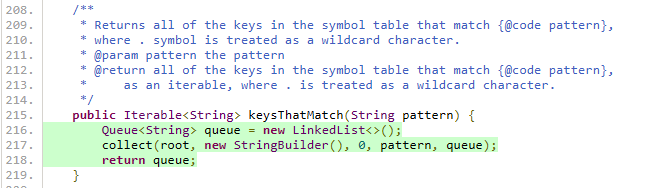
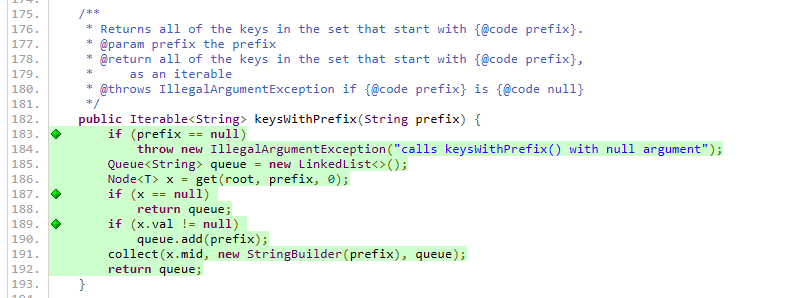
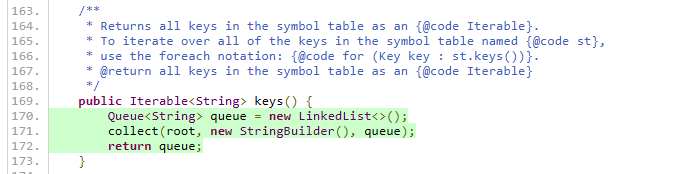
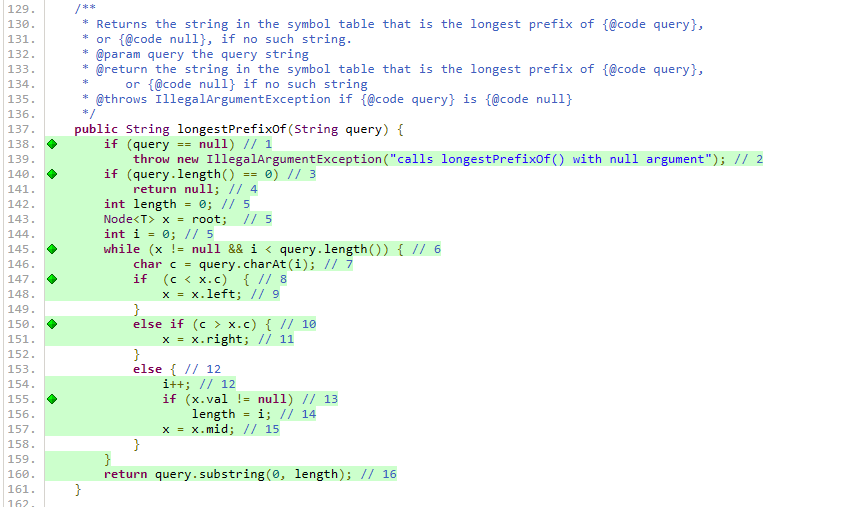
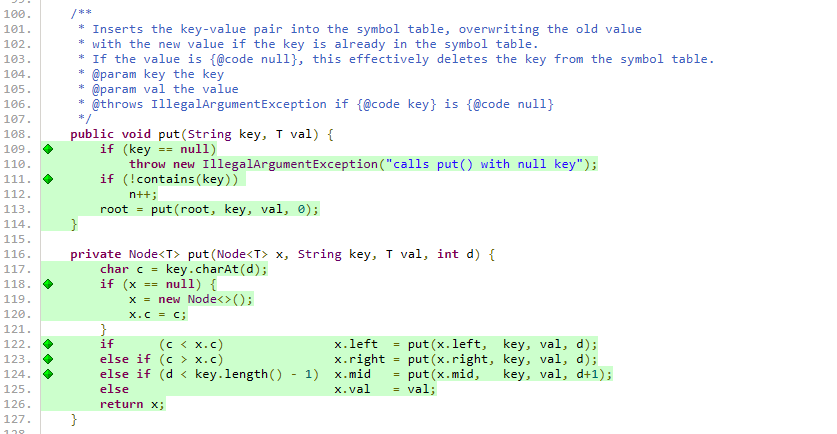
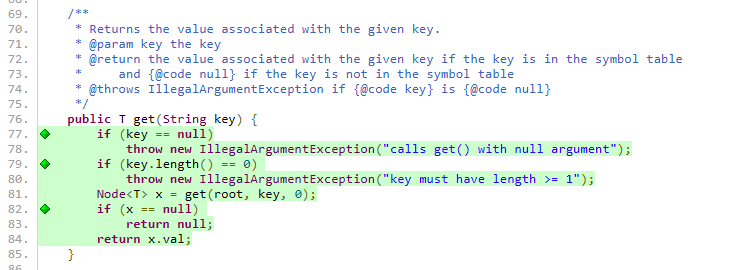
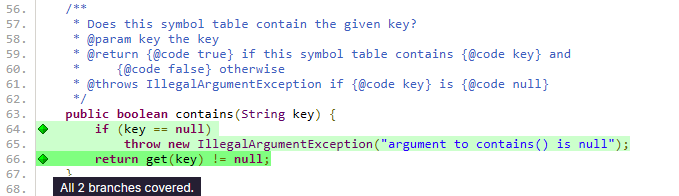
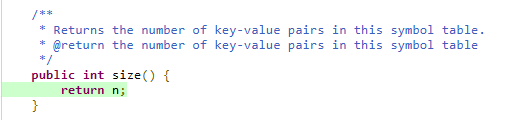
**Verificação e Validação de Software 24/25**

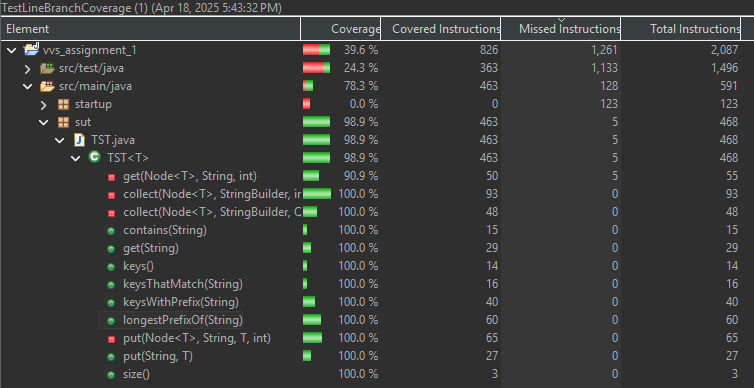
**Assignment 1 Report**

**Tiago de Almeida – fc58161**

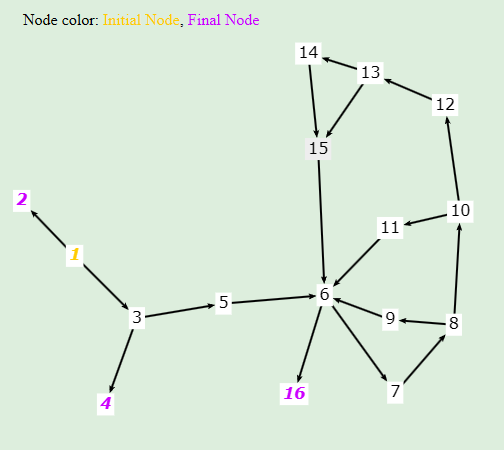
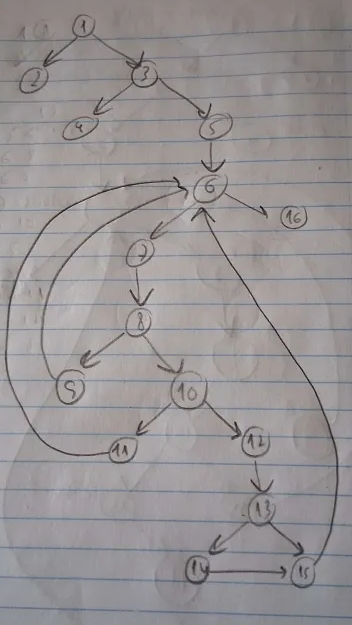
**Line and Branch Coverage for all public methods**

**Coverage Outcomes:**



****

**Edge-Pair and Prime Path Coverage for method *longestPrefixOf()***

**Graph outcomes:**

**Graph Nodes:**

* **N1:** if (query == null)
* **N2:** throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
* **N3:** if (query.length() == 0)
* **N4:** return null;
* **N5:** int length = 0; Node<T> x = root; int i = 0;
* **N6:** while (x != null && i < query.length()) {
* **N7:** char c = query.charAt(i);
* **N8:** if (c < x.c)
* **N9:** x = x.left;
* **N10:** else if (c > x.c)
* **N11:** x = x.right;
* **N12:** else { i++;
* **N13:** if (x.val != null)
* **N14:** length = i;
* **N15:** x = x.mid;
* **N16:** return query.substring(0, length);

**Graph Edges:**

* 1 -> 2
* 1 -> 3
* 3 -> 4
* 3 -> 5
* 5 -> 6
* 6 -> 16
* 6 -> 7
* 7 -> 8
* 8 -> 9
* 8 -> 10
* 10 -> 11
* 10 -> 12
* 12 -> 13
* 13 -> 14
* 13 -> 15
* 14 -> 15
* 9 -> 6
* 11 -> 6
* 15 -> 6

**Coverage Outcomes:**

A screenshot of a computer

AI-generated content may be incorrect.

**All-Du-Paths Coverage for method *longestPrefixOf()***

A diagram of a network

AI-generated content may be incorrect.**A drawing of a diagram

AI-generated content may be incorrect.Graph outcomes:**

**Graph Nodes:**

* **N1:** if (query == null)
* **N2:** throw new IllegalArgumentException("calls longestPrefixOf() with null argument");
* **N3:** if (query.length() == 0)
* **N4:** return null;
* **N5:** int length = 0; Node<T> x = root; int i = 0;
* **N6:** while (x != null && i < query.length()) {
* **N7:** char c = query.charAt(i);
* **N8:** if (c < x.c)
* **N9:** x = x.left;
* **N10:** else if (c > x.c)
* **N11:** x = x.right;
* **N12:** else { i++;
* **N13:** if (x.val != null)
* **N14:** length = i;
* **N15:** x = x.mid;
* **N16:** return query.substring(0, length);

**Graph Edges:**

* 1 -> 2
* 1 -> 3
* 3 -> 4
* 3 -> 5
* 5 -> 6
* 6 -> 16
* 6 -> 7
* 7 -> 8
* 8 -> 9
* 8 -> 10
* 10 -> 11
* 10 -> 12
* 12 -> 13
* 13 -> 14
* 13 -> 15
* 14 -> 15
* 9 -> 6
* 11 -> 6
* 15 -> 6

**Variables Considered:** x, i, c, length, query

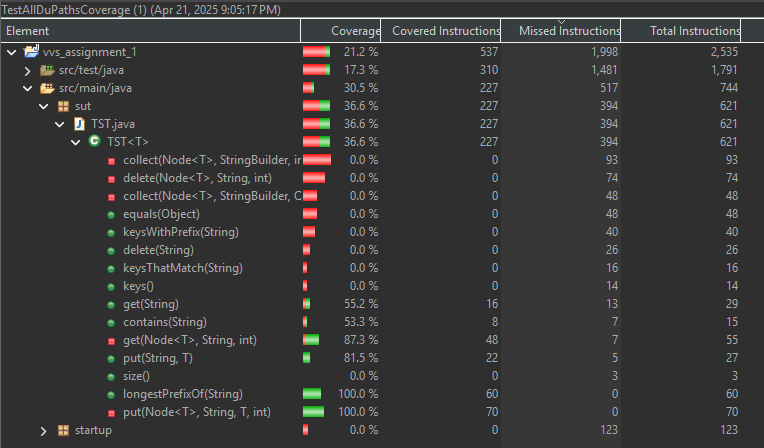
**Definitions:**

* **x:** 5, 9, 11, 15
* **i**: 5, 12
* **c:** 7
* **length:** 5, 14
* **query:** -

**Uses:**

* **x:** 6, 8, 9, 10, 13, 15
* **i**: 6, 7, 12, 14
* **c:** 8, 10
* **length:** 16
* **query:** 1, 3, 6, 7, 16

**Coverage Outcomes:**



**All-Coupling-Use-Paths Coverage for private method *put()***

A computer screen shot of code

AI-generated content may be incorrect.

|  |  |
| --- | --- |
| **last-def** | **first-use** |
| key: {1} | key: {i} |
| val: {1} | val: {vi,viii,x,xi} |
| x: {1,4} | x: {ii} |
| d: {1,10} | d: {i} |
| x’: {vi,viii,x,xi} | x’: {12} |

**Coverage Outcomes:**

A screenshot of a computer

AI-generated content may be incorrect.

**Logic-Based Coverage for method *longestPrefixOf()***

**Selected Coverage Criteria:** Predicate Coverage (PC)

**Justification:** It guarantees that each decision point is tested for both possible outcomes (e.g., a null input vs. a valid string, an empty string vs. a non-empty string, traversing the trie vs. terminating early). It exposes edge cases, like null queries, empty strings, and scenarios where no prefix matches. It helps validate correct traversal logic in the while loop and that the substring computation is only reached with valid conditions.

**List of predicates:**

* (query == null)
* (query.length() == 0)
* (x != null && i < query.length())
* (c < x.c)
* (c > x.c)
* (x.val != null)

*(In the code, above each test, there is the respective predicate coverage of each test)*

**Coverage Outcomes:**

A screenshot of a computer

AI-generated content may be incorrect.

**Base Choice Coverage for method *put()***

**Highest probability of outcomes to create Base Choice:**

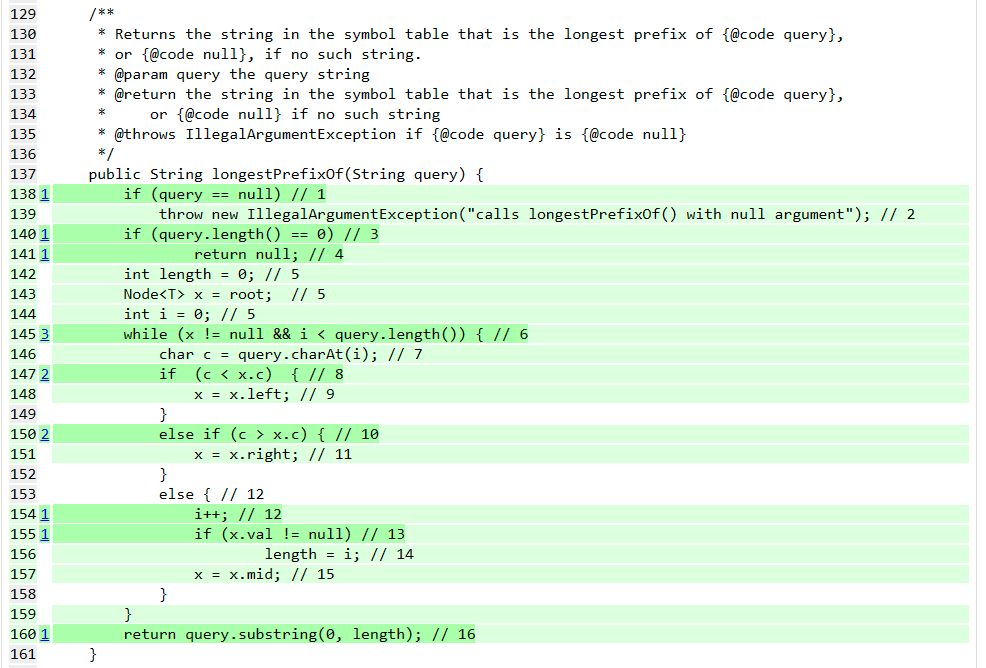
* Trie already includes the new key **->** **NO**
* Trie already includes some new key prefix **->** **NO**
* Trie is empty **->** **NO**
* The new key is the smallest/largest/typical key **-> TYPICAL**

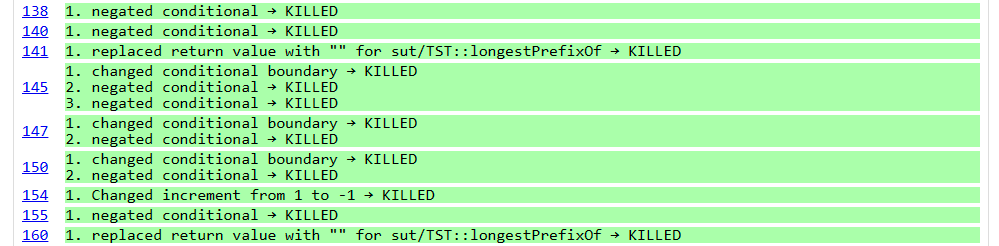
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Method | Includes Key | Prefix Exists | Empty | Lex Order |
| testBaseChoice() | No | No | No | Typical |
| testIncludesNewKey() | Yes | No | No | Typical |
| testIncludesNewPrefix() | No | Yes | No | Typical |
| testEmpty() | No | No | Yes | Typical |
| testSmallest() | No | No | No | Smallest |
| testLargest() | No | No | No | Largest |

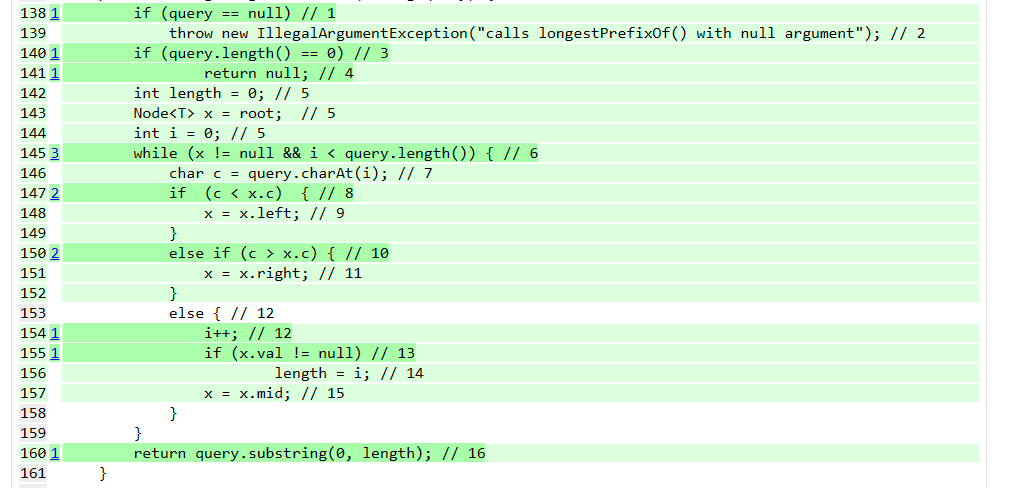
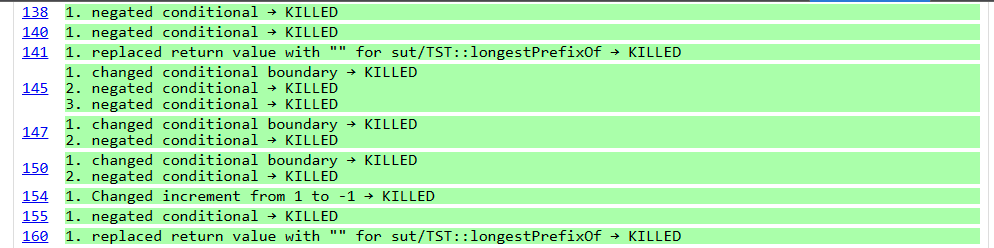
A screenshot of a computer

AI-generated content may be incorrect.**Coverage Outcomes:**

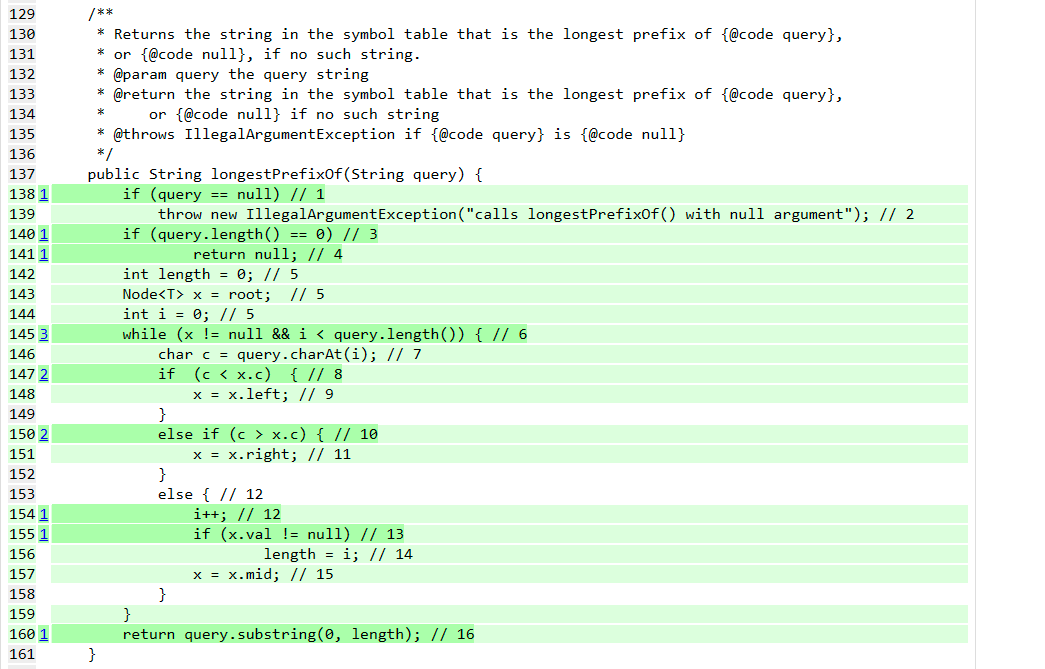
**Mutation coverage achieved by each criteria for method *longestPrefixOf()***

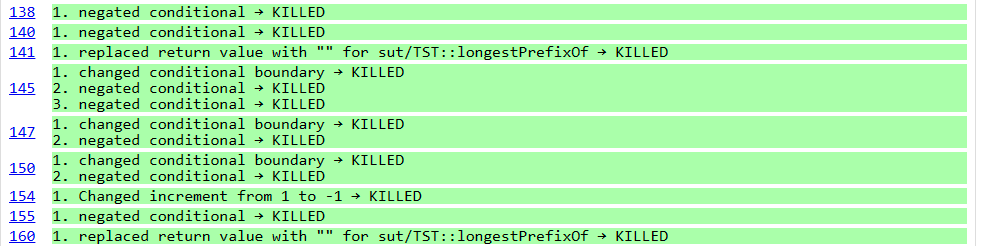
**Line and Branch Coverage: **

****

**Edge-Pair and Prime Path Coverage:**

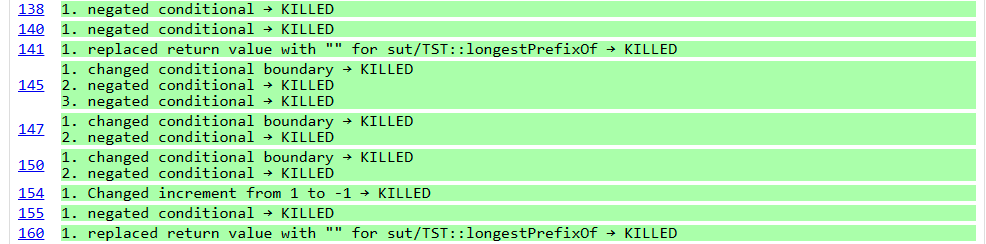
**All-Du-Paths Coverage:**

****

****

**Logic-Based Coverage:A screenshot of a computer program

AI-generated content may be incorrect.**

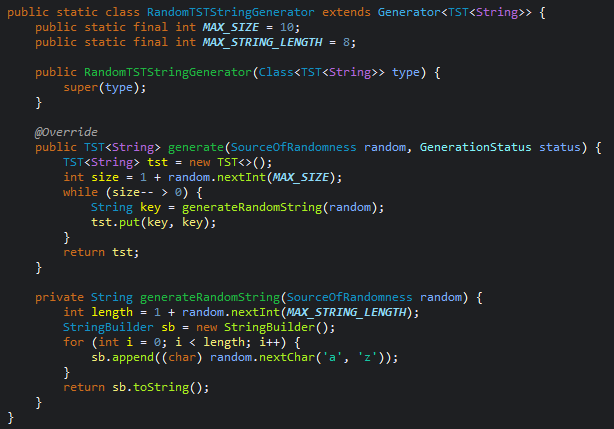
****

**JUnit QuickCheck**

**equals() and delete():**

A screen shot of a computer code

AI-generated content may be incorrect.

**Created a generator of TST<String>:**

**List of test descriptions:**

* **insertionOrderIndependence() :** Created 2 trees, shuffled the words of one of them, and compared their states with equals(), asserting equals;
* **removeAllKeys() :** Using the list of keys, deleted them all, and then asserted true that the size of the tree is 0;
* **insertRemoveDoesNotChangeInitialValue() :** Created a copy of the tree, to save the initial state, then removed one of the keys and added it again, then asserting that both tree states are equal;
* **prefixSubset() :** Added some words that are in keysWithPrefix(“su”) and keysWithPrefix(“sub”) to the tree, then asserted true that all the keys present in the “sub” set are also in the “su” set.