Final Report

Xiang Li
932-514-926

Accomplish:

During this term, I have learned how to use Template Scripting Testing Language tool to test python codes. At the beginning of this term, I was trying to understand the syntax of the TSTL. Through learning in this term, I can write TSTL code to test python files. The basic idea of the TSTL testing is writing actions. An action includes generating a bunch of random number and passing them into the functions that you want to test. Then I should write down the guard for the action. Guard is a way to check the properties of the function you test. So after an action takes place, the guard should be wrote after the symbol "=>". After understanding the syntax of the TSTL, I tried to test all the functions in a TSTL file.

My python file realizes the basic data structure which is single linked list. The linked list python file contains fourteen functions. They are insert, append, returnIndex, updateIndex, deleteIndex, insertBeforeIndex, insertAfterIndex, deleteData, deleteAlldata, insertAfterEveryData, insertBeforeEveryData, deleteList, copyList and findMthToLastNode functions. In the next section, I will briefly describe all the functions and illustrate how did I test them.

insert: Insert a data at the head of the list
1. length == PRE length + 1
2. Get the first data after inserting

append: Append a data at the end of the list
1. length == PRE length + 1
2. Get the last data after inserting

returnIndex: Return the index of a data you first meet
1. Insert a data at the beginning of the list, and return the index 0

updateIndex: Update the data at a specific index
1. Update a data at a index
2. Get the data at the index and compare the data with what I have updated

deleteIndex: Delete the data at the a specific index
1. length + 1 == PRE length

insertBeforeIndex: Insert a data after a specific index
1. length == PRE length + 1
2. Get the data at the index and compare the data with what I have updated

insertAfterIndex: Insert a data before a specific index
1. length == PRE length + 1
2. Get the data at the index + 1 and compare the data with what I have updated

deleteData: Delete the data you first meet
1. length + 1 == PRE length

deleteAllData: Delete all data you indicate

1. Count the number (a) of data in the list
2. length + a == PRE length

InsertAfterEveryData: Insert a data after every data you indicate
1. Count the number (a) of data you indicate
2. length == PRE length + a
3. the number of data you insert >= a

InsertBeforeEveryData: Insert a data Before every data you indicate
1. Count the number (a) of data you indicate
2. length == PRE length + a
3. the number of data you insert >= a

deleteList: Delete the list
1. the length of the list == 0

copyList: Copy a list to a
1. The length of the list == the length the list a
2. Two lists are equal

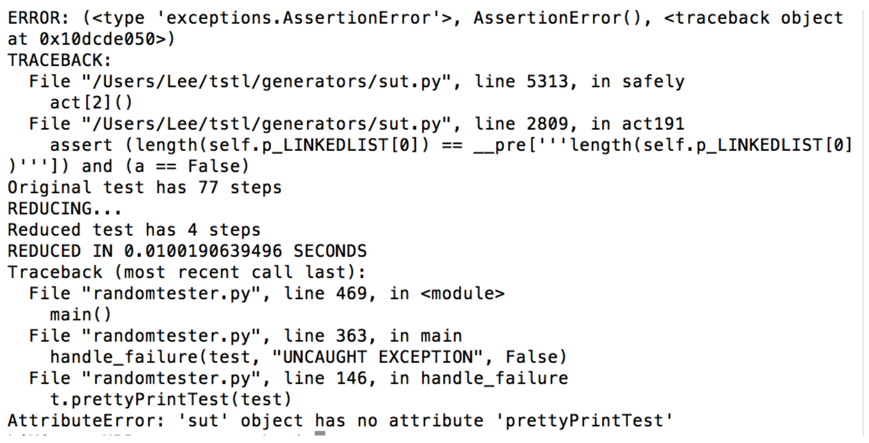
findMthToLastNode: Find a n^th data from the last
1. the index of the data == the length of the list – n

Bugs:

Now, I have tested all of them. However, I did not find any bug in the python. As far as I have accomplished, I think the LinkedList.py has a strong robustness. Although I could not find any bug in LinkedList.py with TSTL, I still recognized some definitions of functions were not very rigorous. Users may feel tricky when they use them. For instance: insertBeforeIndex() function. This function uses for insert a data before an index you indicate. If here is a list 1 → 2 → 3 → 4, it calls the function insertBeforeIndex(0, 0). The list is changed to 0 → 1 → 2 → 3 → 4. So at first, my TSTL guard is:

```
0 <= %INDEX% < length(%LINKEDLIST%) -> a = ~%LINKEDLIST%.insertBeforeIndex(%INDEX%, %VALUE%) => \
                                        (length(%LINKEDLIST,1%) - 1 == PRE%(length(%LINKEDLIST,1%))%) \
                                        and (a == True) and ( returnValue(%LINKEDLIST%, %INDEX%) == %VALUE%)
```

In the TSTL, the %INDEX% was larger equal than 0 and was less than the length of the list.The result is:



```
ERROR: (<type 'exceptions.AssertionError'>, AssertionError(), <traceback object
at 0x10dcde050>)
TRACEBACK:
  File "/Users/Lee/tstl/generators/sut.py", line 5313, in safely
    act[2]()
  File "/Users/Lee/tstl/generators/sut.py", line 2809, in act191
    assert (length(self.p_LINKEDLIST[0]) == __pre['''length(self.p_LINKEDLIST[0]
)''']) and (a == False)
Original test has 77 steps
REDUCING...
Reduced test has 4 steps
REDUCED IN 0.0100190639496 SECONDS
Traceback (most recent call last):
  File "randomtester.py", line 469, in <module>
    main()
  File "randomtester.py", line 363, in main
    handle_failure(test, "UNCAUGHT EXCEPTION", False)
  File "randomtester.py", line 146, in handle_failure
    t.prettyPrintTest(test)
AttributeError: 'sut' object has no attribute 'prettyPrintTest'
```

As the result shown in the terminal, the insertBeforIndex function had a bug. Then I tested in the python file. When I inserted a data 9 into index 4 in the 1 → 2 → 3 → 4 by calling insertBeforeIndex(4, 9). The result was 1 → 2 → 3 → 4 → 9. So this function can insert a data at the end of the list. However, the list did not have index 4. This tricky would cause confusing. At least, I changed my TSTL to:

```
0 <= %INDEX% < length(%LINKEDLIST%) -> a = ~%LINKEDLIST%.insertBeforeIndex(%INDEX%, %VALUE%) => \
                                        (length(%LINKEDLIST,1%) - 1 == PRE%(length(%LINKEDLIST,1%))%) \
                                        and (a == True) and ( returnValue(%LINKEDLIST%, %INDEX%) == %VALUE%)
```

The %INDEX% was larger equal than 0 and was less equal than the length of the list. Then the test passed eventually. This issue also appeared on the functions insertAfterIndex and findMthToLastNode. Although this is not a bug, it just a logical mistake. I would like to suggest the author to change the logic definition a little bit, which will give us a clear understanding.

Tester's work

TSTL as a tool gives a convenience way to test python codes. It randomly generates thousands of data you designed and passes them to functions that can help us test many boundaries we can not realize. With the tester what I wrote, I did not find any bug in the python. However, some of my TSTL guards to test the properties of the functions are very simple. For example, when I tested the deleteIndex function my guard was:
(length(%LINKEDLIST,1%) + 1== PRE%(length(%LINKEDLIST,1%))%)
I just compared the length of before and after inserting a data. I thought that if I compare two lists before and after inserting, I had to write a new function. This helping function did not like my other helping functions. It was very complex. I have confusion about what if my helping functions had bug too? Generally speaking, my TSTL tester covers all the situations that would cause bugs. It works very well.

For the python file LinkedList.py, I could not find any bug. With the result of the testing, the python considered many boundaries. I think that the author's python has a very strong robustness.

Wrong with TSTL:

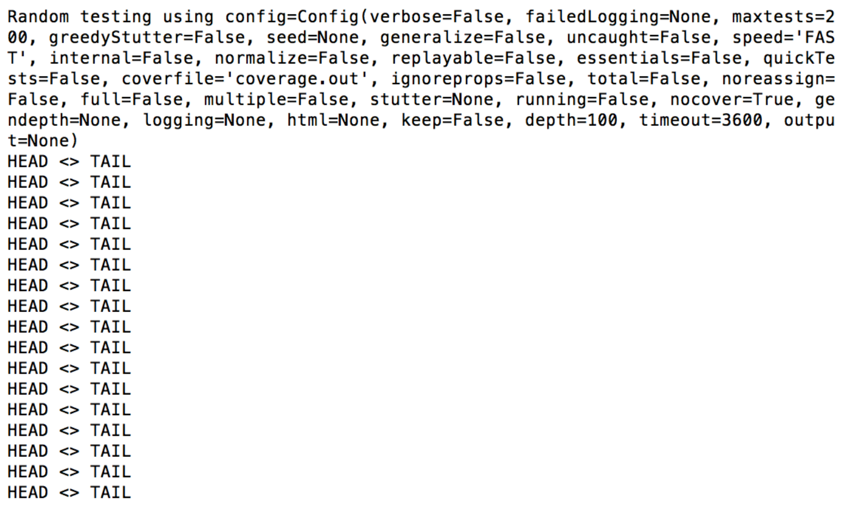
As far as I know, TSTL can generate many of the conditions in testing. For the most of time, TSTL works very well. But I still found a bug during my testing. When I tested the function returnIndex, I tried to generate a number, which was not in the list. My data range in the list is from 1 to 20. I hoped TSTL can generate a number that was greater than 20. So at first I wrote these codes:

```
Pool %NUMBER% 1
%NUMBER% := [25]
a = %LINKEDLIST%.returnIndex(%NUMBER%) => ( a == None )
```

These codes meant that I generated a constant number 25 in the pool and passed it to the returnIndex function. But the result was TSTL could not generate a constant number. In order to verified the TSTL could not generate a constant number, I tried to insert a constant number into the liked list.

Pool %NUMBER% 1
%NUMBER% := [25]
a = %LINKEDLIST%.insert(%NUMBER%) => (a == None)
print %LINKEDLIST%

The printed result shown that the list was empty. TSTL did not insert a constant number into the list successfully. The screen shot shows the result:



```
Random testing using config=Config(verbose=False, failedLogging=None, maxtests=2
00, greedyStutter=False, seed=None, generalize=False, uncaught=False, speed='FAS
T', internal=False, normalize=False, replayable=False, essentials=False, quickTe
sts=False, coverfile='coverage.out', ignoreprops=False, total=False, noreassign=
False, full=False, multiple=False, stutter=None, running=False, nocover=True, ge
ndepth=None, logging=None, html=None, keep=False, depth=100, timeout=3600, outpu
t=None)
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
HEAD <> TAIL
```

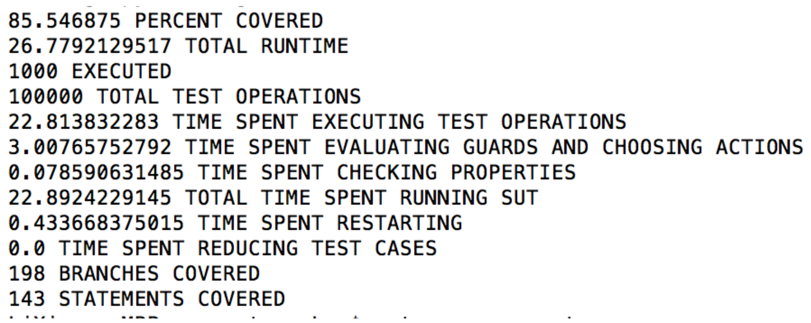
As we can see, the list does not contain any data. So for the expressions are shown above, at that condition, TSTL can not generate a constant number.

Good about TSTL:

To my opinion, I think the most important and good thing for TSTL is TSTL can generate a lot of random data and can pass them to the tested function. The only thing that testers should do is initialing data pool, passing data to functions, and writing the guards to check the properties of each function. TSTL can do the testing for testers spontaneously. TSTL can test many boundaries if you wrote right guards. So the test effort of test is very big.

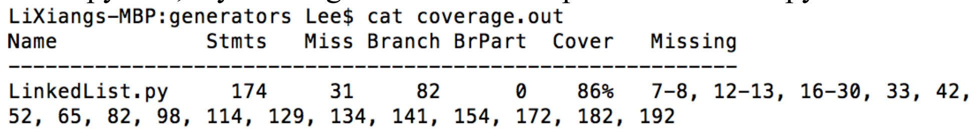
Coverage Summaries:

For the final testing, my test effort is 1000 times testing:
python randomtester.py --maxtest=1000
The final result is:



```
85.546875 PERCENT COVERED
26.7792129517 TOTAL RUNTIME
1000 EXECUTED
100000 TOTAL TEST OPERATIONS
22.813832283 TIME SPENT EXECUTING TEST OPERATIONS
3.00765752792 TIME SPENT EVALUATING GUARDS AND CHOOSING ACTIONS
0.078590631485 TIME SPENT CHECKING PROPERTIES
22.8924229145 TOTAL TIME SPENT RUNNING SUT
0.433668375015 TIME SPENT RESTARTING
0.0 TIME SPENT REDUCING TEST CASES
198 BRANCHES COVERED
143 STATEMENTS COVERED
```

As I have mentioned in the assignment part 3, my perspective of coverage was 80 percent. I hoped I could reach that goal when I finished the testing. After testing all the functions in the python, my coverage has reached 86 percent of whole python.



```
LiXiangs-MBP:generators Lee$ cat coverage.out
Name            Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------
LinkedList.py     174     31     82      0    86%   7-8, 12-13, 16-30, 33, 42,
52, 65, 82, 98, 114, 129, 134, 141, 154, 172, 182, 192
```

The picture shows the cover and missing code of my testing. After checking the missing codes, I found that all the missing codes are either the declaration of the functions or the initial functions that I could not call. So I think I have covered all the branches of the functions. 88 percent coverage is a satisfied number.