Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

June 24, 2022

Merklized B+Trees

Fast state synchronization in blockchain systems

Michele Cattaneo

Abstract

Merkle trees, or hash trees, allow secure content verification for large data structures and are widely used in blockchain systems. This thesis explores a novel concept of a merklized B+Tree to improve scalability of blockchains by making state synchronization for incoming peers faster. By exploiting the block oriented memory layout of B+Trees, we can verify the validity of chunks of data independently, without having to reconstruct the whole state, which is a great advantage in the case of misbehaving peers sharing invalid data. We initially propose a classical B+Tree where nodes have been adapted to contain the hash value of all the child nodes and then propose an evolution of a B+Tree, where advantageous properties of both B+Trees and binary trees are merged into a new data structure called B*Tree. This data structure has an overall binary nature but nodes are grouped together following the rules of a B+Tree into wrapping nodes. In both the merklized B+Tree and B*Tree, only the leaves of the tree are exchanged between peers, who can reconstruct an exact copy of the original tree via reconstruction algorithms.

Advisor
Prof. Fernando Pedone
Assistant
Eliã Rafael de Lima Batista

Advisor's approval (Prof. Fernando Pedone): Date:

Contents

1 Motivation

1.1 State-of-the-art

An important concept in distributed systems, such as blockchains, is state synchronisation. A new peer that joins the network must be able to catch up with the current state that is up to date with the one of the other operational peers. A possible way to obtain the state is to download the entire blockchain and execute every transaction that is present in the log. Clearly this effort can become huge with the growing number of transactions.

A possible solution adopted by some systems is to create a snapshot of the state which is a serialized representation of the data structures present in the blockchain. A new peer can join the blockchain by downloading the blocks with the transactions and the snapshot. To reconstruct the state, the peer needs to replay all transactions since the snapshot was taken.

The state can be stored in a Merkle Tree, which is a type of tree where leaf nodes store a value and its cryptographic hash. Inner nodes, instead, store the hash of their child nodes. As a consequence, any change in the tree will be reflected in the hash of the root node.

To reduce the time that it takes for a peer to join the network, the snapshot can be divided into chunks, where each of them contains multiple part of the state. The direct consequence is that a new peer can download chunks from many peers concurrently instead of transferring individual state units. When the state is stored in a tree, a possible strategy to assign nodes to chunks is by traversing the tree, for example using a depth-first search, and fill up fixed size chunks with them in the order of traversal [2]. The joining peer can then check the validity of the state by recomputing the hash of the reconstructed tree. The hash must then correspond to the value found in the trusted block header.

1.2 Limitation of current systems

An important property of Merkle trees is that subtrees can be validated independently. Leaf nodes belonging to a certain subtree will have a single common ancestor, and the hash value of this ancestor depends on the content of those leaves. With a Merkle proof, one can compare the hash value obtained with the hash present in the trusted block header, to prove that the content of that subtree is valid. However, if chunks are filled up by traversing the tree until the chunk is full, the Merkle tree loses this property, and consequently the snapshot can not be validated unless the whole state has been downloaded. Assuming that a misbehaving peer is providing wrong chunks, this limitation does not allow the joining peer to notice the problem until the whole state is downloaded, effectively prolonging the time that requires the new peer to join the blockchain.

1.3 Solutions

A possible approach is an AVL* tree [2], where leaves are organized into chunks, so that a chunk always contains a subtree. Individual chunks can be downloaded concurrently and verified independently.

This project aims to explore the possibility to use a B+tree, which naturally groups leaves into chunks. A B-Tree is a self-balancing search tree made of nodes with a branching factor of k; each node has $k-1$ keys and k children. Whenever a node has reached its full capacity, it is split in half into two nodes, resulting in the middle key being promoted the parent node. If no parent node exists, a new root is created with that key. A B+tree instead is a special version of a B-Tree, where data pointers are only present in the leaf nodes, while the inner nodes only contain keys that are found in leaves, and are only used for navigation. This is exactly what we need; the values in the tree and their direct hashes are found at the lowest level, and there are inner nodes having a hash value representing all the content of the subtrees rooted at those nodes. With a Markle proof, individual leaves can be verified independently avoiding the problem described above. In case of a merklized B+Tree, the terms *leaf* and *chunk* are interchangeable. It is important to note that in these frameworks, where peers only exchange leaves with each other, one must be able to reconstruct the tree having only the leaves and must be able to obtain an exact copy of the tree. The order of insertion leads to many different trees with different shapes, and therefore different hash values in the inner nodes and, most importantly, in their root nodes. It is therefore necessary to develop a deterministic algorithm that is capable of rebuilding a tree in this scenario.

2 Background

B+Trees (Figure 1) are k-ary trees where k is the branching factor. Each inner node contains $k-1$ keys and k child nodes. At the lowest level there are leaf nodes, where the number of keys corresponds with the number of children, and children are pointers to the actual data stored. Whenever a node is full and a new key needs to be inserted, that node is split into two nodes and the middle key is promoted to the upper level (Figure 2). If the upper node is full itself, this causes a recursive split until space is found or a new root node is created. Keys within a node are sorted and navigation is done with a binary search, which is slightly different for inner nodes and leaf nodes, as inner nodes have a pointer more than the number of keys.

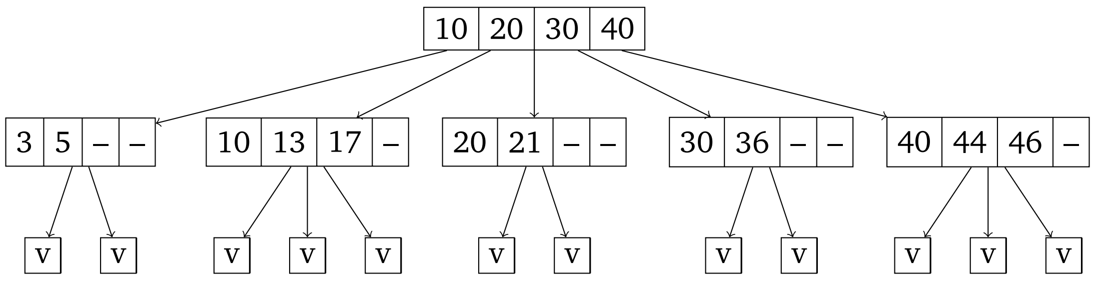


Figure 1. A generic B+Tree with branching factor $B = 5$, where inner nodes have 1 pointer more than keys, leaves have the same number of keys as pointers, and at at the last level there are values.

Splits in leaf nodes behave differently than splits in inner nodes; while promoted keys from inner nodes are simply brought up to the upper level, promoted keys in leaf nodes are copied and remain present in the right side node resulting from the split routine (Figure 3). This means that for every leaf node, its left-most key is also present as a copy somewhere up in the inner structure of the tree. The only case where this does not hold is for the left-most leaf node, where its left-most key is the smallest key in the tree and is not present anywhere else, as it is not affected by any split. Leaf nodes can also be traversed linearly as a linked list of nodes, which can be useful in case of queries with a range of keys.

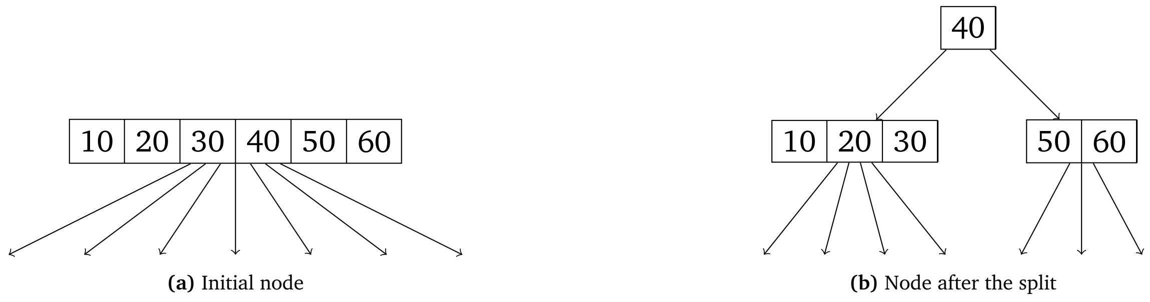


Figure 2. Split operation on an inner node

There are two ways of implementing the split routines that guarantee a correct balancing of the tree.

The first one requires that all full nodes encountered on the path that leads to the target leaf be split in advance. This ensures that there will be no recursive splits after the key

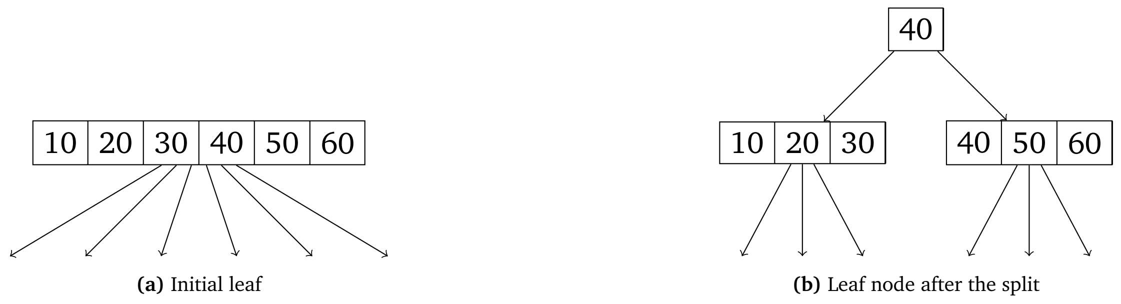


Figure 3. Split operation on an leaf node

in inserted in the target leaf node, as it is ensure that all nodes above will have space to host the promoted key. This method is simple but can lead to unnecessary splits, ultimately resulting in sparser nodes. The second one simply finds the target leaf, and in case of a split, promotes a key that can potentially cause recursive splits up to the root node, creating a new root.

B-Trees are mostly used for storage systems that read and write data in blocks, such as databases and file systems. Our goal is to have data grouped into large chunks that can be downloaded from peers independently, which is naturally achieved with a B+Tree. With a merklized B+Tree we would obtain a result similar to the one of a chunked binary tree, where leaves belonging to some subtrees are chunked together. However, with a branching factor of B, the size of a proof of inclusion grows linearly for each node. The size of a proof in a balanced binary merkle tree is in the order of

$$\mathcal{O}(\log_2(n))$$

In the case of a B+Tree the size of a proof is in the order of

$$\mathcal{O}(\log_B(n) \cdot B)$$

Even though in both cases complexity is logarithmic, the latter one is bigger by a certain constant. For this reason we discuss in Section 4 the possibility of adding a small merkle tree inside each node, to reduce this linear term into a logarithmic term. The array of keys and pointers used for navigation within each node is replaced with a self balancing binary tree. Within this framework, a Merkle proof does not require a linear number of hashes for each node anymore, but only a number in a logarithmic order of the size of the node.

3 Merklized B+Tree

3.1 B+Tree design and implementation

This section explains how we obtained a merklized B+Tree, such that only leaves are stored in stable storage and, starting from those, each peer can obtain an exact copy of the tree as all other peers, by means of a reconstruction algorithm. Hash computation and Merkle proofs are also discussed for the case of a B+Tree. Finally, the serialization and storage of leaves is explained.

3.1.1 Data structures

The two main data structures needed are the node and the tree. The first one will hold an array of keys and an array of pointers to other nodes. The latter will keep track of the root node, the left-most leaf node, the order (or branching factor) and the next unique identifier available for a new leaf node.

A node can take three forms; an *inner node*, a *leaf node* or a *record*. Inner nodes have $B-1$ keys and B pointers to other inner nodes or leaf nodes. They belong to the part of the tree used for navigation. Leaf nodes have a direct pointer for each key, as they effectively hold the data. The left-most key of a leaf is present in an inner node at a certain height. This integer value is stored in leaf nodes and will be used for the reconstruction. Additionally, leaves have a unique identifier, along with a pointer to the next leaf so that range queries can be executed efficiently. Record nodes (or simply records) are pointed directly from the keys within a leaf node and contain the K-V pair. All nodes but the record have a pointer to their parent node, so that traversals can be done in both directions. This pointer is particularly useful in the handling of splits, which propagate upwards.

3.1.2 B+Tree Insertion

Data mapped to a specific key can be added to the tree through the insertion routine. Both key and value are arrays of bytes which can be sorted lexicographically to determine the position in the tree. When a new K-V pair is added, the tree is traversed to find the target leaf, which is the leaf that should host the pair. If there is space, the key is added in the correct spot, so that all the keys remain sorted. Then a record is created with a pointer starting from the added key. If the target leaf is full however, a split must be executed and a key will be promoted to the upper node, which itself could be full, causing yet another split. The promoted key is selected by assuming that there is space available, so that the key has space to be inserted and then the middle key is chosen. This behaves different to splitting the node and then inserting the key. In the second case, for example, the added key can never be the promoted one, since the promoted key is selected first. Both ways are equally valid but we chose the first one. When a key is promoted to an upper level, there are the two nodes, *left* and *right*, resulting from the split that need to get a pointer. Recall that inner nodes have B pointers and $B-1$ keys, therefore each key in the node has a *left pointer* and a *right pointer* to other nodes. Before a split, there was a key k_0, whose right pointer is the node that will be split. The split will generate a promoted key k_1 such that $k_0 < k_1$ hence the promoted key will be placed on the right of k_0. Then the left half generated by the split will be pointed by k_0's right pointer, while the right half generated by the split will be pointed by k_1's right pointer. We have an edge case when the promoted key needs to be placed in the left-most position of the array, but the idea is similar. The insertion routine stops when the promoted key is placed in a node that still has space for it, or a new root is generated if the root needs to split.

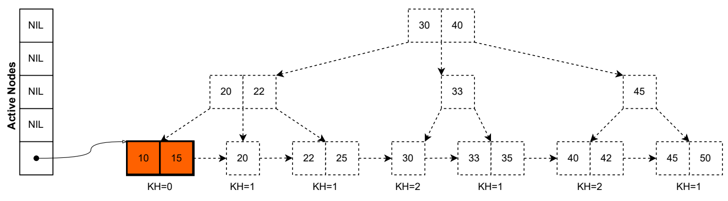
3.1.3 B+Tree Reconstruction

The B+Tree should not be stored as a whole in stable storage. We want to be able to only store leaf nodes that represent the actual state of the application. An algorithm must be developed in order to rebuild the exact same B+Tree that generated the leaf nodes that have been stored, since the order of insertion determines the shape of the tree. To do so, additional information has to be added inside the leaf nodes, so that when they are retrieved, they can be used by the algorithm. It is important to observe that when a leaf node is split, the left-most key of the second node resulting from that split (the right hand side one), is copied and promoted to the upper level. Further splits of inner nodes can bring that key to higher levels, without being copied. Therefore, every left-most key of any leaf node is also present somewhere up in the inner structure of the tree. This information is sufficient to reconstruct the structure of the original tree, which can simply be kept inside the leaf structure. We call the height of the key `keyHeight` and we call the leaf node that contains that specific key `associated leaf`. The value of `keyHeight` changes during the insertion routine at every split, and therefore it is convenient to have a direct pointer from each key to their `associated leaf`, so that we can update the value without having to traverse the tree. This requires that an additional array of pointers, the same size as the array of keys, is stored within each node. The value of `keyHeight` is vital for a correct rebuilding of the tree, and must then be part of the values to be hashed, so that misbehaving peers can not provide wrong chunks, without being noticed.

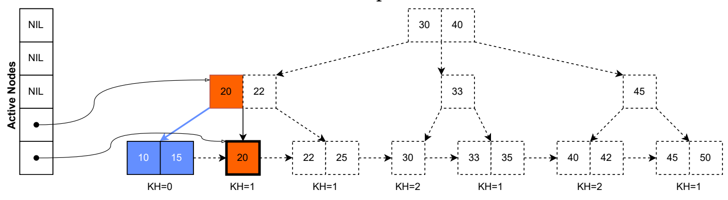
The algorithm requires, as an input, the leaf nodes provided as a sorted list, according to the left-most keys, meaning that they will be in the same order as the lowest level of the final reconstructed tree. Leaves will be processed one by one in the order they are found in the list. By *processing* a leaf we mean that its left-most key and `keyHeight` value are read and some actions are taken depending on those values, to advance the reconstruction by one step. For the following discussion we will define an *active node* as an inner node that is being built but not yet finished, and a *closed node,* as an inner node that is completely reconstructed and will not change. The algorithm keeps track of an array of active nodes by knowing that at most one node for each level will be active at the same time. Once a node at a certain level is closed, the value in the array for that level will be set to NIL. For every leaf processed, we need to place its left-most key k_i in a node at a height h_i determined by its `keyHeight` value. It is possible that an inner node n_h at that height is *active*, meaning that it was not closed yet and k_i can be added in the already existing node n_h. Otherwise, a new node is initialised and set as the currently active node for height h_i.

Every key k_r added in an inner node is a root key for a left and right subtree. After any of these keys is added, because a leaf has been processed, we can close its left subtree by closing and connecting all *active* nodes found at a lower height than the current one. This is because all keys, both in inner and leaf nodes are smaller than k_r and therefore must have already been inserted in the tree. Because of this, we always close an active node that is the left child of a key k_r, but then the remaining active nodes are right children of the right-most key in node that has just been closed. This is because all left subtrees were closed in previous steps. With this information we can define *active* nodes as nodes that are left-children of a key that has not been added yet, while *closed* nodes as nodes that are either the left child of an already inserted key, or the right child or the right-most key in a node. Initially all levels are set to closed.

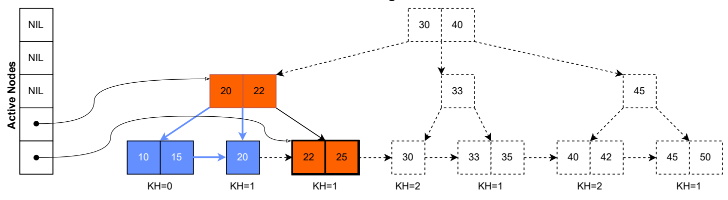
Figure 4 shows four steps of the reconstruction, where we denote the current leaf being processed with a bold border. Nodes colored in orange are pointed by a pointer in the list of active nodes. It suffices that one key is orange for the whole node to be active, however only those keys with an orange background are currently known and present in the node. Nodes and pointers in blue are considered closed and therefore fully reconstructed. Pointers and nodes with dotted black lines are currently not known and represent the original tree which is obtained once the reconstruction routine ends.



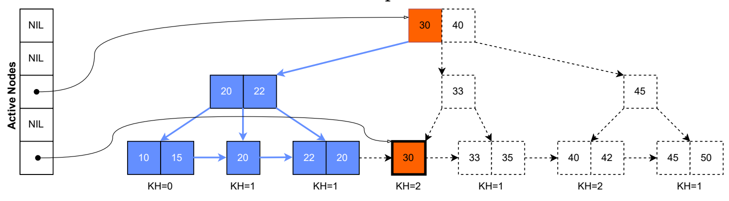
(a) First leaf processed



(b) Second leaf processed



(c) Third leaf processed



(d) Fourth leaf processed

Figure 4. B+Tree partial reconstruction

In Figure 4a the first leaf is processed. As it has height 0, the node is set as active and the step finishes. In Figure 4b, a node at height 1 is created, as there was no pointer for that height yet, and the key 20 is placed in it. Because a key $k_r = 20$ is added in an inner node, we can close its left subtree by connecting all active nodes found at a lower height, as shown in blue. In Figure 4c, key 22 is added in the existing node at height 1, and again the active nodes below are closed, determining the left subtree of $k_r = 22$. Figure 4d shows the last step of this example, where a new node is created at height 2, since there was no pointer yet. Then $k_r = 22$ is inserted in the inner node and its left subtree can be closed; the first active node is the left child of the added key, while the other active node is the right child of the right-most key in the first active node.

When the last leaf in the list is processed, the list of active nodes will still contain pointers to the nodes that belong to the right-most path, from the root to the last leaf. Those nodes can simply be connected together from the highest to the lowest and the algorithm terminates.

Algorithm 1 B+Tree reconstruction

```
procedure RECONSTRUCT(L: []*Node)                       ▷ Given a list of sorted leaves
    $h \leftarrow$ max. height of the tree
    levels $\leftarrow$ *Node[h]
    for leaf $\in L$ do
        curr_h $\leftarrow$ leaf.h

        if levels[curr_h] = NIL then
            levels[curr_h] $\leftarrow$ new empty Node
        ADDKEY(levels[curr_h], leaf.left_key)
        for $0 < k <$ curr_h do
            parent $\leftarrow$ levels[k]
            child $\leftarrow$ levels[k-1]
            ADDCHILD(parent, child)
            child.parent $\leftarrow$ parent
        for $0 < k \leq$ curr_h do
            levels[k-1] $\leftarrow$ NIL                           ▷ close nodes below
    for $0 < k \leq h$ do                          ▷ connect the nodes left in the list
        parent $\leftarrow$ levels[k]
        child $\leftarrow$ levels[k-1]
        ADDCHILD(parent, child)
        child.parent $\leftarrow$ parent

procedure ADDKEY(N: *Node, K: []Bytes)        ▷ Add a key in the first spot available
    N.keys[N.size] $\leftarrow$ K
    N.size++

procedure ADDCHILD(parent: *Node, child: *Node)                        ▷ Add a pointer
    if parent.pointers[parent.size-1] != nil then
        parent.pointers[parent.size] $\leftarrow$ child
    else
        parent.pointers[parent.size-1] $\leftarrow$ child
```

3.1.4 B+Tree hash computation

Hash values are computed using the sha256 algorithm which for our purposes is provided by the crypto package of the standard library [3] of the Go languages. The hash value of a record is obtained by concatenating the key and value pair together, and then the concatenated value is hashed. The hash value of a leaf nodes is obtained by concatenating the hash values of its child nodes, which are all records. Additionally the value of `keyHeight` is appended to the array of bytes, which is then hashed. It must be ensured that every value stored in a leaf which can lead to an incorrect reconstruction, is part of the hash value, since leaves are exchanged between peers. The hash value of inner nodes is simply obtained by the concatenation of the child nodes's hash values.

After every insertion, the tree has the necessary hash values modified so that the root's hash is up-to-date according to the content of the tree. Insertions lead to splits, for each of those, the two halves need to update their hash values. Additionally, a split leads to a key to be promoted, resulting in its `keyHeight` value to increase by 1. Being this value part of the hash value of the leaf that contains it, the whole path from the leaf up to the node that received the promoted key must be updated, bottom up. The last node to receive the promoted key resulting from the last split is the root of a whole subtree with valid hash values. The last thing to do is to update the hash values from that node up to the root of the tree, so that the whole tree is updated.

3.1.5 B+Tree Merkle proofs

Merkle proofs are made of a number of hash values that when processed in a certain way allows peers to obtain a final value that corresponds with the root's hash if the proof and data to be verified are correct.

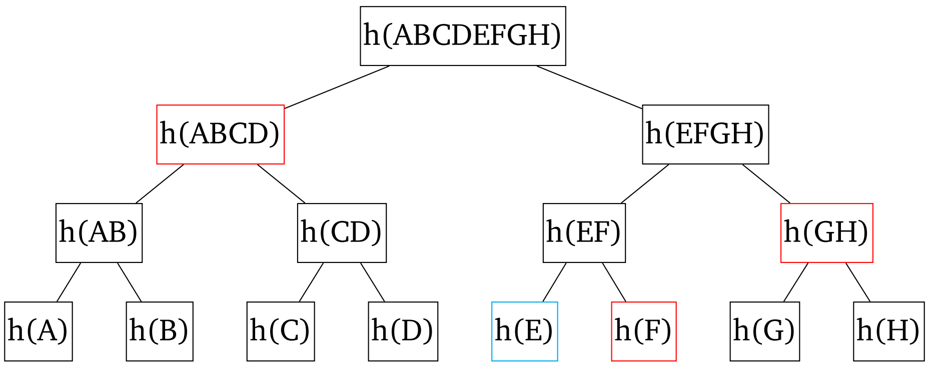


Figure 5. A Merkle proof in a binary tree. Highlighted in red are the nodes belonging to the proof, in blue the node to be verified.

In Figure 5 we can prove that element E is part of the tree with the proof that is composed by $h(F)$, $h(GH)$ and $h(ABCD)$. Let's assume that the hash value of a certain node is obtained by concatenating the hash values of the two child nodes, and by then hashing this value. With the additional information telling us whether a hash in the proof is part of the left or right branch, we can get the root's hash. The proof will look like this: $[(h(F),R),(h(GH),R),(h(ABCD),L)]$. To prove the that E is in the tree, one has to take the first value in the proof and knowing that it belongs to the right branch, concatenate $h(E)$

with $h(F)$ and hash them to obtain the intermediate result $h(EF)$. Then one has to take the next value in the proof and knowing that it belongs to the right branch, concatenate the intermediate result $h(EF)$ with $h(GH)$ and hash the value. Finally one has to concatenate the last value in the proof $h(ABCD)$ with the intermediate result $h(EFGH)$, because this time the element belongs to the left branch, and hash them together to obtain the root's hash, which will match the expected value only if E belongs to the tree.

In a B+Tree the situation is similar, however the merkle path will not be binary anymore, and for each node in the path from the root to the value to be verified, there will be B hash values, corresponding to the children values of that node. The additional information this time is not whether the hash value in the proof comes from the left or right branch, but rather at what position the intermediate result must be put to obtain the concatenation of the B values that constitute the values defining the hash of the node.

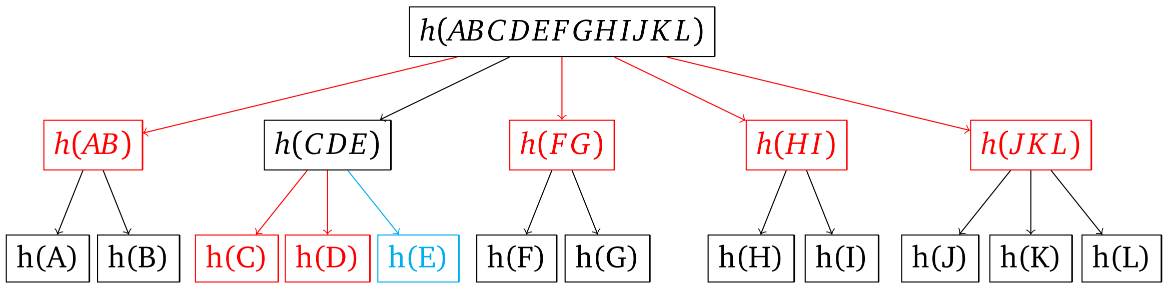


Figure 6. Proof of inclusion of a single element in a B+Tree tree. Highlighted in red are the nodes belonging to the proof, in blue the node to be verified, which is children number 2 of its parent node

Figure 6 shows an example of a proof inclusion for a single value in a B+Tree. If we want to prove the inclusion of element E for this tree, the proof will need the values of $h(C), h(D), h(AB), h(FG), h(HI)$ and $h(LMN)$. Element E is child number 2 of its parent node, then the proof will have the following form:

$$[(h(C), h(D), 2), (h(AB), h(FG), h(HI), h(LMN), 1)]$$

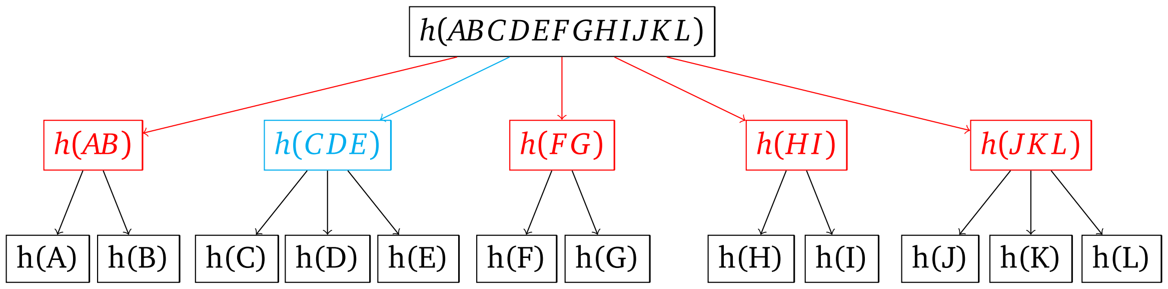


Figure 7. Proof of inclusion of a leaf node in a B+Tree tree. Highlighted in red are the nodes belonging to the proof, in blue the node to be verified, which is children number 1 of its parent node.

Figure 7 shows an example of a proof of inclusion of a whole leaf for a small B+Tree. Note this time that a whole leaf can be verified, as opposed to a single value; this is a big advantage of a B+Tree. Upon receiving the leaf node with values C, D, E, the peer can compute for himself the value of $h(CDE)$. Then a proof would look like this:

$$[(h(AB), h(FG), h(HI), h(LMN), 1)]$$

This means that the peer must put his intermediate value, that for the first and only step in the proof is $h(CDE)$, at position 1 of the concatenation to then hash these values and obtain $h(ABCDEFGHILMN)$. The hash obtained will match the root's value, only if the peer received the leaf with the correct content. The leaf was verified without having to reconstruct the whole tree.

3.2 Storage

Peers must be able to store and exchange chunks, which in this case are the leaves of the B+Tree. The in-memory representation of the leaves must be converted into a stream of bytes. This operation is called *serialization* and we have done it using the Amino encoding protocol, which is an object encoding specification and a subset of Proto3 [4]. Tendermint [6] provides Go bindings for the protocol that are available on their GitHub page[5]. Leaf nodes in our merklized B+Tree have one direct pointer to a record for each key. The records hold the actual key-value pair, and therefore we should serialize both the leaf and its records together.
To encode a leaf we need to determine an order for the elements that are put into the byte stream, so that, when decoding, we process the byte stream knowing the order of the elements and their size. We encode the node size first, the unique identifier, then for each key in the leaf, we encode the byte array containing the key and then the byte array of the record containing the value mapped to the key. Finally, the height of the key must be encoded since it determines the shape of the tree when it gets rebuilt. Recall that the height of the key signifies the level of the B+Tree, where the left-most key of a leaf is found, and it's used by the reconstruction algorithm.
To instead decode a leaf, we first read the size and ID in byte form and decode it back into an integer format. It is important that the size is placed before the keys and values in the byte stream, because we need to know how many key-value pairs we need to read. The length of those does not matter, as the encoding mechanism guarantees to correctly decode the byte slice that was given during the encoding procedure, but we do need to know how many byte slices were encoded. Once the size and ID are decoded, an in-memory leaf can be created, and for each key decoded, a record is created to hold the value, and a pointer from the key to the record is added. Lastly, the key height is decoded, and the leaf is fully formed. The leaf can be now used to rebuild the tree.

The code base is provided in the Go programming language. The reason is that we wanted this implementation to be possibly adapted and utilised within the Tendermint framework [6], the same way as the AVL* was. Additionally, Go is a simple language, yet has high performance and it is gaining a lot of attention in distributed systems and cloud computing. Lastly, Go's mascot is a cute gopher (a rodent from North and Central America), so I told myself: *Why not, let's go-pher it!*

4 Merklized B*Tree

4.1 Motivation

In a balanced binary tree of size n, a merkle proof has a length of $log_2(2n-1) = \mathcal{O}(\log_2(n))$, where n is the number of keys, hence the numbers of leaves in the tree. The total number of nodes in the binary tree is then $2n-1$. The length of a proof is in the same order as the length of a path from the root to the data being proven.

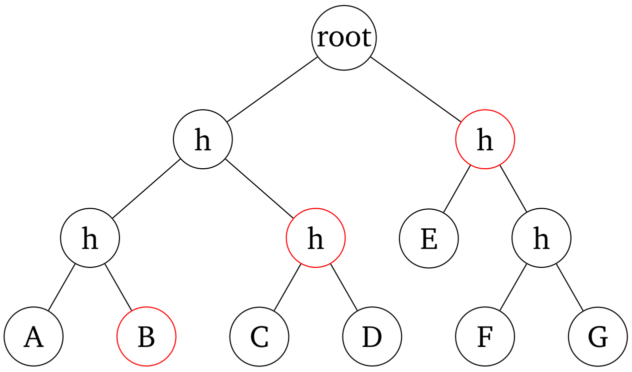


Figure 8. An example of a merkle proof; the hash of the red nodes is needed to prove that value A is part of the tree

In a B+Tree, instead, since each node has a higher branching factor, for each of those nodes, $B-1$ hashes have to be provided, so that the hash of the node can be fully recomputed. The hashes can not be provided as a unique hash because the position of the branch that is being proven contributes to the final hash value, as it is the result of the concatenation of all the child pointers' hash values. Therefore the overall cost of a merkle proof is in the order of

$$\log_B(n) \cdot (B-1) = \mathcal{O}(\log_B(n) \cdot B)$$

We want to explore the possibility of inserting a small binary merkle tree inside each node of the B+Tree, in order to reduce complexity of the linear term of those proofs. Each inner binary tree has B leaves, where B is the branching factor, hence $2B-1$ nodes. If such a data structure can be developed, the cost of the proofs should be reduced to be in the order of:

$$\begin{aligned}
\log_B(n) \cdot \log_2(2B-1) &= \frac{\log(2B-1) \cdot \log(n)}{\log(2) \cdot \log(B)} \\
&= \mathcal{O}\left(\frac{\log(B) \cdot \log(n)}{\log(2) \cdot \log(B)}\right) \\
&= \mathcal{O}\left(\frac{\log(n)}{\log(2)}\right) \\
&= \mathcal{O}\left(\log(n)\right)
\end{aligned}$$

This shows that implementing such an inner tree will lead to the same complexity as the one of a binary tree. We can think of this new concept by as a binary tree sub-divided into several groups of self-balancing trees, that balance themselves independently from the others. Those groups of self-balancing trees are kept balanced by following the invariants that keep the nodes of a B+Tree balanced. We can say that we keep an overall binary nature in the tree while also maintaining the property of the B+Tree leaves grouping together the values, which is an ideal scenario. The flip-side is that these inner merkle trees need to be kept up-to-date after each insertion.

4.2 First idea: use IAVL merkle trees

An IAVL tree is a slightly modified AVL tree [1], where values are only kept in leaf nodes, so that the balanced tree can be used as a merkle tree. Another important property is that the key value of inner nodes is the smallest key in the node's right subtree, which will be contained in the left-most leaf node. This is very similar in a B+Tree. Figure 9 shows an example of a IAVL tree using letters as keys. A first idea was to use such a tree inside each B+Tree node.

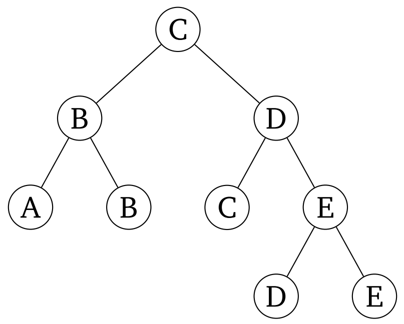


Figure 9. An example of IAVL tree

In the current B+Tree implementation, each node contains a sorted array of keys and an array of pointers used for navigation and for the hash computation of each node. Those could be substituted by an IAVL tree that will keep the logarithmic cost for navigation within the node, while introducing the benefit of amortizing the proofs' cost as discussed above. Removing the root of an IAVL tree, two valid IAVL subtrees are obtained. If those two subtrees have similar number of leaves, a B+Tree node split would be made remarkably easier and cheaper, since by simply removing the root of the IAVL tree of the node being split, the two subtrees would be ready to be used as the two new IAVL trees for the two nodes obtained by the split. No further hash computation would be needed for a leaf node split, as the two new root nodes already contain their corresponding hash values. Splits for inner nodes behave differently, because a split requires the middle key to be removed. The middle key is always the left-most key in the right subtree, which then requires an update to his hash values.

Experimentally we constructed random IAVL trees containing 2048 values, and therefore 2048 leaves. We noticed that the average difference between the number of leaves of the two subtrees, is in the order of 12% of the tree's total number of leaves. This means that the downside of having very efficient node splits, is to have splits that produce, in the average, nodes that differ in size by 12%. This result was a promising finding so we proceeded to construct 100'000 random IAVL trees with 1024 unique keys. For each of these trees, the value L and R was defined as the number of leaves in the left and right subtree respectively, if the root was removed. The measure of error was $|L-R|/1024 \cdot 100$. Then these errors were put into classes of errors by rounding their values to the closest multiple of 5. The sizes of these classes are plotted in Figure 10, showing the percentage that they represent out of the total 100'000 trees. The first 4 classes, until the 15% class, represent $\sim 72\%$ of the total.

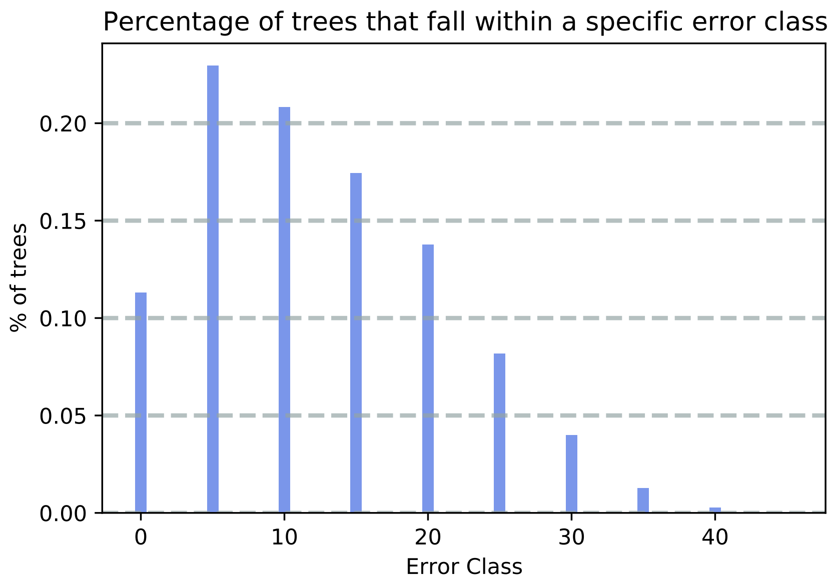


Figure 10. Percentage of splits that fell into a certain class of error, when splitting the tree in two by removing the root node, over 100'000 random trees.

Active peers will have identical IAVL trees for each node, because they all follow the same order of insertion, which creates the same splits in the nodes and, as a consequence (because of promoted keys), the same insertions within the IAVL trees.

A problem appears for recovering peers. We only want to store the leaf nodes of the B+Tree, which represent the actual data we want to store. Inner nodes should be generated during insertions or rebuilt based on the content of the stored leaf nodes. For this reason, a recovering peer will not be able to obtain identical IAVL trees as the other active peers, unless the insertion order for those IAVL trees is given. Different IAVL trees would lead to an overall different hash of the whole B+Tree's root, as the hash of the single nodes of the B+Tree would be the root hash of the inner IAVL tree. This poses a limitation on this idea, but it can be solved. An IAVL tree can be reconstructed if we know, for each leaf node, its height in the IAVL tree. Each key in the inner B+Tree nodes, hence in the IAVL of those nodes, is also a left-most key present in a leaf node. We can simply add an integer value that will be stored in the leaf node of the B+Tree which will be the key height of the key in the IAVL tree. The heights for the IAVL of the B+Tree leaf nodes will simply be stored with the other information. Figure 11 shows a schematic view of a B+Tree when binary trees are inserted into each node. It also shows that we can imagine this new tree from two points of view: a B+Tree containing binary trees, or a binary tree where groups of nodes are grouped and balanced together. The second point of view reflects the bound of a proof size that was discussed above, which should be in the same order as a generic binary tree. Note that to maintain the invariant for which B+Tree inner nodes have $B-1$ keys and B pointers, the left-most node of the inner IAVL has two pointers, which will be treated in a special way.

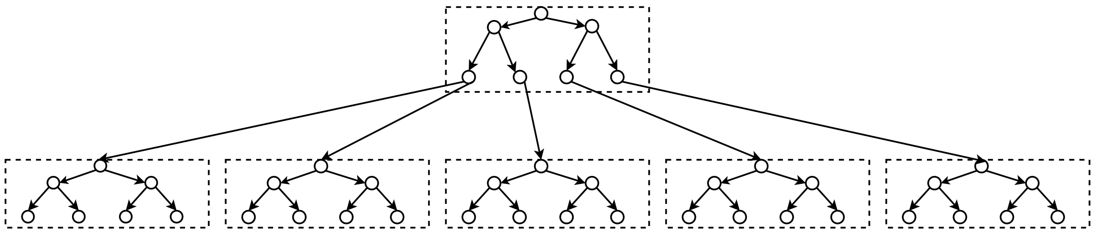


Figure 11. Schematic view of the B+Tree containing iavl trees for each node, delimited with a dotted line.

4.3 Second idea: use a heap-like tree

Another option would be to have a fixed size binary tree stored like a heap. This means that for each B+tree node, a fixed size array would be allocated to store the nodes of the binary tree. The second half will store leaf nodes, while the other half will store inner nodes. Heaps uses positions in the array instead of explicit pointers, for example a node found at position i in the array has its left-child at position $(2 \cdot i) + 1$ and its right-child at position $(2 \cdot i) + 2$.

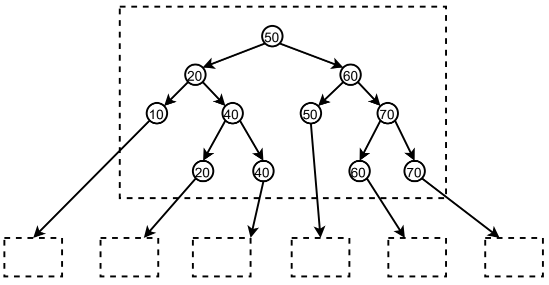
In this method, the array of child pointer to the lower B+Tree nodes that is found in all B+Tree nodes will be used as the second half of the heap, since they represent the leaves of the inner Merkle tree. The insertion routines already produce a shift in the pointers and keys when a new value is inserted, in order to make space for a new value. The heap will never grow in size but a shift in the leaf nodes will lead to the hashes of the subtree subject to the shift to be recomputed. This means that before the B+Tree nodes reached their full capacity, many nodes of the inner binary tree will have a NIL value. The hash value of an inner node with two NIL child nodes stays NIL, while if only one of the two branches has a NIL hash value, the value of the other branch is simply repeated for the parent. This creates a deterministic insertion routine for the inner tree; no matter the order of insertion, the leaf level will always be the same and therefore the whole heap will have the same shape, hence the same root hash value. This is an advantage in the case of rebuilding a tree. While for IAVL tree extra data is needed for a correct rebuilding of the inner tree, in this case no extra data is needed. Another possible advantage is the fact that the array is pre-allocated when the B+Tree node is allocated, while an IAVL tree requires dynamic memory and rotations to maintain the tree balanced. However insertion and splits lead to a more costly hash re-computation.

4.4 B*Tree

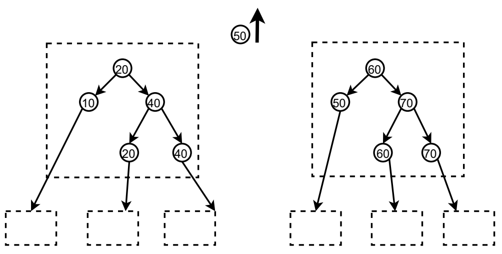
We call B*Tree a concrete implementation of the idea proposed in section 4.2. From this point onwards, we will refer to B*Tree nodes as *Bnodes*; for example an inner B*Tree node will simply be an inner Bnode. We will instead refer to *nodes* for inner IAVL tree nodes and *leaves* for IAVL tree leaves. The data structure is similar to the one of a B+Tree, but instead of arrays of key and pointers, an IAVL tree is used. The code that implements the algorithm of a IAVL tree has been taken from [1] and has been adapted for the needs of the B*Tree. Specifically the insertion and removal routines, together with the rotations routines have been reused. An IAVL tree invariant ensures that every inner node contains a key that correspond to the smallest key in the right subtree. The inner node has a specific height in the tree and this height value is stored in the relative leaf node that has the same key and will be called `kIAVLHeight`. Insertions and rotations modify the height value and in order to efficiently update this value in the leaf node every time it changes, a direct pointer from the inner node to the corresponding leaf is added.

4.4.1 B*Tree splits and insertions

A split in a leaf node is carried out by removing the root node from the inner IAVL tree, resulting in two subtrees which will serve as inner IAVL tree for the new left and right node respectively. The key from the old root, which corresponds to the smallest value in the right subtree, is promoted to the upper level, meaning that it will be inserted in the parent node, or it will form a new root node. This key was also present in a B*Tree leaf, which stores at what level of the B*Tree its key is currently found, and therefore this value must be increased by one. We will call this value `kHeight`. When a leaf node gets split, the promoted key must stay present in the right half. This means that keys found in inner Bnodes are also found in the left-most position of the leaf Bnodes. The two inner IAVL trees are balanced and the hash value of the two new roots is valid.



(a) A leaf node about to be split

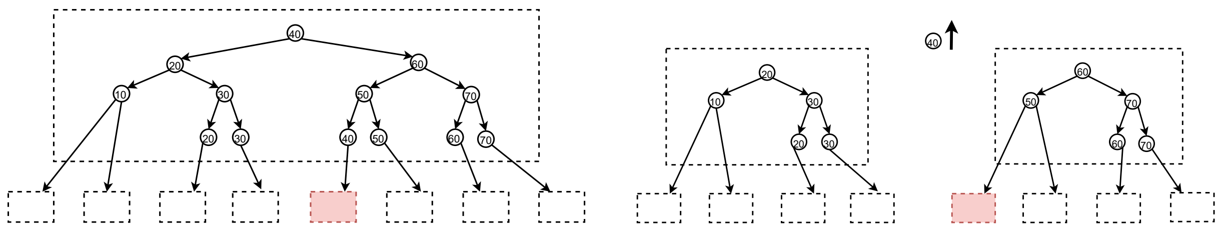


(b) The result of a leaf split

Figure 12. B*Tree leaf node spliting.

A split in an inner node requires more work; the IAVL leaf containing the promoted key must be removed from the right subtree, otherwise the key is duplicated again, and furthermore the new left-most IAVL leaf must inherit the pointer to the lower B+Tree node that was once a pointer starting from the removed IAVL leaf (Figure 13 in red). Recall that inner nodes have $B-1$ keys and B pointers; each IAVL leaf represents a key and the left-most IAVL leaf has two pointers to B+Tree nodes. The removal of a key from the tree results in the hash value of the right IAVL tree to not be valid anymore and the need to recompute it. This operation has a logarithmic cost in the size of the tree. Furthermore, all the B+Tree nodes that are children of the newly created B+Tree node serving as a right half, must have the pointers to their parent node updated, as they are still pointing to the in-memory address of the B+tree node before the split, which is now used as the left node.

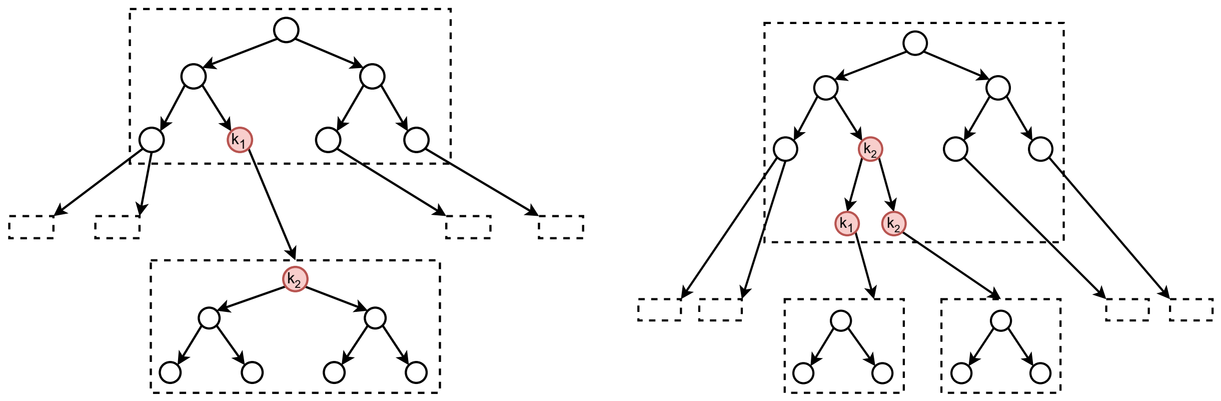
A leaf node split turns out to be very efficient, as the content and hash values of the two halves are already valid, due to the nature of a IAVL tree. An inner node split is slightly less efficient, as the right half requires a removal and a re-hashing, both with a logarithmic cost in the size of the half.



(a) A inner node about to be split **(b)** The result of an inner node split

Figure 13. B*Tree inner node splitting.

Insertions in a B*Tree as any other B+Tree can cause nodes to split in two. When this happens, we need to insert the promoted key into the inner IAVL tree of the parent node. Figure 14 shows the before and after of an insertion. When inserting k_2, the algorithm will traverse the IAVL tree and arrive at the leaf node containing k_1. To ensure the invariant for which each inner node contains the key of the smallest key in its right subtree, the algorithm will produce the subtree shown in Figure 14a in red. At this point the two new IAVL leaves will get a pointer to the left and right half respectively. The IAVL tree of the parent node has now grown in size and could potentially have exceeded its maximal size. If that is the case, the parent node itself will split and the procedure is repeated. If the root node has split, a new root node with a single key is created. A high level description of the insertion routine is presented in Algorithm 2.



(a) The inner node subject to the insertion **(b)** The inner node after the insertion

Figure 14. B*Tree key promotion; insertion of a key in an upper node.

4.4.2 B*Tree hash computations

The hash value of a B*Tree node depends on the hash values of its children nodes. The following cases are possible:

- The hash value of an inner Bnode corresponds to the hash value found at the root of its IAVL tree.
- The hash value of a leaf Bnode corresponds to the hash value found at the root of its IAVL tree, the `kIAVLHeight` value and the `kHeight` value concatenated and hashed together.

Algorithm 2 B*Tree Set

```
 1: procedure SET(K: key, V: value, tree: B*Tree)        ▷ Takes a K-V pair of byte slices
 2:     node ← tree.findLeaf(key)                        ▷ Traverse down to the leaf
 3:     node.addRecordInLeaf(key, value)
 4:     while node.mustSplit() do
 5:         left, midKey, right ← node.split()
 6:         if node.parent = NIL then
 7:             tree.addRoot(left, right, midKey)
 8:             break
 9:         else
10:             node.parent.innerNodeInsertion(left, midKey, right)
11:             node ← node.parent
12:     return
```

- The hash value of a record node corresponds to its key and value concatenated and hashed together.

Since the hash value of inner and leaf Bnodes depends on the hash value of its inner IAVL Tree root node, we must define how the hash of an IAVL node is computed:

- The hash value of an inner node corresponds to the hash value of its two child nodes concatenated and hashed together.
- The hash value of an IAVL leaf node inherits the hash value of the B*Tree node that is pointing to, which can be a leaf, inner or record Bnode. A leaf IAVL node can point to a record only if it belongs to a B*Tree leaf. A leaf IAVL node that belongs to a inner Bnode can both point to another inner Bnode or a leaf Bnode.

In case of inner Bnodes, the left-most leaf of the IAVL tree has two pointers to other Bnodes, and hence its hash value is the concatenation of the hashes coming from its left and right pointer.

B*Tree insertions have the side effect of changing the content of leaf nodes by adding new records. The path from the target leaf (the leaf receiving the record) up to the root Bnode contains hashes that are not up to date anymore and have to be recomputed. Insertions that lead to Bnode splits have the side effect of changing the values of `kIAVLHeight` and `kHeight`. Each split promotes a key, changing a value of `kHeight` in some leaf. The path from that leaf up to the inner node that contains the promoted key, is formed by hashes that need to be recomputed as they are not up-to-date anymore. Promoting a key means to add it in an inner IAVL tree, which to be kept balanced needs to perform rotations, effectively changing some `kIAVLHeight` values. For any of these changes, the path from the affected leaf up to the inner Bnode that caused the changes, must have its hashes updated. Finally, splitting an inner Bnode in two, requires that the promoted key is removed from the IAVL tree of the right half (See Figure 13b). This operation also results in some rotations to keep the tree balanced, and hence more `kIAVLHeight` values changed. Again, the path from the affected leaf up to the right half has to have its hash values updated.

For all of these cases, the path of hashes to update has a binary nature as it acts on the IAVL trees, meaning that for each BNode, only part of its IAVL tree is affected by the

changes (from the IAVL root down to the leaf leading to the next Bnode), while the rest is left untouched.
We can summarize the situation as follows: Inserting a value results in some potentials splits, which cause keys to change level in the B*Tree, hence their value of `kHeight` and their height within the Bnode's inner IAVL Tree, and therefore their value of `kIAVLHeight`. Both these values define the hash value of their associated Bnode leaf, therefore every time one of these values changes, the path from the current inner node subject of a split, down to the associated leaf has to be updated by rehashing it. When splits stop, or if there was simply space in the leaf to contain the new key, then the path from the tree root down to the last affected node has to be updated by rehashing it. This means that we need two procedures to update hashes along a path; one from the root Bnode down to a certain Bnode, the other from a Bnode, down to a certain Bnode leaf.
In Algorithm 3 we provide the procedure that updates the hash value from the root of the B*Tree, down to the last node affected by the insertion and potential splits, called *target*. The last node affected by splits is the last node that received a promoted key and had space for it. This algorithm assumes that once it is called, the target node has its hash, hence the hashes of its IAVL tree, up-to-date. After this procedure is called, the whole B*Tree has an up-to-date hash in its root. The procedure traverses with recursion all Bnode on the path from the root to the target, and for each of those Bnodes, traverses their IAVL Tree setting a boolean flag within each IAVL node to denote that their hash is not valid. Let's call *current*, the node that called the procedure. After the recursive call of the procedure on the child Bnode, the current Bnode can update the hash of its IAVL tree, since the lower nodes have finished their operation and are therefore updated. To update the hash of its IAVL tree, the current Bnode uses the function at line 17, which makes use of the boolean flag that has been previously set; if invalid, compute the hash by recomputing the hash of its subtree. If valid, use it as it is. The base case occurs when a leaf has an invalid hash; then the hash is computed by using the one of the Bnode pointed by that leaf.

Algorithm 3 Update Hash up to the root

```
1: procedure ROOTTRAVERSEHASHUPDATE(n Node, target Node)
2:     if n = target then                              ▷ Reached the last node to update
3:         return
4:     iavlNode := n.iavl.root
5:     for !iavlNode.isLeaf() do                       ▷ Set the path on the IAVL to be updated
6:         iavlNode.hashIsValid ← false
7:         if target.key < iavlNode.key then
8:             iavlNode ← iavlNode.left
9:         else
10:            iavlNode ← iavlNode.right
11:    iavlNode.hashIsValid ← false
12:                                                    ▷ Recursion on the child node
13:    if iavlNode.leftBnode ≠ NIL & target.key < iavlNode.key then
14:        ROOTTRAVERSEHASHUPDATE(iavlNode.leftBnode, target)
15:    else
16:        ROOTTRAVERSEHASHUPDATE(iavlNode.rightBnode, target)
17:    n.iavl.root.recursiveHash()                     ▷ All children updated, update this hash
18:    return
```

4.4.3 B*Tree Merkle Proofs

Merkle proofs in a B*Tree have the same structure that they have in a binary tree, as described at the beginning of Section 3.1.5. The hash values of the Bnodes are defined by the hash values of their inner IAVL trees and each of these trees can produce its proof. The overall proof is the concatenation of all the IAVL's proofs found on the path from the B*Tree root, down to the leaf Bnode that needs to be proven.

4.5 IAVL tree reconstruction

When rebuilding a B*Tree, we need to rebuild the IAVL trees within each node. To do so, we propose an algorithm (Algorithm 4) that makes use of the height of the keys in the inner nodes, that works in a similar way as Algorithm 1. This time around, it must be taken into consideration that leaf nodes in a balanced binary tree are not found at the same level. We define level as the distance from the root, while the height is the maximal depth of any subtree of a node. For a balanced tree we allow the difference in height between the two subtrees to differ at most by 1. Figure 14b shows the result of an insertion in an IAVL tree; the keys k_1 and k_2 are found in leaves at the third level, while the other leaves are found at level 2. However all the leaves are found at height 0.

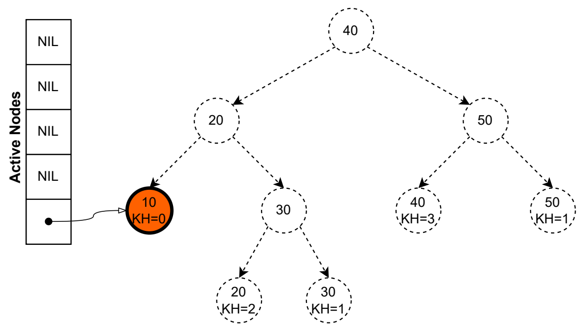
The algorithm requires a sorted list of leaf nodes, which is processed linearly. There is a list of active nodes currently placed at a certain height. Every time a leaf node is processed, the value of its key-height is read and a node is placed as active at that height. Then its left subtree can be generated by connecting all the other active nodes in the active list until level 0. The leaf node and the corresponding inner node stay active, while the other nodes are removed from the list and considered as closed. Closed nodes belong to subtrees that will not be affected anymore by the construction algorithm and are considered completed. When all leaves have been processed, the active list will still contain all the nodes that belong to the path starting at the root and ending at the right-most leaf, which still need to be connected together. At this point the tree is rebuilt and the hash value can be computed. For large trees, it could be interesting to compute the hash value of completed subtrees in parallel, while the rest of the tree is still being reconstructed, however careful synchronization would be needed. This problem is left as a possible future work. An example of a partial reconstruction is shown in Figure 15. Orange nodes are active nodes and are pointed by the active nodes list, blue nodes are closed nodes and the leaf with a bold border is the leaf currently being processed. The dotted black nodes and connection are not known, but are shown as a reference of what the final tree should look like.

In Figure 15a, the processed leaf has key 10 and key-height 0. Since key-height is 0, there is no subtree to connect. The leaf is placed as active at height 0.

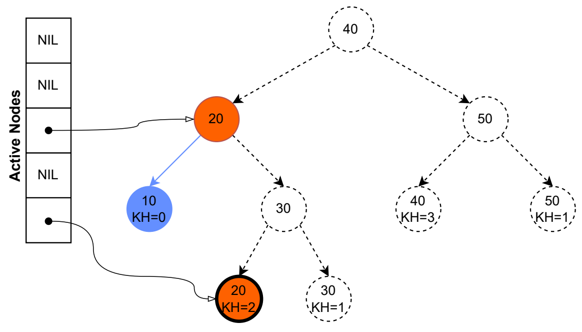
In Figure 15b, the processed leaf has key 20 and key-height 2. The left subtree of the new inner node, which is added as active node, can be generated by connecting all the active nodes in the list, from height 1 to height 0. There is no active node at height 1, therefore the next active level is checked, where the leaf with key 10 is found.

In Figure 15c, the processed leaf has key 30 and key-height 1. Again, an inner node is created and set active for that height and its left subtree can be generated by connecting all lower active nodes. In this case only the prevously processed leaf is found.

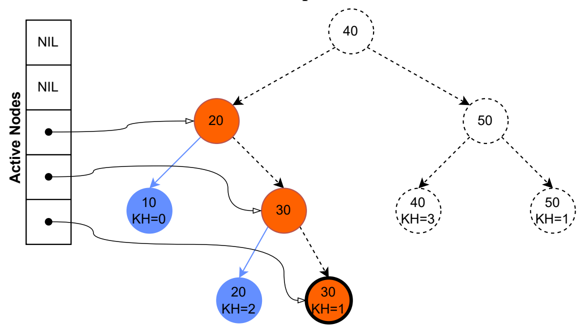
In Figure 15d, the processed leaf has key 40 and key-height 3. A new inner node is created and set as active at height 3, and all the active nodes at a lower height are connected, forming its subtree. At this point the left subtree of the tree's root is completed. The root node and corresponding leaf remain active for the following steps, which are not shown.



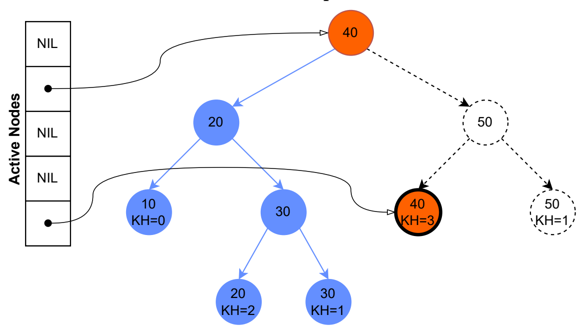
(a) First leaf processed



(b) Second leaf processed



(c) Third leaf processed



(d) Fourth leaf processed

Figure 15. IAVL tree partial reconstruction

Correctness of the algorithm
Let's try to give an informal intuition for the correctness of this algorithm. We know that in an IAVL Tree for every leaf apart from the left-most one, its key is found in a certain inner node at a certain height. This inner node is a root for two subtrees and the key is the smallest key that is found in the right subtree. Whenever we process leaf i with key k_i, we place a new inner node at height h_i. We then know that the left subtree of the newly added node will be formed by smaller keys and all the inner nodes were formed by processing the previous leaves. Whenever we process a leaf we add a new inner node and we close its left subtree, but the new node stays active. This means that of those roots that are active, the left subtree is closed. Now, when we process leaf i, it means that we processed all leaves $0, ..., (i-1)$ and their roots can either be closed, hence both the left and right subtrees are closed, or they can be active, hence only the left subtree is closed. Closing the left subtree of the root of leaf i, means to close all those lower roots that remain active, effectively closing their right subtree, completing the whole branch. This reasoning can be proven with an induction. The reconstruction trivially holds for a tree of height 2, with two leaves and a root. Then take a tree of height 3. As the tree is balanced, the difference in height between the subtrees of the tree's root can be at most of 1. This means that the subtrees can only be of height 1 and 2, 2 and 1 or 2 and 2. When we process the leaf whose root is the tree's root, it means that its left subtree was either of height 1 or 2 and we know that it was reconstructed correctly, and by connecting it to the root, we compete the whole branch. Then the reasoning can hold for a generic tree of height h. This is an informal proof but it can help understand the reasoning behind the algorithm. The reconstruction algorithm was tested over various random trees that have then been rebuilt. The hash value at the root, that depends both on the content and shape of the tree, of the initial and the rebuilt tree, have been found to be matching.

Algorithm 4 IAVL reconstruction

```
 1: procedure IAVL-Rebuild(list: []*Node)
 2:     maxH ← maximal height of tree
 3:     activeNodes ← *Node[maxH]
 4:     for n ∈ list do          ▷ For each leaf, add its inner node and connect its left subtree
 5:         h ← n.keyHeight
 6:         if h ≠ 0 then
 7:             newNode ← newNode(n.key, n.keyHeight)
 8:         else
 9:             newNode ← n
10:         activeNodes[h] ← newNode
11:         for i,j ← 0, 1; j ≤ $h$ & i < h; i++ do
12:             if activeNodes[i] ≠ NIL then
13:                 for activeNodes[j] = NIL do
14:                     j++
15:                 if activeNodes[j].leftNode = NIL then
16:                     activeNodes[j].leftNode ← activeNodes[i]
17:                 else
18:                     activeNodes[j].rightNode ← activeNodes[i]
19:                 activeNodes[i] ← nil                          ▷ Close lower nodes
20:                 j++
21:         if h ≠ 0 then
22:             activeNodes[0] ← n          ▷ All leaves apart from the first, must be set active
23:
24:     for i, j ← 0, 1; j ≤ maxH;  do          ▷ Close the remaining nodes in the active list
25:         for activeNodes[j] = NIL do
26:             j++
27:         activeNodes[j].rightNode ← activeNodes[i]
28:         i ← j
29:         j++
```

4.6 B*Tree reconstruction

It must be possible to reconstruct a B*Tree starting from its leaf nodes, which should contain all the information needed for this process. In Section 3.1.3 we discussed the reconstruction of a B+Tree, where the left-most key in each leaf node is present somewhere in the inner structure of the tree at a certain height and we called it `keyHeight`. This information is needed for an exact reconstruction. With a B*Tree, we don't only need to know at what level the key is found, but also at what height within the inner IAVL tree that key is found, because we need to rebuild the B*Tree nodes as normal B+Tree nodes, however the keys are not simply put linearly one after the other inside an array, but they are put into the IAVL tree. The inner tree needs to have the exact shape for each peer, because it determines the IAVL root hash's value and therefore, the hash value of the whole B*Tree node.

Leaf Bnodes are obtained as a block, either from stable storage or from another peer, and we have all the keys of that specific leaf, therefore their inner IAVL tree can be reconstructed using the reconstruction algorithm (Algorithm 4) that requires a list of sorted IAVL leaf nodes as input. Inner Bnodes, however, are reconstructed key by key every time a leaf Bnode is processed, hence we do not have all the IAVL leaf nodes for a specific inner Bnode at the same time. For this reason the algorithm of an IAVL tree reconstruction (Algorithm 4) can be modified so that it can be executed leaf by leaf and the intermediate state is saved between each call of the procedure. It is enough to have a data structure that keeps track of the active nodes after the last call and the current root node of the partially rebuilt IAVL tree. Assuming that the caller will provide the leaf nodes in the correct order, which is the case during the reconstruction, the result is the same as providing the sorted list as a whole, with the advantage that progress in the rebuilding can be done each time a new leaf node is processed. Remember from 3.1.3 that every time a B+Tree leaf node is processed, a key is added to a partially rebuilt inner B+Tree node, and a whole subtree can be closed. In a B*Tree, every time a key is added to a partially rebuilt inner B*tree node, it corresponds to adding another IAVL tree leaf, meaning that the IAVL tree can have a progress to its reconstruction as well. When the B*Tree inner node is closed, all the keys have been put and the IAVL tree can also be finalised, by creating the connection with the nodes left active. In Algorithm 6 we define the two procedures `addLeaf` and `finalize`. The first procedure processes a leaf, making the IAVL reconstruction progress by one step, and saves the current state in a rebuilder structure, that will be used by following calls. The second procedure is called once all leaves have been processed, meaning that the IAVL Tree can be finalized by creating the last connection between the nodes left active in the rebuilder. Note that the first procedure is corresponds to the content of the for loop that processes the list of leaves in Algorithm 4, and the second procedure is corresponds to the last for loop in that algorithm. To reconstruct a B*Tree, we will follow the same idea shown in Algorithm 1 to reconstruct a B+Tree and additionally, for each active Bnode, we keep track of a rebuilder, that holds the current state of the partially rebuilt IAVL for that Bnode. Whenever a leaf Bnode is processed, we can add a key in an inner Bnode, which corresponds to adding that key to the partially rebuilt inner IAVL Tree, using the rebuilder and the `addLeaf` routine. Every time a key is added, we can close its left subtree, which is defined by all active nodes with a lower height. This is where we make use of the `rebuilder.secondLastAdded` and `rebuilder.lastAdded`. They represent the second-last and last leaf inserted in the partially rebuilt IAVL Tree. To decide how to connect the subtree, after a key is added at height j, we refer to Algorithm 5.

Algorithm 5 Connect active nodes to form a subtree

```
1: procedure ConnectSubtree(builders []*IAVLRebuilder, activeNodes []*Node, j int)
2:     for ;$j > 0$;$j--$ do
3:         parent ← activeNodes[j]
4:         child ← activeNodes[j-1]
5:         builderParent ← builders[j]
6:         builderChild ← builder[j-1]
7:         if builderParent.secondLastAdded = NIL then
8:             builderParent.lastAdded.leftBnode ← child
9:         else
10:            if builderParent.secondLastAdded.rightBnode ≠ NIL then
11:                builderParent.lastAdded.rightBnode ← child
12:            else
13:                builderParent.secondLastAdded.rightBnode ← child
14:        child.parent = parent
15:        if $j - 1 > 0$ then
16:            child.tree ← FINALIZE(builderChild)
```

Algorithm 6 Partial IAVL reconstruction

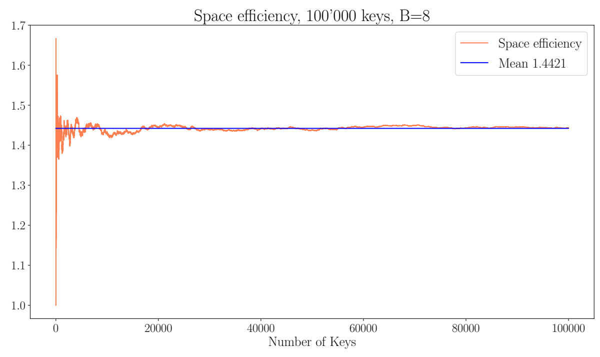
```
procedure ADDLEAF(rebuilder *IAVLRebuilder, l *Node)
    if rebuilder.leftMost = NIL then
        rebuilder.leftMost ← l
                        ▷ The following values are used to rebuild the whole B*Tree
    rebuilder.secondLastAdded ← rebuilder.lastAdded
    rebuilder.lastAdded ← l

    h ← l.keyHeight
    if h ≠ 0 then
        newNode ← newNode(l.key, l.keyHeight)
    else
        newNode ← l
    rebuilder.activeNodes[h] ← newNode
    for i,j ← 0, 1; j ≤ $h$ & i < h; i++ do
        if rebuilder.activeNodes[i] ≠ NIL then
            for rebuilder.activeNodes[j] = NIL do
                $j \leftarrow j + 1$
            if rebuilder.activeNodes[j].leftNode = NIL then
                rebuilder.activeNodes[j].leftNode ← rebuilder.activeNodes[i]
            else
                rebuilder.activeNodes[j].rightNode ← rebuilder.activeNodes[i]
            rebuilder.activeNodes[i] ← nil                    ▷ Close lower nodes
            $j \leftarrow j + 1$
    if h ≠ 0 then
        rebuilder.activeNodes[0] = l
    if rebuilder.maxH ≤ h then
        rebuilder.root = rebuilder.activeNodes[h]
        rebuilder.maxH = h

procedure FINALIZE(rebuilder *IAVLRebuilder)
    for i,j ← 0,1; j ≤ rebuilder.maxH do
        while rebuilder.activeNodes[j] = NIL do
            $j \leftarrow j + 1$
        rebuilder.activeNodes[j].rightNode ← rebuilder.activeNodes[i]
        $i \leftarrow j$
        $j \leftarrow j + 1$
    return rebuilder.root
```

5 Results

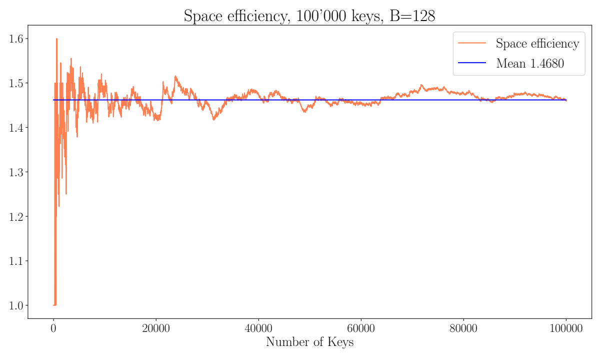
In this section, we experimentally verify some properties of the data structures created, with the goal of quantifying their benefits and trade-offs. All experiments have been run on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 and 16GB of memory, running MacOS Catalina 10.15.6.

5.1 Space Efficiency

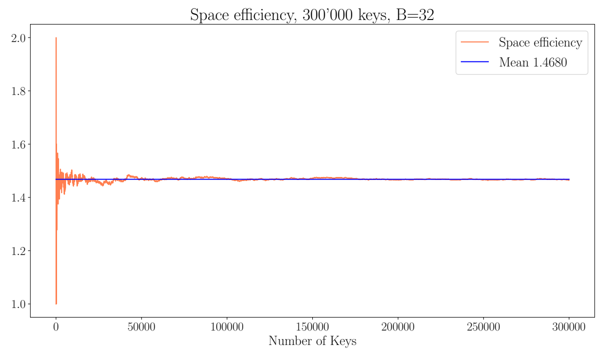
We define the *space efficiency* as the number of leaves in the tree divided by the ideal number of fixed-size chunks for the amount of values contained in the tree. A space efficiency of 1 represents the perfect scenario, and a space efficiency of 2 means that there are twice as many leaves as the theoretical number of chunks needed. Assuming the naive assignment of nodes to chunks following a tree traversal, we would obtain a perfect space efficiency. When leaves are assigned to chunks, space efficiency becomes a trade-off, in exchange of concurrent Merkle proofs. We are interested in this metric because it reflects the amount of chunks that a recovering peer needs to request in order to obtain the working state, which directly affects the time taken for state synchronization.



(a)



(b)



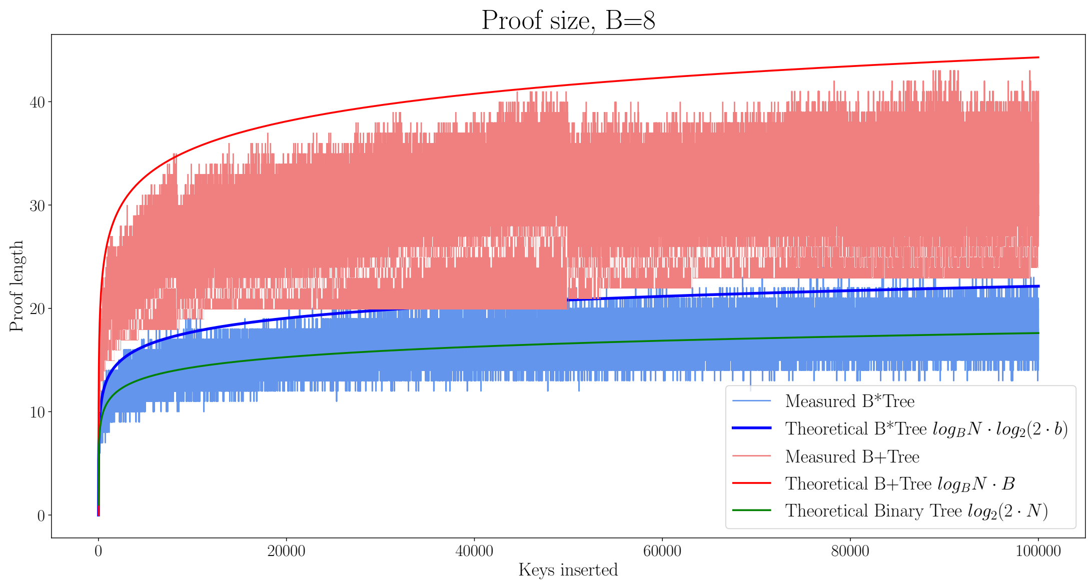
(c)

Figure 16. Space efficiency for a B*Tree

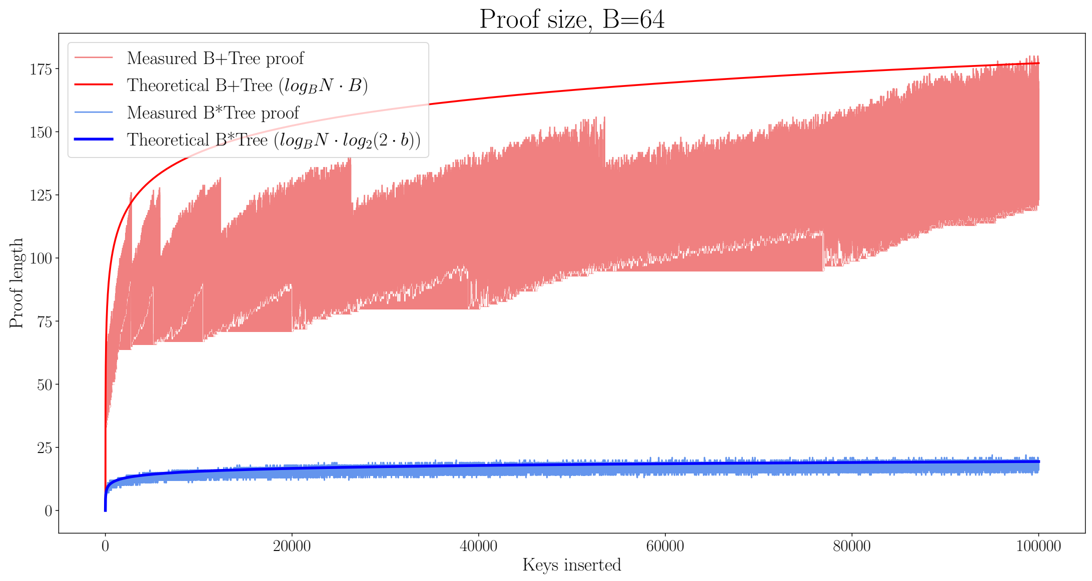
Figure 16 shows how the space efficiency develops over time, as keys are inserted into a B*Tree. The keys were randomly shuffled and after each insertion, the efficiency has been measured. In Figure 16a and 16b we show both an example with a branching factor of 8 and 128. Both curves behave similarly; after a period of instability, the efficiency settles down to a constant value, at around 1.45. This value reflects the invariant of a classical B+Tree, which ensures that leaves can never have less keys than half their capacity. A space efficiency of 1 is reached when all leaves are full, while a space efficiency of 2 is reached when all leaves are half-full. However in a B*Tree, leaves are split by removing the root node of the inner IAVL tree, resulting in two unbalanced halves. This is not a problem because while the smaller half will have a lower space efficiency, the bigger half will have a higher efficiency. In Figure 16c we show yet another example for 300’000 keys and a branching factor of 32. We can see that the space efficiency reached a value of 2 at the beginning, but still settled at around 1.45 overtime.

5.2 Proofs Size

To prove the inclusion of an element in a merkle tree, a sequence of hashes has to be provided, so that together with the element to be proven, the hash value found at the root of the tree can be recomputed. The length of this proof depends on the type of tree and is provided by traversing the tree looking for the element to be proven. For each node traversed, the hash value of its siblings has to be provided. In a binary merkle tree, each node has a sibling. Therefore the length of the proof is bounded by the height of the tree, which is in the order of $\mathcal{O}(\log_2(2n)$, since there are n leaves and $2n-1$ nodes in total, where n is the number of values. In a B+Tree, the number of siblings of each node is in the worst case $B-1$, when all nodes are full. The size of the proof is then in the order of $\mathcal{O}\log_B(n) \cdot B$. As discussed in Section 4, a B*Tree should provide a smaller proof, that is in the order of the one of a binary tree. To generate Figure 17 we inserted in both a B+Tree and B*Tree 100’000 keys in a random order. After each insertion the length of the proof for the last added element was measured. Figure 17a shows the results in the case of a very small branching factor, where the proofs for a B*Tree can be very close in size to those of a binary tree. Figure 17b shows the results for a higher branching factor, where the linear term affects a lot more the proof length in the case of the B+Tree, while the proof length for a B*Tree remains a lot smaller.



(a) Branching factor: 8



(b) Branching factor: 64

Figure 17. Comparing the proof size of a B+Tree and a B*Tree.

5.3 Reconstruction

To asses the performance of the rebuilding algorithms we considered random trees with 300'000 keys and a branching factor $B = 32$. We compare the time it takes to insert all the keys one by one as opposed to rebuilding the tree starting from the leaves. To this end, we assume that leaves have already been verified and are ready to be used by the rebuilding routine. With a B*Tree, we obtained an average of $4494.8135ms$ to complete all single insertions, while the rebuilding took an average of $85.6739ms$. Using a B+Tree we obtained an average of $3584.1169ms$ for the single insertions and an average of $21.3241ms$ for the rebuilding.

Tree	Single insertions (*ms*)	Rebuilding (*ms*)	Speedup
B*Tree	4494.8135	85.6739	52.46 times
B+Tree	3584.1169	21.3241	168.08 times

Table 1. Time taken to insert 300'000 single values in a tree with the hash values updated after each insertion, compared with the time taken to rebuild the tree, assuming the leaves were verified.

The values are reported in Table 1 which also shows how much faster the rebuilding routine performs. We called this value *speedup* and it is the ratio of the two average values. We can clearly see that in both cases rebuilding is more efficient than executing single insertions. This is because leaves allow to process many K-V pairs at once; for each leaf which represents 32 pairs, only the left-most key has to be inserted in the inner structure of the tree. Furthermore, the single insertion routine updates the hash values of the tree after every single insertion. Most importantly we can notice that rebuilding a simple B+Tree results in a greater speedup compared to a B*Tree. Inserting a key in an inner node during the rebuilding procedure is as simple as adding a value at the end of an the array in case of a B+Tree, however, when using a B*Tree, the key has to first be inserted in a IAVL Tree which needs to be balanced after each insertion.

Tree	Single insertions (*ms*)	Rebuilding (*ms*)	Speedup
B*Tree	1328.0658	82.0645	16.18 times
B+Tree	1003.3739	19.5426	51.34 times

Table 2. Time taken to insert 300'000 single values in a tree with the hash value computed once at the end compared with the time taken to rebuild the tree, assuming the leaves were verified.

We are interested in understanding how much of the single insertion time is used by hash computations. For this reason, we conducted the same experiment by removing the hash computations after each insertion and executed the hashing once on the whole tree after the last insertion. In Table 2 we can notice that the insertion time has decreased to 1.328 *s* for the B*Tree and 1.003 *s* for the B+Tree on average. This means that hashing only at the end results in the two data structures to perform more than 3 times better. This shows that depending on the need of the application, hashing only after a certain number of insertions (or when needed) can be beneficial option even if the whole tree is rehashed completely. A more sophisticated algorithm could keep track of which nodes have their hash value up-to-date because since the previous hashing procedure they have not been affected by any change and the hash computation for their subtrees can be avoided. As expected, the rebuilding time has not changed much. The small variation can be explained by the fact that the experiments have been done on two different days and the machine might have had a slightly different load. Clearly the speedup obtained by the rebuilding decreases as a direct consequence of the absence of the hash computations at each insertion. However their ratio stayed rather similar, in both cases the speedup of rebuilding a B+Tree is around 3 times faster than the speedup of rebuilding a B*Tree, since $\frac{51.34}{16.18} \approx \frac{168.08}{52.46} \approx 3.2$.

5.4 Insertion

We measured the time required for a single insertion over time in a B*Tree, as the tree grows. We considered 400′000 insertions of 1kB values on a tree with a branching factor of 32. Among the insertion time values there were many outliers found at what we think are random points during the insertion, not linked to splits in the tree. We can guess that the big variance in these values can be associated with the underlying CPU scheduler of the system. The trend of the values suggested a slow growth in the time necessary for single insertions, which we can associate with the tree getting deeper. In Figure 18 we present a cumulative distribution function (CDF) of the insertion times. The data has been rounded up to be grouped into several classes, which represent a certain percentage of the total measurements and the CDF for these classes has been computed. It can be seen that 98.1% of the insertions took 40μs or less and 99.9% of the insertions took 80μs or less.

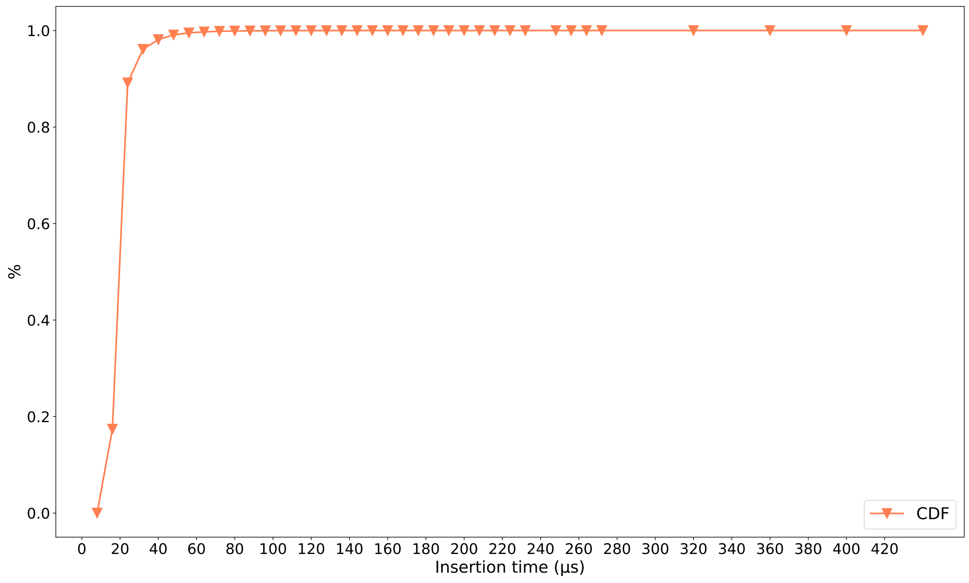


Figure 18. Cumulative distribution function of the insertion time in a B*Tree, for 400′000 keys and values of 1kB and a branching factor of 32.

6 Conclusions and future work

In this project, we address the problem of scalability in blockchain systems regarding the time needed for state synchronization, where new peers want to join the blockchain and need to catch up with the current state of the system. They do so by requesting other peers to provide them the necessary data that needs to be verified for validity. The state of the system can be organized into snapshots and each snapshot broken down into several chunks. Chunking the data allows to obtain large amount of data at once. If the state is represented in the form of a binary Merkle tree, the chunks are formed by a fixed number of nodes of the tree and can be assigned in ways that allow the verification of validity to happen concurrently and without having to reconstruct the whole state first. One of these methods is to assign only leaf nodes belonging to a certain subtree to a specific chunk. We instead try a novel method by representing the state in the form of a B+Tree, which naturally groups leaves into blocks. However, the size of Merkle proofs in the case of a B+Tree has a linear term for each node that is part of the proof which makes them less attractive when compared to binary trees. For this reason we explored the possibility of merging the advantages of both B+Trees and binary trees into a novel data structure called a B*Tree, where nodes contain self balancing merkle trees instead of arrays of keys and pointers, with the goal of reducing the size of the proofs. Additionally, updating hash values along a path has a logarithmic cost for each node on the path, while a B+Tree requires a concatenation of values to be hashed in order to update each node on the path. Storing only the leaves of these trees required the development of algorithms capable of reconstructing the trees in a deterministic way. Such algorithms have been developed for a merklized B+Tree, an IAVL tree and the B*Tree. The algorithms for a B+Tree and B*Tree have been proven to provide a speed up compared to inserting single elements, as whole leaf nodes can be processed at each step of the procedure. We then showed that updating the necessary hash values of the tree after each insertion slows down the insertion algorithm by a considerable amount and it could be beneficial to only updates the values after a certain number of insertions or on demand. Furthermore, we demonstrated that a B*Tree maintains a good space efficiency with an expected value of around 1.45, even if nodes are not split exactly in half. We then assessed experimentally that the proof sizes in a B+Tree tend to grow rather large as the branching factor of the nodes grows, while a B*Tree offers a valid solution having proof sizes closely bound to the size that a normal binary tree would have.
An important next step for this project would be to embed the B*Tree within a working system to assess the performance, and decide if it brings advantages compared to binary versions of Merkle Trees. To this end, it would be necessary to make the tree multi-versioned using a copy on write strategy, where new versions of the tree keep pointers to nodes belonging to older versions, as long as they are left unchanged.

Furthermore, during the insertion routine of both the B*Tree and B+Tree, as the values of `keyHeight` and `kIAVLHeight` change, paths in the tree must have their hash values updated. In case of recursive splits, it is possible that some nodes (the whole Bnode in the case of a B+Tree and a path inside the inner IAVL Tree in the case of a B*Tree) are updated just to get a new hash value overwritten in a later step because they belong to another path that needs to be updated. This happens for the nodes that are common ancestors of nodes along paths that need to be updated, because the two paths will meet at at least one common node, which will get its hash value recomputed twice. It could be interesting to determine whether this situation arises often and whether it poses a real performance issue. If this were the case, the hashing procedure after each insertion can be changed to be a recursive top-down traversal. After the insertion is finished, nodes that have been affected by the insertion, from the root down to the various leaves that changed their content will have a boolean flag stating the need to update their hash values. A recursive top-down approach will exactly update the hash values once for each node. This method would also allow for hash values to be updated only on demand or after a certain number of insertions.

References

[1] Cosmos. Iavl. https://github.com/cosmos/iavl, 2022.

[2] E. Fynn. Scaling blockchains. https://www.inf.usi.ch/faculty/pedone/PhDThesis/enrique.pdf, 2021.

[3] Golang. crypto/sha256. https://pkg.go.dev/crypto/sha256.

[4] Google. Language guide (proto3). https://developers.google.com/protocol-buffers/docs/proto3.

[5] Tendermint. go-amino. https://github.com/tendermint/go-amino.

[6] Tendermint. Tendermint core. https://github.com/tendermint/tendermint, 2022.