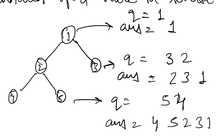
Binary tree 35

Sunday, 1 October 2023 5:29 AM

① Level Order Traversal / BFS
— using deque to print nodes at each level

② Reverse Level order Traversal
— Idea is to use deque to get result
(allows appending on leftside in O(1) time)
— we process the children of a node in reverse order

ans = deque()
q = deque()



③ Height of a tree
— Recursively do a DFS
— If tree is empty, return 0
— Otherwise, return (max(maxdepth^left, maxdepth^right)) + 1

Time: O(n)
Space: O(h)

— Another method is, level order traversal
Time: O(n)
Space: O(n)

④ Diameter of a tree
— calculating height in same recursion

— Morris traversal
— traverse binary tree using cur = root
— while cur not null
— If left child is NULL, move to right child
— If left child not NULL, find rightmost node in left subtree
— If right child of rightmost is NULL, set right to cur and move left child to current's left
— If right child of rightmost is not NULL, visit current node and move to right child
— For each visited node, calculate the left and right subtree heights using maximum func and update the diameter as max of sum of their heights and current diameter

Time: O(n)
Space: O(h)

⑤ Mirror of a tree
— Simply swap the left and right link of each node.
— If node is leaf, then do nothing

⑥ Inorder
* In case of BST, inorder traversal gives nodes in non-decreasing order.
To get nodes in non-increasing order, simply inorder traversal is reversed is used.

Recursion
— Traverse the left subtree
— Visit the root
— traverse right subtree

Iterative (using stack)
1. Create empty stack (S)
2. Initialize current node as root
3. Push current node to S and set cur = cur.left until cur is NULL
4. If current is NULL and stack not empty,
— Pop the top item from stack
— Print the popped item and set current = popped.right
— Go to step 3

Inorder Traversal Using Morris Traversal

⑦ Preorder
Recursion
— Visit root
— traverse left subtree
— traverse right subtree

* Preorder traversal is used to create a copy of tree.
Also used to get prefix expressions on an expression tree.

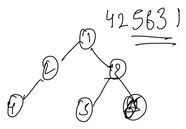
Iterative
1. Create a stack and push root node
2. Do following while stack not empty:
1. Pop an item from stack and print it
2. Push right child of popped item
3. Push left child of " "

O(n)
Space: O(h)

— Right child is pushed before left, to make sure left tree is processed first

Morris Optimization: start traversing tree from the root node, and keep printing left child while exists and simultaneous, push right child of every node in auxiliary stack.
— Once, we reach NULL, pop a right child from auxiliary stack and repeat the process until auxiliary stack is not empty

Using Morris Traversal
1. If left child is None, print current node data. Move to right child
Else, make right child of inorder predecessor point to current node.
Two case:
(a) Right child of inorder predecessor already points to current node
— Set right child to NULL
— Move right child to current node
(b) Right child is NULL,
— Set it to current node
— Print current nodes data and move to left child of current
2. Iterate until current node is NULL



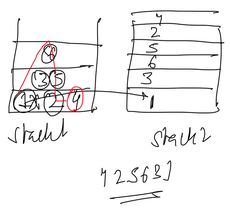
4 2 5 6 3 1

⑧ Post Order
— Recursive Method
— Traverse left subtree
— " right "
— Visit the root

* Used to delete the tree,
To get postfix expression of an expression tree

We can delete with other traversals also with extra space complexity
→ Before deleting parent delete child nodes
∴ post order is best solution

— Iterative
— Using two stacks (to make easier and efficient)
— Morris Traversal → totally opposite of inorder and reverse the result



4 2 5 6 3 1

⑨ Left View of Binary Tree
— Using BFS

⑩ Right View of Binary Tree
— Using BFS
— learn morris traversal

⑪ Bottom View / Top View
— Using Unordered_map O(n)
Space: O(n)

⑫ Zig-Zag Tree traversal
— Using two stacks
— Using deque and an extra parameter, flag

⑬ Check if tree is balanced
Naive: Get height of left subtree and right subtree using diff.
Return true if difference is not more than 1 and left + right subtrees are balanced, otherwise return false
Time: O(n²) — traversing tree twice
O(n) — space for call stack using recursion

Efficient: calculating height in same recursion (helper fun)

Better: Idea is to use a pair [Balanced or not, height]
— Define a helper function

⑭ Diagonal Traversal
— store root in deque, initialize a list, res
— while q:
popleft from q and make it cur
while cur:
if cur.left → append to queue
res.append(cur.data)
cur → cur.right

⑮ Boundary Traversal
1. Right Boundary — Go Right Right until no Right. Don't include leaf nodes
2. Left Boundary — " Left Left " " Left. " " " "
3. Leaf Boundary — Do inorder/preorder, if leaf node add to list
4. We pass array as reference, so its same memory location used by all functions

⑯ Construct Binary Tree from string with bracket representation
Algo
1. The first element is root
2. If next two consecutive elements are "(" and ")", this means no left child otherwise we will create and add left child recursively
3. Once left child is added recursively we will look for consecutive "(" and add the right child to parent
4. Encountering ")" means the end of either left or right node and we will increment the start index
5. Recursion ends when start ≥ end index

⑰ Convert Binary Tree to Doubly Linked list
— Do traversal of tree (inorder)
— In helper func, go to left most node (explain in BST)

⑱ Convert given tree to its sum tree
— Do traversal of given tree
— Store old_value of current node
— Recursively call for left and right and change value of node as sum of values returned by recursive calls
— Finally return sum of new value and old_value

```
def toSumTree(root):
    if root is None:
        return 0
    old = root.data
    root.data = toSumTree(root.left) + toSumTree(root.right)
    return old + root.data
```

O(n)
Space: O(1)

⑲ Find min number of swaps required to convert a Binary Tree to BST

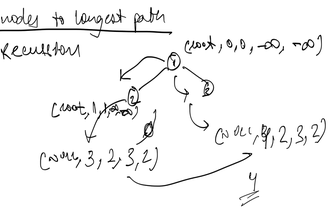
⑳ Check if binary tree is sum tree or not

① Check if all leaves are at same level:
→ Get leaf at max distance
— root is None: return 0
— traverse for left and right recursively
— return 1 + max(left, right)
→ Get leaf at min distance
— root is None: return infinity
— root's left and right is None: return 1
— traverse for left and right recursively
— return 1 + min(left, right)

② Check if Subtree
a) If two tree identical:
— If both roots None: True
— If one root None: False
— return (root1.data == root2.data) and (root1.left and root2.left are identical and root1.right " root2.right " " ")

b) isSubTree(T, S) → S is present in T or not
— If S None: True
— If T None: False
— If T and S identical: True
— return (T.left and S) identical or (T.right and S) identical

③ Extension of above problem
Check if binary tree contains duplicates subtree of size 2 or more
→ Efficient: Serialization and Hashing
— Serialize subtree and strings and store in hash tables
— Once we find a serialized tree (which is not leaf) already existing in hash table, we return true.

→ Find duplicate Subtree

④ Sum of nodes to longest path
— Recursion