Preliminary Assignment of Helsinki Master Program in Computer Science

Hazem Mohamed Safwat Mesilhy

February 23, 2016

Application number : UAF1604997

Abstract

The Preliminary Assignment was discussing Amortized Analysis as a time complexity measurement method. First task require implementing tree and calculate the number of steps required to finish in-order traversal. While, the second task was handling stack structure and the complexity of pop and push operations. In order to complete this tasks, I used Python 2.7 to implement and solve the problems.

1 Task I

To make a program that randomly generates trees in range of 50- 100 items.

To test the theory given in the assignment that In-order traversal has O(n-1) complexity. with (n-1) calls will cost 2(n-1)

1.1 Results

The results is as shown in the following Table

no.	number of items	number of calls	number of steps
—	—	—	—
0	75	74	148
1	60	59	118
2	59	58	116
3	84	83	166
4	78	77	154
5	60	59	118
6	73	72	144
7	83	82	164
8	92	91	182
9	58	57	114
10	72	71	142
11	90	89	178
12	95	94	188
13	57	56	112
14	76	75	150
15	52	51	102
16	68	67	134
17	64	63	126
18	80	79	158
19	61	60	120
20	74	73	146
21	85	84	168
22	53	52	104
23	80	79	158
24	86	85	170
25	71	70	140
26	99	98	196
27	53	52	104
28	66	65	130
29	79	78	156
30	98	97	194
31	82	81	162
32	90	89	178
33	92	91	182
34	96	95	190
35	69	68	136
36	94	93	186
37	82	81	162
38	50	49	98
39	52	51	102
40	54	53	106
41	80	79	158
42	52	51	102
43	82	81	162
44	60	59	118
45	67	66	132
46	66	65	130
47	64	63	126
49	92	91	182

Table 1: Results from In-order traversal. Shown number of iteration, number of items and number of calls

Results are matching the theory and the output of the program showed that, the number of calls is n-1. Hence, the total number of steps is 2(n-1) and the complexity is 2(n-1) as Amortized analysis predict. As, every call cost 2 (2 traversal each time). The worst case is so pessimistic with $O(n^2)$ complexity, but amortized analysis show a real complexity of 2(n-1) for (n-1) calls, which is lower than worst case with a respectful amount.

1.2 Implementation

The implementation of my solution is a Python class called BST.py containing the tree structure and In-order function and main.py to generate trees and print the results table.

All source codes can be found on my account on github: `https://goo.gl/EfX2Wa`

1.2.1 Tree Class BST.py

```
__author__ = 'Hazem Safwat'
class BST :
    def __init__(self , root):
        self.left = None
        self.right = None
        self.root = root
    def left(self):
        return self.left
    def right(self):
        return self.right
    def setval(self,val):
        self.root = val
    def getval(self):
        return self.root
    def insert(self,item):
        if (item < self.getval()):
            if (self.left != None):
                self.left.insert(item)
            else:
                self.left = BST(item)
            else:
                if (self.right != None):
                    self.right.insert(item)
                else:
                    self.right = BST(item)
    def inorder (tree):
        if tree != None:
            if tree.left !=None:
                BST.inorder(tree.left)
```

```
                print(tree.getval())
                if tree.right !=None:
                    BST.inorder(tree.right)
        def stepCounter (tree , counter=0):
            if tree != None:
                if tree.left !=None:
                    counter=BST.stepCounter(tree.left , counter+1)
            if tree.right !=None:
                    counter=BST.stepCounter(tree.right , counter+1)
            return counter
```

1.2.2 main.py

```
__author__ = 'Hazem Safwat'
from BST import BST
import random
from tabulate import tabulate
table = [[]]
for i in range(51):  #creating 50 different trees
    size = random.randint(50,100)
    items = random.sample(range(1,12000) , size) #random
        items in tree has a size = size
    tree = BST(items[0])    #Root of the tree
    for j in range(1,size):     #inserting items in the
        tree
        tree.insert(items[j])
    counter = BST.stepCounter(tree)
    table.append([i , size , counter])
print(tabulate(table[1:51] , headers= ["no." , "number of
    items" ,"number of steps" ]))
```

Please refer to This github repository for better visual and formatting.

2 Task II

Amortized analysis over a stack operation $m_1$pushes and $m_2$pops

Assumption:

- Stack is initially empty.
- pop(k) preform a multi-pop k times.

Using Aggregate method:

If the stack is empty initially.

then $m_1 = n$, where n is the stack length.

so, $m_1$calls to push(x) cost m_1.

If pop(k) use multi pops k times then the max number of pops is n. lets assume that pop(k) cost m_2n where $m_2$is

In the worst case, $m_2n \leq m_1$. as we can't pop more than the number of elements in the stack m_1.

thus, $m_2n = m_1$and $m_1 = n$

so, the complexity now $= 2m_1 = 2n$ for the n operations.

single operation $= \frac{2n}{n} = 2$.

$\therefore$ *actual cost* 2.

Same will be valid if pop(k) is preforming pop for the item with index k as n pushes cost n and pop(k) in worst case will cost n as well, the total cost will be 2n and for single operation that will cost 2 as previousally mentioned.

3 References

- http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm
- https://www.youtube.com/watch?v=UYcWpldlX-o
- https://www.youtube.com/watch?v=B3SpQZaAZP4

3.1 References for task I

- https://docs.python.org/2/library/random.html
- http://effbot.org/pyfaq/how-do-i-generate-random-numbers-in-python.htm
- http://www.pythonschool.net/data-structures-algorithms/binary-tree/
- https://pypi.python.org/pypi/tabulate

3.2 References for task II

- http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStackinPython.html
- http://www.cs.toronto.edu/~krueger/cscB63h/lectures/tut08.txt

Source codes are available on My github repository (link)

my github account : `https://github.com/hazemIII`