

Chapter Four

Structural (White Box) Testing

- At the end of this chapter , the student will be able to achieve the following Objectives.
 - structural testing
 - Structure based Techniques
 - ✓ Statement testing
 - ✓ Decision testing
 - ✓ branch testing
 - ✓ Condition testing
 - ✓ Path testing
 - ✓ Data flow testing

What is Structural Testing?

- **Structural testing** is a type of software testing that uses the **internal design** of the software for testing.
- **Structural testing** is related to the **internal design** and **implementation of the software** i.e.
 - It involves the development team members in the testing team.
 - It tests different aspects of the software according to its types.

Structure-Based Techniques

- The structural test case design techniques are used to design test cases based on an analysis of the internal structure of the component or system.
- These techniques are also known as white-box tests.
- Traditionally the internal structure has been interpreted as the structure of the code.
 - These focus on the testing of code and they are primarily used for component testing and low-level integration testing.
- In newer testing literature, structural testing is also applied to architecture where the structure may be a call tree, a menu structure, or a Webpage structure.

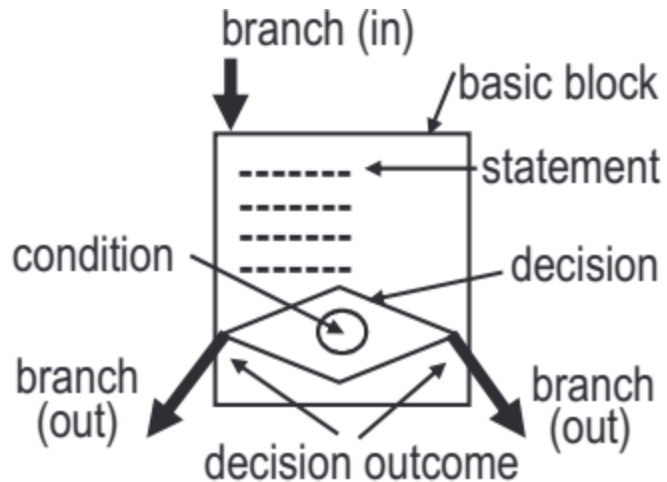
Structure-Based Techniques

- The structural test case design techniques covered here are:
 - Statement testing
 - Decision testing/branch testing
 - Condition testing
 - Multiple condition testing
 - Condition determination testing
 - loop testing
 - Path testing
 - (Interunit testing—not part of the ISTQB syllabus)
- The test case design techniques in this category all require that the tester understands the structure (i.e., has some knowledge of the coding language).

Structure-Based Techniques

- Most structural testing of code is performed by **programmers**.
- Structural testing is very often **supported by tools**.

White-Box Concepts



- An executable statement is defined as a non comment or nonwhite space entity.
 - In a programming language, the smallest indivisible unit of execution.
- A group of statements always executed is called a basic block.
- A basic block can consist of only one statement or more statements .

White-Box Concepts Cont.

- The last statement in a basic block leads to another building block, or stops the execution of the component (e.g., return) or the entire software system (end).
- when a basic block ends in a decision, the further flow depends on the outcome of the decision.
- A decision statement is also called a branch point.
- The branches out of a basic block are connected to the outcomes of the decision also called decision outcome or branch outcome.
- Most decisions have two outcomes (True or False), but some have more, for example Case statements.
- A condition is a logical expression that can be evaluated to either True or False.
- A decision may consist of one simple condition, or a number of combined conditions.

Statement Testing

- Statement coverage is one of the widely used software testing. It comes under **white box testing**.
- Statement coverage technique is used to design white box test cases.
 - This technique involves execution of all statements of the source code **at least once**.
 - It is used to **calculate the total number of executed statements** in the source code out **of total statements present in the source code**.
- Statement coverage derives scenario of test cases under **the white box testing process** which is based upon the structure of the code.

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

Statement Testing Cont.

- In the internal source code, there is a wide variety of elements like operators, methods, arrays, looping, control statements, exception handlers, etc.
- Based on the input given to the program, some code statements are executed and some may not be executed.
- The goal of statement coverage technique is to cover all the possible executing statements and path lines in the code.

Statement Testing Cont.

■ Example 1

```
print (int a, int b) {  
    int sum = a+b;  
    if (sum>0)  
        print ("This is a positive result")  
    else  
        print ("This is negative result")  
}
```

■ Source Code Structure:

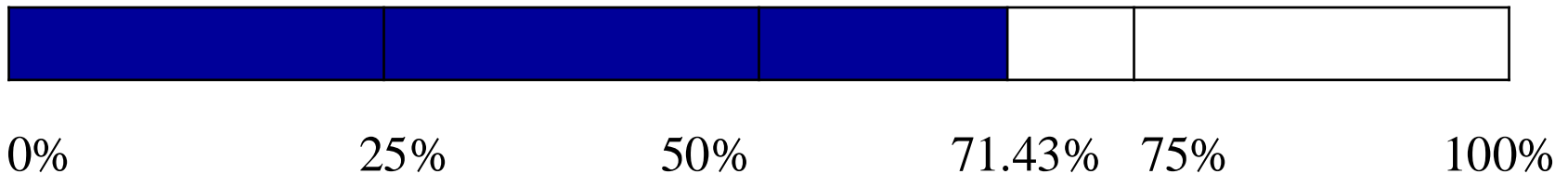
- Find the sum of these two values.
- If the sum is greater than 0, then print "This is the positive result."
- If the sum is less than 0, then print "This is the negative result."

Statement Testing Cont.

- If $a = 5$, $b = 4$
- ✓ we can see the value of sum will be 9 that is greater than 0 and as per the condition result will be **"This is a positive result."**
- ✓ To calculate statement coverage ,take
 - total number of statements= 7
 - number of executed statements = 5.

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

■ statement coverage $= 5/7 * 100$
 $= 71.43\%$



Statement Testing Cont.

- If $A = -2$, $B = -7$
- we can see the value of sum will be -9 that is less than 0 and as per the condition result will be **"This is a negative result."**
- ✓ To calculate statement coverage, take total number of statements are 7 and number of executed statements are 6 .

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

- statement coverage $= 6/7 * 100$
 $= 85.71\%$



0% 25% 50% 75% 85.71% 100%

- we can see all the statements are covered in both scenario and we can consider that the overall statement coverage is 100% .



Decision Coverage Testing



- At 100% coverage **branch coverage** and **decision coverage** give identical results.
- The **result of a decision** determines **the control flow** alternative taken.
- To define test cases for decision testing we have to:
 - Divide the code into basic blocks,
 - Identify the **decisions** (and hence the **decision outcomes** and the **branches**)
 - Design test cases to cover **the decision outcomes or branches**

Decision Coverage Testing Cont.

- Decision coverage technique comes under white box testing which gives decision coverage to Boolean values.
- This technique reports true and false outcomes of Boolean expressions.
- Whenever there is a possibility of two or more outcomes from the statements like do while statement, if statement and case statement (Control flow statements), it is considered as decision point because there are two outcomes either true or false.
- Decision coverage covers all possible outcomes of each and every Boolean condition of the code by using control flow graph or chart.

Decision Coverage Testing Cont.

- A decision point has two decision values one is true, and another is false that's why most of the times the total number of outcomes is two.
- The percent of decision coverage can be found by dividing the number of exercised outcome with the total number of outcomes and multiplied by 100.

$$\text{Decision Coverage} = \frac{\text{Number of Decision Outcomes Exercised}}{\text{Total Number of Decision Outcomes}} * 100$$

Decision Coverage Testing Cont.

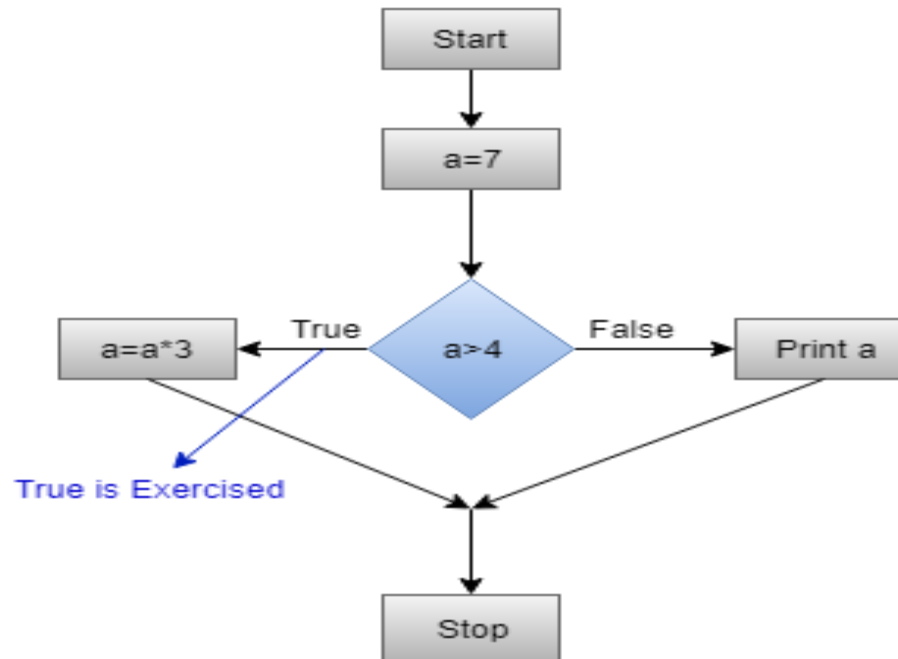
- Example 1

Test (int a)

```
{  
If(a>4)  
a=a*3  
Print (a)  
}
```

- If Value of a is 7 (a=7)

- Control flow graph when the value of a is 7.



- Calculation of Decision Coverage percent:

$$\text{Decision Coverage} = \frac{\text{Number of Decision Outcomes Exercised}}{\text{Total Number of Decision Outcomes}} * 100$$

Decision Coverage Testing Cont.

Decision Coverage = $\frac{1}{2} * 100$ (Only "True" is exercised)

$$= 100 / 2$$

$$= 50$$

Decision Coverage is 50%

- **Value of a is 3 (a=3)** Control flow graph when the value of a is 3

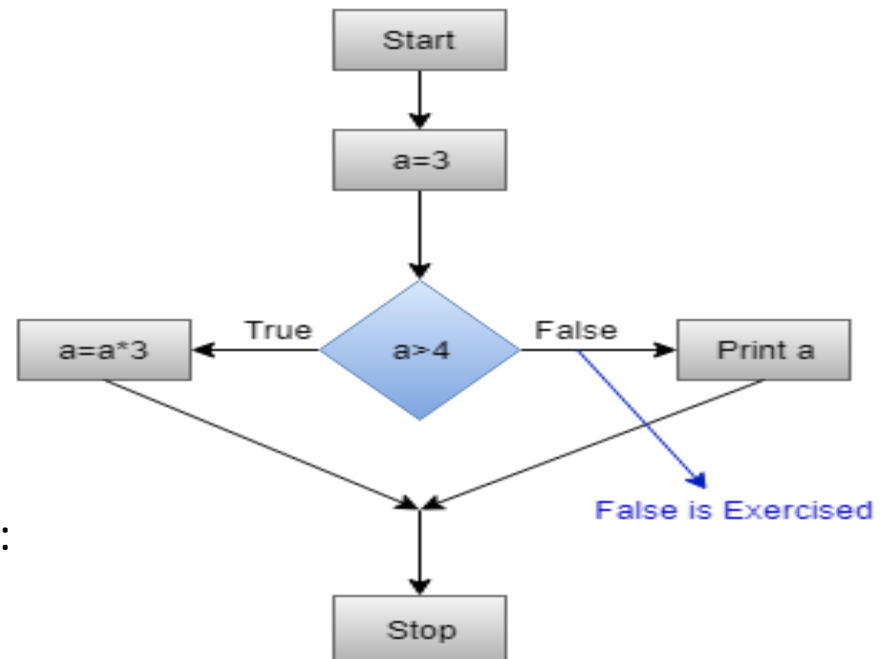
```
Test (int a=3)
```

```
{ if (a>4)
```

```
  a=a*3
```

```
  print (a)
```

```
}
```



Calculation of Decision Coverage percent:

Decision Coverage Testing Cont.

- Calculation of Decision Coverage percent:

$$\text{Decision Coverage} = \frac{\text{Number of Decision Outcomes Exercised}}{\text{Total Number of Decision Outcomes}} * 100$$

$$= \frac{1}{2} * 100 \text{ (Only "False" is exercised)}$$

$$= 100 / 2$$

$$= 50$$

$$\text{Decision Coverage} = 50\%$$

Test Case	Value of A	Output	Decision Coverage
1	3	3	50%
2	7	21	50%

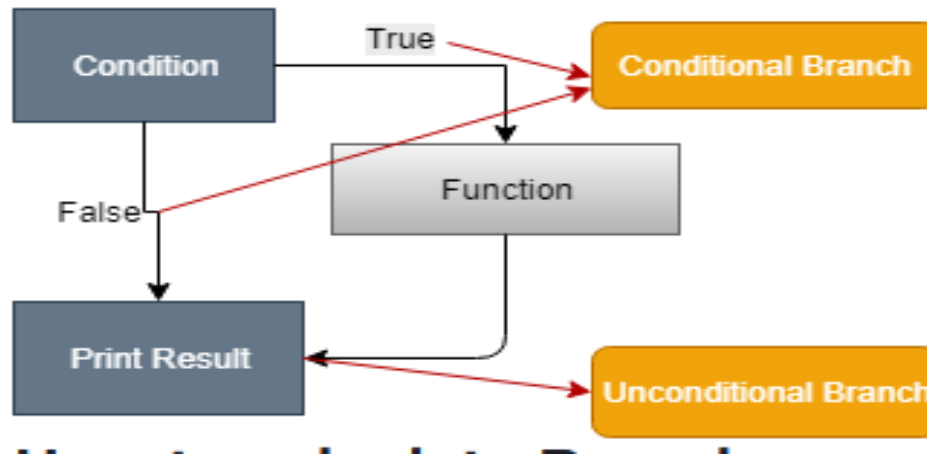
Branch Coverage Testing

- In most cases a decision has **two outcomes** (True or False), but it is possible for a decision to have more outcomes, for example, in “case of ...” statements.
- Branch coverage technique is used to cover **all branches of the control flow graph**.
 - It covers all the possible outcomes (true and false) of each condition of decision point at least once.
- Branch coverage technique is a white box testing technique that ensures **that every branch of each decision point must be executed**.

Branch Coverage Testing Cont.

- branch coverage technique and decision coverage technique are very similar, but there is a key difference between the two.
 - Decision coverage technique covers all branches of each decision point whereas branch testing covers all branches of every decision point of the code.
- In other words, branch coverage follows decision point and branch coverage edges.
- Many different metrics can be used to find branch coverage and decision coverage, but some of the most basic metrics are: finding the percentage of program and paths of execution during the execution of the program.
- Like decision coverage, it also uses a control flow graph to calculate the number of branches.

Branch Coverage Testing Cont.

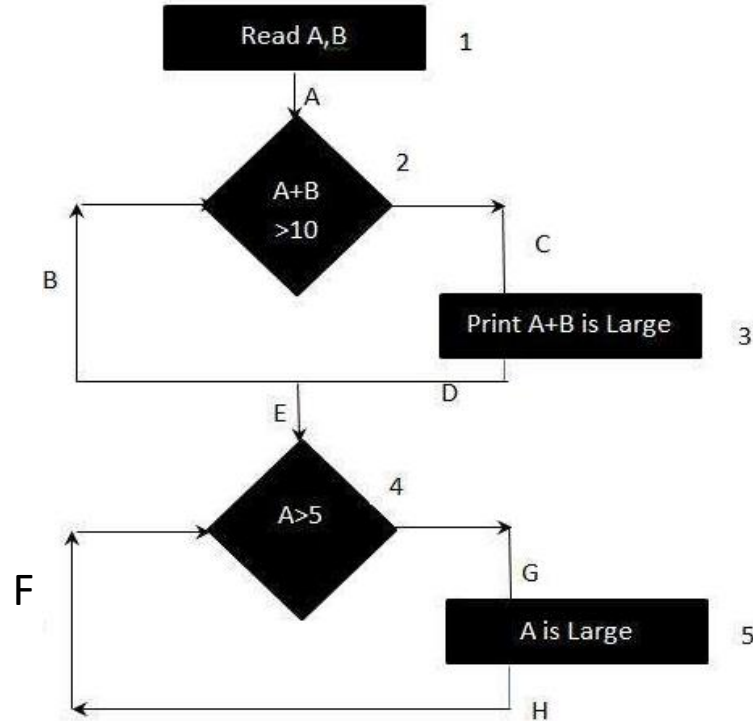


■ How to calculate Branch coverage?

- pathfinding is the most common method for calculate branch coverage.
 - ✓ the number of paths of executed branches is used to calculate Branch coverage.
- Branch coverage technique can be used as the alternative of decision coverage.
- Somewhere, it is not defined as an individual technique, but it is distinct from decision coverage and essential to test all branches of the control flow graph.

Branch Coverage Testing Cont.

Read A
Read B
IF A+B > 10 THEN
Print "A+B is Large"
ENDIF
If A > 5 THEN
Print "A Large"
ENDIF



Branch Coverage Testing Cont..

- Output
 - To calculate Branch Coverage, one has to find out the minimum number of paths which will ensure that all the edges are covered.
 - In this case there is no single path which will ensure coverage of all the edges at once.
 - The aim is to cover all possible true/false decisions.
 - (1) 1A-2C-3D-E-4G-5H
 - (2) 1A-2B-E-4F
- Hence Branch Coverage is 2.

Condition Testing

- A condition is a Boolean expression containing no Boolean operators, such as AND or OR. A condition is something that can be evaluated to be either TRUE or FALSE, like: “ $a < c$.”
- A statement like “X OR Y” is not a condition, because OR is a Boolean operator and X and Y Boolean operands. X and Y may in themselves be conditions
- Conditions are found in decision statements.



Condition Testing

- Decision statements may have one condition like:
 - if ($a < c$) then ...
- They may also be composed of more conditions combined by Boolean operands, like:
 - if (($a=5$) or (($c>d$) and ($c<f$))) then ...
- or for short: if (X or (Y and Z)) then ...
- The condition outcome is the evaluation of a condition to be either TRUE or FALSE.
- In condition testing we test condition outcomes.
- **The condition coverage** is the percentage of condition outcomes in every decision (in a component) that have been exercised by the test.
- So to get 100% condition outcome coverage we need to get each condition to be True and False (i.e., two test cases).

Condition Testing

- Example :-

In the first example with $(a < c)$ we can design the test cases:

Test case	a	c	Outcome
1	5	7	True
2	6	2	False

- more complex, example with $(X \text{ or } (Y \text{ and } Z))$ we also need to get each of the condition to be True and False.
- we can still get 100% condition coverage with 2 test cases, namely for example:

Test case	X	Y	Z
1	True	True	True
2	False	False	False

Home work.

- Condition testing may be weaker than decision testing. Why ?

Multiple Condition Testing

- With this test technique we test combinations of condition outcomes.
- To get 100% multiple combination coverage we must test all combinations of outcomes of all conditions.
- Because there are two possible outcomes for each condition (True and False) it requires 2^n test cases, where n is the number of conditions, to get 100% coverage.
- Example :if (X or (Y and Z)) then ..
- we have three conditions. We therefore need $2^3 = 8$ test cases to get 100% multiple condition coverage.
- The test cases we need are:

Multiple Condition Testing Cont.

Test case	X	Y	Z
1	True	True	True
2	False	True	True
3	False	False	True
4	False	False	False
5	False	True	False
6	True	False	False
7	True	True	False
8	True	False	True

- The number of test cases grows exponentially with the number of conditions!

Condition Determination Testing.

- Sometimes testing to 100% multiple condition coverage would be to go overboard in relation to the risk associated with the component.
- In these cases we can use the condition determination testing technique.
- With this technique we should design test cases to execute branch condition outcomes that independently affect a decision outcome.
- The number of test cases needed to achieve 100% condition determination coverage depends on how the conditions are combined in the decision
- statements:
 - As a minimum we need $n+1$ test cases
 - As a maximum we need 2^n test cases
- where n is the number of conditions.

Condition Determination Testing.

- Example
 - if (X or (Y and Z)) then ..
- we need the test cases listed in this table.

Test case	X	Y	Z
1	True	-	-
2	False	True	True
3	False	False	-
4	False	True	False

- A “-” means that we don’t care about what the outcome is, because it does not have impact on the result.
- We can hence get 100% modified condition decision coverage or condition determination coverage with just four test cases.

Path Testing

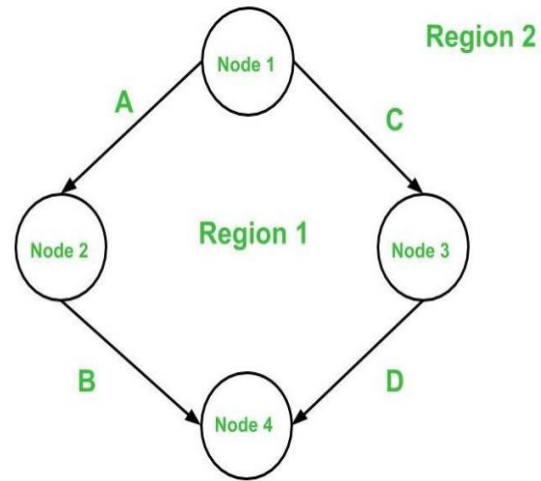
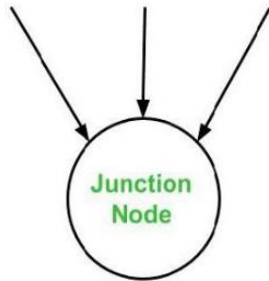
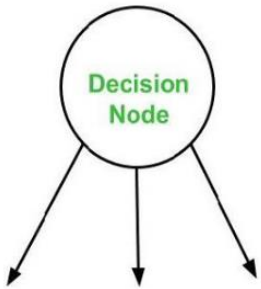
- A path is a sequence of executable statements in a component from an entry point to an exit point.
- Path coverage is the percentage of paths in a component exercised by the test cases.
- When you execute test cases based on any of the other structure-based techniques described above, you will inevitably execute paths through the code.
- Four stages are followed to create test cases using this technique –
 - Create a Control Flow Graph.
 - Calculate the Graph's Cyclomatic Complexity
 - Identify the Paths That Aren't Connected
 - Create test cases based on independent paths.

Path Testing Cont.

Control Flow Graph

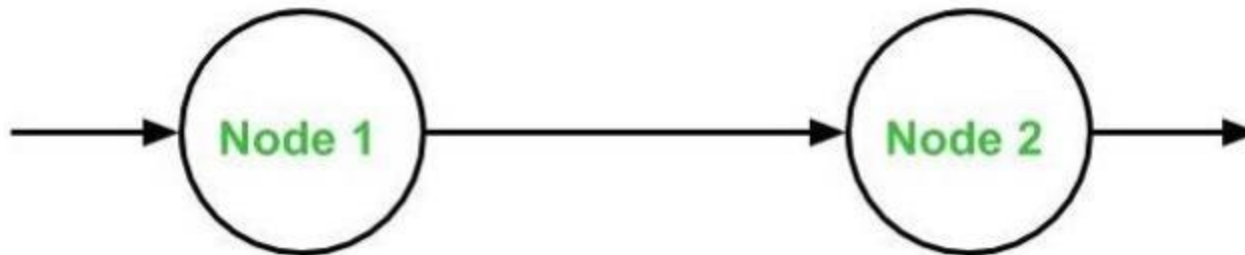
- A control flow graph (or simply flow graph) is a directed graph that depicts a program's or module's control structure.
- V number of nodes/vertices and E number of edges make up a control flow graph (V, E).
- A control graph can also include the following
 - A node with multiple arrows entering it is known as a junction node.
 - A node having more than one arrow leaving it is called a decision node.
 - The area encompassed by edges and nodes is referred to as a region (area outside the graph is also counted as a region.).

Path Testing Cont.



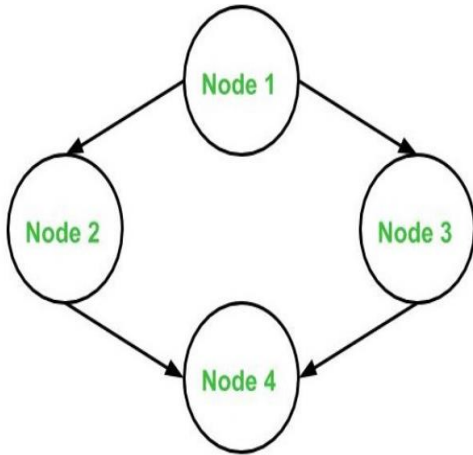
- Below are the notations utilized while constructing a flow graph
 - Sequential Statements

Sequence

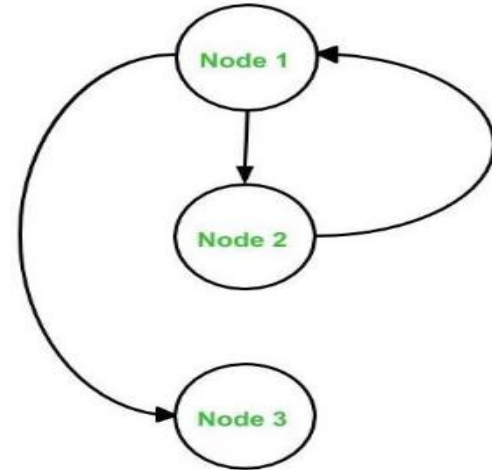
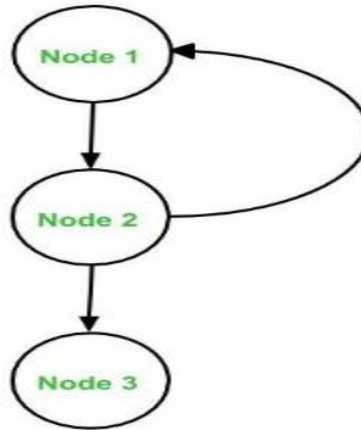


Path Testing Cont.

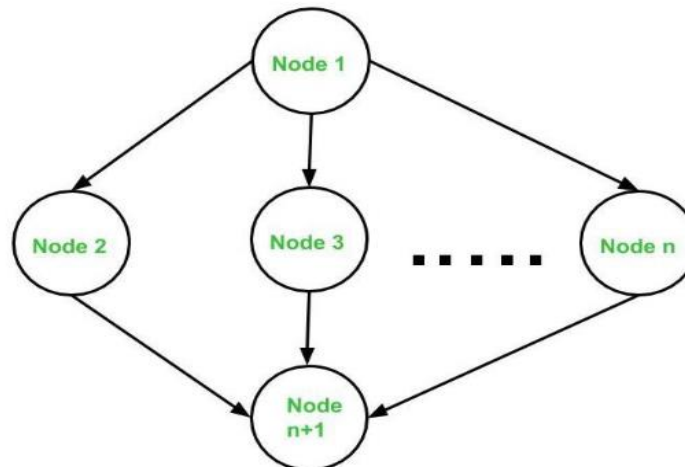
- If – Then – Else



- Do – While • While – Do



- Switch – Case



Path Testing Cont.

- Cyclomatic Complexity

- what is measurement ?

Measurement :- is noting but quantitative indication of size , dimension , capacity of an attributes of products or processes .

✓ Example : Numbers of errors

- What is software metrics?

Software metrics is defined as quantitative measure of an attribute a software system process with respect to cost, quality, size and schedule.

✓ Example : numbers of errors found per person.

Cyclomatic Complexity is a software metric used to measure the complexity of a program.

- It is quantitative measure of independent paths in the source code of the program.

Path Testing Cont.

- **independent path** is defined as a path that has at least one edge which has not been traversed before in any other path.

Cyclomatic Complexity calculate with flow graph

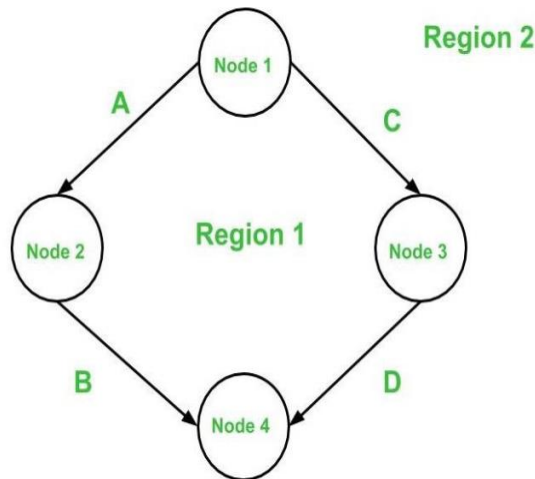
- **cyclomatic complexity** can be calculated with respect to **functions, modules , methods or classes** with in a program.
- The cyclomatic complexity $V(G)$ is said to be a metric for a program's logical complexity.
- Three distinct formulas can be used to compute it :-
 - 1. Formula based on edges and nodes :**
 - $V(G) = e - n + 2$ is a formula based on **edges** and **nodes**.
 - Where **e is the number of edges**, and **n denotes the number of vertices**.
 - 2. Formula based on Decision Nodes :**
 - $V(G) = P + 1$ where P number of predicate nodes (that contain conditions like if and while condition statements.).

Path Testing Cont.

3. Formula based on Regions :

- $V(G)$ = number of regions in the graph

Example :



Method 2: Formula based on Decision Nodes :

$$V(G) = d + P$$

where, $d = 1$ and $p = 1$ So,

$$\text{Cyclomatic Complexity } V(G) = 1 + 1 = 2$$

Method 1: Formula based on edges and nodes :

$$V(G) = e - n + 2$$

where, $e = 4$, $n = 4$

So,

$$\begin{aligned} \text{Cyclomatic complexity } V(G) &= 4 - 4 + 2 \\ &= 2 \end{aligned}$$

Method 3: Formula based on Regions

$V(G)$ = number of regions in the graph

Cyclomatic complexity

$$\begin{aligned} V(G) &= 1 \text{ (for Region 1)} + 1 \\ &\text{(for Region 2)} = 2 \end{aligned}$$

Path Testing Cont.

■ Example 2:

```
int i=0;
```

```
N=4;// number of nodes present in the graph
```

```
While(i<n-1) do
```

```
  j=i+1;
```

```
  While (j<n)do
```

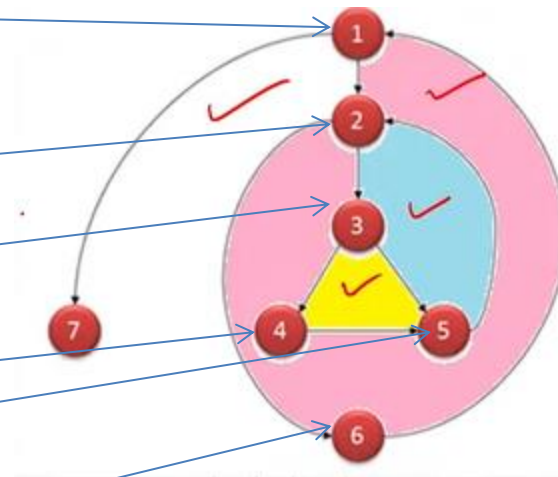
```
    If A[i]<A[j] then
```

```
      Swap(A[i],A[J]);
```

```
    enddo
```

```
  i=i+1;
```

```
endDo
```



- How many predicate condition have the above program or flow graph end mention the node ? 3 such as node 1(While(i<n-1)),2(While (j<n)),3(If A[i]<A[j])

Path Testing Cont.

- **Method 1: Formula based on edges and nodes :**

$$V(G) = e - n + 2$$

where, $e = 9$, $n = 7$

So,

$$\begin{aligned}\text{Cyclomatic complexity } V(G) &= 9 - 7 + 2 \\ &= 4\end{aligned}$$

- **Method 3: Formula based on Regions**

$V(G)$ = number of regions in the graph

Cyclomatic complexity

$$V(G) = 4$$

- **Method 2: Formula based on Decision Nodes :**

$$V(G) = d + P$$

where, $d = 1$ and $p = 3$ So,

$$\text{Cyclomatic Complexity } V(G) = 1 + 3 = 4$$

- Set of possible execution path of a program

1. 1,7

2. 1,2,3,4,5,2,6,1,7

3. 1,2,6,1,7

4. 1,2,3,5,2,6,1

- The shortest independent path of cyclometric complexity path is 1,7

- the longest independent path of Cyclomatic complexity is 1,2,3,4,5,2,6,1,7

Data flow Testing

- Data Flow Testing is a type of structural testing .
- It is a method that is used to find the **test paths of a program** according to the **locations of definitions** and **uses of variables** in the program.
- It is concerned with:
 - Statements where **variables receive values**,
 - Statements where these values are **used** or **referenced**.
- By analyzing **control flow graphs**, this technique aims to identify issues such as **unused variables** or **incorrect definitions**, **ensuring proper handling** of data within the code.

Data flow Testing Cont.

- To illustrate the approach of data flow testing, assume that each statement in the program is assigned a unique statement number. For a statement number S-
 - $DEF(S) = \{X \mid \text{statement } S \text{ contains the definition of } X\}$
 - $USE(S) = \{X \mid \text{statement } S \text{ contains the use of } X\}$
- If a statement is a loop or if condition then its DEF set is empty and the USE set is based on the condition of statement s.
- Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.
- Reference or defined anomalies in the flow of the data are detected at the time of associations between values and variables.
- These anomalies are:
 - A variable is defined but not used or referenced,
 - A variable is used but never defined,
 - A variable is defined twice before it is used

Data Flow Testing Cont.

Types of Data Flow Testing

The different types of the software data flow testing are listed below

1. **Testing for All-Du-Paths** : It refers to the **All Definitions Use Paths**.
 - It verifies all possible paths from the variable definition to the usage.
2. **All-Du-Paths Predicate Node Testing** : It verifies the **predicate nodes**, or **decision points** which are the part of **the control flow graph**.
3. **All-Uses Testing** : It verifies every place where a variable is **utilized**.
4. **All-Defy Testing** :It verifies every place where a variable is **specified**.
5. **Testing for All-P-Uses** :It refers to the All Possible Uses.
 - It verifies all probable usages of a variable.

Data Flow Testing Cont.

6. **All-C-Uses Test** :It refers to the **All Computation Uses**. It verifies all possible paths where **a variable has been used for calculations**.

7. **Testing for All-I-Uses** : It refers to the **All Input Uses**. It verifies all possible paths where **a variable is obtained from external inputs**.

8. **Testing for All-O-Uses** : It refers to the **All Output Uses**. It verifies all possible paths where **a variable is used to generate outputs**.

9. **Testing for Definition Use Pairs** : It focuses on specific pairs of definitions, and uses for variables.

10. **Testing for Use-Definition Paths** : It evaluates the path that leads to the point where **a variable is used and then defined**.

Data Flow Testing Cont.

Advantages of Data Flow Testing

- Data Flow Testing is used to find the following issues:-
 - To find a variable that is used but never defined,
 - To find a variable that is defined but never used,
 - To find a variable that is defined multiple times before it is use,
 - Deallocating a variable before it is used.

Disadvantages of Data Flow Testing

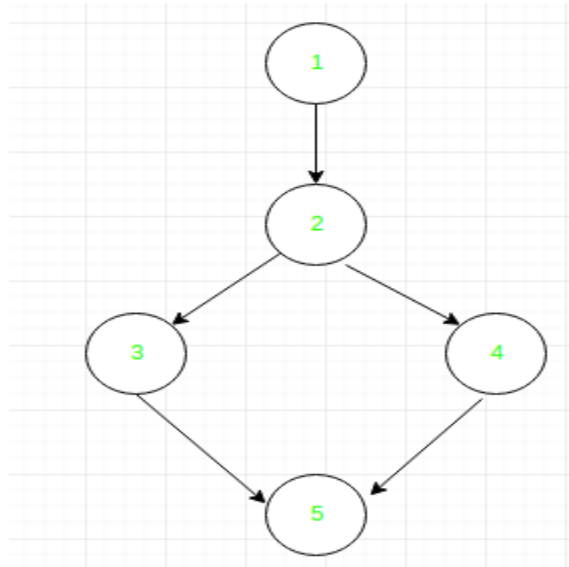
- Time consuming and costly process
- Requires knowledge of programming languages

Data Flow Testing Cont.

Example 1

1. read x, y;
2. if(x>y)
3. a = x+1
- else
4. a = y-1
5. print a;

Control flow graph of above example:



Define/use of variables of above example:

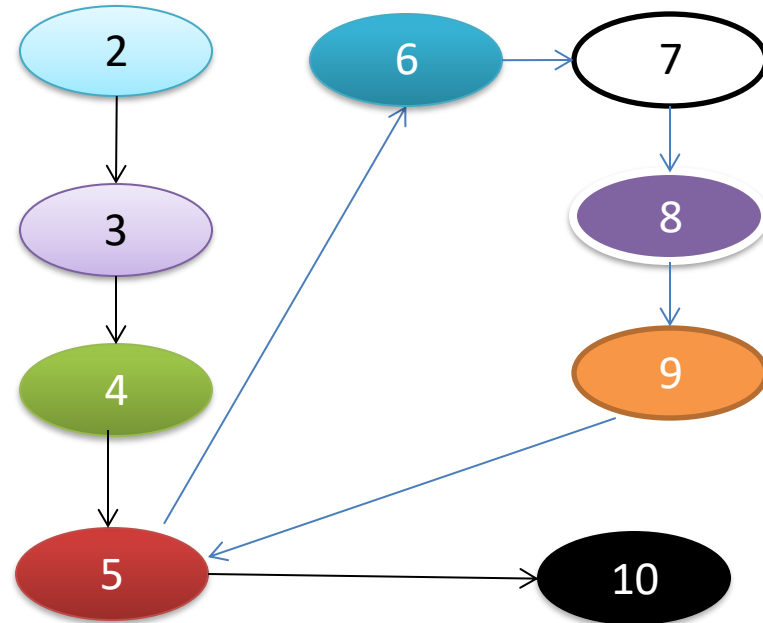
Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

Data Flow Testing Cont.

Example2

1. var x,y,sum
2. x=0
3. sum=0
4. read (y)
5. while (y!=0)
6. sum=sum+y
7. x=x+1
8. read(y)
9. end while
10. Print("the sum of"+ x+" number is"+ sum)

Control flow graph

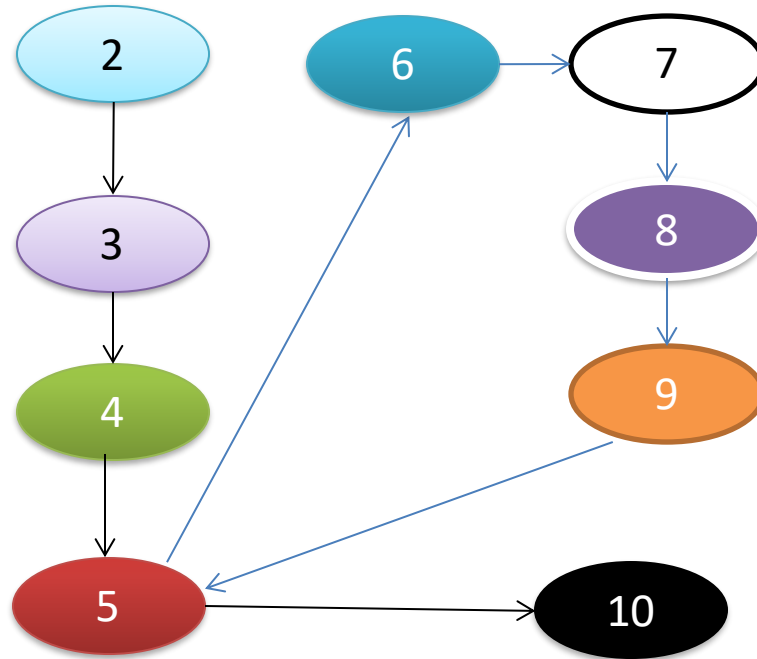


➤ Assume at step 4 read 0 for y value ,then the control flow graph will be



Data Flow Testing Cont.

Assume at step 4 read 5 for y value and at step 8 read 0 for y value then the control flow graph will be



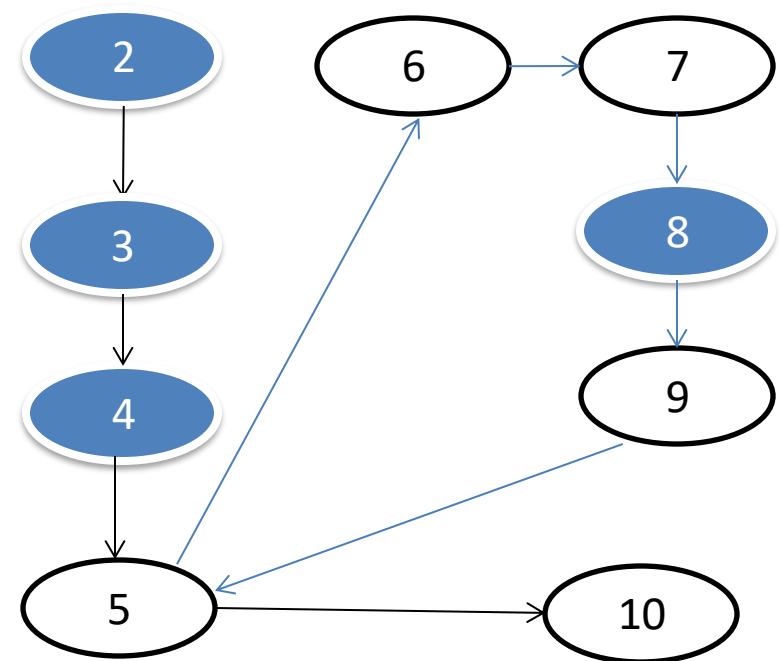
Data Flow Testing Cont.

❖ Define use testing

- One of data flow testing techniques
- Use paths related to variable in CFG.
- variables are either define or used.

■ Define use testing - define nodes

```
1.  var x,y,sum
2.  x=0
3.  sum=0
4.  read (y)
5.  while (y!=0)
6.    sum=sum+y
7.    x=x+1
8.    read(y)
9.  end while
10. Print("the sum of"+ x+" number is"+ sum)
```

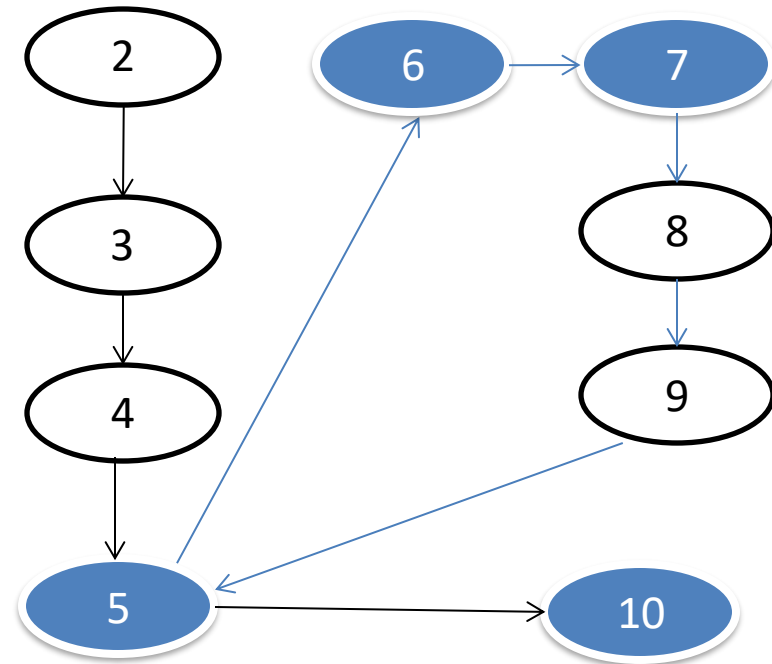


Data Flow Testing Cont.

❖ Define use testing - use nodes

- P-use(predicate use) (eg. $Y \neq 0$)
- C-use(computation use) (eg. For variable y: $\text{sum} = \text{sum} + y$)
 - ✓ (O-use(output use) .L-use (Location use) or I-use(Iteration use))

```
1.  var x,y,sum
2.  x=0
3.  sum=0
4.  read (y)
5.  while (y!=0)
6.  sum=sum+y
7.  x=x+1
8.  read(y)
9.  end while
10. Print("the sum of" + x + " number is" + sum)
```



Data Flow Testing Cont.

- Define use node for variable x

1	var x,y,sum	
2	x=0	Define
3	sum=0	
4	read (y)	
5	while (y!=0)	
6	sum=sum+y	
7	x=x+1	Define , use
8	read(y)	
9	end while	
10	Print("the sum of"+ x+" number is"+ sum)	Use

Data Flow Testing Cont.

- Define use node for variable y

1	var x,y,sum	
2	x=0	
3	sum=0	
4	read (y)	Define
5	while (y!=0)	Use
6	sum=sum+y	Use
7	x=x+1	
8	read(y)	Define
9	end while	
10	Print("the sum of"+ x+" number is"+ sum)	

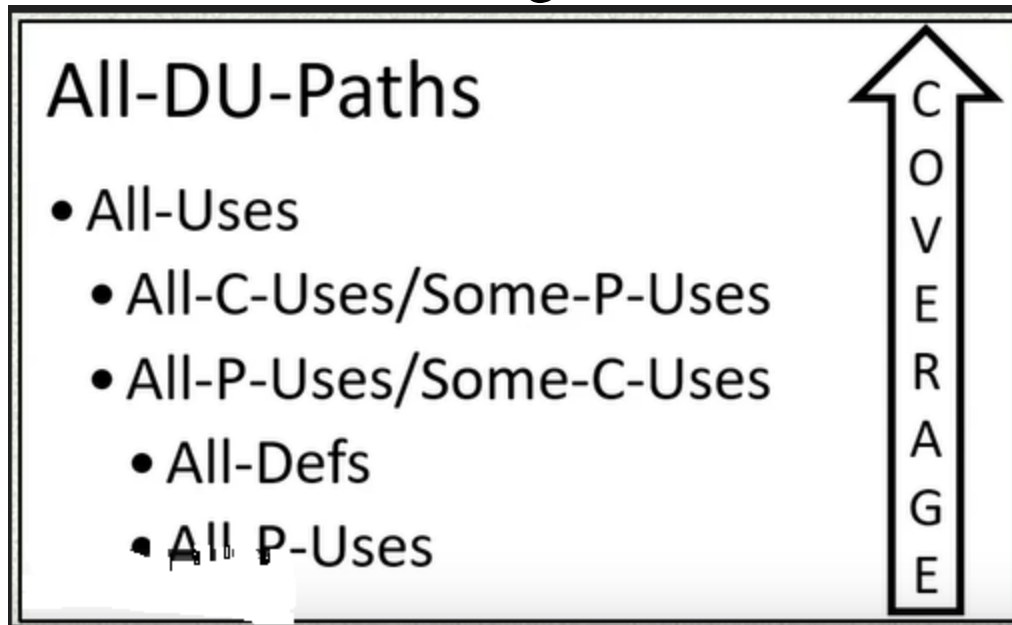
Data Flow Testing Cont.

- Define use node for variable sum

1	var x,y,sum	
2	x=0	
3	sum=0	Define
4	read (y)	
5	while (y!=0)	
6	sum=sum+y	Define, Use
7	x=x+1	
8	read(y)	
9	end while	
10	Print("the sum of"+ x+" number is"+ sum)	Use

Data Flow Testing Cont.

- define use test
 - Select a set of path and test them to find anomalies (bugs)
 - all path (too many !), all edges and nodes
 - DU paths(define use paths)
 - DC Paths(Define Clear paths)
- In define use testing , there are 6 data flow testing metrics



Data Flow Testing Cont.

- Program slices

- one of data flow testing techniques
- a set of statements using which the variable value is computed
- slice can be up to any point in the program.

Data Flow Testing Cont.

- Slices for variable x

1	var x,y,sum		
2	x=0	S(x,2)	{2}
3	sum=0		
4	read (y)		
5	while (y!=0)		
6	sum=sum+y		
7	x=x+1	S(x,7)	{2,5,7,9}
8	read(y)		
9	end while		
10	Print(“the sum of”+ x+” number is”+ sum)	S(x,10)	{2,5,7,9}

Data Flow Testing Cont.

- Slices for variable y

1	var x,y,sum		
2	x=0		
3	sum=0		
4	read (y)	S(y,4)	{4}
5	while (y!=0)	S(y,5)	{4,5,8,9}
6	sum=sum+y	S(y,6)	{4,5,8,9}
7	x=x+1		
8	read(y)	S(y,8)	{8}
9	end while		
10	Print("the sum of"+ x+" number is"+ sum)		

Data Flow Testing Cont.

- Slices for variable sum

1	var x,y,sum		
2	x=0		
3	sum=0	S(sum,3)	{3}
4	read (y)		
5	while (y!=0)		
6	sum=sum+y	S(sum,6)	{3,4,5,6,8,9}
7	x=x+1		
8	read(y)		
9	end while		
10	Print("the sum of"+ x+" number is"+ sum)	S(sum,10)	{3,4,5,6,8,9}

Data Flow Testing Cont.

■ Definition Use paths for variable x

1	var x,y,sum			
2	x=0	Def	S(x,2)	{2}
3	sum=0			
4	read (y)			
5	while (y!=0)			
6	sum=sum+y			
7	x=x+1		S(x,7)	{2,5,7,9}
8	read(y)			
9	end while			
10	Print("the sum of"+ x+" number is"+ sum)		S(x,10)	{2,5,7,9}

<2,7>

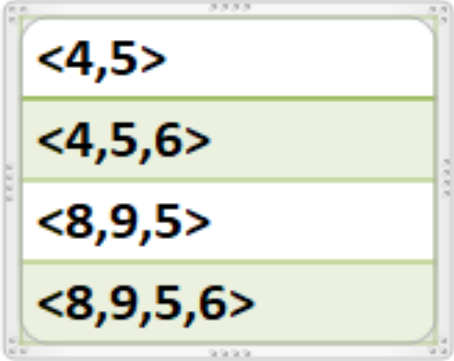
<2,7,10>

<7,9,10>

<7,5,7,9,10>

Data Flow Testing Cont.

■ Definition Use paths for variable y

1	var x,y,sum				
2	x=0				
3	sum=0				
4	read (y)		Def	S(y,4)	{4}
5	while (y!=0)		Use	S(y,5)	{4,5,8,9}
6	sum=sum+y		Use	S(y,6)	{4,5,8,9}
7	x=x+1				
8	read(y)		Def	S(y,8)	{8}
9	end while				
10	Print(“the sum of”+ x+” number is”+ sum)				

Data Flow Testing Cont.

■ Definition Use paths for variable sum

1	var x,y,sum			
2	x=0	<3,6>		
3	sum=0	<3,4,5,6,8,9,5,10>	Def	S(sum,3) {3}
4	read (y)	<6>		
5	while (y!=0)	<6,8,9,5,10>		
6	sum=sum+y		Def , use	S(sum,6) {3,4,5,6,8,9}
7	x=x+1			
8	read(y)			
9	end while			
10	Print(“the sum of”+ x+” number is”+ sum)		Use	S(sum,10) {3,4,5,6,8,9}

Thank You !!!

