

WEEK 9

CSE 331

CC7

Instructor Sebnem Onsay



Binary Search Tree (BST)

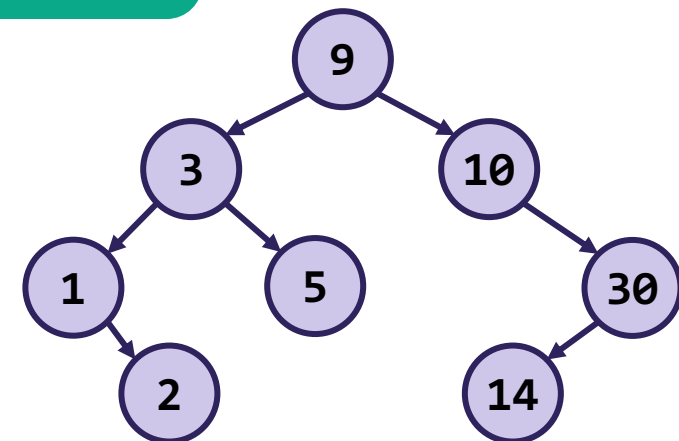
• Invariants

- The *rules* for your data structure or algorithm
- Whenever you implement any operation on your data structure:
 - You know the invariants are true at the beginning. Great! Simpler code, fewer cases
 - But you must leave the invariants true at the end.
- Accidentally violating invariants a common source of bugs. Defensive programming: check invariants at beginning/end of methods!

✓ INVARIANT

```
def search(self, value, node):  
    # Check root  
    if node is None:  
        return None  
    if node.value == value:  
        return node  
    # Check left subtree (less than case)  
    elif value < node.value and node.left is not None:  
        return self.search(value, node.left)  
    # Check right subtree (greater than case)  
    elif value > node.value and node.right is not None:  
        return self.search(value, node.right)  
    # Otherwise, value not found but return closest leaf  
    return node
```

✓ INVARIANT



INVARIANT

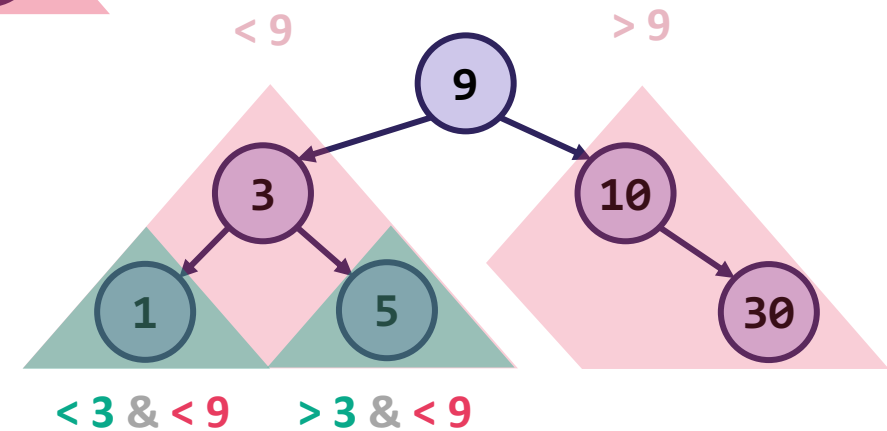
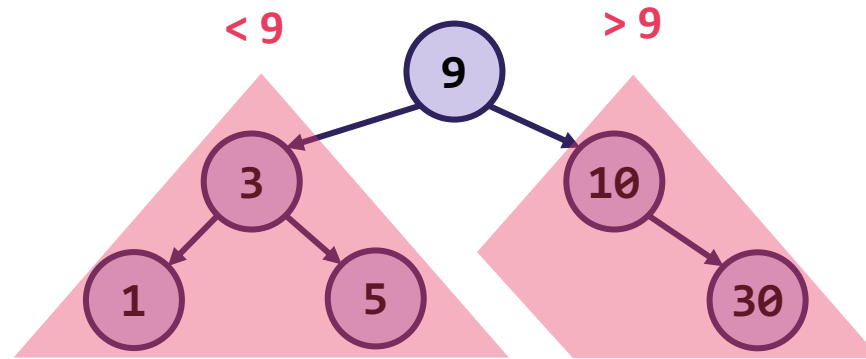
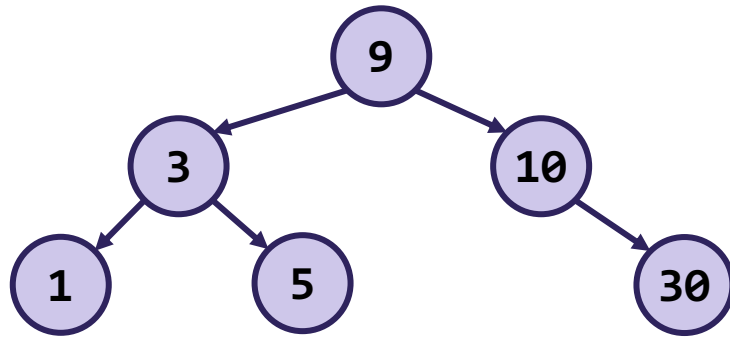
Binary Search Tree Invariant:

For every node with key k :

- The left subtree has only keys smaller than k .
- The right subtree has only keys greater than k .

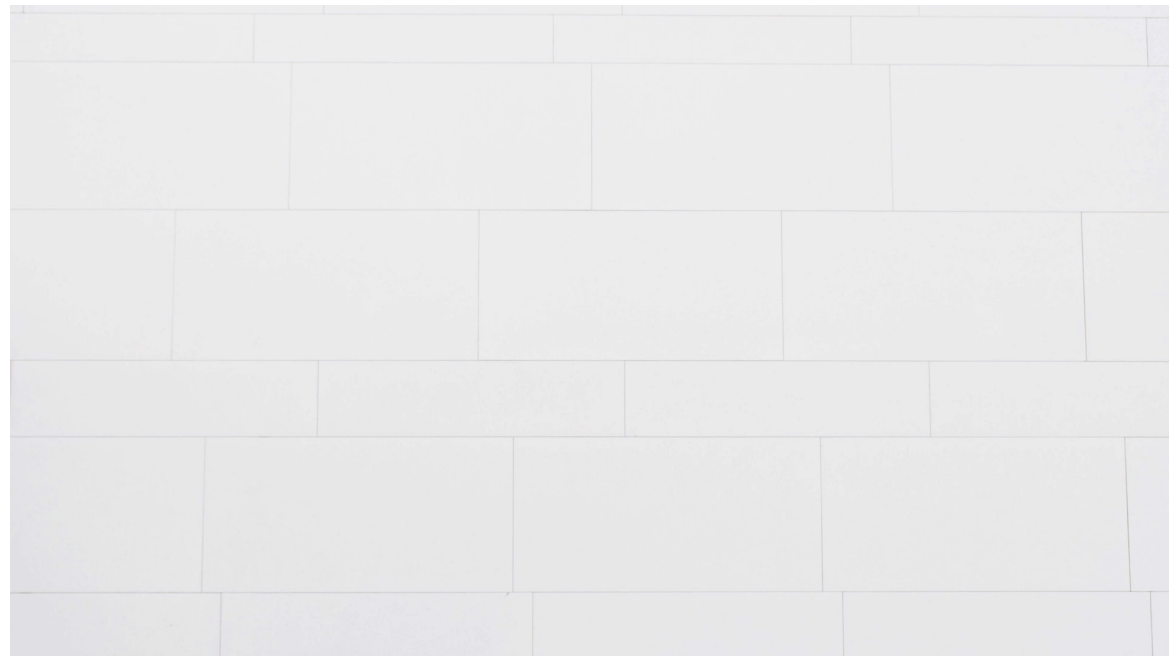


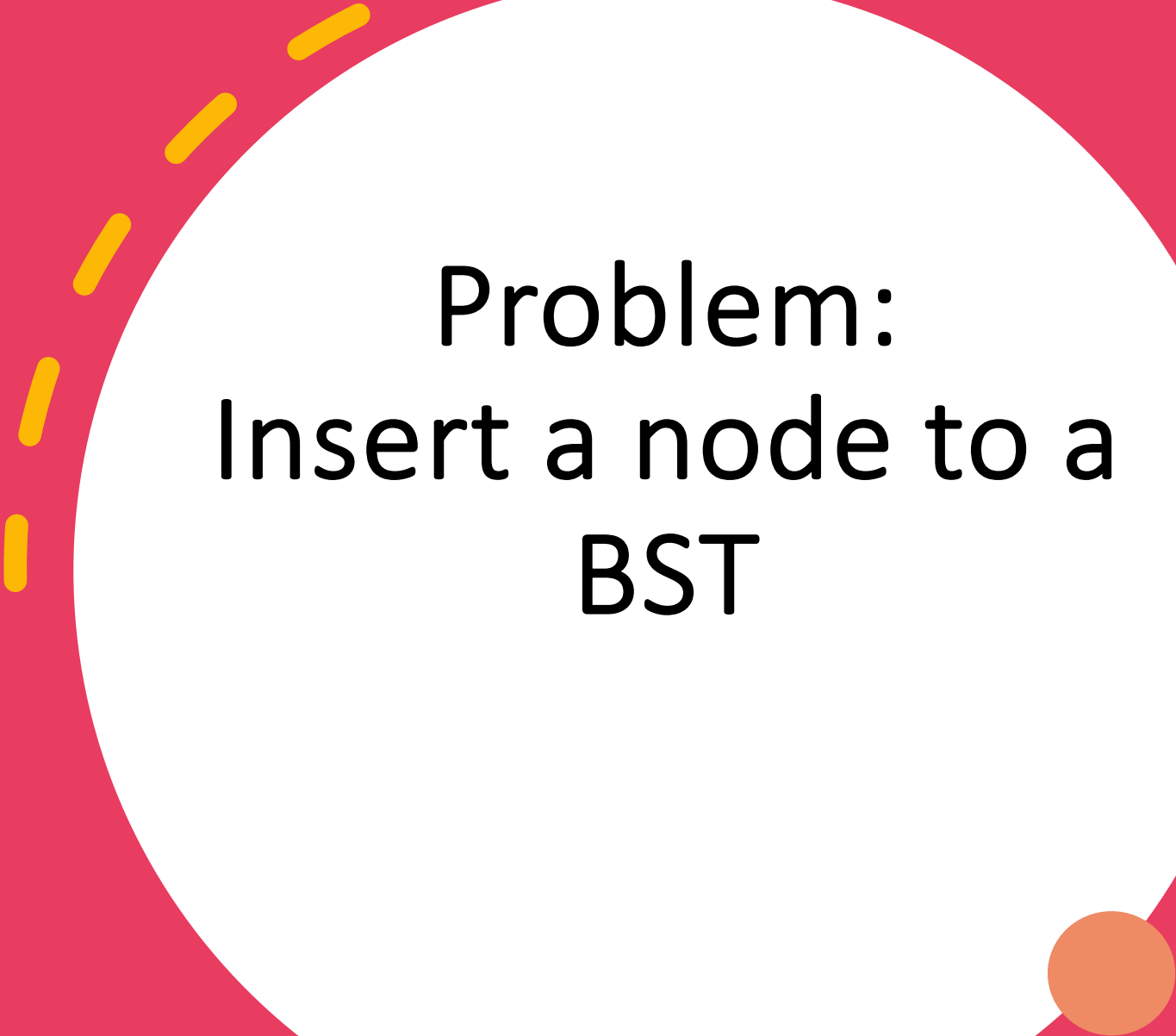
Binary Search Trees



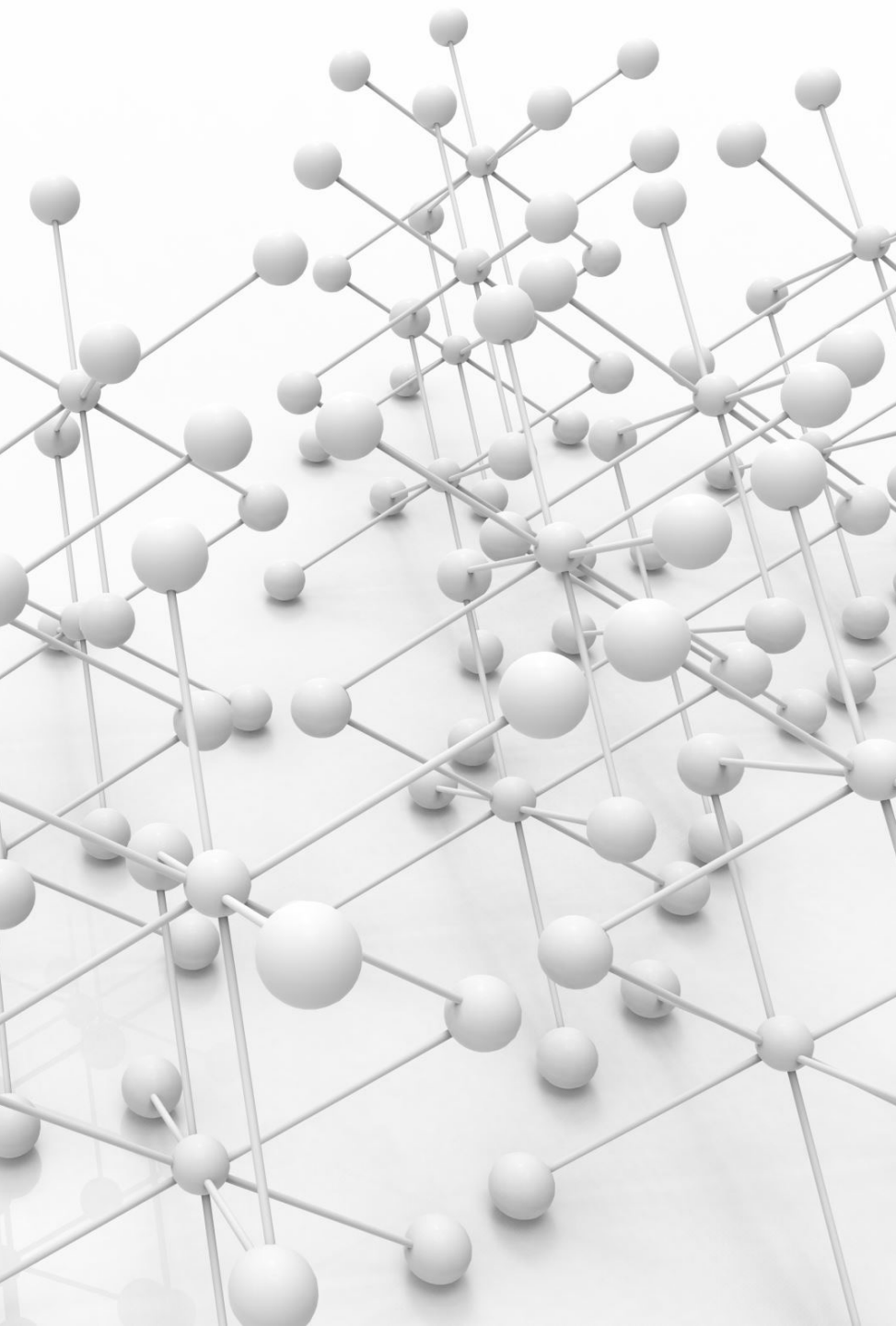


Let's go to
work





Problem:
Insert a node to a
BST



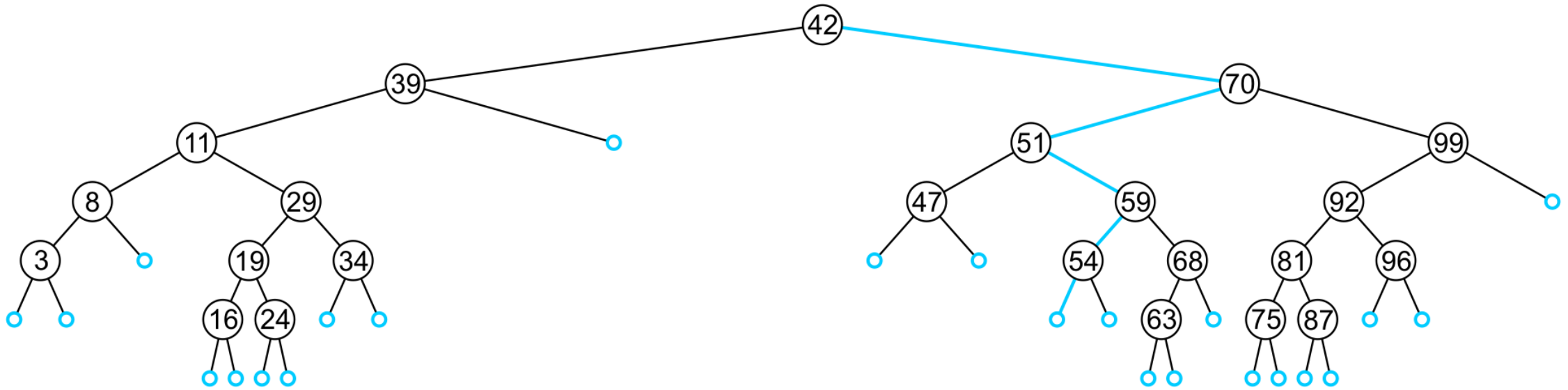
BST insert

Write a recursive algorithm to **insert a node with value into BST.**

- If we find the object already in the tree, we will return
 - The object is already in the binary search tree ,
- Otherwise, we will arrive at an empty node
 - The object will be inserted into that location

The run time is $O(h)$

Space complexity $O(n)$



Insert Example

In inserting the value **52**, we traverse the tree until we reach an empty node

- The left sub-tree of 54 is an empty node



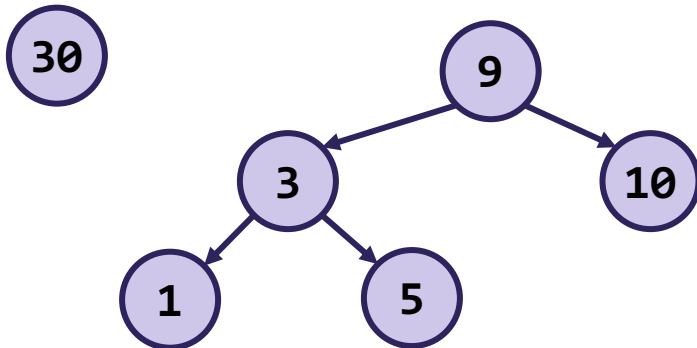
The insert Algorithm → *must be written recursively*

```
def insert(self, node, value) -> None:
    """
    Inserts a node with a value into the BST - don't do anything if value is in BST already
    :param node: the root of the subtree we are traversing
    :param value: the value to insert into the BST
    """

    # empty case
    # val exists
    # right subtree
    # left subtree
    pass
```

```
def test_insert(self):
    bst = BSTree()
    bst.insert(bst.root, 5)
    bst.insert(bst.root, 10)
    bst.insert(bst.root, 1)
    bst.insert(bst.root, 3)
    bst.insert(bst.root, 7)

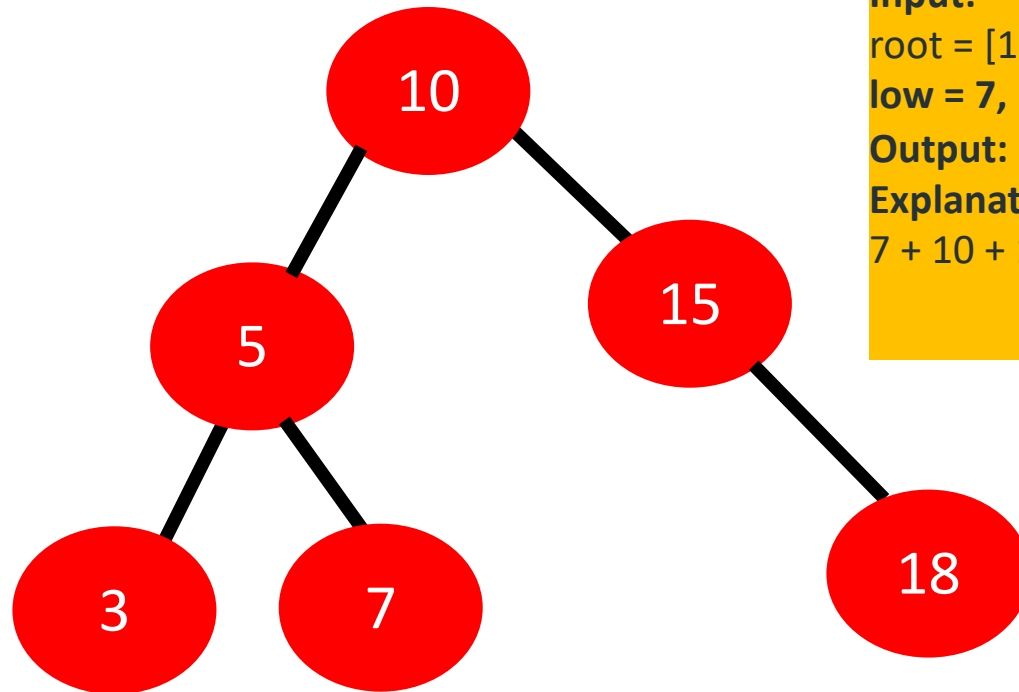
    self.assertEqual(bst.root.value, 5)
    self.assertEqual(bst.root.left.value, 1)
    self.assertEqual(bst.root.left.right.value, 3)
    self.assertEqual(bst.root.right.value, 10)
    self.assertEqual(bst.root.right.left.value, 7)
```



BST Application Problem

Problem: Range Sum of a Binary Search Tree

Given the **root** node of a binary search tree and two integers **low** and **high**, return the sum of values of all nodes with a value in the inclusive range [low, high].



Input:

root = [10,5,15,3,7,null,18],

low = 7, high = 15

Output: 32

Explanation: Nodes 7, 10, and 15 are in the range [7, 15].
7 + 10 + 15 = 32.

Optimum run time complexity $O(N)$

Space $O(N)$

Problem: Range Sum of a Binary Search Tree

Input:

root = [10,5,15,3,7,null,18],

low = 7, high = 15

Output: 32

Explanation: Nodes 7, 10, and 15 are in the range [7, 15].

$7 + 10 + 15 = 32$.

Problem: Range Sum of a Binary Search Tree

Input:

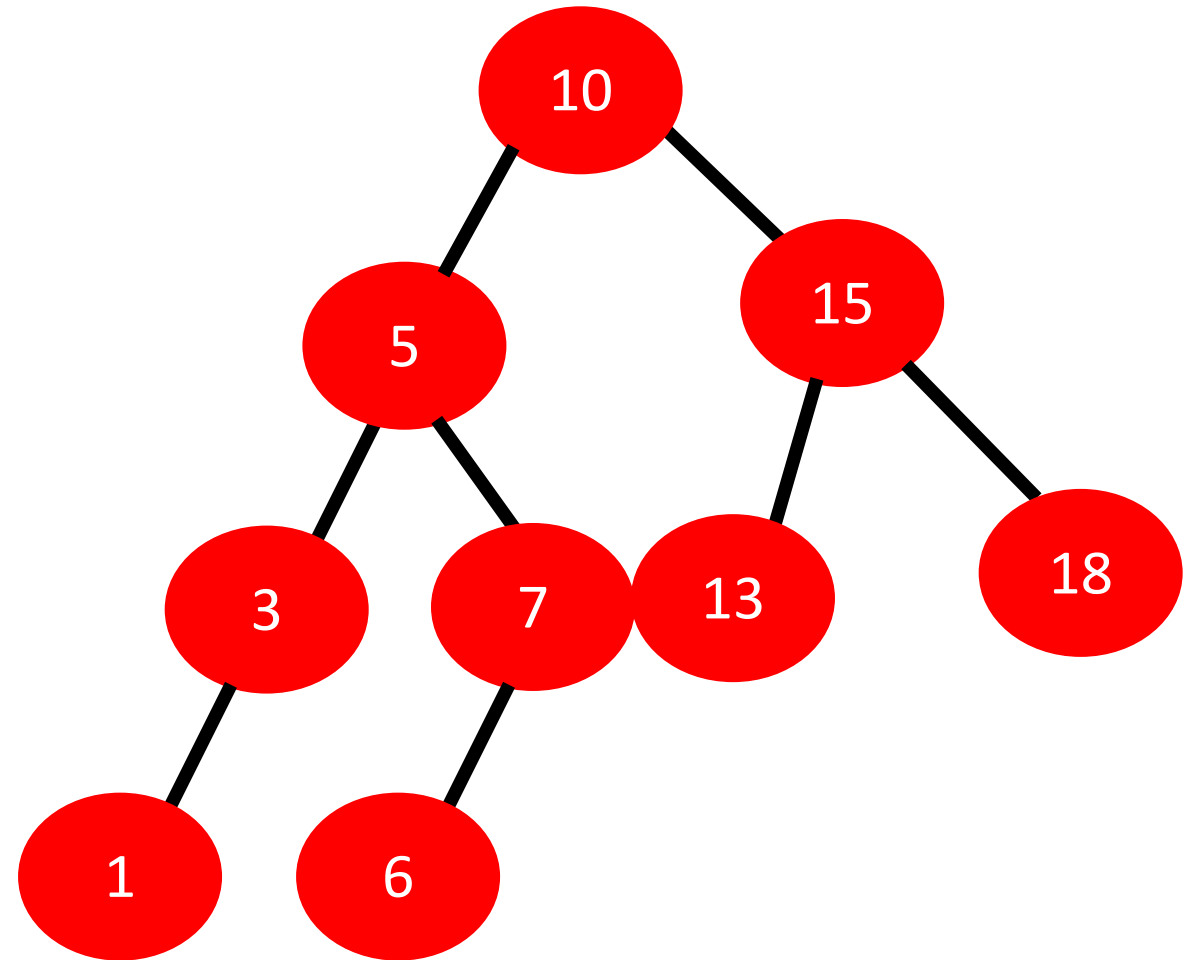
root = [10, 5, 15, 3, 7, 13, 18, 1, None, 6]

low = 6

high = 10

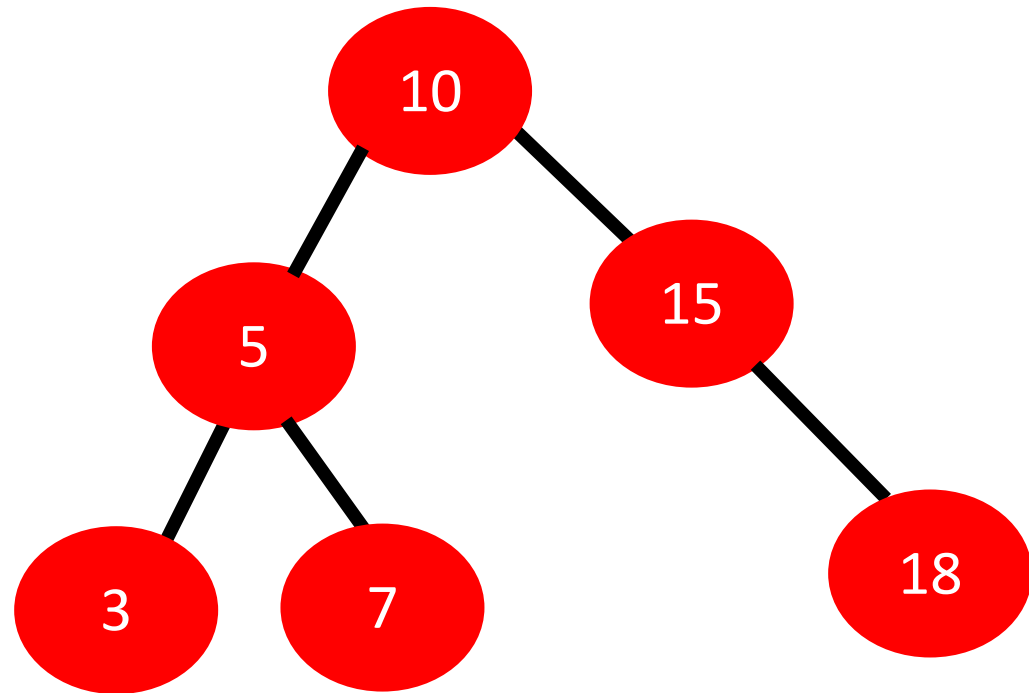
Output: 23

Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. $6 + 7 + 10 = 23$



Approach 1: Using Depth First Approach

Approach 1



Let us try solving this using recursion:

First question:

Is the node.val in the range of low and high?

if yes then add to the sum

Second question:

Is the $low < node.value$? Keep going on left...

Third question:

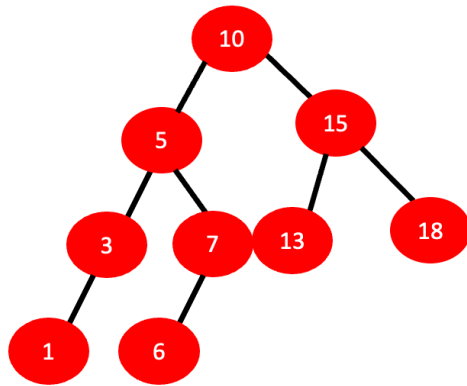
Is the $high > node.value$? Keep going on right



Complete range_sum1

Input:
root = [10, 5, 15, 3, 7, 13, 18, 1, None, 6]
low = 6
high = 10

Output: 23
Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. $6 + 7 + 10 = 23$



```
def range_sum1(self, root, low, high) -> int:
    def dfs(node):
        pass
    dfs(root)
    return self.answer
```

Let us try solving this using recursion:

First question:

Is the node.val in the range of low and high?
if yes then add to the sum

Second question:

Is the low < node.value ? Keep going on left...

Third question:

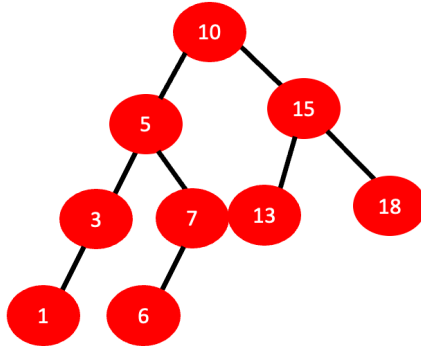
Is the high > node.value ? Keep going on right

Approach 2:

Approach 2

Input:
root = [10, 5, 15, 3, 7, 13, 18, 1, None, 6]
low = 6
high = 10

Output: 23
Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. $6 + 7 + 10 = 23$



We **traverse** the tree using a **breadth first search (not using a queue)**

Store root in a list.

Iterate over the list

pop from the list

check if the value is within the range of L and H,

add to accumulator (answer)

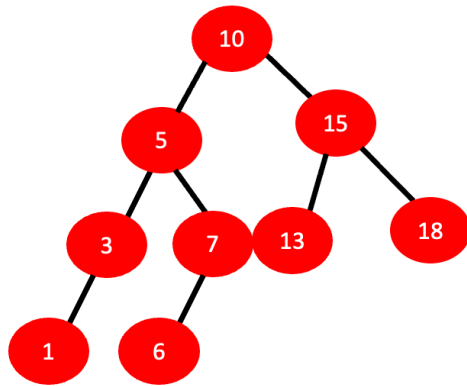
decide to go to left or right based on comparing the L, H to current node val



Approach 2

Input:
root = [10, 5, 15, 3, 7, 13, 18, 1, None, 6]
low = 6
high = 10

Output: 23
Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. $6 + 7 + 10 = 23$



```
# Breadth First Search
```

```
def range_sum2(self, root, low, high) -> int:  
    pass
```

We **traverse** the tree using a **breadth first search (not using a queue)**

Store root in a list.

Iterate over the list

- pop from the list

- check if the value is within the range of L and H,

 - add to accumulator (answer)

- decide to go to left or right based on comparing the L, H to current node val

Approach 3:
Deque
No recursion



Approach 3

Input:

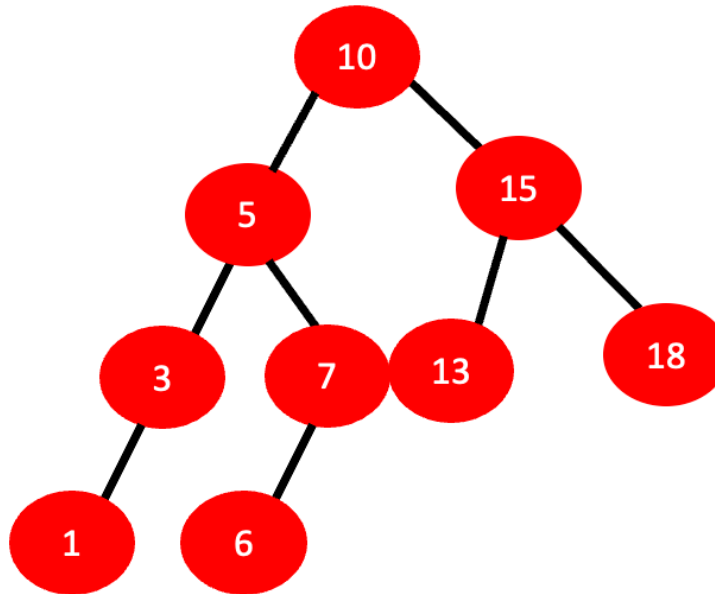
root = [10, 5, 15, 3, 7, 13, 18, 1, None, 6]

low = 6

high = 10

Output: 23

Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. $6 + 7 + 10 = 23$



Use a deque?
Just Like Approach 2, BFS



Complete the code, does not have to be written *recursively*...

Input:

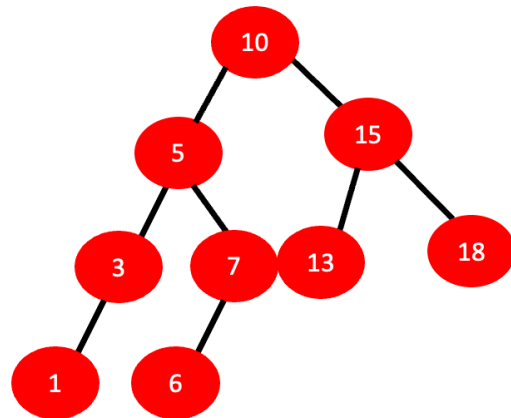
root = [10, 5, 15, 3, 7, 13, 18, 1, None, 6]

low = 6

high = 10

Output: 23

Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. $6 + 7 + 10 = 23$



```
def range_sum3(self, root, low, high) -> int:  
    rs_deque = deque()  
    pass
```



Another BST application
problem,
related to deep learning
😊

Find the closest Value in a BST

Complete the `find_closest` function, this function takes a BST and a target value and *returns the closest value* to that target value contained in the BST.

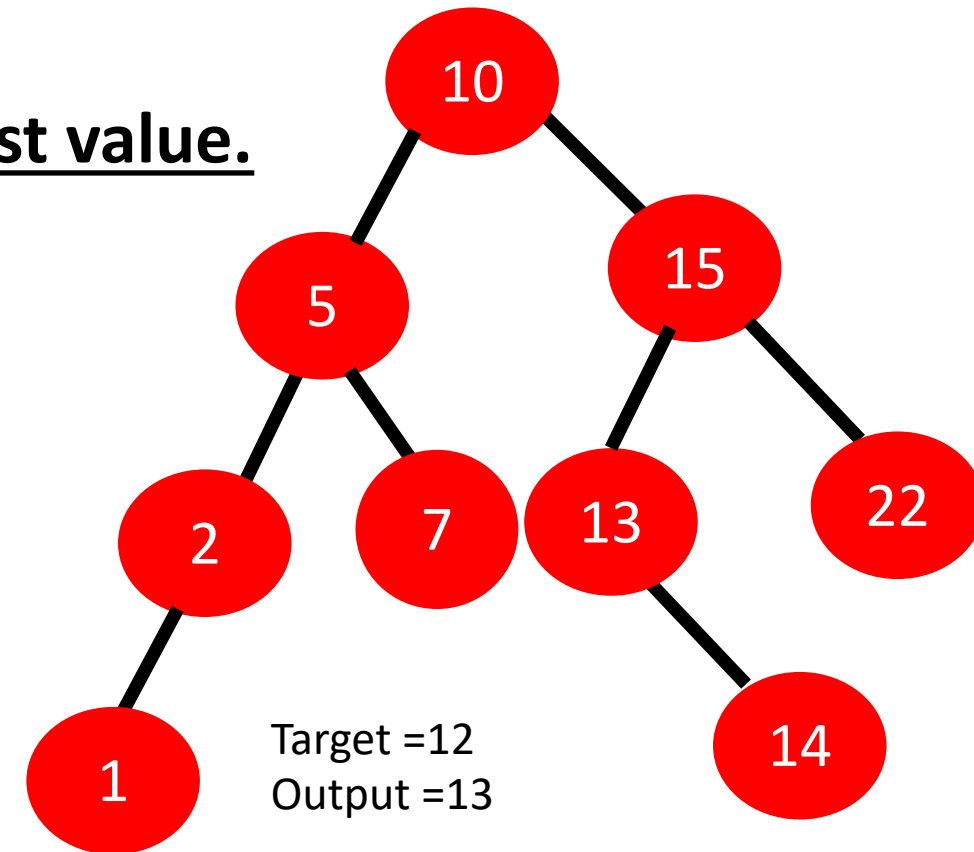
You can assume that there will only be one closest value.

Optimum Run time complexity:

Average $O(\log n)$

Worst Case $O(n)$

Space Complexity $O(1)$



Find the closest Value in a BST

How to approach?

Think of traversing the BST node.

Keep track of the node with the value closest to the target.

Think of using Absolute value....

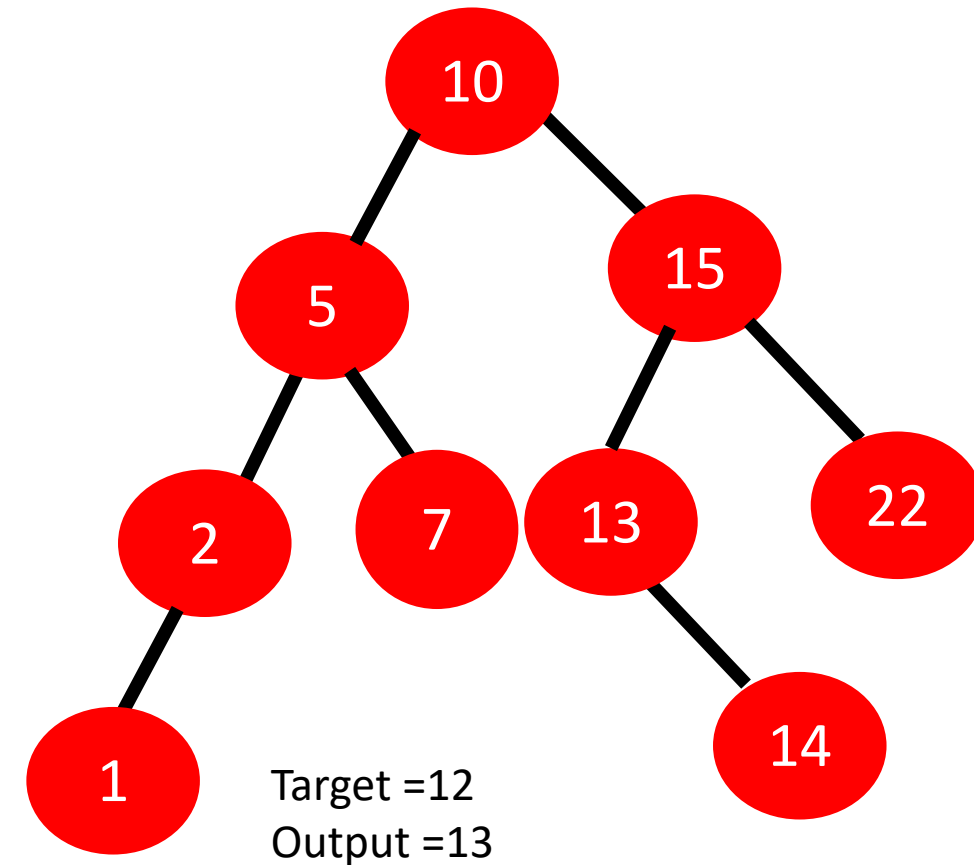
Try calculating the difference

Between the node's val and the target val ..

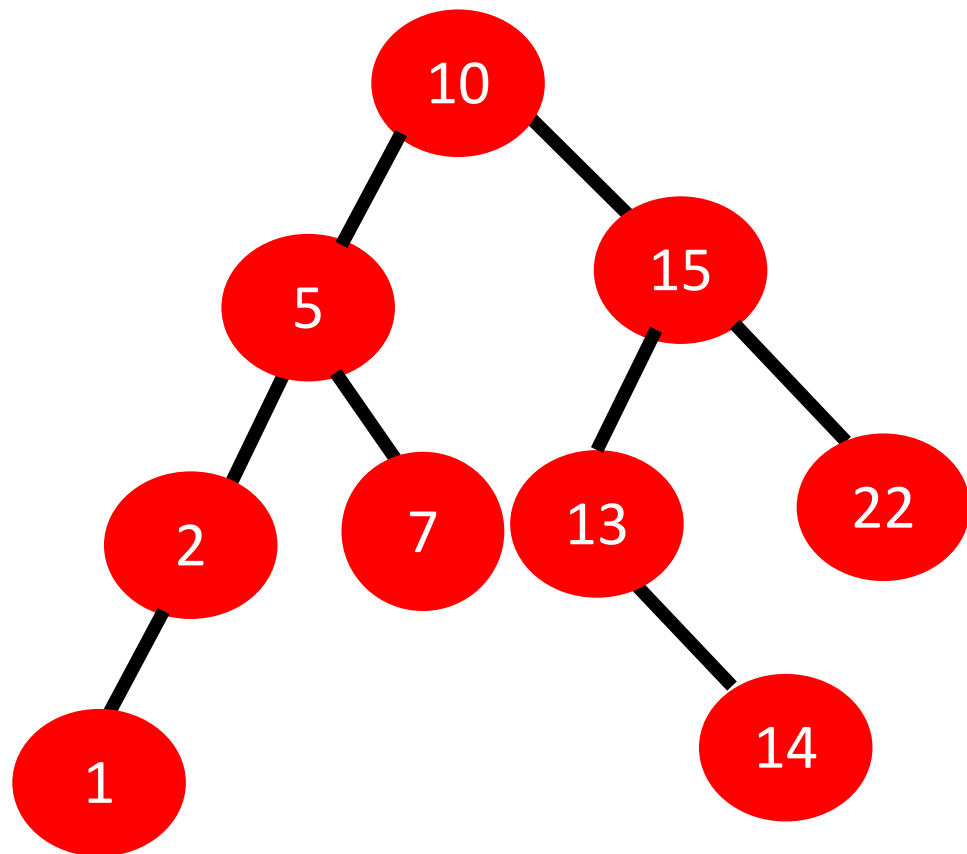
It should allow you to check if the that node is closer than the current closest one

Remember your variant!

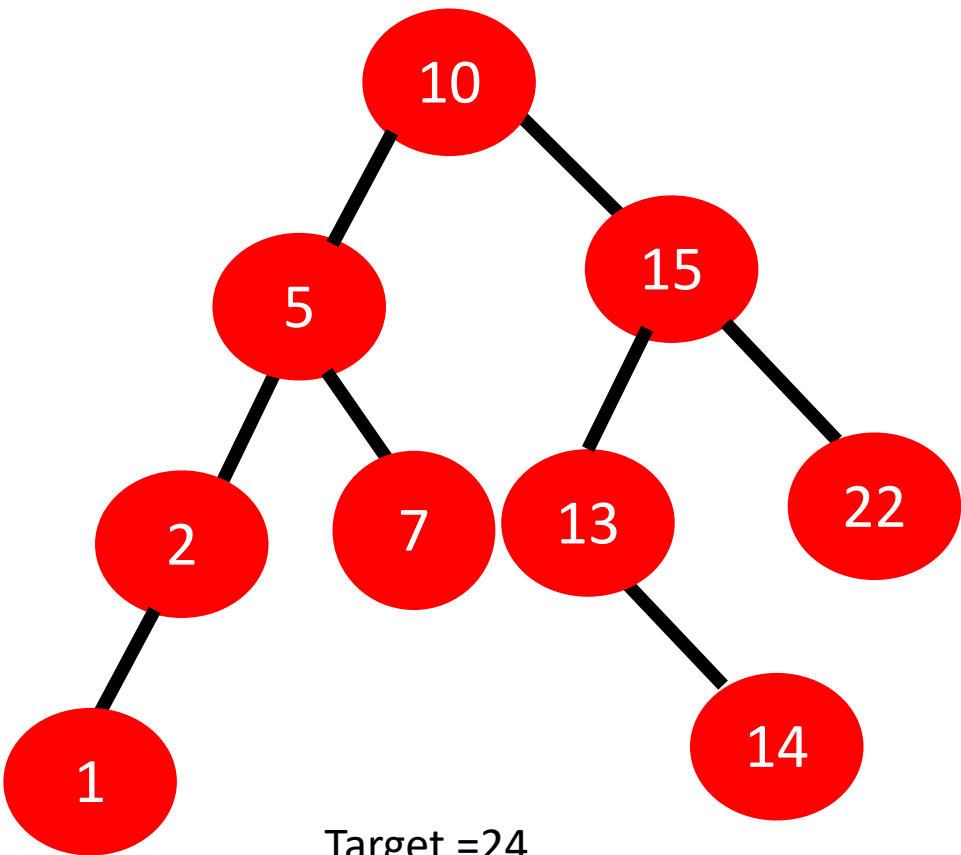
Use the BST variant to decide which side of your BST has the closest value.



Find the closest Value in a BST



Target =15
Output =15



Target =24
Output =22

Approach 1: Using Recursion



Solution 1

How to approach?

Think of traversing the BST node.

Keep track of the node with the value closest to the target.

Think of using Absolute value....

Try calculating the difference

Between the node's val and the target val ..

It should allow you to check if the that node is closer than the current closest one

Remember your variant!

Use the BST variant to decide which side of your BST has the closest value.

```
def find_closest(self, root, target) -> int:
    pass

def find_closest_helper(self, root, target, closest) -> int:
    pass
```

You can write `find_closest_helper` as your recursive function and simply call from `find_closest`.

Approach 2: Using iteration

Solution 2

```
def find_closest_iter(self, root, target) -> int:  
    pass
```