📝

Rust Take home Exercise: Merkle Trees

Note. For this exercise you should write as you would normally write public production facing code. That means including tests, input validation, documentation, and appropriate handling of edge cases. Please work on your solution in a private Github repository and use git as you would normally. Keep track of the time spend and limit yourself to 4 hours. When you are done, invite `@r3ld3v, @dzejkop,` `@recmo`, and `@BioMark3r` to the private repository.

Make sure to write your implementation in Rust, we suggest using the sha3 crate but you are free to use anything else.

Binary Merkle Trees

Given a binary tree:

```
flowchart TD
    subgraph root
        0(["0"])
    end
    subgraph intermediate nodes
        1(["1"])
        2(["2"])
        3(["3"])
        4(["4"])
        5(["5"])
        6(["6"])
    end
    subgraph leaves
        7(["7"])
        8(["8"])
        9(["9"])
        10(["10"])
        11(["11"])
        12(["12"])
```

```
        13(["13"])
        14(["14"])
    end
    0 --> 1
    0 --> 2
    1 --> 3
    1 --> 4
    2 --> 5
    2 --> 6
    3 --> 7
    3 --> 8
    4 --> 9
    4 --> 10
    5 --> 11
    5 --> 12
    6 --> 13
    6 --> 14
```

The single top node is called the *root*, the lowest nodes are called *leaves* and the remaining nodes are called *intermediate nodes*. Layers are counted zero based from the root, called the *depth*. The number of layers is called the depth of the tree. The nodes are numbered zero-based left-to-right and top-to-bottom called their *index*.

Index calculus

In addition to the index, we can also identify the nodes by their depth and offset from the left:

```
flowchart TD
    0(["(0,0)"])
    1(["(1,0)"])
    2(["(1,1)"])
    3(["(2,0)"])
    4(["(2,1)"])
    5(["(2,2)"])
    6(["(2,3)"])
    0 --> 1
    0 --> 2
    1 --> 3
    1 --> 4
    2 --> 5
    2 --> 6
```

Exercise 1. Write a function that returns for a given $(\mathrm{depth}, \mathrm{offset})$ returns the corresponding index.

Exercise 2. Write three separate functions that given an index,

1. return the $(\mathrm{depth}, \mathrm{offset})$,
2. return the index of the parent, and
3. return the index of the left-most child.

Merkle Arboriculture

To construct a Merkle tree, we assign hash values to each node as follows: Each leaf gets assigned a value and the other nodes have their value computed using the recursive definition:

$$\mathsf{hash}(\mathrm{node}) = \mathsf{SHA3}(\mathsf{hash}(\mathsf{leftChild}(\mathrm{node})||\mathsf{hash}(\mathsf{rightChild}(\mathrm{node}))))$$

where $||$ means concatenation.

Exercise 3. Write a data structure to store node hashes. The constructor should take as arguments the desired tree depth and a single initial leaf value to assign to all leaves. Assume $\mathsf{depth} < 30$ and it is feasible to store all nodes in memory. Add a method to return the root hash of the tree. Here's an example usage:

Hint. Initialization can be done with only $O(\mathrm{depth})$ invocations of $\mathsf{SHA3}$.

```
initial_leaf = 0xabababababababababababababababababababababababababababababababab
tree = MerkleTree::new(depth = 20, initial_leaf = initial_leaf)
assert tree.root() == 0xd4490f4d374ca8a44685fe9471c5b8dbe58cdffd13d30d9aba15dd29efb92930
```

Exercise 4. Add a `set` method that updates a single leaf value and recomputes any affected nodes. Come up with test values for this method

The raison-d'etre of Merkle trees is to *commit* to an array of size n through the root hash and then proof the value of entry i through a $O(\log n)$ sized proof. The proof consists of the path from leaf to the root and all the sibling hash values along the way.

Exercise 5. Create a `proof` function that given a tree and a leaf index returns a merkle proof for that leaf.

```
initial_leaf = 0x0000000000000000000000000000000000000000000000000000000000000000
tree = MerkleTree::new(depth = 5, initial_leaf = initial_leaf)
for i in 0..tree.num_leaves():
    tree.set(i, i * 0x1111111111111111111111111111111111111111111111111111111111111111)
assert tree.root() == 0x57054e43fa56333fd51343b09460d48b9204999c376624f52480c5593b91eff4
assert tree.proof(3) == [
    right, sibling = 0x2222222222222222222222222222222222222222222222222222222222222222
    right, sibling = 0x35e794f1b42c224a8e390ce37e141a8d74aa53e151c1d1b9a03f88c65adb9e10
    left,  sibling = 0x26fca7737f48fa702664c8b468e34c858e62f51762386bd0bddaa7050e0dd7c0
    left,  sibling = 0xe7e11a86a0c1d8d8624b1629cb58e39bb4d0364cb8cb33c4029662ab30336858
]
```

Note that the leaf value, path and sibling hashes are **all** the information you need to recompute the root hash.

Exercise 6. Create a `verify` function that takes a leaf value and Merkle proof and returns a root hash.

```
initial_leaf = 0x0000000000000000000000000000000000000000000000000000000000000000
tree = MerkleTree::new(depth = 5, initial_leaf = initial_leaf)

for i in 0..tree.num_leaves():
    tree.set(i, i * 0x1111111111111111111111111111111111111111111111111111111111111111)

leaf_5 = 5 * 0x1111111111111111111111111111111111111111111111111111111111111111

root = tree.root()
proof = tree.proof(3)

assert verify(proof, leaf_5) == root
```