ECS32B

Introduction to Data Structures

Hashing

Lecture 18

Announcements

- Homework 4 due Tuesday May 14 at 11:59pm
- Stay Late And Code night will be Monday May 13th from 6:30-9:00pm in 73 Hutchison
- The more interesting the assignment, the more fragile the test script. For HW4 there is a new version of test script for HW4 on Canvas.

Knowing where items are

- You saw how binary search takes advantage of **information about where items are located** in the collection to speed up search.
- In a binary search the relative ordering of items is known and O(log(n)) search can be achieved
- That's huge, by the way. Now you know how to search for someone in the 7.5 billion person world phone book with 33 comparisons or less! That's over 200 million times better than sequential search.

Knowing where items are

- The goal of **hashing** is to take this one step further.
- The goal of hashing is to know **exactly where the item is located most of the time**, so that search can be done in constant time*.
- The Python dictionary type uses hashing to locate keys in the dictionary.

*more precisely, $O(1)$ average time complexity

Hash Table

The data structure that can be searched using hashing is known as a **hash table**.

It is a collection of items which are stored in such a way as to make it easy to find them later.

The locations where items are stored in a hash table are referred to as **slots.**

None	None	None	None	None	None	None	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10	11	12

Hash Table

We can implement a hash table in Python using the built in list type.

In this example, we have a table of size 13 with a slot named 0, a slot named 1, a slot named 2, and so on up to 12.

Initially, the hash table contains no items. So every slot is empty and initialized to the special Python value None

None	None	None	None	None	None	None	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10	11	12

Hash function

A **hash function** gives us a **mapping** between an item and the slot where that item belongs.

If the hash table is size **m**, the hash function h(item) will take an item and give us an index between 0 and m - 1.

For a simple example of a hash function, suppose our items are all integers. We could use the modulo (%) operator to divide the item by the hash table size and get a remainder between 0 and m.

$$h(item) = item \ \% \ m$$

Hash Table

None	None	None	None	None	None	None	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % m

Hash Table

None	None	None	None	None	None	6	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6

Hash Table

13	None	None	None	None	None	6	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6
13	0

Hash Table

13	14	None	None	None	None	6	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6
13	0
14	1

Hash Table

13	14	None	None	None	None	6	None	None	None	None	24	None
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6
13	0
14	1
24	11

Hash Table

13	14	None	None	2097	None	6	None	None	None	None	24	None
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6
13	0
14	1
24	11
2097	4

Hash Table

13	14	None	None	2097	None	6	None	None	None	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6
13	0
14	1
24	11
2097	4
4809	12

Hash Table

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item	h(item) = item % 13
6	6
13	0
14	1
24	11
2097	4
4809	12
1309	9

Hash Table

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

We were able to insert each of those integer values in the hash table in O(1) time.

To search for 2097 we would first compute

$$h(2097) = 4$$

Then look at position 4 to find the item 2097.

We'll look at how to handle exceptions to the best case scenario soon.

Hash Table

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

A hash table usually sacrifices some space to achieve its performance.

The **load factor** is:

$$\frac{\text{number of items in the table}}{\text{number of slots in the table}}$$

Hash Table

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

For the example above **load factor** is:

The table has 7 items in it and a total of 13 slots

So the load factor is 7/13 = 0.54

Collisions

131

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table. What if the item 131 is the next item in our collection, it will have a hash value of 1 since (131%13 == 1)

But 14 also had a hash value of 1, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision**. Clearly, collisions create a problem for the hashing technique. We will discuss them in detail soon.

Perfect Hashing

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

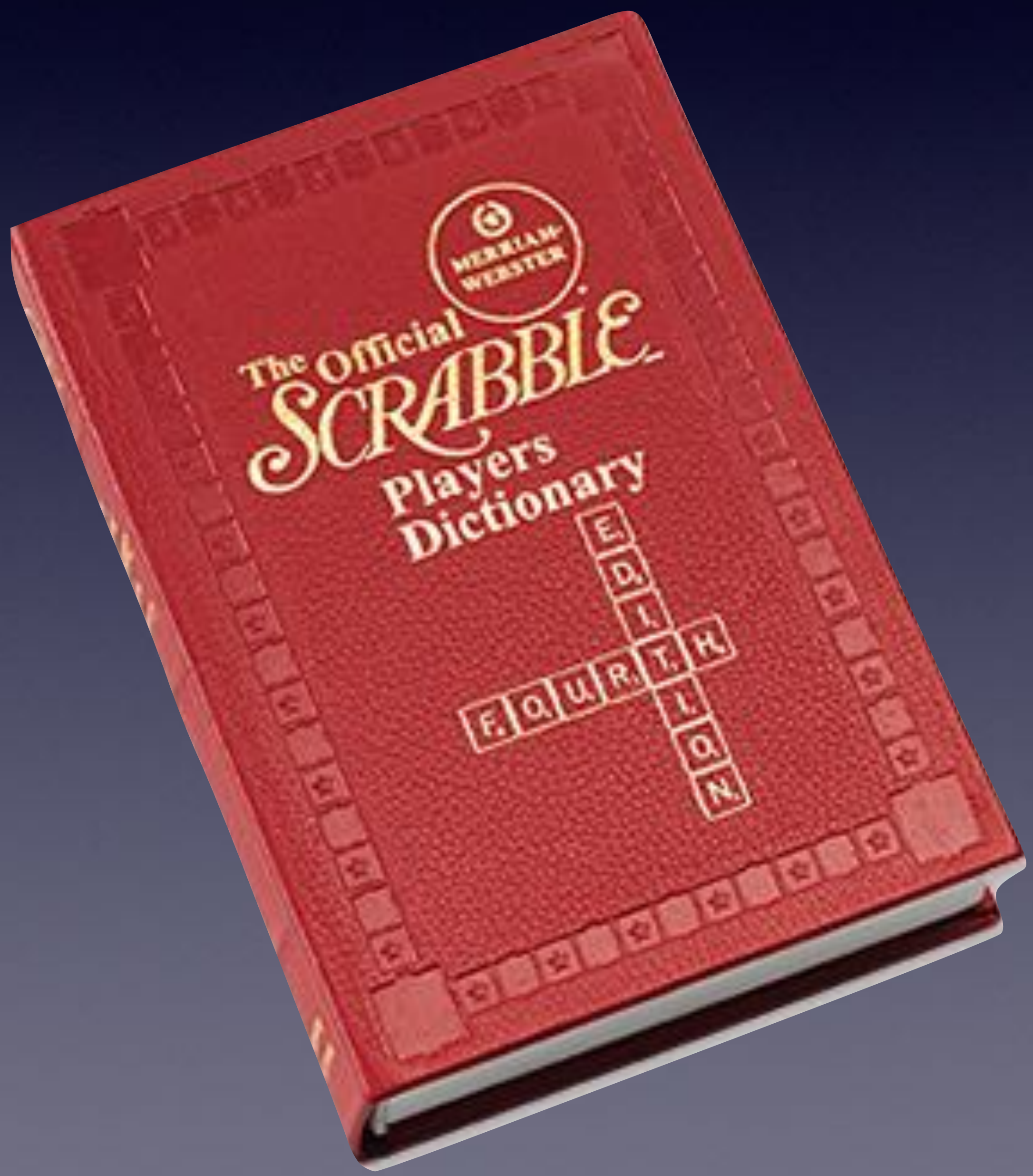
A hash function that maps each item to a unique slot without collision is referred to as a **perfect hash function**. Perfect hashing is possible **only if the items are known in advance**.

One way to have a perfect hash function is to increase the size of the hash table so that each possible value can be accommodated without collisions.

It's impractical for the general case, but luckily, we do not need the hash function to be perfect to still gain performance efficiency….

Hashing Scrabble Dictionary

Let's explore string hashing and dealing with collisions by hashing words from the Scrabble dictionary.



Hashing Strings

In Python we can use the ord() function to map each character to an integer value.

```
>>> ord('s')
115
>>> ord('p')
112
>>> ord('a')
97
>>> ord('m')
109
```

To the left are the integers corresponding to the characters in the string 'spam'

Hashing Strings

In Python we can use the ord() function to map each character to an integer value.

```
>>> ord('a')
97
>>> ord('z')
122
>>> ord('A')
65
>>> ord('Z')
90
```

These values are a-z = 97-122, and A-Z = 65-90. They also determine the sort order of strings.

Hashing Strings

Extending this idea, one way to map any string to an integer value would be to add up the integer values of each character.

```
>>> ord('s')+ord('p')+ord('a')+ord('m')
433
```

Applying the modulo (%) operator to this gives us the final slot

```
>>> 433 % 13
4
```

Hashing Strings

All of that can be done in one line of Python using a list comprehension.

```
>>> sum(ord(c) for c in 'spam') % 13
4
```

This hash function works for any type that can be converted to a string

```
>>> sum(ord(c) for c in str(130) ) % 13
5
>>> sum(ord(c) for c in str(3.14159) ) % 13
6
>>> sum(ord(c) for c in str('(530) 755-5555') ) % 13
7
```

Hashing Strings

What happens with spam and maps?

```
>>> sum(ord(c) for c in 'spam') % 13

>>> sum([115, 112, 97, 109]) % 13

>>> sum(433) % 13
4

>>> sum(ord(c) for c in 'maps') % 13

>>> sum([109, 97, 112, 115]) % 13
```

Hashing Strings

What happens with spam and maps?

>>> sum(ord(c) for c in '**spam**') % 13

>>> sum([115, 112, 97, 109]) % 13

>>> sum(**433**) % 13
4

>>> sum(ord(c) for c in '**maps**') % 13

>>> sum([109, 97, 112, 115]) % 13

>>> sum(**433**) % 13
4

Hashing Strings

Words that use the same letters will have the same hash value

```
>>> sum(ord(c) for c in 'spam') % 13
4
```

```
>>> sum(ord(c) for c in 'maps') % 13
4
```

So anagrams are a source of collisions

Dealing with Collisions

This is called **collision resolution**

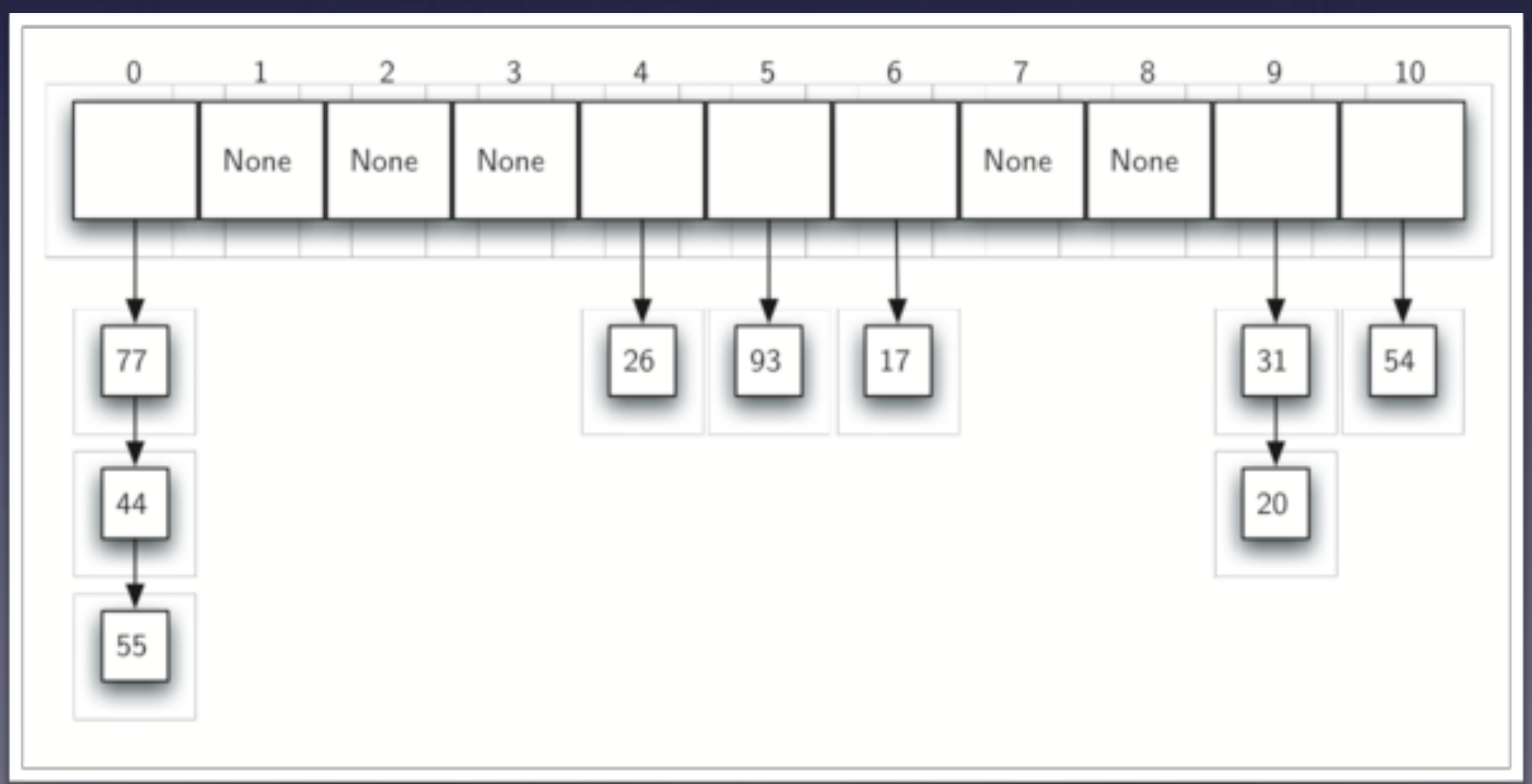
When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. One that allows us to later find the item.

For our scrabble anagrams example we will allow each slot to hold a collection of items.

This is known as **chaining**.

Chaining

The name **chaining** comes from the chain of items that may be present in a hash table slot. Think linked list, although other data structures work, such as the Python list.



Hashing Scrabble Dictionary

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

We will insert each string item into the hash table as the data in a node of a linked list with the slot referencing the head of the linked list.

Item to be inserted →

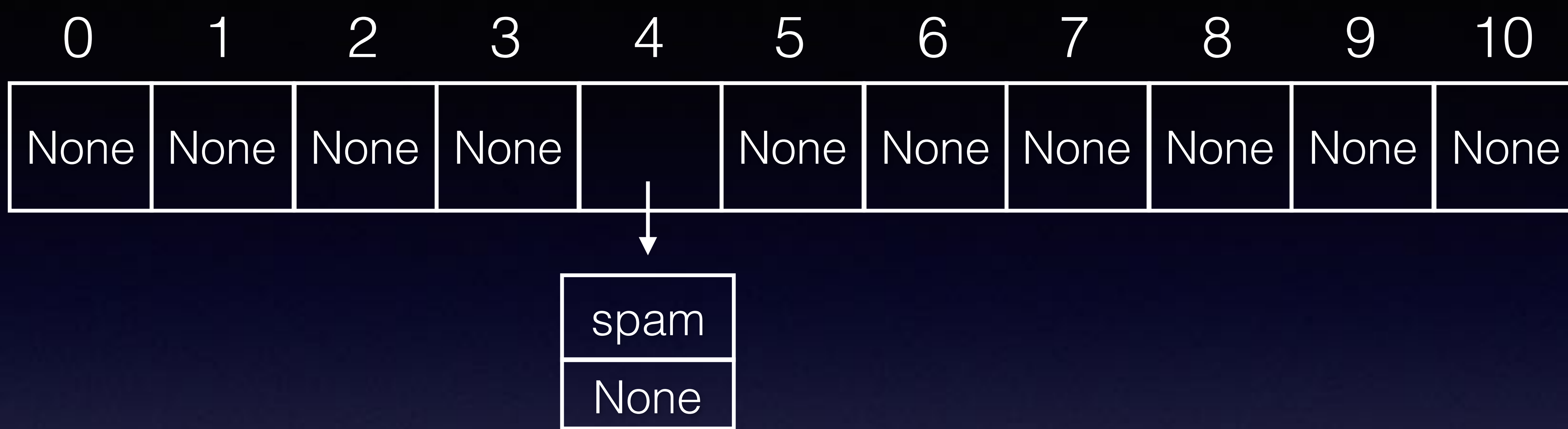
spam
None

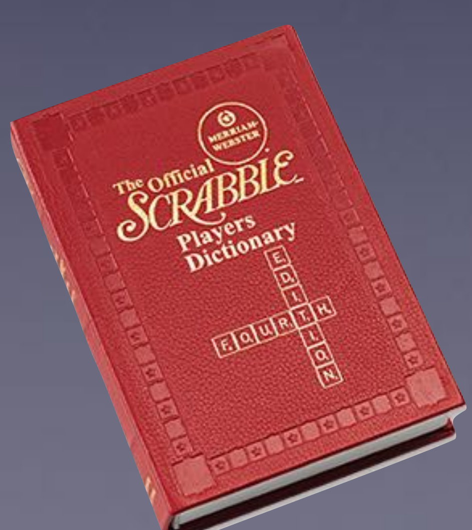
slot=sum(ord(c) for c in 'spam') % 11

slot=sum([115, 112, 97, 109]) % 11

slot=433 % 11

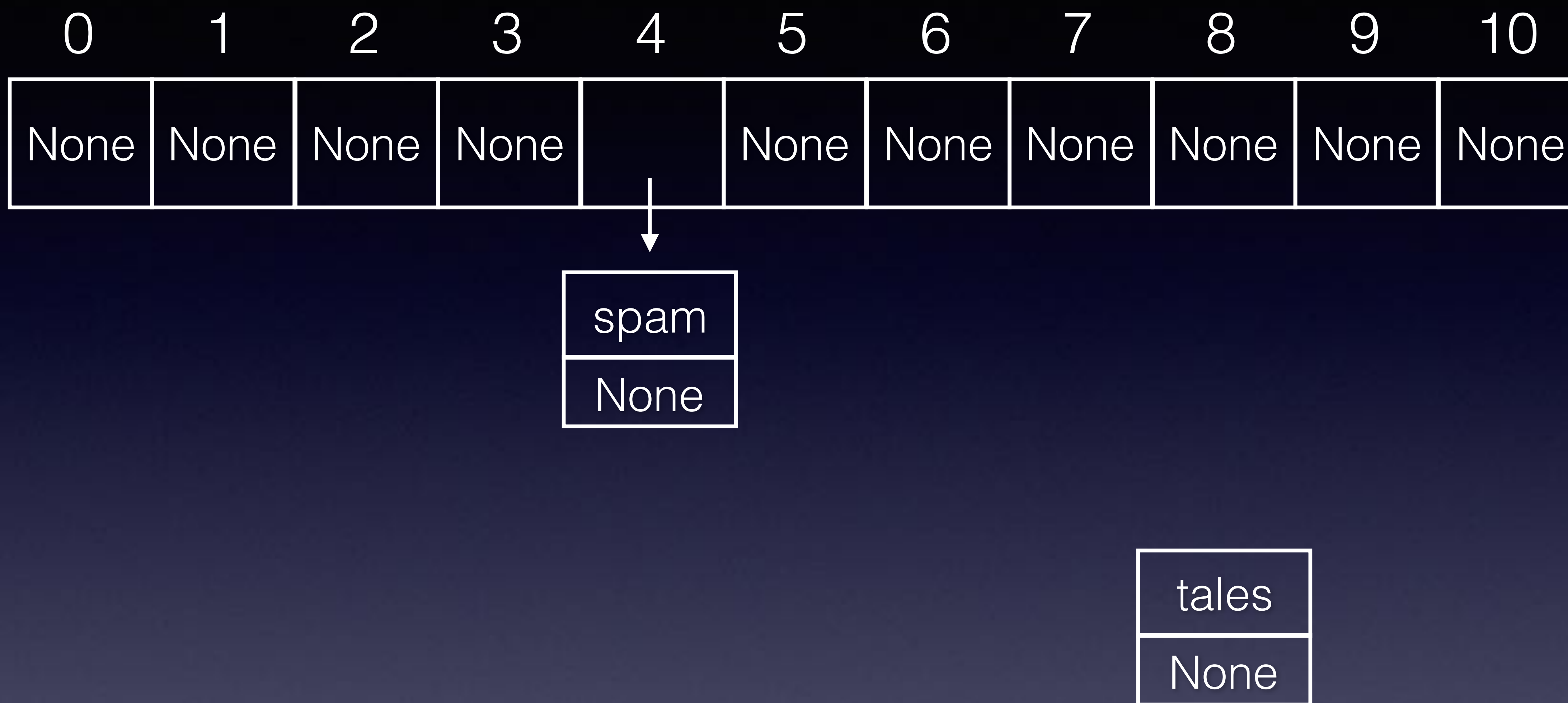
Hashing Scrabble Dictionary

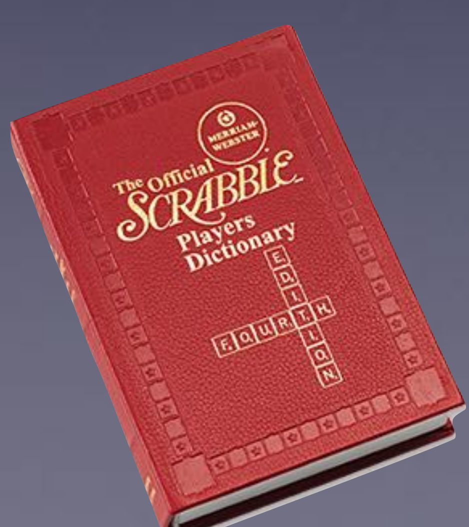




slot=4

Hashing Scrabble Dictionary



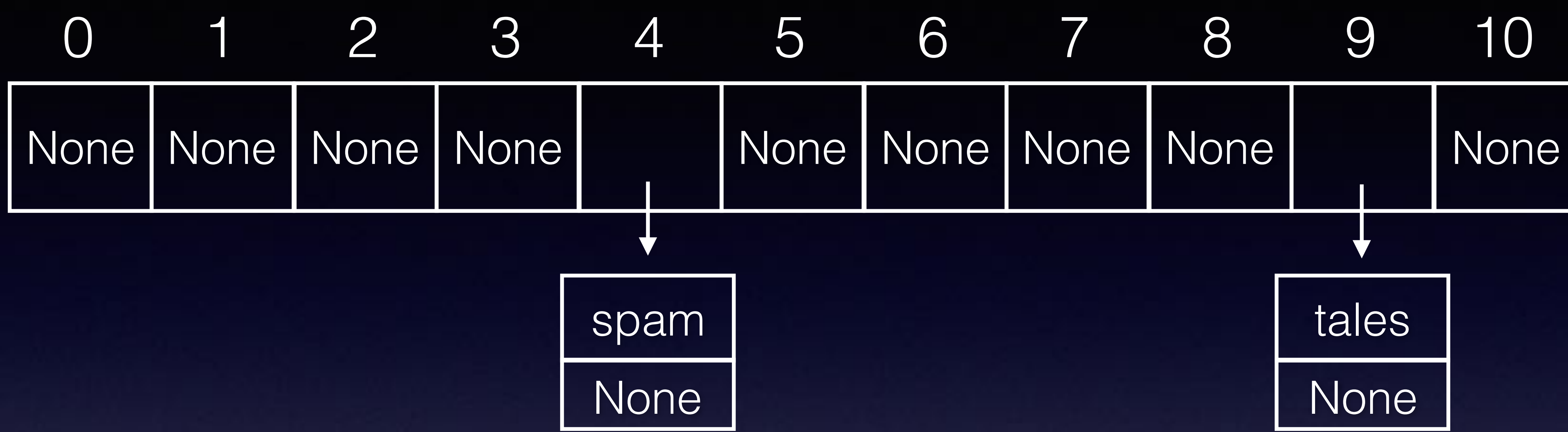


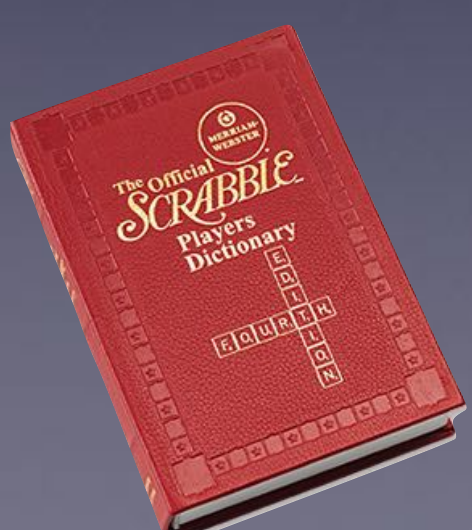
slot=sum(ord(c) for c in 'tales') % 11

slot=sum([116, 97, 108, 101, 115]) % 11

slot=537 % 11

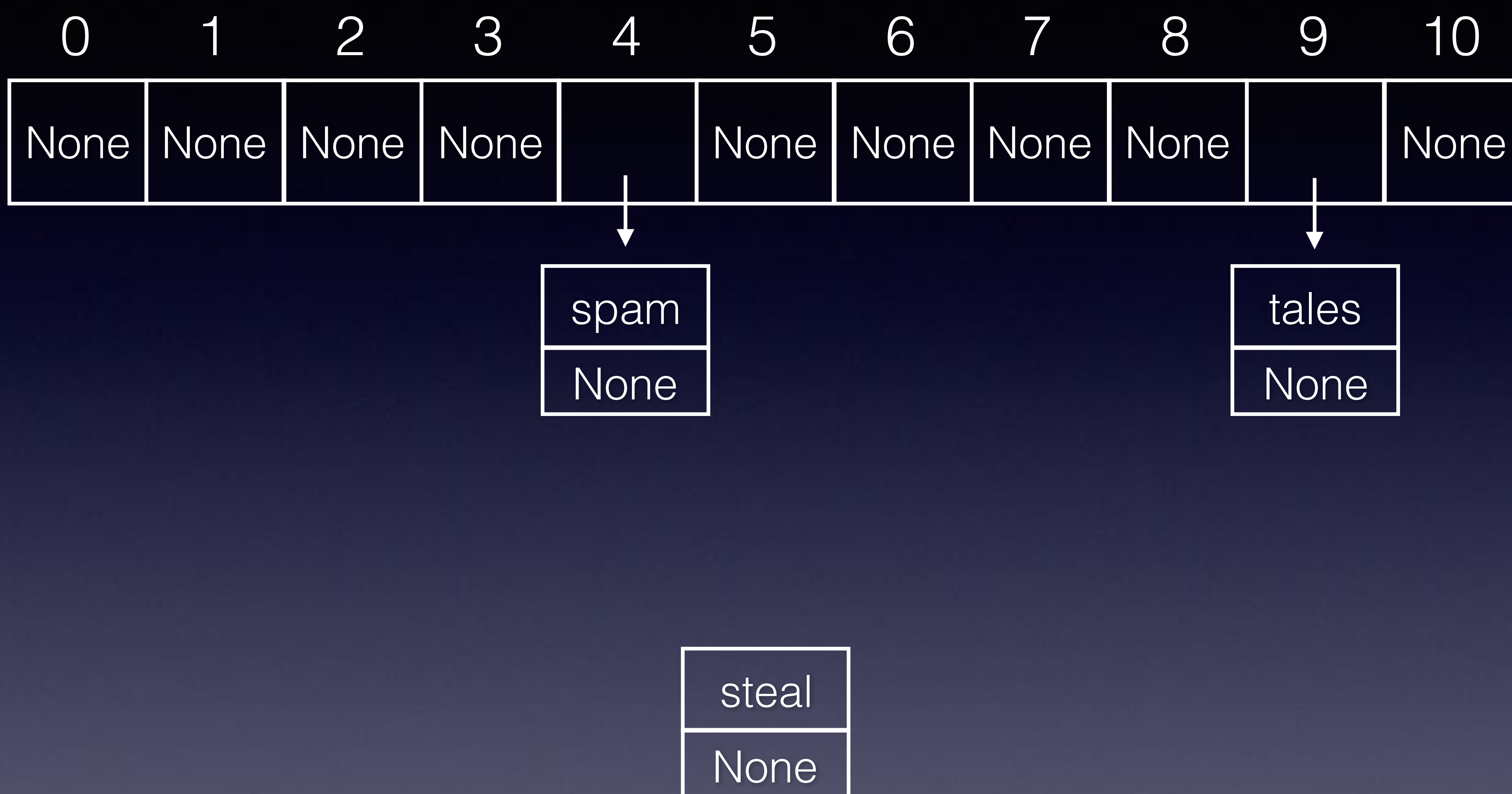
Hashing Scrabble Dictionary

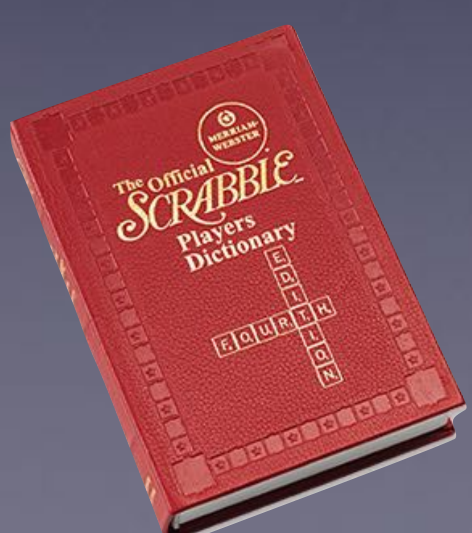




slot=9

Hashing Scrabble Dictionary



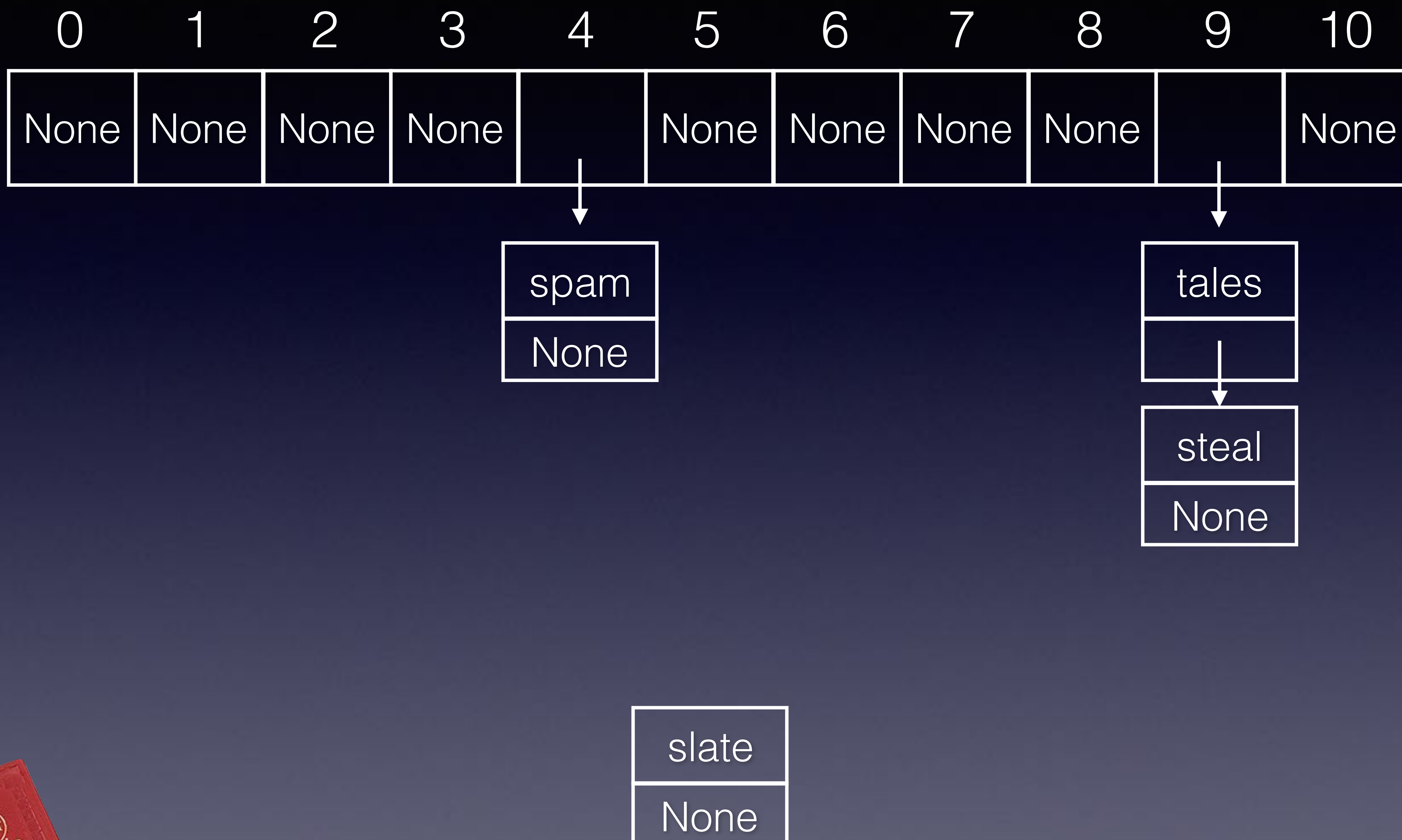


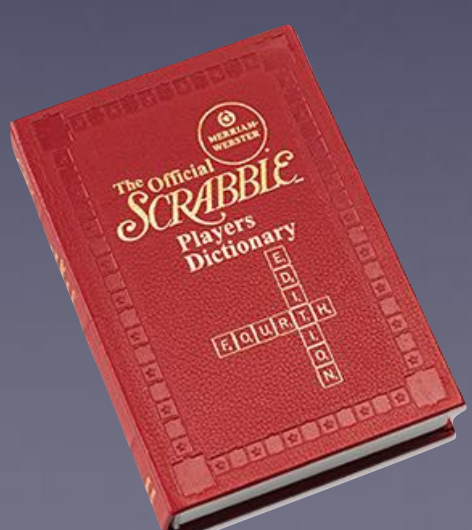
slot=sum(ord(c) for c in 'steal') % 11

slot=sum([115, 116, 101, 97, 108]) % 11

slot=537 % 13

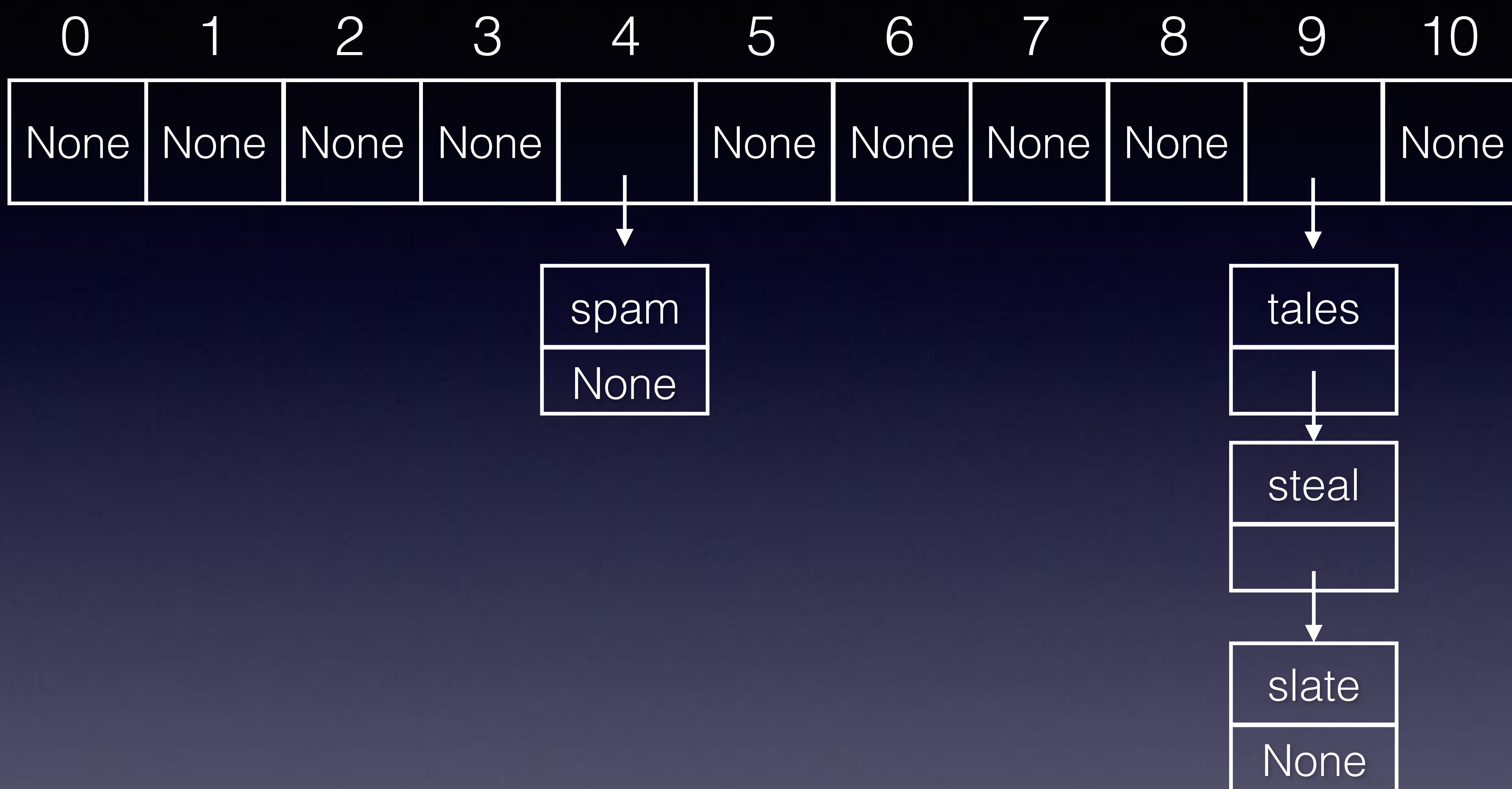
Hashing Scrabble Dictionary

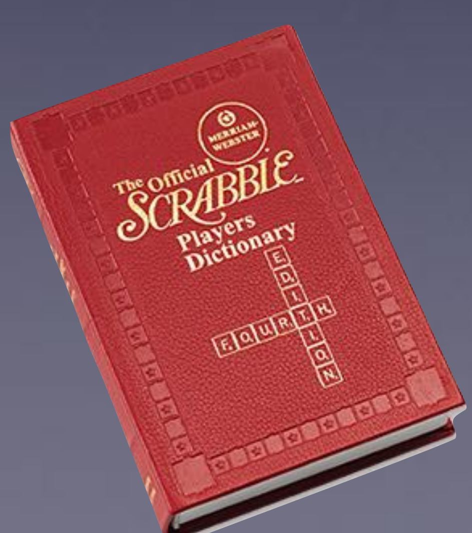




slot=9

Hashing Scrabble Dictionary



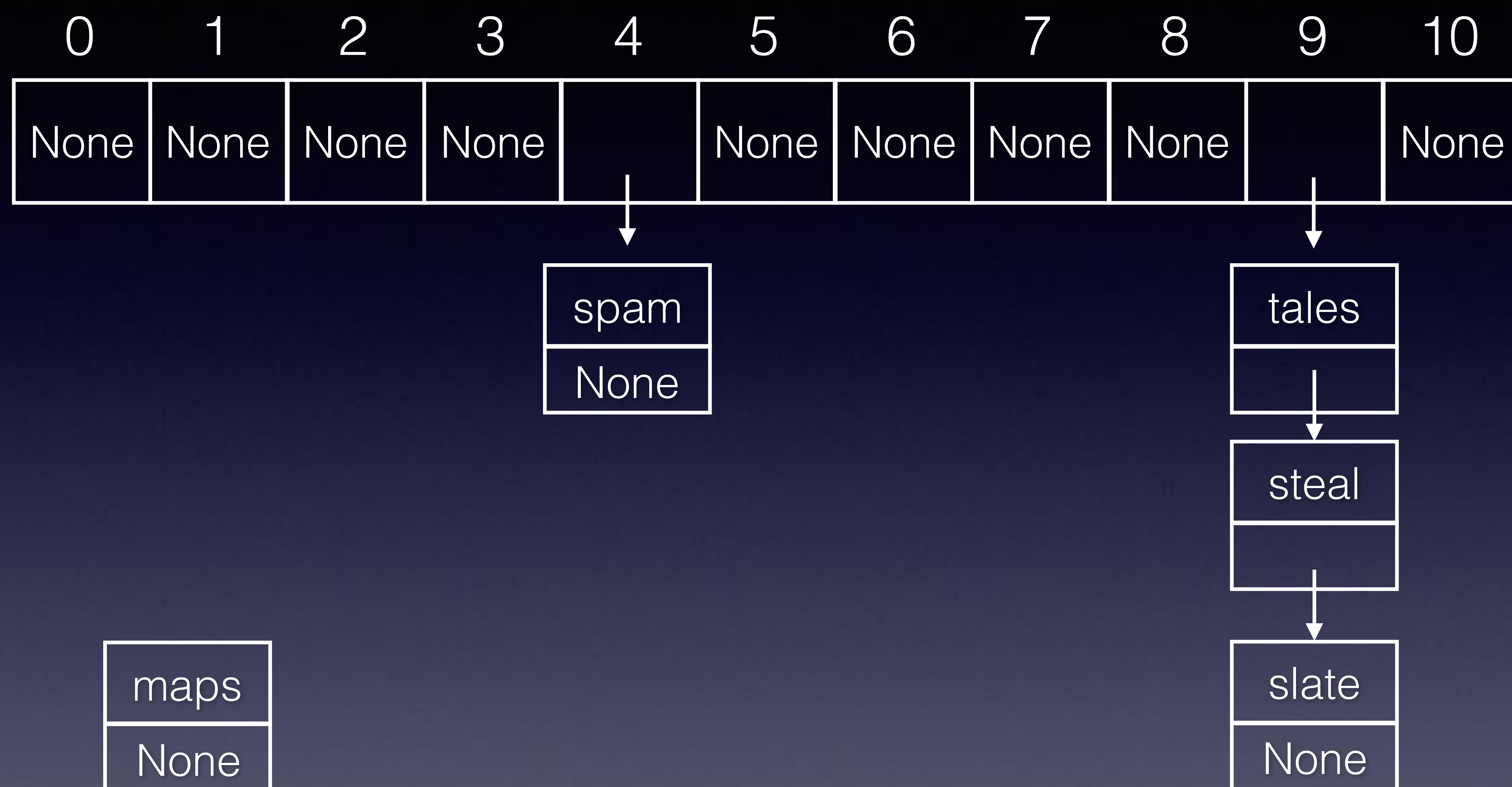


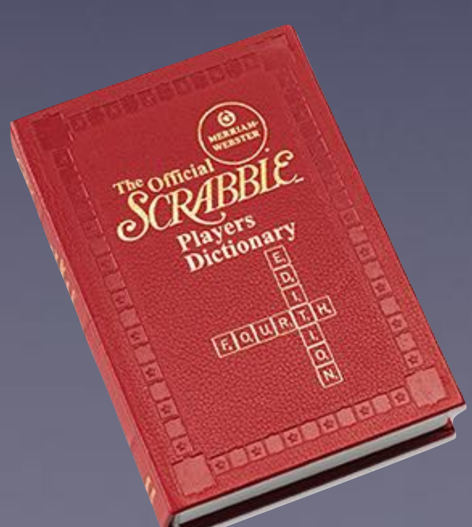
slot=sum(ord(c) for c in 'slate') % 11

slot=537 % 11

slot=9

Hashing Scrabble Dictionary



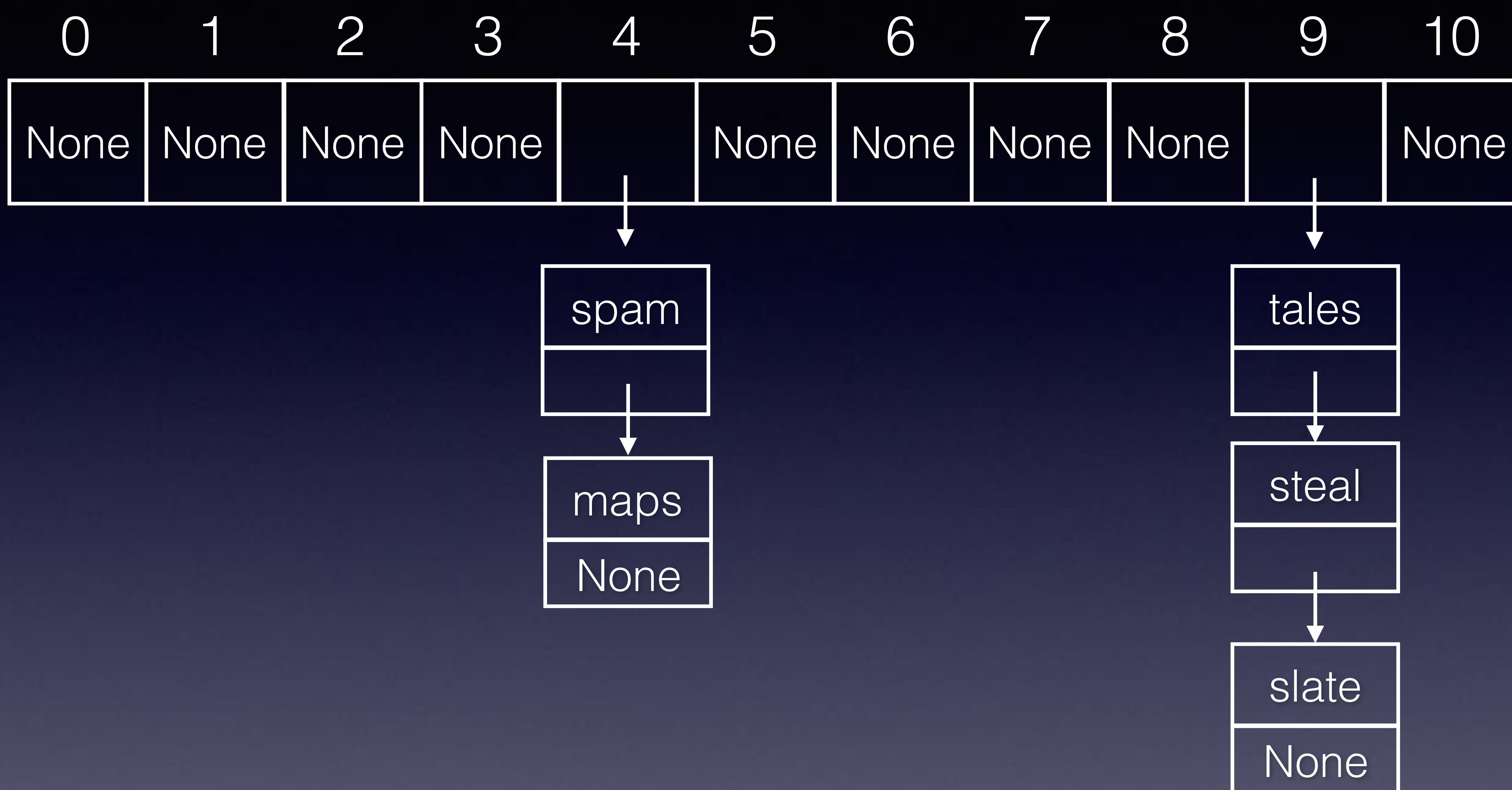


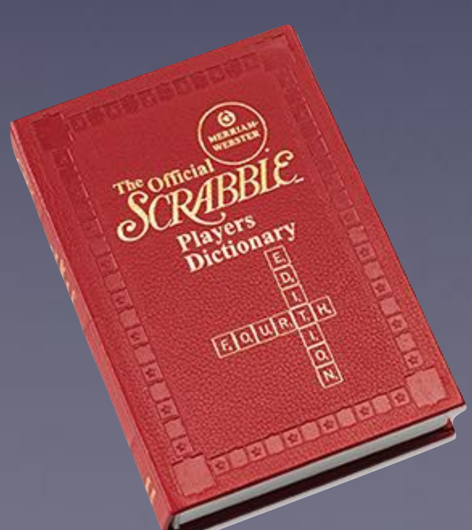
slot=sum(ord(c) for c in 'maps') % 11

slot=433 % 11

slot=4

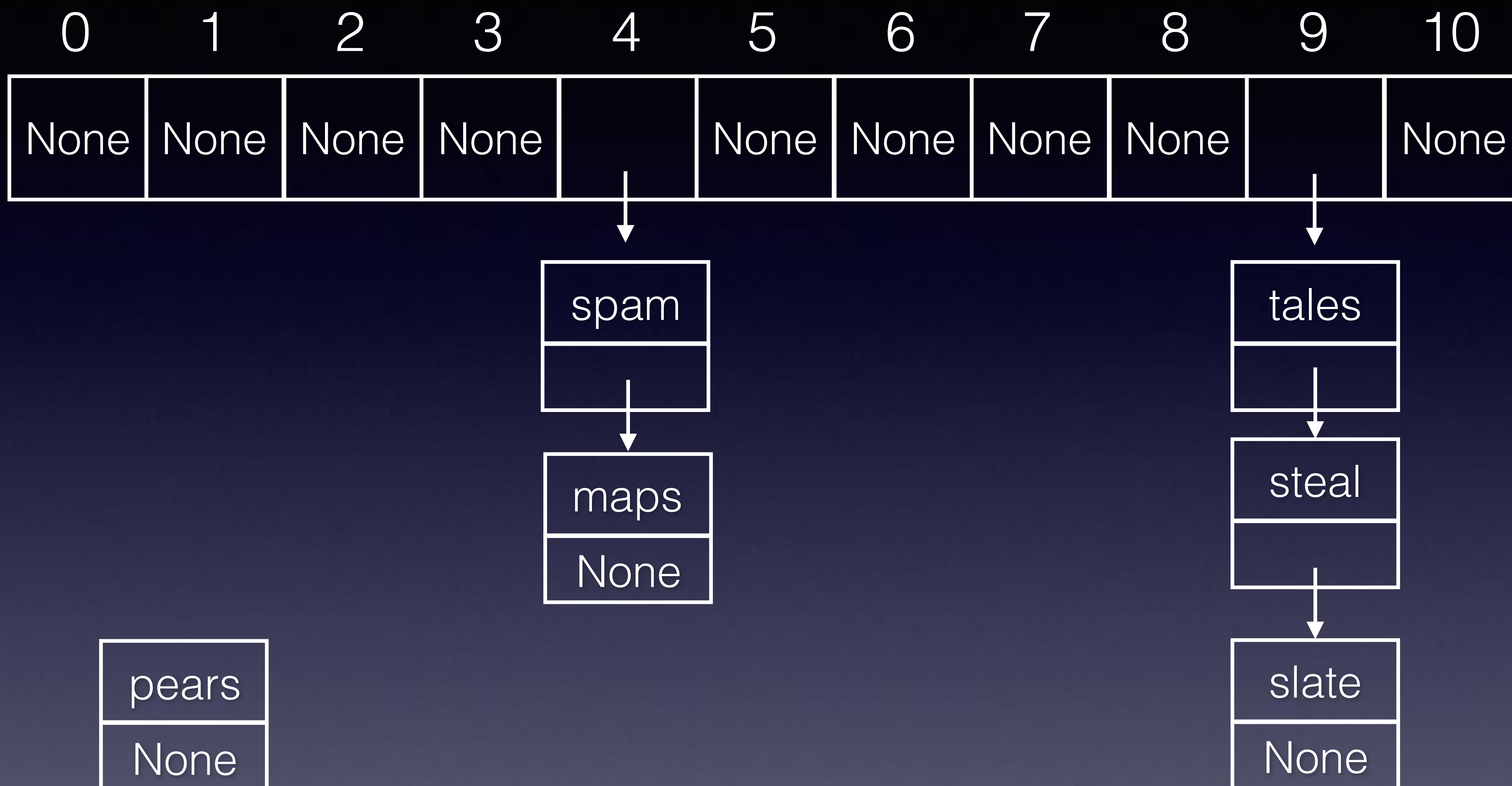
Hashing Scrabble Dictionary

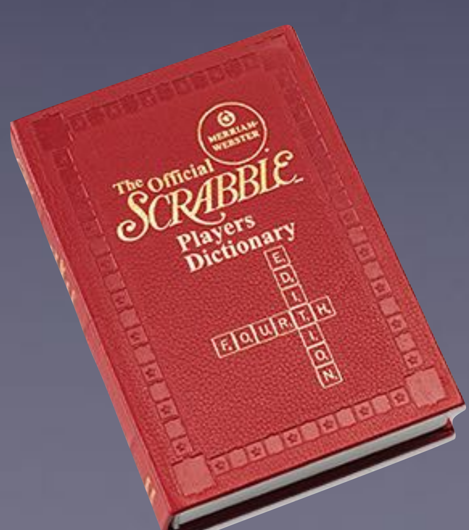





```
slot=sum(ord(c) for c in 'maps') % 11
slot=433 % 11
slot=4
```

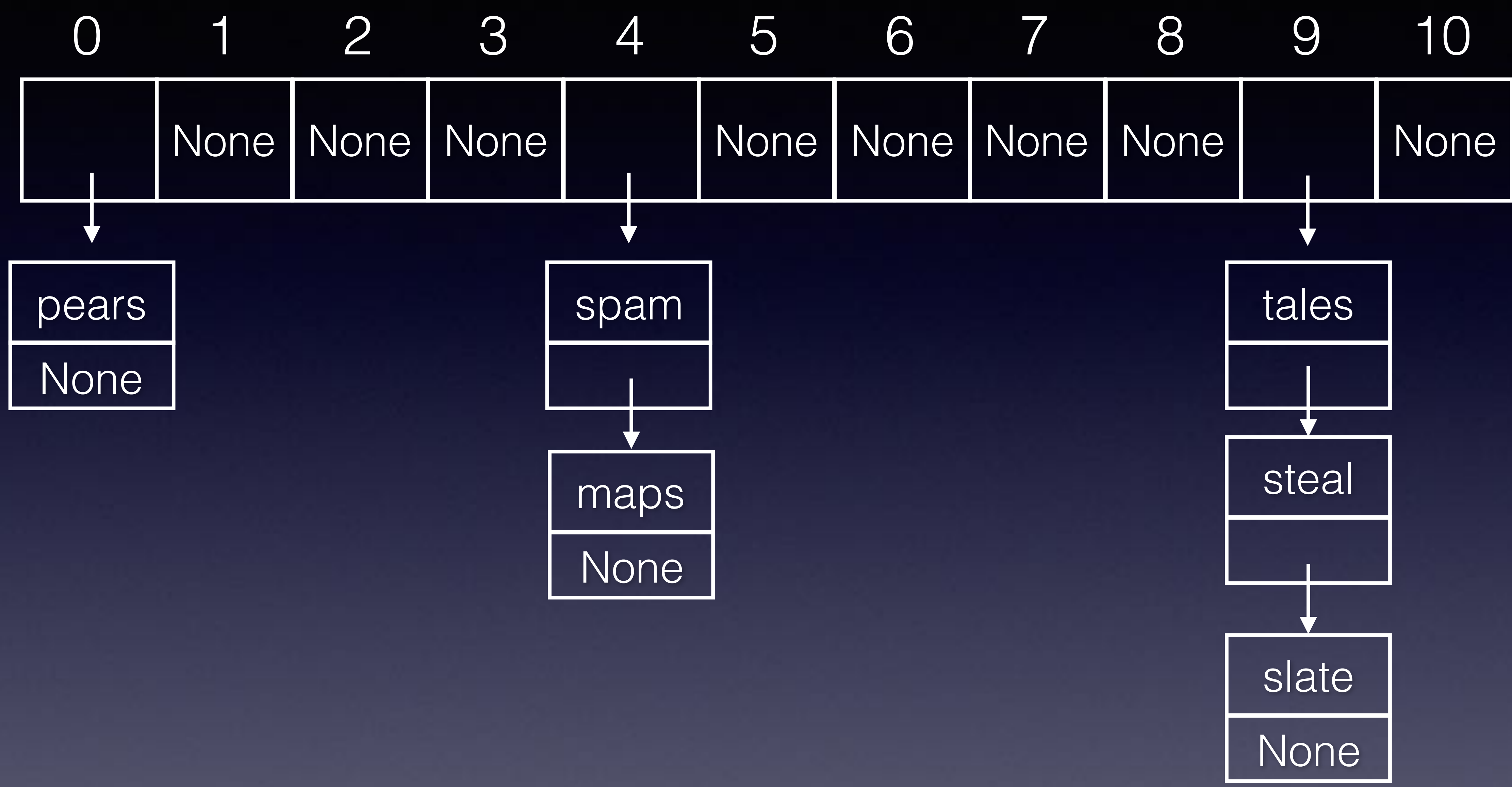
Hashing Scrabble Dictionary





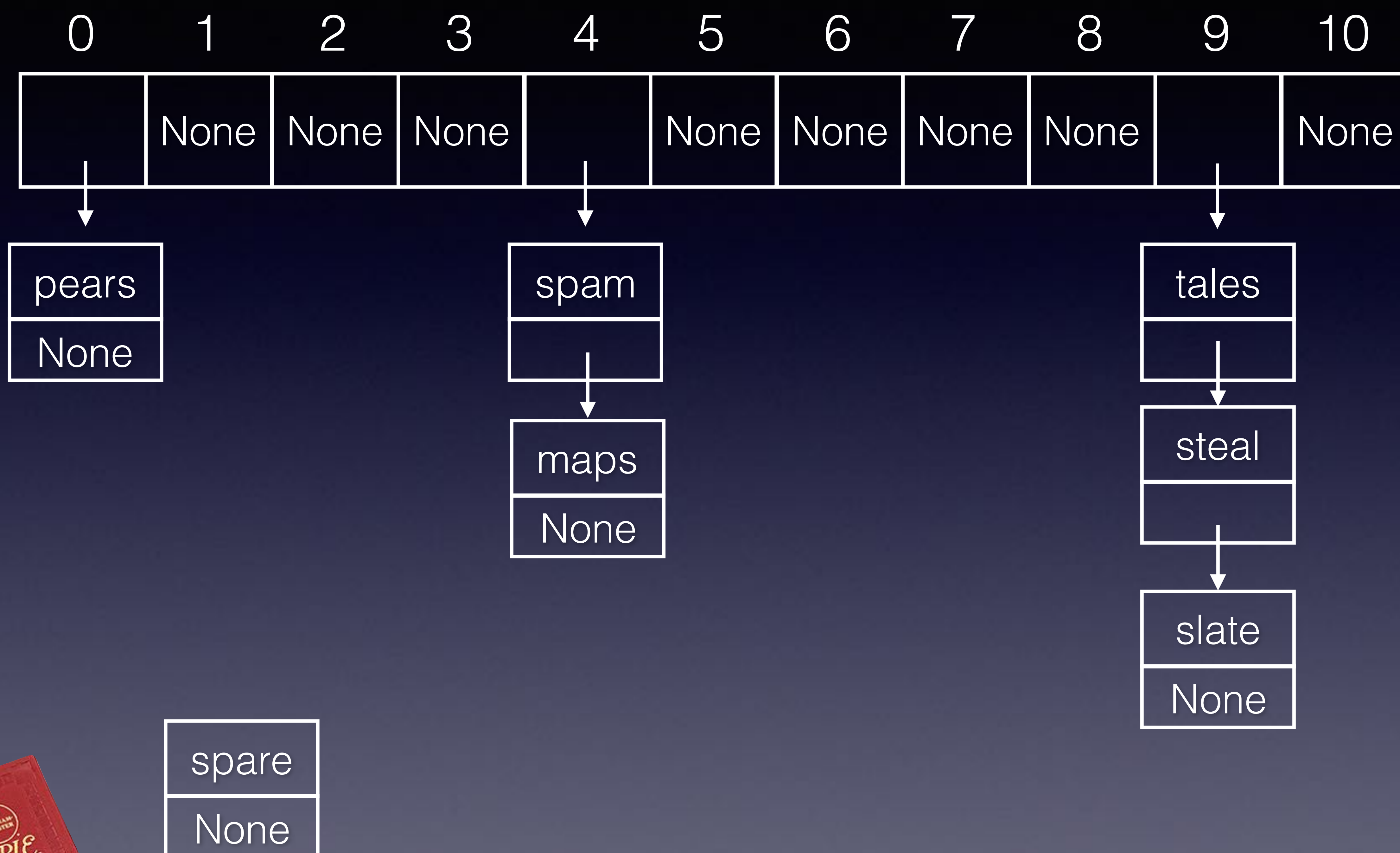

```
slot=sum(ord(c) for c in 'pears') % 11
slot=539 % 11
slot=0
```

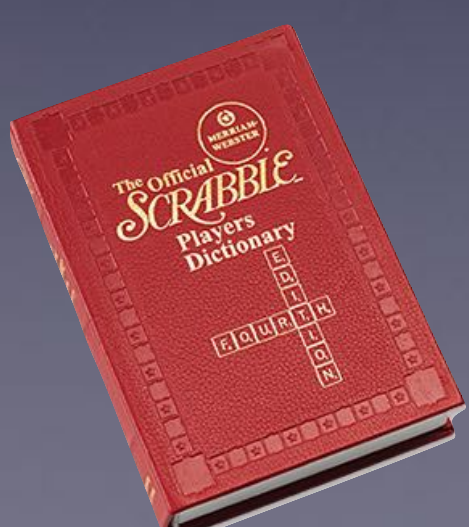
Hashing Scrabble Dictionary



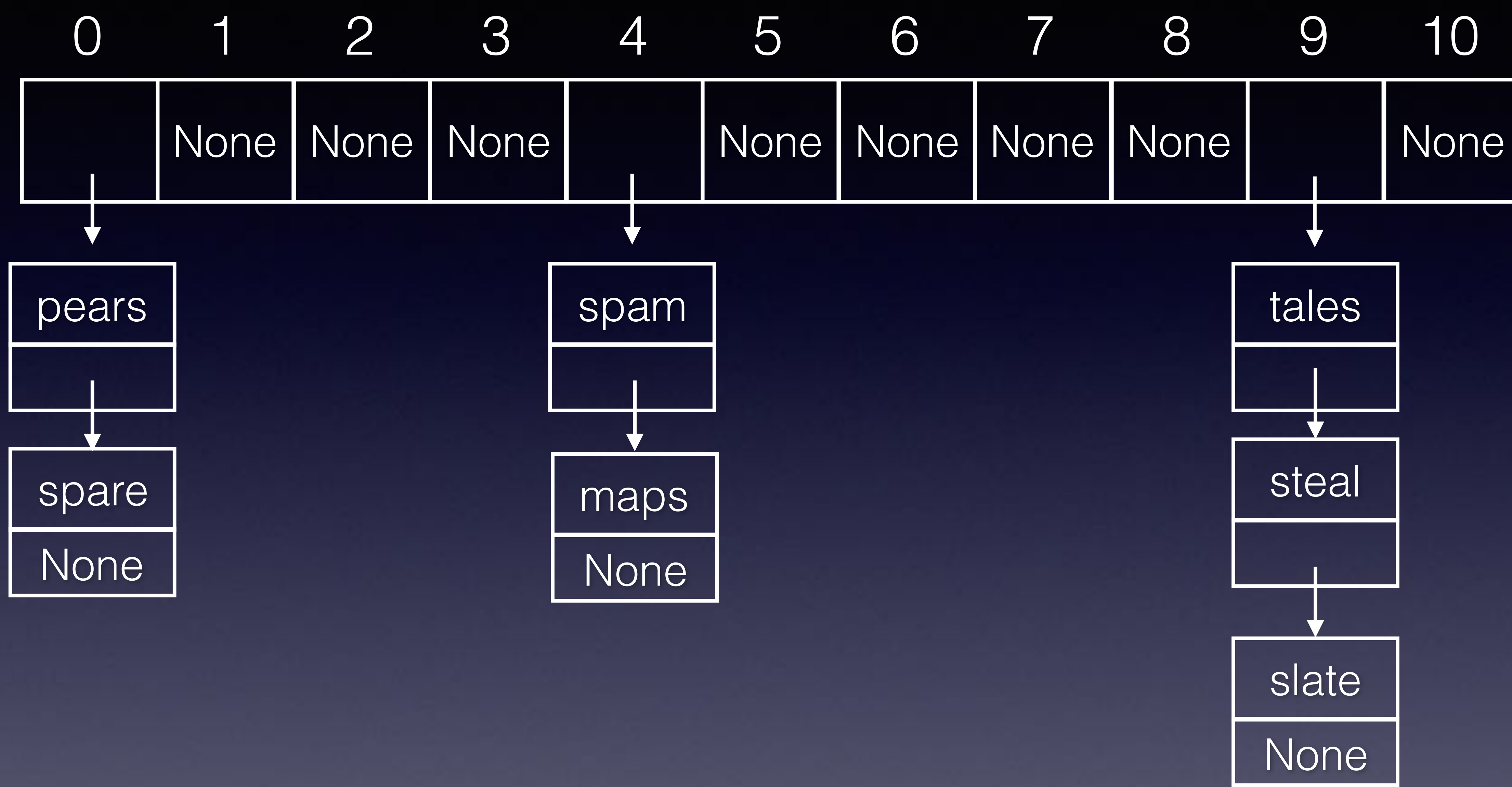

```
slot=sum(ord(c) for c in 'pears') % 11
slot=539 % 11
slot=0
```

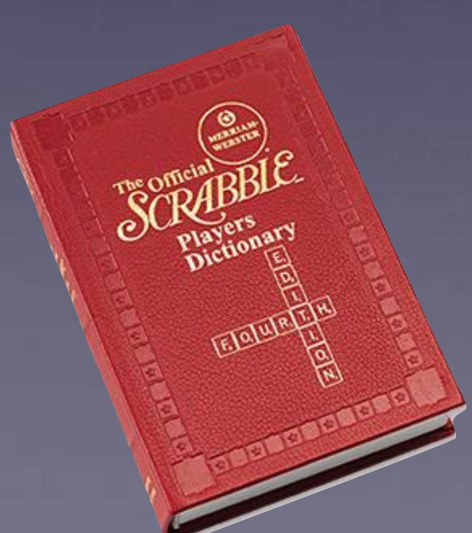
Hashing Scrabble Dictionary





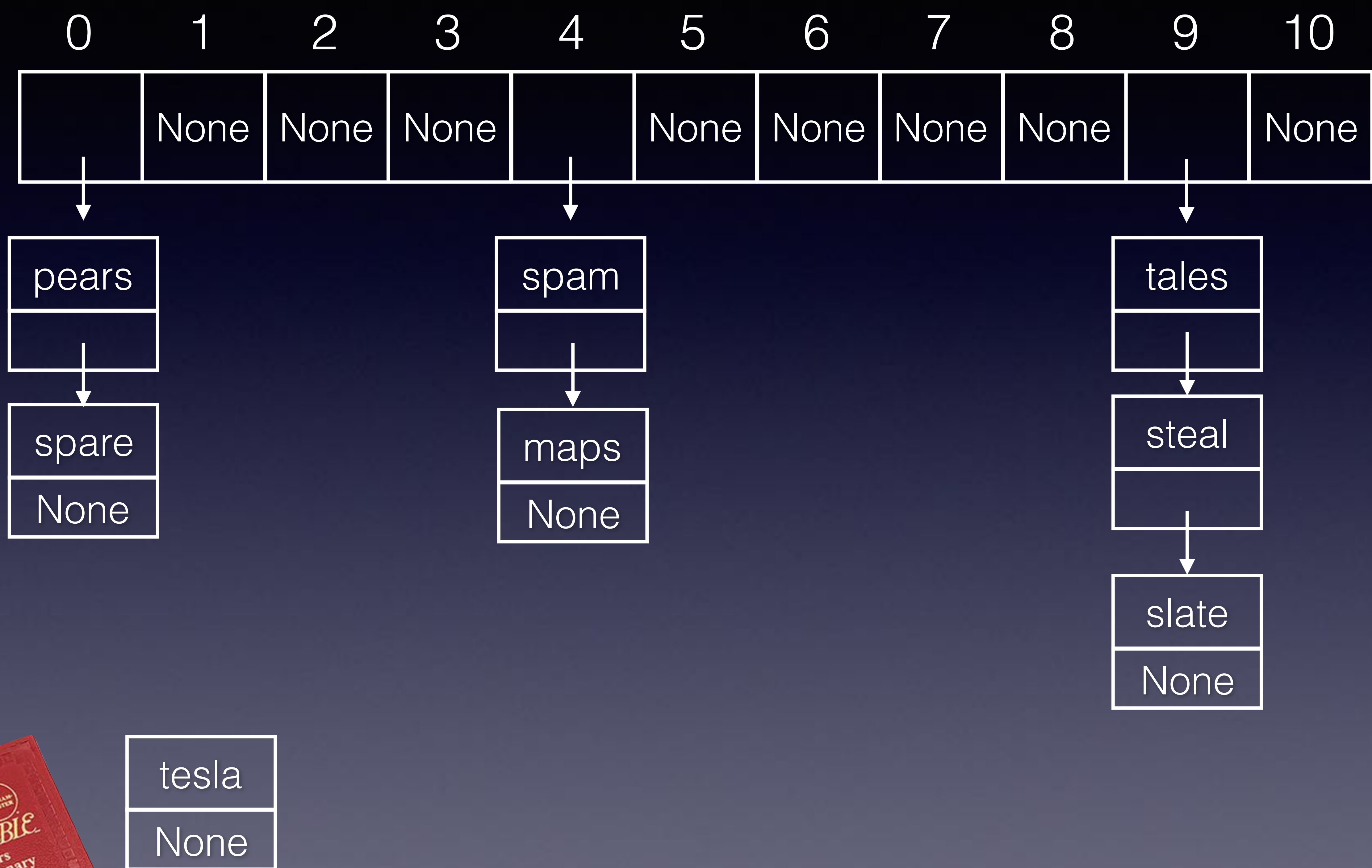
Hashing Scrabble Dictionary

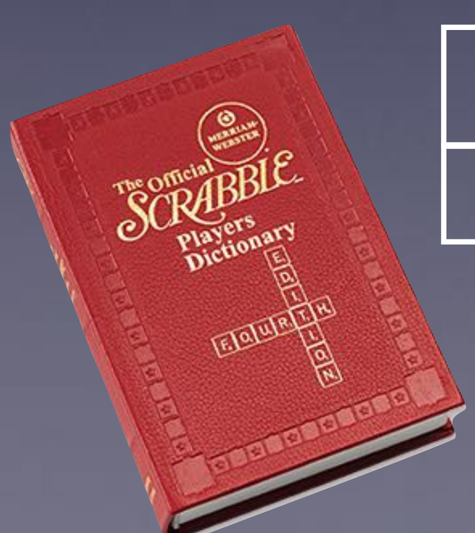




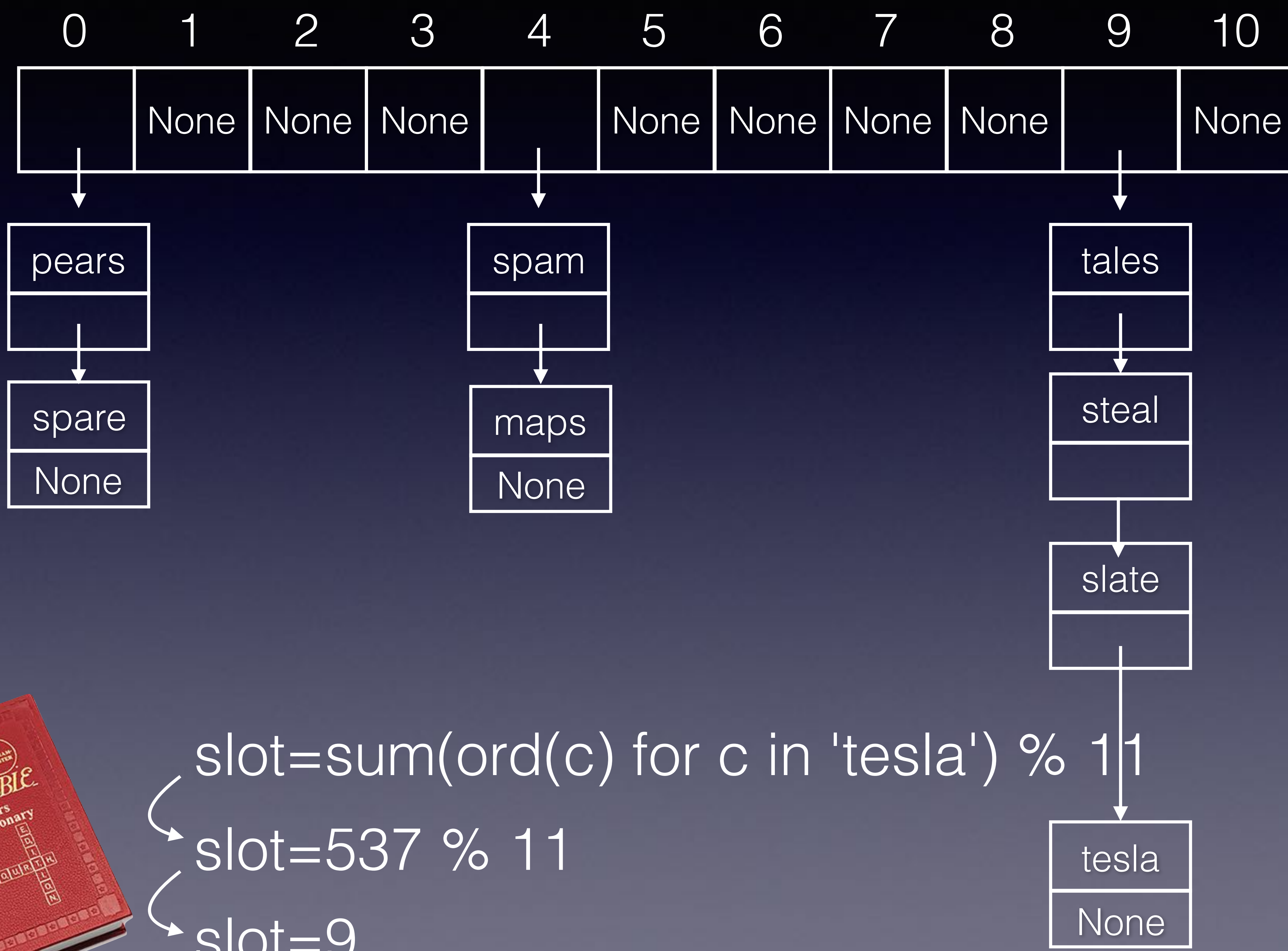

```
slot=sum(ord(c) for c in 'spare') % 11
slot=539 % 11
slot=0
```

Hashing Scrabble Dictionary





Hashing Scrabble Dictionary



slot=sum(ord(c) for c in 'tesla') % 11

slot=537 % 11

slot=9

Hashing Scrabble Dictionary

Chains can tell interesting stories

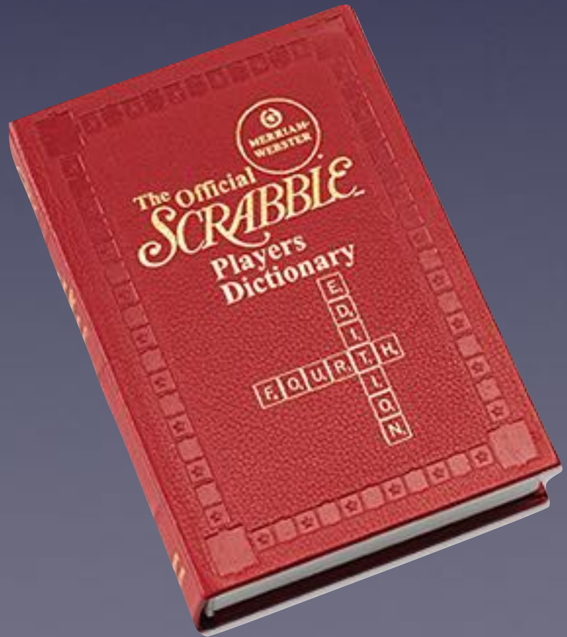
tales of **steal**ing a **slate tesla**

Hashing Scrabble Dictionary

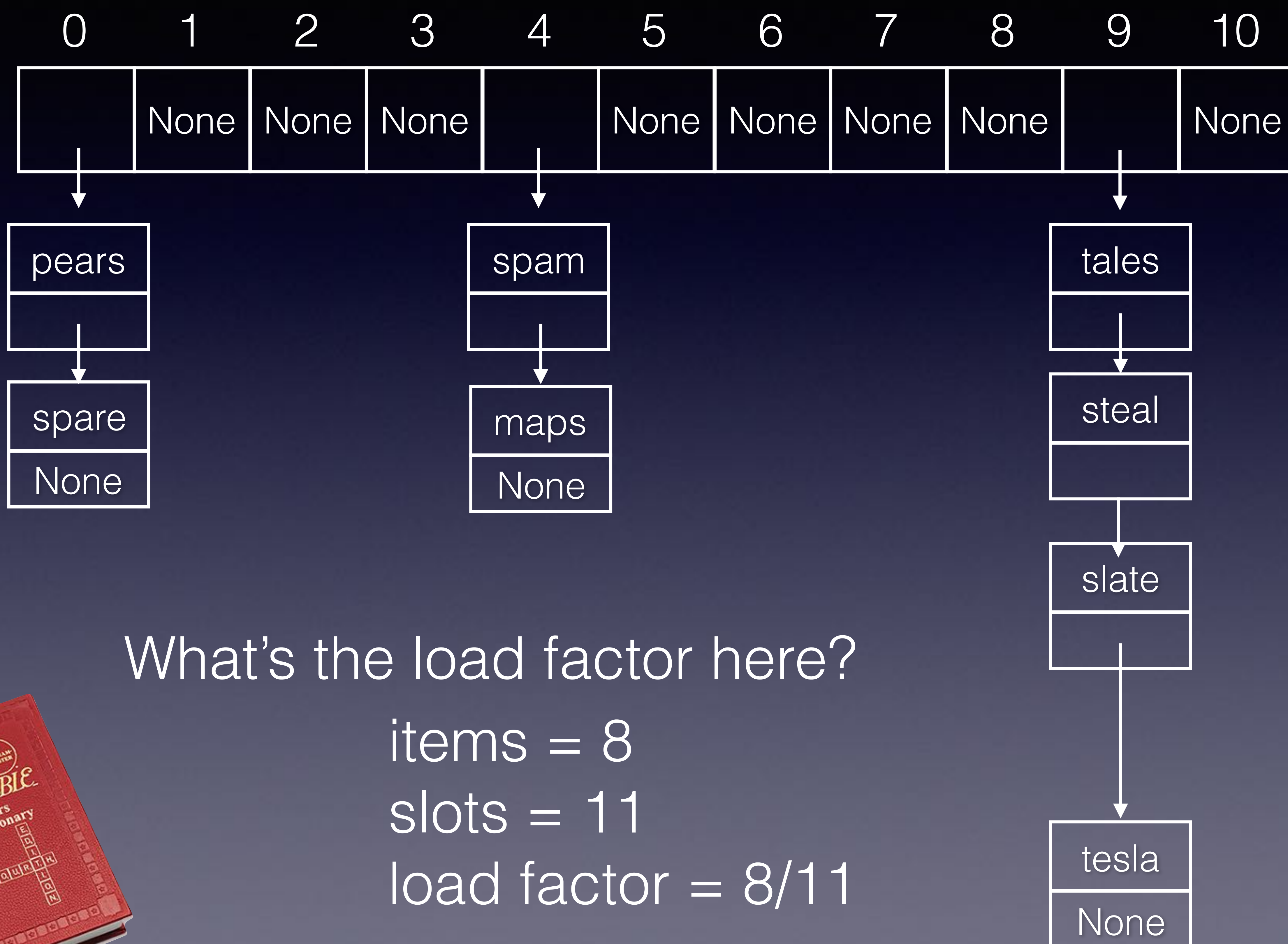
If you hash all the words in the Scrabble dictionary, you will find these entries in one slot:

apers, apres, asper, pares, parse, pears, prase, presa, rapes, reaps, spaer, spare, spear

13 words

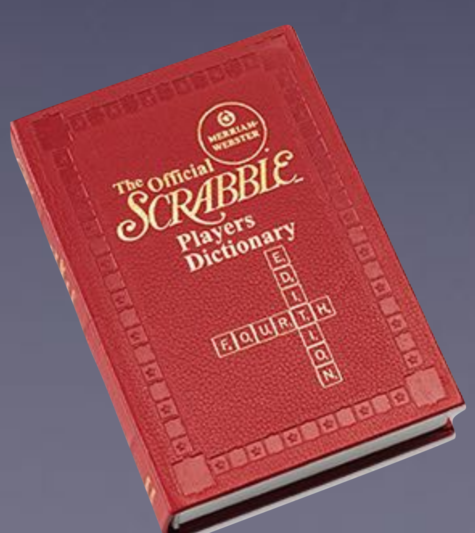


Hashing Scrabble Dictionary

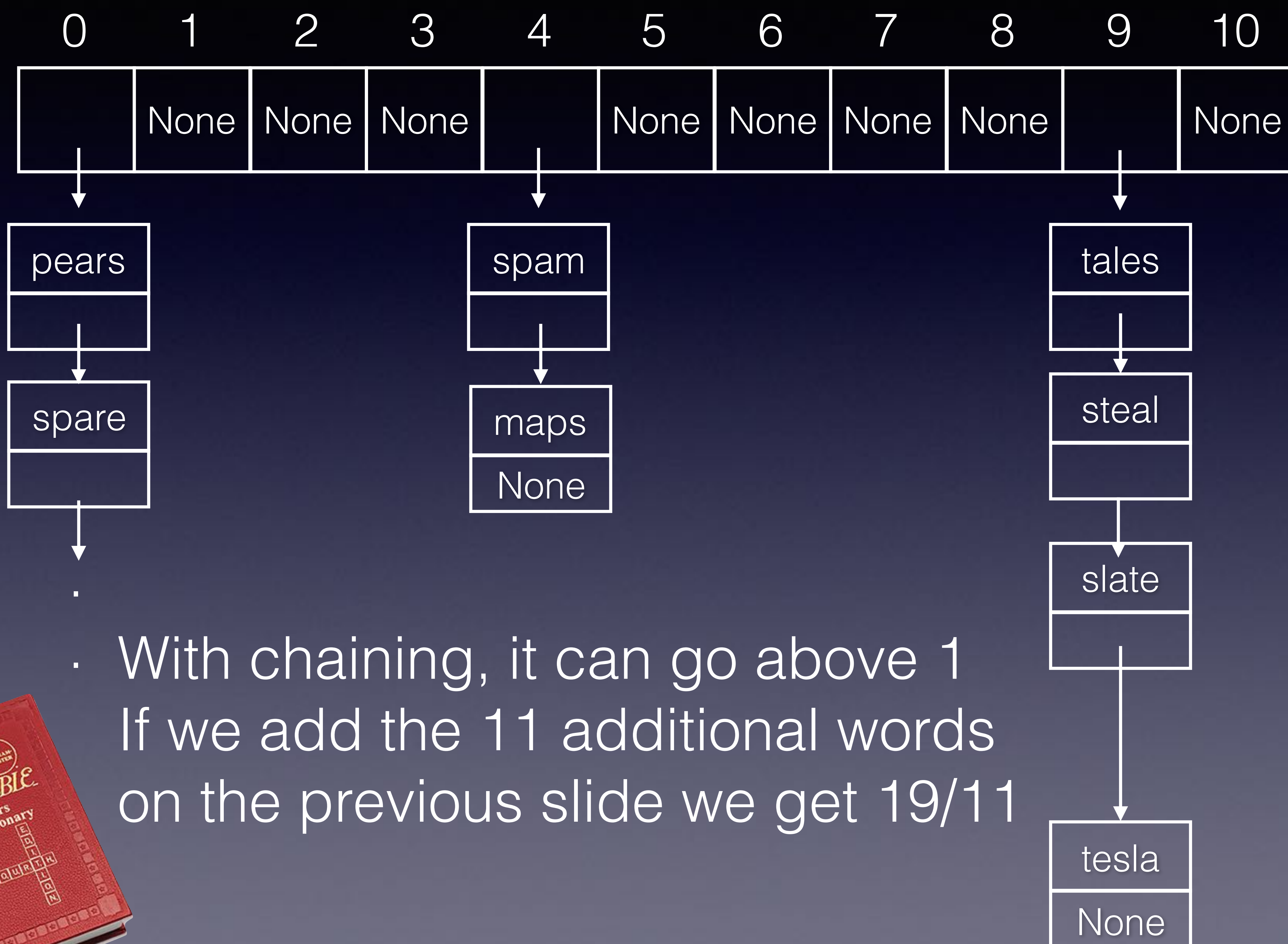


What's the load factor here?

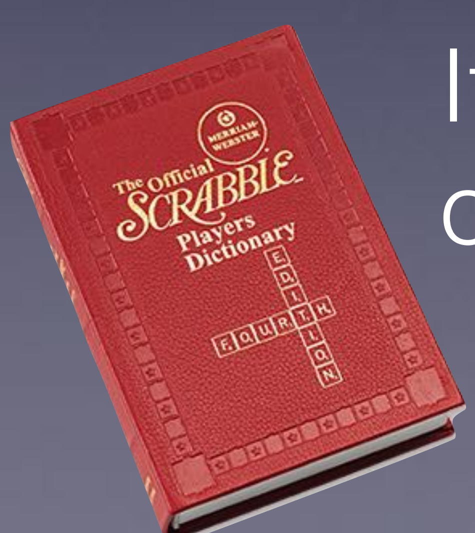
items = 8
slots = 11
load factor = 8/11



Hashing Scrabble Dictionary



With chaining, it can go above 1
If we add the 11 additional words
on the previous slide we get 19/11



Open Addressing

Open addressing tries to find the next open slot or address in the hash table.

This method keeps the hash table simple.

A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty.

This is known as **linear probing**.

When searching for a value using linear probing, you must keep looking until an empty slot is identified.

Linear Probing

If the variable i represents the i^{th} location searched beginning at 0, then **linear probing** precisely defines the order of the slots visited as follows**:**

slot = (h(value) + i) % table_size

original hash value

wrap at end of table

search by moving
over by one each time

Linear Probing

13	14	None	None	2097	None	6	None	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item stored	h(item) = item % 13	slots examined with linear probing
136	6	6,7

Linear Probing

13	14	None	None	2097	None	6	**136**	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item stored	h(item) = item % 13	slots examined with linear probing
136	6	6,7

Linear Probing

13	14	**11**	None	2097	None	6	136	None	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item stored	h(item) = item % 13	slots examined with linear probing
136	6	6,7
11	11	11,12,0,1,2

Linear Probing

13	14	11	None	2097	None	6	136	**1307**	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item stored	h(item) = item % 13	slots examined with linear probing
136	6	6,7
11	11	11,12,0,1,2
1307	7	7,8

Linear Probing

0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	11	None	2097	None	6	136	1307	1309	None	24	4809

Let's look at search with linear probing

item searched	h(item) = item % 13	slots examined with linear probing	Found
136	6	6,7	TRUE

Linear Probing

13	14	11	None	2097	None	6	136	1307	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item searched	h(item) = item % 13	slots examined with linear probing	Found
136	6	6,7	TRUE
141	11	11,12,0,1,2,3	FALSE

Linear Probing

13	14	11	None	2097	None	6	136	1307	1309	None	24	4809
0	1	2	3	4	5	6	7	8	9	10	11	12

item searched	h(item) = item % 13	slots examined with linear probing	Found
136	6	6,7	TRUE
141	11	11,12,0,1,2,3	FALSE
2096	3	3	FALSE

Rehashing

- The name for this process resolving collisions by systematically finding a new slot is known as **rehashing**.
- We showed you a linear probing algorithm, that increased the hash value by a skip value of 1 each time until an empty spot is found.
- Larger skip values can be used as well to spread out the colliding items more.

Linear Probing with Skip

If the variable i represents the ith location searched beginning at 0, then **linear probing with a skip value** precisely defines the order of the slots visited as**:**

slot = (h(value) + i*skip) % table_size

original hash value

wrap at end of table

search by moving
over by skip each time

Rehashing

0	1	2	3	4	5	6	7	8	9
0	None	10	None	20	None	30	None	40	None

- In this example, we have hashed 0, 10, 20, 30, 40 into this table.
- Because out table is length 10 our hash values are 0, 0, 0, 0, 0
- We used linear probing and a skip value of 2 for collision resolution, so we ultimately inserted the values into slots 0, 2, 4, 6, 8.
- Half of the table is unoccupied, however when we hash 50 we will fail to find a free location.

Rehashing

0	None	10	None	20	None	30	None	40	None
0	1	2	3	4	5	6	7	8	9

- Half of the table is unoccupied, however when we hash 50 we will fail to find a free location, because we don't have access to all of the slots.

- The problem is even worse using a skip value of 5! Only one extra slot is available for collision resolution.

- Choosing a **prime number** for the table size. A number where no other numbers evenly divide into avoids this problem entirely.

Rehashing

0	None	None	None	None	10	None	None	None	None
0	1	2	3	4	5	6	7	8	9

- The problem is even worse using a skip value of 5!
- We can only insert 0 and 10, because only one extra slot is available for collision resolution.
- There is no place to put 20 in this table!

Rehashing

0	1	2	3	4	5	6	7	8	9
0	None	None	None	None	10	None	None	None	None

- The problem is the skip value evenly divides the table.
- Whenever this happens you don't have access to all the slots during the collision resolution.
- The simple solution is to choose a **prime number** for the table size. A number where no other numbers evenly divide into avoids this problem entirely.

Rehashing

0	None	10	None	20	None	30	None	40	None
0	1	2	3	4	5	6	7	8	9

- Still, a problem with linear probing is the clustering of values in the table.
- Larger skip values are one way to address this.
- Another way to disperse the values is **quadratic probing.**

Quadratic Probing

If the variable i represents the ith location searched beginning at 0, then **quadratic probing with a skip value** precisely defines the order of the slots visited:

slot = (h(value) + i*i) % table_size

original hash value

wrap at end of table

search by moving over by i^2 each time

Quadratic Probing

If the variable i represents the ith location searched beginning at 0, then **quadratic probing with a skip value** precisely defines the order of the slots visited**:**

slot = (h(value) + i*i) % table_size

original hash value

wrap at end of table

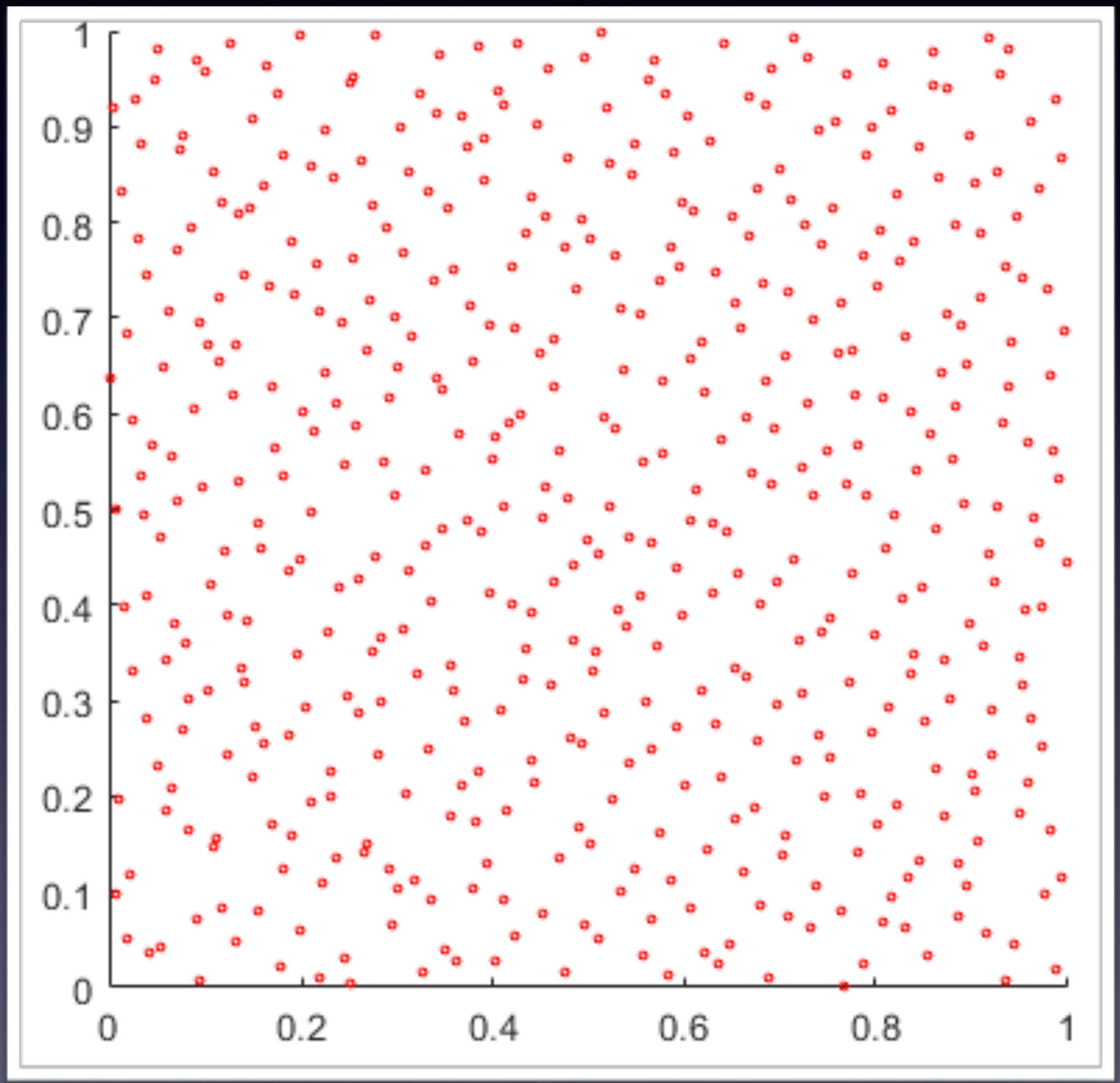
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81, …)

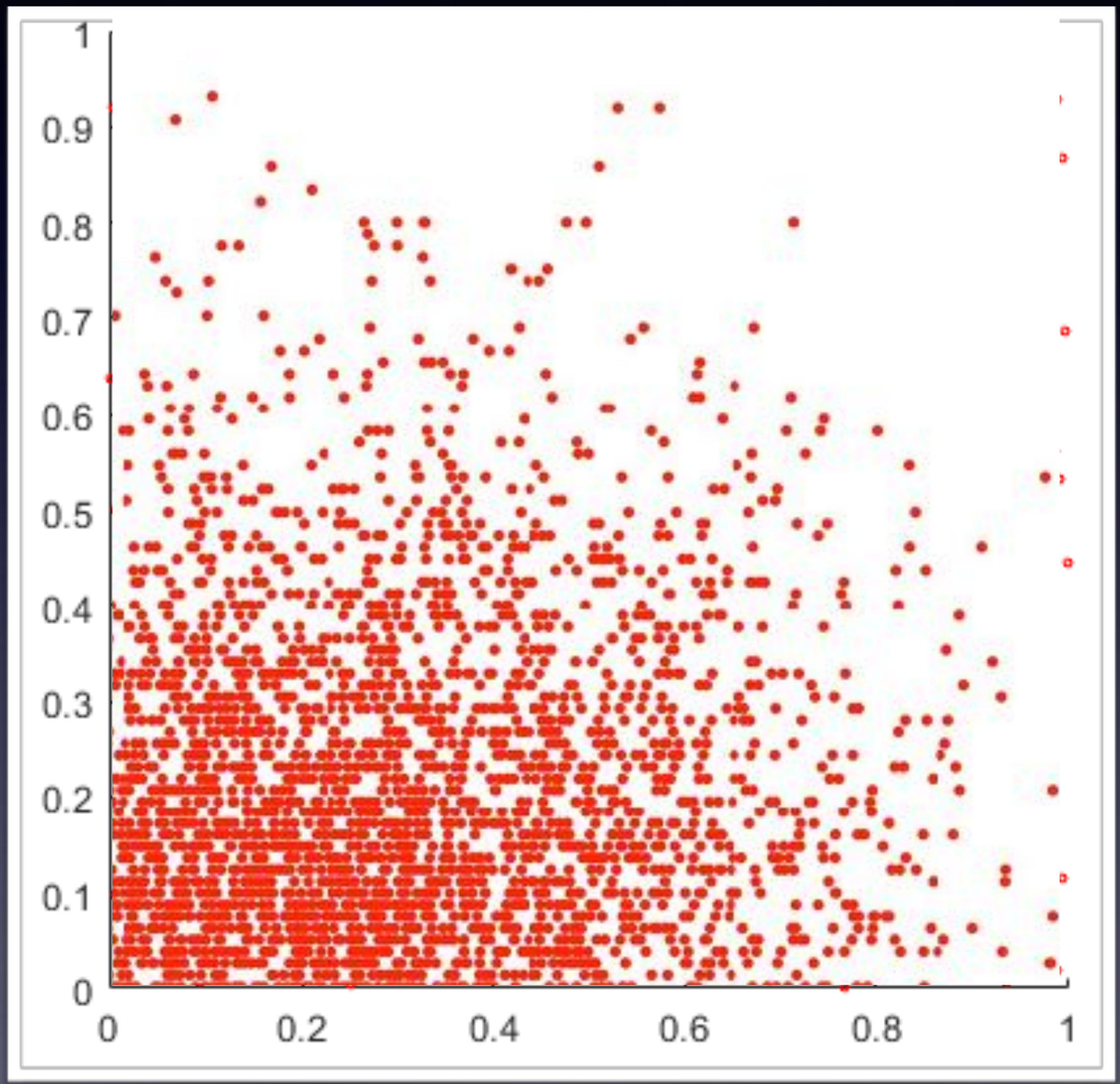
Good Hash Functions

- A good hash function should have the following characteristics:
    - Be easy and quick to compute
    - Minimize collision
    - Distribute key values uniformly in the hash table
    - Use all the information provided in the key to distinguish between keys. (e.g. our string hash function did not, so there were collisions between anagrams)

Good Hash Functions

Why uniform? less density, fewer collisions





2D visualizations of uniform (left) and non-uniform (right) numeric values. More dot collisions in the non-uniform distribution.

Analysis of Hashing

- Perfect hashing provides the desired O(1) time complexity for search.
- For other cases we can use the load factor to obtain approximate functions for complexity.

$$\lambda = \frac{number\ of\ items}{number\ of\ slots}$$

Analysis of Hashing

$$\lambda = \frac{number\ of\ items}{number\ of\ slots}$$

- If the load factor is low then there are a lot of empty slots and there will be fewer collisions. So, the typical case will be the no collision case.
- As the load factor increases, so does the number collisions which we need to account for.

Analysis of Hashing

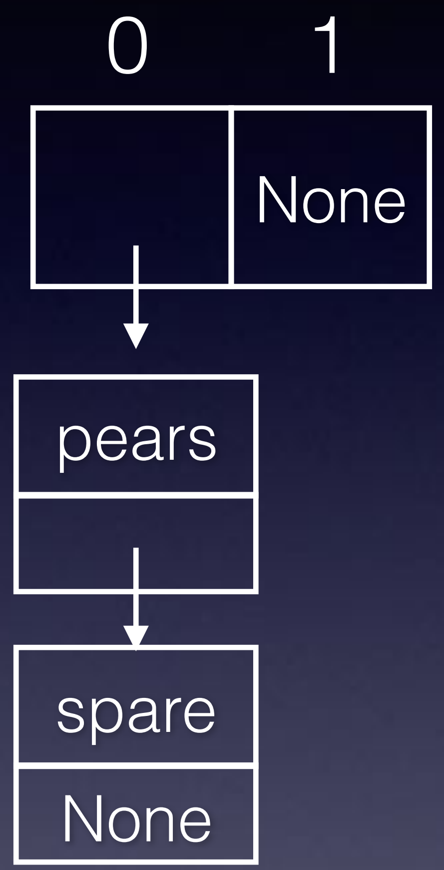
- The book gives approximate formulae. We will look at the simpler case when chaining is employed. It will be enough to make the point that hashing can be O(1).

- Using **chaining**

$$\lambda = \frac{number\ of\ items}{number\ of\ slots}$$

gives the average length of a chain

Analysis of Hashing



In this small example, the load factor is 2 / 2

My average chain length is (2 + 0) / 2

Analysis of Hashing

- Using **chaining** the average comparisons of unsuccessful queries is

$$\lambda$$

- Why? Because we will need to always look at the entire chain. The load factor gives us the average chain length.

Analysis of Hashing

- Using **chaining** the average comparisons for successful queries is

$$1 + \frac{\lambda}{2}$$

When the load factor is low this term dominates. It takes one comparison when there are no collisions.

When the load factor is high this term dominates. On average, half the chain will be sequentially searched.

Why Hashing is O(1)

- With both chaining and open addressing, complexity is a function of the load factor, and the load factor is a function of the number of elements and the table size.

- We may look at this doubt that hashing is O(1). It should be O(λ) right?

- The trick is to keep the load factor roughly constant by increasing the number of slots (hash table size) as the number of elements increases.

- Remember that constants are ignored in asymptotic complexity analysis, so O(λ) is O(1).