21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2022**

Lecture 1: AVL tree

Lecturer: Deshi Ye *Scribe: D. Ye*

1.1 Overview

Balanced binary searchs (BST) are used quite often for many applications. In this lecture, we will study how BST works.

Definition 1.1 height:

The **height of a node** *is the length of the longest downward path to a leaf from that node.*

The **height of a tree** *is the longest path from the tree's root to one of its' leave. The height of an empty tree is defined to be* -1*.*

Operations of binary search tree are: Search, Min, Max, Select, Predecessor, Successor, Insert, Delete. In general, search tree operations run in time proportional to the height of the tree. In a binary search tree with n nodes, the height can be $\log n$ (if the tree is perfectly balanced) to $n-1$ (if the nodes form a single chain). A **balanced search tree** is a search tree to keep its height at most of $O(\log n)$ after a sequence of insertions and deletions.

When to use Balanced Search Tree? If you use a dynamically changing (i.e., support the insert and delete operations) set of an ordered data set, the BST is usually a good choice.

In this lecture, we are going to discuss the technique of the BST that can maintain the height at most of $O(\log n)$ for insertions and deletions.

1.2 AVL tree

The AVL tree was invented by Adelson-Velskii and Landis [1] in 1962. An AVL tree is a height-balanced search tree, which is defined as below.

Definition 1.2 height-balanced tree*:*

An empty binary tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is **height-balanced** *iff*

(1) T_L and T_R are height-balanced, and

(2) $|h_L - h_R| \leq 1$, where h_L and h_R are the heights of T_L and T_R, respectively.

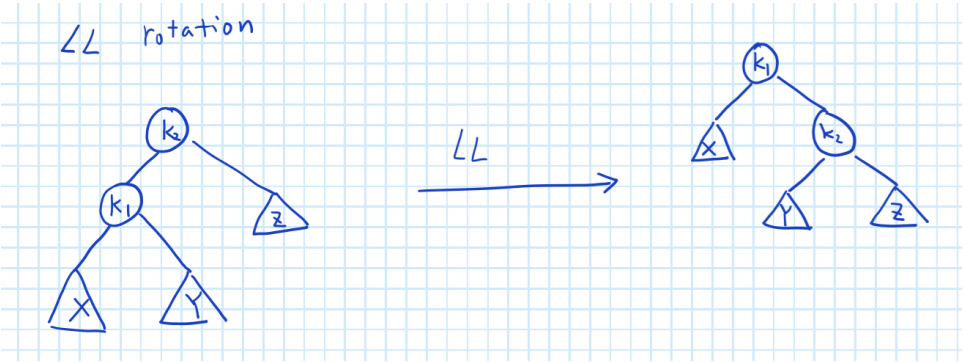

Figure 1.1: $RR(k_2)$: Right rotation on the subtree rooted at k_2.

Remark: the subtree root by every node in the AVL tree is a height-balanced tree. To implement an AVL tree, we need to keep the height information in each node. To save the space of the AVL tree, we only store the height difference between its left child and its right child, which is defined as the balance factor.

Definition 1.3 Balance factor:

The balance factor of the node x, $BF(x) = h_L - h_R$, where h_L is the height of x' left child and h_R is the height of x's right child.

In an AVL tree T, for any node x in T, we have $BF(x) = -1, 0, 1$.

Note that give an AVL tree, insert a new node or delete a node, balance factors in this tree might change, and some nodes may not be the value of $-1, 0, 1$ anymore. We could do some extra work such as moving some nodes to other positions. One question that comes to our mind is how much work do we need to maintain it is an AVL tree after inserting a node or deleting a node?

In the following, we will introduce an important operation, named rotation, which plays a key role in rebalancing the binary search tree.

1.2.1 Rotations

Let k_2 be the root of a subtree. Denote $RR(k_2)$ to be the function of right rotation based on k_2. A right rotation applies when the child k_1 is the left child of its parent k_2 (and so k_1 has a smaller key than k_2); after the rotation, k_2 is the right child of k_1, and k_1 becomes the new root.

See Fig. 1.1 for an example of right rotation, in which X denotes the nodes with the key value smaller than k_1, and Y denotes the nodes with key values between k_1 and k_2, and Z denotes the key values larger than k_2.

Remark: After rotation $RR(k_2)$, the new tree rooted at k_1 is still a binary search tree.

When k_1 is the right child of k_2, a left rotation makes k_2 the left child of k_1. We denote $LL(k_2)$ to be the function of left rotation based k_2; after the rotation, k_2 is the left child of k_1.

Proposition 1.4 • *A single rotation will preserve the search tree property.*

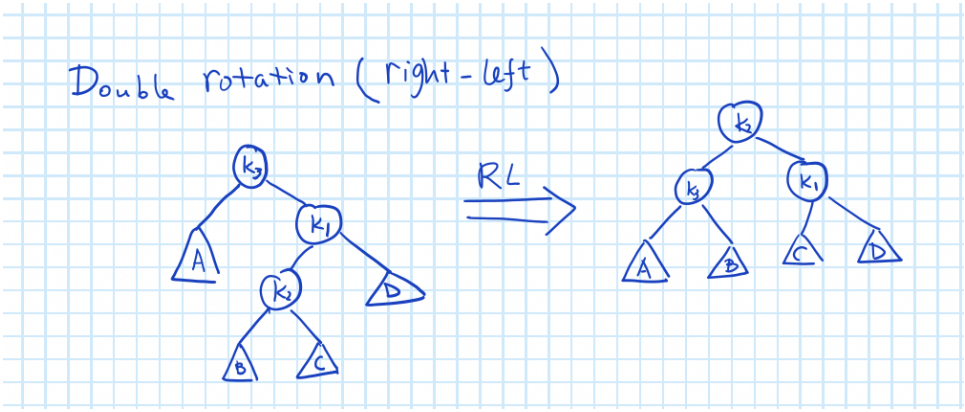

Figure 1.2: $RL(k_3)$: Double rotation on the subtree rooted at k_3.

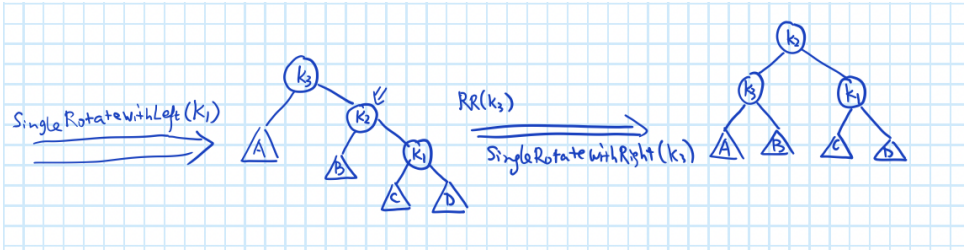

Figure 1.3: A double rotation viewed as two single rotations.

- *Because a rotation only reconnect a few pointers, it can be implemented with a constant number of operations.*

Double rotation The double rotation of first right and then left on the subtree rooted at k_3 is illustrated in Fig. 1.2, and the new root is k_2.

A double rotation is two single rotations, which is shown in Fig. 1.3.

1.2.2 Insert in AVL

Suppose we have an AVL tree (the height is already balanced), then we insert a new node as the normal insertion of BST. This new insertion might be out of balance, and we need to rebalance it by the following procedure.

How to rebalance an AVL tree after inserting a new node? Starting at the inserted node, traverse up the tree to find the first unbalanced node x, otherwise, it is done. Take two steps downward in the tree, starting at x, each time toward the higher of the left and right subtrees.

- If the two steps are in the same direction, either both to the left or both to the right, then it is a **zig-zig**. A single rotation is called for.

- If the two steps are in opposite directions (one to the left and the other to the right), then it is a **zig-zag**, and a double rotation is called for.
- Update the height of each node along the path of the inserted node to the root.

Proposition 1.5
- *Inserting a node will change the height (increase by $\leq$ 1) of nodes on the path from the root to the newly inserted node only.*
- *Only nodes on the path from the root to the newly inserted node can become height imbalanced*

Correctness of the rebalancing an AVL tree The correctness can be proved in four cases. In the following, we only prove two cases, while the other two cases are similar. The correctness is done if we show the height of the affected subtree after the insertion is the same as before.

Proof: We distinguish two cases: zig-zig, and zig-zag.

It is a zig-zig case. See Fig. 1.1 for an example, the new inserted node w is in k_1's left subtree, and k_2 is the first unbalanced node, i.e. $BF(k_2) = 2$. Thus before insertation $BF(k_2) = 1$, otherwise, it is impossible to have $BF(k_2) = 2$.

Let h be the height of the subtree Z, as it is an AVL tree, we know the height of tri-nodes X, Y is also h. Then the height of the tree root at k_2 is $h + 2$.

After insertation of w, the height of $X \bigcup \{w\}$ is now $h + 1$. Then the height of subtree k_1 is $h + 2$, the height of tree root at k_2 is $h + 3$.

After the single right-rotation, we obtain the right sub-figure in Fig. 1.1.

The height of the tree root at k_1 is $h + 2$.

We have proved that after a single rotation, the height of the affected subtree after the insertion is the same as before.

It is a zig-zag case. Let $h(A) = h$, $h(D) = h$, $h(B) = h - 1$, $h(C) = h - 1$. This is an AVL tree, and $h(k_3) = h + 2$.

Now, insert a new node w in B or C, w.l.o.g, let $h(B) = h$. Then, $h(k_3) = h + 3$, it is the first unbalanced node.

After the double rotation (right-left) as illustrated in Fig. 1.2, $h(k_3) = h(k_1) = h + 1$, and $h(k_3) = h + 2$. The affected subtree achieves the same height as the one before insertion. ■

1.2.3 Running time of an Insertation

Let n_h be the minimum number of nodes in a height-balanced tree of height h. Now we assume that the tree T root at A has height h. In an AVL tree, then the height of A's left and right child differs one, otherwise, it is not the minimum number of nodes with the height of h for the tree T. Without loss of generality, we assume its left child has height $h-1$, and its right child has height $h-2$. So we obtain the following equation:

$$n_h = n_{h-1} + n_{h-2} + 1 \tag{1.1}$$

One can check that $n_h = F_{h+2} - 1$, where F_i is the i-th Fibonacci numbers and $F_0 = 0, F_1 = 1$. Note that $F_i = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^i$, so we have $n_h = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{h+2} - 1$, and $h = O(\ln n)$.

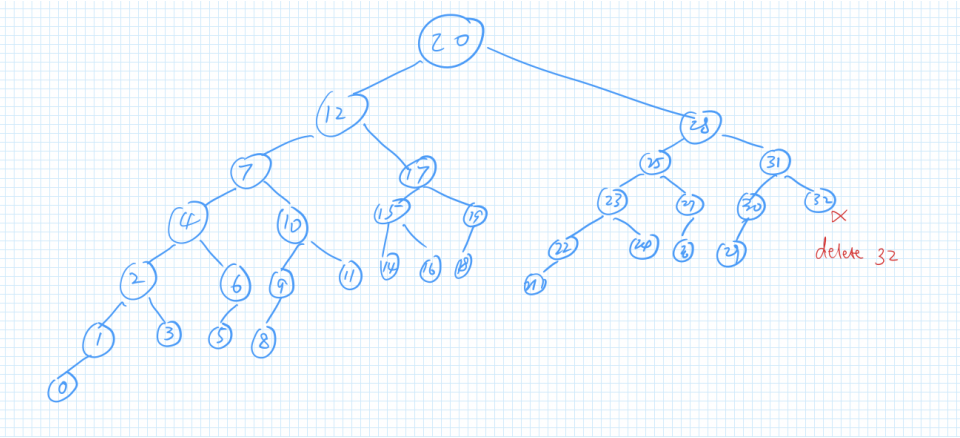

Figure 1.4: More than one rotation to keep the balance by deleting the node 32

Theorem 1.6 *In an AVL tree with n nodes, insert a new node and rebalance the tree to keep the property of AVL, the total cost is $O(\ln n)$.*

Proof: We have already proven that the height of an AVL tree with n nodes is $O(\ln n)$. Hence, the insertion operation costs $O(\ln n)$. During the rebalancing, it spends $O(\ln n)$ to find the first unbalanced nodes, and then a single rotation or a double rotation is sufficient to keep the balance. Moreover, inserting a node will change the height (increase by ≤ 1) of nodes on the path from the root to the newly inserted node only. Thus, we require $O(\ln n)$ to maintain the height information (or balance factor) of the path from the root to the newly inserted node. In all, it costs $O(\ln n)$ for the AVL tree to handle a new insertion. ■

1.2.4 Deletion of AVL

AVL tree can also keep the balance after deleting a node. Firstly, we do the operation Delete the same as it does for the binary search tree. Then we rebalance the tree along with the deleted node to the root by rotations, a single rotation for the zig-zig case, and the double rotation for the zig-zag case.

Theorem 1.7 *In an AVL tree with n nodes, insert a new node and rebalance the tree to keep the property of AVL, the total cost is $O(\ln n)$. Moreover, we might require $\Omega(\ln n)$ rotations to fix the balance after a deletion.*

Proof: Delete as BST tree costs $O(\ln n)$ time. Rebalancing along the path from the deleted node to the root is also $O(\ln n)$. But, we might need $O(\ln n)$ rotations for the delete, unlike the insertion case that requires only one single rotation or one double rotation.

■

Example in Fig. 1.4: We have a 32-node AVL tree. It requires more than one rotation to keep the balance by deleting the node 32.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.