SOFTWARE TESTING PROJECT REPORT (Mutation Testing on Source Code)

Pranjal Walia (IMT2019062)
Samaksh Dhingra (IMT2019075)

November 28, 2022

Contents

Abstract

This project majorly focuses on the practical aspects of 'mutation testing on a source code' in the field of software testing. We will be looking in detail about the mutation testing and the software tools used to execute it. The source code chosen will also be explained briefly. We then discuss, how the overall work flow to attempt the mutation testing in this project. Then we will be seeing various experiments conducted for designing the test cases. At last, some inferences are drawn from the results obtained.

1 Introduction

Test Driven Development is now a days a very popularly used approach used by software developers for building applications. Mutation testing is one of the most effective techniques existing out there. In the course of software testing, we saw various interesting concepts related to mutation testing done upon source codes, design, documents, etc. However for the purpose of this project, we will be restricting our discussions to mutation testing done upon the source code covering the practical aspects as well. We will be discussing about the testing technique we used in more detail, showcasing our work upon which we applied this technique using various open source tools.

2 Testing technique used: Mutation Testing

We have performed Mutation Testing upon the existing source code (used as part of one of the projects). In mutation testing, the bugs or *mutants* are manually inserted into the production code. Then the set of test cases are run on these mutants. If the test fails, the mutant is said to be *killed*, however, if test is passed the mutant is said to be *survived.*

Mutation testing also helps to test the quality of the defined test cases or the test suites with a bid to write more effective test cases. Generally, the more mutants we can kill, the higher the quality of our tests. But that is not always the case.

Basic concepts related to mutation testing:-

- **Mutants**: It is merely the source code that has been altered. The code itself contains tiny adjustments. Ideally, the mutant should produce results that differ from those of the original source code when the test data is passed through it. Mutant programmes are another name for mutants.

  There are different types of mutants:-

  - *Survived Mutants*: They still pass the test cases.
  - *Killed Mutants*: They fail the test cases.
  - *Equivalent Mutants*: These resemble real mutants in that they continue to function after test data has been passed through them. They differ from others because, although having a different syntax, they have the same meaning as the original source code.

- **Mutators/Mutation Operators**: These are in the 'driver's seat' and they are what allow mutations to occur. In essence, they specify the type of source code modification needed to create a mutant version. They are also known as mutation rules or defects.

- **Mutation Score**: This rating is determined by the quantity of mutations. It is calculated using following formula:-

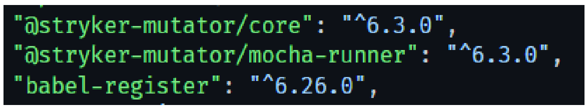
$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants (Survived + Killed)}} \times 100\%$$

3 Software Tools Used

The source code chosen was written in Javascript that runs in a NodeJS runtime. For performing unit testing upon the source code, we used open source frameworks - Mocha – popular testing library and Chai – popular assertions library. The choice for such language was due to its huge popularity in industry in current scenario and it's easy readability of code, hence it would help us to even perform testing in software development projects as well. Mutation Testing was performed by the help of a muatiton testing tool called Stryker (Such a naming convention of this tool itself has a very interesting reference to it). Stryker is the library used for facilitating the automatic mutation of source code. The mutation testing report generated by it is very detailed, hence also helps in test case designing.

Stryker - Mutation Testing Tool For Javascript

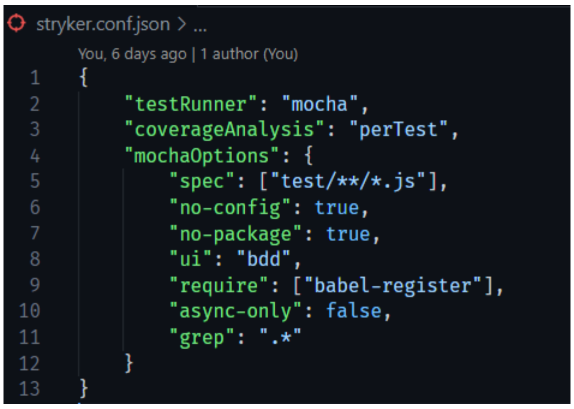
The packages required for succeessfully running Stryker in this project are as follows:



```
"@stryker-mutator/core": "^6.3.0",
"@stryker-mutator/mocha-runner": "^6.3.0",
"babel-register": "^6.26.0",
```

Figure 1: Packages for Adding Stryker to the workspace

Additionally, the configuration file for Stryker is required to specify it's runtime behaviour.



stryker.conf.json > ...

You, 6 days ago | 1 author (You)

```
{
    "testRunner": "mocha",
    "coverageAnalysis": "perTest",
    "mochaOptions": {
        "spec": ["test/**/*.js"],
        "no-config": true,
        "no-package": true,
        "ui": "bdd",
        "require": ["babel-register"],
        "async-only": false,
        "grep": ".*"
    }
}
```

Figure 2: Packages for Adding Stryker to the workspace

(More details about the installation and setting up stryker is provided in the documentation provided along)

Mocha and Chai

Figure 3: Write and Execute Unit Tests For JavaScript

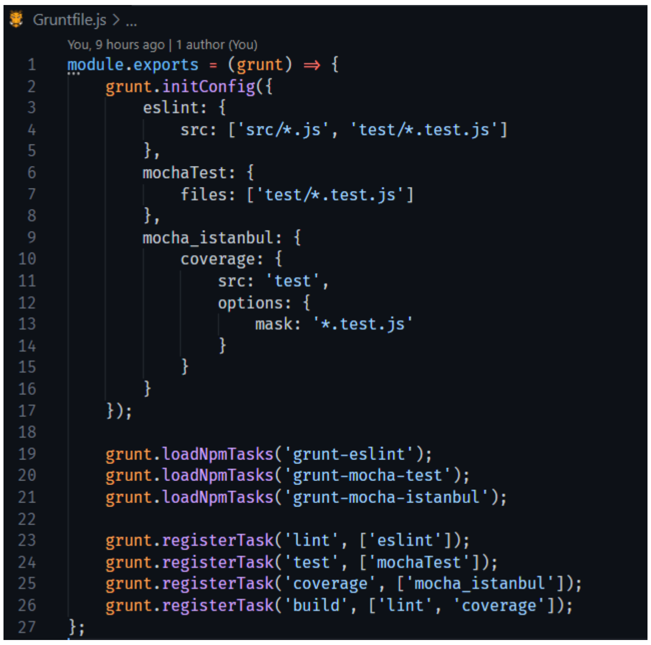
Mocha is a testing library used to provide access to wrappers for unit tests and additionally, chai is used for asserting conditions as per test requirements. These libraries work out of the box and do not require any additional configuration.
Following packages are required for working with these:

1. chai @4.3.7
2. mocha @10.1.0

Grunt

Figure 4: Batch Task Runner For JavaScript

Grunt is a batch task runner that automates the entire workflow of running the unit tests, generation of code coverage, linting and formatting the code base etc. This requires additional configuration as specified in the project root workspace:



```javascript
module.exports = (grunt) => {
    grunt.initConfig({
        eslint: {
            src: ['src/*.js', 'test/*.test.js']
        },
        mochaTest: {
            files: ['test/*.test.js']
        },
        mocha_istanbul: {
            coverage: {
                src: 'test',
                options: {
                    mask: '*.test.js'
                }
            }
        }
    });

    grunt.loadNpmTasks('grunt-eslint');
    grunt.loadNpmTasks('grunt-mocha-test');
    grunt.loadNpmTasks('grunt-mocha-istanbul');

    grunt.registerTask('lint', ['eslint']);
    grunt.registerTask('test', ['mochaTest']);
    grunt.registerTask('coverage', ['mocha_istanbul']);
    grunt.registerTask('build', ['lint', 'coverage']);
};
```

Figure 5: Gruntfile

4 Understanding the source code chosen

The source code we chose was a data structures and algorithms utility library which comprised of four modules - Stacks, Queues, Linked-Lists, Binary Search Tree implemented in Javascript language.

The source code consists of 1000+ lines of code making it a suitable choice being a sufficiently large scale code base. All the best practices were followed at the time the code was written.

The main purpose of choosing this source code was due to its appropriately complex nature as well as not too complex that testing would become difficult for it. Hence, it was an ideal choice for the mutation testing we performed upon it.

More details about the code base have been provided in the documentation file provided along with this report.

5 Workflow followed for Mutation Testing

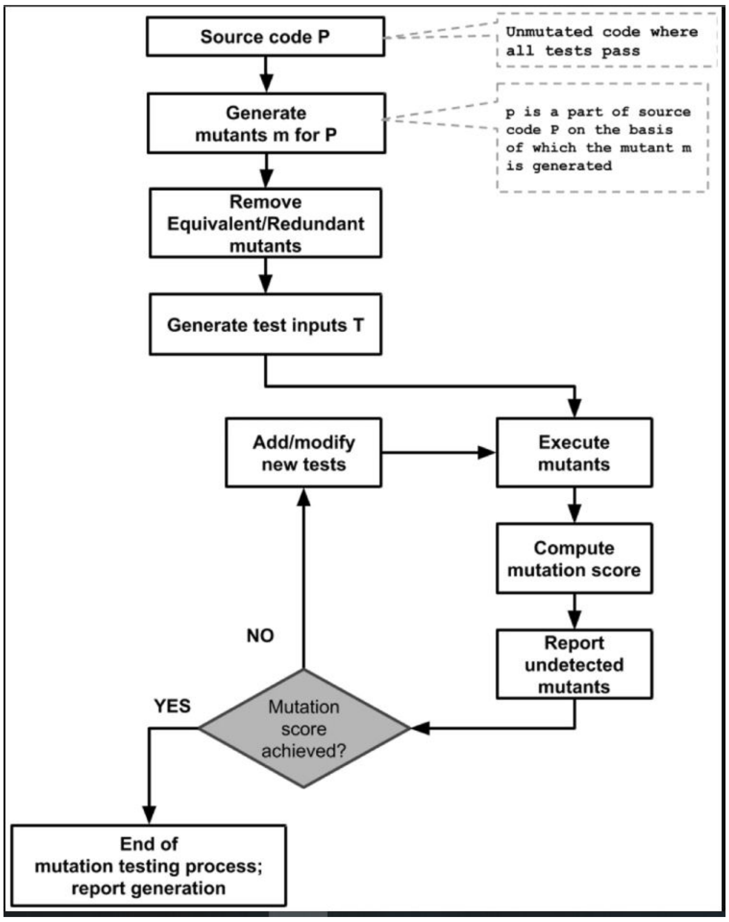

Source code P
Unmutated code where all tests pass
Generate mutants m for P
p is a part of source code P on the basis of which the mutant m is generated
Remove Equivalent/Redundant mutants
Generate test inputs T
Add/modify new tests
Execute mutants
Compute mutation score
Report undetected mutants
NO
YES
Mutation score achieved?
End of mutation testing process; report generation

Figure 6: Mutation Testing Workflow Diagram (Source: [4])

Basic workflow that we followed to perform mutation testing on our javascript source code using stryker is as follows:-

1. Initially, we designed some simple testcases for every relevant function and tried to make the code coverage as high as possible. This was done majorly by thinking by self about various ways to cover every segment of code in a function and also took some help from the code coverage metric results generated by *Mocha* library.

2. Then with the help of *Stryker*, mutants were generated from the source code based upon the mutant operators which were feasible and enabled by us.

3. Then the same test cases which were passed initially, were given as an input while running the mutation tester, based upon which some mutants were killed and some survived.
4. Based upon this, *stryker* generated a detailed report as well as overall *mutation score* for the source code corresponding to given input test cases.
5. The survived mutants were then analyzed and based upon that we got to see some boundary test cases that got missed initially and the new test cases were added appropriately or existing ones were modified for more robust testing.
6. After repeating the above steps for some cycles, we stopped when the mutants generated with sufficient number of mutant operators were *strongly killed* by designed test cases.

6 Experiments

Designing Test Cases

Coming up with effective test cases was overall an incremental process. Initially, the test cases were written such that maximum code coverage could be achieved. For example, if a function contains an if-else statement, we cover test cases such that both flows of control can be tested through the designed test cases. After initial test cases were set up, the mutation tester was runned upon those giving the overall mutation score as an output and a detailed report which showed which mutant survived. According to that, we could see what more test cases could be added to cover the killing of such mutant. This included adding boundary value test cases, Null cases, etc. This incremental process helped to make the test cases more effective.

Mutation Score

Although, it is generally said that the mutation score is correlated to the quality of test cases, but it is not always true. For example, a code that is very complex or calculates some values randomly in its internal working, this would majority of the times give different results for second iteration over any mutated code. Hence, main aim to improve the quality of test cases was not increasing mutation score blindly. It required some diligence to understand what changes were feasible to improve the mutation score. Many different things were explored to study the variations of mutations scores. One of them was to enforce only specific mutators that were relevant. Also some efforts were made to choose parts of code which should not be mutated. For example, if the part of code has some equivalent mutant, that can be discarded from mutation. StrykerJS provides the functionality to change the configuration according to fulfil such needs of a tester. However, keeping in mind the future scope of the functionality used in the code, it did not seem feasible to discard those pieces of code from mutation. Finally, after collective efforts of experimenting different things, we decided to stick to the default configuration only.

7 Results and Inferences

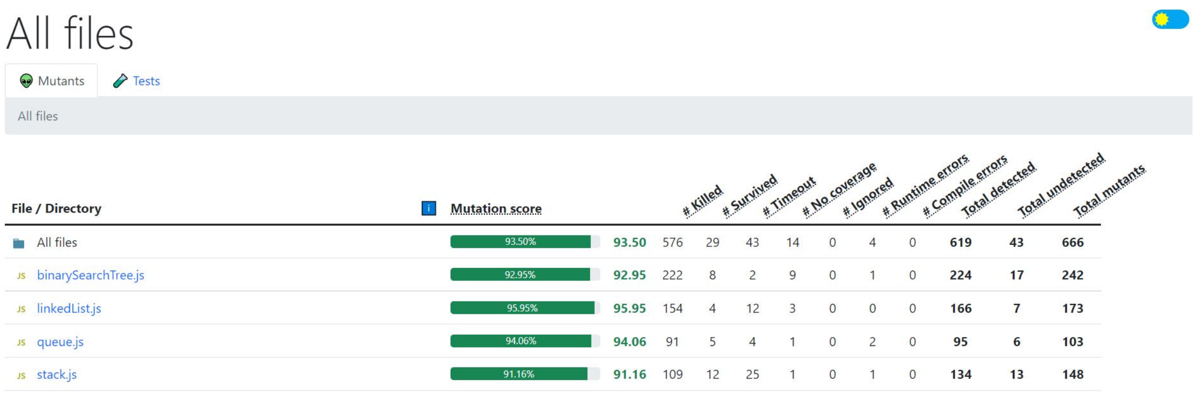
7.1 Mutation Operators

The mutation operators which were used to run the mutation tests on overall source code were as follows:-

- Arithmetic Operator
- Array Declaration
- Block Statement
- Boolean Literal
- Conditional Expression
- Equality Operator
- Logical Operator
- Method Expression
- String Literal
- Unary Operator

After running mutation tests using *stryker* on the finally designed test cases we got following results:-

Overall Source Code Report



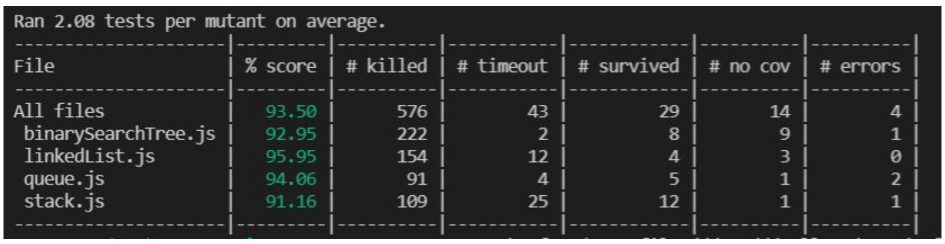
All files

Mutants | Tests

All files

File / Directory	Mutation score		# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	93.50%	93.50	576	29	43	14	0	4	0	619	43	666
binarySearchTree.js	92.95%	92.95	222	8	2	9	0	1	0	224	17	242
linkedList.js	95.95%	95.95	154	4	12	3	0	0	0	166	7	173
queue.js	94.06%	94.06	91	5	4	1	0	2	0	95	6	103
stack.js	91.16%	91.16	109	12	25	1	0	1	0	134	13	148

Figure 7: Overall Performance Report - Mutation Testing

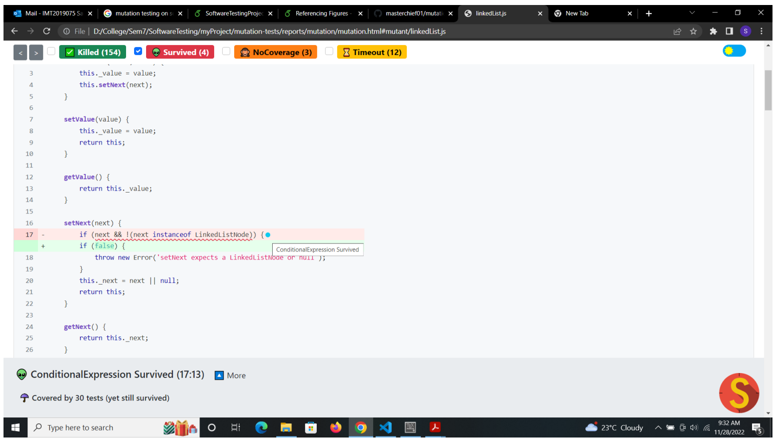


```
Ran 2.08 tests per mutant on average.
--------------------|---------|----------|-----------|------------|----------|----------|
File                | % score | # killed | # timeout | # survived | # no cov | # errors |
--------------------|---------|----------|-----------|------------|----------|----------|
All files           |   93.50 |      576 |        43 |         29 |       14 |        4 |
 binarySearchTree.js |   92.95 |      222 |         2 |          8 |        9 |        1 |
 linkedList.js      |   95.95 |      154 |        12 |          4 |        3 |        0 |
 queue.js           |   94.06 |       91 |         4 |          5 |        1 |        2 |
 stack.js           |   91.16 |      109 |        25 |         12 |        1 |        1 |
--------------------|---------|----------|-----------|------------|----------|----------|
```

Figure 8: Overall Performance Report (Terminal) - Mutation Testing

Following were some of the Survived Mutants in the Source codes for respective modules:-

<u>Linked List</u>

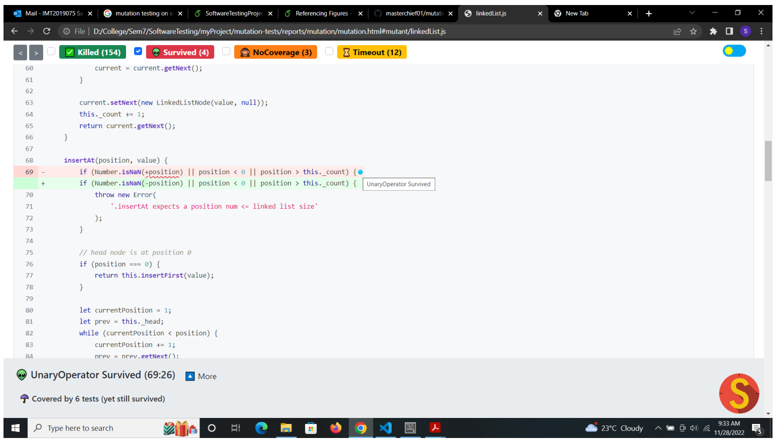


```
        this._value = value;
        this.setNext(next);
    }

    setValue(value) {
        this._value = value;
        return this;
    }

    getValue() {
        return this._value;
    }

    setNext(next) {
-       if (next && !(next instanceof LinkedListNode)) {
+       if (false) {                                        ConditionalExpression Survived
            throw new Error('setNext expects a LinkedListNode or null');
        }
        this._next = next || null;
        return this;
    }

    getNext() {
        return this._next;
    }
```

ConditionalExpression Survived (17:13) More

Covered by 30 tests (yet still survived)

Figure 9: Mutant Survived (Conditional Expression) by Linked List

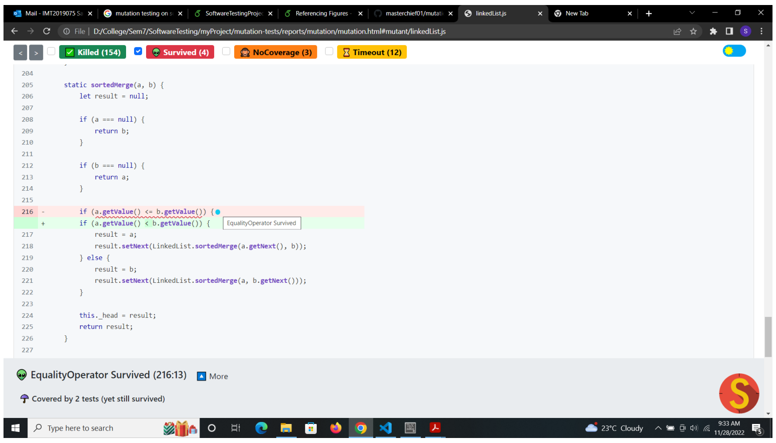


```
            current = current.getNext();
        }

        current.setNext(new LinkedListNode(value, null));
        this._count += 1;
        return current.getNext();
    }

    insertAt(position, value) {
-       if (Number.isNaN(+position) || position < 0 || position > this._count) {
+       if (Number.isNaN(-position) || position < 0 || position > this._count) {   UnaryOperator Survived
            throw new Error(
                '.insertAt expects a position num <= linked list size'
            );
        }

        // head node is at position 0
        if (position === 0) {
            return this.insertFirst(value);
        }

        let currentPosition = 1;
        let prev = this._head;
        while (currentPosition < position) {
            currentPosition += 1;
            prev = prev.getNext();
```

UnaryOperator Survived (69:26) More

Covered by 6 tests (yet still survived)

Figure 10: Mutant Survived (Unary Operator) by Linked List



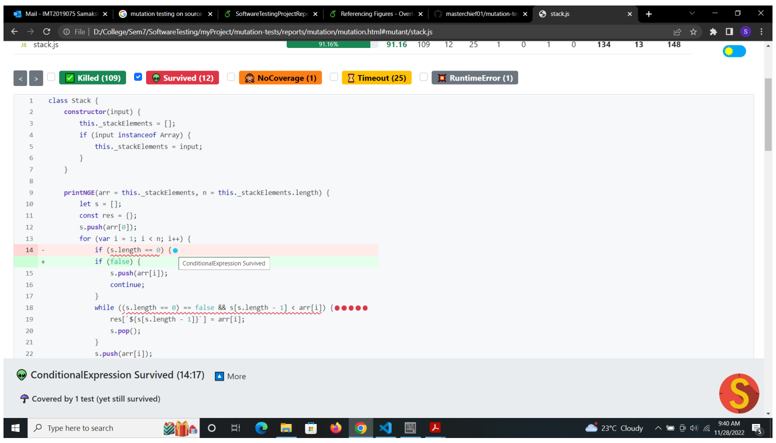
```

    static sortedMerge(a, b) {
        let result = null;

        if (a === null) {
            return b;
        }

        if (b === null) {
            return a;
        }

-       if (a.getValue() <= b.getValue()) {
+       if (a.getValue() < b.getValue()) {      EqualityOperator Survived
            result = a;
            result.setNext(LinkedList.sortedMerge(a.getNext(), b));
        } else {
            result = b;
            result.setNext(LinkedList.sortedMerge(a, b.getNext()));
        }

        this._head = result;
        return result;
    }

```

EqualityOperator Survived (216:13) More

Covered by 2 tests (yet still survived)

Figure 11: Mutant Survived (Equality Operator) by Linked List

Stacks

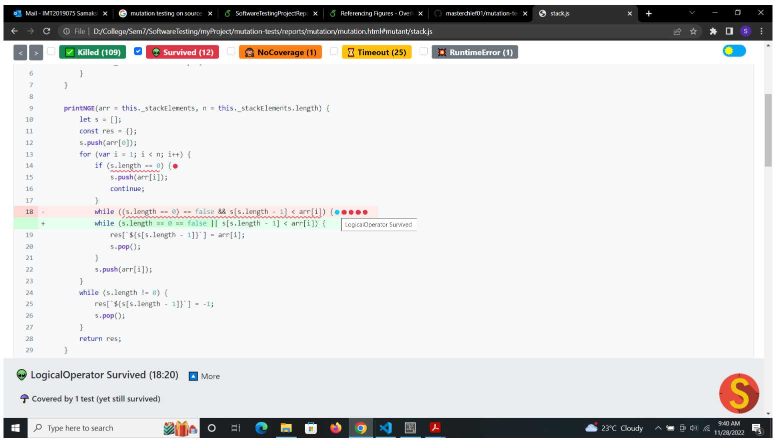


```
class Stack {
    constructor(input) {
        this._stackElements = [];
        if (input instanceof Array) {
            this._stackElements = input;
        }
    }

    printNGE(arr = this._stackElements, n = this._stackElements.length) {
        let s = [];
        const res = {};
        s.push(arr[0]);
        for (var i = 1; i < n; i++) {
-           if (s.length == 0) {
+           if (false) {                ConditionalExpression Survived
                s.push(arr[i]);
                continue;
            }
            while ((s.length == 0) == false && s[s.length - 1] < arr[i]) {
                res[`${s[s.length - 1]}`] = arr[i];
                s.pop();
            }
            s.push(arr[i]);
```

ConditionalExpression Survived (14:17) More

Covered by 1 test (yet still survived)

Figure 12: Mutant Survived (Conditional Expression) by Stacks

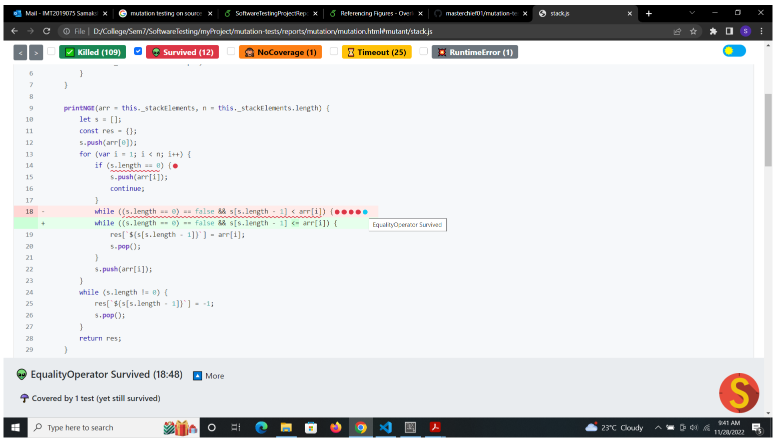


```
        }
    }

    printNGE(arr = this._stackElements, n = this._stackElements.length) {
        let s = [];
        const res = {};
        s.push(arr[0]);
        for (var i = 1; i < n; i++) {
            if (s.length == 0) {
                s.push(arr[i]);
                continue;
            }
-           while ((s.length == 0) == false && s[s.length - 1] < arr[i]) {
+           while (s.length == 0 == false || s[s.length - 1] < arr[i]) {      LogicalOperator Survived
                res[`${s[s.length - 1]}`] = arr[i];
                s.pop();
            }
            s.push(arr[i]);
        }
        while (s.length != 0) {
            res[`${s[s.length - 1]}`] = -1;
            s.pop();
        }
        return res;
    }
```

LogicalOperator Survived (18:20) More

Covered by 1 test (yet still survived)

Figure 13: Mutant Survived (Logical Operator) by Stacks



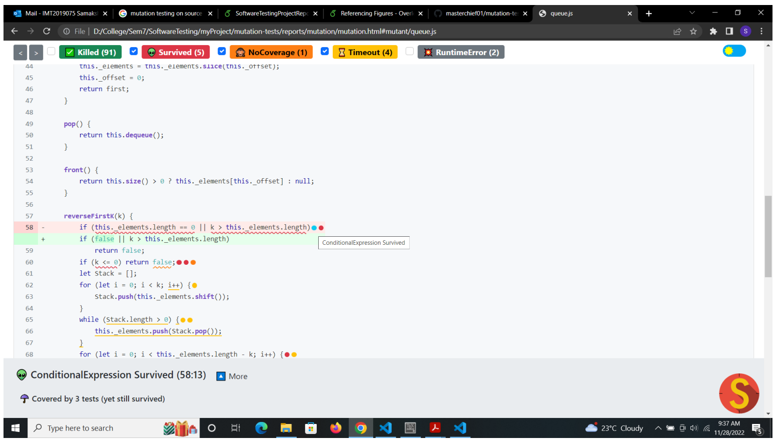
```
        }
    }

    printNGE(arr = this._stackElements, n = this._stackElements.length) {
        let s = [];
        const res = {};
        s.push(arr[0]);
        for (var i = 1; i < n; i++) {
            if (s.length == 0) {
                s.push(arr[i]);
                continue;
            }
-           while ((s.length == 0) == false && s[s.length - 1] < arr[i]) {
+           while ((s.length == 0) == false && s[s.length - 1] <= arr[i]) {      EqualityOperator Survived
                res[`${s[s.length - 1]}`] = arr[i];
                s.pop();
            }
            s.push(arr[i]);
        }
        while (s.length != 0) {
            res[`${s[s.length - 1]}`] = -1;
            s.pop();
        }
        return res;
    }
```

EqualityOperator Survived (18:48) More

Covered by 1 test (yet still survived)

Figure 14: Mutant Survived (Equality Operator) by Stacks

Queues



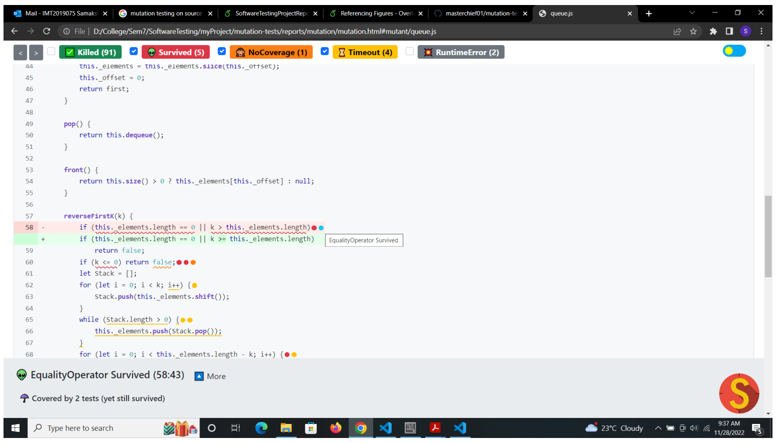
```
        this._elements = this._elements.slice(this._offset);
        this._offset = 0;
        return first;
    }

    pop() {
        return this.dequeue();
    }

    front() {
        return this.size() > 0 ? this._elements[this._offset] : null;
    }

    reverseFirstK(k) {
-       if (this._elements.length == 0 || k > this._elements.length)
+       if (false || k > this._elements.length)
            return false;
        if (k <= 0) return false;
        let Stack = [];
        for (let i = 0; i < k; i++) {
            Stack.push(this._elements.shift());
        }
        while (Stack.length > 0) {
            this._elements.push(Stack.pop());
        }
        for (let i = 0; i < this._elements.length - k; i++) {
```

ConditionalExpression Survived

ConditionalExpression Survived (58:13) More

Covered by 3 tests (yet still survived)

Figure 15: Mutant Survived (Conditional Expression) by Queues



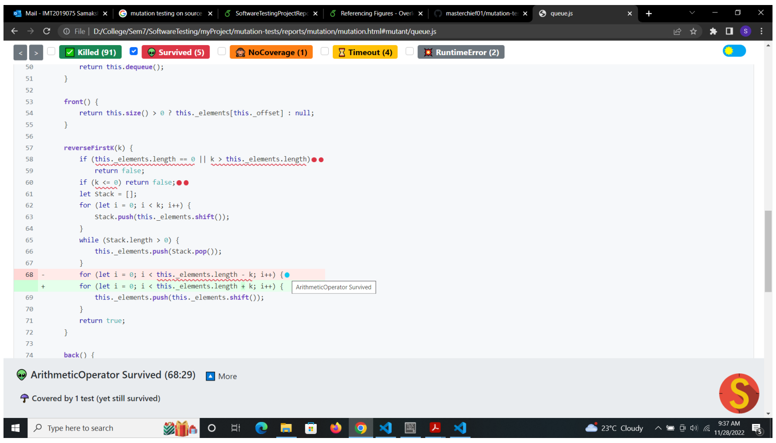
```
        this._elements = this._elements.slice(this._offset);
        this._offset = 0;
        return first;
    }

    pop() {
        return this.dequeue();
    }

    front() {
        return this.size() > 0 ? this._elements[this._offset] : null;
    }

    reverseFirstK(k) {
-       if (this._elements.length == 0 || k > this._elements.length)
+       if (this._elements.length == 0 || k >= this._elements.length)
            return false;
        if (k <= 0) return false;
        let Stack = [];
        for (let i = 0; i < k; i++) {
            Stack.push(this._elements.shift());
        }
        while (Stack.length > 0) {
            this._elements.push(Stack.pop());
        }
        for (let i = 0; i < this._elements.length - k; i++) {
```

EqualityOperator Survived

EqualityOperator Survived (58:43) More

Covered by 2 tests (yet still survived)

Figure 16: Mutant Survived (Equality Operator) by Queues



```
        return this.dequeue();
    }

    front() {
        return this.size() > 0 ? this._elements[this._offset] : null;
    }

    reverseFirstK(k) {
        if (this._elements.length == 0 || k > this._elements.length)
            return false;
        if (k <= 0) return false;
        let Stack = [];
        for (let i = 0; i < k; i++) {
            Stack.push(this._elements.shift());
        }
        while (Stack.length > 0) {
            this._elements.push(Stack.pop());
        }
-       for (let i = 0; i < this._elements.length - k; i++) {
+       for (let i = 0; i < this._elements.length + k; i++) {
            this._elements.push(this._elements.shift());
        }
        return true;
    }

    back() {
```

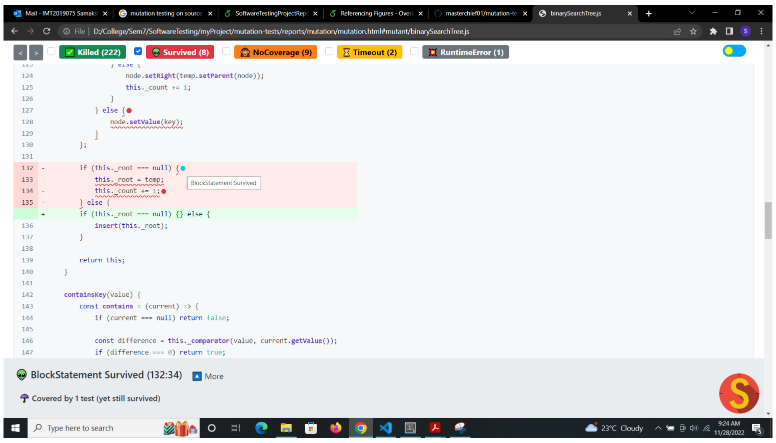
ArithmeticOperator Survived

ArithmeticOperator Survived (68:29) More

Covered by 1 test (yet still survived)

Figure 17: Mutant Survived (Arithmetic Operator) by Queues

Binary Search Trees

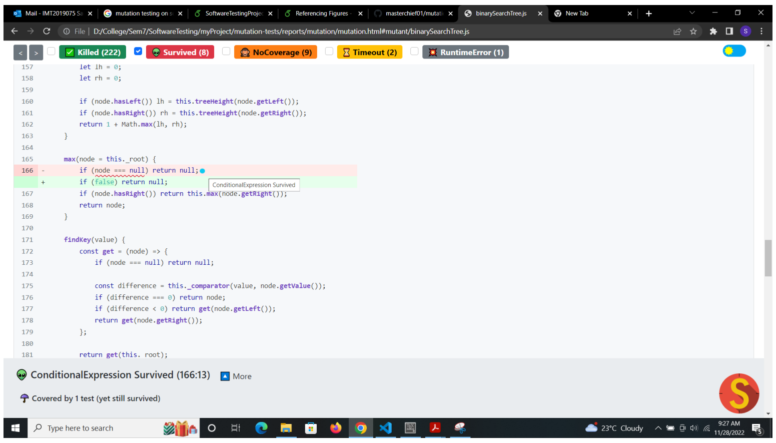


```
                } else {
                    node.setRight(temp.setParent(node));
                    this._count += 1;
                }
            } else {
                node.setValue(key);
            }
        };

-       if (this._root === null) {
-           this._root = temp;
-           this._count += 1;
-       } else {
+       if (this._root === null) {} else {
            insert(this._root);
        }

        return this;
    }

    containsKey(value) {
        const contains = (current) => {
            if (current === null) return false;

            const difference = this._comparator(value, current.getValue());
            if (difference === 0) return true;
```

BlockStatement Survived (132:34) More

Covered by 1 test (yet still survived)

Figure 18: Mutant Survived (Block Statement) by Binary Search Trees

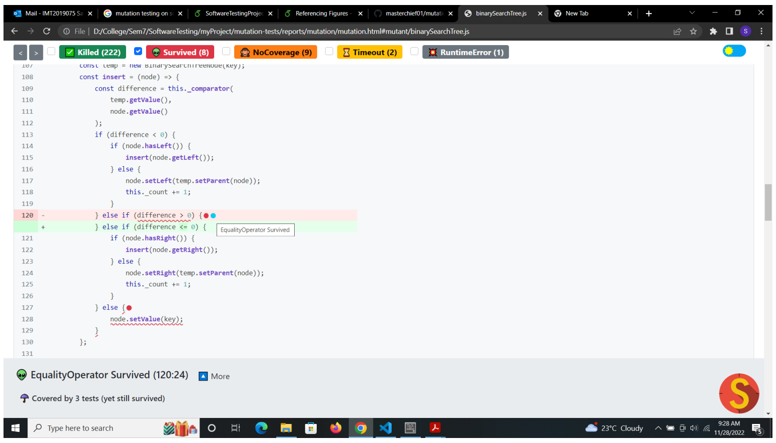


```
        let lh = 0;
        let rh = 0;

        if (node.hasLeft()) lh = this.treeHeight(node.getLeft());
        if (node.hasRight()) rh = this.treeHeight(node.getRight());
        return 1 + Math.max(lh, rh);
    }

    max(node = this._root) {
-       if (node === null) return null;
+       if (false) return null;
        if (node.hasRight()) return this.max(node.getRight());
        return node;
    }

    findKey(value) {
        const get = (node) => {
            if (node === null) return null;

            const difference = this._comparator(value, node.getValue());
            if (difference === 0) return node;
            if (difference < 0) return get(node.getLeft());
            return get(node.getRight());
        };

        return get(this._root);
```

ConditionalExpression Survived (166:13) More

Covered by 1 test (yet still survived)

Figure 19: Mutant Survived (Conditional Expression) by Binary Search Trees



```
        const temp = new BinarySearchTreeNode(key);
        const insert = (node) => {
            const difference = this._comparator(
                temp.getValue(),
                node.getValue()
            );
            if (difference < 0) {
                if (node.hasLeft()) {
                    insert(node.getLeft());
                } else {
                    node.setLeft(temp.setParent(node));
                    this._count += 1;
                }
-           } else if (difference > 0) {
+           } else if (difference <= 0) {
                if (node.hasRight()) {
                    insert(node.getRight());
                } else {
                    node.setRight(temp.setParent(node));
                    this._count += 1;
                }
            } else {
                node.setValue(key);
            }
        };
```

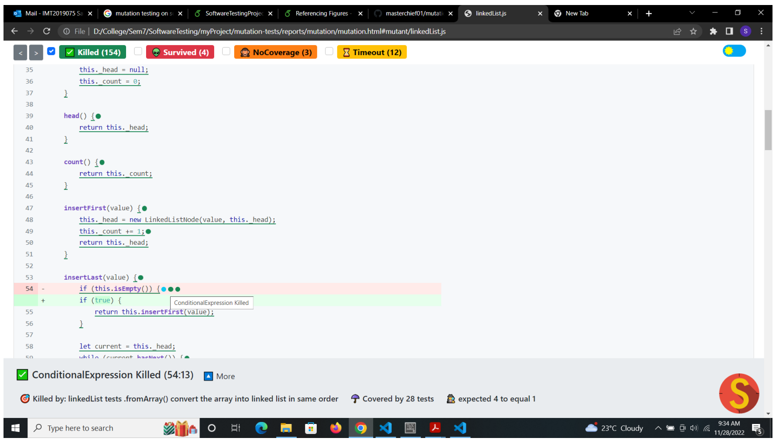
EqualityOperator Survived (120:24) More

Covered by 3 tests (yet still survived)

Figure 20: Mutant Survived (Equality Operator) by Binary Search Trees

Following were some of the Killed Mutants in the Source codes for respective modules:-

<u>Linked List</u>



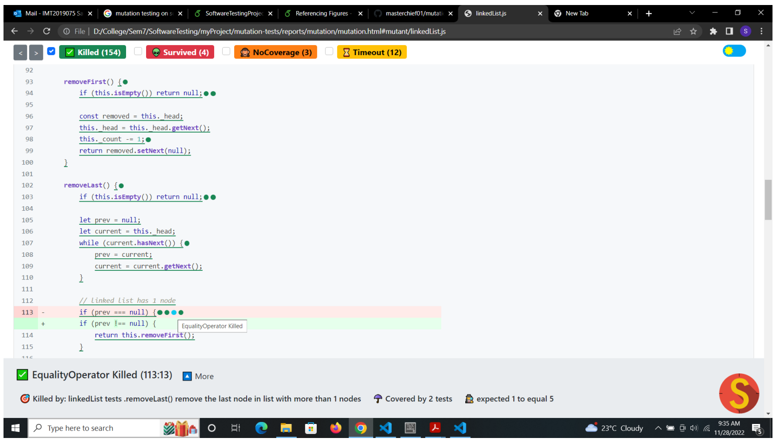
Killed (154) | Survived (4) | NoCoverage (3) | Timeout (12)

```
        this._head = null;
        this._count = 0;
    }

    head() {●
        return this._head;
    }

    count() {●
        return this._count;
    }

    insertFirst(value) {●
        this._head = new LinkedListNode(value, this._head);
        this._count += 1;●
        return this._head;
    }

    insertLast(value) {●
-       if (this.isEmpty()) {●●●
+       if (true) {          ConditionalExpression Killed
            return this.insertFirst(value);
        }

        let current = this._head;
```

ConditionalExpression Killed (54:13) More

Killed by: linkedList tests .fromArray() convert the array into linked list in same order | Covered by 28 tests | expected 4 to equal 1

Figure 21: Mutant Killed (Conditional Expression) by Linked List



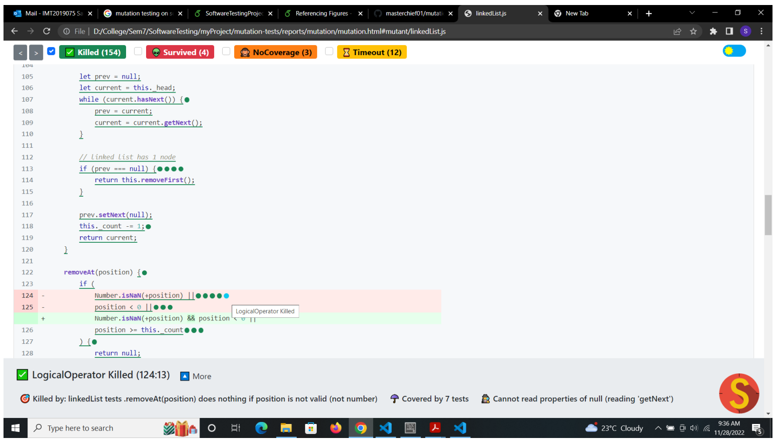
Killed (154) | Survived (4) | NoCoverage (3) | Timeout (12)

```
    removeFirst() {●
        if (this.isEmpty()) return null;●●

        const removed = this._head;
        this._head = this._head.getNext();
        this._count -= 1;●
        return removed.setNext(null);
    }

    removeLast() {●
        if (this.isEmpty()) return null;●●

        let prev = null;
        let current = this._head;
        while (current.hasNext()) {●
            prev = current;
            current = current.getNext();
        }

        // linked list has 1 node
-       if (prev === null) {●●●●
+       if (prev !== null) {          EqualityOperator Killed
            return this.removeFirst();
        }
```

EqualityOperator Killed (113:13) More

Killed by: linkedList tests .removeLast() remove the last node in list with more than 1 nodes | Covered by 2 tests | expected 1 to equal 5

Figure 22: Mutant Killed (Equality Operator) by Linked List



Killed (154) | Survived (4) | NoCoverage (3) | Timeout (12)

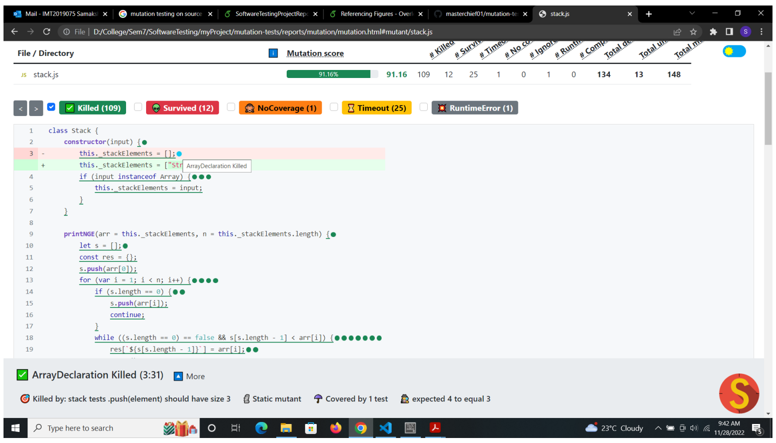
```
        let prev = null;
        let current = this._head;
        while (current.hasNext()) {●
            prev = current;
            current = current.getNext();
        }

        // linked list has 1 node
        if (prev === null) {●●●●
            return this.removeFirst();
        }

        prev.setNext(null);
        this._count -= 1;●
        return current;
    }

    removeAt(position) {●
        if (
-           Number.isNaN(+position) ||●●●●●
-           position < 0 ||●●●
+           Number.isNaN(+position) && position < 0 ||          LogicalOperator Killed
            position >= this._count●●●
        ) {●
            return null;
```

LogicalOperator Killed (124:13) More

Killed by: linkedList tests .removeAt(position) does nothing if position is not valid (not number) | Covered by 7 tests | Cannot read properties of null (reading 'getNext')

Figure 23: Mutant Killed (Logical Operator) by Linked List

Stacks



Mail - IMT2019075 Samaks
mutation testing on source
SoftwareTestingProjectRep
Referencing Figures - Overl
masterchief01/mutation-te
stack.js
File | D:/College/Sem7/SoftwareTesting/myProject/mutation-tests/reports/mutation/mutation.html#mutant/stack.js
Killed (109)
Survived (12)
NoCoverage (1)
Timeout (25)
RuntimeError (1)
More
Type here to search
23°C Cloudy
9:42 AM
11/28/2022

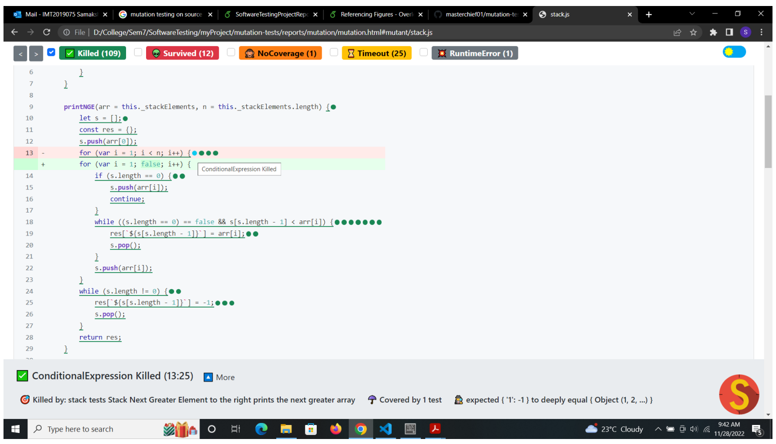
File / Directory	Mutation score		# Killed	# Survi	# Timeo	# No co	# Ignor	# Runti	# Comp	Total de	Total un	Total m
stack.js	91.16%	91.16	109	12	25	1	0	1	0	134	13	148

```
class Stack {
    constructor(input) {
-       this._stackElements = [];
+       this._stackElements = ["Str    ArrayDeclaration Killed
        if (input instanceof Array) {
            this._stackElements = input;
        }
    }

    printNGE(arr = this._stackElements, n = this._stackElements.length) {
        let s = [];
        const res = {};
        s.push(arr[0]);
        for (var i = 1; i < n; i++) {
            if (s.length == 0) {
                s.push(arr[i]);
                continue;
            }
            while ((s.length == 0) == false && s[s.length - 1] < arr[i]) {
                res[`${s[s.length - 1]}`] = arr[i];
```

ArrayDeclaration Killed (3:31)

Killed by: stack tests .push(element) should have size 3 · Static mutant · Covered by 1 test · expected 4 to equal 3

Figure 24: Mutant Killed (Array Declaration) by Stacks



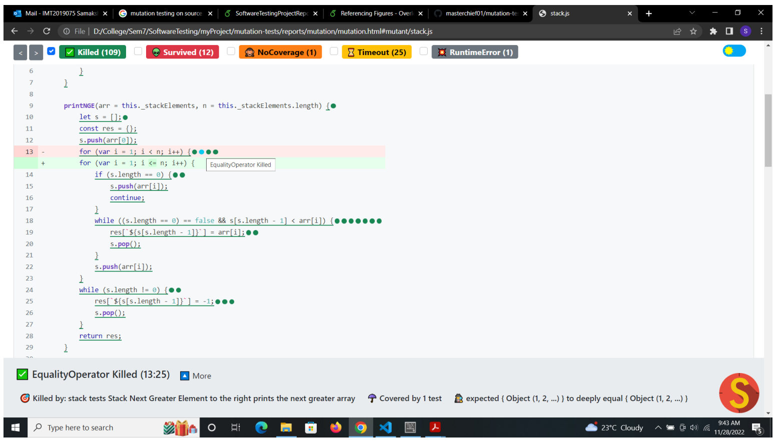
Mail - IMT2019075 Samaks
mutation testing on source
SoftwareTestingProjectRep
Referencing Figures - Overl
masterchief01/mutation-te
stack.js
File | D:/College/Sem7/SoftwareTesting/myProject/mutation-tests/reports/mutation/mutation.html#mutant/stack.js
Killed (109)
Survived (12)
NoCoverage (1)
Timeout (25)
RuntimeError (1)
More
Type here to search
23°C Cloudy
9:42 AM
11/28/2022

```
        }
    }

    printNGE(arr = this._stackElements, n = this._stackElements.length) {
        let s = [];
        const res = {};
        s.push(arr[0]);
-       for (var i = 1; i < n; i++) {
+       for (var i = 1; false; i++) {    ConditionalExpression Killed
            if (s.length == 0) {
                s.push(arr[i]);
                continue;
            }
            while ((s.length == 0) == false && s[s.length - 1] < arr[i]) {
                res[`${s[s.length - 1]}`] = arr[i];
                s.pop();
            }
            s.push(arr[i]);
        }
        while (s.length != 0) {
            res[`${s[s.length - 1]}`] = -1;
            s.pop();
        }
        return res;
    }
```

ConditionalExpression Killed (13:25)

Killed by: stack tests Stack Next Greater Element to the right prints the next greater array · Covered by 1 test · expected { '1': -1 } to deeply equal { Object (1, 2, ...) }

Figure 25: Mutant Killed (Conditional Expression) by Stacks



Mail - IMT2019075 Samaks
mutation testing on source
SoftwareTestingProjectRep
Referencing Figures - Overl
masterchief01/mutation-te
stack.js
File | D:/College/Sem7/SoftwareTesting/myProject/mutation-tests/reports/mutation/mutation.html#mutant/stack.js
Killed (109)
Survived (12)
NoCoverage (1)
Timeout (25)
RuntimeError (1)
More
Type here to search
23°C Cloudy
9:43 AM
11/28/2022

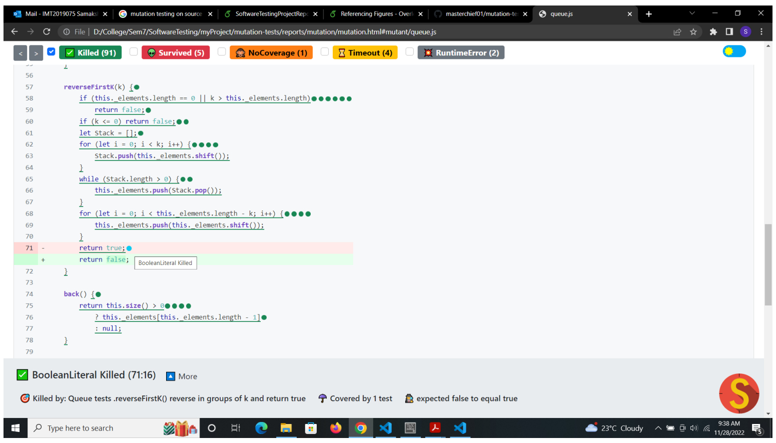
```
        }
    }

    printNGE(arr = this._stackElements, n = this._stackElements.length) {
        let s = [];
        const res = {};
        s.push(arr[0]);
-       for (var i = 1; i < n; i++) {
+       for (var i = 1; i <= n; i++) {    EqualityOperator Killed
            if (s.length == 0) {
                s.push(arr[i]);
                continue;
            }
            while ((s.length == 0) == false && s[s.length - 1] < arr[i]) {
                res[`${s[s.length - 1]}`] = arr[i];
                s.pop();
            }
            s.push(arr[i]);
        }
        while (s.length != 0) {
            res[`${s[s.length - 1]}`] = -1;
            s.pop();
        }
        return res;
    }
```

EqualityOperator Killed (13:25)

Killed by: stack tests Stack Next Greater Element to the right prints the next greater array · Covered by 1 test · expected { Object (1, 2, ...) } to deeply equal { Object (1, 2, ...) }

Figure 26: Mutant Killed (Equality Operator) by Stacks

Queues

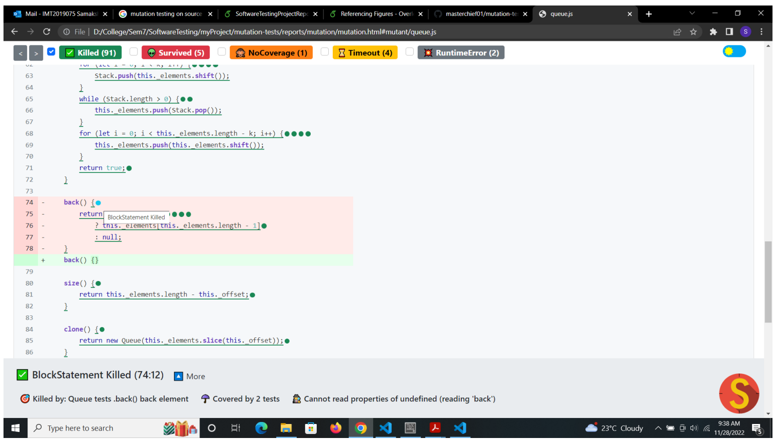


```
    reverseFirstK(k) {
        if (this._elements.length == 0 || k > this._elements.length)
            return false;
        if (k <= 0) return false;
        let Stack = [];
        for (let i = 0; i < k; i++) {
            Stack.push(this._elements.shift());
        }
        while (Stack.length > 0) {
            this._elements.push(Stack.pop());
        }
        for (let i = 0; i < this._elements.length - k; i++) {
            this._elements.push(this._elements.shift());
        }
-       return true;
+       return false;   BooleanLiteral Killed
    }

    back() {
        return this.size() > 0
            ? this._elements[this._elements.length - 1]
            : null;
    }
```

BooleanLiteral Killed (71:16) More

Killed by: Queue tests .reverseFirstK() reverse in groups of k and return true   Covered by 1 test   expected false to equal true

Figure 27: Mutant Killed (Boolean Literal) by Queues

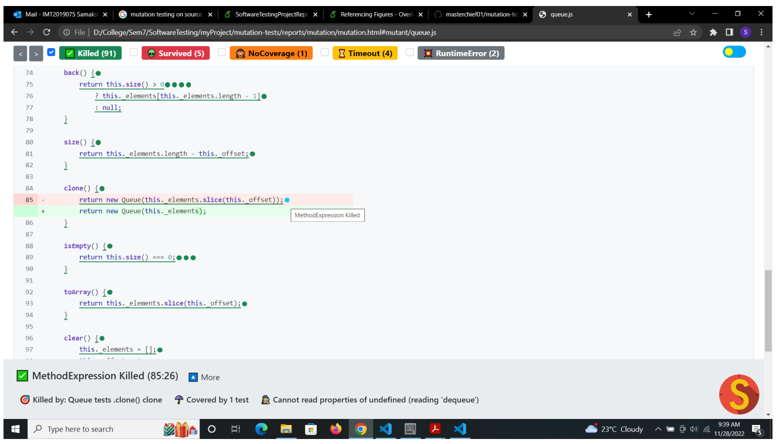


```
            Stack.push(this._elements.shift());
        }
        while (Stack.length > 0) {
            this._elements.push(Stack.pop());
        }
        for (let i = 0; i < this._elements.length - k; i++) {
            this._elements.push(this._elements.shift());
        }
        return true;
    }

-   back() {
-       return   BlockStatement Killed
-           ? this._elements[this._elements.length - 1]
-           : null;
-   }
+   back() {}

    size() {
        return this._elements.length - this._offset;
    }

    clone() {
        return new Queue(this._elements.slice(this._offset));
    }
```

BlockStatement Killed (74:12) More

Killed by: Queue tests .back() back element   Covered by 2 tests   Cannot read properties of undefined (reading 'back')

Figure 28: Mutant Killed (Block Statement) by Queues



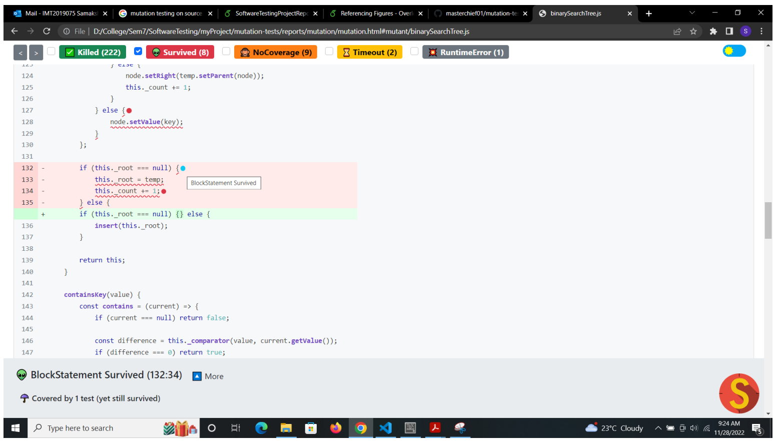
```
    back() {
        return this.size() > 0
            ? this._elements[this._elements.length - 1]
            : null;
    }

    size() {
        return this._elements.length - this._offset;
    }

    clone() {
-       return new Queue(this._elements.slice(this._offset));
+       return new Queue(this._elements);   MethodExpression Killed
    }

    isEmpty() {
        return this.size() === 0;
    }

    toArray() {
        return this._elements.slice(this._offset);
    }

    clear() {
        this._elements = [];
```

MethodExpression Killed (85:26) More

Killed by: Queue tests .clone() clone   Covered by 1 test   Cannot read properties of undefined (reading 'dequeue')

Figure 29: Mutant Killed (Method Expression) by Queues

Binary Search Trees



Killed (222) Survived (8) NoCoverage (9) Timeout (2) RuntimeError (1)

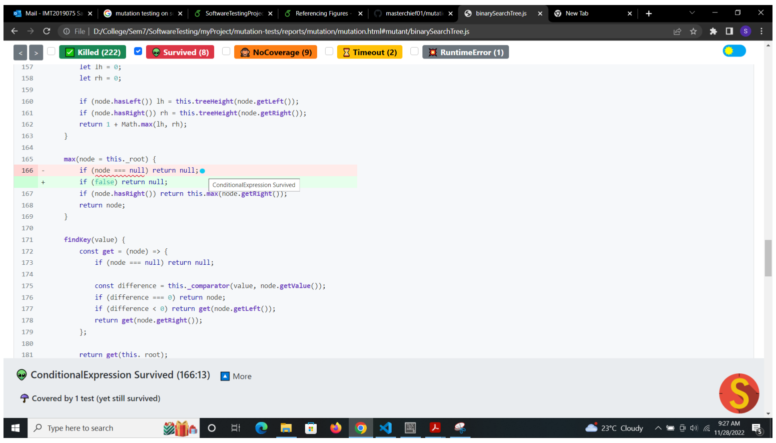
```
                } else {
                    node.setRight(temp.setParent(node));
                    this._count += 1;
                }
            } else {
                node.setValue(key);
            }
        };

-       if (this._root === null) {
-           this._root = temp;
-           this._count += 1;
-       } else {
+       if (this._root === null) {} else {
            insert(this._root);
        }

        return this;
    }

    containsKey(value) {
        const contains = (current) => {
            if (current === null) return false;

            const difference = this._comparator(value, current.getValue());
            if (difference === 0) return true;
```

BlockStatement Survived

BlockStatement Survived (132:34) More

Covered by 1 test (yet still survived)

Figure 30: Mutant Killed (Block Statement) by Binary Search Tree



Killed (222) Survived (8) NoCoverage (9) Timeout (2) RuntimeError (1)

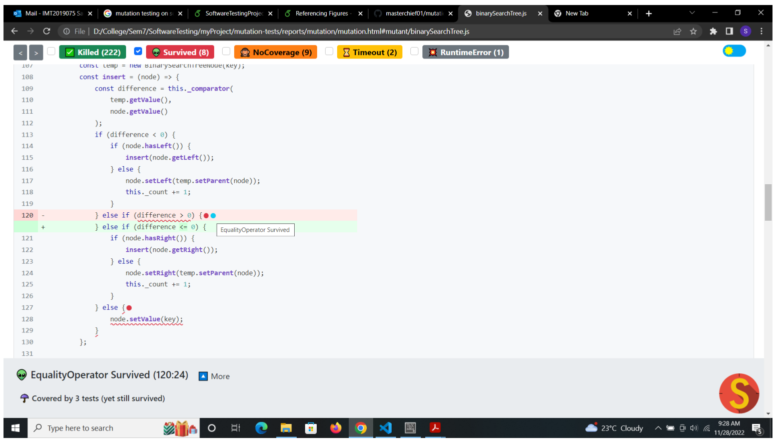
```
        let lh = 0;
        let rh = 0;

        if (node.hasLeft()) lh = this.treeHeight(node.getLeft());
        if (node.hasRight()) rh = this.treeHeight(node.getRight());
        return 1 + Math.max(lh, rh);
    }

    max(node = this._root) {
-       if (node === null) return null;
+       if (false) return null;
        if (node.hasRight()) return this.max(node.getRight());
        return node;
    }

    findKey(value) {
        const get = (node) => {
            if (node === null) return null;

            const difference = this._comparator(value, node.getValue());
            if (difference === 0) return node;
            if (difference < 0) return get(node.getLeft());
            return get(node.getRight());
        };

        return get(this._root);
```

ConditionalExpression Survived

ConditionalExpression Survived (166:13) More

Covered by 1 test (yet still survived)

Figure 31: Mutant Killed (Conditional Expression) by Binary Search Tree



Killed (222) Survived (8) NoCoverage (9) Timeout (2) RuntimeError (1)

```
        const temp = new BinarySearchTreeNode(key);
        const insert = (node) => {
            const difference = this._comparator(
                temp.getValue(),
                node.getValue()
            );
            if (difference < 0) {
                if (node.hasLeft()) {
                    insert(node.getLeft());
                } else {
                    node.setLeft(temp.setParent(node));
                    this._count += 1;
                }
-           } else if (difference > 0) {
+           } else if (difference <= 0) {
                if (node.hasRight()) {
                    insert(node.getRight());
                } else {
                    node.setRight(temp.setParent(node));
                    this._count += 1;
                }
            } else {
                node.setValue(key);
            }
        };
```

EqualityOperator Survived

EqualityOperator Survived (120:24) More

Covered by 3 tests (yet still survived)

Figure 32: Mutant Killed (Equality Operator) by Binary Search Tree

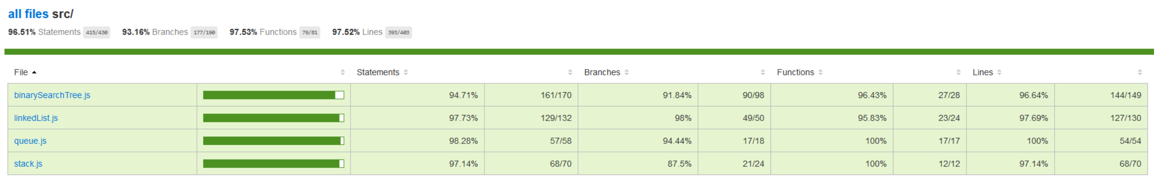
The *Overall Mutation Coverage* Score obtained is **93.5**%. Hence we can say that the mutants obtained from source code were strongly killed by the designed test cases. Hence from general observations, we can infer that the test cases designed are strong enough to test the code properly.

Let us try to infer some results with the example of source code of Linked Lists. Let us try to see each survived mutant and reason out how fixing code based upon it is not possible further.

- In Figure 4, we can see that the mutant is based upon the mutator - Conditional Expression, where we see that the statement highlighted with red never gets true for any test case. There is no feasible fix present for this, as all the functions that are setting the 'Next' reference in a linked list are doing it internally. So internal code is strong enough that there is no other type other than 'LinkedListNode' or 'Null' passed to it atleast in the given context.
- In Figure 5, we can see that the unary operator mutator has been applied on the mutant. In this example, even if the unary operator '+' is changed to '-', an NaN would still be an Nan and a number would still remain a number so it would not make any difference.
- In Figure 6, we can see in the function of *merge sort*, the comparison in helper function *merge* takes place which would not get affected if the values are equal. Although the linked list node references would be different for both the configurations but the list would still said to sorted. Hence, the test would pass in either case. This is the *equivalent mutant*, as we discussed in one of the previous sections.

7.2 Code Coverage of Unit Tests

The following is the overall code coverage report obtained:



all files src/

96.51% Statements 415/430 93.16% Branches 177/190 97.53% Functions 79/81 97.52% Lines 393/403

File		Statements		Branches		Functions		Lines	
binarySearchTree.js		94.71%	161/170	91.84%	90/98	96.43%	27/28	96.64%	144/149
linkedList.js		97.73%	129/132	98%	49/50	95.83%	23/24	97.69%	127/130
queue.js		98.28%	57/58	94.44%	17/18	100%	17/17	100%	54/54
stack.js		97.14%	68/70	87.5%	21/24	100%	12/12	97.14%	68/70

Figure 33: Overall Code Coverage Report

As evident from the report, the following overall coverage metrics were obtained:

1. Statement Coverage - 96.51%
2. Branch Coverage - 93.16%
3. Function Coverage - 97.53%

8 Conclusion

The goal of learning mutation testing with practical aspects is successful. **Overall Mutation Score obtained is 93.5** %. Hence, the mutants are strongly killed by the designed test cases. The designed test cases are hence generally said to be strong enough for robust testing.

9 Contribution

The overall project was carried out as a team effort, we conducted research and analyzed pros and cons of taking each decision as part of the overall design process of the project, all while sticking to the prescribed guidelines of the project. We preferred to write code in JavaScript in order to ensure high readability of code. We experimented with various source code mutation testing libraries and arrived at *Stryker* to be the most optimal one as it is maintained, does periodic releases and stable in nature.

Additionally, we stuck to best industry practices in order to run our workflows as part of the project, we use task runners to automate the execution of the entire unit test case suite, automate the mutation testing procedure and automate the generation of both the code coverage and mutation coverage reports as well.

9.1 Pranjal Walia

1. Setup Mocha and Grunt for running unit tests and coverage generation
2. Setup and configured Stryker for performing mutation testing
3. Designed test cases for BSTs, Queues

9.2 Samaksh Dhingra

1. Designed test cases for linkedLists, Stacks
2. Wrote codebase documentation

10 Relevant Links

1. GitHub Repository
2. Codebase Documentation

11 References

1. `https://stryker-mutator.io/docs/`
2. `https://www.softwaretestinghelp.com/what-is-mutation-testing/`
3. `https://github.com/stryker-mutator`
4. `https://rdcu.be/c0wHD`