EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Maps & Hashing

Readings:

- Weiss 5.1 – 5.4
- **Cormen 11.1 – 11.4**

The Map ADT

Maps: Introduction

- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be (keys are a Set!).

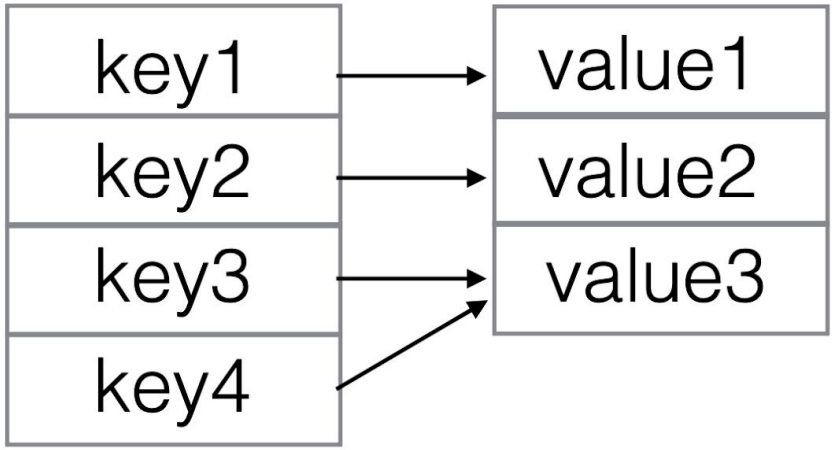
Maps: Introduction

- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be (keys are a Set!).
- Two operations:
  - `get(key)` returns the value associated with this key
  - `put(key, value)` (overwrites existing keys)

The Map ADT

Maps: Introduction

- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be (keys are a Set!).
- Two operations:
  - `get(key)` returns the value associated with this key
  - `put(key, value)` (overwrites existing keys)



Maps are known as **associative containers** in STL (as opposed to the sequential ones like vector or list)

Goal: implement both get() and put() in average-case **O(1)**

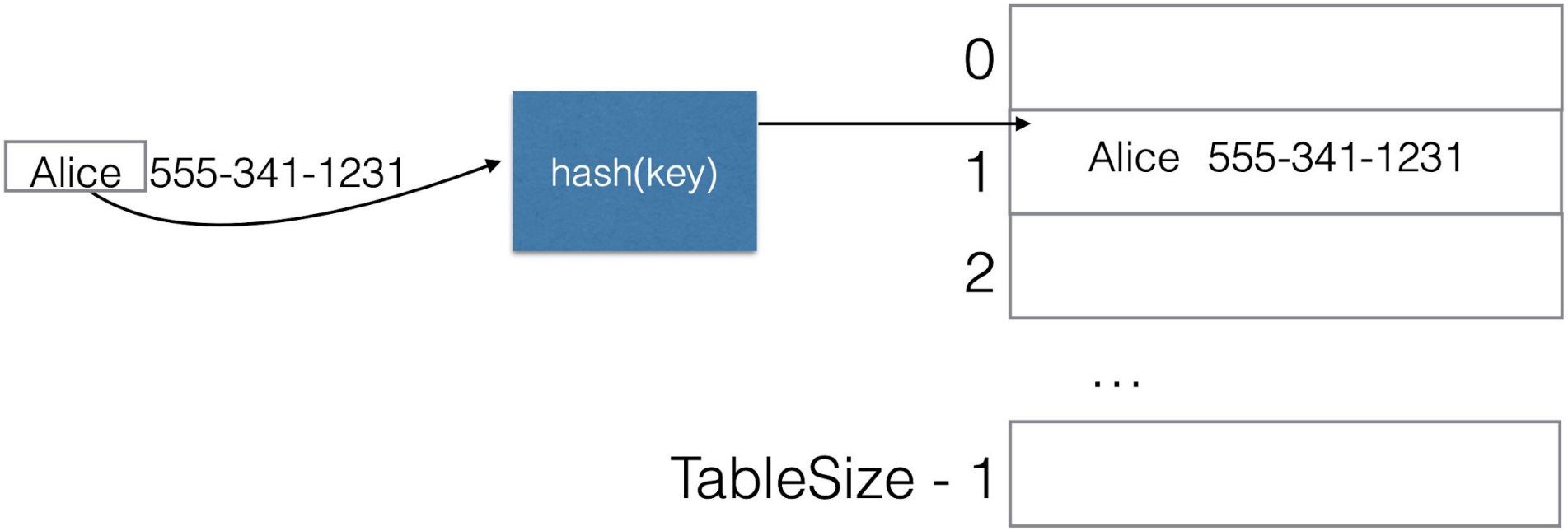
How to get O(1) for maps?

- Define a table (an array) of some length *TableSize.*

- Define a function `hash(key)` that maps key objects to an integer index in the range *0 … TableSize -1*

How to get O(1) for maps?

- Define a table (an array) of some length *TableSize.*
- Define a function `hash(key)` that maps key objects to an integer index in the range *0 … TableSize -1*

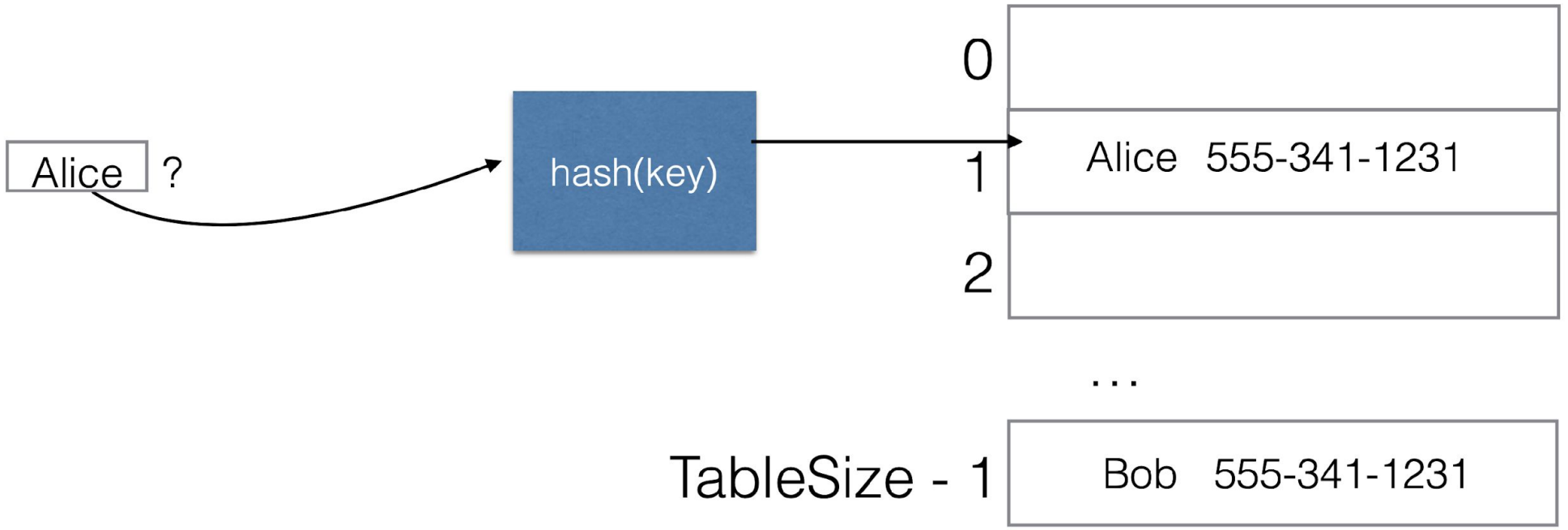


How to get O(1) for maps?

- Lookup/`get`: Just hash the key to find the index.
- Assuming `hash(key)` takes constant time, `get` and `put` run in O(1).

How to get O(1) for maps?

- Lookup/`get`: Just hash the key to find the index.
- Assuming `hash(key)` takes constant time, `get` and `put` run in O(1).

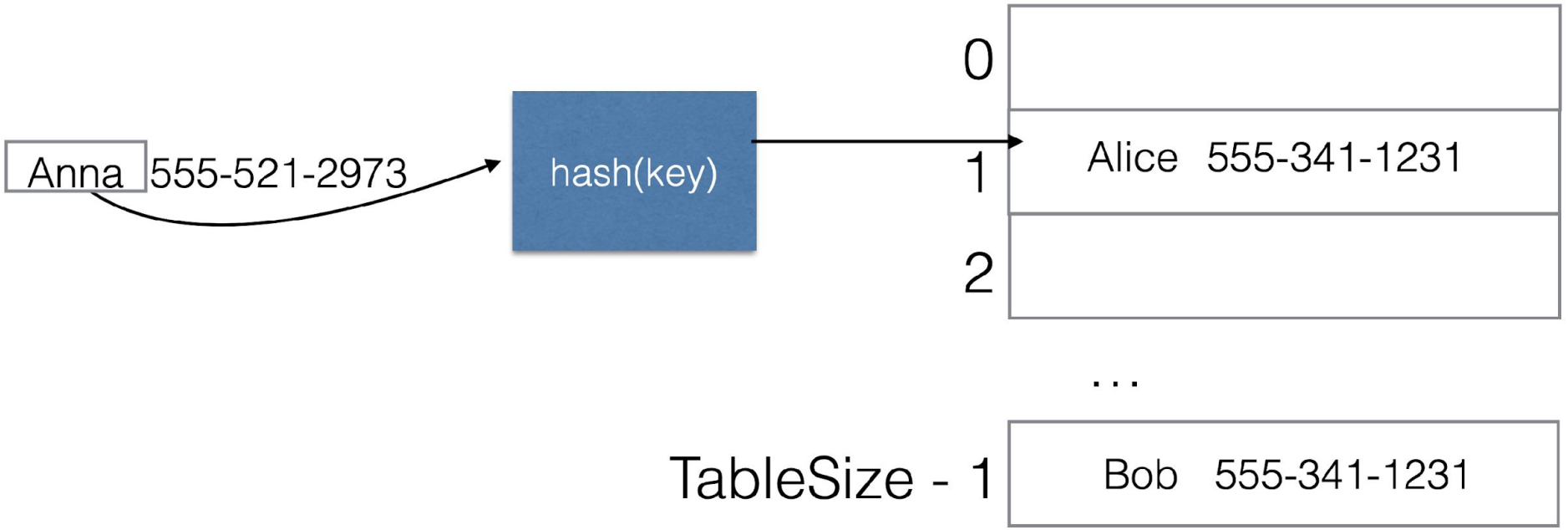


Hash Collisions

- Problem: There is an infinite number of keys, but only *TableSize* entries in the array.
  - How do we deal with collisions? (new item hashes to an array cell that is already occupied)

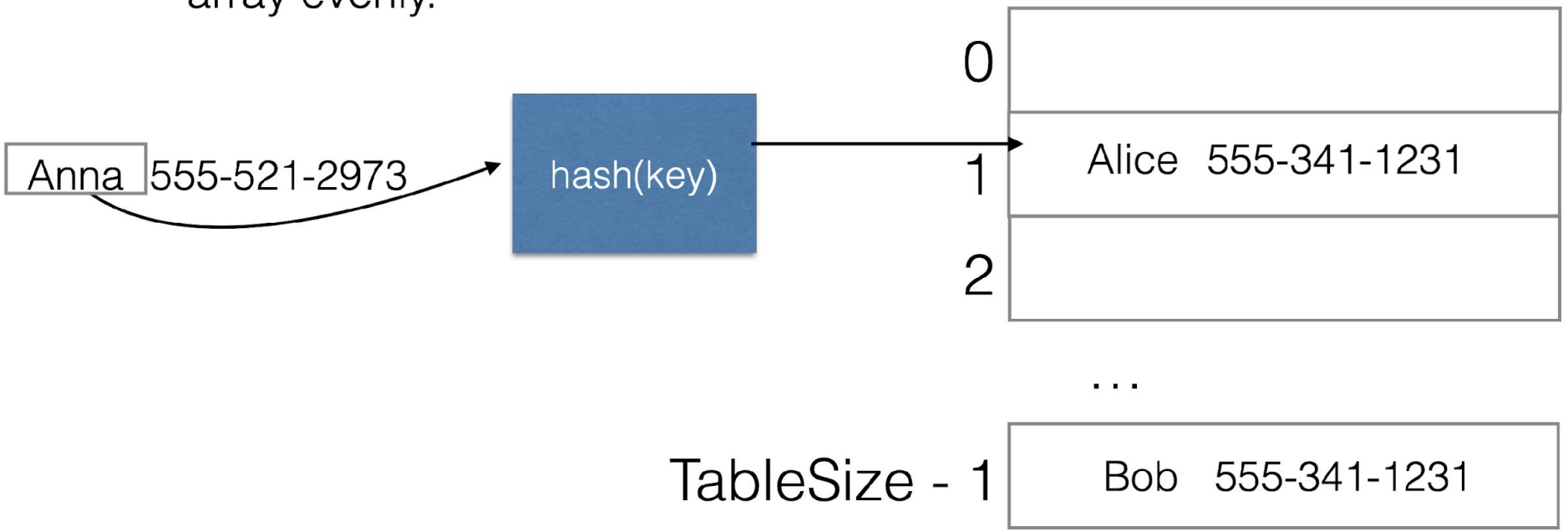
Hash Collisions

- Problem: There is an infinite number of keys, but only *TableSize* entries in the array.
  - How do we deal with collisions? (new item hashes to an array cell that is already occupied)



Hash Collisions

- Problem: There is an infinite number of keys, but only *TableSize* entries in the array.
  - How do we deal with collisions? (new item hashes to an array cell that is already occupied)
  - Also: Need to find a hash function that distributes items in the array evenly.

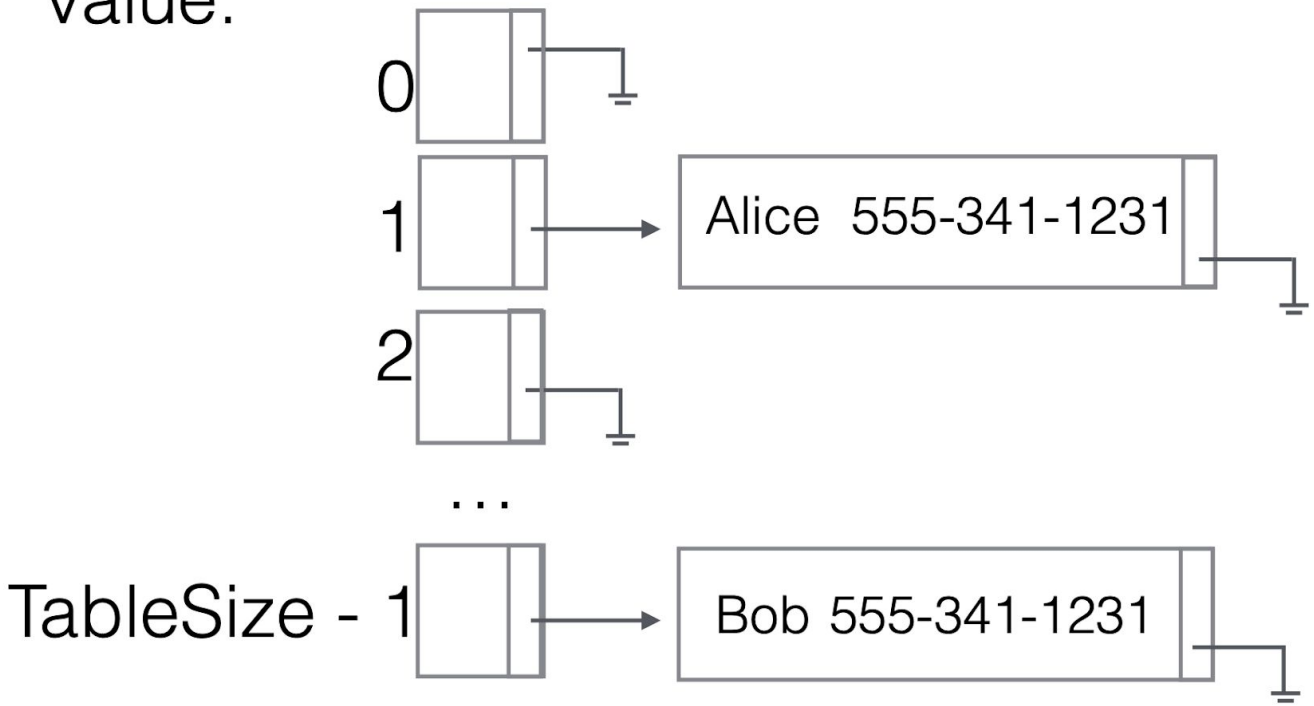


Hash Collisions Method 1: Separate Chaining

- Keep all items whose key hashes to the same value on a linked list.

Hash Collisions Method 1: Separate Chaining

- Keep all items whose key hashes to the same value on a linked list.
- Can think of each list as a *bucket* defined by the hash value.

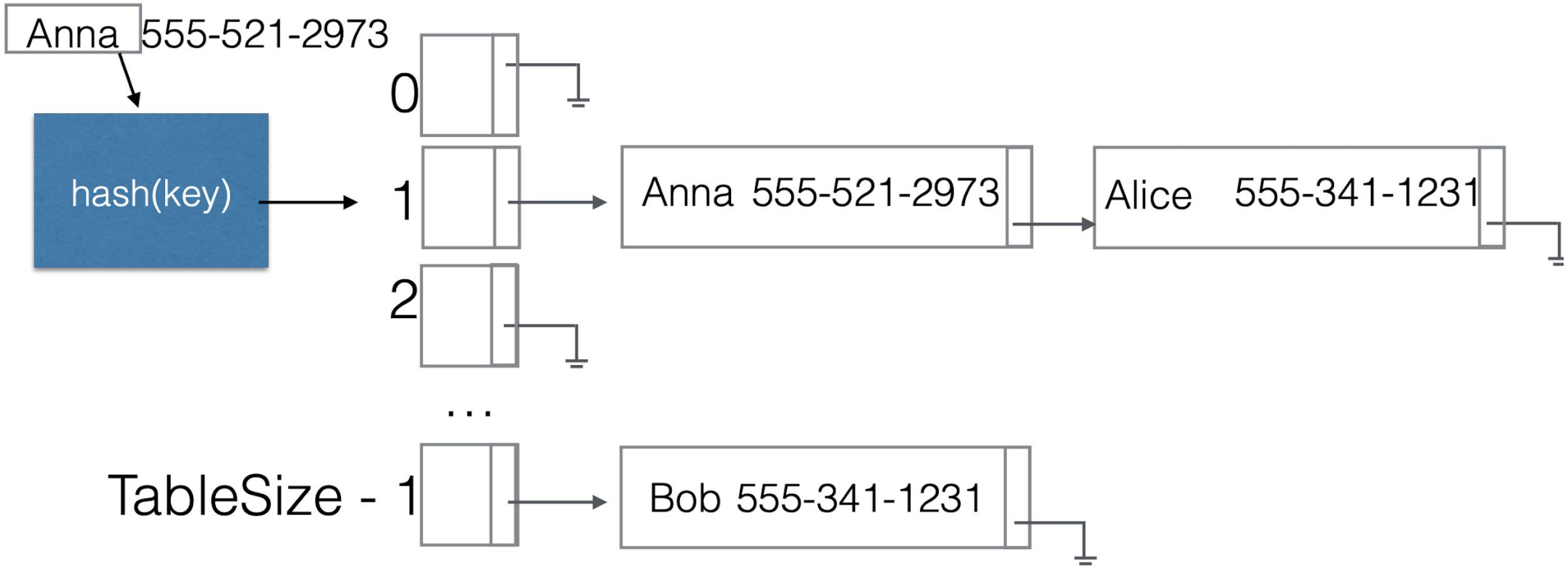


Hash Collisions Method 1: Separate Chaining

- To insert a new key in cell that's already occupied prepend to the list.

Hash Collisions Method 1: Separate Chaining

- To insert a new key in cell that's already occupied prepend to the list.



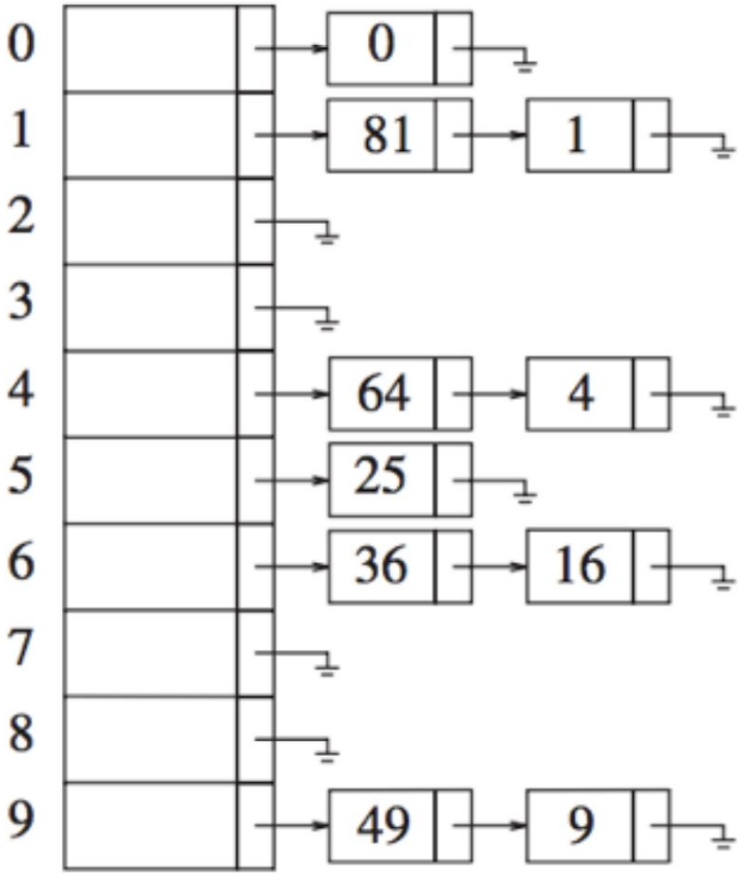
Running Time Analysis

Separate Chaining

- Time to find a key = time to compute hash function
  + time to traverse the linked list.
- Assume hash functions computed in O(1).
- How many elements do we expect in a list on average?

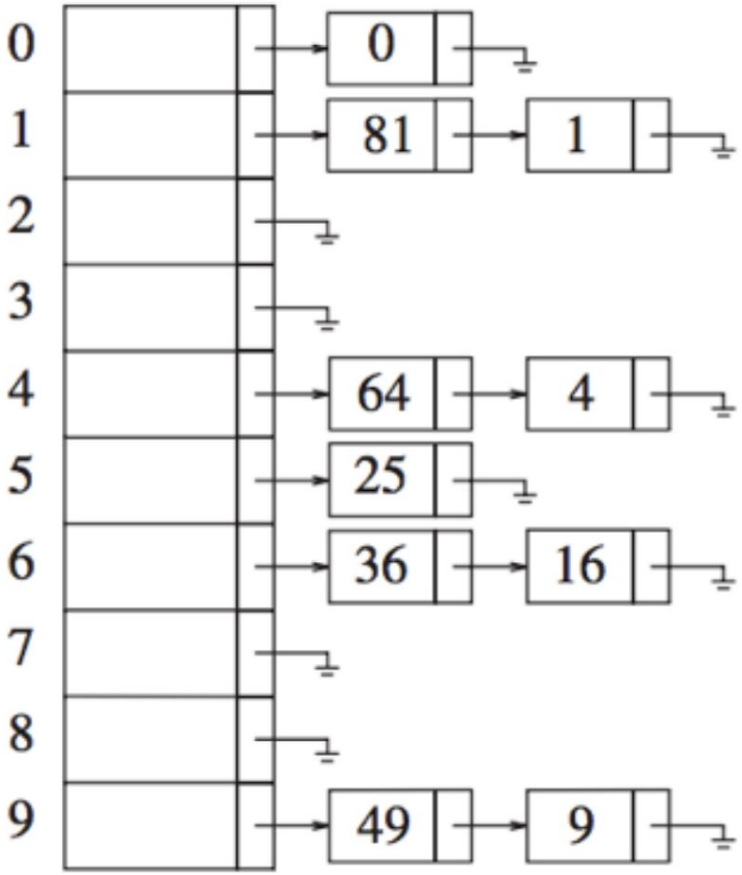
Load Factor

Separate Chaining



Load Factor

Separate Chaining



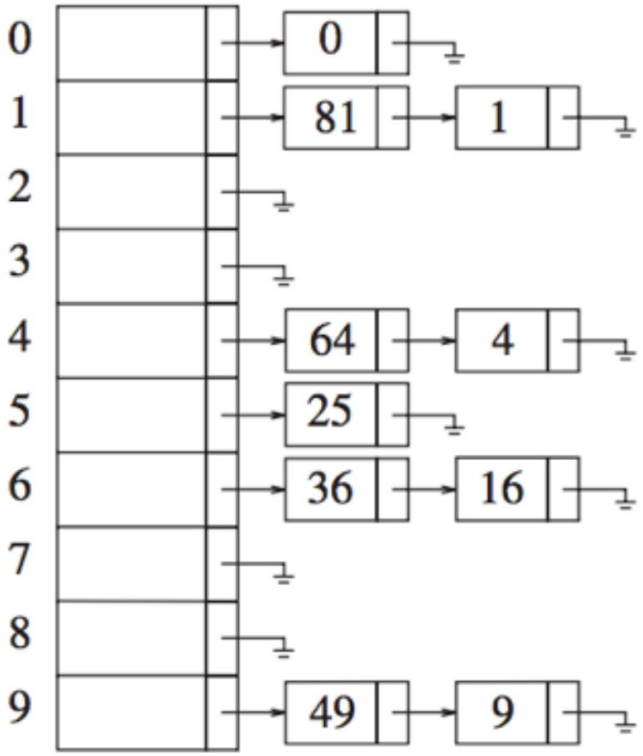
- Let *N* be the number of keys in the table.
- Define the load factor as

$$\lambda = \frac{N}{TableSize}$$

- The average length of a list is λ .

Running time Using Load Factor

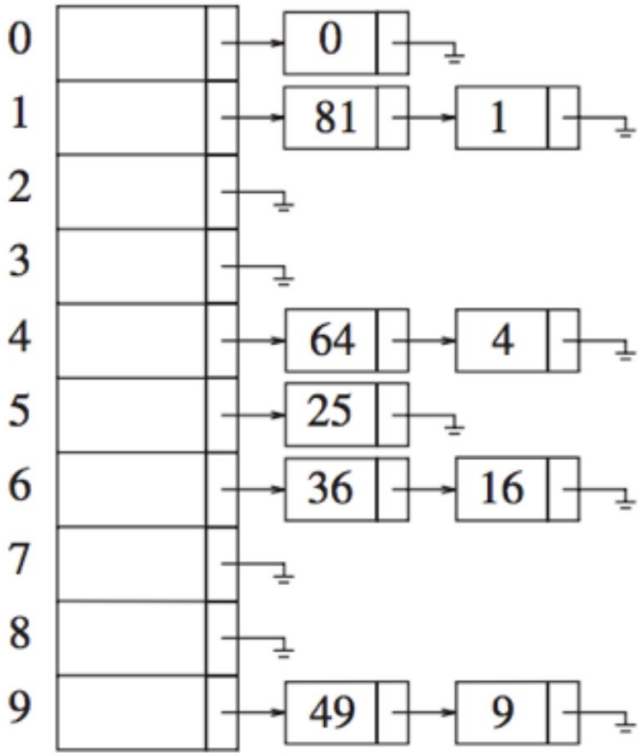
Separate Chaining



- If lookup fails (table miss):
  - Need to search all λ nodes in the list for this hash bucket.

Running time Using Load Factor

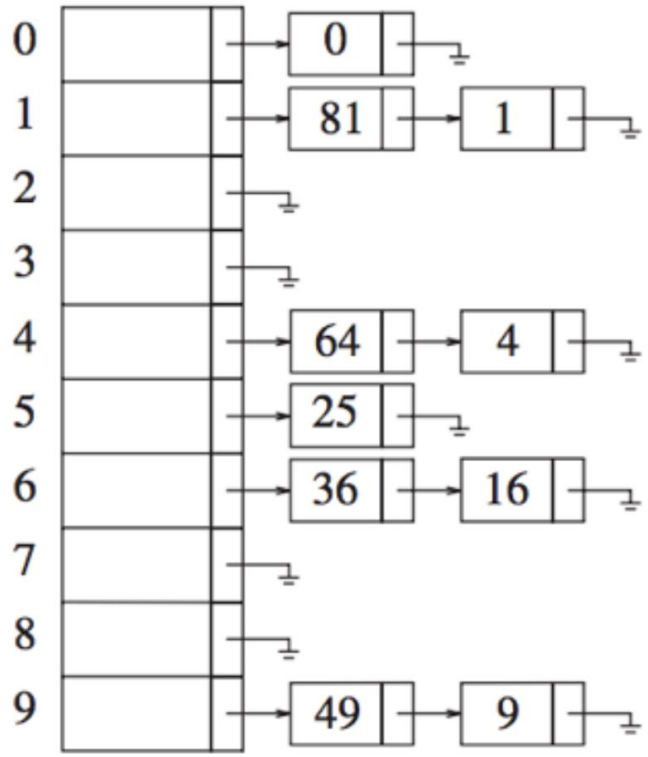
Separate Chaining



- If lookup fails (table miss):
  - Need to search all λ nodes in the list for this hash bucket.
- If lookup succeeds (table hit):
  - There will be about λ other nodes in the list.
  - On average we search half the list and the target key, so we touch $\lambda/2 + 1$ nodes.

Running time Using Load Factor

Separate Chaining



- If lookup fails (table miss):
  - Need to search all λ nodes in the list for this hash bucket.
- If lookup succeeds (table hit):
  - There will be about λ other nodes in the list.
  - On average we search half the list and the target key, so we touch $\lambda/2 + 1$ nodes.

Design rule: keep $\lambda \approx 1$. If load becomes too high increase table size (rehash).

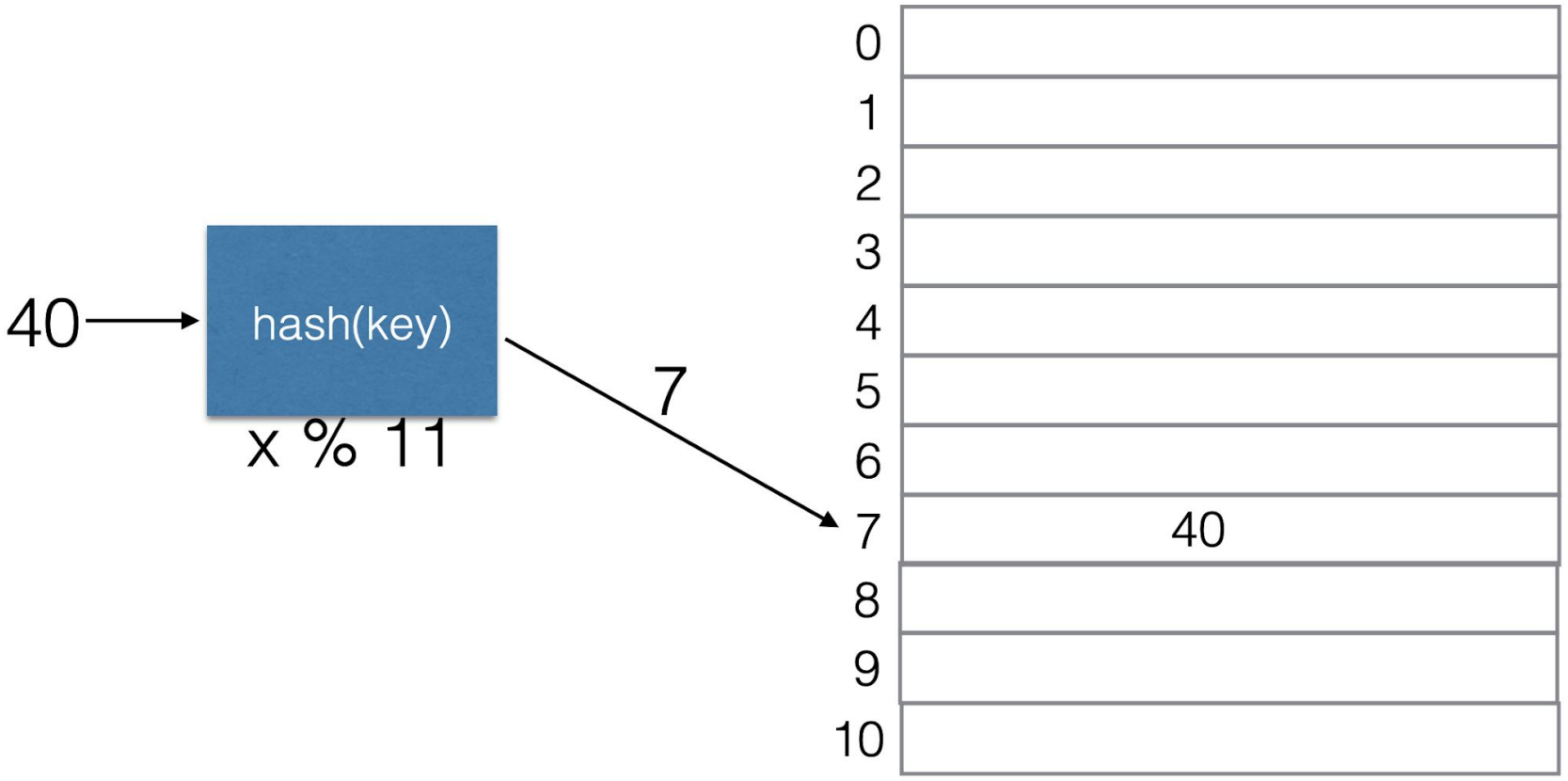
Separate Chaining: Downsides

- Requires allocation of new list nodes, which introduces overhead.
- Requires more code because it requires a linked list data structure in addition to the hash table itself.

Probing

Hash Collisions Method 2: Open Addressing

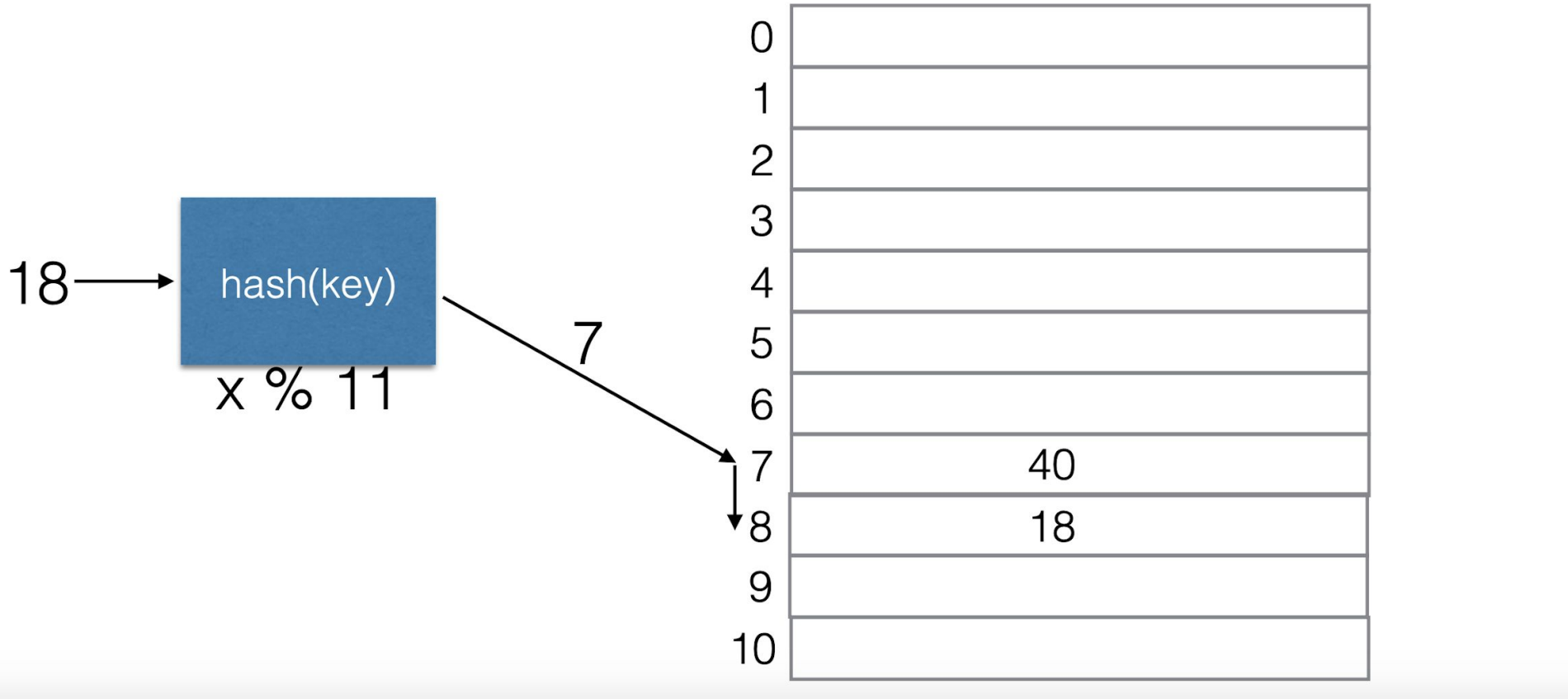
- When a collision occurs put item in an empty cell of the hash table itself.



Probing

Hash Collisions Method 2: Open Addressing

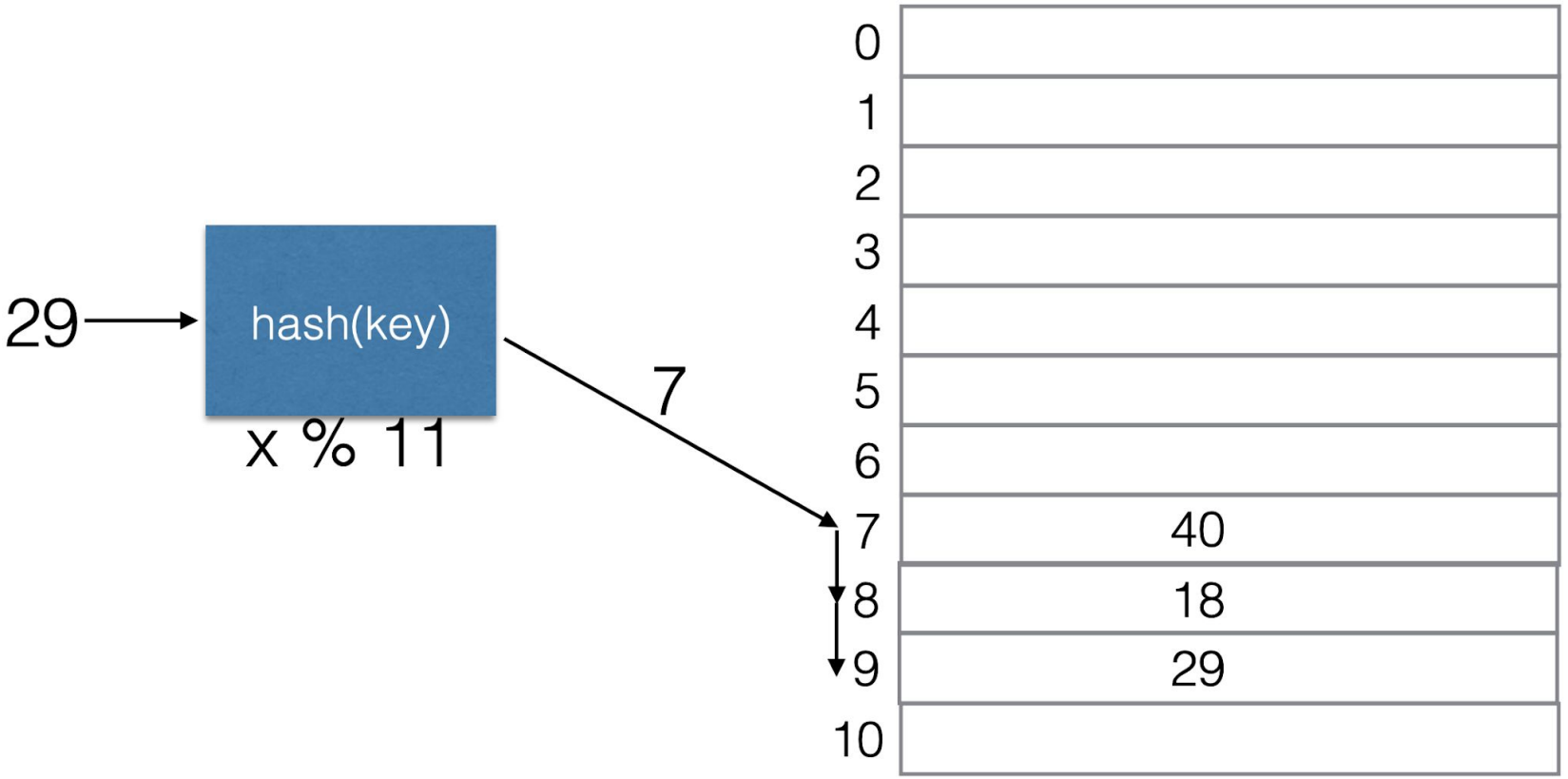
- When a collision occurs put item in an empty cell of the hash table itself.



Probing

Hash Collisions Method 2: Open Addressing

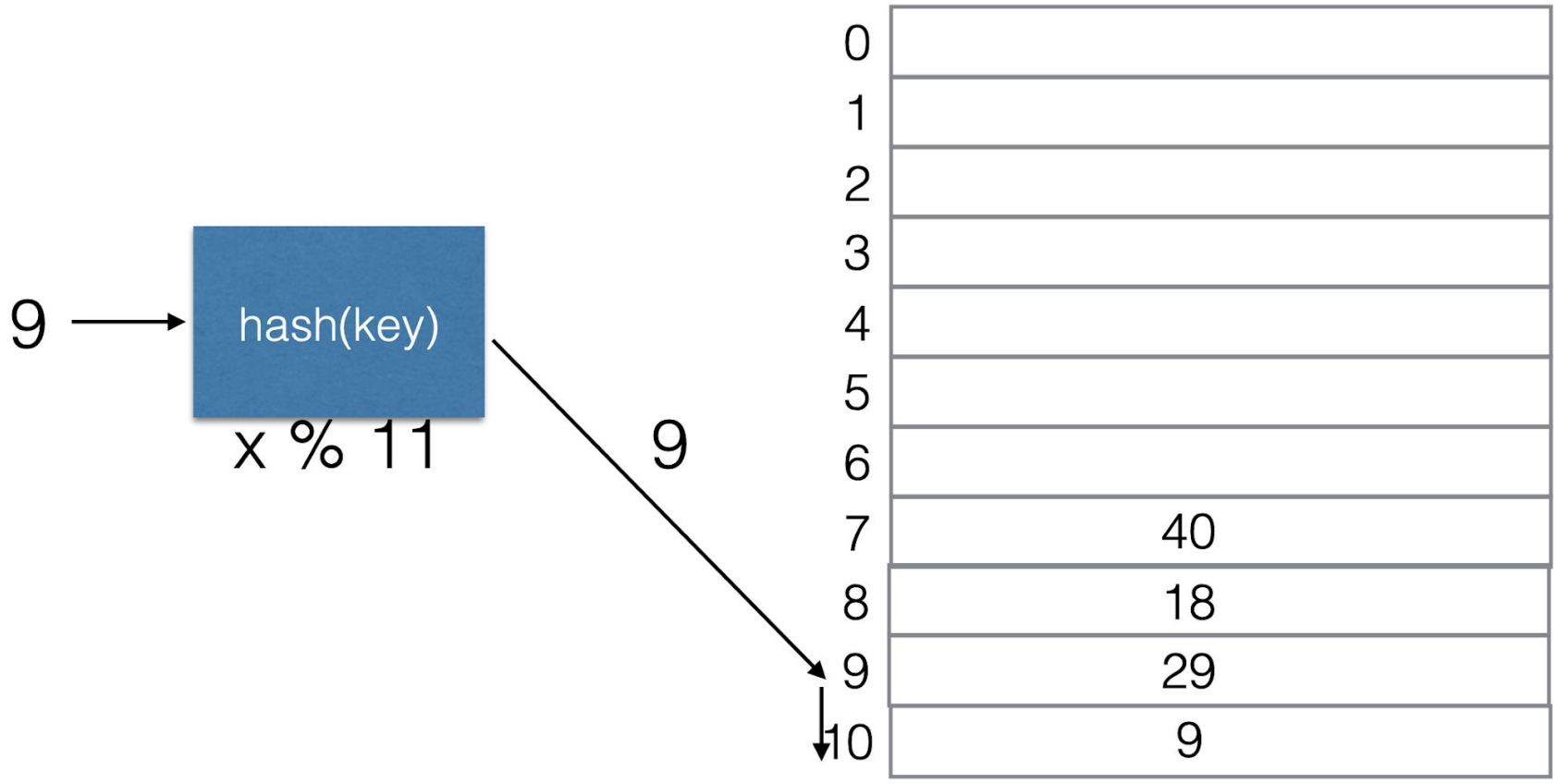
- When a collision occurs put item in an empty cell of the hash table itself.



Probing

Hash Collisions Method 2: Open Addressing

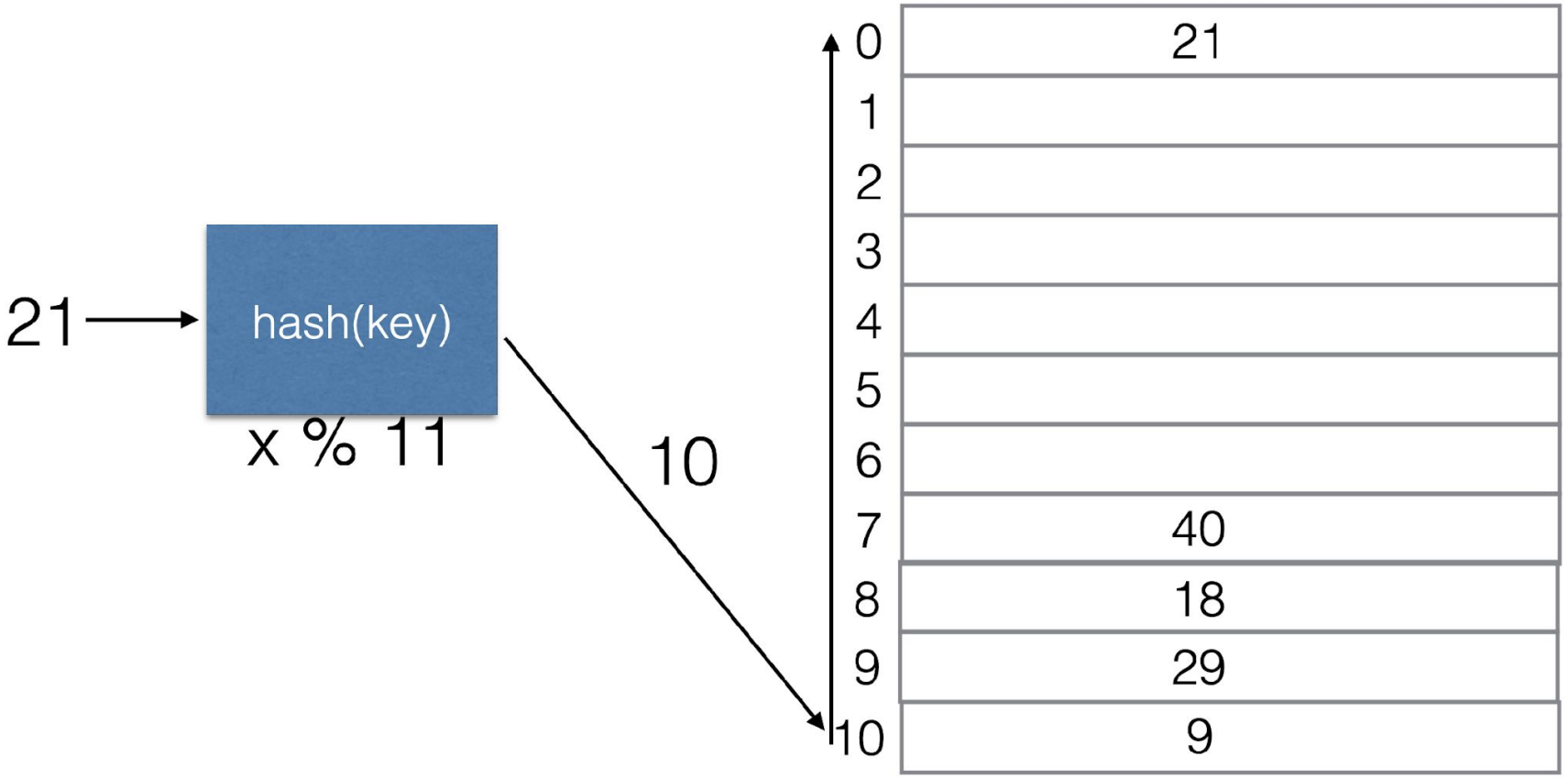
- When a collision occurs put item in an empty cell of the hash table itself.



Probing

Hash Collisions Method 2: Open Addressing

- When a collision occurs put item in an empty cell of the hash table itself.



Open Addressing: Probing

- To insert an item, we probe other table cells in a systematic way until an empty cell is found.

Open Addressing: Probing

- To insert an item, we probe other table cells in a systematic way until an empty cell is found.
- To look up a key, we probe in a systematic way until the key is found.

Open Addressing: Probing

- To insert an item, we probe other table cells in a systematic way until an empty cell is found.
- To look up a key, we probe in a systematic way until the key is found.
- Different strategies to determine the next cell
  - Example: Just try cells sequentially (with wraparound).

Open Addressing: Probing

- Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is

$$(hash(x) + f(i)) \,\%\, TableSize$$

Open Addressing: Probing

- Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is

$$(hash(x) + f(i))\ \%\ TableSize$$

- Linear Probing (previous example):
  - f(i) is some linear function of i, usually $f(i) = i$.

If *hash(x) = 7*, try cell *7* first, then try cell *7+f(1)=8*, cell *7+f(2)=9*, cell *7+f(3)=10*, …

Open Addressing: Probing

- Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is

$$(hash(x) + f(i)) \,\%\, TableSize$$

- Linear Probing (previous example):
  - f(i) is some linear function of i, usually $f(i) = i$.

If *hash(x) = 7*, try cell *7* first, then try cell *7+f(1)=8*, cell *7+f(2)=9*, cell *7+f(3)=10*, …

- Quadratic probing $f(i) = i^2$

Open Addressing: Probing

- Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is

$$(hash(x) + f(i)) \,\%\, TableSize$$

- Linear Probing (previous example):
  - f(i) is some linear function of i, usually $f(i) = i$.

If *hash(x) = 7*, try cell *7* first, then try cell *7+f(1)=8*, cell *7+f(2)=9*, cell *7+f(3)=10*, …

- Quadratic probing $f(i) = i^2$
- Double hashing $f(i) = i \cdot hash_2(x)$

Linear Probing f(i) = i

Linear Probing

- Can always find an empty cell (if there is space in the table).

Linear Probing f(i) = i

Linear Probing

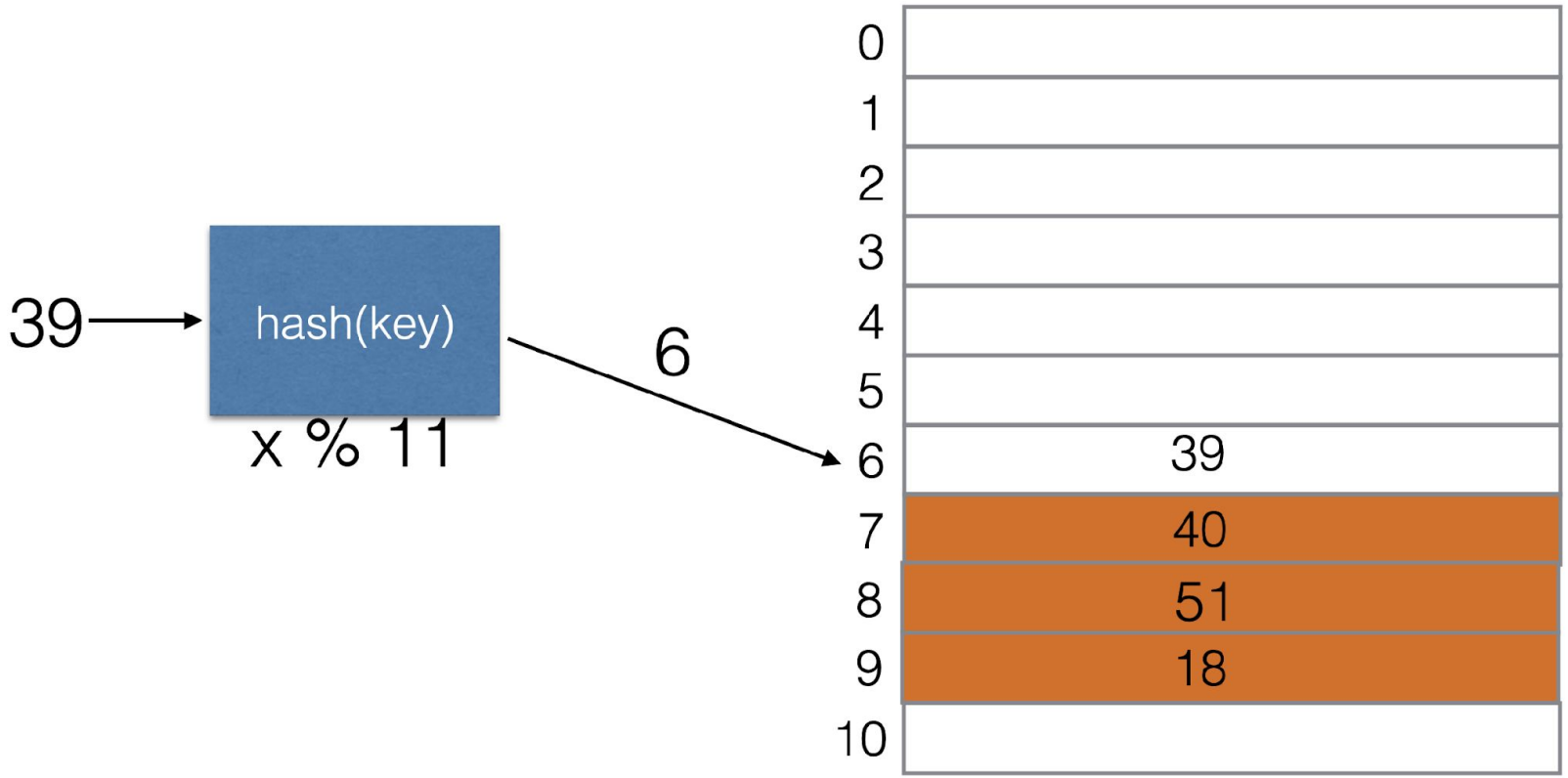
- Can always find an empty cell (if there is space in the table).
- Problem: **Primary Clustering.**
  - Full cells tend to cluster, with no free cells in between.

Linear Probing Downsides

- Can always find an empty cell (if there is space in the table).
- Problem: **Primary Clustering.**
  - Full cells tend to cluster, with no free cells in between.
  - Time required to find an empty cell can become very large if the table is almost full (λ is close to 1).

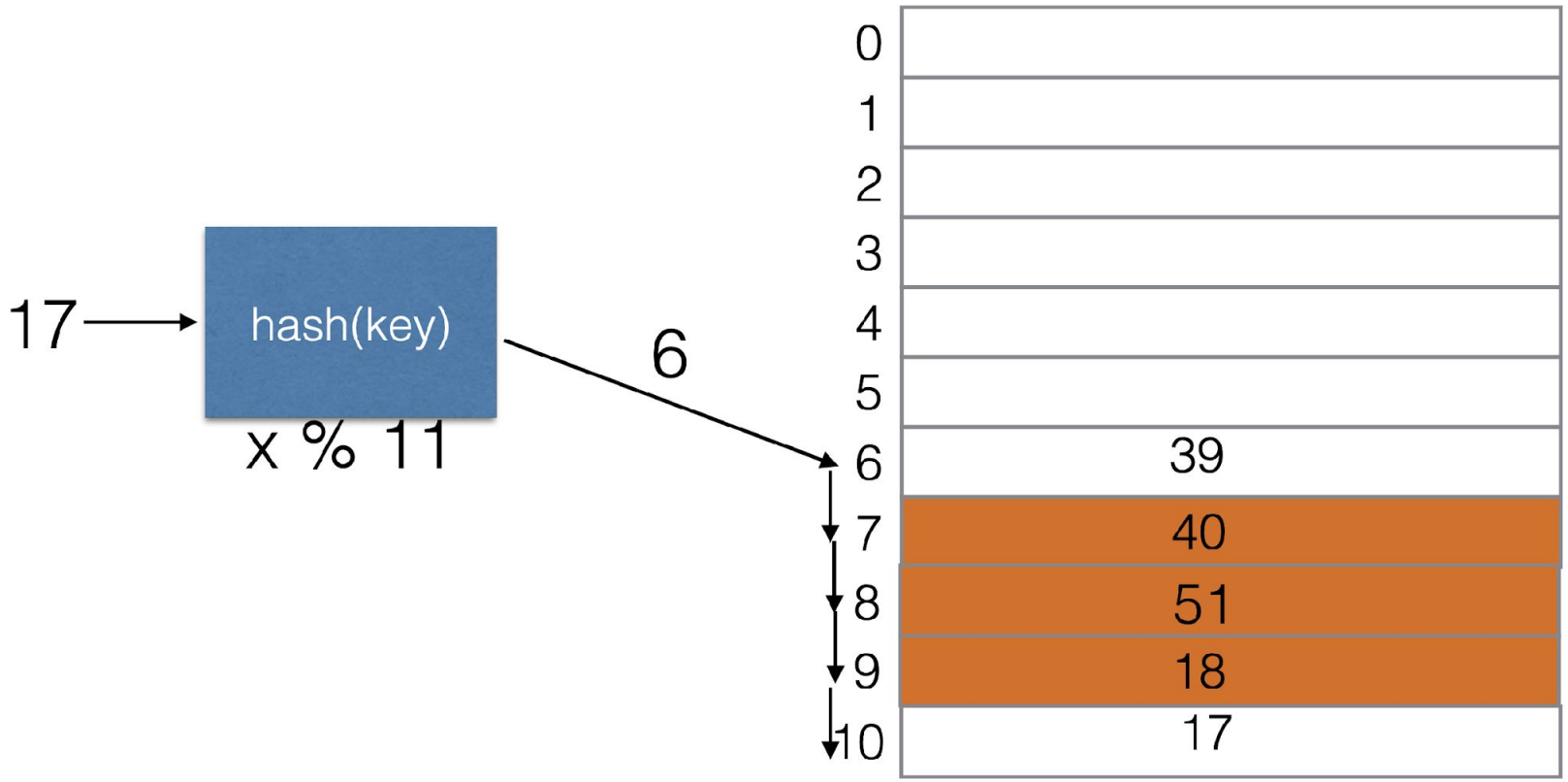
Primary Clustering
Linear Probing Downsides

- Cells 7-8 are occupied with keys that hash to 7. The entire block is unavailable to keys that hash to k<7.



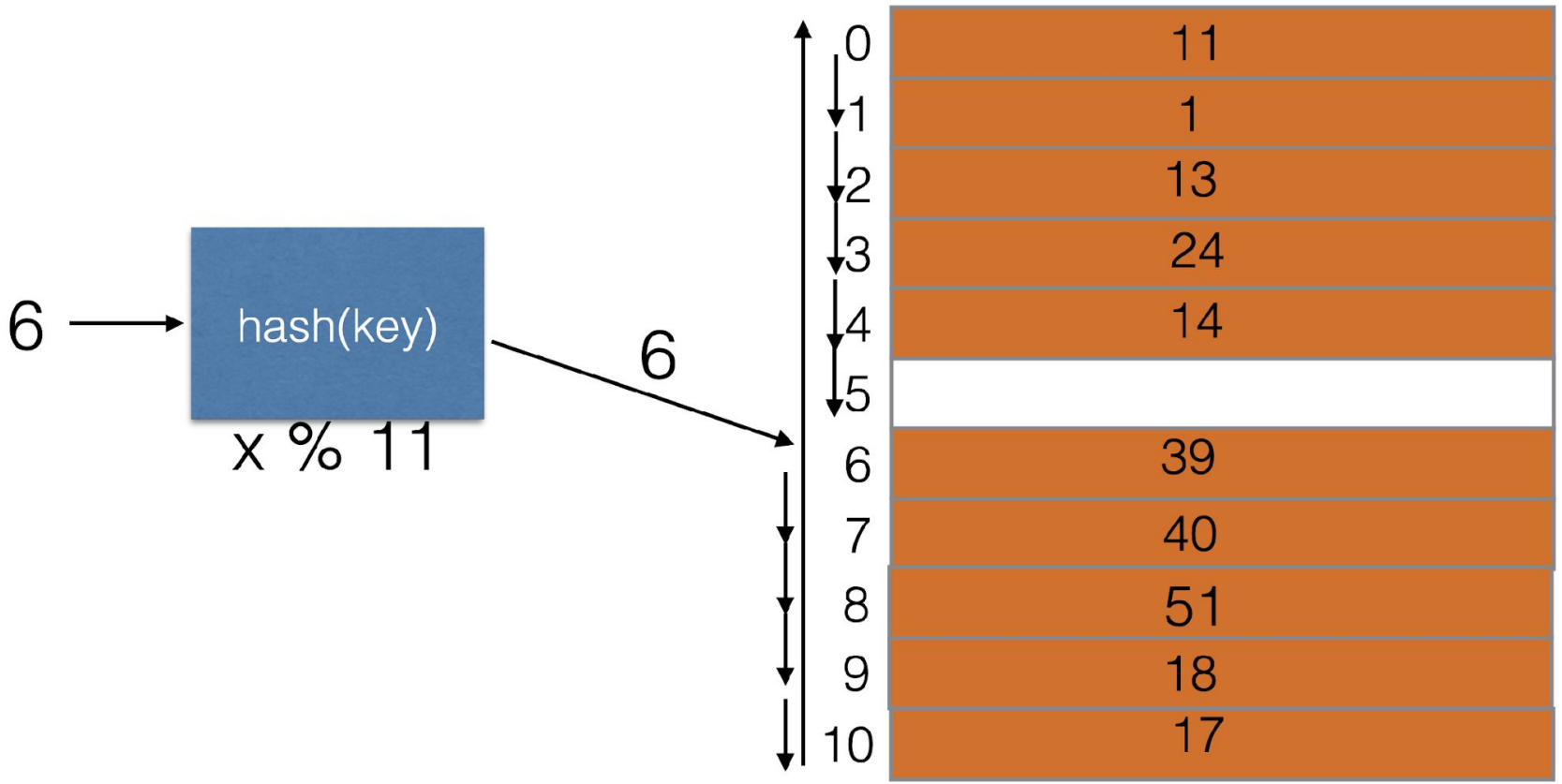
Primary Clustering
Linear Probing Downsides

- Cells 7-8 are occupied with keys that hash to 7. The entire block is unavailable to keys that hash to k<7.



Primary Clustering
Linear Probing Downsides

- This becomes really bad if λ is close to 1



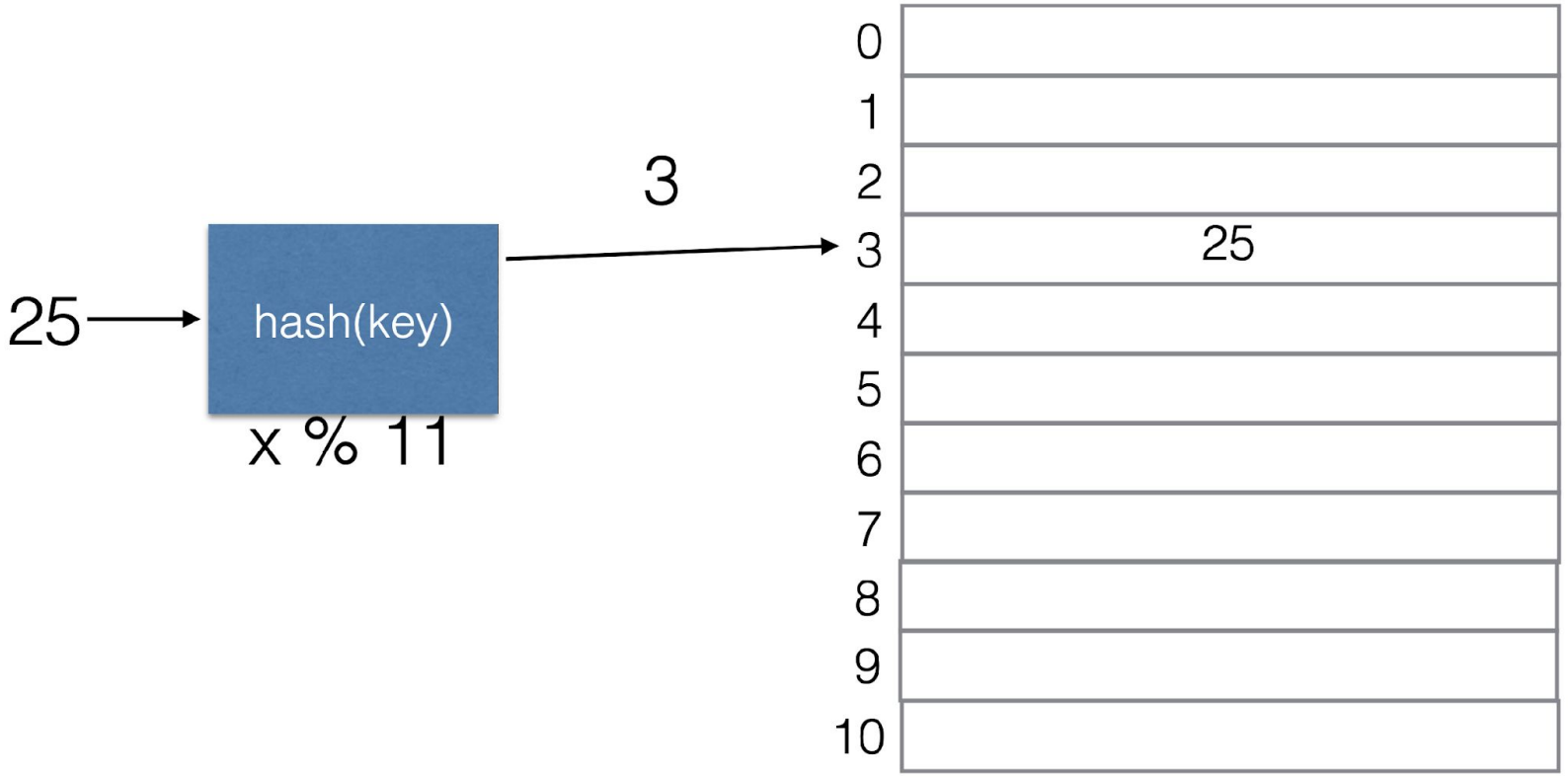
Quadratic Probing

$$(hash(x) + f(i)) \,\%\, TableSize \qquad f(i) = i^2$$

Quadratic Probing

Quadratic Probing

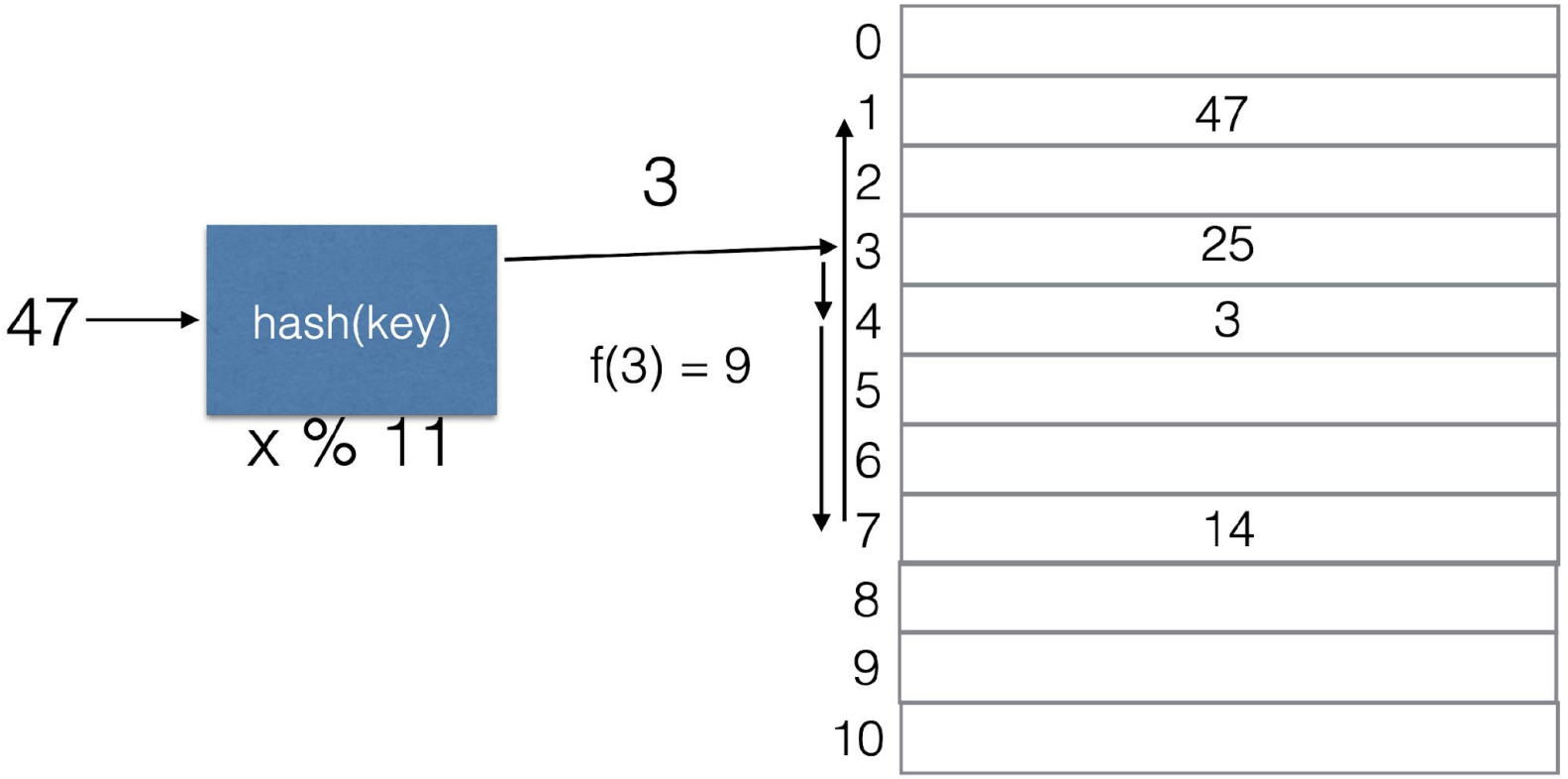
$$(hash(x) + f(i)) \,\%\, TableSize \qquad f(i) = i^2$$



Quadratic Probing

Quadratic Probing

$$(hash(x) + f(i)) \,\%\, TableSize \qquad f(i) = i^2$$

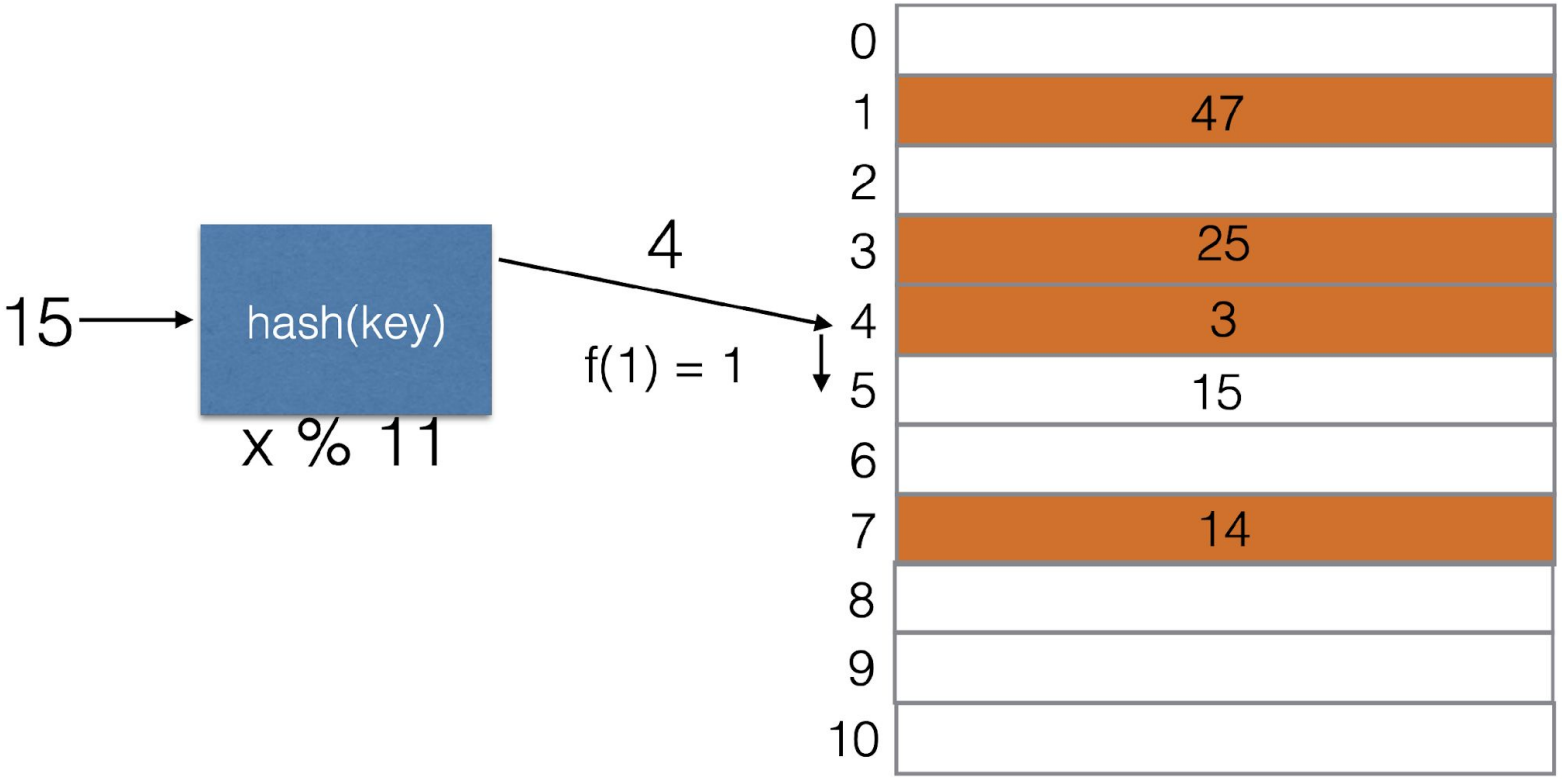


Quadratic Probing: Characteristics

Quadratic Probing

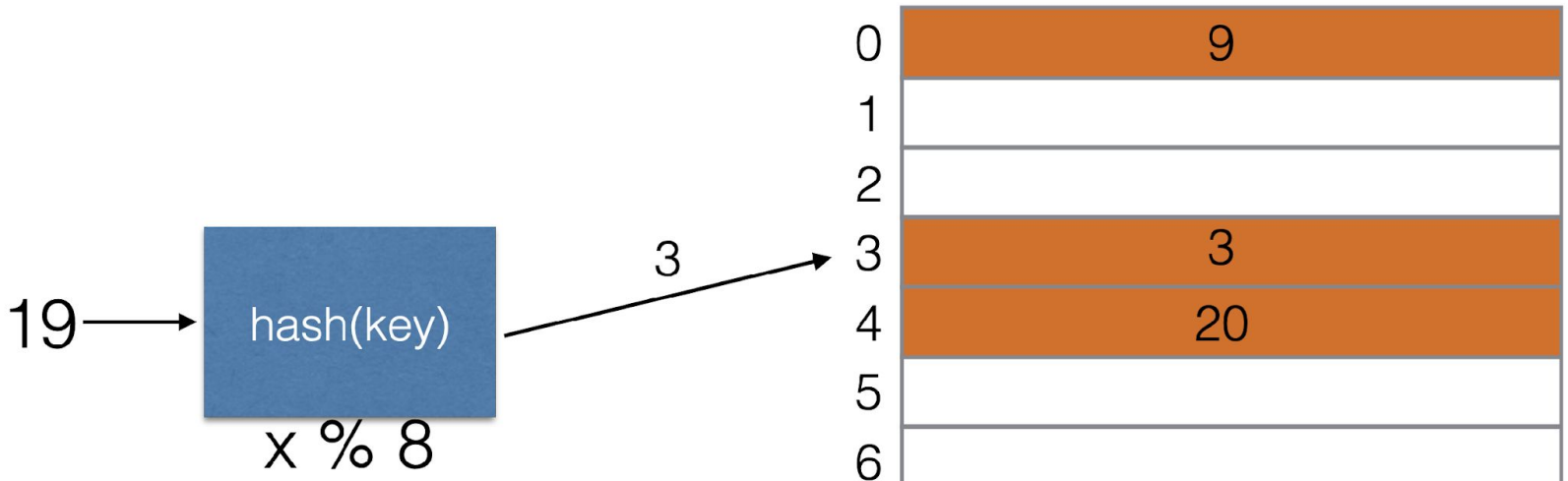
$$(hash(x) + f(i)) \,\%\, TableSize \qquad f(i) = i^2$$

- Primary clustering is not a problem.



Quadratic Probing: Characteristics

Quadratic Probing: Downsides

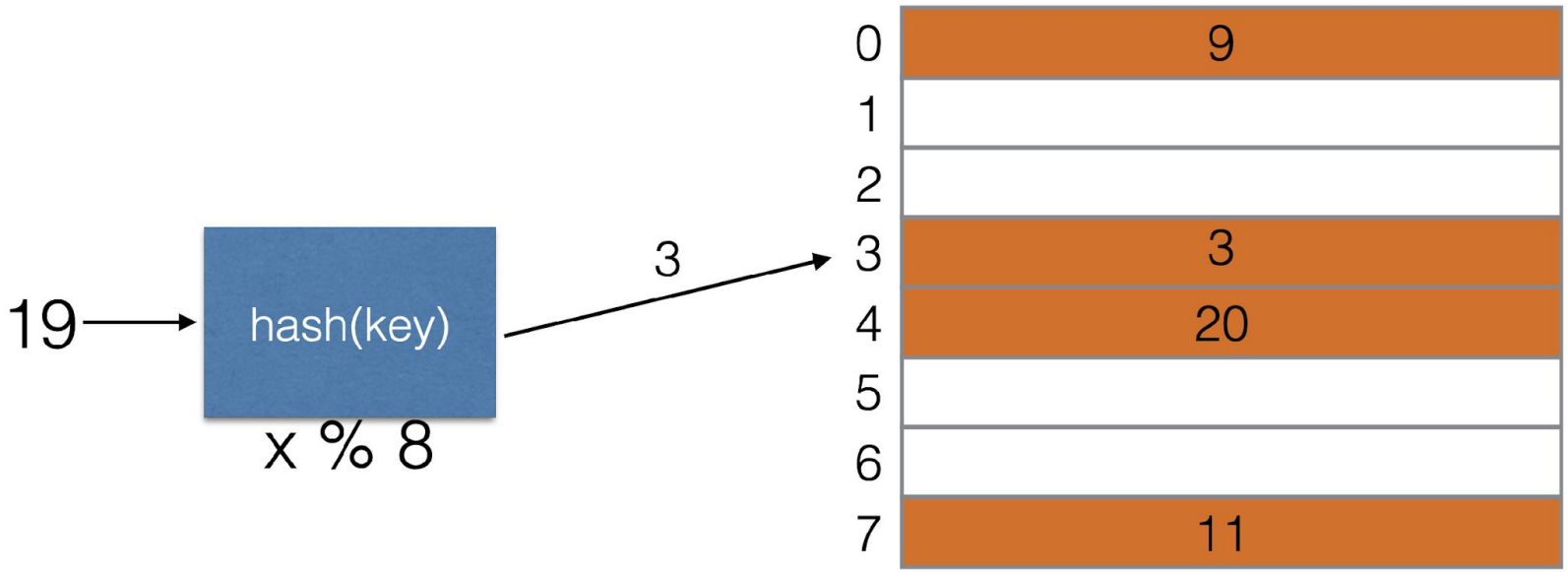


i	1	2	3	4	5	6	7	8	
3 + f(i) % 8	4	7	4	3	4	7	4	3	…

Quadratic Probing: Characteristics

Quadratic Probing: Downsides

- Important: With quadratic probing, *TableSize* should be a prime number! Otherwise it is possible that we won't find an empty cell, even if there is plenty of space.

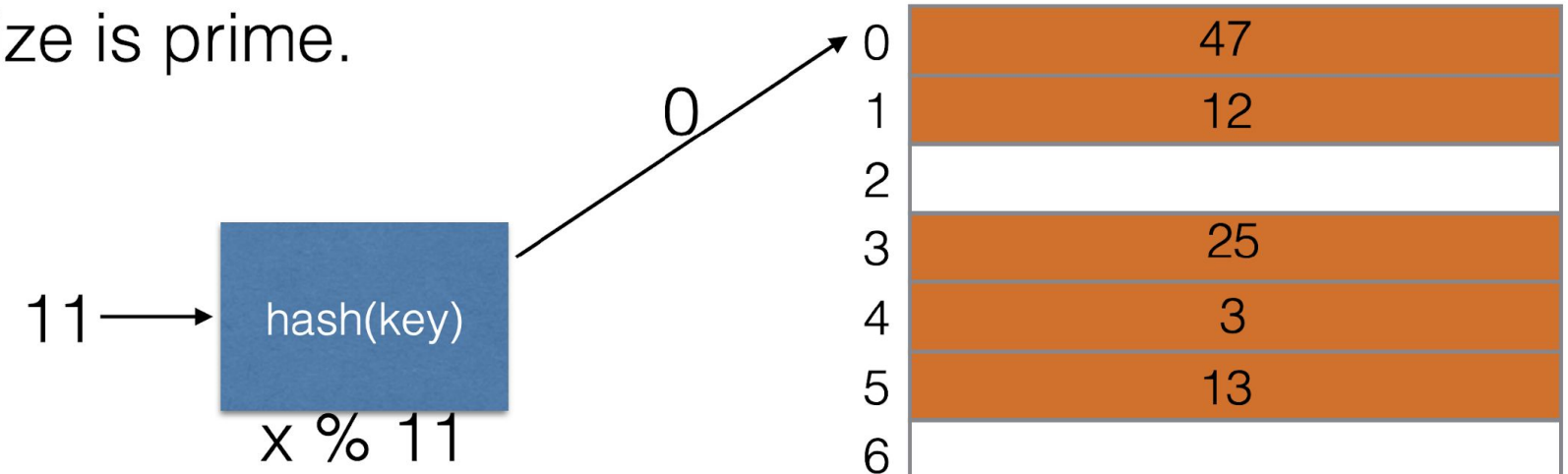


i	1	2	3	4	5	6	7	8	
3 + f(i) % 8	4	7	4	3	4	7	4	3	…

Quadratic Probing: Characteristics

Quadratic Probing

- Problem: If the table gets too full ($\lambda > 0.5$), it is possible that empty cells become unreachable, even if the table size is prime.



i	1	2	3	4	5	6	7	8
0 + f(i) % 11	1	4	9	5	3	3	5	9

…

$$\lambda = 7/11 \approx 0.64$$

Double Hashing

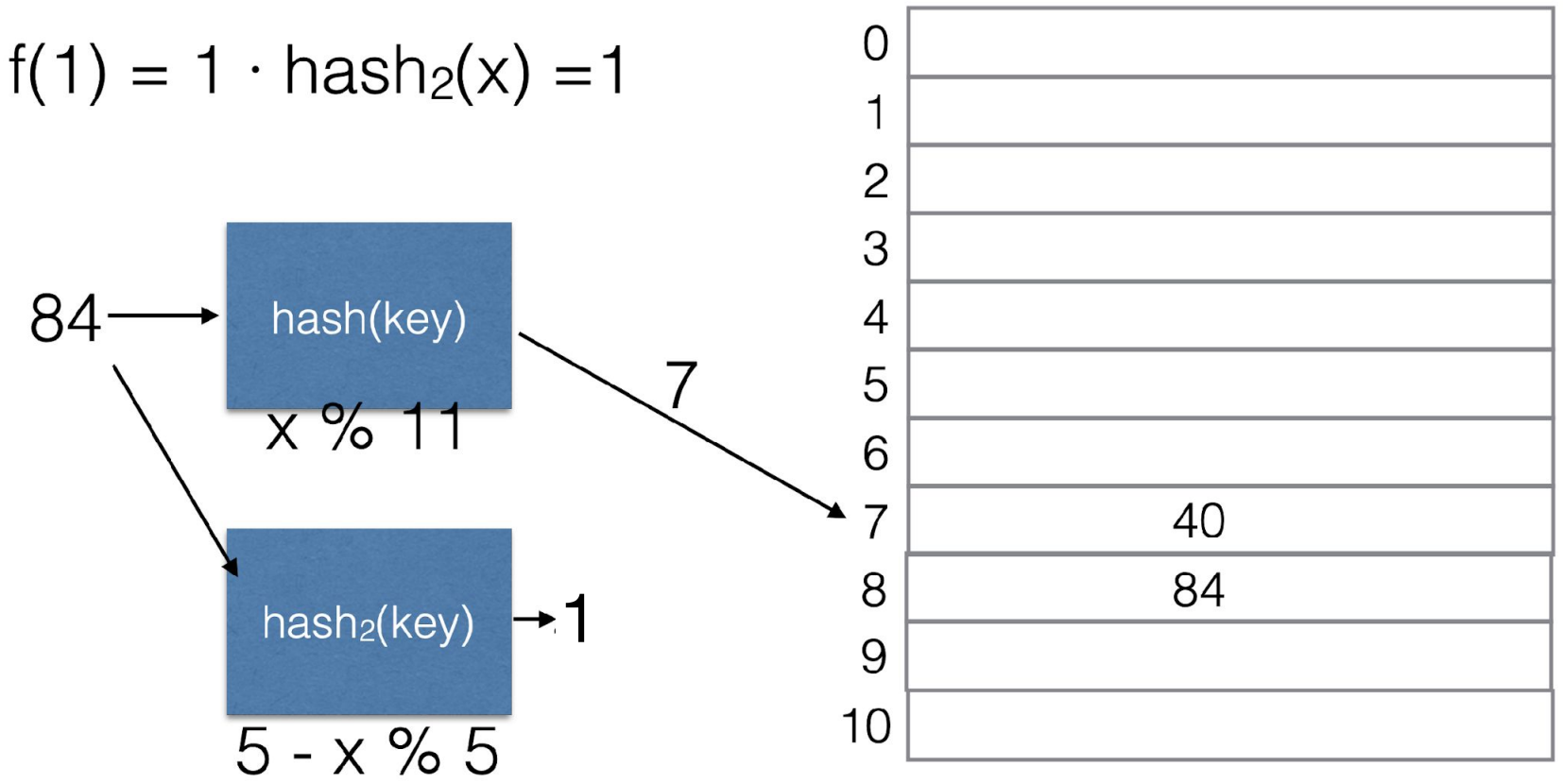
$$f(i) = i \cdot hash_2(x)$$

Compute a second hash function to determine a linear offset for this key.

Double Hashing

$$f(i) = i \cdot hash_2(x)$$

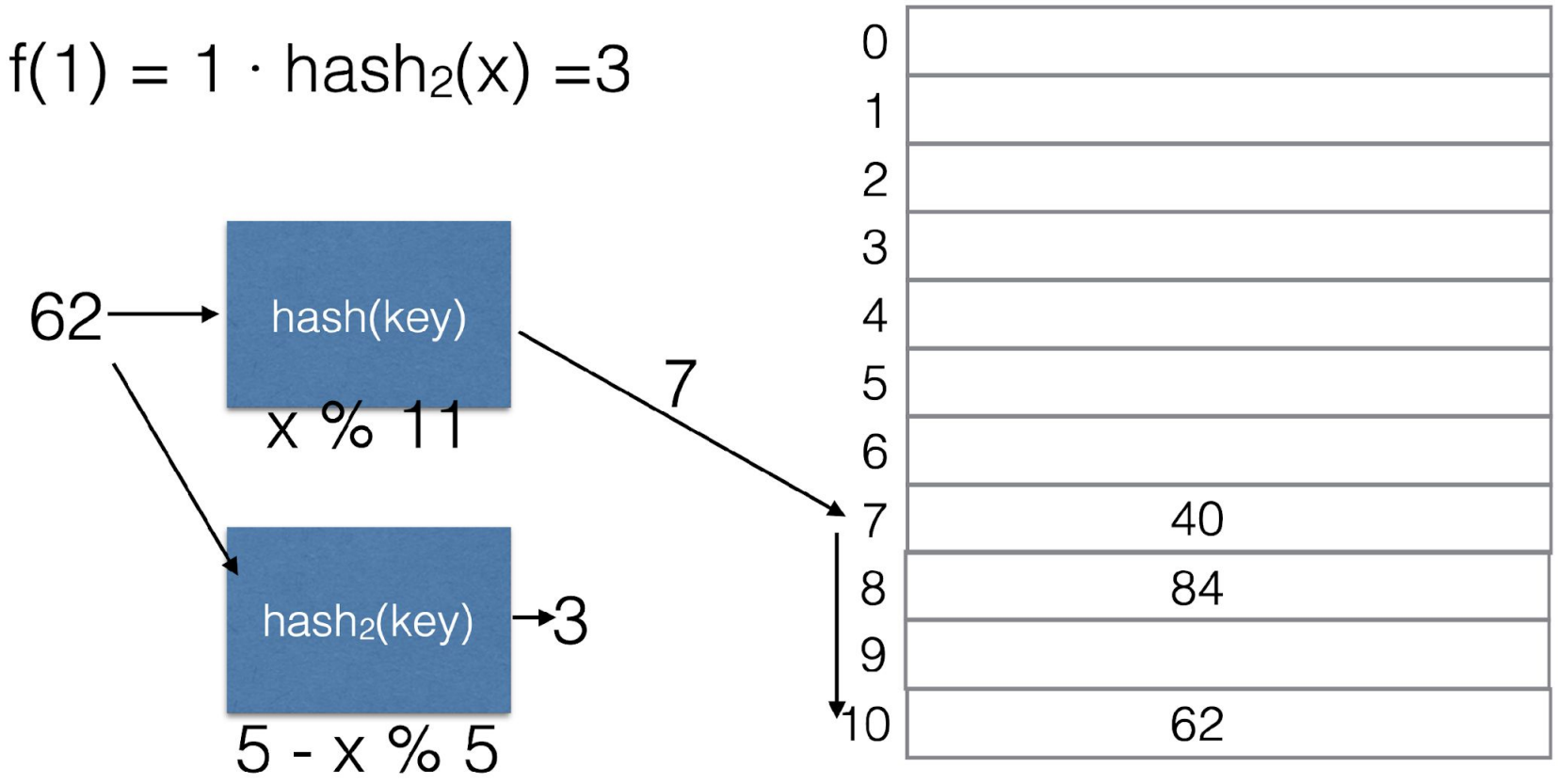
Compute a second hash function to determine a linear offset for this key.



Double Hashing

$$f(i) = i \cdot hash_2(x)$$

Compute a second hash function to determine a linear offset for this key.



Choosing the Secondary Hash

Double Hashing

- Need to choose $hash_2$ wisely!
- What what happen with the following function?

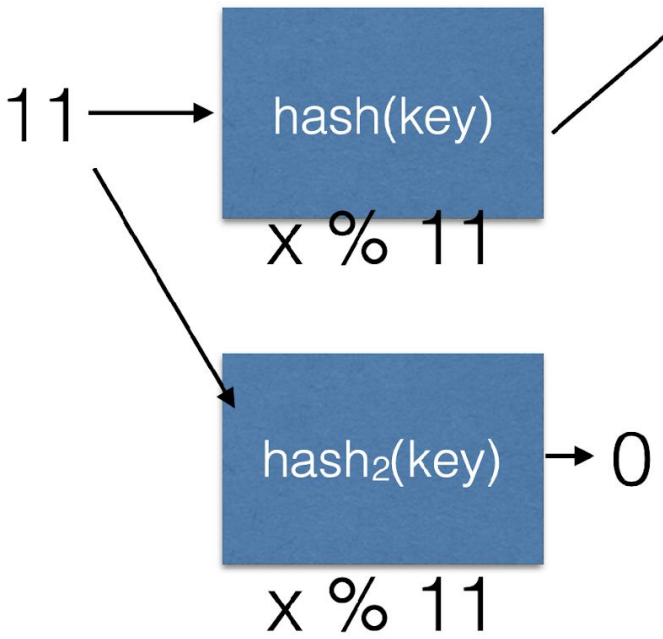
Choosing the Secondary Hash

Double Hashing

- Need to choose $hash_2$ wisely!
- What what happen with the following function?

$f(1) = 1 \cdot hash_2(x) = 0$

$f(2) = 2 \cdot hash_2(x) = 0$



0	22
1	
2	
3	
4	
5	
6	
7	40
8	84
9	29
10	62

Choosing the Secondary Hash

Double Hashing

- A good choice for integers is $hash_2(x) = R - (x \,\%\, R)$

Choosing the Secondary Hash

Double Hashing

- A good choice for integers is $hash_2(x) = R - (x \,\%\, R)$
- As with quadratic hashing, we need to choose the table size to be prime (otherwise cells become unreachable too quickly).
- Properly implemented, double hashing produces a good distribution of keys over table cells.

Summary of Tradeoffs

Summary of Tradeoffs

- Separate Chaining Hash Tables become inefficient if the load factor becomes too large (lists become too long).

Summary of Tradeoffs

- Separate Chaining Hash Tables become inefficient if the load factor becomes too large (lists become too long).

- Hash Tables with Linear Probing become inefficient if the load factor approaches 1 (primary clustering) and eventually fill up.

Summary of Tradeoffs

- Separate Chaining Hash Tables become inefficient if the load factor becomes too large (lists become too long).
- Hash Tables with Linear Probing become inefficient if the load factor approaches 1 (primary clustering) and eventually fill up.
- Hash Tables with Quadratic Probing and Double Hashing can have failed inserts if the table is more than half full.

Summary of Tradeoffs

- Separate Chaining Hash Tables become inefficient if the load factor becomes too large (lists become too long).
- Hash Tables with Linear Probing become inefficient if the load factor approaches 1 (primary clustering) and eventually fill up.
- Hash Tables with Quadratic Probing and Double Hashing can have failed inserts if the table is more than half full.
- Need to copy data to a new table.

Rehashing

- Allocate a new table of twice the size as the original one.

Rehashing

- Allocate a new table of twice the size as the original one.
- For probing hash tables, we cannot simply copy entries to the new array.
  - Different modulo wraparound won't cause the same collisions.

Rehashing

- Allocate a new table of twice the size as the original one.
- For probing hash tables, we cannot simply copy entries to the new array.
  - Different modulo wraparound won't cause the same collisions.
  - Since the hash function is based on the TableSize,keys won't be in the correct cell, anyway.

Rehashing

- Allocate a new table of twice the size as the original one.
- For probing hash tables, we cannot simply copy entries to the new array.
  - Different modulo wraparound won't cause the same collisions.
  - Since the hash function is based on the TableSize,keys won't be in the correct cell, anyway.
- Remove all N items and re-insert into the new table.
  This operation takes O(N), but this cost is only incurred in the rare case when rehashing is needed.

STL Maps

- **Maps** are STL containers are defined as template with two parameters:

```
map<string, int> m;
```

- Insertion of new key-value pairs:

```
m["test"] = 1; // option 1
m.insert({"test", 1}); // option 2, equivalent
```

- Looking up items:

```
int r = m["test"]; //option 1, creates an empty element if no "test"
int r = m.at("test"); //option 2, throws an exception if no "test"
```

- Deletion is performed via an iterator:

```
m.erase(m.find("test"));
```

- Iterating through the items:

```
for(map<string,int>::iterator it = m.begin(); it != m.end(); ++it)
{ //some code using it->first() and it->second() }
```

STL Unordered Maps

- Standard implementation of `map`: *red-black trees* with O(log) insert/lookup/delete
  - *Reason:* `map` is specified as an **ordered** container, storing keys in order
  - A hash map would not work for ordering
- If you want a hash table implementation, use `unordered_map`
  - Similar operations to the map, plus **separate chaining** for collisions

```
unordered_map<string, int, myhash> m; //3rd argument optional
```

- Class `myhash` needs to overload `operator(string s)`
  - By default, std::hash<type> implements hashing for all standard types
- Useful functions:
  - `rehash(int bucketCount)` by setting the bucket count & rehashes
  - `reserve(int elemCount)` sets aside the space for `elemCount` elements without exceeding the load factor
  - `load_factor()` returns the current load factor (average elements per bucket)