Persistent Data Structures

From Lists to Hashmaps

Slides @ binderdavid.github.io/talks

David Binder, August 2024

Who uses Git?

Git is a Persistent Data Structure

- You can add, delete and change files.
- You can undo all these operations.
- You can take a previous version in your history and branch from that.
- The implementation is not immutable! (Cf. pack files, delta compression)

If you think Git is useful, then persistent data structures are for you :)

Beginner Technique: Structural Sharing

It's all about sharing...

...memory.

Structural Sharing by Example

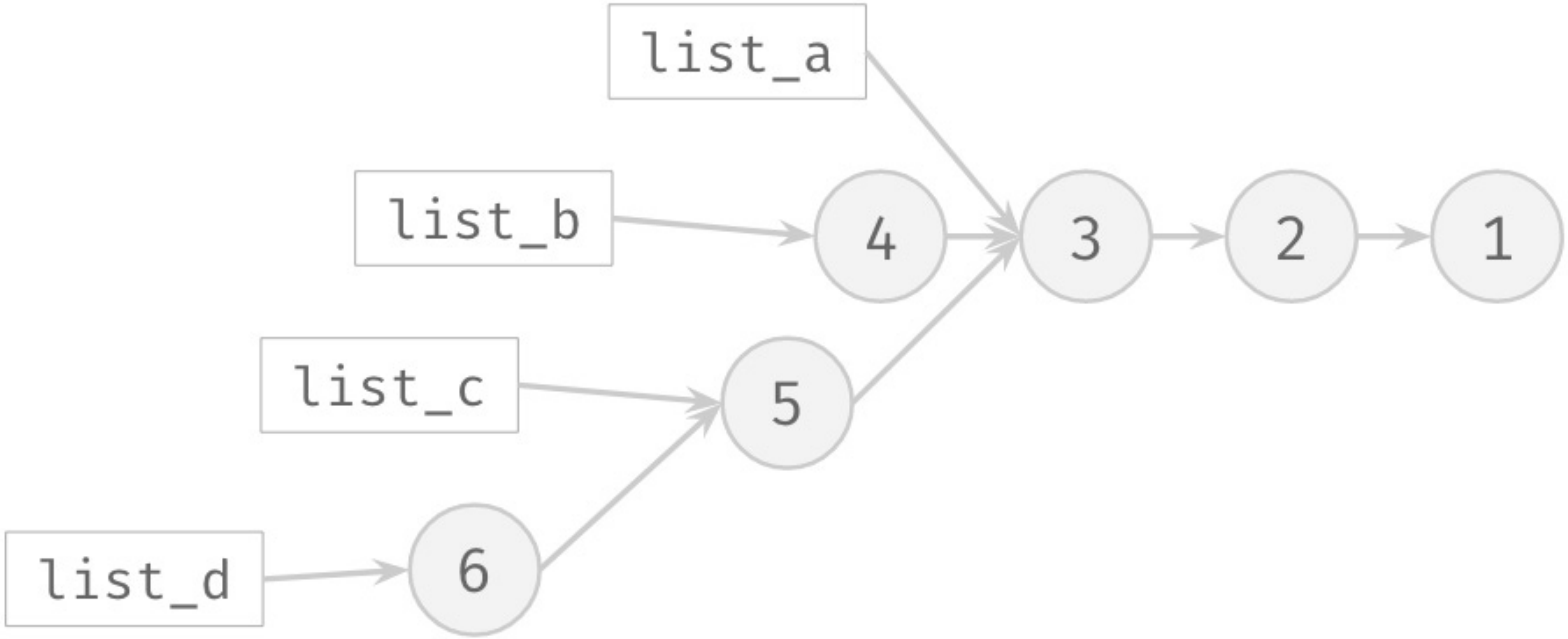
Singly-Linked Lists

```
list_a = [3,2,1]               = [3,2,1]
list_b = [4] ++ list_a         = [4,3,2,1]
list_c = [5] ++ list_a         = [5,3,2,1]
list_d = [6] ++ list_c         = [6,5,3,2,1]
```

We don't want to do defensive copying!

Structural Sharing by Example

Singly-Linked Lists



(c) Jeff Schwab, https://nested.substack.com/p/intro-to-persistent-data-structures

Structural Sharing in Practice

- Data structures have to be designed with sharing in mind.
- Algorithms have to be written with sharing in mind.
- Challenge: Write a filter function which shares the tail!
- Okasaki's "Purely Functional Data Structures" contains many examples.

Recommended!

Hashmaps

What are hashmaps / dictionaries?

What operations are supported?

```
{
    id: 14823777
    name_first: "David",
    name_last: "Binder",
    favourite_movie_of_all_time_and_space: "La Nuit de Varennes"
}
```

Supports insertion of key-value pairs.

Supports key lookup.

Supports deletion of key.

Supports update of a key with a value.

What are hashmaps / dictionaries?

What are some problems for implementations?

```
{
    id: 14823777
    name_first: "David",
    name_last: "Binder",
    favourite_movie_of_all_time_and_space: "La Nuit de Varennes"
}
```

Bad: Different key length.

Bad: Keys are not random, which can lead to degenerate data structures.

What are hashmaps / dictionaries?

Use hashed keys instead!

```
{
   fqrstvnx: 14823777,
   yxdagxqz: "David",
   krgjkhsc: "Binder",
   etrcukad: "La Nuit de Varennes"
}
```

hash(id) = fqrstvnx
hash(name_first) = yxdagxqz
hash(name_last) = krgjkhsc
hash(favourite_movie_of_all_time_and_space) = etrcukad

Random distribution of keys!

Fixed length of keys!

We use short, non-cryptographic hashes.

We also have to deal with hash-collisions :(

Advanced Technique:
Hash Array Mapped Tries

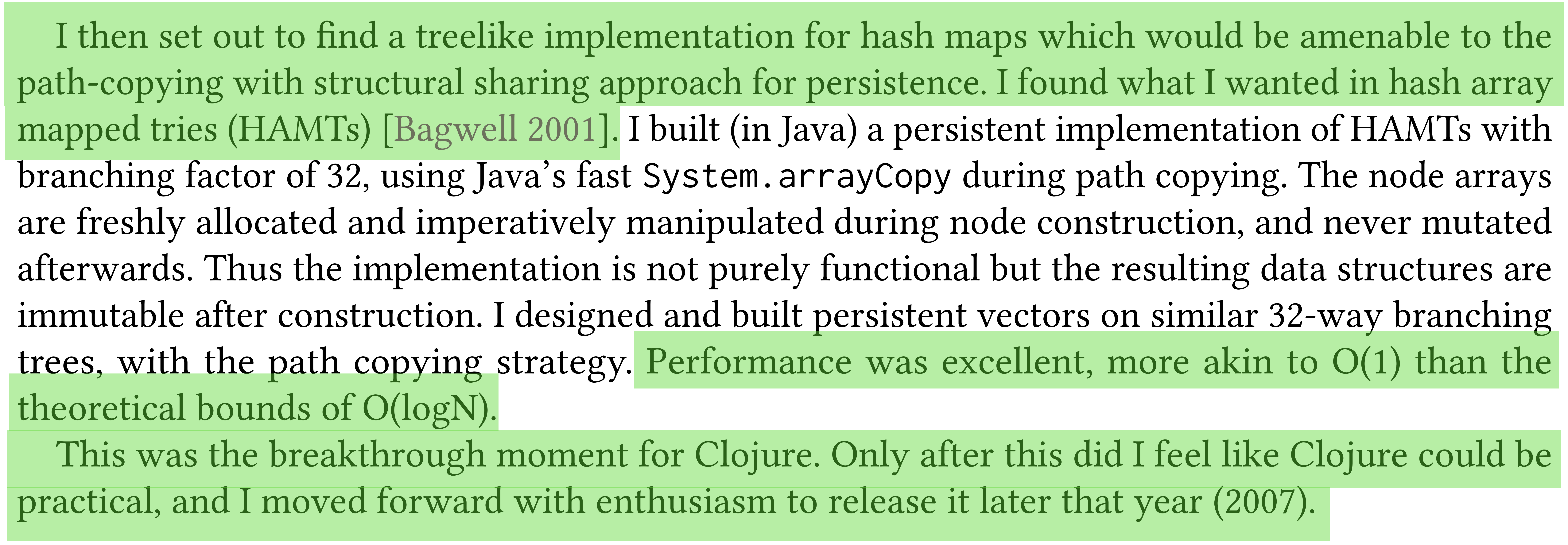
A History of Clojure

Rich Hickey @ History of Programming Languages Conference

3.4.1 Persistence and Immutability. What I thought would be a simple matter of shopping for best-of-breed functional data structures ended up being a search and engineering exercise that dominated my early Clojure work, as evidenced by the gap in commits during the winter of 2006/7 (figure 2). I started by looking at Okasaki [1999], and found the data structures too slow for my use, and felt some of the amortized complexity would be difficult to explain to working programmers. I also concluded that it mattered not at all to me that the implementation of the data structures be purely functional, only that the interface they presented to consumers was immutable and persistent.

A History of Clojure

Rich Hickey @ History of Programming Languages Conference



I then set out to find a treelike implementation for hash maps which would be amenable to the path-copying with structural sharing approach for persistence. I found what I wanted in hash array mapped tries (HAMTs) [Bagwell 2001]. I built (in Java) a persistent implementation of HAMTs with branching factor of 32, using Java's fast `System.arrayCopy` during path copying. The node arrays are freshly allocated and imperatively manipulated during node construction, and never mutated afterwards. Thus the implementation is not purely functional but the resulting data structures are immutable after construction. I designed and built persistent vectors on similar 32-way branching trees, with the path copying strategy. Performance was excellent, more akin to O(1) than the theoretical bounds of O(logN).

This was the breakthrough moment for Clojure. Only after this did I feel like Clojure could be practical, and I moved forward with enthusiasm to release it later that year (2007).

Hash Array Mapped Trie

It's (Hash (Array-Mapped (Trie))), really.

- Q1: What is a trie?

  A: A tree that contain prefixes.

- Q2: What is an array-mapped trie?

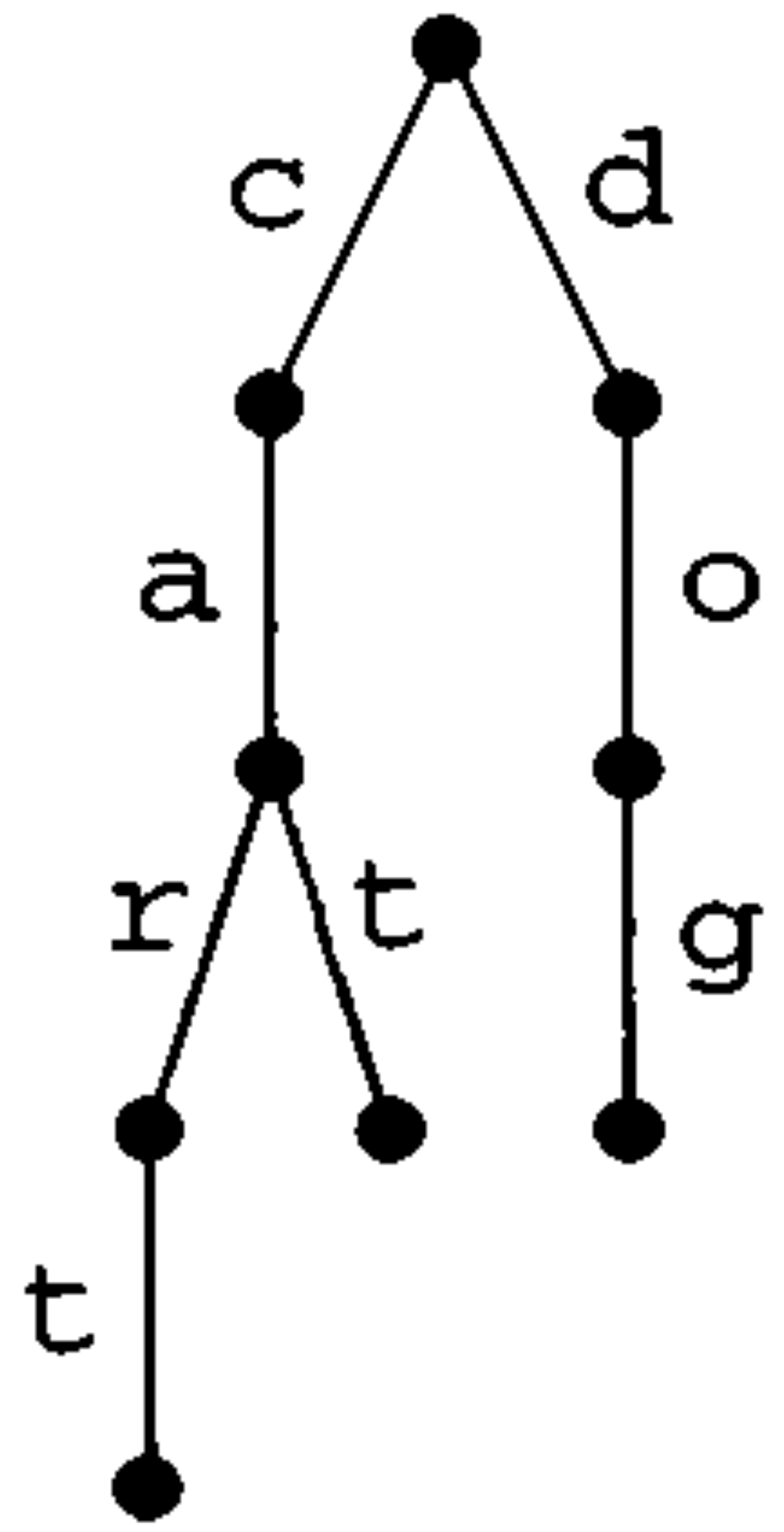
  A: An efficient representation of tries.

- Q3: What is a hash array-mapped trie?

  A: An efficient trie that contains prefixes of hashes.

Q1: What is a trie?

A tree that contains prefixes



Q: What words are present in this trie?

A: Unclear. We have to store info in nodes.

Q: What is the branching factor?

A: 26 (letters of alphabet)

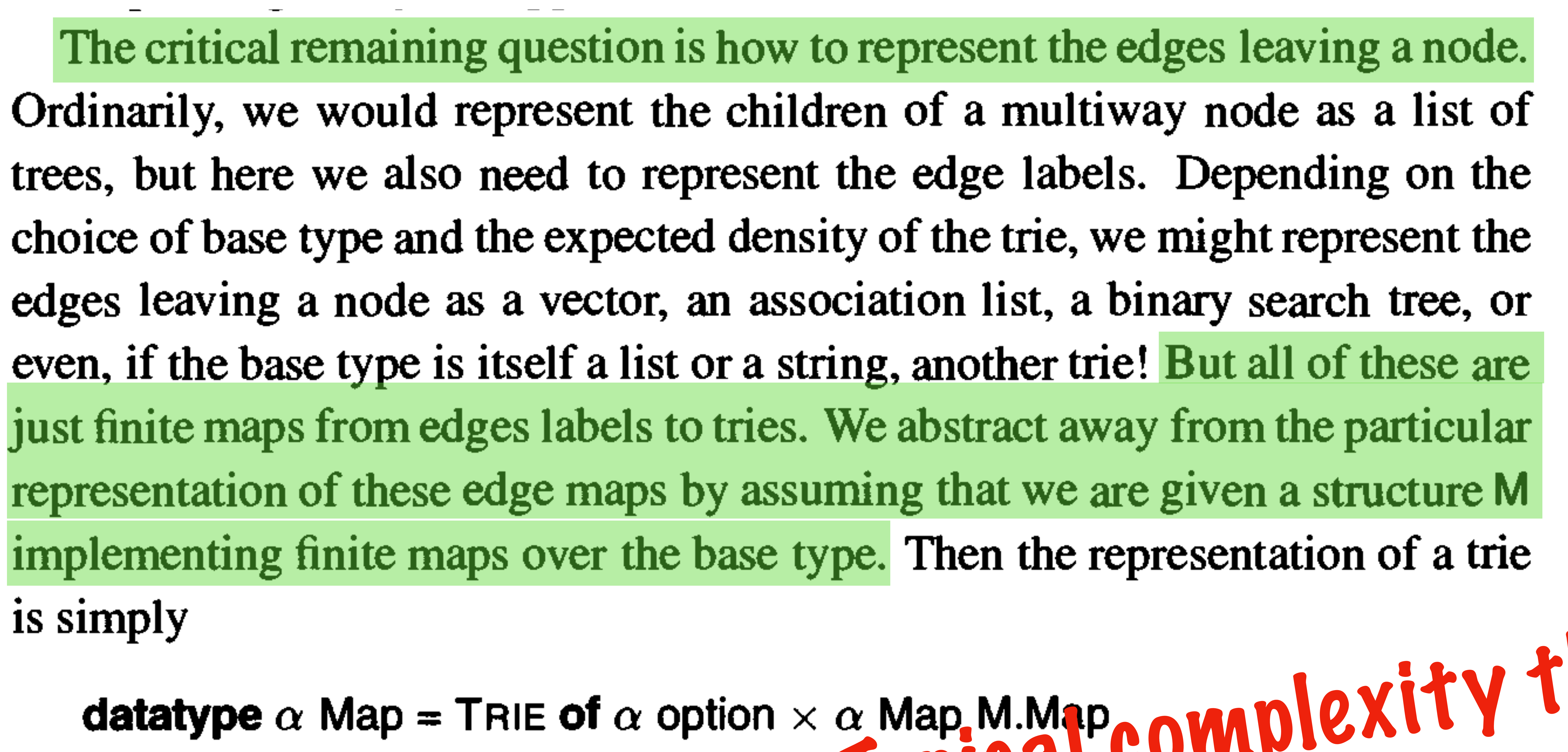
A*: A power of 2 in real implementations

A**: 2^5 = 32 very common

Q2: What is an array-mapped trie?

An efficient representation of tries

What does Okasaki have to say?



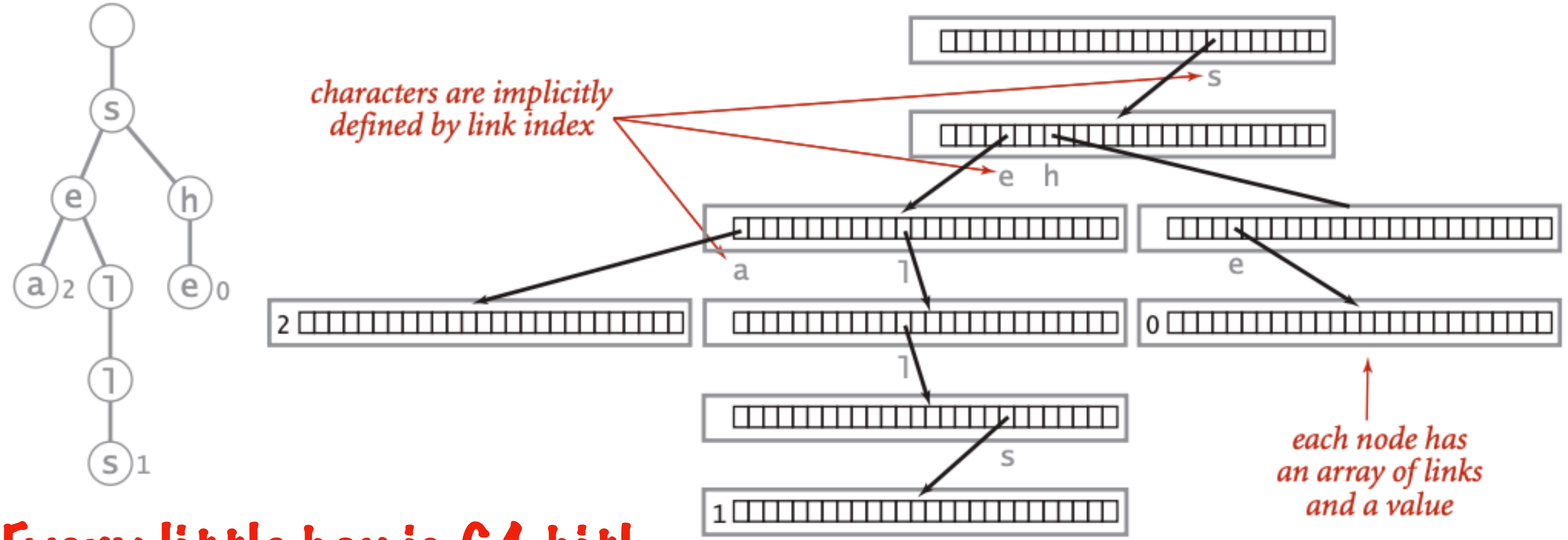
The critical remaining question is how to represent the edges leaving a node. Ordinarily, we would represent the children of a multiway node as a list of trees, but here we also need to represent the edge labels. Depending on the choice of base type and the expected density of the trie, we might represent the edges leaving a node as a vector, an association list, a binary search tree, or even, if the base type is itself a list or a string, another trie! But all of these are just finite maps from edges labels to tries. We abstract away from the particular representation of these edge maps by assuming that we are given a structure M implementing finite maps over the base type. Then the representation of a trie is simply

datatype α Map = TRIE **of** α option $\times$ α Map M.Map

Typical complexity theorist :D

Q2: What is an array-mapped trie?

A naive representation of tries:


s
e
h
a
2
l
e
0
l
s
1
characters are implicitly defined by link index
s
e h
a
l
e
2
0
l
s
1
each node has an array of links and a value

Every little box is 64 bit!

Vast majority contain null pointers!

Q3: What is a hash array-mapped trie?

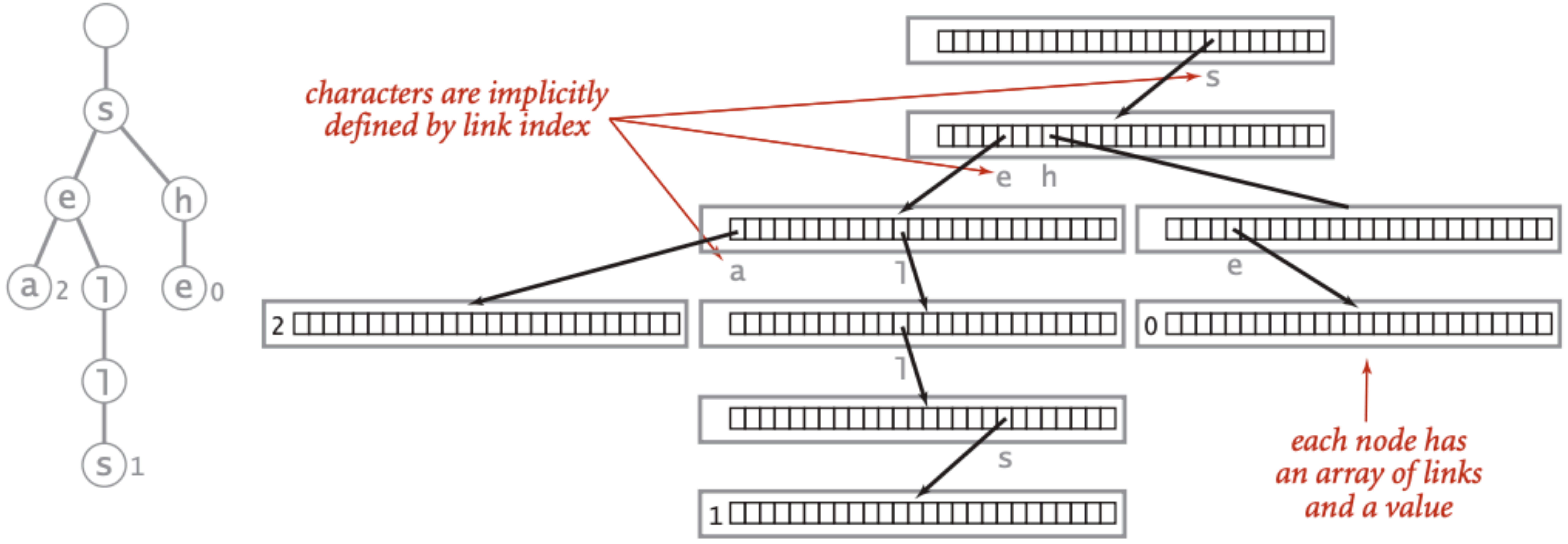
An efficient trie that contains prefixes of hashes

Enter Phil Bagwell!

- Problem: Most nodes are sparse! There are only very few outgoing edges.
- Solution: Every node contains a bitmap and an array of pointers!
- The bitmap says which outgoing edges exist. Usually only needs 32 or 64 Bit!
- The array contains the pointers for the outgoing edges.

Only needs: #outgoing-edges * size-of-pointer space.

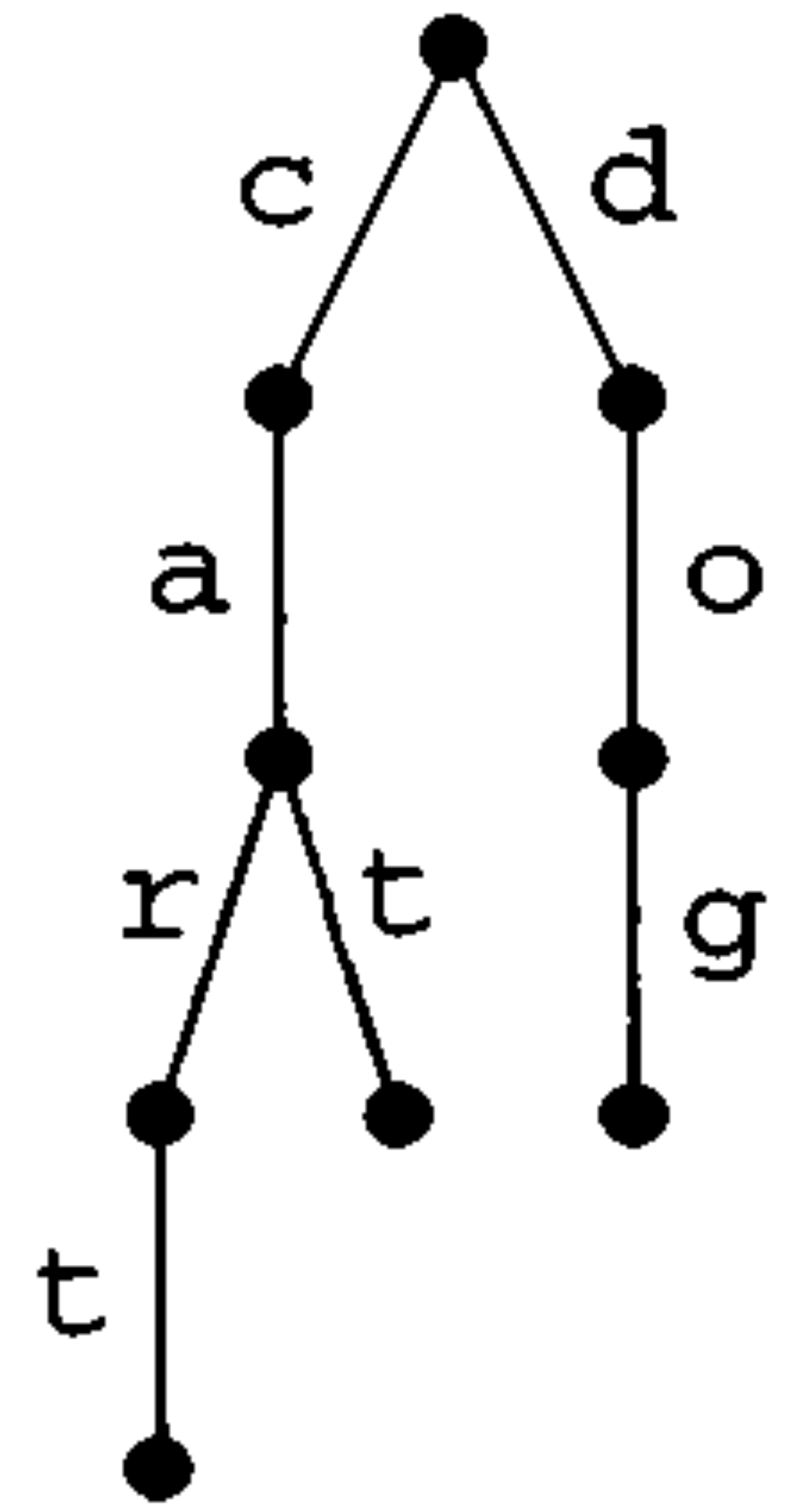
Q2: What is an array-mapped trie?



s
e
h
a
2
l
e
0
l
s
1
characters are implicitly defined by link index
s
e
h
a
l
e
2
0
l
s
1
each node has an array of links and a value

Q3: What is a hash array-mapped trie?

A tree that contains prefixes of hashes



Q: What is the maximal depth?

A: 45
pneumonoultramicroscopicsilicovolcanoconiosis

A*: Length of Hash / Bits per Edge

A**: 64 / 5 (unordered-containers)

Let's (finally) see some code!

Hash Array-Mapped Tries in Haskell

The unordered-containers library

```
data HashMap k v
    = Empty
    | BitmapIndexed !Bitmap !(A.Array (HashMap k v))
    | Full !(A.Array (HashMap k v))
    | Leaf !Hash !(Leaf k v)
    | Collision !Hash !(A.Array (Leaf k v))
```

Resources

Some Resources I Used

- Chris Okasaki: Purely Functional Data Structures
- Phil Bagwell: Fast And Space Efficient Trie Searches (2000)
- Phil Bagwell: Ideal Hash Trees (2001)
- Ritch Hickey: A History of Clojure
  https://www.youtube.com/watch?v=nD-QHbRWcoM
- The unordered-containers library on Hackage