**Binary Search Tree(BST)** :

1. Define a binary search tree, complete BT, full BT.
2. **Check if a given BT is a BST**. Note : The supposedly correct way of recursively checking left and right subtree is wrong. There are two approaches :

Approach 1(inorder traversal) : Do an inorder traversal and store the values in an array. Check if the array is sorted. This leads to O(n) time and O(n) space. To save space, use a variable(prev)ious) to store the value of inorder predecessor and check if current node value is smaller than the previous node value, its not a BST. This is O(1) space, not considering recursion stack.

public boolean isBST1(Node root) {

if (root != null) {

if (!isBST1(root.left))

return false;

if (prevNode != null && prevNode.data >= root.data) {

return false;

}

prevNode = root;

return isBST1(root.right);

}

return true;

}

Approach 2(min-max approach) : Just check that the root node lies in a given range which gets narrowed down as we move down the tree. Recursively apply this range check to left and right subtrees also. Initialize the min=-infinity and max = +infinity for root. If the node value lies outside this range, its not a BST. This approach is better and efficient.

public boolean isBST2(Node root, int min, int max) {

if (root != null) {

if (root.data > max || root.data < min) {

return false;

}

return isBST2(root.left, min, root.data)

&& isBST2(root.right, root.data, max);

} else {

return true;

}

}

Note : We have assumed above that there are no duplicate keys in the BST. If there is, then apply that check too.

1. **Find the size of the largest BST in a given BT**. Size = number of nodes, not height of the tree. Use the min max approach to check for BST and return size in the same function.

public static int findSize(Tree root) {

int self = isBST(root,Integer.MIN\_VALUE, Integer.MAX\_VALUE);

if(self>=0) return self;

int left = findSize(root.left);

int right = findSize(root.right);

return Math.max(left,right);

}

public static int isBST(Tree root, int min, int max) {

if(root==null) return 0;

if(root.data<min || root.data >max) return -1;

int left = isBST(root.left,min,root.data);

int right = isBST(root.right, root.data, max);

if(left <0 || right <0) return -1;

return left+right+1;

}

1. **Evaluate an expression tree**.An expression tree is a full BT with operands at leaf nodes and operators at internal nodes. Trick : In a tree of size n(no of nodes,,root at index -1), there are n/2 leaf nodes. So, do the following operation on leaf nodes : a[i/2] = process a[I] and a[I+1] with operator a[I/2]…..for <n/2<I<=n. Again parents of these leaf nodes become the leaf nodes, solve recursively for tree of size(n/2)

Double solve(double a[], int n) {

If(n==1) return a[n];

For(int i=n/2+1; I<=n; I++) {

a[i] = a[I] op a[I+1] op = a[I/2] which may be +, -, \*, /

}

Solve(a,n/2);

}

Here we are doing recursion in a bottom -up manner solving tree of size n, n/2, n/4, n/……1

Time complexity : perhaps nlogn….check it.

1. Multiply two LL
2. Detect & remove cycle in LL.Hint : Use floyd warshal’s cycle detection algo using two pointers.
3. Searching in a rotated array,circularly sorted array, row-wise&col-wise sorted matrix.
4. BT traversal preorder/inorder/postorder/levelorder both recursive and iterative(iterative part is important). Generate a bst using given inorder&(preorder/postorder/levelorder) traversals. Level order traversal can be done using recursion(naive) or using Queue(better approach) . Print all nodes of tree level wise : <http://algorithms.tutorialhorizon.com/level-order-traversal-print-each-level-in-separate-line/>

Application : print extreme nodes of a tree in alternate order. See my soln here <http://www.practice.geeksforgeeks.org/problem-page.php?pid=700308>

1. Print bt in vertical manner from left to right. Print top, bottom,left,right view of a bt.level order traversal in zig-zag/spiral, print all nodes b/w to given levels.