Trees - Notes

What is a Tree?

It is a non-linear data structure in which each node may be associated with more than one node. Each node in the tree points to its children. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

1. The root of a tree is the node with no parent. There can be only one root node in a tree.
2. A node with no children is called a leaf.
3. A node p is the ancestor of node q if the node p lies on the path from q to the root of the tree.
4. If each node of a tree has only one child then it is called skew.

The Tree data structure looks the reverse of the actual physical tree, and has root on top.

Basic Terminology in Trees

1. **Node :** It is the structure that contains the data about each element in the tree. Each element in a tree constitutes a node.
2. **Root :** The top node in a tree.
3. **Children :** A node directly connected to another node when moving away from the Root.
4. **Parent :** The immediate node to which a node is connected on the path to root.
5. **Siblings :** A group of nodes with same parent.
6. **Ancestor :** A node reachable by repeated proceeding from child to parent.
7. **Descendant :** A node reachable by repeated proceeding from parent to child.
8. **Leaves :** The nodes that don't have any children.
9. **Path :** The sequence of nodes along the edges of the tree.
10. **Height of Tree :** The number of edges on the longest downward path from root to a leaf.
11. **Depth :** The number of edges from the node to the root node.
12. **Degree :** The number of edges that are being connected to the node, can be further subdivided into in-degree and out-degree.

Binary Trees and Binary Search Trees

A binary tree is one where each node can have no more than two children. By limiting the number of children to two, we can write efficient programs for inserting data, searching for data, and deleting data in a tree.

Before we discuss building a binary tree in JavaScript, we need to add two terms to our tree lexicon. The child nodes of a parent node are referred to as the left node and the right node. For certain binary tree implementations, certain data values can be stored only in left nodes, and other data values must be stored in right nodes.

Identifying the child nodes is important when we consider a more specific type of binary tree, the binary search tree. A binary search tree is a binary tree in which data with lesser values are stored in left nodes and data with greater values are stored in right nodes. This property provides for very efficient searches and holds for both numeric data and non-numeric data, such as words and strings.

Implementation Of Binary Search Tree (BST)

A binary search tree is made up of nodes, so the first object we need to create is a Node object, which is similar to the Node object we used with linked lists.
The definition for the Node class is:

```
function Node(data, left, right) {
this.data = data;
this.left = left;
this.right = right;
this.show = show; }
function show() {
return this.data;
}
```

The Node object stores both data and links to other nodes (left and right). There is also a show() function for displaying the data stored in a node. Now we can build a class to represent a binary search tree (BST). The class consists of just one data member: a Node object that represents

the root node of the BST. The constructor for the class sets the root node to null, creating an empty node.

The first function we need for the BST is insert(), to add new nodes to the tree. This function is complex and requires explanation. The first step in the function is to create a Node object, passing in the data the node will store.

The second step in insertion is to check the BST for a root node. If a root node doesn't exist, then the BST is new and this node is the root node, which completes the function definition. Otherwise, the function moves to the next step.

If the node being inserted is not the root node, then we have to prepare to traverse the BST to find the proper insertion point. This process is similar to traversing a linked list. The function uses a Node object that is assigned as the current node as the function moves from level to level in the BST. The function also has to position itself inside the BST at the root node.

Once inside the BST, the next step is to determine where to put the node. This is per- formed inside a loop that breaks once the correct insertion point is determined. The algorithm for determining the corrent insertion point for a node is as follows:

1. Set the root node to be the current node.
2. If the data value in the inserted node is less than the data value in the current node, set the new current node to be the left child of the current node. If the data value in the inserted node is greater than the data value in the current node, skip to step 4.
3. If the value of the left child of the current node is null, insert the new node here and exit the loop. Otherwise, skip to the next iteration of the loop.
4. Set the current node to be the right child of the current node.
5. If the value of the right child of the current node is null, insert the new node here and exit the loop. Otherwise, skip to the next iteration of the loop.

BST & Node Class

```
function Node(data, left, right) {
this.data = data;
this.left = left;
this.right = right;
this.show = show; }
function show() {
return this.data;
}
function BST() {
```

```
this.root = null;
this.insert = insert;
this.inOrder = inOrder;
}
function insert(data) {
var n = new Node(data, null, null);
if (this.root == null) {
this.root = n;
}
else {
var current = this.root; var parent;
while (true) {
parent = current;
if (data < current.data) {
current = current.left; if (current == null) {
                parent.left = n;
break; }
} else {
current = current.right;
if (current == null) {
parent.right = n;
break; }
}}}}
```

Traversing a Binary Search Tree

There are mainly four traversal functions used with Binary Trees and BSTs: inorder, preorder, and postorder.

An inorder traversal visits all of the nodes of a BST in ascending order of the node key values. In this each node is processed after all nodes in between its left subtree, and its right subtree.

```
function inOrder(node) { if (!(node == null)) {
          inOrder(node.left);
          console.log(node.show());
          inOrder(node.right);
       }
}
```

A pre-order traversal visits the root node first, followed by the nodes in the sub-tree under the left child of the root node, followed by the nodes in the subtrees under the right child of the root node.

```
function preOrder(node) { if (!(node == null)) {
          console.log(node.show());
          preOrder(node.left);
          preOrder(node.right);
} }
```

A post-order traversal visits all of the child nodes of the left sub-tree up to the root node, and then visits all of the child nodes of the right subtree up to the root node.

```
function postOrder(node) { if (!(node == null)) {
          postOrder(node.left);
          postOrder(node.right);
          console.log(node.show());
} }
```

A level-order traversal visits each node level by level from left to right before any nodes in their subtrees with the help of queue data structure.

```
function levelOrder() {
        var queue = [];
        queue.push(root);
        while (queue.length != 0) {
            var tempNode = queue.shift();
            console.log(tempNode.data);
            /* Enqueue left child */
            if (tempNode.left != null) {
                queue.push(tempNode.left);
            }
            /* Enqueue right child */
            if (tempNode.right != null) {
                queue.push(tempNode.right);
            }
```

```
        }
    }
```

Searching for A Value In BST

Searching for a specific value in a BST requires that a comparison be made between the data stored in the current node and the value being searched for. The comparison will determine if the search travels to the left child node, or to the right child node if the current node doesn't store the searched-for value.

We can implement searching in a BST with the find() function, which is defined here:

```
function find(data) {
var current = this.root;
while (current.data != data) {
if (data < current.data)
{ current = current.left;}
else { current = current.right; }
if (current == null) { return null; }
}
return current; }
```

This function returns the current node if the value is found in the BST and returns null if the value is not found.

Applications Of Tree Data Structure:-

1. File System Inside The Computer.
2. Organizational Structure Of Employees Inside An Organization.
3. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
4. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
5. In Trie Data Structure which is used to implement dictionaries with prefix lookup.
6. Machine learning algorithm.
7. For indexing in the database.