Empirical Algorithm Analysis
Efficiency of the in-order traversal algorithm

CAB301 - Assessment 2

Braydan Newman - n11272031

Contents

0.1 Introduction

In this empirical investigation, the primary aim is to assess the operational effectiveness of the in-order traversal method applied to a binary search tree (BST). The goal lies in determining the traversal process's efficiency class through counting the basic operations performed. The ratio of this count to the number of nodes in the Binary Tree determines the efficiency class.

0.2 Purpose of the Empirical Study

The objective of this empirical analysis of algorithms is to formulate a hypothesis regarding the time complexity class of the in-order traversal algorithm presented in Appendix A.

0.3 Efficiency Metric to be Measured

The efficiency metric used in this empirical algorithm analysis is the number of times the algorithm's basic operations are performed. These basic operations include:

- Adding of the current node's tool to the ordered list of tools
- Visiting a node in the Binary Tree

0.4 Implementation for Experimentation

The algorithm was implemented in C# and a counter was inserted into the algorithm implementation and this was used to count the total number of times the basic operations occurred the In-order traversal implementation.

0.5 Generation of a Sample of Inputs

A test program was written in C# that generates random test data for the binary tree, the generated data is 20 sets at 20 different intervals, with the length of the set ranging from 1000 to 20,000 at intervals of 1000. In total 400 different sets where tested each set ranging in length from 1000 to 20,000. The C# program can be found in Appendix C.

0.6 Running the Algorithm Implementation on the Sample Inputs and Recording Results

For each of the 20 intervals the 20 tests where averaged to get 20 results, one for each interval of length of the dataset. Each test the counter was used to count the number of basic operations and then after each test the count was recorded, the collated results of this are shown in the table below. The raw experimental results have been submitted through Gradescope.

0.6.1 Experimental Results

Test Data Table of Results

Number of Tools	Number of Operations (Avg)
1000	2000
2000	4000
3000	6000
4000	8000
5000	10000
6000	12000
7000	14000
8000	16000
9000	18000
10000	20000
11000	22000
12000	24000
13000	26000
14000	28000
15000	30000
16000	32000
17000	34000
18000	36000
19000	38000
20000	40000

Table 1: Basic opperation count results of 20 sets of 20 interval ranges of random data

0.7 Analysis of the Experimental Results

As shown by the results of the test data, the number of operations is double the number of nodes/tools in the collection. This is to be expected, as in this traversal of the binary tree every node is visited at least once and, in this implementation, it is only visited once. As well as traversal, each nodes "tool" is added to an array, this is also a basic operation and must happen once on every node, making the basic operations per node be a ratio of 2:1, operation count to node count.

- **Traversal Operation:** Every node in the BST must be visited at least once during the in-order traversal.
- **Addition Operation:** Every tool in each node in the BST is added to the array.

This can be calculated as the following, let the efficiency function $t(n)$ represent the total number of basic operations for n nodes in the binary tree. $t(n) = cg(n)$ where c is a constant and $g(n)$ is a simple function. Calculating the values of $t(2n)$ and $t(n)$ then calculating the ratio with $\frac{t(2n)}{t(n)}$ using the results in table.

$$\begin{aligned} n &= 1000 \\ t(2n) = t(2000) &= 4000 \\ t(n) = t(1000) &= 2000 \\ \frac{t(2n)}{t(n)} = \frac{4000}{2000} &= 2 \end{aligned} \tag{1}$$

This gives a ratio of 2:1, meaning the efficiency function is linear as doubling the number of nodes doubles the number of basic operations performed in this in-order traversal.

0.7.1 Empirical Observation

The behaviour above shows a linear correlation with the Complexity being $O(2n)$ which is then simplified to the complexity class $O(n)$, the constant 2 is omitted from Big-O notation as it doesnt add much complexity when input sizes are very large.

Appendix A

A.1 Pseudocode

A.1.1 ToArray

Algorithm 1 ToArray()

1: $tools \leftarrow []$
2: $index \leftarrow 0$
3: **if** $root \neq null$ **then**
4: $\quad inOrderTraversal(root, tools, index)$
5: **end if**
6: **return** $tools$

A.1.2 In-Order Traversal

Algorithm 2 inOrderTraversal(node, tools, index)

1: **if** $node.leftChild \neq null$ **then**
2: $\quad inOrderTraversal(node.leftChild, tools, index)$
3: **end if**
4: $index \leftarrow index + 1$
5: $tools[index] \leftarrow node.tool$
6: **if** $node.rightChild \neq null$ **then**
7: $\quad inOrderTraversal(node.rightChild, tools, index)$
8: **end if**

A.2 Code

A.2.1 ToArray

```
public ITool[] ToArray()
    {
        ITool[] tools = new ITool[count];
        int index = 0;
        if (root != null) InOrderTraversal(root, ref tools, ref index);
        return tools;
    }
```

A.2.2 In-Order Traversal

```
private void InOrderTraversal(BTreeNode node, ref ITool[] tools, ref int index)
    {
        if (node.lchild != null) InOrderTraversal(node.lchild, ref tools, ref index);
        tools[index++] = node.tool;
        if (node.rchild != null) InOrderTraversal(node.rchild, ref tools, ref index);
    }
  return go(f, seed, [])
}
```

A.2.3 Testing

```
static void Main(string[] args)
{
    for (int i = 1000; i <= TEST_SETS * 1000; i += 1000)
    {
        for (int j = 0; j < TEST_SETS; j++)
        {
            // Create Collection
            ToolCollection collection = new ToolCollection();
            // Create Random Set
            int[] randomSet = GenerateRandomArray(i);
            // Add all tools
            foreach (int item in randomSet)
            {
                collection.Insert(new Tool(item.ToString(), 100));
            }
            (ITool[] allTools, int basicCount) =
collection.ToArrayCount();
        }
}

static int[] GenerateRandomArray(int size)
{
    int[] A = new int[size];
    int seed = (int)DateTime.Now.Ticks;
    Random rnd = new Random(seed);
    for (int i = 0; i < A.Length; i++) A[i] = rnd.Next(Int32.MaxValue);
    return A;
}
```