AVL TREES

BALANCED TREES

Motivation

- **linked lists**: quite easy to implement
  - Stores lots of pointers
    - **O(N)** search operation time complexity

- **binary search trees**: we came to to conclusion that **O(N)** searh complexity
  - can be reduced to **O(logN)** time complexity
    - But if the tree is unbalanced : these operations will become
      - slower and slower

- **balanced binary trees**: AVL trees or red-black trees
  - They are guaranteed to be balanced
    - Why is it good? **O(logN)** is guaranteed !!!

Motivation

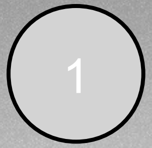
Construct a BST from a sorted array

[1,2,3,4]

Motivation

Construct a BST from a sorted array

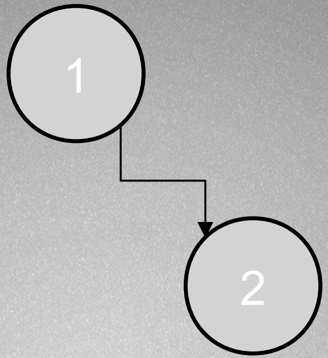
[**1**,2,3,4]



Motivation

Construct a BST from a sorted array

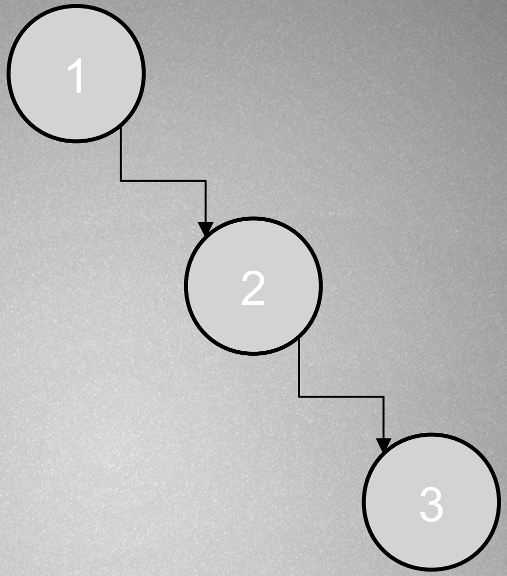
[1,**2**,3,4]



Motivation

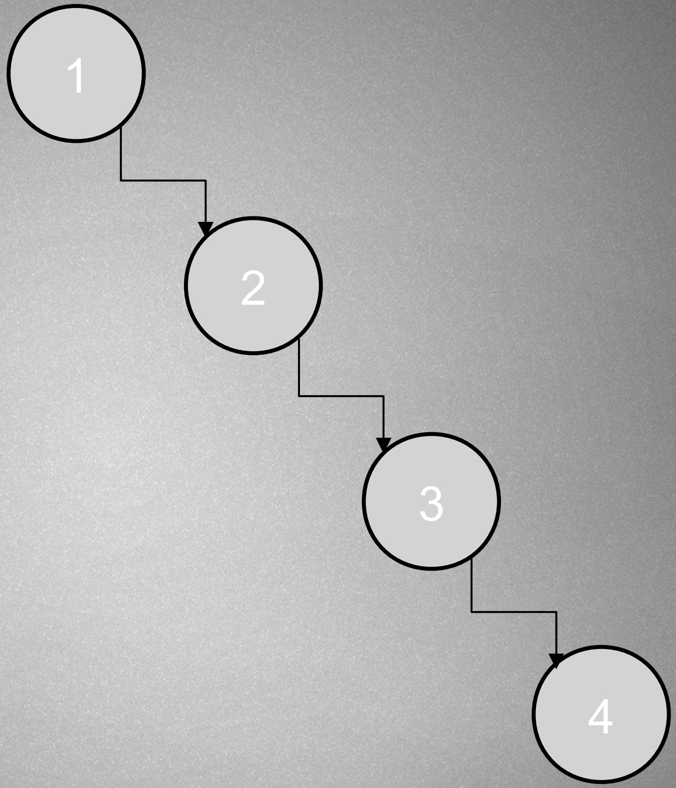
Construct a BST from a sorted array

[1,2,**3**,4]



Motivation

Construct a BST from a sorted array
[1,2,3,**4**]



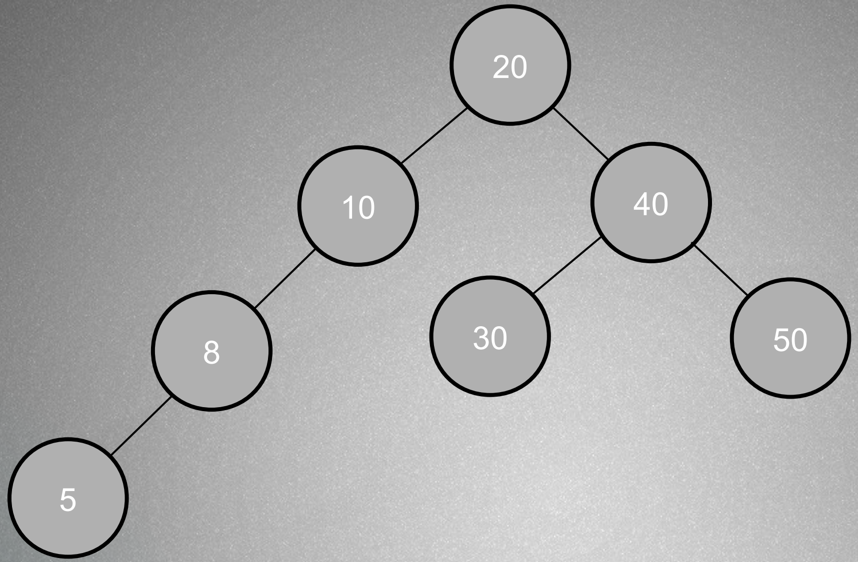
Conclusion: if we construct a binary search tree from a sorted array, we end up with a linked list !!!
O(logN) reduced to **O(N)**

AVL TREES

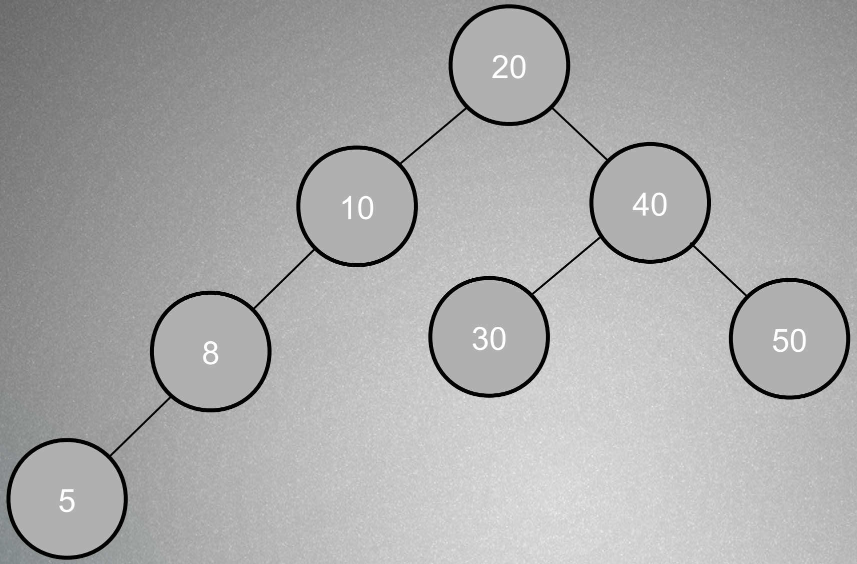
BALANCED TREES

- The running time of BST operations depends on the height of the binary search tree: we should keep the tree balanced in order to get the best performance
- Thats why AVL trees came to be
- 1962: invented by two russian computer scientist
- In an AVL tree, the heights of the two child subtrees of any node differ by at most one
- Another solution to the problem is a red-black trees
- AVL trees are faster than red-black trees because they are more rigidly balanced BUT needs more work
- Operating systems relies heavily on these data structures !!!

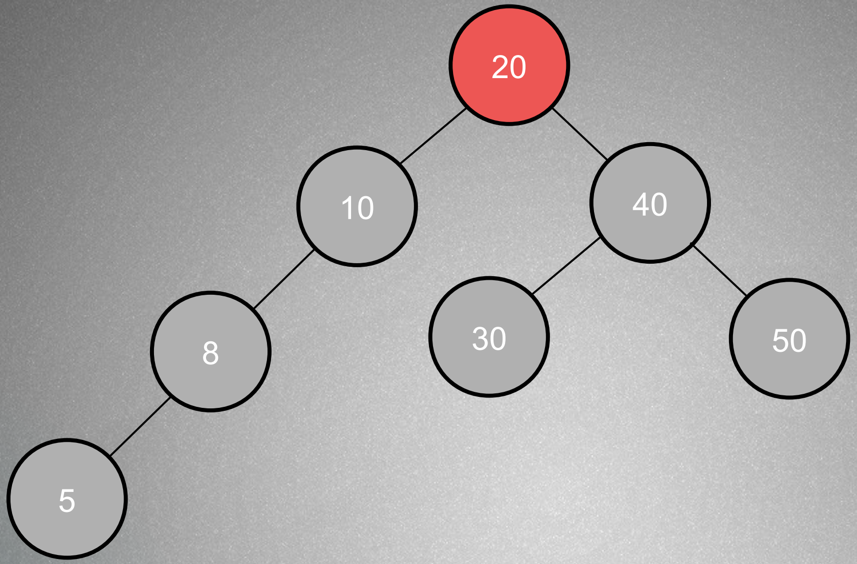
- Most of the operations are the same as we have seen for binary search trees
- Every node can have at most 2 children: the leftChild is smaller, the rightChild is greater than the parent node
- The insertion operation is the same BUT on every insertion we have to check whether the tree is unbalanced or not
- Deletion operation is the same
- Maximum / minimum finding operations are the same as well !!!


20
10
40
8
30
50
5

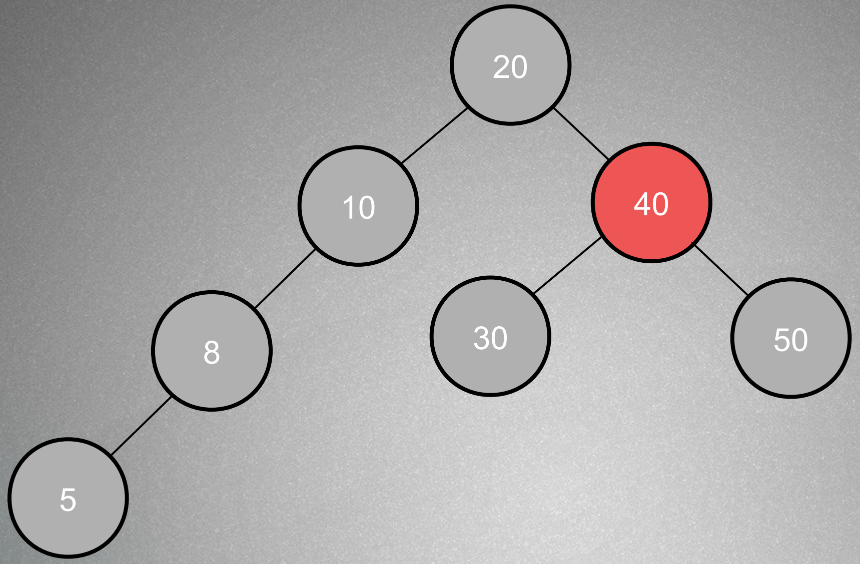
```
balancedTree.find(30);
```



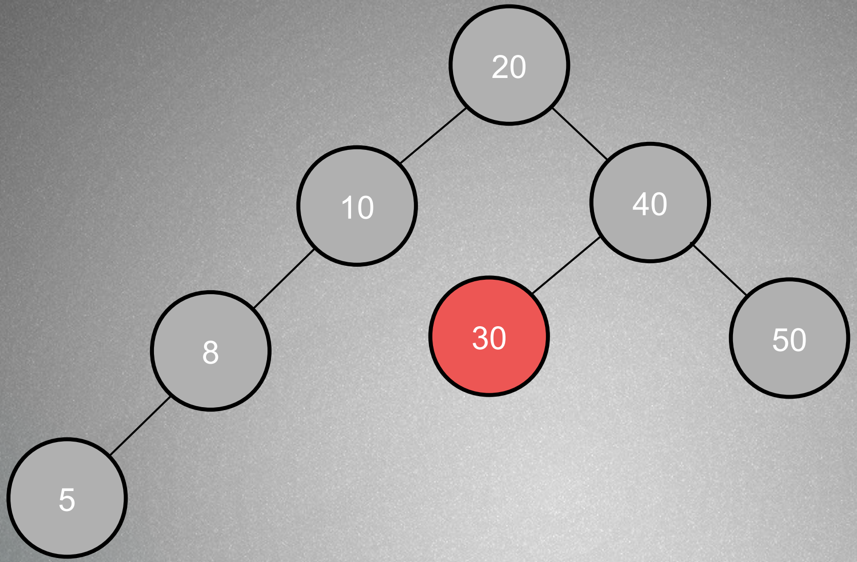
balancedTree.find(30);

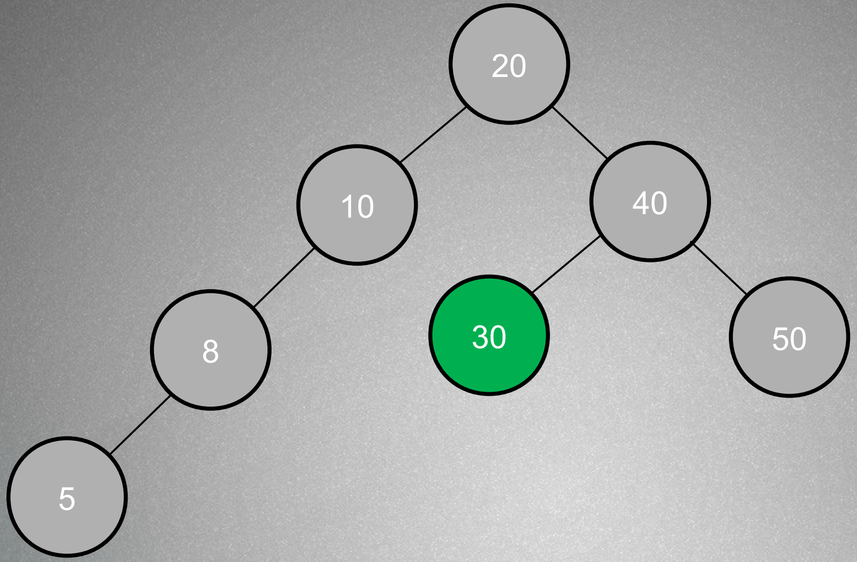


balancedTree.find(30);

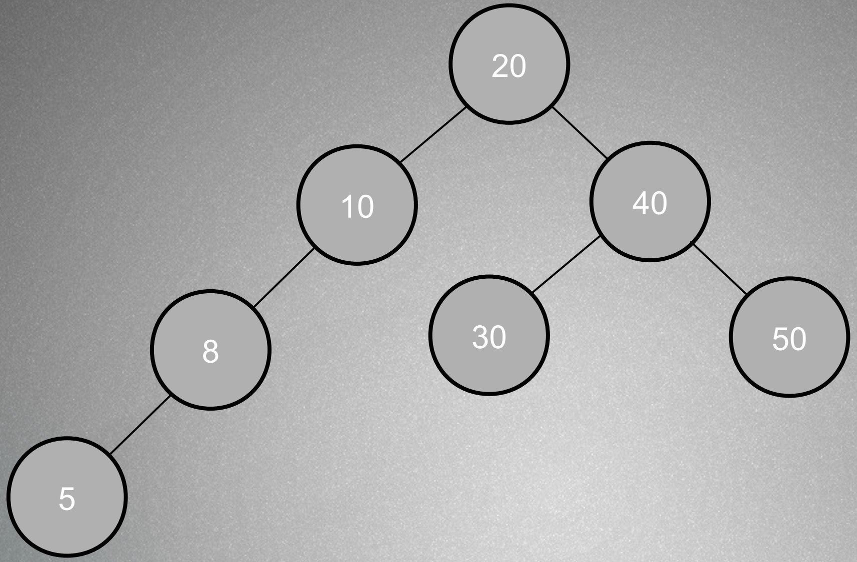


balancedTree.find(30);

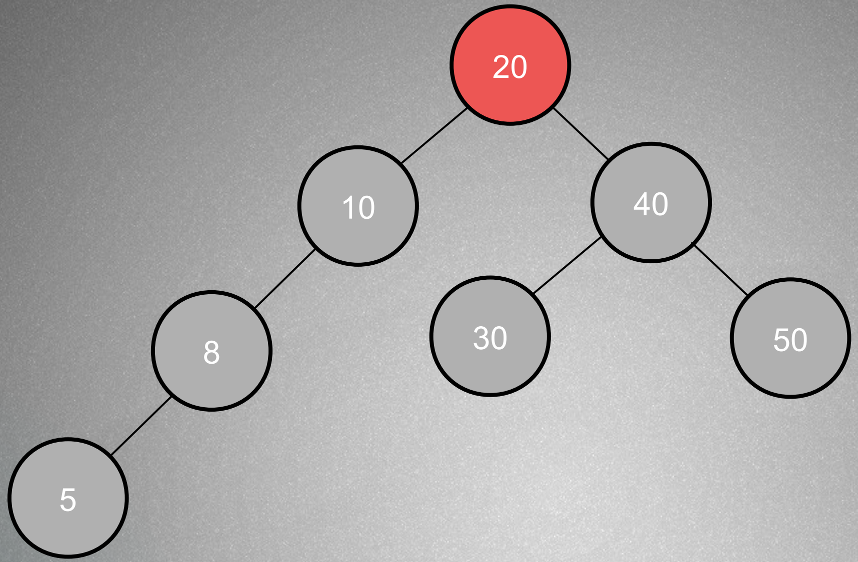



20
10
40
8
30
50
5

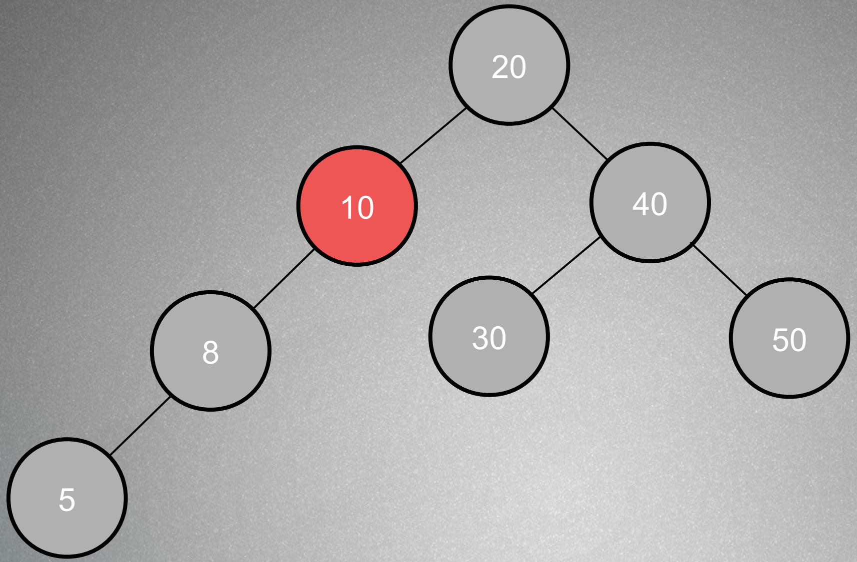
findMin();



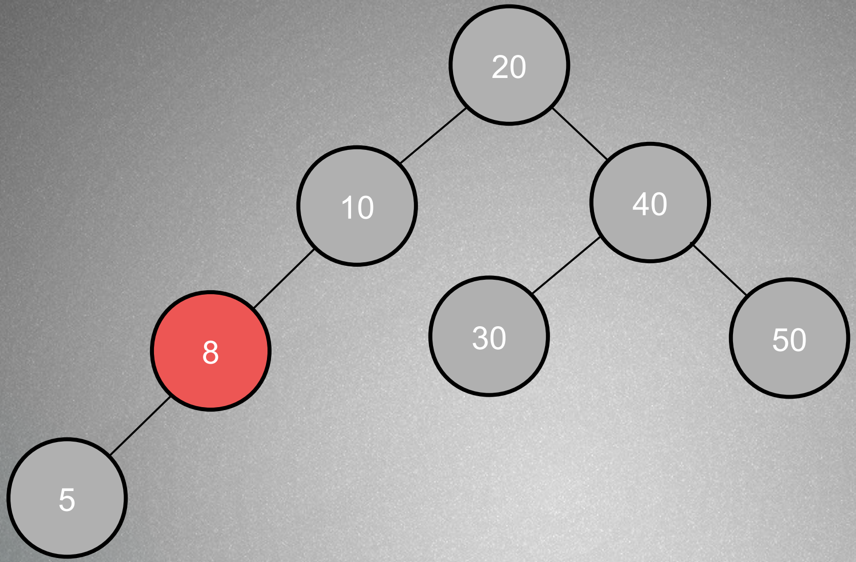
findMin();



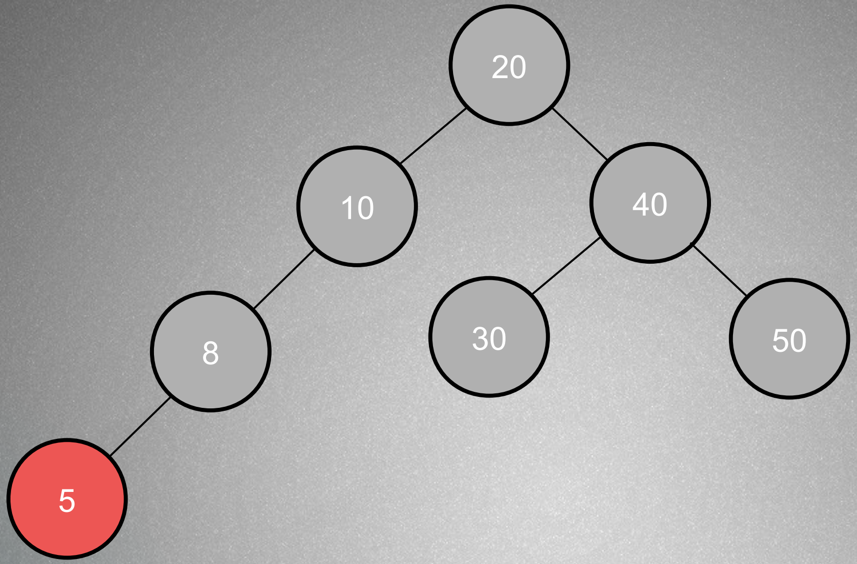
findMin();



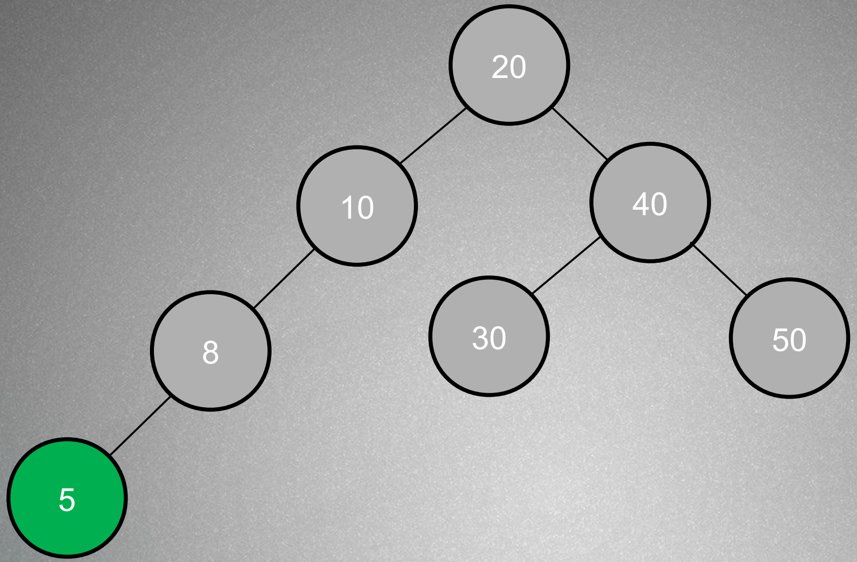
findMin();



findMin();

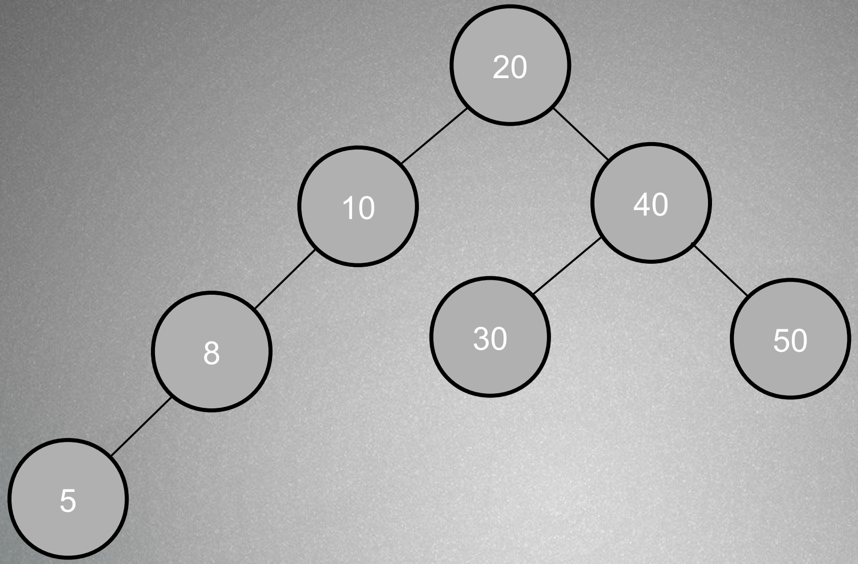


findMin();

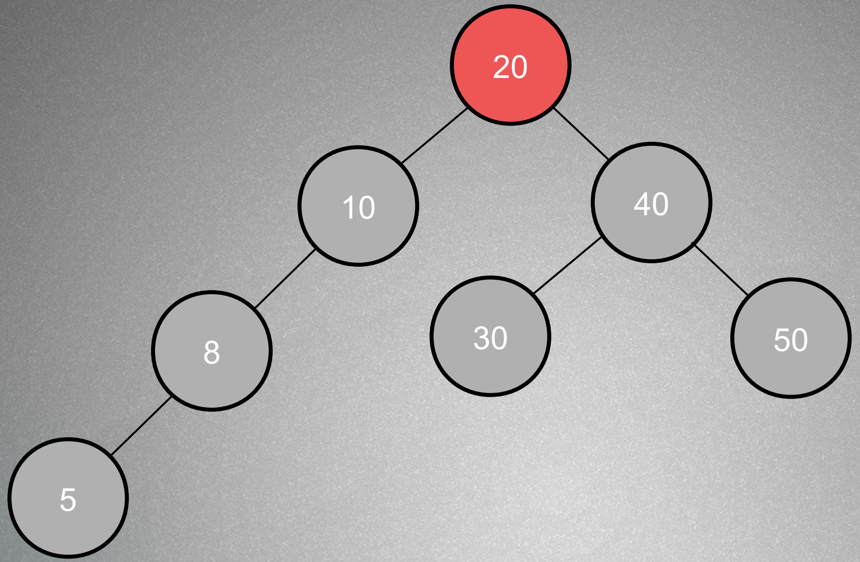


The minimum value in the tree: 5

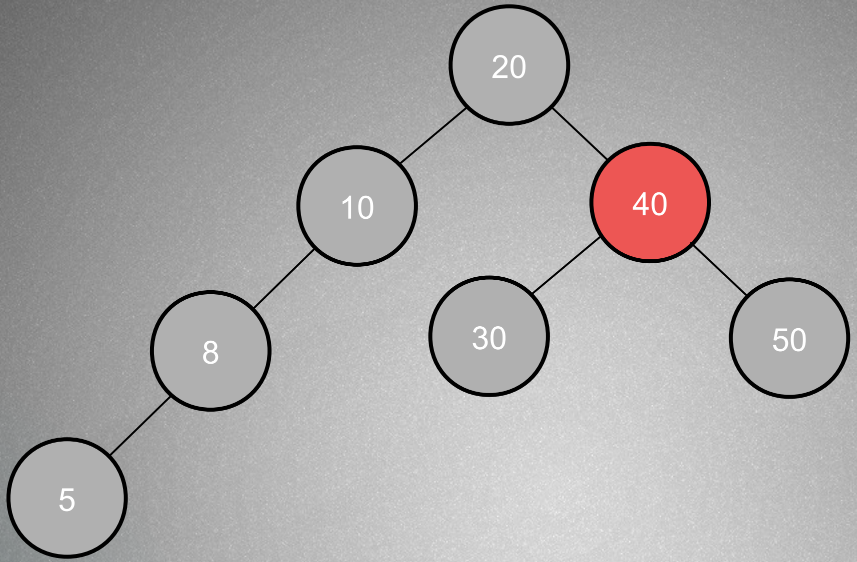
findMax();



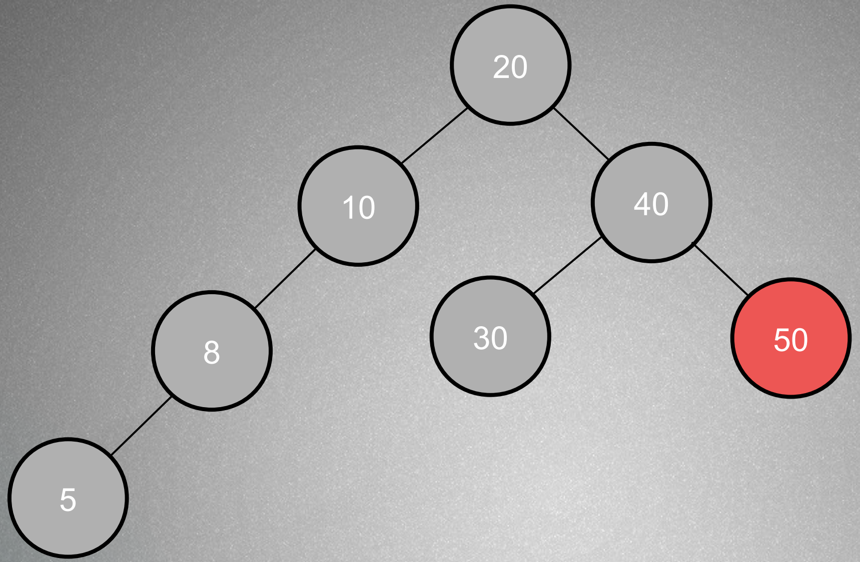
findMax();



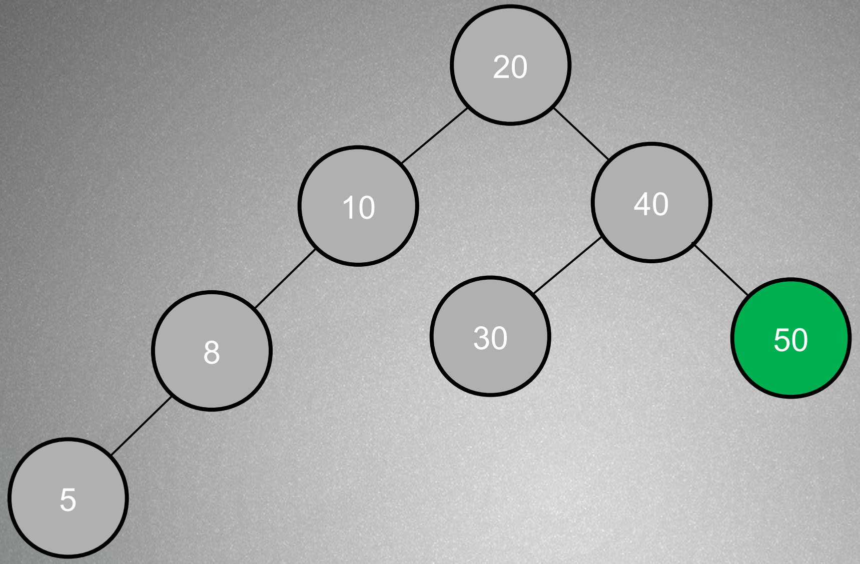
findMax();



findMax();



findMax();



The maximum value in the tree: 50

Binary search trees

	Average case	Worst case
Space	O(n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

Balanced trees

	Average case	Worst case
Space	O(n)	O(n)
Insert	O(log n)	**O(log n)**
Delete	O(log n)	**O(log n)**
Search	O(log n)	**O(log n)**

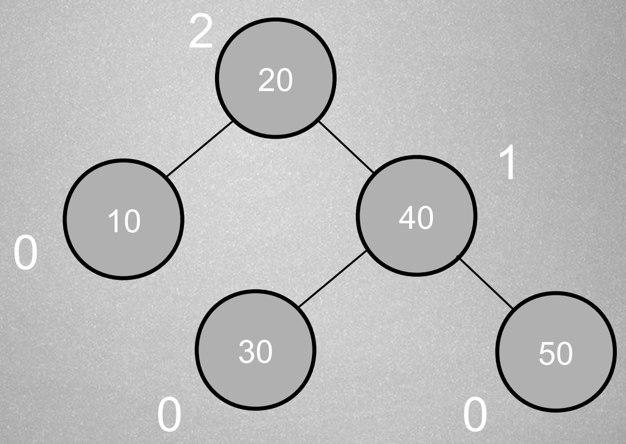
AVL TREES

BALANCED TREES

Height of a node: length of the longest path from it to a leaf

We can use recursion to calculate it:

height = max(leftChild.height(),rightChild.height())+1 !!!



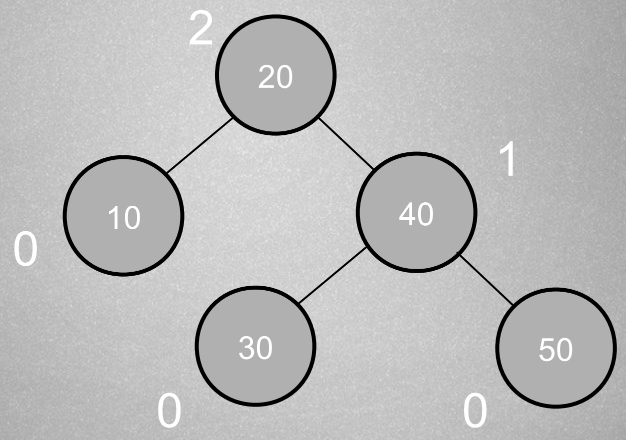
The leaf nodes have NULL children: we consider the height to be -1 for NULLs !!!

AVL algorithm uses heights of nodes, we want the heights as small as possible: we store the height parameters → if it gets high, we fix it

Height of a node: length of the longest path from it to a leaf

We can use recursion to calculate it:

height = max(leftChild.height(),rightChild.height())+1 !!!

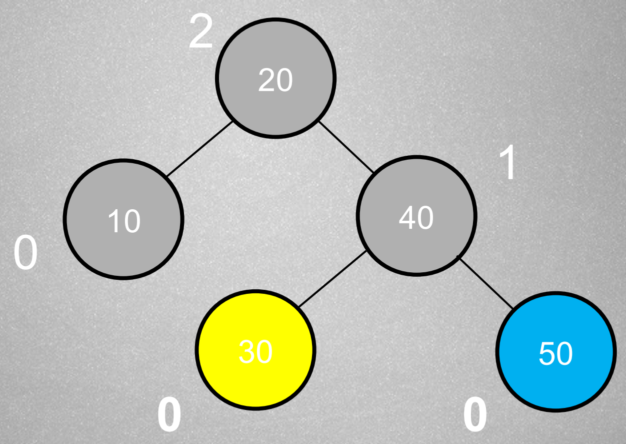


All subtrees height parameter does not differ more than 1 !!!

Height of a node: length of the longest path from it to a leaf

We can use recursion to calculate it:

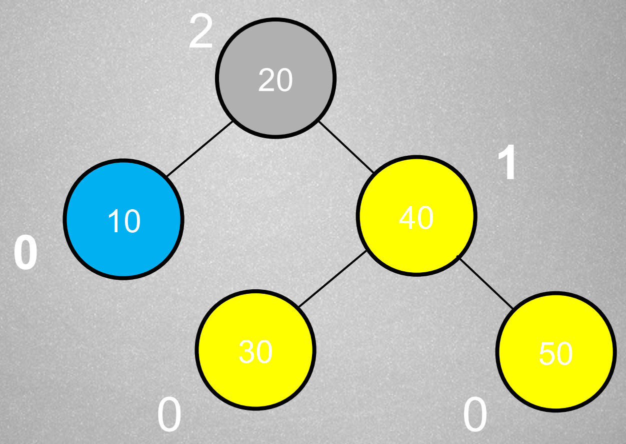
height = max(leftChild.height(),rightChild.height())+1 !!!



Height of a node: length of the longest path from it to a leaf

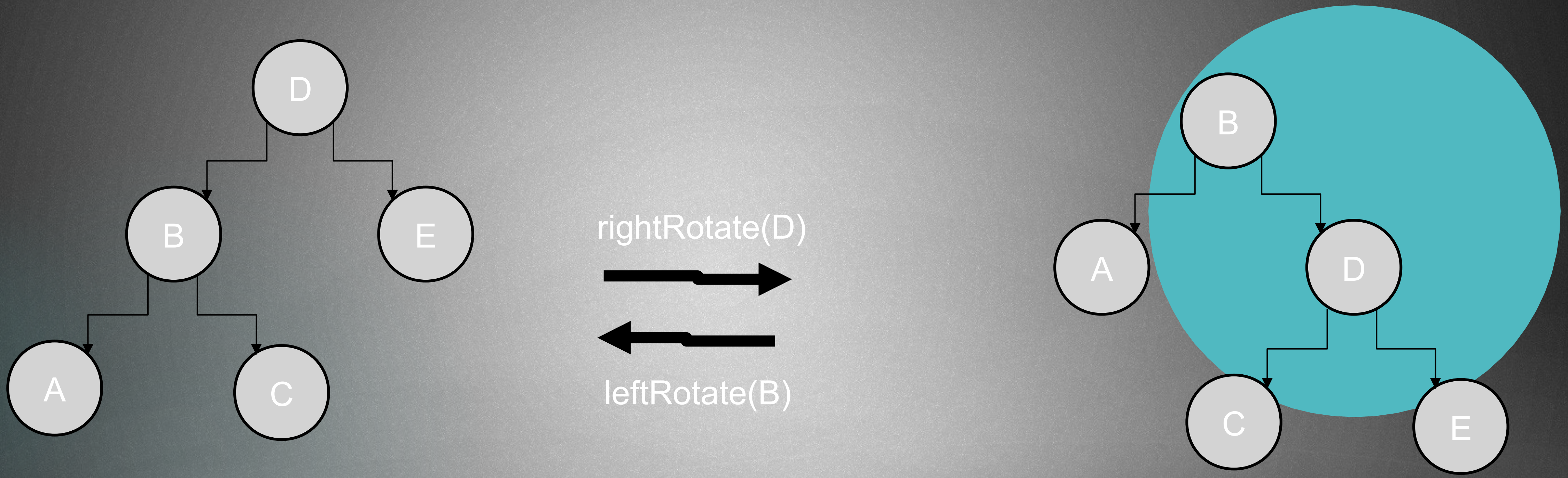
We can use recursion to calculate it:

height = max(leftChild.height(),rightChild.height())+1 !!!



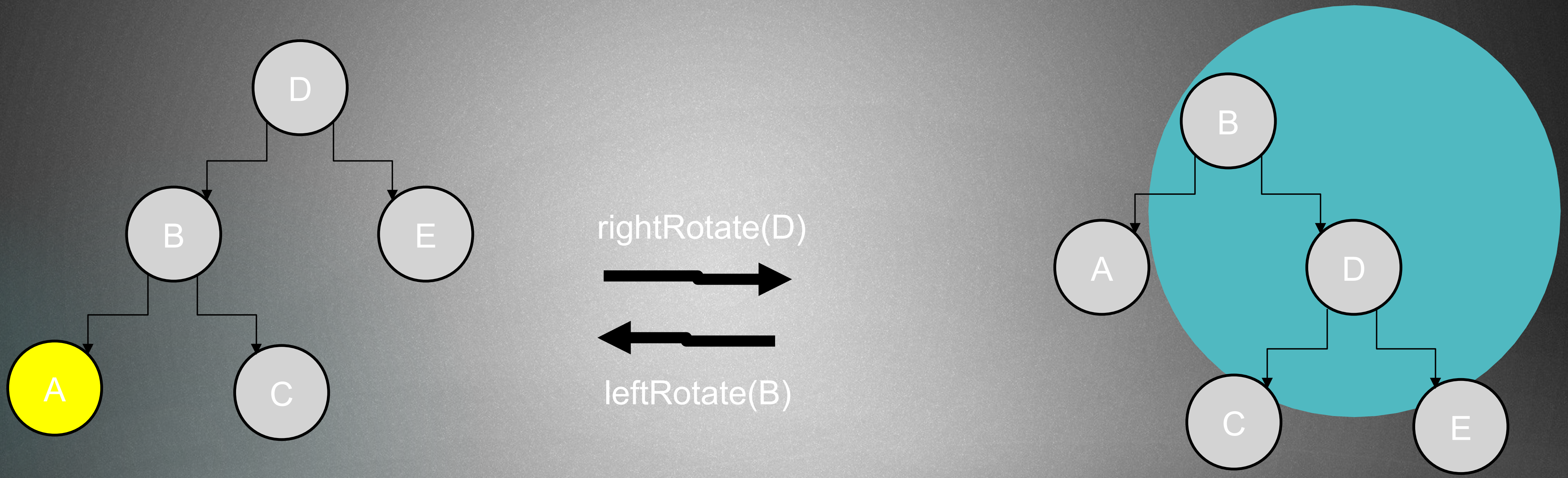
- AVL tree requires the heights of left and right child of every node to differ at most +1 or -1 !!!
- | height(leftSubtree) – height(rightSubtree) | < 1
- So for a balanced tree the height is in the range [-1;+1]
- We can maintain this property in O(logN) time which is quite fast !!!
- Insertion:
  - 1.) a simple BST insertion according to the keys
  - 2.) fix the AVL property on each insertion from insertion upward
- There may be several violations of AVL property from the inserted node up to the root!!!
- We have to check them all

Rotations



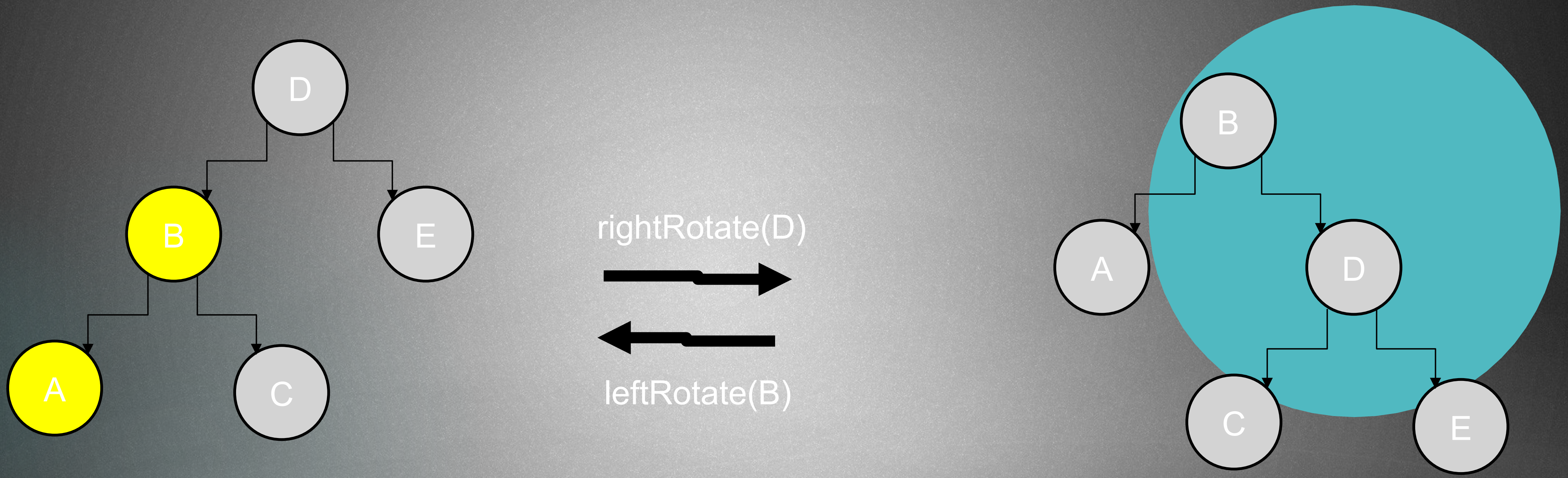
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



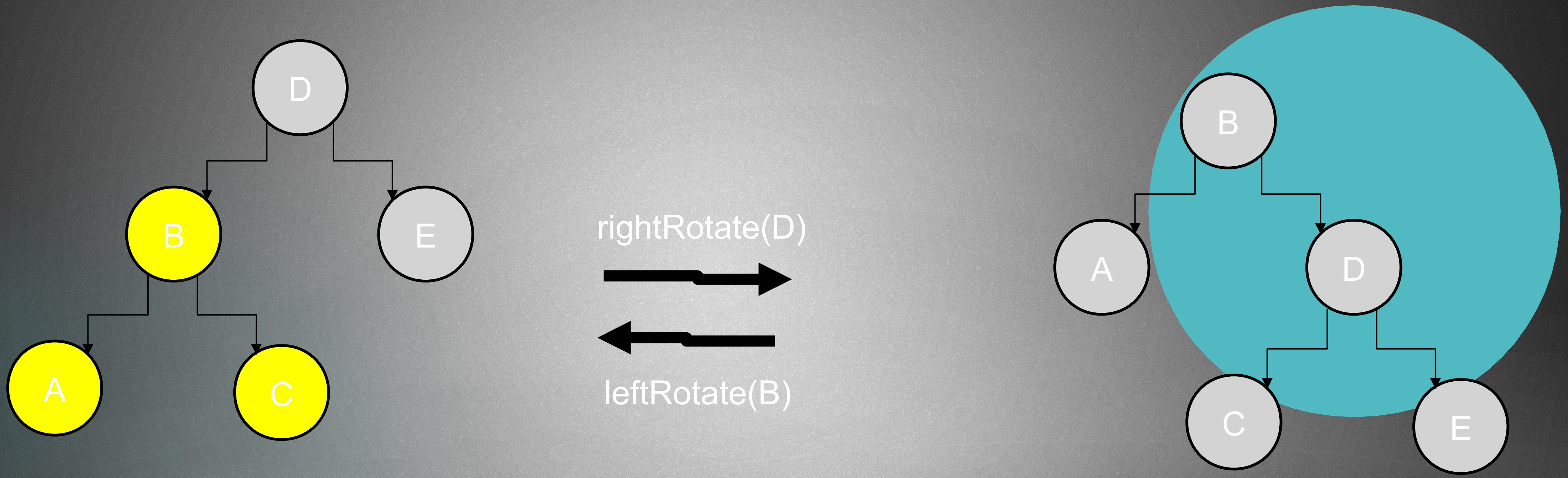
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



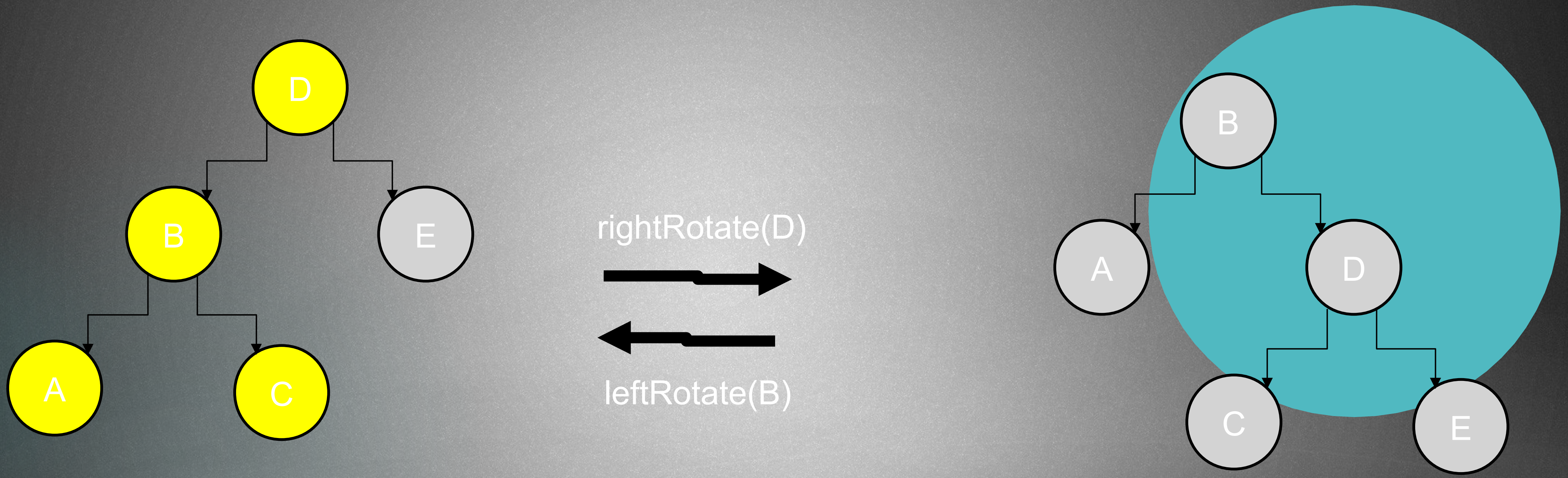
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



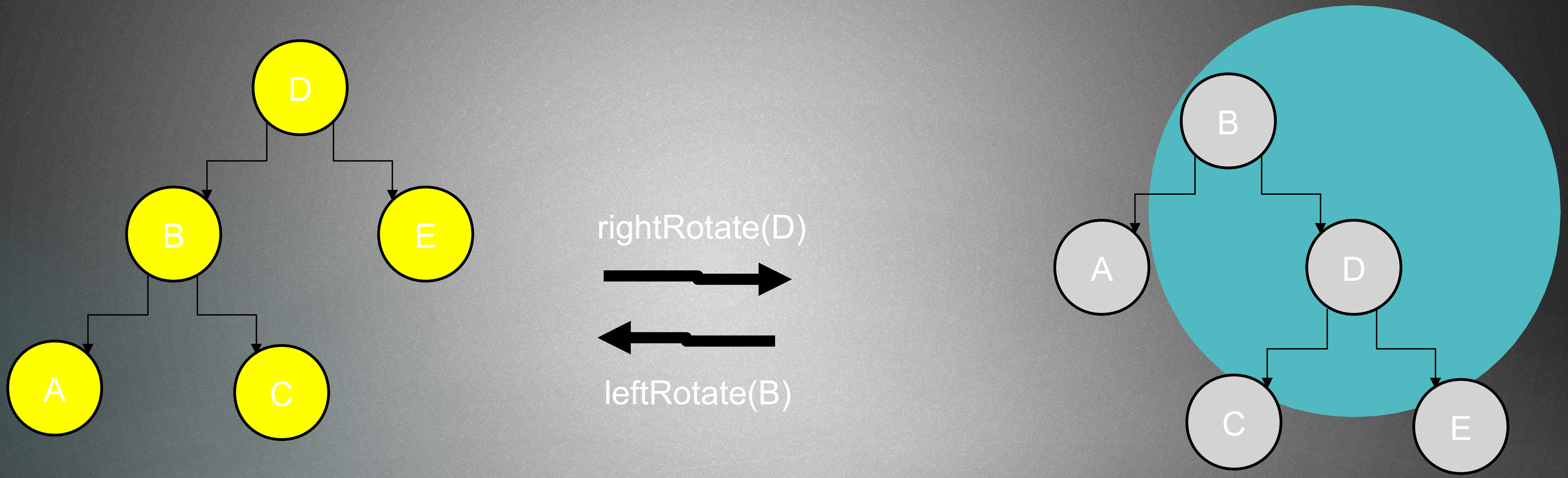
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



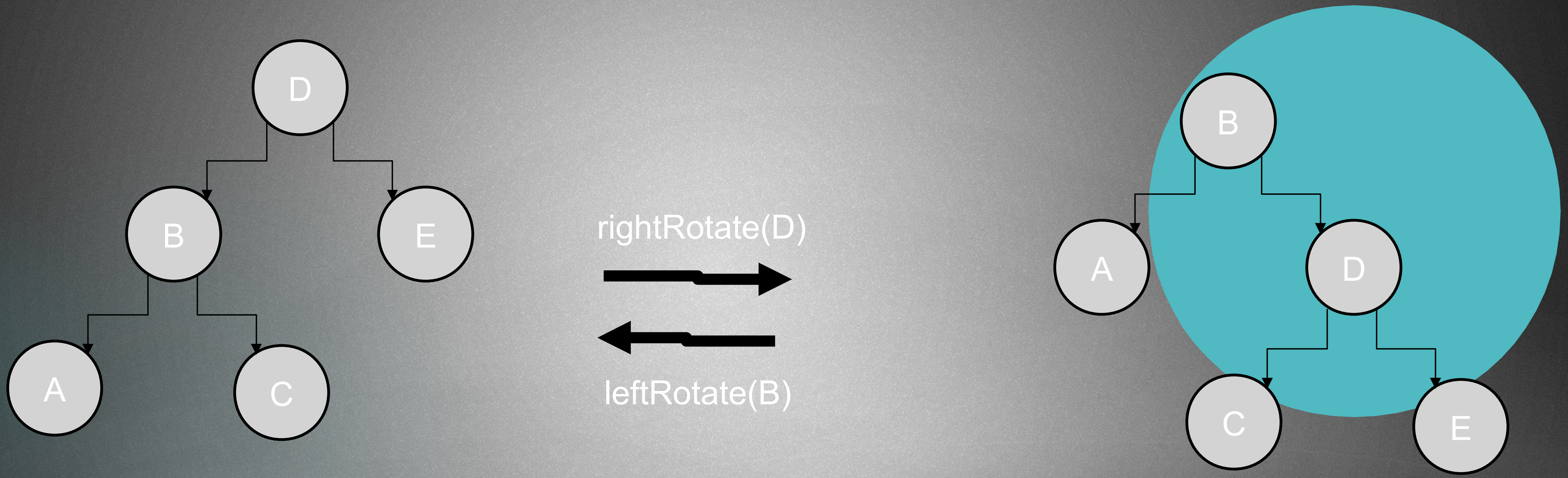
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



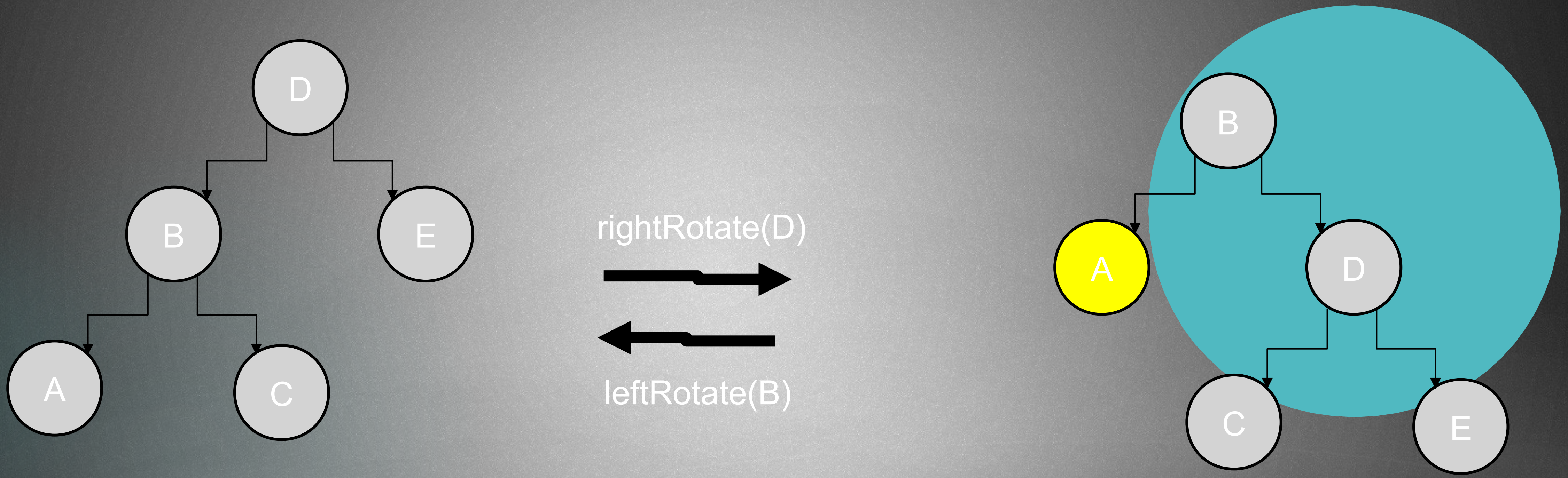
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



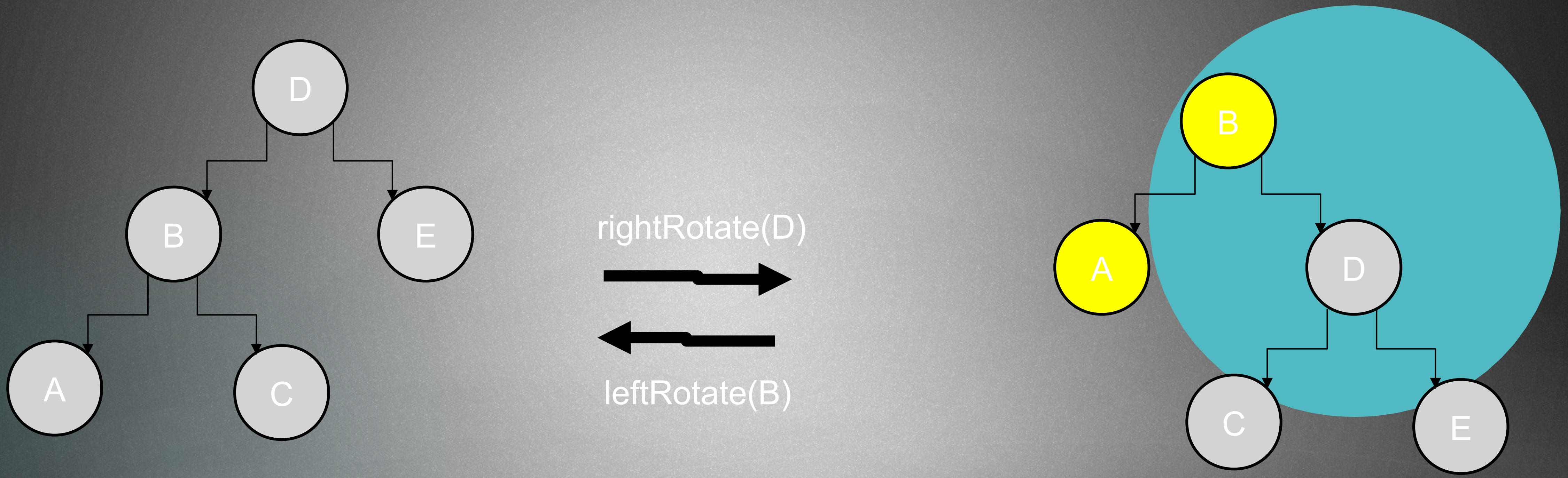
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



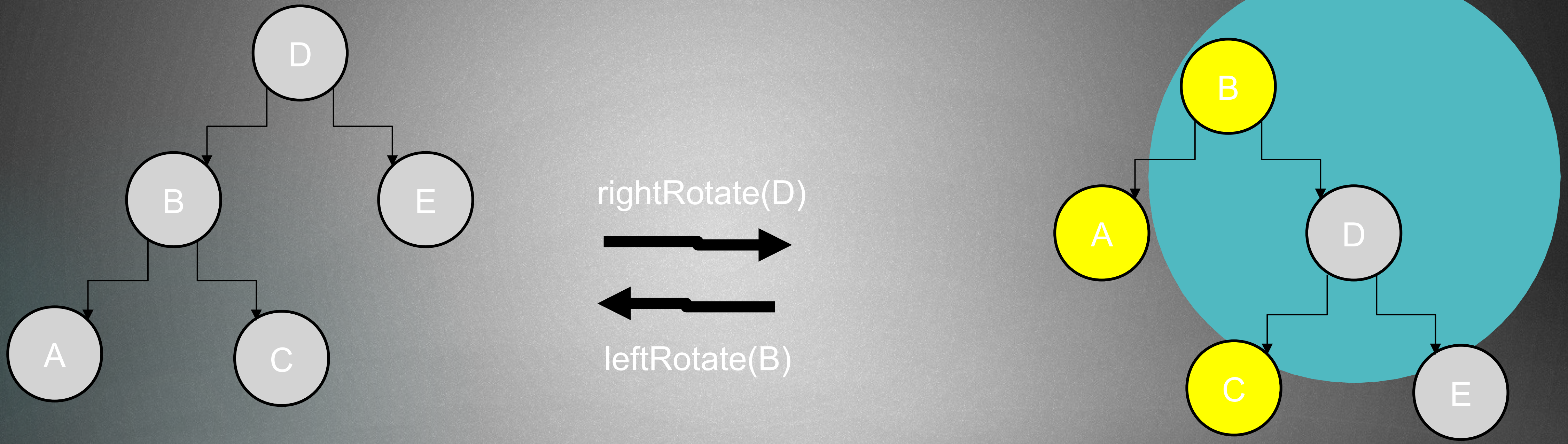
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



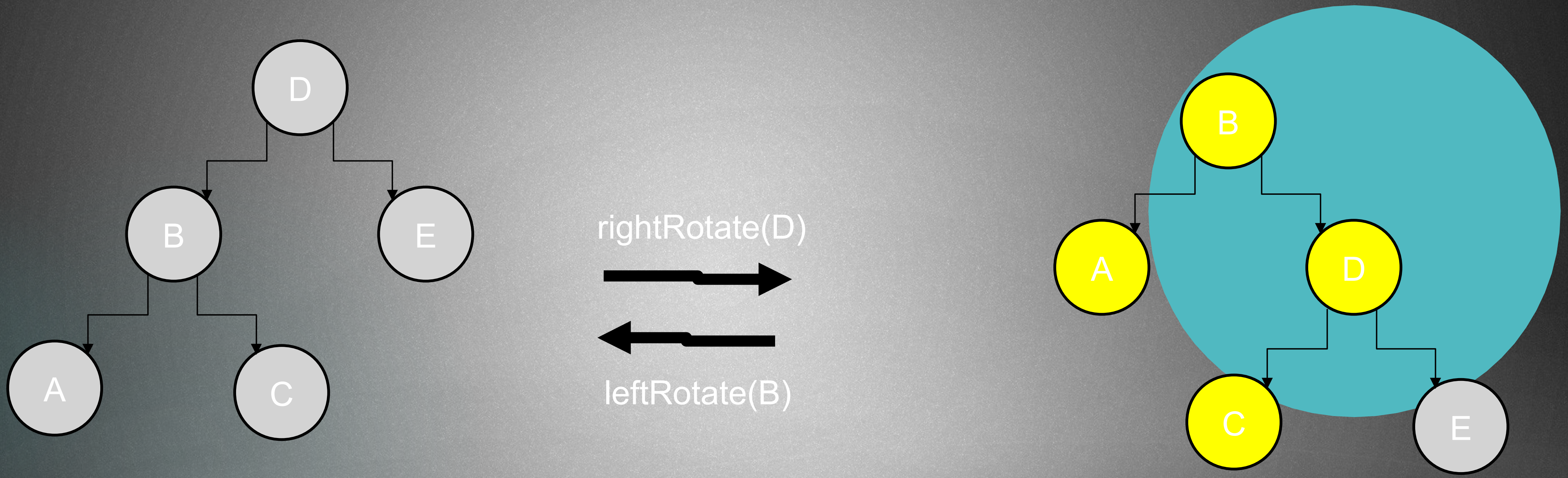
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



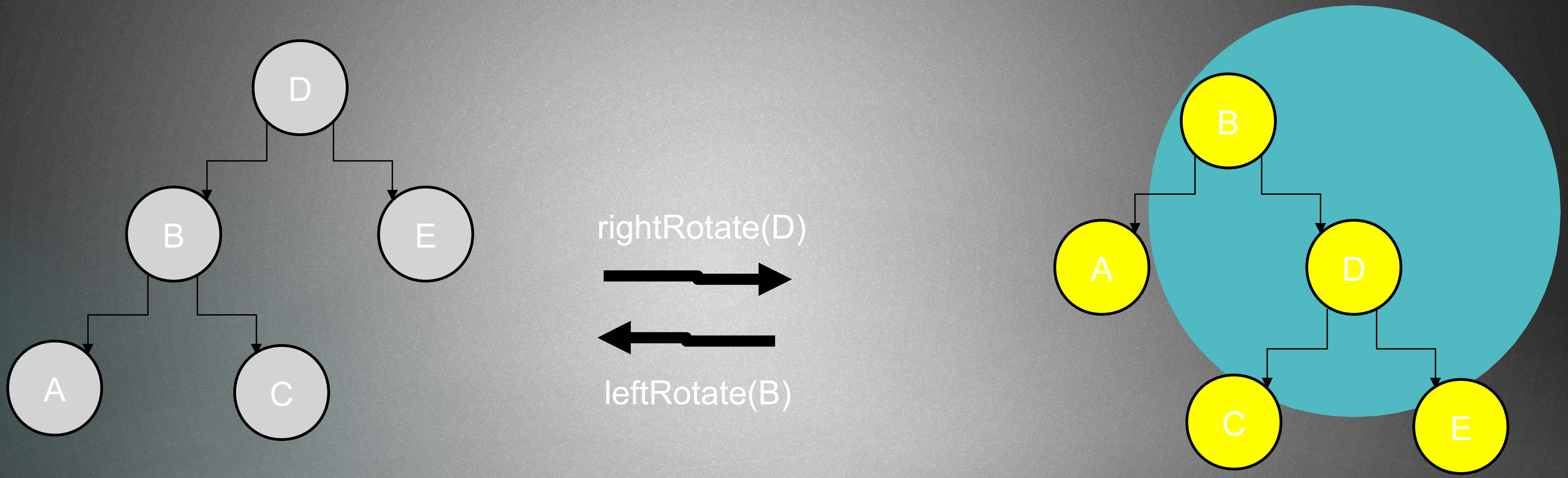
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations



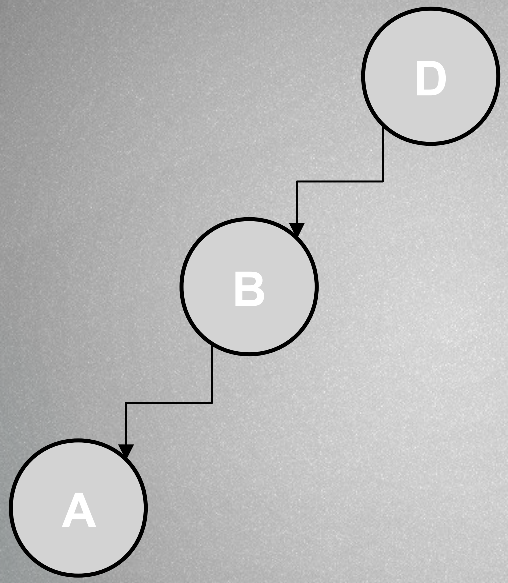
We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

Rotations

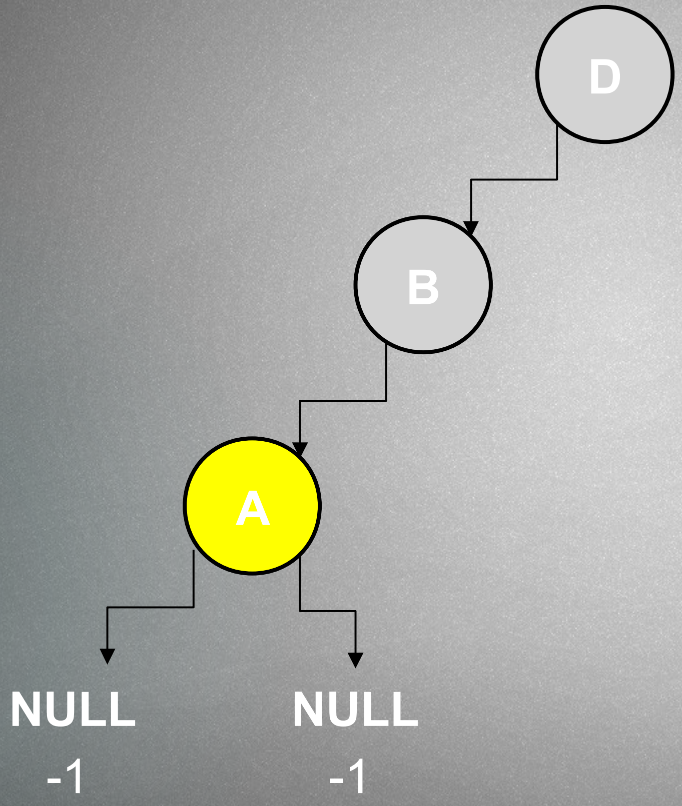


We just have to update the references which can be done in **O(1)** time complexity !!! (the in-order traversal is the same)

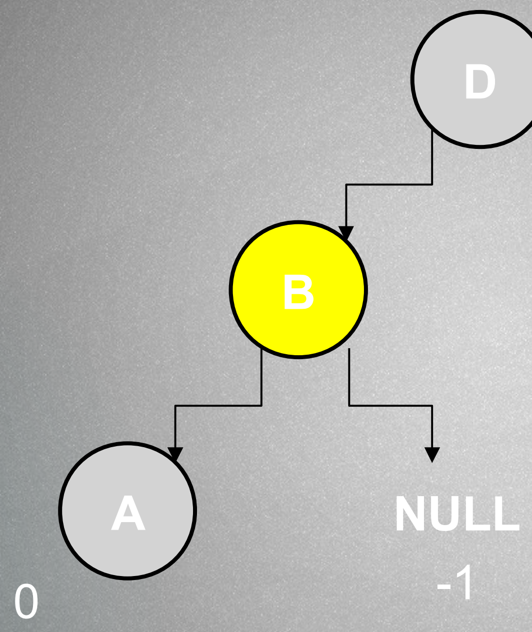
Rotations case I



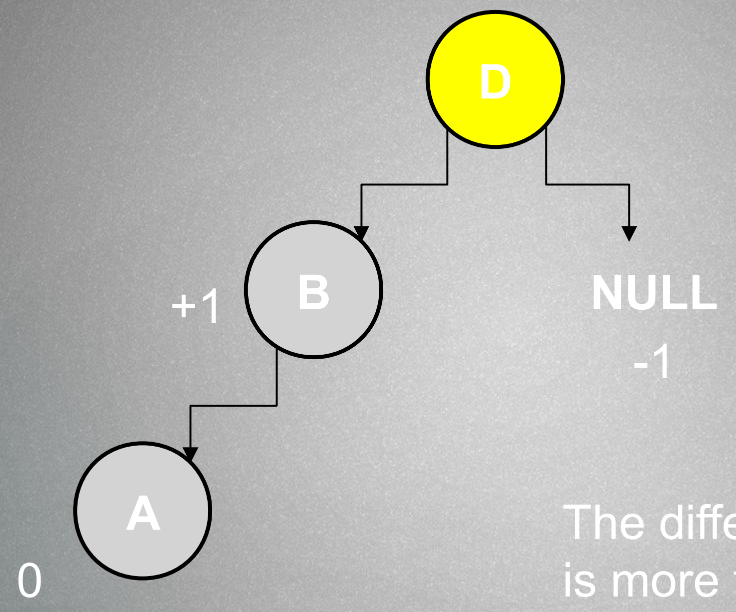
Rotations case I



Rotations case I

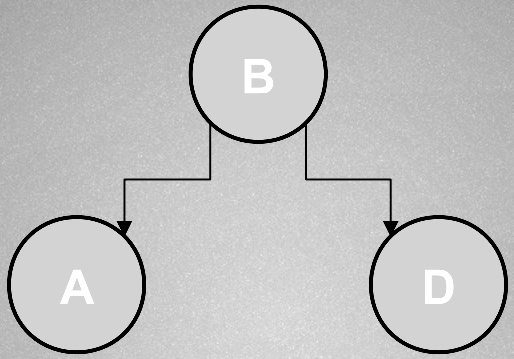


Rotations case I



The difference of height parameters
is more than 1 !!! (actually it is 2)
~ so we make rotation to the right

Rotations case I

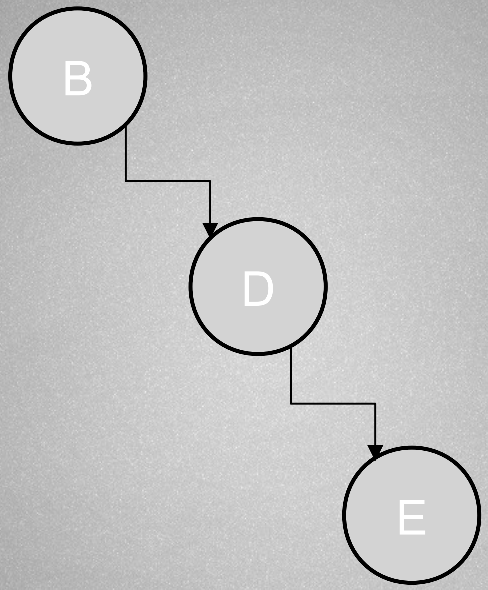


The new root node is the **B**, which was the left child of **D** before the rotation !!!

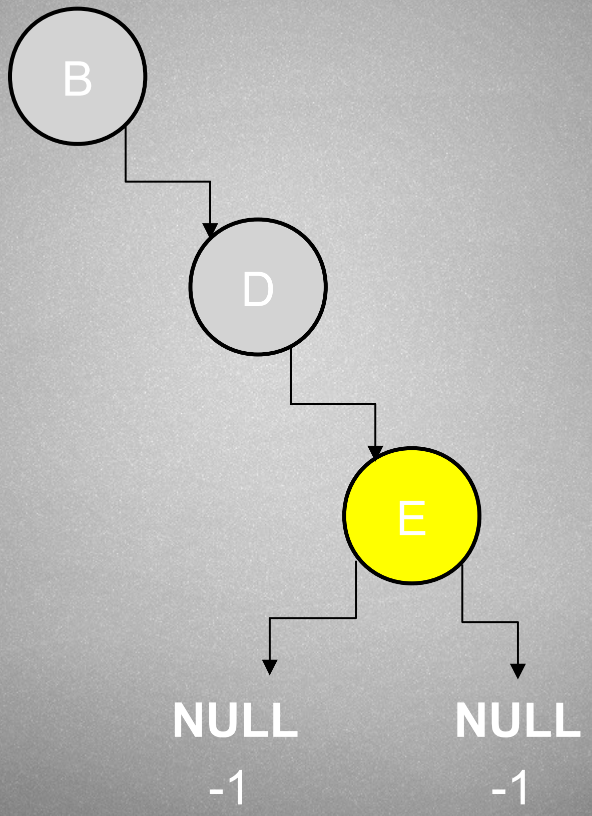
Rotations case I

```
BEGIN rotateRight(Node node)

    Node tempLeftNode = node.getLeftNode()
    Node t = tempLeftNode.getRightNode()

    tempLeftNode.setRightNode(node)
    node.setLeftNode(t)

    node.updateheight()
    tempLeftNode.updateHeight()

END
```

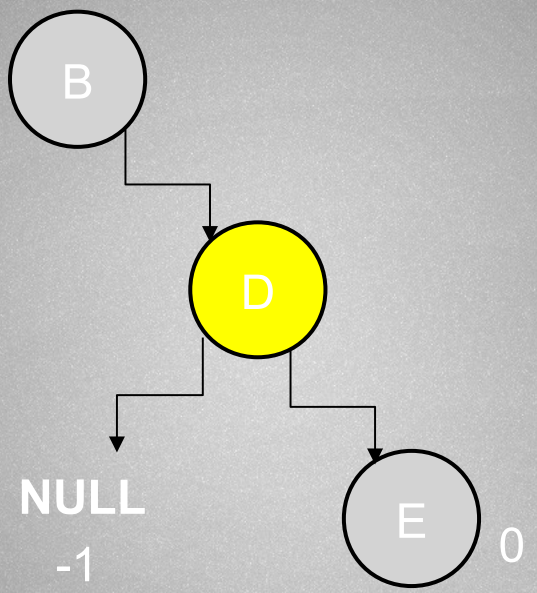
Rotations case II



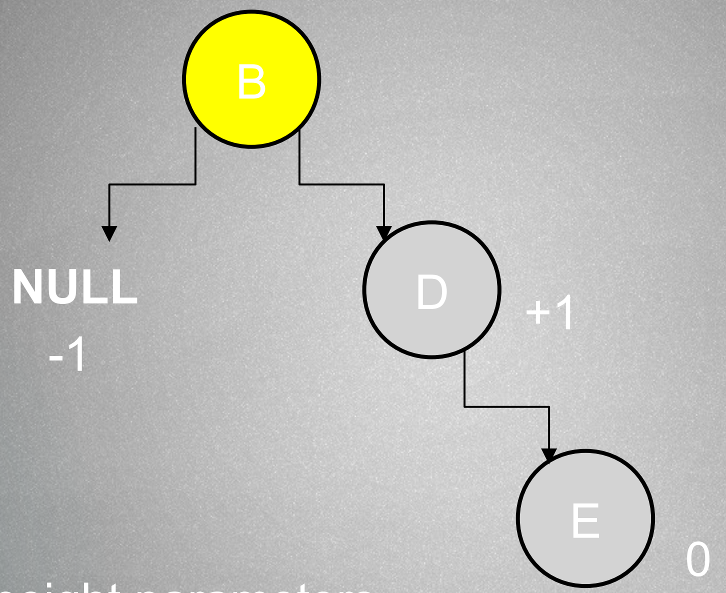
Rotations case II



Rotations case II

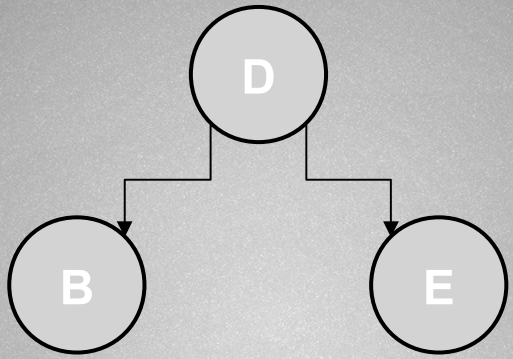


Rotations case II



The difference of height parameters
is more than 1 !!! (actually it is 2)
~ so we make rotation to the left

Rotations case II

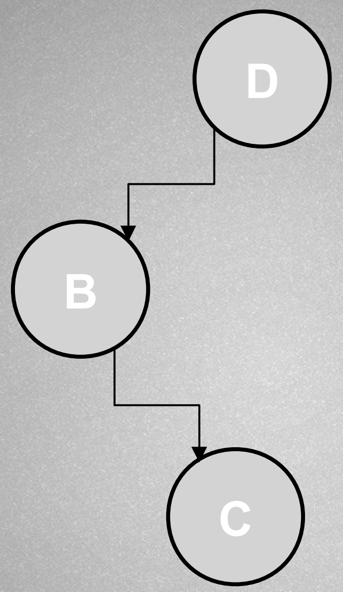


The new root node is the **D**, which was the right child of **B** before the rotation !!!

Rotations case II

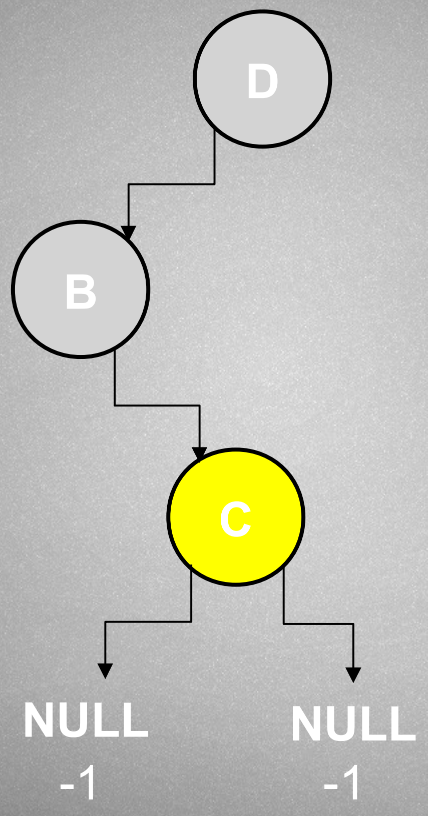
```
BEGIN rotateLeft(Node node)

	Node tempRightNode = node.getRightNode()
	Node t = tempRightNode.getLeftNode()

	tempRightNode.setLeftNode(node)
	node.setRightNode(t)

	node.updateheight()
	tempRightNode.updateHeight()

END
```

Rotations case III

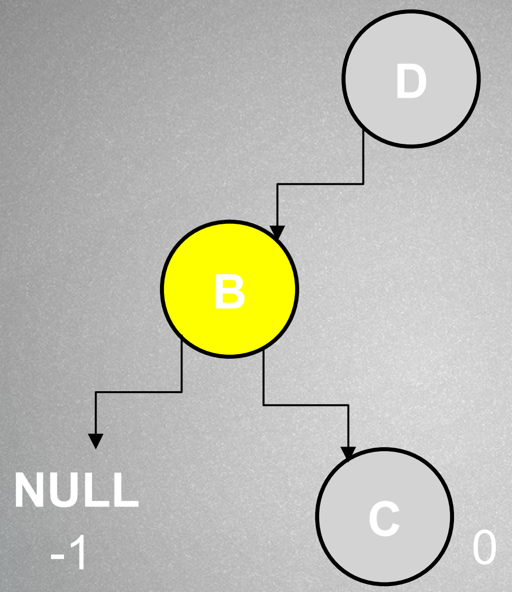


IMPORTANT: these nodes may have left and right children but it does not matter // we do not modify the pointers for them !!!

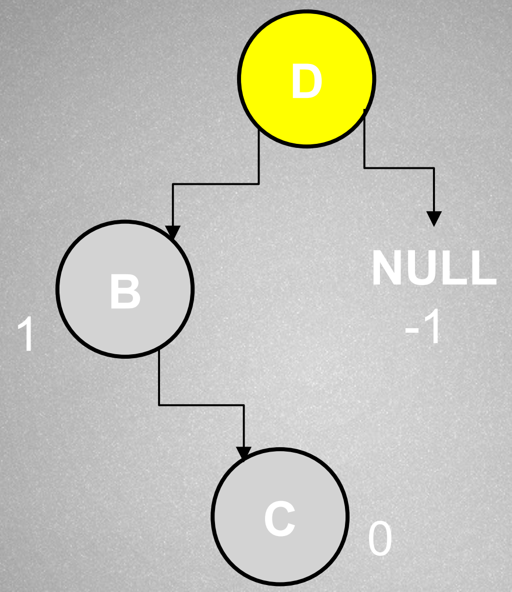
Rotations case III



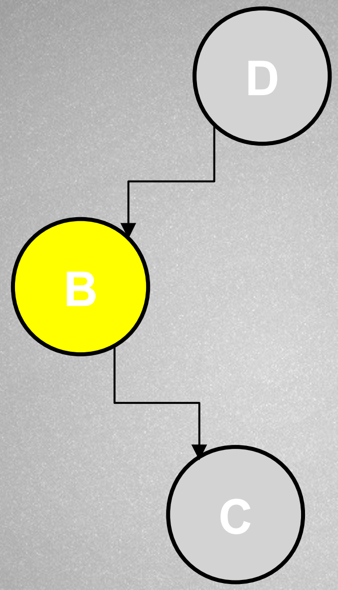
Rotations case III



Rotations case III

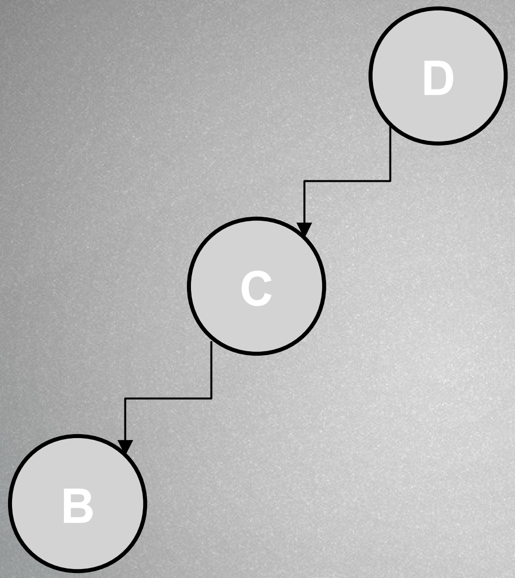


Rotations case III

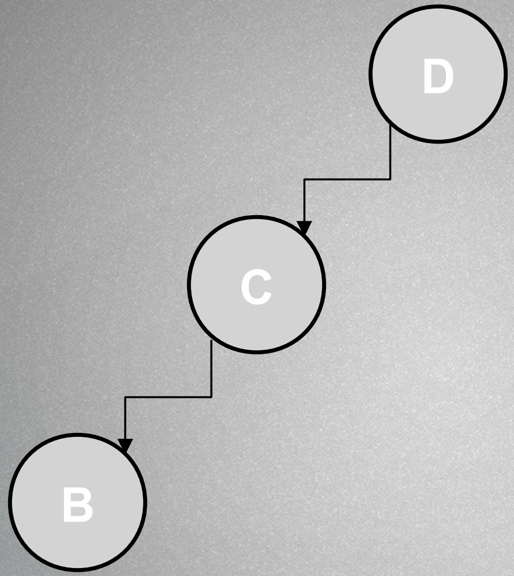


We have to make a left rotation on the node B

Rotations case III

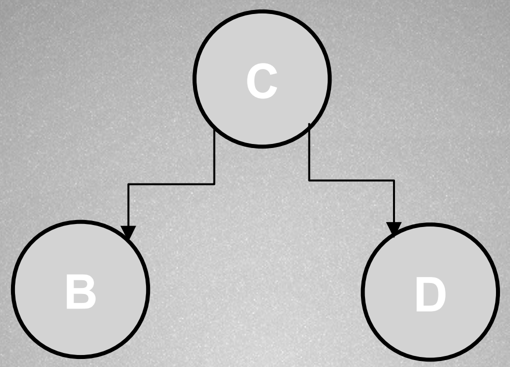


Rotations case III

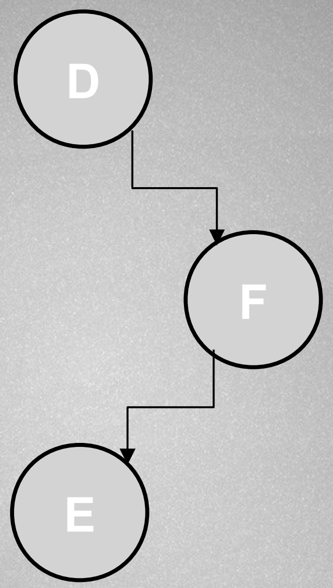


We have to make a left rotation on the root node D

Rotations case III

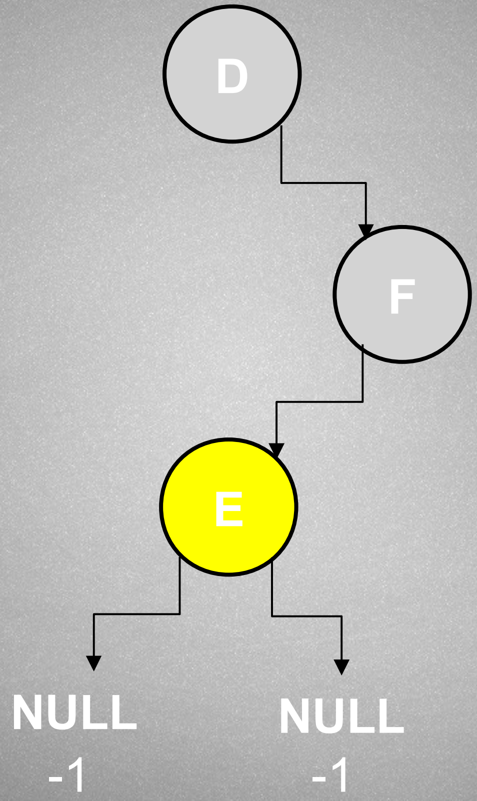


Rotations case IV

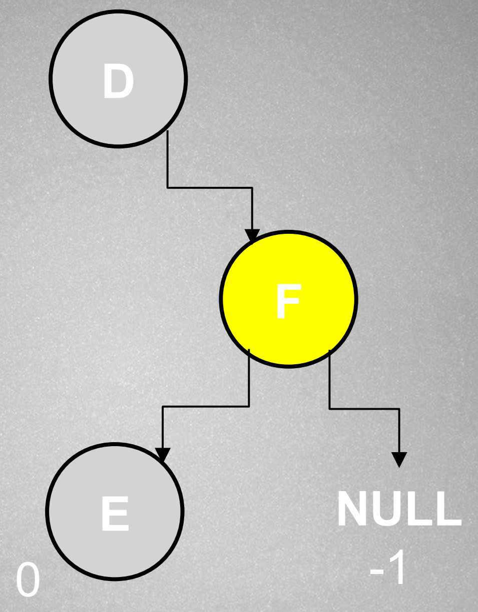


IMPORTANT: these nodes may have left and right children but it does not matter // we do not modify the pointers for them !!!

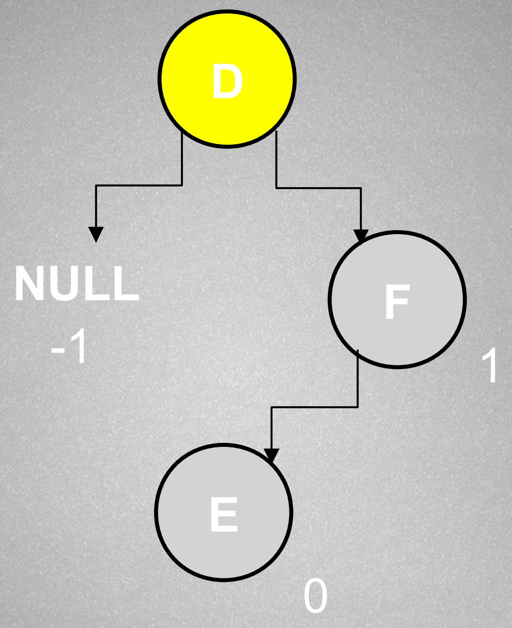
Rotations case IV



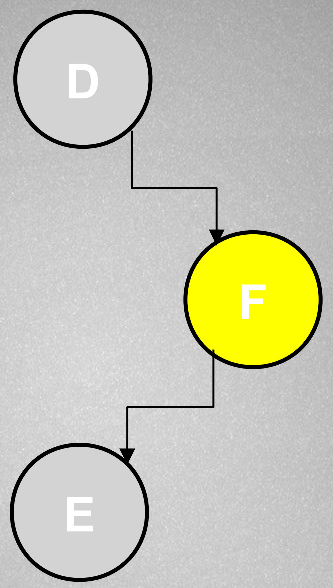
Rotations case IV



Rotations case IV

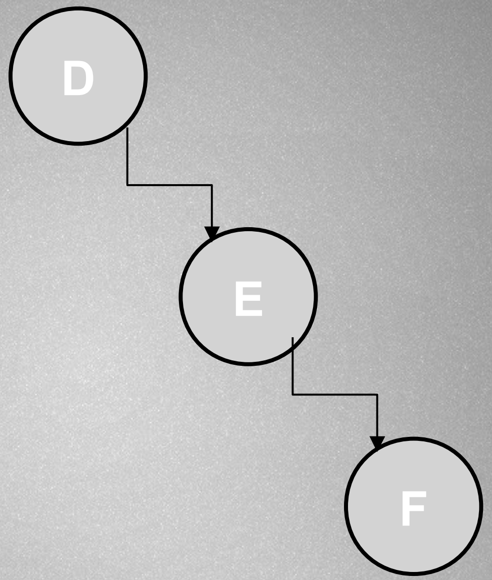


Rotations case IV

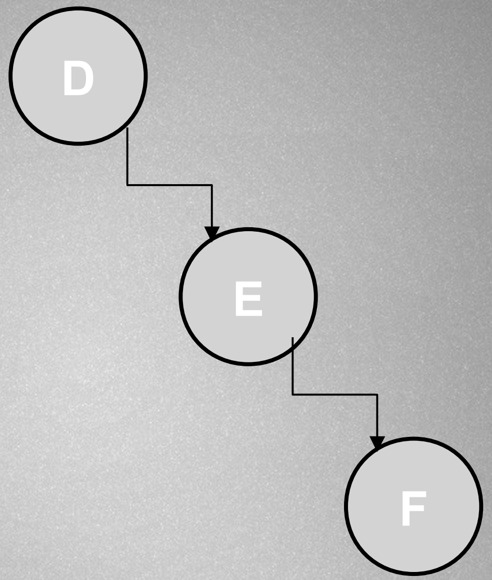


We have to make a right rotation on the node F

Rotations case IV

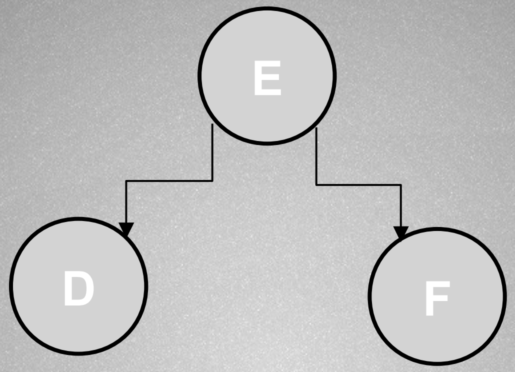


Rotations case IV



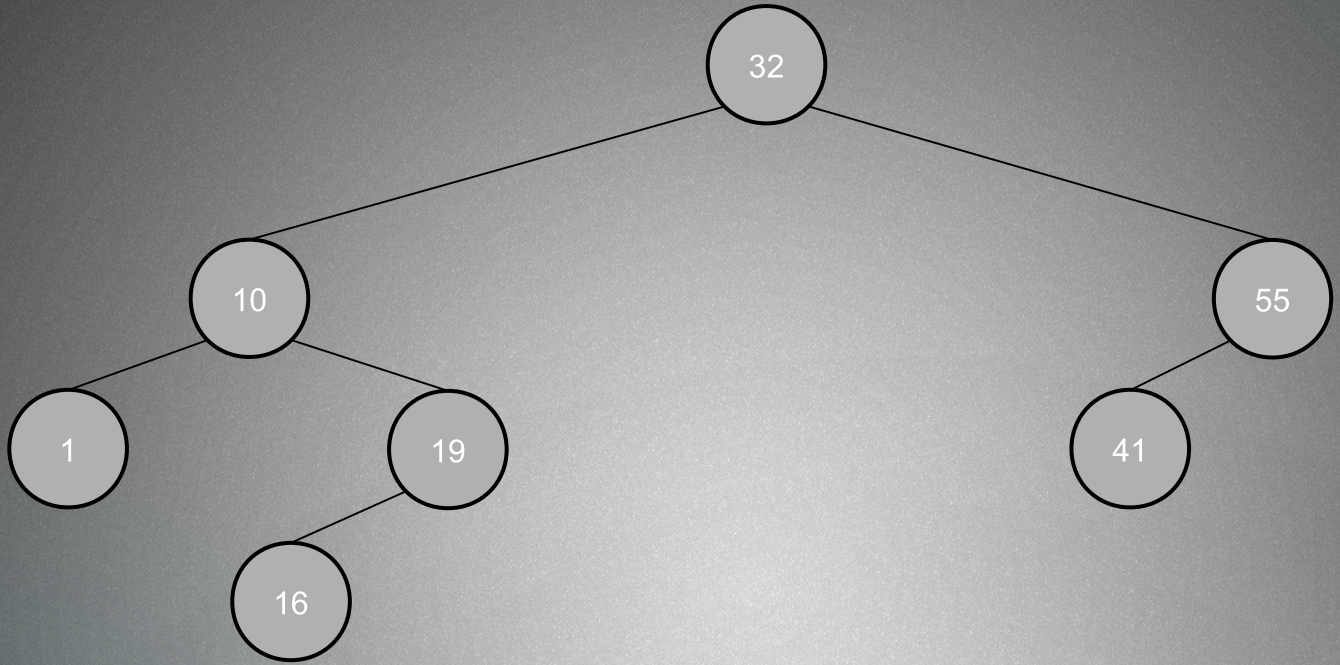
We have to make a left rotation on the root node D

<u>**Rotations**</u> case IV

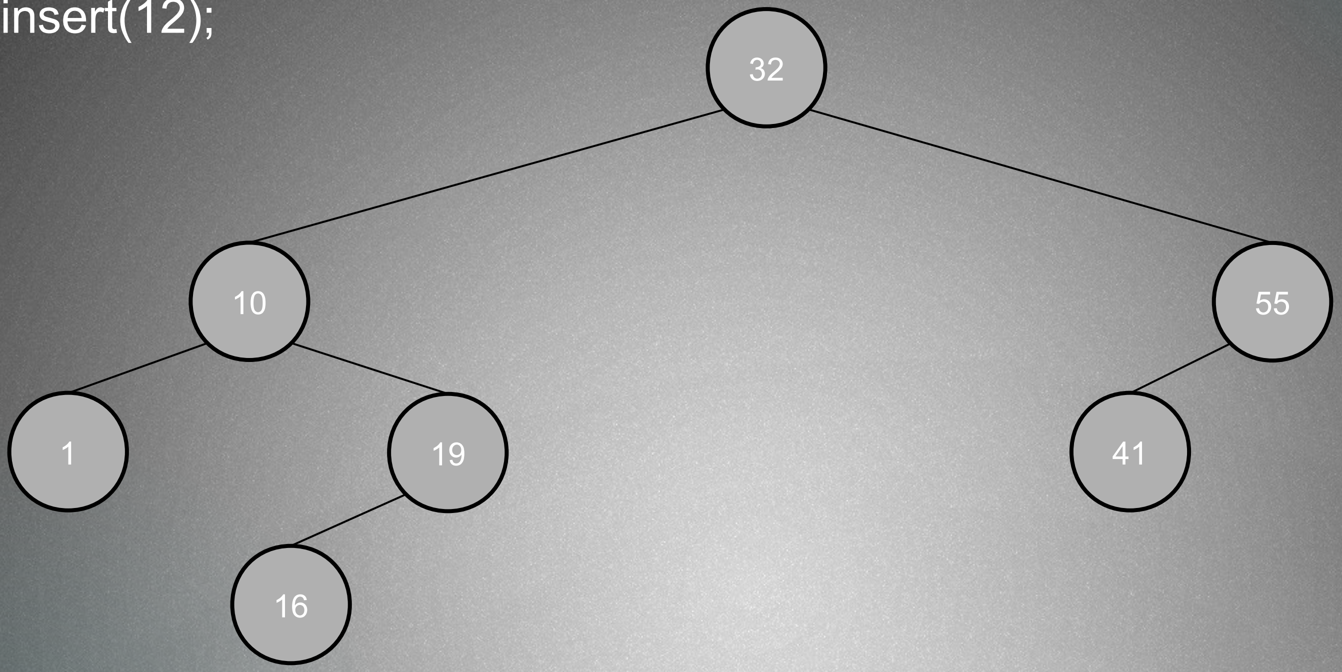


AVL TREES

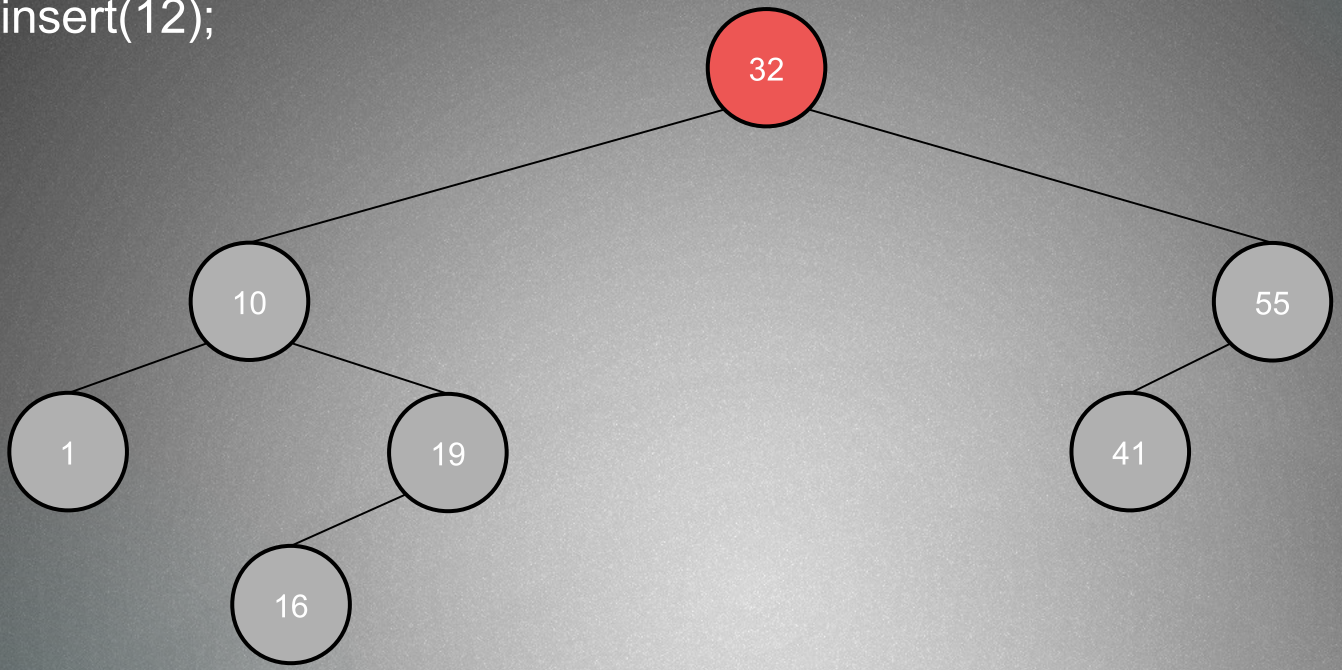
BALANCED TREES


32
10
55
1
19
41
16

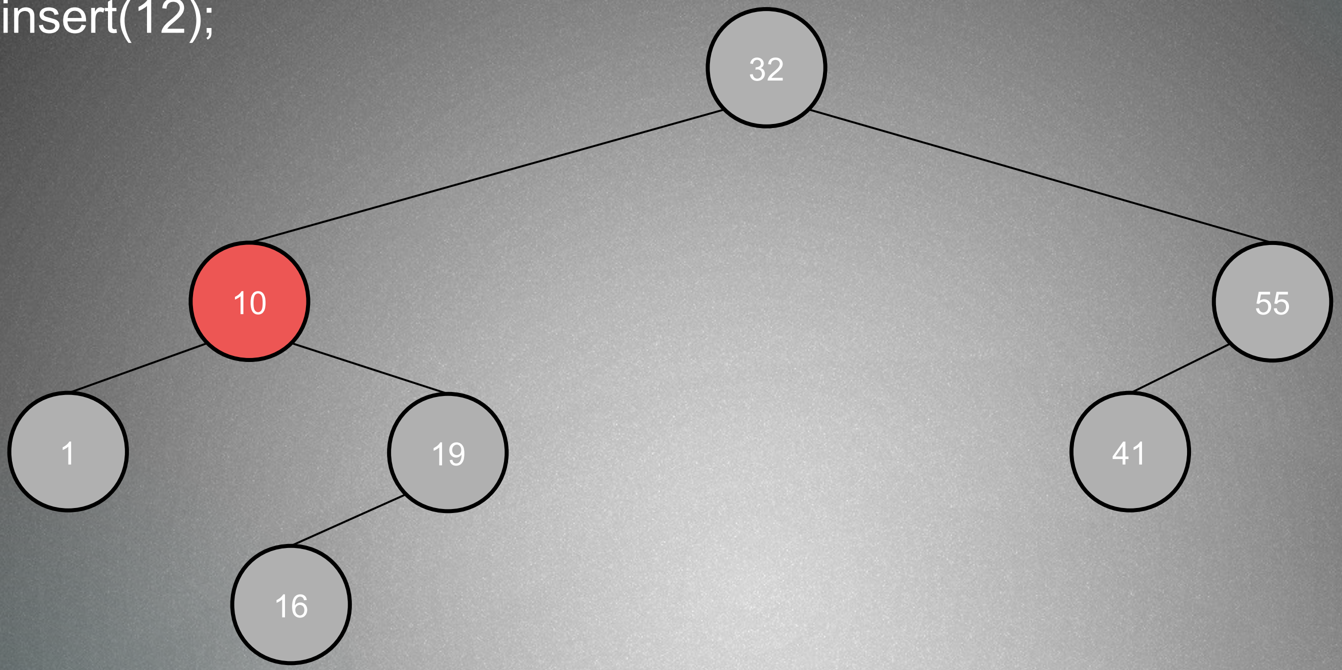
balancedTree.insert(12);



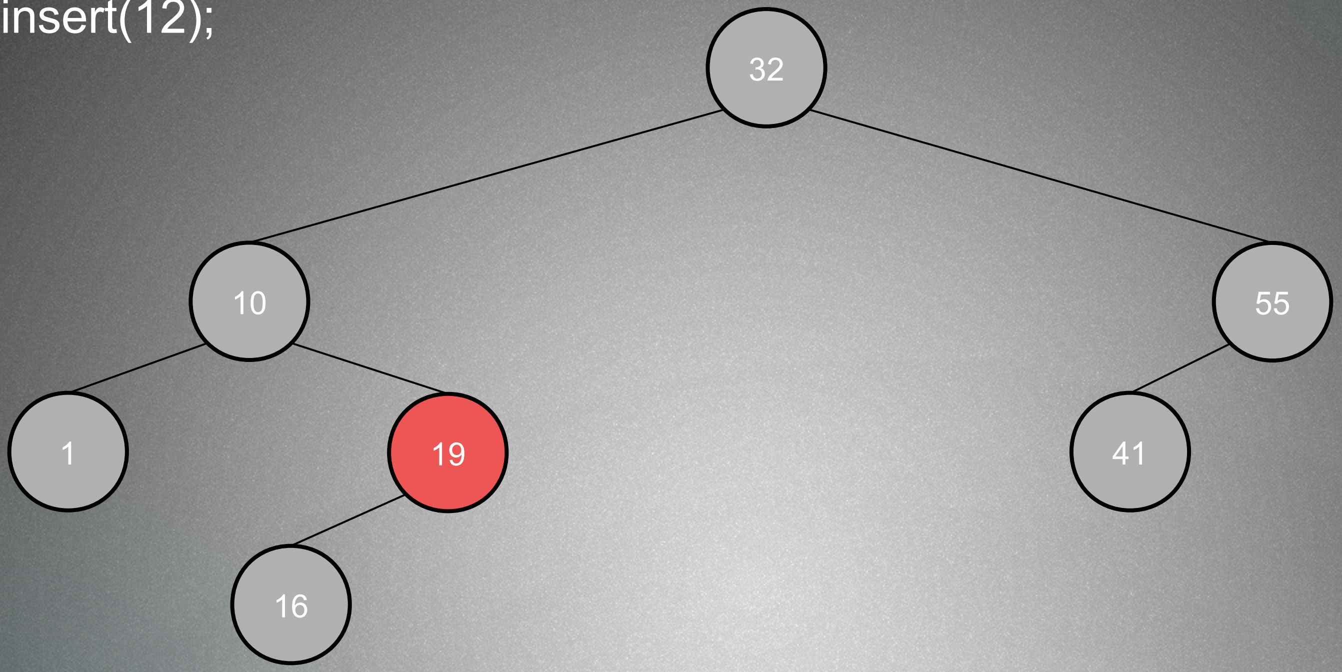
balancedTree.insert(12);



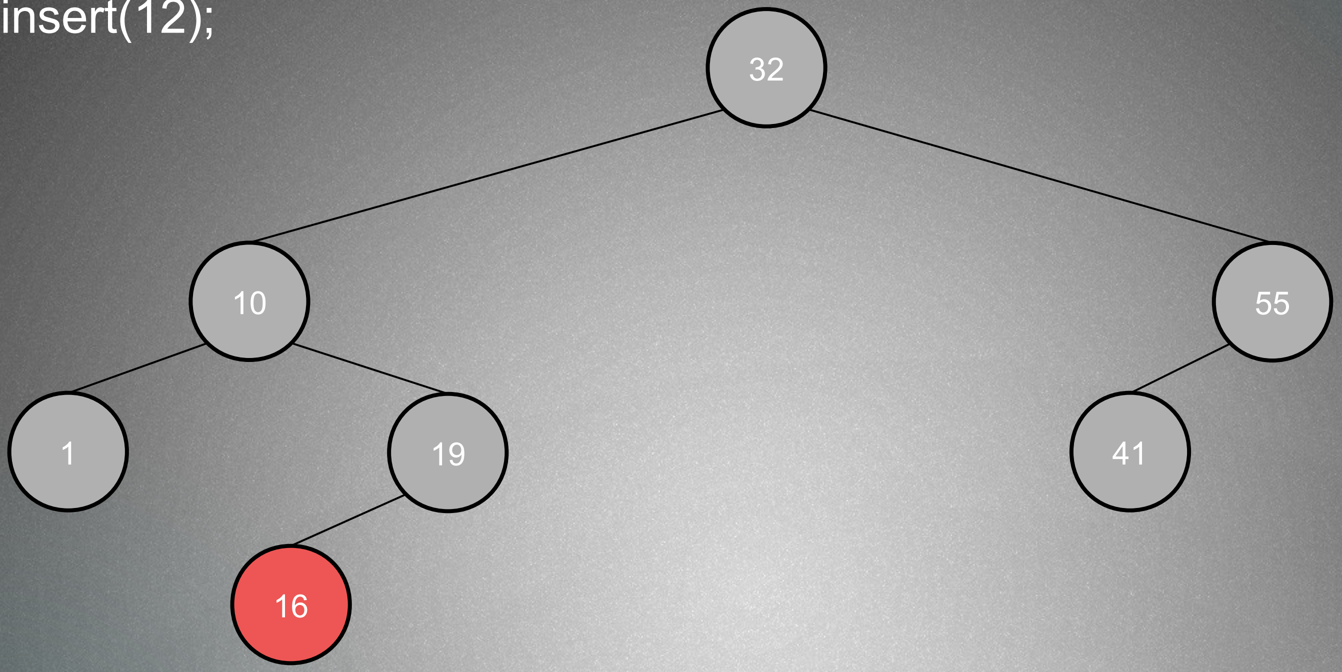
balancedTree.insert(12);



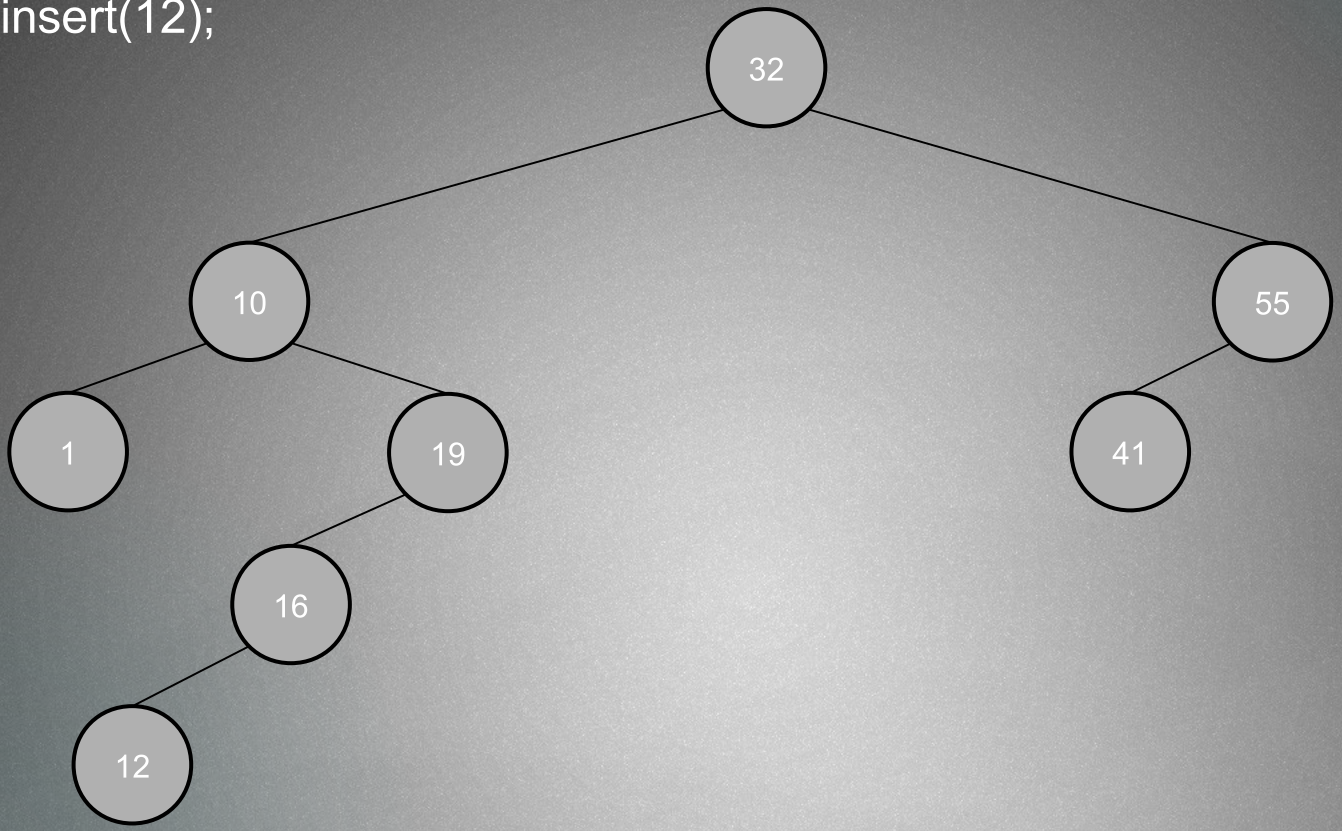
balancedTree.insert(12);



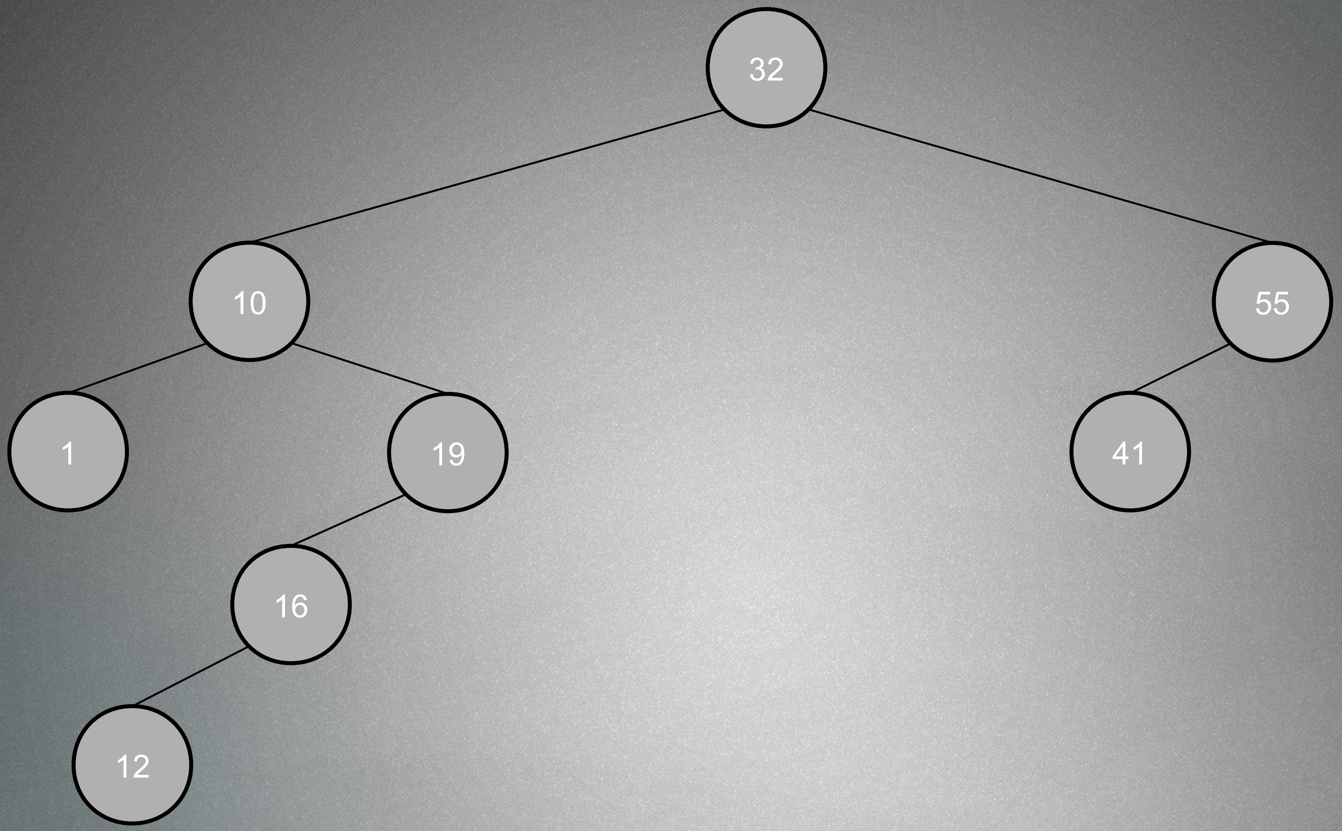
balancedTree.insert(12);



balancedTree.insert(12);

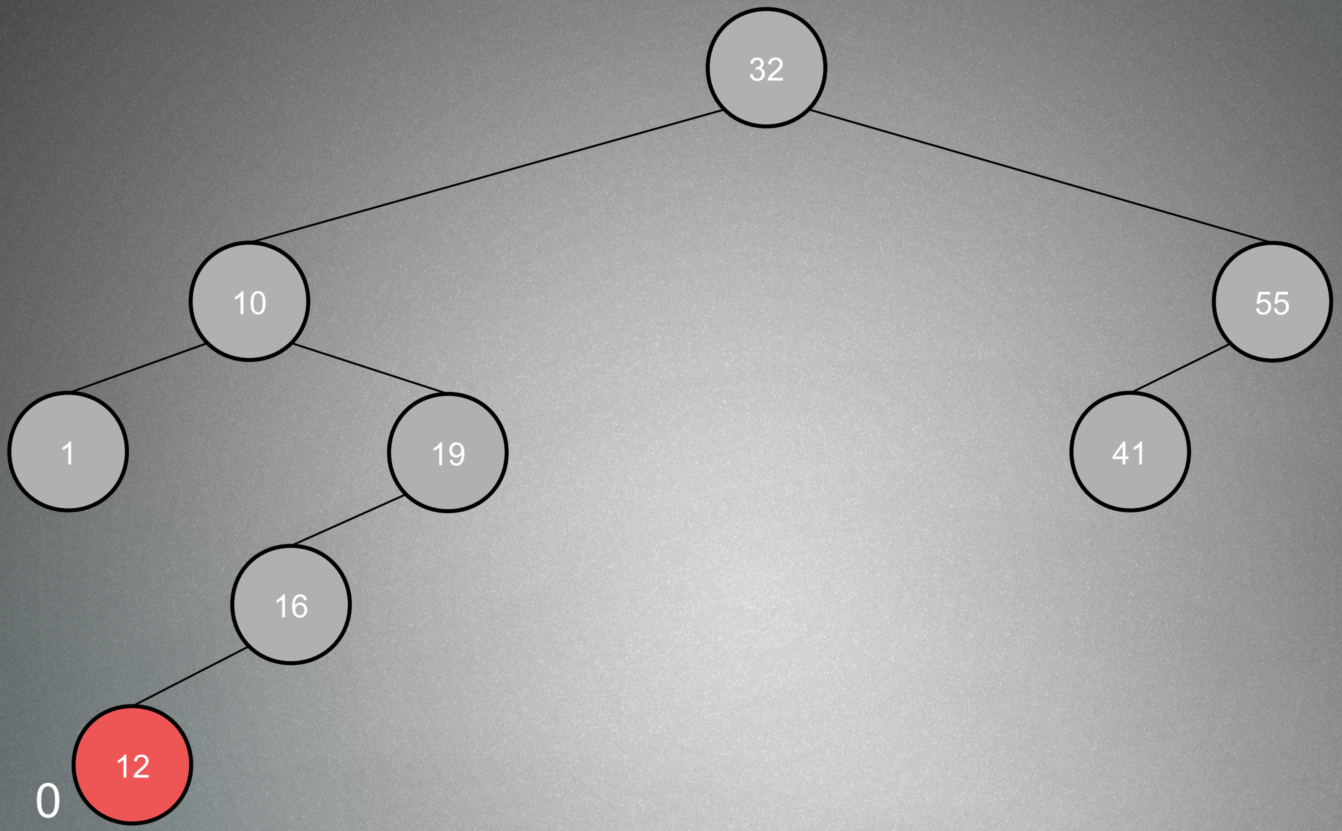


Let’s calculate the height for each node



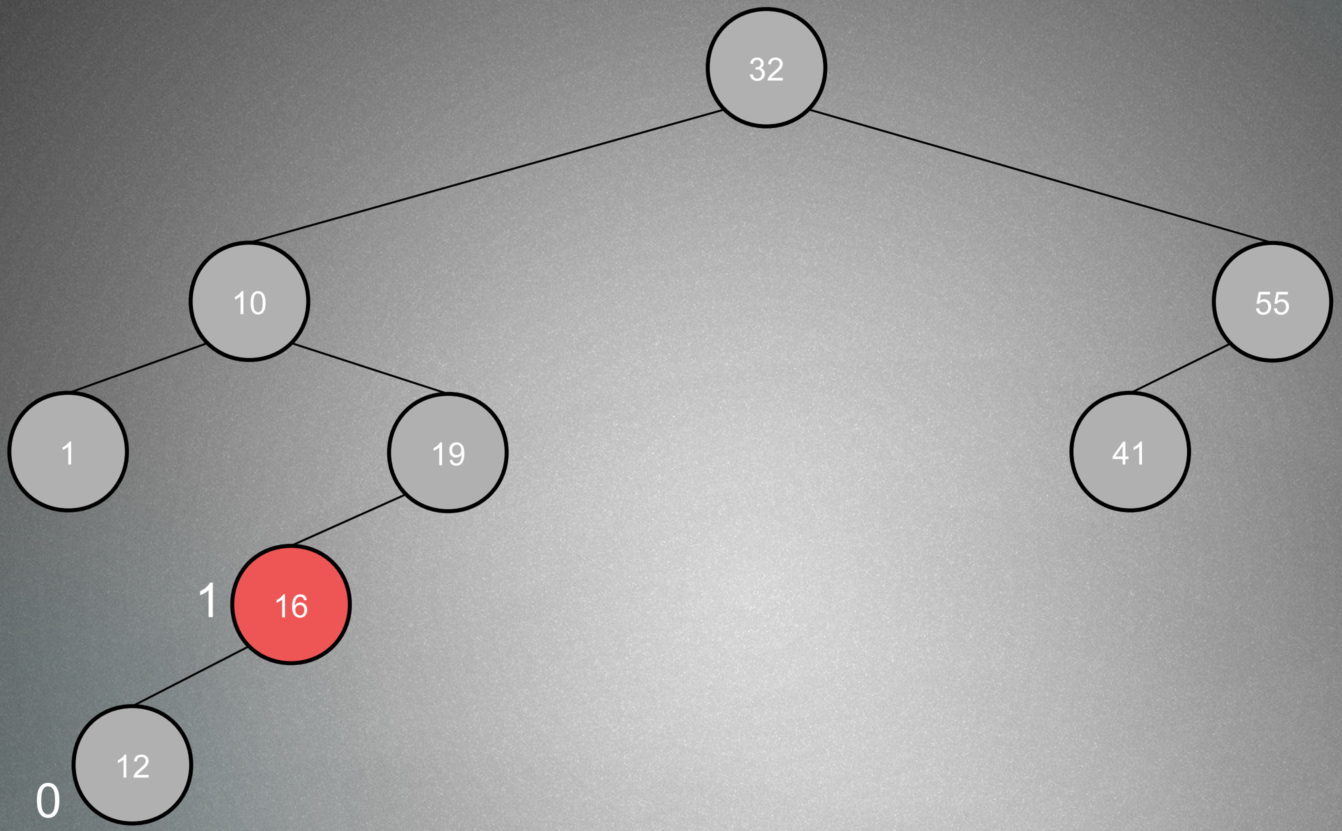
Important: to be able to write algorithm for calculating the height, we consider null pointers (when a node have no left child for example) to be of height -1 !!!

Let’s calculate the height for each node



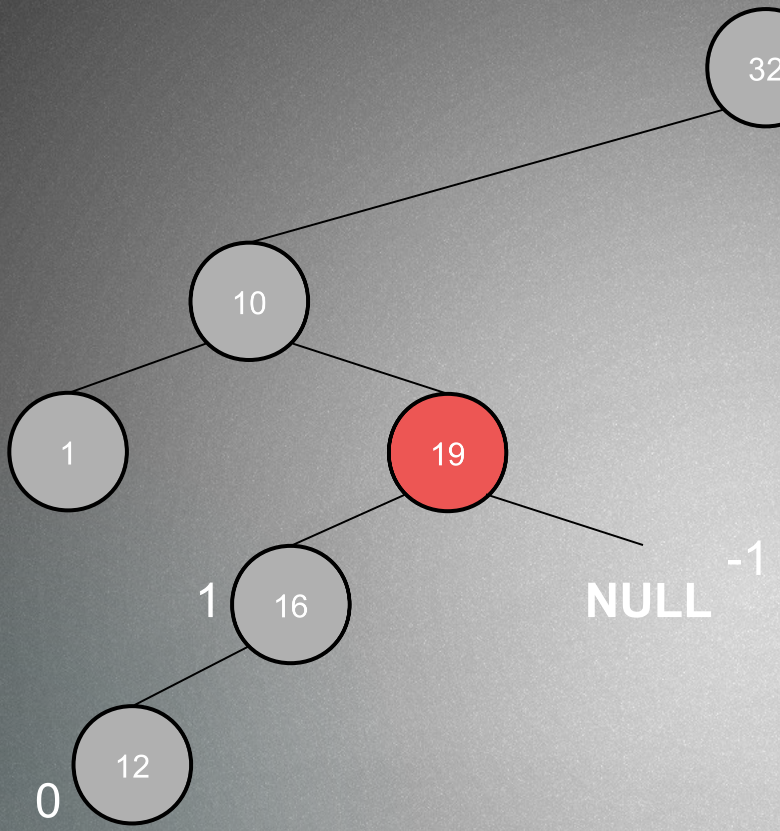
height = max(leftChild.height(),rightChild.height())+1

Let's calculate the height for each node



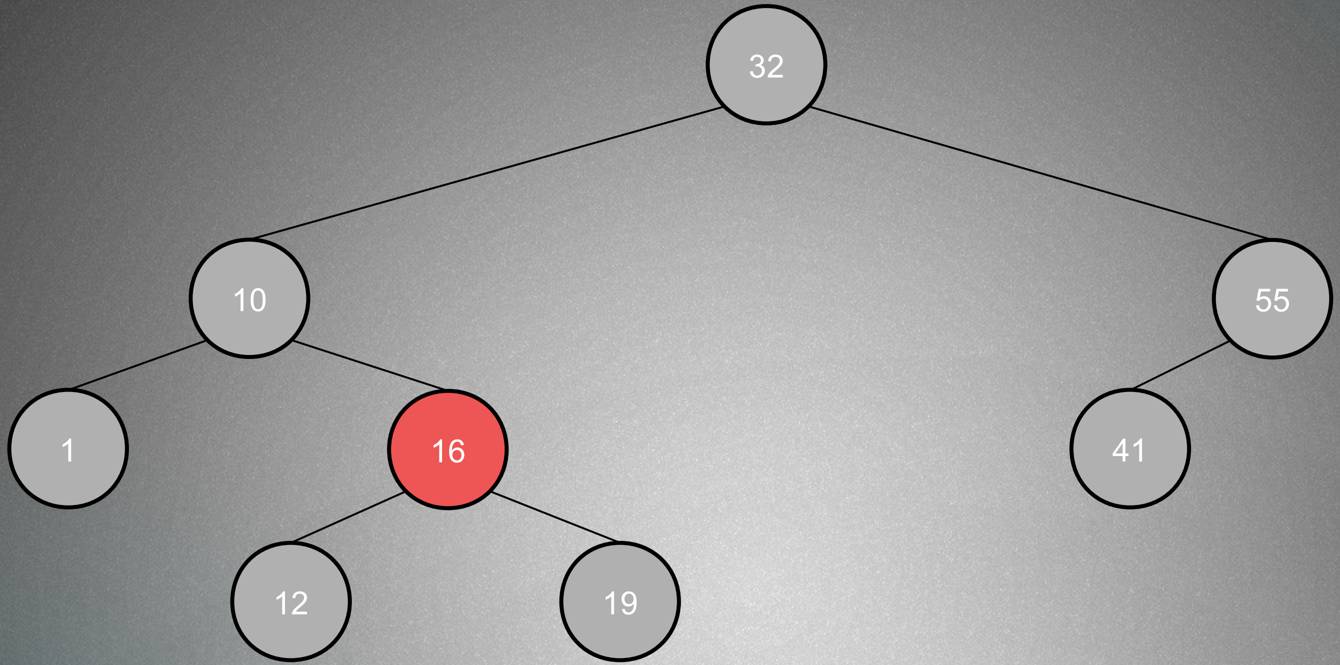
height = max(leftChild.height(),rightChild.height())+1

Let’s calculate the height for each node

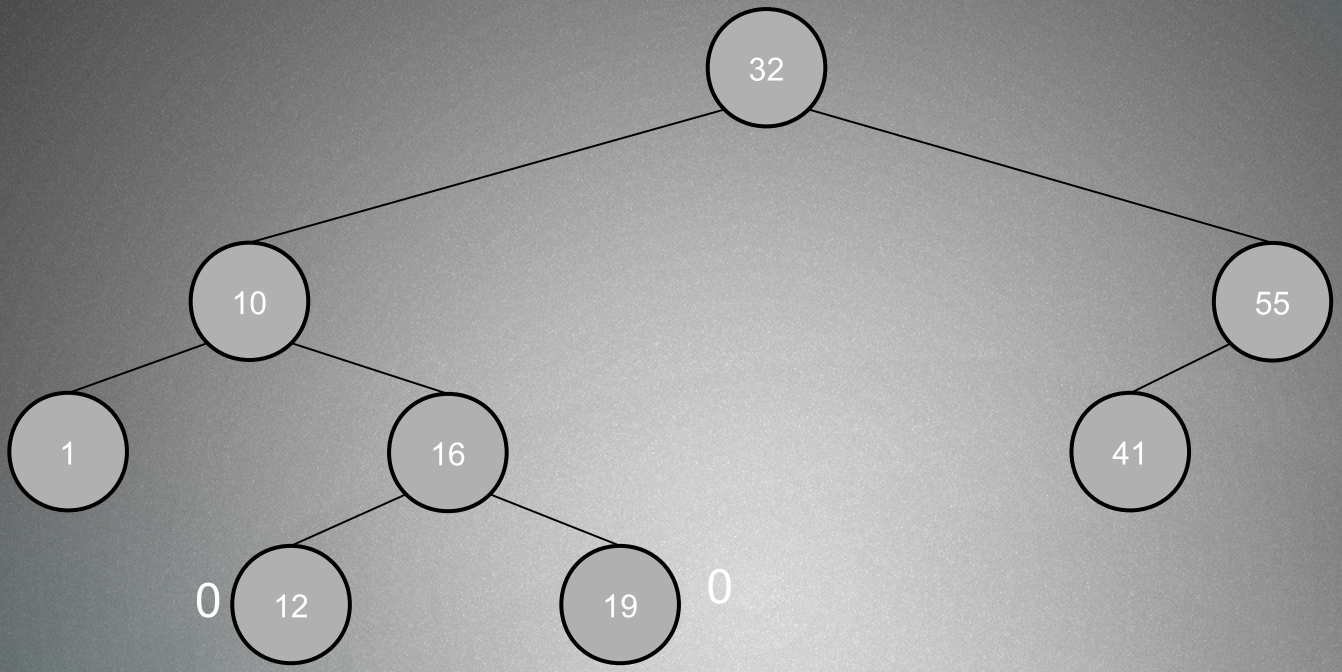


Problem: right child height is -1, left child height is +1 !!!
We have to make rotations // NULL objects have height -1 !!!

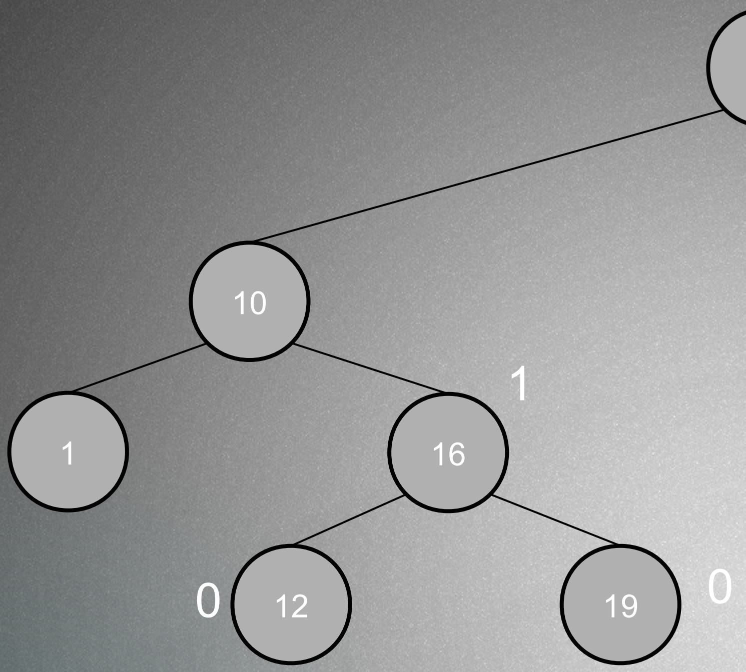
Let’s calculate the height for each node



Let’s calculate the height for each node

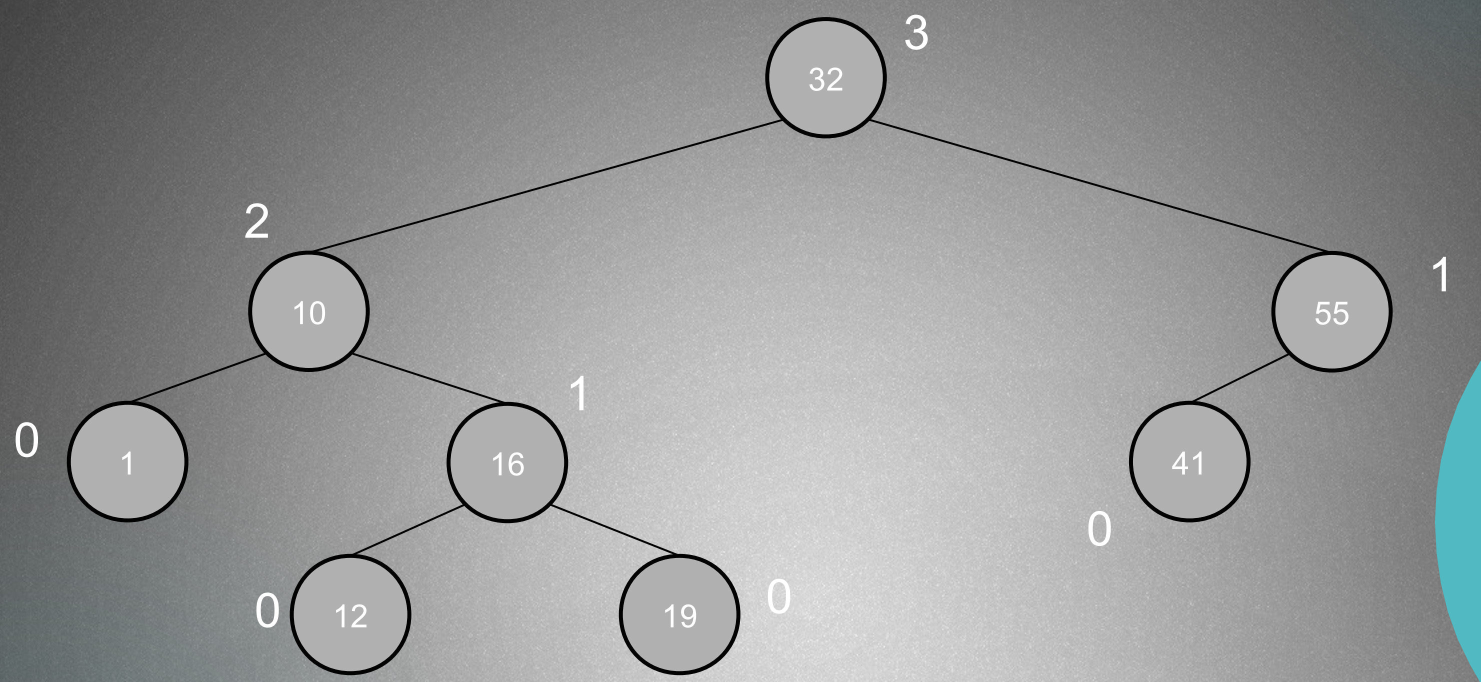


Let’s calculate the height for each node



height = max(leftChild.height(),rightChild.height())+1

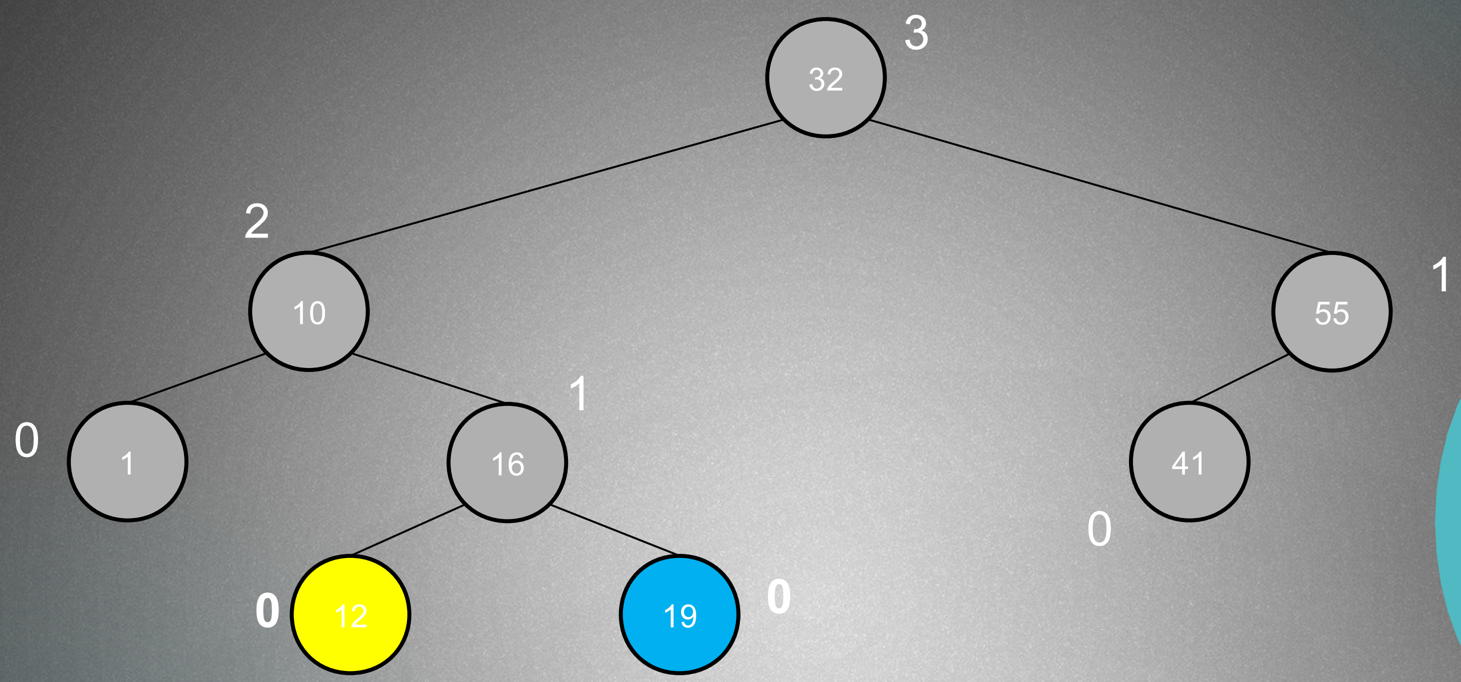
Let’s calculate the height for each node



height = max(leftChild.height(),rightChild.height())+1

After the rotation: it is a valid balanced tree, the height of any left and right subtree do not differ more than 1 → so no further rotations are needed !!!

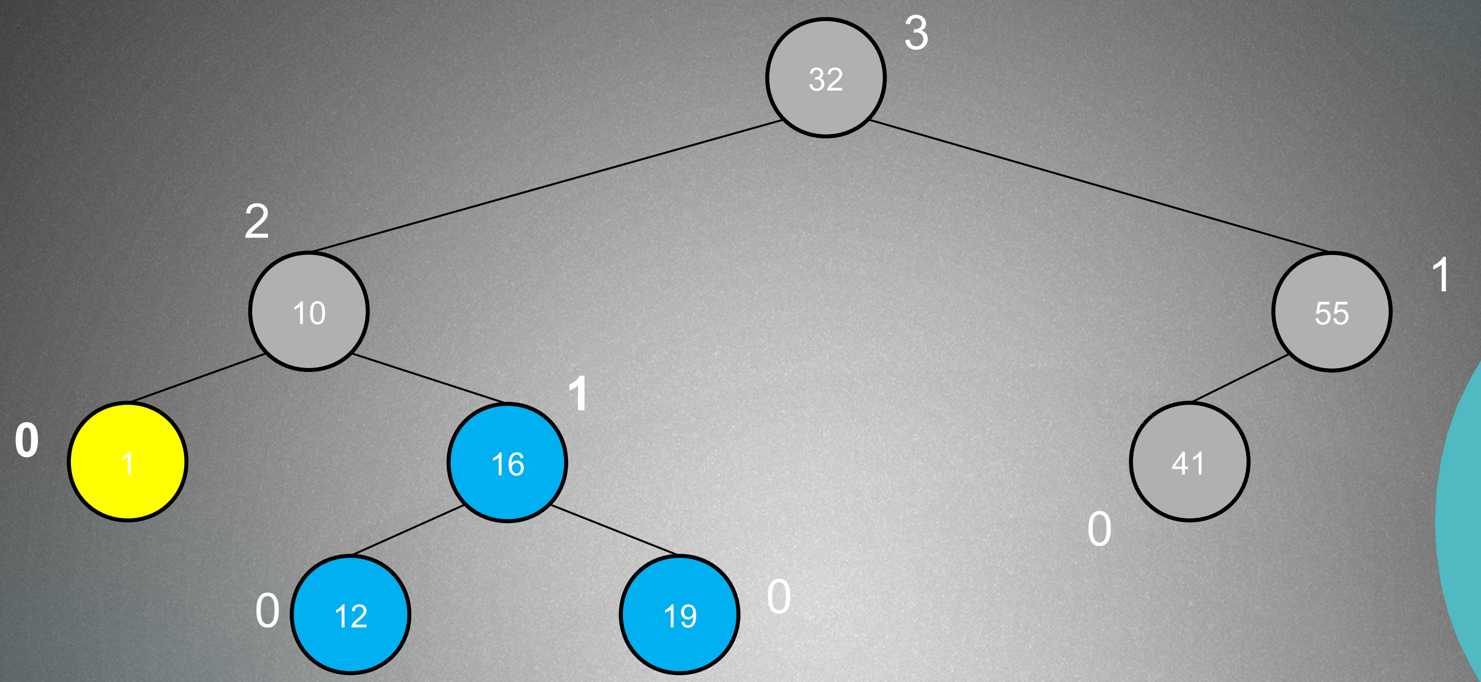
Let's calculate the height for each node



height = max(leftChild.height(),rightChild.height())+1

After the rotation: it is a valid balanced tree, the height of any left and right subtree do not differ more than 1 → so no further rotations are needed !!!

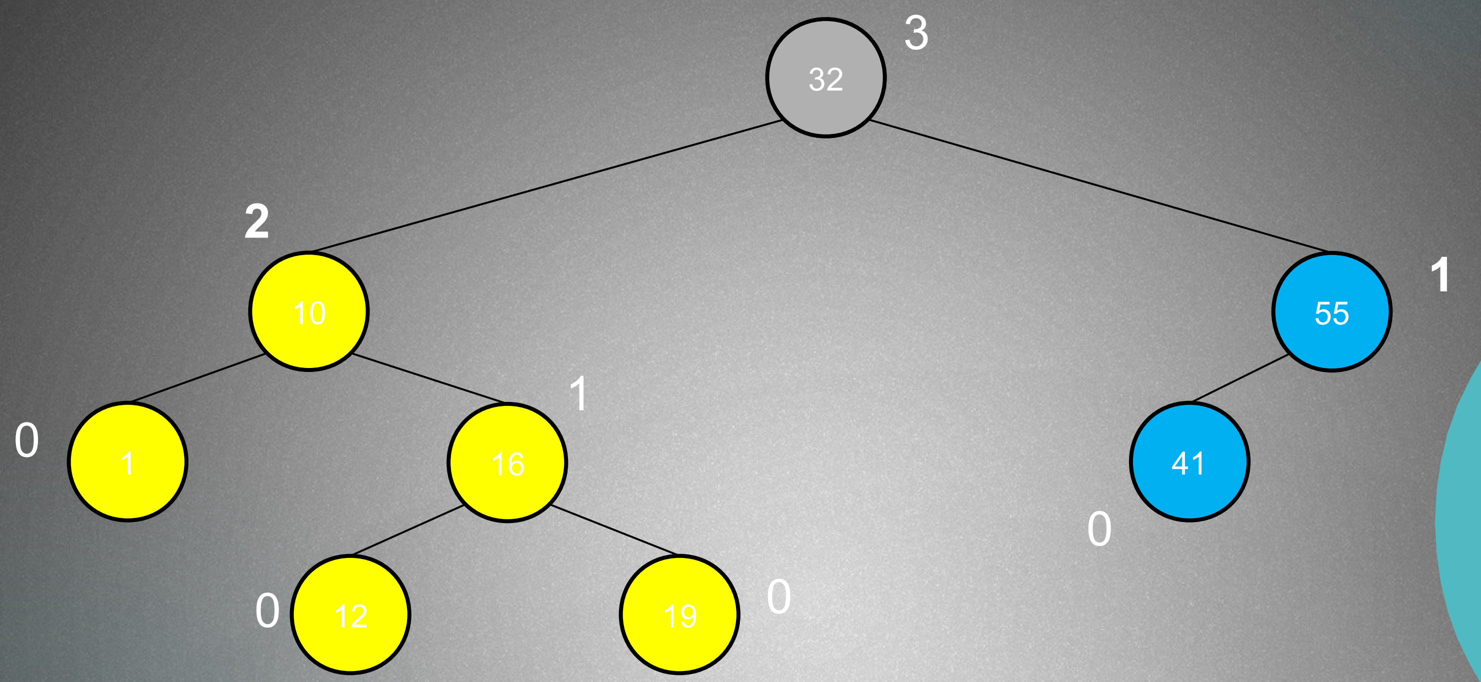
Let’s calculate the height for each node



height = max(leftChild.height(),rightChild.height())+1

After the rotation: it is a valid balanced tree, the height of any left and right subtree do not differ more than 1 → so no further rotations are needed !!!

Let’s calculate the height for each node



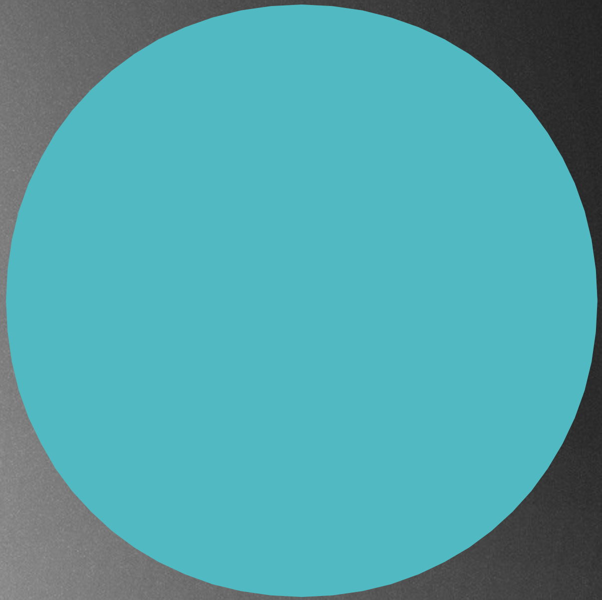
height = max(leftChild.height(),rightChild.height())+1

After the rotation: it is a valid balanced tree, the height of any left and right subtree do not differ more than 1 → so no further rotations are needed !!!

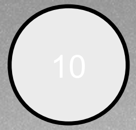
Rotations

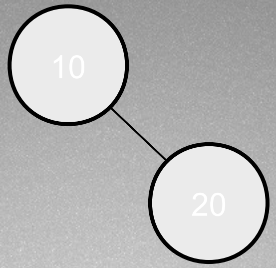
- Four types of unbalanced situations
  - LL: doubly left heavy situation…we have to make a right rotation
  - LR: we have to make a left and a right rotation
  - RL: we have to make a right and left rotation
  - RR: we have to make a left rotation

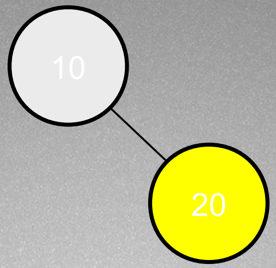
```
balancedTree.insert(10);
```

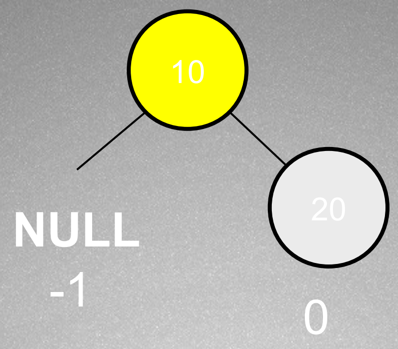


balancedTree.insert(20);

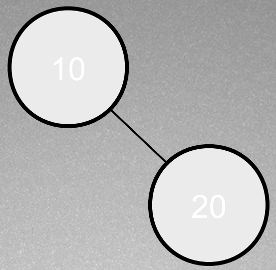


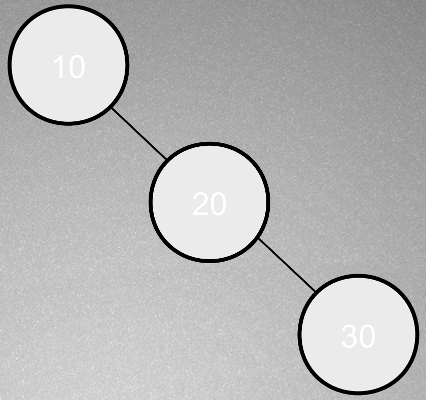

10
20

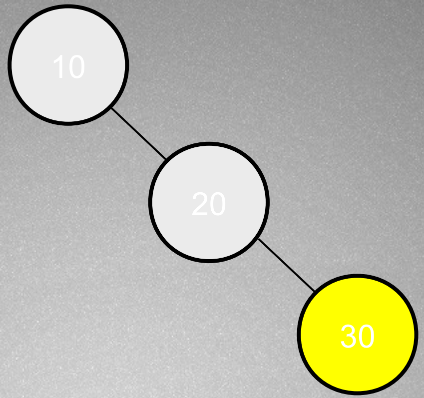

10
20

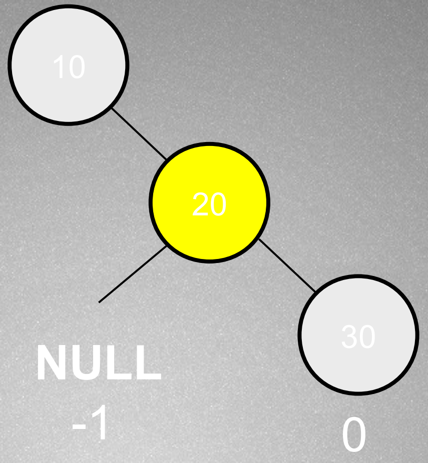

10
NULL
-1
20
0

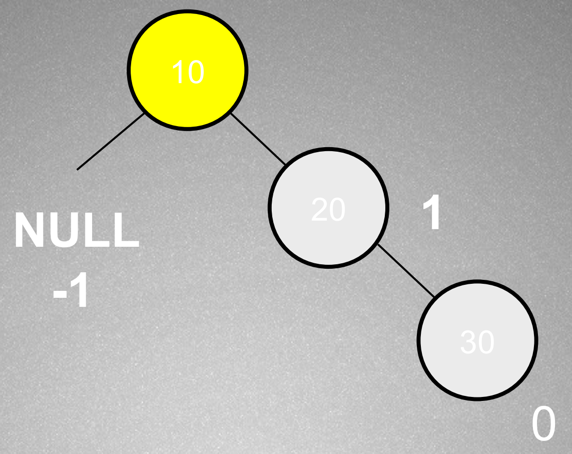
balancedTree.insert(30);



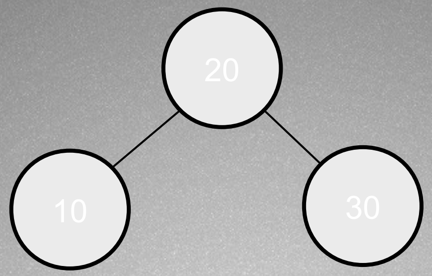

10
20
30


10
20
30

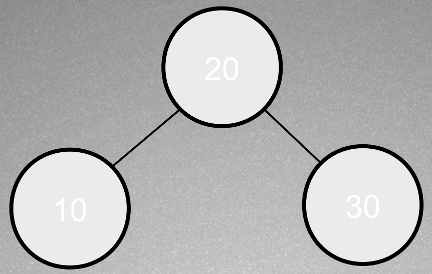

10
20
30
NULL
-1
0

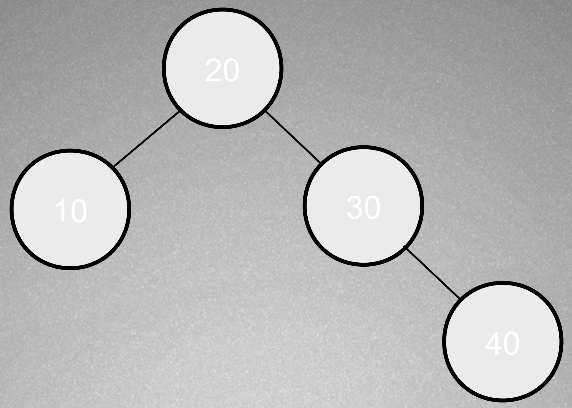


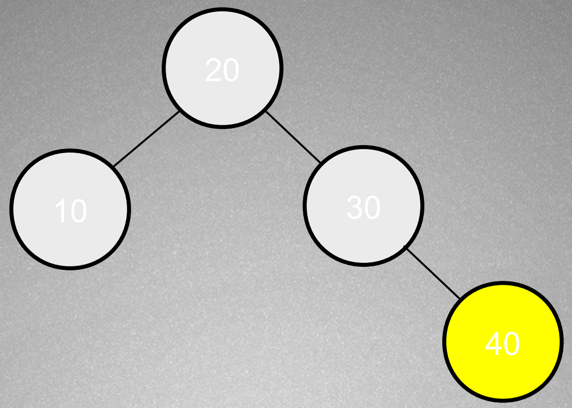
The difference between the height parameters is greater than 1 → rotations are needed !!!

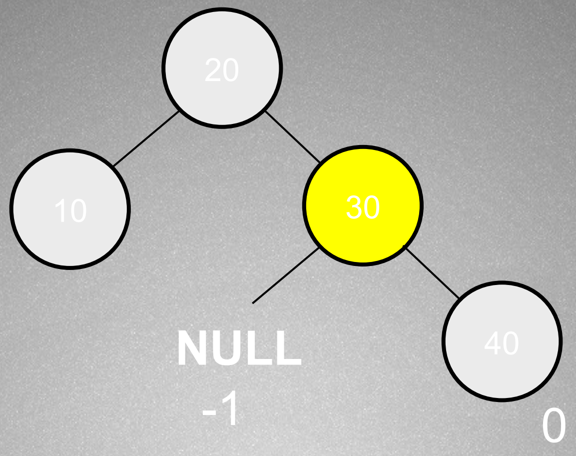

20
10
30

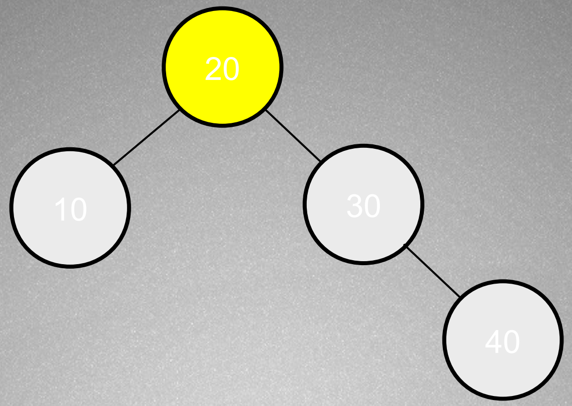
balancedTree.insert(40);

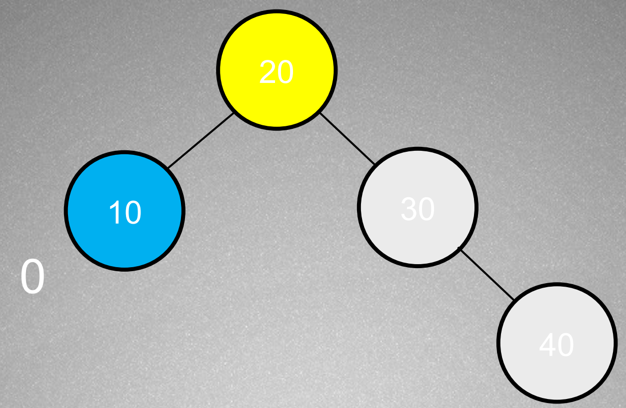


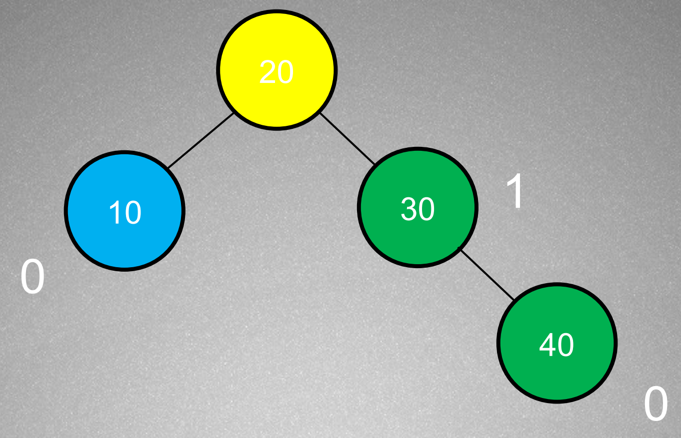

20
10
30
40


20
10
30
40

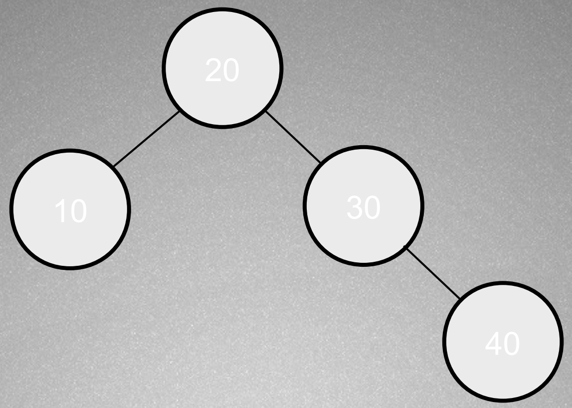

20
10
30
NULL
-1
40
0

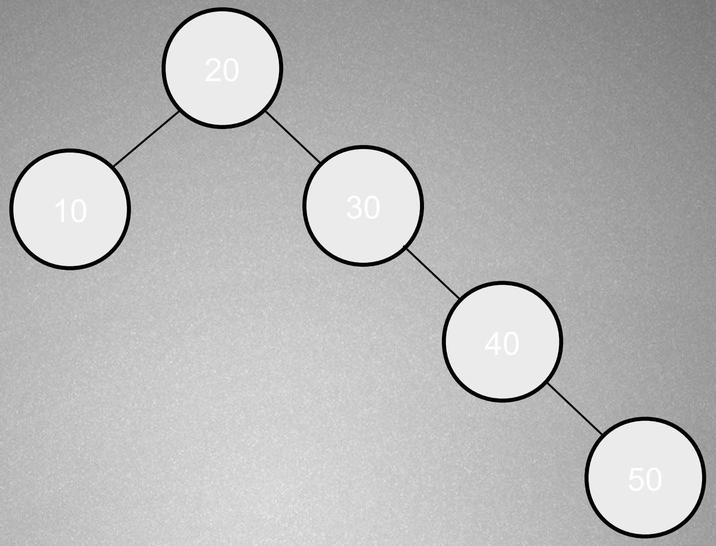

20
10
30
40

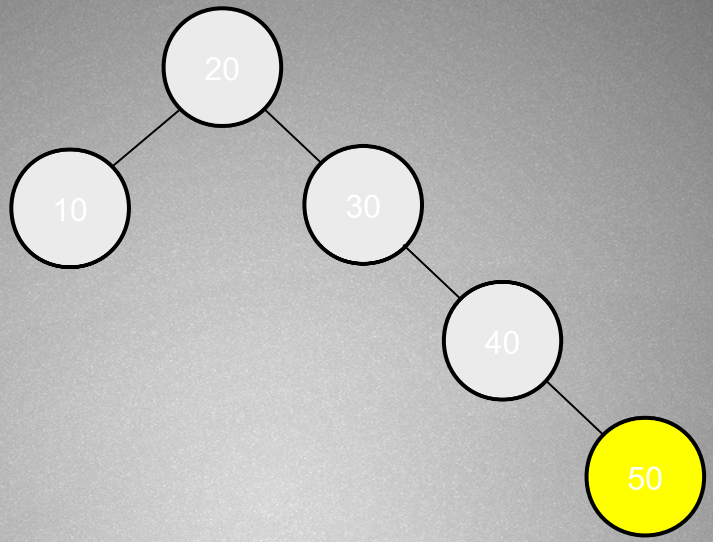

20
10
30
40
0

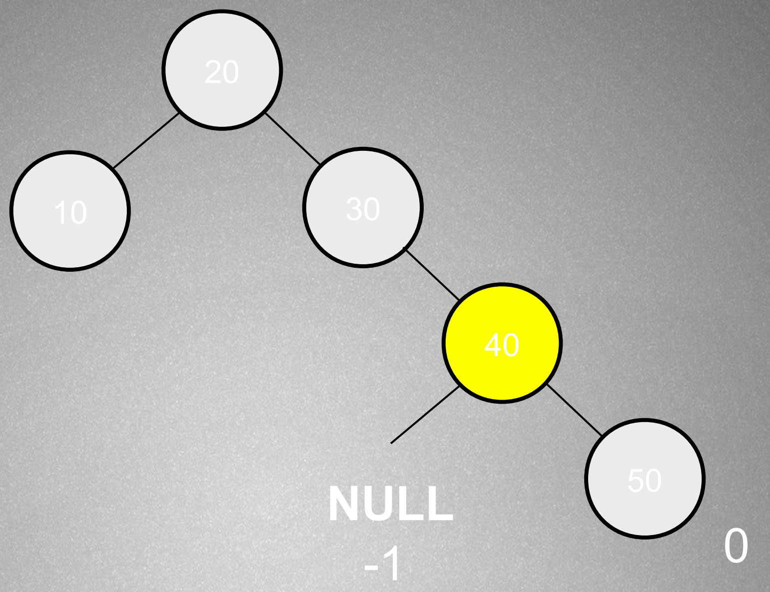

20
10
30
40
0
1
0

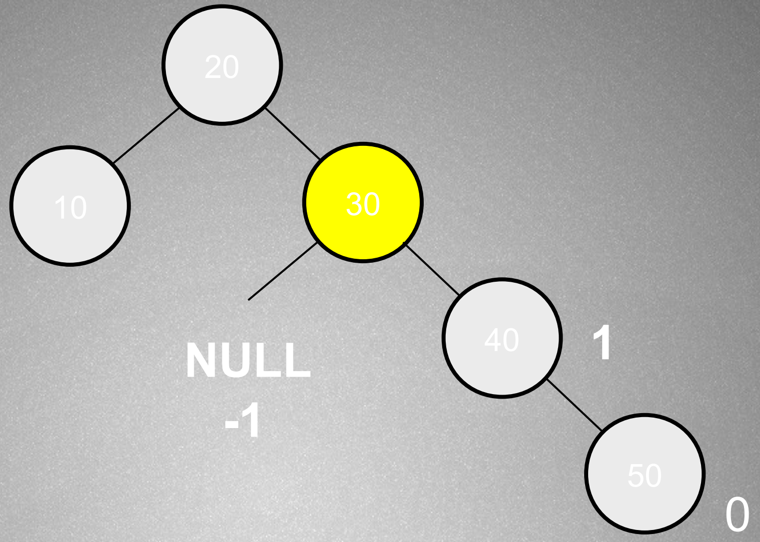
balancedTree.insert(50);



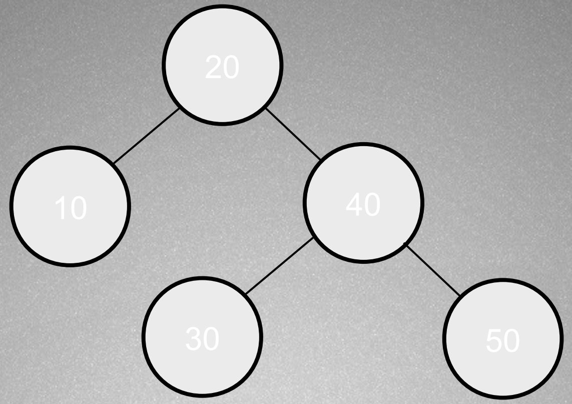

20
10
30
40
50


20
10
30
40
50

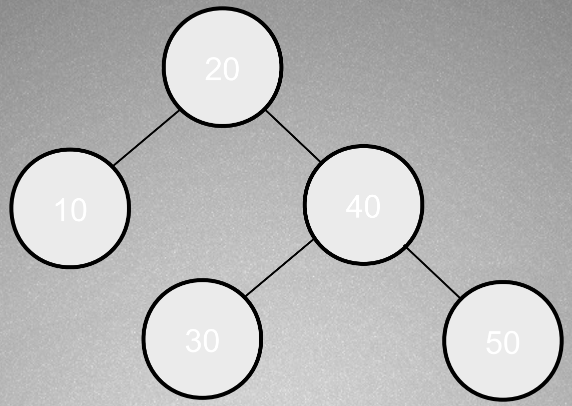

20
10
30
40
50
NULL
-1
0



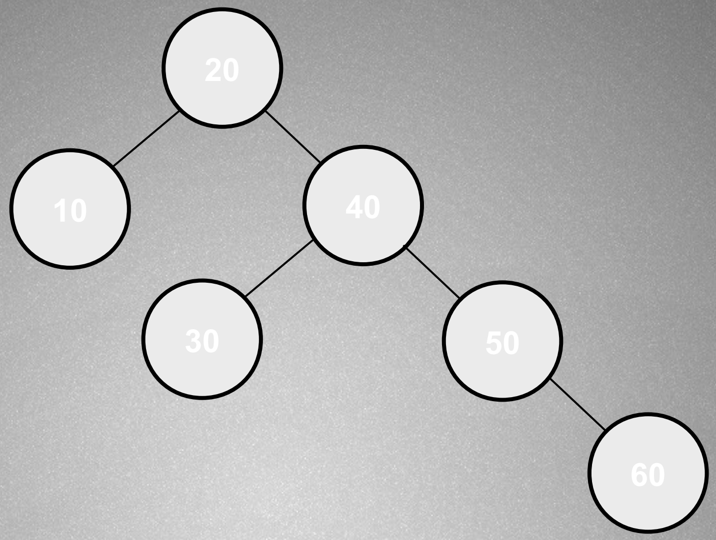
The difference between the height parameters is greater than 1 → rotations are needed !!!

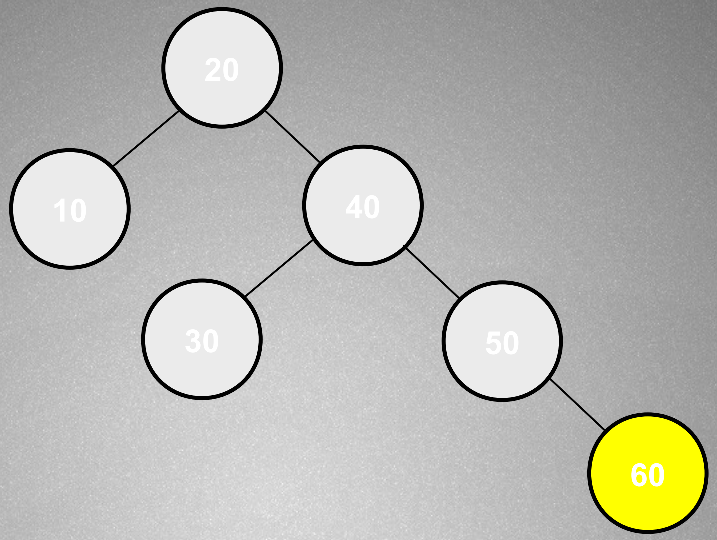

20
10
40
30
50

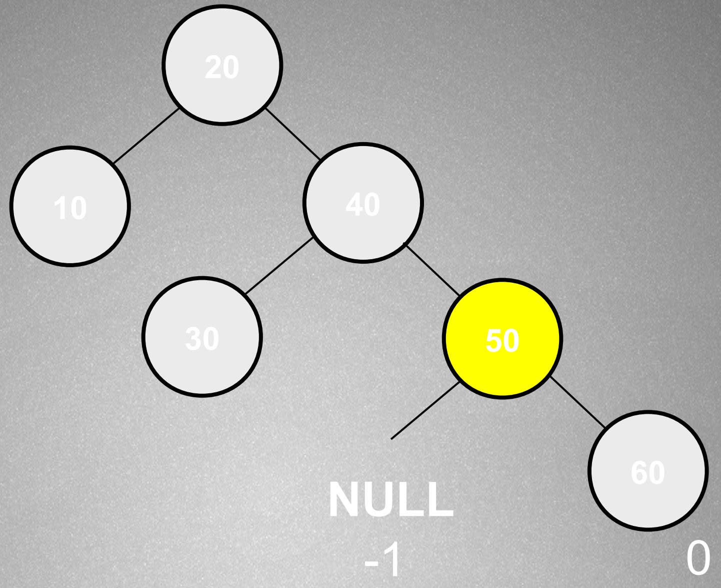
balancedTree.insert(60);



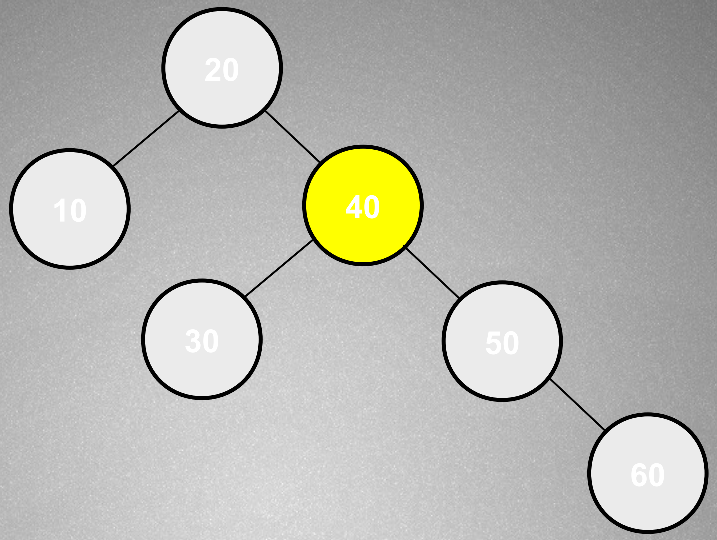
balancedTree.insert(60);

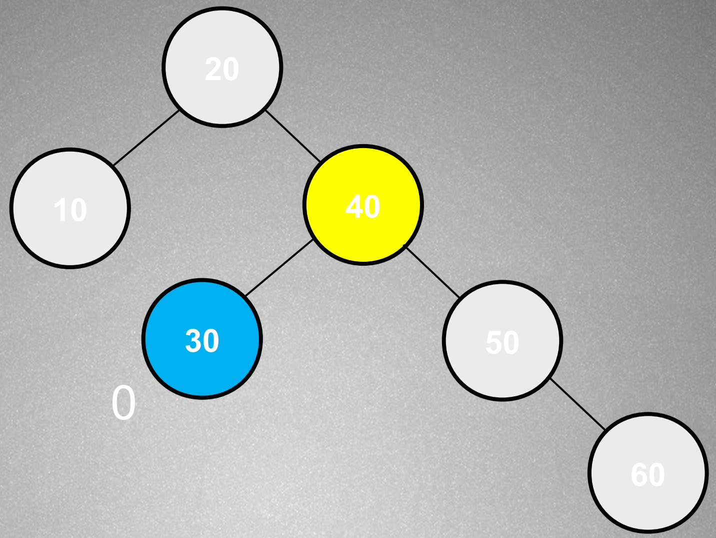


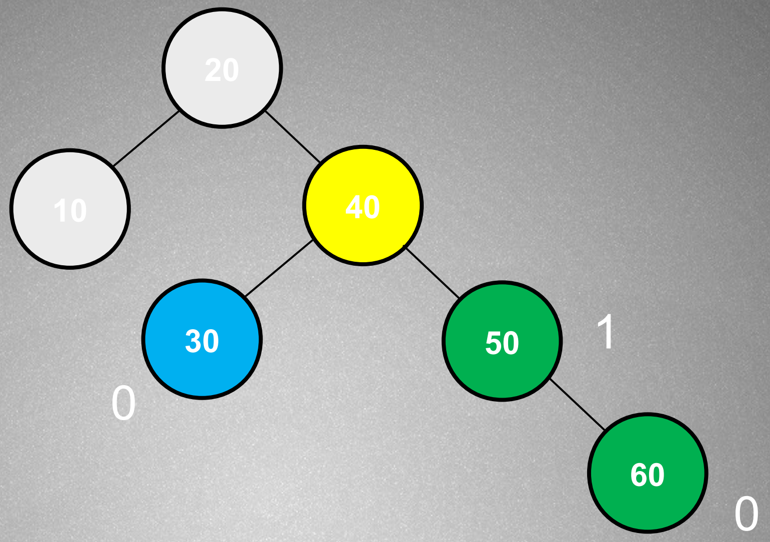

20
10
40
30
50
60

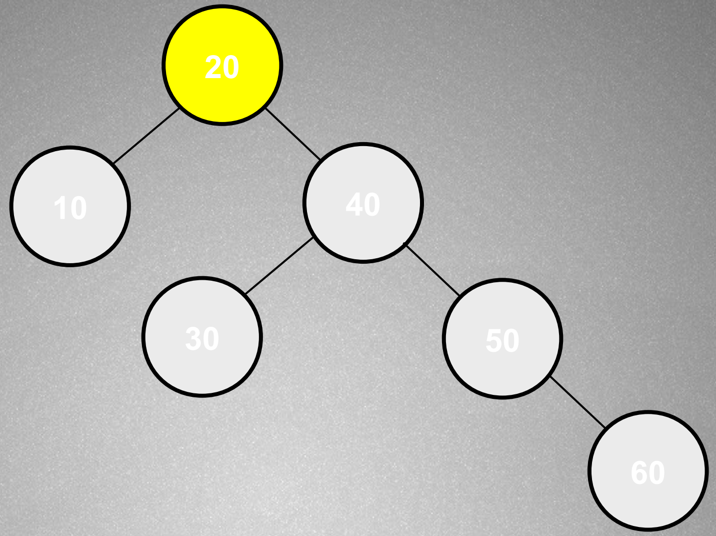


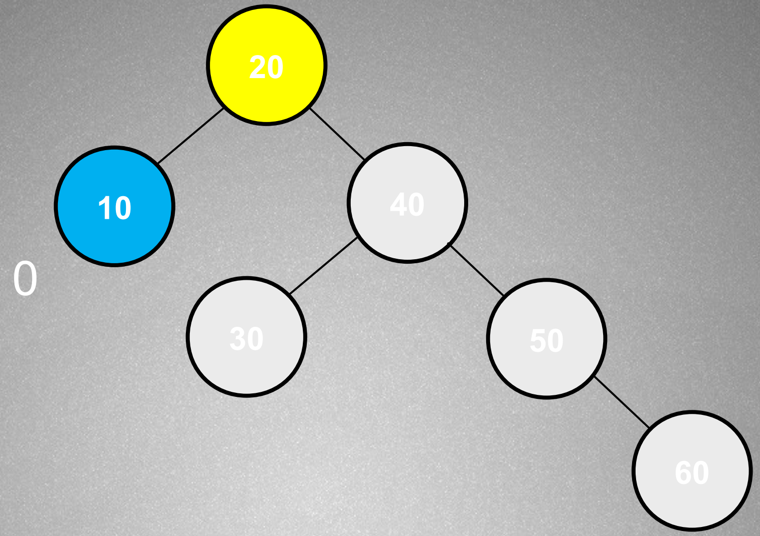
20
10
40
30
50
60
NULL
-1
0

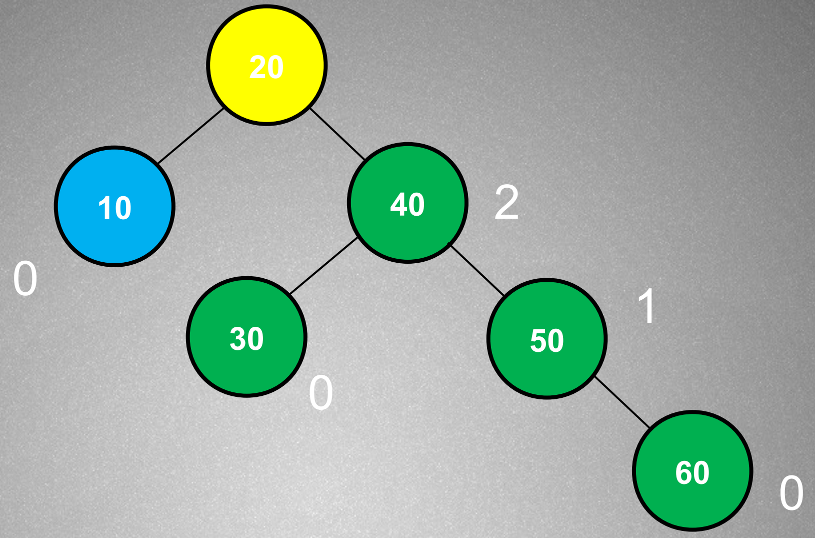

20
10
40
30
50
60


20
10
40
30
50
60
0

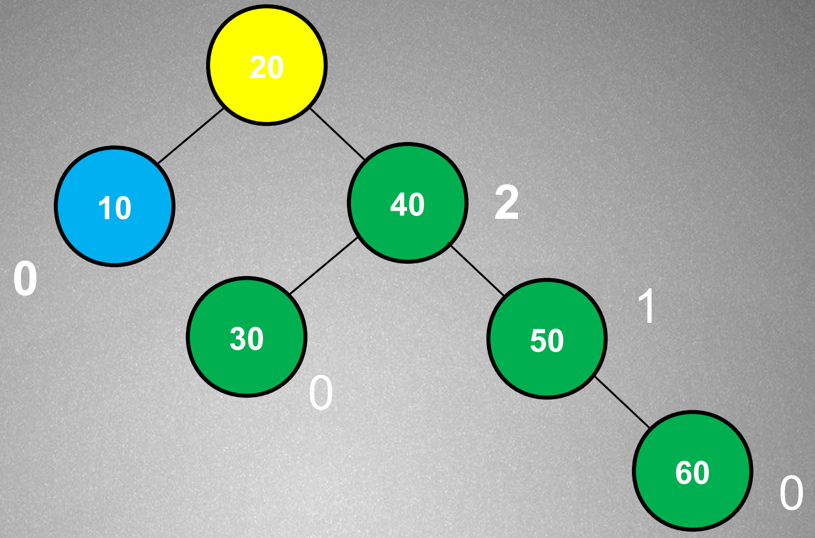

20
10
40
30
50
60
1
0
0


20
10
40
30
50
60

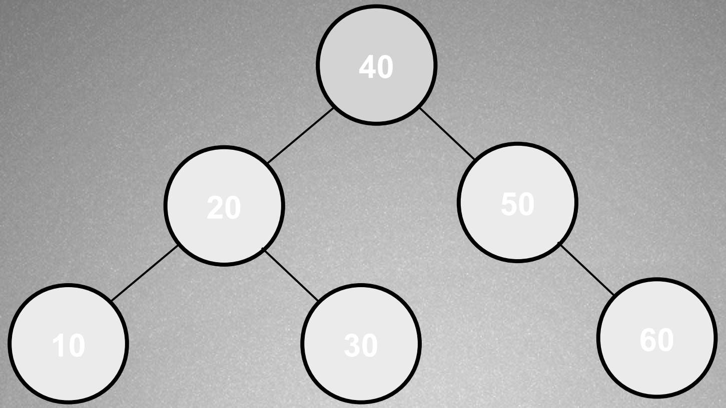

20
10
40
0
30
50
60



20
10
40
2
0
30
50
1
0
60
0



The difference between the height parameters
is greater than 1 → rotations are needed !!!


40
20
50
10
30
60

AVL TREES

REMOVE

Delete: soft delete → we do not remove the node from the BST we just mark that it has been removed

~ not so efficient solution

Delete: soft delete → we do not remove the node from the BST we just mark that it has been removed

~ not so efficient solution

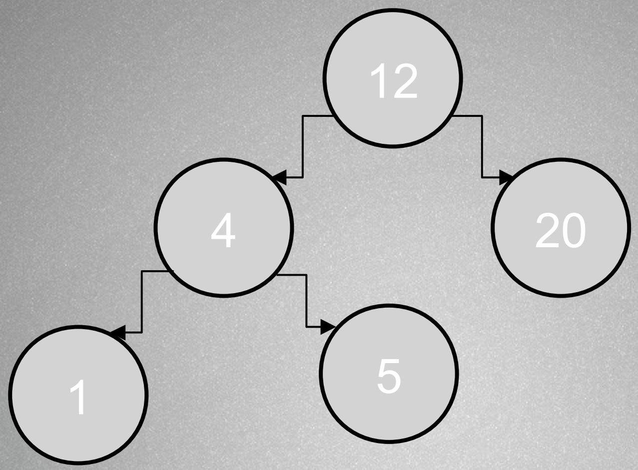
In the main **three** possible cases:

1.) The node we want to get rid of is a leaf node

2.) The node we want to get rid of has a single child

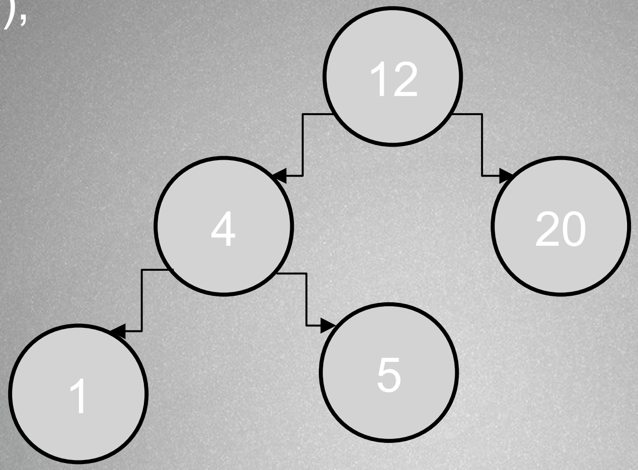
3.) The node we want to get rid of has 2 children

Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)



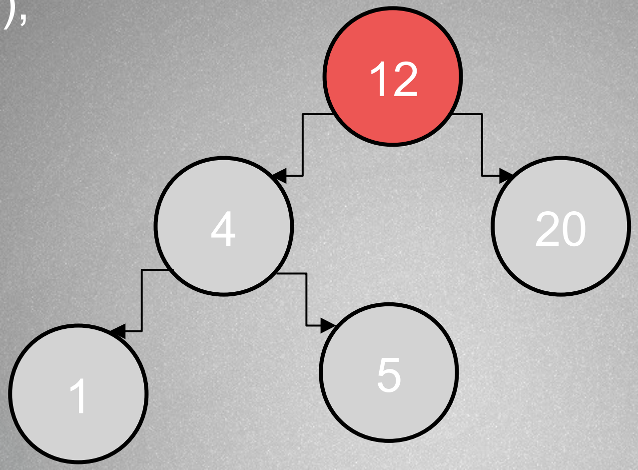
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



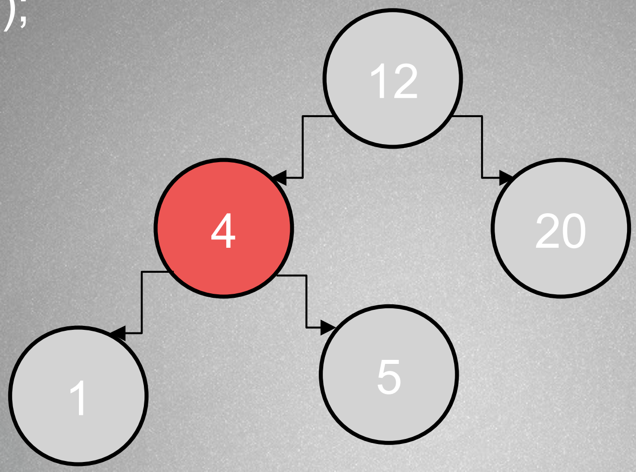
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



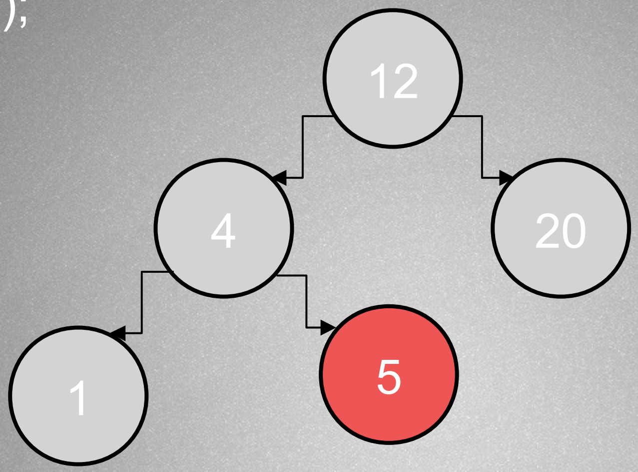
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



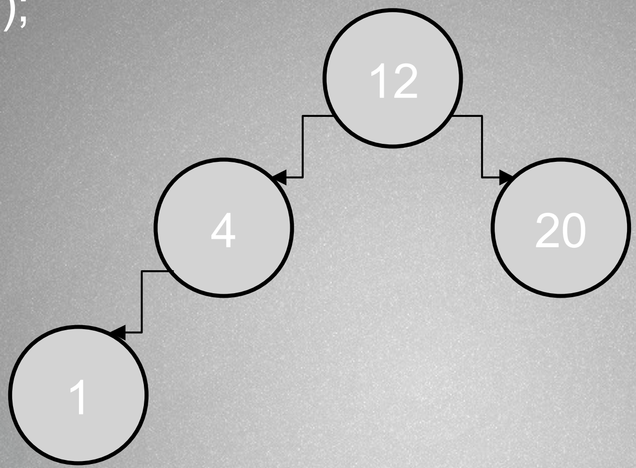
Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);



Delete: 1.) We want to get rid of a leaf node: very simple, we just have to remove it (set it to null whatever)

binarySearhTree.remove(5);

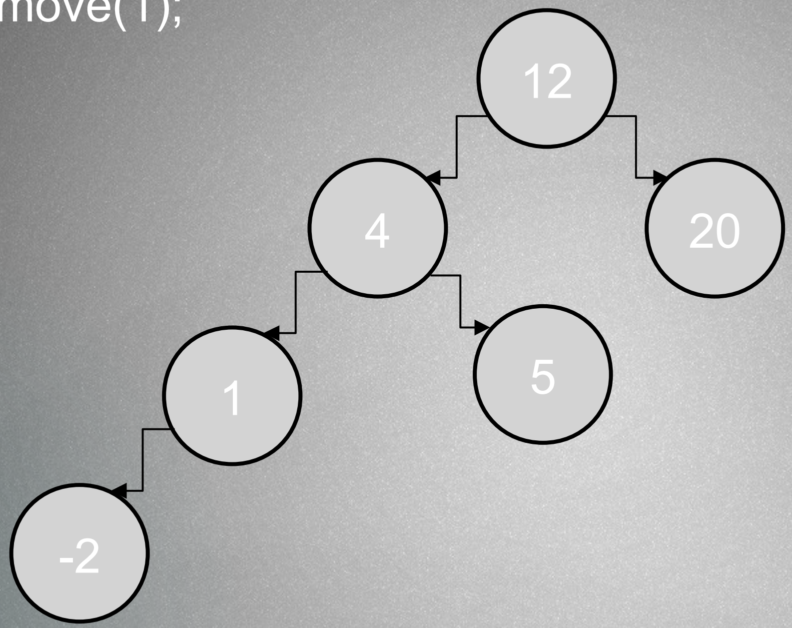


Complexity: we have to find the item itself + we have to delete it or set it to NULL

~ **O(logN)** find operation + **O(1)** deletion = **O(logN)** !!!

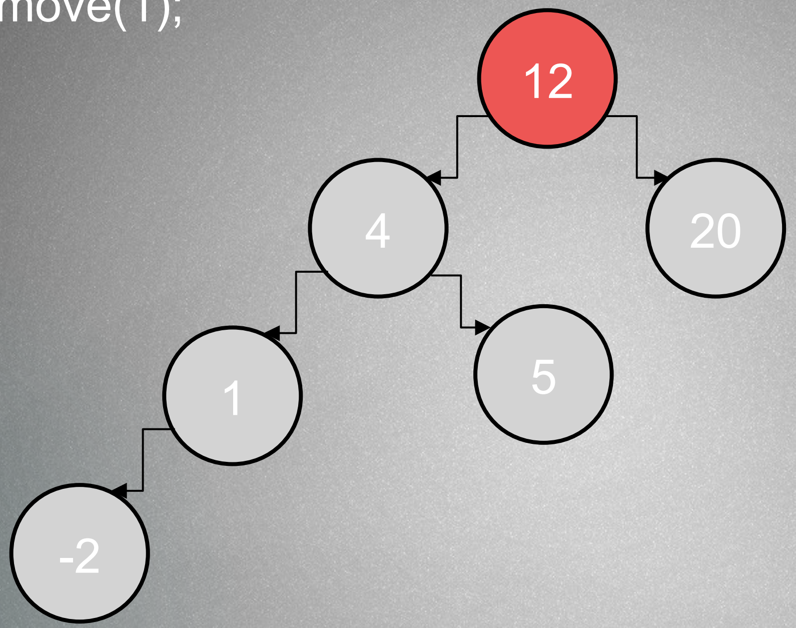
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



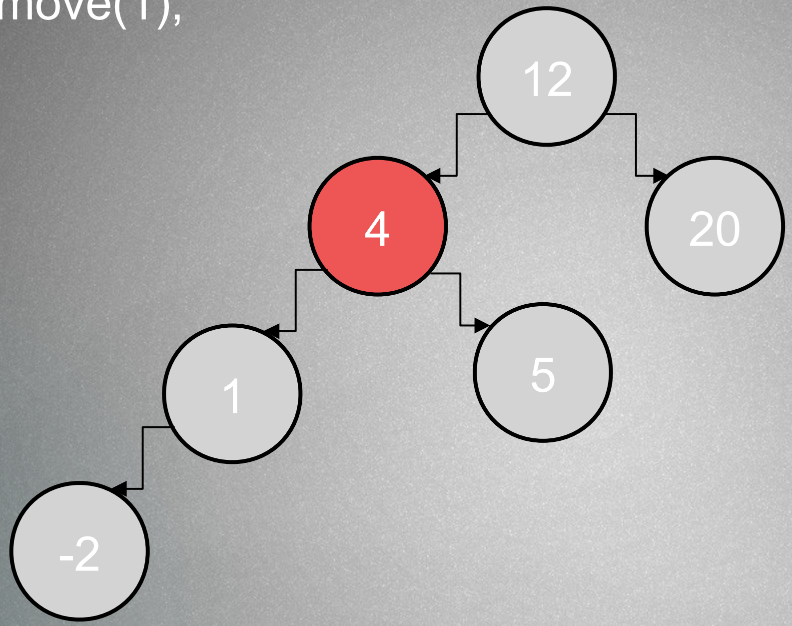
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



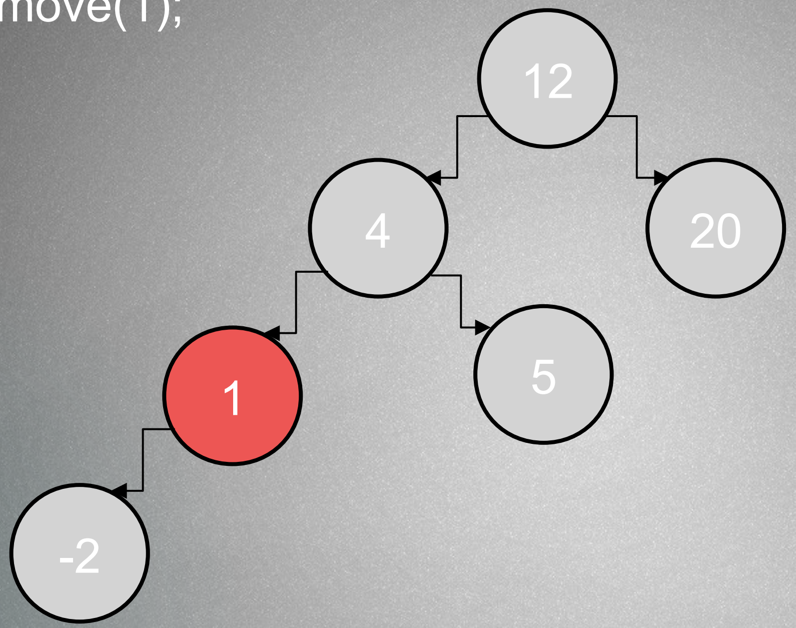
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



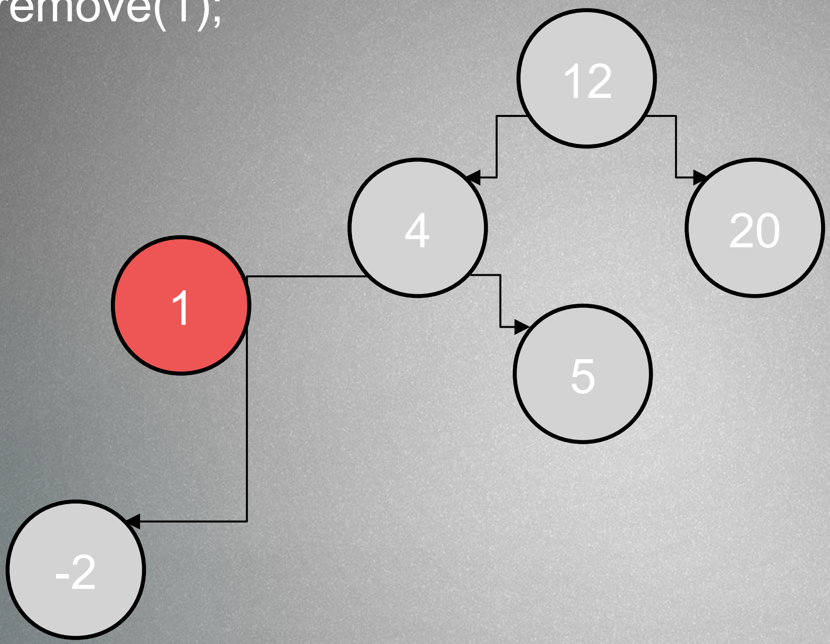
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



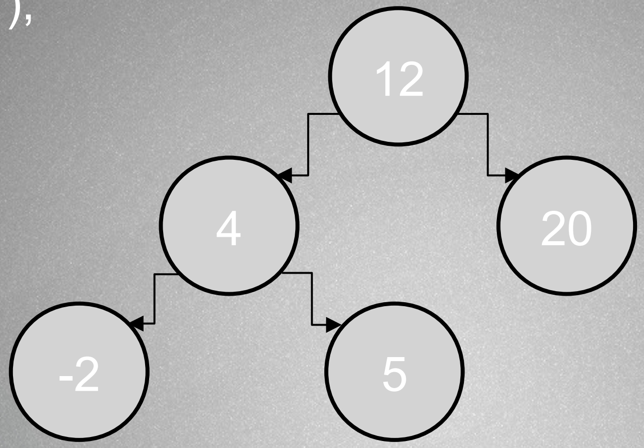
Delete: 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



<u>Delete:</u> 2.) We want to get rid of a node that has a single child, we just have to update the references

binarySearhTree.remove(1);



Complexity: first we have to find the item we want to get rid of and we have to update the references

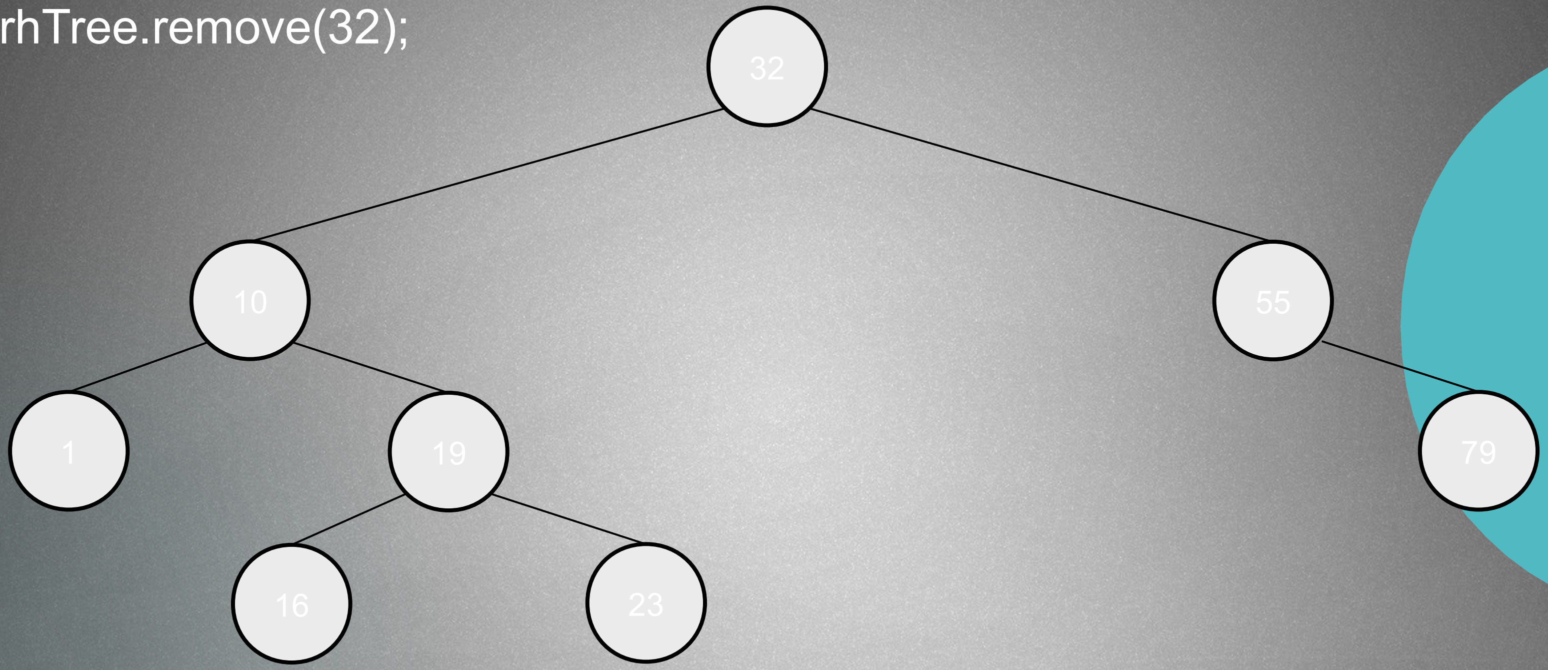
~ set parent's pointer point to it's grandchild directly

O(logN) find operation + **O(1)** update references = **O(logN)** !!!

Delete: 3.) We want to get rid of a node that has two children

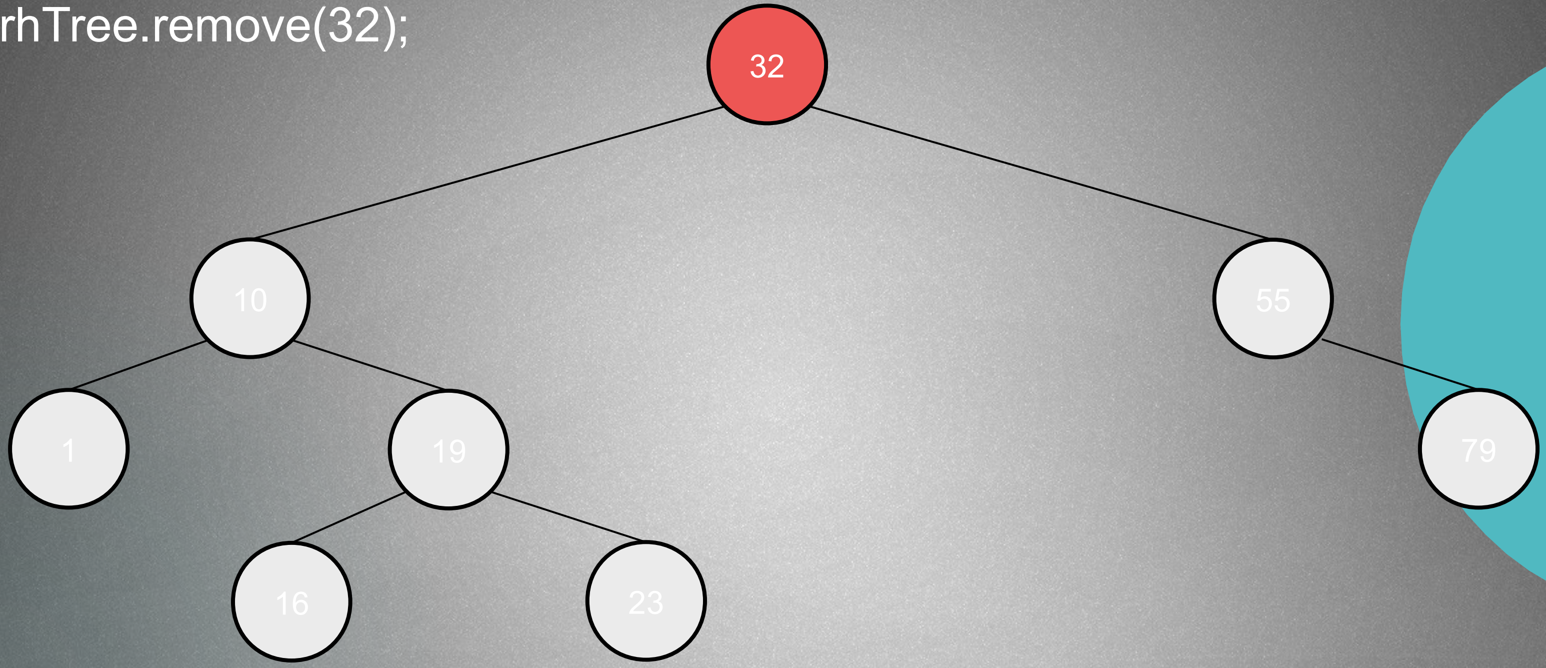
Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);



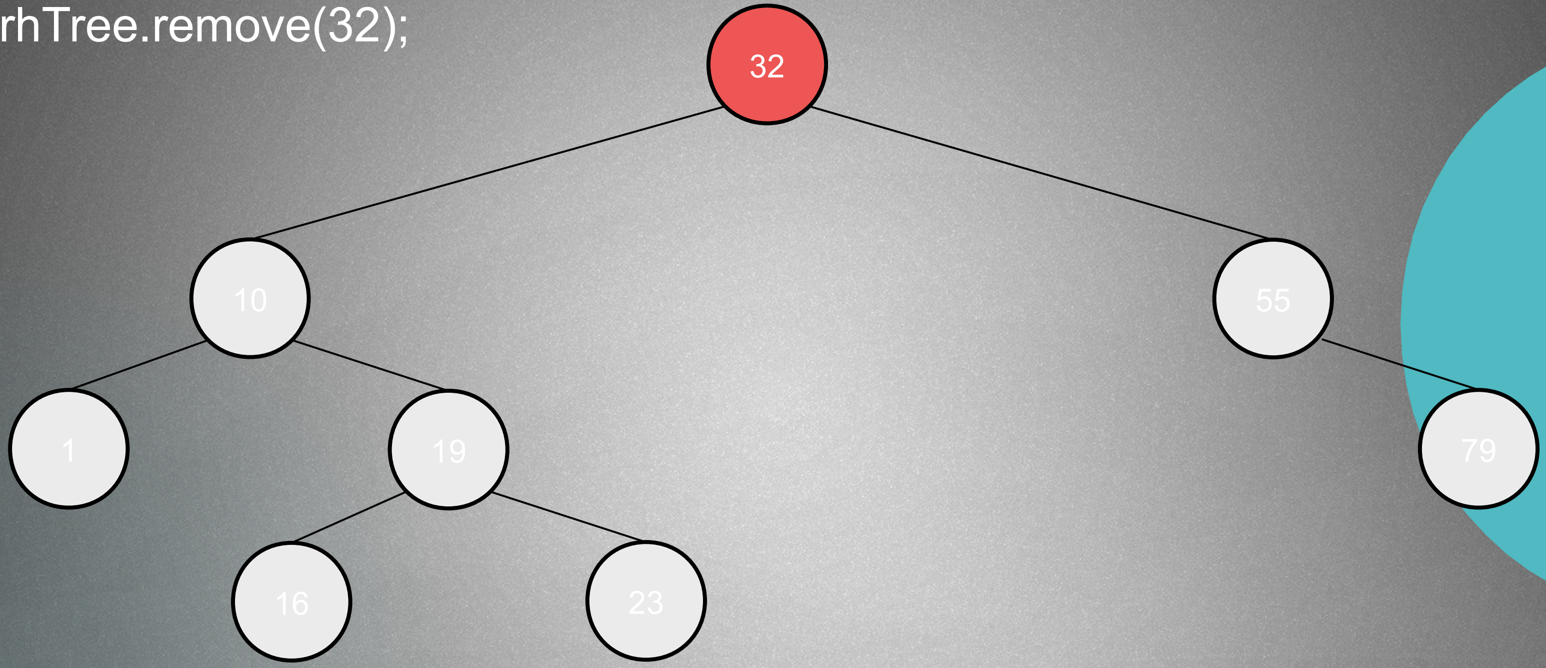
Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);



Delete: 3.) We want to get rid of a node that has two children

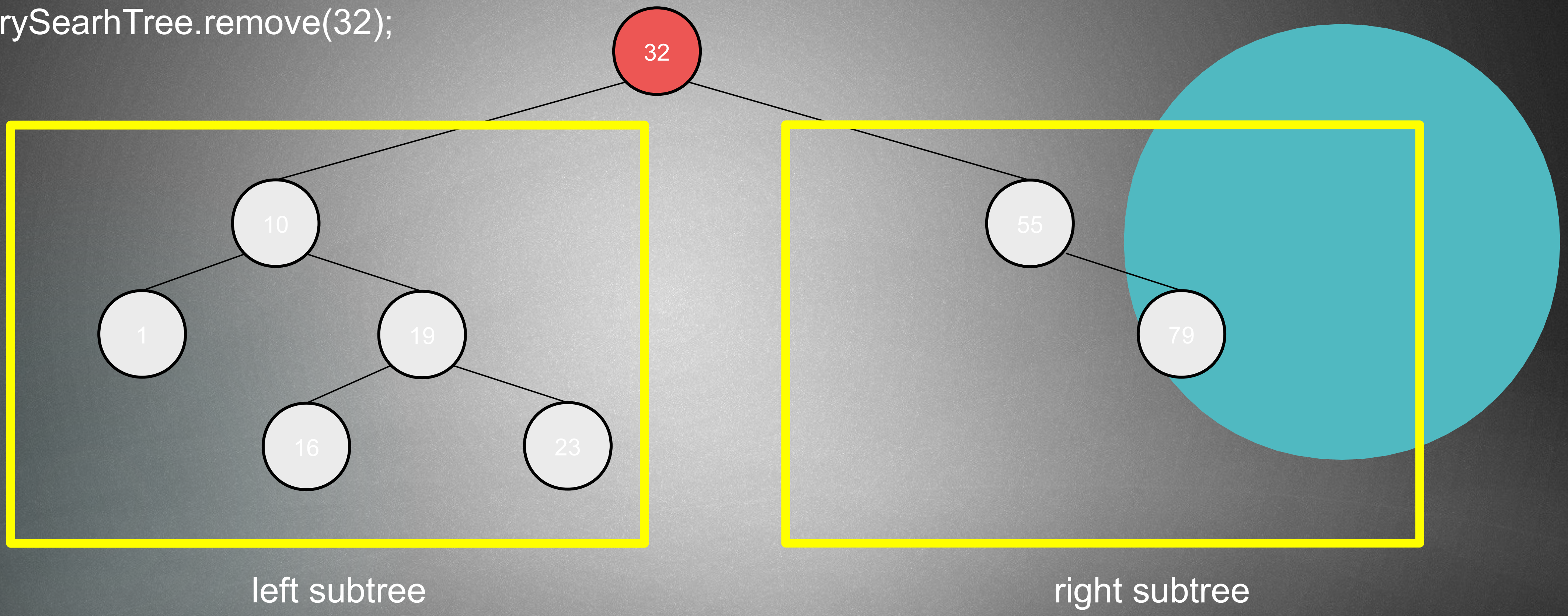
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

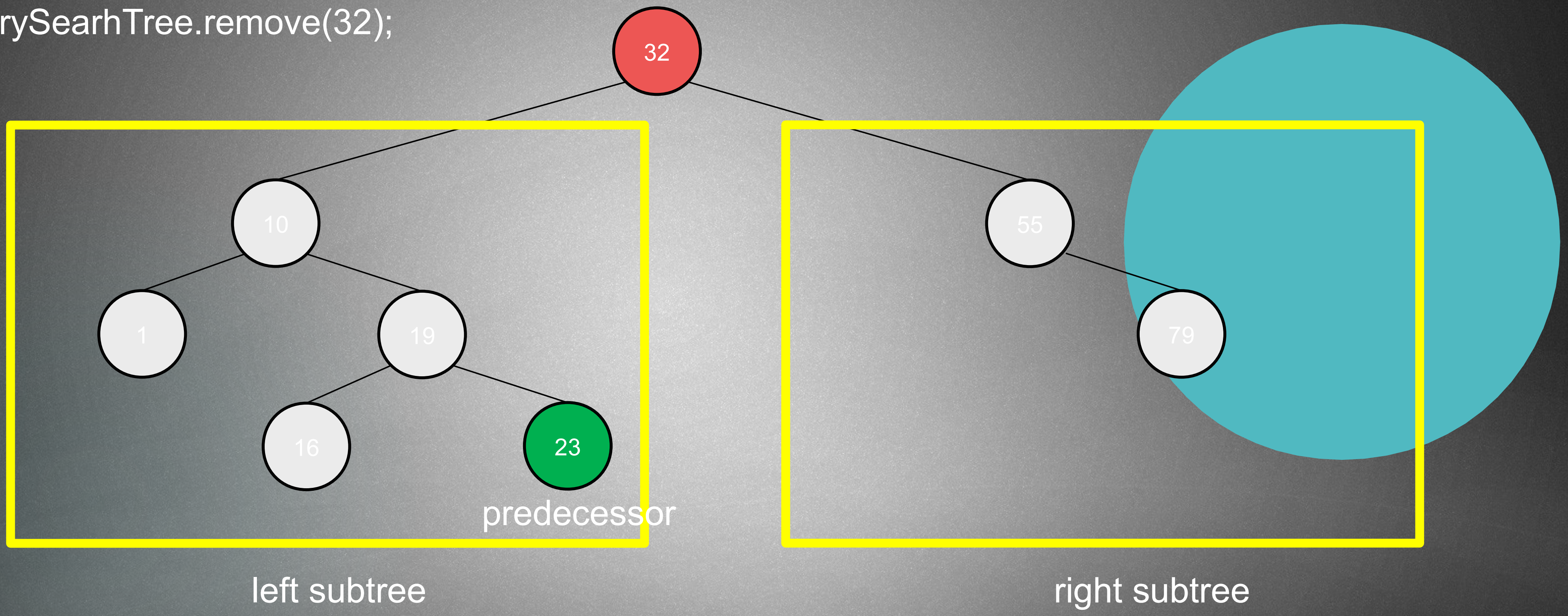
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

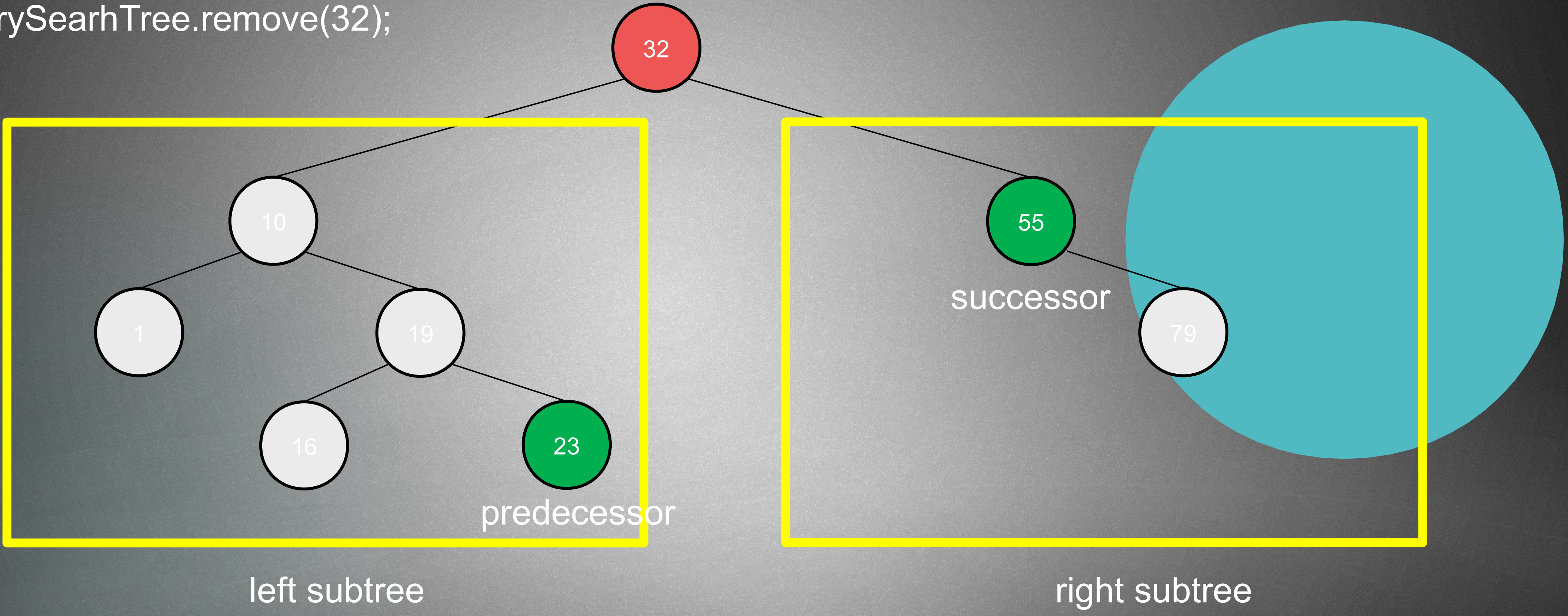
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

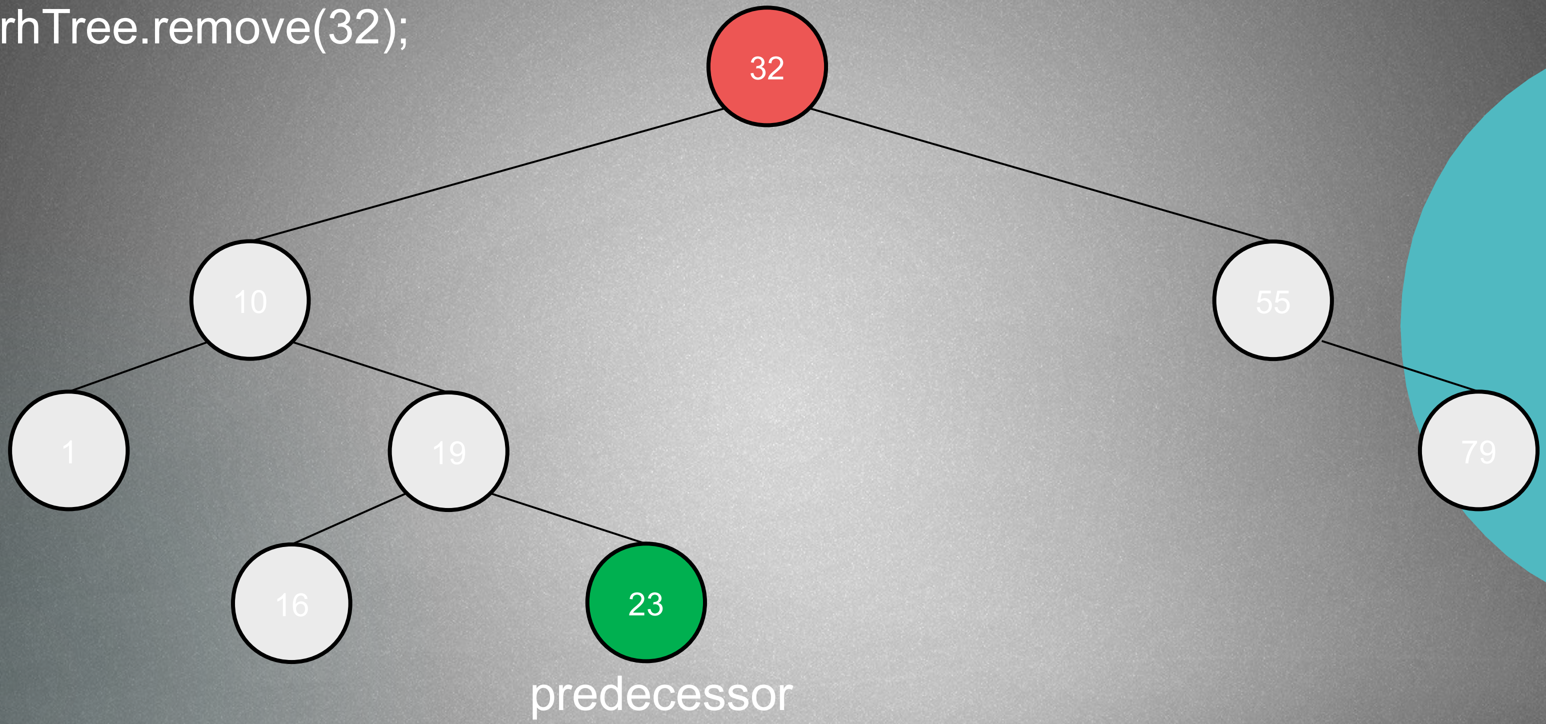
binarySearhTree.remove(32);



We have two options: we look for the largest item in the left subtree
OR the smallest item in the right subtree !!!

Delete: 3.) We want to get rid of a node that has two children

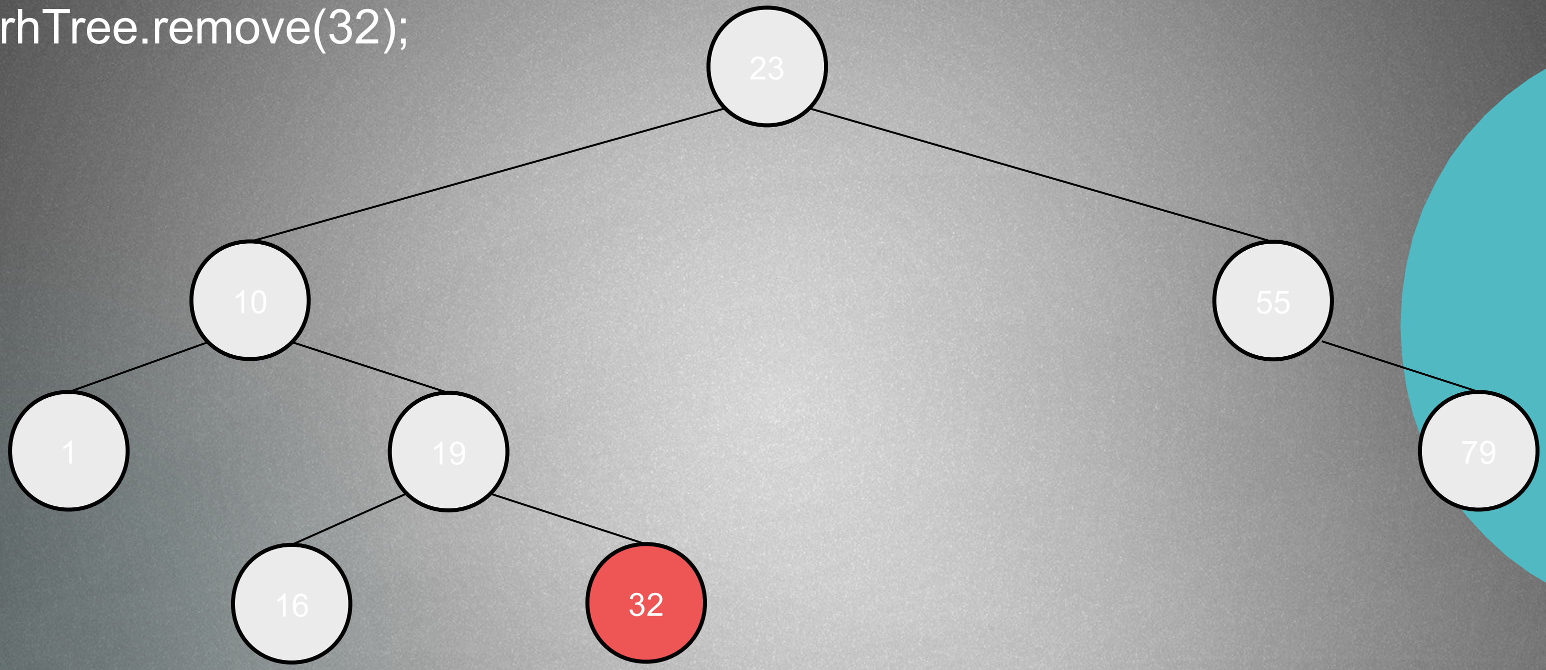
binarySearhTree.remove(32);



We look for the predecessor and swap the two nodes !!!

Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);

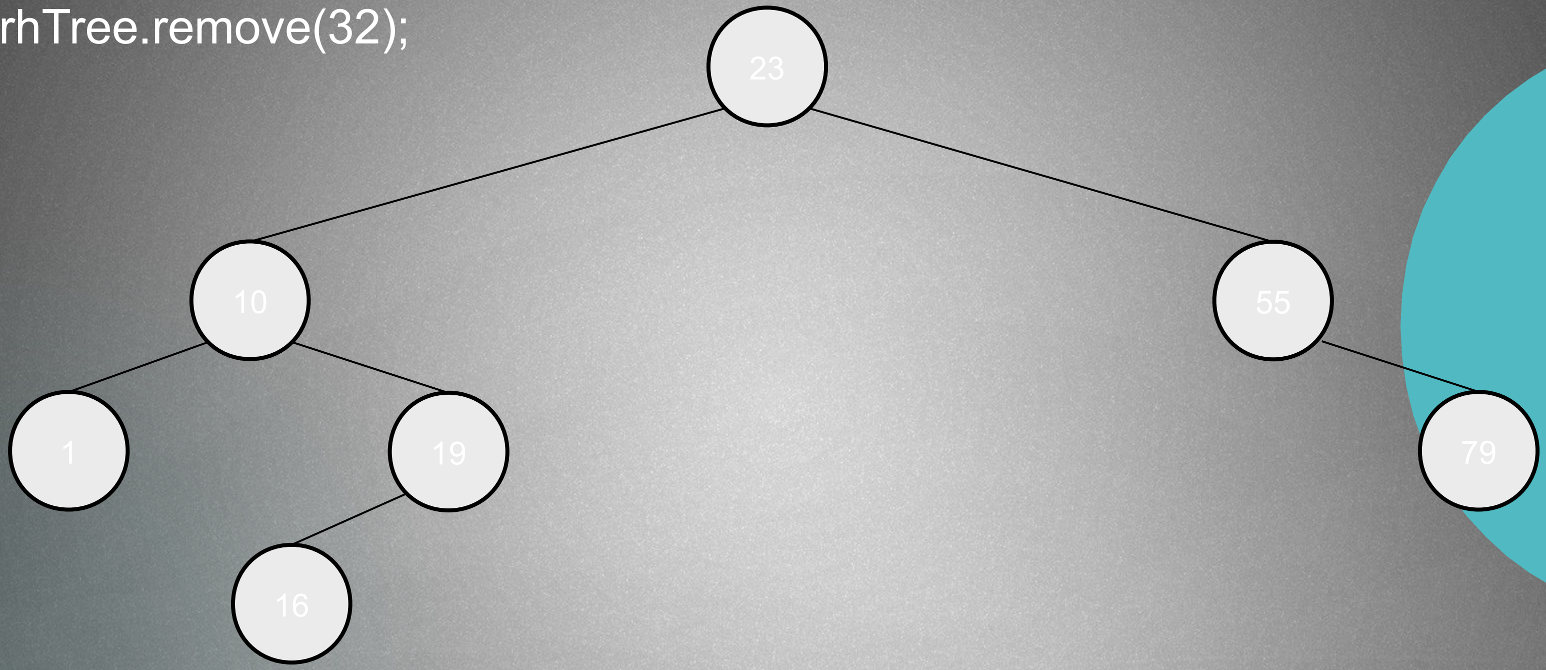


We look for the predecessor and swap the two nodes !!!

We end up at a case 1.) situation: we just have to set it to NULL

Delete: 3.) We want to get rid of a node that has two children

binarySearhTree.remove(32);

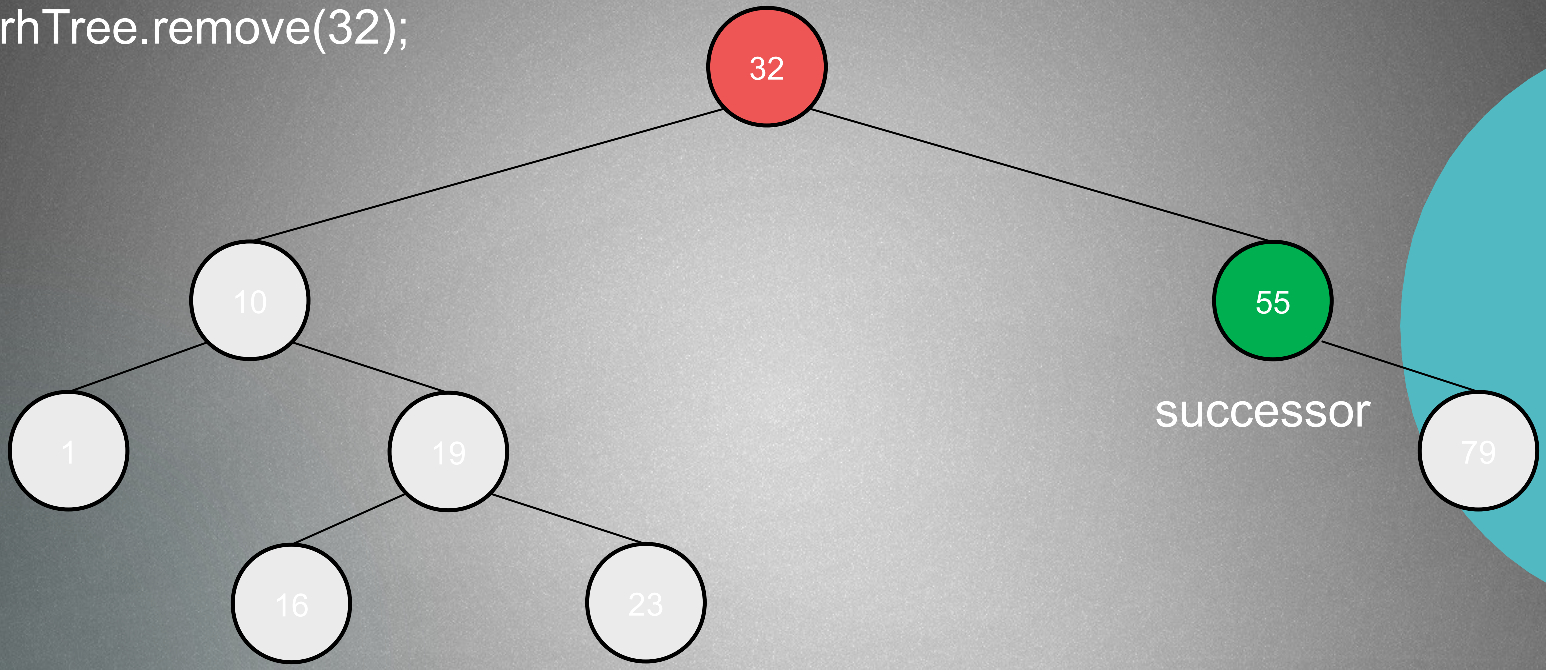


We look for the predecessor and swap the two nodes !!!

We end up at a case 1.) situation: we just have to set it to NULL

Delete: 3.) We want to get rid of a node that has two children

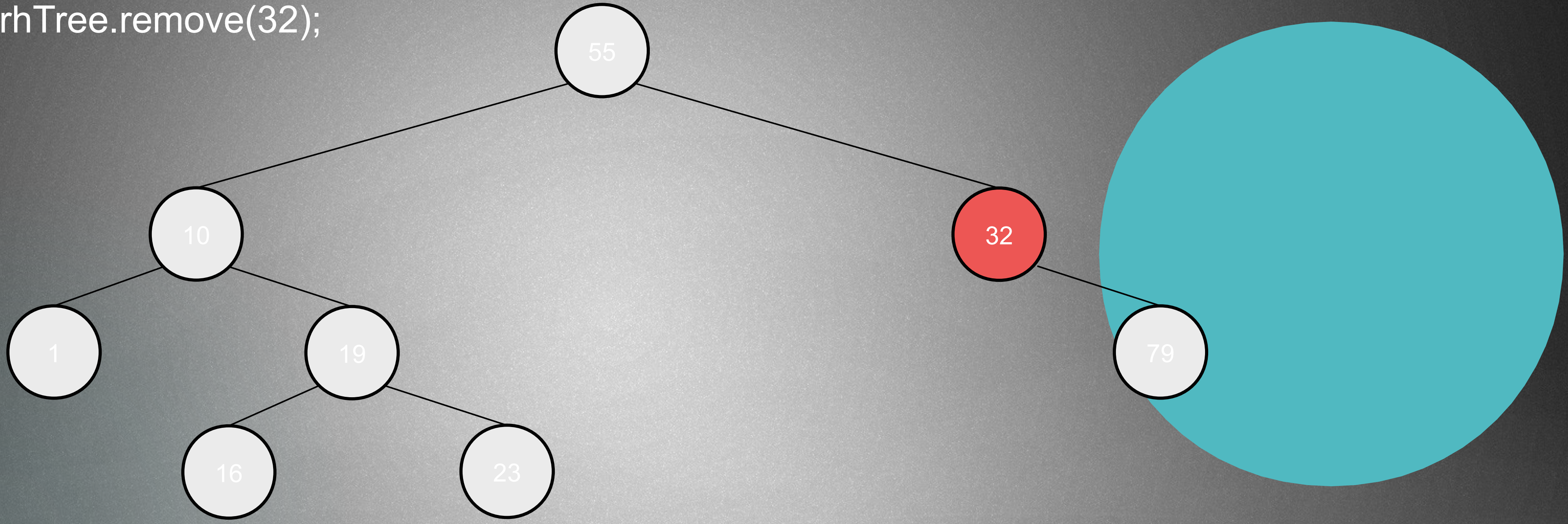
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!

Delete: 3.) We want to get rid of a node that has two children

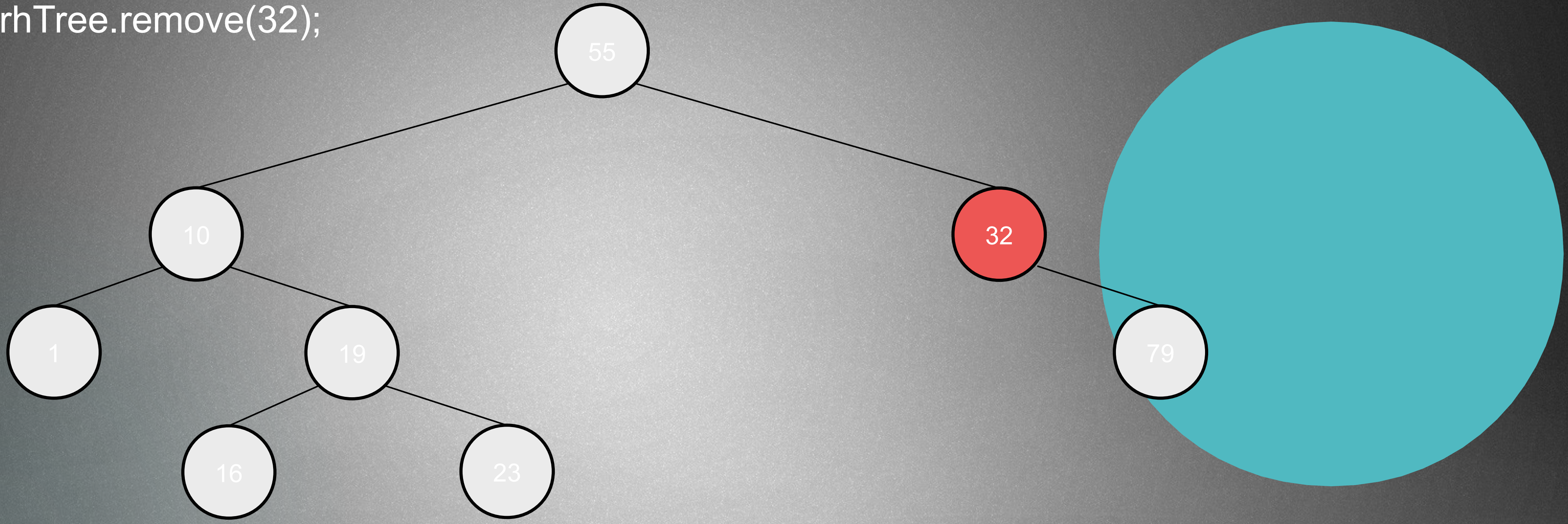
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!

Delete: 3.) We want to get rid of a node that has two children

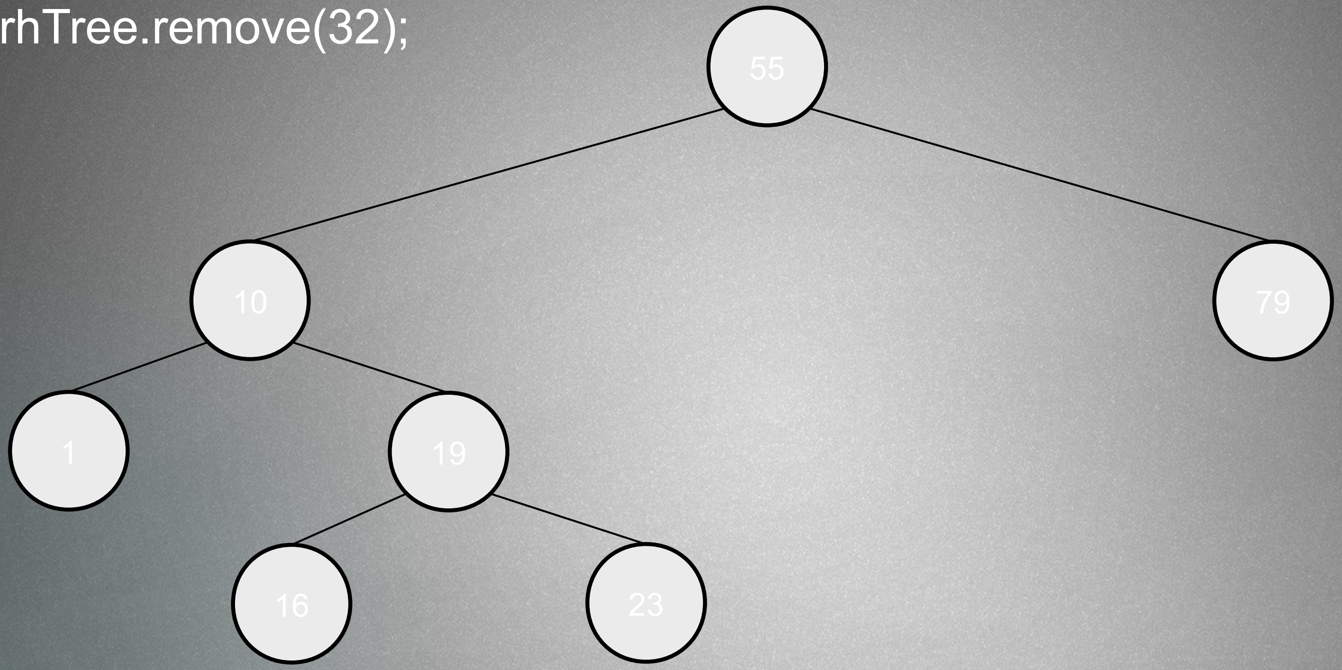
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!
This becomes the Case 2.) situation, we just have to update the references

Delete: 3.) We want to get rid of a node that has two children

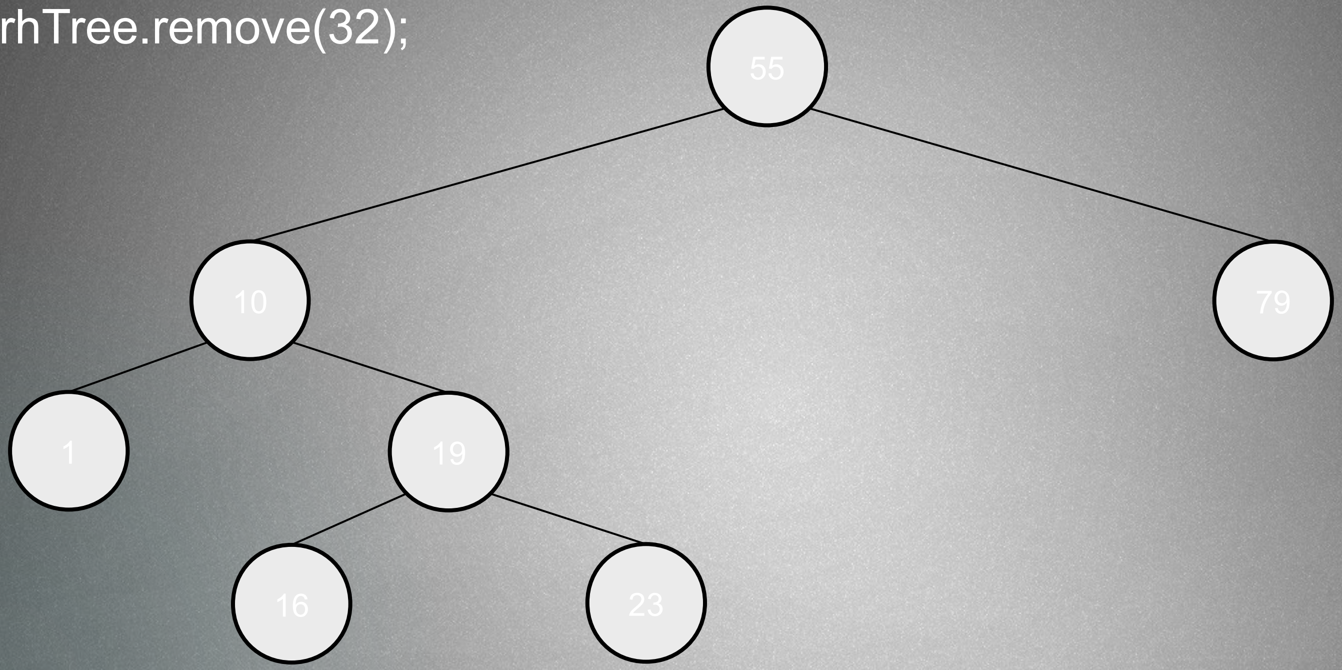
binarySearhTree.remove(32);



Another solution → we look for the successor and swap the two nodes !!!
This becomes the Case 2.) situation, we just have to update the references

Delete: 3.) We want to get rid of a node that has two children

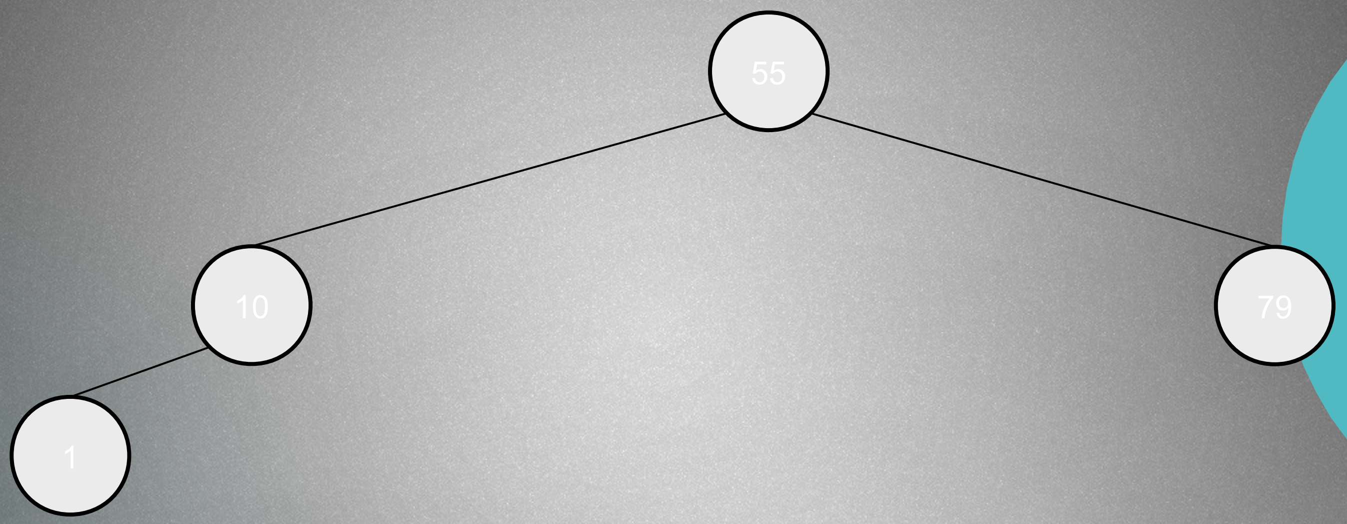
binarySearhTree.remove(32);



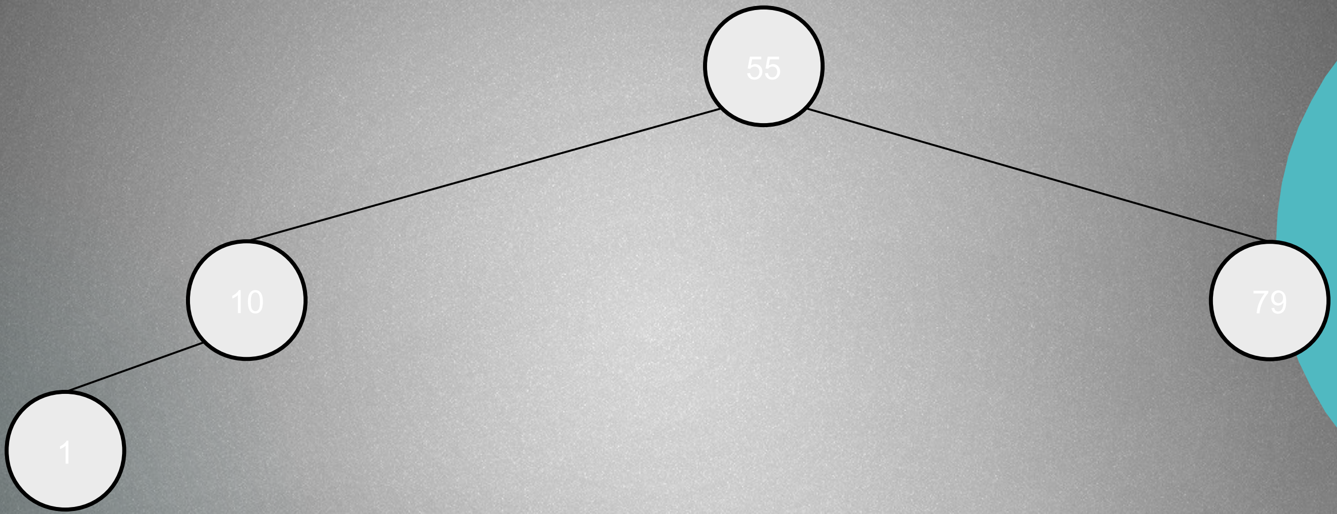
Complexity: **O(logN)**

Conclusion

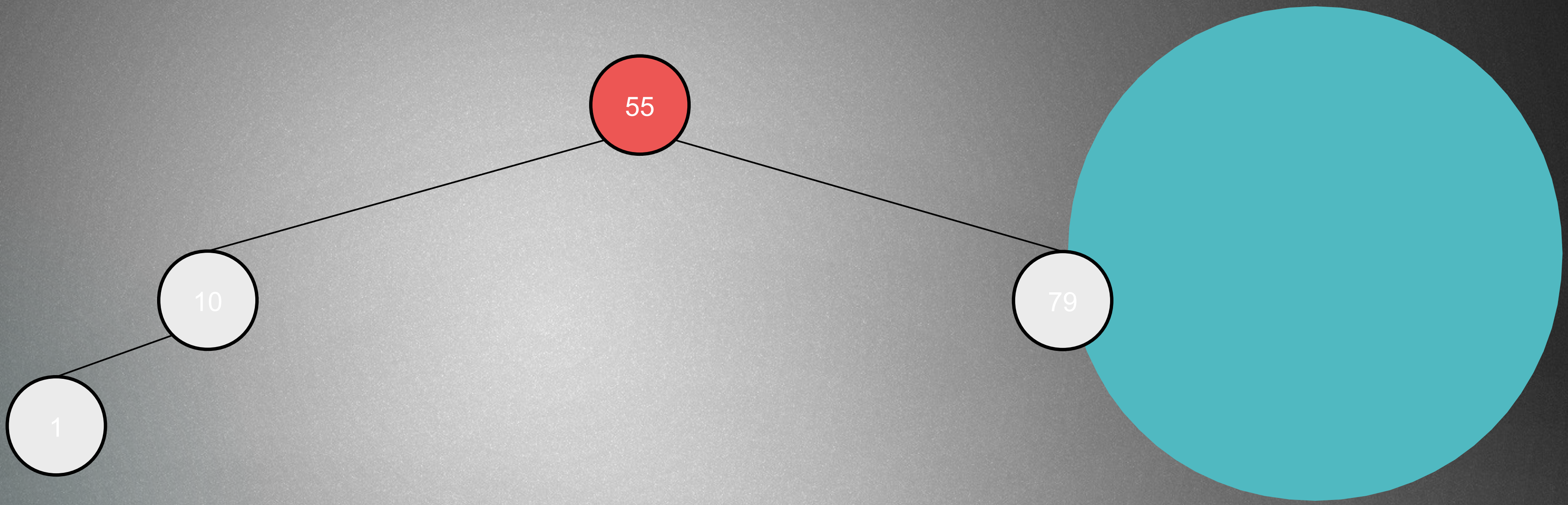
- It is basically the same as we have seen for simple binary search tree node deletion
- BUT there is a problem
- When we remove a node → it may get unbalanced because of that given node is no more in the tree


55
10
79
1

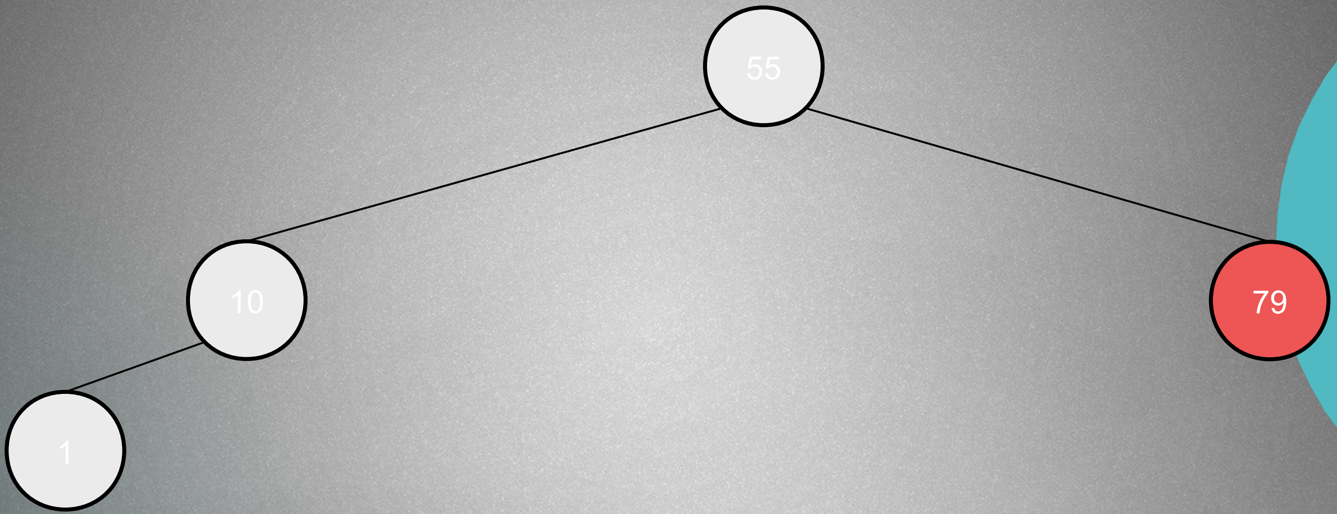
tree.remove(79);



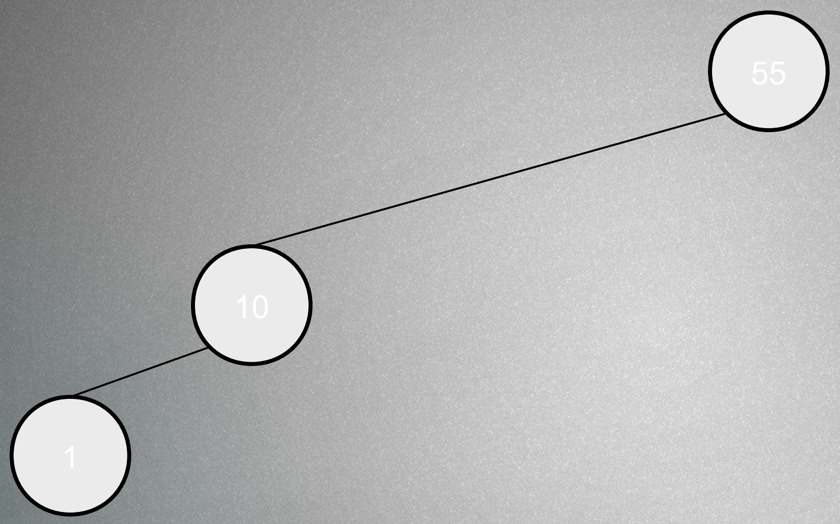
tree.remove(79);



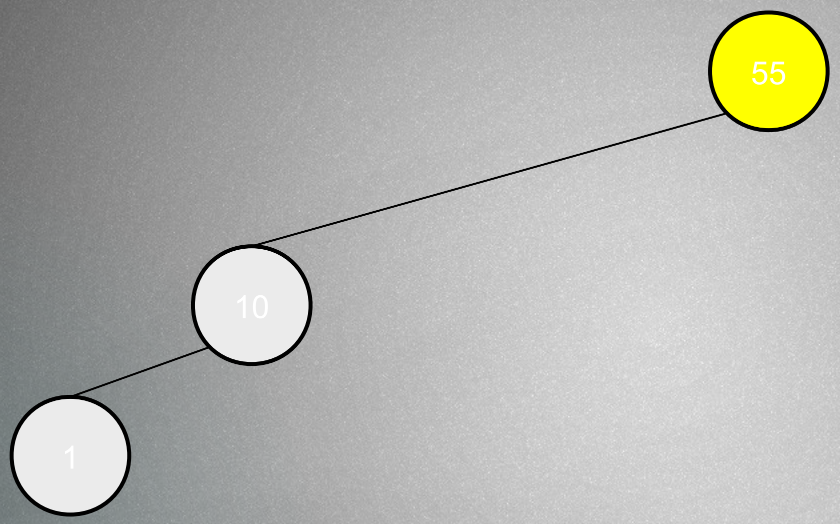
tree.remove(79);



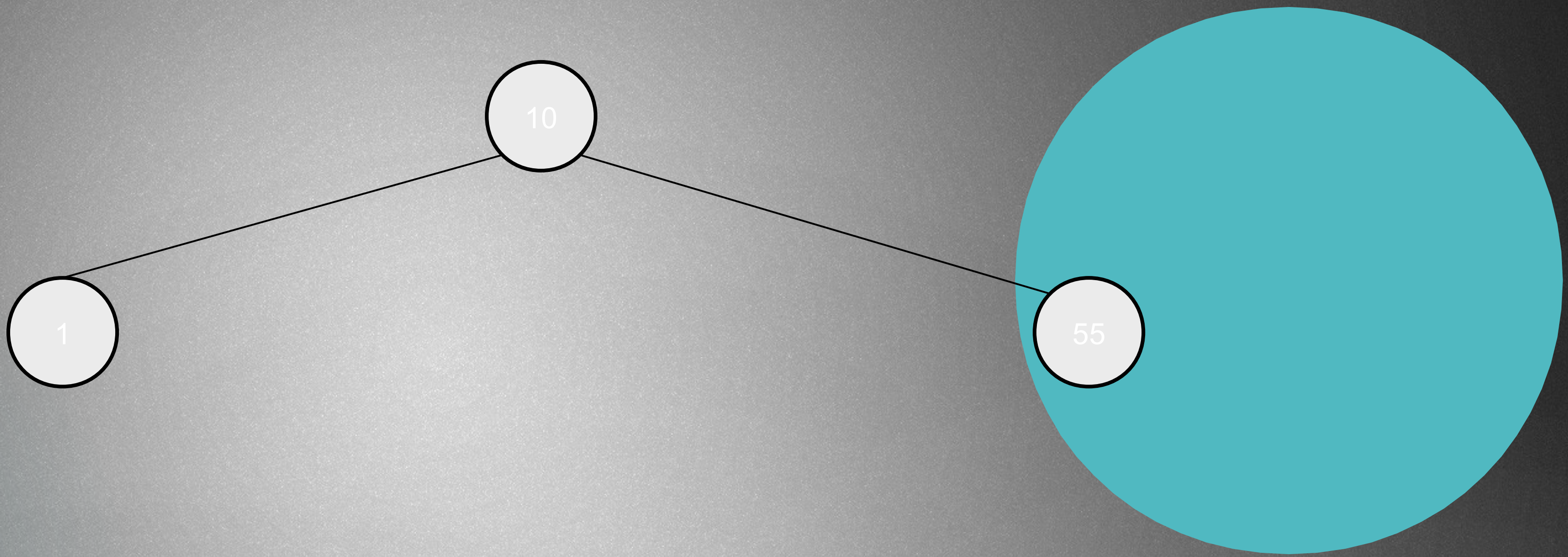
tree.remove(79);



tree.remove(79);



tree.remove(79);



AVL TREES

BALANCED TREES

AVL sort

- We can use this data structure to sort items
- We just have to insert the **N** items we want to sort
- We have to make an in-order traversal → it is going to yield the numerical or alphabetical ordering !!!

Insertion: O(N*logN)

In-order traversal: O(N)

Overall complexity: **O(N*logN)**

Applications

- Databases when deletions or insertions are not so frequent, but have to make a lot of look-ups
- Look-up tables usually implemented with the help of hashtables BUT AVL trees support more operations in the main
- We can sort with the help of AVL trees !!!
- // red-black trees are a bit more popular because for AVL trees we have to make several rotations ~ a bit slower