



Assignment 1

Mestrado em Engenharia Informática
Verificação e Validação de Software
2018/2019

Grupo 6:
Gonçalo Lobo 44870
Nuno Sousa 47164

Índice:

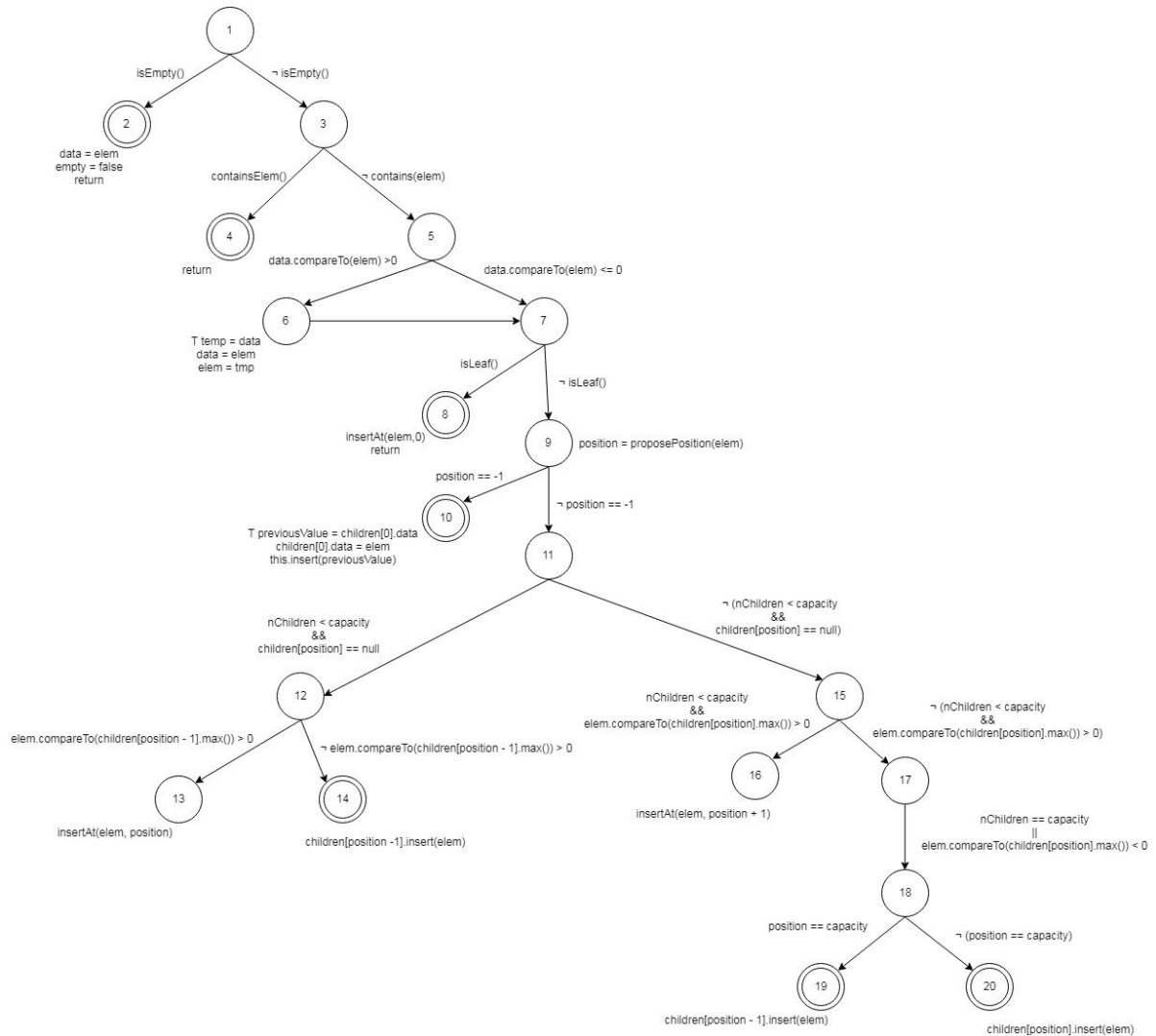
1. Line and Branch Coverage	2
2. Edge-Pair Coverage e Prime Path Coverage	4
3. All-Coupling-Use-Coverage	9
4. Logic-based test coverage	14
5. Base Choice Coverage	16
6. PIT	17
7. JUnit QuickCheck	20
8. Lista de Falhas Corrigidas	20

1. Line and Branch Coverage

De forma a proceder à cobertura de todas as linhas e de todos os ramos foi criada a classe `LineAndBranchCoverage.java`. O método `testCloneTree` testa o método `clone`. Os métodos `testContainsEmptyTree`, `testContainsElementAtRoot`, `testContainsNotContainSmaller`, `testContainsEqual`, `testContainsNotContainLarger` e `testContainsInChildren` testam o método `contains`. Os métodos `testCountLeavesTreeWithOneElement` e `testCountLeavesTreeMoreElements` testam o método `countLeaves`. Os métodos `testDeleteEmptyTree`, `testDeleteRoot`, `testDeleteSmallerThanRoot`, `testDeleteBiggerElement`, `testDeleteSmallerElement` e `testDeleteCompact` testam o método `delete`. Os métodos `testEqualsTwoEmptyTrees`, `testEqualsTreesSameReferences`, `testEqualsEqualTrees`, `testEqualsFirstTreeBigger`, `testEqualsSecondTreeBigger`, `testEqualsNotEqualTrees` e `testEqualsObjectOther` testam o método `equals`. Os métodos `testHeightEmptyTree` e `testHeightMultipleElementsTree` testam o método `height`. Os métodos `testInfoTree` e `TestInfoEmptyTree` testam o método `info`. Os métodos `testInsertEmptyTree`, `testInsertTreeIsLeafSmaller`, `testInsertTreeIsLeafBigger`, `testInsertContains`, `testInsertNewRoot`, `testInsertAtPositionPlusOne`, `testInsertSmallerThanChildren`, `testInsertBiggerThanChildren`, `testInsertMenorQMax`, `testInsertSmallerThanLastChild`, `testInsertSpecialCase` testam o método `insert`. Os métodos `testEmpty` e `testEmptyTreeWithElements` testam o método `isEmpty`. Os métodos `testIsLeafEmptyTree`, `testIsLeafTreeWithOneElement`, `testIsLeafTreeWithMoreElements` testam o método `isLeaf`. Os métodos `testMaxLeafTree` e `testMaxElementOfTree` testam o método `max`. Os métodos `testMinLeafTree` e `testMinTreeMoreElements` testam o método `min`. O método `testNextEmptyStack` testa o método `next`. Os métodos `testHasNextTrue` e `testHasNextFalse` testam o método `hasNext`. Os métodos `testSizeWithOneElement`, `testSizeWithTwoElements` e `testSizeEmptyTree` testam o método `size`. O método `testToListCompare` testa o método `toList`. Os métodos `testToStringEmptyTree`, `testToStringLeaf`, `testToStringMultipleElems` testam o método `toString`.

2. Edge-Pair Coverage e Prime Path Coverage

Método insert:



Nodes & Edges (i)	def (i)	use (i)
1	{}	{}
(1,2) (1,3)	{}	{}
2	{data, empty}	{elem}
3	{}	{}
(3,4) (3,5)	{elem}	{elem}

5	{}	{}
(5,6) (5,7)	{elem}	{elem}
6	{tmp, data, elem}	{data, elem, tmp}
(6,7)	{}	{}
7	{}	{}
(7,8) (7,9)	{}	{}
8	{elem}	{}
9	{position}	{elem}
(9,10) (9,11)	{}	{position}
10	{previousValue, children[0].data}	{children[0].data, elem, previousValue}
11	{}	{}
(11,12) (11,15)	{}	{nChildren, capacity, children, position}
12	{}	{}
(12,13) (12,14)	{}	{elem, children, position}
13	{}	{elem, position}
14	{}	{position, elem, children}
15	{}	{}
(15,16) (15,17)	{}	{nChildren, capacity, elem, position}
16	{}	{elem, position}
17	{}	{}
(17,18)	{}	{nChildren, capacity, elem, position}
18	{}	{}
(18,19) (18,20)	{}	{position, capacity}
19	{}	{children, position, elem}
20	{}	{children, position, elem}

Edge-Pair
[1,2],[1,3],[3,4],[3,5],[5,6],[6,7],[5,7],[7,8],[7,9],[9,10],[9,11],[11,12],[12,13],[12,14],[11,15],[15,16],[15,17],[17,18],[18,19],[18,20],[1,3,4],[1,3,5],[3,5,6],[3,5,7],[5,6,7],[5,7,8],[5,7,9],[6,7,8],[6,7,9],[7,9,10],[7,9,11],[9,11,12],[9,11,15],[11,12,13],[11,12,14],[11,15,16],[11,15,17],[15,17,18],[17,18,19],[17,18,20]

	Test Case Values	Expected Value	Test Path
t1	ArrayNTree<Integer> tree = new ArrayNTree<>(1); tree.insert(1);	[1]	[1,2]
t2	ArrayNTree<Integer> tree = new ArrayNTree<>(3, 1); tree.insert(1);	[1:[3]]	[1,3,5,6,7,8]
t3	ArrayNTree<Integer> tree = new ArrayNTree<>(1, 1); tree.insert(2);	[1:[2]]	[1,3,5,7,8]
t4	List<Integer> list = Arrays.asList(39, 59, 17); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(17);	3	[1,3,4]
t5	List<Integer> list = Arrays.asList(5, 10, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(1);	[1:[5]][10][15]]	[1,3,5,6,7,9,10]
t6	List<Integer> list = Arrays.asList(5, 10, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(20);	[5:[10][15][20]]	[1,3,5,7,9,11,12,13]
t7	List<Integer> list = Arrays.asList(2, 5, 10, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(1);	[1:[2:[5]][10][15]]	[1,3,5,6,7,9,10]
t8	List<Integer> list = Arrays.asList(17, 39, 41, 59, 70, 43, 61); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 4); tree.delete(70); tree.insert(60);	[17:[39][41:[43]][59:[60] [61]]]	[1,3,5,7,9,11,12,14]
t9	List<Integer> list = Arrays.asList(1, 5, 10, 15, 20); ArrayNTree<Integer> tree = new	[1:[5][10][15:[19][20]]]	[1,3,5,7,9,11,15,17,18, 20]

	ArrayNTree<>(list, 3); tree.insert(19);		
t10	List<Integer> list = Arrays.asList(17, 39, 41, 59, 70); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(43);	[17:[39][41:[43]][59:[70]]]	[1,3,5,7,9,11,15,17,18,19]
t11	List<Integer> list = Arrays.asList(1, 5, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(10);	[1:[5][10][15]]	[1,3,5,7,9,11,15,16]

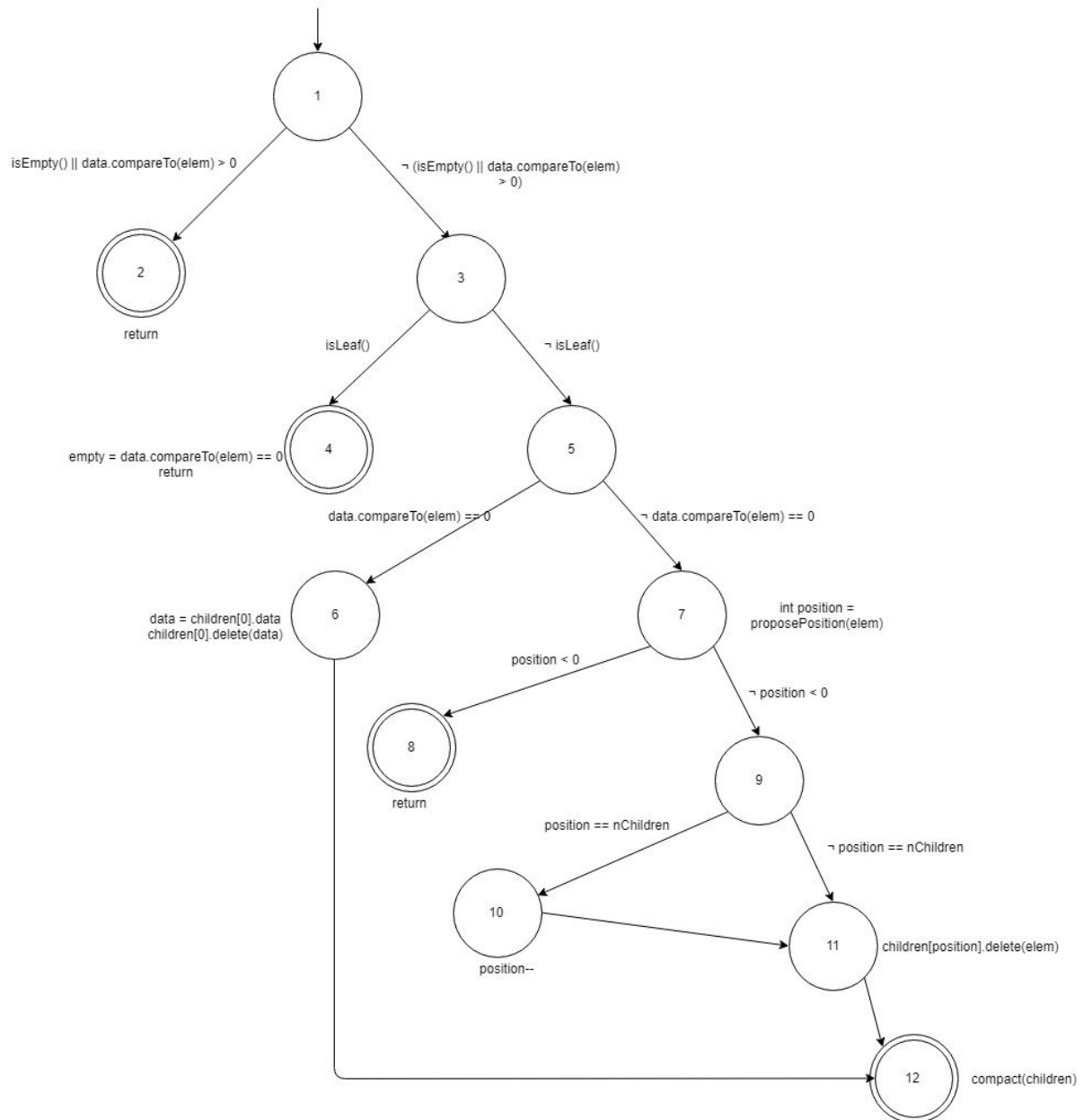
Prime Paths
[1,3,5,6,7,9,11,15,17,18,19],[1,3,5,6,7,9,11,15,17,18,20],[1,3,5,7,9,11,15,17,18,19],[1,3,5,7,9,11,15,17,18,20],[1,3,5,6,7,9,11,15,16],[1,3,5,6,7,9,11,12,13],[1,3,5,6,7,9,11,12,14],[1,3,5,7,9,11,15,16],[1,3,5,7,9,11,12,14],[1,3,5,7,9,11,12,13],[1,3,5,6,7,9,10],[1,3,5,6,7,8],[1,3,5,7,9,10],[1,3,5,7,8],[1,3,4],[1,2]

	Test Case Values	Expected Value	Test Path
t1	ArrayNTree<Integer> tree = new ArrayNTree<>(1); tree.insert(1);	[1]	[1,2]
t2	ArrayNTree<Integer> tree = new ArrayNTree<>(3, 1); tree.insert(1);	[1:[3]]	[1,3,5,6,7,8]
t3	ArrayNTree<Integer> tree = new ArrayNTree<>(1, 1); tree.insert(2);	[1:[2]]	[1,3,5,7,8]
t4	List<Integer> list = Arrays.asList(39, 59, 17); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(17);	3	[1,3,4]
t5	List<Integer> list = Arrays.asList(5, 10, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(1);	[1:[5][10][15]]	[1,3,5,6,7,9,10]
t6	List<Integer> list = Arrays.asList(5, 10, 15);	[5:[10][15][20]]	[1,3,5,7,9,11,12,13]

	<pre>ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(20);</pre>		
t7	<pre>List<Integer> list = Arrays.asList(2, 5, 10, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(1);</pre>	[1:[2:[5]][10][15]]	[1,3,5,6,7,9,10]
t8	<pre>List<Integer> list = Arrays.asList(17, 39, 41, 59, 70, 43, 61); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 4); tree.delete(70); tree.insert(60);</pre>	[17:[39][41:[43]][59:[60][61]]]	[1,3,5,7,9,11,12,14]
t9	<pre>List<Integer> list = Arrays.asList(1, 5, 10, 15, 20); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(19);</pre>	[1:[5][10][15:[19][20]]]	[1,3,5,7,9,11,15,17,18,20]
t10	<pre>List<Integer> list = Arrays.asList(17, 39, 41, 59, 70); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(43);</pre>	[17:[39][41:[43]][59:[70]]]	[1,3,5,7,9,11,15,17,18,19]
t11	<pre>List<Integer> list = Arrays.asList(1, 5, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(10);</pre>	[1:[5][10][15]]	[1,3,5,7,9,11,15,16]

3.All-Coupling-Use-Coverage

Método delete:



Nodes & Edges (i)	def (i)	use (i)
1	{elem}	{}
(1,2) (1,3)	{}	{data, elem}

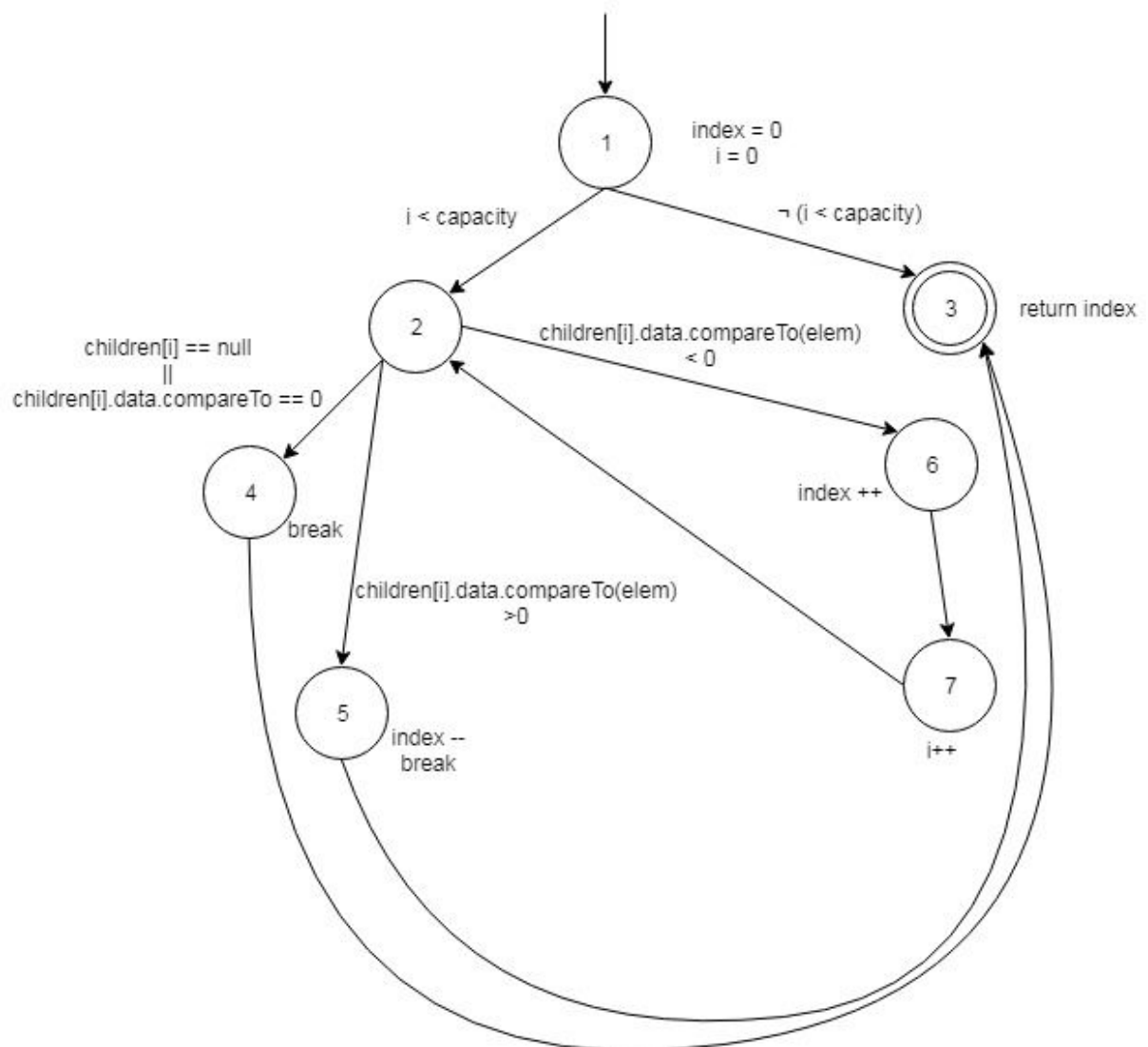
2	{}	{}
3	{}	{}
(3,4) (3,5)	{}	{empty, nChildren}
4	{empty}	{data, elem}
5	{}	{}
(5,6) (5,7)	{}	{data, elem}
6	{data}	{children, data}
(6,12)	{}	{}
7	{position}	{elem}
(7,8) (7,9)	{}	{position}
8	{}	{}
9	{}	{}
(9,10) (9,11)	{}	{position, children}
10	{position}	{position}
(10,11)	{}	{}
11	{}	{children, position, elem}
(11,12)	{}	{}
12	{}	{children}

Variable (v)	last-def (v)	first-use (v)
elem	{1}	{3,5,11}
data	{6}	{6}
position	{7,10}	{8,9,10,11}

Variable (v)	last-def (v)	first-use (v)	Test Path
elem	1	3	[1,3,5,6]
	1	5	[1,3,5,6]
	1	11	[1,3,5,7,9,10,11,12]
data	6	6	[1,3,5,6]

position	7	8	[1,3,5,7,8,12]
	7	9	[1,3,5,7,9,10,11,12]
	7	10	[1,3,5,7,9,10,11,12]
	10	11	[1,3,5,7,9,10,11,12]

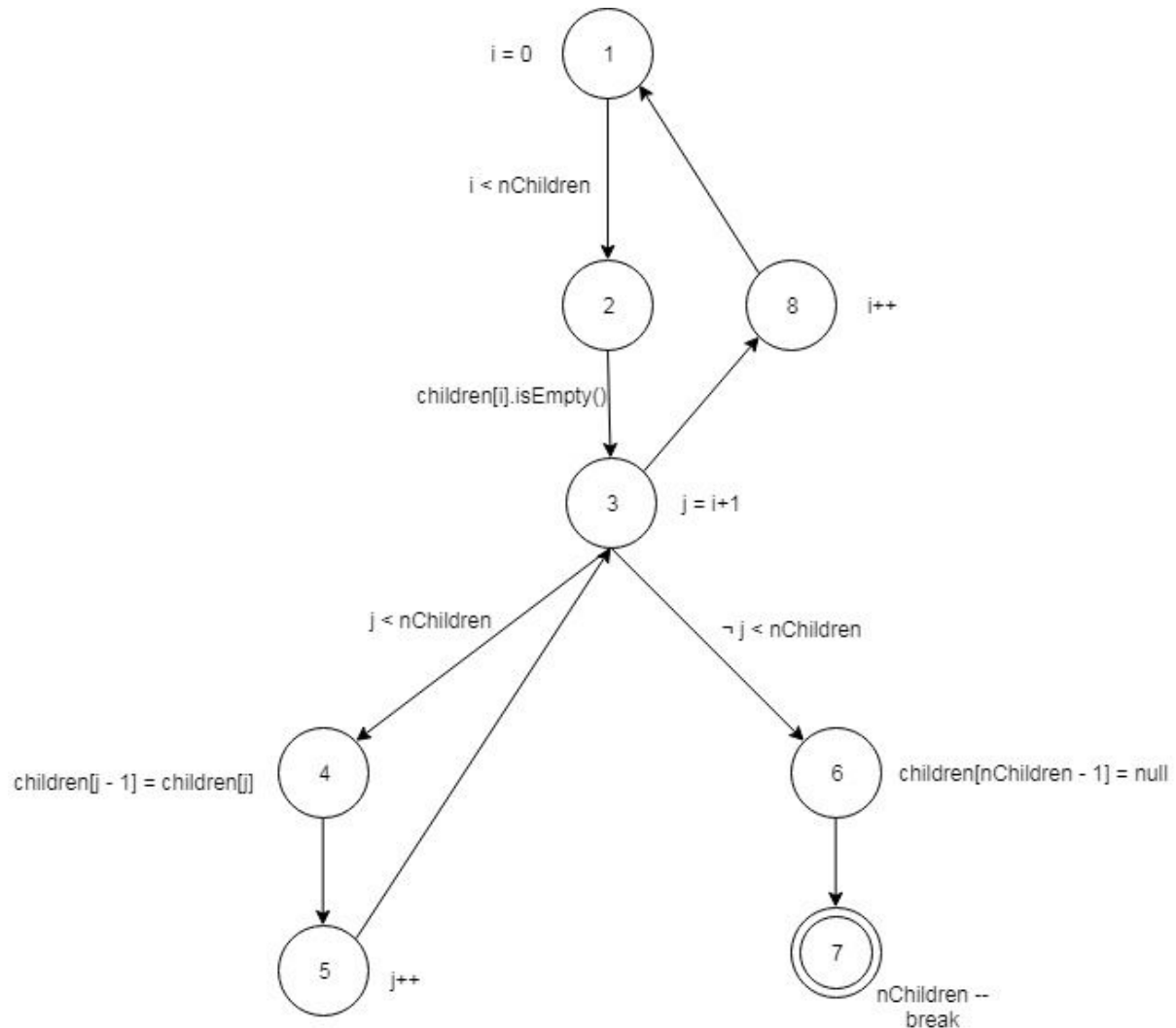
Método proposePosition:



Variable (v)	Last-def (v)	First-use (v)
index	{1,5,6}	{3,5,6}
i	{1,7}	{2,3}
elem	{1}	{4,5,6}

Variable (v)	last-def (v)	first-use (v)	Test Path
index	1	3	<i>infeasible</i>
	1	5	[1,2,6,7,2,5,3]
	1	6	[1,2,6,7,2,5,3]
	5	3	[1,2,6,7,2,5,3]
	5	5	<i>infeasible</i>
	6	3	[1,2,6,7,2,5,3]
	6	5	[1,2,6,7,2,5,3]
	6	6	[1,2,6,7,2,6,7,2,5,3]
i	1	2	[1,2,6,7,2,5,3]
	1	3	[1,2,6,7,2,5,3]
	7	2	[1,2,6,7,2,5,3]
	7	3	[1,2,6,7,2,5,3]
elem	1	4	[1,2,6,7,2,4,3]
	1	5	[1,2,6,7,2,4,3]
	1	6	[1,2,6,7,2,4,3]

Método compact:



Variable (v)	last-def (v)	first-use (v)
i	{1}	{1,2,3,8}
j	{3}	{3,4,5}
nChildren	{1}	{3,4,6,7}

Variable (v)	last-def (v)	first-use (v)	Test Path
i	1	1	[1,2,8,1,2,3,4,5,3,6,7]
	1	2	[1,2,8,1,2,3,4,5,3,6,7]
	1	3	[1,2,8,1,2,3,4,5,3,6,7]
	1	8	[1,2,8,1,2,3,4,5,3,6,7]
	3	3	[1,2,8,1,2,3,4,5,3,6,7]

j	3	4	[1,2,8,1,2,3,4,5,3,6,7]
	3	5	[1,2,8,1,2,3,4,5,3,6,7]
nChildren	1	3	[1,2,8,1,2,3,4,5,3,6,7]
	1	4	[1,2,8,1,2,3,4,5,3,6,7]
	1	6	[1,2,8,1,2,3,4,5,3,6,7]
	1	7	[1,2,8,1,2,3,4,5,3,6,7]

4.Logic-based test coverage

O teste lógico escolhido foi o Predicate Coverage (PC), que para cada predicado p , TR contém dois requirements: p ser avaliado a true e p ser avaliado a false. Pelo facto de o método insert ter várias condições que podem ser avaliadas a true ou a false sentimos que o Predicate Coverage seria o teste lógico que mais se adequava.

Método insert:

Predicates and Clauses
P1: c_1 , where c_1 : isEmpty()
P2: c_2 , where c_2 : contains(elem)
P3: c_3 , where c_3 : data.compareTo(elem) > 0
P4: c_4 , where c_4 : isLeaf()
P5: c_5 , where c_5 : position == -1
P6: $c_6 \ \&\& \ c_7$, where c_6 : nChildren < capacity; c_7 : children[position] == null
P7: c_8 , where c_8 : elem.compareTo(children[position - 1].max()) > 0
P8: $c_9 \ \&\& \ c_{10}$, where c_9 : nChildren < capacity; c_{10} : elem.compareTo(children[position].max()) > 0
P9: $c_{11} \ \ c_{12}$, where c_{11} : nChildren == capacity; c_{12} : elem.compareTo(children[position].max()) < 0
P10: c_{13} , where c_{13} : position == capacity

Tests	P	R(P)
ArrayNTree<Integer> tree = new ArrayNTree<>(3); tree.insert(1);	P1	True
List<Integer> list = Arrays.asList(39, 59, 17); tree = new ArrayNTree<>(list, 3); tree.insert(17);	P2	$R(P_1) \ \&\& \ !P_1$
List<Integer> list = Arrays.asList(5, 10, 15); tree = new ArrayNTree<>(list, 3); tree.insert(1);	P3	$R(P_2) \ \&\& \ !P_2$

tree = new ArrayNTree<>(3); tree.insert(5); tree.insert(10);	P4	R(P3) && !P3
List<Integer> list = Arrays.asList(5, 10, 15); tree = new ArrayNTree<>(list, 3); tree.insert(8);	P5	R(P4) && !P4
List<Integer> list = Arrays.asList(5, 10, 15); tree = new ArrayNTree<>(list, 3); tree.insert(20);	P6	R(P5) && !P5
List<Integer> list = Arrays.asList(5, 10, 15); tree = new ArrayNTree<>(list, 3); tree.insert(20);	P7	R(P6) && P6
List<Integer> list = Arrays.asList(1, 5, 15); ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3); tree.insert(10);	P8	R(P6) && !P6
List<Integer> list = Arrays.asList(1, 5, 10, 15, 20); tree = new ArrayNTree<>(list, 3); tree.insert(19);	P9	R(P8) && !P8
List<Integer> list = Arrays.asList(1, 5, 10, 15, 20); tree = new ArrayNTree<>(list, 3); tree.insert(19);	P10	R(P9) && P9

5.Base Choice Coverage

Método equals:

1. Comparar duas árvores, Tree 1 e Tree 2:
 - a. Tree 1 está vazia (tree1empty)
 - b. Tree 2 não está vazia (!tree1empty)
2. Comparar duas árvores, Tree1 e Tree 2:
 - a. Tree 2 está vazia (tree2empty)
 - b. Tree 2 não está vazia (!tree2empty)
3. Comparar duas árvores, Tree1 e Tree 2:
 - a. Tree 2 está null (tree2null)
 - b. Tree 2 não está a null (!tree2null)
4. Comparar interseção de duas árvores, Tree 1 e Tree 2:
 - a. Não existe interseção (empty)
 - b. Tree 1 e Tree 2 partilham elementos (partial)
 - c. Tree 1 e Tree 2 são iguais (full)

Partitions	Base Choice	Tests
[tree1empty, !tree1empty] [tree2empty, !tree2empty] [tree2null, !tree2null] [empty, partial, full]	[!tree1empty, !tree2empty, !tree2null, empty]	[!tree1empty, !tree2empty, !tree2null, empty]
		[tree1empty, !tree2empty, !tree2null, empty]
		[!tree1empty, tree2empty, !tree2null, empty]
		[!tree1empty, !tree2empty, tree2null, empty]
		[!tree1empty, !tree2empty, !tree2null, partial]
		[!tree1empty, !tree2empty, !tree2null, full]

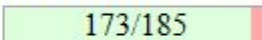
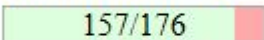
O teste [!tree1empty, !tree2empty, tree2null, empty] não é possível de cobrir, uma vez que no a Tree2 não pode ter elementos e estar a null.

6.PIT

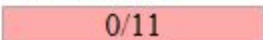
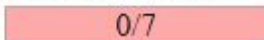
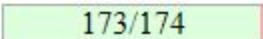
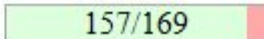
Utilização da ferramenta PIT na classe LineAndBranchCoverage.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	94%  173/185	89%  157/176

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
startup	1	0%  0/11	0%  0/7
sut	1	99%  173/174	93%  157/169

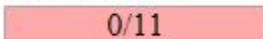
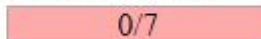
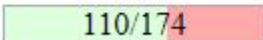
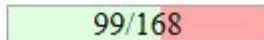
Utilização da ferramenta PIT na classe EdgePairCoverageInsert.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	59%  110/185	57%  99/175

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
startup	1	0%  0/11	0%  0/7
sut	1	63%  110/174	59%  99/168

Utilização da ferramenta PIT na classe PrimePathCoverageInsert.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	59% <div><div>110/185</div></div>	57% <div><div>99/175</div></div>

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
startup	1	0% <div><div>0/11</div></div>	0% <div><div>0/7</div></div>
sut	1	63% <div><div>110/174</div></div>	59% <div><div>99/168</div></div>

Utilização da ferramenta PIT na classe AllCouplingsUsePaths.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	48% <div><div>89/185</div></div>	41% <div><div>71/175</div></div>

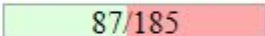
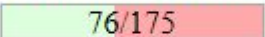
Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
startup	1	0% <div><div>0/11</div></div>	0% <div><div>0/7</div></div>
sut	1	51% <div><div>89/174</div></div>	42% <div><div>71/168</div></div>

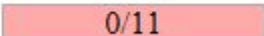
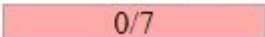
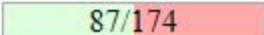
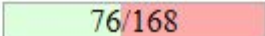
Utilização da ferramenta PIT na classe LogicBaseCoverage.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	47%  87/185	43%  76/175

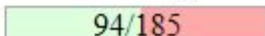

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
startup	1	0%  0/11	0%  0/7
sut	1	50%  87/174	45%  76/168

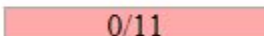
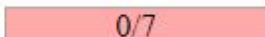
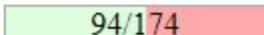
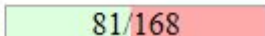
Utilização da ferramenta PIT na classe BaseChoiceCoverage.java:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	51%  94/185	46%  81/175

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
startup	1	0%  0/11	0%  0/7
sut	1	54%  94/174	48%  81/168

7. JUnit QuickCheck

Para efetuar os testes foi criada uma classe `TreeGenerator.java`, para gerar n-trees de forma aleatória. Estas n-trees são de tamanho aleatório e têm elementos aleatórios. Para executar os vários testes foi criada a classe `ArrayNTreeQuickCheck.java`, que contém 5 métodos relativamente a cada uma das propriedades que têm de ser cobertas.

8. Lista de Falhas Corrigidas

Falha nº1:

Foi detectada uma falha ao efectuar testes sobre o método `size`. Este, quando utilizado para saber o tamanho de uma árvore vazia, sem elementos, retorna 1. De este erro resultava que qualquer árvore vazia teria `size` 1, o que não se verifica.

```
/**
 * Caso de teste para o método size para uma árvore vazia
 */
@Test
public void testSizeEmptyTree() {
    ArrayNTree<Integer> tree = new ArrayNTree<>(0);

    int size = tree.size();
    assertEquals(0, size, "size");
}
```

O teste `testSizeEmptyTree` foi o responsável por apanhar esta falha. No teste é possível ver que a `tree` é uma árvore vazia e ao fazer `assertEquals(0, size, "size")` dá erro, o que permite concluir que o tamanho da árvore vazia não seria 0, como seria suposto, mas 1, de acordo como estava codificado o método `size`.

Falha nº2:

Foi detectada uma outra falha ao efetuar os testes sobre o método `equals`, mais em particular sobre o seu método privado `equalTrees`. Este é um método auxiliar do método public `equals` e é chamado na condição de as árvores serem diferentes entre si e a árvore com a qual estamos a comparar ser uma instância de `NTree`. Na implementação desse método está uma condição que verifica se os dois objectos a comparar, o objecto `one` e o objecto `other`, ambos recebidos como argumentos, são o mesmo objeto, ou seja, se têm a mesma referência. Esta condição nunca será possível de alcançar porque o método privado `equalTrees` só é executado no caso de as árvores serem iguais ou diferentes entre si, mas nunca com a mesma referência. Ao efetuar o método `equals` com duas árvores com a mesma referência a condição `this == other` no método `equals` torna-se verdadeira e assim nunca é chamado o método privado `equalTrees`.

```
/**
 * Caso de teste para o método equals a comparar duas árvores com a mesma referência
 */
@Test
public void testEqualsTreesSameReferences() {
    List<Integer> list1 = Arrays.asList(10, 20, 21);
    ArrayNTree<Integer> tree = new ArrayNTree<>(list1, 3);
    ArrayNTree<Integer> tree2 = tree;

    boolean equals = tree2.equals(tree);
    assertTrue(equals);
}
```

O teste *testEqualsTreesSameReferences* permitiu detectar esta falha e assim proceder à sua correção.