Coverage testing

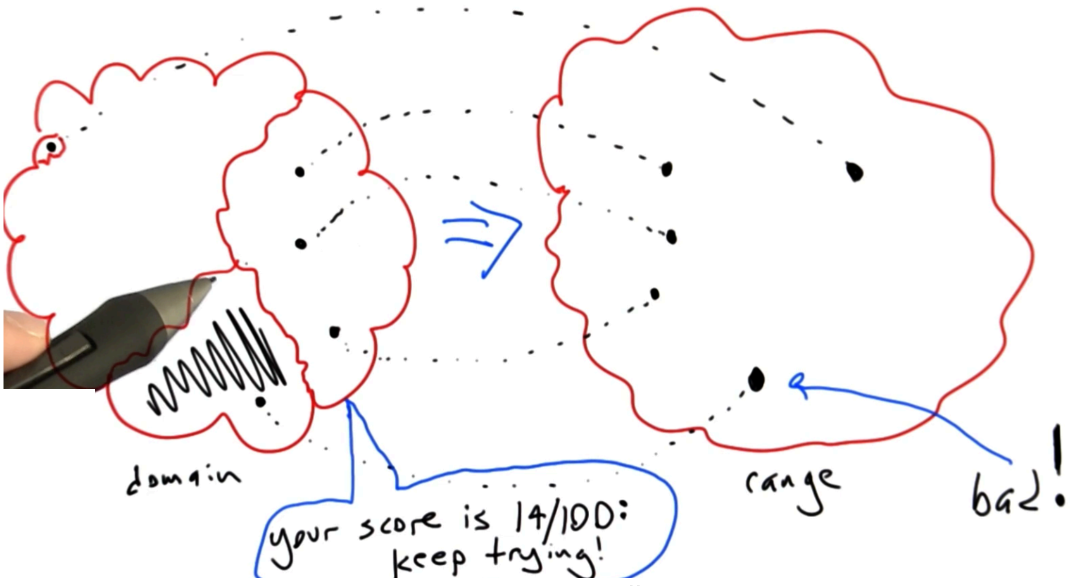
https://www.udacity.com/course/software-testing--cs258

1. Introduction

- Usually problems in released software are coming from things that people forgot to test, i.e. testing was inadequate and the developers were unaware of that fact.
- In the following we present a collection of techniques called **code coverage** where automated tools can tell us places where our testing strategy is not doing a good job.

2. How much testing is enough?

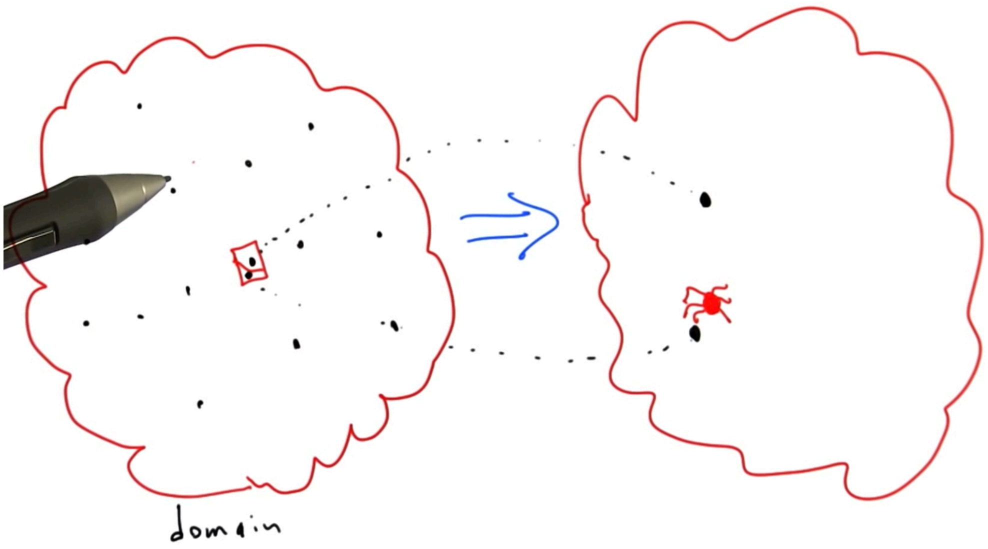
- One of the trickiest things about testing software is that it's hard to know when you've done enough testing and the fact is, it is really easy to spend a lot of time testing and to start to believe that you did a good job and then to have some really nasty bug show up that are triggered by parts of the input space that we just didn't think to test.
- We consider some test cases of the domain, i.e. testing is being compliant to some small part of the input domain and the problem, is that even the small part of the domain may contain an infinite number of test cases.
- Next we consider test cases for other parts of the domain we didn't think to test, putting the results and outputs that are not okay (bad range).
- Assume we have a small Python program that cause the Python runtime to crash.
    - The distinguishing feature of it seem to be a large number of cascaded if statements.
    - It's easy if we're testing to remain in the part of the space where, for example, we have < 5 nested ifs.
    - Another region contains >= 5 nested ifs that cause the Python virtual machine to crash.



- Let's say that we're testing some software that somebody has inserted a back door so that can’t be triggered it accidentally (extremely bad range). We didn't tested inputs triggering the back door because we just didn't know it was there.
- We need some sort of a tool (automated scoring system) or methodology that if we are in fact testing only a small part of the input domain for a system to look at our testing effort and to say that the score is, for example, 14 out of 100. Reasons:
  - Our testing efforts will improve by helping us find the input domain that need more testing.
  - We can argue that we've done enough testing.
  - We can identify parts of the test suite that are completely redundant.

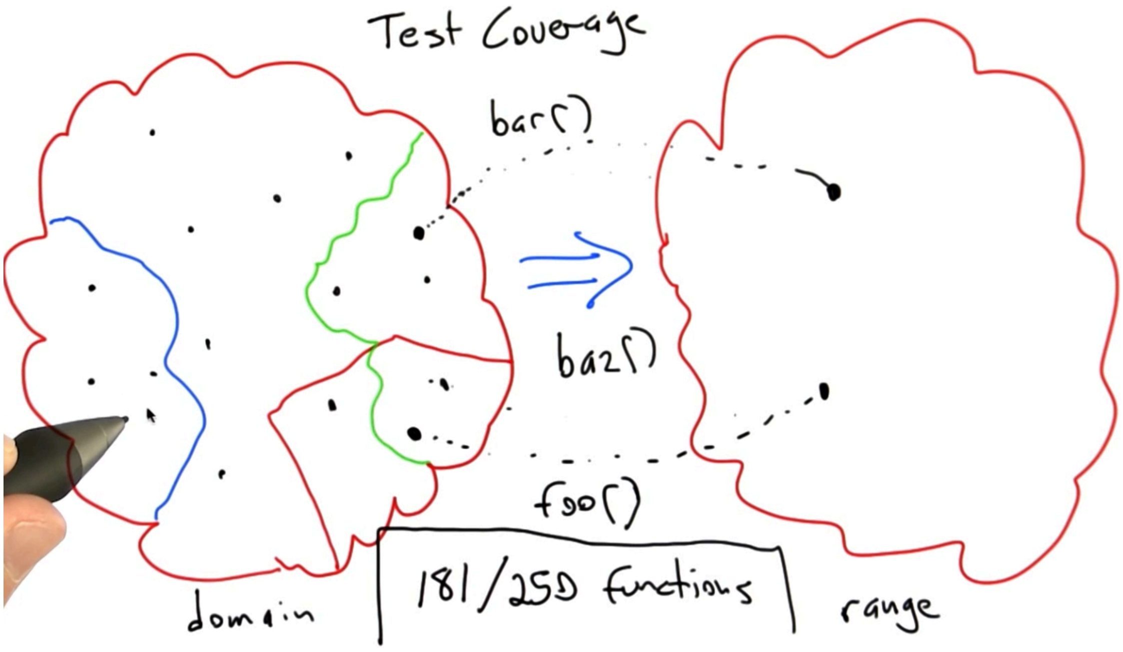
3. Partitioning the input domain

- We start with some SUT and it’s going to have a set of possible inputs, i.e. an input domain, and usually it consists of many possible test cases, that there is no way we can possibly test them all.
- People were often interested in ways to partition the input domain into a no. of different classes so that all of the points within each class are treated the same by the SUT.
- Let’s consider a subset of the input domain. For purposes of finding defects in the SUT, we pick an arbitrary point and execute the system on it. We look at the output, and if it is acceptable, then we’re done testing that class of inputs.
- In practice sometimes what we thought was a class of inputs that are all equivalent might be in different classes. We can blame the partitioning and the unfortunate fact is the original definition of this partitioning scheme didn’t give us good guidance in how to actually do the partitioning.



4. Coverage

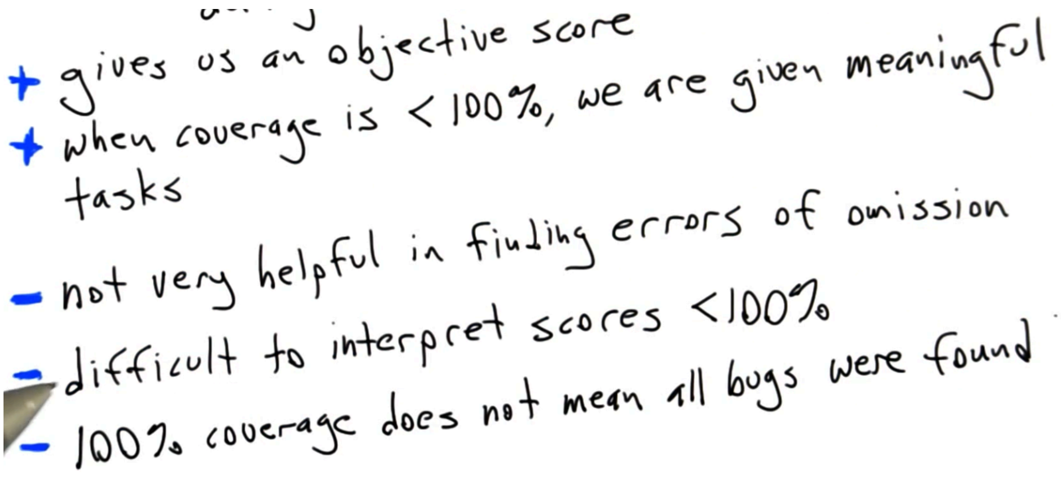
- In practice we ended up with the notion of test coverage instead of a good partitioning of the input domain.
- **Test coverage** is trying to accomplish exact the same thing that partitioning was accomplishing, but it goes about in a different way.
- Test coverage is an **automatic way of partitioning the input domain with some observed features of the source code.**
- One particular kind of test coverage is called **function coverage** and is achieved when every function in our source code is executed during testing.
- We subdivide the input domain for the SUT until we have split it into parts that results in every function being called.



- In practice, we start with a set of test of cases, and we run them all through the SUT. We see which functions are called and then we end up with some sort of a **score** called a **test coverage metric**.
- The score assigned to a collection of test cases is very useful.
  - For each of the functions that wasn't covered, we can go and look at it and we can try to come up with the test input that causes that function to execute.
  - If there is some function baz() for which we can't seem to devise an input that causes it to execute, then there are a couple of possibilities. One possibility is that it can't be called at all. It's **dead code**. Another possibility is that we simply don't understand our system well enough to be able to trigger it.

5. Test coverage

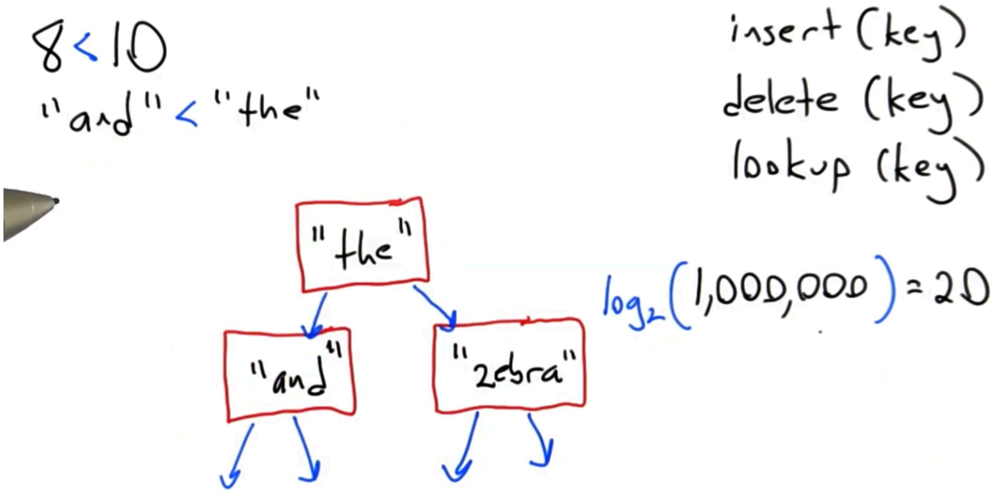
- Test coverage is **a measure of the proportion of a program exercised during testing**.



+ gives us an objective score
+ when coverage is < 100%, we are given meaningful tasks
- not very helpful in finding errors of omission
- difficult to interpret scores <100%
- 100% coverage does not mean all bugs were found

6. Splay tree

- Let's take a concrete look of what a coverage can do for us in practice.
- We'll look at some random open source Python codes that implements a splay tree, a kind of binary search tree.
- A binary search tree is a tree where every node has 2 leaves and it supports operations such as insert, delete, and lookup. The main important thing is the keys have to support an order in relation.
    - If we're using integers for keys then we can use < for order in relation.
    - If we're using words as our keys, then we can use dictionary order.

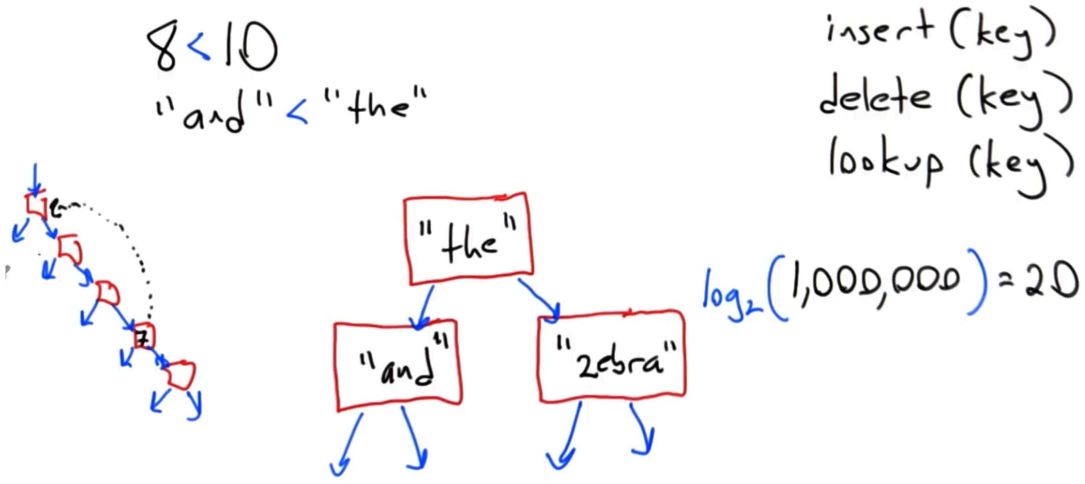


- The way the binary search tree is going to work is, we're going to build up a tree under the invariant that the left child of any node always has a key that's ordered before the key of the parent node and the right child is always ordered after the parent node using the ordering.

- For a large tree with this kind of shape we have a procedure for fast lookup.
- The way that these trees are set up means that, on average, each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree, O(log n).

7. Splay tree issues

- Splay tree is a the simplest example of self-balancing binary search tree. As we add elements a procedure keeps the tree balanced so that tree operations remain very fast.
- It has a really cool property that when we access the nodes, let's say we do a lookup of this node which contains 7, what's going to happen is as a side-effect of the lookup that node is going to get migrated up to the root and then whatever was previously at the root is going to be pushed down and possibly some sort of a balancing operation is going to happen.
  - The point is, that frequently accessed elements end up being pushed towards the root of a tree and therefore, future accesses to these elements become even faster.



- In the following we will look at an open source splay tree implemented in Python, found on the web, that comes with its own test suite and we're going to look at what kind of code coverage this test suite gets on the splay tree.

8. Splay tree example

https://codereview.stackexchange.com/questions/209904/unit-testing-for-splay-tree-in-python

```
class Node:
      def __init__(self, key):
            self.key = key
            self.left = self.right = None

      def equals(self, node):
            return self.key == node.key

class SplayTree:
      def __init__(self):
            self.root = None
            self.header = Node(None)  # for splay()

      def insert(self, key):
            if (self.root == None):
                  self.root = Node(key)
            return

            self.splay(key)
            if self.root.key == key:
                  # If the key is already there in the tree, don't do
anything.
                  return

            n = Node(key)
            if key < self.root.key:
                  n.left = self.root.left
                  n.right = self.root
                  self.root.left = None
            else:
                  n.right = self.root.right
                  n.left = self.root
                  self.root.right = None
            self.root = n

      def remove(self, key):
            self.splay(key)
            if key != self.root.key:
                  raise 'key not found in tree'

            # Now delete the root.
            if self.root.left == None:
                  self.root = self.root.right
            else:
                  x = self.root.right
                  self.root = self.root.left
                  self.splay(key)
                  self.root.right = x
```

```
    def findMin(self):
        if self.root == None:
            return None
        x = self.root
        while x.left != None:
            x = x.left
        self.splay(x.key)
        return x.key

    def findMax(self):
        if self.root == None:
            return None
        x = self.root
        while x.right != None:
            x = x.right
        self.splay(x.key)
        return x.key

    def find(self, key):
        if self.root == None:
            return None
        self.splay(key)
        if self.root.key != key:
            return None
        return self.root.key

    def isEmpty(self):
        return self.root == None
```

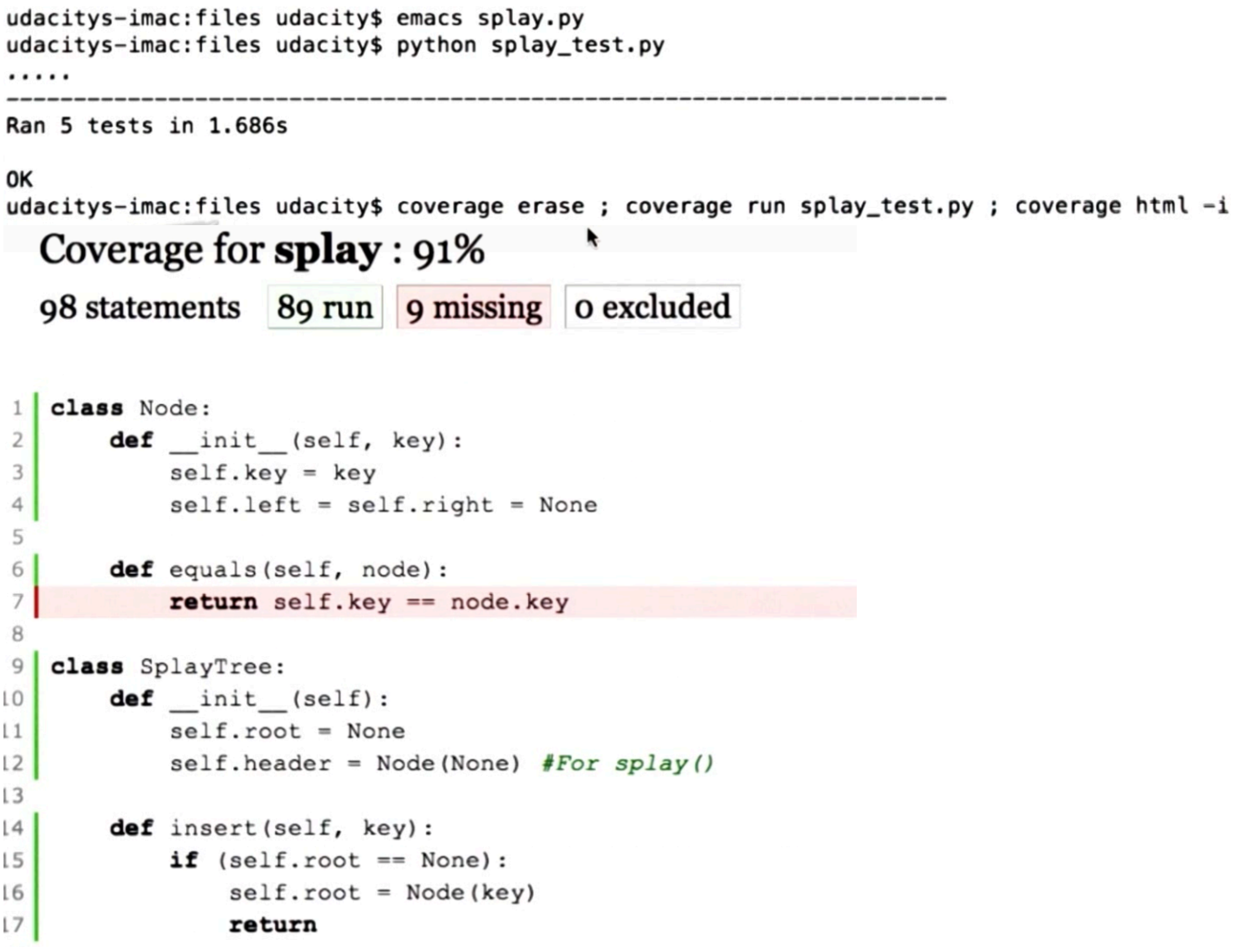
- The splay operation moves a particular key up to the root of the binary search tree. This serves as both the balancing operation and also the look up:

```
    def splay(self, key):
        l = r = self.header
        t = self.root
        self.header.left = self.header.right = None
        while True:
            if key < t.key:
                if t.left == None:
                    break
                if key < t.left.key:
                    y = t.left
                    t.left = y.right
                    r.right = t
                    t = y
                    if t.left == None:
                        break
                r.left = t
                r = t
                t = t.left
            elif key > t.key:
                if t.right == None:
                        break
                if key > t.right.key:
                    y = t.right
                    t.right = y.left
                    r.left = t
                    t = y
                    if t.right == None:
                        break
                l.right = t
                l = t
                t = t.right
            else:
                break
        l.right = t.left
        r.left = t.right
        t.left = self.header.right
        t.right = self.header.left
        self.root = t
```

- Now, let's look at the test suite:

```python
import unittest  # module for unit testing
from splay_tree import SplayTree



class TestCase(unittest.TestCase):
    def setUp(self):
        self.keys = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        self.t = SplayTree()
        for key in self.keys:
            self.t.insert(key)

    def testInsert(self):
        for key in self.keys:
            self.assertEquals(key, self.t.find(key))

    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)

    def testLargeInserts(self):
        t = SplayTree()
        nums = 40000
        gap = 307  # stress testing
        i = gap
        while i != 0:
            t.insert(i)
            i = (i + gap) % nums

    def testIsEmpty(self):
        self.assertFalse(self.t.isEmpty())
        t = SplayTree()
        self.assertTrue(t.isEmpty())

    def testMinMax(self):
        self.assertEquals(self.t.findMin(), 0)
        self.assertEquals(self.t.findMax(), 9)


if __name__ == "__main__":
    unittest.main()
```

- After running the tests we can use a **code coverage tool** to measure coverage. We can generate a HTML report. In our case, it's telling us that when we run the splay tree on its own unit test suite, out of the 98 statements in the file, 89 of them got run, 9 of them failed to run.



```
udacitys-imac:files udacity$ emacs splay.py
udacitys-imac:files udacity$ python splay_test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 1.686s

OK
udacitys-imac:files udacity$ coverage erase ; coverage run splay_test.py ; coverage html -i
```

Coverage for **splay** : 91%

98 statements | 89 run | 9 missing | 0 excluded

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = self.right = None

    def equals(self, node):
        return self.key == node.key

class SplayTree:
    def __init__(self):
        self.root = None
        self.header = Node(None) #For splay()

    def insert(self, key):
        if (self.root == None):
            self.root = Node(key)
            return
```

- The test suite failed to test the case where we inserted an element into the tree and it was already there.
- In the splay tree's removing function the first thing this function does is, splays the tree based on the key to be removed and so this is intended to draw that key up to the root note of the tree. If the root node of the tree does not have the key that we're looking for, then we're going to raise an exception saying that this key wasn't found. But this wasn't tested.
- If we look in the body of the delete function, we see a pretty significant chunk of code that wasn't tested, so we have to go back and revisit this.

- In conclusion, the tool is showing us what we didn't think to test with the unit test suite that we wrote so far.

9. Improving coverage

- We will modify in the test suite, testRemove function:

```
    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)
        self.t.remove(-999)
```

- After running the tool the line that we just added causes an exception re-thrown in the splay function. So we have found a bug not anticipated by the developer of the splay tree.
- It often turns out that the stuff that we thought was going to run might be running only some of it. So the coverage tool told us something interesting.
- On the other hand, if the coverage tool hasn't told us anything interesting, i.e. everything we hoped was executed well when we run the unit test suite then that's good, too.
- Another thing we noticed is that the bug was somewhere completely different buried in the splay routine and if we go back and look at the coverage information, it turns out that the splay routine is entirely covered, i.e. every line of code was executed during the execution of the unit test for the splay tree.
- We deduce that just because some code was covered, especially at the statement level, this does not mean that it doesn't contain a bug in it.
  - We have to ask the question, "What do we want to really read into the fact that we failed to cover something?" The coverage tool has given an example suggesting that our test suite is poorly thought out.
  - So, when coverage fails its better to try to think about why went wrong rather than just blindly writing a test case and just exercise the code which wasn't covered.

10. Problems with coverage

- We looked to an example where measuring coverage was useful in finding a bug in a piece of code. The coverage is not particularly useful in spotting the bug.
- In the following we will discuss about another piece of code, a broken function whose job is to determine whether a number is prime.

```
# CORRECT SPECIFICATION:
#
# isPrime checks if a positive integer is prime.
#
# A positive integer is prime if it is greater than
# 1, and its only divisors are 1 and itself.


import math

def isPrime(number):
    if number<=1 or (number%2)==0:
        return False
    for check in range(3,int(math.sqrt(number))):
        if (number%check) == 0:
            return False
    return True

def test():
    assert isPrime(1) == False
    assert isPrime(2) == False
    assert isPrime(3) == True
    assert isPrime(4) == False
    assert isPrime(5) == True
    assert isPrime(20) == False
    assert isPrime(21) == False
    assert isPrime(22) == False
    assert isPrime(23) == True
    assert isPrime(24) == False
```

- isPrime function has successfully identified whether the input is prime or not for the 10 numbers in the assertions.
- When we run the code coverage tool we can see out of the 20 statements in the file, all of them run, and none of them failed to be covered. Statement coverage gives us a perfect result for this particular code and yet another set is wrong.

```
# TASKS:
#
# 1) Add an assertion to test() that shows
#    isPrime(number) to be incorrect for
#    some input.
#
# 2) Write isPrime2(number) to correctly
#    check if a positive integer is prime.

# Note that there is an error where isPrime() incorrectly returns False for
an input of 2, despite 2 being a prime number.
# This has been fixed in the following.
# In isPrime function the range loops between 3 and the square of the input-1


import math

def isPrime2(number):
    if number == 2:
        return True
    if number<=1 or (number%2)==0:
        return False
    for check in range(3,int(math.sqrt(number)) + 1):
        if (number%check) == 0:
            return False
    return True

def test():
    assert isPrime(6) == False
    assert isPrime(7) == True
    assert isPrime(8) == False
    assert isPrime(9) == True
    assert isPrime(10) == False
    assert isPrime(25) == True
    assert isPrime(26) == False
    assert isPrime(27) == False
    assert isPrime(28) == False
    assert isPrime(29) == True

def test2():
    assert isPrime2(6) == False
    assert isPrime2(7) == True
    assert isPrime2(8) == False
    assert isPrime2(9) == False
    assert isPrime2(10) == False
    assert isPrime2(25) == False
    assert isPrime2(26) == False
    assert isPrime2(27) == False
    assert isPrime2(28) == False
    assert isPrime2(29) == True
```

- Why did test coverage fail to identify the bug? Statement coverage is a rather crude metric that only checks whether each statement executes once. Each statement executes at least once that lets a lot of bugs slip through.

- The lesson here is we should not let complete coverage plus a number of successful test cases fool as into thinking that a piece of code of right. It's often the case that deeper analysis is necessary.

TO BE CONTINUED…