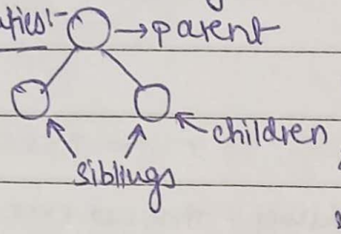
Ajanta
Page No.
Date 26-06-25

Binary Trees and BST

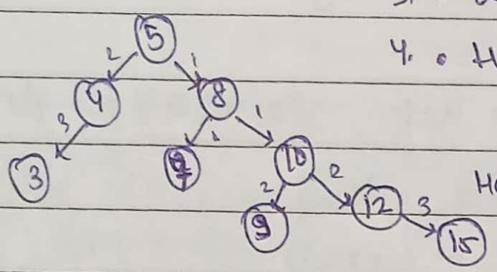
Implementation & Top use of recursion

- Optimind approach → O(logN) ↳ except skewed = LL
- BST → small = left ; big = right
  - ↳ skewed for sorted data ∴ AVL is used.

Properties:-



1. size of tree = total no. of nodes
2. edge = lines connecting nodes
3. leaf = last nodes
4. Height = max. no. of ~~nodes~~ edges b/w that node to leaf



Height of 8 = 3 (max for leaf 3, 15)

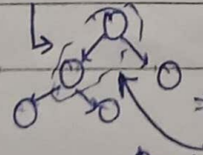
5. • Starting node = ancestor
   • leaf (last) node = descendent
   } → b/w a node to leaf.
6. • Degree = no. of children a node has (leaf node degree is 0).

Types:-

1. Complete : all levels filled apart from last level + last level filling from left to right
2. Full/Strict : either 0 or 2 children of any node.
3. Perfect : all leaf on same level & all levels are filled.
4. Balanced : when avg. ht. is O(logN) → opp. of skewed → eg. ~~BST~~, AVL
5. Skewed : every node has 1 child.
6. Ordered : when every node has some properties, e.g. BST.

Observations:-

- (take eg. & check yourself) Total nodes of perfect BT of height H → $2^{(H+1)}-1$.
- For perfect B.T. → total no. of leaves = 2^H.
- " Strict " → " " " " = internal nodes + 1



⇒ leaf = 3
Proof. ⇒ internal = 3-1 = 2

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
int t = sc.nextBoolean();
int f = sc.nextFloat();
```

Page No. _____ Date _____

* Implementation:- (Binary Tree) → import java.util.Scanner;

Vicary → Insert → Display

import scanner coz you gonna use it

Basic Declaration:- public class BinaryTree {

class Node {

① Diff. from LL this node has left & right rather next.

- int val; → same
- Node left;
- Node right;

Constr of val.

defined node of tree needs declaration (not general, a constant ~~is~~ ~~used~~ ∴ declared) ← private Node root;

→ How beautifully recursion tree is made; just understand thoroughly

* Insertion:-

a. • Firstly it will take root only & itself will call for further insertion via method overloading / may use different method name too!

• Takes first value from user

return type void; arg → Scanner sc

① public void insert (Scanner sc) {
② print ("enter root node"); // Integer type
③ int val = sc.nextInt();
④ root = new Node(val); ← make a new node & put that val in it as root.
⑤ insert (sc, root); ← [first value as root reserved]

b. • Further insertions based on left or right of root.

① public void insert (Scanner sc, Node node) { ← root will be passed initially than next nodes
② print ("enter left of:", node.val);
③ boolean left = sc.nextBoolean();
④ if (left) { → (if user write true after this only)
⑤ print ("enter left of:", node.val);
⑥ int val = sc.nextInt(); ★ thought process checked ✓
⑦ node.left = new Node(val);
⑧ insert (sc, node.left); ← this will now carry forward

repeat from ② for right!

all in easy order no need to remember

Display

a. first root then carry forward

return type = void , no arg.

* [this.root why?] display (this.root, " -> ") ;

↓

first one passed will be root in display(node, indent) → display will be passed from tree initiated by the root.

b. All display including root as start (overloading)

return type = void, arg = node, String indent

① if (node == null) return;

Displaying based on insertions.

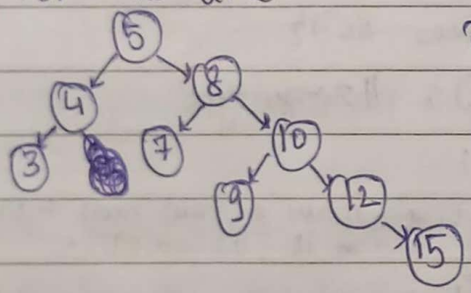
② print (node.value, indent);

③ left → display (node.left, indent);

④ right → " (" right ");

think recursively as tree

For this tree :-



Displayed as: 5 4 3 8 7 10 9 12 15

↓

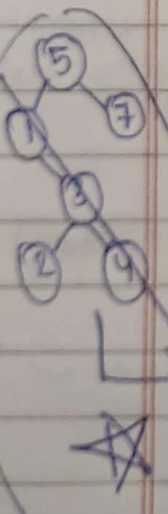
(N-L-R)

(Pre-order)

For these 3 steps :- (Traversal methods implement)

1. Pre-order (N-L-R) : same as in display ②→③→④
2. In-order (L-N-R) : ③→②→④
3. Post-order (L-R-N) : ③→④→②

Learn



for this example Inorder : 1-2-3-4-5-7

" " " Postorder : 1-2-3-4-7-5

" " " Pre-order : 5-1-3-2-4-7

Page No. 8
Date 27-06-25

BST Implementation → for checking only; no need in implementation

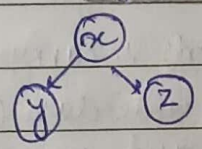
Only height and balance fn. are new here.

- Basic Declaration → only diff. is a new data member of int height inside Node class rest same.

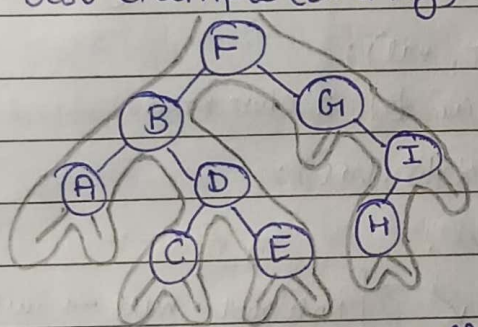
Concept :- * in order traversal of BST always gives sorted seq.
* Search in BST comprises of O(H) = O(log n)
(H = height)

Traversal method - Pre, In, Post order - correction and clarity

Pre-order - ~~Left~~ Root left right : x y z
Inorder - Left root right : y x z
Postorder - Left right root : y z x



Best example (clarity):



Preorder - F B A D C E G I H
Inorder - A B C D E F G H I
Postorder - A C E D B H I G F

Based on Priority

method-2 : To write all 3 directly → Kisi bhi node pe pehli baar pahuche Preorder
dusri baar = Inorder
Teesri baar = Postorder

Preorder - F B A D C E G I H
Inorder - A B C D E F G H I
Postorder - A C E D B H I G F

29-06-25

* Insert in BST
* Search in BST
* Delete in BST
* Print in range → direct use of concept
* Print Path
* Inorder display

Page No.
Date 29/06/25

Basic Declaration :-

```
public class BST {
    class Node {
        int data;
        Node left;
        Node right;
        constr of data only
```

that's it!

Insert :-

return type = Node , arg = Node root, int val

∵ since starting with root

reason? = null check karna aur update krna

∵ inserting val.

each insertion starts from root

① if (root == null) { → ∵ initialised root with null to place first val as root
    root = new Node(val); → first val gets inserted in root
    y return root; → ∵ return type Node ∴ carry on after root

② concept of BST :- ∵ data member of Node class (not always val, understand conceptually)
    if (val < root.data) { → ∴ left

    y root.left = insert(root.left, val);

③ else
    root.right = insert(root.right, val);

④ return root; → return type purposes.

In main :-

```
Node root = null → initialised
val[i] = {5,1,3,4,2,7}
root = insert(root, val[i])
```

Same store hoga since :- DEBUG :-

1. val[0] = 5 → root == null → true ∴ root = 5 ✓
2. val[1] = 1 → " " " → false ∵ jo val root ki update hua wo ~~kisi~~ to root ki doobaara aake store ho gya.

read Debugging line by line tabhi samaj aayega

→ 1 < 5 ∴ left (true)

→ then root.left (null initially) ∴ jo root.left inside insert call hoga wo null hoga (insert(root.left, val)) — talking about this — null initially

∴ goes to first line if (root == null) → true ∴ root = 1 → jo root.left hai

and isse root.left ki ∃ store karado]

root.left = insert(root.left, val) → then return root

in the end ki wapas root se checking start ho.

Idly for right.

Page No.
Date

✓ * Inorder → return type = void , arg = Node root ← ∵ display starting from root.
(LNR)

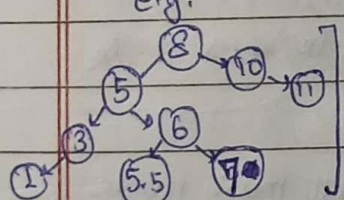
① if root == null ∴ return
② inorder (root.left) ; → left
print (root.data + " ") ; → root
inorder (root.right) ; right.

* Search → return type = boolean , arg = Node root, int key
O(H) ∵ Yes or No (exist or not) ↓ from root start ↓ to be searched

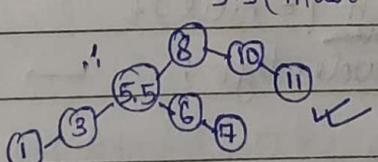
① if (root == null) → return false ;
② if (root.data == key) → return true ;
③ else if (key < root.data) { ∴ left search
~~return~~ search (root.left , key) ;
~~based on this line further it will say true or false~~ ★★★★★
④ dly right for else
~~⑤ return false ; → if not found~~

★ Adobe
* Delete → return type = Node , arg = Node root, int val
Concept : 1. No child / leaf Node → return null to it's parent
think by visualising ↓ 2. One child → replace with child node
BST Tree 3. Two children → " " inorder successor
and further inorder " can be deleted by 1 or 2.

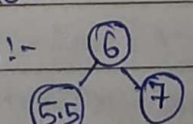
e.g.



→ To delete 5 → inorder successor = 5.5 ∴ replace with 5.5 then delete 5.5 (must die in case 1 or 2 only)



★ Note :- Inorder successor is always the left most node of right sub-tree for that node

Here, right subtree of 5 :-  leftmost in this → 5.5 ✓

★ Hence, get the inorder successor always from right.

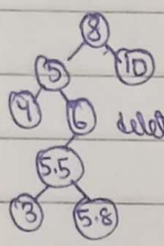
Ajanta
Page No.
Date

① if (val > root.dat){ → i. delete from right subtree
root.right = delete(root.right, val);
isme wapas store kara rahe hai as again root for right subtree proceed karo.
}

② else if (val < root.data){
root.left = delete(root.left, val);
} similarly left

Then,

③ else {
// case 1 (NO child) → if(root.left == null && root.right == null){
return null; → return that node as null
// case 2 (1 child)
in left only → else if (root.right == null){
return root.left;



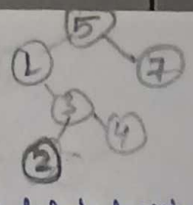
i. replace with 5.5

in right only → ✓
// case 3 (2 child)
get inorder successor from right subtree (reason page before)
Node IS = findIS(root.right);
root.data = IS.data → (replaced) ✓
then delete this IS from its older position
root.right = delete(root.right, IS.data); ★
}

④ return root;

Function → findIS : return type Node, arg Node node

① while (node.left != null){
node = node.left; → left most node.
}return node;



← 5, 1, 3, 4, 2, 7

insertion (each) starts from root

Ajanta

Page No.

Date

Q.1 Print in range → get inorder ~~delete~~ & print from particular to particular

Q.2 Print Paths in BST from Root to leaf.

printRoot2Leaf → return type = void, arg = Node root, ArrayList<Integer> path.

for storing all paths one by one.

① if (root == null) → return;

② path.add(root.data);

③ leaf → if (root.left == null && root.right == null){

then print ~~all~~ path if reached till leaf → printPath(path);

④ Check for right & left :-

printRoot2Leaf (root.right, path); } → proceed further

" " " (" left ");

⑤ start removing from end to check for further paths from second last, if not then third last, proceed.

path.remove (path.size() - 1);

Func. printPath → return type void, arg = ArrayList<Integer> path.

```
for (int i=0; i<path.size(); i++){
    print (path.get(i) + "->");
}
println("Null");
```

✱ Main function :-

```
psvm {
BST bst = new BST();
int[] val = {5, 1, 3, 4, 2, 7};
```

initialise null to ← Node root = null;

start inserting

```
for ( i=0 to val.length ){
    root = bst.insert (val[i]);
}
```

root,

```
bst.inorder(root);
println(bst.search(root, 2));
bst.delete(root, 4);
bst.inorder(root); → check purpose
bst.printRoot2Leaf(root, new ArrayList<>());
}
```