Final Project Report

Name: Akshay Salvi

ID: 932705076

Class: CS- 562

Introduction:

In my midterm report I submitted my preliminary analysis after performing tests on the binary_search_tree.py library. I had addressed most of the functions in that library. In this report I have included results after thoroughly testing the binary_search_tree.py library along with stack.py, queue.py and singly_ linked_list.py libraries. I have tested these 4 libraries on the basis of how well the developer has designed the SUT and analyze if there are any functional flaws in them. I have cloned the libraries from the below links:

Binary Search Tree:
https://github.com/agroce/cs562w16/blob/master/projects/patelra/binary_search_tree.py

Stack:
https://github.com/agroce/cs562w16/blob/master/projects/patelra/stack.py

Queue:
https://github.com/agroce/cs562w16/blob/master/projects/patelra/queue.py

Singly Linked List:
https://github.com/agroce/cs562w16/blob/master/projects/patelra/singly_linked_list.py

1. **What did I accomplish?**

    a. **Binary Search Tree Library.**
    This is a data structure in which the element at the left side of the root should be smaller than the root and the element at the right side should be greater than the root. In testing this library I have written 7 TSTL test cases. I have tried to maximize the complexity of the test case to ensure that TSTL is able to correctly test the SUT. Some of the tests that I have added after the midterm report include using variables to compare values arriving from the test statements.

Below are the new test cases that I have added

1. <btree>.put(<key>,<value>) ; temp = <btree,1>.min_key() ; temp1 = <btree,1>.delete_min(); (temp == temp1)

In this tstl test statements I used a variable to store the value of the minimum key and then asserted it with another variable having the value returned after deleting the smallest key.

2. <btree>.put(<key>,<value>) ; temp = <btree,1>.min_key() ; temp1 = <btree,1>._min_node() ; temp == temp1.key

Here I inserted a key-value pair into the binary tree and then stored the value of the minimum key in one variables and the value of the minimum node in the other variable. While asserting I extracted the value from the key stored in one variable and compared it with the variable containing the smallest node.

3. (<btree>.is_empty() == True) -> <btree,1>.put(<key>,<value>) ; <btree,1>.delete(<key,1>) => (<btree,1>.is_empty() == True)

I further checked if the SUT has handled the deletion of a key properly or not.

Below are the coverage results from my tests.

104 BRANCHES COVERED
72 STATEMENTS COVERED

Name	Stmts	Miss	Branch	BrPart	Cover
binary_search_tree.py	169	97	80	7	**44%**

b. Stack library

This data structure follows the LIFO rule; meaning last in first out. In this library I have covered all the functions defined in the library. Below are the test cases:

1. <stack>.add(<data>) -> not <stack>.is_empty() => <stack,1>.remove()

   This is a simple test case where an element is pushed on to the stack and later it is removed after checking that the stack is not empty.

2. <stack>.add(<data>) -> (not <stack>.is_empty()) -> temp = <stack,1>.stack_list; value = temp[len(temp)-1]; pop = <stack,1>.remove() => value == pop

   Initially I pushed an element on the stack and after checking that the stack is not empty, I used a variable to store the stack list and then access the same element of the array that

was just pushed and asserted it with the element that was obtained using the pop operation.

3. <stack>.add(<data>) => (<data,1> in <stack,1>.stack_list) and (<stack,1>.size() == (pre<(<stack,1>.size())>+1))

   Here I tried to use almost all of the functions in a single test statement to check whether the TSTL is able to handle complex statements and also if the quality of the SUT is good. Here, after adding an element onto the stack the test statement checks whether adding an element has increased the size of the stack by 1 or not.

   Below is the coverage obtained :

   4 BRANCHES COVERED

   2 STATEMENTS COVERED

Name	Stmts	Miss	Branch	BrPart	**Cover**	Missing
stack.py	12	10	0	0	**17%**	12-15, 18, 25-45

c. Queue Library

This data structure follows the FIFO rule; meaning the element which is pushed first will be popped first. In this library I have tried to thoroughly test all the function in the library. Below are some of the test cases:

1. <queue>.add(<int>) -> not <queue>.is_empty() -> <queue,1>.remove() => <queue,1>.size() == pre<(<queue,1>.size())>

   Here after adding an element onto the queue and later removing, it we check whether the size of the queue gets properly updated or not.

2. <queue>.add(<int>) -> <queue,1>.remove() => (<queue,1>.is_empty() == True)

   Here after successful operations of adding an element and later removing it from the queue, we check whether the queue remains empty or not.

   Below are the coverage results as obtained for this library:

4 BRANCHES COVERED

2 STATEMENTS COVERED

Name	Stmts	Miss	Branch	BrPart	**Cover**	Missing
queue.py	13	11	0	0	**15%**	13-19, 22, 29-49

d. Singly Linked List Library

In this data structure, the data is an ordered set of elements each having a link to its successor. Below are some of the test cases I have tried while testing this library.

1. <link>.add(<data>) -> (<link,1>.size != 0) => <link,1>.remove(<data,1>) -> (<link,1>.size == 0)

   Here I have tried to check that after adding and then subsequently deleting an element from the linked list does the size of the linked gets properly updated or not.

2. <link>.add(<data>) -> l = <link,1>.size -> <link,1>.add(<data>) -> new = <link,1>.size -> (len == new-1)

   Similarly here I tried adding two elements on to the linked list and then checked whether the size of the linked list after addition of every element gets properly updated or not.

   Below is the coverage obtained:

   13 BRANCHES COVERED

   9 STATEMENTS COVERED

Name	Stmts	Miss	Branch	BrPart	**Cover**	Missing
singly_linked_list.py	45	36	10	0	**16%**	14-16, 20-26

2. Bugs found in the libraries

Below is the report on bugs found in the libraries under test:

a. **Binary Search Tree:** There were no bugs found in the library and hence the quality of the SUT is very good. I thoroughly tried to test the library but still did not find any bug.
b. **Stack :** There was one bug that I came across during the testing that when I try to delete an element from an empty stack, the SUT does not handle that case correctly.
c. **Queue:** Same was noticeable about queues that when deletion is performed on an empty queue, the SUT fails to handle such a condition.
d. **Linked List :** I tested all the functions thoroughly and did not find any bugs in the library. The quality of the SUT is good.

3. How well the tester works:

The tester works quite well. Whenever I make certain errors while writing the test cases, TSTL points it out quite well, which is very helpful. Also TSTL is able to handle complicated test cases as well as large data sets well. The randomtester.py does not lag even if the range specified by the tester is quite large. Overall the tester works quite well.

4. What is wrong with TSTL

Firstly, TSTL does not have tutorials available about its syntax, working and hence learning TSTL can be a hindrance. Also TSTL is only confined to python right now, it would be great if TSTL would also be able to test code from JAVA or C++ . Apart from this, I also noticed that TSTL has a tendency to perform unneeded runs, which in a way is good but it has a performance hit on the working of TSTL eventually. Another thing I found about TSTL is code coverage, where even if I tried to test all the functions in the library the code coverage won't reflect it.

5. What is good about TSTL

One of the best features I feel about TSTL is that it accepts a range of values to test within your test statements as inputs. Another good about TSTL is that it has an easy to understand syntax and hence people who have sparse knowledge about python are also able to understand quickly. Since it is built on python it is very easy to build helper functions and test whether the required goal is achieved or not. Apart from random testing, TSTL also supports performance and regression testing.