

AVLTreeRotateRight(tree,node D)

```

AVLTreeUpdateHeight(node) {
    leftHeight = -1
    if (node->left != null)
        leftHeight = node->left->height
    rightHeight = -1
    if (node->right != null)
        rightHeight = node->right->height
    node->height = max(leftHeight, rightHeight) + 1
}
    
```

```

AVLTreeSetChild(parent, whichChild, child) {
    if (whichChild != "left" && whichChild != "right")
        return false
    if (whichChild == "left")
        parent->left = child
    else
        parent->right = child
    if (child != null)
        child->parent = parent
    AVLTreeUpdateHeight(parent)
    return true
}
    
```

update the height of the AVL Tree

this sets B as the left child of P

this sets P as the parent of B

```

AVLTreeReplaceChild(parent, currentChild, newChild) {
    if (parent->left == currentChild)
        return AVLTreeSetChild(parent, "left", newChild)
    else if (parent->right == currentChild)
        return AVLTreeSetChild(parent, "right", newChild)
    return false
}
    
```

```

AVLTreeRotateRight(tree, node) {
    CleftRightChild = node->left->right
    if (node->parent != null)
        AVLTreeReplaceChild(node->parent, node, node->left)
    else { // node is root
        tree->root = node->left
        tree->root->parent = null
    }
    AVLTreeSetChild(node->left, "right", node)
    AVLTreeSetChild(node, "left", leftRightChild)
}
    
```

```

AVLTreeUpdateHeight(node) {
    leftHeight = -1
    if (node->left != null)
        leftHeight = node->left->height
    rightHeight = -1
    if (node->right != null)
        rightHeight = node->right->height
    node->height = max(leftHeight, rightHeight) + 1
}
    
```

Sets the new height of node B

the height of D

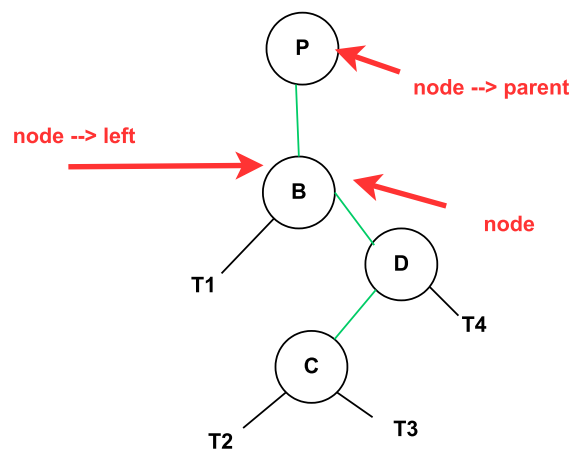
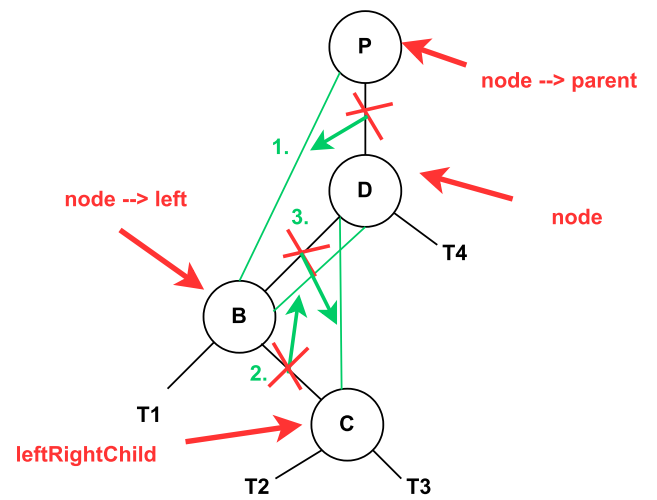
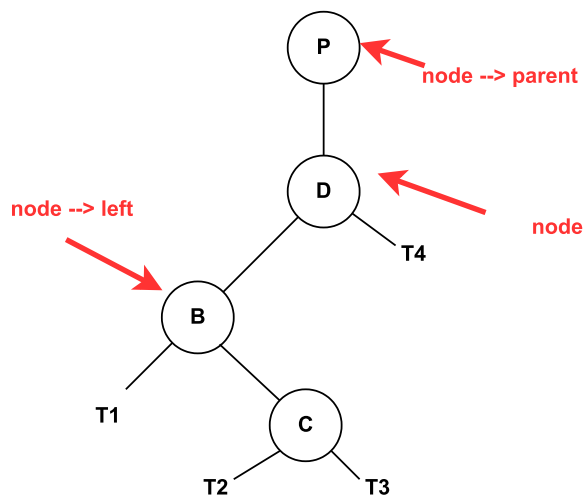
```

AVLTreeSetChild(parent, whichChild, child) {
    if (whichChild != "left" && whichChild != "right")
        return false
    if (whichChild == "left")
        parent->left = child
    else
        parent->right = child
    if (child != null)
        child->parent = parent
    AVLTreeUpdateHeight(parent)
    return true
}
    
```

this sets D as the right child of B

this sets B as the parent of D

AVLTreeRotateRight(tree,node D)



Insertion with Rebalancing

- An AVL tree insertion involves
 - Searching for the insert location
 - Inserting the new node
 - Updating balance factors
 - Rebalancing
- Balance factor updates are only needed on the nodes ascending along the path from the inserted node up to the root

MyNotes:

- For n nodes, an AVL tree has a height equal to the floor($\log(n)$)
- For n nodes, an AVL tree has height $O(\log(n))$
- For AVL insert operations the complexity is $O(\log(n))$

Red-black tree: A balanced tree

- Every node is colored either red or black
- The root node is black
- A red node's children cannot be red
- A null child is considered to be a black leaf node
- All paths from a node to any null leaf descendant node must have the same number of black nodes