Complete Binary Tree (CBT)

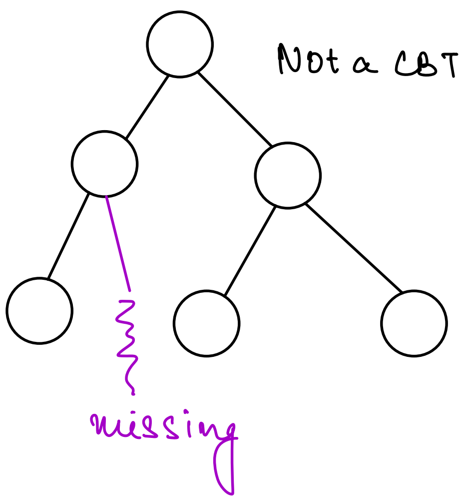
A Binary Tree is said to be a CBT if it satisfies below properties.

i) All the nodes have to be filled level by level from left to right.

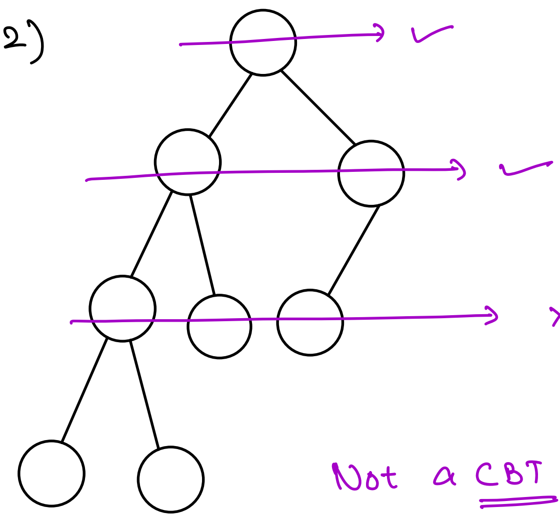
ii) All the levels should be completely filled except the last level.

Ex:-

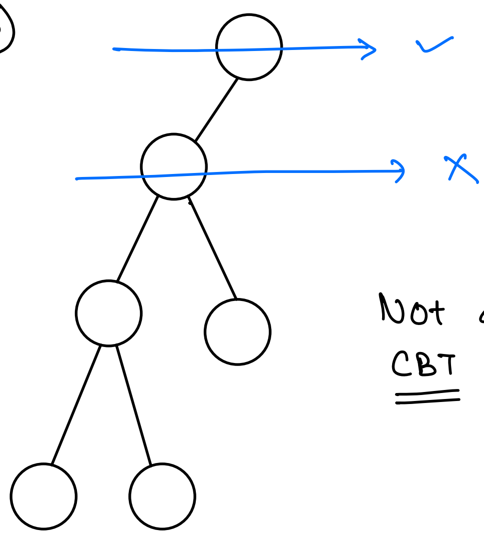
1)



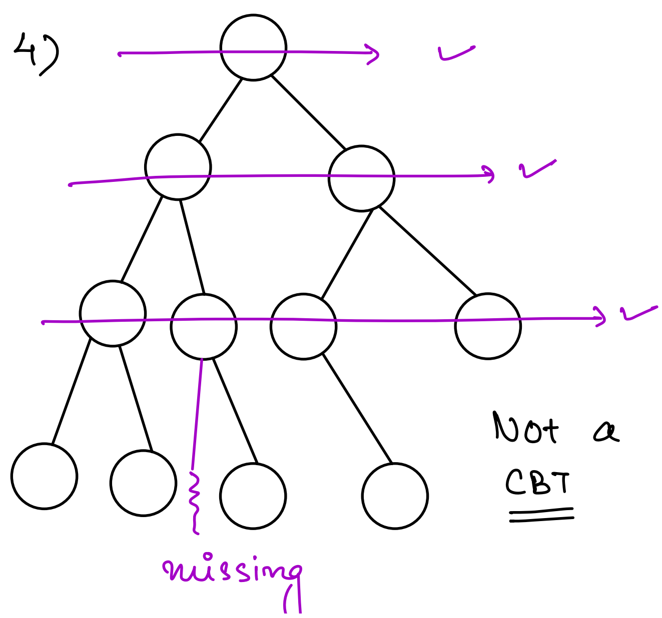
2)



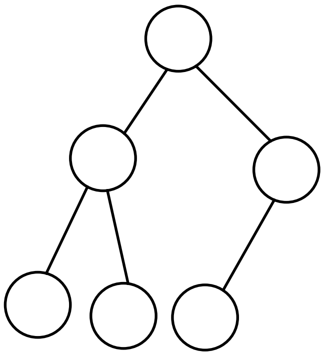
3)



4)

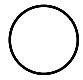


5)



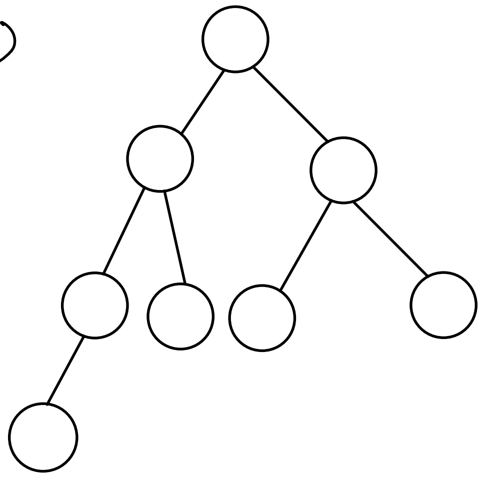
CBT ✓

6)



CBT ✓

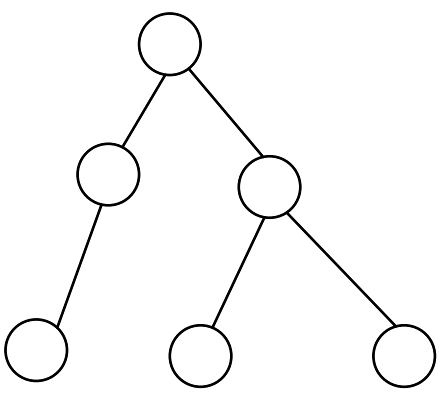
7)



✓

Note :- All Balanced Binary Tree's are CBT ? **NO**

$\forall$ nodes $|h(LST) - h(RST)| <= 1$



Balanced : ✓

CBT ✗

<u>Note</u> All CBT's are Balanced Binary Tree's ?

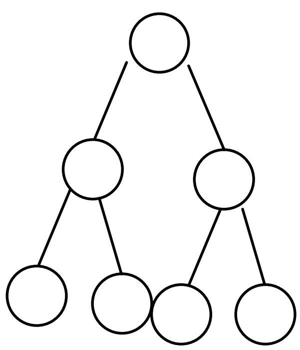
<u>YES</u>

→ Every CBT will have height difference of max 1.

<u>Quiz</u> If there are N nodes in a CBT, <u>height</u>?

Height(Balanced B.T) = Height (CBT) = $\log_2 N$

Height (CBT)	min Nodes	max Nodes
1	$2 = 2^1$	$3 = 2^2 - 1$
2	$4 = 2^2$	$7 = 2^3 - 1$
3	$8 = 2^3$	$15 = 2^4 - 1$
4	$16 = 2^4$	$31 = 2^5 - 1$
⋮	⋮	⋮
H	2^H	$2^{H+1} - 1$



H → 2^H (Min Nodes)

H → $2^{H+1} - 1$ (Max Nodes)

$$2^H = N \Rightarrow H = \log_2 N$$

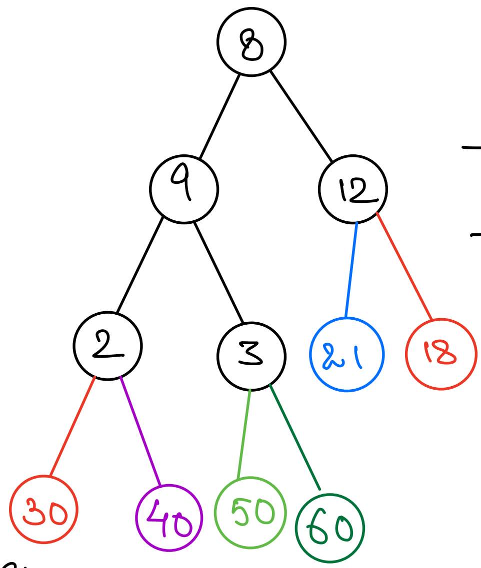
$$2^{H+1} - 1 = N \Rightarrow 2^{H+1} = N + 1$$

$$H + 1 = \log(N+1)$$

$$H = \log(N+1) - 1 \simeq O(\log N)$$

Implementation of CBT :-

1) Binary Tree
(CBT)



insert : 21, 18, 30, 40, 50, 60

~~8~~, ~~9~~, ~~12~~, ~~2~~, ~~3~~, 21, 18, 30, 40, 50, 60

Steps :-

⇒ Level Order traversal.

⇒ Whenever a new node is created, insert it in a queue & delete the front of the queue only if it's both left & right children are filled.

TC : O(N) for inserting N nodes.

SC : $\boxed{O(N)}$ ↳ Queue

Disadvantages :-

1) SC : O(N)

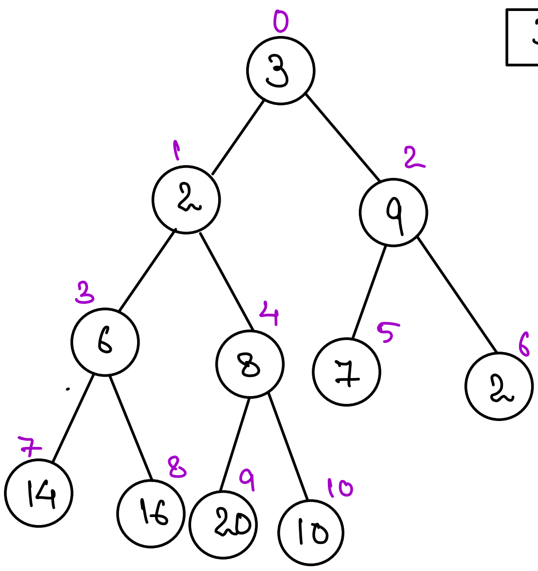
2) Iterating from child to parent is NOT allowed.

2) <u>Arrays</u>

: 3, 2, 9, 6, 8, 7, 2, 14, 16

→ List<int> | Vector<int>

0	1	2	3	4	5	6	7	8	9	10
3	2	9	6	8	7	2	14	16	20	10



Parent	left	right
0	1	2
1	3	4
2	5	6
3	7	8
4	9	10

Parent index = i → $2i+1$ (left), $2i+2$ (Right)

index i ⇒ Parent index = $\frac{i-1}{2}$

TC of inserting N elements : $O(N)$

SC : $O(1)$

Heap DS :-

→ it's a CBT

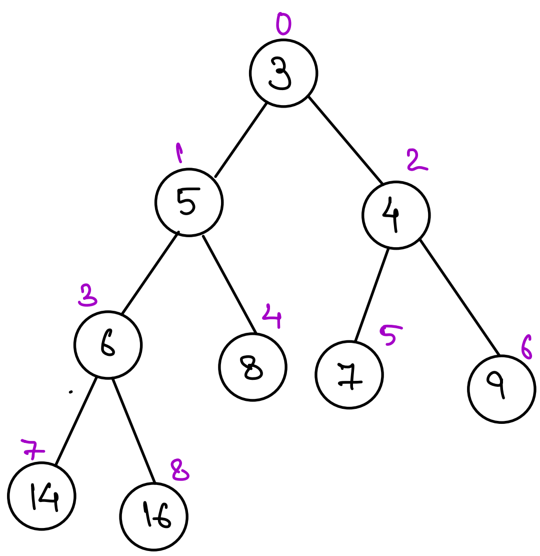
→ for every node ≥ Both the children

→ Max Heap

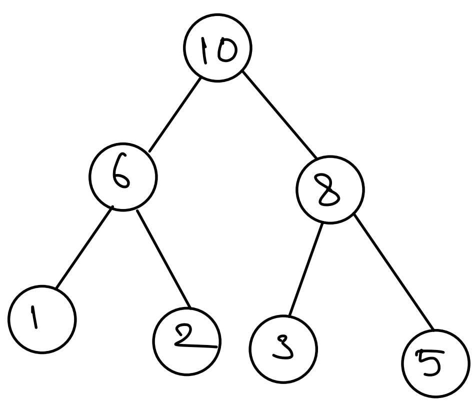
OR

for every node ≤ Both the children

→ Min Heap.



min Heap



Max Heap.

Min Heap

Insert : O(log N)
getMin() : O(1)
Search : O(N)
deleteMin() : O(log N)

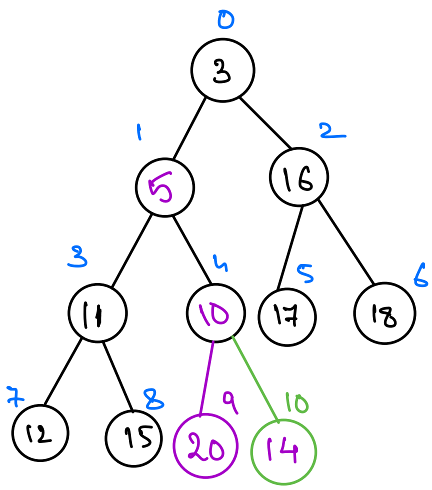
Max Heap

Insert : O(log N)
getMax : O(1)
Search : O(N)
deleteMax() : O(log N)

Insert in Min heap.

List

0	1	2	3	4	5	6	7	8	9	10
3	10	16	11	14	17	18	12	15	20	5

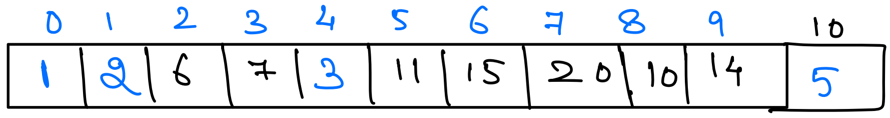


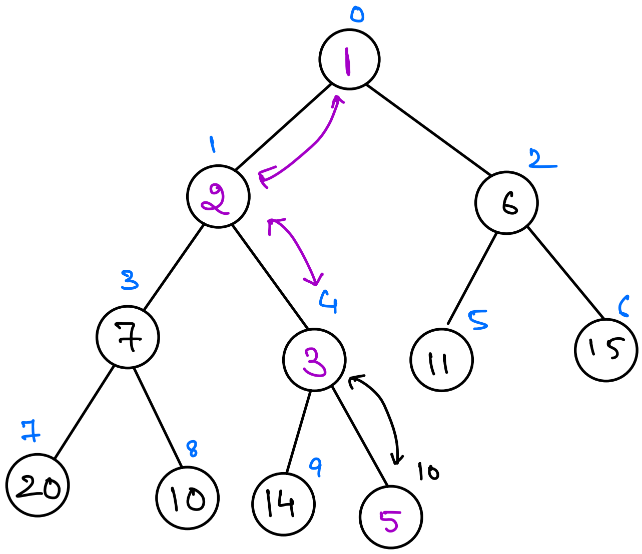
min heap ✓

insert 20

index	Parent	if(A[ind] < A[Parent])
10	4	⇒ Swap.
4	1	⇒ Swap.

TC : O(log N)





index	parent	if (A[P] > A[i])
10	4	swap
4	1	swap
1	0	swap
0	0	

Stop | Break

TC : O(log N) [Min | Max Heap]

```
void insert( List<int> arr, int ele) {
    arr.add(ele)
    index = arr.size() - 1;
    parent = (index - 1) / 2;
    while (index != 0 && arr[parent] > arr[index]) {
        swap(arr[parent], arr[index]);
        index = parent
        parent = (index - 1) / 2;
    }
}
```

getMin() in Min Heap / getMax() in Max Heap ⇒ $\underline{\underline{O(1)}}$

return list[0];

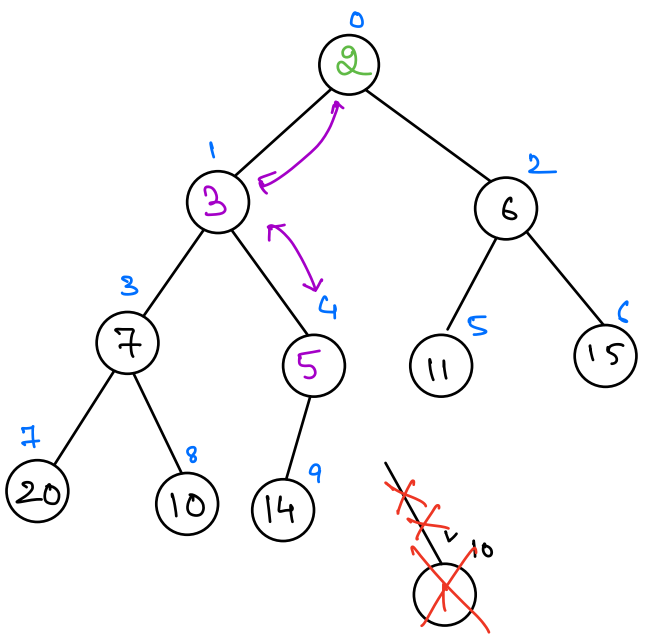
Search Operation :-

↳ Linear search on the list of elements.

TC : O(N) {Both min / max heap}

Delete min operation (MIN HEAP)

0	1	2	3	4	5	6	7	8	9	10
~~5~~ 2	~~2~~ ~~5~~ 3	6	7	~~3~~ 5	11	15	20	10	14	1



Steps :-

i) swap (A[0], A[N-1])

ii) Delete last element.

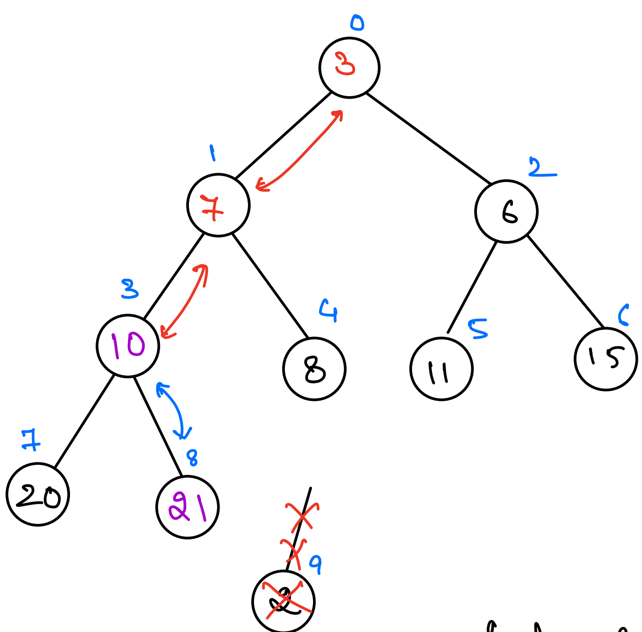
iii) Propogate down.

index	left	right	min_index (if A[index] > A[min_index])
0	1	2	1 : A[1] > A[0] ✓ swap.
1	3	4	4 : A[1] > A[4] ✓ swap.
4	9	×	9 : A[4] > A[9] ×

Break.

0	1	2	3	4	5	6	7	8	9
3	7	6	10	8	11	15	20	21	2

#



Delete min

i) swap(A[0], A[9])

ii) Delete A[9]

ind	l	r	min_ind	
0	1	2	1	swap
1	3	4	3	swap
3	7	8	8	swap
8	? 17 ×	? 18 ×		⇒ **Break.**

TC: O(log N)

deleteMin()
deleteMax() } ⇒ O(log N)

Delete any random element :-

i) Search : O(N)

ii) swap with last index : O(1)

iii) Delete the last index : O(1)

iii) Propogate Down : O(log N)

TC: O(N)

Heap :-

	Heap	BBST
1) Insert	O(log N)	O(log N)
2) getMin() | getMax	O(1)	O(log N)
3) deleteMin() | deleteMax()	O(log N)	O(log N)
4) Search	O(N)	O(log N)
5) Delete any random ele.	O(N)	O(log N)

If below ③ operations are frequent the go with Heap DS.

1) Insert
2) getMin() | getMax
3) deleteMin() | deleteMax()

Inbuilt library

i) C++ : priority_queue.
ii) Java : PriorityQueue< — >
iii) Python : heapq

⋮ ⋮

—— * ——