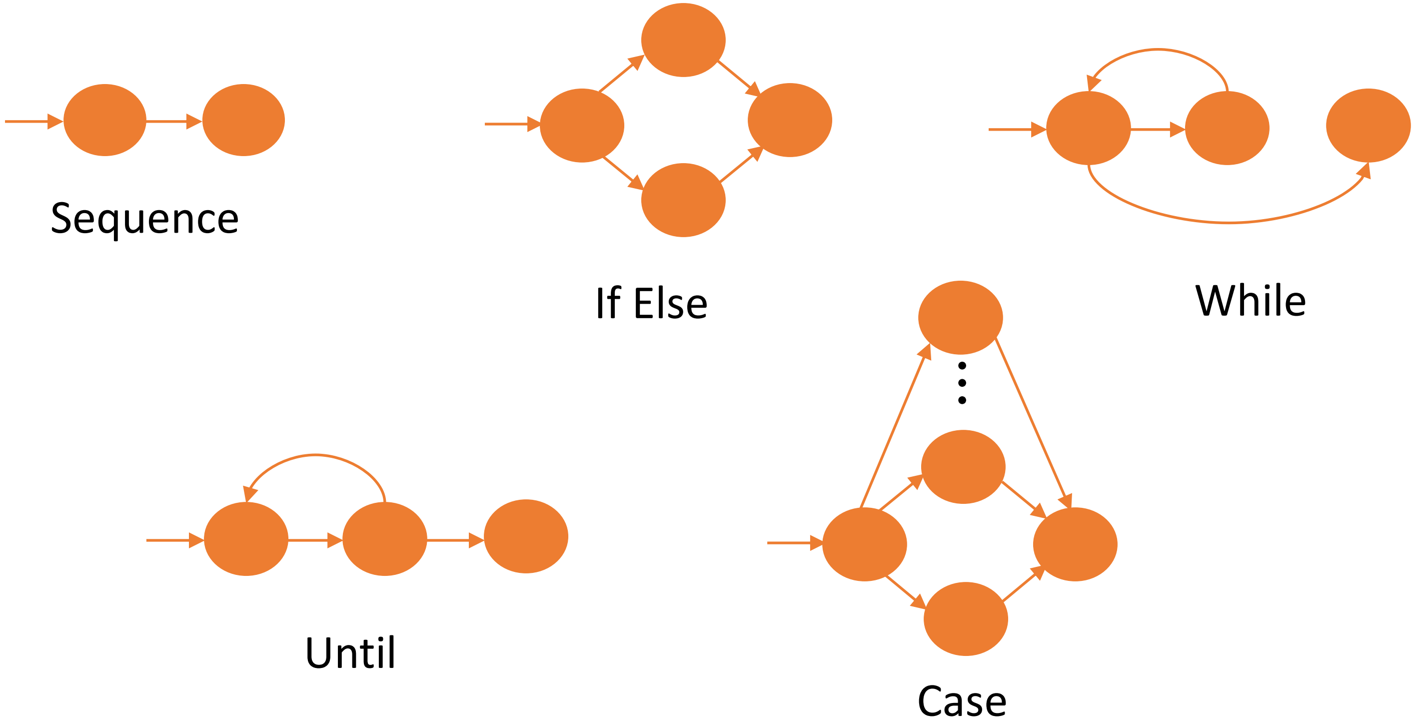
Basis Path Testing

Basis Path Testing

- Basic path testing (a white-box testing technique):
  - First proposed by Tom McCabe.
  - Can be used to derive a logical complexity measure for a procedure design.
  - Used as a guide for defining a basis set of execution path.
  - Guarantee to execute every statement in the program at least one time.

Basis Path Testing (cont’d)

- The basic structured-constructs in a flow graph :



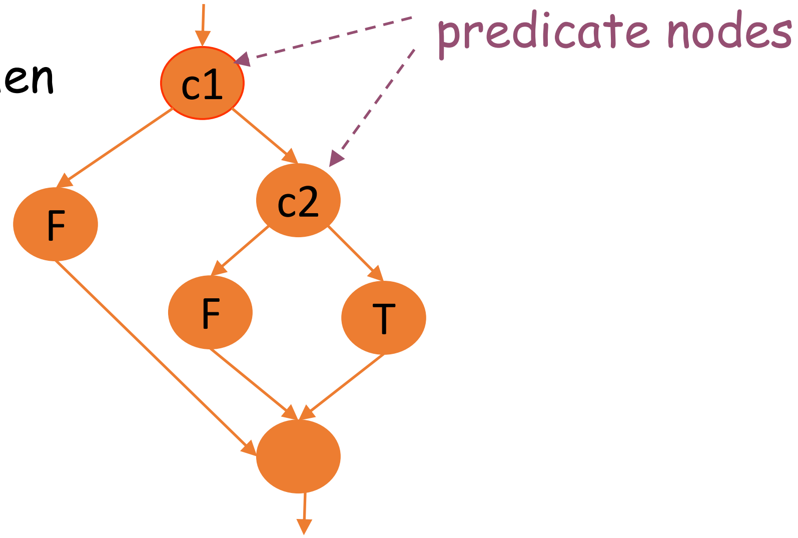
Basis Path Testing (cont’d)

- Flow graph notation (control flow graph)
  - Node represents one or more procedural statements.
    - A sequence of process boxes and a decision diamond can map into a single node
    - A predicate node is a node with two or more edges emanating from it
  - Edge (or link) represents flow of control
  - Region: areas bounded by edges and nodes
    - When counting regions, include the area outside the graph as a region

Basis Path Testing (cont'd)

- Compound condition
  - Occurs when one or more Boolean operators (OR, AND, NAND, NOR) is present in a conditional statement
  - A separate node is created for each of the conditions *C1* and *C2* in the statement *IF C1 AND C2*

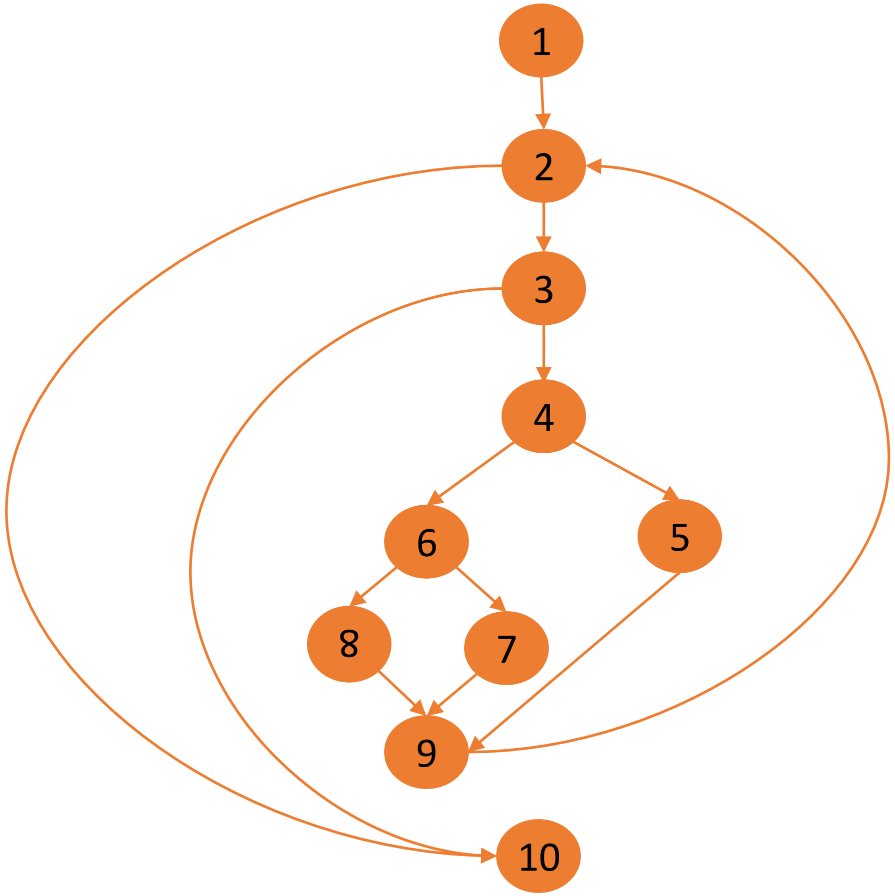
```
if (c1 AND c2) then
    print T;
else
    print F;
end if;
```



binarySearch() Example

```
     public int binarySearch(int sortedArray[ ], int searchValue)
     {
         int bottom = 0;
         int top = sortedArray.length - 1;
         int middle, locationOfsearchValue;                          2
 1       boolean found = flase;
         locationOfsearchValue = -1;        /* the location of searchValue in the sortedArray    */
                                            /* location = -1 means that searchValue is not found */
         while ( bottom <= top && !found)
         {                                                           3
 4           middle = (top + bottom)/2;
             if (searchValue == sortedArray[ middle ])
             {
 5               found = true;
                 locationOfsearchValue = middle;
             }
 6           else if (searchValue < sortedArray[ middle ])
                     top = middle - 1;                               7
 8           else
                     bottom = middle + 1;
         } // end while                                              9
10       return locationOfsearchValue;
     }
```

The Control Flow Graph (CFG) of Function binarySearch()



Cyclomatic Complexity (cont'd)

- Cyclomatic complexity is a software metric
  - provides a quantitative measure of the global complexity of a program.
  - When this metric is used in the context of the basis path testing
    - the value of cyclomatic complexity defines the number of independent paths in the basis set of a program
    - the value of cyclomatic complexity defines an upper bound of number of tests (i.e., paths) that must be designed and exercised to guarantee coverage of all program statements

Cyclomatic Complexity (cont'd)

- Independent path
  - An independent path is any path of the program that introduce at least one new set of procedural statements or a new condition
  - In a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined
    - Examples: consider the CFG of binarySearch()
      - Path 1: 1-2-10
      - Path 2: 1-2-3-4-6-8-9-2-10
      - Path 3: 1-2-3-4-6-8-9-2-3-10 independent paths
      - Path 4: 1-2-3-4-6-8-9-2-3-4-6-8-9-2-10 (not an independent path)

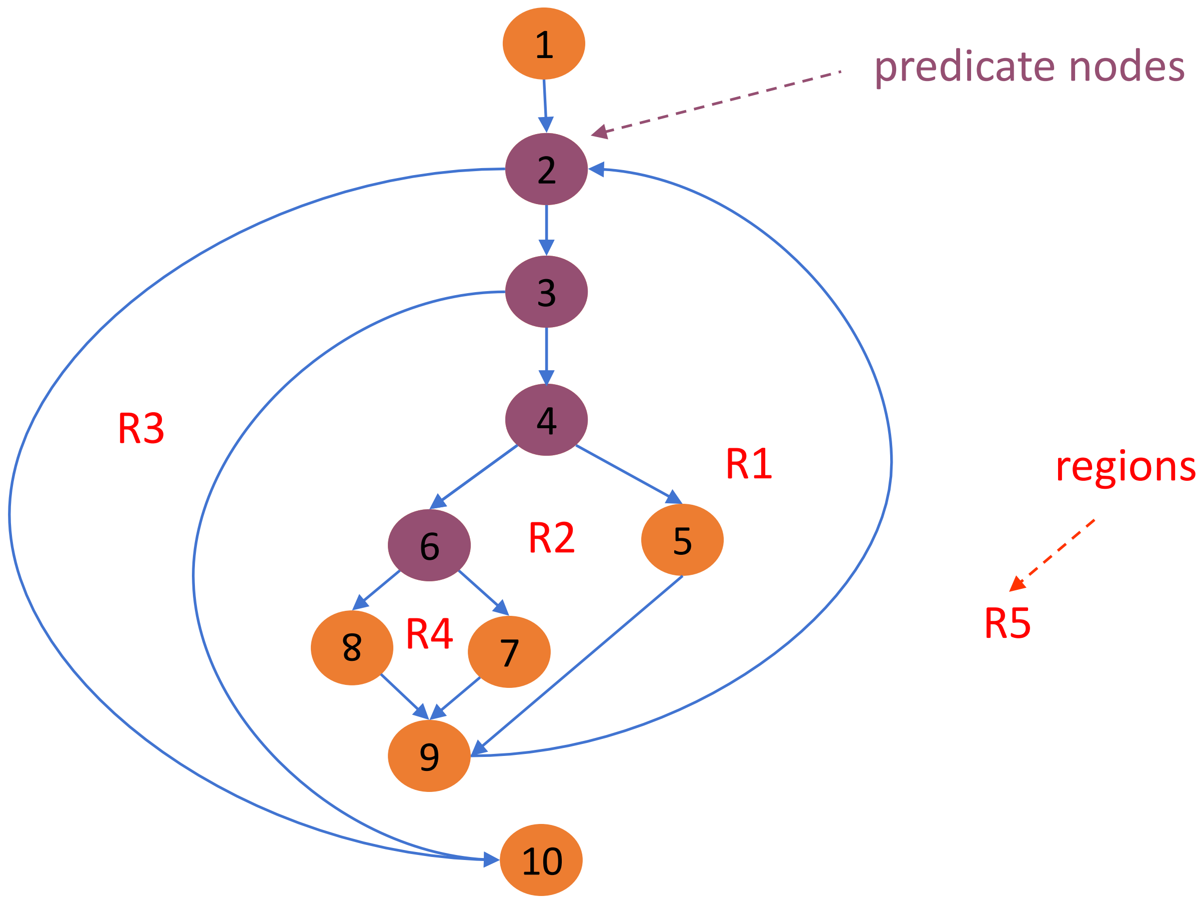
Cyclomatic Complexity (cont'd)

- Three ways to compute cyclomatic complexity:
  - The number of regions of the flow graph correspond to the cyclomatic complexity.
  - Cyclomatic complexity, V(G), for a flow graph G is defined as $V(G) = E - N + 2$

    where E is the number of flow graph edges and N is the number of flow graph nodes.
  - Cyclomatic complexity, $V(G) = P + 1$

    where P is the number of predicate nodes contained in the flow graph G.

Cyclomatic Complexity of Function binarySearch()



Deriving Basis Test Cases

- The following steps can be applied to derive the basis set:
  1. Using the design or code as a foundation, draw the corresponding flow graph.
  2. Determine the cyclomatic complexity of the flow graph.
     - V(G) = 5 regions
     - V(G) = 13 edges – 10 nodes + 2 = 5
     - V(G) = 4 predicate nodes + 1 = 5

Deriving Basis Test Cases (cont'd)

3. Determine a basis set of linearly independent paths.
   - Path 1: 1-2-10
   - Path 2: 1-2-3-10
   - Path 3: 1-2-3-4-5-9-2-3-10- …
   - Path 4: 1-2-3-4-6-7-9-2-…
   - Path 5: 1-2-3-4-6-8-9-2-…
4. Prepare test cases that force the execution of each path in the basis set
   - Path 1 test case:
     - Inputs: sortedArray = { }, searchValue = 2
     - Expected results: locationOfSearchValue = -1

Deriving Basis Test Cases (cont'd)

- Path 2 test case: cannot be tested stand-alone!
  - Inputs: sortedArray = {2, 4, 6}, searchValue = 8
  - Expected results: locationOfSearchValue = -1
- Path 3 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 6
  - Expected results: locationOfSearchValue = 2
- Path 4 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 4
  - Expected results: locationOfSearchValue = 1
- Path 5 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 10
  - Expected results: locationOfSearchValue = 4

Deriving Basis Test Cases (cont'd)

- Each test cases is executed and compared to its expected results.
- Once all test cases have been exercised, we can be sure that all statements are executed at least once
- Note: some independent paths cannot be tested stand-alone because the input data required to traverse the paths cannot be achieved
  - In binarySearch(), the initial value of variable *found* is FALSE, hence path 2 can only be tested as part of path 3, 4, and 5 tests