THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

Hash Tables

CS 124 / Department of Computer Science

Hashing

Hashing is a *big* subject.

What we will study is how to use hashing to construct and use *hash tables.* A hash table is a data structure that provides for access to elements in $O(1)$ time.

We will *not* be studying cryptographic hashing, or hashing used for data security, authentication, blockchain, proof-of-work, or other applications. That's an entirely different subject. (If you're interested in this, see: CS 166, CS 167, and CS 266.)

Hashing

We've looked at many ways to find an element in various data structures. We've seen binary search trees, b-trees, binary search, interpolation search, and more. These approaches are all based on *finding* an object within a data structure.

But what if there were a way to know exactly where to find a given object?

How might that work?

Motivation

In your projects, you've been working with a set of objects that have at least one unique value for each object.

Let's say you wanted to store a collection of your objects in some data structure, say a vector.

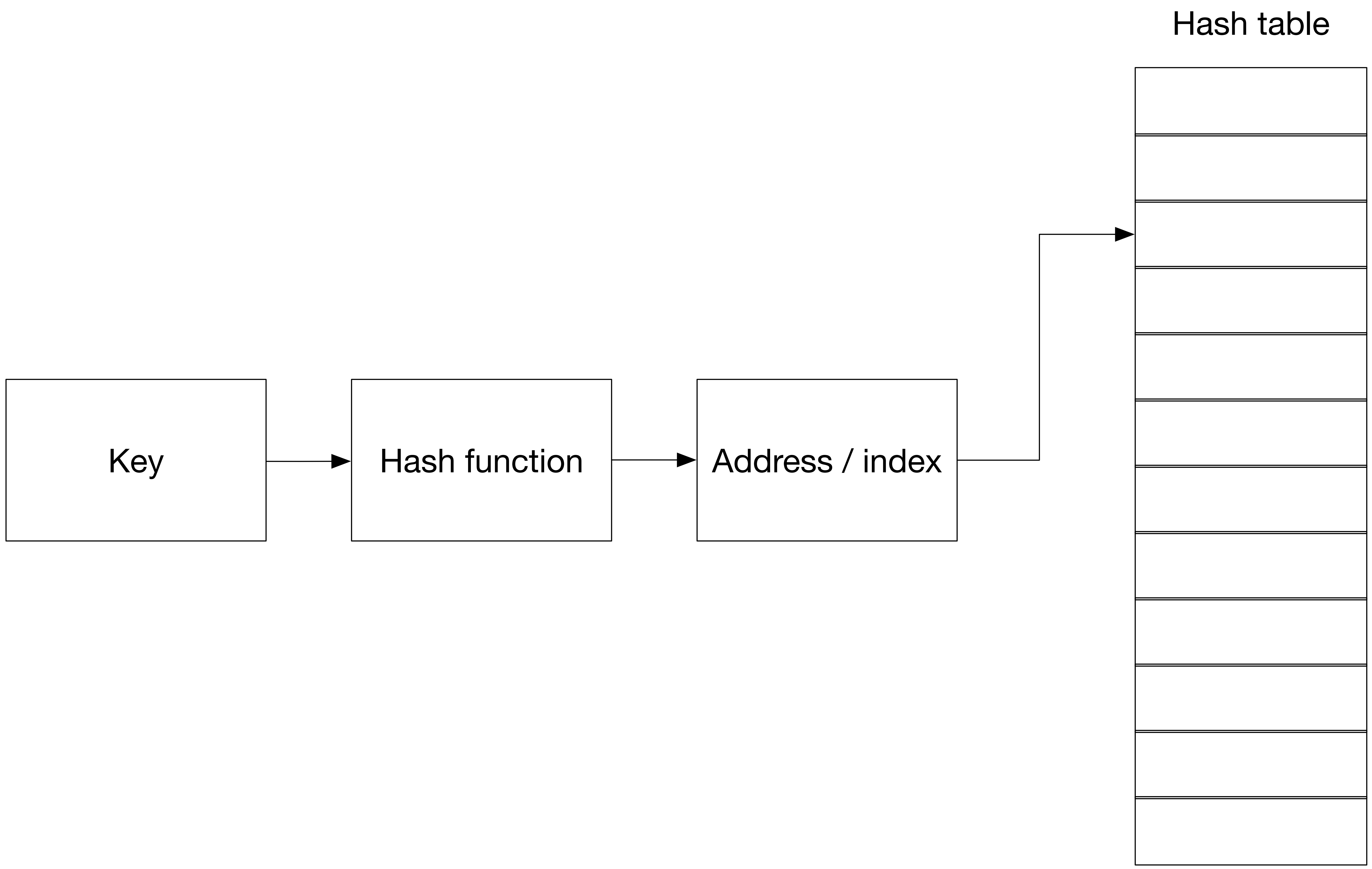
How would you store your objects in the vector, and how would you retrieve them?

You could add all your objects to the vector, sort the vector, and then binary search (or interpolation search if your objects' unique values have a uniform distribution), but there's a faster way.

Hashing

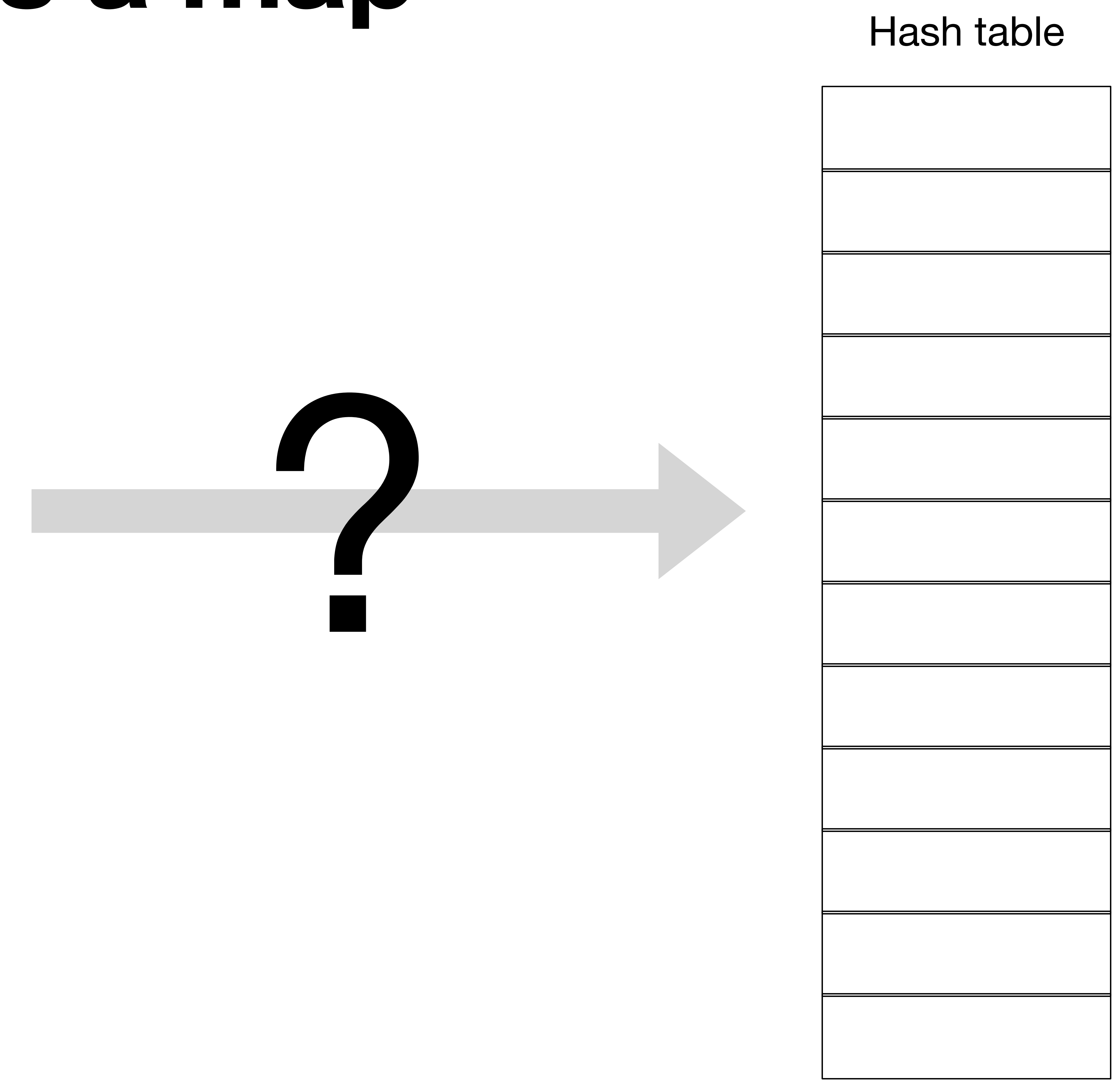
A hash table stores items in a way that lets you calculate *the item*'s *index* directly.

A hash function is a map



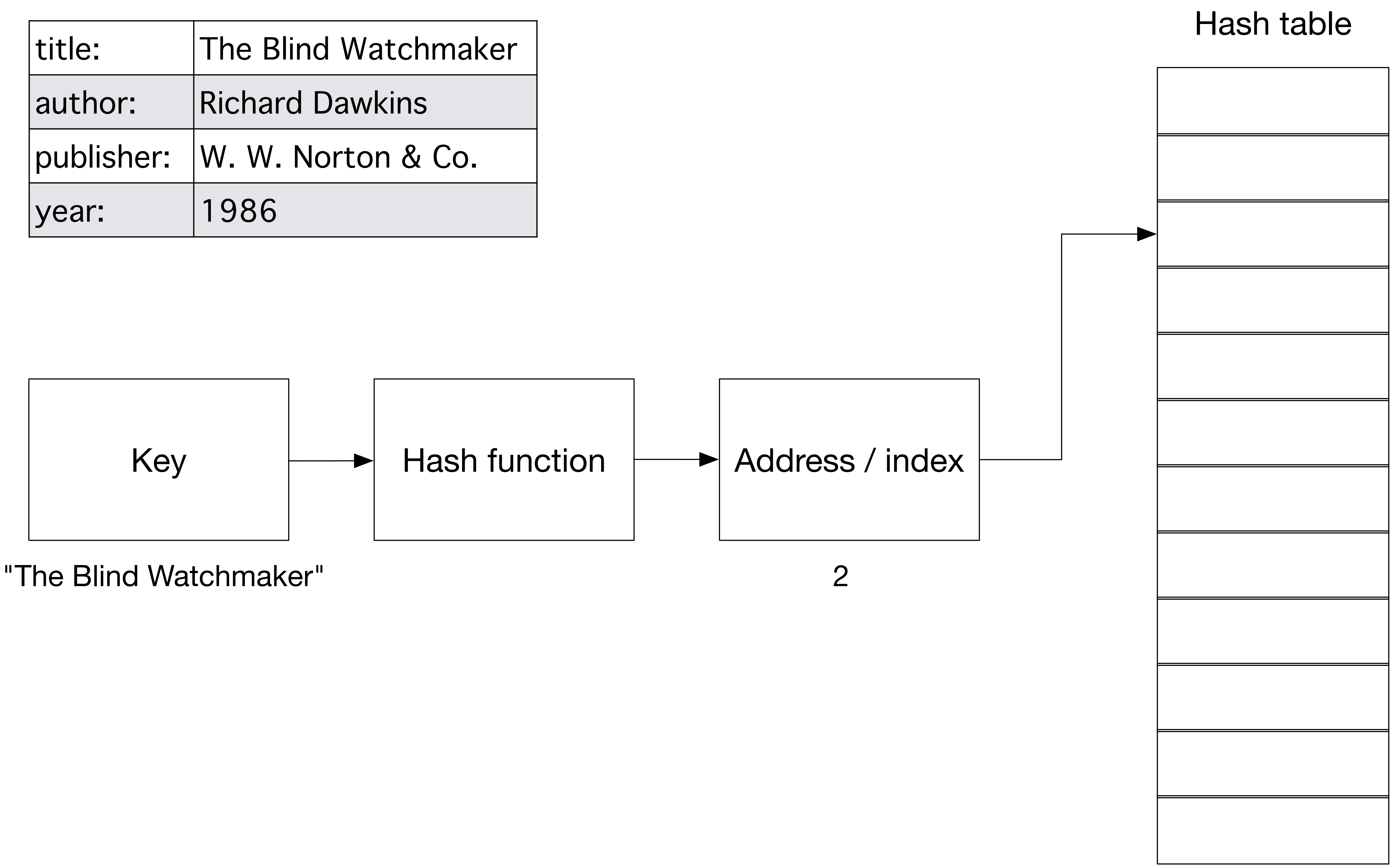
A hash function is a map

title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986

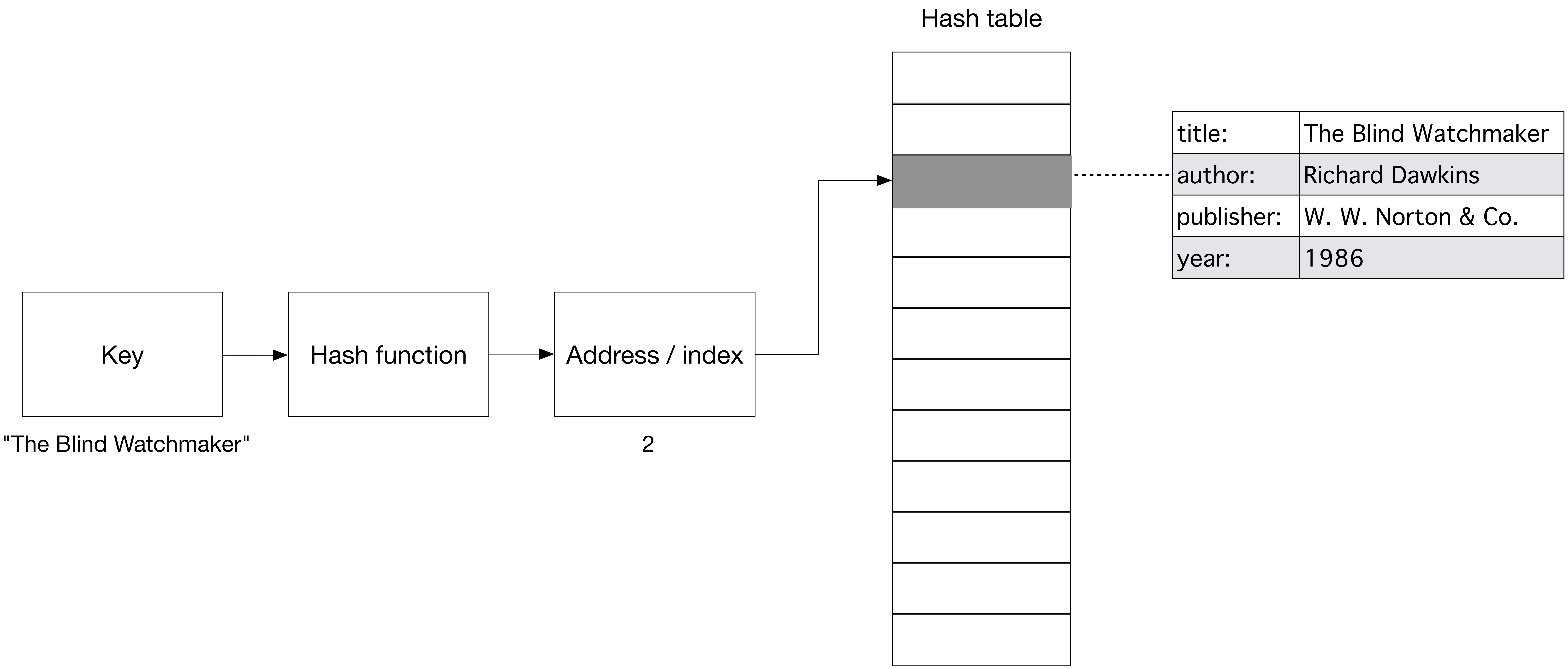


A hash function is a map

title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986



A hash function is a map



Key

"The Blind Watchmaker"

Hash function

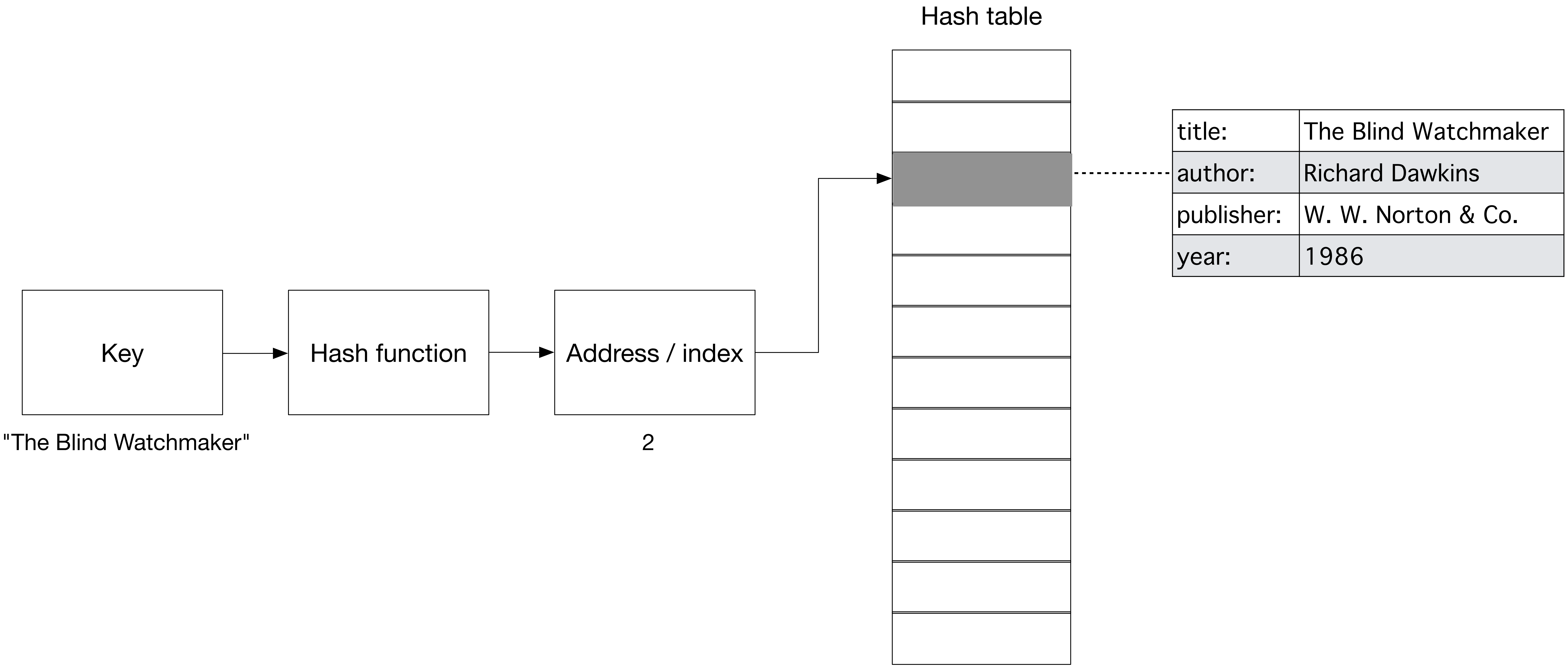
Address / index

2

Hash table

title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986

A hash function is a map



title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986

Choosing a size for our hash table

- If we have *n* objects our hash table should be at least of size *n*.

- We want to have our objects distributed within the table as uniformly as possible.

- We'd like the capacity to be able to add new objects. (We call the ratio of the number of objects to the size of our hash table the "fill percentage" or "load factor.")

- Size of our hash table should be a prime number.

Choosing a hash function

- We want a function that takes a key (a string, an integer, even a combination of multiple values) that returns a value we can use as an index into our hash table.
- This function should be easy to compute.
- This function should produce outputs that are as close to uniformly distributed as possible.
- The range of our function should be from 0 to the size of our hash table - 1.
- We want to avoid different keys resulting in the same index.

Choosing a hash function

Let's say we have a table of size 17. The range of outputs for our hash function should be 0 through 16 -- the valid indices of our hash table. How do we accomplish this?

We calculate a number and then return that number modulo 17.

A not-so-good hash function

```
int notSoGoodHash(std::string s, int maxIndex) {
    int hash = 0;
    for (const char& c : s) {
        hash = hash + c;
    }
    return hash % maxIndex;
}
```

Dog → 10
Catamount → 5
Squirrel → 5
Rabbit → 1
Toad → 1
Monkey → 15
Chipmunk → 15

A better hash function

```
unsigned long betterHash(std::string s, int maxIndex) {
    unsigned long hash = 0;
    for (const char& c : s) {
        hash = hash * 37 + c;
    }
    return hash % maxIndex;
}
```

Dog → 11
Catamount → 9
Squirrel → 12
Rabbit → 3
Toad → 3
Monkey → 14
Chipmunk → 7

Horner hash function

```
unsigned long hornerHash(std::string s, int maxIndex) {
    unsigned long hash = 0;
    for (const char& c : s) {
        hash = hash * 37 + c;
    }
    return hash % maxIndex;
}
```

Horner hash function

```
unsigned long hornerHash(std::string s, int maxIndex) {
    unsigned long hash = 0;
    for (const char& c : s) {
        hash = hash * 37 + c;
    }
    return hash % maxIndex;
}
```

"dcba" → d

→ (37 x d) + c

→ 37 x ((37 x d) + c) + b

→ 37 x (37 x ((37 x d) + c) + b) + a

with x = 37

$a + bx + cx^2 + dx^3$

Horner hash function

```
unsigned long hornerHash(std::string s, int maxIndex) {
    unsigned long hash = 0;
    for (const char& c : s) {
        hash = hash * 37 + c;
    }
    return hash % maxIndex;
}
```

"dcba" → d
→ (37 x d) + c
→ 37 x ((37 x d) + c) + b
→ 37 x (37 x ((37 x d) + c) + b) + a

with x = 37

$a + bx + cx^2 + dx^3$

Choosing a hash function

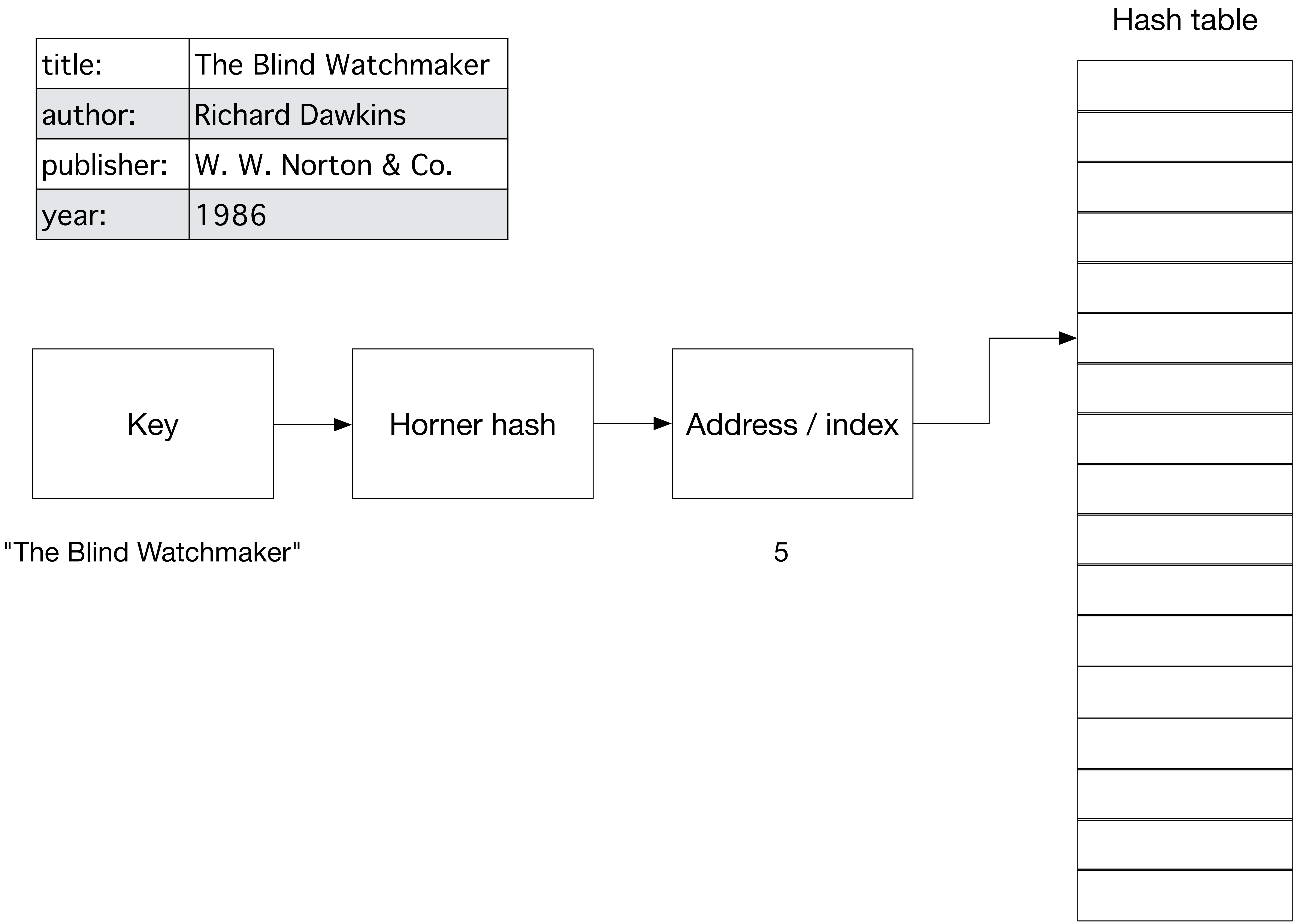
- We want a function that takes a key (a string, an integer, even a combination of multiple values) that returns a value we can use as an index into our hash table.

- This function should be easy to compute.

- This function should produce outputs that are as close to uniformly distributed as possible.

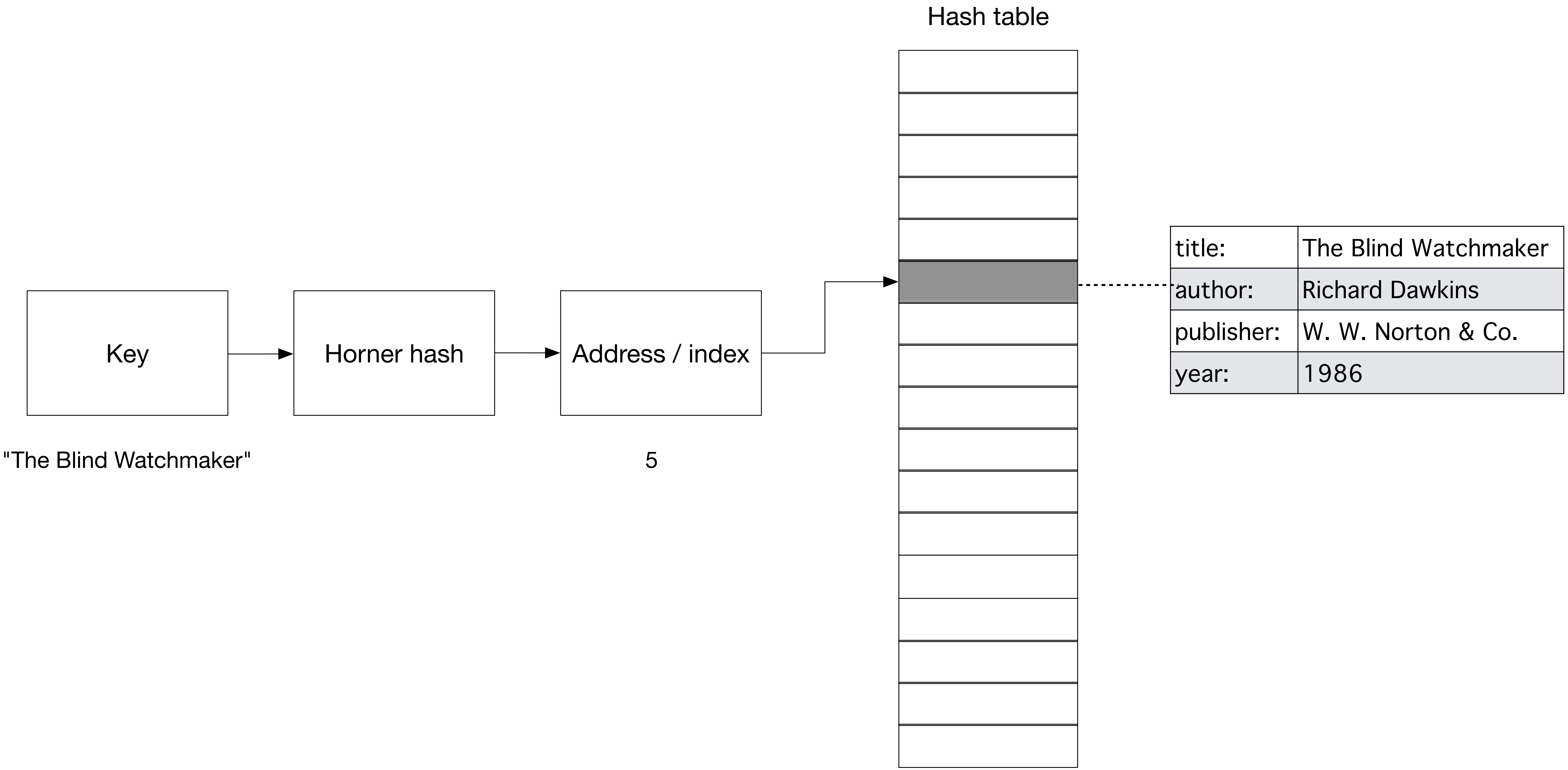
- The range of our function should be from 0 to the size of our hash table - 1.

Inserting



title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986

Inserting



Key

Horner hash

Address / index

"The Blind Watchmaker"

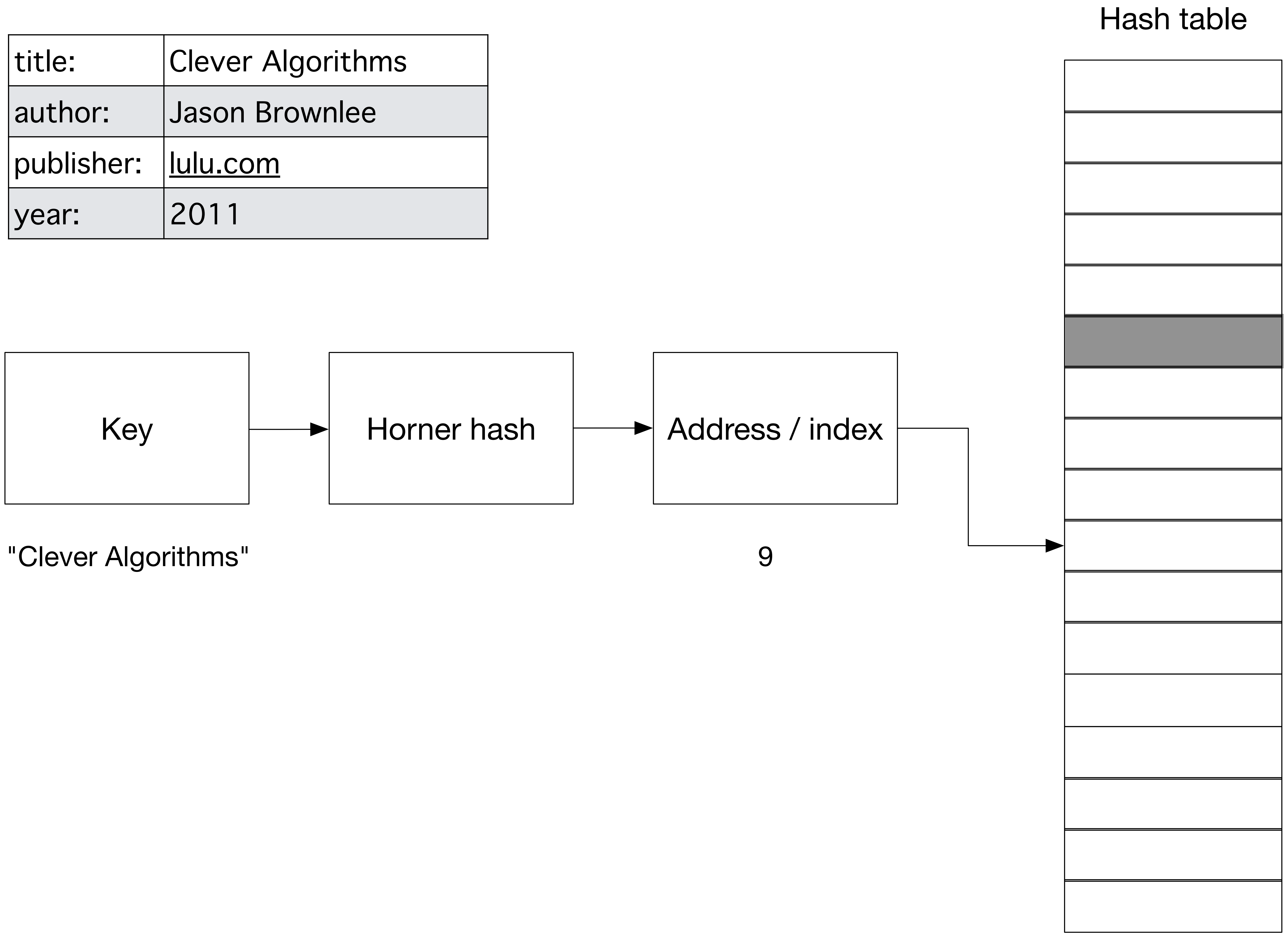
5

Hash table

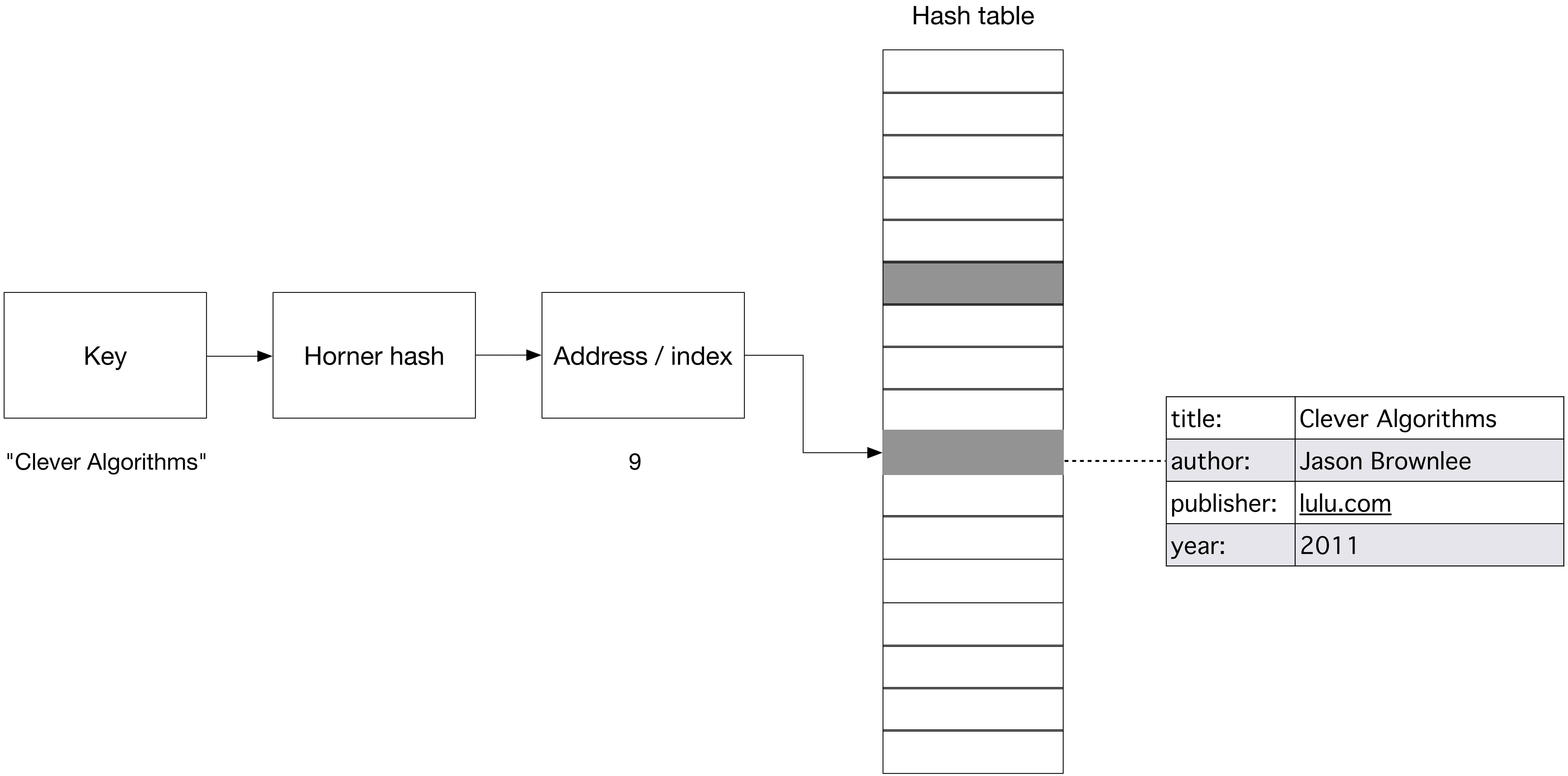
title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986

Inserting

title:	Clever Algorithms
author:	Jason Brownlee
publisher:	lulu.com
year:	2011



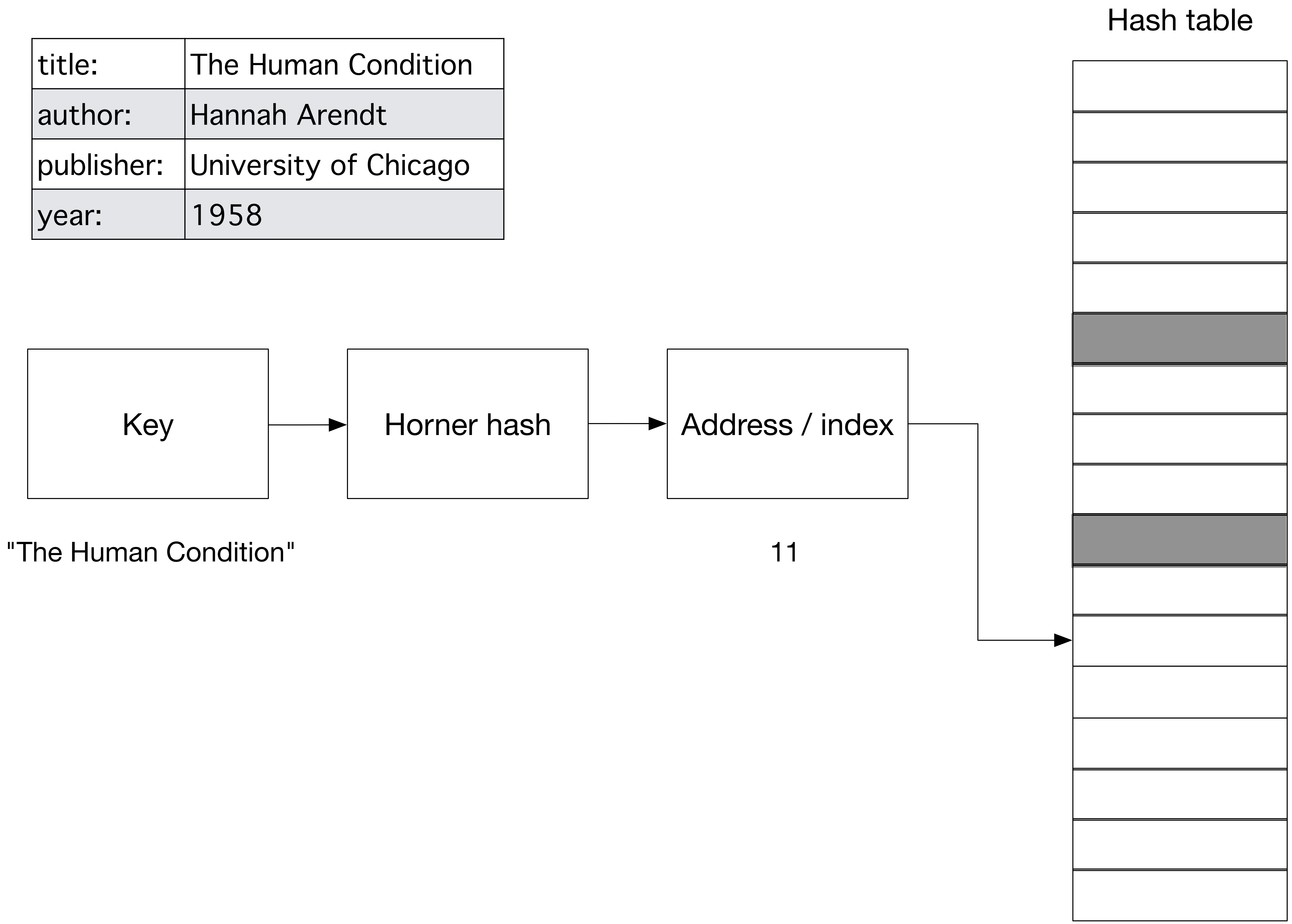
Inserting



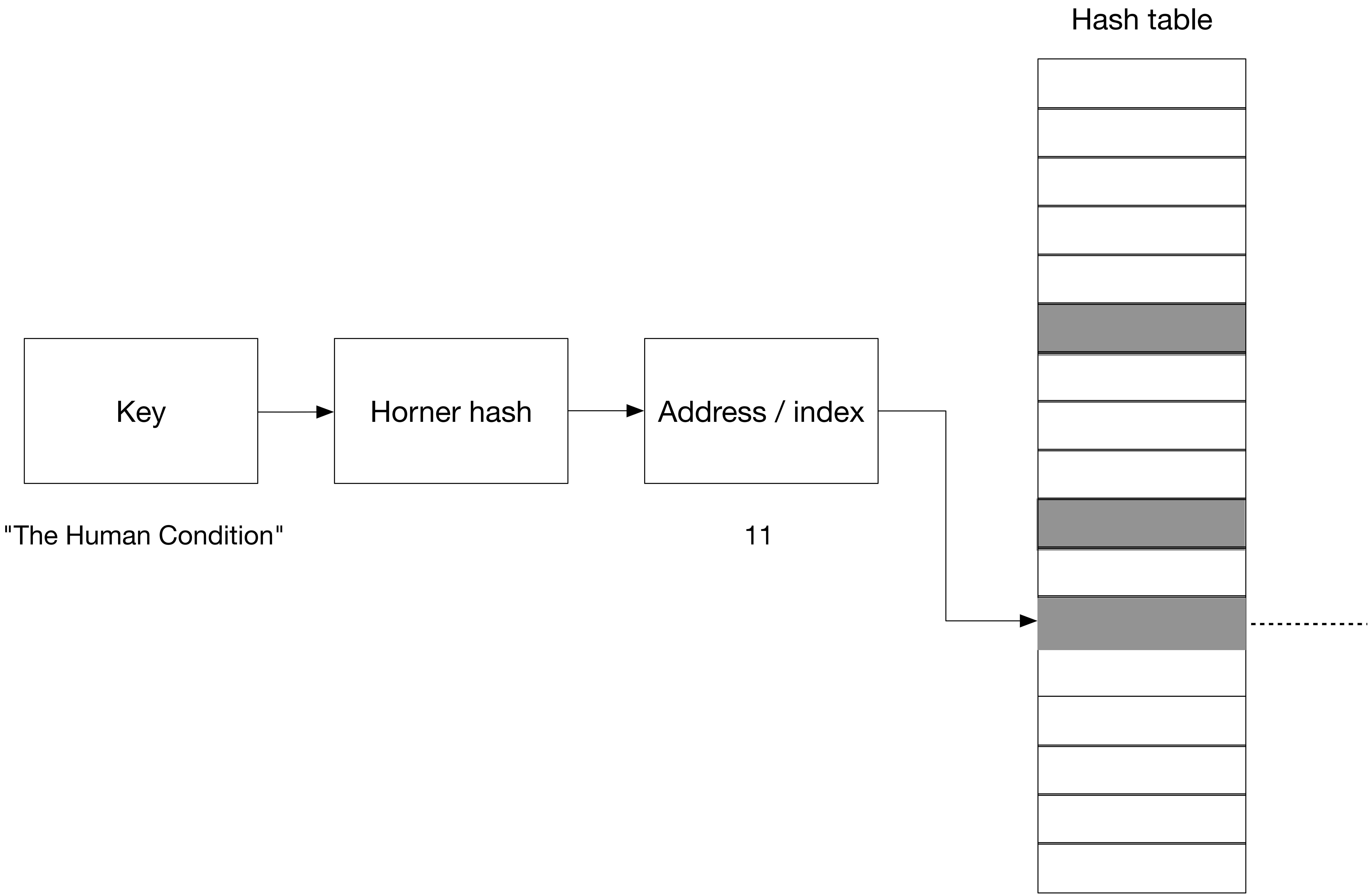
title:	Clever Algorithms
author:	Jason Brownlee
publisher:	lulu.com
year:	2011

Inserting

title:	The Human Condition
author:	Hannah Arendt
publisher:	University of Chicago
year:	1958



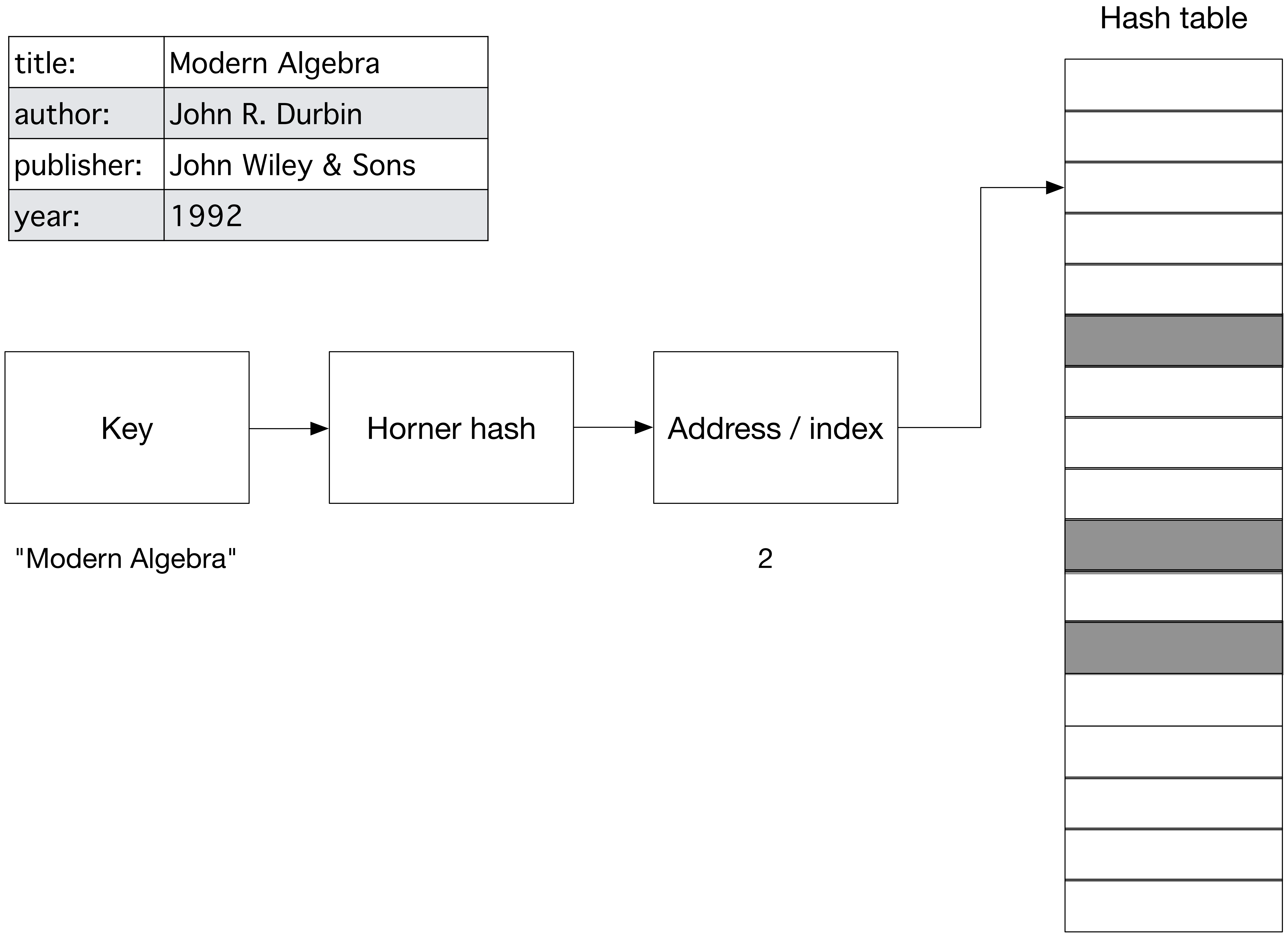
Inserting



title:	The Human Condition
author:	Hannah Arendt
publisher:	University of Chicago
year:	1958

Inserting

title:	Modern Algebra
author:	John R. Durbin
publisher:	John Wiley & Sons
year:	1992



Hash table

Key

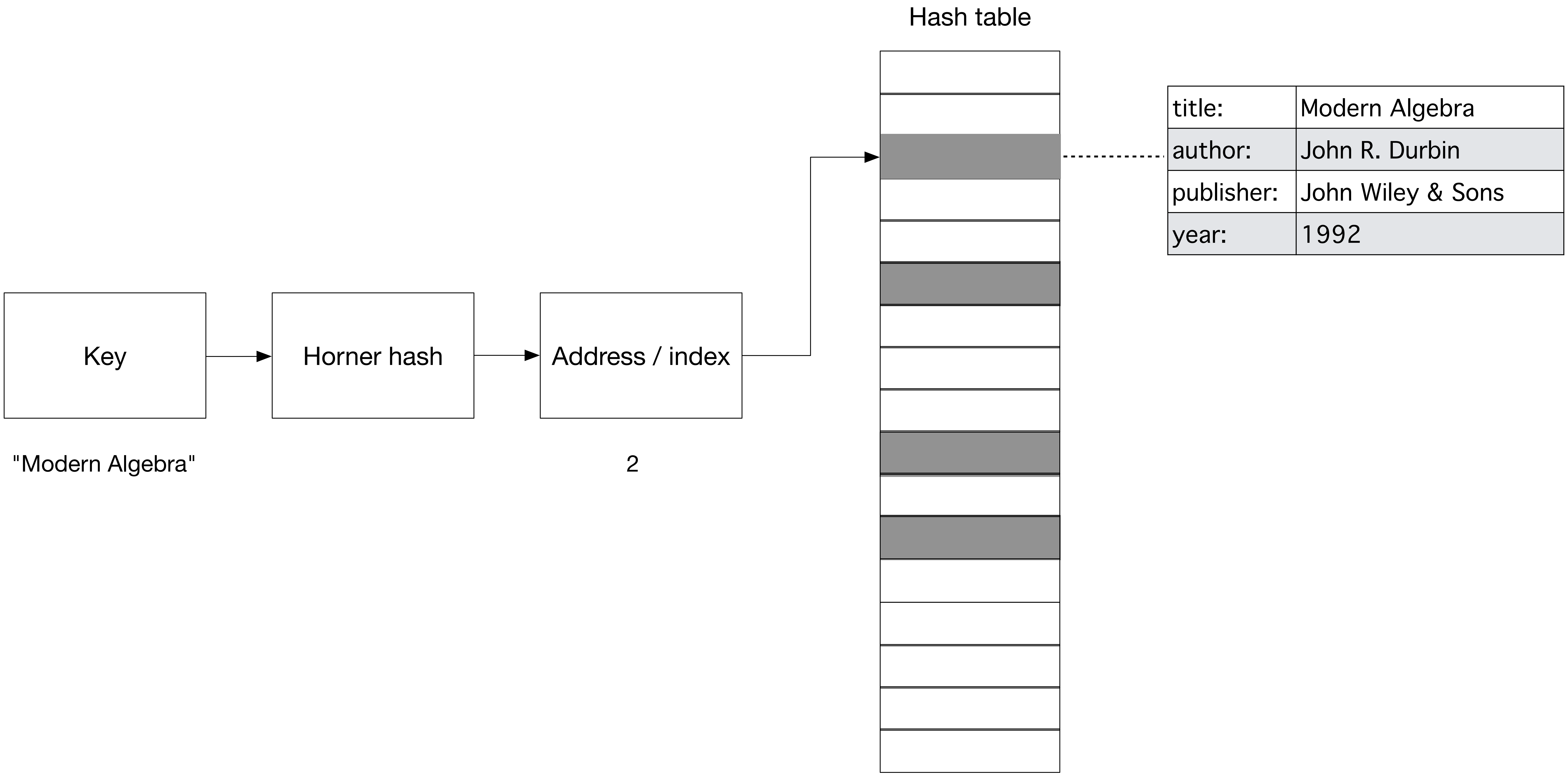
Horner hash

Address / index

"Modern Algebra"

2

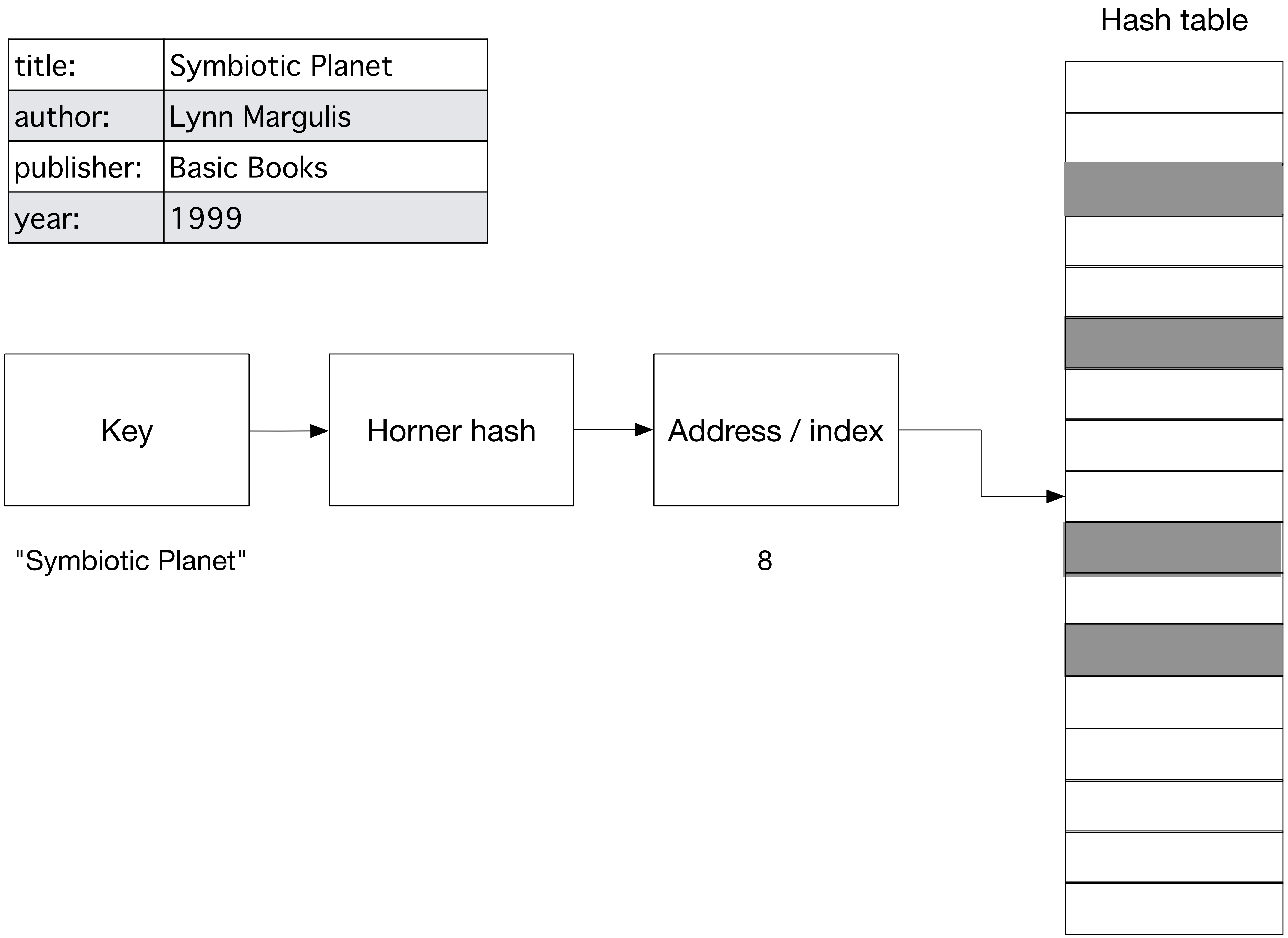
Inserting



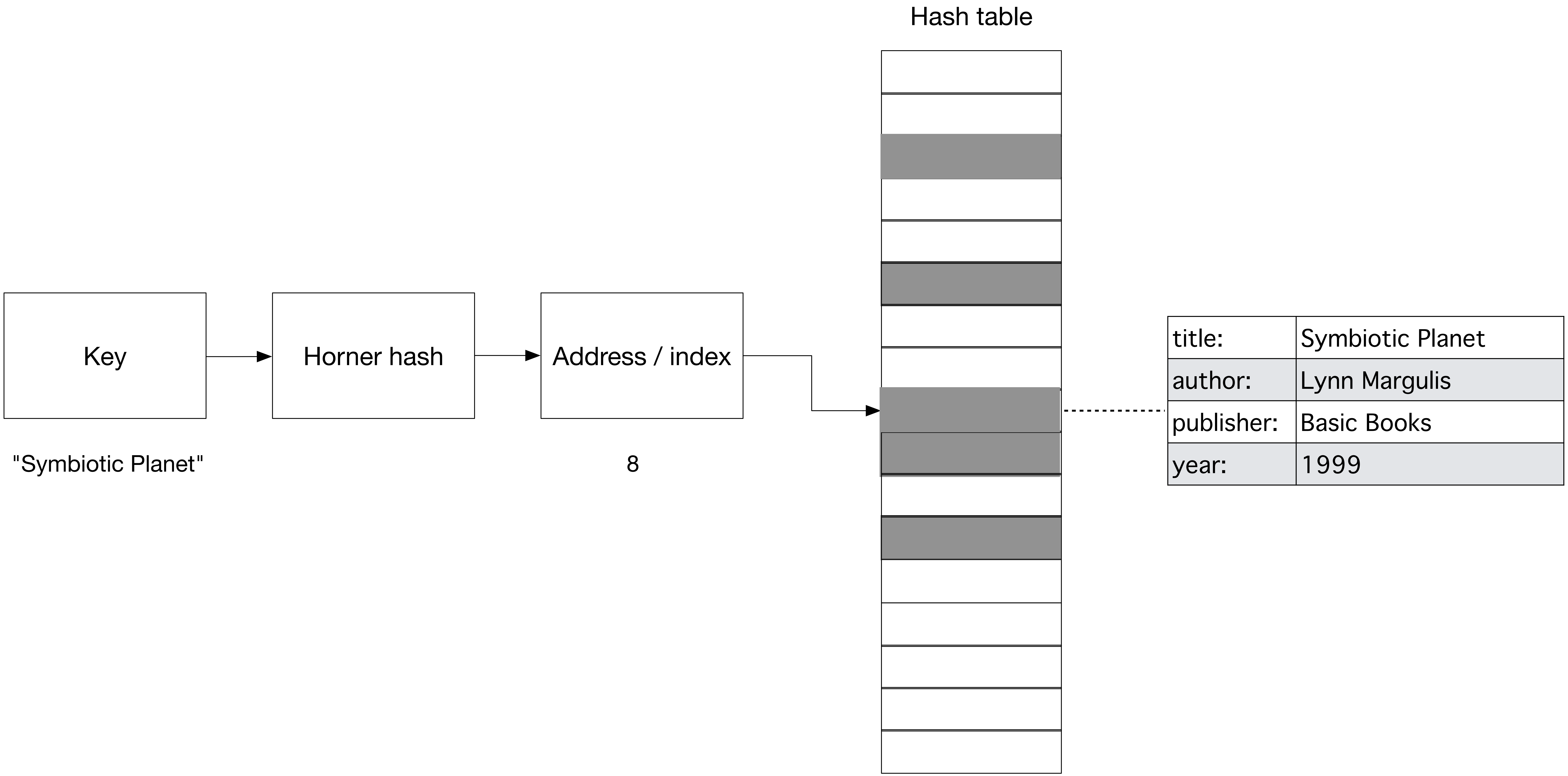
title:	Modern Algebra
author:	John R. Durbin
publisher:	John Wiley & Sons
year:	1992

Inserting

title:	Symbiotic Planet
author:	Lynn Margulis
publisher:	Basic Books
year:	1999



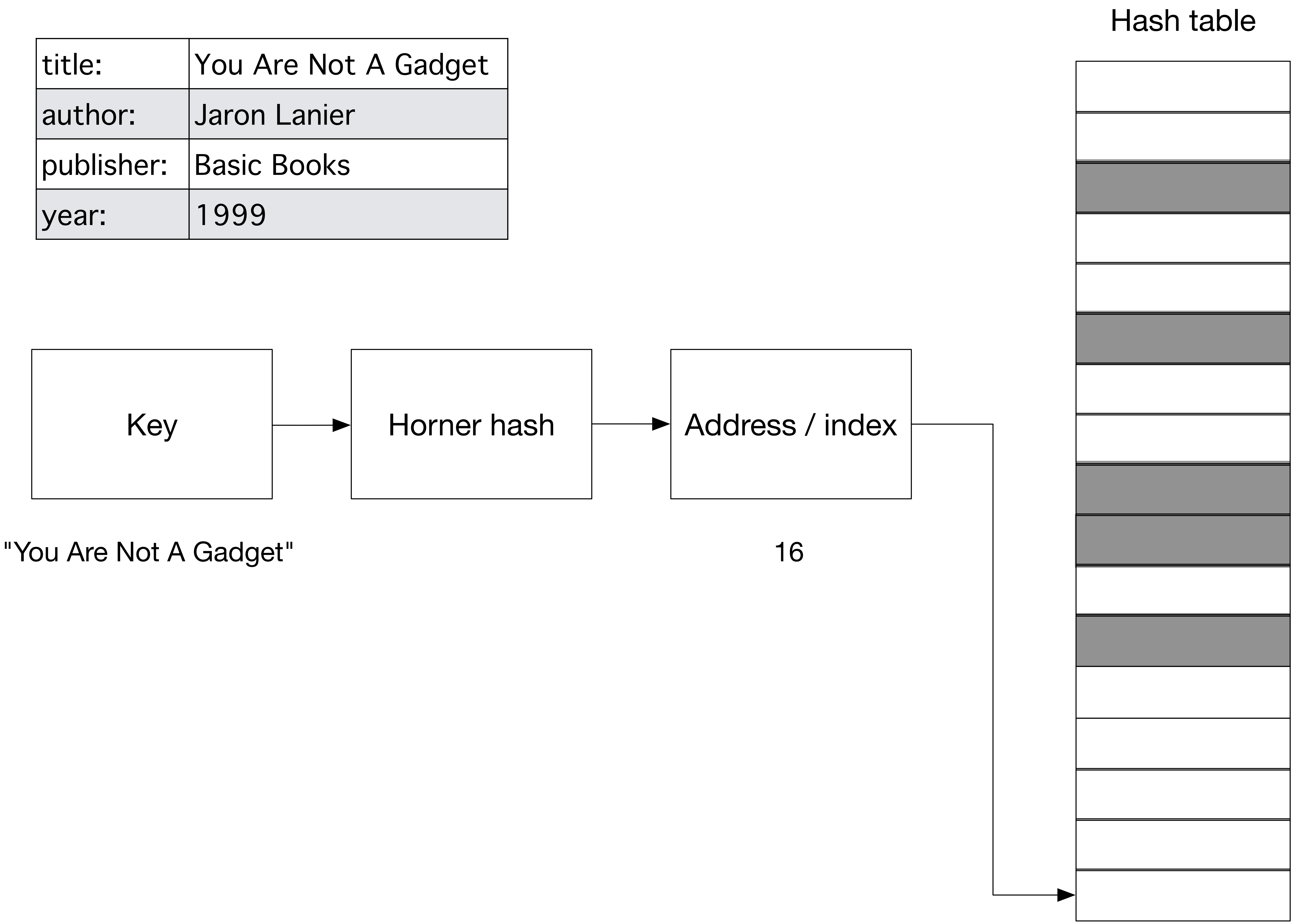
Inserting



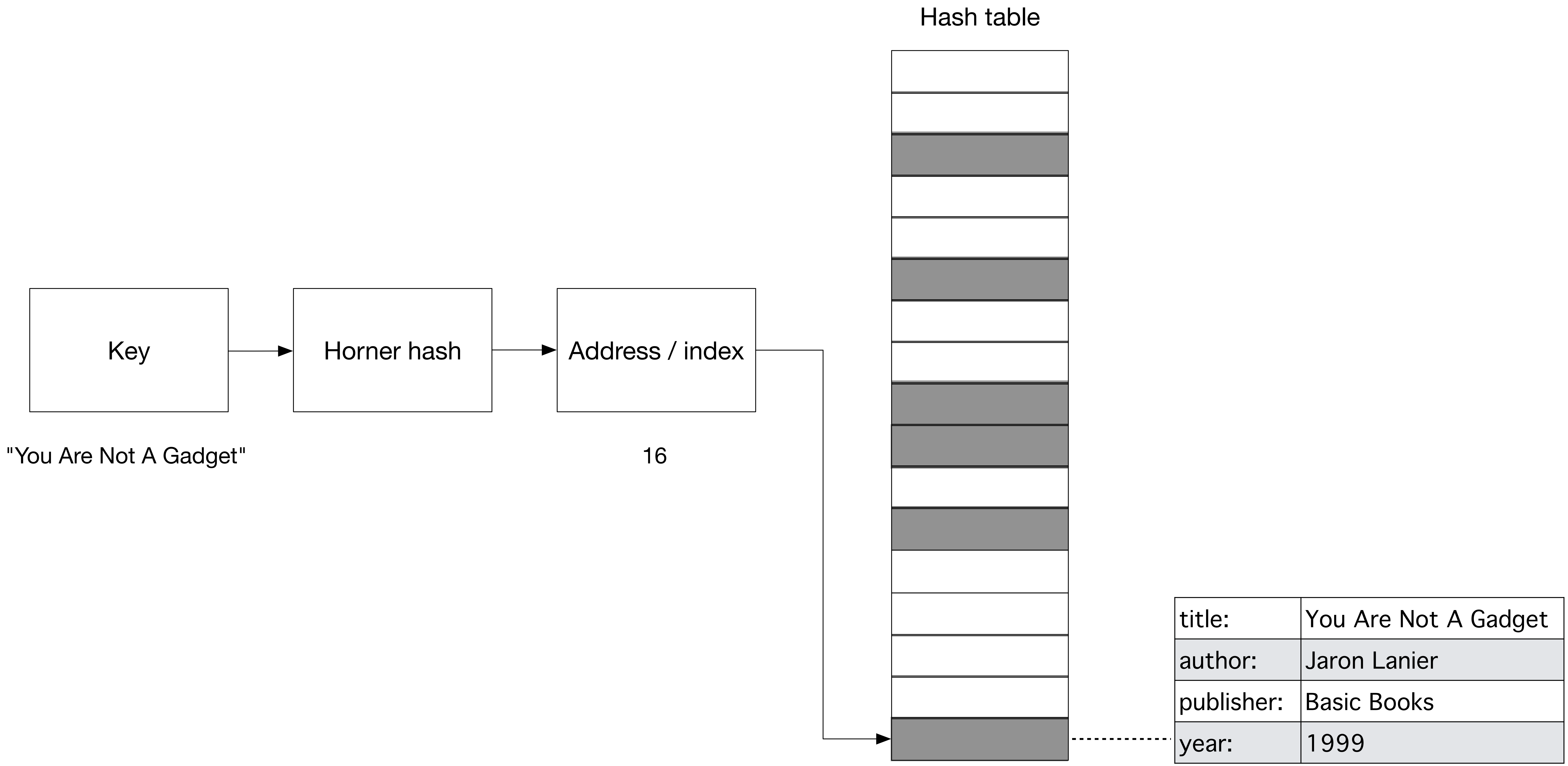
title:	Symbiotic Planet
author:	Lynn Margulis
publisher:	Basic Books
year:	1999

Inserting

title:	You Are Not A Gadget
author:	Jaron Lanier
publisher:	Basic Books
year:	1999



Inserting



title:	You Are Not A Gadget
author:	Jaron Lanier
publisher:	Basic Books
year:	1999

Inserting - what's the complexity?

What's the complexity of inserting into a hash table?

We have to calculate the hash value: constant time.

We have to perform the insert into the vector: constant time.

Inserting - what's the complexity?

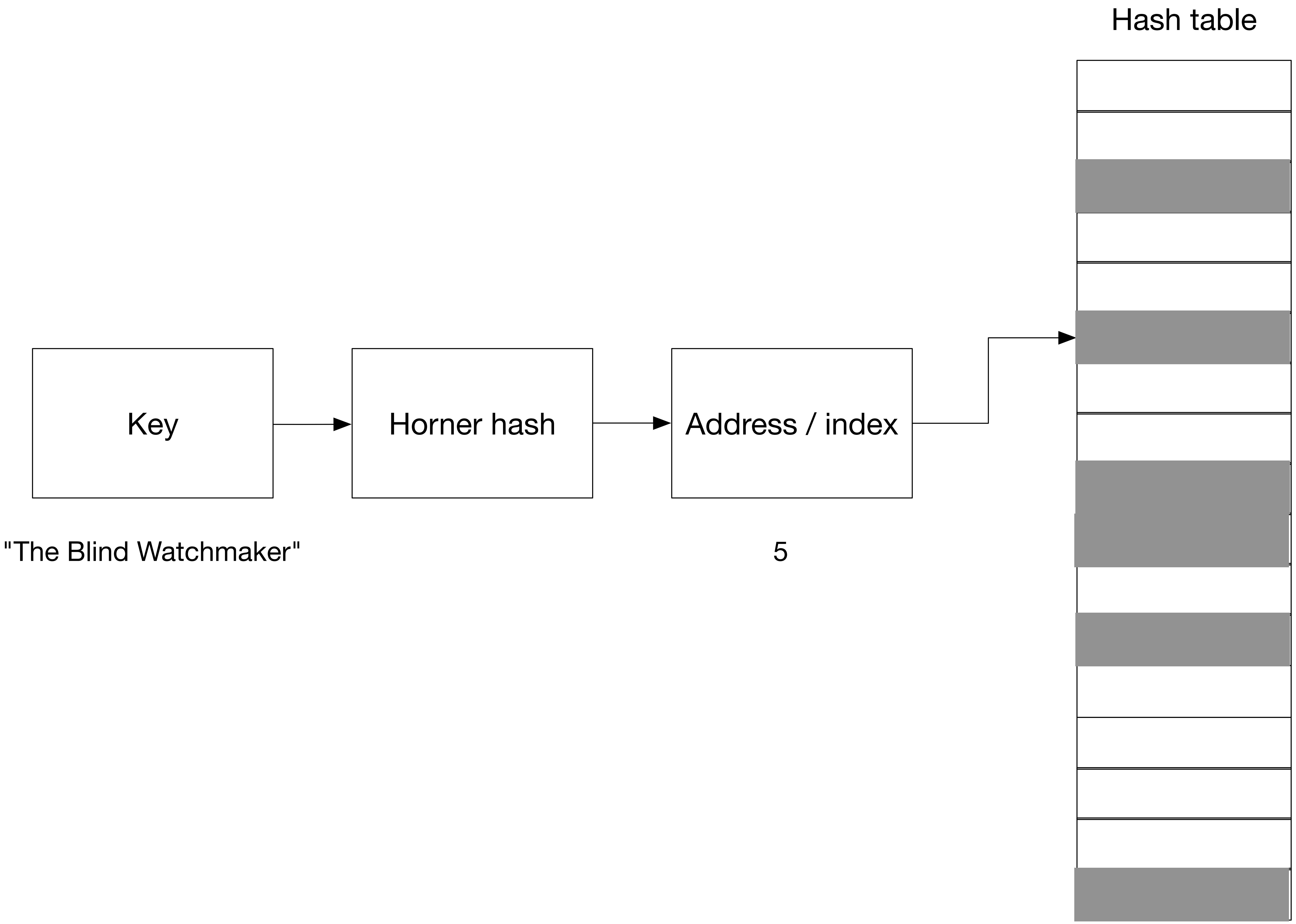
What's the complexity of inserting into a hash table?

We have to calculate the hash value: constant time.

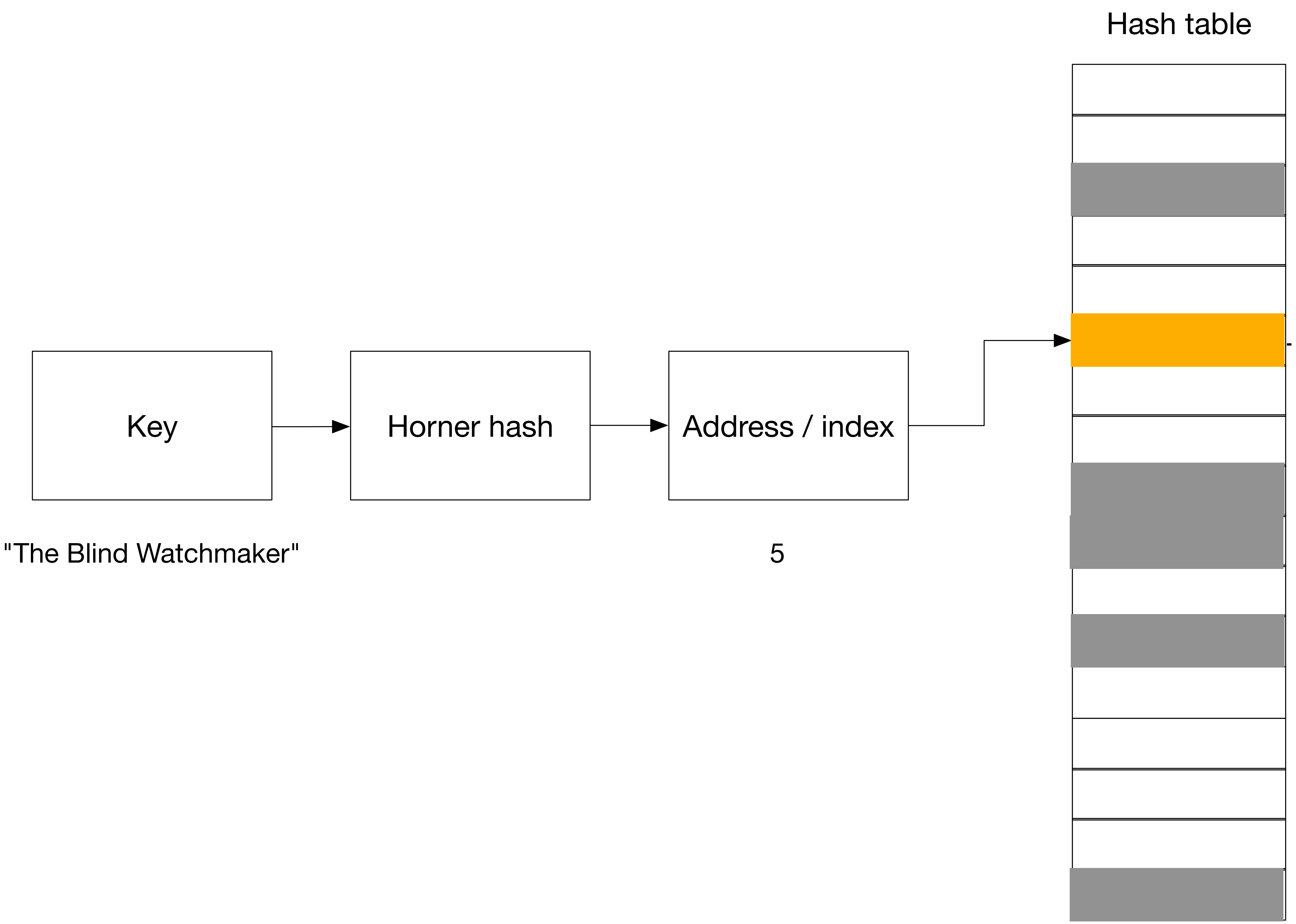
We have to perform the insert into the vector: constant time.

$$\mathcal{O}(1)$$

Finding


Hash table
Key
Horner hash
Address / index
"The Blind Watchmaker"
5

Finding



Hash table

Key

Horner hash

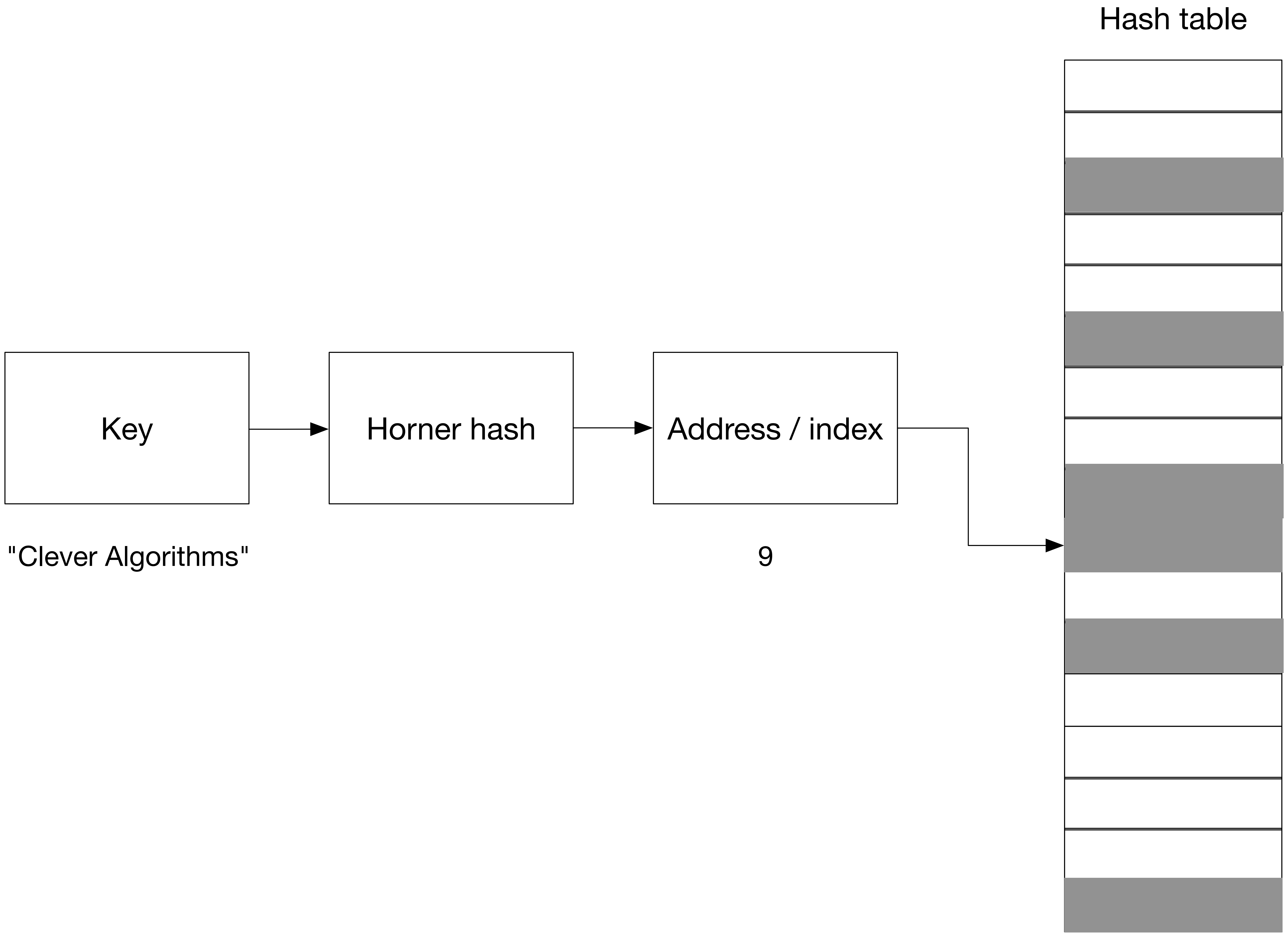
Address / index

"The Blind Watchmaker"

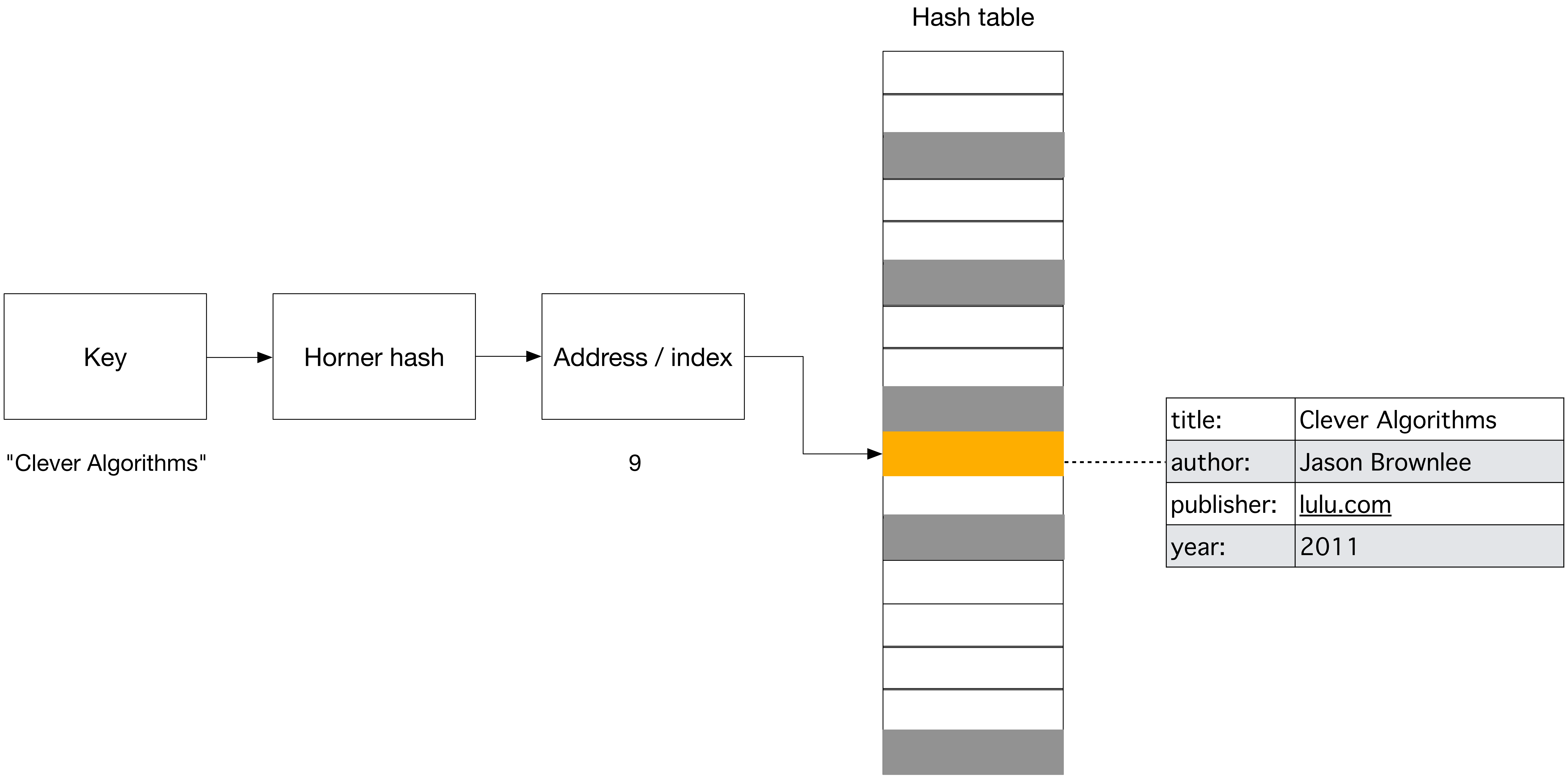
5

title:	The Blind Watchmaker
author:	Richard Dawkins
publisher:	W. W. Norton & Co.
year:	1986

Finding

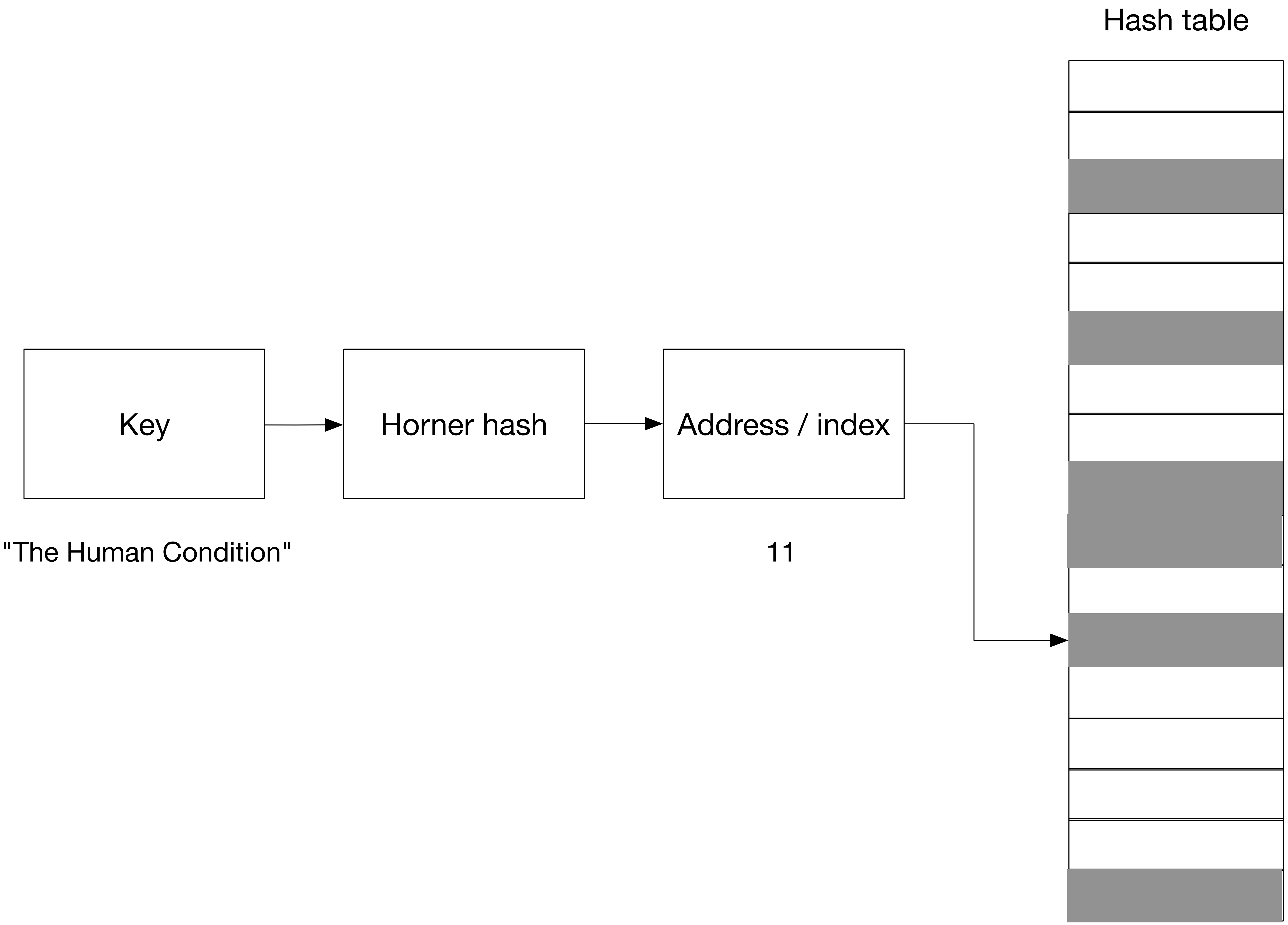

Hash table
Key
Horner hash
Address / index
"Clever Algorithms"
9

Finding



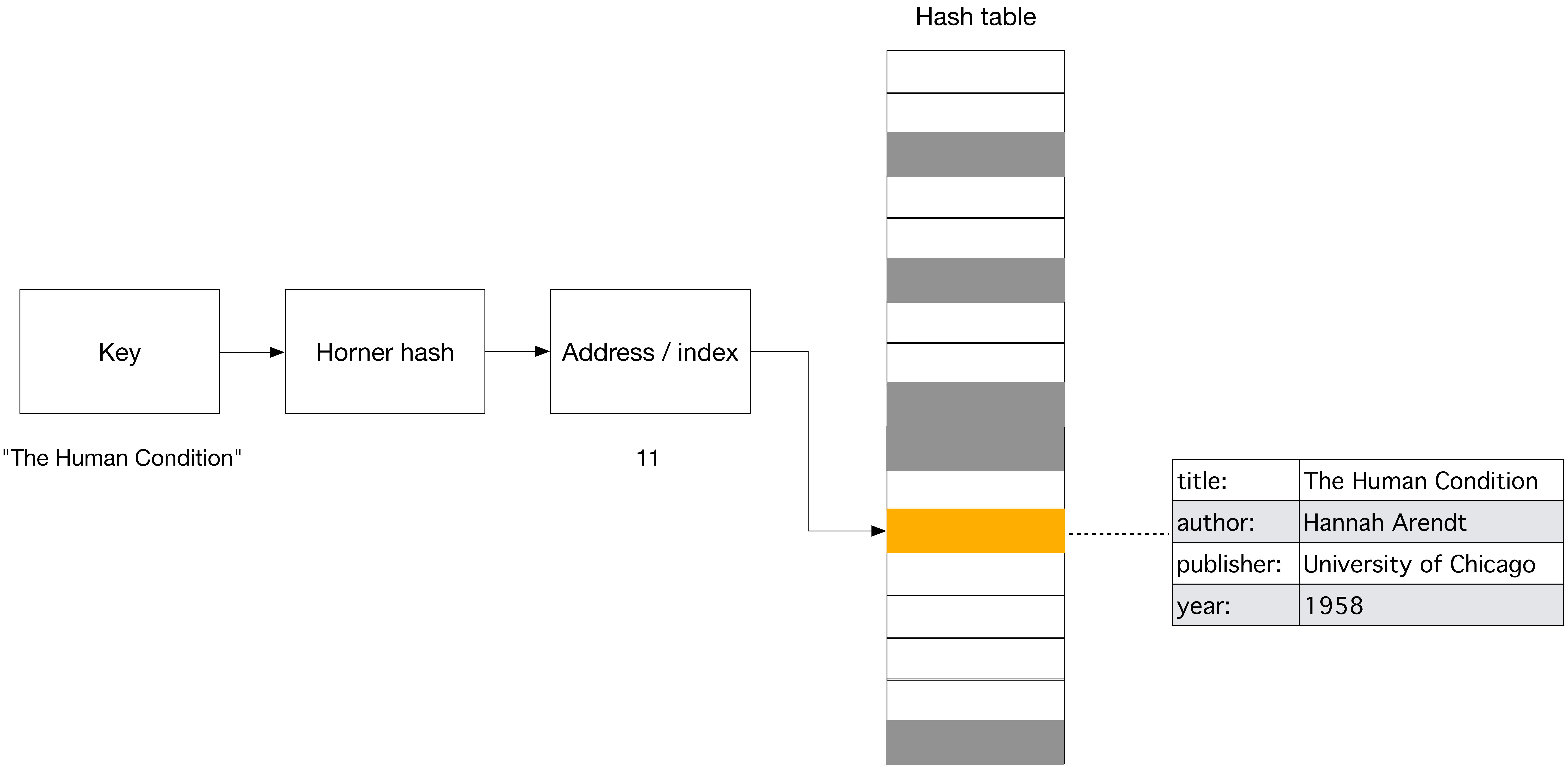
title:	Clever Algorithms
author:	Jason Brownlee
publisher:	lulu.com
year:	2011

Finding



Hash table
Key
Horner hash
Address / index
"The Human Condition"
11

Finding



Hash table

Key

Horner hash

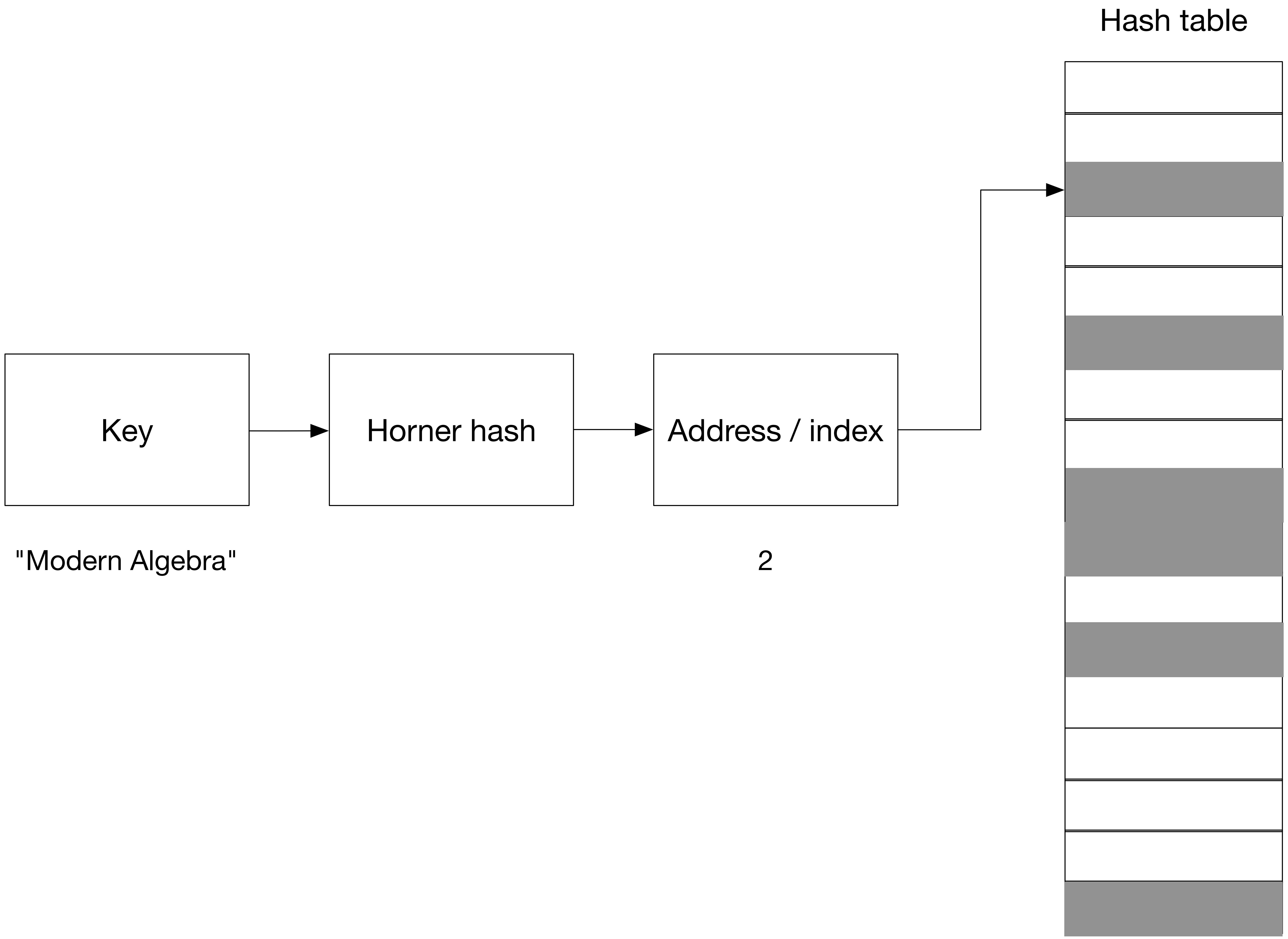
Address / index

"The Human Condition"

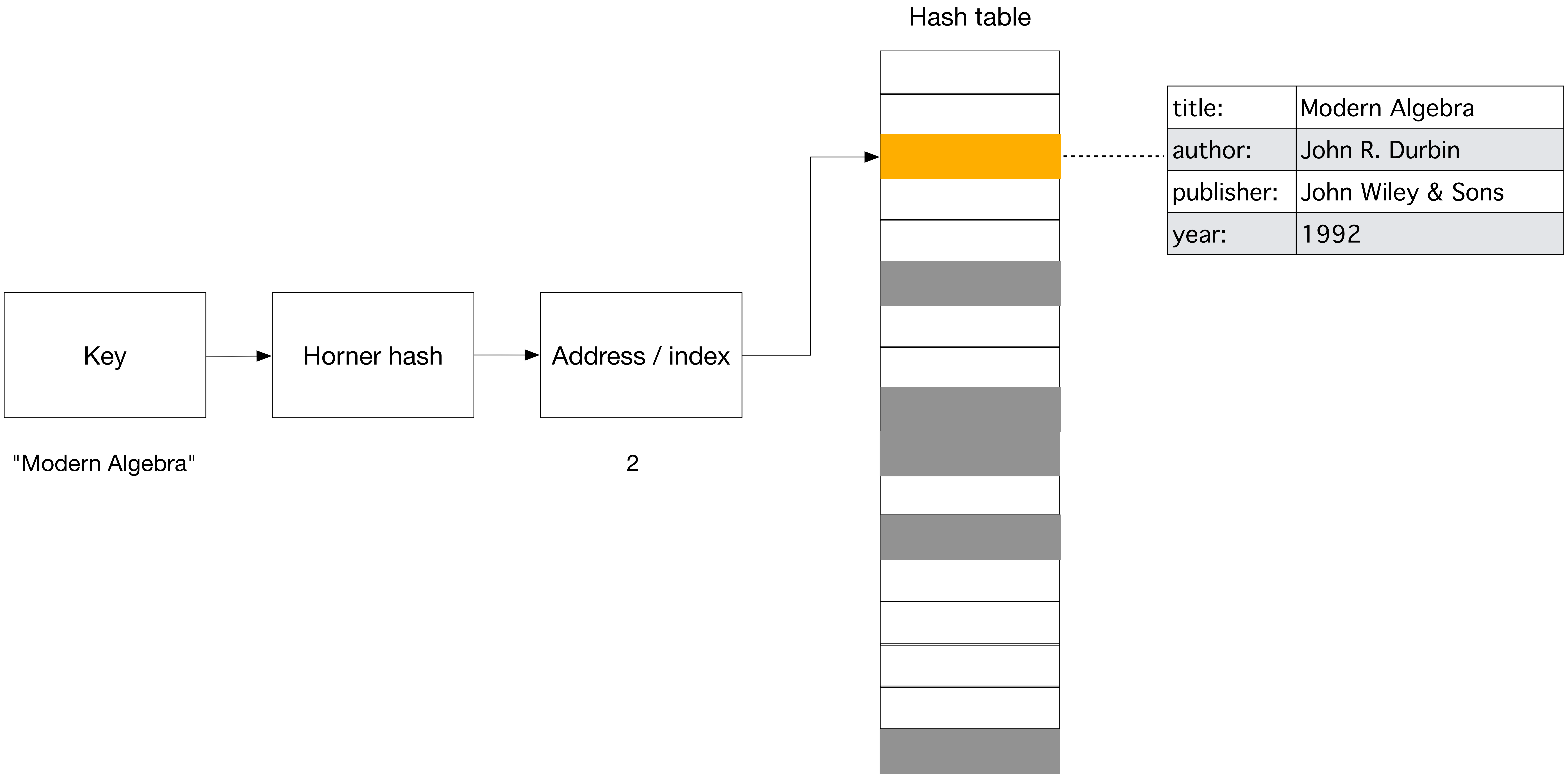
11

title:	The Human Condition
author:	Hannah Arendt
publisher:	University of Chicago
year:	1958

Finding

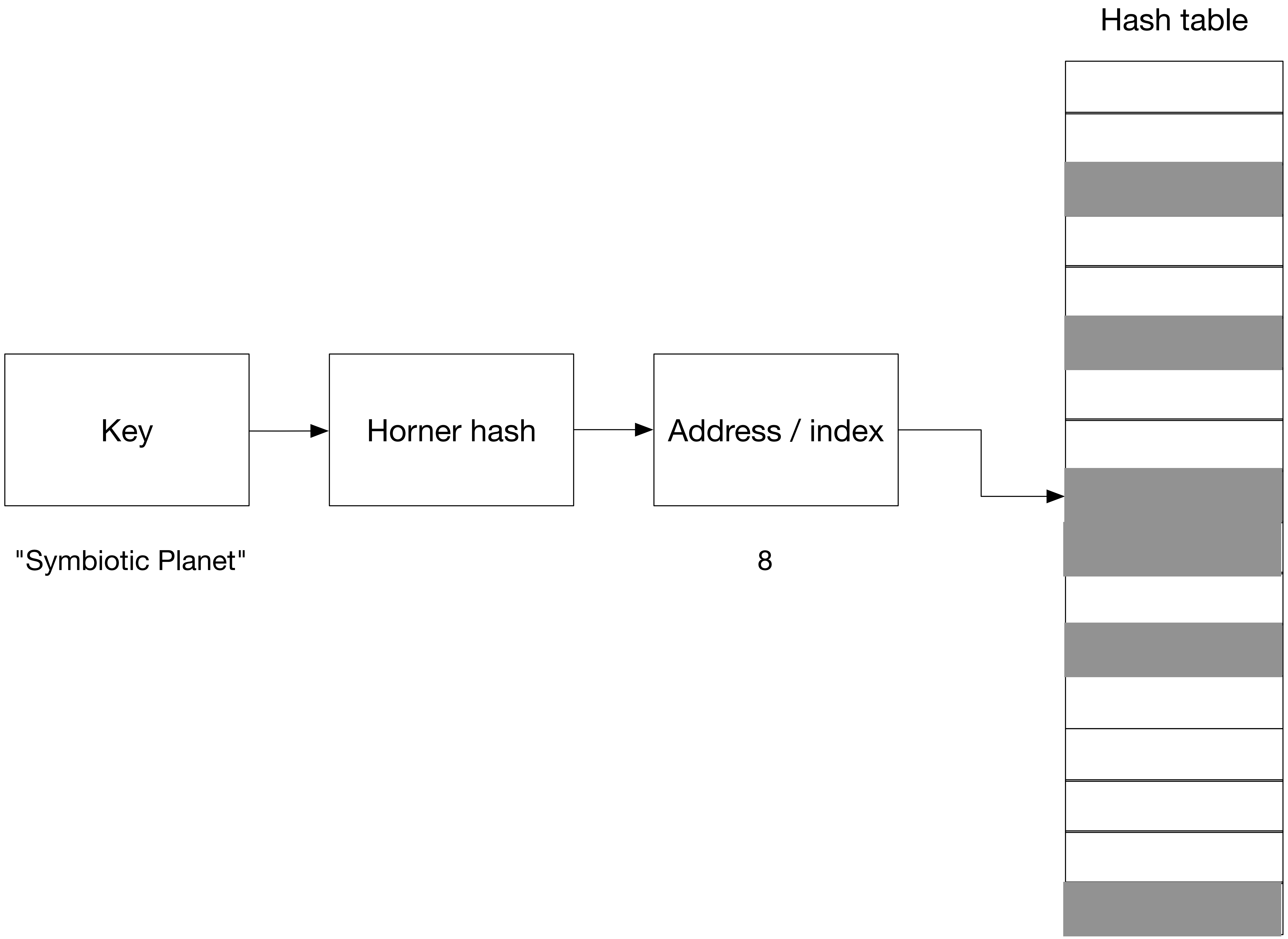


Finding

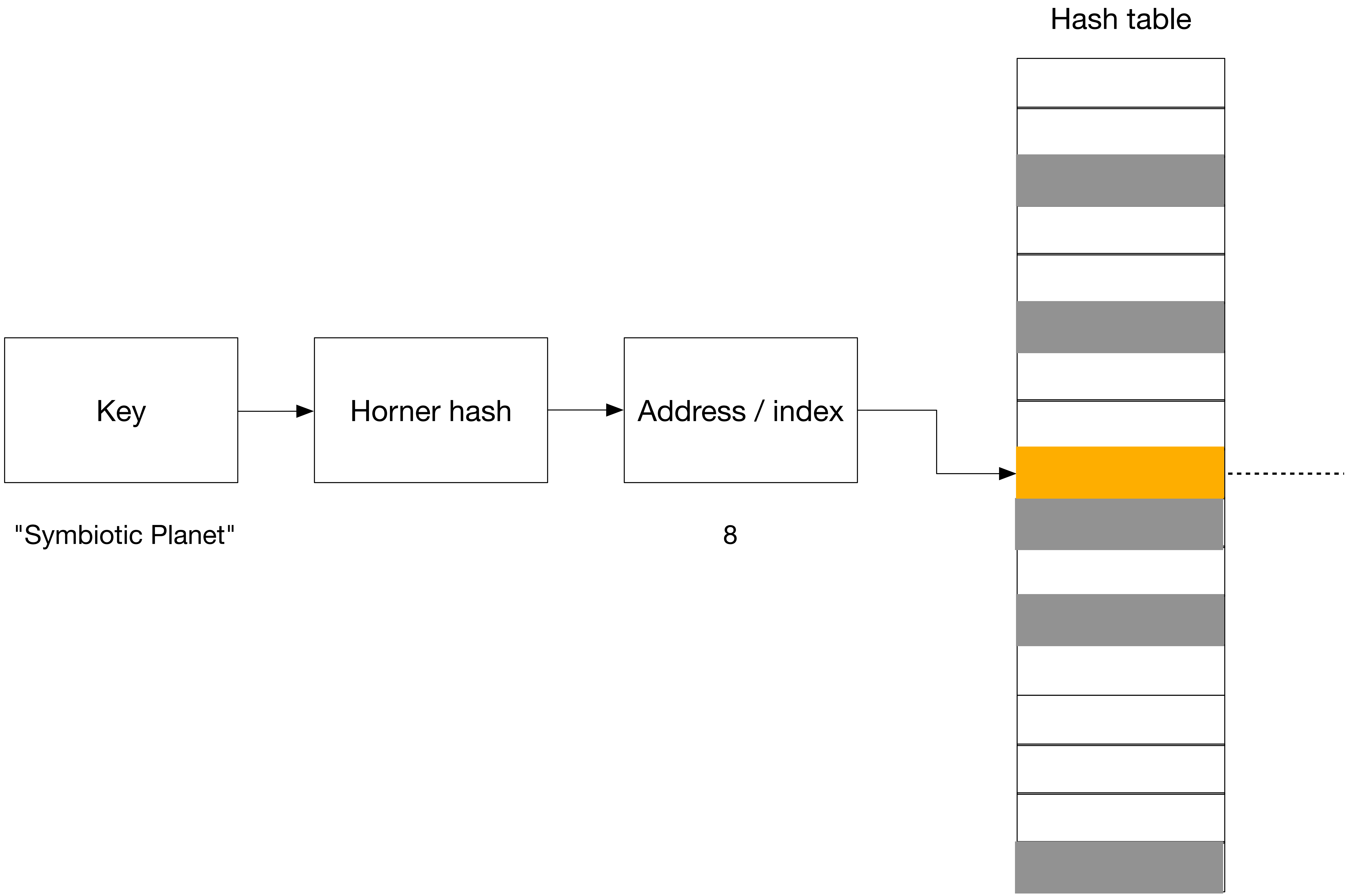


title:	Modern Algebra
author:	John R. Durbin
publisher:	John Wiley & Sons
year:	1992

Finding


Hash table
Key
Horner hash
Address / index
"Symbiotic Planet"
8

Finding



Hash table

Key

Horner hash

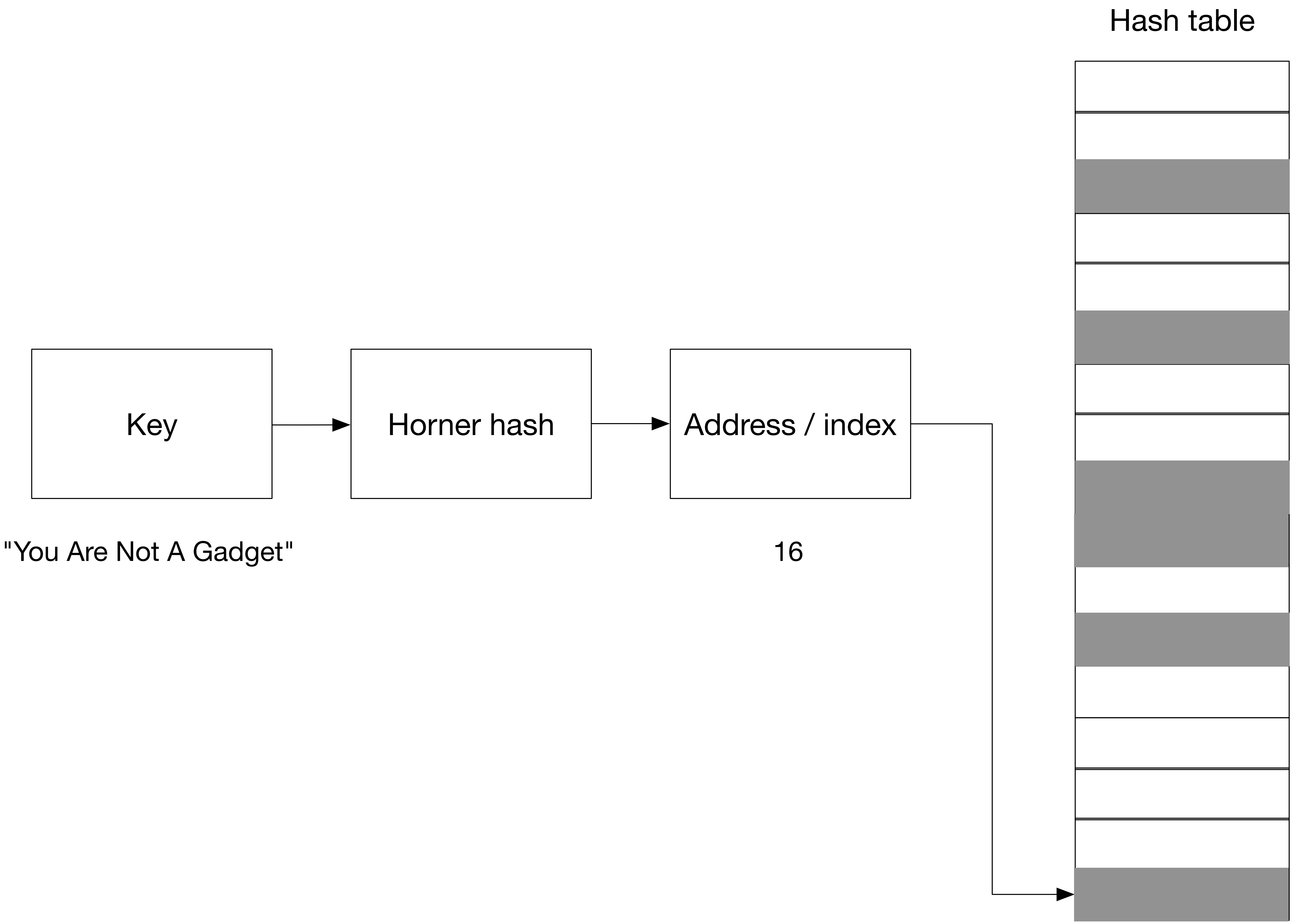
Address / index

"Symbiotic Planet"

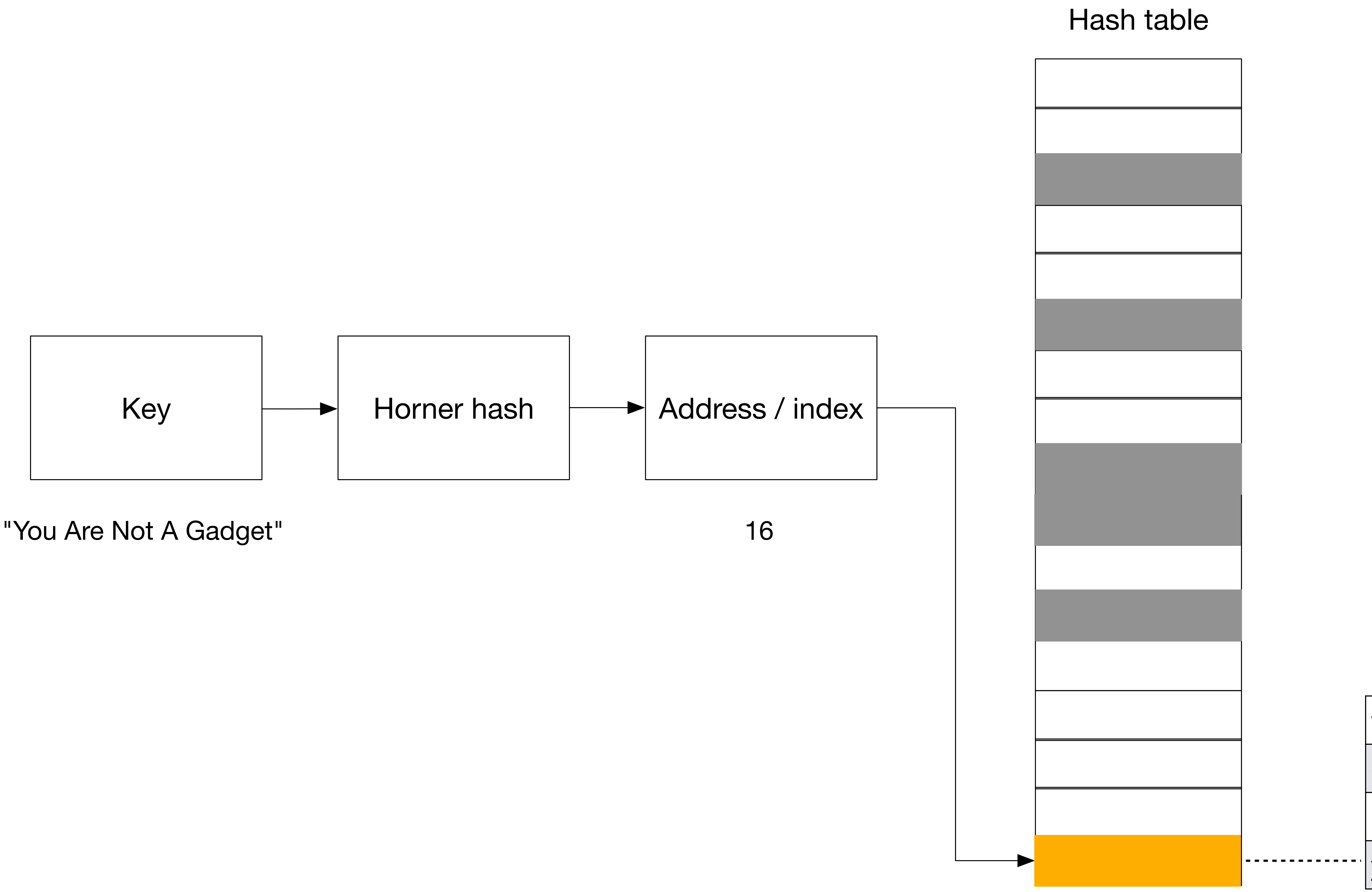
8

title:	Symbiotic Planet
author:	Lynn Margulis
publisher:	Basic Books
year:	1999

Finding

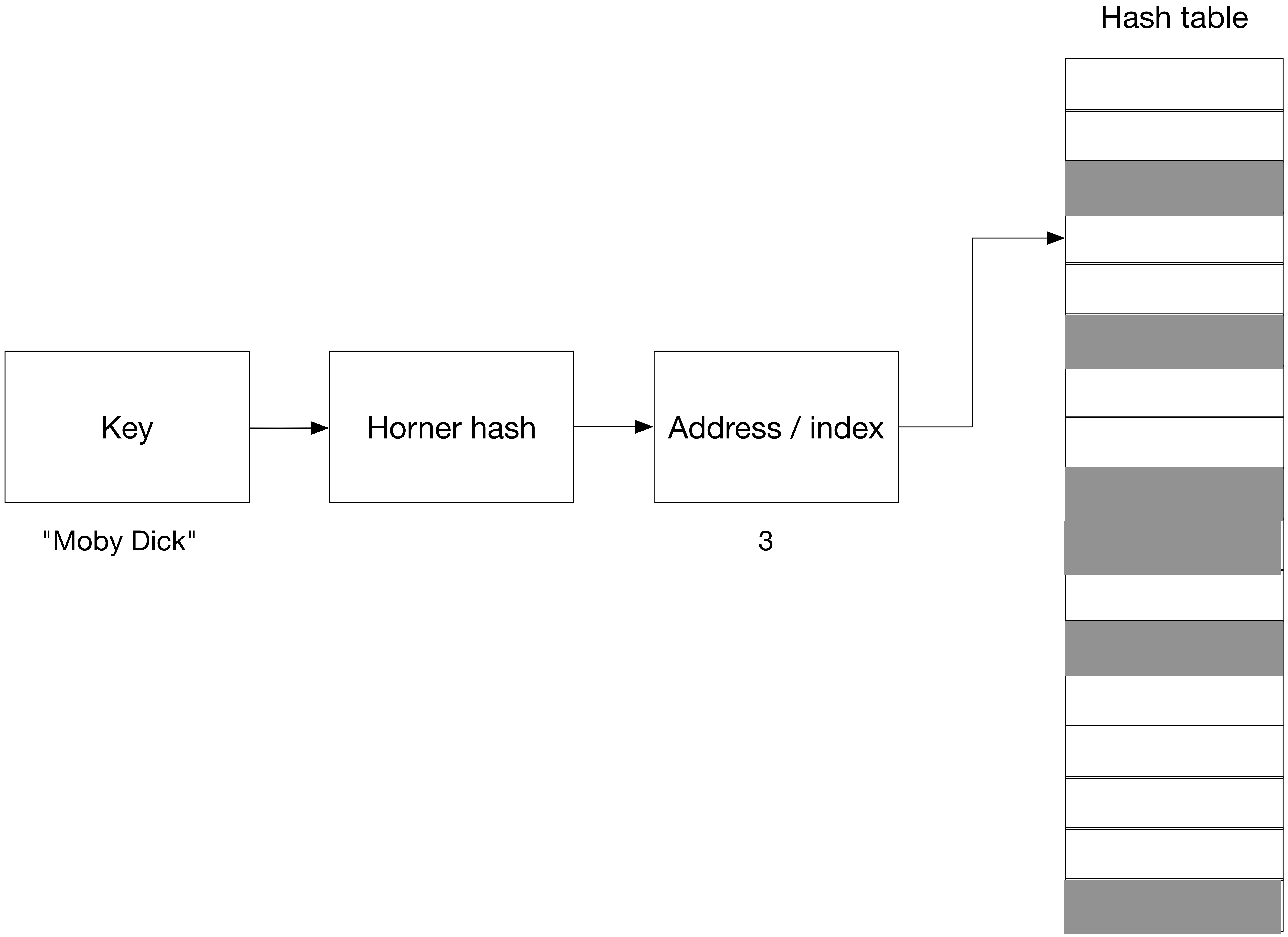

Hash table
Key
Horner hash
Address / index
"You Are Not A Gadget"
16

Finding

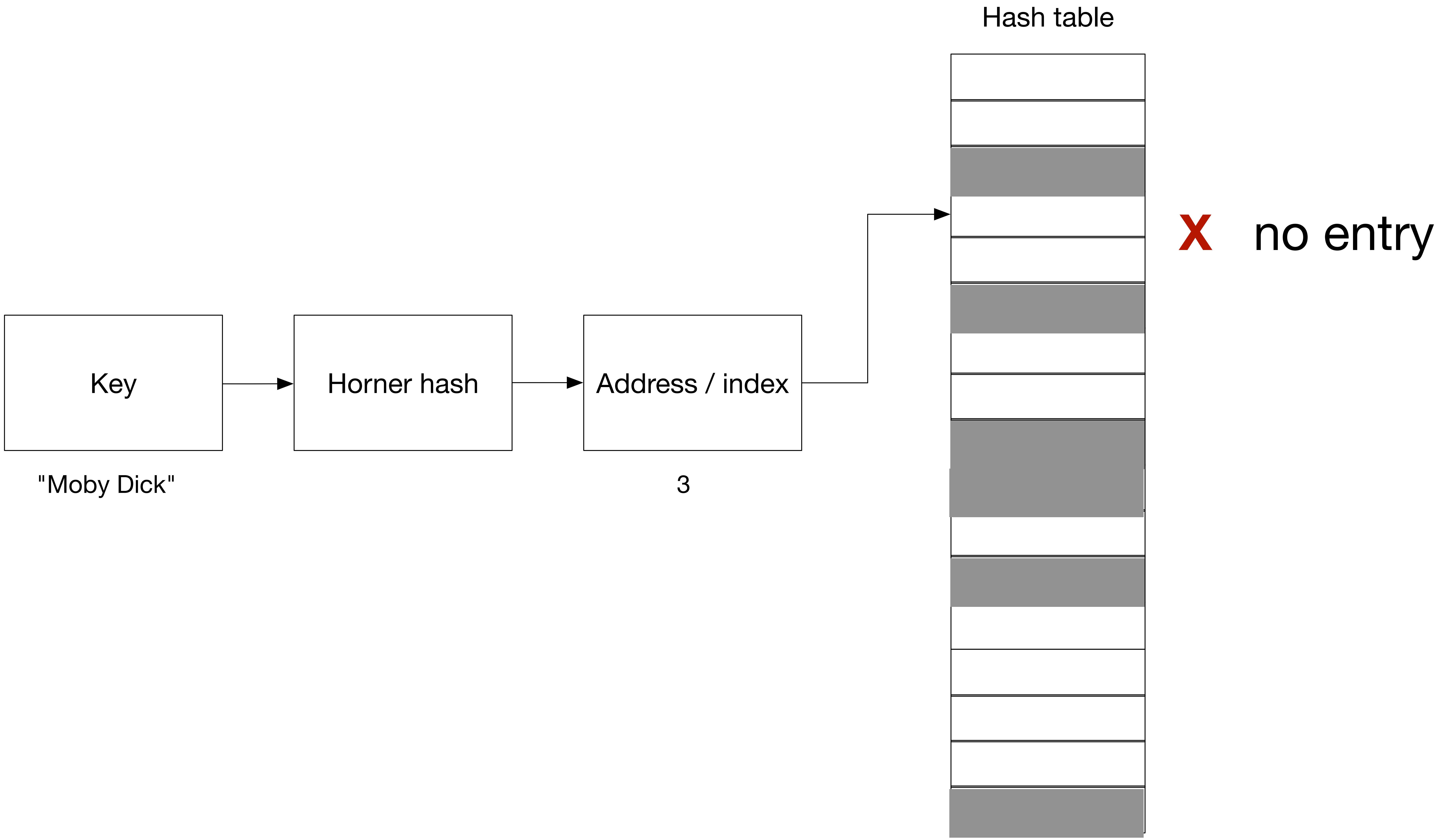


title:	You Are Not A Gadget
author:	Jaron Lanier
publisher:	Basic Books
year:	1999

Finding



Finding


Hash table
Key
"Moby Dick"
Horner hash
Address / index
3
X
no entry

Finding - what's the complexity?

What's the complexity of finding an element in a hash table?

We have to calculate the hash value: constant time.

We have to fetch the item stored at that index in the vector: constant time.

Finding - what's the complexity?

What's the complexity of finding an element in a hash table?

We have to calculate the hash value: constant time.

We have to fetch the item stored at that index in the vector: constant time.

$$\mathcal{O}(1)$$

Deleting - what's the complexity?

What's the complexity of finding an element in a hash table?

We have to calculate the hash value: constant time.

We have to delete the item stored at that index in the vector: constant time.

$$\mathcal{O}(1)$$

What have we left out?

There are quite a few implementation details we've left out but the most important thing we've left out of our discussion so far is: what to do when hashing two different keys yields the same value? This is a challenge for hash tables called "*hash collisions*" or just "*collisions.*"

We'll learn more about collisions and what to do when they occur in future lectures. It turns out there are many different strategies -- called "*collision resolution policies*," and we'll look at some of the most common ones.

Summary

- A hash table allows for $O(1)$ insertion, access, and deletion of objects.
- A hash table requires a hash function that turns keys into indices.
- Choosing a good hash function is important:
  - We want values in its range to be as close to uniformly distributed as possible.
  - The hash function should be easy to compute.
- We have yet to address collisions and collision resolution policies.

Some questions

- You'll recall we said it's best if the size of the hash table is a prime number. Can you think of any reason why this might be so?
- Random numbers can be more-or-less uniformly distributed. Why don't we use random numbers for hashing?
- How might we handle collsions when they occur?