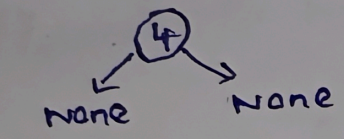
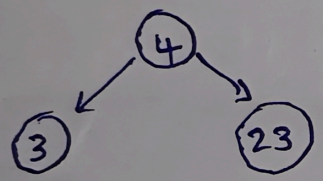
Trees.

inary tree.

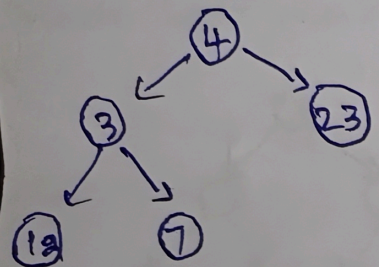


⇒ { 'value': 4,
'left': None,
'right': None }

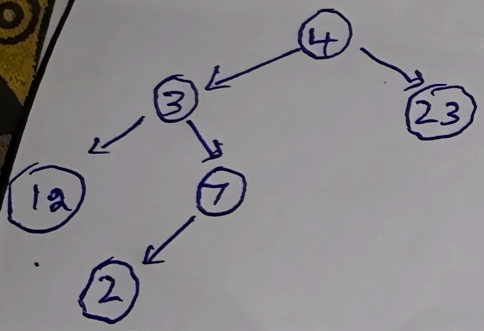


⇒ {'value': 4,
'left': {'value': 3,
'left': None,
'right': None},
'right': {'value': 23,
'left': None,
'right': None} }

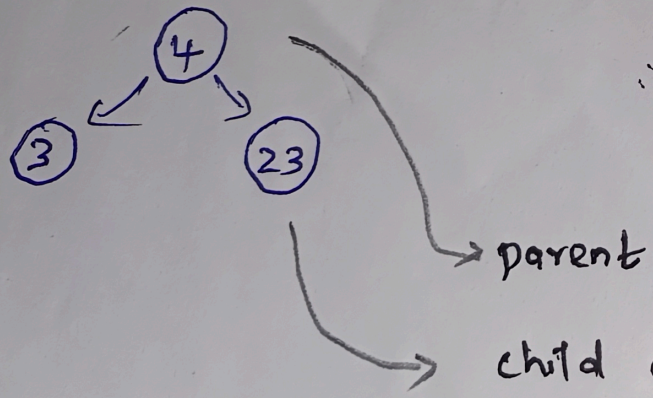
Terminalogy. (full, perfect, complete).



→ full tree (every node have 2 or no pointer)
not completed (left right not balanced)

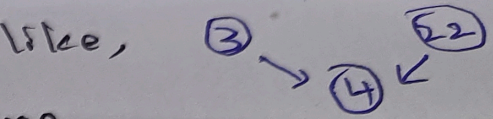


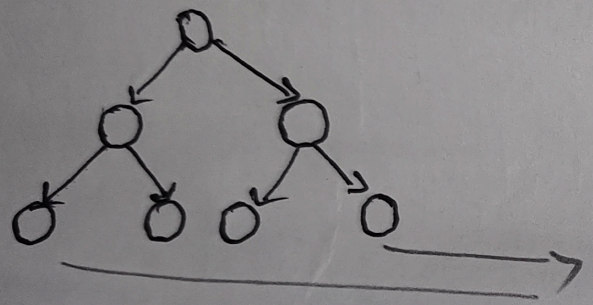
→ not full
not completed.



∴ → full node
And perfect tree.

every node only have one parent if something like, node have more than one parent thats not a tree.





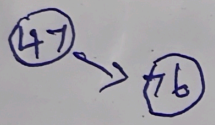
leafs because they dont have a child node.

Binary tree, → overview.

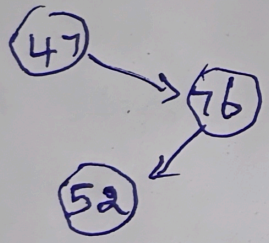
(47, 76, 52, 21, 82, 18, 27)

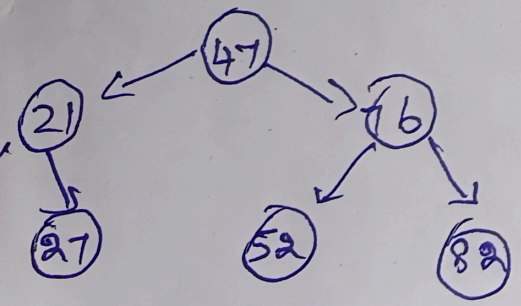
→ first node in our tree. to add next node 

if new node value greater than the before it goes right, if less goes left. here 76 > 47 so right



and next 52 is also greater than 47 so it goes right but in that place there is a node there so compare with that 52 < 76 so need a add left to 76.

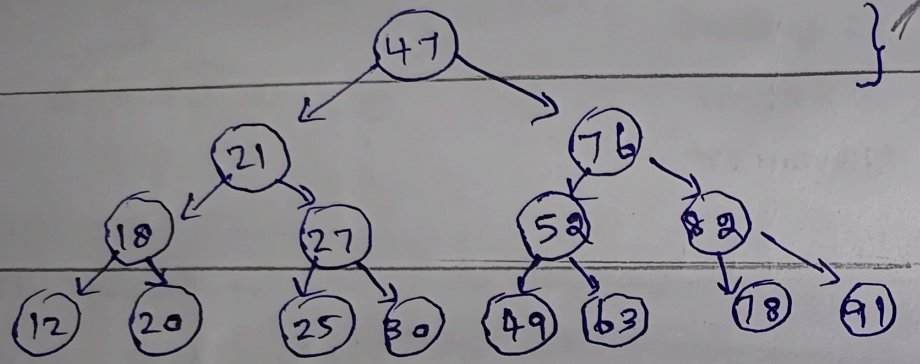


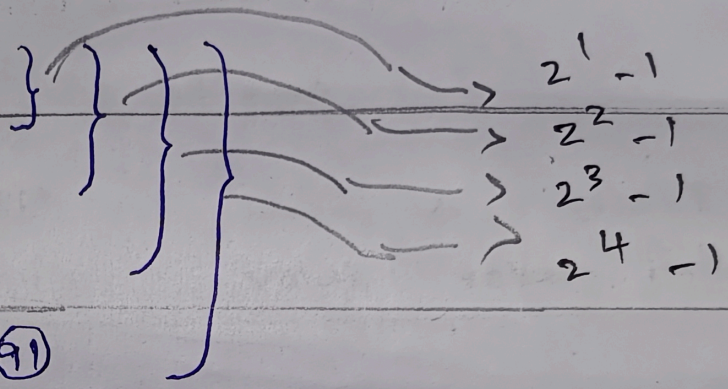


← Continuing that process ....

→ if the tree is in this format its called binary tree.

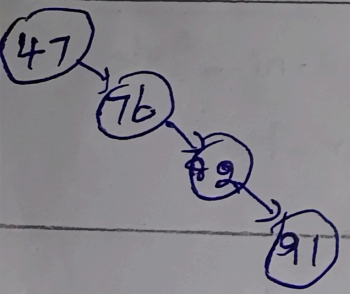
binary tree big o.





In this tree if we need to find 49 it takes ~~49~~ 4 steps same for remove and add so this is o(log n) (divide & conquer) because each steps the input size reduced by half not exactly.

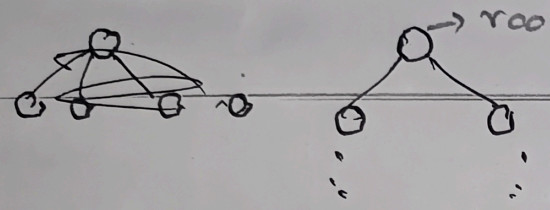
in our worst case.



→ here the tree never forks, so its essentially a linked list here if we need to find 91 it takes 4 steps. for input size 4 it takes 4 steps so its o(n).

④. So binary tree technically a o(n) not o(log n).

If our binary tree not ~~leafe~~ in worst case like all



lookup()
insert()
remove()
} O(logn).

binary tree didnt have a duplicate value return false in that case.

```
class Node :
    def __init__(self, value):
        self.value = value
        self.right = None
        self.left = None

class BinarySearchTree :
    def __init__(self):
        self.root = None.
```

constructor.