VSTTE 2010 software verification competition

Problem 3: Searching a Linked List

I. Description and analysis

Problem:

Given a linked list representation of a list of integers, find the index of the first element that is equal to 0.

Properties:

You have to show that the program returns an index *i* equal to the length of the list if there is no such element. Otherwise, *i*'th element of the list me be equal to 0, and all the preceding elements must be non-zero.

Problem analysis:

For this problem, the linked list is represented as a sequence of integers. The difference between both is the indexation: the index of the first element in a list is zero while the index of the first element of a sequence is one.
A sequence of integers is define as a total function from the set 1..n (n a natural number) to the set of integers. Therefor the length of the list is n.
The ambiguity for this problem is that when the function returns *n* we can't deduce if the last element is equal to zero, of if none of the elements differs from zero.
We adapt the problem to our case: if the list has no element equal to zero, the function must return *i = n+1*.

From this adapted specification, two main functionalities are extracted:

FUNC1: If the sequence contains an element equal to zero, *i* is its index; otherwise *i = n+1*.
FUNC2: All the elements preceding the *i*'th are non-zero. (i.e. *i* is the index of the first element)

The mathematical expressions for such functionalities are:

FUNC1: Zero belongs to ran(list) => *i*: 1..NN and list(i)=0)
Zero does not belong to ran(list)) => *i* = n+1
FUNC2: !x.(x:1..*i*-1 => not(list(x) =0))

Architecture:

The two functionalities extracted are both dependent of *i*. The project will be made of one main machine containing the two functionalities: The most abstract functionality (FUNC1) is introduced in the abstract machine, the other one in the refinement.

II. Software modeling

Reminder: for the operations in the abstract machine, we have to data type the input and the output parameters, and specify the conditions (if any) needed to execute the operation in the PRE substitution. The THEN substitution specifies the state of the variables at the exit of the operation.

Acronyms:
- POG: Proof Obligation Generator
- PO: Proof Obligation

A. Machine CTX

As usual, the machine CTX is the machine containing all the constants of the project.

```
MACHINE
   CTX
CONCRETE_CONSTANTS
   list, NN
PROPERTIES
   NN: NAT1 &
   NN <= MAXINT-1 &
   list: 1..NN --> INT
END
```

The property NN <= MAXINT-1 is added because if the list does not contain any zero, the operation returns *i=n+1*. If n is MAXINT, there will be an overflow.

B. Machine List

This is the machine that contains the operation.

1. Abstract machine:

```
index <-- IndexNull =
PRE
   index : 1..NN+1
THEN
   IF(0: ran(list)) THEN index :: 1..NN
   ELSE index := NN+1 END
END
```

The FUNC1 functionality is introduced in this abstraction.
The case where list = {} is treated in the ELSE substitution, since zero is not in the range of the list; and we well have *i* = n+1 (NN = 0).

2. Refinement:

The FUNC2 functionality is introduced in the refinement, and details are added to the FUNC1 functionality.

```
index <-- IndexNull =
BEGIN
   index: (index : 1..NN+1                                 &
           !ii.(ii: 1..index-1 => not(list(ii)=0))         &
           (0:ran(list) => (index<=NN & list(index)=0))    &
           (not(0:ran(list)) => index = NN+1)              )
END
```

3. Implementation:

```
index <-- IndexNull =
VAR ind, temp IN
   ind := 1;
   temp := list(ind);
   WHILE(ind<=NN & not(temp=0)) DO
      ind := ind +1;
      IF(ind <= NN) THEN temp := list(ind) END
   INVARIANT
      ind: 1..NN+1 &
      (ind <= NN => temp = list(ind)) &

      //FUNC2 functionality
      !ii.(ii: 1..ind-1 => not(list(ii)=0)) &

      (#jj.(jj: ind..NN & list(jj)=0) => 0: ran(list)) &
      (!jj.(jj: 1..NN => not(list(jj)=0)) => not(0: ran(list)))
   VARIANT
      NN+1-ind
   END;
   index := ind
END
```

The local variable ‘temp’ is added because having ‘not(list(ind))’ in the condition of the loop generates a warning from the b0 checker.

Note that we have the FUNC2 property proved when a index *j* such that list(j)=0 is found.

Construction of the invariant:

The invariant is obtained by replacing 'index' by the index of the element currently read. Some details are yet added. Indeed, 'index' cannot be changed in the property 'list(index) = 0' of the refinement, this would add false hypotheses and the POG would generate wrong POs.
All we know is: "if there is an element equal to zero, at an iteration 'j' we will have list(j)=0.". Translation: An iteration j exists such that list(j)=0.

Construction of the variant:

- Since 0 ≤ ind ≤ NN+1, we have NN+1-index ≥ 0. The expression is positive.
- The variable 'ind' is incremented for each iteration in the loop, and NN is a constant. So the expression NN+1-index decreases for each iteration in the loop.

III. Proof obligations

Au total of 32 POs are generated for the Liste machine. The force 0 prover automatically discharges 24 POs, and the predicate prover (**p1**) discharges 5 of them.

3 POs remains to prove:

- 1 PO in relation with the refinement;
- 2 POs related to the conservation of the invariant.

One rule is added.

A. Functional proof obligations

- First PO:

```
…
"`Local hypotheses'" &
!jj.(jj: 1..NN => not(list(jj) = 0)) &                    ①
"`Check preconditions of called operation, or While loop
construction, or Assert predicates'"
=>
not(0: ran(list))
```

The only hypothesis we have means that for every element j in the domain of list, list(j) is never equal to zero. The Goal to prove is that zero is not in the range of the list.

The prover is not able to do the link between both the hypothesis and the goal, so we add the following rule:

```
binhyp(!j.(j : dom(l) => not(l(j)=0)))
=>
not(0: ran(l))
```

Add the only hypothesis we have in the stack of hypotheses, and apply an equality on hypothesis on 1..NN (in order to change 1..NN to dom(list) so we can apply the rule). Add the new hypothesis to the stack of hypothesis with the command **dd**.
Apply this rule (**ar**) on the current goal and the PO is discharged.

The tree of commands for this PO is:

```
dd(0)
    ah(①)
        eh(1..NN, _h, Goal)
            dd
                ar(Rule.1, Once)
```

- <u>Second and third PO:</u>

The next two POs have the same Goal, only one hypothesis changes. The demonstration can be applied on both.

```
index$1: 1..NN+1 &
"`Local hypotheses'" &
index$2: 1..NN+1 &
index$2<=NN => temp$0 = list(index$2) &
!ii.(ii: 1..index$2-1 => not(list(ii) = 0)) &
#jj.(jj: index$2..NN & list(jj) = 0) => 0: ran(list) &
!jj.(jj: 1..NN => not(list(jj) = 0)) => not(0: ran(list)) &
index$2<=NN &
not(temp$0 = 0) &
not(index$2+1<=NN) &
ii: 1..index$2+1-1 &
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
not(list(ii) = 0)
```

Those two POs are linked to the conservation of the invariant.
We start the demonstration by simplifying the hypothesis ①. Add the hypothesis ‘ii: 1..ind’, then add the hypothesis ① so we can prove it. Add the hypothesis ①_ to the stack of hypotheses.

The demonstration of the goal is made by cases. The first case where we prove the invariant is correct to this step (ii: 1..ind-1), and the second case where we prove the invariant is conserved (ii=ind).

<u>First case:</u>

Start the demonstration by cases using the command **dc**. Apply a deduction so we have the hypothesis of the case in the stack of hypotheses. Particularize the hypothesis ② for the variable ii and call the prover: this case is discharged.

<u>Second case:</u>

For this case, we first have to demonstrate that ii=ind. After adding this as a hypothesis, we have to prove it: add the hypotheses ‘not(ii: 1..ind-1)’ and ①_, then call the predicate prover. The goal becomes:

Not(list(ind)) = 0

We have ind+1<=NN so ind<=NN. We can do a modus ponens on the hypothesis ③ in order to obtain temp=list(ind). Helped by the hypothesis not(temp=0), the goal is proved.

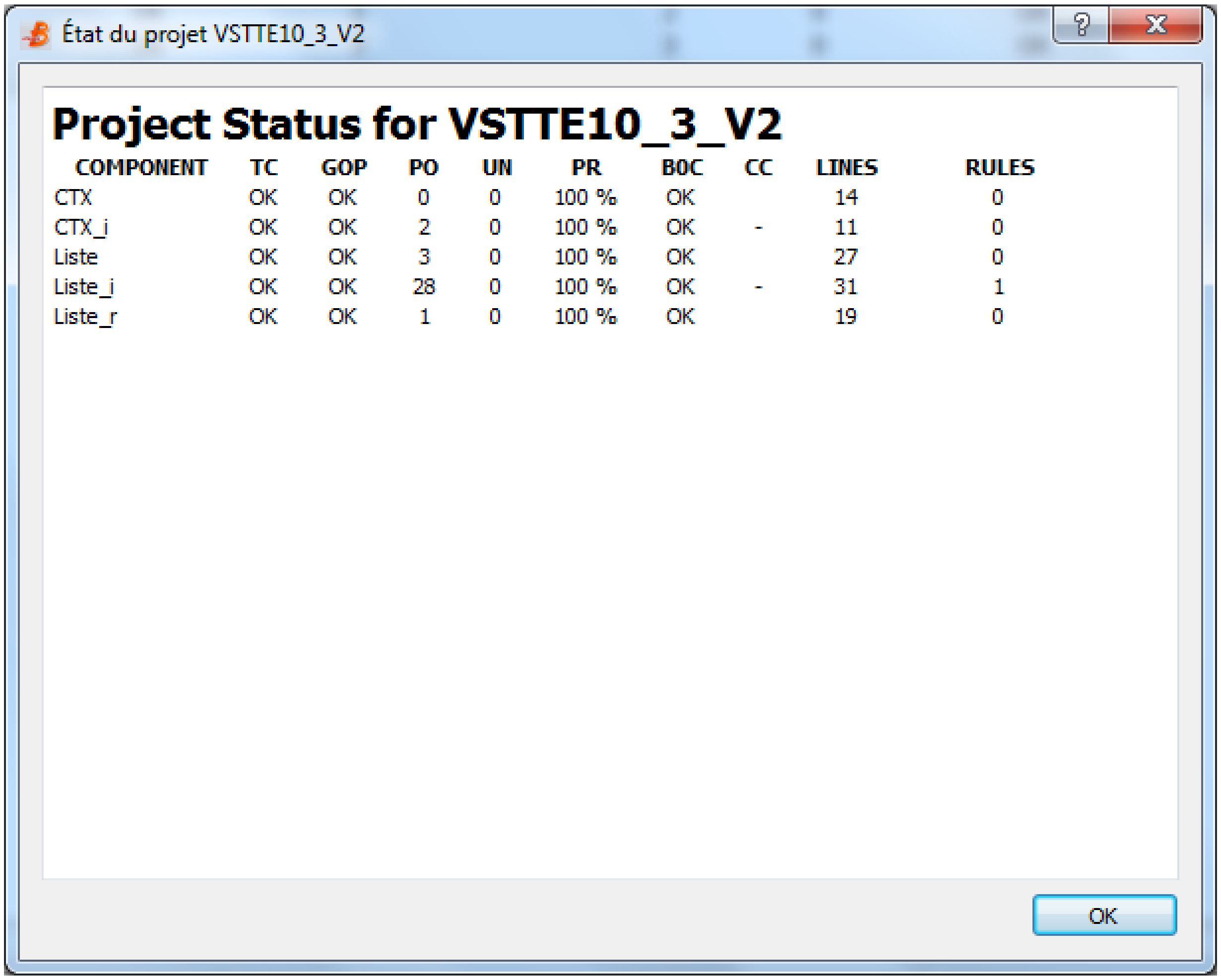
The tree of commands for the two POs is:

```
dd(0)
    ah(①_)
        ah(①)
            pp(rt.0)
        dd
            dc(ii: 1..ind-1)
                dd
                    ph(ii,②)
                        pr
                dd
                    ah(ii = ind)
                        ah(not(ii: 1..ind-1))
                            ah(①_)
                                pp(rt.0)
                        ah(not(temp = 0))
                            mh(ind<=NN => temp = list(ind))
```

B. Well-definedness proof obligations

A total of 6 POs of well-definednesss are generated. All the 6 are automatically discharged by the prover.

C. Project status



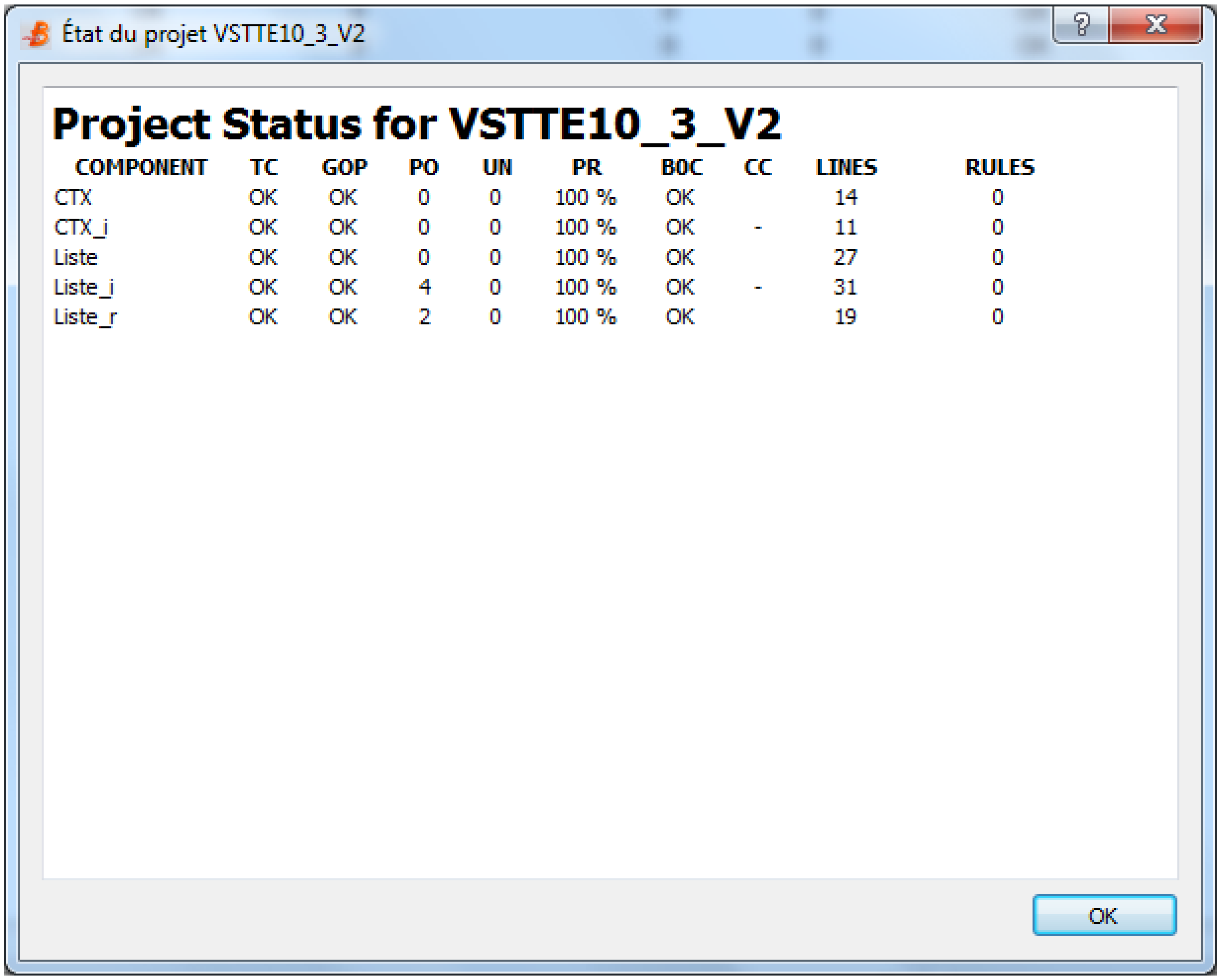
État du projet VSTTE10_3_V2

Project Status for VSTTE10_3_V2

COMPONENT	TC	GOP	PO	UN	PR	B0C	CC	LINES	RULES
CTX	OK	OK	0	0	100 %	OK		14	0
CTX_i	OK	OK	2	0	100 %	OK	-	11	0
Liste	OK	OK	3	0	100 %	OK		27	0
Liste_i	OK	OK	28	0	100 %	OK	-	31	1
Liste_r	OK	OK	1	0	100 %	OK		19	0

COMPONENT: the components (machines) of the project.
TC: Type Check.
GOP: Générateur d'Obligations de Preuve (the POG).
PO : Proof Obligation.
UN : The number of unproved PO in the component

PR: The rate of proved PO of the component.
B0C: B0 Check.
CC:
LINES: The number of lines in the component.
RULES: The number of added rules for the component.



État du projet VSTTE10_3_V2

Project Status for VSTTE10_3_V2

COMPONENT	TC	GOP	PO	UN	PR	BOC	CC	LINES	RULES
CTX	OK	OK	0	0	100 %	OK		14	0
CTX_i	OK	OK	0	0	100 %	OK	-	11	0
Liste	OK	OK	0	0	100 %	OK		27	0
Liste_i	OK	OK	4	0	100 %	OK	-	31	0
Liste_r	OK	OK	2	0	100 %	OK		19	0

Well-definedness status.

IV. Generated code

We can only generate the code in Ada because the lower bound of arrays in Ada is 1.
B0 check the project then generate the code (Project > Code Generator) with the List machine as the start point of the program (i.e. it could be the start point since it contains only one operation and it returns nothing).

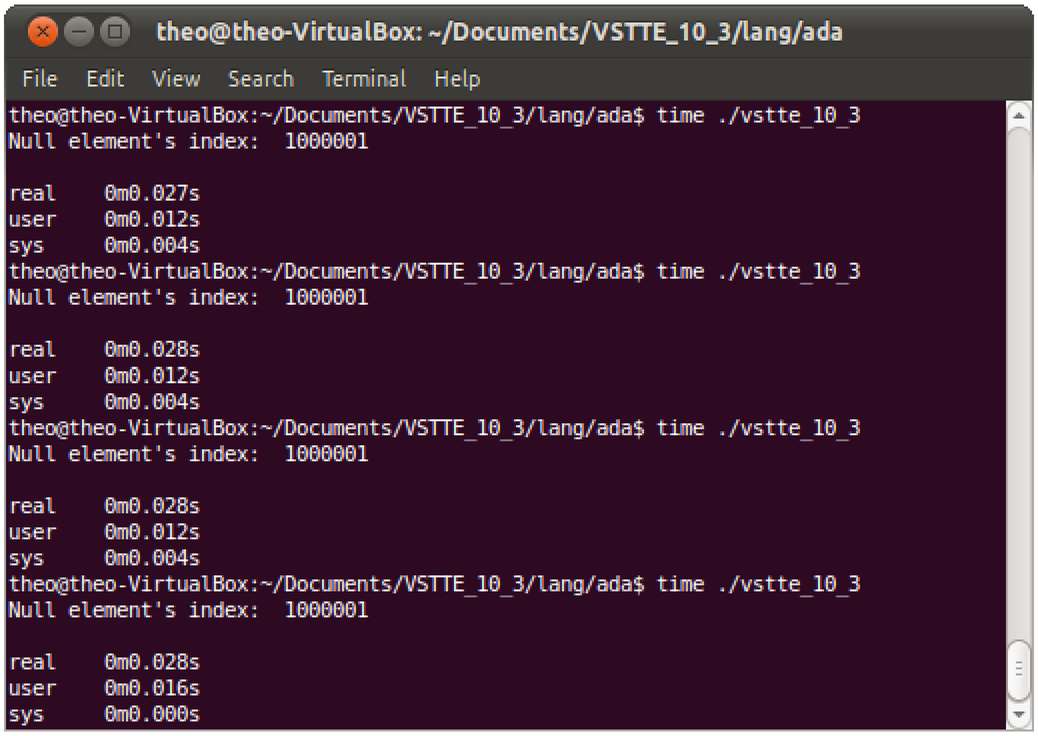
Edit the sets.ads and the ctx.bfl files in order to change the NN and the list'values.
Edit the list.adb file:

- Add "use Ada.Text_IO; with Ada.Text_IO;" before the 'package' keyword
- Insert "Put_Line(Integer'Image(this.index))" right after the exit of the loop in order to print the index of the first nul element in the array

Once the code is generated, under linux, go to the Project/lang/ada repertory and use the command:

```
gnatmake my_project
./my_project
```

The execution time is measured with the *time* command of linux.


theo@theo-VirtualBox: ~/Documents/VSTTE_10_3/lang/ada
File Edit View Search Terminal Help

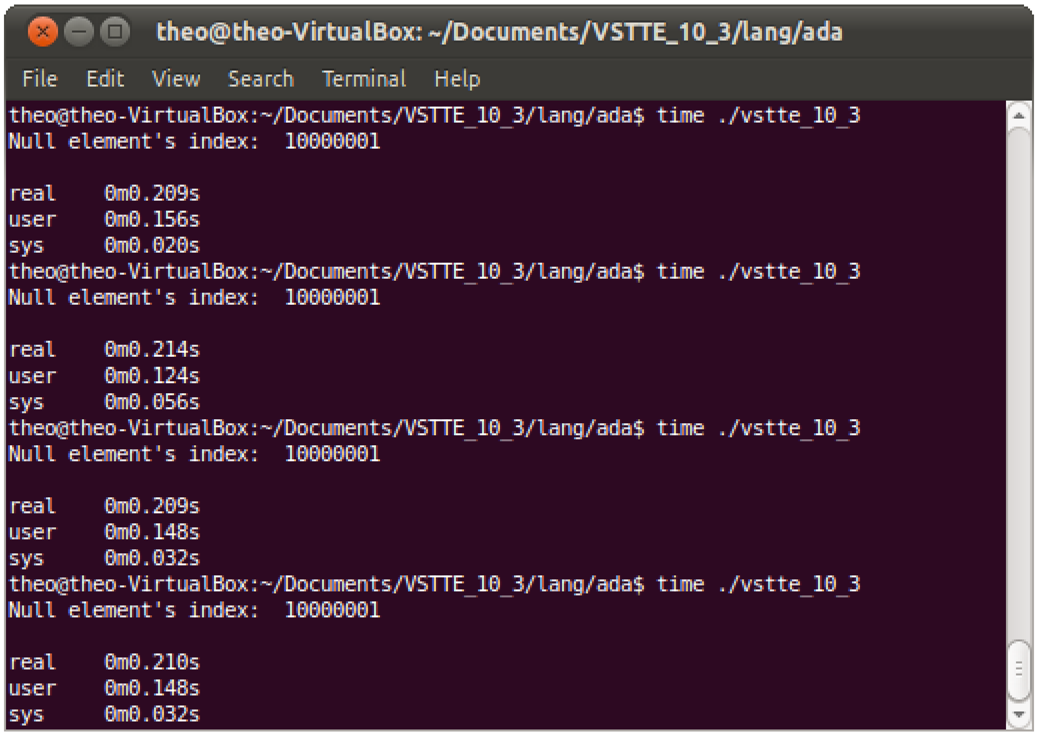
```
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  1000001

real	0m0.027s
user	0m0.012s
sys	0m0.004s
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  1000001

real	0m0.028s
user	0m0.012s
sys	0m0.004s
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  1000001

real	0m0.028s
user	0m0.012s
sys	0m0.004s
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  1000001

real	0m0.028s
user	0m0.016s
sys	0m0.000s
```


theo@theo-VirtualBox: ~/Documents/VSTTE_10_3/lang/ada
File Edit View Search Terminal Help

```
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  10000001

real	0m0.209s
user	0m0.156s
sys	0m0.020s
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  10000001

real	0m0.214s
user	0m0.124s
sys	0m0.056s
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  10000001

real	0m0.209s
user	0m0.148s
sys	0m0.032s
theo@theo-VirtualBox:~/Documents/VSTTE_10_3/lang/ada$ time ./vstte_10_3
Null element's index:  10000001

real	0m0.210s
user	0m0.148s
sys	0m0.032s
```