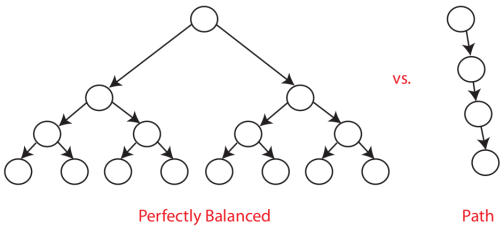
DSA Tutorial - AVL Trees

March 28, 2018

1 AVL Trees - Adel'son-Vel'skii & Landis 1962

BSTs offer wide functionality: search, min, max, successor, predecessor (queries) and insert, delete (updates). The runtime of all these operations was $O(h)$ where h represents the height of the BST. However in BSTs, h can conveniently be $O(\log n)$ in some situations and $O(n)$ in other. In the former case, the BST is said to be 'balanced'. AVL trees are binary search trees that balance themselves whenever an element is inserted or deleted, thus maintaining an $O(\log n)$ runtime for all the operations mentioned above.

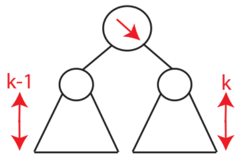


It is not easy to think of keeping the height as $O(\log n)$. It can be easier to think of balance in the following way: *AVL trees require that for all nodes the heights of their left and right children differ at most by 1* (keeping the left and right subtrees of more or less the same height). This requires us to define the height of a node: The height of a node is the length of the longest path from that node to a leaf. It can be shown that maintaining this invariant ensures that the height of the tree is always $O(\log n)$:

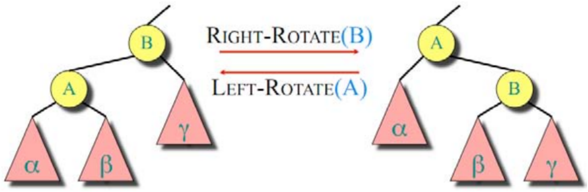
$$|\text{height of left child } (h_L) - \text{height of right child } (h_R)| \leq 1 \text{ for all nodes}$$

2 Maintaining the AVL property

Let us call a node 'right-heavy' if $h_R - h_L = 1$ and 'doubly right-heavy' if $h_R - h_L = 2$. Similarly we can define 'left-heavy' and 'doubly left-heavy' nodes. Thus, nodes that are doubly right-heavy or doubly left-heavy do not satisfy the AVL invariant property and should be fixed. These nodes are fixed via rotations. (The need to fix nodes arises when we insert a new node into an AVL tree. Insertion of a new node changes the heights of all its ancestors. We start with the newly inserted node, and fix the AVL invariant property as we go up.)



A rotation is simply a restructuring of BST nodes that maintains their order. This illustrates a right and left rotation:



Consider a doubly right-heavy node, x. Consider the right child of this node, y. Now y can be one of three things: right-heavy, left-heavy, or balanced. Convince yourself that the following operations successfully balance x.

- When y is right-heavy or balanced: **Left-Rotate(x)**
- When y is left-heavy: **Right-Rotate(y)** and **Left-Rotate(x)**

We can similarly look at the case when x that are doubly left-heavy.

- When y is left-heavy or balanced: **Right-Rotate(x)**
- When y is right-heavy: **Left-Rotate(y)** and **Right-Rotate(x)**

3 Questions

1. **Insertion**: Insert the following nodes into an AVL tree - 10, 20, 30, 40, 50, 60 (Perform the normal BST insert. If the newly inserted node is a leaf, it is trivially balanced. If not, update its height and check for the AVL invariant property. Convince yourself that the height of any node is updated as follows: $h_n = \max(h_L, h_R) + 1$, where h_L and h_R are heights of n's left and right children.)
2. **Deletion**: Insert 9, 5, 10, 0, 6, 11, -1, 1, 2. Delete 10 from the tree. (Perform normal BST delete. Heights of its ancestors would be affected. Check for the AVL invariant property.)
3. **AVL Sort**: We know that performing an inorder traversal on a BST gives us a sorted list of key values. An inorder traversal is an $O(n)$ operation, whereas insertion was $O(h)$ in a BST. Now with AVL trees, insertion is guaranteed to be $O(\log n)$. Thus, inserting nodes into an AVL tree and performing an inorder traversal on them yields yet another $O(n \log n)$ sorting algorithm. What are the other $O(n \log n)$ sorting algorithms we know, and how does this compare?
4. **AVL/BST with duplicate keys**: The basic idea is to augment a 'count' field in the node structure, and whenever a duplicate is inserted, increase the count of the node already present in the tree: **GeeksForGeeks page**.

4 References

- **MIT 6.006 lecture note on AVL trees**