CS 341 – Fall 2020
Assignment #4 – Color Me Dawgs
Due: 10/29/2020

You are tasked with creating a binary tree but told that it must remain balanced for optimal searching – how can you accomplish this?

Red-Black Trees are self-balancing binary trees. It achieves this by coloring the nodes of the tree to ensure that insertions and deletions are handled in a balanced sense. The really nice benefit of a Red-Black Tree is that it provides a worst-case guarantee on insertion, deletion, and search times. This is great for real-time applications with strict requirements. On top of that by ensuring balance – they are ideal for searching and provide a significant benefit over using a Linked (or Doubly Linked) List.

For this assignment you are tasked with creating a Red-Black Tree. Blue IV (our faithful CEO) has heard about such a data structure and wants to use it to model athletic roster numbers (here we assume that no double numbers are possible). Blue IV wants to know what the root node of our tree is after all insertions and which nodes are RED and which nodes are BLACK after its all said and done. He has given us a text file that contains athletic roster numbers (integers) and our job is to read-in this file and build a tree from it. He only wants us to deal with insertions and searches for this project – no deletions. Just a reminder (discussed in lecture) a Red-Black Tree has the following properties:

- A Red-Black Tree must be a type of Binary Search Tree
- Every node has a color either RED or BLACK
- The root of the tree is always BLACK
- When a node is added to the tree it begins its life as a RED node
- There are no two adjacent RED nodes
    - A RED node cannot have a RED parent or a RED child
- Every path from a node to any of its descendant (down to the `nullptr`) has the same number of BLACK nodes

A few notes about the specific requirements of the program:

- The program should load SPACE delimited numerical data from a text file (`data.txt`).
- The program should then insert piece of data into a Tree.
    - The Nodes should contain a pointer to their left child, right child, and parent.
    - The Nodes should be assigned an initial color of RED.
- The program will need to ensure balance of the tree by shifting Nodes either left or right in order to maintain optimal ordering.
    - By doing this, Nodes may need to have their color updated to reflect their new position within the Tree to ensure adhering to the Red-Black Tree rules.
- You should display (output) into the console window the following information to the user:
    - Tree Root
    - Red Nodes (Inorder Traversal)
    - Black Nodes (Preorder Traversal)
- The program should handle invalid cases (e.g., invalid text entry, file I/O, etc.).
- The Tree should be placed on the Heap (memory management).
    - The program should contain no memory leaks – make sure to use Valgrind!
        - `valgrind --log-file=valgrind.txt A4.exe`

Development Process:

For the development of your code: you should create three new classes: `RedBlackTree`, `BinaryTree` and `TreeNode`. We will reuse our existing `Node` class – no changes are needed for it.

The `TreeNode` Class should inherit publically from our `Node` Class. This will allow us to access the attributes and methods of that Class to store our data for our tree nodes. The `TreeNode` Class should have the following attributes and operations:

- Private:
  - `TreeNode * leftChild_;`
  - `TreeNode * rightChild_;`
  - `TreeNode * parentNode_;`
  - `Color color_;`
    - This is an Enumeration type in my instance – but you could make it a Class if you so desire.
- Public:
  - Contructor(s)
  - Destructor
  - Accessor Method(s)

The `BinaryTree` Class should have the following attributes and operations:

- Private:
  - `TreeNode * root_;`
- Public:
  - Constructor(s)
  - Destructor
  - Accessor Method(s)
  - `virtual void insert(int data);`
    - This is a convenience method that allows the User to insert data into a Tree without needing to know the specifics – it will simply call the function below.
  - `TreeNode * insertNode(TreeNode * root, TreeNode * node);`
    - This method is responsible for inserting a new `TreeNode` into the tree. The result should be returned as a pointer to the root of the tree.

The `RedBlackTree` Class should inherit publically from our `BinaryTree` Class as a Red-Black Tree is a type of Binary Tree. This will allow us to access the attributes and methods of that Class to allow us to represent a specific type of Binary Tree and include color as a tree feature. The `RedBlackTree` Class should have the following attributes and operations (`*&` = Pass-by-Pointer Reference):

- Private:
  - `void rotateLeft(TreeNode *& root, TreeNode *& newNode);`
    - This allows us to perform a left rotational shift of our tree to ensure proper balance is maintained. It will be called internally from our `balanceColor` method.
  - `void rotateRight(TreeNode *& root, TreeNode *& newNode);`
    - This allows us to perform a right rotational shift of our tree to ensure proper balance is maintained. It will be called internally from our `balanceColor` method.
  - `void balanceColor(TreeNode *& root, TreeNode *& newNode);`
    - This method allows us to maintain proper balance within our tree and properly adjusts the color of each node, if necessary, to adhere to the rules of a Red-Black Tree. This will be a large and complex function – you will need to implement the Red-Black Tree rules that we discuss in lecture here.
- Public:
  - Constructor(s)
  - Destructor
  - `virtual void insert(int data);`
    - We need to override (from our Base Class) this method as in a Red-Black Tree when we insert a new `TreeNode` it must have an associated color designation. We can use the Base Class method `insertNode` within this methods definition. This method will also call the `balanceColor` method described above.
  - `void printRedNodes(TreeNode * root);`
    - You will need to use an Inorder Traversal in this method to search for Red colored `TreeNodes`.
  - `void printBlackNodes(TreeNode * root);`
    - You will need to use a Preorder Traversal in this method to search for Black colored `TreeNodes`.

Tying this all together you will need to write a `driver` that will test your Red-Black Tree and provide the necessary functionality as described earlier as outlined by Blue IV. Your driver program should allow for the user to input a space delimited text file (`data.txt`) to be loaded and built into our Red-Black Tree and then display the Red Nodes, Black Nodes, and Root Node of the constructed Red-Black Tree. Finally, you will need to create a `makefile` that properly links all of this code together and creates an executable named **A4.exe**

An example of sample output is as follows - given the following sample input file:

```
3 18 7 10 22 8 15 29
```

Your output would contain the following:

```
Red Nodes: 8  15  18  29
Black Nodes: 7  3  10  22
Root: 7
```

I will be grading what is located in the **master branch** of your GitHub repository. It is strongly recommended that you commit and push often! Please make use of the Git feature of leaving descriptive messages along with your commits. Be sure to add me to any repository as a collaborator so I can view and grade your submissions. Failure to do so will result in a 0 on the assignment as I will have nothing to grade!

Submission:

All assignments must be submitted on Butler GitHub (github.butler.edu). I will allow you to work on this assignment with up to one (1) other person. If you choose to work on this project with a partner you need to email me letting me know of your "group." Be sure to make note of specific contributions of each team member so I can assign grades accordingly. **Note:** You are NOT required to work with a partner. The name of your Butler GitHub repository must be as follows: **cs341_fall2020_redblack**

The directory structure of the repository must contain the following files:

- **driver.cpp**
- **RedBlackTree.cpp**
- **RedBlackTree.h**
- **BinaryTree.cpp**
- **BinaryTree.h**
- **TreeNode.cpp**
- **TreeNode.h**
- **Node.h**
- **Node.cpp**
- **data.txt**
- **makefile**

Each source file (`.cpp/.h`) **must** include the Honor Pledge and digital signature – if working in pairs, please ensure both students' digital signatures are present on the files.