University of Helsinki

Computer Science Department

PreAssignment 2016

Author:
Andres Medina
(UAF 1608168)

February 29, 2016

Contents

Chapter 1

Preassignment - Task 1

1.1 Introduction

The first task of the preassignment consists in experimentally test that the amortized complexity of $(n-1)$ calls to **inorder_next** in a binary tree is $2(x-1)$.

To achieve this task, the following strategy was followed:

1. First, a program that generates random Binary Trees was created. This program makes possible to have a test set in which the current hypothesis could be proved.
2. After creating the random Binary Tree generator, a second program that allows to transverse the binary tree in inorder was developed *(inorder means, first visiting the left subtree, then the root and finally the right subtree)*. The developed program also returns the amount of steps involved in this operation.
3. By using both programs, a table that compares the amount of steps taken to transverse every binary tree in comparison with the number of nodes of each tree was made.
4. Finally, the data of the table was plotted. By using linear regression, the equation corresponding to the slope of that line plot was obtained. The expected value of the slope of that line plot should be something bigger or equal to $2(x-1)$ *(it should be bigger or equal because the amortized complexity correspond to an upper bound of the actual complexity of an algorithm)*.

The results obtained in the analysis are described in the results section of this paper.

1.2 Materials and Methods

In this section, the tools and the logic used to solve every single step that composed the task 1 of this preassignment are described. This section include topics like the programming framework used to solve this task and also what was the logic applied to develop those programs.

1.2.1 Programming Tools

For developing the required programs the following tools were used:

- Python 3.5.1 as the programming language
- Github as a repository tool
- Sublime Text as the code editor

In addition, the random library of python was used for accomplish the tasks related with randomness.

1.2.2 Programming Logic

The two programs involved in this task were programmed by using OOP paradigm. Considering the fact that the task 1 of the preassignment involves the creation of two different programs, the logic of each program is explained in a different subsection.

Creating a Random Binary Tree

The creation of the binary tree was implemented by developing two different classes: a binary node class and a binary tree class. The binary node class contains 3 different attributes, an Id, a reference to a left child node and a reference to a right child node while the binary tree class contains only two attributes, a list of binary nodes and the total cost of traversing the tree in inorder. By using this two classes, creating a random binary tree was possible. Also when every binary tree is created, a random tree setting method is called so that every tree created has a different and also random structure.

The random tree setting method work as follows:

- First, a integer number between 50 and 100 is choose randomly. This number will represent the number of nodes of the corresponding binary tree
- Second, a left-right flag is set by choosing a random integer number between 0 and 1
- Then, after the left-right flag is set, a random node of the tree is chosen. Only the nodes that doesn't already have the corresponding child are candidates for this random choice. *For example, if the left-right flag is set to left, only nodes that doesn't have left childs are possible candidates of this random choice. Conversely, if the left-right flag is set to right, only nodes that doesn't have right childs are possible candidates*
- Finally, the corresponding child is created in the randomly chosen node of the tree

The described logic repeats until the number of nodes of the binary tree is reached.

Traversing a Binary Tree in inorder

The traversing of a binary tree in inorder can be made mainly in two different ways: by using recursion and by not using recursion. In this case, the traversing function was implemented using recursion because by using recursion was possible to achieve a smaller and cleaner code and also allows to avoid the implementation of a link between a child node and its parent node.

The idea of this implementation is mainly the following:

- Start by applying this method to the root node of the tree
- If the root node has a left child, then the traverse_inorder method is called again but this time with the left child node as an argument to the method
- When the corresponding node has no left child node, then the program ask if the node has a right child. If the node has a right child, then the traverse_inorder method is called again but this time the right child node is passed as an argument to the method

- After no more left and right child are found, the traverse_inorder method returns. Of course a logic for counting the corresponding steps in each call of the method is considered so that when the method ends, the total amount of steps is already calculated.

In the next section the results of the above implementations and their analysis are shown.

1.3 Results

When the two generated programs are run, an output table is created. This table includes three columns:

1. **Binary Tree ID** Represents the Id of the binary tree
2. **Number of nodes of the Tree** Represents the number of nodes of the binary tree
3. **Total steps of inorder traverse** Represents the amounts of steps involved in the inorder traverse operation

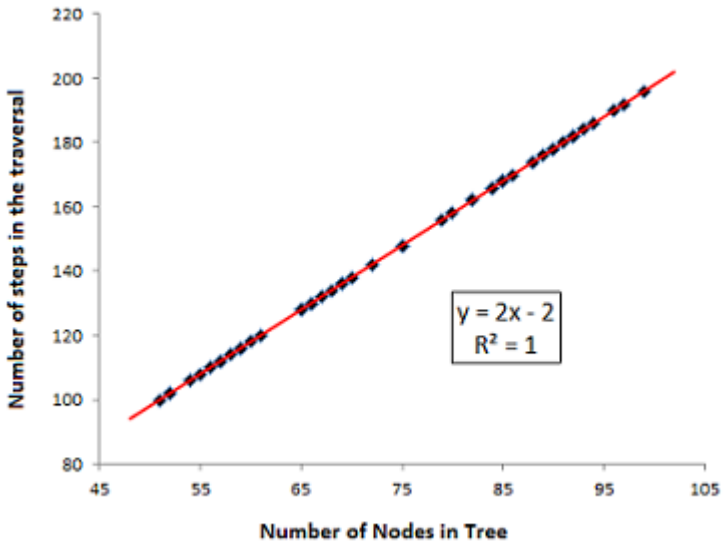
The results obtained in this computation are shown in the following table:

Table 1.1: Binary Tree traverse results using 50 different trees

Tree Id	Number of Nodes	Total Steps	Tree Id	Number of Nodes	Total Steps
1	83	164	26	70	138
2	94	186	27	95	188
3	69	136	28	68	134
4	99	196	29	88	174
5	53	104	30	98	194
6	63	124	31	99	196
7	72	142	32	63	124
8	61	120	33	66	130
9	55	108	34	59	116
10	90	178	35	94	186
11	81	160	36	85	168
12	74	146	37	59	116
13	92	182	38	68	134
14	85	168	39	93	184
15	75	148	40	62	122
16	66	130	41	75	148
17	95	188	42	100	198
18	64	126	43	92	182
19	71	140	44	82	162
20	95	188	45	84	166
21	52	102	46	58	114
22	77	152	47	63	124
23	53	104	48	69	136
24	84	166	49	51	100
25	80	158	50	66	130

By inspecting the results of the table clearly can be seen that the amount of steps that takes to traverse a binary tree, directly depends of the amount of nodes that compose that particular binary tree.

To obtain the equation that describes the relation between the number of nodes and the amount of steps that takes to traverse a binary tree, the data in the previously shown table was plotted. In addition, linear regression was used to obtain the desired equation. The result of this linear plot is shown in the following image:



As could be seen, the value of the **coefficient of determination** also know as R^2 is 1. This means that the linear regression line perfectly fits the plotted data. The equation that describes this relation is correspond to $y = 2x - 2$ where x represents the number of nodes in a binary tree and y represents the amounts of steps that takes to traverse that particular tree in inorder.

The meaning of this result will be discussed in the next section of the text.

1.4 Discussion

The amount of steps that takes to traverse a random binary tree in inorder is $2x - 2$ where x represents the number of nodes of that particular tree. In this section, this result will be used to test experimentally that the amortized complexity of $(n - 1)$ calls of **inorder_next** correspond to $2(x - 1)$.

To do this task, the following information was considered:

- First, the amortized complexity is an upper bound on total actual complexity. *This means that the amortized complexity is always bigger or equal than the total actual complexity*

- Second, the operation of traverse an entire binary tree with n nodes is exactly the same than call the **inorder_next** function n times
- Third, if n calls are made to a function, the actual complexity of this operation will always be bigger or equal than the actual complexity of $n-1$ calls to the same function

Using the above information and also considering the results obtained in the last section, its possible to state the following:

- The actual complexity for n calls of **inorder_next** can be represented by the equation $2(x-1)$. *where x represents the number of nodes in a binary tree*
- Considering that $n-1$ calls to a function is les expensive than n calls in the same function (in terms of complexity), its possible to state than $2(x-1)$ **is an upper bound of the actual complexity of $n-1$ calls**. *This state is based in the fact that $2(x-1)$ will always be bigger or equal than the actual complexity of $n-1$ calls*
- Finally as $2(x-1)$ is an upper bound of the actual complexity of $n-1$ calls to the function **inorder_next**, its also possible to said that $2(x-1)$ **is a possible value for the amortized complexity of this function**

In conclusion, by collecting data and using the method shown in this paper, its possible to test experimentally that $2(x-1)$ or $2x-2$ is a totally possible value for the amortized complexity of $n-1$ calls of the **inorder_next** function.

1.5 Appendices

1.5.1 Python code used for Task1

```
"""
-------------------------------------
HEADER OF THE PROGRAM
-------------------------------------
"""

# Import Section of the code
import random

# Constants definition section
LOWER_SIZE_BOUND = 50
UPPER_SIZE_BOUND = 100
RANDOM_TREES = 50
Id = 0
n = 0

"""
-------------------------------------
BINARY NODE CLASS DEFINITION
-------------------------------------
"""

# Class that creates a node of a binary tree
class binary_node()     :
    # Sets the initial configuration for a binary tree
        def __init__(self):
        # A new binary tree has no left or right Child
                global Id
                Id = Id + 1
                self.Id = Id
                self.leftChild = None
                self.rightChild = None

        # Returns the left Child node
        def get_lChild(self):
                return self.leftChild

        # Returns the right Child node
        def get_rChild(self):
                return self.rightChild

        # Creates a new left Child node
        def create_lChild(self, nodes, leftChildAbleNode, rightChildAbleNode
        , index):
```

```
                self.leftChild = binary_node()
                nodes.append(self.leftChild)
                del (leftChildAbleNode[index])
                leftChildAbleNode.append(self.leftChild)
                rightChildAbleNode.append(self.leftChild)
                return self.leftChild

        # Creates a new right Child node
        def create_rChild(self, nodes, leftChildAbleNode, rightChildAbleNode
        , index):
                self.rightChild = binary_node()
                nodes.append(self.rightChild)
                del (rightChildAbleNode[index])
                leftChildAbleNode.append(self.rightChild)
                rightChildAbleNode.append(self.rightChild)
                return self.rightChild


"""
-------------------------------------
BINARY TREE CLASS DEFINITION
-------------------------------------
"""

# Class that creates a random binary tree
class random_binary_tree():

        def __init__(self):
                self.nodes = self.create_randomTree()
                self.total = 0
                self.total_first = 0

        def create_randomTree(self):
                # An empty list with every node is created. Also one with the
                # left child capables is created and a third one with the right
                # child capables is also created
                nodes = []
                leftChildAbleNode= []
                rightChildAbleNode= []
                index = 0

                # Generates a random number between 50 and 100 for the size of tree
                treeNodesMax = random.randint(LOWER_SIZE_BOUND,UPPER_SIZE_BOUND)

                # First, the root of the tree is created
                root = binary_node()

                # The root is added to the nodes list
                nodes.append(root)
                leftChildAbleNode.append(root)
```

```
                rightChildAbleNode.append(root)

                while len(nodes) < treeNodesMax:

                        # If the flag is 0, then a new left child will be created
                         else a new right child will be created
                        leftRightFlag = random.randint(0,1)

                        # Choose a random Node from the correspondent list
                        if leftRightFlag == 0:
                                index = random.randint(0,len(leftChildAbleNode)-1)
                                leftChildAbleNode[index].create_lChild
                                (nodes, leftChildAbleNode, rightChildAbleNode
                                , index)
                        elif leftRightFlag == 1:
                                index = random.randint(0,len(rightChildAbleNode)-1)
                                rightChildAbleNode[index].create_rChild
                                (nodes, leftChildAbleNode, rightChildAbleNode
                                , index)

                return nodes


        # Return the list with nodes of the random Tree
        def get_nodes(self):
                return self.nodes

        # Transverse the binary tree using recursion
        def transverse(self, rootNode):
                if rootNode is not None:
                        if rootNode.get_lChild() is not None:
                                self.total = self.total + 1
                                self.transverse(rootNode.get_lChild())
                                self.total = self.total + 1
                        if rootNode.get_rChild() is not None:
                                self.total = self.total + 1
                                self.transverse(rootNode.get_rChild())
                                self.total = self.total + 1
                return None

"""
-------------------------------------
        MAIN PROGRAM
-------------------------------------
"""

# Creates a list of random trees
randomTrees = []

# Creates 50 Random Trees
```

```
while len(randomTrees) < RANDOM_TREES:
        tree = random_binary_tree()
        randomTrees.append(tree)


# Open file for writing the results in CSV format
f = open("Output.txt", "w")
f.write("RANDOM TREE ID;NUMBER OF NODES IN TREE;
        TOTAL COST OF INORDER_NEXT OPERATION\n")

# Prints the whole tree structure
for tree in randomTrees:
        n = n + 1
        tree.transverse(tree.get_nodes()[0])

        inorderNext_Cost = tree.total

        f.write("%d;%d;%d\n" % (n,len(tree.get_nodes()),inorderNext_Cost))

        print("\n               TREE  N  %d" % n)
        print("=================================================")
        print("NUMBER OF NODES IN TREE: %d  NODES" % len(tree.get_nodes()))
        print("TOTAL COST OF INORDER_NEXT: %d  STEPS" % inorderNext_Cost)
        print("=================================================")

f.close()
```

1.5.2 Github URL for python code

https://github.com/JokerswildX/T1HSK

Chapter 2

Preassignment - Task 2

2.1 Introduction

The second task of the preassignment consists of making an estimation of the amortized analisis of $m1$ calls to push(x) and $m2$ calls to pop(k) in a stack data structure. It is important to remember that applying the push(x) function to a stack, pushes an item x onto the top of that particular stack. In contrast, applying the pop(k) function to a stack allows us to pop out k entries of the stack at once.

The strategy followed to solve this task is described in detail in the following section.

2.2 Materials and Methods

To solve this task, the known paradigm of "divide and conquer" is used.

Its known that the amortized complexity of a whole operation is the sum of the amortized complexities of its stages. Therefore, instead of estimating the total amortized complexity of $m1$ calls to push(x) and $m2$ calls to pop(k) 2 different amortized complexities will be estimated:

- Amortized Complexity of call push(x)
- Amortized Complexity of call pop(k)

For estimating the above complexities the following equation needs to be considered:

$$a_i = t_i + \Phi(S_i) - \Phi(S_{i-1})$$

Where a_i represents the amortized complexity of the ith call, t_i represents the actual worst case complexity of the ith call, $\Phi(S_i)$ represents the potencial function of the ith call and $\Phi(S_{i-1})$ the potencial function of the $(i-1)$th call

In addition to the above equation, a right potencial function ($\Phi(S)$) needs to be chosen so that $\Phi(S_i) \geq \Phi(S_{i-1})$. This behaviour allows the amortized complexity to work as an upper bound of the total actual complexity of the operation. In the next section, all the above information will be used to estimate the amortized complexity of the sequence of push(x) and pop(k) in the stack.

2.3 Results

First of all, as seen in the last section, before even begun estimating the amortized complexity of the sequence of pop and push calls, a right potencial function needs to be chosen. **In this particular case, the potencial function $\Phi(S_i)$ was chosen to be the current number of items in the stack**. This chosen potencial function seems to be adequate as the bigger the number of elements in the stack, the more steps, potencially the operation must do in order to reach the next state. Also, considering that the initial state of the stack is empty, after n operations its known that $\Phi(S_n) \geq \Phi(S_0)$ as no negative index are allowed into a stack.

Having decided the potencial function $\Phi(S)$, the next task is to estimate the amortized complexity for the both considered cases, the $m1$ calls to push(x) and the $m2$ calls to pop(k). Let's begin with the $m1$ calls to push(x):

1. $m1$ calls to push(x):
   Let's start by estimating the push(x) amortized complexity. If the ith call in the stack is a push(x) and the stack has s elements in it, the following estimation could be made:

$$a_i = t_i + \Phi(S_i) - \Phi(S_{i-1})$$
$$a_i = 1 + (s+1) - (s)$$
$$a_i = 2$$

So its possible to conclude that $m1$ calls to push(x) has an amortized complexity of $2 \cdot (m1)$. Let's now examine the case of the $m2$ calls of pop(k).

2. $m2$ calls to pop(k):

For estimating the pop(k) amortized complexity, its possible to consider that the ith call in the stack is a pop(k) operation and that the stack has a initial state with s elements in it. In this case, the following estimation could be made:

$$a_i = t_i + \Phi(S_i) - \Phi(S_{i-1})$$
$$a_i = min(k, s) + (s - min(k, s)) - s$$
$$a_i = 0$$

The value $min(k, s)$ represents the minimum value between the number k and s. This value needs to be considered because the minimum value between k *(number of elements that are popped out of the stack)* and s *(the total amount of elements in the stack)* will be the maximum possible value to pop out of the stack for the case described above. Considering the above result, its possible to conclude that $m2$ calls to pop(k) has an amortized complexity of 0.

Considering the results of the $m1$ calls to push(x) and the $m2$ calls to pop(k) and also considering that the total amortized complexity is the sum of its stages, its possible to conclude that the total amortized complexity of $m1$ calls to push(x) and the $m2$ calls to pop(k) is $2 \cdot (m1)$. The maximum value that $2 \cdot (m1)$ can take is n where n is the maximal size of the stack. **Therefore its possible to conclude that the amortized analysis of the sequence is O(n)**.

2.4 Discusion

When comparing the result of the amortized complexity analysis obtained in section 3 (complexity O(n)) with the result of the worst case complexity analysis (complexity O(n^2)), its possible to infer the following conclusions:

- The worst case complexity analysis is much more pessimistic than the amortized complexity analysis
- Certainly, the amortized analysis is more accurate than the worst case complexity analysis, however, define an adequate potential function can be really challenging in some specific cases (requires some previous experience)
- Considering the fact that choosing a potential function can be sometimes challenging, estimating an amortized complexity can take more time than estimating the worst case complexity

In conclusion, the amortized complexity analysis is a very helpful and also more precise method for estimating an algorithm complexity than the worst case complexity method. However, in some cases could be a little more challenging to estimate, therefore I strongly recommend to consider the final goal of the complexity estimation before deciding which method its better to use in a particular case.