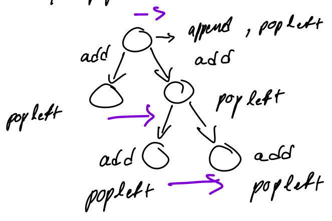
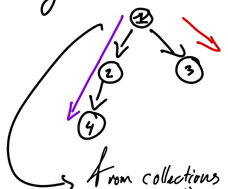
Binary Tree level order traversal

AKA BFS.
Use a deque. pop left



```
from collections import deque
    res:[]
    q = deque( )
    if root :
        q.append(root)
    while q:
        val = []
        for i in range(len(q)):
            node = q.popleft( )
            val.append(node.val)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
        res.append(val)
    return res
```

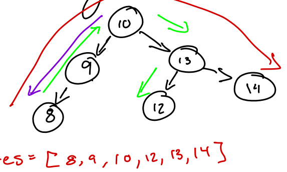
Binary Tree Preorder Traversal



res = [1, 2, 4, 3]

```
from collections import deque
-> q.deque(), res = [ ]
-> if root:
       q.append(root)
-> while q:
       for i in range(len(q))
           node = q.pop()
           if node.right:
               q.append(node.right)
           if node.left:
               q.append(node.left)
           res.append(node.val)
   return res
```

Binary Tree in-order traversal:

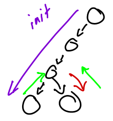


res = [8, 9, 10, 12, 13, 14]

```
def inOrderTraversal
    res, stack = [], []
    cur = root
    while cur or stack:
        while cur:
            stack.append(cur)
            cur = cur.left
        cur = stack.pop()
        res.append(cur.val)
        cur = cur.right
    return res
```

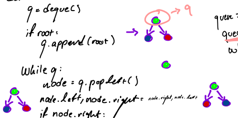
Binary tree in-order iterator:

```
class BSTIterator:
    def __init__( ):
        self.stack = []
        cur = root
        while cur :
            stack.append(cur)
            cur = cur.left
    def next( ):
        res = self.stack.pop()
        cur = res.right
        while cur:
            self.stack.append(cur)
            cur=cur.left
        return res.val
    def hasNext( ):
        return self.stack != [ ]
```



invert tree

```
def invertTree :
    q = deque()
    if root:
        q.append(root)
    while q:
        node = q.popleft( )
        node.left, node.right = node.right, node.left
        if node.right:
            q.append(node.right)
        if node.left:
            q.append(node.left)
    return root
```



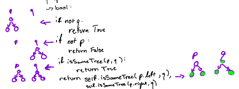
q -> [●   ]
q -> [●, ●]

same tree -> BFS

```
def sametree(self, p : Optional[treenode],
                   q : Optional[TreeNode])
                   -> bool:
    queue = deque()
    queue.append((p,q))          q = [(●, ●)]
    while queue :
        pNode, qNode = queue.popleft   -> (●, ●) -> unwrap tuple
        if not pNode and not qNode:    -> cannot compare
            continue
        elif not pNode or not qNode :  -> mismatch
            return False
        else:
            if pNode.val != qNode.val:  -> mismatch
                                        MUST CHECK IF NOT NONE FIRST
                return False
            else:
                q.append((pNode.left, qNode.left))    q = [(●, ●)]
                q.append((pNode.right, qNode.right))  q = [(●, ●), (●, ●)]
```

subtree of another tree

```
def isSubTree(self, p: optional[TreeNode]
                    q : optional[TreeNode])
                    -> bool:
    if not q.
        return True
    if not p :
        return False
    if isSameTree(p, q ):
        return True
    return self.isSameTree(p.left, q),
        self.isSameTree(p.right, q)
def isSameTree(self, p, q) -> bool :
```



return True

insert into binary tree (iterative DFS)

```
def insertIntoBST (self, root, val) :
    parent = None
    child = root
    newNode = TreeNode(val)
    q = deque( )
    left = None
    if root:
        q.append(root)
    while q and child:
        child = q.pop( )
        parent = child
        if newNode.val < child.val :
            child = child.left
            q.append(child)
            left = True
        else:
            child = child.right
            q.append(child)
            left = False
    if left is True :
        parent.left = NewNode
    elif left is False :
        parent.right = newNode
    else:
        root = newNode
```

delete node in BST (recursive DFS):

```
def deleteNode(self, root, key):
    if not root:
        return None
    if root.val == key:
        if not root.right:
            return root.left
        if not root.left:
            return root.right
        if root.left and root.right:
            temp = root.right
            while temp.left:
                temp = temp.left
            root.val = temp.val
            root.right = self.deleteNode(root.right, temp.val)
    elif root.val > key:
        root.left = self.deleteNode(root.left, key)
    else:
        root.right = self.deleteNode(root.right, key)
    return root
```

Iterative DFS vs BFS

Cornell Univ: https://youtu.be/Ba53oitxDLo?si=dUsxEg8HI0wQ01hh

- ↳ Iterative DFS processes nodes from right to left
- ↳ recursive DFS processes nodes from left to right
- ↳ to maintain order
  - ↳ reverse the way nodes are added to the stack.

iterative DFS vs BFS:
- ↳ DFS: pop right (stack or deque)
- ↳ BFS: pop left

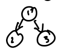
* iterative DFS reverse node addition

Find duplicate subtrees: -> pre reqs: Same Tree + Subtree of another tree
↳ BFS ↳ BFS

↳ Find multiple and output root

Serializing a Binary Tree & Deserializing:

↳ serializing.... What's that? => compress a BST to a string. Parsing a BST as a string => base 64 encode string?. => interesting stuff....
=> machine learning decision tree compress and encode.



root = [1, 2, 3, null, null, null, null]
output = "1, 2, 3, null, null, null, null"

↳ recursive DFS => DFS from Grad Algo
↳ iterative DFS => Explore from Grad Algo
-> whole tree -> recursive DFS
↳ def dfs (node):

```
def dfs (node):
    if not node:
        return "null"
    s = ",".join([str(node.val), dfs(node.left), dfs(node.right)])
```

```
res = []
def dfs(node):
    if not node:
        res.append("null")
        return
    res.append(str(node.val))
    dfs(node.left)
    dfs(node.right)

dfs(root)
return ",".join(res)
```

deserializing a binary tree => s = "1,2,3,null,null,null,null"
root = [1,2,3,null,null,null,null]
↗ TreeNode

```
def deserialize(self, data):
    vals = data.split(",")
    self.i = 0          -> global variable that holds the current index of the string

    def dfs():
        if vals[self.i] == "null":
            self.i += 1
            return None
        node = TreeNode(int(vals[self.i]))
        self.i += 1
        node.left = dfs()
        node.right = dfs()
        return node

    return dfs()
```