COM 3529
Software Testing & Analysis

Professor Phil McMinn

5.3 Symbolic Execution

dynamic

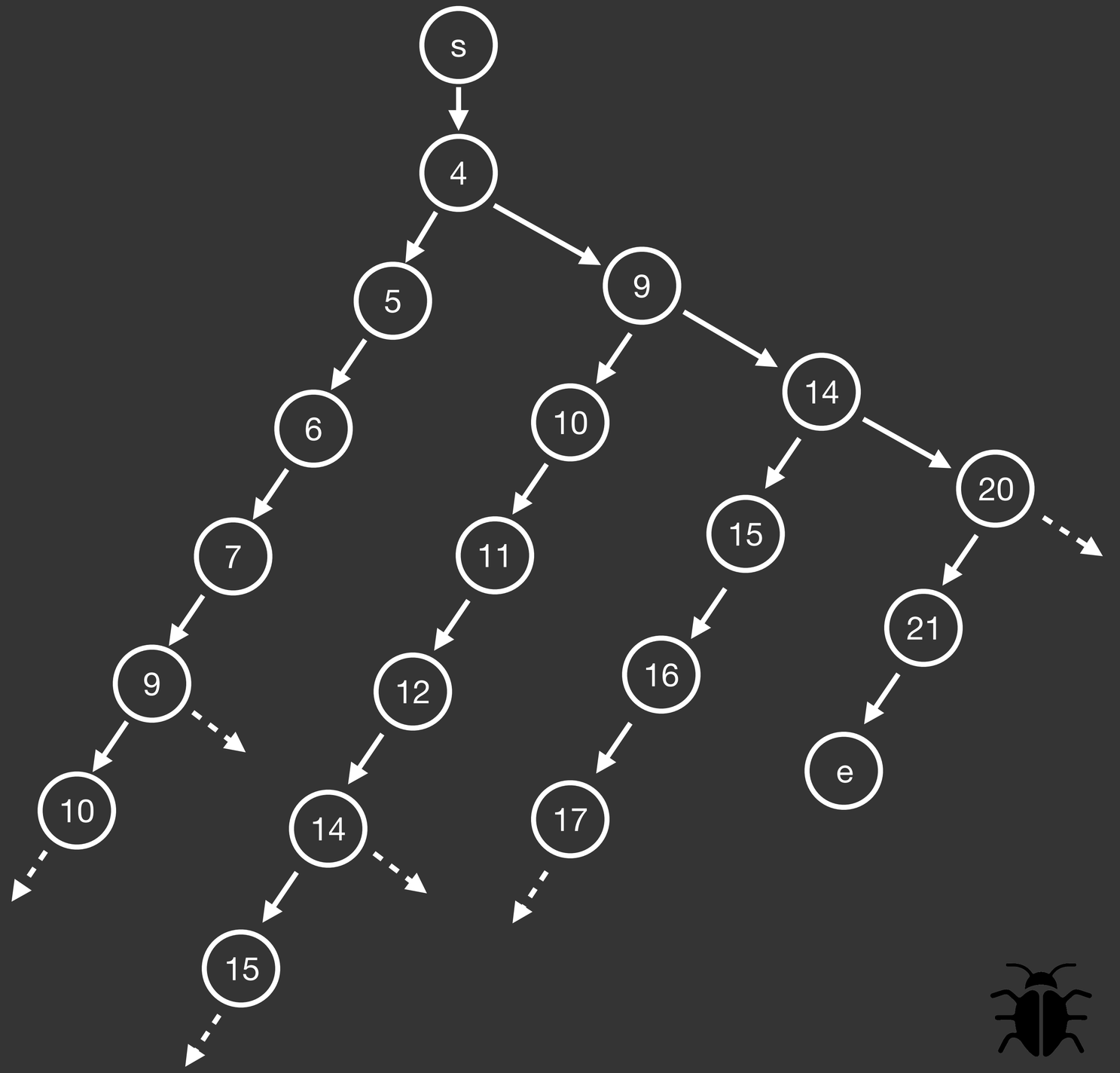
(execute the program under test)

static

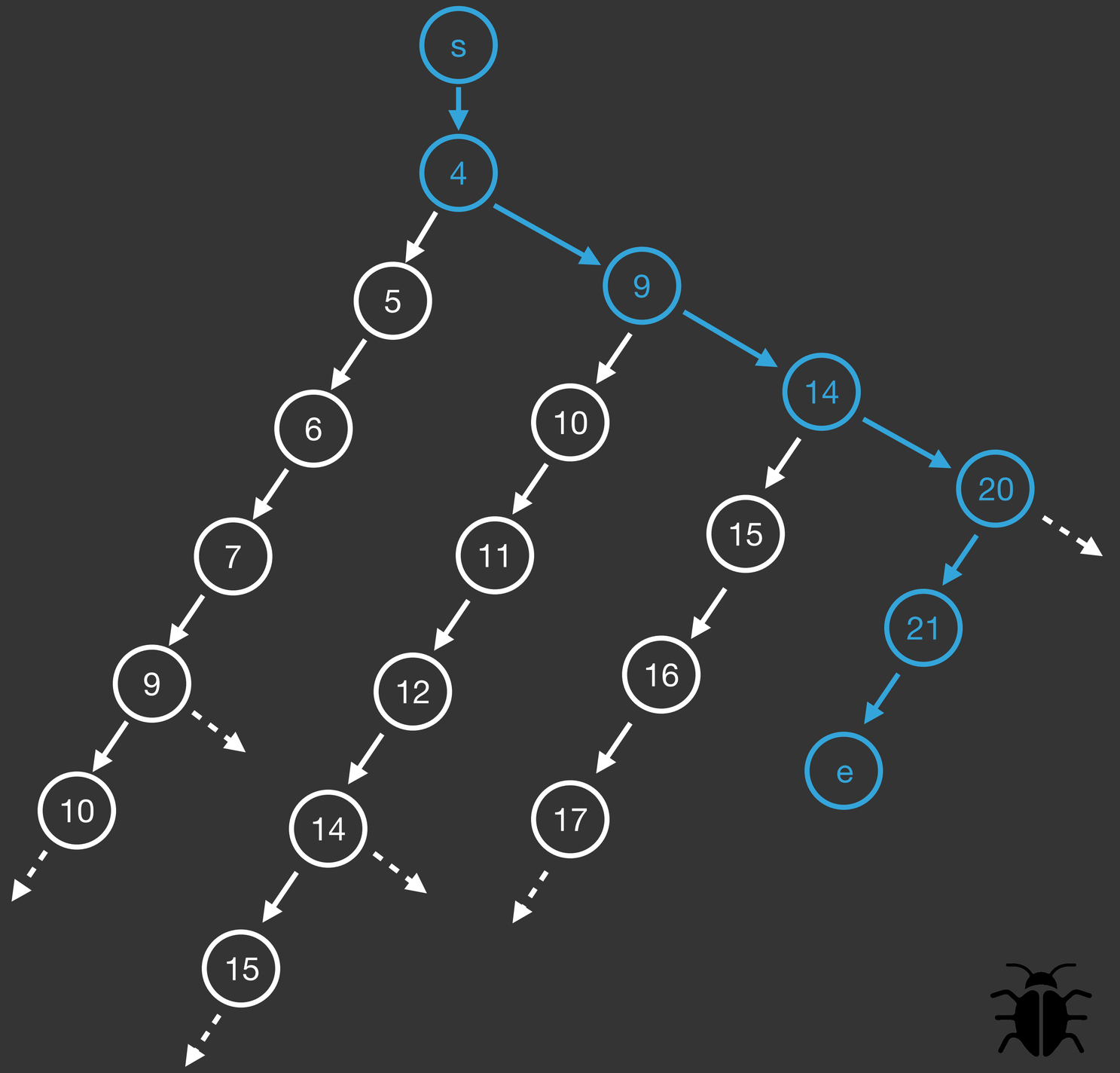
(examine the program without fully executing it)

symbolic execution

```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```

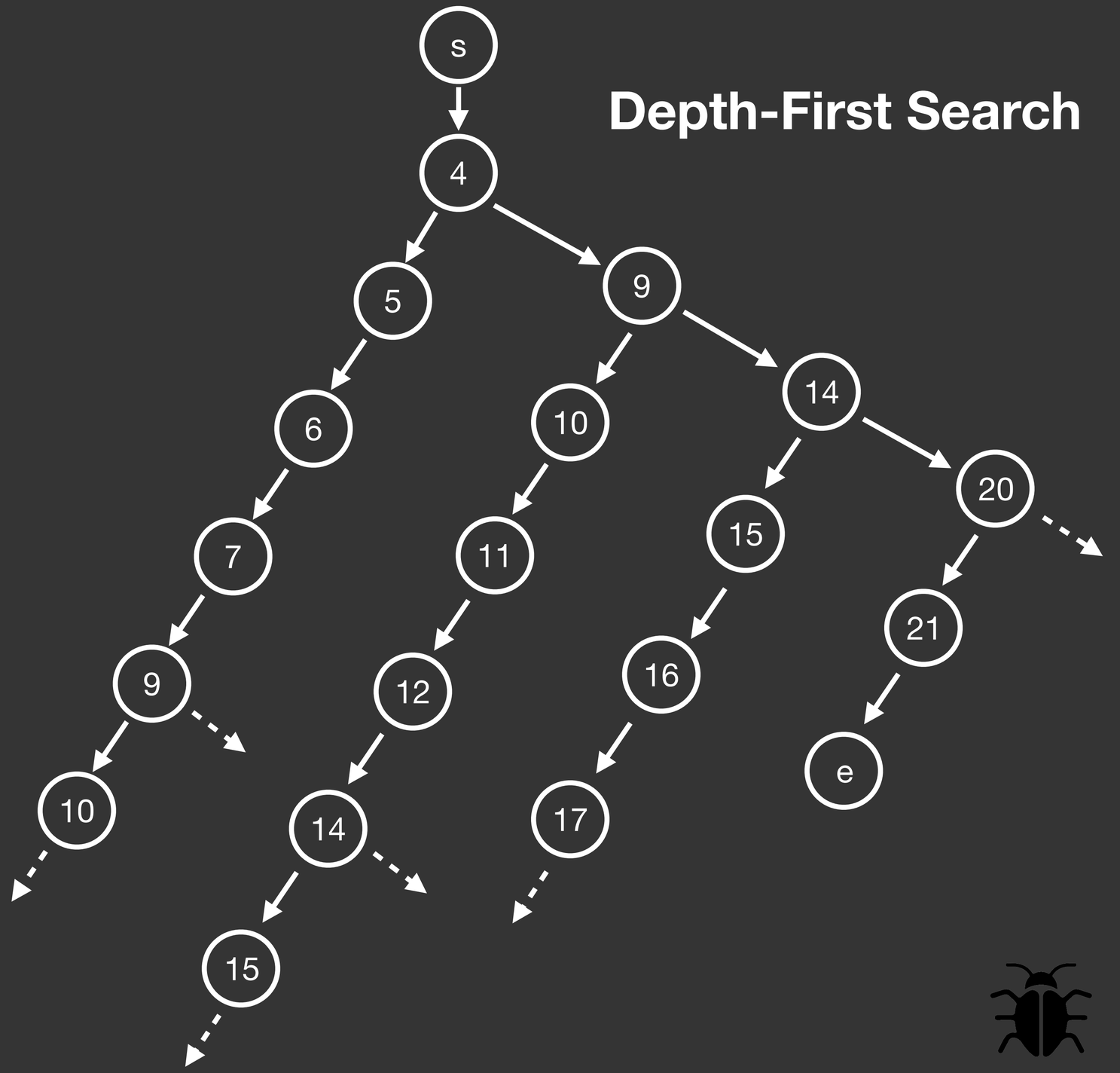


```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```



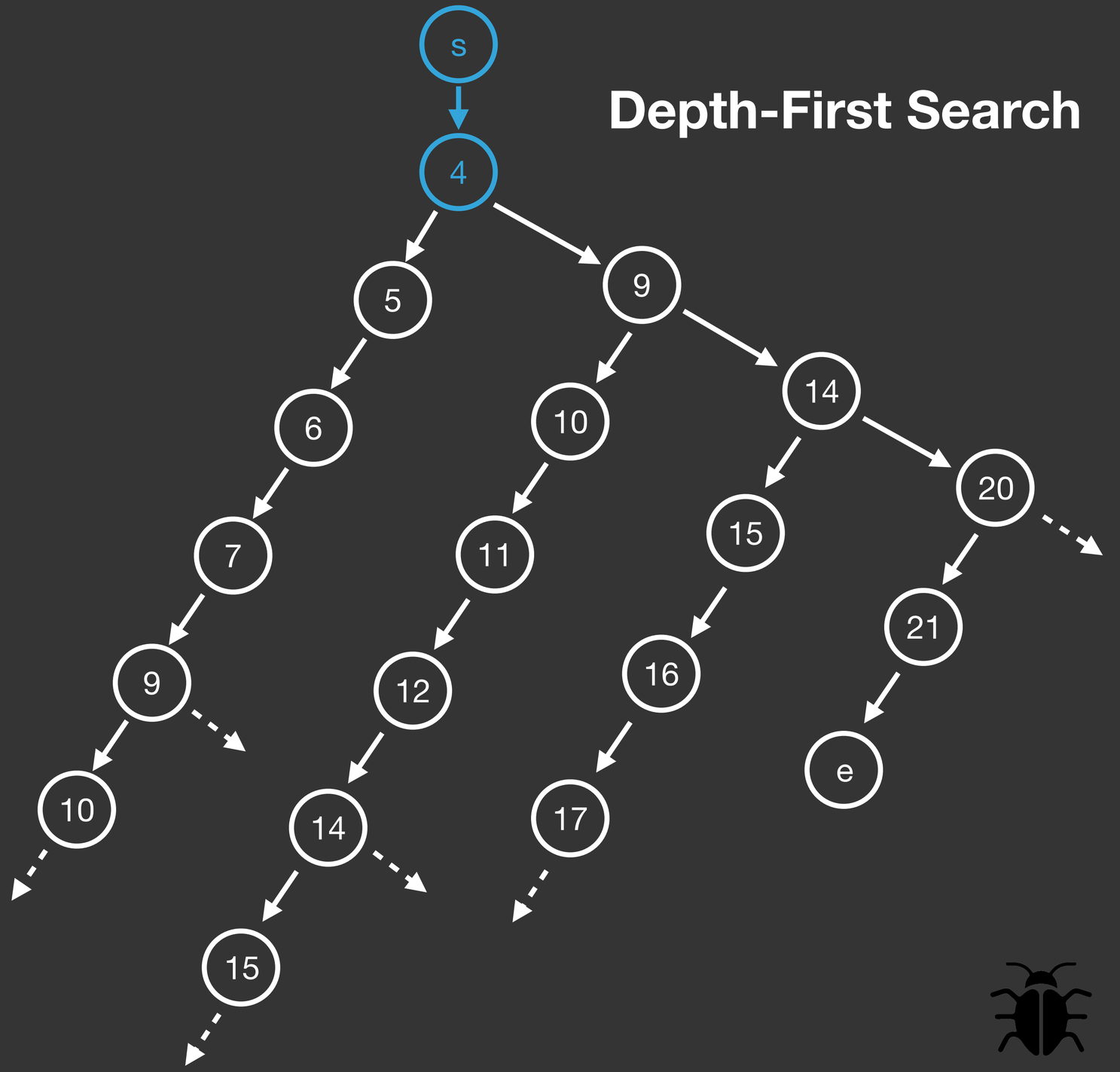
```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```

Depth-First Search



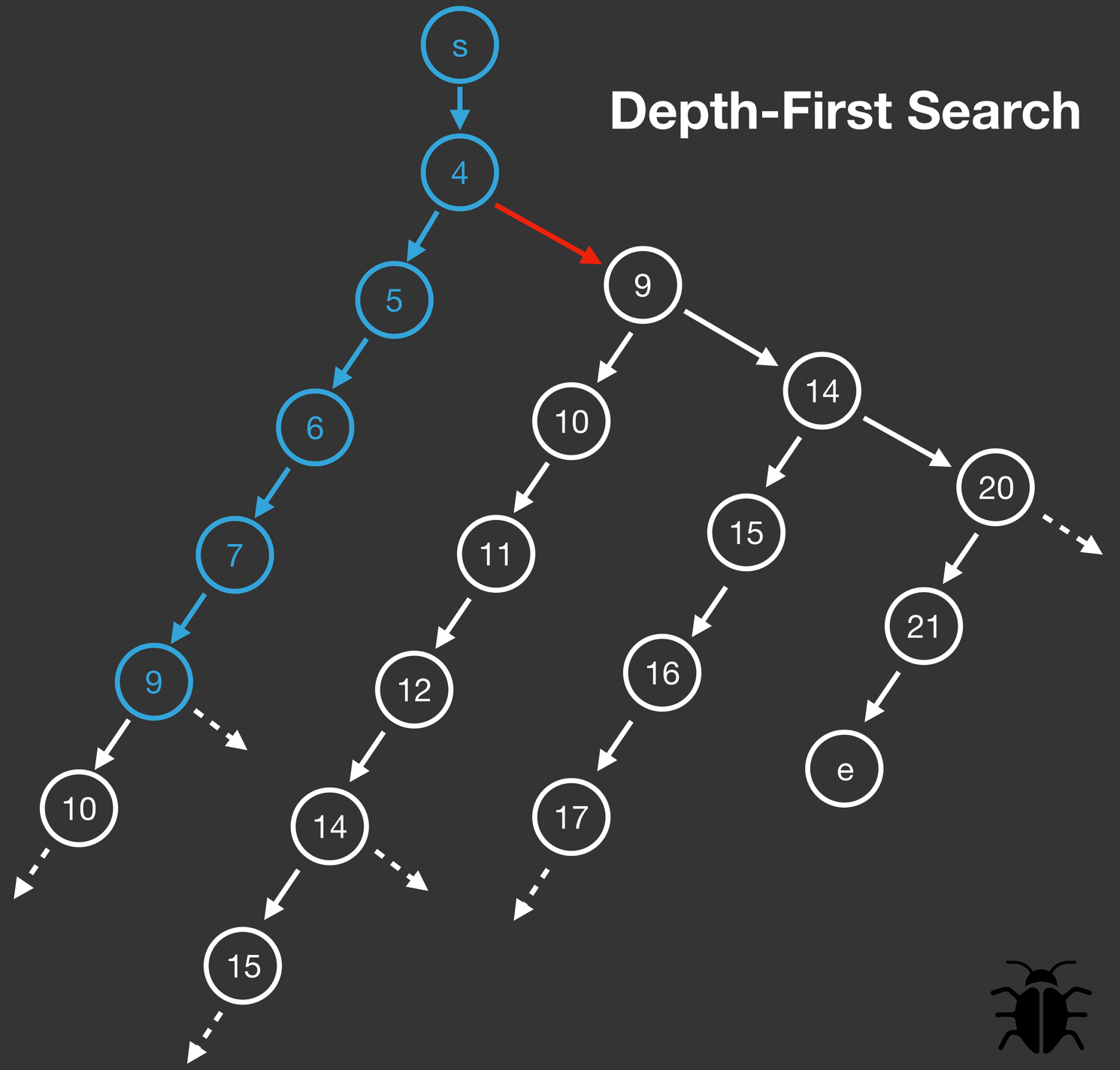
```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```

Depth-First Search



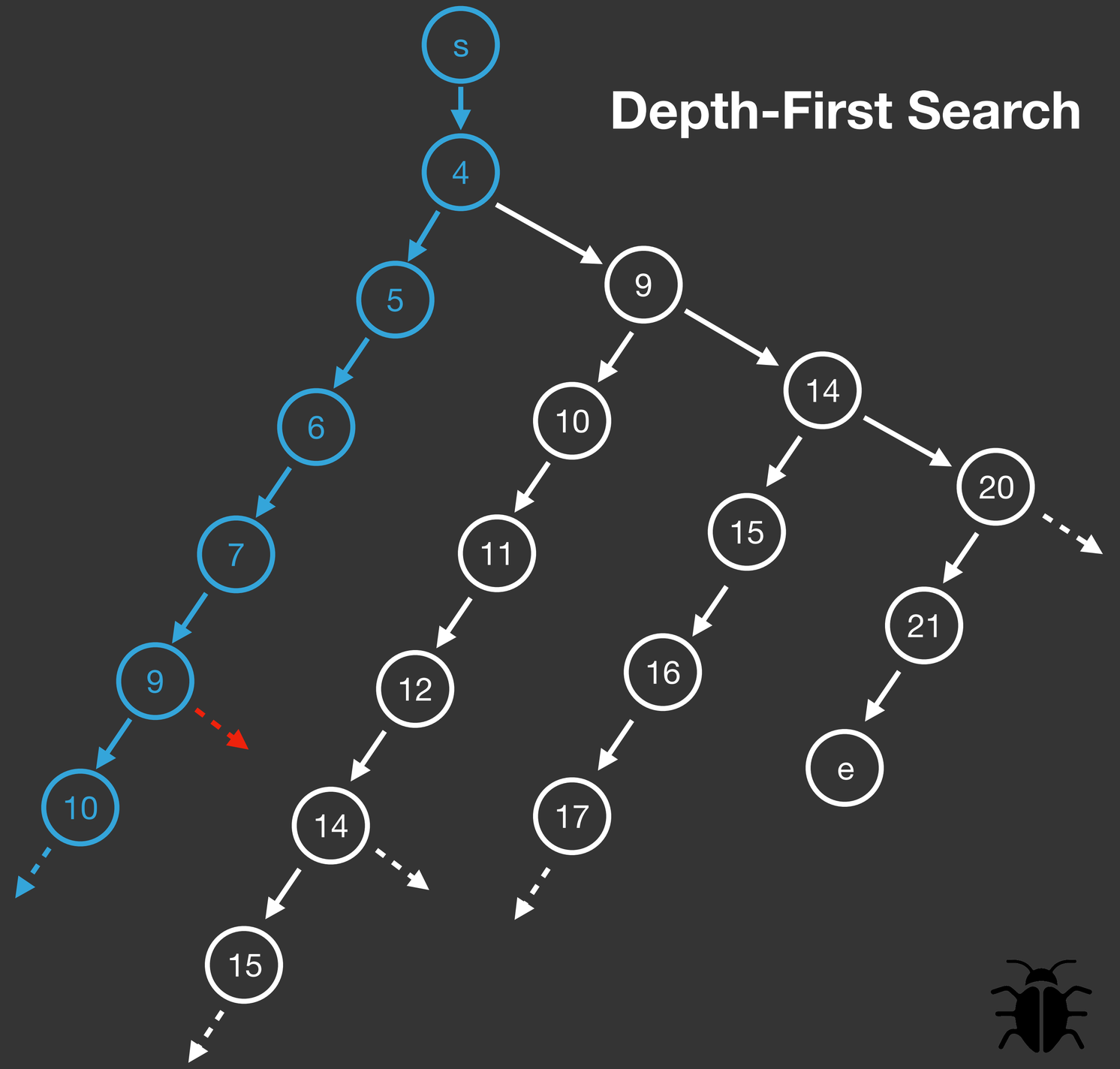
```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```

Depth-First Search



```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```

Depth-First Search



CFG Node	Code	Symbolic State				Path Condition
		side1	side2	side3	temp	
s						
4	`if (side1 > side2)` *as TRUE*					
5	`int temp = side1`					
6	`side1 = side2`					
7	`side2 = temp`					
9	`if (side1 > side3)` *as TRUE*					
10	`int temp = side1`					
11	`side1 = side3`					
12	`side3 = temp`					
14	`if (side2 > side3)` *as TRUE*					
15	`int temp = side2`					
16	`side2 = side3`					
17	`side3 = temp`					
20	`if (side1 + side2 <= side3)` *as TRUE*					
21	`...`					
e	`...`					

CFG Node	Code	Symbolic State				Path Condition
		`side1`	`side2`	`side3`	`temp`	
s		α	β	γ	-	*true*
4	`if (side1 > side2)` *as TRUE*					
5	`int temp = side1`					
6	`side1 = side2`					
7	`side2 = temp`					
9	`if (side1 > side3)` *as TRUE*					
10	`int temp = side1`					
11	`side1 = side3`					
12	`side3 = temp`					
14	`if (side2 > side3)` *as TRUE*					
15	`int temp = side2`					
16	`side2 = side3`					
17	`side3 = temp`					
20	`if (side1 + side2 <= side3)` *as TRUE*					
21	`...`					
e	`...`					

CFG Node	Code	Symbolic State				Path Condition
		`side1`	`side2`	`side3`	`temp`	
s		α	β	γ	-	*true*
4	`if (side1 > side2)` *as TRUE*	α	β	γ	α	$\alpha > \beta$
5	`int temp = side1`	α	β	γ	α	$\alpha > \beta$
6	`side1 = side2`	β	β	γ	α	$\alpha > \beta$
7	`side2 = temp`	β	α	γ	α	$\alpha > \beta$
9	`if (side1 > side3)` *as TRUE*					
10	`int temp = side1`					
11	`side1 = side3`					
12	`side3 = temp`					
14	`if (side2 > side3)` *as TRUE*					
15	`int temp = side2`					
16	`side2 = side3`					
17	`side3 = temp`					
20	`if (side1 + side2 <= side3)` *as TRUE*					
21	`...`					
e	`...`					

CFG Node	Code	Symbolic State				Path Condition
		`side1`	`side2`	`side3`	`temp`	
s		α	β	γ	-	*true*
4	`if (side1 > side2)` *as TRUE*	α	β	γ	α	$\alpha > \beta$
5	`int temp = side1`	α	β	γ	α	$\alpha > \beta$
6	`side1 = side2`	β	β	γ	α	$\alpha > \beta$
7	`side2 = temp`	β	α	γ	α	$\alpha > \beta$
9	`if (side1 > side3)` *as TRUE*	β	α	γ	-	$\alpha > \beta \wedge \beta > \gamma$
10	`int temp = side1`	β	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
11	`side1 = side3`	γ	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
12	`side3 = temp`	γ	α	β	β	$\alpha > \beta \wedge \beta > \gamma$
14	`if (side2 > side3)` *as TRUE*					
15	`int temp = side2`					
16	`side2 = side3`					
17	`side3 = temp`					
20	`if (side1 + side2 <= side3)` *as TRUE*					
21	`...`					
e	`...`					

CFG Node	Code	Symbolic State				Path Condition
		`side1`	`side2`	`side3`	`temp`	
s		α	β	γ	-	*true*
4	`if (side1 > side2)` *as TRUE*	α	β	γ	α	$\alpha > \beta$
5	`int temp = side1`	α	β	γ	α	$\alpha > \beta$
6	`side1 = side2`	β	β	γ	α	$\alpha > \beta$
7	`side2 = temp`	β	α	γ	α	$\alpha > \beta$
9	`if (side1 > side3)` *as TRUE*	β	α	γ	-	$\alpha > \beta \wedge \beta > \gamma$
10	`int temp = side1`	β	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
11	`side1 = side3`	γ	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
12	`side3 = temp`	γ	α	β	β	$\alpha > \beta \wedge \beta > \gamma$
14	`if (side2 > side3)` *as TRUE*	γ	α	β	-	$\alpha > \beta \wedge \beta > \gamma$ (no change)
15	`int temp = side2`	γ	α	β	α	$\alpha > \beta \wedge \beta > \gamma$
16	`side2 = side3`	γ	β	β	α	$\alpha > \beta \wedge \beta > \gamma$
17	`side3 = temp`	γ	β	α	α	$\alpha > \beta \wedge \beta > \gamma$
20	`if (side1 + side2 <= side3)` *as TRUE*					
21	`...`					
e	`...`					

CFG Node	Code	Symbolic State				Path Condition
		`side1`	`side2`	`side3`	`temp`	
s		α	β	γ	-	*true*
4	`if (side1 > side2)` *as TRUE*	α	β	γ	α	$\alpha > \beta$
5	`int temp = side1`	α	β	γ	α	$\alpha > \beta$
6	`side1 = side2`	β	β	γ	α	$\alpha > \beta$
7	`side2 = temp`	β	α	γ	α	$\alpha > \beta$
9	`if (side1 > side3)` *as TRUE*	β	α	γ	-	$\alpha > \beta \wedge \beta > \gamma$
10	`int temp = side1`	β	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
11	`side1 = side3`	γ	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
12	`side3 = temp`	γ	α	β	β	$\alpha > \beta \wedge \beta > \gamma$
14	`if (side2 > side3)` *as TRUE*	γ	α	β	-	$\alpha > \beta \wedge \beta > \gamma$ (no change)
15	`int temp = side2`	γ	α	β	α	$\alpha > \beta \wedge \beta > \gamma$
16	`side2 = side3`	γ	β	β	α	$\alpha > \beta \wedge \beta > \gamma$
17	`side3 = temp`	γ	β	α	α	$\alpha > \beta \wedge \beta > \gamma$
20	`if (side1 + side2 <= side3)` *as TRUE*	γ	β	α	α	$\alpha > \beta \wedge \beta > \gamma \wedge \gamma + \beta \leq \alpha$
21	`...`					
e	`...`					

CFG Node	Code	Symbolic State				Path Condition
		`side1`	`side2`	`side3`	`temp`	
s		α	β	γ	-	*true*
4	`if (side1 > side2)` *as TRUE*	α	β	γ	α	$\alpha > \beta$
5	`int temp = side1`	α	β	γ	α	$\alpha > \beta$
6	`side1 = side2`	β	β	γ	α	$\alpha > \beta$
7	`side2 = temp`	β	α	γ	α	$\alpha > \beta$
9	`if (side1 > side3)` *as TRUE*	β	α	γ	-	$\alpha > \beta \wedge \beta > \gamma$
10	`int temp = side1`	β	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
11	`side1 = side3`	γ	α	γ	β	$\alpha > \beta \wedge \beta > \gamma$
12	`side3 = temp`	γ	α	β	β	$\alpha > \beta \wedge \beta > \gamma$
14	`if (side2 > side3)` *as TRUE*	γ	α	β	-	$\alpha > \beta \wedge \beta > \gamma$ (no change)
15	`int temp = side2`	γ	α	β	α	$\alpha > \beta \wedge \beta > \gamma$
16	`side2 = side3`	γ	β	β	α	$\alpha > \beta \wedge \beta > \gamma$
17	`side3 = temp`	γ	β	α	α	$\alpha > \beta \wedge \beta > \gamma$
20	`if (side1 + side2 <= side3)` *as TRUE*	γ	β	α	α	$\alpha > \beta \wedge \beta > \gamma \wedge \gamma + \beta \leq \alpha$
21	...					
e	...					

α =`side 1`, β =`side2`, and γ =`side3`

$$\texttt{side1} > \texttt{side2} \wedge \texttt{side2} > \texttt{side3} \wedge \texttt{side3} + \texttt{side2} \leq \texttt{side1}$$

e.g. `side1`= 3, `side2`= 2, and `side3`= 1

Dynamic Symbolic Execution

Concolic Execution

Dynamic Symbolic Execution

Concrete Symbolic Execution

Dynamic Symbolic Execution

DSE

Concrete Symbolic Execution

```java
public static Type classify(int side1, int side2, int side3) {
    Type type;

    if (side1 > side2) {
        int temp = side1;
        side1 = side2;
        side2 = temp;
    }
    if (side1 > side3) {
        int temp = side1;
        side1 = side3;
        side3 = temp;
    }
    if (side2 > side3) {
        int temp = side2;
        side2 = side3;
        side3 = temp;
    }

    if (side1 + side2 <= side3) {
        type = Type.INVALID;
    } else {
        type = Type.SCALENE;
        if (side1 == side2) {
            if (side2 == side3) {
                type = Type.EQUILATERAL;
            }
        } else {
            if (side2 == side3) {
                type = Type.ISOSCELES;
            }
        }
    }
    return type;
}
```

side1 = 4
side2 = 2
side3 = 3

```
1  public static Type classify(int side1, int side2, int side3) {
2      Type type;
3
4      if (side1 > side2) {
5          int temp = side1;
6          side1 = side2;
7          side2 = temp;
8      }
9      if (side1 > side3) {
10         int temp = side1;
11         side1 = side3;
12         side3 = temp;
13     }
14     if (side2 > side3) {
15         int temp = side2;
16         side2 = side3;
17         side3 = temp;
18     }
19
20     if (side1 + side2 <= side3) {
21         type = Type.INVALID;
22     } else {
23         type = Type.SCALENE;
24         if (side1 == side2) {
```

```
side1 = 4
side2 = 2
side3 = 3
```

DSE makes Symbolic Execution faster and more efficient

Dealing with Path Explosion

Tools

EvoSuite (Java) https://www.evosuite.org

KLEE (C) https://klee.github.io