Database Processing
CS 451 / 551

Lecture 6:
Hashing

Suyash Gupta
Assistant Professor
Distopia Labs and ORNG
Dept. of Computer Science
(E) suyash@uoregon.edu
(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Assignment 1 is Out!

Deadline: Oct 28, 2025 at 11:59pm

Start collaborating with your groups!

Term Paper for Graduate Students

- **Select one area**.
- **Select one paper published** in 2025 from the following 4 conferences:
  - **No two students can select the same paper**.
  - Your selected paper **needs my approval**.

- VLDB, SIGMOD, OSDI, SOSP.
- Describe the following in 4-page style ACM Sigmod double-column style.
  - What is the paper's goal?
  - How is it meeting its goal?
  - What are the disadvantages of the proposed design and advantages of the proposed design?
  - Explain how can you improve the proposed design?
  - What architectural changes you need to do?
  - How to provide support for queries, say Natural Join?
- Topics:
  - Federated Learning, Vector Databases, Graph Databases, Privacy-Preserving Databases

Unordered Indexing

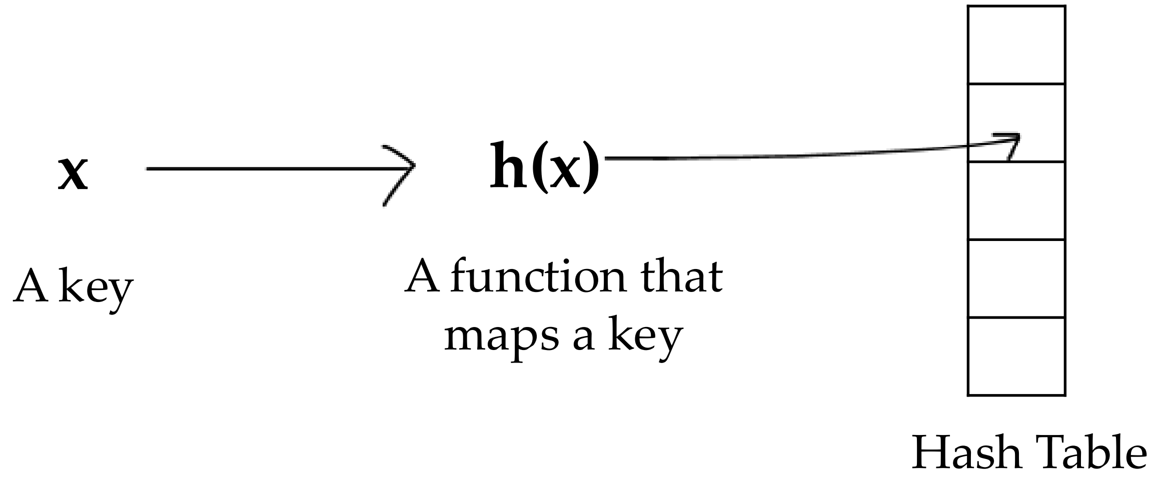
- Until now, we studied ordered indexes, such as clustered indexes and trees.

- Next, we will look at unordered indexes → Hash indexes.

Hashing

Hashing

- Three key components of a hash index:
  - A **hash table**, which stores all the keys.
  - A **function** that helps to map the key to hash table.
  - An **hashing algorithm**



Types of Hashing

Types of Hashing

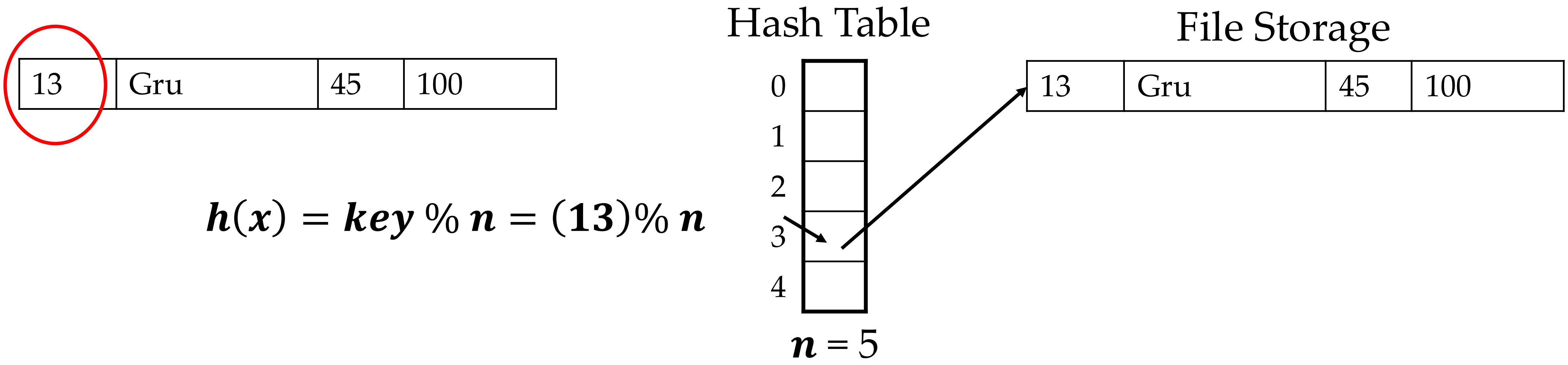
- Two types of hashing schemes:
  - **Static Hashing →** Size of hash map is fixed; cannot be increased.
  - **Dynamic Hashing →** Size of hash map can increase as needed.
    - Essentially as your databases increases over time, you can accommodate more data.

Complexity of Hashing

- As hashed indexes are unordered, they do not force maintaining any specific order.
- The position of a key in the hash table is dictated by the hash function.
- **Average case complexity** for insertion, deletion, and search → $\boldsymbol{O}(\mathbf{1})$
  - But, there are constants, which matter.
- **Worst case complexity,** given $\boldsymbol{n}$ keys → $\boldsymbol{O}(\boldsymbol{n})$
- Hash tables support **random access**, unlike earlier indexes, which support sequential access.

Static Hashing

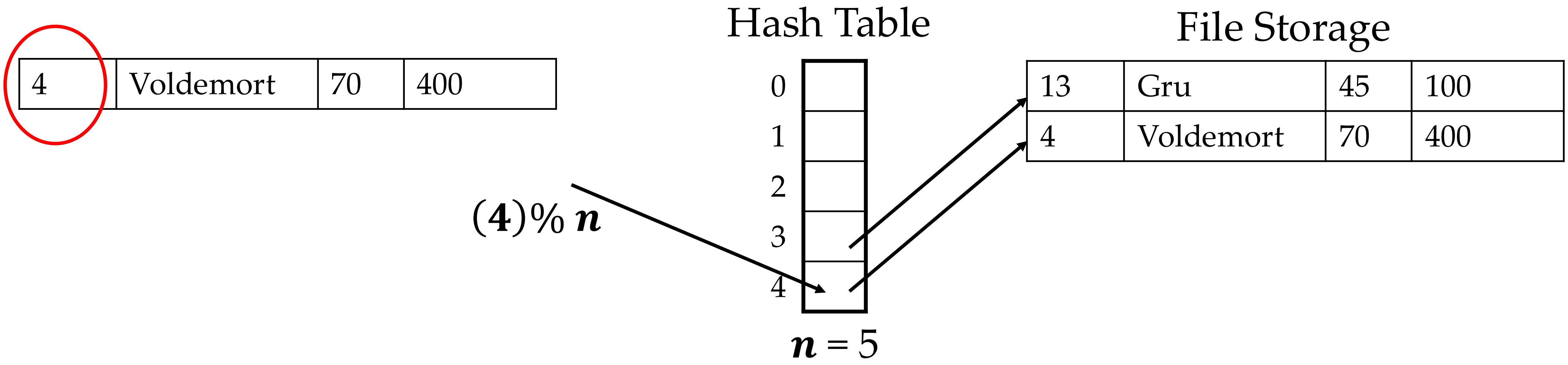
- Say, we know that in our database there will be **5 records**.
- So, we select a hash function and create a **hash table (array) of size 5**.



$$h(x) = key \,\%\, n = (13)\%\, n$$

Static Hashing

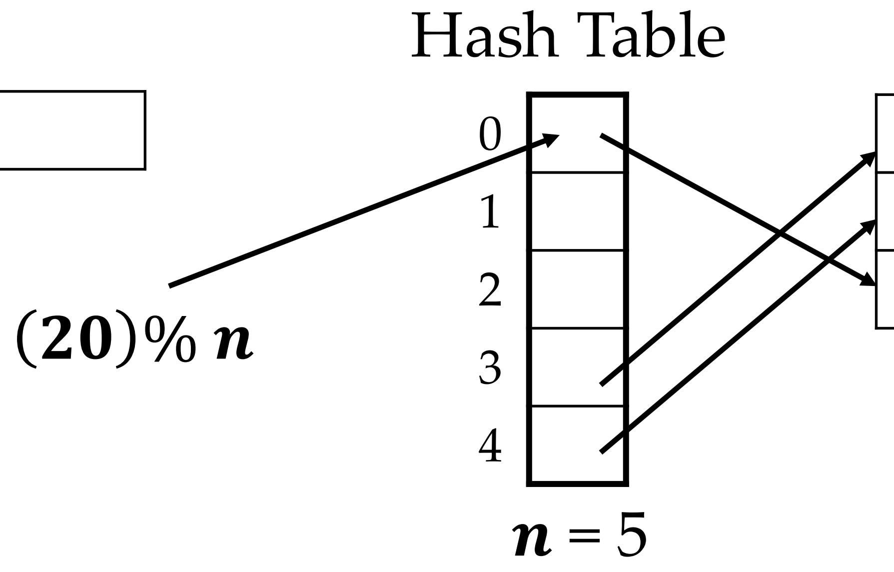
- Say, we know in our database there will be **5 records**.
- So, we select a hash function and create a **hash table (array) of size 5**.



Static Hashing

- Say, we know in our database there will be **5 records**.
- So, we select a hash function and create a **hash table (array) of size 5**.

20	Thanos	100	400



File Storage

13	Gru	45	100
4	Voldemort	70	400
20	Thanos	100	400

Challenges for Static Hashing

Challenges for Static Hashing

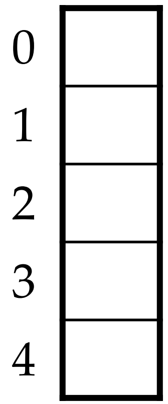
- Fixed number of Keys
- Duplicate Keys
- Collisions
- Disk Access Cost

Challenges for Static Hashing

- **Fixed number of Keys** → You should know the total size of the database in the future, and it cannot grow any further!

- For example, this hash table can only store 5 keys and if in the future your database gets a **6th record**, you need to reorganize → change hash table → too expensive!

Hash Table

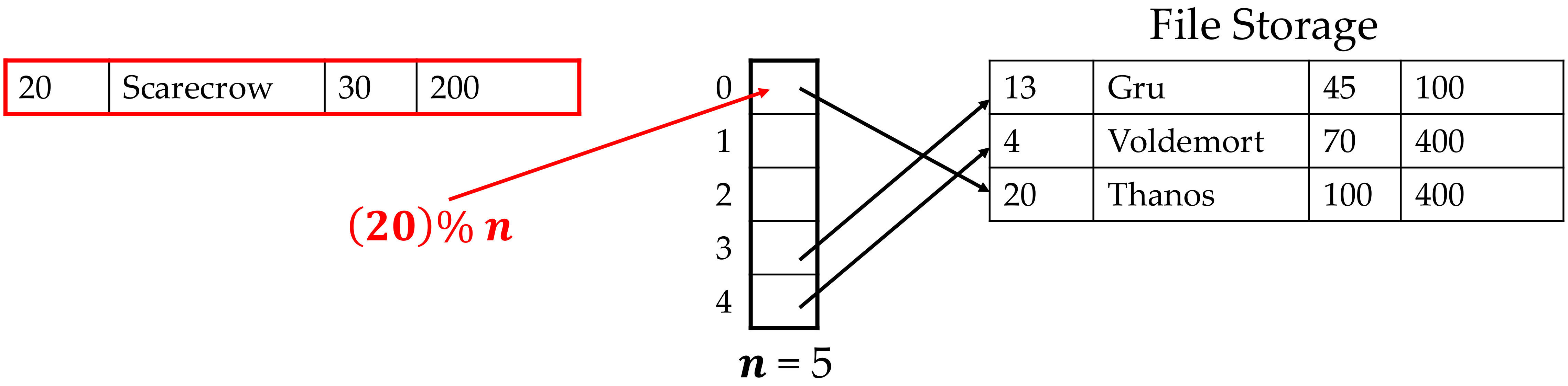


$\boldsymbol{n} = 5$

Challenges for Static Hashing

- **Unique Keys →** How to store and search for duplicate keys?

- Hash function would map duplicate keys to the same location.
  - Overwrite pointer to existing record?
  - How do you search for an existing record with duplicate keys?

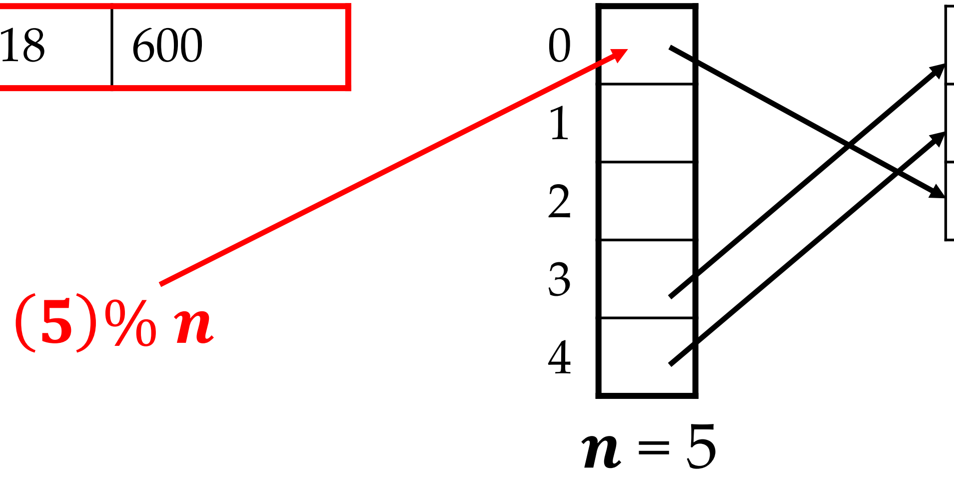


Challenges for Static Hashing

- **No Collisions** → Perfect hashing function that ensures there are no collisions.

- Hash function may end up assigning the same location to two or more records.

5	Jeoffrey	18	600



File Storage

13	Gru	45	100
4	Voldemort	70	400
20	Thanos	100	400

Challenges for Static Hashing

- **Disk Access Cost** and Lack of opportunities **for Pre-fetching.**
- Fetching a single record (point query) is fast. But, say I want to fetch a range of records. These records could be spread **across the disk → multiple blocks!**
- No longer sequential access. Moreover, **File Manager cannot even predict**!

Design Decisions for Static Hashing

Design Decisions for Static Hashing

- **Good Hash Function**:
  - Maps a large set of keys to a small array.
  - Dilemma b/w using a fast hash function vs. a hash function with low collisions.

- **Hashing Algorithm**:
  - How to handle key collisions when they occur?
  - Dilemma b/w allocating a large table to prevent collisions vs. setting up rules that allow storing duplicate and colliding keys!

Hash Functions

- Given an input key, it return an integer representation of that key.
  - Essentially, you can use hash function to convert an arbitrary byte array into a fixed-length code.

- We want a hash function that is both **fast** and has a **low collision rate**.

- Notice that we are allowing collisions as we desire fast hashing!

- Alternatively, you can use a cryptographic hash function, like SHA256.
  - No collisions!
  - Extremely secure → NIST recommended
  - Extremely slow!

Hash Functions

- Fortunately, we don’t have to create a hash functions!

- CRC-64 (1975)
  - Used in networks for error detection

- MurmurHash (2008)
  - Fast, general-purpose hash function.

- Google CityHash (2011)
  - Fast for keys of short length.

- Facebook XXHash (2012)
  - State-of-the-art

- Google FarmHash (2014)
  - Better version of CityHash; reduced collisions

Hash Schemes Performance

If you want to test the performance of various hash functions, or play with different hash functions, check out SMHasher.

Static Hashing Algorithms

- We will be looking at two common algorithms:
  - **Linear Probe Hashing**
  - **Cuckoo Hashing**

- These algorithms are also termed as **open addressing**:
  - Essentially, the key may not be in the location where the hash function points.

- More advanced algorithms (**not part of this course**)
  - Robinhood hashing
  - Hopscotch hashing
  - Swiss Tables

Linear Probe Hashing

Linear Probe Hashing

- Simplest hashing algorithm → resolves collision by searching for next empty **slot.**
- Requires a **fixed-size giant array** (smaller the size, more collisions).
  - Hash table's **load factor** (like a threshold) determines when the **table is too full.**
  - No new key should be added, otherwise collisions → **allocate new table**!
- **Inserting a key**:
  - Use your hash function to find a **slot** (position).
  - If the location is empty, store the key in that slot.
  - Otherwise, start sequential scanning from that location.
  - When you find an empty slot, insert your key in that slot.
- **Deletion and Search**:
  - Same as insertion.

Linear Probe Hashing

Hash Table

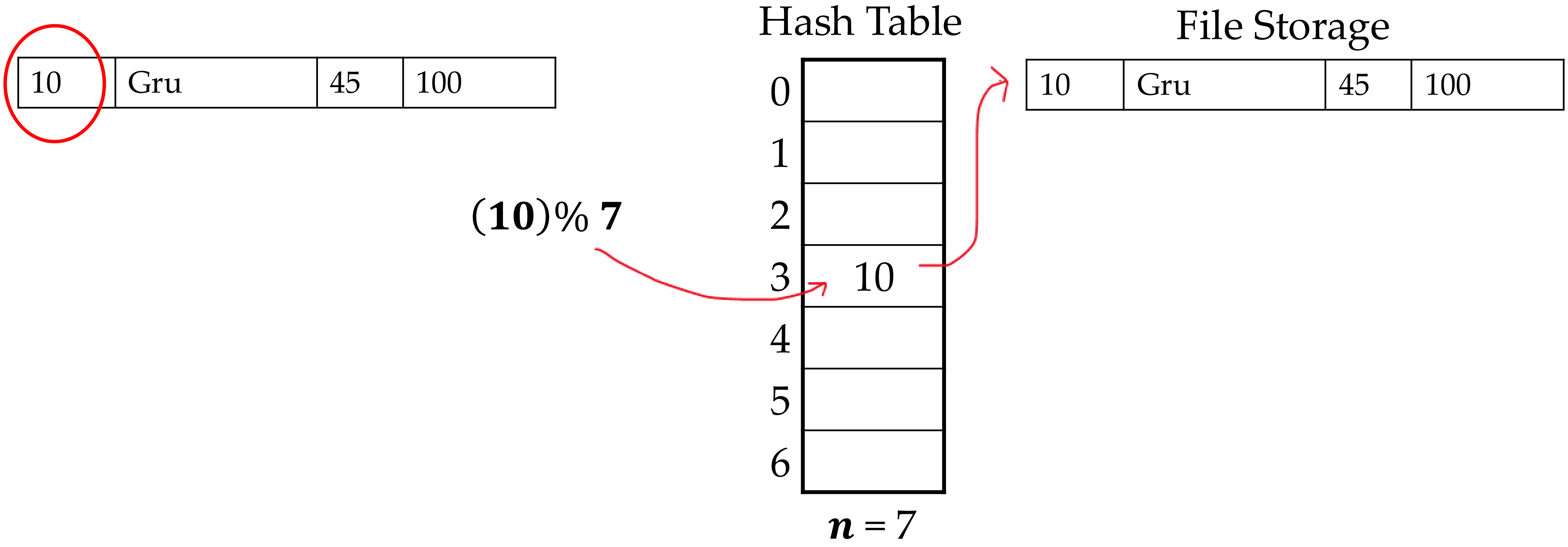
File Storage

$h(key) \% \ 7$

0	
1	
2	
3	
4	
5	
6	

$n = 7$

Linear Probe Hashing



Linear Probe Hashing

7	Voldemort	70	400

$(7)\%\ 7$

Hash Table

0	7
1	
2	
3	10
4	
5	
6	

$n = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400

Linear Probe Hashing

17	Anakin	45	300

$(\mathbf{17})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	10
4	
5	
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400

Linear Probe Hashing

17	Anakin	45	300

(**17**)% **7**

Hash Table

0	7
1	
2	
3	10
4	
5	
6	

$n = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400

Linear Probe Hashing

17	Anakin	45	300

$(17)\%\ 7$

Hash Table

0	7
1	
2	
3	10
4	17
5	
6	

$n = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300

Linear Probe Hashing

24	Joker	60	300

$$(\mathbf{24})\%\ \mathbf{7}$$

Hash Table

0	7
1	
2	
3	10
4	17
5	
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300

Linear Probe Hashing

24	Joker	60	300

$$(\mathbf{24})\%\ \mathbf{7}$$

Hash Table

0	7
1	
2	
3	10
4	17
5	
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300

Linear Probe Hashing

24	Joker	60	300

$(\mathbf{24})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	10
4	17
5	
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300

Linear Probe Hashing

24	Joker	60	300

$(\mathbf{24})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	10
4	17
5	24
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300
24	Joker	60	300

Linear Probe Hashing

5	Thanos	100	500

$(\mathbf{5})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	10
4	17
5	24
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300
24	Joker	60	300

Linear Probe Hashing

5	Thanos	100	500

$(\mathbf{5})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	10
4	17
5	24
6	

$\boldsymbol{n} = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300
24	Joker	60	300

Linear Probe Hashing

5	Thanos	100	500

$(\mathbf{5})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	10
4	17
5	24
6	5

$n = 7$

File Storage

10	Gru	45	100
7	Voldemort	70	400
17	Anakin	45	300
24	Joker	60	300
5	Thanos	100	500

Searching in Linear Probe Hashing

- Follow the same algorithm as you are trying to insert.
    - If the slot is empty, key not found.
    - If the slot is full, then continue to next slot.
    - Stop when you reach an empty slot or have covered all the slots.

Deleting in Linear Probe Hashing

- How can we delete a record?
- Say, we want to delete the **record 10**, which maps to **slot 3**.

10	Gru	45	100

$(\mathbf{10})\%\ \mathbf{7} \longrightarrow 3$

Hash Table

0	7
1	
2	
3	10
4	17
5	24
6	5

$\boldsymbol{n} = 7$

Deleting in Linear Probe Hashing

- How can we delete a record?
- Say, we want to delete the **record 10**, which maps to **slot 3**.
- Can we set slot 3 to **empty**?

10	Gru	45	100

$(\mathbf{10})\%\ \mathbf{7}$

Hash Table

0	7
1	
2	
3	
4	17
5	24
6	5

$\boldsymbol{n} = 7$

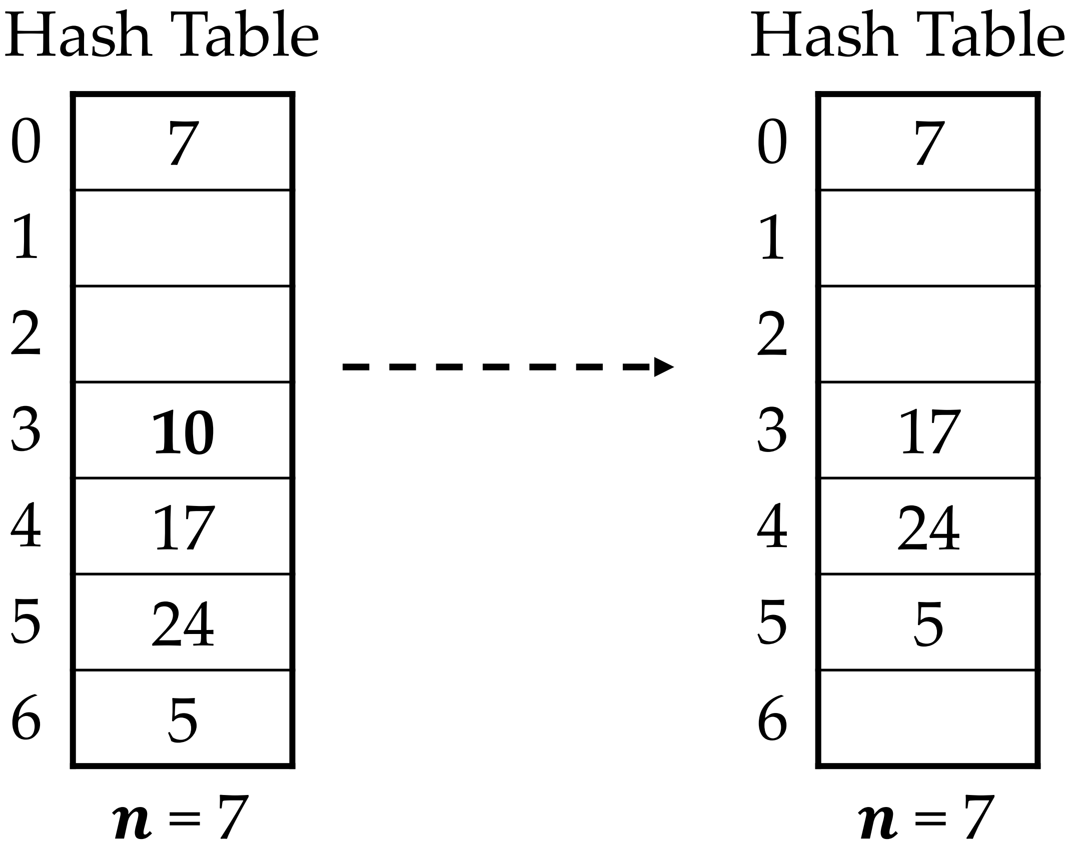
Deleting in Linear Probe Hashing

- On deleting a record, setting a slot to empty is dangerous!
  - Other keys could have also mapped to the same slot, but due to the slot being full, they were in subsequent locations.
  - By emptying the slot, you are indicating that other keys also do not exist!

- Two possible solutions:
  - Rearrangement
  - Tombstones

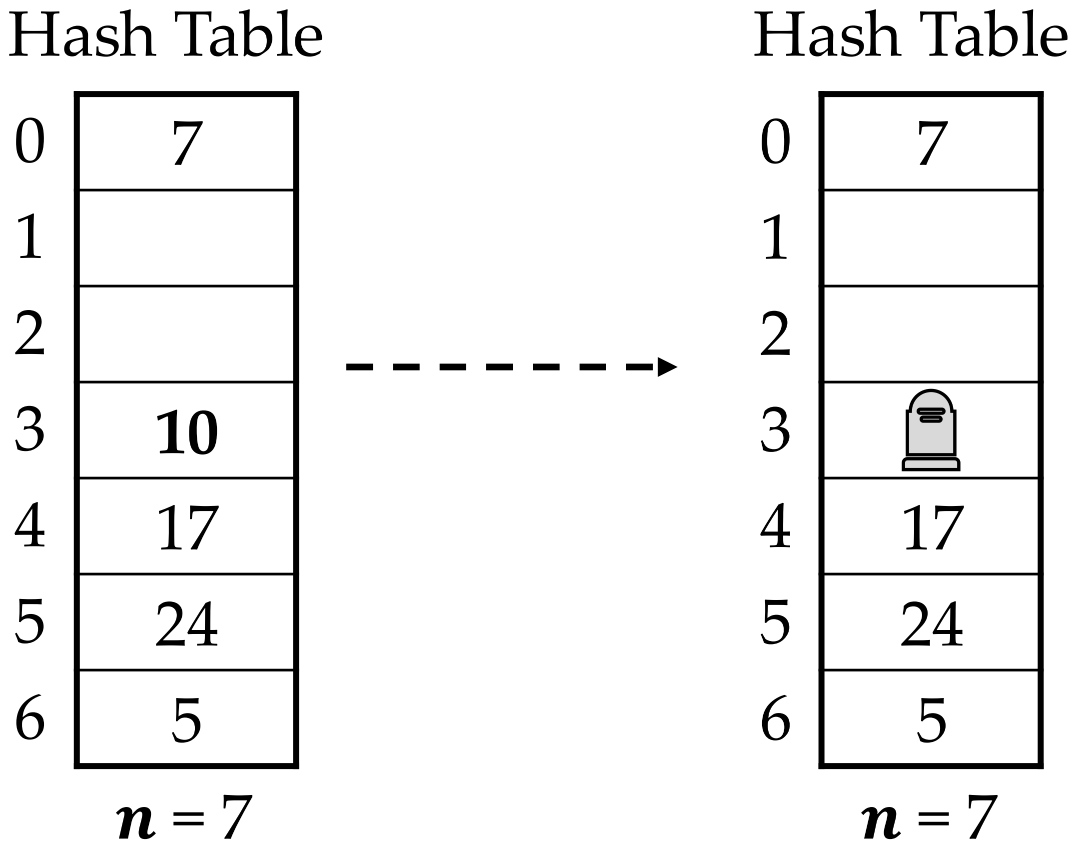
Deletions: Rearrangement

- Once a key is deleted, you can rehash all the keys again.
- Any key that was supposed to be mapped to the same slot can now take place.
- Too expensive! No database does this.



Deletions: Tombstones

- Once a key is deleted, you place a **tombstone for that key** in that slot.
- Tombstone informs any future query that the specific key does not exist.
- However, other keys may still exist!



For each tombstone, you need to maintain the list of keys that have been deleted!

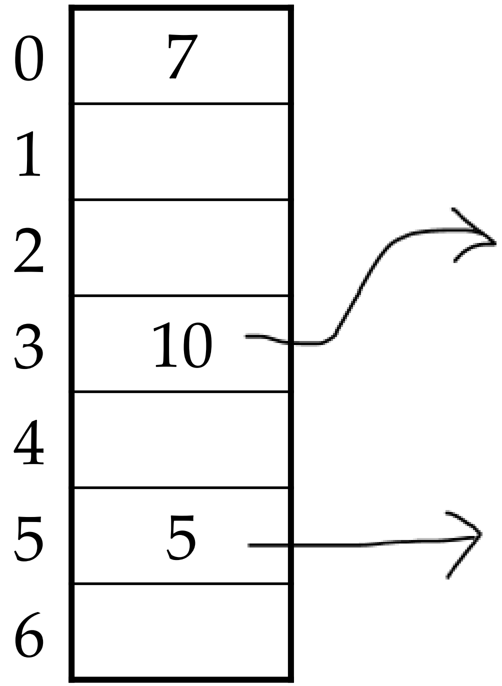
Duplicate Keys in Linear Probe Hashing

- How do you handle **duplicate (non-unique)** keys?

- Two ways:
  - Maintain a list of values
  - Just simply allow adding redundant keys

Duplicate Keys: List of Values

Hash Table



$n = 7$

Lists of Values

10	Gru	45	100
10	Voldemort	70	400

5	Anakin	45	300
5	Joker	60	300
5	Thanos	100	500

Duplicate Keys: Allow Redundant Keys

Hash Table

0	7
1	5
2	
3	10
4	5
5	5
6	10

$\boldsymbol{n} = 7$

Cuckoo Hashing

- Why the name cuckoo?

- Like the bird cuckoo, if we do not find a free slot for a key, we may kick out an existing key!

- In cuckoo hashing, we use multiple **hash functions** to find free slots to store the key.
    - Each hash function may give us a slot to place and if any of those slots is free, we store the key!

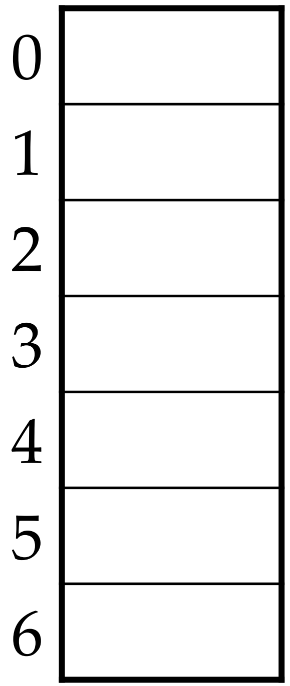
- If no slot is free, evict an existing key!

Cuckoo Hashing: Insertion

$h1(key)$

$h2(key)$

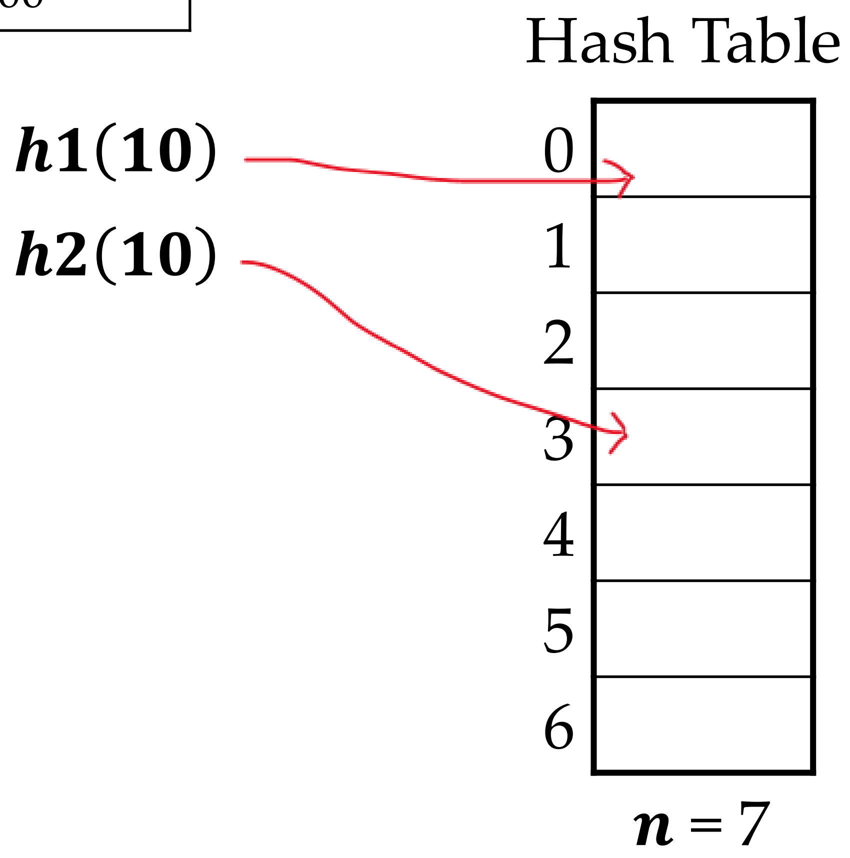
Hash Table



$n = 7$

Cuckoo Hashing: Insertion

10	Gru	45	100



Randomly select a slot, say it selects slot 3 for storing key 10.

Cuckoo Hashing: Insertion

10	Gru	45	100

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

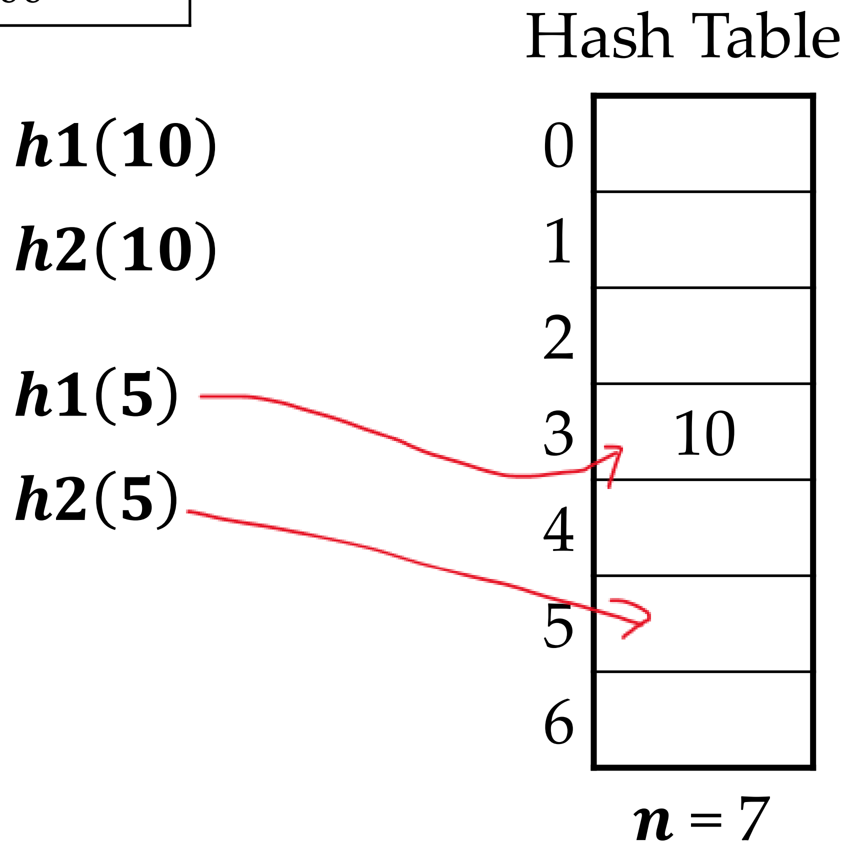
Hash Table

0	
1	
2	
3	10
4	
5	
6	

$\boldsymbol{n} = 7$

Cuckoo Hashing: Insertion

5	Anakin	25	400



Cuckoo Hashing: Insertion

5	Anakin	25	400

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

$\boldsymbol{h1(5)}$

$\boldsymbol{h2(5)}$

Hash Table

0	
1	
2	
3	10
4	
5	5
6	

$\boldsymbol{n} = 7$

As slot 3 is occupied, we select slot 5 to store key 5.

Cuckoo Hashing: Insertion

18	Joker	66	300

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

$\boldsymbol{h1(5)}$

$\boldsymbol{h2(5)}$

$\boldsymbol{h1(18)}$

$\boldsymbol{h2(18)}$

Hash Table

0	
1	
2	
3	10
4	
5	5
6	

$\boldsymbol{n} = 7$

Cuckoo Hashing: Insertion

18	Joker	66	300

$h1(10)$

$h2(10)$

$h1(5)$

$h2(5)$

$h1(18)$

$h2(18)$

Hash Table

0	
1	
2	
3	10
4	
5	5
6	

$n = 7$

Say, it decides to kick key 5

Cuckoo Hashing: Insertion

18	Joker	66	300

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

$\boldsymbol{h1(5)}$

$\boldsymbol{h2(5)}$

$\boldsymbol{h1(18)}$

$\boldsymbol{h2(18)}$

Hash Table

0	
1	
2	
3	10
4	
5	18
6	

$\boldsymbol{n} = 7$

Say, it decides to kick key 5

Cuckoo Hashing: Insertion

18	Joker	66	300

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

$\boldsymbol{h1(5)}$

$\boldsymbol{h2(5)}$

$\boldsymbol{h1(18)}$

$\boldsymbol{h2(18)}$

$\boldsymbol{h1(5)}$

Hash Table

0	
1	
2	
3	10
4	
5	18
6	

$\boldsymbol{n} = 7$

So, we need to rehash key 5, and only remaining slot is the slot occupied by key 10

Cuckoo Hashing: Insertion

18	Joker	66	300

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

$\boldsymbol{h1(5)}$

$\boldsymbol{h2(5)}$

$\boldsymbol{h1(18)}$

$\boldsymbol{h2(18)}$

$\boldsymbol{h1(5)}$

Hash Table

0	
1	
2	
3	5
4	
5	18
6	

$\boldsymbol{n} = 7$

So, kick out key 10!

Cuckoo Hashing: Insertion

18	Joker	66	300

$\boldsymbol{h1(10)}$

$\boldsymbol{h2(10)}$

$\boldsymbol{h1(5)}$

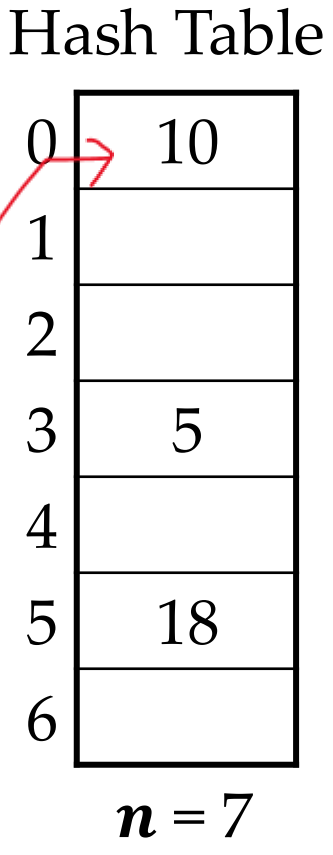
$\boldsymbol{h2(5)}$

$\boldsymbol{h1(18)}$

$\boldsymbol{h2(18)}$

$\boldsymbol{h1(5)}$

$\boldsymbol{h1(10)}$



And now, rehash key 10

Challenges with Cuckoo Hashing

- So what are the challenges with cuckoo hashing?

- Insertions are **expensive** → We need to do rehashing!

- We can get stuck into an **infinite loop**.
  - To exit the infinite loop, add more hash functions, or increase size of table, or maintain some list.

Dynamic Hashing

- The biggest challenge for static hashing remains to be **fixed size of hash table**.

- Alternatively, use dynamic hashing algorithms:
  - Chained Hashing
  - Extensible Hashing
  - Linear Hashing

Chained Hashing

Chained Hashing

- For each slot in the hash table, there is a **linked list of buckets**.

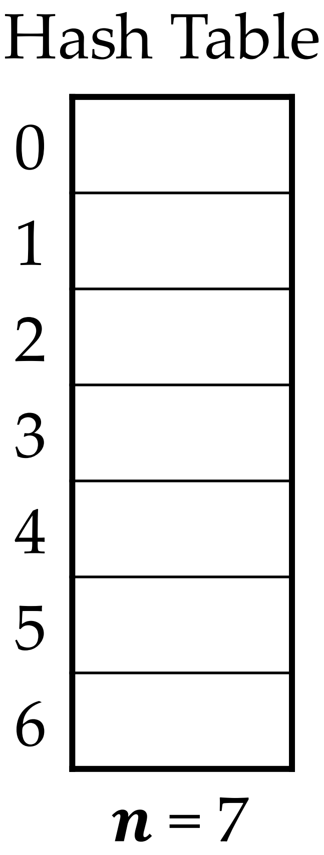
- Essentially, collisions are resolved by **placing all keys with the same slot** into same linked list.

- Searching for a key requires scanning the linked list till you find the key or have reached end of the list.

Chained Hashing

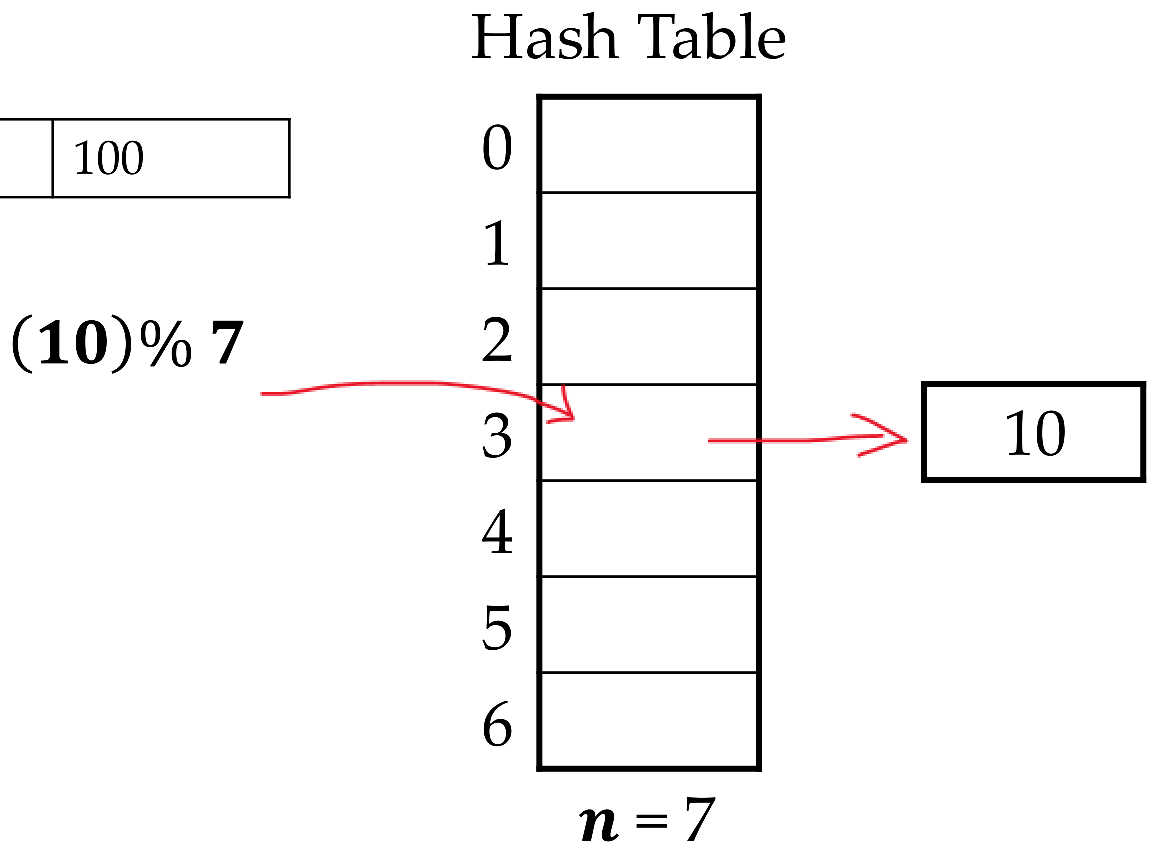
Simple hash function

$$h(key) = key \,\%\, n = (key) \,\%\, 7$$



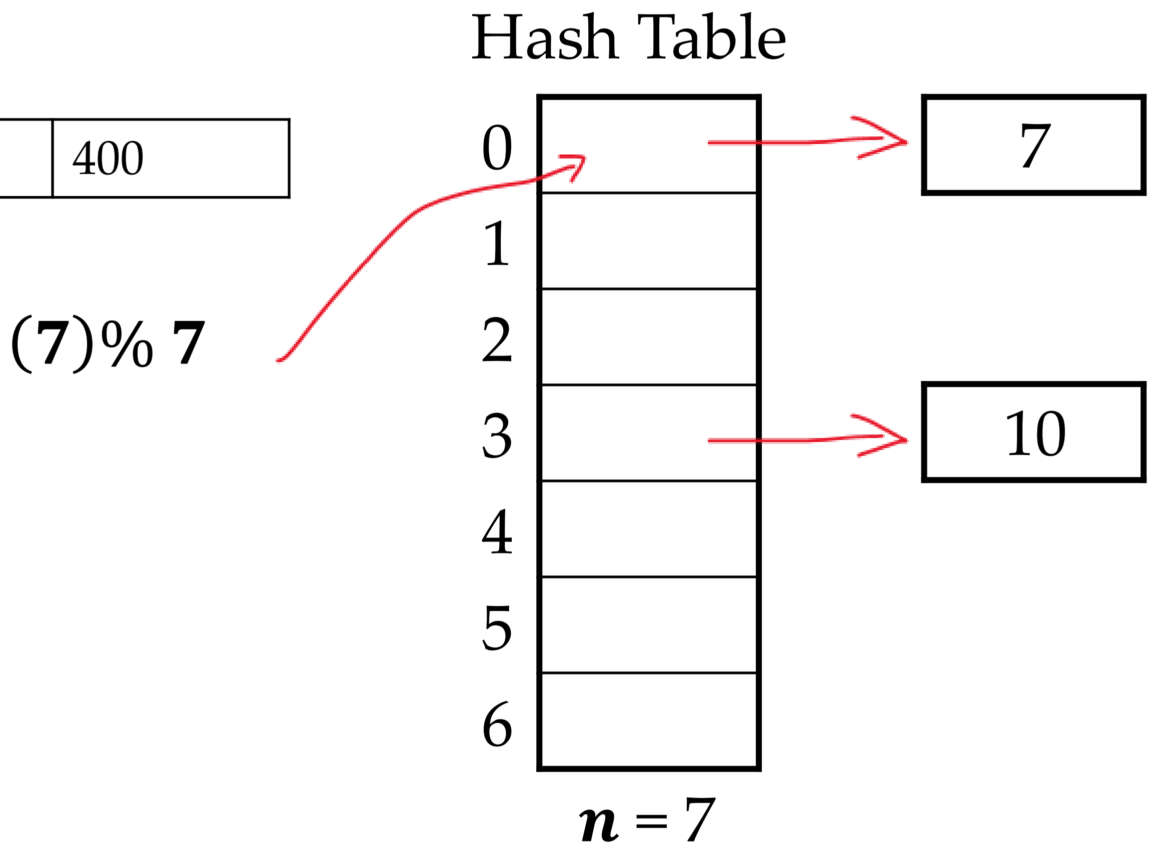
Chained Hashing

10	Gru	45	100



Chained Hashing

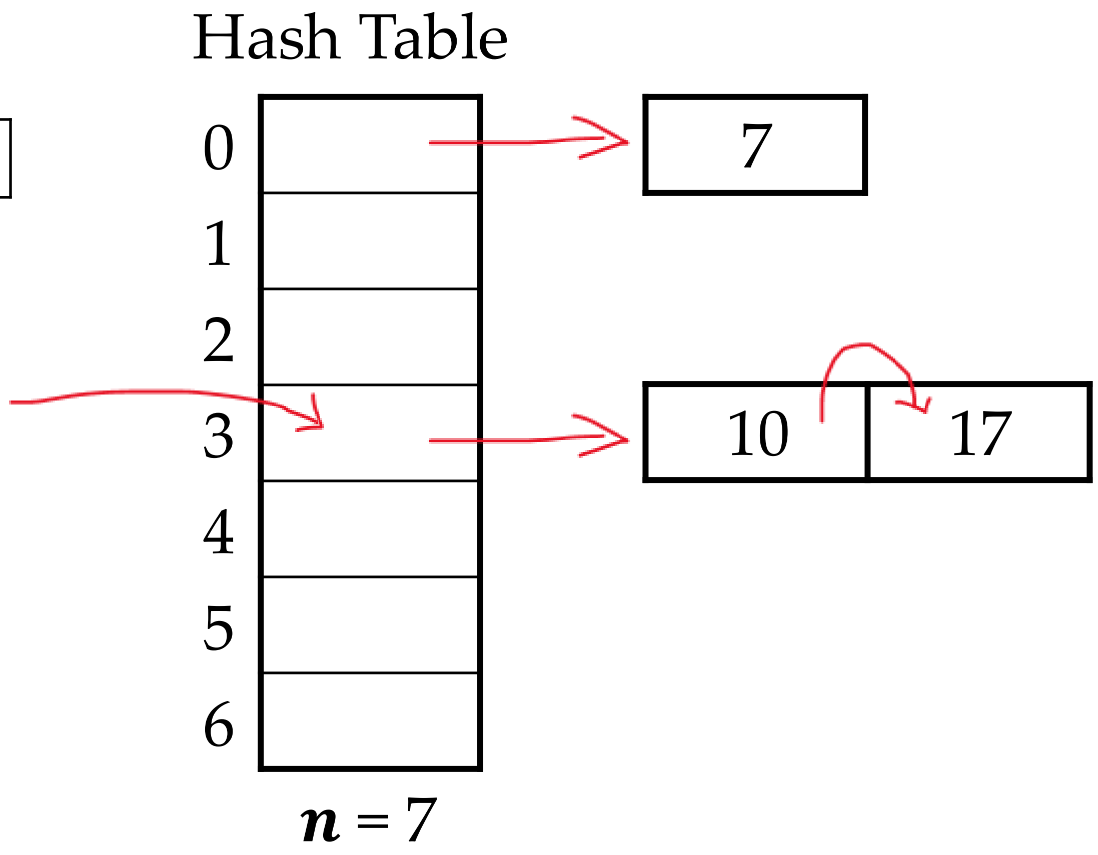
7	Voldemort	70	400



Chained Hashing

17	Anakin	45	300

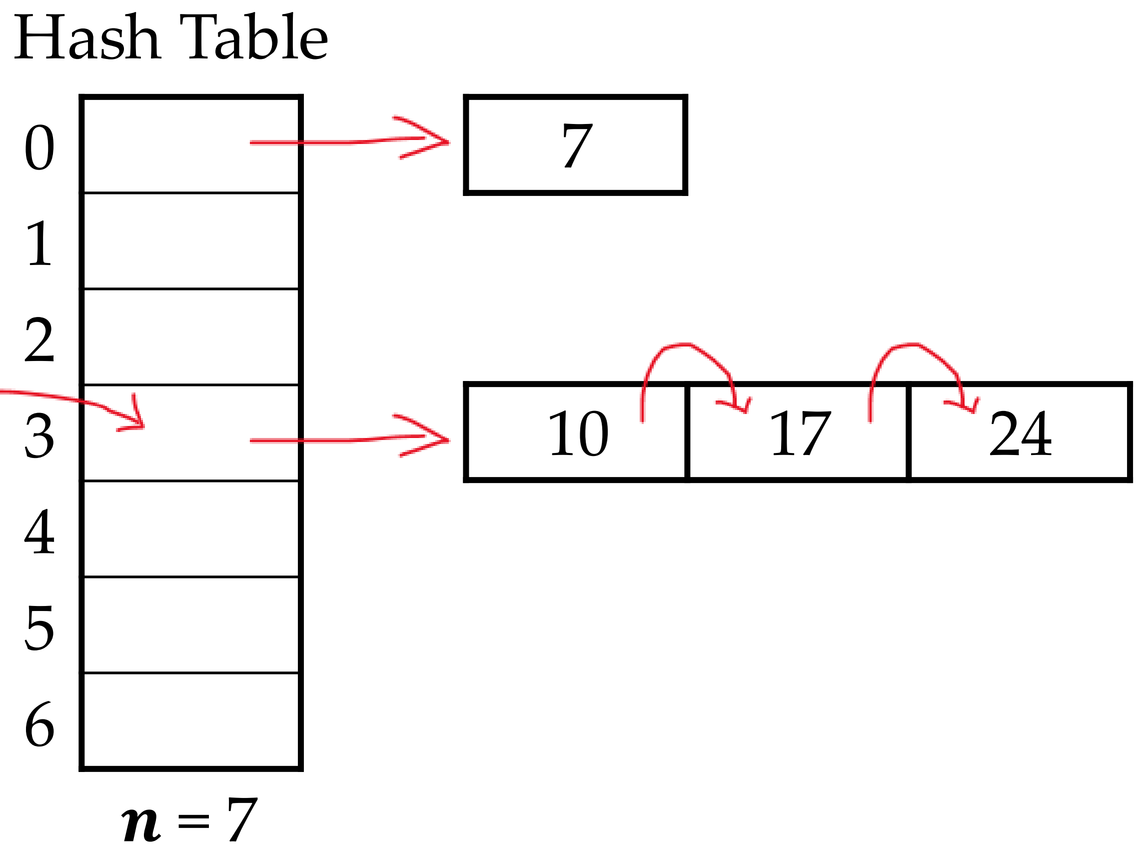
$(\mathbf{17})\%\ \mathbf{7}$



Chained Hashing

24	Joker	60	300

$$\mathbf{(24)\%\ 7}$$



Searching in Chained Hashing

- Use the hash function to reach the specific slot, and then scan the linked list till you find the key or have reached end of the list.

- For example, **on searching 17**, you would first reach **slot 3**, and then scan the list for **slot 3** and find it is as the **second entry in the linked list**.

Challenges with Chained Hashing

- What is the key challenge with chained hashing?

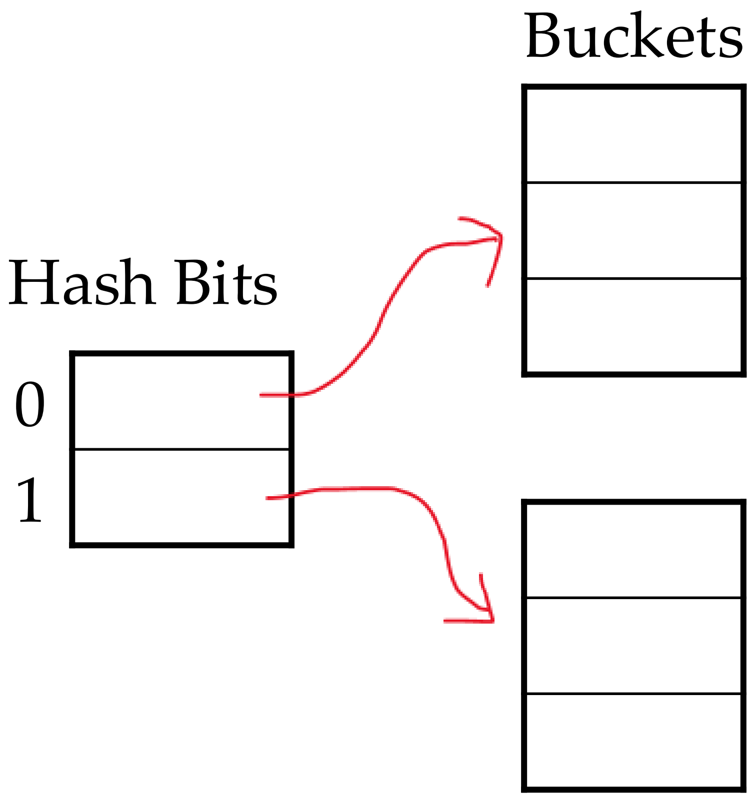
- If a lot of keys are hashed to the same slot, then
  - You have a massively large linked list, and
  - Searching a key comes expensive → same cost as linear scan.

Extensible Hashing

- Solves the problem of massively large linked lists.
- Requires linked lists to be split, when size crosses a threshold.
- Requires observing each key in a bit format.
- When you hash a key, you get a numeric (base-10 or base-16) representation.
  - You can convert that base-10 to binary format (base-2).
- For example: 4 can be represented as 100 in a 3-bit representation.

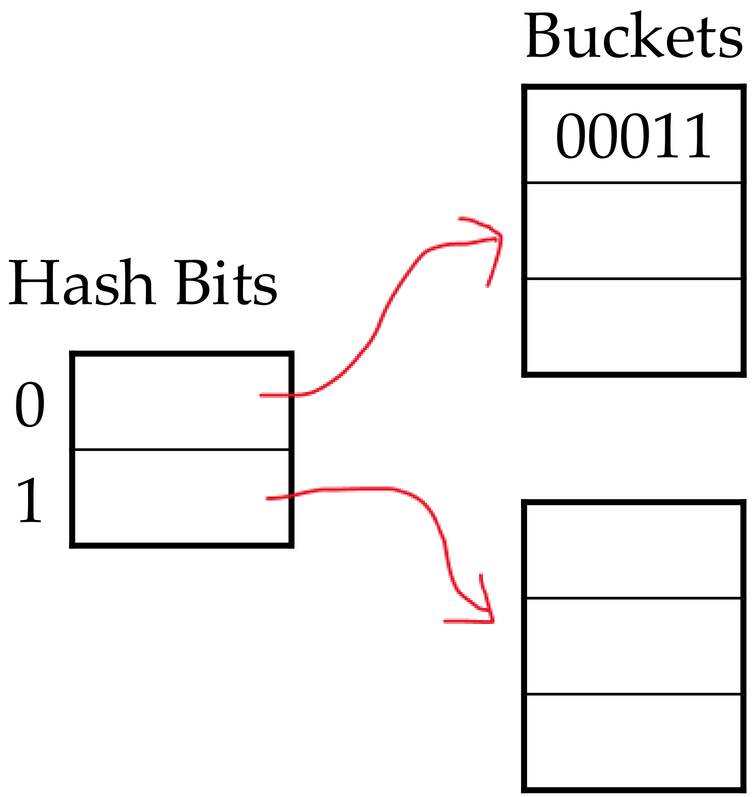
Extensible Hashing

- Initially, your hash map is **1-bit**, and you have some fixed number of buckets for each bit → Say 3 buckets.



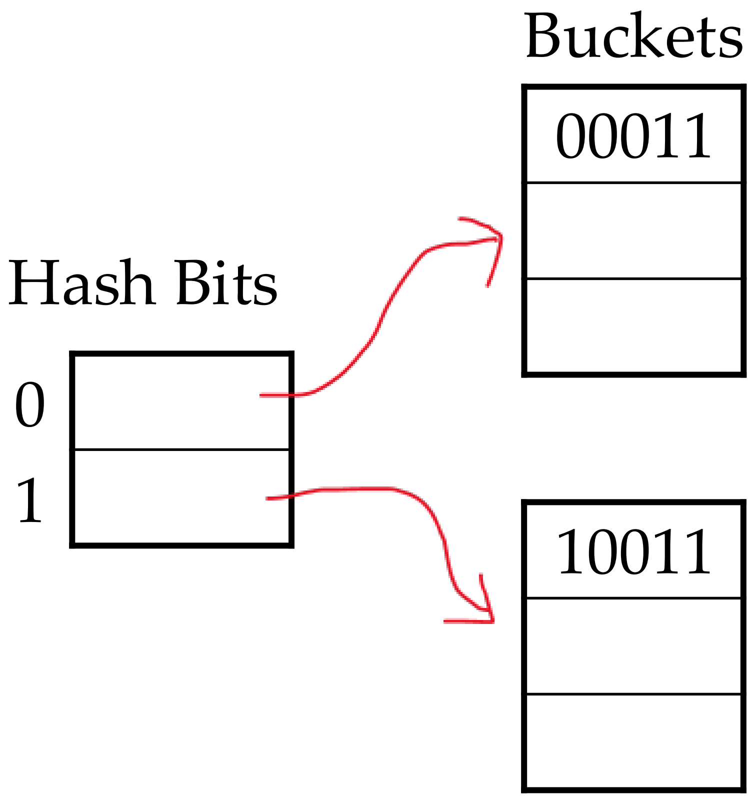
Extensible Hashing

- Assume on passing the **key 13 through a hash function**, the binary representation is **00011**.



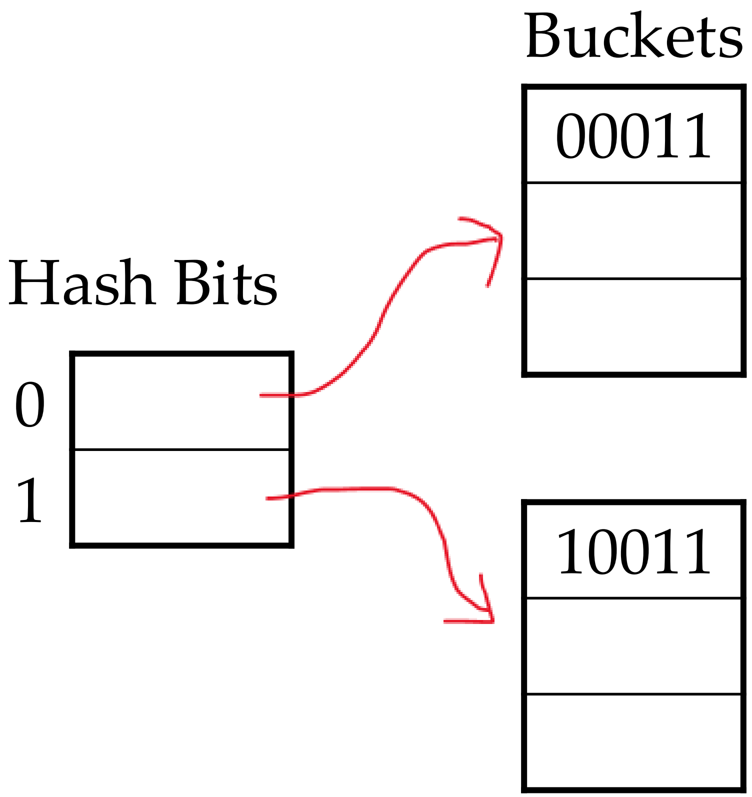
Extensible Hashing

- Another **key 7,** after passing it **through a hash function**, let the binary representation be **10011**.



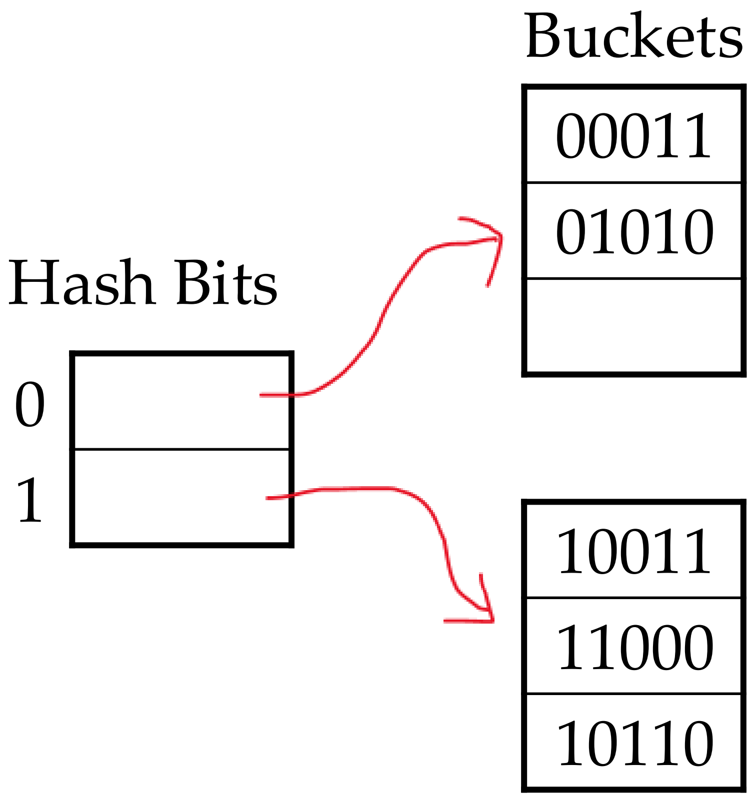
Extensible Hashing

- This way, all keys with binary representation starting from 0 go to buckets for bit 0, and vice versa for buckets for bit 1.



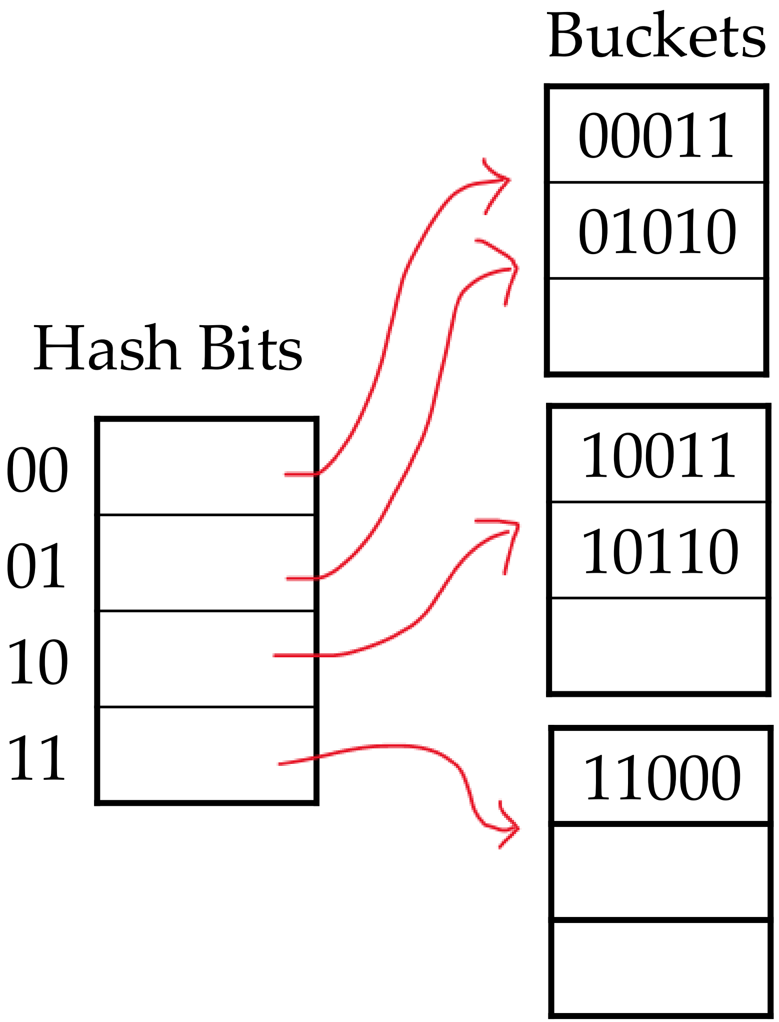
Extensible Hashing

- Let's assume all the buckets for **bit 1** are full.



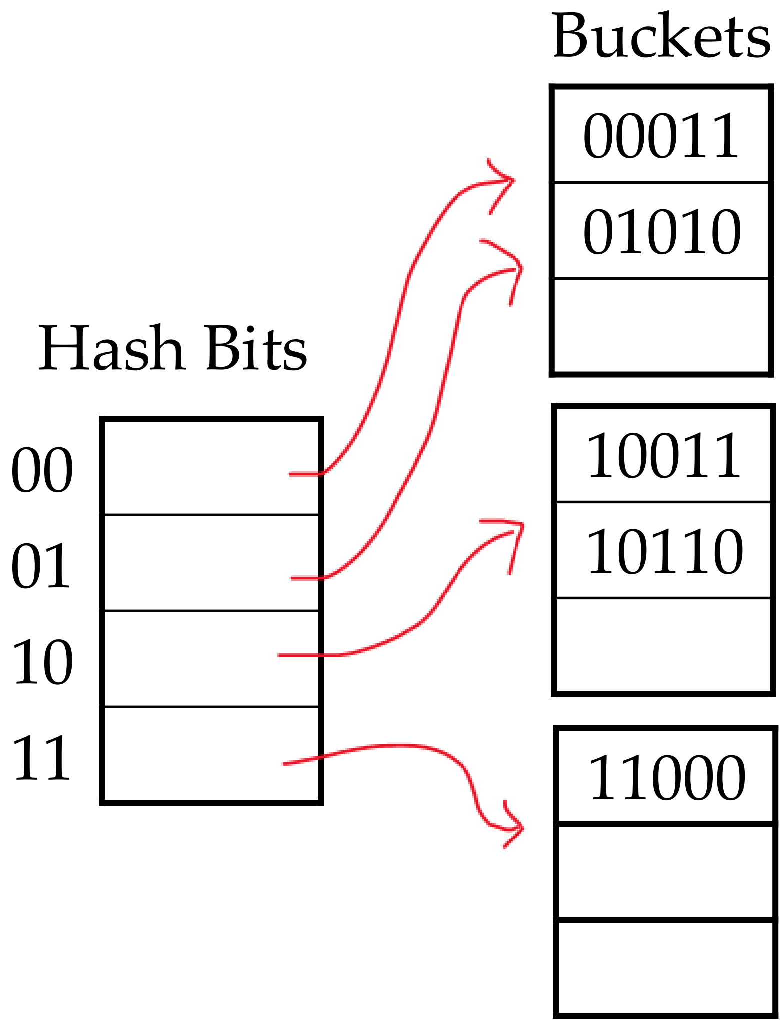
Extensible Hashing

- So, now we need to **split the buckets** for bit 1. This will require expanding the bit representation from 1-bit to 2-bits.



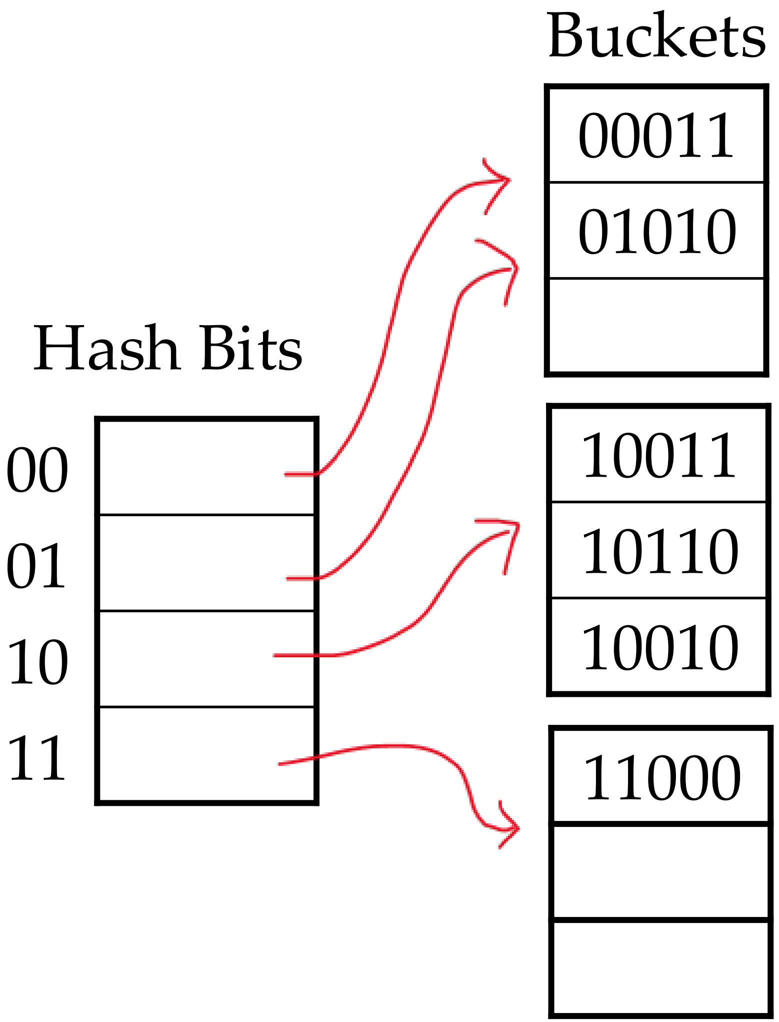
Extensible Hashing

- Notice that all the **2-bit** representations **starting with bit-0** continue pointing to the old buckets.



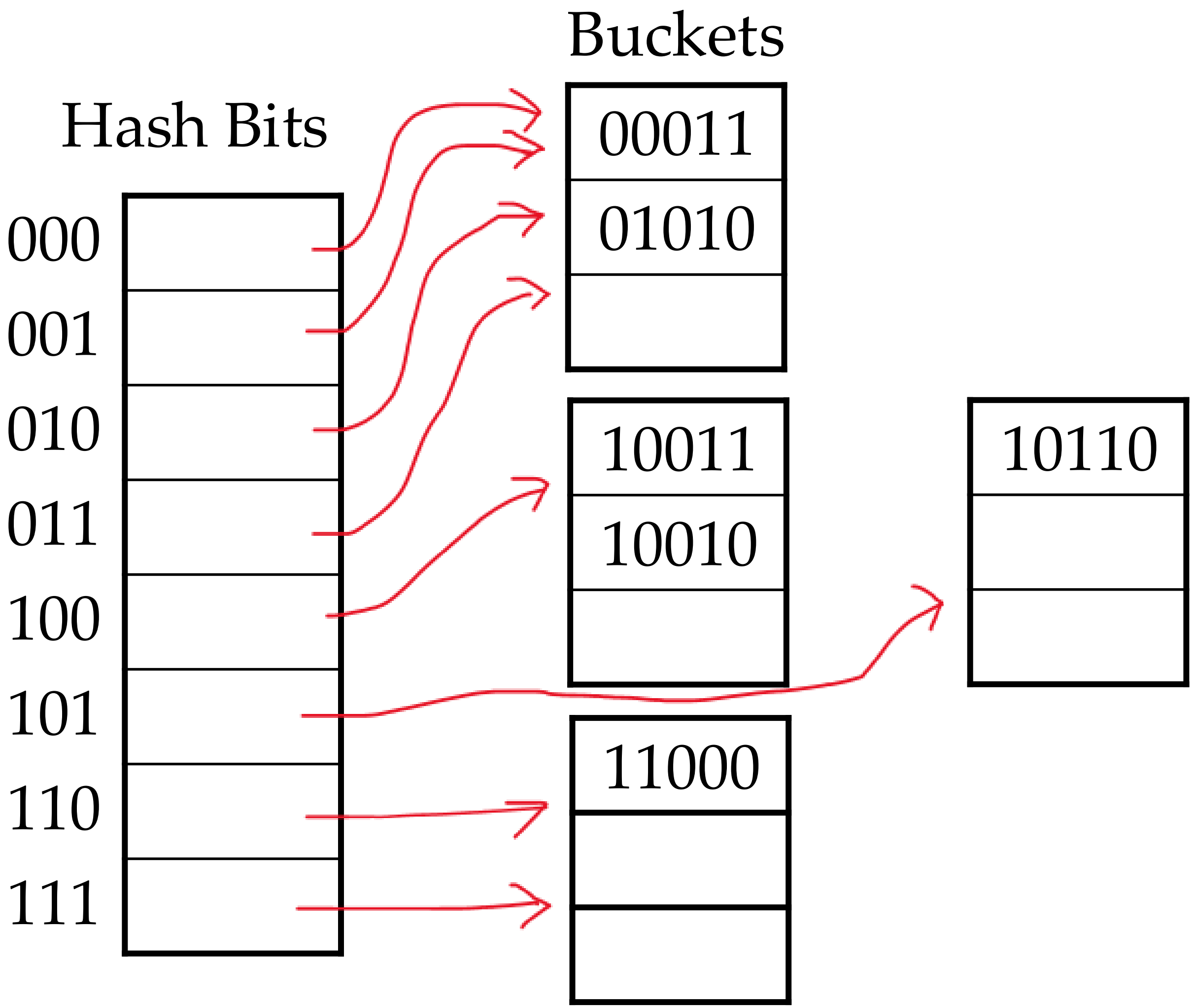
Extensible Hashing

- Next, assume we received a **key 18,** and on passing it **through the hash function,** the binary representation is **10010**.



Extensible Hashing

- Observe that all the buckets for bits 10 are full → Need to split again buckets for 10.
- Now, **3-bits**.



Linear Hashing

Linear Hashing

- Extensible hashing works well, but we perform the splitting **lazily** when the buckets for some bit(s) are full.

- What if we allow splitting to happen **eagerly** in the hope that in the future we would anyways need to split.

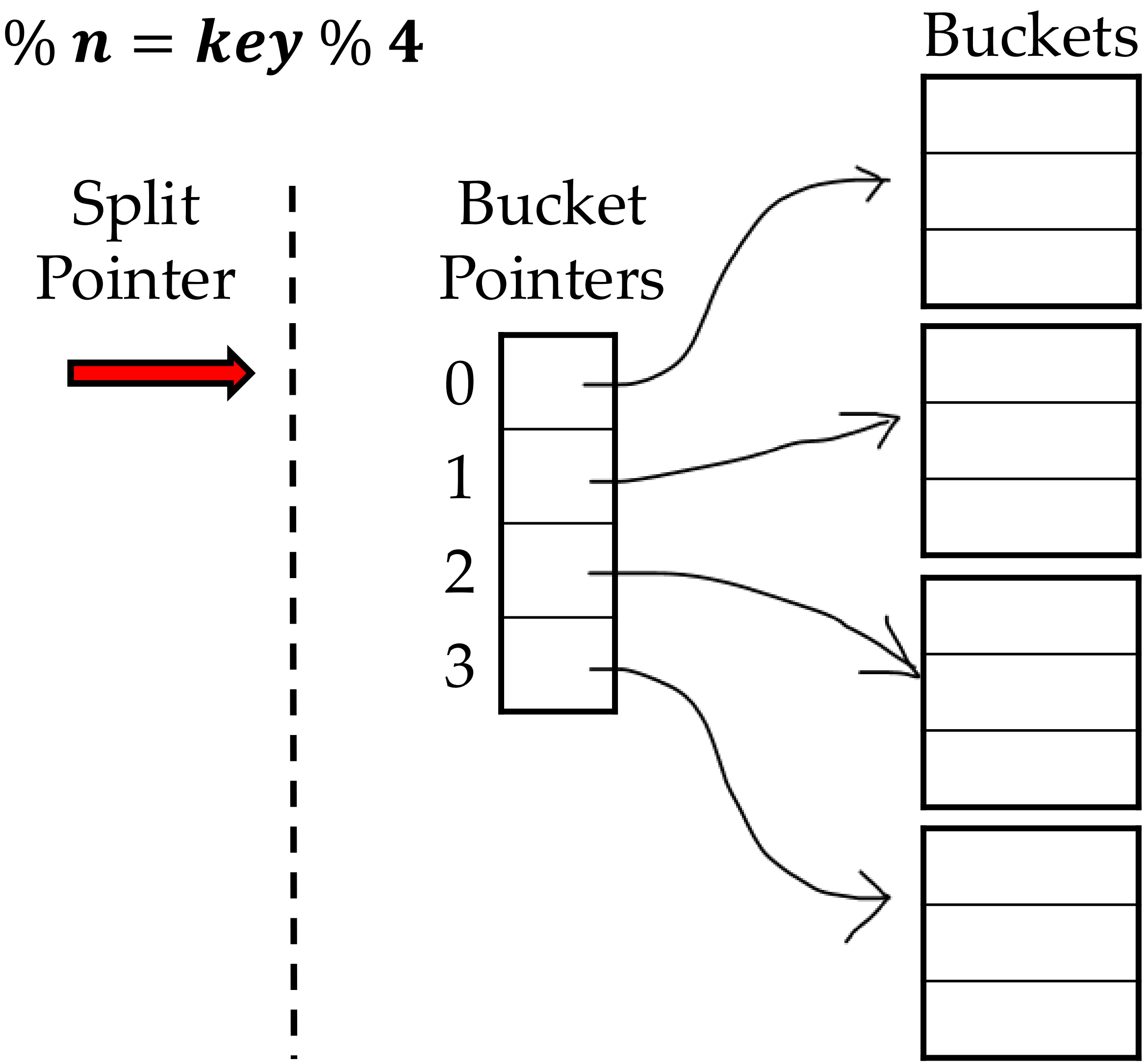
- **Linear hashing** performs eager and random splitting.
  - We call the splitting random because you may end up splitting empty buckets.

- Note: there is no longer tracking of buckets via binary representation.

- What we need is a **split pointer** that tells where did the last split took place.
  - Every **n-th split** introduces a new hash function.

Linear Hashing

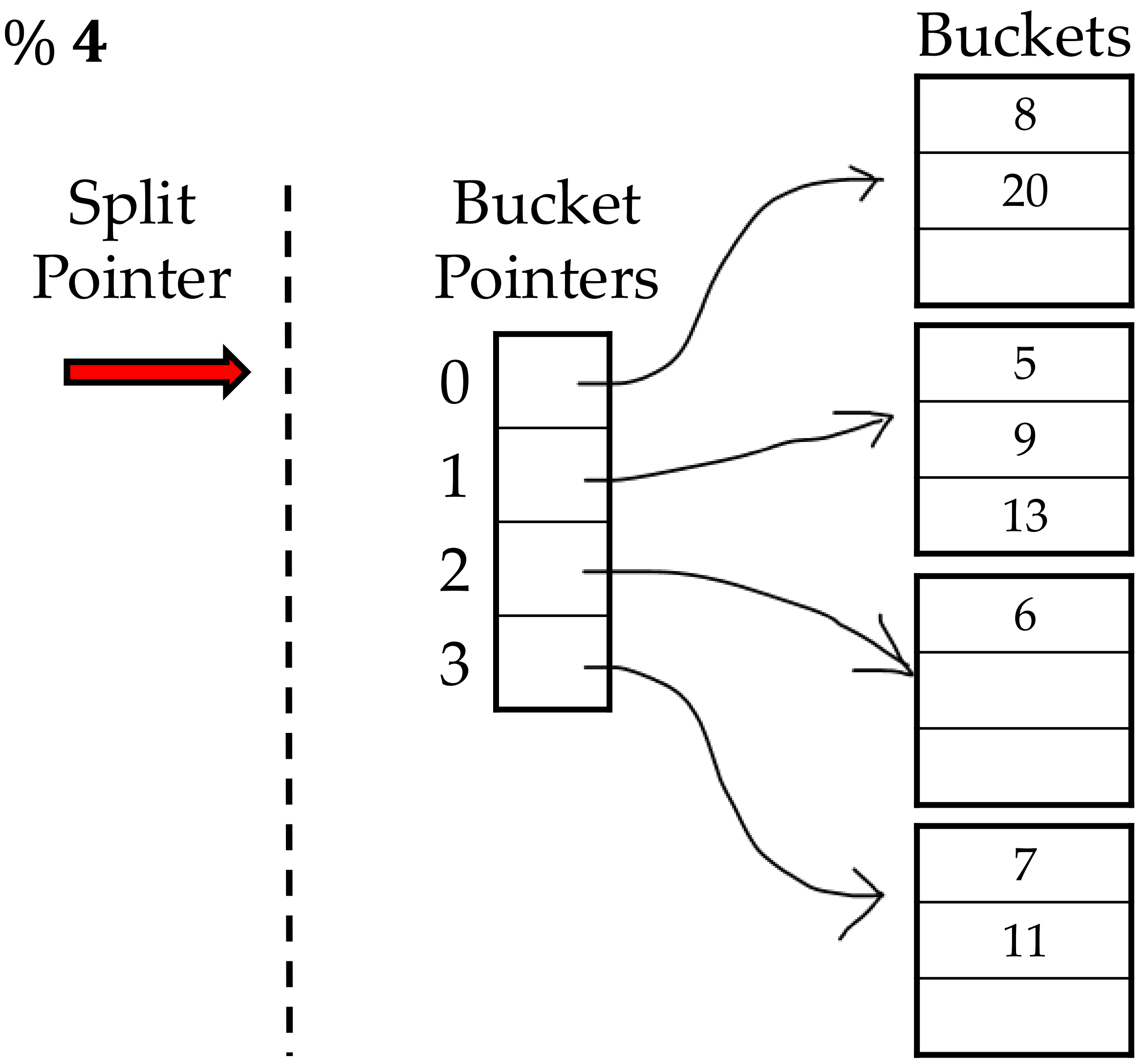
- Initially say our hash function is:
  $\boldsymbol{h_1(key) = key \; \% \; n = key \; \% \; 4}$



Linear Hashing

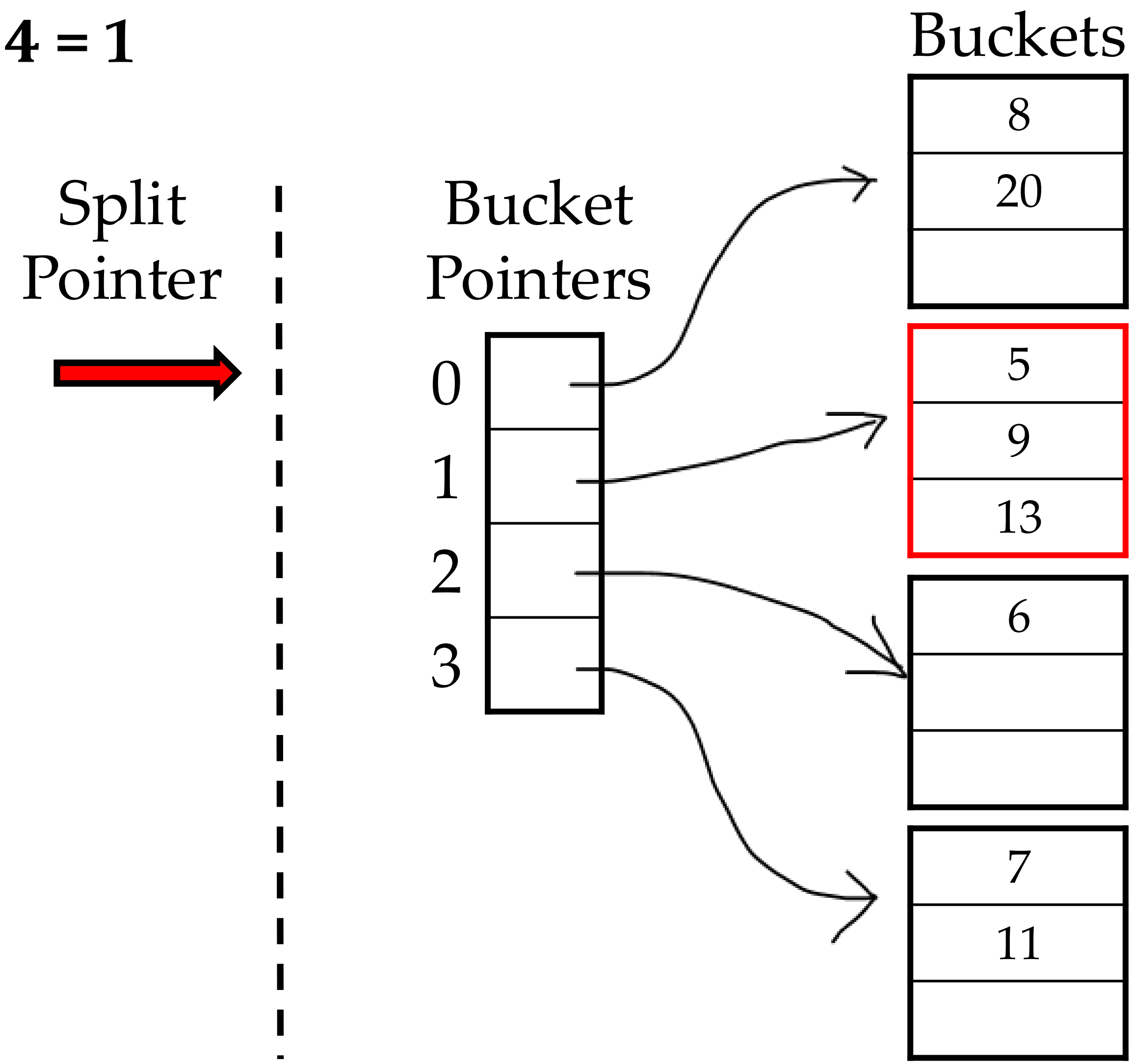
- Say our buckets look like this:

  $h_1(key) = key \% 4$



Linear Hashing

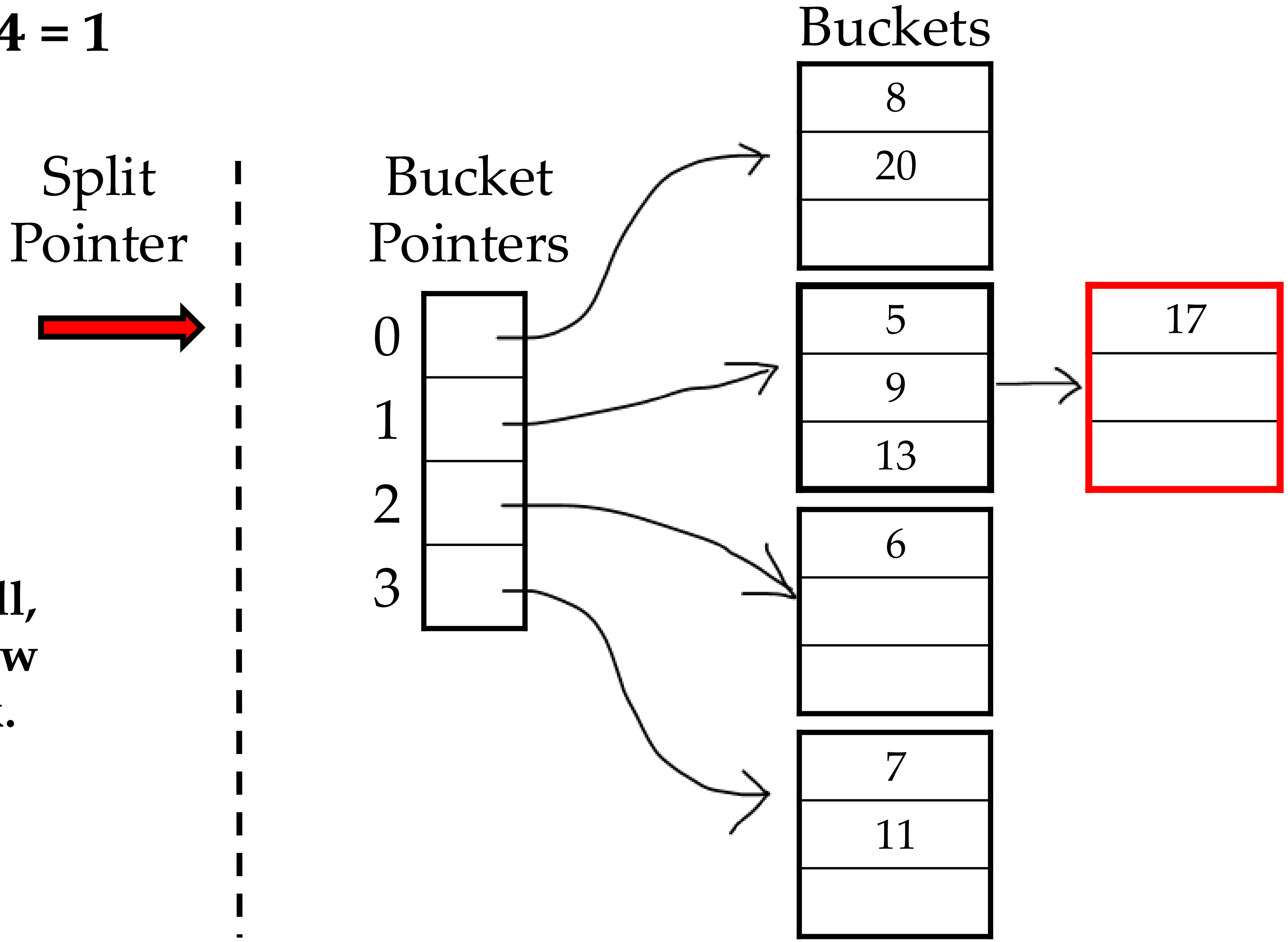
- Let's insert a **key = 17**:
  $\boldsymbol{h_1(17) = 17 \% 4 = 1}$



Linear Hashing

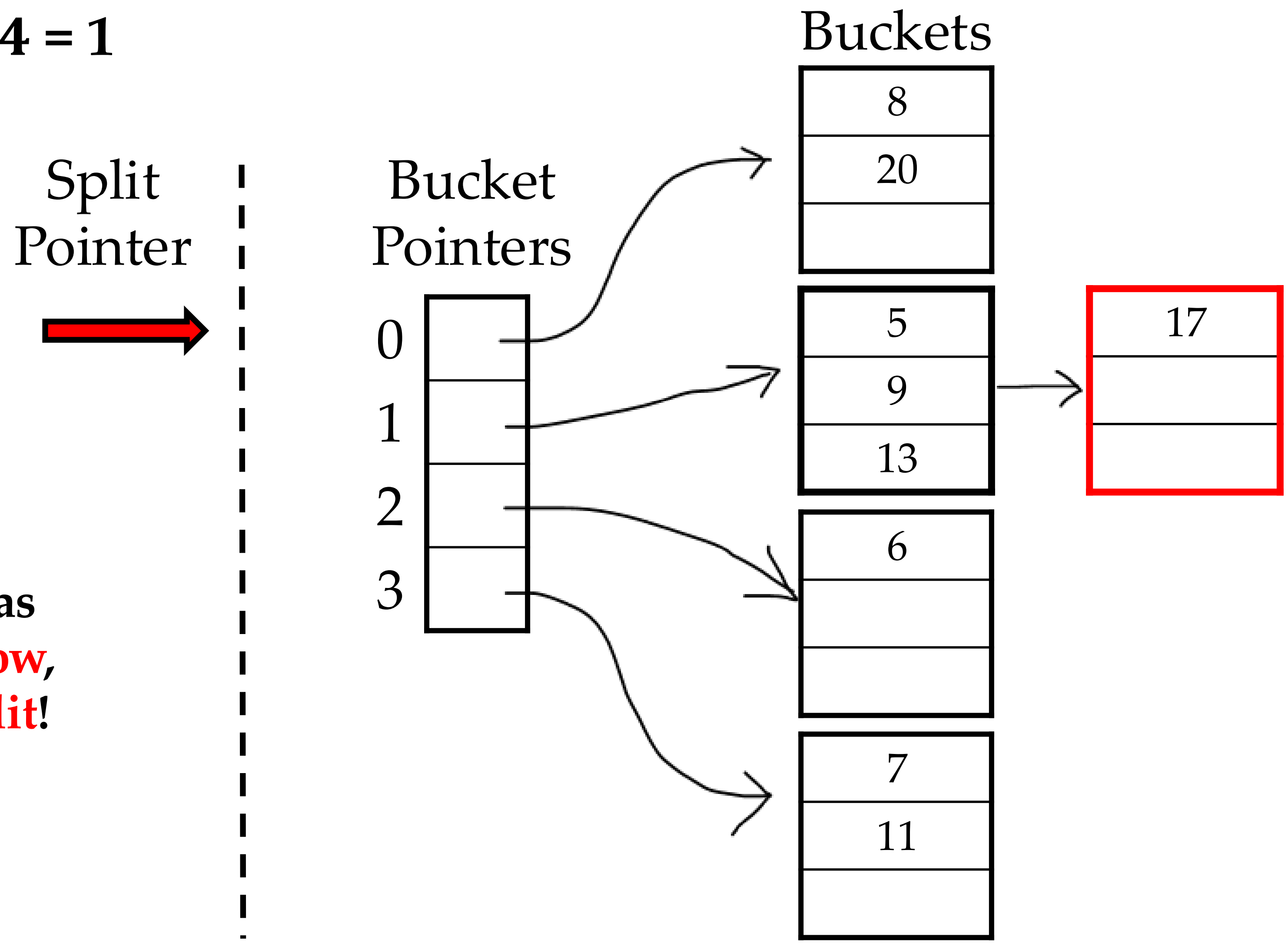
- Let's insert a **key = 17**:
  $\boldsymbol{h_1(17)} = \mathbf{17}\ \%\ \mathbf{4 = 1}$

The bucket is full, so we create a new bucket and link.



Linear Hashing

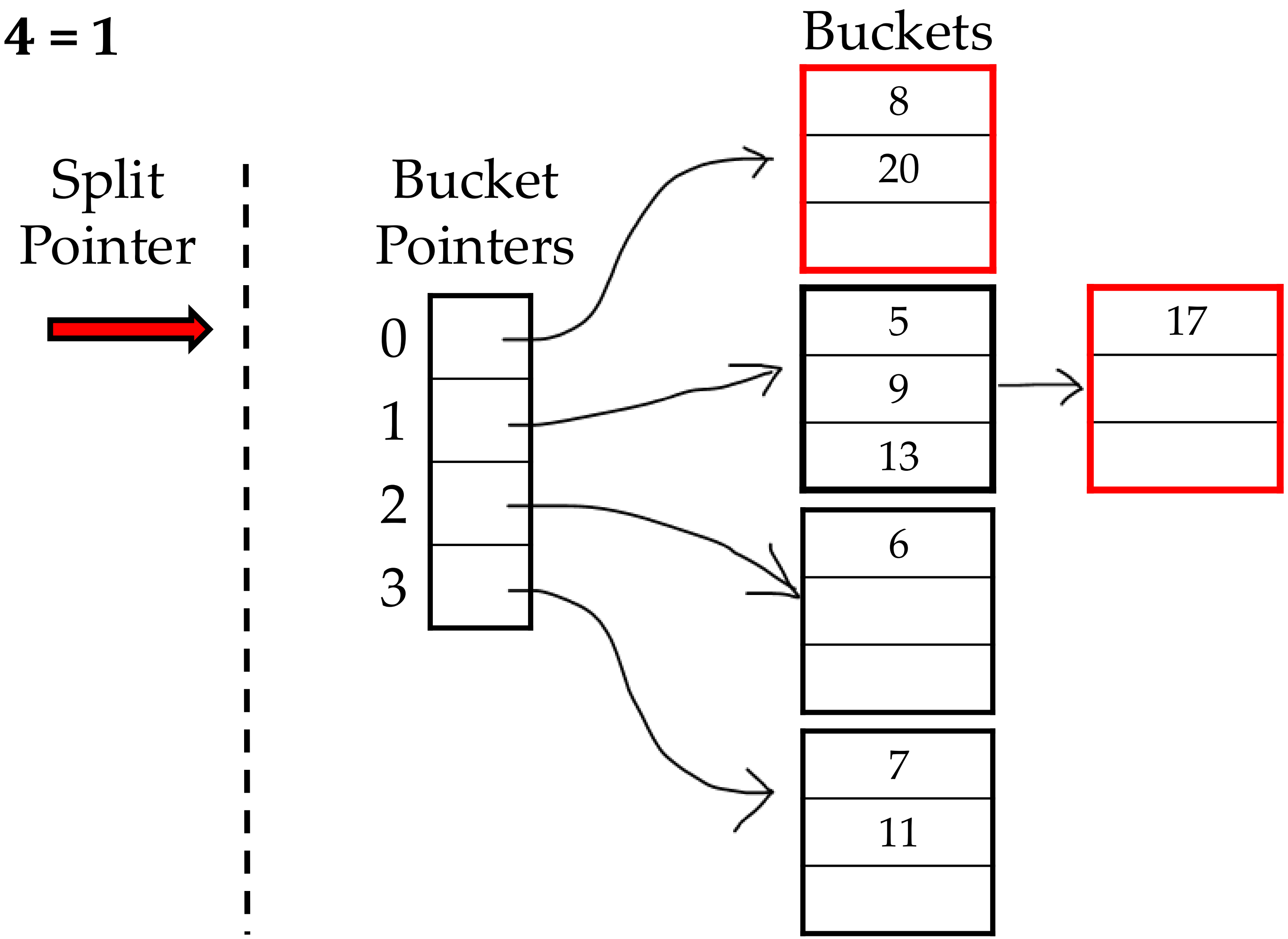
- Let's insert a **key = 17**:
  $\boldsymbol{h_1(17) = 17 \% 4 = 1}$



This situation has caused an overflow, so we need to split!

Linear Hashing

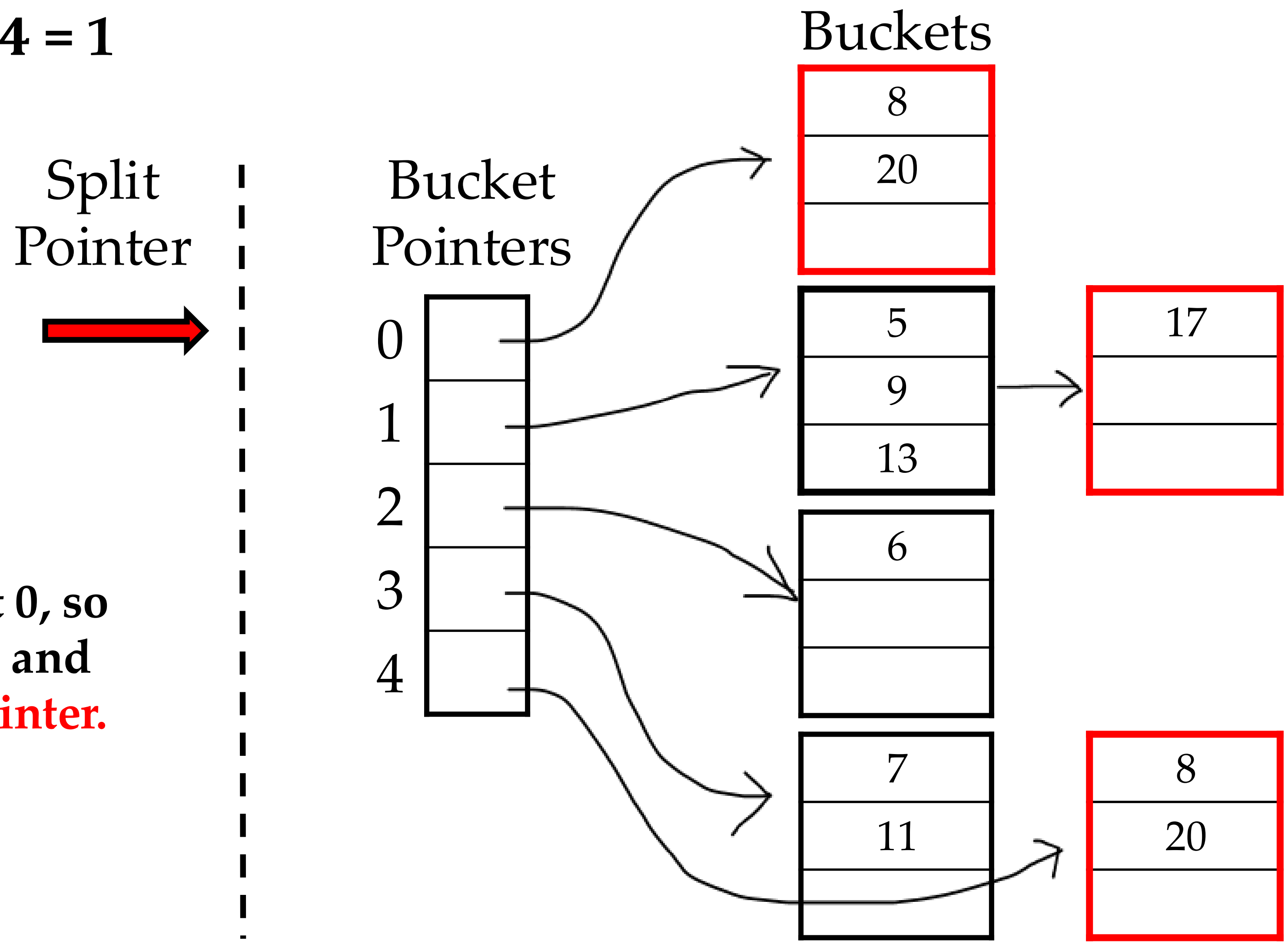
- Let's insert a **key = 17**:
  $\boldsymbol{h_1(17) = 17 \% 4 = 1}$



My split pointer is at 0, so I will split bucket 0.

Linear Hashing

- Let's insert a **key = 17**:
  $\boldsymbol{h_1(17)} = \mathbf{17}$ % $\mathbf{4 = 1}$



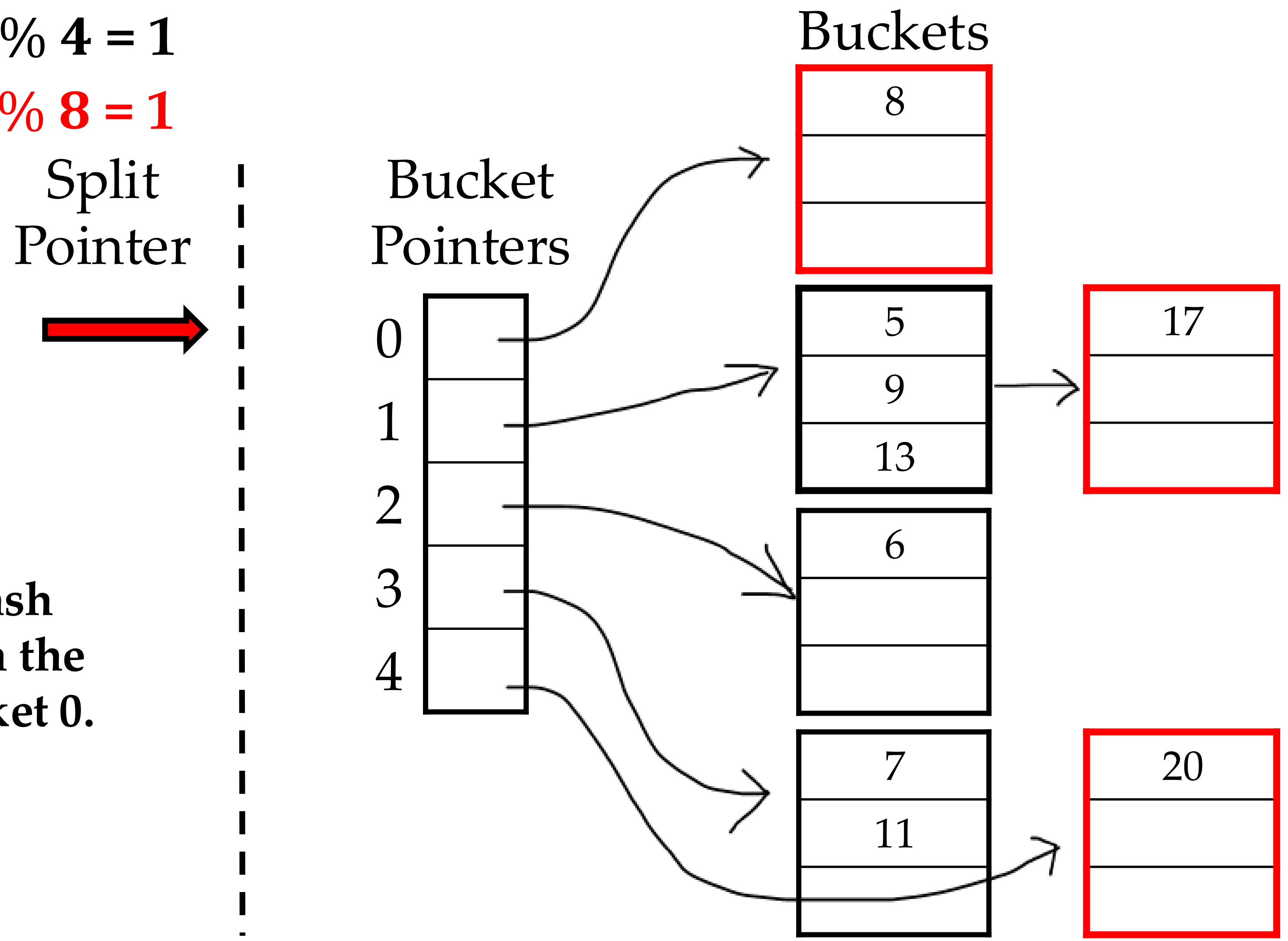
My split pointer is at 0, so I will split bucket 0, and add a new bucket pointer.

Linear Hashing

- Let's insert a **key = 17**:

  $\boldsymbol{h_1(key) = key \;\%\; 4 = 1}$

  $\boldsymbol{h_2(key) = key \;\%\; 8 = 1}$



Introduce a new hash function and Rehash the keys in original bucket 0.

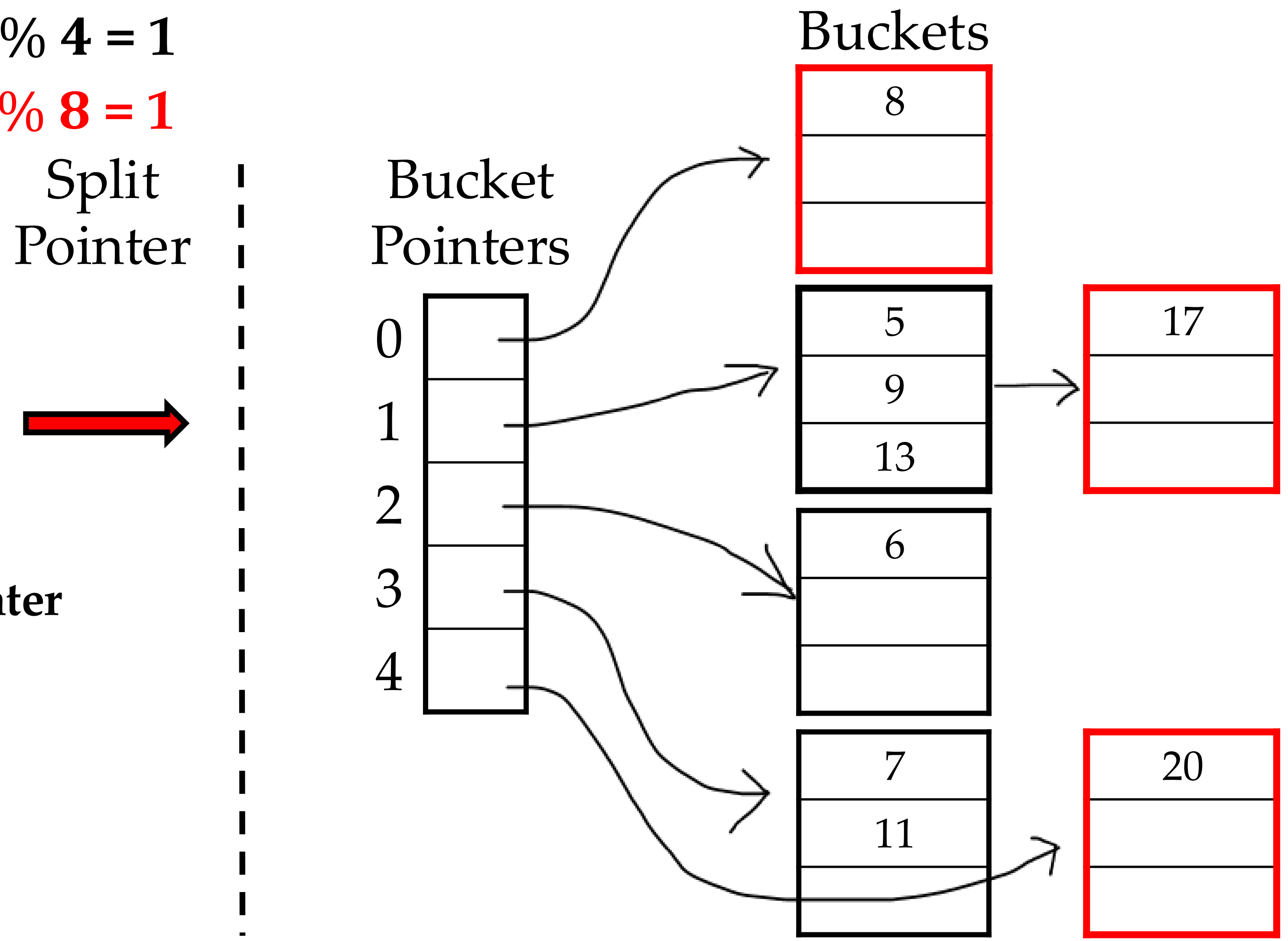
8 % 8 = 0
20 % 8 = 4

Linear Hashing

- Let's insert a **key = 17**:

  $\boldsymbol{h_1(key) = key \,\%\, 4 = 1}$

  $\boldsymbol{h_2(key) = key \,\%\, 8 = 1}$



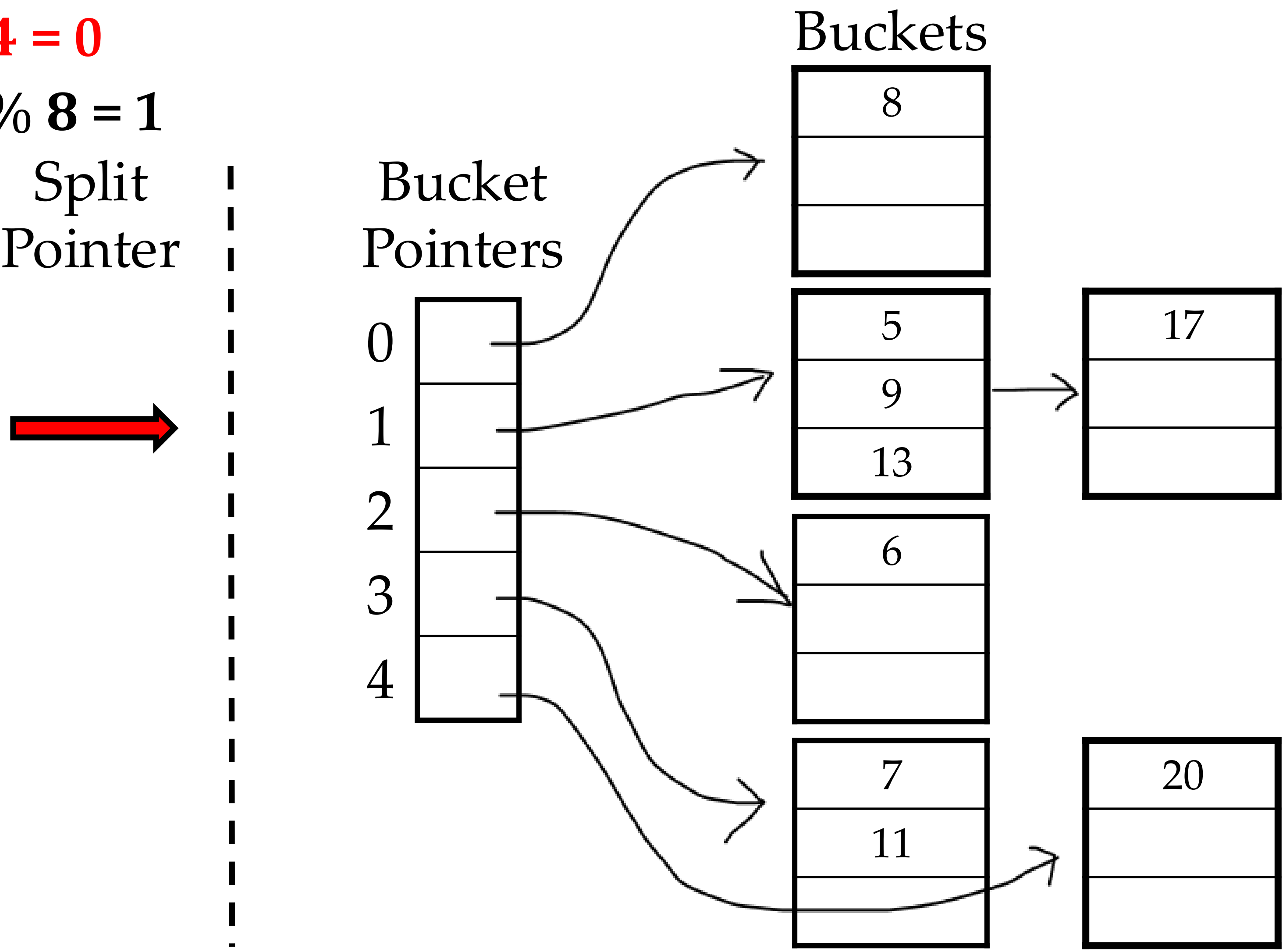
Move the split pointer

Linear Hashing

- Let's insert a **key = 16**:

  $\boldsymbol{h_1(16) = 16 \% 4 = 0}$

  $\boldsymbol{h_2(key) = key \% 8 = 1}$



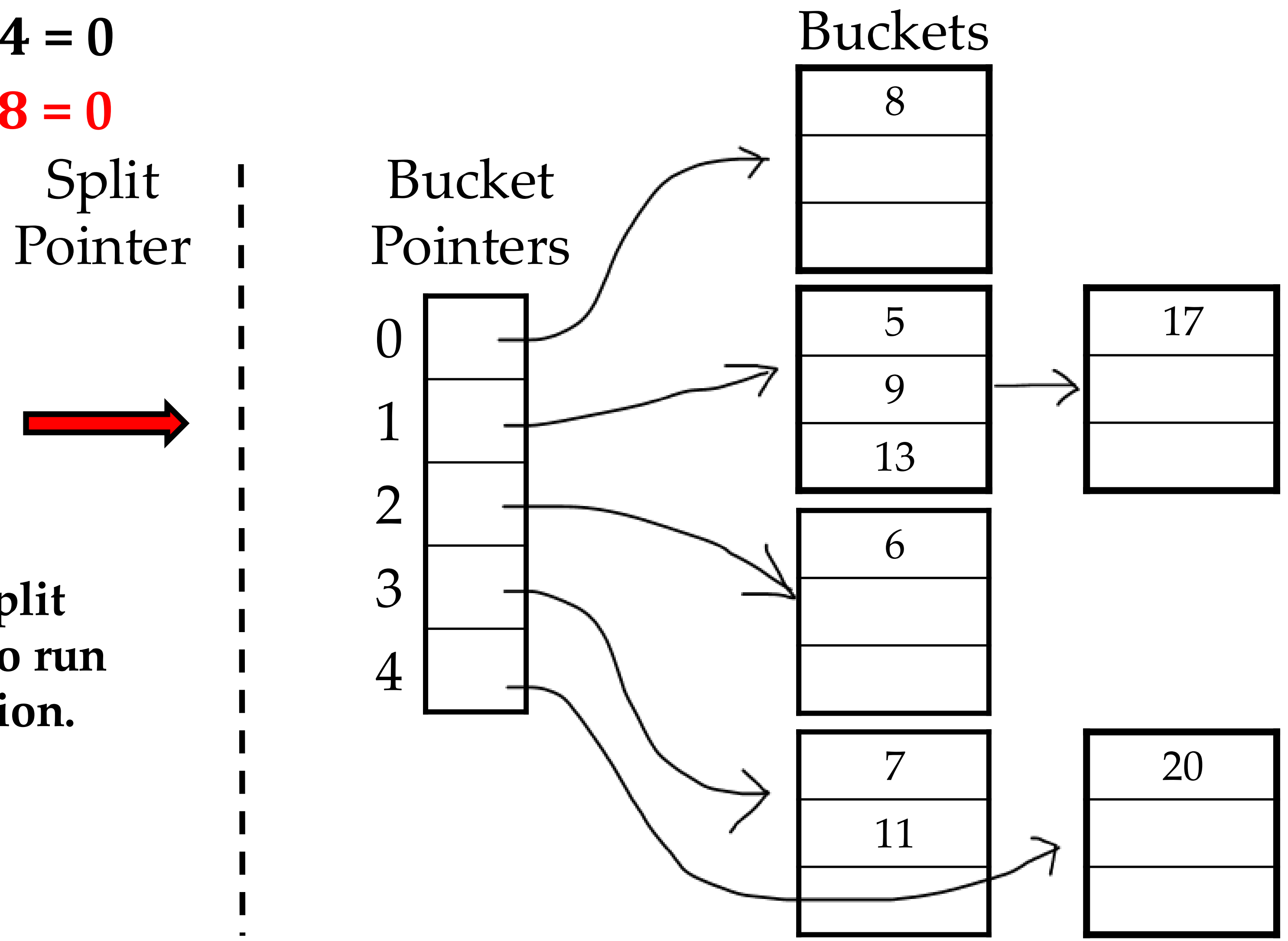
First try the hash function h_1(key).

Linear Hashing

- Let's insert a **key = 16**:

  $\boldsymbol{h_1(16) = 16 \% 4 = 0}$

  $\color{red}{\boldsymbol{h_2(16) = 16 \% 8 = 0}}$



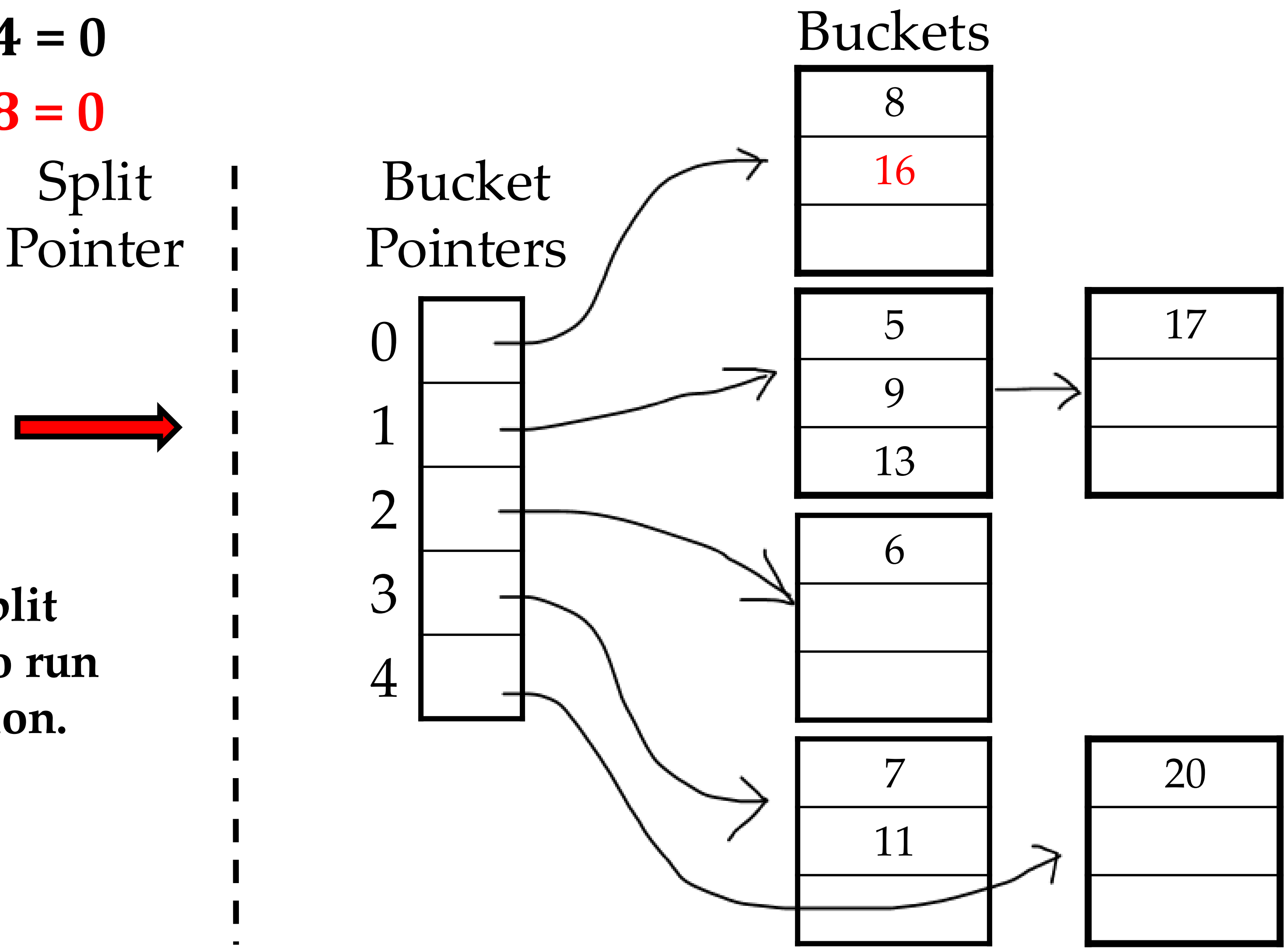
As 0 is above the split pointer, so we need to run the next hash function.

Linear Hashing

- Let's insert a **key = 16**:

  $\boldsymbol{h_1(16) = 16 \% 4 = 0}$

  $\boldsymbol{h_2(16) = 16 \% 8 = 0}$



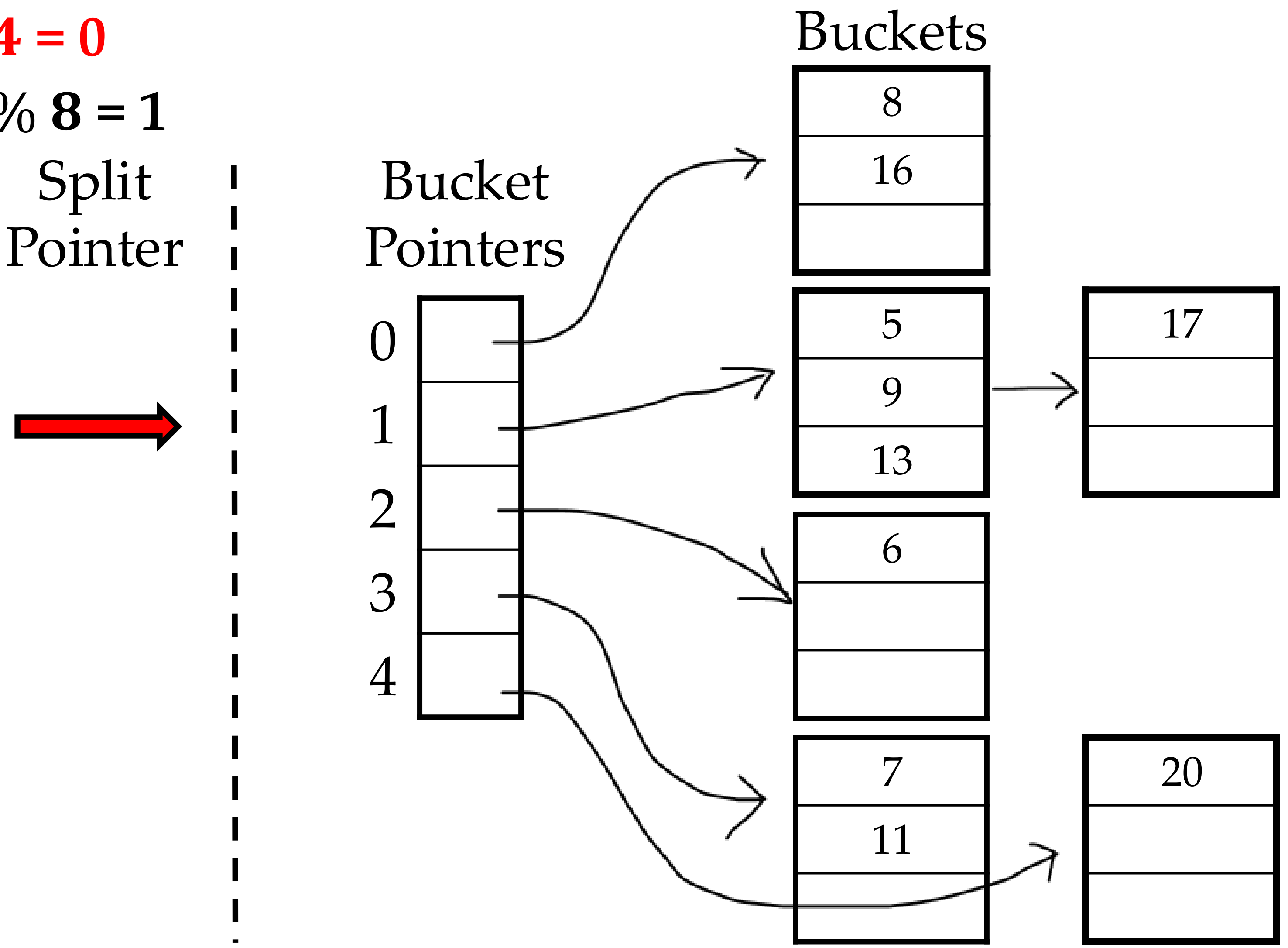
As 0 is above the split pointer, so we need to run the next hash function.

Linear Hashing

- Let's insert a **key = 12**:

  $\boldsymbol{h_1(12) = 12\ \%\ 4 = 0}$

  $\boldsymbol{h_2(key) = key\ \%\ 8 = 1}$



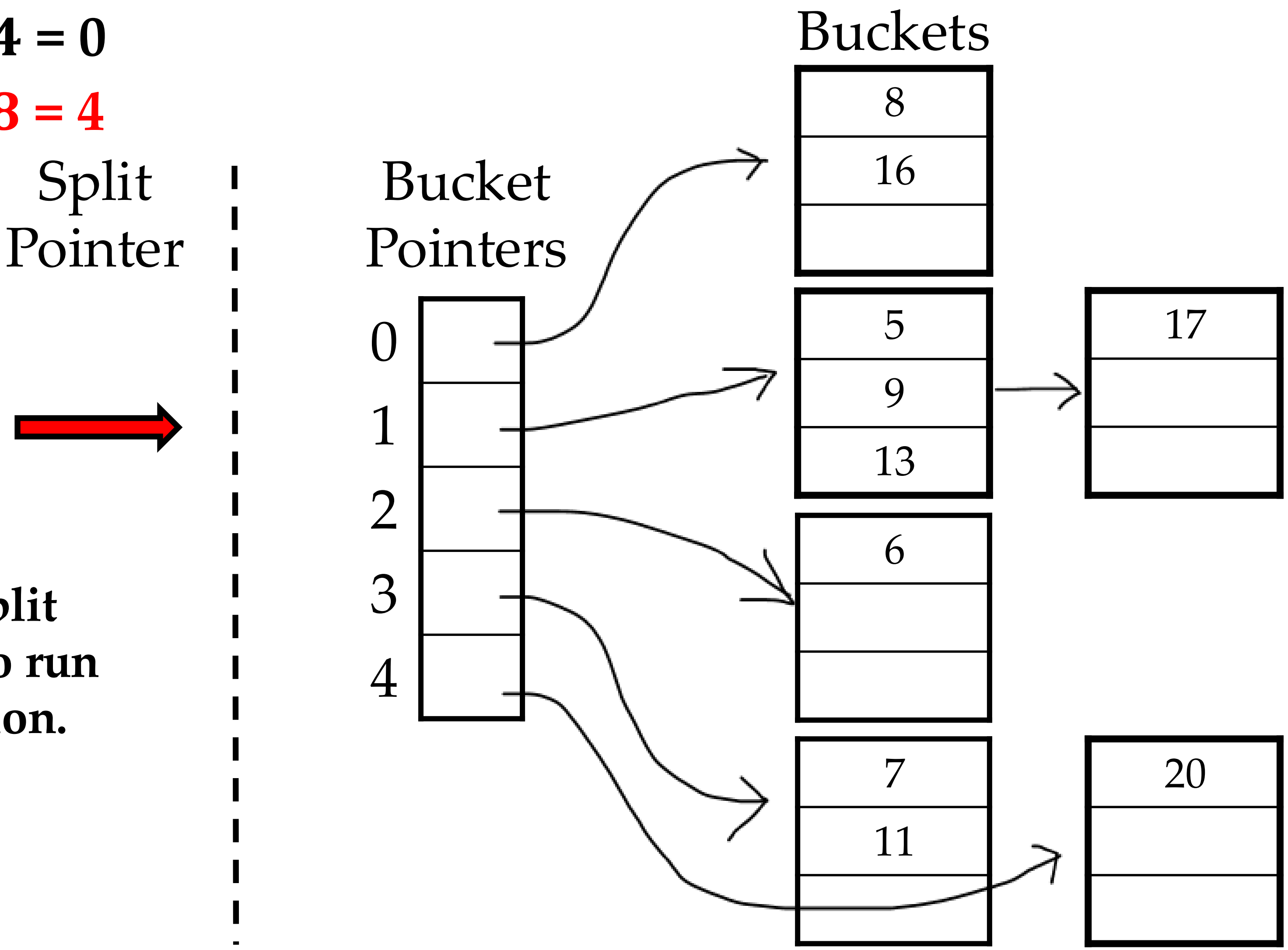
First try the hash function h_1(key).

Linear Hashing

- Let's insert a **key = 12**:

  $\boldsymbol{h_1(12) = 12 \% 4 = 0}$

  $\boldsymbol{h_2(12) = 12 \% 8 = 4}$



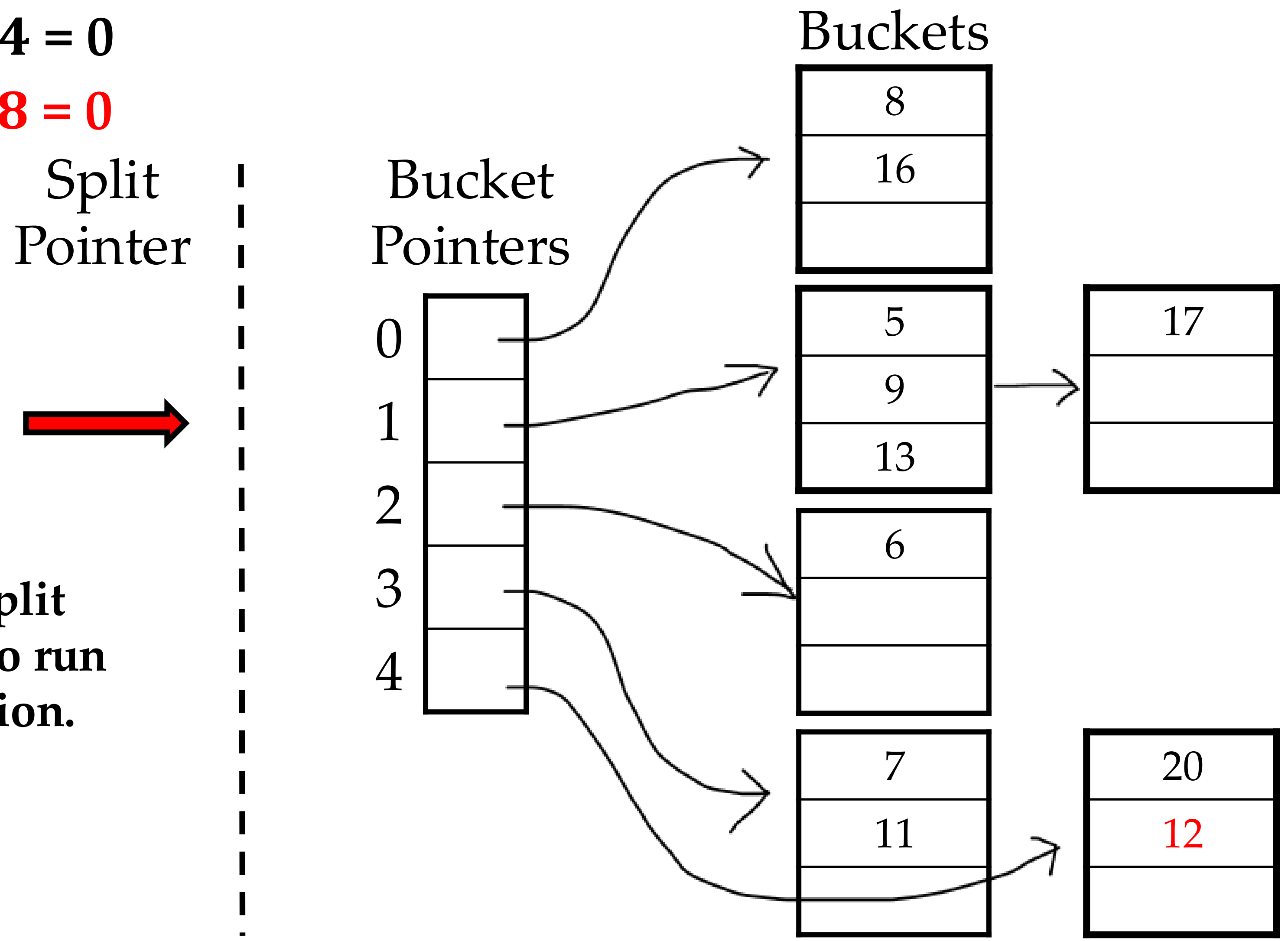
As 0 is above the split pointer, so we need to run the next hash function.

Linear Hashing

- Let's insert a **key = 12**:

  $\boldsymbol{h_1(12) = 12 \% 4 = 0}$

  $\boldsymbol{h_2(12) = 12 \% 8 = 0}$



As 0 is above the split pointer, so we need to run the next hash function.

Linear Hashing