



Trees Live Class

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)

```
1 #Handle an empty tree as a special edge case
2
3 #Initialize an empty result array
4
5 #Create an empty queue and push the root of the tree into it.
6 #While the queue is not empty:
7     #Count how many nodes there are in the queue
8     #Repeat that many times:
9         #Pop the next node from the front of the queue
10        #Append it to the result
11        #If it has a left child, push it into the back of the queue
12        #If it has a right child push it into the back of the queue
13
```

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] ,



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 = ?

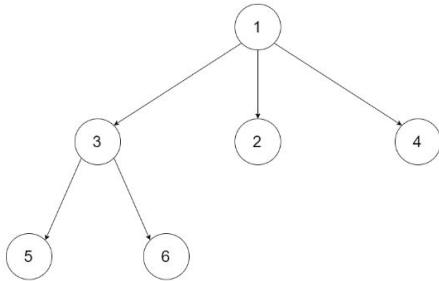
Space complexity = ?

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.

```
1 """
2 # Definition for a Node.
3 class Node(object):
4     def __init__(self, val, children):
5         self.val = val
6         self.children = children
7 """
8 class Solution(object):
9     def levelOrder(self, root):
10        """
11            :type root: Node
12            :rtype: List[List[int]]
13        """
14
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             for child in node.children:
27                 q.append(child)
28                 temp.append(node.val)
29             result.append(temp)
30
31     return result
```

```
1 """
2 # Definition for a Node.
3 class Node(object):
4     def __init__(self, val, children):
5         self.val = val
6         self.children = children
7 """
8 class Solution(object):
9     def levelOrder(self, root):
10         """
11         :type root: Node
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:
21         numnodes = len(q)
22         temp = []
23         for _ in range(numnodes):
24             node = q.popleft()
25             temp.append(node.val)
26             for child in node.children:
27                 q.append(child)
28             result.append(temp)
29
30     return result
```

Time complexity = ?

Space complexity = ?

107. Binary Tree Level Order Traversal II

Easy

960

179

Add to List

Share

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

For example:

Given binary tree [3,9,20,null,null,15,7],



return its bottom-up level order traversal as:

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

103. Binary Tree Zigzag Level Order Traversal

Medium

1447

83

Add to List

Share

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

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



return its zigzag level order traversal as:

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

```
9 def zigzagLevelOrder(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     ltor = True
21
22     while len(q) > 0:
23         numnodes = len(q)
24         temp = []
25         for _ in range(numnodes):
26             node = q.popleft()
27             temp.append(node.val)
28             if node.left is not None:
29                 q.append(node.left)
30             if node.right is not None:
31                 q.append(node.right)
32             if not ltor:
33                 temp.reverse()
34             ltor = not ltor
35             result.append(temp)
36
37     return result
```

199. Binary Tree Right Side View

Medium

1541

78

Add to List

Share

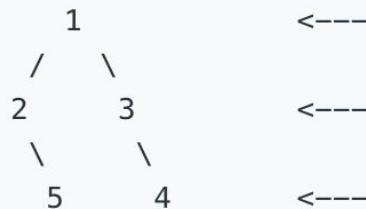
Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.

Example:

Input: [1,2,3,null,5,null,4]

Output: [1, 3, 4]

Explanation:



```
8 class Solution(object):
9     def rightSideView(self, root):
10         """
11             :type root: TreeNode
12             :rtype: List[int]
13         """
14         if root is None:
15             return []
16
17         result = []
18         q = collections.deque([root])
19         while len(q) > 0:
20             numnodes = len(q)
21             temp = []
22             for _ in range(numnodes):
23                 node = q.popleft()
24                 temp.append(node.val)
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(temp[-1])
30
return result
```

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:          Add a base case to the recursion (if
17           return                  the node is a leaf node).
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)
```

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

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

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

112. Path Sum

Easy

1198

380

Favorite

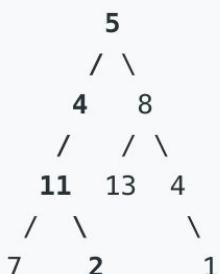
Share

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.

```
9 ▼    def hasPathSum(self, root, sum):  
10     """  
11         :type root: TreeNode  
12         :type sum: int  
13         :rtype: bool  
14     """  
15  
16 ▼        if root is None:  
17            return False  
18  
19        flag = [False]  
20  
21        #Top-down DFS => Additional arguments  
22 ▼    def dfs(node, target):  
23  
24        #Base case: Leaf node  
25 ▼        if node.left is None and node.right is None:  
26            if target == node.val:  
27                flag[0] = True  
28  
29        #Recursive case: Internal node  
30 ▼        if node.left is not None:  
31            dfs(node.left, target - node.val)  
32        if node.right is not None:  
33            dfs(node.right, target - node.val)  
34  
35        dfs(root, sum)  
36        return flag[0]
```

```
8 ▼ class Solution(object):
9 ▼     def hasPathSum(self, root, sum):
10
11         """
12             :type root: TreeNode
13             :type sum: int
14             :rtype: bool
15         """
16
17         if root is None:
18             return False
19
20         flag = [False]
21
22         def dfs(node, target):                Common to leaf and
23             target -= node.val               non-leaf nodes
24
25             #Base case: Leaf node
26             if node.left is None and node.right is None:
27                 if target == 0:
28                     flag[0] = True
29
30             #Recursive case: Internal node
31             if node.left is not None:
32                 dfs(node.left, target)
33             if node.right is not None:
34                 dfs(node.right, target)
35
36         dfs(root, sum)
37         return flag[0]
```

```
16    if root is None:  
17        return False  
18  
19    flag = [False]      Isn't the alterna-  
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
```

Wrong Answer

Details >

Input

```
[1,2]  
1
```

Output

```
true
```

Expected

```
false
```

```
8 v class Solution(object):
9 v     def hasPathSum(self, root, sum):
10
11         """
12             :type root: TreeNode
13             :type sum: int
14             :rtype: bool
15             """
16
17         if root is None:
18             return False
19
20         def dfs(node, sum):
21             #Base case: Leaf node
22             if node.left is None and node.right is None:
23                 return node.val == sum
24
25             #Recursive case: Internal node
26             bleft, bright = False, False
27             if node.left is not None:
28                 bleft = dfs(node.left, sum - node.val)
29             if node.right is not None:
30                 bright = dfs(node.right, sum - node.val)
31             return bleft or bright
32
33         return dfs(root, sum)
```

Isn't the alternate template easier?

```
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)
```

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

Wrong Answer

Details >

Input

```
[1,2]  
1
```

Output

```
true
```

Expected

```
false
```

113. Path Sum II

Medium

1136

42

Favorite

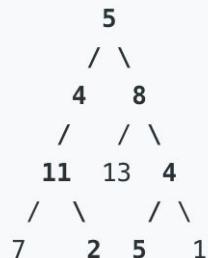
Share

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`,



Return:

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

```
9 def pathSum(self, root, sum):
10     """
11         :type root: TreeNode
12         :type sum: int
13         :rtype: List[List[int]]
14     """
15
16     if root is None:
17         return []
18
19     result = []
20
21     def dfs(node, target, slate):
22
23         slate.append(node.val)
24
25         #Base case: Leaf node
26         if node.left is None and node.right is None:
27             if node.val == target:
28                 result.append(slate[:])
29
30         #Recursive case: Internal node
31         if node.left is not None:
32             dfs(node.left, target-node.val, slate)
33         if node.right is not None:
34             dfs(node.right, target-node.val, slate)
35
36         slate.pop()
37
38     dfs(root, sum, [])
39     return result
```

```
9 def pathSum(self, root, sum):
10     """
11         :type root: TreeNode
12         :type sum: int
13         :rtype: List[List[int]]
14     """
15
16     if root is None:
17         return []
18
19     result = []
20
21     def dfs(node, slate, target):
22
23         slate.append(node.val)
24         target -= node.val
25
26         #Base case: Leaf node
27         if node.left is None and node.right is None:
28             if target == 0:
29                 result.append(slate[:])
30
31         #Recursive case: Internal node
32         if node.left is not None:
33             dfs(node.left, slate, target)
34         if node.right is not None:
35             dfs(node.right, slate, target)
36
37         slate.pop()
38
39     dfs(root, [], sum)
40     return result
41
```

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
```

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



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.

```

15 ▼
16     if root is None:
17         return 0
18
19     #Global problem: Find the diameter of the whole tree
20     #Local (per-node) problem: Find the longest inverted-V shaped path through the node.
21     #Local -> Global: Global solution is the max of all the local solutions
22     #To get the local solution, each node needs to know from its two subtrees what their heights are
23
24     treediameter = [0]
25
26     def dfs(node): #Returns the height of the subtree rooted at node
27
28         mydiameter = 0
29         myheight = 0
30         #Base case: Leaf node
31         if node.left is None and node.right is None:
32             pass
33
34         #Recursive case: Internal node
35         if node.left is not None:
36             leftsubtreeheight = dfs(node.left)
37             myheight = 1 + leftsubtreeheight
38             mydiameter = 1 + leftsubtreeheight
39             if node.right is not None:
40                 rightsubtreeheight = dfs(node.right)
41                 myheight = max(myheight, 1+rightsubtreeheight)
42                 mydiameter += 1 + rightsubtreeheight
43
44             if mydiameter > treediameter[0]:
45                 treediameter[0] = mydiameter
46
47         return myheight
48
49     dfs(root)
50     return treediameter[0]

```

250. Count Univalue Subtrees

Medium

351

81

Favorite

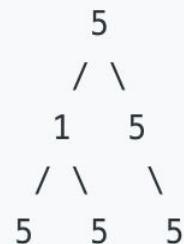
Share

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]



Output: 4

```
15 ▼
16     if root is None:
17         return 0
18
19 #To count the number of unival subtrees in the whole tree, we split the global problem into
20 #a bunch of local problems. Each node will determine whether the subtree rooted at it is unival.
21 #If yes, then we add 1 to the global count. But to determine whether a subtree is unival, the root
22 #node will need to know whether its (upto) two children are unival or not. So that is the information
23 #which needs to be returned by the child to its parent.
24 globalcount = [0]
25
26 def dfs(node):
27
28     #Base case: Leaf node
29     if node.left is None and node.right is None:
30         globalcount[0] += 1
31         return True
32
33     #Recursive case: Internal node
34     amiunival = True
35     if node.left is not None:
36         isleftunival = dfs(node.left)
37         if not isleftunival or node.val != node.left.val:
38             amiunival = False
39     if node.right is not None:
40         isrightunival = dfs(node.right)
41         if not isrightunival or node.val != node.right.val:
42             amiunival = False
43
44     if amiunival:
45         globalcount[0] += 1
46
47     return amiunival
48
49 dfs(root)
return globalcount[0]
```

```
15 ▾ if root is None:  
16     return 0  
17  
18 #To count the number of unival subtrees in the whole tree, we split the global problem into  
19 #a bunch of local problems. Each node will determine whether the subtree rooted at it is unival.  
20 #If yes, then we add 1 to the global count. But to determine whether a subtree is unival, the root  
21 #node will need to know whether its (upto) two children are unival or not. So that is the information  
22 #which needs to be returned by the child to its parent.  
23 globalcount = [0]  
24  
25 ▾ def dfs(node):  
26  
27     amiunival = True  
28     #Base case: Leaf node  
29 ▾ if node.left is None and node.right is None:  
30     pass  
31  
32     #Recursive case: Internal node  
33 ▾ if node.left is not None:  
34     isleftunival = dfs(node.left)  
35     if not isleftunival or node.val != node.left.val:  
36         amiunival = False  
37 ▾ if node.right is not None:  
38     isrightunival = dfs(node.right)  
39     if not isrightunival or node.val != node.right.val:  
40         amiunival = False  
41  
42 ▾ if amiunival:  
43     globalcount[0] += 1  
44  
45     return amiunival  
46  
47 dfs(root)  
48 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?

```
9 ▼     def preorderTraversal(self, root):  
10      """  
11          :type root: TreeNode  
12          :rtype: List[int]  
13          """  
14  
15 ▼         if root is None:  
16             return None  
17  
18         result = []  
19  
20 ▼     def dfs(node):  
21  
22         #Base case: Leaf node  
23 ▼         if node.left is None and node.right is None:  
24             result.append(node.val)  
25             return  
26  
27         #Recursive case: Internal node  
28  
29         #Preorder processing  
30         result.append(node.val)  
31  
32 ▼         if node.left is not None:  
33             dfs(node.left)  
34 ▼         if node.right is not None:  
35             dfs(node.right)  
36  
37         dfs(root)  
38         return result
```

```
9 def preorderTraversal(self, root):
10     """
11         :type root: TreeNode
12         :rtype: List[int]
13     """
14
15     if root is None:
16         return []
17
18     result = []
19
20     def dfs(node):
21
22         #Preorder processing
23         result.append(node.val)
24
25     if node.left is not None:
26         dfs(node.left)
27
28     if node.right is not None:
29         dfs(node.right)
30
31     dfs(root)
32     return result
```

```
15 ▼         if root is None:  
16             return []  
17  
18         result = []  
19  
20         stack = [(root, None)]  
21  
22 ▼         while len(stack) != 0:  
23             (node, zone) = stack[-1]  
24  
25 ▼             if zone is None:  
26                 stack[-1] = (node, "arrival")  
27                 result.append(node.val)  
28             if node.left is not None:  
29                 stack.append((node.left, None))  
30  
31 ▼             elif zone == "arrival":  
32                 stack[-1] = (node, "interim")  
33             if node.right is not None:  
34                 stack.append((node.right, None))  
35  
36 ▼             elif zone == "interim":  
37                 stack[-1] = (node, "departure")  
38                 stack.pop()  
39  
40         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?

```
15    if root is None:  
16        return []  
17  
18    result = []  
19  
20    stack = [(root, None)]  
21  
22    while len(stack) != 0:  
23        (node, zone) = stack[-1]  
24  
25        if zone is None:  
26            stack[-1] = (node, "arrival")  
27        if node.left is not None:  
28            stack.append((node.left, None))  
29  
30        elif zone == "arrival":  
31            stack[-1] = (node, "interim")  
32            result.append(node.val)  
33        if node.right is not None:  
34            stack.append((node.right, None))  
35  
36        elif zone == "interim":  
37            stack[-1] = (node, "departure")  
38            stack.pop()  
39  
40    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]



Output: [3,2,1]

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

```
9 ▼     def postorderTraversal(self, root):  
10    """  
11        :type root: TreeNode  
12        :rtype: List[int]  
13        """  
14  
15 ▼     if root is None:  
16         return []  
17  
18     result = []  
19  
20     stack = [(root, None)]  
21  
22 ▼     while len(stack) != 0:  
23         (node, zone) = stack[-1]  
24  
25 ▼         if zone is None:  
26             stack[-1] = (node, "arrival")  
27             if node.left is not None:  
28                 stack.append((node.left, None))  
29  
30 ▼             elif zone == "arrival":  
31                 stack[-1] = (node, "interim")  
32                 if node.right is not None:  
33                     stack.append((node.right, None))  
34  
35 ▼             elif zone == "interim":  
36                 stack[-1] = (node, "departure")  
37                 result.append(node.val)  
38                 stack.pop()  
39  
40             return result
```

108. Convert Sorted Array to Binary Search Tree

Easy

1545

159

Favorite

Share

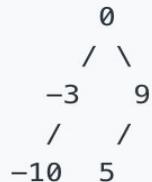
Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

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:



```
9 ▼     def sortedArrayToBST(self, nums):
10
11         """
12             :type nums: List[int]
13             :rtype: TreeNode
14         """
15
16     def helper(A, start, end):
17         if start > end:
18             return None
19         elif start == end:
20             return TreeNode(A[start])
21         else:
22             mid = (start+end)/2
23             subtreetroot = TreeNode(A[mid])
24             subtreetroot.left = helper(A, start, mid-1)
25             subtreetroot.right = helper(A, mid+1, end)
26             return subtreetroot
27
28     return helper(nums, 0, len(nums)-1)
```

Time and space 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.

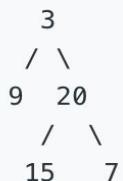
Note:

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:



9 ▾
10
11
12
13
14
--

```
def buildTree(self, preorder, inorder):
    """
    :type preorder: List[int]
    :type inorder: List[int]
    :rtype: TreeNode
    """
```

```
16     ino_map = {} #Store the index of every number in the inorder traversal in a hashmap
17     for i in range(len(inorder)):
18         ino_map[inorder[i]] = i
19
20     def helper(A1, start1, end1, A2, start2, end2):
21         #return the subtree root of the binary tree constructed from the preorder subarray
22         #A1[start1..end1] and inorder subarray A2[start2..end2]
23
24         #Base case
25         if start1 > end1:
26             return None
27         elif start1 == end1:
28             return TreeNode(A1[start1])
29         #At this point, we know that preorder list is more than 1 long
30
31         #Recursive case
32         #The first value is the root of the subtree
33         rootval = A1[start1]
34         #Find the index of this value in the inorder subarray
35         #It is guaranteed to be present in the inorder subarray
36         rootindex = ino_map[rootval]
37
38         #Everything to its left is the left subtree. Everything to its right is the right subtree.
39         numleft = rootindex-start2
40         numright = end2-rootindex
41         subtreetroot = TreeNode(A1[start1])
42         subtreetroot.left = helper(A1, start1+1, start1+numleft, A2, start2, start2+numleft-1)
43         subtreetroot.right = helper(A1, start1+numleft+1, start1+numleft+numright, A2, rootindex+1, rootindex+numright)
44         return subtreetroot
45
46     return helper(preorder, 0, len(preorder)-1, inorder, 0, len(inorder)-1)
47
```

1008. Construct Binary Search Tree from Preorder Traversal

Medium 427 18 Favorite Share

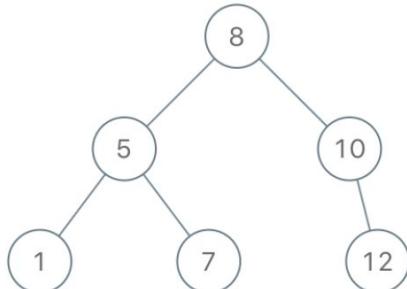
Return the root node of a binary **search** tree that matches the given `preorder` traversal.

(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)
16
```

(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:



9
10
11
12
13
14

```
def buildTree(self, inorder, postorder):
    """
    :type inorder: List[int]
    :type postorder: List[int]
    :rtype: TreeNode
    """
```

```
15     ino_map = {} #Store the index of every number in the inorder traversal in a hashmap
16     for i in range(len(inorder)):
17         ino_map[inorder[i]] = i
18
19     def helper(A1, start1, end1, A2, start2, end2):
20         #return the subtree root of the binary tree constructed from the postorder subarray
21         #A1[start1..end1] and inorder subarray A2[start2..end2]
22
23         #Base case
24         if start1 > end1:
25             return None
26         elif start1 == end1:
27             return TreeNode(A1[start1])
28         #At this point, we know that postorder list is more than 1 long
29
30         #Recursive case
31         #The last value is the root of the subtree
32         rootval = A1[end1]
33         #Find the index of this value in the inorder subarray
34         #It is guaranteed to be present in the inorder subarray
35         rootindex = ino_map[rootval]
36
37         #Everything to its left is the left subtree. Everything to its right is the right subtree.
38         numleft = rootindex-start2
39         numright = end2-rootindex
40         subtreetroot = TreeNode(rootval)
41         subtreetroot.left = helper(A1, start1, start1+numleft-1, A2, start2, start2+numleft-1)
42         subtreetroot.right = helper(A1, start1+numleft, start1+numleft+numright-1, A2, rootindex+1, rootindex+numright)
43         return subtreetroot
44
45     return helper(postorder, 0, len(postorder)-1, inorder, 0, len(inorder)-1)
46
```

426. Convert Binary Search Tree to Sorted Doubly Linked List

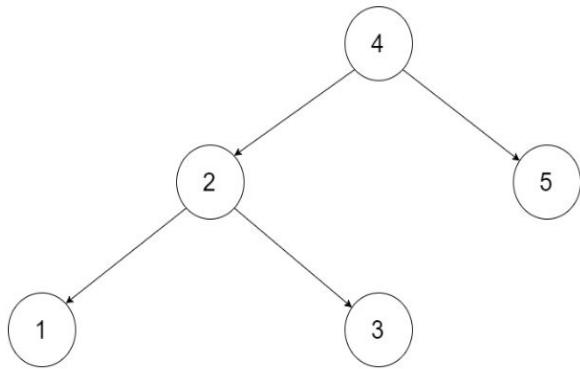
Medium 656 73 Add to List Share

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

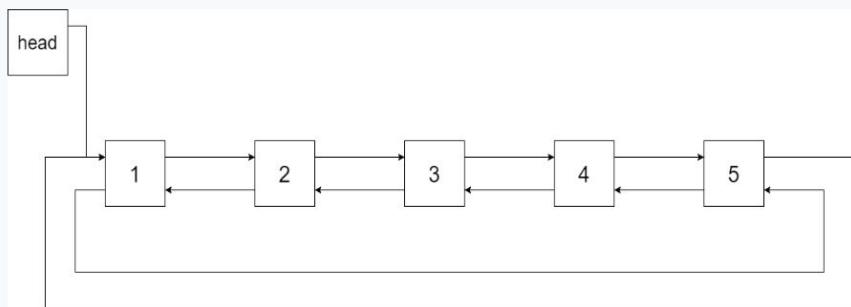
You can think of the left and right pointers as synonymous to the predecessor and successor pointers in a doubly-linked list. For a circular doubly linked list, the predecessor of the first element is the last element, and the successor of the last element is the first element.

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:

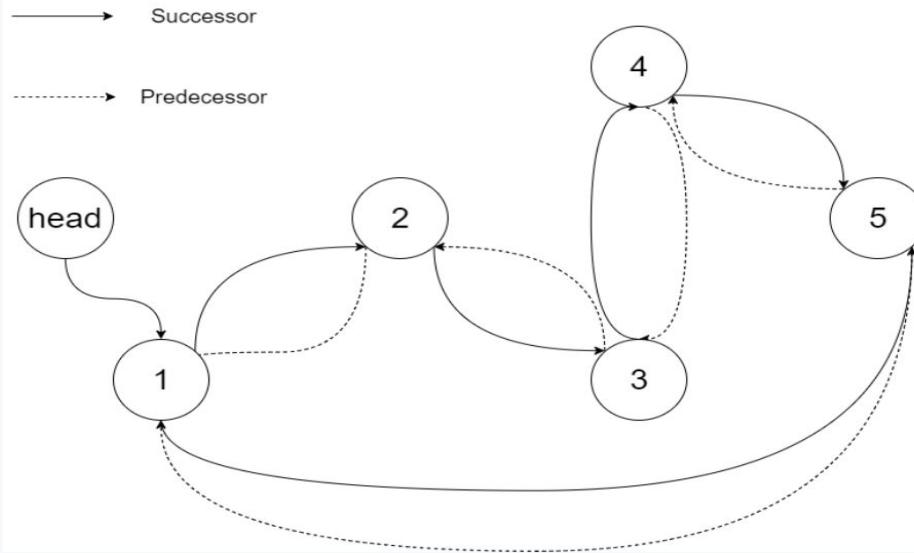


Input: root = [4,2,5,1,3]



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

Explanation: The figure below shows the transformed BST. The solid line indicates the successor relationship, while the dashed line means the predecessor relationship.



Example 2:

Input: root = [2,1,3]

Output: [1,2,3]

Example 3:

Input: root = []

Output: []

Explanation: Input is an empty tree. Output is also an empty Linked List.

Example 4:

Input: root = [1]

Output: [1]

Constraints:

- $-1000 \leq \text{Node.val} \leq 1000$
- $\text{Node.left.val} < \text{Node.val} < \text{Node.right.val}$
- All values of `Node.val` are unique.
- $0 \leq \text{Number of Nodes} \leq 2000$

```
19     sentinel = TreeNode("sentinel")
20     pred = [sentinel]
21
22     def dfs(node):
23
24         if node.left is not None:
25             dfs(node.left)
26
27         pred[0].right = node
28         node.left = pred[0]
29         pred[0] = node
30
31         if node.right is not None:
32             dfs(node.right)
33
34     dfs(root)
35     pred[0].right = sentinel.right
36     sentinel.right.left = pred[0]
37     return sentinel.right
```

Slides Created by:
Omkar Deshpande