



University of  
Sheffield



COM3529 Software Testing and Analysis

White-Box Coverage Criteria based on

# Control Flow Analysis

Professor Phil McMinn

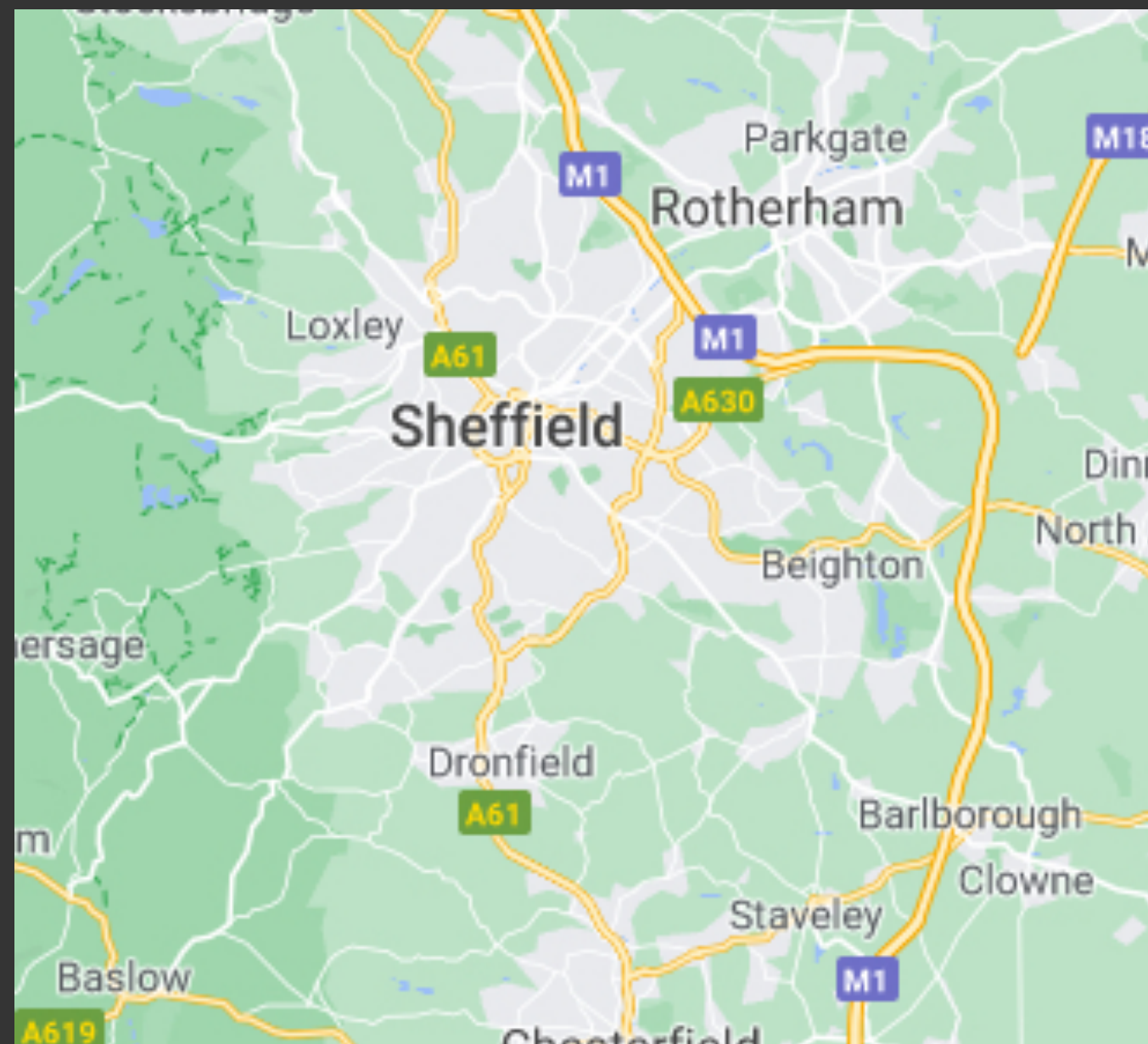
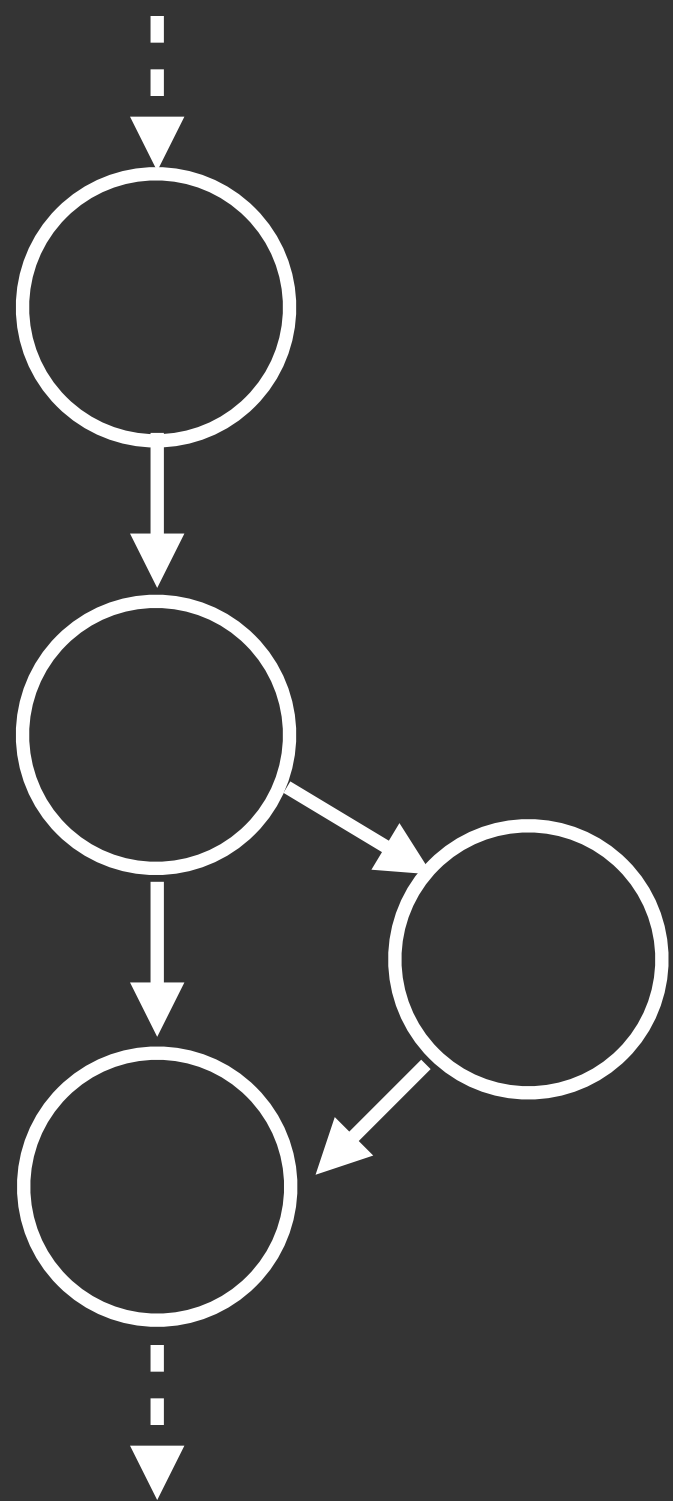
# Control Flow

```
public static Set<Character> duplicateLetters(String s) {  
    // lower case the string and remove all characters  
    // that are not letters  
    s = s.toLowerCase().replaceAll("[^a-z.]", "");  
  
    // initialise the result set  
    Set<Character> duplicates = new TreeSet<>();  
  
    // iterate through the string  
    for (int i = 0; i < s.length(); i++) {  
        char si = s.charAt(i);  
  
        // iterate through the rest of the string,  
        // checking for the same letter  
        for (int j = i + 1; j < s.length(); j++) {  
            char sj = s.charAt(j);  
  
            if (si == sj) {  
                // a match has been found, add it to  
                // the result set  
                duplicates.add(si);  
            }  
        }  
    }  
  
    return duplicates;  
}
```

# Control Flow Graph

```
public static Set<Character> duplicateLetters(String s) {  
    // lower case the string and remove all characters  
    // that are not letters  
    s = s.toLowerCase().replaceAll("[^a-z.]", "");  
  
    // initialise the result set  
    Set<Character> duplicates = new TreeSet<>();  
  
    // iterate through the string  
    for (int i = 0; i < s.length(); i++) {  
        char si = s.charAt(i);  
  
        // iterate through the rest of the string,  
        // checking for the same letter  
        for (int j = i + 1; j < s.length(); j++) {  
            char sj = s.charAt(j);  
  
            if (si == sj) {  
                // a match has been found, add it to  
                // the result set  
                duplicates.add(si);  
            }  
        }  
    }  
  
    return duplicates;  
}
```

# Control Flow Graph (CFG)

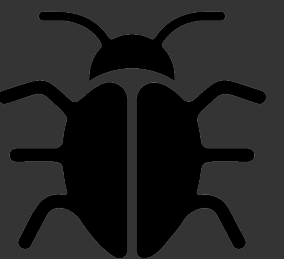
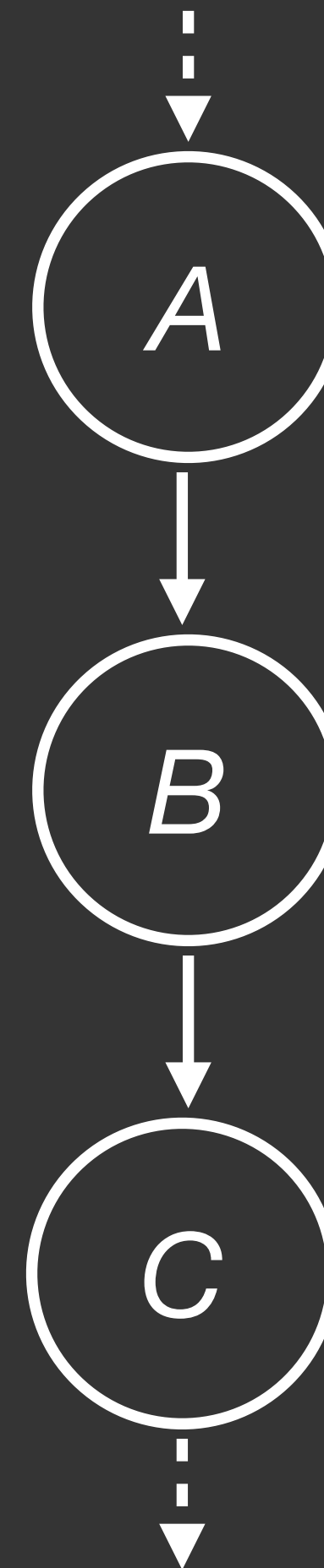


```
public static Set<Character> duplicateLetters(String s) {  
    // lower case the string and remove all characters  
    // that are not letters  
    s = s.toLowerCase().replaceAll("[^a-z.]", "");  
  
    // initialise the result set  
    Set<Character> duplicates = new TreeSet<>();  
  
    // iterate through the string  
    for (int i = 0; i < s.length(); i++) {  
        char si = s.charAt(i);  
  
        // iterate through the rest of the string,  
        // checking for the same letter  
        for (int j = i + 1; j < s.length(); j++) {  
            char sj = s.charAt(j);  
  
            if (si == sj) {  
                // a match has been found, add it to  
                // the result set  
                duplicates.add(si);  
            }  
        }  
    }  
  
    return duplicates;  
}
```



# Linear Sequences of Statements

```
doSomething(); /* A */  
doSomethingElse(); /* B */  
doSomethingDifferent(); /* C */
```



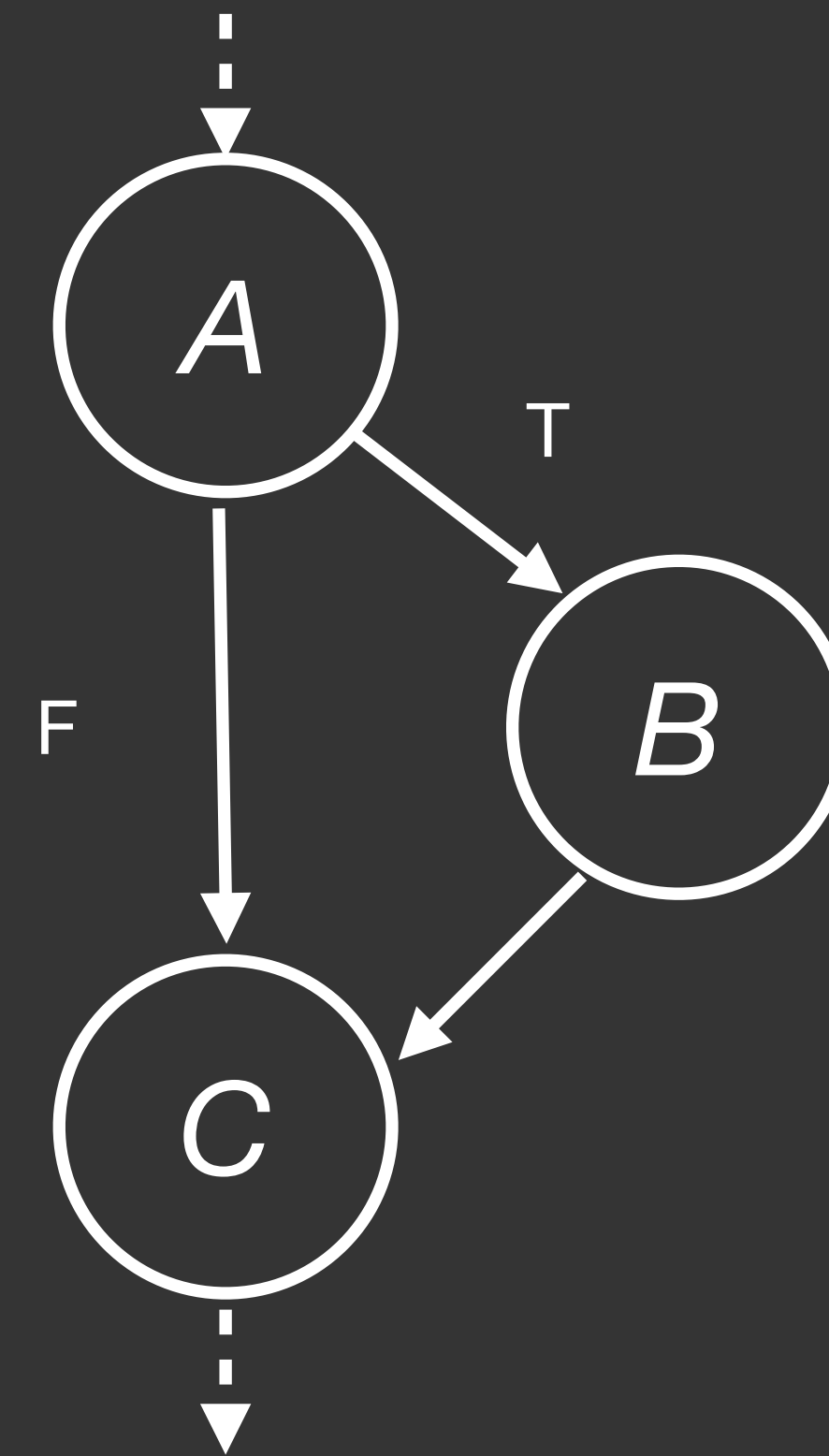
# Linear Sequences of Statements

```
doSomething(); /* A */  
doSomethingElse(); /* B */  
doSomethingDifferent(); /* C */
```

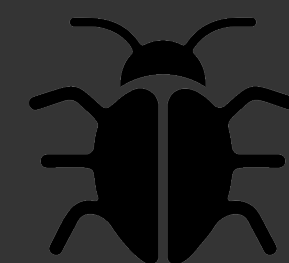
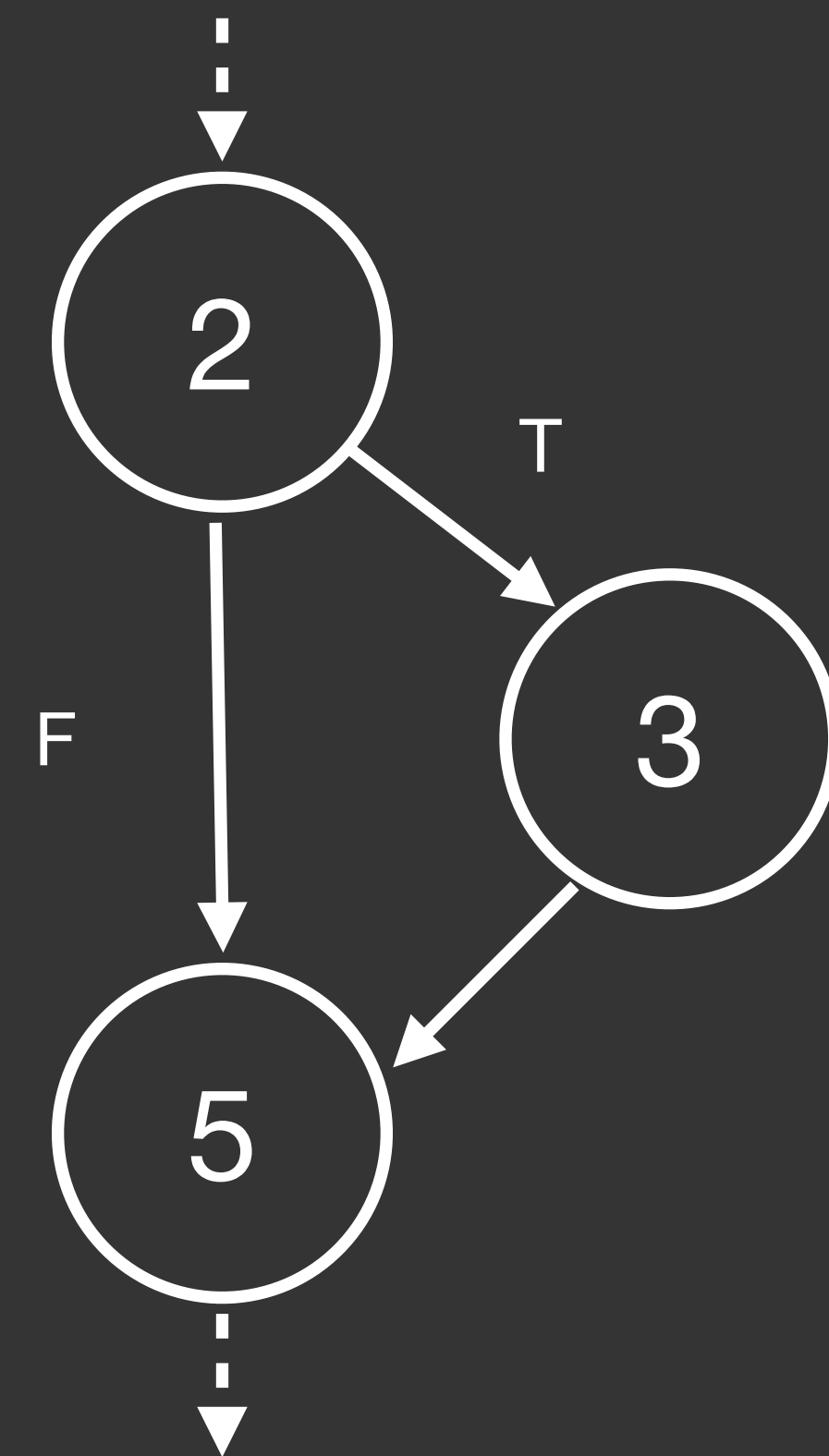


# “If” Constructs

```
if (condition) { /*A*/  
    doSomething(); /* B */  
}  
/* C */
```

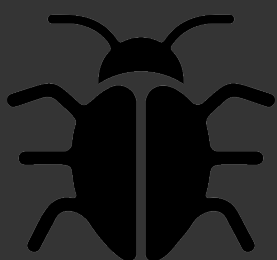
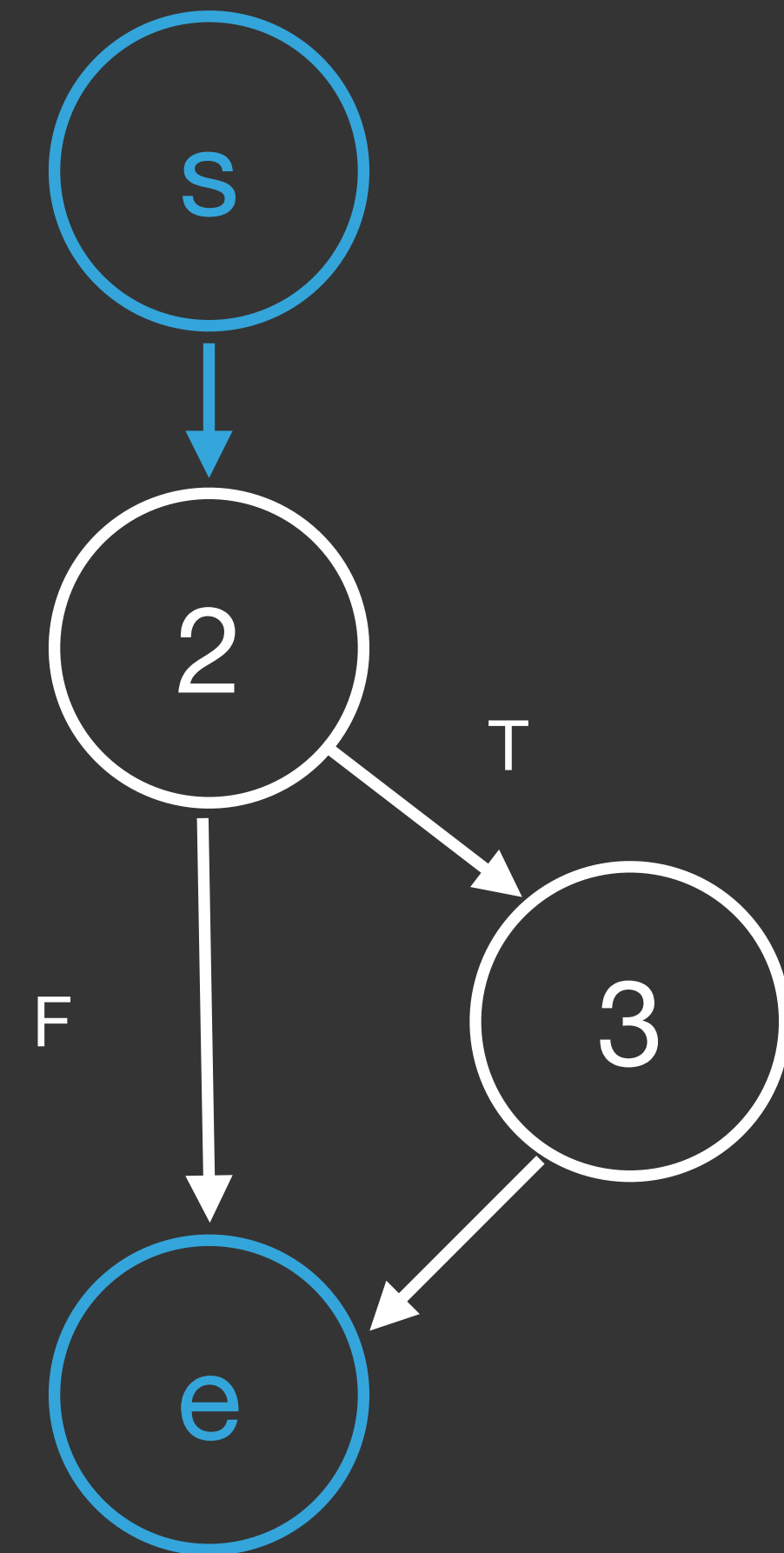


```
1 public void printGreeting() {  
2     if (isMorning()) {  
3         System.out.println("Good Morning!");  
4     }  
5 }
```

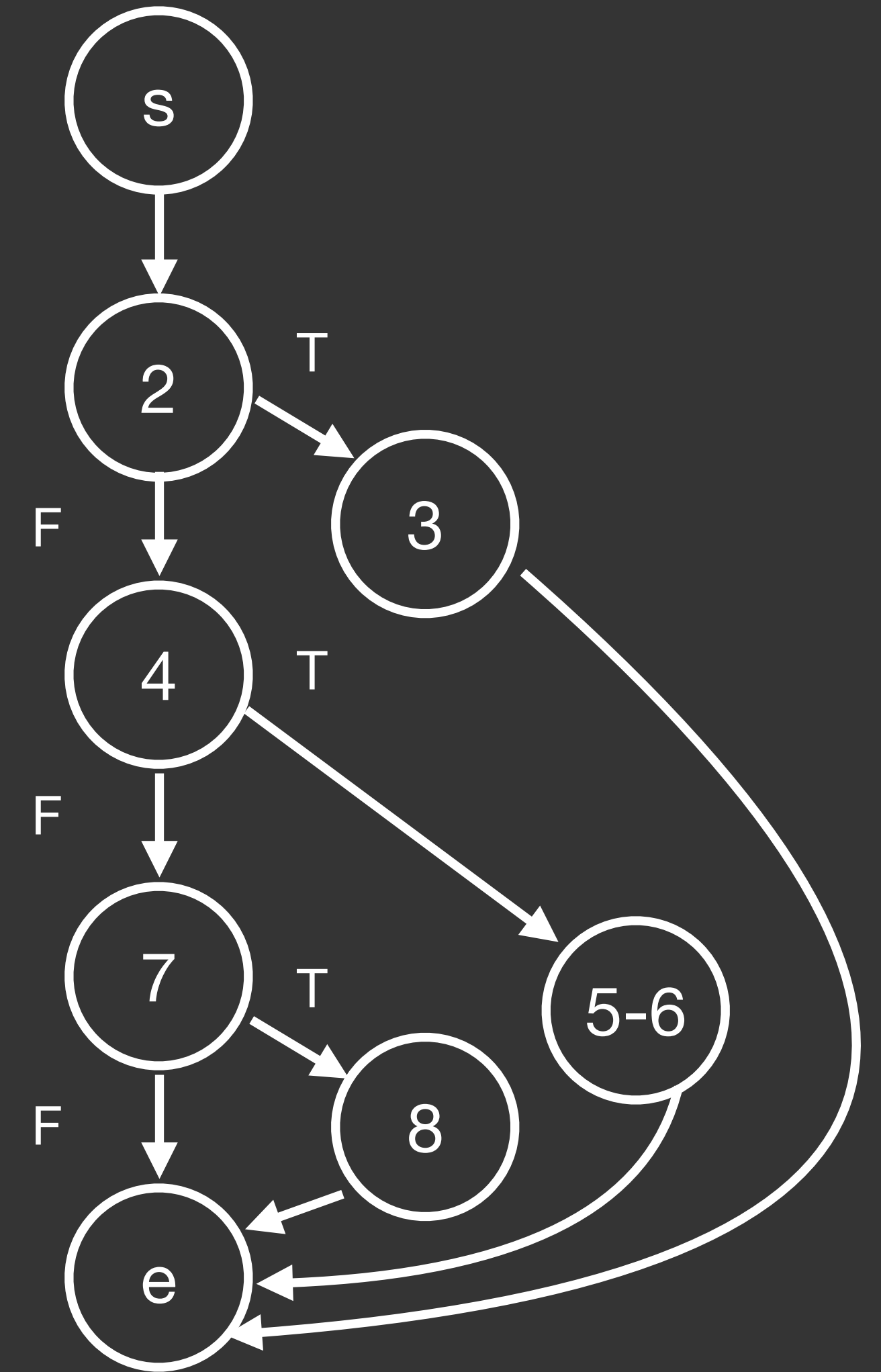




```
1 public void printGreeting() {  
2     if (isMorning()) {  
3         System.out.println("Good Morning!");  
4     }  
5 }
```

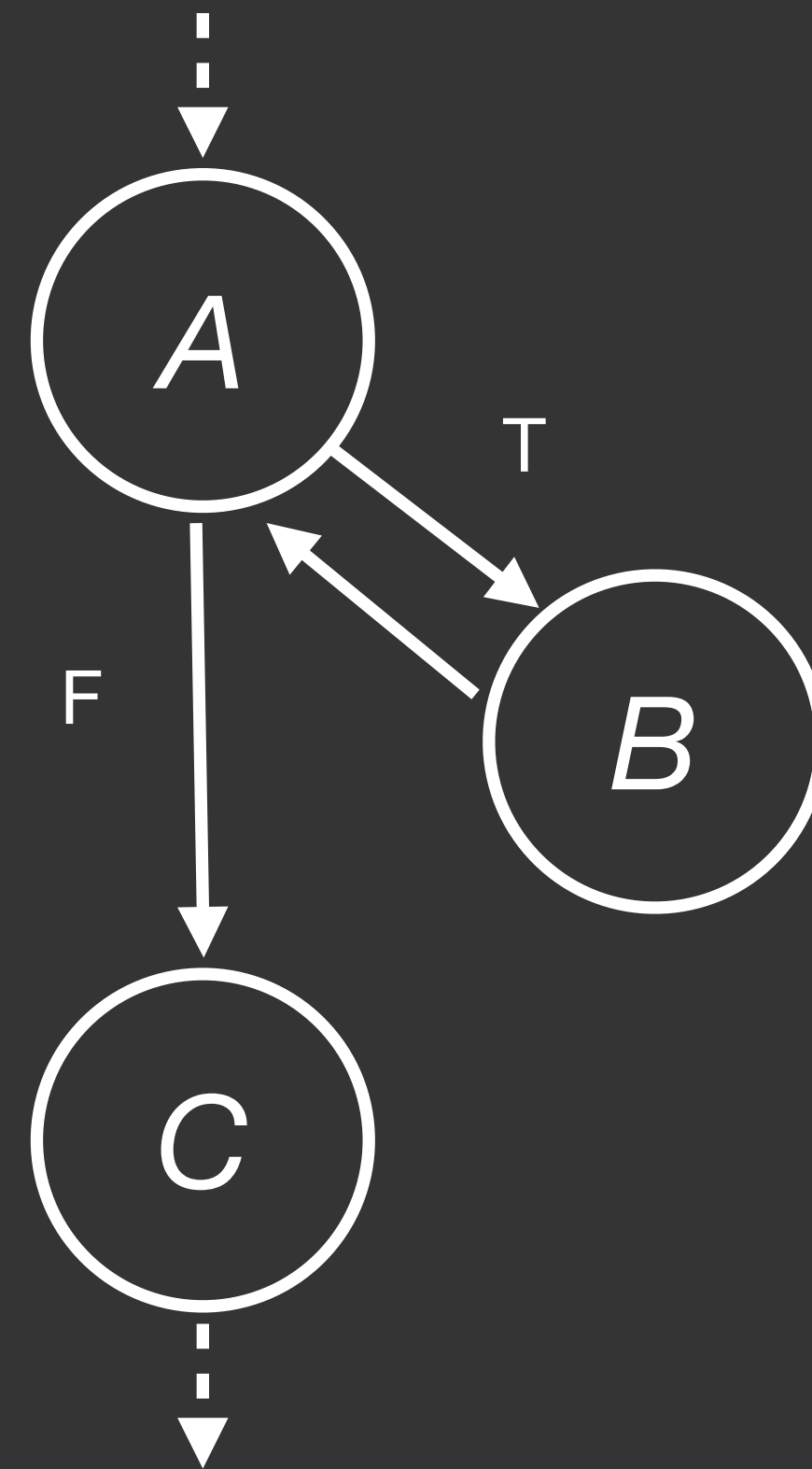


```
1 public void printGreeting() {  
2     if (isMorning()) {  
3         System.out.println("Good Morning!");  
4     } else if (isAfternoon()) {  
5         System.out.println("Good Afternoon!");  
6         System.out.println("Lovely day for a stroll!");  
7     } else if (isEvening()) {  
8         System.out.println("Good Evening!");  
9     }  
10 }
```



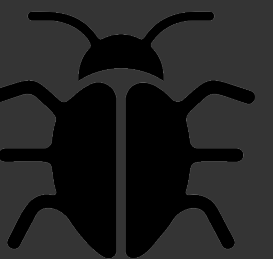
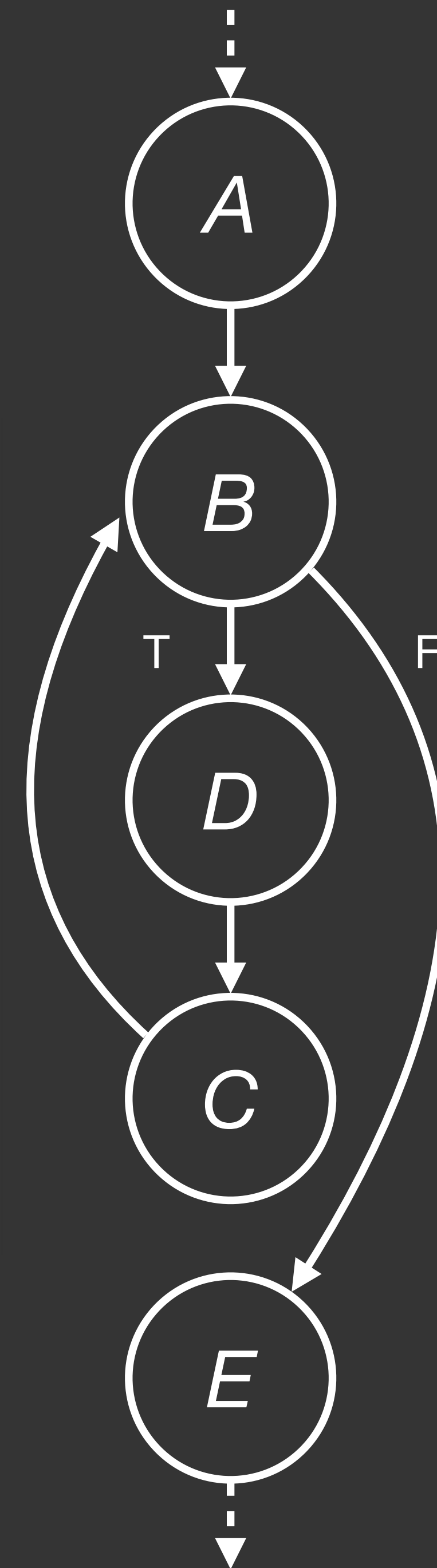
# “While” Constructs

```
while (condition) { /*A*/  
    doSomething(); /* B */  
}  
/* C */
```

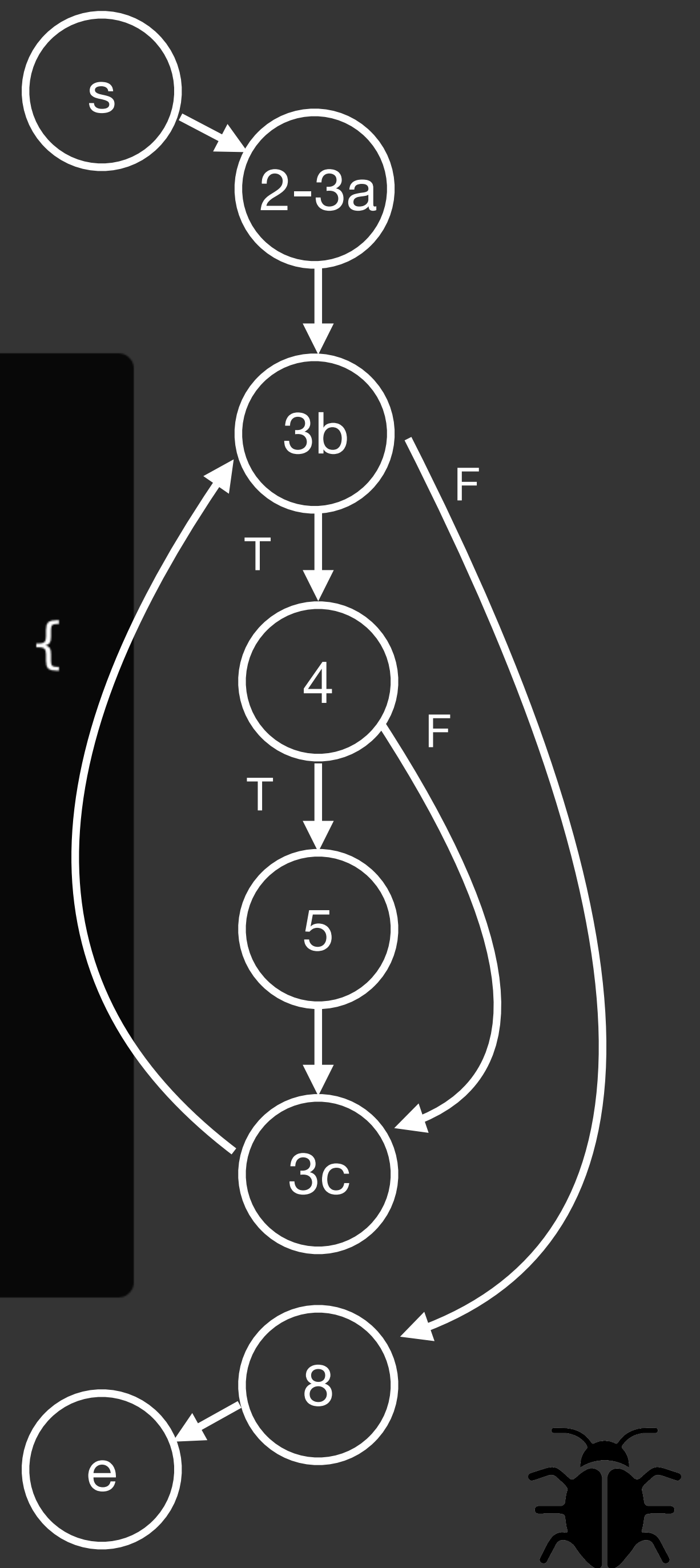


# “For” Constructs

```
for ( /*A*/ int i = 0; /*B*/ i < 10; /*C*/ i ++ ) {  
    doSomething(); /* D */  
}  
// E
```



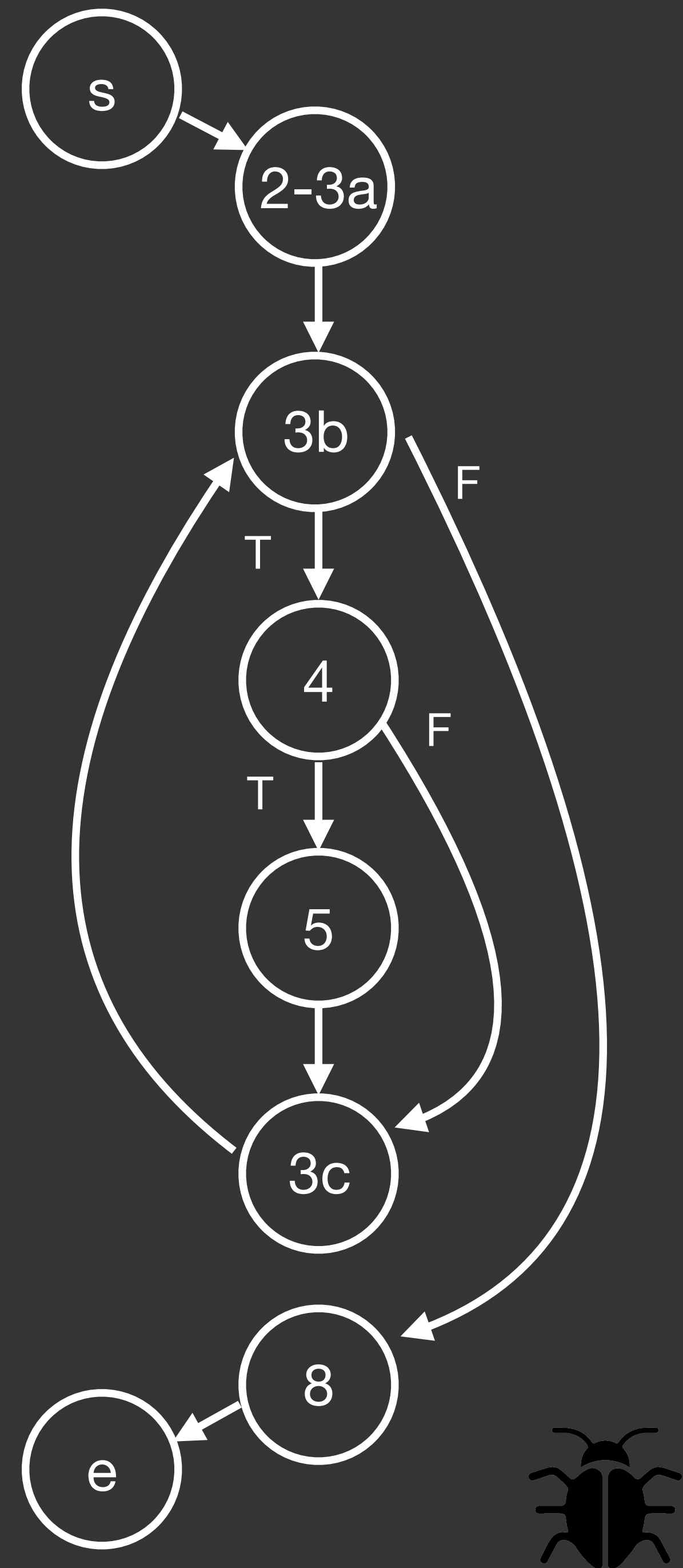
```
1 public int countZeros(int[] x) {  
2   int count = 0;  
3   for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4     if (x[i] == 0) {  
5       count ++;  
6     }  
7   }  
8   return count;  
9 }
```



# Statement Coverage

The test suite should execute all nodes of the CFG

```
1 public int countZeros(int[] x) {  
2   int count = 0;  
3   for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4     if (x[i] == 0) {  
5       count++;  
6     }  
7   }  
8   return count;  
9 }
```





# Statement Coverage

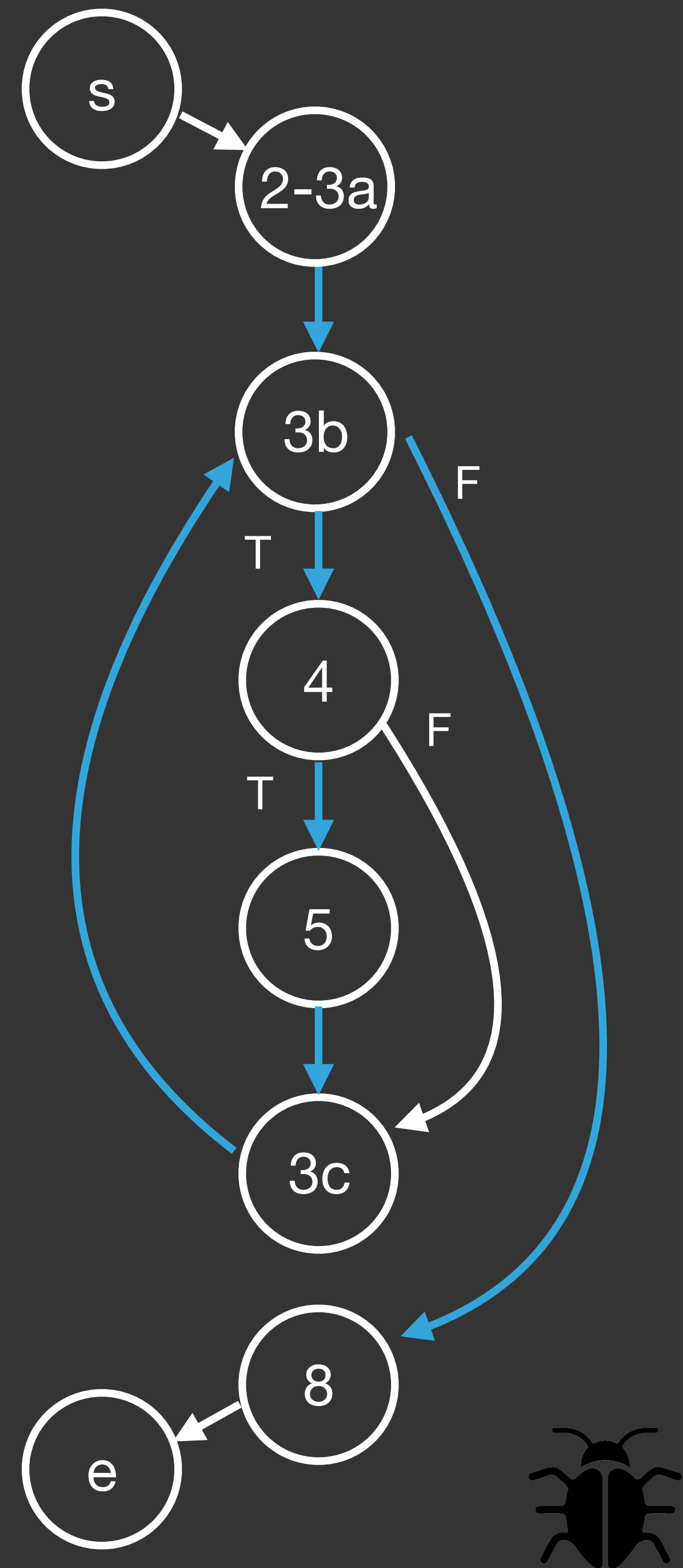
The test suite should execute all nodes of the CFG

The test case  $x = [0]$  would execute all nodes

It takes the path  $s \rightarrow 3a \rightarrow 3b \rightarrow 4 \rightarrow 5 \rightarrow 3c \rightarrow 3b \rightarrow e$

(In practice covering all nodes may need several test cases)

```
1 public int countZeros(int[] x) {  
2     int count = 0;  
3     for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4         if (x[i] == 0) {  
5             count++;  
6         }  
7     }  
8     return count;  
9 }
```



# Statement Coverage

The test suite should execute all nodes of the CFG

Also known as:

- Line Coverage
- Node Coverage
- Basic Block Coverage



# Statement Coverage

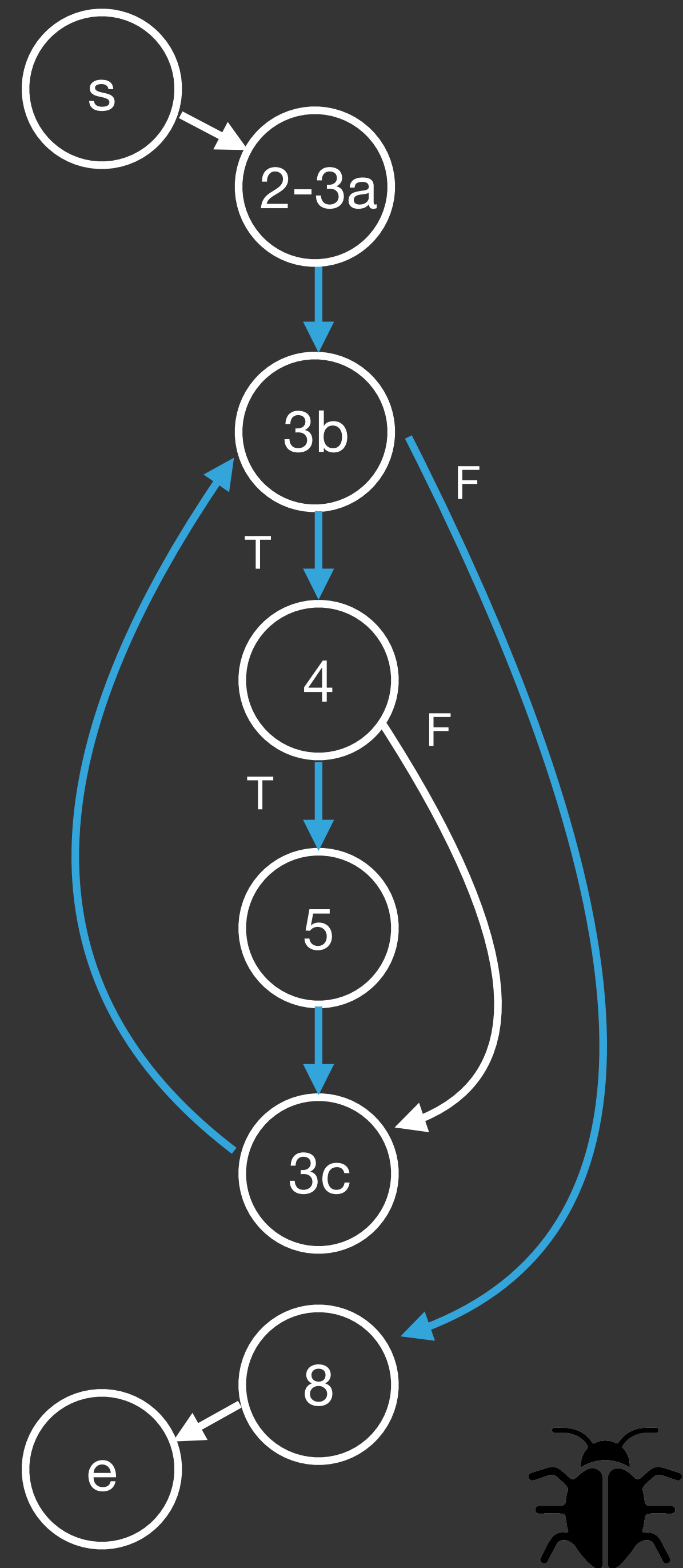
The test suite should execute all nodes of the CFG

The test case  $x = [0]$  would execute all nodes

It takes the path  $s \rightarrow 3a \rightarrow 3b \rightarrow 4 \rightarrow 5 \rightarrow 3c \rightarrow 3b \rightarrow e$

(In practice covering all nodes may need several test cases)

```
1 public int countZeros(int[] x) {  
2     int count = 0;  
3     for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4         if (x[i] == 0) {  
5             count++;  
6         }  
7     }  
8     return count;  
9 }
```



# Statement Coverage

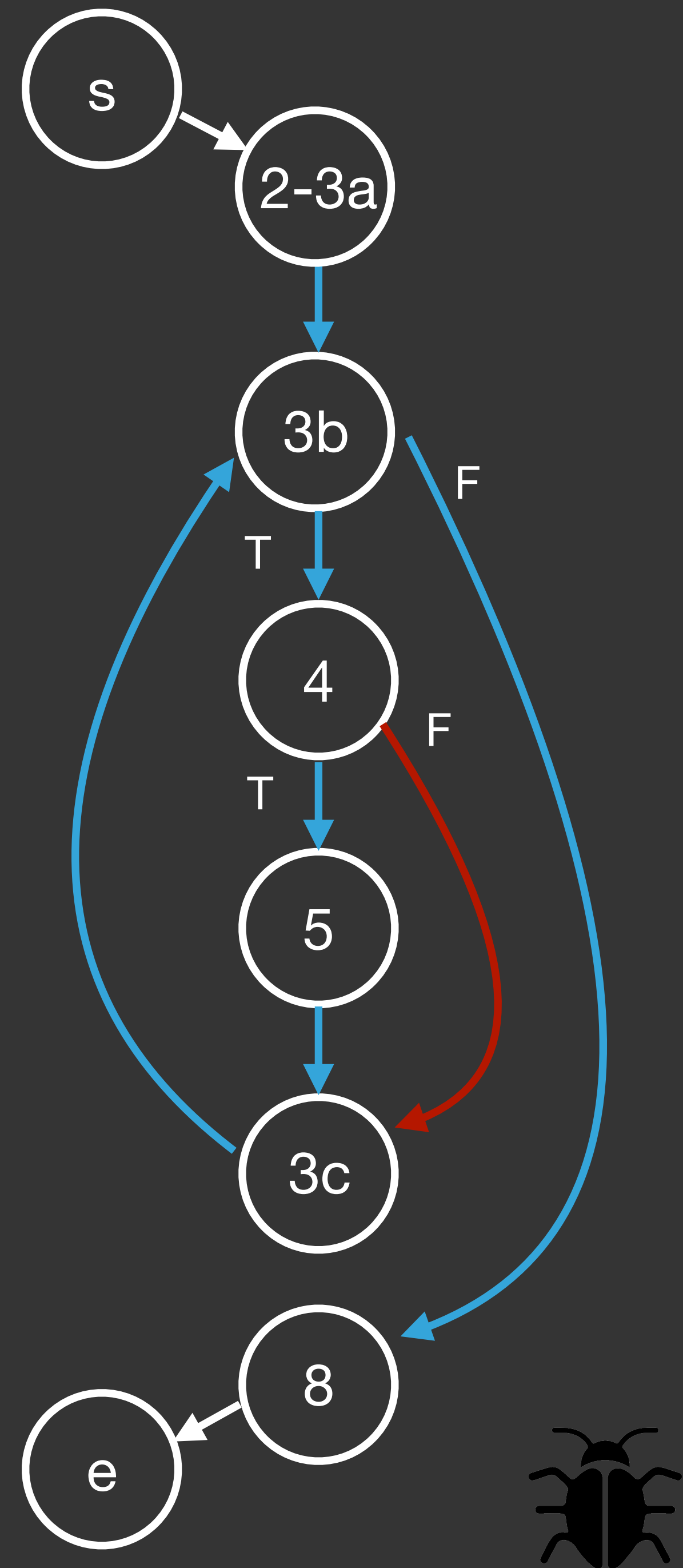
The test suite should execute all nodes of the CFG

The test case  $x = [0]$  would execute all nodes

It takes the path  $s \rightarrow 3a \rightarrow 3b \rightarrow 4 \rightarrow 5 \rightarrow 3c \rightarrow 3b \rightarrow e$

(In practice covering all nodes may need several test cases)

```
1 public int countZeros(int[] x) {  
2   int count = 0;  
3   for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4     if (x[i] == 0) {  
5       count++;  
6     }  
7   }  
8   return count;  
9 }
```



# Branch Coverage

The test suite should execute all true/false edges of the CFG



# Branch Coverage

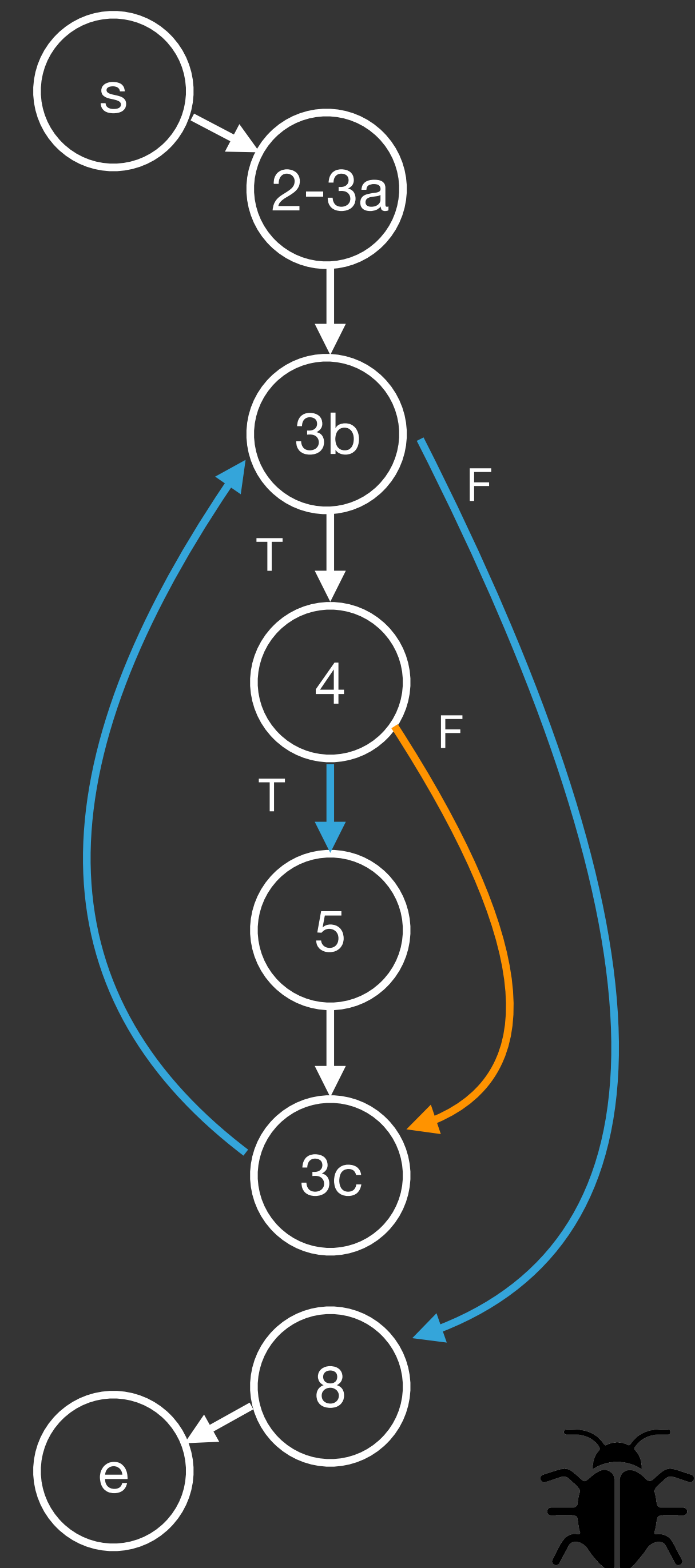
The test suite should execute all true/false edges of the CFG

The test case  $x = [0]$  and  $x = [1]$  would execute all true/false branches

$x = [0]$  takes the true branch from node 4,  $x = [1]$  takes the false branch

(Note: the test cases could be merged into one:  $x = [0, 1]$ )

```
1 public int countZeros(int[] x) {  
2     int count = 0;  
3     for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4         if (x[i] == 0) {  
5             count++;  
6         }  
7     }  
8     return count;  
9 }
```





# Branch Coverage

The test suite should execute all true/false edges of the CFG

Also known as:

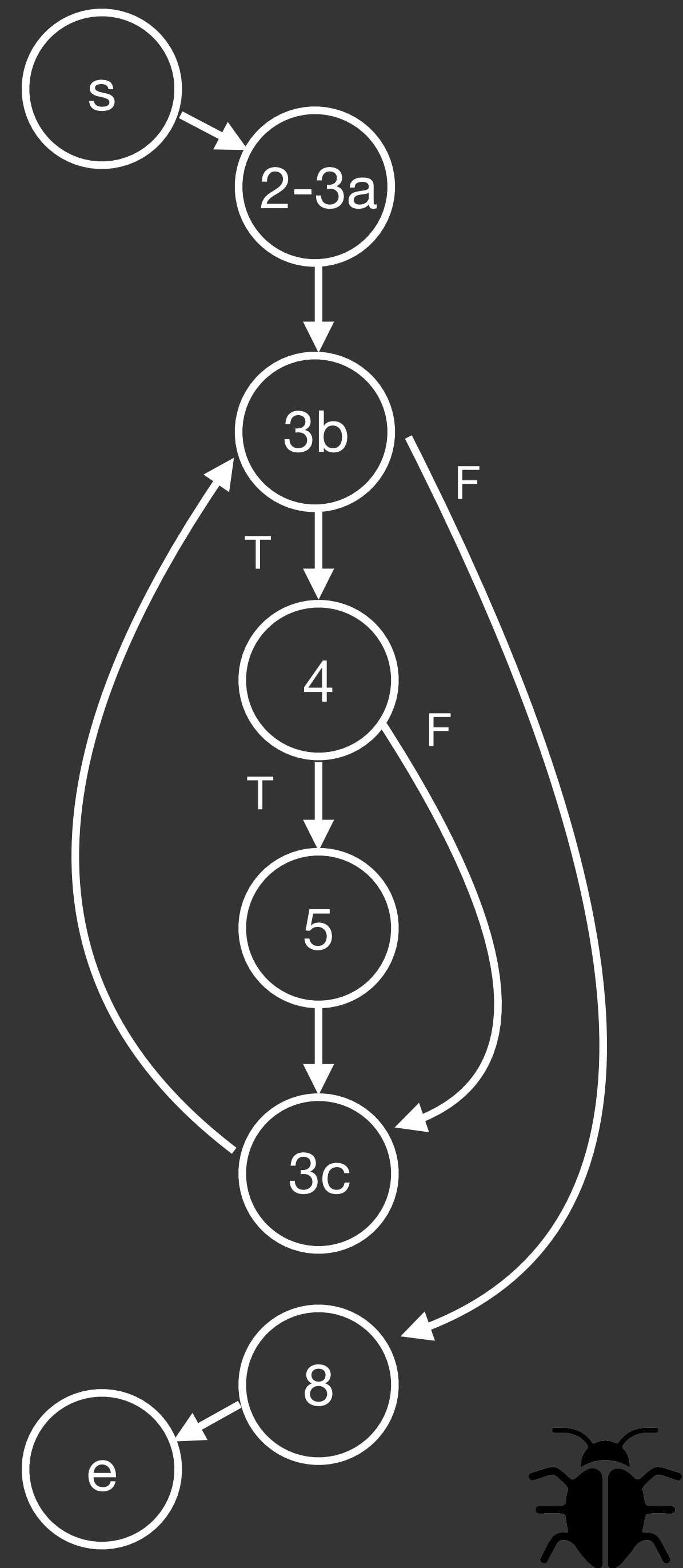
- Decision Coverage
- Predicate Coverage
- Edge Coverage



# Path Coverage

The test suite should execute all paths through the CFG

```
1 public int countZeros(int[] x) {  
2   int count = 0;  
3   for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4     if (x[i] == 0) {  
5       count++;  
6     }  
7   }  
8   return count;  
9 }
```



# Path Coverage

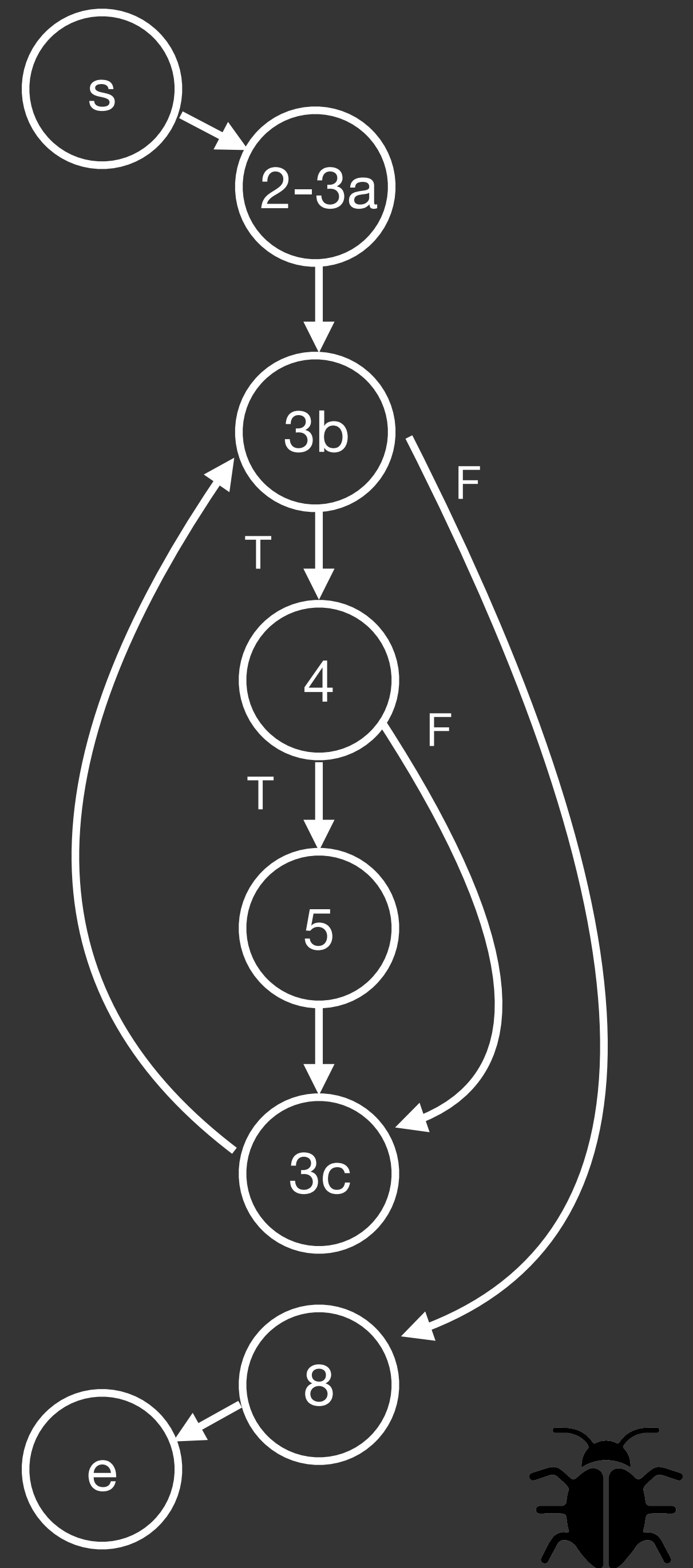
The test suite should execute all paths through the CFG

Usually not possible in practice

The number of paths through countZeros is dependent on the length of x

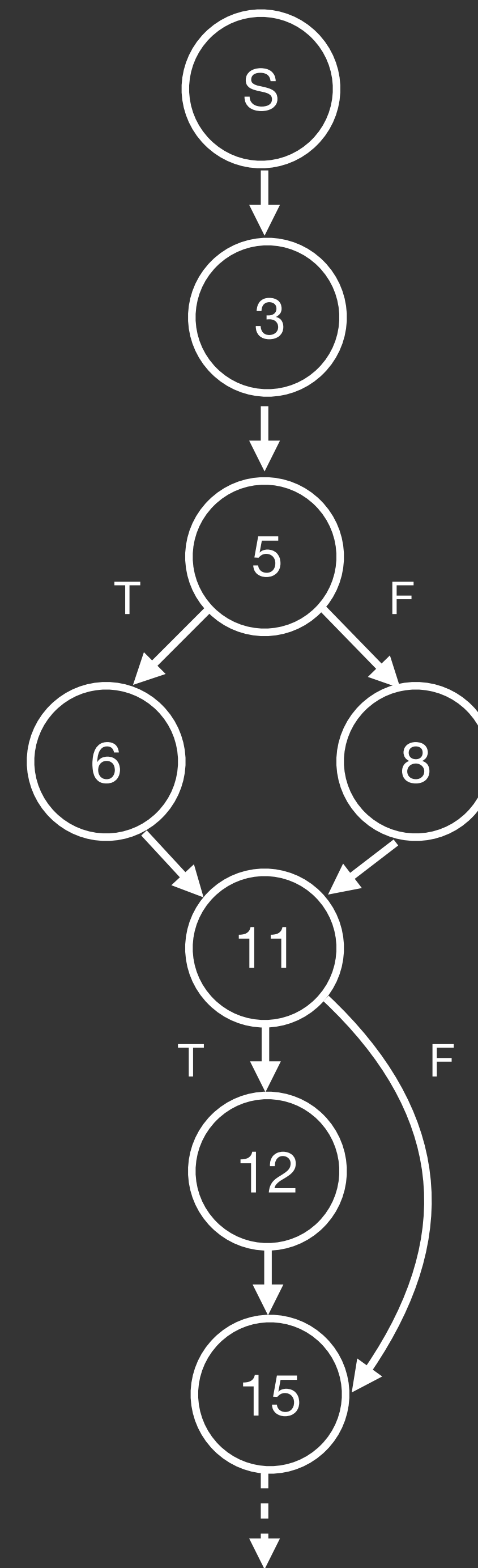
Some versions of Path Coverage concentrate on 0, 1 or more executions of every loop to mitigate potentially infinite numbers of paths

```
1 public int countZeros(int[] x) {  
2     int count = 0;  
3     for (int i=0 /* 3a */; i < x.length /* 3b */; i++ /* 3c */) {  
4         if (x[i] == 0) {  
5             count++;  
6         }  
7     }  
8     return count;  
9 }
```



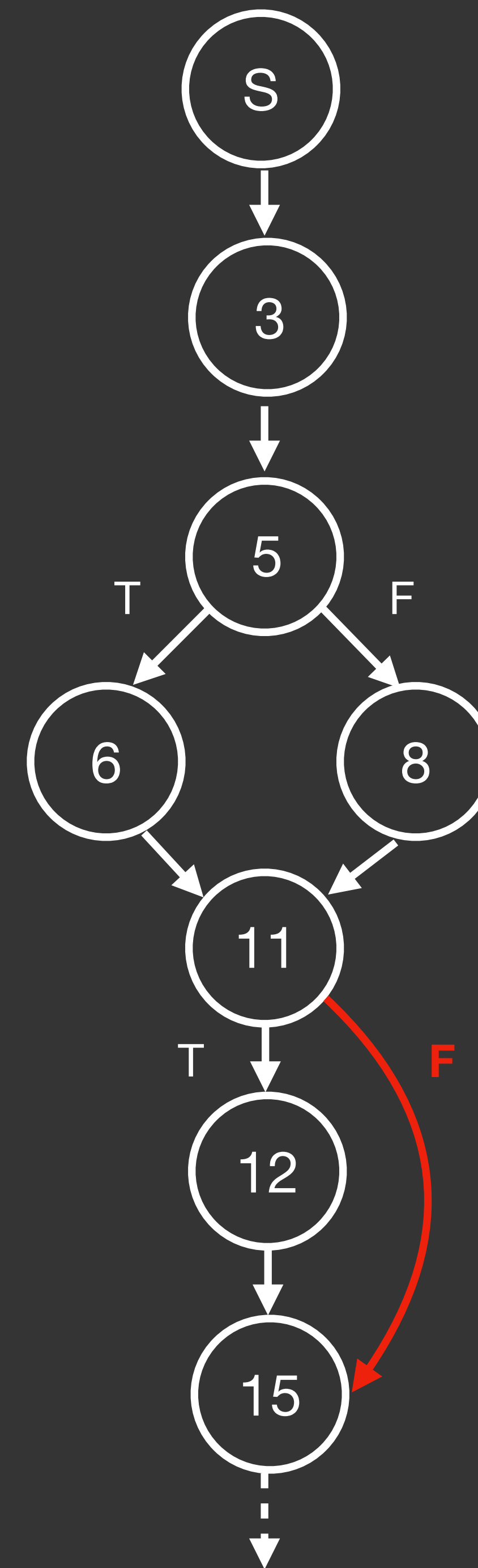
# There is a problem lurking here for Branch Coverage. What is it?

```
1 public void testMe(int x) {  
2  
3     int y = 0;  
4  
5     if (x > 0) {  
6         y = y + 1;  
7     } else {  
8         y = y + 2;  
9     }  
10  
11     if (y > 0) {  
12         y = y + 1;  
13     }  
14  
15     // ...  
}
```



# There is a problem lurking here for Branch Coverage. What is it?

```
1 public void testMe(int x) {  
2  
3     int y = 0;  
4  
5     if (x > 0) {  
6         y = y + 1;  
7     } else {  
8         y = y + 2;  
9     }  
10  
11     if (y > 0) {  
12         y = y + 1;  
13     }  
14  
15     // ...  
}
```



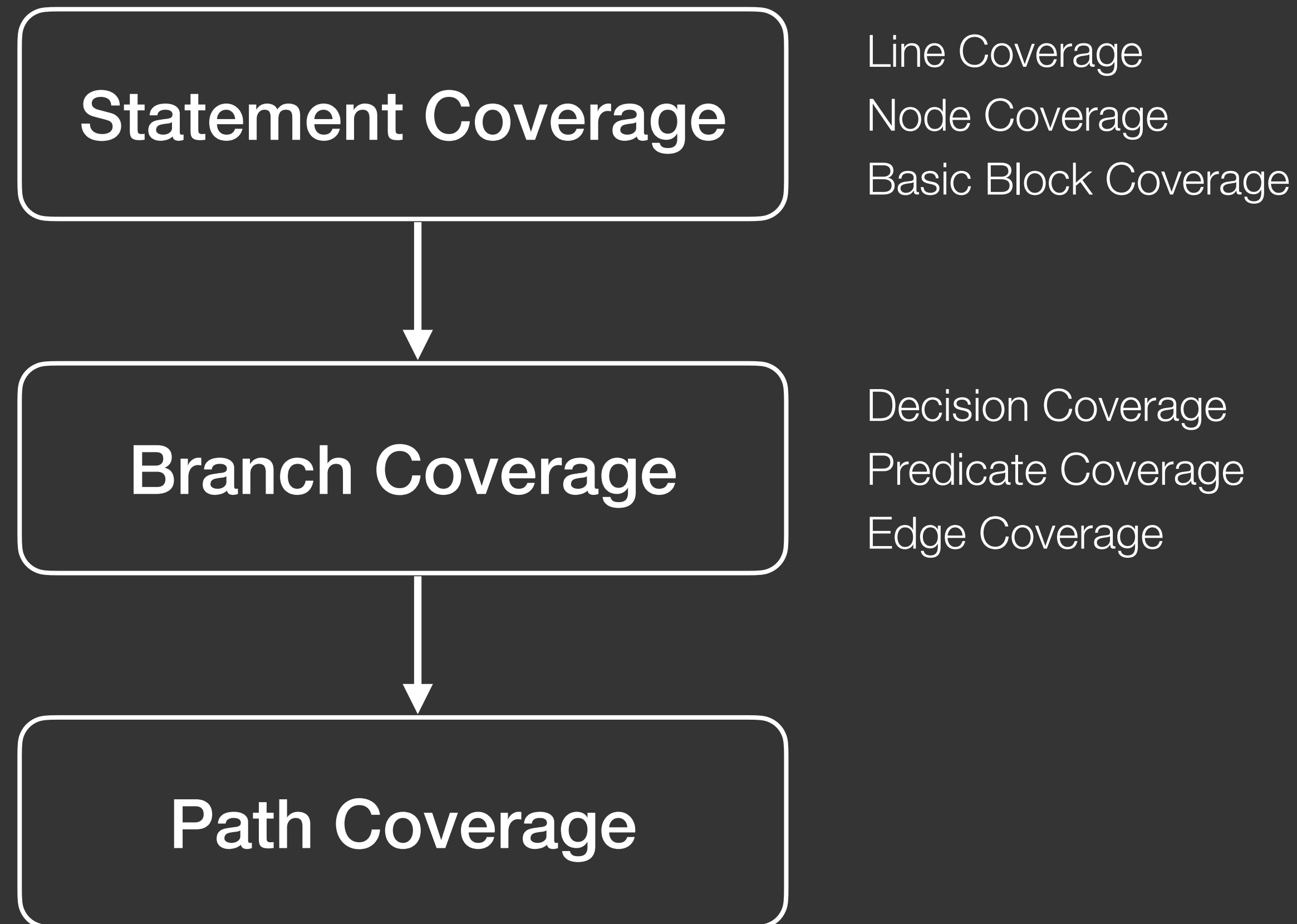
# Coverage Criteria are Subject To Infeasible Test Requirements

- This *can* also happen with Statement Coverage
  - Infeasible statements point to dead code
- It is *more likely* to happen with Branch Coverage
  - Infeasible branches point to redundant decisions in the code
- It is *very likely* with Path Coverage
  - Infeasible paths are not necessarily the result of redundancy
  - Not all the paths through the CFG are legitimately possible in the actual code





# Subsumption of Structural Coverage Criteria



# When to Use Structural Coverage

- Structural coverage level is a **useful metric** to understand how much of your code is executed by your test suite.
- **Common Rationale:** you wouldn't want to release parts of your code that weren't exercised at least once by at least one test
  - As such Statement/Line Coverage is a commonly used metric
  - But Branch Coverage is stronger, and obtainable without much more additional effort
  - Path Coverage is less common and often intractable

