BST Average-Case Time Complexity

Step 1

As we mentioned previously, the worst-case time complexity of a **Binary Search Tree** is quite poor (linear time), but when the **Binary Search Tree** is well-balanced, the time complexity becomes quite good (logarithmic time). On average, what do we expect to see?

Notice that we can define a **Binary Search Tree** by the insertion order of its elements. If we were to look at all possible **BSTs** with *n* elements, or equivalently, look at all possible insertion orders, it turns out that the average-case time complexity of a successful "find" operation in a binary search tree with *n* elements is actually **O(log *n*)**. So how can we prove it? To do so (with a formal mathematical proof), we will make two assumptions:

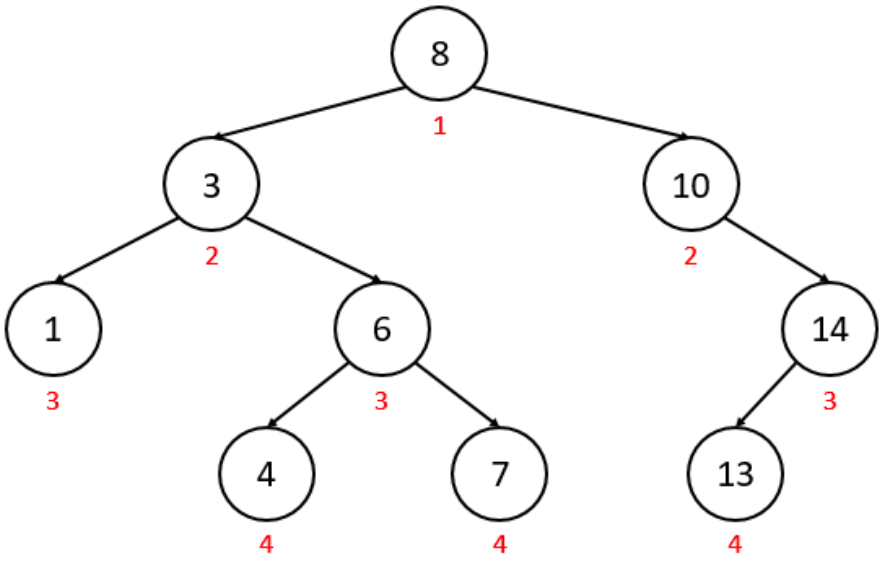
1. **All *n* elements are equally likely to be searched for**
2. **All *n*! possible insertion orders are equally likely**

Step 2

The first step of the proof is to simply change around nomenclature (easy enough, right?). Recall that "time complexity" is really just a metric of the number of operations it takes to execute some algorithm. Therefore, "average-case time complexity" is really just "average-case *number of operations*" (in Big-O notation, as usual). Also, recall from statistics that the "average" of some distribution is simply the "expected value" (a computable value) of the distribution. In our case, our "distribution" is "number of operations", so finding the "average number of operations" is equivalent to computing the "*expected* number of operations". In other words, "finding the average-case time complexity" to perform a successful "find" operation in a **BST** is simply **"computing the expected number of operations"** needed to perform a successful "find" operation in a **BST**.

"Number of operations" is a value we can compute during the execution of our "find" algorithm in a specific case (by simply counting the number of operations executed), but it's a bit of an abstract term. We want to do a formal mathematical proof, so we need to use values that can be derived from the tree itself. Recall that the "find" algorithm starts at the root and traverses left or right, performing a "single comparison" (it really does 3 comparisons in the worst case, but even if it does 3 comparisons at each node, that's still O(1) comparisons at each node) until it finds the node of interest. Therefore, it performs O(1) comparisons at each node on the path from the root to the node.

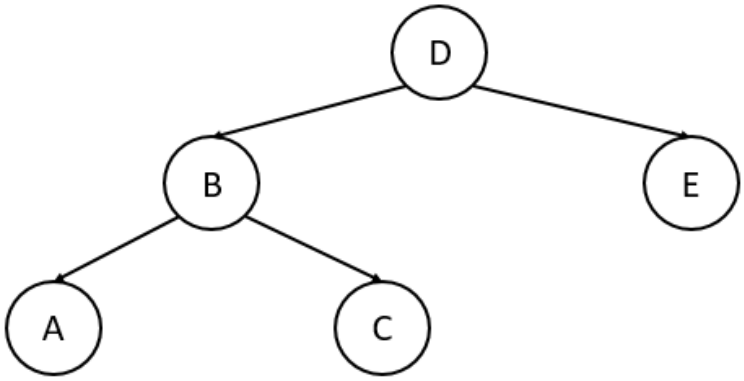
Let's slap on a formal definition: for a node *i* in a **BST**, define the **depth of node *i***, d_i, to be the number of nodes along the path from the root of the **BST** to *i*. The depth of the root is 1, the depths of the root's children are 2, etc. Below is a **BST** with the depth of each node labeled below it in red:



With this definition, we can revise the value we are trying to compute: we are now "**computing the expected *depth***" of a **BST** with *n* nodes.

Step 3

EXERCISE BREAK: Using the same numbering scheme as discussed on the previous step (where the root has a depth of 1), fill in the table below with the depth of each node in the following tree:



To solve this problem please visit https://stepik.org/lesson/28730/step/3

Step 4

Recall from statistics that the expected value of a discrete random variable *X* is simply $\sum_{i=1}^{n} p_i X_i$, where p_i is the probability that outcome X_i occurs. As we mentioned before, our discrete random variable is "depth". In a specific **BST** *j* with *n* nodes, the probability of searching for a node *i* can be denoted as p_{ji}, and the depth of node *i* can be denoted as d_{ji}. Therefore, "computing the expected depth" of a specific **BST** *j* is simply computing $E_j(d) = \sum_{i=1}^{n} p_{ji} d_{ji}$.

Remember those two assumptions we made at the beginning? The first assumption was that "all *n* elements are equally likely to be searched for". Mathematically, this means $p_1 = p_2 = \ldots = p_n = \frac{1}{n}$, so, $E_j(d) = \sum_{i=1}^{n} p_{ji} d_{ji} = \sum_{i=1}^{n} \frac{1}{n} d_{ji} = \frac{1}{n} \sum_{i=1}^{n} d_{ji}$.

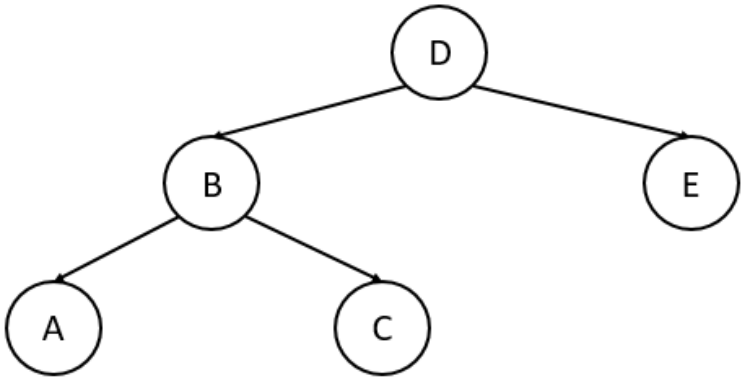
Now, let's introduce yet another formal definition: let the **total depth** of a specific **BST** *j* with *n* nodes be denoted as $D_j(n) = \sum_{i=1}^{n} d_{ji}$. Thus, $E_j(d) = \frac{1}{n} D_j(n)$. However, this is for a *specific* **BST** *j*, but we need to generalize for any arbitrary **BST**!

Let $D(n)$ be the **expected total depth** among ALL **BSTs** with *n* nodes. If we can solve $D(n)$, then our answer, the expected depth of a **BST** with *n* nodes, will simply be $\frac{1}{n} D(n)$ (again because of assumption 1). However, we just introduced a new term! How will we simplify our answer?

Each **BST** *j* is simply the result of insertion order *j* (because a **BST** can be defined by the order in which we inserted its elements). For the first insertion, we can insert any of our *n* elements. For the second insertion, we can insert any of the *n*-1 remaining elements. By this logic, there are $n * (n-1) * (n-2) * \ldots = n!$ possible insertion orders. Based on our second assumption, all insertion orders are equally likely. Therefore, technically, we could rewrite $D(n) = \frac{1}{n!} \sum_{j=1}^{n!} D_j(n)$. But wait... Does this mean we have to enumerate all *n*! possible **BSTs** with *n* nodes and compute the total depth of each? This naive approach sounds far too inefficient! Let's take a step back and rethink this.

Step 5

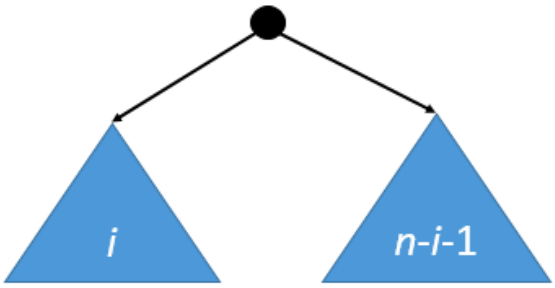
EXERCISE BREAK: What is the total depth of the following tree?



To solve this problem please visit https://stepik.org/lesson/28730/step/5

Step 6

Recall from the previous step that, if we can solve $D(n)$, our answer, the expected depth of a **BST** with *n* nodes, will simply be $\frac{1}{n}D(n)$. Instead of brute-forcing the solution, let's define it as a recurrence relation. Define $D(n|i)$ as the expected total depth of a **BST** with *n* nodes given that there are *i* nodes in its left subtree (and therefore, *n-i*-1 nodes in its right subtree). Below is a general example of such a **BST**:



If we want to rewrite $D(n|i)$ in terms of $D(i)$ and $D(n-i-1)$, we can note that the resulting tree will have all of the depths of the left subtree, all of the depths of the right subtree, and a depth for the root. However, note that every node in the left subtree and every node in the right subtree will now have 1 greater depth, because each node in the two subtrees now has a new ancestor. Thus, $D(n|i) = D(i) + D(n-i-1) + i + (n-i-1) + 1 = D(i) + D(n-i-1) + n$. Note that *i* can be at minimum 0 (because you can have at minimum 0 nodes in your left subtree) and at most *n*-1 (because we have a root node, so at most, the other *n*-1 nodes can be in the left subtree).

Let $P_n(i)$ denote the probability that a **BST** with *n* nodes has exactly *i* nodes in its left subtree (i.e., the probability of having the tree shown above). With basic probability theory, we can see that $D(n) = \sum_{i=0}^{n-1} D(n|i)P_n(i)$. Recall, however, that by definition of a **BST**, the left subtree of the root contains all of the elements that are smaller than the root. Therefore, if there are *i* elements to the left of the root, the root must have been the (*i*+1)-th smallest element of the dataset. Also recall that, also by definition of a **BST**, the root is simply the first element inserted into the **BST**. Therefore, $P_n(i)$ is simply the probability that, out of *n* elements total, we happened to insert the (*i*+1)-th smallest element first. Because of assumption 2, which says that all insertion orders are equally probable (and thus all of the *n* elements are equally likely to be inserted first), $P_n(i) = \frac{1}{n}$, so $D(n) = \sum_{i=0}^{n-1} \frac{1}{n} D(n|i) = \frac{1}{n} \sum_{i=0}^{n-1} [D(i) + D(n-i-1) + n]$.

Therefore, $D(n) = \frac{1}{n} \sum_{i=0}^{n-1} D(i) + \frac{1}{n} \sum_{i=0}^{n-1} D(n-i-1) + n$

Step 7

EXERCISE BREAK: Recall that, in the previous step, based on one of our two initial assumptions, we derived that the probability that a **BST** with *n* nodes has exactly *i* nodes in its left subtree is $P_n(i) = \frac{1}{n}$. Which of our two initial assumptions was this based on?

To solve this problem please visit https://stepik.org/lesson/28730/step/7

Step 8

As we finished on the previous step, $D(n) = \frac{1}{n}\sum_{i=0}^{n-1} D(i) + \frac{1}{n}\sum_{i=0}^{n-1} D(n-i-1) + n$. However, note that the two sums are actually exactly the same, just counting in opposite directions: $\sum_{i=0}^{n-1} D(i) = D(0) + \ldots + D(n-1)$ and $\sum_{i=0}^{n-1} D(n-i-1) = D(n-1) + \ldots + D(0)$. Therefore, $D(n) = \frac{2}{n}\sum_{i=0}^{n-1} D(i) + n$

Multiply by *n*: $nD(n) = 2\sum_{i=0}^{n-1} D(i) + n^2$

Substitute *n*-1 for *n*: $(n-1)D(n-1) = 2\sum_{i=0}^{n-2} D(i) + (n-1)^2$

Subtract that equation from the one before it: $nD(n) - (n-1)D(n-1) = 2D(n-1) + (n^2) - (n-1)^2$

Collecting terms results in the final recurrence relation: $nD(n) = (n+1)D(n-1) + 2n - 1$

To solve this recurrence relation, divide by $n(n+1)$ to get $\frac{D(n)}{n+1} = \frac{D(n-1)}{n} + \frac{2n-1}{n(n+1)}$

This telescopes nicely down to *n* = 1: $\frac{D(1)}{2} = \frac{D(0)}{1} + \frac{2-1}{(1)(2)}$, so $D(1) = 1$ because $D(0) = 0$ (a tree with 0 nodes by definition has a total depth of 0).

Collecting terms, we get: $\frac{D(n)}{n+1} = \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} = \sum_{i=1}^{n} \frac{2}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}$

To simplify further, we can prove this identity for the second term (by induction): $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$

Therefore: $\sum_{i=1}^{n} \frac{2}{i+1} = 2\sum_{i=1}^{n} \frac{1}{i} + \frac{2}{n+1} - 2 = 2\sum_{i=1}^{n} \frac{1}{i} - \frac{2n}{n+1}$

Thus, the final closed form solution is: $D(n) = 2(n+1)\sum_{i=1}^{n} \frac{1}{i} - 3n$

Step 9

As we concluded on the last step, the final equation is: $D(n) = 2(n+1)\sum_{i=1}^{n} \frac{1}{i} - 3n$

We can approximate $D(n)$: $\sum_{i=1}^{n} \frac{1}{i} \approx \int_1^n \frac{1}{x} dx = \ln(n) - \ln(1) = \ln(n)$

Therefore, the average-case number of comparisons for a successful "find" operation is approximately $2\frac{n+1}{n}\ln(n) - 3 \approx 1.386 \log_2(n)$ for large values of *n*.

Thus, since 1.386 is just a constant, we have formally proven that, in the average-case, given those two assumptions we made initially, a successful "find" operation in a **Binary Search Tree** is indeed **O(log *n*)**.

Step 10

As can be seen, the average-case time complexity of a "find" operation in a **Binary Search Tree** with *n* elements is **O(log *n*)**. However, keep in mind that this proof depended on two key assumptions:

1. All *n* elements are equally likely to be searched for
2. All *n*! possible insertion orders are equally likely

It turns out that, unfortunately for us, real-life data do not usually follow these two assumptions, and as a result, in practice, a regular **Binary Search Tree** does not fare very well. However, is there a way we can be a bit more creative with our binary search tree to improve its average-case (or hopefully even worst-case) time complexity? In the next sections of this text, we will discuss three "self-balancing" tree structures that come about from clever modifications to the typical **Binary Search Tree** that will improve the performance we experience in practice: the **Randomized Search Tree (RST)**, **AVL Tree**, and **Red-Black Tree**.