DS using python.

Time complexity of fibonacci series iterative programming.

a	b	n	print(a)	print(b)	i<n+1	c = a+b
a=0	b=1	n=5	print(a) = 0	print(b) = 1	i<n+1 [True]	c = 0+1 = 1
~~a=1~~					2<6 True	2
					3<6 True	3
					4<6 True	5
					5<6 True	8
					6<6 ✗	
1	1	1	1	1	n+1	n

a = b	b = c	print(c)	i++
a=b=1	b=c=1	print(c) = 1	i+1 = 0+1 = 1
1	2	2	2+1 = 3
2	3	3	3+1 = 4
3	5	5	4+1 = 5
5	8	8	5+1 = 6
n	n	n	n

$T(n) = 6n + 6$

will discard the constant.

$T(n) = n$

$T(n) \propto n$.

$O(n)$

Space complexity of fibonacci series iterative programming.

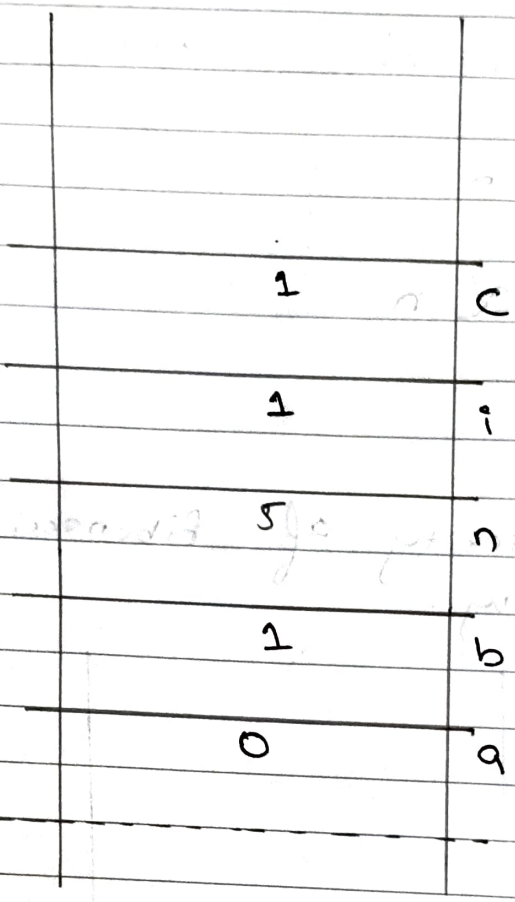
	c	n(1)
	i	n(1)
	n	n(1)
	b	n(1)
	a	n(1)

$n(1) + n(1) + n(1) + n(1) + n(1)$

$= O(n)$

$\therefore O(1)$

Space complexity of fib series



= O(1) + O(1) + O(1) + O(1) + O(1)

= 5 * O(1)

will discard the constent

= O(1)

Ds using Py 23-09-2024.

In linear search algorithm.

↳ The best case complexity is $O(1)$.

↳ The worst case complexity is $O(n)$.

$O(1)$ - Very first element is the target.

$O(n)$ - last element is the target.

↳ Average case.

$$\frac{\text{All possible cases time}}{\text{No. of cases}}$$

time complexity is = $O\left(\frac{n+1}{2}\right)$

$$= \frac{1+2+3+4+\dots n}{n}$$

$$= \frac{n\left(\frac{n+1}{2}\right)}{n}$$

$$= \frac{n+1}{2}$$

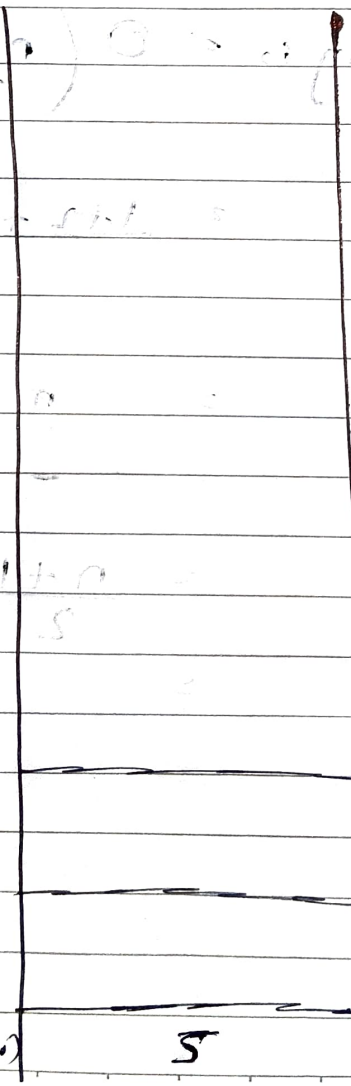
=

Linear search code:

```
def linear_search(nums, target):
    for i in range(len(nums) - 1):
        if nums[i] == target:
            return f"The element found at {i}"
        else: break.

    return element
```

Space complexity, stack frame.



3 x O(1)

= O(1)

DS using Python 30-Sept-2024

⇒ Bubble Sort

30
20
40
50
10

when we scan all the elements from the list, then its called as 1st pass.

In Bubble sort, we compare 2 element at a time to sort it in an ascending order.

1st pass

n = 5.

comparison in 1st pass = 4.

possible swapping in maximum = 4.

IInd pass.

comparison = 3.

possible swapping = 3.

IIIrd pass.

comparison = 2

possible swapping = 2

M T W T F S S

Page No.: YOUVA

Date:

IV pass

comparisons are 1.

possible swapping : 1.

30	20	20	20	10
20	30	30	10	20
40	40	10	30	30
50	10	40	40	40
10	50	50	50	50

why it is called bubble sort, because it is bubbling up the lower values.

comparisons are $n-1$.

$= 1+2+3+\ldots+n-1$

$= \frac{n.(n+1)}{2}$

$\frac{n^2+n^1}{2}$

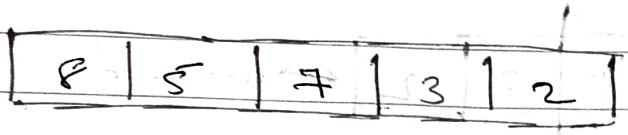
will always consider the higher degree

$= n^2$

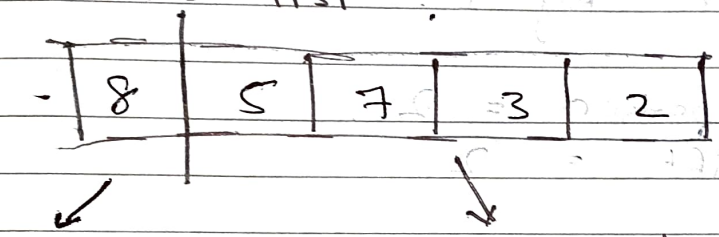
$O(n^2)$

DS using py 1-Oct-2024

Insertion sort.



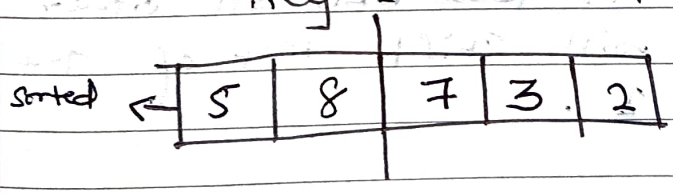
Initially, we take the very first element from the list, and consider it as a sorted list.



Now, will imply the insertion sort logic on our unsorted list to make it sorted.

I^st pass.: Insert 5 in sorted part.

Key = x = 5.

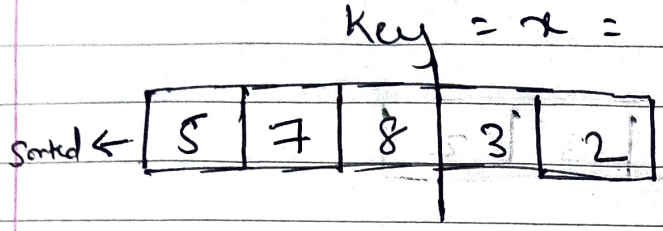


will insert 5 before 8 because, 8 is greater then 5.

comparison = 1.

shift = 1.

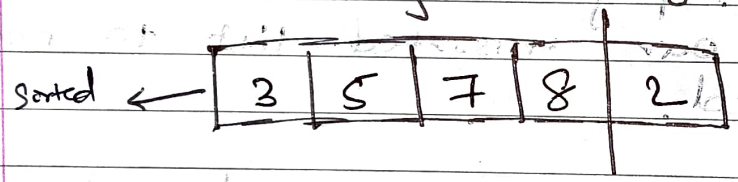
Pass II : Insert 7 in a sorted portion.



will insert 7 in between those 5 and 8 numbers because it comes in that order, and shift the 8 number by 1.

Comparison = 2
Shift = 2.

Pass III : Insert 3 in a sorted portion.

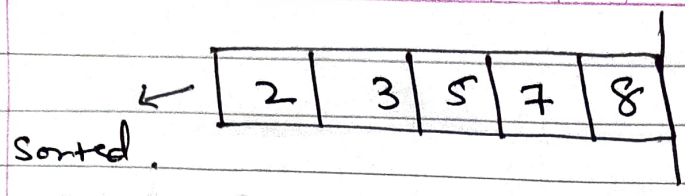


will insert 3 at the very first position after comparing it with all other elements because the number 5 is less than all other elements and will shift rest of the elements by 1.

Comparison = 3
Shift = 3.

Pass IV : Insert 2 in a sorted portion.

Key = x = 2.



Sorted.

Same like the previous pass, will insert at the very first position and shift of the elements by 1.

comparison = 4
Shift = 4.

Time complexity: len(arr) = 5.

comparisons = 1 + 2 + 3 + 4.
= 5 - 1 = 4.
= (n - 1)

= 1 + 2 + 3 + ... + n - 1.

$= \frac{n \cdot (n-1)}{2}$.

$= \frac{n^2 - n}{2}$.

will always consider the highes order and discard rest.

$T(n) \propto O(n^2)$

$O(n^2)$.

Code:

```
arr = 6, 8, 12, 15, 18, 20, 10

def Insertion_Sort(arr):
    for i in range(1, len(arr)):
        j = i - 1  # index 0.
        x = arr[i]

        while (j > -1 and arr[j] > x):
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = x

# Function calling
Insertion_Sort(arr = arr)
```

Dry run on the same list.

arr = [6, 8, 12, 15, 20, 2]

i = 1	i = (i-1)	x = arr[i]	j > -1 and arr[j] > x	while. arr[j+1] = arr[j]	j -= 1	arr[j+1] = x
1	0	$8^{[1]}$	False.	~~false~~ X arr[1] ≠ arr[0]	X	as it is.
2	1	$12^{[2]}$	False	False X	X	X
3	2	$15^{[3]}$	False.	X	X	X
4	3	$20^{[4]}$	False	X	X	X
5	4	$2^{[5]}$	True ^	arr[5] = 20	3	
				arr[4] = 15	2	
				arr[3] = 12	1	
				arr[2] = 8	0	
				arr[1] = 6	-1	arr[0] = 2
false	false.	fale.	fale.			

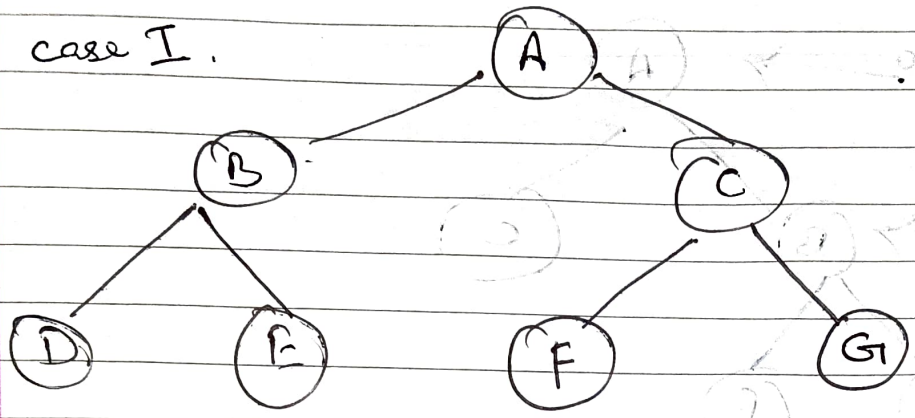
8	5	7	3	2
0	1	2	3	4

i	j = i-1	x = arr[i]	j >-1 && arr[j] > x	arr[j+1] = arr[j]	j -= 1	arr[j+1] = x	array
1	j = 0	x = 5	0 > -1 && 8 > 5 (True)	arr[1] = 8	j = -1		
			-1 > -1 (False)	———	———	arr[0] = 5	5, 8, 7, 3, 2
2	j = 1	x = 7	1 > -1 && 8 > 7 (True)	arr[2] = 8	j = -1		
			0 > -1 && 5 > 7 (false)	———	———	arr[1] = 7	5, 7, 8, 3, 2
3	j = 2	x = 3	2 > -1 && 8 > 3 (True)	arr[3] = 8	j = -1		
			1 > -1 && 7 > 3 (True)	arr[2] = 7	j = -1		
			0 > -1 && 5 > 3 (True)	arr[1] = 5	j = -1		
			-1 > -1 (false)	———	———	arr[0] = 3	3, 5, 7, 8, 2
4	j = 3	x = 2					

level by level.

Heap sort.

case I.



List from above Tree. =

1	2	3	4	5	6	7
A	B	C	D	E	F	G
0	1	2	3	4	5	6.

1 → It should be in the list.

2 → some Relations.

(1) left child.

formula to identify the left child of a node.

Formula:

$= 2 * i$

Suppose i want to see the left child of 1 & 3.

$= 2 * 1$

$X = 2$. [0] is the left child.

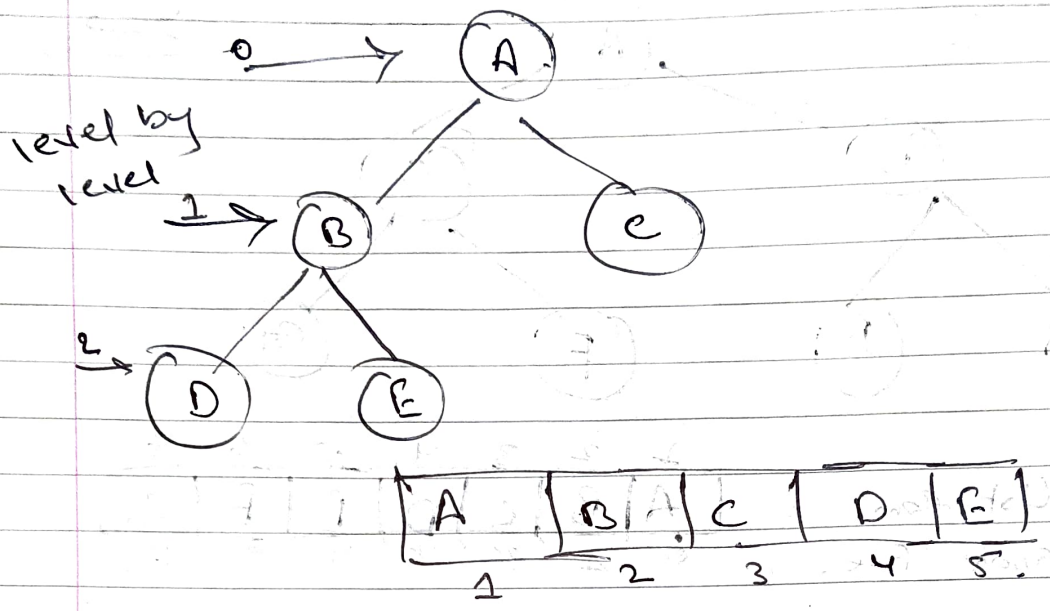
(2) Right child. formula

$2 * i + 1$.

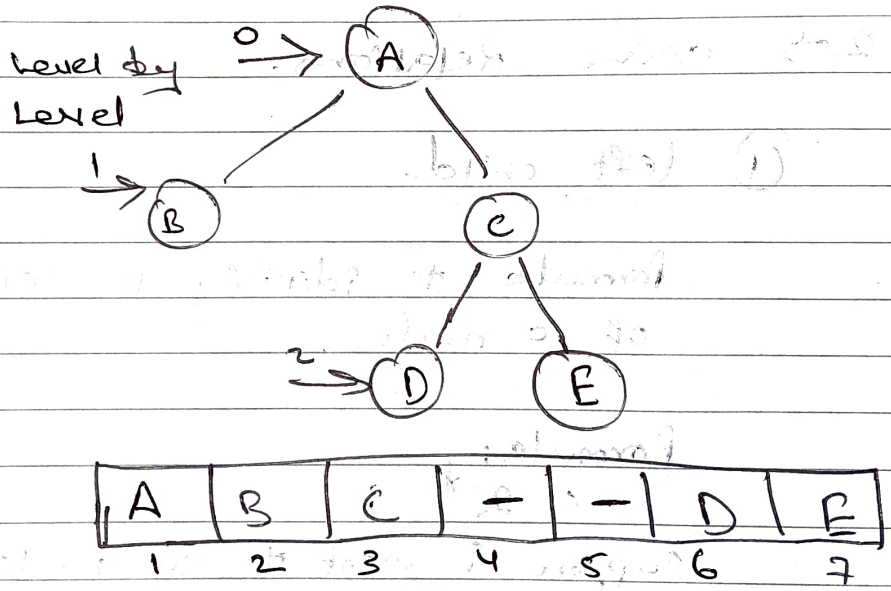
(3) Parent node formula. $\left\lfloor \frac{i}{2} \right\rfloor$ ← floor value.

03-Oct-2024

Case I :



Case II :



In incomplete Tree, we have to leave the left and right node as empty node, where we create a list from tree.

If there the element is missing, then we must leave the blank spaces in the array.

To find the max. numbers of nodes in a tree with hieght.

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

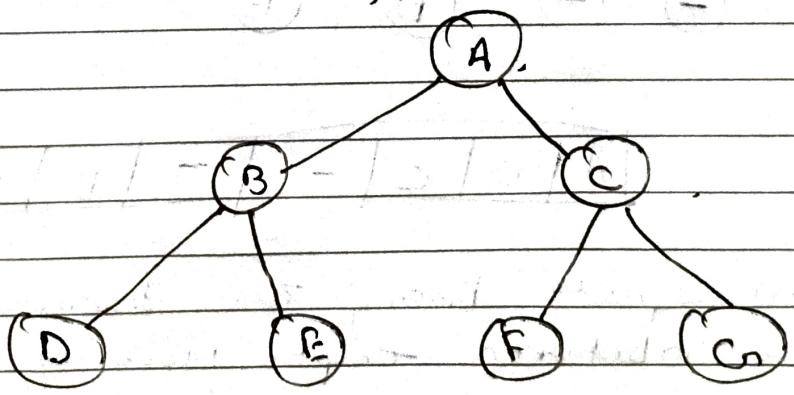
where,
h is the height, starting from 0.

for 1st case.

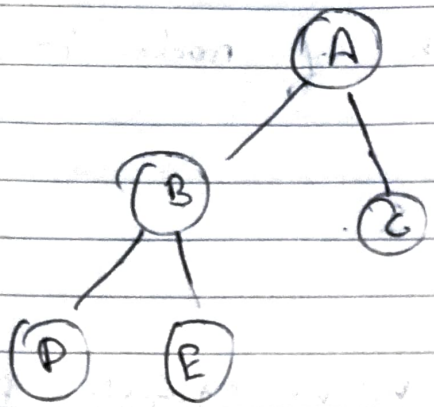
$$= 2^{2+1} - 1$$
$$= 2^3 - 1.$$
$$= 8 - 1$$
$$= 7.$$

The binary tree with maximum number of nodes is called, full binary tree like case I.

complete binary Tree →

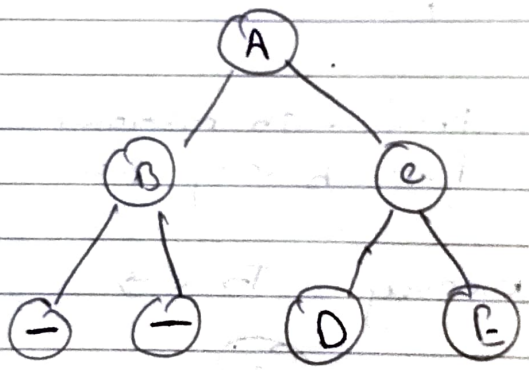


A	B	C	D	E	F	G



A	B	C	D	E

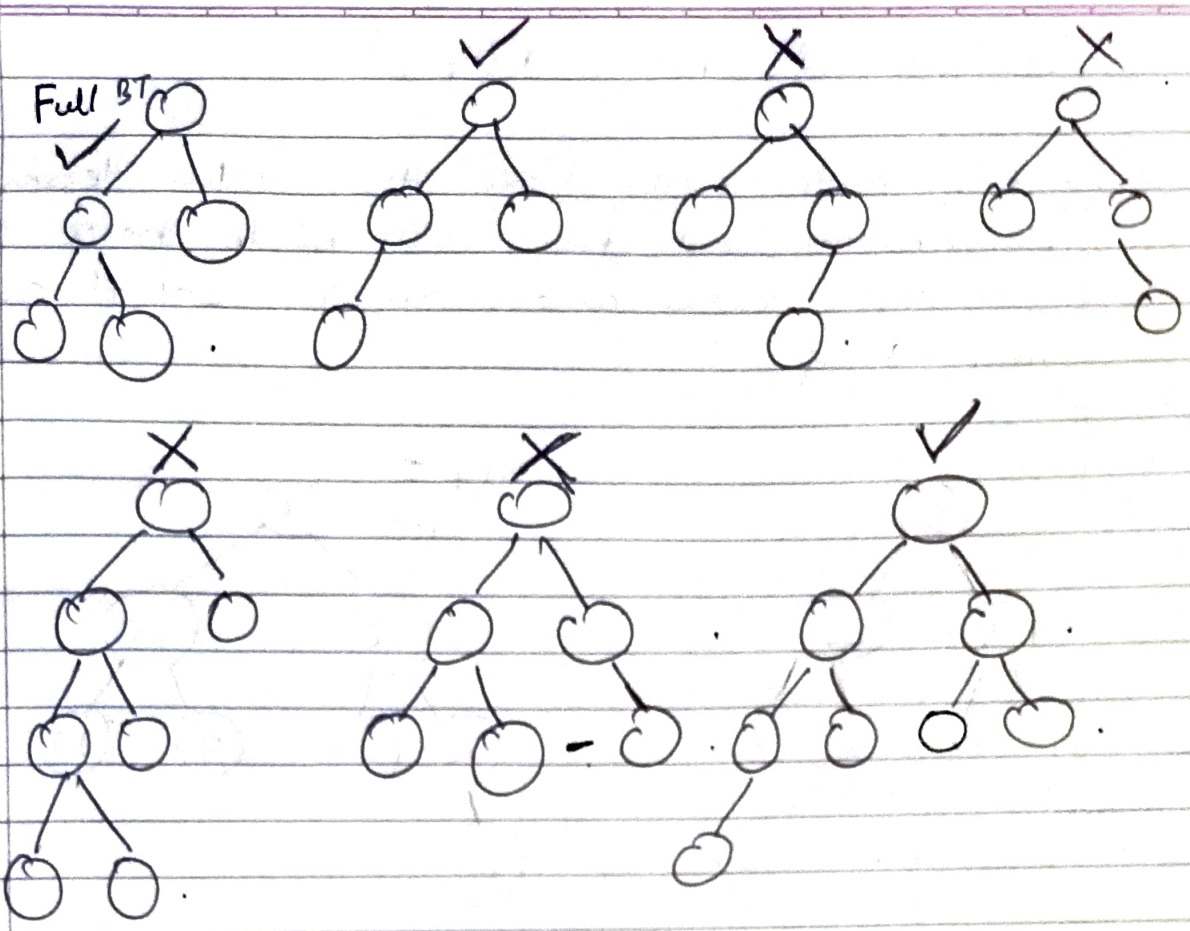
1) complete because.
2) H-1 level is complete.



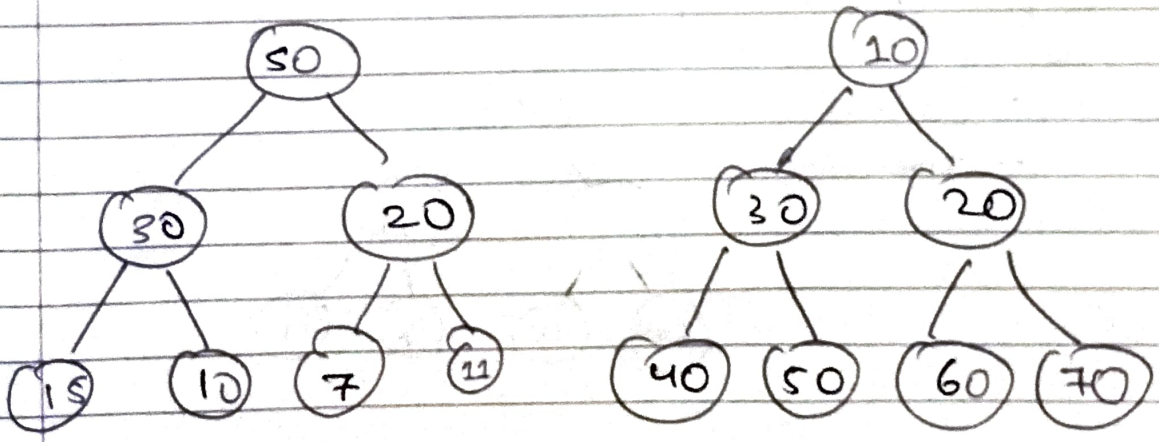
A	B	C	-	-	D	E

Incomplete binary tree because, there are in between empty block of nodes are there.

⇒ complete binary tree is the full binary tree upto the level h-1, provided the last elements are filled from left to right, not from right to left.



Heap



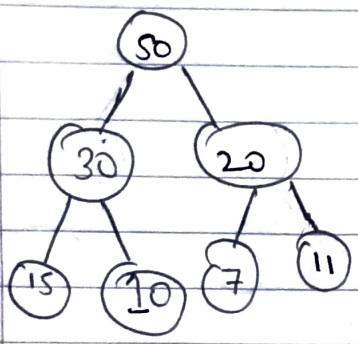
Max Heap

If Heap, Every node will have value greater than or equal to its descendent

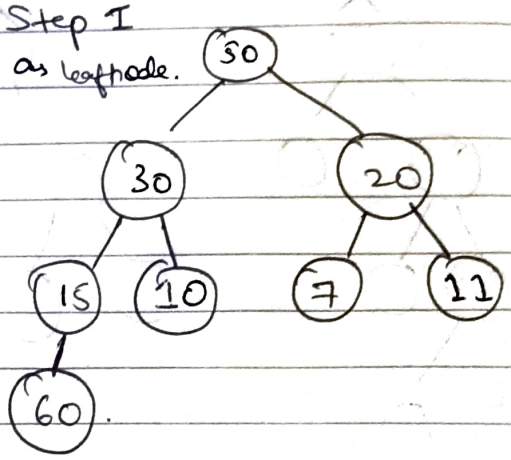
where a root node is having the longest value

its called max heap.

⇒ Every node will have the value lesser than descendents, and the root node have the smallest value among all is called as min heap.



Step I
as leaf node.



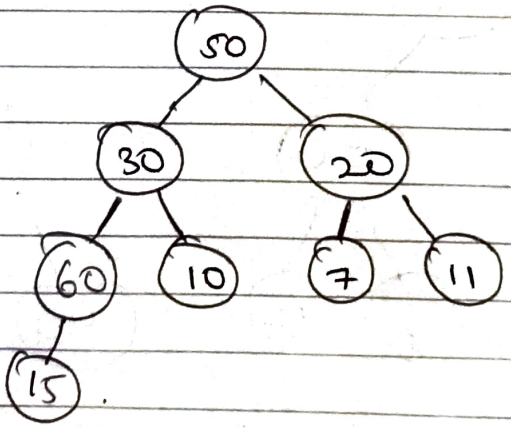
Insert a new node
60.

Step II

Compare 60 with its parent node

∵ 60 > 15

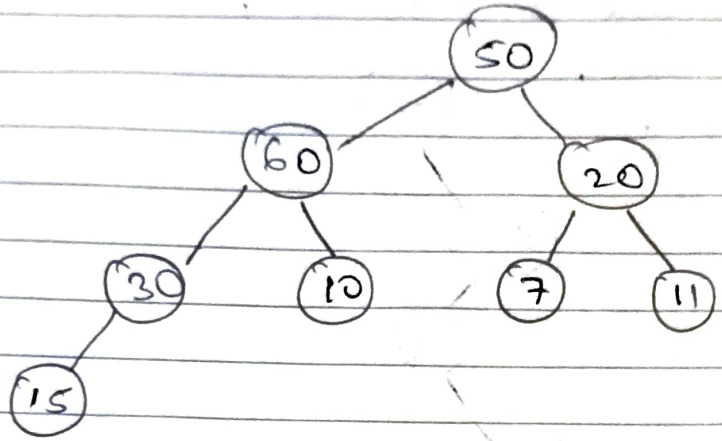
∴ Interchange



Step III → compare 60 with its parent node 30.

∵ 60 > 30

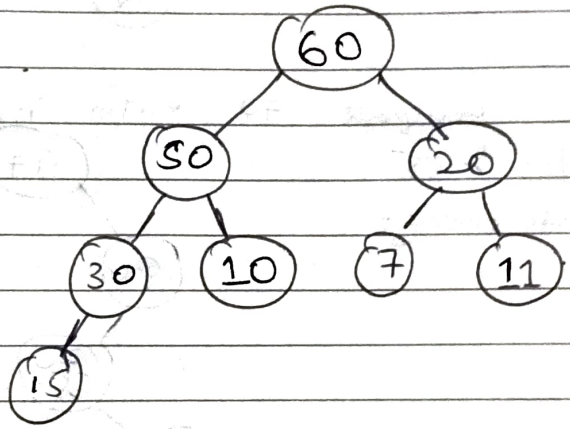
∴ Interchange.



Step IV compare 60 with its parent node 50

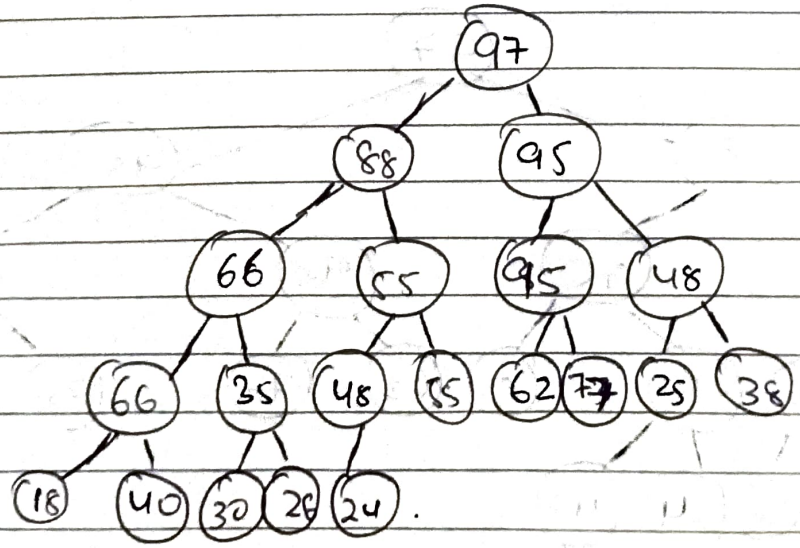
∵ 60 > 50.

∴ interchange.



Q. Tree = [97, 88, 95, 66, 55, 95, 48, 66, 35, 48, 55, 62, 77, 25, 38, 18, 40, 30, 26, 24]

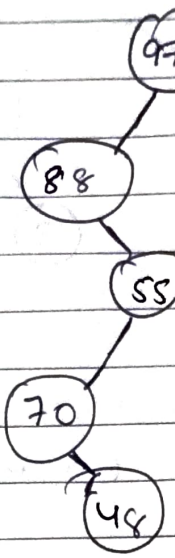
Insert a new element 70.



Step I : compare 70 with its parent

∵ 70 > 48.

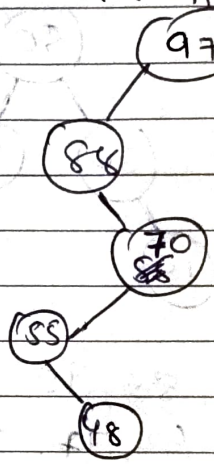
∴ interchange.



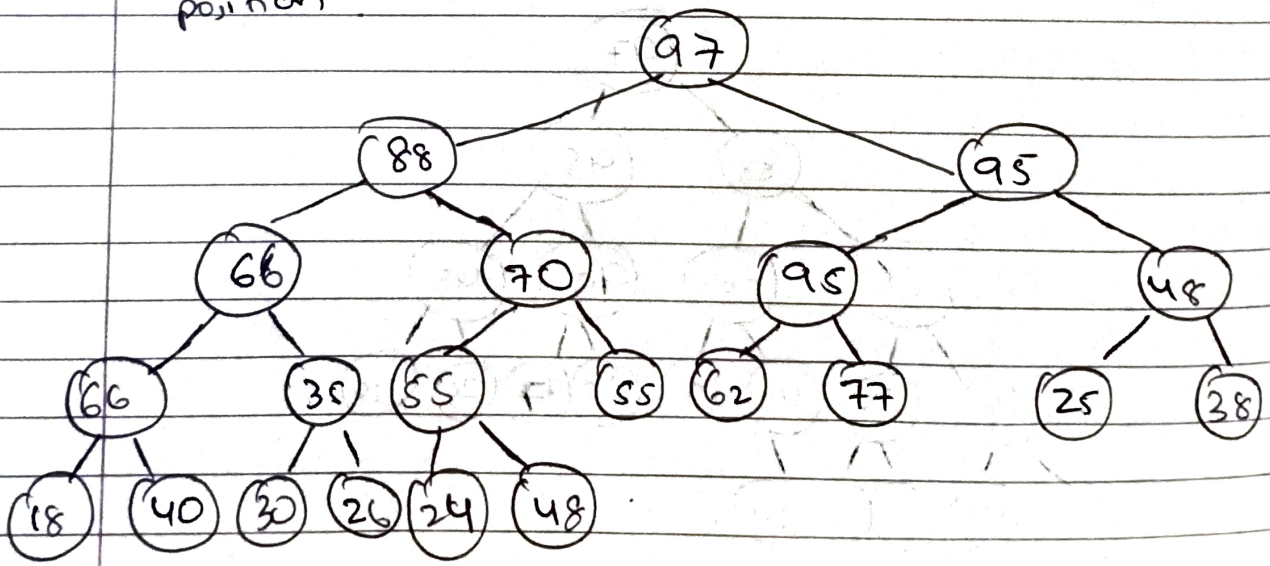
Step II: compare 70 with its parent

∵ 70 > 55

∴ interchange.



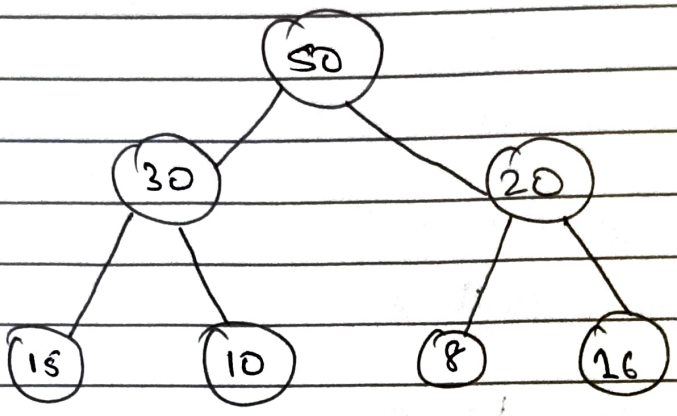
Now, 70 has been placed at the right position.



DS using python

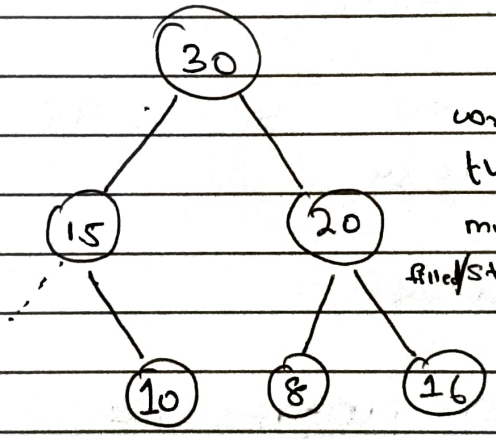
07-Oct-2024

Heap sort - Division of nodes.



Division of nodes.

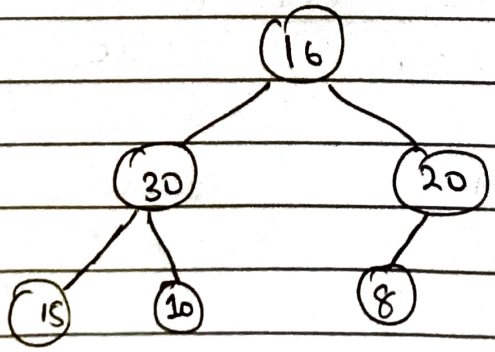
50	30	20	15	10	8	16



wrong procedure, coz the comple binary tree must have leaf node filled/start from left to right.

Step I:

correct procedure is to place last node to the very first root node, and root node will be taken out.



step II : compare siblings of root 16.

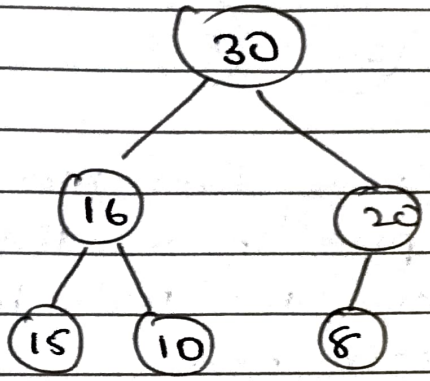
∵ 30 > 20.

∴ we will compare parent 16 with 30.

step III : compare 16 with 30.

∵ 30 > 16

∴ Interchange.



step IV : compare sibling of node 16.

∵ 15 > 10.

∴ keep as it is.

step V : compare 15 with parent 16.

∵ 16 > 15

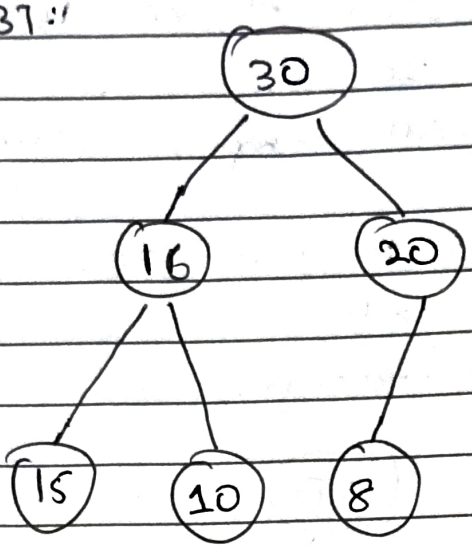
∴ keep as it is.

Hence, the BT is max heap.

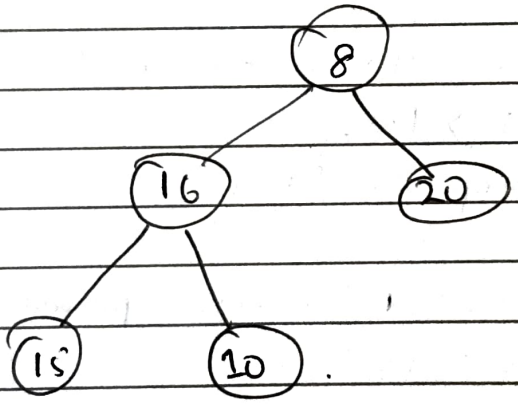
Current BT :

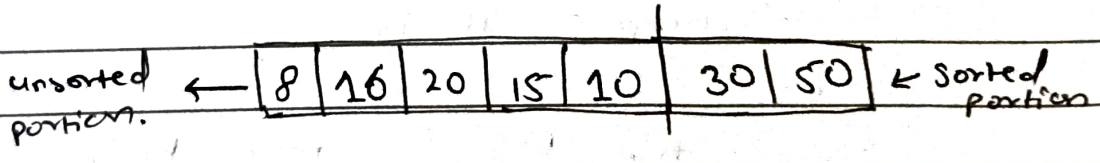
16	30	20	15	10	8	50

current BT:



Step VI : Take the very first root node from the list and place it in the list before 50, and replace root node in the Tree with last node.





Step VII : compare siblings of root node 8.

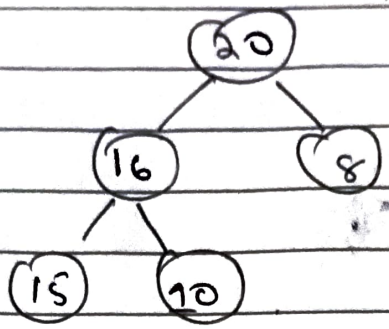
∵ 16 < 20.

∵ 20 > 16

∴ compare 8 with 20.

∵ 20 > 8.

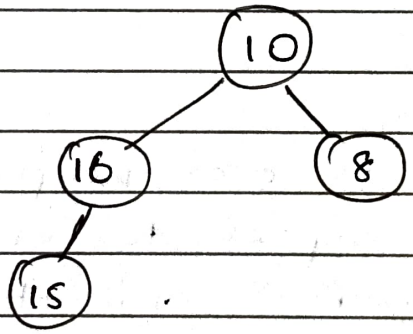
∴ Interchange.



Step VIII : Now it is max heap, then again we remove very first node from the tree and place it in a list before previously inserted node and replace root with last node.

current BT : | 20 | 16 | 8 | 15 | 10 | 30 | 50 |

After removing first element/node : | 10 | 16 | 8 | 15 | 20 | 30 | 50 |



Step IX : compare siblings of parent 10.

∵ 16 > 8.

∴ compare parent 10 with child 16.

∵ 16 > 10.

∴ Interchange