# Depth Analyzing Binary Sort Tree

Michael McMahon

May 2014

# 1. Functional Requirements

This section lists the required functionality of the process, as specified by the problem description.

1. Process will construct a binary sort tree (BST).
   1. BST will contain 1023 nodes.
   2. Values of nodes will be randomly generated, real numbers.
   3. Values will be inserted by calling the *insert()* method.
2. Process will analyze the depth of the BST
   1. Process will output the average depth of leaves in the BST.
   2. Process will output the maximum depth of leaves in the BST.
   3. Requirements 2.a and 2.b will be achieved with three recursive subroutines:
      1. Process uses a recursive routine to count the number of leaves in the BST.
      2. Process uses a recursive routine to sum the depths of all leaves in the BST.
      3. Process uses a recursive routine to find the maximum depth of all leaves in the BST.
      4. Subroutines described in requirements 2.c.ii and 2.c.iii will take a parameter *depth* which is incremented on each recursive call of those subroutines.
3. Process will be run from the command line.
   1. User will enter “java BSTanalyzer” to invoke a main class.

# 2. Solution

This section will address each of the requirements listed in section one. A pseudo-coded algorithm will describe how to implement the logic which satisfies each requirement. Non-obvious design considerations are discussed where applicable.

1. Process will construct a binary sort tree (BST).
   1. **BST will contain 1023 nodes.**

**Pseudo Code:**

//Store constant value 1023 as size

//Instantiate a binary search tree

//Loop from 0 to size

//Get random real number and store in nextReal

//Insert real number into binary search tree

1. Values of nodes will be randomly generated, real numbers.

**Definition of real numbers**

Real numbers can positive or negative, whole or fractional numbers. A random real number can be obtained from a series of calls to the java.util.Random class. The first call will generate a base integer using Random.nextInt(). This base integer will be within the range of Integer.MIN\_VALUE (= -2147483648) and Integer.MAX\_VALUE (= 2147483647). The second call to Random will generate a fraction using Random.nextFloat(). This fraction will be in the range of 0.0 to 1.0. Finally, the random real will be calculated by multiplying the base integer with the fraction. This result in a random real number in the range of Integer.MIN\_VALUE and Integer.MAX\_VALUE. A value in such a range can be safely stored as a float.

**Storing real numbers as float or double**

The randomly generated numbers can be stored as either a float or a double. While a double would provide a higher level of precision, the float will use only half as many bytes (32 vs 64) and so the float is more efficient. Since this process does not perform arithmetic on the random values it generates, loss of precision is not a concern and the float type will be used for its performance advantages.

**Pseudo Code**

//randomReal()

//Get random integer and store as base

//Get random fraction and store as scale

//Return base multiplied by scale

1. Values will be inserted by calling the insert() method.

**Recursive Traversal of Tree**

Inserting a new value into the tree will be achieved by recursively traversing the tree, traversing left if the new value is less than the root’s value, and traversing right if the new value is greater than or equal to the root’s value.

No traversal is necessary if inserting into an empty tree. Insertion consists of simply creating a new node with the inserted value, and storing that node as the top of the tree.

**Recursive and Base Cases**

The recursive case happens when the inserted value is less than the root’s value and the left child is not null, or if the inserted value is greater than the root’s value and the right child is not null. In this case, the method will call itself, passing either the left or right child as the root.

The base case happens when the inserted value is less than the root’s value and the left child is null, or if the inserted value is greater than root’s value and right child is null. The method will create new node and set it as either the left or right child of root.

**Pseudo Code:**

//insert(Float value)

//if value is not null

//If top most root is null

//Create new node with value

//Set new node as the top of the tree

//Return

//If top most root is not null

//Call addChild(root, value)

//addLeaf(Node root, float value)

//If value is greater than or equal to root's value

//If root's rightChild is null

//Create new node with value

//Store new node as root's rightChild

//If root's rightChild is not null

//Call addLeaf(root.rightChild, value)

//If new value is less than root's value

//If root's leftChild is null

//Create new node with value

//Store new node as root's leftChild

//If root's leftChild is not null

//Call addLeaf(root.leftChild, value)

1. Process will analyze the depth of the BST
   1. Process will output the average depth of leaves in the BST.

Calculation

The average depth will be computed by dividing the sum of all leaf depths by the total count of leaves in the tree.

**Pseudo Code:**

//getAverageDepth()

//Get value of sumDepths(root, 1) and store as depthSum

//Get value of countLeaves(root) and store as leafCount

//Get value of depthSum / leafCount and store in depthAvg

//Return depthAvg

* 1. Process will output the maximum depth of leaves in the BST.

The maximum depth will be determined by the algorithm described in section 2.c.iii.

* 1. **Requirements 2.a and 2.b will be achieved with three recursive subroutines:**
     1. **Process uses a recursive routine to count the number of leaves in the BST.**

**Recursive Traversal and Addition**

Leaf counting will be achieved by recursively traversing the tree and summing the count of leaves under the root node’s left and right subtrees.

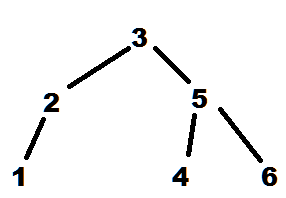
**Recursive and Base Cases**

The recursive case happens when the root is not a leaf and is not null, the method will return the sum of leaves in the root’s left and right sub-trees.

The base case happens when the root is a leaf or the root is null. The method will return 1 for a leaf, or 0 for null.

**Example**

The tree depicted below has 3 leafs.



The algorithm will flow as follows:

**Level 0:**

**countLeaves(node3)**: Not leaf, return countLeaves(node2) + countLeaves(node5).

**Level 1:**

**countLeaves(node2):** Not leaf, return countLeaves(node1) + countLeaves(null).

**countLeaves(node5):** Not leaf, return countLeaves(node4) + countLeaves(node6).

**Level 2:**

**countLeaves(node1):** Is leaf, return 1.

**countLeaves(null):** Is null, return 0.

**countLeaves(node4):** Is leaf, return 1.

**countLeaves(node6):** Is leaf, return 1.

**Level 1:**

**countLeaves(node2):** return 1 + 0 = 1.

**countLeaves(node5):**  return 1 + 1 = 2.

**Level 0:**

**countLeaves(node3):** return 1 + 2 = 3.

**Pseudo Code**

//countLeaves(Node root)

//If root is null

//Return 0

//If root is leaf

//Return 1

//If root is not null or leaf

//Return countLeaves(root.left) + countLeaves(root.right)

* + 1. **Process uses a recursive routine to sum the depths of all leaves in the BST.**

**Recursive Traversal and Addition**

Calculating the sum of all leaf depths will be achieved by recursively traversing the tree and summing the depths of all leaves under the root node’s left and right subtrees.

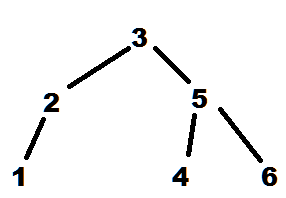
**Recursive and Base Cases**

The recursive case happens when the root is not a leaf and is not null. The method returns the sum of depths in the root’s left and right sub-trees.

The base case happens when the root is a leaf or is null. The method will return the current depth if a leaf, or 0 if null.

**Example**

The tree depicted below has 3 leafs all at depth 3.



The algorithm will flow as follows:

**Level 0:**

**sumDepths(node3, depth = 1)**: Not leaf, return sumDepths(node2, depth + 1) + sumDepths(node5, depth + 1).

**Level 1:**

**sumDepths(node2, depth = 2):** Not leaf, return sumDepths(node1, depth + 1) + sumDepths(null, depth + 1).

**sumDepths(node5, depth = 2):** Not leaf, return sumDepths(node4, depth + 1) + sumDepths(node6, depth + 1).

**Level 2:**

**sumDepths(node1, depth = 3):** Is leaf, return depth 3.

**sumDepths(null, depth = 3):** Is null, return 0.

**sumDepths(node4, depth = 3):** Is leaf, return depth 3.

**sumDepths(node6, depth = 3):** Is leaf, return depth 3.

**Level 1:**

**sumDepths(node2, depth = 2):** return 3 + 0 = 3.

**sumDepths(node5, depth = 2):**  return 3 + 3 = 6.

**Level 0:**

**sumDepths(node3, depth = 1):** return 3 + 6 = 9.

**Pseudo code**

//sumDepths(Node root, int depth)

//If root is null

//Return 0

//If root is leaf

//Return depth

//If root is not null or leaf

//Return sumDepths(root.left, depth + 1) + sumDepths(root.right, depth + 1)

* + 1. **Process uses a recursive routine to find the maximum depth of all leaves in the BST.**

**Recursive Traversal and Comparison**

Finding the maximum leaf depth will be achieved by recursively traversing the tree and comparing the maximum leaf depth under the tree’s left and right subtrees.

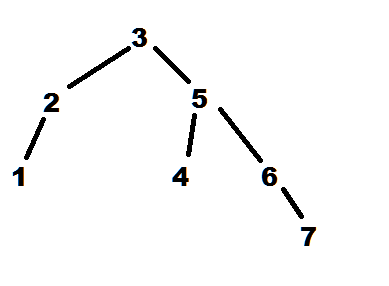
**Recursive and Base Cases**

The recursive case happens when the root is not a leaf and not null. The method compares the maximum leaf depth in the root’s left and right sub-trees, and returns the greater of the two depths.

The base case happens when the root is a leaf or is null. The method will return the current depth if a leaf, or 0 if null.

**Example**

The tree depicted below has 3 leaves. 2 leaves are at depth 3, and one leaf is at depth 4.



**Level 0:**

**maxDepth(node3, depth = 1)**:

Not leaf, return maximum of maxDepth(node2, depth + 1) and maxDepth(node5, depth + 1).

**Level 1:**

**maxDepth(node2, depth = 2):**

Not leaf, return maximum of maxDepth(node1, depth + 1) and maxDepth(null, depth + 1).

**maxDepth(node5, depth = 2):**

Not leaf, return maximum of maxDepth(node4, depth + 1) and maxDepth(node6, depth + 1).

**Level 2:**

**maxDepth(node1, depth = 3):**

Is leaf, return depth 3.

**maxDepth(null, depth = 3):**

Is null, return 0.

**maxDepth(node4, depth = 3):**

Is leaf, return depth 3.

**maxDepth(node6, depth = 3):**

Not leaf, return maximum of maxDepth(null, depth + 1) and maxDepth(node7, depth + 1).

**Level 3:**

**maxDepth(node7, depth = 4):**

Is leaf, return depth 4.

**maxDepth(null, depth = 4):**

Is null, return 0.

**Level 2:**

**maxDepth(node6, depth = 3):** Return maximum of 0 and 4 = 4

**Level 1:**

**maxDepth(node2, depth = 2):**

Return maximum of 3 and 0 = 3.

**maxDepth(node5, depth = 2):**

Return maximum of 3 and 4 = 4.

**Level 0:**

**maxDepth(node3, depth = 1):**

Return maximum of 3 and 4 = 4.

**Pseudo code**

//maxDepth(Node root, int depth)

//If root is null

//Return 0

//If root is leaf

//Return depth

//If root is not null or leaf

//Return maximum of maxDepth(root.left, depth + 1) and maxDepth(root.right, depth + 1)

# 3. Structure

This section will describe the classes which make up the structure of the solution in section 2.

**Class: RandomRealGenerator**

Generates random real numbers in the range of Integer.MIN\_VALUE

(-2147483648) and Integer.MAX\_VALUE (2147483647).

**Fields:**

**private Random randomGenerator**

Instance of the java.util.Random class which generates random values.

**Constructors:**

**public RandomRealGenerator()**

Creates a new random real number generator with a unique seed.

**Methods:**

**public float nextReal()**

Generates the next random real number.

**Class: Node<T>**

Represents a node in a binary tree.

**Fields:**

**public T value**

The value stored by this node

**public Node<T> leftChild**

The left subtree of this node

**public Node<T> rightChild**

The right subtree of this node

**Constructors:**

**public Node(T value)**

Creates a new node with the specified value.

**Class: BinarySerachTree<T extends Comparable<T>>**

A basic binary search tree which stores values in a sorted order. The root’s left children store values less than the root’s value, and the root’s right children store values greater than or equal to the root’s value. Values can be any object type which implements the Comparable interface, such as Float.

**Fields:**

**private Node<T> topRoot**

The top most node of the tree.

**Methods:**

**public void insert(float value)**

Inserts a value into the tree.

**private void addLeaf(Node<T> root, float value)**

Recursively traverses the tree, adding a new leaf such that all left children of root have a value less than or equal to root’s value, and all right children have a value greater than root’s value.

**Class: DepthAnalyzingBST<T extends Comparable<T>>**

This class extends BinarySearchTree, adding methods for analyzing the depth of the tree.

**Methods:**

**public float getAverageDepth()**

Gets the average depth of all leaves in the tree

p**ublic int** **getMaxDepth()**

Gets the maximum depth of all leaves in the tree

**Class: BST\_Analyzer**

This class contains static methods which analyze a binary tree. Trees can be analyzed for number of leaves, sum of leaf depths, and maximum leaf depth.

**Methods:**

**private static boolean isLeaf(Node node)**

Helper method to determine if a node is a leaf.

**public static int countLeaves(Node root)**

Counts the number of leaves in a tree.

**public static int sumDepths(Node root, int depth)**

Sums the depths of all leaves in a tree.

**public static int maxDepth(Node root, int depth)**

Gets the maximum depth of all leaves in a tree.

**Class: Main**

The main class creates a BST and reports the depth analysis results.

**Fields:**

**private static final int DEFAULT\_SIZE**

The default number of nodes to be stored in the tree. Set to value 1023.

**Methods:**

**private DepthAnalyzingBST<Float> createTree(int size)**

Creates a new binary search tree containing the specified amount of randomly generated real numbers

**private void reportAverageDepth(DepthAnalyzingBST<Float> bst)**

Gets the average leaf depth from the specified tree and prints the result to the standard output stream.

**private void reportMaxDepth(DepthAnalyzingBST<Float> bst)**

Gets the maximum leaf depth from the specified tree and prints the result to the standard output stream.

**public void run(int size)**

Creates a binary search tree of the specified size and reports the tree’s average and maximum depth on the standard output stream

**4. Testing**

Since the process generates 1023 random numbers, the output differ on each run. This makes it impossible to define a predictable result which can be tested for when the process is run as a whole. In order to assure that the process is following the expected behavior, the test plan will run validations against isolated logic within the process. If each logical component passes testing in isolation then it can be assured that process is correct when these components are integrated as a complete solution. The logical components to test are as follows:

1. Random number generation
2. Tree insertion
3. Average depth calculation
4. Maximum depth calculation

**4.1 Random Number Generation**

While a predictable outcome for random number generation is not possible, it is possible to test for properties of randomness. In a sequence of randomly generated numbersit would be very unlikely for the same number to appear more than once, or for the same sequence of numbers to appear more than once. This property of non-repetition can be tested for generating a set of random numbers, taking a range of values from the set, and checking if that same sequence of repeats within the entire set. This test is described by the following algorithm:

**Pseudo Code**

//getRandomSet()

//Instantiate an array of size 10,000 and store as setArray

//Loop index from 0 to 9,999

//Get random number

//Store random number in setArray at index

//Return setArray

//checkSequences(seqSize, setArray)

//Instantiate an array of size seqSize and store as seqArray

//Loop seqStart from 0 to setArray size – seqSize

//Loop copyIndex from seqStart to seqSize

//Set seqArray at copyIndex to setArray at seqStart + copyIndex

//Set frequency to result of getFrequency(seqArray, setArray, seqStart + 1)

//If frequency is greater than 0

//Create new results object which stores sequence and frequency

//Store results object in results array

//Return results array

//getFrequency(seqArray, setArray, start)

//Set frequency to 0

//Loop checkIndex from start to (setArray size – seqArray size)

//Set repeat to true

//Set seqIndex to 0

//While repeat is true and seqIndex less than seqArray size

//Set repeat to setArray at start + seqIndex equals seqArray at seqIndex

//Increment seqIndex

//If repeat is true

//Increment frequency

//Return frequency