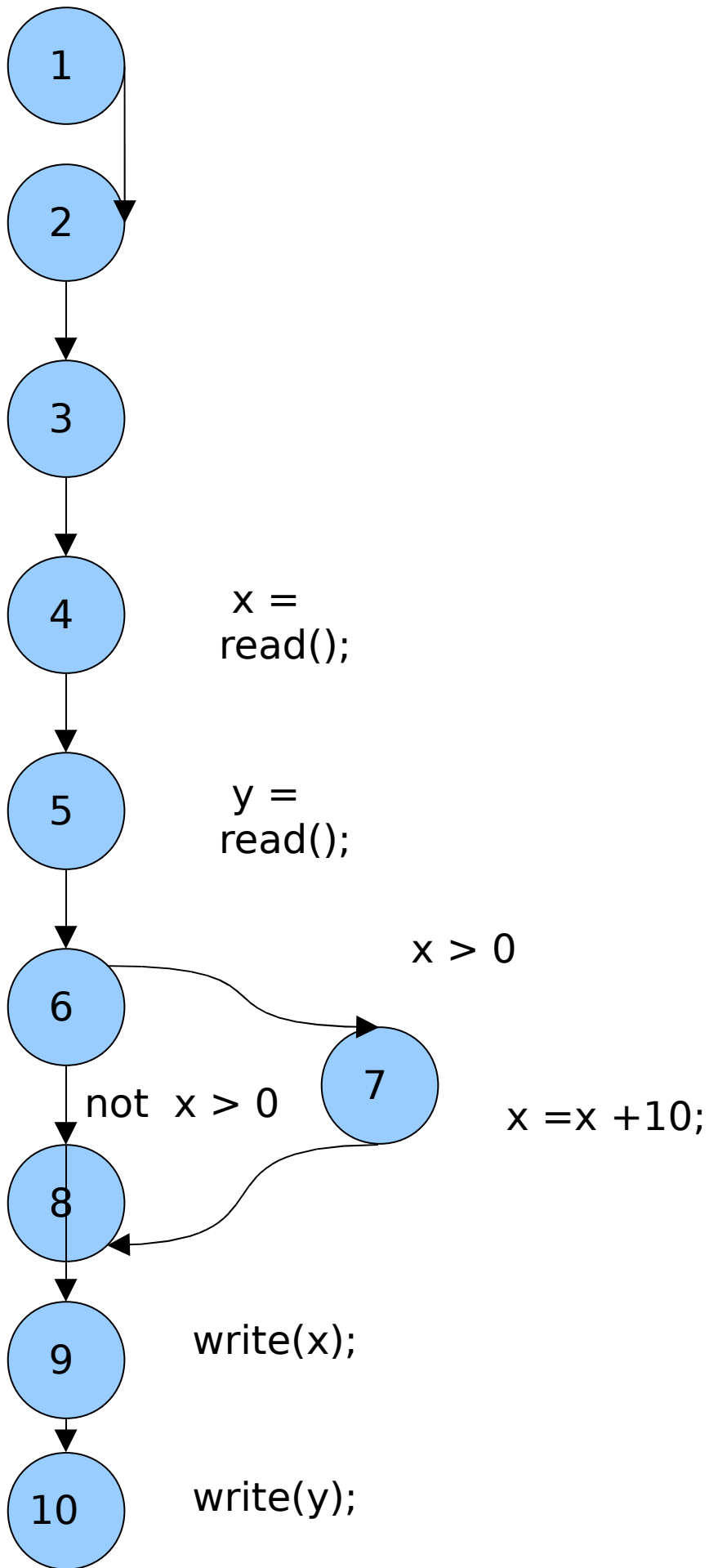


Exercise 1

```
1. void f() {  
2.     float x;  
3.     float y;  
4.     x = read();  
5.     y = read();  
6.     if (x > 0)  
7.         x += 10;  
8.     y = y / x;  
9.     write(x);  
10.    write(y);  
11. }
```

- Create the control flow graph of this program



- Write test cases using the following coverage criteria. Do they identify the fault ?

- Statement coverage

T1 ($x = 20, y = 30$);

- Branch coverage

T1, T2 ($x = -1, y = 30$);

Notice: a test T satisfies branch coverage criteria if each edge in the CFG is executed at least once ; if a test satisfies branch test then it satisfies statement test, but not viceversa. In fact, T1 does not satisfy branch coverage criterion: edge (6, 8) is never traversed.

In order to obtain Branch coverage too, we need to add T2.

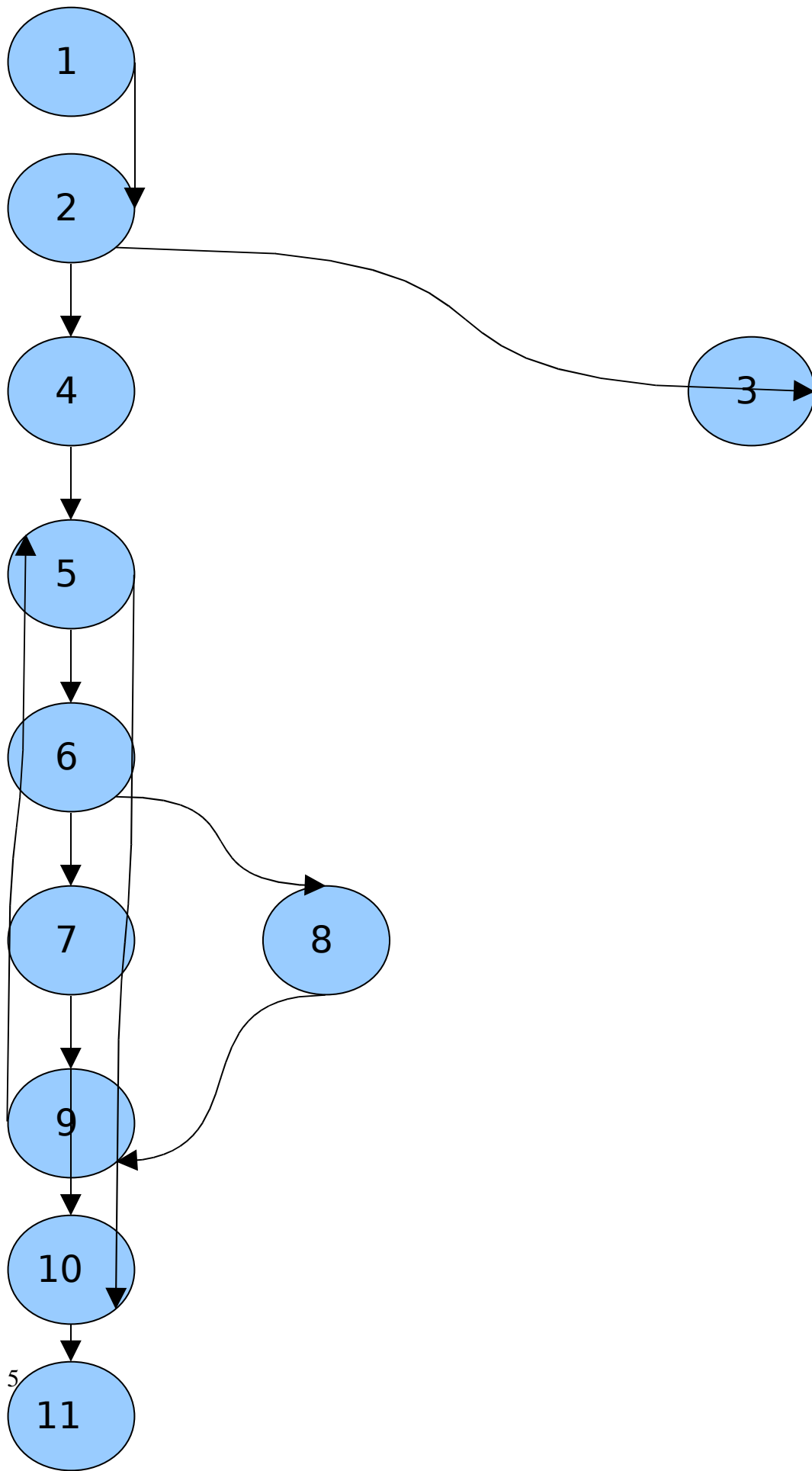
T1 and T2, even if they permit to have 100% branch and statement coverage, don't find out the fault.

To find out the fault, we should add a further test T3($x=0,y=any\ value$): fault is a is at line 8, and it's a division by 0.

Exercise 2

```
1. public int max_absolute(int[] numbers){
2.     if(numbers.length > 5)
3.         return -1;
4.     int max_value = 0;
5.     for(int i = 0; i<numbers.length; i++){
6.         if (numbers[i] < 0 )
7.             max_value =
Math.max(max_value,Math.abs(numbers[i]));
8.         else max_value = Math.max(max_value, numbers[i]);
9.     }
10.    return max_value;
11. }
```

- Draw the control graph (next page)



- The function is tested with the following test cases (separately): for each or them, define statement, branch and loop ratio coverage.

```

T1 int[] all_equals = {0,0,0,0,0}; ( 0 )
T2 int[] all_positive = {1,2,3,4,5}; ( 5 )
T3 int[] all_negative = {-1,-2,-3,-4,-5}; ( 5 )
T4 int[] out_of_size = {1,2,3,4,5,6}; ( -1 )
T5 int[] mixed = {-10,10,3,5,-6}; ( 10 )
T6 int[] empty = {}; ( -1 )

```

| ARRAY | % STATEMENT COVERAGE | % BRANCH COVERAGE | % LOOP COVERAGE * |
|--------------|----------------------|-------------------|-------------------|
| all_equals | 9/11 | 9/13 | 1/3 |
| all_positive | 9/11 | 9/13 | 1/3 |
| all_negative | 9/11 | 9/13 | 1/3 |
| out_of_size | 3/11 | 2/13 | 1/3 |
| mixed | 10/11 | 11/13 | 1/3 |
| empty | 7/11 | 6/13 | 1/3 |

*Definition of **Loop coverage**: select test cases such that every loop boundary and interior is tested
Boundary: 0 iterations ; Interior: 1 iteration and > 1 iterations

- Which test case will fail?

`int[] empty = {};` will fail because 0 is returned instead of -1 is expected (empty array is bad input)

- Which combination of tests can assure 100% coverage of branches, statements and loops ?

Given T7 = `T6 int[] one_element = {3};`

T_FULL = { T4, T5, T6, T7 } will give a 100% coverage of statements, branches, loops.

Exercise 3

Assume following specification for some piece of code which could be part of a bisection algorithm to find $\pi/2$:

- Input parameters are the float values a and b .
- Swap a and b unless $a \leq b$.
- Set a and b to 1 and 3 unless $\cos(a) \geq 0$ and $\cos(b) \leq 0$.
- Set x to the arithmetic mean of a and b .
- Set a to x if $\cos(x) > 0$ and b to x otherwise.
- Print a and b .

```

1. if (a > b) {
2.     float tmp=b; b = a; a = tmp;
3. }
4. if (cos(a) < 0 || cos(b) > 0) {
5.     a = 1; b = 3;
6. }
7. x = (a + b) / 2;
8. if (cos(x) > 0) {
9.     a = x;
10. } else {
11.     b = x;
12. }
```

1. Write test cases that are able to cover the following situations:

- (A) swap code at line 2
- (B) line 2 is not executed
- (C) line 5 is executed
- (D) line 5 is not executed
- (E) line 9 is executed
- (F) Line 11 is executed

| Coverage | Input | Coverage | Desired | Observed |
|----------|-------|----------|---------|----------|
| C0: | 5,2 | ACF (TT) | 1,2 | 1,2 |
| | 8,6 | ADE (FF) | 7,8 | 7,8 |

| | | | | |
|-----|-----|----------|-----|-----|
| C1: | 2,5 | BCF (TT) | 1,2 | 1,2 |
| C2: | 4,4 | BDF (TF) | 4,4 | 1,2 |
| | 7,5 | ADE (FT) | 6,7 | 1,2 |
| C7: | 4,8 | BDE (TF) | 6,8 | 1,2 |
| | 9,7 | ADF (FF) | 7,8 | 7,8 |

(TT), (FF), (FT), and (TF) specify the boolean values of the two conditions within the (faulty) second if statement.

C0 tries to cover all statements but not the empty else clauses, i.e. B is not included.

C1 adds a test case for B.

C2 completes the set of possible boolean values (there were just (TT) and (FF) so far).

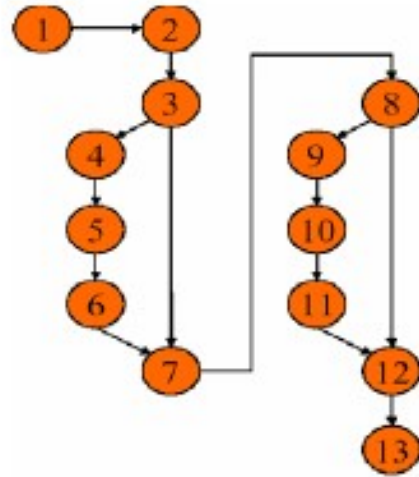
C7 adds BDE and ADF which were missing from the set of 6 possible paths. Note that ACE and BCE are impossible because if C is taken, E can no longer be reached.

Exercise 4

```

1.      public int[] order(int v[])
2.      {
3.          int tmp;
4.          if (v[0]<v[1]) {
5.              tmp = v[0];
6.              v[1] = v[1];
7.              v[1] = tmp;
8.          }
9.          if (v[1]<v[2]) {
10.             tmp = v[0];
11.             v[1] = v[2];
12.             v[2] = tmp;
13.          }
14.          return v;
      }

```



Test cases:

- Node coverage
 - TC1: { 1, 2, 3 } 3 ; {3,2,1}
- Edge coverage: TC1 +
 - TC2: { 3, 2, 1 } 3 ; {3,2,1}
- Path coverage: TC1+TC2+
 - TC3: { 2, 3, 1 } 3 ; {3,2,1}
 - TC4: { 3, 1, 2 } 3 ; {3,2,1}

Exercise 5

```
public double elonianTaxCalculator(double income, int nDependents) {

    double TaxSubTotal,Exemption,TaxTotal;

    // first if - check income
    if (income < 0) {
        System.out.println("You cannot have a negative
income.\n");
        return -1;
    }

    // second if - check dependents
    if (nDependents <= 0) {
        System.out.println("You must have at least one
dependent.\n");
        return -2;
    }

    // third if (else-if) - compute tax subtotal
    if (income < 10000)
        TaxSubTotal = .02 * income;
    else if (income < 50000)
        TaxSubTotal = 200 + .03 * (income - 10000);
    else
        TaxSubTotal = 1400 + .04 * (income - 50000);
    Exemption= nDependents * 50;
    TaxTotal=TaxSubTotal - Exemption;

    // last if - check negative tax

    if (TaxTotal<0) //In case of negative tax
        TaxTotal=0;

    System.out.println(
"$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ \n");
    System.out.println( "Elbonian Tax Collection Agency
\n");

    System.out.println( "Tax Bill \n");
    System.out.println( "Citizen's Income: " + income + '\
```


| | | | | |
|----------|-------|--|----------|--|
| ≥ 0 | > 0 | | ≥ 0 | |
|----------|-------|--|----------|--|

To test this, we must come up with eight sets of data (income and number of dependents), one to test each possible path. Ranges for income are fairly evident for each case; we need only select an appropriate number of dependents for each case. The following table shows one test suite, and the expected results (TaxTotal):

| Income | NDependents | Expected Result |
|--------|----------------|-----------------------------|
| -5 | Doesn't matter | negative income error |
| 0 | 0 | invalid dependents error |
| 100 | 1 | 0 (bracket 1, negative tax) |
| 20000 | 11 | 0 (bracket 2, negative tax) |
| 50000 | 100 | 0 (bracket 3, negative tax) |
| 9000 | 1 | 130 (bracket 1) |
| 15000 | 1 | 300 (bracket 2) |
| 100000 | 1 | 3350 (bracket 3) |

Exercise 6

```
1. float foo (int a, int b, int c, int d, float e) {
2.     float e;
3.     if (a == 0) {
4.         return 0;
5.     }
6.     int x = 0;
7.     if ((a==b) || ((c == d) && bug(a) )) {
8.         x=1;
9.     }
10.    e = 1/x;
11.    return e;
12. }
```

Function bug(a) should return a value of true when passed a value of a=1.

- full method coverage

T1 = foo(0, 0, 0, 0, 0), 0.

All methods of our program are tested at least once...we have just one method !

- full statement coverage

T1 +

T2 = foo(1,1, 1, 1, 1.) , 1.

T1 covers lines 1-5, T2 covers lines 6-12.

- full branch coverage

Decision or branch coverage is a measure of how many of the Boolean expressions of the program have been evaluated as both true and false in the testing.

The program has 2 decision points:

```
3.     if (a == 0) {
7.     if ((a==b) || ((c == d) && bug(a) )) {
```

We need to ensure that each of these predicates (compound or single) is tested as both true and false.

Since with T1 and T2 we achieved 75% of branch coverage, we need to add a further test:

T1 + T2 +
T3 = foo(1, 2, 1, 2, 1).

The last added test (T3) is able to discover a division by zero problem on line 10.

- full condition coverage

Condition coverage reports the true or false outcome of each Boolean sub-expression of a compound predicate.

Notice that in line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a). Condition coverage measures the outcome of each of these sub-expressions independently of each other. With condition coverage, you ensure that each of these sub-expressions has independently been tested as both true and false. We consider our progress thus far in the following table:

| Predicate | True | False |
|-----------|--------|---------------|
| a == b | T2 , 0 | T3, div by 0! |
| c == d | | T3, div by 0! |
| bug(a) | | |

We add two new tests to achieve the 100% condition coverage:

T4 foo(1, 2, 1, 1, 1)

T5 foo(3, 2, 1, 1, 1)

| Predicate | True | False |
|-----------|--------|---------------|
| a == b | T2 , 0 | T3, div by 0! |
| c == d | T4, 1 | T3, div by 0! |
| bug(a) | T4, 1 | T5 |

!: Notice that 100% condition coverage **isn't** 100% multiple condition coverage(mcc). In order to obtain 100% mcc, we need 2ⁿ tests (the full truth table), where n is the number of inputs.

Exercise 7

A function converts a sequence of chars in an integer number. The sequence can start with a '-' (negative number). If the sequence is shorter than 6 chars, it is filled with blanks (to the left side).

The integer number must be in the range $\text{minint} = -32768$ to $\text{maxint} = 32767$.

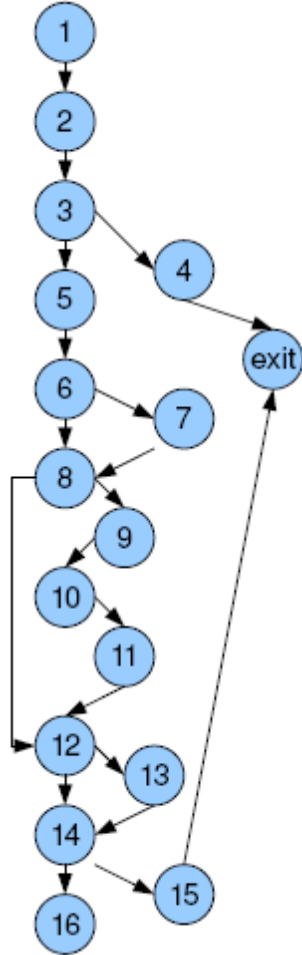
The function signals an error if the sequence of chars is not allowed.

```
1. public class ConvertInt {
2.     public int convert(char[] str) throws Exception{
3.         if (str.length > 6)
4.             throw new Exception();
5.         int number=0;int digit; int i=0;
6.         if (str[0]=='-')
7.             i=1;
8.         for(; i<str.length; i++){
9.             digit = str[i] - '0';
10.            number = number * 10 + digit;
11.        }
12.        if (str[0]=='-')
13.            number = -number;
14.        if (number > 32767 || number < -32768)
15.            throw new Exception();
16.        return number;
17.    }
18. }
```

Write tests in order to achieve:

- node coverage
- edge coverage
- condition coverage for all if statements
- multiple condition coverage of if statement at line 14
- path coverage

Flow:



- Node coverage

Nodes: 16

| TEST-EXPECTED OUTPUT | NODES COVERED | CUMULATIVE NODE COVERAGE |
|---------------------------------|-----------------------------------|--------------------------|
| T1:convert("256")256 | 1,2,3,5,6,8,9,10,11,12,14,16 | 12/16 |
| T2:convert("-100");-100 | 1,2,3,5,6,7,8,9,10,11,12,13,14,16 | 14/16 |
| T3:convert("1234567");exception | 1,2,3,4 | 15/16 |
| T4:convert("32800");exception | 1,2,3,5,6,8,9,10,11,12,15 | 16/16 |

- Edge coverage

Edges: 18

| TEST ; EXPECTED OUTPUT | CUMULATIVE EDGE COVERAGE |
|---------------------------------|--------------------------|
| T1:convert("256");256 | 11/18 |
| T2:convert("-100");-100 | 15/18 |
| T3:convert("1234567");exception | 16/18 |
| T4:convert("32800");exception | 17/18 |
| T9: convert("-"); 0 | 18/18 |

- Condition coverage

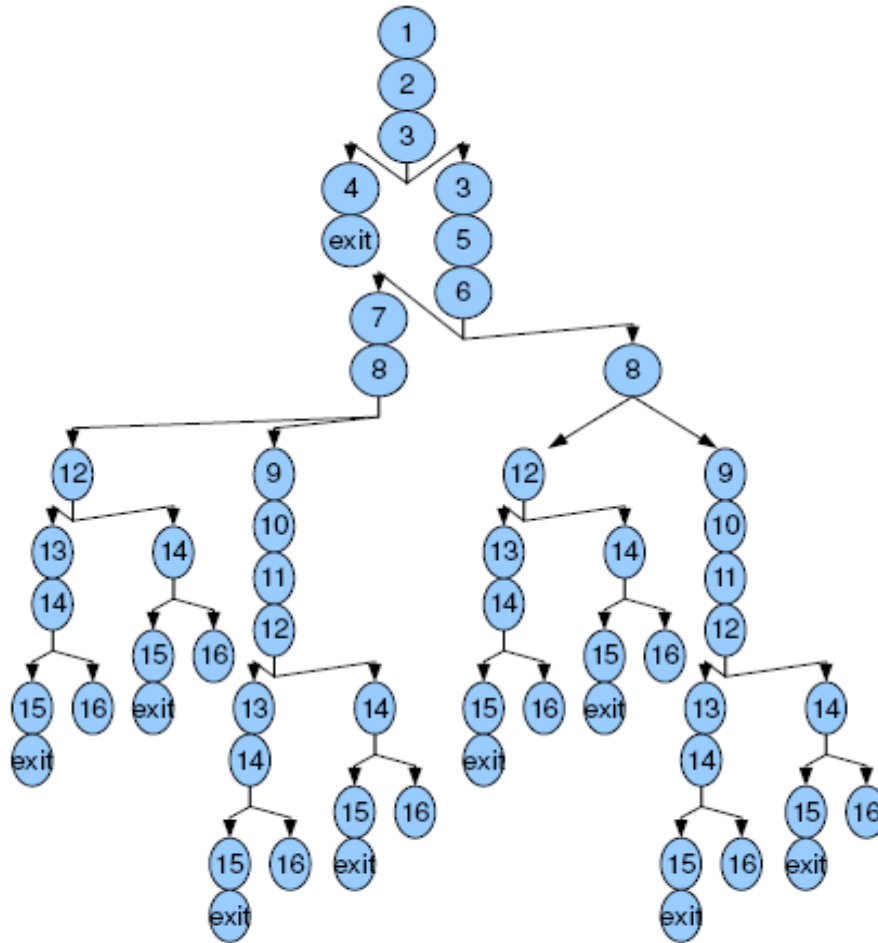
| IF | LINE | TRUE | FALSE |
|---|----------|--------------------------|-----------------------|
| if (str.length > 6) | 3 | convert("1234567")- exc. | T5:convert("352")-352 |
| if (str[0]=='-') | 6 and 12 | T6:convert("-23")- -23 | convert("352")-352 |
| if (number > 32767 number < -32768) | 14 | convert("352")- 352 | convert("352")-352 |

- Multiple condition coverage on if at line 14

| | | number > 32767 | |
|---------------|-------|-------------------------------|--------------------------------|
| | | TRUE | FALSE |
| number<-32768 | TRUE | Impossible case | T7:convert(-32800) - exception |
| | FALSE | T8: convert(33000) -exception | T11: convert(125) - 125 |

- Path coverage

There are 15 Possible paths (but, as you will see, a lot of them never happen):



| PATH | TEST | EXPECTED OUTPUT |
|--|-------------------------------|-----------------|
| 1-2-3-4-exit | convert("1234567") | exception |
| 1-2-3-5-6-7-8-12-13-14-15-exit | convert("-32800") | exception |
| 1-2-3-5-6-7-8-12-13-14-16 | convert("-") | 0 |
| 1-2-3-5-6-7-8-12-14-15-exit | It never happens ¹ | - |
| 1-2-3-5-6-7-8-12-14-16 | It never happens ² | - |
| 1-2-3-5-6-7-8-9-10-11-12-13-14-15-exit | T13 :convert("-33000") | exception |

1 Because if the if statement at line 6 is true, the if statement at line 12 is also true(the condition is the same)

2 See 1

| | | |
|--------------------------------------|-------------------------------|-----------|
| 1-2-3-5-6-7-8-9-10-11-12-13-14-16 | T10: convert("-30000") | -30000 |
| 1-2-3-5-6-7-8-9-10-11-12-14-15-exit | It never happens ³ | - |
| 1-2-3-5-6-7-8-9-10-11-12-14-16 | It never happens ⁴ | - |
| 1-2-3-5-6-8-9-10-11-12-13-14-15-exit | It never happens ⁵ | - |
| 1-2-3-5-6-8-9-10-11-12-13-14-16 | It never happens ⁶ | - |
| 1-2-3-5-6-8-9-10-11-12-14-15-exit | convert("33000") | exception |
| 1-2-3-5-6-8-9-10-11-12-14-16 | convert("125") | 125 |
| 1-2-3-4-5-6-8-12-13-14-15-exit | It never happens ⁷ | - |
| 1-2-3-4-5-6-8-12-13-14-16 | It never happens ⁸ | - |
| 1-2-3-4-5-6-8-12-14-15-exit | It never happens ⁹ | - |
| 1-2-3-4-5-6-8-12-14-16 | T12: convert("") | 0 |

Finally, we have discovered that 9 paths over 15 are impossible to happen.

3 See 1

4 See 1

5 Because if the if statement at line 6 is false, the if statement at line 12 is also false(the condition is the same)

6 See 5

7 See 5

8 See 5

9 Because the only case the program doesn't enter the for loop (when the first character isn't '-') is convert("-"),that never arises exception

Exercise 8

A queue of events in a simulation system receives events. Each event has a time tag. It is possible to extract events from the queue, the extraction must return the event with lower time tag.

The queue discards events with negative or null time tag.

The queue must accept at least 100.000 events.

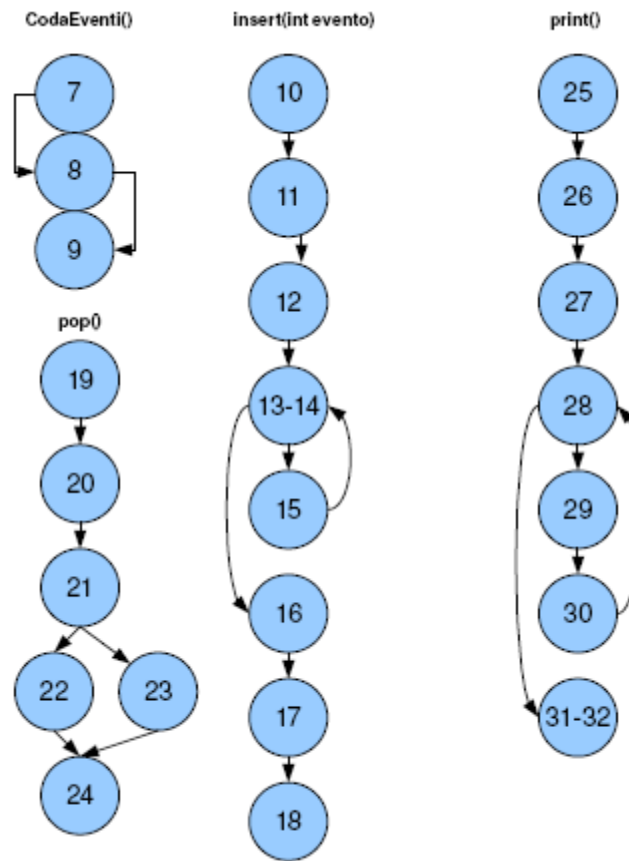
Events with the same time tag must be merged (i.e. the second received is discarded)

Define test cases to achieve

- Node coverage
- Edge coverage

```
1. import java.util.Iterator;
2. import java.util.LinkedList;
3.
4. public class EventsQueue {
5.     private LinkedList queue;
6.
7.     public EventsQueue(){
8.         queue = new LinkedList();
9.     }
10.    public void insert(int event){
11.        int index = 0;
12.        int size = queue.size();
13.        while (index < size &&
14.            ((Integer)queue.get(index)).intValue() < event){
15.            index++;
16.        }
17.        queue.add(index, new Integer(event));
18.    }
19.    public int pop(){
20.        Object o = queue.getFirst();
21.        if (o != null)
22.            return ((Integer) o).intValue();
23.        else return -1;
24.    }
25.    public void print(){
26.        Iterator i = queue.iterator();
27.        int event;
28.        while (i.hasNext()){
29.            event = ((Integer) i.next()).intValue();
30.            System.out.println(event + " ");
31.        }
32.    }}
```

Control flows :



For the sake of semplicity, nodes from 1 to 6 are not considered

- Edge coverage

| | Edges covered | Cumulative edge coverage |
|---|--|--------------------------|
| <code>EventsQueue eq = new EventsQueue()</code> | 2 = {7-8, 8-9} | 2/23 = 8.7% |
| <code>eq.print()</code> | 4 = {25-26,26-27,27-28,28-(31,32)} | 6/23 = 26.1% |
| <code>eq.pop()</code> | 4={19-20,20-21,21-23,23-24} | 10/23 = 43.5% |
| <code>eq.insert(1)</code> | 6={10-11,11-12,12-(13,14),(13,14)-16,16-17,17-18} | 16/23= 69.6% |
| <code>eq.insert(2)</code> | 8 = {10-11,11-12,12-(13,14),(13,14)-15,15-(13,14), (13,14)-16,16-17,17-18} | 18/23 = 78.3% |
| <code>eq.print()</code> | 7 = {25-26,26-27,27-28,28-29,29-30,30-28,28-(31,32)} | 21/23 = 91.3% |

| | | |
|-----------------------|--|--------------|
| <code>eq.pop()</code> | 4 = {19-20,20-21, 21-22,22-24 } | 23/23 = 100% |
|-----------------------|--|--------------|

When an edge that is covered by a method's call is visited for the first time, it's **highlighted in yellow**.

- Node coverage

Since edge coverage implies node coverage, we can reuse the same sequence of test we used above.

| | Nodes covered | Cumulative edge coverage |
|---|---|--------------------------|
| <code>EventsQueue eq = new EventsQueue()</code> | 3 = { 7, 8, 9 } | 3/24 = 12.5% |
| <code>eq.print()</code> | 5 = { 25,26,27,28,(31,32) } | 8/24 = 33.3% |
| <code>eq.pop()</code> | 5={ 19,20,21,23,24 } | 13/24 = 54.2% |
| <code>eq.insert(1)</code> | 7={ 10,11,12,(13,14),16,17,18 } | 20/24= 83.3% |
| <code>eq.insert(2)</code> | 8 = {10,11,12,(13,14), 15 ,16,17,18} | 21/24 = 87.5% |
| <code>eq.print()</code> | 7 = {25,26,27, 29,30 ,28,(31,32)} | 23/24 = 95.8% |
| <code>eq.pop()</code> | 4 = {19,20,21, 22,24 } | 24/24 = 100% |

When a node that is covered by a method's call is visited for the first time, it's **highlighted in yellow**.