**Lecture 10**

**Tree**

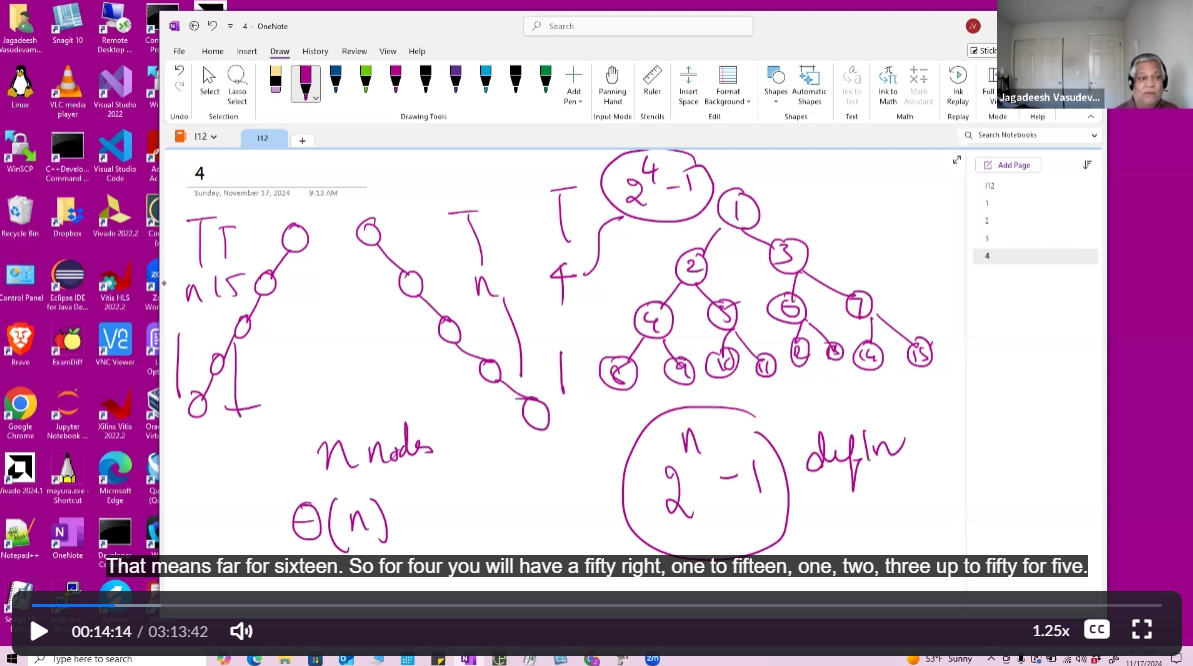
Tree has n nodes and n-1 edges. Only root does not have parents. Tree cannot have a loop. We always keep a pointer to the root. We traverse the tree from the root. A tree can have any number of edges. Binary tree has up to 2 kids for every node. Most of operations in binary tree is log(n).

A whiteboard with writing on it

Description automatically generated

Tree can only left kids or only right kids. Then, it will be a linked list.

If we stack the tree so that each node has 2 kids. Then for n levels, there are 2\*\*n – 1 nodes in the tree.

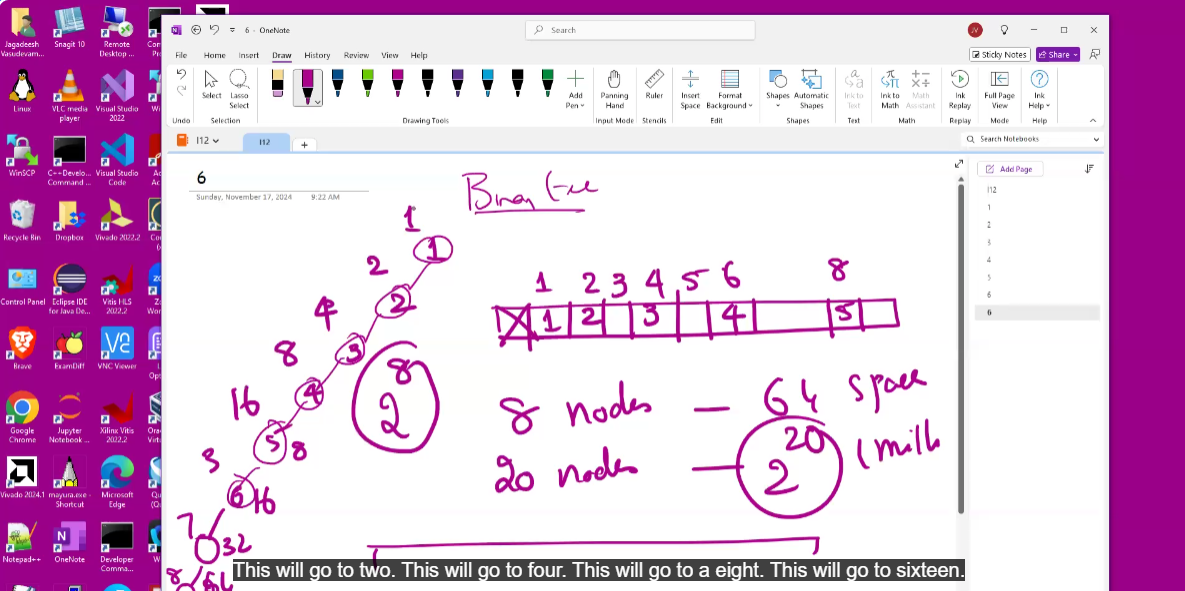


Height of the tree is the worst case in O(n) since tree has only left kids or only right kids but in the best case, it’s log(n) where a node has both left and right kids.(Space complexity)

List growing space is 2\*n.

…

In heap, we can do everything in a array. Right Kid is 2\*I and left kid 2\*I + 1. Father of the node is i//2. But why do we need pointers for Binary Tree?

Binary tree can have only right kids or only left kids, if we represent that in a array, for 8 nodes, we need 64 space. For 20 nodes, we need 2\*\*20 space(1 million). It’s a inefficient use of space so we use pointer to represent a BT.

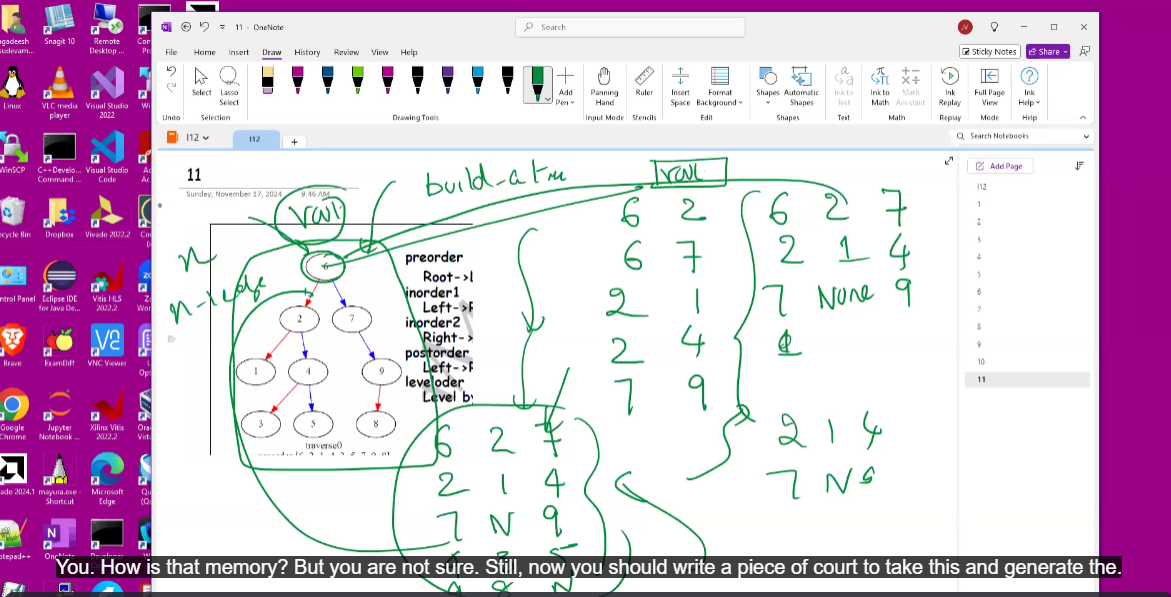
For linked list, we require only pointer in the data structure. If there is a pointer in the node, previous pointer is one before the node and there is a after pointer after the node so we can access everything. Hence one pointer in the data structure is sufficient.

A whiteboard with green text and symbols

Description automatically generated

Similar to this, tree has only 2 pointers in it’s data structure. In BT, for n nodes, you require 2\*n extra space. In linked list, for n nodes, you have n extra space(because of the pointer).

For traversal of tree, we have to read the inputs from a file and build a tree. We traverse the tree in different ways based on the problem given to us. Lly to graph. Examples are given in the picture.



In graph, shortest path -> BFS. longest path -> DFS.

In tree, we do different traversal for diff questions.

**PreOrder**

A screenshot of a computer

Description automatically generated

For the tree in the above photo,

Preorder traversal is print the node before you call left, right child

Node, Left, Right

Def preorder(root r):

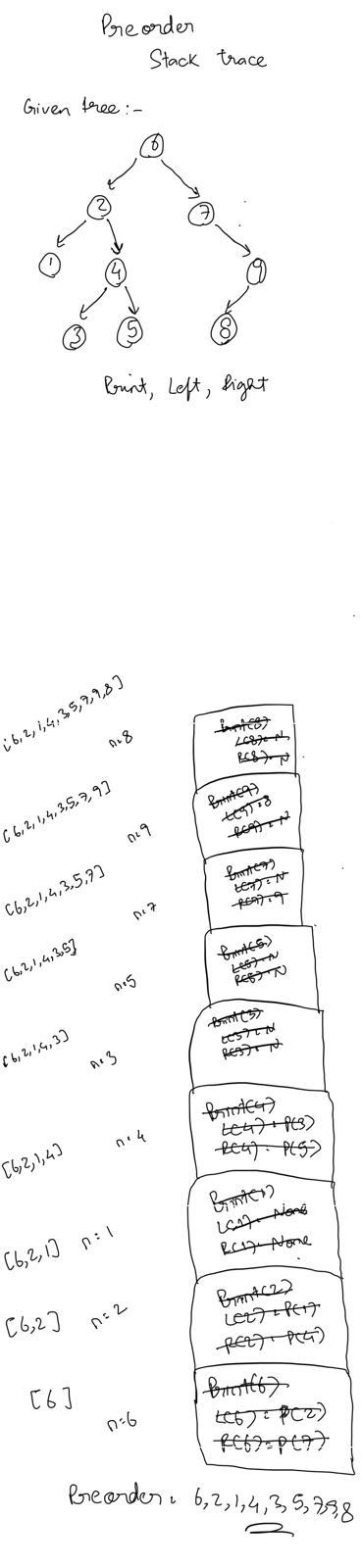
If r!= None:

Print(r.value)

Preorder(root.left)

Preorder(root.right)

Stack Trace:



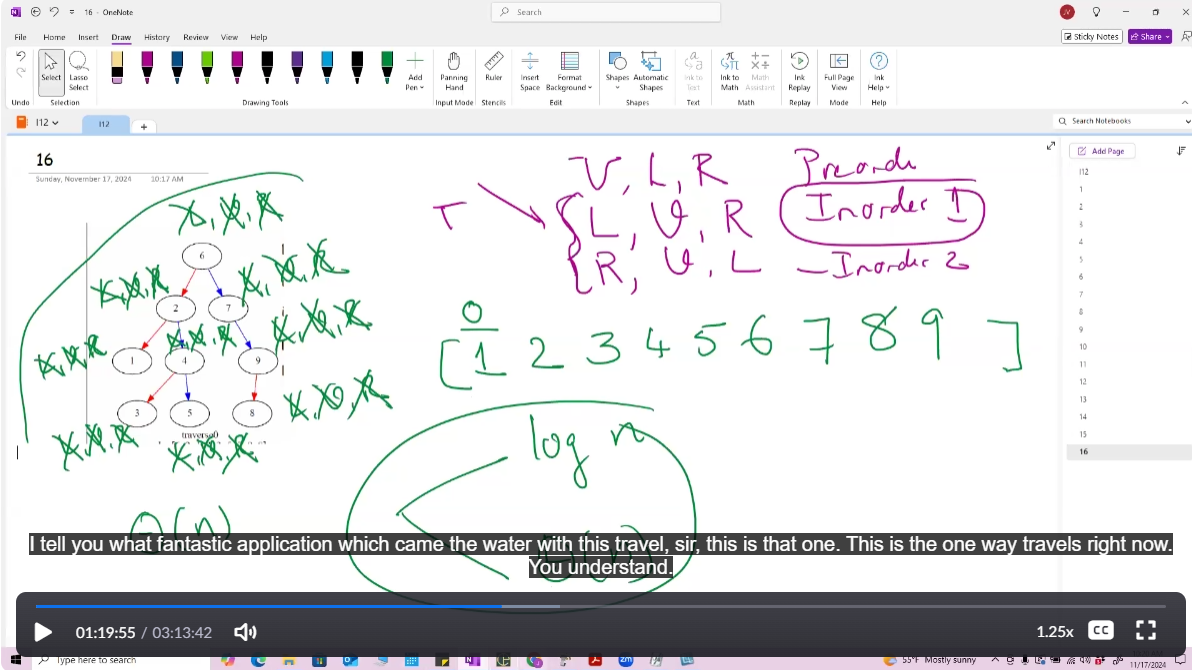
For above tree, Inorder traversal -> 6,2,1,4,3,5,7,9,8

Each node is visited exactly once.

The complexity of this code in the best case scenario where the BT is balanced is O(n) because we visit each node once and for space complexity, the stack trace is of the height logn where n is the number of nodes(balanced tree). In the worst case scenario, where the tree has only left kids or only right kids, height of the stack trace is n so the space complexity is O(n).

When we use recursion, in the worst case, height of stack trace is n. so, we write a iterative solution using our own stack trace(Stack).

**InOrder**

****

Inorder traversal is you call left, then print node and go right child

Node, Left, Right

Def inorder(root r):

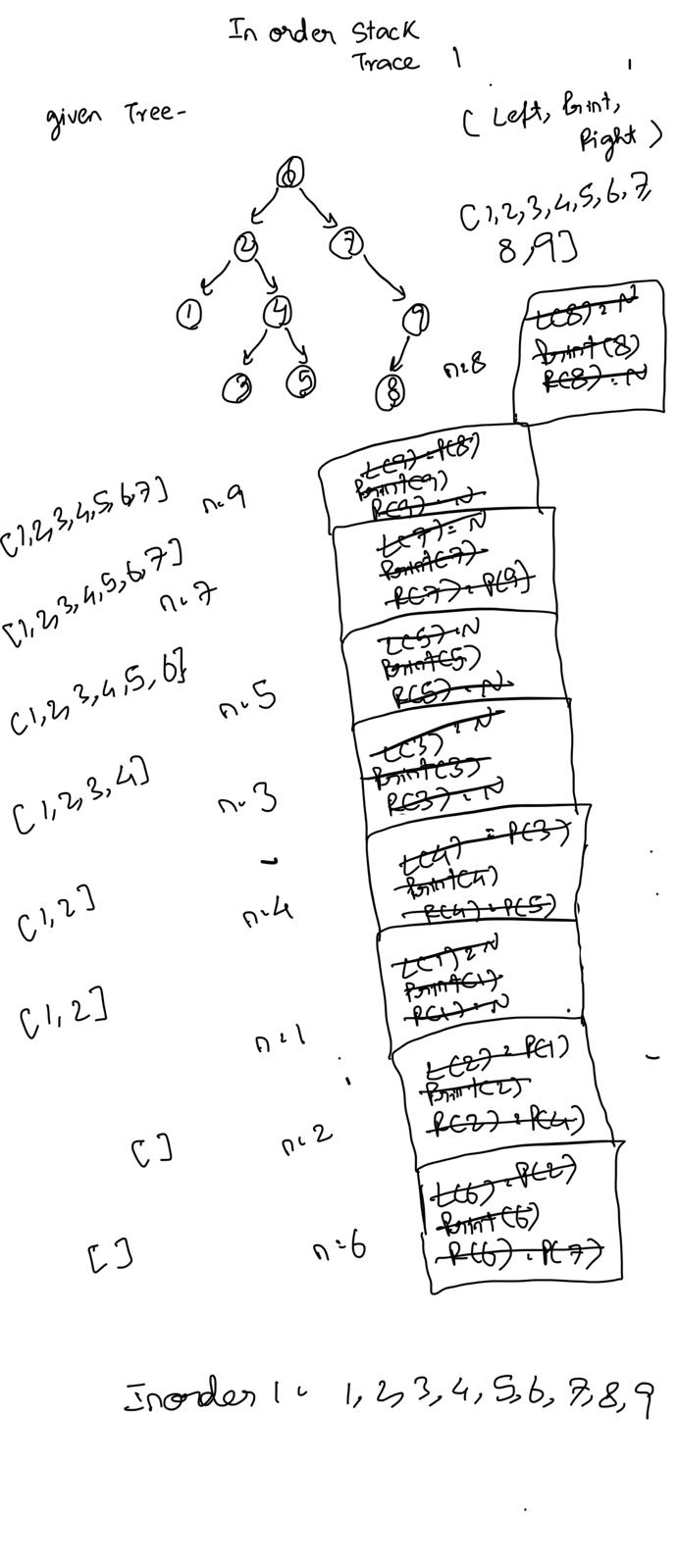
If r!= None:

inorder (root.left)

Print(r.value)

inorder (root.right)

Stack Trace:

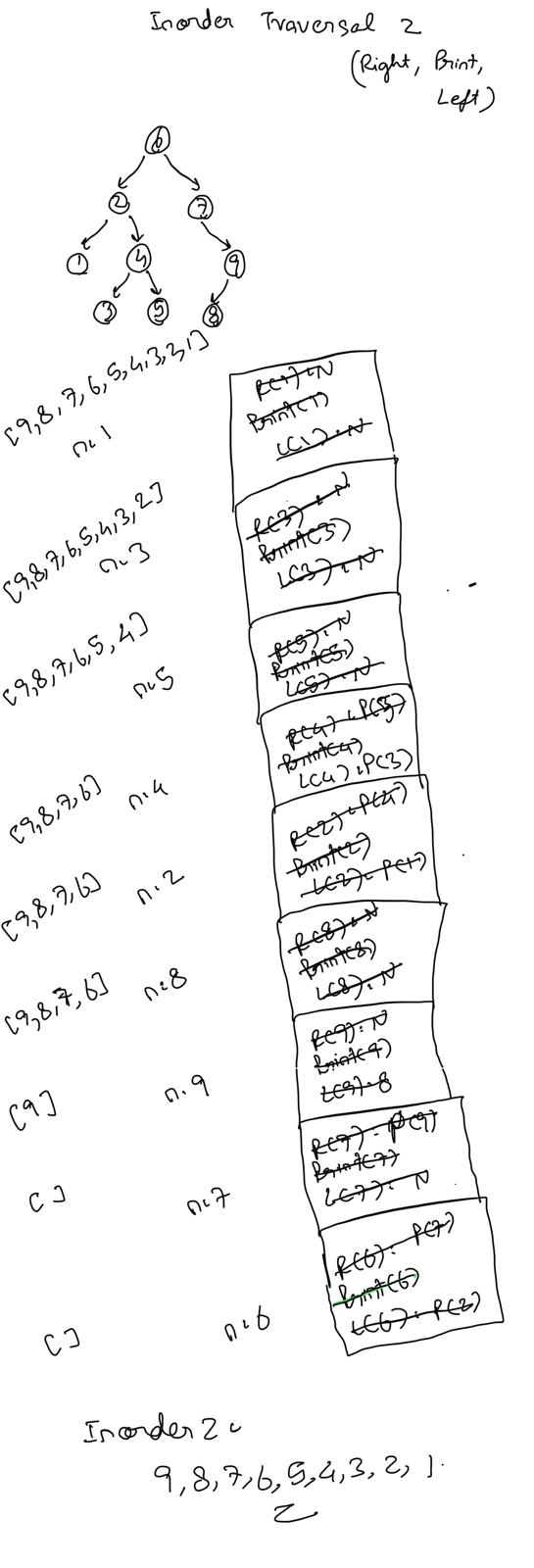


For above tree, inorder traversal -> 1,2,3,4,5,6,7,8,9

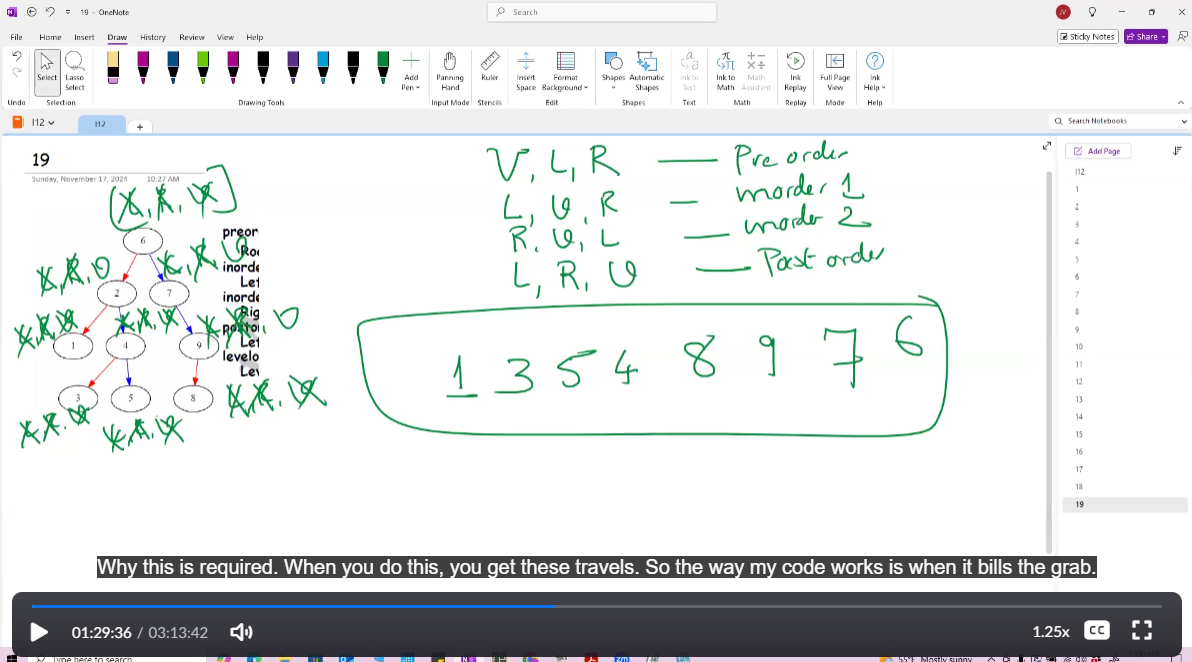
Each node is visited exactly once.

Inorder 2 is Right,Print,Left.

Stack Trace-



**PostOrder**



postorder traversal is you call left, right and then print node

Left, Right, Node

Def postOrder(root r):

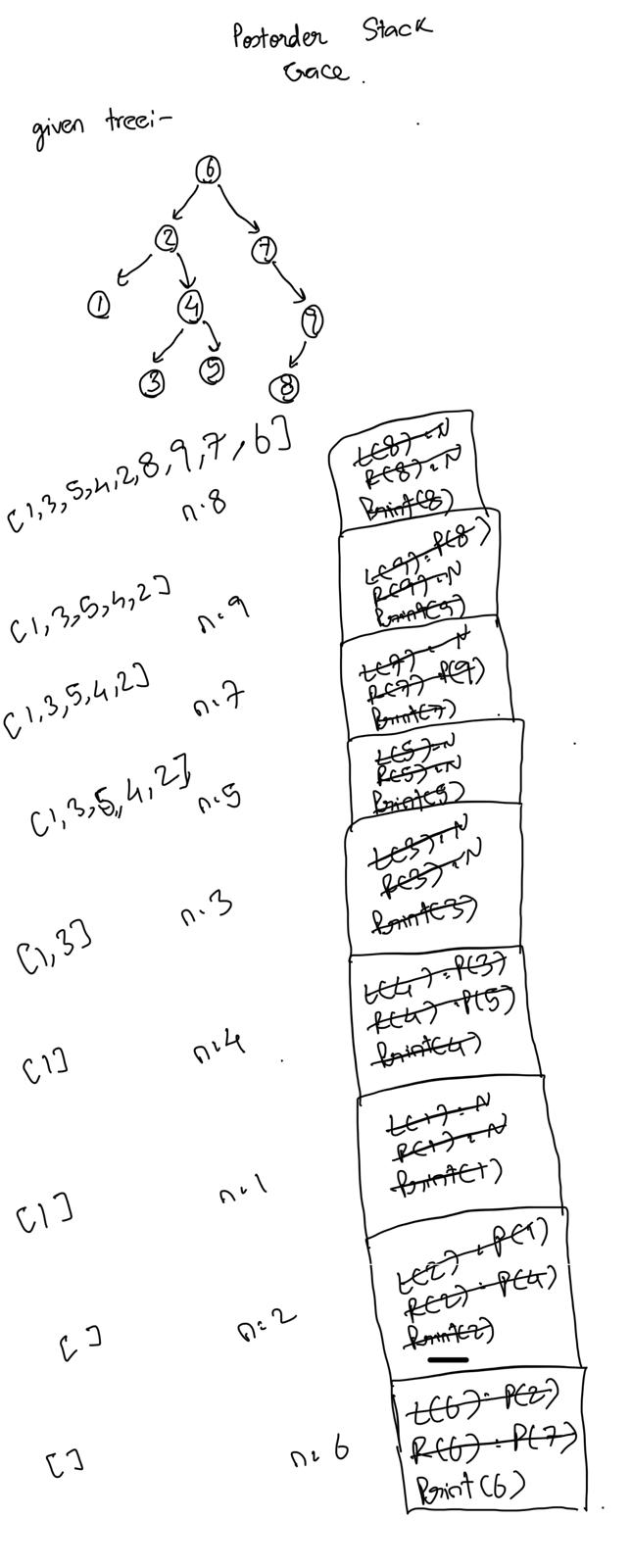
If r!= None:

postOrder (root.left)

postOrder (root.right)

Print(r.value)

Stack Trace:



For above tree, postorder traversal -> 1, 3, 5, 4, 2, 8, 9, 7, 6

Professor forgot 2 in the class

Each node is visited exactly once.

[Assignment] : postOrder -> Implement Iterative algorithm

Given a tree, print the level order traversal of the tree.

For this algo, we can use a queue and do the same thing as BFS

**Pseudo code**

Def levelOrder(root r):

q = deque()

q.append(root)

while(q):

Node n = q.popleft()

Print(n.value)

If n.leftChild:

q.append(n.leftChild)

if n.rightChild:

q.append(n.rightChild)

This algo will print all nodes in level order.

But if we want to print level by level

Then we put None in the queue after putting the root, when we deque none, we print a new line and enqueue None.

**Pseudo code**

Def levelOrder(root r):

q = deque()

q.append(root)

q.append(None)

while(q):

Node n = q.popleft()

If n is None:

Print()

q.append(None)

Print(n.value)

If n.leftChild:

q.append(n.leftChild)

if n.rightChild:

q.append(n.rightChild)

**Postorder traversal application – deleting a tree**

A whiteboard with purple writing

Description automatically generated

For the whole tree, post order traversal will visit left and right of every node and it will arrive at the node which don’t have a right or left child so, we can delete them. After it will go to the parents of the deleted nodes, which can also be deleted without any issue. For every node, it will visit left first, then right, if left and right are visted then, delete the node.

Def deleteTree(root r):

If r is not None:

deleteTree(r.left)

deleteTree(r.right)

Delete(r) // delete node

**Inorder traversal application – Sorting**

The are two inOrders 1 and 2

Inorder 1 – Left, Print, Right

Inorder 1 will sort ascending order

Tree stacktrace – Prof showed

Inorder 2 – Right, Print, Left

Inorder will sort descending order

Tree stacktrace –A diagram of a tree

Description automatically generated

For this to work, tree has to be a BST(Binary Search Tree). In a BST, for each node, it’s left sub tree nodes should be lesser than the node and Right sub tree nodes should be greater than the node.

We can check if the tree is a BST using inorder traversal

Def checkBST(root r):

If r is None:  
return

checkBST(r.left)

print(r.value)

checkBST(r.right)

Then check if it’s ascending order in const time to prove if it’s a BST.

**preorder traversal application – Print all paths from root to leaf**

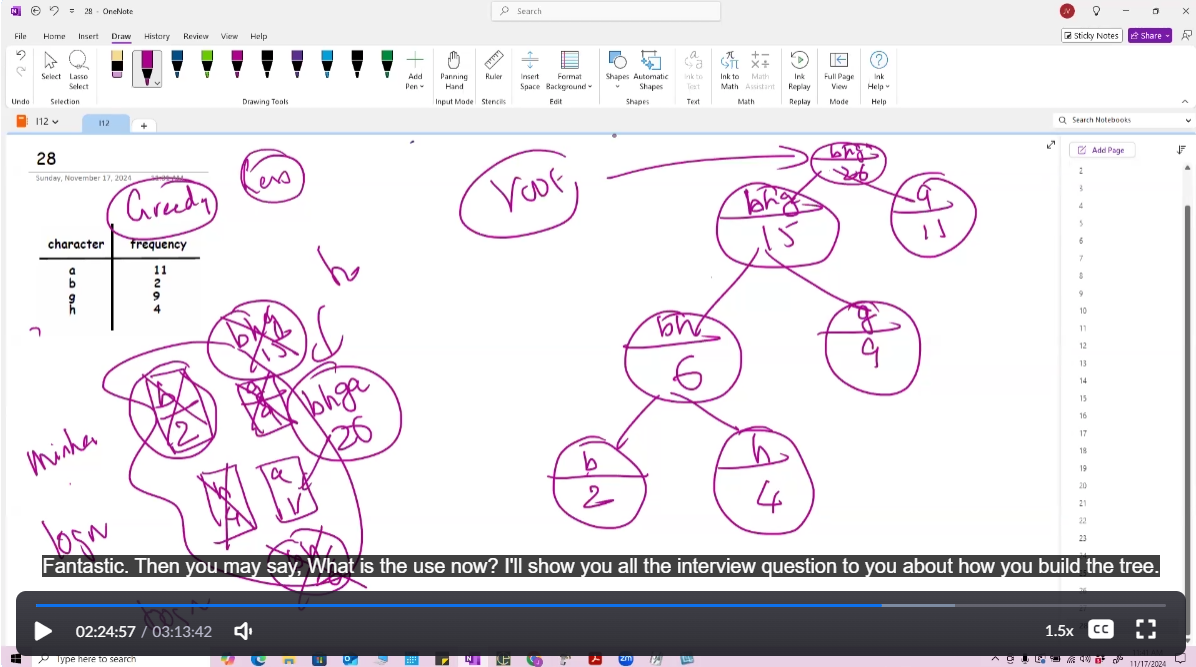
for the n+1 nodes, there are n/2 paths from root to leaf

we use a python list for this and preorder traversal. We keep a level counter(i), for every node we visit, new lv = prev lv + 1 and we put the list[lv] as the value of the node. If we reach the node with no right and left child, we print the list.

**Huffmann encoding**

In World War time, for sending messages, Germany encrypts it. So lets say the message will be from a to f, and we know how much time each letter occurs

We should know ASCII for this, ASCII will take 8 bits



First from the words, we find the frequency of all the words. Then, construct a min heap of it. Get the root of min heap, create a node and delete that from min heap(logn time). Take the new root and delete it from the min heap. Now, create a new node (bg) and add the weights to create bg:2 and connect it to b node and g node. Then insert the new node into the min heap. Similarly take all nodes from min heap using the algo and make a binary tree. The algo ends when min heap has only one node(bfga:26).

The right child paths have 1 and left child paths have 0.

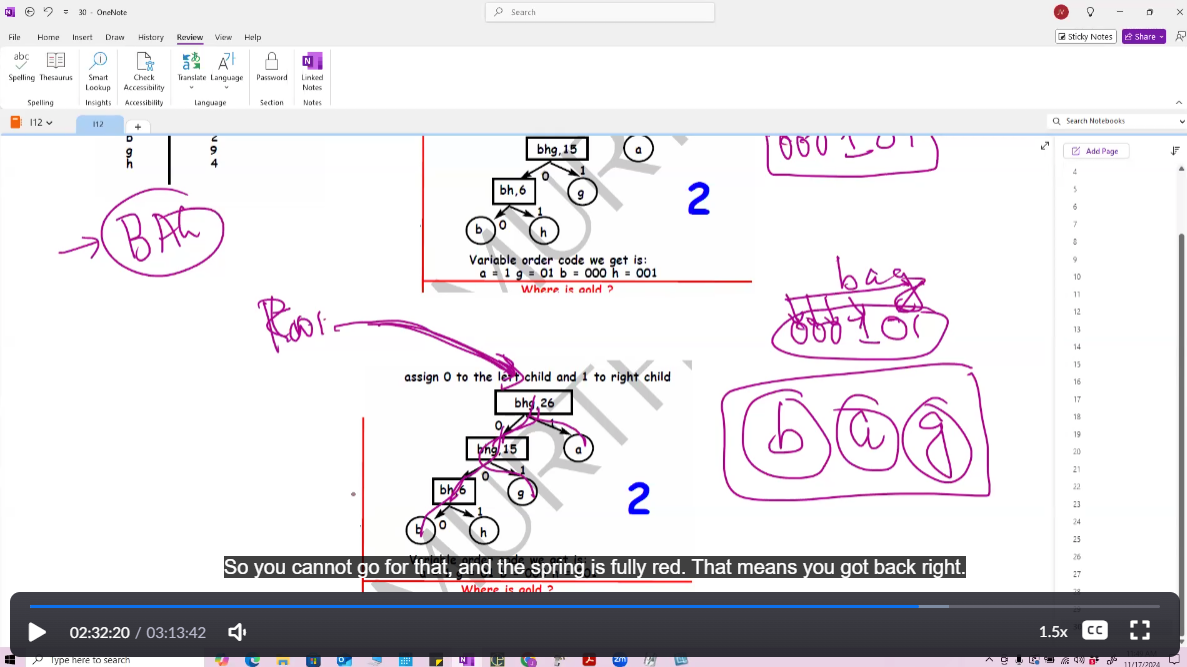
After this, find all the root to leaf paths by following 0 and 1.

Each leaf(each letter) should have a encoding now.

a= 1, g = 01, b= 000, h = 001

Use these encoding to form the encoded word.

**Huffmann decoding**



Given the binary tree, we have to decode the nodes and their frequencies then form the words. We are also given the encoded word. (000101)

We traverse the tree using the encoded word. 0 -> 0 -> 0

0 means go left

1 means go right

When you reach the node with no right and left child then that node is the word.

0 0 0 – b

1 – a

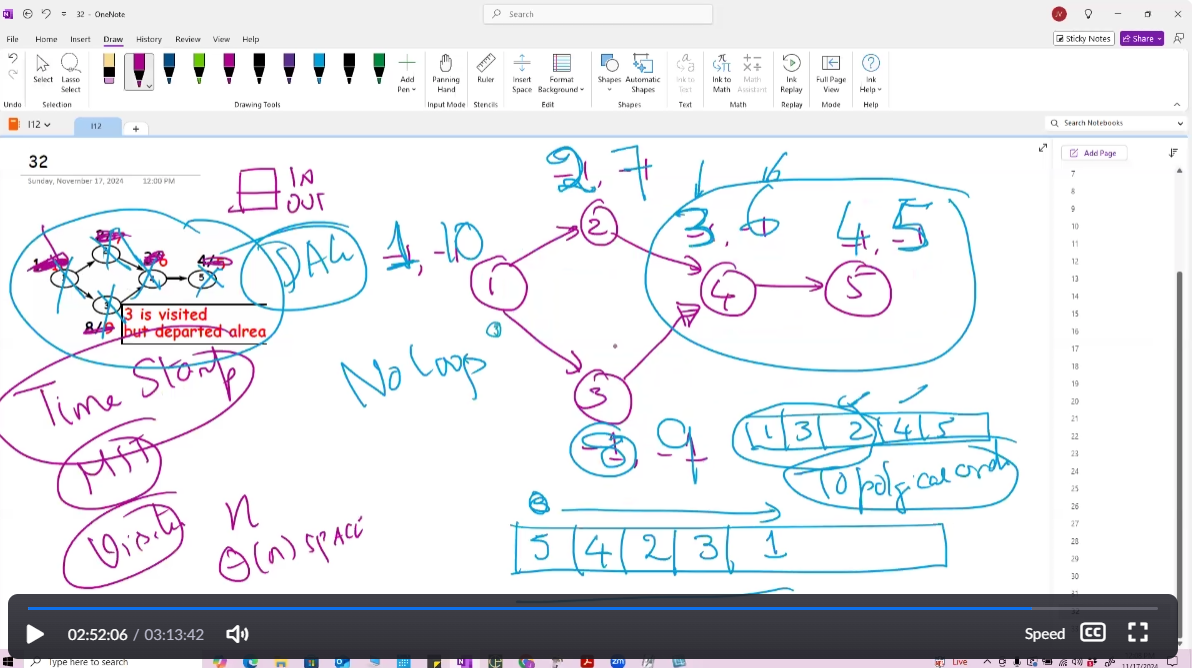
0 1 – g

Decoded word – bag

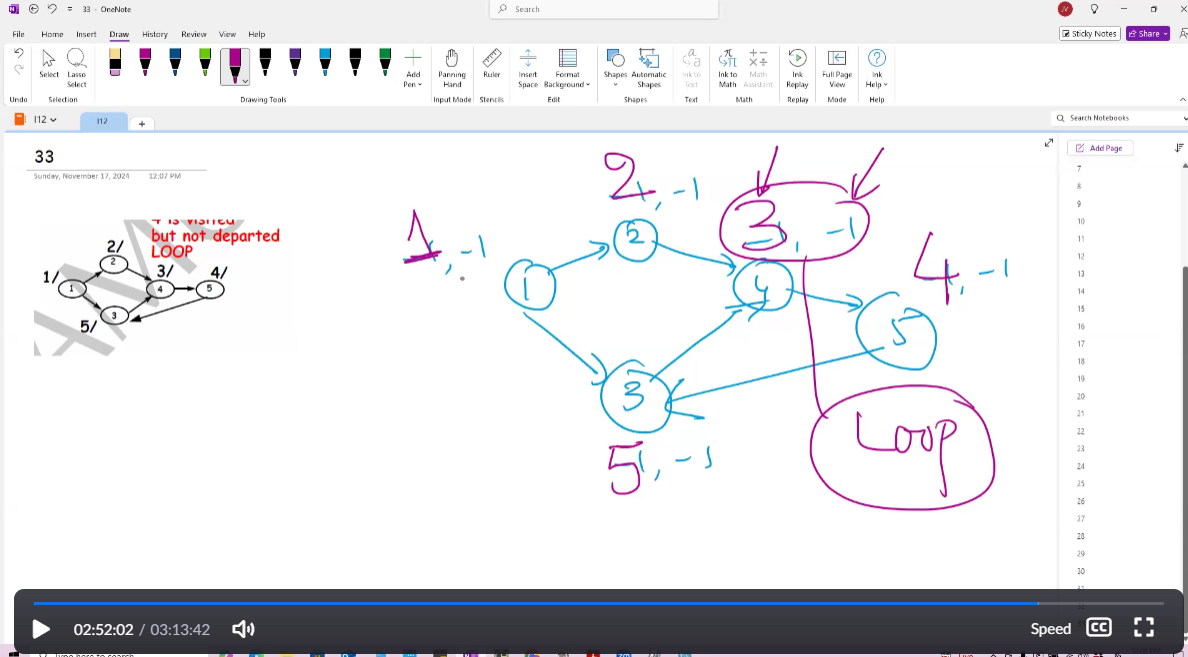
We use binary tree because we need to encode with 0 and 1.

**DFS(Depth First Search)**

In DFS, we start from a node and go on the last node in the depth from that node and then find alternate paths after. We can find the longest path using DFS. We can also find the loop using DFS. We use a algo called time stamp algorithm. If there are n node, we need 2n space which is O(n) space



We have two things [In,Out] for each node. We start from the node and keep on going till the end. While going, we fill In incrementally starting from 1. If I can’t go anywhere else, now I keep an extra python list ds and append the node name. After we reach the end, we backtrack and add the node value in the list, and fill Out incrementally, we look for a alternate path where IN is -1. When that node joins with the already visted node where in,out are filled. Then, we fill the OUT of the new node and algo ends there. In the list, node are in a topological order.



For a graph with a loop, we repeat the same process but, after a while, you will enter a node again(In is not -1) while OUT is -1. Without exiting, you are entering the node again. This means that the graph has a loop.