**Proiect TSS**

**Specificațiile problemei:**

Să se implementeze o metodă Java care primește la intrare:

• un vector array de numere întregi de lungime arbitrară,

• o valoare lowerbound, care reprezintă indicele de început al vectorului v,

• o valoare upperbound, care reprezintă indicele de sfârșit al vectorului v,

• o valoare key.

Metoda returnează indicele elementului key, dacă acesta se află în vector, sau -1, dacă nu se află în vector.

**Input:**

* array = vector de numere întregi
* start = indice de început
* end = indice de sfârșit
* search = elementul căutat

**Output:**

* index = poziția elementului căutat
* -1 = caz în care nu a fost găsit elementul
* eroare = dacă parametrii sunt incorecti

**Testare Funcțională**

1. **Partiționare de echivalență**

Domeniul de intrări:

* array = vector de numere întregi
* start = valoarea de început a vectorului (0)
* end = lungimea vectorului – 1 (practic numărul de elemente din vector, dar numarand de la 0 trebuie scăzut 1)
* search = valoarea căutată în vector

Pentru fiecare intrare, distingem următoarele clase de echivalență:

* + Pentru n:
    - array = { n | n numar natural }
  + Pentru start:
    - ST\_1 = { st | st = 0 }
    - ST\_2 = { st | st != 0}
  + Pentru end:
    - E\_1 = { e | e = (array.length – 1) }
    - E\_2 = { e | e != ( array.length -1 )}
  + Pentru search:
    - S = {s | s număr întreg}

Output-ul posibil este format din cele 3 valori posibile:

* + Eroare, în cazul în care parametrii nu coincid
  + Poziția elementului găsit
  + -1, dacă elementul nu a fost gasit

Clasele de echivalenta rezultate din domeniul de intrari și iesiri sunt:

C\_1={ array = {a1, a2, ....}, st != 0, e != array.length, oricare s}

C\_2={ array = {a1, a2, ....},st = 0, e != array.length, oricare s}

C\_3={ array = {a1, a2, ....}, st != 0, e = array.length, oricare s}

C\_4={ array = {a1, a2, ....}, st = 0, e = array.length, s=a1 || s=a2 || ... }

C\_5={ array = {a1, a2, ....} , st = 0, e = array.length, s != a1 && s != a2 ...}

Pentru a acoperi cele 5 clase de echivalenta, se aleg urmatoarele date de test:

|  |  |
| --- | --- |
| **Date de intrare** | **Răspuns** |
| **T1 ({0,1,2,3}, 1, 1, 2)** | Error = invalid start and end |
| **T2 ({0,1,2,3}, 0, 1, 2)** | Error = invalid end |
| **T3 ({0,1,2,3}, 1, 3, 2)** | Error = invalid start |
| **T4 ({0,1,2,3}, 0, 3, 2)** | 2 |
| **T5 ({0,1,2,3}, 0, 3, 99)** | -1 |

1. **Analiza valorilor de frontieră**

Valorile de frontiera sunt:

* + start = {-1, 0, 1}
  + end = {array.length – 1, array.length, array.length +1}

Exista 3 \* 3 = 9 posibile alegeri de valori pentru a acoperi valorile de frontiera:

|  |  |
| --- | --- |
| **Date de intrare** | **Răspuns** |
| **T1 ({0,1,2,3}, -1, 2, 2)** | Error = invalid start and end |
| **T2 ({0,1,2,3}, -1, 3, 2)** | Error = invalid start |
| **T3 ({0,1,2,3}, -1, 4, 2)** | Error = invalid start and end |
| **T4 ({0,1,2,3}, 0, 2, 2)** | Error = invalid end |
| **T5 ({0,1,2,3}, 0, 3, 2)** | 2 |
| **T6 ({0,1,2,3}, 0, 4, 2)** | Error = invalid end |
| **T7 ({0,1,2,3}, 1, 2, 2)** | Error = invalid start and end |
| **T8 ({0,1,2,3}, 1, 3, 2)** | Error = invalid start |
| **T9 ({0,1,2,3}, 1, 4, 2)** | Error = invalid start and end |

1. **Partiționarea în categorii**
   1. Descompunerea specificațiilor în unități: avem o singură unitate
   2. Identificarea parametrilor: array, start, end, search
   3. Găsirea categoriilor:

* start: dacă este 0 sau nu
* end: dacă este egal cu lungimea array-ului sau nu
* search: dacă se află sau nu în array

Ar putea exista cazul în care lungimea array-ului de 0 să fie un indicativ pentru a nu cauta în vector ( nu se poate găsi elementul dacă vectorul nu are elemente ). Acest lucru nu este specificat în mod explicit în cod, dar se poate deduce din conditia pentru iterarea de la i = 0 la i < array.length, care nu va fi niciodată îndeplinită dacă vectorul nu are elemente, deci s-ar întoarce în continuare rezultatul dorit ( -1 ).

* 1. Partiționarea fiecărei categorii în alternative:
* v

1. array.length = 0
2. array.length > 0

* start

1. start = 0
2. start != 0

* end

1. end = array.length
2. end != array.length

* search

1. search este pe prima poziție în array
2. search este pe ultima poziție în array
3. search este în array dacă are mai multe elemente
4. search este în array dacă e singurul element
5. search nu este în array dacă array are elemente
6. search nu este în array dacă array are lungimea 0

Din specificația de testare rezulta 2 \* 2 \* 2 \* 6 = 36 cazuri de testare. Pe de altă parte, unele combinații de alternative nu au sens si pot fi eliminate. Vom reduce numărul cazurilor de testare la 14.

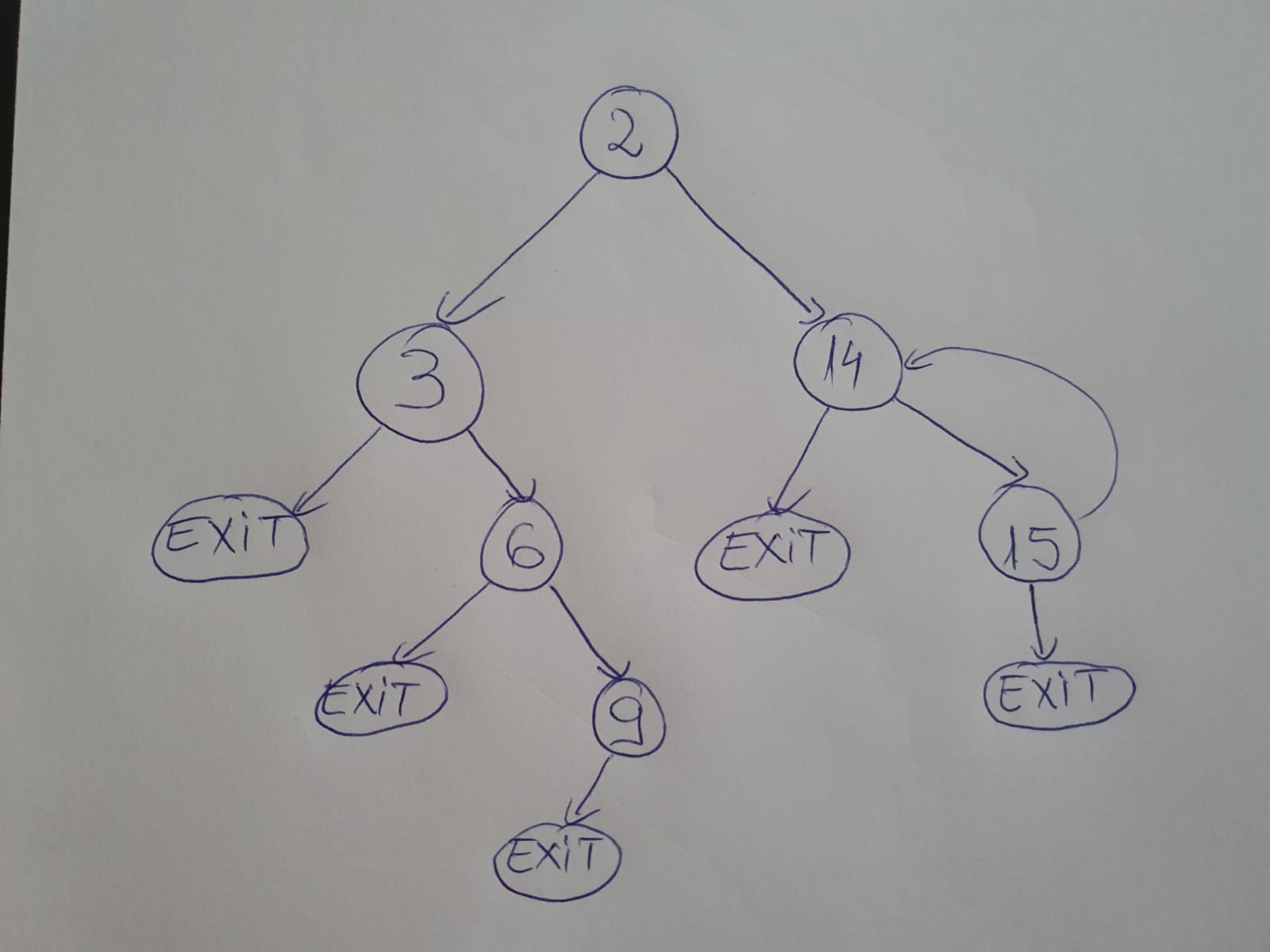
|  |  |
| --- | --- |
| **Date de intrare** | **Răspuns** |
| **T1 ({0,1,2,3}, -1, 2, 2)** | Error = invalid start and end |
| **T2 ({0,1,2,3}, -1, 3, 2)** | Error = invalid start |
| **T3 ({0,1,2,3}, -1, 4, 2)** | Error = invalid start and end |
| **T4 ({0,1,2,3}, 0, 2, 2)** | Error = invalid end |
| **T5 ({0,1,2,3}, 0, 3, 2)** | 2 |
| **T6 ({0,1,2,3}, 0, 4, 2)** | Error = invalid end |
| **T7 ({0,1,2,3}, 1, 2, 2)** | Error = invalid start and end |
| **T8 ({0,1,2,3}, 1, 3, 2)** | Error = invalid start |
| **T9 ({0,1,2,3}, 1, 4, 2)** | Error = invalid start and end |
| **T10 ({0,1,2,3}, 0, 3, 0)** | 0 |
| **T11 ({0,1,2,3}, 0, 3, 3)** | 3 |
| **T12 ({1}, 0, 0, 1)** | 0 |
| **T13 ({}, 0, -1, 1)** | -1 |
| **T14 ({0,1,2,3}, 0, 3, 99)** | -1 |

**Testarea structurală**

Programul în Java:



Graful programului este:



**Statement Coverage**

Pentru a realiza acoperirea la nivel de instrucțiune dăm următoarele teste:

|  |  |
| --- | --- |
| **Date de intrare** | **Răspuns** |
| **T1 ({0,1,2,3}, 1, 1, 2)** | Error = invalid start and end |
| **T2 ({0,1,2,3}, 0, 1, 2)** | Error = invalid end |
| **T3 ({0,1,2,3}, 1, 3, 2)** | Error = invalid start |
| **T4 ({0,1,2,3}, 0, 3, 2)** | 2 |
| **T5 ({0,1,2,3}, 0, 3, 99)** | -1 |

**Branch coverage**

Metoda testata este una foarte simpla, fără foarte multe posibile cai de urmat. Cum **branch coverage** este o extindere a metodei **statement coverage**, testele care se impun, pe lângă cele de mai sus, sunt cele care demonstrează corectitudinea programului atunci când bucla nu se executa nici măcar o data:

|  |  |
| --- | --- |
| **Date de intrare** | **Răspuns** |
| **T1 ({}, 0, -1, 99)** | -1 |
| **T2 ({0,1,2,3}, 0, 3, 99)** | -1 |

**Condition coverage**

Pentru a acoperi toate condițiile din setul de mai sus, folosim următoarea suită de teste:

|  |  |
| --- | --- |
| **Date de intrare** | **Răspuns** |
| **T1 ({0,1,2,3}, -1, 2, 2)** | Error = invalid start and end |
| **T2 ({0,1,2,3}, -1, 3, 2)** | Error = invalid start |
| **T3 ({0,1,2,3}, -1, 4, 2)** | Error = invalid start and end |
| **T4 ({0,1,2,3}, 0, 2, 2)** | Error = invalid end |
| **T5 ({0,1,2,3}, 0, 3, 2)** | 2 |
| **T6 ({0,1,2,3}, 0, 4, 2)** | Error = invalid end |
| **T7 ({0,1,2,3}, 1, 2, 2)** | Error = invalid start and end |
| **T8 ({0,1,2,3}, 1, 3, 2)** | Error = invalid start |
| **T9 ({0,1,2,3}, 1, 4, 2)** | Error = invalid start and end |
| **T10 ({0,1,2,3}, 0, 3, 0)** | 0 |
| **T11 ({0,1,2,3}, 0, 3, 3)** | 3 |
| **T12 ({1}, 0, 0, 1)** | 0 |
| **T13 ({}, 0, -1, 1)** | -1 |
| **T14 ({0,1,2,3}, 0, 3, 99)** | -1 |

**Complexitatea programului**

Formula lui McCabe pentru complexitate ciclomatică: Dat fiind un graf orientat conectat G cu e arce si n noduri, atunci numărul de circuite liniar independente este dat de:

V(G) = e - n + 1 unde :

G = graf complet conectat ( există o cale între oricare două noduri)

Circuit = cale care începe și se termină în același nod

Circuite liniar independente = niciunul nu poate fi obținut ca o combinație a celorlalte

Adăgăm 5 arce în graful de mai sus pentru a deveni complet conectat.

Atunci V(G) = 16 – 11 + 1 = 6

Circuite independente:

* 2 -> 3 -> exit -> 2
* 2 -> 3 -> 6 -> exit -> 2
* 2 -> 3 -> 6 -> 9 -> exit -> 2
* 2 -> 14 -> exit -> 2
* 2 -> 14 -> 15 -> exit -> 2

**Acoperirea la nivel de cale**

Putem a descrie programul ca o expresie regulată folosind nodurile grafului, conform Paige și Holtouse.

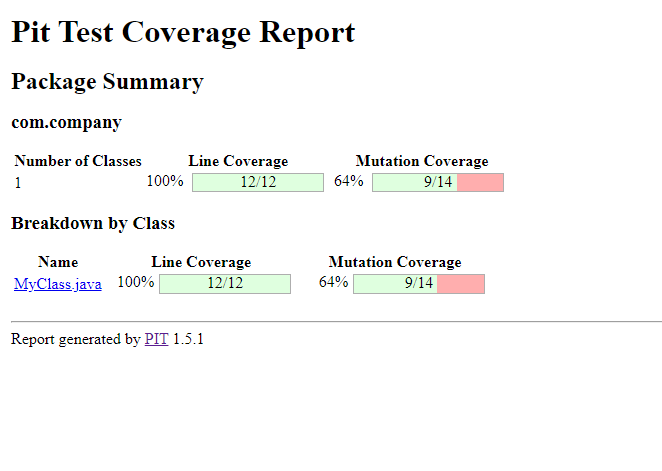
Pentru graful de mai sus, avem expresia regulată:

2.((3.(4+null).6.(7+null).9.(10+null))+null).14.(15.(16+null).14)\*.19

Numărul de căi este 5.

**Generator de mutanți**

Pentru generarea mutanților s-a folosit Pit.



**Concluzii și înlăturarea mutantilor**

După multe verificari, am observat ca mutantii generati nu pot fi reparati, indiferent de numărul de teste scrise.

Rezolvarea mutantilor poate fi făcută prin înlăturarea unei condiții sigure ( deci care nu ar fi trebuit să fie în cod de la bun început).

Acest lucru demonstrează eficacitatea testarii codului, dar și a testarii testelor: s-a observat o greșeala de design care poate fi îndreptată.

Tinand cont ca toate testele trec în continuare, code coverage-ul este de 100%, iar mutantii generati sunt omorâți în totalitate, putem modifica codul existent fără frica de a schimba comportamentul programului; avem astfel garantia unei testari corecte și eficiente a programului.

În acest mod, având încredere deplina în teste, scrierea și intretinerea acestora nu este un lucru care încetinește viteza de dezvoltarea a aplicatiilor; mai mult, aceste teste ofera siguranța și produsul poate fi livrat mai repede.

Se poate observa noua versiune a codului în imaginea de mai jos ( modificarile au fost comentate pentru a ilustra schimbarile ).

