# Merkle trees and their implementations in Starknet

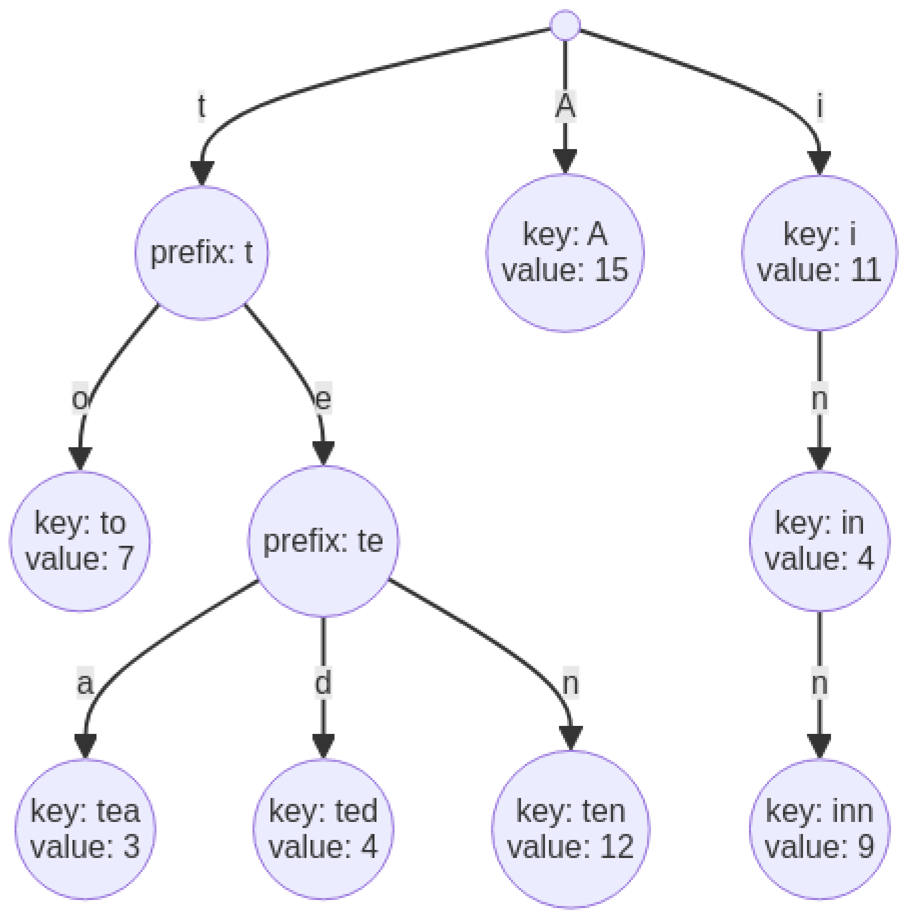
# Chapter 1. General explanation of Merkle trees

## 1.1. Genealogy

### 1.1.1. Trie

A trie, also known as a prefix tree, stands as a tree-based data structure specifically crafted for efficient storage and retrieval of string sets. Unlike conventional methods like arrays or linked lists, tries organize data hierarchically, with each node representing a single character in a string. This unique design renders tries particularly well-suited for tasks involving string matching, prefix searches, and autocomplete functionalities. Notably, strings within a trie share common prefixes, leading to space efficiency, especially when dealing with large collections of similar strings. This remarkable data structure finds applications in diverse domains, including dictionary implementations, spell checkers, and IP routing tables. This introduction serves as a preliminary exploration of the trie data structure, highlighting its unique properties and widespread utilization in efficiently addressing string-related challenges.

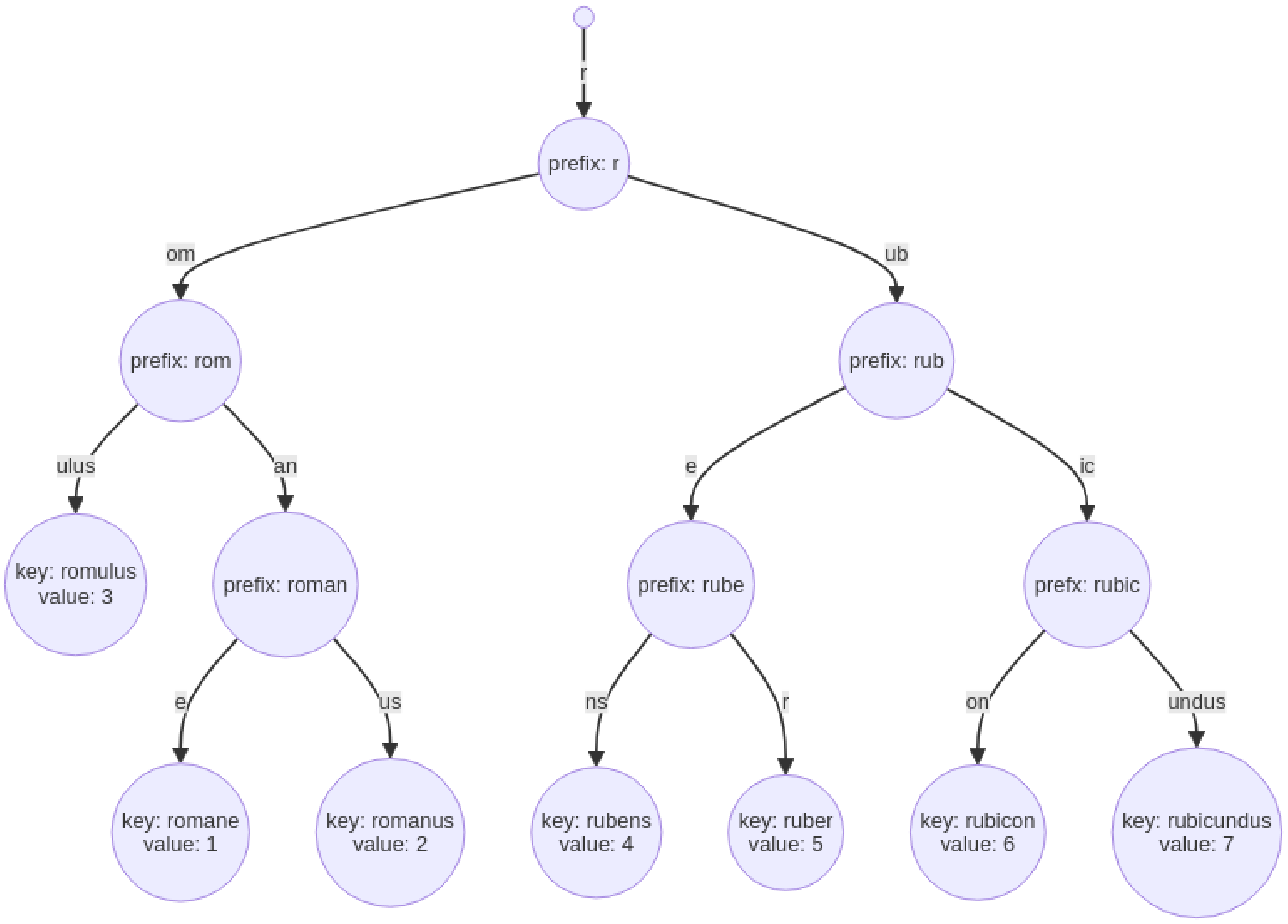
Note: - Prefix and key denote the same concept: They represent the path from the root node to the node they are attached. As such, they are not stored in the node. The position in the tree determines the value of a nodes prefix/key. - Only leaf nodes store a value



## 1.2. PATRICIA trie

Also know as Radix trie

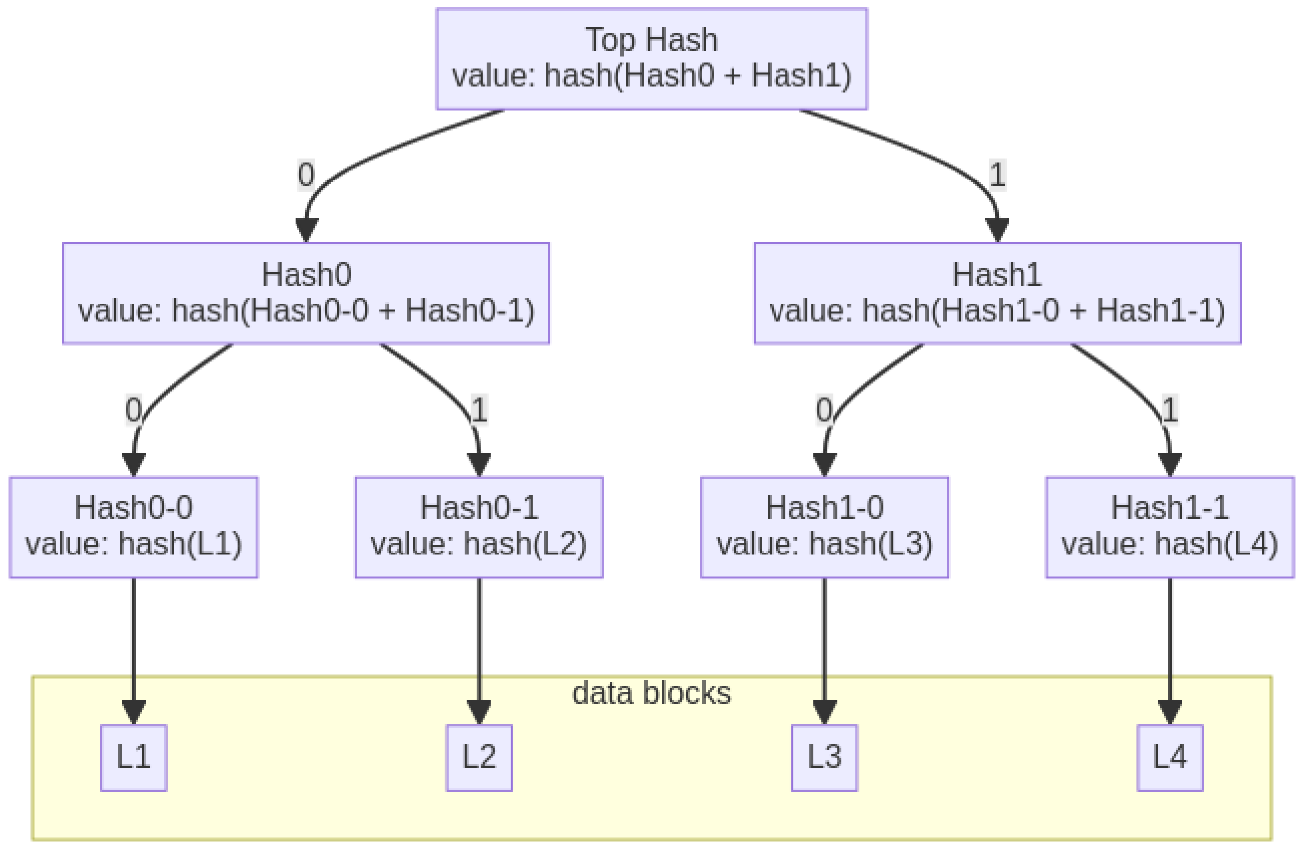
Patricia tries are a compact form of tries. This is accomplished by fusing the nodes that have only one child. As in tries, prefix/key denote the same concept. Contrary to regular tries where the position of a node implicitly represents its prefix or key, in Patricia tries, due to node fusion, "fused nodes" need to store additional information about what they represent. This representation includes details about the characters or paths that are fused together to form the node. This ensures that even though nodes are merged for compactness, the trie can still accurately represent and retrieve the original keys or prefixes associated with each node.



## 1.3. Merkle tree

A Merkle Tree is a specialized tree where every "leaf" or node is labeled with the cryptographic hash of a data block. Conversely, every non-leaf node, often referred to as a branch, inner node, or inode, carries the cryptographic hash of its child nodes' labels. This design enables efficient verification of large data sets by validating only a small portion of the tree, known as a Merkle proof. In decentralized systems, Merkle trees are used to ensure the integrity of data stored across multiple nodes without the need for a trusted central authority. The proof size is logarithmic in the number of elements in the tree.

Following is a visual representation of a Merkle tree:



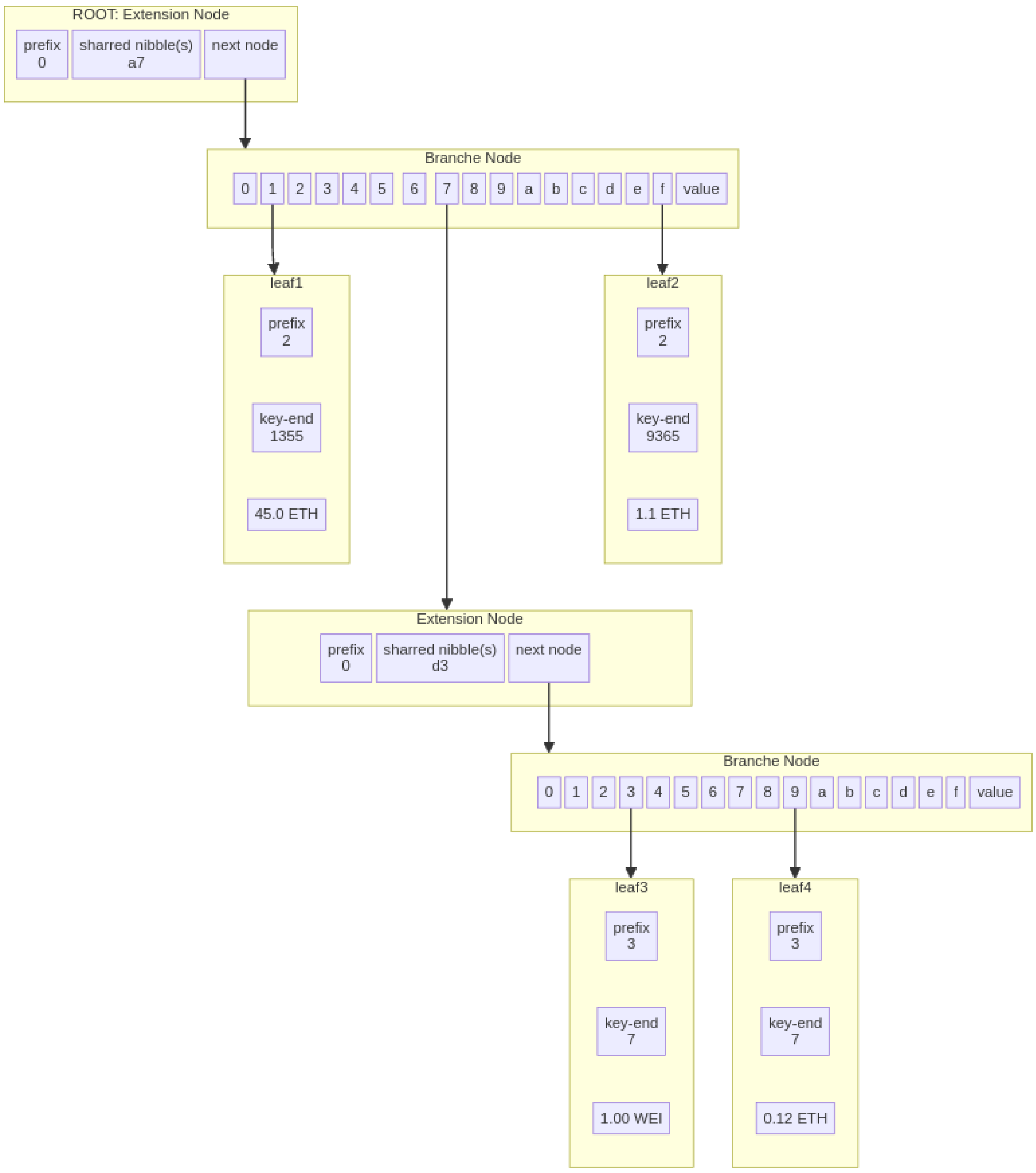
In this depiction, hashes 0-0 and 0-1 represent the hash values of data blocks L1 and L2, respectively. Meanwhile, hash 0 is derived from the combined hashes 0-0 and 0-1.

For comprehensive information on the Merkle Tree, visit here. The benefits of the Merkle Tree data structure are outlined here.

## 1.4. Merkle - Patricia trie

A Merkle-Patricia Trie (MPT) is a combination of a Merkle Tree and a Patricia Tree. This data structure is famous because it is being used by Ethereum to store the state of an Ethereum blockchain. The Ethereum version of the MPT is composed of 3 types of nodes:

- `Branch`: A node with up to 16 child links, each corresponding to a hex character.
- `Extension`: A node storing a key segment with a common prefix and a link to the next node.
- `Leaf`: An end-node holding the key's final segment and its value.


ROOT: Extension Node
prefix 0
sharred nibble(s) a7
next node
Branche Node
0
1
2
3
4
5
6
7
8
9
a
b
c
d
e
f
value
leaf1
prefix 2
key-end 1355
45.0 ETH
leaf2
prefix 2
key-end 9365
1.1 ETH
Extension Node
prefix 0
sharred nibble(s) d3
next node
Branche Node
0
1
2
3
4
5
6
7
8
9
a
b
c
d
e
f
value
leaf3
prefix 3
key-end 7
1.00 WEI
leaf4
prefix 3
key-end 7
0.12 ETH

# Chapter 2. Merkle trees uses and peculiarities in Starknet

In Starknet, Merkle trees serve as the backbone for organizing and securing the state of smart contracts and transactions. By utilizing custom Binary Merkle-Patricia Trees of height 251, Starknet optimizes state storage and retrieval, enabling efficient execution.
From an external perspective the tree is similar to a key-value store, where both key and value are Felt.

- Felt : a field element is a number 0..p-1 with $p=2^{\{251\}+17*2}\{192\}+1$, and it forms the basic building block of most Starknet interactions.

Refer to Evolution of a binary PMT upon data insertion for more details.

## 2.1. About Nodes

Each node in the trie is represented by a triplet (length, path, value), where:

- `path`: is the path from the current node to its unique non-empty subtrie.
- `len`: the length of the path, measured in nodes.
- `value`: is the value of the node, which can be either data, or the hash of two non-empty child nodes..

An empty node is one whose triplet values are (0,0,0) . Leaf nodes and internal nodes can be empty. A subtrie rooted at a node (□□□□□□,□□□□,□□□□□) has a single non-empty subtrie, rooted at the node obtained by following the path specified by □□□□ .

## 2.2. State commitment

```
The state commitment is a digest that represents the state. In Starknet, the state
commitment combines the roots of two binary Merkle-Patricia tries of height 251 in the
following manner:
```

```
state_commitment = hPos(
    "STARKNET_STATE_V0",
    contract_trie_root,
    class_trie_root
)
```

Where:

- `hPos` is the Poseidon hash function.
- `STARKNET_STATE_V0` is a constant prefix string encoded in ASCII (and represented as a field element).

- `contract_trie_root` is the root of the contract trie, a Merkle-Patricia trie whose leaves are the contracts' states.
- `class_trie_root` is the root of the class trie, a Merkle-Patricia trie whose leaves are the compiled class hashes.

## 2.3. The contract trie

As with Ethereum, this trie is a two-level structure, whose leaves correspond to distinct contracts. The address of each contract determines the path from the trie’s root to its corresponding leaf, whose content encodes the contract’s state.

The information stored in the leaf is as follows:

```
hPed(
    class_hash,
    storage_root,
    nonce,
    0
)
```

Where:

- `hPed` is the Pedersen hash function.
- `class_hash` is the hash of the contract’s definition.
- `storage_root` is the root of another Merkle-Patricia trie of height 251 that is constructed from the contract’s storage.
- `nonce` is the current nonce of the contract.

# Chapter 3. Introduction to the different implementations

## 3.1. PathFinder - Rust

Pathfinder tree is immutable, and any mutations result in a new tree with a new root. This mutated variant can then be accessed via the new root, and the old variant via the old root. Trees share common nodes to be efficient. These nodes perform reference counting and will get deleted once all references are gone. State can therefore be tracked over time by mutating the current state, and storing the new root. Old states can be dropped by deleting old roots which are no longer required.

### 3.1.1. Tree definition

It is important to understand that since all keys are Felt, this means all paths to a key are equally long - 251 bits. Starknet defines three node types for a tree.
* `Leaf nodes` which represent an actual value stored.
* `Edge nodes` which connect two nodes, and *must be* a maximal subtree (i.e. be as long as possible). This latter condition is important as it strictly defines a tree (i.e. all trees with the same leaves must have the same nodes). The path of an edge node can therefore be many bits long.
* `Binary nodes` is a branch node with two children, left and right. This represents only a single bit on the path to a leaf.

A tree storing a single key-value would consist of two nodes. The root node would be an edge node with a path equal to the key. This edge node is connected to a leaf node storing the value.

### 3.1.2. Implementation details

Pathfinder has defined an additional node type, an `Unresolved node`. This is used to represent a node who's hash is known, but has not yet been retrieved from storage (and we therefore have no further details about it). This implementation is a mix of nodes from persistent storage and any mutations are kept in-memory. It is done this way to allow many mutations to a tree before committing only the final result to storage. This may be confusing since we just said trees are immutable — but since we are only changing the in-memory tree, the immutable tree still exists in storage. One can therefore think of the in-memory tree as containing the state changes between tree `N` and `N + 1`.

The in-memory tree is built using a graph of `Rc<RefCell<Node>>` which is a bit painful.

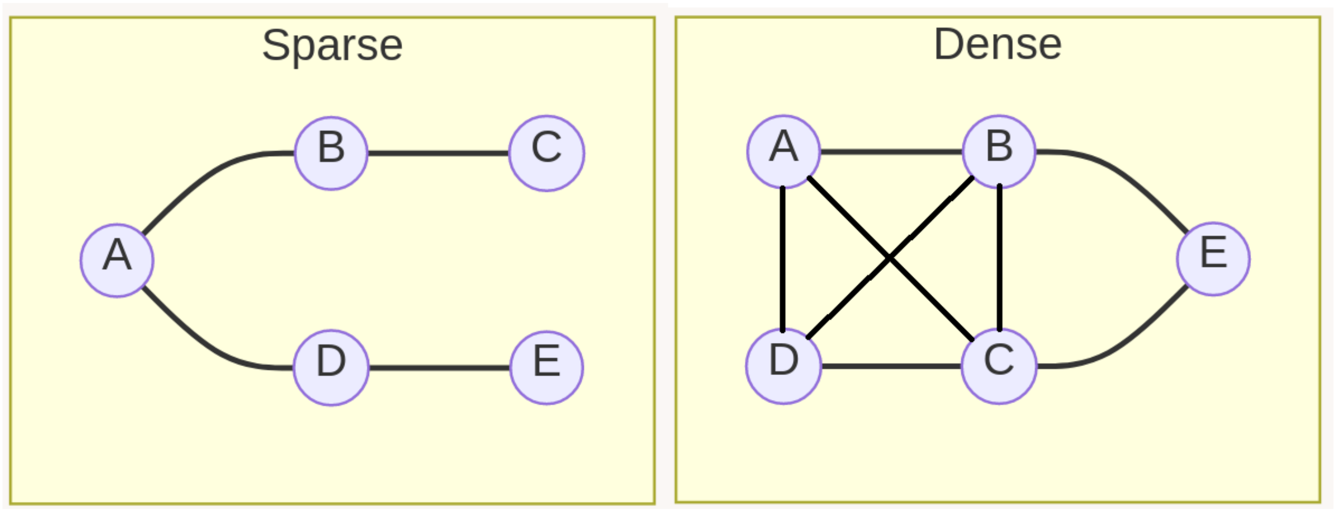
```
Pathfinder is the first Starknet full node implementation.
```

## 3.2. NethermindEth - Juno - Go

Juno is a dense Merkle Patricia Trie (i.e., all internal nodes have two children). This implementation allows for a "flat" storage by keying nodes on their path rather than their hash, resulting in O(1) accesses and O(log n) insertions.

The state trie Specification describes a sparse Merkle Trie. Note that this dense implementation results in an equivalent commitment.

Dense graph is a graph in which the number of edges is close to the maximal number of edges. Sparse graph is a graph in which the number of edges is close to the minimal number of edges. Sparse graph can be a disconnected graph.



Dense Graph If all vertices or nodes in a graph is densely connected (i.e. a node that connects with all neighbour nodes with all possible edges). Here possibly, total number of edges > total number of nodes.

Sparse Graph Its an vice versa of dense graph, Here we can observe that a node or vertices is not fully connected to its neighbouring nodes (i.e. it has unconnected/remaining edges). Here possibly, total number of edges ⇐ total number of nodes.

## 3.3. Starkware - Python

This PMT is immutable Patricia-Merkle tree backed by an immutable fact storage. Any modification applied to the PMT implies the creation of a new one representing the fact of the root of the new tree.

For efficiency, the update function does not compute and store the new facts while traversing the tree. Instead, it first traverses the tree to fetch the needed facts for the update. It then computes the new facts. Once all facts are computed, it finally stores them.

Used in production by the Starknet mainnet and testnets.

## 3.4. Bonsai - Rust

Besu bonsai storage is an advanced storage management system developed by HyperLedger for their Etherum client.
Originally, it was using a forest of PMT to store the world state. This had some drawbacks in terms of storage and acces to data. reorganizations (reorgs) as it only maintains a single state. Bonsai addresses this limitation by introducing “trielogs” (https://hackmd.io/@kt2am/BktBblIL3)

Quick comparision between forest of tries and bonsai tries

The state commitment scheme uses a binary Merkle-Patricia trie with the Pedersen hash function

This is used to update, mutate and access global Starknet state as well as individual contract states

Bonsai offers a compact data representation through its trie structure, reducing storage space requirements by storing nodes based on location rather than hash, thus optimizing read performance.
Direct data access using item keys simplifies read operations and enhances performance compared to hash-based methods. Automatic pruning of orphaned nodes and old branches ensures a clean and efficient data structure. Bonsai enables scalability and improved performance, catering to applications requiring fast and efficient data access.
Managing historical data becomes simpler with Bonsai, as only relevant data is retained, facilitating efficient data management.

In summary, Bonsai Trie offers space savings, improved read efficiency, automatic data management, enhanced performance, and simplicity in historical data management, making it a compelling choice for various applications, particularly within the Starknet ecosystem.

## 3.5. Comparaison (qualities and drawbacks).

WIP (should be update with starknet team comments)

| Implementation | Language | Qualities | Drawbacks |
|---|---|---|---|
| Pathfinder | Rust | - High performance,<br>- Scalability,<br>- Cairo specification adherence | - Low portability, Huge storage print |
| Juno | Golang | - Operational efficiency,<br>- User-friendliness,<br>- Ease of deployment,<br>- Ethereum compatibility,<br>- Use less storage than Pathfinder,<br>- JSON-RPC Api | - Not keeping reccord of past states,<br>- Can only answer calls about the chain head state |
| Starkware - Patricia tree | Python | | - Doesn’t have get_proof (not standard API)/ verify_proof method,<br>- Low portability,<br>- High dependency with Cairo |

| | | | |
|---|---|---|---|
| Bonsai | Rust | - Can revert to specific commit,<br>- Optimized for holding Starknet Felt items,<br>- Madara-compatible root hash,<br>- A Flat DB allowing direct access to items without requiring trie traversal,<br>- Trie Logs,<br>- Thread-safe transactional states | - Read data history becomes more resource-intensive than Forest,<br>- Does not support managing reorganizations, |

# Chapter 4. Testing implementations

**WIP**
This section is a preview of the next milestone
as such it is incomplete, subject to change and may contain errors.

## 4.1. Overview

This projet provides a validation framework for various implementations of PMT in various languages. The primary targeted languages are:

- C,
- go,
- rust,
- typescript,
- python,
- zig.

The solution is extensible to other languages.

### 4.1.1. Definitions

Validating an implementation of a PMT involves three actors:

- The PMT implementation to be validated: *Implementation* (for short),
- The validation framework: *Framework*,
- A test runner implemented in the same language as the PMT implementation to be tested: **Runner**.

#### *Implementation* role

State of the art implementation of the PMT in the targeted language.

#### *Framework* role

Offer an uniform way to execute operations over an *Implementation* and validate its states.

#### *Runners* role

Glues the *Framework* and the *Implementation*. Its role is to:

- initialize the *Implementation*,
- request test instructions from the *Framework*,
- call the *Implementation* methods accordingly to test instructions then pass the result back to the

*Framework*.

## 4.2. Design considerations

### 4.2.1. Languages interoperability

Generally, languages offer interoperability with other language with via the so called `FFI` foreign function interface. `FFI` is closely related to the language memory model and many low level details like function arguments passing for function calls, etc.

The defacto standard for `FFI` is the `C` ABI which literarily means that all languages have to pass (and retrieve) arguments to(from) functions the `C` way and organize struct fields the `C` way (to name few).

Given that the `C` language has no memory management, allocated objects have to be manually freed to prevent memory leak. Language with a garbage collector

ABI has an impact over API. API in `C` are provided through so called `.h` files that defines `structs`, `enum` and functions signatures.

Here are some details for each targeted languages:

- C: obviously `C` uses the `C` ABI. In order to use a library build for interoperability with `C` a client only need the binary of that library and ideally its associated `.h`. Provided those two elements
- go,
- rust,
- typescript,
- python,
- zig.

#### ***Implementation***

By definition a PMT *Implementation* defines its API, the set of operations to interact with the tree. From an implementation to another that set of operations may vary, see [main document](./main.md) for a comparison between known implementations. Because of those variations in operation sets some test cases may fail for some and succeed for others.

#### ***Framework***

The *Framework* is implemented in Rust. It is delivered as a library that exposes its interfaces with the `C` ABI in order to be interoperable with as many language as possible.

#### ***Runners* and bindings**

For *Runners* to call the *Framework* we also provide thin bindings to C for go, python and zig. The Rust *Implementation* can obviously go the native way.

For the TypeScript *Implementation* using a `wasm` interface was envisioned. That would have imposed some constraints upon the *Framework* (most notably being `no-std`). Thanks to Deno (a JS/TS runtime) it is possible call Rust fonction natively through ffi and keep the same logic for all targeted implementation.

## 4.3. Tests workflow

As said in *Runners* role, the test workflow is implemented by the *Runners*. *Runners* implementations are implemented in the same language as the *Implementation*. That leaves room for more flexibility.

*Runners* take the form of this pseudo code (Rust):

```rust
pub fn run_test(implementation, framework, test_case) {
    commands = framework.get_commands(test_case);
    for command in commands {
        run_command(command, implementation);
    }
}

pub fn run_all_test(framework) {
    tests = framework.get_all_tests();
    for test in tests {
        run_test(implementation, framework, test)
    }
}
```

The *Framework* provides a set of commands to be executed on the target *Implementation*. Each command should be implemented in the target language to execute the corresponding operation on the *Implementation*. (Golang example)

```go
func run_command(command *C.Command, tree *trie.Trie) {
    /// TODO : implement each command who can be executed on the trie
    switch command.id {
    case C.Insert:
        panic("Insert not implemented")
    case C.Remove:
        panic("Remove not implemented")
    case C.CheckRootHash:
        panic("CheckRootHash not implemented")
    case C.Get:
        panic("Get not implemented")
    case C.Contains:
        panic("Contains not implemented")
    }
}
```

Implementation example : TBD

Test case example :

```yaml
commands:
  - action: insert
    arg1: [1,1,1]
    arg2: "0x66342762FDD54D033c195fec3ce2568b62052e"

  - action: check_root_hash
    arg1: "0x4dff6cff9ce781e7700c6f43fd1e944736a9144ca946a280afe5fafea4e44cc"

  - action: insert
    arg1: [1,1,2]
    arg2: "0x66342762FDD54D033c195fec3ce2568b62052e"

  - action: check_root_hash
    arg1: "0x21c57215657fa4a475a54f99ad8571836dec23cd724ca25650dfdc7b554981b"

  - action: insert
    arg1: [1,1,3]
    arg2: "0x66342762FDD54D033c195fec3ce2568b62052e"

  - action: get
    arg1: [1,1,1]
    arg2: "0x66342762FDD54D033c195fec3ce2568b62052e"

  - action: check_root_hash
    arg1: "0x67fa4710f37a846c966d087db635a6be4d22451e1ba774f3f6f52829228eeb1"
```

## 4.4. Write your runner

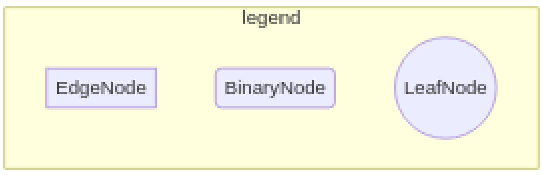
## 4.5. Implement the interface / trait that glued your implementation with the test framework

# Chapter 5. Annexes

## 5.1. Evolution of a binary PMT upon data insertion

In order to better understand how bonsai trie work, below are represented the different state of a trie while data are inserted.

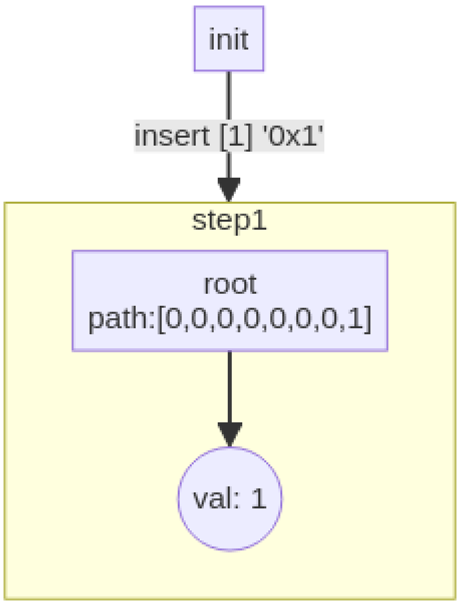
The different kind of nodes in the trie are represented according to:



**NOTE** In the following, node labels e.g. `n3`, `n4` are used to identify nodes and their evolution in the graph.

---

Upon insertion of the value `0x1` at key `1`, the trie is composed of root node (of type EdgeNode) with one child (of type LeafNode). The key to reach the value `0x1` is encoded in the `path` contained in the root node.



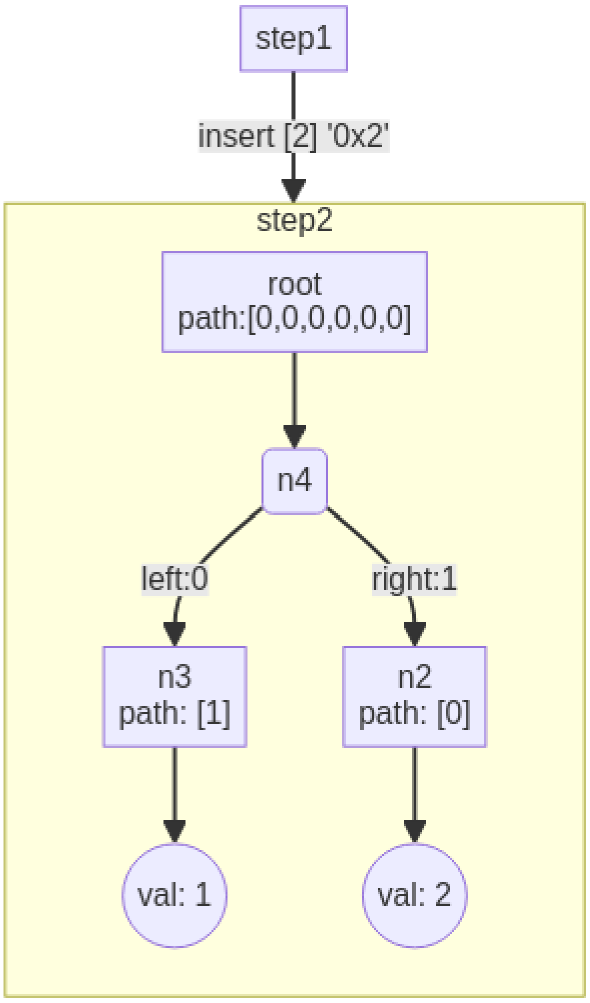
**NOTE** `init` is virtual state symbolizing an empty trie.

---

Upon insertion of the value `0x2` at key `2`, the trie reorganize itself as shown. As two keys (`[0,0,0,0,0,0,1,0]` and `[0,0,0,0,0,0,0,1]`) have now to be encoded:

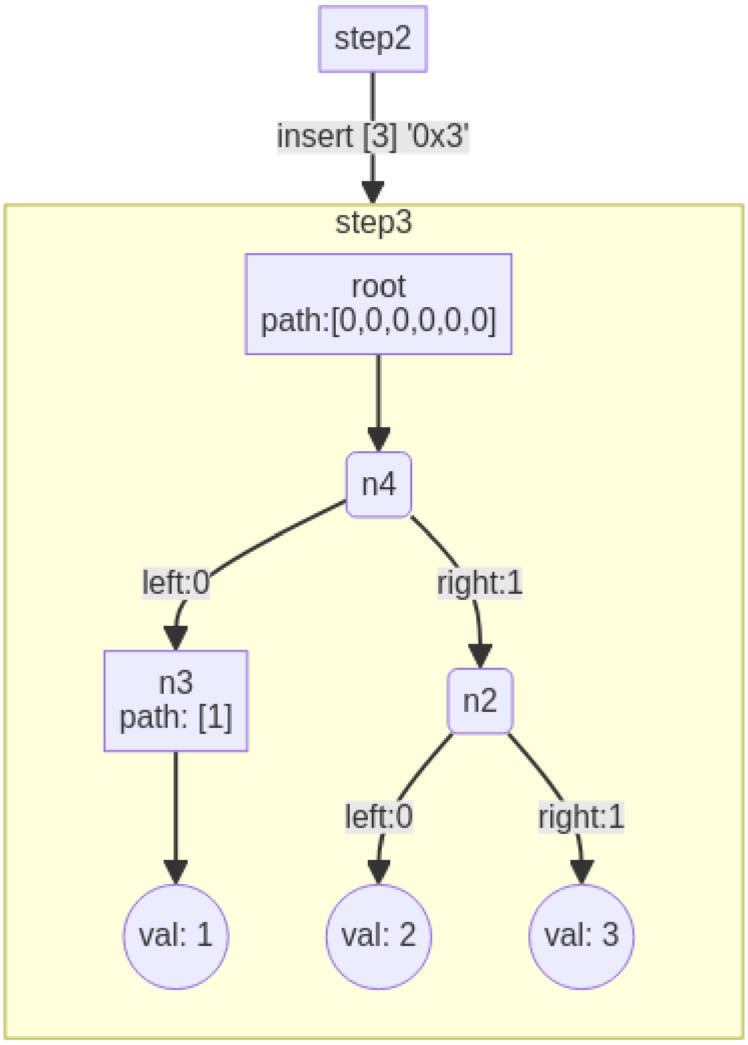
* the `path` of the root node is truncated to the common part of the two keys `[0,0,0,0,0,0]`.

- a binary node is inserted as the child of the root node. Each link to its children encode implicitly a part on the key, left for `0` right for `1`. Hence on the right path `[0,0,0,0,0,0,0]` is represented and `[0,0,0,0,0,0,1]` on the left.
- in order to encode the last bit of each key, an edge node with the appropriate `path` is inserted on both paths. As the keys are fully encoded at this step, both edge nodes have the corresponding leaf node as child.

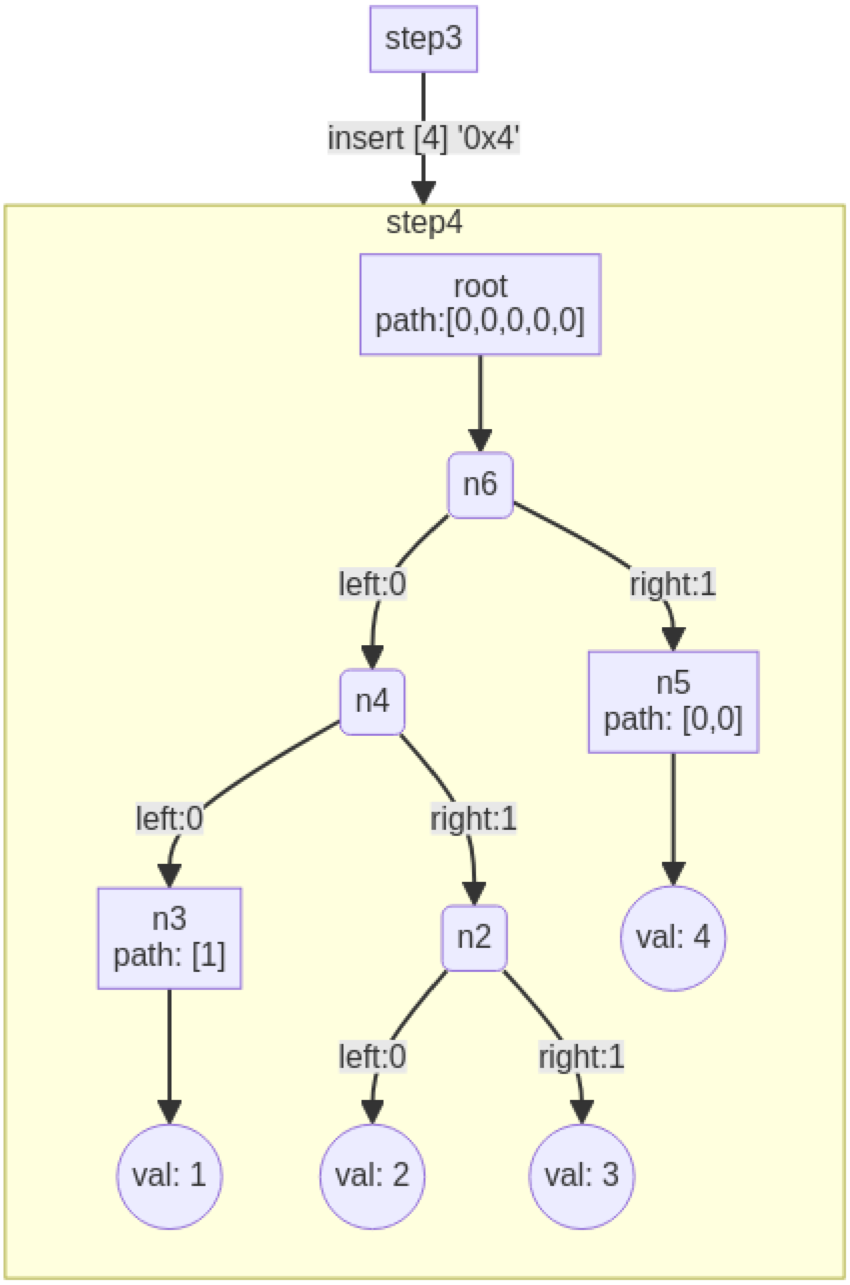


---

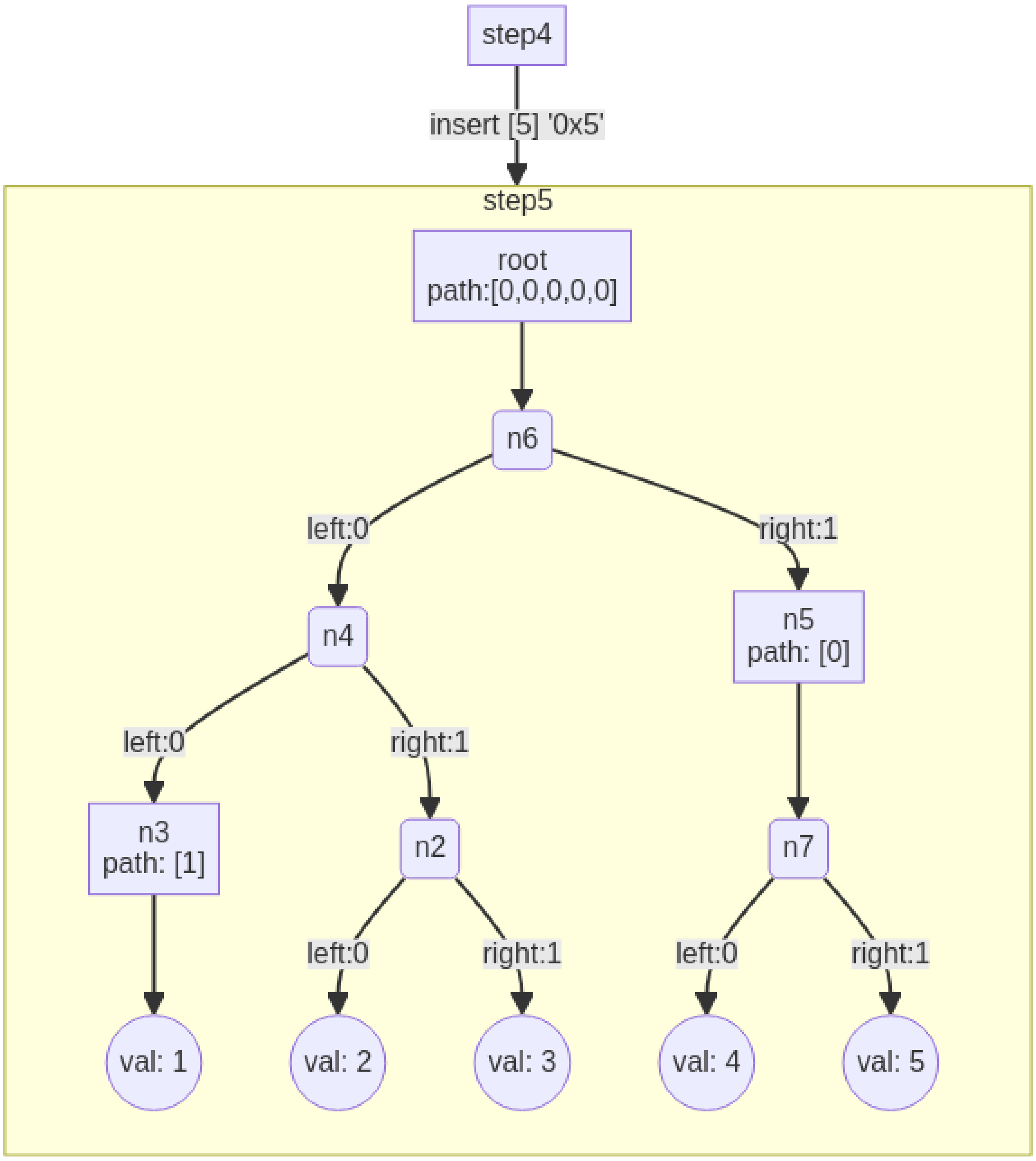
Upon insertion of the value `0x3` at key `3`, the trie reorganize again. The reorganization happens only to the subtree where new path have to be encoded. Hence the right child of the node `n4` is transformed into a binary node and the same reorganizations has explained above apply.



Upon insertion of the value `0x4` at key `4`, the trie reorganize itself as explained above.



The same kind of reorganizations apply upon insertion of the value `0x5` at key `5`.



**NOTE** upon keys removal "inverse" transformations apply.

## 5.2. Resources

- https://github.com/keep-starknet-strange/bonsai-trie
- https://github.com/keep-starknet-strange/bonsai-trie/blob/oss/src/trie/merkle_tree.rs
- https://hackmd.io/@kt2am/BktBblIL3
- https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture/6413#6413
- https://github.com/keep-starknet-strange/madara
- https://github.com/lambdaclass/merkle_patricia_tree
- https://github.com/massalabs/madara-bonsai/blob/main/documentation/specification.html (remove the link to the css to fix the display)

- https://github.com/massalabs/madara-bonsai/blob/d86abde7b0d5d0eb0503104ff69ace53e0b33ff9/documentation/specification.pdf
- https://github.com/massalabs/madara-bonsai/blob/d86abde7b0d5d0eb0503104ff69ace53e0b33ff9/documentation/specification.pdf

  ```
  https://docs.alchemy.com/docs/patricia-merkle-tries
  ```

- starknet docs : https://docs.starknet.io/documentation/architecture_and_concepts/Network_Architecture/starknet-state/#merkle_patricia_trie
- starknet book : https://book.starknet.io/

### 5.2.1. Hyperledger - Besu - Java

Similar data structure, for reference

Hyperledger besu implementation

Bonsai Tries is a data storage layout policy designed to reduce storage requirements and increase read performance.

Bonsai stores leaf values in a trie log, separate from the branches of the trie. Bonsai stores nodes by the location of the node instead of the hash of the node. Bonsai can access the leaf from the underlying storage directly using the account key. This greatly reduces the disk space needed for storage and allows for less resource-demanding and faster read performance. Bonsai inherently prunes orphaned nodes and old branches.