1. Aiken-Markle-Tree

<!-- markdown-toc start - Don't edit this section. Run M-x markdown-toc-refresh-toc -->

# Table of Contents

- [Aiken Merkle Tree](#aiken-merkle-tree)

- [Introduction](#introduction)

- [Documentation](#documentation)

- [Merkle Tree](#merkle-tree)

- [Aiken Merkle Tree](#aiken-merkle-tree-implementation)

- [Getting Started](#getting-started)

- [Prerequisites](#prerequisites)

- [Building and Developing](#building-and-developing)

- [Sample validator](#sample-validator)

- [Case study](#case-study)

- [Acknowledgments](#acknowledgments)

<!-- markdown-toc end -->

# Aiken Merkle Tree

## Introduction

The Aiken Merkle Tree project provides a Plutarch-based implementation of Merkle Trees for the Cardano blockchain. This project allows developers to leverage the security and efficiency of Merkle Trees in their Cardano smart contracts, ensuring data integrity and efficient data verification.

This project is funded by the Cardano Treasury in [Catalyst Fund 10](https://projectcatalyst.io/funds/10/f10-osde-open-source-dev-ecosystem/anastasia-labs-the-trifecta-of-data-structures-merkle-trees-tries-and-linked-lists-for-cutting-edge-contracts) and is aimed at enhancing the capabilities of Cardano smart contracts in handling complex data structures.

## Documentation

### Merkle Tree

A Merkle tree, named after its inventor Ralph Merkle, is a fundamental data structure in computer science and cryptography. It's particularly well-suited for managing and verifying large data structures, especially in distributed systems like blockchain technologies. Here's a detailed explanation:

#### Basic concept

A Merkle tree is a type of binary tree, consisting of nodes. Here's how it's structured:

- \*\*Leaf Nodes\*\*: These are the bottom-most nodes in the tree and contain hashed data. The data could be transactions (as in blockchain), files, or any data chunks.

- \*\*Non-Leaf (Intermediate) Nodes\*\*: These nodes store a cryptographic hash of the combined data of their child nodes.

- \*\*Root Node\*\*: The single node at the top of the tree contains a hash formed by its child nodes, ultimately representing the hashes of all lower levels.

#### Hash function

The core of a Merkle tree is the hash function (like SHA-256 in Bitcoin). This function takes digital data of any size and produces a fixed-size string of bytes, typically a unique digital fingerprint of the input data.

#### Construction

- \*\*Hashing the Data\*\*: First, each piece of data at the leaf level is hashed.

- \*\*Pairing and Hashing Upwards\*\*: These hashes are then paired and concatenated, and the resultant string is hashed again. This process continues until you reach the single hash at the top - the root hash.

- \*\*Tree Structure\*\*: This process creates a tree-like structure where each parent node is a hash of its children, providing a secure and efficient means of verifying the contents of the tree.

#### Features

- \*\*Efficiency in Verification\*\*: To verify any single data chunk's integrity, you don't need to download the entire tree. You only need the hashes of the nodes along the path from your data chunk to the root.

- \*\*Tamper-Proof\*\*: Any change in a leaf node (data) will result in a completely different root hash through a cascading effect of changes in the intermediate hashes. This makes it easy to detect alterations.

- \*\*Concurrency Friendly\*\*: Multiple branches of the tree can be processed simultaneously, making Merkle trees highly efficient for parallel processing.

#### Example

Consider a Merkle tree with four leaf nodes (A, B, C, D).

```

Merkle Root

|

+-----------+-----------+

| |

Hash(A+B) Hash(C+D)

| |

+---+---+ +---+---+

| | | |

Hash(A) Hash(B) Hash(C) Hash(D)

```

1. Each of A, B, C, and D is hashed: Hash(A), Hash(B), Hash(C), Hash(D).

2. The hashes of A and B are combined and hashed: Hash(Hash(A) + Hash(B)). Similarly for C and D.

3. The hash results from step 2 are combined and hashed to give the Merkle root.

Thus, the Merkle root is a digest of all the data in the leaf nodes.

In conclusion, Merkle trees offer a secure and efficient way to summarize and verify large data sets.

### Aiken Merkle Tree implementation

The Aiken Merkle Tree implementation provides several functions to create and manipulate Merkle Trees. Below is a brief overview of each function:

-`from\_list`: Constructs a Merkle Tree from a list of serialized data.

-`to\_list`: Deconstructs a Merkle Tree back into a list of elements.

-`root`: Retrieves the root hash of a Merkle Tree.

-`is\_empty`: Checks if a Merkle Tree is empty.

-`size`: Returns the number of leaf nodes in a Merkle Tree.

-`get\_proof`: Generates a proof of membership for an element in the Merkle Tree.

-`is\_member`: Verifies if an element is part of a Merkle Tree using a proof.

-`combine`: Combines two hashes into a new one.

## Getting Started

### Prerequisites

Before you begin, ensure you have [aiken](https://aiken-lang.org/installation-instructions) installed on your system.

### Building and developing

Once Aiken is installed, you should be able to seamlessly use the repository to

develop, build and run packages.

Download the Git repository:

```sh

git clone https://github.com/Anastasia-Labs/aiken-merkle-tree.git

```

Navigate to the repository directory:

```sh

cd aiken-merkle-tree

```

Execute the test suite:

```sh

aiken check

```

Build:

```sh

aiken build

```

![aiken-merkle-tree.gif](/assets/images/aiken-merkle-tree.gif)

# Sample validator

For a complete example, including tests and further explanations, reger to the provided sample validator: [sample.ak](validators/sample.ak).

# Case study

For an in-depth real-world case study on the application of Merkle Trees within the Cardano blockchain environment, particularly in the context of sidechain to main chain token transfers, refer to the following resource:

[Cardano Sidechain Toolkit - Main Chain Plutus Scripts](https://docs.cardano.org/cardano-sidechains/sidechain-toolkit/mainchain-plutus-scripts/)

This case study provides valuable insights into how Merkle Trees are integrated into blockchain transactions, offering practical examples and detailed workflows.

# Acknowledgments

This library takes the Aiken code from the aiken-lang.

The repository can be found at <https://github.com/aiken-lang/trees>

**Validator:**

**// /\*\***

**// \* This file contains the implementation of a spend\_validator function that validates a spend transaction**

**// \* using a Merkle tree. It also includes a test case for the spend\_validator function.**

**// \*/**

**use aiken/transaction.{OutputReference, ScriptContext, Spend, TransactionId}**

**use aiken\_merkle\_tree/mt.{Proof, Root, from\_list, get\_proof, is\_member, root}**

**type MyDatum {**

**merkle\_root: Root,**

**}**

**type MyRedeemer {**

**my\_proof: Proof<ByteArray>,**

**user\_data: ByteArray,**

**}**

**// /\*\***

**// \* The spend\_validator function validates a spend transaction by checking if the user data is a member of**

**// \* the Merkle tree with the given Merkle root and proof.**

**// \***

**// \* @param datum - The MyDatum object containing the Merkle root.**

**// \* @param redeemer - The MyRedeemer object containing the proof and user data.**

**// \* @param \_ctx - The ScriptContext.**

**// \*/**

**validator {**

**fn spend\_validator(datum: MyDatum, redeemer: MyRedeemer, \_ctx: ScriptContext) {**

**let MyDatum { merkle\_root } = datum**

**let MyRedeemer { my\_proof, user\_data } = redeemer**

**is\_member(merkle\_root, user\_data, my\_proof, identity)**

**}**

**}**

**// /\*\***

**// \* Test case for the spend\_validator function.**

**// \*/**

**test spend\_validator\_1() {**

**let data =**

**["dog", "cat", "mouse"]**

**let merkle\_tree = from\_list(data, identity)**

**let datum = MyDatum { merkle\_root: root(merkle\_tree) }**

**expect Some(proof) = get\_proof(merkle\_tree, "cat", identity)**

**let redeemer = MyRedeemer { my\_proof: proof, user\_data: "cat" }**

**let placeholder\_utxo =**

**OutputReference { transaction\_id: TransactionId(""), output\_index: 0 }**

**let context =**

**ScriptContext {**

**purpose: Spend(placeholder\_utxo),**

**transaction: transaction.placeholder(),**

**}**

**spend\_validator(datum, redeemer, context)**

**}**