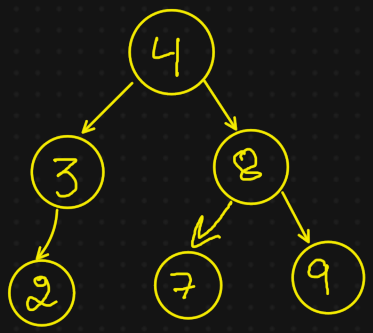
# Binary Search Tree (BST)

Binary Search tree (BST) are special kind of binary tree with following property.



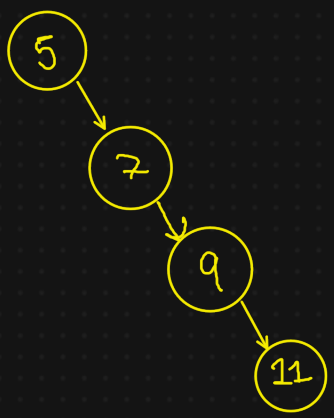

→ Left subtree: value of each node is less than root's node value.

→ Right subtree: value of each node is greater than root's node value.

→ the left and right subtree are also BSTs

Inspired by Binary Search algorithm.

$O(n) \rightarrow O(\log n)$




Real world example:-

→ Phone book/dictionary

→ File search

→ Auto complete

Industry :-

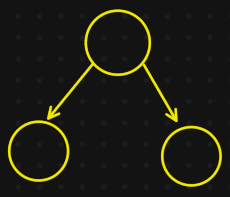
→ Database (indexing)

→ ML (Decision tree)

→ Range Queries in DBs

The structure of BSTs allow for efficient searching, insertion and deletion.

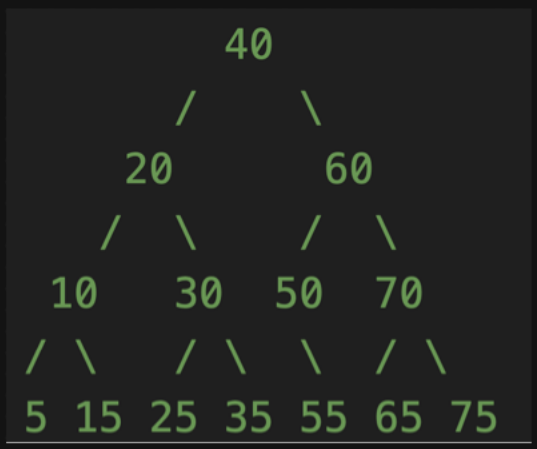
# BST node



## Search in a BST

value = 25

```
if root.data == value
    return True
if root.data > value :
    search (root.left)
    # not be in right subtree
elif root.data < value :
    search (root.right)
```



# Sorted array / list to a BST

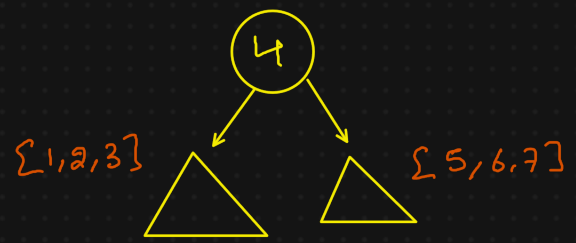
1 2 3 4 5 6 7


skewed BSTs

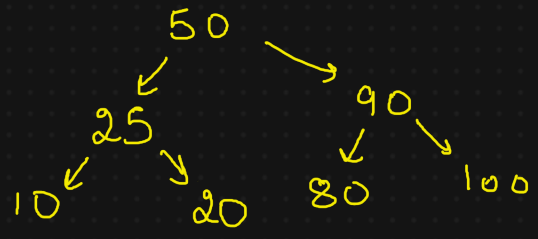
balanced BSTs

1. pick mid and start by creating its node

2.



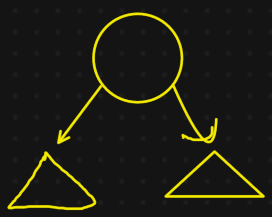
# Check BST




node.left < node & node < node.right

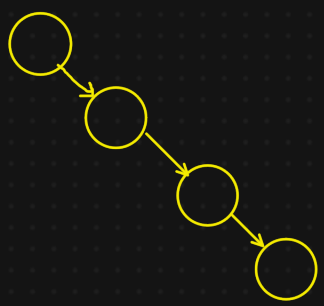
1. Check with maximum value of left subtree and minimum value of right subtree
2. each and every subtree also a BST.

## Time Complexity of Check BST approch



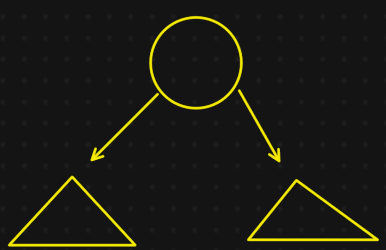
$T(n) = Kn + 2T(n/2)$

$O(n \log n)$



$T(n) = Kn + T(n-1)$

$O(n^2)$



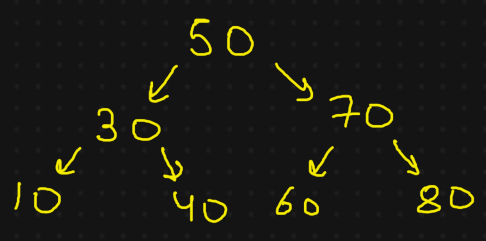
min , max , isBST  min , max , isBST

ans = root.data > max (left)  root.data < min (right)  isBST left  isBST

return ans, min(left.min, node.data), max(right.max, root.data)

Print elements in a range

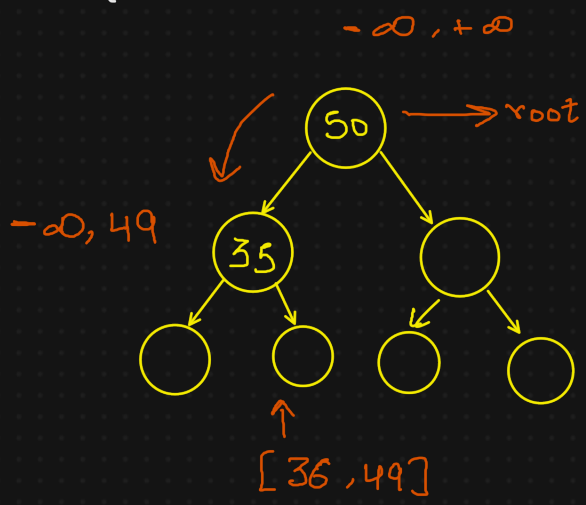
1. [15,45]
2. [55,75]



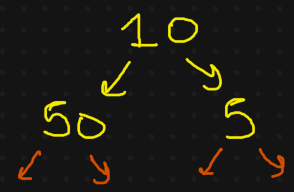

[55,75]

Check BST using range limits

I am reducing / redefining the range in which my nodes can take value.



# BST class



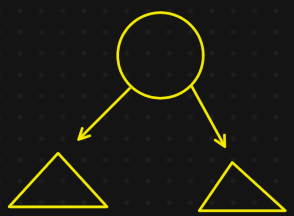

20

For Binary tree and generic trees, no logic is defined.

3 major methods.

1. Insert
2. Search
3. Delete

## Search



## Insert

None | 20

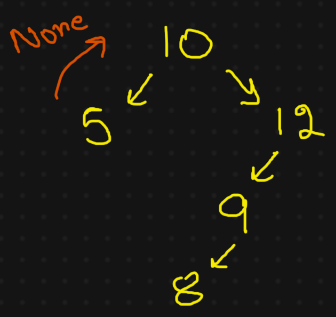
root is getting changed so we should return and update.

# Delete a node in BST

1. if root is None
   return None

None ⟶ None

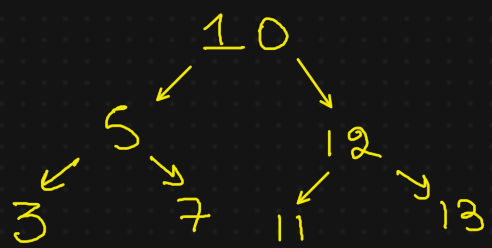
2. if root.data > data
   root.left = delete (data, root.left)



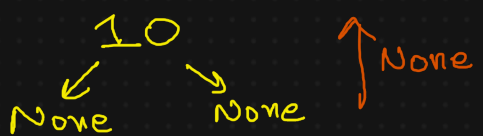

3. if root.data < data
   root.right = delete (data, root.right)

4. if root has to be deleted

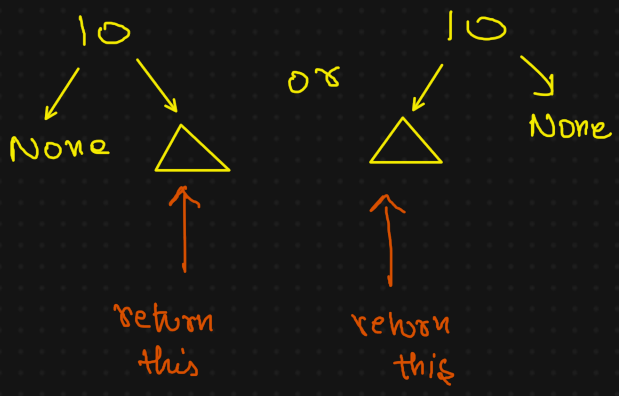
Delete can have multiple cases when we are on the node which is to be deleted.




1. When left and right are None
   i.e. node to delete is a leaf.

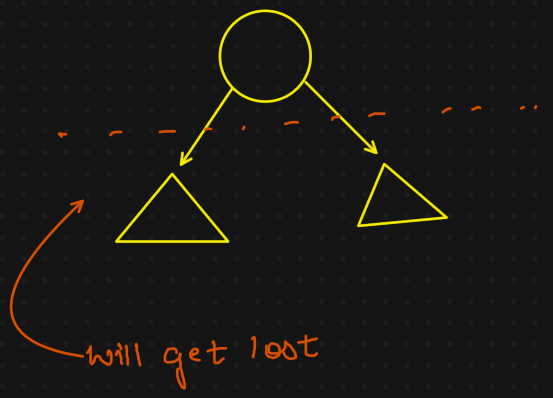



2. When either of left or right is None




When we delete 10, the leftover tree will be returned

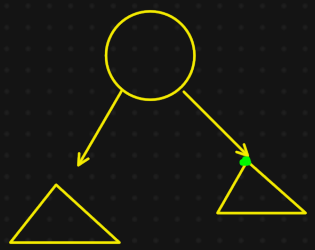
4. When left and right both are Not None




We need a replacement in place of the deleted node.

There can be 2 good replacement for node.

1. Lowest node on right side
2. Highest node on left side

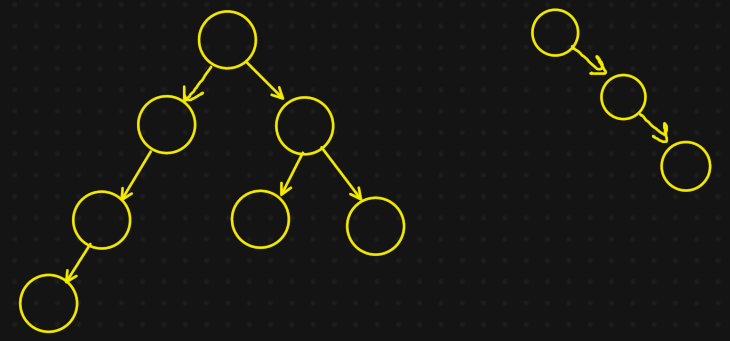


When we go left, value decreases.

Once we change the data in my node to be deleted, we delete the replacement.

# Complexity of BST class

## 1. Search

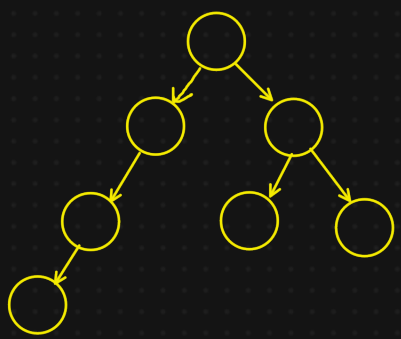


$O(h)$

$\Downarrow$

height of tree

## 2. Insert

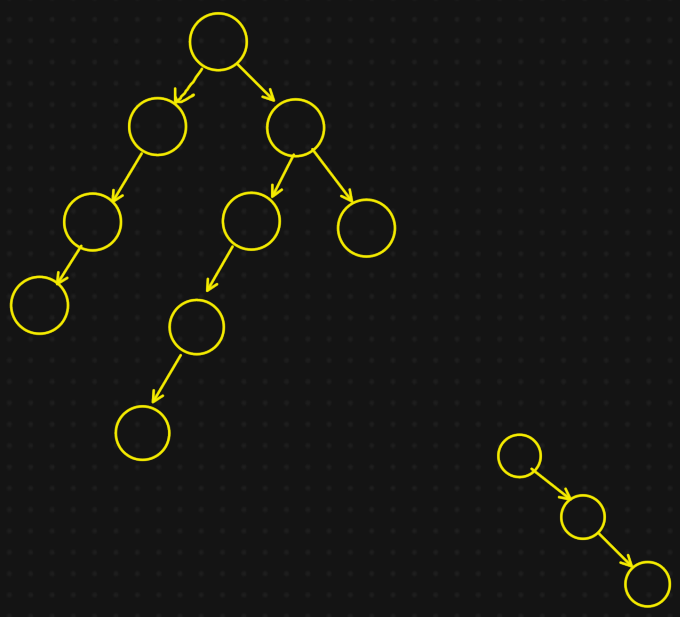


$O(h)$

$\downarrow$

height of my tree

## 3. Delete



(a) going/finding node we want to delete $\rightarrow h_1$

(b) find replacement $\rightarrow h_2$

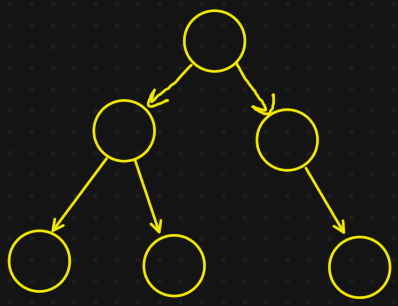
(c) delete the replacement $\rightarrow h_2$

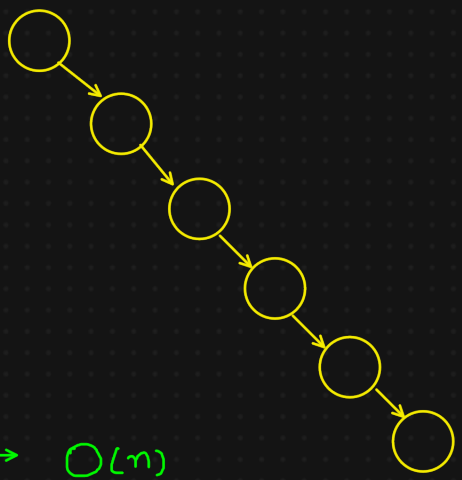
$h_1 + h_2 + h_2 < 2h$

$O(h)$

For a balanced BST, all operation are $O(\log n)$ (approx.)

# Balancing the BST





$O(\log n)$ ⟵ $O(h)$ ⟶ $O(n)$

## BST types

1. 2-4 tree
2. Red Black
3. AVL tree