Software Quality Assurance: SOFT3302

Tutorial — Week 5

Objectives

This tutorial gives experience with the design of tests based on white-box testing methods. By the end of this tutorial you should have an understanding of the correspondence between an implementation and a graph representing the implementation, which may then be subject to path analysis; you should also have experience in developing test cases based on the testing paths. You should also have an understanding of coverage metrics and what they represent.

Prework

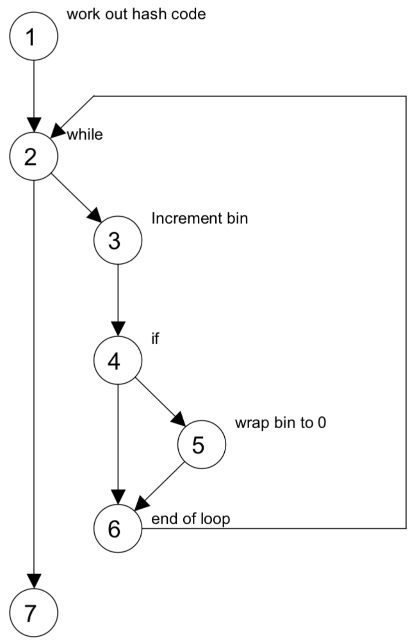
The SUT is the code attached in the appendices that inserts a value into a hash table.

Note: one would almost never write one's own code to do this since it's a prime way to introduce bugs into a programme. Python, C++ and Java all have hashing containers that have been written and debugged already.

I covered this in lectures as 'one way to write good quality code is to not write bad quality code.' Stress the importance of not rewriting library routines. Take the effort to understand what a particular language environment provides, *e.g.* the Python library or the C++ STL, and use it.

Draw a flow graph from the code and label the nodes in preparation for the tutorial.

Something similar to the following. Node node 6 which corresponds to no actual line of code but is useful to include as it makes the graph simpler. Note also that node 3 may not appear in a naïf translation of code that has an increment for bin as part of the *if* although it should be put in.



Okay, not the most sophisticated of examples but should be enough for thought and for some fruitful discussion. Some students may notice the bug in the code. It's deliberate and is asked about later in the tutorial.

Labwork

1. Develop test cases to provide 100% statement coverage.

   Should be fairly simple. A single test of inserting two elements with the same hash value into the table where *the hash value* mod *size of table* == *size of table – 1* will do this (see the 1-path case in (3) below). This requires an understanding of the hash value function and the size of the hash table being used.

2. Develop test cases to provide 100% branch coverage.

   Insert into empty table: exercises 2–7; insert two values into the table not at the end of the table, *i.e.* at *n* and *n+1* where 0 < n < *size of table – 1*: exercises 4–6; insert two values at the end of the table where *n* == *size of table – 1*: exercises 4–5.

3. Develop test cases to provide 0-path and 1-path independent path (basis path) tests of the loop.

   0-path: insert a value into an empty array.

   1-path: insert a value into an empty array when an existing value in the array hashes to the same value. For example, put 105 into a 100-element array of integers which already contains the value 5 when the hash function for integers simply returns the integer.

4. Note that all of the tests so far will not necessarily catch a certain fault with this code. (Can you see what it is?) Develop test cases that use Beizer's loop test criteria to test iterations of the loop.

   The problem is that a full hash table will cause the loop to loop forever. This will only be caught by a maximum iteration test. Code coverage, branch coverage and 0/1-path basis path coverage will not catch this situation.

   This kind of subtle hidden bug is precisely the reason one doesn't write one's own versions of these sorts of containers. Stress this point.

Test cases should include the following information at a minimum.

Module/Component:				
Tested by:				
Version:				
Testing date:				
Test Id	Path	Description	Inputs	Outcome

Appendix

The following code samples purport to (although may not) implement the same algorithm to insert a value into a hash table. Hash collisions are handled by a linear search through the table for the next empty bin.

To repeat the admonition above: one almost never needs to implement one's own version of a hash table in the languages below. Use the supplied hashed containers in the accompanying libraries. Even if one were to implement a hash table, choosing linear search on collision is most likely a bad choice.

In the three languages below, only one is not amenable to resizing the array within the insert if it is deemed that the hash table is too densely populated. Which? Why?

C++ supports reference arguments and vectors can be resized so the vector argument can be resized. Even if vectors could not be resized because the argument is a reference argument it may be changed.

Python passes reference arguments by value but a list may be dynamically resized (hashTable = [None] * 200 would not work but hashTable[:] = [None] * 200 would).

Java passes reference arguments by value but arrays are not dynamically resizable. This possibly means that *ArrayList* or *Vector* might have been a better choice of type. (Personally, I find the whole necessity for Integer, Character, Boolean, *&c.* wrapper types in Java and the endless casting to and from Object when generics are not available enormously obscuring of the intent of a piece of code. What sort of language claims not to have pointers and then raises a *NullPointerException*?? ☺)

Python

```
def insert(hashTable, v):
        bin = hash(v) % len(hashTable)
        while hashTable[bin] is not None:
                bin += 1
                if bin == len(hashTable):
                        bin = 0
        hashTable[bin] = v
```

C++

```
template <typename T>
void
insert(vector<T *>& hashTable, T *v)
{
        // int hash(const T& v) is a template function defined elsewhere
        int bin = hash(*v) % hashTable.size();
        while (hashTable[bin])
                if (++bin == hashTable.size())
                        bin = 0;
        hashTable[bin] = v;
}
```

Java

```
void
insert(Object[] hashTable, Object v)
{
        int bin = v.hashCode() % hashTable.length;
        while (bashTable[bin] != null)
                bin += 1;
                if (bin == hashTable.length)
                        bin = 0;
        hashTable[bin] = v;
}
```