Coursework 2: My Binary Tree

Module: CIS2147 Programming Languages: Theory to Practice 2021

Module Leader and Lecturer: Ardhendu Behera

By: Christopher Diaz Montoya
Student ID: 24707686

Contents

ss

Contents of figures

Introduction

Within this report the project carried out on traversing a given binary tree in multiple ways will be discussed, the tree is shown in Figure 1 - Assignment binary tree, this image was given in the project brief. The tasks at hand were fivefold and each has a dedicated section in this report, showcasing the skills and techniques learnt both in lectures and researched independently. This was completed using the C# programming language on the visual code studios IDE.

Jack (2020) along with Khot and Misra (2017) explained that binary trees start off with the root node at the top, a node being one blue circle in Figure 1 - Assignment binary tree. The tree then branches down until it reaches the final node called a leaf node. The numbers seen in the blue circles was the data stored in that node. The nodes with arrows below them are parent nodes as each arrow below points to its left or right child node, with any arrows above the node pointing back to that nodes parent, unless it is the root node which only has child nodes.

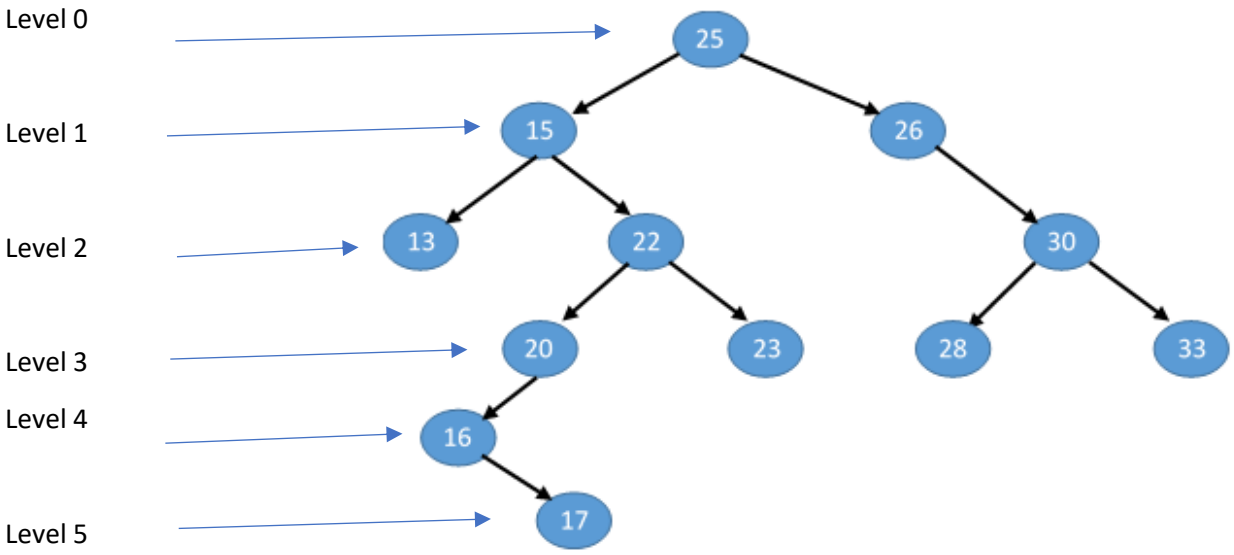

Figure 1 - Assignment binary tree

This project was tackled by breaking down each task as much as possible, researching each technique needed before attempting each task. Attempts were documented in a test table to show the problems encountered along with the solutions found and how they were discovered to complete each task.

Skeleton code

The checking and reading over the skeleton code will be discussed within this section.

Firstly, the skeleton code was read over, after viewing this, the following was inferred:

```
public void Inorder(MyNode tmpNode)
{
    if (tmpNode == null)
        return;
    else
    {
        //Add your code for In-order traversal
        //Look at the lecture slides for pseudocode

    }
}
```

Figure 2 - Inorder skeleton code

- There were three classes, one for the main method, one for the nodes and one for the tree.
- Some hints and help were given as to where some code should go for exercise one. This is shown in Figure 2 - Inorder skeleton code.
- The traversal methods/behaviors were written in the tree class.

After this, some research was done to understand binary trees. The lecture material revised confirmed the earlier made assumption from reading the skeleton code, was correct.

Next, the code was tested to see if it populated the tree. As shown in Figure 3 - InsertNode() skeleton code the InsertNode() method was provided in the skeleton code. This was checked in tests 1-3 which explain how the WriteLine statements were added along with why break statements were added later to the nested else statements within this figure.

The adding of the break statements was wrong and showed that the research done over binary trees was a rough overview instead of requiring an in-depth understanding, this will be explained in exercise 1.

```
while (true)
{
    //Add your code for building Binary tre
    if (id < current.item)  //if id is less
    {
        if (current.leftChild == null)  //i
        {
            current.leftChild = newNode; //
            break;
        }
        else //if not null
            current = current.leftChild; //
    }
    else //else the id is greater than or e
    {
        if (current.rightChild == null) //i
        {
            current.rightChild = newNode; /
            break;
        }
        else //if not null
        {
            current = current.rightChild; /
        }
    }
```

Figure 3 - InsertNode() skeleton code

Exercise 1

Overview

Exercise 1 asked for the tree to be traversed with three depth-first traversals which will be discussed within each subsection. These all had to be done recursively, this meant that the method had to call itself, within itself so it would loop until it did what was needed. Each subsection will explain, the problem breakdown/analysis, the research required, and the problems encountered when implementing and testing the traversal. Geeks for geeks (2021), Chang (2003) and the lecture material given all explained the three traversals and what was learnt and understood from all three will be amalgamated and explained in the exercise 1 subsections.

Implement an inorder traversal

The first step to implement an inorder traversal recursively was to understand what this was. The inorder traversal needs to go down the left child nodes, then when a leaf was found it needed to print the item and go back to the parent node, print it and then traverse and print the right node, or the left most child in that right node. As shown in Figure 2 - Inorder skeleton code, the skeleton code already came with a method for this and gave a hint that lecture slides had the pseudocode for it. As the research was in line with the lecture material and pseudocode the method was called recursively in the else statement with the left child node as a parameter, then the item in that node was printed and below, the method was called recursively with the right child node.

This did not work the first time as shown in test 4. This forced further research to be done to understand if the binary tree was populated correctly as the inorder method was in line with the

research and lecture slides. The issue found was the break statements added in the skeleton code section, this was not allowing the tree to populate itself properly as the break statement were not allowing the node to be assigned a place breaking the loop early. After the break statements were removed the output was as expected, shown in Figure 5 - Exercise 1 outputs.

```
public void Preorder(MyNode tmpNode)
{
    // Conditional statements.
    // If node returned is non existant (Node
    if (tmpNode == null)
        return; // Returns to parent node.
    // else execute code in parenthsis in els
    else
    {
        // Prints node item, tmpNode = node o
        // Each node stores something differe
        // .Write used instead of .Writeline
        Console.Write(tmpNode.item + "  ");
        // Recursive call, Preorder() calls t
        Preorder(tmpNode.leftChild);
        // If the node does not have a left c
        Preorder(tmpNode.rightChild);
    }
```

Figure 4 - Preorder method/behaviour

Implement a preorder traversal

The first step to implement a preorder traversal recursively was to understand what this was. As the research was in line with the lecture slides the expected outcome was a preorder traversal of the whole tree. This traversal needed the root node to be printed out first and then the whole left side printed one by one and then the right side.

As shown in Figure 4 - Preorder method/behaviour this was implemented by printing each item within the node so the root would be first then recursively calling the method down the left child nodes and printing then down the right nodes. The output was as expected shown in Figure 5 - Exercise 1 outputs.

```
Inorder recursive traversal:
13  15  16  17  20  22  23  25  26  28  30  33
Preorder recursive traversal:
25  15  13  22  20  16  17  23  26  30  28  33
Postorder recursive traversal:
13  17  16  20  23  22  15  28  33  30  26  25
```

Figure 5 - Exercise 1 outputs

Implement a postorder traversal

The first step to implement a postorder traversal recursively was to understand what this was. As the research was in line with the lecture slides the expected outcome was a postorder traversal of the whole tree. This meant to print the left most leaf first, then the right most then their parent node and work their way back up.

This was implemented with help of the psudo code provided, using recursion all the way down the left side calling the method with the left child node, then right side and finally printing each item within the node. The output was as expected shown in Figure 5 - Exercise 1 outputs.

Exercise 2

This exercise was to use a depth-first search (DFS), using a stack, to find the node in the binary tree containing item 20 and printing each item within each visited node, and once the element was identified to then print the path back to the root node. The task was broken down in steps on paper as follows:

1. Create a new behaviour/ method in the MyBinaryTree class. Must print items and not return anything to main so keyword void was used. Method called findItem20PreOrder
2. Learn how stacks work and are implemented.
3. Figure out how to make the root node print first then down the left child nodes and right child nodes after with a stack. Stacks has a Last-in first out (LIFO) (Chastine, 2013) structure so the first item is the root node which was added to stack, printed, and stored in a list, then right node and then left node so left could be printed first.
4. Create a base case.
5. Create a while loop to traverse the nodes and break when a condition was met.
6. Exit the loop with a conditional statement checking each loop and use the break statement when item 20 was found, and print the items found.
7. Create a way to store the items traversed to reverse and path backwards at the end, must be able to change size as the program will allow the user to enter a custom tree. Use lists. Use the .reverse() method learnt last academic year to reverse the list.

```
while (nodeDataStack.Count != 0)
{
    // .peek() returns the top node within
    MyNode topNode = nodeDataStack.Peek();
    // prints item (field) in the top node
    Console.Write(topNode.item + "  ");
    // Adds item to the list to print back
    backPath.Add(topNode.item);
    // .pop() removes node at the top of st
    nodeDataStack.Pop();
```

Figure 6 - Exercise 2 code snippet

Some of the above steps can be seen implemented in Figure 6 - Exercise 2 code snippet This shows the while loop with the top element in the sack being printed stored and popped.

The expected output was that the items in each node, were printed in a preorder traversal, the algorithm to stop and state when item 20 in a node was found, and then to print out the path back of the preorder traversal. As shown in Figure 7 - Exercise 2 output, this was accomplished.

```
Find item 20 by printing each visted node using a preorder traversal with a stack:
25  15  13  22  20  ->  Item 20 found!
Path back to root is:
20  22  13  15  25
```

Figure 7 - Exercise 2 output

Chastine (2013) and Microsoft (n/a) explained how to use a stack adding and removing items and explained to create a stack use:

```
Stack <dataType> variableName = new Stack <dataType> ();
```

The loop would store, print and pop the top item with backPath.Add(classNode.fieldItem), Console.WriteLine(classNode.fieldItem) and stackName.Pop() respectively, with stack.Pop() deleting the top item, each time. Also in the loop, three separate conditional statements were created; to exit the loop if item 20 was found with a string saying it was found; to use stack.Push(node.item) to add the right child to the stack first; and for the left child, the same as for the right child. Once the item at the top of the stack met the conditional statement for equalling integer twenty, (this was done with a double equal operator in the program) the loop was exited.

When implementing this it was not successfully the first time around, as shown in test 8, there was a looping issue as the top item left in the stack was item 13. Item 13 was shown in Figure 1 - Assignment binary tree as a leaf node as it has no child nodes. This needed to be popped so the item below, the right child of item 13s parent node could be printed. This would print item 22 which was below item 13 in the stack.

Exercise 3

Exercise 3 was to use a breadth-first search (BFS), using a queue to find the node in the binary tree containing item 30, printing each item within each visited node, and once the element was identified to print the path back to the root node.

A breadth first search was learnt in lectures and is to traverse the binary tree horizontally in levels, the levels are shown in Figure 1 - Assignment binary tree the first traversal would have started at level 0 and go left to right within each level before continuing onto the next (Dot net for all, 2018).

The task was broken down in steps, as it was like exercise 2, the code and layout were similar and went as follows:

1. Create a new behaviour/ method in the MyBinaryTree class. Must print items and not return anything to main so keyword void was used. Method called findItem30BFS
2. Learn how queues work and are implemented.
3. Figure out how to make the tree print out the nodes in a level order so all the nodes on level 0 are printed out first then level 1 then level 2 and so forth. The queue had a first in first out (FIFO) structure, so the first item is the root node which added, printed, and stored, then the left node, then the right node so the left could be printed first.
4. Create a base case.
5. Create a while loop to traverse the nodes and break when a condition was met.
6. Exit the loop with a conditional statement checking each loop and use the break statement when item 30 was found, and print the items found.
7. Create a way to store the items traversed to reverse and path backwards at the end, must be able to change size as the program will allow the user to enter a custom tree. Use lists.

```
while (queue.Count != 0)
{
    // .peek returns the first node in qu
    MyNode topNode = queue.Peek();
    // prints item in the first node.
    Console.Write(topNode.item + "  ");
    // Adds topNodes data to the list to
    backPath.Add(topNode.item);
    // .Dequeue() removes node at the fro
    queue.Dequeue();
```

Figure 8 - Exercise 3 code snippet

Some of the steps can be seen implemented in Figure 8 - Exercise 3 code snippet, if compared to Figure 6 - Exercise 2 code snippet they can be seen to be almost the same, as the code was almost identical for both methods. With the exception of instead of using, .Pop() and .Push(). .Enqueue() and .Dequeue() were used as the lecture material explained this was to be used for queues. The pseudocode in the lecture slides was also used to identify the need to add the left child first then the right.

Chastine (2013) and Microsoft (n/a) explained how to us a queue and to initialise a queue use:

```
Queue <dataType> variableName = new Queue <dataType> ();
```

The expected output was that the items in each traversed node, in a breadth-first order were to be printed out, the algorithm to stop and state when item 30 in a node was found, and then to print out the path back of the traversal. As shown in Figure 9 - Exercise 3 output This was accomplished. As it was like the previous task this had no issues as shown in test number 10.

```
Find item 30 by printing each visted node using a breadth first search with a queue:
25  15  26  13  22  30  ->  Item 30 found!
Path back to root is:
30  22  13  26  15  25
```

Figure 9 - Exercise 3 output

Exercise 4

Exercise 4 was to use recursion with an unspecified search algorithm to search for item 20 and 30. This was done printing out a message when item 20 and 30 was found and the method created for this to end when both were found. An inorder traversal was used. The steps were as follows:

- Create a new behaviour/ method in the MyBinaryTree class. Must print items and not return anything to main so keyword void was used. The method was called Inorder20and30.
- Create a conditional statements:
    - One for the base case.
    - One if item 20 was the currently traversed node and to print item 20 found, has recursive calls for left child nodes, printing the item found and continuing the recursive method call on the right child.
    - One if item 30 is in the current node and to print item 30 found, break recursion by not including recursive call after the message is printed.
    - One to just continue the recursive calls and print the items in the correct order as exercise 1 inorder traversal did.

The expected output was the items in the nodes to be printed out in an inorder traversal when each node was visited, with a message printed when item 20 was found along with another printed for when item 30 was found, and for the traversal to end there. This worked as expected shown in Figure 10 - Exercise 4 output and test number 11. No additional research was needed as this was covered in exercise 1 and the conditional statements were taught last academic year.

```
Find item 20 and 30 using inorder algorithm with recursion:
13  15  16  17  20  ->  Item 20 found! Traversal will continue below.
22  23  25  26  28  30  ->  Item 30 found! End of recursive call.
```

Figure 10 - Exercise 4 output

Extra work

Custom tree

As there was extra time, two extra features were added to the project. The first was to allow the user the choice to use the given binary tree for the project or to make a custom one but following the rule that lower numbers are left child nodes and vice versa for the right child nodes. This implementation was explained in test 6 and 12. No additional help was needed for this as all the required techniques had been learnt through lectures and seminars.

Display binary tree

The second was to display the binary tree the user had chosen. Like Figure 1 - Assignment binary tree. This feature proved difficult and although an attempt was made it was not completed. Help from online was required and most of the code used was from Pena (2016), it was understood (showed in the code comments) and edited to make the binary tree appear as shown in Figure 11 - Binary tree displayed.

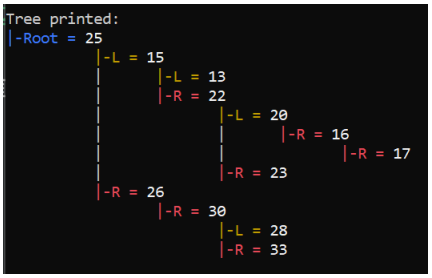

Figure 11 - Binary tree displayed

Reflection

The outcome of all the tasks was as expected, with additional work. Although there was room for improvement. The following could have been done to improve the project:

- The complexity should have been checked.
- Other new techniques could have been explored such as switch statements.
- When users entered a decimal number or letter, when a whole number was required, the program could have identified what they entered and told the user to not enter a decimal or letter and to try again with a whole number.
- When a user is inputting integers for the nodes, if they entered anything else the program takes them back to choosing a custom tree or the assignment tree. The program could have been designed (or attempted) in a way to just ask for the item again.

The project was a success as it was completed on time, had clear comments, new techniques were explored, and the project was improved.

Bibliography

DOTNETFORALL., 2018. *Breadth First Search (BFS) in C#* [online]. Available from: https://www.dotnetforall.com/breadth-first-searchbfs-in-c/ [Accessed 24 December 2021].

CHANG, K, S., 2003. *Data structures and Algorithms* [ebook]. Available from: https://www.google.co.uk/books/edition/Data_Structures_and_Algorithms/1lCHYj5eV-EC?hl=en&gbpv=0 [Accessed 10 December].

CHASTINE, J., 2013. *Tutorial 20.2 – Stacks and Queues in C#* [online video]. Available from: https://www.youtube.com/watch?v=tW75yz3X_M4. [Accessed 20 December 2021]

GEEKSFORGEEKS., 2021. *Tree Traversals (Inorder, Preorder and Postorder)* [online]. Available from: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ [Accessed 9 December 2021].

JACK, A., 2020. *Binary Trees – Data Structures Explained* [online video]. Available from: https://www.youtube.com/watch?v=GzJoqJO1zdI [Accessed 1 December 2021].

KHOT, A. and MISHRA, K, R., 2017. *Learning Functional Data Structures and Algorithms* [eBook]. Birmingham: Packt Publishing. Available from: https://www.oreilly.com/library/view/learning-functional-data/9781785888731/ch04.html [Accessed 1 December 2021].

MICROSOFT., n/a. *Queue<T> Class* [online]. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-6.0 [Accessed 25 December].

MICROSOFT., n/a. *Stack Class* [online]. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack?view=net-6.0 [Accessed 21 December].

PENA, X., 2016. C# Display a Binary Search Tree in Console. *Stack overflow.* [Blog online]. Available from: https://stackoverflow.com/questions/36311991/c-sharp-display-a-binary-search-tree-in-console/36496436#36496436 [Accessed 31 December 2021].