COMPARISON BETWEEN BST(BINARY SEARCH TREE) AND AVL TREE  
Fozilbek Kholboev  
25.01.2022

**INTRODUCTION**

Hello again. In this paper we are going to compare two amazing data structures BST and AVL trees AKA is it really worth the investment in AVL trees in subjects of searching and inserting. In previous paper, we got introduction to algorithms and complexity classes and term of “efficiency” in context of sorting algorithms. So now we know where , why and how algorithms run “efficiently”. Choosing the right data structure is also crucial step for “best” or rather “efficient” algorithm. When it comes to the algorithm which is responsible for searching, the complexities, performance and efficiency of the algorithm boils down to data structure being utilized. Same goes for inserting as well. Having convinced that it is well-rewarding to take some time to investigate these data structures, lets dive right in …

**TREE**

Up until this time, we have seen or dealt with what we can call “linear-data-structures”: such as arrays, linked lists, stacks, queues. All of those are collection of data which is stored in sequential manner. They have logical start and logical end. All you need is a pointer and with pointer arithmetic you can traverse whole data-structure in any direction you want.

Let us introduce ourselves to new type of data structure that is not linear. Tree. Tree is not linear data structure and often used to represent hierarchical data. Tree is most suitable option to represent data that is naturally hierarchical. But this is NOT its ONLY application!

A picture containing text

Description automatically generatedA black and white drawing of a lion

Description automatically generated with medium confidence

Well, you have to look at upside down. Then it resembles our tree.  
We have our root at the top and we branch out to downwards direction.

Logical model of Tree data structure:

A tree can be defined as collection of entities called nodes linked together to simulate a hierarchy.

Icon

Description automatically generated

Now let us familiarize ourselves with the **vocabulary** and introduce some **terminologies** used in tree data structure.

Node – An element which stores a value (predefined and user-defined) and may contain a reference to another node, which is called its Child.  
Parent – Node which has a link to its children.  
Child – Node which has incoming link from another node  
Sibling – Children of same parent   
Edge/Link – One directional reference from parent to its child.  
Root – A topmost node which does Not have a parent.  
Internal nodes – Nodes which have at least one child.  
Leaf – A node which do not have any child.

If we can go from A to B then A is ancestor of B, B is descendent of A.

**Now, lets us talk about properties of Tree:**

* Tree is a recursive data structure. We can define trees recursively as a structure that consists of a distinguished node called a root and some subtrees. And arrangement is such that root of the tree contains link to roots of all subtrees.

A picture containing text, watch

Description automatically generated

* If we have N nodes in a tree, then we will have N-1 edges or links in that tree. All subtrees except the root node.
* Depth of node x: Number of edges in path from root to node x
* Height of node x: Number of edges in longest path from node x to a leaf node.
* Height of tree: Height of root node

Types of Trees.

Based on properties, trees are classified into various categories. There are several trees which are used in different scenarios. Most common type of tree is **Binary Tree**.   
Binary tree is a tree in which any node can have at most 2 children. It can have 0, 1, 2 children. Throughout the paper we are going to deal with Binary Trees (Binary Search Tree and AVL tree is variety forms of Binary Tree) which is the gest of this paper.

**Types of Binary Tree**

* Strict/Proper Binary Tree – each node can have either 2 or 0 children.**A picture containing text, whiteboard

  Description automatically generated**
* Complete Binary Tree – all levels except possible last level are completely filled and all nodes are as left as possible. And maximum number of nodes at level i is 2i
* Perfect Binary Tree – if all levels are completely filled. If h is height of Perfect Binary Tree (Height of tree is number of edges in longest path from root to leaf node). Maximum number of nodes with height h = 20 + 21 + … 2h = 2h+1- 1
* Balanced Binary Tree – difference between height of left and right subtree for every node is not more than k (mostly 1)

Now, we are talking about the things that we are not required to do so, I should confess. However, learning the topic thoroughly always pays off, I believe.

So, lets us talk about **Tree Traversals** which is going to come in handy later.

Tree Traversal is a process of visiting each node in the tree exactly once in some order.

By visiting we mean reading/processing or even printing the data.

Diagram

Description automatically generated with medium confidence

* **Breadth-first AKA Level-Order traversal:**F D J B E G K A C I H
* **Depth-first:**PreOrder - <root> <left> <right> D L R  
  F D B A C E J G I H K  
  InOrder - <left> <root> <right> L D R  
  A B C DE E F GH I J K   
  PostOrder - <left> <right> <root> L R D  
  A C B E D H I G K J F

**BINARY SEARCH TREE**

Binary Search Tree is used for quick insertion/deletion and search operations.

Binary Search Tree:

* a variant of Binary Tree. So, the main property of Binary Tree (every node having at most 2 children) is applied.
* **Binary Search + Binary Tree = Binary Search Tree.** Means that all elements of left subtree must be LESS THAN root and all elements of right subtree must be GREATER THAN root.**A picture containing diagram

  Description automatically generated**

**Insertion**

Our method to insert a new element into BST is somehow similar to what we do in Binary Search method. Let us recall that BST is recursive data structure. So, our insert function better be a recursive one. Plus, BST follows the property such that left subtree of root is smaller (or equal to handle takrorlanuvchi) than root and right subtree is greater than root node.

That is all we need to take consideration into.

Basically, our insert method works in this way. When we are inserting a first node (root) we give one allocation and that will serve as root during the lifetime of the program. Then during next insertions, we just check if new value is less than or equal to root value then we insert the go into left of root. And we keep searching its place till we find the location such that all properties of BST are followed. If new value is greater than root value, then we are going to do same for right subtree of the root.

The key is that the insertion operation requieres time proportional to the height of the tree. If you really think about it, it is true that for a BST containing n elements its height can be organized in various ways.

For example, a BST having 7 elements in this form.**Text

Description automatically generated with medium confidence**

You know every rule is being followed, each node containing maximum 2 nodes and left subtree is smaller than and right subtree is greater than root node value. This is Perfect Binary (which is also Complete Binary Tree) which is the best condition to have when inserting a new element int Tree. Anyway, the sweet spot is in its height. Lesser the height denser the Tree and lesser traversals needed in order to find the correct position for a new element. You check if you need to go towards left/right subtree of BST you do so by reducing whole space by ½ and in next level of recursion you do by ¼ and so on until you reach last level of the tree which happens considering the worst-case scenario! So, on average it takes order of O(n logn) to insert a new element into tree which is containing n elements. But we will achieve this time complexity only when we have a balanced binary search tree.

I mentioned BST containing n elements its height can be organized in various ways. Take the last example, a BST containing 7 elements can also be in this format

**A picture containing key, scissors

Description automatically generated**

Which is as good as linked list. Time complexity for this case will be order of O( n).

This type of BST is a skewed or an unbalanced binary search tree we have to travel from root to last or deepest leaf node and height of the tree becomes n.

**Search**

Searching process will be similar to what we did in insertion. But instead of looking for a suitable location in our recursive BST we look for target element. There can be situations. Either the target will be in the Tree or will NOT be in the tree. So possibly you might sense that time complexity for the situations will be different.

Anyway, in short here is what we plan to do. We check if target is equal to key of the root, we found the element we can return true (or anything to mean existence of the target). If target happens to be smaller, we dive into left subtree of root recursively, or else we dive into right subtree recursively. By doing so, we are reducing the search space by ½ next ¼ … until either we find the element, or we find ourselves at deepest level of the BST left with one element concluding the target value is not in our BST.

Time Complexity

Best-case: Best-case is order of O( 1 ) which is constant, and this happens when the target you are searching for happens to be the root value. More of often than not, this is possible when you have only one element which is the root itself.

Average:

Worst-case: Again, the time taken to search an element boils down to the fact whether the BST is balanced and its height. If it is skewed or unbalanced, we have to do the max recursive dive into deepest levels of BST depending on the key of target.

So, we can claim in worst case, Search is order of O ( height ).

In that case, what is maximum height of the BST containing n nodes? Maximum height (worst balancing of the BST) is n which is as good as linked list. O( n )

What is minimum height? (Best orientation for BST): O( logn )

Anyway, that was when the target existed in the BST.

* **Element that is Not in the Tree**Well, if the target is not in the Tree, we can expect the maximum time complexity for both balanced and unbalanced(skewed) BST. Because we have to go into deepest level of Tree anyway.

**Search** may be successful (element exists) or unsuccessful (not in tree). If successful, means after some recursive dives we find the element and we are going to talk about its time complexity in a second. But if unsuccessful we have to dive recursively at maximum steps (edges). So, what is that going to be equal to? We know that maximum edges are equal to Height of the Tree. So, we can claim that time complexity of Search operation depends on the Height of the Tree.

Minimum Height is log2n  
Maximum Heigh is n.   
  
So, BST Search has best-case as O(log2n) and worst-case O(n) because there is no guarantee that Tree will be Height balanced.

**AVL TREE**

* Variant of Binary Search Tree: So, it follows the property of each node containing at most 2 children. And all elements in left subtree are less than root and in right subtree is bigger.
* SELF-BALANCING Binary Search Tree.

We talked about the most crucial side when it comes to performance characteristics of the insertion and search operations on BST containing N elements. That is its height. Time Complexity of Insertion and Search operation is proportional to Tree’s height. Having know that height of the tree has significant role. What can we do about it? Especially in the times when our Tree is unbalanced.

That’s where AVL Tree comes into play. It is created by  
**A**delson  
**V**elsky  
**L**indas

What does SELF-BALANCING mean? Means balancing heights of left subtree and right subtree. Which is measured by Balancing factor. Balancing factor = height of left – height of right subtree.  
which can be either 0, 1, -1. Every node in AVL Tree should have balancing factor 0, 1, -1.

If Balancing factor is :

0 – Balancing

1 – RIGHT-HEAVY

2 – LEFT-HEAVY

And importantly every node should have these values.

To get make the Tree balanced we are going to use these **Rotations** below:

* LL imbalance

Shape

Description automatically generated

We have to rotate this tree toward right so it will get like this shape:

Diagram

Description automatically generated

* RR imbalance

Shape

Description automatically generated

We must rotate this toward left and will get this shape:

Diagram

Description automatically generated

* LR imbalance

Shape

Description automatically generated

This imbalance can be balanced in two steps. First, we get this one as LL and from there we rotate this towards right.

* RL imbalance

Shape

Description automatically generated

This imbalance also, can be done with two steps. First, we get from this to RR and from there we rotate towards left.

**Insertion** in AVL tree.

Insertion operation of AVL tree is similar to Insertion of BST. But if we concentrate on the definition of the AVL tree carefully, it says AVL tree is Height Balanced BST. Of course, we insert at most 2 children and smaller to left child and bigger to right child. Additionally, we also have to check if the insertion of new node does not violate the property of height balancing. So, time taken by AVL Tree insertions = BST insertions plus at most two rotations. Since binary search tree insertions take O(h) time, rotations are O(1) time, and AVL trees have h = O(log n), AVL insertions take **O(log n)**

**as its worst-case.**

**Search** in AVL tree

As we know, our search may be successful or unsuccessful. Apart from unsuccessful part, you are getting a guarantee that our target will be found in O(log2n) because the height is balanced.

That’s why time complexity of Search operation has **O(log2n) as its worst-case**.

Coming to the situation where the element does not exist in the Tree during the Search, we still do maximum recursive dive till the deepest level of Tree.

**METHODOLOGY**

The data structure and algorithms are implemented in C++. We tried get as much of C++ high level topics to ease our “already-complicated-enough” task. Specifically, we used abstract type (class) and inheritance. Because BST and AVL has “is-a” relationship such that everything applies in BST should apply in AVL as well as AVL extends the BST by self-balancing. We have 3 types of data (pseudo-random, ordered, low entropy) in which we tested insertion operation of new element to our data structures containing n elements. As well as searching an element that is in the Tree and element that is NOT in the Tree. We created 3 separate loops and after we insert the element, we try to search the element inserted (that will be in the Tree) and the one element after (will not be in the Tree) and stored the result data in corresponding files.

**OBSERVATIONS**

In this section I would like to tell some theoretical assumptions that can be made how our data structures will perform based on the core ideas we have used to implement the data structures and algorithms. If we end up with near as our expectation in RESULTS section, we know we succeeded bringing from theory to reality.

First of Insertion, we know that BST will take the input and inserts it and that is it. But AVL Tree insert the element (like BST does) but additionally takes care of balance of Height which is going to take more time. So, we can expect with all 3 data types (pseudo-random, ordered, low entropy) AVL Tree takes several times more time.

Coming to Search, we have to see the effect of all time spent of balancing the heights in AVL Tree.

We know that the height of AVL Tree is always going to be log2h. By that fact we are earning ourselves a guarantee such that time complexity of the operation will be O(log2n) in **worst case.**

So, what does that mean is that AVL search will be several times faster that BST search (especially when we are dealing with ordered data).

**RESULTS**

The results of Insertion operation pseudo random data of both data structures are displayed in fig1

As expected AVL tree took several more times longer in Insertion operation. But we can also observe that as the size of AVL Tree n increases time is also slowly increasing. Because when we insert new element there may be several nodes in several levels might be rebalanced.

Here is how they are performing with ordered data

Again, AVL is expected to take more time than BST. But here are more interesting things happen. We can see that when we are working with ordered data BST took considerable amount of more time than it does with pseudo-random data. Curious! Of course, that is because our data is ordered(ascending) and BST inserts the new element to new level (height increases simultaneously). That is why we have to go to deepest levels of Tree at every insertion!

Here is with low entropy data

Search Operation.

**CONCLUSION**