[Serialize and deserialize n-ary tree](https://stackoverflow.com/questions/20108347/interview-questions-serialize-and-deserialize-n-ary-tree)

I recently faced this question during the interview and the interviewer asking me to create two function. Function1 should take the n-ary tree and convert to byte array and function2 should take the byte[] and build the n-ary tree. If it was a binary tree, i would have made pre-order traversal with the special character for null and stored in an array and converted to byte[] but here n-ary tree (with many children). I don't know how to store this and rebuild the n-ary tree with an array. Any ideas or formula to store this n-ary tree into array?. I appreciate your help.

class Node:

def \_\_init\_\_(self, val):

self.val = val

self.children = []

class Codec:

def serialize(self, root):

if root is None:

return ""

res = [root.val, "#"]

q = collections.deque([root])

while q:

node = q.popleft()

for child in node.children:

res.append(child.val)

q.append(child)

res.append("#")

return ",".join(res)

def deserialize(self, s):

if len(s) == 0:

return

vals = s.split(",")

q = collections.deque()

root = Node(vals[0])

q.append(root)

i = 1

while q:

node = q.popleft()

i += 1

while vals[i] != "#":

child = Node(vals[i])

node.children.append(child)

q.append(child)

i += 1

return root

# Find the largest BST subtree in a given Binary Tree

|  |
| --- |
| class MinMax: |
|  |  |
|  | def \_\_init\_\_(self): |
|  | self.min = float("inf") |
|  | self.max = float("-inf") |
|  | self.isBST = True |
|  | self.size = 0 |
|  |  |
|  |  |
|  | class LargestBSTBinaryTree: |
|  |  |
|  | def largestBST(self, root): |
|  | m = self.largest(root) |
|  | return m.size |
|  |  |
|  | def largest(self, root): |
|  |  |
|  | if root is None: |
|  | return MinMax() |
|  |  |
|  | leftMinMax = self.largest(root.left) |
|  | rightMinMax = self.largest(root.right) |
|  |  |
|  | m = MinMax() |
|  |  |
|  | if ((leftMinMax.isBST == False or rightMinMax.isBST == False) |
|  | or (leftMinMax.max > root.data or rightMinMax.min <= root.data)): |
|  |  |
|  | m.isBST = False |
|  | m.size = max(leftMinMax.size, rightMinMax.size) |
|  | return m |
|  |  |
|  | m.isBST = True |
|  | m.size = leftMinMax.size + rightMinMax.size + 1 |
|  | m.min = leftMinMax.min if root.left is not None else root.data |
|  | m.max = rightMinMax.max if root.right is not None else root.data |
|  | return m |
|  |  |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | lbi = LargestBSTBinaryTree() |
|  | ctf = ConstructTreeFromInorderPreOrder() |
|  | inorder = [-7, -6, -5, -4, -3, -2, 1, 2, 3, 16, 6, 10, 11, 12, 14] |
|  | preorder = [3, -2, -3, -4, -5, -6, -7, 1, 2, 16, 10, 6, 12, 11, 14] |
|  | root = ctf.createTree(inorder, preorder) |
|  | largestBSTSize = lbi.largestBST(root) |
|  | print "Size of the largest BST in the Binary Tree is ", largestBSTSize |
|  | assert 8 == lbi.largestBST(root) |

# Reverse alternate levels of a perfect binary tree

Given a [Perfect Binary Tree](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees), reverse the alternate level nodes of the binary tree.

Given tree:

a

/ \

b c

/ \ / \

d e f g

/ \ / \ / \ / \

h i j k l m n o

Modified tree:

a

/ \

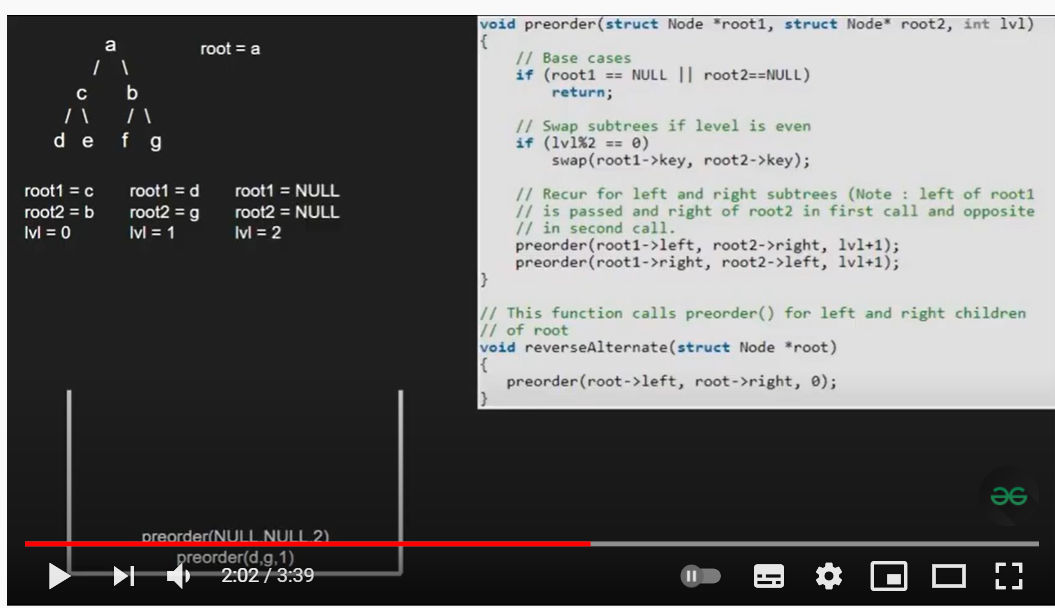
c b

/ \ / \

d e f g

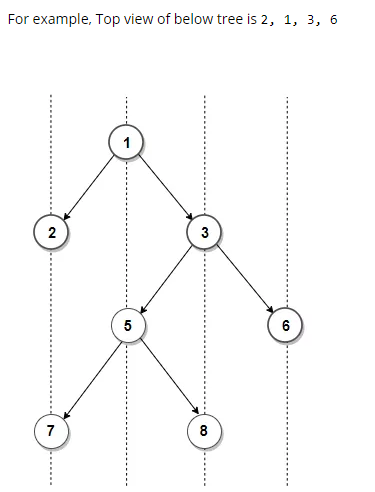
/ \ / \ / \ / \

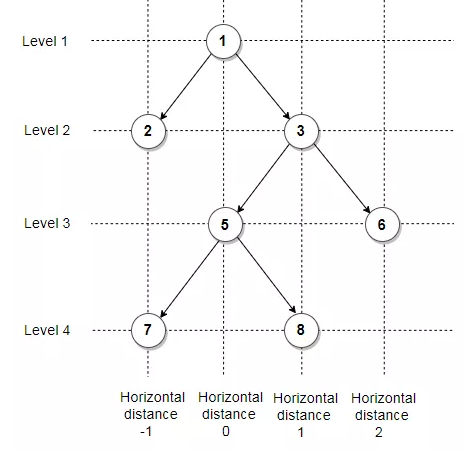
o n m l k j i h



|  |
| --- |
| # Given a Perfect Binary Tree, reverse the alternate level nodes of the binary tree |
|  |  |
|  |  |
|  | class Node: |
|  | def \_\_init\_\_(self, data): |
|  | self.data = data |
|  | self.left = None |
|  | self.right = None |
|  |  |
|  |  |
|  | def reverse\_alt\_levels(root): |
|  | pre\_order\_rev(root.left, root.right, 0) |
|  |  |
|  |  |
|  | def pre\_order\_rev(root\_left, root\_right, level): |
|  | # Base case |
|  | if (root\_left or root\_right) is None: |
|  | return |
|  |  |
|  | # swap the data of nodes if at an alternate level |
|  | if level % 2 == 0: |
|  | root\_left.data, root\_right.data = root\_right.data, root\_left.data |
|  |  |
|  | # go to the next level with left of left root and right of right root |
|  | # and vice versa |
|  | pre\_order\_rev(root\_left.left, root\_right.right, level+1) |
|  | pre\_order\_rev(root\_left.right, root\_right.left, level+1) |
|  |  |
|  |  |
|  | def in\_order(root): |
|  | if root is None: |
|  | return |
|  | in\_order(root.left) |
|  | print(root.data, end=' -> ') |
|  | in\_order(root.right) |
|  |  |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | root = Node('a') |
|  | root.left = Node('b') |
|  | root.right = Node('c') |
|  | root.left.left = Node('d') |
|  | root.left.right = Node('e') |
|  | root.right.left = Node('f') |
|  | root.right.right = Node('g') |
|  | root.left.left.left = Node('h') |
|  | root.left.left.right = Node('i') |
|  | root.left.right.left = Node('j') |
|  | root.left.right.right = Node('k') |
|  | root.right.left.left = Node('l') |
|  | root.right.left.right = Node('m') |
|  | root.right.right.left = Node('n') |
|  | root.right.right.right = Node('o') |
|  |  |
|  | print('Before Reversal:') |
|  | in\_order(root) |
|  | print() |
|  |  |
|  | # Call the reverse alternate levels function |
|  | reverse\_alt\_levels(root) |
|  |  |
|  | print('After Reversal:') |
|  | in\_order(root) |
|  | print() |

Print Top View of a binary tree





|  |
| --- |
| # Print Nodes in Top View of Binary Tree |
|  | from collections import deque |
|  |  |
|  |  |
|  | class Node: |
|  | def \_\_init\_\_(self, data): |
|  | self.data = data |
|  | self.left = None |
|  | self.right = None |
|  |  |
|  |  |
|  | def top\_view(root): |
|  | if root is None: |
|  | return |
|  |  |
|  | # make an empty queue for BFS |
|  | q = deque() |
|  |  |
|  | # empty set |
|  | sets = set({}) |
|  |  |
|  | # list to store top view keys |
|  | topview = [] |
|  |  |
|  | # append root in the queue with horizontal distance as 0 |
|  | q.append((root, 0)) |
|  |  |
|  | while q: |
|  | # get the element and horizontal distance |
|  | elem, hd = q.popleft() |
|  |  |
|  | # if the hd is seen first time it will be top view |
|  | if hd not in sets: |
|  | topview.append((elem.data, hd)) |
|  | sets.add(hd) |
|  |  |
|  | # add left and right child in the queue with hd - 1 and hd + 1 |
|  | if elem.left is not None: |
|  | q.append((elem.left, hd - 1)) |
|  | if elem.right is not None: |
|  | q.append((elem.right, hd + 1)) |
|  |  |
|  | # return the sorted topview on the basis of hd |
|  | return sorted(topview, key=lambda x: x[1]) |
|  |  |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | root = Node(1) |
|  | root.left = Node(2) |
|  | root.right = Node(3) |
|  | root.left.right = Node(4) |
|  | root.left.right.right = Node(5) |
|  | root.left.right.right.right = Node(6) |
|  |  |
|  | for i in top\_view(root): |
|  | print(i[0], end=' ') |

Bottom View:-

|  |
| --- |
| # Print Nodes in Bottom View of Binary Tree |
|  | from collections import deque |
|  |  |
|  |  |
|  | class Node: |
|  | def \_\_init\_\_(self, data): |
|  | self.data = data |
|  | self.left = None |
|  | self.right = None |
|  |  |
|  |  |
|  | def bottom\_view(root): |
|  | if root is None: |
|  | return |
|  |  |
|  | # make an empty queue for BFS |
|  | q = deque() |
|  |  |
|  | # dict to store bottom view keys |
|  | bottomview = {} |
|  |  |
|  | # append root in the queue with horizontal distance as 0 |
|  | q.append((root, 0)) |
|  |  |
|  | while q: |
|  | # get the element and horizontal distance |
|  | elem, hd = q.popleft() |
|  |  |
|  | # update the last seen hd element |
|  | bottomview[hd] = elem.data |
|  |  |
|  | # add left and right child in the queue with hd - 1 and hd + 1 |
|  | if elem.left is not None: |
|  | q.append((elem.left, hd - 1)) |
|  | if elem.right is not None: |
|  | q.append((elem.right, hd + 1)) |
|  |  |
|  | # return the bottomview |
|  | return bottomview |
|  |  |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | root = Node(20) |
|  | root.left = Node(8) |
|  | root.right = Node(22) |
|  | root.left.left = Node(5) |
|  | root.left.right = Node(3) |
|  | root.right.left = Node(4) |
|  | root.right.right = Node(25) |
|  | root.left.right.left = Node(10) |
|  | root.left.right.right = Node(14) |
|  |  |
|  | bottomview = bottom\_view(root) |
|  |  |
|  | for i in sorted(bottomview): |
|  | print(bottomview[i], end=' ') |

# Print Ancestors of a given node in Binary Tree

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.

1

/ \

2 3

/ \

4 5

/

7

# Python program to print ancestors of given node in

# binary tree

# A Binary Tree node

class Node:

    # Constructor to create a new node

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

# If target is present in tree, then prints the ancestors

# and returns true, otherwise returns false

def printAncestors(root, target):

    # Base case

    if root == None:

        return False

    if root.data == target:

        return True

    # If target is present in either left or right subtree

    # of this node, then print this node

    if (printAncestors(root.left, target) or

        printAncestors(root.right, target)):

        print root.data,

        return True

    # Else return False

    return False

# Driver program to test above function

root = Node(1)

root.left = Node(2)

root.right = Node(3)

root.left.left = Node(4)

root.left.right = Node(5)

root.left.left.left = Node(7)

printAncestors(root, 7)

# Lowest Common Ancestor in a Binary Search Tree.

Given values of two values n1 and n2 in a Binary Search Tree, find the **L**owest **C**ommon **A**ncestor (LCA). You may assume that both the values exist in the tree.



LCA of 10 and 14 is 12

LCA of 14 and 8 is 8

LCA of 10 and 22 is 20

# Function to find LCA of n1 and n2.

# The function assumes that both

#   n1 and n2 are present in BST

def lca\_iterative(root, n1, n2):

    while root:

        # If both n1 and n2 are smaller than root,

        # then LCA lies in left

        if root.data > n1 and root.data > n2:

            root = root.left

        # If both n1 and n2 are greater than root,

        # then LCA lies in right

        elif root.data < n1 and root.data < n2:

            root = root.right

        else:

            break

    return root

recursive approach:-

# A recursive python program to find LCA of two nodes

# n1 and n2

# A Binary tree node

class Node:

    # Constructor to create a new node

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

# Function to find LCA of n1 and n2. The function assumes

# that both n1 and n2 are present in BST

def lca(root, n1, n2):

    # Base Case

    if root is None:

        return None

    # If both n1 and n2 are smaller than root, then LCA

    # lies in left

    if(root.data > n1 and root.data > n2):

        return lca(root.left, n1, n2)

    # If both n1 and n2 are greater than root, then LCA

    # lies in right

    if(root.data < n1 and root.data < n2):

        return lca(root.right, n1, n2)

    return root

# Driver program to test above function

# Let us construct the BST shown in the figure

root = Node(20)

root.left = Node(8)

root.right = Node(22)

root.left.left = Node(4)

root.left.right = Node(12)

root.left.right.left = Node(10)

root.left.right.right = Node(14)

n1 = 10 ; n2 = 14

t = lca(root, n1, n2)

print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 14 ; n2 = 8

t = lca(root, n1, n2)

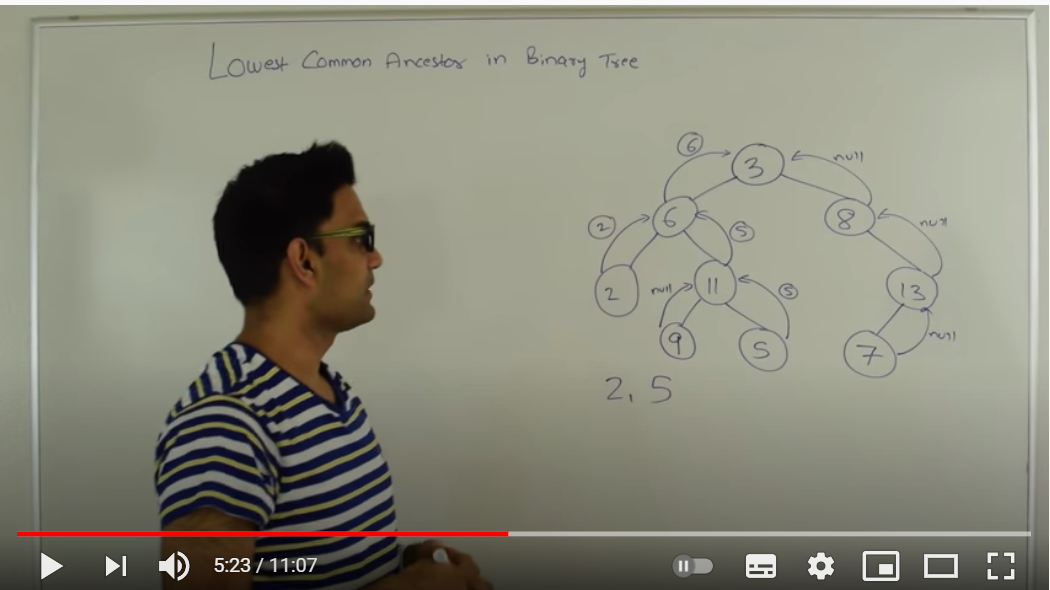
print "LCA of %d and %d is %d" %(n1, n2 , t.data)

n1 = 10 ; n2 = 22

t = lca(root, n1, n2)

print "LCA of %d and %d is %d" %(n1, n2, t.data)

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)



def findLCA(root, n1, n2):

    # Base Case

    if root is None:

        return None

    # If either n1 or n2 matches with root's key, report

    #  the presence by returning root (Note that if a key is

    #  ancestor of other, then the ancestor key becomes LCA

    if root.key == n1 or root.key == n2:

        return root

    # Look for keys in left and right subtrees

    left\_lca = findLCA(root.left, n1, n2)

    right\_lca = findLCA(root.right, n1, n2)

    # If both of the above calls return Non-NULL, then one key

    # is present in once subtree and other is present in other,

    # So this node is the LCA

    if left\_lca and right\_lca:

        return root

    # Otherwise check if left subtree or right subtree is LCA

    return left\_lca if left\_lca is not None else right\_lca

# Driver program to test above function

# Let us create a binary tree given in the above example

root = Node(1)

root.left = Node(2)

root.right = Node(3)

root.left.left = Node(4)

root.left.right = Node(5)

root.right.left = Node(6)

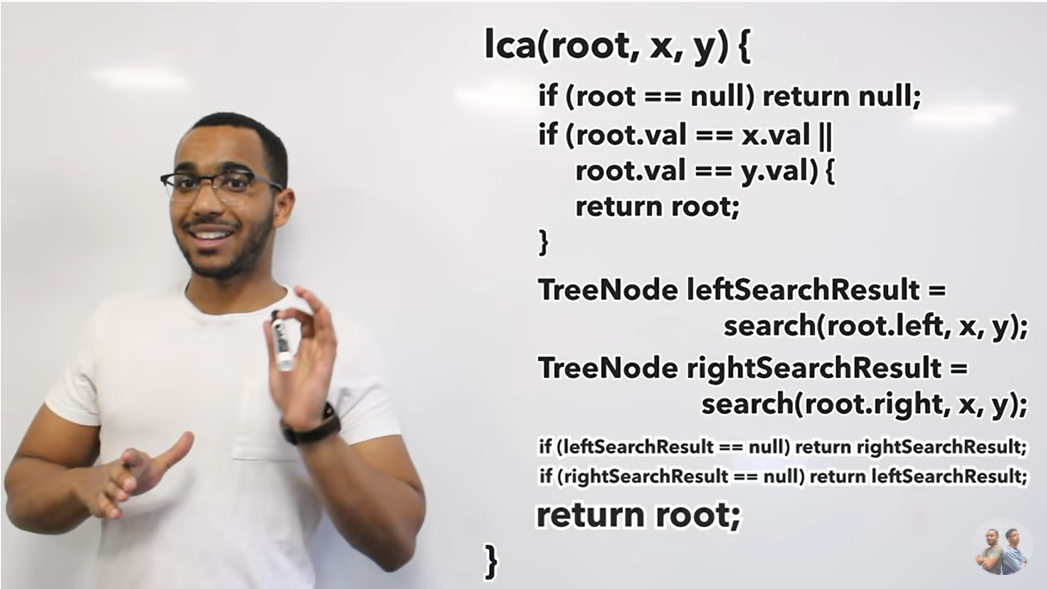
root.right.right = Node(7)

print "LCA(4,5) = ", findLCA(root, 4, 5).key

print "LCA(4,6) = ", findLCA(root, 4, 6).key

print "LCA(3,4) = ", findLCA(root, 3, 4).key

print "LCA(2,4) = ", findLCA(root, 2, 4).key



**Print Binary Tree in Spiral Form:-**

