Algorithm Analysis

Complex computational problems require efficient algorithms. This empirical study focuses on evaluating the time efficiency of a Binary Search Tree (BST) sorting algorithm. The objective is to develop a hypothesis about the time efficiency class of the sorting algorithm using the empirical algorithm techniques introduced.

1. The purpose of this empirical study

The purpose of this empirical algorithm analysis is to develop a hypothisis regarding the time efficiency class of the BST sorting algorithm in *Appendix A*.

2. The efficiency metric to be measured in this study

The efficiency metric used in the empirical algorithm analysis is the number of times the algorithm's basic operation is executed.

3. The basic operation in the sorting algorithm

Because the BST is already in alphabetical order the sorting algorithm simply needs to traverse the BST. This is completed by the basic operation of determining if the next node is null. The problem size is characterised by the paramiter n, which is the number of nodes in the Binary Search Tree.

4. Implementation of the BST Sort algorithm for experimentation.

The algorithm was implimented in C# anad a counter (testCount), which was used to count the number of times that the basic operation was performed, was inserted into the algorithm implementation. *Appendix B* is the algorithm implementation.

5. Generation of a sample of inputs

A C# program that can randomly generate a BST where n is the depth of the BST. This program was used to generaate 20 sets of problems (BSTs). The first set of test problems contained 20 randomly generated BSTs of depth 4; the second set of test problems contained 20 randomly generaated BSTs of depth 5; ...; and the eighteenth set of test problems contained 20 randomly generated BSTs of depth 22. The C# program can be found in *Appendix C*.

6. Running the algorithm imiplementation on the sample's inputs and recording the experimental results

For each set of the test problems, we ran the experimental program with each of the 20 test problems from each of the 20 sets of randomly generated test problems and recorded the counter’s value for each run. The raw experimental results have been submitted through Gradescope. Table 1 shows the average number of times the basic operation was performed for each set of the experiments.

Table 1. The staaticstics of the experiment results for 20 sets of test problems of different sizes ranging from a tree depth of 4 to 23 with an encremental of 1.

Depth	Average Runs	Average Problem Size

Depth	Average Runs	Average Problem Size
4	10	10
5	21	21
6	46	46
7	88	88
8	175	175
9	347	347
10	706	706
11	1413	1413
12	2834	2834
13	5652	5652
14	11319	11319
15	22613	22613
16	45222	45222
17	90471	90471
18	180912	180912
19	361727	361727
20	723325	723325
21	1446915	1446915
22	2893927	2893927
23	5788143	5788143

7. Analysis of the experimental results

Let (efficiency function) $t(n) = cg(n)$ where c is a constant and $g(n)$ is a simple function. Firstly, we calculate the ratio of $t(2n)$ and $t(n)$ using the above experimental results. For example $t(175)/t(88)=1$ Since the ratio of $t(2n)$ and $t(n)$ is less than two the algorithm is O(n). This makes sence because in the program the binary tree is already sorted into alphabetical order so each node only needs to be visited once.

Appendix A

```
ALGORITHM BSTSort(B)
    \\Outputs an array representation of a binary tree
    \\Input: A Binary Tree (B[n]) of pre-sorted tree
    \\Output: Array A[n] sorted in ascending alphabetical order
    WalkTree(x)
        IF x != NIL THEN
            WalkTree(x.left)
            visit node
            WalkTree(x.right)
```

Appendix B

```
    static int ToArray(tree)
    {
        ITool[] toolsInOrder = new ITool[number];
        int i = 0;
        int count = 0;
        void getToolsInOrder (BTreeNode currentNode)
        {
            count++;
            if (currentNode == null)
                return;
            if (currentNode.lchild != null)
                getToolsInOrder(currentNode.lchild);
            toolsInOrder[i] = currentNode.tool;
            i++;
            if (currentNode.rchild != null)
                getToolsInOrder(currentNode.rchild);
            return;

        }
        tree.getToolsInOrder(root);
        return count;
    }
```

Appendix C

```
namespace Assignment2
{
    public class TreeGenerator
    {
        private static ToolCollection? tree;

        static public void Main(String[] args)
        {
            tree = new ToolCollection();
            for (int depth = 4; depth <= 23; depth++)
            {
                Console.WriteLine("Depth: " + depth);
                for (int i = 1; i <= 20; i++)
                {
                    Console.WriteLine("Test#: " + i);
                    GenerateTree(depth);
                    tree.ToArray();
                    tree.Clear();
                }

Console.WriteLine("========================================");
            }
        }

        public static string ToBase26(int num)
        {
            string[] alpha = { "a", "b", "c", "d", "e", "f", "g", "h",
"i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w",
"x", "y", "z" };
            string[] name = new string[0];

            string[] addChar(string[] name, string newValue)
            {
                int len = name.Length + 1;
                string[] oldName = name;
                name = new string[len];
                for (int i = 0; i < oldName.Length; i++)
                {
                    name[i] = oldName[i];
                }
                name[len - 1] = newValue;
                return name;
            }

            int i = 0;
            while (num >= 26)
            {
                int value = num % 26;
                if (value == 0)
                {
                    num /= 25;
                    name = addChar(name, "a");
                } else
                {
                    num /= 26;
                    name = addChar(name, alpha[value]);
                }
            }
            if (num > 0)
            {
                name = addChar(name, alpha[num]);
            }
            return string.Join("",name);
        }

        public static void GenerateTree(int Depth)
        {
            Random rnd = new Random();
            int nodesThisRow = 1;
            tree = new ToolCollection();
            int i;
            for (i = Depth; i >= 1; i--)
            {
                int lmna = (int)Math.Pow(2, i);
                if (lmna >= 2147483591)
                {
                    return;
                }
                int name = lmna >> 1;
                for (int j = 0; j < nodesThisRow; j++)
                {
                    var node = new Tool(ToBase26(name));
                    int random = rnd.Next(0, 100);
                    if (random <= 30)
                    {
                        continue;
                    }
                    tree.Insert(node);
                    name += lmna;
                }
                nodesThisRow *= 2;
            }
            return;
        }
    }
}
```