I think all the nodes which are in tree are close to the O(log2(n)) because if you saw the average case math formula to calculate it n=2^0+2^1+...+2^{k-1} nodes. This is a geometric series, which implies n=2^k-1, equivalently: k = log(n+1).

I don’t think it is not so close to the O(n) because if we saw the nodes and graph values are near to O(log2n).

The reasonable range and the time complexity should range to O(1), O(2), O(logn) and O(n^2).

I feel height for random trees would lead to efficient searches compare to linear search we don’t need to go and check all the nodes we can search the middle and can do fast.

For me the easiest way was to look at a graph of log2(n), where n is the number of nodes in the binary tree. As a table this looks like:

log2(n) = d

log2(1) = 0

log2(2) = 1

log2(4) = 2

log2(8) = 3

log2(16)= 4

log2(32)= 5

log2(64)= 6

and then I draw a little binary tree, this one goes from depth d=0 to d=3:

d=0 O

/ \

d=1 R B

/\ /\

d=2 R B R B

/\ /\ /\ /\

d=3 RBRBRBRB

So as the number of nodes, n, in the tree effectively doubles (e.g., n increases by 8 as it goes from 7 to 15 (which is almost a doubling) when the depth d goes from d=2 to d=3, increasing by 1.) So, the additional amount of processing required (or time required) increases by only 1 additional computation (or iteration), because the amount of processing is related to d.

We can see that we go down only 1 additional level of depth d, from d=2 to d=3, to find the node we want out of all the nodes n, after doubling the number of nodes. This is true because we've now searched the whole tree, well, the half of it that we needed to search to find the node we wanted.

We can write this as d = log2(n), where d tells us how much computation (how many iterations) we need to do (on average) to reach any node in the tree, when there are n nodes in the tree.

Diagram

Description automatically generated

Binary search is an example with complexity O(log n). Let's say that the nodes in the bottom level of the tree in represents items in some sorted collection. Binary search is a divide-and-conquer algorithm, and the drawing shows how we will need (at most) 4 comparisons to find the record we are searching for in this 16-item dataset.

Assume we had instead a dataset with 32 elements. Continue the drawing above to find that we will now need 5 comparisons to find what we are searching for, as the tree has only grown one level deeper when we multiplied the amount of data. As a result, the complexity of the algorithm can be described as a logarithmic order.

It is close to come O(log2(n)).

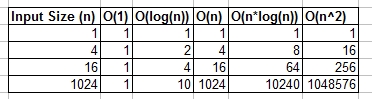
I don’t think it is worst case so there is less chance to get O(n) in this assignment.

Searching: For searching element 2, we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of O(n).

Insertion: For inserting element as left child of 2, we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of O(n).

Deletion: For deletion of element 2, we have to traverse all elements to find 2 (assuming we do breadth first traversal). Therefore, deletion in binary tree has worst case complexity of O(n).

You can think of O(1), O(n), O(logn), etc as classes or categories of growth. Some categories will take more time to do than others. These categories help give us a way of ordering the algorithm performance. Some grown faster as the input n grows. The following table demonstrates said growth numerically. In the table below think of log(n) as the ceiling of log\_2.



The average height of a binary tree with n internal nodes is shown to be asymptotic to 2√πn. This represents the average stack height of the simplest recursive tree traversal algorithm.

**Average Height of Random Binary Search Tree**

In this post, we discuss the average height of a Random Binary Search Tree (BST) (that is 4.31107 ln(N) - 1.9531 ln(N) + O(1)) by discussing various lemmas and their proofs. We omit full proofs and discuss the essential key points for easier understanding.

In short, Average Height of Random Binary Search Tree is:

HN = 4.31107 ln(N) - 1.9531 ln(N) + O(1)

where N is the number of nodes in the Random Binary Search Tree and ln is natural logarithm.

For example, N = 100,000; then Average Height of Random Binary Search Tree is 54.40536 + O(1). The Height of a Balanced BST will be 11.5129.

I feel the tree height for random trees would lead to efficient searches.

Binary tree is a case where a problem of size n is divided into sub-problem of size n/2 until we reach a problem of size 1:

The reasonable range that my heights is 6 foot.

**height of a binary tree**

And that's how you get O(log n) which is the amount of work that needs to be done on the above tree to reach a solution.

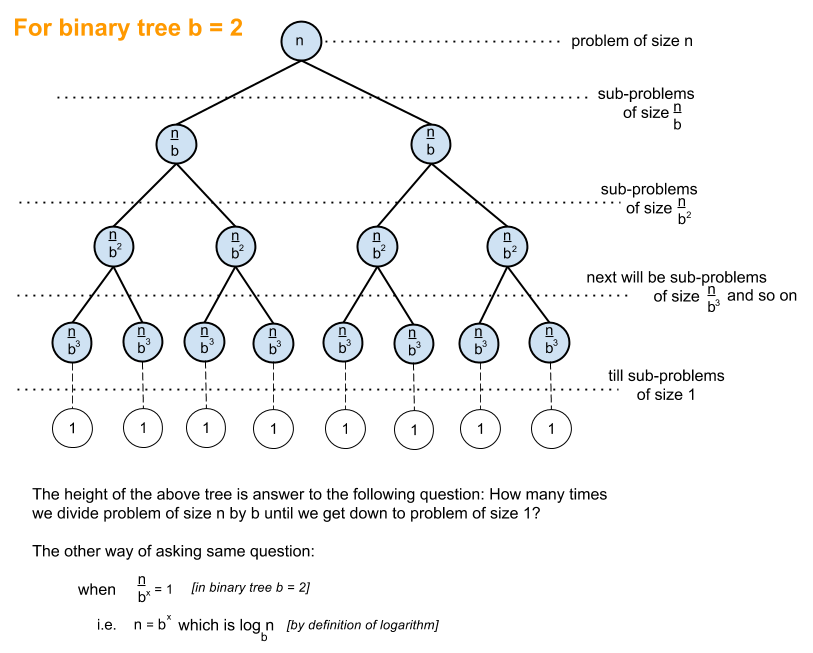
1)Input data needs to be sorted in Binary Search and not in Linear Search

2)Linear search does the sequential access whereas Binary search access data randomly.

3)Time complexity of linear search -O(n) , Binary search has time complexity O(log n).

Linear search performs equality comparisons and Binary search performs ordering comparisons

I feel the tree height for random trees would lead to efficient searches where the binary search is better than the linear search to find the node in the tree.



A common algorithm with O(log n) time complexity is Binary Search whose recursive relation is T(n/2) + O(1) i.e. at every subsequent level of the tree you divide problem into half and do constant amount of additional work.

How do you prove that the expected height of a randomly built binary search tree with 𝑛 nodes is 𝑂(log𝑛)? There is a proof in CLRS Introduction to Algorithms (chapter 12.4), but I don't understand it.

<https://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly>

<https://stackoverflow.com/questions/14426790/why-lookup-in-a-binary-search-tree-is-ologn>