Mutation Testing

#### 

#### 

Team 16:

Raghav Gupta

Chelsea Jaculina

Bala Nyan Kwaw

Mike Lee

Yihua Li

**February 18, 2018**

**Professor Badari**

**CMPE 187**

# Table of Contents

[**Table of Contents**](#_fvgztqx5fz61) **1**

[**Overview**](#_8pvx7xuczgkg) **2**

[Introduction](#_rfoa3jdhntvi) 2

[Purpose](#_gxjj4t1fsork) 2

[**Oracle**](#_mcynqj6harbr) **3**

[Environment](#_hxp4hhr1f9w7) 3

[Scope](#_nwq4ro20xmw6) 3

[**Test Suite, Mutants, and Mutation Testing**](#_ofs5nz8bmdo6) **4**

[**Mutation Score**](#_yo7qkh4f5f9j) **6**

[**Results**](#_2hi1hjv6arwr) **7**

[**Lessons Learned**](#_996e3tixdkie) **7**

## 

# Overview

## Introduction

This is a program that implements the data structure of Binary Search Tree (BST) using nodes. Initially, an empty BST will be created. It will then insert different nodes with different keys (value) into the BST in which it will change the current state of each node in the BST. The properties of a BST are of the following,1, the left subtree contains only nodes with keys less than the node’s key; 2, the right subtree contains only nodes with keys greater than the node’s key; 3, the left and right subtree can also be a BST.

## Purpose

The purpose of this program is to use the concept of mutation testing to test the complexity of the Binary Search Tree program and to introduce us to the white-box unit testing technique. Our goal of using mutation testing is to check if our Binary Search Tree source code detects any mutants. That is, when our source code is executed the mutant will be discovered and be killed by the test suite that we provide.

# Oracle

## Environment

The hardware used for programming and testing is a Macbook Pro 13.3 from 2015, with a 2.7GHz Intel Core I5 processor, 8GB 1867MHz DDR4 of memory.

The software used for programming and testing is OS X High Sierra version 10.13.3. The program is written and tested in Java code using the Eclipse version Oxygen, with Java version 8 build 1.8.0\_144. As we discussed in the purpose statement, the development of our project incorporated test-driven development using starter code that we retrieved from the website Geeksforgeeks.com (<https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/>). We then created test cases for each conditional and return statements. The test cases each held its set of mutations which were then tested in the IDE by an automated mutation testing software. Below there are test cases and the mutations for each.

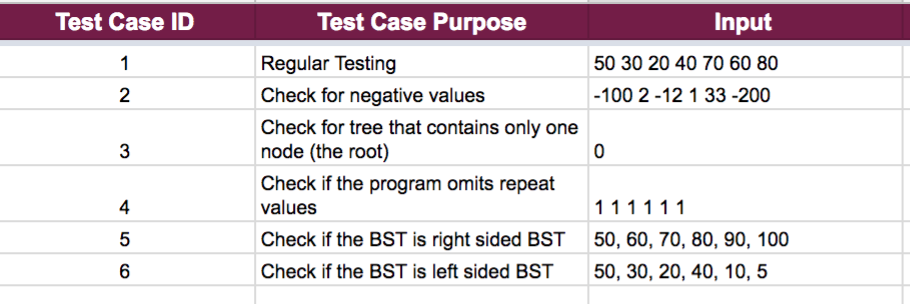
## Scope

Our primary goal is to test the Binary Search Tree program by modifying several lines of code and testing whether or not it will pass the test suite. Our unit of testing will focus on checking the validity of a BST. Some test cases that we incorporated was checking for a single node, an empty tree, determine whether or not both sides of the BST were valid, and check if the tree is traversal.

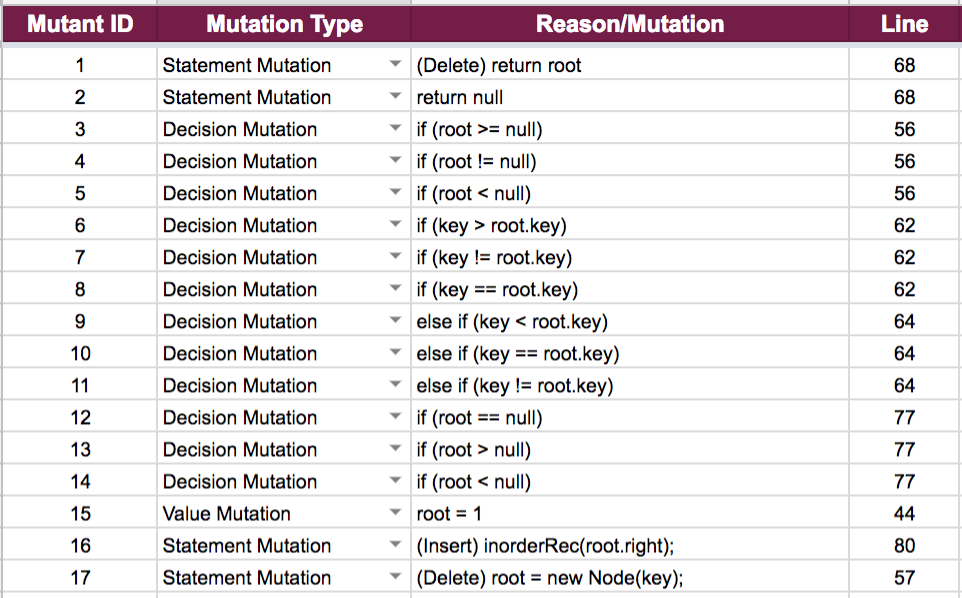
# Test Suite, Mutants, and Mutation Testing

A test suite is a set of tests that tend be in done in units. Each test suite helps validate if the program is working the way that it is expected to. In **Figure 1**, shows a table consisting of our test cases that our team has written for our Binary Search Tree Program.

Mutation testing involves mutants, mutant kills, and equivalents. Mutants are defined as a program in which a bug is introduced. Mutant kills are where the test discovers the error and equivalents are where mutants are not identifiable and need new test cases to be discovered. **Figure 2** summarizes the number of mutants that our team did during the unit testing process. **Figure 3** (see Mutation Testing.pdf attachment), showcases what we did for the mutation testing and if the mutation is an equivalent or killed.



**Figure 1. Test Cases for Binary Search Tree Java Program**

****

**Figure 2. Mutants for Binary Search Tree Java Program**

**Figure 3. Mutation Testing for Binary Search Tree Java Program**

***(Check Attachments)***

# 

# Mutation Score

The mutation score is determined by taking the ratio of the number of mutations discovered and dividing it by the total number of mutations we came up with. This ratio will give us the percentage of how much are test cases are adequate. Ideally, we want our test cases to be at a percentage of 100.

As a team, our Binary Search Tree Program has a total of 6 test cases, each with 17 mutants. The result of mutation testing for each test case is:

|  |  |  |
| --- | --- | --- |
| Test case | Killed Mutant | Equivalent |
| 1 | 16 | 1 |
| 2 | 16 | 1 |
| 3 | 10 | 7 |
| 4 | 12 | 5 |
| 5 | 16 | 1 |
| 6 | 15 | 2 |

Additionally, there is a functionally-equivalent mutant in each test case (Mutant #11, total of 6), therefore, the Mutation Score of the mutation testing is:

Ms = == = 88.54%.

# Results

Our team was able to test our mutations and identify which test cases were a mutant killed or an equivalent. As it can be seen from our mutation score, we were able to test 6 different test cases. We were able to identify 85 mutants killed and 17 mutations as equivalent.

# Lessons Learned

Overall as a team, we learned how to write test cases so that our Binary Search Tree program was mutant free. We discovered modifying our source code in units and executing the modified program against our test suite will give us our mutation score. Also, we learned how to derive a “killer” test by adding extra tests so that we can achieve a mutation score that was near 100%.