Merkle Tree Design Document

By Alyson Ngonyama

Table of Contents

Project Files:

Github Repo: https://github.com/AlysonNumberFIVE/MerkelTree

Summary

This is an implementation of my Merkle Tree design document accompanying the code in the codebase linked in the **Repository** section above. Design a Merkle Tree that can insert hashes, update data, look it up and has a layer of 128 bit security implemented in each of the nodes.

IMPORTANT: This document is intended to be read **alongside** code exploration and should not be read as a substitute as not every line of code is meticulously documented in the code section of this document. I highlight the details that I suspect might be a bit erroneous in logic to give walkthroughs and/or reasoning behind certain decisions I think will be of interest to the reader.

Goal

Implement a basic Merkle Tree that has the following functionality:

- **Create/Insert** - Create a new leaf and add it to the Merkle Tree
- **Update** - Change the data in an existing leaf node
- **Lookup** - Query the data for a node
- **Generate Proof** - Create a proof for a leaf node
- **Verify Proof** - Verify the existence of a leaf node in the tree

A “main” function isn’t included. This tree can be considered a backend API. For usage examples, see merkel_tree_tests.go for detailed execution details.

Non-Goals

Implementation of custom hashing functionality is omitted as the use of a hashing library is permitted.

The structure of the data required to add to the Merkle Tree is also not implemented; **duplicate data detection is implemented** but malicious tampering is assumed if this is detected as, unlike in this assignment, timestamping would be added to the data to make it unique.

The data in unittests are simple “A”, “B”, “C”s etc.

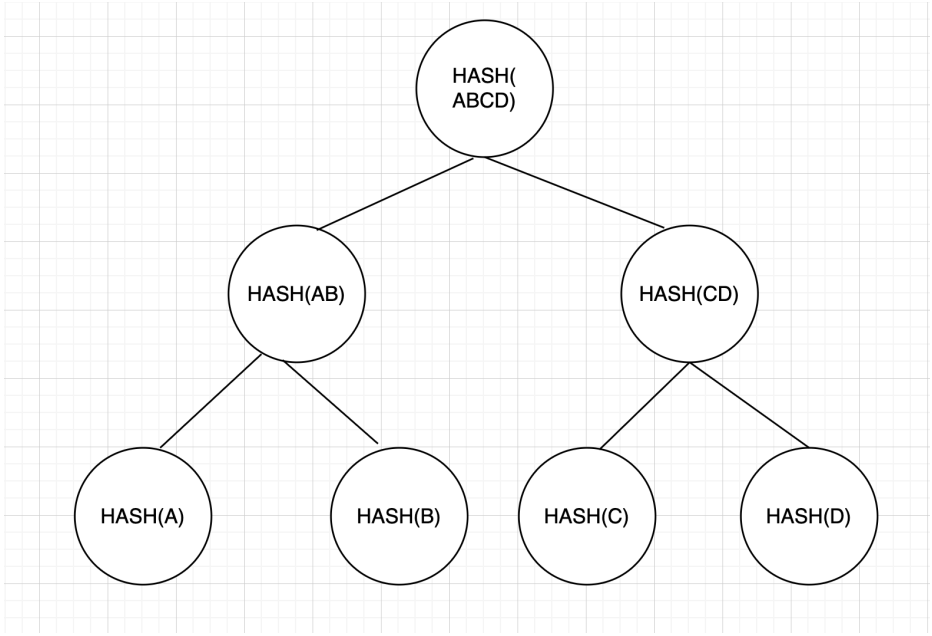
Project Implementation

The following sections go into technical implementation details, ideas considered and discarded, as well as security considered and any other miscellaneous information that will be helpful in understanding this Merkle Tree's design.

Overview: Merkle Tree - What Is It

A Merkle Tree is a binary tree structure designed to hold data in its leaf nodes. Each of the nodes; from the root node all the way down to each of the leaves all contain an individual hash signature. These hashes are generated by concatenating the hashes of each of the child nodes from the leaves all the way to the root, with the exception of the leaf nodes themselves; their hashes are individual to their own data, often created from the data they hold internally.

This ensures security by design as any data update to leaf nodes creates a chain reaction of hash updates, ensuring an update trail exists within the tree.



Example diagram of a Merkle Tree's hashing pattern/strategy

IMPORTANT: Only the leaves of a Merkle Tree contain data. Branch nodes only contain the hashes of their children.

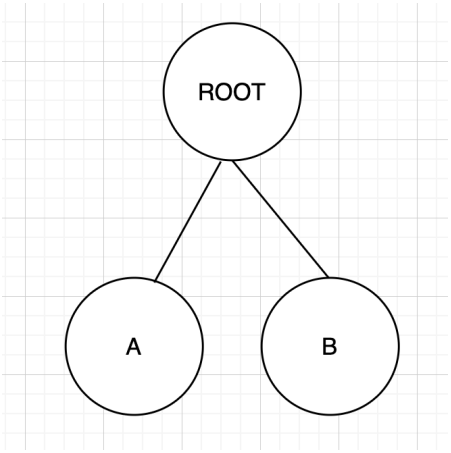
Merkle Tree Algorithm Overview

A Merkle Tree is designed to only ever have data in its child nodes, so when new data is created and the tree's structure needs to be update, unlike traditional binary trees where a node with data would simply create a new data node and parent itself to it, a Merkle Tree needs to:

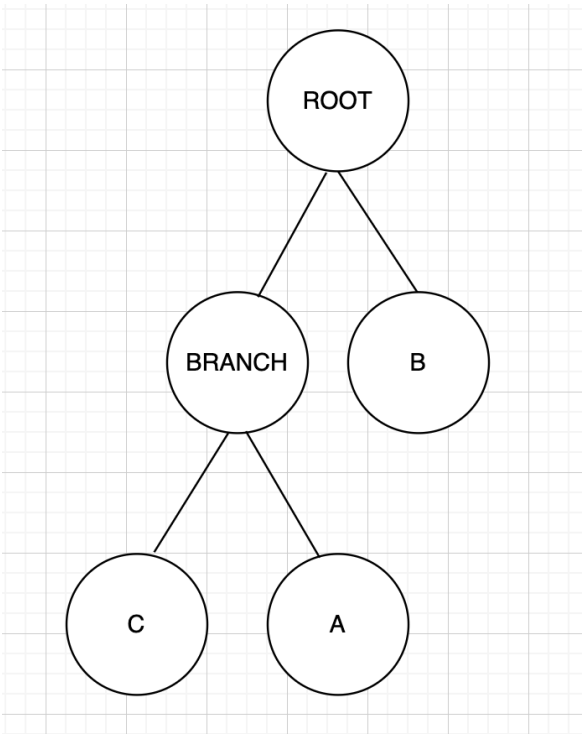
- Create a branch node
- Attach the newly created leaf node to this branch node's **left branch**
- Attach the already existing leaf's node to this branch node's **right branch**

(the branch directions "left" and "right" for newly created data/already existing data is used for simplicity of explanation):

Before data insertion:

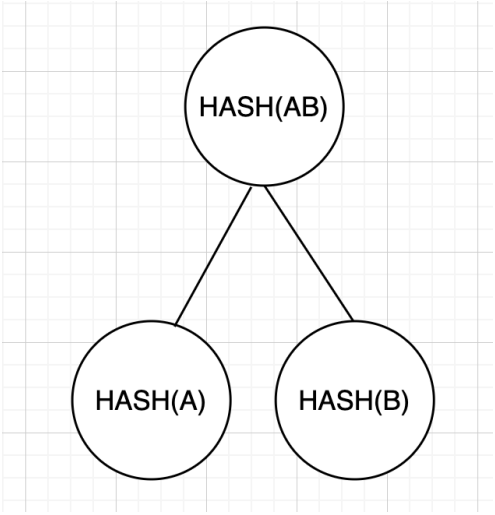


After Node "C" is added, a new BRANCH node is created and A is demoted to this new BRANCH node's children.

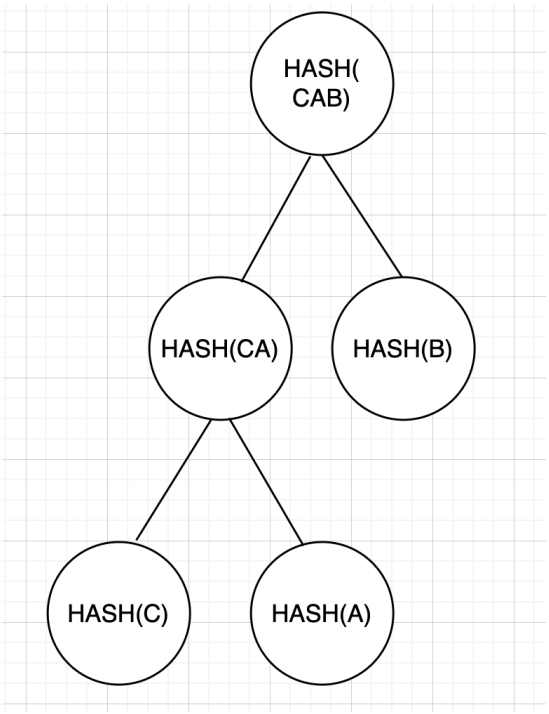


Likewise, once C has been created, all the hash nodes parenting it up all the way to ROOT will have their hashes updated.

Before the insertion of Node C:



After C node insertion:



Merkle Tree Security Integrity and Data Verification

One of the important functions of a Merkle Tree is the ability to verify the existence of data publicly without revealing the actual data itself.

The ability to do this is built into the design of the tree itself and highlights the importance of the hashing methodology (a branch's hash being the concatenation of its children's hashes all the way to the leaves), and the fact that the tree is explicitly a balanced binary tree.

This process can be separated into two specific operations;
Proof Generation - This involves generating data about a leaf that can be used by the Merkle Tree to prove the existence of the data.
Proof Verification - This involves determining that a given leaf does indeed exist in the Merkle Tree.

Proof Generation

The algorithm to generate a proof is as follows;

1. **Start at the leaf node and save its hash in a list.**

This initializes the proof with the hash of the leaf node for which you're generating the proof.

2. **Traverse one node upwards to the leaf node's parent.**

Move to the parent node of the current node. This step prepares for determining which sibling's hash to include next.

3. **In the parent node, determine the branch you just navigated from.**

Identify whether you came from the left or right child node of the parent.

4. **Save the hash from the opposite branch of the parent node in the list.**

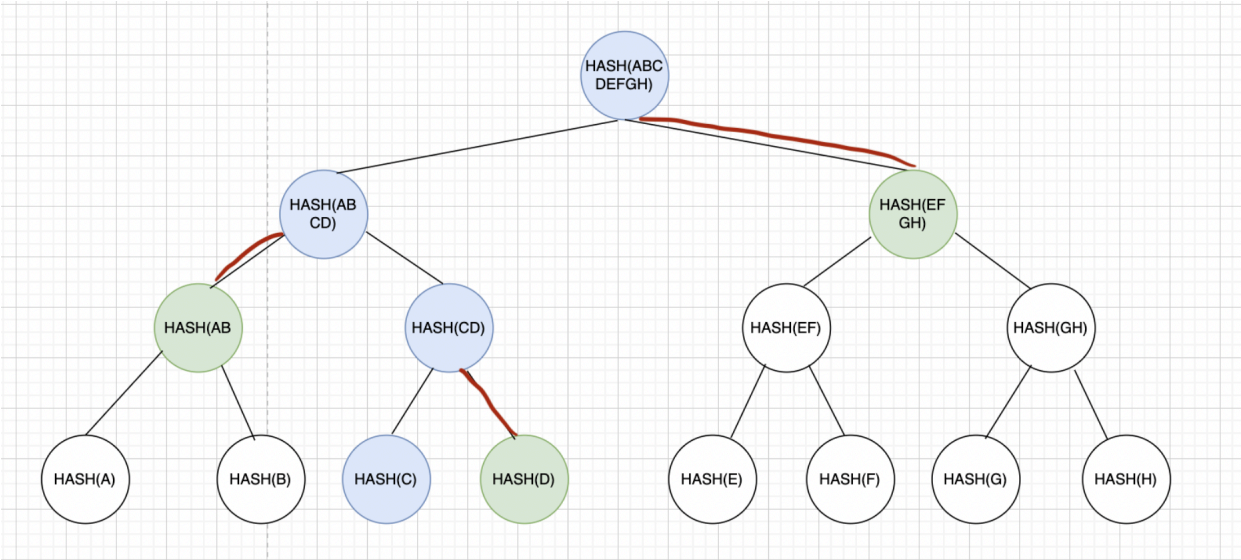
If you came from the left child, save the hash of the right child (and vice versa). This ensures that the proof contains the necessary hashes to validate the path up the tree.

5. **Repeat this process all the way up to the root.**

Continue moving up the tree, adding the opposite child's hash at each parent node until you reach the root.

The saved hashes in your list, when concatenated, will form a hash that equates to the root node's hash. This allows you to prove that the original leaf is part of the tree.

A visual representation of the proof generation algorithm:



Green nodes are the hashes we'll be collecting
Blue nodes are where "we" (the visitor) are at each step.
Red lines lines are our directions; 1 = LEFT TURN, 0 = RIGHT TURN
Our list starts with [C], our turn list is at [1]
Starting at Hash(C), we move up to Hash(CD)
We grab Hash(D)
Hash(D) is at a RIGHT TURN, we add 0 to our turn list
Our list contains [C, D], our turn list contains [1, 0]
From Hash(CD) we move up to Hash(ABCD)

We grab Hash(AB)
Hash(AB) is at a LEFT TURN, we add 1 to our turn list
Our list contains [C,D, AB], [1,0,1]
From Hash(ABCD), we move up to Hash(ABCDEFGH)
We grab Hash(EFGH)
Has(EFGH) is at a RIGHT TURN, we add 0 to our turn list,
Our list now contains [C,D, AB, EFGH], and our turns are [1,0,1, 0]

Our proof is represented by this list that contains all the hashes needed to prove that C is indeed part of this Merkle Tree.

IMPORTANT: alongside saving the list of hashes, we also save a list of the direction each of those hashes comes from in a new list. This list is often a boolean/binary list; **TRUE/1** representing a **left-turn** in the tree, **FALSE/0** representing a **right-turn**. This is important for reconstructing the path from root to the verifying leaf when proving authenticity.

So Node Hashes: [C,D, AB, EFGH]
Directions [1,0, 1, 0]

This is important to keep track of as the order of TRUE/FALSE/right/left turns changes depending on the direction of the branch the proving leaf begins on.

Proof Verification

On the flipside of generating a proof, we would also need to verify the validity of a proof. This algorithm is the proof generation in reverse. This time we start from the root node and with the root node's hash and using the Node Hashes list from the previous example, as well as the directions we'd taken, verify that the hashes at each step correlate with the hashes in our saved list.

The numbers will also help us determine how we rebuild our hash; 1, being a right join, 0, being a left

So Node Hashes: C, [D, AB, EFGH]
Directions [,1, 0, 1]

1 CD, 0 ABCD, 1 ABCDEFGH

Using the lists above we have the following instructions for reaching the leaf node for C

Step 1: 0 = LEFT TURN, RIGHT JOIN [CD] - we join D to C from the right, making it CD
Step 2: 1= RIGHT TURN, LEFT JOIN [AB, CD] - we join AB from the left of CD, making it ABCD
Step 3: 0 = LEFT TURN, RIGHT JOIN [ABCD, EFGH] - we join EFGH from the right, making it ABCDEFGH;

At this point, we are at the leaf node of C all the way from root and we have verified that it is indeed correct

Code Implementation

The following goes into the details of my code implementation in Golang.

Considerations

Before beginning with my implementation, I present a list of the design decisions i made that are unique to my tree as well and why I considered them to be important.

Left-leaning node bias
In my implementation, I've prioritized that all new nodes generated are attached to the **left branch of the parent**. This had some unforeseen consequences where the right child interaction had to be tailored in some circumstances. These will be highlighted in the code section below.

Reason: Preference.

1. **Function implementation errs on the side of specificity and explicitness findHash() vs Lookup()**

Because of the nature of sensitive data, my code is written to avoid the reliance on side effects of functions, i.e the function Lookup()` can also verify that a hash for a specific node exists but because Lookup() also returns node data, a separate findHash() is implemented with the specific intent of determining if a node exists or not.

Reason: Avoiding potential user-side accidents. Data must not be accessible if its retrieval isn't intended

2. **No Node Deletion**

For data updates, hash chaining is implemented in m Merkle Tree. This means that if new data updates an existing node, the newly created hash is **added to a look up registry**. This means that upon running Lookup() with a stale hash, a node that used to implement that hash will be returned and this node will have the **valid latest hash.**

Stale hashes **cannot be used for verifying a proof.** Proof generation also makes use of the latest, accurate information.

Reason: Proof verification is a high priority, high security task and must reflect accurate, up-to-date information.

3. **Hash chaining not implemented in Proof verification**

For data updates, hash chaining is implemented in m Merkle Tree. This means that if new data updates an existing node, the newly created hash is **added to a look up registry**. This means that upon running Lookup() with a stale hash, a node that used to implement that hash will be returned and this node will have the **valid latest hash.**

Stale hashes **cannot be used for verifying a proof.** Proof generation also makes use of the latest, accurate information.

4. **No Node Deletion**

Node deletion is not implemented as this violates the fundamental immutability that makes a Merkle tree what it is. Existing data must never be moved.

5. **My 128-bit security algorithm choice**

For this project, I made use of Golang's built-in SHA-256 function and returned a truncated output from its 32 bit output to 16 bits to explicitly obtain 128-bit security on all hashes.

Initializing the Merkle Tree

The core Merkle Tree's data structure consists of a simple root node as well as node pointer list.

```
// MerkelTree holds the root node as well as a list for easy lookup for
// searching/updating nodes.
type MerkelTree struct {
    root           *Node
    lookupNodeList map[string]*Mapping
}
```

- **root** is the root node of the tree where all paths will branch out from.
- **lookupNodeList** contains a list of all the Merkle Tree's hashes and pointers to the leaf nodes that each of these hashes represent. **Mapping** is a structure that contains the respective pointers as well as a historical list of all the hash updates made to the specific node.

```
type Mapping struct {
    node               *Node
    hashUpdateHistroy [][]byte
}
```

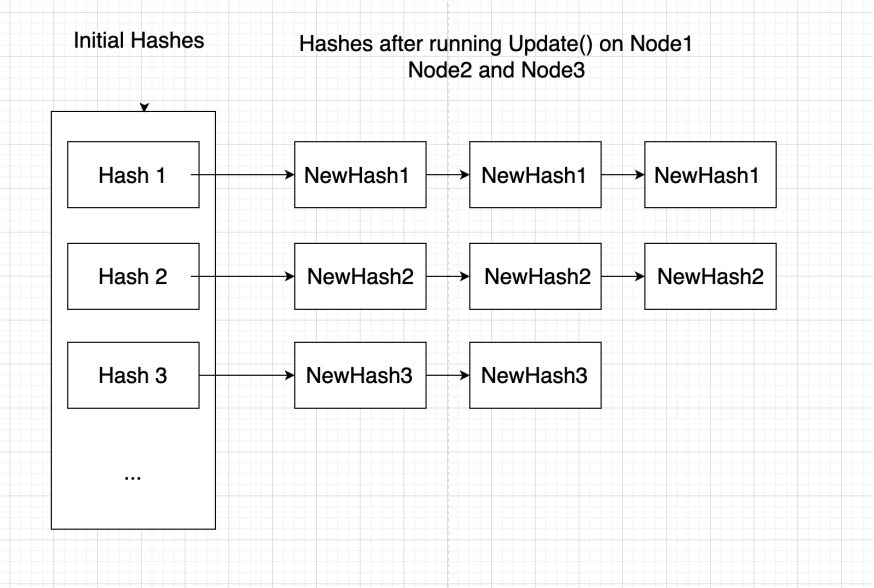
Unpacking the **Mapping** struct, we see the following;

- **node** is the node pointer pointing directly to the leaf referenced by the hash in **lookupNodeList** above.
- **hashUpdateHistory** is a historical list of all the hashes that have ever been associated with this node in its lifetime across all its updates.

Hashmap Chaining

In keeping with preserving all modification history of the Merkle Tree, hash chaining is used to ensure a historical list of all updates done to every leaf node are kept in a chained list.

From a performance perspective, initial lookup speed of an un-updated node runs at O(1) search complexity and for updated nodes, would scale to O(N) (N == number of updates done to this node).



Lastly, we have our individual Node structure.

```go
type Node struct {
    left  *Node
    right *Node
    prev  *Node
    data  []byte
    hash  []byte
}
```

Each node contains its **data,** its **hash** and is bidirectional with **left** and **right** child nodes, and a **prev** node for convenient backtracking up the tree.

And lastly, a simple InitMerkelTree() function that creates an empty Merkle Tree pointer. It is initialized as a pointer to ensure an explicit single version.

```go
func InitMerkelTree() *MerkelTree {
    return &MerkelTree{
        root:           nil,
        lookupNodeList: map[string]*Mapping{},
    }
}
```

Insert

Insert creates a new leaf node. It has differing behavior depending on whether we're creating an initial root node, an initial root branch or appending to an established tree (all conditions after the first 2).

The basic structure of the logic is as follows.

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {

    if merkelTree.root == nil {
        // We're at the first node
    } else if merkelTree.root.left == nil && merkelTree.root.right == nil {
        // We're creating our first branch node (that will have 2 children)
    } else {
        // We're appending normally to the Merkel Tree
    }

    return hash, nil
```

We have a unique condition for the **first branch node** because of the reliance on an existing **prev** node that our algorithm will be making use of that we'll talk about later on.

Our insert will return the newly generated hash upon its completion.

Let's step through each of the **if blocks** to build an understanding of the code logic.

1. **If the Merkle Root is nil**

```
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        newNode, err := CreateNode(data, hash)
        if err != nil {
            return nil, err
        }
        //...
```

The first thing we do is create a single node and assign it to the root.

CreateNode implementation

```
func CreateNode(data, hash []byte) (*Node, error) {
    if hash == nil {
        return nil, errors.New("hash data missing")
    }

    return &Node{
        data:  data,
        hash:  hash,
        left:  nil,
        right: nil,
        prev:  nil,
    }, nil
}
```

CreateNode has a hard check against **the hash being empty.** In a Merkle Tree, branch nodes don't hold any data and only their hash, so an empty **data** value is acceptable.

Once we've finished creating our root node, we add its hash to the **lookupNodeList** in the **MerkelTree** struct.

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
        err = merkelTree.newHash(newNode, hash)
        if err != nil {
            return nil, err
        }
        // ...
```

We run newHash() which will attempt to create the hash for us if it doesn't exist.
IMPORTANT: In this condition, an error here is impossible as the node has no data but good defensive programming is never a bad thing.

```go
func (merkelTree *MerkelTree) newHash(node *Node, hash []byte) error {
    if merkelTree.findHash(hash) {
        return errors.New("Hash already exists. Use Update() to update an existing hash")
    }

    merkelTree.lookupNodeList[string(hash)] = &Mapping{
        node:              node,
        hashUpdateHistroy: [][]byte{},
    }

    return nil
}
```

findHash works by interacting with the hash chain to determine if this hash already exists in both the base map as well as each of the update history lists.

And finally, we assign the root to our newly created node.

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
        merkelTree.root = newNode
    } else if //...
```

The full block of code;

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    var newNode *Node
    // If the root is nil, make a new node
    if merkelTree.root == nil {
        newNode, err := CreateNode(data, hash)
        if err != nil {
            return nil, err
        }
        err = merkelTree.newHash(newNode, hash)
        if err != nil {
            return nil, err
        }
        merkelTree.root = newNode
        // If we're at the first node, initialize it's children
    } else if //...
```

2. **If we've created our root but only have a root**

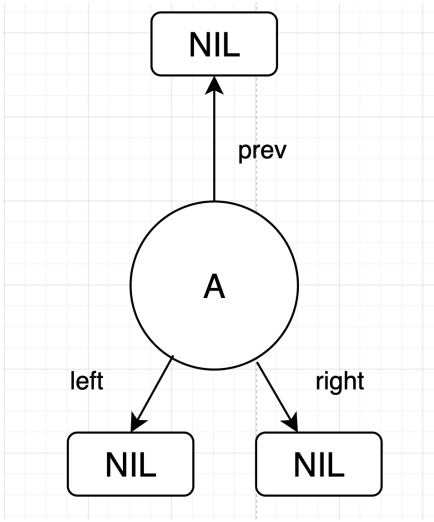
This logic entails taking our root node, making a new node and making both the root and the new node both child nodes of a new **root branch.**

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
    } else if merkelTree.root.left == nil && merkelTree.root.right == nil
        // We're creating our first branch node (that will have 2 childre
        newNode, err := CreateRootBranch(&merkelTree.root, data, hash)
        if err != nil {
            return nil, err
        }
```

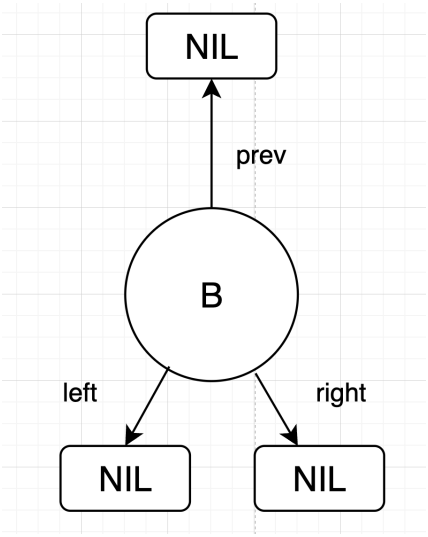
The logic takes place in CreateRootBranch

Before looking at the code, this code might be complicated enough to justify a diagram.

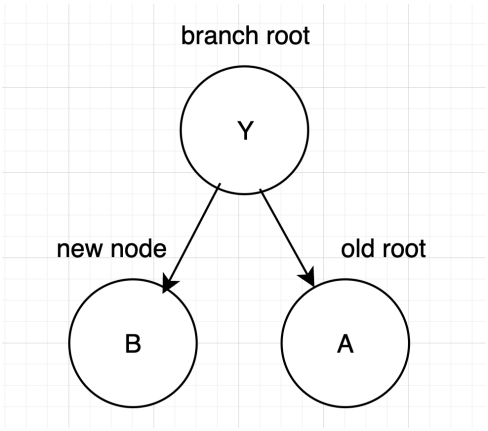
Consider this as our root node;



All 3 of its pointers point to **nil**. Now consider the new value we're adding, **B** (same picture, but with a **B).**



We need a 3rd node that will serve as an **empty root node** that will server as a parent to both **A and B.**

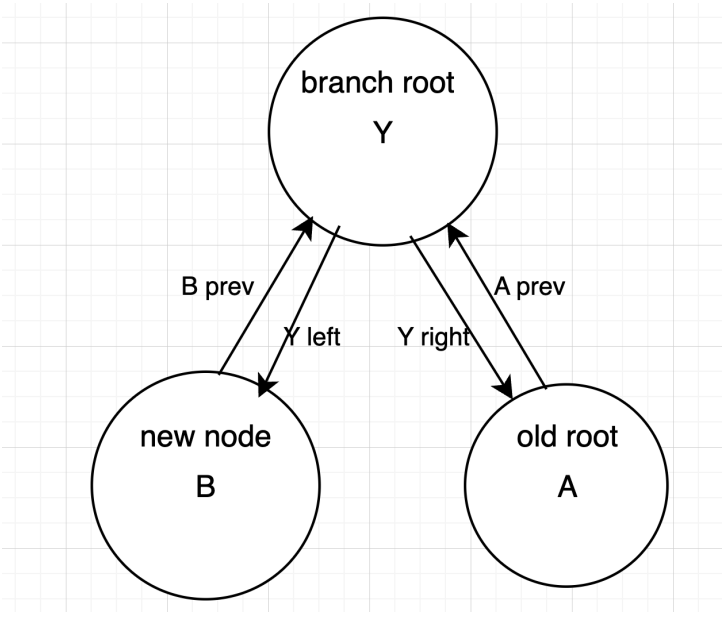


The goal.

The process is as follows;

1. We'll start with creating a new node; **Y. This will serve as our new branch**
2. We'll create a new node **B. This has our actual data.**
3. We'll first **point Y's right branch at the root, A**
4. We'll then **point Y's left branch at B.**
5. Then, we point **both A and B's prev branches at Y.**

We want our nodes to look like this



In the code it translates to this;

```go
func CreateRootBranch(root **Node, data, hash []byte) (*Node, error) {
    currentNode := *root
    // 1. We'll start with creating a new node; Y. This will serve as our new branch
    branchNode, err := CreateNode([]byte("Y"), []byte(currentNode.hash))
    if err != nil {
        return nil, err
    }
    // 3. We'll first point Y's right branch at the root, B // I mixed up the directions
    // soz
    branchNode.left = currentNode

    // 2. We'll create a new node B. This has our actual data.
    right, err := CreateNode(data, hash)
    if err != nil {
        return nil, err
    }
    // 4. We'll then point Y's left branch at B.
    branchNode.right = right
    currentNode = branchNode

    // Then, we point both A and B's prev branches at Y.
    branchNode.right.prev = branchNode
    branchNode.left.prev = branchNode

    right.prev = branchNode
    *root = branchNode
```

NOTE: I messed up the mapping of directions in my explanation and realized too late that i did this. Apologies.

We then loop back to the root branch and proceed to create a new hash from it made up of the hashes of it's 2 child nodes.

And finally add the new hash to the **LookupNodeList** map.

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
    } else if merkelTree.root.left == nil && merkelTree.root.right == nil {
        // ...
        traverse := newNode.prev
        for traverse != nil {
            traverse.hash = GenerateHash(traverse.left.hash, traverse.right.hash)
            traverse = traverse.prev
        }
        merkelTree.newHash(newNode, hash)
    } else {
        // ...
    }
```

3. Else insert a regular node

The logic for this block works for when the tree has 2 or more leaves and can be broken into 2 key steps;

a. Determine which branch we want to append to.
b. Navigate to that branch and repeat the node insertion process.

The key difference between inserting when we have multiple leaves as opposed to inserting on the root node is the existence of a **prev** node that doesn't point to **nil.** This **non-nil** requirement is what made the function InsertNode insufficient for the root branch.

Determining a branch

Before inserting a node, a target branch must be selected. The logic prioritizes the most shallowest branches at any given time. To do this, we make use of a method in the Merkle Tree called **navigateTree**.

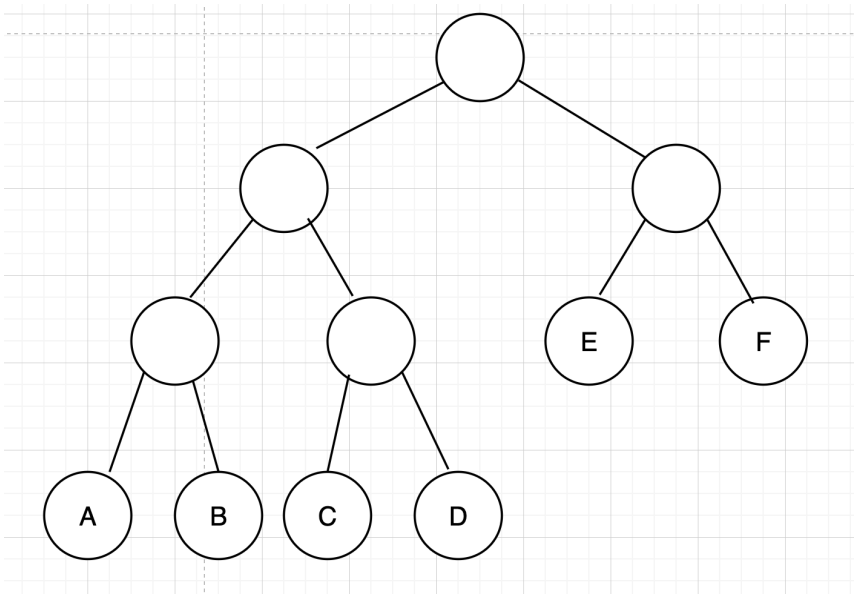
```
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
    } else if merkelTree.root.left == nil && merkelTree.root.right == nil {
        // We're creating our first branch node (that will have 2 children)
    } else {

        // Get all the deepest nodes and get the first shallow node you find.
        deepestNodes := merkelTree.navigateTree()
        // ...
    }
```

This function returns a list of NodeDepth structs;

```
type NodeDepth struct {
    depth int
    node  *Node
}
```

A single NodeDepth contains the **depth** of a leaf node as well as a pointer to that **node.**



On the above tree, **navigateTree** would return;
[A, 3], [B, 3], [C, 3], [D, 3], [E,2], [F,2].
And from this list, E and F would be prioritized as they are the most shallow nodes currently as we need to prioritize tree balancing as often as possible.

```go
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
    } else if merkelTree.root.left == nil && merkelTree.root.right == nil {
        // ...
    } else {
        // ...

        targetNode := deepestNodes[0]
        for _, node := range deepestNodes {
            if node.depth < targetNode.depth {
                targetNode = node
            }
        }
        // ...
    }
```

Once we have our nodes, we loop through the pointer/depth pairs, looking for the first shallow node we can find in the list. Once found, we proceed with node insertion.

```
func (merkelTree *MerkelTree) Insert(data []byte) ([]byte, error) {
    // ...
    if merkelTree.root == nil {
        // ...
    } else if merkelTree.root.left == nil && merkelTree.root.right == nil {
        // ...
    } else {
        // ...

        // Insert at this shallow node.
        newNode = InsertNode(targetNode.node, &targetNode.node.prev, data, hash)
        traverse := newNode.prev
        for traverse != nil {
            traverse.hash = GenerateHash(traverse.left.hash, traverse.right.hash)
            traverse = traverse.prev
        }
        merkelTree.newHash(newNode, hash)
        // ...
    }
```

The insertion code from hereon is follows the same logic as the previous **Else/If** block;

- Add the new node to the leaf
- Backtrack and rehash all parent nodes with their new children hashes all the way up to the root.

NOTE: **Where ease of readability isn't a concern, I have opted to reuse code like in the above snippet. Because of the permanent nature of Merkle Trees, for readability, I avoid pointer arithmetic wherever possible.**

Update

Update takes in brand new data and the hash of an existing node and replaces the data at the node that the hash represents with the new data, and returns the new hash value for the newly updated node.

This new hash is added to the **LookupNodeList.**

Update's logic works like this

1. Look up the hash to update to see if it exists.
2. If it exists, replace its data and create a new hash for it.
3. Loop back up the tree from it to the root, regenerating new.
4. Add the new hash to the LookupNodeList.

```go
func (merkelTree *MerkelTree) Update(newData, hash []byte) ([]byte, error) {
    // 1. Look up the hash to update to see if exists.
    node, err := merkelTree.Lookup(hash)
    if err != nil {
        return nil, errors.New("Hash not found")
    }

    // 2. If it exists, replace its data and create a new hash for it
    newHash := Hash128(newData)
    node.data = newData
    node.hash = newHash
    node = node.prev

    // 3. Loop back up the tree from it to the root, regenerating new
    // hashes for every parent node above it.
    for node != nil {
        node.hash = GenerateHash(node.left.hash, node.right.hash)
        node = node.prev
    }

    // 4. Add the new hash to the LookupNodeList.
    merkelTree.updateHashVersionHistory(hash, newHash)

    return newHash, nil
```

Lookup

Lookup takes in a hash for a node and returns it. The hash can be the node's current hash or one of the hashes it had before an update overwrote it.

The node returned will have the most recent hash in its hash field.

Lookup works by first assessing the Merkle Tree's **LookupNodeList**'s hashmap value with the hash as the key. If it fails to find an immediate key, it loops over each of the hash values in the map, iterating over their **hashUpdateHistroy.** If a match is found, that node's pointer is returned.

```go
func (merkelTree *MerkelTree) Lookup(hash []byte) (*Node, error) {
    // 1. Try an O(1) hashmap lookup to find hte node.
    block, ok := merkelTree.lookupNodeList[string(hash)]

    if !ok {
        // 2. If unsuccessful, loop over all hashes and their historical changes
        for oldHash, updateHistory := range merkelTree.lookupNodeList {

            for _, updatedHash := range updateHistory.hashUpdateHistroy {
                if compareHash(hash, updatedHash) {

                    block, ok := merkelTree.lookupNodeList[oldHash]
                    if ok {
                        return block.node, nil
                    }
                }
            }
        }

        return nil, errors.New(fmt.Sprintf("hash (%v) not found", hash))
    }

    return block.node, nil
}
```

GenerateProof

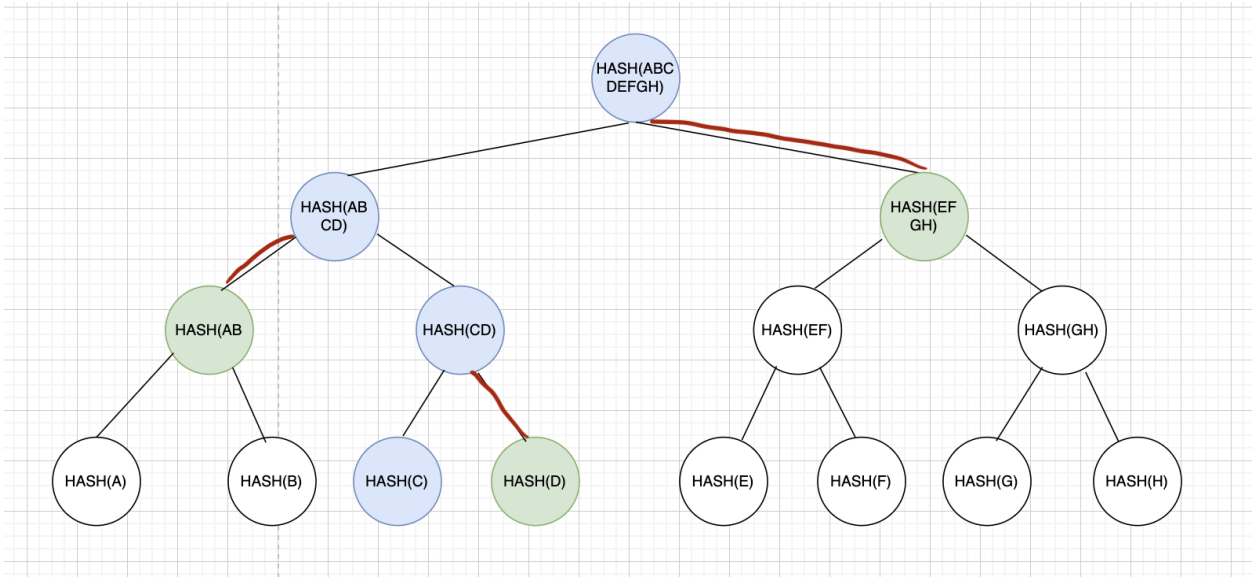
GenerateProof implements the algorithm mentioned in the Merkle Tree Algorithm Overview section above.

GenerateProof takes in the hash of the leaf we're interested in generating a proof for and returns a MerkelProof struct. This struct will be used in the proving process.

```go
type MerkelProof struct {
    LeafHash   []byte
    ProofList  [][]byte
    Directions []bool
}
```

- **LeafHash** is the original leaf's hash we want to check against
- **ProofList** is going to hold the list of hashes we gather on our climb up the Merkle Tree
- **Directions** holds the left/right direction information that will be used to eventually determine (during verification) how to navigate from root down to the leaf, and also how to rebuild root's hash from the hashes in the **ProofList** (see Merkle Tree Algorithm Overview section for details on this process)

(copy from Merkle Tree Algorithm Overview):



Green nodes are the hashes we'll be collecting
Blue nodes are where "we" (the visitor) are at each step.
Red lines lines are our directions; 1 = LEFT TURN, 0 = RIGHT TURN
Our list starts with [C], our turn list is at [1]
Starting at Hash(C), we move up to Hash(CD)
We grab Hash(D)
Hash(D) is at a RIGHT TURN, we add 0 to our turn list
Our list contains [C, D], our turn list contains [1, 0]
From Hash(CD) we move up to Hash(ABCD)
We grab Hash(AB)
Hash(AB) is at a LEFT TURN, we add 1 to our turn list
Our list contains [C,D, AB], [1,0,1]
From Hash(ABCD), we move up to Hash(ABCDEFGH)
We grab Hash(EFGH)
Has(EFGH) is at a RIGHT TURN, we add 0 to our turn list,
Our list now contains [C,D, AB, EFGH], and our turns are [1,0,1, 0]

The main **for loop** follows this logic.

```go
func (merkelTree *MerkelTree) GenerateProof(leafHash []byte) (*MerkelProof, error) {
    // ...
    proofChain := [][]byte{leafHash}
    pathway := []bool{true}
    current := node
    previous := node.prev
    for previous != nil {
        if previous.left == current {
            proofChain = append(proofChain, previous.right.hash)
            pathway = append(pathway, true)
        } else {

            proofChain = append(proofChain, previous.left.hash)
            pathway = append(pathway, false)
        }
        current = previous
        previous = previous.prev
    }
    // ...
```

The **previous and current** values follows the path of our **blue nodes** in the diagram above and the **proofChain** is the list of hashes being built up by following the **green nodes**.

The detail not mentioned in the above diagram is the **pathway** values that are being saved. We will see in the verification process.

VerifyProof

IMPORTANT: VerifyProof is kept decoupled from MerkelTree to make it explicit to the user that verification does not benefit from Merkle Tree's convenient functionality of lookups and hash chain lookup support. The hash passed into VerifyProof must be the most up to date information available. GenerateProof can generate a proof with an out-of-date hash but the contents of GenerateProof will **always be the most current information**

Proof verification works like GenerateProof but in reverse-sort of. The algorithm is as follows. In my algorithm, it is assumed that the leaf node's value is the "first" node in the list and that it's direction is assumed to be "true" (imagine that we came from one of its nil children and we came from the right/1)

(copy from Merkle Tree Algorithm Overview):

Taking the same tree from GenerateProof, this what our tree would look like.
C is not considered a "step" and is included here for simplicity of algorithm implementation in code.

So Node Hashes: C [D, AB, EFGH]
Directions 1 [0, 1, 0]

1 CD, 0 ABCD, 1 ABCDEFGH

Using the lists above we have the following instructions for reaching the leaf node for C
Step 1: 0 = LEFT TURN, RIGHT JOIN [CD] - we join D to C from the right, making it CD
Step 2: 1= RIGHT TURN, LEFT JOIN [AB, CD] - we join AB from the left of CD, making it ABCD
Step 3: 0 = LEFT TURN, RIGHT JOIN [ABCD, EFGH] - we join EFGH from the right, making it ABCDEFGH;

The Hashing Process

Hash128

The hashing process is kept to a simple SHA-256 and makes use of Golang's native **crypto/sha256** library.

I implemented a wrapper function, Hash128, which calls sha256.Sum256(data) and returns the first 16 bytes of the result. This truncation is used to obtain a 128-bit hash from the 256-bit SHA-256 output.

```
func Hash128(data []byte) []byte {
    hash := sha256.Sum256(data)
    return hash[:16]
}
```

The GenerateHash function initially implemented a double-hash function; Basically

Hash128(Hash128(data1) + Hash128(data1))

But this turned out to be too unpredictable, routinely outputting different values even between the same reruns of the same data. I suspect Golang's Sum256 might be behaving unexpectedly.

Due to time constraints, I opted to implement a simple concatenation of Hash128 outputs as we navigate up the tree;

GenerateHash:

```
func GenerateHash(LeftChildHash, RightChidlHash []byte) []byte {
    joinHashes := string(LeftChildHash) + string(RightChidlHash)
    return []byte(joinHashes)
}
```

Lastly, an important one;

compareHash

The naming is a bit deceptive as I use this to compare all byte arrays. Golang has a convenient method of converting byte arrays to strings and then comparing them against one another;

```
byteArray1 := []byte(“first”)
byteArray2 := []byte(“second”)
string(byteArray1) == string(byteArray2)
```

But because we’re first hashing our strings and the data within our hashes might not be entirely ASCII, I don’t have confidence in the accuracy of string conversion outside of debug/unittest error strings for human readability. Comparing each individual byte is more explicit. I can look at what its doing and be confident that the comparison is accurate.

And so I implemented a sort of byteCompare() (the actual name that compareHash should have been). This naming was chosen when it only applied to comparing hash outputs and I didn’t foresee its potential universality in advance.

```go
func compareHash(hash1, hash2 []byte) bool {
    for index, data1 := range hash1 {
        if data1 != hash2[index] {
            return false
        }
    }

    return true
}
```

Conclusion

Overall, this was a really fun project. I hope you found this project and technical explanations helpful in parts that might be confusing in the implemented code.

Regards,
Alyson Ng