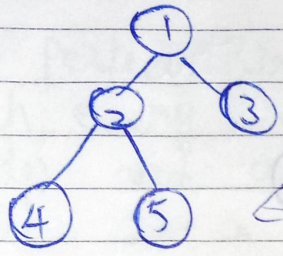
DATA STRUCTURES

Binary Tree

FRIDAY
JANUARY
22

'21

Binary Tree is a data structure in which each parent node can have atmost two children.



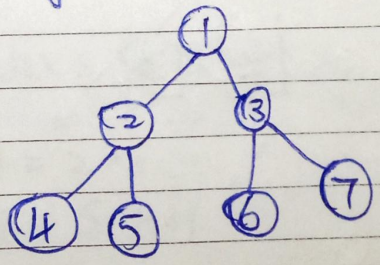
Types of Binary Tree:

(i) Full Binary Tree

In a full binary tree, every parent node or internal node has either two or no children.

(ii) Perfect Binary Tree

In a Perfect Binary Tree, every internal node has exactly two children.



and all the leaf nodes are at the same level

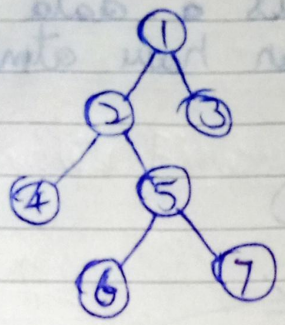
(iii) Complete Binary Tree

Just like a full Binary Tree, but

- Every level must be completely filled.
- All leaf elements must lean towards the left.
- The last leaf element might not have a right sibling.

23 SATURDAY
JANUARY
023-342 • WK 04

Full Binary Tree:



Either two or no children.

Theorems

i = no. of internal nodes
n = total number of nodes
l = no. of leaves
h = no of levels

then,

$l = i + 1$	$l = (n+1)/2$	$\max(\text{leaves}) = 2^{h-1}$
$n = 2i + 1$	$n = 2l - 1$	
$i = (n-1)/2$	$i = l - 1$	

Class Node:

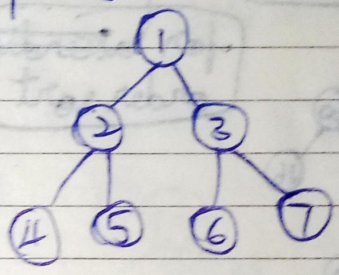
def __init

24 SUNDAY

2021

Perfect Binary Tree:

A perfect Binary tree is a type of a tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Theorems

no of nodes = $2^{h+1} - 1$

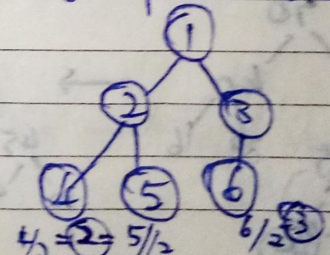
no of leaf nodes = 2^h

height = $\lfloor \log n \rfloor$

n = no. of nodes

Complete Binary Tree:

A complete Binary tree is a tree in which all the levels are completely filled except possibly the lowest one which is filled from the left.



parent index = $\lfloor \frac{i}{2} \rfloor$

left child = $2 * i$

right child = $(2 * i) + 1$

(REPUBLIC DAY) TUESDAY 26

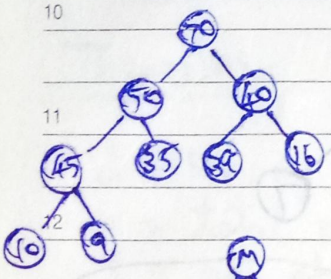
2021

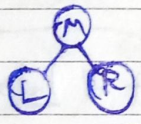
Heap

It is almost a complete Binary tree

Max-heap

descending order sort (or) for ~~ascending~~ order sort



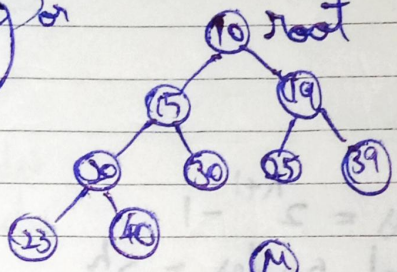


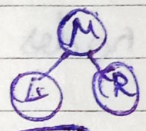
$M \geq$ (L or R)

mostly, (L > R)

Min-heap

ascending order sort (or) for ~~descending~~ order sort

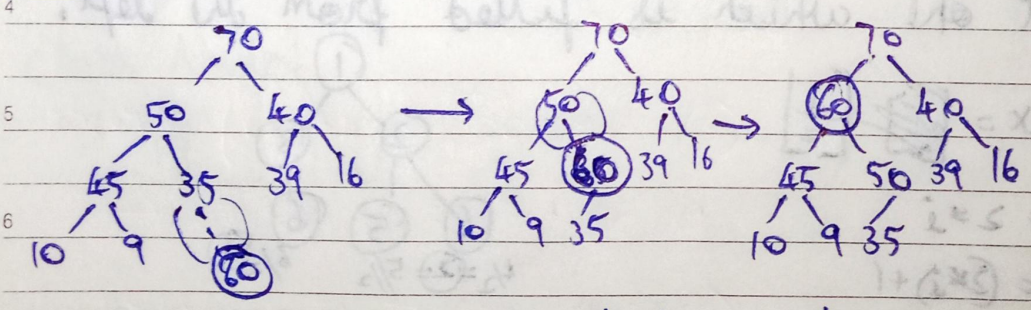




$M \leq$ (L or R)

mostly (L < R)

Insertion:



Always start from leftmost leaf node, and check if $M \geq$ (L or R) and if ~~so~~ true swap with M.

[70, 50, 40, 45, 35, 39, 16, 10, 9, 60] ⟶

[70, 50, 40, 45, 60, 39, 16, 10, 9, 35] ⟶

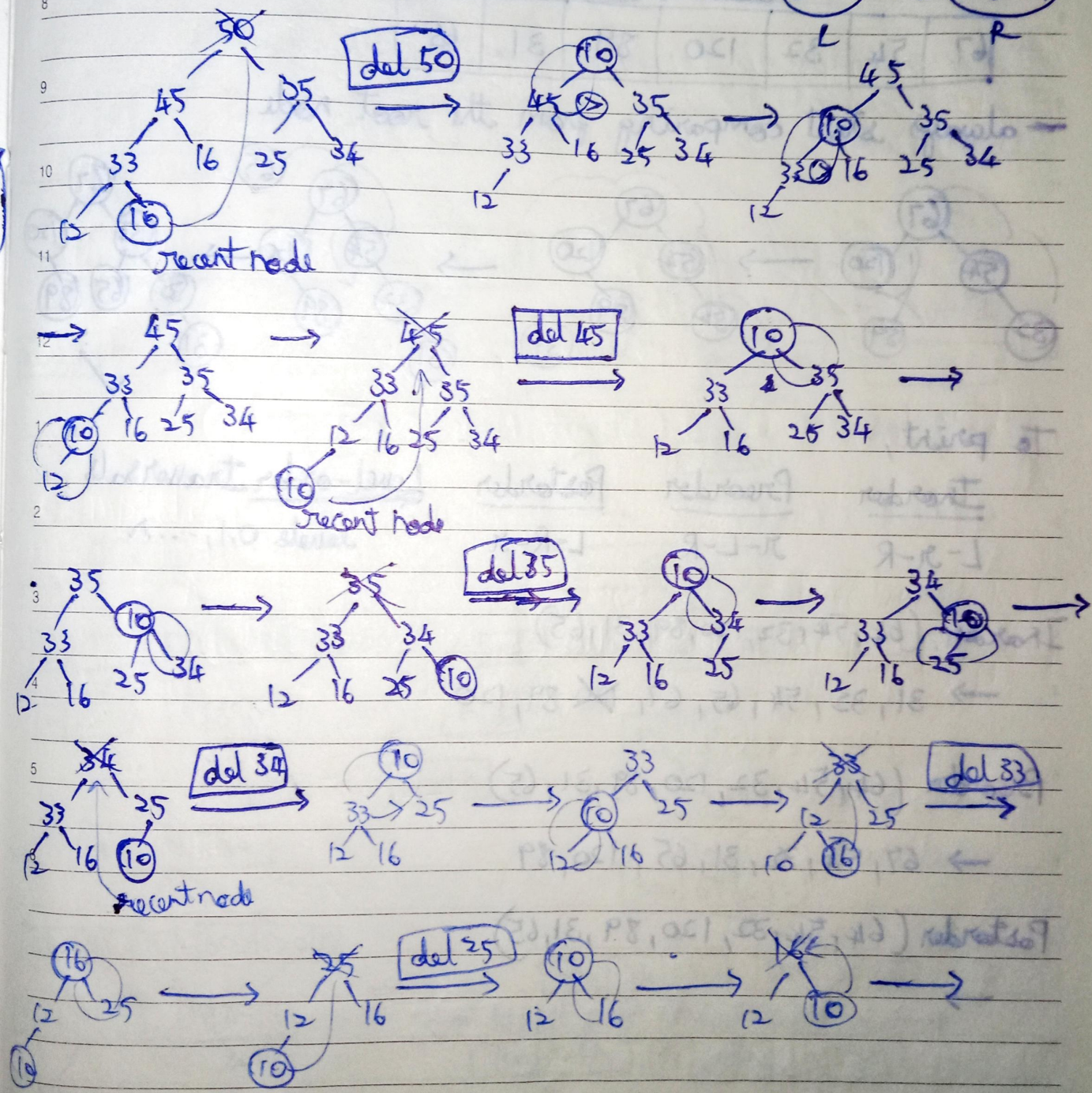
[70, 60, 40, 45, 50, 39, 16, 10, 9, 35]

2021

Deletion: (only delete root node)

$2*i$ and $2*i+1$

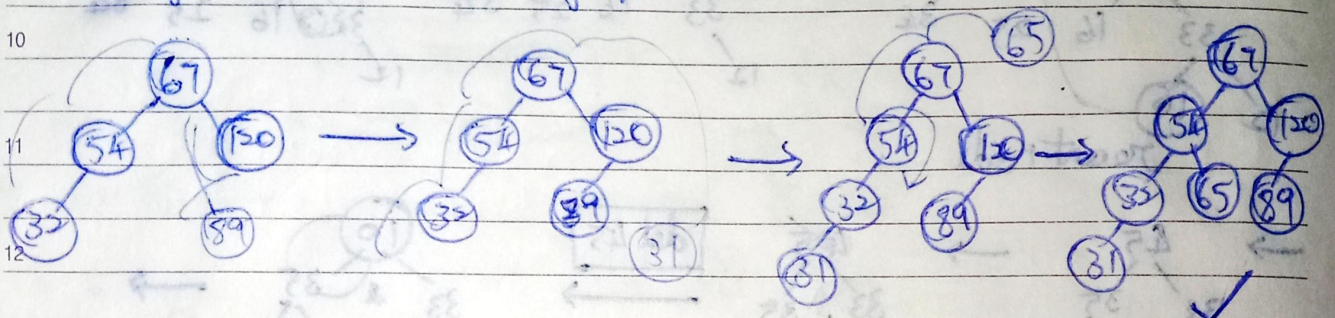
L R



Binary Search Tree

67	54	32	120	89	31	65

- always start comparing from the root node



To print,

Inorder	Preorder	Postorder	Level-order Traversals
L-R-R	R-L-R	L-R-R	levels 0, 1, ... n

Inorder (64, 54, 32, 120, 89, 31, 65)

→ 31, 32, 54, 65, 67, ~~89~~ 89, 120

Preorder (64, 54, 32, 120, 89, 31, 65)

→ 67, 54, 32, 31, 65, 120, 89

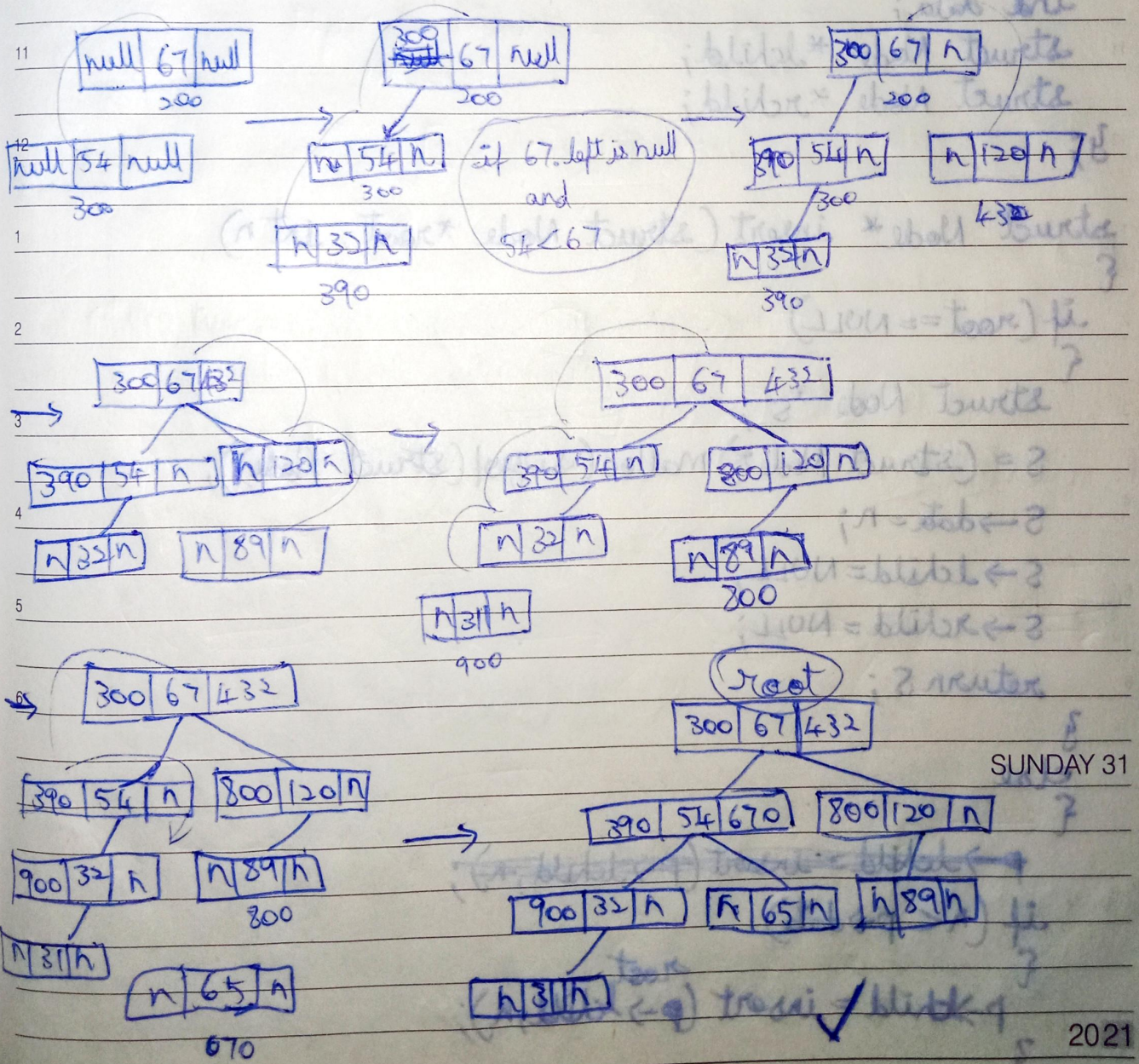
Postorder (64, 54, 32, 120, 89, 31, 65)

→

Node - | null | data | null |

left addr. | data | right addr.

| 67 | 54 | 32 | 120 | 89 | 31 | 65 |



SUNDAY 31

2021

AVL TREE

AVL Tree: root changes with balancing

AVL tree is a balanced Binary Search Tree.

Must check if every node is balanced or not.

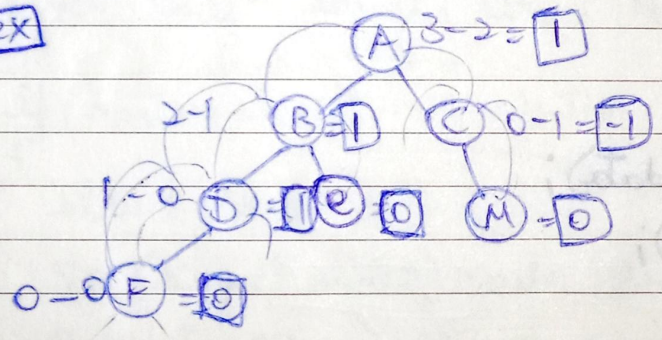
Find balanced factor

height of left subtree − height of right subtree

$$BF = hl - hr$$

where $BF = [-1, 0, 1]$

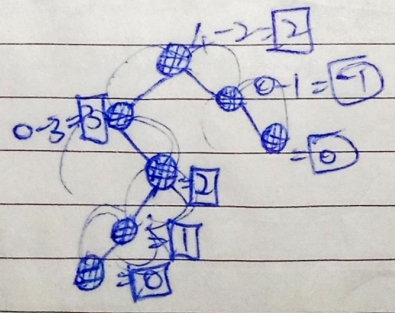
ex



✓ balanced

hence AVL

ex

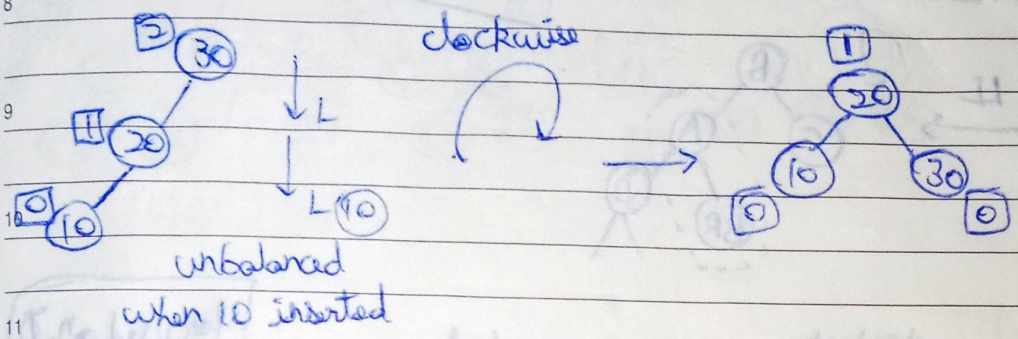


✗ not balanced

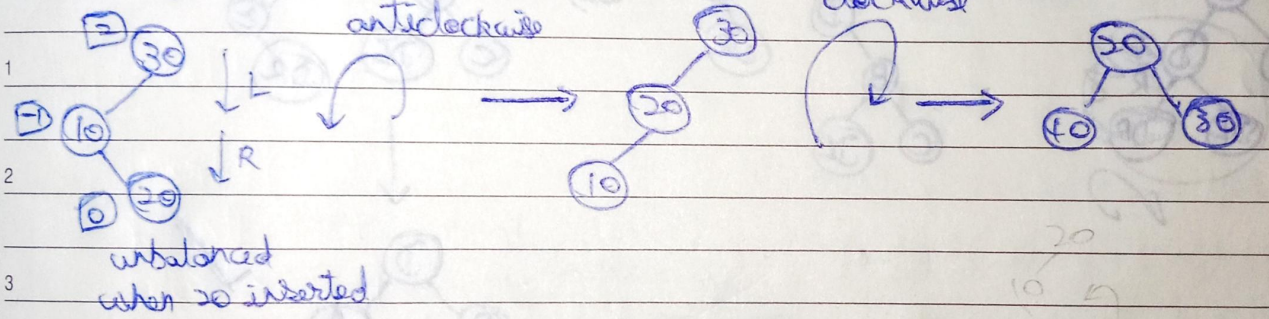
In order to balance,

LL LR RR RL rotations

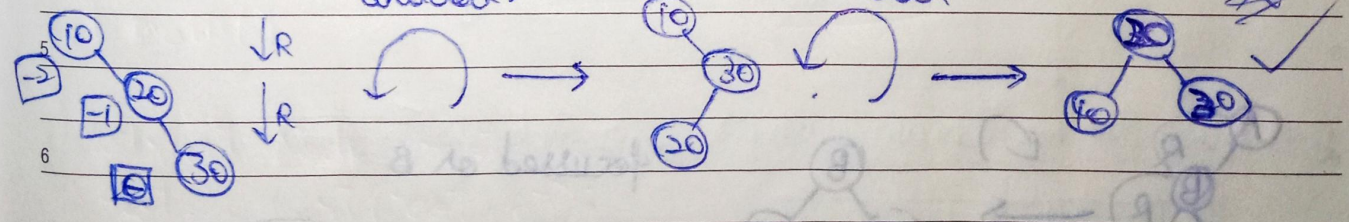
Left-left :



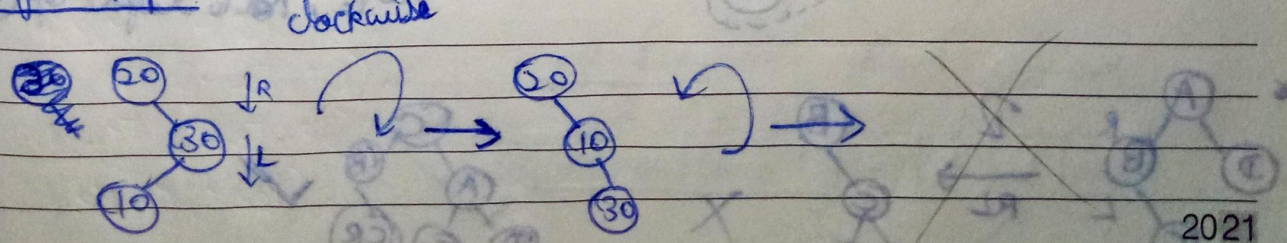
Left-Right :



Right-Right: (once right)

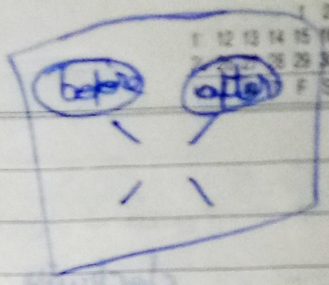


Right-Left:

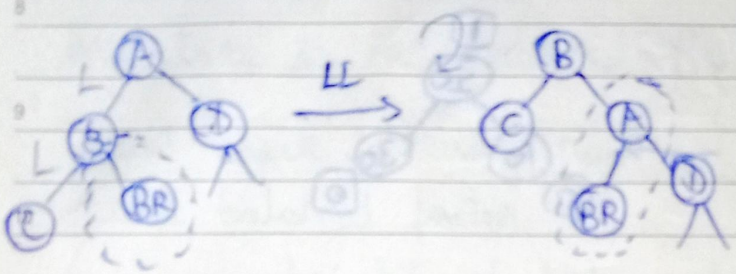


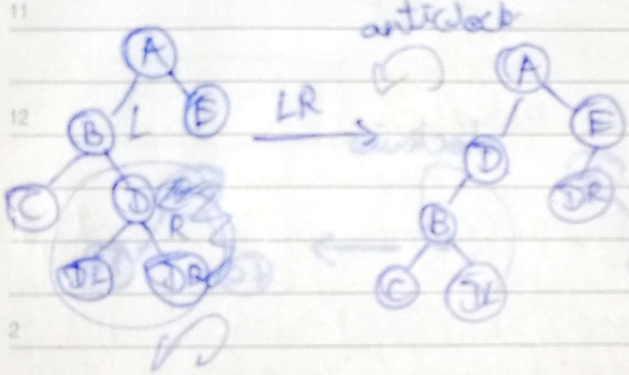
2021

05 FRIDAY
FEBRUARY
036-329 • WK 06

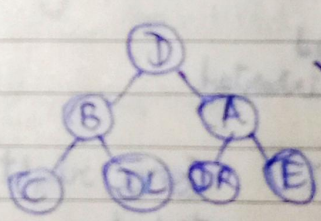


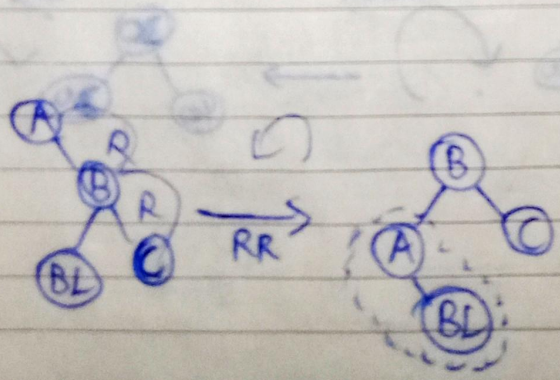
Formulate the rotation:





focussed on D

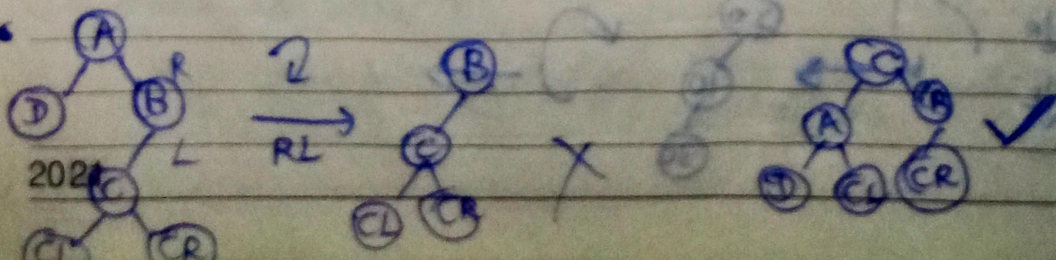




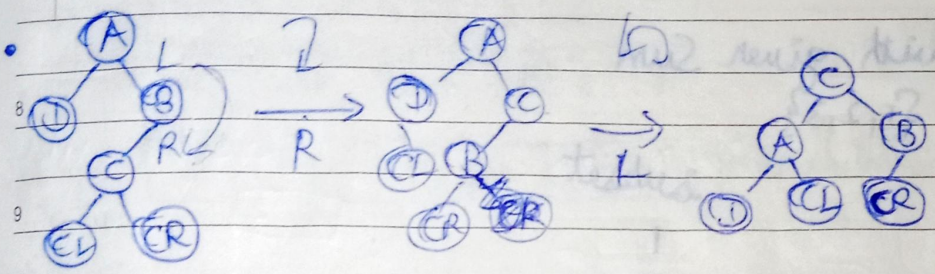
focussed on B

always BL < B

here, not to right of C



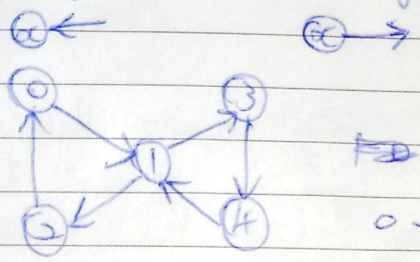
2021


A
B
C
D
CL
CR
L
R
RL
R
L

Heirholzer's Algorithm:

When a graph is traversed from a particular vertex and ends in the same vertex, its called

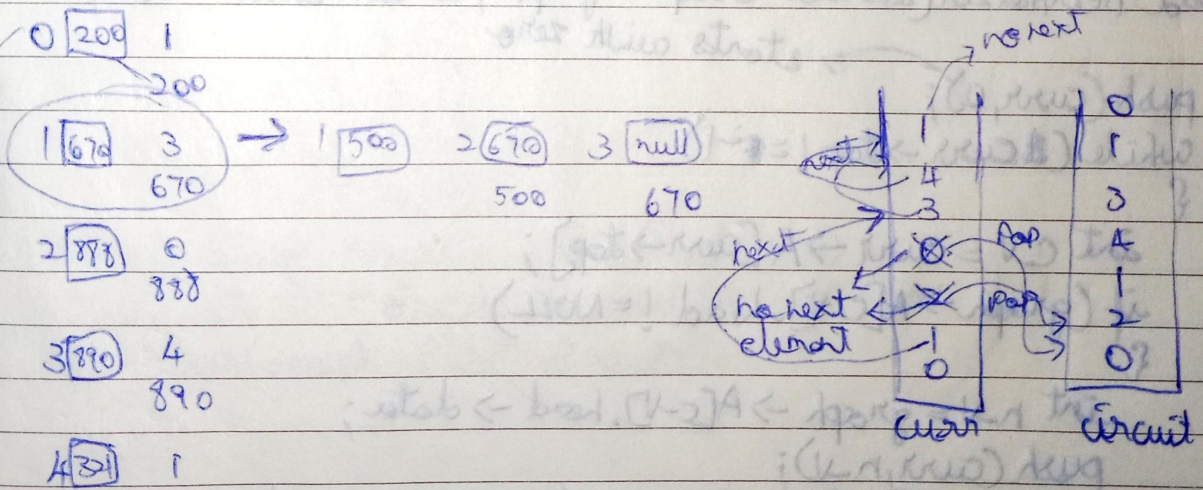
- Must be closed
- Indegree and Outdegree must be equal.



1-2-0-1-3-4-1

0-1-3-4-1-2-0

- create graph with size V → 2V for connection



- delete the top element ~~of~~ curr stack in the adjacency list.

circuit = [0, 1, 3, 4, 1, 2, 0]

when starting from zero

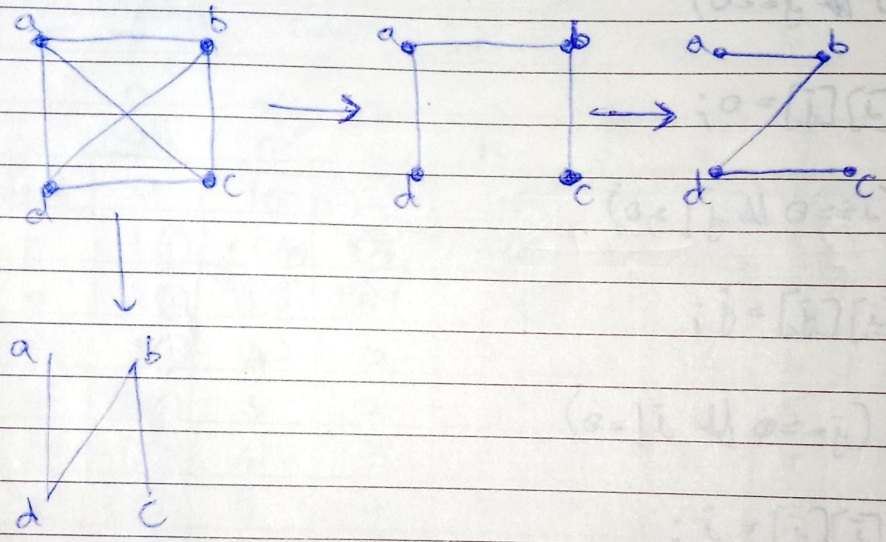
15 MONDAY
MARCH
074-291 • WK 12

SPANNING TREES

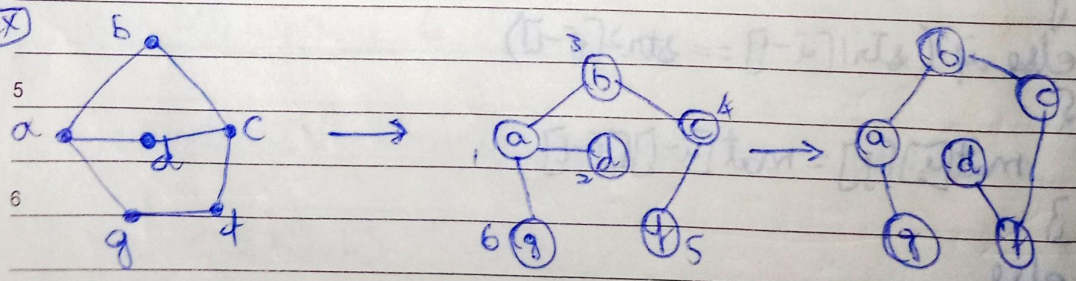
Spanning Trees:

Tree representation of a graph:

- All the vertices must be present.
- Minimum edges in the tree (vertices − 1)



ex



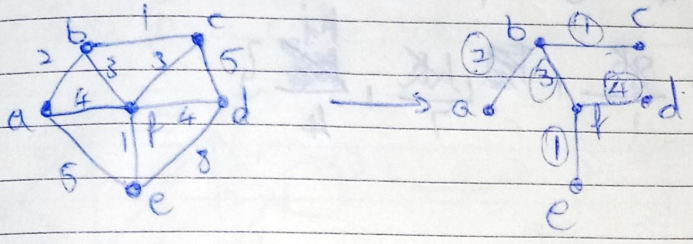
2021

PRIM'S

TUESDAY
MARCH
16
WK 12 • 075-290

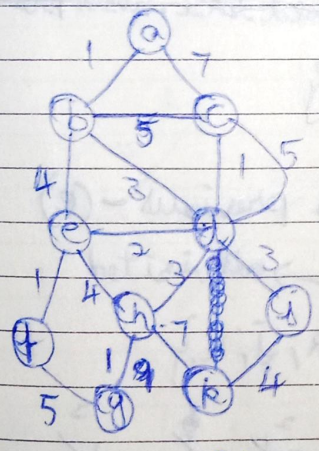
Minimum Spanning Tree:

Each edge has its own weight



start from minimum weight

But for efficient and bigger graphs, use algorithms.



Prim's Algorithm: (minimum spanning tree)

- Remove loops and parallel edges (keep minimum weight) (c-d) ✓ and ✗
- While adding new edge, select edge with minimum weight (no cycles)
- Stop at n-1 edge.

2021

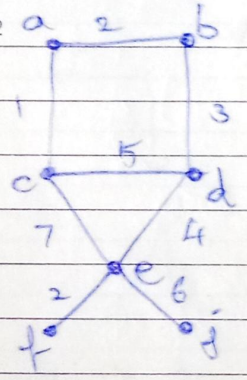
closed tree if used, so choose minimum in edges

visited = {a, b, d, c, e, f, g, h, k, j}

edges = {ab, ac, bc, be, bd, dk, de, dh, dj,
1 7 5 4 3 1 2 3 3
ek, eh, fg, gk, hk, kj}
1 4 5 1 7 4

weight = 1 + 3 + 1 + 1

ex

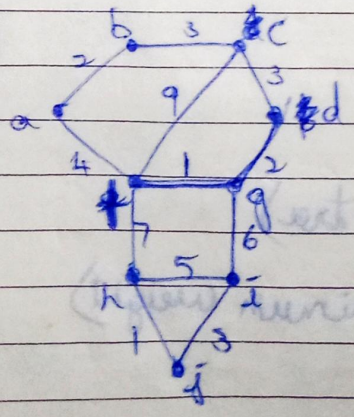


edges = {ab, ac, cd, ce, bd, de, ef, eg}
2 1 5 7 3 4 2 6

visited = {a, c, b, d, e, f, g}

all visited, go to previous - (e)

ex



visited = {f, g, d, c, b, a, j, h}

all visited

edges = {fg, fc, fa, fh, gd, gd, dc, cb, cf, bd, ih, ij, jk}
1 9 4 7 2 6 3 3 9 2 5 3 1

APR '21

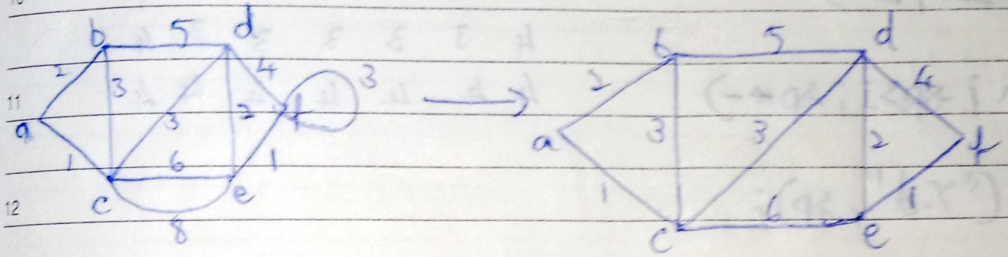
KRUSKAL'S

THURSDAY 18 MARCH

WK 12 • 077-288

Kruskal's Algorithm: (minimum spanning tree)

- Remove loops and parallel edges (keep min. weight)
- List all edges and sort according to weight.
- Take (n-1) edges from sorted list

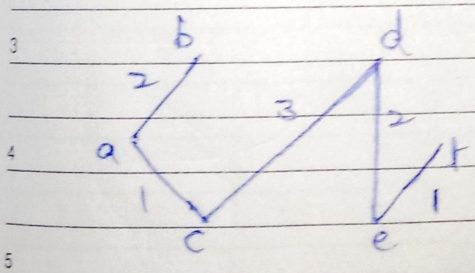


ab = 2 bc = 3 dc = 3 ef = 1 ec = 6

ac = 1 bd = 5 de = 2 df = 4

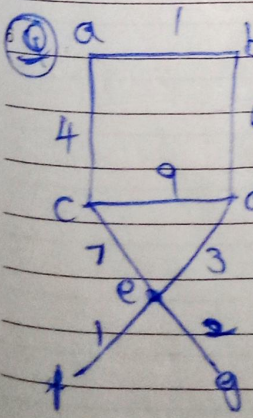
sort with weight

ac = 1 ✓
ef = 1 ✓
ab = 2 ✓
de = 2 ✓
bc = 3 — cycle
dc = 3 ✓
df = 4 — cycle
bd = 5 → 5 edges ✓
ec = 6



weight = 9

(2)

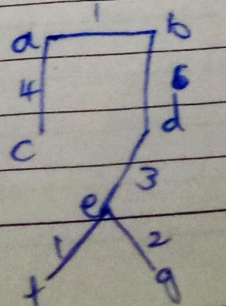


ab = 1 bd = 6 de = 3

ac = 4 cd = 9 ef = 1

ce = 7 eg = 2

ab = 1 ✓
ef = 1 ✓
eg = 2 ✓
de = 3 ✓
ac = 4 ✓
bd = 6 ✓
ce = 7
cd = 9



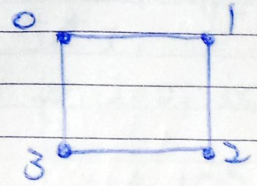
⇒ (17)

2021

M-COLOURING

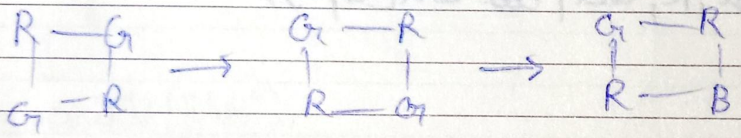
M-colouring problem:

In a given graph, color the vertex.



m=3, R, G, B

- no same colour for adjacent vertices
- all the colors may or may not be used.



- Represent as Adjacency Matrix

	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

- array = | -1 | -1 | -1 | -1 | if coloured or not
- take a vertex, check for adjacent in the matrix

0 → 1, 3

both | -1 | -1 |

use C_1 on 0 → | 1 | -1 | -1 | -1 |

ind

• 1 → 0, 2
C1

but C_1 already on 0, so use C_2 on 1 →

ind				
	1	2	-1	-1

• 2 → 1, 3
C2

but C_2 already on 1, so use C_3 on 2 →

ind				
	1	2	1	-1

or
C_1

• 3 → 0, 2
C1 C1

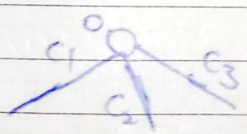
but C_1 already on 0, 2, so use C_2 on 3 →

ind				
	1	2	1	2

(V)

or
C_3

this is one solution.



Binary Search Trees

	Array (unsorted)	Linked List	Array (sorted)	BST	Binary Tree
Searching	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Insertion	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$	$O(n)$
Deletion	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$

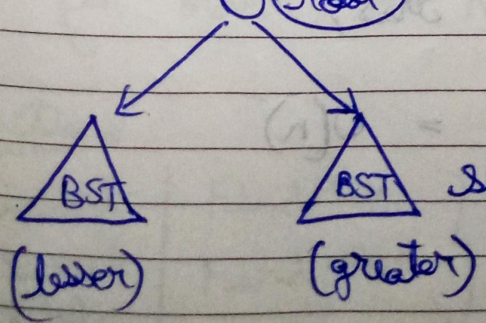
maybe..?

Thats why BST s are implemented — for lower time complexity of operations

since its sorted, binary search is used.

BST conditions

~~L < R~~ $L < r < R$



subtree

all elements of left subtree is lesser than all elements in right subtree.

Search, Insert, Delete

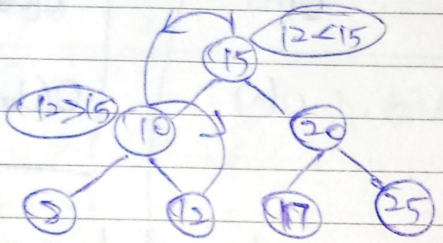
8, 10, 12, ~~15, 17, 20, 25~~ [Search for 10]

⇒ ~~8~~, 10, 12

⇒ 10, ~~12~~

⇒ [10] ✓

$\log_2(7) = 3$ steps

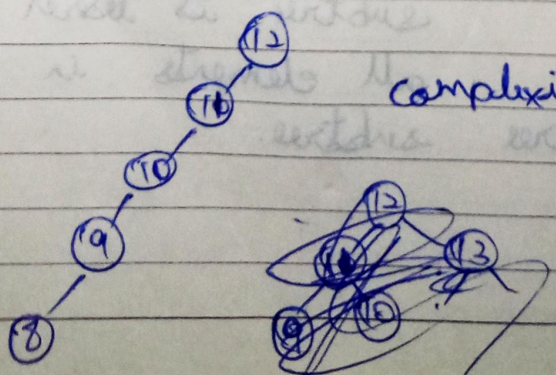


search (12)

It is a traversal where in each step we go towards left or right, in other words in each step a subtree is discarded.

- The BST is ~~not~~ always balanced in order to keep the complexity as O(logn)

incase of a tree like this,



complexity = O(n)

if balanced, the complexity = $O(\log n)$

Always start comparing from the root node.

$L \leq R < R$

Implementation

- Dynamic Nodes ✓
- Array

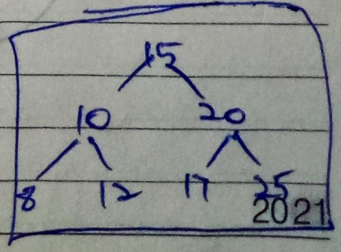
The main problem now is after insertion or deletion the tree may become unbalanced and we should balance the tree again

(in order to maintain $O(\log n)$)

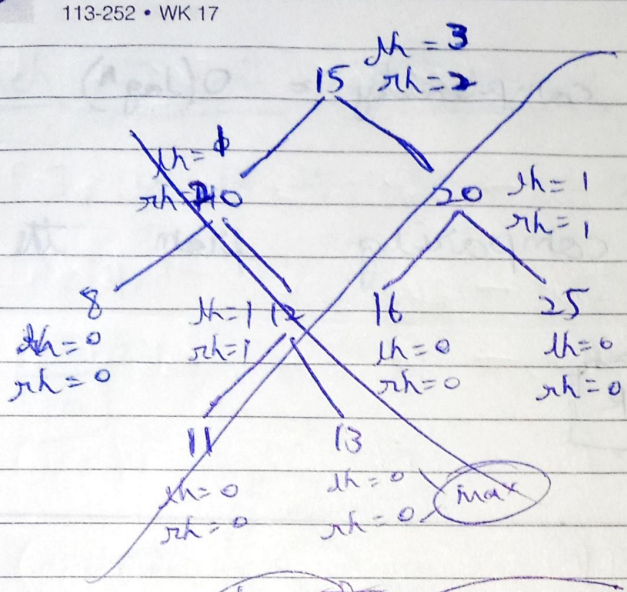
functions

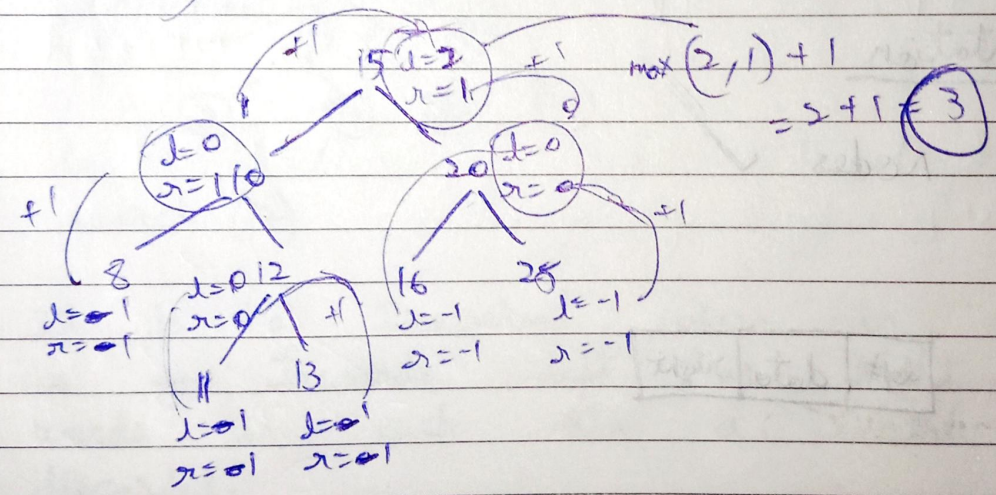
insert
search
delete
print

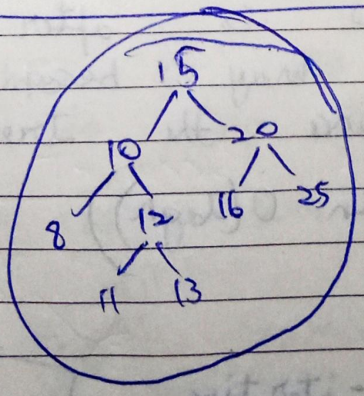
FindMax } — iterative
FindMin } — recursion



2021


15
lh=2
rh=2
10
lh=1
rh=1
20
lh=1
rh=1
8
lh=0
rh=0
12
lh=1
rh=1
16
lh=0
rh=0
25
lh=0
rh=0
11
lh=0
rh=0
13
lh=0
rh=0
max


15
l=2
r=1
+1
+1
max(2,1)+1
=2+1=3
10
l=0
r=1
20
l=0
r=0
+1
+1
8
l=0
r=0
12
l=0
r=0
+1
16
l=-1
r=-1
25
l=-1
r=-1
11
l=0
r=0
13
l=0
r=0


15
10
20
8
12
16
25
11
13

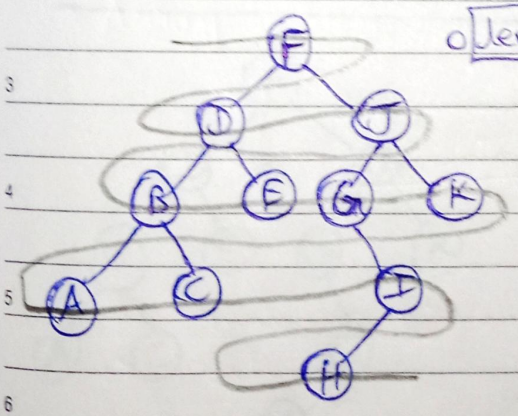
Tree Traversal

Breadth-first traversal (level order)

Depth-first traversal

Breadth-First Traversal

level order = F, D, J, B, E, G, K, A, C, I, H



starts from level 0, from left to right.

level-by-level

Depth First Traversal

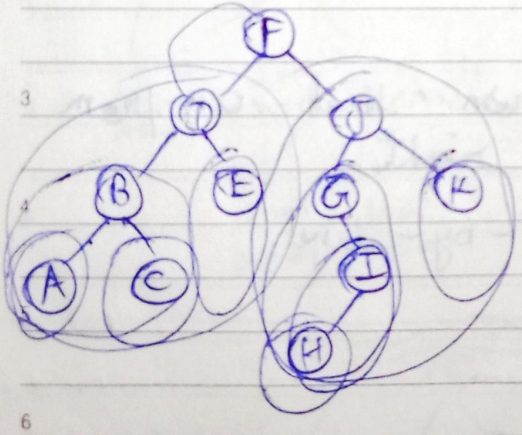
Preorder = root – left – right

Inorder = left – root – right

Postorder = left – right – root

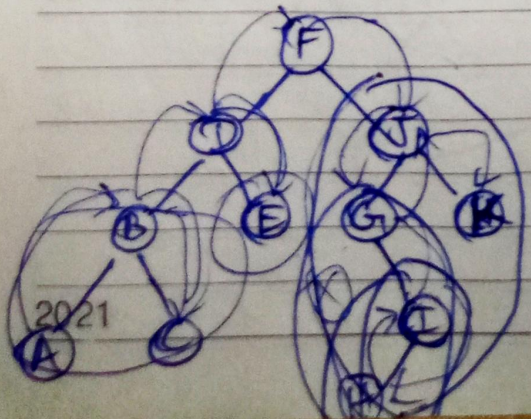
Preorder Traversal (r-L-R)

F D B A C E J G I H K

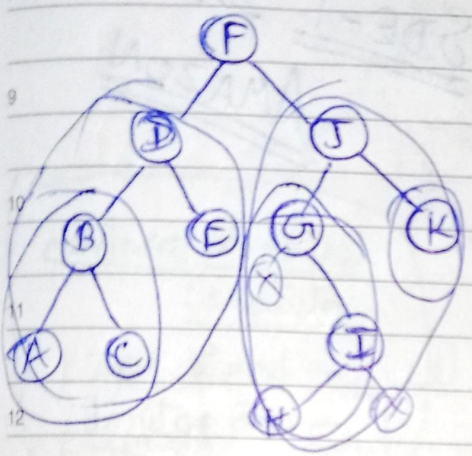


Inorder Traversal (L-r-R)

A B C D E F G H I J

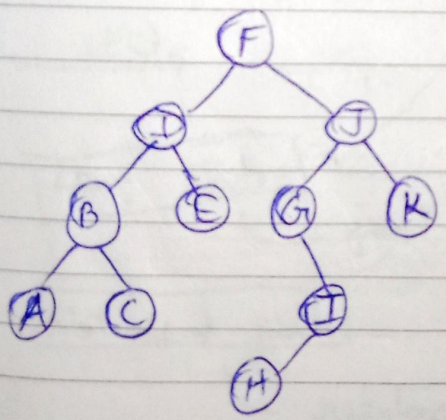


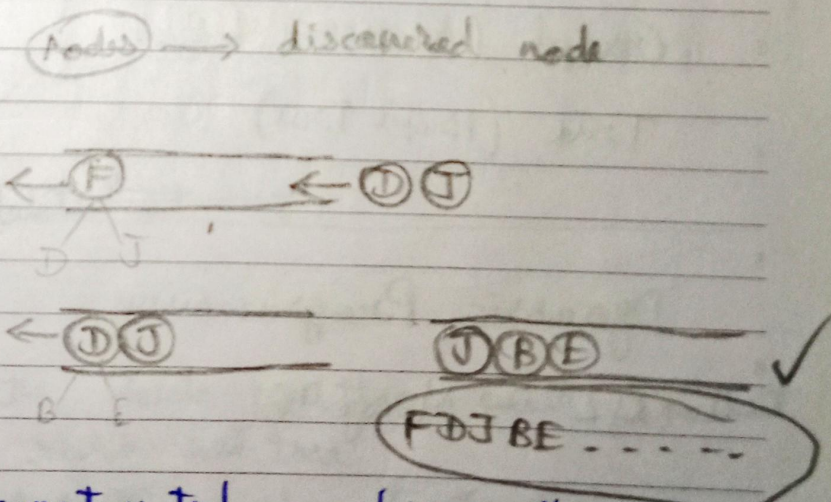
Postorder Traversal (L-R-R)



A C B E D H I G K J F

Level Order Traversal (implementation)





A Queue is constructed where the discovered node's children are inserted. Once an element is popped out it becomes a discovered node and its children are inserted.

28 WEDNESDAY
APRIL
118-247 • WK 18

MAR '21

Array ✓

Linked List ✓

Stack ✓

Queue ✓

Binary Tree ✓

BST ✓

Heap — MinHeap, MaxHeap

HashTable (vs HashMap) — (dictionary in Python)

Graph (Adjacency List)

Trie (Prefix Tree)

Dynamic Programming

SQL — MongoDB

System Design — Design a Parking lot
Design an Online Book Store

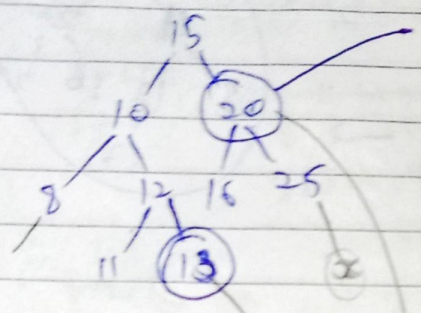
Open Source Projects

FOR SDE-1 IN AMAZON

leetcode problem solutions:
- brute force
- sub optimal
- super optimal

2021

Delete :



root = 20
temp = minVal(root.right)
= 25
root = 25
root.right = deleteNode(root, 25)

Deletion will pick and remove only the element and the rest will be printed in the same order.

20 30 40 50 60 70 80

del(20) → 30 40 50 60 70 80

del(50) → 30 40 60 70 80

Leaf Node — just delete it

Not Leaf Node

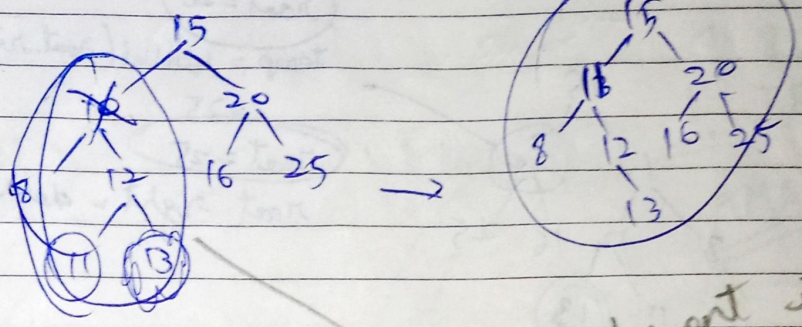
↳ replace the delete position with smallest element in the right subtree.
And delete that element (same way)

[Inorder Successor]

if the smallest element is it self or none

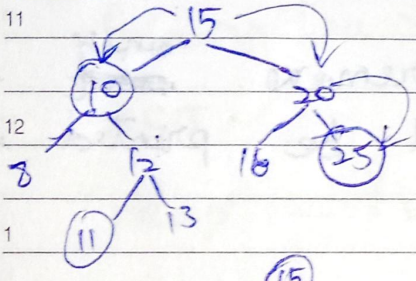
return

30 FRIDAY
APRIL
120-245 • WK 18



Smallest element in Right subtree

delete 10



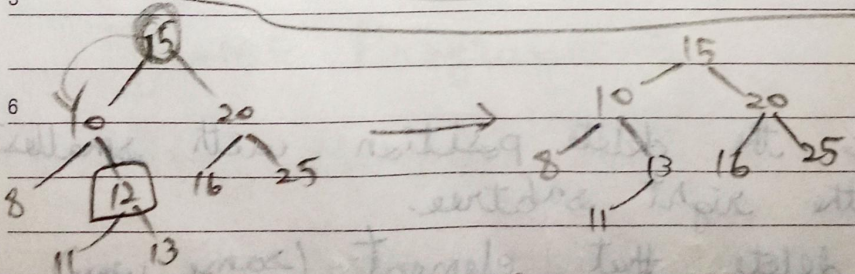
root = delete(15, 25)

rootNode.right = self.delete(rootNode.right, 25)

rootNode.right = self.delete(25, 25)

return None

None



(12) key < root.val (15) ⇒ delete(10, 12)

(13) key > root.val (10) → delete(12, 12)

(13) key = root.val (12) → succPar = 12

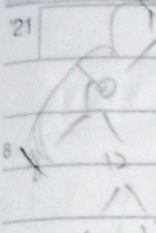
succ = 13

succPar = rootNode

succPar.right = succ.right

2021

SATURDAY
MAY
01
WK 18 • 121-244



Inorder (L→R) = 8 10 11 12 13 15 16 20 25

Preorder (R L R) = 15 10 8 12 11 13 20 16 25

Postorder (L R →) = ~~8 11 13 12 10 15 16 25 20 15~~ 8 11 13 12 10 16 25 20 15 ✓

Inorder(15)
Inorder(15, left)
print(15)
Inorder(15, right)

Inorder(10)
Inorder(10, left)
print(10)
Inorder(10, right)

Inorder(
Base Case
if(root is Null:
return

A
B C
D E

~~delete_recursive(4)~~

~~def remove(self, root, key)~~
~~root = self.root~~

Binary Tree

deletion

SUNDAY 02