



CISC/CMPE 327 Software Quality Assurance

Queen's University, 2019–fall

Lecture #16 White Box Testing – Path & Data Coverage

White Box Testing

- Today we continue our look at **white box** testing with another **code coverage** method, and some **data coverage** methods
- We'll look at:
 - Code coverage testing:
 - **Path** coverage
 - Data coverage testing
 - **Data value** coverage
 - **Data flow** coverage
 - **Data interface** coverage

Execution Paths

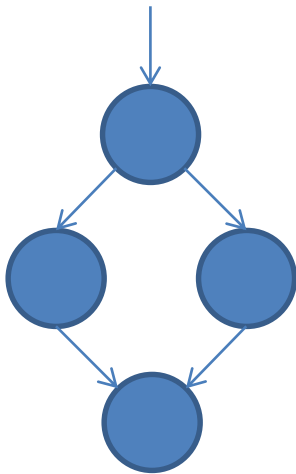
- An **execution path** is a sequence of executed statements starting at the **entry** to the unit (usually the first statement) and ending at the **exit** from the unit (usually the last statement)
- Two paths are **independent** if there is **at least one statement on one path which is not executed on the other**
- **Path analysis** (also known as **cyclomatic complexity*** analysis) identifies all the **independent paths** through a unit

* a **code metric** we will look at later in the course

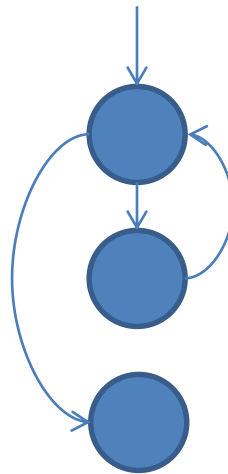
Execution Path Analysis

- **Flow Graphs**

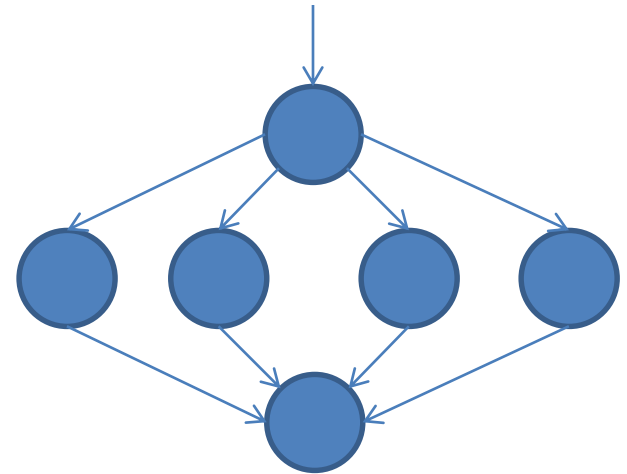
- It is easiest to do path analysis if we look at the execution **flow graph** of the program or unit
- The flow graph shows program **control flow** between **basic blocks**



if-then-else



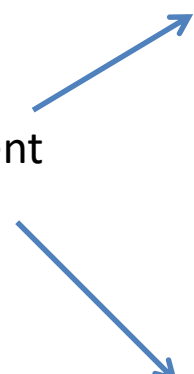
while



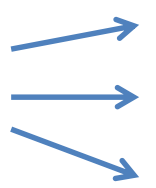
switch

Example: Path Analysis

Each statement
sequence
is a node



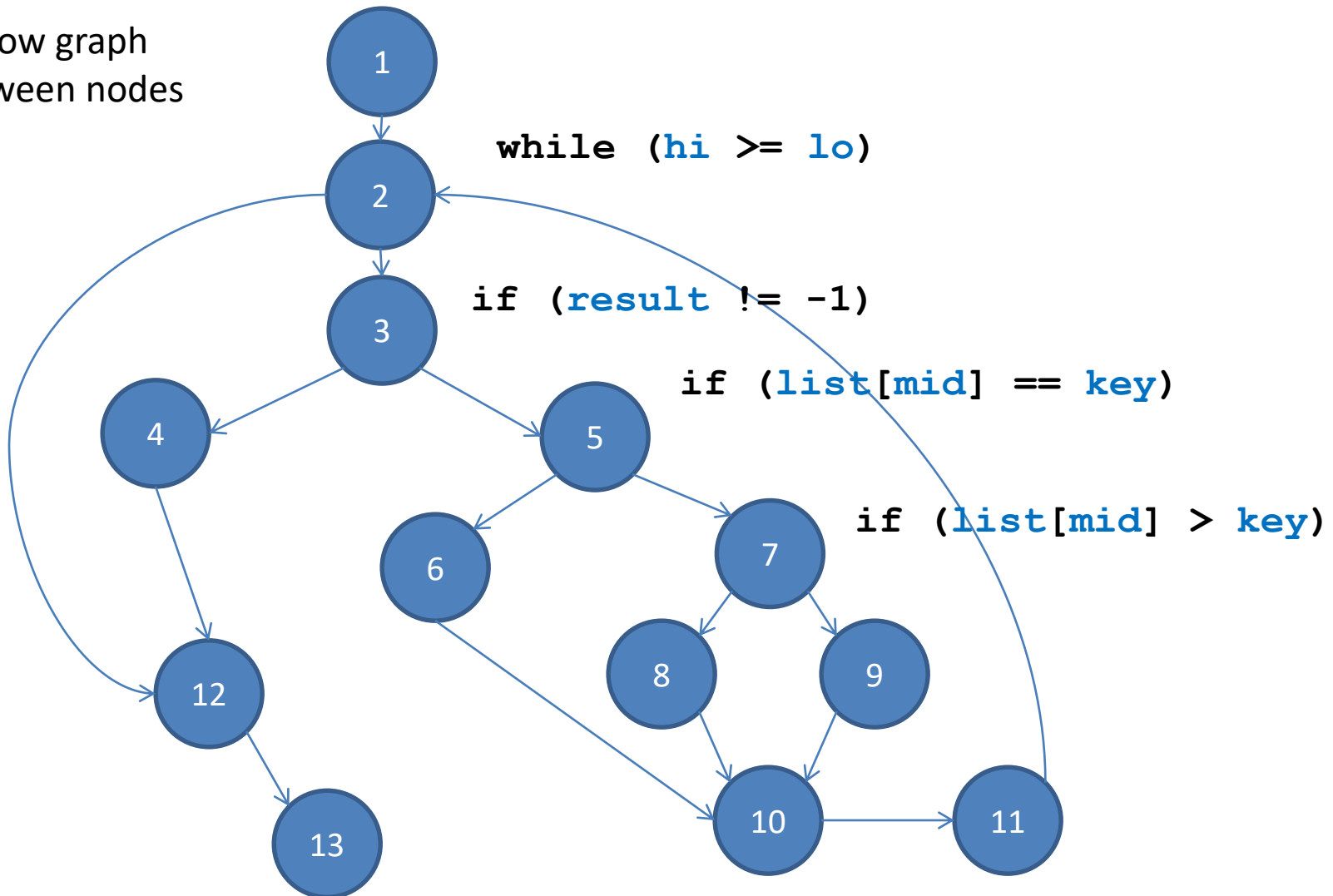
Each end
of scope
is a node



```
static int find (int list[], int n, int key) {  
    // binary search of ordered list  
    int lo = 0;  
    int hi = n - 1;  
    int result = -1;  
  
    while (hi >= lo) {  
        if (result != -1)  
            break;  
        else {  
            final int mid = (lo + hi) / 2;  
            if (list[mid] == key)  
                result = mid;  
            else if (list[mid] > key)  
                hi = mid - 1;  
            else // list[mid] < key  
                lo = mid + 1;  
        }  
    }  
    return result;  
}
```

Example: Path Analysis

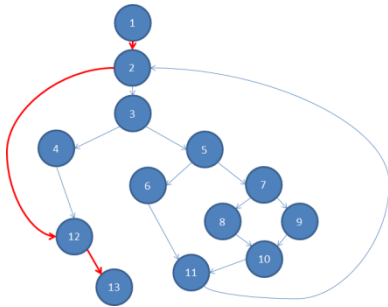
Flow graph
between nodes



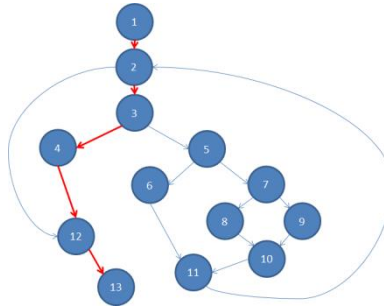
Example: Path Analysis

- Independent Paths

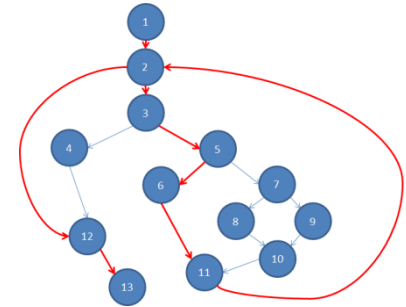
- This program has only **five** independent paths (CC = 5)



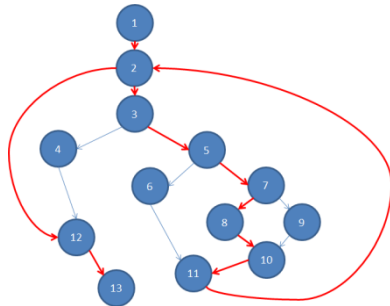
P1: 1, 2, 12, 13



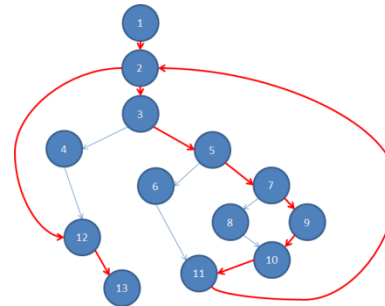
P2: 1, 2, 3, 4, 12, 13



P3: 1, 2, 3, 5, 6, 10, 11, 2, 12, 13



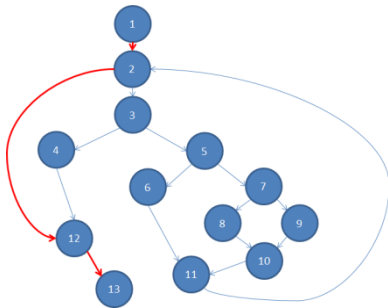
P4: 1, 2, 3, 7, 8, 10, 11, 2, 12, 13



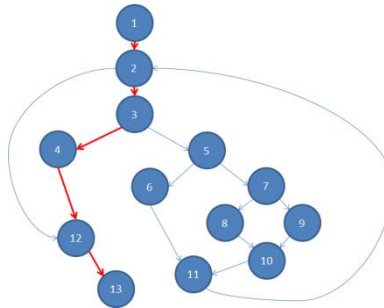
P5: 1, 2, 3, 7, 9, 10, 11, 2, 12, 13

Independent Paths

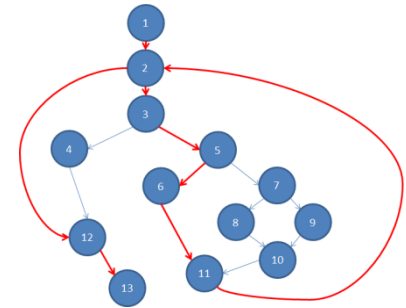
- Easiest way to find this set is to work from **top-to-bottom, left-to-right** to create paths



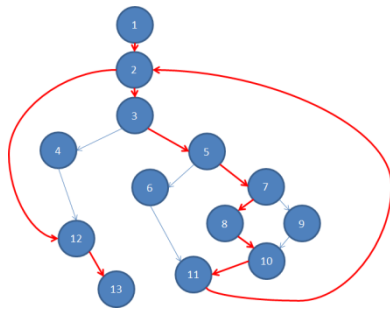
P1: 1, 2, 12, 13



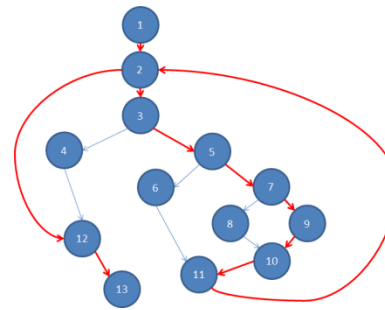
P2: 1, 2, 3, 4, 12, 13



P3: 1, 2, 3, 5, 6, 11, 2, 12, 13



P4: 1, 2, 3, 7, 8, 10, 11, 2, 12, 13



P5: 1, 2, 3, 7, 9, 10, 11, 2, 12, 13

Independent Paths

$x < 0$ $x \geq 0$

1 if ($x < 0$)

2 $y -= 1$; yes no

3 else

4 $y += 2$; no yes

5 if ($x < 0$)

6 $y -= 3$; yes no

7 else

8 $y += 5$; no yes

Not “semantically independent” paths.

1, 2, 5, 7, 8 is an independent path,

though impossible

(2 only hit if $x < 0$, 8 only hit if $x \geq 0$)

Path Coverage Testing

- **System**: Make one test case for each independent path analyzing which **inputs** are needed to exercise the path
- **Completion criterion**: A test for each path
- Test creation is easy once paths have been identified

Path	list[]	n	key
P1	(empty)	0	0
P2	1 2 3 4	4	0
P3	1 2 3 4	4	1
P4	1 2 3 4	4	4
P5	1 2 3 4	4	2

Path Coverage Testing

- Advantages

- Covers all **basic blocks**:
does all of basic block testing
- Covers all **conditions**:
does all of decision/condition testing
- Does all of both, but with **fewer tests!**
- Automatable

- Disadvantages

- Does not take **data** complexity into account at all
(also a disadvantage of basic block coverage and condition coverage)

Path Coverage Testing

- **Disadvantage Example**

- These fragments have the same outcome and should be tested the same way, but the one on the left gets five tests whereas the one on the right gets only one

// control-centric solution

```
switch (n) {  
    case 1:  
        s = "One"; break;  
    case 2:  
        s = "Two"; break;  
    case 3:  
        s = "Three"; break;  
    case 4:  
        s = "Four"; break;  
    case 5:  
        s = "Five"; break;  
}
```

// data-centric solution

```
String[] numbers =  
    { "One", "Two",  
      "Three", "Four",  
      "Five" };  
  
s = numbers[n-1];
```

White Box Data Coverage

- **Data coverage methods** explicitly try to cover the **data** aspects of the program code, rather than the **control** aspects
- Three kinds: data **value** coverage, data **flow** coverage, data **interface** coverage

White Box Data Coverage

- **System:** Identify **critical variables**, analyze code to find the different **values** or sets of values each can take on, partition and design tests to cover
- **Completion criterion:**
Test for each value partition

Data Path Coverage

- **System:** Identify **output variables**, analyze code to find data flow paths that affect their value (technically called **slices**)
- **Completion criterion:** Test for each data flow path to output
- Much like control path testing, but additionally identifies **data-centric paths**

Data Path Coverage

- For example, if we look at a program slice for the variable **sum**, we can eliminate portions of the source that do not affect the value of **sum**

```
int i;  
int sum = 0;  
int product = 1;  
for (i = 0; i < n; i++) {  
    sum = sum + i;  
    product = product * i;  
}
```

```
int i;  
int sum = 0;  
// line not in slice  
for (i = 0; i < n; i++) {  
    sum = sum + i;  
    // line not in slice  
}
```


Data Interface Coverage

- **System:** Identify unit's interface **input variables**, analyze code to find classes of values that cause different code behaviours, partition and test
- **Completion criterion:**
Test for each input value partition
- Really just like black box **input partitioning** again, but with added advantages of ability to analyze code when creating partitions

Summary

- White Box Testing
 - Code coverage methods:
 - Decision analysis methods: path coverage
 - Data coverage methods:
 - Data value coverage, flow coverage, interface coverage
- Next Time
 - Mutation testing