Masters in Informatics Engineering

Software Quality and Dependability

Assignment 2

Dynamic Software Testing

João Melo | 2019216747

Miguel Faria | 2019216809

Index

1 Introduction

The purpose of this test plan is to outline the testing approach and strategies for conducting dynamic software testing on the Splay function for Splay trees. The Splay function is viewed as a critical component of the target software, and its functionality will be tested to ensure correctness and reliability.

This test plan provides a comprehensive view of the testing procedure, techniques, and deliverables, while also considering the potential risks and challenges associated with the software being tested.

Throughout this assignment report, we will describe the testing activities, deliverables, and criteria for test completion, as well as the features that will be analysed. The plan also addresses software risk issues, potential impacts of imperfect testing, and mitigation actions to ensure thorough testing and reliable results.

In this project, the main objective is to demonstrate the effectiveness of the Splay function through comprehensive testing and provide valuable insights for further improvements or enhancements.

2 Software Risk Issues

In this assignment, and as previously referred, the piece of code to be analysed will be the Splay function, used with the data structure of Splay trees. As it belongs to the recursive type, in cases where the tree to be analysed has a large number of nodes, it becomes a task of high complexity to investigate how the algorithm works.

As this type of structure is applied in parts of critical systems, such as in caching systems, for example, if the Splay function isn't well implemented, it can lead to unbalanced trees, which may lead to data loss/corruption, degradation in the performance of the system and consequently increased complexity and maintenance.

For the implementation of the code itself, as it is applied in basic *Python3*, there are no dependencies which may corrupt the well functioning of it.

3 Items and Features to be Tested

In this project, the item that will be tested is only the execution of the Splay function. Although this method calls other functions, such as *rightRotate, leftRotate* and itself, while making use of the class *Node*, the analysis will only be focused on the *splay* function. The features to be tested from an end-user perspective include maintaining tree balance during search operations, correctly bringing the searched key to the root, and handling edge cases.

4 Items and Features not to be Tested

All the items besides the previously mentioned *splay* function are not covered by this testing plan. This includes other components of the software system that are directly or indirectly related to the Splay function, like the methods to apply Rotations or insert new Nodes.

5 Testing Approach

In the testing approach for the Splay function, both black box testing and white box testing techniques were applied.

White box testing examined the internal structure and implementation of the Splay function. Test cases were designed based on knowledge of the code and algorithmic logic of the function. The goal was to achieve high code coverage and ensure that all possible paths and conditions within the function were tested to identify any code errors or inaccuracies.

Black box testing, on the other hand, focused on testing the functionality of the Splay function from an external perspective, without considering its internal implementation. Test cases were designed based on expected behaviour and requirements, with the goal of validating the correctness of the function's output based on given inputs.

5.1 White Box Testing

The white box testing techniques used when testing this software were *Data Flow* and *Control Flow*. In both, the thought process while developing them was to create the graph associated with it and identify the different kinds of possible paths, in accordance with each type.

5.1.1 Control Flow

Control flow refers to the order in which statements are executed in a program, determined by the flow of control statements, such as conditionals (if-else), loops, and function calls. The control flow graph, located in *Attachment 1*, visually represents the paths of execution in a program, with code segments representing specific blocks and edges representing the flow of control between them. Understanding control flow helps in analysing program behaviour, identifying potential errors, and optimising code execution.

By verifying this graph, we can conclude that the cyclomatic complexity is as follows:

$$V(G) = P + 1 = 10 + 1 = 11$$

with P being the number of predicate nodes. In addition, it is possible to identify several distinct paths, presented in *Table 1*.

Path ID	Path
P1	1,2
P2	1,3,4,6
P3	1,3,5,8
P4	1,3,4,7,10,13
P5	1,3,4,7,12,13
P6	1,3,4,7,12,11,14,13
P7	1,3,4,7,12,11,14,15,13
P8	1,3,5,9,17,18,19,20
P9	1,3,5,9,17,18,20
P10	1,3,5,9,16,20
P11	1,3,5,9,16,21,20

Table 1 - Distinct Paths

5.1.2 Data Flow

Data flow refers to the movement of data within a program, tracking how values are assigned, modified, and used throughout the execution. The data flow graph, posted in *Attachment 2*, visually represents the dependencies and transformations of data within a program, with nodes representing variables or expressions and edges representing the data flow between them. Understanding data flow helps in analysing how data is manipulated, identifying data dependencies, and detecting potential data-related issues such as uninitialized variables or incorrect data transformations.

Through the analysis of the data flow graph, it's easy to verify that the variables of the Splay function, *root* and *key*, are being used in almost all nodes/edges. This would make the definition of All Definition-Use Paths (ADUPs) somewhat meaningless, as the criteria in the function don't present any valuable information.

5.2 Black Box Testing

To perform this testing method, we took into consideration several analysis techniques, namely the concepts of:

- Equivalence Class Partitioning: dividing the input data into groups or classes that are expected to exhibit similar behaviour; by testing representative inputs from each class, we can ensure that we cover the different possible scenarios efficiently.
- Boundary Value Analysis: testing the boundary or edge conditions of input values; by examining values at the lower and upper boundaries, as well as just inside and outside these boundaries, we can identify potential issues related to boundary conditions that may affect the system's behaviour.

Consequently, several tests were defined, including cases with valid, invalid, and boundary inputs, aiming to expand the testing coverage.

5.2.1 Valid Test Cases

Valid tests are designed to evaluate the expected behaviour of a system by providing inputs that fall within the acceptable range of values. These tests verify that the system functions correctly under normal operating conditions, producing the expected outputs.

The test cases considered for valid inputs are the following:

Test ID	Name	Description
1	Root node with a balanced Splay Tree	Test the splay function's behaviour on a balanced tree structure where the tree is evenly distributed
2	Root node with an unbalanced Splay Tree	Evaluate the splay function's performance on a tree that is heavily skewed to one side, either left or right
3	Single node tree	Assess how the splay function handles a tree with only one node
4	Smallest and largest keys	Test the behaviour of the splay function when searching for the smallest and largest possible key values
5	Multiple repeated keys	Evaluate how the splay function handles a tree with multiple nodes containing the same key value

Table 2 - Valid Test Cases

5.2.2 Invalid Test Cases

Invalid tests are used to assess how the system handles unexpected or erroneous inputs. These tests aim to find potential issues, such as error handling or improper input validation, and that the system correctly handles them.

The test cases considered for invalid inputs are the following:

Test ID	Name	Description
6	Empty tree	Test how the splay function handles the scenario when the root node is None, indicating an empty tree
7	Invalid key	Test the behaviour of the splay function when searching for a key that does not exist in the tree
8	Invalid tree structure	Evaluate how the splay function handles invalid or inconsistent tree structures
9	Unexpected data types	Check how the splay function handles unexpected data types for inputs

Table 3 - Invalid Test Cases

5.2.3 Boundary Test Cases

Boundary tests focus on values that are at the edge or limit of what the system can handle. These tests aim to identify any issues that may arise when the input values are at their minimum or maximum limits, validating that it responds correctly with edge cases.

The test cases considered for boundary inputs are the following:

Test ID	Name	Description
10	Single-node tree with the lowest and highest possible keys	Evaluate how the splay function handles a tree with a single node containing the lowest and highest possible key values
11	Tree with all keys being the same	Assess how the splay function behaves when all nodes in the tree have the same key value
12	Tree with two nodes and specific key orders	Test the splay function's behaviour with a small tree containing two nodes and different key orders
13	Tree with extreme unbalanced structure	Evaluate the splay function's performance on a tree that is extremely unbalanced, either heavily skewed to the left or right

Table 4 - Boundary Test Cases

5.3 Complementary Testing

It was decided to do some additional testing, where we check if the operations involved in the Splay function return the correct value and maintain the proper tree structure. For that, a Python library, *Hypothesis*, is used to do property-based testing, generating random lists of integers with varying lengths within the specified range (1000 in this case) to serve as input cases.This library autonomously explores different combinations of input data to cover a wide range of scenarios and edge cases, increasing the likelihood of finding bugs or unexpected behaviour in the splay function.

6 Item Pass/Fail Criteria

The Splay function takes as input the root of a Splay tree and the key/value to look for within the tree, and it returns the node that contains it if the value is kept in the tree. If the value isn't present in the Splay tree, in the case of this Splay function, the last checked node will be returned, this being the node that could be closest to it.

When defining the test cases for the white box testing, the criteria for it being passed or failed relied on how the code behaved, more so, the path that was followed by specific inputs.

In black box testing, if there was an error that interrupted the execution of the code, it would be considered a failure, and when a value was returned from the Splay function, it would be compared with the expected returning value: if equal, a pass; otherwise, a fail.

7 Test Deliverables

In this test plan, several files were used in testing and created to document the procedure. The following list enumerates the delivered file:

- *splay_tree.py* - original code, containing the *splay* function
- *white_box_testing.py* - do White Box Testing
- *black_box_testing.py* - do Black Box Testing
- *function_testing.py* - do Functionality Testing

8 Environmental Needs

During the execution of the described tests, the version of *Python* used was *Python 3.11* with an IDE, in this case, PyCharm. Regarding the tools involved, *PyTest* was used for verifying the test inputs in Black Box Testing, while *Hypothesis* was employed for the overall testing of the function, including the tree structure and correct element search. In this way, to replicate the tests, simply run the corresponding files.

9 Staffing and Responsibilities

While creating and executing the different tests and documenting the analysis of the function flows, the distribution of roles and responsibilities was always balanced and distributed within the task at hand, so there wasn't a separation of tasks in the different segments analysed. Overall, the staffing for this project involved a collaborative and equal partnership between both team members, with shared responsibilities and active participation throughout the development and testing phases.

10 Test Completion Report

In this segment of the report, the results from the tests developed and applied in each of the previously described testing phases will be presented in tables, following the pass/fail criteria defined in the sixth section.

10.1 White Box Testing

10.1.1 Control Flow

The results obtained while testing using the control flow paths were the following.

Path ID	Test ID	Input	Output	Result
1	0	root = None root = splay(root, 0)	1,2	Pass
	1	root = newNode(50) root = splay(root, 50)	1,2	Pass
2	2	root = newNode(50) root = splay(root, 30)	1,3,4,6	Pass
3	3	root = newNode(50) root = splay(root, 70)	1,3,5,8	Pass
4	4	root = newNode(50) root.left = newNode(30) root = splay(root, 20)	1,3,4,7,10,13	Pass
5	5	root = newNode(50) root.left = newNode(30) root = splay(root, 30)	1,3,4,7,12,13	Pass
6	6	root = newNode(50) root.left = newNode(30) root = splay(root, 35)	1,3,4,7,12,11,14,13	Pass
7	7	root = newNode(50) root.left = newNode(30) root.left.right = newNode(35) root = splay(root, 35)	1,3,4,7,12,11,14,15,13	Pass
8	8	root = newNode(50) root.right = newNode(70) root.right.left = newNode(65) root = splay(root, 65)	1,3,5,9,17,18,19,20	Pass
9	9	root = newNode(50)	1,3,5,9,17,18,20	Pass

		root.right = newNode(70) root = splay(root, 65)		
10	10	root = newNode(50) root.right = newNode(70) root = splay(root, 70)	1,3,5,9,16,20	Pass
11	11	root = newNode(50) root.right = newNode(70) root = splay(root, 80)	1,3,5,9,16,21,20	Pass

Table 5 - Tests done for Control Flow Testing

As observable in *Table 5*, all the tests were successful, thus proving that there are no impossible paths present in the Splay function.

10.1.2 Data Flow

For the reasons explained in *5.1.2*, no tests were done regarding Data Flow Testing.

10.2 Black Box Testing

All the tests executed in Black Box Testing passed, as it's observable in *Figure 1*.

The tests and the results obtained from executing them will be presented in the subpoints to come from this type of testing.

```
============================= test session starts ==============================
collecting ... collected 13 items

black_box_testing.py::test_valid_balanced_tree PASSED                    [  7%]
black_box_testing.py::test_valid_unbalanced_tree PASSED                  [ 15%]
black_box_testing.py::test_valid_single_node_tree PASSED                 [ 23%]
black_box_testing.py::test_valid_smallest_and_largest_keys PASSED        [ 30%]
black_box_testing.py::test_valid_multiple_repeated_keys PASSED           [ 38%]
black_box_testing.py::test_invalid_empty_tree PASSED                     [ 46%]
black_box_testing.py::test_invalid_key PASSED                            [ 53%]
black_box_testing.py::test_invalid_tree_structure PASSED                 [ 61%]
black_box_testing.py::test_invalid_unexpected_data_types PASSED          [ 69%]
black_box_testing.py::test_boundary_single_node_tree_lowest_and_highest_keys PASSED [ 76%]
black_box_testing.py::test_boundary_tree_all_keys_same PASSED            [ 84%]
black_box_testing.py::test_boundary_tree_two_nodes_specific_key_orders PASSED [ 92%]
black_box_testing.py::test_boundary_tree_extreme_unbalanced_structure PASSED [100%]

============================== 13 passed in 0.04s ==============================
```

Figure 1 - Output from executing all Black Box Testing

10.2.1 Valid Test Cases

The results obtained while testing using the valid inputs were the following.

Test ID	Input	Assertion	Result
1	root = newNode(100) root.left = newNode(50) root.right = newNode(200) root.left.left = newNode(40) root.left.left.left = newNode(30) root.left.left.left.left = newNode(20) splayed_root = splay(root, 20)	splayed_root.key == 20	Pass
2	root = newNode(100) root.right = newNode(200) root.right.right = newNode(300) root.right.right.right = newNode(400) splayed_root = splay(root, 400)	splayed_root.key == 400	Pass
3	root = newNode(100) splayed_root = splay(root, 100)	splayed_root.key == 100	Pass
4	root = newNode(100) root.left = newNode(50) root.left.left = newNode(20) root.right = newNode(200) root.right.right = newNode(300) splayed_smallest = splay(root, 20)	splayed_smallest.key == 20	Pass
5	root = newNode(100) root.left = newNode(50) root.left.left = newNode(50) root.right = newNode(200) root.right.right = newNode(200) splayed_root = splay(root, 50)	splayed_root.key == 50	Pass

Table 6 - Tests done for Valid Test Cases

As observable in Table 6, the tests for this section of black box testing passed.

10.2.2 Invalid Test Cases

The results obtained while testing using the invalid inputs were the following.

Test ID	Input	Assertion	Result
6	root = None splayed_root = splay(root, 100)	assert splayed_root is None	Pass
7	root = newNode(100) root.left = newNode(50)	splayed_root.key == 200	Pass

	root.right = newNode(200) splayed_root = splay(root, 300)		
8	root = newNode(100) root.left = newNode(50) root.left.right = newNode(120) root.right = newNode(200) splayed_root = splay(root, 120)	splayed_root.key != 120	Pass
9	root = newNode(100) splayed_root = splay(root, "invalid")	pytest.raises(TypeError)	Pass

Table 7 - Tests done for Invalid Test Cases

As observable in Table 7, the tests for this section of black box testing passed.

10.2.3 Boundary Test Cases

The results obtained while testing using the boundary inputs were the following.

Test ID	Input	Assertion	Result
10	root = newNode(float('-inf')) root.right = newNode(float('inf')) splayed_lowest = splay(root, float('-inf'))	splayed_lowest.key == float('-inf') splayed_highest.key == float('inf')	Pass
11	root = newNode(100) root.left = newNode(100) root.right = newNode(100) splayed_root = splay(root, 100)	splayed_root.key == 100	Pass
12	root = newNode(100) root.left = newNode(50) root.right = newNode(200) splayed_asc = splay(root, 200) splayed_desc = splay(root, 50) splayed_same = splay(root, 100)	splayed_asc.key == 200 splayed_desc.key == 50 splayed_same.key == 100	Pass
13	root = newNode(100) root.right = newNode(200) root.right.right = newNode(300) root.right.right.right = newNode(400) root.right.right.right.right = newNode(500) splayed_root = splay(root, 500)	splayed_root.key == 500	Pass

Table 8 - Tests done for Boundary Test Cases

As observable in Table 8, the tests for this section of black box testing passed.

10.3 Complementary Testing

The property-based testing approach provides a systematic and automated way to assess the correctness of the implementation across various inputs, increasing confidence in its reliability. The successful passing of the test indicates that the splay tree implementation appears to be functioning correctly, as it maintains the expected properties during the splay operation.

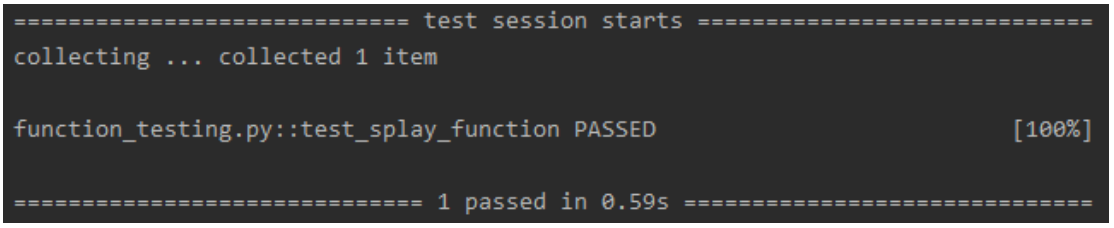


Figure 2 - Output from executing the Complementary Testing

Conclusions

Throughout the execution of this assignment, it was possible to get a better understanding of how Dynamic Software Testing works and how it can be applied. A real-life scenario was simulated to show the importance of the code to work properly, and the integration of tools to help when testing using the different types of techniques.

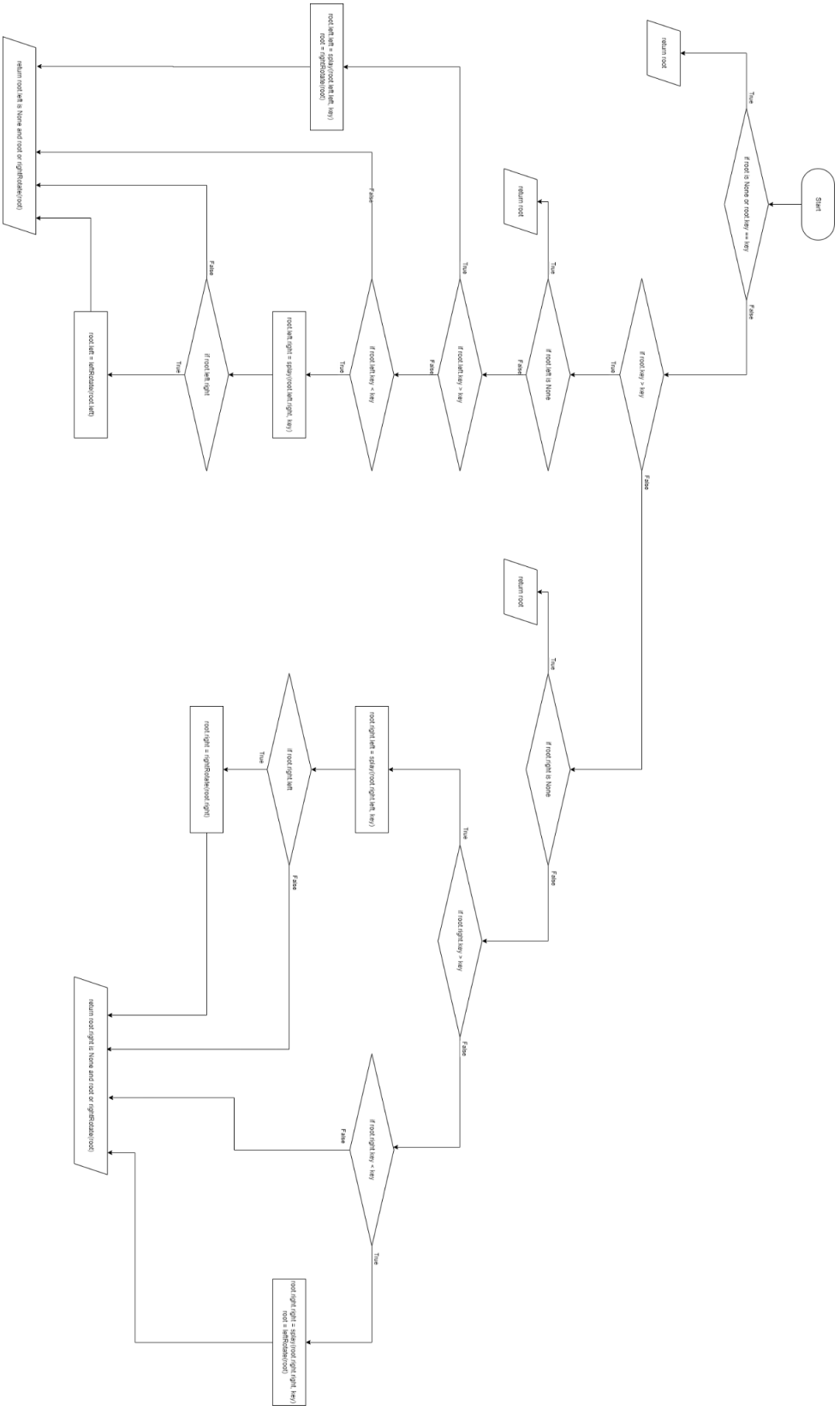
By applying both black box and white box testing techniques, comprehensive test coverage was achieved. Black box testing focused on external behaviour and requirements, while white box testing delved into the internal structure and code of the function. This combination allowed us to verify that the function has no apparent problems and ensure the overall Splay tree implementation.

Resources

[1] Source Code, https://www.geeksforgeeks.org/searching-in-splay-tree/

Attachments

Attachment 1 - Control Flow Graph



Attachment 2 - Data Flow Graph