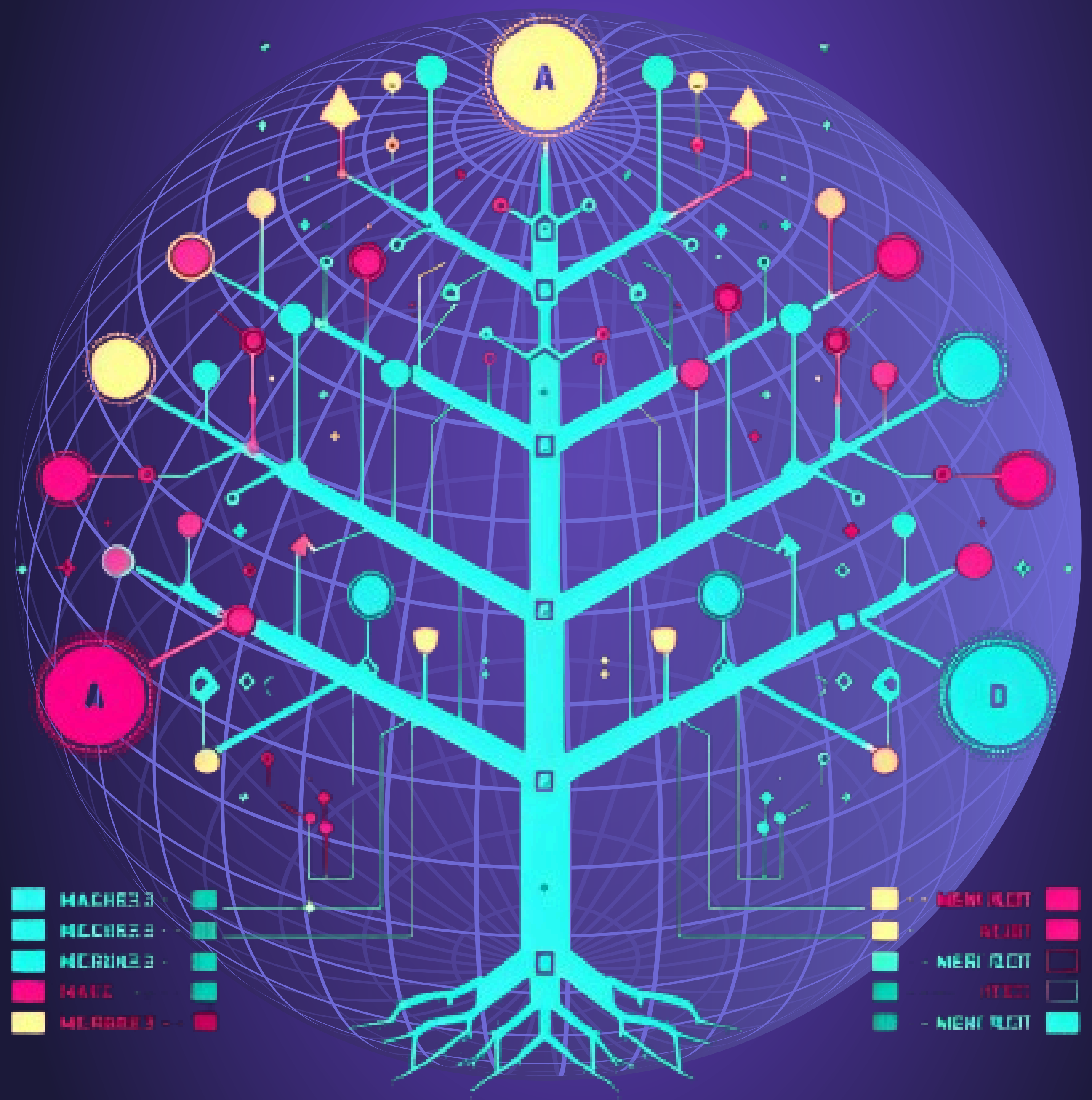
001

MERKLE TREE

DATA STRUCTURE



MERKLE TREE | DATA STRUCTURE

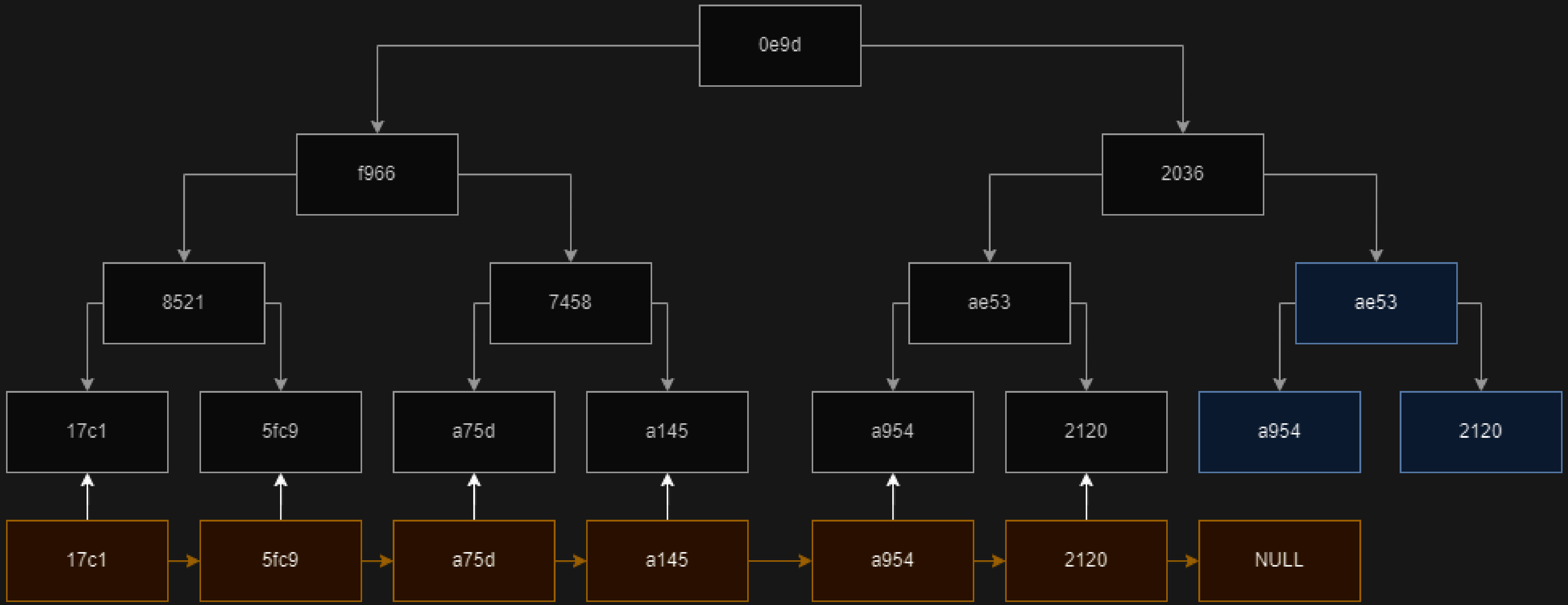
WELCOME TO CLASS!

Today's Agenda

- Introduction
- Merkle Proof
- Use cases
- Second Preimage Attack
- Complexity
- Implementation
- Conclusion

INTRODUCTION

Merkle Tree or Hash Tree



MERKLE PROOF

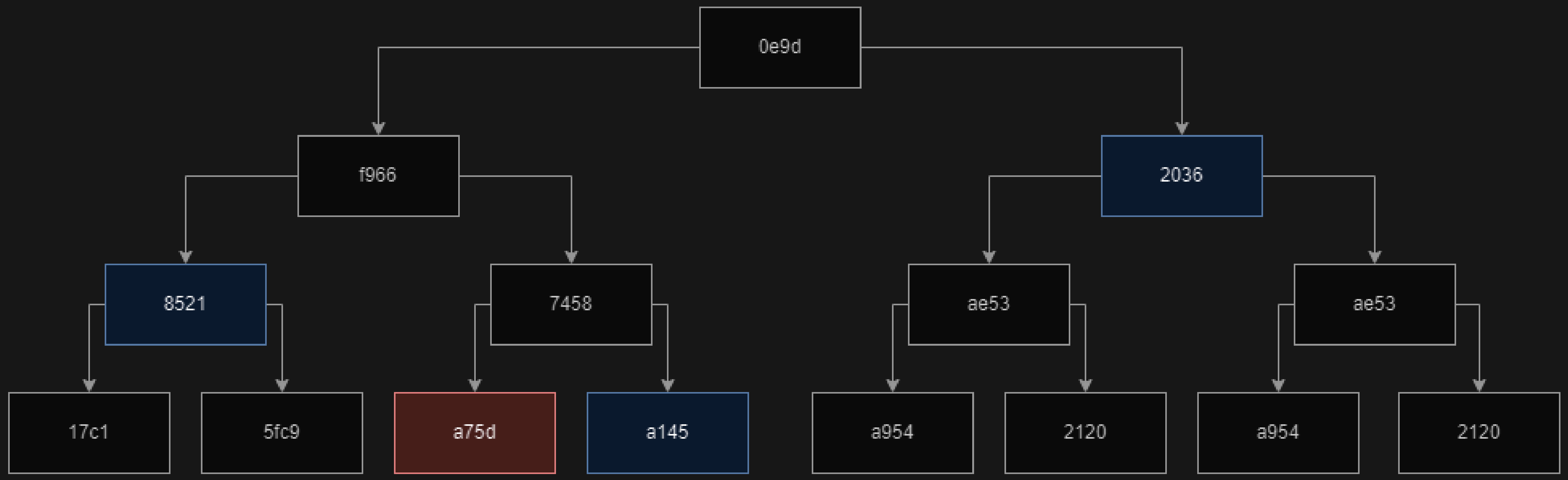
Merkle Path and minimum proofs

For merkle root [0e9d] and leaf [a75d]

Merkle Path: [a75d, a145, 8521, 2036]

Pattern on validation

We only need *log(n)* elements of the tree to check a hash.



USE CASES

How does this work?

HASH-BASED CRYPTOGRAPHY
Digital signatures schemes based on Merkle signature

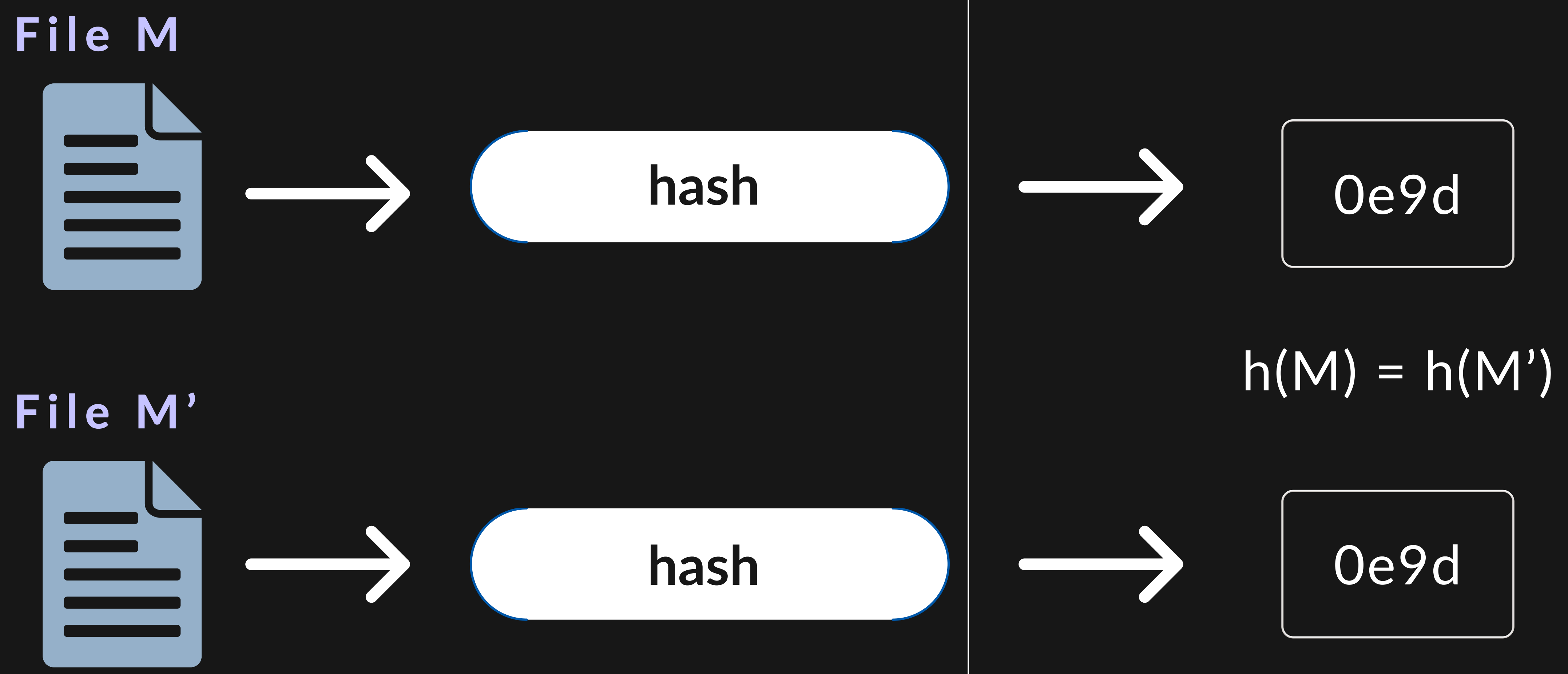
GIT AND MERCURIAL
Distributed revision control systems

BITCOIN AND ETHEREUM
Peeer-to-peer network

NOSQL SYSTEMS
Find inconsistencies in replicas

SECOND PREIMAGE ATTACK

Based on hash collisions



SECOND PREIMAGE ATTACK

How can we ensure that?

h(M): 0e9d

size(M): 15

size(M) != size(M’)

h(M’): 0e9d

size(M’): 7

COMPLEXITY

Branching factor 2 and *k*

	Average	Worst
Space	$O(n)$	$O(n)$
Search	$O(\log_2(n))$	$O(\log_k(n))$
Traversal	$^*O(n)$	$^*O(n)$
Insert	$O(\log_2(n))$	$O(\log_k(n))$
Delete	$O(\log_2(n))$	$O(\log_k(n))$
Syncronization	$O(\log_2(n))$	$O(n)$

IMPLEMENTATION

Hands On!

merkle_node.h


```
#pragma once

#include <string>
#include <cassert>

#include "sha256.h"

class MerkleNode {
public:

    MerkleNode();
    MerkleNode(const std::string hash);

    void SetChildren(MerkleNode *left, MerkleNode *right);
    MerkleNode *Left();
    MerkleNode *Right();

    void ComputeNodeHash();
    std::string NodeHash();

private:
    MerkleNode *left_, *right_;
    std::string hash_;
};
```

merkle_node.cc

```
MerkleNode::MerkleNode() {
    left_ = right_ = nullptr;
}

MerkleNode::MerkleNode(const std::string reference) {
    left_ = right_ = nullptr;
    hash_ = reference;
}

void MerkleNode::SetChildren(MerkleNode* l, MerkleNode* r){
    left_ = l;
    right_ = r;
}
```

merkle_node.cc


```
MerkleNode *MerkleNode::Left() {
    return left_;
}

MerkleNode *MerkleNode::Right() {
    return right_;
}

void MerkleNode::ComputeNodeHash() {
    std::string aux = left_->NodeHash() + right_->NodeHash();
    hash_ = sha256(aux);
}

std::string MerkleNode::NodeHash() {
    assert(hash_ != "");
    return hash_;
}
```

sha256.h

```
#pragma once

#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>

#include <openssl/evp.h>

#define OPENSSL_ENGINE NULL

 std::string sha256(const std::string reference);
```

sha256.cc

```
std::string sha256(const std::string reference) {
    EVP_MD_CTX *mdCtx = EVP_MD_CTX_new();
    unsigned char mdVal[EVP_MAX_MD_SIZE], *md;
    unsigned int mdLen, i;

    if (!EVP_DigestInit_ex(mdCtx, EVP_sha256(), OPENSSL_ENGINE)) {
        printf("Message digest initialization failed.\n");
        EVP_MD_CTX_free(mdCtx);
        exit(EXIT_FAILURE);
    }

    if (!EVP_DigestUpdate(mdCtx, reference.c_str(), reference.length())) {
        printf("Message digest update failed.\n");
        EVP_MD_CTX_free(mdCtx);
        exit(EXIT_FAILURE);
    }

    if (!EVP_DigestFinal_ex(mdCtx, mdVal, &mdLen)) {
        printf("Message digest finalization failed.\n");
        EVP_MD_CTX_free(mdCtx);
        exit(EXIT_FAILURE);
    }

    EVP_MD_CTX_free(mdCtx);

    std::stringstream ss;
    for(int i = 0; i < mdLen; ++i) {
        ss << std::hex << std::setw(2) << std::setfill('0') << (int)mdVal[i];
    }

    return ss.str();
}
```

merkle_tree.h


```cpp
#pragma once

#include <algorithm>
#include <cassert>
#include <string>
#include <vector>

#include "merkle_node.h"

/**
 * Represents all nodes at a level of Merkle Tree
*/
typedef std::vector<MerkleNode *> TreeLevel;

class MerkleTree {
public:
    void FromHashList(std::vector<std::string> leaves);

    std::vector<std::string> GenerateMerkleProof(std::string hash);

    void PrintTree();
    void PrintSubTree(MerkleNode *node, uint level);
    void PrintByLevels();

private:
    std::vector<MerkleNode *> leaves_;

    MerkleNode *merkle_root_;
    std::vector<TreeLevel> tree_levels_;

    void GenerateTreeLevels(TreeLevel level);
    int HashIndexInLevel(std::string hash, TreeLevel level);

};
```

merkle_tree.cc


```
#include "merkle_tree.h"

void MerkleTree::FromHashList(std::vector<std::string> leaves) {
    for(std::string leaf : leaves) {
        MerkleNode *node = new MerkleNode(leaf);
        leaves_.push_back(node);
    }

    GenerateTreeLevels(leaves_);
}
```

merkle_tree.cc


```
void MerkleTree::GenerateTreeLevels(TreeLevel level){
    tree_levels_.push_back(level);
    if(level.size() == 1) {
        merkle_root_ = level.front();
        std::reverse(tree_levels_.begin(), tree_levels_.end());
        return;
    }

    while(level.size() % 2 != 0) level.push_back(level.back());

    TreeLevel generated_level = std::vector<MerkleNode *>();
    for(int i = 0; i < level.size(); i += 2) {
        MerkleNode *node = new MerkleNode();
        node->SetChildren(level[i], level[i+1]);
        node->ComputeNodeHash();
        generated_level.push_back(node);
    }

    GenerateTreeLevels(generated_level);
}
```

merkle_tree.cc


```
std::vector<std::string> MerkleTree::GenerateMerkleProof(std::string hash) {
    assert(hash != "");
    assert(hash.length() != 0);

    int depth = tree_levels_.size() - 1;
    std::vector<std::string> proof({hash});

    for(int level = depth; level > 0; level--) {
        int index = HashIndexInLevel(hash, tree_levels_[level]);

        bool left_node = (index % 2 == 0);
        int pair_index =  left_node ? index + 1 : index - 1;

        MerkleNode *base, *pair;
        base = tree_levels_[level][index];
        pair = tree_levels_[level][pair_index];
        proof.push_back(pair->NodeHash());

        if(!left_node) std::swap(base, pair);

        MerkleNode *node = new MerkleNode();
        node->SetChildren(base, pair);
        node->ComputeNodeHash();
        hash = node->NodeHash();
        delete node;
    }
    return proof;
}
```

merkle_tree.cc

```
int MerkleTree::HashIndexInLevel(std::string hash, TreeLevel level) {
    MerkleNode* target = nullptr;
    for(MerkleNode* node : level) {
        if(node->NodeHash() == hash) {
            target = node;
            break;
        }
    }
    assert(target != nullptr);

    auto it = std::find(level.begin(), level.end(), target);
    assert(it != level.end());

    return std::distance(level.begin(), it);
}
```

merkle_tree.cc


```
void MerkleTree::PrintTree() {
    PrintSubTree(merkle_root_, 0);
}

void MerkleTree::PrintSubTree(MerkleNode *node, uint level) {
    if(node == nullptr) return;

    std:: string buff;
    buff = std::string(level*2, ' ');
    buff += std::to_string(level);
    buff += " - ";

    std::cout << buff << node->NodeHash() << std::endl;
    PrintSubTree(node->Left(), level + 1);
    PrintSubTree(node->Right(), level + 1);
}
```

merkle_tree.cc


```
void MerkleTree::PrintByLevels() {
    for(int i = 0 ; i < tree_levels_.size(); i++) {
        std:: string buff;
        buff = "level: ";
        buff += std::to_string(i) + "\n";
        for(MerkleNode* node: tree_levels_[i]) {
            buff += node->NodeHash() + "\n";
        }
        std::cout << buff << std::endl;
    }
}
```

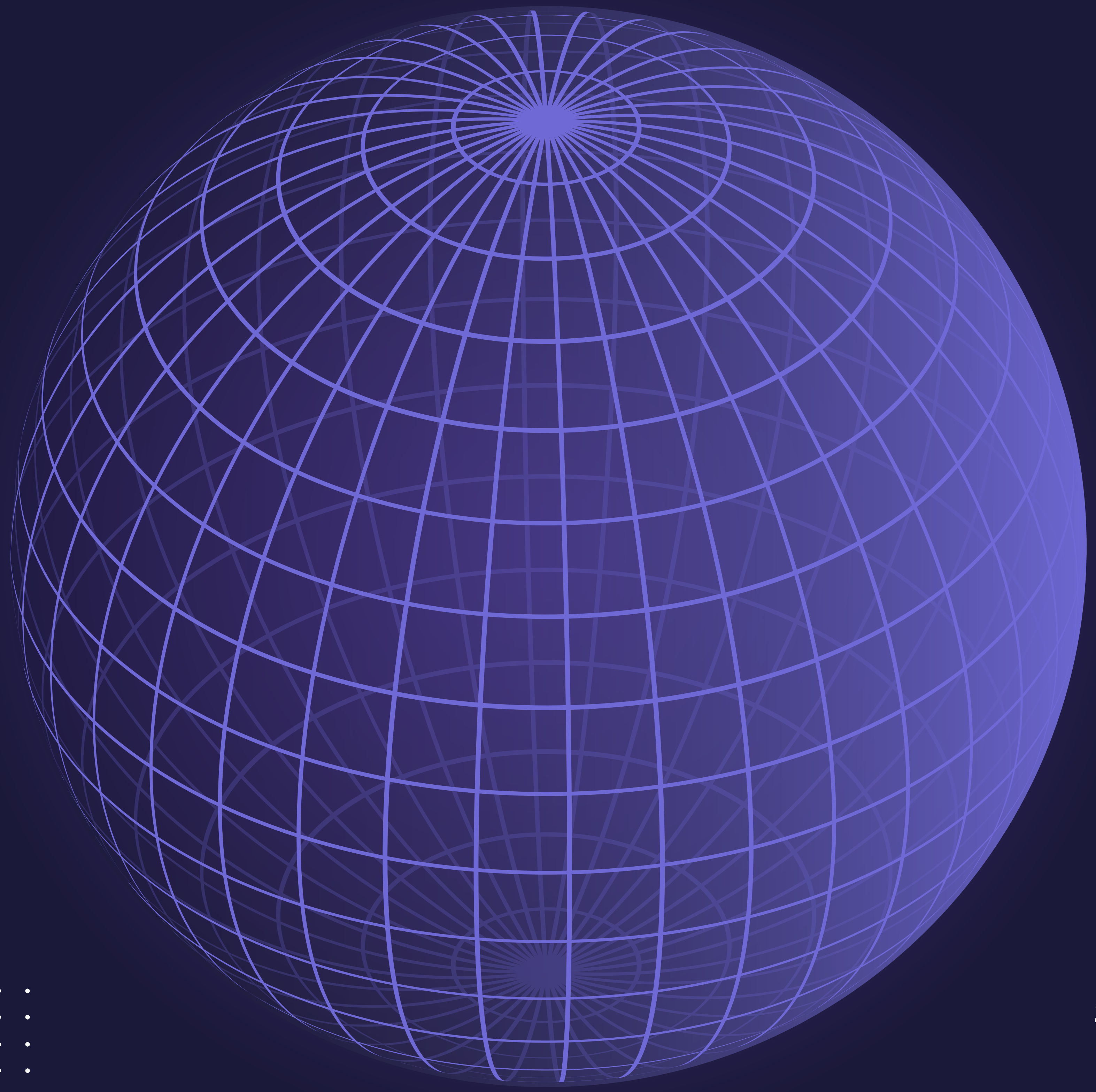
main.cc


```cpp
#include <string>
#include <vector>

#include "merkle_tree/merkle_tree.h"

using namespace std;

int main() {
    vector<string> hash_list{
        "17c1532ca6cff8f6a3a8200028af6c2580bf37f39e10cb0966e8a573e3b24a1f", //professor
        "5fc90ab335783816990ffd960cbad0afd64510a53f895b4d02b9f8b279c0ed08", //usa
        "a75d067fa44bca815126dbf606a73907c9e68f1cd892d413424edfa84a0d4058", //IA
        "a1453f380fa9f1a08e84d4703e9c168fda1fb9a36976c41a03c8af842aa04ce5", //para
        "a9548df8134a9ffbd22aea3ec97488969f455113be568351ea6a8db8bfe6b663", //jogar
        "21209597f685c8bef83dfa0b7cb389453074695a4f0ed6d6b6bca574a2d5d77f" //dados
    };

    MerkleTree tree = MerkleTree();
    tree.FromHashList(hash_list);
    tree.PrintTree();
    cout << endl;
    tree.PrintByLevels();

    vector<string> merkle_proof =
tree.GenerateMerkleProof("a75d067fa44bca815126dbf606a73907c9e68f1cd892d413424edfa84a0d4058");
    cout << merkle_proof << endl;

    exit(EXIT_SUCCESS);
}
```

CONCLUSION

BASE OF MODERN CRYPTOGRAPHY

As we can see, Merkle trees are an essential data structure in the context of contemporary peer-to-peer technologies and cryptography systems (like blockchain). They make it possible to verify huge data structures quickly and securely while maintaining data consistency and integrity and preventing recomputation.



THANK YOU!

Email: mauricioleite.fe@gmail.com
github: MauricioLeite
LinkedIn: mauricio-lefe