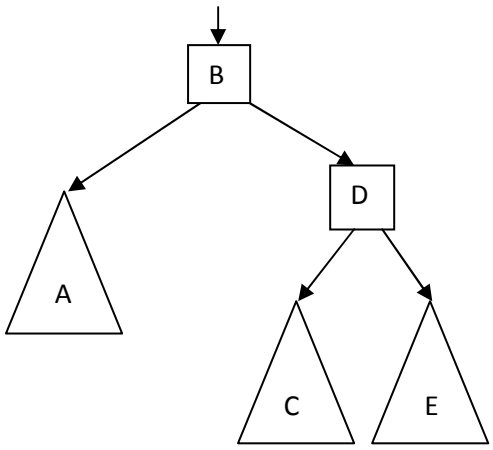
Rebalancing a tree

A tree is *balanced* if the difference in height of the left and right sub-trees of every node is less than or equal to 1. The more unequal the distribution the more unbalanced the tree is. Balanced trees will produce the most efficient search times.

When we rebalance a tree we are attempting to reduce the height difference between the left and right sub-trees of every node in the tree. To do this we need some operations to restructure trees.

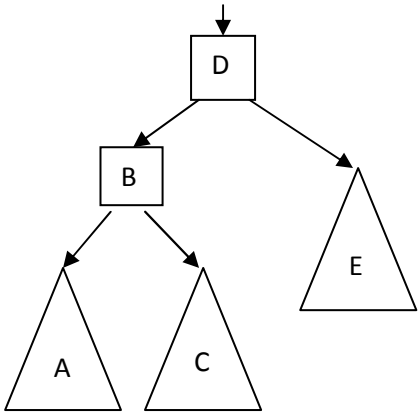
Rotate Operations

Consider the following binary tree



The tree has two nodes B and D, with the nodes pointing to sub-trees A, C and E. Note that the ordering of the letters shows us the tree is properly ordered.

An operation to *rotate* the tree anti-clockwise would result in

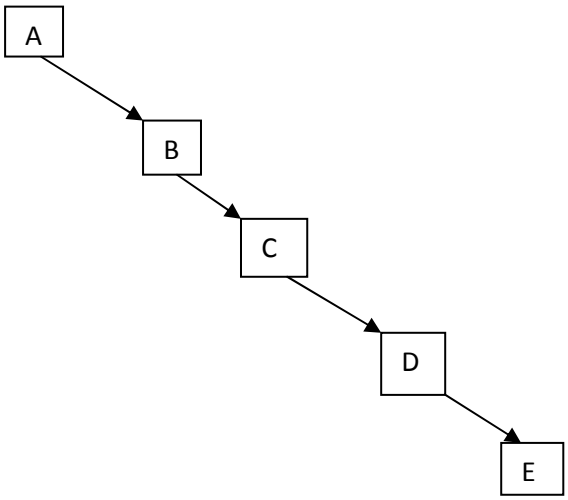


Note that the tree is still properly ordered. Rotating the tree clockwise would bring us back to the original tree.

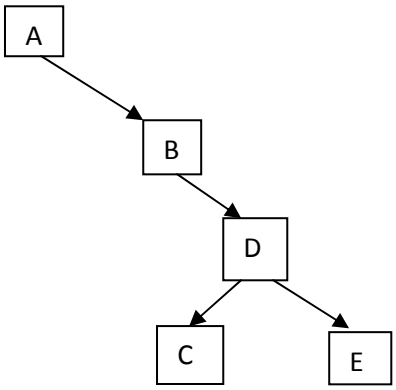
We can use these rotate operations to try and restructure a tree so it is more balanced.

Example of Rebalancing

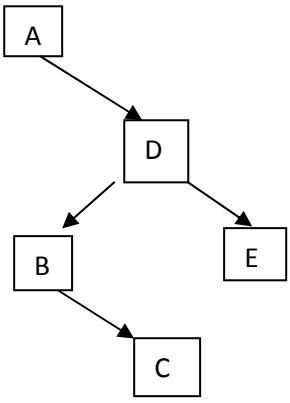
Consider the following tree



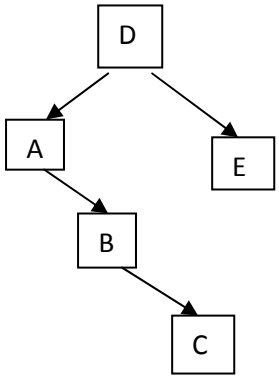
As you can see, the tree is maximally unbalanced. The difference in height between the left and right sub-trees of nodes A, B and C is greater than 1. If we apply anti-clockwise rotation to node C we get.



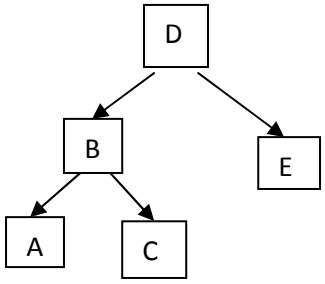
The tree is a bit better balanced as only nodes A and B are now unbalanced. Apply anti-clockwise rotation to node B and we get.



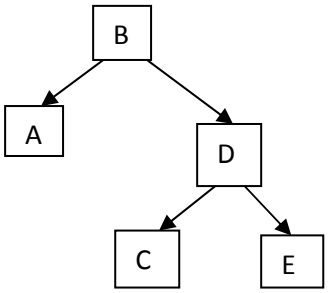
Only node A is now unbalanced. Apply anti-clockwise rotation to node A and we get



Nodes A and D are now unbalanced. However D is less unbalanced than A was. If we apply anti-clockwise rotation to node A we get



The tree is now balanced. No set of rotations will produce a better balanced tree. For example, if we did clockwise rotation to node D we would get



Which is no better than what we had before.