**Report**

# Line/Branch Coverage

O construtor do iterador está mal, a raiz é pushed para a stack sem verificar se a tree é emtpy. Isto fazia com que caso a tree fosse vazia, e não haver elementos para iterar, como a stack não estava vazia, a função hasNext retornava true. Este erro foi encontrado no teste testArrayNTreeEquals2EmptyNTreesda classe ArrayNTreeLineBranchCoverageTest.

É impossível cobrir todos os branches do método equalTrees. O branch “one == other” já é coberto no método equals e que caso aconteça, o método equalTrees não é chamado. O branch e “one == null || other == null” também pois, o one aqui seria o this, que não poderia ser null se não ao tentar chamar o método equals seria lançada uma NullPointerException, e o other também nunca poderia ser null pois a condição “other instanceof NTree” não se verificaria e o método equalTrees não seria chamado.

É também impossível cobrir todos os branches do método proposePosition. No if “if (children[i].data.compareTo(elem)<0)” é impossível cobrir o branch em que esta condição não se verifica, isto porque todas as outras condições (ser igual ou maior do que 0) já foram verificadas em partes anteriores do código. Portanto para obter cobertura total, o código teria de ser escrito com if, else if e else, visto que no fim não há outra alternativa a “children[i].data.compareTo(elem)<0”.

# Edge-Pair / Prime Path Coverage

Como o método equals é apenas um return, o seu grafo de controlo de fluxo seriam apenas dois nós. Portanto, para testar este método, foi feito o grafo para o método privado equalTrees, que é onde estão as instruções mais relevantes para a comparação de igualdade entre duas NTrees.

## 2.1. Edge-Pair Coverage

TR(EPC) = { [0,1], [0,2,9], **[0,2,3]**, **[2,3,4]**, **[3,4,5]**, **[3,4,7]**, **[4,5,4]**, **[4,5,6]**, **[4,7,8]**, **[4,7,9]**, **[5,4,5]**, **[5,4,7]** }

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do teste | Valores  (this,other) | Resultado esperado | Caminho | Requerimentos cobertos |
| testArrayNTreeEqualsFirstChildDiff | ([1:[2]], [1:[3]]) | false | [0,2,3,4,5,4,5,6] | [0,2,3] [2,3,4] [3,4,5] [4,5,4] [5,4,5] [4,5,6] |
| testArrayNTreeEqualsEmptyTrees | ([],[]) | true | [0,2,3,4,7,8] | [0,2,3] [2,3,4] [3,4,7] [4,7,8] |
| testArrayNTreeEqualsSingleEqualRoots | ([1],[1]) | true | [0,2,3,4,5,4,7,8] | [0,2,3] [2,3,4]  [3,4,5] [4,5,4]  [5,4,7] [4,7,8] |
| testArrayNTreeEqualsOneEmptyTree | ([1], []) | false | [0,2,3,4,7,9] | [0,2,3] [2,3,4]  [3,4,7] [4,7,9] |

Passar por [0,1] é impossível pq isso e verificado antes de se chamar equalTrees

Passar por [0,2,9] também porque se one não pode ser null porque é o this e o other tmb n pq faz se o instanceof e null não é, logo não passa para o equalTrees. Best effort seria fazer estes testes, que não se vao refletir aqui no equalTrees mas sim no equals,

## 2.2 Prime Path Coverage

Comecemos por determinar os prime paths (marcados a negrito):

(paths marcados a vermelho são caminhos impossíveis)

[0] -> **[0,1]!** [0,2] -> [0,2,3] **[0,2,9]!** -> [0,2,3,4] -> [0,2,3,4,5] [0,2,3,4,7] -> **[0,2,3,4,5,6]! [0,2,3,4,7,8]! [0,2,3,4,7,9]!**

[1]!

[2] -> [2,9]![2,3] -> [2,3,4] -> [2,3,4,5] [2,3,4,7] -> [2,3,4,5,6]! [2,3,4,7,8]! [2,3,4,7,9]!

[3] -> [3,4] -> [3,4,5] [3,4,7] -> [3,4,5,6]! [3,4,7,8]! [3,4,7,9]!

[4] -> [4,5] [4,7] -> **[4,5,4]\*** [4,5,6]! [4,7,8]! [4,7,9]!

[5] -> [5,4] [5,6]! -> **[5,4,5]\*** [5,4,7] -> **[5,4,7,8]!** **[5,4,7,9]!**

[6]!

[7] -> [7,8]! [7,9]!

[8]!

[9]!

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do teste | Valores  (this,other) | Resultado esperado | Caminho | Requerimentos cobertos |
| testArrayNTreeEquals  RootDifferent | ([1],[2]) | false | [0,2,3,4,5,6] | [0,2,3,4,5,6] |
| testArrayNTreeEquals  EmptyTrees | ([],[]) | true | [0,2,3,4,7,8] | [0,2,3,4,7,8] |
| testArrayNTreeEquals  OneEmptyTree | ([1], []) | false | [0,2,3,4,7,9] | [0,2,3,4,7,9] |
| testArrayNTreeEquals  2EqualTreesWith2Elems | ([1:[2]], [1:[2]]) | true | [0,2,3,4,5,4,5,4,7,8] | [4,5,4] [5,4,5] [5,4,7,8] |
| testArrayNTreeEquals  EqualRootsOnlyRootTreeAndRootWithChildTree | ([1:[2]], [1]) | false | [0,2,3,4,5,4,7,9] | [4,5,4] [5,4,7,9] |

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

# All-Coupling-Use Coverage

A função delete, invoca os métodos privados proposePosition e compact isLeaf e isEmpty são só returns.

## 3.1 proposePosition

|  |  |
| --- | --- |
| Last-def | First-use |
| elem: {1} | elem: {d,e} |
| index: {f,h} | position: {8,9} |

Através da tabela, é possível extrair os seguintes pares (last-def -> first-use):

a) (delete, elem, (1) line 204) -> (proposePosition, elem, (d & e) line 268)

b) (proposePosition, (f) index, line 273) -> (delete, position, (8 & 9) line 225)

c)(proposePosition, (h) index, line 278) -> (delete, position, (8 & 9) line 225)

As variáveis de parâmetro e retorno, nos first-uses são arcos, logo são responsáveis os nos finais dos arcos. Isto resulta em pares “diferentes” terem o mesmo fluxo de execução.

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do teste | Valores  (ntree;2) | Resultado Esperado | Requerimentos cobertos |
| testArrayNTreeDeleteFirstChild | ([1:[2]];2) | [1] | a) |
| testArrayNTreeDeleteElemSmallerThanFirstChild | ([1:[3]];2) | [1:[3]] | b) |
| testArrayNTreeDeleteElemGraterThanFirstChild | ([1:[2]];3) | [1:[2]] | c) |

\*Para a representação das árvores foi usada a representação toString dos objetos ArrayNTree

## 3.2 compact

Para o compact, não há return value coupling porque é um método void, portanto a sua tabela apenas irá conter entradas relativas a parameter value coupling.

|  |  |
| --- | --- |
| Last-def | First-use |
| children: {6,11} | chidren: {D,H} |

Através da tabela, é possível extrair os seguintes pares (last-def -> first-use):

1. (delete, children, (6) line 221) -> (compact, children, (D & H) line 243)
2. (delete, children, (11) line 230) -> (compact, children, (D & H) line 243)

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do teste | Valores  (ntree;e) | Resultado esperado | Requerimentos cobertos |
| testArrayNTreeDeleteRootWithChildren | ([1:[2] [3]];1) | [2:[3]] | a) |
|  |  |  |  |

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

# Logic Coverage

Comecemos por identificar os predicados, clausulas e as determinações de predicado para cada cláusula:

* p1: a || b && c
  + a: this == other
    - d(a): ~b || ~c
  + b: other isntanceof NTree
    - d(b): ~a && c
  + c: equalsTrees(this, (NTree<T>) other)
    - d(c): ~a && b
* p2: d
  + d: one == other
    - d(d):true
* p3: e && f
  + e: one != null
    - d(e): f
  + f: other != null
    - d(f): e
* p4: g && h
  + g: it1.hasNext()
    - d(g): h
  + h: it2.hasNext()
    - d(h): g
* p5: !i
  + i: it1.next().equals(it2.next()))
    - d(i): true
* p6: !j && !k
  + j: it1.hasNext()
    - d(j): ~k
  + k: it2.hasNext()
    - d(k): ~j

Com estes dados, é possível extrair os requerimentos para o critério Correlated Active Clause. Este foi o critério escolhido pois com este, estão incluídos os critérios Predicate e Clause Coverage, o que demonstra que este é um critério bastante geral, que fornecerá uma boa cobertura para o conjunto de testes.

TR(CACC):{

(1) a && (~b || ~c) && (a || b && c), (2) ~a && (~b || ~c) && ~(a || b && c),

(3) b && (~a && c) && (a || b && c), **(4) ~b && (~a && c) && ~(a || b && c)**,

(5) c && (~a && b) && (a || b && c), (6) ~c && (~a && b) && ~(a || b && c),

**(7) d && d**, (8) ~d && ~d,

(9) e && f && (e && f), **(10) ~e && f && ~(e && f)**,

(11) f && e && (e && f), **(12) ~f && e && ~(e && f)**,

(13) g && h && (g && h), (14) ~g && h && ~(g && h),

(15) h && g && (g && h), (16) ~h && g && ~(g && h),

(17) i && i, (18) ~i && ~i,

(18) j && ~k && ~(~j && ~k), (19) ~j && ~k && (~j && ~k),

(20) k && ~j && ~(~j && ~k), (21) ~k && ~j && (~j && ~k),

}

A vermelho estão os requisitos para os quais não existem nenhuma combinação de cláusulas exequíveis devido à lógica do programa.

A laranja estão os requisitos que não são possíveis cobrir, pois o predicado já havia sido verificado anteriormente. Estes apenas aparecem para cláusulas de predicados relativos ao método privado equalsTrees, em que há predicados que já foram verificados pelo método equals.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do teste | Valores  (this;other) | Resultado esperado | Predicado e Valores das clausulas | Requerimentos cobertos |
| testArrayNTreeEqualsSameRefNotNTree | Mesma ref | true | p1 :a ~b ~c | (1) a && (~b || ~c) && (a || b && c) |
| testArrayNTreeEqualsOtherNotNTrees | ([],1) | false | p1: ~a ~b ~c | (2) ~a && (~b || ~c) && ~(a || b && c) |
| testArrayNTreeEqualsDiffRefEqualNTrees | ([1],[1]) | true | p1: ~a b c  p2: ~d  p3: e f  p4: g h  p5: i  p6: ~j ~k | (3) b && (~a && c) && (a || b && c)  (5) c && (~a && b) && (a || b && c)  (8) ~d && ~d  (9) e && f && (e && f)  (11) f && e && (e && f)  (13) g && h && (g && h)  (15) h && g && (g && h)  (17) i && i  (19) ~j && ~k && (~j && ~k)  (21) ~k && ~j && (~j && ~k) |
| testArrayNTreeEqualsDiffRoots | ([1],[2]) | false | p1: ~a b ~c  p2: ~d  p3: e f  p4: g h  p5: ~i | (6) ~c && (~a && b) && ~(a || b && c)  (8) ~d && ~d  (9) e && f && (e && f)  (11) f && e && (e && f)  (13) g && h && (g && h)  (15) h && g && (g && h)  (18) ~i && ~i |
| testArrayNTreeEqualsThisEmpty | ([],[1]) | false | p4: ~g h  p6: ~j k | (14) ~g && h && ~(g && h)  (20) k && ~j && ~(~j && ~k) |
| testArrayNTreeEquals  OtherEmpty | ([1],[]) | false | p4: g ~h  p6: j ~k | (16) ~h && g && ~(g && h)  (18) j && ~k && ~(~j && ~k) |

Na escolha dos valores para os predicados foi necessária alguma atenção, pois nem todas as combinações que cobrem requisitos eram exequíveis. Um exemplo deste caso seria para p1 com cláusulas a b e ~c que cobre o requisito a && (~b || ~c) && (a || b && c). Esta combinações de cláusulas é impossível porque caso a e b sejam verdadeiras então c também tem de ser.

\* Todos os resultados foram obtidos com a ferramenta: *Logic Coverage Web Application*

# Base Choice Coverage

Seguindo as seguintes características:

1. Tree 1 is empty – [T1Empty, T1NotEmpty]

2. Tree 2 is empty – [T2Empty, T2NotEmpty]

3. Tree 2 is null - [T2Null, T2NotNull]

4. Tree 1 intersection of Tree 2 is empty/full/partial – [E,F,P]

Um bom caso base seria aquele que seria o mais comum num ambiente de execução normal do sistema, portanto sugere-se: (T1NotEmpty, T2NotEmpty, T2NotNull, E). Para a característica 4, o valor que aparenta ser o mais comum seria o P (partial intersection), o problema deste valor é que seriam gerados muitos requisitos impossíveis (com T1 ou T2 empty interseção parcial é impossível). Portanto, a partir do caso base escolhido, os requerimentos para testar são:

TR(BCC): {

(1) (T1NotEmpty, T2NotEmpty, T2NotNull, E), (2) (**T1Empty**, T2NotEmpty, T2NotNull, E),

(3) (T1NotEmpty, **T2Empty**, T2NotNull, E), (4) (T1NotEmpty, T2NotEmpty, **T2Null**, E),

(5) (T1NotEmpty, T2NotEmpty, T2NotNull, **F**),(6) (T1NotEmpty, T2NotEmpty, T2NotNull, **P**)

}

Em cada requisito há uma característica a negrito, que indica a que foi alterada, de modo a seguir o Base Choice Coverage.

O requisito (4) é impossível pois não há como T2 ter valores e ser null.

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do teste | Valores  (this;other) | Resultado Esperado | Requerimentos cobertos |
| testArrayNTreeEqualsDiffNotNullTrees | ([1:[2]];[5:[8][11]]) | false | (1) |
| testArrayNTreeEqualsThisEmpty | ([],[1:[3]]) | false | (2) |
| testArrayNTreeEqualsOtherEmpty | ([2:[5] [6]];[]) | false | (3) |
| testArrayNTreeEqualsThisEqualsOther | ([3:[4] [5]]; [3:[4] [5]]) | true | (5) |
| testArrayNTreeEqualsThisSubsetOther | ([3:[4]]; [1:[3] [4:[5] [6]]] | false | (6) |

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

# Verificação dos conjuntos de testes para o equals com program mutation (PIT)

So para o base choice ou para todos?

Mutatações noutros métodos não relacionados com o equals, mas que são usados nos testes?