**Data Structures and Algorithms: Tree**

Tree Basic:

* Trees data structure is like a Tree. They have roots and when added data called branches.
* A big tree fan out with lots of different branches in different directions
* Different terms related to tree:
  + They have leaves
  + The collection of trees called forest etc.

Trees have a lot of properties that make them useful, however, tree is an extension of Linked list

|  |  |
| --- | --- |
| Linked Lists | Trees |
| Linked List has one node next to previous node.  Next 2:  Next 1: | Tree can have various nodes  Root  Branches  Leaves |
| Linked List Often drawn horizontally | Tree is verticle |
| Each element contains data | In tree also each element contains data |

Tree must be connected: that means when you start from the root, there must be some way to connect each node of the tree.

There must not be any cycles on the tree. The cycle can be possible this way:

Tree Terminology: Ancestor Level

P = Parent

C = Child

Descendant

**“Children is only allowed to have one parent, and parent can have multiple child”**

The node that don’t have any children are called Leaf. The height of a node is the number of edges between it and the furthest leaf on the tree. Here, a leaf has a height of zero and the parent has 1 and the root has 2. The height of a tree overall is just a height of the root node. On the flip side, the depth of a node is the number of edges to the root. Height and depth should move inversely. If a node is closer to a leaf, then its further from the root.

Tree Traversal:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H |

For tree traversal we just needed to go through the list in the order to make sure that we looked at every element. Trees are not linear so there is no clear way to traverse through everything.

Suppose we start at the root; do we go left first or right first? Should we completely traverse one subtree or one section of the tree including a parent and all its descendants or traverse everything at the same level first?

We can’t search or sort elements unless we have a way to make sure we can visit all element first. There are two different broad approaches to treat traversal.

* One is called depth-first search or DFS for short
  + In DFS, the philosophy is if there are children nodes, to explore, exploring them is the priority.
* The another one is called breadth-first search or BFS.
  + In BFS, the priority is visiting every node on the same level we’re currently on before visiting child nodes.

BFS, and DFS are kind of vaguely defined since we can apply their principles but actually traverse the tree in several different ways. For trees a level order traversal is a BFS with a more exact algorithm to implement.

A level order starts at the root then visit its children on the second level then all of their children on the third level until you’ve visited every leaf. By convention, we start on the left most side of the level and move right. This traversal is definitely a BFS.

There are several different approaches to DFS and Trees.

Pre-Order traversals: We start at the route and immediately, check off that we’ve seen it. Next, we pick one of the children. Normally the left one by convention, check it off too . We would continue traversing down the left most nodes until we had a leaf. We check of the leaf and from there go back up to the parent. Now we can traverse to the right child and check it off too.

Once, we are done with this subtree, so we can travel back up to the root and look at its right child. We do the same steps until we done everything.

DFS- In order traversal: we will move through the node in the same order, since this is still DFS and we need to explore all the children first. However, this time we’re going to check off the nodes in a different order. We are only going to check off a node when we’ve seen its left child and come back to it.

Again, we start at the root, since we haven’t seen the left child yet. We have to keep traversing down until we hit a leaf. We check off the leaf and move up to the parent.

Because, we have seen the left child now we can actually check off the parent this time. We move out to this right node, which has no children, so we can check it off too. We go back up to the root and repeat all of this on the right side until we’re done.

Post-Order traversal: We won’t be able to check off a node until we’ve seen all of its descendants or we visited both of its children and returned. Similar steps, we begin at the root, don’t check it off, but continue down to the leaf. We check off the leaf and move to the parent. This time we don’t check off the parent through, we just move on the right node. Once we’ve checked off the right child we can go back up to the parent and finally check it off too. Again, we’ll skip over the root node and just move all the way down to the right. Once, everything there is good we can move back up to the root and get it. Binary Trees: Binary trees are simply trees for where parents have at most two children. That means nodes an have 0,1 or 2 children.

F has 2 nulls

Search in Binary Tree: We could start off by using any of the traversal algorithms to go through the tree. Because there is no real order to the element, its needed to go through every single element in the tree if the required value doesn’t exist. Time complexity O(n). linear time search.

Delete in Binary Tree: A delete operation often starts out with a search since you need to find the node you want to delete. If you are deleting a leaf, you can simply delete it and move on. However, if you delete an internal node, you’ll suddenly have a gap in the tree. If the node you deleted only has one child, you can actually just take it out move the child up and attach it to the old node’s parent. If you are trying to delete the node that has two children, you actually have few options. You can promote the child up. **What if both children also have two children?** In the worst case you’ll need to keep traversing down to the sub tree until you hit a leaf. Since there is no real order required here you can just put the leaf where your deleted node without a problem. Since there is a search involved and some additional work to shift around the elements after deletion, the run time is linear.

Insert: Inserting an element in to our tree when it has no order is relatively easy. We will just tuck our node in to another node. Maybe it’s a leaf or parent only one child. We only need to make sure that we’re obeying the two children rule. We’re given the root at the beginning and we will need to keep moving down the tree until we find an open spot.

Root **How long it will take to find an open spot?**

The worst case is that we travel down the longest path until we

Find the farthest leaf. In that case, we’ll travel through the

number of nodes equal to the height of the tree.

**What is the height of a binary tree?**

**Perfect trees:** Because every node has two children (3 Nodes

2 Levels ) Looks like (7 Nodes

3 Levels)

As the tree grows bigger, each new level has the capacity to hold a number of nodes equivalent to a power of two. Each node can have two children, so each new level can have twice as many as the one before it.

Log(n) doesn’t work here.

Binary Search Trees: Trees inherently aren’t really organized. Where we know that how the structure looks like, but we don’t know that where the specific element will be.

Some Specific Rules Regarding the Ordering of Elements.

**BST:** There’s more specific type of binary tree called a binary search tree. Every parent node has at most two children. BST is just a binary tree. There’s a specific rule of how the values associated with each node are arranged. BSTs are sorted, so every value on the left of a particular node is smaller than it and every value on the right of a particular node is larger than it. Because BSTs have this structure, we can do operations like search, insert, and delete pretty quickly.

Let say we want to find 7. We would start at the root. We see 7 is larger than 5 so we go right next. 7 < 8 so we go left. And then 7 is found. We don’t need to search every element to figure out where the we were looking for belonged. We just had to look at one value in each level of the tree, and then we can just make decision with just comparison to the element we were looking for (here it is 7). That means the run time of a search on a BST is just the height of the tree, which was log(n).

**Inserting** is pretty much the same process. Starting from top, make decisions about where to look at each step by comparing to the element you want to add. Eventually, you’ll hit that open spot in the tree. As long as you compared your element correctly to each step, you can put your new element there and not violate the core BST properties. O(log(n)).

**Deleting** is bit complicated.

BST Complications: Binary trees are nice to look at when they are full, but there is no rule that states that BST need to look that way.

This is a completely legitimate BST, but it looks different.

There are two or fewer children for every parent, and everything on the right is bigger than its parent. It is called an unbalanced binary tree. Since the distribution of nodes is skewed to the right side. A weird structure like this could start at the root, but it also can take place in one of the sub trees. It can also be considered as the worst-case scenario of BSTs. When the tree looks like this; search, insert and delete al actually take linear time in the worst case. Here, the worst case would be searching through every element from the root to the leaf. Thus, the average case for these operations is Big O for log n, and the worst case for all of them is Big O of n.

Pros: Trees are faster retrieving data because you don’t have to touch unnecessary nodes.

Tree based machine learning model (Tree-base machine learning)

The average time complexity for the tree operations is log (n).

**Pre-Order Traversing: Breadth First Search:** BFS is vising one level at a time.