(CS510)
Software Engineering
Project 1 (70 Points)
Sample Solutions

Instructors: Lin Tan
TA: Yi Sun

September 30, 2019

Question 1

This problem is that "when either dividend or divisor is negative, the naive definition of modulo operation breaks down and programming languages differ in how these values are defined" (Wikipedia). So, the value of -5%2 is -1 or 1 depending on the compiler. For example, it is -1 for C (ISO 1999), C++ (ISO 2011), C#, Java, JavaScript, PHP, bash. For Python, Ruby (using "`%`"), and Perl (using "`print`"), it is 1. More information can be found at `http://en.wikipedia.org/wiki/Modulo_operation`.

In addition, if you use *printf* in Perl, e.g., `printf "%d" -5%2`, the output is -1.

To address this issue, it is an open question. Possible solutions include:

- **Standard** Set a standard for different programming languages on how % should be handled, so there is no inconsistency. Or, design a common interface for modulo operation and let every programming language provide the corresponding library.
- **Programmer** Programmers take a defensive approach, and always take the absolute value before performing a % operation, e.g., abs(x) % 2.

  Alternatively, we can test the variable to determine if it is positive or negative and perform the modulus accordingly, e.g.,

```
if (a<0) then
    b = c +(a%c)
else
    b = a%c
end
```

  where in our case, a is -3 and c is 2.

  Another option is to continue to add c to a until the result is greater than 0, then perform the modulus, e.g.,

```
while (c<0)
    a = a + c
end
b = a % c
```

- **Compiler** Extend compilers to warn developers of % operation on negative values. Or compilers can provide an option for toggling which kind of output should be generated.
- **Others** Better documentation about this problem, educating programmers, etc.

Marking Scheme [Divide the marks by 2 to make it out of 5]

Source code using 4 different programming languages accounts for 2 points. Reporting your findings accounts for 2 points. Proposing two strategies for the problem takes the rest 3 points.

- **Source code** Should have source code in 4 different programming languages, each taking 0.5 points.
- **Findings** Should report *clearly* what you have found by running the above 4 programs, and explain why it happens like this.
- **Strategies** Each strategy takes 3 points, 2 for correctness, 1 for clearness. And your two strategies should come from different points of view; otherwise, I will lower your score by 2 points.

Common Mistakes

This problem is because modulo operation with negative numbers is not well defined. So different programming languages have different implementations.

- It has nothing to do with compiled/scripting languages. It is not safe to derive from 4 programming languages that compiled languages will follow the sign of dividend and scripting language will take the sign of divisor. Bash is a scripting language, but its result has the same sign as dividend; C is a compiled language, but it can output result with the sign of divisor (ISO 1990).
- It also has nothing to do with precedence. Unary operations normally has higher precedence than binary operations. (For C-style languages, please refer to "Programming Languages" of `http://en.wikipedia.org/wiki/Order_of_operations`) So, `-5%2` is interpreted as `(-5)%2` in C/C++/Java. The reason of C/C++/Java outputting -1 is not C/C++/Java interprets `-5%2` as `-(5%2)`.

Question 2

a) Memory leak. `struct node` uses `str`, which is a pointer to a C string, for the memory storing the user name. That memory is allocated in `fgets_enhanced()` but when deallocating a node in `delete_node()`, it is not deallocated and no pointer is pointing to it thereafter, which causes a memory leak. Figure **??** is the Valgrind output.

Adding `free(temp->str);` (at Line 84) before `free(temp);` fixes the problem.

b) Dangling pointer. In `delete_all()`, all the `struct node`s are deallocated but the global pointer `p` is not set to `NULL`. So, when `delete_all()` is called the second time for operation 'x', the program is trying to deallocate the memory which is already freed.

Setting `p` to `NULL` at the end of `delete_all()` (Line 107) solves the problem.

c) An example of buffer overflow. A trace like (Insert, Duplicate, Exit) causes Valgrind to show "Invalid write of size 1", which means buffer overflow. The reason is that in `duplicate()`, the memory allocated for `name` is not enough. C string is null terminated, nonetheless `strlen()` only returns the real number of characters in the string. But `strcpy()` will copy everything upto and including the null terminator. So this causes a off-by-one write.

At Line 190, changing `char* name = malloc(len);` to `char * name = malloc(len + 1);` solves the problem.

```
==10572== Command: ./sll_buggy
==10572==
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Tom
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>111
enter the name:>Mary
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:d
enter the tel :>111
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==10572==
==10572== HEAP SUMMARY:
==10572==     in use at exit: 9 bytes in 1 blocks
==10572==   total heap usage: 5 allocs, 4 frees, 67 bytes allocated
==10572==
==10572== 9 bytes in 1 blocks are definitely lost in loss record 1 of 1
==10572==    at 0x4C2C6AE: realloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10572==    by 0x40096F: fgets_enhanced (in /home/lei/testing/assignments/a2/valgrind/sll_buggy)
==10572==    by 0x400FC5: main (in /home/lei/testing/assignments/a2/valgrind/sll_buggy)
==10572==
==10572== LEAK SUMMARY:
==10572==    definitely lost: 9 bytes in 1 blocks
==10572==    indirectly lost: 0 bytes in 0 blocks
==10572==      possibly lost: 0 bytes in 0 blocks
==10572==    still reachable: 0 bytes in 0 blocks
==10572==         suppressed: 0 bytes in 0 blocks
==10572==
==10572== For counts of detected and suppressed errors, rerun with: -v
==10572== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Figure 1: Valgrind output for TC1

Marking Scheme

a) Report the problem (1 mark for valgrind output). Explain the problem (2 marks for correctly identifying the problem and explaining it. Ex: memory leak issue - did not free allocated memory for variable $temp->str$). Fixed the bug (sll_fixed.c) (1 marks for the fixing the problem free($temp->str$)). Explain how you fix the bug (1 marks for explaining the fix. Ex: Calling free on the variable $temp->str$ to free the memory that has been allocated for char *str) - (Total of 5 points)

b) Report and explain the problem (2 marks for correctly identifying the problem and explaining it. Ex: dangling pointer issue - the variable p has to be set to NULL after all the nodes are removed). Fixed the bug (sll_fixed.c) (1 marks for fixing the problem p = NULL). Explain how you fix the bug (1 marks for explaining the fix. Ex: You have to set p = NULL so that it is not accessed by mistake when adding/removing things from the list) - (Total of 5 points)

c) Report the problem (2 marks for the test case that revealed the bug). Explain the problem (2 marks for explaining the problem. Ex: why the buffer overflow problem happens and where the bug is in the code). Fix the bug (1 mark for fixing the problem). - (Total of 5 points)

Question 3

A) Answers to part (a) Control Flow Graphs:

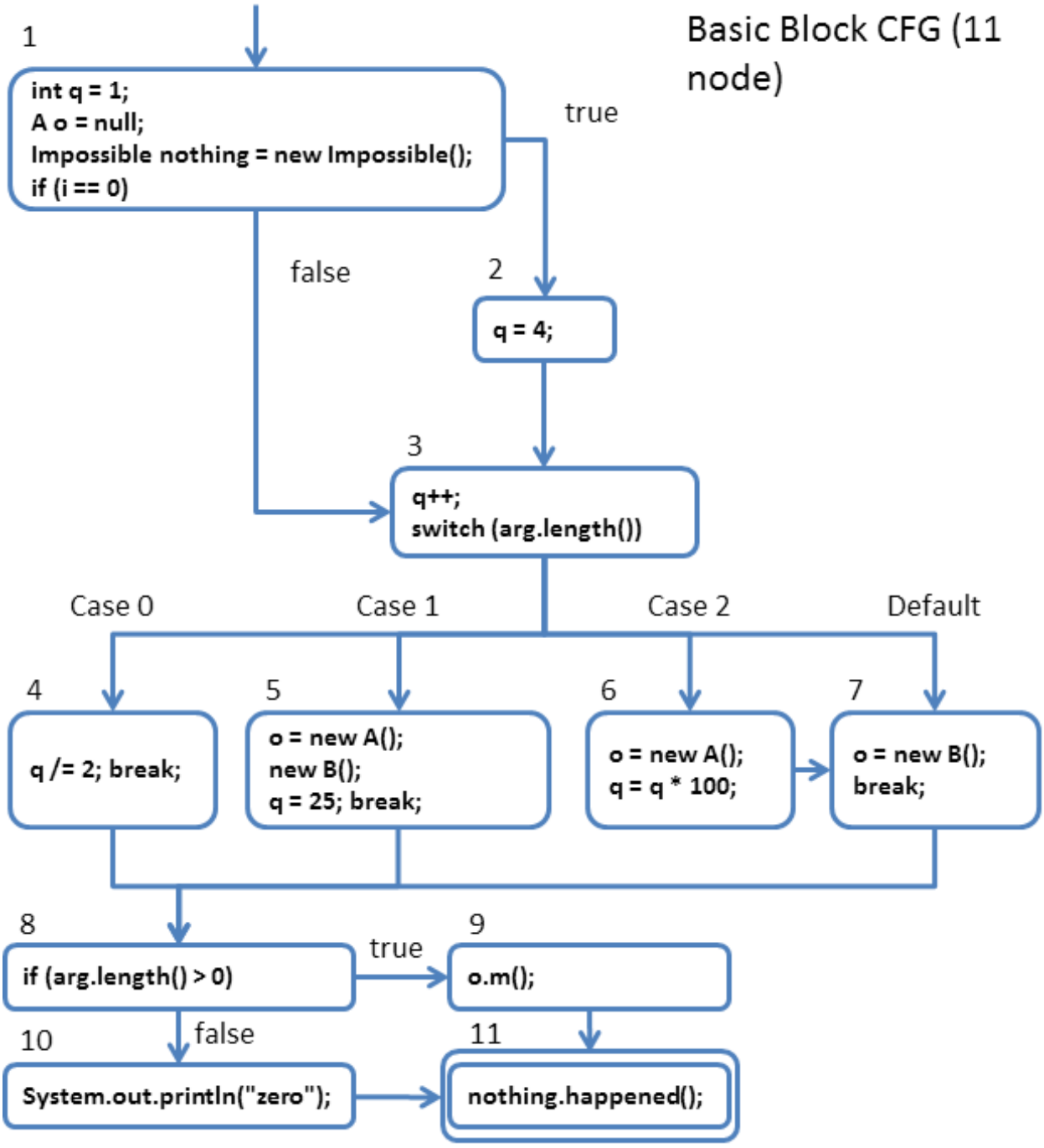

Figure 2: Minimal (11) Node Basic Block CFG

B) Answers to part (b) Test Requirements:

The Test Requirements for the Basic Block CFG is listed below.

$TR_{NodeCoverage}$ = {[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]}

$TR_{EdgeCoverage}$ = {[1,2], [1,3], [2,3], [3,4], [3,5], [3,6], [3,7], [4,8], [5,8], [6,7], [7,8], [8,9], [8,10], [9,11], [10,11]}

$TR_{EdgePairCoverage}$ = {[1,2,3], [1,3,4], [1,3,5], [1,3,6], [1,3,7], [2,3,4], [2,3,5], [2,3,6], [2,3,7], [3,4,8], [3,5,8], [3,6,7], [3,7,8], [4,8,9], [4,8,10], [5,8,9], [5,8,10], [6,7,8], [7,8,9], [7,8,10], [8,9,11], [8,10,11]}

$TR_{FeasibleEdgePairCoverage}$ = {[1,2,3], [1,3,4], [1,3,5], [1,3,6], [1,3,7], [2,3,4], [2,3,5], [2,3,6], [2,3,7], [3,4,8], [3,5,8], [3,6,7], [3,7,8], [4,8,10], [5,8,9], [6,7,8], [7,8,9], [8,9,11], [8,10,11]}

Infeasible Edge Pairs = {[4,8,9], [5,8,10], [7,8,10]}

These are due to the contradictory logic in the SWITCH statement and the second IF statement. Whenever arg.length() = 0 then nodes 4 and 10 happens together; similarly whenever arg.length() > 0 then either nodes 5 or 7, and 9 happen together.

$TR_{PrimePathCoverage}$ = {[1,2,3,4,8,9,11], [1,2,3,4,8,10,11], [1,2,3,5,8,9,11], [1,2,3,5,8,10,11], [1,2,3,6,7,8,9,11], [1,2,3,6,7,8,10,11], [1,2,3,7,8,9,11], [1,2,3,7,8,10,11], [1,3,4,8,9,11], [1,3,4,8,10,11], [1,3,5,8,9,11], [1,3,5,8,10,11], [1,3,6,7,8,9,11], [1,3,6,7,8,10,11], [1,3,7,8,9,11], [1,3,7,8,10,11]}

$TR_{FeasiblePrimePathCoverage}$ = {[1,2,3,4,8,10,11], [1,2,3,5,8,9,11], [1,2,3,6,7,8,9,11], [1,2,3,7,8,9,11], [1,3,4,8,10,11], [1,3,5,8,9,11], [1,3,6,7,8,9,11], [1,3,7,8,9,11]}

Infeasible Prime Paths = {[1,2,3,4,8,9,11], [1,2,3,5,8,10,11], [1,2,3,6,7,8,10,11], [1,2,3,7,8,10,11], [1,3,4,8,9,11], [1,3,5,8,10,11], [1,3,6,7,8,10,11], [1,3,7,8,10,11]}

Any Prime Path containing an infeasible Edge Pair will also be infeasible. There are no other factors in this code which contribute to any other infeasible Prime Paths.

C) Answers to part (c):

1) You can achieve node coverage but not edge coverage by not exercising edge [1,3], and/or [3,7]. This can be achieved by having a minimum of 3 test cases all which pass 0 for the value of i for method m, with strings of length of empty (“”), 1 character long (“a”), and 2 characters long (“ab”) passed into the parameter arg, respectively. Please see the code below for examples.

2)

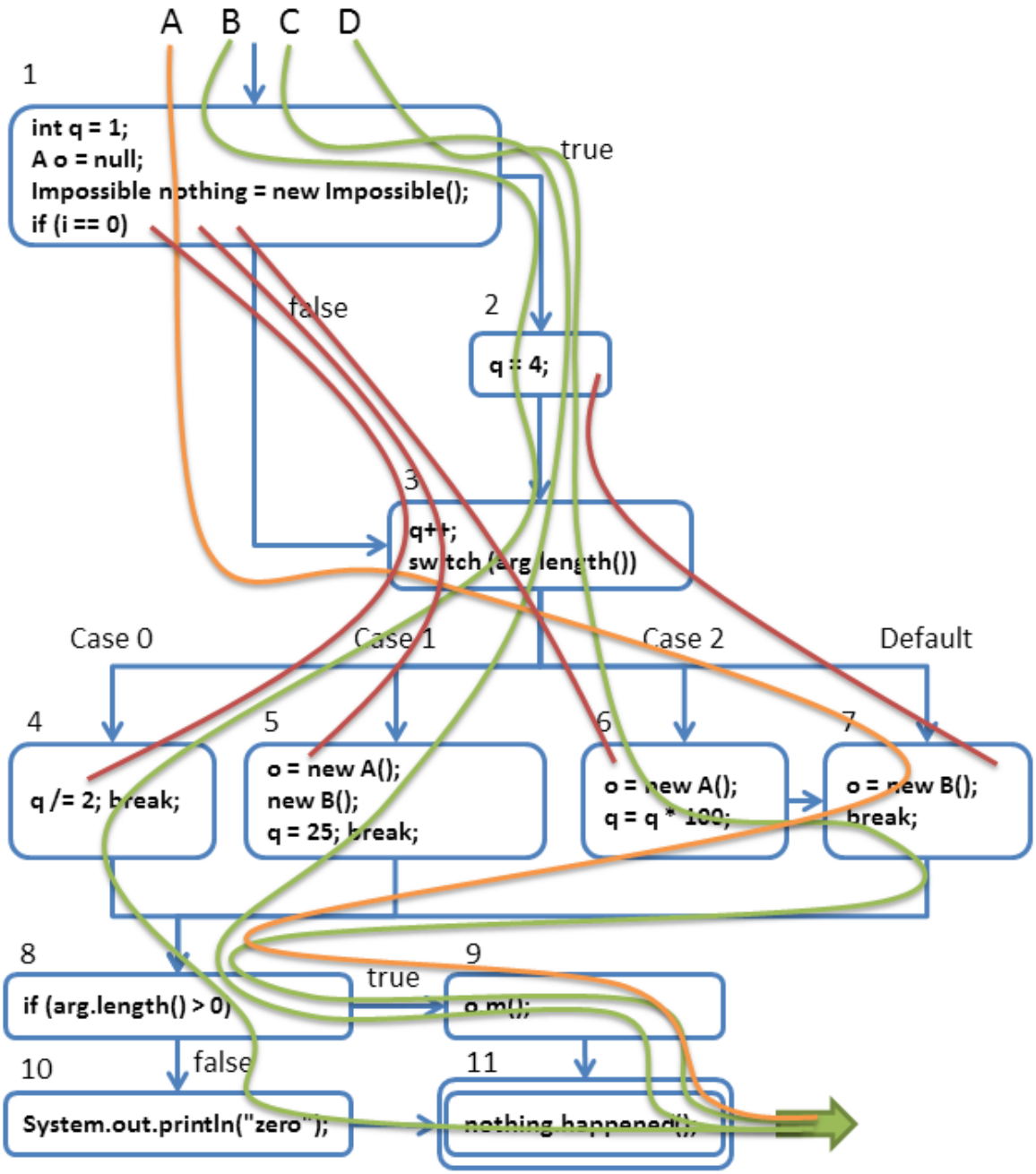

Figure 3: Identifying Edge but not Edge Pair Tests

As an example, the 4 test paths A, B, C, and D will test all Edges, but will not invoke 4 Edge Pairs: [1,3,4], [1,3,5], [1,3,6], and [2,3,7] (red lines). The inputs for the 4 test paths are given in the test code solution section below.

There are 4 sets of correct Edge Pairs not invoked by Edge Coverage Tests, which depend on which one of the 4 paths traverses through Node 2. In total, there are 4 Edge Pairs not covered by any minimal set of test paths which satisfy Edge Coverage.

A minimum of 4 test cases are required to satisfy Edge Coverage, which are the three from Node Coverage and 1 additional test case of passing a string of length greater than 2 characters ("abc") for arg, and value

of 1 for i for method m.

3) We explained earlier why there are infeasible Edge Pair and Prime Path TRs. Considering only feasible Edge Pair and Prime Path TRs in their respective Coverages, the minimum test cases to achieve Edge Pair coverage will also achieve Prime Path coverage. One can see that every three consecutive nodes in each Prime Path is also an Edge Pair. Every Edge Pair also appears in at least one Prime Path. Hence, to find a test set which covers EPC but not PPC is not possible in this question.

4) The test cases which achieve Prime Path Coverage are the ones in 1) and 2) as well as 4 additional test cases which achieve the 4 Red paths shown in the CFG earlier. Please see the code below for examples.

D) Test Code Solution:

The function M.m() provides three possible outputs to System.out in the form of a string. As such, it is expected that these outputs are captured and compared to expected outputs from test case design. The code template provides three import declarations for `PrintStream`, `OutputStream`, and `ByteArrayOutputStream`, which is a hint for how to handle the `System.out.println()` outputs created by function `M.m()`.

In this example Test Code solution, we create a helper test function called `myTest()` which helps us cut down the number of repeating lines of code in the test program. In addition, the use of @Before and @After test functions are used to properly handle pre-test initialization and post-test clean up tasks. We also capture and release the System output stream for System.out. We utilize `System.getProperty("line.separator")` to make the test code platform independent for handling string outputs.

```
import org.junit.*;
import static org.junit.Assert.*;

import java.io.PrintStream;
import java.io.OutputStream;
import java.io.ByteArrayOutputStream;

/* Test class for class M, author: Song Wang. */
public class TestM {
        private M mInstance;
        private String myA;
        private String myB;
        private String myZero;
        private String separator;

        private PrintStream originalOut; // used to remember the original System.out print stream
        private OutputStream os;
        private PrintStream ps;

        public TestM() {
                // Constructor Function sets up our output streams and our expected outputs
                os = new ByteArrayOutputStream();
                ps = new PrintStream(os);
                // Using Separator is much safer than using "\n" (correct only in Linux)
                // or "\r\n" (correct only in Windows)
                separator = System.getProperty("line.separator");
                myA = "a" + separator;
                myB = "b" + separator;
                myZero = "zero" + separator;
        }

        // Use redirection to capture output
        private void getOut() {
                originalOut = System.out;
                System.setOut(ps);
        }

        // Reset System.out to original output stream (console)
        private void releaseOut() {
                System.setOut(originalOut);
        }

        // We have to create a new instance of M for each test of the method M.m()
        @Before
        public void setup() {
```

```
                mInstance = new M();     // Create new M object
                getOut();           // Capture and redirect System.out
        }

        @After
        public void tearDown() {
                releaseOut();   // Release System.out and revert to original settings
                mInstance = null;
        }

        // General helper function which can execute any black box test for M.m()
        public void myTest(String argval, int i, String expected_response){
                // Run the Function and provide the inputs of the black box test
                mInstance.m(argval, i);
                // Check the output of the black box test against expected output
                assertEquals(os.toString(), expected_response);
        }

/*NODE BUT NOT EDGE COVERAGE*/

        /*
        * Nodes: 1,2,3,4,8,10,11
        * Edges: [1,2] [2,3] [3,4] [4,8] [8,10] [10,11]
        * Edge Pairs: [1,2,3] [2,3,4] [3,4,8] [4,8,10] [8,10,11]
        * Prime Path: [1,2,3,4,8,10,11]
        */
        @Test
        public void testM_NC_1() {
                myTest("", 0, myZero);          // Green Line B
        }

        /*
        * Additional Nodes: 5,9
        * Edges: [3,5] [5,8] [8,9] [9,11]
        * Edge Pairs: [2,3,5] [3,5,8] [5,8,9] [8,9,11]
        * Prime Path: [1,2,3,5,8,9,11]
        */
        @Test
        public void testM_NC_2() {
                myTest("1", 0, myA);            // Green Line C
        }

        /*
        * Additional Nodes: 6,7
        * Edges: [3,6] [6,7] [7,8]
        * Edge Pairs: [2,3,6] [3,6,7] [6,7,8] [7,8,9]
        * Prime Path: [1,2,3,6,7,8,9,11]
        */
        @Test
        public void testM_NC_3() {
                myTest("22", 0, myB);           // Green Line D
        }

/*FOR EDGE BUT NOT EDGE PAIR COVERAGE, THE CASES ABOVE PLUS THE ONES SHOWN IN THIS SECTION*/

        /*
        * Edges: [1,3] [3,7]
        * Edge Pairs: [1,3,7] [3,7,8]
        * Prime Path: [1,3,7,8,9,11]
        */
        @Test
        public void testM_EC_4() {
                myTest("grumpy cat", 1, myB);   // Orange Line A
        }

/*FOR EDGE PAIR COVERAGE AND PRIME PATH COVERAGE, THE CASES ABOVE PLUS ONES SHOWN BELOW */

        /*
        * Edge Pair: [1,3,4]
        * Prime Path: [1,3,4,8,10,11]
        */
        @Test
        public void testM_EPC_PPC_5() {
                myTest("", 1, myZero);
        }

        /*
        * Edge Pair: [1,3,5]
        * Prime Path: [1,3,5,8,9,11]
        */
```

```
        @Test
        public void testM_EPC_PPC_6() {
                myTest("1", 1, myA);
        }

        /*
        * Edge Pair: [1,3,6]
        * Prime Path: [1,3,6,7,8,9,11]
        */
        @Test
        public void testM_EPC_PPC_7() {
                myTest("22", 1, myB);
        }

        /*
        * Edge Pair: [2,3,7]
        * Prime Path: [1,2,3,7,8,9,11]
        */
        @Test
        public void testM_EPC_PPC_8() {
                myTest("nyan cat", 0, myB);
        }
}
```

General Observations and Comments:

The only function for which a CFG should be drawn is M.m() as stated in the assignment instructions, the sub-CFGs of classes A and B, as well as the super-CFG of M.main() are not supposed to be drawn or included in the CFG. Similarly, analysis for Test Cases and Test Requirements should not include them. An explicit reason for why M.main() is not included in any analysis is because M.m() is a public function, which implies that it can be called by other functions which are not shown in the code that was provided.

Using Numbers as the IDs for nodes in the CFG is preferred over Letters. If a program has more than 26 nodes then letters will get confusing. Also, the first node should use the number 1 instead of 0, which makes it easier to know the total number of nodes in the graph.

Using "\n" (works only in Linux/Unix) or "\r\n" (works only in Windows) instead of System.getProperty("line.separator") leads to cross-platform errors as the Assert comparison will fail depending on which operating system the test code is executed on.

Passing a value of null into arg for function M.m() will cause an exception at the switch statement, and subsequent code in the program will not be executed. Exception handling is not considered in this assignment, in fact the lack of exception handling is the only bug in this code! However, passing a value of an empty string ("") into arg will execute the function, in this case arg.length() == 0.

The most frequent types of mistakes encountered are:

- CFG drawing:
  - constructing a CFG consisting only of node IDs without separately providing indication of which code lines correspond to which node
  - having an edge between Case 2 of the SWITCH statement node and the second IF statement
  - expanding o.m() to include the CFGs of the A and B classes
  - incorrectly constructing CFG to include code from M.main()
  - creating an END node after the node containing nothing.happened(), and inadvertently analyzing the END node as an extra edge in test requirements. The use of an extra END node is unnecessary in this question because the node containing nothing.happened() is the end node.
- Listing TRs:
  - forgetting to list the Edge between the code lines q=4; and q++; even though CFG has it

  - listing an Edge between Case 2 of the SWITCH statement node and the second IF statement, even though CFG does not have it
  - not identifying (all) correct Edges, Edge Pairs and/or Prime Paths
  - not identifying (all) infeasible Edge Pairs and/or Prime Paths
  - not justifying why the infeasible Edge Pairs and Prime Paths are infeasible

- Thinking Questions and Coding:
  - treating an entire test criterion as infeasible when some test requirements are infeasible. Infeasible test requirement is not the same thing as infeasible test criterion. A test criteron includes all feasible TRs, which exists for Edge Pair coverage and Prime Path coverage in this question.
  - forgetting to identify which Edge Pairs are not invoked in Edge Coverage Test
  - forgetting to use string based Assert comparison statements to complete the black box tests
  - forgetting to provide documentation or comment lines on which TRs are being tested
  - not writing code for all feasible Edge Pairs and Prime Paths
  - submitting code files with no content inside :((please double check next time)
  - not submitting any code at all :(

Test Requirement Counts and Marking Scheme:

Table 1: Marking Scheme for Q3

Item	Points	Notes
Correct Basic Block CFG Drawn	5	-0.5 points for not drawing an arrow pointing to the first node; -0.5 points for not double lining or bolding the border of the last node; -1 points for CFG not condensed to minimal number of basic blocks; -0.5 points for having nodes with the word "START" or "END", if such nodes caused extra test requirements; -2 points for not associating code lines to node numbers, or have otherwise caused ambiguity in the graph; -0.5 points for each unlabeled branch condition, incorrect edge or node up to full amount
Correct Node Coverage	1	points given as a % of number of correct nodes (see table below)
Correct Edge Coverage	1	points given as a % of number of correct edges (see table below)
Correct Edge Pair Coverage	2	points given as a % of number of correct feasible and infeasible Edge Pairs (see table below)
Correct Infeasible Edge Pair TRs listed and Reason identified	1	points given as a % of number of correct infeasible Edge Pairs identified (see table below)
Correct Prime Path Coverage	2	points given as a % of number of correct feasible and infeasible Prime Paths (see table below)
Correct Infeasible Prime Path TRs listed and Reason identified	1	points given as a % of number of correct infeasible Prime Paths identified (see table below)
Test Code Executes	1	-0.5 points if proper line separator is not used; -0.5 points if other minor coding error(s) are present
Test Code is Correct, valid cases for (1),(2),(3),(4) identified and justified, all feasible TRs are tested	5	-1.5 points if string based assert comparison statement(s) are missing -0.5 points for each critical test code mistake up to full amount
Test Code is documented	1	-0.5 points if basic sections for NC, EC, EPC, and PPC are not identified; -0.5 points if detailed TRs are not listed
Total	20	

Table 2: Correct Number of Nodes, Edges, Edge Pairs, and Prime Paths for CFGs

Nodes	Edges	Total Edge Pairs	Infeasible Edge Pairs	Total Prime Paths	Infeasible Prime Paths	Notes
11	15	22	3	16	8	Minimal Node Basic Block CFG

Question 4

`CFG.addNode()`

Test Cases and subsequent Test Paths:

- `addNode()` : $[A, B, C, D, E, F]$
- `addNode_duplicate()` : $[A, B, C, F]$

Achieve *Node* and *Edge* coverage for `CFG.addNode()` method.

`CFG.addEdge()`

Test Cases and subsequent Test Paths:

- `addEdge()` : $[A, B, C, D, E, F, G, H]$
- `addEdge_oneNewNode()` : $[A, B, C, D, E, G, H]$

Achieve *Node* coverage for addEdge() method.

`CFG.deleteNode()`

Test Cases and subsequent Test Paths:

- `deleteNode()` : $[A, B, C, D, E, F, G, H, I, J, G, K]$ and $[G, H, I, G]$ cycle.
- `deleteNode_missing()` : $[A, B, C, K]$

Achieve *Node* and *Edge* coverage for `CFG.deleteNode()` method.

`CFG.deleteEdge()`

Test Cases and subsequent Test Paths:

- `deleteEdge()` : $[A, B, C, D, E, F, G]$
- `deleteEdge_missing()` : $[A, B, C, G]$
- `deleteEdge_missingSrcNode()` : $[A, B, C, G]$

Achieve *Node* coverage for `CFG.addNode()` method.

`CFG.isReachable()`

Test Cases and subsequent Test Paths:

- `reachable_true()` :
  $[A, B, C, E, F, G, H, I, J, K, L, S, T]$ and the following cycles: $[J, K, L, M, J]$ and $[J, K, L, M, N, J]$ AND $[H, I, J, H]$.

- `reachable_unreachable()` : $[A, B, C, E, F, G, H, R, T]$ and the following cycles: $[J, K, L, M, J]$ and $[J, K, L, M, N, J]$ AND $[H, I, J, H]$.
- `reachable_missingSrc()` : $[A, B, C, O, T]$
- `reachable_missingTarget()` : $[A, B, C, D, P, T]$

These test cases achieve **neither** *Node* nor *Edge* coverage for `CFG.isReachable()` method. Additional test case to achive *Node* coverage:

```
@Test
public void reachable_start_equals_end() {
    assertTrue(cfg.isReachable(20, m_m, m, 20, m_m, m));
}
```

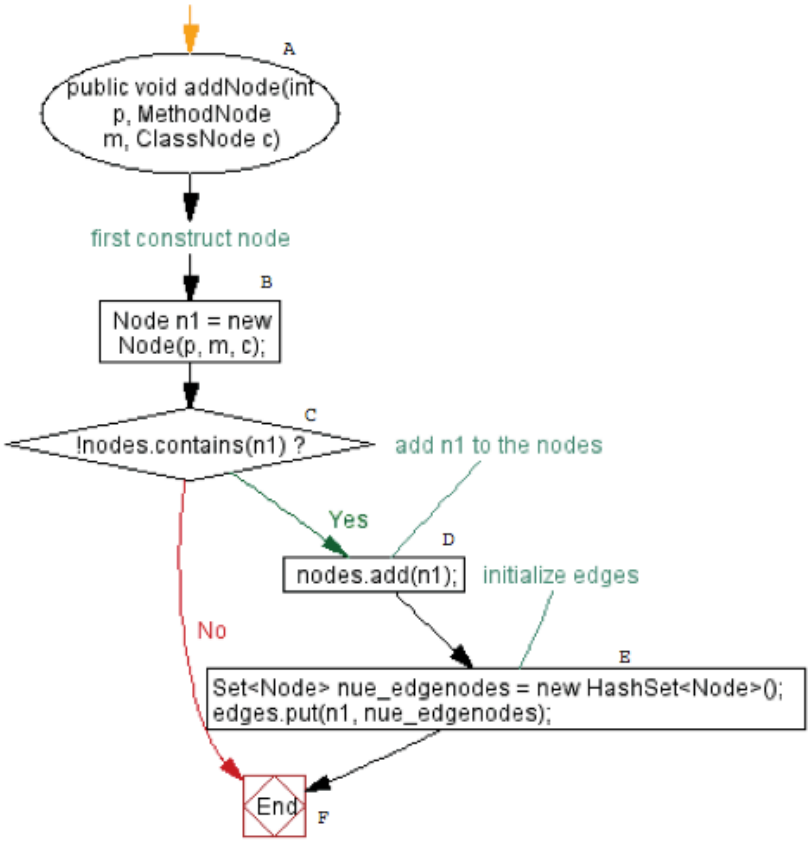

Figure 4: addNode()

Marking Scheme

After adding the test case `reachable_start_equals_end()`, there are 14 test cases totally:

- void addNode() 1'
- void addNode_duplicate() 1'
- void addEdge() 1'

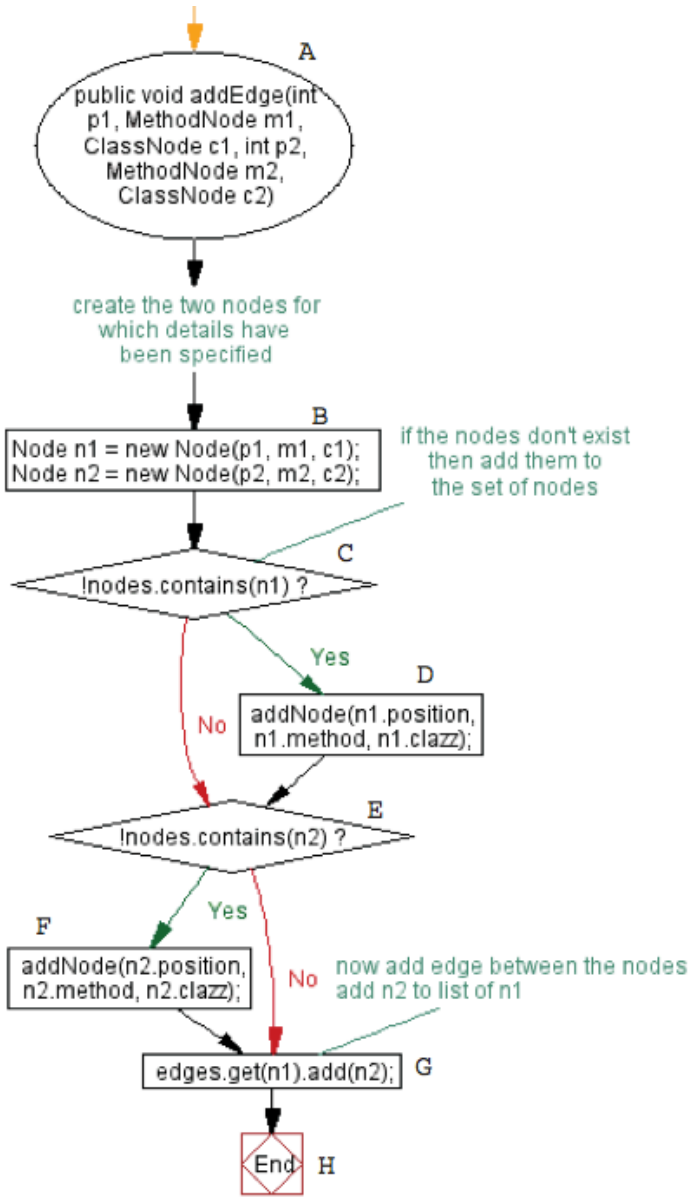

Figure 5: addEdge()

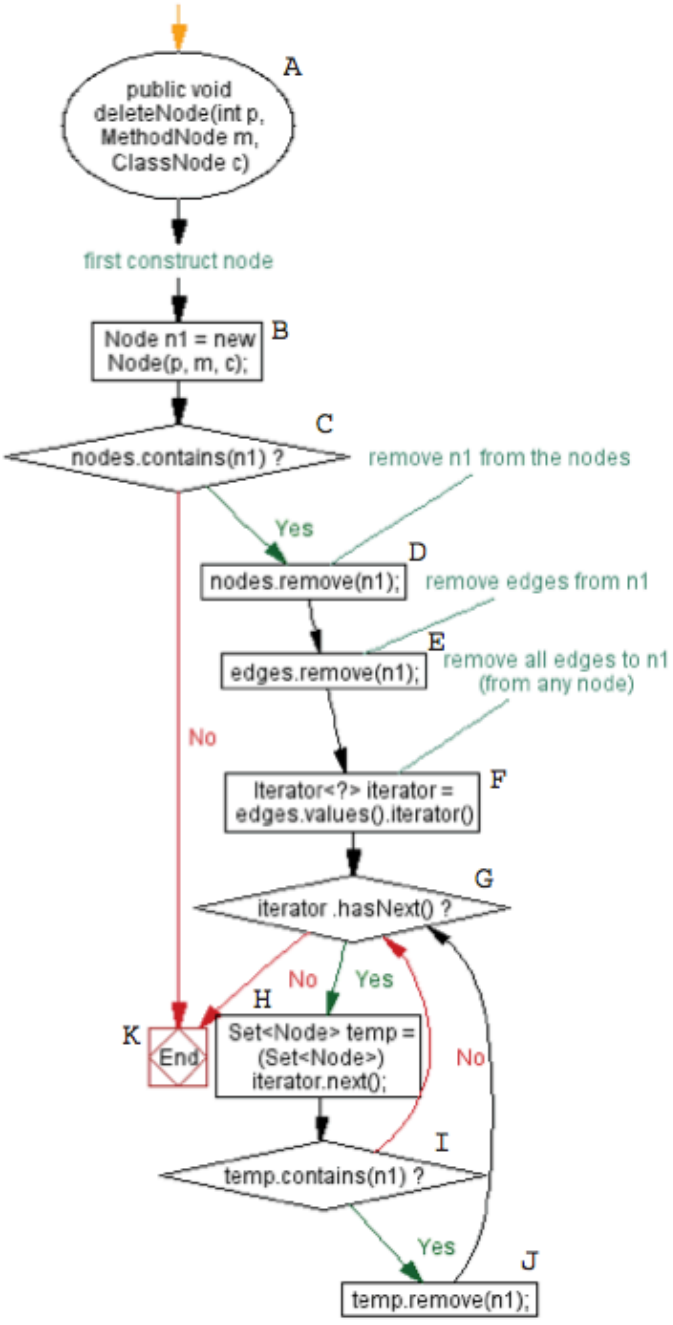

Figure 6: deleteNode()

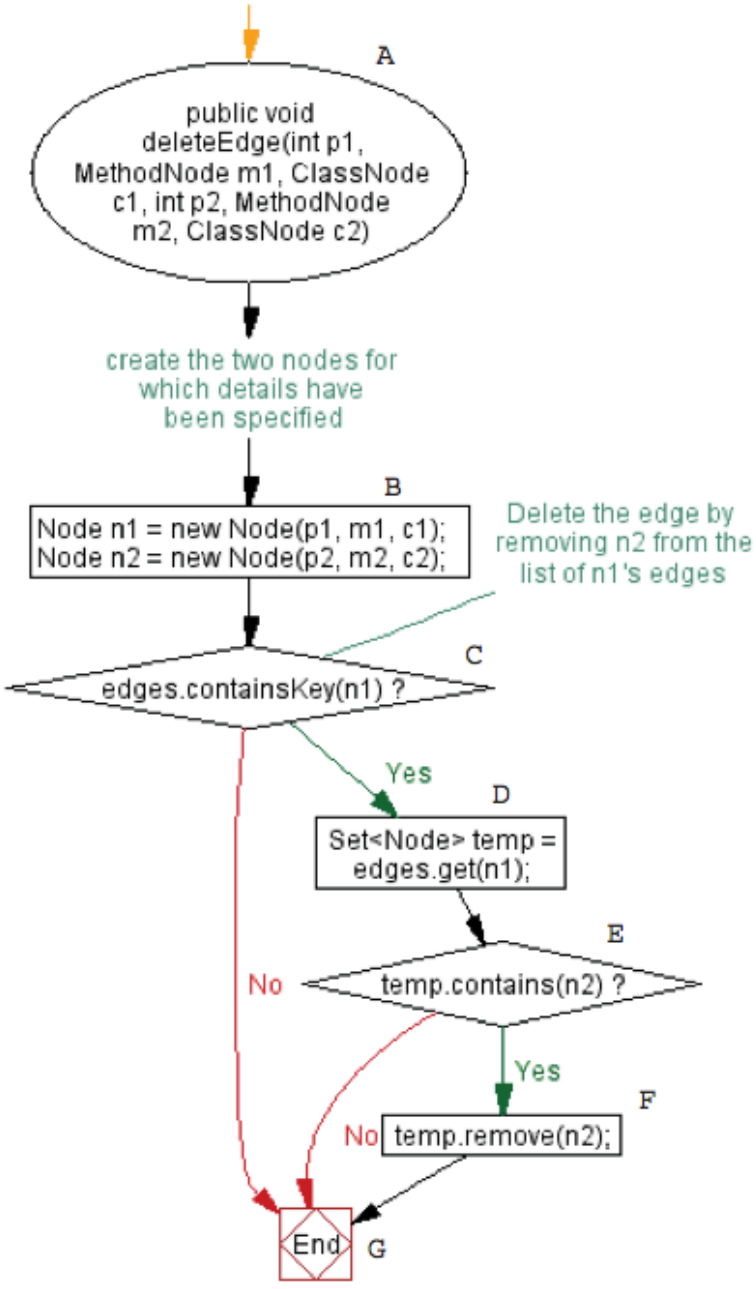

Figure 7: deleteEdge()

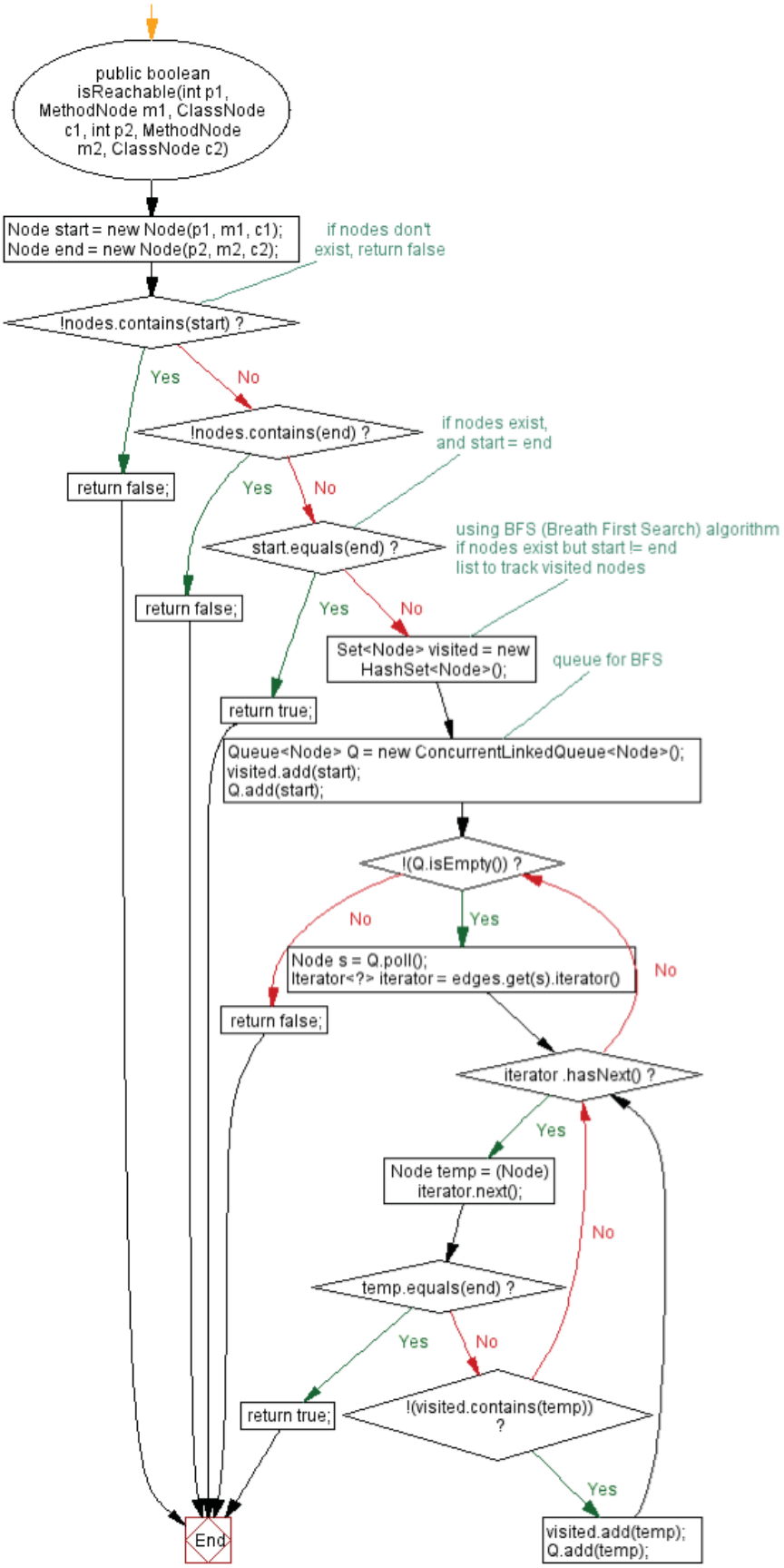

Figure 8: isReachable()

- void addEdge_oneNewNode() 1'
- void deleteNode() 2'
- void deleteNode_missing() 2'
- void deleteEdge 2'
- void deleteEdge_missing() 1'
- void deleteEdge_missingSrcNode() 1'
- void reachable_true() 2'
- void reachable_unreachable() 1'
- void reachable_missingSrc() 1'
- void reachable_missingTarget() 1'
- void reachable_start_equals_end() 1'

If you added test cases that failed, you also get -1.

For 4(f), if the answers do not have enough explainations (e.g. only yes/no node/edge coverage, only state the definitions of node/edge overage), you will lose up to 8 points for explainations. In this question, You will NOT lose points if you miss some corner cases in your implementation. However, if the fundemantal test cases for each method (e.g. test case addEdge()) fail, you will lose 2 points for each method. For the code I cannot compile, your mark for this question is currently 0. I have tried to fix some problems (e.g., fix typos). If you get the comment about 'code cannot compile' and think it only requires a few changes, you can email me the way to fix it or make an appointment, so I can remark your assignment.