01 07 24

Parent with 2 child

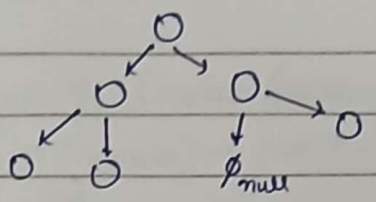
TREES

★★★ V.V.V.I

One node has two pointers

Why? → Trees

We use trees for efficiently inserting and deleting.



Complexity → O(log N).

Ordered insertion & deletion storage

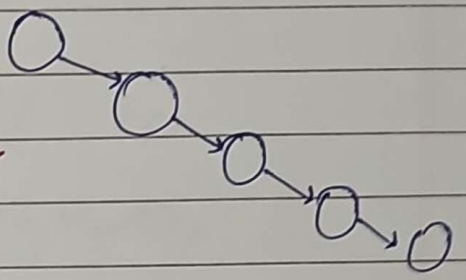
cost efficient

Unbalanced binary tree →

O(n)

∵ checking all node to find a particular node.

[Left side pointing is null for all]

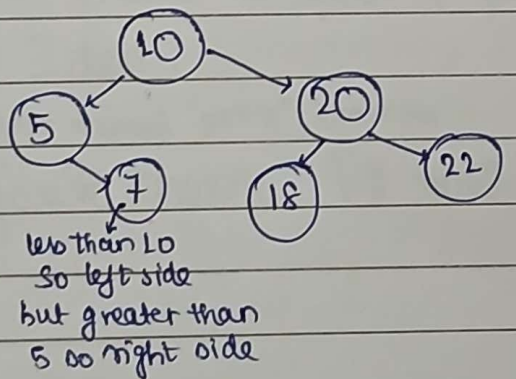


Binary Search Tree

Smaller than Left
Bigger than right

→ To its respective items step by step.

e.g.

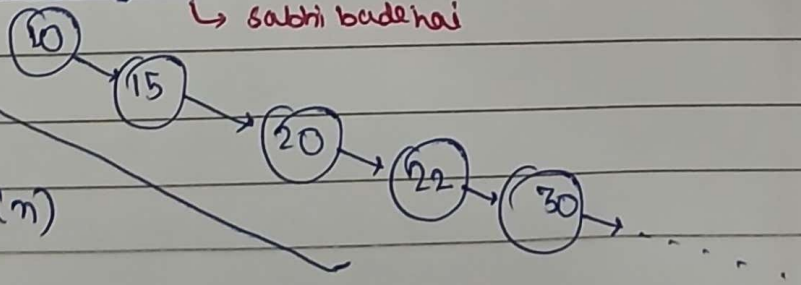


less than 10 so left side but greater than 5 so right side

here unbalanced binary tree arised as :-

↳ sabhi badenai

Problem



32 16 8 4 2 1
0 1 1 1 0 1 → 1 0 1 1 1
1 2 0 16 4 2 1

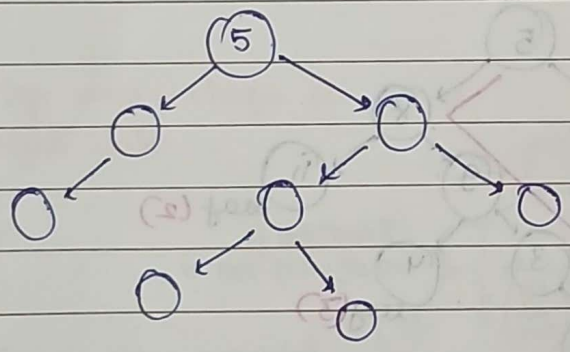
Date : / /
Page :

- How to maintain the tree balanced all the time
  → Self balanced Binary Tree (like AVL, etc. next lecture)

- another limitation of B.T. is it is inefficient for sorted datas.

- Where it is used?
  - File systems → every node represent parent child relationship as files.
  - Databases
  - Algorithms / Computer Networking
  - Mathematical equations
  - Decision Trees → machine learning algorithm
  - Compression of files
  - Future Data structures → Heaps, etc.

  Note: Tree is a type of Graph.



like in linkedLue were having

```
Node {
    int value;
    Node next;
}
```

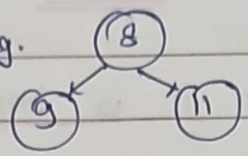
Similarly here we have in Binary Trees :-

```
Node {
    int value;
    Node left;
    Node right;
}
```

root node: node that doesn't have parent.

***. Properties:- OF BINARY TREES :

(1) Size = Total no. of nodes

(2) If e.g. 

then 9 and 11 are children of parent 8.

(3) And, if any two nodes have similar parents then they are sibling. e.g. here 9 & 11.

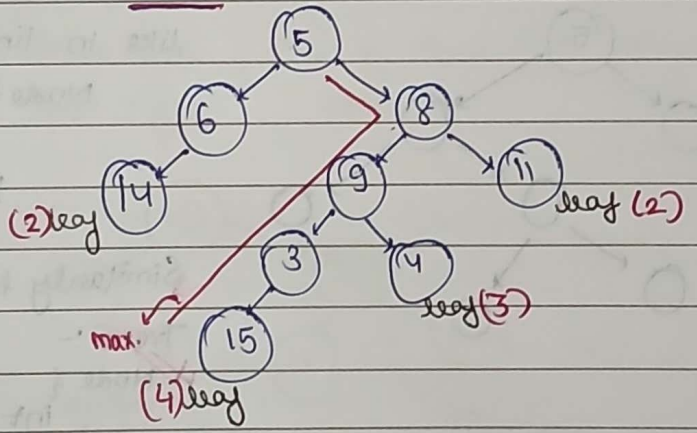
(4) Edge : lines that connects 2 nodes together

(5) Height : Maximum no of edges b/w 2 nodes.

(6) Leaf : bottom nodes or last nodes.

∴ [Height of Tree] → max. no. of edges from the node & leaf node.

e.g.



e.g. Height of 5 = Max (2, 4, 2, 3) = 4 ans.

Height of 8 = max (1, 2, 3) = 3 → check from every option

(7) Level : difference of height b/w that node & root node.

or basically → [(height of root − height of node)]

Note:- root level = 0. (obviously). ∵ ($H_r - H_n = 0$) where $H_n = H_r$

e.g. level of 8 = 4 - 3 = 1.

(8) Ancestor and descendant (read only underlined)

When you choose a path from a node to a leaf node or any node after that then starting node will be ancestor & other nodes in the path or last one in path will be its descendant.

e.g. in path (8 → 9 → 4) → 8 = ancestor, 4 = descendant.

(9) Degree → no. of children you have e.g. 5 has degree of 2, 6's degree = 1.

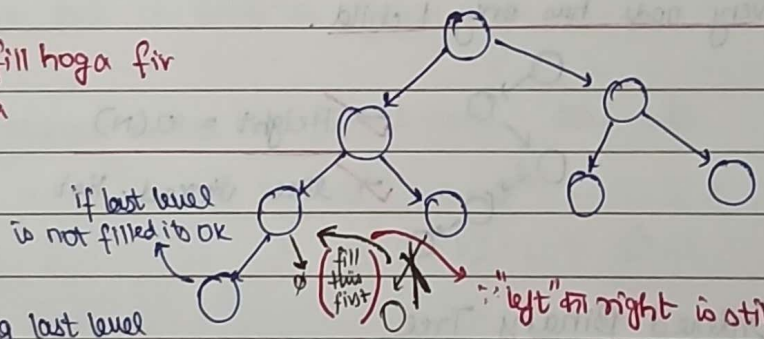
leaf degree = 0

$0 \leq \text{Degree} \leq 2$.

★ Types of Binary Tree :-

(1) Complete Binary Tree

In this tree, all the levels are filled apart from last level but the last level is filled from left to right.

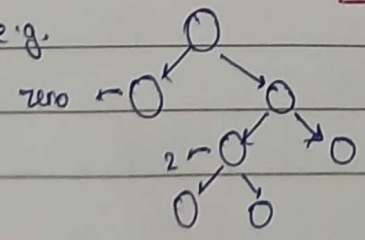


But if filling last level it must be from left to right.

(2) Full Binary Tree / Strict Binary Tree

Each node has either 0 children or 2 children [no 1 children. (ignore english). i.e. siblings or nothing.

e.g.



Used cases:- compression, Hoffman coding, segment trees.

↳ imp. for competitive programming but not for interview prep.

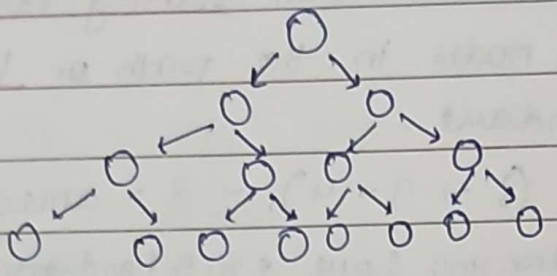
(3) Perfect Binary Tree

"nothing is perfect in life, apart from Perfect Binary Trees", -KK

→ all the internal nodes have 2 children & all the leaf nodes are on the same level. → Different from Full/strict BT

→ basically → all levels are filled.

e.g.



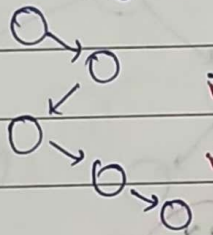
(4) Height Balanced Binary Tree

→ When Avg. Height of B.T is $O(\log N)$ where N = no. of nodes.

e.g. AVL trees.

(5) Skewed Binary Tree

→ Every node has only 1 child.



→ Height = O(N)

→ like linked list.

(6) Ordered Binary Tree

→ Every node follow some properties

e.g. Binary search Trees → properties = left smaller, right bigger

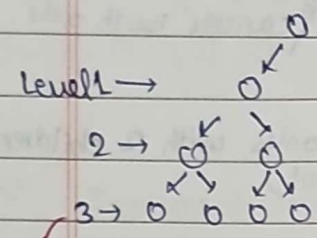
- Important points that help in solving Questions

(1) Total no. of nodes in a perfect binary tree of height 'H' is equals $[2^{(H+1)} - 1]$.

Perfect B.T. is the tree that takes max. no. of nodes.

[Total nodes in Perfect B.T. = $2^{(h+1)} - 1$]

where h = height of tree.



$\rightarrow 1 = 2^0$

$\rightarrow 2 = 2^1$

$\rightarrow 4 = 2^2$

$\rightarrow 8 = 2^3$

16

32

for perfect BT max left = height (obviously)

if level = i ⇒ no. of nodes = 2^i

$\therefore 2^0 + 2^1 + 2^2 + \ldots + 2^h \rightarrow$ sum GP

$\frac{1 \cdot 2^{(h+1)} - 1}{2 - 1} = \boxed{2^{(h+1)} - 1} = N$

Total nodes

$\because S = \frac{ar^{n+1} - 1}{r - 1}$

(2) For perfect B.T. → total no. of leaf node in it = 2^h

$\because$ Total = $2^h \times 2 - 1$ (root)

$2^h + 2^h$ (left, right)

$\therefore$ Internal nodes = Total node − leaf node

$= 2^{h+1} - 1 - 2^h$

$= 2^h(2-1) - 1 = 2^h - 1$

directly all left nodes − 1 $= 2^h - 1$

(3) 1. N = no. of leaves

then $(\log N) + 1$ levels atleast

2. If N = no. of nodes → min. levels = $\log(N+1)$.

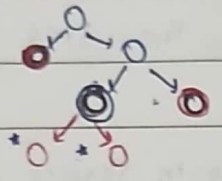
(4) Strict B.T. → total leaf nodes = N

↓ level must not be same

then internal nodes = N − 1

∴ [no. of leaf nodes = no. of internal nodes + 1]

Proof by eg. →



earlier → internal = 2
leaf = 3

final → internal = 3
leaf = 4

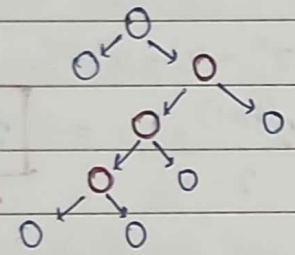
internal = 4
leaf = 2

now internal = 5
leaf = 3

both increased by 1 cancels both side.

(5) No. of leaf nodes = 1 + no. of internal nodes with 2 children (not including root)

check.



~~here internodes~~

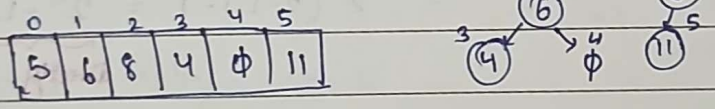
internal nodes with 2 children = 3

leaf nodes = 5

Implementation :-

(1) Linked representation (using pointers) → we know as in LL

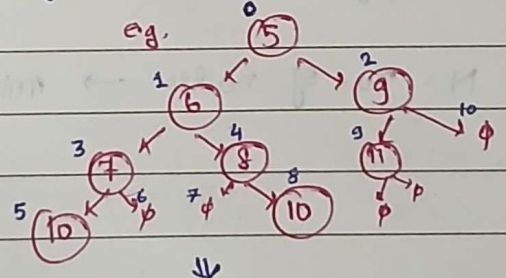
(2) Sequential (using array) →

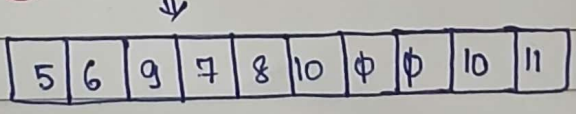


uses

eg. will use in heap → representation of complete B.T. in an array

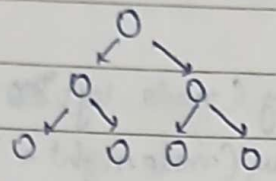
eg. strict Binary tree i.e. segment tree





(ii) ① Description of Need :-

Both qualitative and quantitative description of the need is done. no purchase should be done without the permission from user department.



$N = 2^{h+1} - 1$

for complexity analysis → $N = 2^h$

$\therefore [h = \log_2 N]$

Implementation → main idea using recursion & how stacks are filling & removing & when inserting meanwhile.

(11) steps go deeply & think of each & obj.

- in populate function → public void (Scanner scanner) &-

~~int value = scanner.nextInt();~~ no need!

int value = scanner.nextInt();

Enter the root node: (which now takes I/P from user) → let's say 15

Create a node out of it → root = new Node (value);

↳ ∵ first one ∴ set that to root

now it will call rest of the tree in the recursion function

populate (scanner, root); → private void populate (Scanner scanner, Node node) {

Now in rec. func. → sout ("Do you want to enter left of" + node.value)

Yes → Type ⇒ true (i.e. 15)

for left → boolean left = scanner.nextBoolean();

if (left) {

sout ("Enter left of" + node.value); (15)

~~int value = scanner.nextInt();~~ ⇒ e.g. 6

node.left = new Node (value);

populate (scanner, node.left); ← calling itself

sout ("do you want to enter right of" + node.value);

boolean right = ...

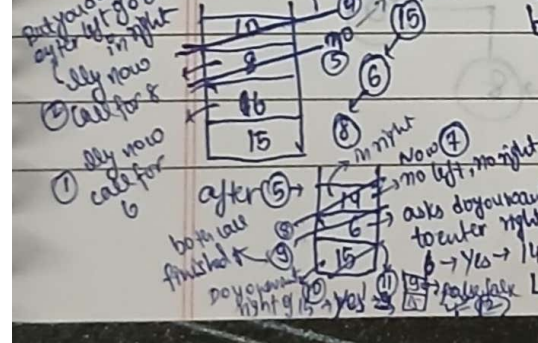
if (right) {

sout (~~Enter value of right of~~ " + node.value) (8)

int value = ... e.g. 10

node.right = new Node (value);

} populate (scanner, node.right); ↳ Now recursion call what you've inserted



• display function →

```
Public void display(){
    display(root, "");
}
```

this.

```
private void display(Node node, String indent){
    System.out.println(indent + node.value);
    if(node == null){
        return;
    }
    display(node.left, indent + "\t");
    display(node.right, indent + "\t");
}
```

indent → just the space providing

base condition :-
no node ∴ return

agar pehle likh dete toh null pointer exception ignore ho jayega.

another ~~same~~ Java file

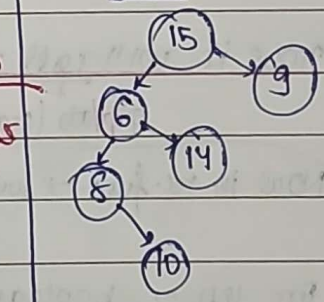
• main →

```
psvm {
    Scanner scanner = new Scanner(System.in);
    BinaryTree tree = new B....();
    tree.populate(scanner);
    tree.display();
}
```

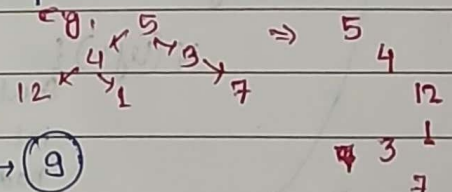
O/P

```
15
    6
        8
            10
    Same level
        14
    9
```

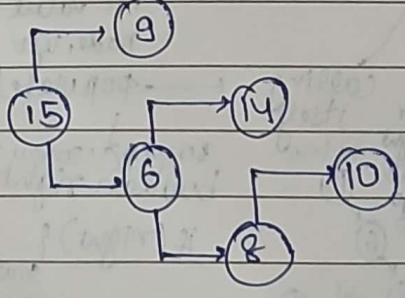
BT.



Confusion from BT → OP is possible but not opp.



• pretty display as :-



→ Pretty display :

```
public void prettyDisplay(){
    prettyDisplay(root, 0);
}
private void prettyDisplay(Node node, int level){
    if(node == null){
    } return;
```

displaying right first:

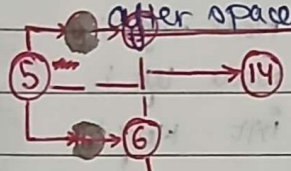
```
prettyDisplay(node.right, level+1);
```

after every level adding 2 spaces :→

```
    if(level != 0){
        for(int i=0; i<level-1; i++){
            sout("|\t\t");
        sout("|------>" + node.value);
    }else{
        sout(node.value);
    }
prettyDisplay(node.left, level+1);
```

don't use "ln" →

after spaces → use ln



recursion →

only do for left.

use tree.prettyDisplay(); rest same in main file.

"Khud se karna seekho, rev. OOPs."

→ things to do in start of codes :-

```
class BinaryTree{
    public BinaryTree(){
    }
```

write imp. points of static :-

```
    private static class Node{
        int value;
        Node left;
        Node right;
```

} same as LL

Constructor where we pass the value helps in storing into int value.

```
        public Node(int value){
            this.value = value
        }
    }
    private Node root;
```

↑ start writing from here from populate.

bcoz root node holds all other nodes of BT

Trees, S+O, LL, OOPs, T+SC, all rest, + Recursion, B Tracking, J.S., CSS, quick HTML ☆ bina nhi soona hai
Date → 27/07/24 + AVL + Heaps & Hashmaps.

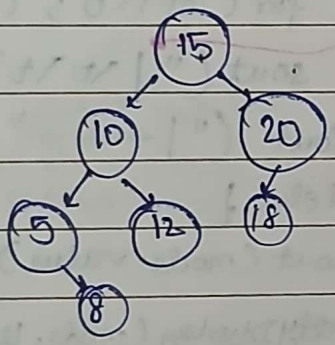
- **Binary Search Tree**

→ for this lecture we consider balanced tree

[any two nodes on same level should have height diff. ≤ 1]

→ main concept left = smaller, right = bigger, always compare from root till leaf.

→ It will then compare 1 item at every level.




```
Node {
    int value;
    Node right;
    Node left;
    int height;
}
```

→ Implementation:-

Initial stuffs:

```
class BST {
    public class Node {
        private int value;
        private int height;
        Node left;
        Node right;
```

considering as the property of the node. ← attributes

node constr ← `public Node(int value){ this.value = value; }`

getters/setters ← `public int getValue() { return value; }`

```
    }
```

declaring root ← `private Node root;`

BST constr ← `public BST(int value){ }`

- Complexity: Total 9 nos.
  for every no. → n^2 possibility.

  Time: $O(9n^2)$, Space: $O(N^2)$

* BST implementation

I. Class BST {

```
public class Node {
    private int value;
    private Node left;
    private Node right;
    "  int/Node height;
```

as we are, BST should be balance ← ∴ using height as property of node.

```
    public Node(int value){ ← passing value to constructor
        this.value = value;
    }

    public int getValue(){
        return value;  → for getting value
    }
}

private Node root;  → first node declaration
public BST(){  → constructor of BST
}
```

II. Height — just returning → height (here we're not considering any logic for height)

```
public int height(Node node){
    if(node == null){
        return -1;
    }
    return node.height;
}
```

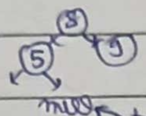
III if root == null → (no node)

```
public boolean isEmpty() {
    return root == null;
}
```

IV. Insert

Now inserting values →

```
public void insert(int value){
    root = insert(value, root);   ∵ returning Node
}
```

eg.


to put 8 here

```
private Node insert(int value, Node node){
    if(node == null){
        node = new Node(value);
        return node;
    }
```

∴ Creating node of certain value →

put value only if node != null

```
    if(value < node.value){
        node.left = insert(value, node.left)
    }
    if(value > " ){
        node.right = " ( " , ". right)
    }
    node.height = Math.max(height(node.left),
                           "       ". right)) + 1;
    return node;
```

if inserted value is less than value, put on left & check again for left. →

lly, for right →

∵ naya node add hua hai ~~to upar wale~~ ~~मा~~ ~~jo height~~ tha wo + 1. + height left aur ~~right (left)~~ ~~से~~ dekha jaata hai jo max. ho) ← node.height

∵ not changing before ones, but height is changed ∴ ~~adjust~~ height change ← return node;

Note:- A balanced binary Tree is such a Tree whose left + right node (subtree's) heights differ ~~to~~ not more by 1.
i.e. height of diff. → −1, 0, 1 only.

V. Balancing

For checking balanced or not :→

```
public boolean balanced(){
    return balanced(root);
}
```

check if root is balanced ← return balanced(root);

```
private boolean balanced(Node node){
    if(node == null){
        return true;
    }
    return Math.abs(height(node.left) - height(node.right))
           <= 1 && balanced(node.left) && balanced(node.right)
```

already balanced ← if(node == null)

abstract/absolute (if(-1) → 1) ← Math.abs

also check for further left + right nodes/subtrees ← balanced(node.left) && balanced(node.right)

VI. Displaying →

```
public void display(){
    display(this.root, "Root node:");
}
```

↑ giving details/msg.

```
private void display(Node node, String details){
    if(node == null){
        return;
    }
    System.out.println(details + node.value);
    display(node.left, "Left Child of " + node.value + ":");
    "    ("  right, Right  "  "  :  "  "  "  ");
```

don't display anything just return

rootnode: 15 — at starting root node i.e. 15

for left →

illy right →

VII. Multiple insertion

Mention this ~~before~~ after insert (for inserting many values) →

```
public void populate(int[] nums){
    for(int i=0; i<nums.length; i++){
        this.insert(num[i]);
    }
}
```

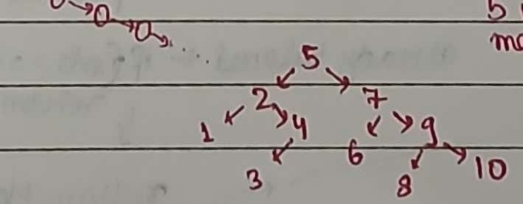
giving an array & then insert on it for multiple times so ~~&~~ we don't have to write again & again at terminal.

VIII Main

at the end → main func. →

```
psvm{
    BST tree = new BST();
    int[] nums = {5,2,7,1,4,6,9,8,3,10};
    tree.populate(nums);
    tree.display();
}
}
```

Problem arises when this is sorted array ∴ will make unbalanced BT.



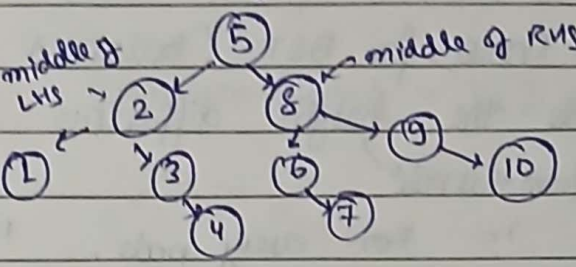
5 is the root node ∴ make BST by own

Sol^n^ ↳ next page

Sol^{n} 1 : Self balancing BT → AVL

Sol^{n} → 2 : take middle element as root node & call LHS & RHS simply.
i.e. using populate LHS & RHS.

eg. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



IX. For sorted array (improved)

∴ new Populate fu. →

```
public void populateSorted(int[] nums){
    populateSorted(nums, 0, nums.length);
}

private void populateSorted(int[] nums, int start, inten
    if(start >= end){
        return;                          // not found ← (base cond$^{n}$)
    }
    int mid = (start + end)/2;
    this.insert(nums[mid]);              // insert the middle element
                                         // then call same for LHS & RHS
    populateSorted(nums, start, mid);    // LHS →  (perform same on this)
    populateSorted(nums, mid+1, end);    // RHS →
```

- Complexity ⇒ $n \times \log n$
  - n : calling n elements
  - $\log n$: inserting

28/07/24

* AVL Trees

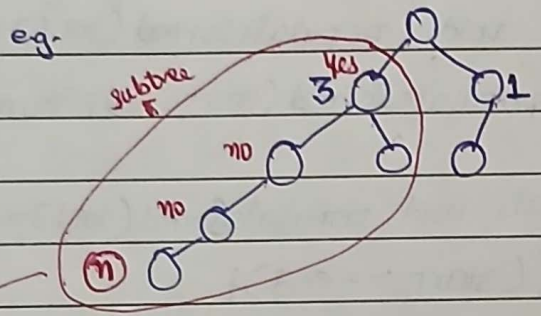
Problem → for sorted node values it takes complexity of O(n) rather BST takes on O(log n).

We know for BST (balanced cond^{n}), for every single node the height difference of left & right <= 1.

max no. of edges from that node to leaf node.

i.e. For every node, H(l) - H(r) = -1, 0, 1 only.

eg.



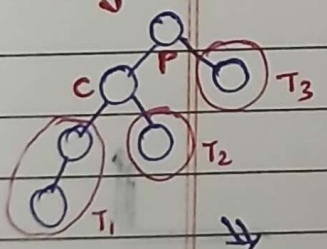
⇒ 3-1 = 2
∴ not balanced

Solution → Self balancing B.T. e.g. AVL

↓

commonsense → restructure the tree as balance when it's creating a scene of unbalanced.

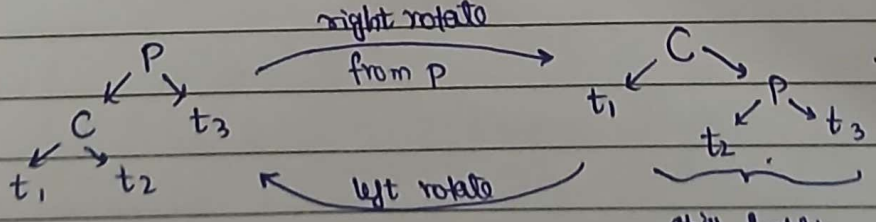
[AVL → Addson-Velskii and Landis]



Algorithm:

① Insert normally node (n)

② Start from node (n) & find the node that makes the tree unbalanced from bottom to up.



parent child
∵ P>C, t2>C but t2<P
t3>P
t1<C

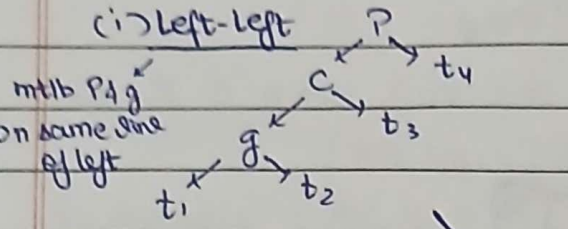
still holding the conditions of BST.

[Sochokaise bara, vvveasy]

③ using one of the ④ rules, rotate.

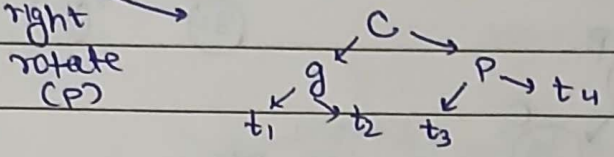
4 rules :-

(i) Left-Left

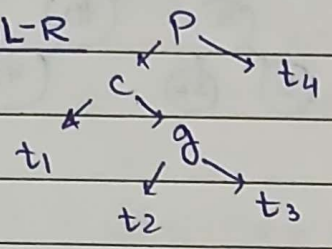


mtlb P, g on same line of left

grandchild (g) will be just before the new node (which changing balance to unbalanced) ① is inserted.
(here item inserted is t_1)

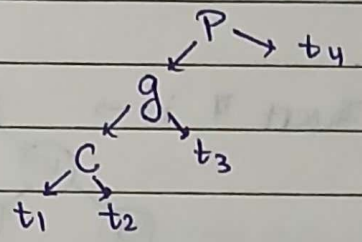


(ii) L-R



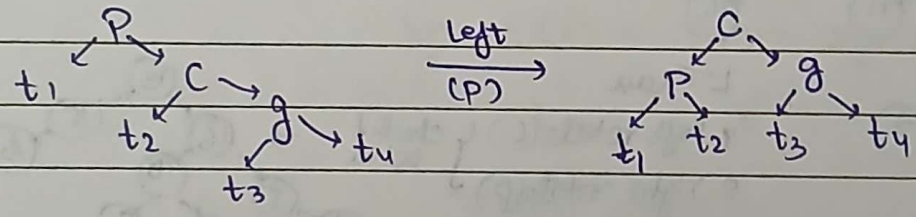
∴ [first make it into a single line]

∴ left rotate (C) ↓ will put g above

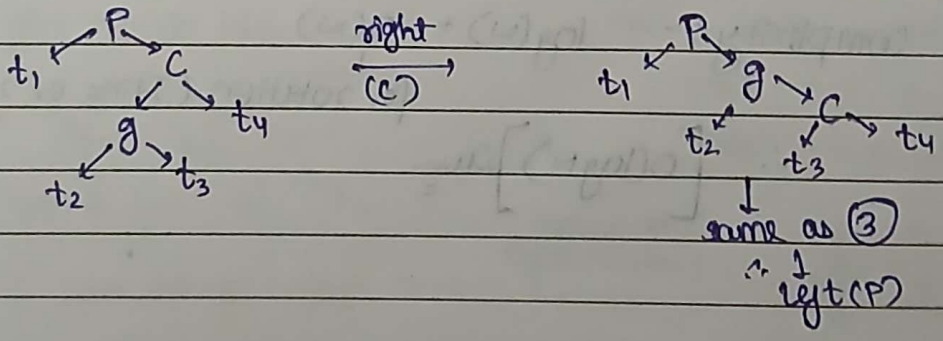


↓ right rotate (P) → same as (i) LL.

(iii) R-R

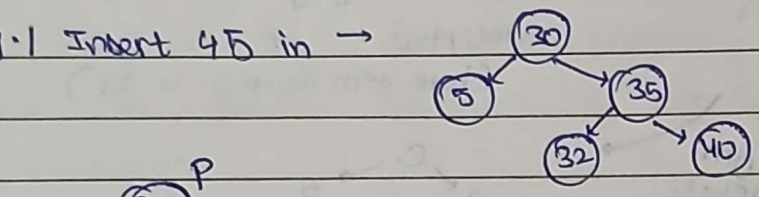


(iv) R-L

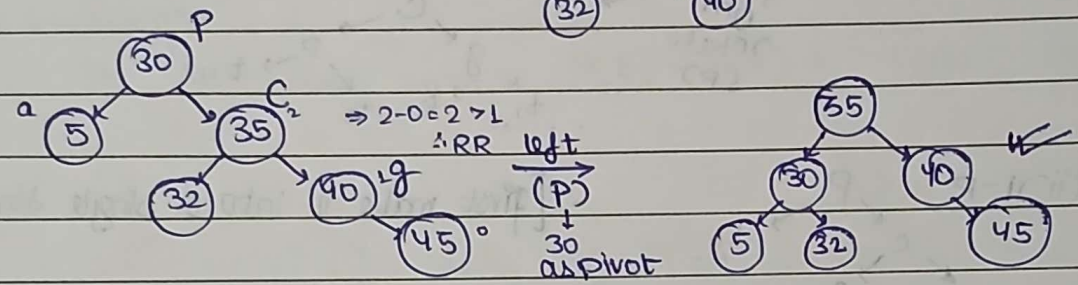


Note:- If a complete tree is getting unbalaced due to that subtree then correcting that subtree will not effect other tree & if that subtree is corrected all tree is corrected.

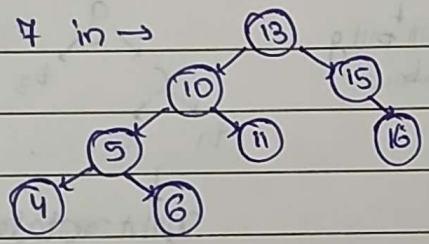
e.g.1 Insert 45 in →



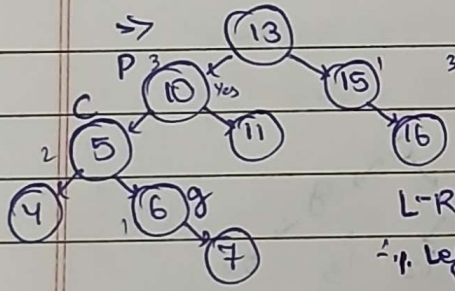
⇒



e.g.2 Insert 4 in →



⇒

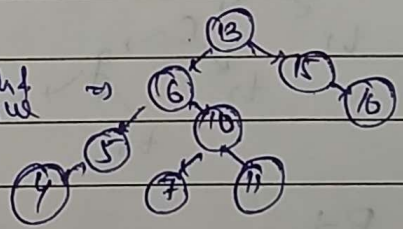


L-R case

∴ 1. Left rotate (C) } check if solved

2. Right rotate (P)

⇒



∴ Time complexity :- $\log(N) + O(N)$

↓ for rotation (1 time or 2 time)

∴ $[O(\log N)]$ Ans.

code copy paste of BST :- and before height fn. public int height(){ return height(root); }

change → at insert fn. after node.height

change return node; → return rotate(node);

*** (checking from bottom to up) →

create a rotate func. → private Node rotate(Node node){

[left heavy] ← if (height(node.left) - height(node.right) > 1){

no thinking in this code just (LL) ← if (height(node.left.left) - height(node.left.right) > 0){

visualisation from 4 cases return rightRotate(node); → (right rotate(P))

}

(LR) ← if (height(node.left.left) - height(node.left.right) < 0){

left rotate (C) ← node.left = leftRotate(node.left);

right " (P) ← return rightRotate(node);

}

}

Ulg. right heavy → < -1 {

(copy paste)

(RR) → node.right.left node.right.right < 0 {

return leftRotate(node);

}

(RL) → > 0 {

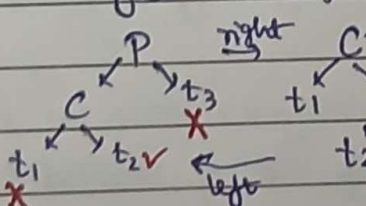
right rotate (C) → node.right = rightRotate(node.right);

left " (P) → return leftRotate(node);

}}

return node;

Right rotate func. → public Node rightRotate(Node P){



Node C = P.left;

t2 → Node t = C.right; // or P.left.right.

Now → (after rotation) C.right = P; P.left = t; } Simple

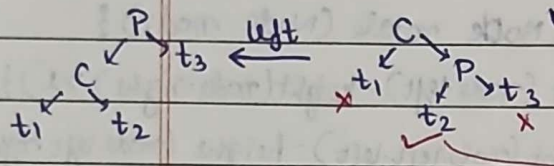
t1 & t3 not gonna change ∴ don't touch them.

update the heights now → P.height = Math.max(height(P.left), height(P.right)) + 1); (already know)

c.height = Math.max(height(c.left), height(c.right)) + 1;

now the new node is 'c' ∴ → return c;

}



Now rotate left

copy paste right → ——— leftRotate(Node c) {

Node p = c.right;

Node t = p.left;

Now rotating → (after rotation)

p.left = c;

c.right = t;

copy paste height for c & p

return p; → new node!

Main func →

inserting in ascending order (sorted)

if BST ∴ unbalanced but

it is AVL 😊

psvm {

AVL tree = new AVL();

for (int i=0; i<1000; i++) {

tree.insert(i);

}

sout(tree.height());

if BST height → O(N) ⇓ 1000

But AVL → $\log N$ ⇓ 3

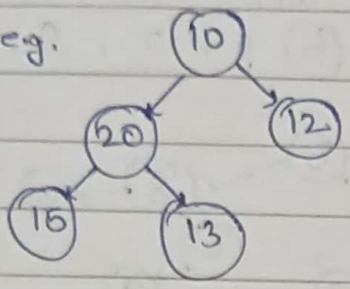
Date 02/07/24
Page

1. Traversal methods (algorithms in Trees) :-

(1) Pre-order

[Node → Left → Right]

e.g.



⇒ 10 → 20 → 15 → 13 → 12

⇒ (10, 20, 15, 13, 12) Ans.

Node Left right

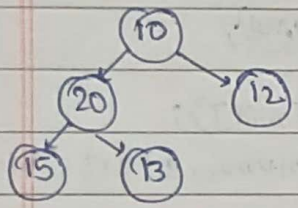
use case :-

⇒ used for evaluating math expression or making a copy.

⇒ Serialisation (converting String/Arrayes → to trees).

(2) In-order

[Left → Node → Right]

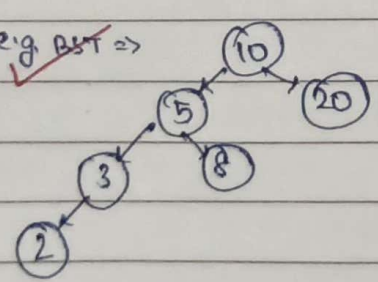


15 → 20 → 13 → 10 → 12

Left node right

Observation ⇒ In Binary Search Tree if you apply in-order traversal then you can visit the node of BST in sorted manner.

e.g. BST ⇒



⇒ 2, 3, 5, 8, 10, 20

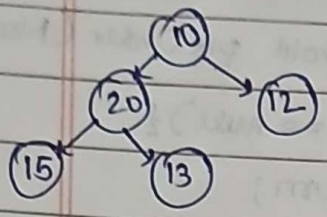
⇓ SORTED

use case :-

⇒ In BST if you want sorted order use in-order traversal.

(3) Post-order

[Left → Right → Node]



15, 13, 20, 12, 10

left right node

use case:-

→ When you need binary tree.

here, e.g. if you want to delete go order wise → you can't delete 20 first ∴ delete 15 & 13 then.

→ when you perform bottom of calculation

e.g. calculating height, diameter.

**** 99% ques. from Trees comes from here.

BFS and DFS or BFT and DFT

BFS → Breadth First Search

DFS → depth first search

BFT → breadth first Traversal

DFT → Depth First Traversal

Breadth →

Depth ↓

Depth ⇒ way of Traversing in a tree.

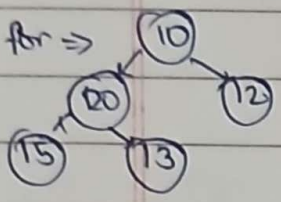
Next lecture video by Kunal Kushwaha.

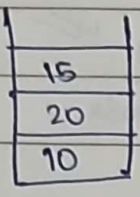
[N → L → R]

1. Implementing preorder in Binary Tree ⇒ assign root first → preOrder(root);

```
public void preOrder() {
}
private void preOrder(Node node){
if (node == null){
} return;
Sout(node.value + " ");
preOrder(node.left);
" (" right);
}
```

for ⇒





∴ 10 node != null

⇒ 10

⇒ lft → 20 != null

⇒ 20

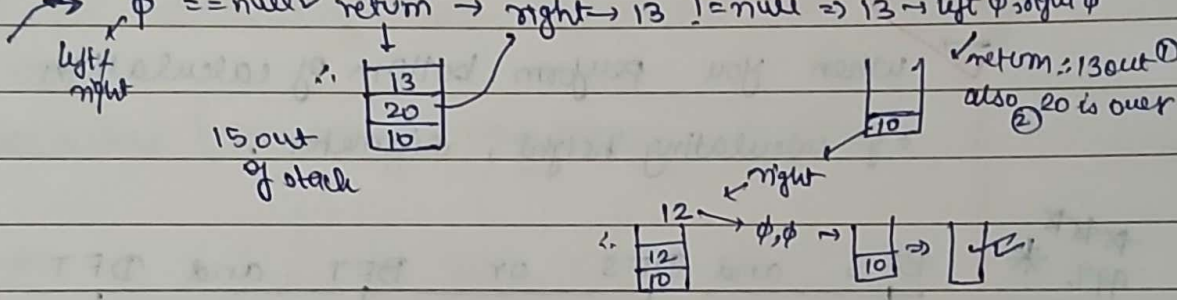
⇒ 15 → != null

⇒ 15

Node ← node

left ← .

Right ← .



2. Similarly in-order & post-order Traversal

in-order: L → N → R

① copy paste

② change name of method

③ inOrder(node.left);
Sout(node.value + " ");
inOrder(node.right);

post-order: L → R → N

① "

② "

③ "
postOrder(node.right);
Sout(node.value + " ");