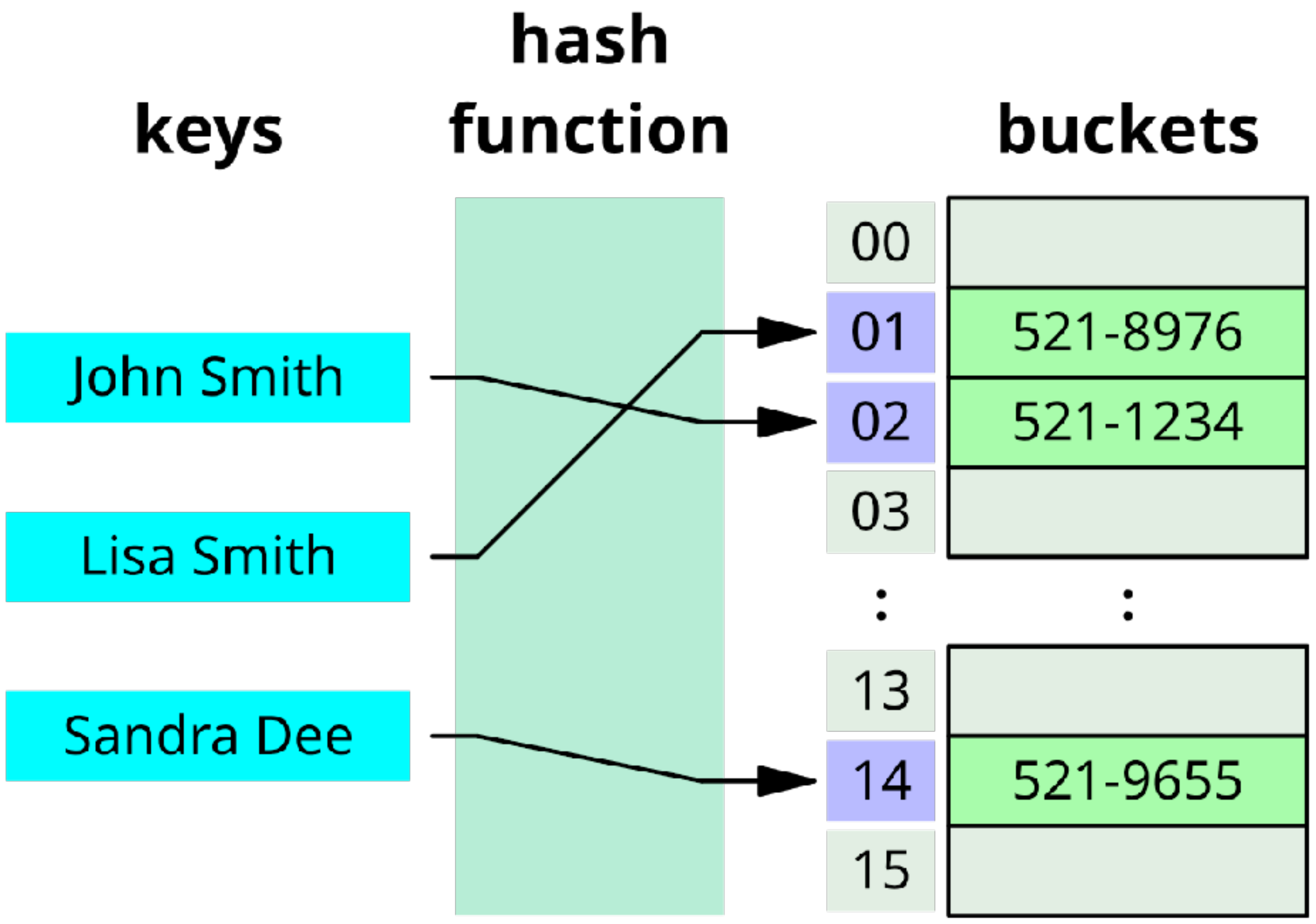
CS62 Class 20: Hash Tables (pt I)

Searching



Agenda

- Motivation
- Deriving hashtables: naive approach
- Separate chaining & handling collisions
- Hash table resizing

Hashtable motivation

We’ve now seen several implementations of maps…

	Worst case			Average case			
	Search	**Insert**	**Delete**	**Search**	**Insert**	**Delete**	**Notes**
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$	Random trees are log n
2-3 Tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	Beautiful idea, very hard to implement
LLRB	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	Bijection with 2-3 tree, hard to implement

Limits of Search Tree Based Maps (and Sets)

Our search tree based sets require items to be comparable.

- Need to be able to ask “is X < Y?” Not true of all types (ex. How do you compare 苹 and 橙?).
- Could we somehow avoid the need for objects to be comparable?

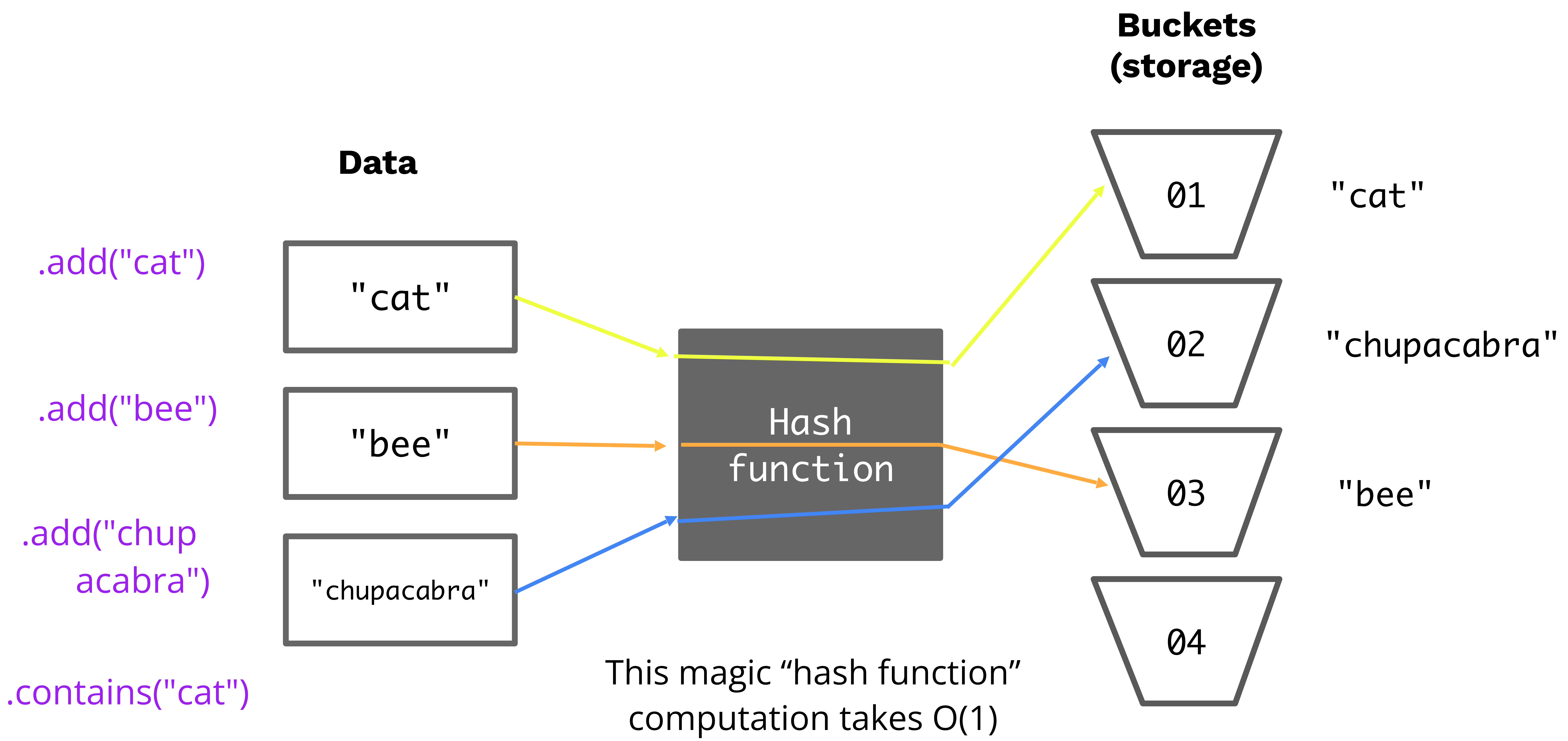
Our search tree sets have excellent performance, but could maybe be better?

- Θ(log N) is amazing. 1 billion items is still only height ~30.
- Could we somehow do better than Θ(log N)?

Today we’ll see the answer to both of the questions above is yes.

Hash functions associate data with a storage bucket

- Hash tables take data, transform them using a *hash function* into integer indices for storage buckets. We'll be focusing on `.add()` and `.contains()` today.



Naive approach:
Data indexed arrays

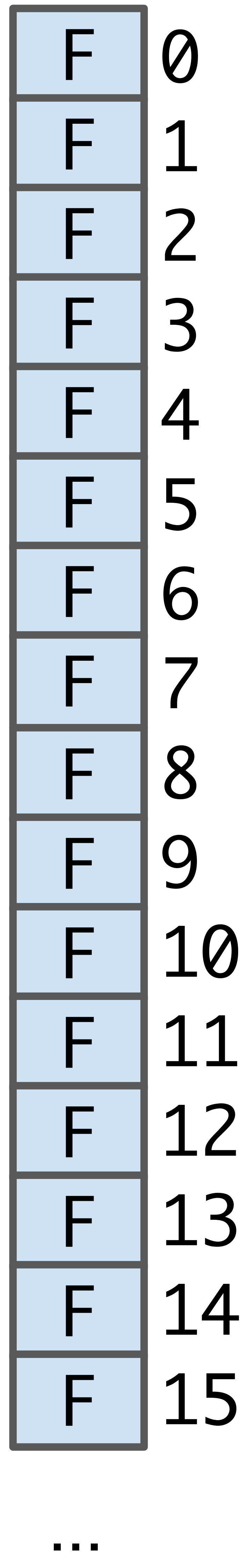
How would we implement extremely fast add/lookups?

One extreme approach: Create an array of booleans indexed by data.
Data = integers, which is also the index of the array.

- Initially all values are false
- When an item is added, set appropriate index to true

```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
```

Set containing nothing



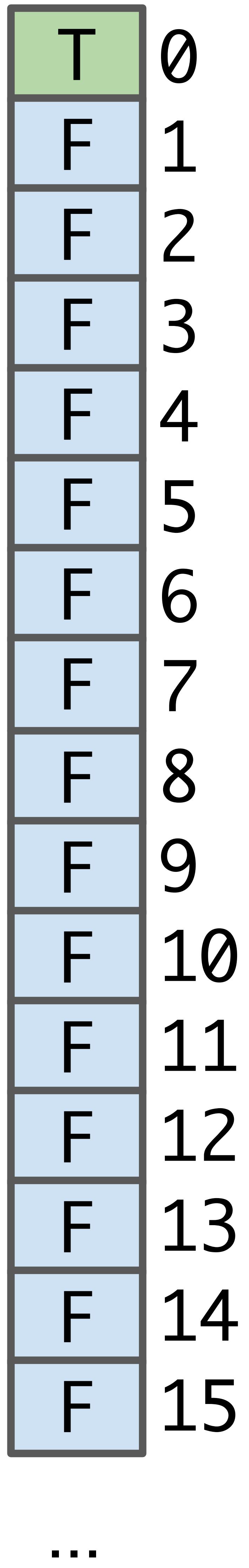
Using Data as an Index

One extreme approach: Create an array of booleans indexed by data.
Data = integers, which is also the index of the array.

- Initially all values are false
- When an item is added, set appropriate index to true

```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
diis.add(0);
```

Set containing 0



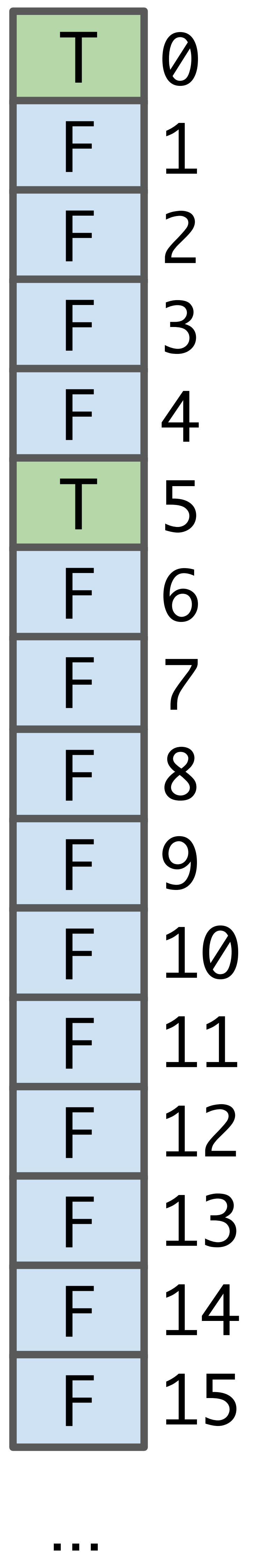
Using Data as an Index

One extreme approach: Create an array of booleans indexed by data.
Data = integers, which is also the index of the array.

- Initially all values are false
- When an item is added, set appropriate index to true

```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
diis.add(0);
diis.add(5);
```

Set containing 0, 5



Using Data as an Index

One extreme approach: Create an array of booleans indexed by data.
Data = integers, which is also the index of the array.

- Initially all values are false
- When an item is added, set appropriate index to true

```
DataIndexedIntegerSet diis;
diis = new DataIndexedIntegerSet();
diis.add(0);
diis.add(5);
diis.add(11);
```

Set containing 0, 5, 11

T	0
F	1
F	2
F	3
F	4
T	5
F	6
F	7
F	8
F	9
F	10
T	11
F	12
F	13
F	14
F	15

...

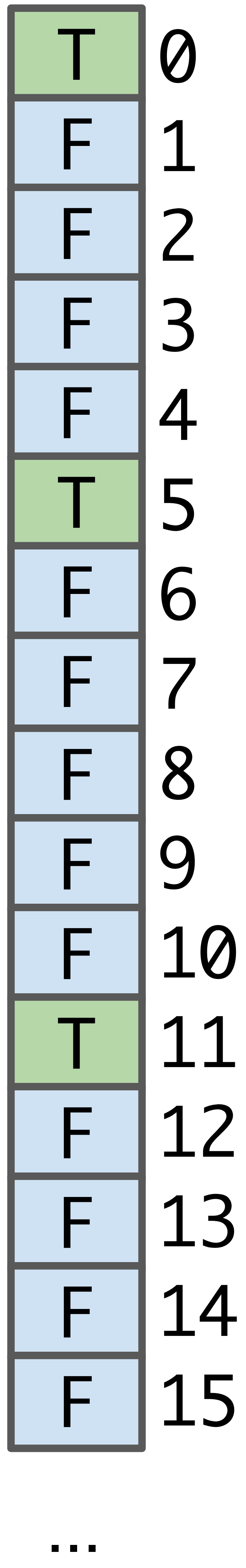
DataIndexedIntegerSet implementation

```
public class DataIndexedIntegerSet {
   private boolean[] present;

   public DataIndexedIntegerSet() {
       present = new boolean[2000000000];
   }

   public void add(int i) {
       present[i] = true;
   }

   public boolean contains(int i) {
           return present[i];
   }
}
```

Q: What are some downsides to this approach?

add() is a *constant* time operation: just set a flag to true

contains() is a *constant* time operation: just look up the value in the array



Set containing 0, 5, 11

DataIndexedIntegerSet downsides

```
public class DataIndexedIntegerSet {
   private boolean[] present;

   public DataIndexedIntegerSet() {
       present = new boolean[2000000000];
   }

   public void add(int i) {
       present[i] = true;
   }

   public boolean contains(int i) {
           return present[i];
   }
}
```

- Extremely wasteful of memory
- Need some way to generalize beyond integers

T	0
F	1
F	2
F	3
F	4
T	5
F	6
F	7
F	8
F	9
F	10
T	11
F	12
F	13
F	14
F	15

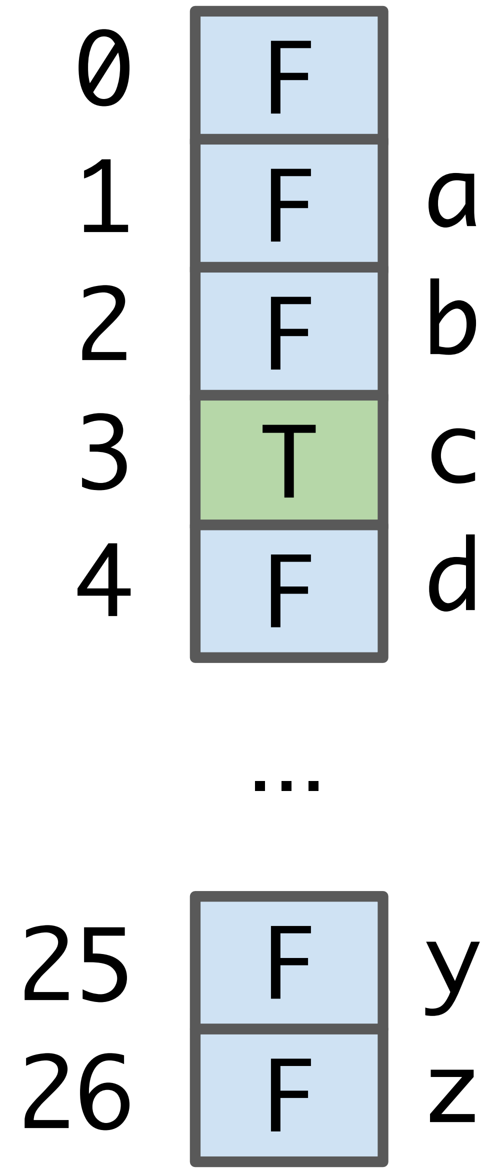
...

Set containing 0, 5, 11

Generalizing to (English) Strings

aka Hashing

Generalizing the DataIndexedIntegerSet Idea



Suppose we want to *add*(“cat”)

The key question:

- What is the catth element of a list?
- One idea: Use the first letter of the word as an index.
  - **a** = 1, **b** = 2, **c** = 3, …, **z** = 26

What’s wrong with this approach?

- Other words start with **c**.
  - contains(“chupacabra”) : true
- Can’t store “ :3”

we say that “chupacabra” **collides with** “cat”

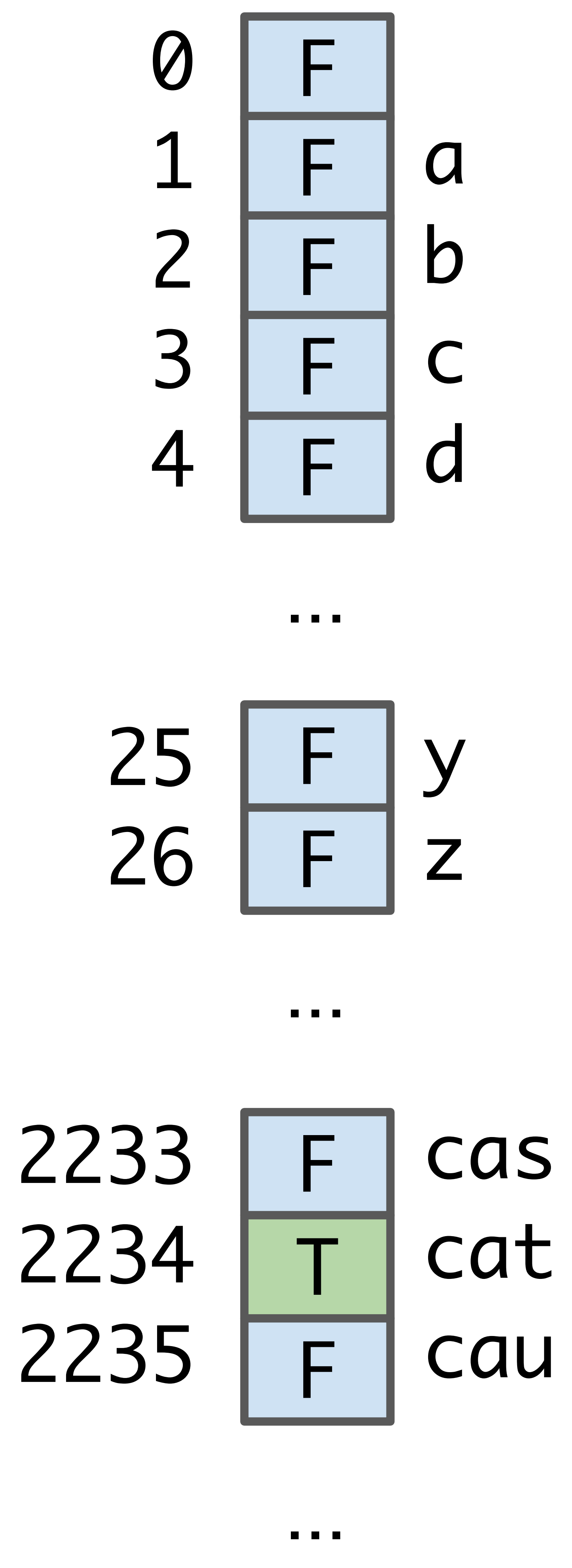
Avoiding Collisions

Here's an idea: To get the new index, we will use all the letters by multiplying each by a power of 27.

- **a** = 1, **b** = 2, **c** = 3, …, **z** = 26
- Thus the index of "cat" is $(3 \times 27^{2}) + (1 \times 27^{1}) + (20 \times 27^{0}) =$ 2234.

Why this specific pattern?

- Let's review how numbers are represented in decimal.



The Decimal Number System vs. Our System for Strings

In the decimal number system, we have 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Want numbers larger than 9? Use a sequence of digits.

Example: 7091 in base 10

- $7091_{10} = (7 \times 10^3) + (0 \times 10^2) + (9 \times 10^1) + (1 \times 10^0)$

Our system for strings is almost the same, but with letters (base 27, we don't use 0).

- $cat_{27} = (3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234_{10}$

This is the beginnings of a *hash function*.

Worksheet time!

Convert the word **“bee”** into a number by using our “powers of 27” strategy. That is, *hash* “bee”.

Reminder: $\text{cat}_{27} = (3 \times 27^{2}) + (1 \times 27^{1}) + (20 \times 27^{0}) = 2234_{10}$

Hint: ‘b’ is letter 2, and ‘e’ is letter 5. And $27^{2} = 729$ (but feel free to use a calculator)

Worksheet answers

Convert the word **"bee"** into a number by using our "powers of 27" strategy.

Reminder: $cat_{27} = (3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234_{10}$

Hint: 'b' is letter 2, and 'e' is letter 5. And $27^2 = 729$

- $bee_{27} = (2 \times 27^2) + (5 \times 27^1) + (5 \times 27^0) =$

  $(1458) + (135) + (5) = 1598_{10}$

Avoiding collisions with uniqueness

- $cat_{27} = (3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234_{10}$
- $bee_{27} = (2 \times 27^2) + (5 \times 27^1) + (5 \times 27^0) = 1598_{10}$

As long as we pick a base ≥ 26, this algorithm is guaranteed to give each lowercase English word a unique number

- Using base 27, no other words will get the number 1598.

This is an example of a *hash function* for hashing lowercase English words where it's guaranteed that we will *never have a collision*.

(Practice problem: write an `englishToInt` hash function that will automatically compute the hash for any input english word.)

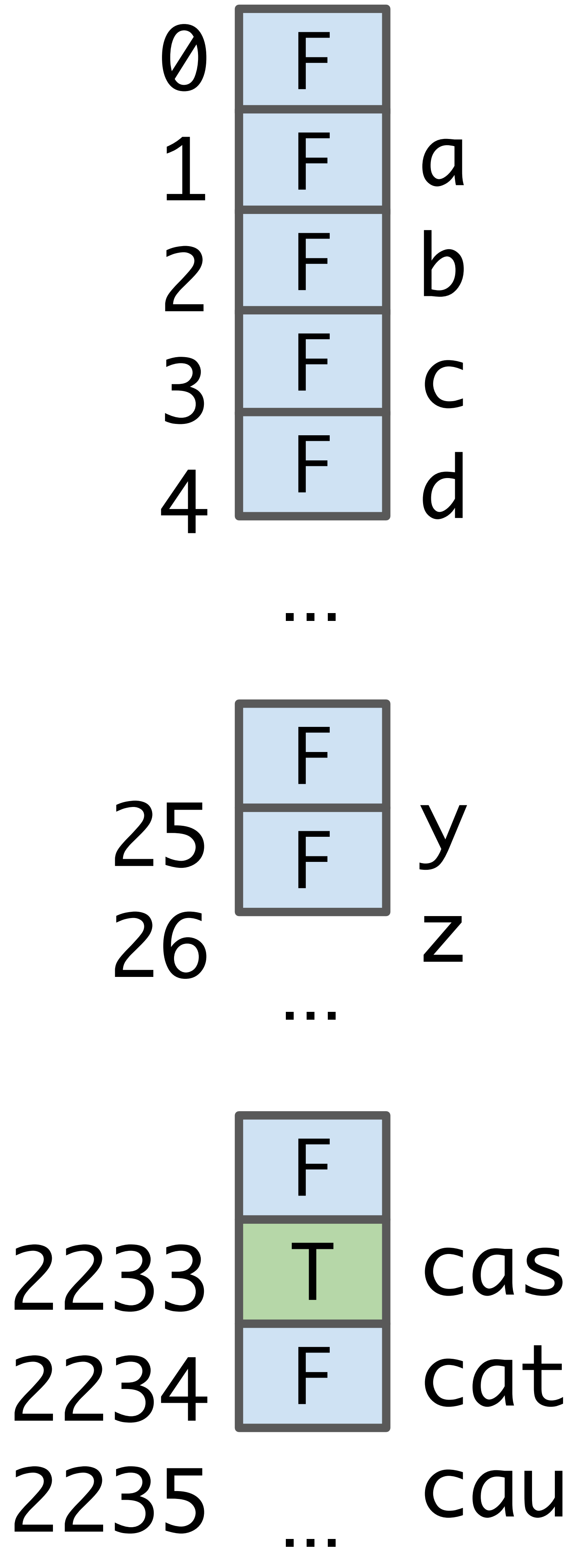
DataIndexedEnglishWordSet Implementation

```
public class DataIndexedEnglishWordSet {
   private boolean[] present;

   public DataIndexedEnglishWordSet() {
       present = new boolean[2000000000];
   }

   public void add(String s) {
       present[englishToInt(s)] = true;
   }

   public boolean contains(String s) {
           return present[englishToInt(s)];
   }
}
```

We've solved the first problem of generalizing from integers through introducing a **hash function** `englishToInt(s)`

```
public void add(int i) {
    present[i] = true;
}
```

^ old way



Set containing “cat”

Generalizing to any String

DataIndexedStringSet

Using only lowercase English characters is too restrictive.

- What if we want to store strings like “2pac” or “eGg!”?
- To understand what value we need to use for our base, let’s briefly discuss the ASCII standard.

ASCII Characters

The most basic character set used by most computers is ASCII format.

- Each possible character (128 total) is assigned a value between 0 and 127.
- Characters 33 - 126 are “printable”, and are shown below.
- For example, `char c = 'D'` is equivalent to `char c = 68`.

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	`	112	p		

biggest value is 126

DataIndexedStringSet

Using only lowercase English characters is too restrictive.

- What if we want to store strings like “2pac” or “eGg!”?
- Let's use the ASCII standard as an encoding, and take 126 as our base.

Examples:

- $\text{bee}_{126} = (98 \times 126^2) + (101 \times 126^1) + (101 \times 126^0) = 1{,}568{,}675$
- $\text{2pac}_{126} = (50 \times 126^3) + (112 \times 126^2) + (97 \times 126^1) + (99 \times 126^0) = 101{,}809{,}233$
- $\text{eGg!}_{126} = (98 \times 126^3) + (71 \times 126^2) + (98 \times 126^1) + (33 \times 126^0) = 203{,}178{,}213$

Implementing the hash function asciiToInt

Finally, if you want to use characters beyond ASCII, you can use the Unicode encoding. This supports, for instance, characters like 员, ☂, and الطبيعة.

```
public static int asciiToInt(String s) {
        int intRep = 0;
        for (int i = 0; i < s.length(); i += 1) {
         intRep = intRep * 126;
         intRep = intRep + s.charAt(i);
        }
        return intRep;
}
```

Strings are composed of chars, so they automatically take their ASCII value in math operations

Another problem: Integer Overflow

In Java, the largest possible integer is 2,147,483,647.

- If you go over this limit, you overflow, starting back over at the smallest integer, which is -2,147,483,648.
- In other words, the next number after 2,147,483,647 is -2,147,483,648.

```java
int x = 2147483647;
System.out.println(x);
System.out.println(x + 1);
```

```
$ javac BiggestPlusOne.java
$ java BiggestPlusOne
2147483647
-2147483648
```

Consequence of Overflow: Collisions for long words

Because Java has a maximum integer, we won't get the numbers we expect!

- With base 126, we will run into overflow even for short strings.
  - Example: omens_{126}= 28,196,917,171, which is much greater than the maximum integer (28 billion versus 2 billion)!
  - `asciiToInt('omens')` will give us -1,867,853,901 instead due to overflow.
- Overflow can lead to **collisions**, resulting in wrong answers.

```
public static void main(String[] args) {
  DataIndexedStringSet disi = new DataIndexedStringSet();
  disi.add("melt banana");
  disi.contains("subterrestrial anticosmetic");
  //asciiToInt for these strings is both 839099497
}
```

returns true!

Hash Codes and the Pigeonhole Principle

The official term for the number we're computing is a **"hash code"**, which is the result of a **hash function**.

- Via Wolfram Alpha: a hash code "projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members."
- Here, our target set is the set of Java integers, which is of size 4294967296 (both negative and positive integers).

Pigeonhole principle tells us that if there are more than 4294967296 possible items, multiple items will share the same hash code.

- There are more than 4294967296 strings.
  - "one", "two", … "nineteen quadrillion", …

Bottom line: Collisions are inevitable.

Hash Tables

Two Fundamental Challenges of Hash Tables

- How do we resolve hash code collisions?

- How do we compute a hash code for arbitrary objects?
  - Example: Our hash code for “cat” was 2234.
  - For Strings, this was relatively straightforward (treat as a base 126 number).
  - What about for `Class PomonaStudent` objects? What about for lists?

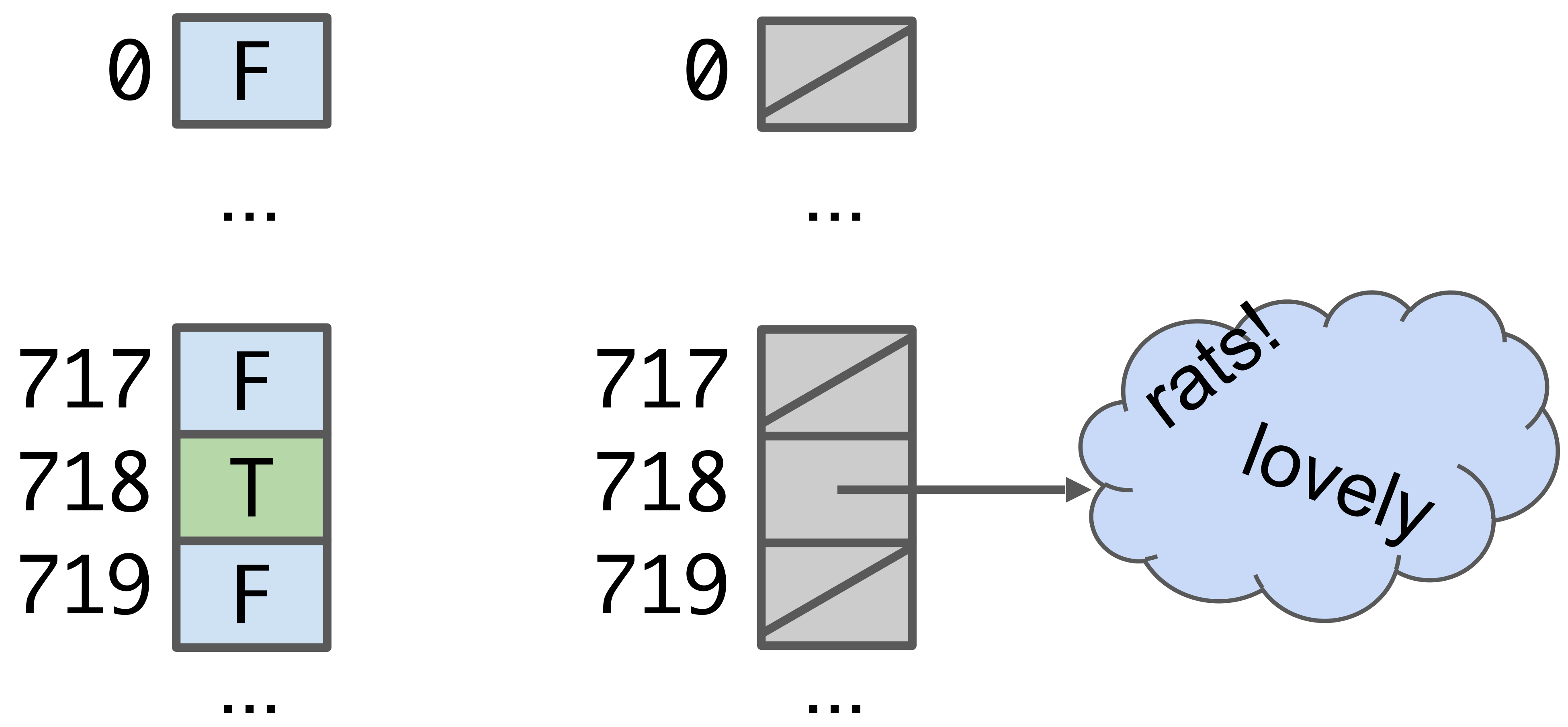
Solution for collisions: buckets

Suppose *N* items have the same numerical representation *h*:

- Example: hash code for “rats!” and “lovely” might both be 718.
- Instead of storing `true` in position *h*, store a “bucket” of these *N* items at position *h*.

How to implement a “bucket”?

- Conceptually simplest way: Singly Linked List.
- Could also use ArrayLists.
- Will see it doesn’t really matter what you do.



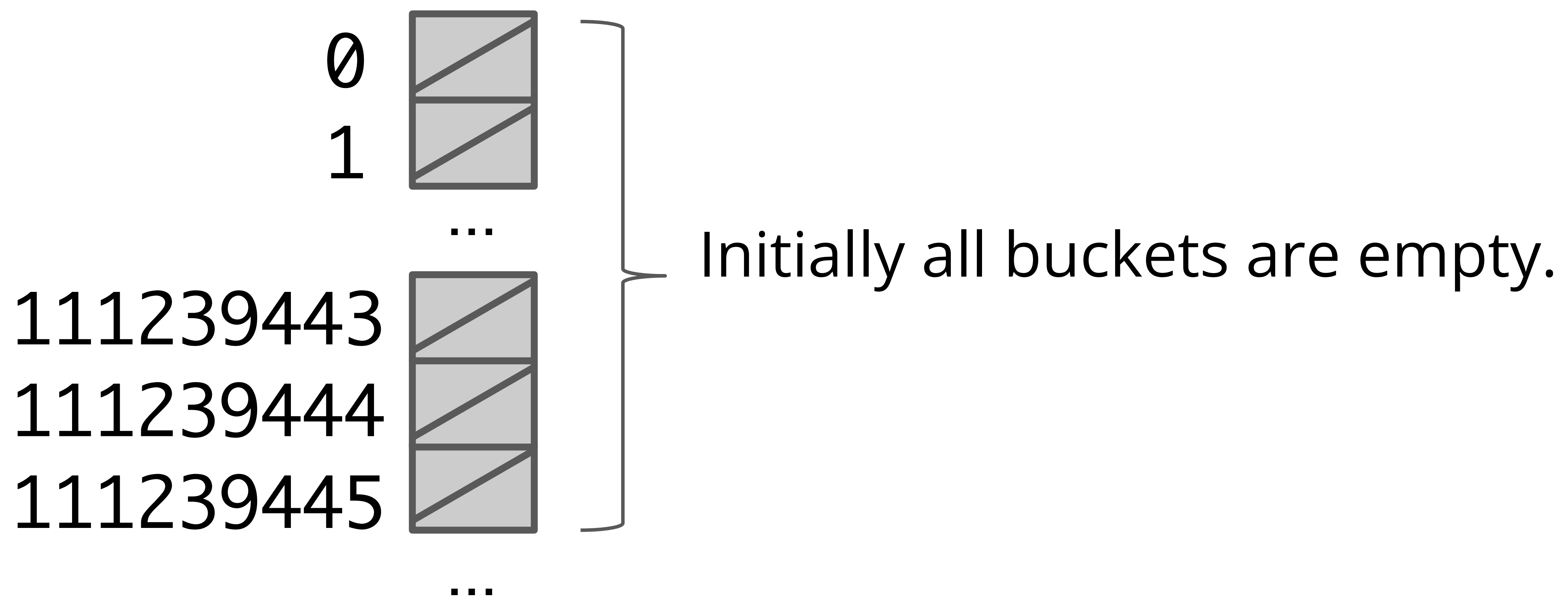
"Separate chaining data indexed array"

Each bucket in our array is initially empty. When an item *x* gets added at index *h*:

- If bucket *h* is empty, we create a new list containing *x* and store it at index *h*.
- If bucket *h* is already a list, we add *x* to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #*h* is a “separate chain” of all items that have hash code *h*.



"Separate chaining data indexed array"

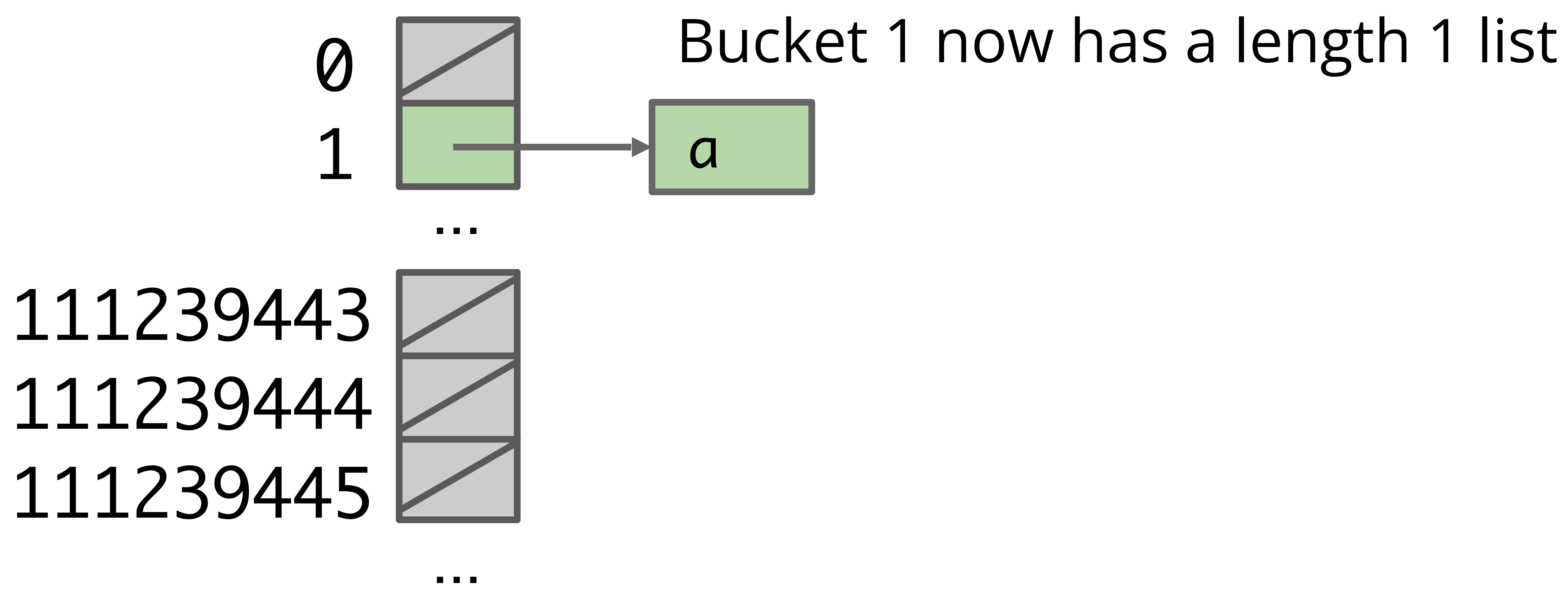
Each bucket in our array is initially empty. When an item *x* gets added at index *h*:

- If bucket *h* is empty, we create a new list containing *x* and store it at index *h*.
- If bucket *h* is already a list, we add *x* to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #*h* is a “separate chain” of all items that have hash code *h*.

```
add("a")
```



"Separate chaining data indexed array"

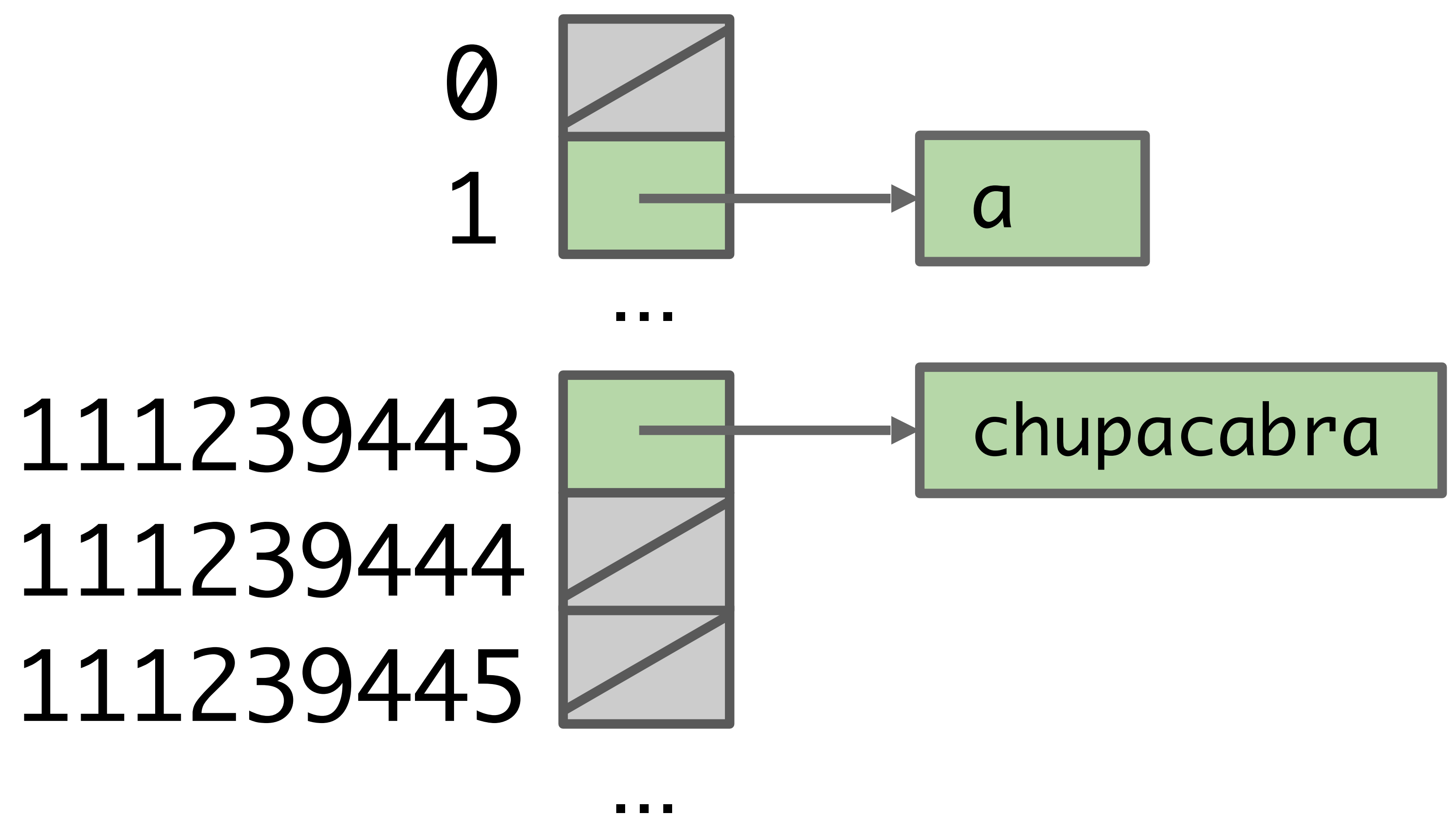
Each bucket in our array is initially empty. When an item *x* gets added at index *h*:

- If bucket *h* is empty, we create a new list containing *x* and store it at index *h*.
- If bucket *h* is already a list, we add *x* to this list if it is not already present.

We might call this a "separate chaining data indexed array".

- Bucket #*h* is a "separate chain" of all items that have hash code *h*.

```
add("a")
add("chupacabra")
```



"Separate chaining data indexed array"

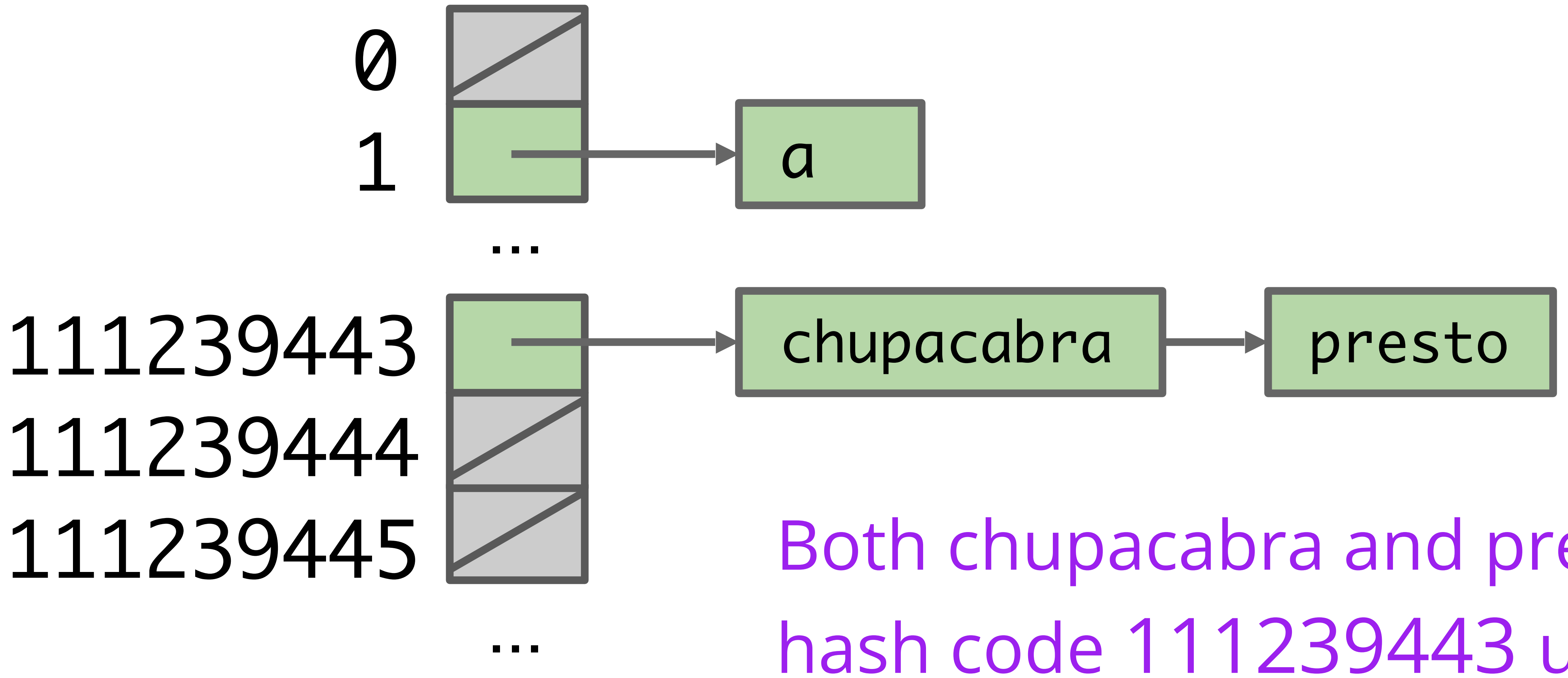
Each bucket in our array is initially empty. When an item *x* gets added at index *h*:

- If bucket *h* is empty, we create a new list containing *x* and store it at index *h*.
- If bucket *h* is already a list, we add *x* to this list if it is not already present.

We might call this a “separate chaining data indexed array”.

- Bucket #*h* is a “separate chain” of all items that have hash code *h*.

```
add("a")
add("chupacabra")
add("presto")
```



Both chupacabra and presto have hash code 111239443 using `englishToInt()`

"Separate chaining data indexed array"

Each bucket in our array is initially empty. When an item *x* gets added at index *h*:

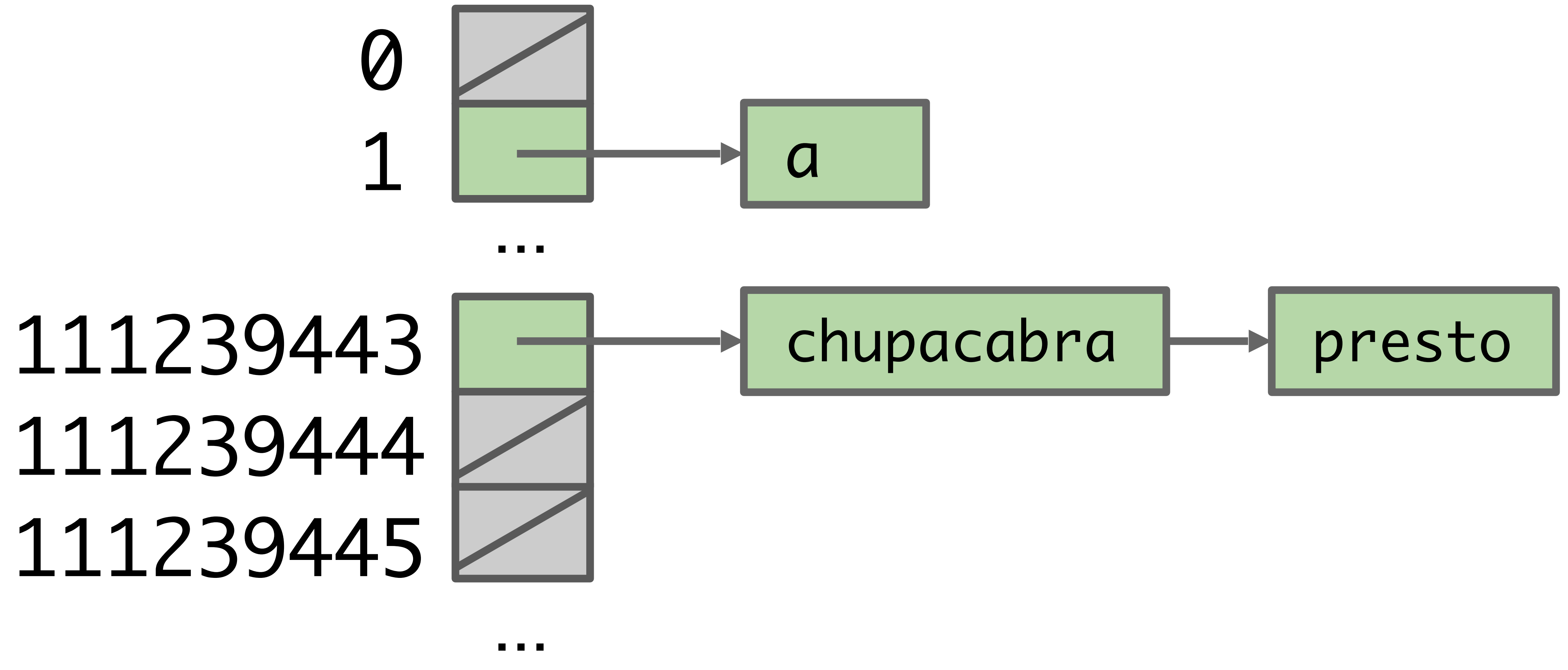
- If bucket *h* is empty, we create a new list containing *x* and store it at index *h*.
- If bucket *h* is already a list, we add *x* to this list if **it is not already present.**
  - Note, if we were storing key/value pairs instead, we would update the value instead.

We might call this a “separate chaining data indexed array”.

- Bucket #*h* is a “separate chain” of all items that have hash code *h*.

```
add("a")
add("chupacabra")
add("presto")
add("chupacabra")
```

Doesn’t do anything, because chupacabra is already present



"Separate chaining data indexed array"

Each bucket in our array is initially empty. When an item *x* gets added at index *h*:

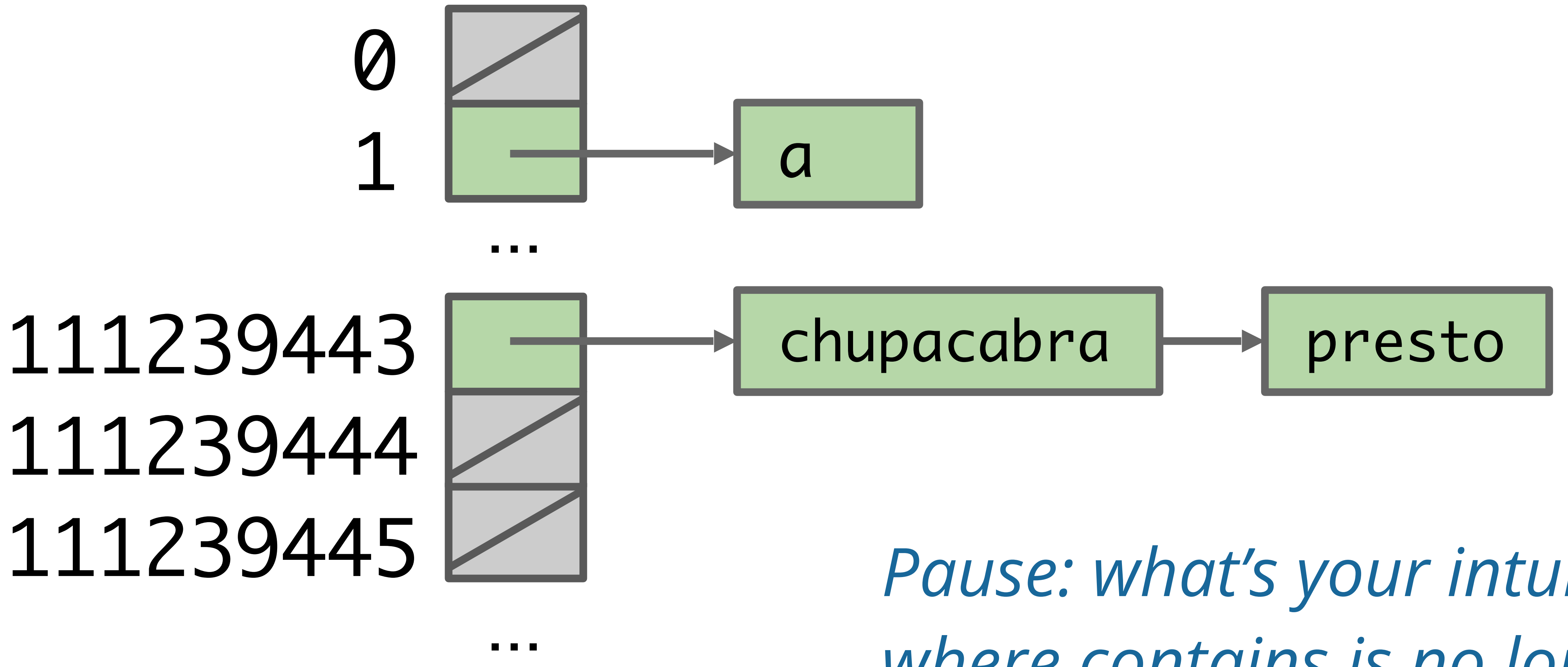
- If bucket *h* is empty, we create a new list containing *x* and store it at index *h*.
- If bucket *h* is already a list, we add *x* to this list if **it is not already present.**

We might call this a “separate chaining data indexed array”.

- Bucket #*h* is a “separate chain” of all items that have hash code *h*.

```
add("a")
add("chupacabra")
add("presto")
add("chupacabra")
contains("presto")
```

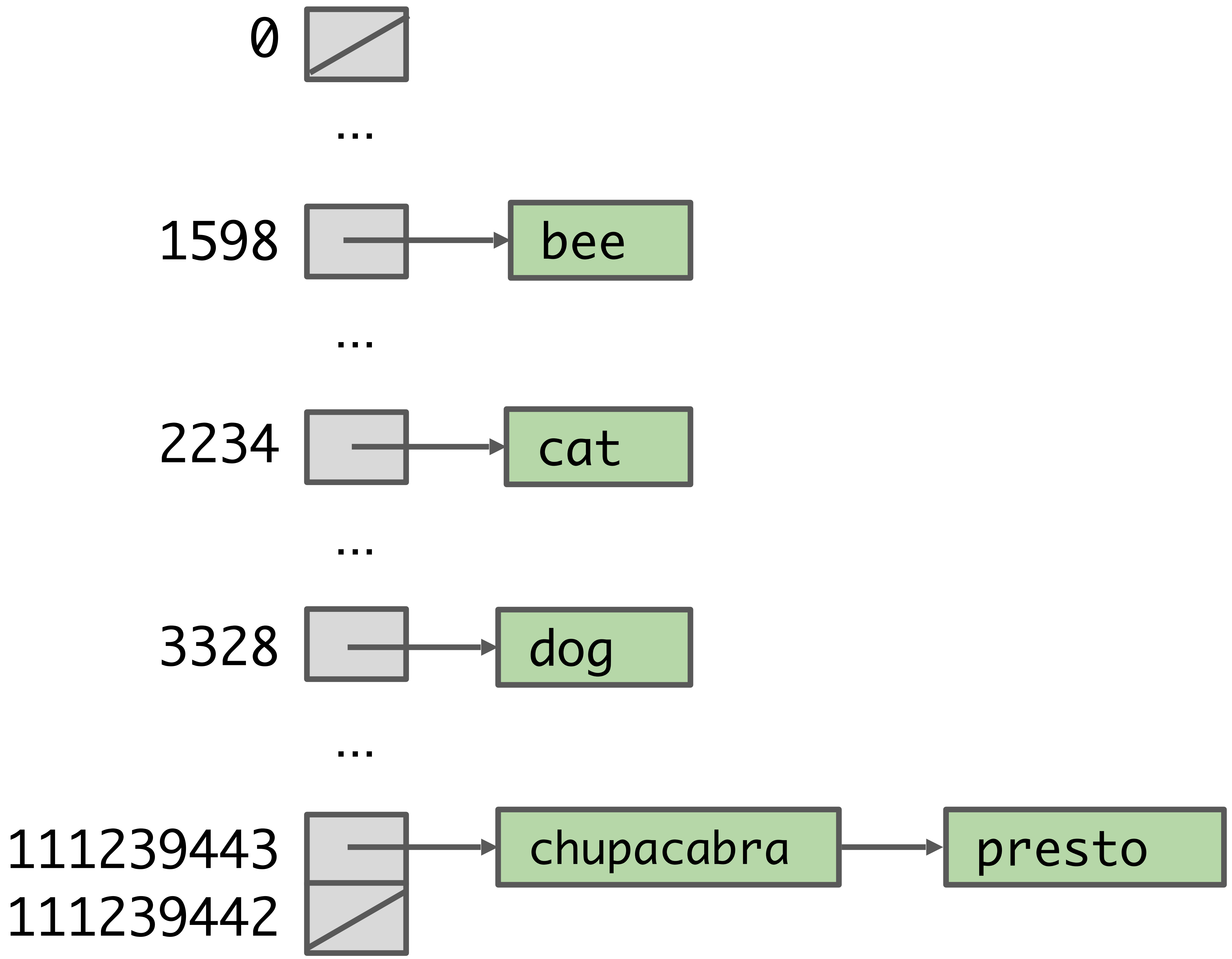
Traverse the list in bucket 111239443 to see if presto exists



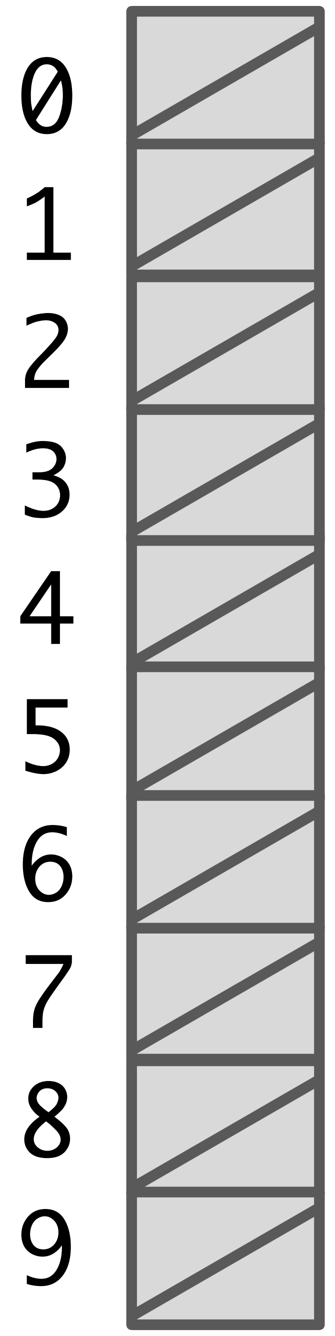
Pause: what’s your intuition of a scenario where contains is no longer O(1), but O(Q) where Q is the length of the linked list?

Saving Memory Using Separate Chaining

Observation: We don't really need billions of buckets.

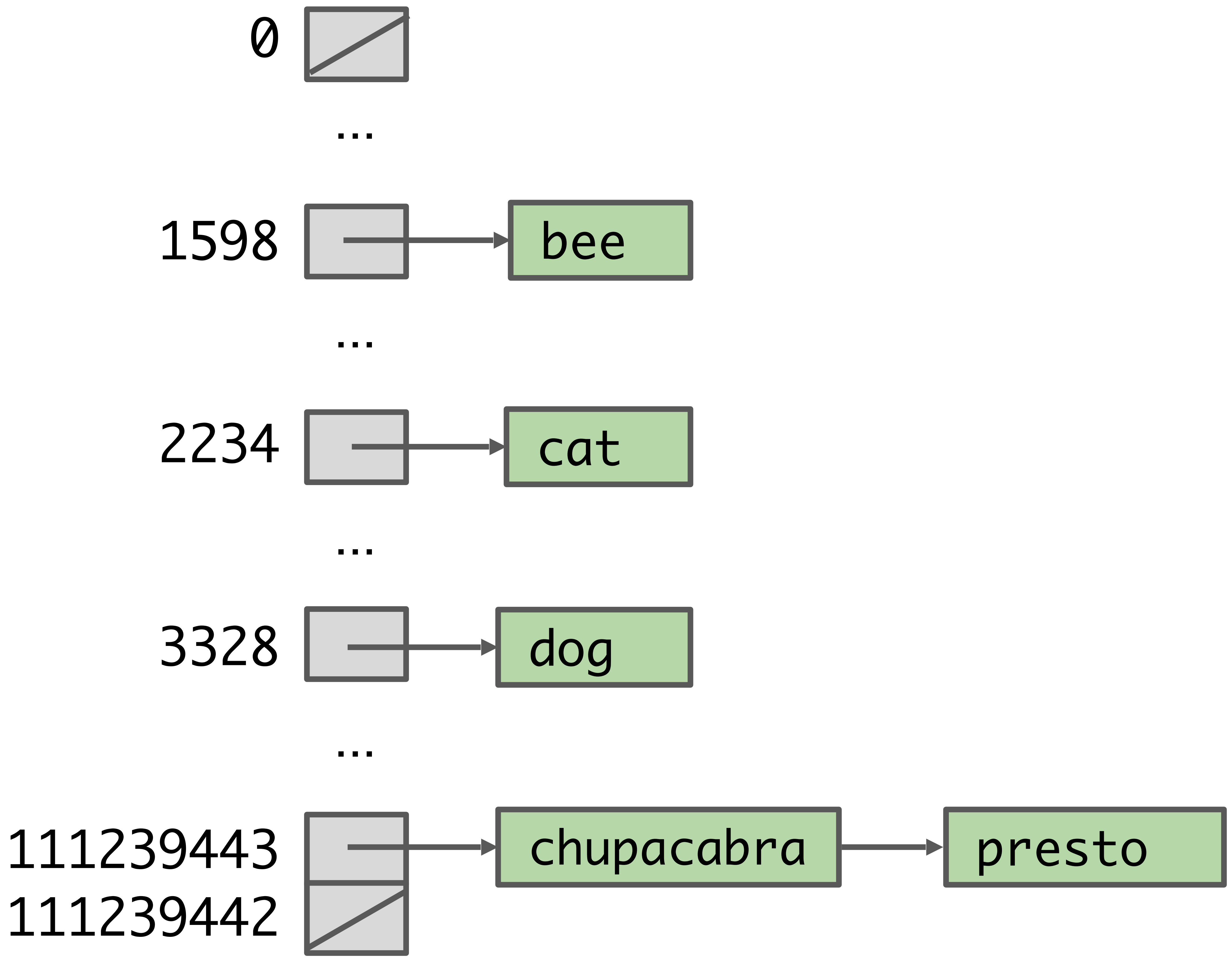


Q: If we use the 10 buckets on the right, where should our five items go?



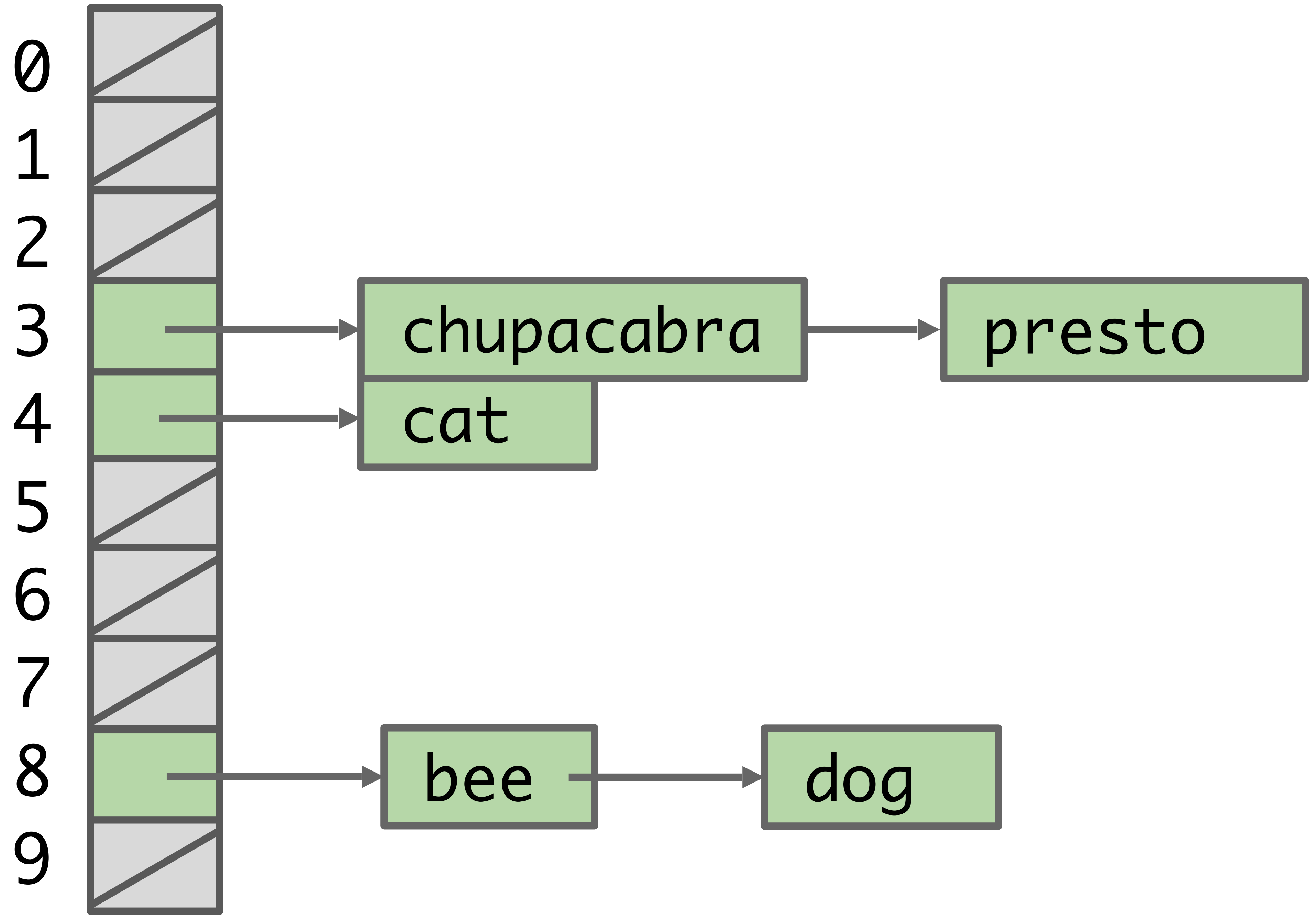
Saving Memory Using Separate Chaining and modulus

We can use the modulus operator to reduce bucket count. (Downside is that the lists will be longer.)



Q: If we use the 10 buckets on the right, where should our five items go?

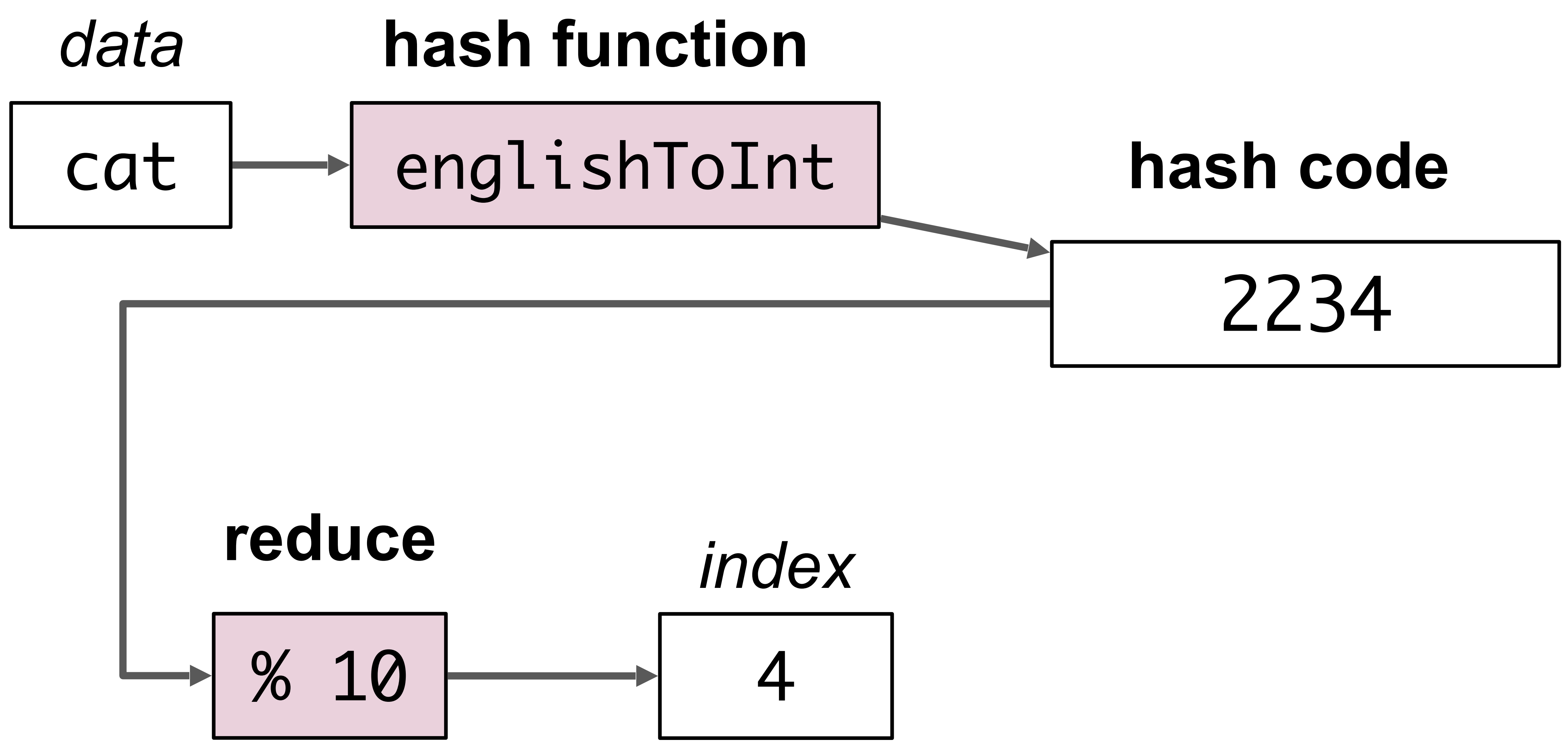
A: Try bucket = hash code % 10 (i.e., look at the last digit of the hash code)

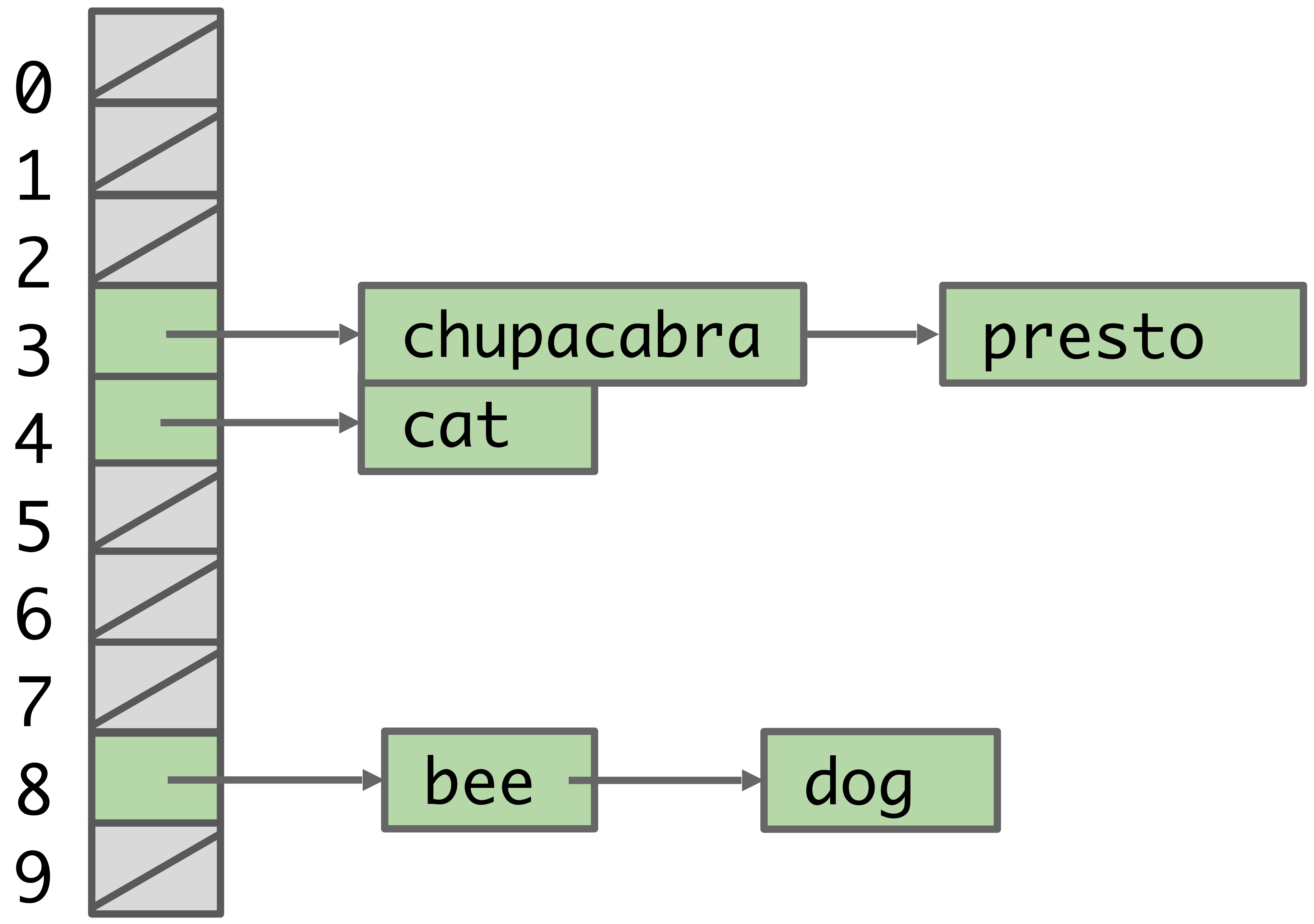


The Hash Table

What we've just created here is called a hash table.

- ***Data*** is converted by a hash function into an integer representation called a hash code.
- The hash code is then reduced to a valid *index*, usually using the modulus operator, e.g. 2348762878 % 10 = 8.





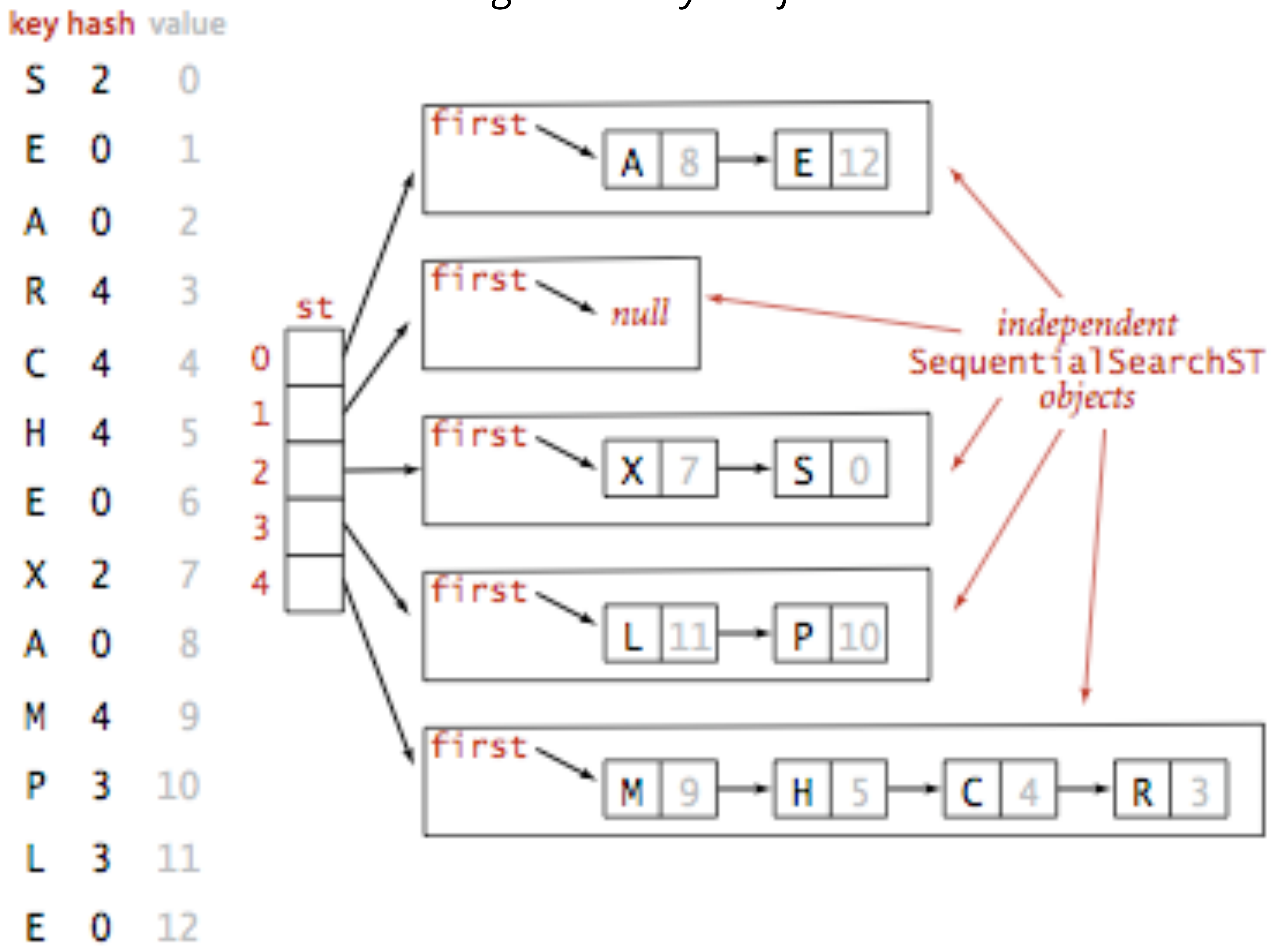
A hash table!

Separate/External Chaining (Closed Addressing)

- The formal name for what we've learned is called "separate chaining", "external chaining", or "closed addressing". (Why can't computer scientists just give one concept one name?)
- Use an array of $m < n$ distinct linked lists (chains) [H.P. Luhn, IBM 1953].
  - Hash: Map key to integer i between 0 and $m - 1$.
  - Insert: Put key-value pair **at front** of i-th chain (if not already there in which case we only update the associated value).
  - Search: Need to only search the i-th chain.

m = # of buckets/chains in a hashtable
n = total elements

Note: In our textbook example, we store key-value pairs, while we've just been talking about keys so far in lecture



Hashing with separate chaining for standard indexing client

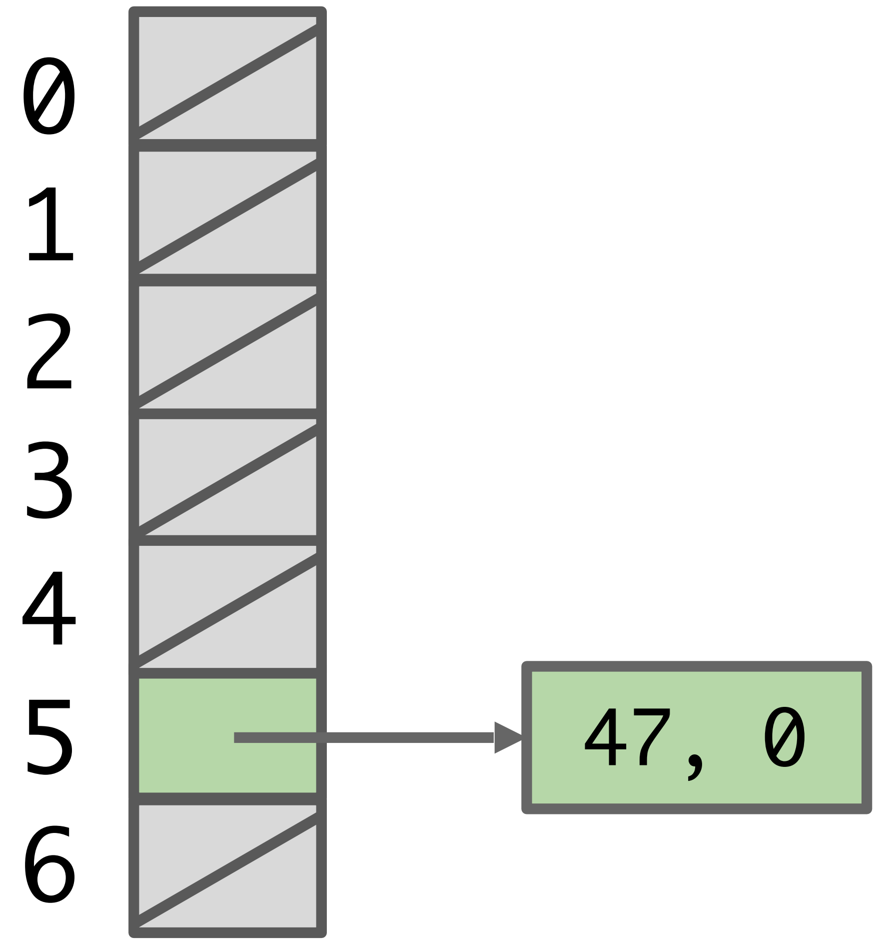
Note: in lecture, we saw putting it at the back of the chain, but the textbook example puts it at the front. (Why?) (More recent data quicker to access.)

Worksheet time!

- Assume a dictionary implemented using hashing and separate chaining for handling collisions.
- Let $m = 7$ be the hash table size.
- For simplicity, we will assume that keys are integers and that the hash value for each key k is calculated as $h(k) = k \% m$.
- Insert the key-value pairs (47, 0), (3, 1), (28, 2), (14, 3), (9,4), (47,5) and show the resulting hash table.

As an example, for (47, 0)

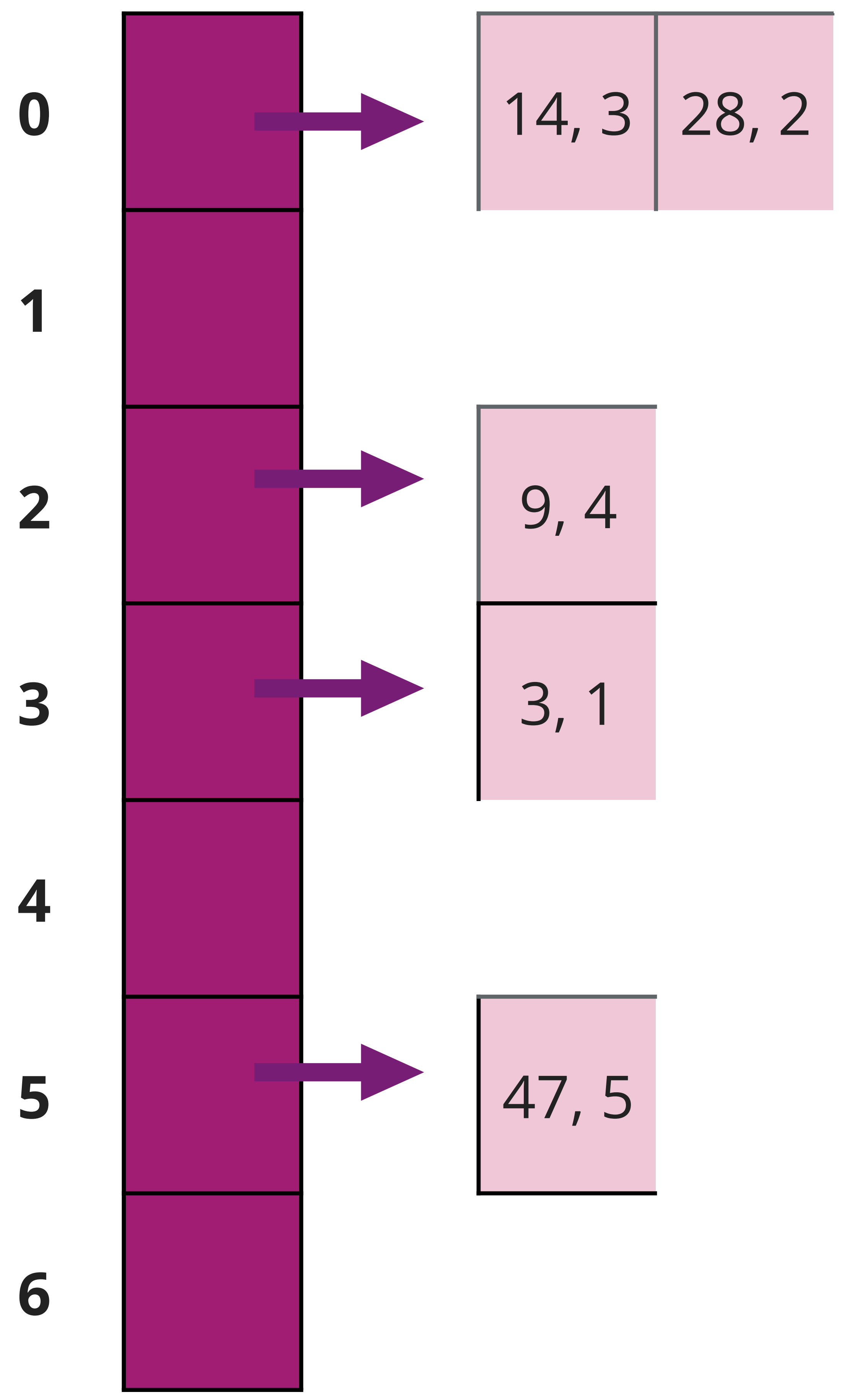
Hash value of 47 = 47 % 7 = 5



store key/value pairs!

Worksheet answers

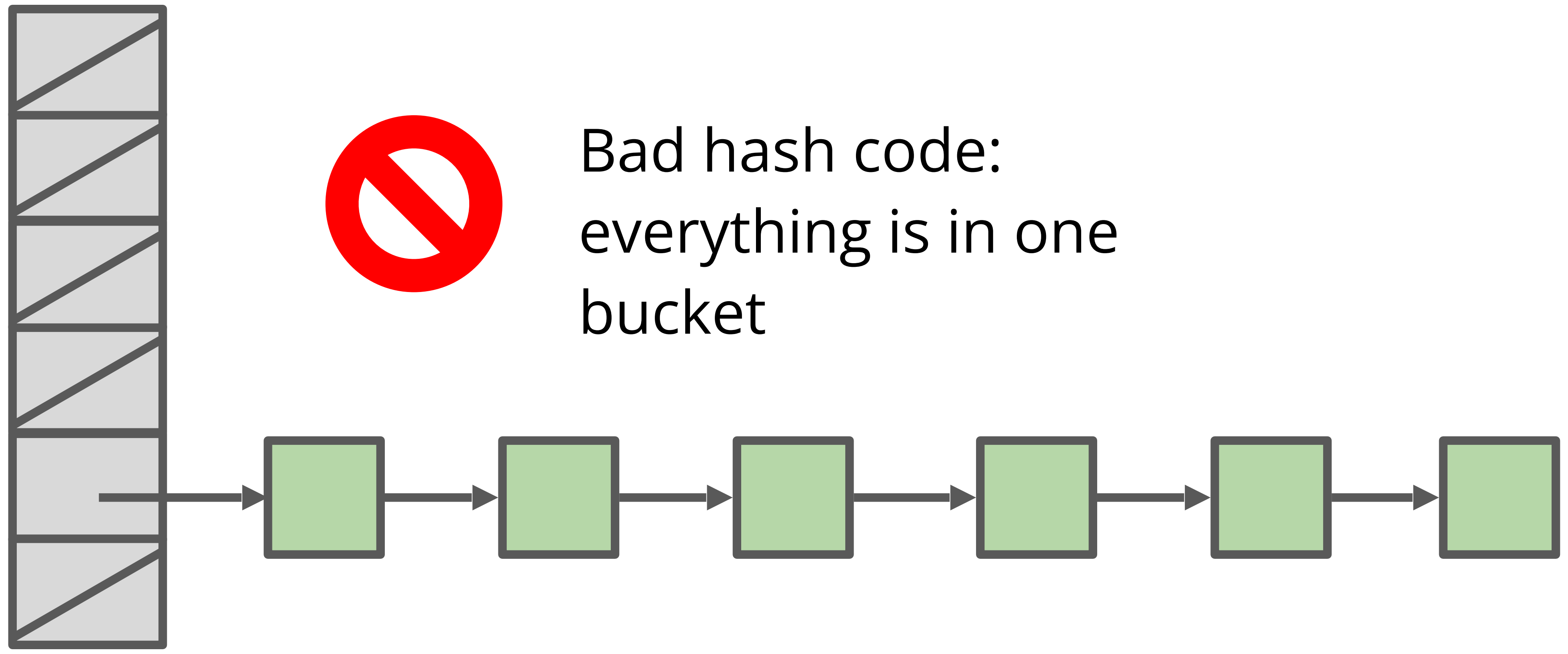
Key	Hash	Value
47	5	0
3	3	1
28	0	2
14	0	3
9	2	4
47	5	5

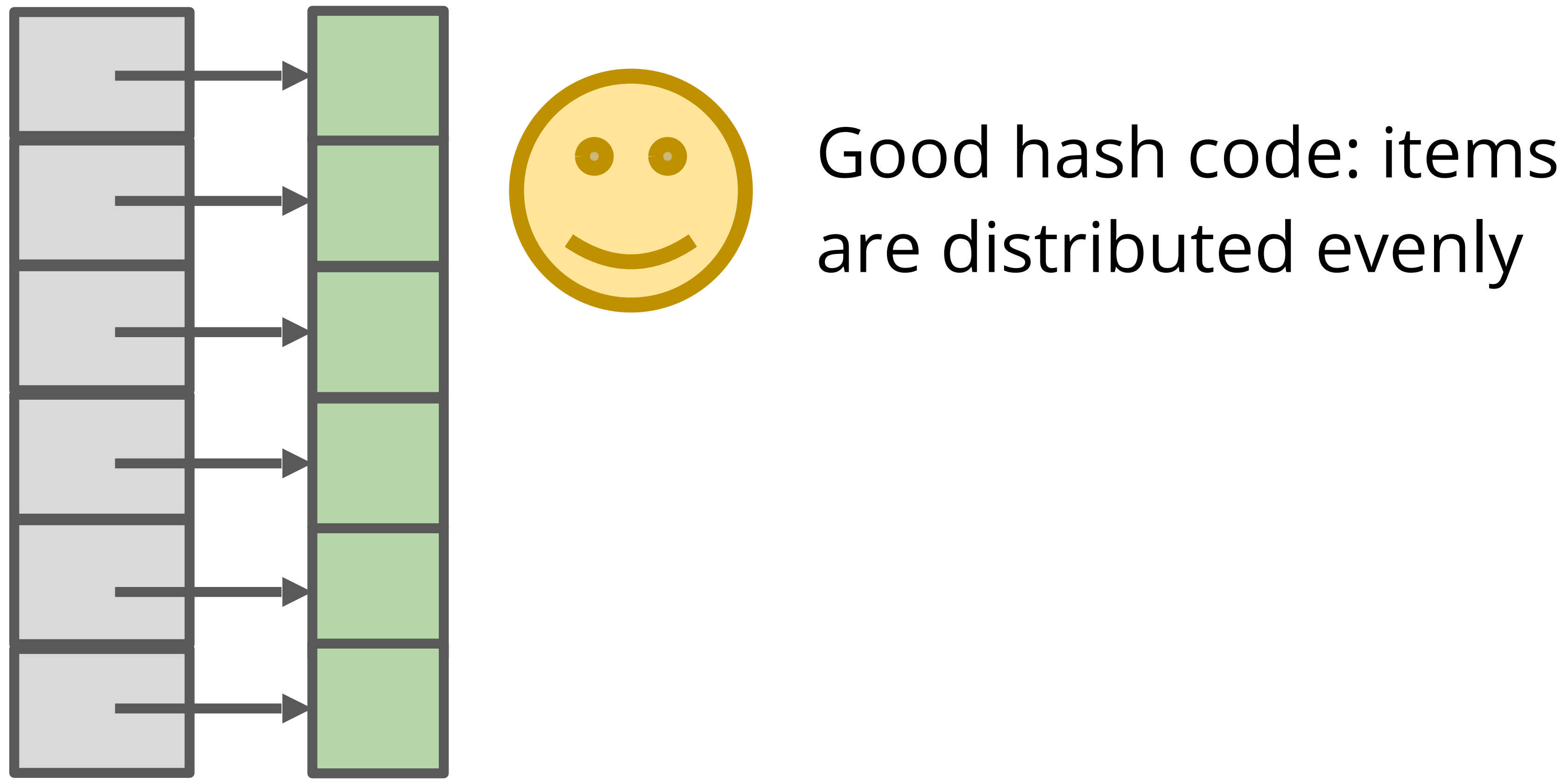


Hash Table Resizing

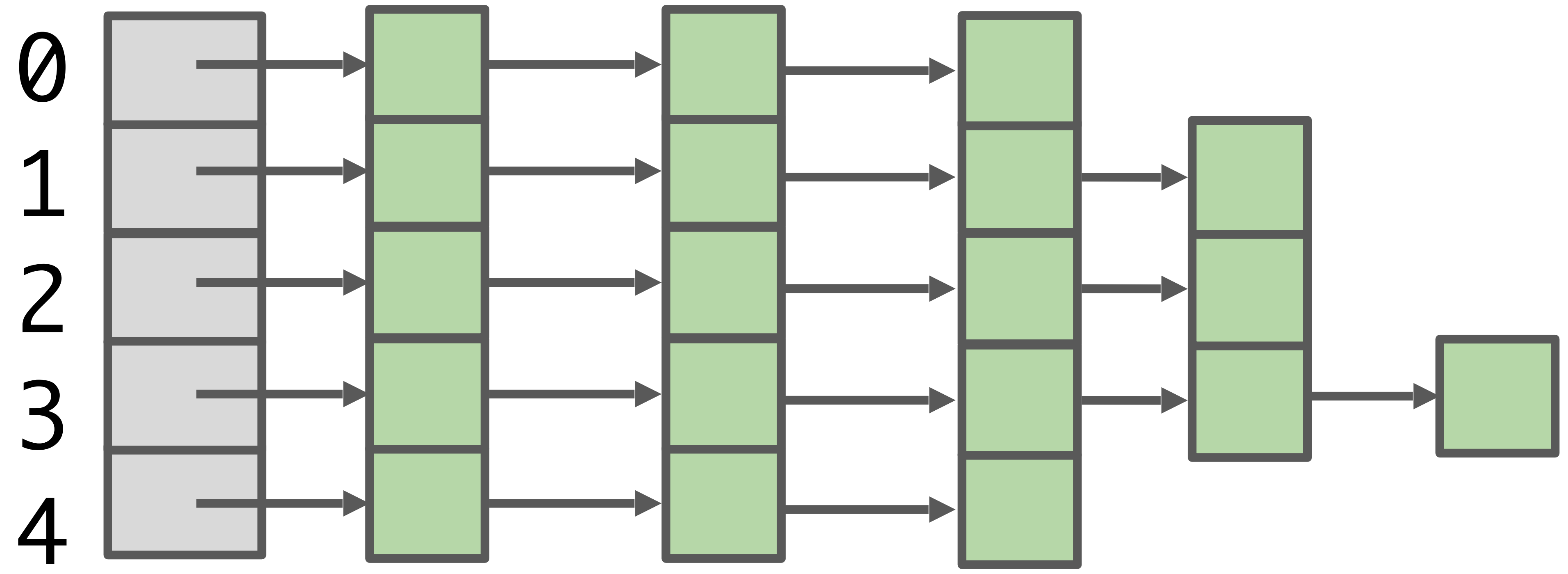
What makes a good hash function?

- We want a hash function that spreads things out nicely on real data.
  - Example #1: return 0 is a bad hash code function.
  - Example #2: just returning the first character of a word, e.g. “cat” → 3 was also a bad hash function.
  - Example #3: adding chars together is bad. “ab” collides with “ba”.
  - Example #4: returning string treated as a base B number can be good!
- A good hash function is hard to write, but it should scramble data seemingly randomly, so they will be *evenly distributed* over the hash table.





Improving the hash table



M = 5,
N = 19

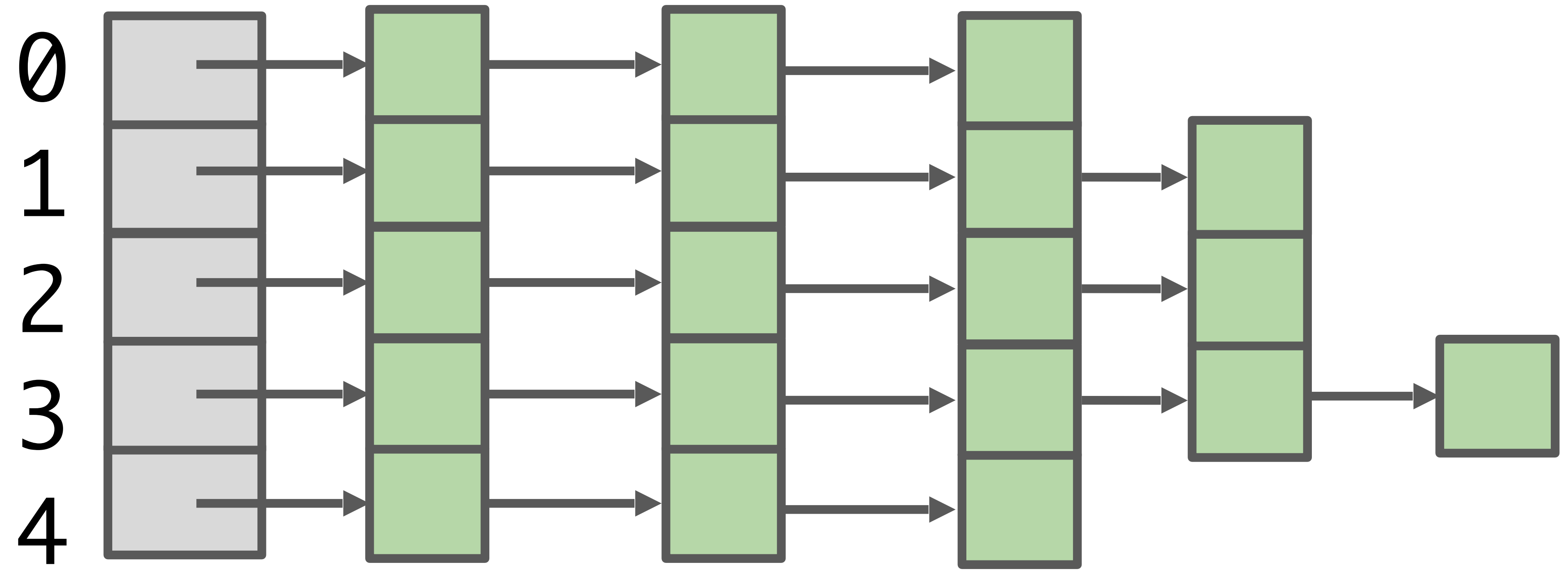
Suppose we have:

- A fixed number of buckets M.
- An increasing number of items N.

Major problem: Even if items are spread out evenly, lists are of length Q = N/M.

- For M = 5, that means the list length Q will scale to the number of items N, which results in linear time operations.
  - The best case is all items are evenly distributed, so Q is N/5. The worst case is all the items are in one bucket, so Q is N.
- Our goal: How can we improve our design to guarantee constant time operations? In other words, how can we make N/M = O(1)?

Improving the hash table via resizing



M = 5,
N = 19

Suppose we have:

- An increasing number of buckets M.
- An increasing number of items N.

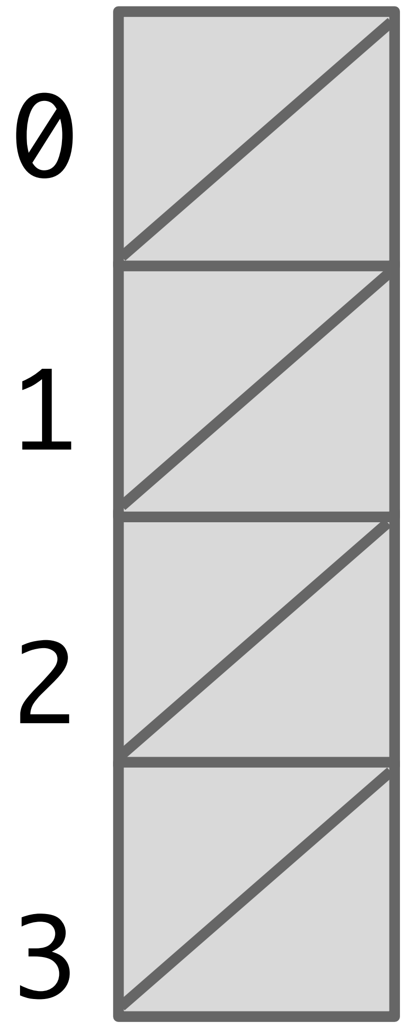
Major problem: Even if items are spread out evenly, lists are of length Q = N/M.

- We can **resize the hash table** and **increase the number of buckets** to scale to the number of items.
  - For example, if we want an average of 1.5 (a constant) number of items in our lists, then N/M = 1.5, so here, instead we would have M = 13 buckets for our N = 19 items.
- N/M is called the "**load factor**" – how "full" the hash table is.

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

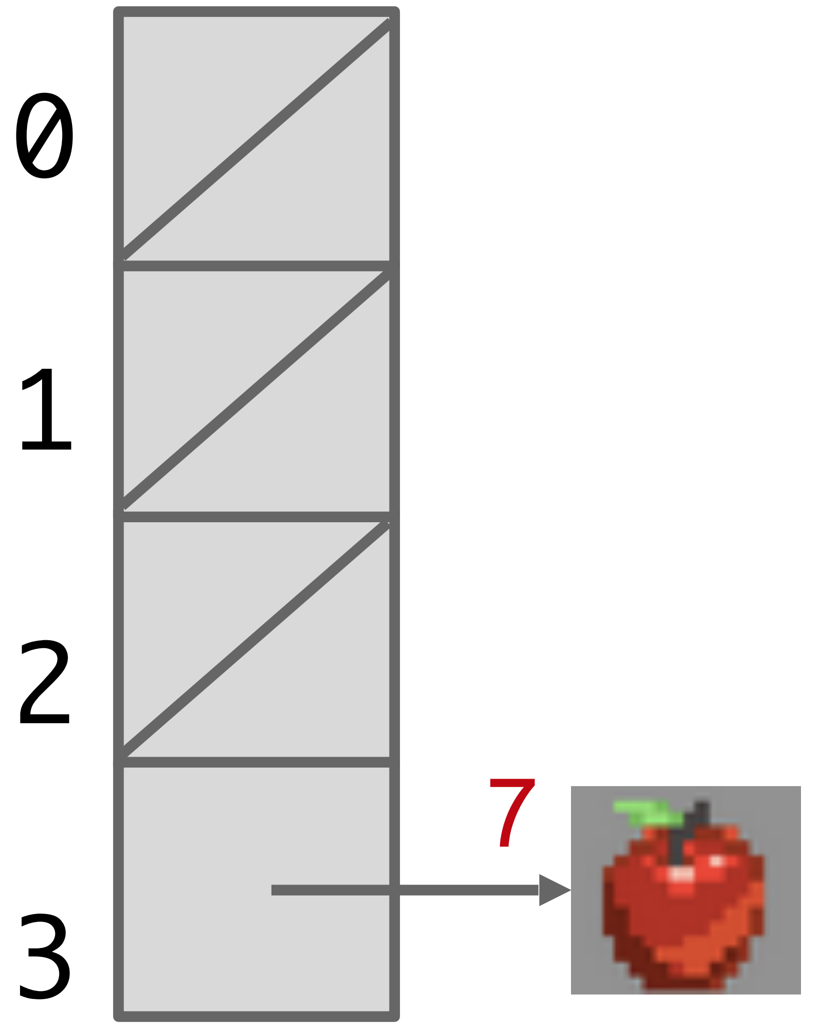
N = 0 M = 4 N / M = 0



Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 1 M = 4 N / M = 0.25

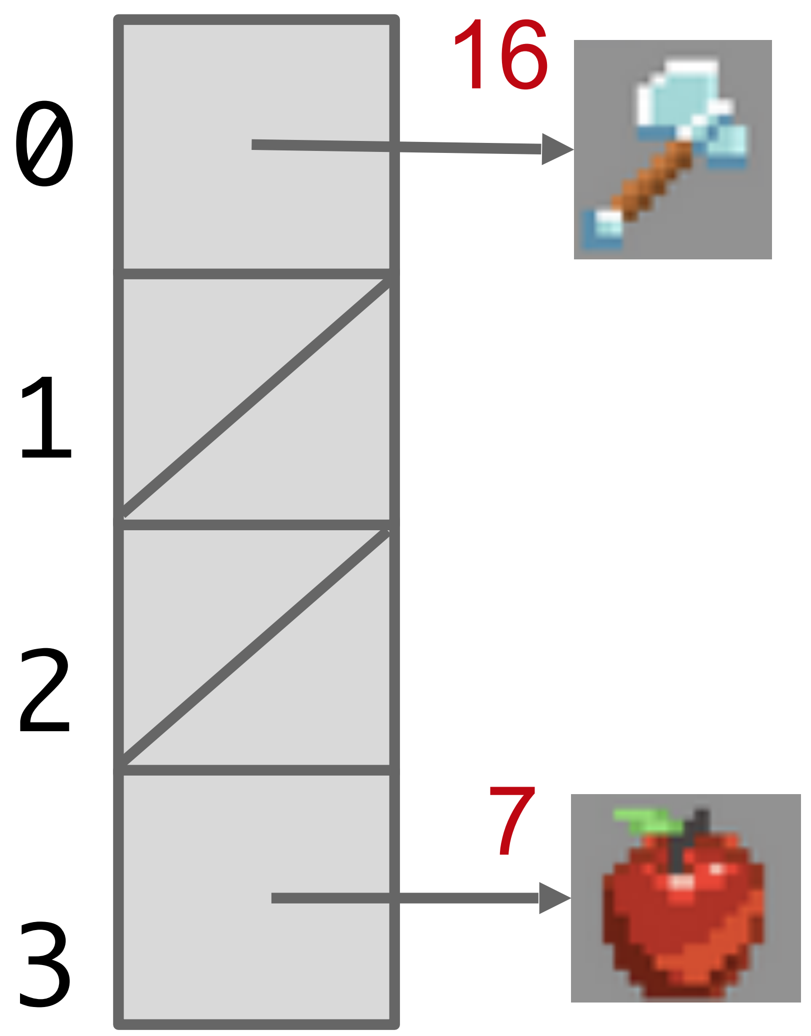


Red number = hashcode result

Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

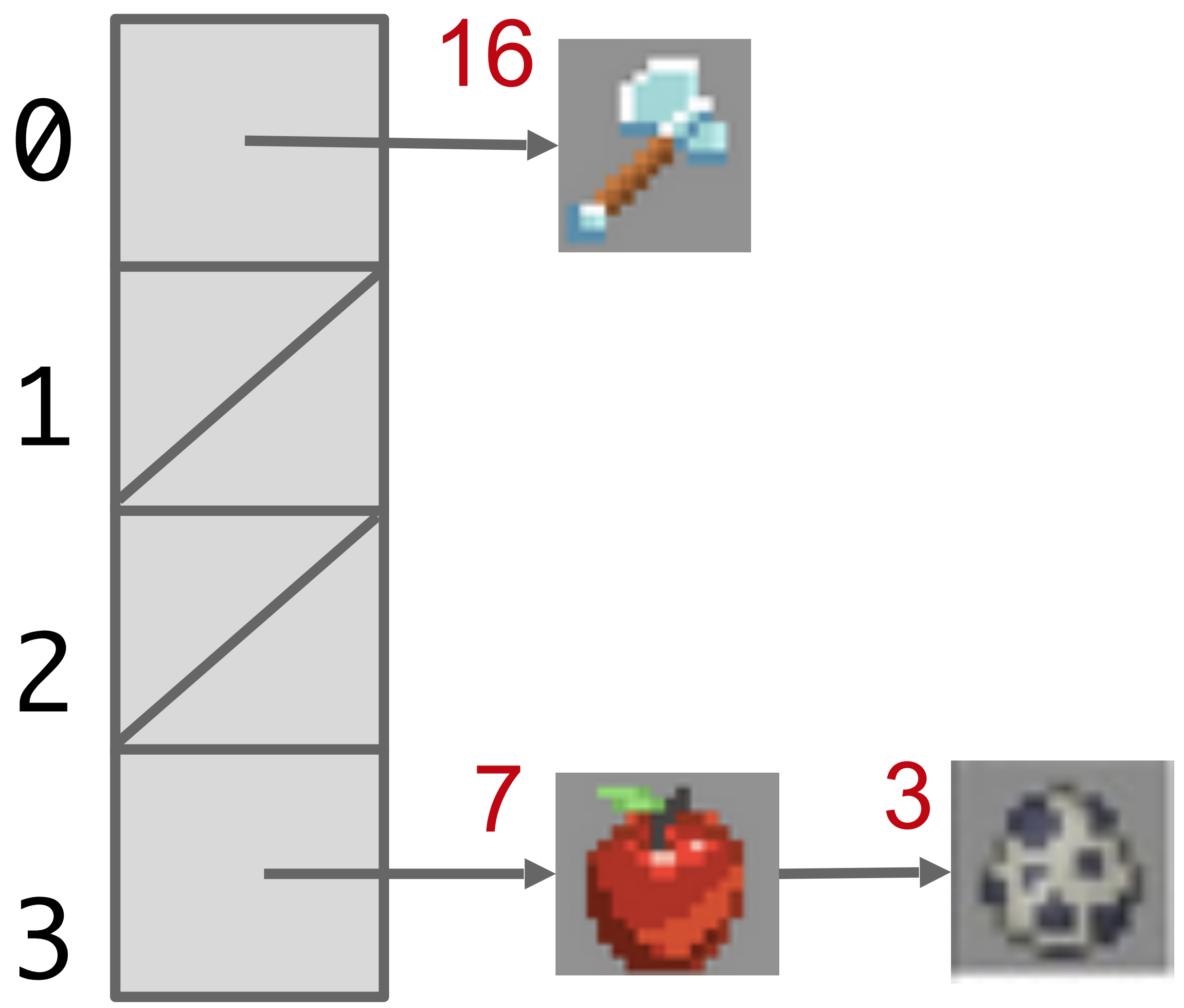
N = 2 M = 4 N / M = 0.5



Hash Table Resizing Example

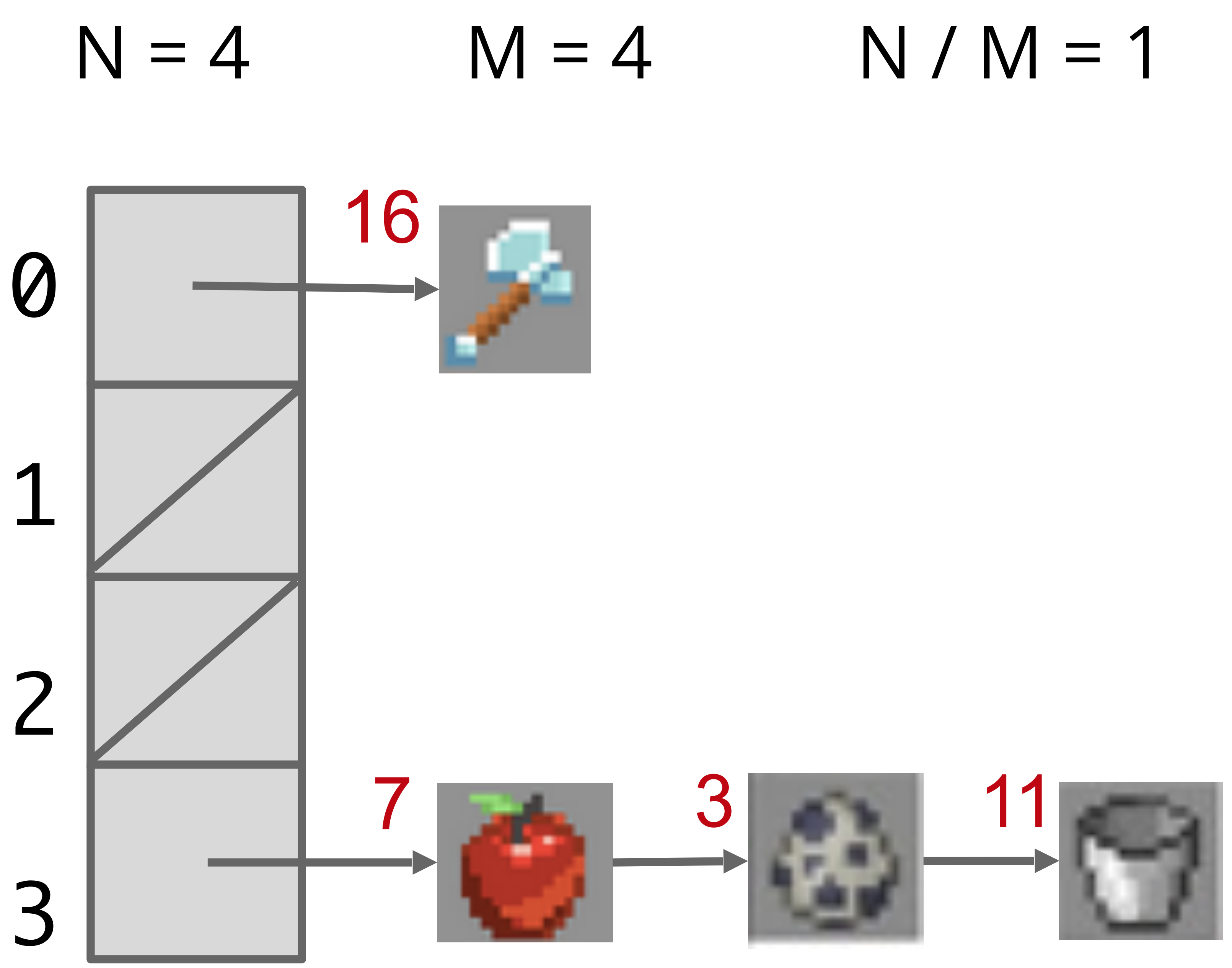
When N/M is ≥ 1.5, then double M.

N = 3 M = 4 N / M = 0.75



Hash Table Resizing Example

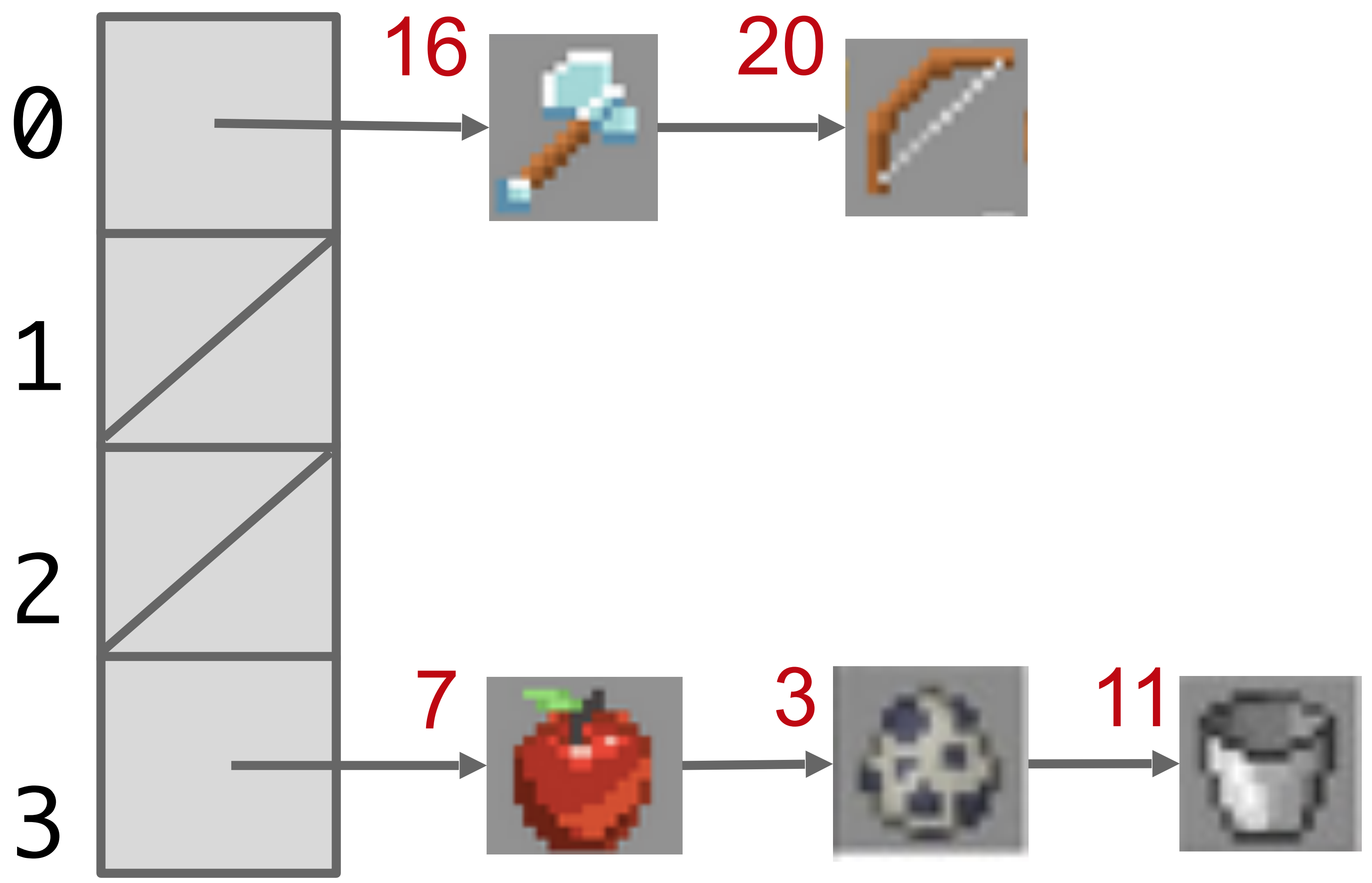
When N/M is ≥ 1.5, then double M.



Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

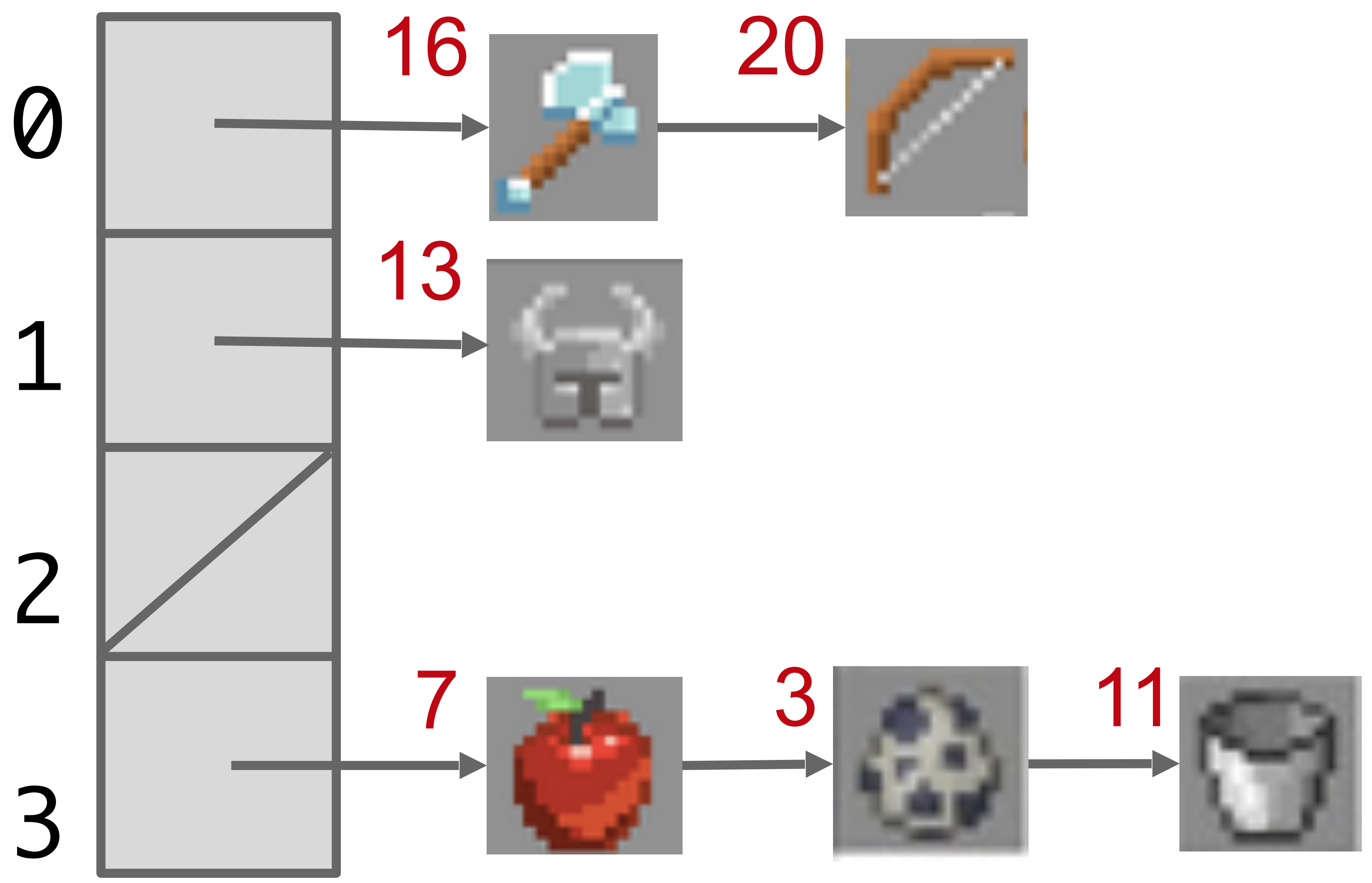
N = 5 M = 4 N / M = 1.25



Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

N = 6 M = 4 N / M = 1.5



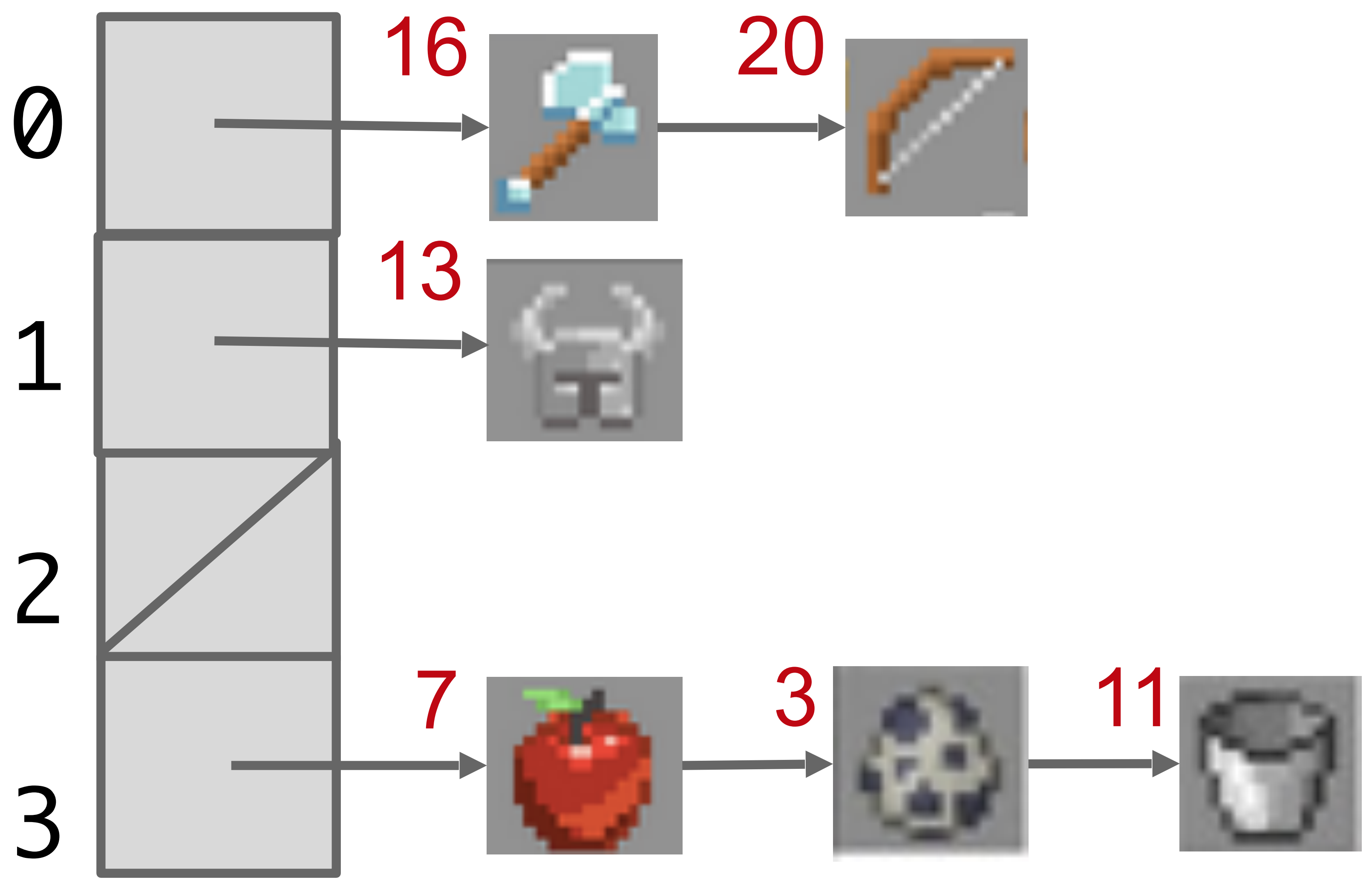
N/M is too large.
Time to double the number of buckets!

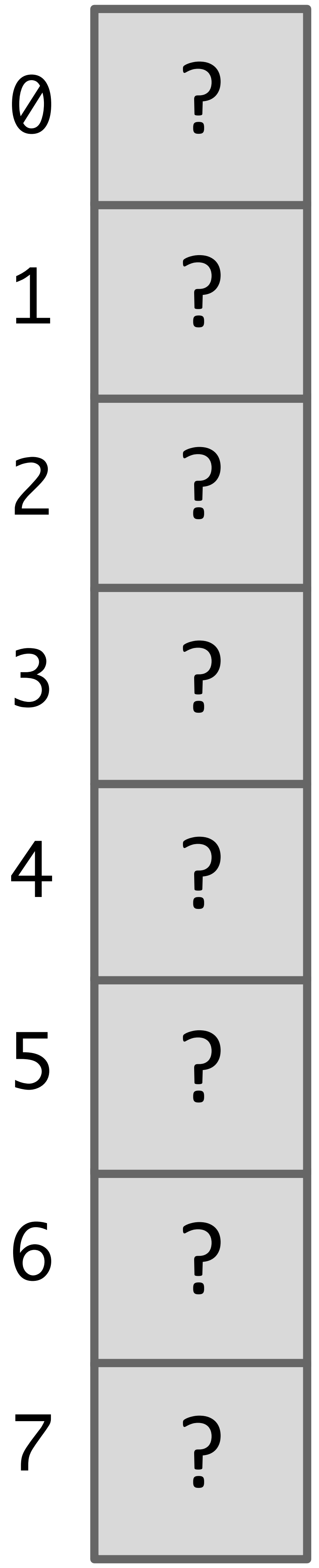
Worksheet time!

Where do all the existing elements go on the new hashtable? Assume we are placing items a
the end, instead of beginning, of the list.
(Hint: how do we reduce hashes to bucket indices? How does that change with resize?)

When N/M is ≥ 1.5, then double M.

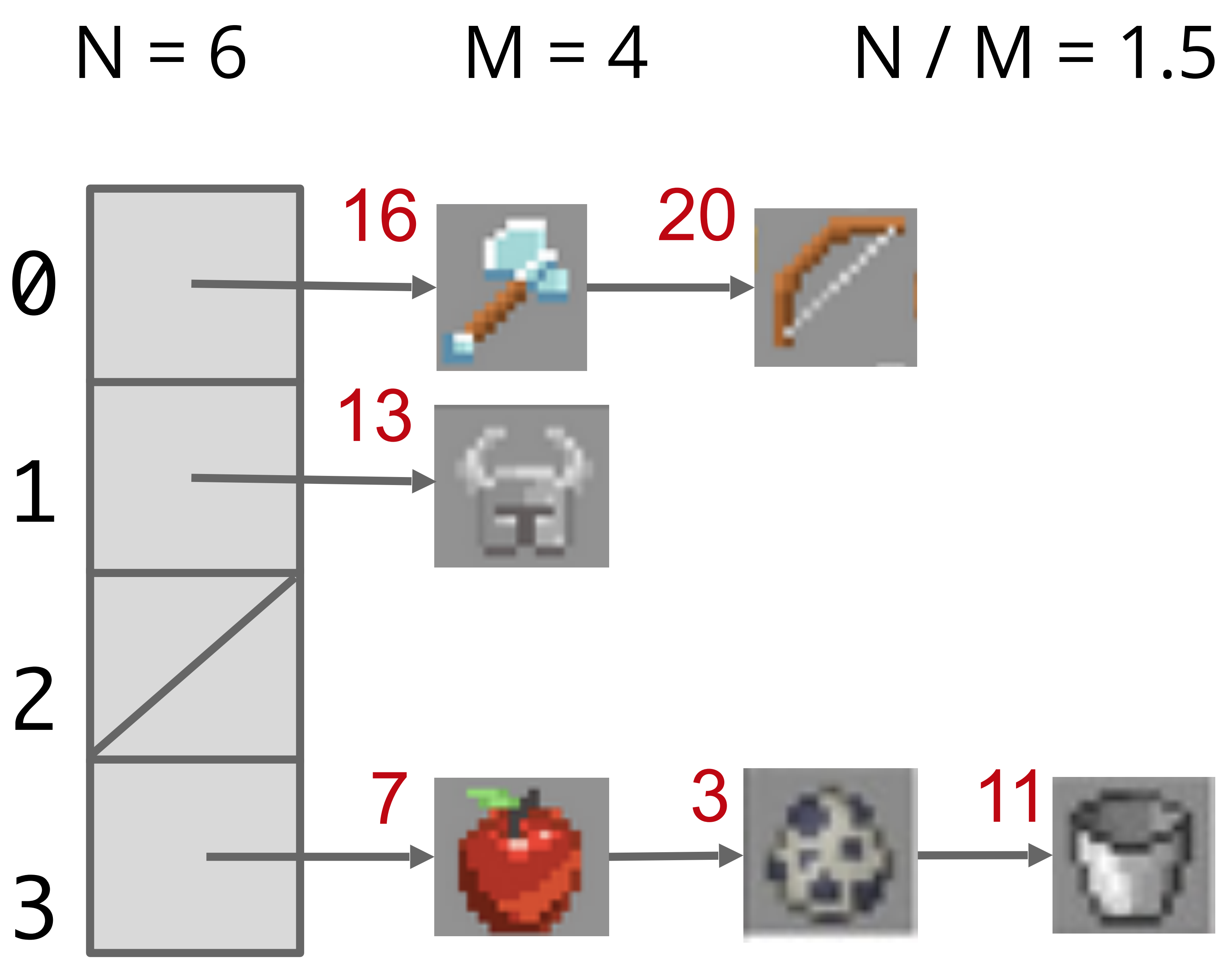
N = 6 M = 4 N / M = 1.5

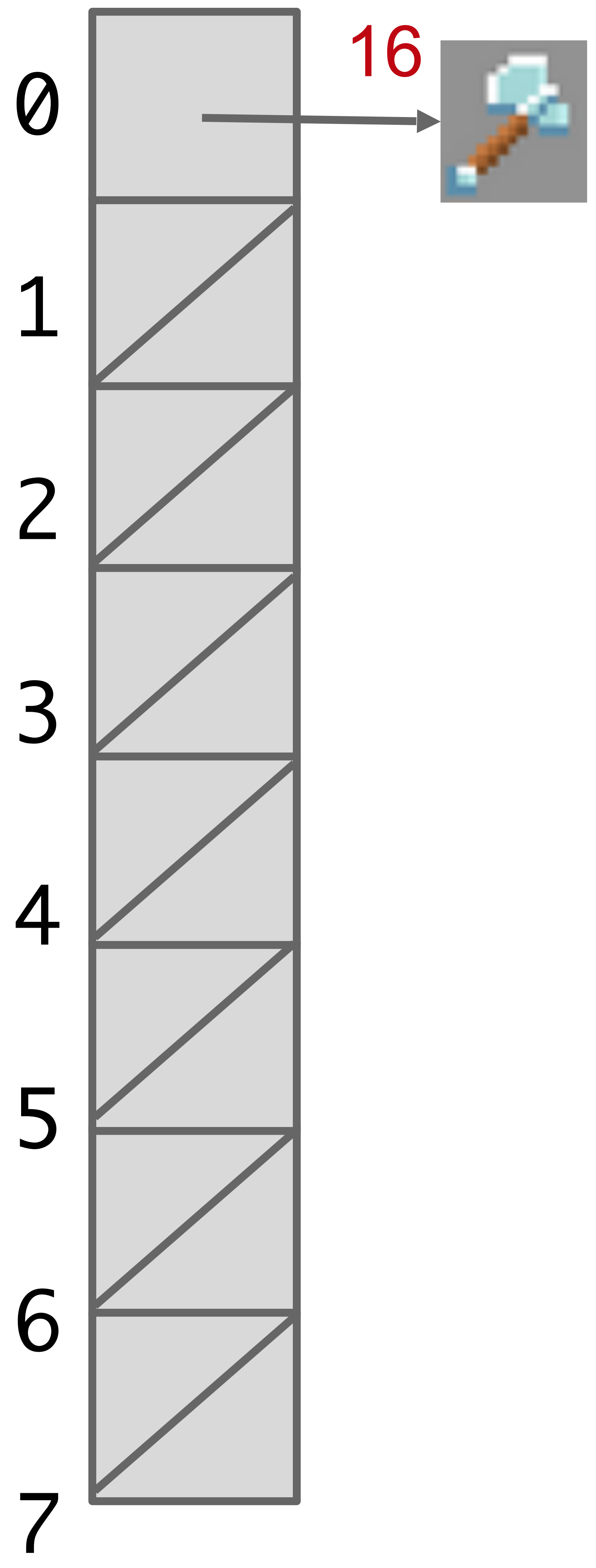




Worksheet answer: **hash % 8 instead of hash % 4**

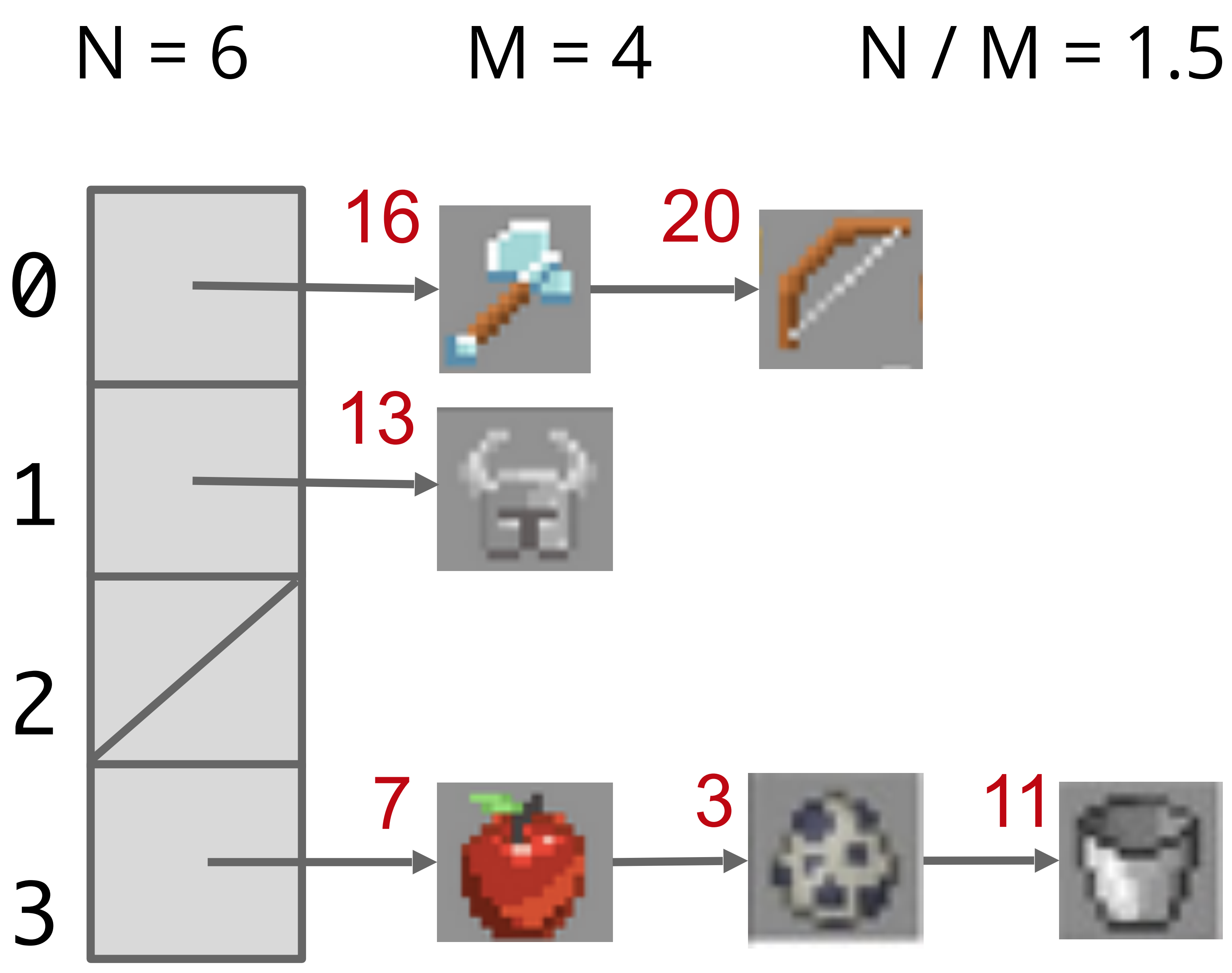
When N/M is ≥ 1.5, then double M.

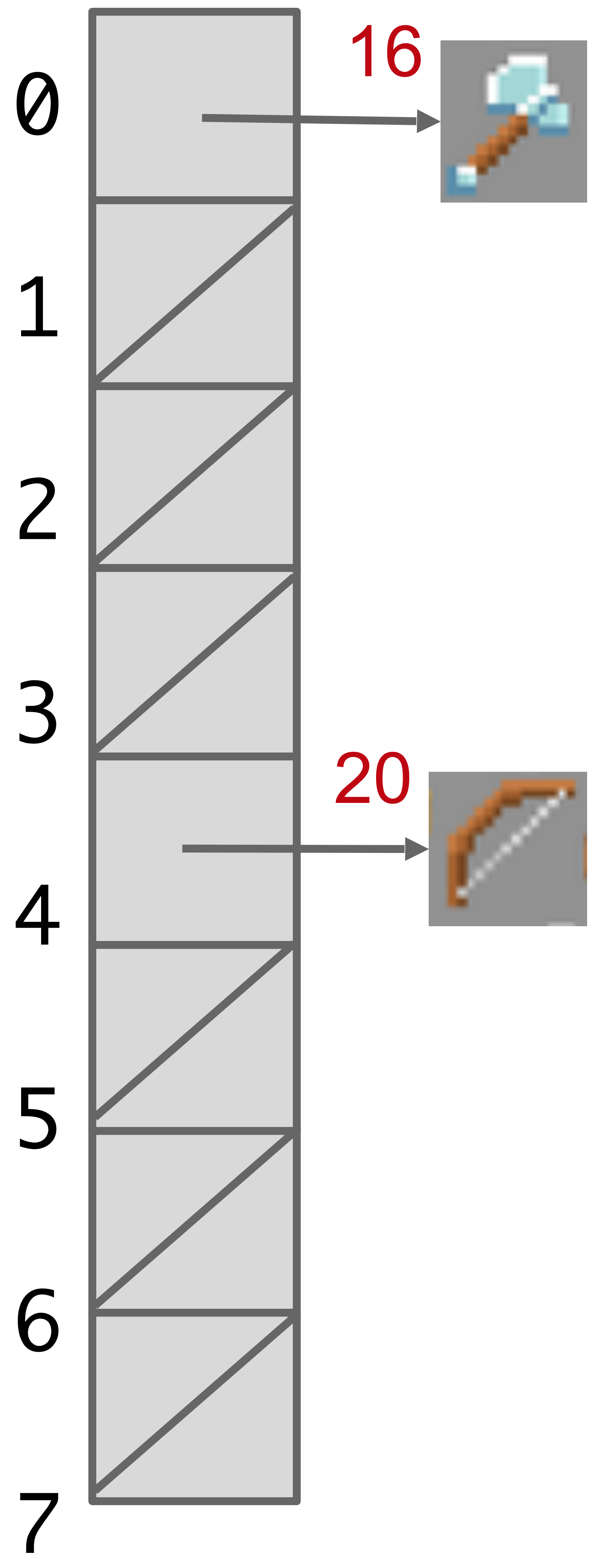




Worksheet answer: hash % 8 instead of hash % 4

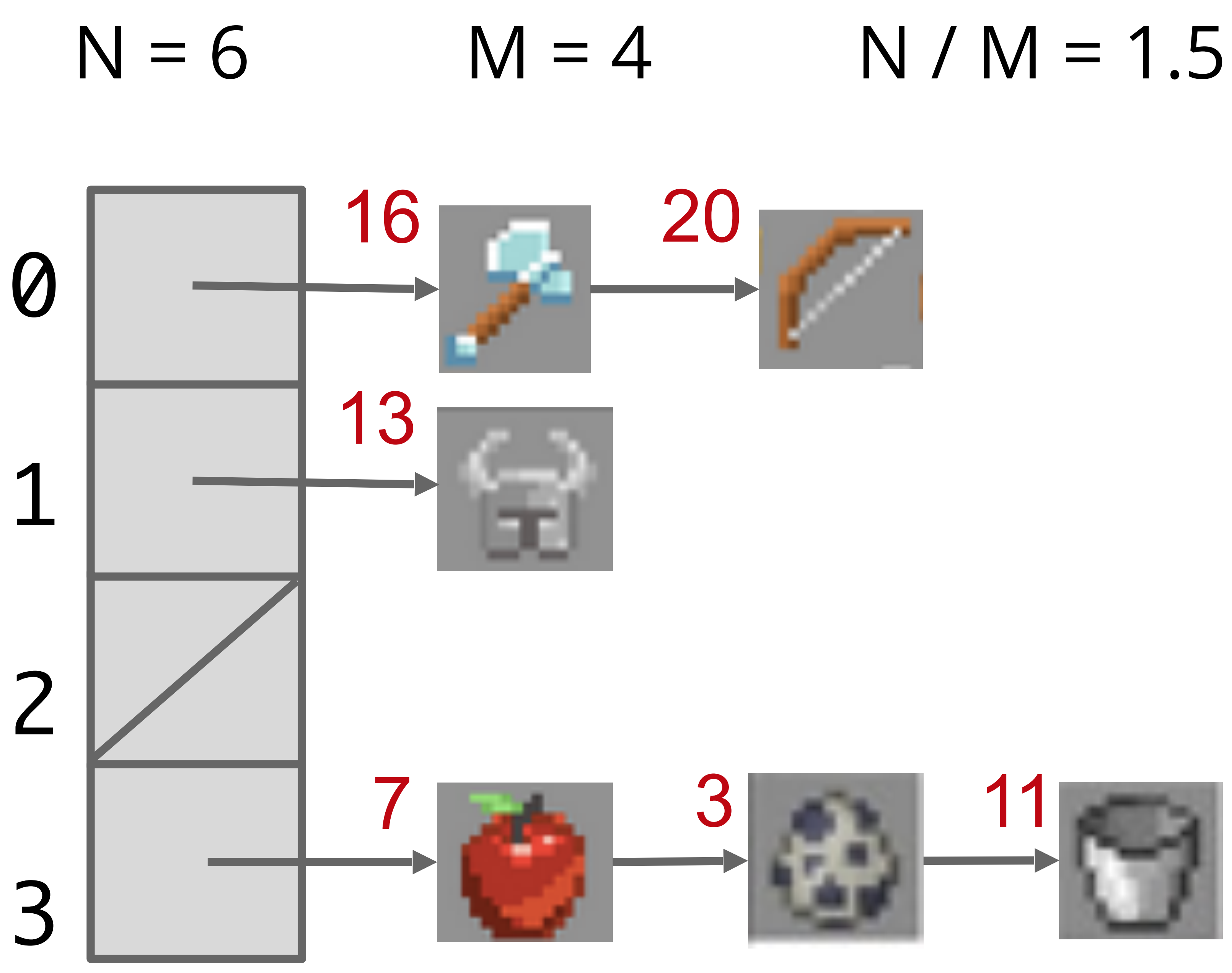
When N/M is ≥ 1.5, then double M.

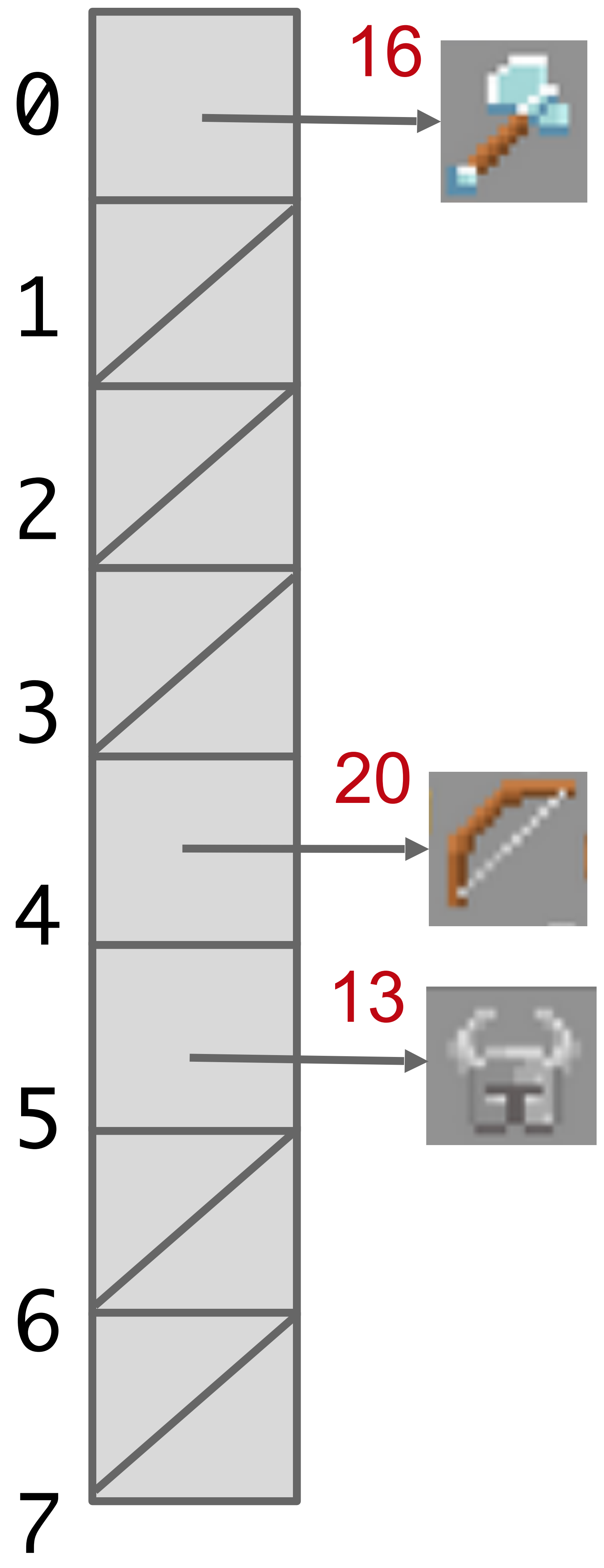




Worksheet answer: hash % 8 instead of hash % 4

When N/M is ≥ 1.5, then double M.

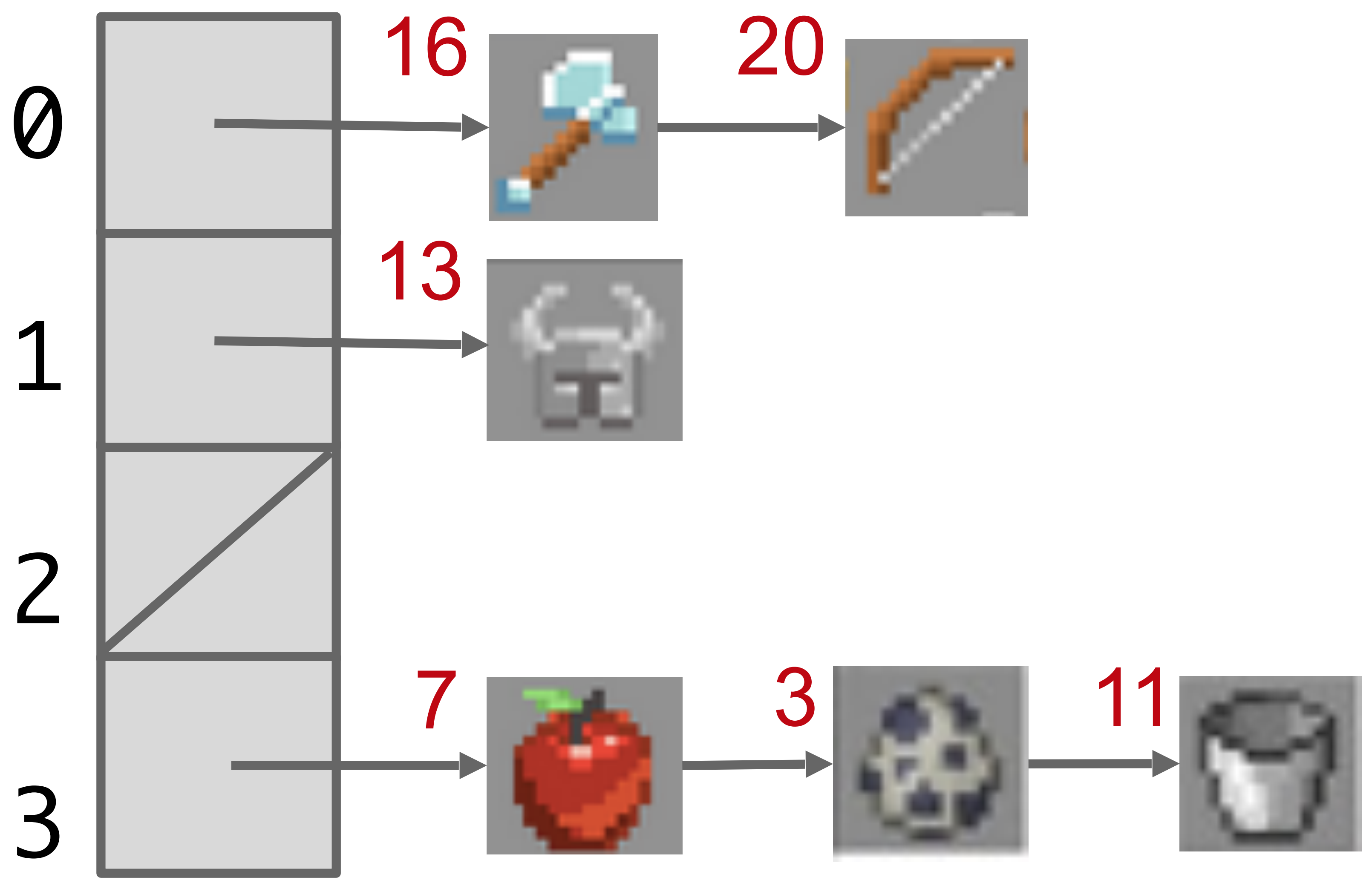


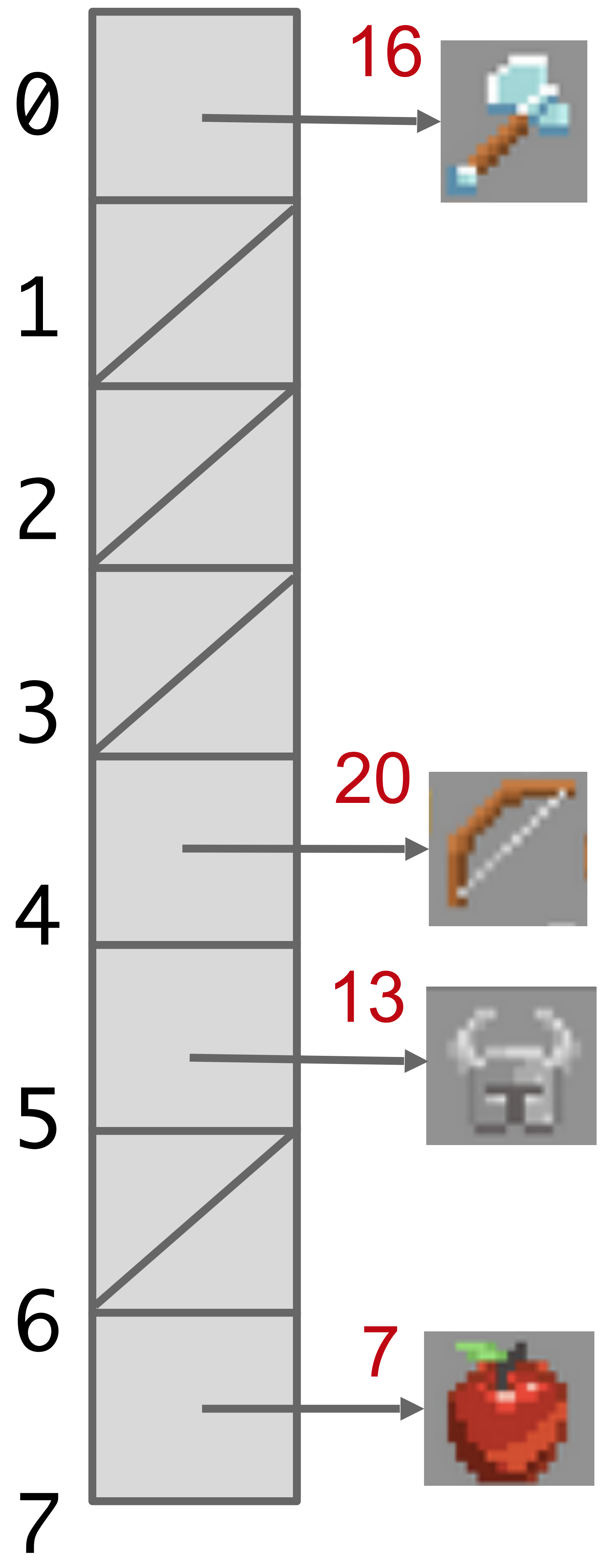


Worksheet answer: **hash % 8 instead of hash % 4**

When N/M is ≥ 1.5, then double M.

N = 6 M = 4 N / M = 1.5

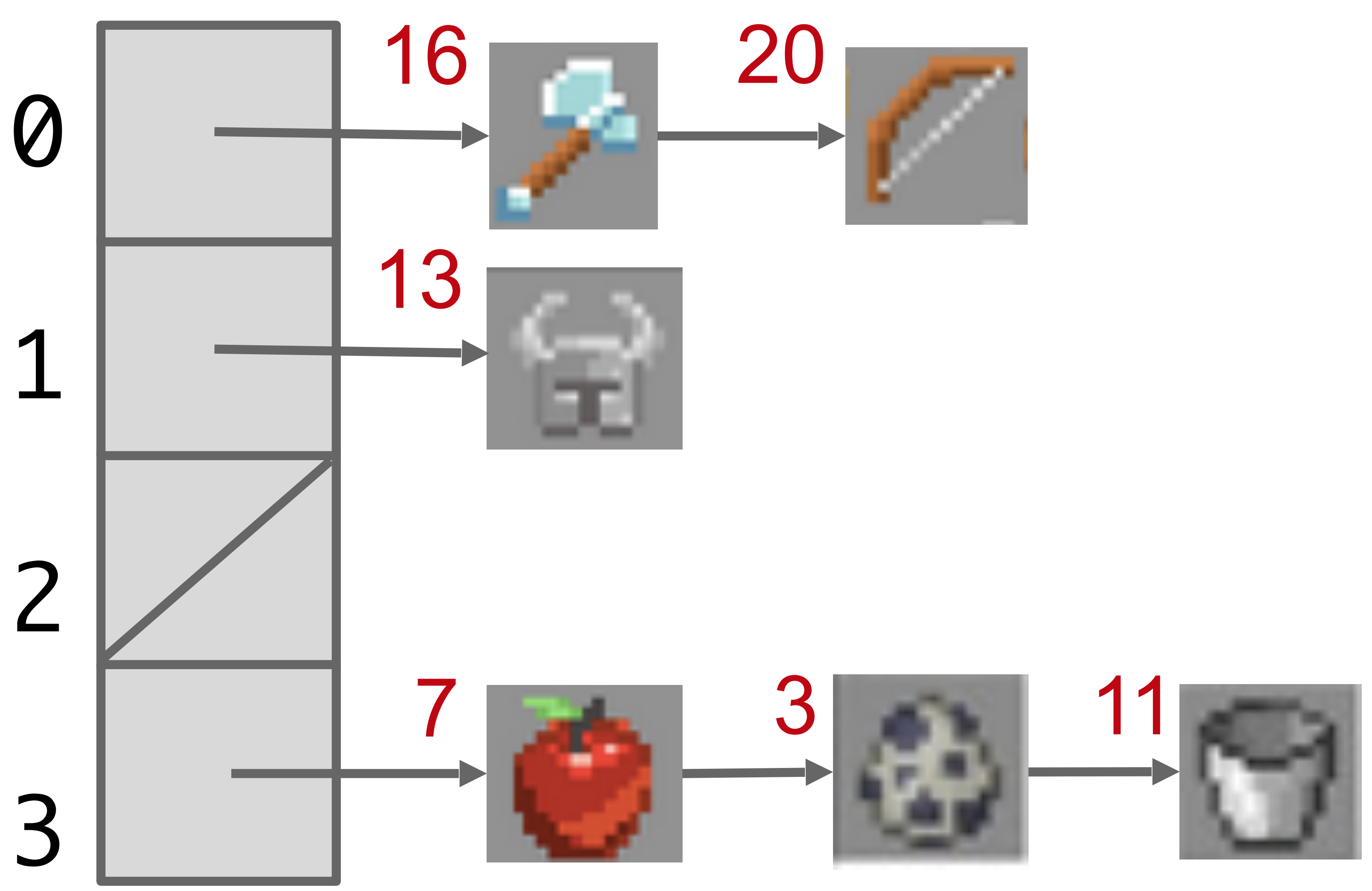


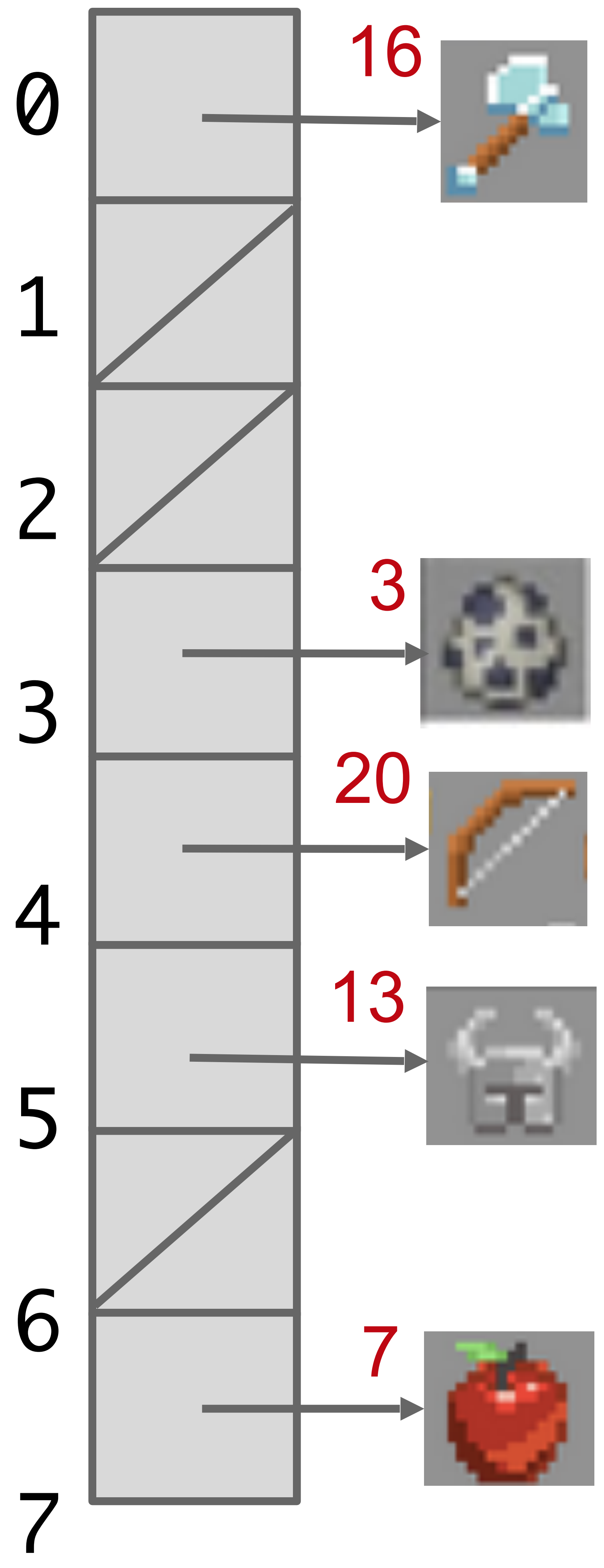


Worksheet answer: hash % 8 instead of hash % 4

When N/M is ≥ 1.5, then double M.

N = 6   M = 4   N / M = 1.5

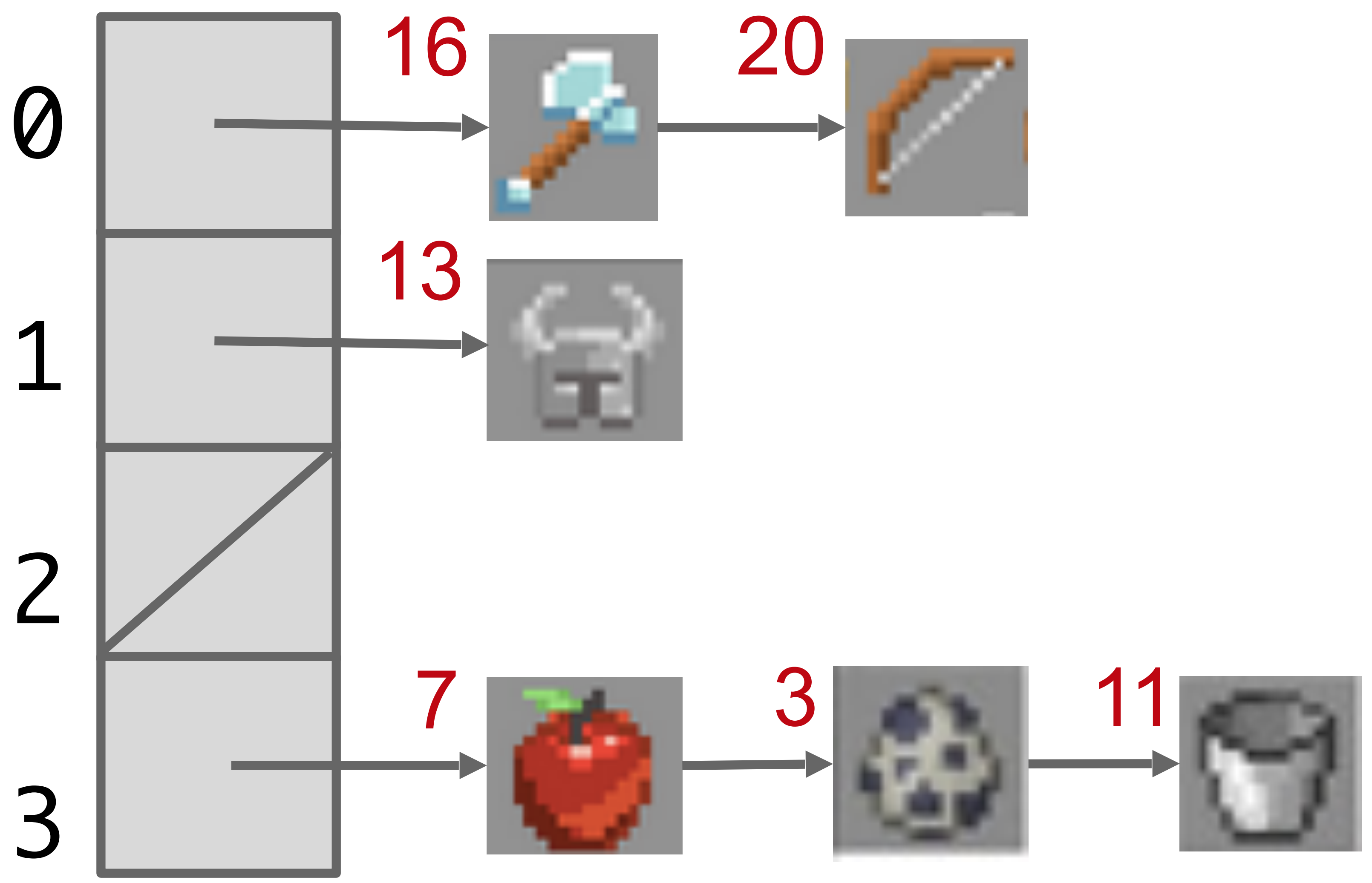


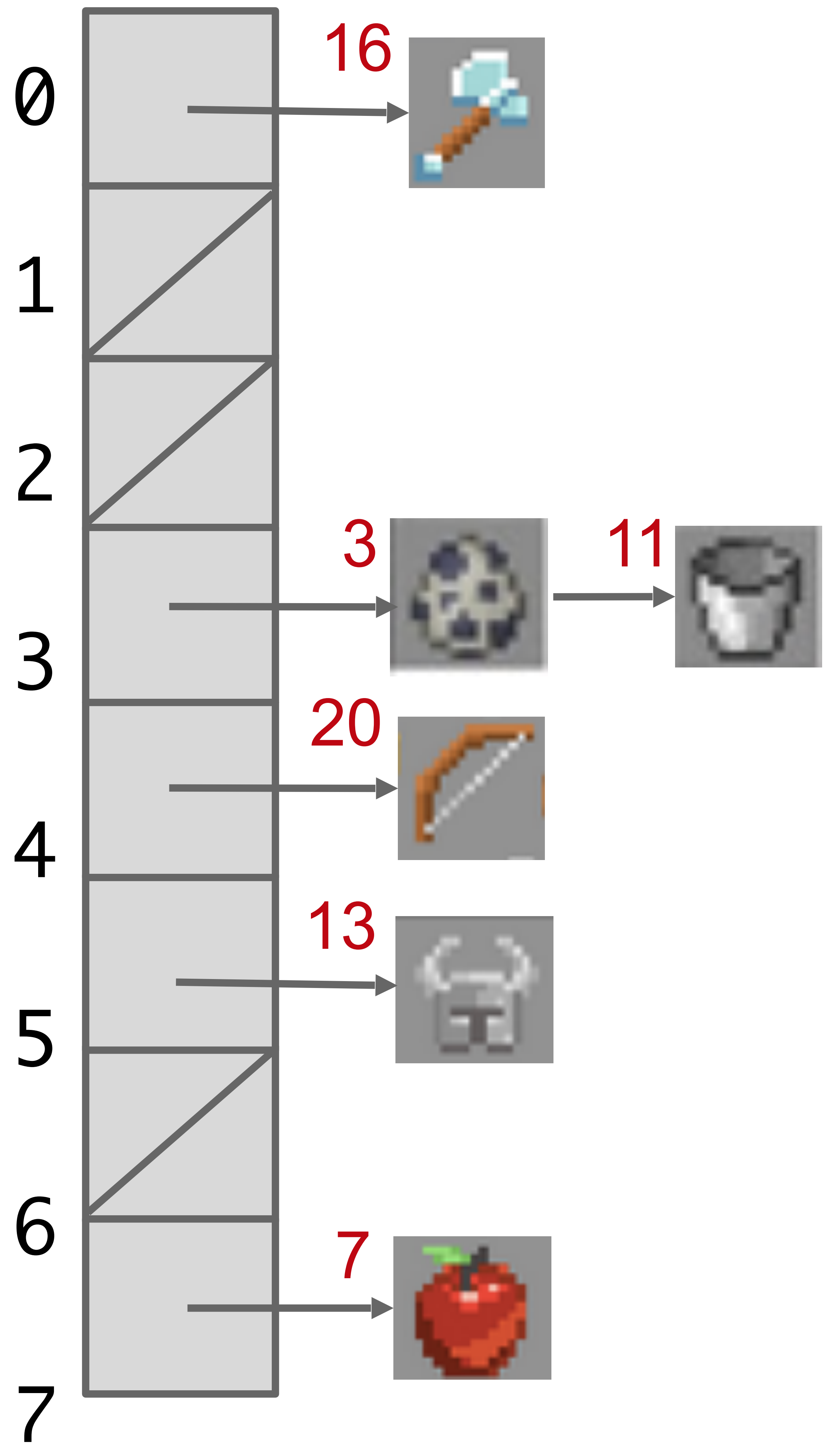


Worksheet answer: **hash % 8 instead of hash % 4**

When N/M is ≥ 1.5, then double M.

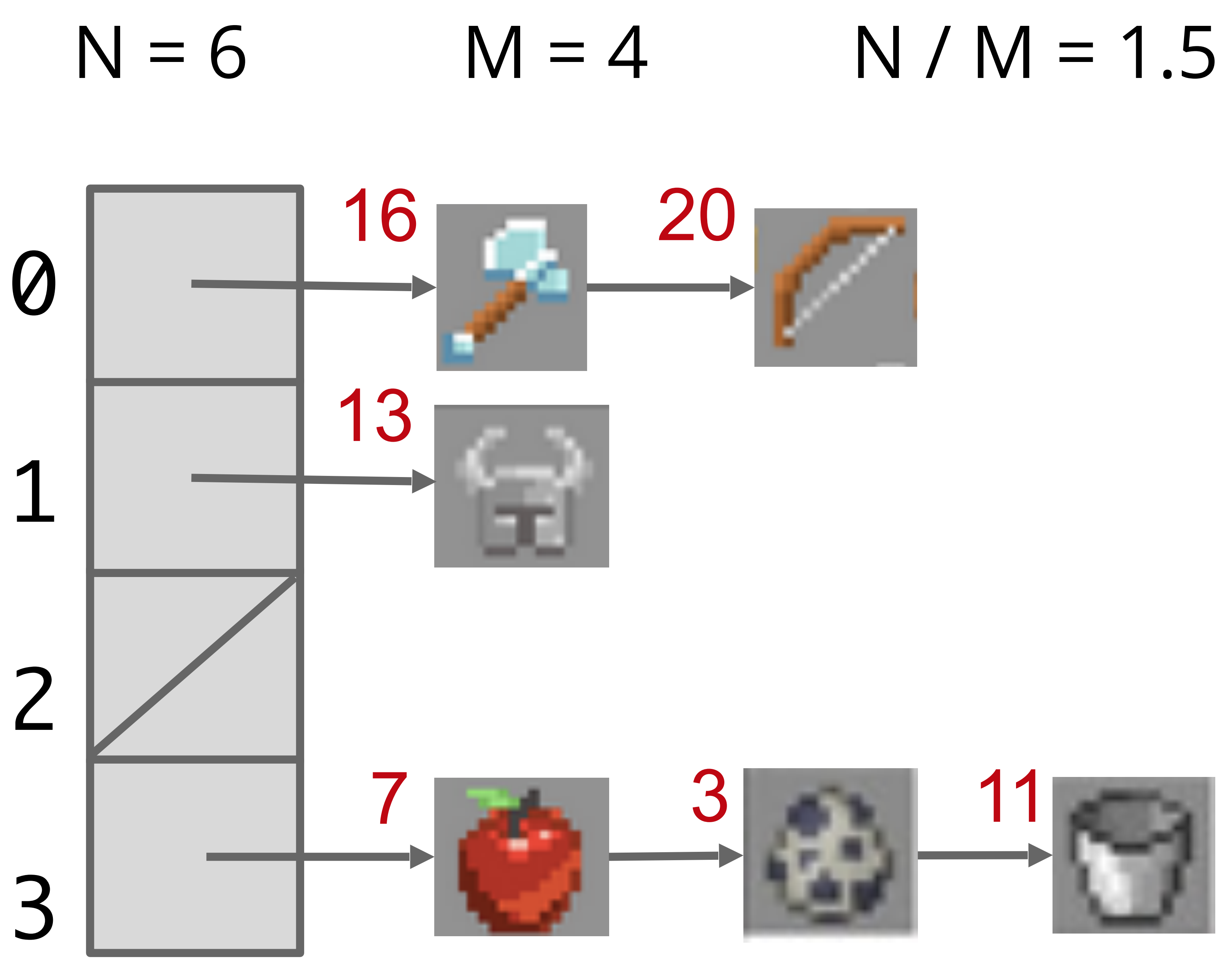
N = 6   M = 4   N / M = 1.5

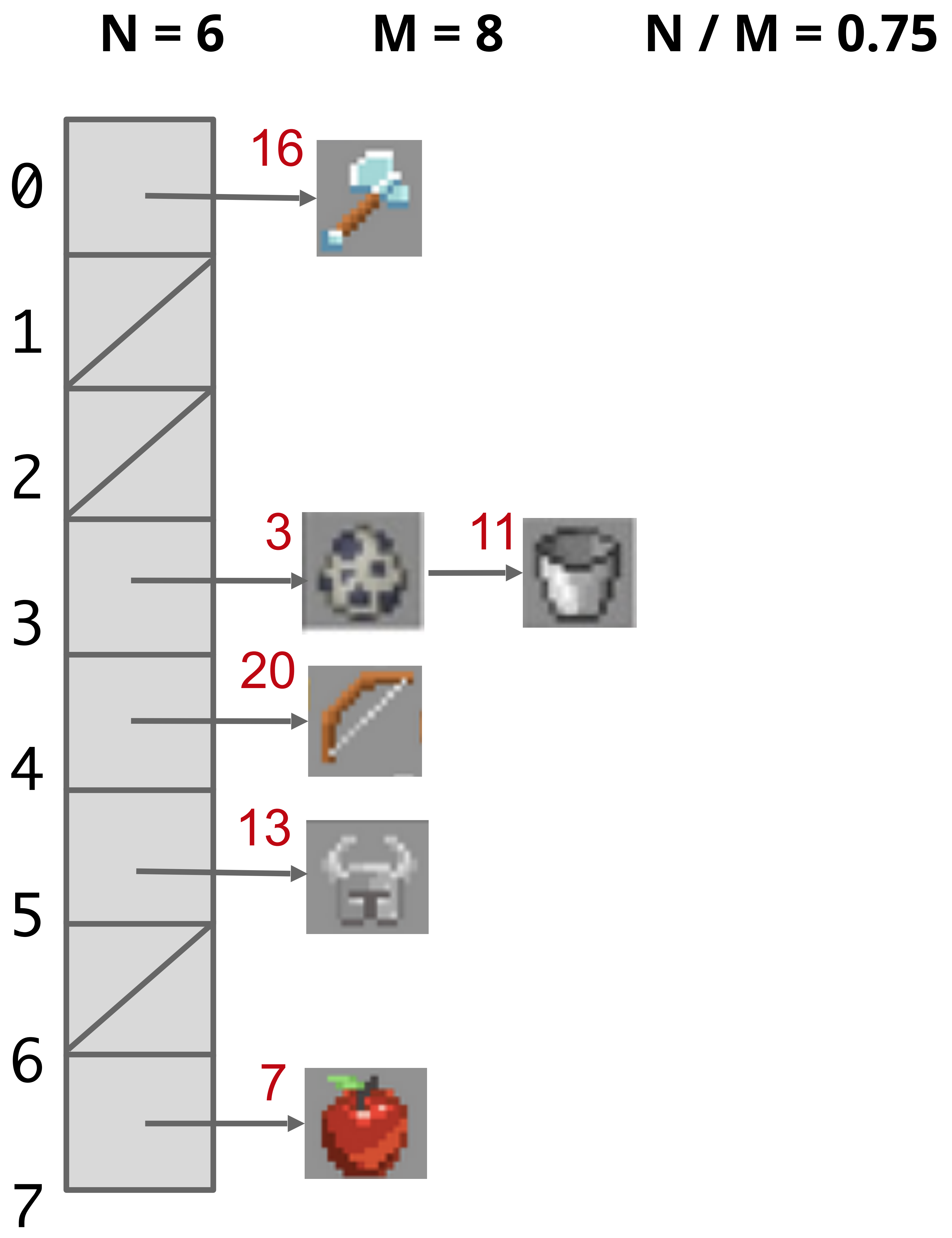




Worksheet answer: hash % 8 instead of hash % 4

When N/M is ≥ 1.5, then double M.





Lecture 20 wrap-up

- **Fill out the pre-pre enrollment survey by 10pm tonight!!!**
  - And the CS climate survey by tomorrow night!
- Exit ticket: https://forms.gle/mjsznbjFtonoRUCm6
- Checkpoint 2 grades will be released tonight. Regrades due by lecture next Thurs.
  - Please submit them **digitally** on Gradescope (I'll make a new assignment) instead of on paper.
- HW8: Hex-A-Pawn due next Tues, 11:59pm

Resources

- Hashtable history (it's really dark. More next lecture): https://cs.pomona.edu/classes/cs62/history/hashtables/
- Reading from textbook: Chapter 3.4 (Pages 458-477); https://algs4.cs.princeton.edu/34hash/
- Hashtable visualization: https://visualgo.net/en/hashtable
- Practice problems behind this slide
- Most of these slides were from my hashtable teaching demo from when I was applying to teaching jobs … :)

Problem 1

- Insert the keys E, A, S, Y, Q, U, E, S T, I, O, N in that order into an initially empty table of **m=5** lists, using separate chaining. Use the hash function **11*k%m** to transform the k-th letter of the English alphabet into a table index.

Problem 2

Try to write a function englishToInt that can convert English strings to integers by adding characters scaled by powers of 27.

Examples:

- a: 1
- z: 26
- aa: 28
- bee: 1598
- cat: 2234
- dog: 3328
- potato: 237,949,071

Answer 1

- Insert the keys E, A, S, Y, Q, U, E, S, T, I, O, N in that order into an initially empty table of m=5 lists, using separate chaining. Use the hash function 11*k%m to transform the k-th letter of the English alphabet into a table index.
- 0 -> O->T->Y->E
- 1 -> U -> a
- 2 -> Q
- 3 -> null
- 4 -> N -> I -> S

Answer 2

```
/** Converts ith character of String to a letter number.
    * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */
public static int letterNum(String s, int i) {
        int ithChar = s.charAt(i);
        if ((ithChar < 'a') || (ithChar > 'z'))
       { throw new IllegalArgumentException(); }
        return ithChar - 'a' + 1;
}

public static int englishToInt(String s) {
        int intRep = 0;
        for (int i = 0; i < s.length(); i += 1) {
       intRep = intRep * 27;
       intRep = intRep + letterNum(s, i);
        }
        return intRep;
}
```