CAPA

## Passo-a-passo de cada requerimento

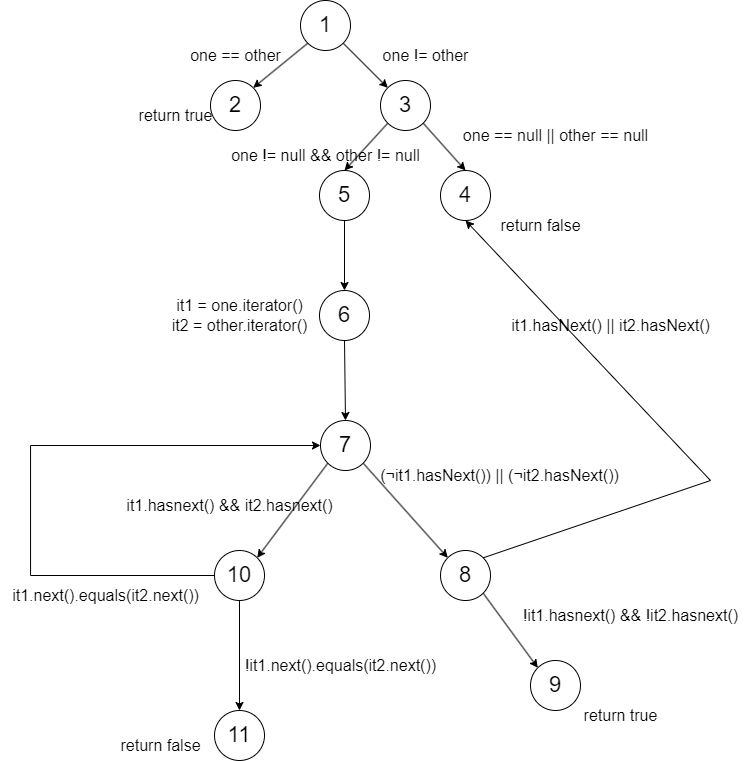
1. Line and Branch Coverage for methods contains and equals;

Para a criação dos testes para o line e branch coverage de cada método foram criados 2 ficheiros: “TestLCAndBCContains.java” e “TestLCAndBCEquals.java”. Para verificar o line e branch coverage foi utilizada uma ferramenta do eclipse, o uso da ferramenta basea-se na execução dos testes utilizando a opção Coverage As -> JUnit Test no respetivo ficheiro a executar.

Após a execução dos testes podemos concluir que a primeira condição no método *equalTrees*, *if (one == other)*,na linha 301, é impossível ser concretizada pois essa condição já é verificada no início do *equals* e quando verdadeira o resultado é logo retornado, nem sendo executada a função *equalTrees*, logo a linha 302 é impossível ser atingida.   
Para além desse branch ser impossível de atingir, existem mais 2 branch que não são possíveis de ser atingidos, na condição *if (one != null && other != null)*, os branches: *one == null && other != null* e *one != null && other != null*. Estes branches são impossíveis de serem atingidos pois o other é brevemente verificado se é null ou não quando executa o programa verifica o seguinte, *other instanceof NTree*. Caso other seja null, o resultado do operador binário, &&, é de imediato false, não sendo executado o equalTrees.

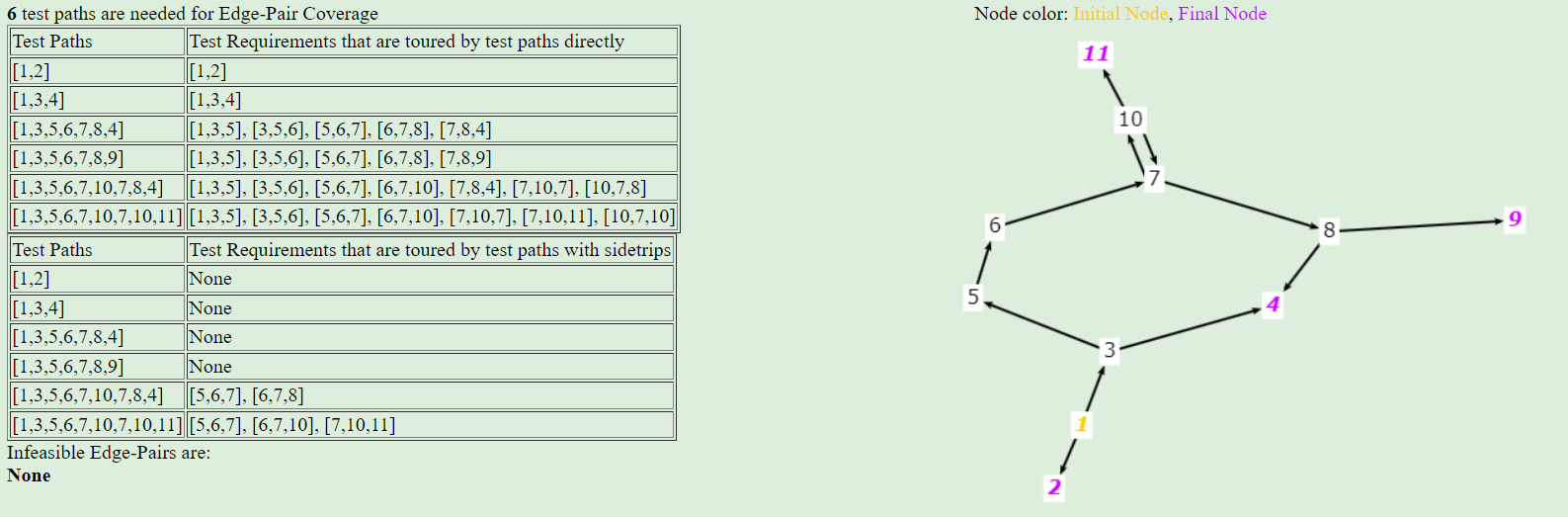
1. Edge-Pair Coverage and at least 50% coverage for Prime Path Coverage for method equals;

Antes de tudo, é preciso construir o grafo para o método equals e este foi o grafo obtido:



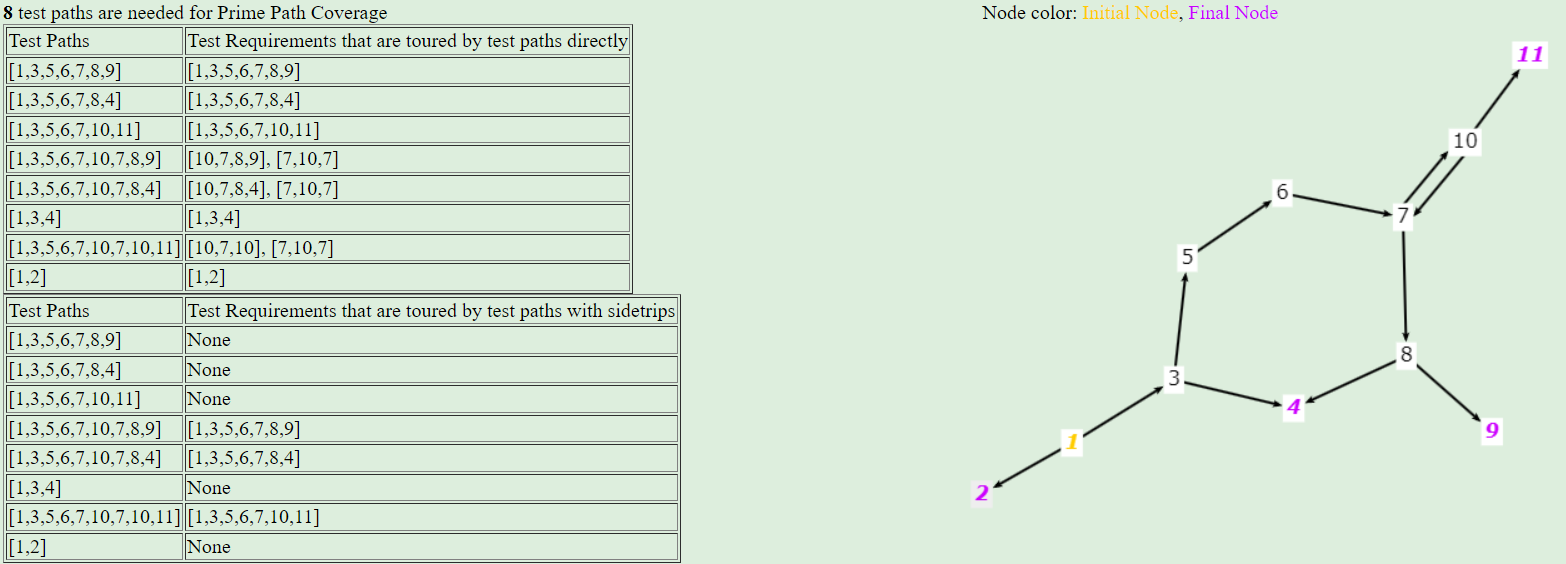
Após a criação do grafo foi utilizado o website Graph Coverage Web Application (<https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage>), e lá foram introduzidas as informações que podemos obter analisando o grafo. Utilizando o website podemos obter automaticamente os testes para o Edge-Pair Coverage ([link para os testes do edge-pair coverage](https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage?edges=1+2%0D%0A1+3%0D%0A3+4%0D%0A3+5%0D%0A5+6%0D%0A6+7%0D%0A7+8%0D%0A7+10%0D%0A8+9%0D%0A8+4%0D%0A10+7%0D%0A10+11%0D%0A&initialNode=1&endNode=2+4+9+11&action=Edge-Pair%20Coverage)) e para o Prime Path Coverage ([link para os teste do prime path coverage](https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage?edges=1+2%0D%0A1+3%0D%0A3+4%0D%0A3+5%0D%0A5+6%0D%0A6+7%0D%0A7+8%0D%0A7+10%0D%0A8+9%0D%0A8+4%0D%0A10+7%0D%0A10+11%0D%0A&initialNode=1&endNode=2+4+9+11&action=Prime%20Path%20Coverage)).

**Edge-Pair Coverage:**



Para o Edge-Pair Coverage foram realizados todos os test paths.

**Prime Path Coverage:**

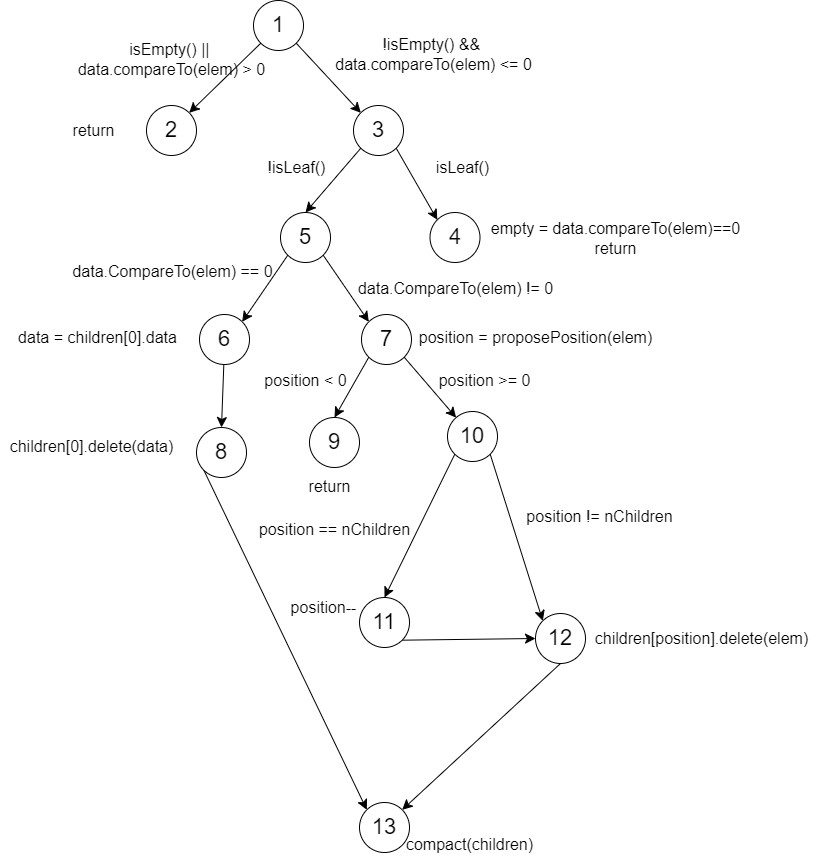
  
Para o Prime Path Coverage foram realizados apenas estes test paths:

De todos os requerimentos obtidos, estes são os infeasible:

1. All-Coupling-Use Coverage for method delete and its private methods;

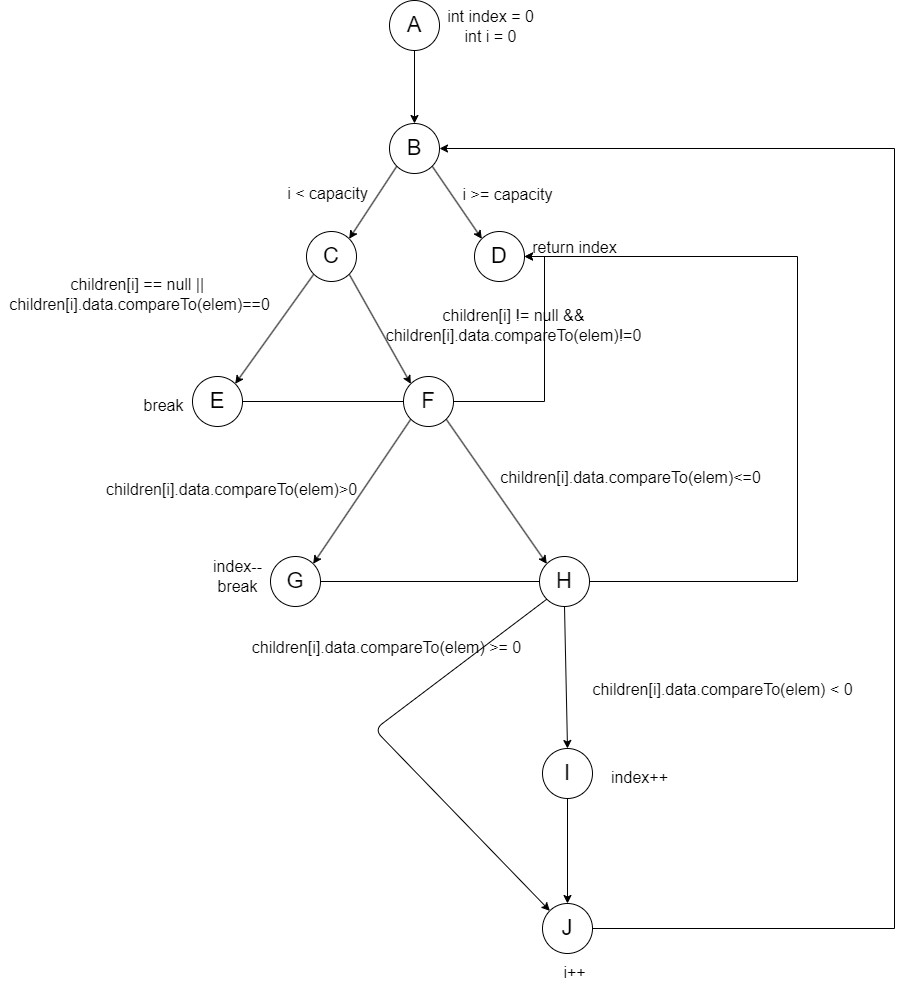
Para perceber os testes necessários para o All-Coupling-Use Coverage sobre o método delete e os outros métodos privados que o delete utiliza, foram construídos os seguintes grafos e as seguintes tabelas.

**Método delete:**

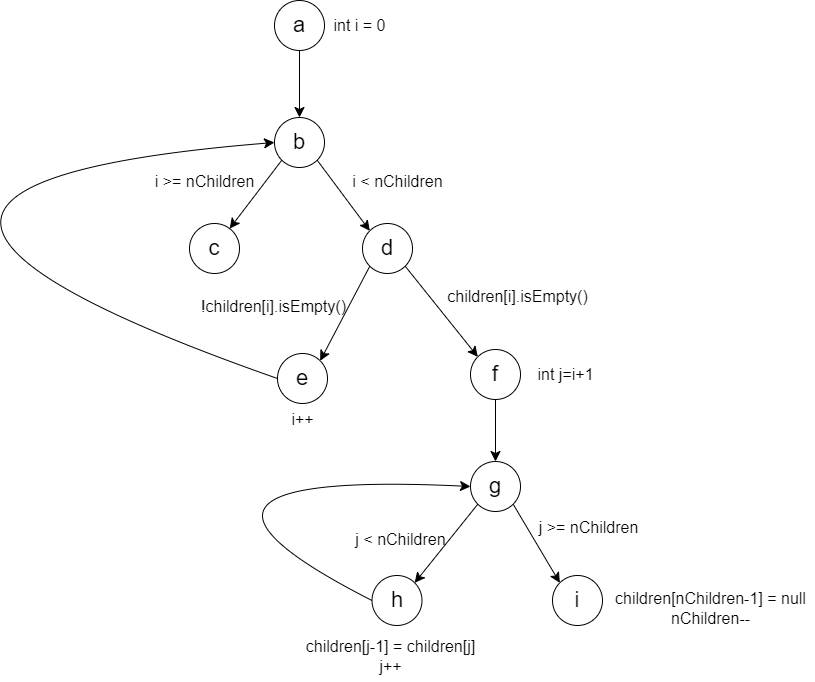


|  |  |  |  |
| --- | --- | --- | --- |
| **Nodes&Edged:i** | **Def(i)** | **Use(i)** | **Functions used** |
| 1 | {empty, nChildren, data, elem, children} |  |  |
| (1,2), (1,3) |  | {empty, data, elem} | empty,compareTo |
| 2 |  |  |  |
| 3 |  |  |  |
| (3,4), (3,5) |  | {empty, nChildren} | isLeaf |
| 4 | {empty} | {data, elem} | compareTo |
| 5 |  |  |  |
| (5,6), (5,7) |  | {data,elem} | CompareTo |
| 6 | {data} | {children} |  |
| (6,8) |  |  |  |
| 7 | {position} | {elem} | proposePosition |
| (7,9), (7,10) |  | {position} |  |
| 8 | {children} | {children,data} | Delete |
| (8,13) |  |  |  |
| 9 |  |  |  |
| 10 |  |  |  |
| (10,11), (10,12) |  | {position,nchildren} |  |
| 11 | {position} |  |  |
| (11,12) |  |  |  |
| 12 | {children} | {children,position,elem} | Delete |
| (12,13) |  |  |  |
| 13 |  | {children} | compact |

**Método proposePosition:**



|  |  |  |  |
| --- | --- | --- | --- |
| **Nodes&Edges:I** | **def(i)** | **use(i)** | **Functions used** |
| A | {index,i,children,elem} |  |  |
| (A,B) |  |  |  |
| B |  |  |  |
| (B,C), (B,D) |  | {i,capacity} |  |
| C |  |  |  |
| (C,E),(C,F) |  | {children,i,elem} | compareTo |
| D |  | {index} |  |
| E |  |  |  |
| (E,D) |  |  |  |
| F |  |  |  |
| (F,G),(F,H) |  | {children,elem} | compareTo |
| G | {index} | {index} |  |
| (G,D) |  |  |  |
| H |  |  |  |
| (H,I),(H,J) |  | {children,elem} | compareTo |
| I | {index} | {index} |  |
| (I,J) |  |  |  |
| J | {i} | {i} |  |
| (J,B) |  |  |  |

**Método compact:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nodes&Edges:I** | **def(i)** | **use(i)** | **Functions used** |
| a | {i, nChildren,empty, children} |  |  |
| (a,b) |  |  |  |
| b |  |  |  |
| (b,c),(b,d) |  | {i,nChildren} |  |
| c |  |  |  |
| d |  |  |  |
| (d,e),(d,f) |  | {i, children, empty} | isEmpty |
| e | {i} | {i} |  |
| (e,b) |  |  |  |
| f | {j} | {i} |  |
| (f,g) |  |  |  |
| g |  |  |  |
| (g,h),(g,i) |  | {j,nChildren} |  |
| h | {children,j} | {children,j} |  |
| (h,g) |  |  |  |
| i | {children,nChildren} | {children,nChildren} |  |

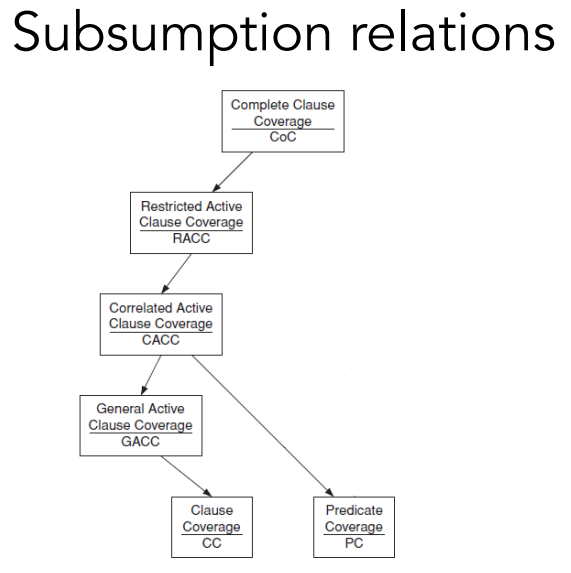
Através das tabelas obtidas foram analisadas as variáveis que provêm de outras funções e o primeiro uso delas e a última definição das mesmas. Através dessa análise obtemos os seguintes resultados:

|  |  |  |
| --- | --- | --- |
| Function | First use | Last definition |
| **delete** | Variável **position** em “position < 0” – edge (7,9) e (7,10) | Última definição na função **proposePosition** nonodeA, G ou I |
| **compact** | Variável **children** em “children[i].isEmpty()” – edge (d,e) e (d,f) | Última definição na função **delete** nonode 8 ou 12 |
| **proposePosition** | Variável **elem** em “children[i].data.compareTo(elem)==0” – edge (c,e),(c,f) | Última definição na função **delete** nonode 1 |

No ficheiro **TestAllCouplingUseCoverage.java** estão presentes os testes necessários para o All-Coupling-Use Coverage com a devida indicação de quais testes cobrem quais requisitos especificamente.

1. Select and apply one Logic-based test coverage for method equals, justify your option.

O teste Logic-based escolhido foi o CACC, apesar do CoC trazer uma abordagem mais completa relativamente a testar um programa o CACC é a melhor abordagem a se fazer sendo mesmo melhor que o RACC. Apesar do RACC ser mais uniforme que o CACC, o RACC neste caso gera mais requerimentos infeasible. Alem disso o CACC, tal como o RACC dá coverage ao PC e ao GACC que por consequência dá coverage ao CC.

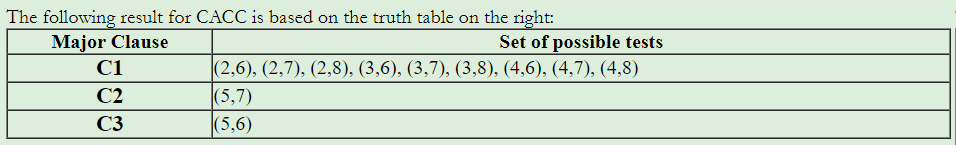
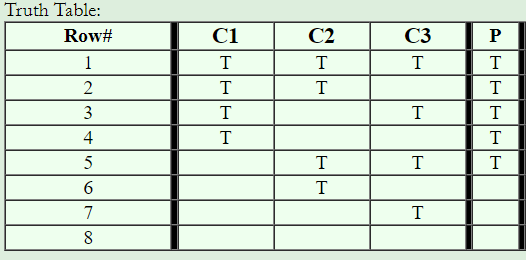


Como auxílio para a criação das tabelas para o CACC foi utilizado o site Logic Coverage Web Application (<https://cs.gmu.edu:8443/offutt/coverage/LogicCoverage>).

Os predicados e clausulas avaliadas foram as seguintes:

**Método equals**

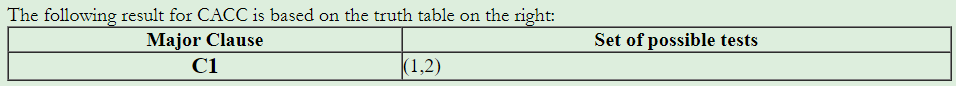
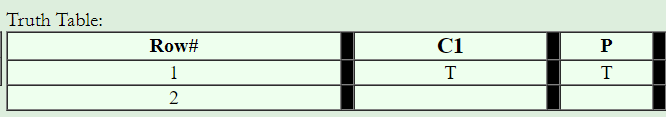
**P1:** C1 || C2 && C3  
C1: this == other  
C2: other instanceof NTree  
C3: equalTrees(this, ((NTree<T>) other))

Testes infeasible:   
2 – É impossível o mesmo objeto da instância NTree possuir árvores diferentes  
3 – É impossível o segundo objeto não ser da instância NTree e possuir uma árvore igual à primeira árvore  
4 – É impossível o mesmo objeto que não é instancia NTree chamar a função equals da classe ArrayNTree   
7 – É impossível 2 objetos diferentes sendo o segundo não ser da instância NTree possuir árvores iguais.

Testes utilizados para cumprir o C1: (8), pois os testes 2, 3, 4 e 7 são impossíveis de serem realizados.  
Testes utilizados para cumprir o C2: (5), pois o teste 7 é impossível de ser realizado.  
Testes utilizados para cumprir o C3: (5,6)

**Método equalTrees**

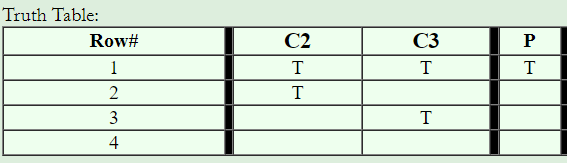
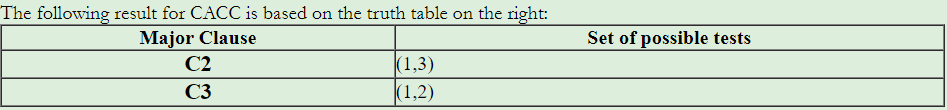
**P1**: C1  
C1: one == other



Testes infeasible:   
1 – É impossível testar quando one == other, pois se one for igual a other essa condição já é previamente verificada e caso seja verdade o método equalTrees nem chega a ser executado.

Testes utilizados para cumprir o C1: (2), pois o teste 1 é impossível de ser realizado.

**P2**: C2 && C3  
C2: one != null  
C3: other != null

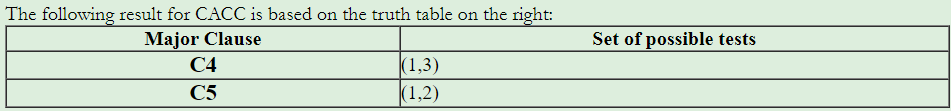
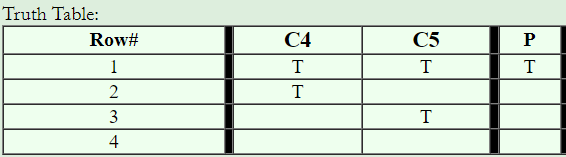
Testes infeasible:   
2 – A cláusula !C3 é impossível de ser cumprida pois caso o other seja null este já é previamente verificado no método equals e o método equalTrees nem chegaria a ser executado.   
3 - A cláusula !C2 é impossível de ser cumprida porque one é o objeto que chama a função equals e caso ele seja null não seria se quer possível chamar a função.  
4 – Este teste é infeasible devido às 2 razões fornecidas anteriormente.

Testes utilizados para cumprir o C2: (1), pois o teste 3 é impossível de ser realizado.

pois é impossível realizar o testar quando C3 é falso, i.e, quando other == null, como já foi explicado anteriormente.  
Testes utilizados para cumprir o C3: (1), pois o teste 2 é impossível de ser realizado.

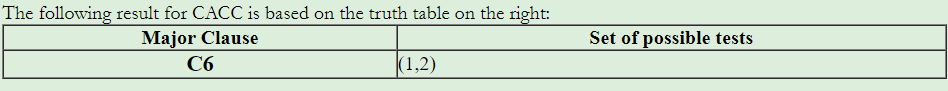
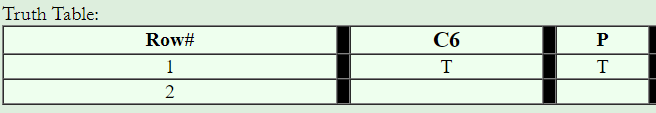
pois é impossível realizar o testar quando C2 é falso, i.e, quando one == null, como já foi explicado anteriormente.

**P3**: C4 && C5  
C4: it1.hasNext()  
C5: it2.hasNext()



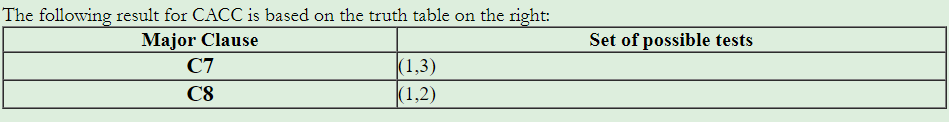
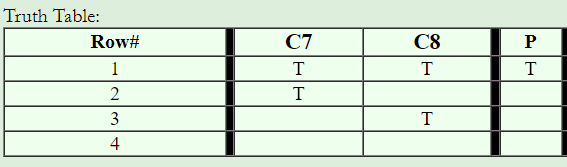
Testes utilizados para cumprir o C4: (1,3).  
Testes utilizados para cumprir o C5: (1,2).

**P4**: C6  
C6: !it1.next().equals(it2.next())



Testes utilizados para cumprir o C6: (1,2).

**P5**: C7 && C8  
C7: !it1.hasNext()  
C8: !it2.hasNext()



Testes impossíveis:   
4 -

Testes utilizados para cumprir o C7: (1,3).  
Testes utilizados para cumprir o C8: (1,2).