AVL Tree

SML Graph Theory (Spring 2024)

Vilem Fiala

July 2024

1 Introduction

In this report I will be introducing a new structure that is related to Binary Search Trees. It is called AVL Tree, and it is a structure above Binary Search Tree that is keeping the whole tree balanced. As with regular Binary Search Tree, it can have any shape, meanwhile AVL Tree keeps the shape of the tree as triangular as possible. Ensuring that functions for searching vertices with specific numbers will be as fast as possible.

2 Binary Search Tree

First I would like to remind us definition of Binary Search Tree from class.

Definition 3.16 (Binary Search Tree (BST)). Binary search tree (BST), also called ordered or sorted bin trees is a binary tree T = (V,E) together with a (weight) function $\omega : V \to S$, with $(S, \leq)$ a totally ordered set, such that for any $x \in V$:

- $\omega(y) \leq \omega(x)$ for any $y \in L(x)$,
- $\omega(x) \leq \omega(y)$ for any $y \in R(x)$,

where $L(x)$ and $R(x)$ are the left and and the right subtree defined below x. The values at the vertices, namely $\{\omega(x) \in S \mid x \in V\}$, are called the keys.

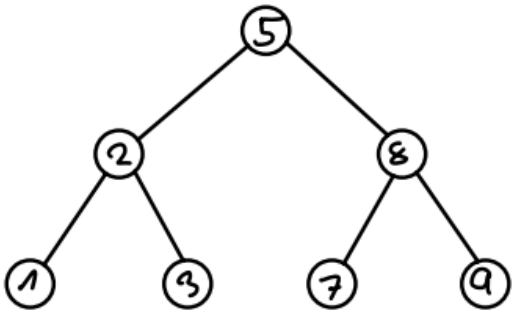


Figure 1: Example of Binary Search Tree

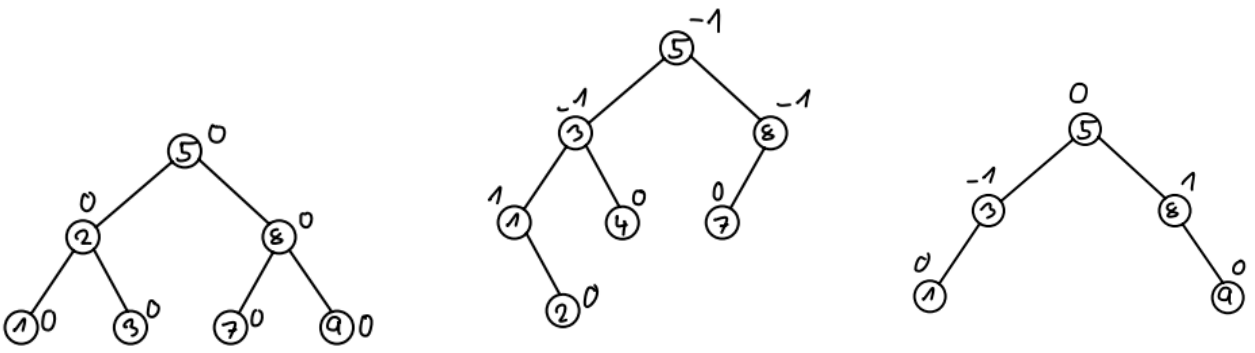
3 Problem of regular Binary Search Tree

With regular Binary Search Tree comes a problem if it is not a balanced tree. As with inserting and deleting vertices from the tree can result into creating tree that is not balanced. Meaning some sides are deeper than others. That can result in a not optimal search time of a vertex with a specific key.

Definition (Height-Balanced Tree). Binary search tree (BST), is called height-balanced tree, if both of his subtrees are equally deep or the difference in depth is at most 1 (it can happen that we have extra vertices making it uneven, we can't always have totally balanced tree with the same amount of vertices on both sides). I will be calling the height-balanced tree just balanced tree. For any $x \in V$ must be true:

- $|d(L(x)) - d(R(x))| \leq 1$,

where $d(L(x))$ and $d(R(x))$ is the depth of the left and the right subtree defined below x.[1]



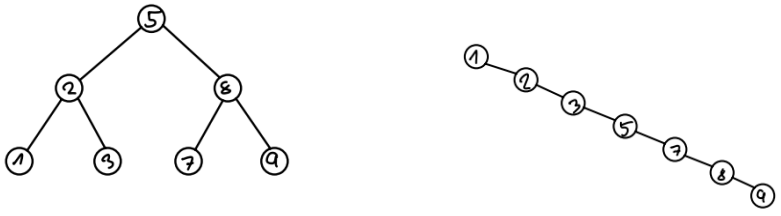
(a) Perfectly balanced BST (b) Balanced BST (c) Balanced BST

Figure 2: Examples of balanced BST with $d(R(x)) - d(L(x))$

3.1 Balanced vs Unbalanced BST

Use of BST is to have a quick way of adding, removing and finding vertices with keys as an ordered set, meaning we want to have these operations as quick as possible.

With that comes a problem if the tree is unbalanced, meaning some of these functions can not be optimal and take longer time. If we have a perfectly balanced BST with n vertices, search function requires an average of $O(log(n))$ operations. But if we insert vertices with key already in ordered form, we will get a tree that has depth of n, meaning that search can take up to $O(n)$ operations.



(a) Perfectly balanced BST (b) Completely unbalanced BST

Figure 3: Comparing balanced and unbalanced BST

In the BST in Figure 3: (b) we can see that if we want to find vertex with key 9, we need to do 7 operations of comparing if the key of the vertex is equal with our target. Meanwhile in the BST in Figure 3: (a) we do only 3 operations of comparing to find the vertex with key 9.

That's why for effectiveness of the BST we need to introduce some operations to keep our BST balanced and making the operations as fast as possible.

4 AVL Tree

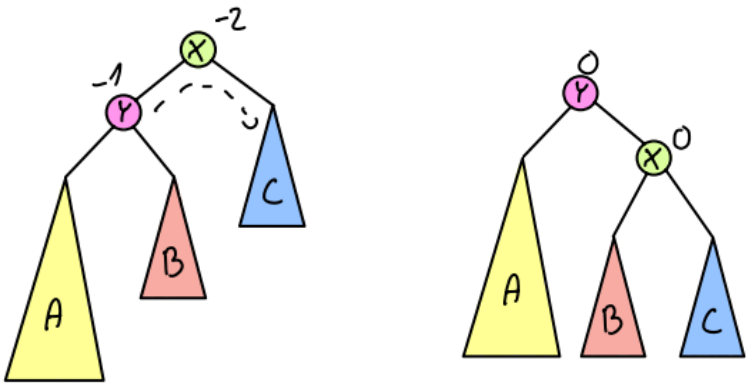
To keep the BST balanced, we are introducing a new structure over BST that will help us keep it balanced, called AVL Tree (after their inventors: Adelson-Velsky and Landis). AVL Tree is a balanced binary search tree where the height of the two subtrees of a vertex differs by at most one. Search, insert, and delete are $O(log(n))$, where n is the number of vertices in the tree.[2]

The only difference is that we have to keep information at every vertex if its subtrees' depths are balanced, thanks to balance factor $BF(x) = d(R(x)) - d(L(x))$. Based on the balance factor we will know what operation to do, when the tree becomes unbalanced, as the BF will be either -2 or $+2$ (left subtree is deeper/right subtree is deeper).

After every insert and delete from BST we have to update these numbers and look for the unbalanced ones. We are going from the inserted or removed vertex in the direction to the root, so accessing their ancestors, updating the balance factor, if the depth changed. If the operations creates an unbalanced BST we have to fix it. We have four operations based on the current state of the BST.

4.1 Simple right rotation

This operation is used if the balance factor of a specific vertex X is -2 and vertex $Y = L(x)$ (which is the root of X's left subtree) has balance factor -1. If BF for Y would be $+1$, we will use an operation that we introduce later. And if the BF was 0, while maintaining the BF at X -2, that wouldn't be possible, as that would mean that the BF at Y was previously either -1 or $+1$, while the balance factor at X was still -2, meaning we would have to do the rebalancing earlier already. The depth of subtrees are $d(B) = d(C) = d(A) - 1$. The depths of these subtrees will always be like that, as if there were different, the rebalancing would come earlier when going up through the tree or not at all. For example if we change the depth of subtree C to $d(C) = d(C) - 1$, we would get the balance factor for vertex X to be -3, which is not possible in correctly working AVL Tree, as if its -3, it had to be -2 before, meaning we were supposed to rebalance it earlier. We can try changing depths of every subtree, and it will always come to a result that either there was supposed to be rebalancing earlier or not at all. For example: new depth for A would be $d(A) = d(A) + 1$ or changing the depth of B to $d(B) = d(B) - 1$, both of those actions making the balance factor at vertex Y to -2, meaning we were supposed to do the rebalancing on vertex Y. Example for removing the need for rebalancing: the depth of subtree C would be changed to $d(C) = d(C) + 1$, that would change the balance factor at X to -1, removing the need for balancing. Or changing the depth of A to $d(A) = d(A) - 1$ would make the balance factor at Y to 0 and balance factor at X to -1. After finishing the rotation, both BF of X and Y will be 0.



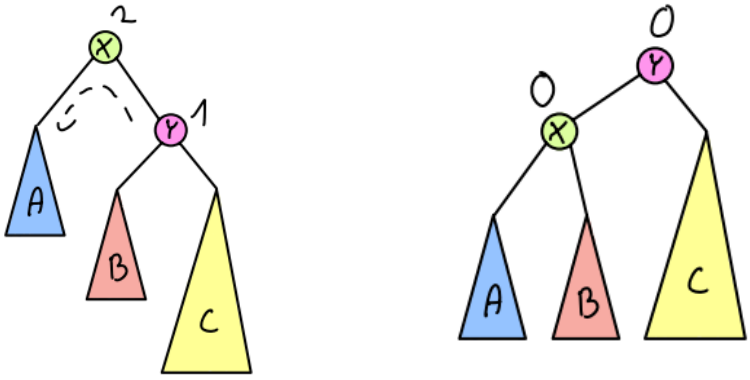
(a) Unbalanced BST (b) Balanced BST

Figure 4: Example of Simple right rotation

We can be sure that the correct order and structure of BST will be the same, as all vertices in subtree B on the picture have key higher than vertex Y, but smaller than vertex X, which is still true after the rotation. We can also look at the position of specified structures (vertices X and Y and subtrees A, B, C are still in the same order and at the same positions between each other, meaning the order was kept.

4.2 Simple left rotation

This situation is just mirrored situation of the simple right rotation. Meaning that the balance factor of specific vertex X is 2 and for vertex Y (which is the root of X's right subtree) is 1. The depths of subtrees are $d(A) = d(B) = d(C) - 1$. We can again check for different depths for subtrees, like we did in previous section, but it will have the same results. Again, after finishing this rotation, both vertices will have the balance factor equal to 0.



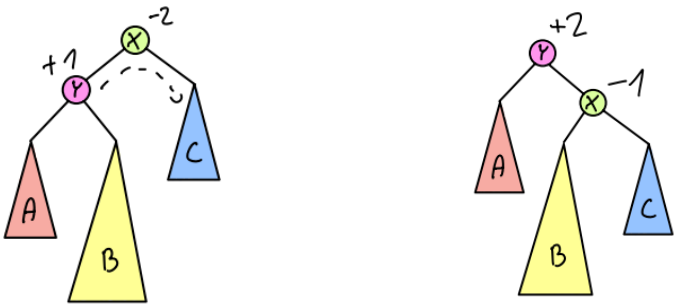
(a) Unbalanced BST (b) Balanced BST

Figure 5: Example of Simple left rotation

Again, the structure and order of BST will be kept, as all keys on vertices in subtree B in the image are bigger than vertex X's key and smaller than vertex Y's key. We can also again look at the position of specified structures (vertices X and Y and subtrees A, B, C are still at the same positions between each other, meaning the order was kept).

4.3 Problem with simple rotations

But we can't always rely on the simple rotations, as there are times when this simple rotation (that's why it is called a simple rotation) can't fix the difference of the depth of the subtrees, as shown in the next example below:



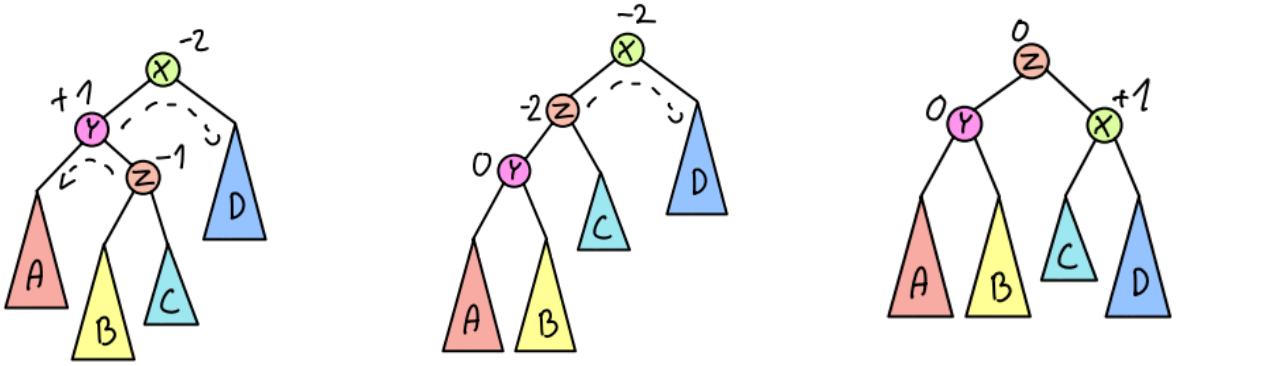
(a) Unbalanced BST (b) Still unbalanced BST

Figure 6: Example of failure of simple rotation

That's why we need to introduce new type of rotation, called double rotation, which will basically do two rotations in a row to fix the difference in the depths.

4.4 Double LR rotation

Double LR rotation will be used when the balance factor of a specific vertex X is -2 and balance factor of his left ascendant Y ($Y = L(X)$) is $+1$. The difference compare to the simple right rotation is, that here the balance factor at Y is $+1$ and not -1 like in the simple right rotation. We have to introduce another vertex, Z ($Z = R(Y)$). The balance factor of Z can be -1 or 1, if it was 0, it would mean that either the BF at Y would be 0 and BF at X would be -1, removing the need for rebalancing. Or other option would be that BF at X and Y remain the same, that would mean we had to do the rebalancing earlier, as the BF at Z was before either -1 or $+1$. The balance factor of Z cannot be $+2$ or -2 as if it was the case, we would be doing some kind of rotation for Z to balance it out. The depths of subtrees are $d(A) = d(B) = d(D) = d(C) + 1$. Here as well we can experiment with different depths, trying to find a different possibility, but it will result in need of earlier balancing or no balancing at all. Except changing the depths of the subtrees that are ascendants of the vertex Z as we stated before, the balance factor of Z can be -1 or 1. After finishing both rotations, vertices Y and Z will have balance factor 0 and X will have $+1$.



(a) Unbalanced BST

(b) After first rotation to the left

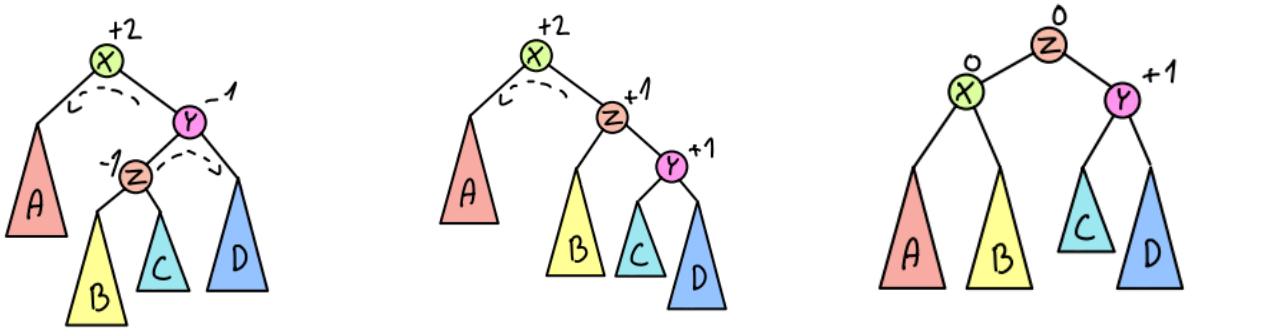
(c) Balanced BST after second rotation to the right

Figure 7: Example of Double LR rotation

We can see that after the first rotation, the balance factor gets bit worse, but finishing the second rotation fixes everything. Also just by looking at the subtrees and highlighted vertices, we can be sure that the structure of BST is intact. Order of subtrees is still A, B, C, D, the order of highlighted vertices is also the same and we can see the subtrees being between the highlighted vertices, for example subtree B is still between Y and Z.

4.5 Double RL rotation

Double RL rotation is just again mirrored version of Double LR rotation. It will be used when the balance factor of a specific vertex X is $+2$ and balance factor of his right ascendant Y ($Y = R(X)$) is -1. We can notice the difference compare to the simple left rotation, as there the balance factor for vertex Y was $+1$. Again, we introduce another vertex, Z ($Z = L(Y)$). Here also the balance factor of Z can be -1 or 1, 0 would have the same problem as in LR rotation. Here we have -1 which is different compared to our example for Double LR rotation, as it isn't mirrored, so we will see different middle picture. The depths of subtrees are $d(A) = d(B) = d(D) = d(C) + 1$. Again changing the depths of individual subtrees will result in the same thing as many times before. After finishing both rotations, vertices X and Z will have balance factor 0 and Y will have $+1$.



(a) Unbalanced BST

(b) After first rotation to the right

(c) Balanced BST after second rotation to the left

Figure 8: Example of Double RL rotation

We can see that the order of both subtrees and vertices is still the same, keeping the structure of BST intact and ensuring it will still be ordered.

5 Conclusion

AVL Tree is really effective improvement for regular BST as it can improve the worst speed for search, insert and delete operations from $O(n)$ to $O(log(n))$, where n is number of vertices in the BST.

This structure would be used for a problem, where we don't know how many vertices we will have in the tree, and that we don't know how often we will access every vertex, meaning that there won't be some vertices that we access more times than others.

Especially in programming, if we ever use BST as a mode of keeping an ordered set of numbers and enabling quick search, insert and delete operation, we should always implement AVL Tree, as it will, in the long run, save a decent amount of time.

References

[1] Paul E. Black, "height-balanced tree", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 20 December 2004. (accessed 7.7. 2024) Available from: https://www.nist.gov/dads/HTML/heightBalancedTree.html

[2] Paul E. Black, "AVL tree", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 12 November 2019. (accessed 7.7. 2024) Available from: https://www.nist.gov/dads/HTML/avltree.html