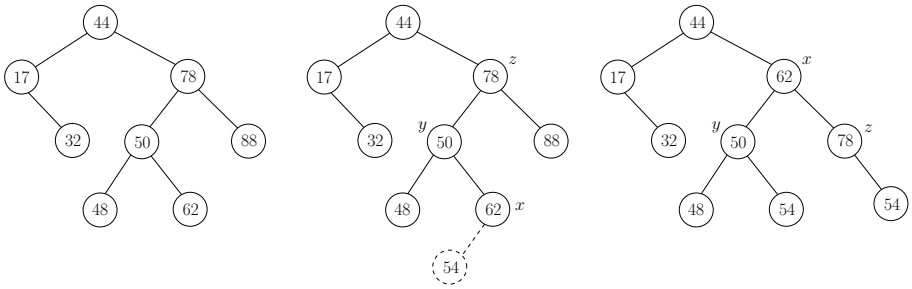
Data Structure and Algorithm
AVL Trees
Insertion and Deletion

Apurba Sarkar

October 11, 2018

1 Insertion

In this handout we are going to discuss about few operations on AVL trees in particular we are going to look at the insertion and deletion procedure in an AVL tree. Suppose we want to insert a node v into an AVL tree. Recall the process of insertion of a node in a binary search tree. To insert a node we first find the location where the key is to be inserted, then we go to that place and put the key there. Let say the node, which is inserted in the AVL, tree is v. If as a consequence of this insertion the tree does not remain an AVL tree then it is because the height balance property is violated in some of the nodes. Now the nodes whose heights could change as a result of this insertion are the ancestors of this particular node v. It is because only in the ancestors of that node whose sub-tree has changed as result of this insertion process. For any other node its sub tree has not changed and they remain the same as before. The change means that their height will only increase because we have added a node. So it is the ancestor of these nodes whose height might increase as a consequence of this insertion. Now, if the insertion causes the tree to become imbalance or unbalanced, then some ancestors of this node v are the culprit and it is the place where one or more ancestors would have height imbalance. Height imbalance means the difference in heights of the left sub-tree and right sub-tree is more than a one. To find the first node, which became imbalance as a result of insertion, we are going to essentially travel up the tree from this node v in the path towards the root. It is to be noted that traveling up the tree means just keep following the parent pointer till we identify the first node z which is unbalanced. It is same as saying traveling till we find the first node x whose grandparent is unbalanced, but you can also think of it as, node z is the first node which is unbalanced and x is its grandchild and x is not any grandchild but the grandchild that we traversed or went through the on the path from newly inserted node towards the root. We will call y as the parent of x. So y is the parent of x and y is the child of z. Consider the tree in the figure Fig. 1a bellow. This is an AVL tree and suppose 54 is inserted into this tree as shown in Fig. 1b. So 54 would come in the position as shown in the figure Fig. 1b by dashed circle. The reason being it is larger than 44, smaller than 78, larger than 50, smaller than 62 and so it would come as a node in the left sub-tree of 62. Now as a result of the insertion this tree does not remain height balanced any



(a) before insertion (b) after insertion (c) after rotation

Figure 1: Tree before and after insertion and after rotation

(a) before rotation (b) after rotation

Figure 2: Understanding the rotation

more. It is to be observed that if there is a problem of height imbalance after insertion, it would be in one of the ancestors of the newly inserted node because it is only the ancestors nodes whose heights are changed. Before insertion, the node 62 had a height of 1, now it is 2. The node 50 had a height of 2, now it is 3. The node 78 had a height of 3, now it is 4. The node 44 had a height of 4, now it is 5. Among these ancestors the first node on this path which is now imbalanced is the node 78. The node 50 is not imbalanced as the difference of height is 1. The node 78 become imbalanced as the difference of height is 2, the node 88 which is the right sub-tree of 78 is of height 1 and the node 50 which is in the left sub-tree of 78 is of height 3. The height of no other node has changed.

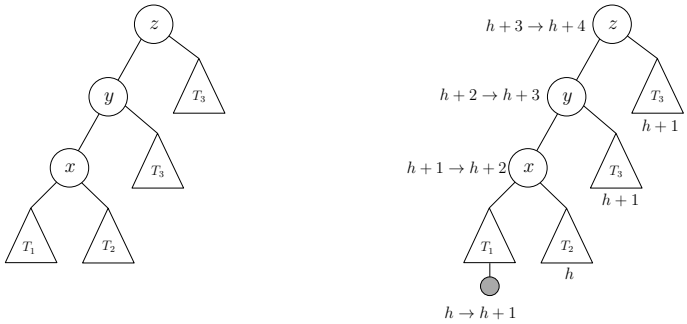
The node 78 will be z and x would be its grandchild on the path and the parent x would be y. So in the figure Fig. 1b, x is the node with key 62, y is the node with key 50 and z is the node with key 78. This is found by traveling up tree from the node 54 to find the first place where the imbalance happens. It is the node 78 and by convention it is marked as z. The grandchild of z is the node 62 so it is marked as x. The parent of x is the node 50 and it is marked as y. To rebalance the tree, the subtree rooted at z has to be rebalanced in particular. The rebalancing is done by performing a rotation. The tree shown in Fig. 1c is what will happen after the rebalancing is done. It can be seen that only the sub tree rooted at 78 i.e. the one containing these 6 nodes $48, 50, 54, 62, 78, 88$ is changed. These 6 nodes are organized in a manner so that the node 78 is not height imbalanced any more and neither the node 44 is height imbalanced.

Now let us understand this process of rearranging the subtree. Actually this is done by performing what is called a rotation. A rotation is the way of locally reorganizing a binary search tree. Let us consider the figure in Fig. 2a,

shown below. This could be a huge tree but only part of it is considered. The root of the tree is u, v is its child and T_1, T_2, and T_3 are some sub trees as shown. T_3 is the sub tree rooted at the right child of u, T_2 is the sub tree rooted at the right child of v and T_1 is the sub tree rooted at the left child of v. Because it is a binary search tree, all the keys in T_1 are less than the key in v. All the keys in T_2 are more than the key in v. Keys in T_2 are less than the key in u and keys in T_3 are more than the key in u i.e. $Keys(T_1) < Key(v) < Keys(T_2) < Key(u) < Keys(T_3)$. This follows from the property of binary search tree. After the rotation, v has become the parent of u. T_1 still remains left child of v, T_3 remains right child of v and as a result of the rotation T_2 changes the loyalty from v to u, i.e it becomes the left child of u. One important thing about the rotation is that the binary search tree property still holds. We still have $Keys(T_1) < Key(v) < Keys(T_2) < Key(u) < Keys(T_3)$. It is still a binary search tree but some local reorganization is performed on it and this will be very useful when insertion of a new node makes the AVL tree height imbalanced. Now let us see how is this rotation used to do insertion in an AVL tree. Suppose there is a huge AVL tree and a part of the tree is shown in Fig. 3a. Suppose an insertion happens in the subtree T_1 (Shown in Fig. 3b). As a result of this insertion suppose the tree becomes imbalanced z is first node in the path from the inserted node to the root which becomes height imbalanced, node x is the grand child of z and y child of z and at the same time parent of x. In the figure Fig. 3b y is the left child of z and x is the left child of y. This is one scenario but there could be more as y could either be left or right child of z and similarly x could also be either left or right child of y. So in general there would be four cases. The cases we are discussing here is symmetric to the scenario where y is the right child of z and x is the right child of y. Let the hight of T_1 is originally h and now as a node is inserted to T_1 its height changed to $h+1$. The height cannot increase by more than a one because just one node is added. Since the height T_1 increases by one, height of node x also increases by one that in turn increases the height of y. The height of z also increases by a one and that is what makes z height imbalanced. If there was no increase in height of y then z would not become imbalanced and height of y is increased because height of x is increased. Height of x is increased because height of T_1 is increased and height of T_1 let say increased from h to $h+1$. What can be said about the height of T_2 now? Since x is balanced even after the insertion (because z was the first node which was imbalanced). So x was balanced after insertion its height can be either h or $h+1$ or $h+2$. So which one of this three? Can it be $h+2$? if so, x was originally imbalanced which should not be as before insertion it was a part of AVL tree. So height of T_2 can not be $h+2$. Can it be $h+1$? If it is $h+1$ then height of x does not increase. If height of x does not increase there is no possibility that z becomes imbalanced. So height of T_2 can not also be $h+2$. Thus the height of T_2 has to be h. So now If height of T_2 is h and height of this T_1 has increased from h to $h+1$, then what about height of x? Height of x will be increased by one. Before the insertion it was $h+1$, now it becomes $h+2$. What about the height of y? Since y remains balanced and the new height of x is $h+2$. The height of T_3 is $h+3$ or $h+2$ or $h+1$. One of these 3, because the difference in heights can only be one. So it is one of these 3 but which one them? If it is $h+3$, y was originally imbalanced. Can it be $h+2$? If it is $h+2$ then height of y has not increased and this should not be the case as then there will be no possibility that z becomes imbalanced. So height of T_3 has

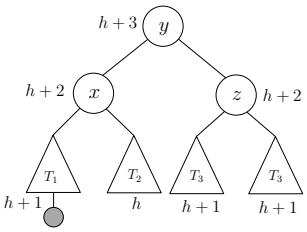
to be $h+1$. If height of T_3 is $h+1$, what would be the height of y? Originally it was $h+2$ because both the sub-tree of y were $h+1$. Now one subtree of y, the one rooted at x is of height $h+2$ and the other one, i.e T_3 is of height T_3, so the height of y now increased from $h+2$ to $h+3$. What about height of z? Note that z is imbalanced now. The new height of y is $h+3$. So what should the height of T_4 be? Initially z was balanced and initial height of y was $h+2$. Since z was balanced, height of T_4 is $h+1$ or $h+2$ or $h+3$. Since it is now unbalanced it cannot be $h+2$ or $h+3$, it has to be $h+1$. So the new height of z becomes $h+4$ and subtree rooted at z is imbalanced. We will perform rotation as discussed before so that subtree become balanced and height of the subtree again becomes $h+3$. The rotation is performed around the pair (y, z). Rotation is going to rotate the subtree so that y is now going to become the parent of z. Basically, the rotation moves y up so that it becomes the root of the subtree, z becomes its right child, the subtree rooted at x remains left subtree of y, and the subtree changes its loyalty from y to z i.e. it now becomes left child of z. The subtree after the rotation with heights of subtrees and nodes specified is shown in figure 3c. It is to be observed that the binary search tree properties are still maintained even after the rotation. So this is still a binary search tree, but now we want to argue that the height balance property is also restored. Let us see this for each node. Is the node x height balanced? Yes, because its left subtree (T_1 with the newly inserted node) is of height $h+1$ and the right subtree (T_2) is of height h. The hight difference at x is 1 and so it is height balanced. Is the node Z height balanced? Yes, because its left subtree (T_3) of height $h+1$ and the right subtree T_4 is of height $h+1$. The hight difference at z is 0 and so it is height balanced. Is the node Y height balanced? Yes, because its left subtree (rooted at x) of height $h+2$ and the right subtree (rooted at y) is of height $h+2$. The hight difference at z is 0 and so it is height balanced. So, every node in this subtree (i.e. subtree consisting of x, y, z, T_2, T_3, T_4 and T_1 with the inserted node) is height balanced now. Another important observation of this rotation is that the height of the subtree before and after rotation does not change, earlier it was $h+3$ (height of z) after the rotation it also remains $h+3$ (height of y). This fact is very interesting because do not have to go up further. Height of any ancestors of the new root (i.e. y) of the subtree would not change any more because whatever was the original height ($h+3$) of the subtree, the new height of this sub tree also remains $h+3$. So height of any of the ancestors of would remain the same as before. And so there will be no imbalance on them. Once this subtree is balanced by performing rotation the rest of tree automatically remain balanced. This kind of rotation is called single rotation. The reason it is called single rotation will be clear when we will discuss two step rotation which is called double rotation.

The case that we have discussed just now has a symmetric as mentioned earlier where x is the right child of y and y is the right child of z. The other case that we are going to discuss is where x is the right child of y which is the left child of z as shown in Fig. 4a. This has also has the symmetric case that is, y is the right child of z and x is the left child of y. Again this is completely symmetric and is left to the reader as an exercise. So let us repeat the argument that we had in case of single rotation. Let us once again assume that the insertion happens in T_2. As a result of the insertion the height $T2$ increased from h to $h+1$. Now let us argue about the height of other nodes. What about the height of T_3? Since x is balanced even after the insertion, the



(a) before insertion

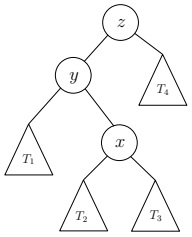
(b) after insertion



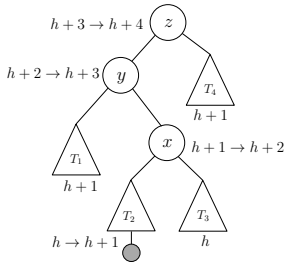
(c) single Rotation

Figure 3: Insertion in an AVL tree

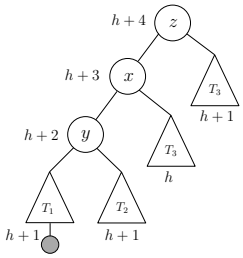
height of T_3 is either $h+2$ or $h+1$ or h. If it is $h+2$ then that means x was originally (before the insertion) imbalanced. If it is $h+1$ then that means the height of x is not increased. So height of T_3 has to be h. Note that the arguments are same as before. What would be the new height of x? Its original height was $h+1$ the new height would be $h+2$ as the height of T_3 has increased from h to $h+1$ and that in turn has increased the height of x. Now let us look at the height of T_1. Since y is still balanced then that means the height T_1 of is either $h+3$ or $h+2$ or $h+1$. If it is $h+3$ then that means y was originally imbalanced. If it is $h+2$ then the height of y has not increased. It has to be $h+1$, which means the height of y has increased from $h+2$ to $h+3$ which in turn means that since z has become imbalanced now, the height of T_4 has to be $h+1$. It implies that the original height of z was $h+3$ and now it becomes $h+4$ and as a result of this it becomes imbalanced. Exactly the same line of argument as before but now the rotations will be a bit different now. Among the nodes x, y and z, x is having the middle key. The key in z is the largest and the key in y is the smallest. Recall that in the previous rotation i.e. in single rotation, among x, y and z y was having the middle key, because they were all in a line, z was at the top, y was its left child and x was its left child. So, the key in y was less than z and the key in x was less than y. In that case, y ended up being the root after the rotation and everything became height balanced. The observation here is that the middle key was moved up to be the root. Here also something similar needs to be done but the middle key is now x. So, 2 step rotation is applied here that is why it is called a double rotation. First, x, y are rotated and as result the tree becomes as shown in Fig. 4c. The node x has moved up, y has moved down, T_1 remains the left child of y, T_3 remains the right sub tree of x but the sub tree T_2 switches loyalties from x to y i.e. T_2 now



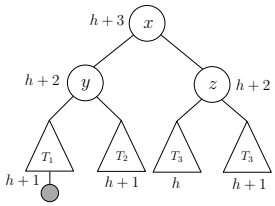
(a) before insertion



(b) after insertion



(c) Step 1 : rotation (x, y)



(d) Step 2 : rotation (x, z)

Figure 4: Insertion in an AVL tree, double rotation

becomes the right child of y. The figure 4c shows the height of each node after the first steps. Now the question is, is the tree thus obtained balanced? To get answer to this question let us look at each node. Is the node y balanced? The height of both the subtrees, T_1 and T_2 of y is $h + 1$, difference in their height is 0 and so y is balanced. Is the node x balanced? Actually now there is an imbalance at x itself because y has $h + 2$ and T_3 has height h and finally height of x is $h + 3$. There is an imbalance in z also and height of z would be $h + 4$ because height of x is $h + 3$ and height of T_4 is $h + 1$. So this rotation has not done the job yet. We need to perform another rotation, rotation between x and z. This rotation i.e. the second step is shown in Fig. 4d. As shown x goes up to be the root, z comes down and x becomes the parent of z. T_4 was the right sub tree of z, it remains as the right subtree of z. The subtree rooted at y had T_1 and T_s as its left and right subtree respectively, they remain as they are. The subtree T_3, which was the right sub tree of x now becomes the left sub tree of z. Now let us compute heights. Height of y is $h + 2$, height of z is $h + 2$ and height of x is $h + 3$. All the nodes x, y and z also become height balanced most importantly the height of this sub tree remains the same as the original sub tree which is $h + 3$. So the final tree has the same height as the original tree and hence we need not go further up the tree. The double rotation also ended up doing the same thing, the middle key ended up being at the top. So basically what rotation does is it split the tree uniformly. The height imbalance was happening because x was the middle key as the insertion happens in x it was coming way down. When the tree is split uniformly the heights reduced and there is a height balance.

Now, how much time does the double rotation take? Just as a quick recap, to find the node to be deleted we need at most $\log n$ number of comparison

and once it is found x, y and z can again be located in $\log n$ time. To perform the rotation few pointers need to be updated which are actually constant in numbers. So the total time for rotation is bounded by $O(\log n)$ time.

2 Deletion

Let us now look at the deletion procedure in an AVL tree. Recall that in a binary search tree when we delete a node, we have 3 cases. Either we

- delete a leaf node or
- delete a node which has one child or
- delete a node which has two children.

To delete a node which has 2 children, the successor of that node is found out and the content of the successor is copied to that node and the successor is deleted. The actual node that is deleted is the successor of the node and the successor node has only one child or no children. Why does the successor has only one child? Because it does not have the left child, because if it had the left child then that left child would have been the successor, however, the successor may have right child. So, the successor has only one child or it has no children. So the actual node that is deleted is either a leaf node or a node with only one child. It is to be observed that in an AVL tree if a node has one child then that should be a leaf node otherwise there would be a problem of height imbalance. So, when a node is deleted from an AVL tree, either a leaf is deleted or the parent of a leaf is deleted. Now when a node with only one child is deleted, then the content of the leaf is copied to the parent and then the child is deleted, which means that essentially a leaf is deleted. So it can always be thought that when a node in an AVL tree is deleted, ultimately a leaf node gets deleted. Let us say w is the node that is to be deleted. When w is deleted, once again the the height of the ancestors of w could reduce and one or more of these ancestors may be unbalanced. We need to define the nodes x, y and z as earlier but this time they are defined slightly differently. Let us suppose z is the first unbalanced node encountered while travelling up the tree from w. Now y is not the child of z on the path but y is defined as child of z with larger height and x is the child of y with larger height. Once again rotations are to be performed to restore the height balance of the sub tree rooted at z. In the case of insertion once the rotation is done, the ancestors node were not examined for height imbalance. Everything was taken care of, as the height of the sub-tree before and after remains the same. However, in deletion it may not happen we might have to continue up to the root.