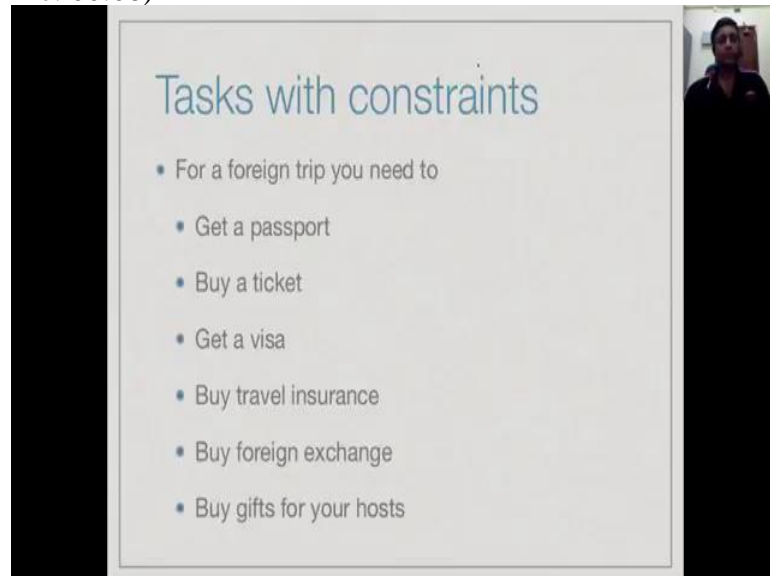


**Design and Analysis of Algorithms, Chennai Mathematical Institute**  
**Prof. Madhavan Mukund**  
**Department of Computer Science and Engineering,**

**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)

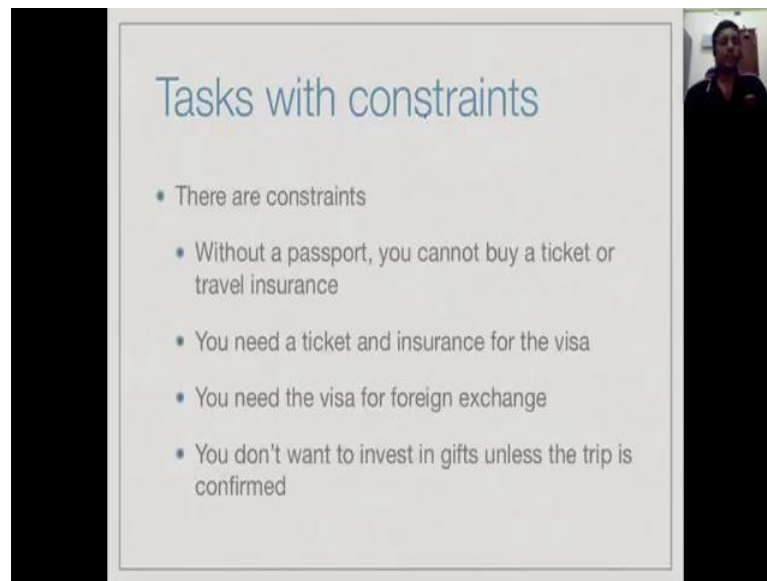
The image shows a video lecture frame. On the left, there is a black vertical bar. In the center, a white rectangular slide is displayed with the title "Tasks with constraints" in a blue, sans-serif font. Below the title, there is a bulleted list of tasks for a foreign trip. On the right side of the frame, there is a small inset video of a man, presumably the professor, wearing a dark shirt and looking towards the camera.

Tasks with constraints

- 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)

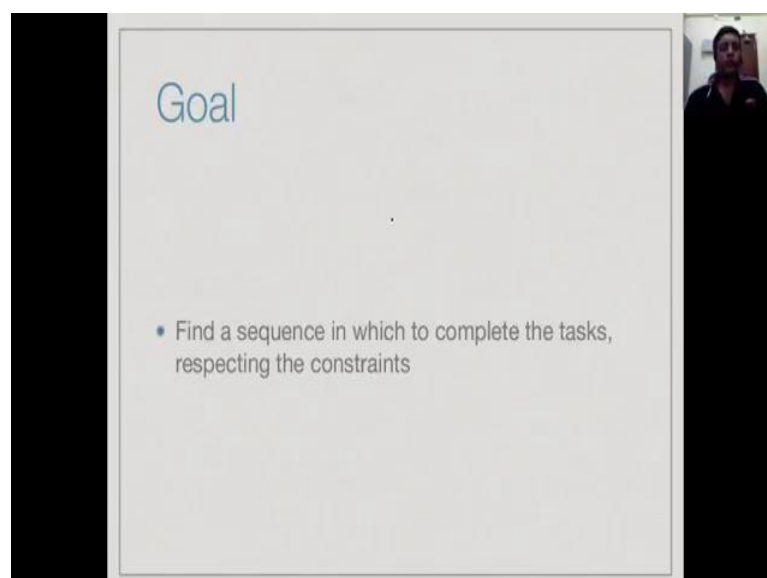


The slide is titled "Tasks with constraints" in a blue font. It contains a bulleted list of tasks and their dependencies. The tasks are: "Without a passport, you cannot buy a ticket or travel insurance", "You need a ticket and insurance for the visa", "You need the visa for foreign exchange", and "You don't want to invest in gifts unless the trip is confirmed".

- 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 bulleted list of the goal: "Find a sequence in which to complete the tasks, respecting the constraints".

- 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

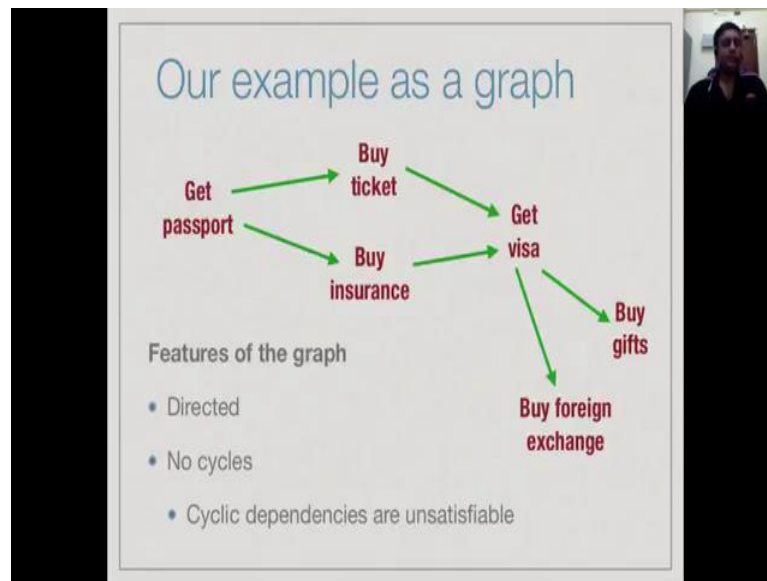
So, if we look at the constraints that we wrote this is the graph that we had, so we had a constraints with sets we need a passport to buy a ticket, we need a passport to buy insurance, we need both a ticket and insurance to get a visa. So, there are two constraints pointing to visa. Then, you need a 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.

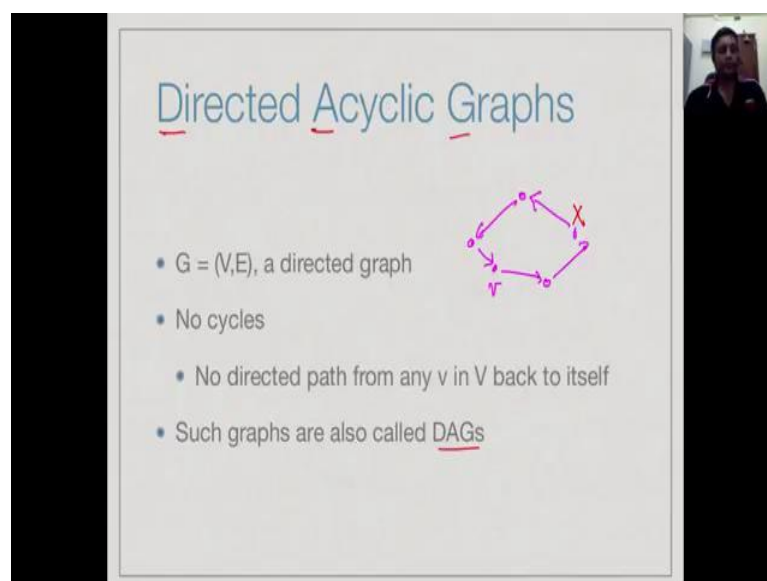


(Refer Slide Time: 03:51)



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.

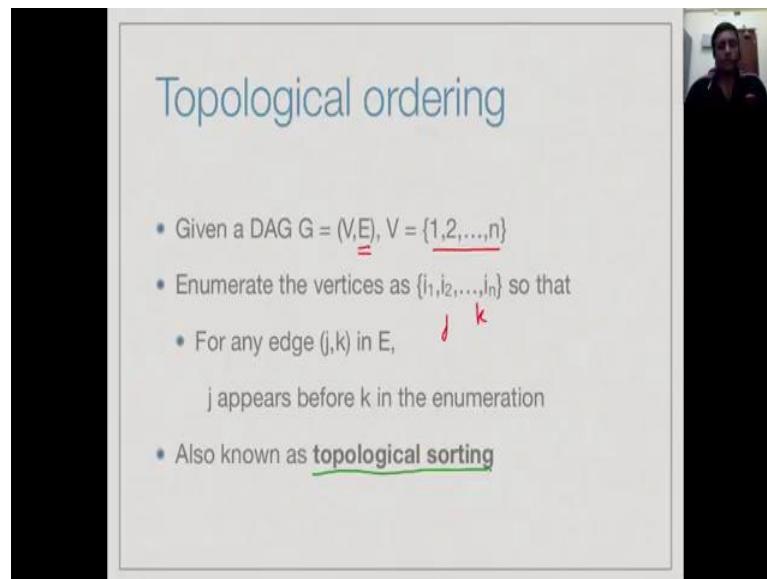
(Refer Slide Time: 04:28)



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)

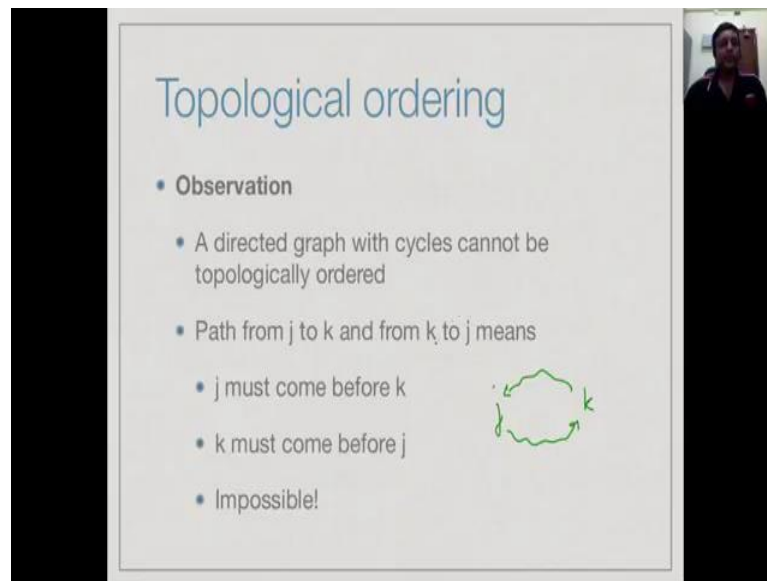


The slide is titled "Topological ordering" in blue text. It contains the following bullet points:

- Given a DAG  $G = (V, E)$ ,  $V = \{1, 2, \dots, n\}$  (The  $V$  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)



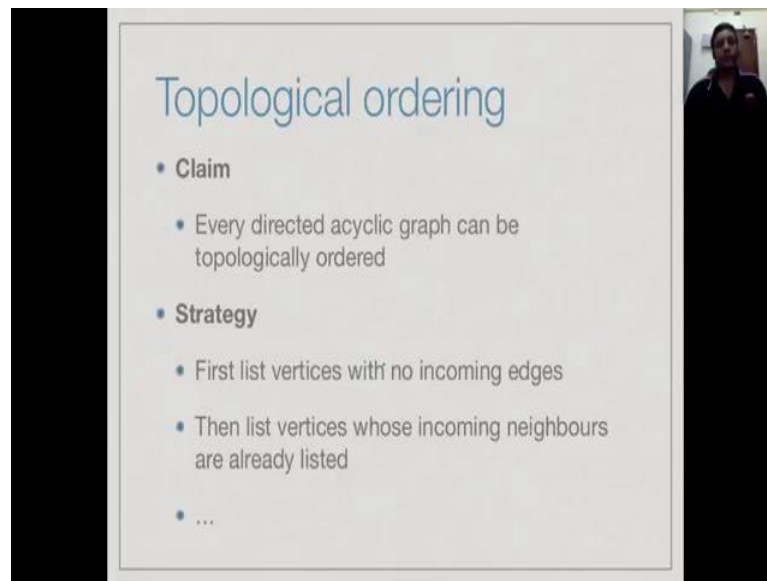
The slide is titled "Topological ordering" in a blue font. Below the title, there is a section labeled "Observation" with a blue bullet point. The text of the observation is: "A directed graph with cycles cannot be topologically ordered". Below this, there is another blue bullet point: "Path from j to k and from k to j means". This is followed by two more blue bullet points: "j must come before k" and "k must come before j". The final bullet point is "Impossible!". To the right of the text, there is a diagram of a cycle with two nodes, j and k, connected by two directed edges forming a loop.

- 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)



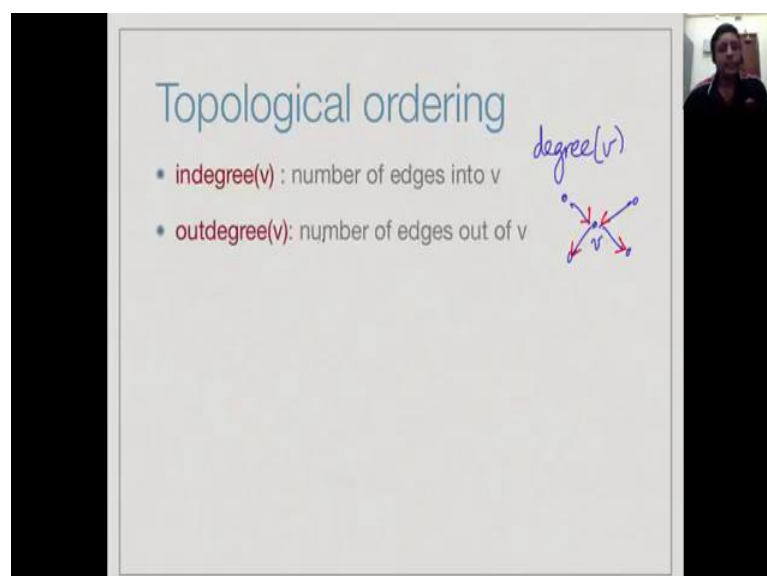
**Topological ordering**

- **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**

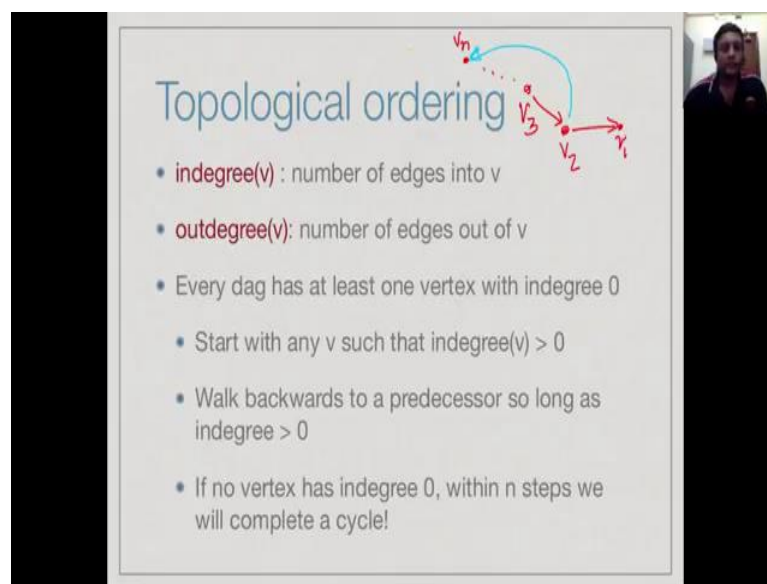
- **indegree(v)** : number of edges into v
- **outdegree(v)** : number of edges out of v

*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 said that the degree of  $v$  is 4. Now, since we have a directed graph we have a directions 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)



The slide is titled "Topological ordering" in blue text. Above the title is a diagram of a directed graph with five vertices labeled  $v_1, v_2, v_3, v_4, v_5$ . Edges are shown as arrows:  $v_4 \rightarrow v_3$  (blue),  $v_3 \rightarrow v_2$  (red),  $v_3 \rightarrow v_1$  (red), and  $v_2 \rightarrow v_1$  (red). Below the title is a list of bullet points:

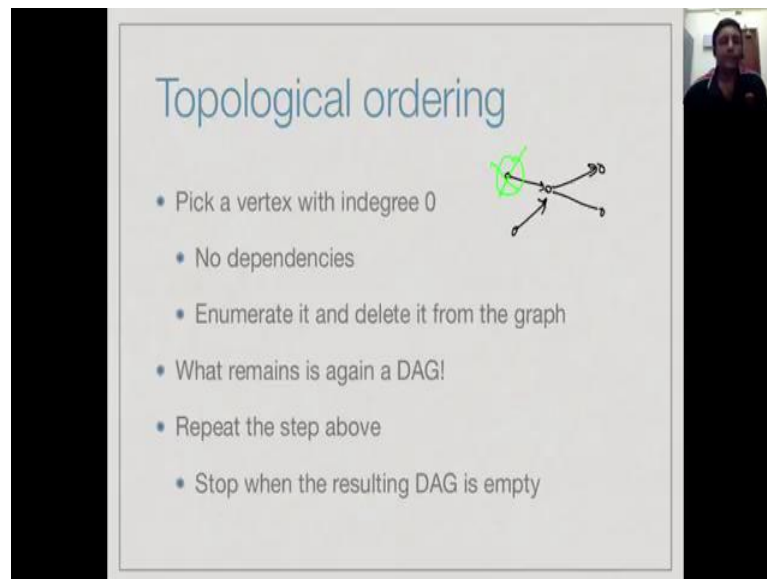
- **indegree(v)** : number of edges into  $v$
- **outdegree(v)**: number of edges out of  $v$
- Every dag has at least one vertex with indegree 0
  - Start with any  $v$  such that  $\text{indegree}(v) > 0$
  - Walk backwards to a predecessor so long as  $\text{indegree} > 0$
  - If no vertex has indegree 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 blue text. To the right of the title is a small directed graph with four nodes. The leftmost node is highlighted with a green circle and has two outgoing arrows to two other nodes, which in turn have arrows pointing to a fourth node on the far right. Below the title is a bulleted list of steps:

- 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.