

**Faculdade de Ciências da Universidade de Lisboa**

**Assignment 1 – Ternary Search Tree Implementation Validation Report**

**Verificação e Validação de Software**

**Miguel Ferreira, nº 57481**

**Mestrado em Engenharia Informática**

**abril de 2022**

**Table of Contents**

[Introduction 4](#_Toc102058285)

[Line and Branch Coverage 4](#_Toc102058286)

[size() Method 4](#_Toc102058287)

[contains(String key) Method 4](#_Toc102058288)

[get(String key) Method 4](#_Toc102058289)

[put(String key, T val) 5](#_Toc102058290)

[longestPrefixOf(String Key) Method 5](#_Toc102058291)

[keys() Method 5](#_Toc102058292)

[keyWithPrefix(String prefix) Method 5](#_Toc102058293)

[keysThatMatch(String pattern) Method 6](#_Toc102058294)

[Edge-pair Coverage and Prime Path Coverage of longestPrefixOf Method 7](#_Toc102058295)

[Edge-pair Coverage 7](#_Toc102058296)

[Edge-Pair Coverage Test #1 8](#_Toc102058297)

[Edge-Pair Coverage Test #2 8](#_Toc102058298)

[Edge-Pair Coverage Test #3 9](#_Toc102058299)

[Edge-Pair Coverage Test #4 9](#_Toc102058300)

[Edge-Pair Coverage #5 9](#_Toc102058301)

[Edge-Pair Coverage #6 9](#_Toc102058302)

[Prime Path Coverage 10](#_Toc102058303)

[Prime path Coverage Tests #1, #2, #4 and #12 11](#_Toc102058304)

[Prime path Coverage #3 12](#_Toc102058305)

[Prime path Coverage #5 12](#_Toc102058306)

[Prime path Coverage #6 12](#_Toc102058307)

[Prime path Coverage #7 13](#_Toc102058308)

[Prime path Coverage #8 13](#_Toc102058309)

[Prime path Coverage #9 13](#_Toc102058310)

[Prime path Coverage #10 13](#_Toc102058311)

[Prime path Coverage #11 14](#_Toc102058312)

[Prime path Coverage #12 14](#_Toc102058313)

[Prime path Coverage #13 14](#_Toc102058314)

[Prime path Coverage #14 15](#_Toc102058315)

[Prime path Coverage #15 15](#_Toc102058316)

[Prime path Coverage #16 15](#_Toc102058317)

[Prime path Coverage #17 16](#_Toc102058318)

[Prime path Coverage #18 16](#_Toc102058319)

[Prime path Coverage #19 16](#_Toc102058320)

[All-Du-Paths Coverage 17](#_Toc102058321)

[All DU Path Coverage Tests #1 - #6, #10 - #36 19](#_Toc102058322)

[All DU Path Coverage Tests #7 19](#_Toc102058323)

[All DU Path Coverage Tests #8 19](#_Toc102058324)

[All DU Path Coverage Tests #9 19](#_Toc102058325)

[All-Coupling-Use-Paths Coverage 20](#_Toc102058326)

[All-Coupling-Use-Paths Test #1 20](#_Toc102058327)

# Introduction

The purpose of this first assignment is to test – execute verifications and validations – of a simple implementation of a Trie data structure, a tree for fast insertion and retrieval of key-value pairs using the JAVA programming language and the Eclipse IDE, alongside with some tools with the intent of making the testing task easier.

This report is divided in various sections that are listed in the table of contents above. In each of these sections, I explain the objective and what *unit tests* were implemented as well as a brief description, which includes the reasoning and logic behind the implementation.

# Line and Branch Coverage

In this section, the main objective was to measure how many statements were executed as well as check if each conditional (if, while, for) was executed having in account the true and false possible outcome branches (testing the guards of the conditional statements).

Below are all the requested (public) methods:

## size() Method

This simple method returns the number of key-value pairs in the symbol table.

The implementation of the unit test was trivial: a key-value pair was inserted in the Trie through the put(String key, T val) method and the SUT should return one.

## contains(String key) Method

This method checks whether the symbol table contains (or not) a given key. Two unit tests we implemented:

1. A method which tests the return an *IllegalArgumentException* if a null argument is passed to it.
2. A method in which a key-value pair is inserted through the put(String key, T val) method. The result of the latter will be true if the key passed as an argument to the SUT exists in the symbol table.

## get(String key) Method

This method returns a generic value corresponding to the key that is passed as an argument to the method. It is worth noting that this “T” can be of any type: in my implementation, I chose Integer.

Four tests were implemented:

1. A method which tests the return of an *IllegalArgumentException* if a null argument is passed to it.
2. A method that also tests the return of an *IllegalArgumentException* if an empty string is passed to it.
3. A method that tests the return of a null value by the SUT if the symbol table contains no key-value pairs but the SUT is executed with a random key.
4. A method that checks the correct and expected functionality of the method with valid symbol table and valid input.

## put(String key, T val)

As already mentioned before, this method inserts a new key-value pair in the symbol table according to the value of each character of the given key, allowing the construction of the ternary search tree.

Two unit tests were implemented:

1. A method which tests the return of an *IllegalArgumentException* if a null key argument is passed to it.
2. A method which tests the correct key-value pair inserted by making use of the get(String key) method – if the key-value pair was indeed inserted and the correct key is used as argument, the get(String key) method will return the correct value.

## longestPrefixOf(String Key) Method

This method returns the value in the symbol table which contains the longest prefix of the query passed as an argument to the method or null if it doesn’t exist.

Five unit tests were implemented:

1. A method which tests the return of an *IllegalArgumentException* if a null query argument is passed to it.
2. A method which tests the return of an *IllegalArgumentException* if a zero length query argument is passed to it.
3. A method which test the correct execution and return of the method: four key-value pairs were inserted to compose the Trie, all with similar prefixes and one which had the prefix itself. The test used as input a string which contained the prefix already inserted, proving the return of the desired output.
4. A method which tests the return of an empty string when the root of the Trie is null.
5. A final and trickiest test. In this one, I check a condition inside the while in which the first clause is true and the second is clause if false. This means that the variable “x” (the current node in the iteration) in the method is not null but the variable “i” was previously incremented, making the second false (because it becomes equal to the query ‘s length, not less) and thus, since it is an “and” conditional, the cycle won’t execute again.

## keys() Method

This method has the purpose of returning all keys in the symbol table.

One unit test was implemented: A key-value pair was inserted through the put(String key, T val) method. An *Iterable<String>* object was created (a *Queue).* Then it was made a comparison between the SUT and the locally created *Queue,* in which the unit test would check if these two were equal (and thus confirming that the method would return they desired key).

## keyWithPrefix(String prefix) Method

This method returns all keys in a subtrie according to a prefix.

Four unit tests were implemented:

1. A method which tests the return of an *IllegalArgumentException* if a null query argument is passed to it.
2. A method in which an empty trie is created and and empty *Queue* is returned by the SUT (since there are no prefixes to look for).
3. A method in which the correct execution of the method is tested: a key-value pair was inserted through the method put(String key, T val) and a *Queue* was locally created. Both of these latter had the same key inserted. Then, the SUT was tested by using a prefix as argument that would return the key previously inserted.
4. This test is the same as the one above, what differs is the fact that the inserted key is the prefix itself.

## keysThatMatch(String pattern) Method

This method returns the keys in the symbol table that match a certain pattern. This patterns can have dots (“.”) that symbolize wildcards.

One unit test was implemented: through the method put(String key, T val) a key-value pair was inserted. Then a local *Queue* was created and a string was added in order to compare the result from the SUT. The SUT had a pattern that matched the string added to the aforementioned *Queue.* This means that the correct key would be returned by the SUT, since the method would recognize the pattern and the key inserted when compared to the locally created *Queue*.

# Edge-pair Coverage and Prime Path Coverage of longestPrefixOf Method

In this section, the objective is to analyze two graph-based test strategies: edge-pair coverage, in which all paths up to length of 2 are covered and Prime path coverage, that consists in a maximal length path in which no node appears more than once (except for the first node and the last node).

This coverage criterion is analyzed in the context of the longestPrefixOf method.

In the figure depicted below, it is possible to observe the graph of the method aforementioned:

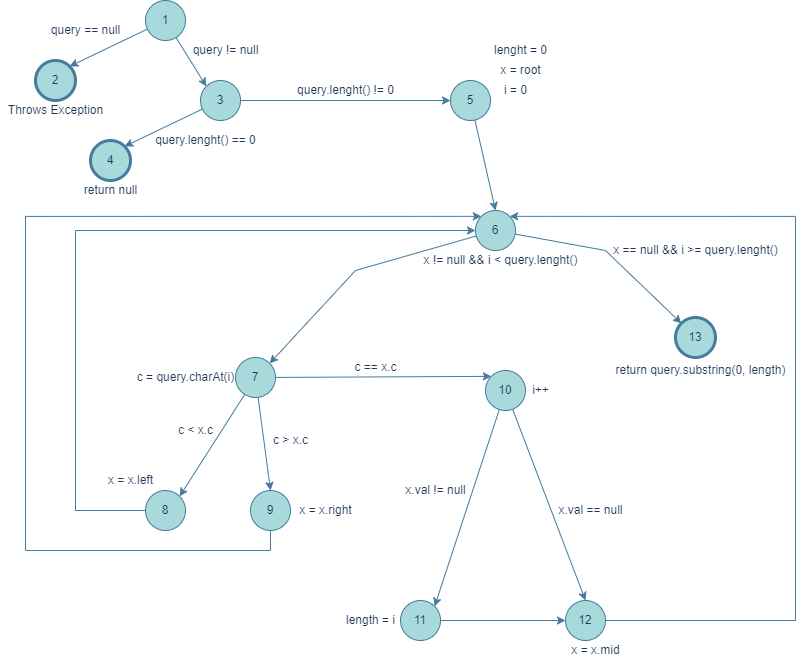


Figure 1 - longestPrefixOf graph

## Edge-pair Coverage

According to the figure above, a total of 6 unit tests are needed to correctly implement the edge-pair coverage criteria. The paths traversed by this tests can be observed in the table below (and also in the implemented code):

|  |  |  |
| --- | --- | --- |
| Test # | Test Paths | Test requirements toured by the paths |
| Test #1 | [1,3,5,6,7,9,6,7,10,12,6,13] | [1,3,5], [3,5,6], [5,6,7], [6,7,9], [6,7,10], [7,9,6], [9,6,7], [7,10,12], [10,12,6], [12,6,13] |
| Test #2 | [1,3,5,6,7,8,6,7,10,11,12,6,7,9,6,13] | [1,3,5], [3,5,6], [5,6,7], [6,7,8], [6,7,9], [6,7,10], [7,8,6], [8,6,7], [7,9,6], [9,6,13], [7,10,11], [10,11,12], [11,12,6], [12,6,7] |
| Test #3 | [1,3,5,6,7,8,6,13] | [1,3,5], [3,5,6], [5,6,7], [6,7,8], [7,8,6], [8,6,13] |
| Test #4 | [1,3,5,6,13] | [1,3,5], [3,5,6], [5,6,13] |
| Test #5 | [1,2] | [1,2] |
| Test #6 | [1,3,4] | [1,3,4] |

Table 1 - longestPrefixOf test paths and respective test requirements toured by the paths (for edge-pair coverage)

### Edge-Pair Coverage Test #1

The inserted keys were the following, in order: “nougat” and “o”. The SUT had as an argument the string “octopus”.

With this set of arguments, it is possible to correctly achieve the test path #1, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “nougat” (“n”) is less than the first letter of “octopus” (“o”), then the nodes 9 and consequently 6 will be executed;
* Then, the cycle repeats, but this time, since we are already in the right node (which is an “o”), the path 7, 10 will be taken.
* Since I associated a null value with the key “o”, the path 12 and consequently 6 will be followed. Since there are no more nodes, the variable “x” will be null and the termination node 13 is reached, ending the test path #1;

### Edge-Pair Coverage Test #2

This is the test with the longest path, making it the trickiest one to implement.

To achieve this path, the inserted keys were the following, in order: “nougat”, “al”, “a”. The SUT had as an argument the string “ambrosia”.

With this set of arguments, it is possible to correctly achieve the test path #2, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “nougat” (“n”) is greater than the first letter of “ambrosia” (“a”), then the nodes 8 and consequently 6 will be executed;
* Then, the cycle repeats, but this time, since “we are” already in the left node (which is an “a”), the path 7, 10 will be taken.
* In contrast with the above test path, a value is associated with the key “a”. The path 11 and consequently 12 and 6 will be executed.
* Path 7 will be executed (“x” is not null), and, because “m” (from “ambrosia”) is greater than the “l” (from “al”), the flow will go to node 9 (meaning “x” will have a null value, because there are no more nodes in the “right side” of the “l” node).
* Consequently, nodes 6 and 13 will be taken, ending the test path #2.

### Edge-Pair Coverage Test #3

To achieve this path, the inserted key the following: “nougat”. The SUT had as an argument the string “fairy”.

With this set of arguments, it is possible to correctly achieve the test path #3, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “nougat” (“n”) is greater than the first letter of “ambrosia” (“a”), then the nodes 8 and consequently 6 will be executed;
* The cycle will not repeat, because the “x” variable becomes null.
* Consequently, the node 13 will be taken, ending the test path #3.

### Edge-Pair Coverage Test #4

To achieve this path, the trick is to not insert any key-value pair, letting the “x” variable be null. The SUT had as an argument the string “test”.

With this set of arguments, it is possible to correctly achieve the test path #4, because of the following:

* Since the root is null, but the argument of the SUT is not, so the path 1, 3, 5 and 6 will be executed;
* Since the variable “x” is null, then node 13 will be reached, ending the test path #4;

### Edge-Pair Coverage #5

This test is trivial, the argument of the SUT (string “query”) must be null. This way, the path 1, 2 will be taken.

### Edge-Pair Coverage #6

This test is also very simple, the trick is to the SUT have an empty string as argument. This way, the path 1, 3, 4 will be taken.

## Prime Path Coverage

According to the figure 1, a total of 9 unit tests are needed to correctly implement the prime path coverage criteria. The paths traversed by this tests can be observed in the table below (and also in the implemented code):

|  |  |  |
| --- | --- | --- |
| Test # | Test Paths | Test requirements toured by the paths |
| Test #1 | [1,2] | [1,2] |
| Test #2 | [1,3,4] | [1,3,4] |
| Test #3 | [1,3,5,6,7,8,6,7,10,12,6,13] | [1,3,5,6,7,8], [6,7,10,12,6], [8,6,7,10,12], [7,10,12,6,13], [7,8,6,7], [6,7,8,6] |
| Test #4 | [1,3,5,6,13] | [1,3,5,6,13] |
| Test #5 | [1,3,5,6,7,10,11,12,6,7,8,6,13] | [1,3,5,6,7,10,11,12], [6,7,10,11,12,6], [7,10,11,12,6,7], [10,11,12,6,7,8], [7,8,6,13], [6,7,8,6] |
| Test #6 | [1,3,5,6,7,8,6,7,10,11,12,6,13] | [6,7,10,11,12,6], [8,6,7,10,11,12], [1,3,5,6,7,8], [7,10,11,12,6,13], [7,8,6,7], [6,7,8,6] |
| Test #7 | [1,3,5,6,7,9,6,7,10,11,12,6,13] | [6,7,10,11,12,6], [1,3,5,6,7,9], [9,6,7,10,11,12], [7,10,11,12,6,13], [6,7,9,6], [7,9,6,7] |
| Test #8 | [1,3,5,6,7,10,11,12,6,7,10,12,6,13] | [1,3,5,6,7,10,11,12], [6,7,10,11,12,6], [10,11,12,6,7,10], [7,10,11,12,6,7], [6,7,10,12,6], [12,6,7,10,12], [7,10,12,6,13] |
| Test #9 | [1,3,5,6,7,10,11,12,6,7,10,11,12,6,13] | [1,3,5,6,7,10,11,12], [6,7,10,11,12,6], [10,11,12,6,7,10], [11,12,6,7,10,11], [12,6,7,10,11,12], [7,10,11,12,6,13], [7,10,11,12,6,7] |
| Test #10 | [1,3,5,6,7,10,12,6,7,10,11,12,6,13] | [1,3,5,6,7,10,12], [6,7,10,11,12,6], [12,6,7,10,11,12], [7,10,11,12,6,13], [6,7,10,12,6], [10,12,6,7,10], [7,10,12,6,7] |
| Test #11 | [1,3,5,6,7,10,11,12,6,7,9,6,13] | [1,3,5,6,7,10,11,12], [6,7,10,11,12,6], [10,11,12,6,7,9], [7,10,11,12,6,7], [6,7,9,6], [7,9,6,13] |
| Test #12 | [1,3,5,6,7,9,6,7,10,12,6,13] | [1,3,5,6,7,9], [6,7,10,12,6], [9,6,7,10,12], [7,10,12,6,13], [6,7,9,6], [7,9,6,7] |
| Test #13 | [1,3,5,6,7,10,12,6,7,10,12,6,13] | [1,3,5,6,7,10,12], [6,7,10,12,6], [10,12,6,7,10], [12,6,7,10,12], [7,10,12,6,7], [7,10,12,6,13] |
| Test #14 | [1,3,5,6,7,10,12,6,7,9,6,13] | [1,3,5,6,7,10,12], [6,7,10,12,6], [10,12,6,7,9], [7,10,12,6,7], [6,7,9,6], [7,9,6,13] |
| Test #15 | [1,3,5,6,7,10,12,6,7,8,6,13] | [1,3,5,6,7,10,12], [6,7,10,12,6], [7,10,12,6,7], [10,12,6,7,8], [7,8,6,13], [6,7,8,6] |
| Test #16 | [1,3,5,6,7,8,6,7,8,6,13] | [1,3,5,6,7,8], [7,8,6,13], [7,8,6,7], [8,6,7,8], [6,7,8,6] |
| Test #17 | [1,3,5,6,7,8,6,7,9,6,13] | [1,3,5,6,7,8], [7,8,6,7], [8,6,7,9], [6,7,9,6], [6,7,8,6], [7,9,6,13] |
| Test #18 | [1,3,5,6,7,9,6,7,9,6,13] | [1,3,5,6,7,9], [6,7,9,6], [7,9,6,13], [9,6,7,9], [7,9,6,7] |
| Test #19 | [1,3,5,6,7,9,6,7,8,6,13] | [1,3,5,6,7,9], [7,8,6,13], [6,7,9,6], [6,7,8,6], [9,6,7,8], [7,9,6,7] |

Table 2 - longestPrefixOf test paths and respective test requirements toured by the paths (for prime path coverage)

### Prime path Coverage Tests #1, #2, #4 and #12

These test paths were already tested in Edge-Pair Coverage section, so the explanation is the same.

### Prime path Coverage #3

The inserted keys were the following, in order: “nougat” and “a”. The SUT had as an argument the string “ambrosia”.

With this set of arguments, it is possible to correctly achieve the test path #3, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “nougat” (“n”) is greater than the first letter of “ambrosia” (“a”), then the nodes 8 and consequently 6 will be executed;
* Then, the cycle repeats, but this time, since we are already in the right node (which is an “a”), the path 7, 10 will be taken.
* Since I associated a null value with the key “a”, the path 12 and consequently 6 will be followed. Since there are no more nodes, the variable “x” will be null and the termination node 13 is reached, ending the test path #3;

### Prime path Coverage #5

The inserted keys were the following, in order: “m” and “moo”. The SUT had as an argument the string “may”.

With this set of arguments, it is possible to correctly achieve the test path #5, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “m” is equal to the first letter of “may” (“m”), then the node 10 will be executed;
* Here, since the value associated with the node is not null, nodes 11 and 12 and consequently 6 will be executed.
* Then, the cycle repeats, but this time, since we are already in the middle node (which is an “o”), the path 7, 8 will be taken (since “a” is less than “o”) and consequently 6 as well.
* Since left node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #6

The inserted keys were the following, in order: “moo” and “e”. The SUT had as an argument the string “elf”.

With this set of arguments, it is possible to correctly achieve the test path #6, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “moo” (“m”) is greater than the first letter of “elf” (“e”), then the node 8 will be executed and consequently node 6;
* Then, the cycle repeats, but this time, since “we are” already in the left node (which is an “e”), the path 7, 10 will be taken.
* This node has a value associated so nodes 11 and 12 plus 6 will be executed;
* Since the node is null, the cycle ends and termination node 13 is reached

### Prime path Coverage #7

The inserted keys were the following, in order: “moo” and “o”. The SUT had as an argument the string “octopus”.

With this set of arguments, it is possible to correctly achieve the test path #7, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “moo” (“m”) is less than the first letter of “octopus” (“o”), then the node 9 will be executed and consequently node 6;
* Then, the cycle repeats, but this time, since “we are” already in the right node (which is an “o”), the path 7, 10 will be taken.
* This node has a value associated so nodes 11 and 12 plus 6 will be executed;
* Since the node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #8

The inserted keys were the following, in order: “e” and “el”. The SUT had as an argument the string “elijah”.

With this set of arguments, it is possible to correctly achieve the test path #8, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “e” is equal to the first letter of “elijah” (“e”), then the node 10 will be executed;
* There is a value associated with the key “e”, so the node 11 and consequently nodes 12 and 6 will be executed as well.
* Then, the cycle repeats, but this time, since “we are” already in the mid node (which is an “l”), the path 7, 10 will be taken (because the second letter from “elijah” (“l”) equals the key in question) and, because there is a null value associated with the key, node 12 will be traversed and consequently node 6;
* Since the node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #9

The inserted keys were the following, in order: “e” and “el”. The SUT had as an argument the string “elijah”.

This is exactly like the test #8, what differs is that the second key-value pair has an associated value, so the flow in the third iteration of the cycle won’t traverse node 11 from the graph.

### Prime path Coverage #10

The inserted key-value pairs were the following, in order: “ma” and “m”. The SUT had as an argument the string “may”.

With this set of arguments, it is possible to correctly achieve the test path #10, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “m” is equal to the first letter of “may” (“m”), then the node 10 will be executed and, since there is no value associated with this key, nodes 10, 12 and consequently 6 will be traversed;
* Then, the cycle repeats, but this time, since “we are” already in the mid node (which is an “a”), the path 7, 10 will be taken (because the second letter from “may” (“a”) equals the key in question) and, because there is a value associated with the key, nodes 11 and 12 will be traversed and consequently node 6;
* Since the node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #11

The inserted keys were the following, in order: “e” and “elijah”. The SUT had as an argument the string “ez”.

With this set of arguments, it is possible to correctly achieve the test path #11, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “e” is equal to the first letter of “elijah” (“e”), then the node 10 will be executed and, since there is a value associated with this key, nodes 10, 11, 12 and consequently 6 will be traversed;
* Then, the cycle repeats, but this time, the path 7, 9 will be taken (because the second letter from “ez” (“z”) is greater than the key in question (“l”) and consequently node 6 will be traversed;
* Since the right node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #12

The inserted keys were the following, in order: “nougat” and “o”. The SUT had as an argument the string “octopus”.

With this set of arguments, it is possible to correctly achieve the test path #12, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “nougat” (“n”) is less than the first letter of “octopus” (“o”), then the nodes 9 and consequently 6 will be executed;
* Then, the cycle repeats, but this time, the path 7, 10 will be taken (because the first letter from “octopus” (“o”) is greater than the key in question (“o”). Because I associated a null value with the key, node 12 will be executed and consequently node 6;
* Since the mid node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #13

The inserted keys were the following, in order: “ma” and “mars”. The SUT had as an argument the string “ma”.

With this set of arguments, it is possible to correctly achieve the test path #13, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “ma” (“m”) is equal to the first letter of “ma” (“m”), then the nodes 10 and consequently 12 (since the key has a null value associated) and 6 will be executed;
* Then, the cycle repeats, and the path 7, 10 will be taken (because the second letter from “ma” (“a”) is equal than the key in question (“a”). Because I associated a null value with the key, node 12 will be executed and consequently node 6;
* Since the mid node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #14

The inserted key pair was “ma”. The SUT had as an argument the string “men”.

With this set of arguments, it is possible to correctly achieve the test path #14, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “ma” (“m”) is equal to the first letter of “men” (“m”), then the nodes 10 and consequently 12 (since the key has a null value associated) and 6 will be executed;
* Then, the cycle repeats, and the path 7, 9 will be taken (because the second letter from “men” (“e”) is greater than the key in question (“a”). Consequently, node 6 is executed.
* Since the right node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #15

The inserted key was “me”. The SUT had as an argument the string “may”.

With this set of arguments, it is possible to correctly achieve the test path #15, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “me” (“m”) is equal to the first letter of “men” (“men”), then the nodes 10 and consequently 12 (since the key has a null value associated) and 6 will be executed;
* Then, the cycle repeats, and the path 7, 8 will be taken (because the second letter from “may” (“a”) is greater than the key in question (“e”). Consequently, node 6 is executed.
* Since the left node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #16

The inserted keys were the following, in order: “moo” and “go”. The SUT had as an argument the string “eduroam”.

With this set of arguments, it is possible to correctly achieve the test path #6, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “moo” (“m”) is greater than the first letter of “eduroam” (“e”), then the node 8 will be executed and consequently node 6;
* Then, the cycle repeats, but this time, since “we are” already in the left node (which is an “g”), the path 7, 8 will be taken because the first letter from “eduroam” (“e”) is less than the aforementioned letter. Consequently, node 6 is executed;
* Since the left node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #17

The inserted keys were the following, in order: “moo” and “alabama”. The SUT had as an argument the string “episode”.

With this set of arguments, it is possible to correctly achieve the test path #17, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “moo” (“m”) is greater than the first letter of “eduroam” (“e”), then the node 8 will be executed and consequently node 6;
* Then, the cycle repeats, but this time, since “we are” already in the left node (which is a “a”), the path 7, 9 will be taken because the first letter from “episode” (“e”) is greater than the aforementioned letter. Consequently, node 6 is executed;
* Since the left node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #18

The inserted keys were the following, in order: “go” and “moo”. The SUT had as an argument the string “zeta”.

With this set of arguments, it is possible to correctly achieve the test path #18, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “moo” (“m”) is less than the first letter of “zeta” (“e”), then the node 9 will be executed and consequently node 6;
* Then, the cycle repeats, but this time, since “we are” already in the left node (which is an “m”), the path 7, 9 will be taken because the first letter from “zeta” (“z”) is greater than the aforementioned letter. Consequently, node 6 is executed;
* Since the left node is null, the cycle ends and termination node 13 is reached;

### Prime path Coverage #19

The inserted keys were the following, in order: “alabama” and “moo”. The SUT had as an argument the string “episode”.

With this set of arguments, it is possible to correctly achieve the test path #19, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the first letter of the string “alabama” (“a”) is less than the first letter of “episode” (“e”), then the node 9 will be executed and consequently node 6;
* Then, the cycle repeats, but this time, since “we are” already in the right node (which is an “m”), the path 7, 8 will be taken because the first letter from “episode” (“e”) is less than the aforementioned letter. Consequently, node 6 is executed;
* Since the left node is null, the cycle ends and termination node 13 is reached;

# All-Du-Paths Coverage of longestPrefixOf Method

According to the figure 1, a total of 21 (unique) unit tests are needed to correctly implement the All-Du-Paths coverage criteria – some of these are repeated amongst the variables, as it is possible to observe below.

The two following tables (tables 3 and 4) contain the variables associated with All-DU-Path coverage (table 3) as well as the DU paths (table 4).

|  |  |  |
| --- | --- | --- |
| Variables | All DU Path Coverage | Test # |
| query | [1,2] | Test #1 |
| [1,3,4] | Test #2 |
| [1,3,5,6,13] | Test #3 |
| [1,3,5,6,7,8,6,13] | Test #4 |
| x | [1,3,5,6,7,8,6,13] | Test #5 |
| [1,3,5,6,13] | Test #6 |
| [1,3,5,6,7,10,12,6,13] | Test #7 |
| [1,3,5,6,7,9,6,13] | Test #8 |
| [1,3,5,6,7,10,11,12,6,13] | Test #9 |
| [1,3,5,6,7,8,6,7,8,6,13] | Test #10 |
| [1,3,5,6,7,8,6,13] | Test #11 |
| [1,3,5,6,7,8,6,7,10,12,6,13] | Test #12 |
| [1,3,5,6,7,8,6,7,9,6,13] | Test #13 |
| [1,3,5,6,7,8,6,7,10,11,12,6,13] | Test #14 |
| [1,3,5,6,7,9,6,7,8,6,13] | Test #15 |
| [1,3,5,6,7,9,6,7,10,12,6,13] | Test #16 |
| [1,3,5,6,7,9,6,7,9,6,13] | Test #17 |
| [1,3,5,6,7,9,6,7,10,11,12,6,13] | Test #18 |
| [1,3,5,6,7,10,12,6,7,8,6,13] | Test #19 |
| [1,3,5,6,7,10,12,6,13] | Test #20 |
| [1,3,5,6,7,10,12,6,7,10,12,6,13] | Test #21 |
| [1,3,5,6,7,10,12,6,7,9,6,13] | Test #22 |
| [1,3,5,6,7,10,12,6,7,10,11,12,6,13] | Test #23 |
| i | [1,3,5,6,7,9,6,13] | Test #24 |
| [1,3,5,6,13] | Test #25 |
| [1,3,5,6,7,10,12,6,13] | Test #26 |
| [1,3,5,6,7,10,11,12,6,13] | Test #27 |
| [1,3,5,6,7,10,12,6,7,9,6,13] | Test #28 |
| [1,3,5,6,7,10,12,6,7,10,12,6,13] | Test #29 |
| [1,3,5,6,7,10,11,12,6,7,9,6,13] | Test #30 |
| [1,3,5,6,7,10,11,12,6,7,10,12,6,13] | Test #31 |
| c | [1,3,5,6,7,9,6,13] | Test #32 |
| [1,3,5,6,7,8,6,13] | Test #33 |
| [1,3,5,6,7,10,12,6,13] | Test #34 |
| length | [1,3,5,6,13] | Test #35 |
| [1,3,5,6,7,10,12,6,13] | Test #36 |

Table 3 – All DU Path Coverage for longestPrefixOf method

|  |  |
| --- | --- |
| Variables | DU Paths |
| query | [1,3] |
| [1,2] |
| [1,3,5] |
| [1,3,4] |
| [1,3,5,6,7] |
| [1,3,5,6,13] |
| x | [5,6,7] |
| [5,6,13] |
| [5,6,7,10] |
| [5,6,7,8] |
| [5,6,7,9] |
| [5,6,7,10,11] |
| [5,6,7,10,12] |
| [5,6,7,10,11,12] |
| [8,6,7] |
| [8,6,13] |
| [8,6,7,10] |
| [8,6,7,8] |
| [8,6,7,9] |
| [8,6,7,10,11] |
| [8,6,7,10,12] |
| [8,6,7,10,11,12] |
| [9,6,7] |
| [9,6,13] |
| [9,6,7,10] |
| [9,6,7,8] |
| [9,6,7,9] |
| [9,6,7,10,11] |
| [9,6,7,10,12] |
| [9,6,7,10,11,12] |
| [12,6,7] |
| [12,6,13] |
| [12,6,7,10] |
| [12,6,7,8] |
| [12,6,7,9] |
| [12,6,7,10,11] |
| [12,6,7,10,12] |
| [12,6,7,10,11,12] |
| i | [5,6,7] |
| [5,6,13] |
| [5,6,7,10] |
| [10,11] |
| [10,12,6,7] |
| [10,12,6,13] |
| [10,12,6,7,10] |
| [10,11,12,6,7] |
| [10,11,12,6,13] |
| [10,11,12,6,7,10] |
| c | [7,9] |
| [7,8] |
| [7,10] |
| length | [5,6,13] |
| [12,6,13] |

Table 3 – DU Paths for longestPrefixOf method

### All DU Path Coverage Tests #1 - #6, #10 - #36

All these tests (excluding #7, #9 and #10) were already implemented and correctly tested before (in coverage criteria Edge-Pair Coverage and Prime-Path).

### All DU Path Coverage Tests #7

The inserted key was “o”. The SUT had as an argument the string “octopus”.

With this set of arguments, it is possible to correctly achieve the test path #7, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “o” is equal to the first letter of “octopus” (“o”), then the nodes 10 and consequently 12 (since the key has a null value associated) and 6 will be executed;
* Since the mid node is null, the cycle ends and termination node 13 is reached;

### All DU Path Coverage Tests #8

The inserted key was “a”. The SUT had as an argument the string “barbecue”.

With this set of arguments, it is possible to correctly achieve the test path #8, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “a” is less than the first letter of “barbecue” (“b”), then the node 9 and consequently 6 will be executed;
* Since the right node is null, the cycle ends and termination node 13 is reached;

### All DU Path Coverage Tests #9

The inserted key was “o”. The SUT had as an argument the string “octopus”.

With this set of arguments, it is possible to correctly achieve the test path #9, because of the following:

* The arguments, not being null, allow the steps 1,3,5,6 and 7 to be executed;
* Since the single character “o” is equal to the first letter of “octopus” (“o”), then the nodes 10, 11 and consequently 12 (since the key has a value associated) and 6 will be executed;
* Since the mid node is null, the cycle ends and termination node 13 is reached;

# All-Coupling-Use-Paths Coverage of put Method

In this section, I analyze the Coupling du-paths and coverage criteria in the context of the put(String key, T val) method.

A coupling du path for a variable is a path from a last definition of the variable to the first use of this same variable.

As it is possible to observe below, the table contains all the coupling use paths, with the last definition of the variable and its first use, alongside with the method and line of code in which is defined/used:

|  |  |  |
| --- | --- | --- |
| Path # | Last definition  (method, variable, code line) | First use |
| Path #1 | (put, key, line 106) | (contains, key 62) |
| Path #2 | (put, val, line 111) | (put, val, 120) |
| Path #3 | (put, val, line 111) | (put, val, 121) |
| Path #4 | (put, val, line 111) | (put, val, 122) |
| Path #5 | (put, val, line 111) | (put, val, 123) |
| Path #6 | (put, root, line 111) | (put, x, 116) |
| Path #7 | (put, key, line 106) | (put, key 115) |

Table 3 – Coupling du pairs for put method

### All-Coupling-Use-Paths Test #1

The implementation of the test covers paths #1, #3, #6 and #7. The reason it does not cover all of the above tests is because it is only needed to cover at least one coupling du-path from every last definition to a first use of a certain variable: in my case, tests #2, #3, #4 and #5 all cover the same variable, so I chose to implement path #3.

There is only one unit test and it was a simple one: I just insert in the root of the tree two strings (with the SUT) in which the second input’s first letter is greater than the first one’s. The input’s were “chewbacca” and “dorothy”. This way, it is possible to cover the aforementioned paths.

# Logic Based Coverage of longestPrefixOf Method

In this section, I analyze the longestPrefixOf method in the context of logic based coverage. This type of coverage basis itself in logical expressions (for instance, *if* statements) which are implemented in the code of the SUT (more precisely, longestPrefixOf method).

More specifically, I will use *CoC* – Combinatorial Coverage – that finds (in each predicate) test requirements that will contain possible combinations of truth values of the clauses in that specific predicate.

Having the aforementioned in account, my choice was based on the following:

1. The method in question has several conditional statements, more precisely, *if* statements, making it suitable for this approach (according to the brief description given above);
2. This approach can become impractical for predicates that contain multiple clauses, but this is not the case: at most, the “longest” predicate has 2 clauses, making, once again, this approach suitable for the intended purpose;

In the table below, it is possible to observe the predicates alongside with the respective clauses:

|  |  |
| --- | --- |
| Predicates | Clauses |
| Predicate #1: Clause #1 | Clause #1: query == null |
| Predicate #2: Clause #2 | Clause #2: query.length() == 0 |
| Predicate #3: Clause #3 && Clause #4 | Clause #3: x != null |
| Clause #4: i < query.length() |
| Predicate #4 | Clause #5: c < x.c |
| Predicate #5 | Clause #6: c > x.c |
| Predicate #6 | Clause #7: x.val != null  Table 3 – CoC: Predicates and respective clauses of longestPrefixOf method |

### Logic Based Combinatorial Coverage Test #1

The inserted key was null. The SUT had as an argument the string “null”.

With this set of arguments, it is possible to correctly achieve the predicate #1, because of the following:

* The argument of the SUT is null, therefore, predicate #1 is achieved;

### Logic Based Combinatorial Coverage Test #2

The inserted key was null. The SUT had as an argument the string “” (empty).

With this set of arguments, it is possible to correctly achieve the predicate #2, because of the following:

* The argument of the SUT is empty, therefore, predicate #2 is achieved;

### Logic Based Combinatorial Coverage Test #3

The inserted key is null. The SUT had as an argument the string “try”.

With this set of arguments, it is possible to correctly achieve the predicates:

* **¬**clause #1;
* **¬**clause #2;
* **¬**clause #3 and clause #4;

because of the following:

* The arguments, not being null nor empty, **¬**clause #1 and **¬**clause #2 are achieved;
* Since the root of the trie is null, then and variable “i” is still 0, then **¬**clause #3 and clause #4 are achieved;

### Logic Based Combinatorial Coverage Test #4

The inserted keys were the following, in order: “nougat”, “al” and “a”. The SUT had as an argument the string “ambrosia”.

With this set of arguments, it is possible to correctly achieve the predicates:

* **¬**clause #1;
* **¬**clause #2;
* clause #3 and clause #4;
* clause #5;
* clause #6;
* clause #7;
* **¬**clause #2 and clause 4;

because of the following:

* The arguments, not being null nor empty, **¬**clause #1 and **¬**clause #2 are achieved;
* In the first iteration, neither the variable “x” is null nor “i” is less than the SUT’s argument’s length, thus, clause #3 and clause #4 are achieved and because there is a value associated, also clause #7;
* Since the first letter of the string “nougat” (“n”) is greater than the first letter of “ambrosia” (“a”), then clause #5 is achieved;
* Then, the cycle repeats;
* Now, because “m” (from “ambrosia”) is greater than the “l” (from “al”), clause #6 is achieved;
* The node reference will be null, therefore **¬**clause #2 and clause 4 is achieved;

### Logic Based Combinatorial Coverage Test #5

The inserted key was the following “biscuit”. The SUT had as an argument the string “biscuit”.

With this set of arguments, it is possible to correctly achieve the predicates:

* **¬**clause #1;
* **¬**clause #2;
* clause #3 and clause #4;
* clause #3 and **¬**clause #4;

because of the following:

* The arguments, not being null nor empty, **¬**clause #1 and **¬**clause #2 are achieved;
* In the first iteration, neither the variable “x” is null nor “i” is less than the SUT’s argument’s length, thus, clause #3 and clause #4 are achieved;
* In the last iteration, because “i” was incremented and because the variable “x” will point to a null reference, **¬**clause #3 and **¬**clause #4. Also, since I associated a null value with the argument given to the SUT as input, **¬**clause #7 is also achieved;
* The node reference will be null, therefore **¬**clause #2 and clause 4 is achieved;

### Logic Based Combinatorial Coverage Test #6

The inserted key was the following “hyperactive”. The SUT had as an argument the string “hyper”.

This test is very similar to the last one, what differs the most is the following: since the size of the SUT’s argument’s length is less than the key inserted in the trie but is a prefix of it, this will force the variable “i” to be incremented in the penultimate iteration. This results in the variable “x” not being null but the variable “i” not being less than the SUT’s argument’s length, verifying the clause #3 and **¬**clause #4;

# Input State Partitioning – Base Choice Coverage for put method

In this section, the objective is to apply the base coverage criteria and generate test that diverge from the base choice in the context of the put (String key, T val) method.

To give some context, the ISP basis itself in the concept of domain and partitions. The domain part consists in possible values (inputs) that the parameters can have (for the software being tested). Each partition offers useful values that let one test the software in a correct and complete manner. In other words, there is a modeling of the domain in which characteristics are identified - these latter define the building blocks that will divide (compose the partitions) of the input space.

In the table below, it is possible to observe the characteristics (already pre-defined), partitions, my base choice definition (most important block) as well as the tests (variations) of the base block, making up the base choice coverage strategy:

**Note:** The partition column is divided in “True” and “False” as well as “Smallest”, “Largest” and “Typical”. The bold Boolean represents my choice of value for that characteristic. Also, infeasible tests are underlined with “infeasible”.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Characteristic | Partitions | | | | Base Choice | Tests |
| Trie already includes the new key | **True** | | False | | **(True, True, False, Smallest)** | **True, True, False, Smallest** |
| True, False, False, Smallest |
| Trie already includes some new key prefix | **True** | | False | | True, True, True, Smallest (infeasible) |
| Trie is empty | True | | **False** | | False, True, False, Smallest |
| The new key is the smallest/largest/a typical key (in lexicographic terms) | **Smallest** | Largest | | Typical | True, True, False, Largest |
| True, True, False, Typical |

Table 3 – Base Choice coverage specifications

### Base choice Coverage test #1

The inserted keys were the following, in order: “chew”, “chord” and the new key is “chew”.

With this set of arguments, it is possible to correctly achieve the first test in the above table, because of the following:

* The first characteristic is validated since the new key is “chew” and the trie already has it;
* The second characteristic is validated since the new key is “chord” contains a prefix belonging to the word “chew” (“ch”);
* The third characteristic is validated since key-values pairs were introduced to compose the trie;
* The fourth characteristic is validated since “chew” is lexicographically less than “chord”;

### Base choice Coverage test #2

The inserted keys were the following, in order: “drain”, “chew” and the new key is “chew”.

With this set of arguments, it is possible to correctly achieve the first test in the above table, because of the following:

* The first characteristic is validated since the new key is “chew” and the trie already has it;
* The second characteristic is validated since the new key is “chew” and the trie has no word that contain a prefix of it (“drain”);
* The third characteristic is validated since key-values pairs were introduced to compose the trie;
* The fourth characteristic is validated since “chew” is lexicographically less than “drain”;

### Base choice Coverage test #3

This test is infeasible due to the following reason: it is no possible to have a trie including the new key if the trie is empty (characteristics one and three for from the table).

### Base choice Coverage test #4

The inserted keys were the following, in order: “drain”, “chewbacca” and the new key is “chew”.

With this set of arguments, it is possible to correctly achieve the first test in the above table, because of the following:

* The first characteristic is validated since the new key is “chew” does not exist in the trie;
* The second characteristic is validated since the new key is “chew” contains a prefix belonging to the word “chewbacca” (“ch”);
* The third characteristic is validated since key-values pairs were introduced to compose the trie;
* The fourth characteristic is validated since “chew” is lexicographically less than “drain” and “chewbacca”;

### Base choice Coverage test #5

The inserted keys were the following, in order: “flame”, “flamingo” and the new key is “flamingo”.

This is very similar to test #1, what changes is the fact that the new key “flamingo” is lexicographically greater than “flame”.

### Base choice Coverage test #6

The inserted keys were the following, in order: “code”, “drill”, “frightening” and the new key is “drill”.

This is very similar to test #1, what changes is the fact that the new key “drill” is lexicographically greater than “code” but less than “frightening”, hence, being typical.

# Test set via PIT

This section has the objective of using PIT - a mutation testing tool for Java, which can be applied to my source code. A comparison is made between the various criteria, putting in evidence what was achieved by each one in terms of test coverage as well as discovering what was lacking in a few of them, making this tool very useful, practical and of easy understanding.

Having in mind the criteria that were tested by the tool, coverage criteria Edge-Pair coverage and All-DU-Paths did not achieve a 100% coverage in the context of the longestPrefixOf method according to the tool PIT. All the other criteria (Prime Path, Logic Based, Line and Branch Coverage) got the maximum coverage according to the PIT tool.

The reason for those tests not achieving 100% coverage was quite simple: in the case of Edge-Pair, a test was missing and an explanation can be understood below:

**Note:** all the test were implemented after the execution of the tool and a comment in the code was written to advertise that they were introduced after the aforementioned execution, achieving for all of the criteria a 100% coverage.

## Edge-Pair missing branch

After the execution of the tool, a branch was missing - check a condition inside the while in which the first clause is true and the second is clause if false. This means that the variable “x” (the current node in the iteration) in the method is not null but the variable “i” was previously incremented, making the second false (because it becomes equal to the query‘s length, not less). The figure below show’s where the problem was:

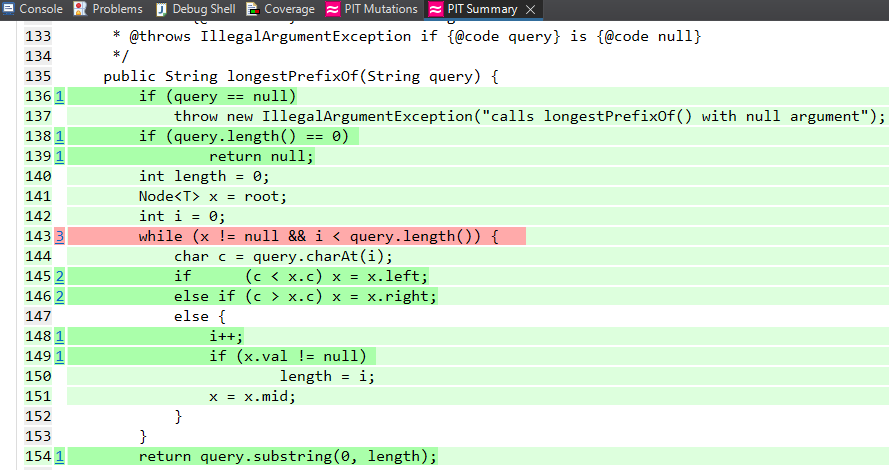


Figure 2 – Non-covered branch by Edge-Pair criteria

## All-DU-Paths missing branches

This coverage criteria lacked some fundamental and trivial branch evaluations, more specifically, the one which are printed in red, in the figure below. No verification was being made for the case when the variable “query” is null nor when it was empty. Besides these two, the other missing branch was exactly the same as the one in the previous criteria.

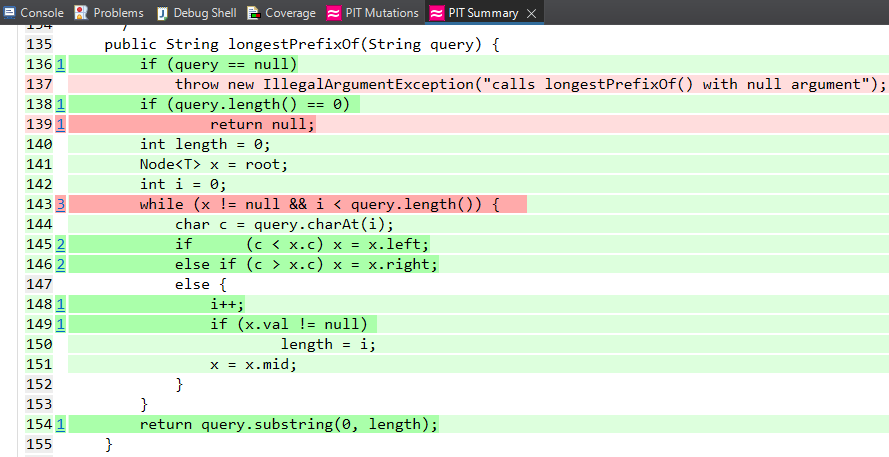


Figure 3 – Non-covered branches by All-DU-Paths criteria

https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage?edges=1+2%0D%0A1+3%0D%0A3+4%0D%0A3+5%0D%0A5+6%0D%0A6+13%0D%0A6+7%0D%0A7+8%0D%0A8+6%0D%0A7+9%0D%0A9+6%0D%0A7+10%0D%0A10+11%0D%0A10+12%0D%0A11+12%0D%0A12+6%0D%0A&initialNode=1&endNode=2+4+13&algorithm2=Edge-Pair%20Coverage