Binary Search Tree

BST property,

$$\boxed{\text{left subtr data} < \text{node.data} < \text{Right subtr data}}$$

this property must be followed by every node and every subtree.

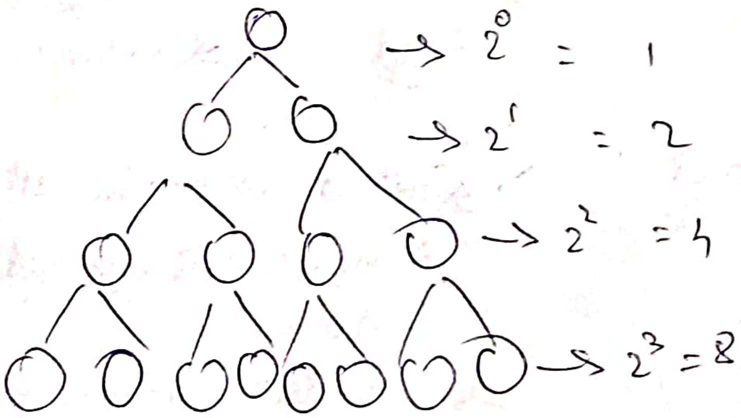
Construct Binary Search Tree from a sorted array.

* Use merge sort kind of recursion where you split array into two haves and make mid ele as root and left half arr for left side tree and right half arr for right side tree.

Base case : if (low > high) "then no elements are left to attach."

* Initially we ~~pre~~ try to attach on left sides and when no element left, then we come to right side

BST contruct differs for binary tree creation



$$n = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots 2^{h-1}$$

$n = 1 \times 2^h$ $a=1$, $r=2$ G.P series

$\boxed{\log_2 n = h}$

For Minimum ~~to~~ element

Go deep in left side of BST. Minimum node
it will never have left node.

Maximum element

Go deep in right side of BST. Maximum
node will never have right node.

Size, Sum, Height, Diameter

Same like Binary trees

Max, Min

find according to the data and move left
right accordingly.

Add nodes in BST

The node will be added ~~at~~ & when
it hits null because we are finding the
node position and the adding.

Remove node in BST

* No child → Return null
* One child → (Left == null) → return right
  (Right == null) → return left
* Two child → Get maximum in left subtree
  → Remove that max node from subtree
  → Set that new (max in left subtree) node accordingly
  → Then return that node

$\boxed{TC = O(\log n)}$

Replace with Sum of Larger

* Do a reverse inorder traversal

Apr-23 Lowest Common Ancestor (LCA)

$\boxed{TC\ O(\log n)}$

* If data > d_1, d_2 → move right
* If data < d_1, d_2 → move left
* Else that is our LCA.

This works only if both d_1 & d_2 are present in the tree.

→ Why BST has (logn) complexity?

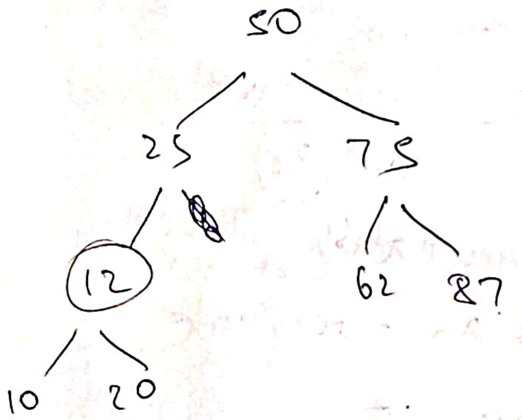
Some pts. to note:

a) Binary ~~Try~~ Tree
* Find is O(n)
* Node to root path O(n)

b) BST
* Find is O(logn)
* ~~So~~ Since find is (logn) then node to root path also has to be (logn).

If nodes are not present,

eg: $d_1 = 11$ $d_2 = 15$



Space for Tree will always ~~not~~ be "logn" in Recursive approach because of height of tree

In this case, 12 will returned as LCA which is not correct because ~~it doesn't~~ both nodes doesn't exist.

So if nodes doesn't exist, then you must find first if both nodes doesn't exist.

Find In Range (Increasing order)

TC : not exactly O(logn)

worst case O(n)

But at best and avg. case, this approach is some what optimized

You can also ~~put~~ do normal inorder which TC is raw O(n), whereas discussed approach may be optimized at cases

* If node.data is inclusive of lo & hi, then call on left & right.
* If node.data < lo & high, then move right
* If node.data > lo & high, then move left.

Target sum pair ~~approaches~~ in BST

Approaches

	Time	Space
1)	n logn	logn
2)	n	n
3)	n	logn

logn ⇒ (Recursion stack space) ↓ Instead 'logn' use 'h → height of tree', because in case of skew tree, 'h' will be n & not logn.

App 1: Find counterpart for each node.

Eg: Tar = 100

node data (12) → 100 − 12 = (88)

find (88)

App: (1) Inoder work, then make a find on every node.

App: (2) Inorder work, store in array list

Apply two pointer approach on array list and find counterpair and print.

App: (3) Perform inorder & reverse inorder "Iteratively" using two stack. "Iteratively" gives you the power to control which is not provided recursively. (→ Do with Pair class)

Finally you can also use two sum technique with Hashmap TC O(n)