Hash Table

CS143: lecture 15

Konstantinos Ameranis, July 28

Hashing

Turning any value into an integer

Hashing

Turning any value into an integer

- A *hash function* maps a key to an integer *deterministically*:

Hashing

Turning any value into an integer

- A *hash function* maps a key to an integer *deterministically*:
    - I.e. the same key is always turned into the same integer

Hashing

Turning any value into an integer

- A *hash function* maps a key to an integer *deterministically*:
  - I.e. the same key is always turned into the same integer
  - Hash functions should run in $O(1)$ time

Hashing

Turning any value into an integer

- A *hash function* maps a key to an integer *deterministically*:
  - I.e. the same key is always turned into the same integer
  - Hash functions should run in $O(1)$ time
- There are good/bad choices for hash functions

Hashing

Example: 2-letter word dictionary

Hashing

Example: 2-letter word dictionary

- Map 2-letter words to definitions:

Hashing

Example: 2-letter word dictionary

- Map 2-letter words to definitions:
  - Key: 2-letter words (string)

Hashing

Example: 2-letter word dictionary

- Map 2-letter words to definitions:
    - Key: 2-letter words (string)
    - Value: definitions (string)

Hashing

Example: 2-letter word dictionary

- Map 2-letter words to definitions:
  - Key: 2-letter words (string)
  - Value: definitions (string)

```
ah: used to express delight, relief, regret, or contempt
as: to the same degree or amount
at: used as a function word to indicate presence or occurrence in, on, or near
do: to bring to pass
go: to move on a course
ha: used especially to express surprise, joy, or triumph
he: that male one who is neither speaker nor hearer
hi: used especially as a greeting
...
```

Hashing

Example: 2-letter word dictionary

- Map 2-letter words to definitions:
  - Key: 2-letter words (string)
  - Value: definitions (string)

```
ah: used to express delight, relief, regret, or contempt
as: to the same degree or amount
at: used as a function word to indicate presence or occurrence in, on, or near
do: to bring to pass
go: to move on a course
ha: used especially to express surprise, joy, or triumph
he: that male one who is neither speaker nor hearer
hi: used especially as a greeting
...
```

- What hash function could we use to map keys to ints?

Hashing

Example: 2-letter word dictionary

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
    - 26 * 26 = 676

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?
  - Idea: map a-z: 0-25

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?
  - Idea: map a-z: 0-25
  - then, first letter's number * 26 + second letter's number

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?
  - Idea: map a-z: 0-25
  - then, first letter's number * 26 + second letter's number

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?
  - Idea: map a-z: 0-25
  - then, first letter's number * 26 + second letter's number

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

  - $hash(\alpha\beta) = 26\alpha + \beta$

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?
  - Idea: map a-z: 0-25
  - then, first letter's number * 26 + second letter's number

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

  - $hash(\alpha\beta) = 26\alpha + \beta$
  - $hash(\text{go}) = 26 \cdot 6 + 14 = 170$

Hashing

Example: 2-letter word dictionary

Hashing

Example: 2-letter word dictionary

- Example!

Hashing

Problem

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare's works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare's works	27

- $26^{27} = 160059109085386090080713531498405298176$

Hashing

Problem

Hashing

Problem

- $26^{27} = 160059109085386090080713531498405298176$

Hashing

Problem

- $26^{27} = 160059109085386090080713531498405298176$
- Too big for an array!

Hashing

Problem

- $26^{27} = 160059109085386090080713531498405298176$
- Too big for an array!
- Also, English has ~700,000 words; we only need a tiny fraction of these.

Hashing

Problem

- $26^{27} = 160059109085386090080713531498405298176$
- Too big for an array!
- Also, English has ~700,000 words; we only need a tiny fraction of these.
- Solution: Compress

Hashing

Compression

Hashing

Compression

- Generally, hash functions do not care about its output range.

Hashing
Compression

- Generally, hash functions do not care about its output range.
- We use a *compression function* to put the integer in the reasonable range [0,size)

Hashing

Compression

- Generally, hash functions do not care about its output range.
- We use a *compression function* to put the integer in the reasonable range [0,size)
- Common choice: modulus

Hashing

Compression

- Generally, hash functions do not care about its output range.
- We use a *compression function* to put the integer in the reasonable range [0,size)
- Common choice: modulus
  - a % b calculates the remainder of a divided by b

Hashing

Compression

- Generally, hash functions do not care about its output range.
- We use a *compression function* to put the integer in the reasonable range [0,size)
- Common choice: modulus
  - a % b calculates the remainder of a divided by b
  - a % b always returns an int in the range [0, b)

Hashing

Compression example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	
2	
3	
4	
5	
6	
7	7
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	
2	
3	
4	
5	
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	41
2	
3	
4	
5	
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35
-

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35
- What if we try to insert 75?

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35
- What if we try to insert 75?

0		
1	41	
2		
3		
4		
5	35	**75**
6		
7	7	
8	18	
9		

Hashing

Collision

Hashing

Collision

- Two different keys sometimes end up in the same slot

Hashing

Collision

- Two different keys sometimes end up in the same slot
  - This is called a collision

Hashing

Collision

- Two different keys sometimes end up in the same slot
  - This is called a collision
- Collision has to happen if we have smaller array than the range of hash function

Hashing

Collision

- Two different keys sometimes end up in the same slot
  - This is called a collision
- Collision has to happen if we have smaller array than the range of hash function
  - Hash function could produce the same integer for two different keys

Hashing
Collision

- Two different keys sometimes end up in the same slot
  - This is called a collision
- Collision has to happen if we have smaller array than the range of hash function
  - Hash function could produce the same integer for two different keys
  - Compression merges different hashes together

Hashing

Collision

- Two different keys sometimes end up in the same slot
  - This is called a collision
- Collision has to happen if we have smaller array than the range of hash function
  - Hash function could produce the same integer for two different keys
  - Compression merges different hashes together
- All tables need to handle collision

Hashing

Handling Collision

Hashing

Handling Collision

1. Avoid collisions when possible:

Hashing

Handling Collision

1. Avoid collisions when possible:
    1. Pick a good hash function (e.g. `strlen` is a terrible hash function)

Hashing

Handling Collision

1. Avoid collisions when possible:
    1. Pick a good hash function (e.g. `strlen` is a terrible hash function)
    2. Pick a good table size

Hashing

Handling Collision

1. Avoid collisions when possible:
    1. Pick a good hash function (e.g. `strlen` is a terrible hash function)
    2. Pick a good table size
2. When they arise (inevitably):

Hashing

Handling Collision

1. Avoid collisions when possible:
    1. Pick a good hash function (e.g. `strlen` is a terrible hash function)
    2. Pick a good table size
2. When they arise (inevitably):
    1. Have a way to put collisions in a table.

Hashing

Picking a good hash function

Hashing

Picking a good hash function

- Minimize collision:

Hashing

Picking a good hash function

- Minimize collision:
    - What is the worst possible hash function?

Hashing

Picking a good hash function

- Minimize collision:
  - What is the worst possible hash function?
  - hash(k) = 1

Hashing

Picking a good hash function

- Minimize collision:
  - What is the worst possible hash function?
  - hash(k) = 1
  - What is the best possible hash function?

Hashing

Picking a good hash function

- Minimize collision:
  - What is the worst possible hash function?
  - hash(k) = 1
  - What is the best possible hash function?
  - Every input maps to a distinct output, $f(x) = f(y) \implies x = y$

Hashing

Picking a good hash function

- Minimize collision:
  - What is the worst possible hash function?
  - hash(k) = 1
  - What is the best possible hash function?
  - Every input maps to a distinct output, $f(x) = f(y) \implies x = y$
  - This is called *perfect* hashing. The two-letter hash function is a perfect hash function.

Hashing

Picking a good hash function (Example)

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
    - Use their birthdays
        - Month (Jan, Feb, Mar, ...)?

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays
    - Month (Jan, Feb, Mar, ...)?
    - Age (0, 1, 2, ..., 100)?

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays
    - Month (Jan, Feb, Mar, ...)?
    - Age (0, 1, 2, ..., 100)?
    - Day of month (1, 2, 3, ..., 31)?

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays
    - Month (Jan, Feb, Mar, ...)?
    - Age (0, 1, 2, ..., 100)?
    - Day of month (1, 2, 3, ..., 31)?
  - Use their first name

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays
    - Month (Jan, Feb, Mar, ...)?
    - Age (0, 1, 2, ..., 100)?
    - Day of month (1, 2, 3, ..., 31)?
  - Use their first name
  - Use their last name

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays
    - Month (Jan, Feb, Mar, ...)?
    - Age (0, 1, 2, ..., 100)?
    - Day of month (1, 2, 3, ..., 31)?
  - Use their first name
  - Use their last name
  - Use their student ID

Hashing

Picking a good hash function

Hashing

Picking a good hash function

- A good hash function should be:

Hashing

Picking a good hash function

- A good hash function should be:
  - fast

Hashing

Picking a good hash function

- A good hash function should be:
  - fast
  - collision with (extremely) low probability

Hashing

Picking a good hash function

- A good hash function should be:
  - fast
  - collision with (extremely) low probability
  - spreads out the keys

Hashing

Picking a good hash function

- A good hash function should be:
    - fast
    - collision with (extremely) low probability
    - spreads out the keys
- CS284: Cryptography

Hash Table

Recap

Hash Table

Recap

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers

Hash Table

Recap

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers
- We have a way to turn anything into an integer -- hash function

Hash Table

Recap

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers
- We have a way to turn anything into an integer -- hash function
- We have a way to force any integers into a reasonable range -- compression (usually modulus)

Hash Table

Recap

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers
- We have a way to turn anything into an integer -- hash function
- We have a way to force any integers into a reasonable range -- compression (usually modulus)
- We need to handle collisions:

Hash Table

Recap

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers
- We have a way to turn anything into an integer -- hash function
- We have a way to force any integers into a reasonable range -- compression (usually modulus)
- We need to handle collisions:
  - Collisions can be the result of the hash function

Hash Table

Recap

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers
- We have a way to turn anything into an integer -- hash function
- We have a way to force any integers into a reasonable range -- compression (usually modulus)
- We need to handle collisions:
  - Collisions can be the result of the hash function
  - ... of compression

Hash Table

Handling Collision

Hash Table

Handling Collision

- Two approaches:

Hash Table

Handling Collision

- Two approaches:
    1. Chaining

Hash Table

Handling Collision

- Two approaches:
    1. Chaining
    2. Probing

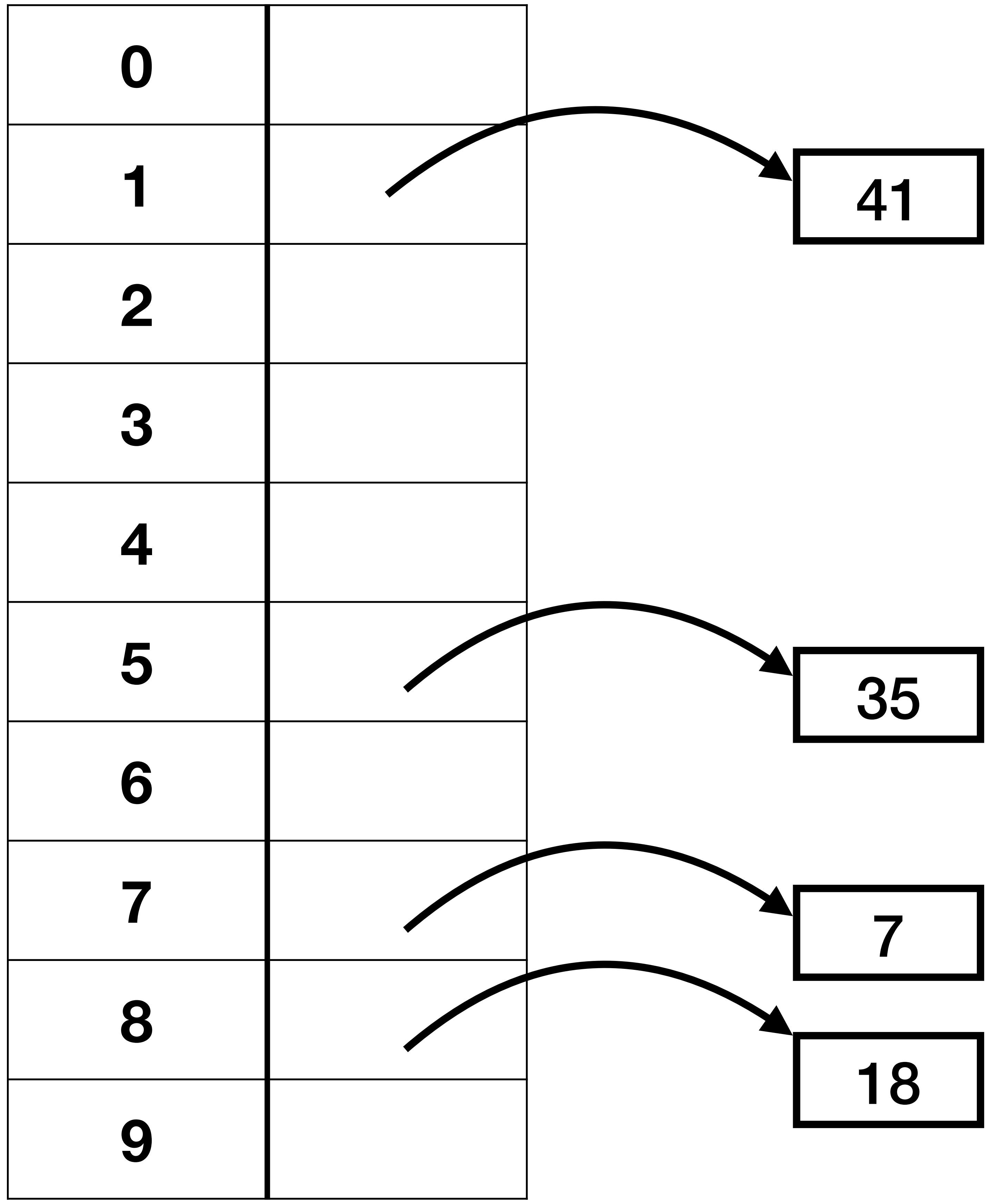
Chaining

- Each slot is a *list* of key-value pairs, called a *bucket*

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

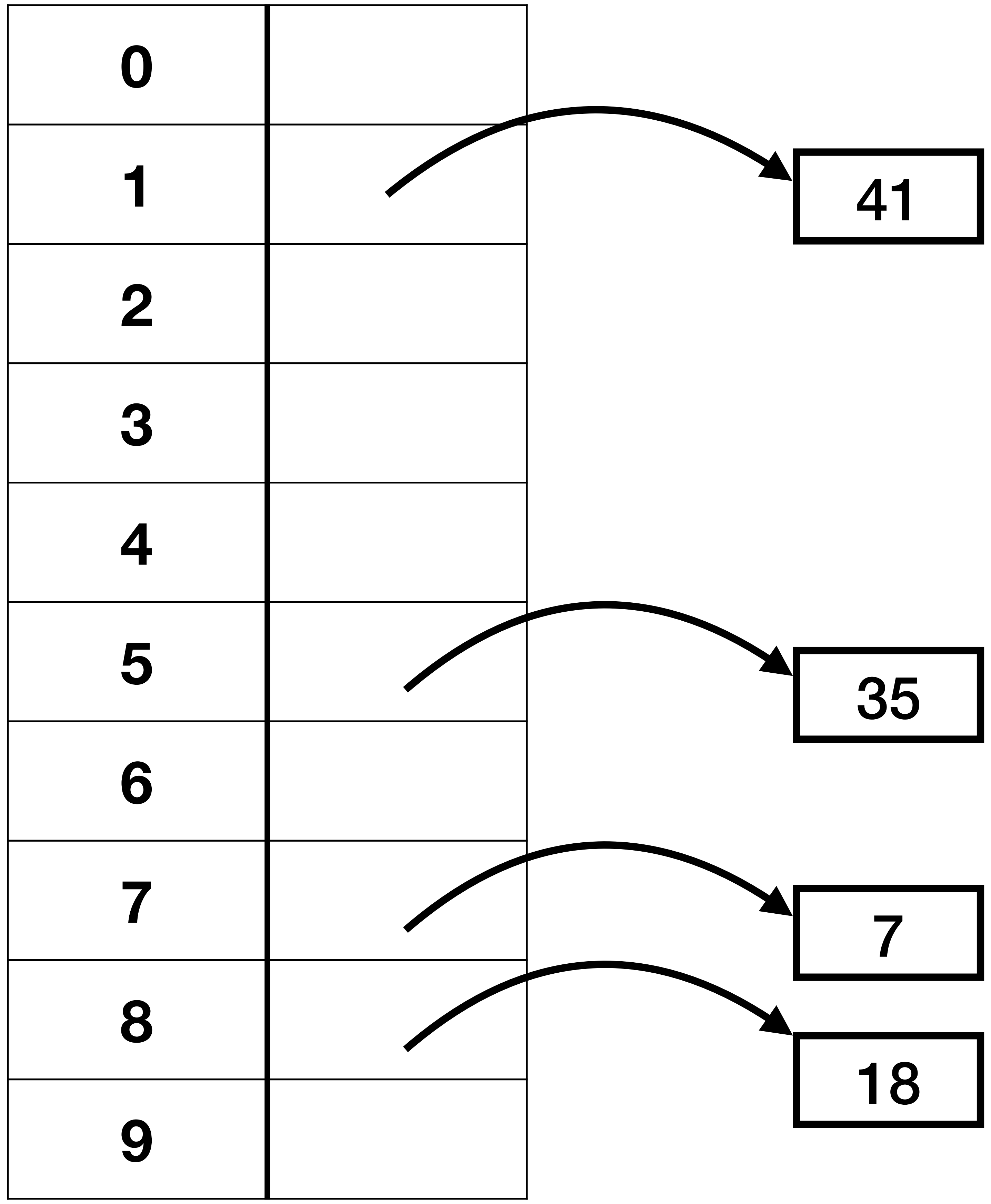
Chaining

- Each slot is a *list* of key-value pairs, called a *bucket*



Chaining

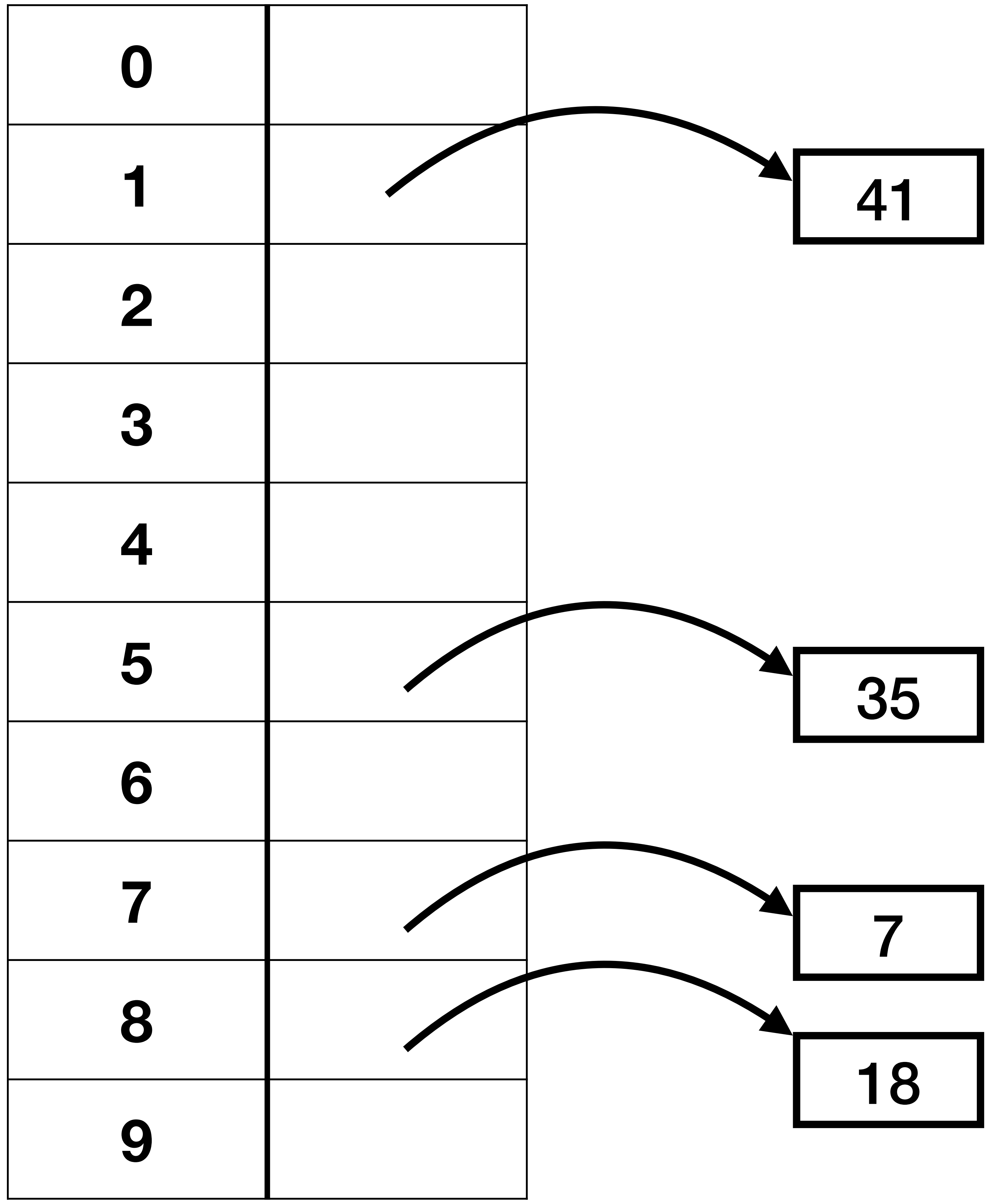
- Each slot is a *list* of key-value pairs, called a *bucket*



- You can use either list implementation

Chaining

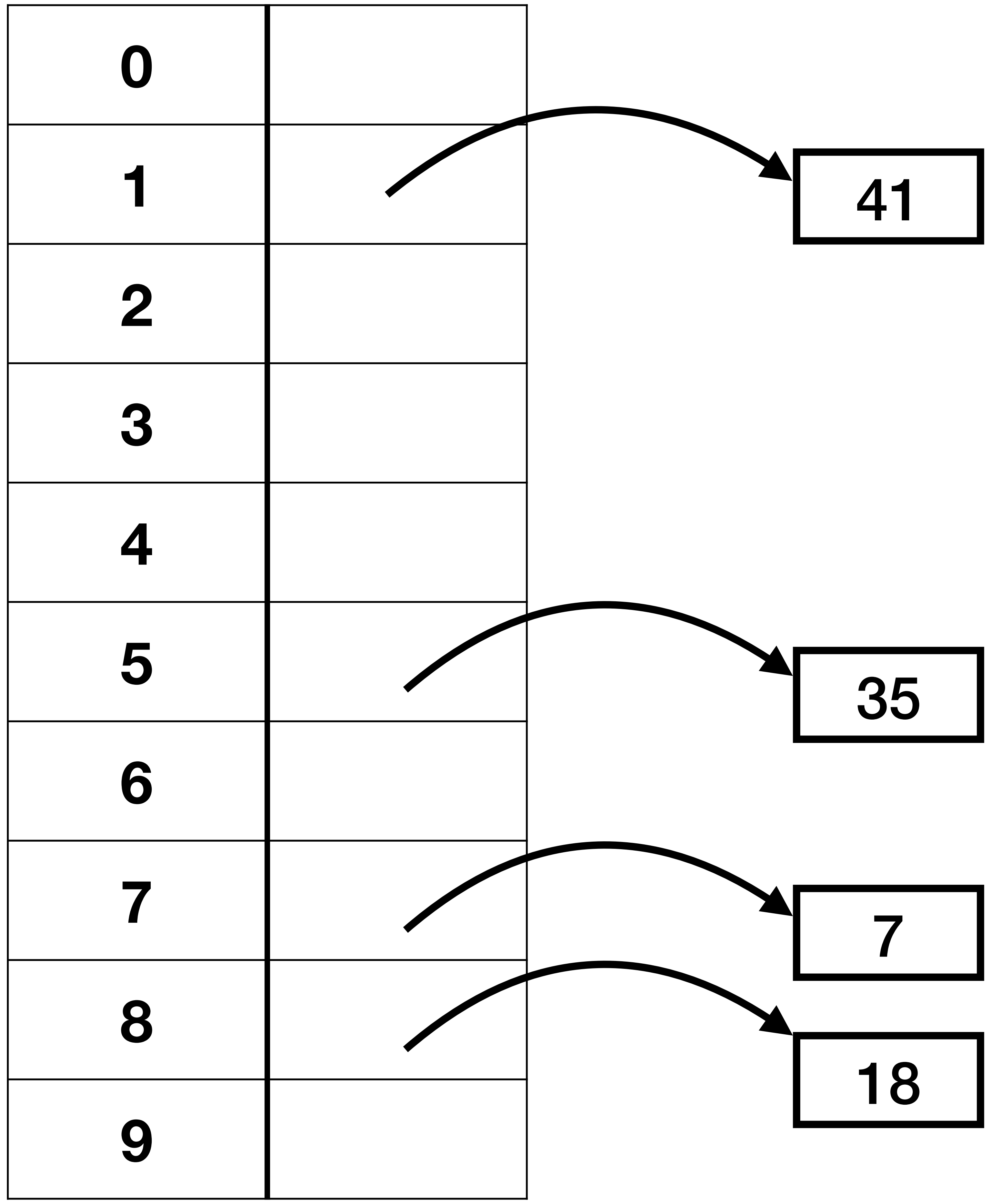
- Each slot is a *list* of key-value pairs, called a *bucket*



- You can use either list implementation
  - ...but there is an obvious choice

Chaining

- Each slot is a *list* of key-value pairs, called a *bucket*

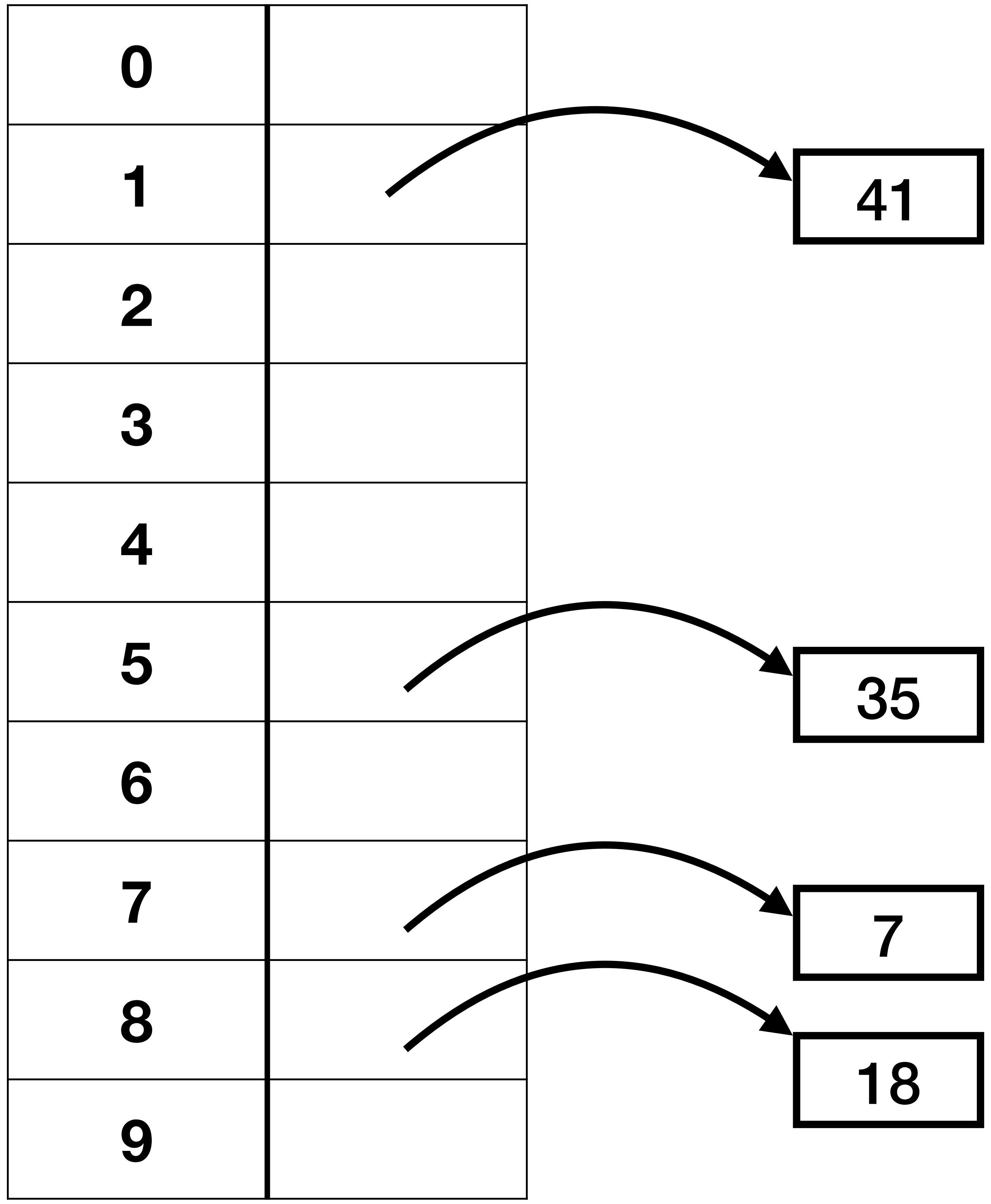


- You can use either list implementation
    - ...but there is an obvious choice
    - linked list, because of deletion

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

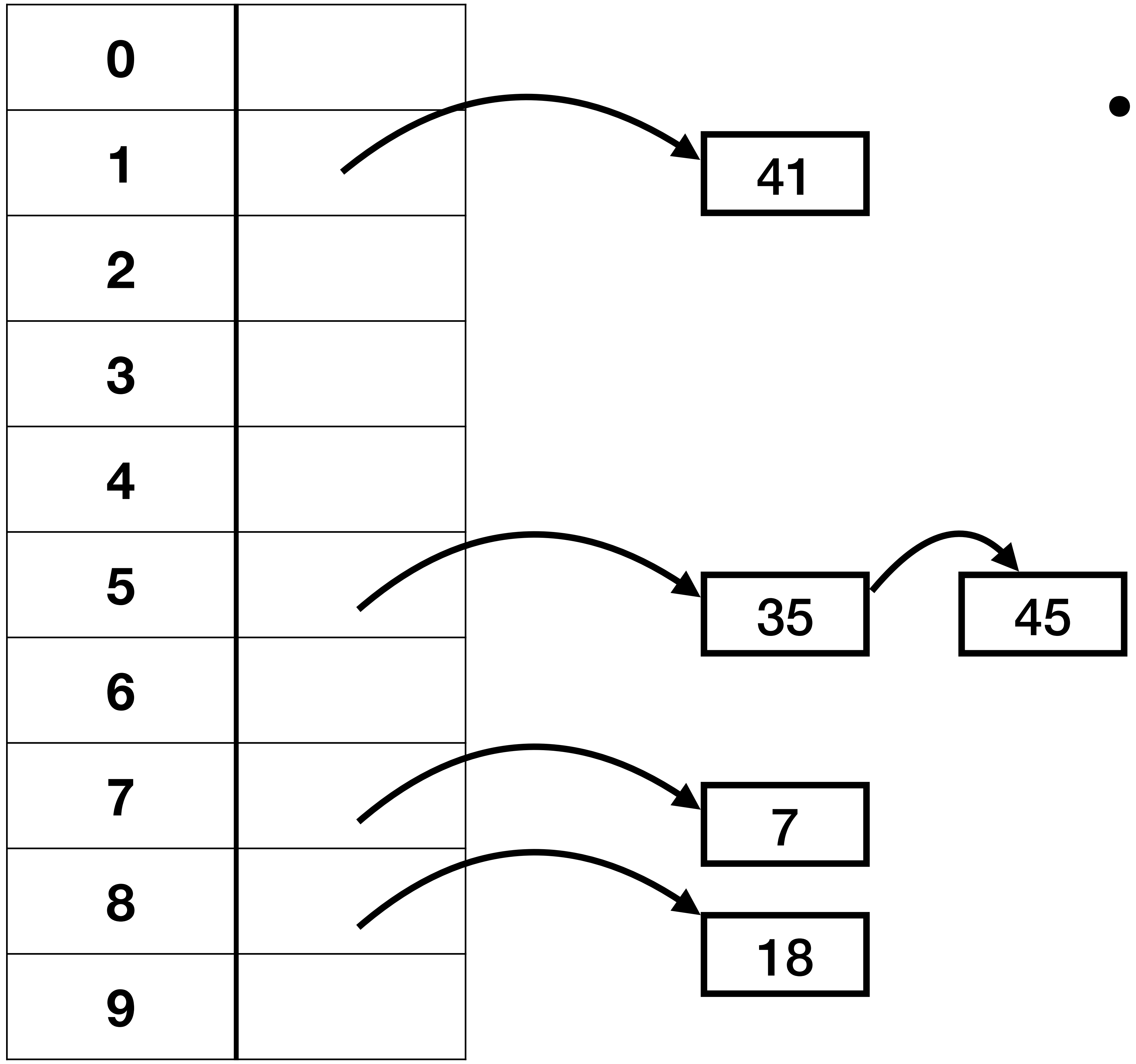


- Collisions will be prepended into the list

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

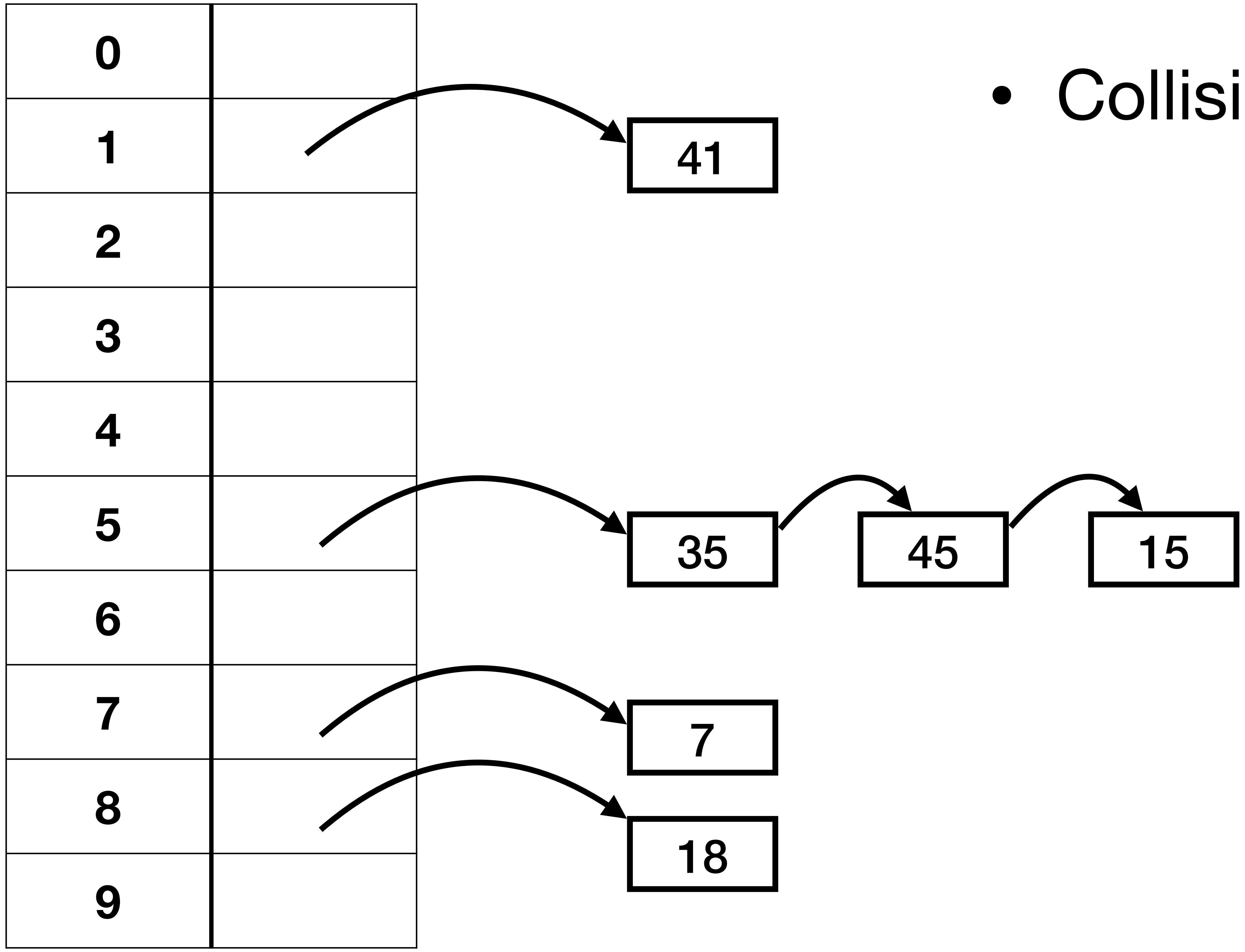


- Collisions will be prepended into the list

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

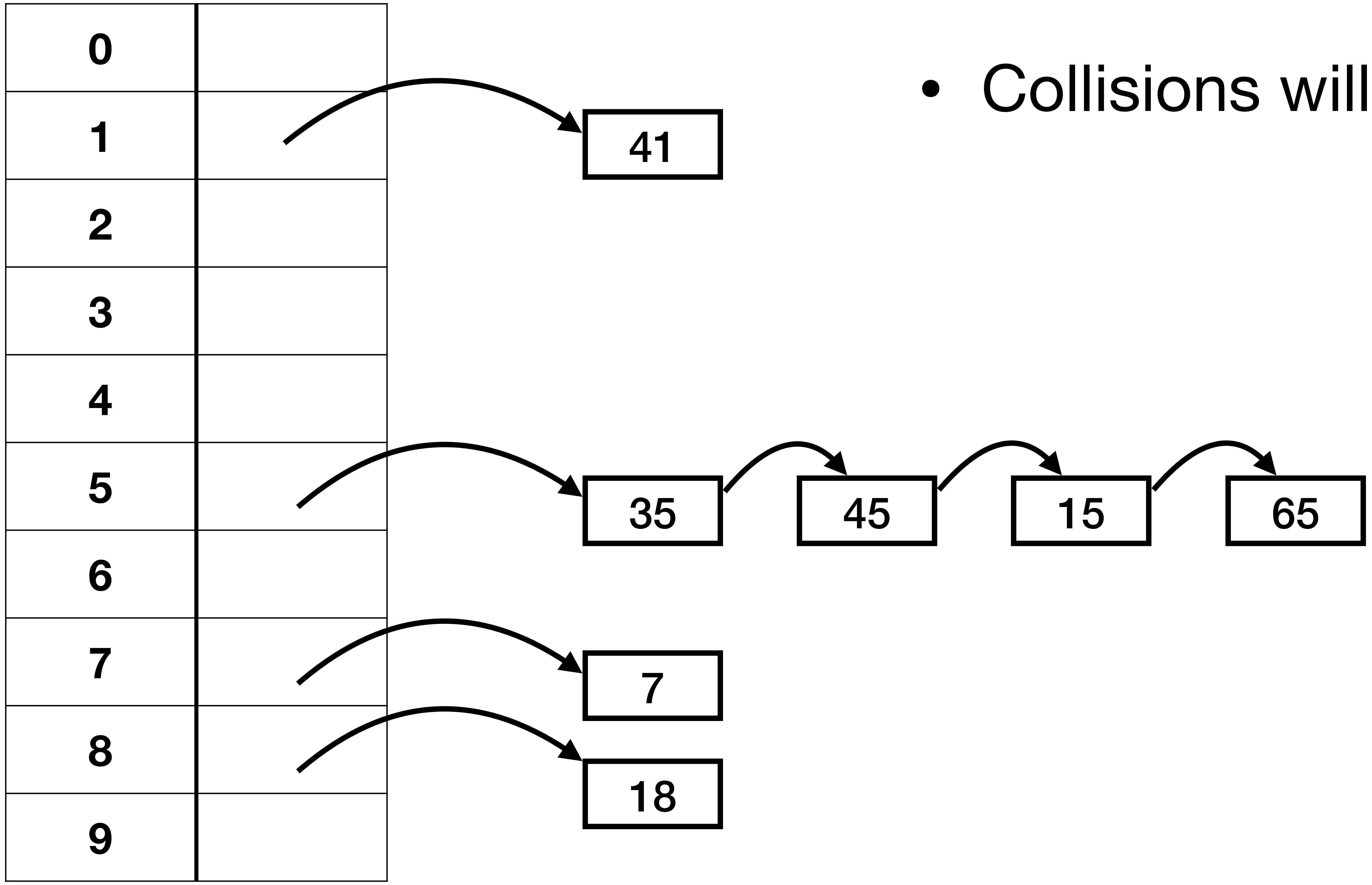


- Collisions will be prepended into the list

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

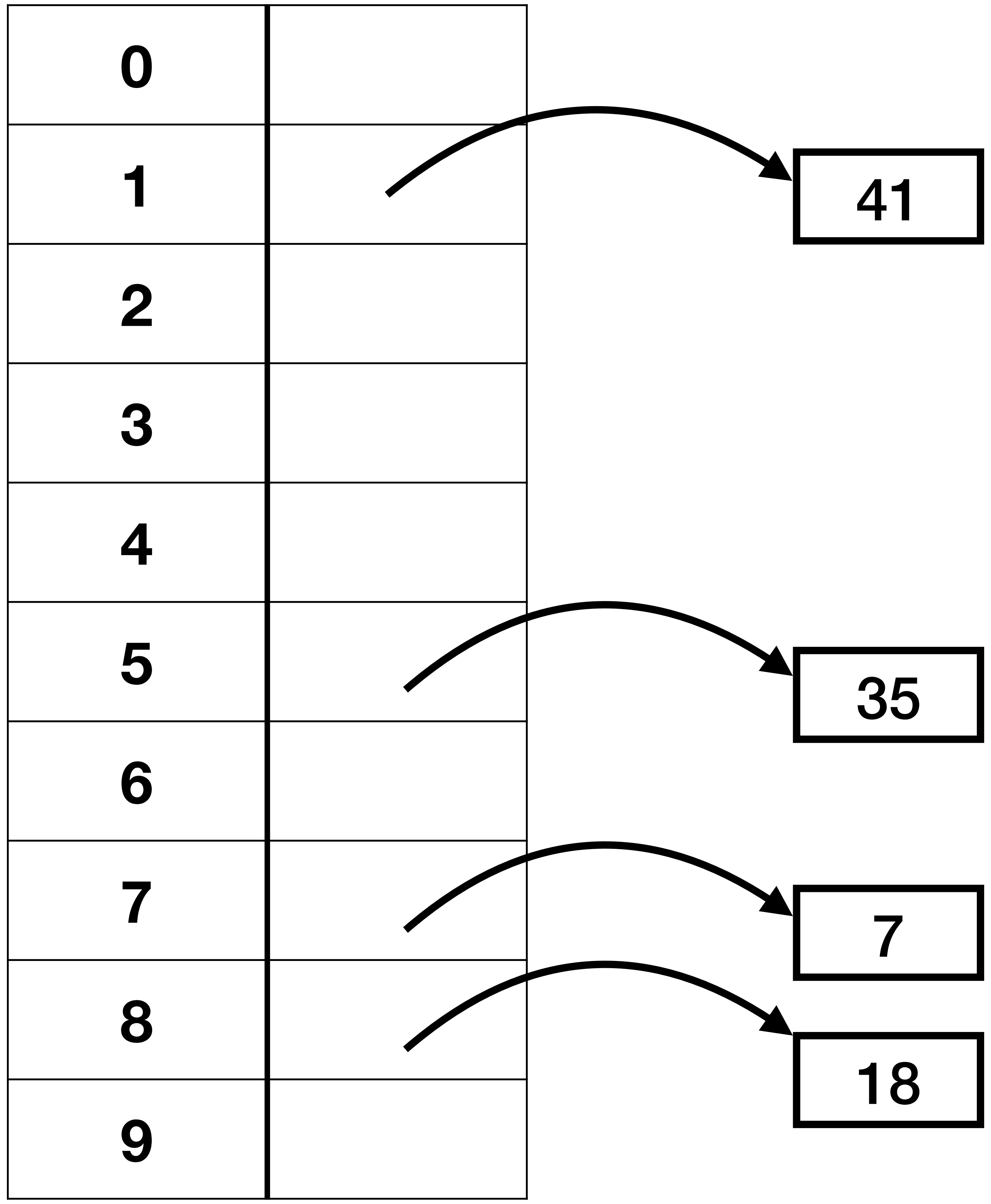


- Collisions will be prepended into the list

Chaining

Insert

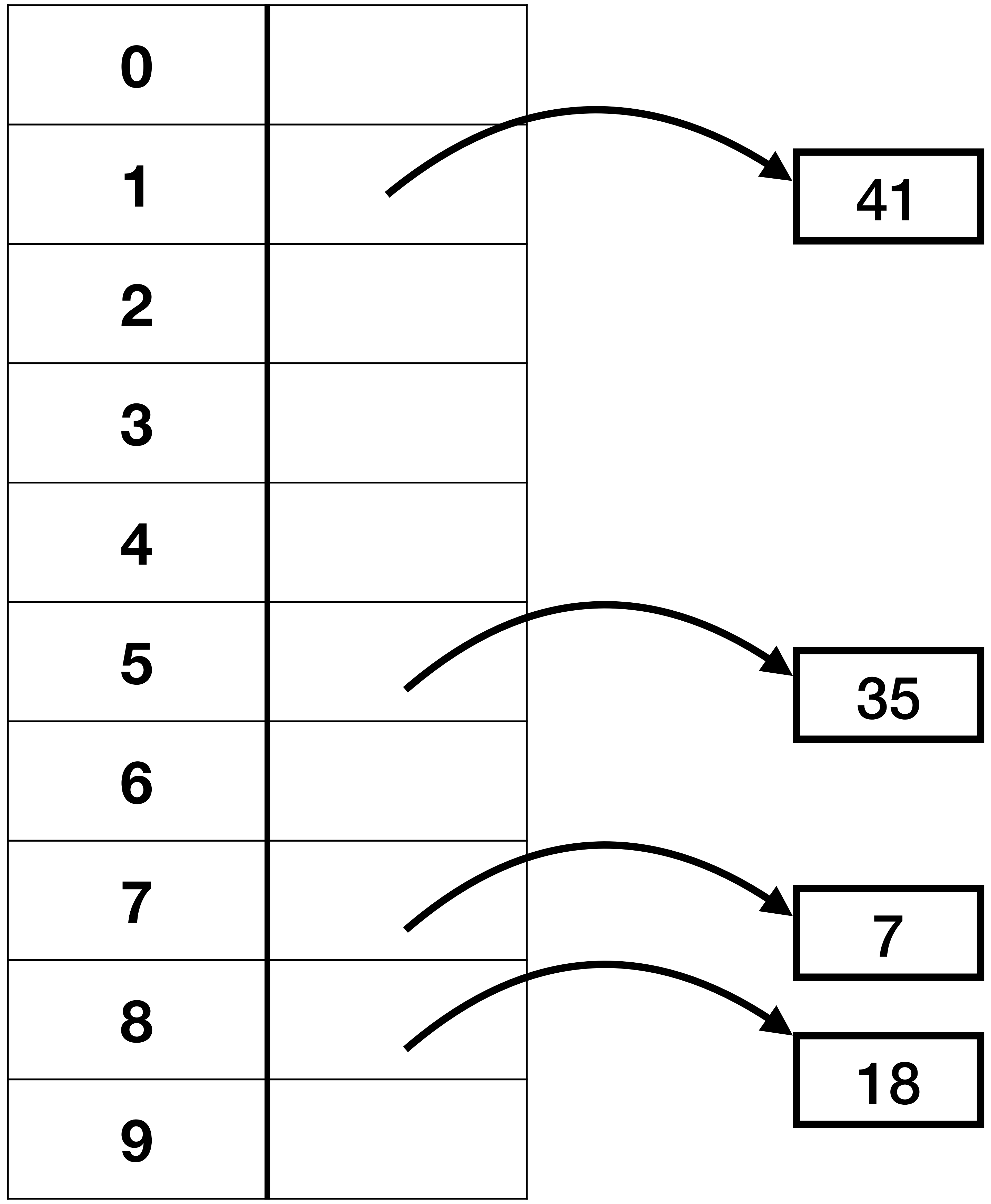
- Each slot is a *list* of key-value pairs, called a *bucket*



Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

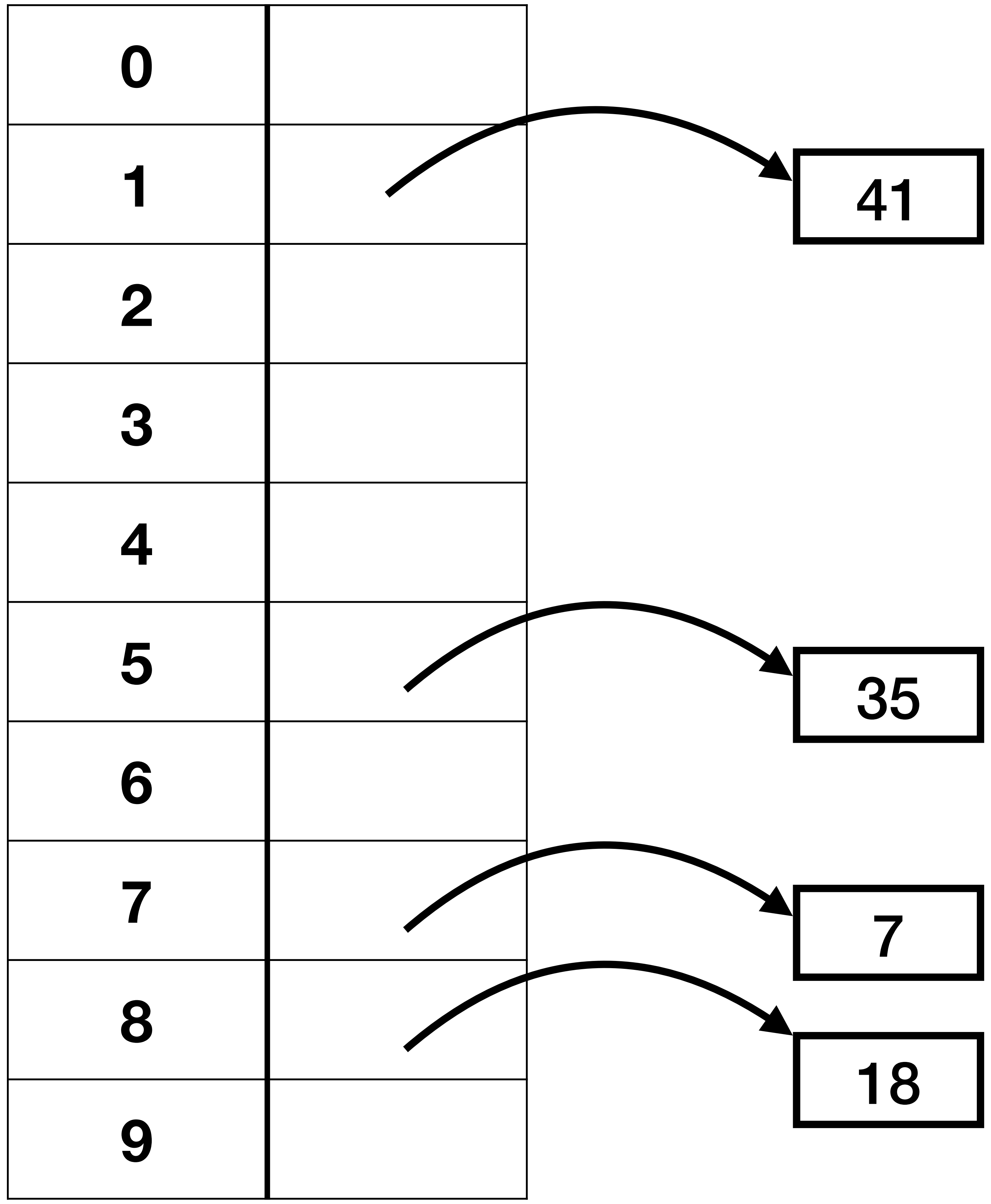



```
insert(table, key, value):
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

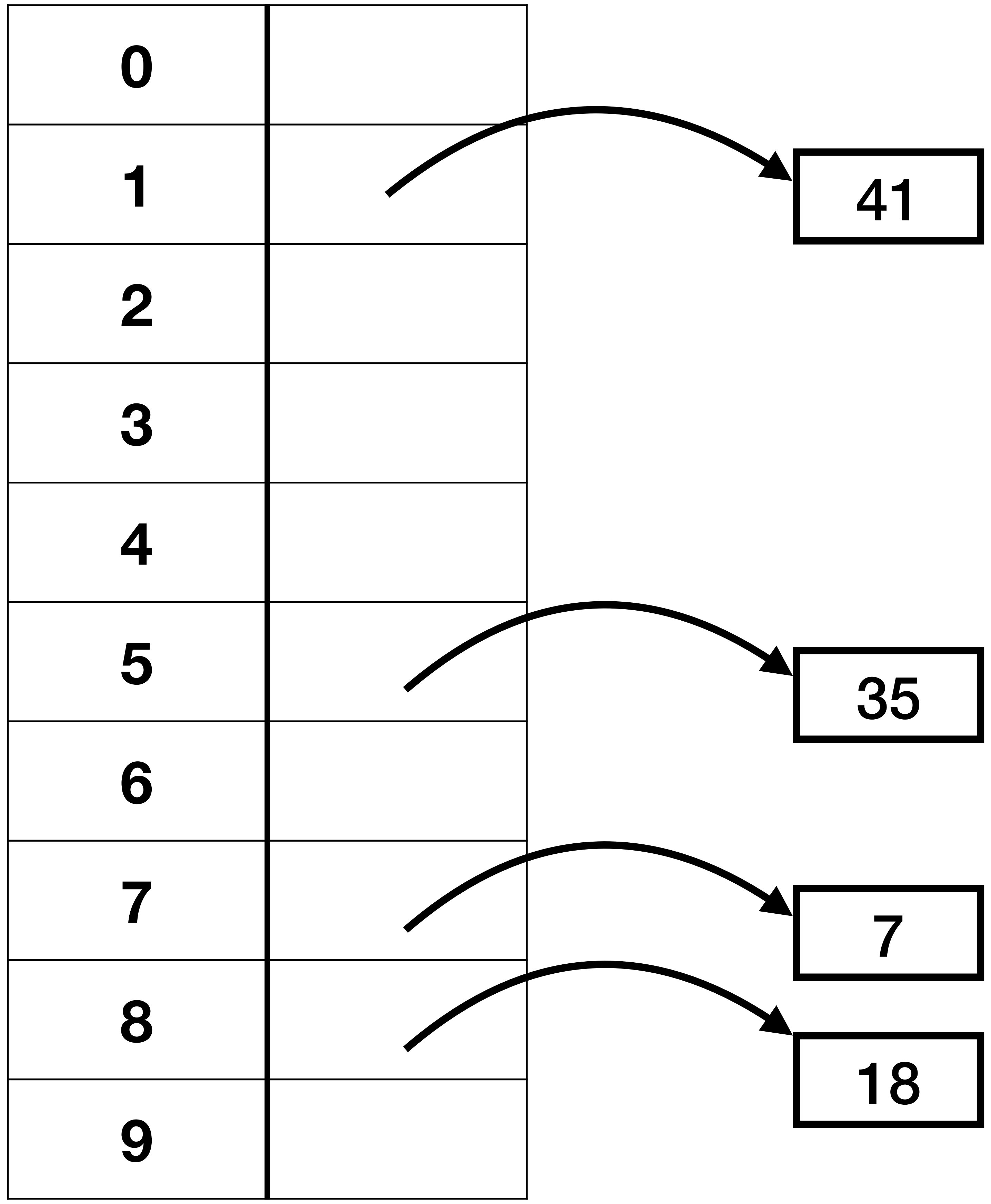



```
insert(table, key, value):
 bucket_idx = hash(key) % table->size
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

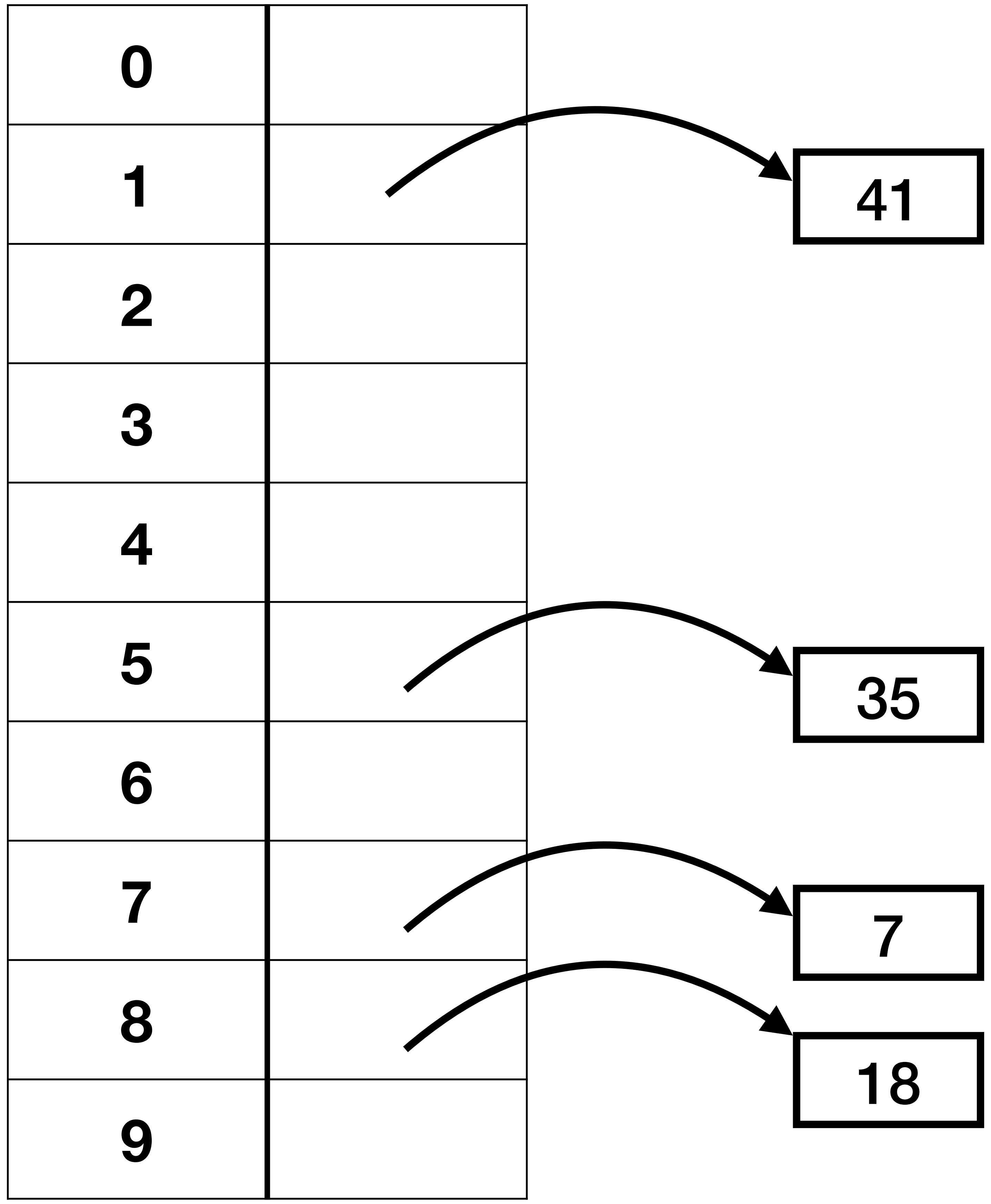



```
insert(table, key, value):
 bucket_idx = hash(key) % table->size
 if found key in table->buckets[bucket_idx]
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

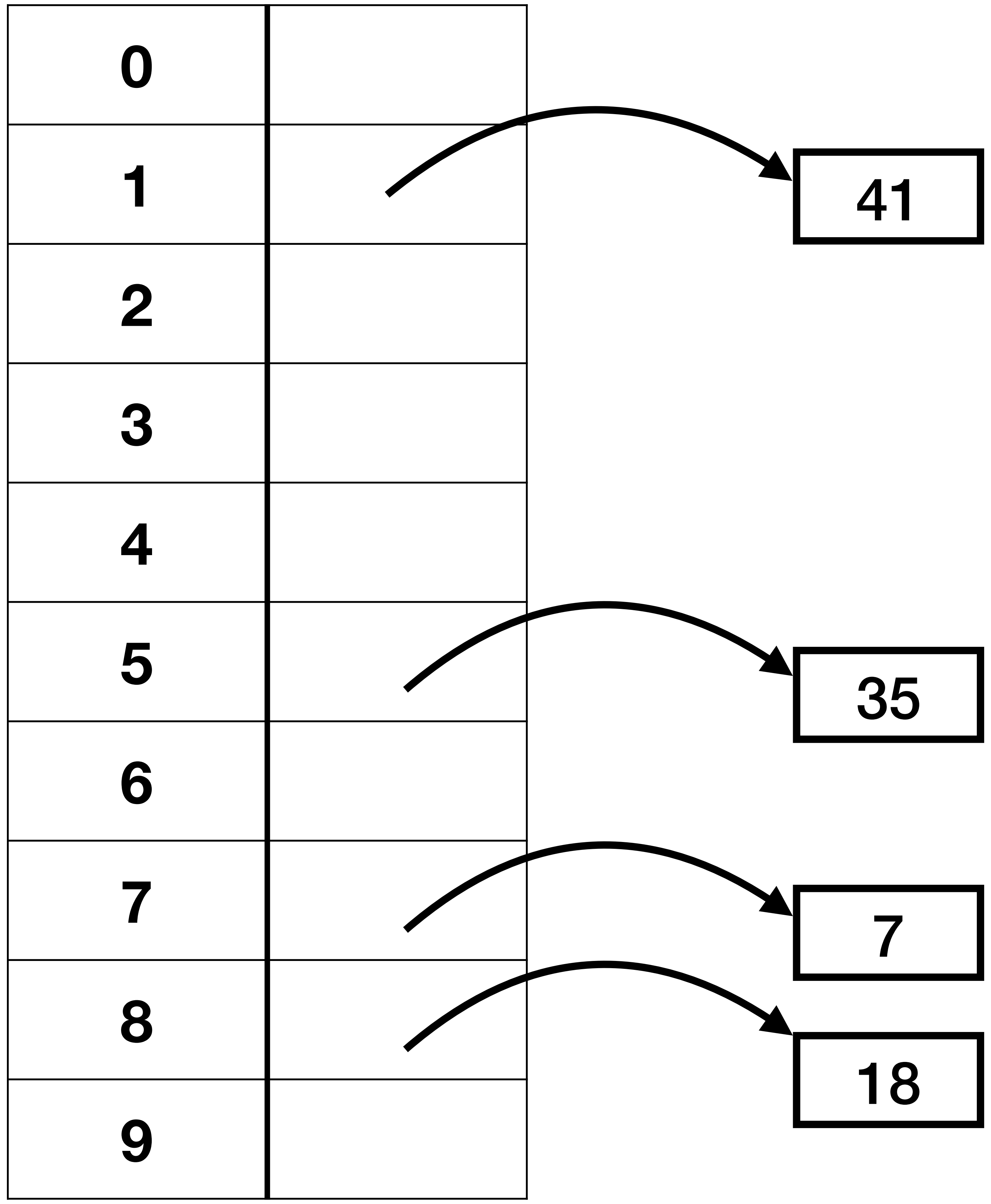



```
insert(table, key, value):
 bucket_idx = hash(key) % table->size
 if found key in table->buckets[bucket_idx]
  replace value
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

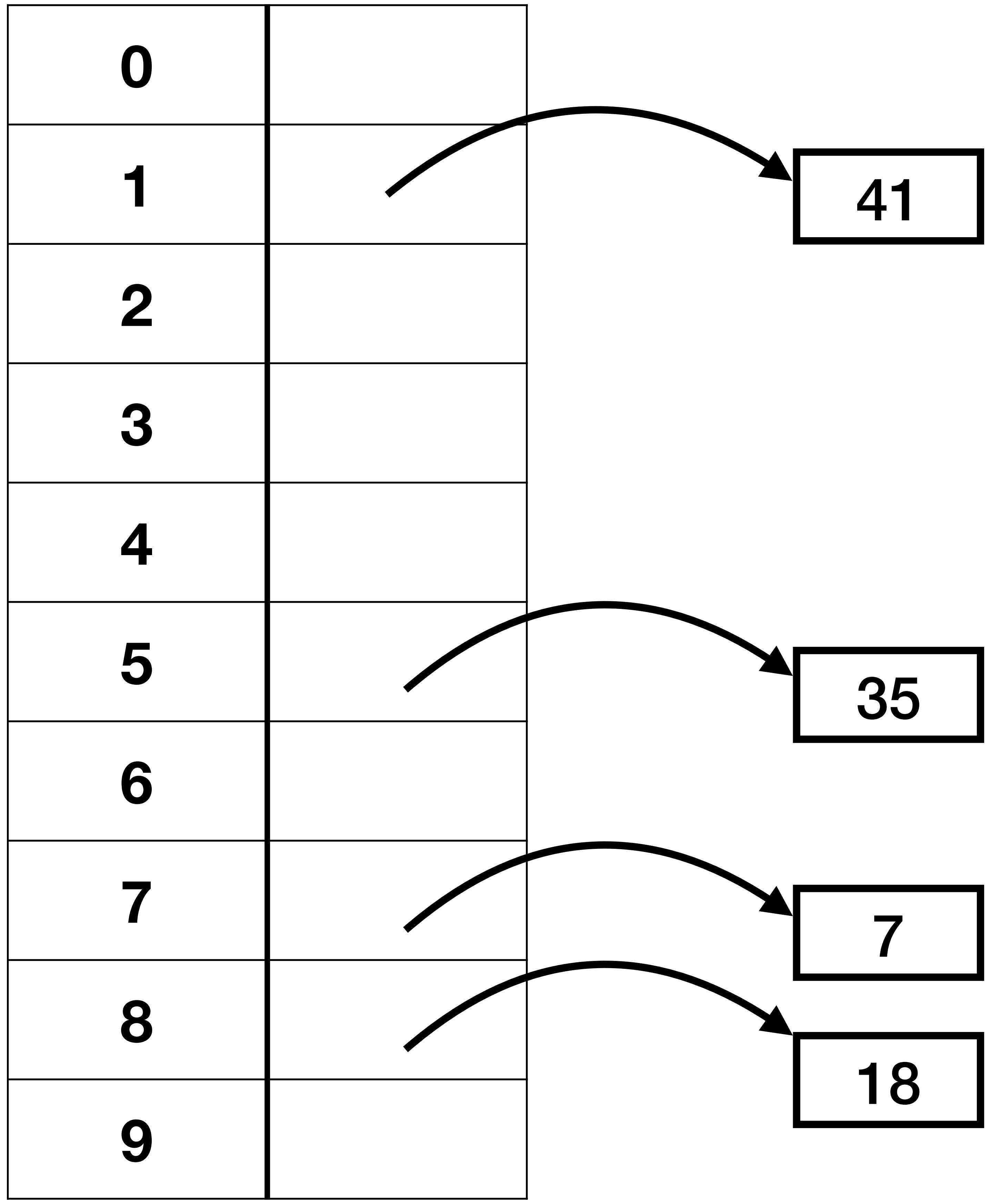



```
insert(table, key, value):
 bucket_idx = hash(key) % table->size
 if found key in table->buckets[bucket_idx]
  replace value
 else:
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

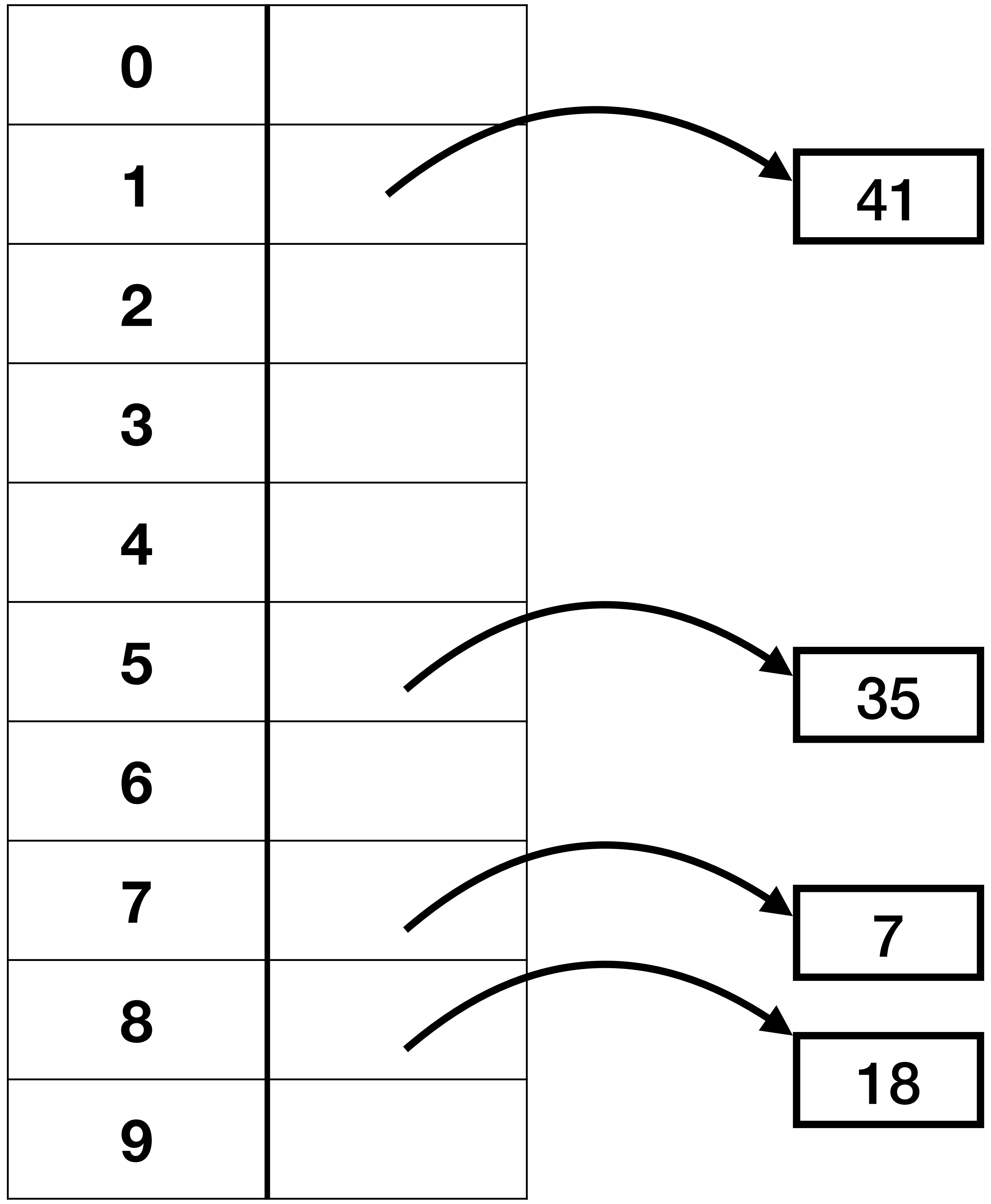



```
insert(table, key, value):
 bucket_idx = hash(key) % table->size
 if found key in table->buckets[bucket_idx]
  replace value
 else:
  add (key, value) into the list
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

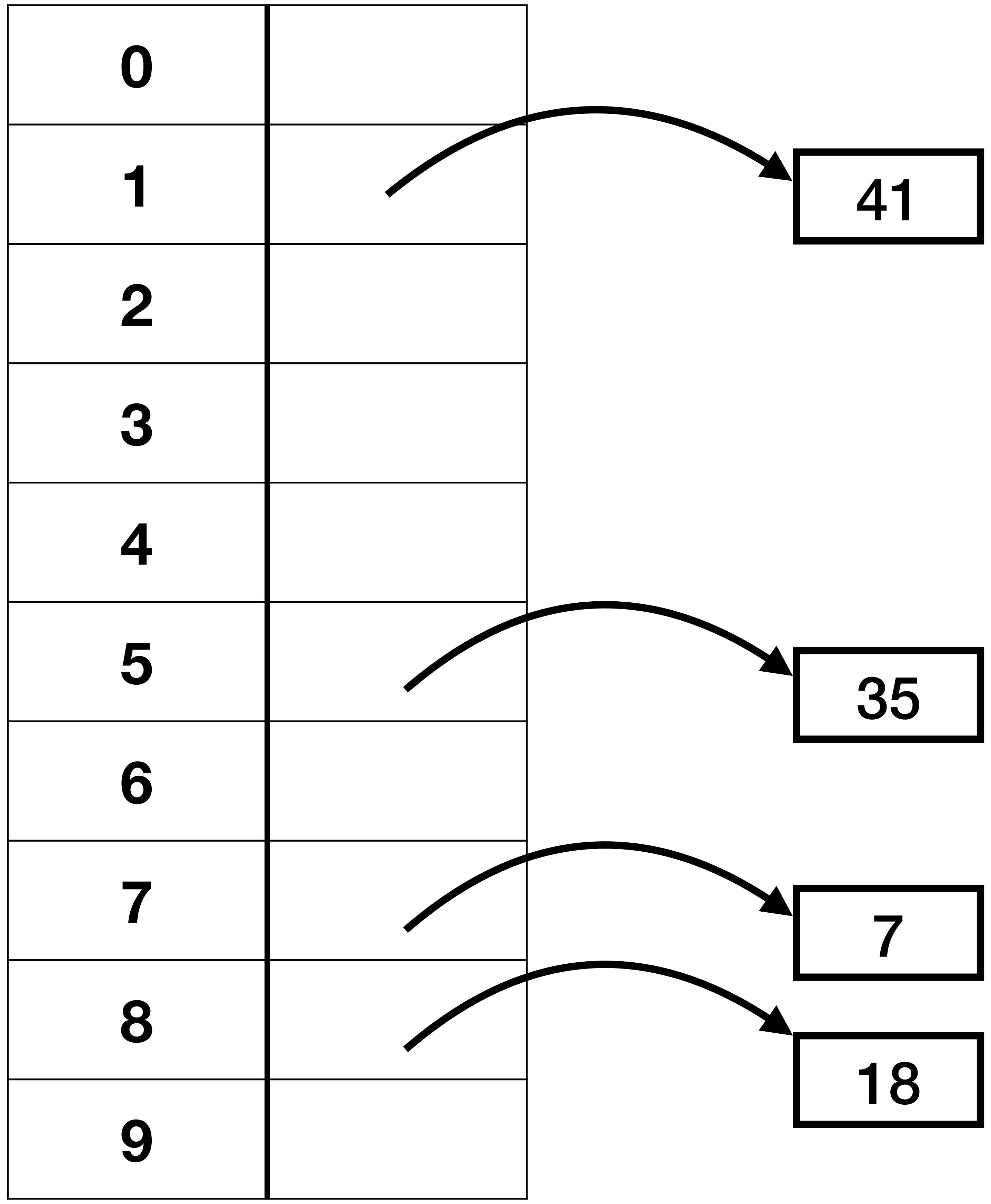



```
insert(table, key, value):
 bucket_idx = hash(key) % table->size
 if found key in table->buckets[bucket_idx]
  replace value
 else:
  add (key, value) into the list
```

Chaining

Insert

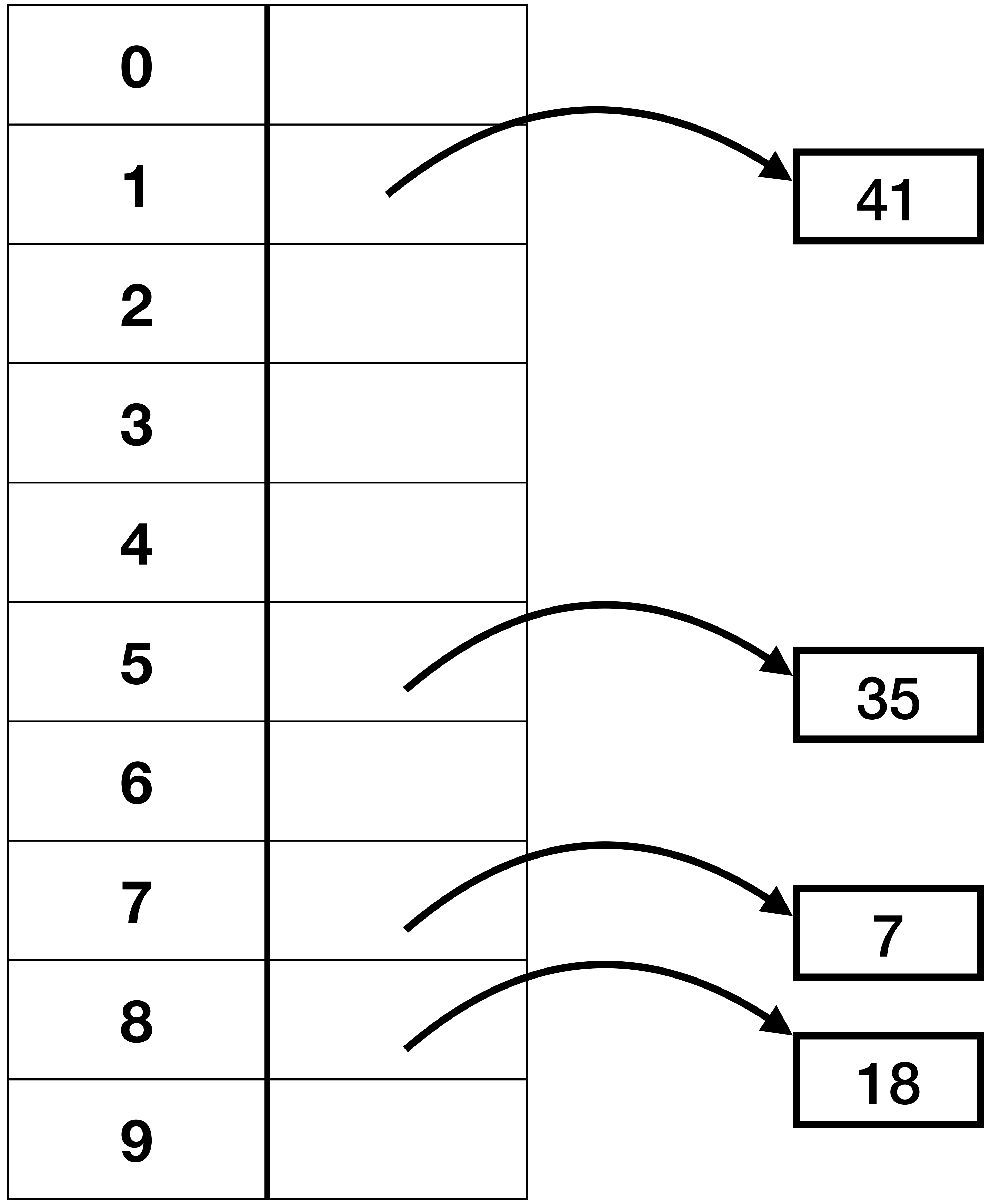
- Each slot is a *list* of key-value pairs, called a *bucket*



Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

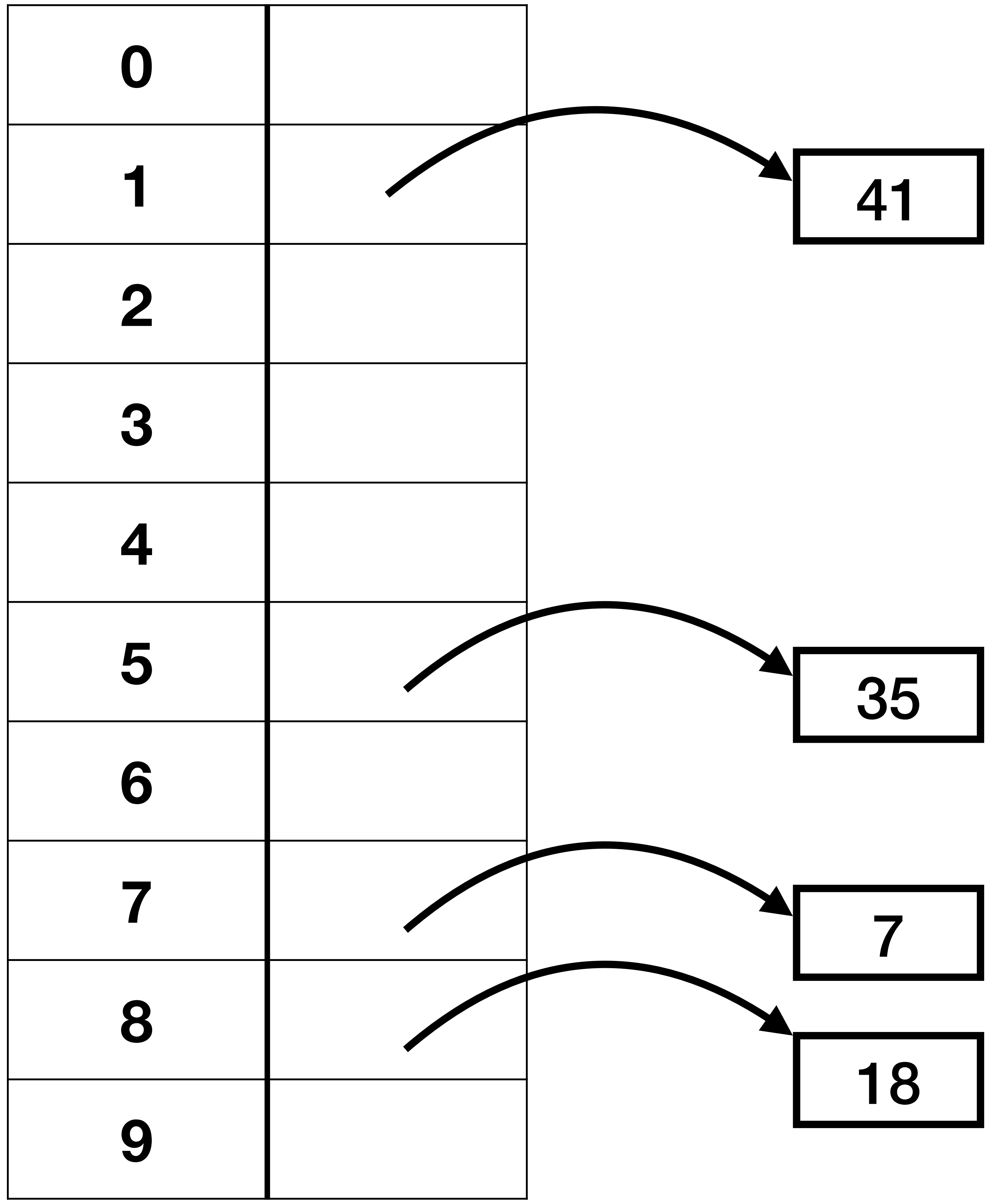



```
struct table {
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

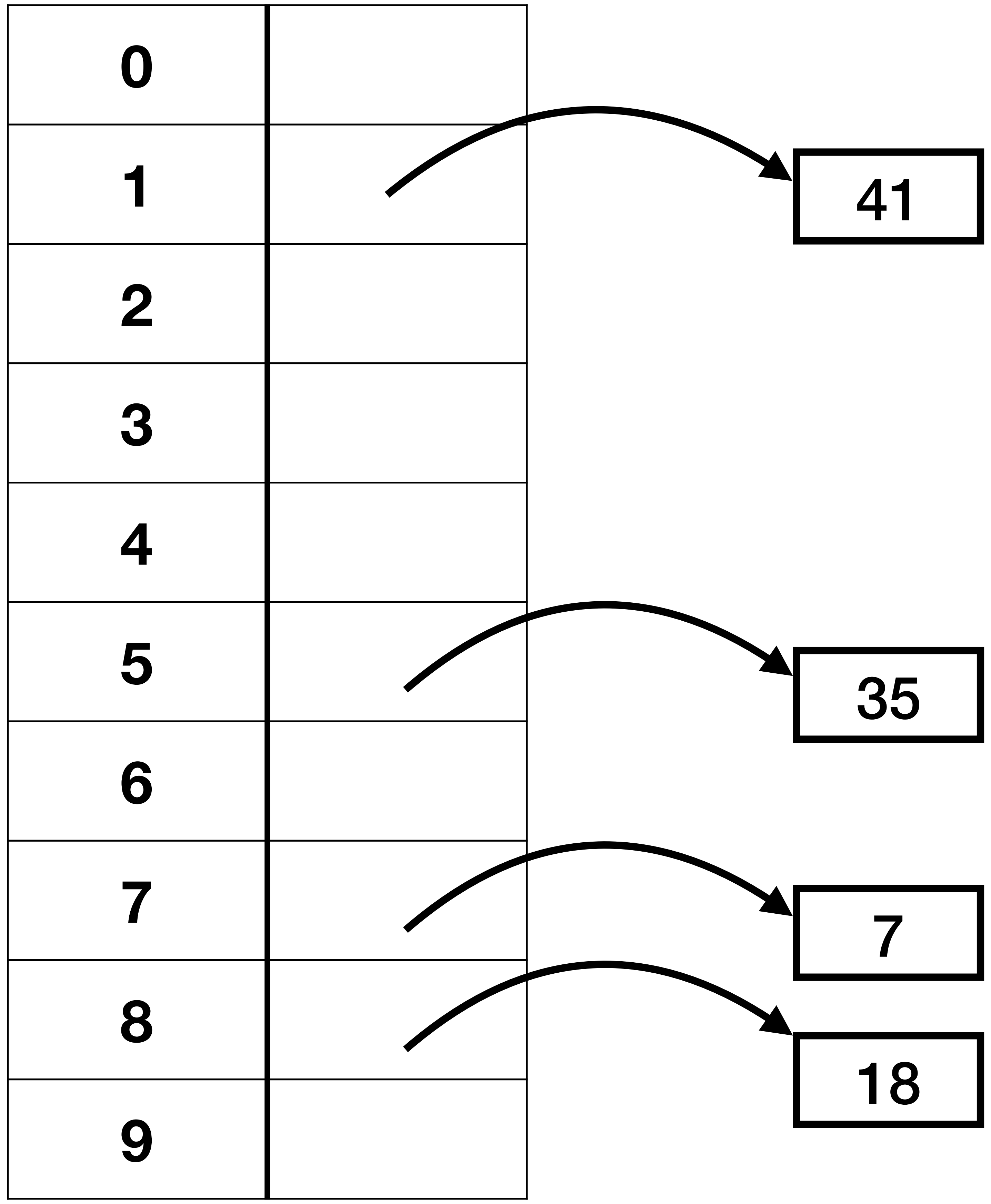



```
struct table {
        int size;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

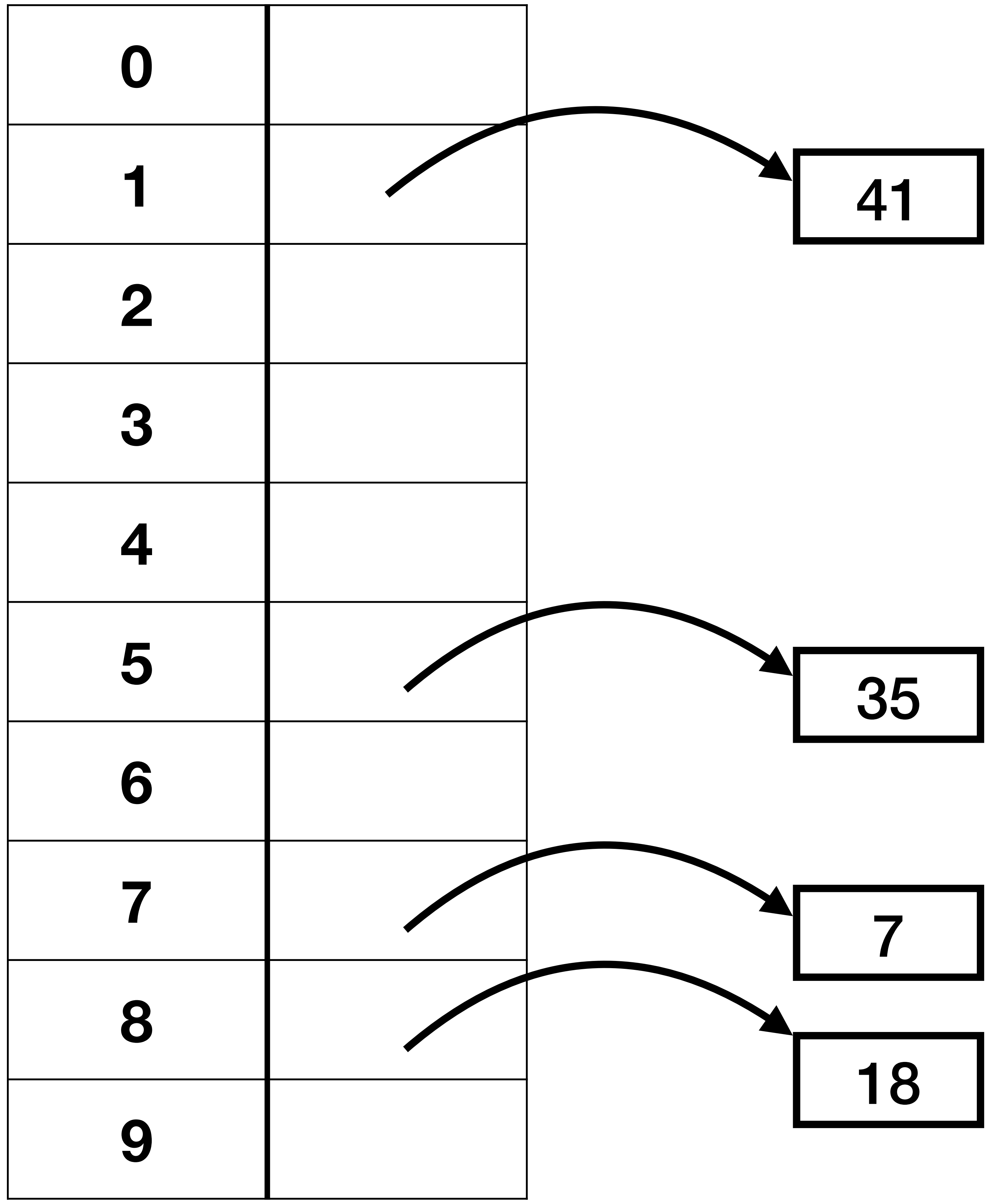



```
struct table {
        int size;
        int length;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

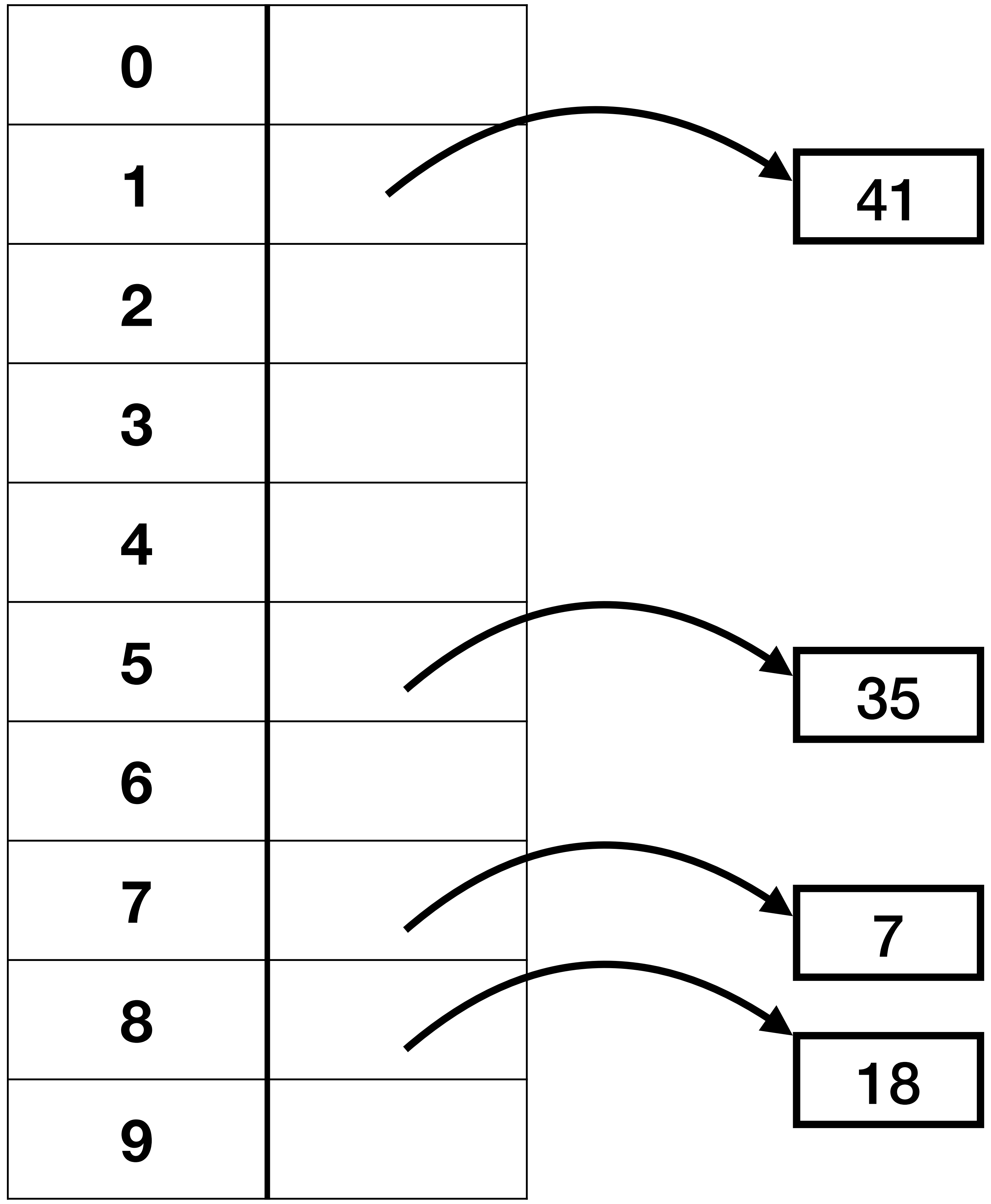



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

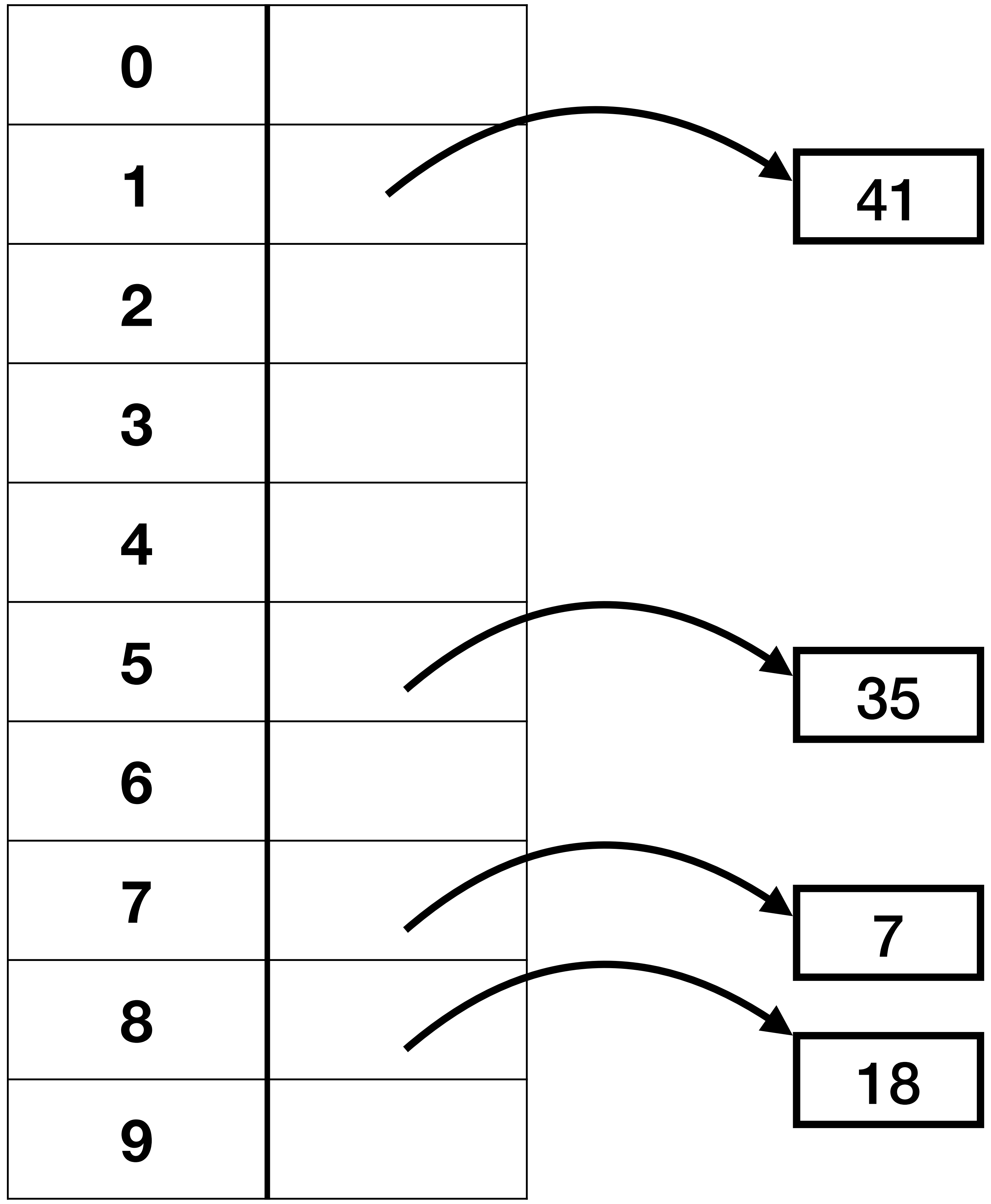



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

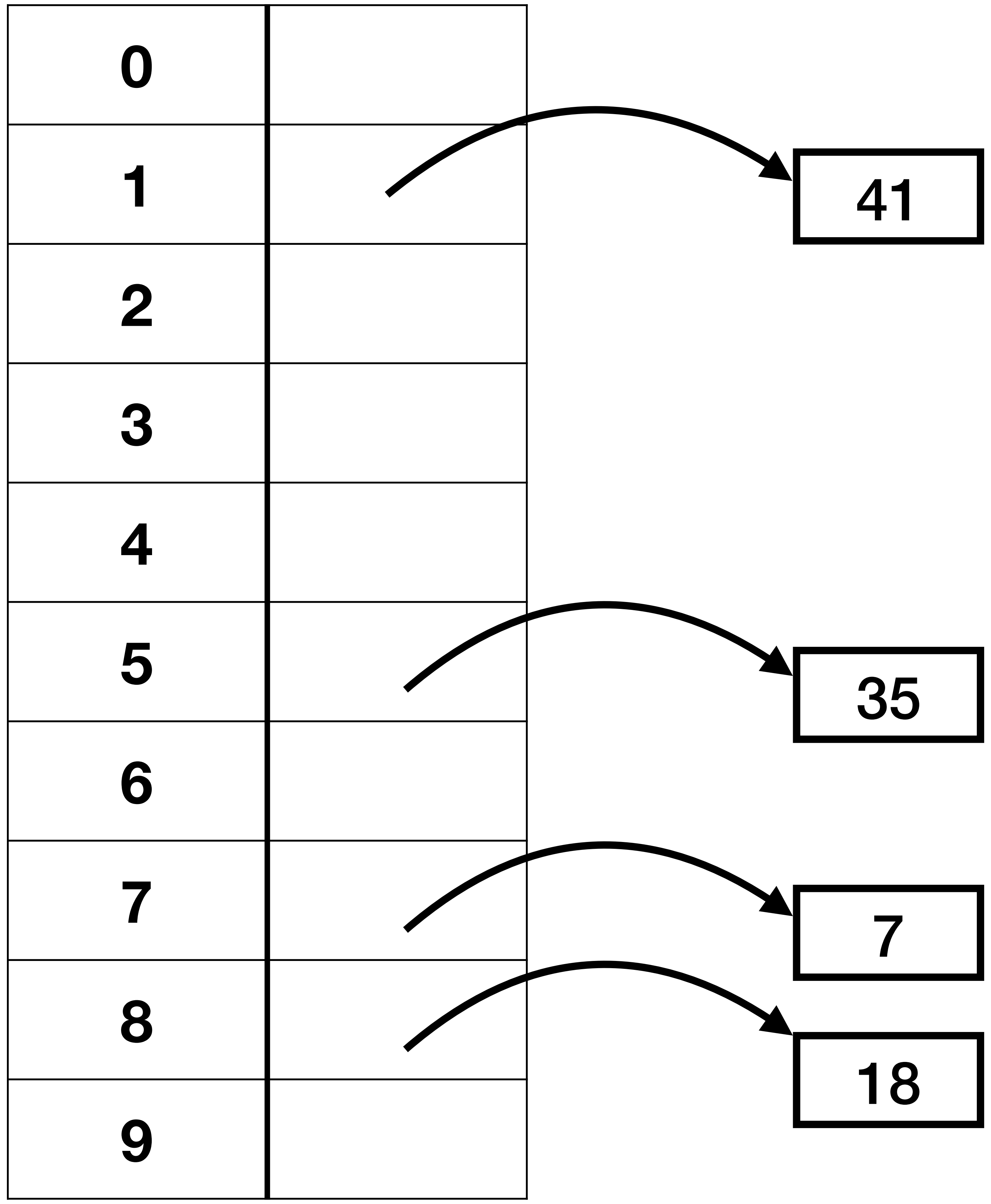



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

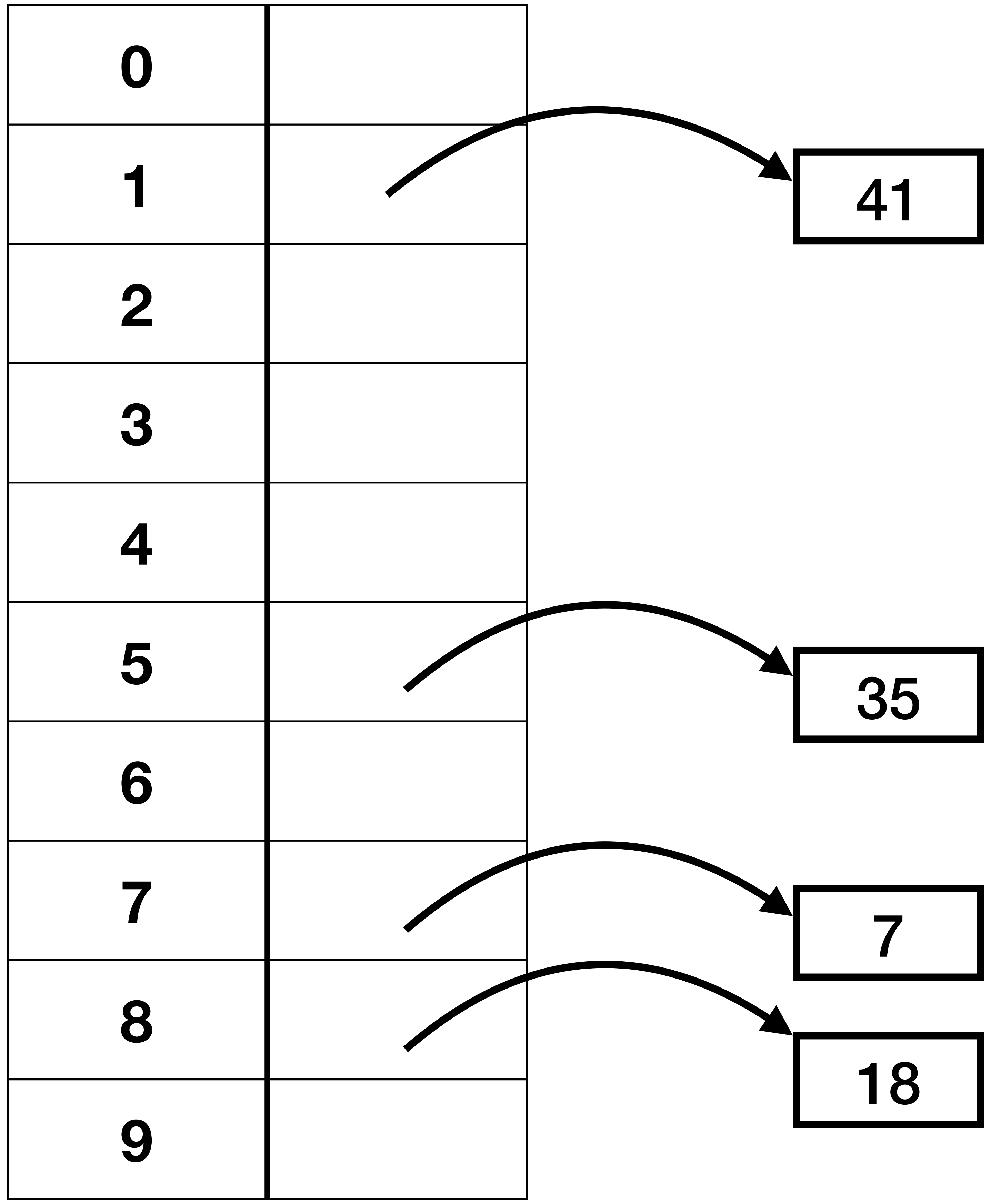



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

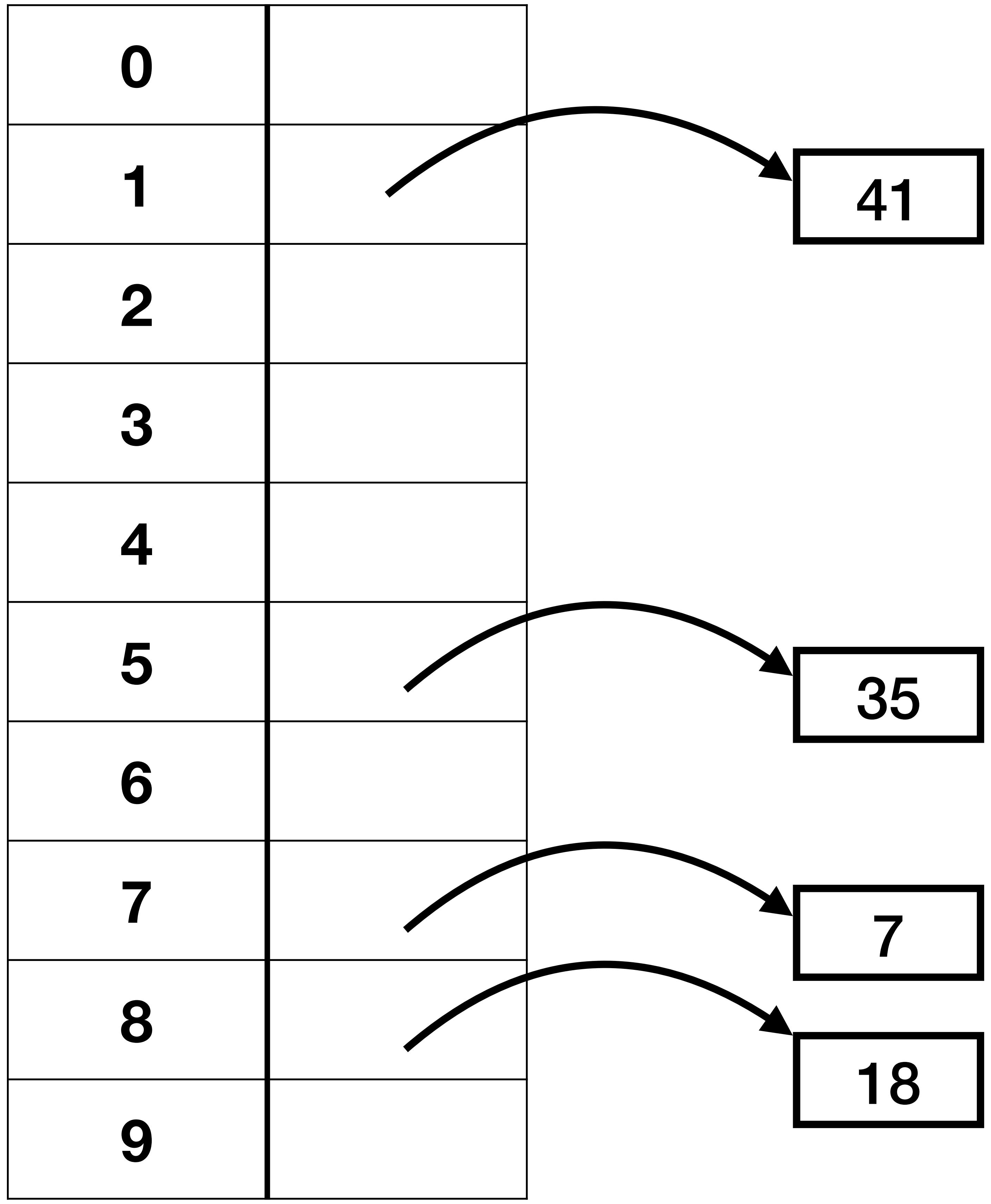



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

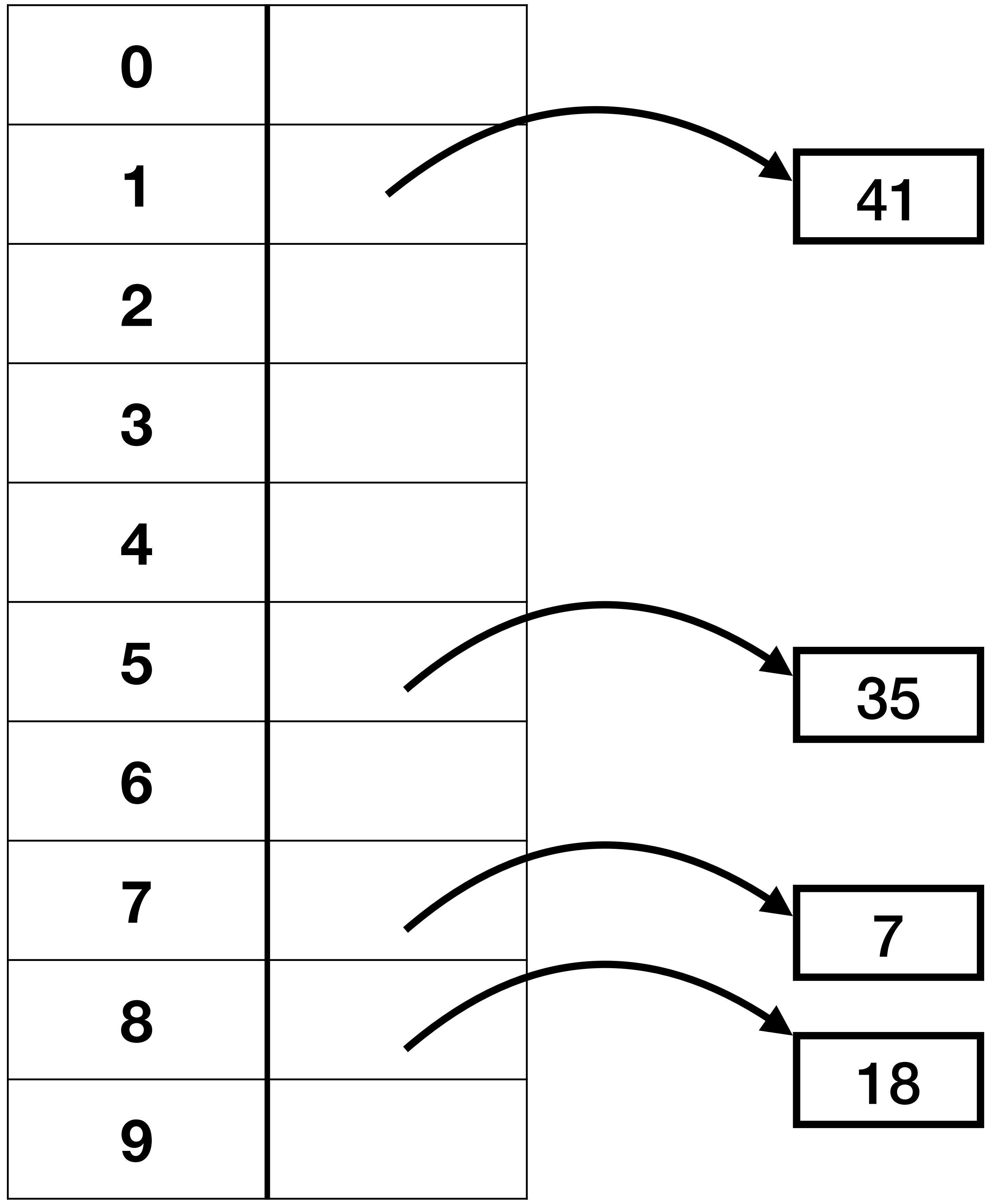



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

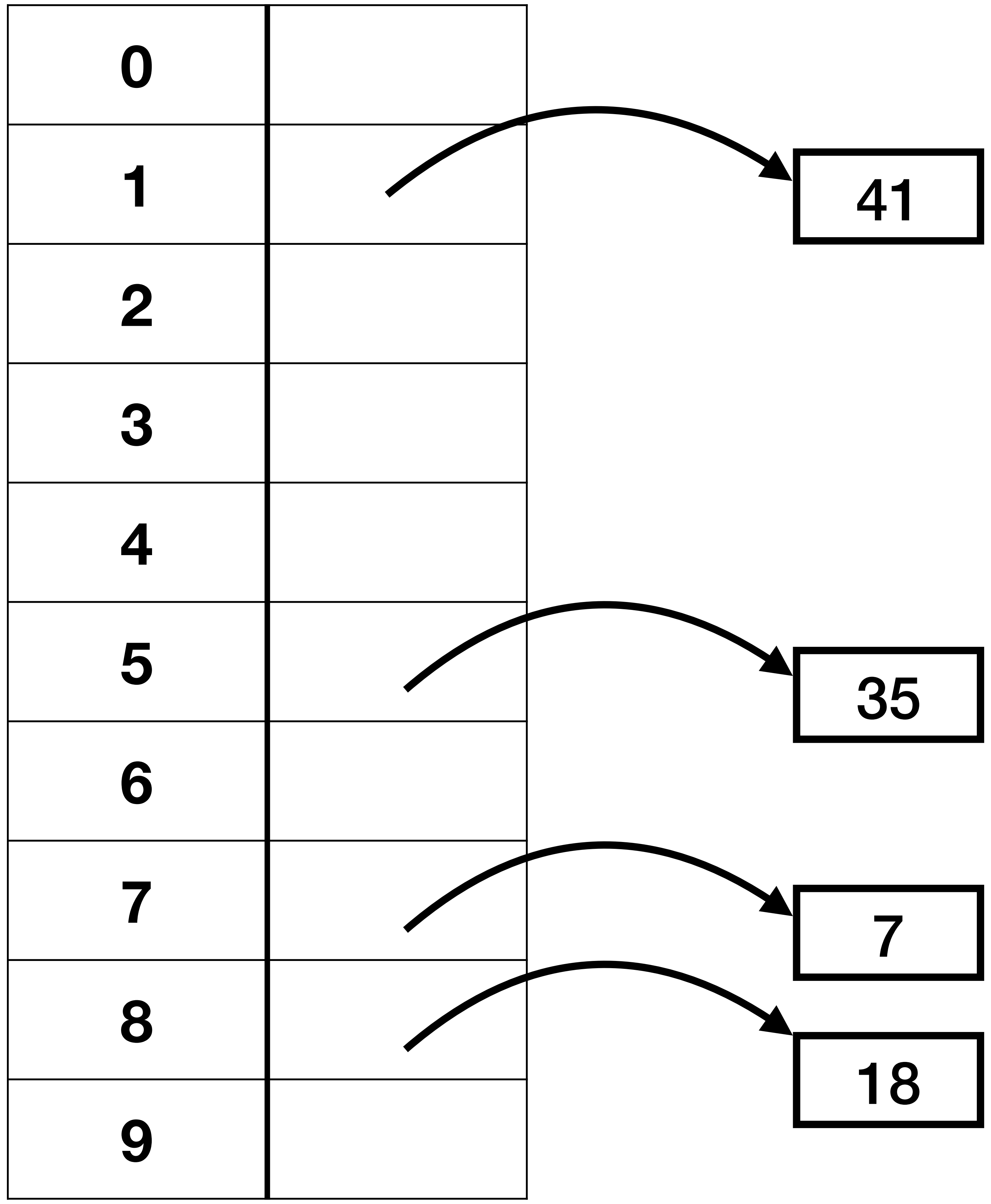



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

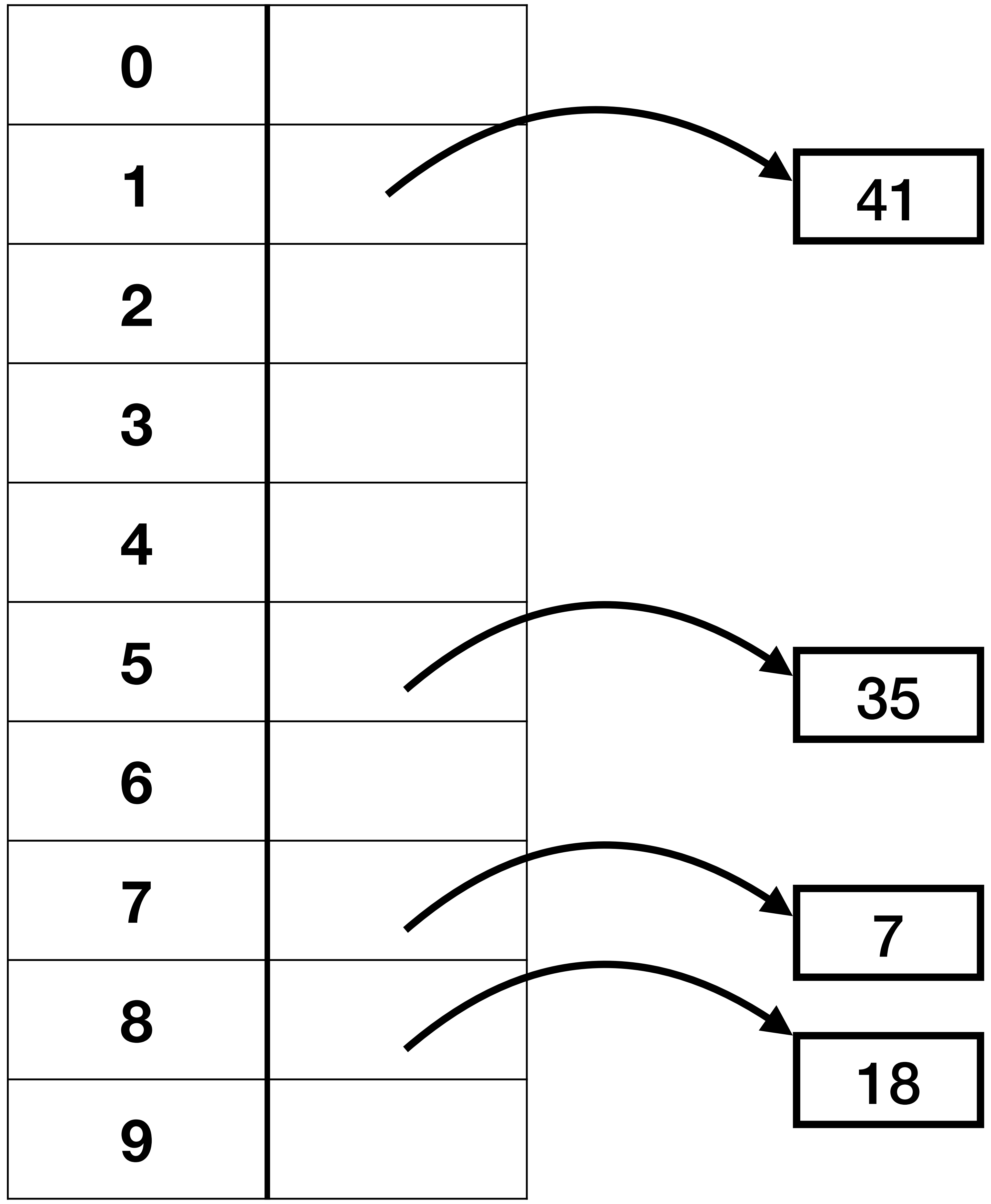



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
        void *value;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

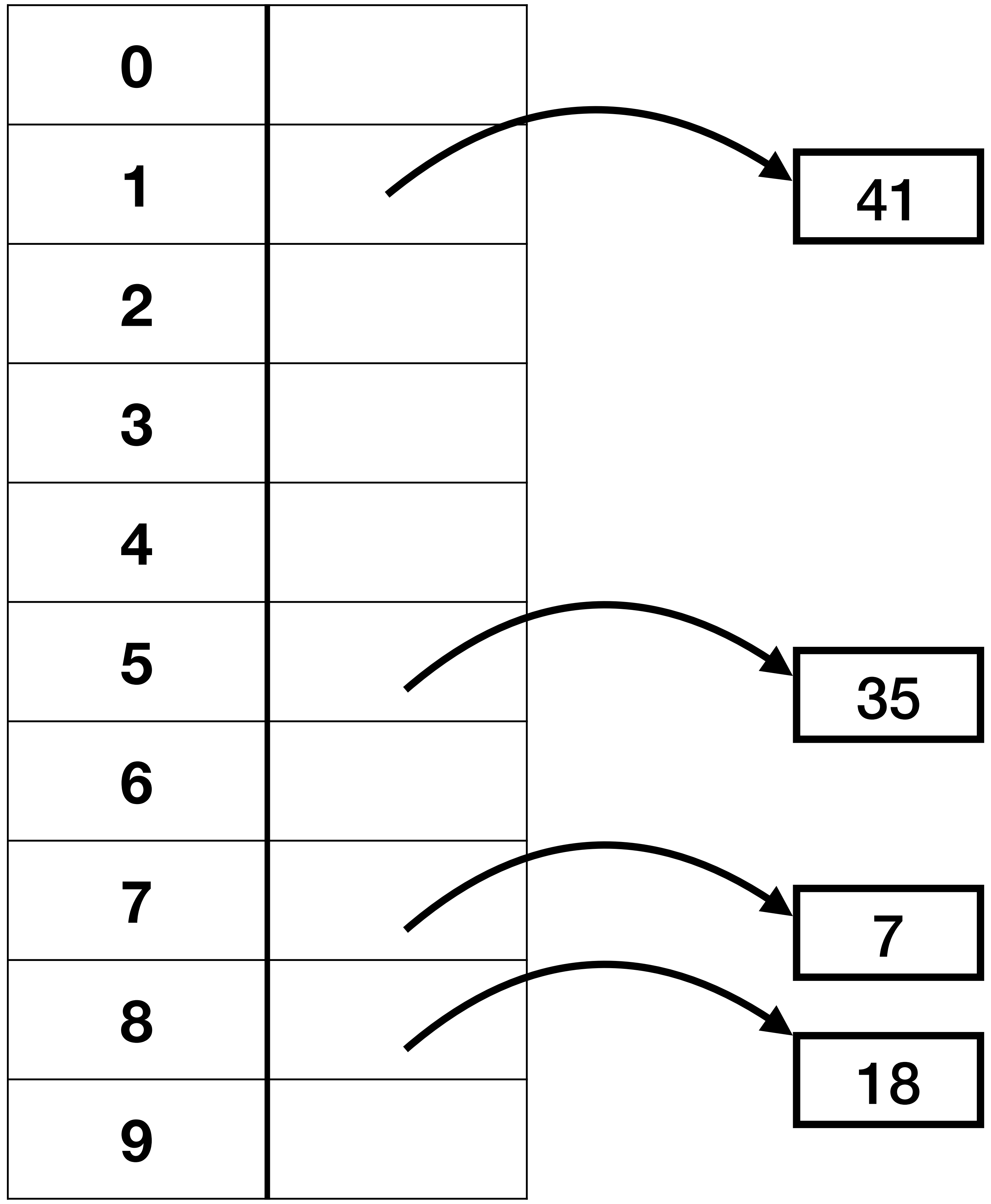



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
        void *value;
        struct bucket *next;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

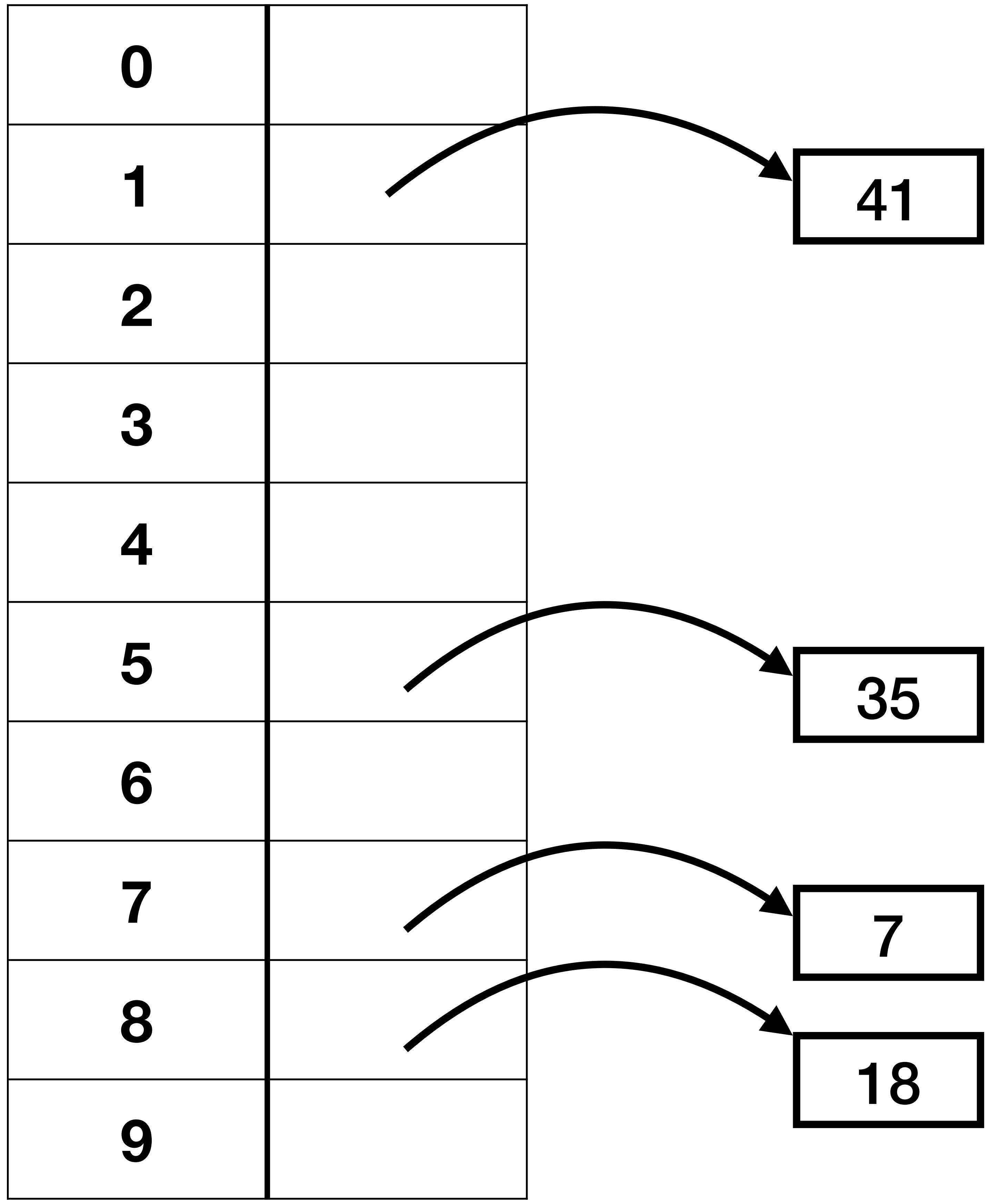



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
        void *value;
        struct bucket *next;
};
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

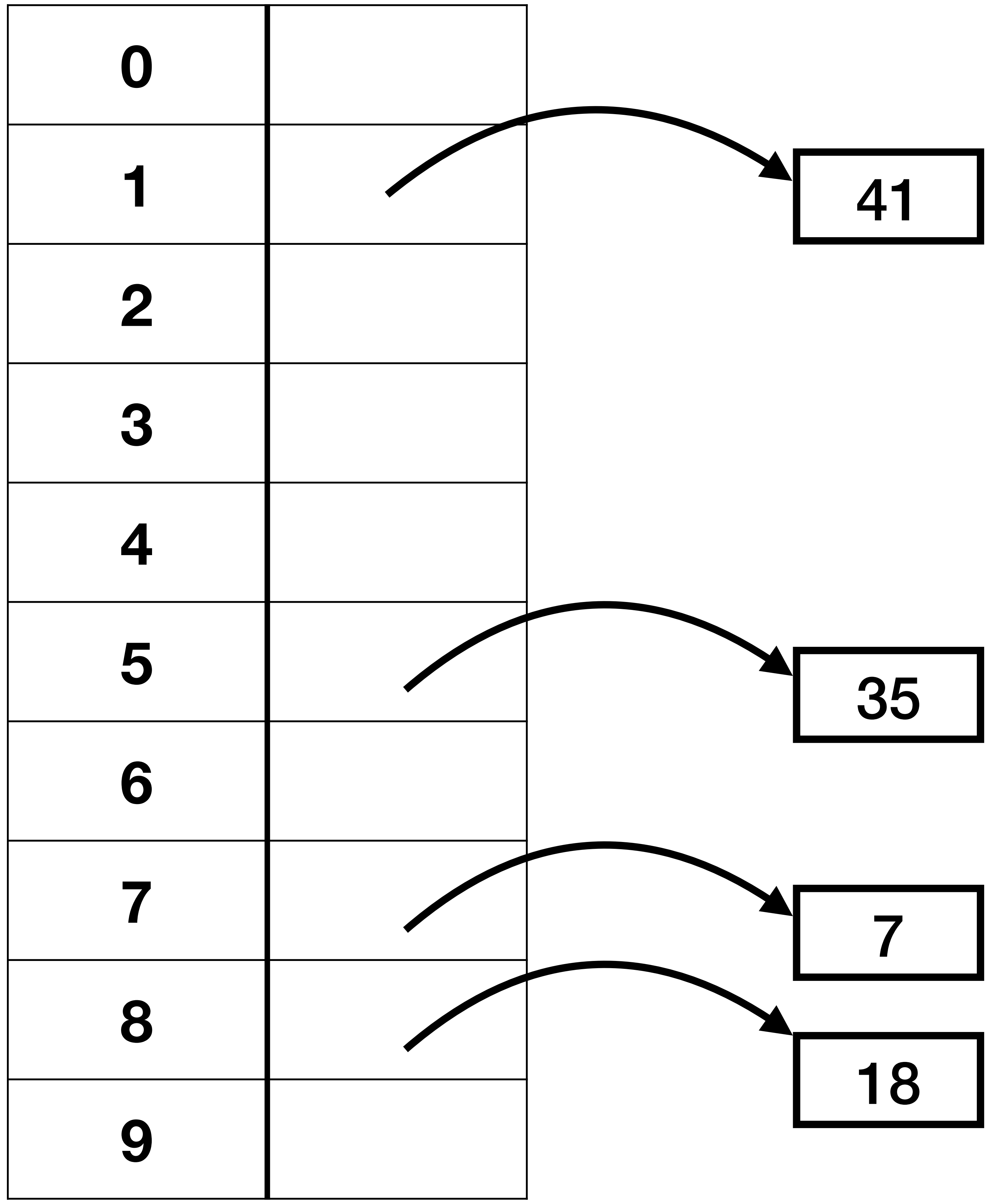



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
        void *value;
        struct bucket *next;
};
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

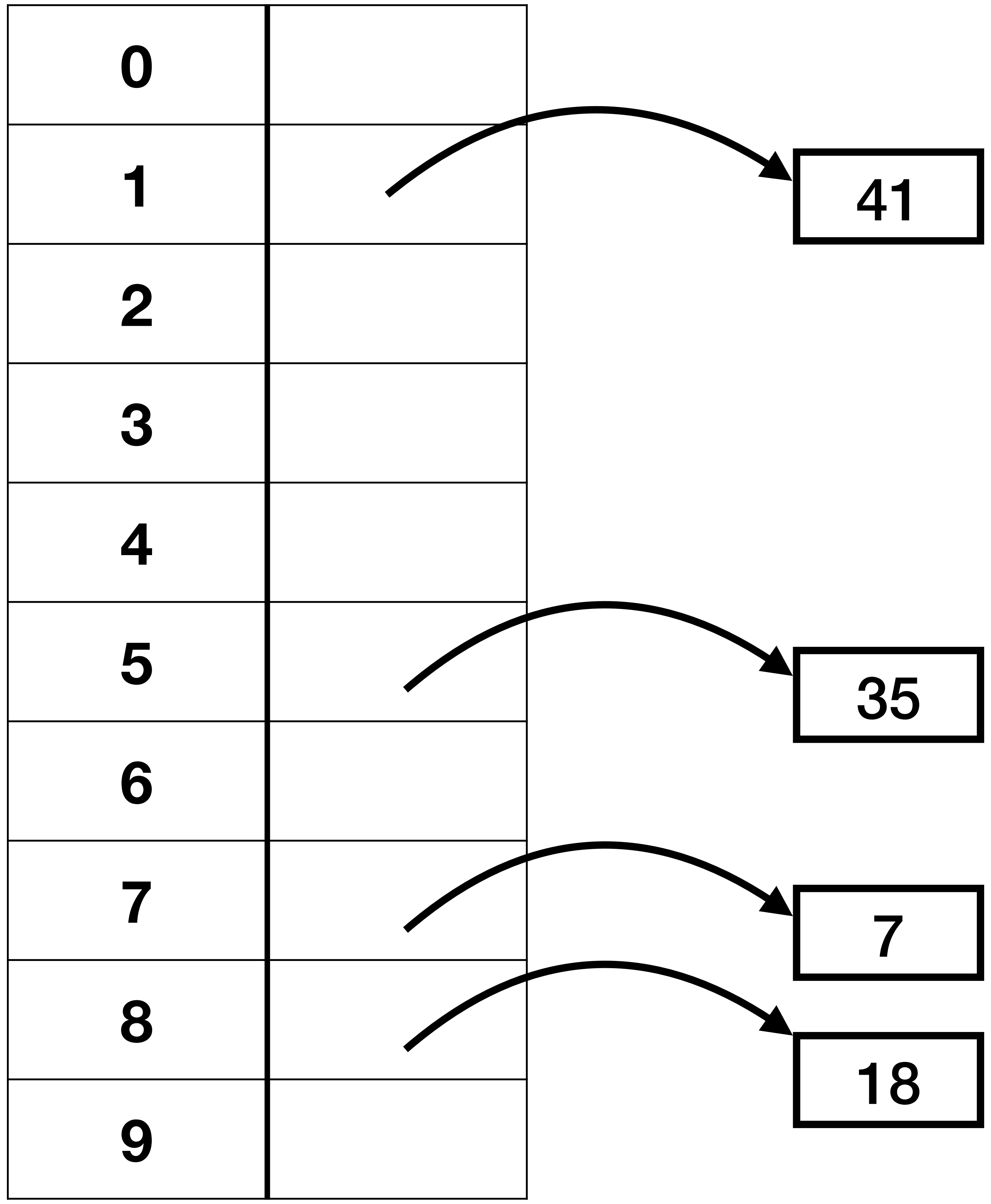


<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



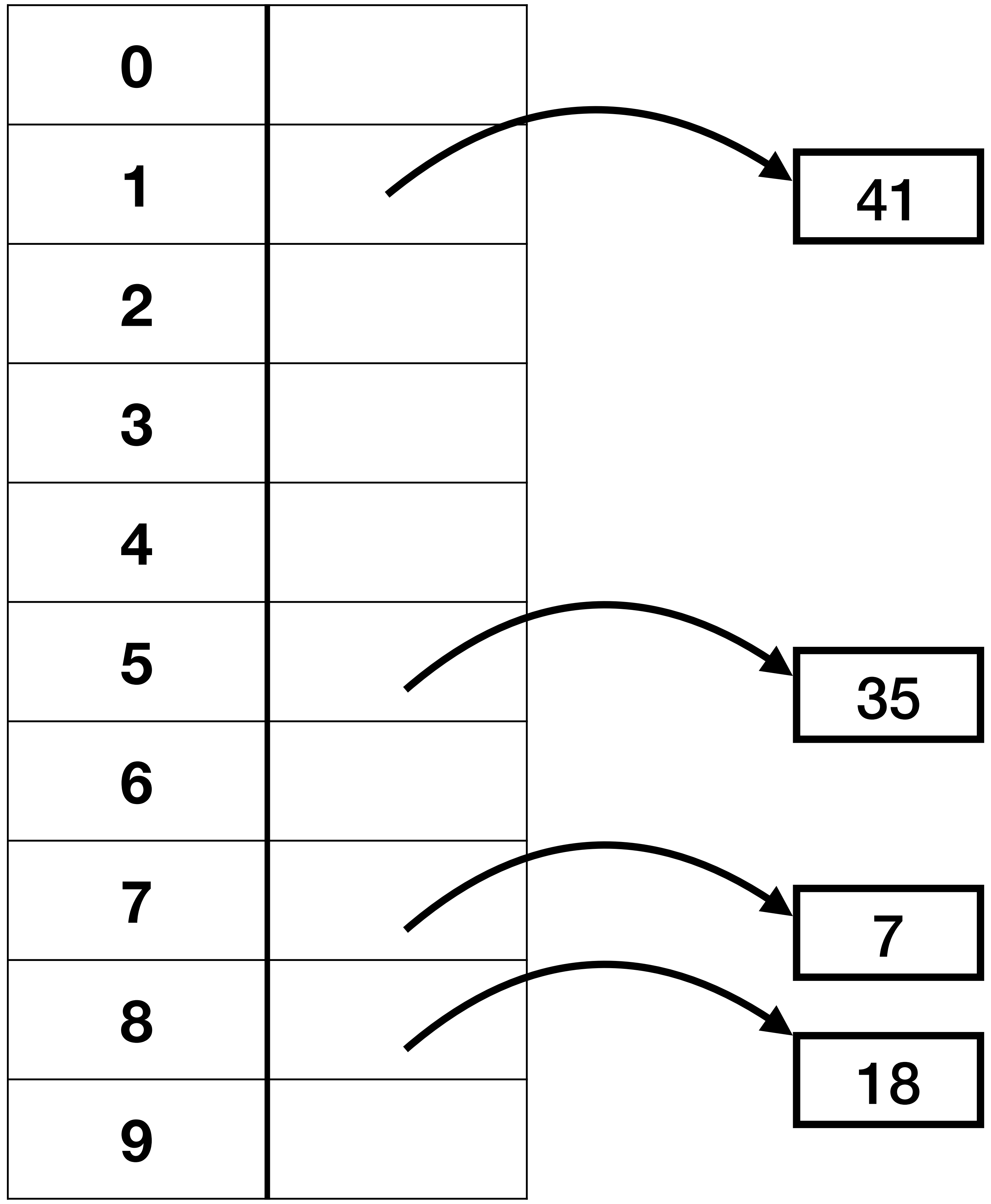

```
struct table {
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



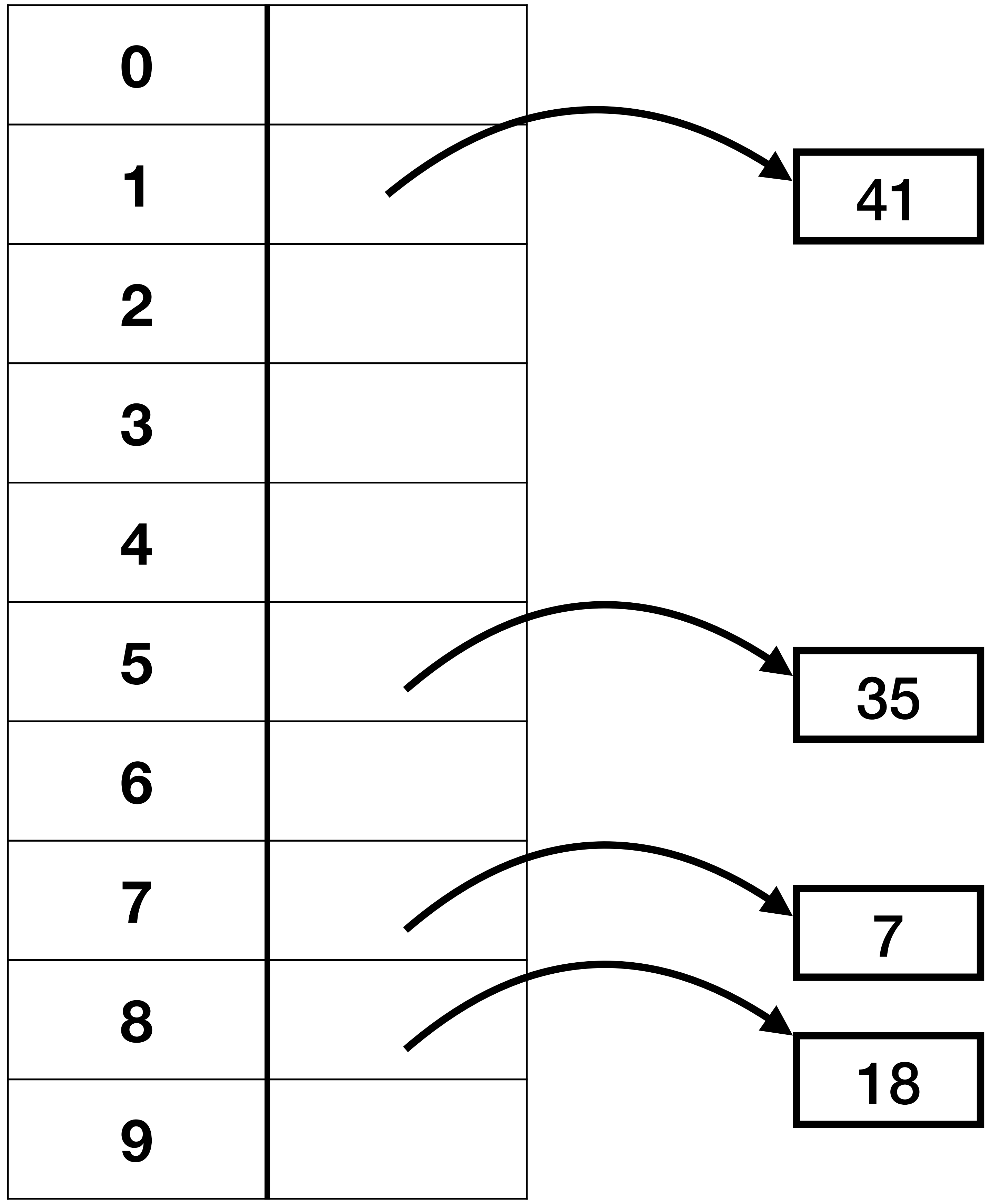

```
struct table {
        int size;
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



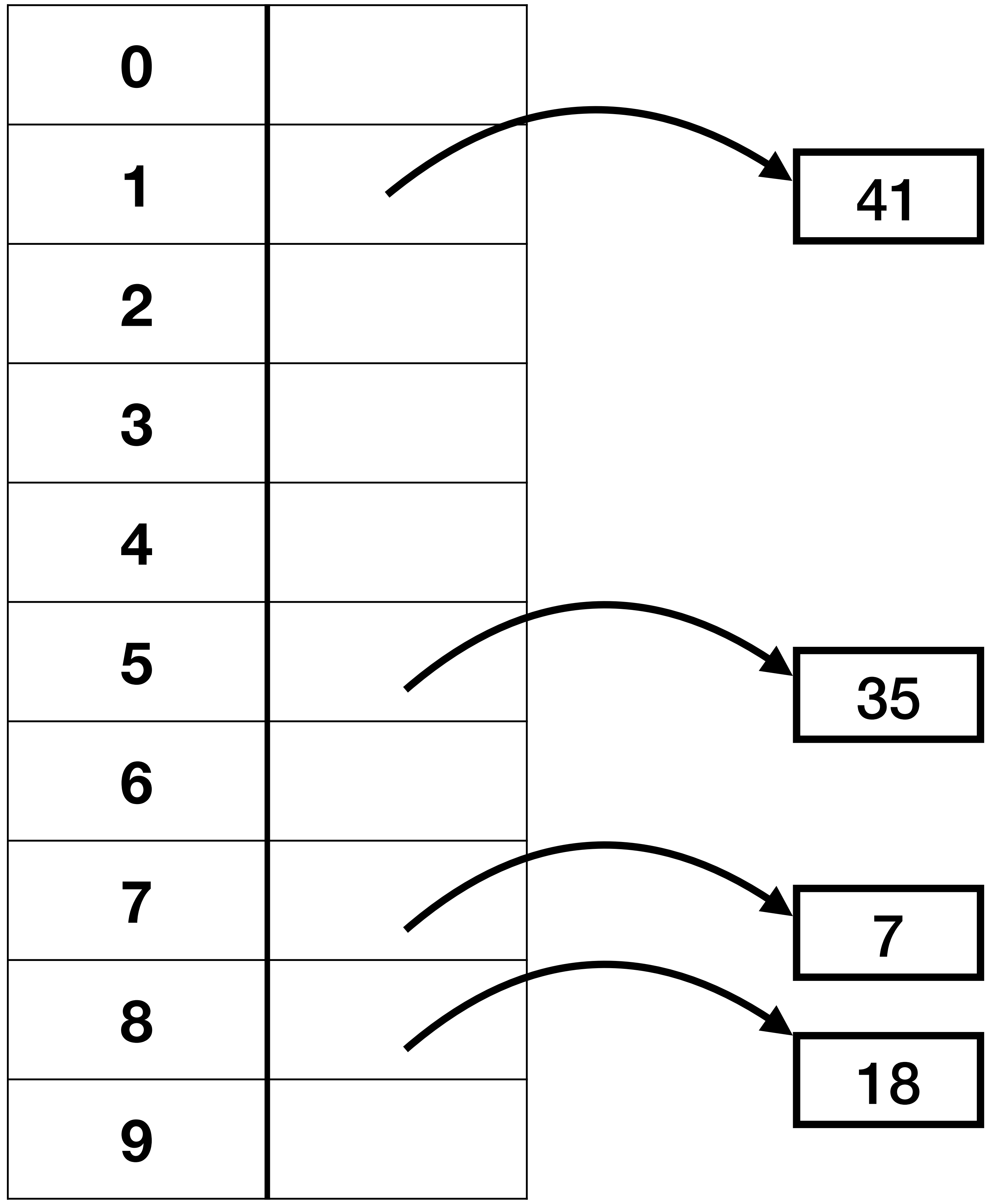

```
struct table {
        int size;
        int length;
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



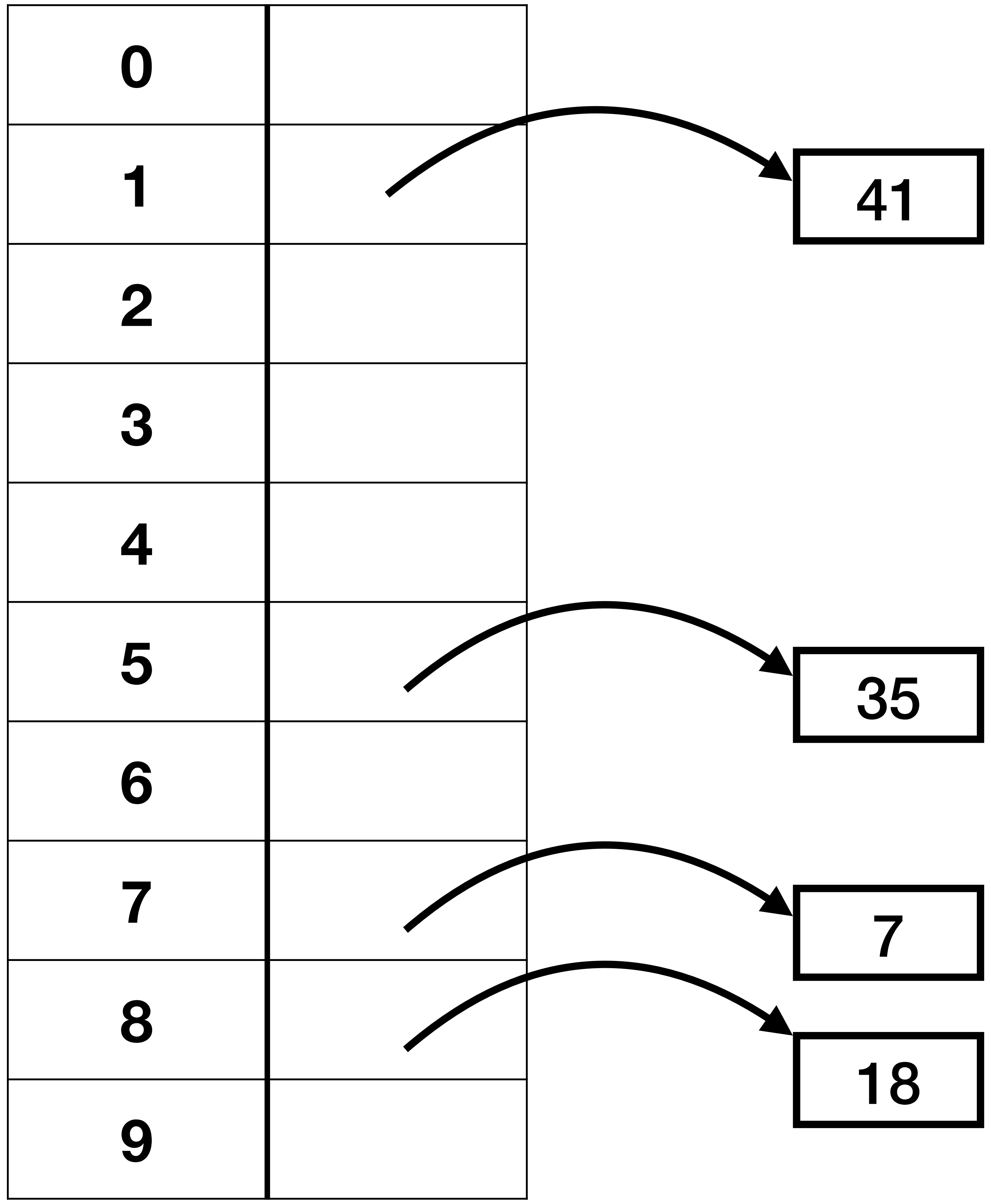

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



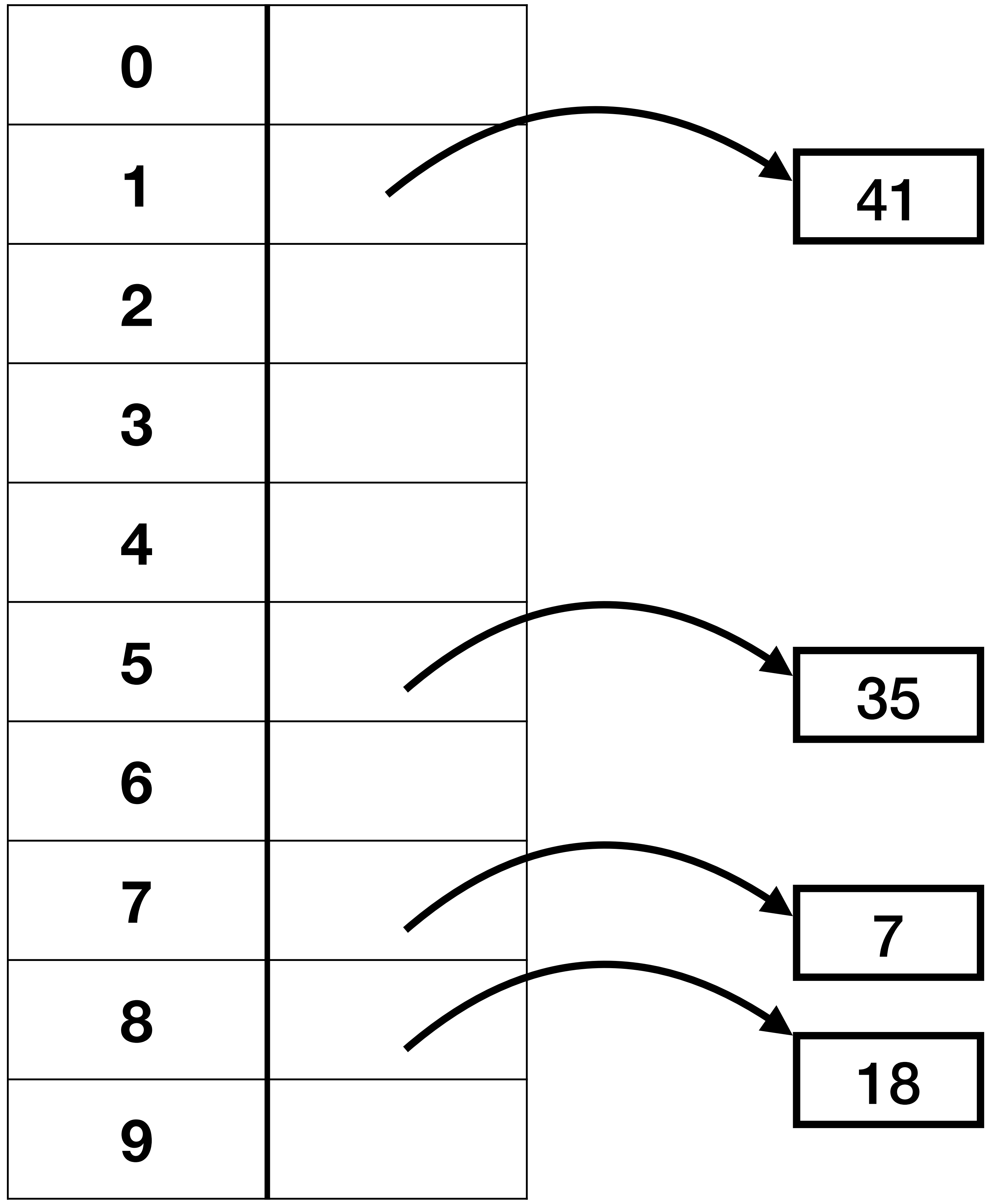

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



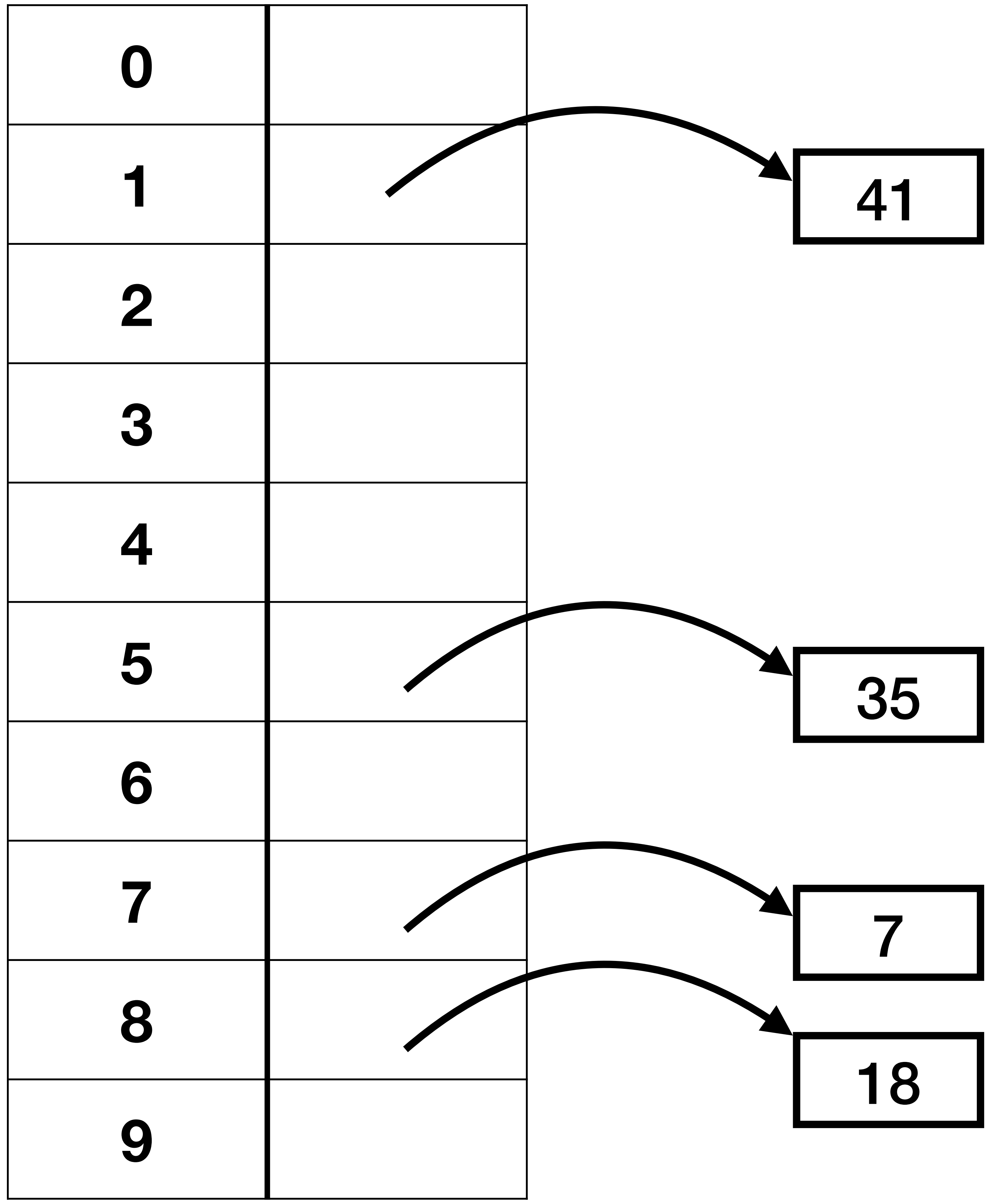

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



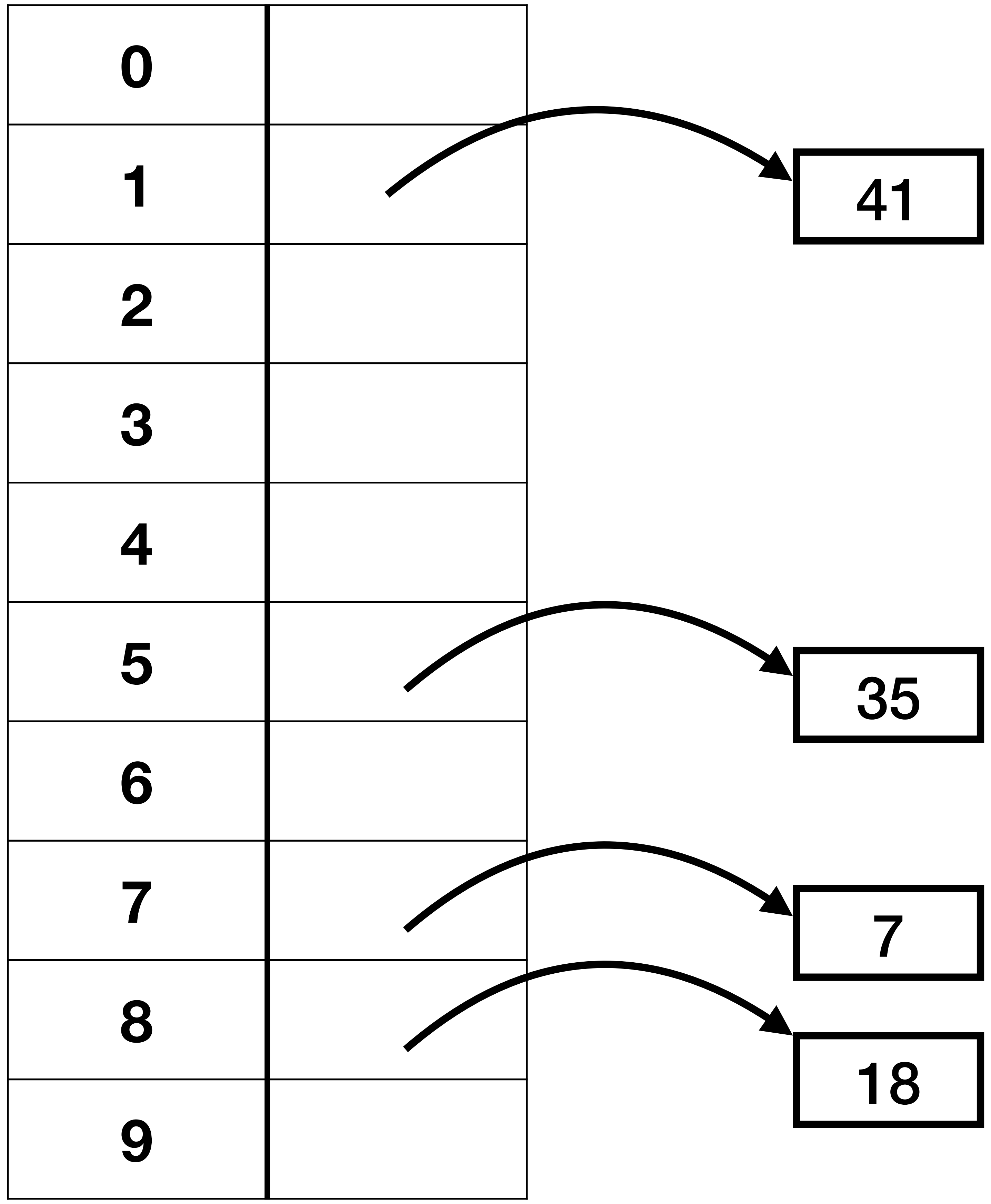

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



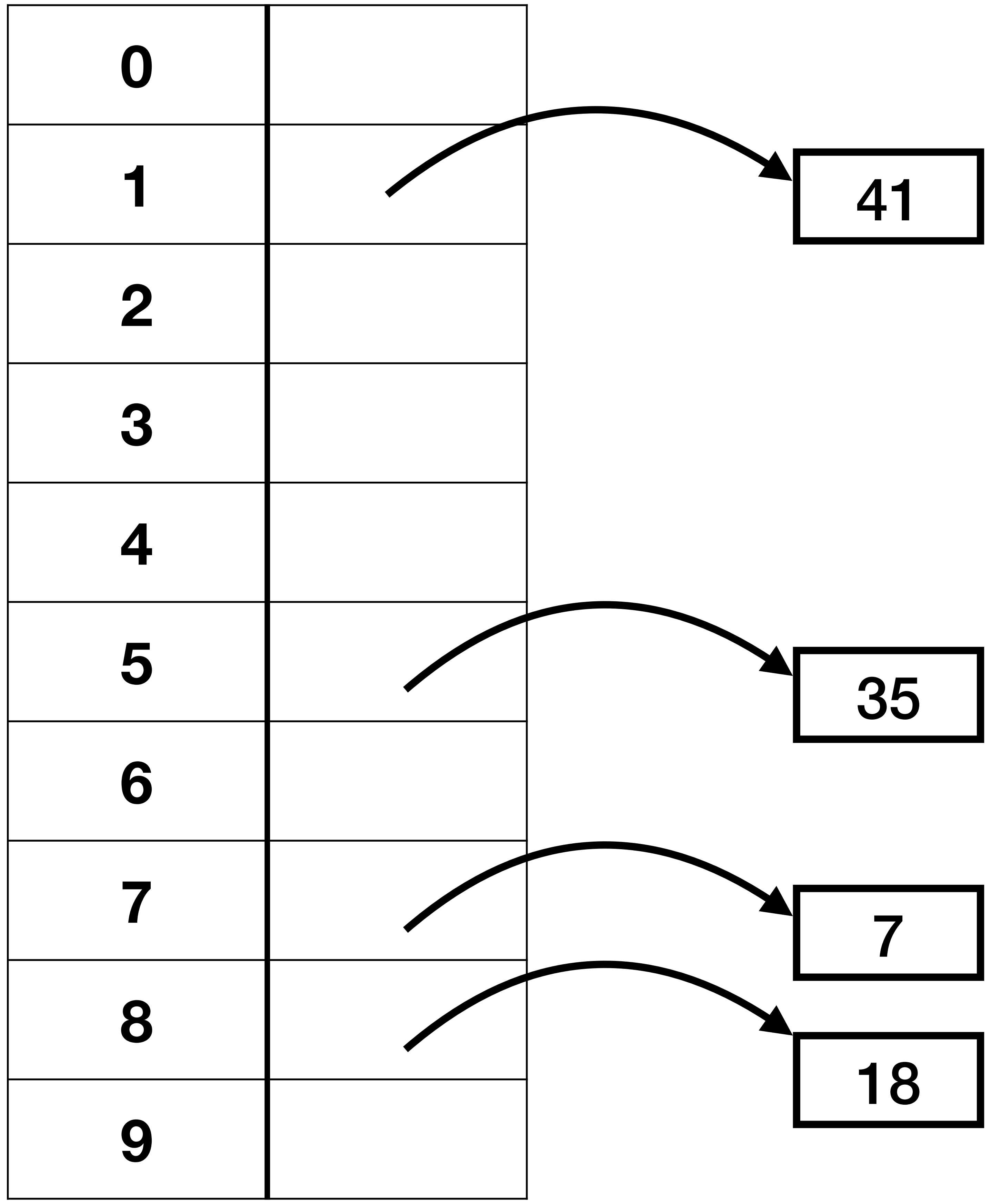

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

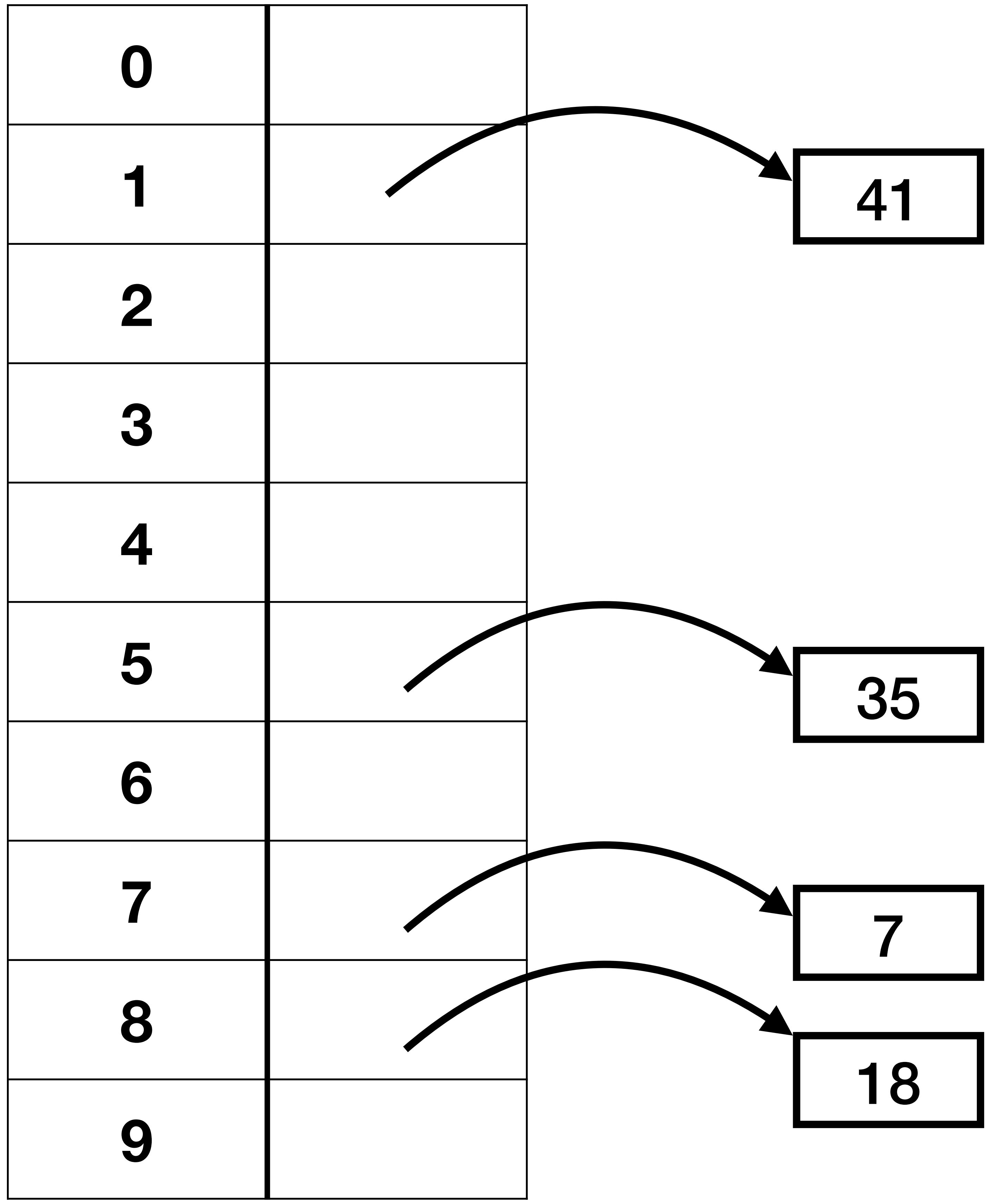



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];   <<-- what is this?
};

struct bucket {
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

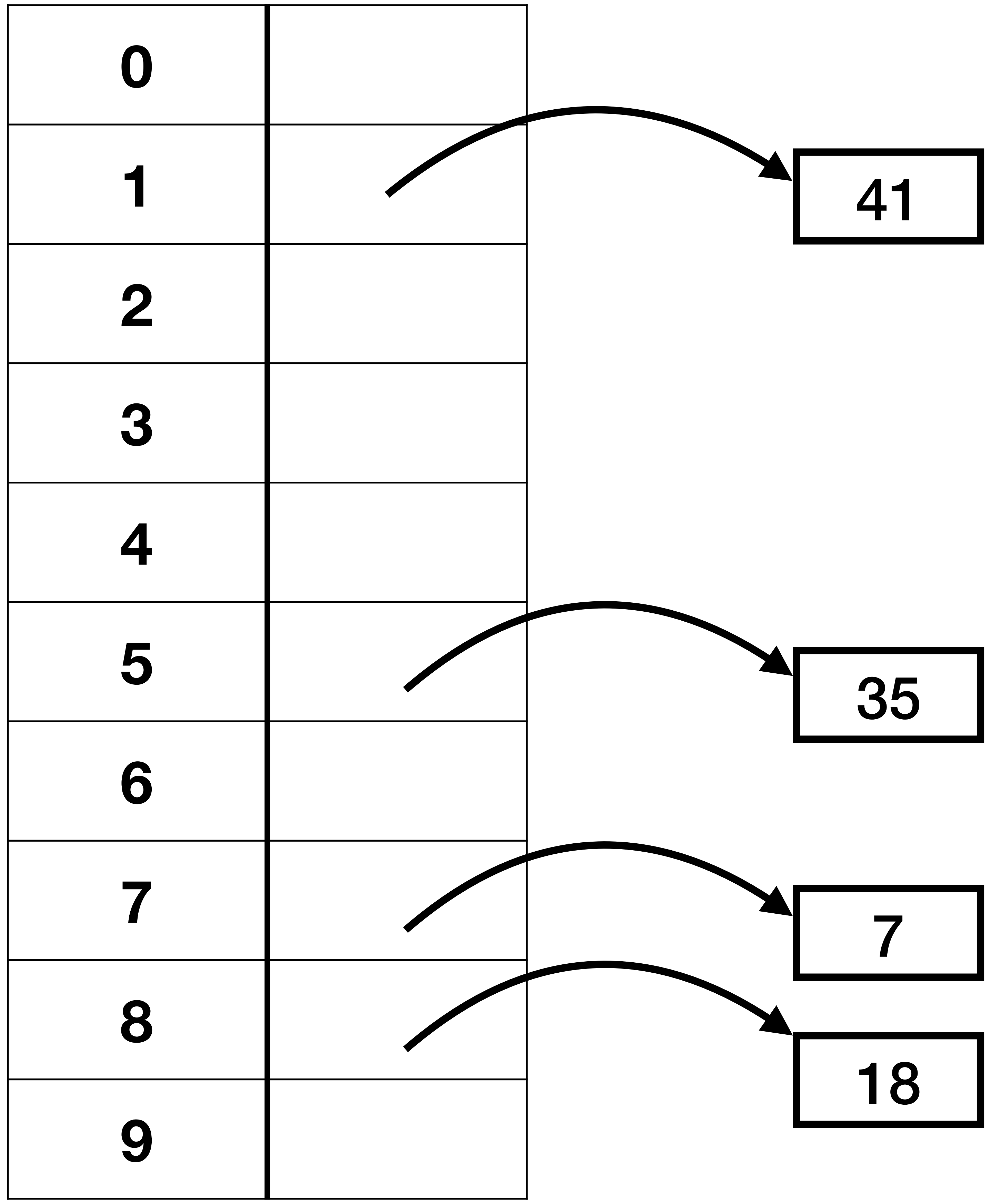



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];   <<-- what is this?
};

struct bucket {
        void *key;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



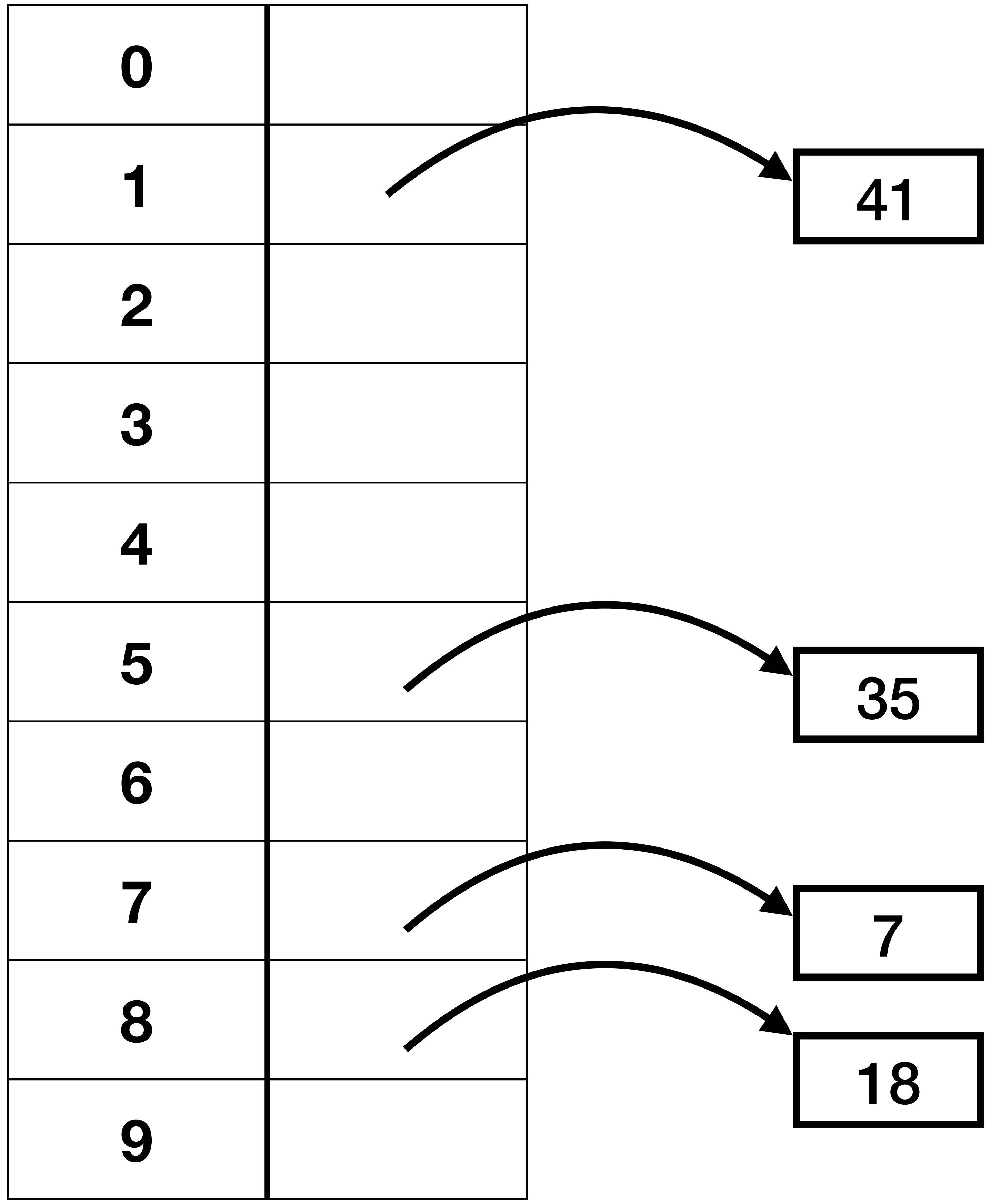

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
        void *value;
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*

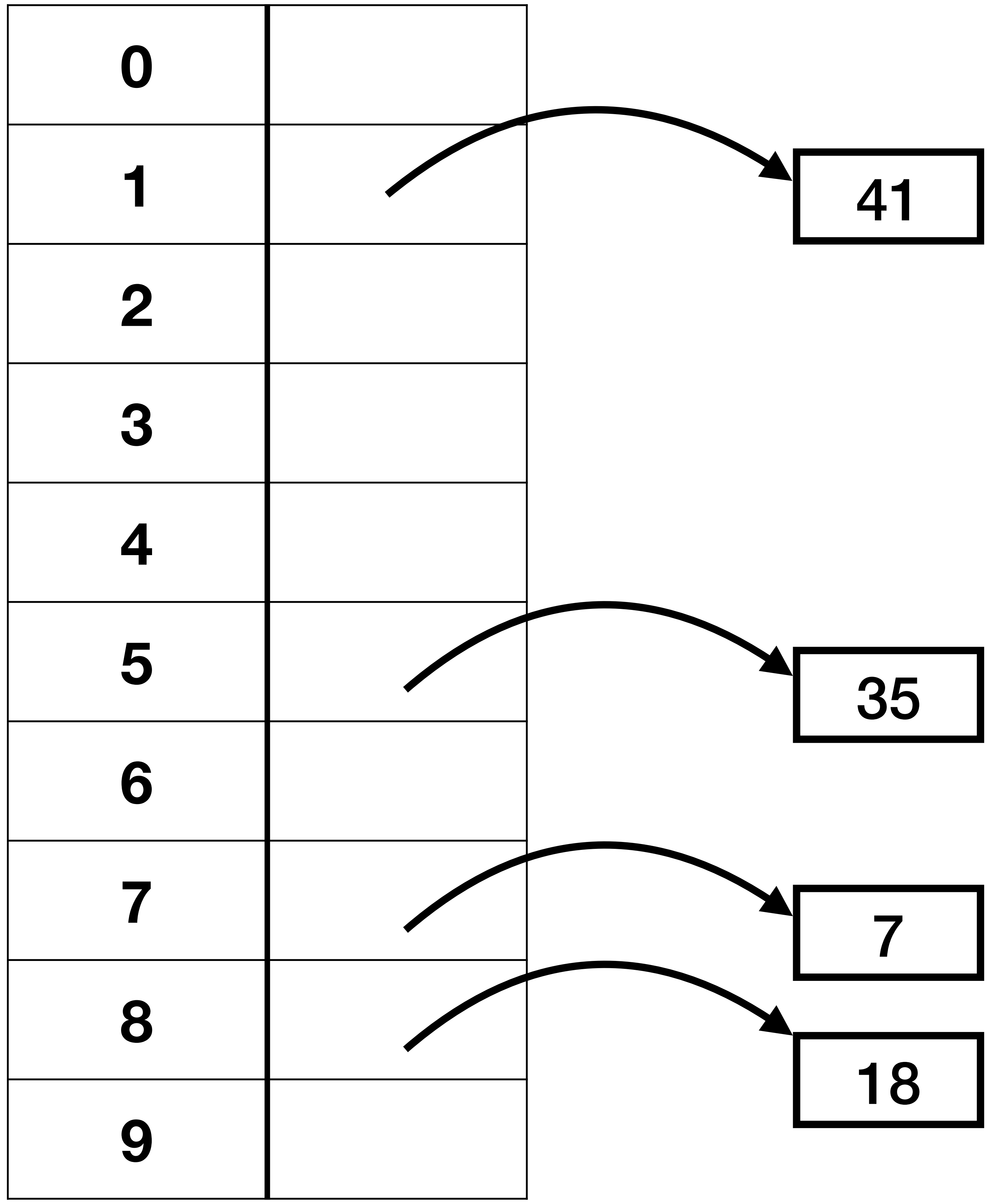



```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];   <<-- what is this?
};

struct bucket {
        void *key;
        void *value;
        struct bucket *next;
```

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*



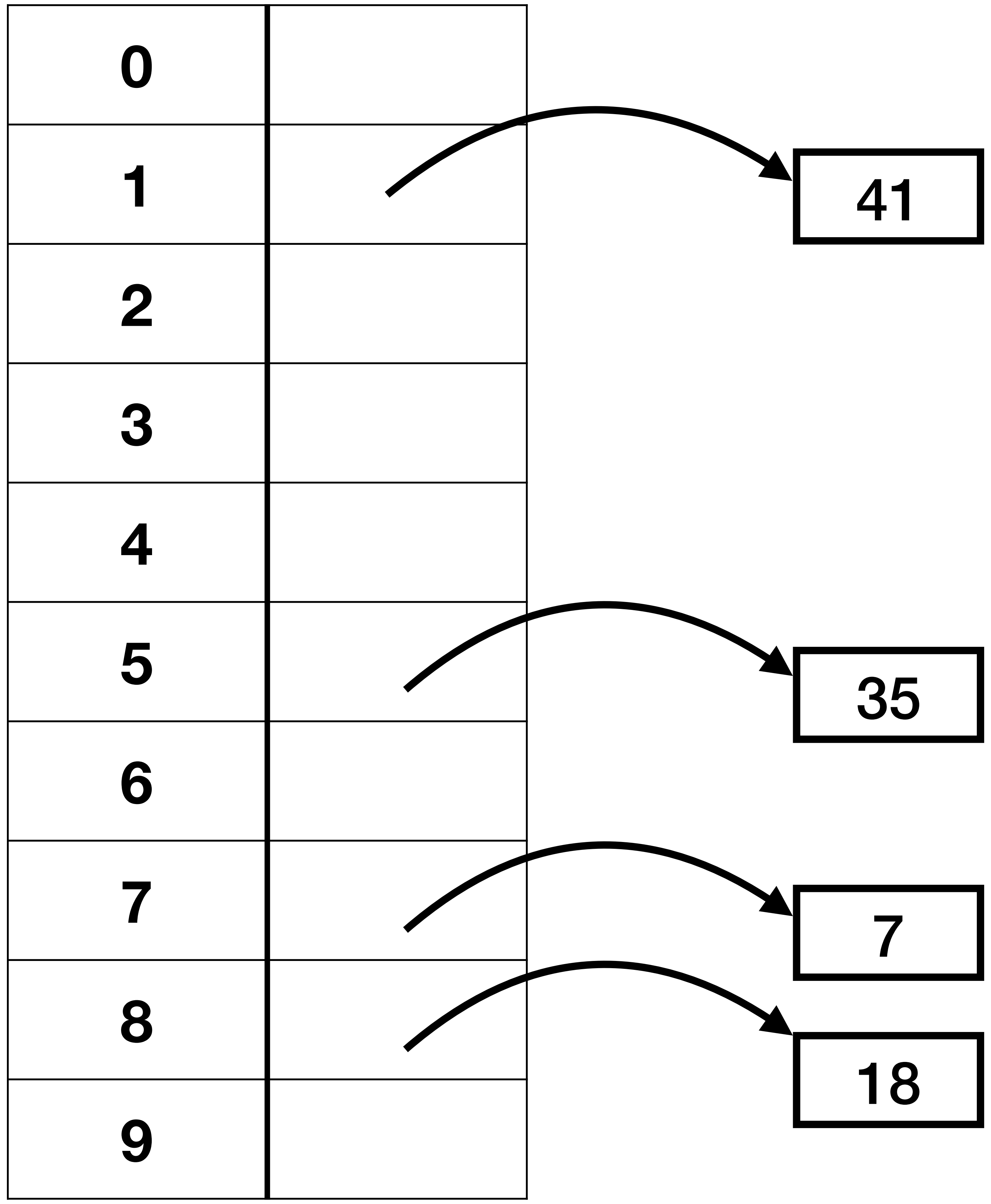

```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];
};

struct bucket {
        void *key;
        void *value;
        struct bucket *next;
};
```

<<-- what is this?

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*




```
struct table {
        int size;
        int length;
        int (*eq)(void *, void *);
        uint64_t (*hash)(void *);
        struct bucket *buckets[];    <<-- what is this?
};

struct bucket {
        void *key;
        void *value;
        struct bucket *next;
};
```

Interlude

Flexible array member

- The last element of a structure may have an incomplete array type (empty bracket)
- `sizeof` does not include the incomplete field
- Why?

Interlude: Flexible Array Member

Memory Layout

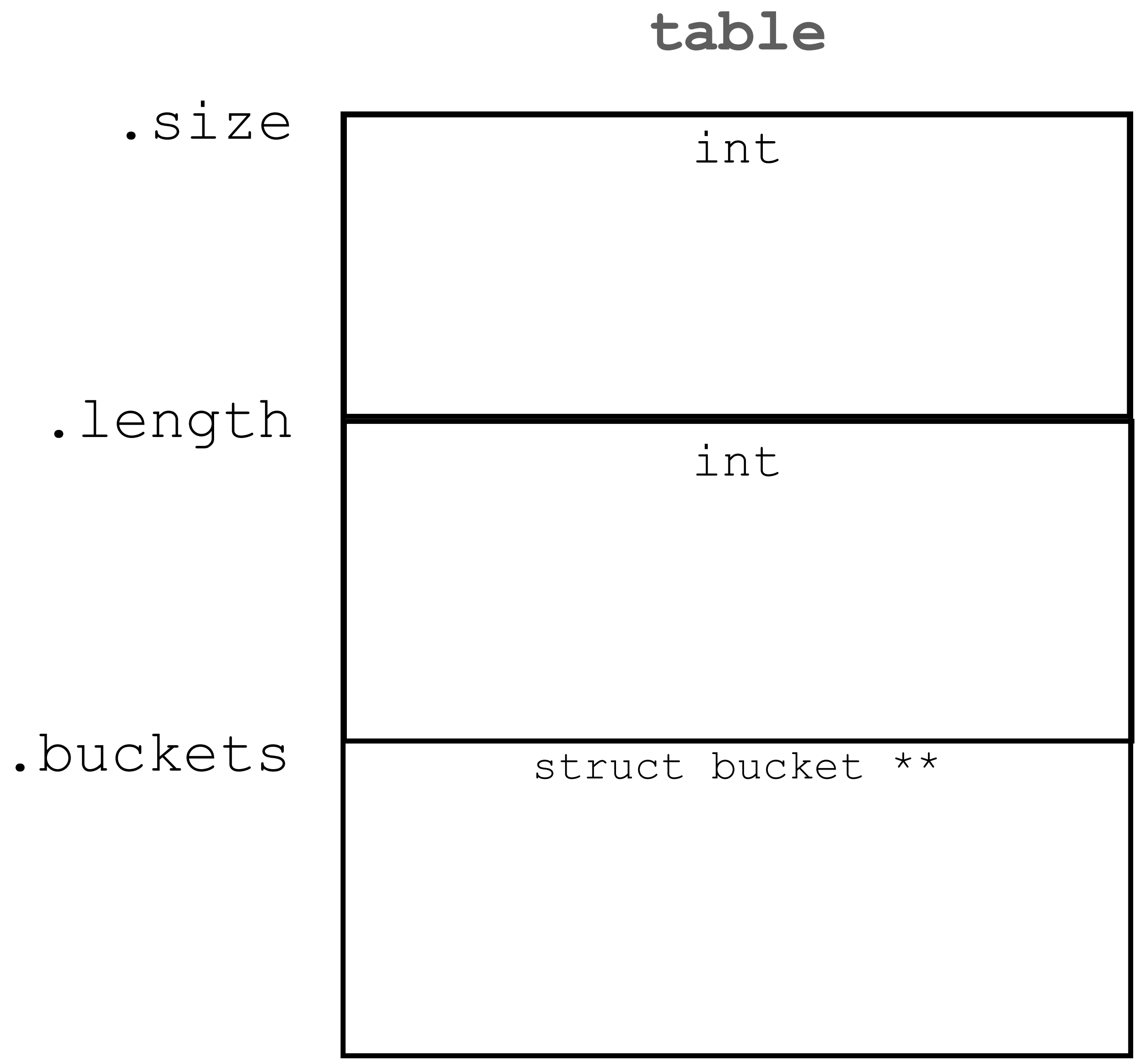
```
struct table {
        int size;
        int length;
        struct bucket **buckets;
};
```

```
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};
```

Interlude: Flexible Array Member

Memory Layout

```c
struct table {
        int size;
        int length;
        struct bucket **buckets;
};
```

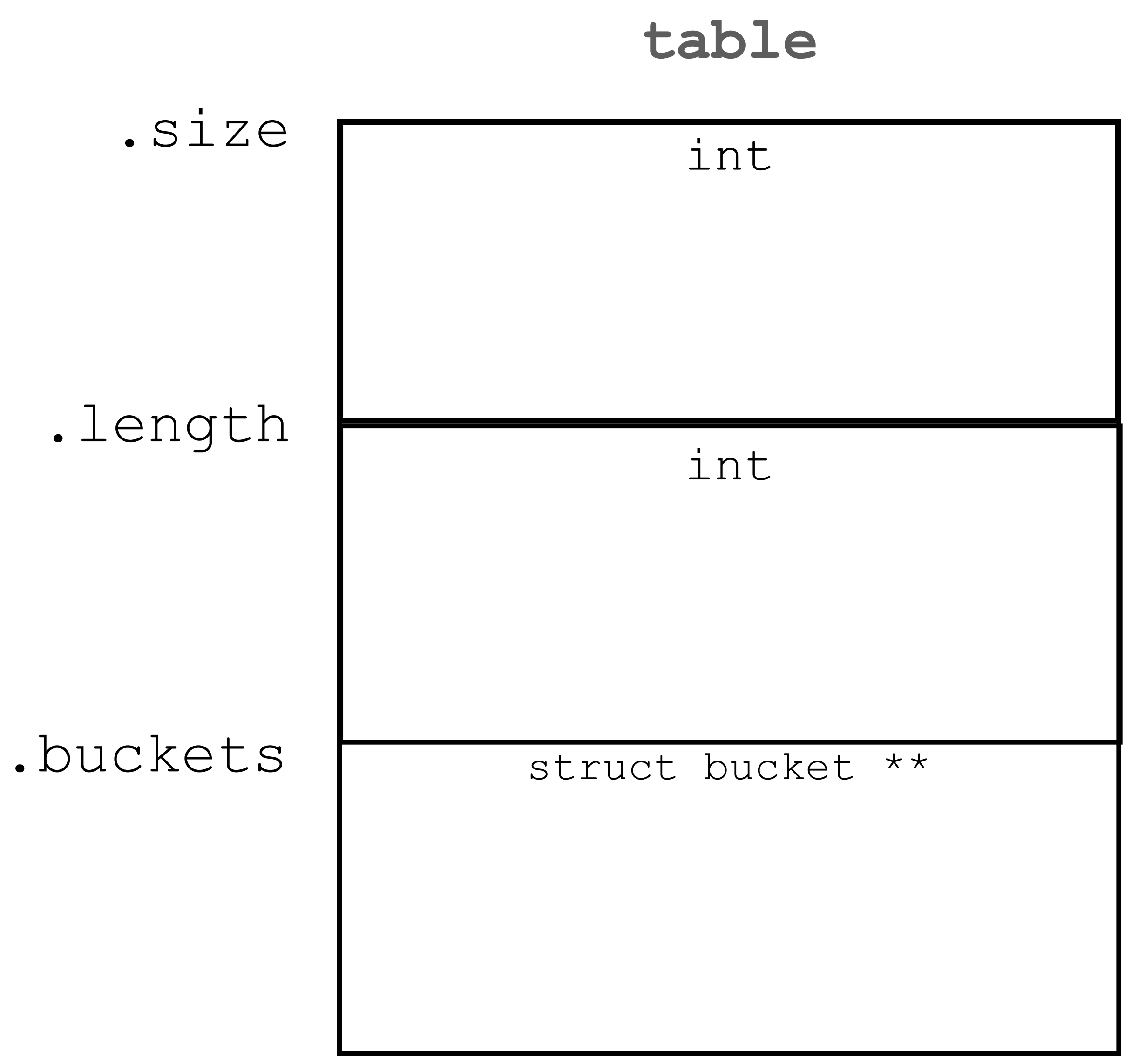



```c
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};
```

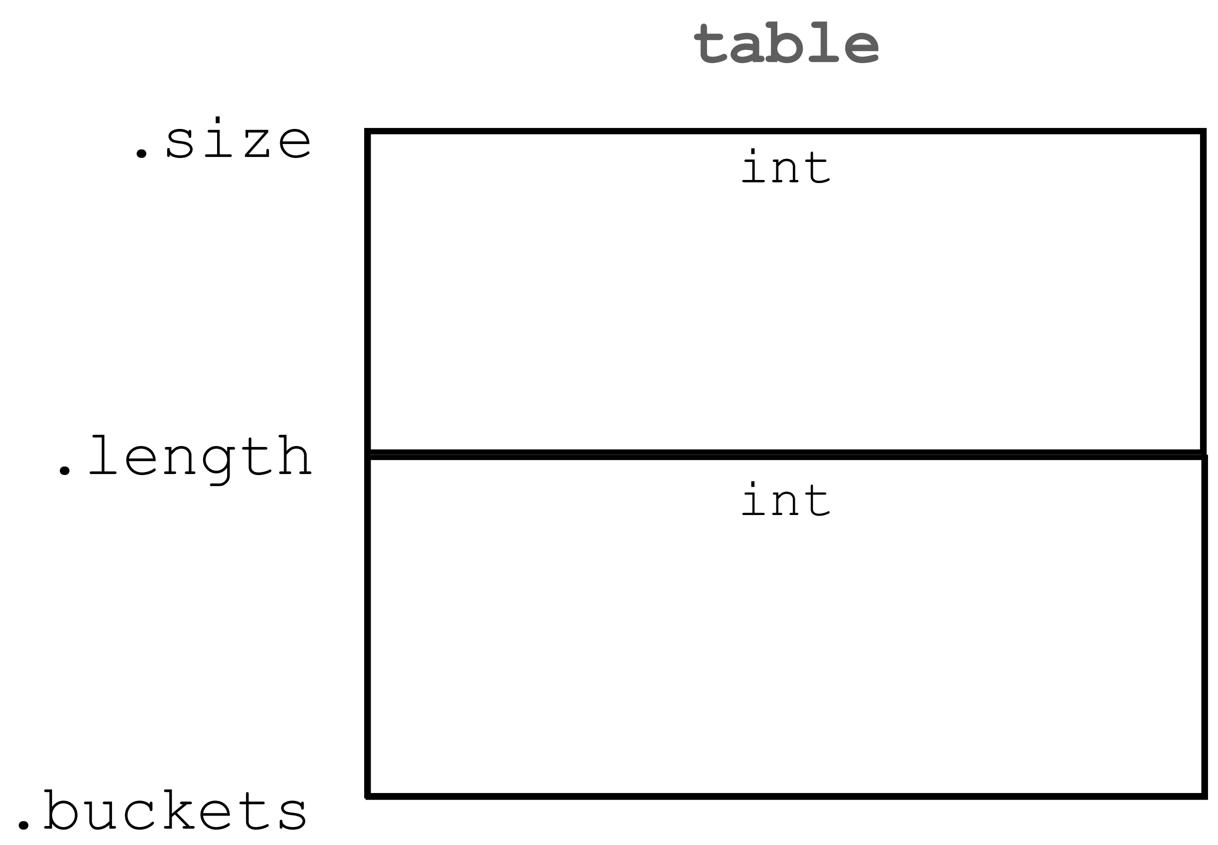
Interlude: Flexible Array Member

Memory Layout

```c
struct table {
        int size;
        int length;
        struct bucket **buckets;
};
```



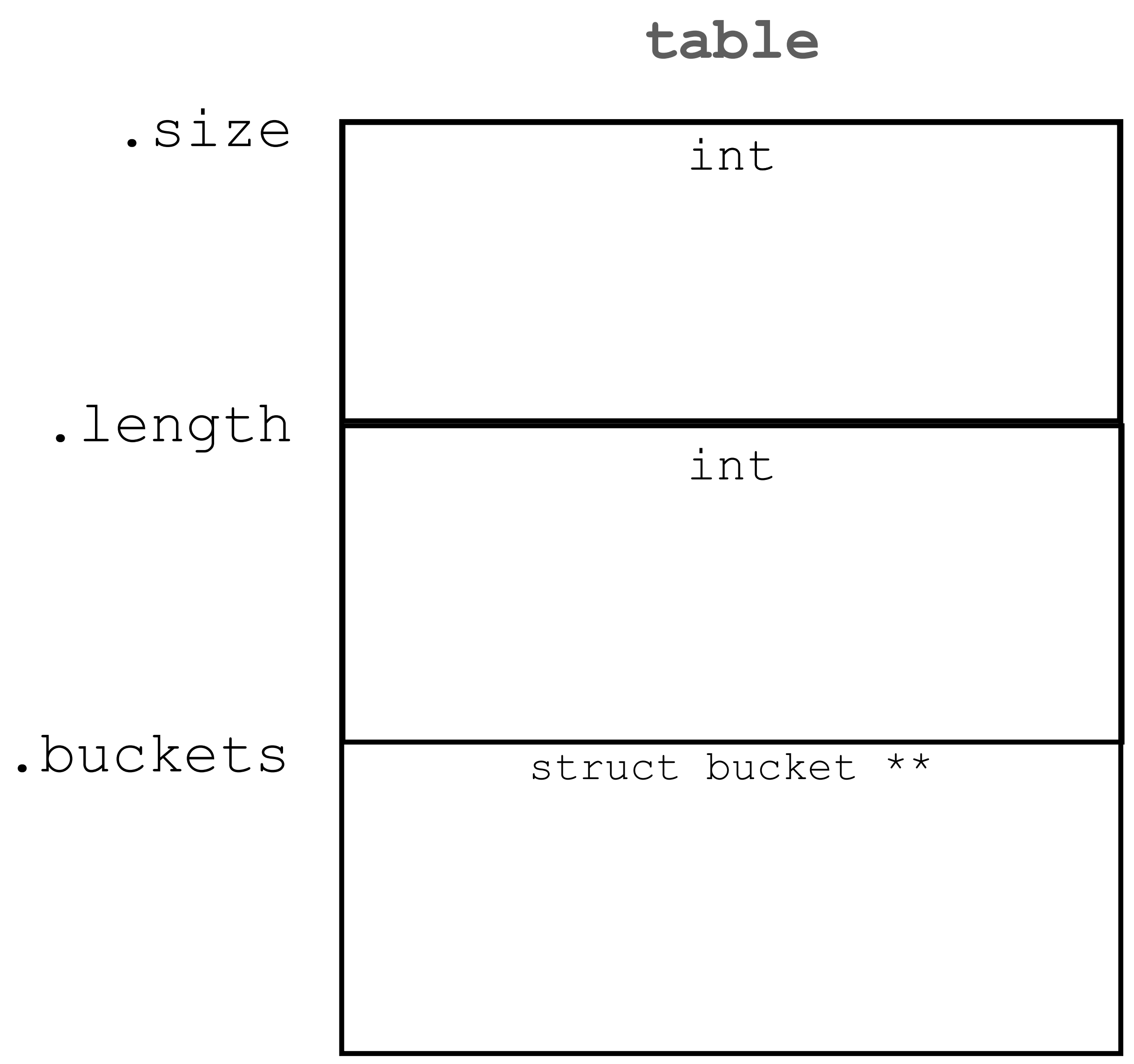

```c
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};
```



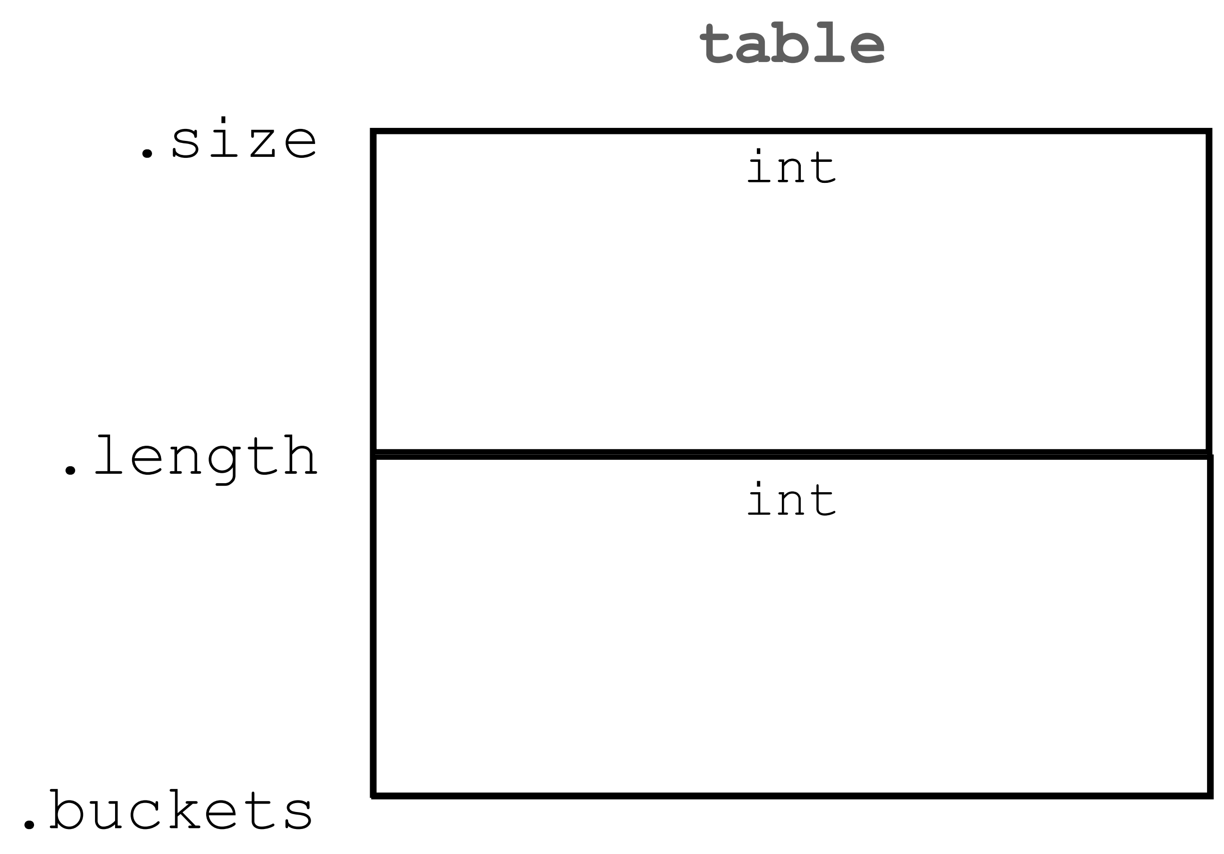
Interlude: Flexible Array Member

Memory Layout

```
struct table {
        int size;
        int length;
        struct bucket **buckets;
};
```



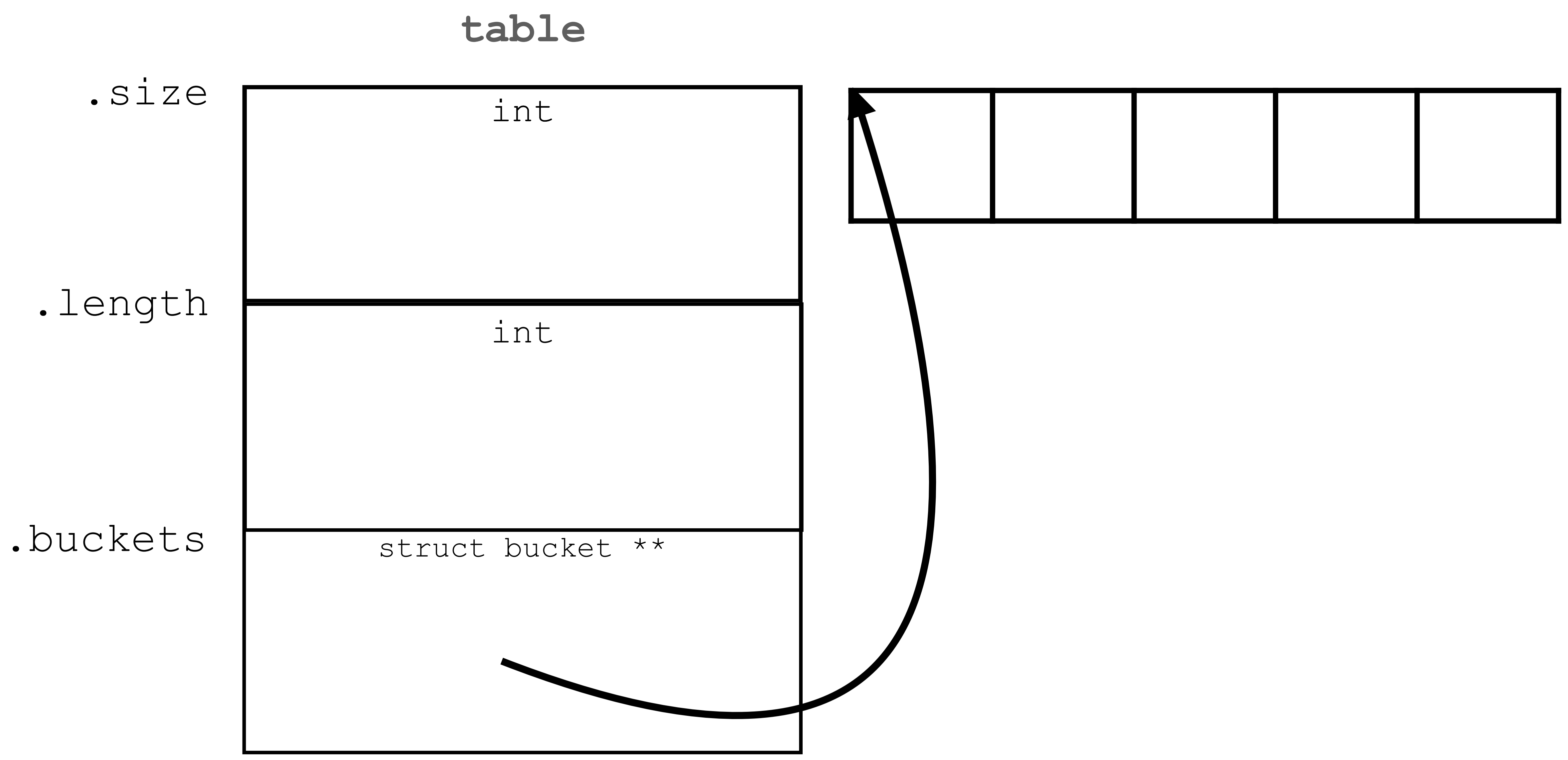

```
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};
```



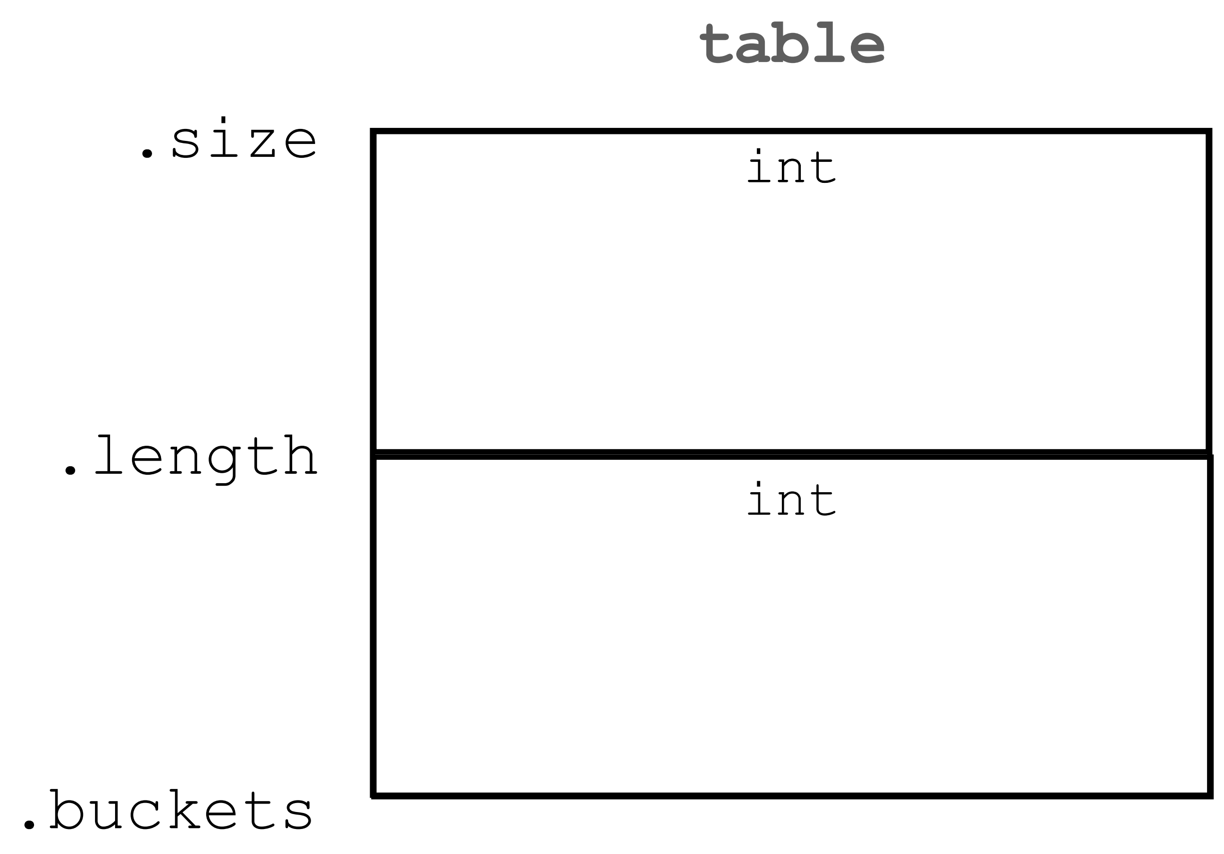
Interlude: Flexible Array Member

Memory Layout

```
struct table {
        int size;
        int length;
        struct bucket **buckets;
};
```



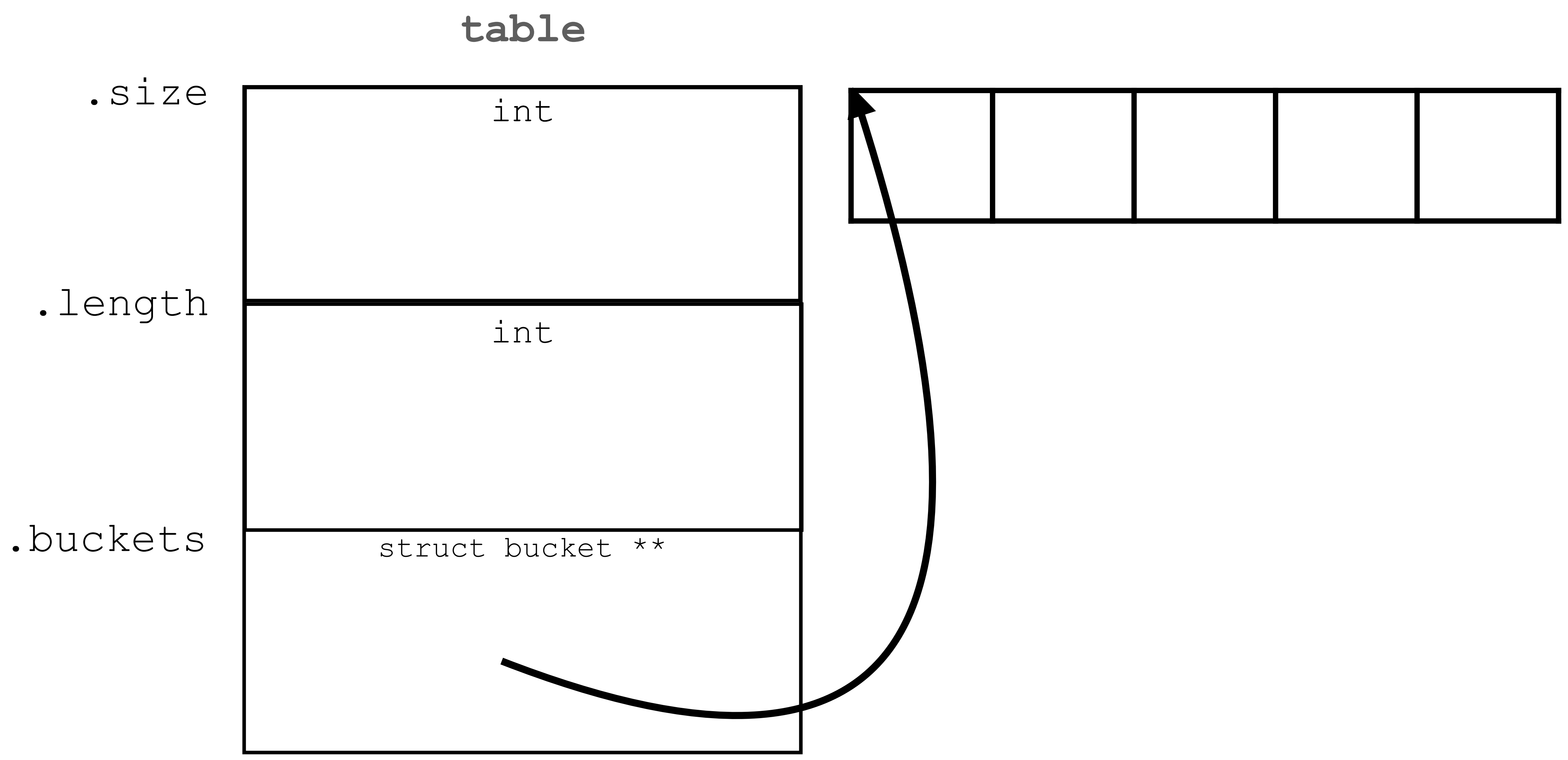

```
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};
```



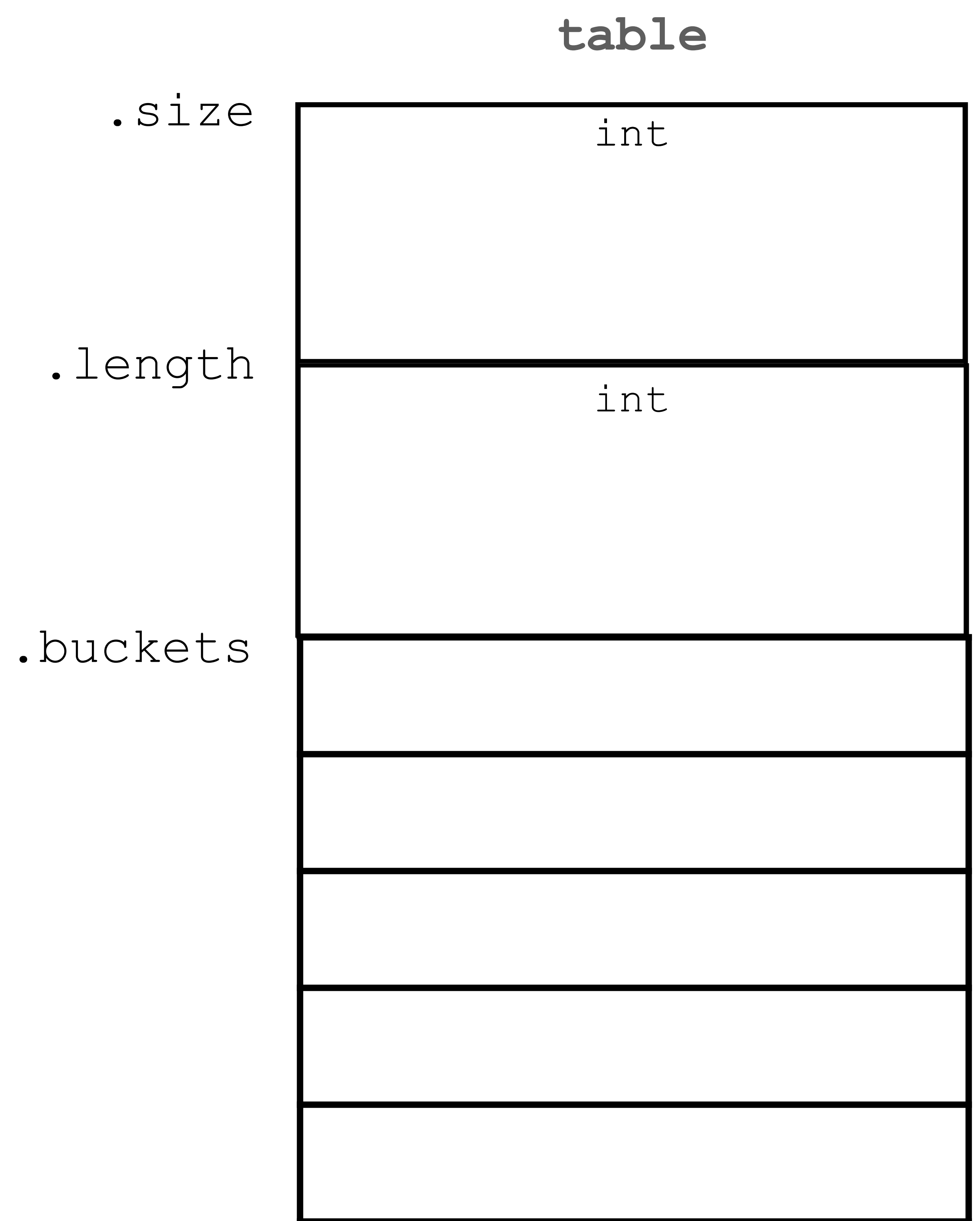
Interlude: Flexible Array Member

Memory Layout

```c
struct table {
        int size;
        int length;
        struct bucket **buckets;
};
```




```c
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};
```



Interlude: Flexible Array Member

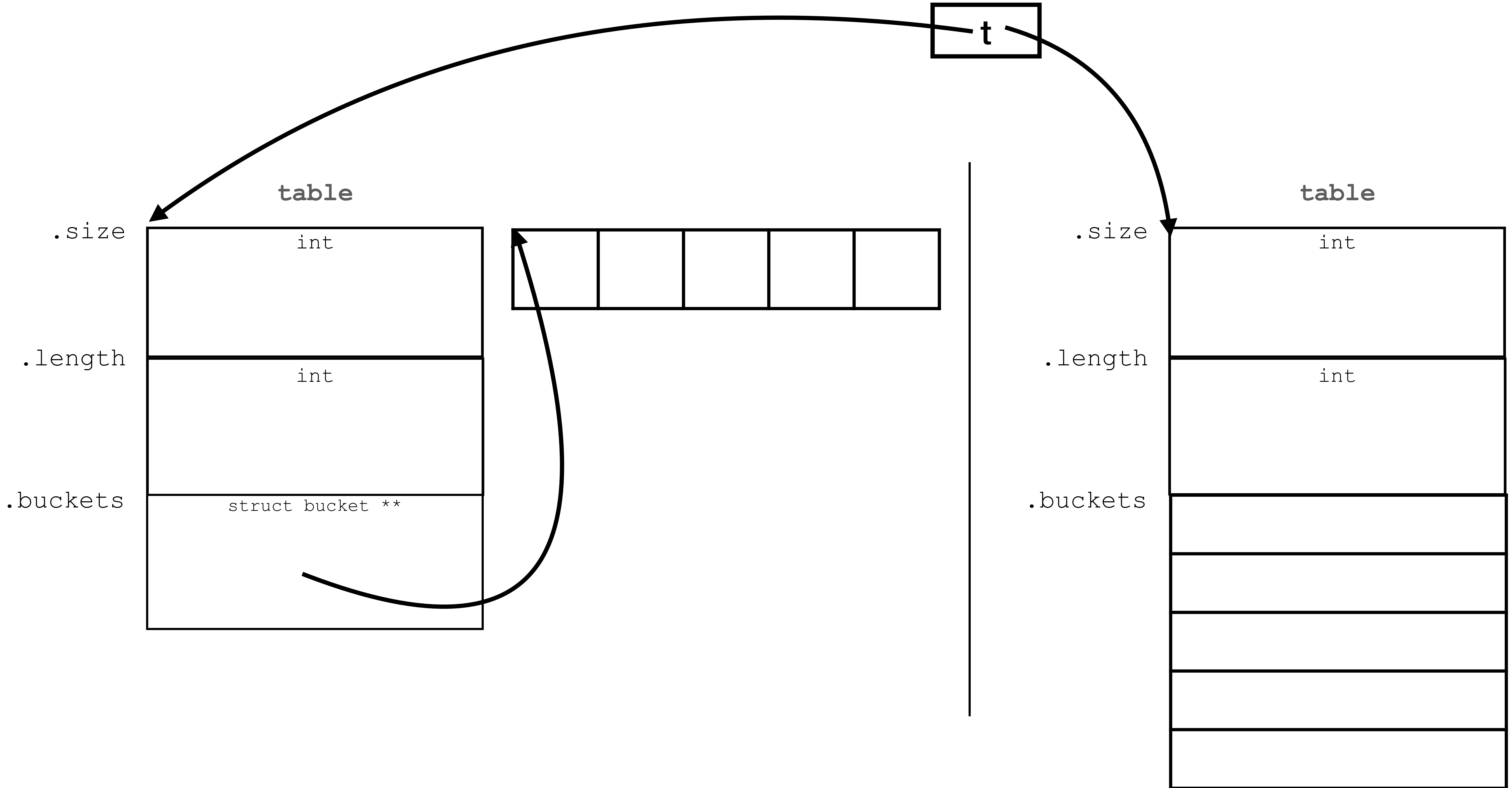
Allocation

```c
struct table {
        int size;
        int length;
        struct bucket **buckets;
};

int size = 1024;

struct table *t =
  malloc(sizeof(struct table));

t->buckets =
  malloc(size * sizeof(struct bucket*));
```

```c
struct table {
        int size;
        int length;
        struct bucket *buckets[];
};

int size = 1024;

struct table *t =
  malloc(sizeof(struct table)
         + size * sizeof(struct bucket*));
```

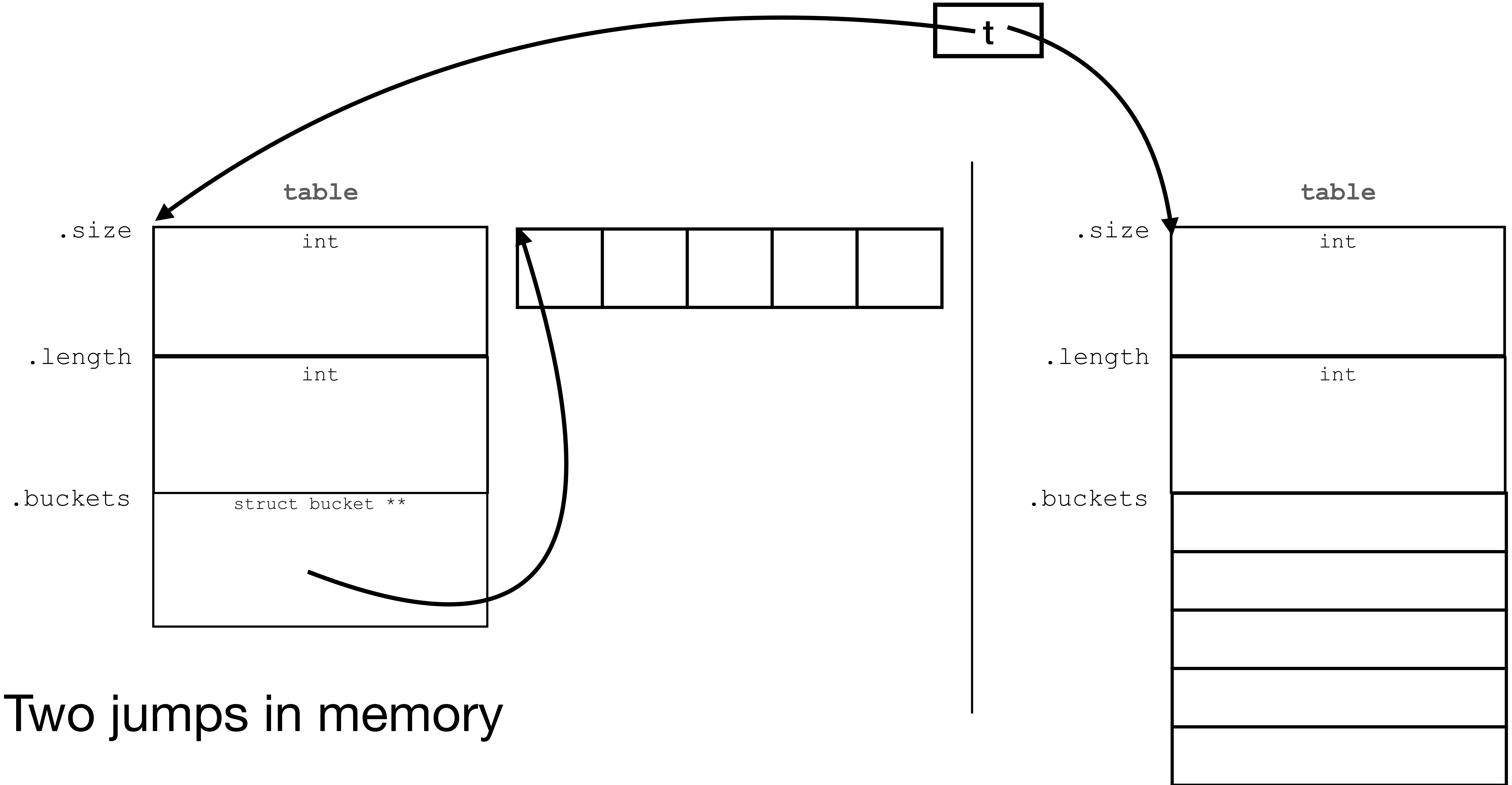
Interlude: Flexible Array Member

Accessing `t->buckets[3];`



Interlude: Flexible Array Member

Accessing `t->buckets[3];`



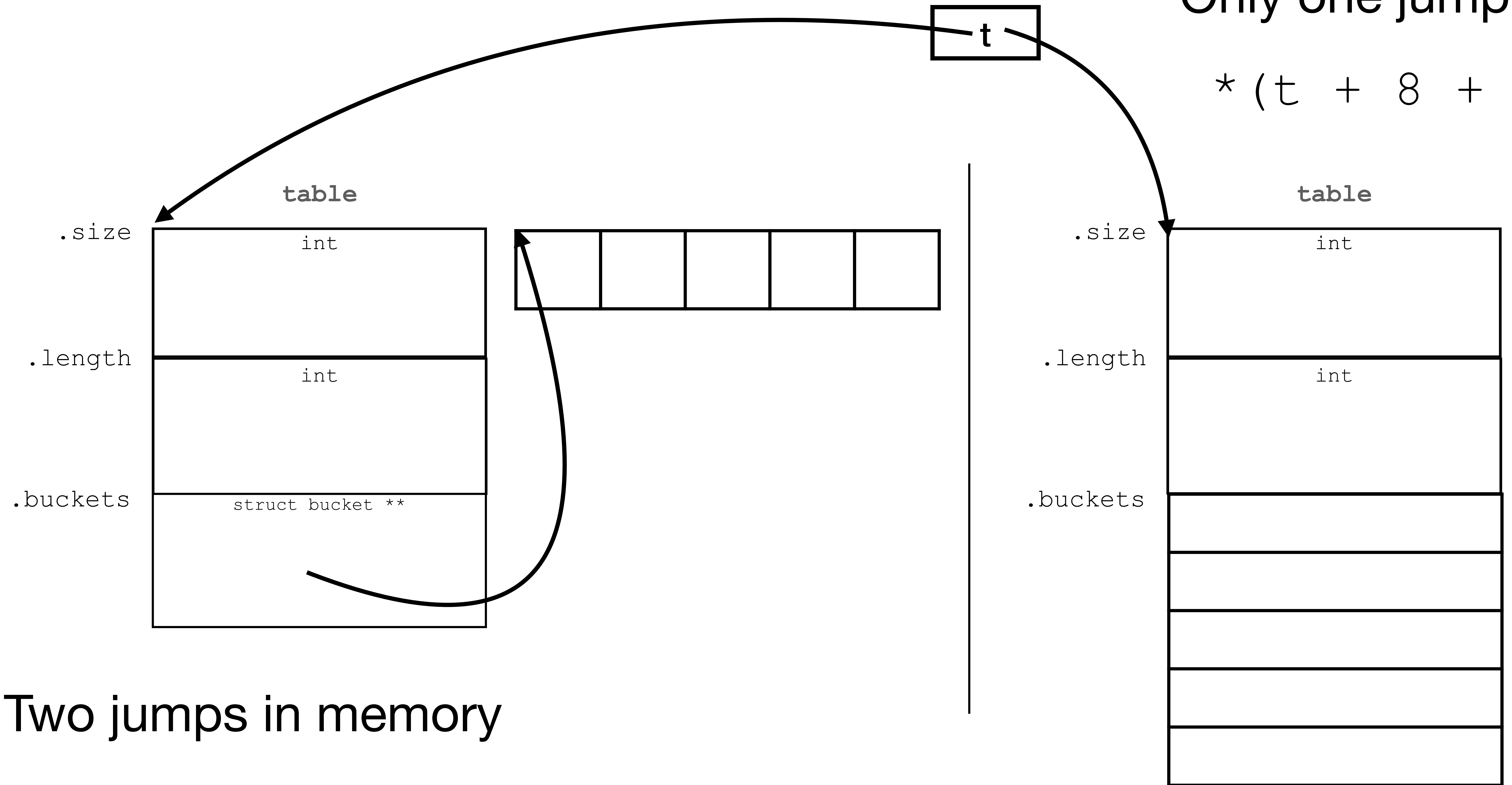
Two jumps in memory

Interlude: Flexible Array Member

Accessing `t->buckets[3];`

Only one jump!

`*(t + 8 + i * size)`



Two jumps in memory

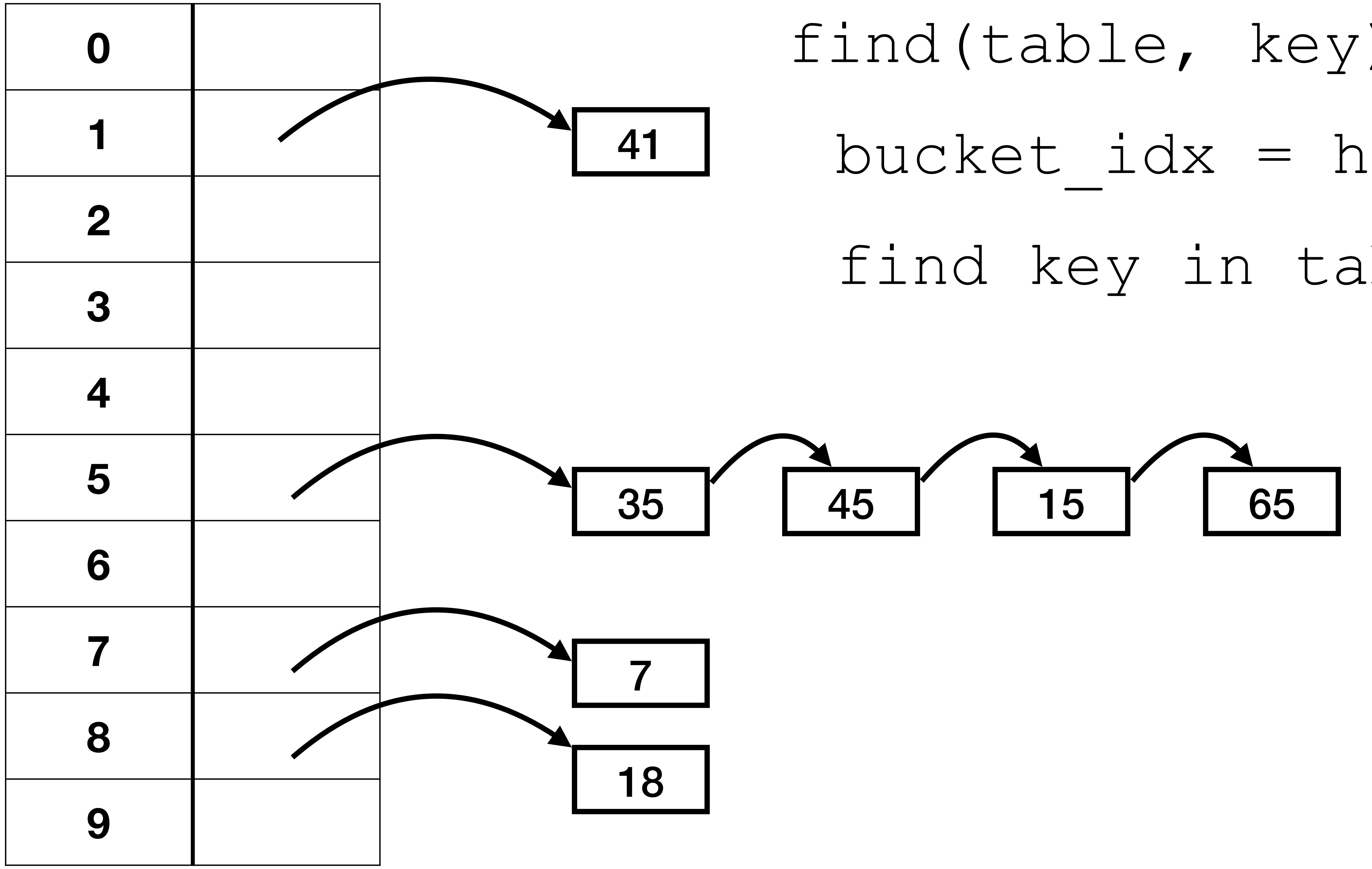
Interlude: Flexible Array Member

- The last element of a structure may have an incomplete array type (empty bracket)
- `sizeof` does not include the incomplete field
- `struct table *ptr = malloc(sizeof(struct table) + extra);`
- Slight performance boost

Chaining

Insert

- Each slot is a *list* of key-value pairs, called a *bucket*




```
find(table, key):
 bucket_idx = hash(key) % table->size
 find key in table->buckets[bucket_idx]:
```

Chaining

Time Complexity

Chaining

Time Complexity

- What is complexity for accessing elements?

Chaining

Time Complexity

- What is complexity for accessing elements?
  - $O(\text{length of the chain})$

Chaining

Time Complexity

- What is complexity for accessing elements?
  - $O(\text{length of the chain})$
- What is the length of the chain in the worst case?

Chaining

Time Complexity

- What is complexity for accessing elements?
  - O(length of the chain)
- What is the length of the chain in the worst case?
- $O(n)$

Chaining

Time Complexity

- What is complexity for accessing elements?
    - O(length of the chain)
- What is the length of the chain in the worst case?
- $O(n)$
    - This happens for a really bad hash function (e.g. $hash(k) = 1$)

Chaining

Time Complexity

- What is complexity for accessing elements?
    - $O($length of the chain$)$
- What is the length of the chain in the worst case?
- $O(n)$
    - This happens for a really bad hash function (e.g. $hash(k) = 1$)
- What if we have a good hash function (that has uniform distribution over a range of integers)?

Chaining

Time Complexity

- What is complexity for accessing elements?
    - O(length of the chain)
- What is the length of the chain in the worst case?
- $O(n)$
    - This happens for a really bad hash function (e.g. $hash(k) = 1$)
- What if we have a good hash function (that has uniform distribution over a range of integers)?
    - What is the average (expected) length of a chain?

Chaining

Time Complexity

- What is complexity for accessing elements?
  - O(length of the chain)
- What is the length of the chain in the worst case?
- $O(n)$
  - This happens for a really bad hash function (e.g. $hash(k) = 1$)
- What if we have a good hash function (that has uniform distribution over a range of integers)?
  - What is the average (expected) length of a chain?
  - $O\left(\frac{\#\text{elements}}{\#\text{buckets}}\right)$: this ratio is called *load factor*.

Chaining

Time Complexity

Chaining

Time Complexity

- In practice, hash tables are very fast

Chaining

Time Complexity

- In practice, hash tables are very fast
  - Typically faster than BSTs

Chaining

Time Complexity

- In practice, hash tables are very fast
  - Typically faster than BSTs
- Especially we can keep the load factor $O(1)$

Chaining

Time Complexity

- In practice, hash tables are very fast
  - Typically faster than BSTs
- Especially we can keep the load factor $O(1)$
  - Analysis deferred to algorithms

Hash Table

Handling Collision

- Two approaches:
  1. ~~Chaining: put a list in each bucket~~
  2. Probing: use spare space in the array

Probing

- If the bucket is occupied, use the next one.

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

75

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

Probing

- If the bucket is occupied, use the next one.
  - Wrap around when reaching the end of array

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

- Wrap around when reaching the end of array
- The table *must* have some extra space, i.e. load factor has to be ≤ 1

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

- Wrap around when reaching the end of array
- The table *must* have some extra space, i.e. load factor has to be ≤ 1
- Many flavors of "next one":

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

- Wrap around when reaching the end of array
- The table *must* have some extra space, i.e. load factor has to be ≤ 1
- Many flavors of "next one":
  - *Linear probing*: +1 at a time

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

- Wrap around when reaching the end of array
- The table *must* have some extra space, i.e. load factor has to be ≤ 1
- Many flavors of "next one":
  - *Linear probing*: +1 at a time
  - *Quadratic probing*: * 2 at a time

Probing

- If the bucket is occupied, use the next one.

0	
1	41
2	
3	
4	
5	35
6	75
7	7
8	18
9	

- Wrap around when reaching the end of array
- The table *must* have some extra space, i.e. load factor has to be ≤ 1
- Many flavors of "next one":
  - *Linear probing*: +1 at a time
  - *Quadratic probing*: * 2 at a time
  - ...

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- insert("alice", 400)

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	
4	
5	("alice", 400)
6	
7	
8	
9	

- insert("bob", 30)

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	
5	("alice", 400)
6	
7	
8	
9	

- insert("carl", 50)

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	("alice", 400)
6	
7	
8	
9	

- insert("eve", 100)

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	("alice", 400)
6	("eve", 100)
7	("david", 60)
8	
9	

- insert("david", 60)

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	("alice", 400)
6	("eve", 100)
7	("david", 60)
8	
9	

- find("eve")
  - Go to 3 bucket
  - Move down until we find "eve" or until we hit empty bucket
- return 100

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	("alice", 400)
6	("eve", 100)
7	("david", 60)
8	
9	

- find("karl")
  - Go to 4 bucket
  - Move down until we find "karl" or until we hit empty bucket
- No "karl" in table

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	("alice", 400)
6	("eve", 100)
7	("david", 60)
8	
9	

- remove("alice")
  - Go to 5
  - Move down until we find "alice"

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- remove("alice")
  - Go to 5
  - Move down until we find "alice"

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find("eve")
  - Go to 3
  - How far do we move down?

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- When we removed "alice" we left a hole

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- When we removed "alice" we left a hole
- When searching for "eve" if we stop at the hole, we won't find "eve"

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- When we removed "alice" we left a hole
- When searching for "eve" if we stop at the hole, we won't find "eve"
- But if we don't stop at empty spots, we have to search through the entire array if a key doesn't exist

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- A bucket can be in one of three states:

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- A bucket can be in one of three states:
    - Occupied (key != NULL)

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- A bucket can be in one of three states:
  - Occupied (key != NULL)
  - Empty, but was always empty

Probing

Linear probing (example)

```
struct bucket {
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- A bucket can be in one of three states:
  - Occupied (key != NULL)
  - Empty, but was always empty
  - Empty, but previously occupied

Probing

Linear probing (example)

true when previously occupied

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- A bucket can be in one of three states:
  - Occupied (key != NULL)
  - Empty, but was always empty
  - Empty, but previously occupied

Probing

Linear probing (example)

true when previously occupied

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	REMOVED
6	("eve", 100)
7	("david", 60)
8	
9	

- Find("eve")
  - Go to 3
  - Move down until we find "eve", or until we hit an empty, non-removed bucket

Probing

Linear probing (example)

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

true when previously occupied

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find("eve")
  - Go to 3
  - Move down until we find "eve", or until we hit an empty, non-removed bucket
- This empty but removed bucket is sometimes called a *tombstone*

Probing

Linear probing

true when
previously occupied

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

Probing

Linear probing

true when previously occupied

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

true when previously occupied

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
  - Move down until first empty bucket

Probing

Linear probing

```
struct bucket {
    bool removed;
    void *key;
    void *value;
};
```

true when previously occupied

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
  - Move down until first empty bucket
  - If tombstone is encountered, continue searching

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

true when previously occupied

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
  - Move down until first empty bucket
  - If tombstone is encountered, continue searching
- Insert:

Probing

Linear probing

true when
previously occupied

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
  - Move down until first empty bucket
  - If tombstone is encountered, continue searching
- Insert:
  - Move down until first empty bucket

Probing

Linear probing

```
struct bucket {
    bool removed;
    void *key;
    void *value;
};
```

true when previously occupied

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
    - Move down until first empty bucket
    - If tombstone is encountered, continue searching
- Insert:
    - Move down until first empty bucket
    - If tombstone is encountered, we can reuse that bucket

Probing

Linear probing

true when previously occupied

```
struct bucket {
    bool removed;
    void *key;
    void *value;
};
```

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
  - Move down until first empty bucket
  - If tombstone is encountered, continue searching
- Insert:
  - Move down until first empty bucket
  - If tombstone is encountered, we can reuse that bucket
  - But to avoid inserting duplicate keys, we need to continue searching until an unremoved bucket

Probing

Linear probing

```c
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

true when previously occupied

- Let's use `strlen` as our (bad) hash function

0	
1	
2	
3	("bob", 30)
4	("carl", 50)
5	
6	("eve", 100)
7	("david", 60)
8	
9	

- Find/Remove:
  - Move down until first empty bucket
  - If tombstone is encountered, continue searching
- Insert:
  - Move down until first empty bucket
  - If tombstone is encountered, we can reuse that bucket
  - But to avoid inserting duplicate keys, we need to continue searching until an unremoved bucket

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- This is why a good hash function spreads out outputs

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- This is why a good hash function spreads out outputs
- If the hash function maps similar inputs to similar outputs, e.g. strlen, we would get clusters in the hash table.

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- This is why a good hash function spreads out outputs
- If the hash function maps similar inputs to similar outputs, e.g. strlen, we would get clusters in the hash table.
    - Really bad for probing

Probing

Linear probing

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- This is why a good hash function spreads out outputs
- If the hash function maps similar inputs to similar outputs, e.g. strlen, we would get clusters in the hash table.
    - Really bad for probing
    - Clusters mean we need to go through more buckets

Probing
Time Complexity

```
struct bucket {
	bool removed;
	void *key;
	void *value;
};
```

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$
- What is the worst case complexity when using probing?

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$
- What is the worst case complexity when using probing?
  - Insertion: $O(n)$

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$
- What is the worst case complexity when using probing?
  - Insertion: $O(n)$
    - Worst case: all elements are in one cluster, need to go through all to find unfilled bucket

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$
- What is the worst case complexity when using probing?
  - Insertion: $O(n)$
    - Worst case: all elements are in one cluster, need to go through all to find unfilled bucket
  - Get: $O(\text{table\_size})$

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$
- What is the worst case complexity when using probing?
  - Insertion: $O(n)$
    - Worst case: all elements are in one cluster, need to go through all to find unfilled bucket
  - Get: $O(\text{table\_size})$
    - Worst case: all empty buckets are tombstones

Probing

Time Complexity

```
struct bucket {
        bool removed;
        void *key;
        void *value;
};
```

- Chaining: worst $O(n)$, average $O(1)$
- What is the worst case complexity when using probing?
  - Insertion: $O(n)$
    - Worst case: all elements are in one cluster, need to go through all to find unfilled bucket
  - Get: $O(\text{table\_size})$
    - Worst case: all empty buckets are tombstones
- On average, the number of probes is at most $1/(1 - \text{load factor})$

Probing

Time Complexity (Appendix)

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots
  - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
    - $\Pr[A_1] = n/m$, assuming n elements and m slots
    - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing
    - $\Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1}$ = load factor^{i-1}

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots
  - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing
  - $\Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1}$ = load factor^{i-1}
- $E[\text{\#probes}] = \sum_{i=1}^{\infty} \Pr[A_1 \cap \ldots \cap A_{i-1}]$

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots
  - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing
  - $\Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1}$ = load factor^{i-1}
- $E[\text{\#probes}] = \sum_{i=1}^{\infty} \Pr[A_1 \cap \ldots \cap A_{i-1}]$
- $\leq \sum_{i=1}^{\infty}$ load factor^{i-1}

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots
  - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing
  - $\Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1}$ = load factor^{i-1}
- $E[\text{\#probes}] = \sum_{i=1}^{\infty} \Pr[A_1 \cap \ldots \cap A_{i-1}]$
- $\leq \sum_{i=1}^{\infty}$ load factor^{i-1}
- $= \sum_{i=0}^{\infty}$ load factor^{i}

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots
  - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing
  - $\Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1}$ = load factor^{i-1}
- $E[\text{\#probes}] = \sum_{i=1}^{\infty} \Pr[A_1 \cap \ldots \cap A_{i-1}]$
- $\leq \sum_{i=1}^{\infty} \text{load factor}^{i-1}$
- $= \sum_{i=0}^{\infty} \text{load factor}^{i}$
- $= \frac{1}{1 - \text{load factor}}$

Probing

Time Complexity (Appendix)

- Let A_i be the event that the ith probe is occupied.
  - $\Pr[A_1] = n/m$, assuming n elements and m slots
  - $\Pr[A_2] = (n-1)/(m-1)$, since $n-1$ elements and $m-1$ slots are remaining, assuming uniform hashing
  - $\Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1}$ = load factor^{i-1}
- $E[\text{\#probes}] = \sum_{i=1}^{\infty} \Pr[A_1 \cap \ldots \cap A_{i-1}]$
- $\leq \sum_{i=1}^{\infty} \text{load factor}^{i-1}$
- $= \sum_{i=0}^{\infty} \text{load factor}^{i}$
- $= \frac{1}{1 - \text{load factor}}$
- E.g. if the table is half full, the average number of probes is 1 / (1 - 0.5) = 2

Load Factor

Notes

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$
- Systems typically keep load factor around 0.7 to 0.75

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$
- Systems typically keep load factor around 0.7 to 0.75
  - This is determined through experimentation

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$
- Systems typically keep load factor around 0.7 to 0.75
  - This is determined through experimentation
  - Space vs. time trade-off

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$
- Systems typically keep load factor around 0.7 to 0.75
  - This is determined through experimentation
  - Space vs. time trade-off
- What should we do when we hit the maximum load factor?

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$
- Systems typically keep load factor around 0.7 to 0.75
  - This is determined through experimentation
  - Space vs. time trade-off
- What should we do when we hit the maximum load factor?
  - Increase the # of buckets

Load Factor

Notes

- Keep load factor $O(1)$ makes all operations $O(1)$
- Systems typically keep load factor around 0.7 to 0.75
  - This is determined through experimentation
  - Space vs. time trade-off
- What should we do when we hit the maximum load factor?
  - Increase the # of buckets
  - Can we just realloc? I.e. put the same elements in the same buckets after expansion?

Maps

Complexity

	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList	O(n)		O(1)	O(n)	O(1)	
Linked List	O(n)		O(1)		O(1)	
ArrayList (sorted)	O(log n)		O(n)		O(n)	
Linked List (sorted)	O(n)		O(1)		O(1)	
BST	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)
Hash Table	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)

Hash Table

Epilogue

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?
- Operations that involve comparisons:

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?
- Operations that involve comparisons:
    - find_min and find_max

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?
- Operations that involve comparisons:
  - find_min and find_max
  - range look up: give me 10 < key < 20

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?
- Operations that involve comparisons:
    - find_min and find_max
    - range look up: give me 10 < key < 20
    - Better to use a heap or BST for these

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?
- Operations that involve comparisons:
    - find_min and find_max
    - range look up: give me 10 < key < 20
    - Better to use a heap or BST for these
- Operations that involve ordering, insert "front" and "back"

Hash Table

Epilogue

- Hash tables are excellent at insertion, removal, and looking up. What operations are they bad at?
- Operations that involve comparisons:
  - find_min and find_max
  - range look up: give me 10 < key < 20
  - Better to use a heap or BST for these
- Operations that involve ordering, insert "front" and "back"
  - Hash tables have no notion of "order" -- in C++, hash tables are called unordered_map

Hash Table

In one slide

- Array access is $O(1)$.
- Using arbitrary keys as array indices:
  - Hash functions turn any values into an integer. Ideally, this should be uniform.
  - Compress function forces integers into [0, table_size).
- Handling Collision:
  - Chaining: put a list in each bucket
  - Probing: use spare space in the array
- Load factor: the expected number of elements to go through
  - #elements / #buckets
  - Chaining: load factor has no limit; probing: load factor at most 1
  - Adjusting #buckets to keep load factor (0.7 - 0.75) -- time/space trade-off

Data Structures

- Establishing structures on the heap:
  - Indices: contiguous
    - $O(1)$ random access
    - difficult to reorder and reallocate
  - Pointer: scattered
    - sequential access
    - easy to reorder and reallocate

	Indices	**Pointers**
List	Array List	Linked List
Map	Hash Table	BST