B TREES (DBMS)

Self Balancing Trees

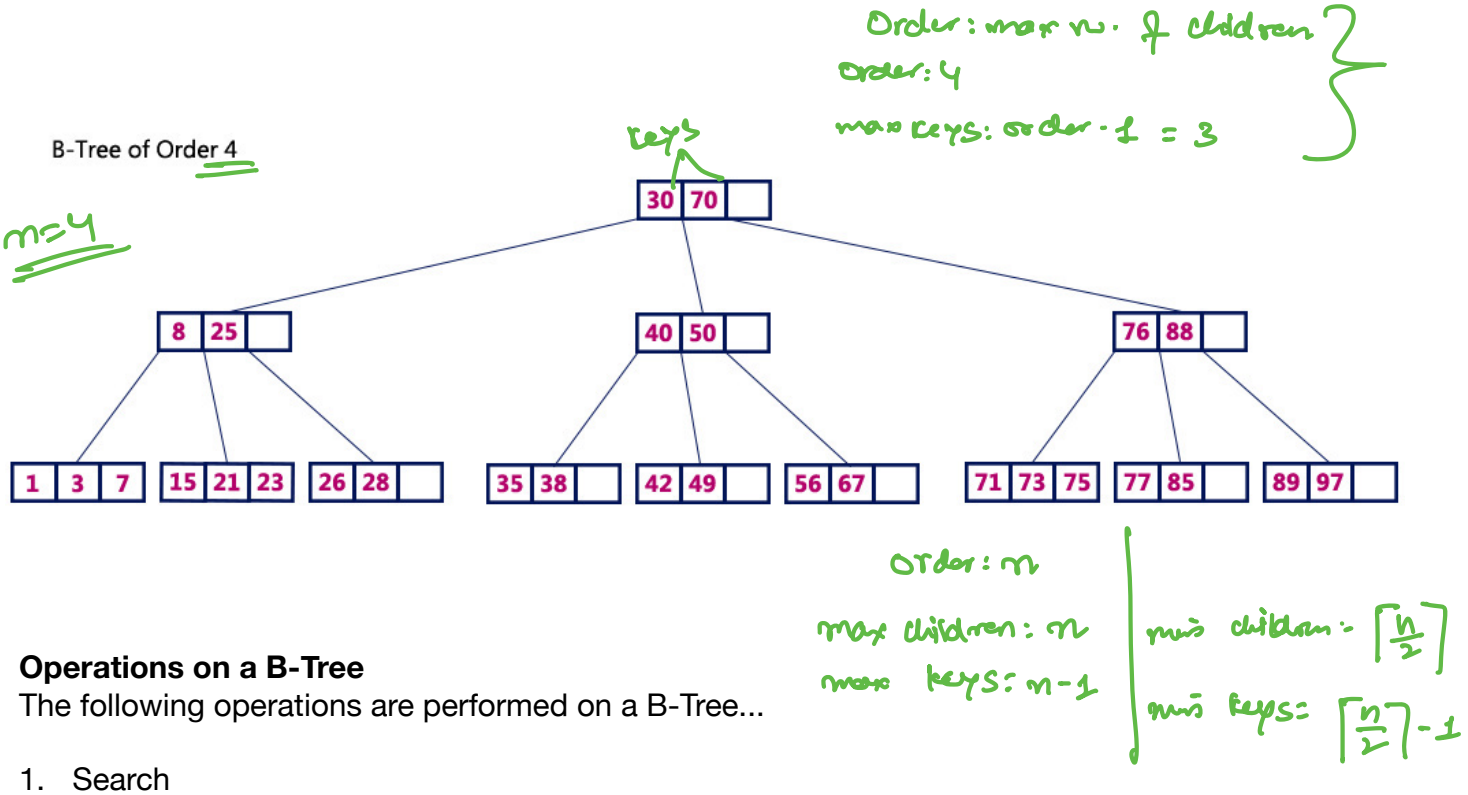
- In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children.
- But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

B-Tree of Order m has the following properties...

Property #1 - All leaf nodes must be at same level.
Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
Property #4 - If the root node is a non leaf node, then it must have atleast 2 children.
Property #5 - A non leaf node with n-1 keys must have n number of children.
Property #6 - All the key values in a node must be in Ascending Order.



Operations on a B-Tree
The following operations are performed on a B-Tree...

1. Search
2. Insertion
3. Deletion

Search Operation in B-Tree

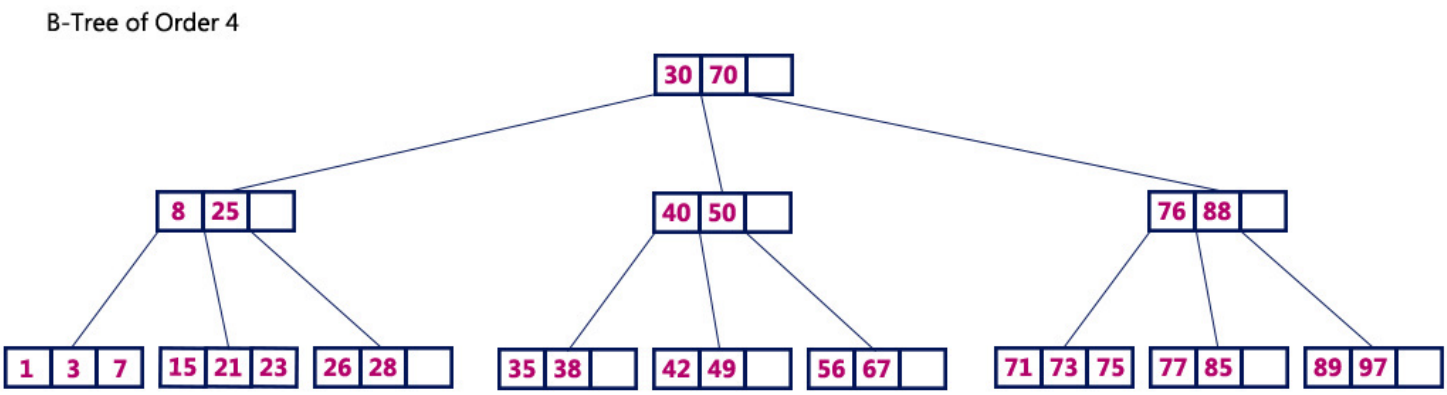
The search operation in B-Tree is similar to the search operation in Binary Search Tree.
In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree).
In B-Tree also search process starts from the root node but here we make an n-way decision every time, where 'n' is the total number of children the node has.
In a B-Tree, the search operation is performed with O(log n) time complexity.

The search operation is performed as follows...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with first key value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always



attached to the leaf node only. The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.
Step 2 - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.
Step 3 - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
Step 5 - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Order: 4

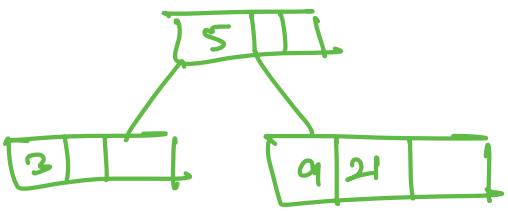
max children = 4	min children = $\lceil \frac{m}{2} \rceil = 2$
max keys = 4 − 1 = 3	min keys = 1

5

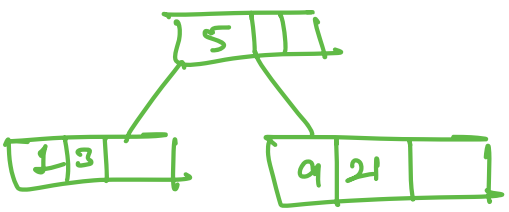
3

21

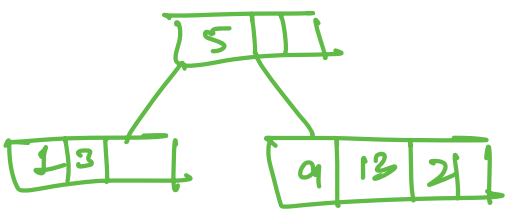
9



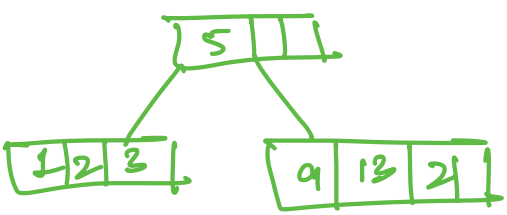
1



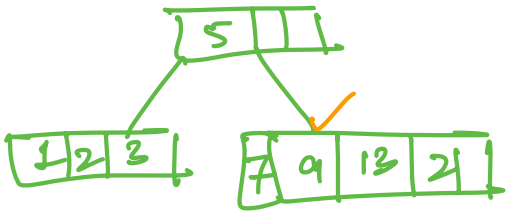
13



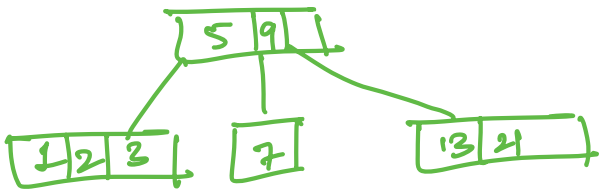
2



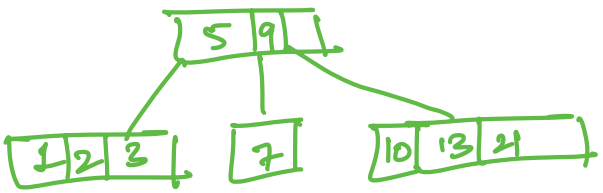
7



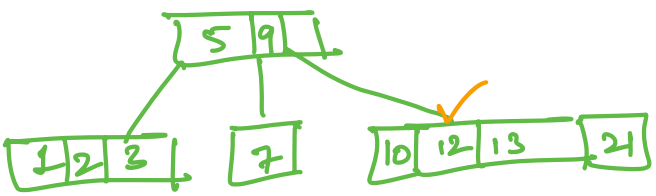
keys: m
child: m+1

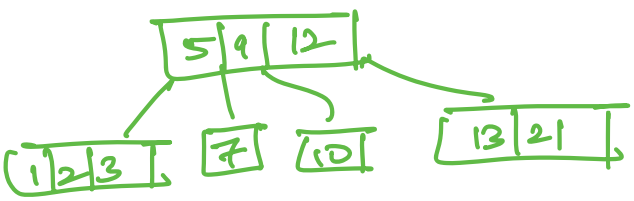


10

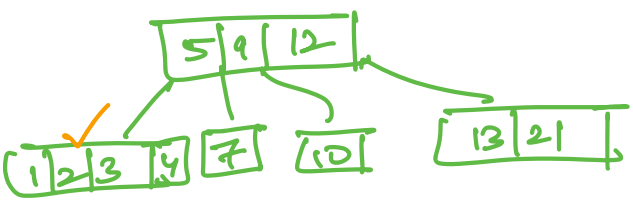


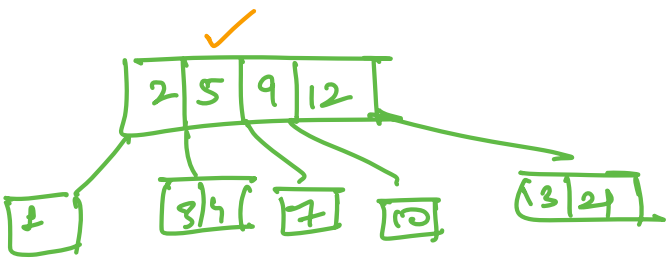
12

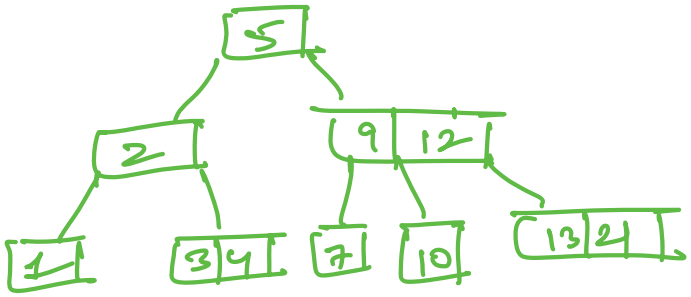




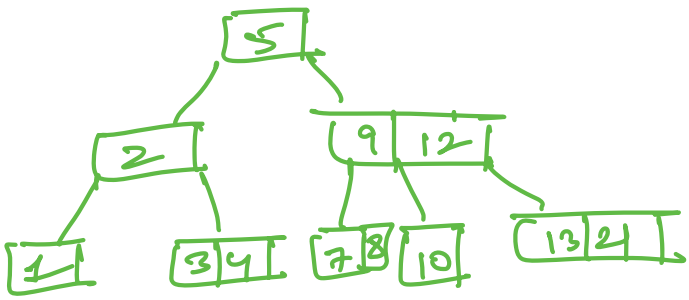
4







8:



Delete:

Order: 5

min child: $\lceil \frac{5}{2} \rceil = 3$

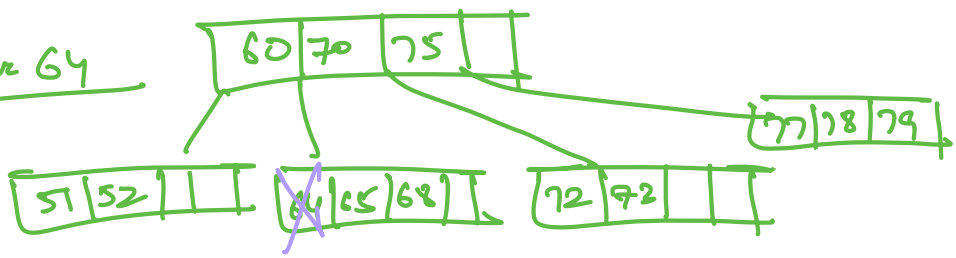
max child: 5

max keys: 4

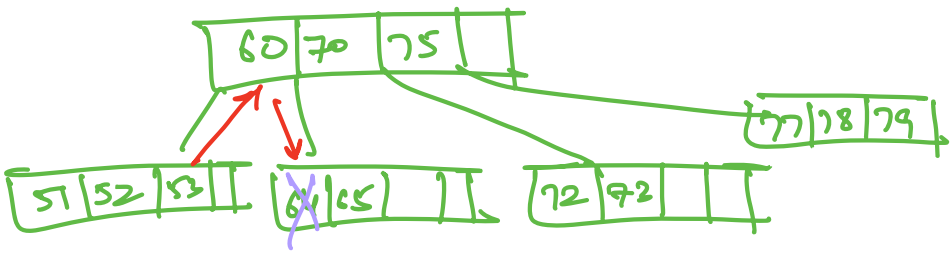
min keys: 2

Case 1:

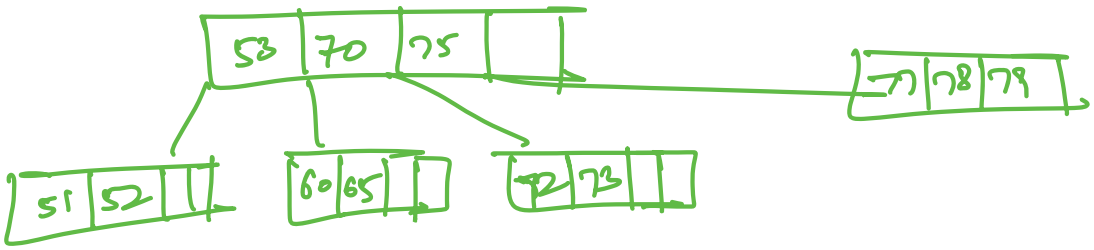
Remove 64



Case 2:

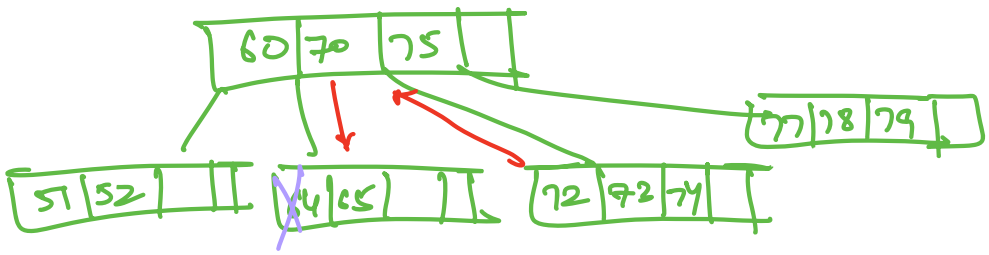


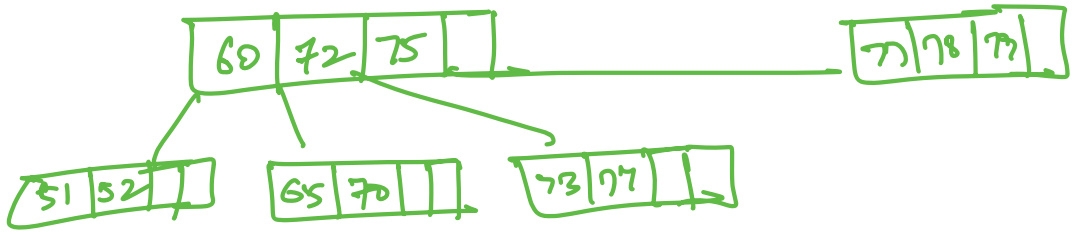
Remove 64



Case 2:

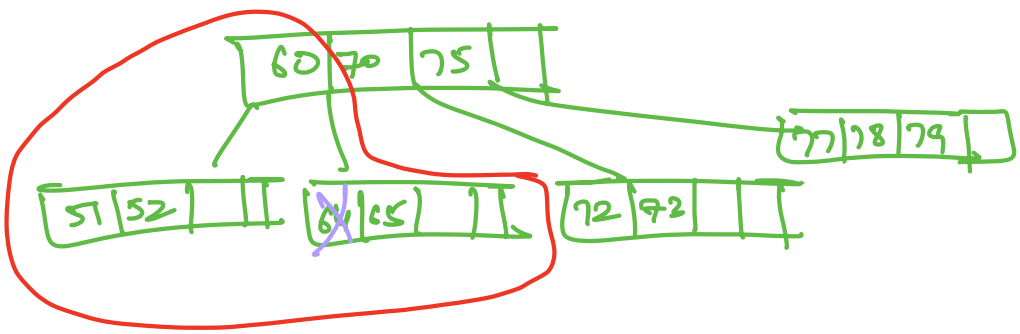
Remove 64

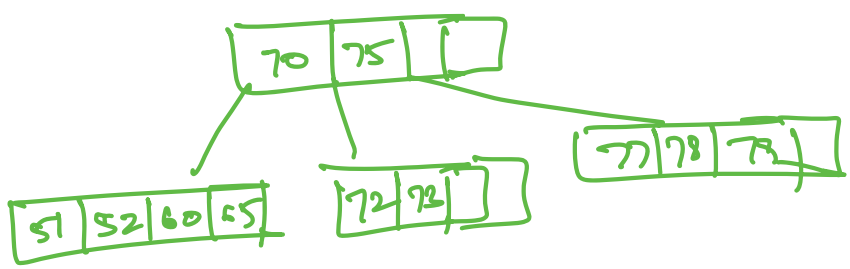




Case 4:

delete 64

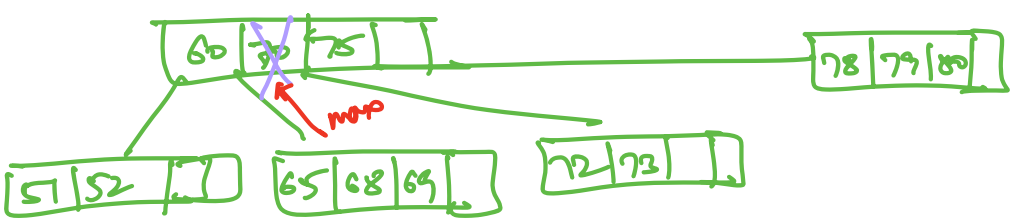


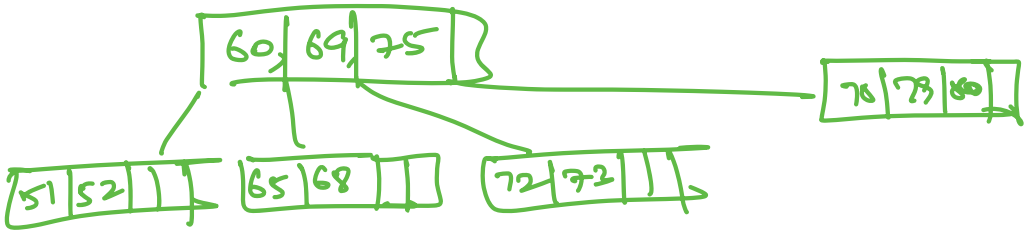


Non leaf node

Case 1:

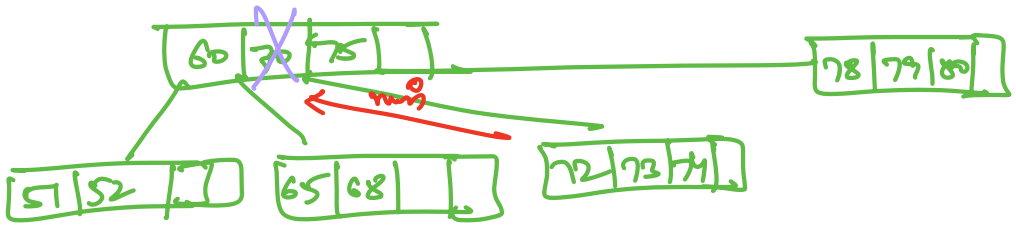
delete 70

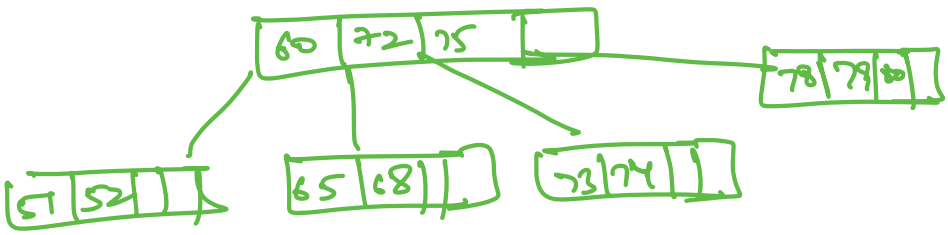




Case 2:

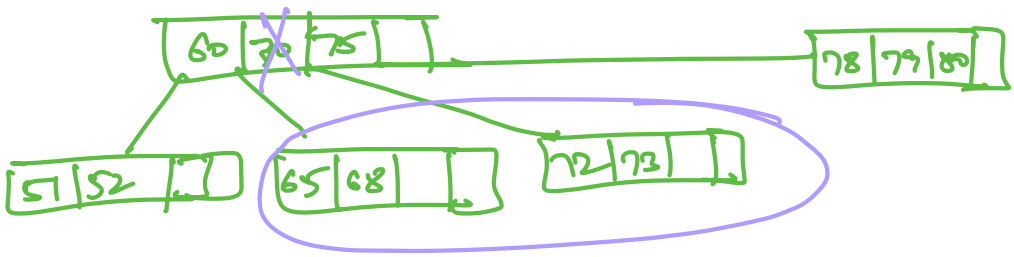
delete 70

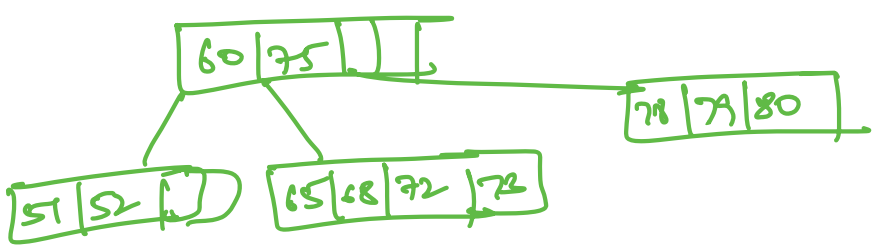




Case 3:

delete 70

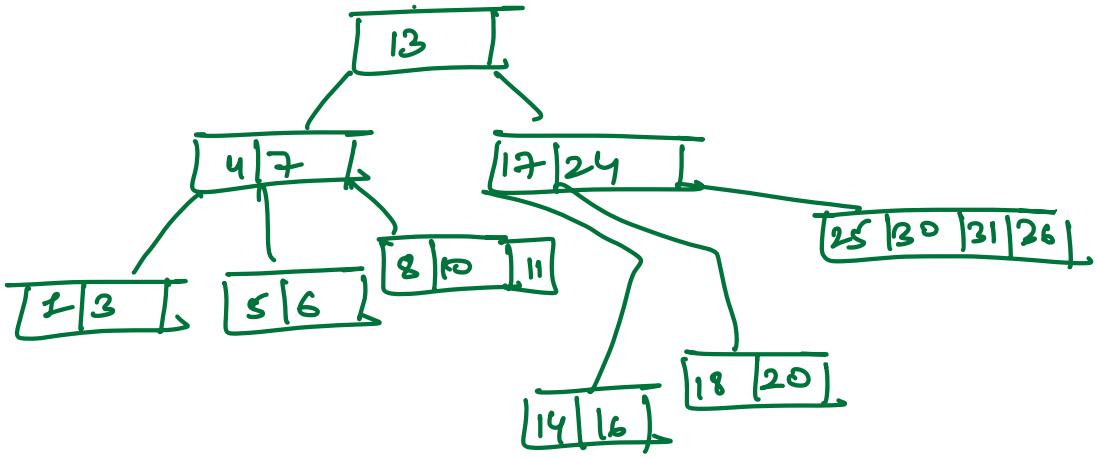


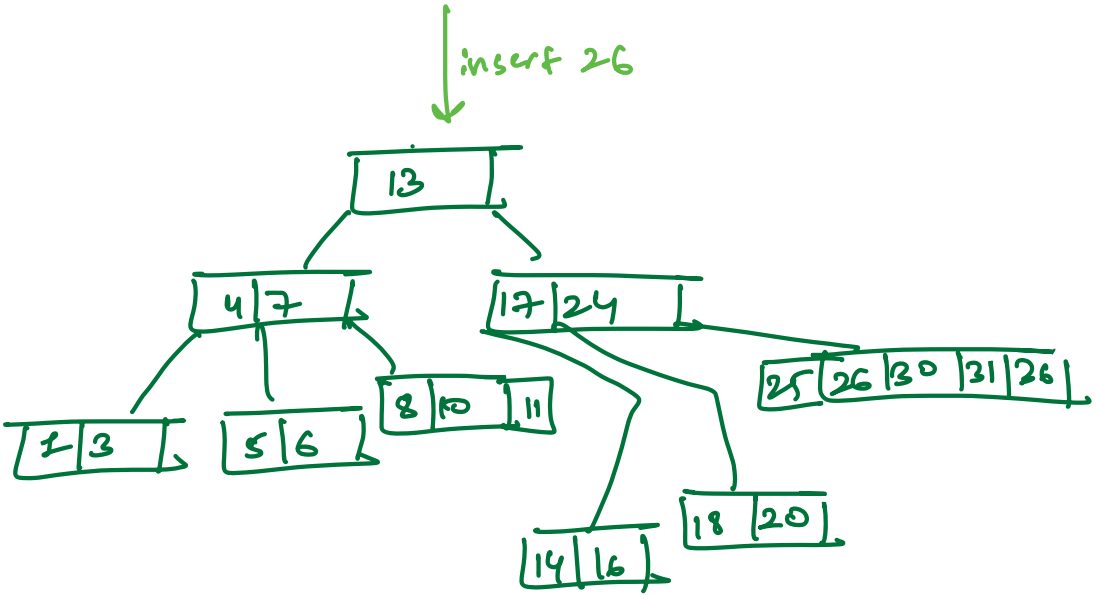


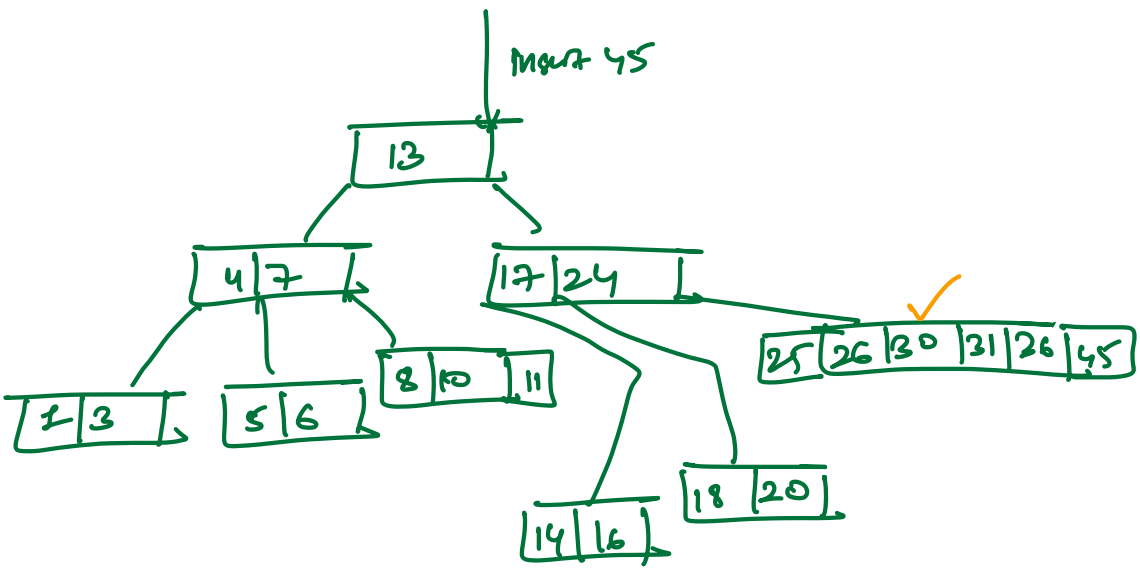
PYQ ETE:

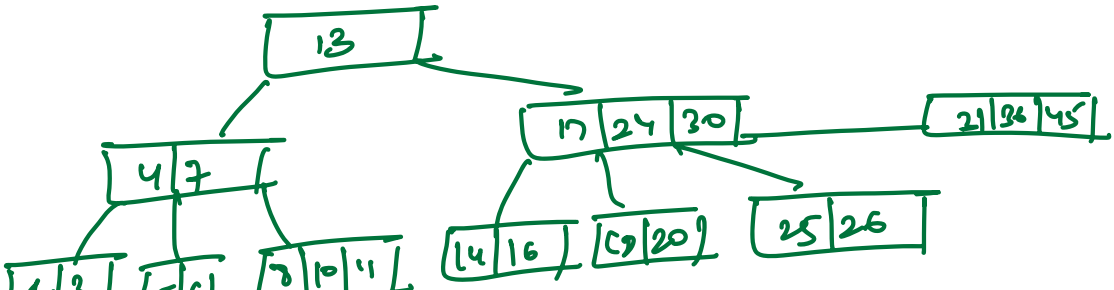
order: 6

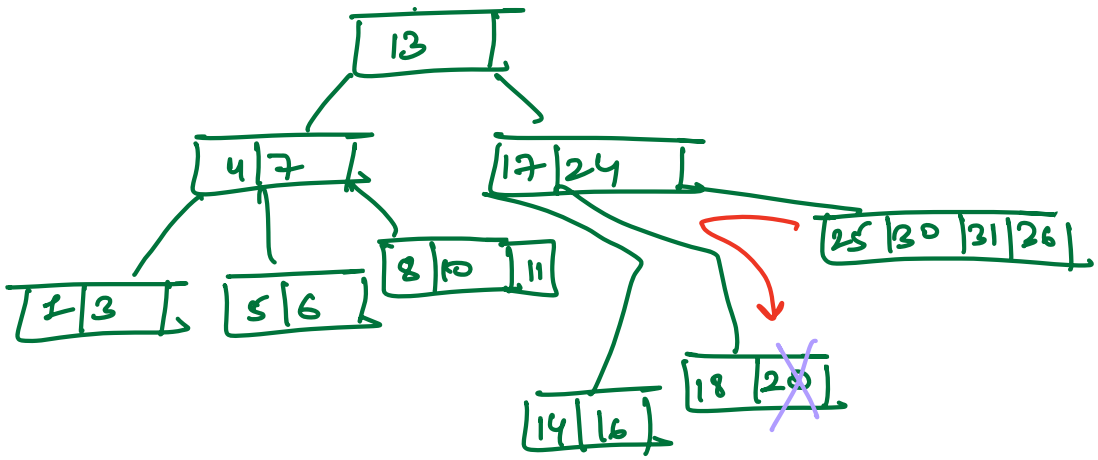
max child: 6	min child: $\lceil \frac{6}{2} \rceil = 3$
max keys: 5	min keys: 2


13
4 7
17 24
25 30 31 26
1 3
5 6
8 10 11
18 20
14 16

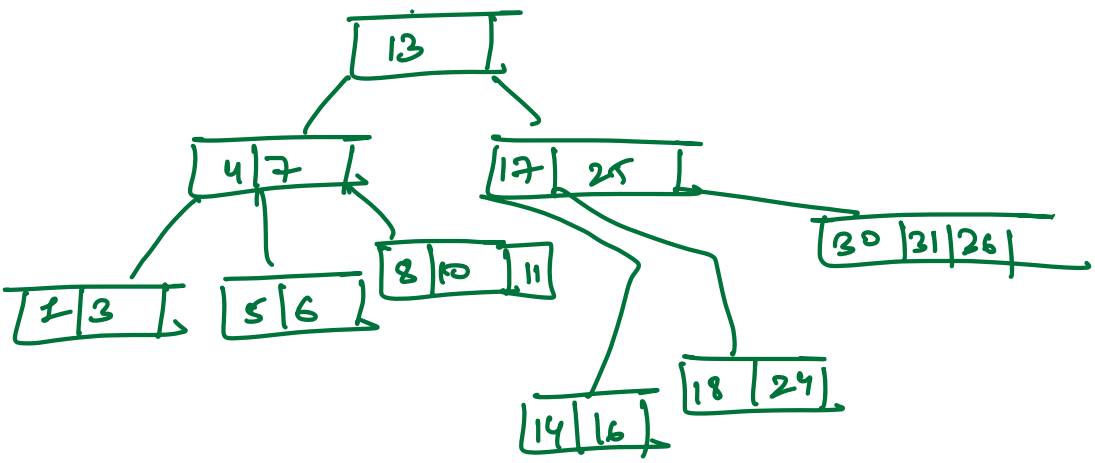

insert 26
13
4 7
17 24
25 26 30 31 26
1 3
5 6
8 10 11
18 20
14 16

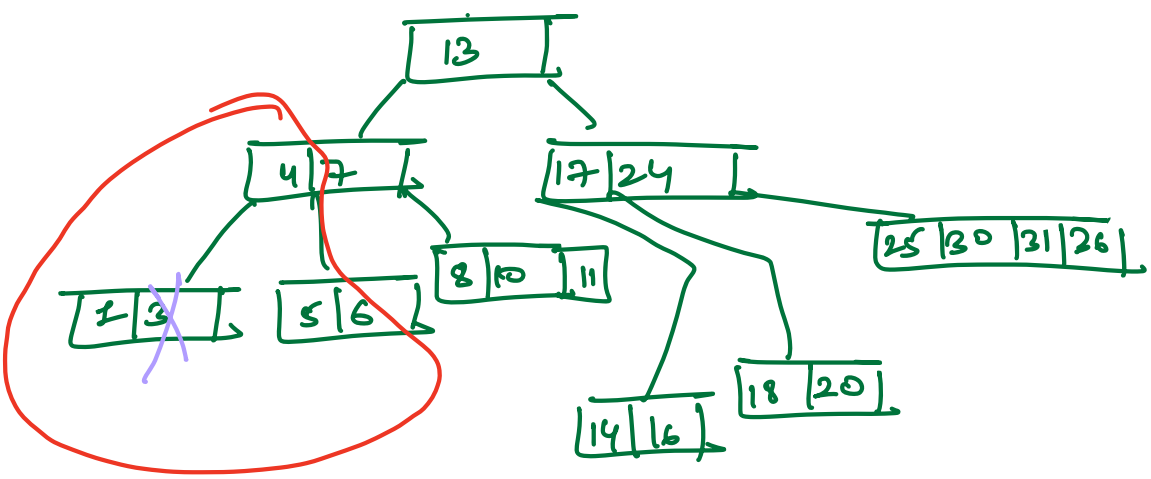

insert 45
13
4 7
17 24
25 26 30 31 26 45
1 3
5 6
8 10 11
18 20
14 16

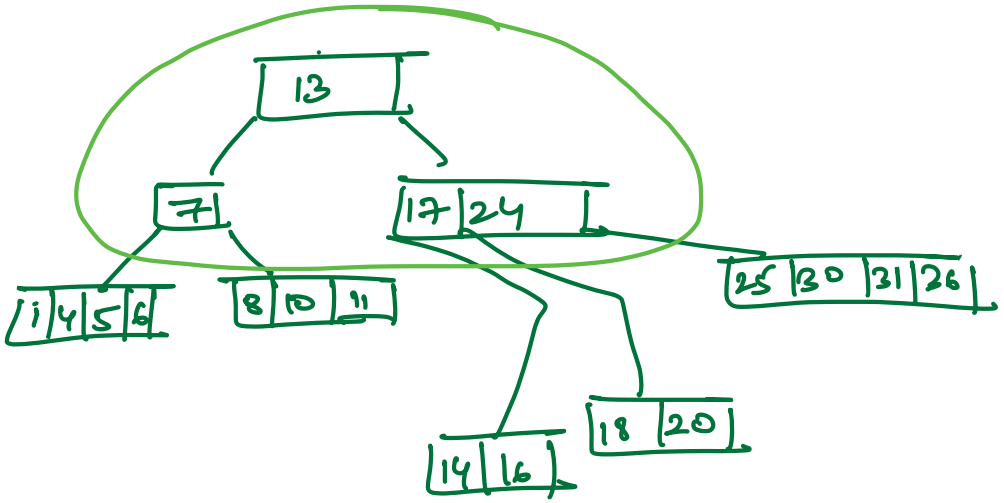

13
17 24 30
4 7
25 26
14 16
18 20

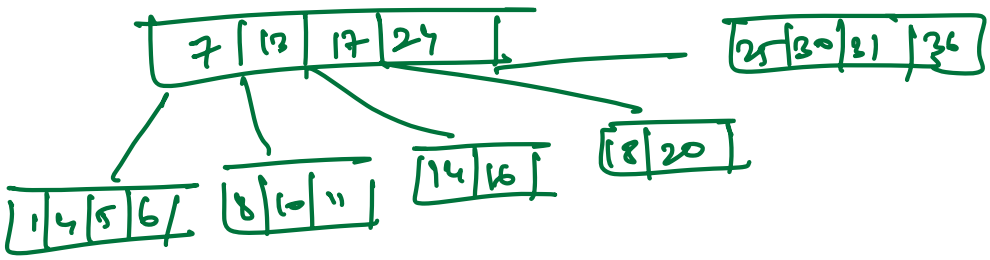

13
4 7
17 24
1 3
5 6
8 10 11
25 30 31 26
18 20
14 16

delete 20


13
4 7
17 25
1 3
5 6
8 10 11
30 31 26
18 24
14 16


13
4 7
17 24
1 3
5 6
8 10 11
25 30 31 26
18 20
14 16





Heap
Graph
Hashing

HEAP / PRIORITY QUEUE

Queue: FIFO

Heap/PQ: insertion can be any order
remove: highest Priority element

PRIORITY

- marks → higher value ⇓ higher Priority → max heap
- Rank → Smaller value ⇓ higher priority → min heap

	Add	min Delete	get min
Unsorted Array	O(1)	O(n)	O(n)
Sorted Array (Desc)	O(n)	O(1)	O(1)
Heap	O(log n)	O(log n)	O(1)

	Unsorted Array	Sorted Array	Heap
Add	1	n	log n
Delete	n	1	log n
	n	n	log n

Heap ?

- Complete BT
- Parent Priority > Child Priority

Max Heap

Parent value > Child value

Min Heap

Parent value < Child value

full BT

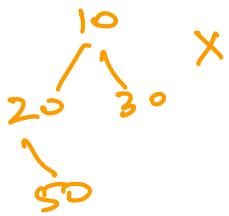
Post + Pre

0 child
2 child

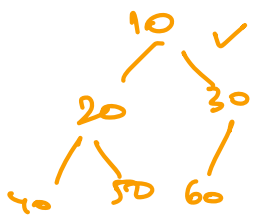
Complete BT

Levels: filled left → Right

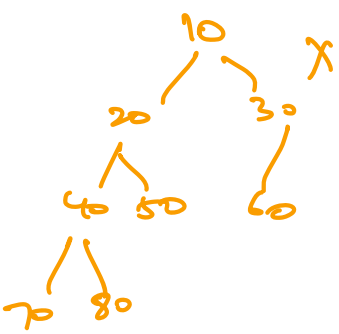
Before starting a new level, previous levels should be filled



X



✓

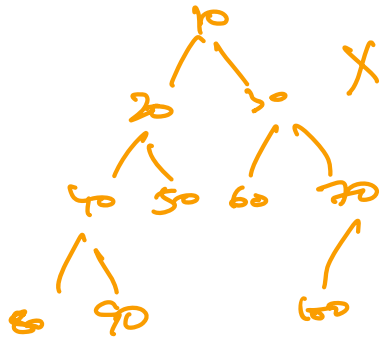


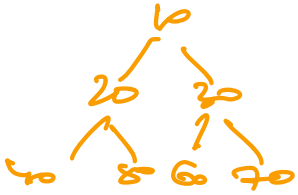
X

Levels:

n-1 levels completely filled

nth level left → right

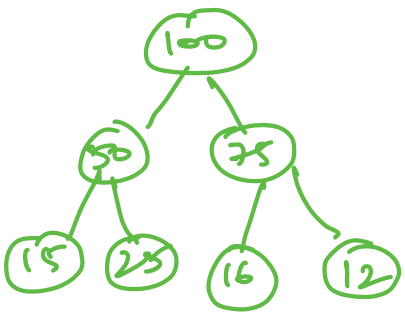




Priority

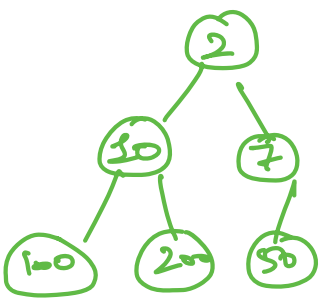
Marks

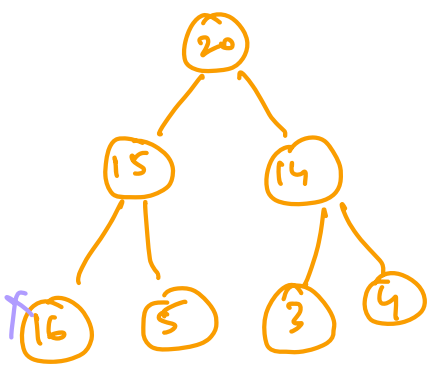
Max Heap

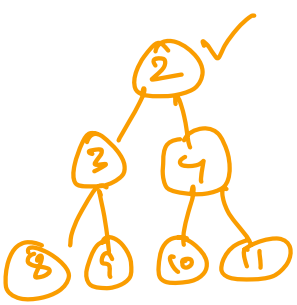


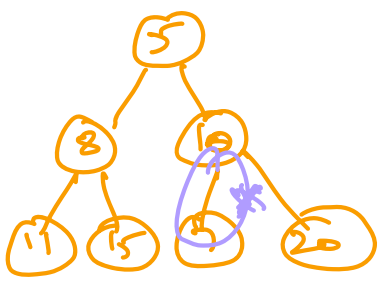
Rank

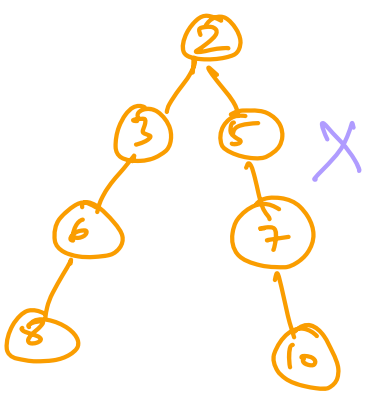
Min Heap





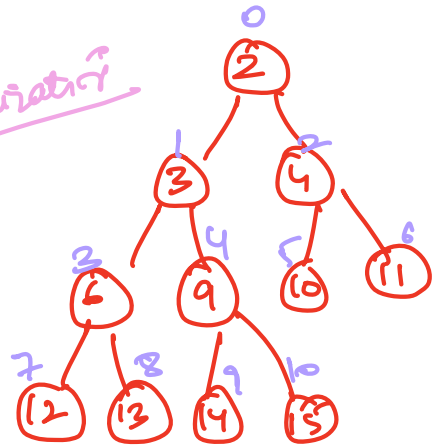






Implementation

Visualization



Exact Storage

0	1	2	3	4	5	6	7	8	9	10
2	3	4	6	9	10	11	12	13	14	15

$pi = 2$ | 3 | pi
$li = 5$ | 7 | $2pi+1$
$ri = 6$ | 8 | $2pi+2$

$li=5$, $ri=6$] → 2 $\quad pi = \frac{(ci-1)}{2}$

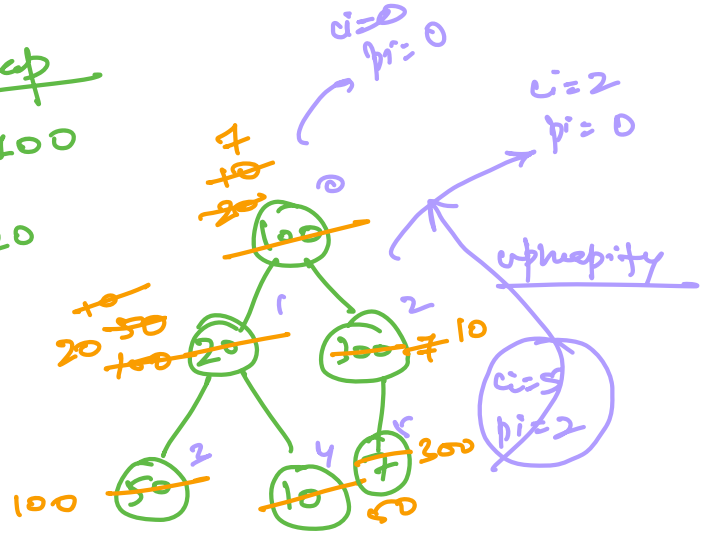
$\frac{5-1}{2} = \frac{4}{2} = 2$

$\frac{6-1}{2} = \frac{5}{2} = 2$

Heapify

add 100

add 20

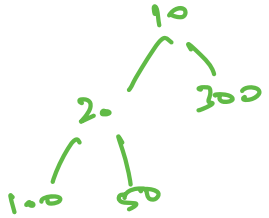


0	1	2	3	4	5
100	20	300	50	10	7

$\frac{2-1}{2} = \frac{1}{2} = 0.5$

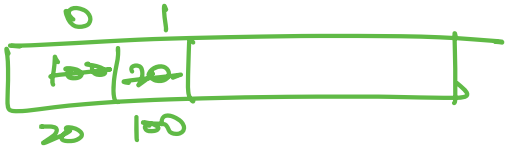
arr: 10 20 300 100 50

Sorted ✗

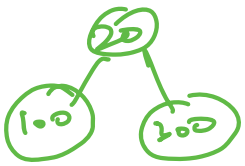


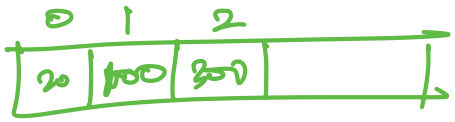
add 100

add 20

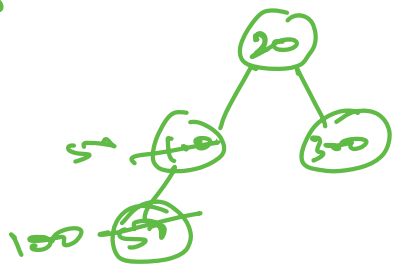


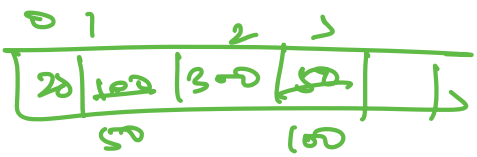
add 300





add 50





add 10