SOSPO

Tree Data Structure (DSA-12)

(New sequence to be like this → Implemented Tree (Techie Delight) and then your deep notes & then GOG other Interviews and then Fenwick tree Segment tree)

(1) Tree self Made Notes (BIT & Fenwick Included)
↳ Implementation Notes depth understanding of in trees.

5 For better Implementation check after BIT or Fenwick Tree

(2) (101) Tree walk through (Interview cake) (Tree Basics words defn)

(3) (102) common Interview Questions (Upgrad) (High Level No codes)

(4) (301) Geeks for Geeks (Usual Tree Question List) ↳ It's Just Questions list (Find it's soln.)

(5) (401) Some coding Interview for tree.

First do this then can go here

5 AS 2

(#) Before moving to Tree and Graph problems first Master (Recursion concepts). Hone your concepts in recursion. Make full notes details in it.

Done few basic Recursion problem and learnt, how to build or come up with Recursion solution.

Tree

(3) Unable to understand why everytime we've to do

```
root = insert(root, 8)
```

Although it is returning some root,

But we are doing changes in root only. so why we've to assign modified root to root again. why can't root be free and keep intact like in LinkedList.

↳ check more different code implementation to understand how it's working.

(4) Try to implement delete code yourself in and check.

(+) Imp Infos To Note When Studying About Tree DSA

(1) You've done much of tree problem using Recursion.
↳ so also check for them, if they can be done using Iteration and build iteration soln. also.

(2) Write your Binary Search own Tree Implementation code in Java Node root, delete etc with well explainy comments, Then,
The way you've did in LinkedList
↳ Also along with Recursion also add iterative approach to do basic tree methods like insert, delete, Inorder etc.

Implementation is correct & well explained → first check whole notes, if somewhere you've added your own implemented binary search tree complete Recursion & Iterative approach and if not add one.

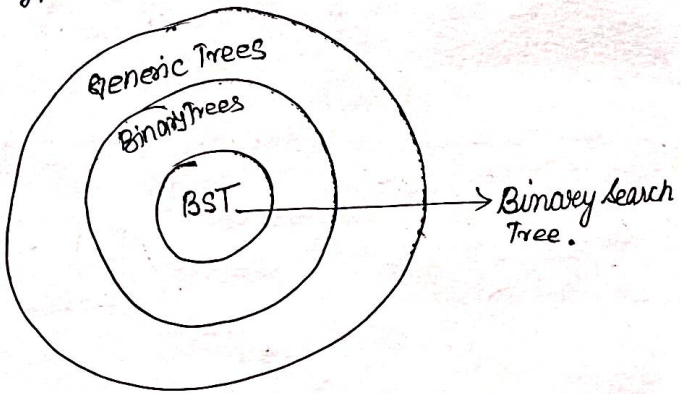
SOOOO ↳ Learn all and need to rearrange all is Required here.

Basic Theory in Tree

can learn about these coding block booklet & Java also

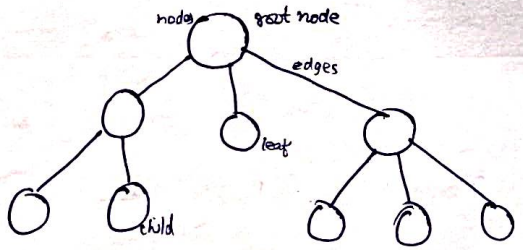
Trees in Java :-

⇒ understanding types of Trees using venn-Diagram.



⇒ Tree are unlike linked list non-linear data structure.

Generic Trees :-



Basic Terminologies :- Read from book only

⇒ Degree = The degree of a node is the number of children that node has.

⇒ For generic Trees :- Degree can be any number from 0 to infinite some finite value.

⇒ Tree Traversal :- ① Pre-order Traversal → each node is processed by any node in it's subtrees.

② Post-order Traversal → Each node is processed after all nodes in its subtrees.

③ Level-order Traversal → Each node is processed level by level from left to right before any nodes in their subtrees.

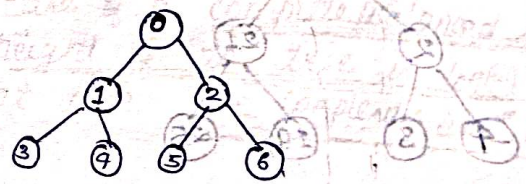
② General Implementation :- Inside the tree class, we make a private class Node which contains a data element and an arraylist of nodes, children. The tree class contains the root node initially null.

Binary Tree :-

has at most two children : left child and rightChild.

A node in binary tree may have 0, 1 or 2 children.

degree of Binary Tree is less than or equal to 2 :

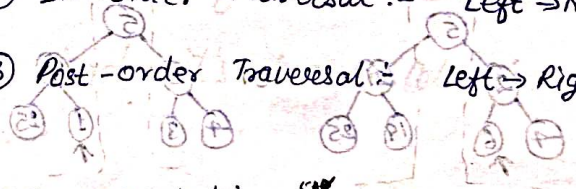


③ Traversals In Binary Tree :-

① Pre-order Traversal : Node → left → Right

② In-order Traversal :- Left → Node → Right

③ Post-order Traversal :- Left → Right → Node.



⇒ Implementation : In the case of Binary tree, we do not need an arraylist of nodes, children. Rather we only need two more nodes inside the node class as Left and Right.

⇒ Similarly will be for right node also if val is greater than Node.data then just put it in the 'node.right' node, so it will return a node only i.e. node.right only so code as :-

```
node.right = insert(node.right, val);
```

returns a node

But is a right node

So

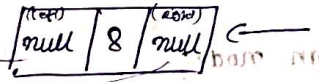
So our code debugging will work as :-

In DSUY

① `Node root = null;` → root = null

② `root = a.insert(root, 8);` → our root node will be created since `root == null` node as root we've passed in parameter.

So it will create root Node as it and returns



These are by default null.

You know this from LinkedList.

After all we are updating our Node only. Nothing to do with BST class. (DSUY) rem!

③ `root = a.insert(root, 3);`

for this code now since our root node which has been already updated in ②. And we've again pass same above root node in our parameter.

So this time {if(node==null){--}} won't be executed first.

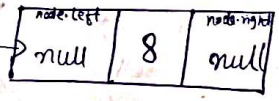
It will first check the value. since {value (3) < node.data (8)} so this statement will be exec

So our debugger moves to following part of the code :-

```
else if(val < node.data){
    node.left = insert(node.left, val);
}
```

So now this line will be executed which again calls the 'insert()' method with taking param as node.left (updated root)

So node.left is → From our update root node is



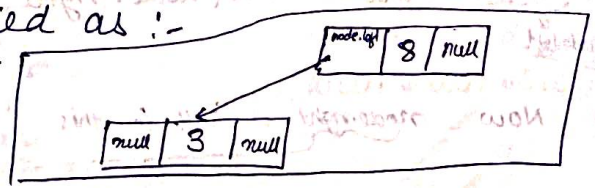
So our code for 'insert(node.left, val)' will now update the left node.

Now since our node.left is null, so on calling insert method it will now execute that {if(node==null){}} and hence create a new node and give value 3 to it.

And this node which it will return we will name it as {node.left = insert(node.left, val);}

Since it's left of that node only (root node).

So our root node now overall will be updated as :-



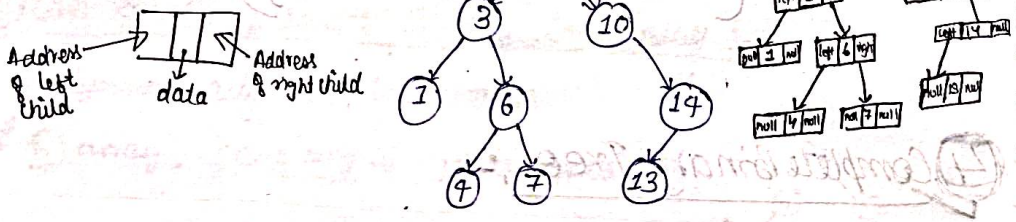
→ This our update tree of node (root) after code ③ execution.

#) Implementation of Binary Search Tree

⇒ Insert(int val) method first

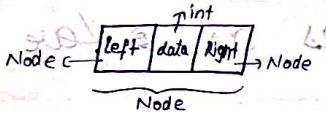
Reference from Youtube Video :-
Binary search Tree in Java-1 : create Binary search tree
Channel: Coding Simplified | 14:49 min video

Binary search Tree is :-




```
class Node {
    int data;
    Node left;
    Node right;
}
```

This gives node as :-




```
public class Main {
    PSVM {
        BST a = new BST();
        Node root = null;
        root = a.insert(root, 8);
        root = a.insert(root, 3);
        root = a.insert(root, 6);
        root = a.insert(root, 10);
        root = a.insert(root, 14);
        root = a.insert(root, 7);
        root = a.insert(root, 1);
        root = a.insert(root, 14);
        root = a.insert(root, 13);
    }
}
```

initially we took root Node with value null as.
So our Tree first part is root as null (Node root)

when we first time

Method insert(Node node, int val);

rem! only identity of the tree that we pass to function i.e. address of the root node and to perform any action on the tree we need to start from the root.

```
class BST {

    public Node createNewNode(int k) {
        Node a = new Node();
        a.data = k;
        a.left = null;
        a.right = null;
        return a;
    }

    public Node insert(Node node, int value) {
        if(node == null) {
            return createNewNode(val);
        }
        if (val < node.data) {
            node.left = insert(node.left, val);
        }
        else if (val > node.data) {
            node.right = insert(node.right, val);
        }
        return node;
    }
}
```

i.e. we are passing local address copy of root's address. If the address is changing we to return at back.

Here this is our initial root Node which we are passing

So initially our root node can be null. and if not null means we've already inserted a node in our root or created our root node itself.

This is same as in LinkedList. done in this way just to save space in above only so.

// If is saying if val is less than take my left node. If in left node there is value then go more deep down if not then add value accordingly

Recursive approach by in Iterative one

and after updating that node we are making tree and returning this updated node and is being updated in our root node here

This means now our root node is not null. So we look for the val. If val is smaller than node.data then put this val on the left node of the node. So we do code as insert(node.left, val), means it will put value in the left

#) Application of Tree :-

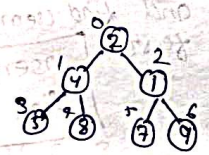
① storing naturally hierarchical data → e.g. file system. ② Organize data for quick search, insertion, deletion. → e.g. Binary search trees. O(log n).

③ Special kind of Trie → It's really fast and efficient and is used for dynamic spell checking. ④ Network Routing Algorithms.

③ We can Implement binary tree using :-

ⓐ dynamically created nodes

ⓑ array. (But only if tree is complete) e.g.



2	4	1	5	8	7	9
0	1	2	3	4	5	6

for node at index = i,

left-child-index = 2i+1

right-child-index = 2i+2

#) Types of Tree :-

① Binary tree → 0, 1 or 2 child nodes only.

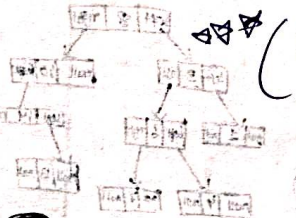
② Binary Search Tree → 0, 1 or 2 child nodes only but left node is smaller and right node is large.

⑤ AVL Tree ⇒ A binary search tree whose left subtree & right differ in height by almost 1 unit.

③ Extended Binary Tree :-

⇒ A binary tree whose every node has either zero or two children.

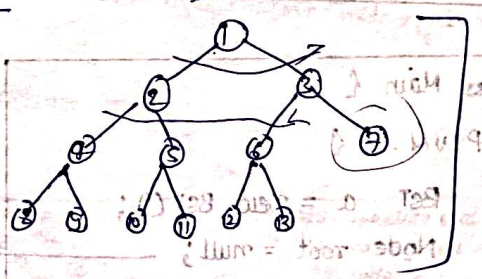
(0 or 2 children only) (& not 1 children)



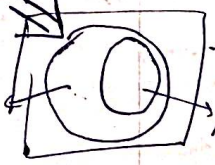
④ Complete Binary Tree :-

⇒ A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes as far as possible.

complete Binary Tree



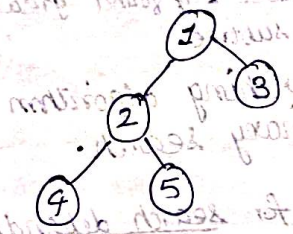
But here also zero or two child nodes only



understand using venn Diagram

Miscellaneous :-

Tree Traversals



A) Depth First Traversals :-

(a) Inorder (Left, node, right) : 4 2 5 1 3

(b) Preorder (Node, Left, right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

B) Breadth First or Level Order Traversal : {1 2 3 4 5}

(First left then only to go to right nodes)

In → means (Node in Mid): L N R ⇒ L N R

Pre → mean (Node in First): N L R ⇒ N L R

Post → mean (Node at End): L R N ⇒ L R N

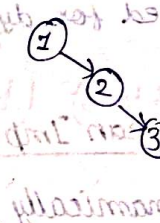
⇒ Height of an empty tree = -1

⇒ Height of tree with 1 node = 0

Binary search tree's particular order or way of arranging its node in tree can result either in balanced or unbalanced tree also its property of insertion.

e.g. if we insert 1, 2, 3, 4.

So we get a data structure of tree as ⇒ which looks less like a tree and more like a long linked list.



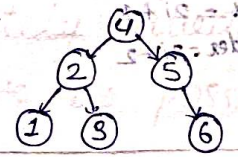
And in this unbalanced tree insertion and find (search) will no longer be fast.

Unbalanced Tree

insert : O(n)
find : O(n)

⇒ height = O(n)

For balanced tree :-

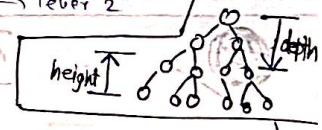


insert : O(log n)
Find or Search : O(log n)

⇒ height = O(log n)

So

In a lot of interview questions or programming you can assume you've a balanced tree.

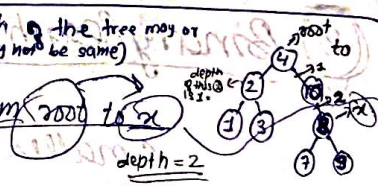


Depth & height of node (x) :

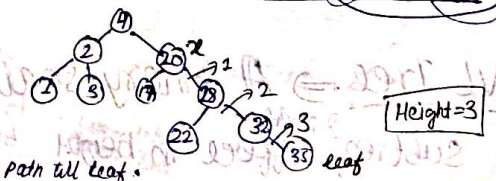
(Height & Depth of the tree may or may not be same)

Depth of (x) = No. of edges in path from root to x

So depth of root node will be always zero.



Height of (x) = No of edges in longest path from x to a leaf.



So height of leaf nodes will be zero.

And height of root node is longest path till leaf.

Also height of tree is defined as height of root node

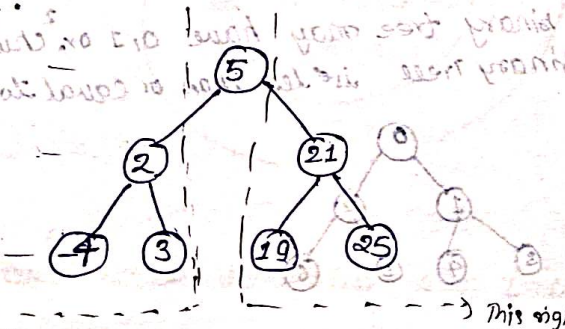
Binary Search Tree :- (BST)

⇒ It is a special type of binary tree in which:-

(1) Every node in the left subtree of a node has a value less than or equal to the value of that node.

(2) Every node in the right subtree of a node has a value greater than or equal to the value of that node.

e.g.:

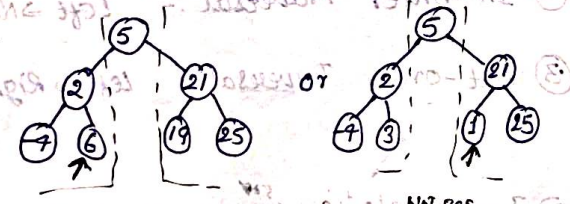


This side part must have values always less than 5.

This right side of the root must always have values greater than 5.

BST

(3) e.g. But this is not BST



or

⇒ In-order traversal in BST gives a sorted order. e.g. in above BST, inorder traversal would be as :- {-4, 2, 3, 5, 19, 21, 25}, numbers in sorted order.

(4) Searching in BST → The main application of BST is efficient searching. Because we compare Node, Left, Right value & find the number & if found greater or less then we just discard that subtree.

⇒ This type of searching algorithm is similar to that of the binary search.

⇒ Time complexity for search depends upon the height of BST. O(Height of BST).

⇒ Only in the case of a complete/balanced binary search tree, the height of the tree is log(n) and hence only in that case Time complexity is O(log(n)).

⇒ BST also can be used for sorting by inorder traversal which gives sorted order.

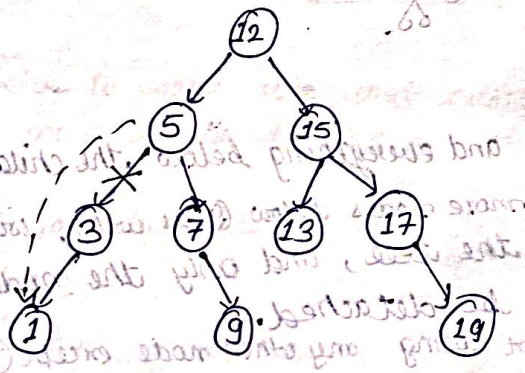
And it's traversal is O(n) → most probably,

n = number of nodes in BST.

⇒ so for this what we are doing in our code is, that we are basically setting node with value (9) to the right child of node 8 value (5).

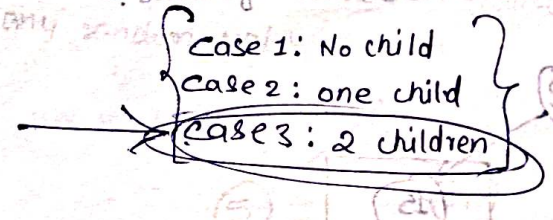
⇒ so this is what we've to do to delete a node with just one sub child or just one subtree connecting to it's parent, and then just wipe it out from the memory.

⇒ so same way could be done for node (3) as well having single child. what we have to do is just set node.left 8 node (5) to (1) instead 8 (3).

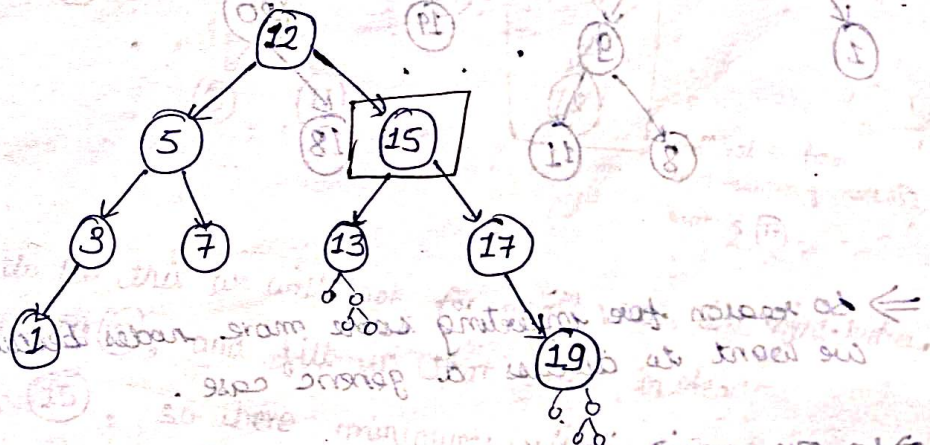


⇒ so now we've seen how to delete node which are leaft nodes and also for nodes which have single child.

(5) Now comes the third case when nodes have two children which we are going to delete.



⇒ so let's come back to node (15) again which we were teying to delete earlier.

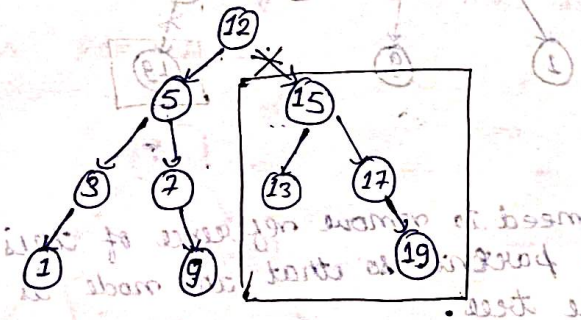


⇒ so here cannot do something like that to connect parent with one of the children while teying to delete (15). e.g. if we add (13) to the parent node (12) then we will lose the right subtree (17) and all it's subtree nodes. similarly for the case if we make (17) as the right child of (12), then we will lose node (13) and its subtree (if any).

⇒ so this case is tricky.

⇒ So deleting a leaf node (node with no children) is easy.

⇒ Now what if we want to delete a none leaf node. e.g. node (15) in our BST.
We can't just cut the link because it will not only remove node (15) but also detach complete subtree of node (15)



So we've to make sure that except (15) no other nodes need to be deleted.

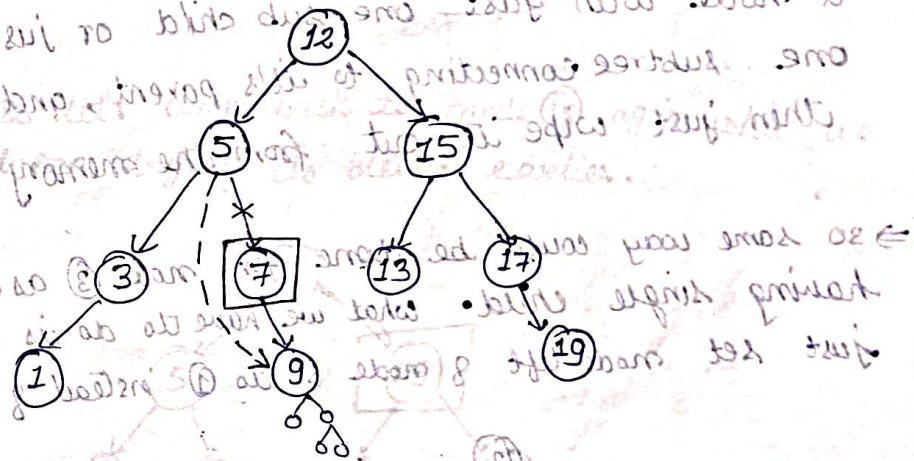
⇒ So this particular node (15) has two child nodes or subtree.

We will see about it later.

First let's discuss the case when node that has to be deleted has only one child.

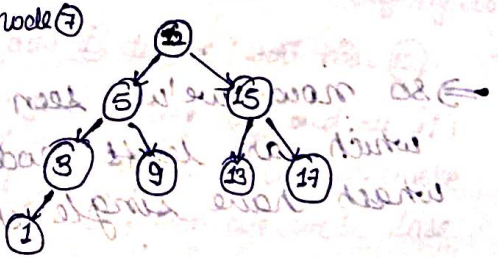
⇒ So for Node to be deleted having only one child e.g. node (7), which has only one child, i.e. right child.

for such a node what we can do is link its parent with it's only child.



So the child and everything below the child, (we could have some more nodes below (9) as well), will remain attached to the tree, and only the node being deleted will be detached.
So we are not losing any other node except (7).

So tree after deletion of node (7)



It's still a BST.

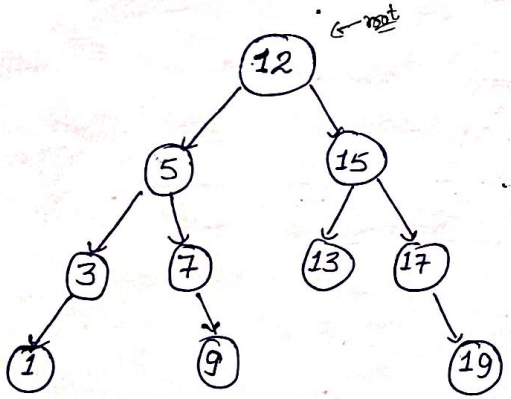
Source Youtube video
Delete a node from Binary search Tree
mycodeschool

(11) Now comes next methods 'Delete'.
To Delete a node from BST.

⇒ But before that we need to understand its various cases and techniques to remove or delete node from tree without affecting it's child node or subtree.

⇒ so here deletion is tricky, its not so straight forward.
so we will discuss here about some of the complications while deleting a node from BST.

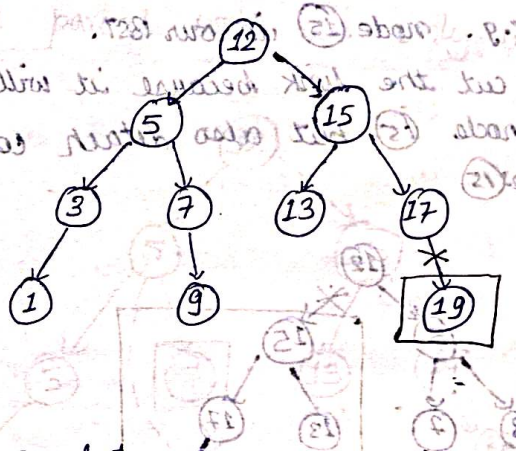
e.g. let our BST be :-



so when we delete a node from BST we need to conserve it's property such that left nodes are smaller than & Right nodes are greater than that particular node.

so while deleting the tree's node we need to again rearrange its node and conserve the property of binary search tree. ✓

⇒ so for example if we want to delete node (19) from our tree then.
we need to do two things :-



(1) we need to remove reference of this node from its parent. so that the node is detached from the tree.
And set the right child of node (17) as null.
i.e. node.right = null.

optional
(2) And the second thing we need to do is to reclaim the memory allocated to the node being deleted. means wipe out the Node object from the memory.

⇒ so this particular node (19) which we are trying to delete is a leaf node. It has no children. And even if we remove this node (19), by simply cutting the link, i.e. removing its reference from parent, & then wiping it out of the memory, there is no problem. Properties of BST is still conserved here. (i.e. left tree less & right subtree greater than node)

⇒ so for now after this code upto ⑧.

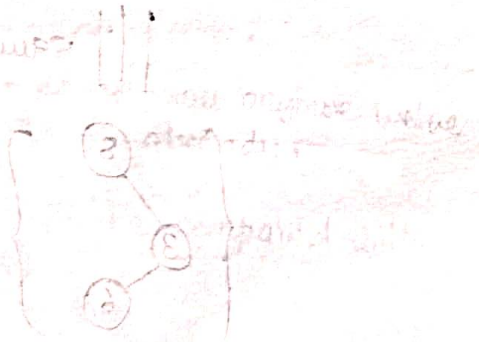
If you want to print these values & trees node then can print it in following way:-

```
Syso(root.data);
Syso(root.left.data);
Syso(root.left.right.nodedata);
```

output

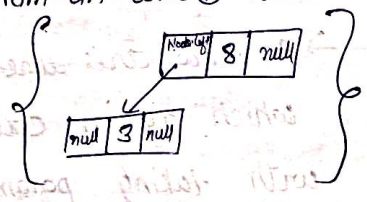
8
3
6

so after all these are all nodes only just arranged in tree way:



④ `root = a.insert(root, 6)`

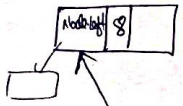
our root has been updated till now was when passing it as a param in code ④ is



so for code ④

→ value(6) < Node.data(8)

→ node.left = insert(node.left, val)



which is now not null anymore.

⇓

so if(node==null) → will not be executed.

*** So now after this iteration our node is node.left as it move forward after each node comparison.

→ it is now node.left

→ So now node.left = is not null anymore we've passed value (3) to it already.

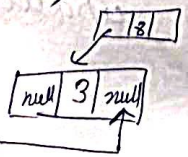
→ So now val(6) will be compared with node.left.data(3),

→ so val(6) > node.left.data(3)

→ node.right = insert(node.right, val)

node.left

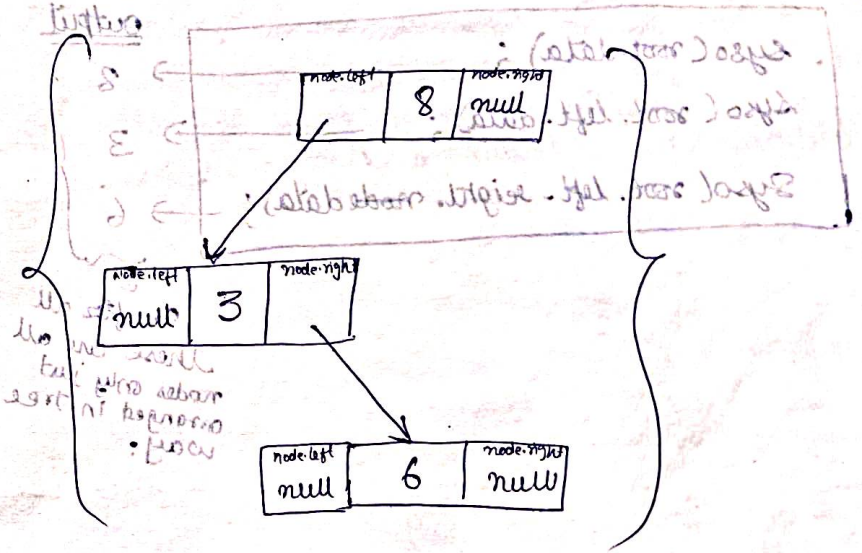
→ Now node.right is null for this



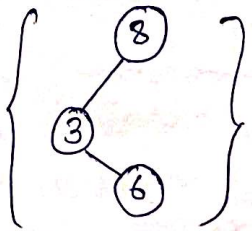
→ so it will create a new node

createNewNode(val) will be executed since (node == null).

and data will be added to it and hence again our tree & node is update as :-



||| equivalent to



in similar way all our nodes will be added and hence our final tree of nodes which start from root a whole tree will be formed. Just interconnected with each other.

More understanding is we are getting node only as Tree

so it's overall root node only which we are updating.

else if (data > root->data) we need to go and delete data from right subtree.

And now if the data is neither greater nor lesser, means it's equal and hence we find our node. And we can try deleting the node containing that data.

else { → so lets handle the three cases one by one.

// case 1: No child

if there is no child of that particular node to be deleted i.e.

(root->left = NULL & root->right == NULL)

then we just simply need to delete that node.

So first we will just wipe out the node from the memory.

{delete root;} → so in here in our root, currently we've the node to be deleted.

So we've used delete operator which is used to deallocate the memory of an object in Heap.

In C we can use 'free()' function.

Now root has become the dangling pointer because object in heap is deleted but root still has its address.

So we can set root as null.

{root = NULL;}

So now we can return root, {return root;}

Reference of this node in its parent will not be fixed tho here. So after this we will have our link corrected by any of above two if else if (data<-) or else if (data>). to connect the link.

// case 2: one child:-

Now handling other cases.

(A) else if (only left child is null) {root->left == NULL}.

then we want to first store the address of the current node which we are trying to delete. Put its address in temporary pointer of node.

{struct Node *temp = root;}

And now we want to move the root or pointer to the right child, so the right child becomes the root of the subtree.

{root = root->right;}

and now can delete the node ie temp;

{delete temp;}

and then return root {return root}.

(B) else if (right-child is null) (root->right == NULL)

Create a temp node, pointing to current node

{struct Node *temp = root;}

and ~~we will move to the left child and~~ then we will make the left child new root of the subtree. so we will move to the left child

and then {root = root->left;}

and then delete the previous root whose address I have in temp and finally return root.

{delet temp;} and then {return root;}.

So as we've seen so we are returning root in all cases so remove this return root statement from all this else if's and write one return root. after everything.

// Case 3:- 2 childre (third case)

else {

So for this case we need to search for minm elements in right subtree of the node that we're trying to delete.

So let's say that fin 'FindMin()' will get us the address of the node with minm value in tree or subtree.

so call the function and collect it's return to temporary node temp

{struct Node *temp = findMin(root->right);}

Now we should set the data in the current node we are trying to delete.

as {root->data = temp->data;}.

Now our problem is getting reduced to deleting this minm value from the right subtree of current node.

i.e. {root->right = Delete(root->right, temp->data);}

So with this much code we are done with delete function.

First we will do code in C++ and then in Java later :-

Because what we are passing here is the only local copy of our root address. If the address is changing we need to return it back.

⇒ 'Delete' function using C++.

This function returns pointer to root node.

```
struct Node* Delete(struct Node *root, int data) {
    //search Node which you want to delete
    if(root == NULL) { return root; }
    else if(data < root->data) { root->left=Delete(root->left, data); }
    else if(data > root->data) {root->right = Delete(root->right, data); }
    else { //whoo... I found you, Get ready to be deleted
        //case 1: No child. (Here root is that node which we've to delete. Here we came after recursive subtree call as done above to search it)
        if(root->left == NULL && root->right == NULL) {
            delete root;  //delete is keyword in c++ to deallocate its address. (Because inside so root->null & that inner subtree node)
            root = NULL;
        }

        //case 2: One child.
        else if(root->left == NULL) {
            struct Node *temp = root;
            root = root->right;
            delete temp;
        }
        else if (root->right == NULL) {
            struct Node *temp = root;
            root = root->left;
            delete temp;
        }
        // Case 3: 2 children
        else {
            struct Node *temp = FindMin(root->right); //After create by yourself to find min value in subtree.
            root->data = temp->data;
            root->right = Delete(root->right, temp->data);
        }
    }
    return root;
}
```

→ In right subtree we will give smaller 1st node is smallest

```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
```

Flow of code (Signature) of method

⇒ So to delete a node we need to first find that node's data in the tree and delete it.

⇒ Rem! only identity of the tree is what we passed in our argument is the root node. And to perform any action on the tree we need to start at root.

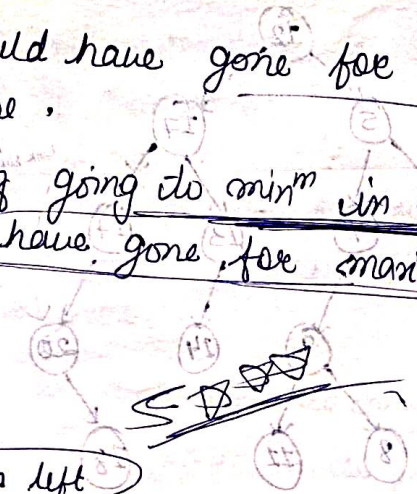
Flow of code (Body) of method

So first case :- If (root==null) means if root is empty, then we would simply say return; means return root, means they are same so return.

else if (data < root→data), means it's in the left of subtree. So we need to go and find data in the left subtree. So we can make a recursive call to delete function passing address of left child & the data to be deleted. i.e. Delete (root→left, data); and this returns the left-child of current node. so set return as left child root→left = Delete(root→left, data);

⇒ Also we could have gone for another approach here.

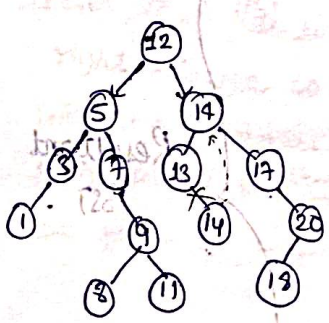
Instead of going to min^m in right subtree, we could have gone for max^m in left subtree.



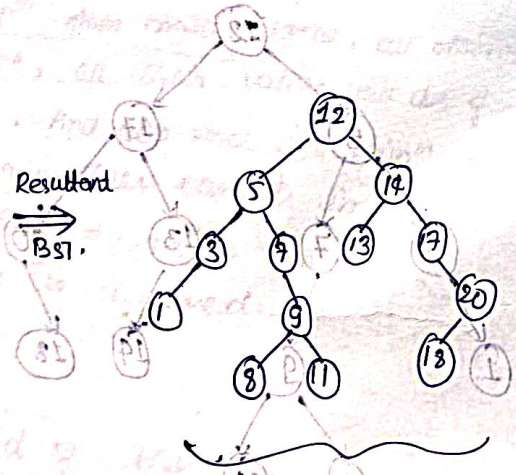
⇒ so steps :-

1. ~~Find~~ max in left
2. copy the value in targeted node
3. Delete duplicate from left-subtree.

So it will follow as :-



Resultant ⇒ BST.

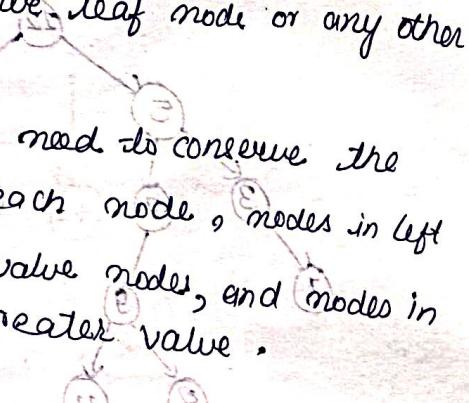


So this also is our answer.

So now let's write overall code for this delete logic

⇒ The final arrangement will be the valid arrangement of our BST.

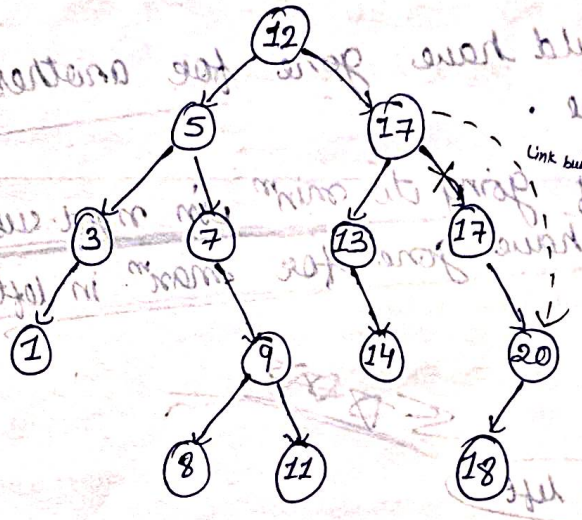
✱ Bid why minm in the right subtree, & why not value with in any other leaf node or any other node with one child.



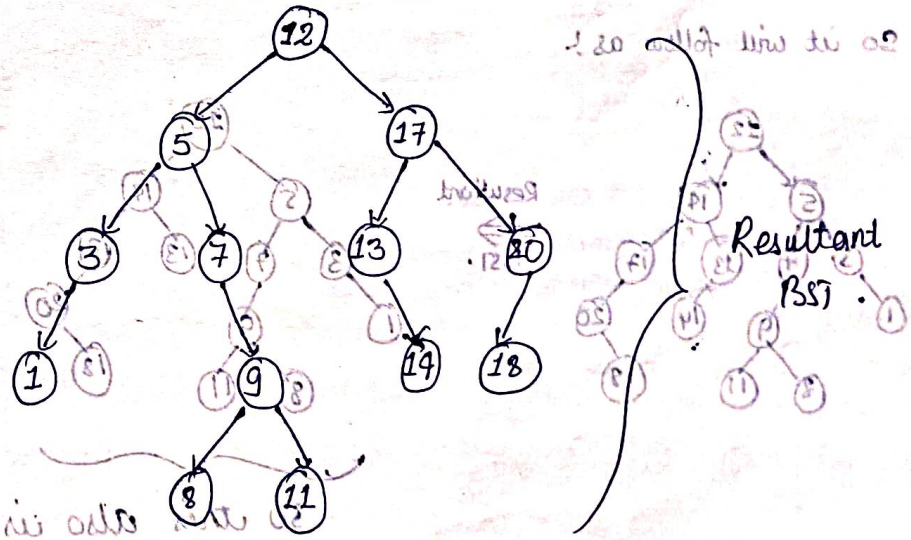
It's because we also need to conserve the property, i.e. for each node, nodes in left should have lesser value nodes, and nodes in right should have greater value.

⇒ so bringing (17) as minm from right subtree, all other subtree's at the left, all their values will be because lesser than (17) so. And also since it's minm in right subtree so all other elements in its right will be greater or equal to no this nodes. once the duplicate is removed everything will be fine.

⇒ so let's get rid of the duplicate. so we just have to build a link. see in fig ⇒



⇒ so after deletion our resultant tree looks like as

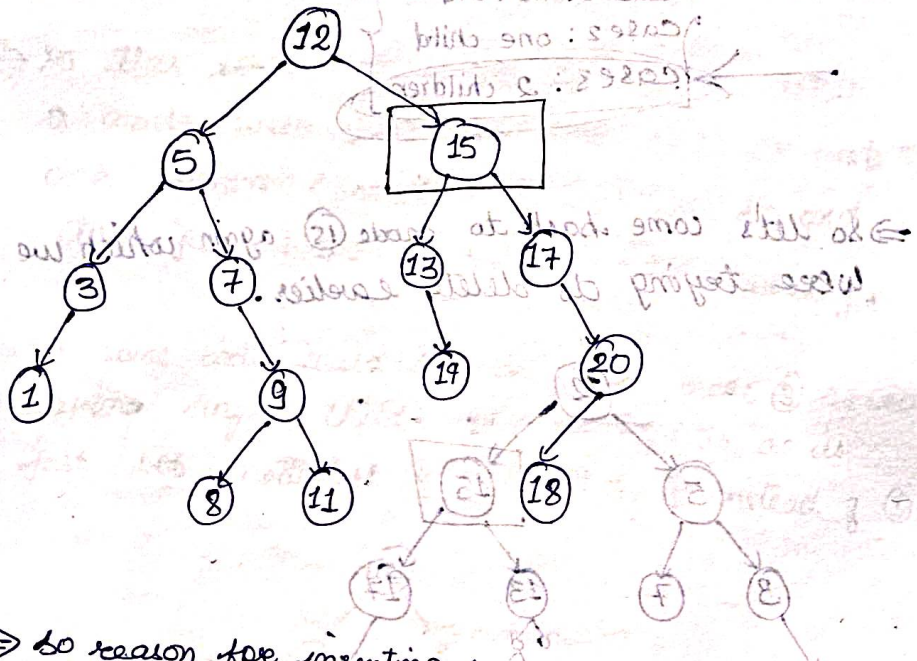


So steps for case-3 :- 2 children :- steps :-

① Find min in right-subtree
② Copy the value in targeted node
③ Delete duplicate from right subtree.

⇒ So before going to the solution let's first insert some more nodes in the subtree for better understanding.

So let our tree be as follows :-



⇒ So reason for inserting some more nodes because we want to discuss a generic case.

⇒ So our main intent is to remove (15) node from our tree.

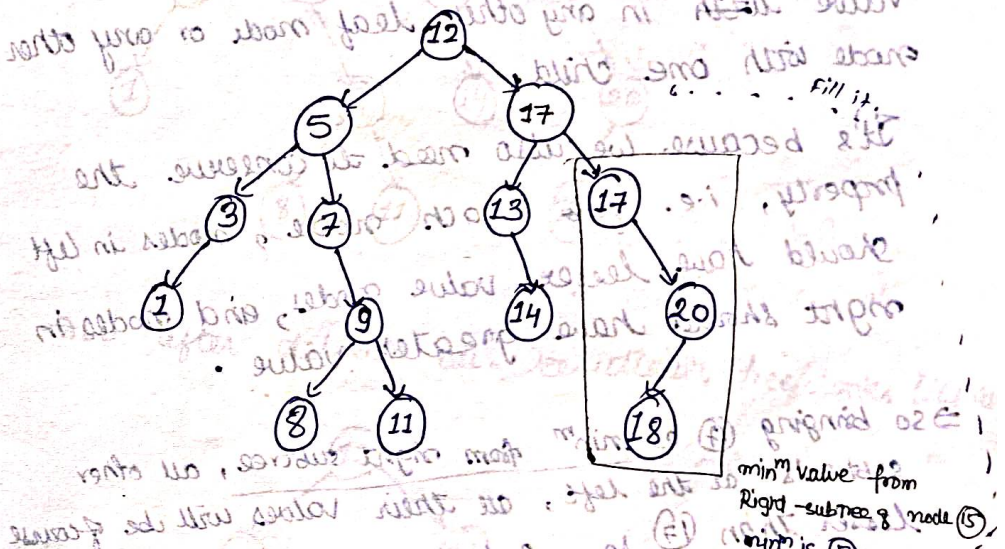
Our delete function will have signature such as :-

```
delete (Node *root, int data){}
```

So it will take pointer or reference to root node and other parameter is the value of node to be deleted in our argument.

⇒ So here our 'into. data = 15'.

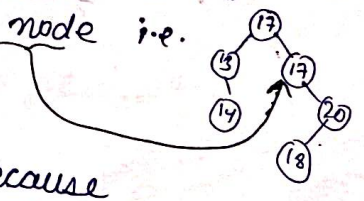
⇒ So what we will do is wipe out this (15) from our node and we will fill in some other value in this node. Ofcourse we can't fill in any random value.



So to fill this we will look for min^m value in right-subtree of node (15), and fill in that value instead of (15). So here minimum is (17).

So fill (17) there.

So after filling node we've two nodes of value (17). but you can notice that this node i.e. has only one child.



We can delete this node, because we know how to delete a node having 1 child. and once this node is deleted our tree will be good.

(11) so if there is nothing on right so it will just call the previous node :- i.e (3)

→ so it prints (3) since it has already covered the left portion.

→ so now again it will go left which is (6) & then goes left which is (4) and in it's left again null so return and //prints (4)

→ Now it goes right of (4) which is again null.

→ it will simply return out of function. Now our real function Node has come to (6)

→ so it will //print (6), and goes right to (7)

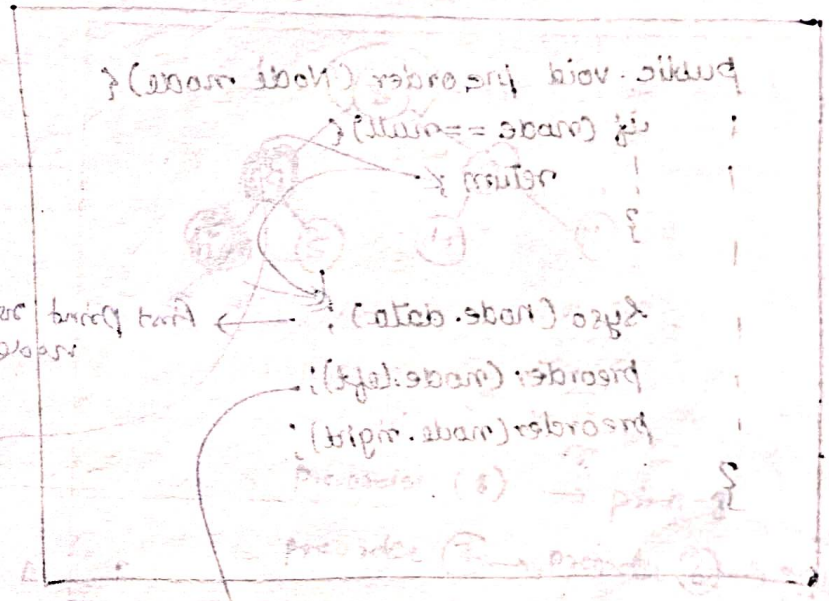
→ so now left is null of (7). So return and just prints (7)

and so on it works.

⇒ If you want to very badly understand it just see the video source link in initial page mentioned.

⇒ so this is how we present tree in Inorder (L N R) or ascending order.

(2) Preorder Traversal :-



Now come to next method :-

"Traversal in Tree":-

Source :- YT video
Binarysearch Tree in Java-3: Print elements in Inorder (sorted/Ascending) in Binary Search Tree.

1) Inorder Traversal :- (Sorted/Ascending)

code for 'inorder(Node node)' method :-

```
public void inorder (Node node) {
    if (node == null) {
        return;
    }

    inorder(node.left);
    syso(node.data + " ");
    inorder(node.right);
}
```

(return means this will directly move to print. our method has finished so directly print.)

inorder(node.left); → 8, 3, 1 (left=null)

syso(node.data + " "); → so prints 1

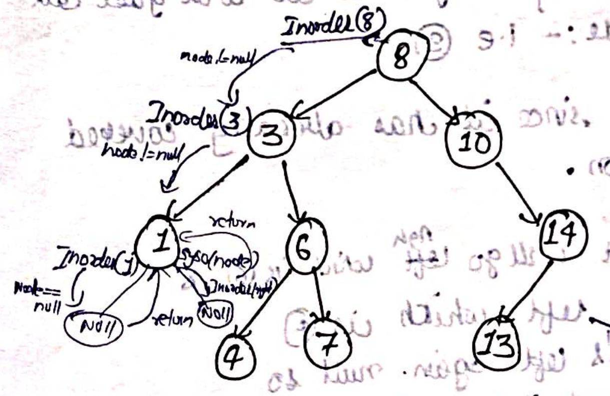
Start

→ This is saying go to left node everytime until you get left node = null. Once you get null

It will print that node (parent) & not node.left → as it is null. So this is happening in deep recursion at leaf node.

```
In Main
    BST a = new BST();
    Node root = null
    root = a.insert()
      "     "
      "     "
    a.inorder(root);
```

e.g. Tree BST



Inorder output
1, 3, 4, 6, 7, 8, 10, 13, 14

→ Make a stack call diagram to understand better.

⇒ How code is working

1. (root) → we've passed
2. (root = ! null) → so moves forward to
3. (recursion) inorder (root.left) → 3
4. (recursion) (root = ! null) → so moves forward to
5. inorder (root.left) → 1
6. (root = ! null) → so moves forward
7. so "return" will be executed.
8. And then comes to syso(node.data + " "); → prints 1
9. And then inorder (node.right) (right of 1)
10. so right is also null so it again simply return.

⇒ You know how it's code works already from video as well as previous explanation from C++ codes.
So you just leave it there.

⇒ Now in case if you just want to see then just visit the video source mention in the 1st video link in start of code :

Now same 'delete' method for tree in Java :-

:- { source :- Yt video :- Binary Search Tree in Java-2: Delete a node of binary search tree }

Code

See if you can find iterative approaches also. like in linkedlist problems

```
public Node delete(Node node, int val) {
    if (node == null) {
        return null;
    }

    if (val < node.data) {
        node.left = delete(node.left, val);
    }
    else if (val > node.data) {
        node.right = delete(node.right, val);
    }

    else { //If(node.data == val) → we found that val to be deleted.
        if (node.left == null || node.right == null) {
            Node temp = null; // No child of that val then null
            temp = node.left == null ? node.right : node.left;

            if (temp == null) {
                return null;
            } else {
                return temp;
            }
        }

        else { // case 3: when Node has two child.
            Node successor = getSuccessor(node);
            node.data = successor.data;
            node.right = delete(node.right, 4);
            return node;
        }
        return node;
    }
}
```

This code is going recursively in the tree nodes traversing by comparing for that data to del

and return null

case 1 :- when node to be deleted has no child.

case 2 :- when node has 1 child either on left or right then store it in temp

and then return that only.

case 3 :- when node has two child.

Now actually find successor (node.data)

See how it's working again

Q. Returns

```
public Node getSuccessor(Node node) {
    if (node == null) {
        return null;
    }

    Node temp = node.right;

    while (temp.left != null) {
        temp = temp.left;
    }

    return temp;
}
```

Since in it right node we need min value

At Right side we want smaller element

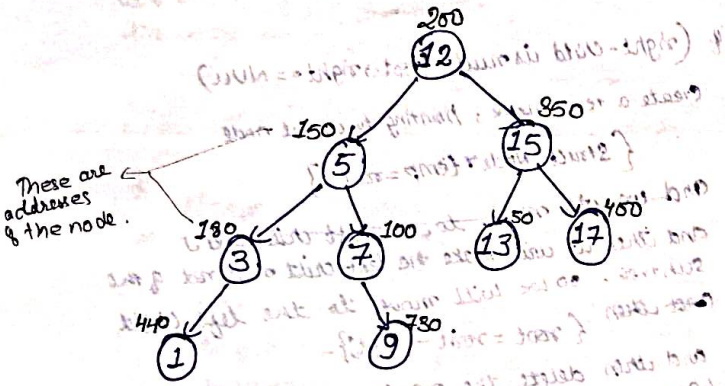
we will find extreme left of the right subtree. that extreme left will be then least number so. to return that.

Go & to left most of tree to get smallest element in that subtree.

one has less node is returned

⇒ So now let's quickly run this code on an example tree.

Given BST :-



Now we want to delete Node (15) from this tree.

So call :- `Delete (200, 15)`

So control will come to

`{ root->right = Delete(root->right, data); }`

So a recursive call will be made and

`Delete (350, 15)`

when execution of Delete (200,15) will pause and it resume to this.

Now for this call we will go to third case of else.
So that is when node has a child.

```
else { //case3 : 2 children
    struct Node* temp = findMin (root->right);
    root->data = temp->data;
    root->right = Delet(root->right, temp->data);
}
```

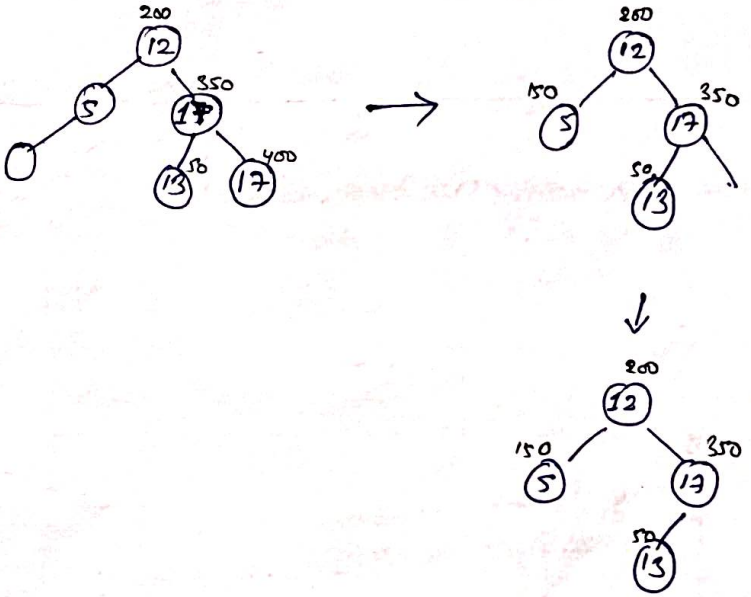
→ Here we find the node with min value at right subtree which is (17). And address = 400

→ So put data in node with address of 350 as (17)

→ And then making recursive call to delete (17) from the right subtree of 350.

⇒ We have only one node in right subtree of 350, so it will call (root->left==Null && root->right==NULL) & then delet current node root, & root = NULL (deallocating the deleted node).

So our trees as



So now you know how our code is working.

(6) **Binary search Tree:** BST is a binary tree with certain properties such as, and left child of the given node contains value less than equal to the given node and right hand child contain node greater than the given node.

(7) **AVL tree or height balanced binary tree :-** It is a variation of the binary tree where height difference b/w left and right sub tree can be at most 1. If at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation.

(8) **Red-Black Tree :-** Another variant of binary tree similar to AVL tree. It is a self balancing binary search tree. In this tree nodes are either colored red or black.

(9) Splay Tree

(10) N-ary tree

(11) Trie Structure (radix tree or prefix tree)

(12) Suffix Tree → (It's like trie to check substring pattern in dictionary)

(13) Huffman Tree

(14) Heap Structure (Min Heap, max Heap)

(15) B-Tree

(16) B+ Tree

(17) R-Tree

(18) Counted-B Tree

(19) K-D Tree (or K-dimensional BST)

(20) Decision Tree

(21) Merkel Tree

(22) Fenwick Tree (or Binary Index Tree)

(23) Segment Tree (or Range Tree)

Tree GK Stuffs

#GK for Tree Data Structure

#1 Types of trees :-

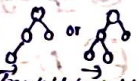
(1) **Binary Tree** :- This is the most basic form of tree structure, where each node has utmost 2 children.

(2) **A Perfect Binary Tree** is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

(3) **A Full Binary Tree** (sometimes refer to as a proper or plane binary tree) is a tree in which every node in the tree has either 0 or 2 children.

(4) In a **Complete Binary Tree** every level except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Strictly complete Binary tree 

Almost complete binary tree  or

Filled from left to right

(5) In the **Infinite Complete Binary Tree**, every node has two children. ...It goes however infinitely.

(6) **Extended Binary tree** :- either 0 or 2 children only for each node.

Tree Data Structure

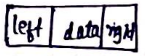
GK OVER TREE

For some basic GK about Tree can refer to our 1st notebook.

⇒ The tree may be defined as a finite collection of special data items called the nodes.

⇒ root node is the main head of tree.

⇒ In a general tree there is no restriction on the number of children associated with any node, but general tree does not make the task easier.
So we better use binary tree.

⇒ A node of binary tree 

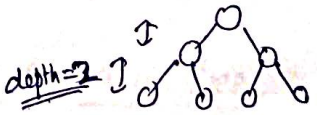
⇒ The maximum possible nodes at any level are 2^n where n is the level number.

Level-1 → 2 nodes
Level-0 → root node
Level-2 → 4 node and so on.

⇒ Total number of nodes in strictly complete binary tree.

$2^{n+1} - 1$

number of nodes.
when n = depth or height of tree node.

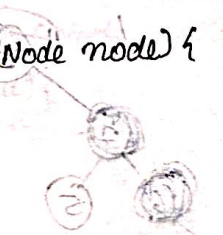


$\Rightarrow 2^{2+1} - 1 = 2^3 - 1 = 7$

$\Rightarrow 2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15$

③ Postorder traversal :-

```
public void postorder(Node node){
    if(node == null){
        return;
    }
    postorder(node.data(left));
    postorder(node.right);
    Syso(node.data);
}
```



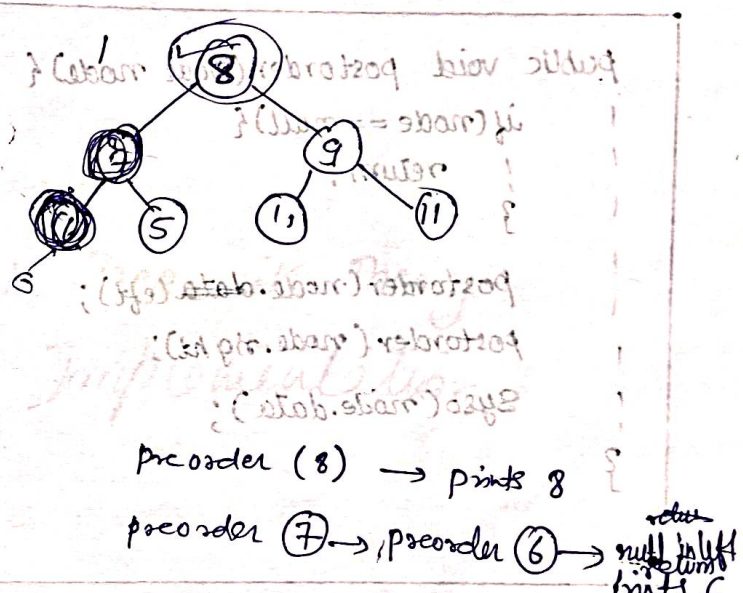
Basic Tree GK Theory
& Implementation

(2) preorder Traversal :-

```
public void preorder (Node node){
    if (node == null){
        return;
    }

    Syso (node.data);
    preorder (node.left);
    preorder (node.right);
}
```

→ First print root node

This goes for recursion until node.left is null then it return nothing, comes back to Syso of parent of that node.left and print that



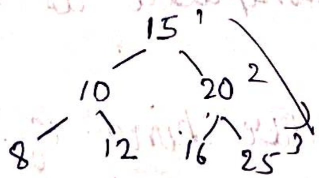
⇒ Node of tree will be as:-

```
class Node {
    int data;
    Node left;
    Node right;
}
```

⇒ Also we can just take `Node root=null;` and update this node as

```
root = bst.insert(root, 5);
root = bst.delete(root, 5);
```

and so on. You know how it works.



(#) To find height of tree:-

⇒ We make recursive calls. O(n) time complexity

code as:-

```
int findHeight( Node root) {        // This will work for both BST & Binary Tree
    if (root==null) {
        return 0; // Doing in a same recursive way you can see for inorder traversal comment.
    }
    return Math.max(findHeight(root.left),      // left subtree
           findHeight(root.right)) +1;          // right subtree
}
```

(To understand its working a recursion tree is best for DP etc)

other way.

```
int maxDepth(Node node) {
    if (node==null){
        return 0;
    }
    else {
        /* compute the depth of each subtree */
        int lDepth = maxDepth (node.left);
        int rDepth = maxDepth (node.right);
        /* use the larger one */
        if(lDepth > rDepth) return (lDepth+1);
        else return (rDepth+1);
    }
}
```

Tree Coding and Implementation Stuff Base on PSS

#1 Very basic Binary Tree Implementation you will find while solving PSS :-

for some basic GK About tree can refer to Crux-1 notebook.

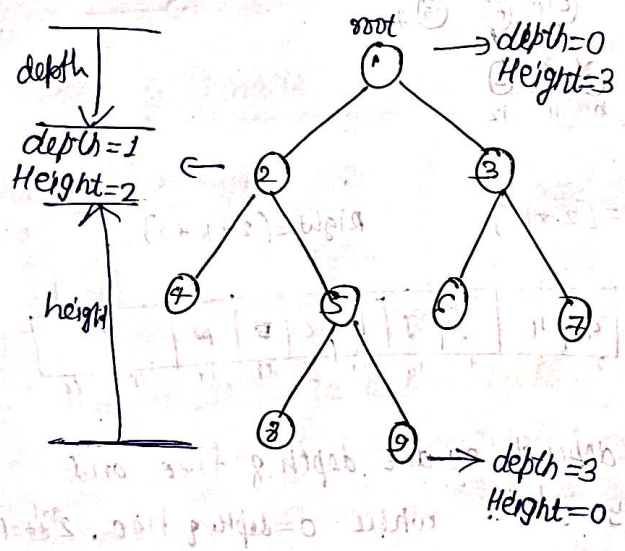
⇒ Binary search tree implementation is done in competitive programming notebook where you can find basic implementation of binary search tree such as adding new node, deleting node, postorder, preorder and inorder traversal. You can see their for very basic understanding.

⇒ Normally the question are regarding tree are only Binary tree or Binary search Tree.

⇒ In PSS it's mainly function based where 'root' node will be provided to you in it's argument, which you've to manipulate as per in the question.

⇒ Other kind of tree questions can be done using simple arrays also like heap and many normal programming questions such as range queries which make use of arrays in segment tree or Binary Indexed

⑤ Height and Depth :-



Height of tree = Height of root

Actually we count number of edges.
But they sometimes also take as number of nodes to define height & depth

⇒ Height of a node = Number of edges in longest path from the node to a leaf node.

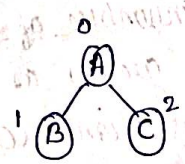
⇒ Depth of a node = No. of edges in path from root to that node.

⇒ Height of empty tree is -1.

⇒ Heigh of leaf node = 0.

(H) Linked Representation of Tree Using an Array :-

(1) using 3 parallel Arrays



root → index	Left	info	right
0	1	A	2
1	-1	B	-1
2	-1	C	-1

and access data as:- Tree[n] ← Array

Tree[n].info

Tree[n].left Tree[n].right,

(2) Sequential Representation of Binary Tree :-

⇒ Instead of 3 Array a almost complete or complete binary tree can be represented using a single array.

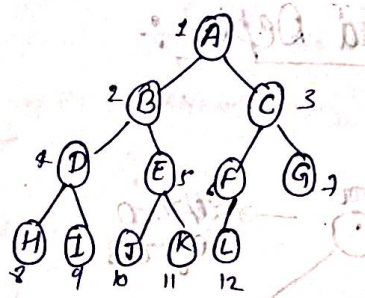
It follows as if K^{th} node then it's

left child at [2*K] & right child at [2*k+1]

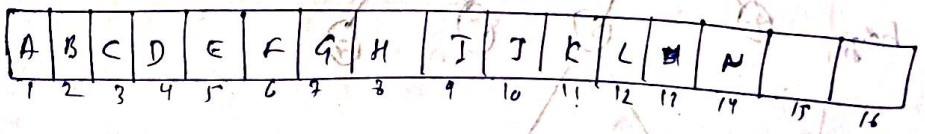
if Root = 1

⇒ If Root = 0 then

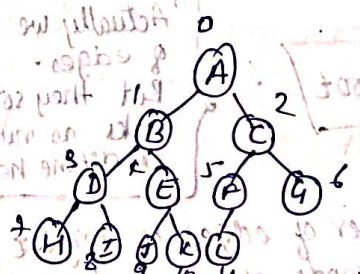
left child [2*k+1] & right child (2*K+2]



Root (A) = 1 left = [2*K] Right = (2*K+1]



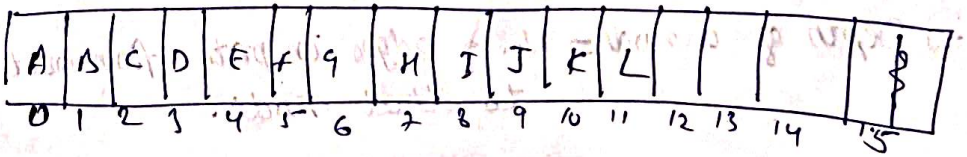
The size of array depends on the depth of tree and it is given by 2^{d+1}. where d = depth of tree. $2^{3+1}=16$



Root(A) = 0

left = [2*k+1]

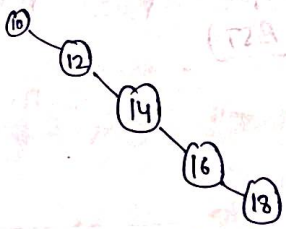
right = [2*k+2]



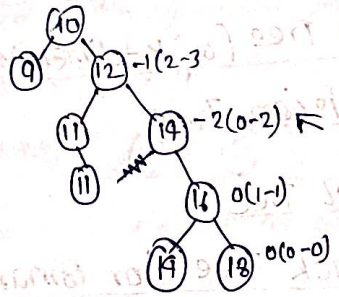
$2^{3+1}=16$

#(2) Balanced Tree :-

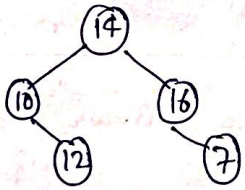
e.g. consider BST



Unbalanced BST



UnBalanced BST



Balanced BST.

⇒ So there are different kinds of balanced trees which are balanced by working with their heights. Some of such trees are as follows:

1. AVL tree
2. B-tree
3. 2-3 tree
4. Red black tree
5. Splay tree

#(3) B-Tree :-

⇒ Binary tree has disadvantage that since it could only have max 2 nodes..

So with number of elements grow, the height of tree can grows as well which makes different operations time consuming.

⇒ Therefore alternative solution is M-way tree in which every node has multiple children.
(M way means M-branches) M-way means multiway.

⇒ M-way search tree have following properties:-

1. Each node contain N-1 key elements, where N is order of tree
2. Each node have N branches
3. It satisfies property of BST. Left small right bigger.

⇒ The disadvantages of M-way tree is it is not Balanced.

⇒ So B-Tree comes in the picture.

So B-Tree is balanced M-way tree.
M-way tree are not balanced.

Fenwick Tree

Level order Traversal in a Tree (specially Binary Trees)

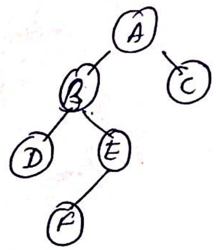
Types

- ① Breadth First Traversal (level order)
- ② Depth First Traversal
  - Pre-order
  - In-order
  - Post-order

Breadth First Traversal (level order Traversal)

- Traverse node in layers.

e.g.



Level-order Traversal → A B C D E F

Recursion is very tough to understand here

Code :- Level-Order Traversal

15

```
void printLevelOrder() {
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++) {
        printCurrentLevel(root, i);
    }
}
```

```
void printCurrentLevel(Node root, int level) {
    if (root == null) {
        return;
    }
    if (level == 1) {          <- now print
        syso(root.data);
    }
    else if (level > 1) {
        printCurrentLevel(root.left, level-1);   // ask the child node to check if their level is 0, or 1 and print.
        printCurrentLevel(root.right, level-1);
    }
}
```

so this way it works when it goes to level 3 it takes two iteration to come to 1 then print.

3-1 → 2
2-1 → 1

so Recursions

see it code in back page also!

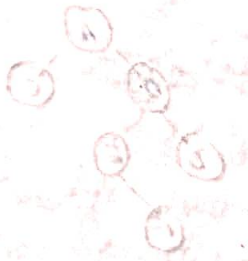
```
int height(Node root) { // Recursive way of finding height ask the leaf node about it's height and add 1.
    if (root == null) {
        return 0;
    }
    else {  int lheight = height(root.left);
            int rheight = height(root.right);
            if (lheight > rheight) {return (lheight+1);} else return(rheight+1)
```

Pre-order :- O(n)

```
Void printPreorder(Node root) {
    if (root == null) {
        return;
    }
    Sysо(root.key);
    printPreorder(root.left);
    printPreorder(root.right);
}
```

Post-order :-

```
Void printPostOrder(Node root) {
    if (root == null) {
        return;
    }
    printPostOrder(root.left);
    printPostOrder(root.right);
    Sysо(root.key);
}
```

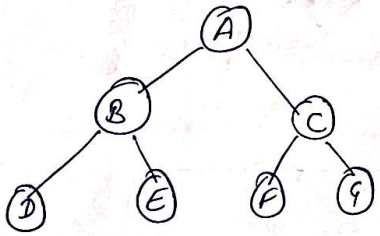


Tree Traversal :- In Binary Tree

3 ways which we use to traverse a tree

1. In-Order Traversal → L (Left) Root R (Right) (Sorted key in binary tree)
2. Pre-Order Traversal → Root L R
3. Post-Order Traversal → L R Root

eg



In-Order → D B E A F C G

Pre-order → A B D E C F G

Post-order → D E B F G C A

Codes in Java :- using Recursion

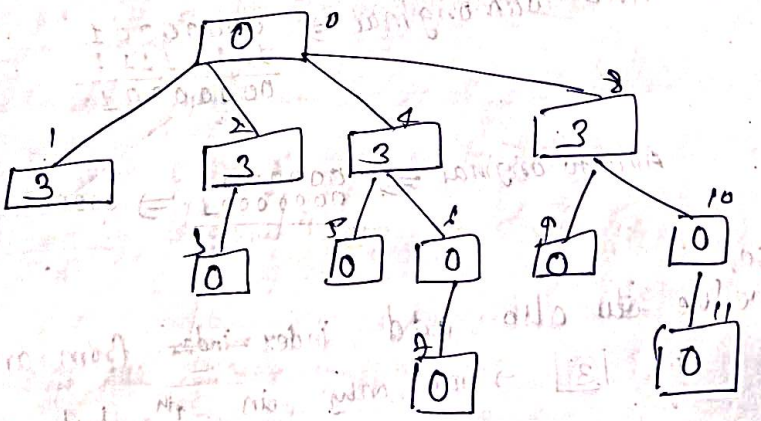
n → number of nodes

⇒ InOrder :- O(n) Traversal always in all three

```
// Given a binary tree, print its nodes in inorder
void printInOrder(Node root) {
    if (root == null) {
        return;
    }
    // first recur on left child
    printInOrder(root.left);
    // then print the data of node
    Sys0(root.key);
    // now recur on right child
    printInOrder(root.right);
}
```

Tree Traversals

⇒ So we have added arr[0] in our Binary Indexed tree.



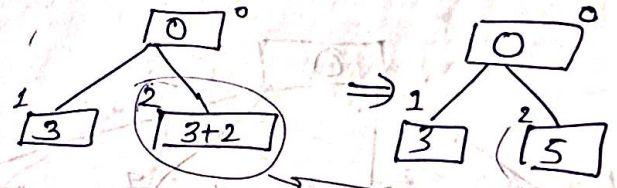
⇒ So amount of time to update this is O(logn). To see how many nodes has been updated.

⇒ So now come to arr[7] = 2

index = index + 1

so BIT index = 2.

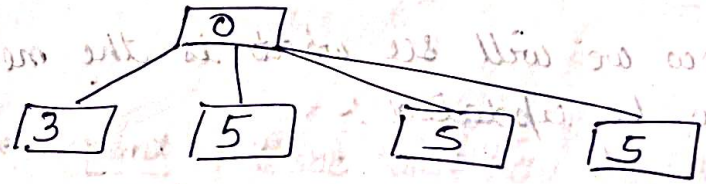
So



So now we also have to update next of '2'.

next of '2' is '4'

next of '4' is '8'. so we get as :-



If there is already element present then just current array value add in that node.

⑤ Now how to fill Binary Index tree from Array in efficient time :-

⇒ Get Next (This is used for filling array values in the tree, we can fill each value in multiple nodes in tree so for that we use GetNext function)

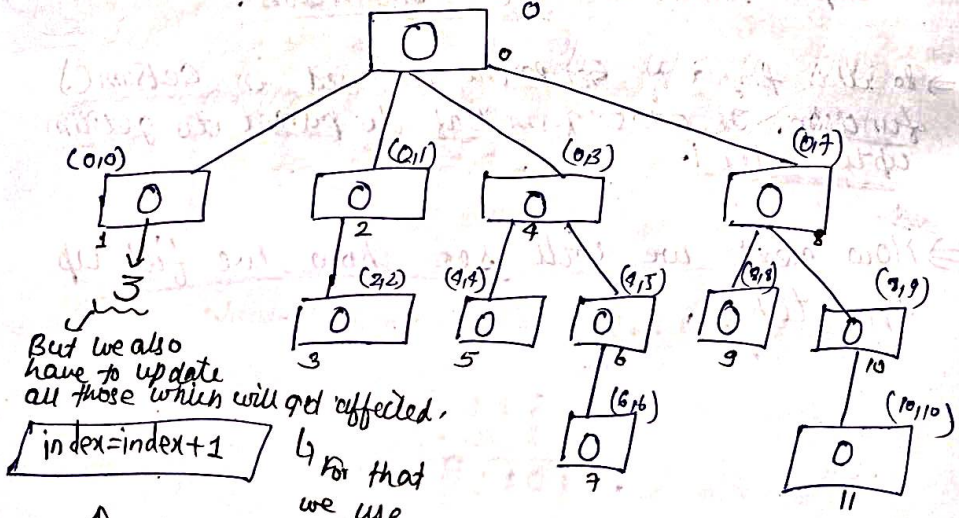
1. 2's complement
2. AND with original number
3. Add to original number.

In code as

```
Index += index & (-index);
```

```
index = index + (index & (-index));
```

⇒ Originally our BITree will be all storing '0' values so now we've to fill our array in it..



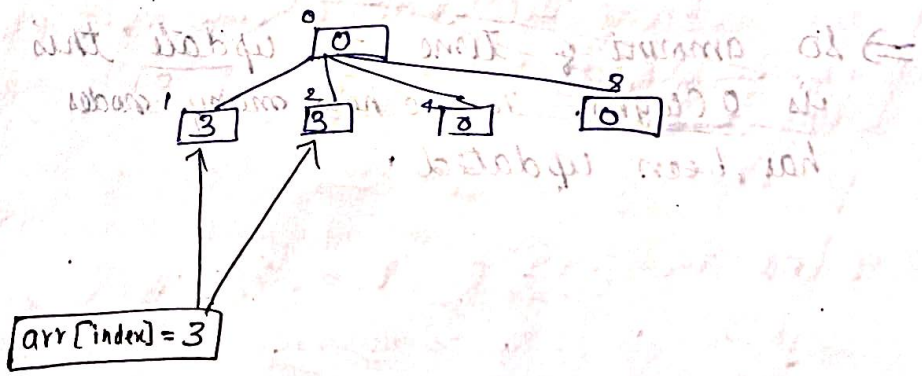
⑤ so index = 1 $\xrightarrow{\text{binary rep.}}$ 00000001

2's complement → 11111110 + 1 → 11111111

AND with original ⇒ 00000001 & 11111111 = 00000001

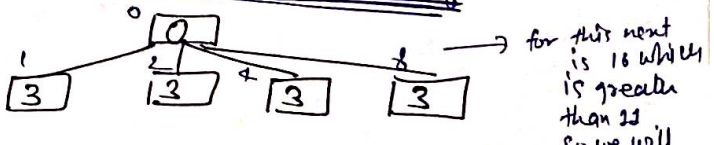
ADD to original ⇒ 00000001 + 00000001 ⇒ 00000010 ⇒ '2'

so, we've to also add index ~~-index~~ from arr i.e 3 → not only in 1st but also in 2nd position also



Also we again have to see if there exist next for 2 or not.

so using Get Next we get 4 as well as 8 as next for 4. so this way we fill.



⑧ So Now How To Get Parent (In coding) :-

Get Parent :-

① 2's complement
② AND it with original number
③ Subtract from original number.

code as

so in code this is doing 2's comp.

```
index -= index & (-index);
```

```
index = index - (index & (-index));
```

e.g. to get parent of index 7.

Step 1:
'7' ⟶ 111
1's complement (Just flip the bits) ⟶ 000
2's complement (Just Add 1) 000 + 001 = 001
so

2's complement = 000 + 1 = 001.

Step 2: Now AND with original 111.

∴
```
 1 1 1
 0 0 1
-------
 0 0 1
```

Step 3: Now subtract this result from original

So :-
```
 1 1 1
-0 0 1
-------
 1 1 0
```
⟶ '6'

∴ 7's parent is '6'.

for '10' ⟶ 1010

2's complement = 0101 + 1 = 0110

AND :-
```
 1 0 1 0
 0 1 1 0
---------
 0 0 1 0
```

Subtract =
```
 1 0 1 0
-0 0 1 0
---------
 1 0 0 0
```
⟶ '8'.

∴ 10's parent is 8.

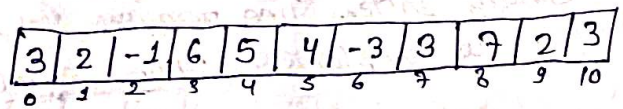
⇒ So to get parent we've to do this three steps which is O(1) operation.

⇒ So this formula of Get Parent is used in GetSum() function. It adds from leaf to parent to getsum upto index i.

⇒ Now next we will see how we fill up the table.

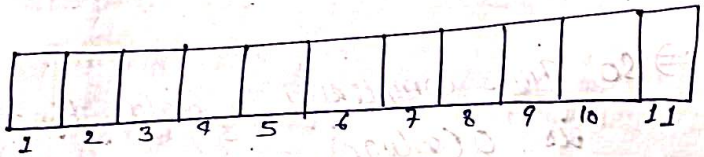
(E) Implementation of fenwick Tree :-

e.g. given input array

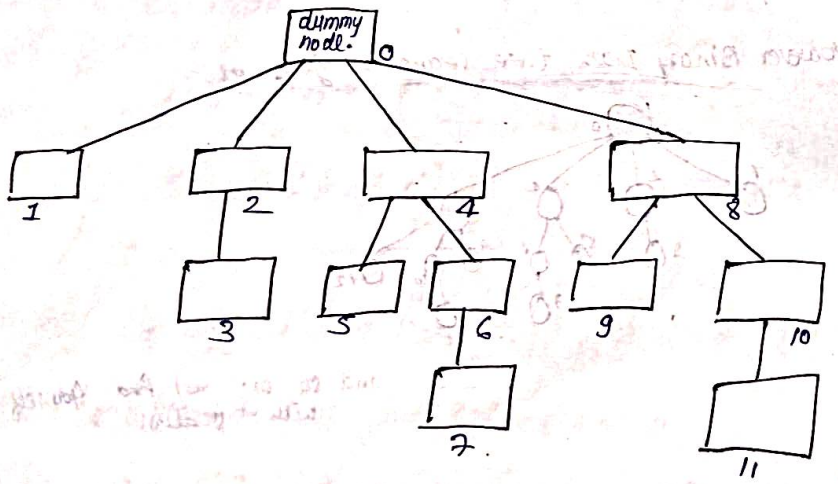


← input array.

BITree[] ⇒



= Our conceptual fenwick tree will look like.



⇒ '0' is dummy node, it does not store any information. From 1 to 11 will store the prefix sum.

⇒ Now why our BITree is like this. i.e. why '0' is parent of '2'.

Parent so

so take '2' (10) —flip rightmost set bit→ 00 → '0'

for '8' (1000) —flip rightmost set bit→ 0000 → '0'

for '10' (1010) —flip rightmost set bit→ 1000 → '8'

for '11' (1011) —flip rightmost set bit→ 1010 → '10'.

50 ☆☆☆

⇒ so now it's clear why our tree looks like this.

Fenwick Tree or Binary Index Tree

Let us consider a problem

we have an array[0.. n-1]; we would like to

1. Compute the sum of the first i elements
2. Modify the value of a specified element of the array arr[i] = x where 0<=i<=n-1.

⇒ So a very very better soln is Binary Indexed Tree, which achieves O(logn) time complexity for both operations.

⇒ Binary Indexed Tree is represented as an array. Let array be BITree[].

⇒ Number of functions do be implemented

1. constructBITree(int arr[], int n)
2. updateBIT(int n, int index, int val)
3. getSum(int index) → from index 0 to some index 'index'.

⇒ Also if you want to range sum i.e. from index l to index r then we use as

rangeSum(l,r) = getSum(r) - getSum(l-1) ← like in segment tree.

⇒ This can be also solved by using segment tree.

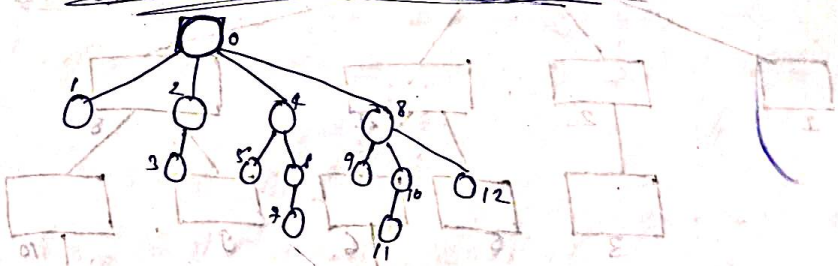
⇒ But since Binary Index Tree requires less space and is easier to implement than segment tree.

⇒ So Time complexity to build Binary Index tree is O(nlogn).

⇒ Time to getSum = O(logn)

⇒ Time to update query = O(logn).

⇒ However Binary Index tree sequence goes as:-



— and so on, rest find yourself sequence with verification.

BST Tree Implementation in JAVA

⇒ Output :- 9 → before update query sum upto 4

11 → after update query sum upto 4.

code for range sum will be as :-

```
static int rangeSum (int l, int r) {
    |
    |
    return (getSum(r) - getSum(l-1));
    |
    |
}
```

e.g. int[] arr = {1,2,3,4,5,6,7,8,9};

(It's always inclusive)

upto index=2

Syso(getSum(2)) // output = 1+2+3 = 6

~~int~~

(for getting index 2 to 5) // 3+4+5+6.

int ans = getSum(5) - getSum(2-1);

Syso(ans) // output = 3+4+5+6 = 18.

18th sep 2022

Ok now understood Completely How BIT tree or FenwickTree is build and how we are updating and How we are getting range sum.

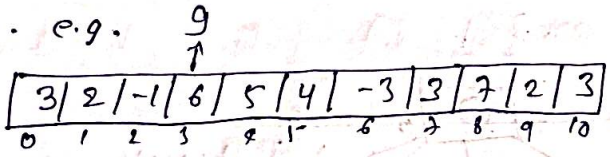
⇒ so now we also know how get sum works.

Overall code for BITree as :-

```
public class Main {

    final static int MAX = 1000;
    static int[] BITree = new int[MAX];

    // Returns sum of first n numbers
    static int getSum(int index) {
        int sum = 0;
        index = index + 1;

        while (index > 0) {
            sum += BITree[index];
            index -= index & (-index);   // get parent node
        }
        return sum;
    }

    // Updates a node in BIT
    static void updateBIT(int n, int index, int val) {
        index = index + 1;

        while (index <= n) {
            BITree[index] += val;        // add val to current node
            index += index & (-index);   // get next node to be updated
        }
    }

    // Constructs BIT for given array
    static void constructBITree(int arr[], int n) {
        // Initialize BITree as 0
        for (int i = 1; i <= n; i++) {
            BITree[i] = 0;
        }

        for (int i = 0; i < n; i++) {
            updateBIT(n, i, arr[i]);
        }
    }

    public static void main(String args[]) {
        int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
        int n = freq.length;
        // Build fenwick tree from given array
        constructBITree(freq, n);
        System.out.println("Sum of elements in [0, 4] is " + getSum(4));
        // Let use test the update operation
        updateBIT(freq.length, 1, 2);
        System.out.println("Sum of elements in [0, 4] is " + getSum(4));
    }
}
```

⇒ So now we will see how to update a value.

For e.g.

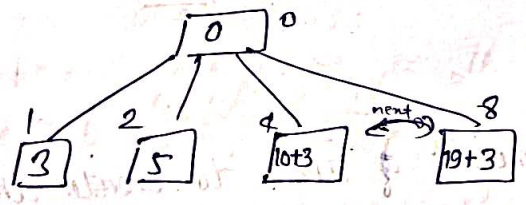


e.g. value 6 at 8 becomes 9

so the difference b/w them is 9-6=3.

So we've to add 3 update in our BITree.

so we will start from Index = Index+1 = 4.



So the amount of time it will take to update is O(log n).

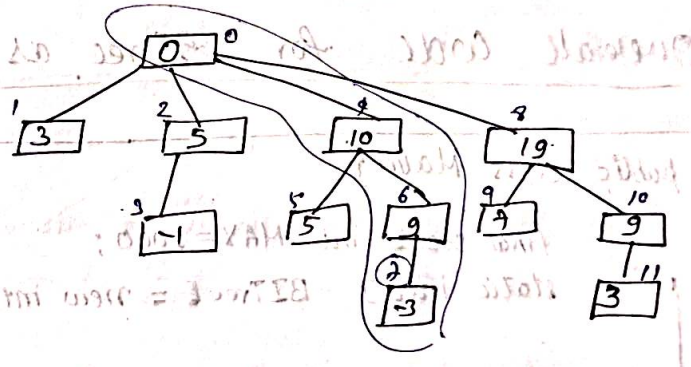
9's next '8' is 8 also add +3.

8's next out of range so stop.

so this way update works.

(3) Now our way to getSum :-

our BITree as :-



for getSum upto '7'.

sum=0

goes to node 7 ⇒ -3 ⇒ sum = sum+(-3)

it's parent (get parent) i=i-i&(-i)

6 ⇒ 9 ⇒ sum = sum+(-3)+(9)

it's parent (get parent)

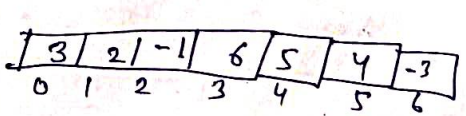
4 ⇒ 10 ⇒ sum = sum+(-3)+(9)+(10)

parent

0 ⇒ 0 ⇒ sum = sum+(-3)+(9)+(10)+0

∴ Ans = 19-3 = 16.

∴ sum upto 7



so.

+3 6+5=11

+2 4-4

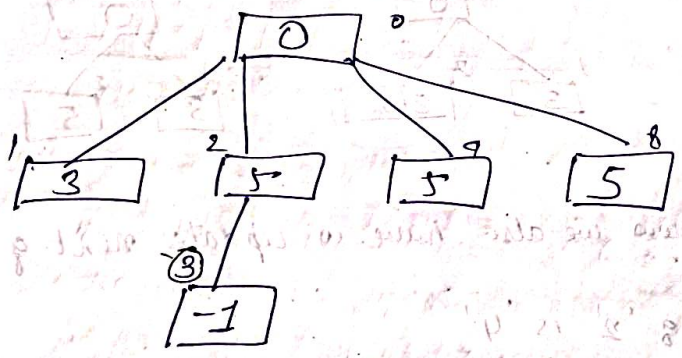
5 0

⇒ 16

ans ⇒ so now come to arr[2]=-1

index=index+1

index=3 in BIT



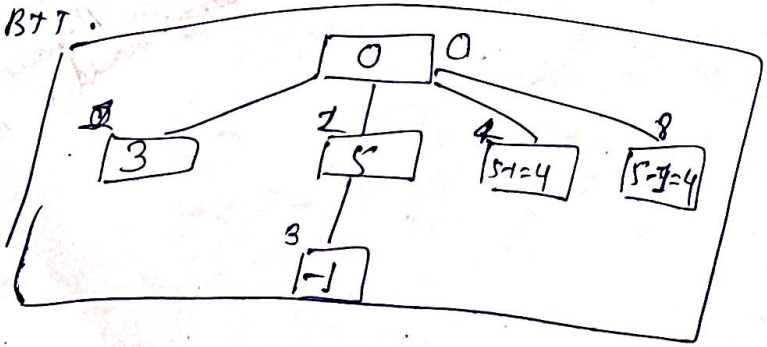
now we will see what is the next number to be updated :-

So next of 3 is → '4'

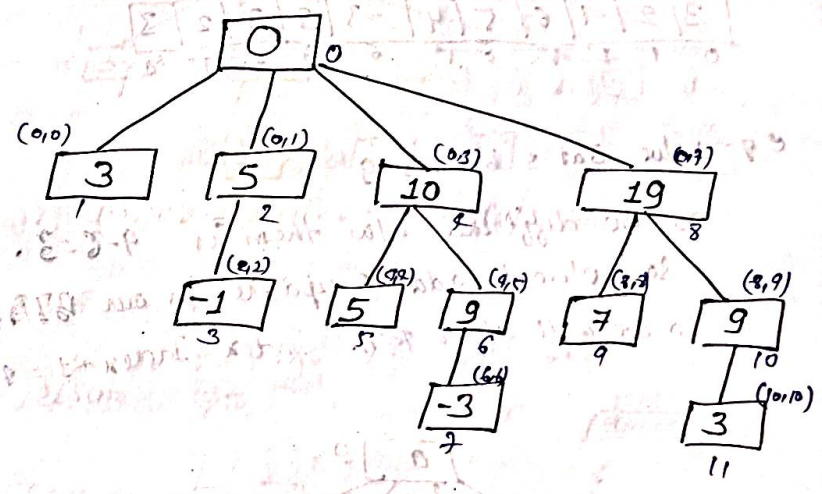
& 4's next is → '8'

So we've to add -1 in node '4' & '8' also. alsosince '8' next is outside range so we stop.

∴ our BIT.



And so in this way our BIT tree is updated in its array and our final BIT tree will be as :-

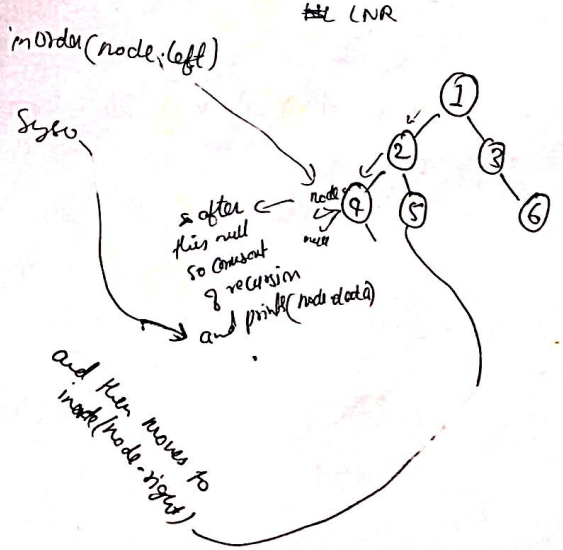


⇒ so the amount of time to create this Binary Index tree is O(nlogn).

```
//Here we require no display() or show method. Instead we use traversal.

//Traversal in BST.

// Inorder Traversal
public void inorder(Node node){
    if(node==null){
        return;
    }
    inorder(node.left);      //It will go to extreme left until node.left=null and then return; and then prints the value as
    Syso(node.data+" ");                                    and so on it works.
    inorder(node.right);
}

// Preorder Traversal
public void preorder(Node node){
    if(node==null){
        return;
    }
    Syso(node.data);
    preorder(node.left);
    preorder(node.right);
}
```

```
// Postorder Traversal
public void postorder(Node node){
    if(node==null){
        return;
    }
    postorder(node.left);
    postorder(node.right);
    Syso(node.data);
}
```

LNR



```
//Delete method in BST
public Node delete(Node node, int val) {
    if (node == null) {
        return null;
    }

    if (val < node.data) {
        node.left = delete(node.left, val);
    } else if (val > node.data) {
        node.right = delete(node.right, val);
    } else {
        if (node.left == null || node.right == null) {
            Node temp = null;
            temp = node.left == null ? node.right : node.left;
            if (temp == null) {
                return null;
            } else {
                return temp;
            }
        }
        // for the case if node we are going to delete has two childs init.
        else {
            Node successor = getSuccessor(node);
            node.data = successor.data;
            node.right = delete(node.right, successor.data);
            return node;
        }
    }
    return node;
}

public Node getSuccessor(Node node) {
    if (node == null) {
        return null;
    }
    Node temp = node.right;
    while (temp.left != null) {
        temp = temp.left;
    }
    return temp;
}
```

Deep going recursively deep inside the tree unless that node is found.

Now here is how we are going to delete that node.

Here temp will take node.left or node.right for each condition. But if both condition false then takes no value and temp = null only.

mean node with no child

mean node with single child.

And so on.

Now we need to delete that duplicate value from the right subtree. so node.right.

cont.

(23) In BST.java :-

```
class BST {

    //creation & insertion of node in BST method.
    public Node createNewNode(int k) {
        Node a = new Node();
        a.data = k;
        a.left = null;
        a.right = null;
        return a;
    }

    public Node insert(Node node, int value) {
        if(node == null) {
            return createNewNode(val);
        }
        if(val < node.data) {
            node.left = insert(node.left, val);
        }
        else if(val > node.data) {
            node.right = insert(node.right, val);
        }
        return node;
    }
}
```

(root we are passing)

500

//while going in deep recursion when reached at point where we need to add that node just add there this node with value

But we need to also keep track of which node it has reached so. it's for further if we've to traversed then so.

This will simply insert at node.left.

otherwise we could have lossed it without assigning to any node which is returning

It's taking the subtree node and so in that it will manipulate and so on.

returns that same above passed root by updating it so)

cont. next page:

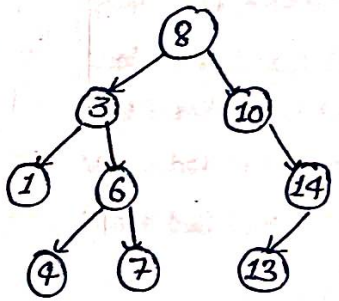
(2) In Main class :-

```
public class Main {

    P  S  V  M {

        BST bst = new BST();
        Node root = null;

        root = bst.insert(root, 8);
        root = bst.insert(root, 3);
        root = bst.insert(root, 6);
        root = bst.insert(root, 10);
        root = bst.insert(root, 4);
        root = bst.insert(root, 7);
        root = bst.insert(root, 1);
        root = bst.insert(root, 14);
        root = bst.insert(root, 13);

    }

}
```

8 Binary Search Tree Implemented

8.1 In Node.java :-

```
class Node {
    int data;
    Node left;
    Node right;
}
```



our resultant tree after insert

⇒ **BST for PSS**

→ Inorder Traversal of BST gives sorted manner.