Binary Search Tree Creation and Traversal Visualization from given Node points using Drawing Algorithms in Java

A TERM PROJECT REPORT

Submitted in partial fulfilment for the 2nd year, 4th semester of

Bachelor of Technology in

Data Structures CSD-223

By

Avinal – 185067, Harsimranjeet – 185087, Pooja – 185101

Under the Guidance of

Er. Nitin Gupta

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

Hamirpur, Himachal Pradesh

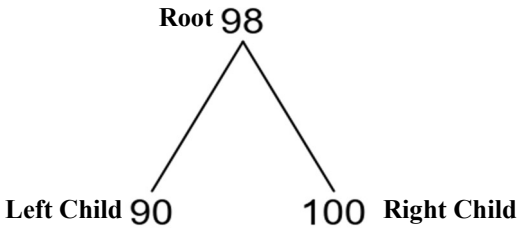
India - 177005

The Problem

Tree is a very important data structure and we can see its use in a variety of program directly or indirectly. But there are lots of hurdles when it comes to really visualizing it. Most of the time only option we have is that is to grab a pen and paper and draw it node by node. Other times we may use web-based tools that just draws the tree hiding all the implementation and they are mostly drag and drop drawing applications rather than actually drawing a tree from data points. As a computer science student these things doesn't help much to the practical understanding of data structures such as trees.

Tree Traversals

Tree offers mainly two types of traversals. Depth-First-Search and Breadth-First-Search. For the sake of this project we will constrict our discussion up to Binary Tree and specifically Binary Search Trees. Furthermore, a Binary Tree is a tree in which any node can have at most 2 children. They are denoted as left and right child. A typical example of the Binary tree is given below: -



Depth-First-Search is further divided into three types: -

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

All three of the above traversals implement similar recursive algorithm only differing in the order of calling of the root and the child node. Given below is the recursive algorithms of all three traversals:

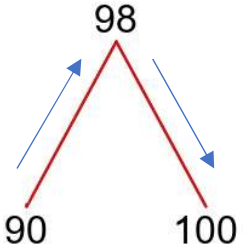
```
1:  function INORDERTRAVERSAL(node)
2:      if node = null then
3:          return
4:      end if
5:
6:      if leftchild[node] ≠ null then
7:          INORDERTRAVERSAL(leftchild[node])
8:      end if
9:
10:     PRINT(data[node])
11:
12:     if rightchild[node] ≠ null then
13:         INORDERTRAVERSAL(rightchild[node])
14:     end if
15: end function
```

Inorder Traversal follows left child, root and then right child strategy.

Left Recursion (L)

Root (T)

Right Recursion (R)



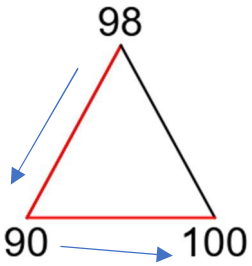
```
1: function PreorderTraversal(node)
2:     if node = null then
3:         return
4:     end if
5:
6:     Print(data[node])
7:
8:     if leftchild[node] ≠ null then
9:         PreorderTraversal(leftchild[node])
10:    end if
11:
12:    if rightchild[node] ≠ null then
13:        PreorderTraversal(rightchild[node])
14:    end if
15: end function
```

Preorder Traversal follows root, left child and then right child strategy.

Root (T)

Left Recursion (L)

Right Recursion (R)



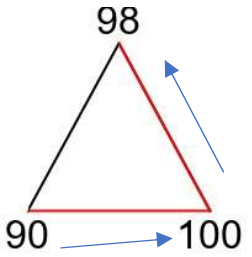

```
1: function PostorderTraversal(node)
2:     if node = null then
3:         return
4:     end if
5:
6:     if leftchild[node] ≠ null then
7:         PostorderTraversal(leftchild[node])
8:     end if
9:
10:    if rightchild[node] ≠ null then
11:        PostorderTraversal(rightchild[node])
12:    end if
13:
14:    Print(data[node])
15: end function
```

Postorder Traversal follows left child, right child and lastly root strategy.

Left Recursion (L)

Right Recursion (R)

Root (T)

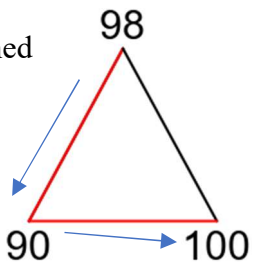


Breadth-First Traversal or Level Order Traversal has many accepted algorithms for now we will see what is called a queue of nodes method.

```
1: function LevelorderTraversal(node)
2:     if node = null then
3:         return
4:     end if
5:
6:     queue ← NULL
7:     Enqueue(queue, node)
8:
9:     while queue ≠ empty do
10:        temp ← Dequeue(queue)
11:        Print(data[temp])
12:
13:        if leftchild[temp] ≠ null then
14:            Enqueue(queue, leftchild[temp])
15:        end if
16:
17:        if rightchild[temp] ≠ null then
18:            Enqueue(queue, rightchild[temp])
19:        end if
20:    end while
21: end function
```

Level Order Traversal follows root then all the child of the root and then proceeds to another level of nodes.

Left and Right child pushed into queue.



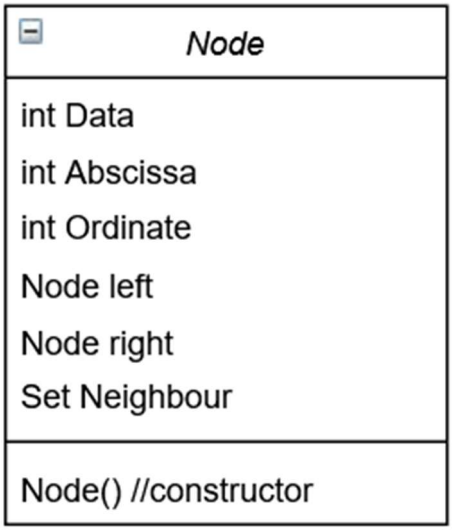
Our Approach

The main problem was to design a algorithm that can draw these nodes as they are traversed. For this we designed several procedures. As we are taking a series of numbers first we needed to create a Binary Search Tree out of it for futher operations. We divided the problem into three parts: -

1. Creation of Binary Search Tree from series of numbers.
2. Allocation of relative coordinates to the Nodes.
3. The Drawing Algorithms.

1. Creation of Binary Search Tree from series of numbers.

We first designed a **Node** class to store data of a single node , given below is the class diagram of the Node class and its resulting code.




```
class Node {
    String data;
    int xpos;
    int ypos;
    Node left;
    Node right;
    Set<Node> neighbour = new HashSet<>();
    Node(String data, Node left, Node right) {
        this.left = left;
        this.right = right;
        this.data = data;
    }
}
```

- **Data –** data of the node
- **Abscissa –** x coordinate of the node on the canvas
- **Ordinate –** y coordinate of the node on the canvas
- **Left, Right -** two children of the node
- **Neighbour -** immidiate neighbours of the node at least one, at most three

Now we needed an algorithm to form the tree. For this we used a recursive insert algorithm as given below.

```
1:  function INSERT(node, data)
2:      if node = null then
3:          node ← NODE(data, null, null)
4:      else
5:          if data < data[node] then
6:              leftchild[node] ← INSERT(leftchild[node], data)
7:              ADD(neighbour[node], leftchild[node])
8:              ADD(neighbour[leftchild[node]], node)
9:          else if data > data[node] then
10:             rightchild[node] ← INSERT(rightchild[node], data)
11:             ADD(neighbour[node], rightchild[node])
12:             ADD(neighbour[rightchild[node]], node)
13:         end if
14:     end if
15: end function
```

This algorithm works by recursing to the left or right after comparing a new node to the root node. At the same time we are also updating the **neighbour** set by adding the new node to the set. This will help us draw the edges between the nodes later.
As of now we have successfully formed our Binary Search Tree of given series of numbers.

2. Allocation of relative coordinates to the nodes.

To draw a node we must know its coordinates on the canvas. Manually providing coordinates fortifies the purpose of algorithm and programming. Let us see how we can do this.
First we did an inorder traversal of the previously formed tree according to following algorithm.

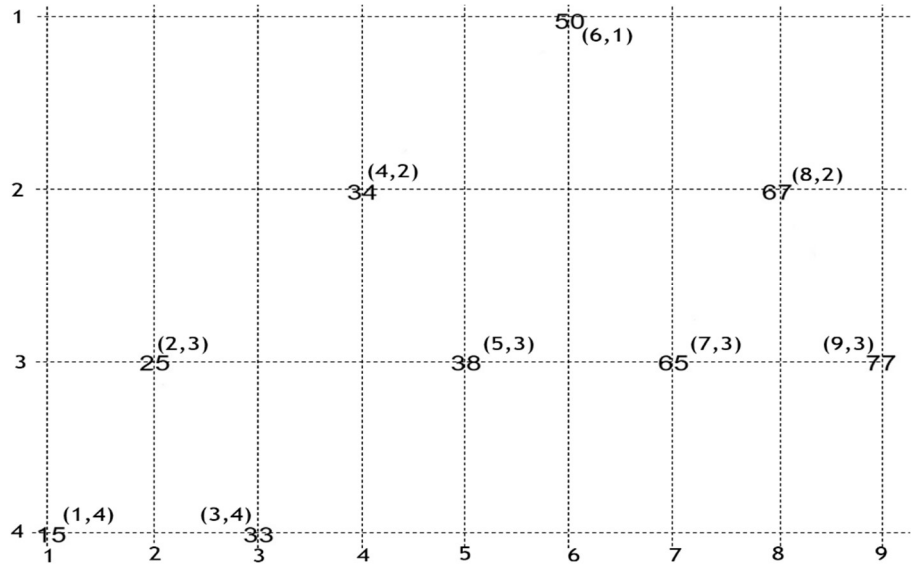
```
1: function COORDINATE(node, depth)
2:     if node ≠ null then
3:         COORDINATE(leftchild[node], depth + 1)
4:         abscissa[node] ← breadth
5:         ordinate[node] ← depth
6:         COORDINATE(rightchild[node, depth + 1)
7:     else
8:         breadth ← breadth + 1
9:     end if
10: end function
```

The algorithm is very simple. Whenever a node is encountered we increse the breadth (initially 0) and store as abscissa and whenever a recursion is called we increase the depth (initially 1). Let us understand how does this help to form a node diagram with the help of an example.

Let us assume a series of numbers 50 34 67 25 38 65 77 15 33. After tree forming and running **Coordinate** algorithm we get the following table:-

Node	15	25	33	34	38	50	65	67	77
X	1	2	3	4	5	6	7	8	9
Y	4	3	4	2	3	1	3	2	3

Nodes are now arranged in increasing order because the algorithm is based on inorder traversal. There are total 9 nodes and 4 levels. Let us plot a graph.



We can easily see that all the nodes are placed on grid with 1 unit distance in both coordinates. This way we have now placed as far as possible from each other. So the resultant tree will be nicely spaced.
Now we have provided relative coordinates to each node we can now draw edges between them and plot them simultaneously.

3. The Drawing Algorithms

Now that our tree is formed and nodes coordinates are assigned we just need to create the edges beween the nodes. This should be simple because to draw a line we just need 2 points and the nodes already have their coordinates. We will now implement an algorithm to draw an inorder traversal of the above tree. All other algorithms are similar.

So the pseodocode shall be,

- Traverse the tree by inorder traversal.
- Plot the Node and mark the Node as visited.
- Check if any of its neighbours have also been visited.
- If yes then draw a line between current Node and the neighbour Node.
- If no then continue.

Final algorithm is as below:-

```
1: function DrawInorder(node)
2:     if node ≠ null then
3:         x1 ← abscissa[node]                          Get First Coordinate
4:         y1 ← ordinate[node]
5:
6:         DrawInorder(leftchild[node])                 Call Drawing routine to draw left
7:                                                      child tree.
8:         Plot(data[node], x1, y1)                     Plot the current Node
9:         visited[node] ← true
10:        for i in neighbour[node] do                  Draw Lines if neighbours have
11:            if visited[i] = true then                been visited.
12:                x2 ← abscissa[i]
13:                y2 ← ordinate[i]
14:                DrawLine(x1, y1, x2, y2)
15:            end if
16:        end for                                      Call Drawing routine to draw
17:        DrawInorder(rightchild[node])                right child tree.
18:    end if
19: end function
```

The algorithm is very similar to the actual Inorder Traversal algorithm with addition of few more lines. But problem is not solved yet. We have only given a relative coordinate to the Nodes. But the actual draw area may not follow the same coordintes. So we need to scale our coordinates to drawing area coordinates. This task can be accomplished as follows.

$$new\ abscissa = \frac{SCREENWIDTH}{total\ number\ of\ nodes} * (old\ abscissa)$$

$$new\ ordinate = \frac{SCREENHEIGHT}{depth\ of\ the\ tree} * (old\ ordinate)$$

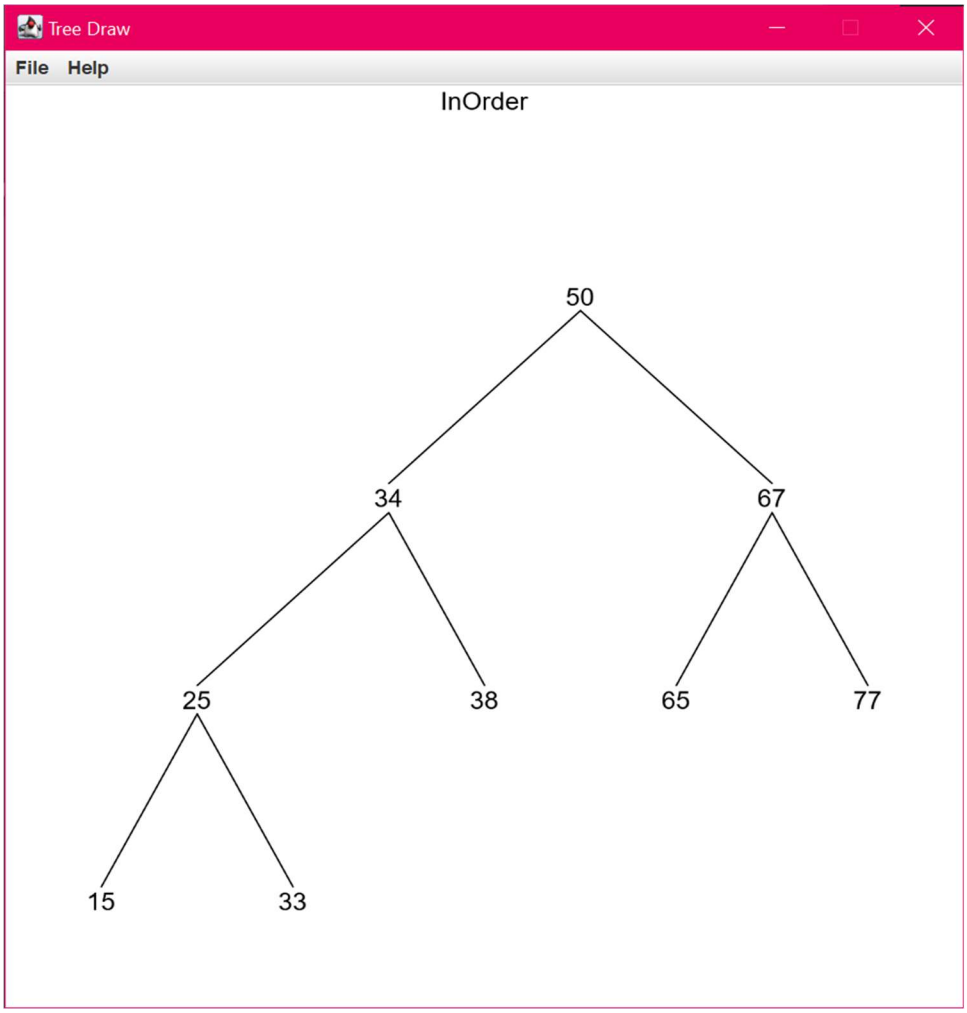
Similar algorithm follows for all other travesals. Now everything is complete and we can implement our algorithm in any programming language. We have used Java for this purpose because Java program have lesser dependencies and creating a GUI application is simpler than other languages e.g. C++ and C.

Programming Challanges

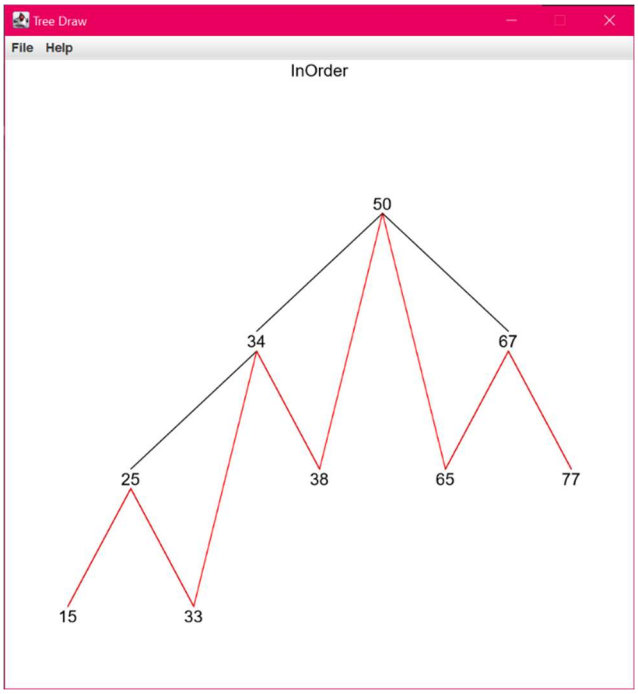
Although Java provides a lot of methods and facilities by default. We faced many challenges during this project. We have made this project to be modular and extensible. In future anyone can add a new feature easily to the project without disturbing the other routines.

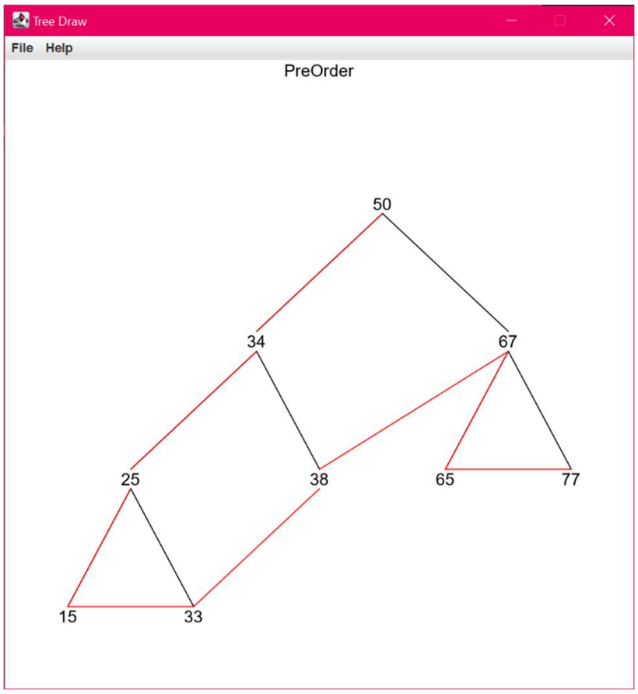
The project is hosted on GitHub as a Open Source Project with API reference and implementation documentation. https://blitzar-to-supernova.github.io/Binary-Search-Tree-Traversal/

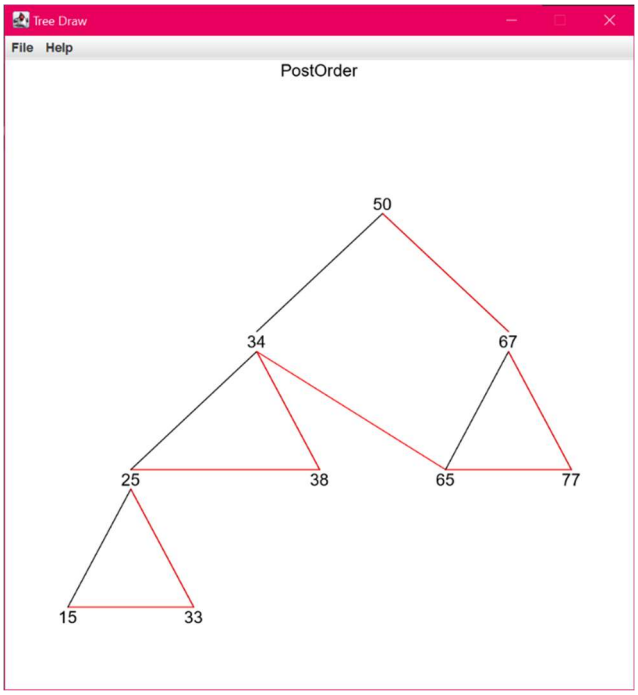
We initially proposed to create an application that can animate the traversals of a Binary Search Tree. But during development we added many other features. Programming the actual project was relatively easy since we already got all the things we need and the algorithm. Here is the tree formed from the example taken in the last section.

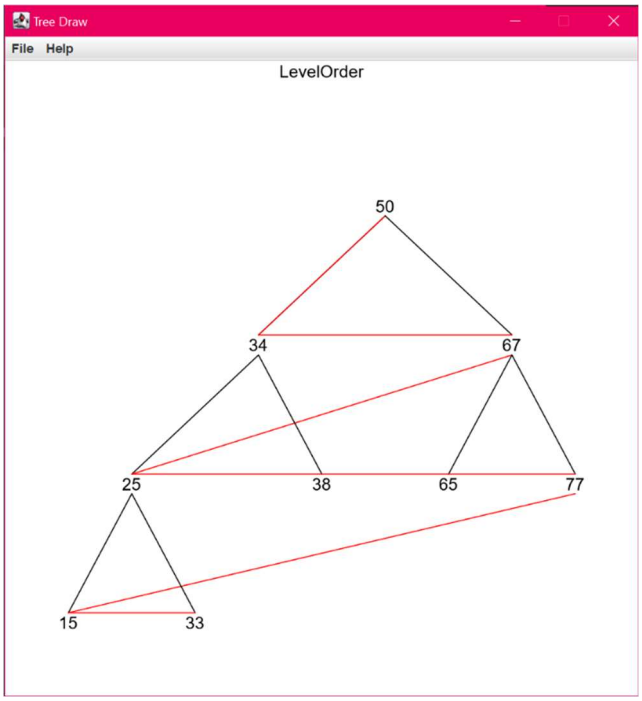


We later added an experimental path showing routine that can show the traversal path. Given below are the paths followed by all four traversals. Paths are shown in red.





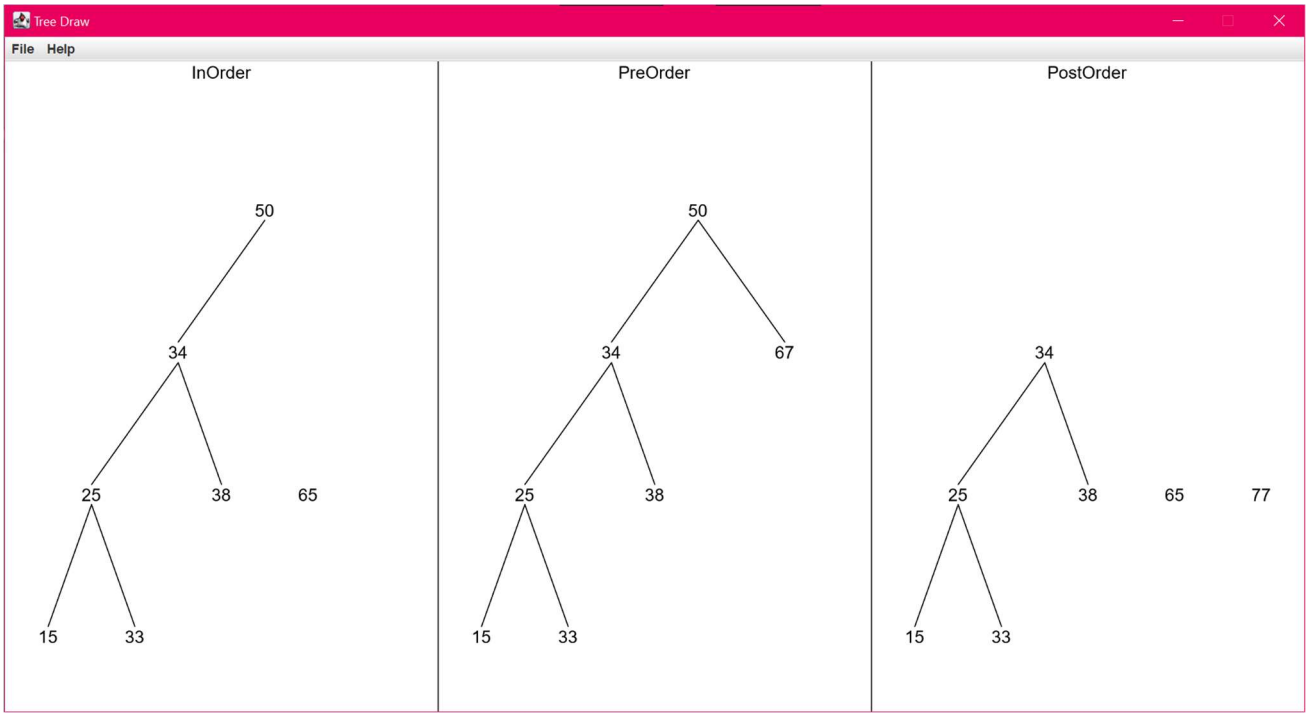




This part was little challanging as we needed to keep track of last encountered Node and draw both the path as well as the actual edges. The path drawing over the edge is not a flaw. We actually designed it to overlap so that the animation and drawing seems seamless. This function is very useful when we are talking about traversals. The path drawn is actually a signature of the traversals. No matter what sequence of numbers you input these paths will always have a similiarity between them.

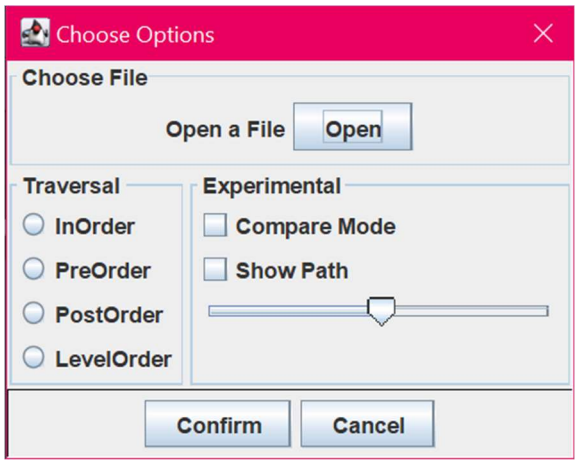
Moreover let us assume if only path drawing is given you can still correctly predict which traversal they belong to.

The other feature we added is the comparision mode. This part was the second most difficult. In this feature we had to use multithreading as we were running three methods at once. This feature development failed multiple times until we finally achieved. There were multiple problems like Thread Interference, Unanimated and unresponsive drawing etc. We can say if this will work correctly for now so we have put it in eperimental section. A glimpse of the comparision mode is below:-



This is a snapshop of trees still being drawn. We can compare in realtime how each traversal is working with this mode.

Finally we added a GUI to make things simpler. This was the most challanging part. As we were working with User Interface for the first time. Combing all the option in one simple UI was quite challanging. Eventually we completed that too.



Choose a file containing numbers.

Options for choosing various traversals and various features. The Slider button is used to control the speed.

Confirm/Camcel Buttons

References

We got the inititial inspiration from a archieved course of Colombia University CS W3137 Data Structures and Algorithms, Spring 2014.
http://www.cs.columbia.edu/~allen/S14/notes.html

For Java API References we used Oracle Java Documentation
https://docs.oracle.com/javase/7/docs/api/

Thanks