Chap 3.Test Models and Strategies

3.3 Code Coverage Model

1. Introduction
2. Control Flow Graph
3. Test Coverage Analysis

1. Introduction

-Test coverage attempts to address questions about when to stop testing, or the amount of testing that is enough for a given program?

-Ideal testing is to explore exhaustively the entire test domain, which in general is impossible.

÷So in practice, some code may never be executed due to the possibility of missing test cases.

÷Hence, one hardly knows how effective test suites are until one finds what code is, or isn't executed.

-A code coverage model calls out the parts of an implementation that must be exercised to satisfy an implementation-based test model.

÷Coverage, as a metric, is the percentage of these parts exercised by a test suite.

-Hundreds of coverage models have been published and used since the late 1960s. Nearly all support implementation-based testing.

÷As such, most coverage models rely on ***control flow graphs***, which give an abstract representation of the code.

2. Control Flow Graph

Code Segments

-A **code segment** consists of one or several contiguous statements with no conditionally executed statements.

÷That means once a segment is entered, all the statements involved will execute.

÷The last statement in the segment must be another predicate, a method exit, a loop control, a break, or a goto.

÷The last part of a segment includes the predicate or exit expression that selects another segment but does not include any of subsequent segment’s code.

-A **predicate** expression contains one or many conditions that evaluate to true or false. One **condition** corresponds to each boolean operator in the predicate expression.

÷Predicates are used in control statements: if, case, do, while, do until, for, and so on.

÷The evaluation of a predicate results in transfer of control to one or many **code segments**.

÷A predicate with multiple conditions is called a **compound predicate.**

Examples: Code segments in Canonical loop structures

For Loop

```
buffer= new char[nchar + 1];
```
A

```
for (n=0;   n< nchars;   ++n) {
```
B, D

```
                    buffer[n] = newChar;
}
```
C

While Loop

```
buffer= new char[nchar + 1];
int n =0;
```
A

```
while ( n< nchars) {
```
B

```
                    buffer[n] = newChar;
                    ++n;
}
```
C

Until Loop

```
buffer= new char[nchar + 1];
int n =0;
```
A

```
do  {
                    buffer[n] = newChar;
                    ++n;
}
```
B

```
While (n < nchars);
```
C

Representation

-A **control flow graph (CFG)** describes code segments and their sequencing in a program. It is a directed graph in which:

÷A **node** corresponds to a code segment; nodes are labeled using letters or numbers.

÷An **edge** corresponds to a conditional transfer of control between code segments; edges are represented as arrows.

-The entry point of a method is represented by the **entry node**, which is a node with no inbound edges.

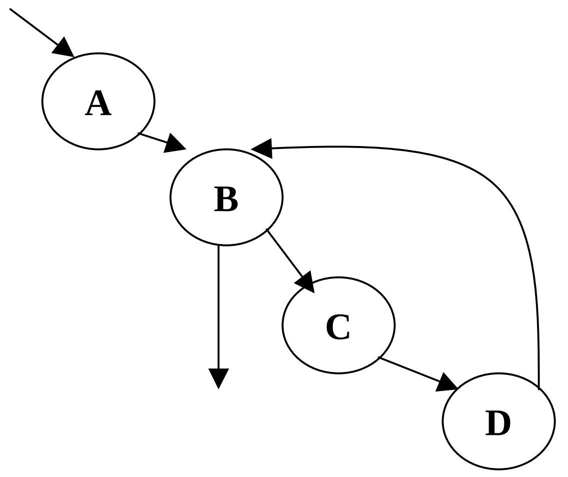
-The exit point of a method is represented by the **exit node**, which is a node with no outbound edges.

Flow graphs for canonical loop structures

For Loop

buffer= new char[nchar + 1];			A
for (n=0;	n< nchars;	++n) {	B, D

buffer[n] = newChar;
}
			C

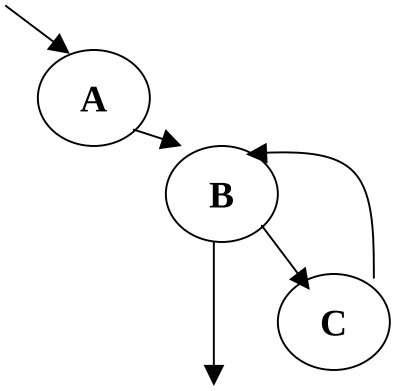


While Loop

```
buffer= new char[nchar + 1];
int n =0;
```
A

```
while ( n< nchars) {
```
B

```
            buffer[n] = newChar;
            ++n;
}
```
C

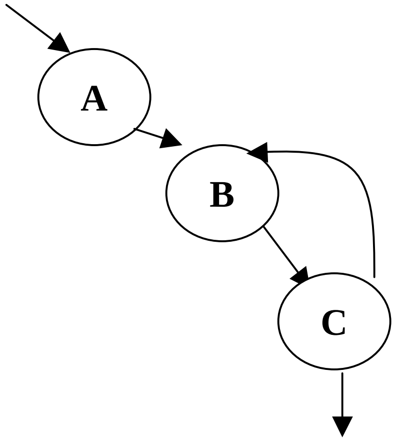


Until Loop

```
buffer= new char[nchar + 1];
int n =0;
```
A

```
do  {
            buffer[n] = newChar;
            ++n;
}
```
B

```
While (n < nchars);
```
C

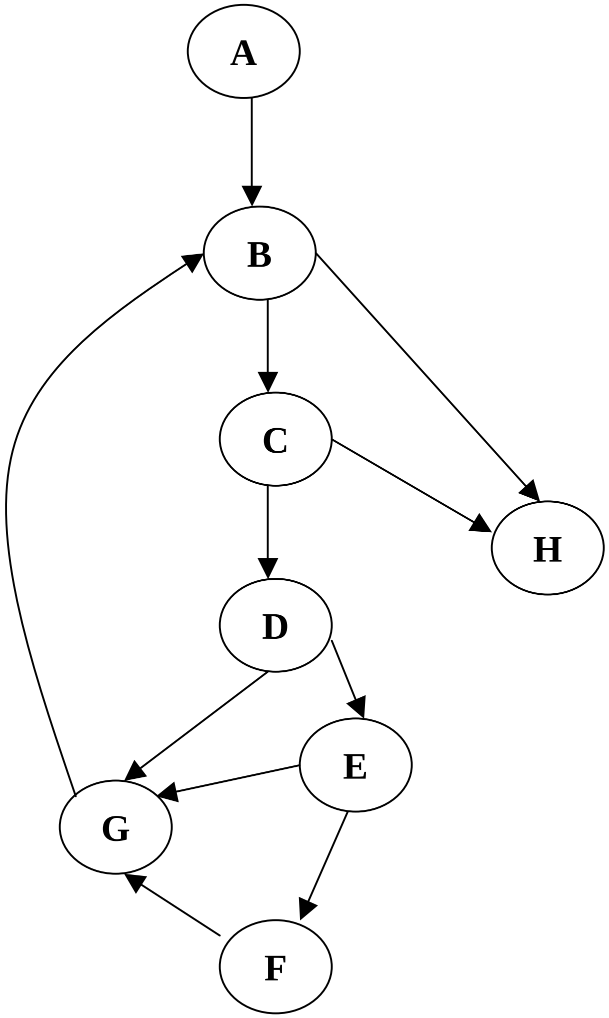


Example: CFG for a method

```
public class Authenticator {
   final int MAX = 10000;
  String [] userids = new String [MAX];
  String [] passwords = new String [MAX];
              …
```

		Segments
public boolean verify (String uid, String pwd) { boolean result =false; int i = 0;		A
while ((result ==false)	&& (i < MAX)) {	B, C
if ((userids[i]==uid)	&& (passwords[i] == pwd))	D, E
result=true;		F
++i; }		G
return result; }		H

```
            //…Other methods
 } //Class end
```



Path Expressions

-A ***path*** corresponds to a sequence of segments connected by arrows.

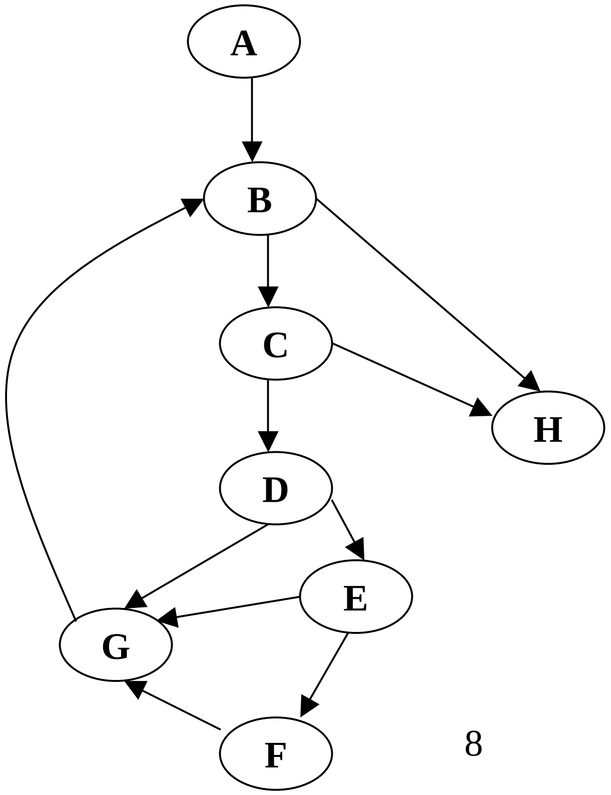
÷A path is denoted by the nodes that comprise the path.

÷Loops are represented by segments within parentheses, followed by an asterisk to show that this group may iterate from zero to *n* times.

-An ***entry-exit path*** is a path starting with the entry node and ending with the exit node

Examples of Entry/Exit Paths:

-ABH
-ABCH
-A(BCDEFG)*BH
-A(BCDEG)*BH
-A(BCDG)*BH
-A(BCDG)*BCH



Compound Predicates

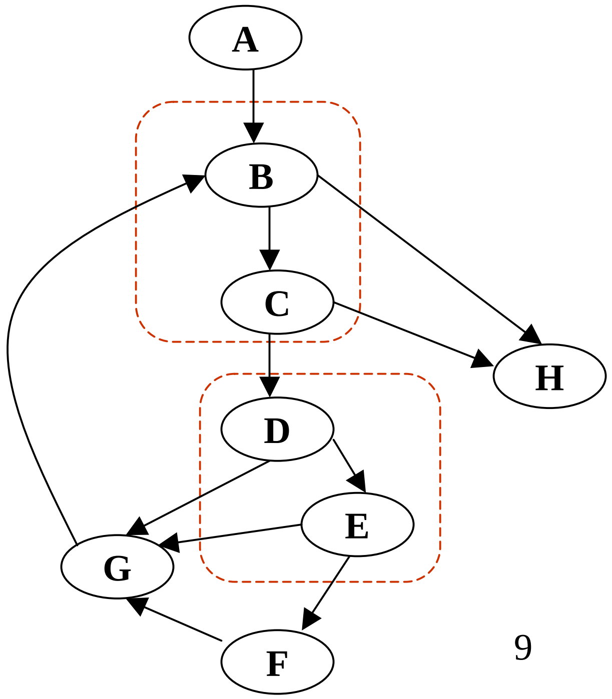
-Compound predicate expressions are modeled separately by specifying a node for each individual predicate involved.

-All true-false combinations in a compound predicate should be analyzed, and the effects of short circuit boolean evaluation should be made explicit.

÷For instance: C++, Java, and Objective-C use C semantics- short circuit boolean evaluation is automatically applied to all multiple-condition boolean expressions.

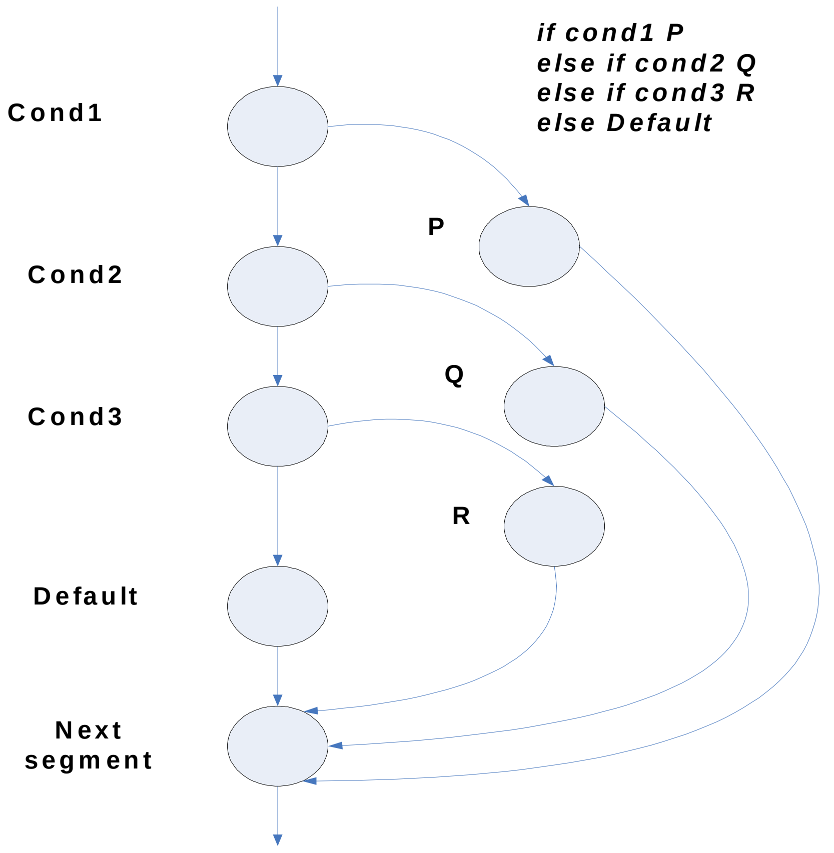
÷Example:

```
                    …
while ((result ==false)   && (i < MAX)) {

    if ((userids[i]==uid)  && (passwords[i] = pwd ))

        result=true;

        ++i;
  }
                    …
```



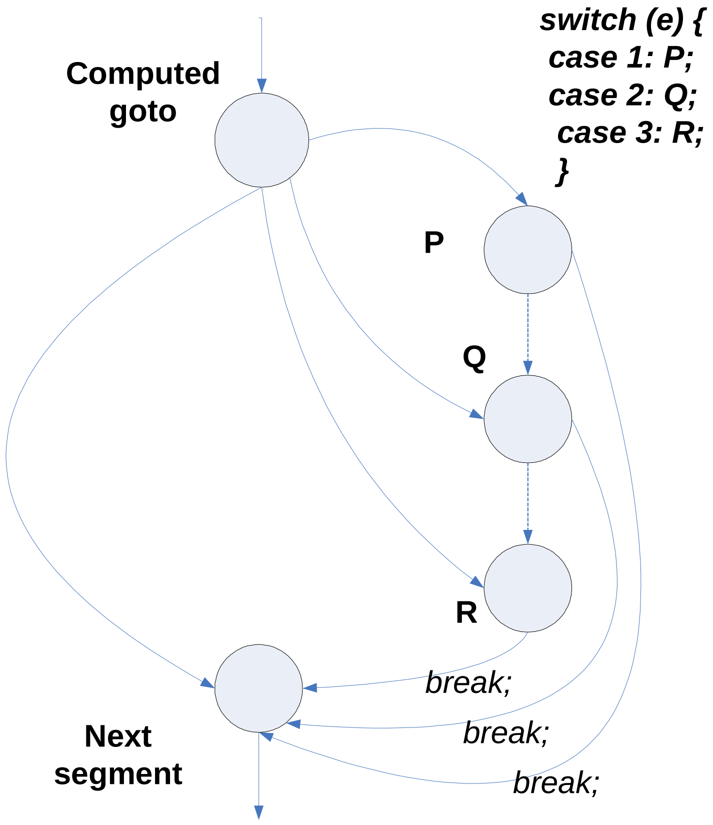
Case and Multiple-if Statements

-Case and multiple-if statements are modeled by specifying a separate node for each predicate, each conditional action, and the default action.



Switch Statements

-Switch statements are modeled by specifying a node for the switch expression, and a separate node for each action.



3. Test Coverage Analysis

-Test coverage analysis uses some adequacy criteria to guide the testing process.

÷This increases the confidence that an implementation has been thoroughly tested.

-It is recommended not to use a code coverage model as a test model.

÷Instead, established test strategies (e.g. equivalence, domain) should be used to devise test suites, while coverage are used at the same time to analyze generated test suites adequacy.

÷Coverage reports can point out a grossly inadequate test suite

÷Coverage reports can help to identify implementation constructs that may require implementation-based test design or the development of special stubs and drivers

-Common coverage criteria include:

÷Statement coverage

÷Branch coverage

÷Condition/Multiple condition coverage

÷Basis-path coverage

÷Data flow coverage

Common Code Coverage Criteria

Statement coverage

-Achieved when all statements in a method have been executed at least once.

÷Statement coverage is also known as line coverage, segment coverage, or basic block coverage.

÷Segment and basic block coverage counts segments instead of individual statements.

÷Segment coverage ensures that all code segments defined in the CFG are covered. For instance, if we can find one entry-exit path that includes all segments, we can realize 100% statement coverage.

-Example:

```
void foo(int x) {
    if (a==b) { ++x; }
    else { --x;}
    return x;
}
```

$\longrightarrow$

Statement coverage is achieved by having test cases involving both:

1. a==b

2. a != b

- If a bug exists in a statement, and that statement is not executed, there is almost no chance of revealing that bug; hence statement coverage is the minimum coverage required by IEEE SENG standards.
- However, it is a very weak criterion and should be viewed as the barest minimum.

- Example:

```
void foo () {
        char* c = NULL;
        if (x == y) {
                c = &aString;
        }
        *c = “test code”;
}
```

→ Statement coverage is achieved for *foo()* by setting *x* equal *y*; however, when the condition is false, the code generates an incorrect pointer and crashes.

Branch Coverage

-Achieved when every path from a node is executed at least once by a test suite.

÷Also known as decision coverage or all-edge coverage.

÷Improve on statement coverage by requiring that each branch be taken at least once. Hence each outcome of a predicate expression is exercised at least once (i.e., true and false).

÷However, it is limited by the fact that it treats a compound predicate as a single statement. Branch coverage may be achieved without exercising all of the conditions

-Example

```
int foo(int x) {
        if (a==b || (x==y && isEmpty()) {
                ++x;
        }
        else {
                --x;
        }
        return x;
}
```

Branch coverage may be Achieved using:

1. a ==b
2. a !=b and x !=y

Without ever exercising *this.isEmpty()*

÷ Branch coverage misses several of the possible entry-exit paths.

Condition coverage/Multiple condition coverage

-Condition coverage requires that each condition be evaluated as true and false at least once.

-Multiple-condition coverage requires that all true-false combinations of simple conditions be exercised at least once.

÷There are at most 2^n true-false combinations for a compound predicate with *n* simple conditions.

÷Multiple-condition coverage ensures that all statements, branches, and conditions are covered, but not all paths.

÷Reaching all condition combinations may be impossible due to short circuit evaluation.

÷Mutually exclusive conditions spanning several predicate expressions may preclude certain combinations.

Basis-Path Model

Cyclomatic Complexity Metric

-The cyclomatic complexity metric C is defined as the number of edges minus the number of nodes plus 2:

$$C = e - n + 2$$

÷Where e and n stand for the number of edges and the number of nodes in corresponding CFG, respectively.

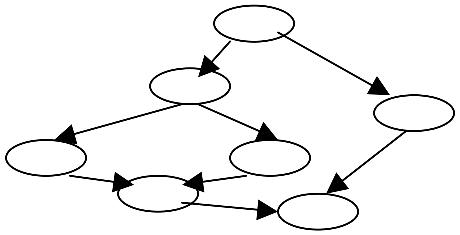
-Also called **McCabe complexity** metric

÷Evaluate the ***complexity of algorithms*** involved in a method.

÷Give a count of the minimum number of test cases needed to test a method comprehensively.

÷Low C value means reduced testing, and better understandability.

-Example:



CC= e- n + 2 = 8 – 7 + 2 = 3

The Basis-Path Test Model

-The **basis-path** test model calls for testing *C* distinct entry-exit paths.

÷A test suite is developed by finding C distinct entry-exit paths and producing test cases by **path sensitization**.

÷Traditionally, it is suggested that the longest entry-exit path be included in the test suite.

÷**Basis-path coverage** is achieved when C distinct entry-exit paths have been exercised.

-The basis-path model is appealing because it is simple and pragmatic.

-However it is unreliable as a coverage metric, because it can be satisfied without meeting the barest minimum generally accepted standards of code coverage:

÷On one hand branch coverage may be achieved with less than C paths in some methods.

÷On the other hand it is possible to select C entry-exit paths and achieve neither statement coverage nor branch coverage.

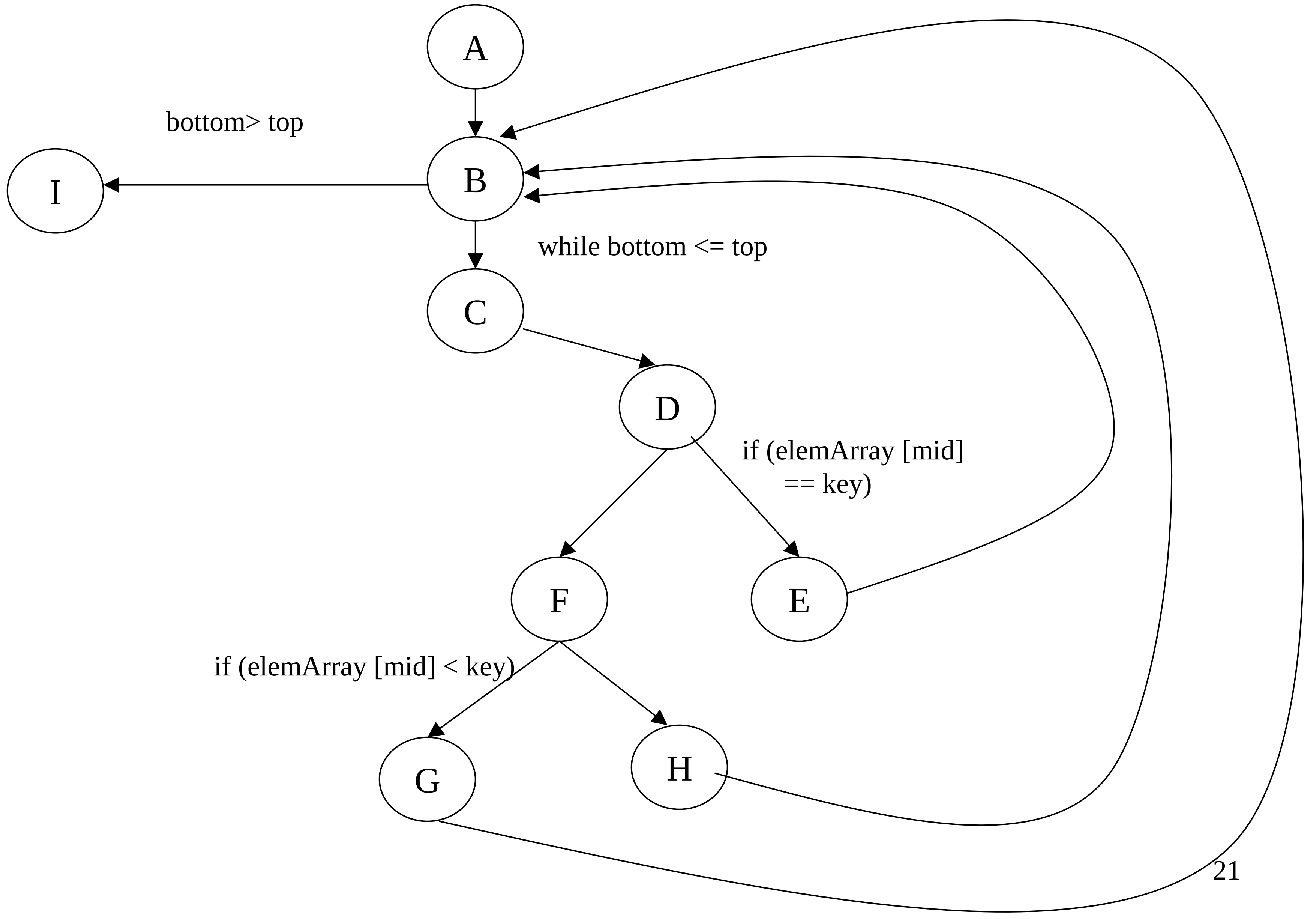
Example: Binary Search Routine

Code

```
Class BinSearch {
        public static void search(int key, int[] elemArray, Result r) {
                int bottom = 0;
                int top = elemArray.length – 1;
                int mid;
                r.found = false; r.index=-1;
                while (bottom <= top) {
                        mid = (top + bottom)/2;
                        if (elemArray [mid] == key) {
                                r.index = mid;
                                r.found = true;
                                return;
                        }
                        else {
                                if (elemArray[mid] < key) bottom = mid + 1;
                                else top = mid – 1;
                        }
                }
        }
}
```

```
Class BinSearch {

        public static void search(int key, int[] elemArray, Result r) {          A
                int bottom = 0;
                int top = elemArray.length – 1;
                int mid;
                r.found = false; r.index=-1;
-----------------------------------------------------------------------------
                while (bottom <= top)    {                                          B
-----------------------------------------------------------------------------
                        mid = (top + bottom)/2;                                     C
-----------------------------------------------------------------------------
                        if (elemArray [mid] == key) {                               D
-----------------------------------------------------------------------------
                                r.index = mid;
                                r.found = true;                                     E
                                return;
                        }
-----------------------------------------------------------------------------
                        else {
                                if (elemArray[mid] < key)                           F
-----------------------------------------------------------------------------
                                        bottom = mid + 1;                           G
-----------------------------------------------------------------------------
                                else top = mid – 1;                                 H
                        }
                }
-----------------------------------------------------------------------------
        }
                                                                                    I
}
```

Flow Graph


A
B
I
bottom> top
while bottom <= top
C
D
if (elemArray [mid] == key)
F
E
if (elemArray [mid] < key)
G
H

Independent Paths and Number of Test cases

ABI
A(BCDE)*BI
A(BCDFG)*BI
A(BCDFH)*BI

-The minimum number of test cases required to test all program paths is equal to the cyclomatic complexity (CC).

CC = Number (edges) – Number (nodes) + 2 = 11 – 9 + 2 = 4

Path Sensitization

-Path sensitization is the process of determining argument and instance variable values that will cause a particular path to be taken.

-Path sensitization is undecidable; no algorithm can solve this problem in all cases.

-Instead it must be solved heuristically:

÷For small, well-structured methods, this task is usually not difficult.

÷It becomes increasingly more difficult as size and complexity increase.

-Using initially other test strategies (e.g., domain analysis etc.) to select test inputs reduce the amount of work required.

÷All that remains is to find additional test values to exercise the missing paths to meet the coverage goal. You can use existing test cases to find values for these missing paths.

Test Coverage, in Practice

-In practice, the three basic criteria most commonly used are *statement coverage*, *branch coverage* and *condition coverage*.

÷It has been suggested that the combination of these three criteria can achieve 80-90 % or more coverage in most cases.

-It is important to note that test coverage is not enough by itself:

÷100% test coverage cannot guarantee achieving error-free software.

÷However, the use of test coverage in combination with appropriate test strategies can mitigate the impact of uncovered code during software testing, and help the tester find some rational points at which to stop testing.

-In practice, coverage data are collected by instrumenting the code. Instrumentation is typically done on a copy of the source code, while debugging changes are made to the un-instrumented code.

÷Manual instrumentation is error-prone and very time consuming.

÷In contrast, automatic instrumentation is easy and cost-effective. Automatic instrumentation is performed by a ***coverage analyzer***, which is part of typical test automation environment.