Implementing a Hash Array Mapped Trie

Peter Zhao, David J. Lee, Andrew Thai
CS 333: Storage Systems

May 22, 2021

Introduction

We used Rust [1] to implement the Hash Array Mapped Trie (HAMT) [2], a wide and shallow trie-based dictionary. The HAMT is often used in functional programming languages like Haskell or Clojure. Our goals in pursuing this project were twofold:

- We wanted to learn about and implement a tree-based data structure that uses hashing.
- We wanted to learn Rust, a modern systems programming language, and explore the language's memory management model. In particular, we wanted to see how Rust facilitates programming data structures that rely on pointers.

This report presents a simple introduction to the HAMT. We describe its general structure and core operations. This report is also supplemented by a tested Rust implementation of the HAMT.

Data Structure

We studied and implemented the HAMT, a trie-based dictionary data structure.

Tries, or prefix trees, are tree data structures that store elements as paths from root to leaf. A single element $w = s_1...s_k$ corresponds to a path from root to leaf $P = n_1...n_k$ where the i-th node n_i contains the symbol s_i. This makes tries compact, because symbols that appear in multiple elements are only represented once.

The HAMT adapts this idea to key-value pairs. Instead of associating elements with root-to-leaf paths, the HAMT hashes the key of each key-value pair and associates the hash with a root-to-leaf path. Additionally, instead of associating each node with a single symbol from the hash, the HAMT associates each node with a fixed number of symbols from the hash (typically, 5 if the underlying architecture is a 32-bit system, or 6 if it is a 64-bit system). In the following discussion, we will assume a 64-bit architecture; therefore, each node will be associated with 6 bits.

With this setup, each node can have at most 64 children where a child may be either a single key-value pair, or another node containing more children.[1] This is a very high branching factor, making the HAMT shallow and wide. With a good hash function that distributes elements evenly, a tree of only two levels can store over 4000 elements. Because of its shallow depth, a HAMT offers very fast lookups, insertions, and deletions.

Because they are tree-based, HAMTs can be easily extended to be *persistent*. Persistence can be achieved by path copying, which works for trees but not arrays. This easy persistence is what makes HAMTs useful for functional programming languages.

We now describe the HAMT's three core operations: queries, insertions, and deletions.

Queries

To check whether a HAMT contains a key k, we first hash k, yielding $h(k)$. We take the first 6 bits of $h(k)$ and examine the root node's children. If the child at position $h(k)$ is a key-value pair (k', v'), we have two options. If the key k' matches the key k, we return the the value v'. Otherwise, we return `None` to indicate that the key does not exist in the HAMT. If the child is a subtree, we recursively query that subtree for the key k.

Insertions

To insert a key-value pair (k, v), we again hash k, yielding $h(k)$. Then, we check if the root's children contains a child at position $h(k)$. If it is empty, then we can immediately insert the (k, v) pair into that position. If there is another key-value pair at that position, then we want to construct a subtree at that index, look at the next 6 bits of both the original element and our current element, and insert them both into this subtree. Otherwise, if there already is a tree, then we simply insert the element by looking at the next 6 bits, and repeating the insert process.

Deletions

To remove a key-value pair (k, v), we first perform the same steps as a query to determine whether the key-value pair exists. If so, we simply remove it from its parent's vector of

[1]We assume that we can generate an arbitrarily long hash for each key. This is easy to do in practice by rehashing the same key with a different seed. Using a hash function with a 128-bit output, we get up to 21 6-bit pieces of the hash before we need to rehash.

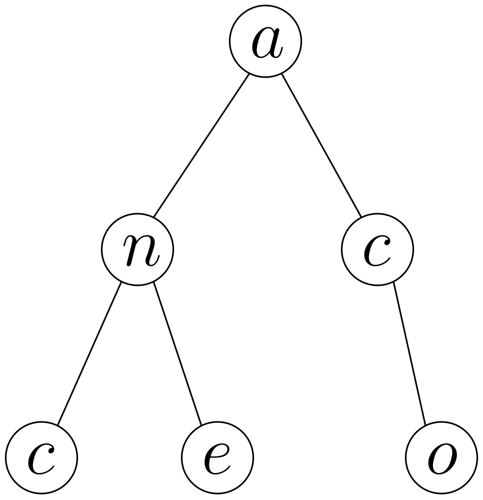

Figure 1: An example trie representing the set $\{anc, ane, aco\}$. Note that each letter, while potentially repeated across words, is only stored once in the trie.

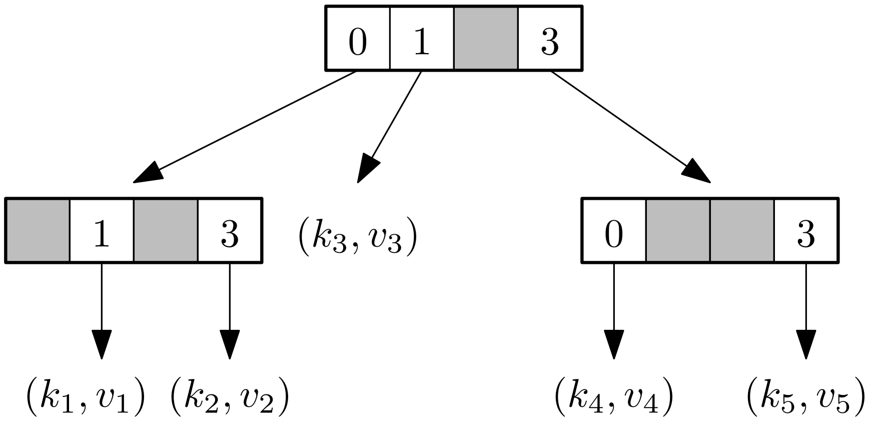

Figure 2: An example HAMT storing the set $\{(k_1, v_1), (k_2, v_2), ..., (k_5, v_5)\}$. We can infer a prefix of each key-value pair's hash. For example, `0x10` $\in \text{pre}(h(k_1))$, `0x30` $\in \text{pre}(h(k_2))$, and `0x1` $\in \text{pre}(h(k_3))$. Shaded cells denote absent children.

children. However, we must also deal with the case where there are only two nodes in a subtree and we remove one of those nodes. To better utilize space, the node that would be kept needs to be merged up the tree since a subtree with a single node is a redundant representation of just that single node.

Optimizations

To store all the children, we can naively allocate an entire array for all possible elements. However, tries are generally sparse and a lot of these entries may be empty, so this requires allocation of a lot of unused space. Instead, a vector is used to store children of nodes, which allows us to store a compact representation without having to allocate an entire array, and uses a 64 bit bitmap to tell us where elements go based on the hash.

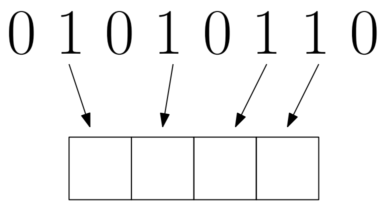

Figure 3: A node's bitmap and its associated vector

This vector stores elements in their relative locations in the bitmap. That is, the set bits in the bitmap reflect the actual location of the elements if they were to be stored in a 64-bit array. Thus, we need a way to determine the relative index in the vector. To do this, we use `popcount` which provides us the relative index in the vector given a bitmap. Since `popcount` is a common operation and designed to be extremely fast, this indirection does not have much overhead.

Results

We've tested all three operations, and we are 95% confident in its correctness. Unfortunately, due to time constraints, we didn't get to pit it again other implementations. Maybe later!

References

[1] The Rust programming language. https://doc.rust-lang.org/book/. Accessed: 2021-05-22.

[2] Bagwell, P. Ideal hash trees. Tech. rep., 2001.