**Data Structures**

We usually work on data while coding, so how it will organize the data in the main memory is all about data structures. This helps in performing any action to the data easily and more efficiently.

**Good Code –** 1.scalable – time and space , 2.readable

**Big O –** depends on how many steps we are performing

O(n!) > O(2n) > O(n2) > O(n log n) > O(n) > O(log n) > O(1)

***Memory***

If I want to store a list in memory : (Visualisation is something like this)

608

6071

606

605

604

603

602

601

0

0

0

0

0

0

0

0000 0010

0000 0011

0

0

0

0

0

0

0000 0010

0000 0100

0

0

0

0

0

0

0

0

LIST = [2,3,4,5] // memory slot is of 1 byte and imagine this is 32 bit system

**LIST**

Access – O(1)  
Update – O(1)  
Traverse – O(n)  
Search – O(n)  
Delete – O(n)  
Insert – O(n),,,,, but in python 🡪 O(n) and O(1) ; depending upon the situation  
len(L) – O(n) //python

In python : list -> dynamic memory allocation  
for 3 elements it already reserve space for 4 elements, for 5-> 8  
21 🡪 2 // till 2 elements 2 spaces will be reserverd

22 🡪 4 // till 4 elements 4 will be reserved

23 🡪 8 // from 5-8 elements 8 spaces will be reserved

24 🡪 16

25 🡪 32

Thereby in pyhton we can freely append in the array without increasing the time complexity for a certain number of times.

Delete -  
First - O(n), Middle – O(n), End – O(1)

**Stack Queue**

Generally for stack we use array and for queue we use Linked List.  
Because in stack we need to perform   
1) pop - delete the last element, which takes O(1) time in array  
2) push – insert at the end, which usually takes O(1) in array in python  
3) top – access the last element , O(1)

In Queue if we use Linked List:

1. Enqueue – insert at the last, if we have the tail of LL, O(1)
2. Dequeue – delete the beginning, we just need to shift the head, O(1)
3. Access the last element – tail, O(1)

But if we use array here:  
1) Enqueue - insert at the last , O(1)  
2) Dequeue – delete the beginning, O(n)  
3) Access the last element – O(1)

**Hash Tables** –has O(1) time complexity for searching

Problem of hash functions – collisions

**Binary Tree**: can be implemented using doubly LL.

Full Binary Tree – a node can have 0 or 2 child.

Complete Binary Tree – leaf nodes should be left aligned if a particular lvel is not completely filled.

Perfect Binary tree – all the leaf nodes should be at same level and all the level should be completely filled.

Balanced Binary Tree – height of the left and right sub-trees should differ by 0 or 1 only.

Degenerate Binary Tree - every node has 1 child only.

When we delete a node from Binary Search Tree, there can be 3 conditions :   
1. IF the node has 0 child: simply delete that last node  
2. If the node has 1 child :

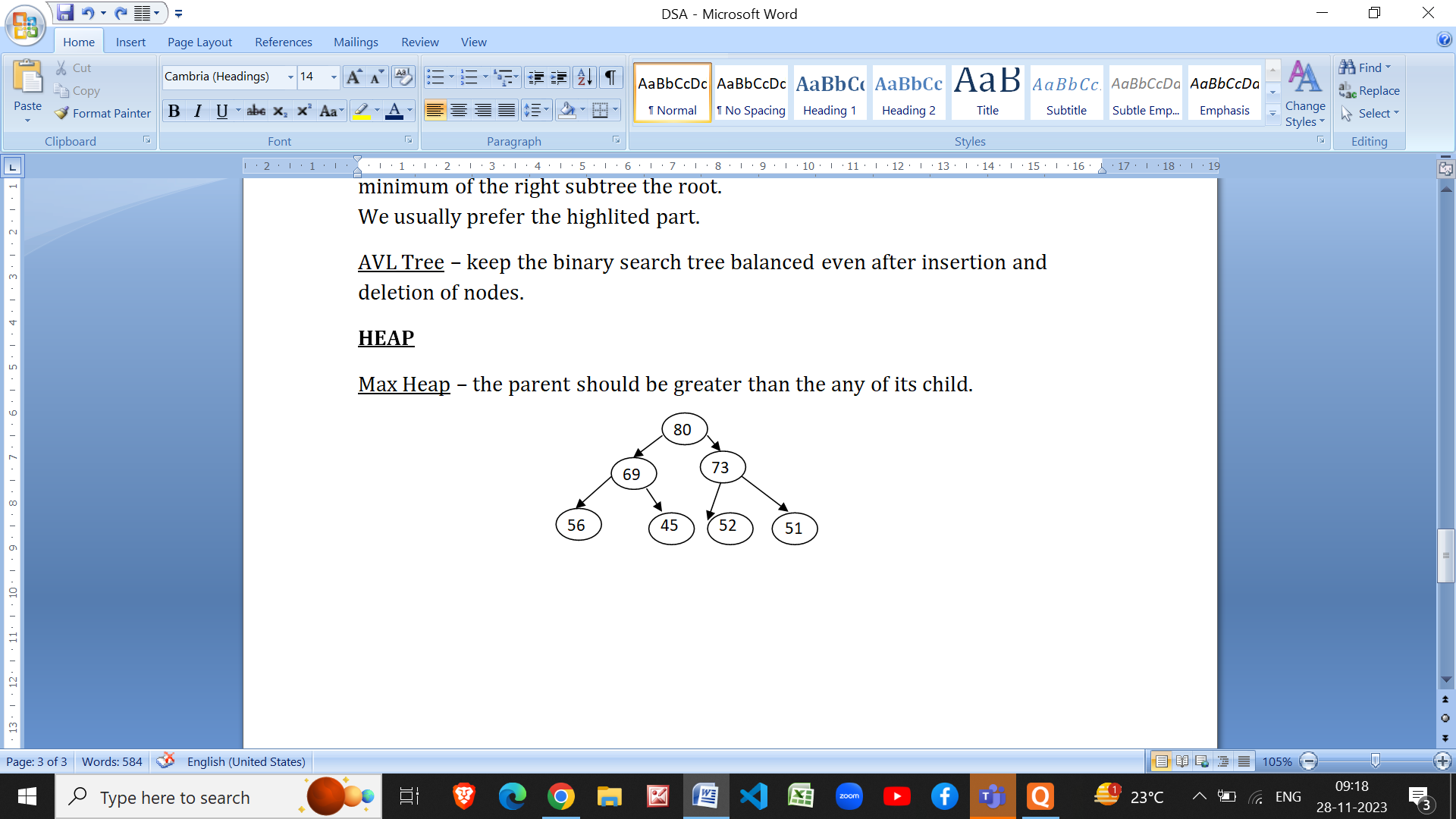
3. If the node has 2 child : make the maximum of left subtree the root or minimum of the right subtree the root.  
We usually prefer the highlited part.

Binary Search Tree – left node should be smaller than the right node.

AVL Tree – keep the binary search tree balanced even after insertion and deletion of nodes.

**HEAP**

Max Heap – the parent should be greater than the any of its child.



Min Heap – the parent node should be smaller than all its child.

They can be represented using an array.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 80 | 69 | 73 | 56 | 45 | 52 | 51 |  |  |  |

Left child at – 2i+1

For Complete binary tree

Right child at – 2i + 2

Search Complexity of Heap – O(n)

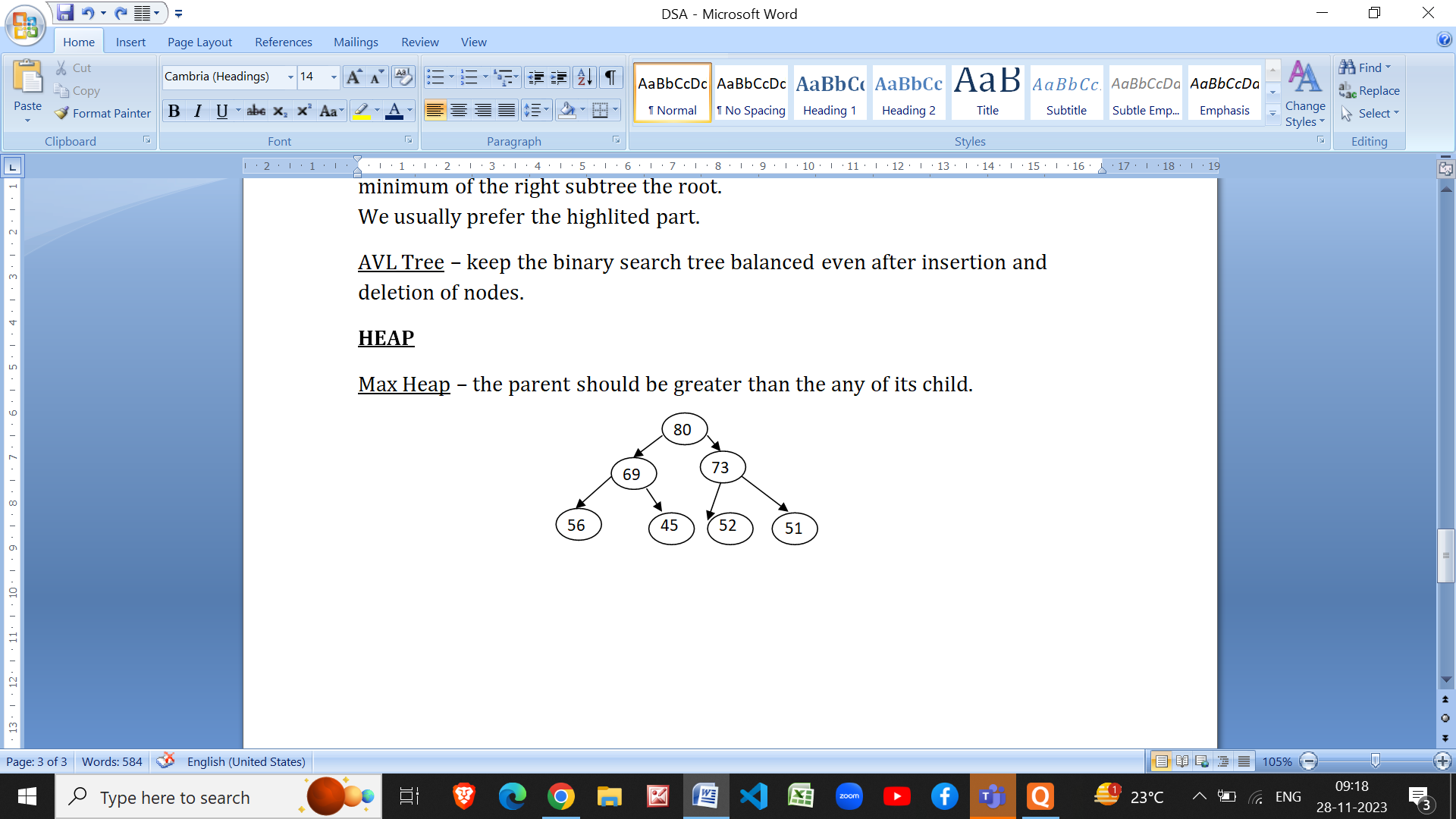
Insertion in Heap(Max Heap) – First add the element in the end of the array which will cost O(1) time. Then compare it with its parent element(if index is even, (i-2)/2), if parent is smaller than the element then swap it with that index element. Keep this doing till we don’t reach the root or parent becomes greater than the child. **🡪 O(log n)** //*it will take time according to the height of the tree.*

Deletion in Heap – **O(log n)**

* In heaps we can only delete the root element to keep the concept of heaps valid.
* Steps : First Swap the root node with the last elemet of the tree
* Check with its child, if both the child is greater then swap the largest child with that element
* Keep doing this untill the parent becomes greater than both the child.

If we keep on deleting roots from the heap(max)(complete binary tree), we will get the sorted array.

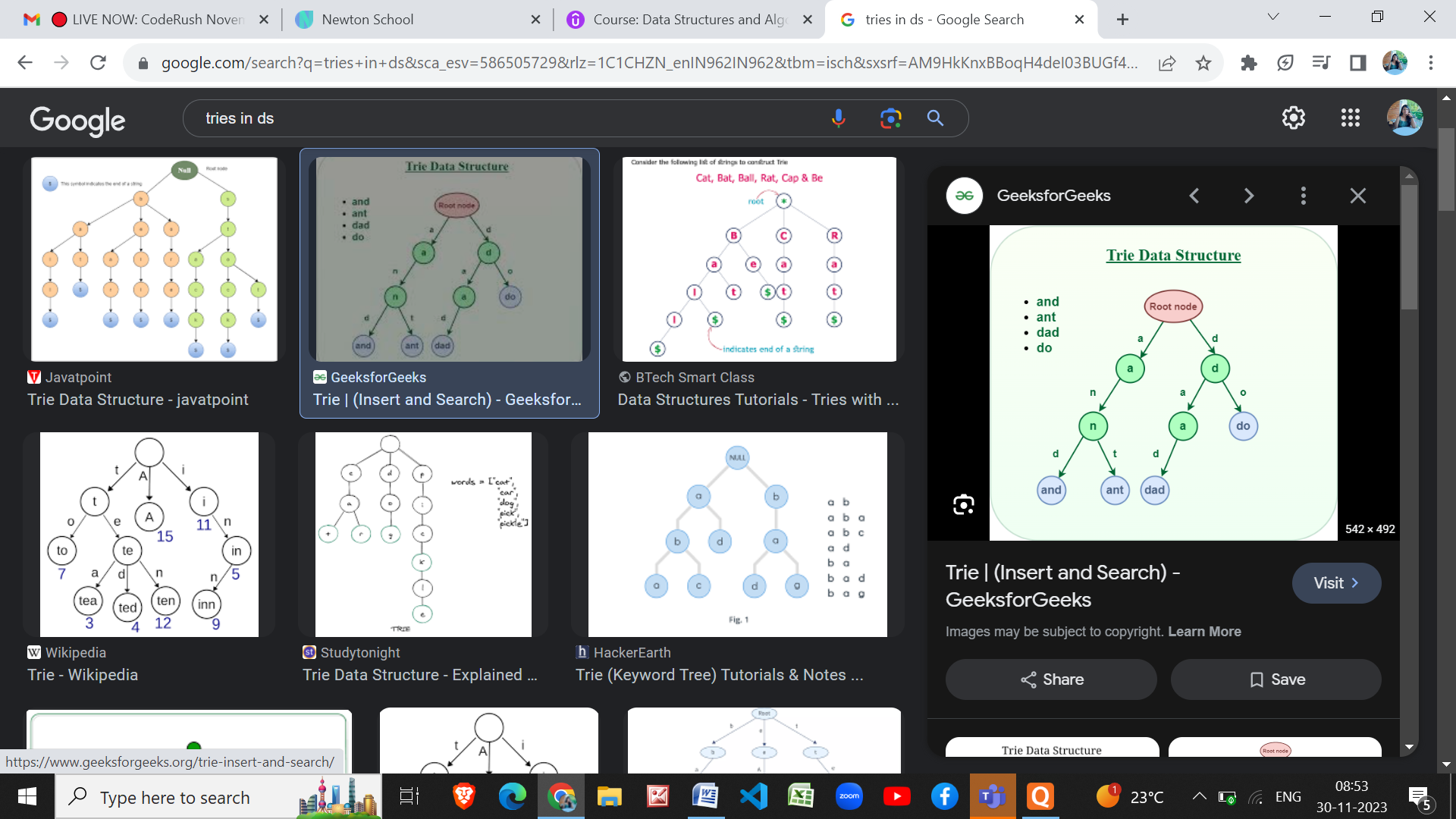
**Priority Queue** ­– Instead of using queue we can use heap. Since, insertion and deletion – O(log n)



**Priority 1**

Here insertion and deletion is O(log n), which is good. We can enter elemnet in the tree according to the priority and then utilize it.

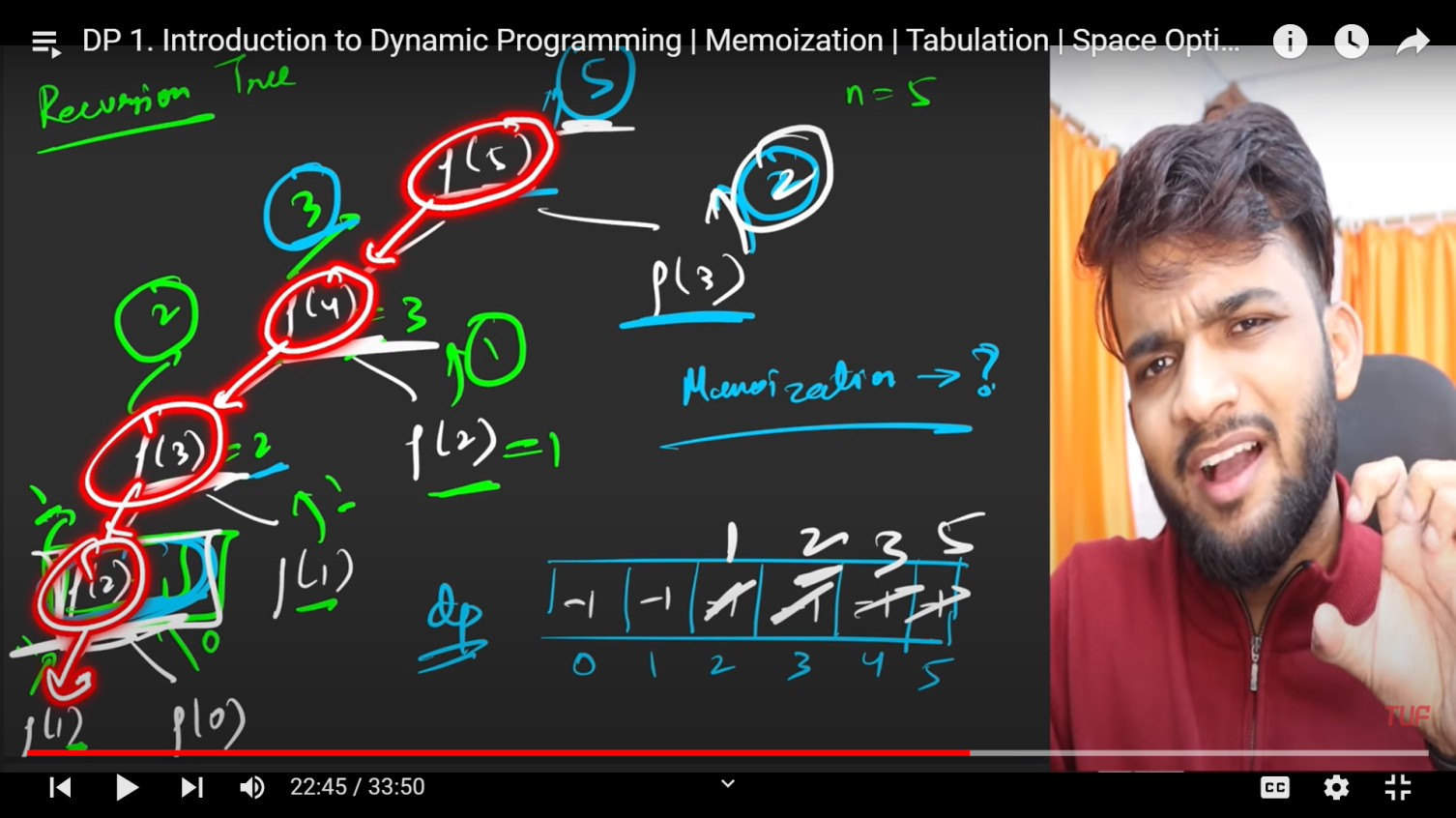
**Tries** - a special data structure used to store **strings** that can be visualized like a graph



Insertion, Deletion, Access – O(k) //*where k is thelength of the word.*

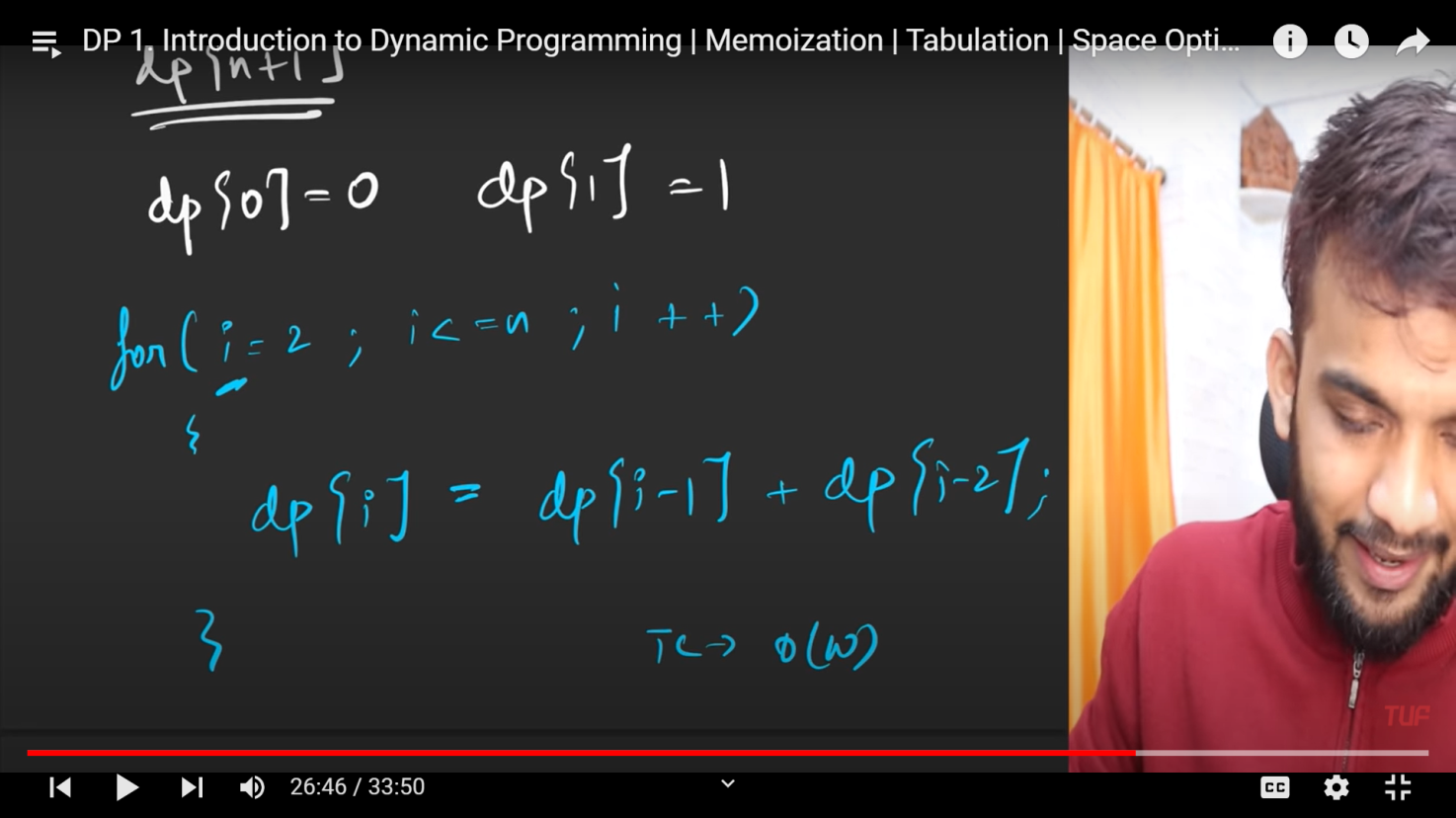
**Dynamic Programming**

1. Memoisation – Top-down approach, we store the value of already solved problem in a hashmap or arrya s.t. access time remains constant.
   1. Analyze how much maximum space it will take and make an array of that space
   2. Check if the value for a particular sub-problem is already saved
   3. If yes, simply return that value, else store its value in the array.



Here Time complexity=O(n) and space=O(n+n) //dp array and stack space.

1. Tabulation – bottom-up approach



Time Complexity – O(n)

Space Complexity – O(n)

1. Most optimal solution – The way I do it.