Project 1 – Group 1

Software Verification and Validation

Pedro Carrega – fc49480
Rodrigo Lourenço – fc47864
Vasco Ferreira – fc49470

Ciências ULisboa
Faculdade de Ciências da Universidade de Lisboa

Instruction Coverage

In this coverage criteria we need to cover all instructions in each public method, to start this coverage we defined each instruction done in each public method, and applying test cases that would cover them, in most of the methods it is not possible to achieve 100% coverage of the method with only one test, so it had to be covered distributed by methods, to confirm the 100% instruction coverage of the public methods we used the eclipse plug in recommended by the professor, has it is possible to verify in the following figure:

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
VVS_ASSIGNMENT1	39,4 %	943	1.448	2.391
src/test/java	26,2 %	436	1.230	1.666
src/main/java	69,9 %	507	218	725
startup	0,0 %	0	123	123
sut	84,2 %	507	95	602
TST.java	84,2 %	507	95	602
TST<T>	84,2 %	507	95	602
collect(Node<T>, StringBuilde	3,2 %	3	90	93
get(Node<T>, String, int)	90,9 %	50	5	55
collect(Node<T>, StringBuilde	100,0 %	48	0	48
contains(String)	100,0 %	15	0	15
delete(Node<T>, String, int)	100,0 %	56	0	56
delete(String)	100,0 %	11	0	11
equals(Object)	100,0 %	67	0	67
get(String)	100,0 %	29	0	29
keys()	100,0 %	14	0	14
keysThatMatch(String)	100,0 %	16	0	16
keysWithPrefix(String)	100,0 %	40	0	40
longestPrefixOf(String)	100,0 %	60	0	60
put(Node<T>, String, T, int)	100,0 %	65	0	65
put(String, T)	100,0 %	27	0	27
size()	100,0 %	3	0	3

Figure 1 100% Instruction Coverage in Public Methods

Here we can see that the public methods (the green circles) achieve 100% coverage, completing the task asked.

Graph Based Coverage

In order to use Graph based coverage it is needed to implement the graph that describes the system under test, in this case it was longestPrefixOf with the following graph:

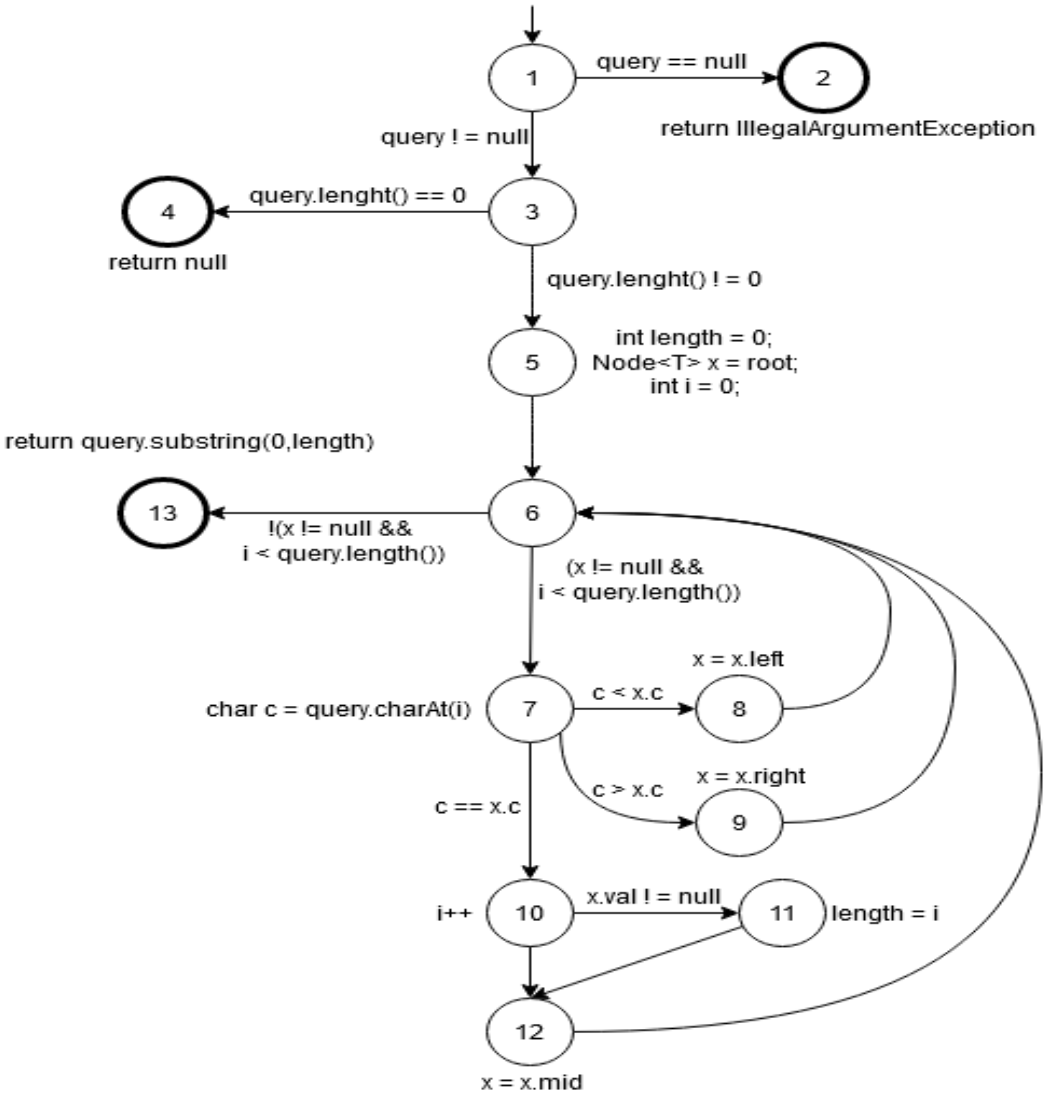

Figure 2 Graph longestPrefixOf

With this graph we could now develop the graph-based coverage that was asked.

Edge Coverage:

The purpose of this test criteria is that edges are covered, edges being every path up to length one, so all nodes and all possible path between every connected node.

All test requirements and coverage of them is presented as comments in the test class EdgeCoverageTests, describing which test requirements each test covers.

Prime Path Coverage:

In order to understand prime path coverage, it is needed to understand the following definition:

Simple Path: A path is a simple path if no node appears more than once, other than possibly the first and last ones, simple paths can't have any internal loops but may represent a loop if the first and last nodes are equal.

With this definition, we can say that a prime path is a maximal length simple path, that is, a simple path that is not a proper sub path of any other simple path. The Prime Path Coverage requires that every prime path be covered by the test set.

This criteria resulted in the following test requirements:

TR(PPC) = {[1,3,5,6,7,10,11,12], [1,3,5,6,7,10,12], [6,7,10,11,12,6], [7,10,11,12,6,13], [1,3,5,6,7,8], [1,3,5,6,7,9], [7,10,11,12,6,7], [8,6,7,10,11,12], [9,6,7,10,11,12], [12,6,7,10,11,12], [11,12,6,7,10,11], [10,11,12,6,7,8], [10,11,12,6,7,9], [10,11,12,6,7,10], [6,7,10,12,6], [7,10,12,6,13], [7,10,12,6,7], [1,3,5,6,13], [8,6,7,10,12], [9,6,7,10,12], [12,6,7,10,12], [10,12,6,7,9], [10,12,6,7,8], [10,12,6,7,10], [7,8,6,13], [7,8,6,7], [7,9,6,7], [7,9,6,13], [6,7,9,6], [6,7,8,6], [9,6,7,9], [9,6,7,8], [8,6,7,9], [8,6,7,8], [1,3,4], [1,2]}

The coverage of the test cases is presented in the the PrimePathCoverage class, with a respective comment header with the tests requirement paths covered.

All Uses Coverage:

To understand the test requirements resulted from the coverage criteria, first it is important to understand the following concepts:

Definition: A definition of a variable (i.e. int x = 0)
Uses: The use of a variable (i.e. if (x == 0))

The test requirements are obtained where each definition reaches all possible uses, but there is no need to repeat paths, that is, if we have [5,6,10] and [5,6,7,8,9,10], there is no need to include both in the test requirements, it is only needed one path per def-use pair.

The following table represents the test requirements obtained:

v	**n**	**du(n,v)**
length	5	[5,6,13]
	11	[11,12,6,13]
root	5	[5,6,13][5,6,7][5,6,7,8][5,6,7,9][5,6,7,10][5,6,7,10,11][5,6,7,10,12]
	8	[8,6,13][8,6,7][8,6,7,8][8,6,7,9][8,6,7,10][8,6,7,10,11][8,6,7,10,12]
	9	[9,6,13][9,6,7][9,6,7,8][9,6,7,9][9,6,7,10][9,6,7,10,11][9,6,7,10,12]
	12	[12,6,13][12,6,7][12,6,7,8][12,6,7,9][12,6,7,10][12,6,7,10,11][12,6,7,10,12]
i	5	[5,6,13][5,6,7][5,6,7,10][5,6,7,10,11]
	10	[10,11][10,12,6,13][10,12,6,7][10,12,6,7,10]
c	7	[7,8][7,9][7,10]
query	1	[1,2][1,3][1,3,4][1,3,5][1,3,5,6,7][1,3,5,6,13]

Table 1 All Uses Test Requirements

This following table presents the test path followed by each test case used to cover the test requirements:

test	test path
NullQuery	[1,2]
EmptyQuery	[1,3,4]
EmptyTST	[1,3,5,6,13]
Valid1	[1,3,5,6,7,8,6,7,9,6,7,10,11,12,6,13]
Valid2	[1,3,5,6,7,10,12,6,7,10,12,6,7,10,12,6,7,9,6,13]
Valid3	[1,3,5,6,7,9,6,13]
Valid4	[1,3,5,6,7,10,11,12,6,7,8,6,7,10,11,12,6,7,9,6,7,9,6,7,10,12,6,7,8,6,7,8,6,13]
Valid3	[1,3,5,6,7,9,6,7,8,6,7,10,12,6,7,10,11,12,6,7,10,12,6,7,10,12,6,13]

Table 2 All Uses Test Cases

With these two tables it is possible to verify that the subset of test done, cover all the test requirements of the All-Uses Coverage Criteria.

Logic Based Coverage

In this coverage we used the general active clause coverage (GACC) to apply to the longestPrefixOf method, this was mainly due to the double clause predicate in the while condition, lets see the reason against the other test criteria's:

CC: It was chosen GACC instead of CC because CC only evaluates clauses independently so in p3 we can have a case where the 2nd clause does not matter since the 1st clause is false.
PC: PC was also not chosen because of taking only into consideration the predicate itself, again in p3 it does not cover some important cases that GACC covers. Since it only requires that p3 is true and another that p3 is false.
CACC & RACC: The GACC was chosen instead of CACC or RACC because in the while predicate the value will have no effect in the sense that when the clause is deterministic, and its value is true then the condition will be true, when the value is false then the predicate will be false also, so this would not have any effect on the while predicate so that was why the GACC was chosen. The other predicates are only composed by one clause so the GACC would be the correct for them too.

The test requirements resulted from the criteria chosen are presented in detail in the GeneralActiveClauseCoverageTests class, followed by the tests itself, which have a header that details the coverage of each test in relation to the test requirements.

Input State Partitioning

It was asked to apply Base Choice Coverage(BCC) Criteria to fulfil the input state partitioning, this time applying it to the put public method, it was also given to us the characteristics to be used for the test, with this we concluded that each characteristic had the following block [true,false], so by definition of the BCC test requirement are defined stating with a base choice, which was [true,true,true] where:

Array Position 0: Trie already includes the new key
Array Position 1: Trie already includes some new key prefix
Array Position 1: Trie is empty

Now the definition of BCC states that starting from the base choice, we derive a second test requirement by changing only value at a time, with this we obtained the following 4 test requirements:

TR1: [true,true,true]
TR2: [false,true,true]
TR3: [false,false,true]
TR4: [false,false,false]

When evaluating the test requirements, it was noted that TR1 is infeasible since we can not have an empty tree where the Trie already includes the new key. With this in consideration it was implemented the 3 remaining test requirements in the BaseChoiceCoverageTest class with the respective comment headers detailing the test requirement covered by the test.

PIT Mutation Tests

Instruction Coverage: This criteria kills 100% of the mutations done in the SUT, this was achieved with only 3 test cases, which shows a great efficiency of the criteria in question.

```
135         public String longestPrefixOf(String query) {
136 1               if (query == null)
137                         throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
138 1               if (query.length() == 0)
139 1                       return null;
140                 int length = 0;
141                 Node<T> x = root;
142                 int i = 0;
143 3               while (x != null && i < query.length()) {
144                         char c = query.charAt(i);
145 2                       if (c < x.c)
146                                 x = x.left;
147 2                       else if (c > x.c)
148                                 x = x.right;
149                         else {
150 1                               i++;
151 1                               if (x.val != null)
152                                         length = i;
153                                 x = x.mid;
154                         }
155                 }
156 1               return query.substring(0, length);
157         }
```

Figure 3 Instruction Coverage Mutation Coverage

Edge Coverage: This criteria could not kill two mutations that were made, those mutation where the negation of the conditions in lines 145 and 147. With this we can verify that even though the criteria coverage was a completed, that does not mean the criteria is the correct for the SUT.

```
135         public String longestPrefixOf(String query) {
136 1               if (query == null)
137                         throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
138 1               if (query.length() == 0)
139 1                       return null;
140               int length = 0;
141               Node<T> x = root;
142               int i = 0;
143 3               while (x != null && i < query.length()) {
144                       char c = query.charAt(i);
145 2                     if (c < x.c)
146                               x = x.left;
147 2                     else if (c > x.c)
148                               x = x.right;
149                       else {
150 1                             i++;
151 1                             if (x.val != null)
152                                       length = i;
153                               x = x.mid;
154                       }
155               }
156 1               return query.substring(0, length);
157         }
```

Figure 4 Edge Coverage Mutation Coverage

Prime Path Coverage: In this test criteria we can check that it can kill all the generated mutations, what was noted was that this was achieved with a huge number of test requirements covered.

```
136 1               if (query == null)
137                         throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
138 1               if (query.length() == 0)
139 1                       return null;
140               int length = 0;
141               Node<T> x = root;
142               int i = 0;
143 3               while (x != null && i < query.length()) {
144                       char c = query.charAt(i);
145 2                     if (c < x.c)
146                               x = x.left;
147 2                     else if (c > x.c)
148                               x = x.right;
149                       else {
150 1                             i++;
151 1                             if (x.val != null)
152                                       length = i;
153                               x = x.mid;
154                       }
155               }
156 1               return query.substring(0, length);
157         }
```

Figure 5 Prime Path Mutation Coverage

All Uses Coverage: This criteria was able to kill all mutants that were generated for the SUT. However, this was achieved with the highest number of tests relatively to for example the Instruction Criteria that obtained the same 13 mutants killed. This does not mean that this test coverage criteria are worse but does show that sometimes a test criteria could generate more test requirements but for the SUT in question be less useful than a simpler test coverage.

```
135         public String longestPrefixOf(String query) {
136 1               if (query == null)
137                         throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
138 1               if (query.length() == 0)
139 1                       return null;
140               int length = 0;
141               Node<T> x = root;
142               int i = 0;
143 3               while (x != null && i < query.length()) {
144                       char c = query.charAt(i);
145 2                     if (c < x.c)
146                               x = x.left;
147 2                     else if (c > x.c)
148                               x = x.right;
149                       else {
150 1                             i++;
151 1                             if (x.val != null)
152                                       length = i;
153                               x = x.mid;
154                       }
155               }
156 1               return query.substring(0, length);
157         }
```

Figure 6 All Uses Mutation Coverage

General Active Clause Coverage: In this logic based coverage chosen by the group can check that even though it focus on the Boolean clauses, it was not capable of killing the mutant in line 147 which negates the conditional clause. This was verified even with the test cases covering all test requirements produced by the method. This could be a reason to rethink of the logic-based coverage criteria chosen, another option would be to include a test that would kill the remaining mutant that was left alive.

```
136 1               if (query == null)
137                         throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
138 1               if (query.length() == 0)
139 1                       return null;
140               int length = 0;
141               Node<T> x = root;
142               int i = 0;
143 3               while (x != null && i < query.length()) {
144                       char c = query.charAt(i);
145 2                     if (c < x.c)
146                               x = x.left;
147 2                     else if (c > x.c)
148                               x = x.right;
149                       else {
150 1                             i++;
151 1                             if (x.val != null)
152                                       length = i;
153                               x = x.mid;
154                       }
155               }
156 1               return query.substring(0, length);
157         }
```

Figure 7 General Active Clause Mutation Coverage

Trie Random Generator:

To achieve the random generator, it was used Junit quick generator, extending it and applying it to generate random Tries so that they are used for the following tests. The Trie generated will have and random number of elements, where the key will have a length of at least 2 characters and the value will also be random with its maximum possible value of 100.

With the generated generator we followed with the development of the equals and delete methods. The methods can be seen implemented at the TST class.

After the implementation we applied four properties to test:

1. The order of insertion of different keys does not change the final tree value
    a. To test this property, the test receives as input a random TST and copies its content to a new TST, but by a random order. At the end of this process we will end up having a trie with the same pair key-value but with a different structure. This fact should not interfere with the equality of the Tries. If the equal method returns true, then the property is verified.
2. If you remove all keys from a tree, the tree must be empty
    a. In order to test this property, we delete all pairs from the generated tree and test if the size of the tree is zero.
3. Given a tree, inserting and then removing the same key value will not change its initial value
    a. To verify this property the test inserts a random pair key-value to a copy of the input trie and then proceeds to its deletion, after these two operations the trie should be equal to the trie received at the input.
4. Selecting a stricter prefix keysWithPrefix returns a strict subset result.
    a. This is the more complex property to be tested, as usual with the last properties, the test receives a random trie from the generator, then it selects a random key present in the trie and searches the prefixes present in the trie. After this it will take the last character of the given key and get its prefixes. Now to test the property, we then verify if the first list of prefixes is smaller then the last one, since if the key is larger, the result list of its prefixes will be smaller, this test will be repeated until the key has size of 1 where it cannot substring anymore due to the trimmed key resulting in an empty string. At the end if this property was not verified in any of the sub tests, it will result in a failed test.

The verification of the properties can be seen at the PropertyGeneratorTest class.

It is important to note that the equals and delete methods where covered in the InstructionCoverageTests class since it was asked to cover all public methods in the TST class. The coverage can be seen in Figure 1, that shows that both methods have sufficient tests to cover 100% of their instructions.