An Interactive Tutorial on Hash Array Mapped Tries

Ambrose Bonnaire-Sergeant

ambrose@ambrosebs.com

Abstract

Clojure provides a suite of persistent data structures implemented by Hickey based on previous work by Bagwell. This tutorial teaches how Hash Array Mapped Tries work using a Clojure port of Hickey's Java implementation.

1. Introduction

Hash Array Mapped Tries (HAMT) provide a key tool for functional programmers as fast, immutable, and persistent alternatives to hash maps. First described by Bagwell [2], they are featured in mainstream functional programming languages like Clojure and Scala, and have been ported to many others.

HAMT's compare very well to other similar data structures. They offer fast lookups by minimizing the depth and branching factors of their tree representation. They minimize memory usage by lazily creating subtries only when necessary.

Clojure is a dynamically typed programming language running on the JVM. It comes with a suite of persistent data structures that form the core of the language. These data structures are based on Array Mapped Tries [1], and include persistent vectors, hash maps, and hash sets.

To understand with a hash array mapped trie is, we first give some definitions. A trie is a way of formatting key/value pairs in a tree, where values are leaves and keys are spread across the paths to those nodes. Key prefixes occur on the shallow levels of the tree, and suffixes occur closer to the leaves. A bit trie assumes the mapping keys are strings of bits. Each level consumes one or more bits to index its elements. An Array Mapped Trie maps the bits of array indices as a bit trie.

In this paper, we explore Clojure's persistent hash map implementation (Section 3). It was implemented by Hickey [3], extending Bagwell's original formulation [2] to be persistent. Persistent data structures are extended using structural sharing, so Clojure necessarily enforces hash maps to be immutable.

Contributions

1. We walk through the mechanics behind HAMTs.

2. We describe the internals of Clojure's persistent HAMT implementation.

3. We present a port of Clojure's HAMT from Java to Clojure for pedagogical purposes, featuring trie visualization (Figure).

2. Walkthrough

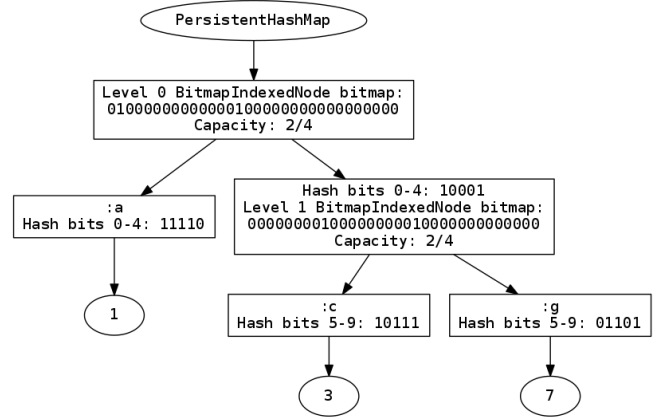Let's walk through some examples about how HAMTs work under different operations.

Figure 1. A hash array mapped trie `{:a 1, :c 3, :g 7}`. The first 5 bits of `:c`'s and `:g`'s hashes collide (`10001`), so a new level is used to disambiguate.

Firstly, a HAMT represents a search tree based on the hash of its keys. Each key is associated with a value. Figure 2 gives sample 32-bit hashes for six keys, which we will use only in this section.

A HAMT starts as an empty tree with no root.

Insertion   Let's insert a mapping into an empty tree. Since the tree is empty, we create the first (root) level, level 0. This corresponds to the first 5 bits of the hash. The maximum branching factor is $2^5 = 32$, but since we only need one entry, we create a resizable root node.

A resizable node of current capacity n entries, contains an array of length $2n$ and a 32-bit bitmap indicating the location of each entry. Initially, the entry array is of length zero, and the bitmap is zero.

To add a mapping from `:a` to `1`, we first retrieve the first 5 bits (since we are current at level 0) of `hash(:a)`—`00001`, or 1 in decimal. So, the second bit in the bitmap is set to 1. (Red text indicates the results of the current operation).

`0000 0000 0000 0000 0000 0000 0000 0010`
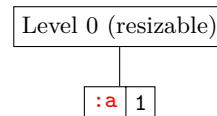
Since there is only one entry, we allocate a length 2 array, and initialize it with the key and value.

Adding another entry reveals a crucial invariant of resizable nodes: the array index of an entry corresponding to bit
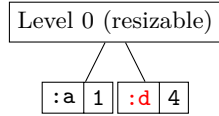
```
             6     5     4     3     2     1     0
hash(:a) = 00 00000 00000 00000 00000 00000 00001
hash(:b) = 00 00000 00000 00000 00000 00000 00011
hash(:c) = 00 00000 00000 00000 00000 00000 00101
hash(:d) = 00 00000 00000 00000 00000 00000 01010
hash(:e) = 00 00000 00000 00000 00000 00000 10100
hash(:f) = 00 00000 00000 00000 00000 00001 10100
```

Figure 2. Example 32-bit binary hashes for six keys, partitioned into 6 groups.

i in the bitmap, is the sum of all the 1 bits below bit i, multiplied by 2.

We add a mapping from `:d` to `4`. The first 5 bits of its hash is 01010, 10 in decimal. so we set the 11th bit of the bitmap to 1.

`0000 0000 0000 0000 0000 0100 0000 0010`

Does the new node go before or after `:a`? It is decided by counting the number of 1's below the 11th bit—assigning the result to i—and inserting the entry into the array after skipping i nodes. Here, the sum is 1, so we insert `:d` after visiting 1 node. The actual index to insert the new key and value is $2i$ and $2i + 1$ respectively.

We repeat this with a few more keys—we leave the intermediate stages as an exercise for the reader.

`0000 0000 0001 0000 0000 0100 0010 1010`
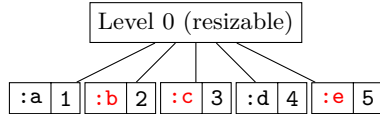
Inserting new levels   We must deal with the case of hashes clashing at a particular level. Notice `hash(:e)` and `hash(:f)` both have 10100 for their level 0 hash. If the hashed values were actually identical, we would simply swap out the old mapped value with the new mapped value. Since they are actually different keys, we must create a new level and compare them on the next 5 bits of their hashes.
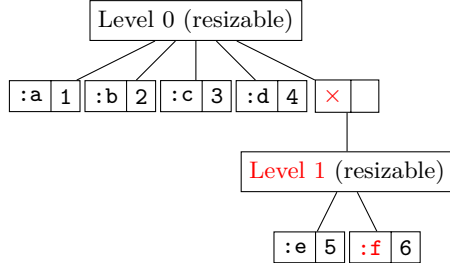
The new level 1 node has its own bitmap based on the 6th-10th bits of hashes. Currently, the bitmap's 1st and 2nd bits are set to 1, since the level 1 hashes of `:e` and `:f` in decimal are 0 and 1 respectively. Since the 2nd bit has a single 1 bit below it in the bitmap, `:f` goes after `:e`.

We can also see how resizable nodes avoid some indirection—if an entry is unambiguous at a level, it is inserted directly into the array; otherwise, a × indicates the next array member is pointer to a subtrie to continue searching.

Search   To perform a lookup on a given key, we use each level of its hash to traverse the tree, until the tree ends, or we find the entry.

For example, let's lookup the entry for `:e` in the previous trie.

At level 0, its hash is 10100, or 20 in decimal. We lookup the 21st bit in the root bitmap (in red) and it is 1—this indicates the entry exists. We count four 1 bits below it (in blue).

`0000 0000 0001 0000 0000 0100 0010 1010`

We lookup entry $2 \cdot 4 = 8$ in the root array, and find an ×. We follow the pointer to level 1, and repeat for the next 5 hash bits (00000; 0 in decimal) with the next node's bitmap:

`0000 0000 0000 0000 0000 0000 0000 0011`

We find a 1 in the 1st bit position, and zero 1's below it, so we lookup entry $2 \cdot 0 = 0$ in the level 1 array. Since it is not ×, we test to see if the entry we have travelled to `:e` is equals to our query—it is, so the search has succeeded and the mapped value is at index $2 \cdot 0 + 1 = 1$; otherwise the search fails.

Deletion   Node deletion follows the same procedure as search to discover the location of an entry. Instead of returning the mapped value, it removes the entry from the array, and sets the bitmap for that entry to 0.

Full nodes   Once a resizable array reaches a certain threshold size, it is no longer efficient to allocate an array of length $2n$ to hold n nodes. For example, if we have over 16 entries, which would require copying arrays over length 32, we could instead once-and-for-all allocate a length 32 array where each member is a subtrie (without a × flag)—we call this a full node.

This removes the need to bitmap bits—the 32 bitmap bits now map one-to-one to the subtries.

Hash collision nodes   If two different keys hash to the same value, we use a hash collision node to differentiate them. One approach is to default to a linear search—with the assumption that the hash function has a low probabilities of collisions, this should rarely be an issue.

3.   Clojure's HAMT, in detail

Now, we will give a detailed description of Hickey's HAMT implementation for Clojure, based on Bagwell's original HAMT.

3.1   Understanding the bit operations

The core of the HAMT implementation has 3 important bit operations, which we will cover in detail.

Bit masking   The `mask` function isolates a multiple of 5 bits from a given hash, interpreting them as a 32-bit integer.

We can view hashes as 5 partitions by of a 32-bit integer from right-to-left.

```
6     5     4     3     2     1     0
00 00000 00000 00000 00000 00000 00000
```

The source for the mask function (below), selects partitions by shifting the hash's bits right until the first 5 bits is the partition needed, then the first 5 bits of the result interpreted as an integer (Appendix A explains `>>>`).

```
// returns integer between 0 and 31
int mask(int hash, int shift) {
  return (hash >>> shift) & 0x01f;
}
```

The return value of `mask` is an integer value from 0 to 31 which indicates which bit in the bitmap is relevant (in `ArrayNode`, it is the actual array index, `BitmapIndexedNode` requires you to count the number of 1's below this index in the bitmap, as described below).

For example, to isolate partition 0 from a hash, we call `mask(hash, 0)`, which simply isolates the first 5 bits (`0x01f` is `11111` in binary). To isolate partition 3 from a hash, we call `mask(hash, 3*5)` which shifts the hash right 15 places, then isolates the first 5 bits of the result.

The Clojure implementation has `shift` as a multiple of 5—instead of the number of times to multiply by 5—presumably for performance reasons. We conjecture that it is faster to pay an incremental cost of adding 5 every time you descend one layer, instead of an extra multiplication every time `mask` is called.

Isolating bits Each node maintains a 32-bit bitmap, which contains a 1 where the node's array has an entry.

To index into this bitmap, we use the `bitpos` function. The return value is a 32-bit integer with exactly 1 bit set to 1.

```
// returns a 32-bit integer with exactly 1 bit
// set to 1
int bitpos(int hash, int shift) {
  return 1 << mask(hash, shift);
}
```

The return value can then be used, combined with the bitmap using bitwise AND, to return the value of the desired bit in the bitmap.

Array indexing For non-resizable nodes (like `ArrayNode`), indexing into the next level of the trie is the result of a `mask`. Resizable nodes (like `BitmapIndexedNode`) are more involved, and require further bit manipulations.

To retrieve the next array index, we count the number of 1's below the given bit in the bitmap (assuming the given bit is set to 1). This number i is the number of nodes before the node of interest—thus indexes $2i$ and $2i + 1$ contain the key and value of interest. To demonstrate this, say we have a bitmap with the first 8 bits 1011 0010, and otherwise zeros. Since four bits are 1, we have four nodes in an array of size $4 \cdot 2 = 8$.

0	:a	1
2	:b	2
4	:c	3
6	:e	5

To lookup the fourth node with key `:e`, `mask` would return the binary number 1000 0000. Since the 8th bit is set to 1 in our bitmap, we count the number of 1's below the 8th bit—there are 3—so the desired key and value are at indexes $2 \cdot 3 = 6$ and $2 \cdot 3 + 1 = 7$, respectively.

This approach also holds when inserting a new node. Say we are inserting a new entry for the 7th bit in the bitmap. Since there are 3 existing entries below the 7th bit in the bitmap, we insert the new entry in the 4th position of the array, moving the existing entry to the 5th position.

0	:a	1				0	:a	1
2	:b	2				2	:b	2
4	:c	3				4	:c	3
⋮			:d	4		6	:d	4
6	:e	5				8	:e	5

Now, updating the bit map to 1111 0010 with the 7th bit set to 1 keeps the bit counting invariant—now the 8th entry is after the 4th entry because there are 4 bits set to 1 before the 8th bitmap bit.

The bit counting is calculated by the `index` function.

```
// returns the number of 1's in bitmap before the
// `bit`th bit
int index(int bit, int bitmap) {
  return Integer.bitCount(bitmap & (bit - 1));
}
```

In the implementation, `bit` is a 32-bit integer with exactly 1 bit set to 1. By decrementing `bit`, we aquire a useful mask to isolate all the necessary bits in the bitmap. For example, if bitmap was `1000`—that is, isolating the 4th bit—decrementing it results in `0111`. Combining `0111` and the bitmap with bitwise AND then isolates the 1st-3rd bits, which we can then use to count the number of 1's below `bit` in `bitmap`.

3.2 Memory layout

The HAMT data structure has a very compact representation that dynamically expands to make room for 0-16 keys at each level. For 17-32 keys, a full array for 32 keys is allocated once and does not expand further.

For presentational purposes,

PersistentHashMap The enclosing class for the Clojure HAMT is the `PersistentHashMap`, which we briefly summarize. It contains a nullable root node, and has a special field to store `null` entries in the map—INodes use `null` as an indicator.

It provides three methods: `assoc` to associate a new key/value pair, `dissoc` to dissociate an entry by its key, and `find` locate an entry by its key.

INode There are various kinds of nodes in `PersistentHashMap`, all implementing the `INode` interface. It provides a similar interface as above.

```
  Bitmap
00000000 00000000 00000000 00000000
```

BitmapIndexedNode The `BitmapIndexedNode` is the meat of the HAMT data structure. It maintains an expandable array for storing up to 16 nodes, where even array indices i contain keys and $i + 1$ have their associated value or subtrie. The bitmap is a 32-bit integer that indexes into the array via some bit manipulation and class invariants.

For sub-tries with n nodes, where $n \leq 16$, we only need to allocate an array of size $2 * n$. This is where the trick of counting number of 1's below the current bit comes into play. This number tells us where to index into this array to find the node in question.
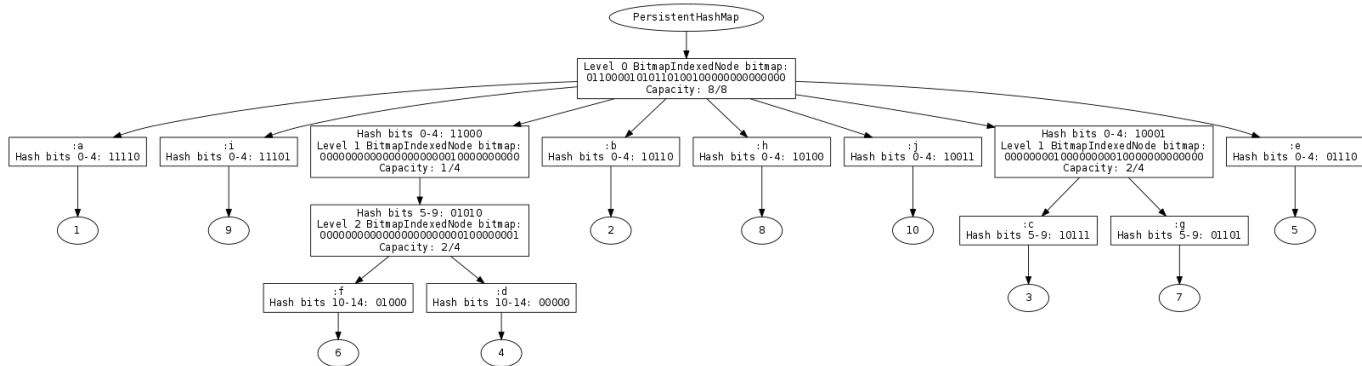
Figure 3 (trie visualization):

```
PersistentHashMap
  └─ Level 0 BitmapIndexedNode bitmap:
     01100001010110100100000000000000
     Capacity: 8/8
       ├─ :a  Hash bits 0-4: 11110 → 1
       ├─ :i  Hash bits 0-4: 11101 → 9
       ├─ Hash bits 0-4: 11000
       │  Level 1 BitmapIndexedNode bitmap:
       │  00000000000000000001000000000000
       │  Capacity: 1/4
       │    └─ Hash bits 5-9: 01010
       │       Level 2 BitmapIndexedNode bitmap:
       │       00000000000000000000000100000001
       │       Capacity: 2/4
       │         ├─ :f  Hash bits 10-14: 01000 → 6
       │         └─ :d  Hash bits 10-14: 00000 → 4
       ├─ :b  Hash bits 0-4: 10110 → 2
       ├─ :h  Hash bits 0-4: 10100 → 8
       ├─ :j  Hash bits 0-4: 10011 → 10
       ├─ Hash bits 0-4: 10001
       │  Level 1 BitmapIndexedNode bitmap:
       │  00000000100000000010000000000000
       │  Capacity: 2/4
       │    ├─ :c  Hash bits 5-9: 10111 → 3
       │    └─ :g  Hash bits 5-9: 01101 → 7
       └─ :e  Hash bits 0-4: 01110 → 5
```

Figure 3. Visualization of `{:a 1 :b 2 :c 3 :d 4 :e 5 :f 6 :g 7 :h 8 :i 9 :j 10}`, hashes in Appendix B.

```
0 |  a  |  6
2 |  b  |  3
: 
4 |  c  |  9            c  |  d
```
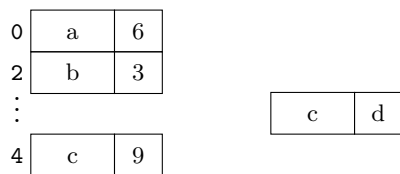
When we need to insert a new node, we allocate a new array of size $2*(n+1)$, and we "fit" the new key at position $2*idx$ and the new value at position $2*idx+1$. The bitmap is still valid!

Take the bitmap `0010 0010` which has the 2nd bit's entry at position 0-1 in the array, and the 6th bit's entry at position 2. If we insert a new entry at bit 5, `0011 0010`, we extend our array to have 3*2 elements, and insert the new entry at position 2-3, bumping the 6th bit to positions 4-5. Notice the bitmap still holds—the 2nd bit has zero 1's below it, so it is at entry $0*2 = 0$; the 5th bit has a single 1 bit below it, so its index is $1*2 = 2$; the 6th bit has two 1 bits below it, so its index is $2*2 = 4$.

This continues until we need a 17th entry, at which point we convert to a `ArrayNode`.

ArrayNode   An ArrayNode allocates a length 32 array with a subnode in every element, rather than every second element in a `BitmapIndexedNode`. Now the `mask` function truly returns the index into the array, so `bitpos` is no longer needed.

3.3 Search

Search is straightforward and similar to Section 2.

3.4 Insertion

Hash collisions   In the case of two non-equal keys hashing to the same value, the implementation then resorts to a linear search amongst the keys to find a match. The `HashCollisionNode` plays this role.

3.5 Deletion

Deletion is slightly more complicated than Bagwell's original HAMT. Since Clojure's HAMT is persistent and immutable, it must copy and alter the node containing the deleting leaf. The rest of the trie can be reused, making this still a cheap operation.

4. Reimplementation in Clojure

We reimplemented [1] the `PersistentHashMap` class from the Clojure standard library in Clojure—it was originally written in Java. This is not the first reimplementation. ClojureScript features similar pure-Clojure port in its standard library, and several other languages have ported their own versions based on Clojure's original implementation.

Our decision to port to Clojure was mainly educational. We extended the implementation with a graphical visualization of the underlying trie (Figure 2).

4.1 Implementation Challenges

We came across several challenges in porting from Java to Clojure.

Automatic widening before bit operations   While Clojure and Java share the same bit layout for numbers, two's complement, Clojure defaults to 64-bit integers, while the Java implementation used 32-bit integers. We came across several situations where we intended 32-bit bit operations, but Clojure upcasted the input to 64-bits. This was a problem especially for negative numbers—in the two's complement representation, the most significant bit is 1, so the output of a bit-operation was wildly incorrect if interpreting the full 64-bits as an answer.

Mutation   Clojure does not expose plain Java variables, so several style and performance decisions were made faithfully port local loops with complicated mutation. We decided to use Clojure's volatiles, which are variables that guarantee ordering constraints (like a `volatile` Java variable).

Autoboxing and object identity   The Java implementation often compares two primitive `int` values with `==`, which uses object identity—written `clojure.core/identical?` in Clojure. Care was needed to reason about when autoboxing was happening, so a literal transliteration of Java's `==` to `clojure.core/identical?` was actually correct. Sometimes, object identity was important, such as comparing two nodes. Other times, it was better to port to `clojure.core/=`, which features more consistent numeric comparisons.

4.2 Tests

We used the excellent `collection-check` [2] library to fuzz test our implementation. It successfully narrowed down sev-

---

[1] https://github.com/frenchy64/optimized-kw-maps

[2] https://github.com/ztellman/collection-check

eral bugs, and greatly increased our confidence that the implementation was correct.

4.3 Visualization

We also implemented graphical trie visualization via `rhizome`[3]. See Figure 3 for an example for a representative trie.

5. Related Work

Recently, Steindorfer and Vinju [4] created Heterogeneous Hash Array Mapped Tries. They are interested in creating a product line of HAMT implementations [5].

6. Dedication

This paper is dedicated to the memory of Phil Bagwell.

References

[1] Phil Bagwell. Fast and space efficient trie searches. Tech. rep. 2000.

[2] Phil Bagwell. Ideal hash trees. Tech. rep. 2001.

[3] Rich Hickey. "The Clojure programming language". In: Proceedings of the 2008 symposium on Dynamic languages. ACM. 2008, p. 1.

[4] Michael J. Steindorfer and Jurgen J. Vinju. "Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections". In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 783–800. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814312. URL: https://doi.acm.org/10.1145/2814270.2814312.

[5] Michael J. Steindorfer and Jurgen J. Vinju. "Towards a Software Product Line of Trie-based Collections". In: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. GPCE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 168–172. ISBN: 978-1-4503-4446-3. DOI: 10.1145/2993236.2993251. URL: https://doi.acm.org/10.1145/2993236.2993251.

Appendices

A. Remark on unsigned bit arithmetic on the JVM

Clojure's implementation of HAMT is implemented on the JVM, where 32-bit integers are signed. The HAMT implementation, however, treats hashes as arbitrary strings of 32-bits, so we need to emulate unsigned arithmetic operations.

The JVM represents integers using 2's complement, which we will briefly describe. To calculate the corresponding negative number for a positive number, simply invert all the bits and add one.

For example, the number -4 can be derived by taking 4, flipping the bits, and adding 1 (below).

```
0000 0000 0000 0000 0000 0000 0000 0100
  flip bits
1111 1111 1111 1111 1111 1111 1111 1011
  add one
1111 1111 1111 1111 1111 1111 1111 1100
```

This representation is mostly transparent for our purposes—except for the bit shift right operation. The JVM exposes two operations bit shift right operations: signed (`>>` in Java, `bit-shift-right` in Clojure), and unsigned (`>>>` in Java, `unsigned-bit-shift-right` in Clojure).

The difference is, `>>` preserves the most significant bit, while `>>>` replaces it with 0.

```
1000 1101 >>  1 = 1100 0110    //signed
1000 1101 >>> 1 = 0100 0110    //unsigned
```

Unsigned bit operations are necessary because no bits are special in hashes and bitmaps.

B. Hashes for examples

```
            6     5     4     3     2     1     0
(hash :a) = 10 00000 10110 11110 10111 11000 11110
(hash :b) = 01 01100 00101 10001 11100 11010 10110
(hash :c) = 10 01011 01110 01111 10100 10111 10001
(hash :d) = 01 11010 11000 11001 00000 01010 11000
(hash :e) = 01 01001 00101 01000 11111 10110 01110
(hash :f) = 10 10000 01100 11011 01000 01010 11000
(hash :g) = 01 10011 11001 10010 01001 01101 10001
(hash :h) = 01 00001 00010 01000 00011 00011 10100
(hash :i) = 10 10110 10101 01100 11110 11000 11101
(hash :j) = 10 10110 01010 11001 00110 01000 10011
```

---

[3] https://github.com/ztellman/rhizome