

**Assignment 1**

Verificação e Validação de Software 2018/2019

Mestrado em Engenharia Informática

**Grupo 20:**

Guilherme Guimarães 46375  
Catarina Guerreiro 46426  
Mário Teixeira 47074

Índice

[1. Line and Branch Coverage 3](#_Toc7980122)

[2. Edge Pair Coverage e Prime Path Coverage 4](#_Toc7980123)

[Control Flow do método insert() 4](#_Toc7980124)

[3. All-Coupling-Use Coverage 10](#_Toc7980125)

[4. Logic-based test coverage for method insert 13](#_Toc7980127)

[5. Base Choice Coverage 17](#_Toc7980128)

[6. JUnit Quick Check 18](#_Toc7980129)

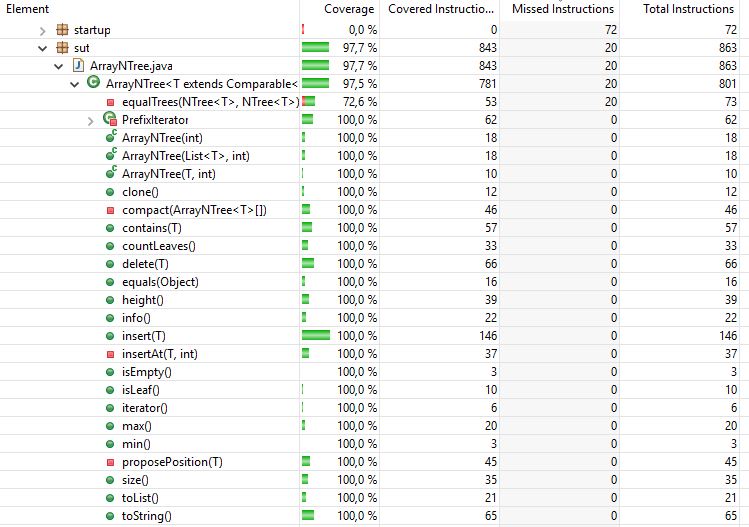
[7. Utilização da ferramenta PIT 19](#_Toc7980130)

[8. Lista de faltas corrigidas 21](#_Toc7980131)

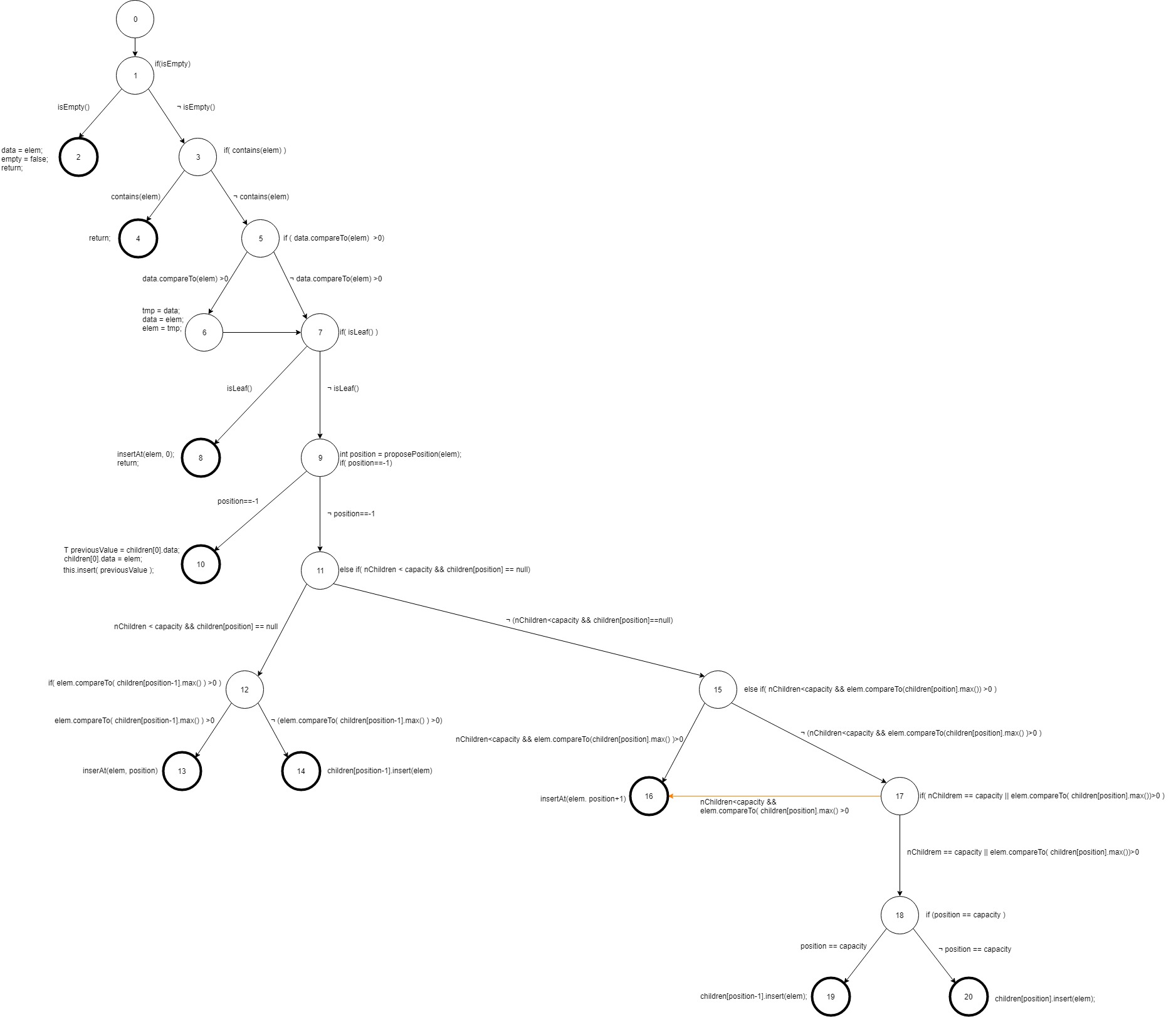
# Line and Branch Coverage

Para proceder à Line and Branch Coverage foram criadas as classes que se encontram dentro do package sut.line\_branch\_coverage.

A classe *TestArrayNTreeClone* tem os métodos necessários para testar o método *clone.* A classe *TestArrayNTreeContains* tem os métodos necessários para testar o método *contains.* A classe *TestArrayNTreeClountLeaves* tem os métodos necessários para testar o método *countLeaves.* A classe *TestArrayNTreeDelete* tem os métodos necessários para testar o método *delete.* A classe *TestArrayNTreeEquals* tem os métodos necessários para testar o método *equals.* A classe *TestArrayNTreeHeight* tem os métodos necessários para testar o método *height.* A classe *TestArrayNTreeInsert* tem os métodos necessários para testar o método *insert.* A classe *TestArrayNTreeIsEmpty* tem os métodos necessários para testar o método *isEmpty.* A classe *TestArrayNTreeisLeaf* tem os métodos necessários para testar o método *isLeaf.* A classe *TestArrayNTreeIterator* tem os métodos necessários para testar todos os métodos públicos da classe *iterator.* A classe *TestArrayNTreeMax* tem os métodos necessários para testar o método *max.* A classe *TestArrayNTreeMin* tem os métodos necessários para testar o método *min.* A classe *TestArrayNTreeSize* tem os métodos necessários para testar o método *size.* A classe *TestArrayNTreeString* tem os métodos necessários para testar os métodos relacionados com a impressão de informação da árvore ArrayNTree em forma de String*.* A classe *TestArrayNTreeToList* tem os métodos necessários para testar o método *toList.*

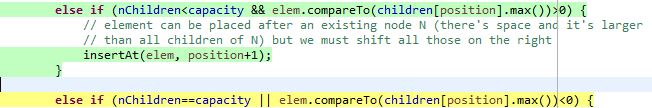
 Obtemos um coverage total de 100% em todos os métodos públicos, como se consegue observar na imagem abaixo.

# Edge Pair Coverage e Prime Path Coverage

Control Flow do método insert():

|  |
| --- |
| Edge-Pair |
| [0,1,2],[0,1,3],[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],[15,17,16],[15,17,18],[17,18,19],[17,18,20] |

|  |  |  |
| --- | --- | --- |
| qNodes & Edges (i) | Def (i) | Use (i) |
| 0 | {} | {} |
| 1 | {} | {} |
| (1,2), (1,3) | {} | {} |
| 2 | {data, empty} | {elem} |
| 3 | {} | {elem} |
| (3,4), (3,5) | {} | {elem} |
| 4 | {} | {} |
| 5 | {} | {data, elem} |
| (5,6), (5,7) | {} | {data, elem} |
| 6 | {tmp, data, elem} | {data, elem, tmp} |
| (6,7) | {} | {} |
| 7 | {} | {} |
| (7,8), (7,9) | {} | {} |
| 8 | {} | {elem} |
| 9 | {position} | {elem, position} |
| (9,10), (9,11) | {} | {position} |
| 10 | {previousValue, children[0].data} | {children[0].data, elem, previousValue} |
| 11 | {} | {nChildren, capacity, children[position]} |
| (11,12), (11,15) | {} | {nChildren, capacity, children[position]} |
| 12 | {} | {elem, children[position-1]} |
| (12,13), (12,14) | {} | {elem, children[position-1]}} |
| 13 | {} | {elem, position} |
| 14 | {} | {children[position-1], elem} |
| 15 | {} | {nChildren, capacity, elem, children[position]} |
| (15,16), (15,17) | {} | {nChildren, capacity, elem, children[position]} |
| 17 | {} | {position, capacity, elem, children[position]} |
| (17,16), (17,18) | {} | {nChildren, capacity, elem, children[position]} |
| 18 | {} | {position, capacity} |
| (18,19), (18,20) | {} | {position, capacity} |
| 19 | {} | {children[position-1], elem} |
| 20 | {} | {children[position], elem} |

O edge-pair [15,17,16] é inatingível, pois, como podemos ver na imagem abaixo, se a intrução a amarelo tiver as suas clausulas a false (nChildren != capacity e ! elem.compareTo(children[position].max())<0), a primeira clausula vai ficar nChildren<capacity, pois nunca pode ser maior e a segunda clausula fica elem.compareTo(children[position].max())>0 porque nunca poderá ser igual, caso fosse igual teria entrado numa das instruções mais acima, que dizia que o elemento já estava contido na árvore. Então, conclui-se que se a instrução a amarela tiver as suas cláusulas a false false entra logo na instrução de cima, logo é inatingível.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Test Case Values | Expected Value | Test Path | Requirements covered |
| T1 | ArrayNTree<Integer> tree = new ArrayNTree<>(2);  tree.insert(null); | [] | [0,1,2] | [0,1,2] |
| T2 | List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | [1] | [0,1,3,4] | [0,1,2],[0,1,3],[1,3,4] |
| T3 | List<Integer> list = Arrays.asList(2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | [1:[2]] | [0,1,3,5,6,7,8] | [0,1,3],[1,3,5],[5,6,7],[6,7,8] |
| T4 | List<Integer> list = Arrays.asList(1,2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(3); | [1:[2][3]] | [0,1,3,5,7,9,11,12,13] | [0,1,3],[1,3,5],[3,5,7],[5,7,9],[7,9,11],[9,11,12],[11,12,13] |
| T5 | List<Integer> list = Arrays.asList(2,3);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | [1:[2][3]] | [0,1,3,5,6,7,9,10] | [0,1,3],[1,3,5],[3,5,6],[5,6,7],[6,7,9],[7,9,10] |
| T6 | List<Integer> list = Arrays.asList(2,7,11,15,20,21,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(20);  tree.delete(21);  tree.insert(18); | [[2:[7][11][15:[16][18][19]]] | [0,1,3,5,7,9,11,12,14] | [0,1,3],[1,3,5],[3,5,7][5,7,9],[7,9,11],[9,11,12],[11,12,14] |
| T7 | List<Integer> list = Arrays.asList(2,7,11,15,25,30,18,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(30);  tree.insert(20); | [2:[7][11][15:[18][19]][20][25]] | [0,1,3,5,7,9,11,15,16] | [0,1,3],[1,3,5],[3,5,7][5,7,9],[7,9,11],[9,11,15],[11,15,16] |
| T8 | List<Integer> list = Arrays.asList(2,7,11,15,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | [2:[7][11][15:[16][18][19]]] | [0,1,3,5,7,9,11,15,17,18,19] | [0,1,3],[1,3,5],[3,5,7][5,7,9],[7,9,11],[11,15,17],[15,17,18],[17,18,19] |
| T9 | List<Integer> list = Arrays.asList(2,7,11,16,19,20);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | [2:[7][11:[16][18][19]][20]] | [0,1,3,5,7,9,11,15,17,18,20] | [0,1,3],[1,3,5],[3,5,7][5,7,9],[7,9,11],[11,15,17],[15,17,18],[17,18,20] |

Os teste da tabela acima encontram-se na classe TestEdgePair dentro do package sut.edge\_pair\_coverage.

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

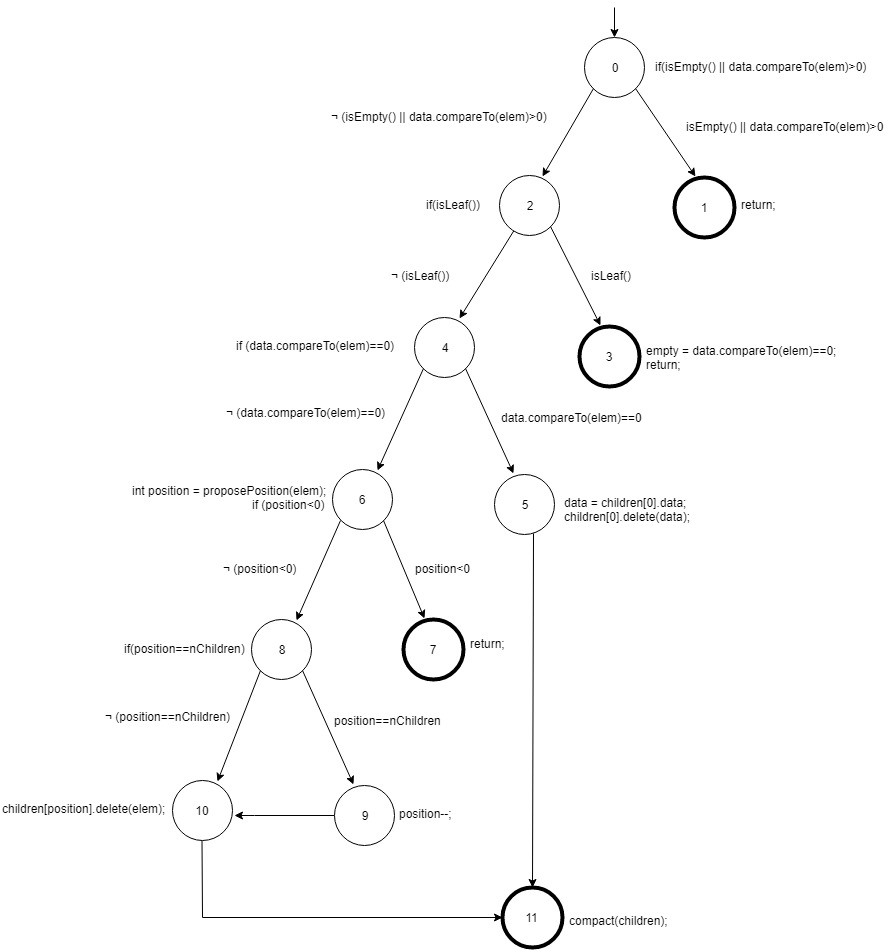
|  |  |  |  |
| --- | --- | --- | --- |
|  | Test Case Values | Expected Value | Test Path |
| T1 | ArrayNTree<Integer> tree = new ArrayNTree<>(2);  tree.insert(null); | [] | [1,2] |
| T2 | List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | [1] | [1,3,4] |
| T3 | List<Integer> list = Arrays.asList(2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | [1:[2]] | [1,3,5,6,7,8] |
| T4 | List<Integer> list = Arrays.asList(1,2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(3); | [1:[2][3]] | [1,3,5,7,9,11,12,13] |
| T5 | List<Integer> list = Arrays.asList(2,3);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | [1:[2][3]] | [1,3,5,6,7,9,10] |
| T6 | List<Integer> list = Arrays.asList(2,7,11,15,20,21,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(20);  tree.delete(21);  tree.insert(18); | [[2:[7][11][15:[16][18][19]]] | [1,3,5,7,9,11,12,14] |
| T7 | List<Integer> list = Arrays.asList(2,7,11,15,25,30,18,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(30);  tree.insert(20); | [2:[7][11][15:[18][19]][29][25]] | [1,3,5,7,9,11,15,16] |
| T8 | List<Integer> list = Arrays.asList(2,7,11,15,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | [2:[7][11][15:[16][18][19]]] | [1,3,5,7,9,11,15,17,18,19] |
| T9 | List<Integer> list = Arrays.asList(2,7,11,16,19,20);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | [2:[7][11:[16][18][19]][20]] | [1,3,5,7,9,11,15,17,18,20] |
| T10 | List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(2); | [1:[2]] | [1,3,5,7,8] |
| T11 | List<Integer> list = Arrays.asList(1,3);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(2); | [1:[2][3]] | [1,3,5,7,9,10] |

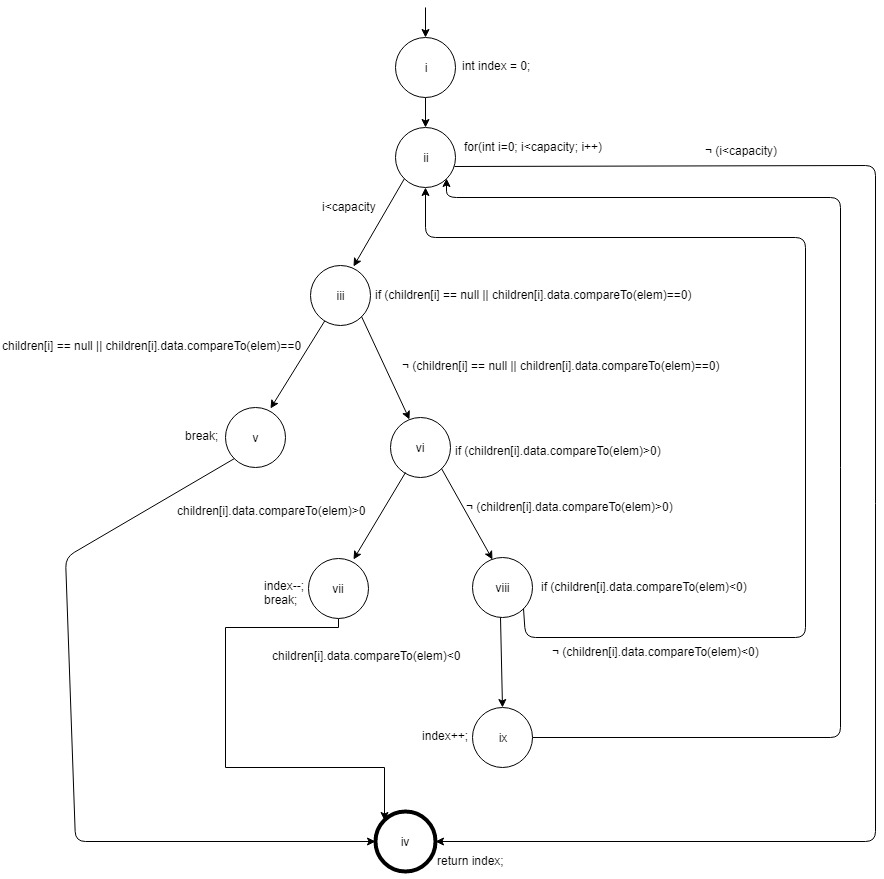
|  |  |
| --- | --- |
| Prime Path | Covered by |
| [1,2] | T1 |
| [1,3,4] | T2 |
| [1,3,5,7,8] | T10 |
| [1,3,5,6,7,8] | T3 |
| [1,3,5,7,9,10] | T11 |
| [1,3,5,6,7,9,10] | T5 |
| [1,3,5,7,9,11,12,13] | T4 |
| [1,3,5,7,9,11,12,14] | T6 |
| [1,3,5,6,7,9,11,12,13] |  |
| [1,3,5,6,7,9,11,12,14] |  |
| [1,3,5,7,9,11,15,16] | T7 |
| [1,3,5,6,7,9,11,15,16] |  |
| [1,3,5,7,9,11,15,17,18,19] | T8 |
| [1,3,5,7,9,11,15,17,18,20] | T9 |
| [1,3,5,6,7,9,11,15,17,18,19] |  |
| [1,3,5,6,7,9,11,15,17,18,20] |  |

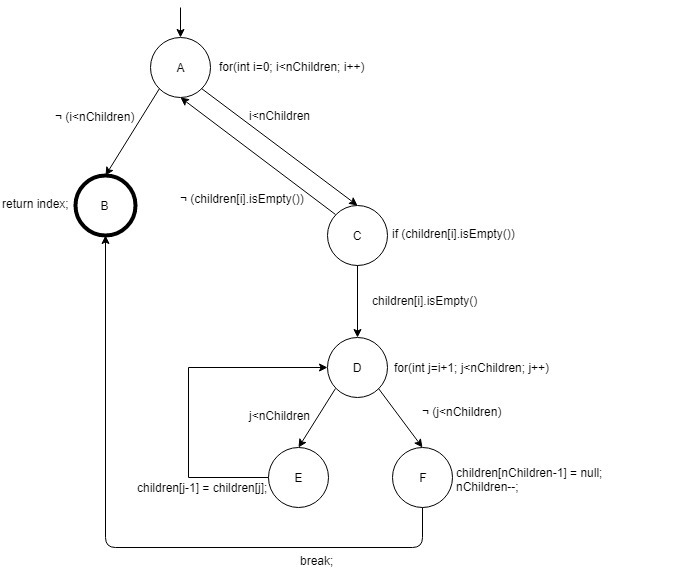
Para os Prime Path conseguimos um coverage de 68.75%.

Os testes da tabela acima encontram-se na classe TestPrimePath dentro do package sut.prime\_path\_coverage.

# All-Coupling-Use Coverage

**Control Flows do método delete():**

**Control Flow do método proposePosition():**

**Control Flow do método compact():**

|  |  |
| --- | --- |
| Last-def | First-use |
| data: {5} | elem{0} |
| index: {vii, ix} | position:{6} |
| children: {E, F} | children: {5,10, iii, C} |

# Logic-based test coverage for method insert

Para o Logic-based coverage obtamos por, primeiro, fazer o Predicate Coverage (PC), mas vimos que não seria o mais indicado, pois cobre muito poucas instruções e queríamos abranger mais casos. Então decidimos fazer o General Active Clause Coverage (GACC), pois assim temos muitos mais casos abrangidos e mais diversidade de testes.

Em baixo, encontram-se as primeiras duas tabelas relativas ao Predicate Coverage (PC) e as últimas duas tabelas relativas ao General Active Clause Coverage (GACC).

**• Predicate Coverage (PC):**

|  |  |
| --- | --- |
| P | R(P) |
| P1 | True |
| P2 | R(P1) ∧ ¬P1 ⇔ !isEmpty() |
| P3 | R(P2) ∧ ¬P2 ⇔ !contains(elem) |
| P4 | R(P3) ∧ ¬P3 ⇔ data.compareTo(elem) ≤ 0 |
| P5 | R(P4) ∧ ¬P4 ⇔ !isLeaf() |
| P6 | R(P5) ∧ ¬P5 ⇔ position != -1 |
| P7 | R(P6) ∧ P6 ⇔ nChildren < capacity && children[position] == null |
| P8 | R(P6) ∧ ¬P6 ⇔ nChildren ≥ capacity || children[position] != null |
| P9 | R(P8) ∧ ¬P8 ⇔ nChildren ≥ capacity || elem.compareTo(children[position].max()) ≤ 0 |
| P10 | R(P9) ∧ P9 ⇔ nChildren == capacity || elem.compareTo(children[position].max()) < 0 |

|  |  |  |
| --- | --- | --- |
| Test Case Values | P | R(P) |
| ArrayNTree<Integer> tree = new ArrayNTree<>(2);  tree.insert(null); | P1 | True |
| List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | P2 | R(P1) ∧ ¬P1 |
| List<Integer> list = Arrays.asList(2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | P3 | R(P2) ∧ ¬P2 |
| List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(2); | P4 | R(P3) ∧ ¬P3 |
| List<Integer> list = Arrays.asList(2,4);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(3); | P5 | R(P4) ∧ ¬P4 |
| List<Integer> list = Arrays.asList(1,2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(3); | P6 e P7 | (R(P5) ∧ ¬P5) ∧ (R(P6) ∧ P6) |
| List<Integer> list = Arrays.asList(2,7,11,15,25,30,18,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(30);  tree.insert(20); | P8 | R(P6) ∧ ¬P6 |
| List<Integer> list = Arrays.asList(2,7,11,15,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | P9 e P10 | (R(P8) ∧ ¬P8) ∧ (R(P9) ∧ P9) |

Os testes da tabela acima encontram-se na classe TestPC dentro do package sut.logic\_based\_coverage.

|  |  |  |
| --- | --- | --- |
| Predicado | Clause Values | Determination |
| P1: isEmpty() | C1: isEmpty() | d(C1) = {} |
| P2: contains(elem) | C2: contains(elem) | d(C2) = {} |
| P3: data.compareTo(elem) > 0 | C3: data.compareTo(elem) > 0 | d(C3) = {} |
| P4: isLeaf() | C4: isLeaf() | d(C4) = {} |
| P5: position == -1 | C5: position == -1 | d(C5) = {} |
| P6: nChildren<capacity ∧ children[position] == null | C6: nChildren<capacity C7: children[position] == null | d(C6) = C7 d(C7) = C6 |
| P7: elem.compareTo(children[position-1].max())>0 | C8: elem.compareTo(children[position-1].max())>0 | d(C8) = {} |
| P8: nChildren<capacity ∧ elem.compareTo(children[position].max())>0 | C9: nChildren<capacity C10: elem.compareTo(children[position].max())>0 | d(C9) = C10 d(C10) = C9 |
| P9: nChildren==capacity V elem.compareTo(children[position].max())<0 | C11: nChildren==capacity C12: elem.compareTo(children[position].max())<0 | d(C11) = ¬ C12 d(C12) = ¬ C11 |
| P10: position==capacity | C13: position==capacity | d(C13) = {} |

**• GACC**

|  |
| --- |
| TR (GACC) = |
| { (1) C1, (2) ¬C1 , (3) C2, (4) ¬C2 , (5) C3, (6) ¬C3 ,  (7) C4, (8) ¬C4 , (9) C5, (10) ¬C5 , (11) C6 ∧ C7 ,   (12) ¬ C6 ∧ C7 , (13) C7 ∧ C6 , (14) ¬C7 ∧ C6 ,  (15) C8, (16) ¬C8 , (17) C9 ∧ C10 , (18) ¬C9 ∧ C10 ,  (19) C10 ∧ C9 , (20) ¬C10 ∧ C9 , (21) C11 ∧ ¬C12 ,  (22) ¬ C11 ∧ ¬C12 , (23) C12 ∧ ¬C11 ,  (24) ¬ C12 ∧ ¬C11 , (25) C13, (26) ¬ C13 } |

As claúsulas 12, 22 e 24 são infeasible, ou seja, são inatingíveis, logo não existem testes possíveis que as cubram.

|  |  |  |
| --- | --- | --- |
| Test # | Test Case Values | Coverage GACC requirements |
| T1 | ArrayNTree<Integer> tree = new ArrayNTree<>(2);  tree.insert(1); | 1. C1 |
| T2 | List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | 1. ¬C1   (3)C2 |
| T3 | List<Integer> list = Arrays.asList(1);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(2); | (4) ¬C2  (5) C3  (7) C4 |
| T4 | List<Integer> list = Arrays.asList(1,2);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(3); | (6) ¬C3  (8) ¬C4  (10) ¬C5  (11) C6 ∧ C7  (13) C7 ∧ C6  (15) C8 |
| T5 | List<Integer> list = Arrays.asList(2,3);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 2);  tree.insert(1); | (9) C5 |
| T6 | List<Integer> list = Arrays.asList(2,7,11,15,20,21,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(20);  tree.delete(21);  tree.insert(18); | (16) ¬ C8 |
| T7 | List<Integer> list = Arrays.asList(2,7,11,15,25,30,18,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 5);  tree.delete(30);  tree.insert(20); | (14) ¬C7 ∧ C6  (17) C9 ∧ C10  (19) C10 ∧ C9 |
| T8 | List<Integer> list = Arrays.asList(2,7,11,15,16,19);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | (25) C13 |
| T9 | List<Integer> list = Arrays.asList(2,7,11,16,19,20);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(18); | (26) ¬ C13 |
| T10 | List<Integer> list = Arrays.asList(2,7,11,16,19,20);  ArrayNTree<Integer> tree = new ArrayNTree<>(list, 3);  tree.insert(21); | (18) ¬C9 ∧ C10  (21) C11 ∧ ¬C12 |

Os testes da tabela acima encontram-se na classe TestGACC dentro do package sut.logic\_based\_coverage.

# Base Choice Coverage

Os testes para o Base Choice Coverage encontram-se na classe BCC dentro do package sut.base\_choice\_coverage.

Para este test set, a base que usamos foi:

Base - [~T1 empty, ~T2 empty, ~T2 null, T1 & T2 empty]

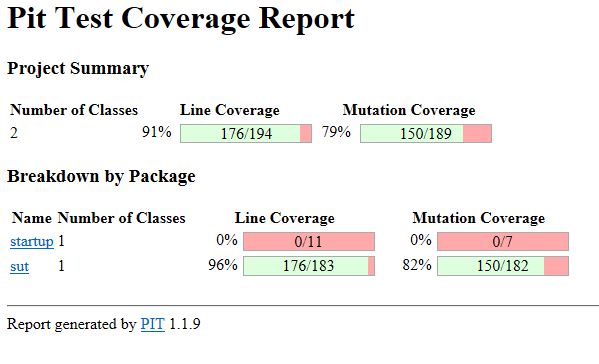
# JUnit Quick Check

Para gerar ArrayNTrees de forma aleatória, foi criada a classe ArrayNTreeGenerator. O gerador gera ArrayNTrees com elementos do tipo Integer que podem ir de 1 a 100, e com no máximo de 50 de tamanho.

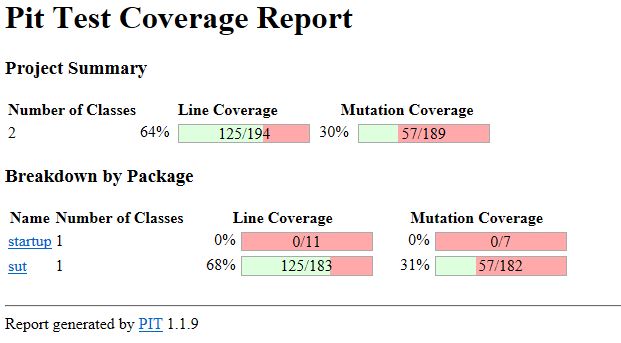
Para executar os testes pedidos foi criada a classe ArrayNTreeQuickCheck. A classe contém os cinco testes que eram pedidos no enunciado, cada um com 15 trials.

# Utilização da ferramenta PIT

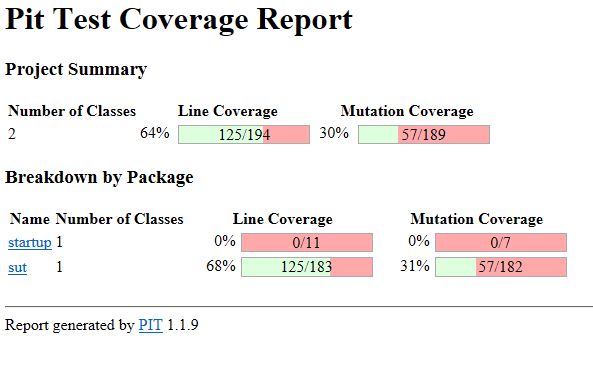
Ferramenta PIT aplicada no package sut.line\_branch\_coverage:



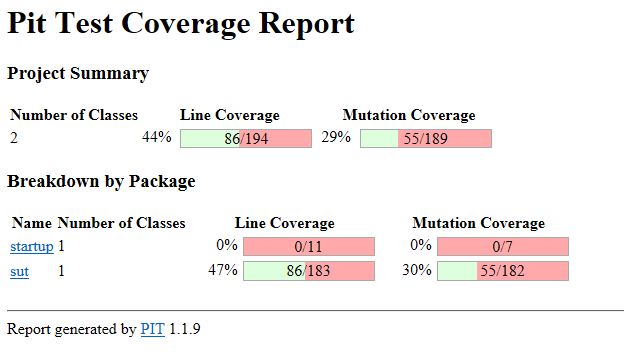
Ferramenta PIT aplicada no package sut.edge\_pair\_coverage:



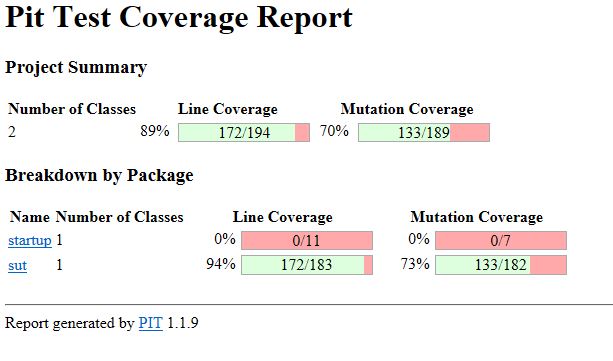
Ferramenta PIT aplicada no package sut.logic\_based\_coverage:



Ferramenta PIT aplicada no package sut.base\_choice\_coverage:



Ferramenta PIT aplicada no package sut.quick\_check:



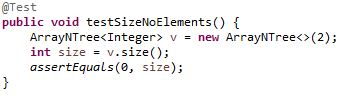
Desta análise podemos concluir que a ferramenta PIT tem um maior coverage no line and branch, pois este package contém as classes com o maior número de testes diversificados para cobrir áreas mais especificas, seguindo-se do quick check.

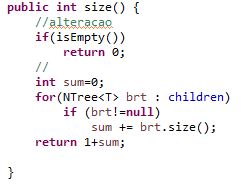
# Lista de faltas corrigidas

Durante o nosso processo de análise ao código fornecido, deparámo-nos com as três seguintes faltas:

1. Método size():

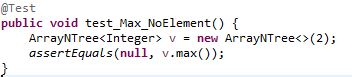
O teste responsável pela deteção desta falha foi:

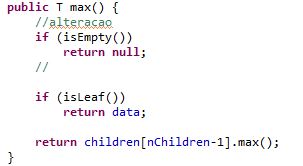


 A alteração realizada foi a de acrescentar o if(isEmpty()), para que retornasse 0, pois quando a árvore estava vazia o seu tamanho nunca era zero mais sim 1.

1. Método max():

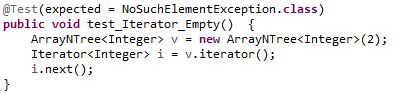
O teste responsável pela deteção desta falha foi:

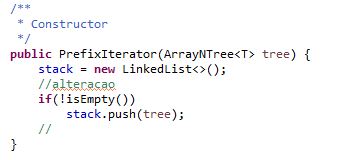


 A alteração feita foi a de acrescentar novamente o if(isEmpty) a retornar null, pois quando se tratava de uma árvore vazia, o seu máximo não dava null e dava IndexOutOfBoundException, pois ia para o return final.

1. Método PrefixIterator(ArrayNTree<T> tree):

O teste responsável pela deteção desta falha foi:



A alteração realizada foi a de acrescentar o if(!isEmpty), para que o push só fosse feito quando a ArrayNTree não é vazia, pois sem esta instrução ao chamar o método next() para devolver o próximo elemento da árvore dava NullPointerException.