Eric Sabelhaus - Project 3 Design Documentation

1. Problem Analysis
   1. Design a balanced binary search tree (BBST).
   2. The primary concern is to design a tree structure of nodes and branches with a root node at center. It must be designed in hierarchal fashion where each node aside from the root has a parent or child node linked by a branch left or right.
   3. The secondary concern is to keep the left and right branches from the root node balanced height wise or weight wise.
2. Design Decisions
   1. For the Node object, I chose to cast the instance variables as private, and provide getter and setter methods to modify the attributes. I find that this design prevents accidental assignment of a variable during the processing and evaluation of the data.
   2. For the BBST, I chose the red black tree method for balancing. The use of the RED and BLACK nodes allows the tree to be self-balancing. The root node must always be black, and the color at each level alternates back and forth from red to black. With each insertion, the color is adjusted, and the positioning is flipped to balance the left and right branch leading from the root.
   3. There are also no public methods which access the root Node directly. This is by design, as it could have an adverse effect on the function of tree traversal.
3. Assumptions
   1. My first assumption is that nodes would not need to be deleted, and hence I did not design a deletion method
   2. My second assumption is that there are a finite amount of nodes being inserted into the BBST, as I do not handle whether there is any space remaining in the buffer before each insertion.
4. Classes
   1. Node<E>
      1. E data,
         1. Variable for storing data in a Node. Type assigned by type casting
      2. Node<E>, l(left) and r(right)
         1. Used for the assignment of leaves/branches to a Node
      3. Integer key
         1. Integer used over int for comparable functionality of the data type
      4. boolean color
         1. Setting of the color of the current node
      5. int sub
         1. A counter for the amount of sub-nodes under a given node
      6. public constructor
         1. accepts arguments to assign private instance variables
      7. public Getter/Setter methods
         1. Each private instance variable has a public get and set methods to allow access and assignment to the internal data.
   2. BBST<E>
      1. private static final boolean RED, BLACK
         1. define the boolean which represents red and black Nodes
      2. private boolean isRed(Node<E> node)
         1. helper method to aid the function of self-balancing the Red Black Tree
      3. public void insert(Integer key, E data)
         1. public method to insert a Node into the tree
         2. executes private helper method to perform insert
      4. private Node<E> insert(Node<E> node, Integer key, E data)
         1. recursively inserts nodes below root
         2. Self-balances once insert has completed.
      5. private Node<E> rotateRight(Node<E> node)
         1. helper method to recursively rotate node height from left to right
      6. private Node<E> rotateLeft(Node<E> node)
         1. helper method to recursively rotate node height from right to left
      7. private void flipColors(Node<E> node)
         1. flip the colors of a node and its children nodes
      8. public int size()
         1. executes private integer size() with root node
      9. private int size(Node<E> node)
         1. get the sub of node
      10. public E search(Integer key)
          1. execute private search for key from the root of the tree
      11. private E search(Node<E> node, Integer key)
          1. Uses the compareTo method of Integer to deterministically call itself recursively until the key is found, then return the data at that key.
      12. public int getNumberOfLeaves()
          1. executes private in-order traversal of tree to get the number of leaves
      13. private int inOrderLeafCount(Node<E> node)
          1. performs recursive in-order traversal of tree to count the number of leaves
      14. public int getNumberOfInternalNodes()
          1. executes private in-order traversal of tree to get the number of internal nodes
      15. private int inOrderNodeCount(Node<E> node)
          1. performs recursive in-order traversal of tree to count the number of internal nodes
      16. public void pre()
          1. execute private recursive pre-order traversal of tree
      17. private void preOrder(Node<E> node)
          1. perform recursive pre-order traversal of tree, printing the data of each node
      18. public void in()
          1. execute private recursive in-order traversal of tree
      19. private void inOrder(Node<E> node)
          1. perform recursive in-order traversal of tree, printing the data of each node
      20. public void post()
          1. execute private recursive post-order traversal of tree
      21. private void postOrder(Node<E> node)
          1. perform recursive post-order traversal of tree, printing the data of each node
      22. public int height()
          1. executes private recursive height function with the root Node
      23. private int height(Node<E> node)
          1. performs recursive height count using Math.max
   3. TestBBST
      1. private void testWithStrings()
         1. Junit test of BBST
         2. instantiate test BBST, cast with String
         3. insert 20 string objects at non directly sequential key values
         4. perform all functions per requirements of assignment, printing headings and tailings to separate the console output
5. User Interface
   1. I chose not to use a GUI for the assignment
6. Testing - in sequence, I performed the following method executions on a test BBST object.
   1. in-order traversal;
   2. search for an existing key;
   3. search for a key that does not belong to the BST;
   4. calculate the height;
   5. count the leaves;
   6. count the internal nodes;
   7. insert a new key;
   8. calculate the height;
   9. inorder traversal;
   10. preorder traversal;
   11. postorder traversal;
   12. count the leaves;
   13. count the internal nodes;
7. Error Handling
   1. at any point when recursion is involved, or booleans are expected as a return, I validate whether or not the root node is null. This prevents a null pointer exception occurring during runtime.
8. Lessons Learned
   1. The primary lesson I learned here was the immense benefit this data structure has when dealing with random data. It allows the application to perform intense searches while gaining the performance boost which comes from recursion.
   2. A secondary lesson I learned was that there are different types of logic which can be performed to balance a BBST. I personally came to like the Red Black Tree approach as opposed to defining an assertion after insertion approach. The Red Black Tree method makes for much cleaner code overall, and it has a visual representation that can aid in understanding how it works.