Ciências ULisboa | Informática

VVS 2017/2018
Exam 1 – June 7, 2018
Duration: 3 hours

Consider the following class excerpt implementing an hash table using linear probing (i.e., check next position) for finding the next available slot if a collision of keys occurs. If a key is deleted, its position becomes vacant and so it is marked with a reference to the special marker DELETED so that it can be reused in a future insertion.

```
1  public class HashtableOpen<K, V> {
2     // Entry is a pair (key, value)
3     private Entry<K, V>[] table; // empty slots == null
4     private int numKeys;     // num. of keys
5     private int numDeletes; // num. of vacant entries
6
7     private final double LOAD_THRESHOLD = 0.5;
8     // marker for a deleted/vacant entry
9     private final Entry<K, V> DELETED = ...
10
11    private int hash(K key) {
12      return Math.abs(key.hashCode()) % table.length;
13    }
14    ...
15    //@ requires key != null && value!= null;
16    public void put(K key, V value) {
17      int index = hash(key);  // get starting index
18      int firstDeleted = -1;
19      // increment index until empty slot is reached
20      // or key is found.
21      while (table[index] != null &&
22             !key.equals(table[index].key)){
23        if (firstDeleted ==-1 && table[index] == DELETED)
24           firstDeleted = index;
25        index = (index + 1) % table.length;
26      }
27      if (firstDeleted != -1) {
28         // if an vancant slot was found, insert new entry
29         table[firstDeleted] = new Entry<>(key, value);
30         numKeys++;
31         numDeletes--;
32      } else if (table[index] == null) {
33         // if an empty slot was found, insert new entry
34         table[index] = new Entry<>(key, value);
35         numKeys++;
36         // check whether rehash is needed.
37         double loadFactor =
38            (double)(numKeys+numDeletes)/table.length;
39         if (loadFactor > LOAD_THRESHOLD)
40           rehash(); // increase table[] size & reinsert
41      } else  // key exists, update value for this key
42           table[index].value = value;
43    }
```

Group 1

1. Draw the control flow graph (CFG) of method `put()`. Assume the calls of `hash()` and `rehash()` as simple commands.
2. How may requirements are produced by edge coverage? List six of them.
3. Write a set of JUnit tests that satisfy those six requirements. For each test, write which requirements it satisfies. Consider tests where keys are integers (their hash code is their own value) and the vector table has size 100. Assume methods `remove(K key)`, `size()`, `equals()`, and `clone()` are correct and can be used.
4. Identify the def-sets and use-sets for all nodes of the CFG.
5. List the def-path sets for every node regarding variable `firstDeleted`. Do not need to include def-paths that are strict sub-paths of others.

Group 2

1. In the context of *Input Space Partitioning* identify the input parameters of the SUT.
2. Model the input domain by identifying three relevant characteristics and present their respective blocks.
3. Present the requirements generated by using All Combinations Coverage (ACoC). Underline those infeasible.
4. If ACoC produced too many requirements, what would be your criteria choice? Present the requirements generated by that criteria.

Group 3

1. Using *Predicate Coverage*, list its requirements.
2. Build a table with the reachability logic formulas for each predicate.
3. Write JUnit tests to test these requirements. If there are infeasible requirements, justify why is it so. Use the same considerations and assumptions as in the previous JUnit exercise.
4. Describe the *General Active Clause Coverage* criteria.

Group 4

Consider that you are working with two mutation operators: a math mutator that switches arithmetic operators, and a logic mutator switching equalities and inequalities or &&'s and ||'s. Assume that tests can use method `equals()` that compares if two hash table have the same set of pairs (key, value) and the same load factor.

1. Present four mutations, two for each mutator, from lines 21–22, 25, 32 and 37–38.
2. Give an example of one of these mutations not being killed because some test was missing.
3. Write JUnit test(s) to kill the logic mutation at line 27. Use the same considerations and assumptions as in the previous JUnit exercise.
4. Provide an example where some test infects some mutation but it does not result on a failure.
5. Describe the *Bomb Strategy* and why it is useful for node coverage.