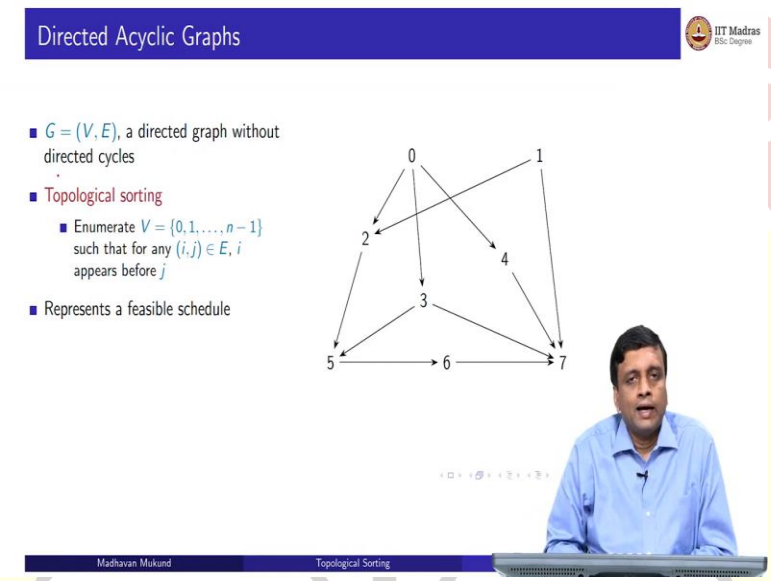
IIT Madras

ONLINE DEGREE

Programming data Structures and Algorithms using Python
Professor. Madhavan Mukund
Topological Sorting

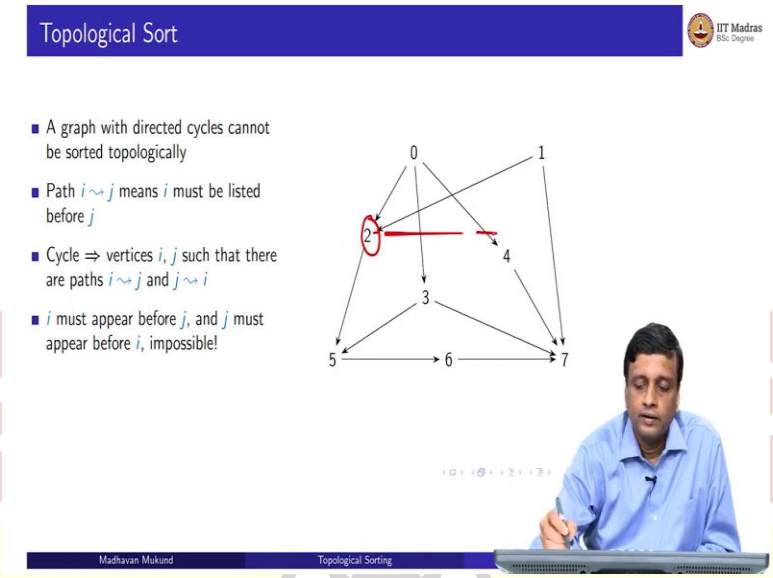
So, we were looking at the Directed Acyclic Graphs. And we said that DAGs are useful for telling us about things like tasks and their dependencies. And one of the fundamental problems that we wanted to solve using a DAG was topological sorting.

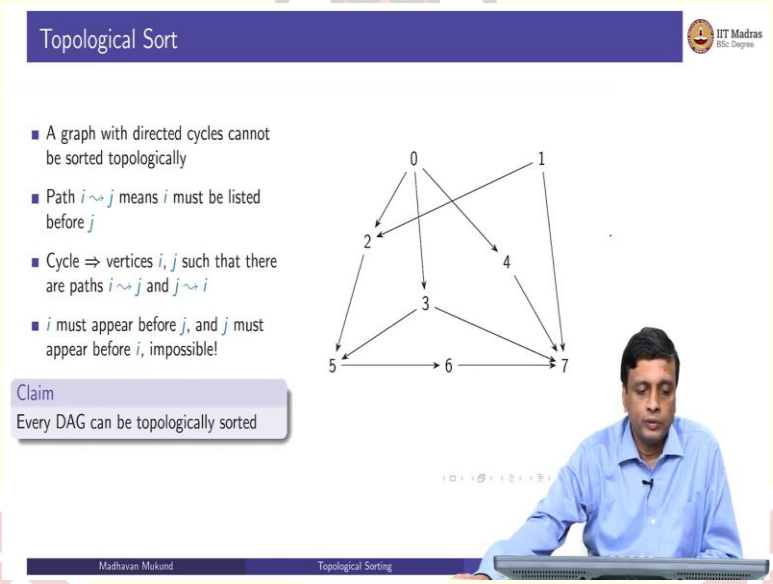
(Refer Slide Time: 00:25)



So, when we have a DAG, that is a directed graph without directed cycles, what we want is to enumerate the vertices in an order which respects the order of the edges. So, we if you think of these as tasks and dependencies, this means that we execute the tasks in such a way that every task that requires another task to be done is done after that prerequisite is completed. So, we are following the dependencies when we are executing. So, given such a set of prerequisites in the form of a directed acyclic graph, how do we find such a valid schedule? So, this is like a feasible schedule.

(Refer Slide Time: 01:03)



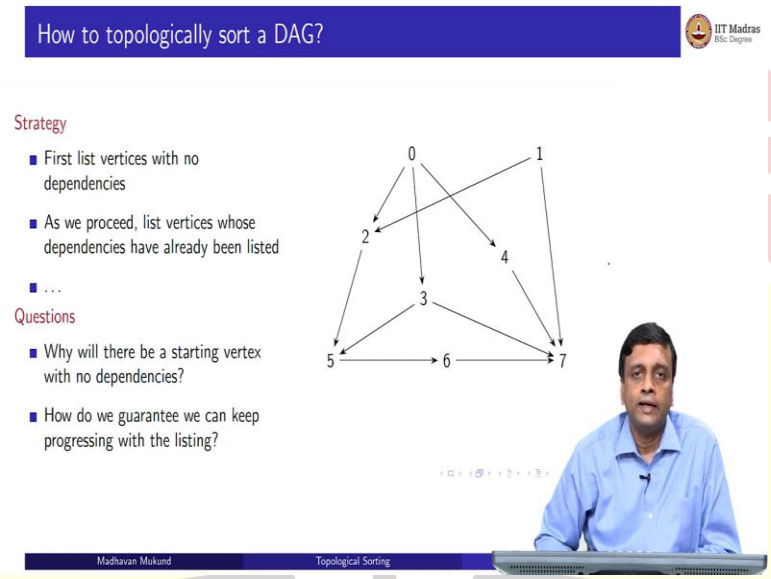


So, here is a DAG on the right for example. So, we would like to say in this DAG, for instance, that I cannot do, for example, task 3, until I have d1 both task 0 and task 1. So, this is a typical example of the kind of constraint that we get. So, it should be clear that if these constraints are what are represented by the edges, and if we have a cycle, then that means that we need something, say task 1 to be done before 2, and 2 to be done before 3, and 3 to be done before 4. And then when we complete the cycle, we will find that 4 has to be done before 1.

So, if we have a cyclic dependency in these task, then clearly there is no possible way to enumerate them or execute them in this order, which respects the dependencies. So therefore, a

graph with directed cycles cannot be sorted topologically. Now, what we are claiming is that if there are no cycles, that is if I have a directed acyclic graph, then we can always sort it topologically, we can always enumerate it in a good order such that whenever there is an edge i comma j i appears before j in the enumeration.

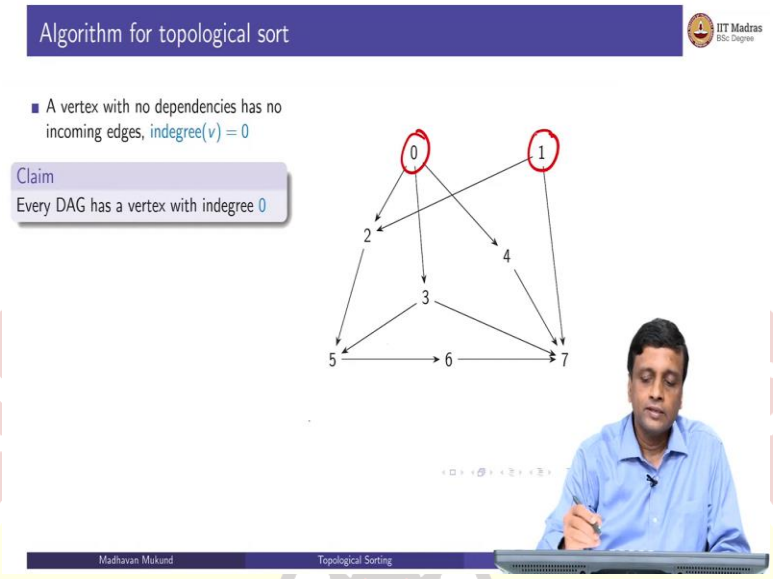
(Refer Slide Time: 02:10)

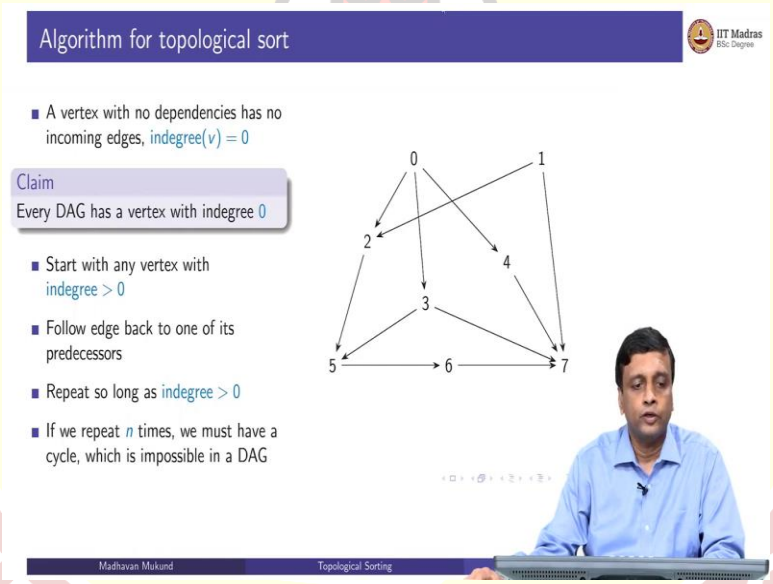


So, the strategy is straightforward. What do you begin with, well you have to begin with vertices or tasks which have no dependencies, you have to start with something that you can start a fresh. So, you begin vertices with no dependencies. And then as dependencies are completed, later tasks now become available, because all their dependencies are also empty. So, in this way, we keep exhausting the dependencies and finding new vertices to enumerate.

So, we have to answer two questions to ensure that this is always possible. First of all, we have to be able to start. So, given a DAG, there must be at least 1 vertex that we can enumerate, which does not have an incoming dependency. And secondly, as we go along, we cannot get stuck. So, every time we finish executing some task, we are left with the remaining tasks. And now we must guarantee that in the remaining tasks, they will again be at least 1 task which we can execute, which has no remaining dependencies.

(Refer Slide Time: 03:06)





So, let us remember that in degree is what we refer to in a directed graph as the number of edges coming into a vertex. So, a task which has no incoming dependencies has indegree 0. So, our first task is to find something that we can enumerate at the beginning. So, this will be a vertex which has no incoming edges, or something which has indegree 0.

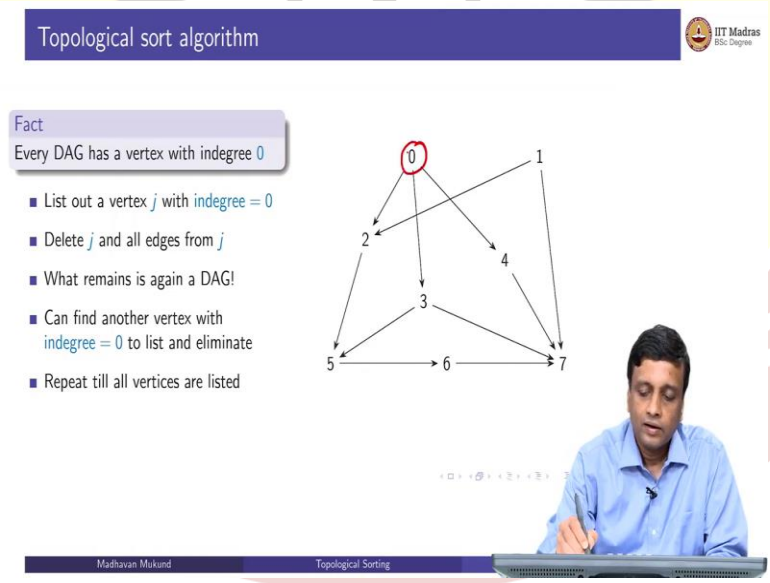
So, we claim that this is always possible. This is the first thing that we need to justify to say that every DAG can be topologically sorted, that there is always a starting point for our topological enumeration. So, every DAG must have a vertex with degree 0, indegree 0. Now there could be more than 1. So, for instance, here you can see that both 0 and 1 have this property that there are

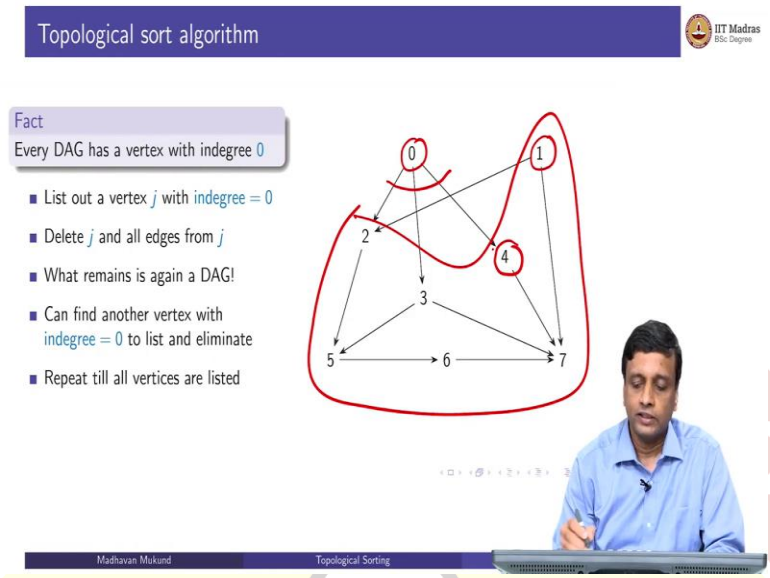
no incoming edges. So, it is not a unique 1, but there must be at least 1, this is what we are claiming.

So, why is this, well suppose there is none. Suppose every vertex actually has an incoming edge, then wherever we are in the DAG, we can go backwards through, through an incoming edge to a previous vertex. So, we go backwards, but by assumption, every vertex has at least 1 incoming edge. So, from the vertex, we just went to the second vertex also has an incoming edge, we go backwards to a third vertex to a fourth vertex, and so on. So, as we keep walking backwards, we keep walking to new vertices. Now, there are only a fixed number of vertices n in my graph, so after I make n moves, I must hit a vertex that I have seen before because I cannot see more than n distinct vertices.

So, if there is no vertex, which has in degree 0, that is, I can always walk backwards from every vertex to a previous vertex, then there must be a cycle in the graph and so it is not right. So therefore, by the converse, if it is a DAG there must be at least 1 such vertex which is indegree 0 from where we can begin our enumeration. So now we listed out right so

(Refer Slide Time: 04:57)



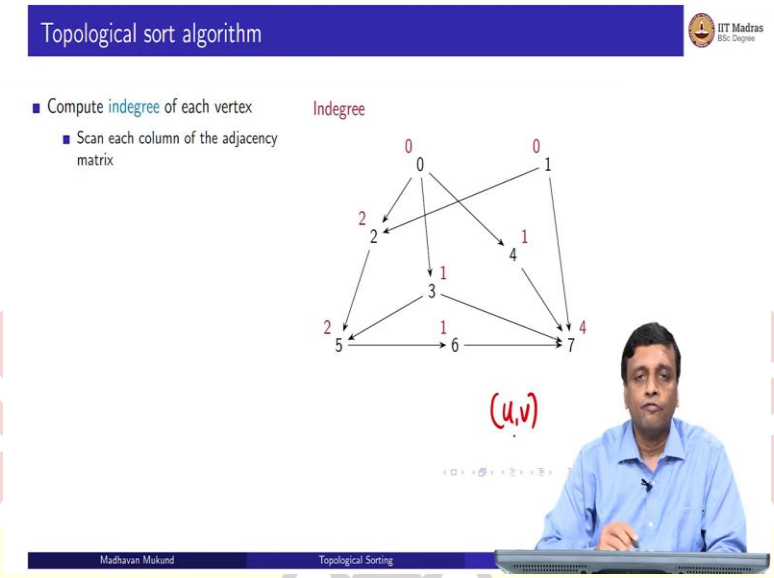


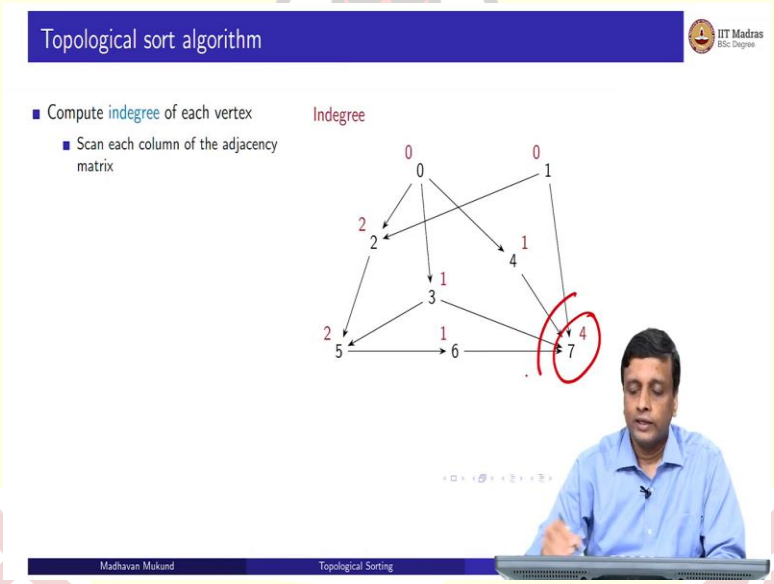
So, now we list it out, so we list out this indegree thing. So, supposing we list out this one. Then in principle this vertex has now been enumerated, so all the dependencies that it has have been satisfied, so we can delete these 3 edges. So, now what we are left with is a smaller graph in which we have removed 1 vertex and all the edges which came out of that vertex. But since we had a DAG to begin with, this new graph must also be acyclic because we have only removed edges, we have not added any edges to create cycles.

So, it is still a directed graph, it is still acyclic. So, this is again a DAG and we just showed that in every DAG, there must be some incoming indegree 0 vertex. So, in the new DAG also, there will be something that I can enumerate. So, in this case, for instance, of course, we know that we can enumerate 0 because it was already there. But we have also found, for instance, a new vertex here 4 that can be enumerated because 4 had only 1 incoming constraint that was from 0 and 0 has been enumerated.

So, in this sense, both are conditions that we required for topological sorting are satisfied, we can always start because there is always an indegree 0 vertex, I can always continue because once we remove a vertex, we still have a DAG, and therefore, by the first condition, we will again have an indegree 0 vertex and we can finish. So, this is the algorithm for topologically sorting a DAG.

(Refer Slide Time: 06:22)

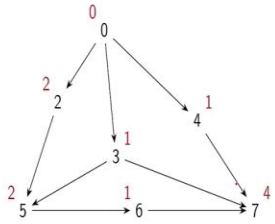




Topological sort algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG

Indegree



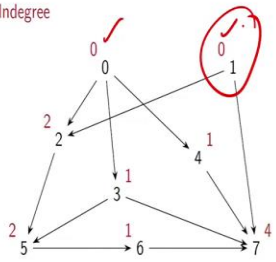
Topologically sorted sequence

1,

Topological sort algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix

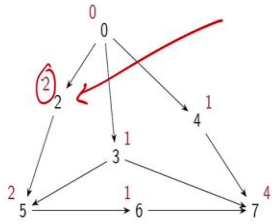
Indegree



Topological sort algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG

Indegree



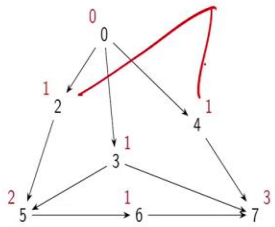
Topologically sorted sequence

1,

Topological sort algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees

Indegree



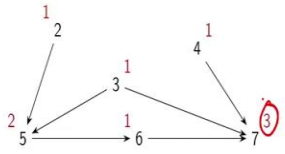
Topologically sorted sequence

1,

Topological sort algorithm

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with indegree = 0 to list and eliminate

Indegree



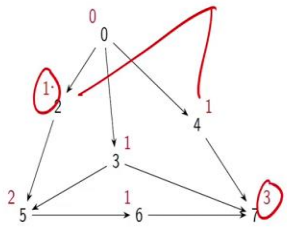
Topologically sorted sequence

1, 0,

Topological sort algorithm

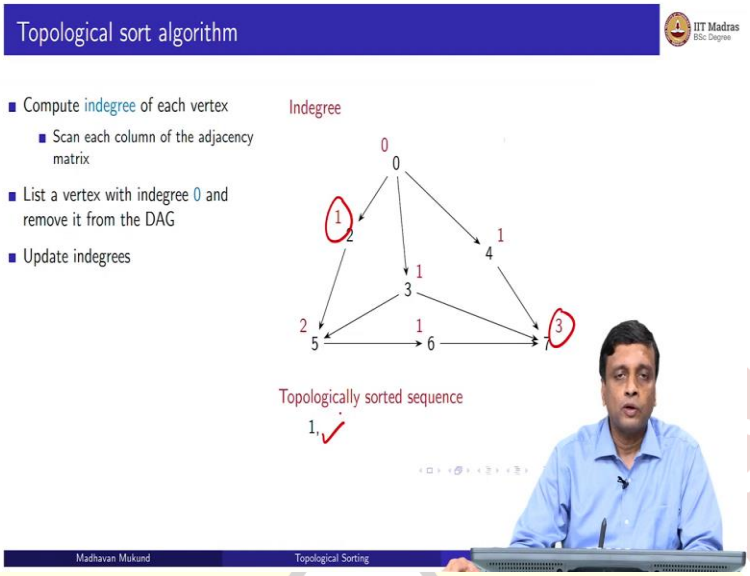
- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees

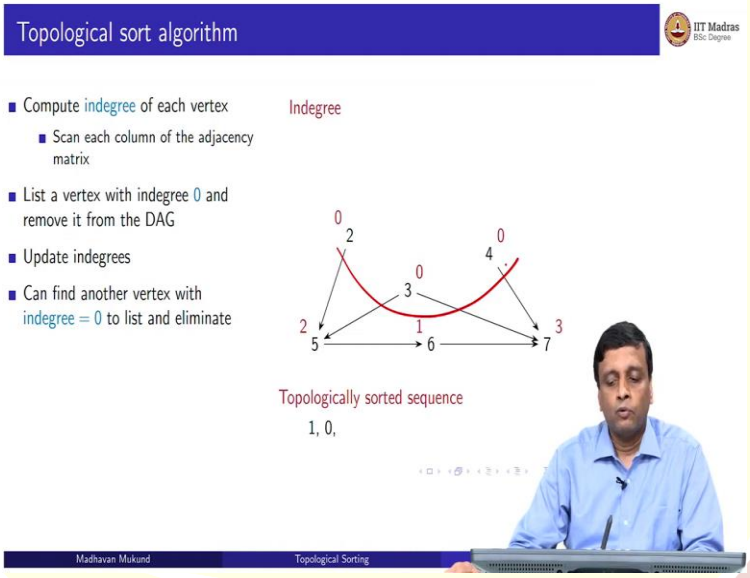
Indegree



Topologically sorted sequence

1,





So, to convert this to an algorithm, we have to first find this indegree 0 vertex. So, we have to scan our representation of the graph and compute the indegrees of each of the vertices in the graph. So, if we do that, by just looking at, for instance, in the adjacency matrix, we will look at the incoming edges, we will look at each column and count the number of edges pointing into a vertex i, if we are looking at an adjacency list, we can do it by looking at all the lists and looking for every second item in a pair, we can count the first item in the pair.

So, if you see an edge u, v, then you will increment the degree of v. So however, we do it, we can come up with this numbering, so this red numbering, which tells us the indegree of every

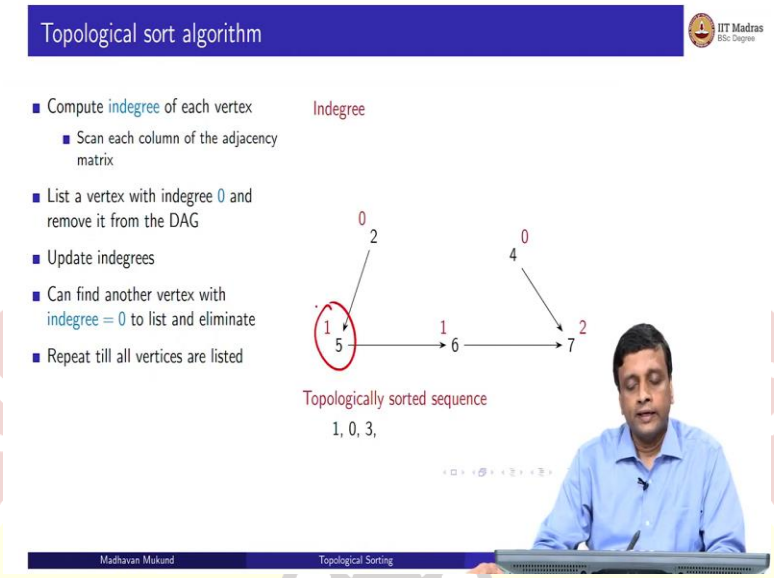
vertex in this particular DAG, so 0 and 1 have indegree 0. And as we go down the DAG, we find for instance, a vertex like 7, which has 4 incoming edges, and therefore it has degree 4.

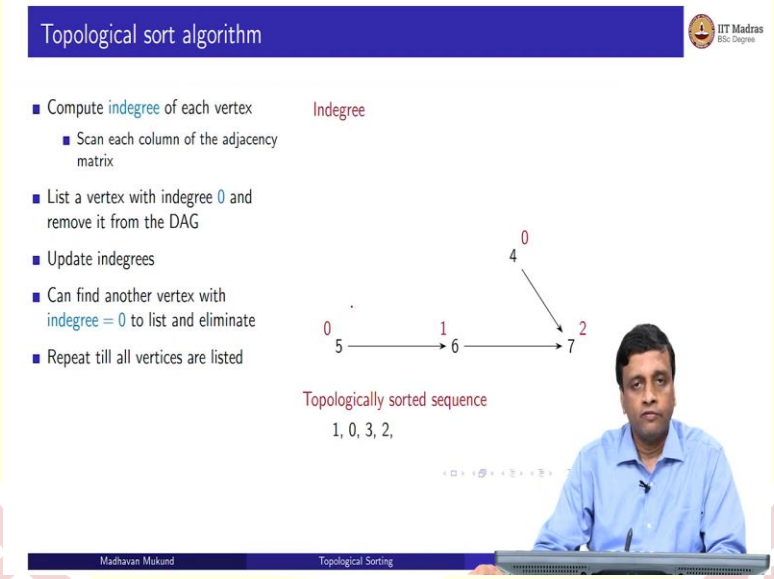
So now our algorithm says pick any vertex which has indegree 0 eliminate it, and remove it from the DAG. So in this case, let us start with 1, so we start, we could choose this or this. So, we happen to start with this, so we remove vertex 1. And now we have a new DAG, and in this new DAG, we have enumerated 1, so we can output this as the first element in our topologically sorted sequence. But we also have to correspondingly recompute the degrees. So, this had degree 2, but 1 of those 2 edges was coming from the vertex we just enumerated.

So, for every vertex, every edge, which is coming out of the vertex that we just enumerated, we have to update the degrees, so we have to update these, so we had edges like, like this. So, if you go back, so we have we had edges from 1 to 2 and 7. So now, basically, we will replace this what used to be 4 by 3. And what used to be 2 by 1, so this is a situation after enumerating 1.

So, we have been 2 things, we have enumerated 1. And we have updated the degrees of all the vertices which 1 was pointing to. So now we find another one, namely 0, for example, and we enumerate it. At this point, notice that there is only 1 vertex with indegree 0, that is the vertex 0 itself, there is no other vertex which can be currently enumerated because everything has some incoming edge. So, I enumerate 0. And now I have a graph in which I update the degrees. And I get now these 3 vertices, all 3 of these now are possible candidates for my next step.

(Refer Slide Time: 09:00)



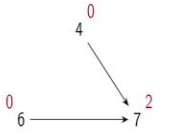


Topological sort algorithm

IIT Madras BSc Degree

- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with indegree = 0 to list and eliminate
- Repeat till all vertices are listed

Indegree



Topologically sorted sequence

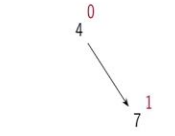
1, 0, 3, 2, 5,

Topological sort algorithm

IIT Madras BSc Degree

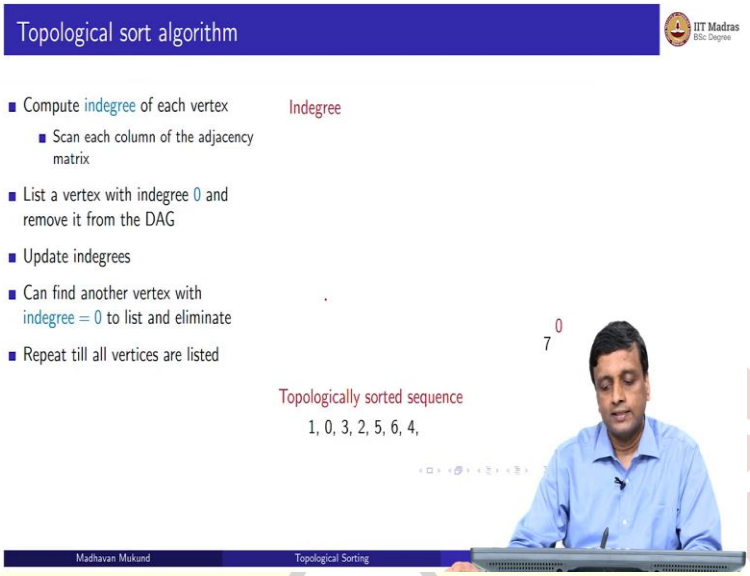
- Compute indegree of each vertex
  - Scan each column of the adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Can find another vertex with indegree = 0 to list and eliminate
- Repeat till all vertices are listed

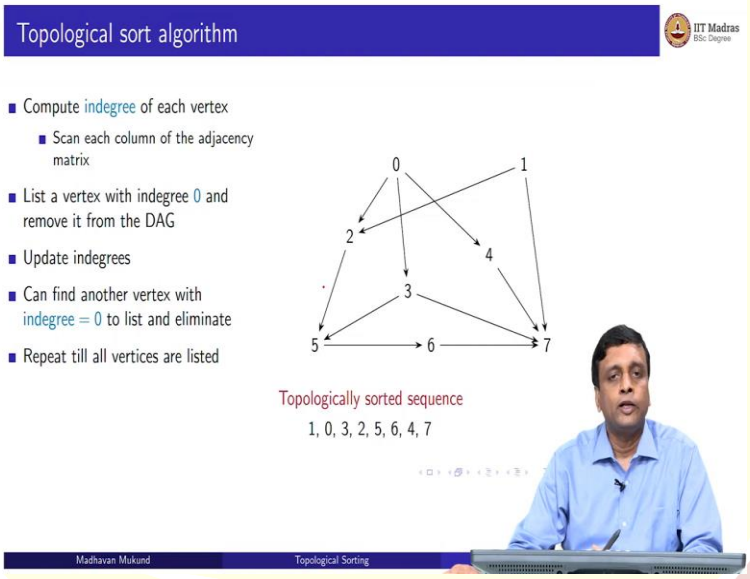
Indegree



Topologically sorted sequence

1, 0, 3, 2, 5, 6,



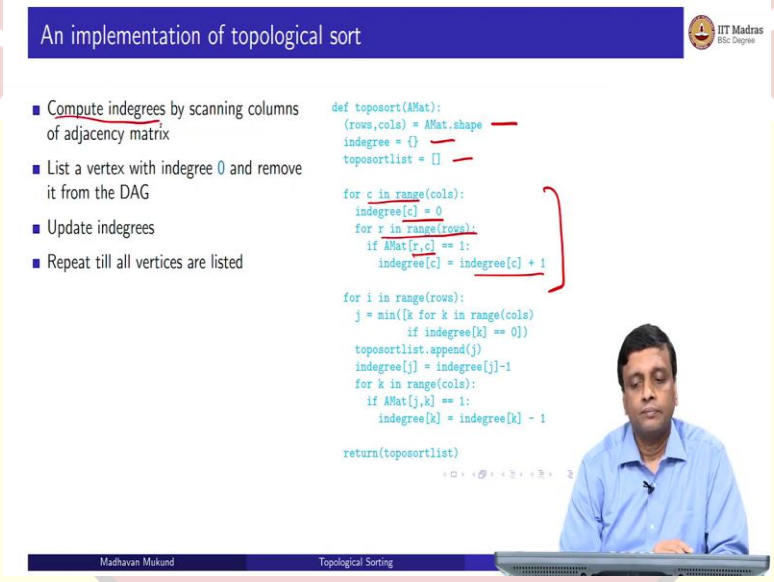


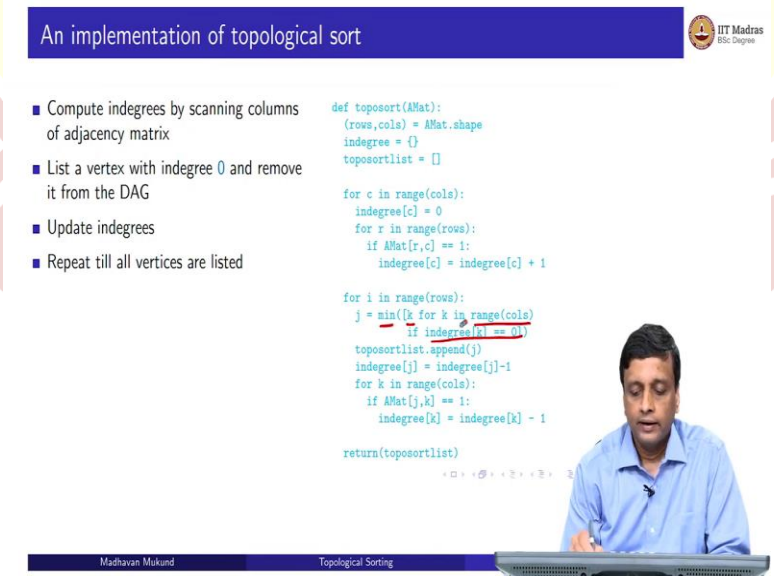
So, I can pick one of them. So maybe I picked the middle one. And now again, I enumerate, update the in degree, and now 3 was pointing into 5, so this now has degree 1. So, in this way, I can continue. So, I choose for instance 2, and then maybe I choose, so if I choose 2, then 5 becomes indegree 0s. So now I have a choice between 5 and 4.

So, then I choose 5. And then I can choose to, I have to update the indegree of 6 to 0. So now I can choose between 4 and six. So maybe I do 6. So, I leave for 4 for very late, even though it was available much earlier to enumerate. But now I have to enumerate 4 before 7 because 7 depends on 4. So, I do 4 and then 7.

So, this now is my topologically sorted sequence, one topologically sorted sequence because remember, we made some choices. Initially, we could have done 0 or 1 and we chose 1, at some point in between we had 2, 3 and 4, we could have chosen any of those and we chose 3. So, there are many situations where we had more than 1 indegree 0 vertex to choose and we picked one of them. So, the topologically sorted sequence is not unique. There are many of them, but all of them have the property that if I is enumerated before j, then if there is an edge from i to j, then i will be enumerated before j.

(Refer Slide Time: 10:06)





An implementation of topological sort

IIT Madras BSc Degree

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees ✓
- Repeat till all vertices are listed

```
def toposort(AMat):
  (rows,cols) = AMat.shape
  indegree = {}
  toposortlist = []

  for c in range(cols):
    indegree[c] = 0
    for r in range(rows):
      if AMat[r,c] == 1:
        indegree[c] = indegree[c] + 1

  for i in range(rows):
    j = min([k for k in range(cols)
          if indegree[k] == 0])
    toposortlist.append(j)
    indegree[j] = indegree[j]-1
    for k in range(cols):
      if AMat[j,k] == 1:
        indegree[k] = indegree[k] - 1

  return(toposortlist)
```

An implementation of topological sort

IIT Madras BSc Degree

- Compute indegrees by scanning columns of adjacency matrix
- List a vertex with indegree 0 and remove it from the DAG
- Update indegrees
- Repeat till all vertices are listed

Analysis

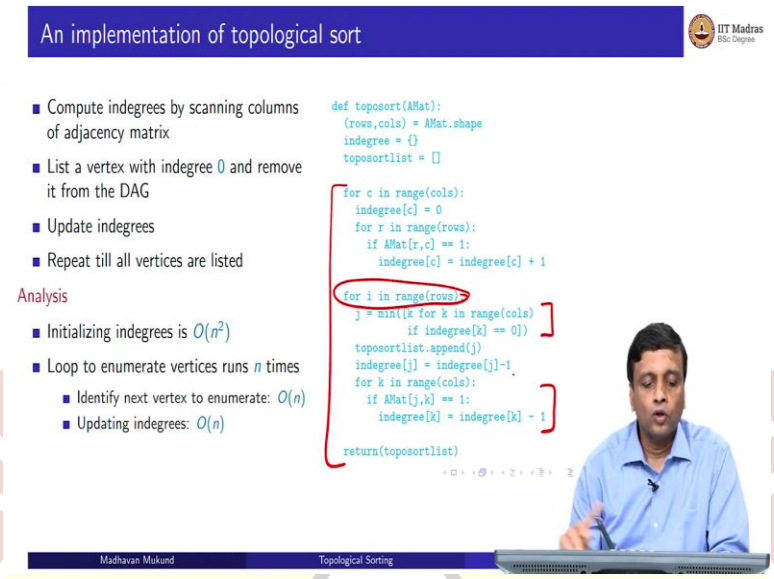
- Initializing indegrees is $O(n^2)$

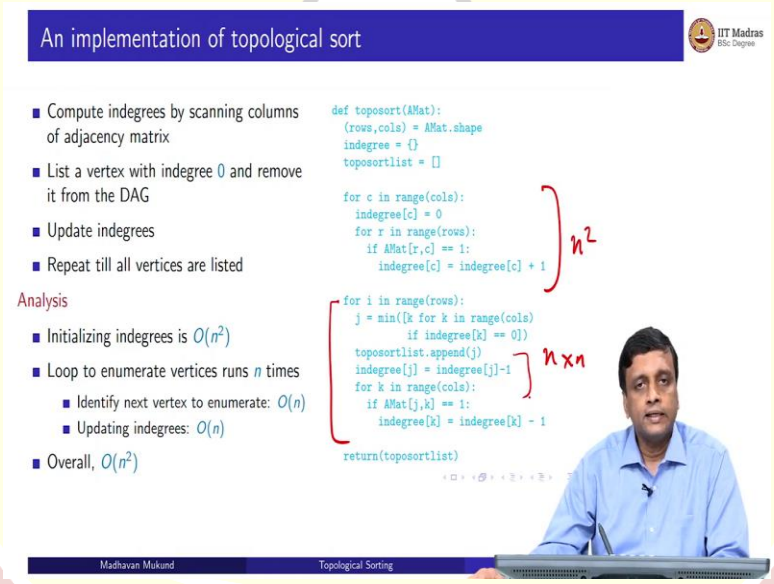

```
def toposort(AMat):
  (rows,cols) = AMat.shape
  indegree = {}
  toposortlist = []

  for c in range(cols):
    indegree[c] = 0
    for r in range(rows):
      if AMat[r,c] == 1:
        indegree[c] = indegree[c] + 1

  for i in range(rows):
    j = min([k for k in range(cols)
          if indegree[k] == 0])
    toposortlist.append(j)
    indegree[j] = indegree[j]-1
    for k in range(cols):
      if AMat[j,k] == 1:
        indegree[k] = indegree[k] - 1

  return(toposortlist)
```



So, let us look at an implementation of this. So, we have to remember, we have to compute the indegrees, then we have to find the degree with vertex with indegree 0, remove it and update indegrees and keep repeating this until all the vertices are listed. So, this is a topological sort function, which takes an adjacency matrix as input.

So here, initially, we as usual, we find out using the NumPy shape attribute, how many vertices there are, so the number of rows and number of columns, the adjacency matrix. So, now we want to keep track of indegrees. And we want to keep track of this list of vertices in the final topological thing. So, we have an empty dictionary to keep track of the industries. And we have

an empty list called topo sort list, which will accumulate the vertices in the order in which they are enumerated.

So, the first thing that we have to do is compute the indegrees. So, computing the in degrees in an adjacency matrix requires me to scan all the columns. So, what I do is I pick each column for every column, I initialize the indegree to 0, and then I walk across that all the rows in that column. And wherever I see an edge of the form r comma c, I update the indegree by 1. So, I increment by 1. So, this whole thing is, is corresponds to computing the indegrees. So, I have done that.

So, having computed the indegrees, now I can start processing these vertices. So, what I have to do is I have to find a vertex, which has n degrees 0. So, here is one way to do it, you can do it in an explicit loop or through this list comprehension. So, you find all the k in 0 to the number of vertices minus 1, such that in degree of k is 0. So, this is the list of all k for which in degrees 0. And you take, in this case, we, we said we had a choice. So, we had making now a choice to take the smallest of these vertices. So, we take the minimum over this list.

Remember that a DAG will always have such a vertex, so this list will always be not empty. So, I find this list of vertices whose indegree is currently 0, I picked the minimum 1 and call it j. So, j is going to be my next vertex to be enumerated. So, what I do is I append it to this list that I had started creating. So, this is the next vertex in my enumeration. And now I have to go around updating in degrees, so for every outgoing vertex from j, so every outgoing vertex from j, so for every k in the columns, if, if I have an edge in my adjacency matrix from j to k, then I update the indegree of k to be the indegree minus 1.

So, I am removing j from the thing. So, there is 1 less edge pointing into k. So, this is the step of updating the indegrees. And then I will keep doing this. Now I know that in every such iteration, I am going to come out with 1 more vertex. So effectively, I have an loop, which runs as many times as there are vertices in my graph.

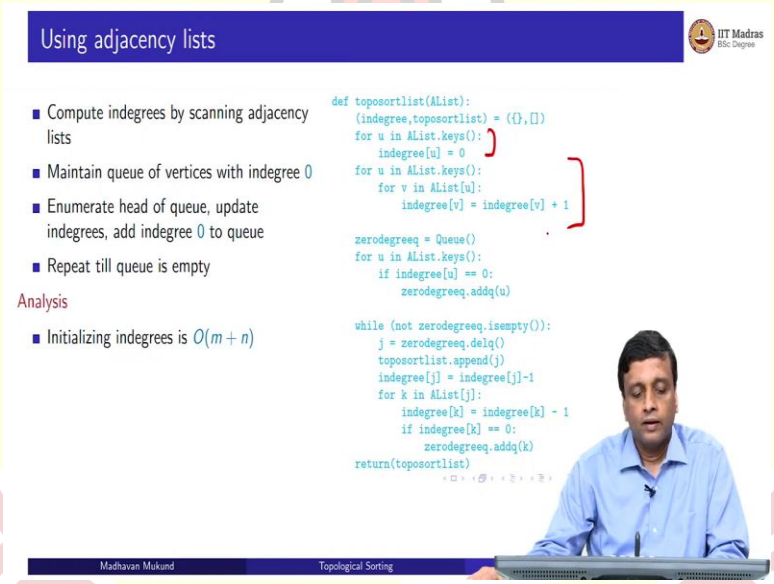
So, if you want to do an analysis, this says that this step of initializing the indegrees, because it requires me to walk down every column in my adjacency matrix, this is order n squared. And then I have to run this loop n times, so I am doing for i in range rows, which is 0 to n minus 1. And inside what am I doing, I am doing the scan. So, I am doing the scan to find the next vertex

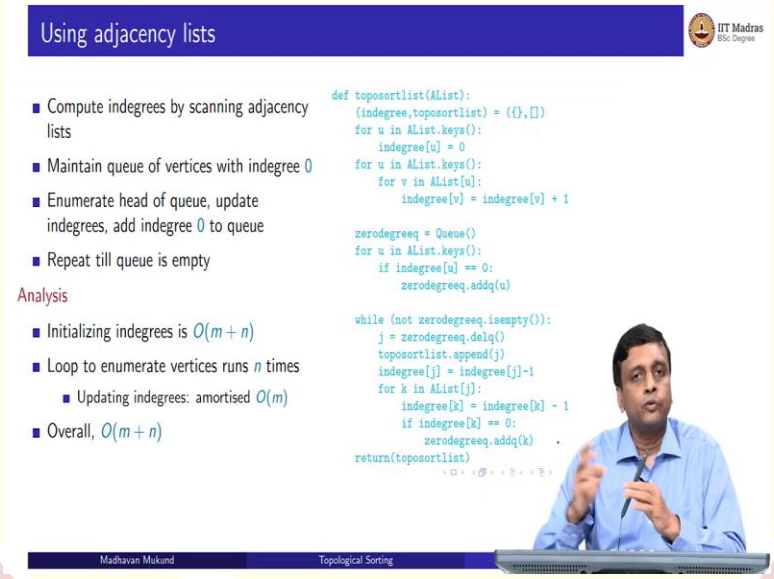
to enumerate, which is again a linear scan, I have to go through all the vertices, find out if they are indegree 0, collect them and then take the minimum.

And then finally, I have to do another order and scan here, because when I take a vertex out of my graph, I have to update all its outgoing neighbors to have 1 less degree. So essentially, what matters is that I have an order an outside loop and inside it, I am doing various order and operations, so the whole thing is going to become order n squared.

So, I have ordered n squared right up front, so this is n squared, and this is n times n because I have an outer loop which is n and I have these inner loops which are n. So, we saw before with breadth first search and depth first search that may be the way to get out of n squared is to use an adjacency list. So, let us see what happens if we use an adjacency list instead for topological sort.

(Refer Slide Time: 14:28)





So, we have a similar algorithm now, but except we are using not the adjacency matrix representation but the adjacency list. So as before, we will keep track of indegree and topological sort, as dictionary and a list respectively. And now we have to initialize, so here we can just go through all the vertices and initialize an indegree to be 0. And now we want to compute the indegrees. So, what we will say is that if I have like a list like this, so says 0 has an edge to 1, 0 has an edge to 2, 1 has edge to 3, 2 has an edge to 4 and so on.

I will just scan each of these lists. And every time I see an edge, I will update the indegree of the target of the edge. So, for every vertex, for every vertex that it is connected to every, every edge outgoing from there, I will take the indegree of that vertex and incremented. So, that is basically

the way that this works. So, this now becomes an indegree update, which processes all the edges, and not necessarily all the non edges, which is the big advantage of working with adjacency lists.

Now, we have this other problem of keeping track of the vertices which are to be enumerated because last time we had to in the adjacency matrix version, we had to keep looking for this vertex which has degree 0. So, here instead, what we will do is we will actually explicitly keep track of these the way we do in breadth first search all the vertices which have to be enumerated which are eligible to be enumerated, we will put them into a queue and we will pick them up one by one.

So, we create a queue call the 0-degree queue. So, this will hold all vertices which are having indegree 0, but which have yet to be enumerated. So, the first thing we do is we go through all the vertices, now we have to remember we have updated the indegrees. So, we know the indegrees of all the vertices. So, we go through all the vertices one more time after updating them. And every time we see a vertex with i degrees, 0, we add it to the queue. So having added it to the queue, now, what we have to do is pick up the first element of the queue and enumerate it.

So, while this queue is not empty, we delete the first element and enumerate it exactly like we did in the previous case, except there, we have to explicitly scan and find this vertex here, it is available to us immediately at the head of the queue. And having enumerated it, then we have to update the outdoing, the degrees of the outgoing vertices. So that is again, proportional to the degree of this vertex. So, for every k, which is in the list of j, I will take indegree of k and reduce it by 1.

And in this process, if I find that the indegree of k becomes 0, then I put it in the queue. So, every time I find that indegree becomes 0 by when, when it becomes 0 when I update it, so when I update the vertex is indegree at that point, I check has it become 0, if it has become 0, now it is a candidate for enumeration, put it into the queue. So, in this way, I have now eliminated the need to scan this thing for 0 degree things. And the updates that I am making are also now across the list of edges, both the indegree calculation and the indegree update.

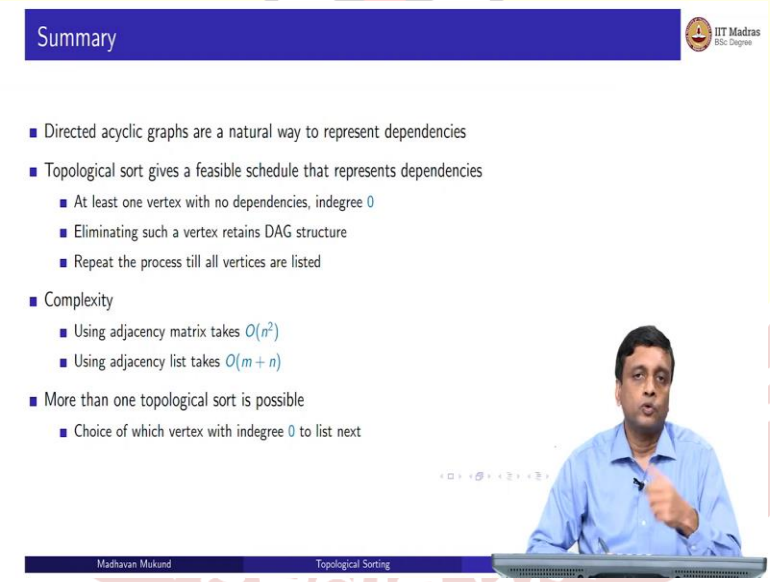
So therefore, in this analysis, we see that analyzing the indegree itself, initializing the indegrees m plus n. Why is it m plus n because I need to first spend order n time initializing it to 0. So even

if there are no edges, I have to set this up. And then I do this once for every edge, for every edge in my represented my adjacency list, I will increment in degree once. So, this is order n plus n.

Now, inside the loop, getting this vertex to enumerate is free, because I just pick it out of the queue. So, what is the complex part of the loop is updating the indegrees. But again, I am doing it in the adjacency list. So, when I take a vertex j enumerated, I am looking at all its outgoing edges. So that is proportional to degree of j. And across all the vertices, we already saw this in breadth first search, that means that I will do this, I cannot tell you how much 1 individual vertex will take. But overall, I know that it is going to be proportional to the sum of the degrees because it is going to be the sum of the number of edges, which is going to be twice the degrees.

So therefore, the sum of the degrees which is twice the number of edges, and so therefore, this whole thing is in this amortized sense going to be order n. So therefore, by moving to adjacency list representation, just like we did for BFS and DFS. In topological sort, also, we move from an n squared algorithm to an m plus n algorithm.

(Refer Slide Time: 19:06)



So, to summarize, DAGs are a natural way to represent dependencies. And what we typically need to do with a DAG is to come up with a feasible schedule, and that is what topological sort does. And we can justify the topological sort is always possible by showing that every DAG will have a vertex within degrees 0. And if we eliminate this vertex, we are left with a DAG, so we can keep repeating this until the DAG becomes empty.

So, with adjacency matrices, the naive implementation takes time n squared, but if we use an adjacency list and maintain the skew for the 0-degree vertices, which have to be enumerated, we can bring it down to order m plus n. And finally, remember that this is not a deterministic process, because there may be multiple 0-degree vertices at any given point. So, there may be more than one feasible schedule. So, what we are doing is that we are using some strategy to choose amongst these and this gives us one of them. So, based on how we choose it, we might get different schedules.

So, in our algorithm we will typically pick a uniform strategy like we pick the minimum vertex to, to bring out. The second algorithm, we actually put them out in the order in which they entered the queue. So, they actually got processed as they became 0 rather than in the, by their vertex order, and so on.