EMPIRICAL ALGORITHM ANALYSIS

CAB301 Assessment 2

Matthew Chambers – n11318546

1. The purpose of this empirical study

The purpose of this empirical study is to assess the time efficiency of the proposed implementation of the ToArray algorithm seen in Appendix A and develop a hypothesis regarding the time complexity of the ToArray algorithm in the given context of the tool collection implementation.

2. The efficiency metric to be measured in this empirical study

In this study, we will first measure the execution time of the ToArray algorithm to convert the tool collection to an array and sort in alphabetical order. Then, use the number of operations done during the conversion process as the efficiency metric.

3. The basic operation in the ToArray algorithm

The ToArray algorithm utilises two helper functions to get the intended output of an alphabetically sorted array representation of the tool collection. The first of which is 'CountNodes' and works by using a recursive approach to count the nodes in the left and right subtrees and adds 1 for the current node. It then creates an array of matching size to the counted nodes.

The next helper function 'FillArray' is called to fill the array with the tools. Another recursive approach is used to traverse the binary tree via an in-order approach. In doing so, it ensures that the tools are sorted alphabetically as the binary tree is a binary search tree in which the nodes are arranged based on the alphabetical order of the tool names. The function then tracks the current index in the array where the next tool should be insert and inserts all tools from the left subtree and once done fills all tools from the right subtree.

'CountNodes' is first called in the ToArray algorithm followed by 'FillArray', then finishes by returning the final array containing the alphabetically sorted tools.

4. Implementation of the ToArray algorithm for experimentation

The ToArray algorithm was implemented within the ToolCollection class and a counter was implemented to count the number of times the main operation was performed. The implementation of the ToArray algorithm can be seen in Appendix B.

5. Generation of sample data

The function 'GenerateRandomTestData' was written to random generate a specified amount of tools to be inserted into the tool collection. The function takes in two parameters, a 'ToolCollection' object, and an integer 'numTools' which represents the number of tools to be generated. The function works by first creating a 'Random' object to generate random numbers. Then, an array of specified tool name is created to later ensure that the ToArray is properly alphabetically sorting the tools. A for loop then runs for the specified amount tools to generate or 'numTools'. Within the loop, a tool name is randomly selected using the Random.Next method on the array containing the specified tool names and a random number between 1 and 10 is generated to represent the number of tools available. Now that the generated tool has a name and number, it is then inserted into the tool collection utilising the 'Insert' function.

6. Running the algorithm implementation on the sample data

With the given random test data generating function, ten test of increasing number of tools size and its resulting operation count were recorded below.

Number of Tools	Operation Count
10	10
20	20
30	30
40	40
50	50
60	60
70	70
80	80
90	90
100	100

7. Analysis of the experimental results

With the given results the ToArray algorithm exhibits the characteristic of linear time complexity denoted as O(N) in Big O Notation. This is evident as the number of tools being inserted into the tool collection is always equal to the operation count. This indicates that the basic operation is occurring once for every tool. Therefore, with this linear scale between input size and operation count it is evident that the algorithm also displays a linear time complexity.

Appendix A

```
ALGORITHM ToArray()

// Returns an array of ITool objects

operationCount ← 0

count ← CountNodes(root)

tools ← array of ITool of size count

index ← 0

FillArray(root, tools, 0)

OUTPUT("Total operations performed: " + operationCount)

RETURN tools


ALGORITHM CountNodes(node)

// Recursively counts the number of nodes in the binary tree rooted at
'node'

IF node = null THEN

    RETURN 0

ELSE

    RETURN 1 + CountNodes(node.lchild) + CountNodes(node.rchild)


ALGORITHM FillArray(node, tools, index)

// Recursively fills the 'tools' array with ITool objects from the binary
tree rooted at 'node'

IF node ≠ null THEN

    operationCount ← operationCount + 1

    index ← FillArray(node.lchild, tools, index)

    tools[index] ← node.tool

    index ← index + 1

    index ← FillArray(node.rchild, tools, index)

RETURN index
```

Appendix B

```
private int operationCount = 0;

    public ITool[] ToArray()

    {

        operationCount = 0; // Reset the operation count

        int count = CountNodes(root);

        ITool[] tools = new ITool[count];

        FillArray(root, tools, 0);

        Console.WriteLine($"Total operations performed: {operationCount}"); // Log the
operation count

        return tools;

    }


    private int CountNodes(BTreeNode? node)

    {

        if (node == null)

            return 0;

        return 1 + CountNodes(node.lchild) + CountNodes(node.rchild);

    }


    private int FillArray(BTreeNode? node, ITool[] tools, int index)

    {

        if (node != null)

        {

            operationCount++; // Increment the operation count

            index = FillArray(node.lchild, tools, index);

            tools[index++] = node.tool;

            index = FillArray(node.rchild, tools, index);

        }

        return index;

    }
```

Appendix C

```
static void GenerateRandomTestData(ToolCollection toolCollection, int numTools)

    {

        Random random = new Random();

        string[] toolNames = { "Hammer", "Screwdriver", "Wrench", "Pliers", "Saw", "Drill",
"Chisel", "Level", "Tape Measure", "Clamp" };

        for (int i = 0; i < numTools; i++)

        {

            string toolName = toolNames[random.Next(0, toolNames.Length)]; // Randomly
select a tool name from the array

            int toolNumber = random.Next(1, 11); // Random number of tools available

            Tool tool = new Tool(toolName, toolNumber);

            toolCollection.Insert(tool);

        }

    }
```