Structural Testing (white box) Exercises

```
public boolean remove(Object o) {
01.  if (o == null) {
02.    for (Node<E> x = first; x != null; x = x.next) {
03.      if (x.item == null) {
04.        unlink(x);
05.        return true;
         }
       }
06.  } else {
07.    for (Node<E> x = first; x != null; x = x.next) {
08.      if (o.equals(x.item)) {
09.        unlink(x);
10.        return true;
         }
       }
     }
11.  return false;
}
```

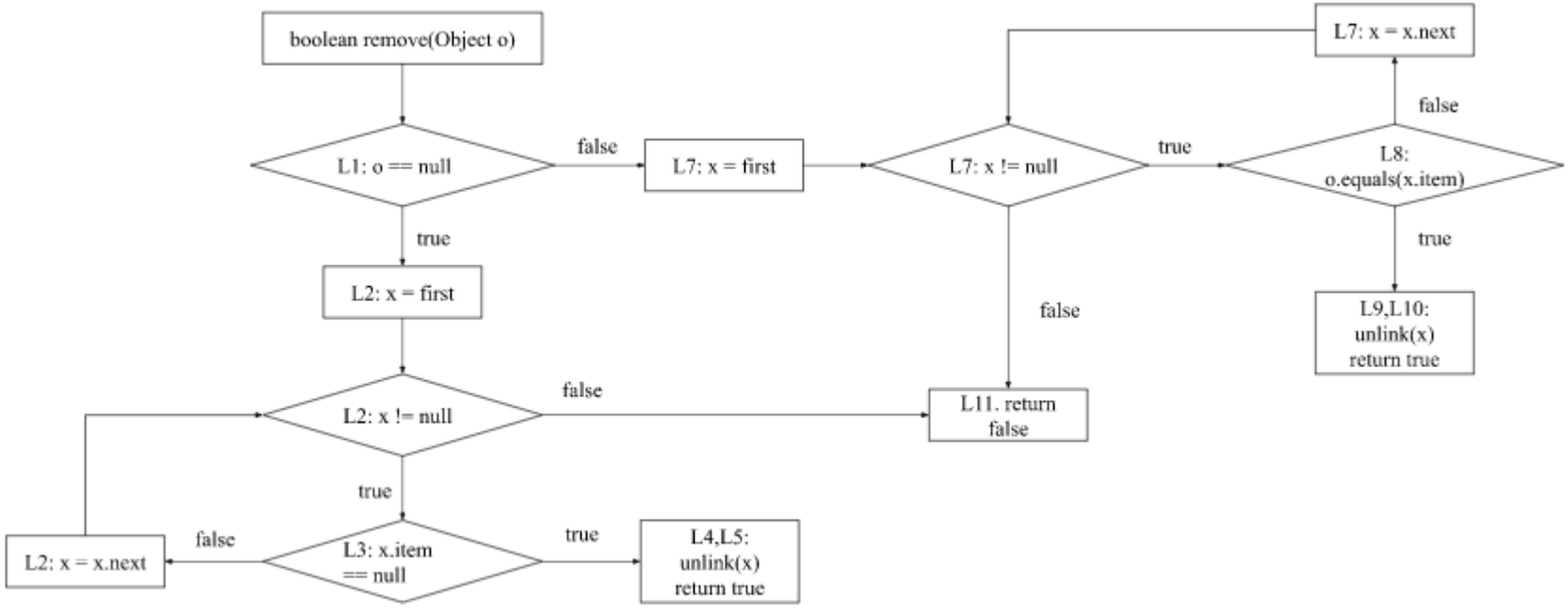
This is the implementation of JDK8's LinkedList remove method. Source: OpenJDK.

Exercise 1. Give a test suite (i.e. a set of tests) that achieves 100% **line** coverage on the `remove` method. Use as few tests as possible.
The documentation on Java 8's LinkedList methods, that may be needed in the tests, can be found in its Javadoc.

```java
@Test
public void removeNullInListTest() {
    LinkedList<Integer> list = new LinkedList<>();

    list.add(null);

    assertTrue(list.remove(null));
}

@Test
public void removeElementInListTest() {
    LinkedList<Integer> list = new LinkedList<>();

    list.add(7);

    assertTrue(list.remove(7));
}

@Test
public void removeElementNotPresentInListTest() {
    LinkedList<Integer> list = new LinkedList<>();

    assertFalse(list.remove(5))
}
```

Note that there exists a lot of test suites that achieve 100\%100% line coverage, this is just an example.
You should have 3 tests. At least one test is needed to cover lines 4 and 5 (`removeNullInListTest` in this case). This test will also cover lines 1-3.
Then a test for lines 9 and 10 is needed (`removeElementInListTest`). This test also covers lines 6-8.
Finally a third test is needed to cover line 11 (`removeElementNotPresentInListTest`).

Exercise 2. Create the control-flow graph (CFG) for the `remove` method.



Exercise 3. Look at the CFG you just created. Which of the following sentences **is false**?

1. A minimal test suite that achieves 100% basic condition coverage has more test cases than a minimal test suite that achieves 100% branch coverage.
2. The method `unlink()` is for now treated as an 'atomic' operation, but also deserves specific test cases, as its implementation might also contain decision blocks.
3. A minimal test suite that achieves 100% branch coverage has the same number of test cases as a minimal test suite that achieves 100% full condition coverage.
4. There exists a single test case that, alone, is able to achieve more than 50% of line coverage.

Option 1 is the false one.

A minimal test suite that achieves 100\% (either basic or full) condition has the same number of tests as a minimal test suite that achieves 100\% branch coverage. All decisions have just a single branch, so condition coverage doesn't make a difference here. Moreover, a test case that exercises lines 1, 6, 7, 8, 9, 10 achieves around 54\% coverage (6/11).

Exercise 4. Give a test suite (i.e. a set of tests) that achieves 100% **branch** coverage on the `remove` method. Use as few tests as possible.
The documentation on Java 8's LinkedList methods, that may be needed in the tests, can be found in its Javadoc.

Example of a test suite that achieves 100% branch coverage:

```java
@Test
public void removeNullAsSecondElementInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  list.add(5);
  list.add(null);

  assertTrue(list.remove(null));
}

@Test
public void removeNullNotPresentInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  assertFalse(list.remove(null));
}

@Test
public void removeElementSecondInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  list.add(5);
  list.add(7);

  assertTrue(list.remove(7));
}

@Test
public void removeElementNotPresentInListTest() {
  LinkedList<Integer> list = new LinkedList<>();

  assertFalse(list.remove(3));
}
```

This is just one example of a possible test suite. Other tests can work just as well. You should have a test suite of 4 tests.

With the CFG you can see that there are decisions in lines 1, 2, 3, 7 and 8. To achieve 100% branch coverage each of these decisions must evaluate to true and to false at least once in the test suite.

For the decision in line 1, we need to remove `null` and something else than `null`. This is done with the `removeElement` and `removeNull` tests.
Then for the decision in line 2 the node that `remove` is looking at should not be null and null at least once in the tests. The node is `null` when the end of the list had been reached. That only happens when the element that should be removed is not in the list. Note that the decision in line 2 only gets executed when the element to remove is `null`. In the tests, this means that the element should be found and not found at least once.
The decision in line 3 checks if the node that the method is at now has the element that should be deleted. The tests should cover a case where the element is not the item that has to be removed and a case where the element is the item that should be removed.

The decisions in lines 7 and 8 are the same as in lines 2 and 3 respectively. The only difference is that lines 7 and 8 will only be executed when the item to remove is not `null`.

Exercise 5. Consider the decision `(A or C) and B` with the corresponding decision table:

Decision	A	B	C	(A | C) & B
1	T	T	T	T
2	T	T	F	T
3	T	F	T	F
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

What is the set with the minimum number of tests needed for 100% MC/DC (Modified Condition / Decision Coverage)?

First, we find the pairs of tests that can be used for each of the conditions:

- A: {2, 6}
- B: {1, 3}, {2, 4}, {5, 7}
- C: {5, 6}

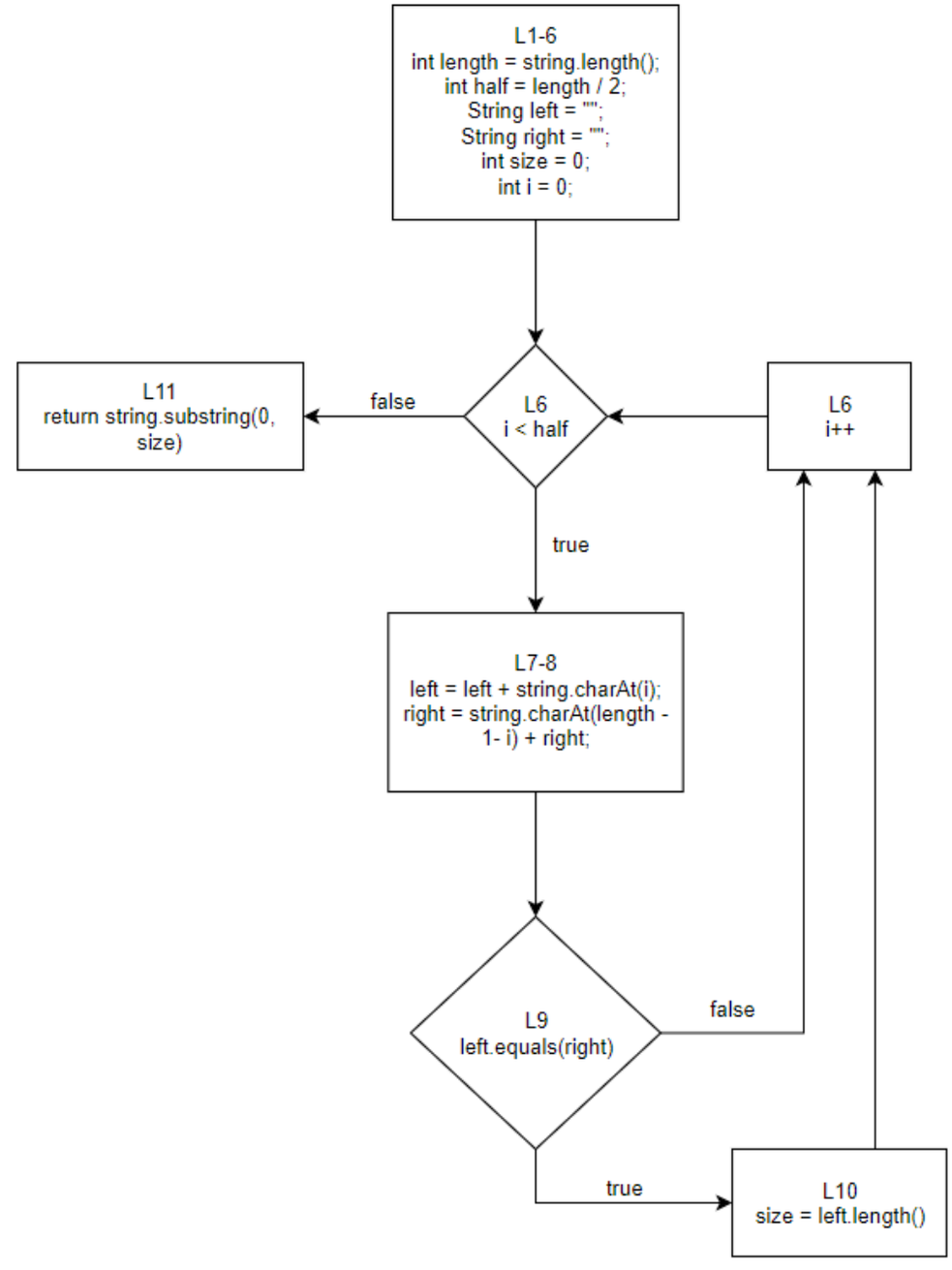
For A and C we need the decisions 2, 5 and 6. Then you can choose to add either 4 or 7 to cover condition B.

The possible answers are: {2, 4, 5, 6} or {2, 5, 6, 7}.

For the next three exercises use the code below. This method returns the longest substring that appears at both the beginning and end of the string without overlapping. For example, `sameEnds("abXab")` returns `"ab"`.

```
public String sameEnds(String string) {
01. int length = string.length();
02. int half = length / 2;

03. String left = "";
04. String right = "";

05. int size = 0;
06. for (int i = 0; i < half; i++) {
07.   left = left + string.charAt(i);
08.   right = string.charAt(length - 1 - i) + right;

09.   if (left.equals(right)) {
10.     size = left.length();
      }
    }

11. return string.substring(0, size);
}
```

Exercise 6. Draw the control-flow graph (CFG) of the source code above.



L\ represents the line numbers that the code blocks cover.

Exercise 7. Give a test case (by the input string and expected output) that achieves 100% line coverage.

A lot of input strings give 100% line coverage. A very simple one is `"aa"`. As long as the string is longer than one character and makes the condition in line 9 true, it will give 100% line coverage. For `"aa"` the expected output is `"a"`.

Exercise 8. Given the source code of the `sameEnds` method. Which of the following statements is **not correct**?

1. It is possible to devise a single test case that achieves 100% line coverage and 100% decision coverage.
2. It is possible to devise a single test case that achieves 100% line coverage and 100% (basic) condition coverage.
3. It is possible to devise a single test case that achieves 100% line coverage and 100% decision + condition coverage.
4. It is possible to devise a single test case that achieves 100% line coverage and 100% path coverage.

Option 4 is the incorrect one. The loop in the method makes it impossible to achieve 100% path coverage. This would require us to test all possible number of iterations. For the other answers we can come up with a test case: `"aXYa"`

Now consider this piece of code for the FizzBuzz problem. Given an integer `n`, it returns the string form of the number followed by `"!"`. So the integer 8 would yield `"8!"`. Except if the number is divisible by 3 it returns "Fizz!" and if it is divisible by 5 it returns "Buzz!". If the number is divisible by both 3 and 5 it returns "FizzBuzz!" Based on a CodingBat problem.

```
public String fizzString(int n) {
1.  if (n % 3 == 0 && n % 5 == 0)
2.        return "FizzBuzz!";
3.  if (n % 3 == 0)
4.       return "Fizz!";
5.  if (n % 5 == 0)
6.       return "Buzz!";
7.  return n + "!";
}
```

Exercise 9. Assume we have two test cases with an input integer: T1 = 15 and T2 = 8.

What is the branch+condition coverage these test cases give combined?

What is the decision coverage?

First the condition coverage. When talking about condition coverage, we first have to split the condition on line 1 (n % 3 == 0 && n % 5 == 0) into two decision blocks for the CFG. In total, we will have 8 conditions:

1. Line 1: `n % 3 == 0`, true and false
2. Line 1: `n % 5 == 0`, true and false
3. Line 3: `n % 3 == 0`, true and false
4. Line 5: `n % 5 == 0`, true and false

T1 makes conditions 1 and 2 true and then does not cover the other conditions. Thus:

- condition 1 = [true: exercised, false: not exercised]
- condition 2 = [true: exercised, false: not exercised]
- condition 3 = [true: not exercised, false: not exercised]
- condition 4 = [true: not exercised, false: not exercised].

At this moment, condition coverage = 2/8.

For T2, the input number 8 is neither divisible by 3, nor divisible by 5. However, since the && operator only evaluates the second condition when the first one is true, condition 2 is not reached. Therefore this test covers condition 1, 3 and 4 as false. We now have:

- condition 1 = [true: exercised, false: exercised]
- condition 2 = [true: exercised, false: not exercised]
- condition 3 = [true: not exercised, false: exercised]
- condition 4 = [true: not exercised, false: exercised].

In total, these test cases then cover $2 + 3 = 5$ conditions so the condition coverage is $\frac{5}{8} \cdot 100\% = 62.5\%$.

Now the decision coverage. We have 6 decisions:

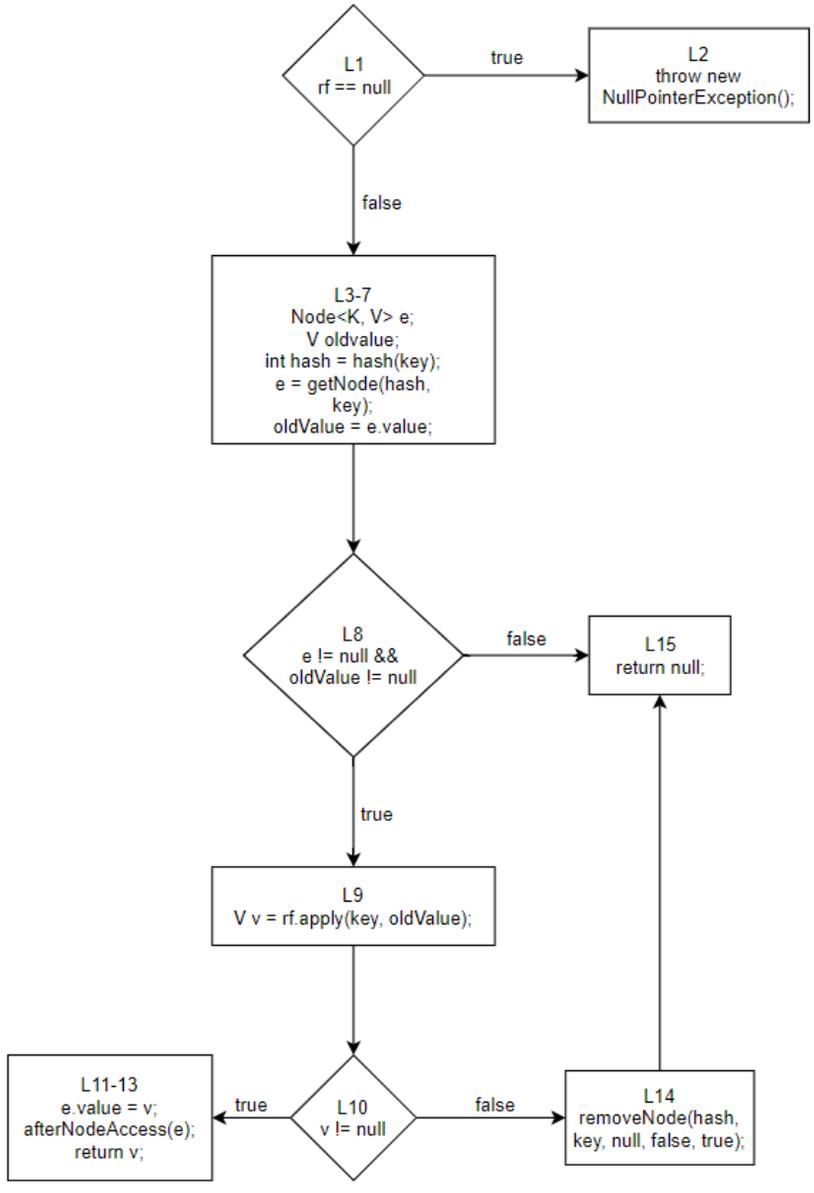
1. Line 1: `n % 3 == 0 && n % 5 == 0`, true and false
2. Line 3: `n % 3 == 0`, true and false
3. Line 5: `n % 5 == 0`, true and false

Now T1 makes decision 1 true and does not cover the other decisions. T2 makes all the decisions false. Therefore, the coverage is $\frac{4}{6} \cdot 100\% = 66\%$.

The next couple of exercises use Java's implementation of the LinkedList's `computeIfPresent()` method.

```
public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> rf) {
01. if (rf == null) {
02.   throw new NullPointerException();
    }

03. Node<K,V> e;
04. V oldValue;
05. int hash = hash(key);
06. e = getNode(hash, key);
07. oldValue = e.value;

08. if (e != null && oldValue != null) {

09.   V v = rf.apply(key, oldValue);

10.   if (v != null) {
11.     e.value = v;
12.     afterNodeAccess(e);
13.     return v;
      }
      else {
14.     removeNode(hash, key, null, false, true);
      }
    }
15. return null;
}
```

Exercise 10. Draw the control-flow graph (CFG) of the method above.



The L\ in the blocks represent the line number corresponding to the blocks.

Exercise 11. How many tests do we need **at least** to achieve 100% line coverage?

Answer: 3.

One test to cover lines 1 and 2. Another test to cover lines 1, 3-7 and 8-13. Finally another test to cover lines 14 and 15. This test will also automatically cover lines 1, 3-10.

Exercise 12. How many tests do we need **at least** to achieve 100% branch coverage?

Answer: 4.

From the CFG we can see that there are 6 branches. We need at least one test to cover the true branch from the decision in line 1. Then with another test we can cover false from L1 and false from L8. We add another test to cover false from the decision in line 10. Finally an additional test is needed to cover the true branch out of the decision in line 10. This gives us a minimum of 4 tests.

Exercise 13. Which of the following statements concerning the subsumption relations between test adequacy criteria **is true**:

1. MC/DC subsumes statement coverage.
2. Statement coverage subsumes branch coverage.
3. Branch coverage subsumes path coverage.
4. Basic condition coverage subsumes branch coverage.

MC/DC does subsume statement coverage. Basic condition coverage does not subsume branch coverage; full condition coverage does.

Exercise 14. A test suite satisfies the loop boundary adequacy criterion if for every loop L:

1. Test cases iterate L zero times, once, and more than once.
2. Test cases iterate L once and more than once.
3. Test cases iterate L zero times and one time.
4. Test cases iterate L zero times, once, more than once, and N, where N is the maximum number of iterations.

Option 1 is correct.

Exercise 15. Consider the expression `((A and B) or C)`. Devise a test suite that achieves 100% *Modified Condition / Decision Coverage* (MC/DC).

Consider the following table:

Decision	A	B	C	(A & B) | C
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	T
8	F	F	F	F

Test pairs for `A = {(2,6)}`, `B = {(2,4)}` and `C = {(3, 4), (5, 6), (7,8)}`. Thus, from the options, tests 2, 3, 4 and 6 are the only ones that achieve 100% MC/DC. Note that 2, 4, 5, 6 could also be a solution.

Exercise 16. Draw the truth table for expression `A and (A or B)`.
Is it possible to achieve MC/DC coverage for this expression? Why (not)?

What feedback should you give to the developer, that used this expression, about your finding?

The table for the given expression is:

Tests	A	B	Result
1	F	F	F
2	F	T	F
3	T	F	T
4	T	T	T

From this table we can deduce sets of independence pairs for each of the parameters:

- `A`: {(1, 3), (2, 4)}
- `B`: { (empty) }

We can see that there is no independence pair for `B`. Thus, **it is not possible to achieve MC/DC coverage for this expression**.
Since there is no independence pair for `B`, this parameter has no effect on the result. We should recommend the developer to restructure the expression without using `B`, which will make the code easier to maintain.
This example shows that software testers can contribute to the code quality not only by spotting bugs, but also by suggesting changes that result in better maintainability.