# Questions that are hard in BST

* Merge two BSTs with limited extra space
* Find if there is a triplet in a balanced BST that adds to zero
* Find a pair with given sum in a balanced BST

# Binary Search Tree in Java

* A lot of questions can be done using stack. Make sure you see if you can solve it using stack.
* For balanced trees O(h) = O(log n)
* For only one child trees O(h) = O(n)
* To find the total number of binary search trees possible. You have to find the Catalan number.
* In this documents in a lot of places, I have created a global variable which is not good.
  + Rather than that I can create a class with the variable and I pass this class all the time
  + Then I can always update my value by accessing the variable of the class.
  + I will have to initialize the class when I will be calling the function.
  + **class Sum {**
  + **int sum = 0;**
  + **}**
  + **Sum value = new Sum();**
  + **add(root, value);**
  + **static void add(Node root, Sum value) {**
  + **if (root == null) return;**
  + **add(root.right, value);**
  + **root.value += value.sum;**
  + **value.sum = root.value;**
  + **add(root.left, value);**
  + **}**

## Advantage of hashtable over BST

* Search is O(1) in hashtable and O(logn) in BST
* Insert is O(1) in hashtable and O(logn) in BST
* Delete is O(1) in hashtable and O(logn) in BST

## Advantage of BST over hashtable

* We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
* Doing [order statistics](https://www.geeksforgeeks.org/find-k-th-smallest-element-in-bst-order-statistics-in-bst/), [finding closest lower and greater elements](https://www.geeksforgeeks.org/floor-and-ceil-from-a-bst/), [doing range queries](https://www.geeksforgeeks.org/print-bst-keys-in-the-given-range/) are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.
* BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.

## Searching

* See if the root is the term you are searching for.
* If yes then return the root.
* If not then see if root is smaller or larger than the term you are searching for.
* If root is smaller then call this function recursively but with root.right as root.
* If root is larger then call this function recursively but with root.left as root.
* Average Time Complexity: O(logn)
* Worst Time Complexity: O(n)

## Insertion

* The most important thing about insertion is that new elements are always added to the leaf node.
* First check if root is null.
* If it is then make a root with the element and return the root.
* If the root is not null then check its value.
* If the root is smaller than the element to be inserted then call the function recursively and make the root as right.
* If the root is larger than the element to be inserted then call the function recursively and make the root as left.
* It will keep on going until it sees a root that is null.
* This would mean that it has reached the end and will make a new node as the leaf node.
* It is important to note that always return the root and in the function call, make sure that you point root.right/root.left as the new node made. This is because if a new node is going to be made then it has to be made sure that it is connected to the parent. This has been shown below:
  + **else if (root.value < key) root.right = insert(root.right, key);**
  + **else if (root.value > key) root.left = insert(root.left, key);**
* Average Time Complexity: O(logn)
* Worst Time Complexity: O(n)

## Deletion

* This is one of the most completed one.
* You will have to say if the node you are going to delete has any children.
* If there are no children i.e. it a leaf node then just delete it.
* If it has only one child then replace that node with its child.
* If it has 2 nodes then there is a problem:
  + In this case both inorder successor or inorder predecessor can be used.
  + We mostly use inorder successor.
  + This is found by the smallest element on the right side of the root
  + **static Node findInOrderSuccessor(Node root) {**
  + **if (root.left == null) return root;**
  + **Node successor = findInOrderSuccessor(root.left);**
  + **return successor;**
  + **}**
* Average Time Complexity: O(logn)
* Worst Time Complexity: O(n)

## Inorder using stack

* Create a stack s and temporary node n.
* Now assign n as the root;
* Keep adding n to the stack and keep going to n.left until you get null;
* Now start popping from stack until stack is empty.
* For every node you pop, you can add that to the inorder array.
* Check if it has a right element.
* If it does then go to the right element, add it to the stack and keep going left until you get null.
* Keep adding this to the stack.

## Preorder to BST

* 1st way
  + Using insert will cause it to do it in O(n^2).
  + So the second trick is used.
  + In this, the index is made a global variable.
  + For every number in the inorder array, it is checked if it should be on the left of the node or right. This is checked using limits.
  + If the current number is between min and max then it should be in this node only.
  + Everytime the function is called a max and min is given
  + For left node, the minimum is the original minimum and maximum is the current node.
  + If it is in this limit then the number should be in the left tree.
  + For right node, the minimum is the current node and maximum is original maximum.
  + If it is in this limit then the number should be in the right tree.
  + The original minimum and maximum are Integer.MIN\_VALUE and Integer.MAX\_VALUE.
* 2nd way
  + Create a stack and push root in stack.
  + Now keep checking each term in preorder array.
  + If it is smaller than the term on top of stack then peek that top term and add a left Node to it using this current value of preorder transversal.
  + Push the left term in the stack as well.
  + If it is bigger then keep popping the stack until you find the term that is smaller than the current value of preorder transversal. Once you find it then create a right node using the current value of preoreder.
  + After making it, push it to the stack.

## Binary Tree to BST without changing shape

* Take the inorder of the binary tree
  + static ArrayList<Integer> inOrderGetArray(Node root, ArrayList<Integer> inOrder) {
  + if (root == null) return inOrder;
  + inOrder = inOrderGetArray(root.left, inOrder);
  + inOrder.add(root.value);
  + inOrder = inOrderGetArray(root.right, inOrder);
  + return inOrder;
  + }
* Use quick sort to sort the tree
* Use inorder again to assign the values back to the same positions
  + **static int index = 0;**
  + **static void inOrderPutBackValues(Node root, Integer[] in) {**
  + **if (root == null) return;**
  + **inOrderPutBackValues(root.left, in);**
  + **root.value = in[index];**
  + **index++;**
  + **inOrderPutBackValues(root.right, in);**
  + **}**

## BST to Greater Sum Tree

* Create a global variable called add = 0
* First do reverse inorder transversal
* Now when you reach at every node.
  + Save the value of node in a temporary variable
  + Put the value of the node as the value of the variable add
  + Add the temporary variable to add

## Inorder/Sorted array to BST

* Take the middle term as the root.
* Then keep doing binary search and if you are doing binary search on the left part then it will be added to the left of root.
* If you are doing binary search on the right part then it will be added to the right of root.
* Keep returning the root.
* Every time you go left or right make it the root.left or root.right.
  + **static Node createFromInOrder(Node[] arr, int min, int max) {**
  + **if (max < min) return null;**
  + **int mid = (min + max)/2;**
  + **Node root = new Node(arr[mid].value);**
  + **root.left = createFromInOrder(arr, min, mid - 1);**
  + **root.right = createFromInOrder(arr, mid + 1, max);**
  + **return root;**
  + **}**

## Find kth smallest/largest term(super Imporant)

* Make a global variable of k and node. K will be a counter and node will save the kth node.
* Now do inorder transversal.
* Every time you reach a node decrement k.
* When k is 1 then you have reached the node.
* Save the node to the global variable then.
* Complexity - O(h + k)

## Inorder successor

* Create a global variable of k = 1 and Node successor = null
* Noe implement inorder and check in the middle of the two recursive calls if the value of the root is the same as the number whose successor has to be found.
* If yes then decrement k to 0.
* Now also check in the middle of the two recursive call if k = 0 then assign successor node as the current root.
* This successor node is now the inorder successor.

## Merge 2 BST

* Take the inorder of both the BST.
* Now sort them using the merge sort principle to create a third inorder array of nodes.
* Then create a BST from this new inorder array of nodes.