need to perform a rotation operation on the tree.

   In class we defined four cases that we need to detect and treat to maintain our tree balance:

   - Left-Left
   - Right-Right
   - Left-Right
   - Right-Left

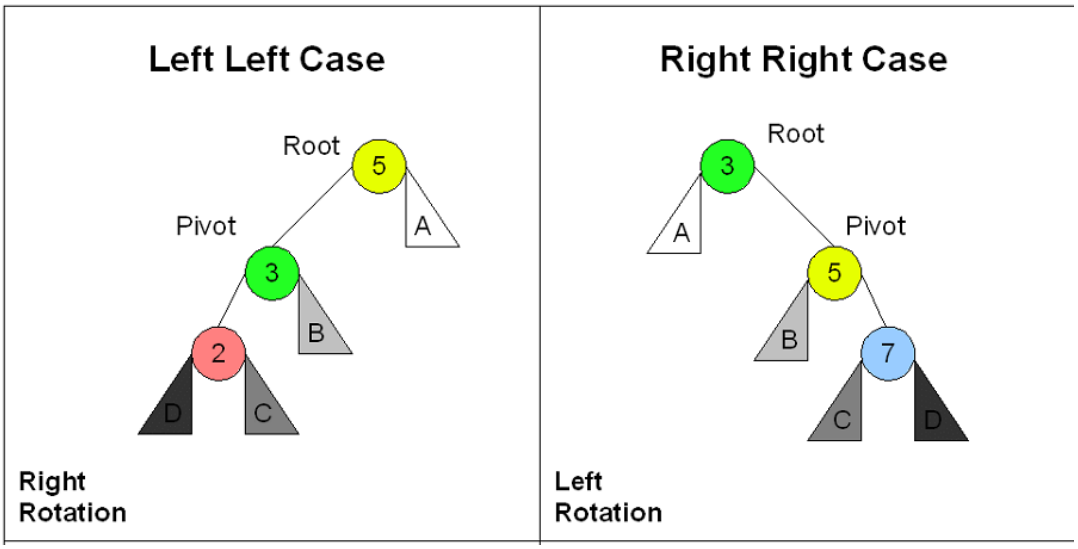


Figure 1: Left-Left and Right-Right before rotation

Implement a `void rotateLeftLeft()` function that can be executed on a root node and will produce the given output. Test this function on an unbalanced tree to verify that it produces the expected result.

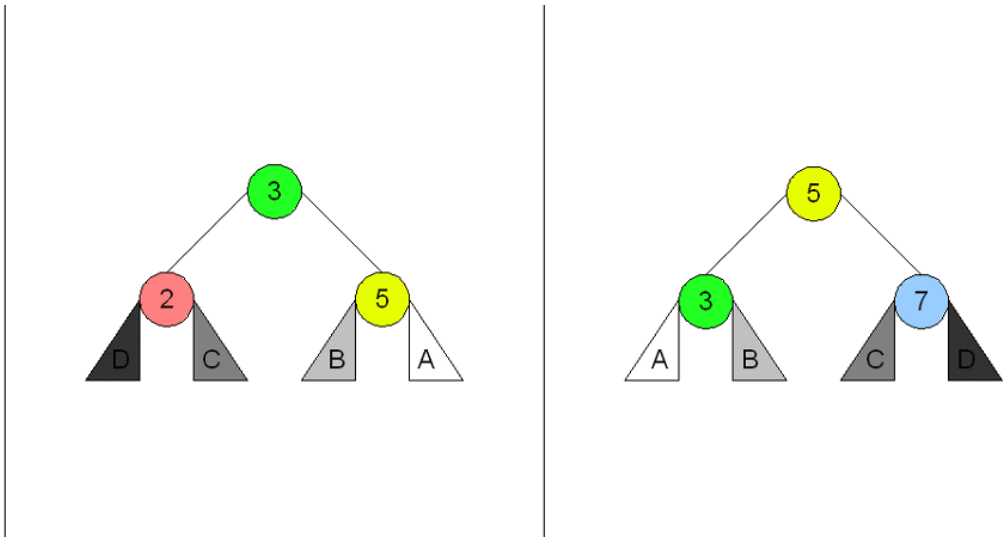


Figure 2: Left-Left and Right-Right after rotation

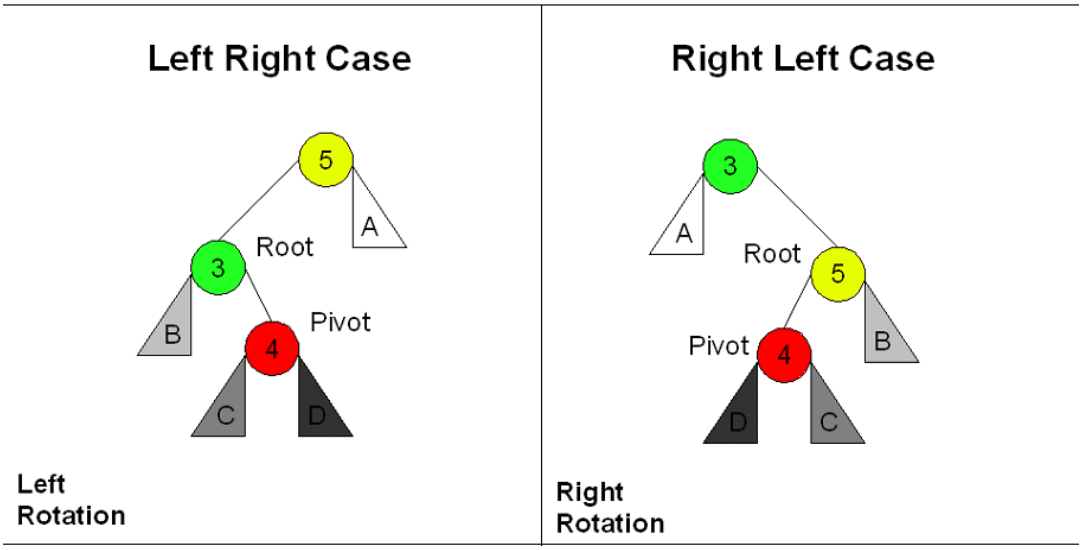


Figure 3: Left-Right and Right-Left before first rotation

Now implement additional operations for `void rotateRightRight()`, `void rotateLeftRight()`, and `void rotateRightLeft()`. Test each of these using an unbalanced binary tree and verify that your resulting tree is balanced.

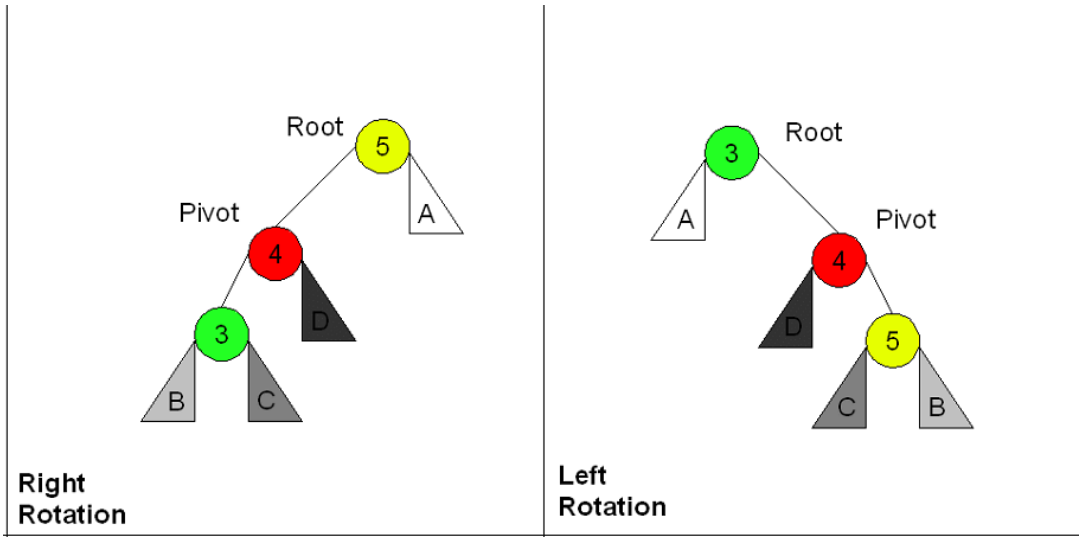


Figure 4: Left-Right and Right-Right after first rotation

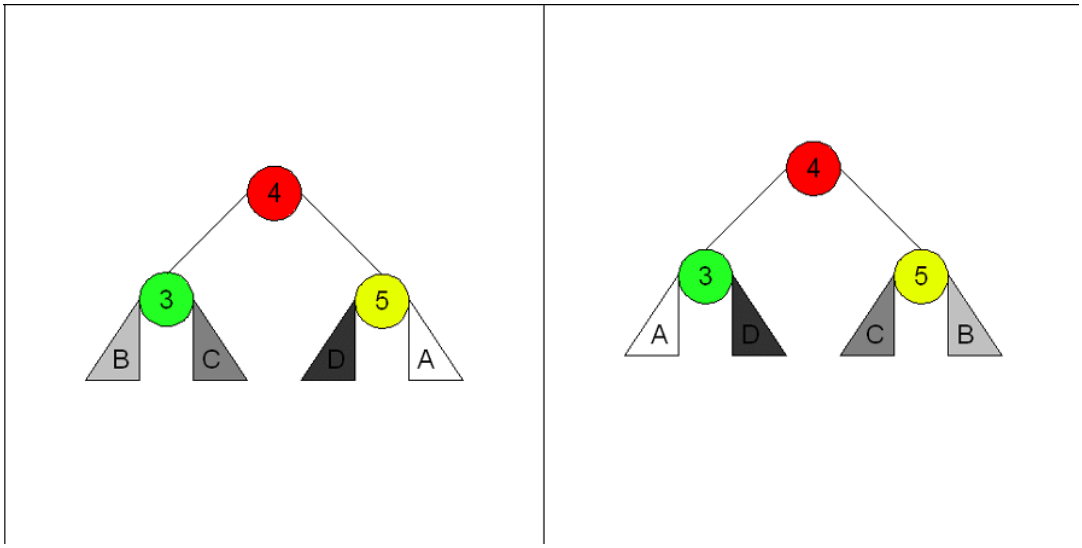


Figure 5: Left-Right and Right-Right after second rotation

2. **Part 2**

   This is an exercise to familiarize yourself with OpenGL and the glut toolkit for graphics development.

   A basic FractalGenerator is available on the course website under project 1. This generator creates a fractal of squares that are nested on two ends of the square. The code that draws this picture does not use any BinaryTree structure; instead it has a series of calls to the drawElement function that draws a picture representing a balanced BinaryTree of depth 4 (you should see 1 + 2 + 4 + 8 = 15 squares in the picture).

   Come up with your own fractal shape (not a square) and modify the given `drawElement` function. You should be able to handle all four arguments for size, center x/y, and rotation for your new fractal shape.

3. **Part 3**

   Modify your BinaryTreeNode class to allow for AVL-Tree balancing. Add an integer field called `balance` and a rebalancing function that calls the four functions you created in Part 1. You will need to modify the functions in Part 1 to return a BinaryTreeNode* to the new root of the subtree.

```
/**
 * Rebalances the subtree rooted at this node. Checks the balance
 * field to determine the correct case and rotation to perform:
 *   LeftLeft case (balance=2 and left child balance = 1)
 *   LeftRight case (balance=2 and left child balance = -1)
 *   RightLeft case (balance=-2 and right child balance = 1)
 *   RightRight case (balance=-2 and right child balance = -1)
 *
 * This function should also update the balance field of this node
 *   and the balance fields of affected children.
 */
BinaryTreeNode* rebalance();
```

4. **Part 4**

   Modify your insert function so that it sets the initial value for `balance` to zero for the new node. The insert function now needs to update the `balance` field every time one of its left or right subtrees increases in depth. The increase in depth should be detected the way we discussed in class (by comparing the left or right child's previous and current balance field) and not by calling a depth function on its children. If the insert function detects that a child's current balance field is equal to -2 or 2 then it should call the rebalance function on that child. Don't forget to update child pointers if the rebalance function returns a different root for the subtree.

5. **Part 5**

   Modify your remove function so that it updates the `balance` field and detects a decrease in subtree depth. If the depth change causes an unbalanced subtree, then call your rebalance function.

6. **Part 6**

   This part begins our customized team based projects (not all of them are ready yet). These all have some background reading to do, and some programming component that uses our binary trees. We will continue to work on these projects as part of the next class project, but this will serve as a starting point.

0.1 Network Security/Packets

Eric, Kevin, Neil

0.2 Text-based Scribblenauts

Austin, Michelle

0.3 Hard Theorem Proving

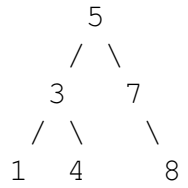
Nick, Pat

Read about automated theorem proving:

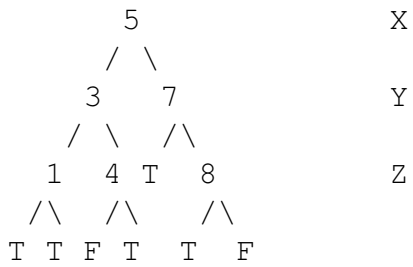
```
http://en.wikipedia.org/wiki/Automated_theorem_proving
http://en.wikipedia.org/wiki/Binary_decision_diagram
```

Given a randomly created binary tree, interpret this tree as a binary decision tree and print a corresponding boolean expression. Each level in the binary tree is a variable. You can assign a random final true/false value to each data element at a leaf node, and any interior node without two children can also have a random final true/false value for the empty child.

For example:



Should be interpreted as a binary decision tree with 3 variables (it has 3 levels), X, Y and Z. If we randomly assign final true/false values to each leaf node we get:



Now we interpret the tree as left meaning false and right meaning true. Thus, if X is true and Y is true and Z is true then the result is false. We could print the boolean expression for this tree as:

```
(X & ((Y & !Z) | !Y)) | (!X & ((Y & Z) | !Y))
```

0.4 Social Network

Jeremy, Diana

0.5 Field/Particle System

Nick, Dylan, Chrissy

Submission of the final Project 1 is due November 10th at midnight. Part 6 will be a starting point for our other projects in the class. You will not have Homework assignments during the Project.