Trees Live Class

Satyen Kansara

Summary of preclass videos

1. The first section discusses the problem of coming up with an efficient data structure to represent dictionaries and sets. Arrays and linked lists provide a brute-force implementation, which is then improved upon by the use of hash tables and finally **binary search trees**. Binary search tree operations like search, insert, delete, predecessor/successor, min, max are discussed along with iterative pseudocode that can be easily mapped to working code.

Summary of preclass videos

2. The second section discusses the use of **binary trees** (and N-ary trees) to represent hierarchical data. Two broad ways of traversing trees are discussed: BFS and DFS. Special types of DFS traversals like Preorder, Inorder and Postorder traversals are discussed with their simple recursive implementations. BFS is discussed using a queue-based iterative pseudocode. Finally, there is some discussion on how to reconstruct a binary tree from its traversals - preorder + inorder, or postorder + inorder. (No pseudocode given for it though)

Trees Patterns

Tree *traversal* patterns:

1. BFS
2. DFS

a) Top-down

b) Bottom-up

c) Boundary walk

d) Iterative boundary walk

Tree *construction* patterns

1. Top-down
2. Left-to-right (inorder)

```
#Handle an empty tree as a special edge case

#Initialize an empty result array

#Create an empty queue and push the root of the tree into it.
#While the queue is not empty:
    #Count how many nodes there are in the queue
    #Repeat that many times:
        #Pop the next node from the front of the queue
        #Append it to the result
        #If it has a left child, push it into the back of the queue
        #If it has a right child push it into the back of the queue
```

102. Binary Tree Level Order Traversal

Medium 1677 44 Favorite Share

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

For example:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

```
15        if root is None:
16            return []
17
18        result = []
19
20        q = collections.deque([root])
21        while len(q) != 0:
22            numnodes = len(q)
23            for _ in range(numnodes):
24                node = q.popleft()
25                if node.left is not None:
26                    q.append(node.left)
27                if node.right is not None:
28                    q.append(node.right)
29                result.append(node.val)
30
31        return result
```

```
15        if root is None:
16            return []
17
18        result = []
19
20        q = collections.deque([root])
21        while len(q) != 0:
22            numnodes = len(q)
23            temp = []
24            for _ in range(numnodes):
25                node = q.popleft()
26                if node.left is not None:
27                    q.append(node.left)
28                if node.right is not None:
29                    q.append(node.right)
30                temp.append(node.val)
31            result.append(temp)
32
33        return result
34
```

```
8  class Solution(object):
9      def levelOrder(self, root):
10         """
11         :type root: TreeNode
12         :rtype: List[List[int]]
13         """
14
15         if root is None:
16             return []
17
18         result = []
19         q = collections.deque([root])
20         while len(q) != 0: #While the queue is not empty
21             numnodes = len(q) #Find out how many nodes are in the current level
22             temp = []
23             for _ in range(numnodes): #Have to pop those many nodes. Note that new nodes in the next level will get
   pushed at the end
24                 node = q.popleft()
25                 temp.append(node.val) #temp array stores all the values in the current level
26                 if node.left is not None:
27                     q.append(node.left)
28                 if node.right is not None:
29                     q.append(node.right)
30             result.append(temp) #Take all the collected values and append them to the result array
31
32         return result
```

Level-order traversal = Traversal of the nodes in increasing order of distance from the root = Breadth-first search (BFS)

Collect the nodes in a level from left to right in a queue. This allows their children to also be collected left to right in another queue.

Time complexity = ?

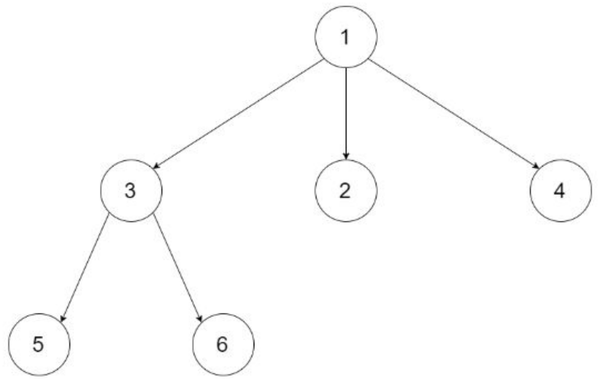
Space complexity = ?

429. N-ary Tree Level Order Traversal

Easy 297 34 Favorite Share

Given an n-ary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example, given a `3-ary` tree:



We should return its level order traversal:

```
[
     [1],
     [3,2,4],
     [5,6]
]
```

Note:

1. The depth of the tree is at most `1000`.
2. The total number of nodes is at most `5000`.

```
"""
# Definition for a Node.
class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children
"""
class Solution(object):
    def levelOrder(self, root):
        """
        :type root: Node
        :rtype: List[List[int]]
        """

        if root is None:
            return []

        result = []

        q = collections.deque([root])
        while len(q) != 0:
            numnodes = len(q)
            temp = []
            for _ in range(numnodes):
                node = q.popleft()
                for child in node.children:
                    q.append(child)
                temp.append(node.val)
            result.append(temp)

        return result
```

```
"""
# Definition for a Node.
class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children
"""
class Solution(object):
    def levelOrder(self, root):
        """
        :type root: Node
        :rtype: List[List[int]]
        """

        if root is None:
            return []

        result = []
        q = collections.deque([root])
        while len(q) != 0:
            numnodes = len(q)
            temp = []
            for _ in range(numnodes):
                node = q.popleft()
                temp.append(node.val)
                for child in node.children:
                    q.append(child)
            result.append(temp)

        return result
```

Time complexity = ?

Space complexity = ?

What other problems can be solved by BFS?

- Zigzag level order traversal
- Right side (or Left side) view of a binary tree
- Avg or Sum of every level
- Completeness of a binary tree **(We will see a problem if time permits)**

Trees DFS template

Note that according to this template, DFS is only called on a non-null node (like Graph DFS)

```
19 ▾        def dfs(node):
20 ▾            if node.left is not None:
21                  dfs(node.left)
22 ▾            if node.right is not None:
23                  dfs(node.right)
```

Trees DFS template

Handle a null tree as a special case outside of DFS code...

```
16 ▾        if root is None:
17              return
18
19 ▾        def dfs(node):
20 ▾            if node.left is not None:
21                  dfs(node.left)
22 ▾            if node.right is not None:
23                  dfs(node.right)
24
25          dfs(root)
```

```
16        if root is None:
17            return
18
19        def dfs(node):
20            #Base case: Leaf node
21            if node.left is None and node.right is None:
22                #Base case answer generated here
23
24            #Recursive case: Internal node
25            if node.left is not None:
26                dfs(node.left)
27            if node.right is not None:
28                dfs(node.right)
29
30        dfs(root)
```

Add a base case to the recursion (if the node is a leaf node).

Alternate template that allows DFS to be called on a null node (unlike Graph DFS)

```
32
33
34
35 ▾     def dfs(node):
36          #Base case: Null node
37 ▾        if node is None:
38              return
39
40          #Recursive case: Non-null node
41          dfs(node.left)
42          dfs(node.right)
43
44      dfs(root)
```

We will NOT use this for our template, as it doesn't distinguish the null child of a leaf from the null child of an internal node.

```
#Handle an empty tree as a special edge case

#General strategy executed by every node in the tree:
#function dfs(node):
    #Base case: If the node is a leaf, do whatever needs to be done for a leaf

    #Recursive case: The node is an internal node
    #If the node has a left child, dfs(node.left)
    #If the node has a right child, dfs(node.right)
```

Top-down DFS: The flow of information is from top to bottom, like a waterfall.

Since the information is processed by the node before passing it down to either child, it is “pre-order”.

```
15  #Handle an empty tree as a special edge case
16
17  #General strategy executed by every node in the tree:
18  #function TOP-DOWN dfs(node, information passed down by parent):
19
20      #Process the information coming into the node.
21
22      #Base case: If the node is a leaf, do whatever needs to be done for a leaf
23
24      #Recursive case: The node is an internal node
25      #If the node has a left child, dfs(node.left)
26      #If the node has a right child, dfs(node.right)
27
```

Bottom-up DFS: The flow of information is from bottom to top, like smoke rising up through vents.

Since the information is processed by the node after getting it from either child, it is “post-order”.

```
30  #General strategy executed by every node in the tree:
31  #function BOTTOM-UP dfs(node):
32
33      #Base case: If the node is a leaf, do whatever needs to be done for a leaf
34
35      #Recursive case: The node is an internal node
36      #If the node has a left child, R1 = dfs(node.left)
37      #If the node has a right child, R2 = dfs(node.right)
38
39      #Process the information (R1 and R2) coming into the node
40      #Return information up to the parent
```

112. Path Sum

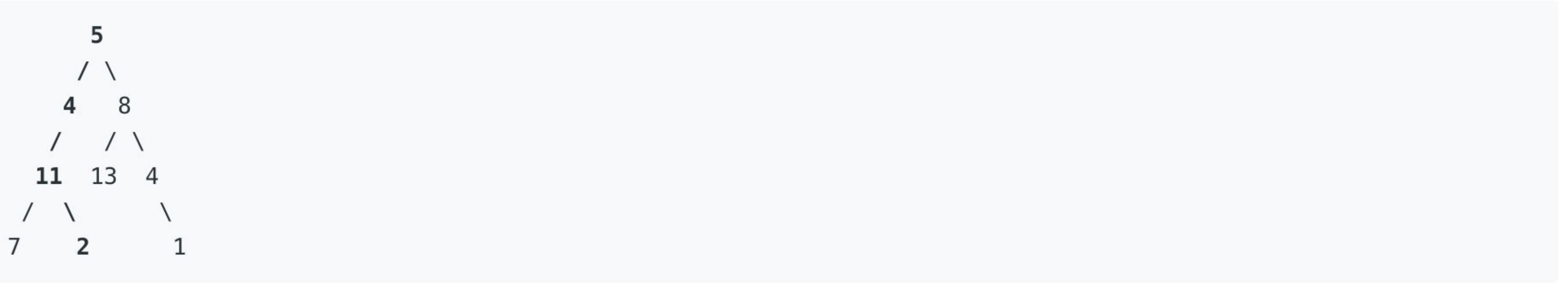
Easy 1198 380 Favorite Share

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

Note: A leaf is a node with no children.

Example:

Given the below binary tree and `sum = 22`,



return true, as there exist a root-to-leaf path `5->4->11->2` which sum is 22.

```
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """

        if root is None:
            return False

        flag = [False]

        #Top-down DFS => Additional arguments
        def dfs(node, target):

            #Base case: Leaf node
            if node.left is None and node.right is None:
                if target == node.val:
                    flag[0] = True

            #Recursive case: Internal node
            if node.left is not None:
                dfs(node.left, target - node.val)
            if node.right is not None:
                dfs(node.right, target - node.val)

        dfs(root, sum)
        return flag[0]
```

```python
class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """

        if root is None:
            return False

        flag = [False]

        def dfs(node, target):

            target -= node.val

            #Base case: Leaf node
            if node.left is None and node.right is None:
                if target == 0:
                    flag[0] = True

            #Recusive case: Internal node
            if node.left is not None:
                dfs(node.left, target)
            if node.right is not None:
                dfs(node.right, target)

        dfs(root, sum)
        return flag[0]
```

Common to leaf and non-leaf nodes

```
16 ▾        if root is None:
17              return False
18
19          flag = [False]
20
21 ▾        def dfs(node, target):
22              #Base case: Null node
23 ▾            if node is None:
24 ▾                if target == 0:
25                      flag[0] = True
26                  return
27
28              #Recursive case: Non-null node
29              dfs(node.left, target - node.val)
30              dfs(node.right, target - node.val)
31
32          dfs(root, sum)
33          return flag[0]
34
```

Isn’t the alternative template easier?

Wrong Answer Details >

Input	[1,2] 1
Output	true
Expected	false

```python
class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """

        if root is None:
            return False

        def dfs(node, sum):
            #Base case: Leaf node
            if node.left is None and node.right is None:
                return node.val == sum

            #Recursive case: Internal node
            bleft, bright = False, False
            if node.left is not None:
                bleft = dfs(node.left, sum - node.val)
            if node.right is not None:
                bright = dfs(node.right, sum - node.val)
            return bleft or bright

        return dfs(root, sum)

```

Bottom-up approach (come back to this later after doing bottom-up DFS)

Isn’t the alternate template easier?

Bottom-up approach (come back to this later after doing bottom-up DFS)

```
19 ▾        def dfs(node, target):
20              #Base case: Null node
21 ▾            if node is None:
22                  return target == 0
23
24              #Recursive case: Non-null node
25              return dfs(node.left, target - node.val) or dfs(node.right, target - node.val)
26
27
28          return dfs(root, sum)
```

Wrong Answer Details >

Input	[1,2] 1
Output	true
Expected	false

113. Path Sum II

Medium 1136 42 Favorite Share

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

Note: A leaf is a node with no children.

Example:

Given the below binary tree and `sum = 22`,

```
      5
     / \
    4   8
   /   / \
  11  13  4
 /  \    / \
7    2  5   1
```

Return:

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

```python
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: List[List[int]]
        """

        if root is None:
            return []

        result = []

        def dfs(node, target, slate):

            slate.append(node.val)

            #Base case: Leaf node
            if node.left is None and node.right is None:
                if node.val == target:
                    result.append(slate[:])

            #Recursive case: Internal node
            if node.left is not None:
                dfs(node.left, target-node.val, slate)
            if node.right is not None:
                dfs(node.right, target-node.val, slate)

            slate.pop()

        dfs(root, sum, [])
        return result
```

```python
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: List[List[int]]
        """

        if root is None:
            return []

        result = []

        def dfs(node, slate, target):

            slate.append(node.val)
            target -= node.val

            #Base case: Leaf node
            if node.left is None and node.right is None:
                if target == 0:
                    result.append(slate[:])

            #Recursive case: Internal node
            if node.left is not None:
                dfs(node.left, slate, target)
            if node.right is not None:
                dfs(node.right, slate, target)

            slate.pop()

        dfs(root, [], sum)
        return result

```

543. Diameter of Binary Tree

Easy 1802 111 Favorite Share

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.

Example:
Given a binary tree

```
          1
         / \
        2   3
       / \
      4   5
```

Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

Note: The length of path between two nodes is represented by the number of edges between them.

```
        if root is None:
            return 0

        #Global problem: Find the diameter of the whole tree
        #Local (per-node) problem: Find the longest inverted-V shaped path through the node.
        #Local -> Global: Global solution is the max of all the local solutions
        #To get the local solution, each node needs to know from its two subtrees what their heights are

        treediameter = [0]

        def dfs(node): #Returns the height of the subtree rooted at node

            mydiameter = 0
            myheight = 0
            #Base case: Leaf node
            if node.left is None and node.right is None:
                pass

            #Recursive case: Internal node
            if node.left is not None:
                leftsubtreeheight = dfs(node.left)
                myheight = 1 + leftsubtreeheight
                mydiameter = 1 + leftsubtreeheight
            if node.right is not None:
                rightsubtreeheight = dfs(node.right)
                myheight = max(myheight, 1+rightsubtreeheight)
                mydiameter += 1 + rightsubtreeheight

            if mydiameter > treediameter[0]:
                treediameter[0] = mydiameter

            return myheight

        dfs(root)
        return treediameter[0]
```

250. Count Univalue Subtrees

Medium 351 81 Favorite Share

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

Example :

```
Input:  root = [5,1,5,5,5,null,5]

              5
             / \
            1   5
           / \   \
          5   5   5

Output: 4
```

```python
        if root is None:
            return 0

        #To count the number of unival subtrees in the whole tree, we split the global problem into
        #a bunch of local problems. Each node will determine whether the subtree rooted at it is unival.
        #If yes, then we add 1 to the global count. But to determine whether a subtree is unival, the root
        #node will need to know whether its (upto) two children are unival or not. So that is the information
        #which needs to be returned by the child to its parent.
        globalcount = [0]

        def dfs(node):

            #Base case: Leaf node
            if node.left is None and node.right is None:
                globalcount[0] += 1
                return True

            #Recursive case: Internal node
            amiunival = True
            if node.left is not None:
                isleftunival = dfs(node.left)
                if not isleftunival or node.val != node.left.val:
                    amiunival = False
            if node.right is not None:
                isrightunival = dfs(node.right)
                if not isrightunival or node.val != node.right.val:
                    amiunival = False

            if amiunival:
                globalcount[0] += 1

            return amiunival

        dfs(root)
        return globalcount[0]
```

```
        if root is None:
            return 0

        #To count the number of unival subtrees in the whole tree, we split the global problem into
        #a bunch of local problems. Each node will determine whether the subtree rooted at it is unival.
        #If yes, then we add 1 to the global count. But to determine whether a subtree is unival, the root
        #node will need to know whether its (upto) two children are unival or not. So that is the information
        #which needs to be returned by the child to its parent.
        globalcount = [0]

        def dfs(node):

            amiunival = True
            #Base case: Leaf node
            if node.left is None and node.right is None:
                pass

            #Recursive case: Internal node
            if node.left is not None:
                isleftunival = dfs(node.left)
                if not isleftunival or node.val != node.left.val:
                    amiunival = False
            if node.right is not None:
                isrightunival = dfs(node.right)
                if not isrightunival or node.val != node.right.val:
                    amiunival = False

            if amiunival:
                globalcount[0] += 1

            return amiunival

        dfs(root)
        return globalcount[0]
```

Code slightly optimized w.r.t previous slide

144. Binary Tree Preorder Traversal

Medium 1017 46 Favorite Share

Given a binary tree, return the *preorder* traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,2,3]
```

Follow up: Recursive solution is trivial, could you do it iteratively?

```python
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """

        if root is None:
            return None

        result = []

        def dfs(node):

            #Base case: Leaf node
            if node.left is None and node.right is None:
                result.append(node.val)
                return

            #Recursive case: Internal node

            #Preorder processing
            result.append(node.val)

            if node.left is not None:
                dfs(node.left)
            if node.right is not None:
                dfs(node.right)

        dfs(root)
        return result
```

```python
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """

        if root is None:
            return []

        result = []

        def dfs(node):

            #Preorder processing
            result.append(node.val)

            if node.left is not None:
                dfs(node.left)

            if node.right is not None:
                dfs(node.right)

        dfs(root)
        return result
```

```
        if root is None:
            return []

        result = []

        stack = [(root,None)]

        while len(stack) != 0:
            (node, zone) = stack[-1]

            if zone is None:
                stack[-1] = (node, "arrival")
                result.append(node.val)
                if node.left is not None:
                    stack.append((node.left, None))

            elif zone == "arrival":
                stack[-1] = (node, "interim")
                if node.right is not None:
                    stack.append((node.right, None))

            elif zone == "interim":
                stack[-1] = (node, "departure")
                stack.pop()

        return result
```

94. Binary Tree Inorder Traversal

Medium 2130 95 Favorite Share

Given a binary tree, return the *inorder* traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,3,2]
```

Follow up: Recursive solution is trivial, could you do it iteratively?

```python
        if root is None:
            return []

        result = []

        stack = [(root,None)]

        while len(stack) != 0:
            (node, zone) = stack[-1]

            if zone is None:
                stack[-1] = (node, "arrival")
                if node.left is not None:
                    stack.append((node.left, None))

            elif zone == "arrival":
                stack[-1] = (node, "interim")
                result.append(node.val)
                if node.right is not None:
                    stack.append((node.right, None))

            elif zone == "interim":
                stack[-1] = (node, "departure")
                stack.pop()

        return result
```

145. Binary Tree Postorder Traversal

Hard 1187 65 Favorite Share

Given a binary tree, return the *postorder* traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [3,2,1]
```

Follow up: Recursive solution is trivial, could you do it iteratively?

```python
    def postorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """

        if root is None:
            return []

        result = []

        stack = [(root,None)]

        while len(stack) != 0:
            (node, zone) = stack[-1]

            if zone is None:
                stack[-1] = (node, "arrival")
                if node.left is not None:
                    stack.append((node.left, None))

            elif zone == "arrival":
                stack[-1] = (node, "interim")
                if node.right is not None:
                    stack.append((node.right, None))

            elif zone == "interim":
                stack[-1] = (node, "departure")
                result.append(node.val)
                stack.pop()

        return result
```

108. Convert Sorted Array to Binary Search Tree

Easy 1545 159 Favorite Share

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

Example:

```
Given the sorted array: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced
BST:

      0
     / \
   -3   9
   /   /
 -10  5
```

```
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """

        def helper(A, start, end):
            if start > end:
                return None
            elif start == end:
                return TreeNode(A[start])
            else:
                mid = (start+end)/2
                subtreeroot = TreeNode(A[mid])
                subtreeroot.left = helper(A, start, mid-1)
                subtreeroot.right = helper(A, mid+1, end)
                return subtreeroot

        return helper(nums,0,len(nums)-1)
```

Correctness: Why will such a tree be balanced?

Time complexity = ?

105. Construct Binary Tree from Preorder and Inorder Traversal

Medium 2253 62 Favorite Share

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

```
9 ▾     def buildTree(self, preorder, inorder):
10          """
11          :type preorder: List[int]
12          :type inorder: List[int]
13          :rtype: TreeNode
14          """
```

```
        ino_map = {} #Store the index of every number in the inorder traversal in a hashmap
        for i in range(len(inorder)):
            ino_map[inorder[i]] = i

        def helper(A1, start1, end1, A2, start2, end2):
            #return the subtree root of the binary tree constructed from the preorder subarray
            #A1[start1..end1] and inorder subarray A2[start2..end2]

            #Base case
            if start1 > end1:
                return None
            elif start1 == end1:
                return TreeNode(A1[start1])
            #At this point, we know that preorder list is more than 1 long

            #Recursive case
            #The first value is the root of the subtree
            rootval = A1[start1]
            #Find the index of this value in the inorder subarray
            #It is guaranteed to be present in the inorder subarray
            rootindex = ino_map[rootval]

            #Everything to its left is the left subtree. Everything to its right is the right subtree.
            numleft = rootindex-start2
            numright = end2-rootindex
            subtreeroot = TreeNode(A1[start1])
            subtreeroot.left = helper(A1, start1+1, start1+numleft, A2, start2, start2+numleft-1)
            subtreeroot.right = helper(A1, start1+numleft+1, start1+numleft+numright, A2, rootindex+1, rootindex+numright)
            return subtreeroot

        return helper(preorder, 0, len(preorder)-1, inorder, 0, len(inorder)-1)

```

1008. Construct Binary Search Tree from Preorder Traversal

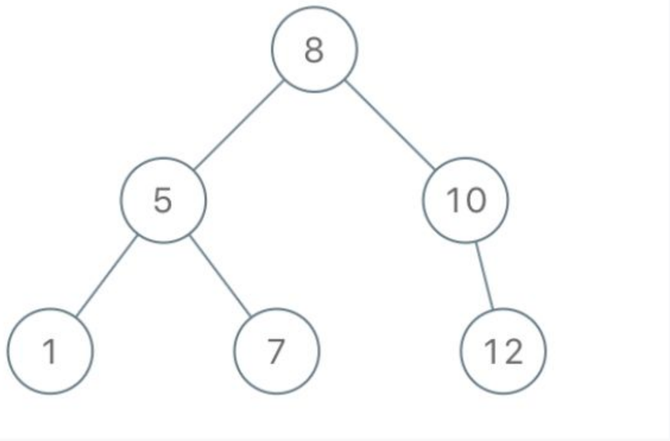
Medium 427 18 Favorite Share

Return the root node of a binary **search** tree that matches the given `preorder` traversal.

(Recall that a binary search tree is a binary tree where for every `node`, any descendant of `node.left` has a value `<` `node.val`, and any descendant of `node.right` has a value `>` `node.val`. Also recall that a preorder traversal displays the value of the `node` first, then traverses `node.left`, then traverses `node.right`.)

Example 1:

Input: `[8,5,1,7,10,12]`
Output: `[8,5,10,1,7,null,12]`



```
9 ▾     def bstFromPreorder(self, preorder):
10          """
11          :type preorder: List[int]
12          :rtype: TreeNode
13          """
14
15          inorder = sorted(preorder)
```

(same code as before here)

106. Construct Binary Tree from Inorder and Postorder Traversal

Medium 1107 24 Favorite Share

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
inorder = [9,3,15,20,7]
postorder = [9,15,7,20,3]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

```
9  ▾    def buildTree(self, inorder, postorder):
10          """
11          :type inorder: List[int]
12          :type postorder: List[int]
13          :rtype: TreeNode
14          """
```

```
        ino_map = {} #Store the index of every number in the inorder traversal in a hashmap
        for i in range(len(inorder)):
            ino_map[inorder[i]] = i

        def helper(A1, start1, end1, A2, start2, end2):
            #return the subtree root of the binary tree constructed from the postorder subarray
            #A1[start1..end1] and inorder subarray A2[start2..end2]

            #Base case
            if start1 > end1:
                return None
            elif start1 == end1:
                return TreeNode(A1[start1])
            #At this point, we know that postorder list is more than 1 long

            #Recursive case
            #The last value is the root of the subtree
            rootval = A1[end1]
            #Find the index of this value in the inorder subarray
            #It is guaranteed to be present in the inorder subarray
            rootindex = ino_map[rootval]

            #Everything to its left is the left subtree. Everything to its right is the right subtree.
            numleft = rootindex-start2
            numright = end2-rootindex
            subtreeroot = TreeNode(rootval)
            subtreeroot.left = helper(A1, start1, start1+numleft-1, A2, start2, start2+numleft-1)
            subtreeroot.right = helper(A1, start1+numleft, start1+numleft+numright-1, A2, rootindex+1, rootindex+numright)
            return subtreeroot

        return helper(postorder, 0, len(postorder)-1, inorder, 0, len(inorder)-1)

```

426. Convert Binary Search Tree to Sorted Doubly Linked List

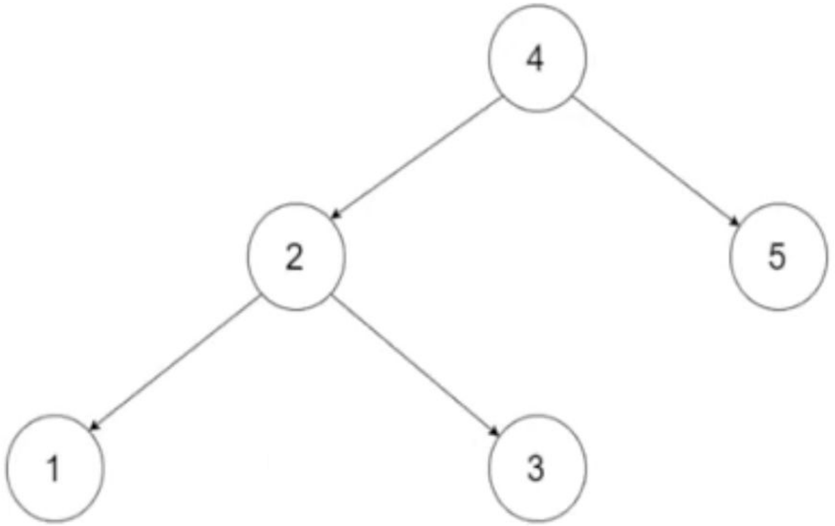
Medium 656 73 Add to List Share

Convert a **Binary Search Tree** to a sorted **Circular Doubly-Linked List** in place.

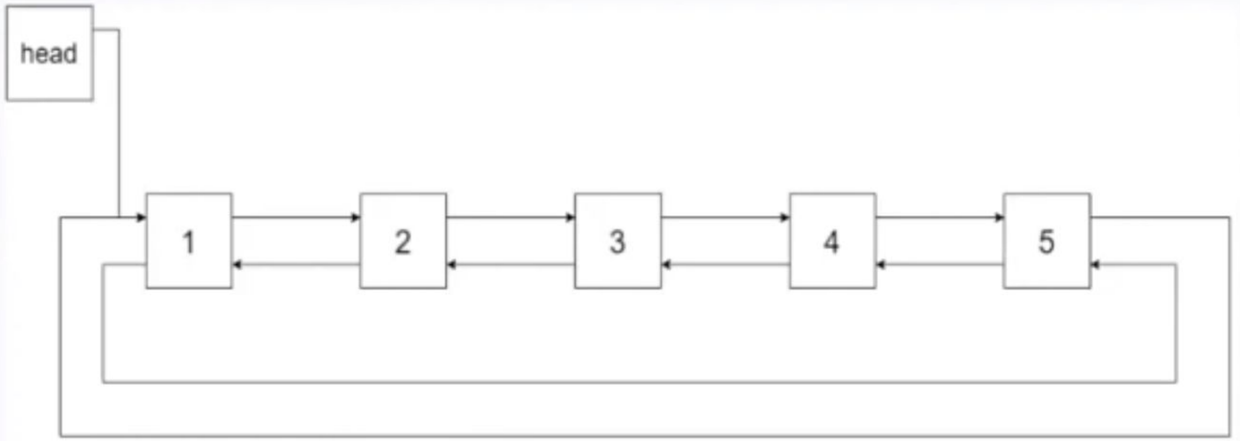
You can think of the left and right pointers as synonymous to the predecessor and successor pointers in a doubly-linked list. For a circular doubly linked list, the predecessor of the first element is the last element, and the successor of the last element is the first element.

We want to do the transformation **in place**. After the transformation, the left pointer of the tree node should point to its predecessor, and the right pointer should point to its successor. You should return the pointer to the smallest element of the linked list.

Example 1:



Input: root = [4,2,5,1,3]



Output: [1,2,3,4,5]

94. Binary Tree Inorder Traversal

Medium 2667 116 Add to List Share

Given a binary tree, return the *inorder* traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,3,2]
```

```
public class Solution {
    public List < Integer > inorderTraversal(TreeNode root) {
        List < Integer > res = new ArrayList < > ();
        Stack < TreeNode > stack = new Stack < > ();
        TreeNode curr = root;
        while (curr != null || !stack.isEmpty())
        {
                while (curr != null)
                {
                stack.push(curr);
                curr = curr.left;
                }
            curr = stack.pop();
            res.add(curr.val);
            curr = curr.right;
        }
        return res;
    }
}
```

144. Binary Tree Preorder Traversal

Medium 1270 54 Add to List Share

Given a binary tree, return the *preorder* traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,2,3]
```

```
void postOrderIterative(struct Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;
    struct Stack* stack = createStack(MAX_SIZE);
    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to
stack.
            if (root->right)
                push(stack, root->right);
            push(stack, root);

            // Set root as root's left child
            root = root->left;
        }

        // Pop an item from stack and set it as root
        root = pop(stack);

        // If the popped item has a right child and the right child is r
        // processed yet, then make sure right child is processed before
        if (root->right && peek(stack) == root->right)
        {
            pop(stack);  // remove right child from stack
            push(stack, root);  // push root back to stack
            root = root->right; // change root so that the right
                                // child is processed next
        }
        else  // Else print root's data and set root as NULL
        {
            printf("%d ", root->data);
            root = NULL;
        }
    } while (!isEmpty(stack));
}
```