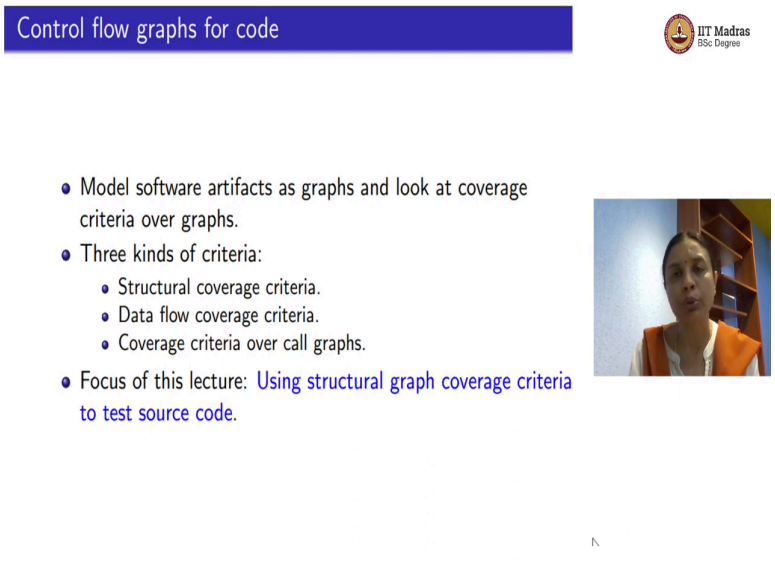
Software Testing
Professor Meenakshi D'Souza
Department of Computer Science Engineering
Indian Institute of Technology, Bangalore
Graph Coverage Criteria: Applied to Test Code

Hello, everyone. Welcome to week 3. If you remember what we did last week, we looked at graphs as data structures, learnt their basics, some properties as we need for testing, and looked at some algorithms and graphs. We ended last week by looking at some coverage criteria on graphs. Test requirement says write test cases to execute every node, to visit every edge, to cover loops, to cover edge pairs, and so on.

Those were coverage criteria based on the structure of the graph. Today, what we are going to do is take software artifacts typically code for the purpose of this lecture, and see how we can derive control flow graphs from code. Most IDEs will have support to derive control flow graphs for you. But for the purposes of these lectures, we will take snippets of code and work out control flow graphs by hand.

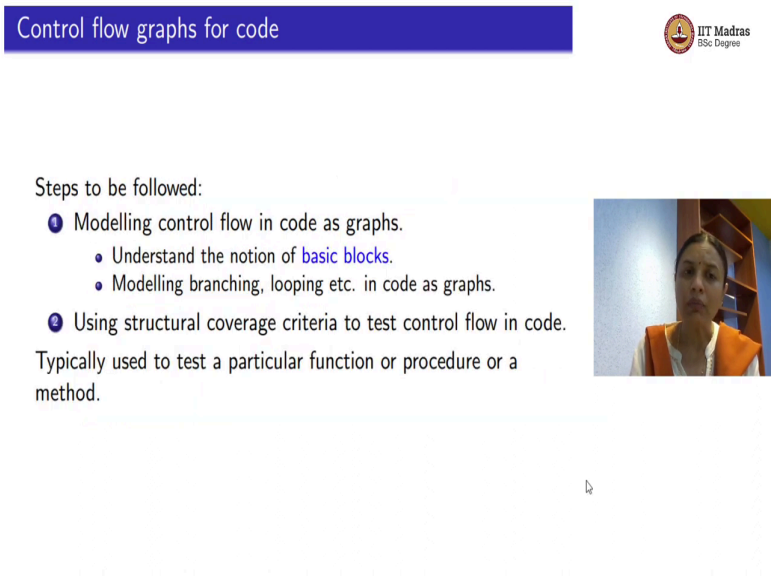
Once we have the control flow graph for the particular method or function, we will learn how to apply the structural coverage criteria that we learned last week to derive test cases for that method or function. So we are going to now learn how to apply structural graph coverage criteria to test code.

(Refer Slide Time: 01:50)



So I have already covered this. The goal of this week, this lecture is predominantly structural graph coverage criteria, post which we will go to Data Flow, call graphs and so on.

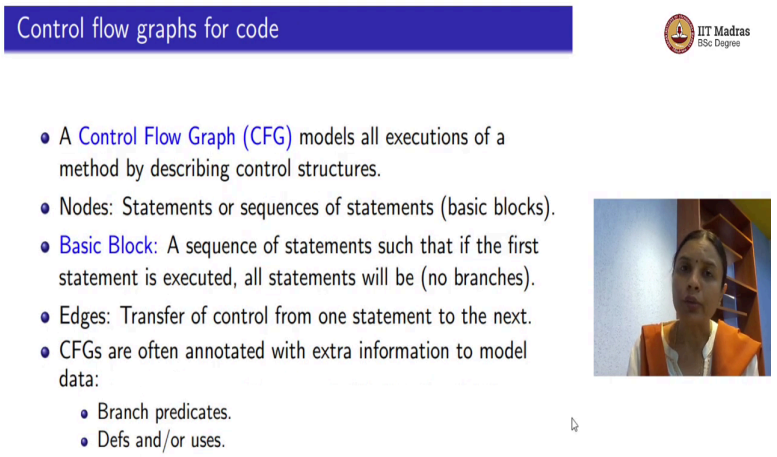
(Refer Slide Time: 02:03)



So when we look at deriving graphs from code, how do we do it? So I take a piece of code. For now, assume that the code is just a method in Java or a function or a procedure in some other programming language.

I have to model the control flow in the code as a graph and use the structural coverage criteria that we learned to derive test cases for this control flow. So this way, we are unit testing the particular function or procedure or a method by modeling it as a graph and such a unit testing would execute every statement in the method if we do structural coverage as node coverage, would execute every edge that will pass on control from one statement to the other when we do edge coverage criteria. And we will consider loops when we do prime parts on the control flow.

(Refer Slide Time: 3:05)



How to get control flow graphs from code? So what is a control flow graph? It is a graph. So it has nodes, it has edges, but it is a graph whose nodes and edges model executions of statements that belong to a particular method or a function. So the nodes or the vertices of the graph are statements or sequences of statements that are clubbed together as one basic block. It could be a contiguous sequence of statement or it could be just one statement that constitutes a basic block.

What is a basic block? It is a sequence such that the sequence is continuous, in the sense that there is no branching inside that. How do branches come as far as statements that are executable are concerned? They could come from if, they could come from while statements. So a basic block is any continuous sequence of statements that has no branching in them.

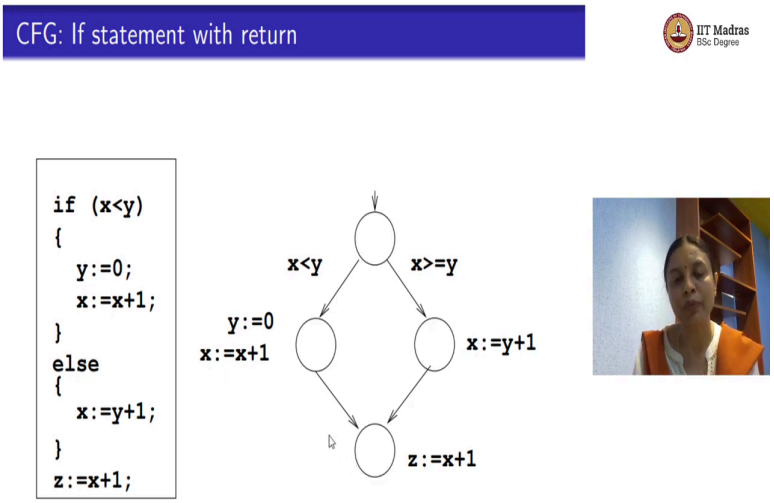
So it could be a sequence of assignment statements, it could be variable declarations followed by assignment statements, it could be a sequence of output statements. Any such continuous sequence of statements without branching is called a basic block. A node in the control flow graph could be one set statement or a basic block of statements. What are the edges of the control flow graph? Edges represent transfer of control from earlier statement to the next statement.

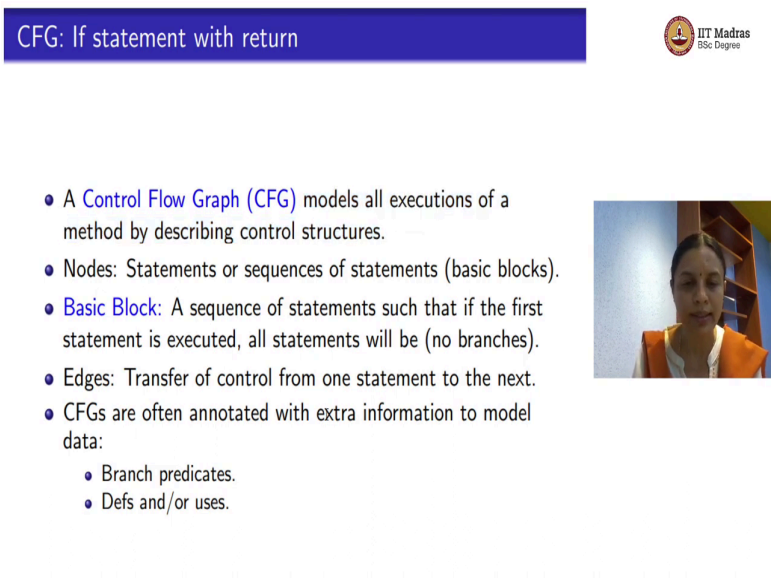
So let us say there are two consecutive statements or two consecutive basic block. Once the first basic block of statement is executed, the program control naturally passes to the second statement. This transfer of program control from the earlier statement to the subsequent statement is what we call a basic block. Edges represent, I mean sorry, is what we call an

edges represent transfer of basic control. Many times control flow graphs will not be just plain graphs with nodes, initial nodes, final nodes and edges, they will have lots of additional information, we call them as annotations.

Those additional information could represent information regarding definitions and uses of variables regarding which is the logical and relational expression that labels a particular branch and so on. We will do a few examples where we will learn how to derive control flow graphs. For this lecture, we will ignore these extra annotations and work with pure structural coverage criteria.

(Refer Slide Time: 5:33)





So from now on, over the next few slides, I am going to walk you through several examples, where on the left hand side, you see a small snippet or a fragment of code. It is just a portion

of code, not a complete piece of code. For this portion of code, the graph on the right hand side represents the control flow graph corresponding to this portion of the code. So we will study this portion and understand how the control flow graph is derived.

And by this trial and error method, by learning through examples, you will eventually learn how to write control flow graphs for methods yourself. You can also use your IDE like Eclipse or IntelliJ, or Visual Studio to get control flow graphs corresponding to code of methods or procedures. So here is the first example. On the left, you find an if statement. So it reads as if x is less than y, then you do these two statements, assign 0 to y and assign x plus 1 to x.

Else, that is if this condition is made false, if x is greater than or equal to y, then you say x is y plus 1. And then when you come out of this if statement, there is a z is equal to x plus 1. How are we going to derive the control flow graph for this if? So, corresponding to this if being here, this happens to be the first statement in this fragment of code we are considering. So this node in the graph, the topmost node that you see is the initial node of the control flow graph.

At this point, a decision happens, decision is to check whether x is less than y. If x is less than y, then the edge that goes towards the left is taken, it takes you to this node, which represents a basic block containing these two statements, y is equal to 0, and x is equal to x plus 1, which are these two statements here in this code fragment. If x happens to be greater than or equal to y, then x is equal to y plus 1 is the statement that is executed, which is labeling this node.

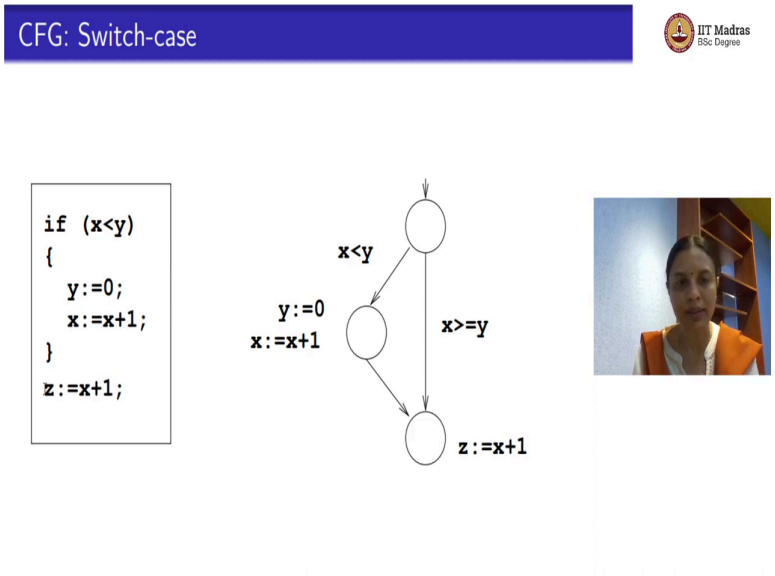
After this if gets executed, whatever happens, z is equal to x plus 1 is the statement that gets executed after this. So if you look at this code fragment on the left, when the actual execution happens, what is going to happen? If x is less than y, then these two statements are going to get executed. Otherwise, this statement is going to get executed. And after the if finishes, this statement is going to get executed.

The graph on the right pretty much represents exactly the same thing. To start with, there is a decision where a check of whether x is less than y or x is greater than or equal to y is done. If x is less than y, this node corresponding to these two statements, this node corresponding to these two statements are executed. And otherwise, this node corresponding to the statement is reached.

Whatever it is, I can do this or this, I will end up going to the last node which represents the exit from the if statement. So the code pretty much takes checks for this condition, executes one of these bunches of statements and goes to z is equal to x plus 1, control flow graph also does the same thing, starts from the initial node, based on the truth or falsity of the condition, executes this statement or this statement, and then goes to the last statement in the code.

So this is what we mean by control flow graph models all the executions of the method describing the control structure. And edges represent transfer of control from one statement to the next. So we will see a few more examples.

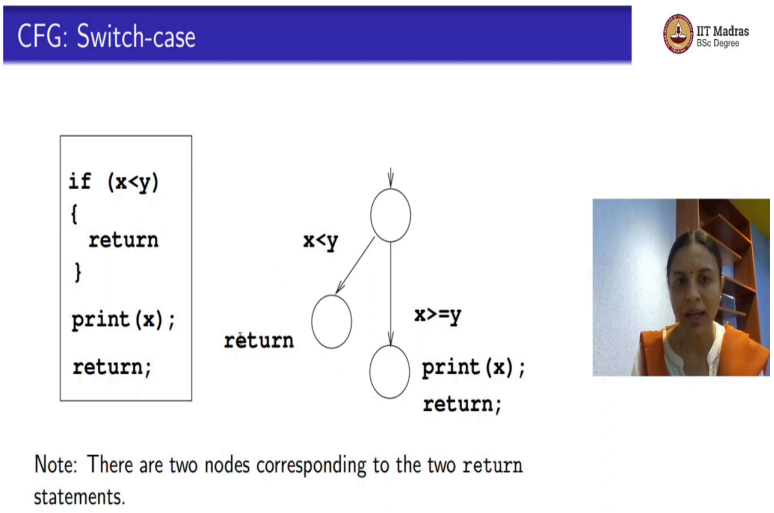
(Refer Slide Time: 9:23)



So here is an if statement. But without an else spot, the earlier one had an else. So it says if x is less than y, then do these two and come out of it. And suppose if x is greater than or equal to y, the earlier code fragment had an else This one does not have. So it says directly execute this statement z is equal to x plus 1. So that is what is happening here. If x is less than y then you go to this node, which represents this basic block of statements y equal to 0 and x equal to x plus 1.

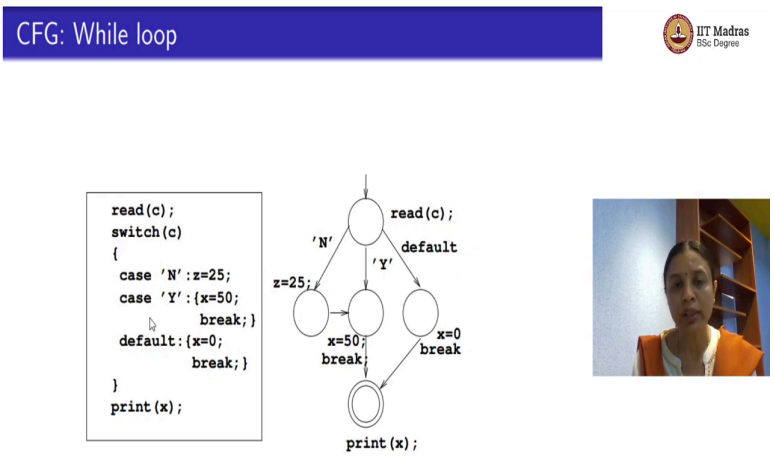
Otherwise, x is greater than or equal to y. And then you either way you end up executing this It is equal to x plus 1. So, this control flow graph on the right has two possible execution paths. When the condition x less than y is true, it comes here, executes these two nodes statements and goes here; when the condition x greater than or equal to y is true, it comes down to this part.

(Refer Slide Time: 10:18)



Here is another example. In this case, we have an if statement with a return inside it. So, if x is less than y, then it directly goes to this node which marks the return. Otherwise, if x is greater than or equal to y, that is the condition is false it goes to the print x followed by return. Please note that this return here which comes in the last line is different from this return here that comes in the last line and the control flow graph distinguishes them by putting two different nodes for these two different return statements.

(Refer Slide Time: 10:51)



So, the next is switch case statement. So, on the left you have a fragment of code that gives a small example of how a possible switch case statement will look like. So it says read a character C and based on what C is, do different-different things. In the case of C being a no,

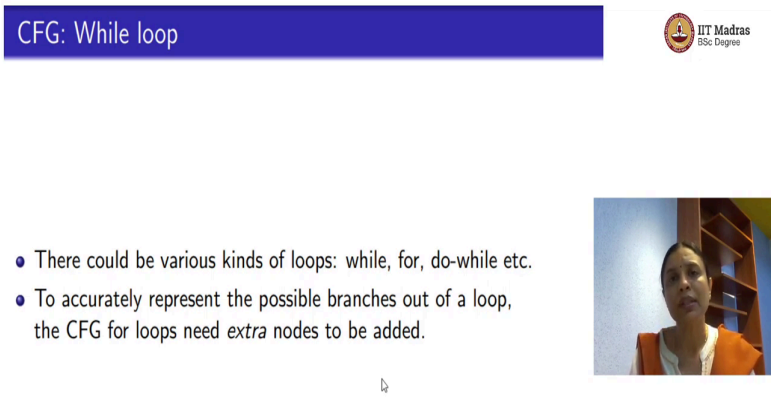
C being and maybe representing no, assign 25 to z; if C is a y, assign x to 0 and x to 50 and then break. Otherwise, do these two- assign x to 0 and then break. Then when you come out of the switch statement, do a print x.

So the control flow graph for this fragment of code is given the right. So this topmost node, the initial node represents one basic block that does both these-reading of the character C and the switch case statement based on C. And there are 3 branches that go for the 3 cases that we have: case N when C is N, case y when c is y or the case default. When the case is no the corresponding statement, note corresponding statement is there, that is this one, z is equal to 25.

When case is yes, then the next, this part is given in this node. And when the case is default, then you have this part, which is given by the last node, this part which is given by the last node. Please note that this particular control flow graph has this one extra edge that goes from this node to this node, I hope you can see my mouse. This corresponds to, because there is no break from case No, it will by default, the execution will go to case y.

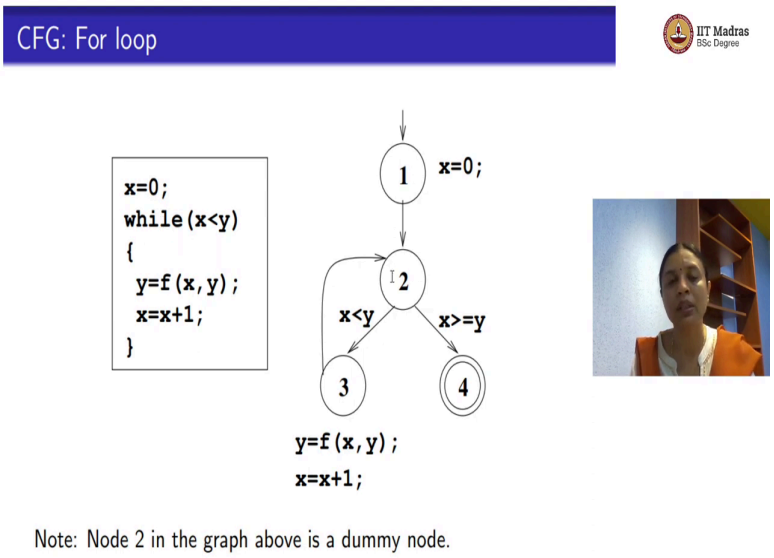
So there is this edge that represents the transfer of control from case corresponding to No to go into the case corresponding to Yes. If there was a break here, then it would directly go to this print statement. It is up to us. This particular thing there was no break, so we put this extra node. So if I had a switch case kind of statement as a part of my method, the control flow graph for that would look like this.

(Refer Slide Time: 13:04)



Now let us go on and look at loops. We all know that irrespective of the programming language we look at, there could be several different kinds of loops. Loops could be for loops, while loops, do-while loops and so on. To accurately represent loops, sometimes certain IDEs, certain semantics might put extra nodes in the control flow graph. So how do Control Flow graphs for loops look like?

(Refer Slide Time: 13:30)



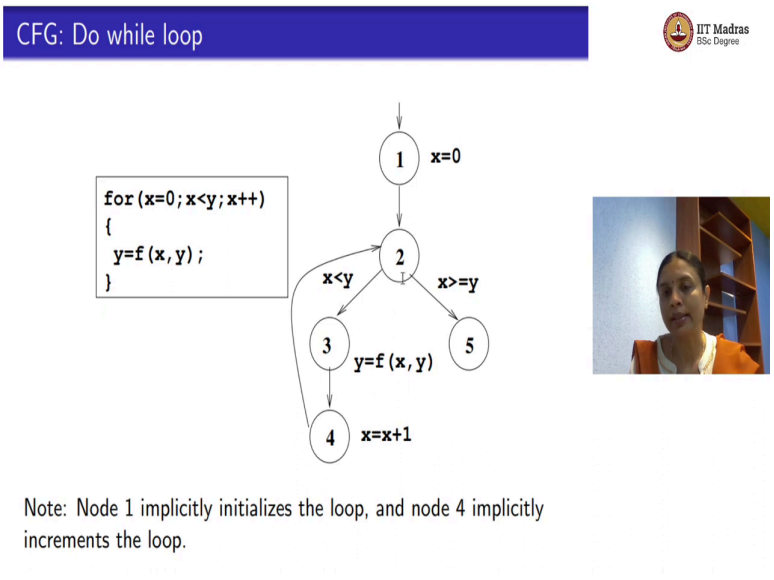
Note: Node 2 in the graph above is a dummy node.

Again, we will see several examples through which we will understand how the CFG looks like. So if I had a loop like this, here is a code fragment on the left. So let us say I start with a statement that assigns x to 0, while x is less than y, you do this function call f of xy, assign it to y, then you do x is equal to x plus 1. So how does the CFG look like? So to start with first x is equal to 0, that is this initial node, node 1, x is equal to 0.

Then for this while I could have clubbed it with node 1, but I distinguish it and put it to node 2. Why? Because when this y repeats, when this while statement repeats, I do not go back to x is equal to 0, I have to go back to checking for this while. So it is better to distinguish for me, the two statements for me. So while x is less than y, you do these two statements and node 3 which represents the body of the while loop you do these two statements at node 3 which represents the body of the while loop and then you go back to checking the condition.

In case x is greater than y, you just exit. This code fragment on the left has not specified any code to be executed after exiting the while loop, so we also do not specify any mapping to node 4. But we have to represent exit. So we do assign a final node 4 which represents exiting from the while loop and leave it without assigning any labeling.

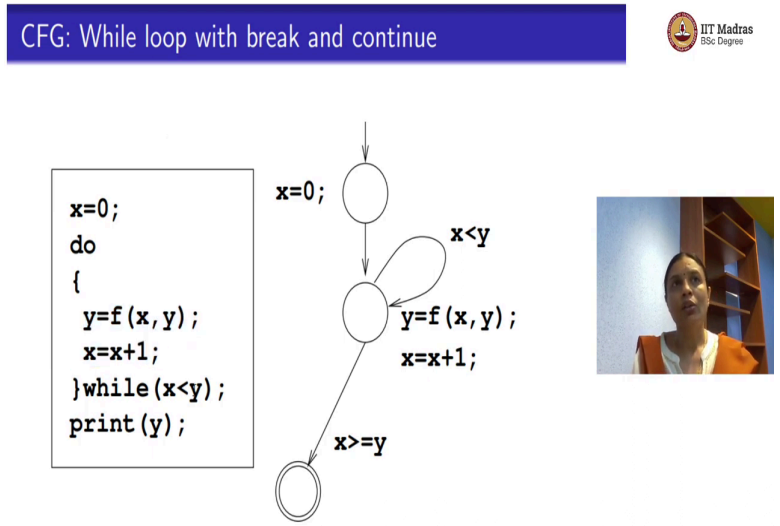
(Refer Slide Time: 14:56)



Suppose we had a for loop, here is a small for loop, how will the control will flow graph look like? How do you read this fragment containing the for loop, which is for x is equal to 0? As long as x is less than y, do x plus plus and then say y is equal to f of x comma y. So the control flow graph is given on the right. So to start with x is equal to 0, and then node 2 represents this actual for loop statement and two edges go out of node 2- one when x is less than y, in which case we enter the for loop and execute node 3, which represents this function y is equal to f of x, y.

And then increment x, x plus plus which we have given it as x is equal to x plus 1, go back to checking whether x is still less than y at node 2, exactly the way for loops works. Suppose and we repeat this process path 2 to 3 to 4, path 2 to 3 to 4 with a loop as long as x less than y is true. The minute x less than y becomes false, that is, if x is greater than or equal to y, we take 2 to 5 branch. Is this clear? So for this for loop fragment on the left, the CFG is going to look like this on the right.

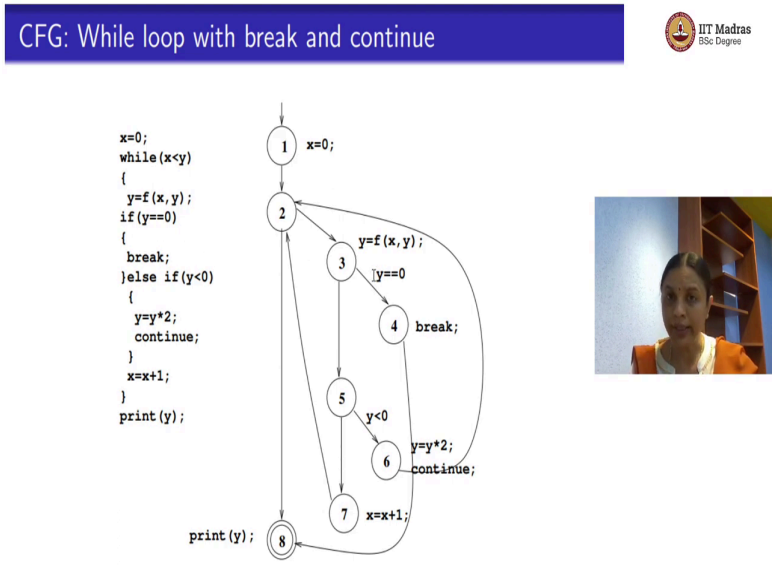
(Refer Slide Time: 16:12)



Here is another example of a do while loop. What happens in a do while loop is the body of the loop, which is these two statements in this example, gets executed first, and then the condition is checked. So the body of the loop gets executed at least once for sure. So the CFG for this fragment of code is given on the right hand side. To start with x is equal to 0 and then you have this node here, which has a self loop that goes around it. That self loop represents the checking of this condition x less than y and it also goes back to the node which executes the body of the loop.

Is this clear? So you go here, you execute the body of the loop, as long as the condition x is less than y is true, keep coming back to the node to execute the code within the loop and the minute the condition is false, that is, if x is greater than or equal to y, you exit, and you print y. I have not given the label here, but this final node has label, print y. I hope this is clear.

(Refer Slide Time: 17:16)



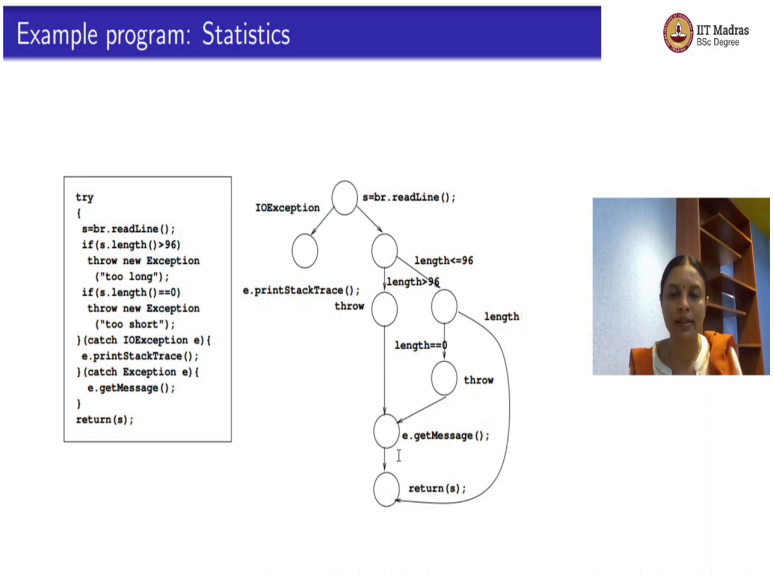
So here is a bigger example, here is a while loop with a break and continue. I hope you can read this comfortably. So let us read through the code and directly see the mapping to the control flow graph. x is less than 0, first statement, here is the first node - node 1, x is less than 0, initial node for this fragment. Then node 2 begins the while loop, first while loop. Body of the while loop just calls this condition, there is an if inside. So node 2 begins the while loop, assuming that the condition x less than y is true, it goes to node 3, where y is equal to f of x, y; f of x, y is called and the return value is assigned to y.

And it is also checked whether Y is equal to 0. So node 3 represents one statement that clubs these two basic blocks of these two statements, y is equal to f of x, y and if y is equal to 0. So suppose the if statement returns true, that is, y indeed happens to be 0, then I go to line break, which is represented by node 4 in the control flow graph. And suppose y does not happen to be 0, then I go here, this part, which is represented by node 5, where there is one more if statement, where you check for whether y is less than 0.

If it is less than 0, then you do this y is equal to y into 2 and continue, which is represented by node 6 in the CFG. Otherwise, you go and do x is equal to x plus 1, which is represented by node 7. By the way, that also marks the end of while loop. So you go back to node 2. This is the back edge to node 2. 2 to 3 to 4 to 8 represents the while and the first if returning true, and coming out, 2 to 3 to 5 to 6, and loop back represents the while and the second if returning true and continuing in the while loop.

2 to 8 represents the while statement itself returning false. I do not do anything, any of this other code, directly go from 2 to 8 which represents the print statement which is the last statement in the graph. So, is this clear? This is how I manually generate CFGs or you can use IDEs to generate CFGs corresponding to code.

(Refer Slide Time: 19:49)



So a few more examples. I am sure all of you are familiar with Java. Suppose I had a try-catch exception whose code looked like this, then the CFG for that is given on the right. So on the left, you have a code fragment, which gives you how to do exception handling using try-catch statement, and on the right, it is a CFG. So let us see what it says.

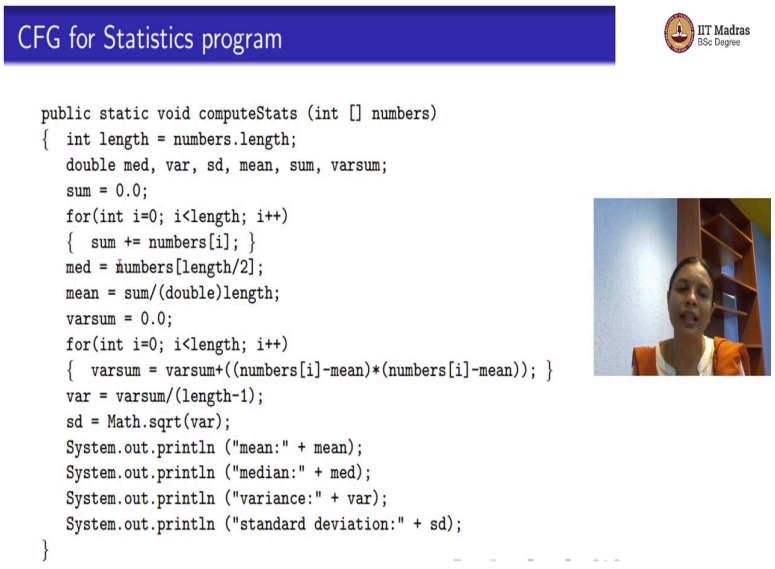
It says you read something. And if you read too much, that is too much means if it is greater than 96, then you throw an exception by printing the string too long. If you read too little, then you throw an exception by printing the string too short. Otherwise, you do this print stacktrace and do what you want. So the CFG on the right basically does that.

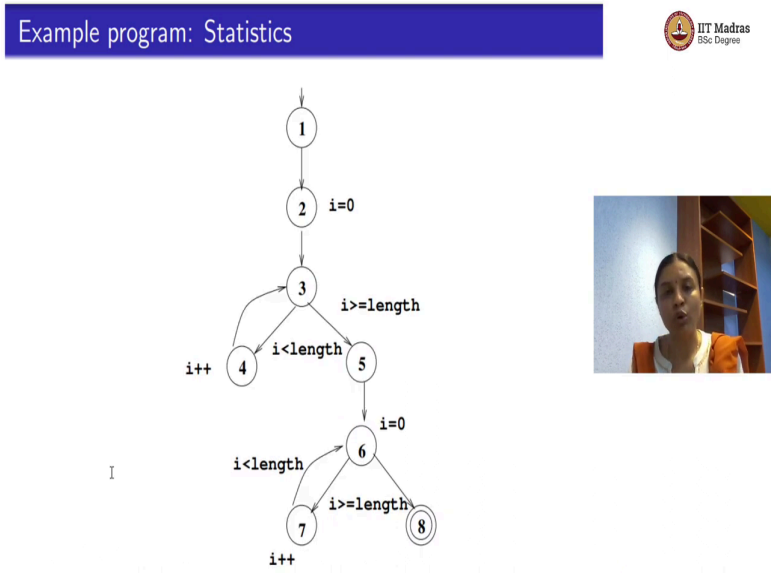
So initially, the first node represents reading of this, and this node here where my mouse is pointing to checks for the first if; based on what the length is, it checks for the second if and based on what that length is, it can throw an exception. And finally it does this get next message and return statement. Is this clear?

In case there is nothing, then it directly goes from the initial node here, from the top node here directly goes left and does this print stacktrace. It is pretty much the same thing. The ifs follow the same, meaning as we saw earlier, the whole thing is just put inside a try-catch code

fragment. So we have seen now several examples of given a particular snippet or a fragment of code, how to generate CFGs. So you can practice with a few more when you do the exercises.

(Refer Slide Time: 21:26)





What I am going to do now is give you a full program and it is a small program, small enough to fit into the slide, we will generate the control flow graph using the technique that we just learned through examples, and apply structural graph coverage criteria on that. So here is a small program written in Java. So it is a program that computes various statistical parameters. So it takes an array of numbers of a particular length, and then computes these values- median, variance, standard deviation, mean, sum, variance sum.

So the code has 2 for loops here. There is one for loop here that computes the sum, and that is used to compute the median mean. And the second for loop computes the varsum, which is used to compute the variance and the standard deviation. And finally, the code has a print statements that will print out all these values. So I hope the structure is clear. So it initializes the variables, all these median, variance, standard deviation, mean, sum, variance sum, it first says what is going to be the length of your array called numbers, and then it initializes sum to be 0, starts a for loop that runs from 0 to the length and computes the sum of all the numbers in the array.

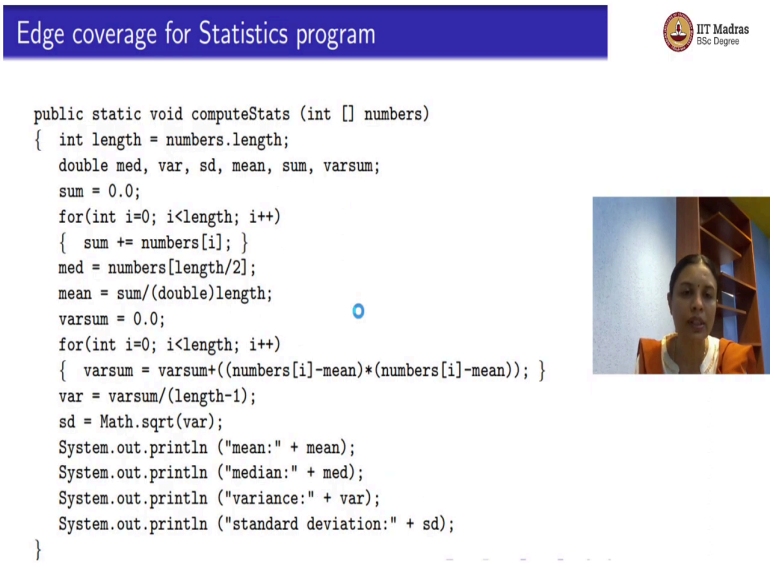
Then it says median is nothing but this formula, mean is nothing but the sum divided by the number of elements. Now it goes into the next for loop. For that it initializes a variable called varsum to 0, and it computes varsum using the standard statistics formula. Once it exits the for loop, it says variance is varsum divided by length minus 1 and standard deviation is the square root of variance.

Then it has these print statements where it prints these values. So here is a small method called compute stats that has this function, takes an array of numbers of a particular length, computes the various statistical parameters and prints them out for you. We are going to derive the control flow graph.

I have not been able to put the code and the control flow graph side by side in one slide. But this is the control flow graph corresponding to this code. You roughly remember the structure of this code. There is this variable initialization, there is one for loop, there are some computations that happen- mean median. Then there is a second for loop, some more computations and printf statements. So that is exactly what the CFG has.

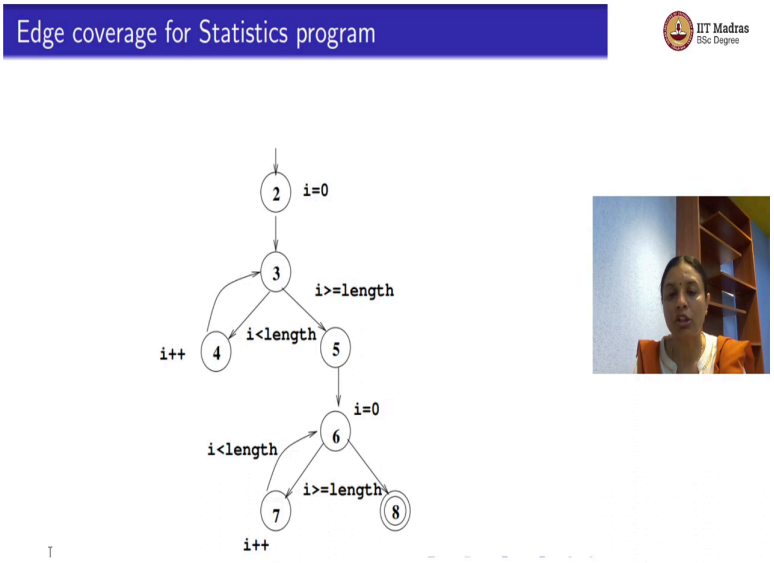
So the near node 1 represents a basic block that has all those initializations node 2, 3, 4 represents repeated iterations of the for loop. This one, it starts the for loop by setting is equal to 0, 3 to 4 to 3 to 4 represents repeated iteration of the for loop. When I compute finishing the sum in the first for loop, I go to node 5, then to 6 to 7 to 6 to 7, which represents the second for loop and printing of all the other values.

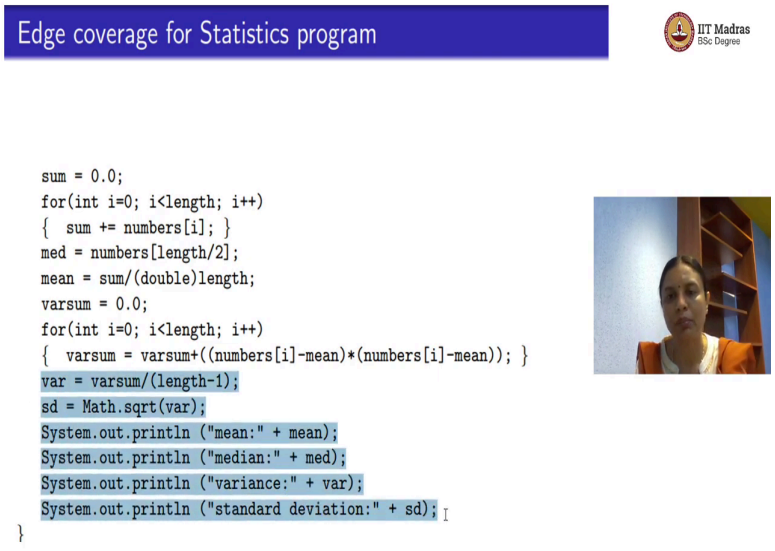
(Refer Slide Time: 24:34)



This 5 is a basic block of statements that represents all these. It represents computing all this median mean, initializing varsum and beginning the second for loop.

(Refer Slide Time: 24:42)

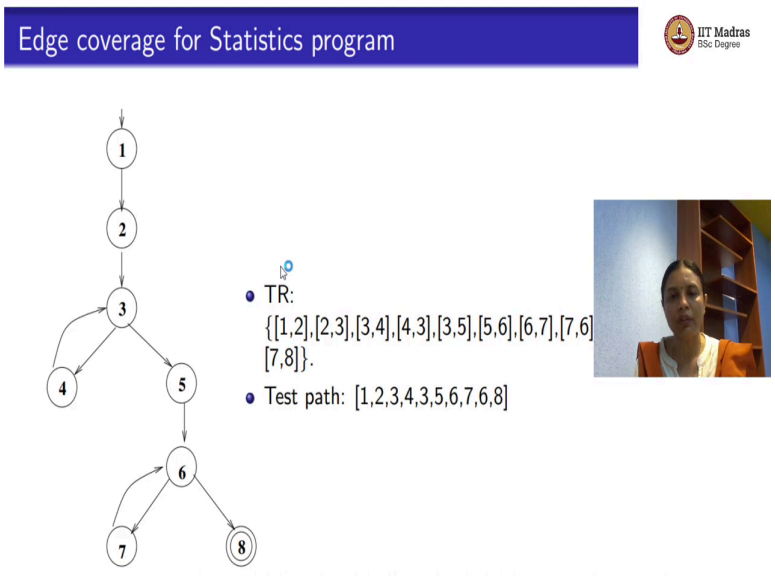




And then this structure. 5 to 6 to 7 to 6 to 7 represents iterations of the for loop and when I exit the second for loop, I go to node 8, which represents another basic block of this part. I compute variance standard deviation and do all the print statements.

Sometimes you could keep them as separate nodes- node 8, 9, 10, one for each statement. But pretty much that is not the standard practice that is followed and CFGs, we like to club together a group of statements that occur without any branching between them into one node in the CFG. So node 8 and node 5, represent basic blocks have statements, which are basically a contiguous sequence of statements in that example. Now, here is the CFG. Let us take the CFG and apply the control flow criteria that we learned.

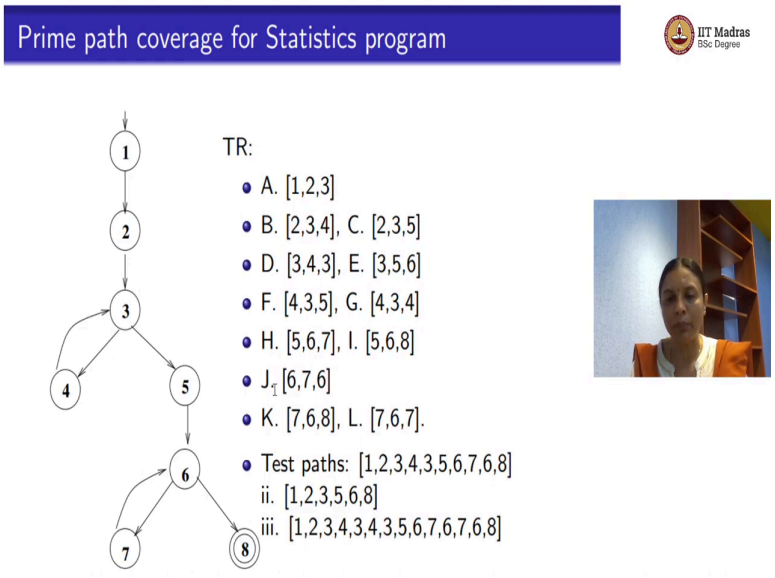
(Refer Slide Time: 25:30)



So simplest control flow criteria that we learned was node coverage and edge coverage. In this example, I have shown you how edge coverage looks like. So this is the replica of the CFG here. I have removed the annotations and just kept the graph. So this is the graph, we are only bothered about structural coverage criteria, we are not going to remember the predicates that label the for loops and all.

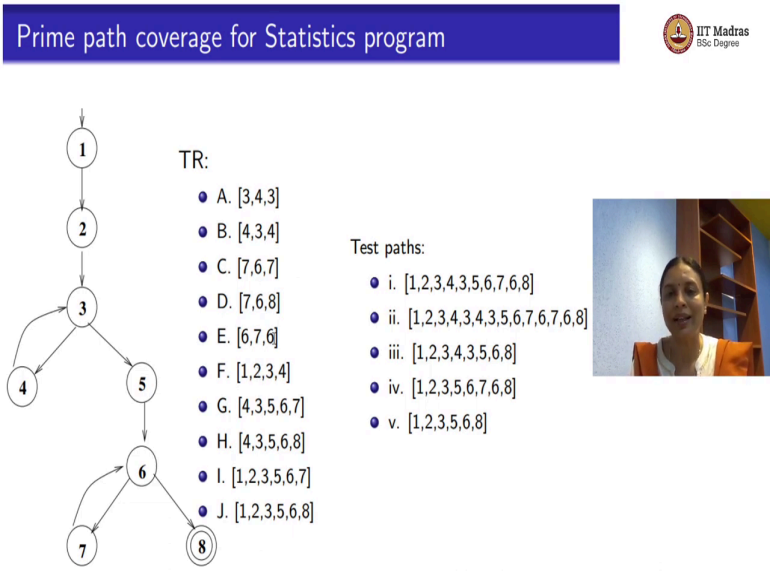
So we just take this graph criteria. Now it is like a good old graph, it has 8 nodes, node 1 is the initial node, node 8 is the final node, and it has all these edges between them. So if I have to achieve edge coverage for statistics program, test requirement is all the edges, how many edges are there? 1 to 2, 2 to 3, 3 to 4, 4 to 3, back and so on, this is just listed here. What would be a test path? One exhaustive test path is enough. 1 to 2 to 3 to 4, back to 3, to 5, to 6, to 7, back to 6, and then to 8. This is one test path.

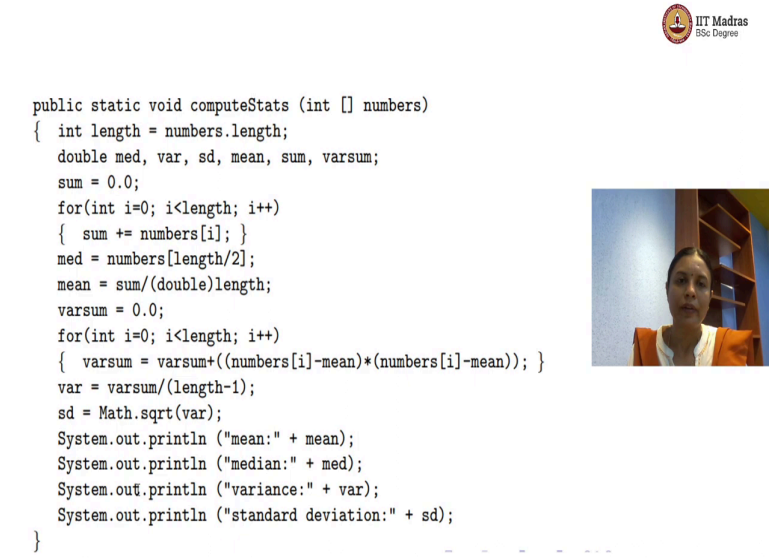
(Refer Slide Time: 26:35)



So now let us take the same CFG, control flow graph for the statistics program and apply edge pair coverage. So the test requirement is pairs of edges, that is paths of length at most 2, which includes edges and vertices. For convenience sake, as we did last time, we just list paths of length 2, these are the paths of length 2, this is the test requirement. And the test paths to satisfy these 3 requirements are given down below here, you can verify that these are indeed the right kind of test paths.

(Refer Slide Time: 27:07)





Now prime path coverage. I have applied the algorithm to compute prime paths that we taught last week, and listed all the prime paths here. It will be good for you to pause here and check whether these prime parts are indeed correct, which prime parts correspond to which loops and all that. See, for example, A, B, and then F correspond to 1 2 3 4, repeated iterations of the 3 4 loop. Similarly, C and E, along with this 1 2 3, 1 2 4 3 5 6 7 correspond to the second for loop, and D represents exiting the second for loop, J also represents keeping both the for loops.

So all the prime paths here, 10 of them correspond to executions of this control flow graph, where the for loops are visited and skipped. There are 2 for loops, both of them are visited, both of them are skipped. And for this, we have 5 test paths that we can derive to cover this

prime paths. When you do prime path coverage, you will realize that this example program has a small error.

Maybe you realized it even without that, but it has a small error, there is a risk of division by 0 here when the second for loop is skipped. And prime path coverage will find that error. This path here 1 2 3 5 6 8, will run into risk of the method doing division by 0, which is an unacceptable error in programming. So I hope this exercise helped you to understand how to derive control flow graphs of methods and how to apply simple structural coverage criteria that we learnt to derive useful test cases.

It will be a good idea to take your IDE like Eclipse, take a method, small method, maybe you could code the same method that is given here, the simple statistics program, generate the control flow graph and see if you can use J unit to execute the test cases. So I will stop here for now. We will continue in the next class. Thank you.