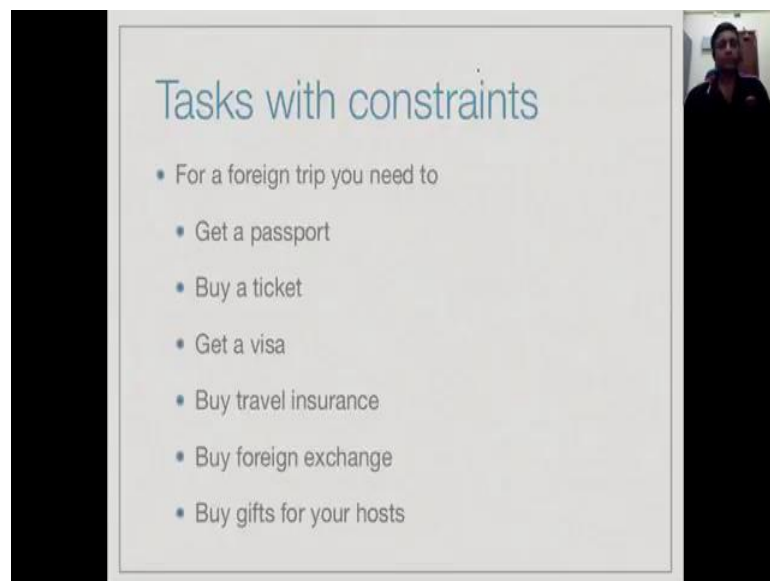


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Module – 06
Lecture - 23
Directed Acyclic Graphs (DAGs)

We now turn our attention to a very interesting and important class of graphs called Directed Acyclic Graphs or DAGs.

(Refer Slide Time: 00:08)

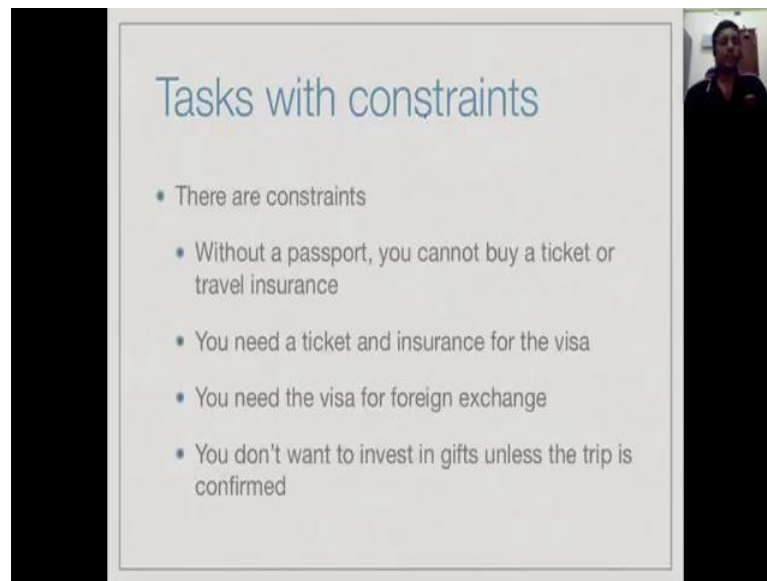


The slide is titled "Tasks with constraints" in a light blue font. It contains a bulleted list of tasks for a foreign trip. The tasks are: "Get a passport", "Buy a ticket", "Get a visa", "Buy travel insurance", "Buy foreign exchange", and "Buy gifts for your hosts". The slide is part of a video lecture, as indicated by the small video feed of the professor in the top right corner.

- For a foreign trip you need to
 - Get a passport
 - Buy a ticket
 - Get a visa
 - Buy travel insurance
 - Buy foreign exchange
 - Buy gifts for your hosts

So, to motivate this class of graphs, let us look at a problem where we have a bunch of task to perform with some constraints. Suppose, we are going on a foreign trip, then of course, we need a passport, we need to buy a ticket, we require a visa probably, we want to buy some travel insurance, we probably need some foreign exchange as well and perhaps you want to buy some gifts for our hosts.

(Refer Slide Time: 00:36)

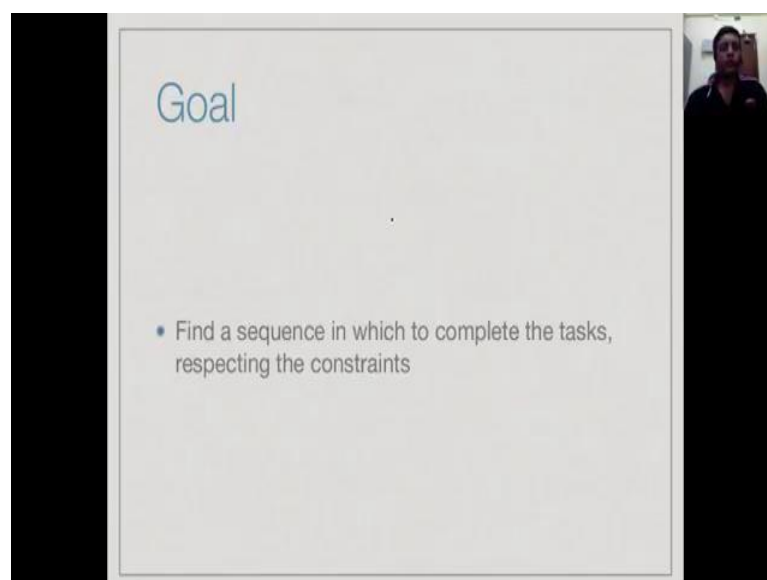


The slide is titled "Tasks with constraints" in a blue font. It contains a bulleted list of tasks and their dependencies. A small video inset of a person is visible in the top right corner of the slide.

- There are constraints
 - Without a passport, you cannot buy a ticket or travel insurance
 - You need a ticket and insurance for the visa
 - You need the visa for foreign exchange
 - You don't want to invest in gifts unless the trip is confirmed

Now, these tasks are dependent on each other in certain ways, without a passport you cannot buy a ticket, not even buy any travel insurance. For the visa, you need both the ticket and the insurance to be available and without a visa, the bank will not give you foreign exchange. And finally, you would not like to buy gifts for your hosts, unless the trip is confirmed. So, unless you have all these things including the visa in hand, you do not want to invest in the gift.

(Refer Slide Time: 01:06)



The slide is titled "Goal" in a blue font. It contains a single bullet point stating the objective. A small video inset of a person is visible in the top right corner of the slide.

- Find a sequence in which to complete the tasks, respecting the constraints

So, our goal is that given these constraints in what sequence should we perform these six operations, getting a passport, buying a ticket, getting insurance, getting a visa, buying foreign exchange and buying gifts for our hosts. What sequence should we do it, so that

whenever we want to approach a task, the constraints that are required for the task are satisfied.

(Refer Slide Time: 01:31)

Model using graphs

$T_1 \rightarrow T_2$

- Vertices are tasks
- Edge from Task1 to Task2 if Task1 must come before Task2
- Getting a passport must precede buying a ticket
- Getting a visa must precede buying foreign exchange

So, as you would expect we will model this using a graph. In this graph, the vertices will be the tasks and then you will have an edge pointing from T 1 to T 2, if T 1 must come before T 2, in other words T 2 depends on T 1 you cannot do T 2 unless T 1 has been completed. So, as an example getting a passport must come before buying a ticket, so T 1 is getting a passport, T 2 could be getting a ticket. Similarly, you must buy a, have a visa before you buy a foreign exchange. So, there will be an edge from getting a visa to buy a foreign exchange.

(Refer Slide Time: 02:13)

Our example as a graph

Get passport → Buy ticket → Get visa → Buy foreign exchange

Get passport → Buy insurance → Get visa → Buy foreign exchange

Order of tasks should respect dependencies

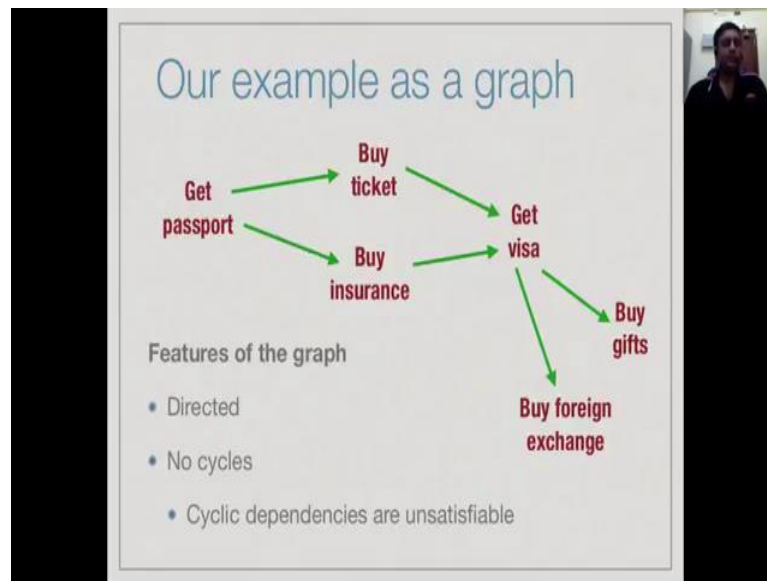
- Passport, Ticket, Insurance, Visa, Gift, Forex
- Passport, Insurance, Ticket, Visa, Forex, Gift
- Passport, Ticket, Insurance, Visa, Forex, Gift
- Passport, Insurance, Ticket, Visa, Gift, Forex

So, if we look at the constraints that we wrote this is the graph that we had, so we had a constraints with sets we need a passport to buy a ticket, we need a passport to buy insurance, we need both a ticket and insurance to get a visa. So, there are two constraints pointing to visa. Then, you need a visa to buy a foreign exchange and finally, you said we will buy a gift only if the trip is confirmed and at some point at this stage when all these operations are done, we can assume that the trip is confirmed, because nothing is blocking as getting on the plane.

So, this is a graph that we have and now our goal is to sequence these six operations, in such a way that whenever we want to perform a task, whatever it depends on has already been done. So, we can see that you need a passport to do anything, so we always need to start with getting a password. Now, there is no dependency between buying a ticket and buying insurance as per become constraints we have, so far. So, after password you can either buy a ticket first and then buy insurance or you can buy insurance first and then buy a ticket.

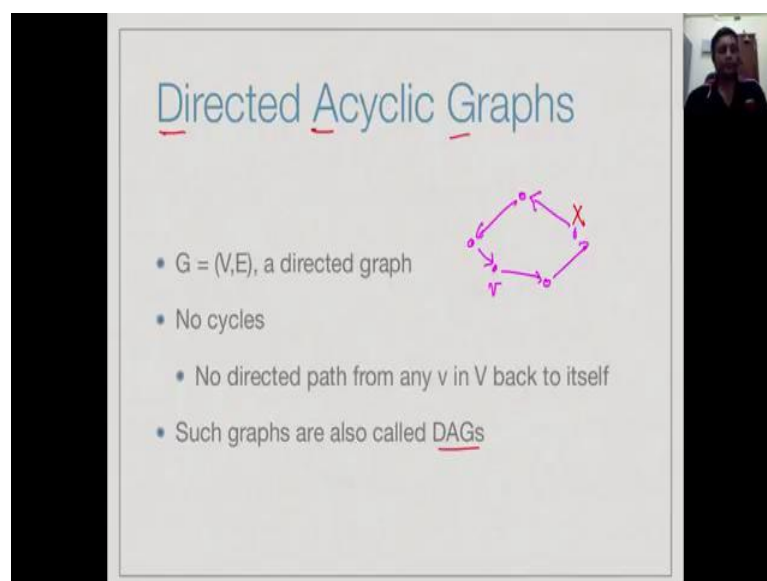
So, there is a different ordering possible which does not violate the constraints, on the other hand for a visa we need both. So, visa must come after both ticket and insurance, but again having done the visa, then there is no constraint between buying the foreign exchange and buying gifts. So, you could do the foreign exchange before the gift or the gift before the foreign exchange, so there are in this particular example there are two possible ways of reordering ticket and the insurance and there are two possible ways of reordering the gifts and the foreign exchange. So, overall there are four different sequences which are compatible with these constraints.

(Refer Slide Time: 03:51)



So, this class of graph is an important class and it has two important features, one is of course, it is directed. Because, these dependencies are from one task to another task, it is not a symmetric dependency and there are no cycles. See, if you had a cycle it would be that group of tasks depend on each other, so there is no way to start, because each task depends on something else in the cycle. So, you have to break the cycle somewhere in order to get started, but you cannot break it anywhere, because each task depends on something else in the cycle. So, this graph will have directions on the edges and they cannot be any cycles in this graph.

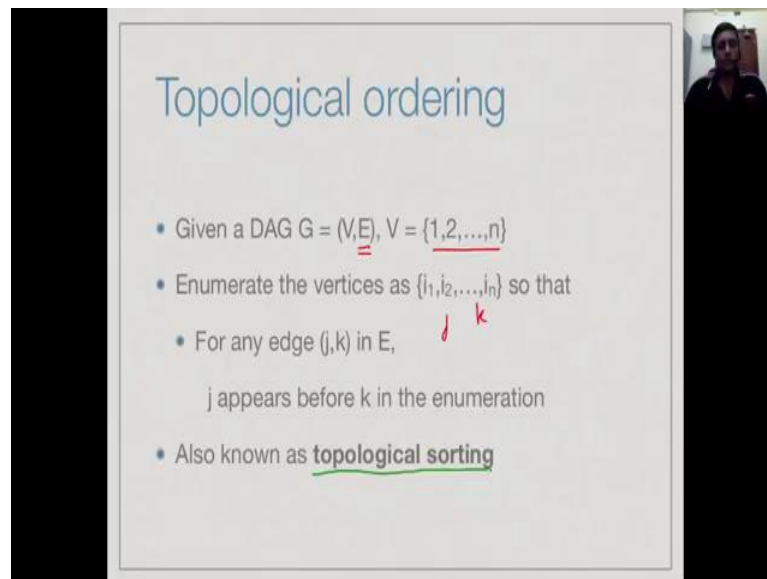
(Refer Slide Time: 04:28)



So, we call such a graph a directed acyclic graph, so a directed acyclic graph is just a

directed graph, in which there is no directed path from any vertex back to itself. So, if I started any vertex V , it should not be the case that I can follow a sequence of directed edges in the same direction and somehow come back to d . So, this should not be there, so it should not be this cycle, we abbreviate the name Directed Acyclic Graph as DAG. So, very often simplicity we will call this graph as DAGs.

(Refer Slide Time: 05:02)

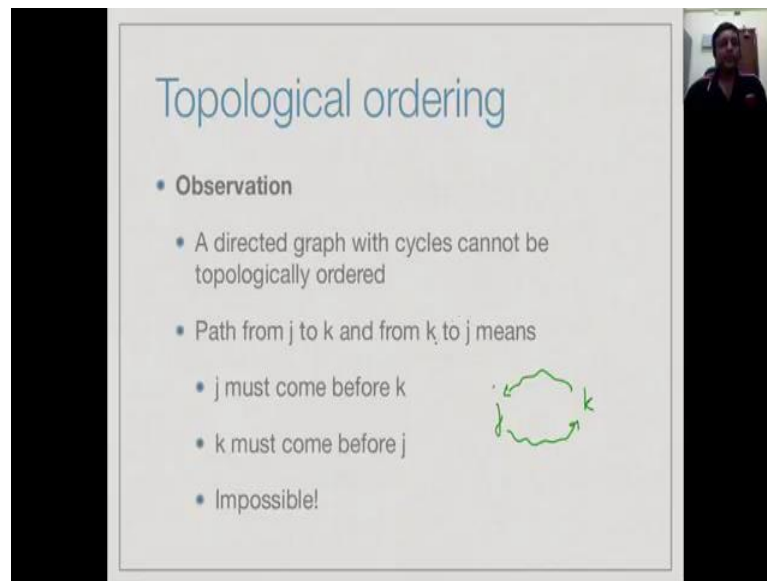


The slide is titled "Topological ordering" in blue text. It contains a list of bullet points:

- Given a DAG $G = (V, E)$, $V = \{1, 2, \dots, n\}$ (The E is underlined in red)
- Enumerate the vertices as $\{i_1, i_2, \dots, i_n\}$ so that
 - For any edge (j, k) in E , j appears before k in the enumeration (The j and k are handwritten in red)
- Also known as topological sorting (The text is underlined in green)

So, the problem that we had discussed in our example is that we have given a set of tasks and we want to write them out in a sequence with respect to the constraints, the constraints are nothing but, the edges. So, in general we are given a set of vertices these are our tasks abstractly 1 to n and we want to read, write our 1 to n in such a way that the constraints are respected. What this means is, that we will write out a sequence of numbers which is a permutation of 1 to n . In such a way that whenever there is a constraint of the form $j \ k$ that is represents edge $j \ k$, then in the numeration that we have perform j must come before k . So, it cannot be that we have to do j before k according to our constraint, but in the sequence that we produced k happens before j . So, the order of vertices in the final sequence must respect the constraints given by DAG, so for various reasons this is known as topologically sorting the DAG.

(Refer Slide Time: 06:02)



The slide is titled "Topological ordering" in a blue font. Below the title, there is a section labeled "Observation" with a blue bullet point. This section contains three main points, each with a blue bullet point: "A directed graph with cycles cannot be topologically ordered", "Path from j to k and from k to j means", and "Impossible!". The third point is further elaborated with two sub-points: "j must come before k" and "k must come before j". To the right of these sub-points is a diagram showing a cycle between two nodes, 'j' and 'k'. A green arrow points from 'j' to 'k', and another green arrow points from 'k' back to 'j', forming a loop.

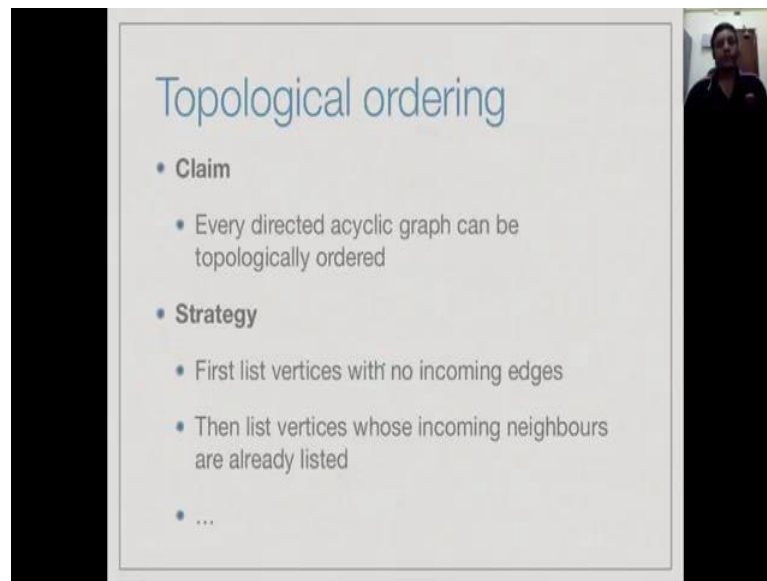
Topological ordering

- Observation
 - A directed graph with cycles cannot be topologically ordered
 - Path from j to k and from k to j means
 - j must come before k
 - k must come before j
 - Impossible!

So, the first observation is that if the directed graph had a cycle, then you will not be able to topologically order it. Because, if it had a cycle then for instance supposing j and k are vertices on the cycle, then you will have a path from j to k and a path from k to j. Now, it is easy to see that the topological ordering constraint extend to paths that is if I have j before k as an edge, I know that j must appear before k in the final sequence, also it has the path from j to k, then there is a sequence of dependencies from j to k. So, j must appear before k.

Now, if I have a cycle it says that j must come before k and k must come before j. So, there is no way to break this ((Refer Time: 06:45)), so we will end up with this situation where we cannot order this set of task to respect the constraints. So, the graph has cycles, then it is clear that there is no topological ordering possible.

(Refer Slide Time: 06:58)



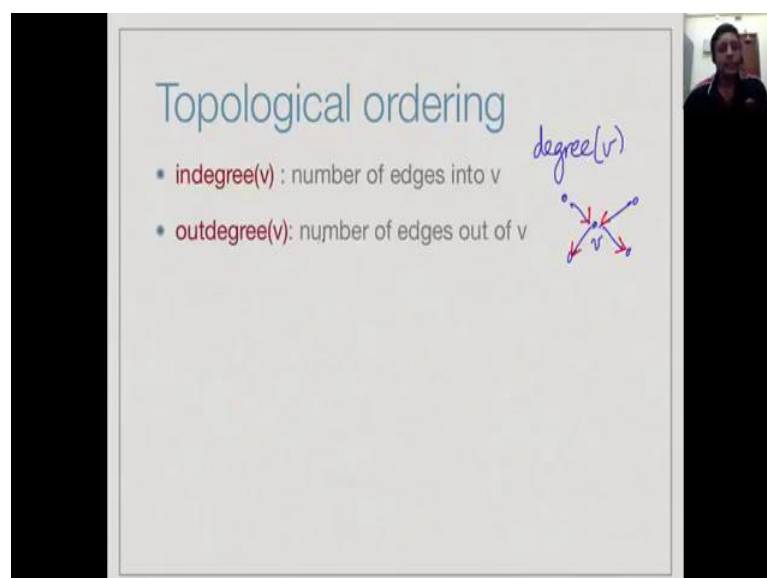
Topological ordering

- **Claim**
 - Every directed acyclic graph can be topologically ordered
- **Strategy**
 - First list vertices with no incoming edges
 - Then list vertices whose incoming neighbours are already listed
 - ...

So, what we claim; however, is that for DAGs there is no cycle, the graph is actually acyclic then we can always order it topologically. So, this strategy is to order the vertices as follows, you first list all the vertices which have no dependencies. In our earlier example, the vertex which has no dependencies was getting passport, we did not need to do anything before getting a passport, so we can do that first.

Now, once we are top-down that the dependency you see any vertex which all its dependencies that now satisfied and then we can numerate that. So, we can systematically list out vertices with no incoming edges, then vertices all whose incoming edges are already been accounted for a numeration and so on.


(Refer Slide Time: 07:45)



Topological ordering

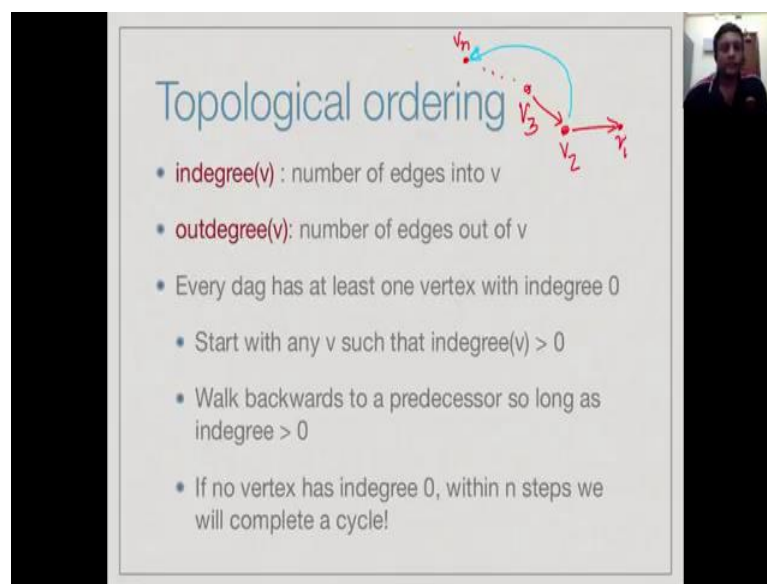
- **indegree(v)** : number of edges into v
- **outdegree(v)** : number of edges out of v

degree(v)



So, to formalize this notion we introduce some terminology, so recall that for an undirected graph, we use the term degree of v to refer to the number of vertices connected to v . So, v was connected by an edge before vertices, then we would say that the degree of v is 4. Now, since we have a directed graph we have a direction on the edges, we have some edges which are coming in and some edges which are going out. So, we separate out the degree in to the indegree and the outdegree. So, the indegree is the number of edges pointing into v directed into v , the outdegree of v is a number of edges pointing out of v .

(Refer Slide Time: 08:24)



Topological ordering

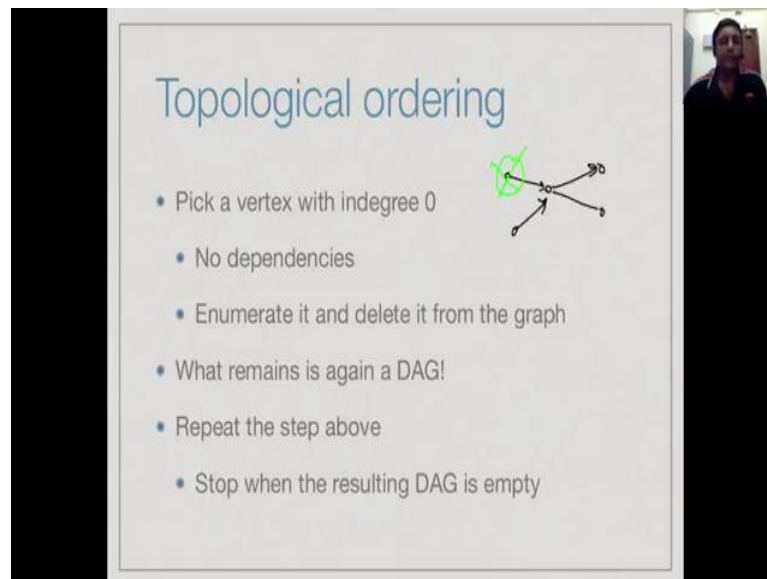
- **indegree(v)**: number of edges into v
- **outdegree(v)**: number of edges out of v
- Every dag has at least one vertex with indegree 0
 - Start with any v such that $\text{indegree}(v) = 0$
 - Walk backwards to a predecessor so long as $\text{indegree} > 0$
 - If no vertex has indegree 0, within n steps we will complete a cycle!

So, our first claim is that every DAG has at least one vertex with in degree 0, in terms of are example a vertex with in degree 0 is something which has no dependencies, nothing it does not depend on anything, this nothing pointing into it. Now, how do we proof this where supposing we start with any vertex v such that has in degree greater than 0, since it has in something pointing into it, then it must have some edge coming into it, so let us called at b 2.

Now, supposing this does not having in degree 0, then it must also have something pointing it to. So, then I get a third vertex, so in this way if I keep finding that the vertices have encountering have in degree greater than 0, eventually I must enumerate all the vertices in my graph. Now, if there is still not a case that the n th vertex there are n vertices in the n th vertex still does not increase 0 then it must have an incoming edge, but they cannot be from a new vertex. So, it must point from one of the existing vertices which have already seen before.

So, therefore, if I have a continuous sequence of vertices all of which are pointing to each the previous one with in degree not equal to 0, then I will end up with a cycle, but this is the contradiction, because we have an acyclic graph. So, in any directed acyclic graph, there must be at least one vertex with in degree 0 which corresponds to a task with more dependencies from where we can start or a numeration of the tasks.

(Refer Slide Time: 09:58)



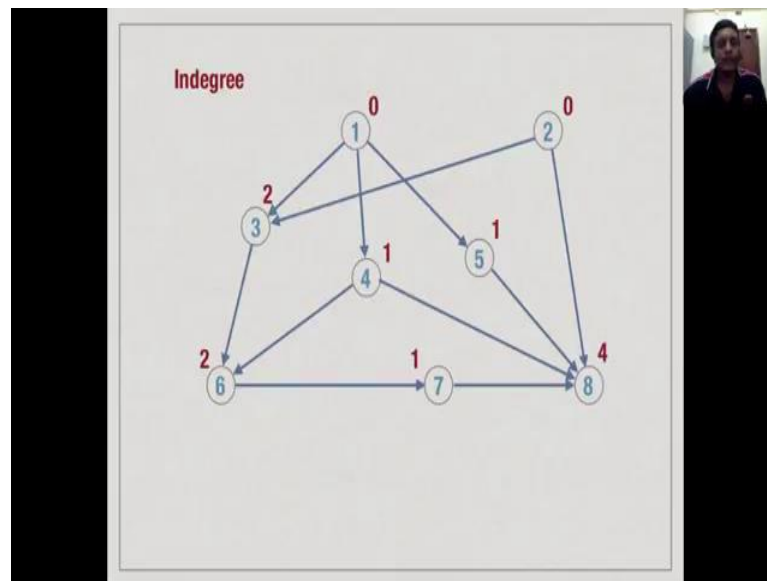
The slide is titled "Topological ordering" in a blue font. To the right of the title is a small directed graph with four nodes. The leftmost node is highlighted with a green circle. Three arrows point from this node to three other nodes on the right. One of these three nodes has two arrows pointing to a final node on the far right. The list of steps is as follows:

- Pick a vertex with indegree 0
- No dependencies
- Enumerate it and delete it from the graph
- What remains is again a DAG!
- Repeat the step above
- Stop when the resulting DAG is empty

So, this is a more elaborate version of the algorithm that it described earlier. So, we pick a vertex with in degree 0, we call that such a vertex has no dependencies, now we enumerated because it now has it is available for enumeration and then we deleted from the graph. So, when we delete a vertex with in degree 0 from a graph is suppose when we have a DAG like this. So, supposing we pick this one and we deleted, then clearly what remains is the DAG.

Because, it still directed and we have not introduced an cycle, so it is already acyclic and by deleting an edge we cannot introduce a cycle. So, clearly it is a DAG, so we can apply the same criterion, this new DAG must also have at least one degree with vertex with in degree 0. So, we can numerate that and keep going, so we keep enumerating vertices with in degree 0 and through the DAG becomes empty, each n vertex to enumerate we will delete from the DAG.

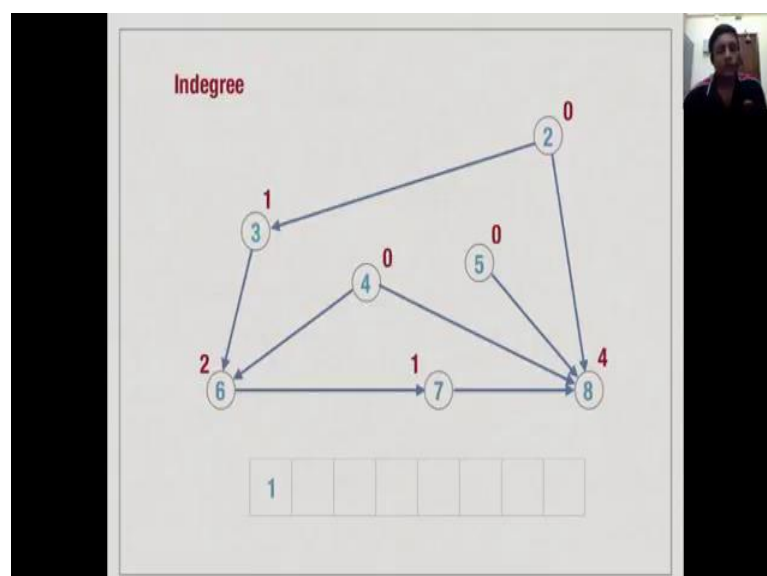
(Refer Slide Time: 11:00)



So, let us apply the strategy to this DAG, so we first begin by labelling every vertex by its in-degree. So, we have indicated the in-degree of every vertex. So, for instance 1 and 2 have no incoming edges. So, they have in-degree 0, vertex 3 has 2 edges coming as in-degree 2, vertex 8 has 4 edges coming inside and in-degree as 4 and so on, now we have to pick up a vertex of in-degree 0 and eliminate it.

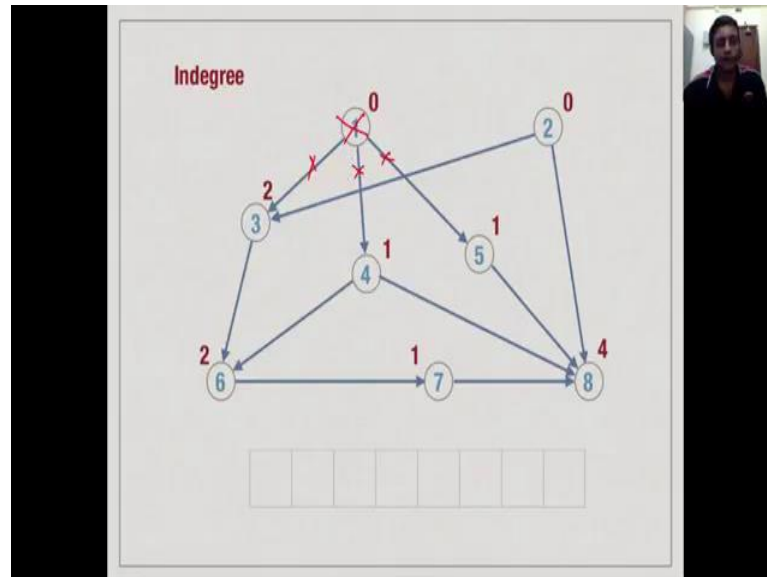
So, we have a choice between 1 and 2, so let us suppose we start with 1, so we start with 1 we eliminated and now when we eliminate we also eliminate the edges going out of it. So, the edges coming into the 3, 4 and 5 will reduce by 1, because a vertex 1 is gone, so the edges coming into them reduce by 1 their in-degree is also reduced by 1.

(Refer Slide Time: 11:57)



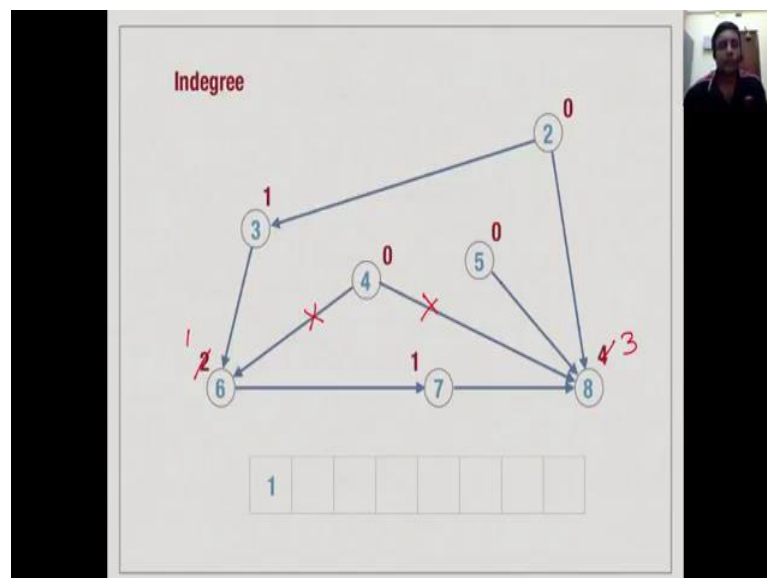
So, what happens on eliminate 1 is that we enumerated and we reduce the in degrees of 3, 4 and 5 from 2, 1 1 to 1 0 0.

(Refer Slide Time: 12:04)



So, recall that before that the in decrease or 2 1 and 1, now these edges which are coming into them have been deleted. So, when we delete this, we also delete the incoming edges.

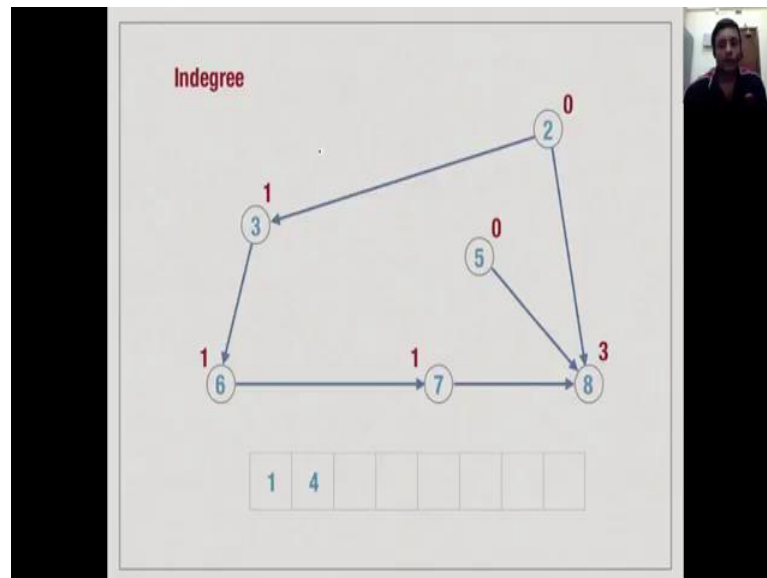
(Refer Slide Time: 12:14)



So, now we have 1 0 0, now we have three choices, two the original one which are in degree 0 and we have two new vertices 4 and 5 which correspond to tasks if you want to call them, whose prerequisite has been completed. So, tasks 1 was a only pre requested for 4, task 1 is only prerequisite ed for 5 it has been completed. So, 4 and 5 or now

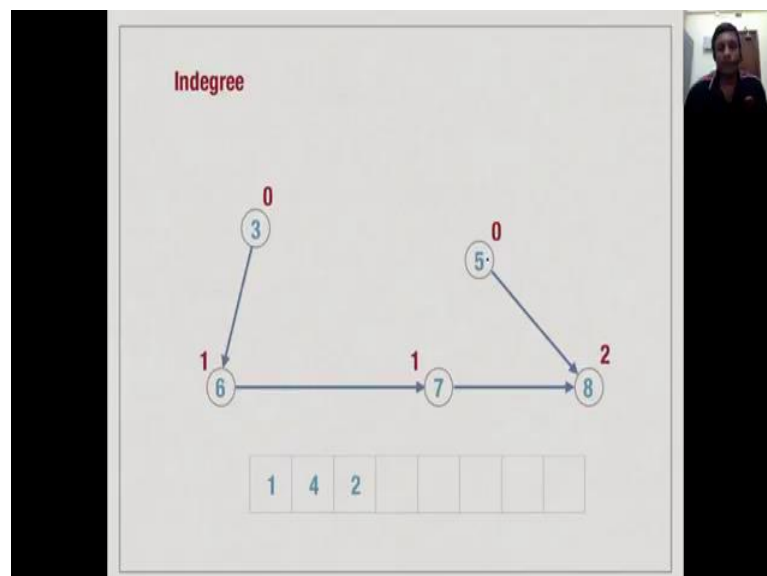
available, so we can choose any of 2, 4 and 5 it does not matter. So, let us suppose we choose 4 then again these two edges will go. So, this will reduce to 1 and this will reduce to 3.

(Refer Slide Time: 12:48)



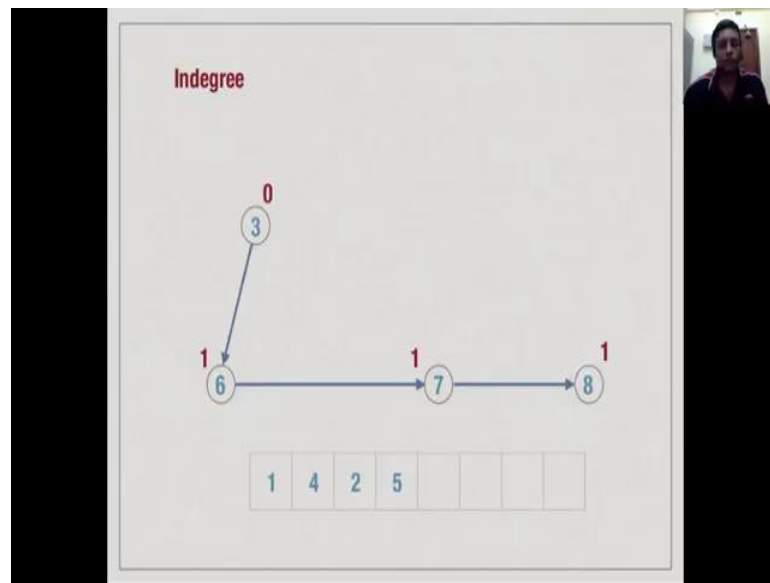
So, we can do that eliminate 4 and reduce the in degree of 6 from 2 to 1 in degree of 8 from 4 to 3, now perhaps which is decide to eliminate task 2.

(Refer Slide Time: 13:00)



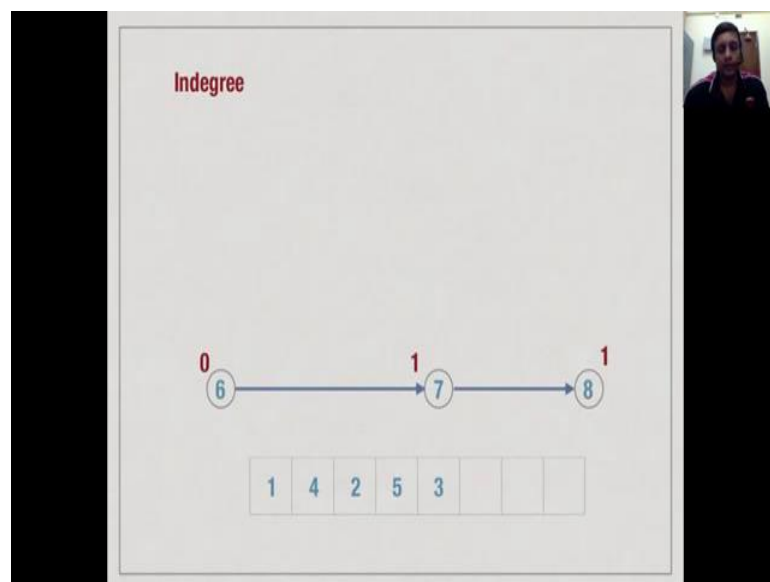
So, when you do pass 2 then the in degree of 3 reduces and the in degree of 8 again reduces. So, notice that 8 still has to pending in requirements only 5 and 7, so it cannot be done yet, but 3 and 5 are the available.

(Refer Slide Time: 13:14)



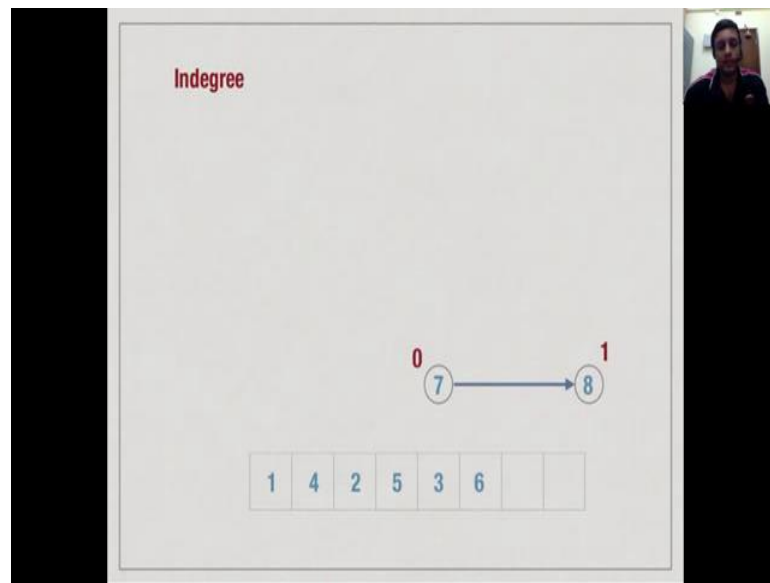
So, perhaps I do 5, so now 8 plus 1 and now have no choice, the only task with in degree 0 is 3. So, then I do 3, so now I can see that this is actually the way it is drawn is that is the sequence, I must do 3 before 6, 6 before 7, 7 before 8. So, I have no choice to this point I must enumerate as 3.

(Refer Slide Time: 13:32)



Then, 6.

(Refer Slide Time: 13:33)



Then, 7.

(Refer Slide Time: 13:34)



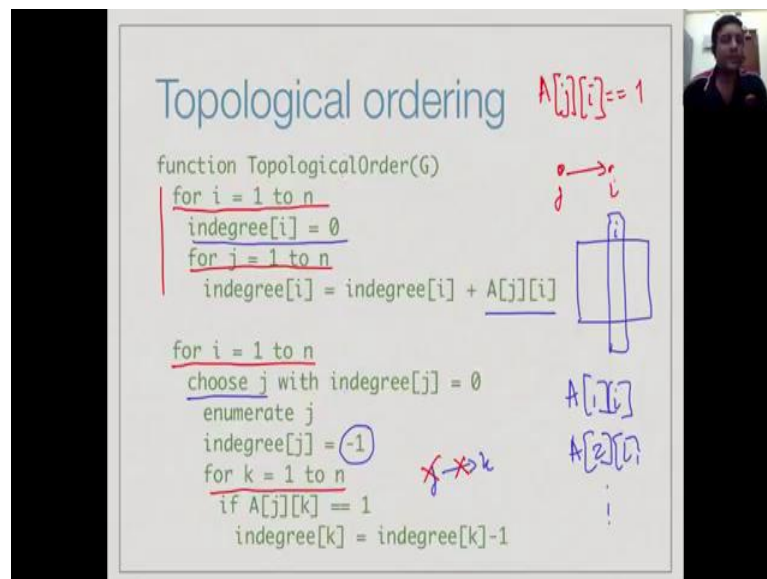
Then finally, 8.

(Refer Slide Time: 13:36)



At this point my graph is empty and I have to obtain a sequence of vertices which is a valid topological ordering, because every pair of vertices which occurs an edge in my original graph is order. So, that source of the edge appears before the target to the edge.

(Refer Slide Time: 13:52)



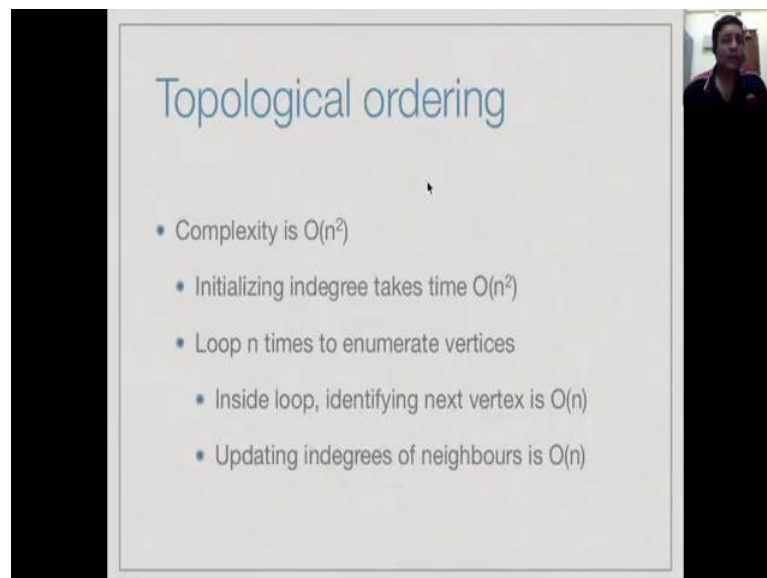
So, let us look at some pseudo code for the algorithm which we just executed by DAG. So, in this particular algorithm we first start by computing the in degree, so in order to compute the in degree, we need to find out how many for a vertex i we need to find out how many j satisfy the property that $A[j][i]$ is equal to 1, because this corresponds to an edge from j to i . So, when an adjacency matrix this corresponds to looking at a column containing line.

Because, in the column containing I , you will have entries at the form A_{1i} , A_{2i} and so on. So, we will have these entries and we want to scan all of these and then add up all the 1s. So, we start by setting in degree equal to 1, a in degree i equal to 0 and then for every row j we add A_{ji} . So, it is either 0 or 1 and so we therefore, collect all the incoming edges which pointing to i as the in degree of i , now we start enumerating. So, we know this is a DAG, so we know there is at least 1 j with in degree 0 at an every point.

So, we choose any such j , choose a j which has in degree 0 enumerate it, now when we enumerate we want to eliminated from the graph. Rather than, going an actually modifying the graph itself, we will just work with in degree as a kind of approximate version of that modified graph. So, we first set the in degree to minus 1 for this particular vertex. So, minus 1 means it cannot be in the graph, because you cannot have minus 1 edges pointing can have at least 0 edges of mode.

So, this effectively means that the j is not going to be considered hence 4 and now for every outgoing edge from j . So, wherever we have j pointing to k we want to decrement this, because we are going to eliminate this edge eliminating j , we eliminate this edge. So, for every k from 1 to n we scan out going neighbours of j and if $j k$ is an edge A_{jk} is 1 we reduce the in degree of k by 1.

(Refer Slide Time: 15:48)



Topological ordering

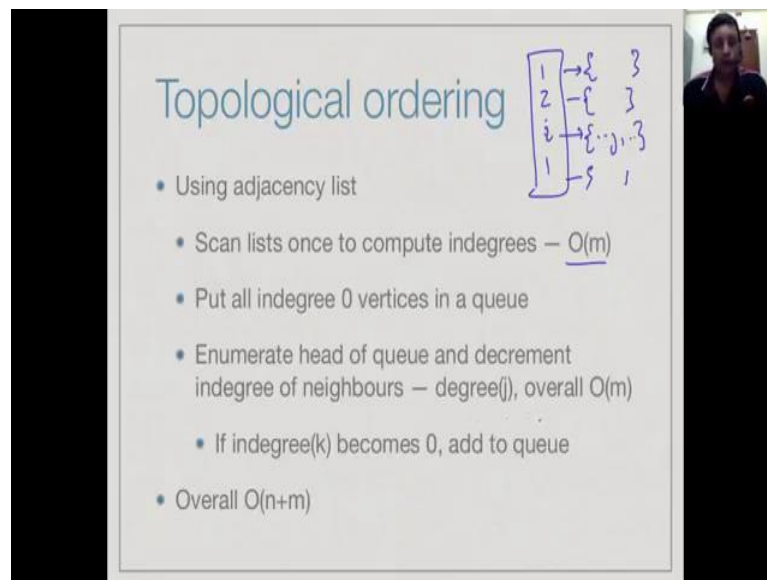
- Complexity is $O(n^2)$
 - Initializing indegree takes time $O(n^2)$
 - Loop n times to enumerate vertices
 - Inside loop, identifying next vertex is $O(n)$
 - Updating indegrees of neighbours is $O(n)$

So, what is the complexity of the algorithm is fairly easy to see that for this adjacency matrix or presents it is n square, as we saw initializing the in degree itself takes time n square ((Refer Time: 16:01)). Because, we have one outer loop from 1 to n and then for

each outer loop we have an inner loop from 1 to n , so this is a n squared loop. And then, when we enumerate the vertices again we have an outer loop which will enumerate every vertex once.

And then for the inner loop, we have to enumerate check all it is neighbours and decrement. So, we have $2n$ square loops and therefore, this whole thing takes order n square.

(Refer Slide Time: 16:29)



Topological ordering

- Using adjacency list
- Scan lists once to compute indegrees — $O(m)$
- Put all indegree 0 vertices in a queue
- Enumerate head of queue and decrement indegree of neighbours — $\text{degree}(j)$, overall $O(m)$
- If $\text{indegree}(k)$ becomes 0, add to queue
- Overall $O(n+m)$

Handwritten diagram showing an adjacency list structure:

```

graph LR
    1 --> 2
    1 --> 3
    2 --> 3
    3 --> 4
    4 --> 5
    5 --> 6
  
```

Now, we can as we saw with BFS and DFS, if we use an adjacency list we can be a little more clever and we can bring down this time to linear from n square we can bring it down to order n plus m . So, how do we do this? Well, we have this list, so we have list say 1, 2 and for each of these we have a list of it is neighbours. So, if you go through this as we said each edge in this, now this is a directed graph. So, each edges represented only once if an edge from i to j it will appear as an entry j in the list for i .

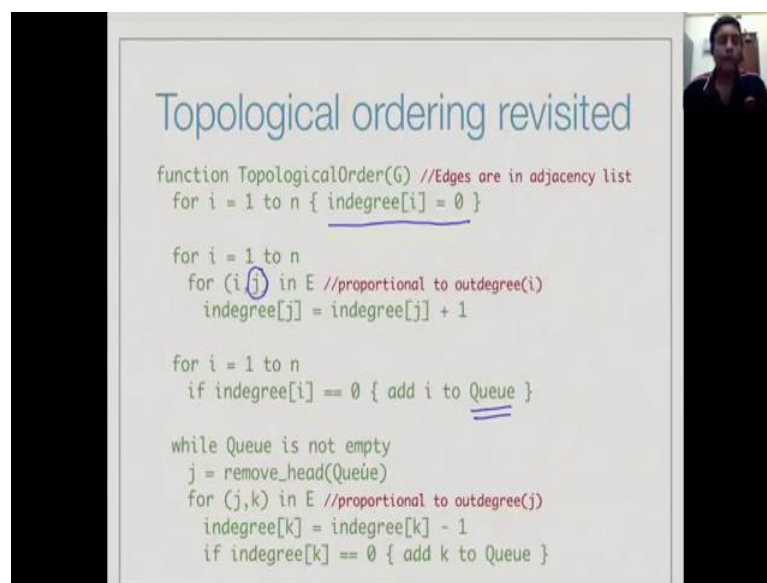
So, if you scan these lists every time we see a j , we know there is an edge pointing in to j and we will increment. So, we start of by setting all the in degrees to 0, we scan all the lists and every time we see an entry in a list, we increment this in degree. So, one scan of the list that is in time order n , we can find the in degrees. Now, we have the list of in degrees, so we can put all the in degree vertices into a queue, this makes it easy to find which vertex to enumerate next.

So, at the end of this scan we have done an order m scan to find all the in degrees, now and an order n scan we can put all the 0 degree vertices in to a queue. Now, we can do

the rest pretty much as we did much before, we enumerate the first vertex in the queue and then we go to its list which is now explicitly available to a adjacency list is outgoing neighbours, decrement its in degree and if any of those in degree is become 0 we can added to the queue.

So, we know it is to be processed now, so this becomes overall it take order m time to scan the list takes order n time to start the queue of and then this is the loop of order n where across all the updates we will overall update in degrees order n times, so this is order n plus m .

(Refer Slide Time: 18:26)



Topological ordering revisited

```
function TopologicalOrder(G) //Edges are in adjacency list
  for i = 1 to n { indegree[i] = 0 }

  for i = 1 to n
    for (i, j) in E //proportional to outdegree(i)
      indegree[j] = indegree[j] + 1

  for i = 1 to n
    if indegree[i] == 0 { add i to Queue }

  while Queue is not empty
    j = remove_head(Queue)
    for (j, k) in E //proportional to outdegree(j)
      indegree[k] = indegree[k] - 1
      if indegree[k] == 0 { add k to Queue }
```

So, here is the corresponding pseudo code, so the first step is to initialise the in degree to 0 for every vertex in our graph, then we go through all the edges. So, we do this by looking at each adjacency list. For each vertex we look at each neighbour i, j and the adjacency list and we for each of these we increment the in degree of j , because we are looking at edges pointing into j , not pointing out of i . So, this pointing into j will come in different list.

But, as and when we encounter them for each of them we will account for them and add one to it. Now, we go through the list one more time a list of vertices and every time will see an in degree 1, 0 we add i to the queue. And now we do this loop, till the queue becomes empty we know this at least one at every point remember along the graph a is DAG is not an empty, we know there is at least one in degree vertex with in degree i 0. So, they must be in the queue, because we adding them all originated the queue and each

one we generate we will add to the queue.

So, along with a queue is not empty we take of the first element of the queue, then we look at its adjacency list, decrement the in degrees of all those vertices and if any of them happens to now become in degree 0, we add to the queue. So, this becomes now a linear implementation of topological sort, using adjacency list and a queue to process the elements. Because, the reason why we need this queue is that otherwise we have to scan all the vertices every time to determine whether a vertex or become in degree 0, then that becomes an order n scan within this become order n square again.

So, we need the queue to make sure that we do not spend time trying to identify the next vertex to enumerate. We do not have to go through all the vertices and check the in degrees, when we see the in degree 0 we put it into the queue. So, automatically it will come out without having to do any further check. So, it gets observed in this order m work that we are doing here.