Todays Content:

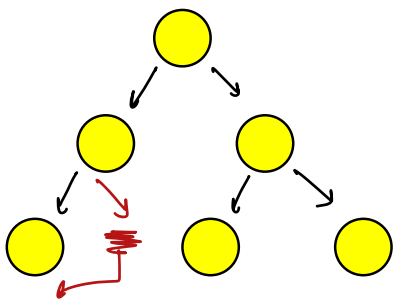
- Complete Binary Tree
- Heaps Intro
- Implementation:
    a) Insert
    b) Delete_min() / Delete_max()
    c) Get_min() / Get_max()

Complete Binary Tree (CBT) → pre-requisits

A BT is said to be CBT if

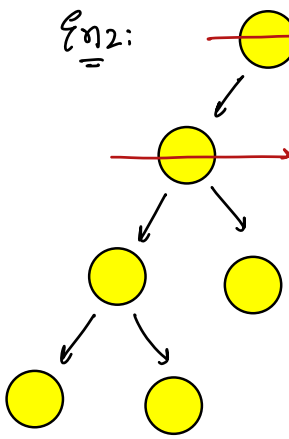
1) All Nodes have to be filled level by level from left → right
2) At all levels it should be completely filled except last level { we can either fill it or Not }

Eg1:



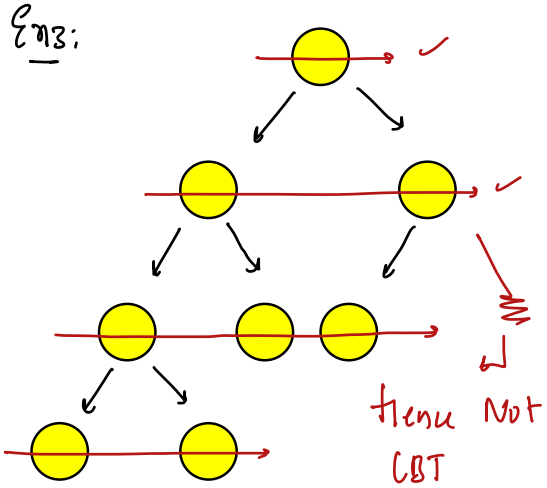
Hence Not CBT

Eg2:



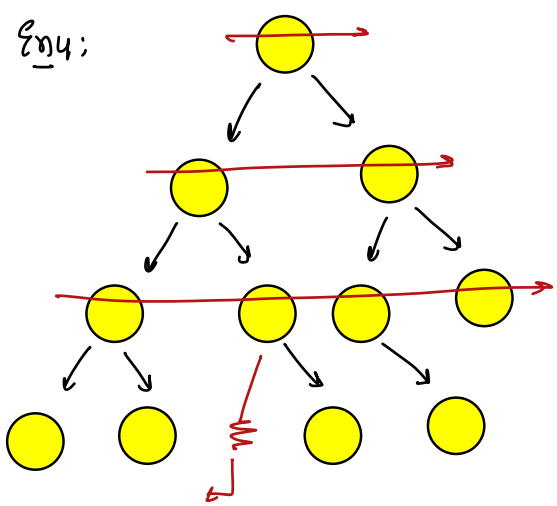
∴ Hence Not CBT

Eg3:



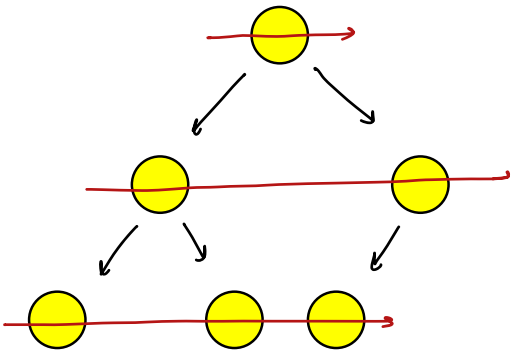
Hence Not CBT

Eg4:

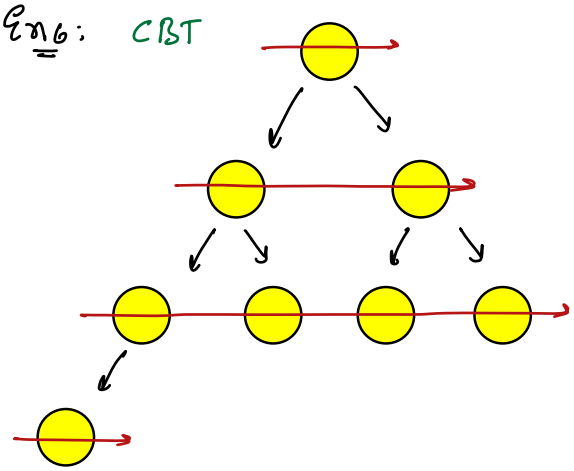


Hence Not CBT

Eg 5: CBT

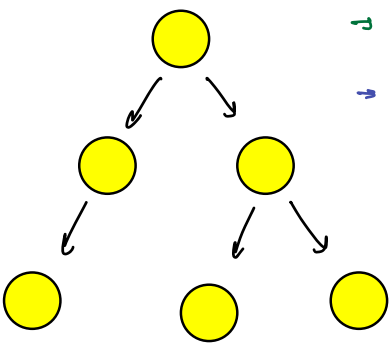


Eg 6: CBT



Are all Balanced Binary Trees are they CBT's = No

Eg:



→ BBT ⇒ log N

→ CBT * ∉ NOT CBT

Are all CBT's are Balanced Binary Trees = YES

for every node | height(LST) − height(RST) | ≤ 1

Height (CBT) $= \log_2 N$

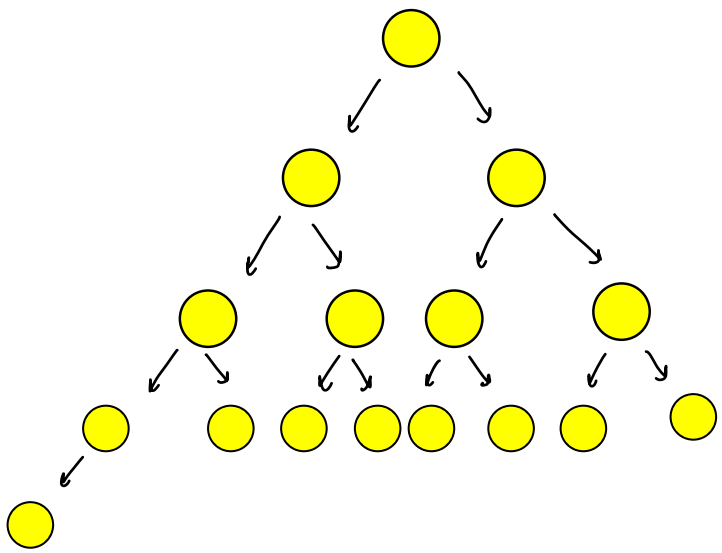
Height of CBT	Min Nodes	Max Nodes
1	$2 : 2^1$	$3 : 2^2 - 1$
2	$4 : 2^2$	$7 : 2^3 - 1$
3	$8 : 2^3$	$15 : 2^4 - 1$
4	$16 : 2^4$	$31 : 2^5 - 1$
h	2^h	$2^{h+1} - 1$

Height Node

h → $2^h = N$, $h = \log_2 N$

h → $2^{h+1} - 1 = N$, $h = (\log_2 N)$

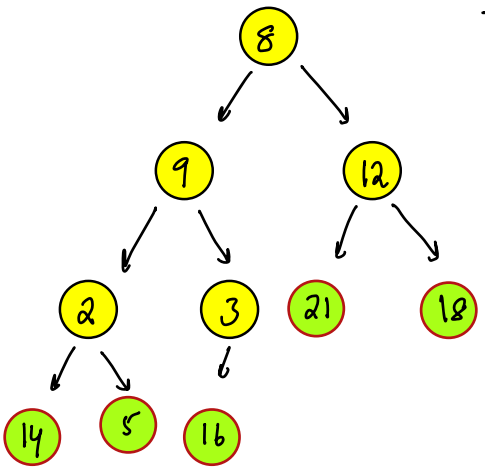
Height of N Node CBT ≈ $O(\log_2 N)$



→ Significant:

→ $\log_2 N$ Height of CBT → Root → leaf ∩ leaf → Root : $(\log_2 N)$

Implementation of CBT using nodes



→ Insert: 21 18 14 5 16

To new node

→ appr:

Inserting in a CBT is by using level order traversal

→ Idea:

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

TC: To Insert N Nodes → O(N)

SC: O(N)

→ using above implementation

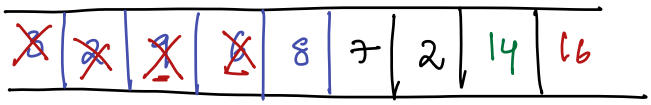
1 Inserting taking Extra Space

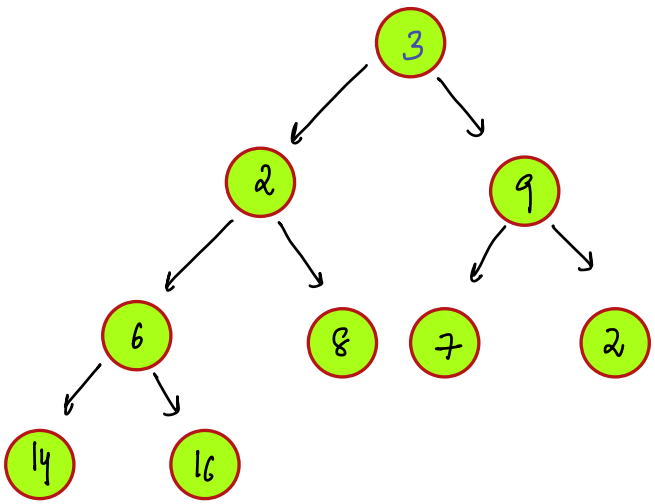
2 Traveling Child → Root Not possible

Dry run for above idea:

Insert: 3 2 9 6 8 7 2 14 16

Queue:





Implementation of CBT using arrays

- Insert : N Element, No Entrapau
- top → down & down → top

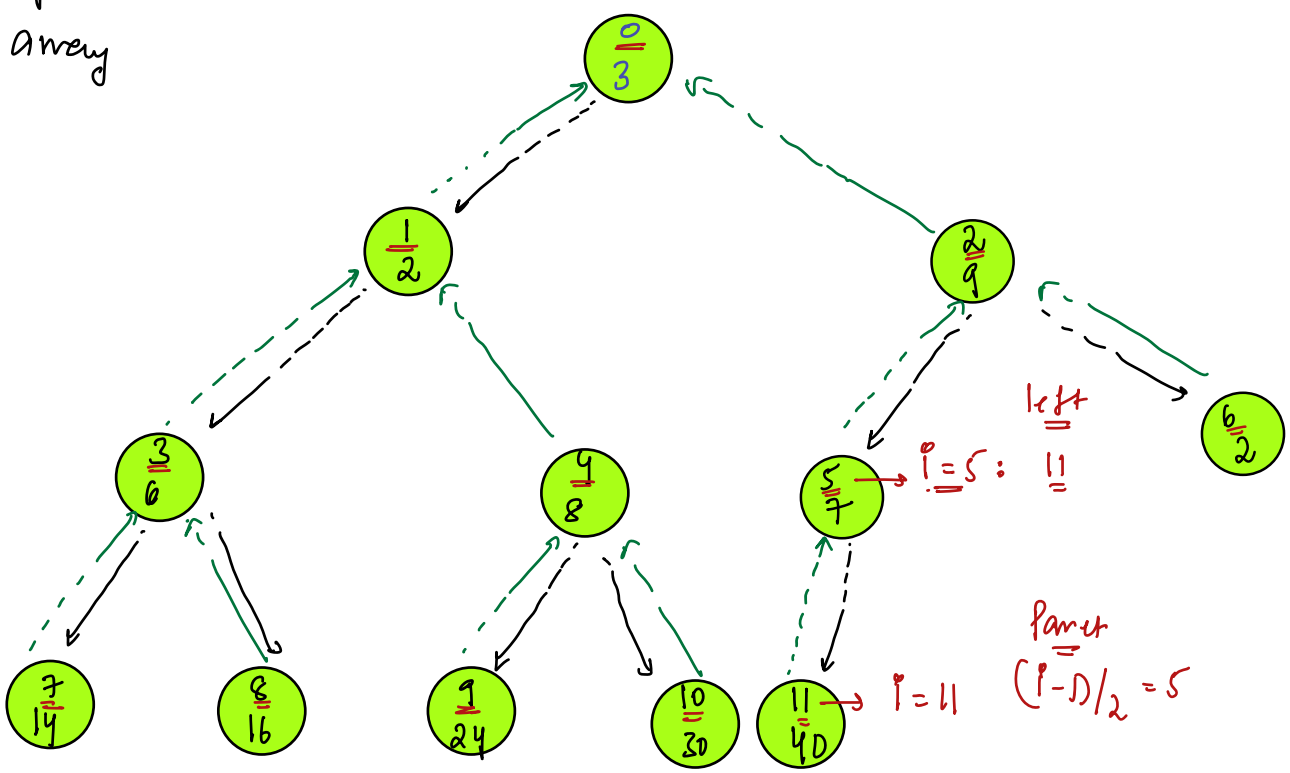
$\log_2 N$ height of CBT

Insert: 3 2 9 6 8 7 2 14 16 24 30

	0	1	2	3	4	5	6	7	8	9	10	11
list<int> l :	3	2	9	6	8	7	2	14	16	24	30	40

vector, ArrayList, Dynamic array

Tree structure is just for understanding } CBT



parent at index i = left $2i+1$, right $2i+2$

index i = parent $(i-1)/2$

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

Heaps : A Binary Tree is said to be heap

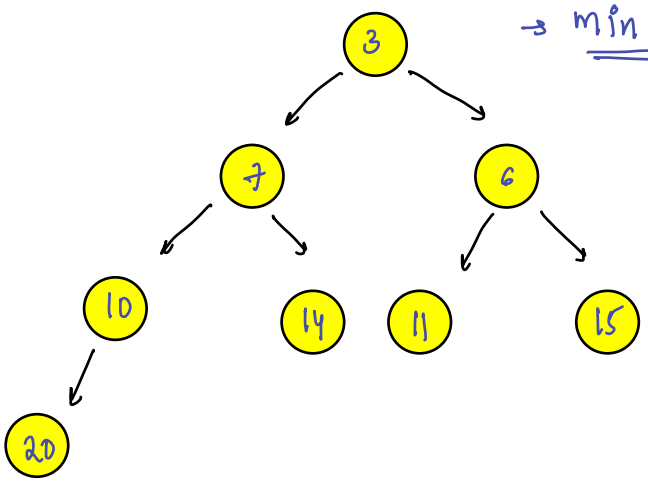
1) It has to be CBT

Either 1 & 2.1 or 1 & 2.2

2) i) For every node > both children : Maxheap

ii) For every node < both children : Minheap

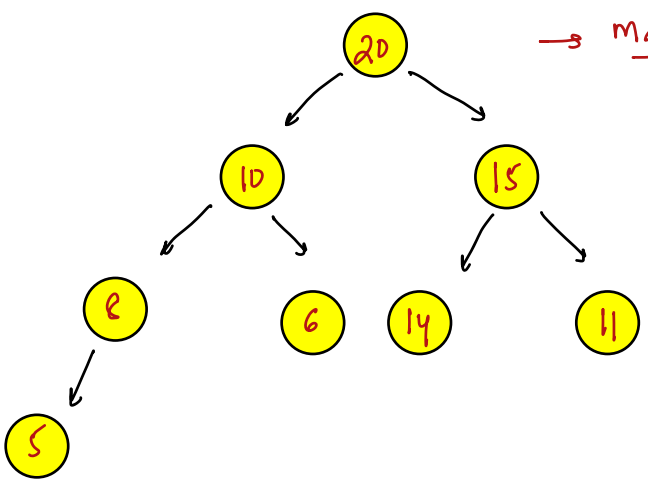
// Given CBT in Tree structure, & Stored in Array find Maxheap/Minheap



→ min heap : Every node < = Both Children

To handles duplicates

→ maxheap : Every Node > = & Both Children



Operations used by Heap

Minheap :

Insert → O(log N) (# size of heap)

getMin → O(1)

search( ) → O(N)

deleteMin → O(log N)

Maxheap :

Insert → O(log N) (# size of heap)

getMax → O(1)

search → O(N)

deleteMax → O(log N)

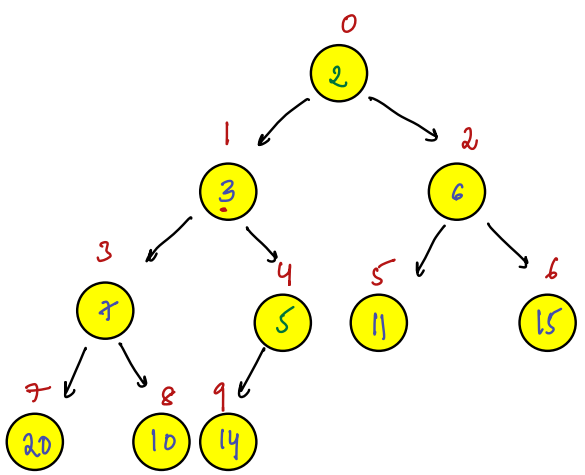
delet( ) → { search + { swap with last index / delet last index / propagate down } }

⇒ O(N) (search: O(N); swap with last index: O(1); delet last index: O(1); propagate down: O(log N))

Minheap operations :

list<int> ar :

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



$\log_2 N$

Insert 5 in minheap

ind	→	pare	if (ar[par] > ar[ind])
8		3	: swap
3		1	: swap
1		0	: no swap, break

Insert 2 in minheap

ind	→	pare	if (ar[par] > ar[ind])
9		4	: swap
4		1	: swap
1		0	: swap

0 : ind == 0, no parent break

Pseudo Code :

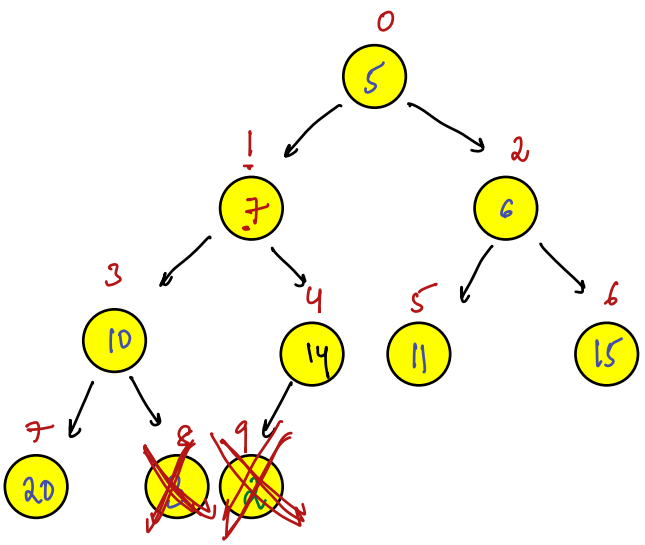
if it is already in min heap

```
void insert ( list<int> ar, int ele) { -> O(log2 N)
    O(1) -> ar.add(ele)
        int ind = ar.length - 1      { at what index ele is getting added }
        int par = (ind - 1)/2
        while ( ind != 0 && ar[par] > ar[ind] ) {
log2 N <-           swap ar[ind] & ar[par]
            ind = par
            par = (ind-1)/2
        }
}
```

Delete Min Operation :

	0	1	2	3	4	5	6	7	8	9
list<int> ar :	5	10	6	7	14	11	15	20	~~3~~	~~2~~

App:



1) Deleting from front in list : O(N)

2) Approach 2 :

O(1) : Step 1 : Swap root & last element

O(1) : Step 2 : Delete last index

O(log n) : Step 3 : propagating down

ind	left	right	min ind
0	1	2	1
1	3	4	4
4	9	10	

if (ar[ind] > ar[min_ind])

: Swap

: Swap

: break

↳ They don't exist

ind	left	right	min ind
0	1	2	1
1	3	4	3
3	7	8	7

if (ar[ind] > ar[min_ind])

: Swap

: Swap

: ar[3] > ar[7] : False break

↳ doesn't exist

final observations :

Heap :

a) Insert ⇒ (log N)

b) getMin / getMax ⇒ O(1)

c) deleteMin / deleteMax ⇒ O(log N)

d) search : O(N)

e) delete any number : O(N)

map / Tree-map /

BBST ⇒ Balanced Binary Search Tree

↳ AVL Tree

a) Insert ⇒ (log N)

b) getMin / getMax ⇒ O(log N)

c) deleteMin / deleteMax ⇒ O(log N)

d) search : O(log N)

e) delete any element : O(log N)

→ When are heaps preferred ?

In your question, if we have to use only following operations

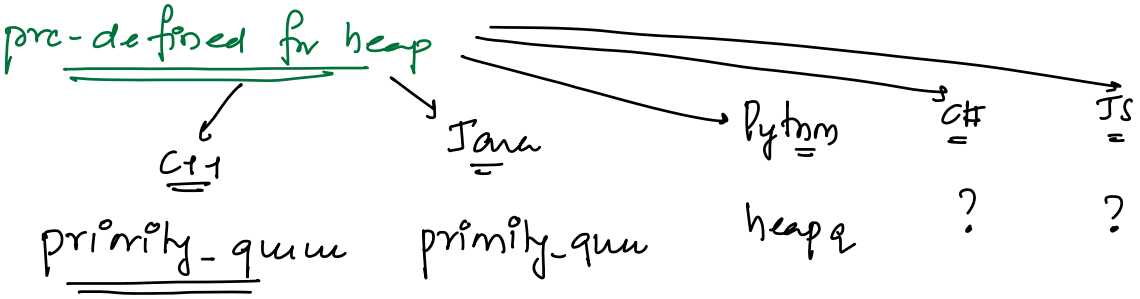
a) Insert()

b) getMin() / getMax()

c) deleteMin() / deleteMax()

} faster in heap when compared to BBST

⇒ pre-defined for heap



Heap Problems :

→ Monday

→ Wednesday

sorting → comperator

Interview: google

1) Dp / graphs / Binary Trees / Hashing + geometry

2) Backtracking / Strings / Binary Search

3) Arrays