Mujtaba Asim

BSCS22012

Q1: Merkle Trees

Introduction

This Go Merkle Tree library provides a reasonably robust implementation for creating and verifying Merkle Trees, leveraging SHA-256 hashing to ensure data integrity.

The user-facing API currently only provides functionality to construct a Merkle Tree from a provided file of arbitrary size, and to verify a content block's existence in the Merkle Tree produced in the prior step, by providing a file, the index, and length of the chunk to verify.

Further improvements involve allowing the user to define their own hash function to be used during the construction of the tree; accomplishing this would be trivial by virtue of using function pointers.

Another possible optimisation, sorting, would allow us to return `false` from the **Verify** API sooner, but has been deliberately overlooked as sorting an arbitrarily-sized file is not viable in all scenarios, and is a non-complex addition anyway; we merely need a copy of the input file where the chunks are sorted instead of being in their original positions. On known, or upper-bound file sizes, e.g the number of transactions in a bitcoin block, this is accomplishable, but would be antithetical to the goal of this project; i.e providing the means to construct Merkle Tree and verify the existence of a data block from an arbitrary-sized file.

The largest reason this package is fairly efficient on large file sizes, as we will see later, is the fact that each chunk is 'streamed' into the hash object, and only when a chunk has been 'streamed' in its entirety is the hashsum calculated, so we are able to generate the hashsum as if the entire abitrarily-sized chunk were loaded into memory (which, as established earlier, is unviable), while actually loading only a fraction of said chunk.

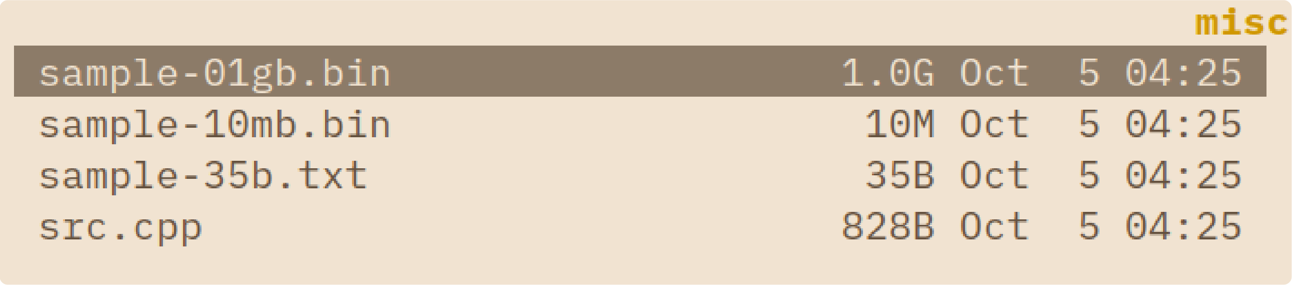
All sample screenshots in this questions are use 16 chunks of the input file; this value can trivially be tweaked however.

Construction & Printing hashes

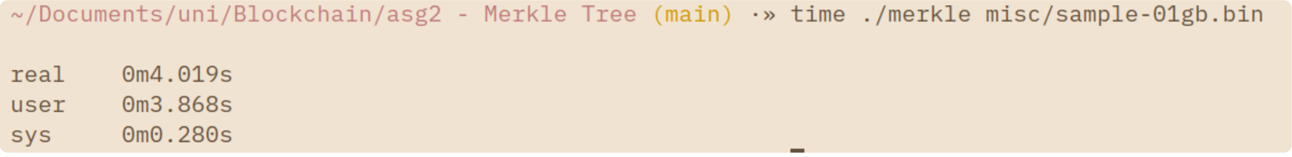
```
~/Documents/uni/Blockchain/asg2 - Merkle Tree (main) ·» ./merkle misc/sample-35b.txt

Level 0

fb8e20fc2e4c3f248c60c39bd652f3c1347298bb977b8b4d5903b85055620603
21e721c35a5823fdb452fa2f9f0a612c74fb952e06927489c6b27a43b817bed4
4ca669ac3713d1f4aea07dae8dcc0d1c9867d27ea82a3ba4e6158a42206f959b
fb2b7fce0940161406a6aa3e4d8b4aa6104014774ffa665743f8d9704f0eb0ec
c9df9c3f2963b19b9b95f58c4d33b053fa9f8586dd6ee04126e52a868f882108
d3f3fa6892497db10a2417fce9b553464cc5d07718419de8b67e73e460c7daab
ea43de53dc947fdf3cedaa4abc519f7889d5cd61f66a5ae764eb30d32c6186f9
037aeaeaf4bbf26ddabe7256a8294dc52da48d575a1247b5c2598c47de7aebab
d847acf7bab1b6f761779f3995c693e25eb899dceea61ef9043532d1ae9923a6
56af4bde70a47ae7d0f1ebb30e45ed336165d5c9ec00ba9a92311e33a4256d74
e6184ce10e266134fdcfa401e8f1a95005bcd4f18d16b62b757323e2833fe9a9
2eb0f31d74a9c43844d3dfd59208eabe5dd5d59cc84c565485f11af676f0e740
68d617d6d2ee5715af77d9795566cfaba9a43a33d14cbbc95831c507c935bad1
6b51d431df5d7f141cbececcf79edf3dd861c3b4069f0b11661a3eefacbba918
86e50149658661312a9e0b35558d84f6c6d3da797f552a9657fe0558ca40cdef
f76043a74ec33b6aefbb289050faf7aa8d482095477397e3e63345125d49f527

Level 1

12a40550c10c6339bf6f271445270e49b844d6c9e8abc36b9b642be532befe94
568a301ab7df10a2aa916d2edc73ff7660409b8223d72b8e6b3259ea551b3326
fa0189dec37b7fe60c3bfc852e7f86a429d95c764a80823fb80dd9bde2d54477
d906447fa4ee3a094fef874f35782eb146b960431c62bface0af13c9593c6ba4
42220d263adabd0cb07fb08a2857e7d07a1b64dcabd58c815458abafe674fa64
53ef16eb7f4dc4187cd91d81bb96f51ece81a7d13b94e56a7a387b2fbcaa993d
a27017b353a55a9d81778d0dce6de085f5768e39efc5e295b10ece4cf2b2811d
bed2507f4b3957554e69881006a2ef105cc8008713efc449e9f1f72b5f4a7603

Level 2

5936153a49ad13a01c17bb6faeb9ec02ff50ea8bda38978e8a19486c3a0fab7f
c8dcc7b350de1612febd951b96596648df0ddbd0a1c00fdd92f7b8b32c99b812
9240f292fe024e24936d76c30f1ba48ecac23f9ad8b58209c834e45463453e6d
1e7e95b6df5ef3271ba35e4c995abe84f05f7c6ada4e8c5320bd2ae0e9fbe1da

Level 3

48e6203fafe3e060e608f81f4c7cee35a4cae1281c8a1164997b278e0e20479d
9276199e6cb4a667e14b332b905b4bb379f9e5ab0faa918768200afa5491c6a9

Level 4

857a51b0311986c84ed67794c70fa4509c0e744aa69cda1774514d02dbbad7cb
```

Performance on a 1 GB file

On a large 1 GB file, the binary generates a Merkle Tree in about 4 seconds on a quad-core **i5-4278U** processor.



```
                                                          misc
 sample-01gb.bin                             1.0G Oct  5 04:25
 sample-10mb.bin                              10M Oct  5 04:25
 sample-35b.txt                               35B Oct  5 04:25
 src.cpp                                     828B Oct  5 04:25
```

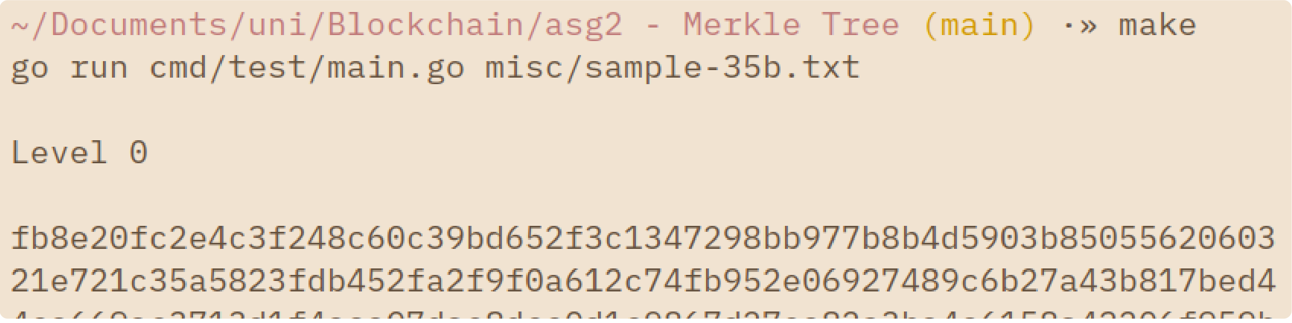


```
~/Documents/uni/Blockchain/asg2 - Merkle Tree (main) ·» time ./merkle misc/sample-01gb.bin

real    0m4.019s
user    0m3.868s
sys     0m0.280s
```

Proving Membership

To prove membership in a straightforward manner, we will run the program on a simple, 35 byte file containing the following: `abcdefghijklmnopqrstuvwxyz123456789`.

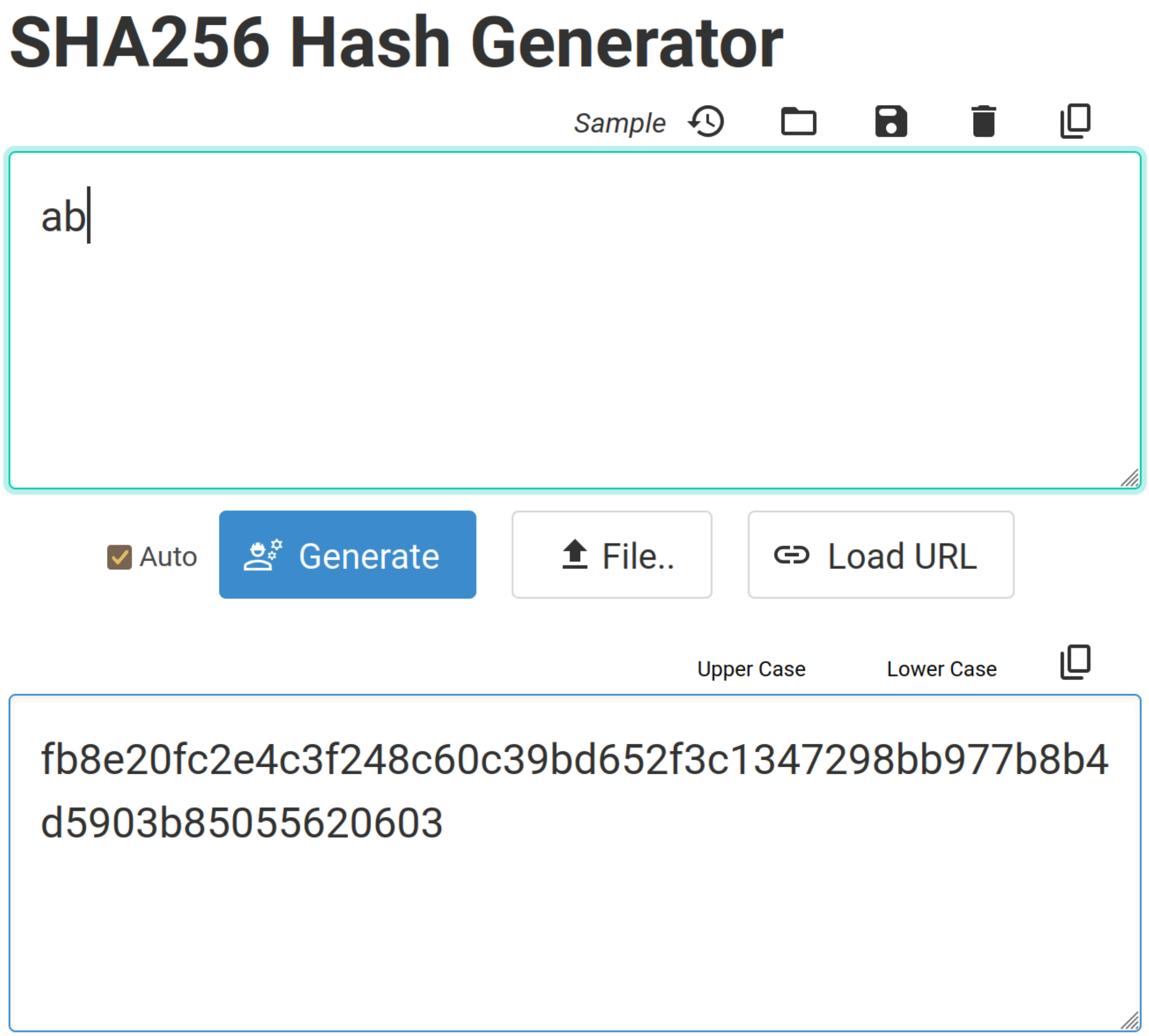
With a chunk count of 16, we can be certain that the first 15 chunks will contain 2 elements (corresponding to 2 bytes) each from the input file and the final chunk will hold the excess.



```
~/Documents/uni/Blockchain/asg2 - Merkle Tree (main) ·» make
go run cmd/test/main.go misc/sample-35b.txt

Level 0

fb8e20fc2e4c3f248c60c39bd652f3c1347298bb977b8b4d5903b85055620603
21e721c35a5823fdb452fa2f9f0a612c74fb952e06927489c6b27a43b817bed4
```

For the first 4 elements, we see the hashes `fb8e...` and `21e7...`, corresponding to the chunks "ab" and "cd" from the file respectively.

Using an online generator to verify the above's validity, we can proceed forward.



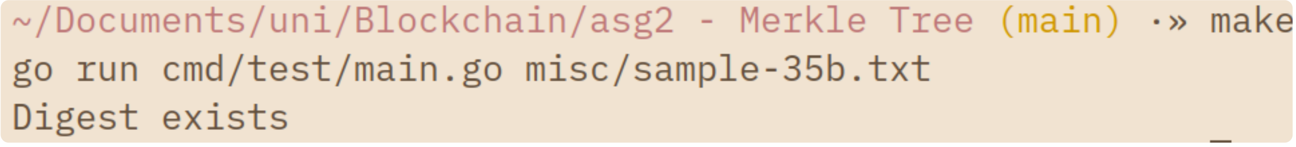
SHA256 Hash Generator

ab

fb8e20fc2e4c3f248c60c39bd652f3c1347298bb977b8b4d5903b85055620603

Now, to verify whether an arbitrary chunk exists in the Merkle Tree, we call the `Verify` function, and pass it the bytes corresponding to the region of the file that holds characters we are certain comprise the first chunk: "ab"

```
exists := merkletree.Verify(mt, f, 0, 2)
// mt = Merkle Tree object
// f = file object
// 0 = index to start reading
// 2 = index to read till

if exists {
        fmt.Println("Digest exists")
} else {
        fmt.Println("Digest does not exist")
}
```

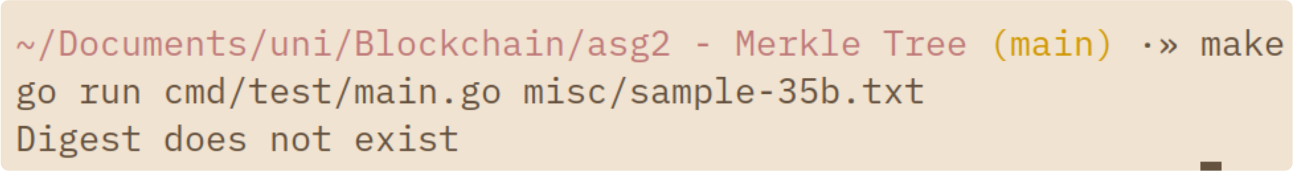


```
~/Documents/uni/Blockchain/asg2 - Merkle Tree (main) ·» make
go run cmd/test/main.go misc/sample-35b.txt
Digest exists
```

As expected, we see that the digest exists. Similarly, to quickly prove non-membership, I changed the function call above to:

```
exists := merkletree.Verify(mt, f, 1, 3)
```

corresponding to the content ranging from byte #1 till byte #3 (which we know would be "bc" - i.e NOT a valid chunk), and we see as expected:



```
~/Documents/uni/Blockchain/asg2 - Merkle Tree (main) ·» make
go run cmd/test/main.go misc/sample-35b.txt
Digest does not exist
```

Under the hood, the `Verify` subroutine iterates through the Merkle Tree from the leaf node corresponding to the specified chunk up to the root node. At each level, it verifies that the hash of the current node is the result of combining the hashes of its child nodes. If all verifications pass, it concludes that the data integrity of the specified chunk is preserved and returns true.

Q2: Verkle Trees vs Merkle Trees

Introduction

A *Verkle Tree* is a more recent data structure introduced in 2018, as a '*bandwidth-efficient alternative to Merkle Trees*' [1] , but follows a different approach; namely, the substitution of cryptographic hash functions with vector commitments.

By virtue of its recency, it's only seen use in some blockchain systems and research applications at the moment, whereas Merkle trees are ubiquitous in blockchain systems.

Merkle Trees are functionally very trivial to implement. This simplicity comes at a cost of decreased performance for large datasets however, because of the sheer number of hash operations required, dominating bandwidth consumption, and disk space in the process.

Verkle Trees on the other hard are much more complex to implement, and require a deep understanding of concepts like vector commitments, but subsequently provide stronger guarantees of data integrity, scalability to larger datasets, and increased efficiency.

Diving Deeper

In a Merkle Tree, a parent node is the hash of its children, whereas in a Verkle Tree, a parent node is the *vector commitment* of
its children.

A Merkle Tree with n leaves has $\log_2 n$-sized proofs, as opposed to 'Vector Commitment Schemes', that have constant-time proofs. One downside of the latter however, is the fact that construction takes n^2 time - a cost that is unacceptable for many applications.

A compromise comes in the form of Verkle Trees, that provide a middle ground between computational power and bandwidth, and can be understood as a *portmanteau of "Vector commitment" and "Merkle Trees."* [2]

This aforementioned compromise is achieved by tweaking the value of the branching factor, k, that allows us to achieve $O(kn)$ construction time and $O \log_k n$ verification time -- striking an optimum balance between Merkle Trees and full-fledged Vector Commitment Schemes.

Q3: Consensus Algorithms

PoW

Nodes on the network are required to solve a hash puzzle by finding a valid nonce that satisfies the target (this process is also known as mining).

The first node that finds the nonce in question get to add a new block of transactions to the blockchain and is awarded coins in the process.

Fairness is assured since every single node's every 'attempt' at the puzzle is equal. The only way to increase your odds of winning at the 'race' to the puzzle is by increasing computational power, thereby allowing you to make more attempts per unit time.

The protocol can adjust the difficulty of the puzzle arbitrarily, depending on how much time it takes on average to produce a new block.

PoS

'Trusted' nodes in the network are tasked with creating blocks. Preference is given based on their total net worth in the network (i.e their stake).

The intuition is that nodes possessing a lot of currency would have more to lose, and have likely been part of the network for a longer period of time and are thus unlikely to misbehave.

PoO

Nodes must prove ownership of specific digital assets before they can participate in transactions. This is achieved through cryptographic methods, where users sign transactions with their private keys, allowing others to verify ownership using the corresponding public keys.

Security and legitimacy are emphasized, ensuring that only rightful owners can transfer or manage their assets, thus maintaining the integrity of the network without relying on computational power or stake.

It has not seen widespread use yet because verifying ownership of assets is non-trivial.

PoA

A small group of pre-selected nodes validate blocks. Authority is based on identity rather than stake or computational power.

Validators are active and trustworthy individuals, often chose for their reputation or expertise, and are identified via their public keys.

Naturally, this is a less decentralised algorithm, and vulnerable to attacks should the validators (i.e central authority) collude together.

DPoS

A variation of the PoS mechanism where nodes on the network first democratically vote and select delegates/witnesses/block produces that have the right to validate blocks.

Users 'vote' by pooling their tokens into a pool and linking it to a particular delegate. The delegate with the largest pool then validates a block, receiving the corresponding transaction fees as a reward. They then distribute rewards to users who supported them based on each user's stake in their pool, hence the title of this consensus algorithm.

PoB

Nodes destroy/burn cryptocurrency to either participate in the validation process, or certify that they left a message at a given time by burning a minute amount of cryptocurrency and leaving a message with it as part of the transaction.

The total number of coins in the currency is permanently reduced.

---

1. https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf ↩

2. https://ethereum.org/en/roadmap/verkle-trees/ ↩