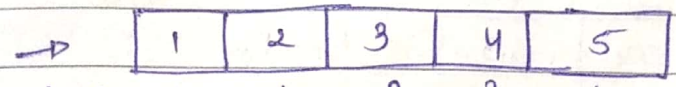
"Swami Shreeji ||"

Data structure & Algorithm

JANUARY 2020 FRIDAY 03

* One dimensional Arrays :-



A[i] = Base address + i

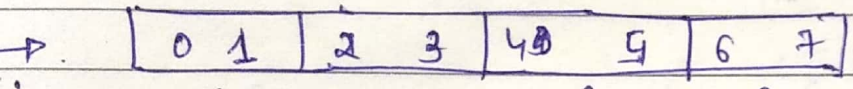
i = 0, 1, 2

A[2] = 100 + 2 ; if B.a = 100.

= 102.

A[i] = (Base address + (i-1))

i = 1, 2, 3, ...



A[i] = (Base address + (2 * i))

↑ size of each element

i = 0, 1, 2, ...

A[i] = (Base address + (2 * [i-1]))

↑ size of each element

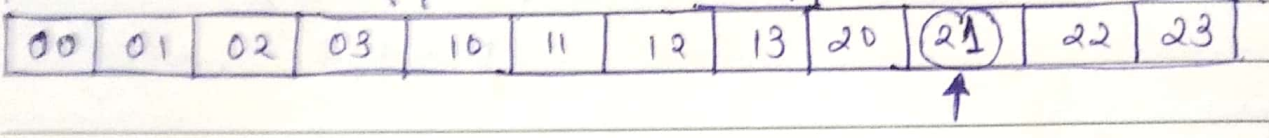
i = 1, 2, 3, ...

* Row major order :

A: 3×4

	0	1	2	3
0	00	01	02	03
1	10	11	12	13
2	20	(21)	22	23

m×n



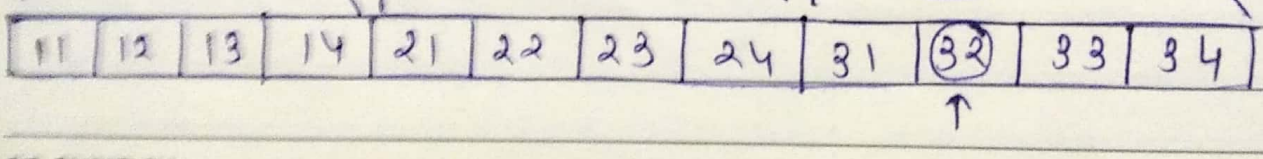
$$A[i][j] = ((i * n) + j) * size + Base$$

while index starts with '0'.

~~A[i][j]~~ = A: 3×4

	1	2	3	4
1	11	12	13	14
2	21	22	23	24
3	31	(32)	33	34

m×n



05 SUNDAY

$$A[i][j] = [((i-1) * n) + (j-1)] * size + Base$$

while index starts with '1'.

* Column major order :

A: 3×4

i ↓ \ j →	0	1	2	3
0	00	01	02	03
1	10	11	12	13
2	20	(21)	22	23

m × n

$$A[i][j] = [(j * m) + i] * size + Base$$

while i, j starts with 0.

$$A[i][j] = [((j-1) * m) + (i-1)] * size + Base$$

while i, j starts with '1'.

---

if number of rows and columns of a matrix is given like

$$A[L_r \ldots U_r][L_c \ldots U_c]$$

$\therefore$ No. of rows $= (U_r - L_r) + 1$

No. of columns $= (U_c - L_c) + 1$

* Address of 3 dimensional array :-

→ Row major order :-

$$A[i, j, k] = Base + size[(D - D_0)RC + (i - R_0)C + (j - C_0)]$$

D = Total no of cells depth wise
D_0 = Lower bound of z.

R = Total no of rows
R_0 = Lower bound of x.

C = Total no of columns
C_0 = Lower bound of y.

→ Column major order :-

$$A[i, j, k] = Base + size[(D - D_0)RC + (i - R_0) + (j - C_0) \cdot R]$$

Ex:- A[-2:0, 1:4, 6:9]

find A[-1, 3, 8], base = 1000

→ Row major order,

$$A[i,j,k] = Base + size[(D-D_0)RC + (i-R_0)C + (j-C_0)]$$

∴ A[-1, 3, 8],

Assume size = 4 for float

$D_x = 9 - 6 + 1 = 4$

$D_0 = 6$

$R_x = 0 - (-2) + 1 = 3$

$R_0 = -2$

$C = 4 - 1 + 1 = 4$

$C_0 = 1$

$$\therefore A[-1,3,8] = 1000 + 4[(4-6)(3)(4) + (-1-(-2))4 + (3-1)]$$

$$= 1000 + 4[-24 + 4 + 2]$$

$$= 928$$

Sorting Algorithms

14 TUESDAY JANUARY 2020

* Criteria for analysis :

1) Number of comparison (by which the time complexity is given)
2) Number of swaps
3) Adaptive
4) Extra memory

* Stable :

→ if a sorting algorithm is preserving the order of a duplicate element in a sorted list then it is called a stable algorithm.

Before :

Name :	A	B	C	D	E	F	G
Marks :	5	8	6	4	6	7	10

After :

Name :	D	A	C	E	F	B	G
Marks :	4	5	6	6	7	8	10

* Sorting techniques :

- Comparison based sorts
  - $O(n^2)$
    1. Bubble sort
    2. Insertion sort
    3. Selection sort
  - $O(n \log n)$
    4. Heap sort
    5. Merge sort
    6. Quick sort
    7. Tree sort –
  - $O(n^{3/2})$ → 8. Shell sort –
- Index based sorts
  - $O(n)$
    9. Count sort –
    10. Bucket / Bin sort –
    11. Radix sort

* Comparison b/w bubble sort and insertion sort :

	Bubble sort	Insertion sort	
min comp.	$O(n)$	$O(n)$	ASC
max comp	$O(n^2)$	$O(n^2)$	DSC
min swaps	$O(1)$	$O(n)$	ASC
max swaps	$O(n^2)$	$O(n^2)$	DSC
Adaptive	✓	✓	
stable	✓	✓	
Linked list	No	Yes	
K passes	Yes	No	

16 THURSDAY JANUARY 2020

JAN 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
M T W T F S S M T W T F S S

1. Bubble sort

→ Algo: n = length [Arr]

```
for i = 0 to length [Arr]
    for j = 0 up to n - 1 - i
        if A[j] > A[j + 1]
            swap ( A[j], A[j+1] )
```

→ No. of passes : n-1
→ No. of comparison : $O(n^2)$
→ No. of swaps : $O(n^2)$
→ Adaptive, Stable
(Not by nature)
→ Time complexity :

min → $O(n)$ (when sorted already)
max → $O(n^2)$

→ If the list is already sorted then it is taking minimum time so bubble sort is adaptive.

```
void bubbleSort(int arr[], int n)
{
    for(int i=0; i<n; i++)
    {
        int flag;
        for(int j=0; j<n-1-i; j++)
        {
            flag = 0;

            if( arr[j] > arr[j+1] )
            {
                swap( arr[j], arr[j+1] );
                flag = 1;
            }
        }
        if( flag == 0 ) // this will execute
            break;      // when list is already
                        // sorted.
    }
}
```

→ it is not adaptive in nature, we make it by adding if(flag==0) condition.

2. Insertion sort

→ Algo :

```
for i = 1 to length[arr]
    key ← arr[i]
    p = i - 1 ;

    while p >= 0 and arr[p] > key
        arr[p+1] ← arr[p] ;
        p ← p - 1
    arr[p+1] ← key
```

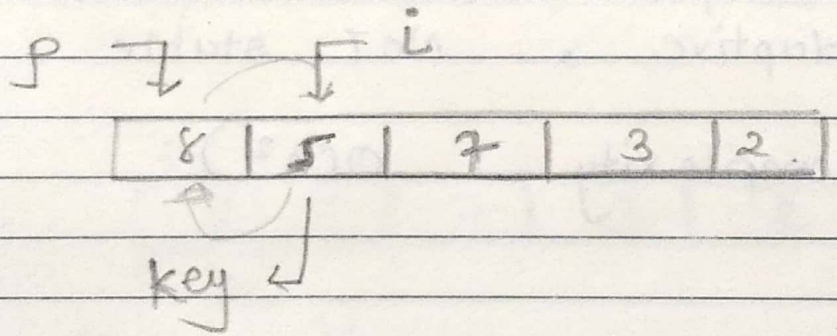
→ No. of passes : n-1
→ No. of swaps : $O(1)$ min
$O(n^2)$ max
→ No. of comparison : $O(n^2)$
→ Adaptive by nature, stable

→ Time Complexity :

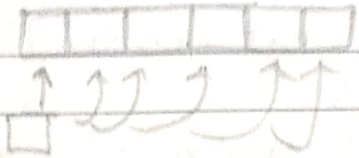
19 SUNDAY min → $O(n)$ (when already sorted)
max → $O(n^2)$

```
void insertionSort ( int arr[] , int n )
{   int p , key ;
    for ( int i = 1 ; i < n ; i++ )
    {
        key = arr[i] ;
        p = i - 1 ;

        while ( p >= 0 && arr[p] > key] )
        {
            arr[p+1] = arr[p] ;
            p-- ;
        }
        arr[p+1] = key ;
    }
}
```



3. Selection Sort



→ Algo :

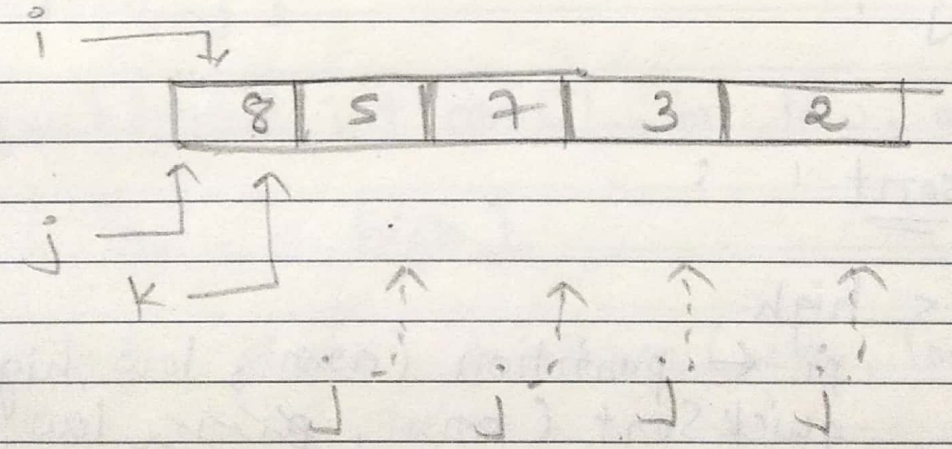
```
for i ← 0   up to   n-1
    for j = k ← i   up to   n
        if  arr[j] < arr[k]
            then   k = j
    exchange ( arr[i] , arr[k] )
```

→ No. of comp. : $O(n^2)$
→ No. of swaps : $O(n)$ (minimum of all sorting tech.)
→ NOT adaptive , NOT stable

→ Time complexity : $O(n^2)$

```
void selectionSort ( int arr[] , int n )
{
    int i , j , k ;

    for ( i = 0 ; i < n-1 ; i++ )
    {
        for ( j = k = i ; j < n ; j++ )
        {
            if ( arr[j] < arr[k] )
            {
                k = j ;
            }
        }
        swap ( arr[i] , arr[k] ) ;
    }
}
```



4. Quick sort

→ Algo :

partitioning array :

```
pivot ← arr[low]
  i ← low , j ← high
while (i<j){while (arr[i] ≤ pivot)
        i++
  while (arr[j] > pivot)
        j--
  if (i<j)
      swap (arr[i], arr[j]) }
swap(arr[low], arr[j]);
return j ;
```

quick sort :

```
if low < high
  then    pi ← partition (arr, low, high)
          quickSort (arr, low, pi-1)
          quickSort (arr, pi+1, high)
```

→ Time complexity : Best case : O(nlogn)
Average case : O(nlogn)
Worst case : O(n²)

```
int partition ( int arr[], int low, int high)
{
    int pivot = arr[low];
    int i = low, j = high;

    while (i < j)
    {
        while ( arr[i] <= pivot)
            i++;
        while ( arr[j] > pivot )
            j--;
        if (i < j)
            swap( arr[i], arr[j] );
    }
    swap( arr[low], arr[j] );
    return j;
}
void quickSort ( int arr[], int low, int high)
{
    if ( low < high)
    {
        int pi = partition ( arr, low, high);

        quickSort ( arr, low, pi-1);
        quickSort ( arr, pi+1, high);
    }
}
```

5. Merge sort

→ Program :-

```
#include <iostream>
using namespace std;

void display ( int a[] , int size )
{
    for (int i=0 ; i < size ; i++)
    {
        cout << a[i] << " ";
    }
}

    void merge (int arr[], int low, int mid,
                                    int high)
    {
        int n1 = mid - low + 1 ;
        int n2 = high - mid ;

        int LeftArray [n1] ;
        int RightArray [n2] ;

        for( int i=0 ; i < n1 ; i++)
            LeftArray [i] = arr [low + i] ;
        for ( int j=0 ; j < n2 ; j++)
            RightArray [j] = arr [ mid +1 + j] ;
```

```
int i=0 , j =0 , k = low ;

while ( i < n1 && j < n2 )
{
    if ( LeftArray [i] <= RightArray [j] )
    {
        arr [k] = LeftArray [i] ;
        i++ ;
    }
    else
    {
        arr [k] = RightArray[j] ;
        j++ ;
    }
    k++;
}

while ( i < n1 )
{
    arr [k] = LeftArray [i] ;
    i++;
    k++;
}
while ( j < n2 )
{
    arr [k] = RightArray [j] ;
    j++;
    k++;
}
}
```

```
void mergeSort(int a[], int low, int high)
{
    if (low < high)
    {
        int mid = low + (high-low)/2;
        mergeSort(a, low, mid);
        mergeSort(a, mid+1, high);
        merge(a, low, mid, high);
    }
}

int main()
{
    int a[] = {6, 5, 12, 10, 9, 1};
    int size = sizeof(a) / sizeof(a[0]);
    mergeSort(a, 0, size-1);
    cout << "Sorted Array : " << endl;
    display(a, size);
    return 0;
}
```

→ Time Complexity :

O(n.logn) in all the cases.

→ NOT inplace .
→ stable
→ NOT adaptive

6. Counting Sort :

```
#include <iostream>
using namespace std;

void countSort ( int a[], int size )
{
    int output [10];
    int count [10];
    int max = a[0];

    // finding the longest element of the array
    for( int i = 1 ; i< size ; i++ )
    {
        if ( a[i] > max)
            max = a[i];
    }

    // initializing count array with all zeros
    for (int i =0 ; i<=max ; ++i )
    {
        count [i] =0 ;
    }
```

```
// storing count of each element.
for( int i=0; i<size; i++)
{
    count[a[i]]++;
}

//Storing cumulative count of each array.
for (int i = 1; i<=max; i++)
{
    count[i] = count[i] + count[i-1];
}

// finding the index of each element of the
original array in count array and
placing the elements in output array.
for (int i=size-1; i>=0; i--)
{
    output[ count[a[i]] - 1 ] = a[i];
    count[a[i]]--;
}

// Copying the sorted elements into original
array.
for (int i=0; i<size; i++)
  {
      a[i] = output[i];
  }
}
```

```
void display ( int a[] , int size )
{
    for ( int i=0 ; i < size ; i++ )
    {
        cout << a[i] << " " ;
    }.
    cout endl ;
}.

int main( )
{
    int array[] = { 4, 2, 2, 8, 3, 3, 1 } ;
    int n = sizeof (array) / sizeof (array[0]);

    countSort ( array , n ) ;
    display ( array , n ) ;

    return 0;
}.
```

output[count[a[i]]-1] = a[i];
count[a[i]] --;

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

count =

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

* Parenthesis matching :-

→ checks whether for every opening bracket, the closing bracket is given or not.

→ ex:. ((a + b) * (c - d))
= ((a +) b (* c - d))

→ if opening bracket then push it in stack and if closing bracket then pop from the stack.

→ At last, if stack is empty then the parenthesis are matching otherwise NOT matching.

→ Practice :① ((a + b) * (c - d))

② { ([a + b] * [c - d]) }

17 TUESDAY

MARCH 2020

M A R

30 31
2 3 4 5 6 7 8 9 10 11 12 13 14
16 17 18 19 20 21 22 23 24 25 26 27 28
M T W T F S S M T W T F S

(1) Infix : Operand operator Operand

→ ex : a + b

(2) Prefix : Operator operand operand

+ a b

(3) Postfix : operand operand Operator

a b +

* Conversions :-

(1) a + b * c

↓

a + (b * c)

Prefix : a + [* b c]

+ a * b c

Postfix : a + [b c *]

a b c * +

(2) a + b + c * d

Prefix : a + b + [*cd]

[+ab] + [*cd]

+ +ab *cd

Postfix : a + b + [cd*]

[ab+] + [cd*]

ab+ cd*+

2020

(3) (a+b) * (c-d)

Prefix : [+ab] * [-cd]

* +ab -cd

Postfix : [ab+] * [cd-]

ab + cd - *

Associativity :-

Symbol	Prese.	Associ.
+, -	1	L → R
*, /	2	L → R
^	3	R → L
-	4	R → L (Unary operators)
( )	5	L → R

(1) $((a+b)+c)-d)$

Post ~~Pre~~fix : $ab+c+d-$

Pre ~~Post~~fix : $-++abcd$

(2) $a \uparrow b \uparrow c \longrightarrow (a \uparrow (b \uparrow c))$

Postfix : $a\,[bc\uparrow]\uparrow$ (R → L Associativity)

$abc\uparrow\uparrow$

(3) $-a$ (Unary minus)

Prefix : $-a$

Postfix : $a-$

(4) $*p$

Prefix : $*p$

Postfix : $p*$

★ First Complete all the unary operators then go for others.

21 SATURDAY MARCH 2020

MAR
30 31
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29
M T W T F S S M T W T F S S

(5) n!

Prefix : !n

Postfix : n!

(6) log x

Prefix : log x

Postfix : x log

★ Examples :-

(1) - a + b * log n! ↳ Unary

Postfix : - a + b * log [n!]

- a + b * [n! log]

22 SUNDAY

[a -] + b * [n! log]

[a -] + [b n! log *]

[a -] a - b n! log * +

2020

→ Operator lower precedence જ or same precedence જ
આવે તો પહેલાંનાં બધાં operator pop કરી દેવાનાં.

MONDAY 23
MARCH 2020

Prefix : [-a] + b * log[n!] R → L

[-a] + b * [log n!] (∵ Unary one there)

[-a] + [* b log n!]

+ -a * b log n!

* Infix to Postfix :-

Sym.	Pre	Assoc
+, -	1	L → R
*, /	2	L → R

(1) a + b * c - d / e

Sym.	stack	Postfix
a	——	a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
-	-	abc*+
d	-	abc*+d
/	/, -	abc*+d
e	/, -	abc*+de
		abc*+de/-

(4) A + (B * C - (D / E ^ F) * G)

Symbol	Action Taken	Stack	Postfix
A	print A	—	A
+	push A	+	A
(	push (	(+	A
B	print B	(+	AB
*	push *	*(+	AB
C	print C	*(+	ABC
—	pop * push -	-(+	ABC*
(	push (	(-(+	ABC*
D	print D	(-(+	ABC*D
/	push /	/(-(+	ABC*D
E	print E	/(-(+	ABC*DE
^	push ^	^/(-(+	ABC*DE
F	print F	^/(-(+	ABC*DEF
)	pop ^/	-(+	ABC*DEF^/
*	push *	*-(+	ABC*DEF^/
G	print G	*-(+	ABC*DEF^/G
)	pop *-	+	ABC*DEF^/G*-

Result : ABC*DEF^/G*-+

* Evaluation of postfix :-

post fix: opnd 2 / opnd 1

Opnd 2 - Opnd 1

(1) 5 4 6 + * 4 9 3 / + *

Symbol	Opnd 1	Opnd 2	Value	Stack
5	—	—	—	5
4	—	—	—	4, 5
6	—	—	—	6, 4, 5
+	6	4	10	10, 5
*	10	5	50	50.
4	—	—	—	4, 50
9	—	—	—	9, 4, 50.
3	—	—	—	3, 9, 4, 50.
/	3	9	9/3=3	3, 4, 50
+	3	4	7	7, 50.
*	7	50	350	350.

(2) 3 * 5 + 6 / 2 - 4

Symbol	Opnd 1	Opnd 2	Value	Stack
3	—	—	—	3
*				
5				
+				
6				
/				
2				
-				
4				

Firstly we need to convert infix into postfix for the evaluation.

∴ Infix : 3 * 5 + 6 / 2 − 4

postfix : [35*] + [62/] − 4

[35 * 62/ +] − 4

35 * 62/ + 4 −

Opnd2 − Opnd1

post fix Opnd2 / opnd1

Symbol	opnd 1	opnd 2	Value	Stack
3	–	–	–	3
5	–	–	–	5, 3
*	5	3	15	15
6	–	–	–	6, 15
2	–	–	–	2, 6, 15
/	2	6	3	3, 15
+	3	15	18	18
4	–	–	–	4, 18
−	18	4	14	14

29 SUNDAY

* Evaluation of prefix :-

prefix: opnd1 - opnd2
opnd1 / opnd2

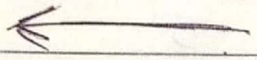
(1) 3 * 5 + 6 / 2 - 4

Infix : 3 * 5 + 6 / 2 - 4

[* 35] + [/62] - 4

[+ *35 /62] -4

- + * 35 /62 4



Symbol	opnd1	opnd2	Value	Stack
-	-	-	-	⊖
+	-	-	-	⊖
*	-	-	-	⊖
3	-	-	-	3
5	-	-	-	5, 3
/	5	3	1.66.	1.66
6	-	-	-	6, 1.66
2				
4				

From prefix start from Right to Left.

Symbol	Opnd 1	opnd 2	Value	stack
	–	–	–	4
4	–	–	–	2, 4
2	–	–	–	6, 2, 4
6	–	–	–	3, 4
/	6	2	3	5, 3, 4
5	–	–	–	3, 5, 3, 4
3	–	–	–	15, 3, 4
*	3	5	15	18, 4
+	15	3	18	14
–	18	4	14	

* Infix to prefix conversion :-

(a+b) * (c-d)

Step : 1 : Reverse the expression

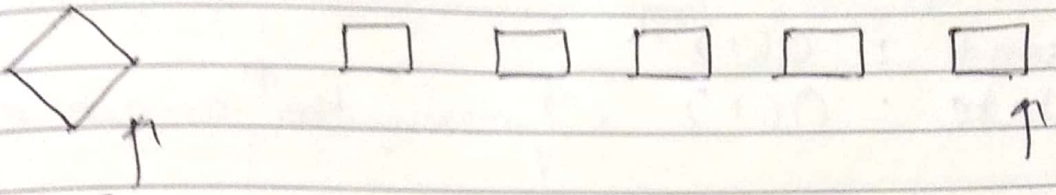
)d-c(*)b+a(

(d-c) * (b+a)

Step : 2 : Convert the expression into postfix.

Step : 3 : Reverse it and You Will Get Prefix

* Queue :-



Queue ADT

* Data :-

1) Space for string elements
2) front - for deletion
3) rear - for insertion.

* Operations :

1) enqueue (data)
2) dequeue ( )
3) isEmpty ( )
4) isFull ( )
5) first ( )
6) last ( )

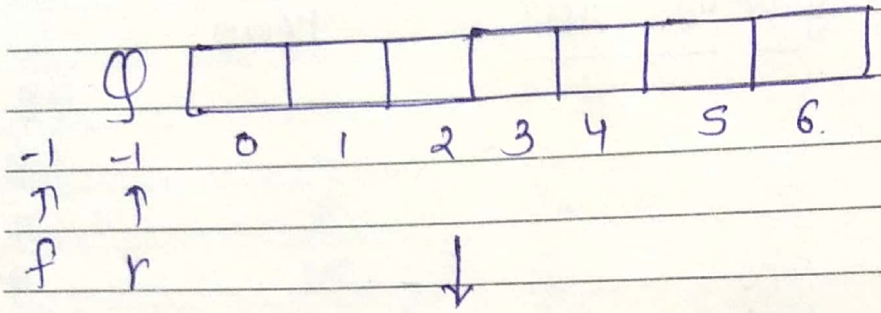
* Queue using Array :

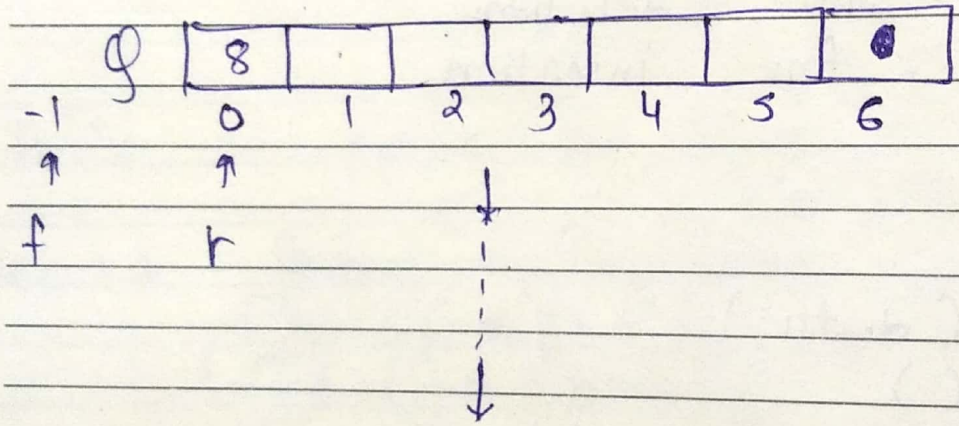
→ Using single pointer rear :-

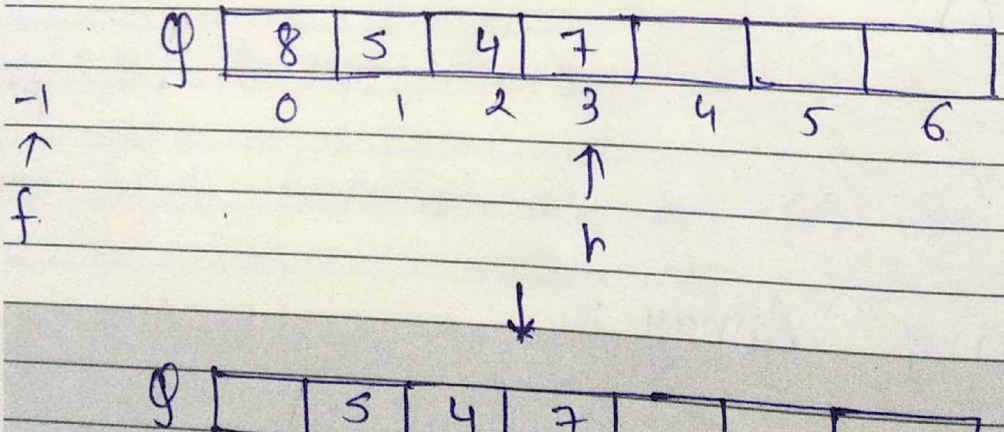
Insert : O(1)
Delete : O(n)

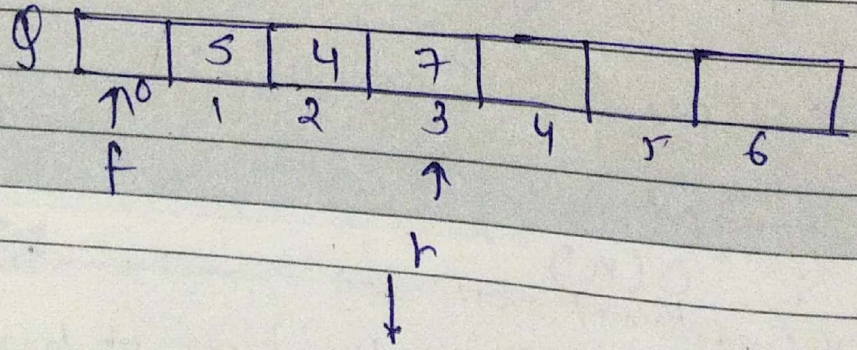
→ Using two pointers front and rear :-

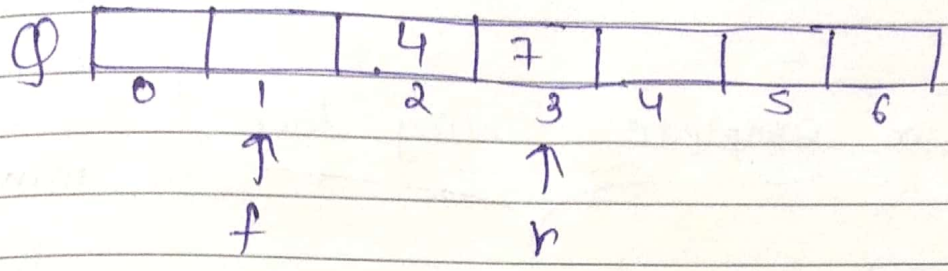
Insert : O(1)
Delete : O(1) (∵ moving the counter using front)











→ While deleting from the front, we don't need to shift all the remaining elements. we just need to shift the front pointer before the queue/first element.

→ Therefore time complexity is O(1) for deletion.

* Initially : front = rear = -1

isEmpty : if (front > rear || front == -1)

isFull : if (rear == size-1)

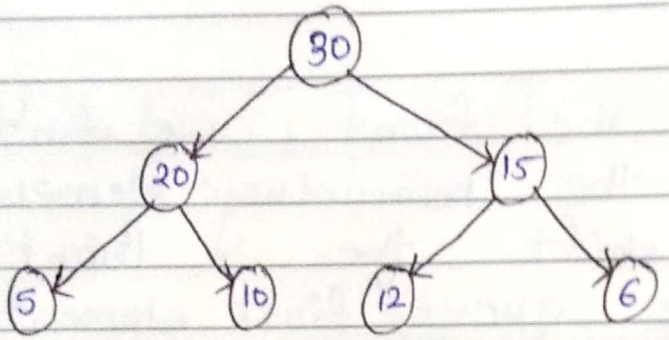
04 SATURDAY
APRIL 2020

* Heap :-

→ Heap is a complete Binary Tree.

Having max heap or min heap pattern.

when we store it's elements then there should not be any empty space.



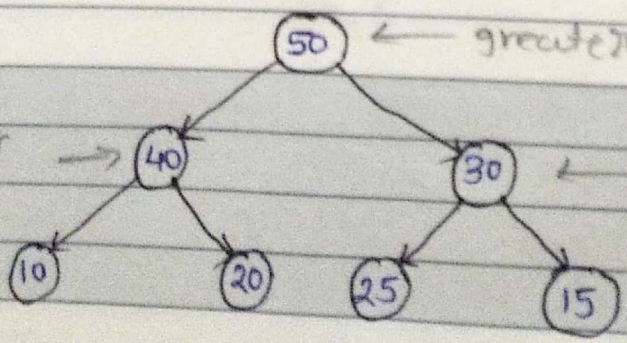
T	30	20	15	5	10	12	6
	1	2	3	4	5	6	7

if i = 0, 1, 2, ... then left child 2i + 1 & right child 2i + 2

→ Node at index i
Left child at 2 * i
Right child at 2 * i + 1

→ Every node should have an element greater than or equal to all it's descendants. then it is called MAX HEAP.
(Duplicates are allowed.)

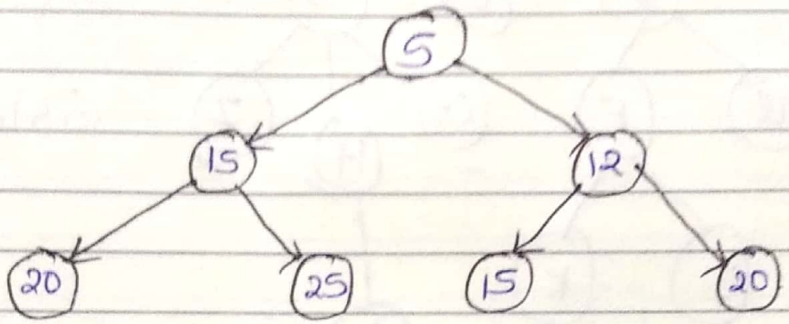
05 SUNDAY



→ O(n.logn) time is taken for creating an entire heap.

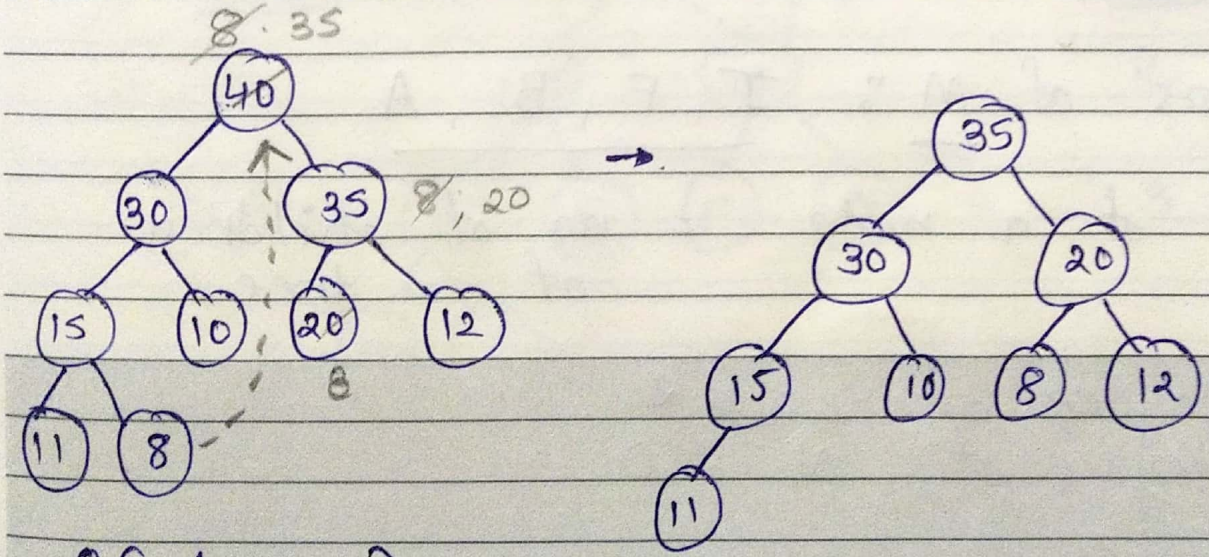
APRIL 2020 | MONDAY | 06

→ If every node has an element lesser than or equal to all its descendants then it is called MIN HEAP.
(Duplicates are allowed.)



→ Height of a complete binary tree is logn.
Therefore the height of a heap is logn.

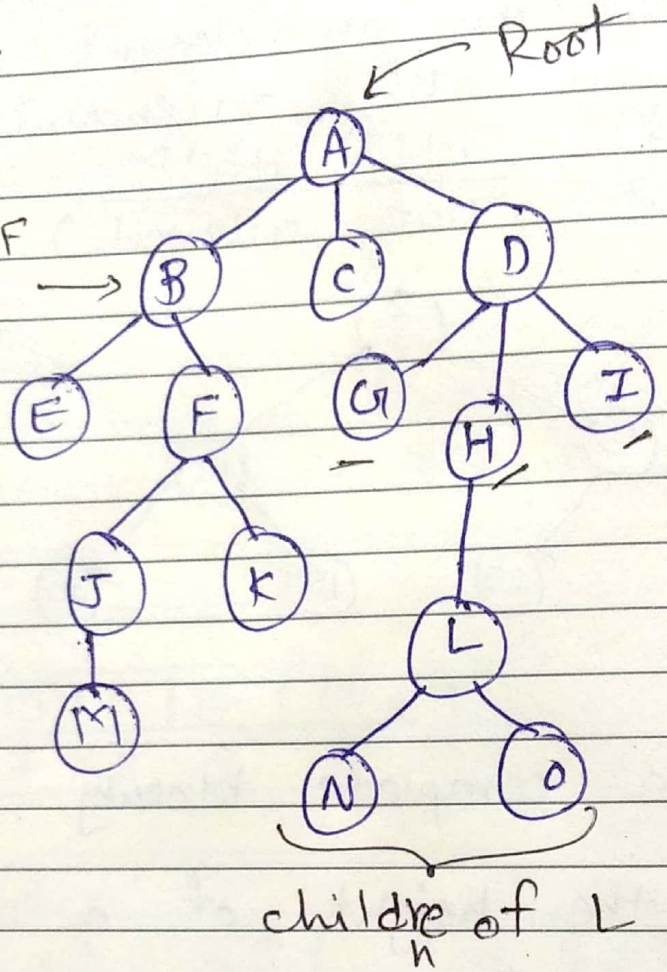
→ delete_max():



O(logn) for deleting & inserting

07 TUESDAY
APRIL 2020

* Tree :-



→ Descendants of B : E, F, J, K, M

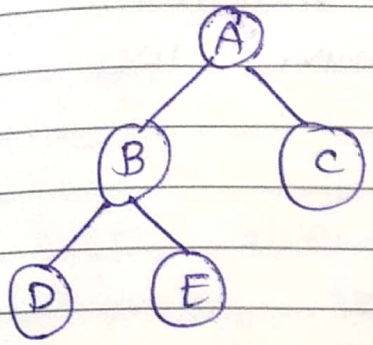
→ Ancestors of M : J, F, B, A

→ Degree of a node = no. of children of a tree.

∴ Degree of L = 2
Degree of N = 0

2020

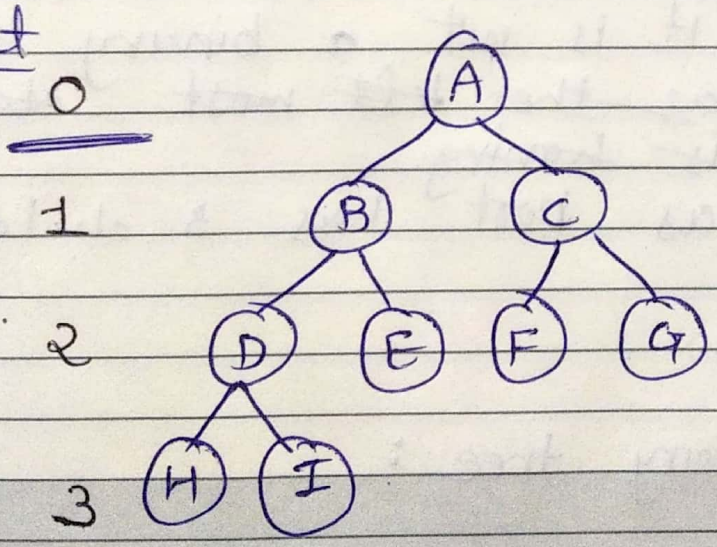
→ Leaf nodes : Nodes with degree zero is called leaf nodes



C, D, E, are leaf nodes/ External nodes/ Terminal nodes.

→ Nodes having degree >0 is called internal nodes.

like in above example, B is internal node
Non-leaf node / Non-leaf node.

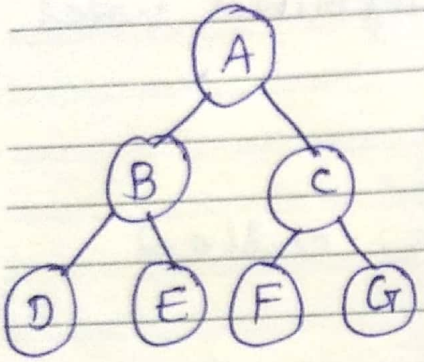


∴ Height is : 3 and levels are : 4.

09 THURSDAY APRIL 2020

* Binary tree :-

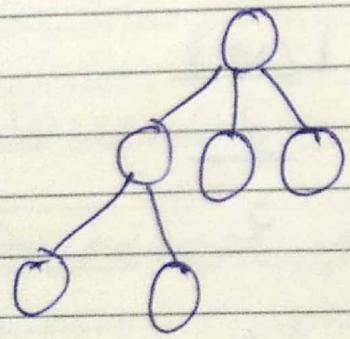
→ Every node should have maximum two children, it can have less than two children.



(T)

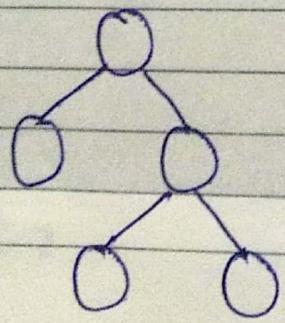
∴ deg(T) = 2

children can be {0, 1, 2}

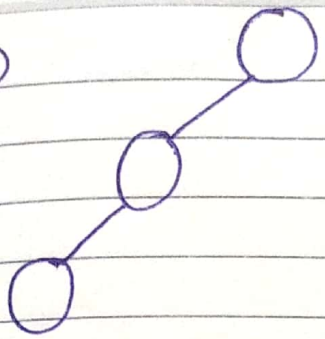
→ 

it is not a binary tree as ~~the left most child is having~~ as root has 3 children.

→ Examples of binary tree :

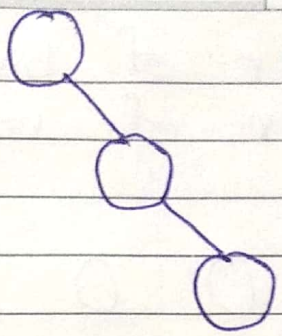
1) 

2)



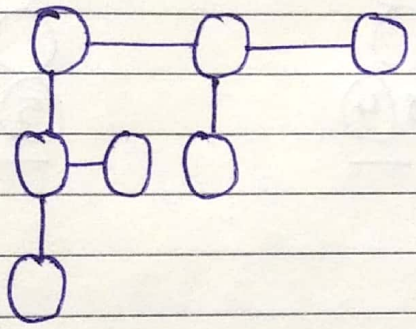
Left skewed binary tree

3)



Right skewed binary tree.

4)



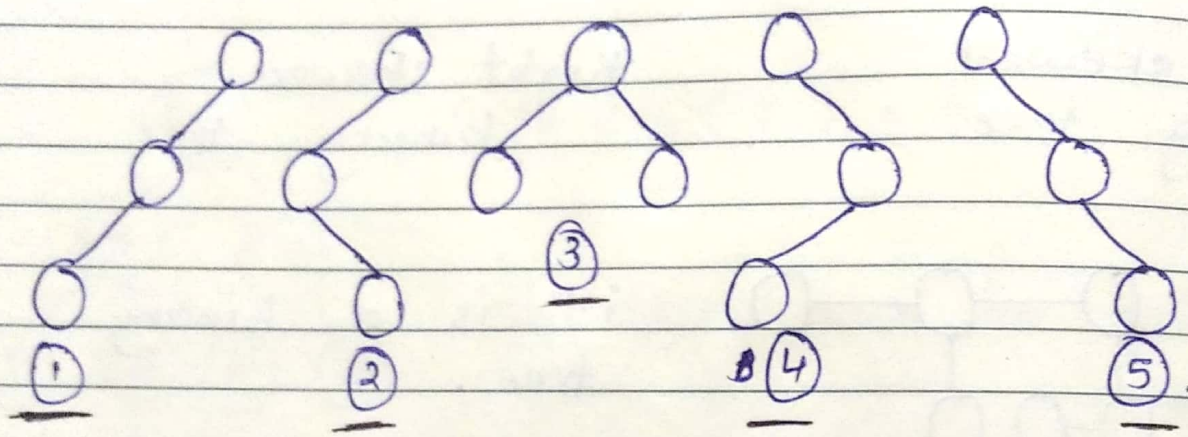
it is a binary tree.

11 SATURDAY APRIL 2020

Unlabelled Nodes

* Number of binary trees from given number of nodes :-

1) O O O 3 nodes



∴ T (3) = 5.

like wise T (4) = 14.

* Formula for counting number of binary trees which can be generated from the given number of nodes.

12 SUNDAY

Catalan Formula. $T(n) = \frac{2nC_n}{n+1}$

2020

∴ 5 nodes is given.

$$\therefore T(5) = \frac{2\times 5 C_5}{5+1} = \frac{10 C_5}{6}$$

$$= \frac{10\times 9\times 8\times 7\times 6}{5\times 4\times 3\times 2\times 1} \Big/ 6$$

$$= 2\times 3\times 7$$

$$\therefore T(5) = 42$$

→ No. of trees having maximum height

$$= 2^{n-1}$$

∴ for $n = 3$ $\quad 2^{3-1} = 2^2 = 4$

14 TUESDAY APRIL 2020

Labelled Nodes.

** No. of binary trees can be generated from the given no. of nodes.

$$T(n) = \frac{2nCn}{n+1} * n!$$

Each tree can be drawn in $n!$ types.

∴ For $n = 3$ (A) (B) (C)

$$T(3) = \frac{2nCn}{n+1} \cdot n! = \frac{2\times3C_3}{3+1} \cdot 3!$$

$$= \frac{6C_3}{4} \times 6$$

$$= \frac{6\times5\times4}{3\times2\times1} \times 6$$

$$= 5 \times 6$$

$$= 30$$

2020

* Height vs Nodes :-

height = h
nodes = n.

→ if height is given ...

Min nodes $n = h + 1$
Max nodes $n = 2^{h+1} - 1$

→ if nodes are given ...

Max ~~Min~~ height $h = n - 1$

Min ~~Max~~ height $h = \log_2(n+1) - 1$

* Height of a binary tree :

$\log_2(n+1) - 1 \leq h \leq n - 1$

$O(\log n)$ $O(n)$.

* No. of nodes of a binary tree :

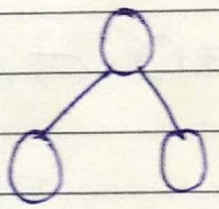
$h + 1 \leq n \leq 2^{h+1} - 1$

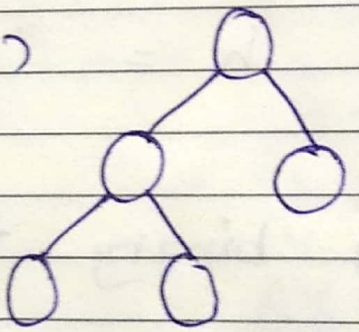
→ deg(0) = deg(2) + 1
(for binary tree)

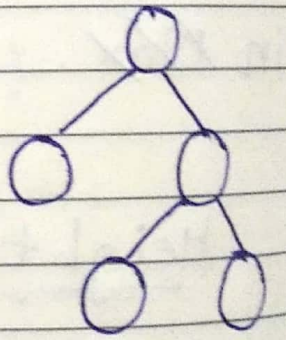
* Strict Binary Trees *

→ Each node should have 0 or exactly 2 children. {0, 2}

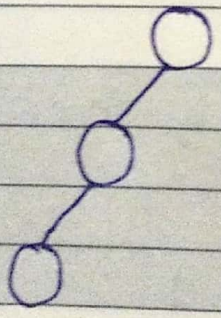
→ Ex:-

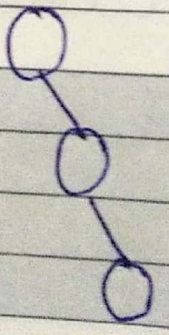
1) 

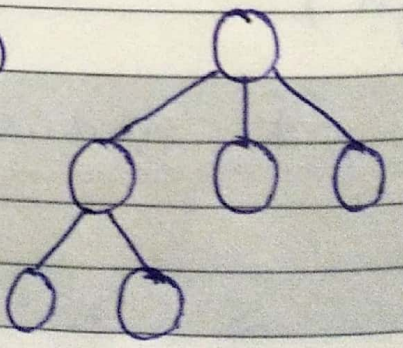
2) 

3) 

→ Non-examples :-

1) 

2) 

3) 

* Height vs Nodes *

→ if height 'h' is given,

Min nodes $n = 2h + 1$

Max nodes $n = 2^{h+1} - 1$

→ if 'n' nodes are given,

Max ~~Min~~ height $h = \frac{n-1}{2}$

Min ~~Max~~ height $h = \log_2(n+1) - 1$.

→ Height of binary tree :

$\log_2(n+1) - 1 \leq h \leq \frac{n-1}{2}$

$O(\log n)$ $O(n)$. *

→ No. of nodes of binary tree :

$2h+1 \leq n \leq 2^{h+1} - 1$

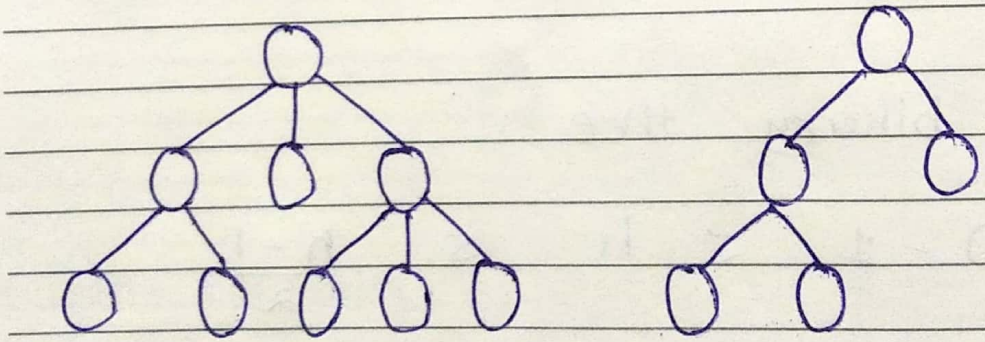
→ $e = i + 1$ Always true for strict binary tree.

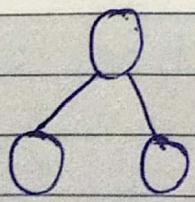
no of external nodes — no of internal nodes.

* m - ary trees :-

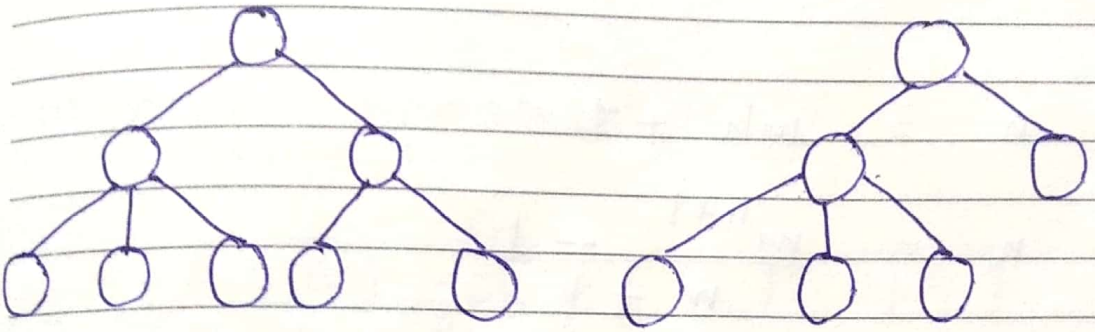
→ every node in a tree can have atmost 'm' children, not more than that.

→ 3 - ary tree {0, 1, 2, 3}





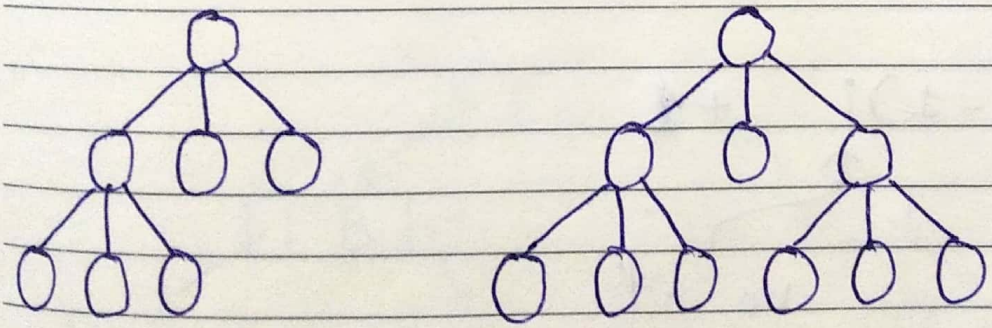
→ 4 - ary tree {0, 1, 2, 3, 4}



* Strict m - ary trees :-

→ Every node in a tree can have either 0 children or exactly 'm' children.

→ Strict 3 - ary tree {0, 3}



→ if height is given,

Min nodes $n = mh + 1$

Max nodes $n = \frac{m^{h+1} - 1}{m - 1}$

→ if nodes are given,

Min height $h = \log_m [n(m-1)+1] - 1$

Max height $h = \frac{n-1}{m}$

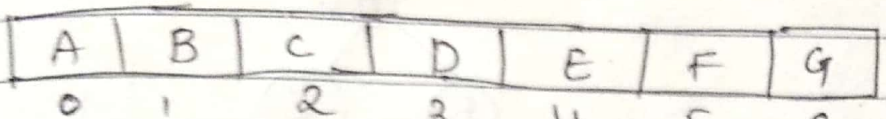
→ $e = (m-1)i + 1$

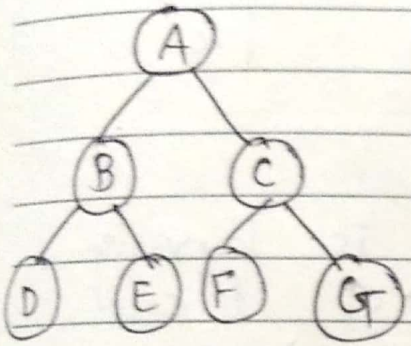
e: no of external nodes

i: no of internal nodes

* Representation of Binary Tree :-

1) Array representation :





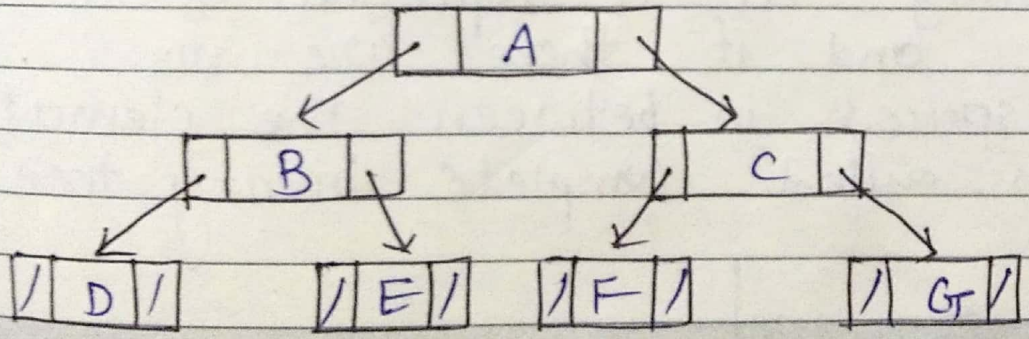
for element i

L. child : $2i + 1$

R. child : $2i + 2$.

Parent's index = $\frac{i_{LR} - 1}{2}$

2) Linked representation :



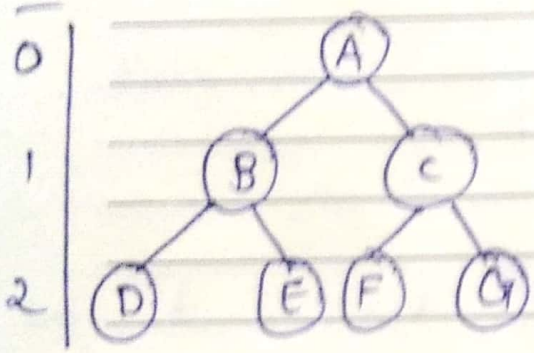
→ no. of NULL pointers = $n + 1$

where n = no. of nodes.

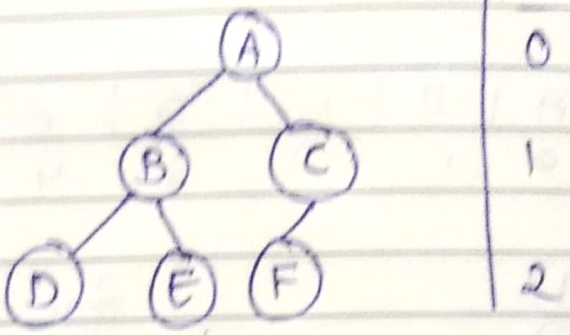
23 THURSDAY APRIL 2020

* Full vs Complete Binary Tree :-

Full BT

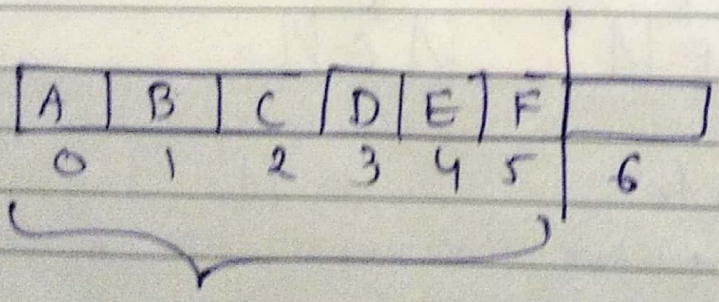


Complete BT



1) → if a tree of height h is having maximum number of nodes it can have, then it is called full binary tree. (Max node = $2^{h+1} - 1$)

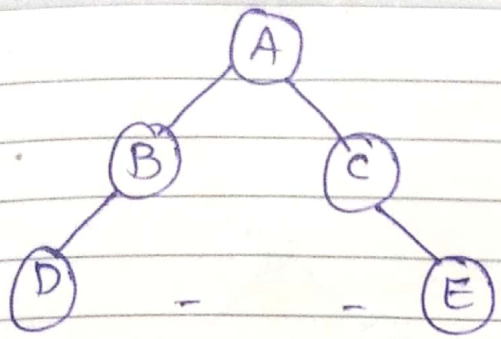
2) → if a binary tree is represented in an array and if there are no blank spaces in between the elements then it is called complete binary tree.



No space in between

∴ Complete Binary Tree.

2020

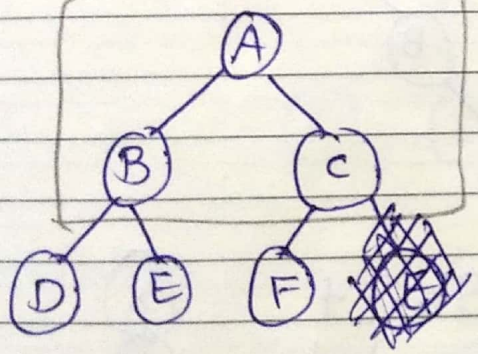


A	B	C	D			E
0	1	2	3	4	5	6

blank spaces are there in between the array elements.

∴ it is NOT a complete binary tree.

→ A complete binary tree of height 'h' will be a full binary tree up to 'h-1' height.



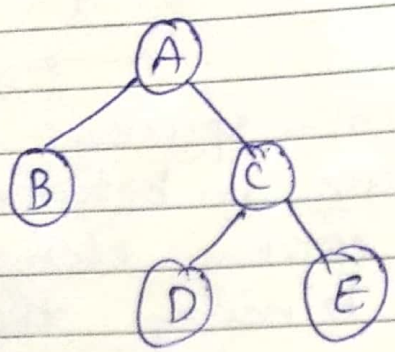
height	
0	Full binary tree.
1	
2	

∴ up to h-1 = 2-1 = 1

→ A Full binary tree is always a complete binary tree. but reverse may not be true.

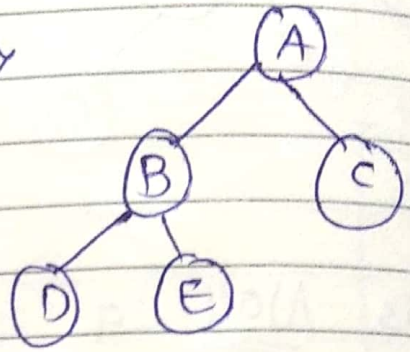
* Examples :-

1)



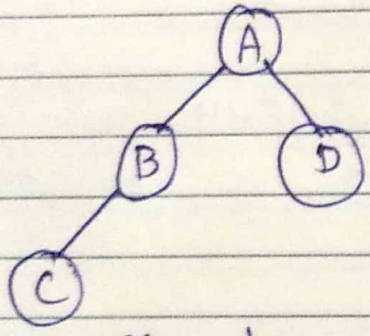
Strict ✓
Complete ✗

2)



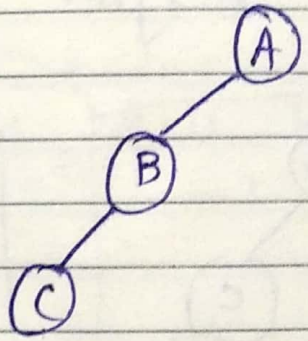
strict ✓
Complete ✓

3)



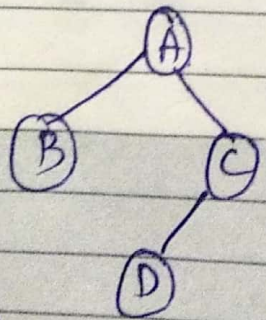
Strict ✓
Complete ✗

4)



Strict ✗
Complete ✗

26 SUNDAY 5)



Strict ✗
Complete ✗

2020

* Tree Traversals *

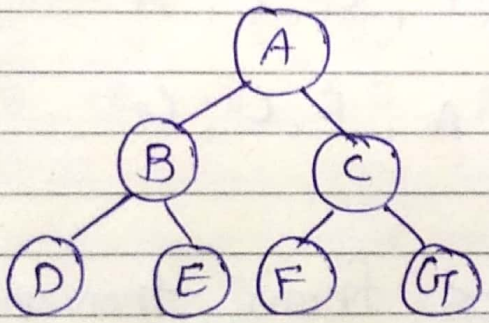
→ Inorder : L V_r R

Preorder : V_r L R

Postorder : L R V_r

Level order : level by level traversing

→ Ex :-



Level order :

A, B, C, D, E, F, G

Inorder : L V_r R

↳ D, B, E, A, F, C, G

Preorder : V_r L R

↳ A, B, D, E, C, F, G.

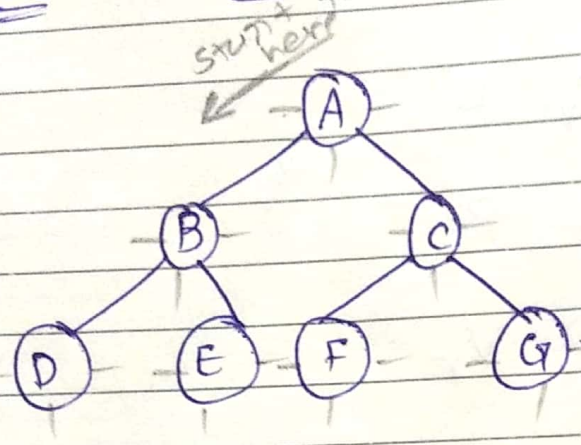
Post order : L R V_r

↳ D, E, B, F, G, C, A.

28 TUESDAY

APRIL 2020

* Easy methods :-



For pre : A

For post : A

For in : A

∴ pre : A, B, D, E, C, F, G

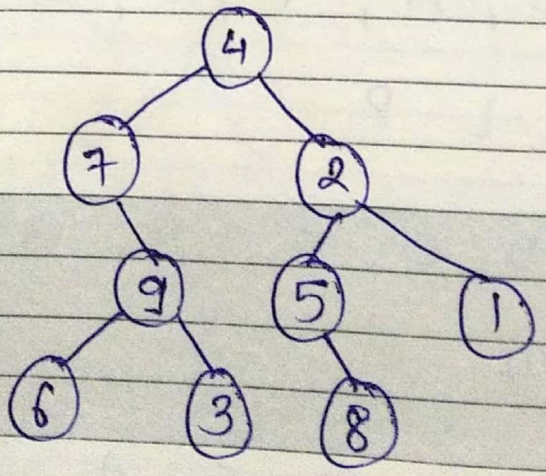
∴ post : D, E, B, F, G, C, A

∴ In : D, B, E, A, F, C, G

* Generating trees from traversals :-

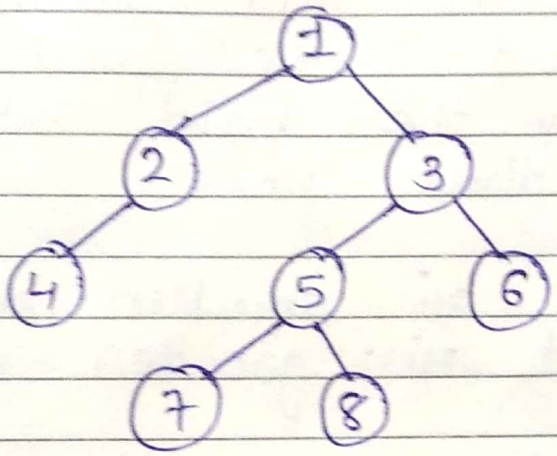
1) Pre : 4, 7, 9, 6, 3, 2, 5, 8, 1

In : 7, 6, 9, 3, 4, 5, 8, 2, 1

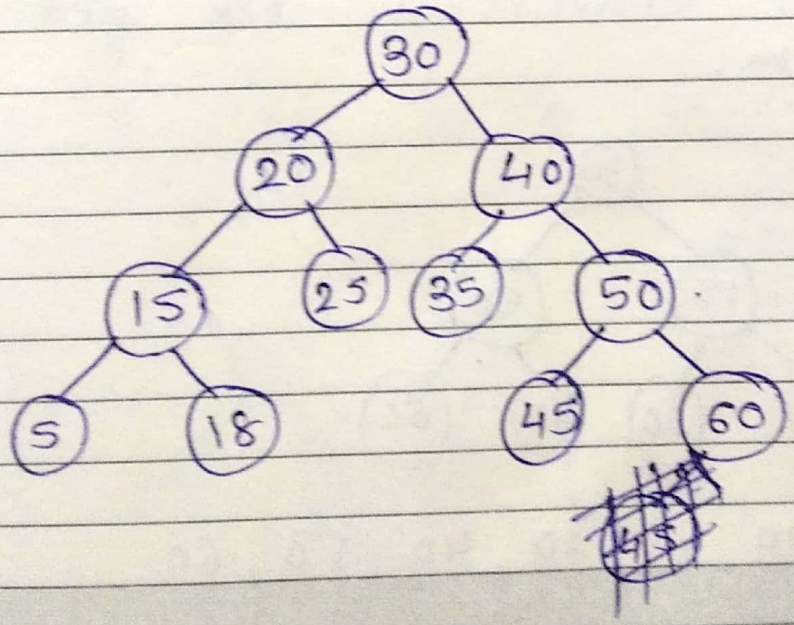


2020

2) In : 4, 2, 1, 7, 5, 8, 3, 6
pre : 1, 2, 4, 3, 5, 7, 8, 6



3) In : 5, 15, 18, 20, 25, 30, 35, 40, 45, 50, 60
post : 5, 18, 15, 25, 20, 35, 45, 60, 50, 40, 30



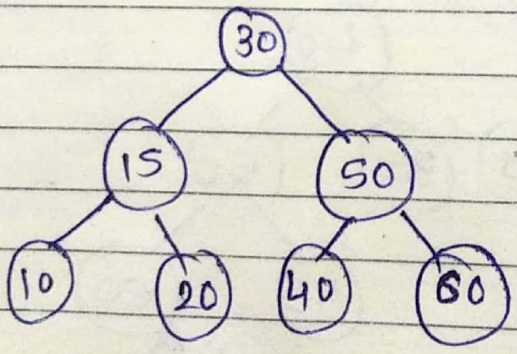
30 THURSDAY APRIL 2020

* Binary Search Tree :-

→ A binary tree in which, every node, all the elements in the left subtree are smaller than that node and all the elements in the right subtree is greater that that node for

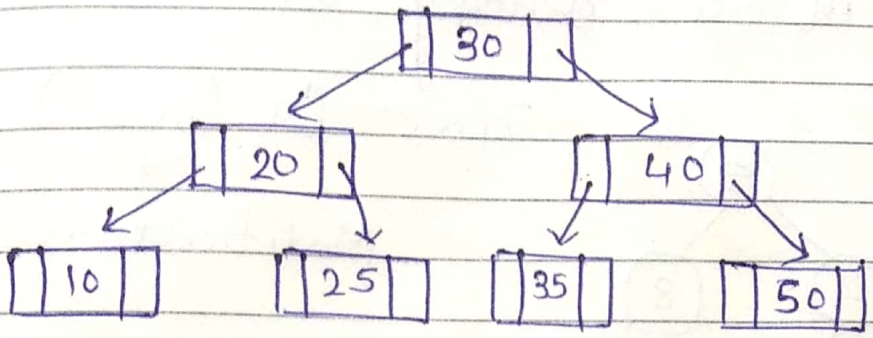
→ In short, Left ki smaller and Right ki greater elements and NO duplicates are allowed.

→ In in-order traversal we get the sorted form.



In order : 10, 15, 20, 30, 40, 50, 60,

→ No. of binary search tree can be generated from given n node with same in-order is : $T(n) = \frac{2nC_n}{n+1}$



h
0
1
2

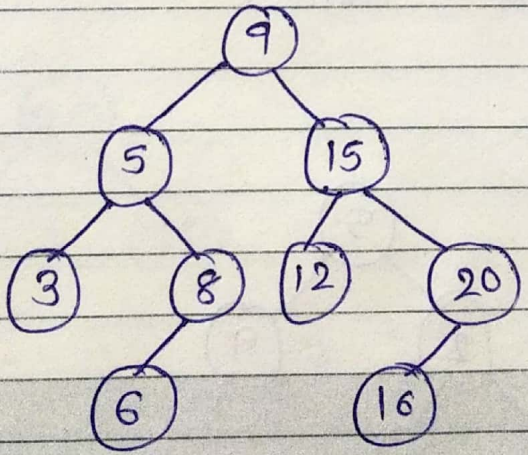
→ For searching time complexity is

$O(h) = O(\log n)$

$\log n \leq h \leq n$

* Creating Binary Search Tree :-

9, 15, 5, 20, 16, 8, 12, 3, 6

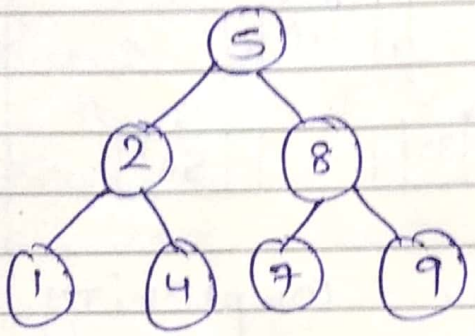


→ each time start from root

Time : $O(n \cdot \log n)$

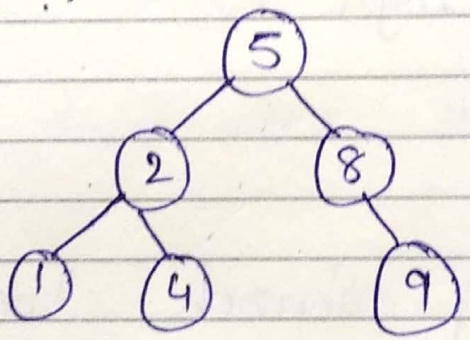
* Deleting in a binary search tree :-

Time: O(logn)

1) 

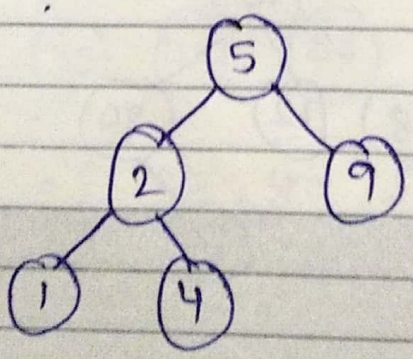
Modifications : O(logn)

Delete (7) :-



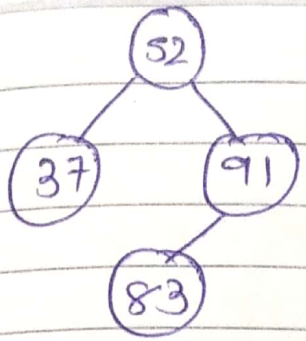
→ when it's a leaf node then just remove it and done.

2) Delete (8) :

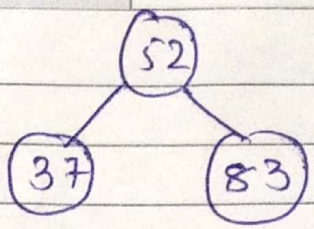


→ if deleted node has only one child then just promote it and done

3)

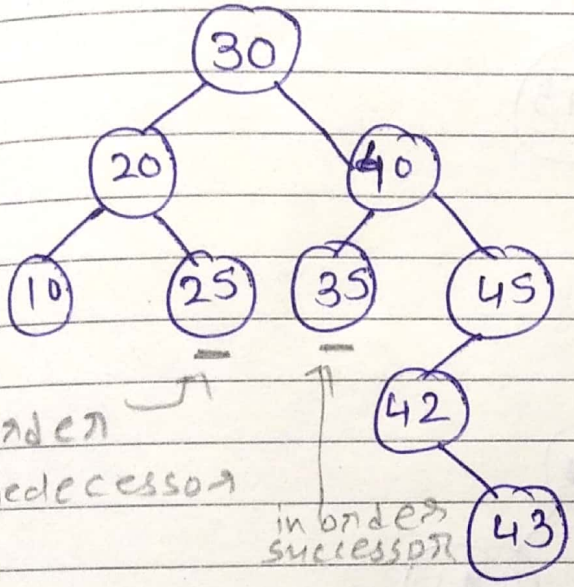


delete (91) :



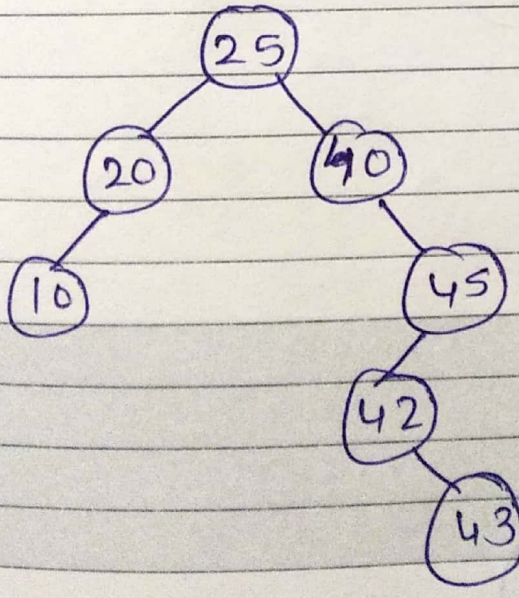
just promote it when it has only one child.

4)

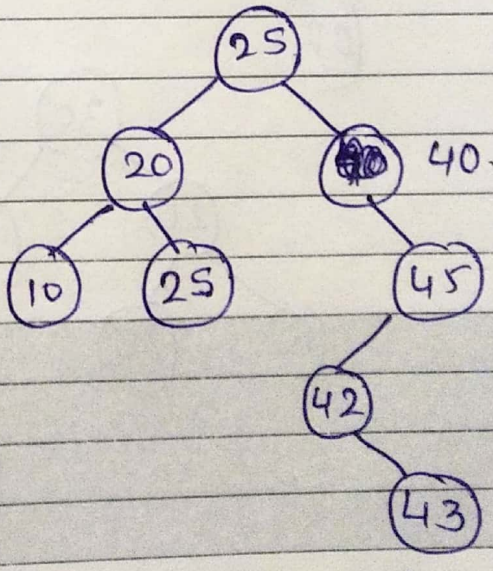


→ when we want to delete a node having more children and levels then it's in-order predecessor or it's in-order successor will take place of that deleted node.

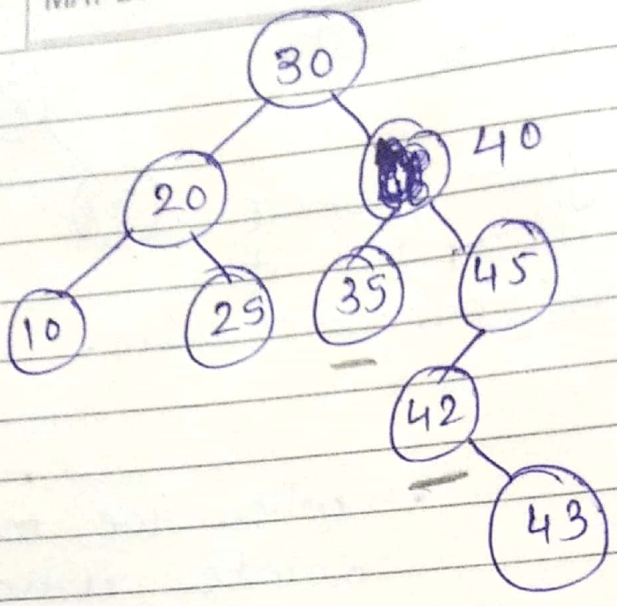
delete (30) :-



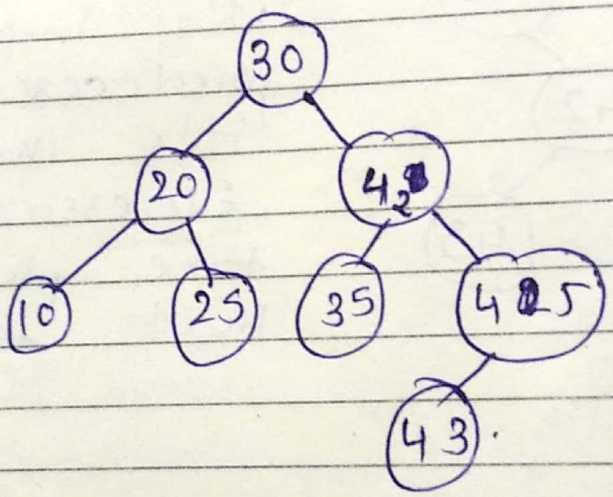
OR



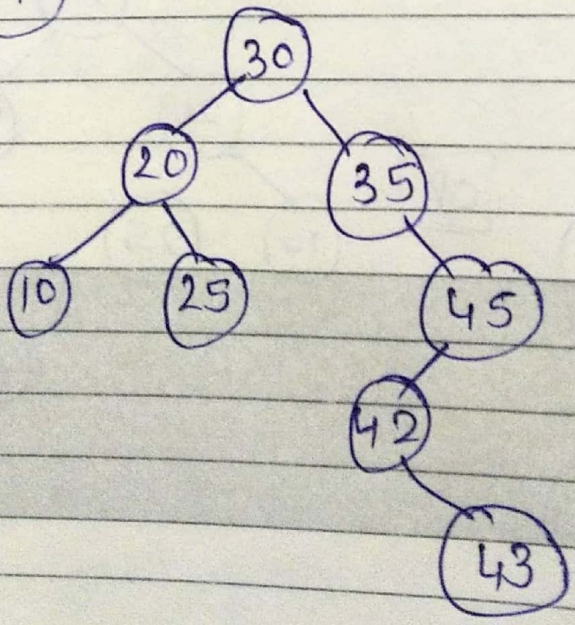
5).



delete (40) :-

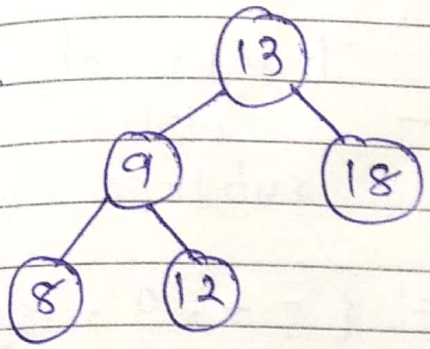


OR

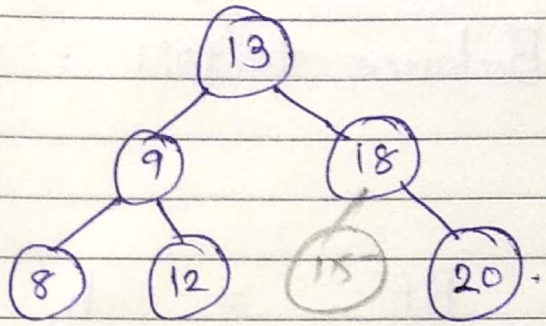


* Inserting in a binary search tree :-

14.



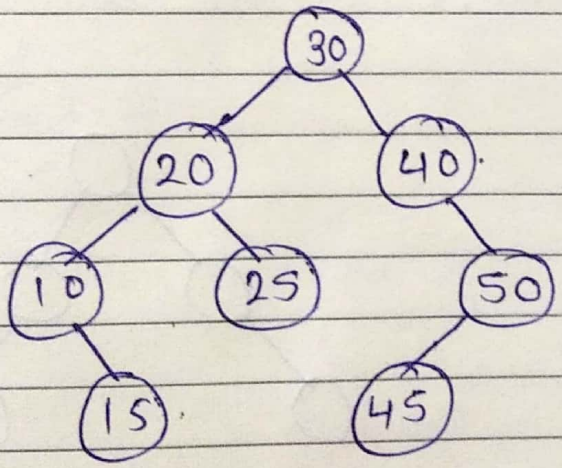
insert :-



That's it !

* Generating BST :-

14 pre : 30, 20, 10, 15, 25, 40, 50, 45
in : 10, 15, 20, 25, 30, 40, 45, 50.



07 THURSDAY MAY 2020

* AVL Tree :-

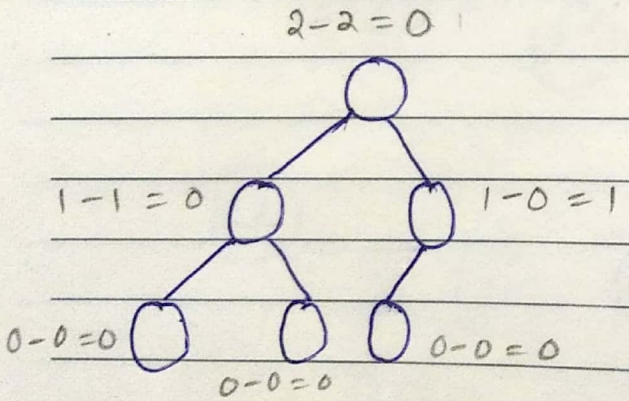
→ Balance factor = height of left subtree − height of right subtree

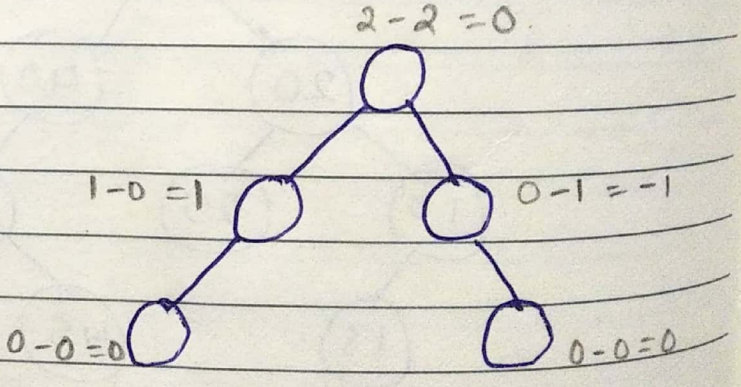
$BF = h_l - h_r = \{-1, 0, 1\}$ (Valid)

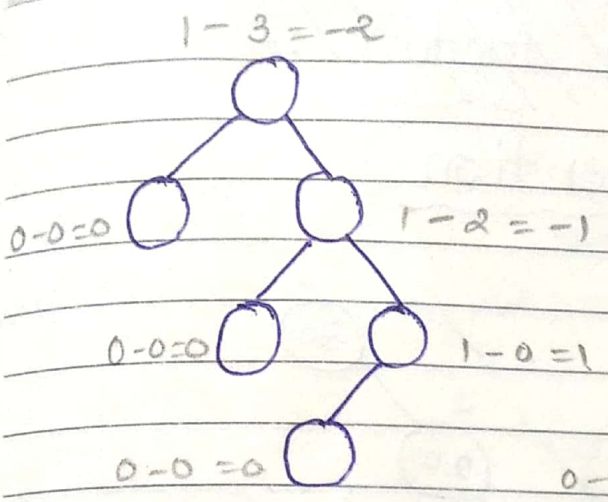
if $|BF| = |h_l - h_r| \leq 1$ => Balanced

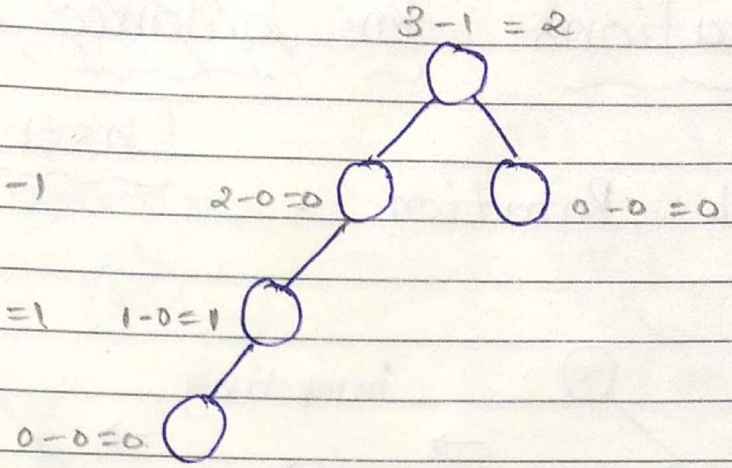
if $|BF| = |h_l - h_r| > 1$ or < -1 => imbalanced

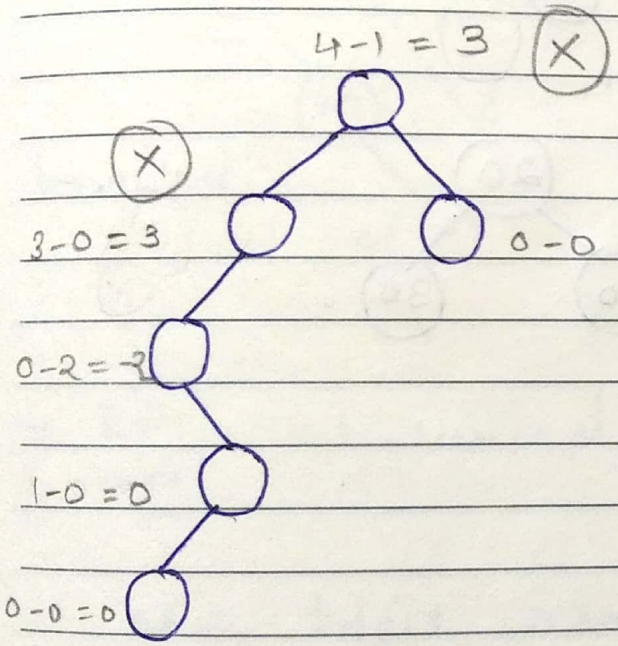
→ Examples :- Here we are considering levels instead of height for the convinience.









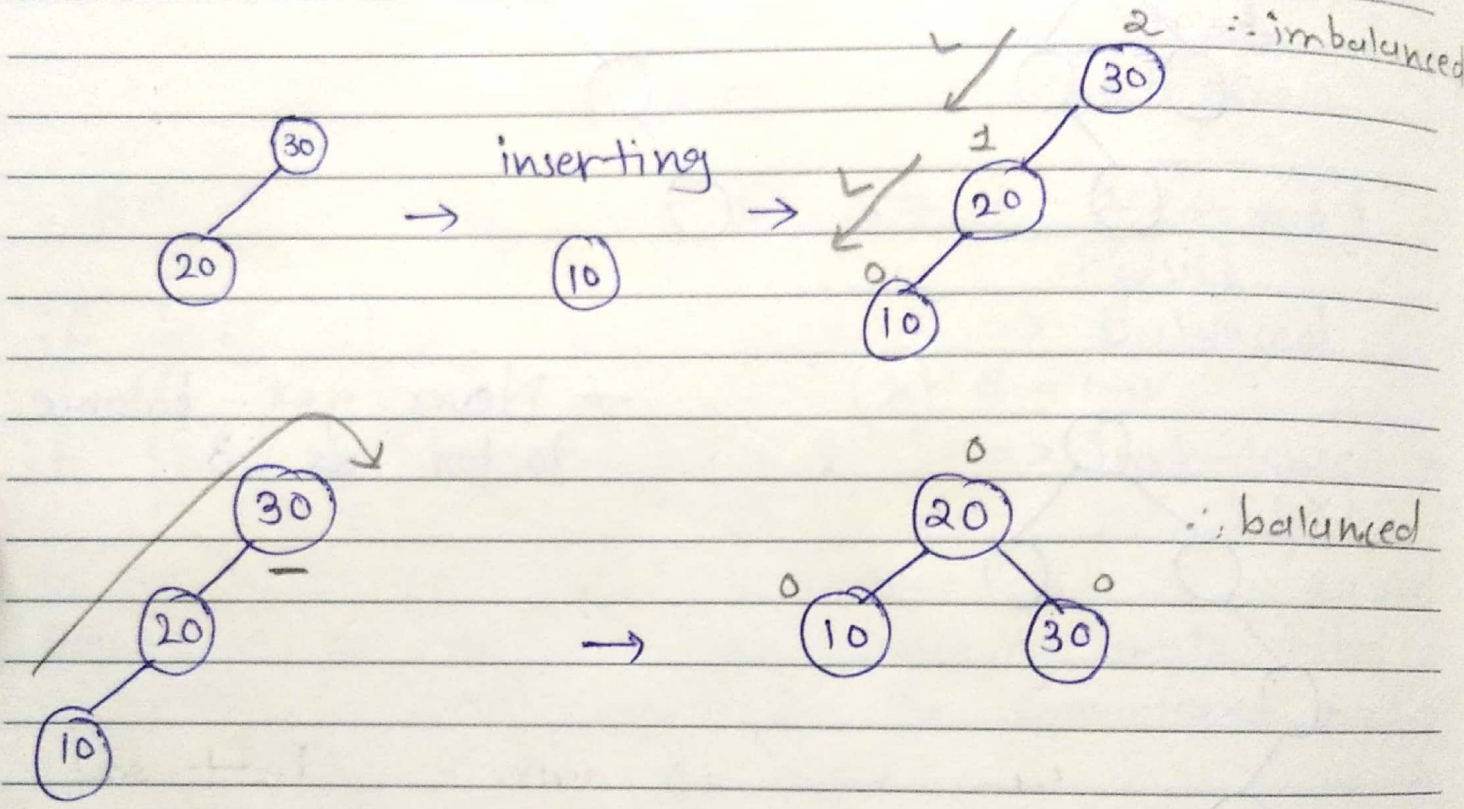


→ Never get balance factor as 3.

* Rotations for balance factor :-

Insertion :-

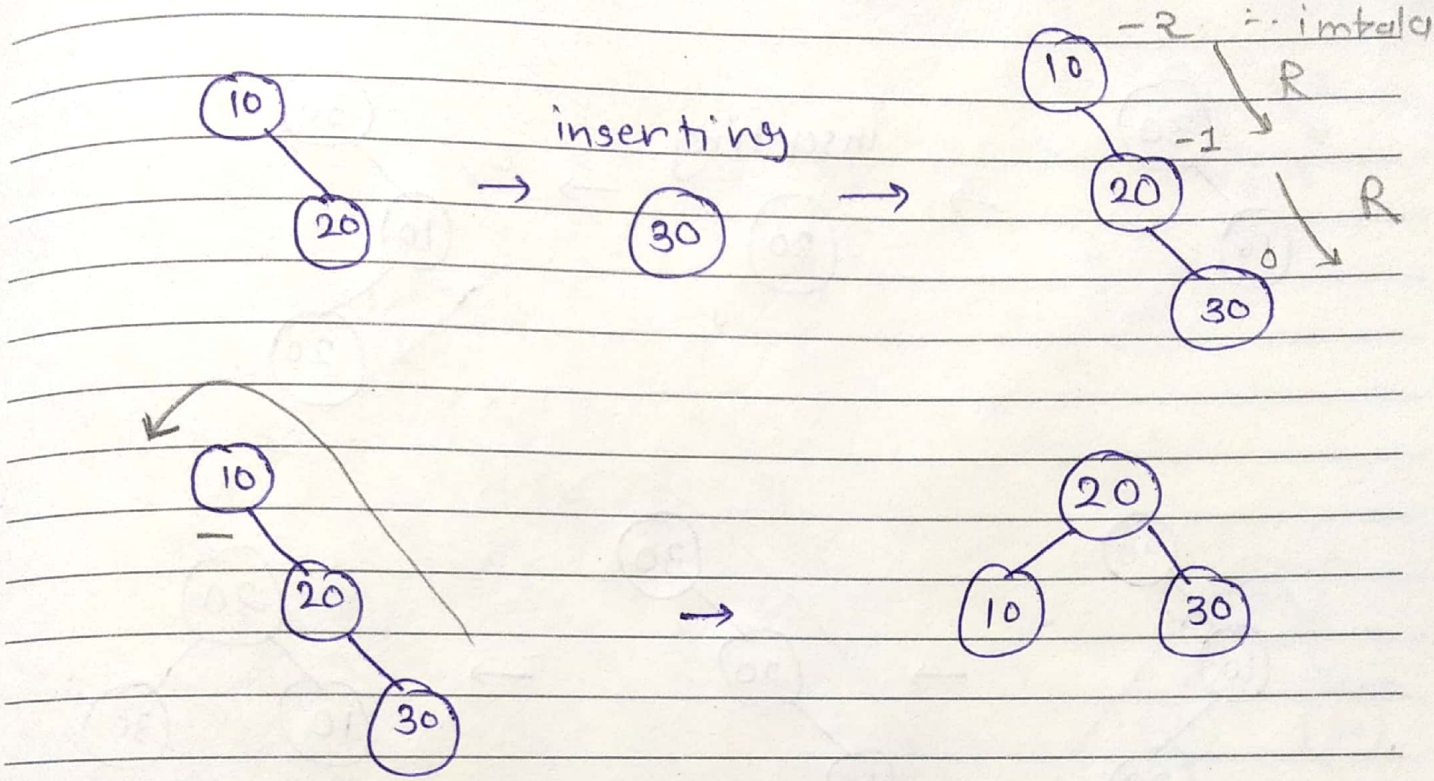
1) LL Rotation :



→ LL imbalanced ⇒ Rotate right side

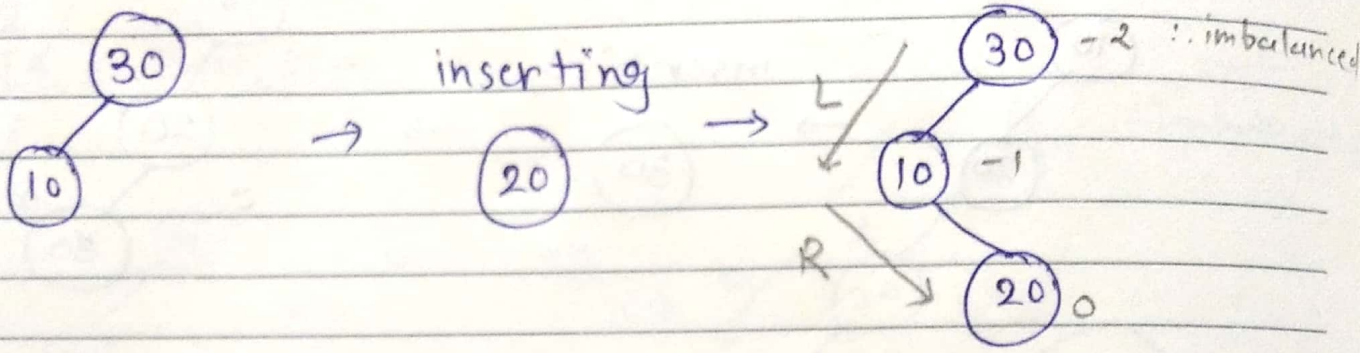
10 SUNDAY

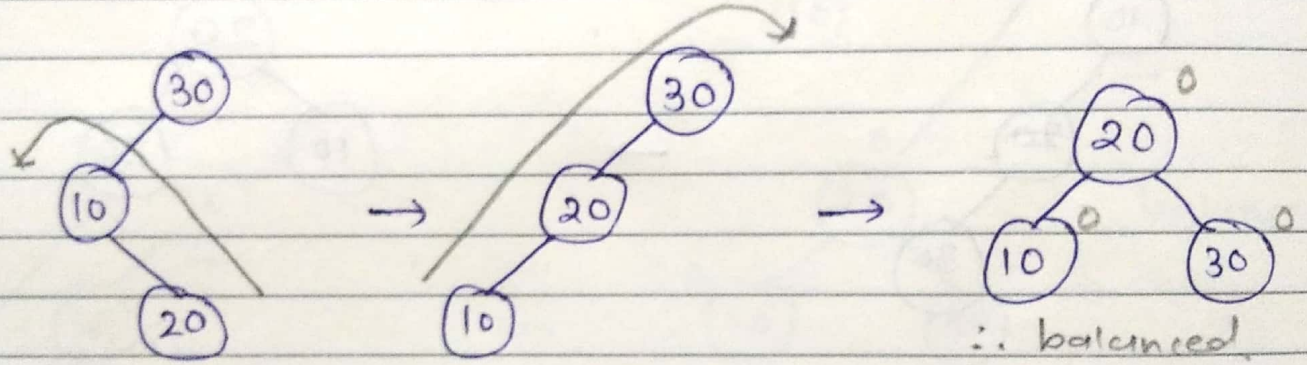
2) RR Rotation :



→ RR imbalanced => Rotate left side

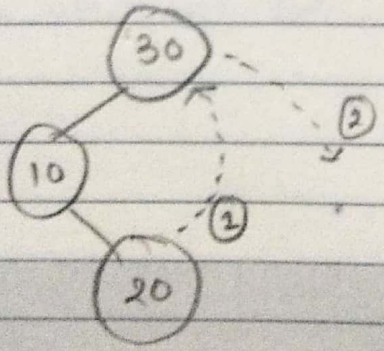
3) LR Rotation :



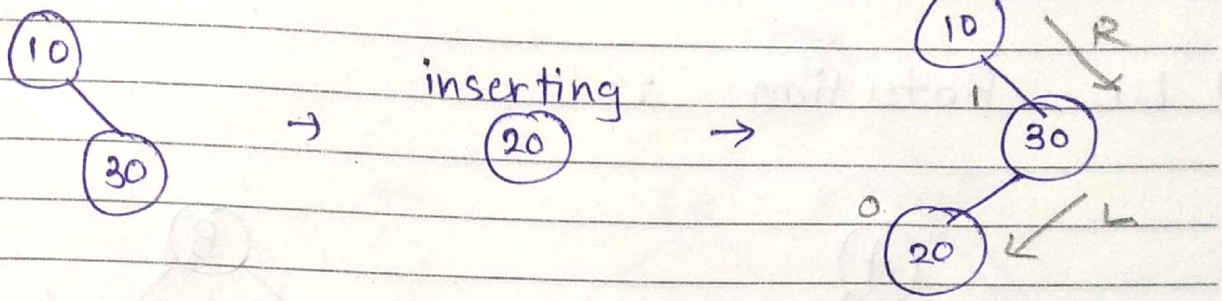


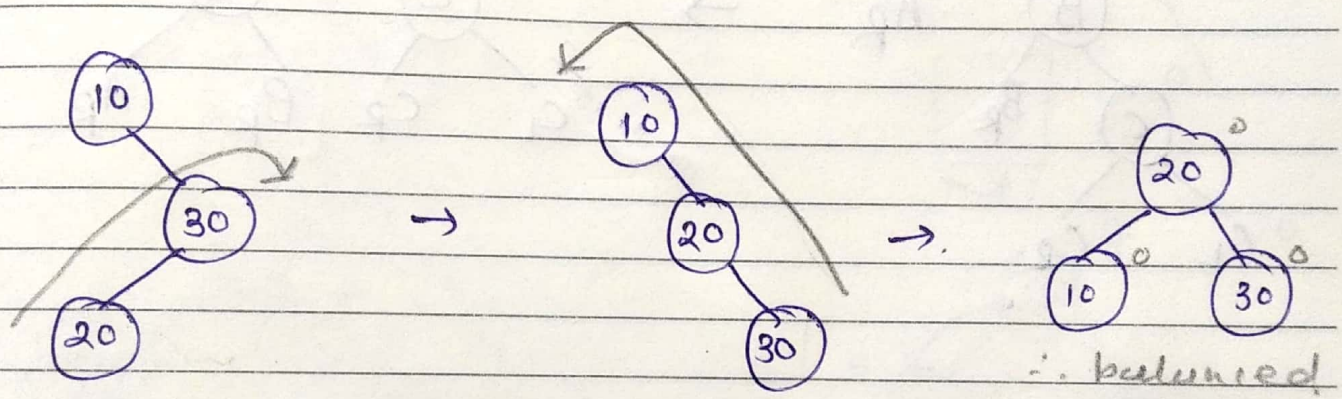
rotate two nodes to left

this is LL rotation.



4) RL Rotation :





rotate two nodes to right

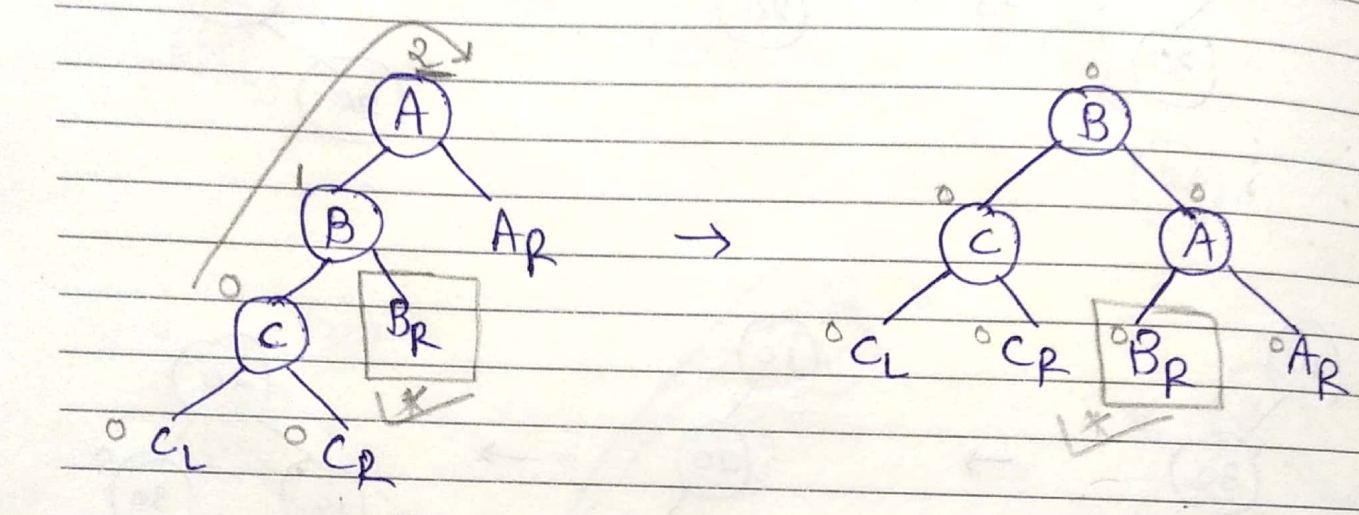
this is RR rotation we get.

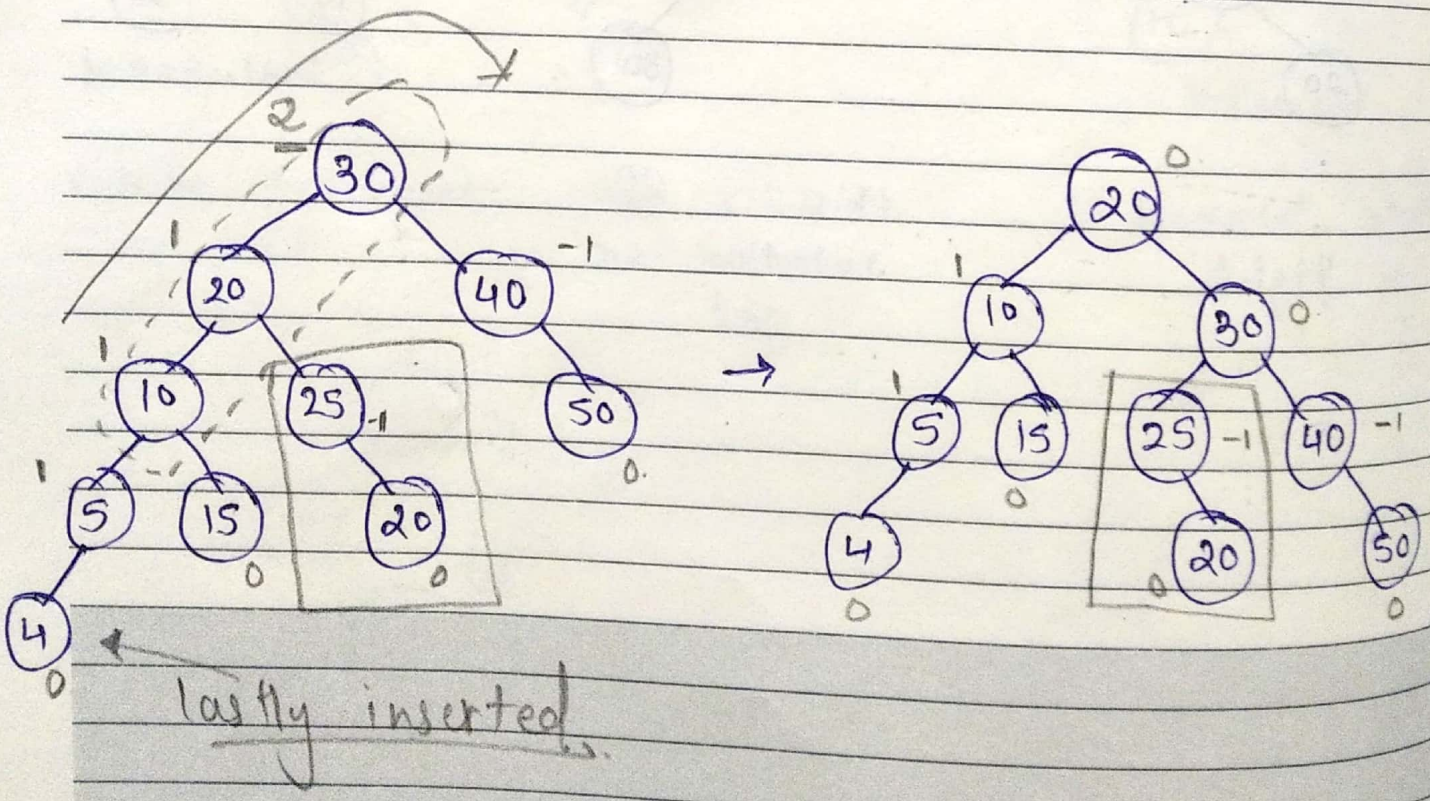
if node has positive balance factor => LL Rotation

14 THURSDAY MAY 2020

* Formula of Rotations for insertion :-

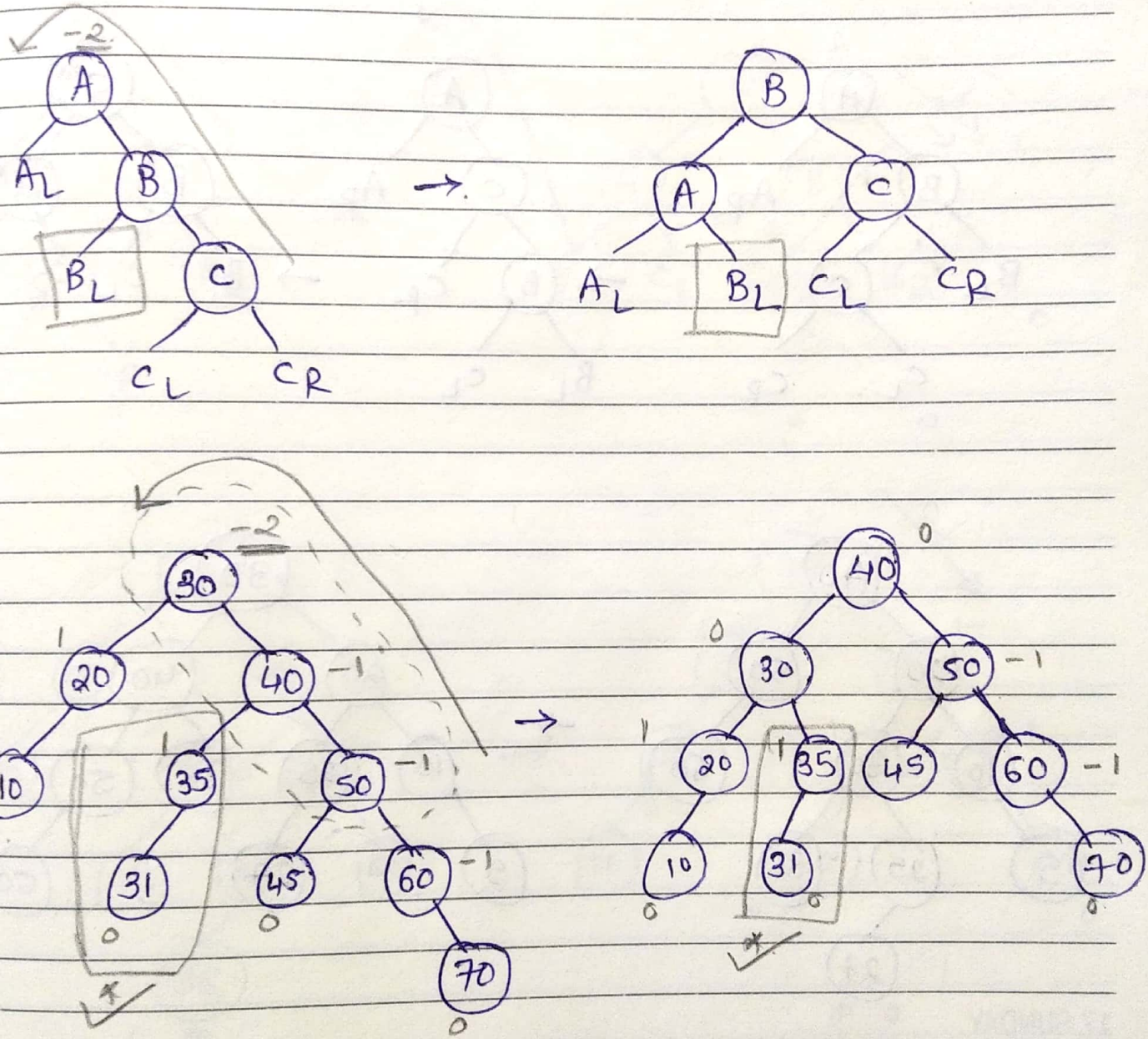
1) LL Rotation :-



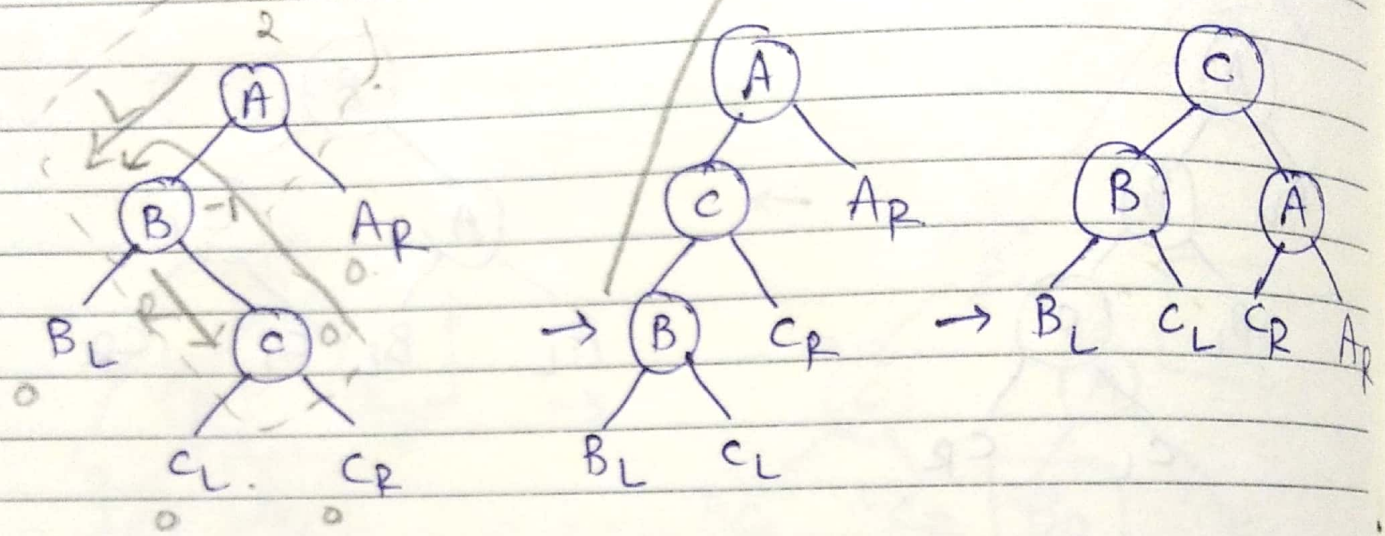


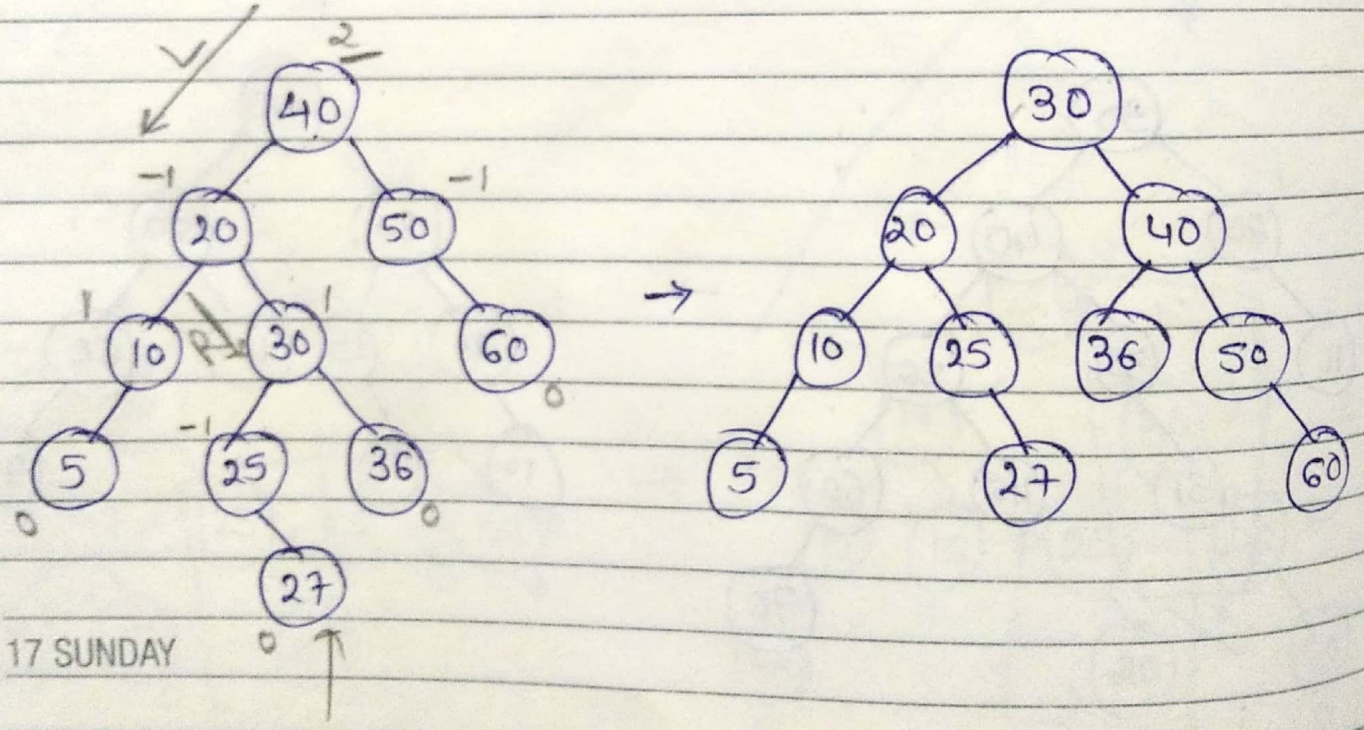
if node has negative balance factor => RR Rotation

2) RR Rotation :-



3) LR Rotation :



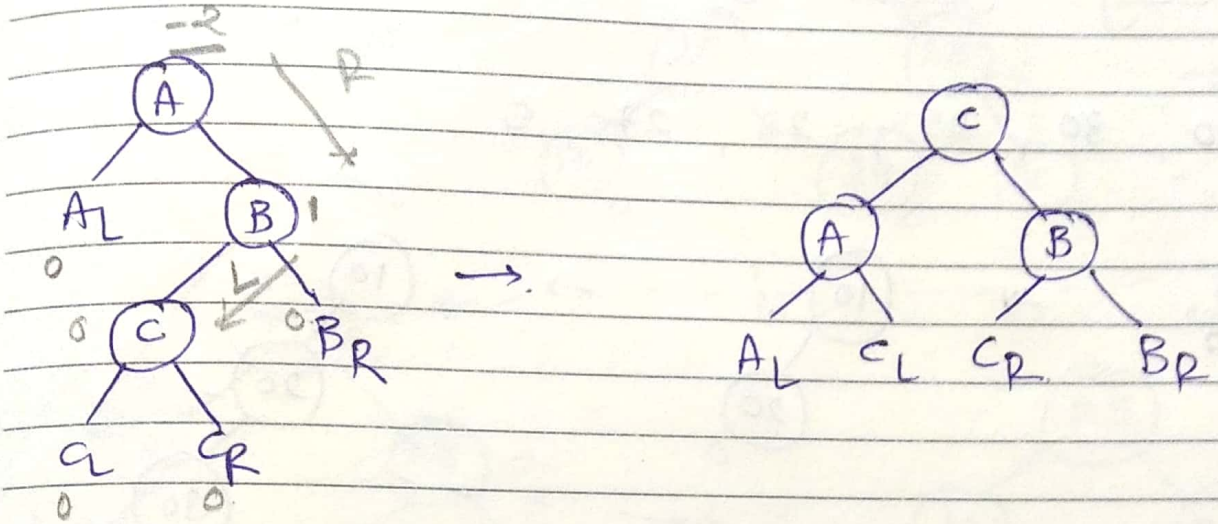


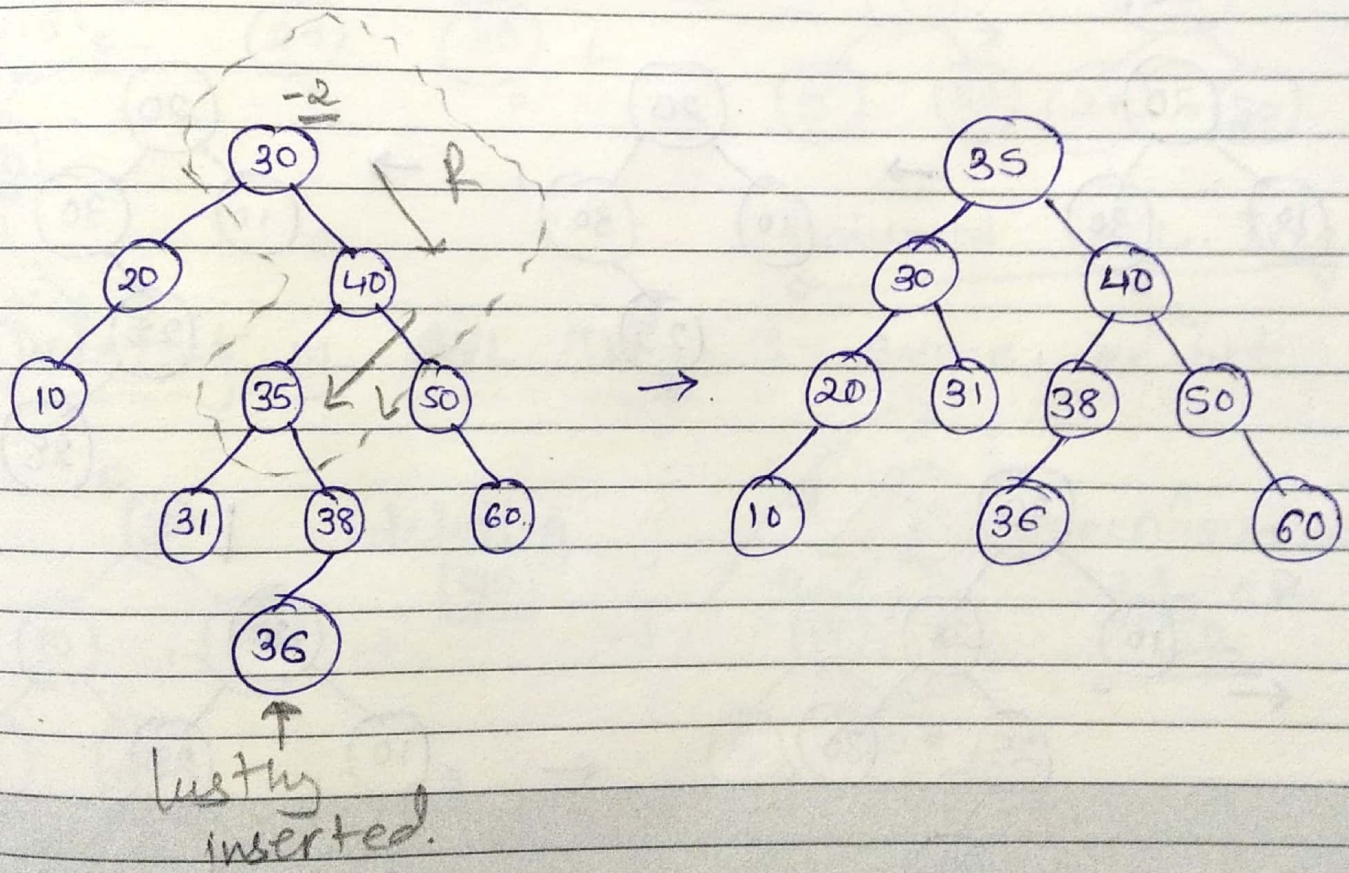
17 SUNDAY

lastly inserted.

Left → Right.

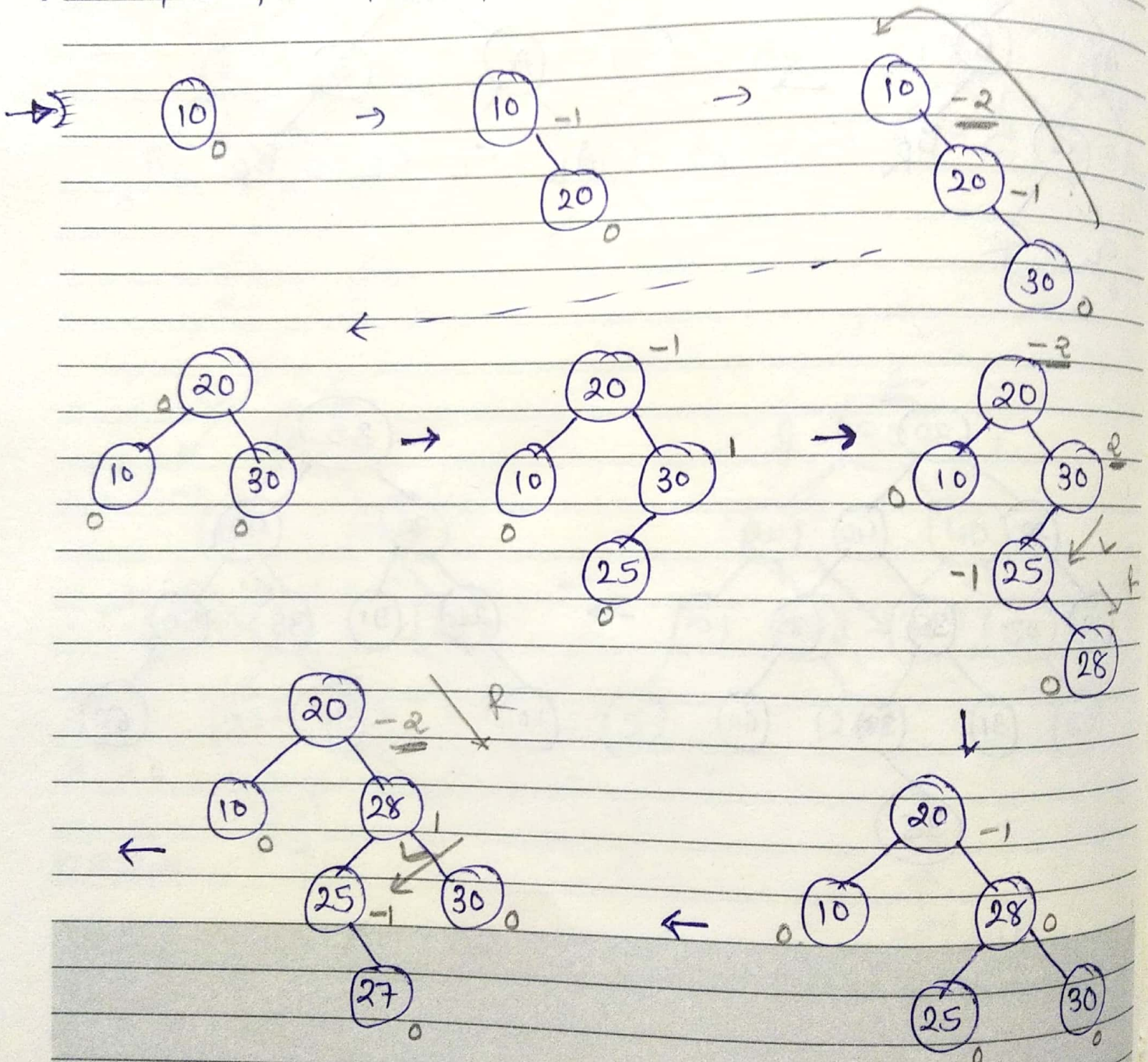
4) RL Rotation :

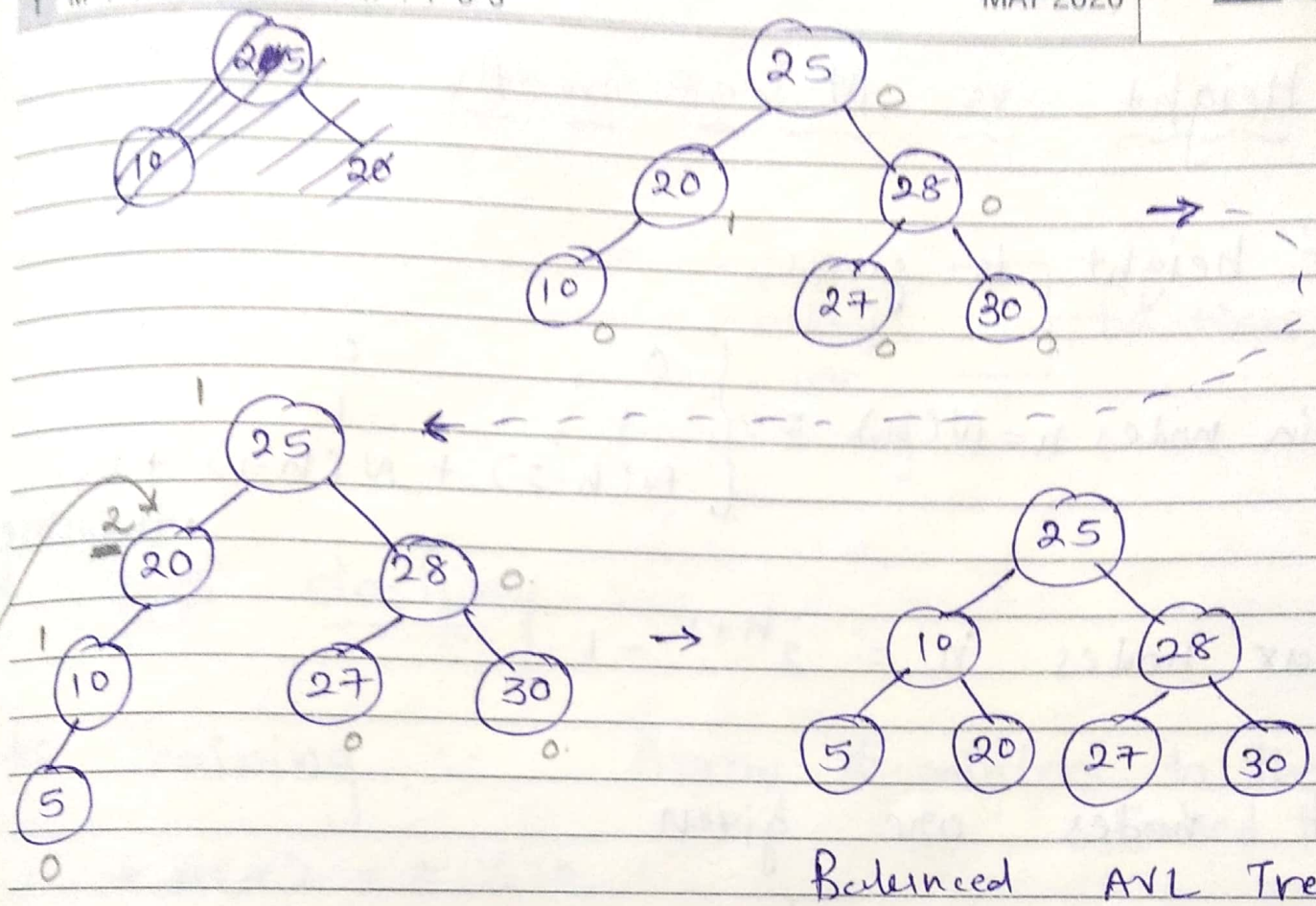




* Creating AVL tree :-

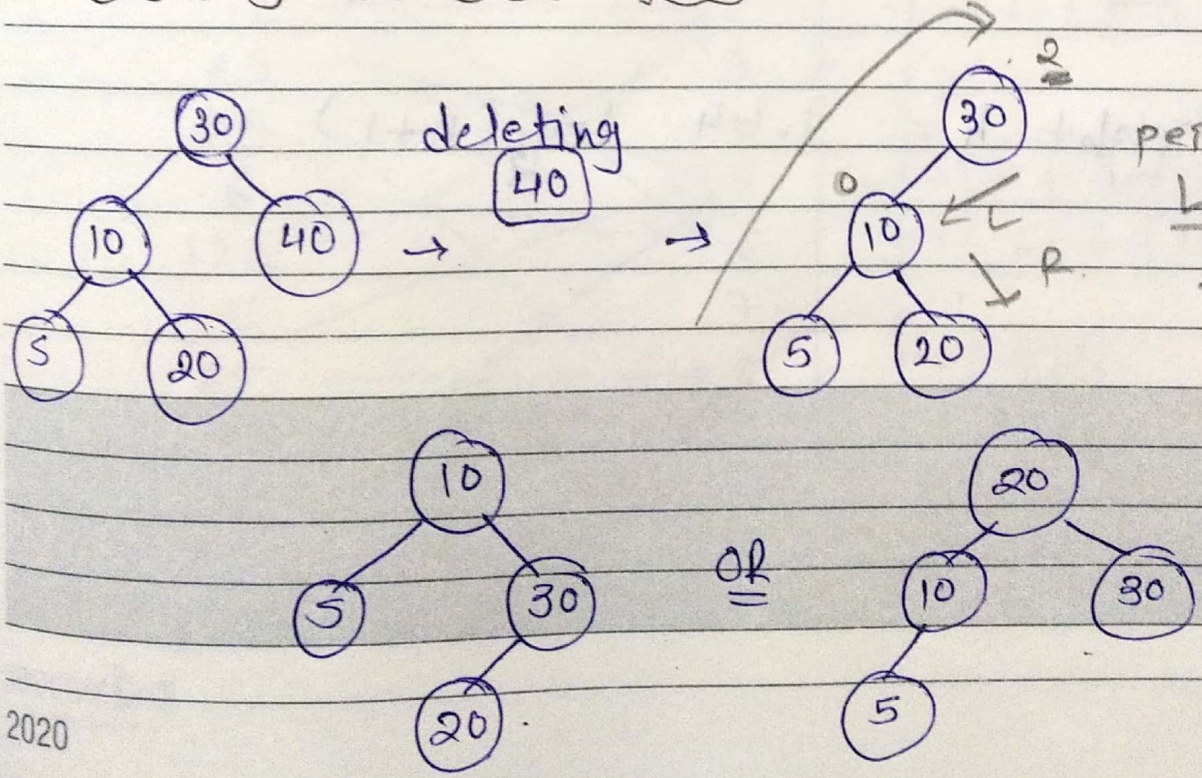
→ 10, 20, 30, 25, 28, 27, 5





Balanced AVL Tree.

* Deleting in AVL Tree :- same as in bst



* Height vs Nodes in AVL tree :-

if height is given.

$$\text{Min nodes } n = N(h) = \begin{cases} 0, & h = 0 \\ 1, & h = 1 \\ N(h-2) + N(h-1) + 1, & \text{otherwise} \end{cases}$$

$$\text{Max nodes } n = 2^{h+1} - 1$$

if nodes are given

$$\text{Min height } h = \log_2(n+1)$$

$$\text{Max height } h = 1.44 \log_2(n+1)$$

* Height vs Nodes in AVL tree :-

if height is given.

$$\text{Min nodes } n = N(h) = \begin{cases} 0, & h = 0 \\ 1, & h = 1 \\ N(h-2) + N(h-1) + 1, & \text{otherwise} \end{cases}$$

$$\text{Max nodes } n = 2^{h+1} - 1$$

if nodes are given

$$\text{Min height } h = \log_2(n+1)$$

$$\text{Max height } h = 1.44 \log_2(n+1)$$

* Height vs Nodes in AVL tree :-

if height is given.

$$\text{Min nodes } n = N(h) = \begin{cases} 0, & 0 \\ 1, & 1 \\ N(h-2) + N(h-1) + 1, & \text{otherwise} \end{cases}$$

$$\text{Max nodes } n = 2^{h+1} - 1$$

if nodes are given

$$\text{Min height } h = \log_2(n+1)$$

$$\text{Max height } h = 1.44 \log_2(n+1)$$

Special Thanks to Abdul Bari Sir