COEN 70: Formal Specification and Advanced Data Structures

Winter 2015

Lab 9: Red-Black Trees

In this lab you will create a Red Black Tree, which is a balanced binary search tree. In a Red Black tree the longest path from the root to a leaf cannot be more than twice of the shortest path from the root to a leaf. This means that the tree is always balanced and the operations are always O(logn).

Since a Red Black Tree is a binary search tree, the following property must be true: the value in every node is larger than the value of the left child (or any value in the left subtree) and smaller than the value of the right child (or any value in the right subtree). Additionally, a Red Black Tree has these properties:

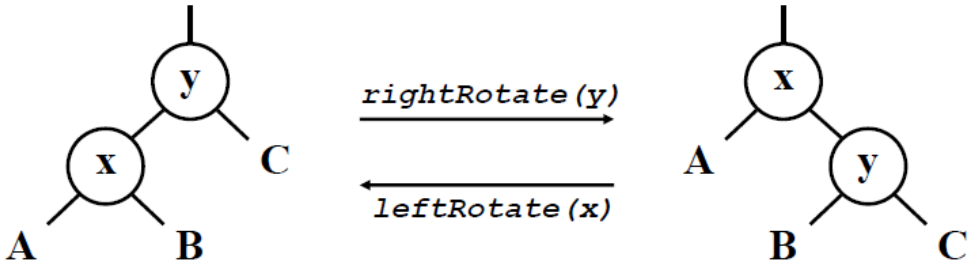
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-black trees consist of nodes which are instances of the class that is given in RBNode.h

Create a Red Black Tree (RBtree) class. First you will have to give the RBtree all the functionality of a binary search tree.

Here are some of the members:

- an instance variable that points to the root RBNode.
- printTree() function: Start at the root node and traverse the tree using preorder
- addNode(int) function: place a new node in the binary search tree with data the parameter and color it red.
- getSibling(RBNode*) function: returns the sibling node of the parameter If the sibling does not exist, then return null.
- getAunt(RBNode*) function: returns the aunt of the parameter or the sibling of the parent node. If the aunt node does not exist, then return null.
- getGrandparent(RBNode*) function: Similar to getAunt() and getSibling().
- rotateLeft(RBNode*) and rotateRight(RBNode*) functions: left, resp. right, rotate around the node parameter. See next figure that demonstrates the two rotations:



- fixTree(RBNode* current) recursive function: recursively traverse the tree to make it a Red Black tree. Here is a description of all the cases for the current pointer:
  1) current is the root node. Make it black and quit.
  2) Parent is black. Quit, the tree is a Red Black Tree.
  3) The current node is red and the parent node is red. The tree is unbalanced and you will have to modify it in the following way.
     I. If the aunt node is empty or black, then there are four sub cases that you have to process.
        A) grandparent –parent(is left child)— current (is right child) case. Solution: rotate the parent left and then continue recursively fixing the tree starting with the original parent.
        B) grandparent –parent (is right child)— current (is left child) case. Solution: rotate the parent right and then continue recursively fixing the tree starting with the original parent.
        C) grandparent –parent (is left child)— current (is left child) case. Solution: make the parent black, make the grandparent red, rotate the grandparent to the right and quit, tree is balanced.
        D) grandparent –parent (is right child)— current (is right child) case. Solution: make the parent black, make the grandparent red, rotate the grandparent to the left, quit tree is balanced.
     II. Else if the aunt is red, then make the parent black. make the aunt black, make the grandparent red and continue recursively fix up the tree starting with the grandparent.

Include a driver program.