RELATIONSHIP BETWEEN NODES AND HEIGHT OF A BINARY SEARCH TREE

BY SUKHVINDER SINGH

DECEMBER 6, 2021

UNIVERSITY OF NORTH ALABAMA

Introduction

Data Structures is way to represent data in the memory in human readable format. It helps to store data in the memory in a certain format which is conveniently faster, usable with real applications and improve performance of the object. There are many Data Structures in Computer Science like – Array, LinkedList, Queue, Tree, Graph, Hash Table and some sorting algorithms in the support for the Data Structures: Quick Sort, Merge Sort, Heap Sort, Dijkstra Algorithm, etc.

We are analyzing Binary Search Tree – BST, it's a type of Binary Tree Algorithm in which you store elements in a sorted order. Each node can have nodes between 0 to 2, which makes it a binary tree and the sorted part makes it search efficient that's why it called as BST – Binary Searching Tree.

Nature of a BST Tree

There's a root node which controls all the operations basically after selecting which is root node. The smaller data should go on the left side/node of the parent node and the larger amount of data goes on the right side/node. Which makes it efficient in many ways like inserting data in a sorted manner, searching is the main focus of this tree.

There are 2 ways implementing tree as an Array or as LinkedList. LinkedList is the better way implementing a BST tree that's my opinion. The size problem is solved in the LinkedList so we don't have to worry about the sizing up or down.

Traversing: To traverse a tree we have 3 approaches: Preorder, Inorder, Postorder.

Preorder: In this traversing algorithm the Parent node is access first till counter of null pointer and after that the Left node and right node at last. It stops after traversing every node of the tree following P->L->R routine.

Inorder: In this traversing algorithm the left node is access first till counter of null pointer and after that the parent node and right node at last. It stops after traversing every node of the tree following L->P->R routine.

Postorder: In this traversing algorithm the Left node is access first till counter of null pointer and after that the Right node and Parent node at last. It stops after traversing every node of the tree following L->R->P routine.

Hypothesis Statement: The Height and Number of nodes in BST Tree are dependent on each other as one start increasing the other also increases.

Methods

The nodes in the BST tree are calculated as we calculate by formula – $n = 2^h$. Where we can define (h) as height and (n) as nodes of the tree.

If we consider a perfect balanced binary tree. So, it'll have 2 conditions:

- An actual branching factor of 2 at each inner node.
- Equal root path lengths for each leaf node.

About the leaf nodes in a perfectly balanced binary tree:

As the number of leaf nodes in the tree is the number of nodes minus the number of nodes in a perfectly balanced binary tree with a height decremented by one, the number of leaf nodes in the tree is half the number of all nodes (to be precise, half of n+1).

So, h just varies by 1, which usually doesn't make any real difference in complexity considerations. that claim can be illustrated by remembering that it amounts to the same variations as defining the height of a single node tree as either 0 (standard) or 1 (unusual, but maybe handy in distinguishing it from an empty tree).

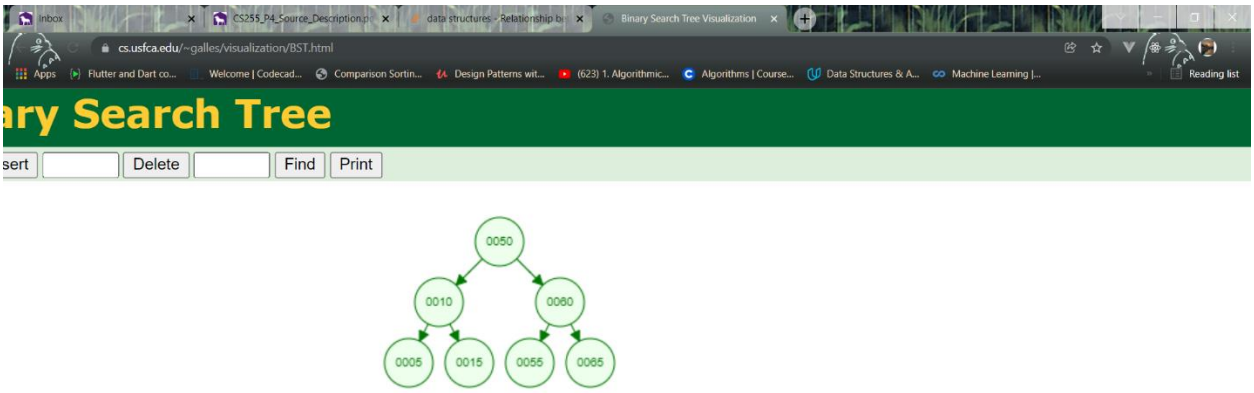
Code

“

```
if(root == nullptr){
    Node *newNode = new Node();
    newNode->data = data;
```

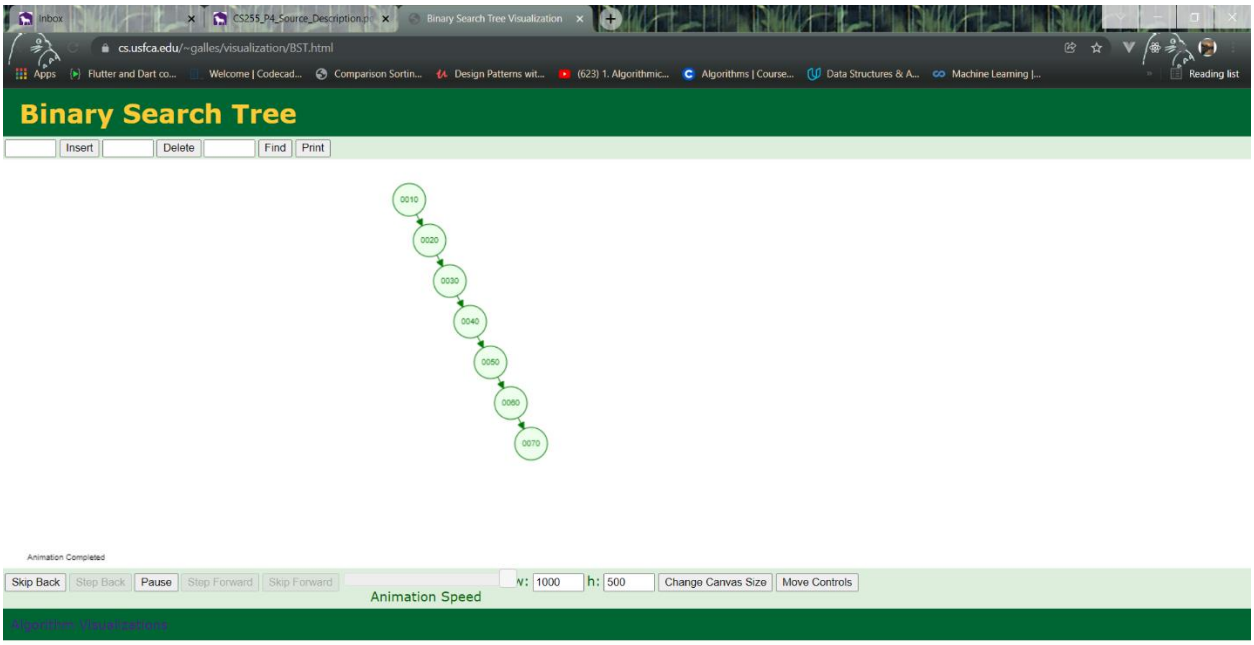
```
        root = newNode;
        nodeCounts++;
        return true;
    }else{
        if(data < root->data) insertNode(data, root->left);
        else if(data > root->data) insertNode(data, root->right);
        else return false;
    }
```

“



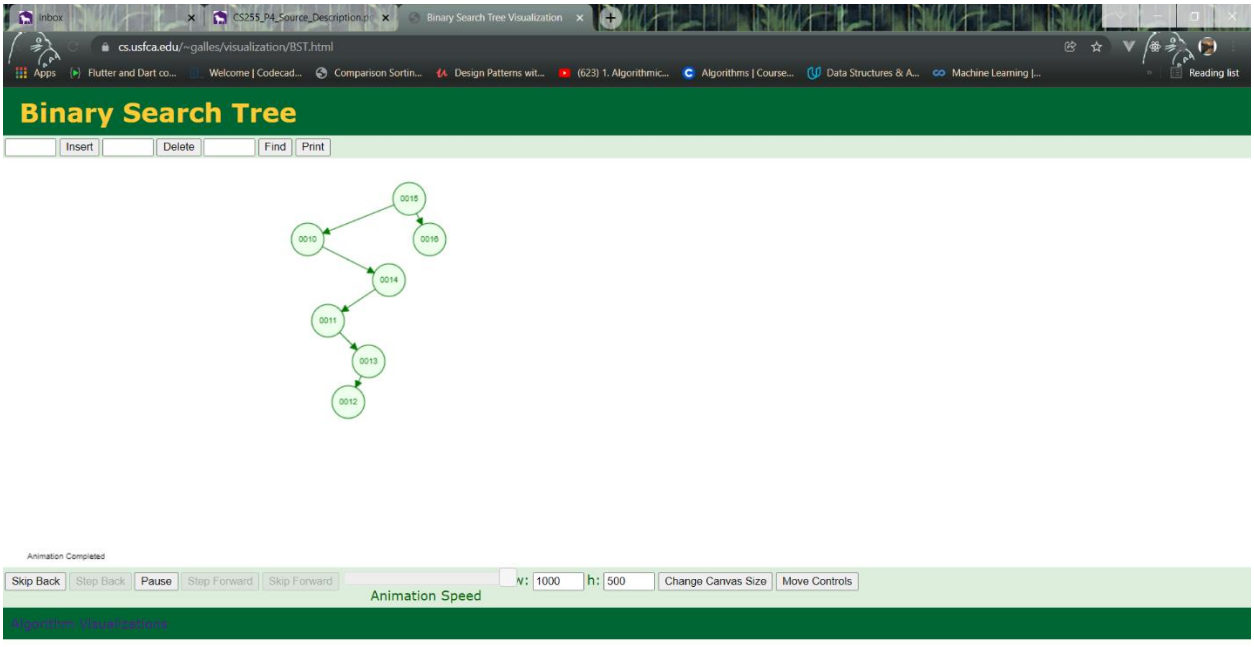
Nodes: 7

Height: 2



Nodes: 7

Height: 6



Nodes: 7

Height: 5

Results

The relation as it stands is precise. however, in applications you most certainly won't have perfectly balanced binary trees as this requires the number of nodes to be a power of 2 (-1). there do exist efficient algorithms to maintain balanced binary trees in the sense that the length of root paths for all leaf nodes in the tree would be varying by no more than 1. in particular you do not lose efficiency as compared to perfectly balanced trees. so, the relation is quite as precise as possible. In searching algorithm of BST tree, it removes half sub-tree at every step. Searching for an element in a binary search tree takes O(log2n) time. In worst case, the time it takes to search an element is O(n) which I think is better and efficient.