Trees and Traversals

Table of Contents

Sets and Maps

There are two very common categories of collections we have seen and used but may not have formally leveraged.

- Sets
- Maps

Sets

A Set is a collection of elements where each element is unique. This means that you do not have duplicate entries in the set. The following are the basic operations for sets:

- Intersection - Given two sets (A and B), an intersection of both sets would result in another set in which elements in A are also in B.
- Union - Given two sets (A and B), the union of two sets will result in all elements of A and B in a single set.

Sets

- Difference - Given two sets (A and B), the difference (A \ B) is all elements in A that are also not in B. (This is not the same as symmetric difference).
- Symmetric Difference - Similar to difference but it is the union of the:
  A \ B and B \ A. For all elements in A that are not in B and for all elements in B that are not in A.

Relation to Programming

You'll find that typescript and javascript have a `Set` datatype that you can use, similar to a collection like a `Dictionary` in which you can think of the items as keys.

Tangent - Set Theory and Mathematics

While outside of the scope of this subject, I would encourage all to look into set theory more deeply as a way to continue and develop a deeper understanding of mathematics and computer science.

Specifically, this can be made fun with Python's set-builder notation and being equipped with how to express and write sets by hand.

However, set-theory and ZF-Set theory has been the standard for proofs and literature in many fields of mathematics.

Maps

Similarly to sets, Maps have the same constraint where each key is considered unique. However, Maps are an association between a key and a value (conventionally, expressed as K and V).

Trees

In computer science, a tree is an abstract model of a hierarchical structure:

- A tree consists of nodes with a parent-child relation
- If n is parent of m, then m is a child of n
- A node has at most ONE parent in a tree
- A node can have zero, one or more children

Trees

Common terms with trees:

- Root: Node without a parent
- Ancestors of a node: itself, parent, grandparent
- Descendant of a node: Itself, any child, any grandchild.
- Subtree: Tree consisting of a node and a its descendants, typically a tree that is inside a tree.

Trees

- Depth of a tree, number of ancestors not including itself
- Level of a tree, The set of nodes with a given depth
- Height of a tree, maximum depth of any node

Tree Traversals

We will look into two kinds of traversals:

- Pre-order
- Post-Order

These are ways that we can navigate the tree and the order we have within it.

Pre-Order Traversal

```
preOrder(node) {

    visit(node)

    foreach child in node.children {
        preOrder(child)
    }

}
```

Hang on!? Why is PreOrder calling itself?

We have a recursive function, it can be an elegant way to demonstrate an algorithm however whenever you see a recursive function, the proper implementation will likely substitute a `Queue` or `Stack` instead.

Post-Order Traversal

```
postOrder(node) {

    foreach child in node.children {
        postOrder(child)
    }

    visit(node)
}
```

There isn't much of a change, the main focus here is we are visiting the children first and then visiting the current node.

Binary Trees

While we have talked about trees generally, we will focus on a special version of a tree. A Binary Tree.

A binary tree is where a node within a tree only has two children.

- Left
- Right

We can apply the same traversal function for pre-order or post-order traversal.

In-Order Traversal

A particular traversal method we can apply here is an `inOrder` traversal.

```
inOrder(node) {

    if (node.left != null) {
        inOrder(node.left);
    }

    visit(node)

    if (node.right != null) {
        inOrder(node.right);
    }
}
```

Consider the similarity with binary search

Binary Tree Properties

There are a few properties relevant to binary trees which are important to understand. A significant reason is because it helps understand why certain constructions can be lop-sided or poorly formed (a linked list is a kind of a binary tree).

Properties

We can have:

- A rooted binary tree where each node has at most two children
- A full binary tree (or proper) binary tree where each node has either 0 or 2 children.
- A perfect binary tree, where all interior nodes have two children and all leaves have the same depth/same level.

Properties - Continued

- A complete binary tree, in which every level, except the last, is completely filled.
- A balanced binary tree, where the left and right subtree can differ in height but by no more than 1.

Heaps

A heap is a complete binary tree, where the height of the heap in which all levels are at the maximum number of nodes that can be present except for the last.

Heaps, also have a property where, given a key k and key p, where k is a key associated with an entry in the heap and p is the key associated with the parent entry in the heap, k must be >= p.

This results in an order where the children of a node must be larger than node itself.

Heaps

A typical use-case for heaps is for what is known as a `priority-queue`. If we know the `min` (or `max`) of a collection and can quickly insert and remove those elements, we have an efficient way of *sorting*.

Try and design a heap for yourself in one of the exercises.

Exercises

1. Constructing a binary tree, construct a class called `BinaryTree` along with a class called `BinaryTreeNode`.
   Each `BinaryTreeNode`
   - Specify a `key` field which can be also compared again
   - Specify a `value` field to hold the object
   - Specify left and right fields which are also `BinaryTreeNode`'s
   - Consider how you will compare two keys

Exercises

1. Continued

With a `BinaryTree` class:

- Make sure there is a `root` field part of the `BinaryTree`.
- Implement `insert`, `get` and `remove` methods
- Implement `preOrder` and `postOrder` traversal

Consider creating an interface that will be associated with the `key` type.

Resources

1. Binary Tree, Yale CS, https://www.cs.yale.edu/homes/aspnes/pinewiki/BinaryTrees.html
2. Tree-Traversals, Wikipedia, https://en.wikipedia.org/wiki/Tree_traversal
3. BFS & DFS, CS225 - Grainger College - Chicago, https://courses.grainger.illinois.edu/cs225/sp2025/resources/bfs-dfs/