Geeks For Geeks DSA Course

Array

Theory recheck

- Second largest element - If new element is greater than larger element change larger element to new element and second larger to larger. If new element is lesser than larger and its greater than second larger then change second larger to new element.
- Maximum difference problem with order - Find the difference between current element with previous minimum element. Update minimum element with current element
- **Trapping rain water**
- **Maximum Sub array sum or Kaden's algorithmi**

- **Maximum circular sub-array sum** - find the normal sub array sum. Find the total sum and add negative of maximum sub array. Add sum with negative of max sum sub array. Return maximum of these two sums
- **Majority element or Moore's voting algorithm**
- **Minimum consecutive flips** - Always second group of elements we need to slip because the difference between group pf 1s and 0s are always either 0 or 1. If beginning and end are same then difference is 1 else it will be 0.

Problem Recheck

- **Maximum occurred integer** - Prefix sum technique
- **Frequencies of limited range array elements** - Rearrange array technique
- Equilibrium point & leaders in an

array

- **Smallest positive missing number**
- **Rearrange array alternately** - Rearrange technique
- Rearrange an array with O(1) space
- **Maximum index**
- **Check if array is sorted and rotated**

Techniques

- **Kadane's Algorithm:** to get the Max value of continuous sub array or Min value of continuous sub array
- **Moore's voting algorithm**: In first phase find the possible majority candidate and in the second phase verify the same
- **Sliding window:** To work on 'k' sized sub array of array at any time. Ex: maximum sum of k

consecutive elements, Sub array of size k with given sum, given array of non negative integers find if there is a Sub array with given sum, count distinct element in every window of size k. This size can grow or shrink in any way.

- **Prefix Sum:** To have a separate array which contains at position the value of total sum of current element plus total sum of previous all elements. Ex: finding if array of integer has equilibrium point.
- **Rearrange Array:** Store two different values at the same position of array using the formula: a[i] = a[i] + (new_value * array_length+1). Later get the old value as a[i]%(array_length+1). And get the new value as a[i]/(array_length+1)

==========

Matrix

- 2-D matrix or 3-D matrix

Theory Recheck

- Transpose of a matrix
- **Rotate matrix anti clockwise 90**
- Search in row wise column wise sorted matrix
- **Spiral traverse of matrix**
- **Median of a row wise sorted matrix**

Problem Recheck

- **Multiply the matrices**
- **Determinant of a matrix (not solved yet)**
- **Boolean matrix**
- **Make matrix beautiful**

==========

Recursion

- Function which has a base case and calls itself with different parameter
- Application which uses recursion are dynamic programming, backtracking, divide and concur, DFS tree etc
- Tail Recursion: A function is called tail recursion if parent function has nothing to do if the child functions are returned. And this kind of recursive functions are faster in performance, because caller no meed to keep any previous data. Last call happens must be recursive nothing should happens after that
- Tail call elimination: In tail recursion modern compilers will remove recursive method calls

and replace it with goto statement and label like n=n-1; goto start; start: This can eliminates multiple method calls on stack thus makes o(1) auxiliary space

Theory Recheck

- ≡ ○ Palindrome check using recursion
- ≡ ○ **Rope cutting problem**
- ≡ ○ **Generate subsets**
- ≡ ○ **Tower of Hanoi**
- ≡ ○ **Josephus problem**
- ≡ ○ **Subset sum problem**
- ≡ ○ **Printing all permutations**

Problem Recheck

- ≡ ○ **Lucky numbers**
- ≡ ○ **Possible words from phone digits**

==========

Searching

- Linear Search or Binary Search
- Binary Search involves finding midpoint and then searching for the element in either first half or second half

Technique

- Two pointer approach: we can use one pointer at start and another at end and move them towards mid depending on some condition. Ex: Given a **sorted** array and a sum, find if there is a pair with given sum. Note that if array is unsorted then hashing technique will be best one. Another example, given a sorted array and a Sub find if there is a triplet with given sum. If unsorted, then first sort it and use 2 pointer approach. If array is not allowed to sort then copy it to another array and continue.

- Slow and Fast pointer: This approach is mainly used in linked list to find whether it has a loop or not.

Theory Recheck

- Index of first\last occurrence sorted
- Count occurrences in sorted
- Count 1s in sorted array
- Square root
- **Search in infinite sized array**
- **Search in sorted rotated array**
- **Find a peak element**
- Repeating element part 1 & **2** (Repeating element can start from 0 or 1):

- O(n^2) & O(1) - 2 for loop
- O(nlogn) & O(1) - sort and search which changes original array
- O(n) & O(n) - extra array with boolean flag as false and change it to true or use hash map

- **Part 2:** O(n) & O(1) - By using slow and fast pointers similar to find loop in a linked list. Once slow and fast pointer meets each other then move both pointer at a speed of 1 and then first meeting point will be repeating element.

≡ ○ **Median of 2 sorted arrays**

≡ ○ **Allocate minimum pages**

Problem Recheck

≡ ○ **Count 1's in a binary array**

≡ ○ Floor in the sorted array

≡ ○ **Minimum number in a sorted roteted array**

≡ ○ **Two repeated elements**

≡ ○ **Maximum water between 2 buildings**

≡ ○ **Count only repeated**

==========

Hashing

- ≡ ○ Dictionary based and set based data structure
- ≡ ○ Hashing is a technique and HashTable is a data structure
- ≡ ○ Search, Insert, Update and Delete operations in O(n) time. This beats all other data structures.
- ≡ ○ Hashing is not used for Sorted data, Searching closet values, Prefix searching(all keys with same prefix)
- ≡ ○ For sorted data and searching closet values we use either AVL tree or Red Black tree instead of hashing
- ≡ ○ For prefix search we use an alternate dictionary data structure called Trie instead of hashing
- ≡ ○ Applications of hashing are like dictionaries, database indexing, cryptography, caches, getting data from databases etc

- Direct Address Table: we can use boolean array of size 1000 to store 1000 students. Then we can use keys as indexes. By default, we initialize all the values as 0. Then we can change value of index as 1 to insert the corresponding nth student. To delete, we simply change value to 0 at that index. For searching we simply return the value present in that index. Now, this technique will bot work for bigger data like phone number of employees. Hence we use Hashing Functions
- Hashing Function: a function which takes bigger input and convert them to a smaller key which can be used as an index of smaller array.This array is called HashTable.
- This Hash function should take O(1) for integers and

O(String.length) for strings

- ≡ ○ Hash Collision: When hash function gives the same key for different input. To overcome this we can use either Hash Chaining or Open Addressing technique
- ≡ ○ Hash Chaining through Hash Nodes to avoid collisions
- ≡ ○ Load factor of hash table is number of keys to be inserted in hash table divided by number of available slots. This load factor should be less always.
- ≡ ○ Data structure to storing chains: linked list(with O(l) for search insert & delete), Array List(with O(l) for search insert & delete), Self Balancing BST like AVL tree or Red Black Tree(with O(logl) for search insert & delete)
- ≡ ○ Map does not maintains the order. To maintains the order we can use **LinkedHashMap** in java

- HashMap or HashSet does not maintains sorting order. To need a sorting use **TreeMap**.
- Hashing technique is the best one to use for unsorted arrays.

Technique

- **Prefix sum with Hashing(As a frequency counter or index counter):** This technique can be used for various types of Sub array related problems. This use this technique to find total sum till a particular index of array and compare it with hash table to find if this value already exist in it. Ex: Sub array with zero sum, Sub array with given sum etc

Theory Recheck

- **Sub array with zero sum** - check for prefix sum itself zero
- **Sub array with given sum:** Note - If array contains only positive integers then we can use sliding

window technique without need for additional space for hashing.

- ≡ ○ **Longest Sub array with given sum**
- ≡ ○ **Longest sub array with equal number of 0s and 1s**
- ≡ ○ **Longest common span with sum in binary array**
- ≡ ○ **Longest consecutive subsequence**
- ≡ ○ **Count distinct element in every window**
- ≡ ○ **More than n/k occurrences (Part 2)**

Problem Recheck

- ≡ ○ **Winner of an election**
- ≡ ○ **Sub array range with given sum**
- ≡ ○ **Positive negative pair**
- ≡ ○ **Zero sum Sub array**
- ≡ ○ **Sub arrays with equal 1s and 0s**
- ≡ ○ **Sort an array according to other**
- ≡ ○ **Sorting elements of an array by**

freequency

==========

String

- Small set of characters
- String can be created in java through Character array, String by literal, String by new keyword, StringBuffer and StringBuilder

Theory Recheck

- Check if a string is subsequence of other
- Check for anagram
- **Leftmost repeating character**
- **Leftmost non repeating character** (Refer the solution in the problem section)
- Reverse words in a string
- **Naive pattern searching**
- **Rabin Karp algorithm**

- **KMP algorithm part 1 and 2**
- **Check if strings are rotations**
- **Anagram search**
- **Lexicographic rank of a string**
- **Longest substring with distinct character**

Problem Recheck

- Binary string
- **Implement strstr**
- Check if string is rotated by 2 places
- **Isomorphic strings**
- **Keypad typing**
- **Non repeating character**
- **Maximum occurring character**
- **Remove common character and concatenate**
- **Sum of numbers in string**
- Minimum indexed character
- ***Smallest window in a string containing all character of***

another string

- **The modified string**
- **Case specific sorting of string**
- **Check if a string is subsequence of other**

==========

Linked List

- Benefit over Array or Array List is, no need to have a continuous memory locations so that insert or delete can be done easily at any position also no need to preallocate any memory/space.
- Singly linked list: First node is called as Head and last node points to 'null'. Each node refer to next node.
- Doubly linked list: Same as Singly linked list. But each node refers to its previous node also. And

first node's previous refers to null

- Advantages of doubly linked list: Can be traversed in both directions, delete a node with O(1) time with given reference to it, insert/delete before a given node in O(1), insert/delete at both ends O(1) by maintaining tail pointer. Disadvantages are extra space and complexity.
- Singly Circular Linked List: last node of singly linked list refers to head node of singly linked list.
- Advantages of Singly CLL: We can traverse the whole list from any node, implementation of algo like round robin, we can insert at beginning or end by just maintaining one tail reference.
- Doubly Circular Linked List: Previous reference of head points to last node and next reference of last node points to head. We can

get all the advantages of circular linked list and doubly linked list.

Theory Recheck

- Traversing a singly linked list in java iteratively or recursively
- Insert at the beginning of singly linked list
- Insert at the end of singly linked list
- Delete first node of SLL
- **Delete last node of SLL**
- **Insert at given position in SLL**
- **Search in SLL**
- **Insert at begin of DLL**
- Insert at end of DLL
- ***Reverse a DLL***
- Delete head of DLL
- Delete last node of DLL
- **CLL traversal in java**
- **Insert at beginning of CLL**
- **Insert at the end of CLL**

- [ ] **Delete head of CLL**
- [ ] **Delete kth of CLL**
- [ ] **Doubly CLL -** Insert at the begin and end
- [ ] **Sorted insert in a SLL**
- [ ] **Middle of SLL**
- [ ] **Nth node from end of SLL**
- [ ] **Reverse a SLL iteratively**
- [ ] ***Remove a duplicate from sorted SLL***
- [ ] **Reverse a SLL in groups of size k**
- [ ] **Detect loop using floyd cycle detection**
- [ ] **Detect and remove loop in SLL**(check for condition if loop exist at the first node) **Find length of a loop, Find the first node of a loop**
- [ ] ***Delete node with only pointer given to it***
- [ ] ***Segregate even and odd list***
- [ ] ***Clone a linked list using a random***

- ***pointer***
- ***LRU cache(Not done)***
- ***Merge two sorted linked list***
- ***Palindrome linked list***

Problem Recheck

- **Remove duplicates from an unsorted linked list**
- ***Swap kth node from ends(tough)***
- **Rotate a linked list**
- ***Add two numbers represented by linked lists***
- **Pairwise swap of nodes in LL**
- ***Given a linked list of 0s, 1s and 2s, sort it***
- ***Merge k sorted linked list***
- ***Intersection point in Y shaped linked list(check for object equality also)***
- ***Merge sort for linked list(Not done)***
- ***Merge sort on doubly linked list(Not done)***

==========

Stack

- Java provides two variations of stack. One is **Stack** class which extends from Collection->List->Vector. And another one is **ArrayDeque** which extends from Collection->Queue->Deque
- Its better to use ArrayDeque over the Stack, since stack extends vector which involves extra complexity to provide thread safety feature.
- Applications of stack: Function calls, checking for balanced parenthesis, reversing items, undo/redo, forward/backward etc

Theory Recheck

- **Array/Array List implementation**

- of stack in java
- Linked list implementation of stack in java
- Balanced parenthesis
- Two stacks in an array

Stock span related problem start

- *Stock span problem*
- Previous greater element
- Next greater element
- *Largest rectangular area in a histogram (Part 2) (By using previous minimum element)*
- *Largest rectangle with all 1's*

Stock span related problem end

- Stock with getMin() in O(1)
- Design a stack with getMin() in O(1) Space

Infix, prefix, postfix problems start

- *Infix to prefix(Efficient solution)*
- *Evaluation of prefix*

Problem Recheck

- Removing consecutive duplicates 1 and 2
- **Get min at pop**
- **Delete middle element of a stack**
- ***Infix to postfix***
- **Evaluation of postfix expression**
- ***The celebrity problem***
- ***Maximum of minimum for every window(Not done)***

==========

Queue

- FIFO. Data are inserted from Rear end through enqueue operation and data are removed from Front end through dequeue operation
- Applications of queue: single resource and multiple consumer (ex: Ticketing system),

synchronization between slow and fast devices, used in variations like Deque, Priority Queue, doubly ended priority queue

- Java provides two ways to use Queue. One is to use Linked list another one is to use ArrayDeque. Queue<Integer> q = new LinkedList<>(); **or** Queue<Integer> q = new ArrayDeque<>();
- Operations: offer, poll, peek, getFront, getRear, isFull, isEmpty, size etc

Theory Recheck

- **Implementation of queue using array/circular array**
- **Implementation of queue using linked list**
- ***Implement stack using one queue***
- **Reversing a queue**
- **Generates numbers with given**

digits

Problem Recheck

- ***Queue using 2 stack***
- **Stack using 2 queues**
- **Generate binary numbers**
- **Reverse first l elements of queue**
- ***Circular tour***

==========

Deque

- Can insert or delete data from both the ends.
- Operations: InsertFront,InsertRear, DeleteFront, DeleteRear,getFront, getRear, isFull, isEmpty, size etc
- Deque can be implemented by using doubly linked list or array data structure
- A Deque can be used as both

queue and stack

- ≡ ○ Deque methods in java: offerFirst, OfferLast, pollFirst, pollLast, peekFirst, peekLast
- ≡ ○ Deque<Integer> d = new LinkedList<>();
- ≡ ○ Use deque.iterator or deque.descendingIterator or for each loop to traverse
- ≡ ○ ArrayDeque has all the methods provided by Stack, Queue and Deque.

Theory Recheck

- ≡ ○ Array implementation of deque(Not done)
- ≡ ○ Design a data structure with min and max operations
- ≡ ○ ***Maximums of all subarrays of size k***

Problem Recheck

- ≡ ○ **Rotate deque by k**
- ≡ ○ **Deque deletion**

==========

Tree

- Consider a problem where we need to represent hierarchy for example organization structure. Need to represent organization structure using data structure in memory. Another example is folder structure.
- All the data structure which we have seen till now are linear in nature, which stores the data in sequential fashion. Tree is the non linear data structure, used for these kind of requirements, which stores the data in hierarchical fashion.
- All the items in tree are called nodes. At the top of the hierarchy we have **Root Node**. Nodes which

are at the bottom and does not have any further nodes below it are called **Leaf Node**. Nodes which are just below a node are called **Children of this Node**. Node which are just above a node are called **Parent of this Node.**

- Tree data structure is recursive in nature. So tree itself contains many trees in it which are called sub-tree.
- **Descendants** of a node are, all the nodes which lie in the sub-tree with this node as root.
- **Degree** of a node is the number pf children it has. All the leaf nodes have 0 degree.
- Application of Tree: Organization structure, folder structure, XML/ HTML content, Inheritance in OOP, Binary Search Tree, Binary Heap, Trie etc.
- **Binary Tree:** In binary tree every

node has at most 2(can be 0, 1 or 2) children or we can say degree of every node is at most 2. In binary tree every node has 3 fields. The middle one contains the data and other two are the reference to 2 left & right children. Most of the practically used and popular tree data structure are the variations of Binary Tree.

- **Tree Traversal:** 2 ways for tree traversal. Breadth First & Depth First. Many operations on tree are based on this tree traversal logic. Depth First has 3 popular variations InOrder, PreOrder and PostOrder.
- For Breadth First traversal we can use Queue data structure and for Depth First traversal we use recursion.
- Breadth First - time & space complexity o(n). Depth First -

time complexity o(n), space complexity o(height)

Theory Recheck

- Implementation of in, pre & post order traversal.
- Height of a binary tree
- Print nodes at k distance
- Level order traversal line by line part 1.
- Size of a binary tree
- **Maximum in binary tree**
- **Print left view of binary tree**
- **Children sum property**
- **Check for balanced binary tree**
- **Maximum width of binary tree**
- **Convert binary tree to doubly linked list**
- **Construct binary tree from inorder amd preorder**
- **Diameter of a binary tree**
- **LCA of binary tree part 2**

- ≡ ○ **Burn a binary tree from a leaf**
- ≡ ○ **Count nodes in a complete binary tree**
- ≡ ○ **Serialize and Deserialize a binary tree**

Problem Recheck

- ≡ ○ Preorder traversal
- ≡ ○ **Determine if two trees are identical**
- ≡ ○ Children sum parent
- ≡ ○ Level order traversal line by line
- ≡ ○ Level order traversal in spiral form
- ≡ ○ Maximum width of Tree
- ≡ ○ **Check for balanced tree**
- ≡ ○ **Left view pf binary tree**
- ≡ ○ **Right view of binary tree**
- ≡ ○ **Diameter of binary tree**
- ≡ ○ **Mirror tree**
- ≡ ○ Height of a binary tree
- ≡ ○ **Lowest common ancestor in a**

- **Binary tree**
- ***Make binary tree from linked list***
- ***Binary tree to DLL***
- ***Connect nodes at same level***
- ***Construct binary tree from parent array***
- **Tree from postorder and inorder**
- ***Maximum path sum from any node***
- ***Maximum difference between node and its ancestor***
- **Foldable binary tree**
- **Count number of subtrees having given sum**
- ***Node at distance***
- **Serialize and de-serialize a binary tree**
- ***Check if subtree(Not done)***

==========

Binary Search Tree (BST)

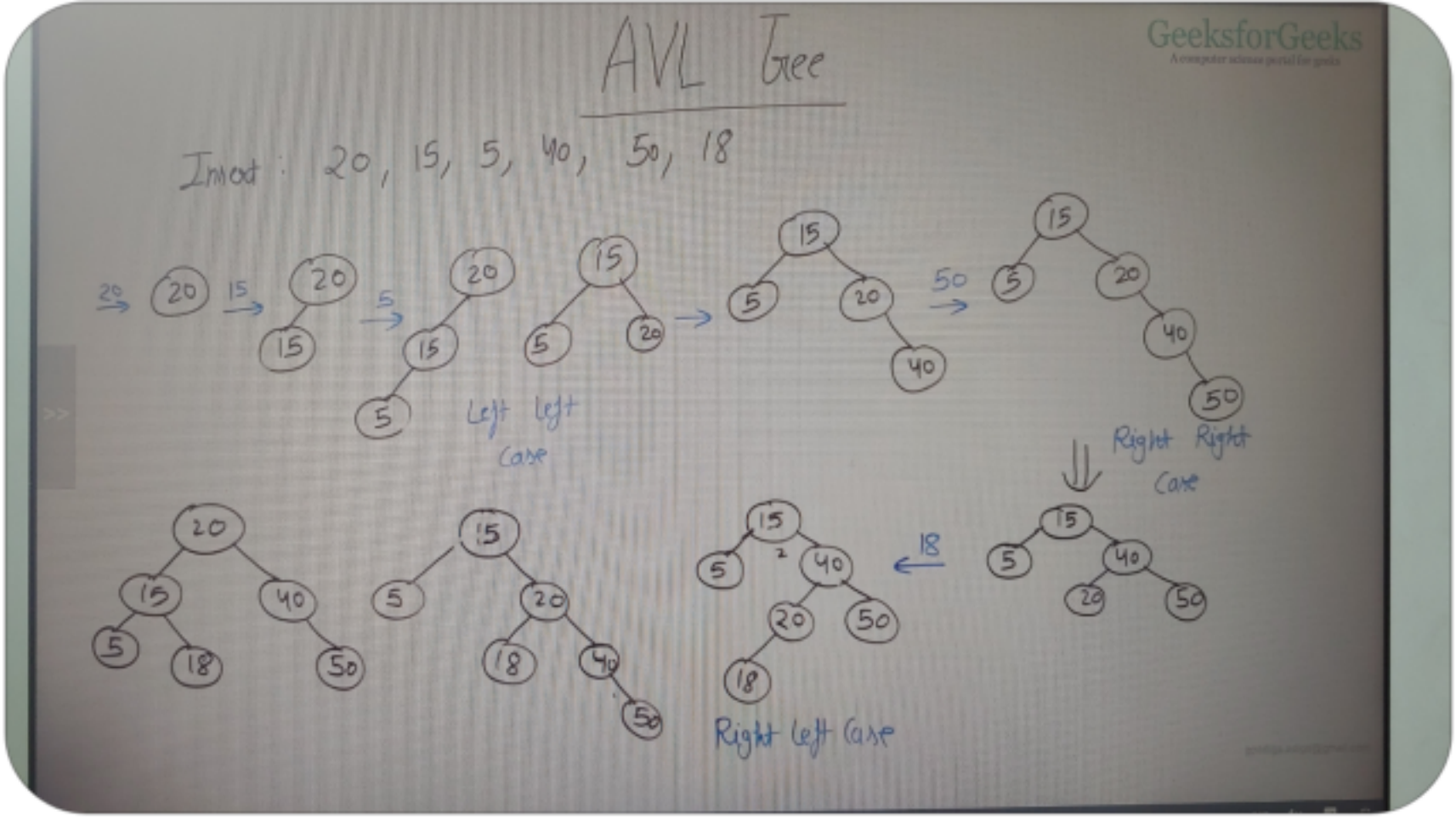
- All the nodes to the left of a node will have smaller values and all the nodes to the right of a node will have higher values. All values are typically considered as distinct.
- The time complexity of access, insert, delete, search and closest value(floor/ceil) find will have O(logn) if the BST is the self balanced one. Else in case of normal BST it will have time complexity of O(height).
- In java BST is implemented through TreeMap and TreeSet which are nothing but self balancing Binary Search Tree.
- By default TreeSet provides methods like floor, ceiling, higher, lower etc. Also when we travel these, we will get items in sorted order. Time complexity of all

these operations are O(logn). Time complexity of size() & empty() is O(1).

- By default TreeMap provides methods like floorKey, floorEntry, ceilingKey, ceilingEntry, higherKey, higherEntry, lowerKey, lowerEntry etc.
- **Self Balancing BST:** To keep the height of BST as O(logn) always. So that all the operations on this BST will have the time complexity as O(logn). Same set of keys in BST can generate different height depending on the order how we insert the keys in to BST. So if we know the keys in advance we can make a perfectly balanced BST, by making the mid element as the root of BST and other elements to its left & right.
- **BST Rotation**: The idea to keep the self balanced BST to remain

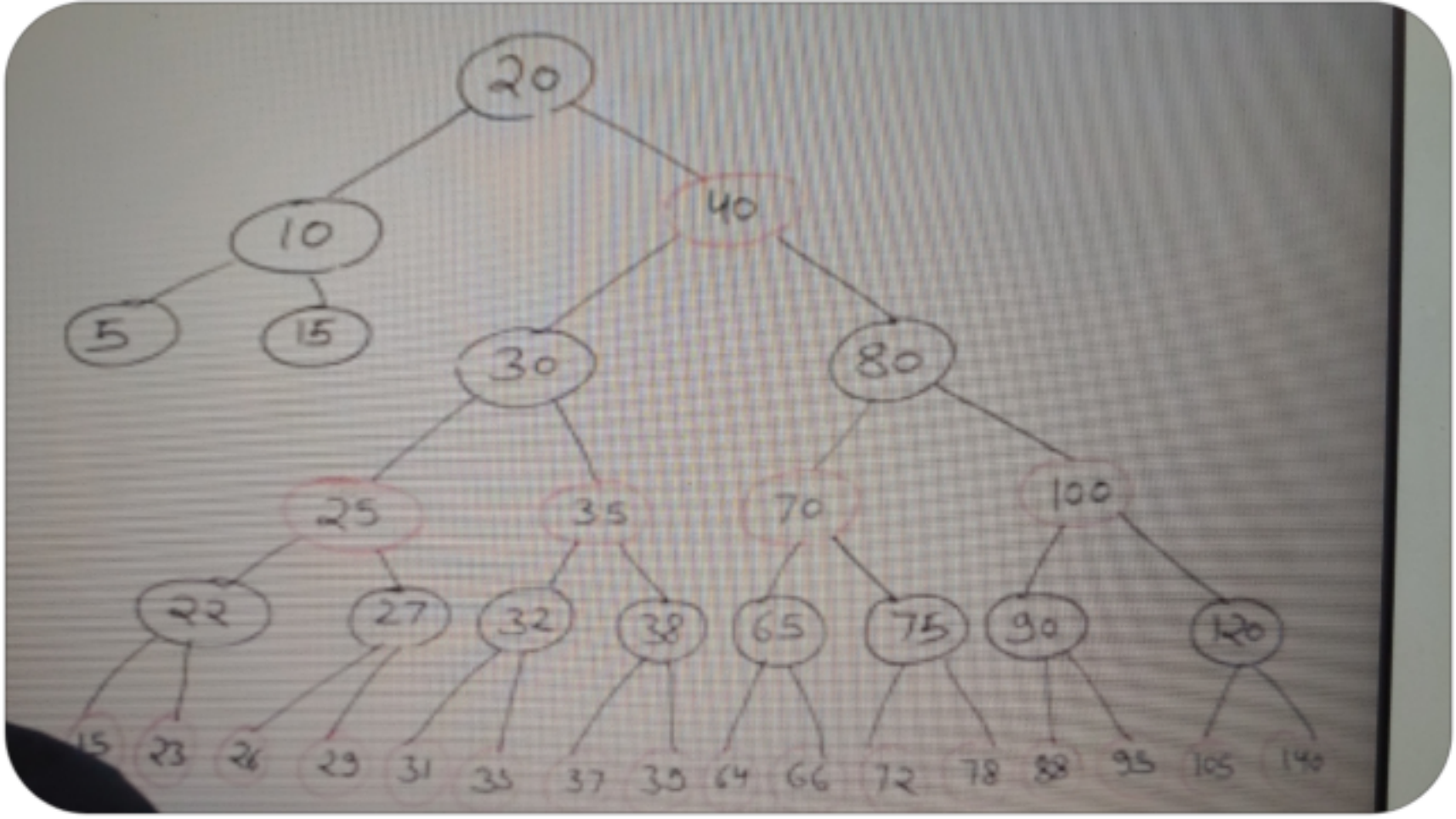
in balanced state is, do some restructuring or re-balancing when doing insertion or deletion operations on BST. This restructuring is called **rotation**, either left rotation or right rotation.

- **Self Balancing BST example:** AVL tree and Red Black tree are the 2 example of self balancing BST.
- **AVL Tree:** It is self balanced BST which maintains the height as O(logn) always. For every node the difference between left and right heights does not exceed one(it is very strict about this). This difference we call **Balance Factor**. Balance factor = |lh-rh| <= 1.
- AVL tree does single or double rotations in order to restructure itself in insert and delete operations.



- **Red Black Tree:** It is also a self balanced BST which maintains the height as O(logn) always. It is not that strict as AVL tree so that number of rotations will be less. It will allow to have difference heights between right and left of a node as maximum of two, instead of 1 in case of AVL tree.
- AVL is best when we have less insertions and deletions compared to many many search operations. But if we have mix of search, insertion and deletion than we use red black tree.

- ○ It allows twice the number of nodes on one path compared to other path from a node to its descendent leafs.
- ○ Every node is either red or black. Root is always black. No 2 consecutive reds. Number of black nodes from every node to all of its descendent leaves should be same.
- ○ Number of nodes on the path from a node to its farthest descendent leaf should not be more than twice than the number of nodes on the path to its closest descendent leaf.
- ○ TreeMap and TreeSet in java uses Red Black Tree for their implementation.



- **Applications of Self Balanced BST:** To maintain sorted stream of data. To implement doubly ended priority queue. To solve problems like count smaller/ greater in a stream, floor/ceiling/ higher/lower in a stream.
- Augmented BST: We can add some additional data to each BST node like left count value to solve some of the problem like finding Kth smallest in BST.

Theory Recheck

- Search in BST Java (recursive & iterative)

- Insert in BST Java (recursive & iterative)
- **BST deletion in Java**
- **Floor in BST Java**
- **Ceil in BST**
- Ceiling on left side in an array
- ***Find Kth smallest in BST***
- ***Check for BST (has 4 solutions)***
- ***Fix BST with two nodes swapped***
- **Pair sum with given BST**
- **Vertical sum in a Binary Tree**
- **Top view of binary tree**
- **Bottom view of binary tree**

Problem Recheck

- Search a node in BST
- Minimum element in BST
- Insert a node in a BST
- **Find common Nodes in two BSTs**
- **Lowest common ancestor in BST**
- Print BST elements in given range
- Check for BST

- ***Delete a node from BST***
- ***Pair sum in BST***
- **Floor in BST**
- **Ceil in BST**
- **Vertical traversal of binary tree**
- **Top view of binary tree**
- **Bottom view of binary tree**
- Find the closest elements in BST
- Count BST nodes that lie in a given range
- ***Preorder to PostOrder***
- ***Fixing two nodes of a BST***
- ***Merge two BSTs***

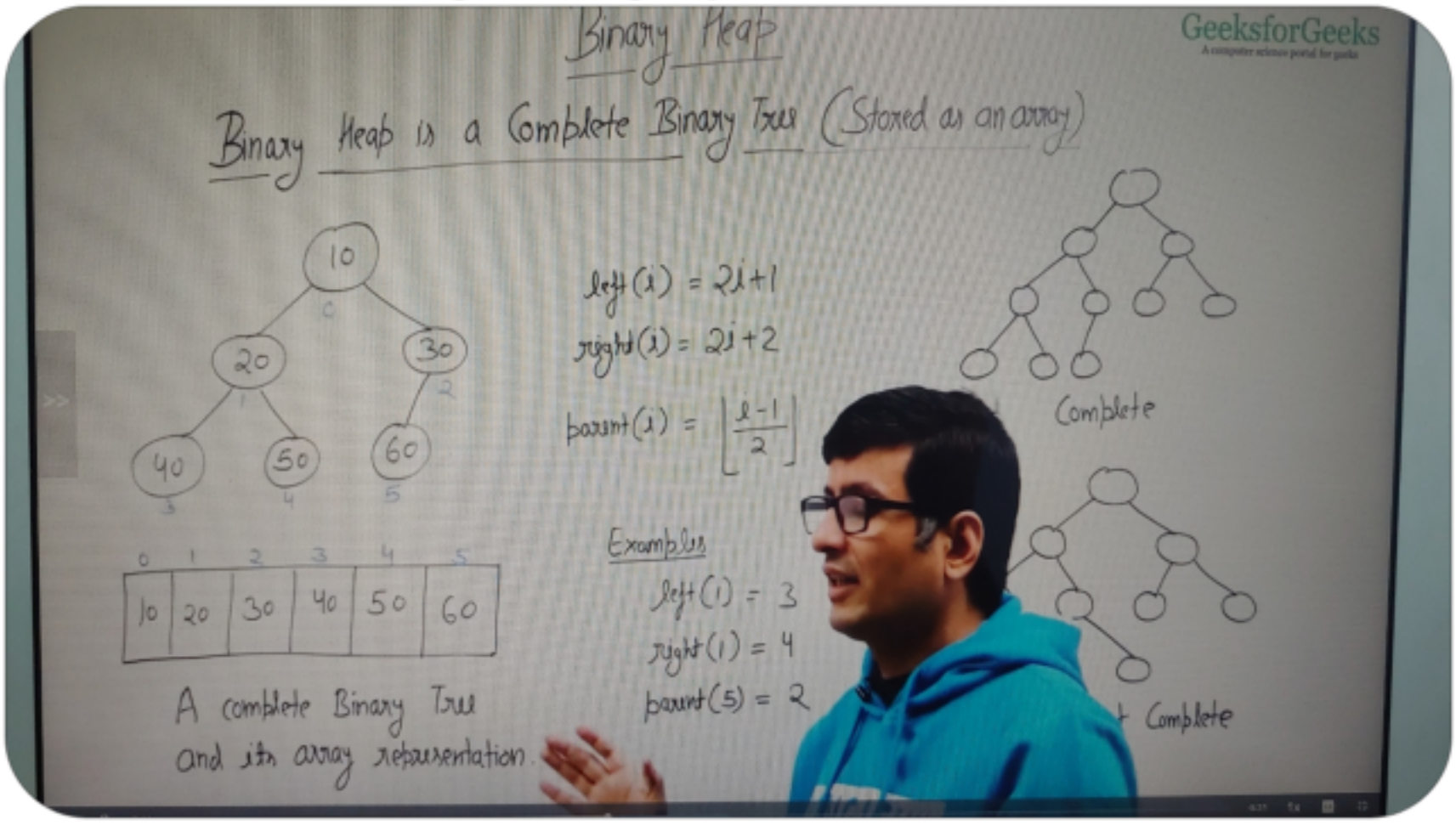
==========

Binary Heap

- Used in Heap Sort Algorithm and to implement Priority Queue. 2 types Min and Max Heap. Min

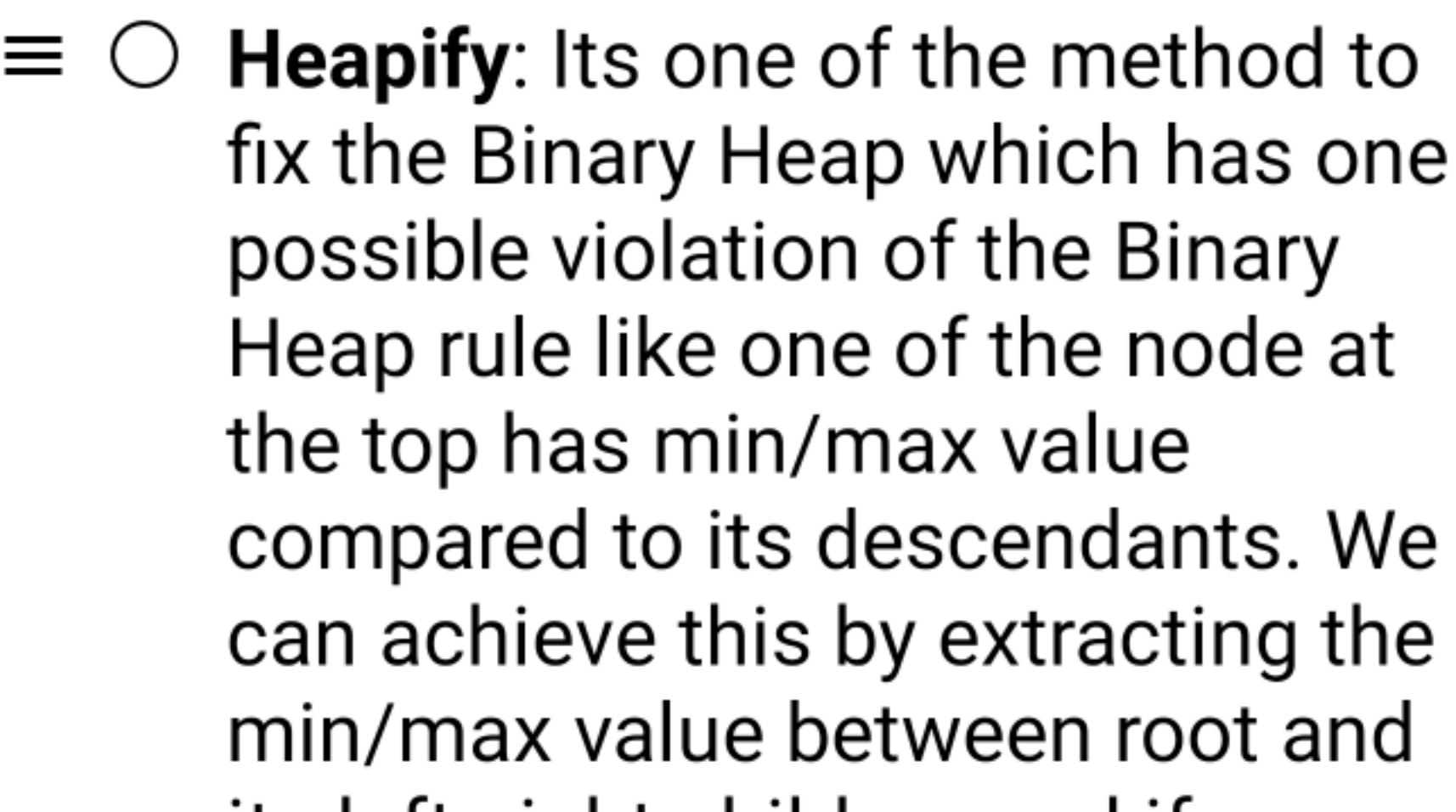
heap stores data in ascending order and max heap stores the data in descending order.

- ○ Binary Heap is a complete binary tree (they are typically stored as an array), which means all the nodes should be occupied without leaving any nodes in middle except the last nodes. All the nodes should be filled from top to bottom, left to right without leaving an gap.



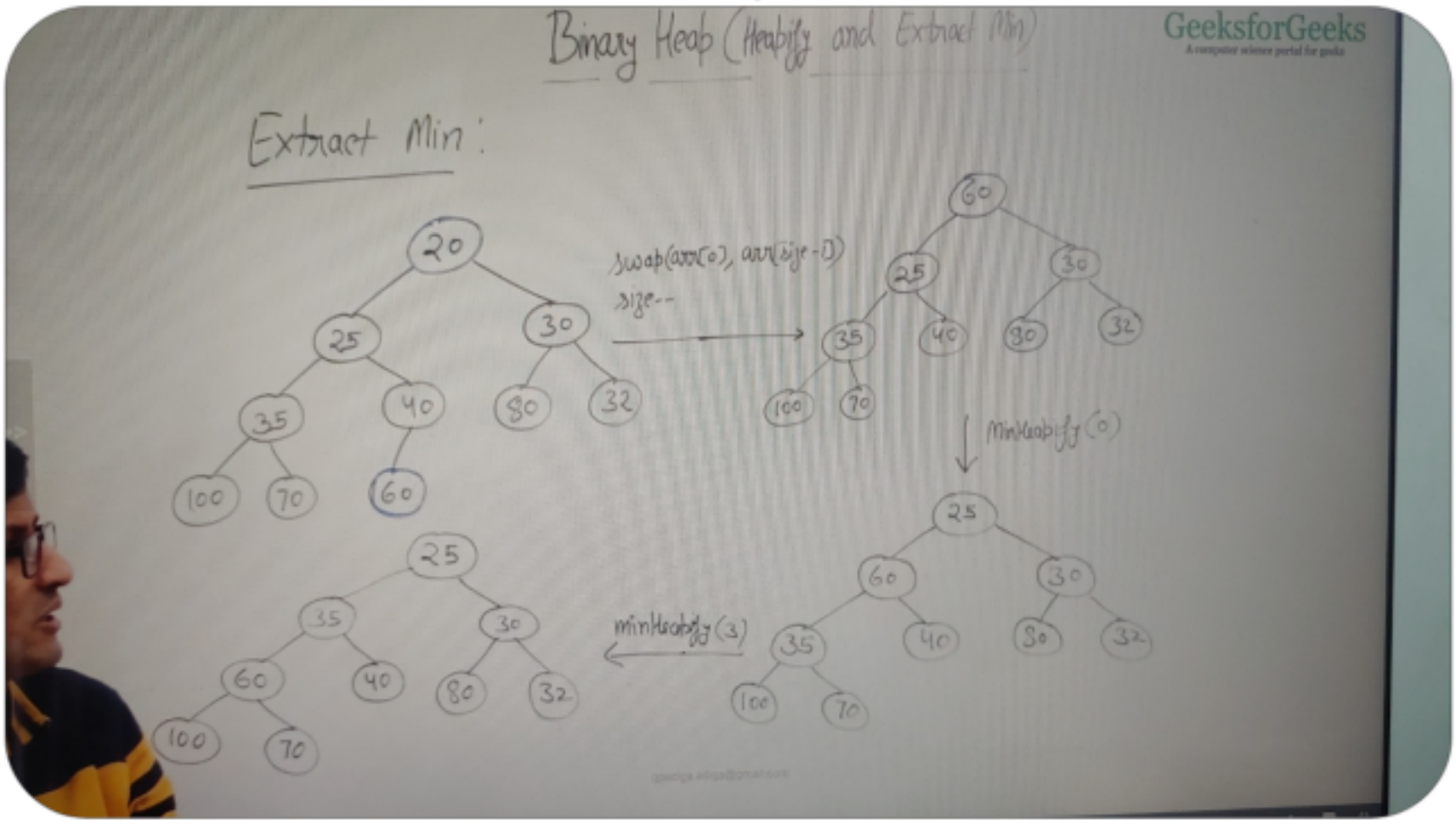
- ○ Min Heap is a complete binary tree where every node has value smaller than its descendants.

- **Heapify**: Its one of the method to fix the Binary Heap which has one possible violation of the Binary Heap rule like one of the node at the top has min/max value compared to its descendants. We can achieve this by extracting the min/max value between root and its left, right children and if we find any violation we swap the root with either left or right node.



- **Extract Min**: This is the method to remove the minimum value from Min Heap and return the same. Also, we will restructure the

Binary Heap to follow its rule. We do this by swapping the root element with last element and reducing the size by 1. And then call minHeapify from root.



- **Priority Queue:** Priority Queue in java by default implements min heap data structure.

Theory Recheck

- Binary Heap implementation
- **Binary Heap insert**
- ***Binary Heap(Heapify and Extract)***
- ***Binary Heap(Decrease key, Delete and build heap)***

- ***Heap sort***
- **Sort K-Sorted Array**
- **Buy maximum items with given sum**
- **K largest elements**
- **K closest element**
- ***Merge K Sorted Arrays***
- **Median of a stream**

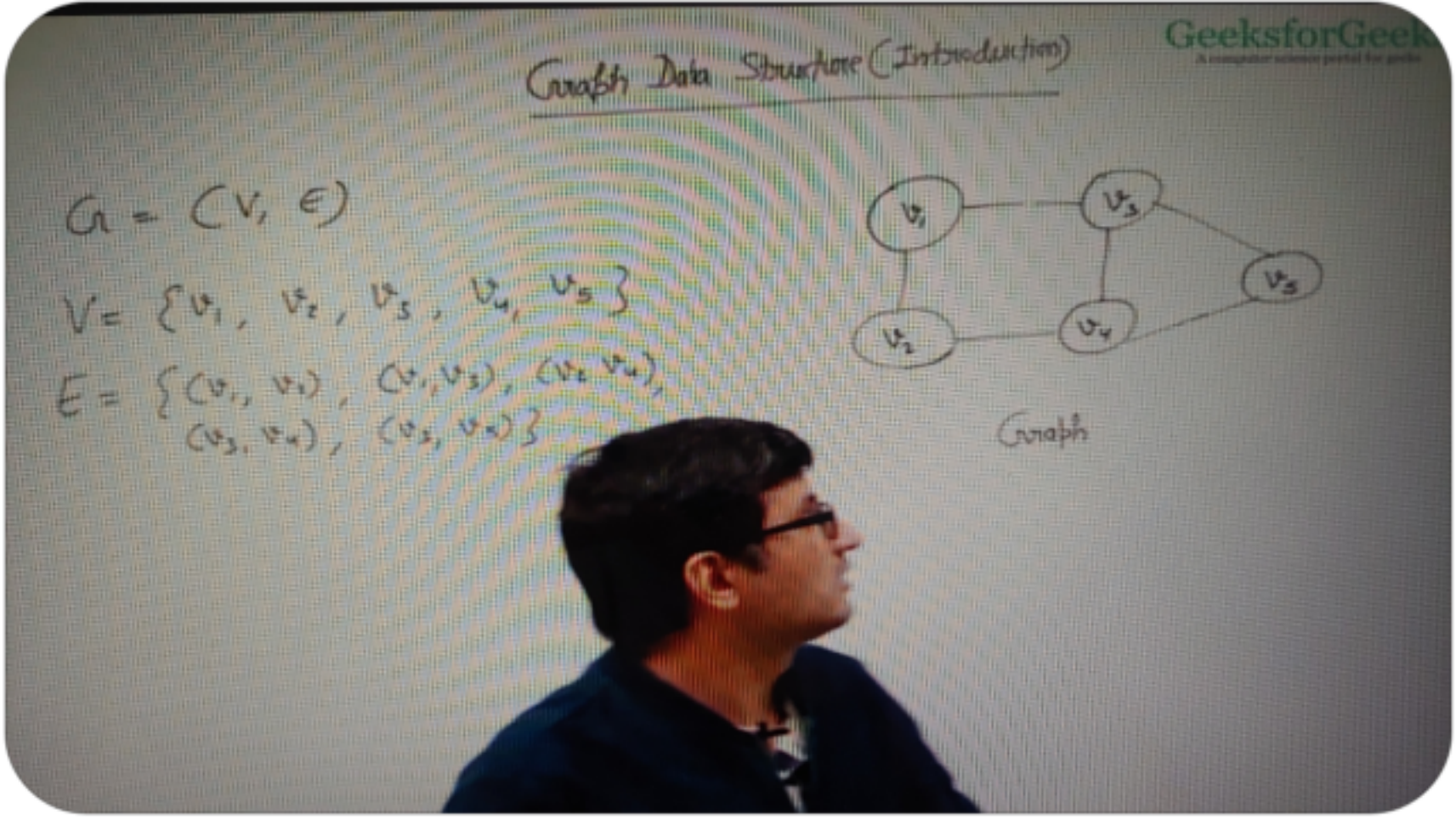
Problem Recheck

- ***Binary Heap Operations***
- **K largest elements**
- **Minimum cost of ropes**
- Kth largest element
- Kth smallest element
- K most occurring element
- **Kth largest element in a stream**
- ***Merge K Sorted Arrays***
- ***Nearly sorted***
- ***Rearrange Characters***
- ***Find median in a stream***
- ***Heap sort***

==========

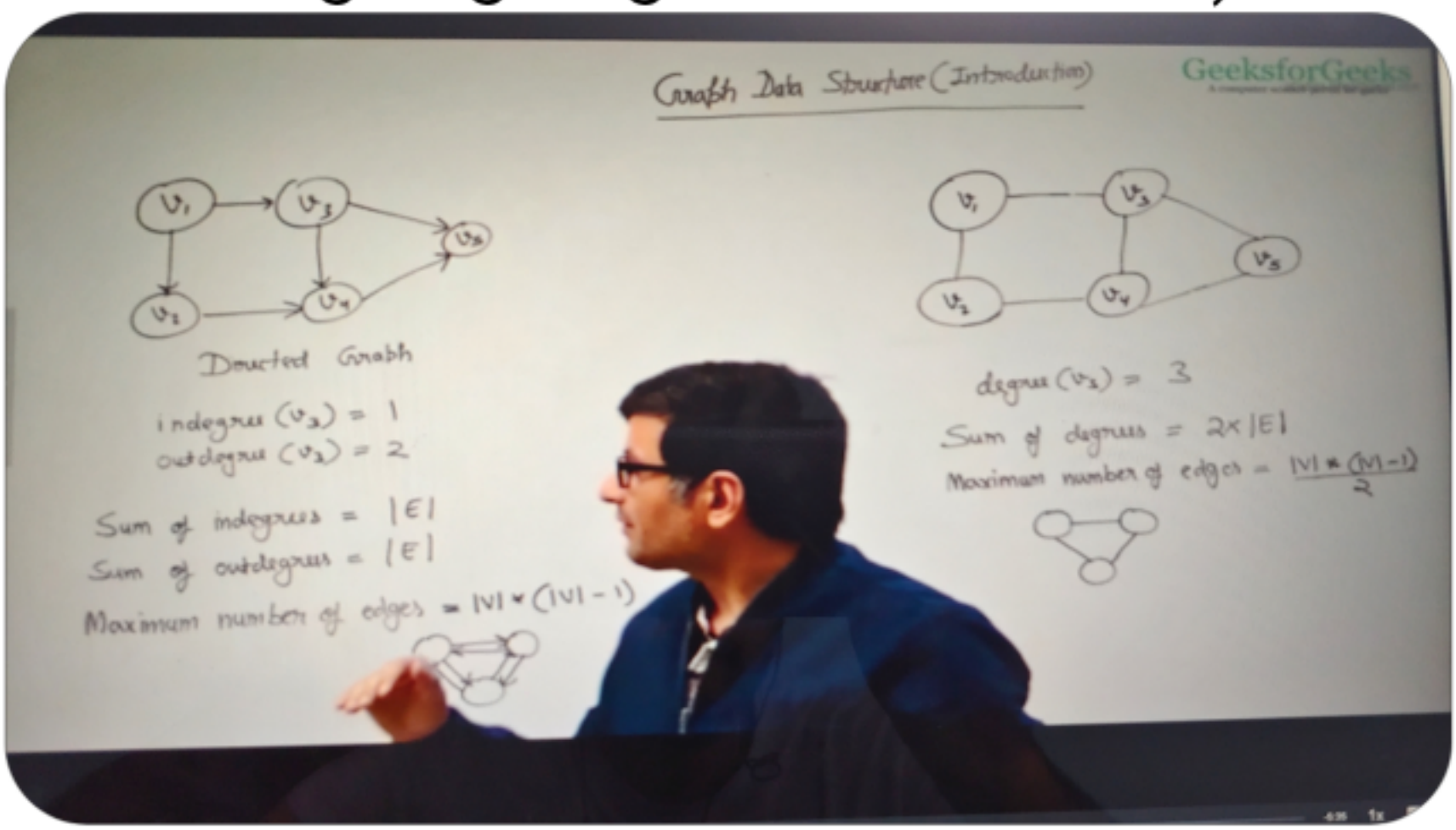
Graph

- Tree data structure is mainly used to represent hierarchy and it uses parent child relationship to represent hierarchy. We can not use tree when we have random connections among nodes. For this purpose we use Graphs.
- Graph data structure allows connections between any node to any other nodes. Its mainly used to represent relationship between multiple nodes.
- It is represented by a pair of sets. First set in this pair is set of vertices and the other set in this pair is set of edges.

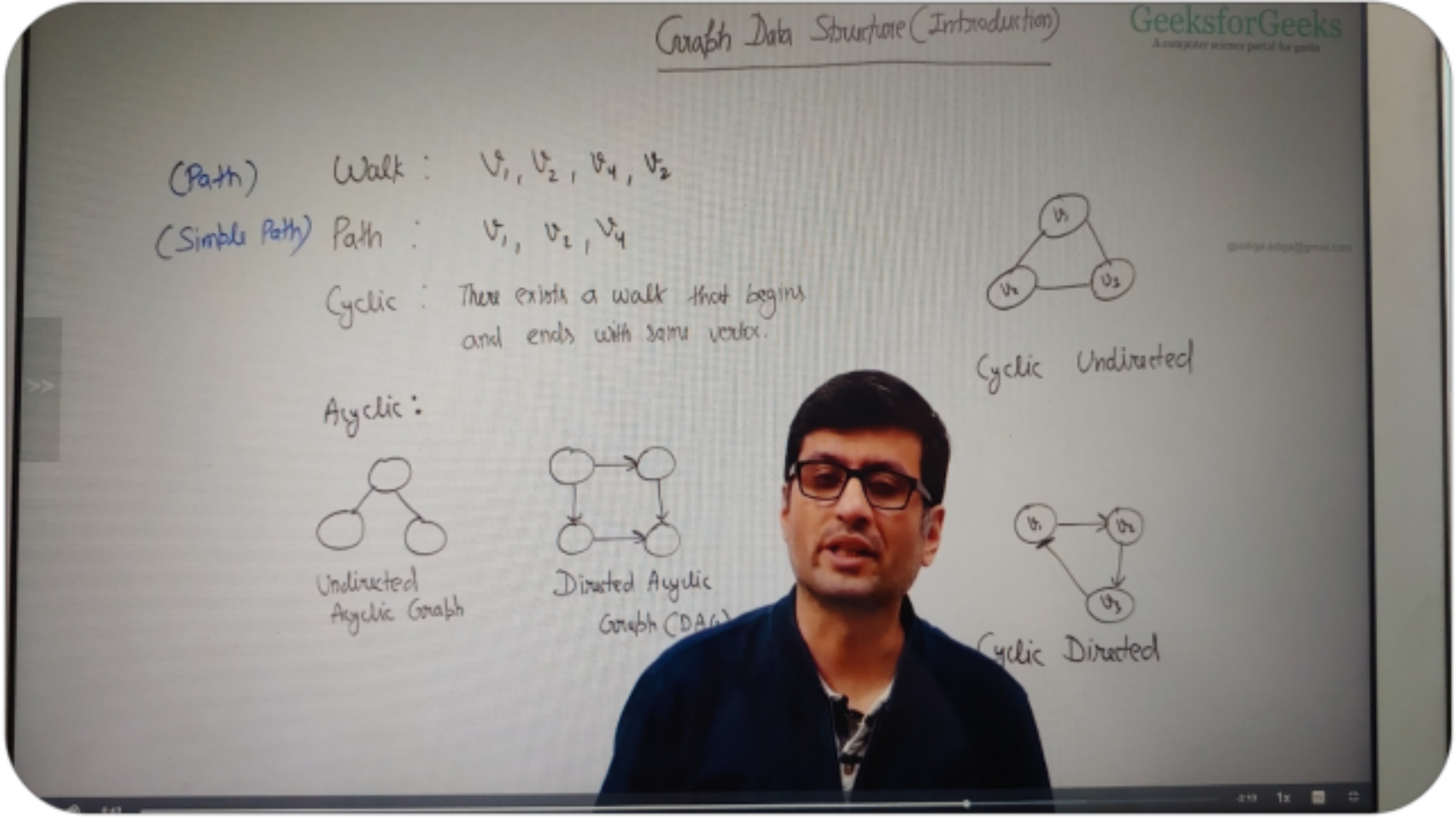


- **Directed vs Undirected Graph**: In a directed graph edges will have a direction (we can move only one direction), where as in undirected graph edges will not have any direction (we can move in both the directions).
- Example of Undirected graph is social network where a person is friend of another we can say another friend is also a friend of first person. Example of directed graph is world wide web, where each page has a link to another pages.

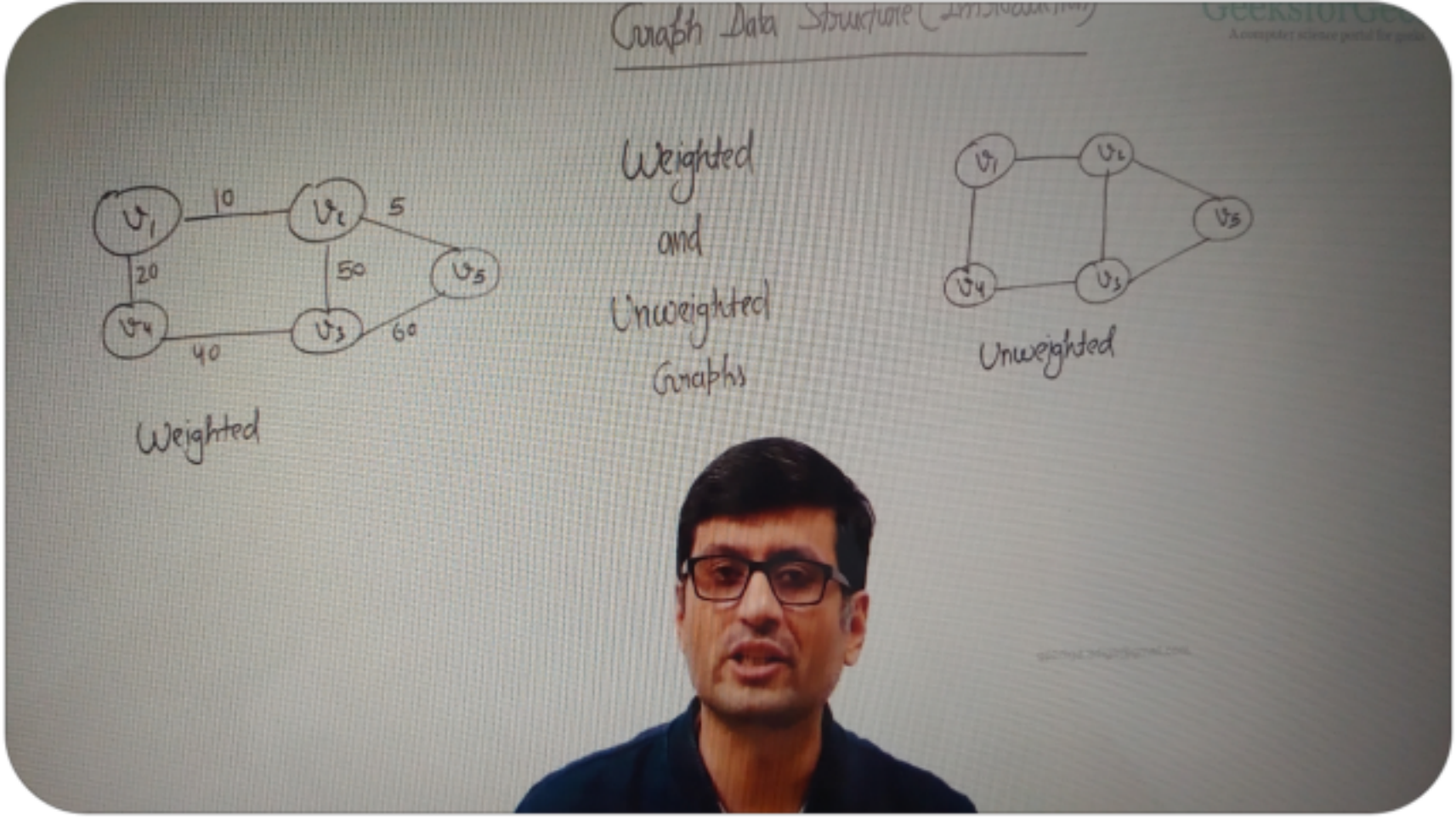
- Degree(number of edges going through a vertex), In Degree (number of edges coming to a vertex) & Out Degree (number of edges going out of a vertex).



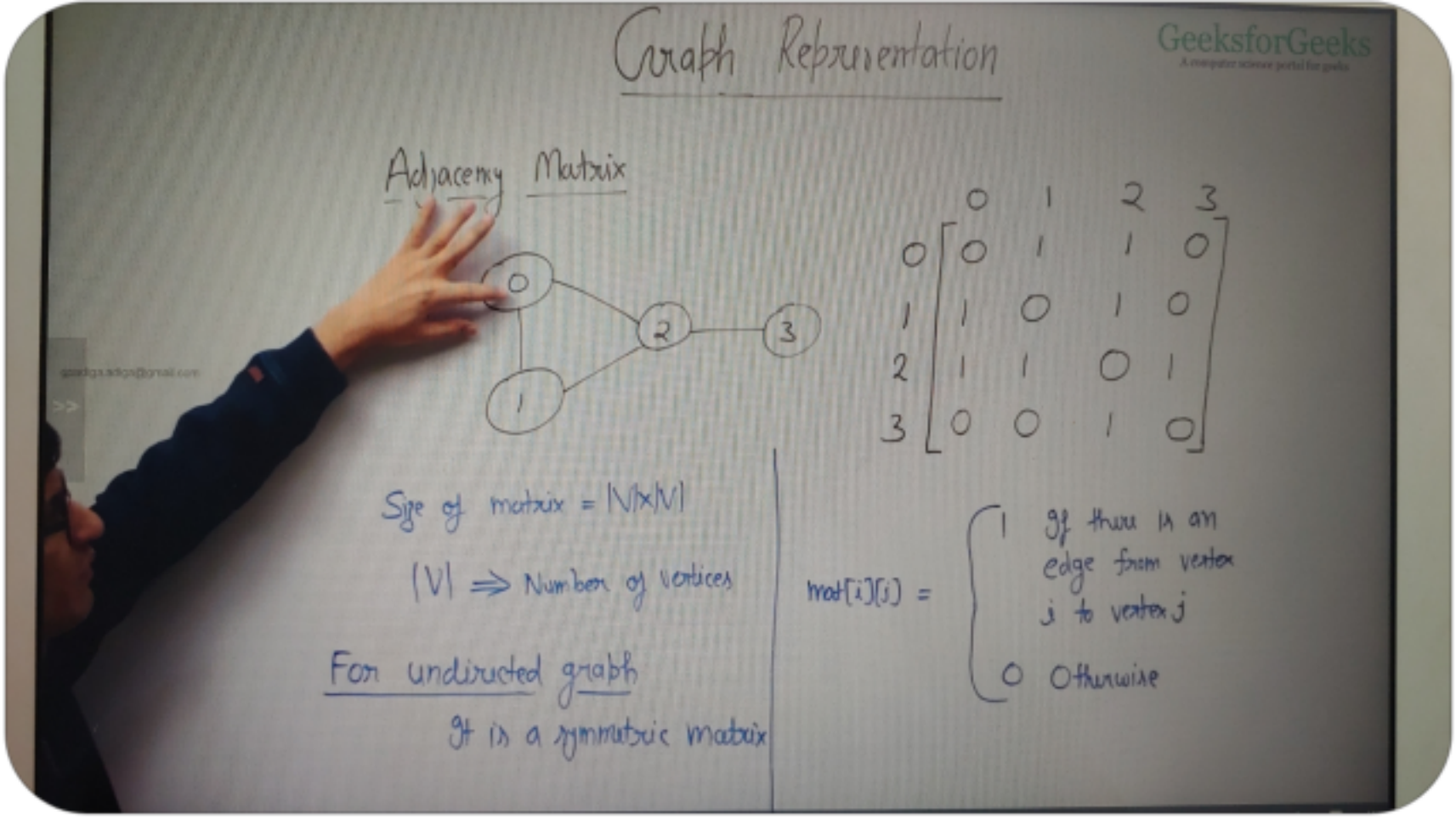
- **Cyclic and Acyclic Graph, Walk & Path:**

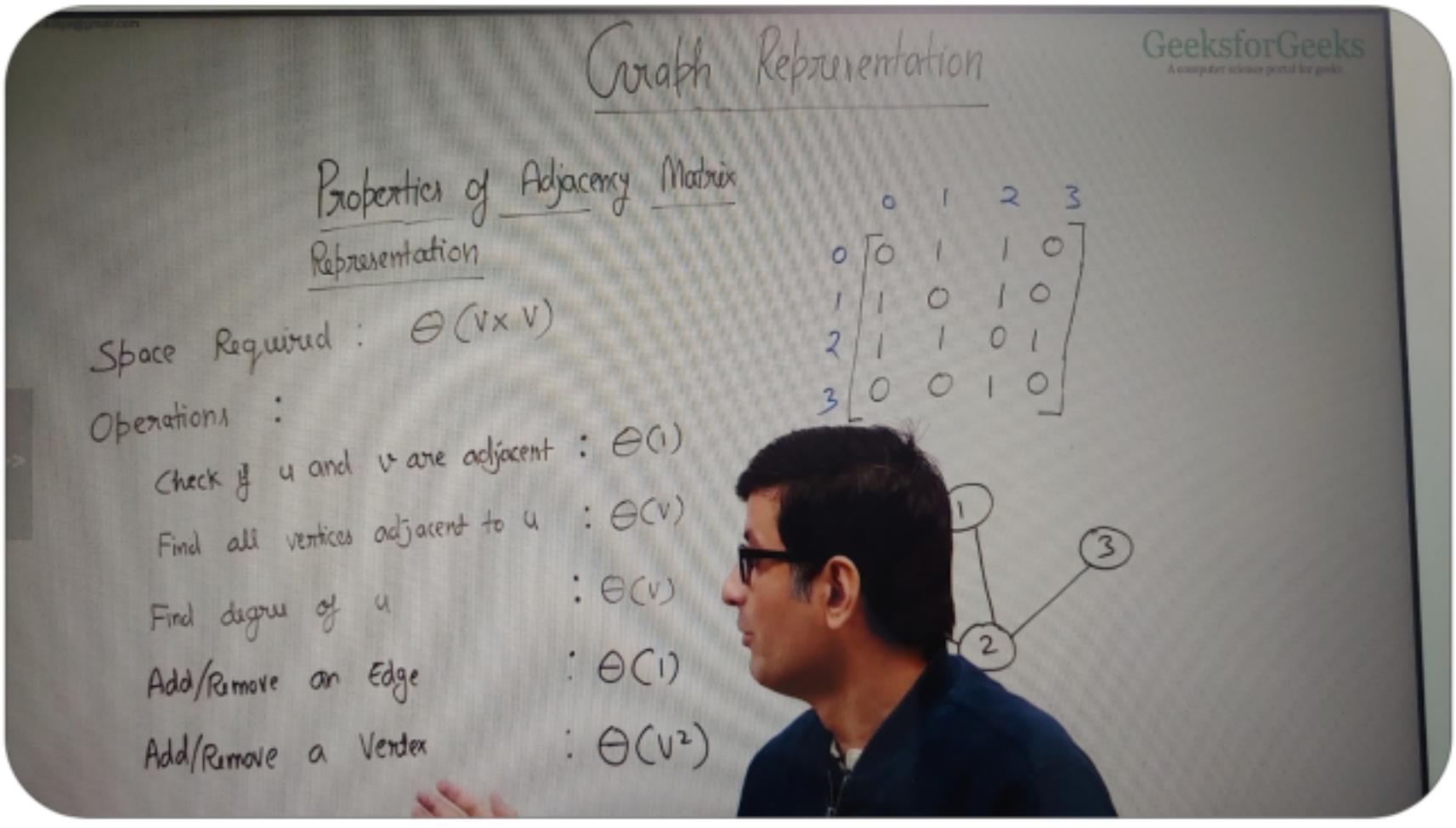


≡ ○ **Weighted & Unweighted Graph:** Real example of this is a graph of Road network where this roads connects multiple cities and weight of this edges are determined by length of roads. If two cities are far from each other then the edge which connects these 2 cities have more weight.



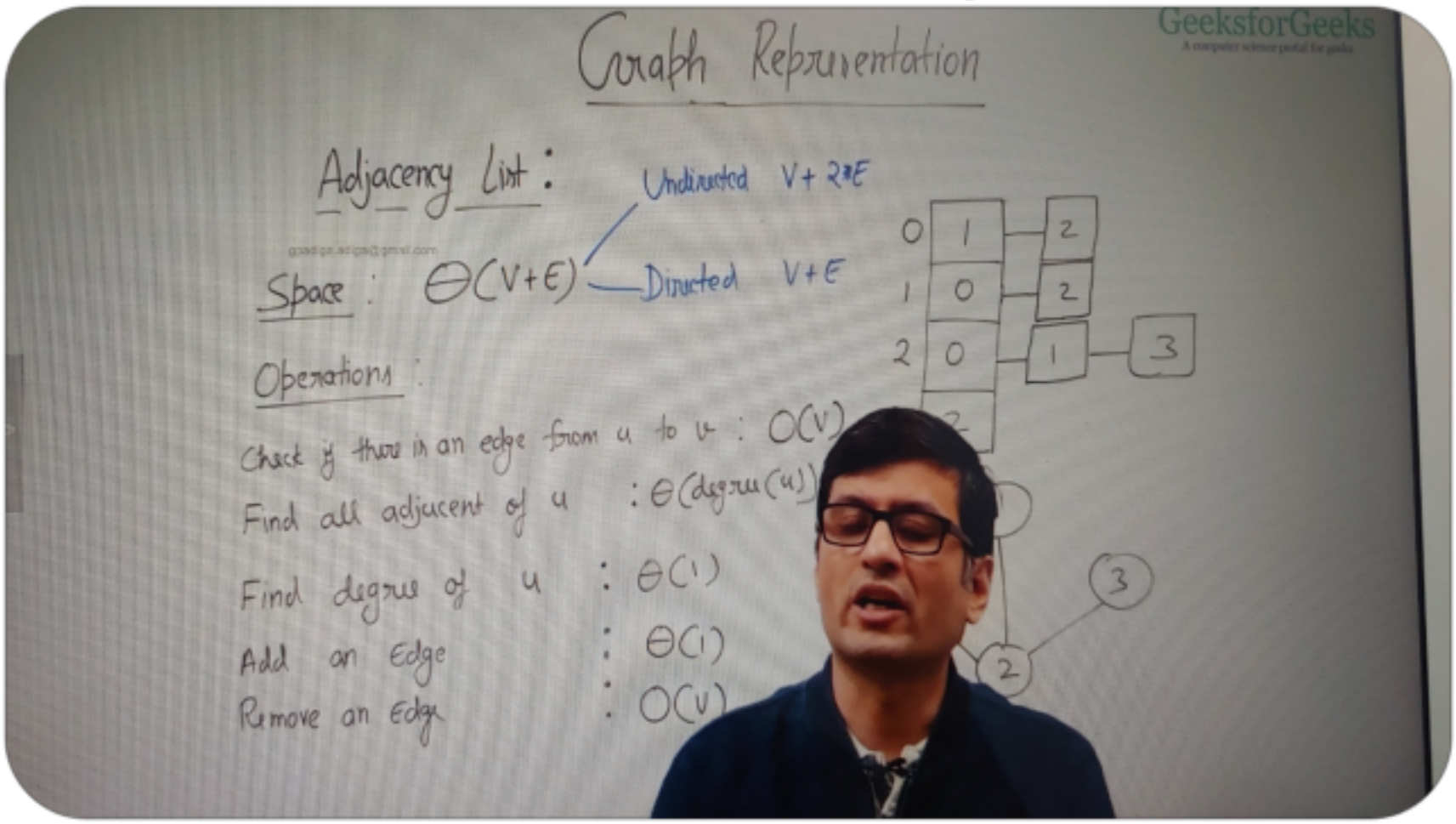
- **Graph Representation:** This defines how do we represent graphs in our programming language. Two of the popular approach is Adjacency Matrix and Adjacency List.
- **Adjacency Matrix**: Representing each relations in a Matrix form. For an Undirected graph its always a symmetric Matrix.





- **Adjacency List:** The problem with Adjacency Matrix representation is its stores the redundant information. It stores both connected and not connected

vertex details in the form of 0 & 1. In order to save memory we use Adjacency List representation which tells only about which vertices are connected to each other. It also makes finding all adjacent of a vertex operation faster than matrix representation.



- **Applications of BFS:** In BFS we go all adjacent first. **Shortest path** in an unweighted graph, Crawler in search engine, peer to peer network, social networking search, cycle detection etc
- **Applications of DFS:** In DFS we

go to all adjacent of one adjacent first and then go to all adjacent of other one. Cycle detection, Topological sorting, Strongly connected components, **Path Finding** (path between 2 cities) solving maze & similar puzzles.

- **Detect cycle:** In undirected path - using DFS with parent vertex. In directed path - Using DFS with recursion stack. Also by using Kahn's BFS Algoritym.
- **Topological Sorting**(Jobs and its dependencies): It works only for directed graphs. 2 solutions. One is by using DFS (through stack. It works for both cyclic or acyclic). Another one is by using Kahn's BFS Algo (through in-degree array. It works only for Acyclic).
- **Shortest Path:** 4 solutions. One is for directed/undirected, cyclic/ acyclic **unweighted** graph(By

using BFS with distance array). Second one is for the directed, weighted and **acyclic(**DAC**)** graph(By using topological sort DFS Stack and distance array). Third one is **Dijkstra's** algorithm which works for directed/ undirected, **cyclic/acyclic weighted** graph. Note that it does not work if any edge has **negative** value.(By distance array and picking minimum distance vertex and relaxing its adjacent). Last one is **Bellman Ford** Algorithm. (By finding shortest path of edge length 1 first, and then edge length 2 and so on)

- So if a graph is unweighted go for BFS approach. If a graph is positive weighted go for Dijkstra's algorithm. If a graph contains any negative weight go for Bellman Ford Algorithm.

- **Minimum spanning tree:** It will have v-1 edges and no loop. Its for the weighted, connected, acyclic & undirected graphs. There are 2 solutions. One is **Prims Algorithm** By keeping two MST sets, where we keep adding vertex which has minimum weight from non MST set to MST set. Another one is **Kruskal's** Algorithm, where we sort all the edges in increasing order and traverse through it and add each edge to MST set if its not causing any cycle.
- **Strongly connected component:** If a set of vertices forms a connection in such a way that every pair reachable from each other then this set of vertices are called strongly connected compoment. **Kosaraju's Algorithm** is used to print such

strongly connected components. It works for directed & cyclic graph. Here we first order the vertices in decreasing order of finish times in DFS. Then reverse all edge and finally do DFS of the reversed graph in the order obtained in step 1.

Theory Recheck

- ***Adjacency List implementation in java***
- ***Breadth First Search***
- ***Depth First Search***
- **Shortest path in an Unweighted graph**
- **Detect Cycle in Undirected graph**
- ***Detect cycle in a directed graph (Part 1)***
- ***Topological sorting (Kanh's BFS based algorithm)***
- **Detect cycle in a directed graph**

(Part 2)

- ***Topological Sorting(DFS Based Algorithm)***
- ***Rest of all the problems***

Problem Recheck

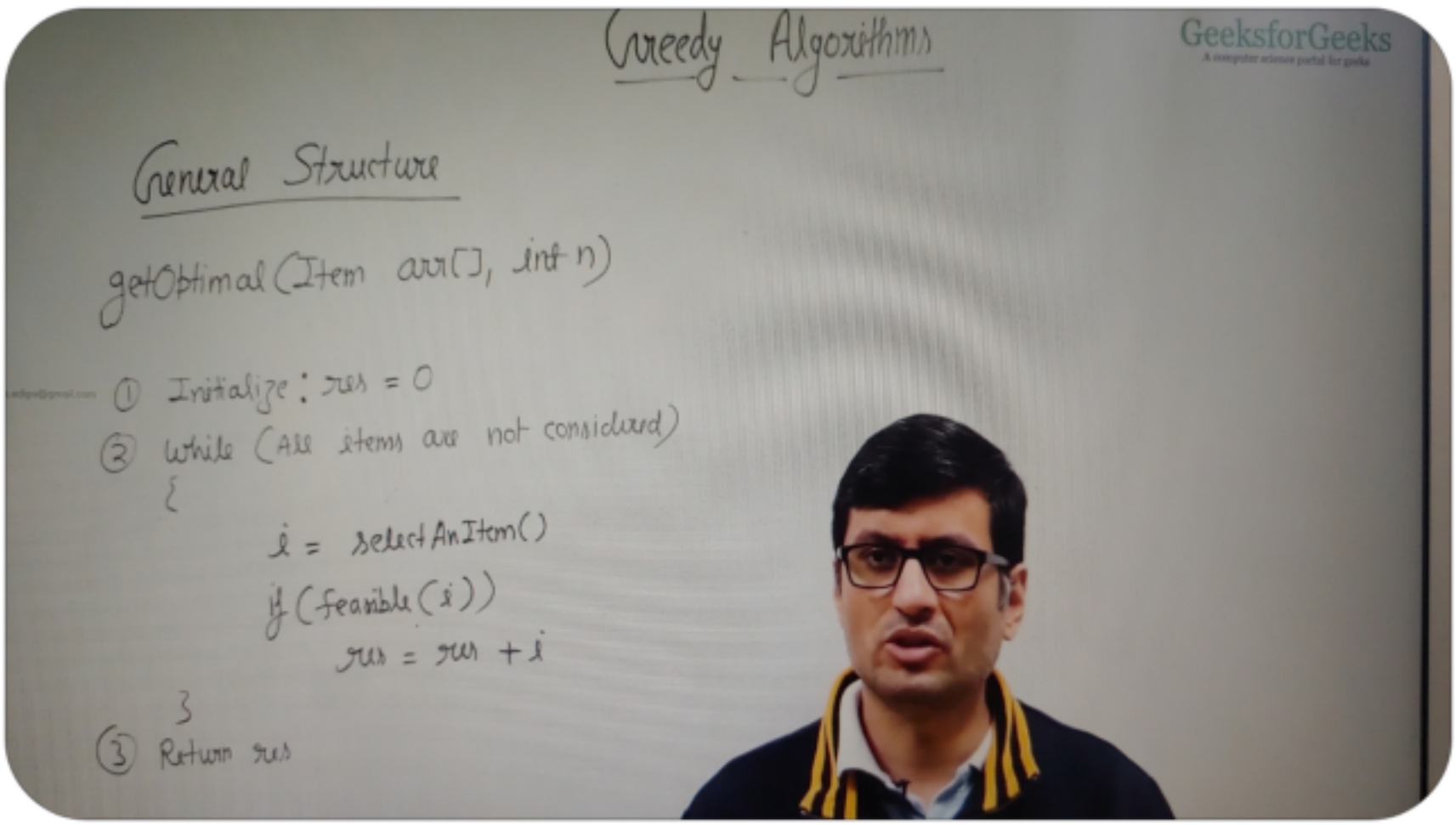
- **BFS of graph**
- **DFS of graph**
- ***Find the nunber of islands***
- ***Detect cycle in a undirected graph***
- **X Total shapes**
- **Unit area of largest region of 1s**
- ***Rotten Oranges***
- ***Steps by Knight***
- ***Minimum swap to sort***

==========

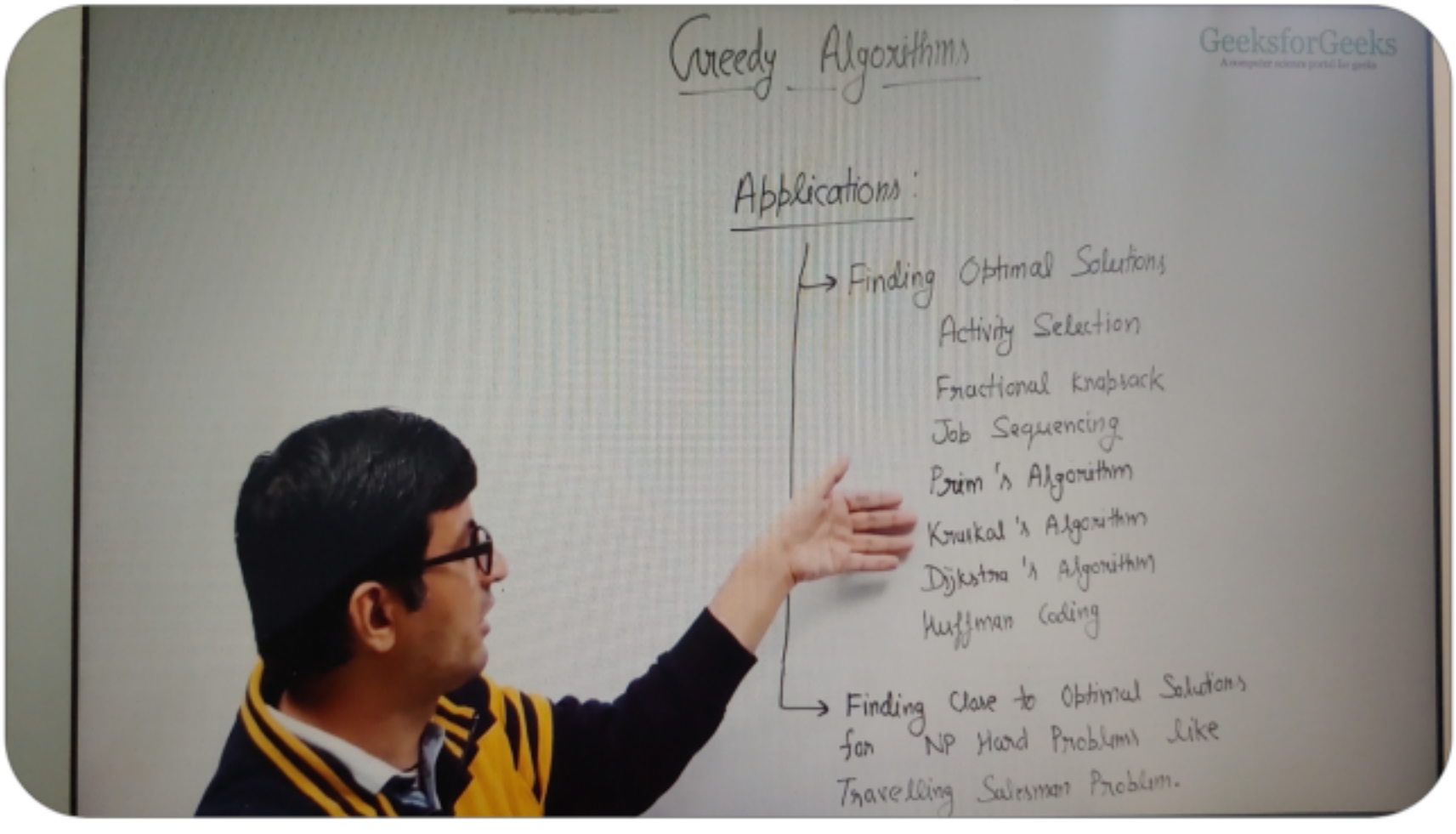
Greedy Algorithm

- It is used to find optimal solution for a problem like find shortest path or longest path or find

minimal number of coins required for an amount etc.

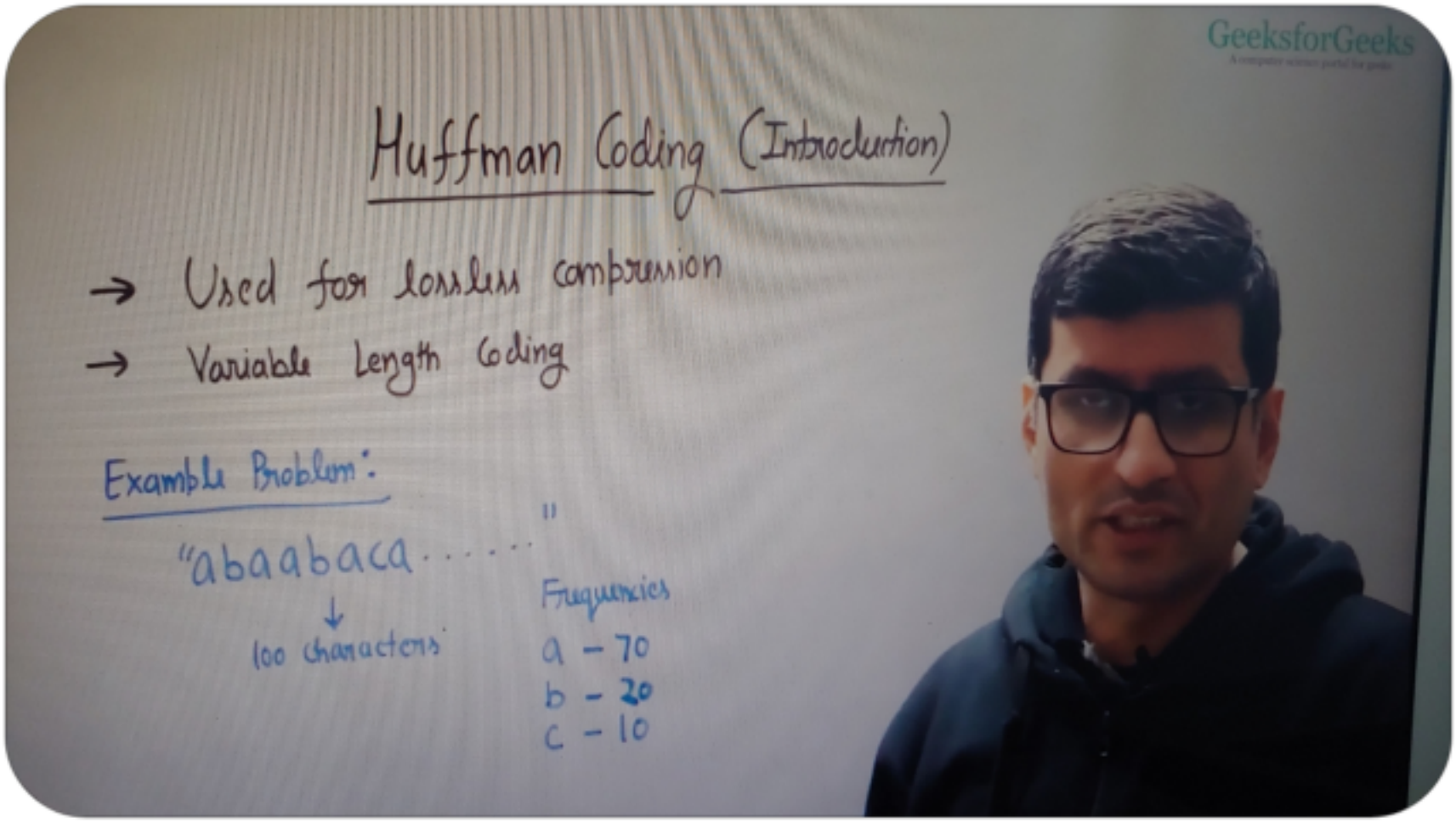


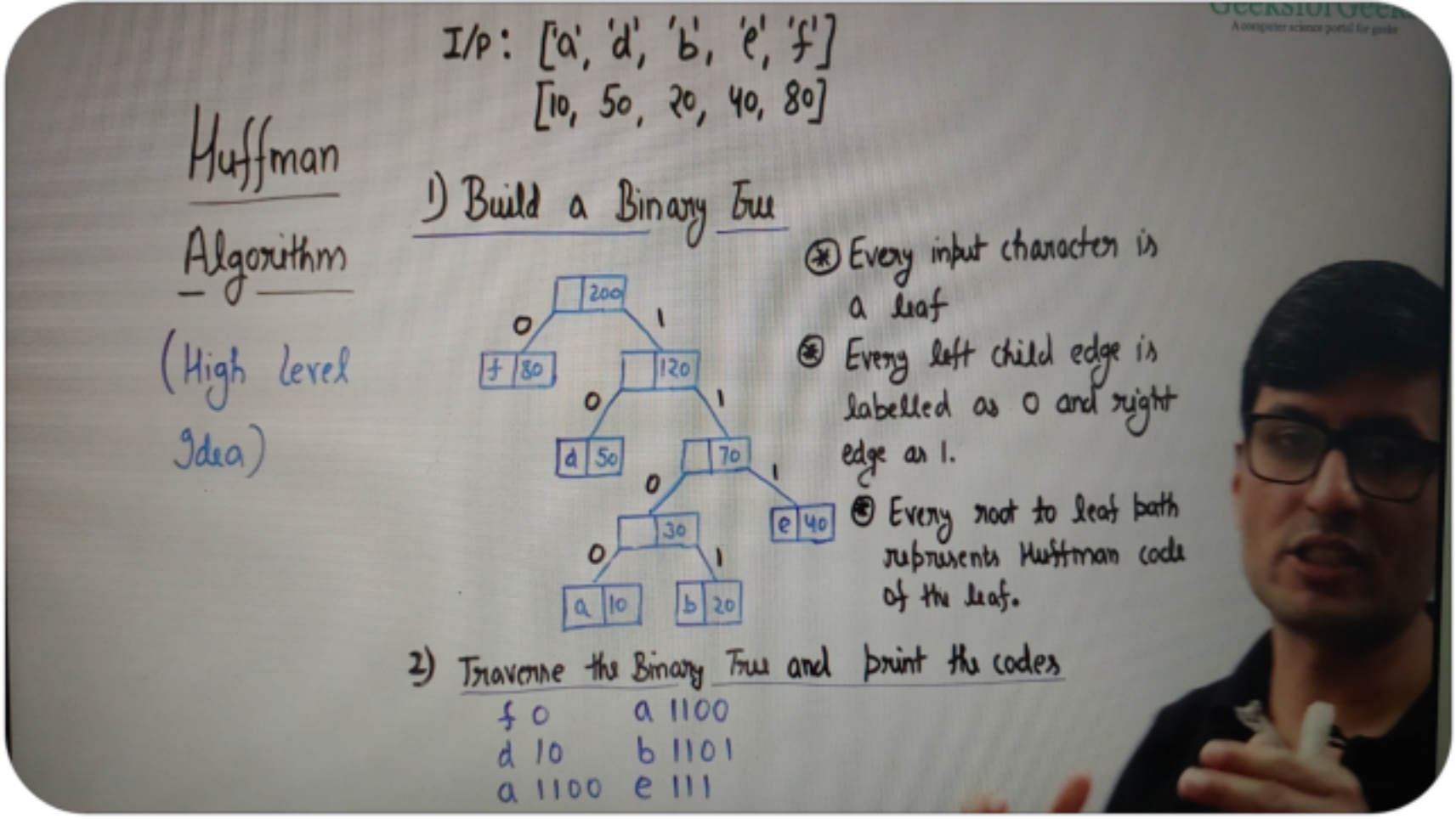
≡ ○ Applications of Greedy Algorithm



≡ ○ **Huffman Coding:** It is used for lossless compression which means when we compress the

data we should be able to decompress the original data. There should not be any loss in the original data. Its variable length coding, prefix free and greedy in approach.





Theory Recheck

- ≡ ○ Introduction to greedy algorithms - minimum coin
- ≡ ○ **Activity selection solution in java**
- ≡ ○ **Fractional Knapsack in java**
- ≡ ○ ***Job sequencing problem***
- ≡ ○ ***Java implementation of Huffman coding***

Problem Recheck

- ≡ ○ **Activity selection**
- ≡ ○ **N meetings in one room**
- ≡ ○ **Fractional Knapsack**
- ≡ ○ **Job sequence problem**

- ***Largest number with given sum***

==========

Backtracking

- We can use backtracking to optimise recursive solutions, instead of generating all possible answers and then filtering it out we can stop making recursive calls all together if we find isSafe as false.

Theory Recheck

- ***Rat in a maze***
- ***N Queen Problem***
- ***Sudoku Problem***

Problem Recheck

==========

Dynamic Programming

- Its an optimization over the plain recursion. In recursion calls, if we are finding solution for the same sub-problem again and again we can store and reuse the result of such sub-problems instead of reevaluating again and again, thus we can optimize the overall recursion problem.
- There are 2 ways to implement the sub-problems one is memoization (also called top down) & other one is tabulation (also called bottom up).