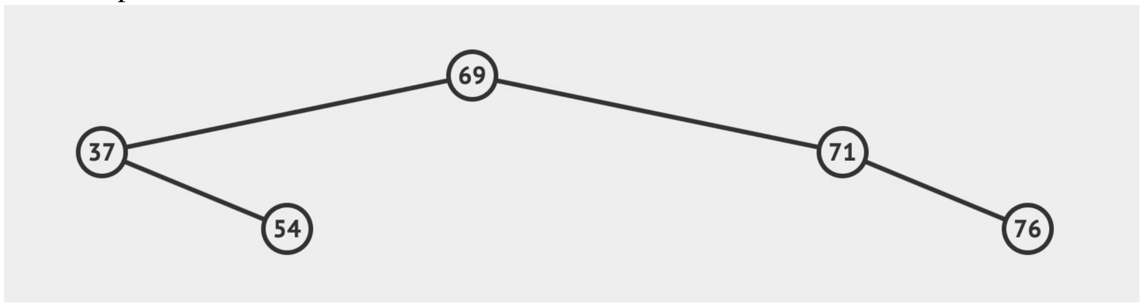
OCW: Data structures and algorithms. Author: I. Segura-Bedmar.

Problem 1 (1 points) – Suppose that BSTree is a class that implements a binary search tree whose elements are integers. Write a method that returns a doubly linked list containing the elements of the tree's nodes that are not leaves. The list should be sorted in **descending order**.

Some help:

- It is not allowed to use any sorting algorithm to sort the list.
- It is not allowed to use the Python List class. You must use the Doubly Linked List class studied during the course. You must also implement those methods of this class that you use in your solution.

For example:



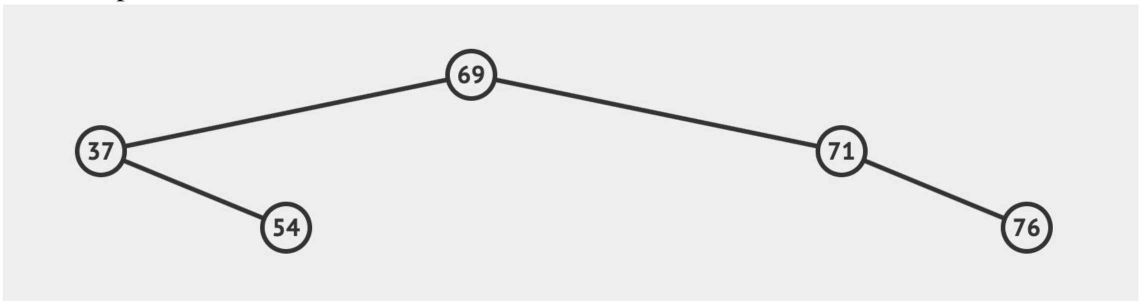
The output will be the doubly linked list containing the following elements: 71,69,37

Solution:

```
def getNonLeaves(self):
    if self.root is None:
        return None

    lst=DoublyLinkedList()
    self._getNonLeaves(self.root,lst)
    return lst

def _getNonLeaves(self,node,lst):
    if node:
        self._getNonLeaves(node.rightChild,lst)
        if node.leftChild!=None or node.rightChild!=None:
            lst.addLast(node.elem)
        self._getNonLeaves(node.leftChild,lst)
    def addLast(self,e):
        """Add a new element, e, at the end of the list"""
        #create the new node
        newNode=DoublyNode(e)

        if self.isEmpty():
          self.head=newNode
        else:
          newNode.prev=self.tail
          self.tail.next=newNode

        #update the reference of head to point the new node
        self.tail=newNode
        #increase the size of the list
        self.size=self.size+1
```

Problem 2 (1 points) – In the BSTree class, write a method that returns the kth smallest element of the binary tree. What is the time complexity of the method?.

For example:

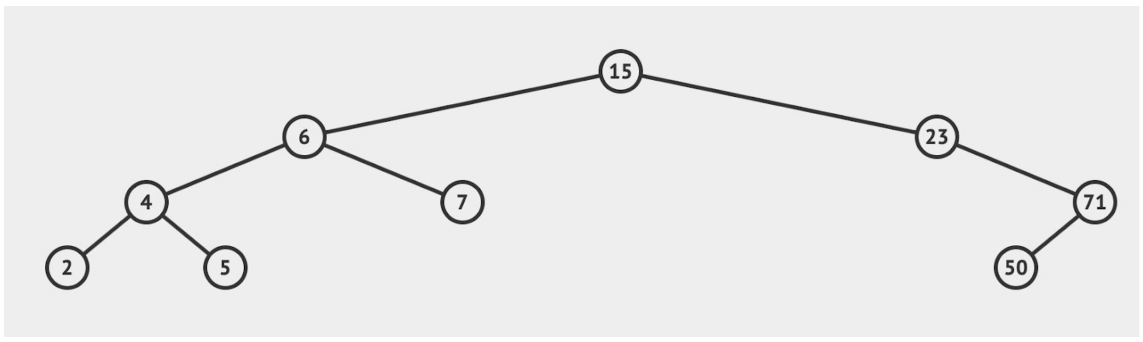


The 1st smallest element is 37
The 2nd smallest element is 54
The 3th smallest element is 69
The 4th smallest element is 71
The 5th smallest element is 76

Solution:

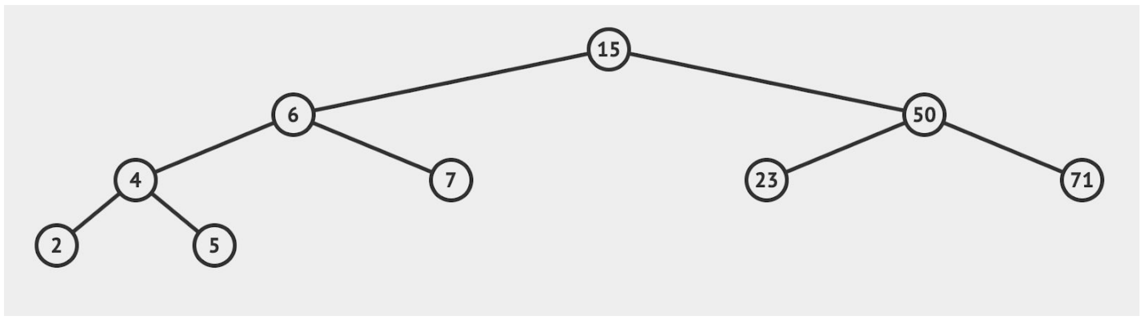
```
def getSortedList(self):
    if self.root is None:
        return None

    lst=DoublyLinkedList()
    self._getSortedList(self.root,lst)
    return lst

def _getSortedList(self,node,lst):
    if node:
        self._getSortedList(node.leftChild,lst)
        lst.addLast(node.elem)
        self._getSortedList(node.rightChild,lst)

def smallestK(self,k):
    if k<=0 or k>self.size():
        print(k, ' does not exist')
        return None
    lst=self.getSortedList()
    return lst.getAt(k-1)
```

Problem 3 (1 point):
A) (0.5) Draw its height-balanced tree.
B) (0.5) Draw its size-balanced tree.

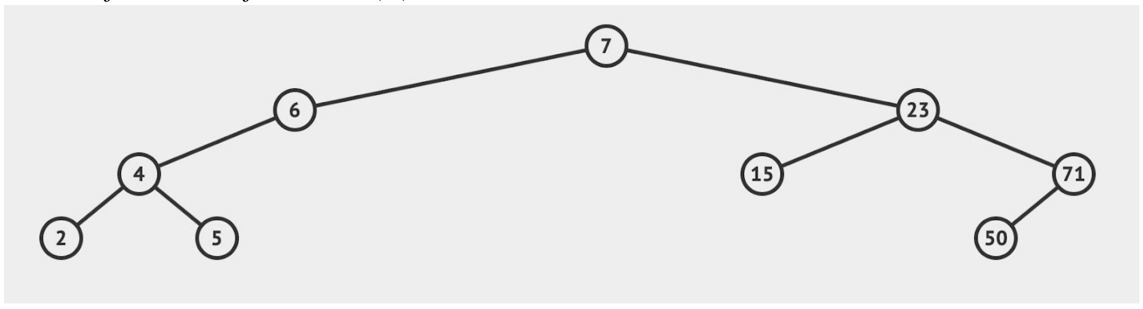


Solution:

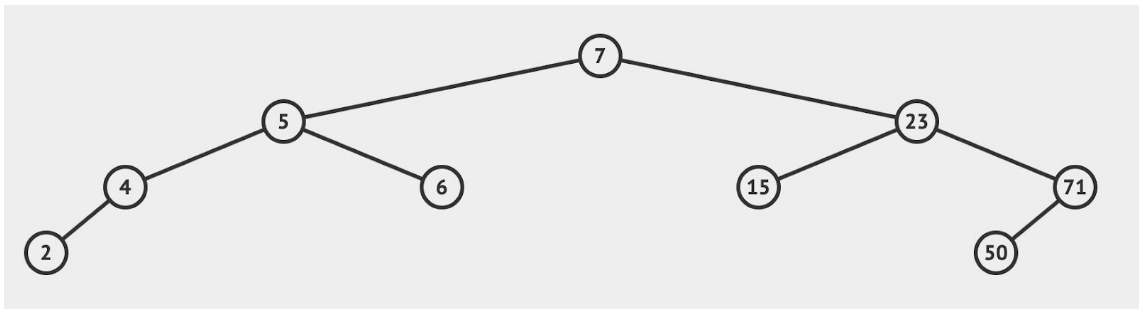
a) *The tree only has an unbalanced node, which is 23. You should apply a Right Left rotation on it:*



b) The tree has two unbalanced nodes in size: 15 and 23. We must start with the root, 15. Therefore, you should insert it on the right subtree and replace it with the maximum element from the left subtree (7).



As resulted of balancing the root, the node with key 23 was also balanced. However, the node with key 6 becomes an unbalanced node. We must move it to its right subtree.



Problem 4 (2 points). Given the following class that implements a social network for UC3M students:

```
class FriendsUC3M:
    def __init__(self):
        self.users = {}

    def addUser(self,email):
        if email in self.users:
            print(email, 'already exists!!!')
            return
        self.users[email] = []

    def getFriends(self,email):
        if email  not in self.users:
            print(email, 'does not exist!!!')
            return
        return self.users[email]


    def areFriends(self,email1,email2):
        if email1 not in self.users:
            return False
        if email2 not in self.users:
            return False
        if email2 in self.users[email1]:
            return True
        if email1 in self.users[email2]:
            return True
        return False


    def addFriends(self,email1,email2):
        if email1 not in self.users:
            self.addUser(email1)
        if email2 not in self.users:
            self.addUser(email2)
        if not self.areFriends(email1,email2):
            self.users[email1].append(email2)
            self.users[email2].append(email1)
```

Write a method, **friendsAtdistance**, which takes a user (an email) and a number, k, and returns a Python lists containing those users with a distance ‘k’ from the input user.

For example, given the social network:

pmf:['isa', 'lourdes']
isa:['pmf']
lourdes:['pmf', 'ana']
ana:['lourdes', 'ines', 'mateo']

ines:['ana']
mateo:['ana']

friendsAtdistance('pmf',1) returns ['isa', 'lourdes']
friendsAtdistance('pmf',2) returns ['ana']
friendsAtdistance('pmf',3) returns [ines, 'mateo']

It is allowed to use data structures such as Python dictionary or Python List to implement your final solution.

Solution:

```
    def friendsAtdistance(self,email,k):
      if k<1:
          print(k, 'should be greater than 1!!!')
          return None
      if email not in self.users:
          print(email, ' no exists')
          return None

      visited=[]

      return self._friendsAtdistance([email],1,k,visited)


  def _friendsAtdistance(self,friends,level,k,visited):

      if level>k: #base case
          print('over',level,k)
          return []

      result=[]

      for user in friends:
          visited.append(user)
          for x in self.getFriends(user):
              if x not in visited:
                  result.append(x)

      if level==k:
          return result

      if level<k:
          return self._friendsAtdistance(result,level+1,k,visited)
```

Problem 5 (1 point). Implement a method based on **divide and conquer** strategy that takes a sorted Python List of numbers, *A*, and a number, x. The method must return the index of x in the list. If x is not found, the method returns -1.

Solution:

```
def binarySearch(arr,x):
    if arr is None:
        return -1

    return _binarySearch(arr,0,len(arr)-1,x)

# Returns index of x in arr if present, else -1
def _binarySearch(arr, l, r, x):

    # Check base case
    if r >= l:
        mid = (l+r)//2
          # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only
        # be present in left subarray
        elif arr[mid] > x:
            return _binarySearch(arr, l, mid-1, x)
        # Else the element can only be present in right subarray
        else:
            return _binarySearch(arr, mid+1, r, x)
    else:
        # Element is not present in the array
        return -1
```