uc3m | Universidad Carlos III de Madrid

Data Structures and Algorithms.
Author: Isabel Segura Bedmar

Unit 5 – Trees

Problem - Implementation of a Binary Tree. This implementation should include the following methods: size, height, depth, preorder, inorder, postorder and levelorder, studied in class.

Solution:

```
import queue #it is Python module to implement queues

class Node:

  def __init__(self,elem=None):
    self.elem=elem
    self.leftChild=None
    self.rightChild=None
    self.parent=None


class BinaryTree:

  def __init__(self):
    self.root=None

  def draw(self):
      """Draw a tree"""
      self._draw('',self.root,False)
      print()

  def _draw(self,prefix, node, isLeft):
    if node !=None:
        self._draw(prefix + "     ", node.rightChild, False)
        print(prefix + ("|-- ") + str(node.elem))
        self._draw(prefix + "     ", node.leftChild, True)
```

```
  def size(self):
    """Returns the number of nodes"""
    return self._size(self.root)

  def _size(self,currentNode):
    if currentNode==None:
      return 0

    return 1 + self._size(currentNode.leftChild) +
self._size(currentNode.rightChild)


  def height(self):
    """Returns the height of the tree"""
    return self._height(self.root)

  def _height(self,currentNode):
    if currentNode==None:
      return -1

    return 1 + max(self._height(currentNode.leftChild),
self._height(currentNode.rightChild))

  def depth(self,currentNode):
    """Returns the depth of a node"""

    if currentNode==None:
      return 0

    return 1 + self.depth(currentNode.parent)


  def preorder(self):
    print('pre-order traversal')
    self._preorder(self.root)
    print()

  def _preorder(self,currentNode):
    if currentNode!=None:
      print(currentNode.elem,end=' ')
      self._preorder(currentNode.leftChild)
      self._preorder(currentNode.rightChild)


  def postorder(self):
    print('post-order traversal')
    self._postorder(self.root)
    print()

  def _postorder(self,currentNode):
```

```
    if currentNode!=None:
      self._postorder(currentNode.leftChild)
      self._postorder(currentNode.rightChild)
      print(currentNode.elem,end=' ')



  def inorder(self):
    print('in-order traversal')
    self._inorder(self.root)
    print()

  def _inorder(self,currentNode):
    if currentNode!=None:
      self._inorder(currentNode.leftChild)
      print(currentNode.elem,end=' ')
      self._inorder(currentNode.rightChild)



  def levelorder(self):
    “””This methods shows the level-order of tree”””
    if self.root==None:
      print('tree is empty')
      return
    print('level-order traversal')
    q=queue.Queue()
    q.put(self.root) #we save the root

    while q.empty()==False:
      current=q.get() #dequeue
      print(current.elem, end=' ')
      if current.leftChild:
        q.put(current.leftChild)
      if current.rightChild:
        q.put(current.rightChild)

    print()
```

Problem - Implementation of a Binary Search Tree. This implementation should include the following methods: search, insert and remove, studied in class.

Solution:

```
from binarytrees import Node
from binarytrees import BinaryTree
```

```
class BinarySearchTree(BinaryTree):

    def insert(self,x):
        """inserts a new node, with element x, into the tree"""
        if self.root==None:
            self.root=Node(x)
        else:
            self.insertNode(self.root,x)

    def insertNode(self,node,x):
        """Inserts a new node (with the element x) inside of the subtree
node"""
        if node.elem==x:
            # Duplicate elements are not allowed
            print(x,'already exists!!!')
            return

        if x<node.elem:
            if node.leftChild!=None:
                self.insertNode(node.leftChild,x)
            else:
                newNode=Node(x)
                node.leftChild=newNode
                newNode.parent=node
        else: #x>node.elem
            if node.rightChild!=None:
                self.insertNode(node.rightChild,x)
            else:
                newNode=Node(x)
                node.rightChild=newNode
                newNode.parent=node

    def search(self,x):
        return self.searchNode(self.root,x)

    def searchNode(self,node,x):
        """Auxiliary method to search a node with value x"""
        if node is None:
            return False

        if node.elem==x:
            return True

        if x<node.elem:
            return self.searchNode(node.leftChild,x)

        if x>node.elem:
            return self.searchNode(node.rightChild,x)
```

```
    def find(self,x):
        """Returns the ndoe whose element is x. If it is not found, it
returns None"""
        return self.findNode(self.root,x)

    def findNode(self,node,x):
        if node is None:
            return None

        if node.elem==x:
            return node

        if x<node.elem:
            return self.findNode(node.leftChild,x)

        if x>node.elem:
            return self.findNode(node.rightChild,x)

    def remove(self,x):
        """Searches and removes the node whose element is x"""
        node=self.find(x)
        if node is None:
            print(x,' does not exist!!!')
            return
        print('removing ', x)
        self.removeNode(node)

    def removeNode(self,node):
        """Auxiliary method to remove the node which takes as parameter"""
        #First case: no children
        if node.leftChild is None and node.rightChild is None:
            parent_node=node.parent
            if parent_node is not None:
                if parent_node.leftChild==node:
                    parent_node.leftChild=None
                else:
                    parent_node.rightChild=None
                node.parent=None
            else:
                self.root=None
            return

        #Second case: only one child
        if node.leftChild is not None and node.rightChild is None:

            parent_node=node.parent
            if parent_node is not None:
                if parent_node.leftChild==node:
                    parent_node.leftChild=node.leftChild
                else:
                    parent_node.rightChild=node.leftChild
```

```
                node.leftChild.parent=parent_node
            else:
                self.root=node.leftChild
            return

        #Second case: only one child
        if node.leftChild is None and node.rightChild is not None:

            parent_node=node.parent
            if parent_node is not None:
                if parent_node.leftChild==node:
                    parent_node.leftChild=node.rightChild
                else:
                    parent_node.rightChild=node.rightChild
                node.rightChild.parent=parent_node
            else:
                self.root=node.rightChild
            return

        #Third case: two children
        successor=node.rightChild
        while successor.leftChild is not None:
            successor=successor.leftChild

        #we replace the node's elem by the successor's elem
        node.elem=successor.elem
        #we remove the succesor from the tree
        self.removeNode(successor)

#Now, we test the BinarySearchTree class:

import random

def test():
    bst=BinarySearchTree()

    for i in range(10):
        n=random.randint(0,25)
        print(n,end=' ')
        bst.insert(n)


    print()
    bst.draw()

    print('traversals')
    bst.inorder()
    bst.preorder()
    bst.postorder()
    bst.levelorder()
    print()
```

```
    print('searching...')
    for i in range(10):
        n=random.randint(0,25)
        print(n,bst.search(n))

    print('removing')
    bst.remove(6)

    bst.remove(2)
    bst.draw()

    bst.remove(12)
    bst.draw()

    bst.remove(17)
    bst.draw()

    bst.remove(bst.root.elem)
    bst.draw()


    print('traversals')
    bst.inorder()
    bst.preorder()
    bst.postorder()
    bst.levelorder()
    print()

test()
```

Problem: Implement an iterative method that returns the smallest element in the tree.

Solution:

```
def smallest(self):
"""Iterative method that returns the smallest element in the tree"""
  if self.root is None:
    print('tree is empty')
    return None

  node=self.root
  while node.leftChild:
    node=node.leftChild

  return node.elem
```

Problem: Implement an iterative method that returns the maximum element in the tree.

Solution:

```
 def maximum(self):
        """Iterative method that returns the maximum element in the tree"""
        if self.root is None:
            print('tree is empty')
            return None

        node=self.root
        while node.rightChild:
            node=node.rightChild

        return node.elem
```

Problem: Implement a recursive method that sums all the elements in the tree and returns this result.

Solution:

```
def sumEtos(self):
  """Implement a recursive method that sums all the elements in the tree
and returns this result."""
   return self._sumEtos(self.root)


def _sumEtos(self,node):
  if node is None:
    return 0

  return node.elem + self._sumEtos(node.leftChild) +
self._sumEtos(node.rightChild)
```

Problem: Implement a recursive method that visits all the nodes and prints those whose grandparent's elements is multiple of 10.

Solution:

```
def print10(self):
  """Implement a recursive method that visits all the nodes
  and prints those whose grandparent's elem is multiple of 10."""

  print('nodes whose grandparent is multiple of 10:', end=' ')
  self.print10Nodes(self.root)

 def print10Nodes(self,node):
   if node is None:
     return

   if node.parent!=None and node.parent.parent!=None and
node.parent.parent.elem % 10==0:
     print(node.elem, end=' ')
```

```
  self.print10Nodes(node.leftChild)
  self.print10Nodes(node.rightChild)
```

Problem: Implement an iterative method that takes a binary search node and returns its predecessor node from its left subtree.

Solution:

```
def predecessor(self,node):
  """returns the predecessor node from its left subtree"""
  if node is None:
    return None

  if node.leftChild is None:
    print(node.elem, 'does not have any predeccessor in its left child')
    return None

  predecessor=node.leftChild
  while predecessor.rightChild:
    predecessor=predecessor.rightChild

  return predecessor
```

Problem: Implement an iterative method that takes a binary search node and returns its successor node from its right subtree.

Solution:

```
def successor(self,node):
  """returns the successor node from its left subtree"""
  if node is None:
    return None

  if node.rightChild is None:
    print(node.elem, 'does not have any successor in its right child')
    return None

  successor=node.rightChild
  while successor.leftChild:
    successor=successor.leftChild

  return successor
```

Problem: Implement a new version of the remove method, where the node's element to be removed is replaced by using its predecessor instead of using its successor in the tree.

Solution:

```
def removeByPred(self,x):
  """Searches and removes the node whose element is x"""
  node=self.find(x)
  if node is None:
    print(x,' does not exist!!!')
    return
  print('removing ', x)
  self.removeNodeByPred(node)

  def removeNodeByPred(self,node):
    """Auxiliary method to remove the node which takes as parameter"""
    #First case: no children
    if node.leftChild is None and node.rightChild is None:
      parent_node=node.parent
      if parent_node is not None:
        if parent_node.leftChild==node:
          parent_node.leftChild=None
        else:
          parent_node.rightChild=None
        node.parent=None
      else:
        self.root=None
            return

    #Second case: only one child
    if node.leftChild is not None and node.rightChild is None:
      parent_node=node.parent
      if parent_node is not None:
        if parent_node.leftChild==node:
          parent_node.leftChild=node.leftChild
        else:
          parent_node.rightChild=node.leftChild
        node.leftChild.parent=parent_node
      else:
        self.root=node.leftChild
      return

      #Second case: only one child
      if node.leftChild is None and node.rightChild is not None:
        parent_node=node.parent
        if parent_node is not None:
          if parent_node.leftChild==node:
            parent_node.leftChild=node.rightChild
          else:
            parent_node.rightChild=node.rightChild
          node.rightChild.parent=parent_node
```

```
        else:
          self.root=node.rightChild
        return

   #Third case: two children
   predecessor=self.predecessor(node)

   #we replace the node's elem by the successor's elem
   node.elem=predecessor.elem
   #we remove the succesor from the tree
   self.removeNodeByPred(predecessor)
```

Problem: Implement a method that takes a binary search node and returns its size balance factor. The size balance factor of a node is the difference between size of the left subtree and the size of the right subtree

Solution:

```
def fsize(self,node):
  """Returns the size balance factor for the input node"""
  if node is None:
    return 0
  return abs(self._size(node.leftChild)-self._size(node.rightChild))
```

Problem: Implement a method that takes a binary search node and returns its height balance factor. The height balance factor of a node is the difference between height of the left subtree and the height of the right subtree

Solution:

```
def fheight(self,node):
  """Returns the height balance factor for the input node"""
  if node is None:
    return 0

  return abs(self._height(node.leftChild)-self._height(node.rightChild))
```

Problem: Implement a method that checks if the tree is size balanced. A BST is size balanced if all its nodes have a size balance factor less or equal to 1.

Solution:

```
def isSizeBalanced(self):
  """Checks if the tree is size-balanced"""
  return self._isSizeBalanced(self.root)

def _isSizeBalanced(self,node):
  if node is None:
    return True

  if self.fsize(node)>1:
    return False
```

```
return self._isSizeBalanced(node.leftChild) and
self._isSizeBalanced(node.rightChild)
```

Problem: Implement a method that checks if the tree is height balanced (AVL). A BST is an AVL if all its nodes have a height balance factor less or equal to 1.

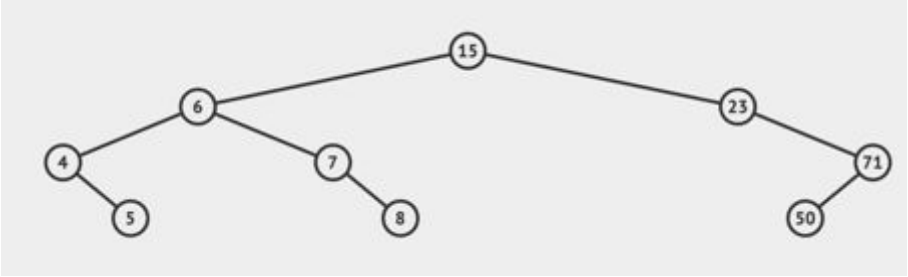
Solution:

```
def isAVL(self):
  """Checks if the tree is AVL (height balanced)"""
  return self._isAVL(self.root)

def _isAVL(self,node):
  if node is None:
    return True
  if self.fheight(node)>1:
    return False

   return self._isAVL(node.leftChild) and self._isAVL(node.rightChild)
```

Problem: Is it a size-balanced binary search tree (BST)?. If it is not, please balance it.

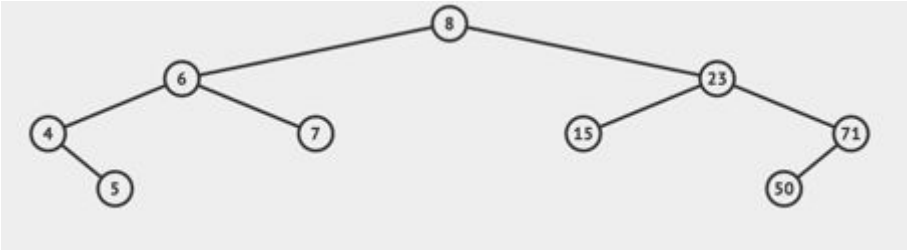


Solution:

Fsize(5)=0, Fsize(4)=1
Fsize(8)=0, Fsize(7)=1
Fsize(50)=0, Fsize(7)=1
Fsize(6)=0
Fsize(23)=2
Fsize(15)=2

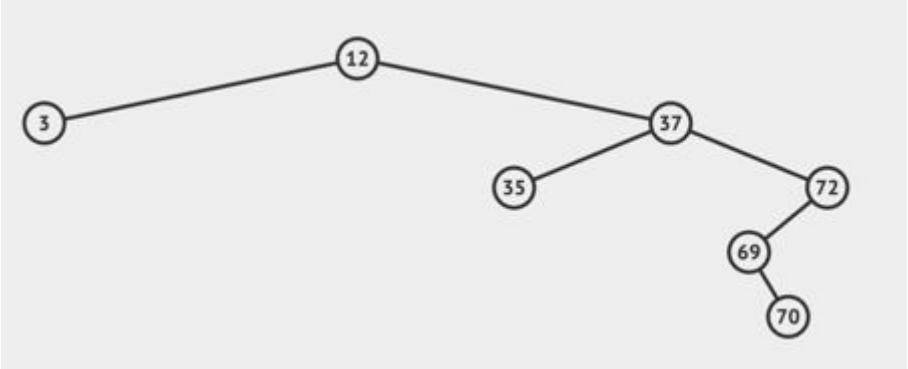
Unbalanced nodes: 15, 23.
Therefore, this tree is not size-balanced. The size-balancing algorithm must be always applied **descending down** from the root to the leaves. We start with 15:

1) Insert 15 into the subtree with less nodes (right subtree).
2) Get the maximum element from the left subtree (which 8) is and replace the root's element with this value.
3) Finally, you must **remove the maximum element from the left subtree**.



Now, the tree is size-balanced.

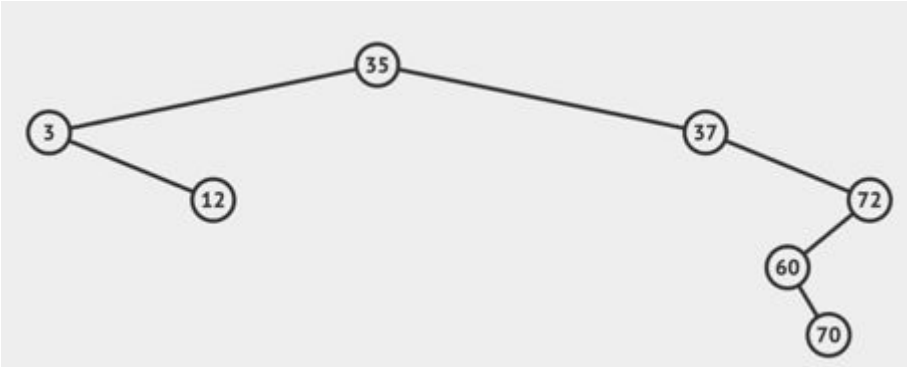
Problem: Is it a size-balanced binary search tree (BST)?. If it is not, please balance it.



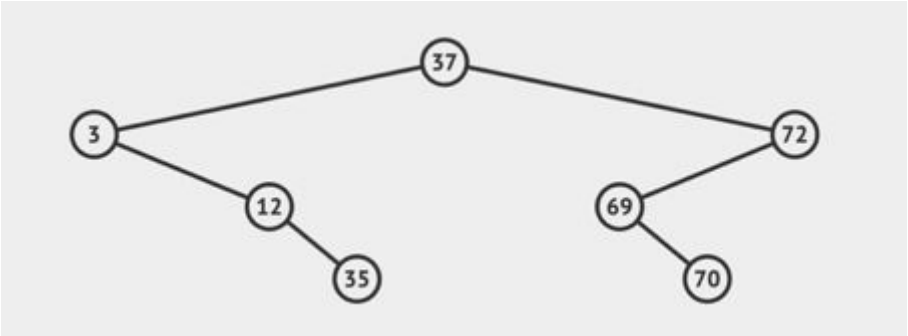
Solution:
Fsize(3)=0
Fsize(70)=0
Fsize(35)=0
Fsize(69)=1
Fsize(72)=2
Fsize(37)=2
Fsize(12)=4

The unbalanced nodes are: 72, 37, 12. We must start with 12.

1) Insert 12 in the left subtree as right child of 3.
2) Now, we should replace the root with the minimum element from the right subtree (35). If you choose any other element from the right subtree, the resulting tree won't a BST.
3) Finally, you must remove 35 from the right subtree. So, the resulting tree is.

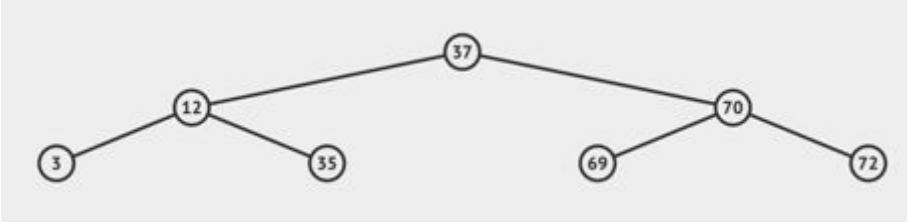


The unbalanced nodes are: 35, 37, 72. So, we must continue balancing the root 35.



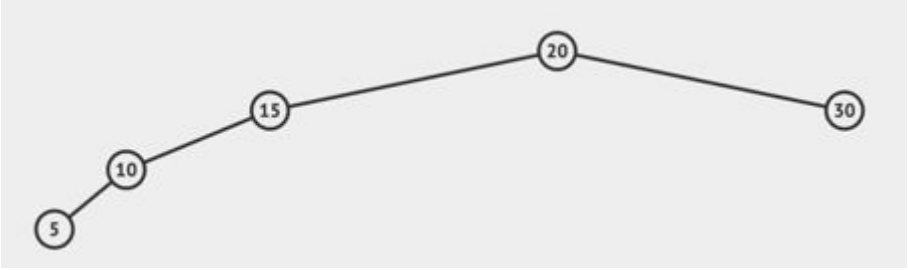
So, you can see that the root is balance, but there are still two unbalanced nodes, 3 and 72. As both node are in the same level, you can start with 3 or with 72.

So the resulting tree will be,



So, this tree is already perfect balanced.

Problem: Is it a height-balanced tree?. If it is not, please balance it.
Solution:

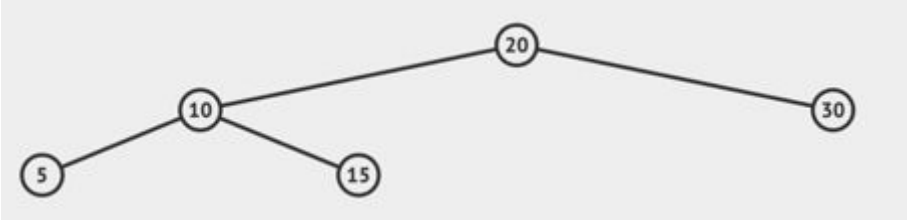


Fheight(5)=0, Fheight(10)=1, Fheight(30)=0
Fheight(15)=2 Fheight(20)=2
.

Unbalanced nodes: 15, 20. Therefore, this tree is not height-balanced.

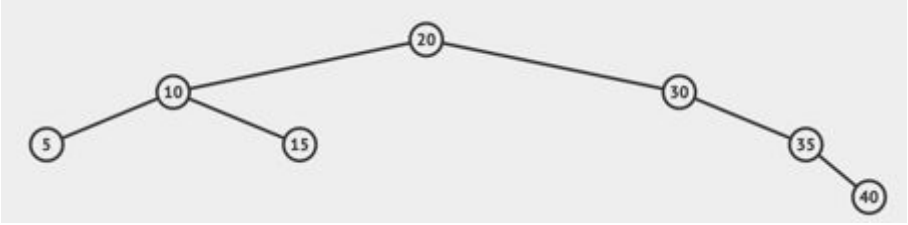
Rotation left-left.

You can see that in the example, the unbalanced node, 15, has a left-left unbalanced. So, now we can perform a LL rotation, in this case, 15 will go down as right child of 10, and 10 will become the new root of this subtree.



So, now, the tree is height-balanced tree.

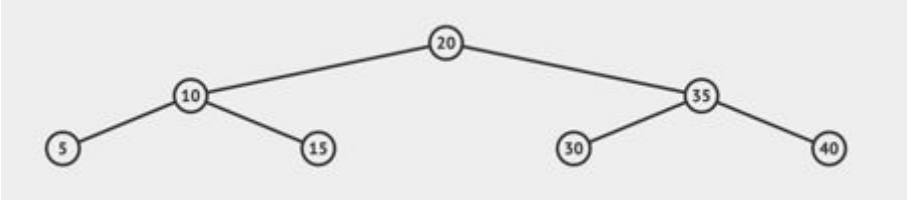
Problem: Please, transform the following tree to its height-balanced tree.



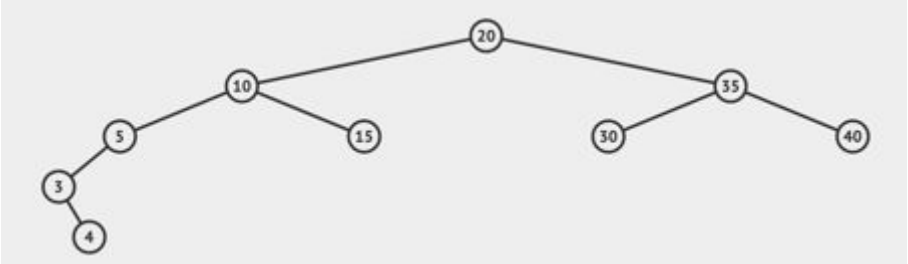
Solution:

Unbalanced nodes: 30. In this case, we can apply a Right Right rotation: 30 will go down as left child of 35, and 35 will become the new root of this subtree.

So, the result is:



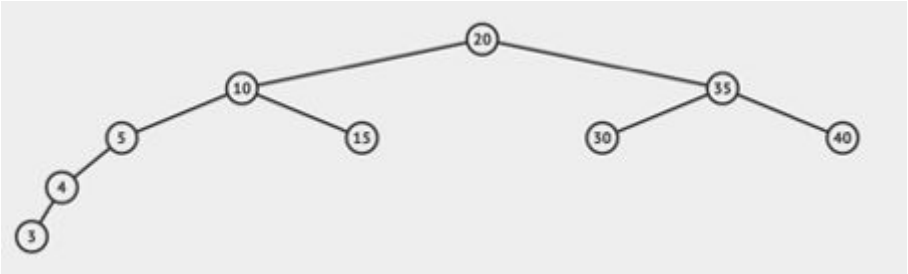
Problem: Please, transform the following tree to its height-balanced tree.



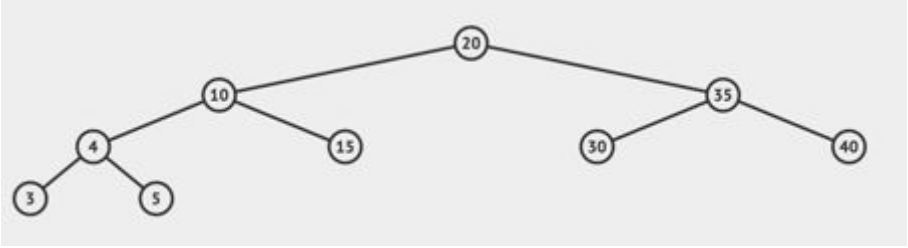
Solution:

The unbalanced nodes are: 5, 10, 20. We must apply the method in an ascending way, so we must start with 5.

Here, you can observe that the unbalance comes first from the left branch, and then from right one. We have a **left-right rotation**. This rotation must first be transformed to a simple rotation: 4 will go up as left child of 5 and 3 will become as the left child of 4.

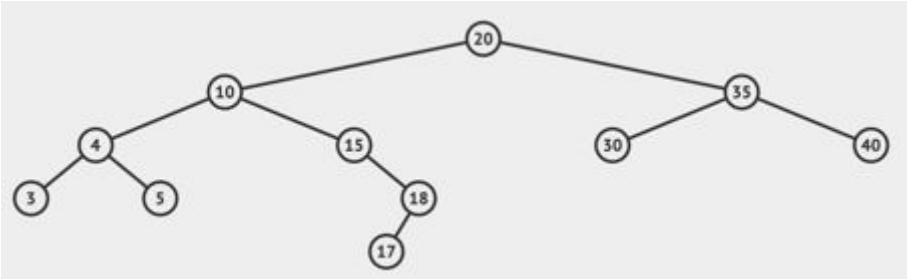


So, now you can apply, the left-left rotation.



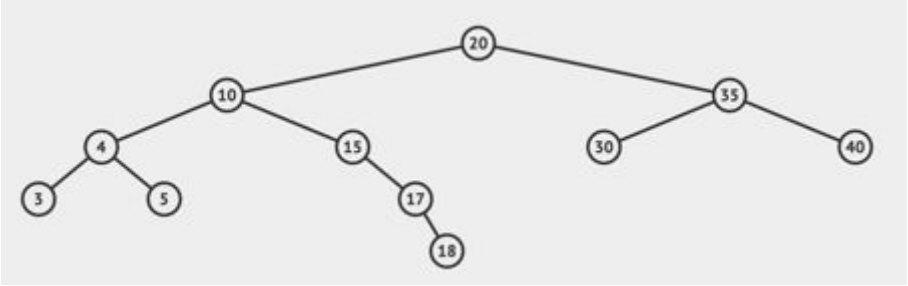
So, we can see that only balancing the node 5, we were able to balance the whole tree.

Problem: Please, transform the following tree to its height-balanced tree.

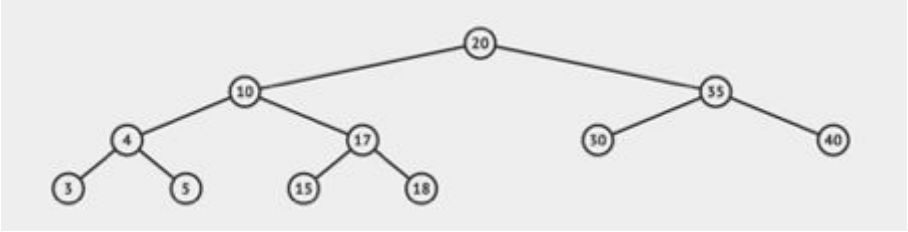


Solution:

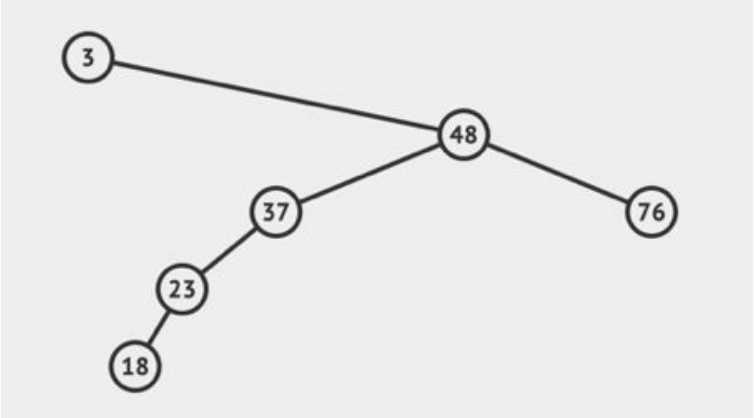
Now, the unbalanced nodes are 15 and 20. But we must start with 15. Now, the unbalance comes from the right branch, and after, from the left branch. So, we can apply the right-left rotation. As in the previous rotation, we first transform it to a simple rotation, so 17 will go up as right child of 15, and 18 will become the right child of 17



So, now, we can apply the right-right rotation, so 15 will go down as left child of 17, and 17 becomes the new root for this subtree.

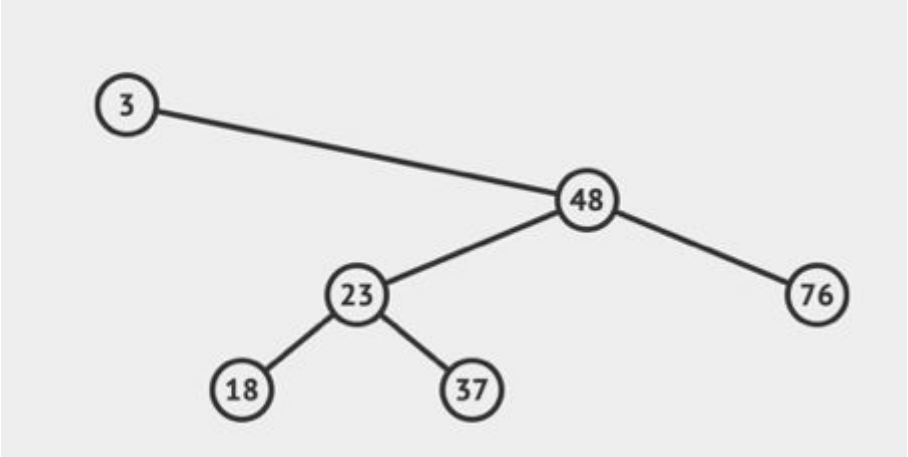


Problem: Please, transform the following tree to its height-balanced tree.



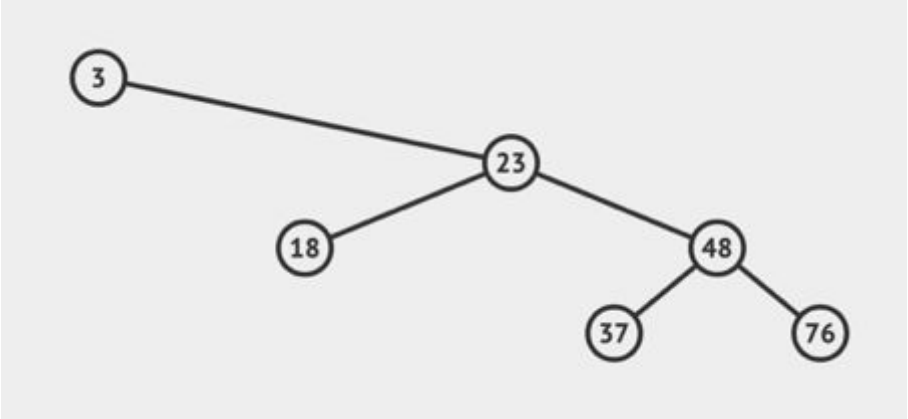
Solution:

Unbalanced nodes: 37, 48, 3. We start 37 by using a left-left rotation.

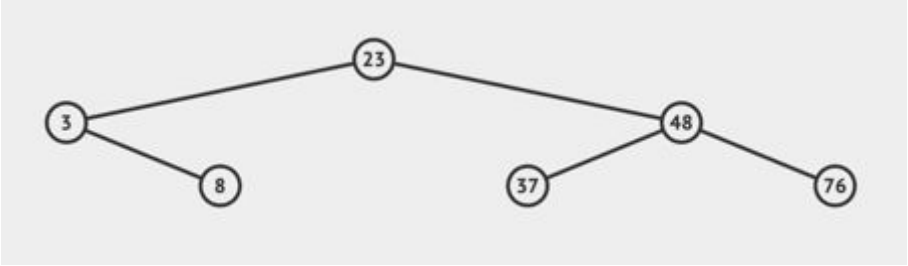


Now, the only unbalanced node 3, you can apply a right-right rotation or a right-left rotation.

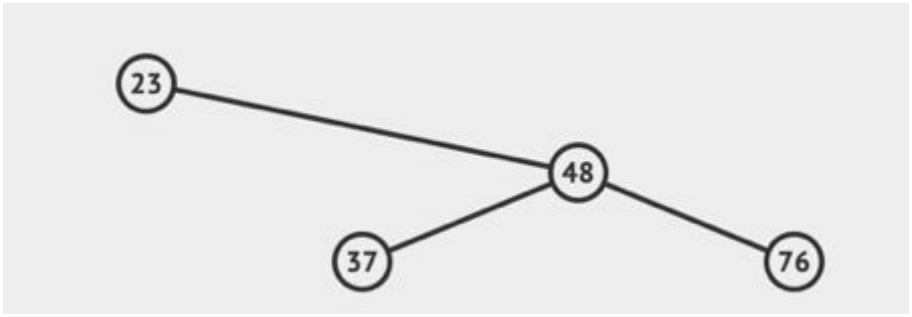
If you can apply two different rotations, it is better to apply the rotation that comes from the largest branch (most cases involve fewer transformations). In this case, it is the right-left rotation.
So, first, 23 will go up as right child of 3, 48 becomes the right child of 23, and then, 37 must become the left child of 48.



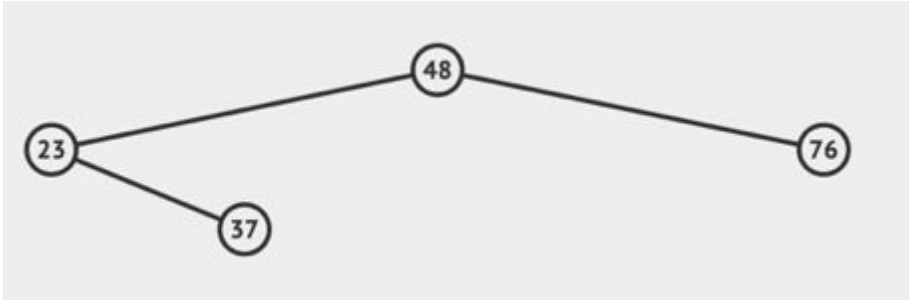
Now, we can apply the right-right rotation, where 3 goes down as left child of 23, 18 becomes the right child of 3, 23 becomes the new root.



Sometimes there are two possible rotation and their branches have the same length, as in this example:



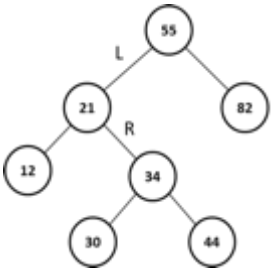
In this case, you can apply a RR rotation and a RL rotation. Choose always the simple rotation:



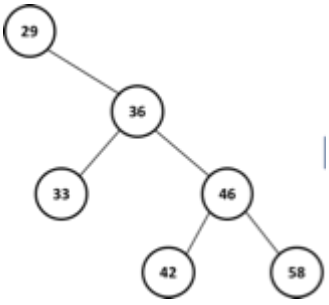
Problems:

Given the following binary search trees,

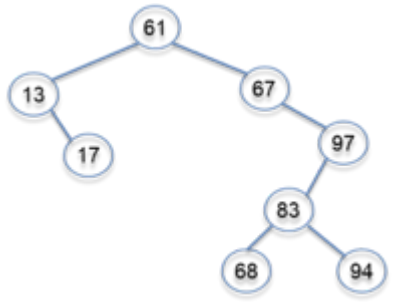
1)



2)



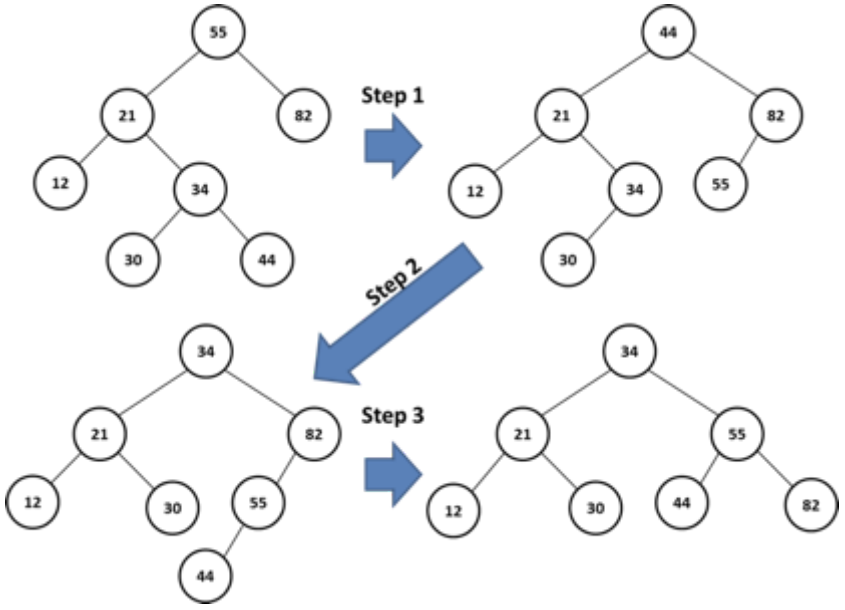
3)



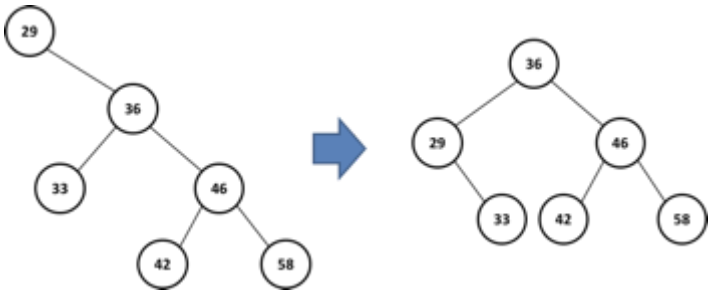
Transform them to
a) obtain their height-balanced trees.
b) obtain their size-balanced trees.

Solution:
1.a) Height-balanced version

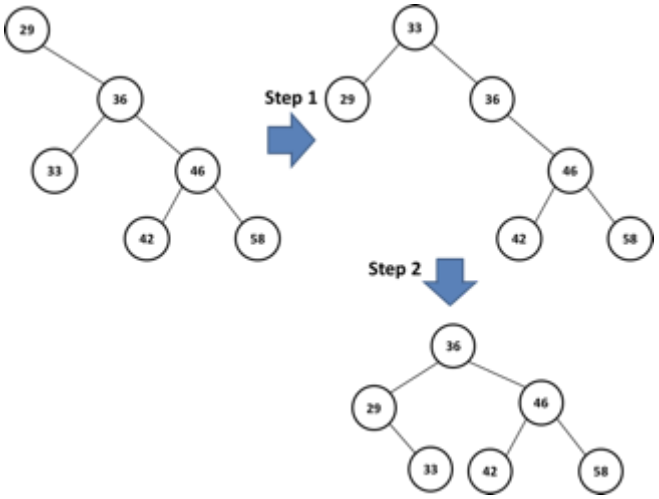
1.b) Size- balanced version



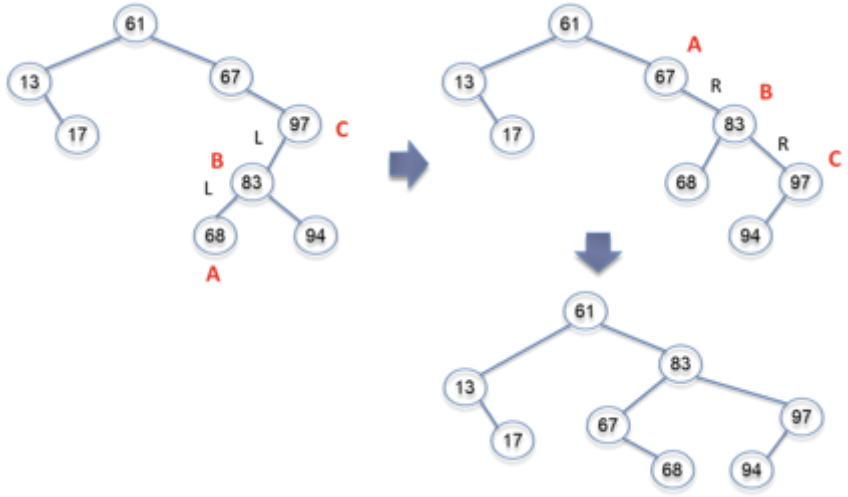
2.a) Height-balanced version:



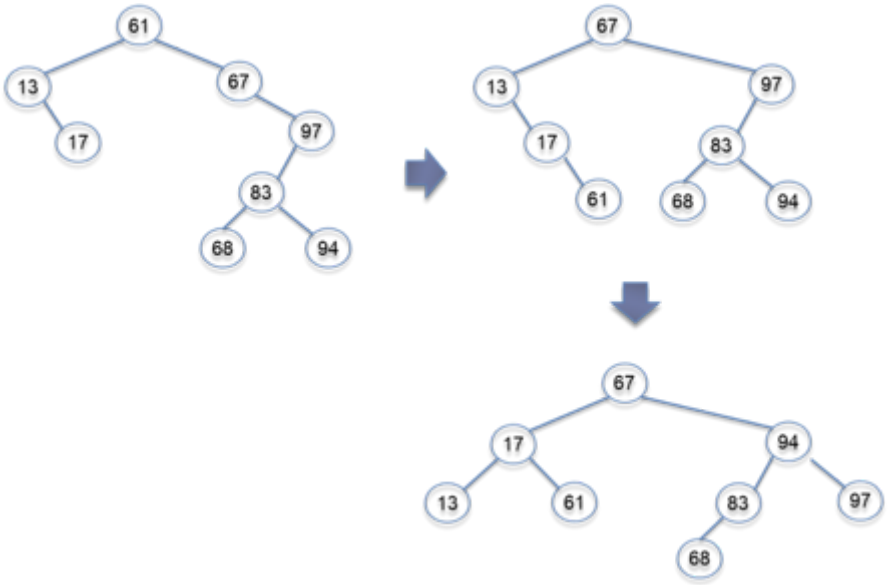
2.b) Size-balanced version:



3.a) Height-balanced version:



3.b) Size-balanced version:


61
13
67
17
97
83
68
94
67
13
97
17
83
61
68
94
67
17
94
13
61
83
97
68