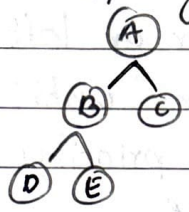
22/7/23

[TREES]

Tree - It is a non-linear datastructure.

Glossary:

Root - A node with no parents.
Edge - Link from parent to child.
Leaf node - A node with no children.
Siblings - Children of same parent.
Ancestor - Analogy of grandchildren



A, B is ancestor of D & E

Level - Set of all nodes at a given depth.
Above tree, at level 2, set of nodes (D, E)
Node depth - Length of path from root to the node.
Node height - Length of path from that node to deepest node

```
func (binaryTree *BinaryTree) NodeDepth(node *BTNode) int {
    root := binaryTree.root
    if root == nil {
        return 0
    } else {
        queue := &Queue{}
        queue.CreateNew()
        queue.Enqueue(root)
        queue.Enqueue(nil) // end of level
        level := 0
    for !queue.IsEmpty() {
        bnode := queue.DeQueue()

        if bnode == nil {
            if !queue.IsEmpty() {
            }       queue.Enqueue(nil)
```

```
        level++ }
        if node.data == bnode.data {
            return level+1
        }
    }else {
        if bnode.left != nil { queue.Enqueue(bnode.left) }
        if bnode.right != nil { queue.Enqueue (bnode.right) }
    }
}//end of for
}//end of function
```

```
func NodeHeight () {
    It has only one change, find nodelevel, find
deepest node level, return deepest node level - node level.+1
}
```

Skew Trees - Every node has one child.

Binary Trees

A tree is binary if it has 0, 1, or 2 children.

Strict BT → Exactly 2 children or No children
Full BT → -11- -11- -11- & all leaf nodes at same level.
Complete BT → All leaf nodes are at height 'h' or 'h-1' & w/o any missing number from the sequence.

Properties

Full BT - No. of nodes $\Rightarrow 2^{h+1} - 1$
- Leaf nodes $\Rightarrow 2^h$

Complete BT - No. of nodes $\Rightarrow 2^h$ (minimum)
$2^{h+1} - 1$ (maximum)

Applications
Expression trees used in compilers
Huffman coding trees used in data compression algorithm.
BST supports Insertion, Search & Deletion in O(log n)

BT Traversals

PreOrder	D	L	R	D - Visit root
InOrder	L	D	R	L - Left subtree
PostOrder	L	R	D	R - Right ↓

LevelOrder
① Visit the root
② While traversing level 'l', keep all elements at level 'l+1' in the queue.
③ Go to next level & visit all nodes at that level.
④ Repeat this until all levels are completed.

Problem 26 - LCA of two nodes in a binary tree.
Algorithm
① Start from the root of the binary tree.
② If the root is null or equal to either of the two nodes, return the root as the LCA.
③ Recursively search for LCA in left & right subtrees.
④ If both nodes are found in different subtrees, return current root as LCA.
⑤ If both nodes are found in the same subtree, continue search in that subtree.
⑥ If one or both nodes are not found return null.

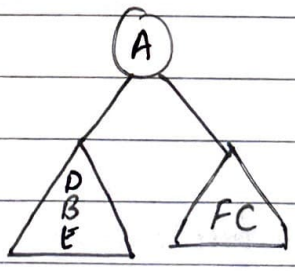
```
func LAc (root *BTNode, firstNode *BTNode, secondNode *BTNode)
        *BTNode {
if root == nil || root == firstNode || root == secondNode {
        return root
}
left = LCA(root.left, firstNode, secondNode)
right = LCA(root.right, firstNode, secondNode)

if left!=nil && right!=nil {
    return root          // Found in different subtrees
}
if left != nil { return left } else { return right }
}
```

Problem 27: Algorithm for constructing a binary tree from Inorder & Preorder traversals.

Inorder : D B E A F C

Preorder : A B D E C F

- Preorder's leftmost element is the root
- Inorder's elements falling to left of A are part of left subtree & elements on right of A are part of right subtree



Algorithm:

1) Select an element from Preorder. Increment a preOrderIndex variable to pick next element in next recursive call.
2) Create a new tree node (newNode) with the data as selected element.
3) Find the selected elements index in Inorder. Let it be inOrderIndex

4) Call BuildBinaryTree for elements before inOrderIndex & make the built tree as left subtree of newNode.

5) Call BuildBinaryTree for elements after inOrderIndex & make the built tree as right subtree of newNode.

6) return newNode

```
func BuildBinaryTree(preorder[] rune, inorder[] rune,
                     preOrderIndex, inOrderIndex int) *BTNode {
    element := preorder[preOrderIndex]
    preOrderIndex++
    var inOrderIndex
    for i, v := range &inorder {
        if element == v {
            inOrderIndex = i
            break;
    }
    newNode := &BTNode{}
    newNode.data = element
    newNode.left = BuildBinaryTree(preorder, inorder[:inOrderIndex],
                                   preOrderIndex, inOrderIndex)
    newNode.right = BuildBinaryTree(preorder, inorder[inOrderIndex+1],
                                    preOrderIndex, inOrderIndex)

    return newNode

}    // Check textbook while coding
```

Problem 28: Given two traversal sequences, can we construct the binary tree uniquely?

→ If one of the traversals is Inorder then the tree can be be constructed uniquely otherwise not.

Following combinations uniquely identify a tree :

- Inorder & Preorder
- Inorder & Postorder
- Inorder & Level-order

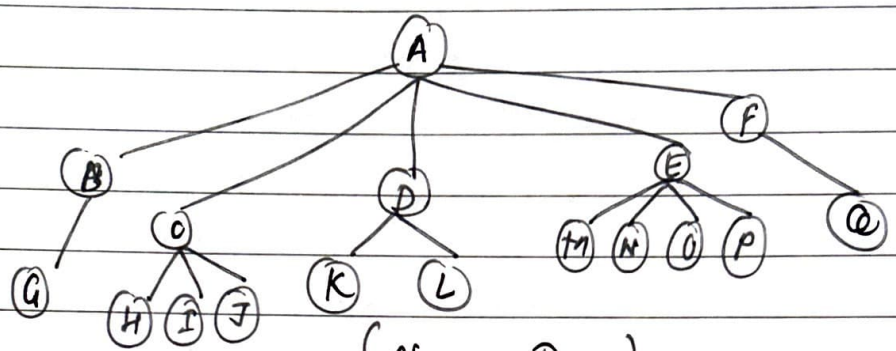
Problem 29 : Algorithm for printing all the ancestor of a node in BT.

Generic Trees (N-any Trees)

A node having 'n' number of childrens.

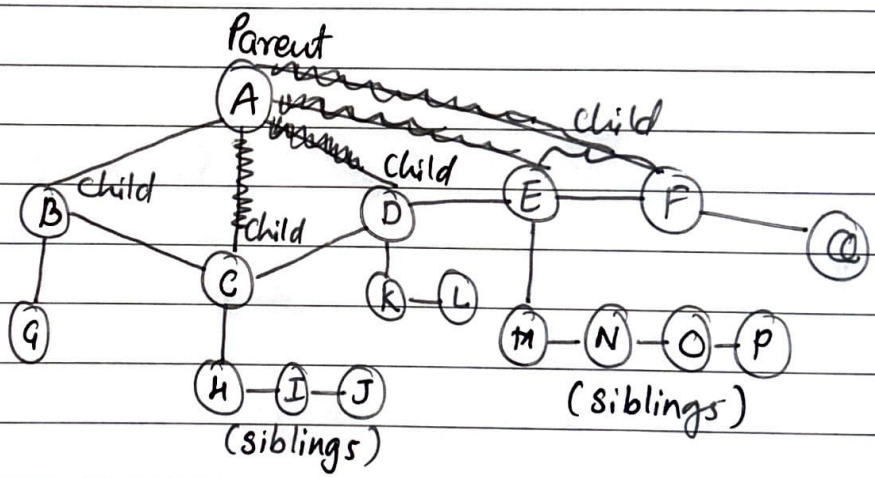
N-any trees are represented using first child/next sibling representation.

Parent node is connected to first child & first child then connects to all its siblings.



(N-any Tree)

Representation:



Structure Definition :

```
type struct NaryTreeNode {
        data int
        firstChild *NaryTreeNode
        nextSibling *NaryTreeNode
}
```

All generic trees with a first child/next sibling representation can be treated as binary trees.

All problems for binary trees are applicable for generic trees also, instead of left and right pointers, just use firstChild & nextSibling.

Problem (36) Given a tree, give an algorithm for finding sum of all elements of the tree.

Nodes →	A	B	C	D	E	F	G	H	I	J	M	N	O	Q
data	A	B	C	D	E	F	G	H	I	J	M	N	O	Q
firstChild	B	G	H	K	M	Q	X	X	X	X	X	X	X	X
nextSibling	X	C	X	X	X	X	X	I	J	X	N	O	P	X

How to construct n-ary tree?

```
FindSum
func (root *NaryTree) int {
    if root == nil { return 0 }
    return findSumHelper(root)
}

func findSumHelper(root *NNode
return
    root.data +
    findSumH(root.firstChild) +
    findSumH(root.nextSibli)
}
```

Threaded Binary Tree Traversals (Stack / Queue-less Traversals)

↳ Issues with regular Binary Tree Traversals:

→ Storage space required for stack/queue is large

→ Majority of pointers in binary trees are wasted (NULL) pointers.

→ It is difficult to find successor node (preorder, inorder and postorder successors) for a given node.

Threaded Binary Tree Classification:

Left Threaded – Predecessor information in NULL left pointer

Right Threaded – ~~Predecessor~~ Successor ⊣ — ⊣ rigut pointer

Full Threaded – Predecessor in Null left pointer & Successor in Null rigut pointer

Types of threaded binary tree:

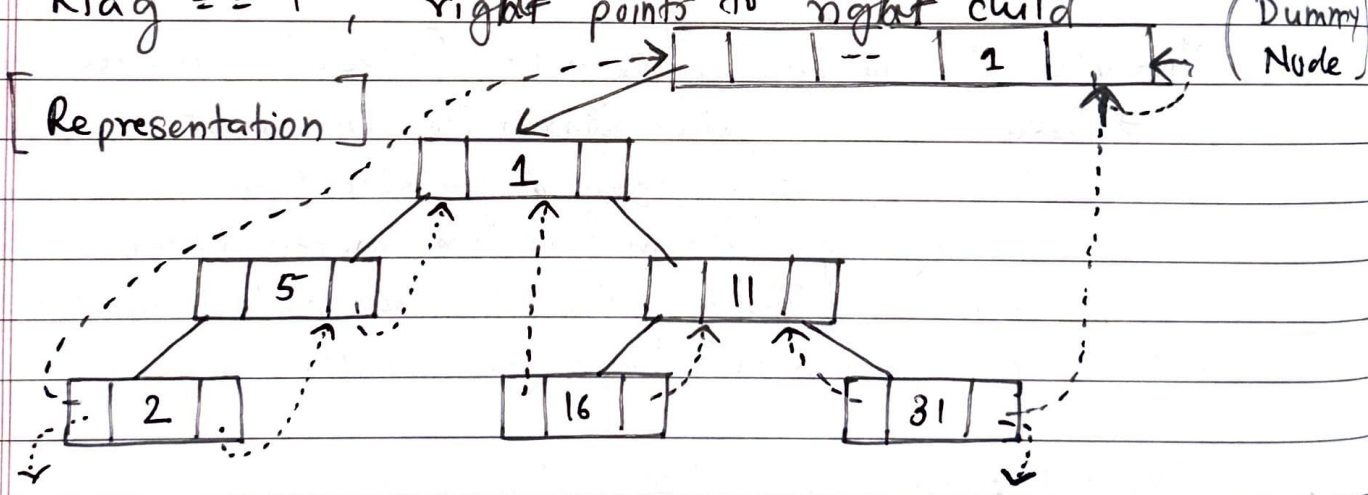
Preorder TBT –	Left pointer → Preorder predecessor	Right → successor
Inorder TBT –	⊣ —	Inorder ⊣ — / ⊣ — → ⊣
Postorder TBT –	⊣	Postorder ⊣ / ⊣ → ⊣

Threaded Binary Tree structure:

Left	LTag	data	RTag	Right

```
type struct ThreadedBinaryTreeNode {
        left   *ThreadedBinaryTreeNode
        LTag   int
        data   int
        RTag   int
        right  *ThreadedBinaryTreeNode
}
```

LTag == 0 , left points to in-order predecessor
LTag == 1 , left points to left child
RTag == 0 , right points to in-order successor
RTag == 1 , right points to right child



What should leftmost & rightmost pointers point to?

Use a special node 'Dummy' whose Right Tag is 1 & its right child points to itself.

Make the leftmost pointer point to left of dummy node & rightmost pointer point to right of dummy node.

Finding Inorder Successor in Inorder Threaded Binary Tree

- If 'P' has no right subtree, then return the right child of 'P'.
  i.e. RTag == 0
- If 'P' has right subtree, then return the left of the nearest node whose left subtree contain 'P'.
  i.e.

```
while (LTag == 1) // indicates having left child
    Position = Position → left
return Position
```

```
func (tBinaryTree *TBT) InorderSuccessor (p *TBTN) *TBTN {
    position := &TBTN{}
    if p.RTag == 0 {
        return p.right
    } else {
        position = p.right
        while (position.LTag == 1) {
            position = position.left
        return position
    }
}
```

Inorder Traversal - Inorder Threaded Binary Tree

```
func (tBT *TBT) InOrderTraversal (root *TBTN) {
    p := tBT.InorderSuccessor (root)
    while (p != root) {
        p = tBT.InorderSuccessor (p)
        fmt.Println (p.data)
    }
}
```

Finding Preorder Successor in Inorder Threaded BT

↳ If P has a left subtree, return 'left' child

↳ If P has no left subtree, return the right child of the nearest node whose right subtree contains 'P.

```
func (tBT *TBT) PreorderSuccessor (node *TBTN) *TBTN {
    if node.LTag == 1 {
        return node.left
    } else {
        position = node.left
        while (position.RTag == 1)
            position = node.right
        return position
    }
}
```

PreOrderTraversal – Same as InOrderTraversal, only difference is of calling PreOrderSuccessor ()

Insertion of Nodes in InOrder TBT

left
data
RTag
LTag
right

left
data
RTag
LTag
right

Expression Trees:

A tree representing an expression is called an expression tree.

Leaf nodes - Operands

Non-leaf nodes - Operators

★ Algorithm: [Building Expression Tree from Postfix expression]

1) Assume postfix expression is given. [A B C * D + D /]
2) Read one symbol at a time.
3) If symbol is an operand, create a tree node & push a pointer to it onto a stack.
4) If the symbol is an operator, pop pointers to two tree nodes T_1 & T_2 and form a new tree whose root is the operator & whose left & right children point to T_2 & T_1 respectively. A pointer to this new tree is then pushed into the stack.

XOR Trees

- Memory efficient trees.
- Left = ⊕ of its parent & left children
- Right = ⊕ " " " " right "
- Root node's parent = NULL
- Leaf node's children = NULL

Binary Search Trees (BSTs)

- Element less than root goes to left subtree
- " greater " " right subtree

Balanced BSTs, AVL Trees

(Adelson-Velskii & Landis)

Notes on Binary Search Trees

→ 1) Performing inorder traversal producer a sorted list.

(2) BST consider only the left/right subtree to search an element and not both.

Finding an Element in BST

→ Algorithm

(1) Accept tree root, & data to be searched.
(2) If data < root then search left subtree.
(3) If data > root then search right subtree.
(4) Return the element

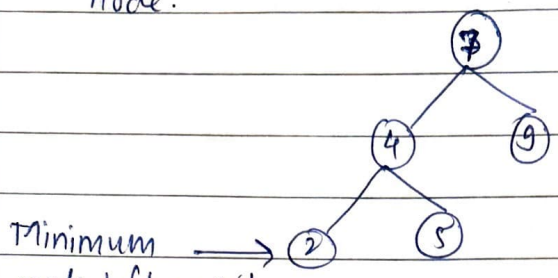
```
func (bst *BST) Find(data int) *BSTN {
    root := bst.root
    if root == nil {
        return nil
    }
    if data < root.data {
return    findHelper(root, data)
    } else if data > root.data {
        return findHelper(root, data)
    }
    return root
}
```

Non-Recursive

```
func FindNonRecursive(root *BSTN, data int) bool {
exists := false  for root != nil { if root.data == data {exists = true}
                                                  return exists
        if ~~root > dat~~ data < root.data {
            root = root.left
        } else if data > root.data {
            root = root.right
        }
}   return exists }
```

Finding minimum element in BST

Algorithm: The minimum element in BST is the leftmost leaf node.




```
func findMinimum (root *BSTN) *BSTN {
    if root == nil {
        return nil
    } else if root.left != nil {
        return findMinimum(root.left)
    } else {
        return root
    }
}
```

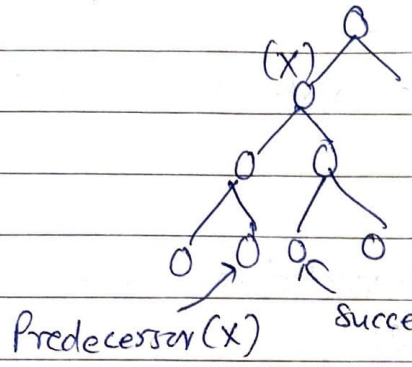
NON-RECURSIVE

```
func findMinNR (root *BSTN) *BSTN {
    for root.left != nil {
        root = root.left
    }
    return root
}
```

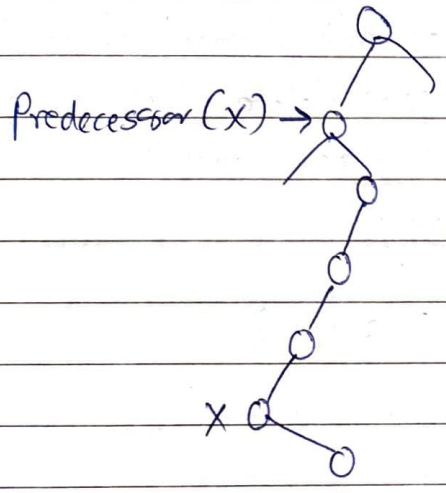
Finding Maximum (opposite of Minimum)

Use above code but instead of 'left' use 'right' pointer.

Inorder predecessor & successor?



When 'x' has 2 children
Predecessor → Max value left subtree
Successor → Min value right subtree



When 'x' does not have a left child then
Predecessor → first left ancestor.

Inserting an element in BST

Algorithm: If element is less than (root) insert in left subtree.
If element is greater than (root) insert in right subtree.

```
func Insert (root *BSTN, data int) *BSTN {
if root == nil {
    newNode := &BSTN{}
    newNode.data = data
    newNode.left = nil
    newNode.right = nil
root.data = data
} else {
    if (root.data > data) {
        root.left = Insert(root.left, data)
    } else if root.data < data {
        root.right = Insert(root.right, data)
    }
    return root
} // level of else.
}
```

Deleting an element from BST

→ 3 possibilities of node deletion

① If node is a leaf node, make its parent point to nil.

② If node has one child,
- → make node's child connected to parent
- → delete node, node = nil

③ If node has two childrens:
- → Replace key of this node, with largest element of left subtree & recursively delete that node.
- → Largest node in left subtree cannot have right child, this delete is easy.

```
func Delete (root *BSTN, data int) *BSTN {
    if root == nil {
        return nil
    } else if data < root.~~lef~~data {
        root.left = Delete(root.left, data)
    } else if data > root.data {
        root.right = Delete(root.right, data)
    } else {
        // found the node
        // Condition 1 : It has 2 childrens
        if root.left != nil && root.right != nil {
            temp := FindMax(root, data)
            root.data = temp.data
            root.left = Delete(root.left, root.data)
        } else {
            // Node has 1 child
            if root.left == nil {
                root = root.right
            }
```

```
    if root.right == nil {
        root = root.left
    }
} else
return root
}
```

Balanced Binary Search Trees (B-BSTs)

This comes into existence when we are trying to reducing the worst time complexity from O(n) to O(log n).
HB(k), k is the difference between left subtree & right subtree height.

Full Balanced Binary Search Trees → HB(k) = 0

AVL – Adelson – Velski – Landis Tree

HB(k), k = 1 i.e. balance factor = 1.

Properties:

① It is a binary search tree.

② For any node 'x', the height of left subtree of x & height of right subtree of 'x' differ by at most 1.

Maximum No. of Nodes

$N(h) = N(h-1) + N(h-1) + 1$
$= 2N(h-1) + 1$
$= O(2^h) \Rightarrow h = \log n \approx O(\log n)$

h-1 : No. of nodes in both left & right subtree.

Minimum No. of Nodes

$N(h) = N(h-1) + N(h-2) + 1$
$= O(1.618^h)$
$\Rightarrow h = 1.44 \log n \approx O(\log n)$

h-1 : No. of nodes in left subtree

h-2 : No. of nodes in right subtree

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with 'n' nodes is O(log n)

AVL Tree Declaration

```
type struct AVLTreeNode {
    left *AVLTreeNode,
    data int,
    right *AVLTreeNode,
    height int
}
```

Height of an AVL Tree

```
func (root *AVLTN) int {
    if root != nil {
        return root.height
    }
    return 0
}
```

Rotations

Insertion & Deletion in AVL trees may cause HB(k) > 1, however, inorder to ensure the tree remains an AVL tree the HB(k) = 1 and to achieve this we use rotations.

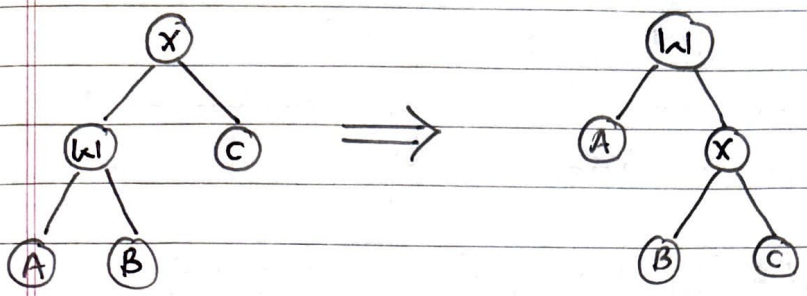
Scenarios, where an AVL tree becomes non-AVL.
When, an insertion happens:

1. Into left subtree of the left child of X.
2. Into right subtree of the left child of X.
3. Into left subtree of the right child of X.
4. Into right subtree of the right child of X.

Case 1 & 3 are symmetric
Case 2 & 4 are symmetric.

Single Rotations

LL Rotation (Case-1)

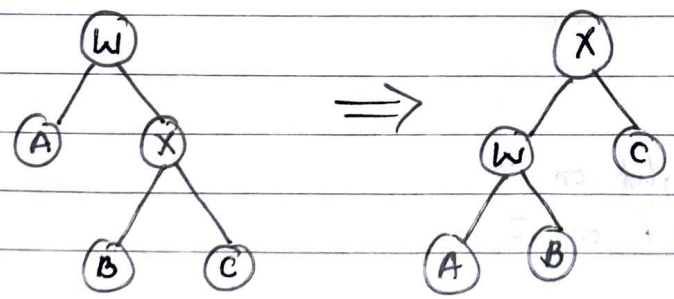


Algorithm:

1. W is x's left child. (wl = X.left)
2. wl's right child becomes x's left child. (X.left = wl.right)
3. wl's right child = X (wl.right = X)
4. x's height = maximum of x's left & right subtree height + 1
5. wl's height = max of wl's left subtree & x's height.

```
func SingleRotateLeft (X *AVLTN) *AVLTN {
    wl := X.left
    X.left = wl.right
    wl.right = X
    X.height = max(Height(X.left), Height(X.right)) + 1
    wl.height = max(Height(W.left), X.height) + 1
    return wl
}
```

RR Rotation (Case-3)



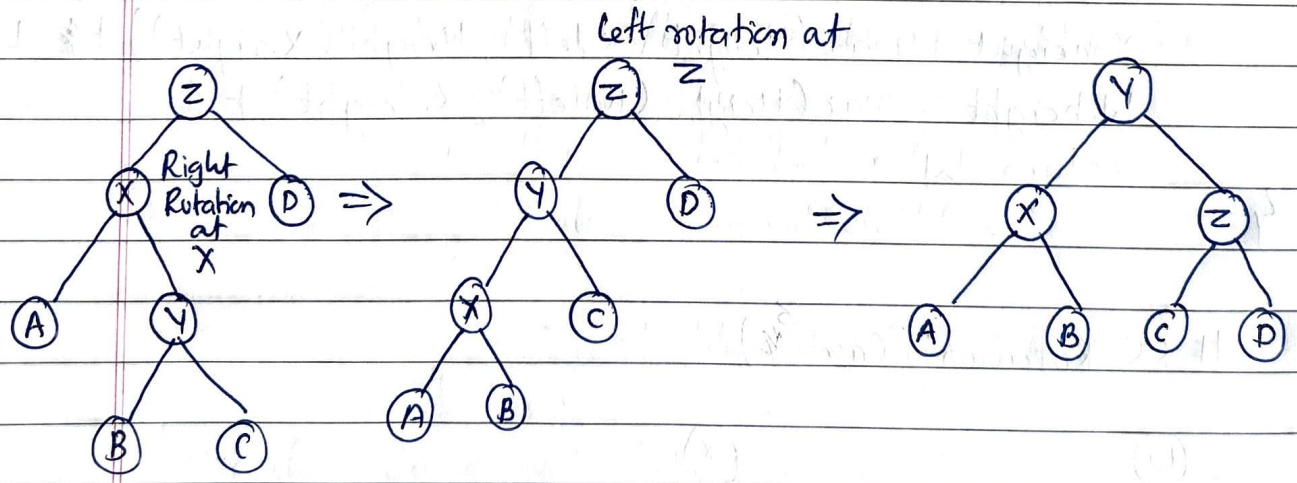
Algorithm:

1. X is W's right child (X = W.right)
2. X's left child becomes W's right child (W.right = X.left)
3. W becomes X's left child (X.left = W)
4. W's height = max of W's left & right subtree + 1
5. X's height = max of W's height & right subtree of X + 1

```
func SingleRotateRight(W *AVLTN) *AVLTN {
    X := W.right
    W.right = X.left
    X.left = W
    W.height = max(Height(W.left), Height(W.right)) + 1
    X.height = ~~max(W.height, X.~~
               max(Height(X.height), W.height) + 1
    return X
}
```

Double Rotations

LR Rotation (Case-2)

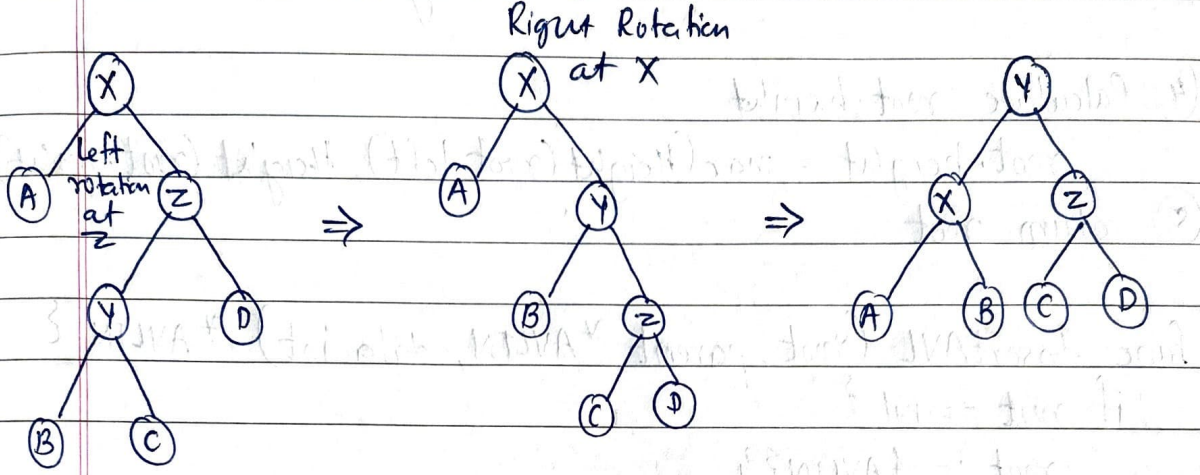


Algorithm:

1. Perform SingleRotateRight on X
2. Perform SingleRotateLeft on Z
3. ~~Done~~ Done

```
func DoubleRotatewithLeft (z *AVLTN) *AVLTN {
    z.left = SingleRotateRight (z.left)   ∵ X ⇒ z.left
    return SingleRotateLeft (z)
}
```

RL Rotation (Case - 4)




```
func DoubleRotatewithRight (X *AVLTN) *AVLTN {
    X.right = SingleRotateLeft (X.right)
    return SingleRotateRight (X)
}
```

INSERTION INTO AN AVL TREE

Insertion into AVL is similar to BST, however, we must check if there is any height imbalance & then apply appropriate rotation functions.

Algorithm :

(1) Pass root, parent & data to Insert.

(2) If data is less than root.data, then Insert root.left.

(2.1) If height after insertion of left node (of left & right subtree) == 2

↳ (2.1.1) If data < root.left.data (Inserting into left subtree of left child of X)

↳ SingleRotateLeft (root)

else

DoubleRotateLeft (root)

③ If data > root.data then (Insert at right subtree)

③.1 If height difference of left & right subtree == 2 then

③.1.1 If data < root.right.data (Insert right subtree left child of X)

root = SingleRotateRight(root)

else

root = DoubleRotateRight(root)

④ Calculate root.height

root.height = max(Height(root.left), Height(root.right))

⑤ return root

```
func InsertAVL (root, parent *AVLTN, data int) *AVLTN {
    if root == nil {
        root := &AVLTN{}
        root.data = data
        root.left = root.right = nil
    } else if data < root.data {
        //Left subtree
        root.left = InsertAVL(root.left, root, data)
        if Height(root.left) - Height(root.right) == 2 {
            //Rotation is required
            if data < root.left.data {
                root = SingleRotateLeft(root)
            } else {
                root = DoubleRotateLeft(root)
            }
    } else if data > root.data {
        // Right Subtree
        root.right = InsertAVL(root.right, root, data)
        if Height(root.right) - Height(root.left) == 2 {
            if data < root.right.data {
                root = SingleRotateRight(root)
            } else { root = DoubleRotateRight(root) }
    }
```

```
    root.height = max (Height (root.left), Height (root.right)) + 1
    return root
}
```

Other Variation on Trees

1. Red-Black Trees
2. Splay Trees
3. B-Trees
4. Augmented Trees
5. Interval Trees (Segment Trees)
6. Scapegoat Trees