Data Structures in Java

Lecture 12: Introduction to Hashing.

10/19/2015

Daniel Bauer

Homework

- Due Friday, 11:59pm.
- Jarvis is now grading HW3.

Recitation Sessions

- Recitations this week:
  - Review of balanced search trees.
  - Implementing AVL rotations.
  - Implementing maps with BSTs.
  - Hashing (Friday/Next Mon & Tue).

Midterm

- **Midterm next Wednesday (in-class)**
  - Closed books/notes/electronic devices.
  - Ideally, bring a pen, water, and nothing else.
  - 60 minutes
  - Midterm review this Wednesday in class.

How to Prepare?

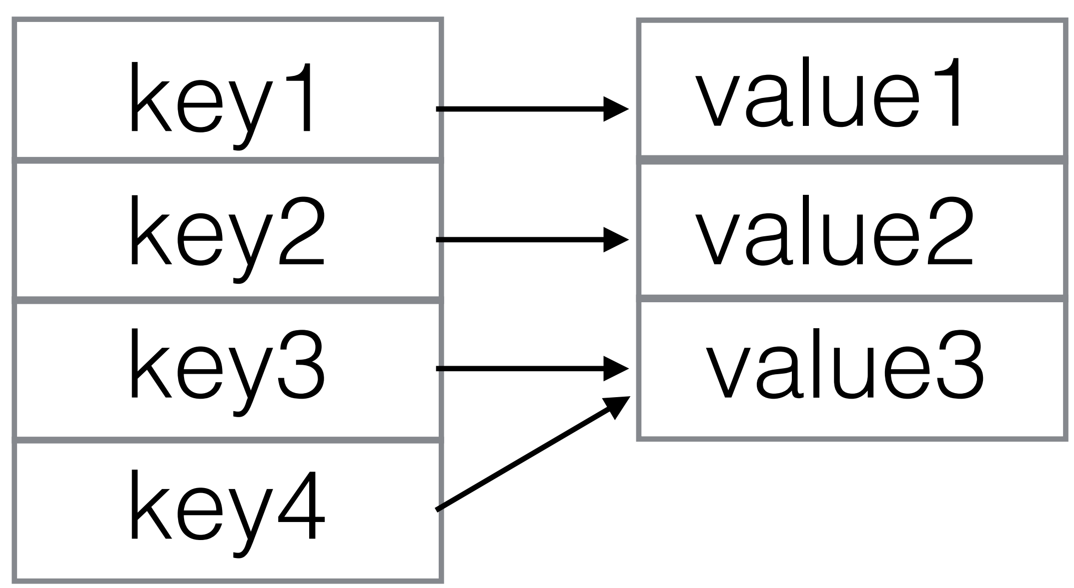
- Midterm will cover all content up to (and including) this week.
  - Know all ADTs, operations defined on them, data structures, running times.
  - Know basics of running time analysis (big-O).
  - Understand recursion, inductive proofs, tree traversals, …
- Practice questions out today. Discussed Wednesday.
- Good idea to review slides & homework!

How to Prepare Even More?

- Optional:
  - Solve Weiss textbook exercises and discuss on Piazza.
  - Try to implement data structures from scratch.

Map ADT

- 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)



Implementing Maps

Implementing Maps

- Option 1: Use any set implementation to store special (key,value) objects.
    - Comparing these objects means comparing the key (testing for equality or implementing the `Comparable` interface)

Implementing Maps

- Option 1: Use any set implementation to store special (key,value) objects.
  - Comparing these objects means comparing the key (testing for equality or implementing the `Comparable` interface)
- Option 2: Specialized implementations
  - B+ Tree: nodes contain keys, leaves contain values.
  - Plain old Array: Only integer keys permitted.
  - Hash maps (this week)

Balanced BSTs

- Runtime of BST operations (`insert, contains/find, remove, findmin, findmax`) depend on height of the tree.

- Balance condition: Guarantee that the BST is always close to a complete binary tree.

  - Then the height of the tree will be O(*log N)*.

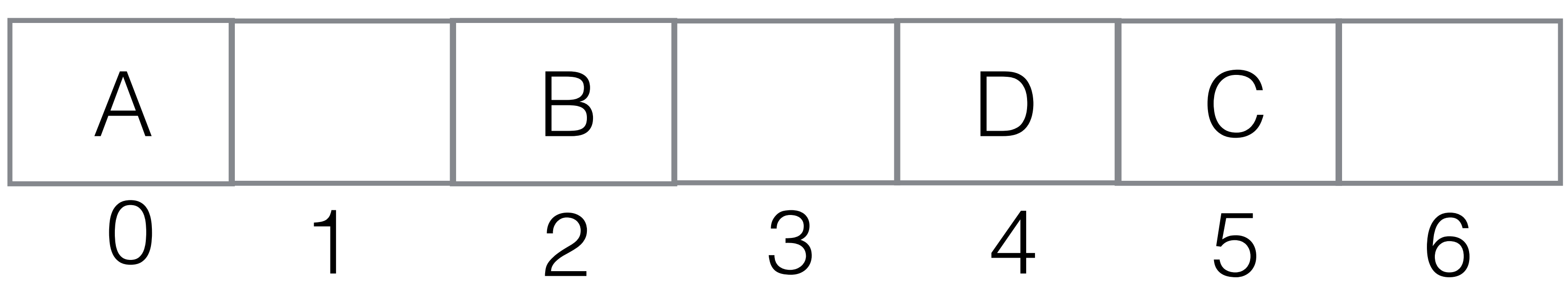
  - All BST operations will run in *O(log N)*.

  - Map operations `get` and `put` will also run in O(log N)

Can we do better?

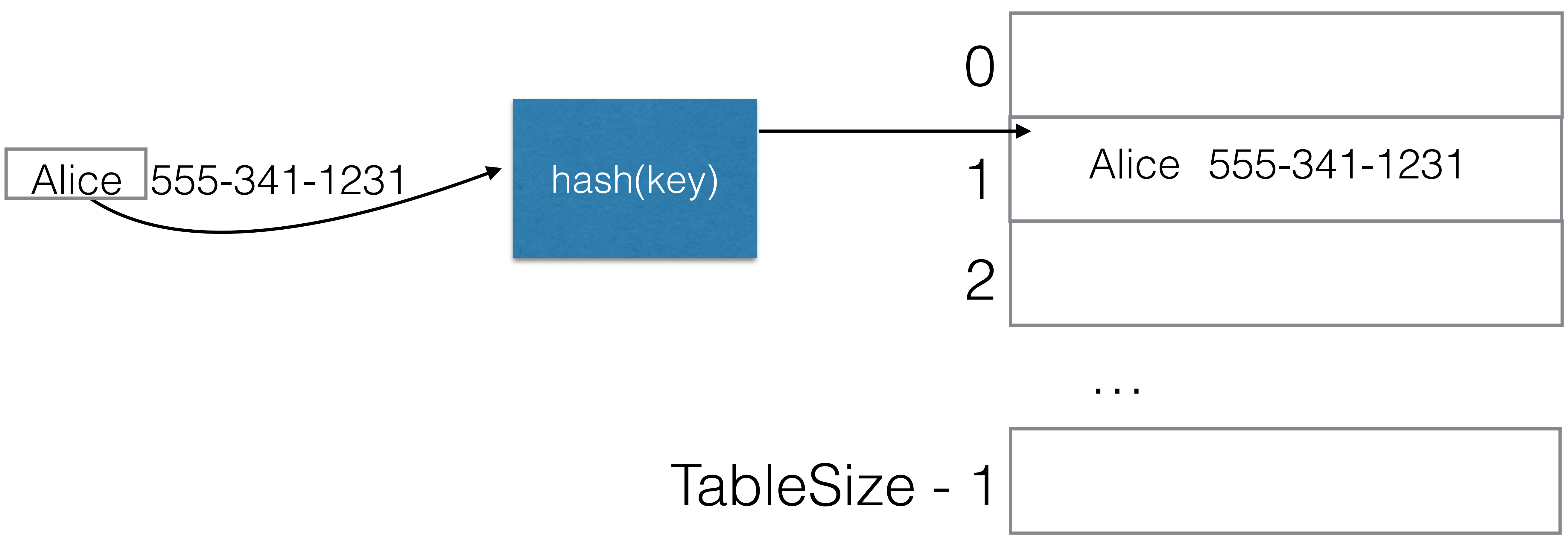
Arrays as Maps

- When keys are integers, arrays provide a convenient way of implementing maps.
- Time for `get` and `put` is O(1).



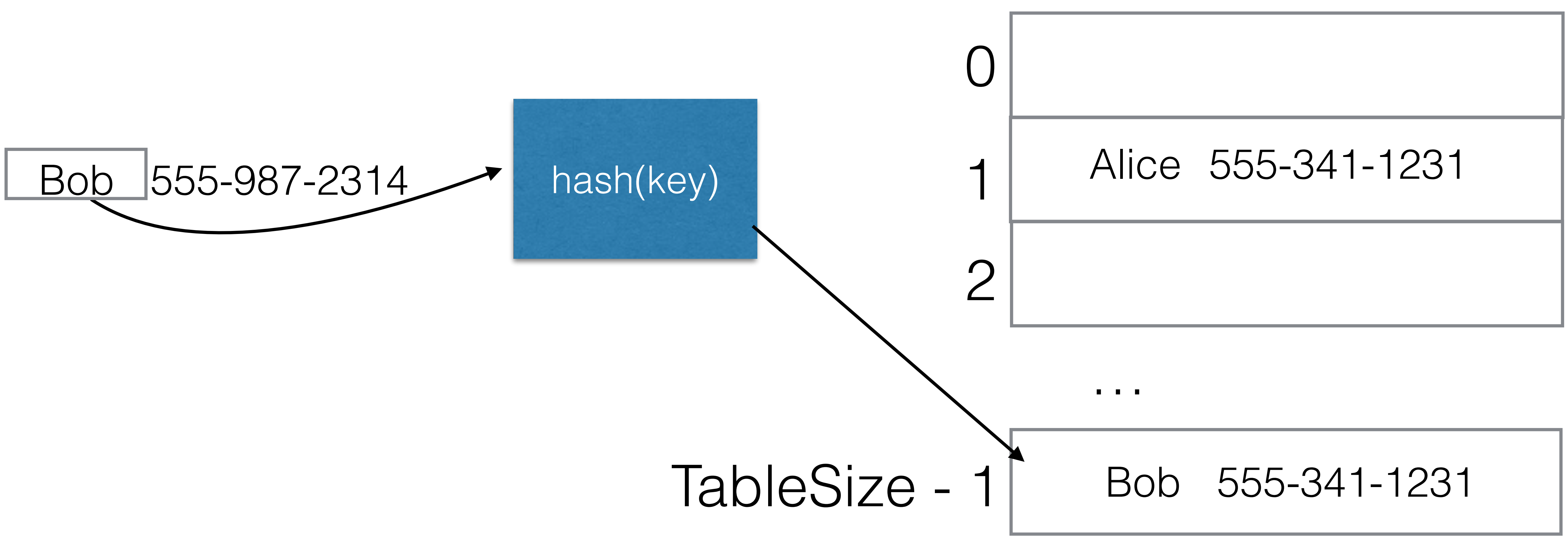
Hash Tables

- 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*



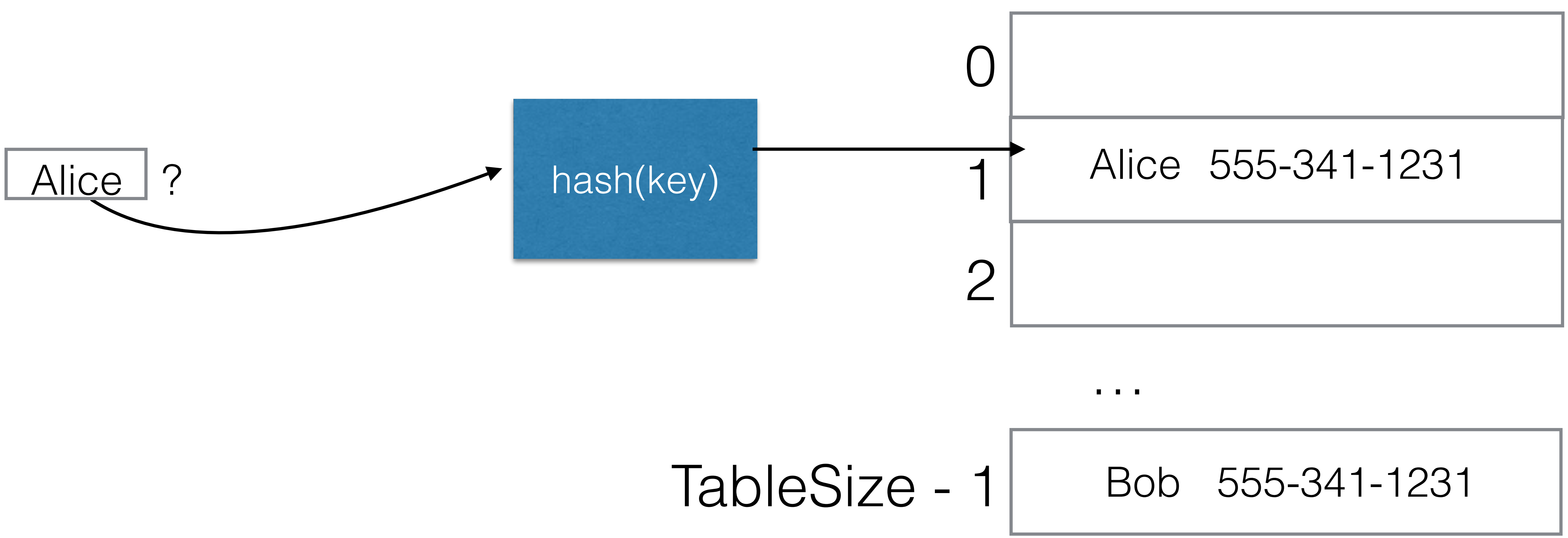
Hash Tables

- 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*



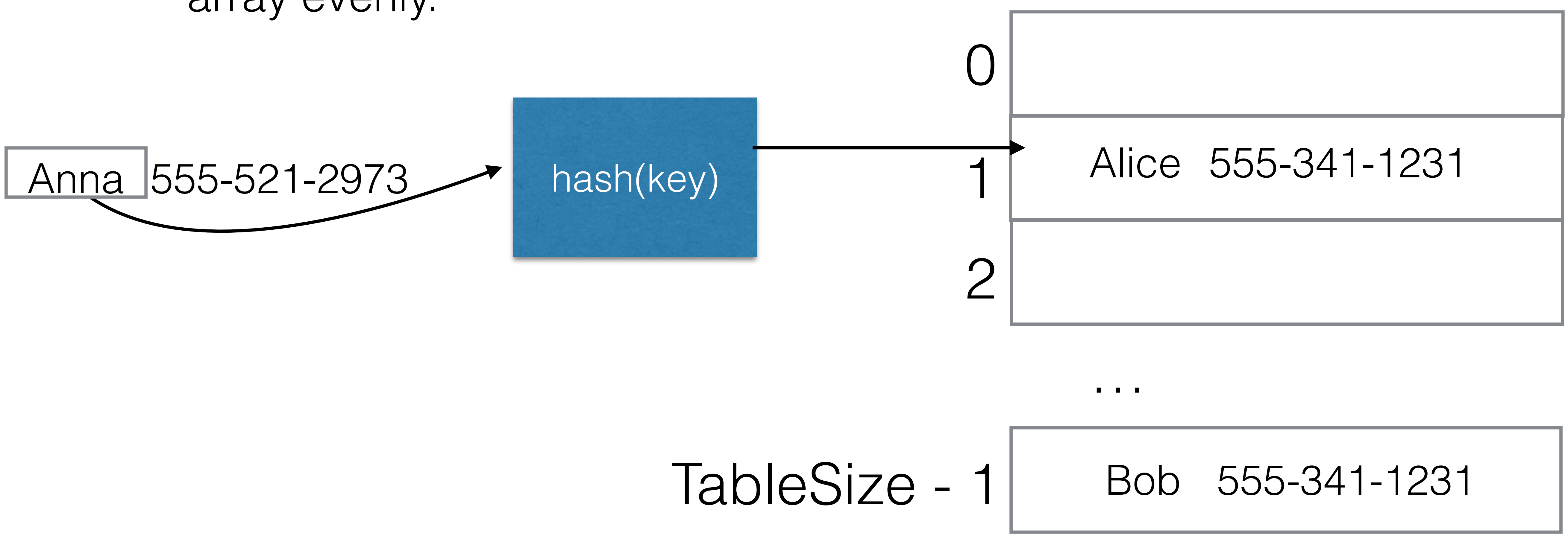
Hash Tables

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



Hash Table 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.



Choosing a Hash Function

- Hash functions depends on: type of keys we expect (Strings, Integers…) and *TableSize*.
- Hash functions needs to:
  - Spread out the keys as much as possible in the table (ideal: uniform distribution).
  - Make sure that all table cells can be reached.

Choosing a Hash Function: Integers

- If the keys are integers, it is often okay to assume that the possible keys are *distributed evenly.*

hash(x) = x % TableSize

```
public static int hash( Integer key, int tableSize ) {
    return key % tableSize;
}
```

e.g. TableSize = 5
hash(0) = 0, hash(1) = 1,
hash(5) = 0, hash(6) = 1

Choosing a Hash Function: Strings - Idea 1

- Idea 1: Sum up the ASCII (or Unicode) values of all characters in the String.

```
public static int hash( String key, int tableSize ) {
        int hashVal = 0;

        for( int i = 0; i < key.length( ); i++ )
            hashVal = hashVal + key.charAt( i );

        return hashVal % tableSize;
}
```

e.g. “Anna” → $65 + 2 \cdot 110 + 97 = 382$

A → 65, n → 110, a → 97

Choosing a Hash Function: Strings - Problems with Idea 1

- Idea 1 doesn't work for large table sizes:
  - Assume *TableSize = 10,007*
  - Every character has a value in the range 0 and 127.
  - Assume keys are at most 8 chars long:
    - hash(key) is in the range 0 and $127 \cdot 8 = 1016$.
    - Only the first 1017 cells of the array will be used!

Choosing a Hash Function: Strings - Problems with Idea 1

- Idea 1 doesn’t work for large table sizes:
  - Assume *TableSize = 10,007*
  - Every character has a value in the range 0 and 127.
  - Assume keys are at most 8 chars long:
    - hash(key) is in the range 0 and 127 · 8 = 1016.
    - Only the first 1017 cells of the array will be used!
- Also: All anagrams will produce collisions: “rescued”, “secured”,”seducer”

Choosing a Hash Function: Strings - Idea 2

- Idea 2: Spread out the value for each character

```
public static int hash( Integer key, int tableSize ) {
    return (key.charAt(0) +
            27 * key.charAt(1) +
       27 * 27 * key.charAt(2));
}
```

Choosing a Hash Function: Strings - Idea 2

- Idea 2: Spread out the value for each character

```
public static int hash( Integer key, int tableSize ) {
    return (key.charAt(0) +
            27 * key.charAt(1) +
       27 * 27 * key.charAt(2));
}
```

- Problem: assumes that the all three letter combinations (*trigrams*) are equally likely at the beginning of a string.
  - This is not the case for natural language
    - some letters are more frequent than others
    - some trigrams (e.g. “xvz”) don’t occur at all.

Choosing a Hash Function: Strings - Idea 3

```
public static int hash( String key, int tableSize ) {
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}
```

$$key[N-1] \cdot 37^N + key[N-2] \cdot 37^{N-1} + \cdots + key[1] \cdot 37 + key[0]$$

This is what Java Strings use; works well, but slow for large strings.

Combining Hash Functions

- In practice, we often write hash functions for some container class:
  - Assume all member variables have a hash function (Integers, Strings…).
  - Multiply the hash of each member variable with some distinct, large prime number.
  - Then sum them all up.

Combining Hash Functions, Example

```
public class Person {
    public String firstName;
    public String lastName;
    public Integer age;
}
```

Combining Hash Functions, Example

```java
public class Person {
    public String firstName;
    public String lastName;
    public Integer age;
}
```

```java
public static int hash( Person key, int tableSize ) {
    int hashVal =  hash(key.firstName, tableSize) * 127 +
                   hash(key.lastName, tableSize) * 1901 +
                   hash(key.age, tableSize) * 4591;
    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;
}
```

Why Prime Numbers?

- To reduce collisions, *TableSize* should not be a factor of any large hash value (before taking the modulo).

Bad example:

TableSize = 8   factors = 2, 4, 6, 8, 16

Why Prime Numbers?

- To reduce collisions, *TableSize* should not be a factor of any large hash value (before taking the modulo).

  Bad example:

  TableSize = 8   factors = 2, 4, 6, 8, 16

- Good practices:
  - Keep *TableSize* a prime number.
  - When combining hash values, make the factors prime numbers.

What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.
- Need to make sure that objects that are used as keys cannot be changed at runtime (they are **immutable**)

What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.

- Need to make sure that objects that are used as keys cannot be changed at runtime (they are **immutable**)

    - Otherwise, if their content changes their hash value should change too!

What Objects Can be Keys?

- Anything can be a key, we just need to find a good hash function.

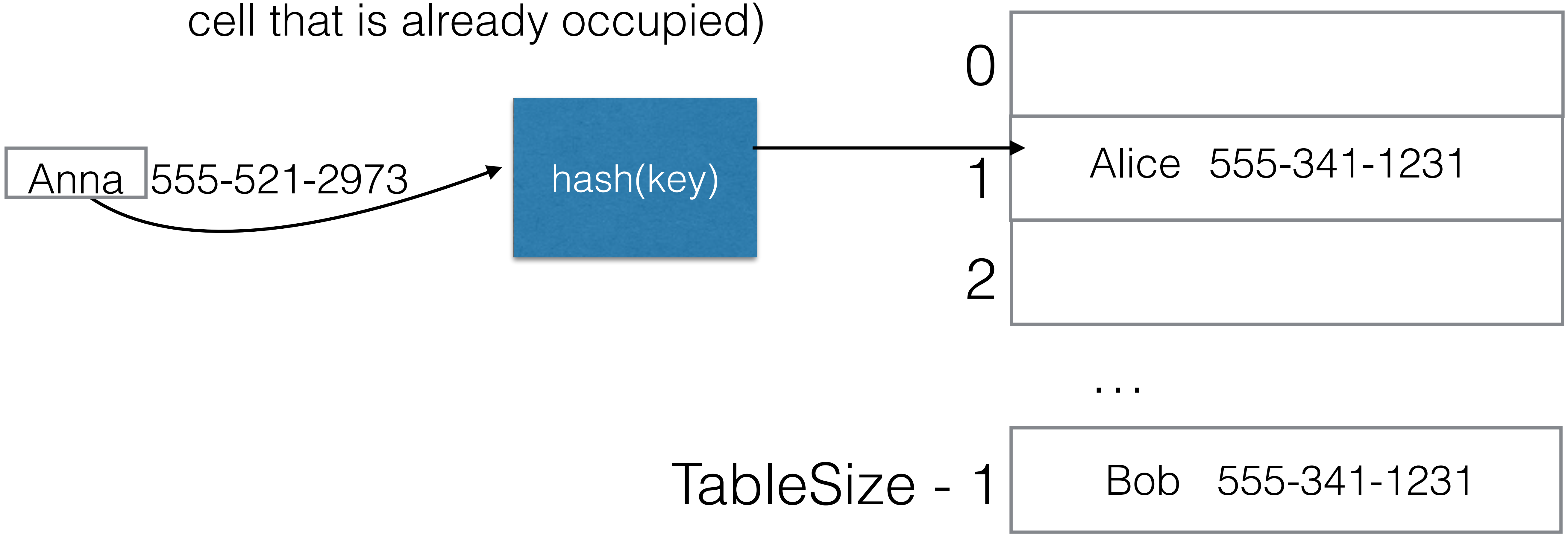
- Need to make sure that objects that are used as keys cannot be changed at runtime (they are **immutable**)

    - Otherwise, if their content changes their hash value should change too!

- How would you compute the hash value for a LinkedList or a Binary Tree?

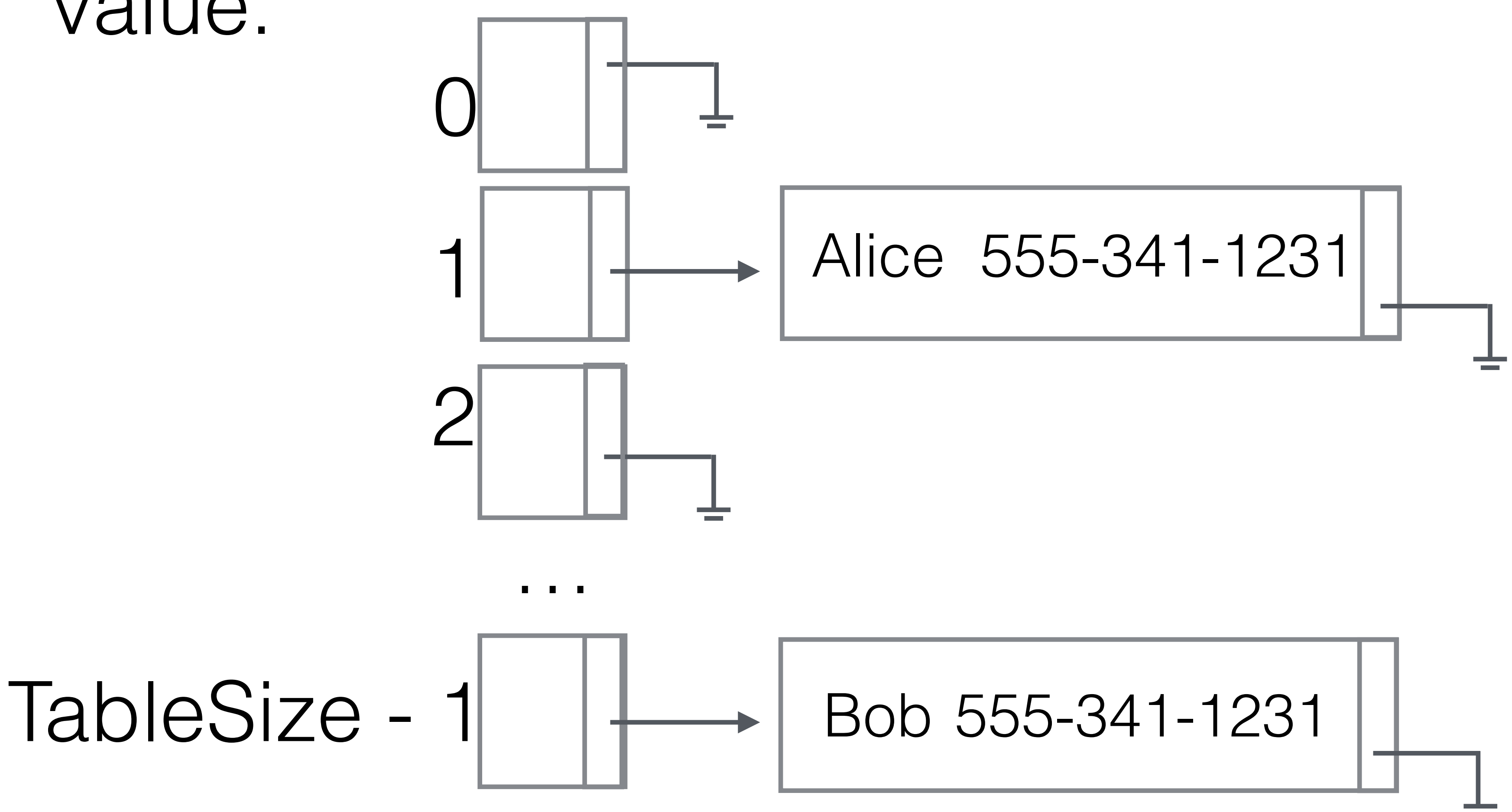
Hash Table Collisions

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



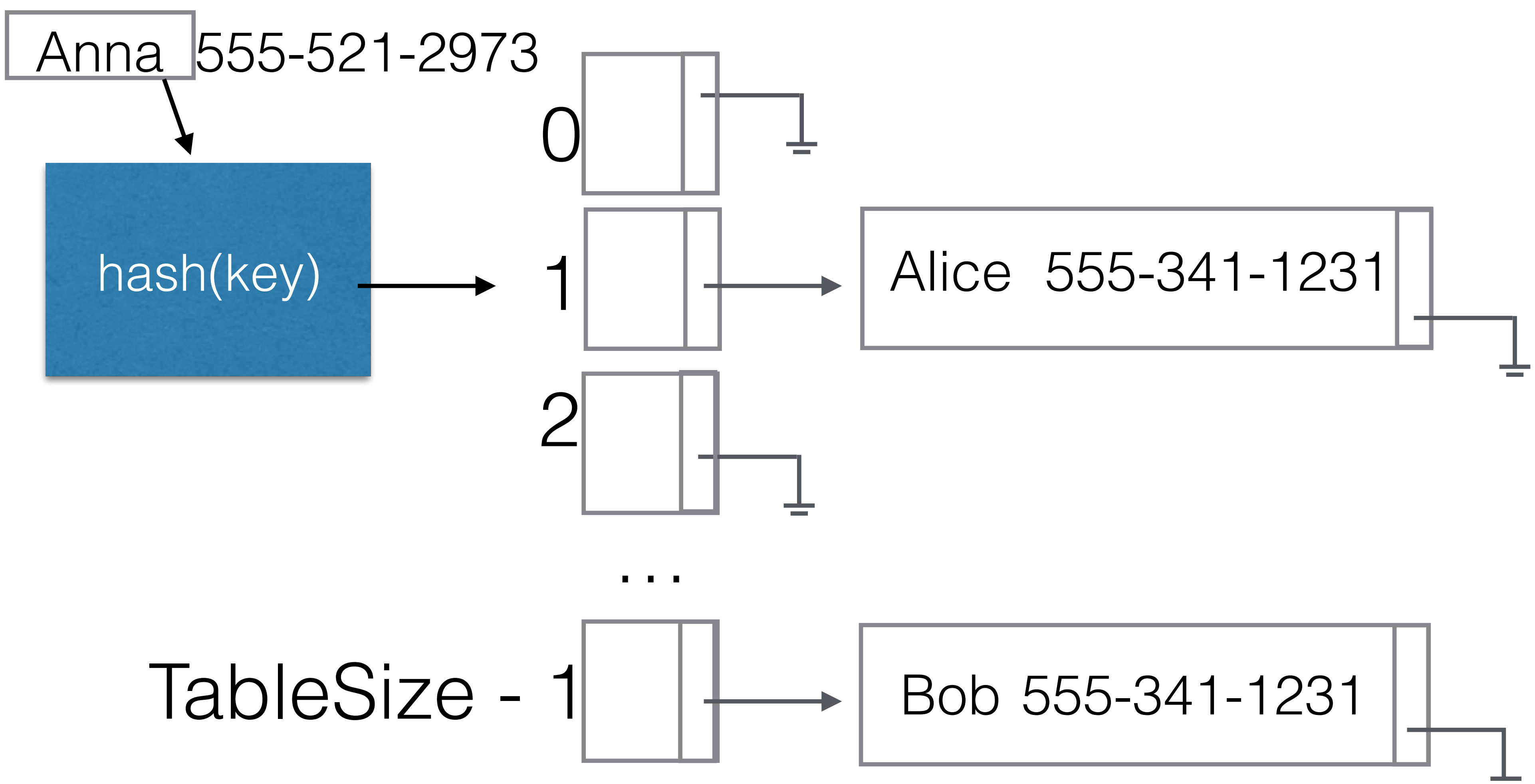
Dealing with Collisions: Separate Chaining

- 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.



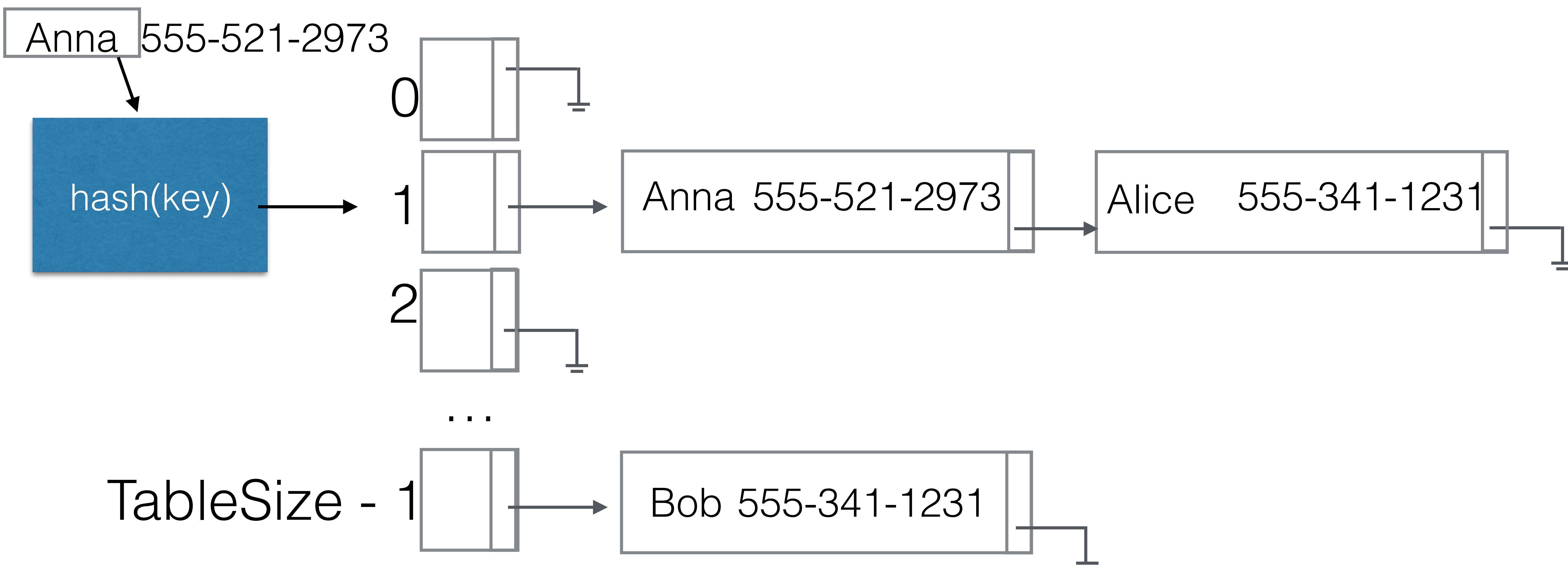
Dealing with Collisions: Separate Chaining

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



Dealing with Collisions: Separate Chaining

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



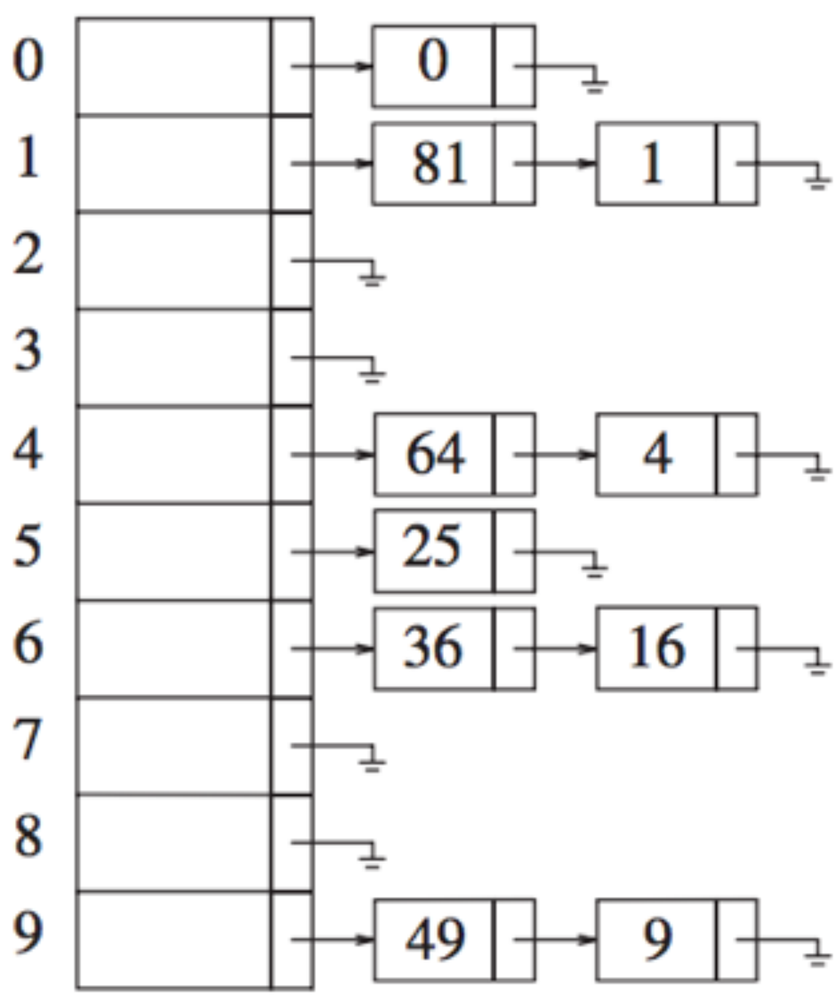
Analyzing Running Time for Separate Chaining (1)

- 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



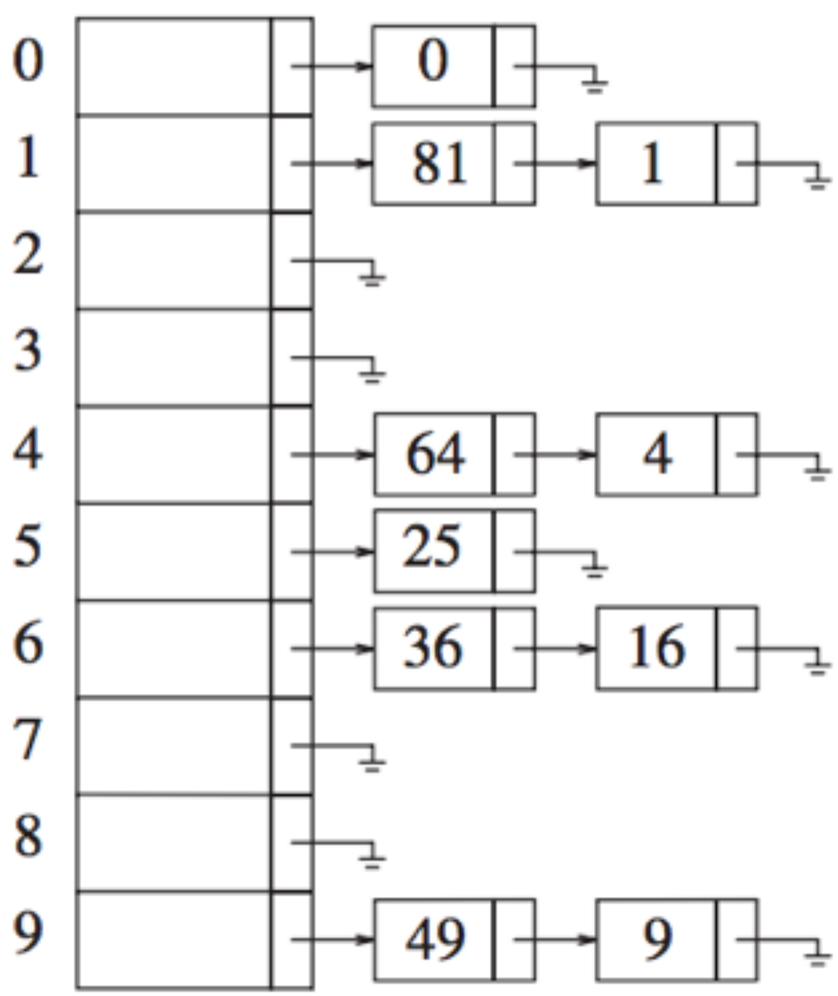
- Let *N* be the number of keys in the table.
- Define the load factor as

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

- The average length of a list is λ .

Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed.

Analyzing Running Time for Separate Chaining (2)



- 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).

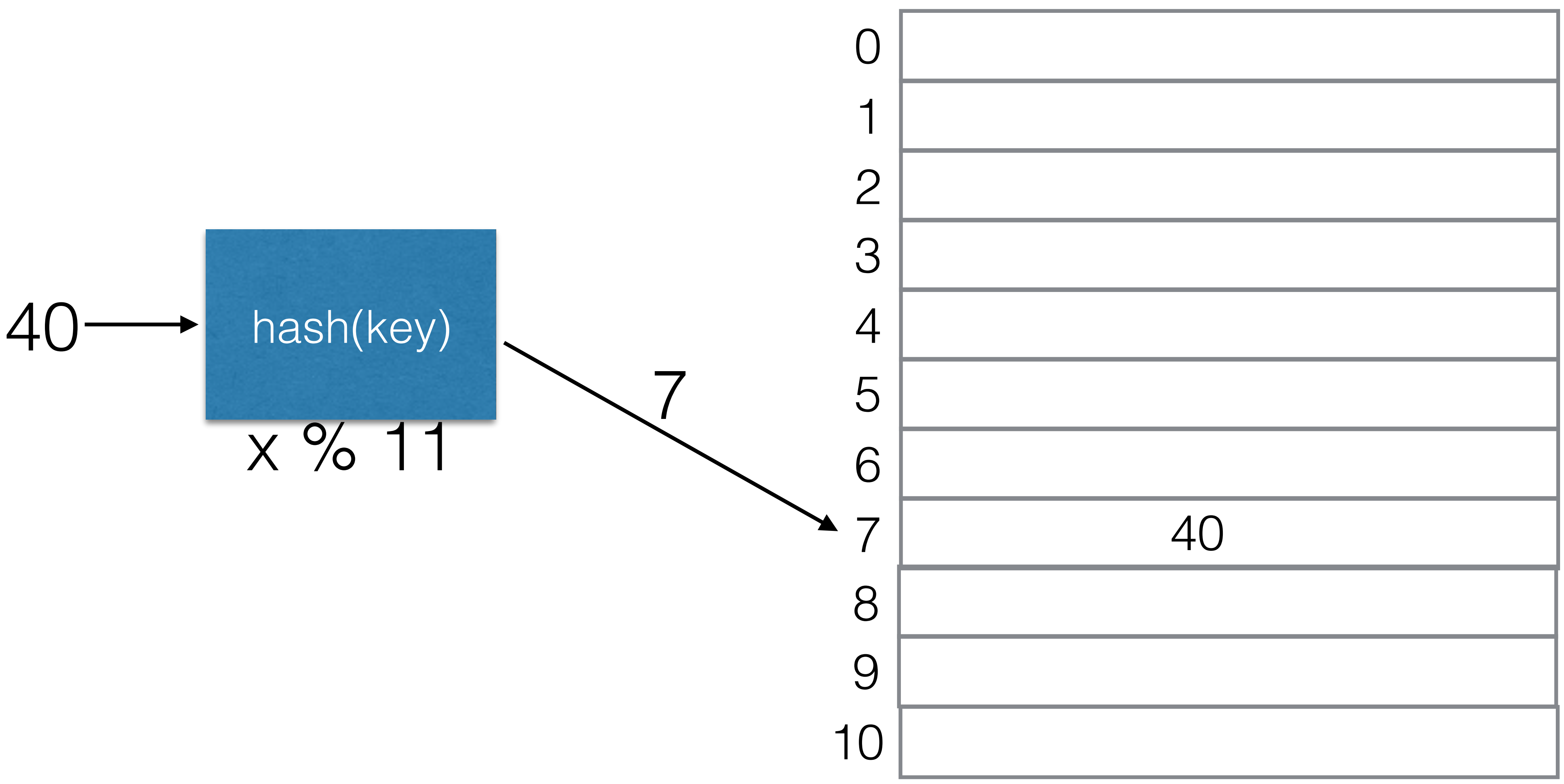
Problems with Separate Chaining

- 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.

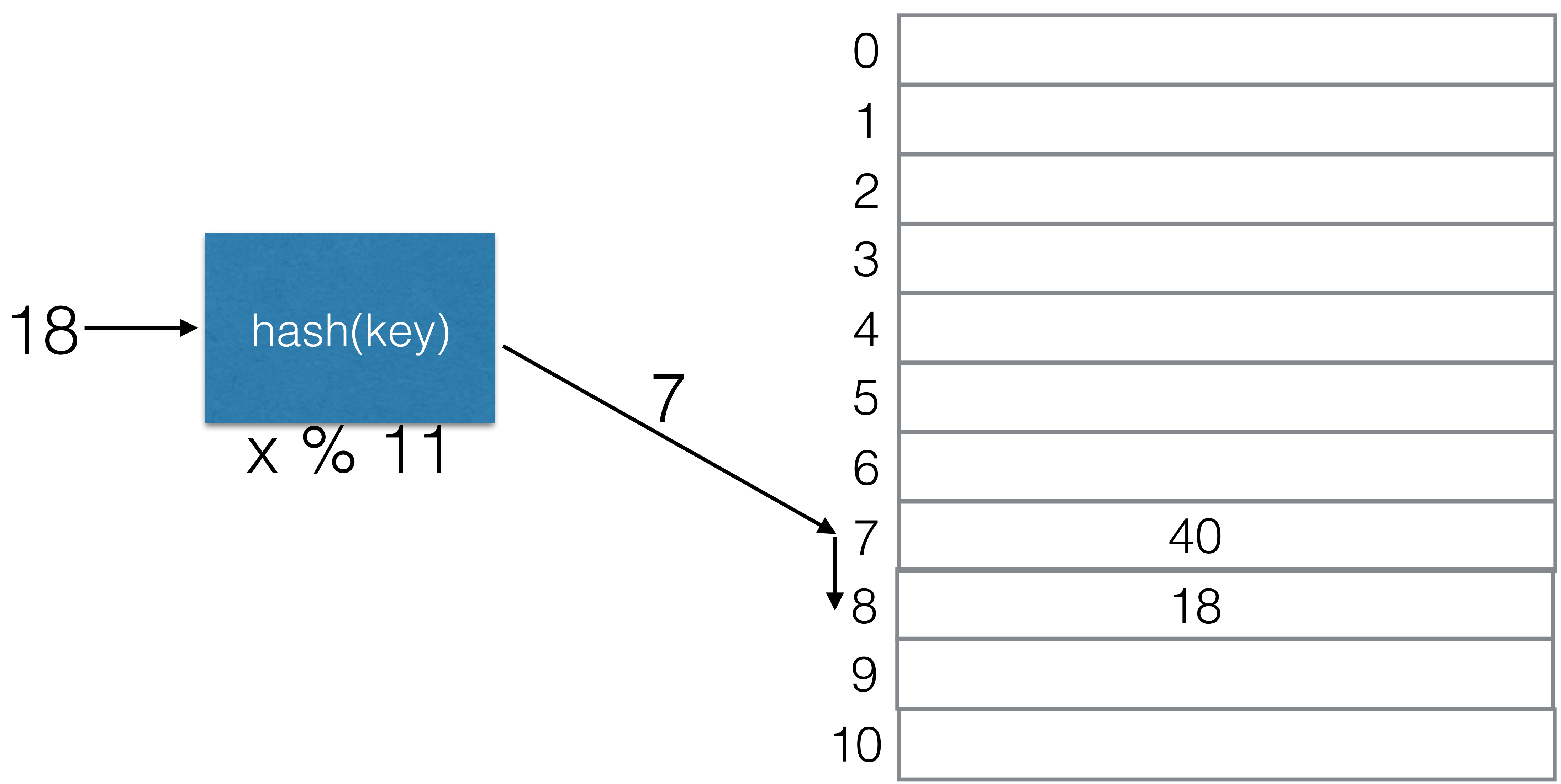
Hash Tables without Linked Lists: Probing

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



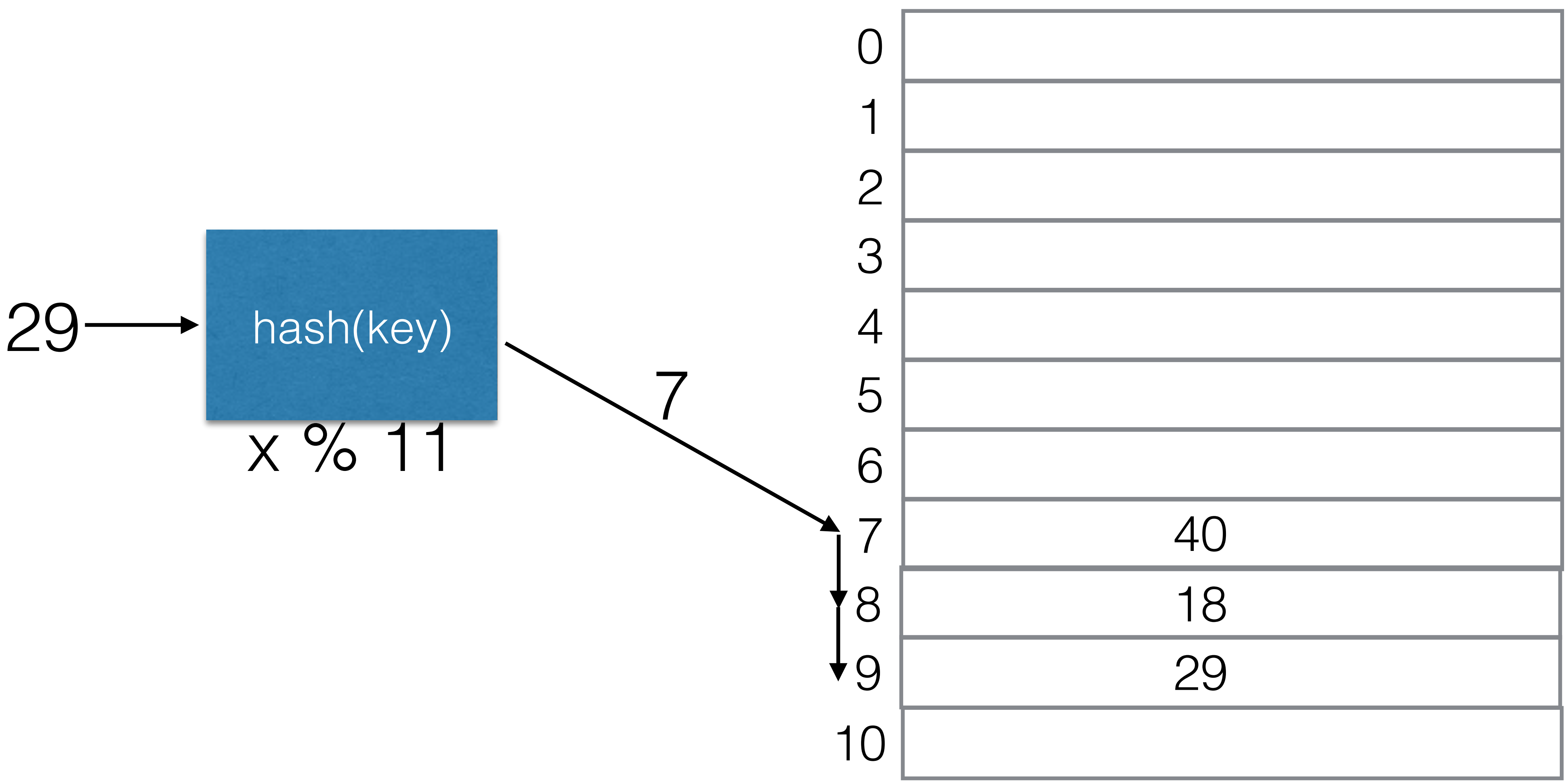
Hash Tables without Linked Lists: Probing

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



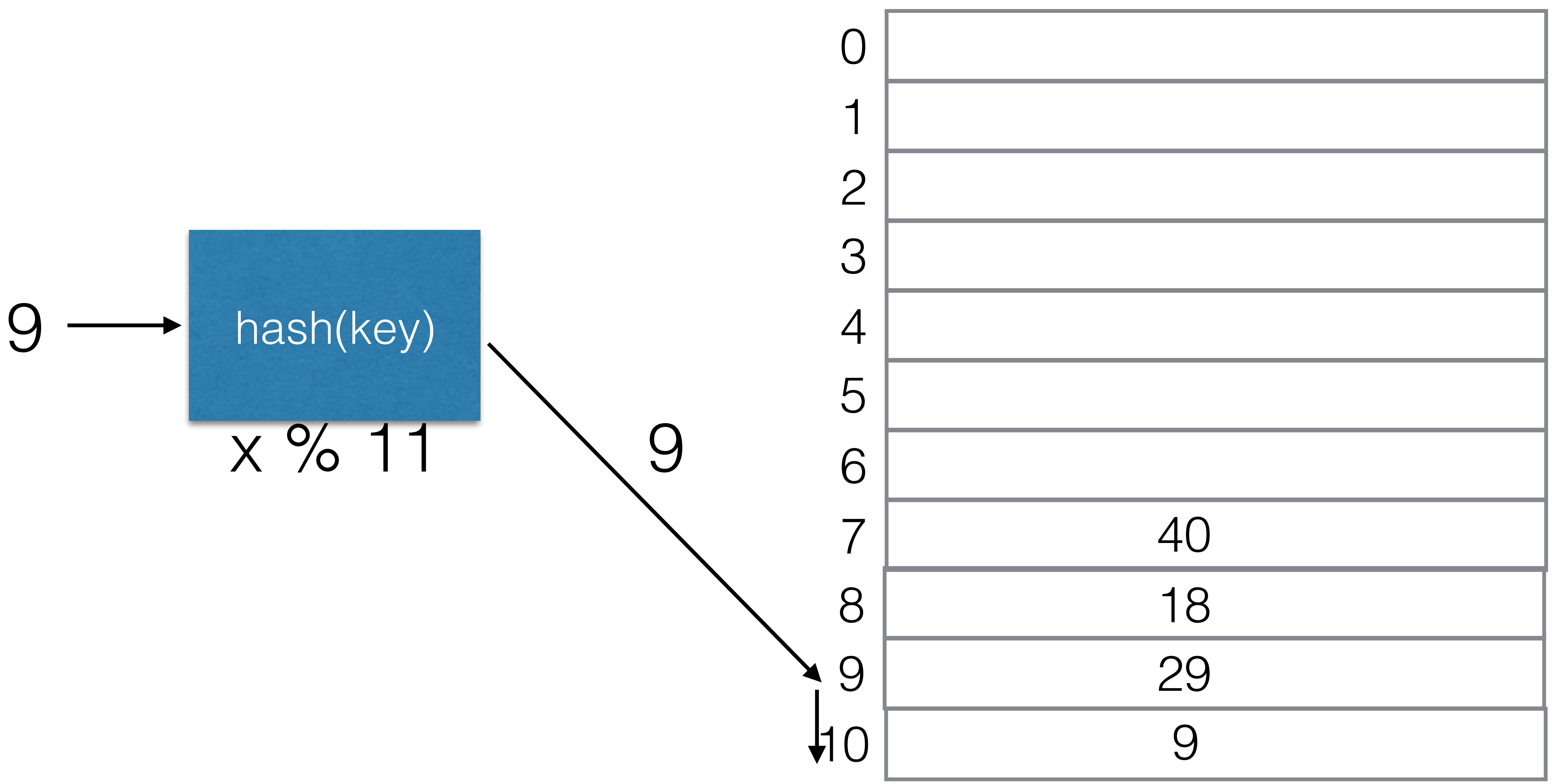
Hash Tables without Linked Lists: Probing

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



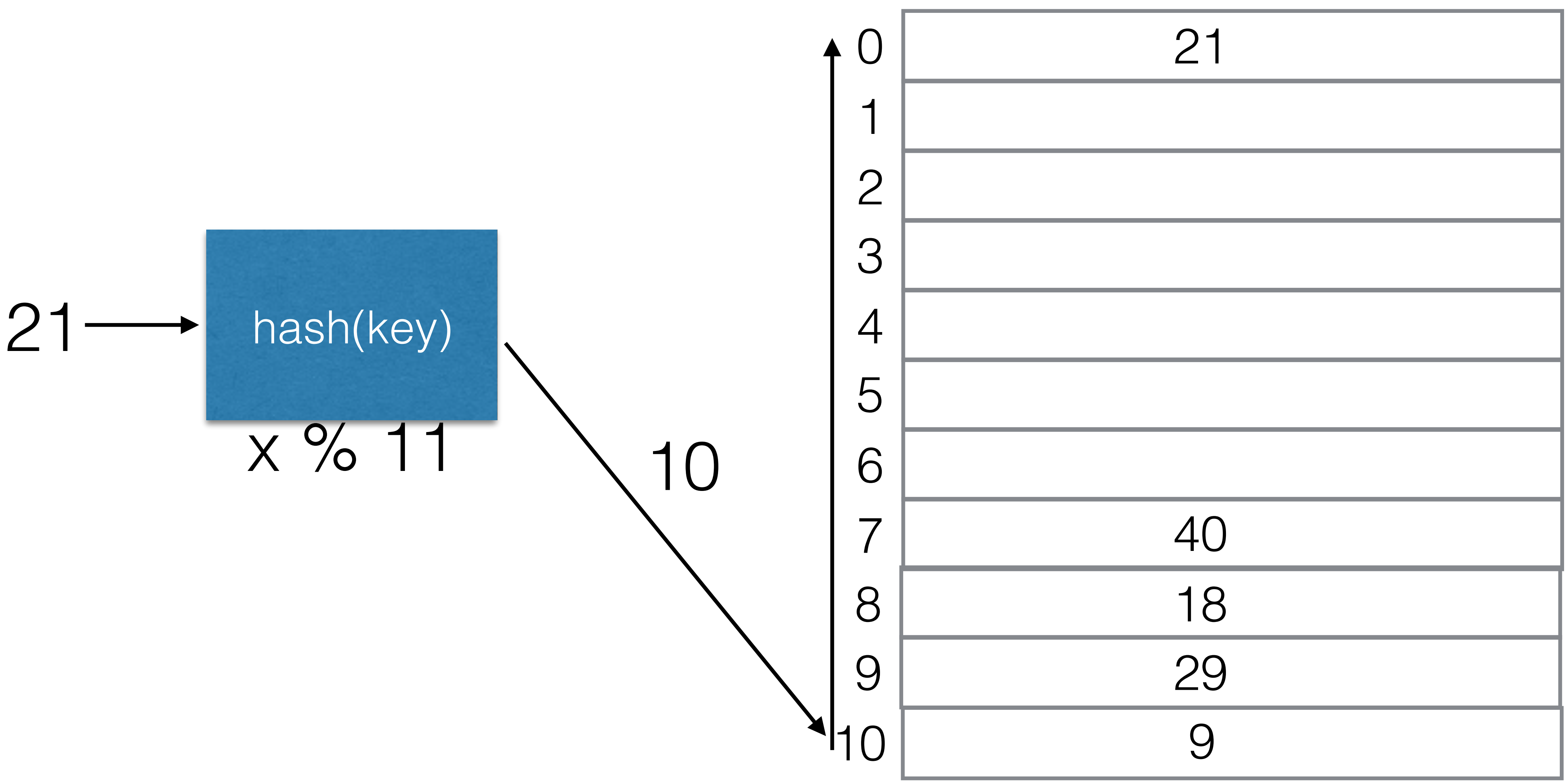
Hash Tables without Linked Lists: Probing

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



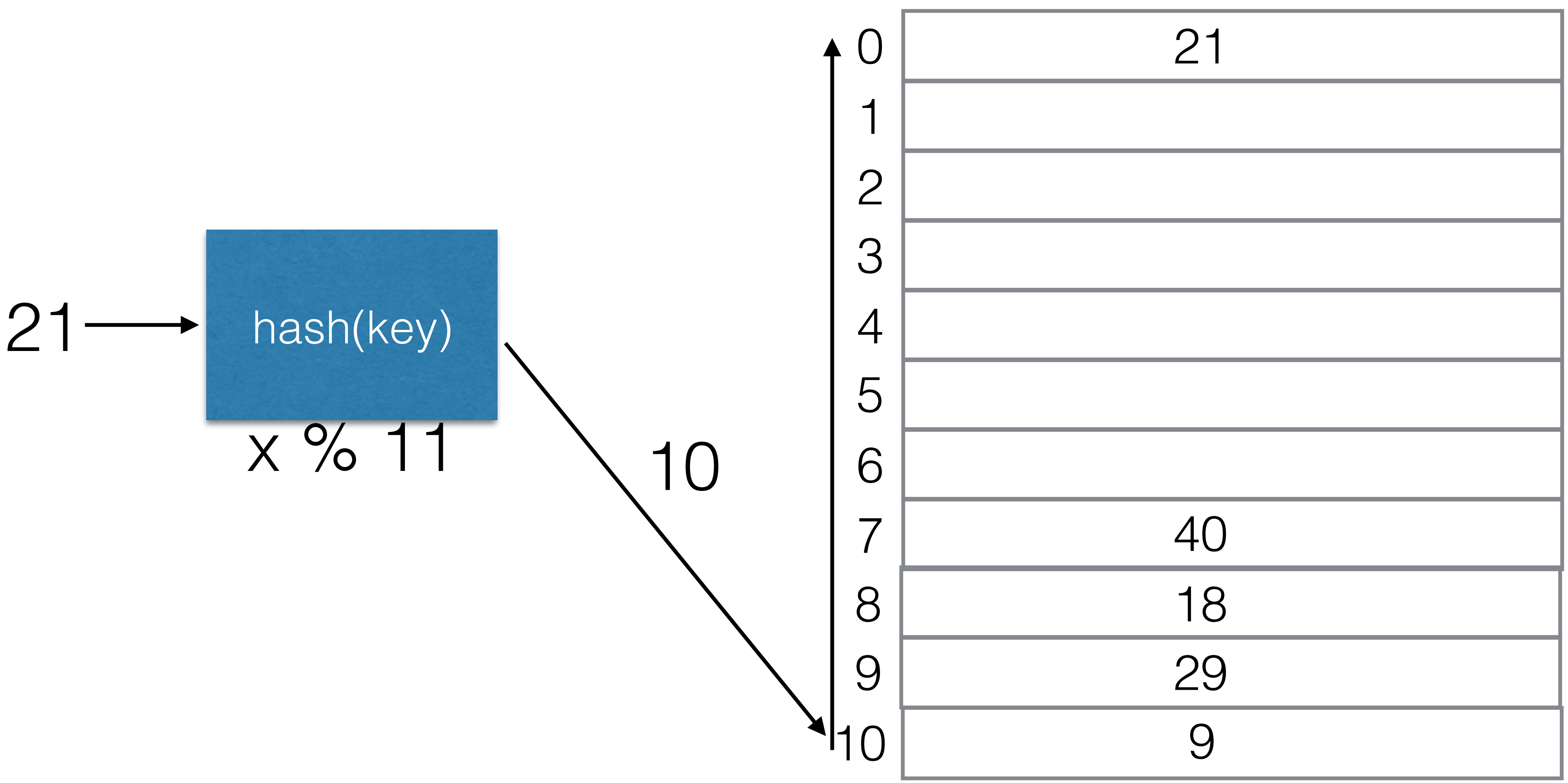
Hash Tables without Linked Lists: Probing

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



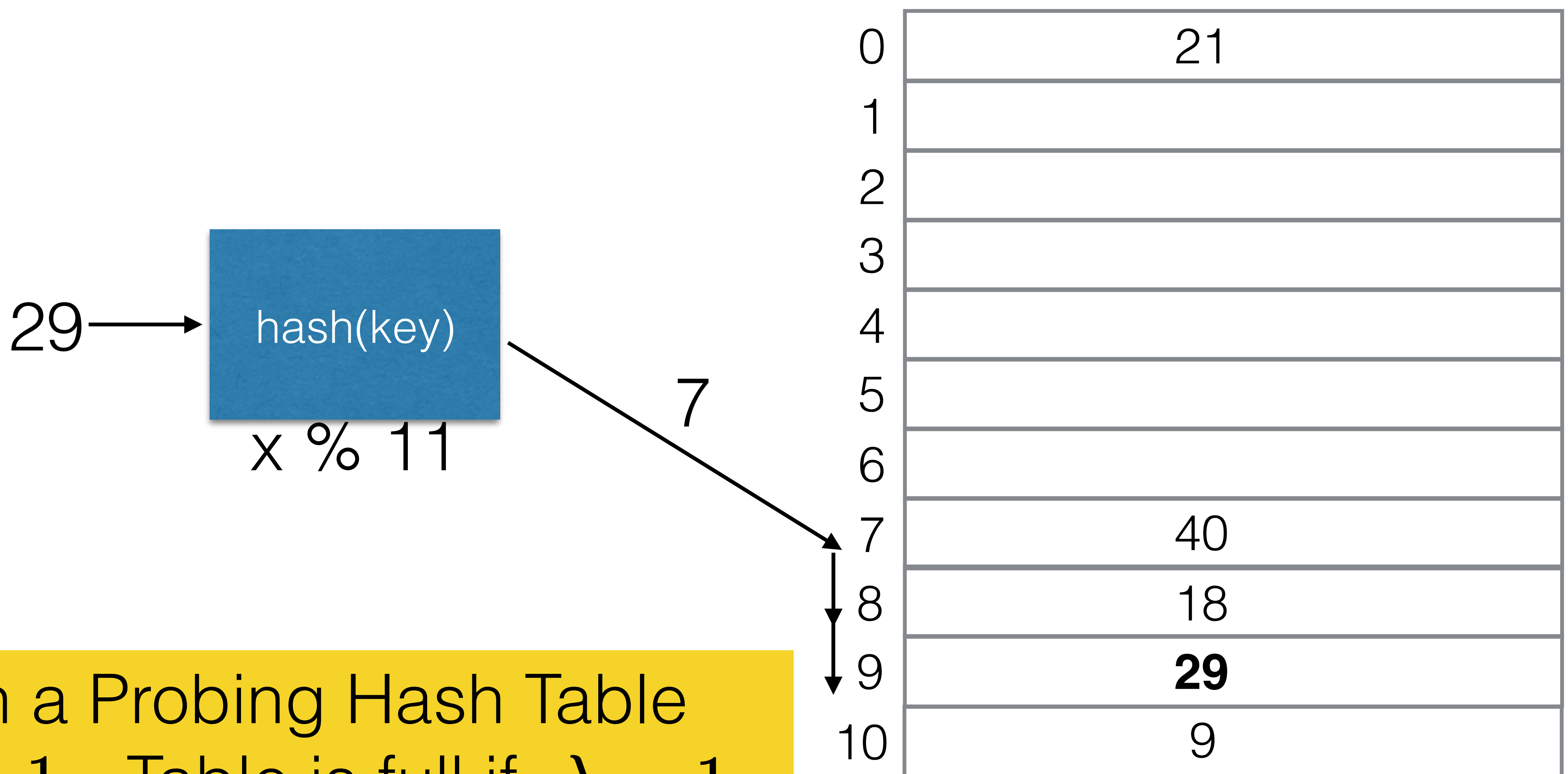
Hash Tables without Linked Lists: Probing

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



Hash Tables without Linked Lists: Probing

- To look up a key, we search the table, starting from the cell the key was hashed to.



With a Probing Hash Table $\lambda \leq 1$. Table is full if $\lambda = 1$.

Probing: Collision Resolution Strategies (1)

- 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).

Collision Resolution Strategies (2)

- Can describe collision resolution strategies using a function $f(i)$, such that the i-th table cell to be probed is
$$(hash(x) + f(i)) \,\%\, TableSize$$.

Collision Resolution Strategies (2)

- 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*, …

Collision Resolution Strategies (2)

- 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 $f(i) = i$

- 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).

Linear Probing $f(i) = i$

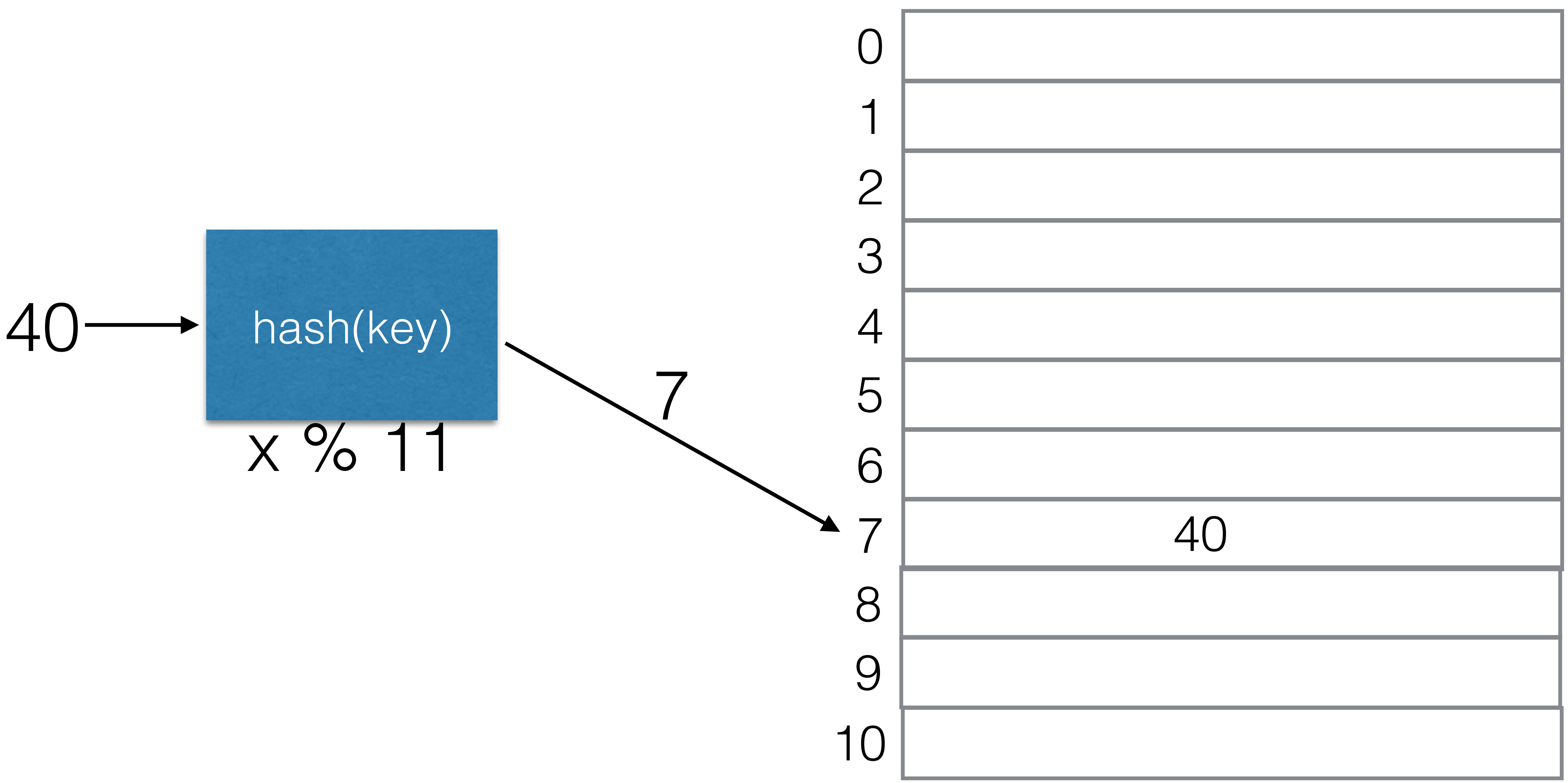
- Can always find an empty cell (if there is space in the table).

- Problem: **Primary Clustering.**

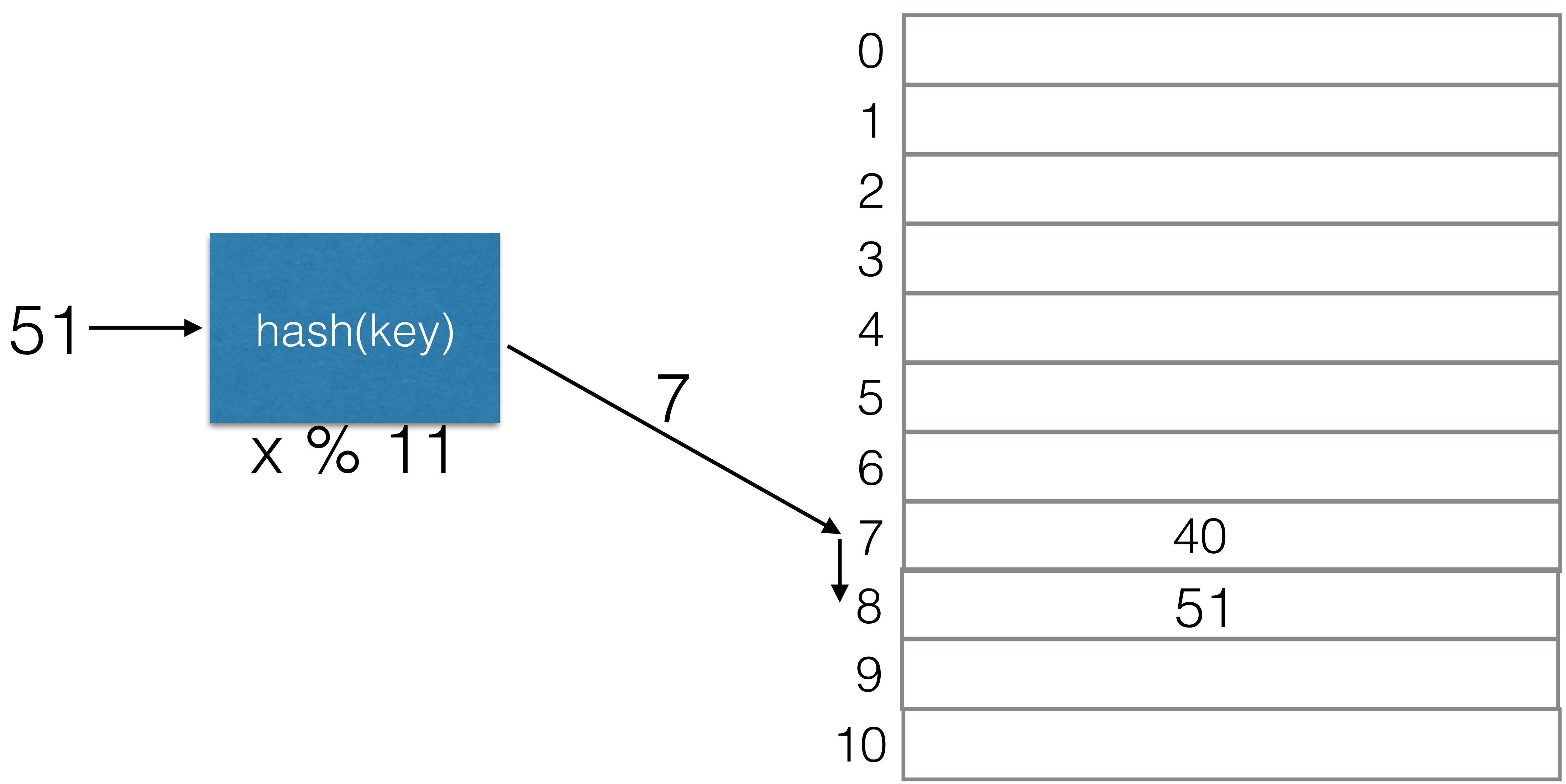
    - Full cells tend to cluster, with no free cells in between.

    - Time required to find an empty cell can become very large if the table is almost full (λ is close to 1).

Primary Clustering

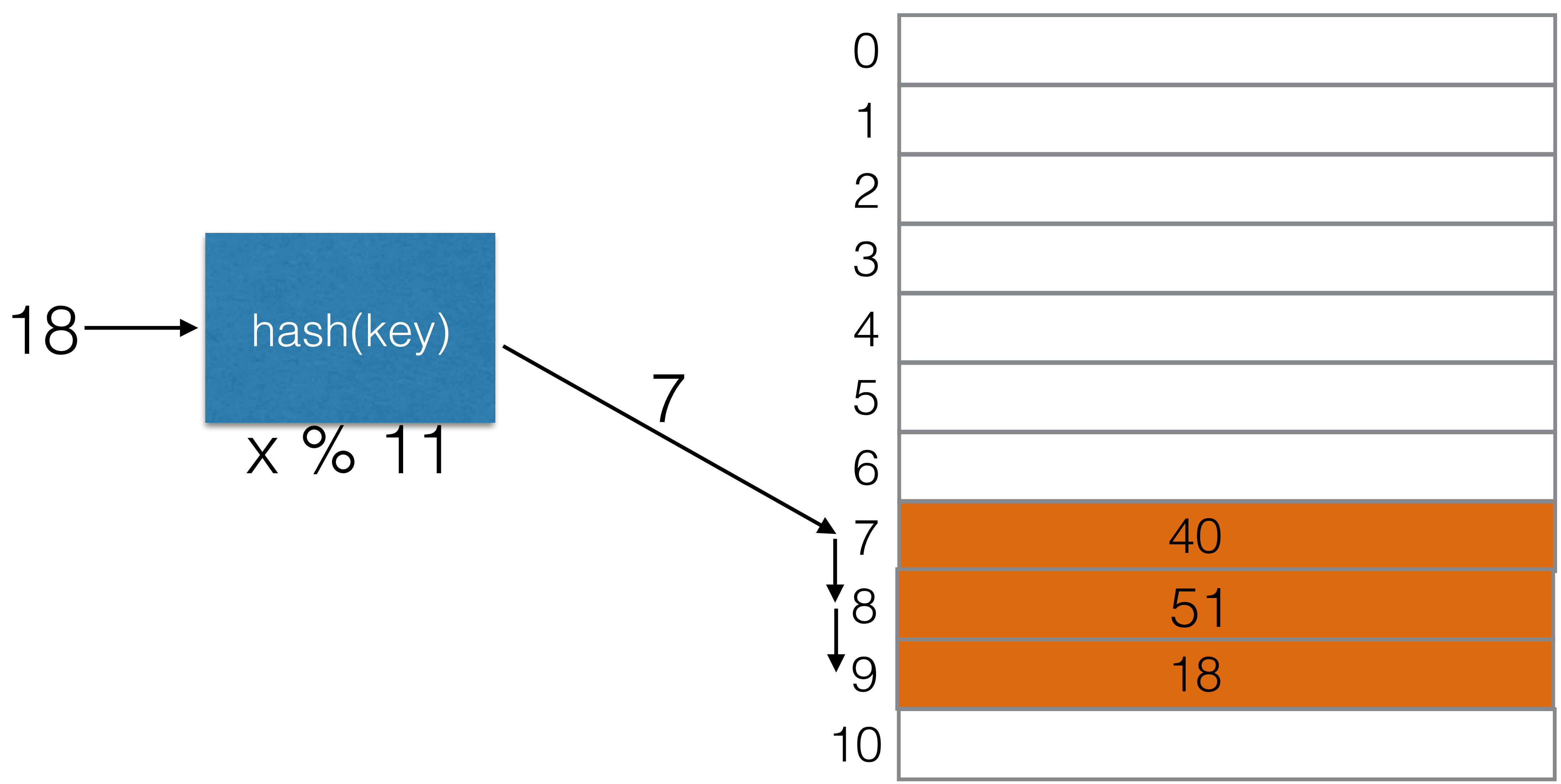


Primary Clustering



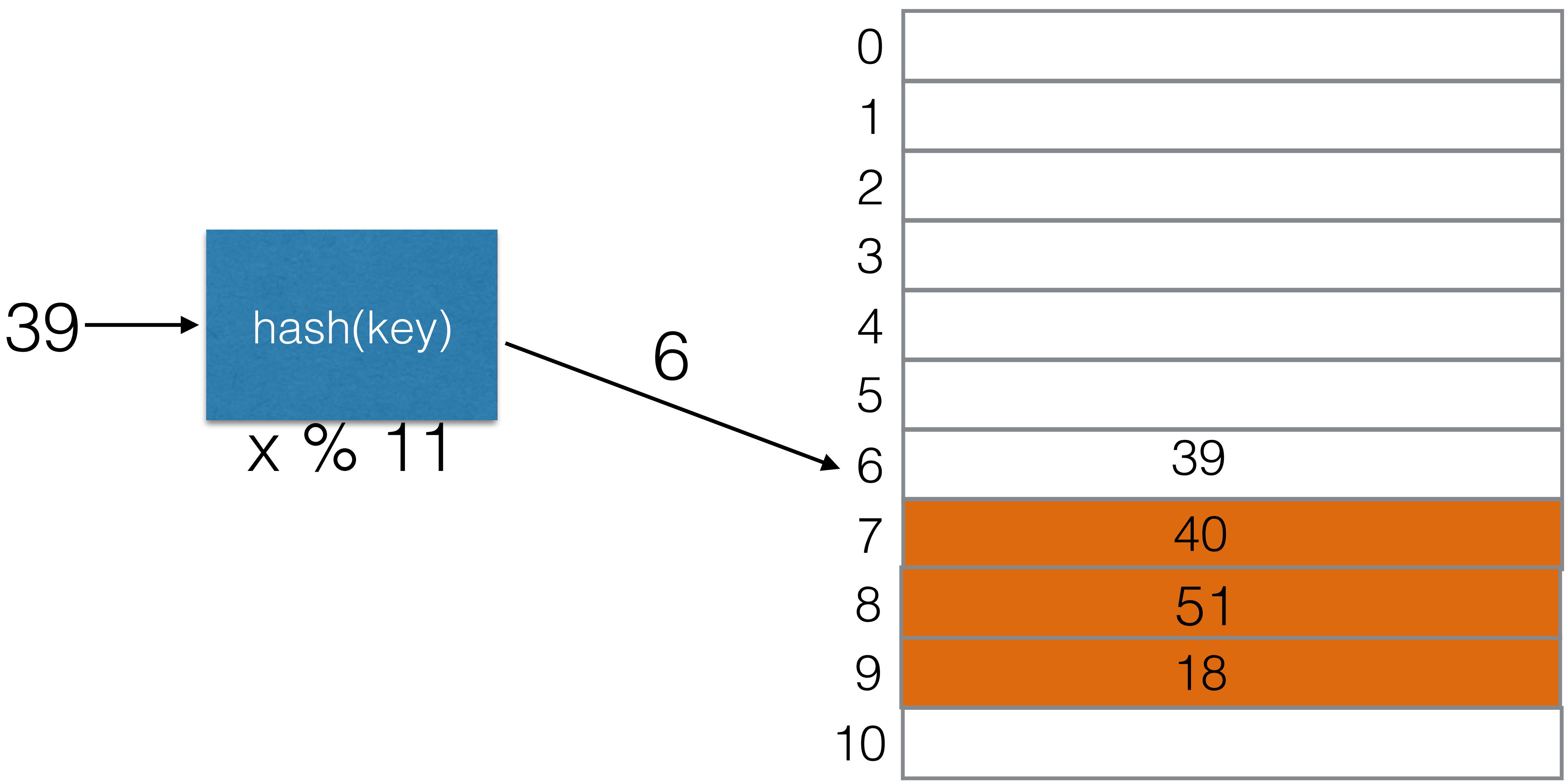
Primary Clustering

- Cells 7-9 are occupied with keys that hash to 7. The entire block is unavailable to keys that hash to k<7.



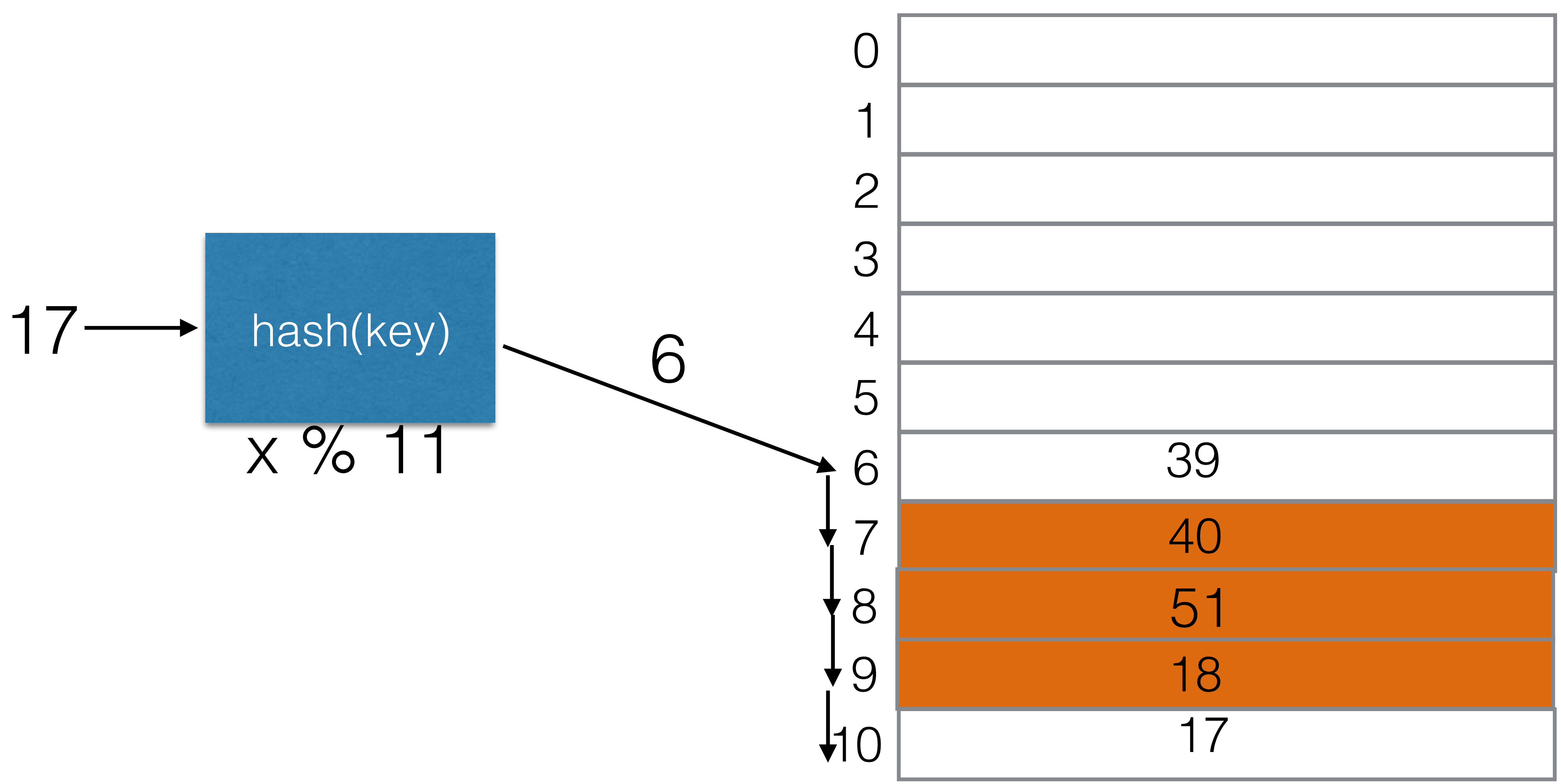
Primary Clustering

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



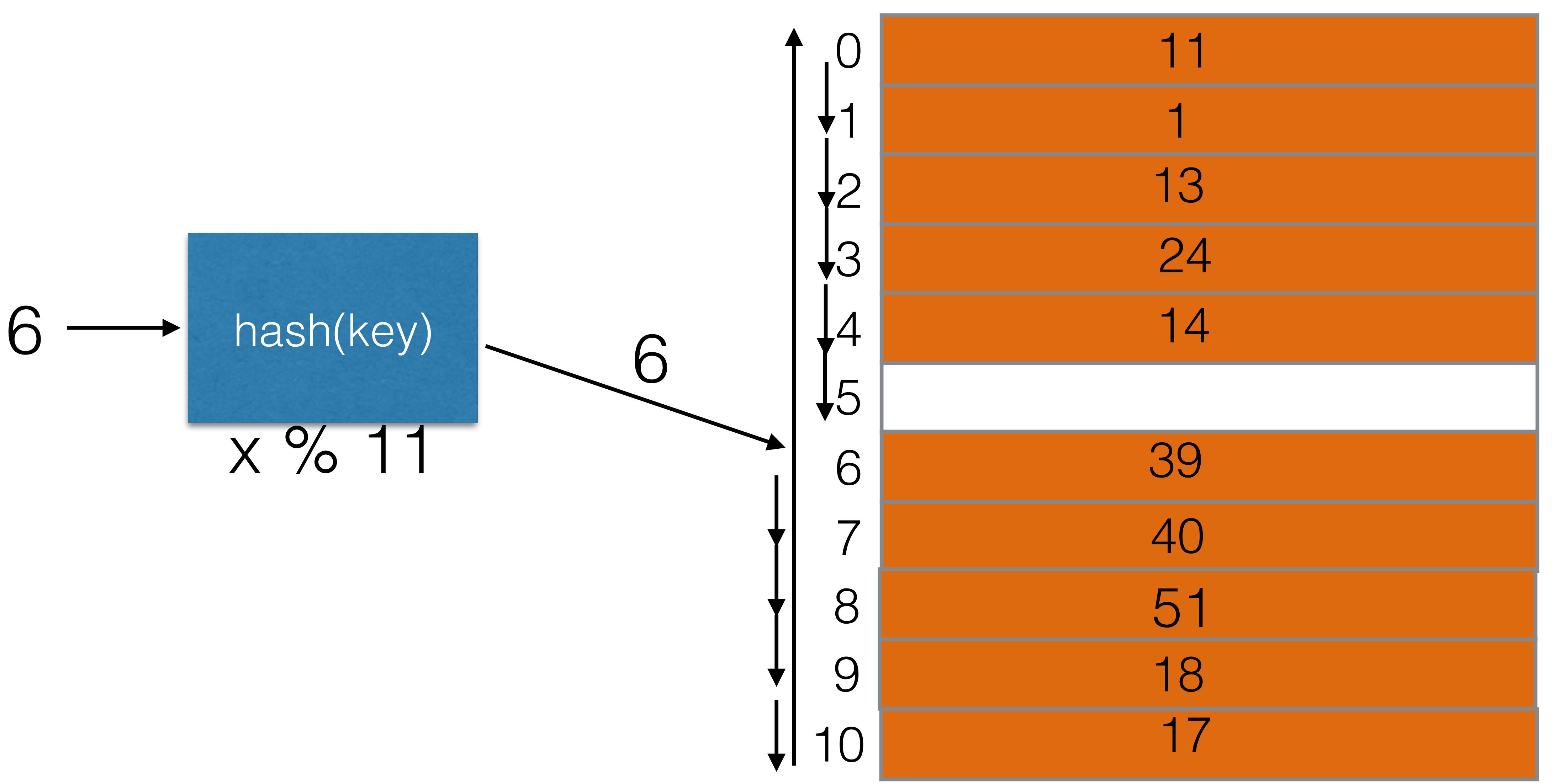
Primary Clustering

- 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

- This becomes really bad if λ is close to 1



Linear Probing vs. Choosing a Random Cell

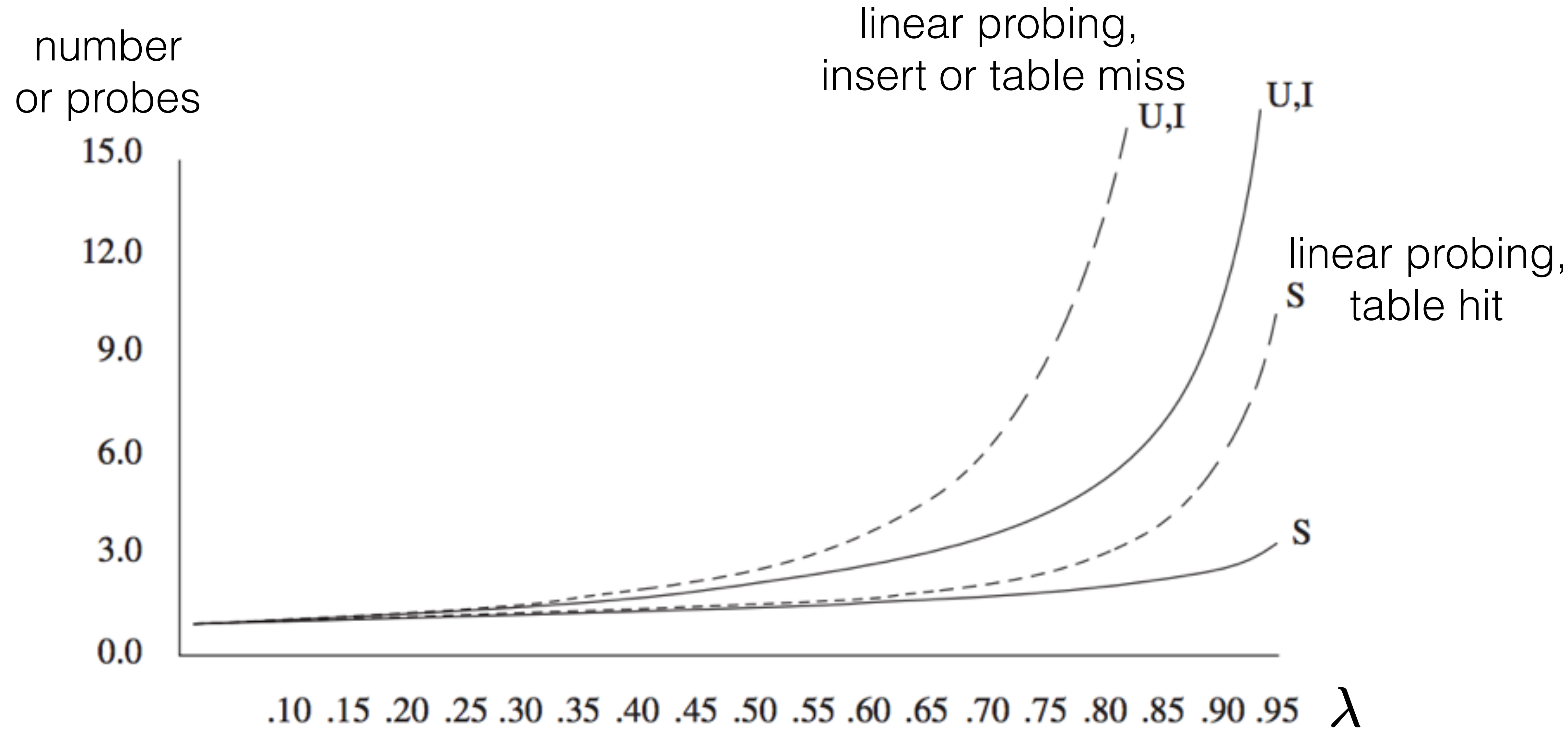
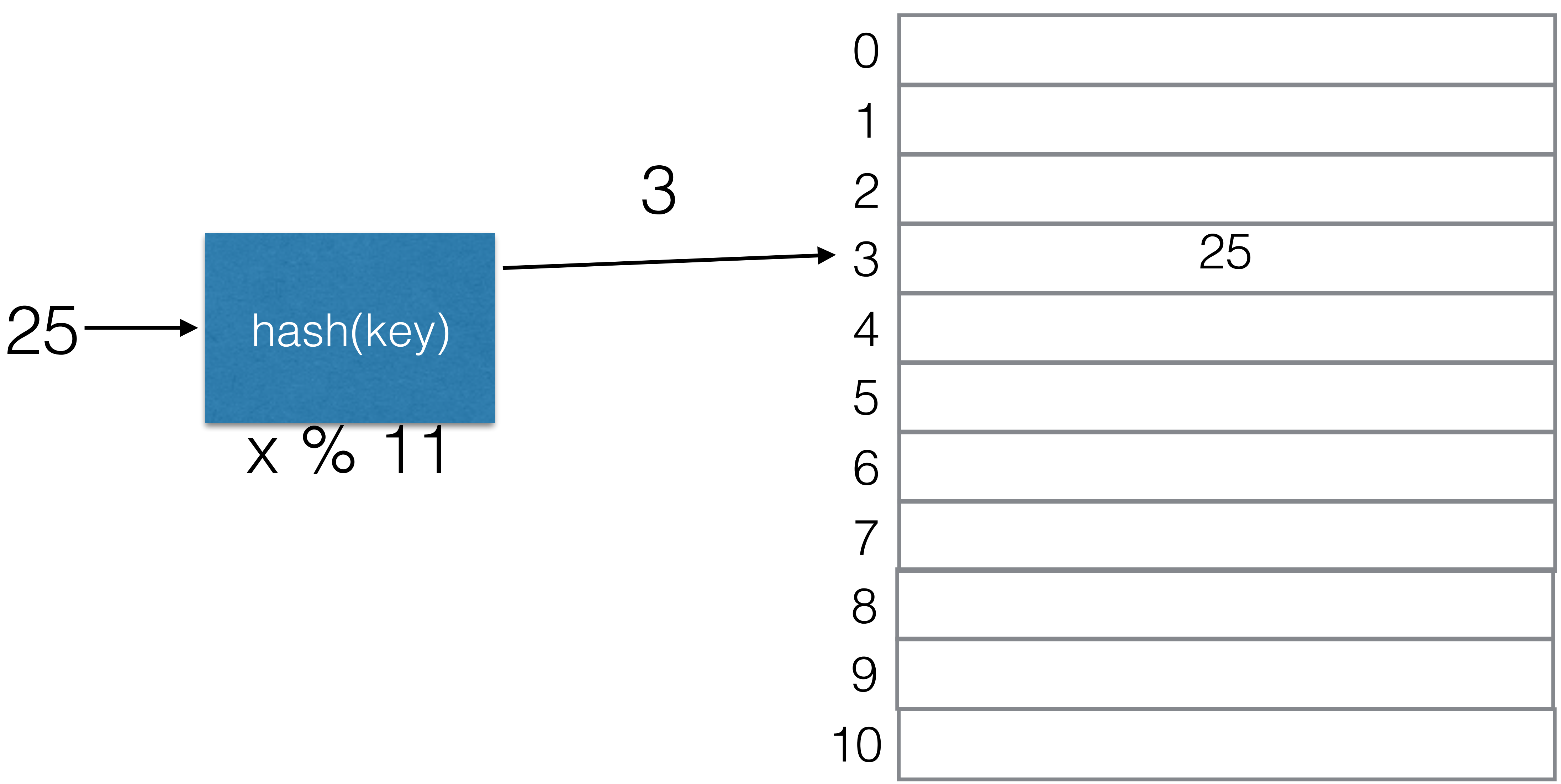


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed.

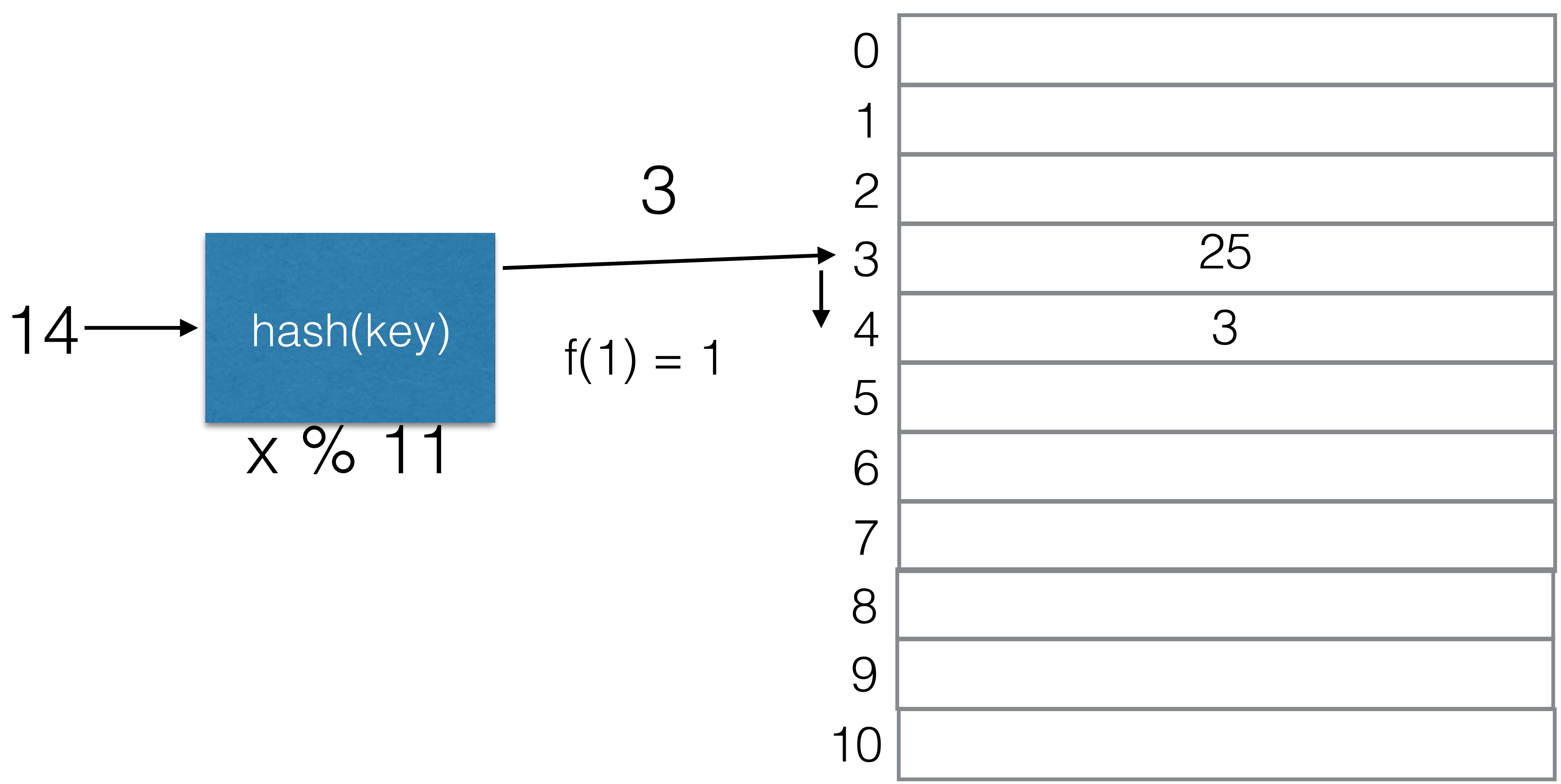
Quadratic Probing

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



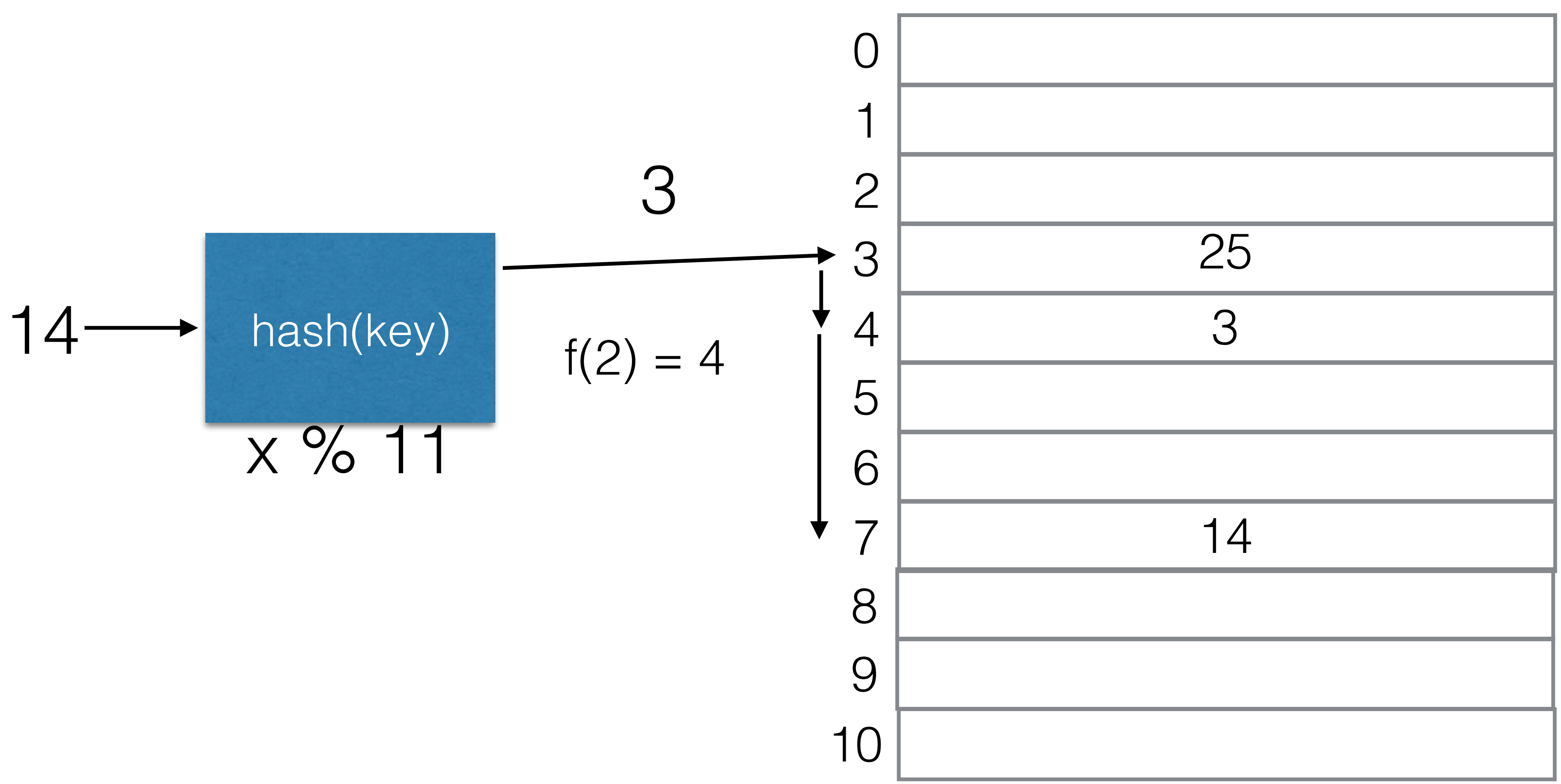
Quadratic Probing

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



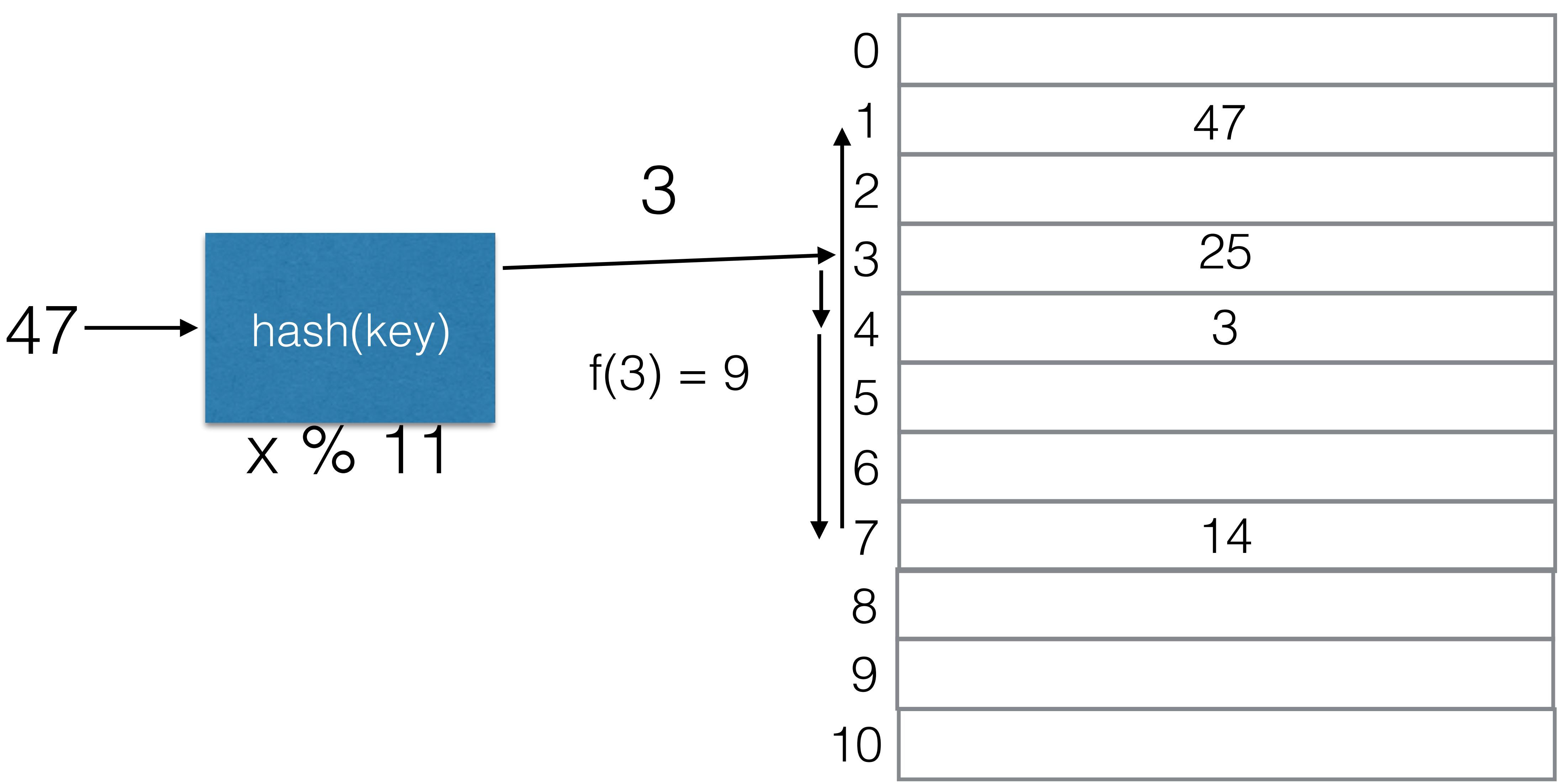
Quadratic Probing

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



Quadratic Probing

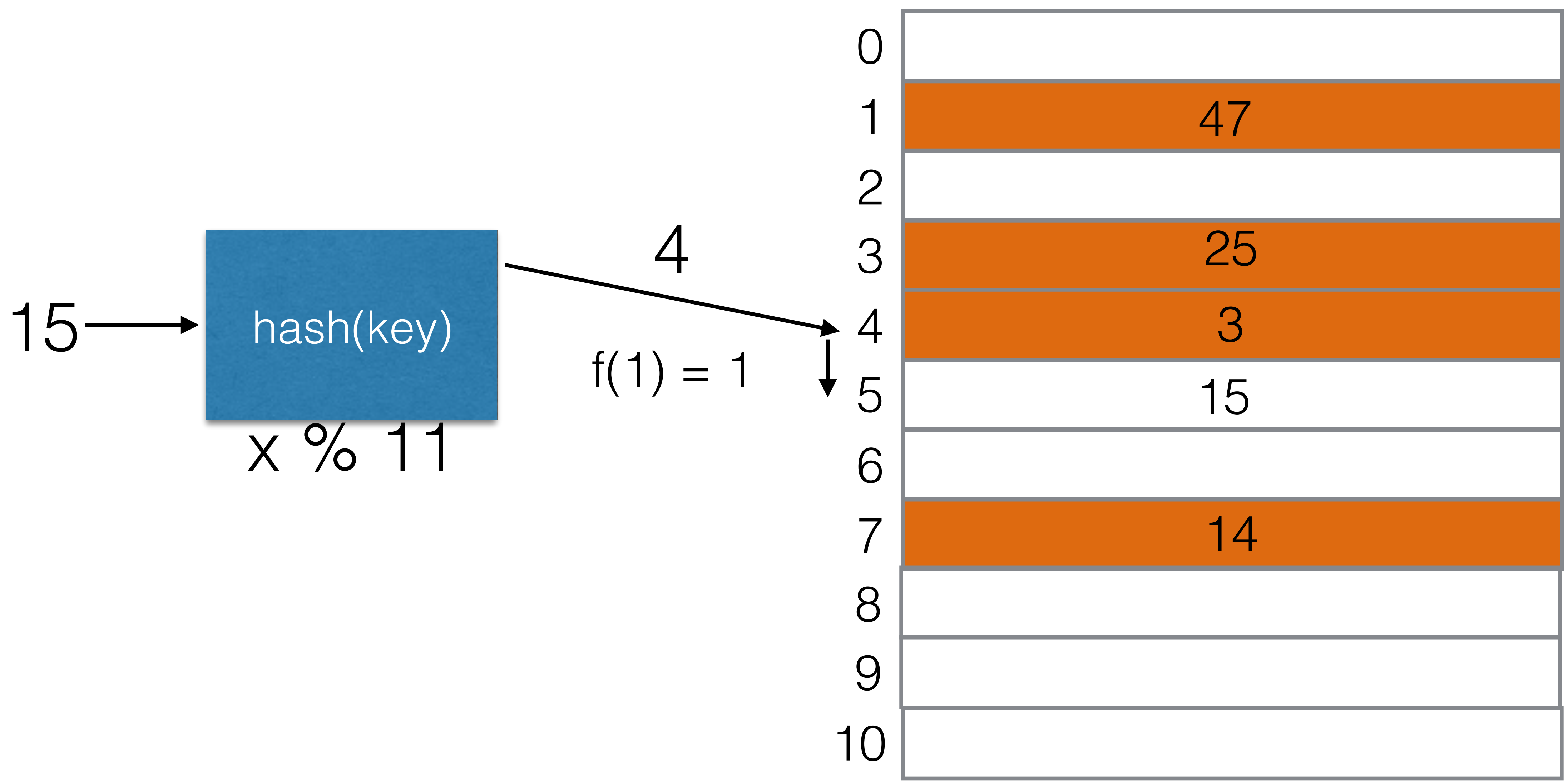
$$(hash(x) + f(i))\ \%\ TableSize \qquad f(i) = i^2$$



Quadratic Probing

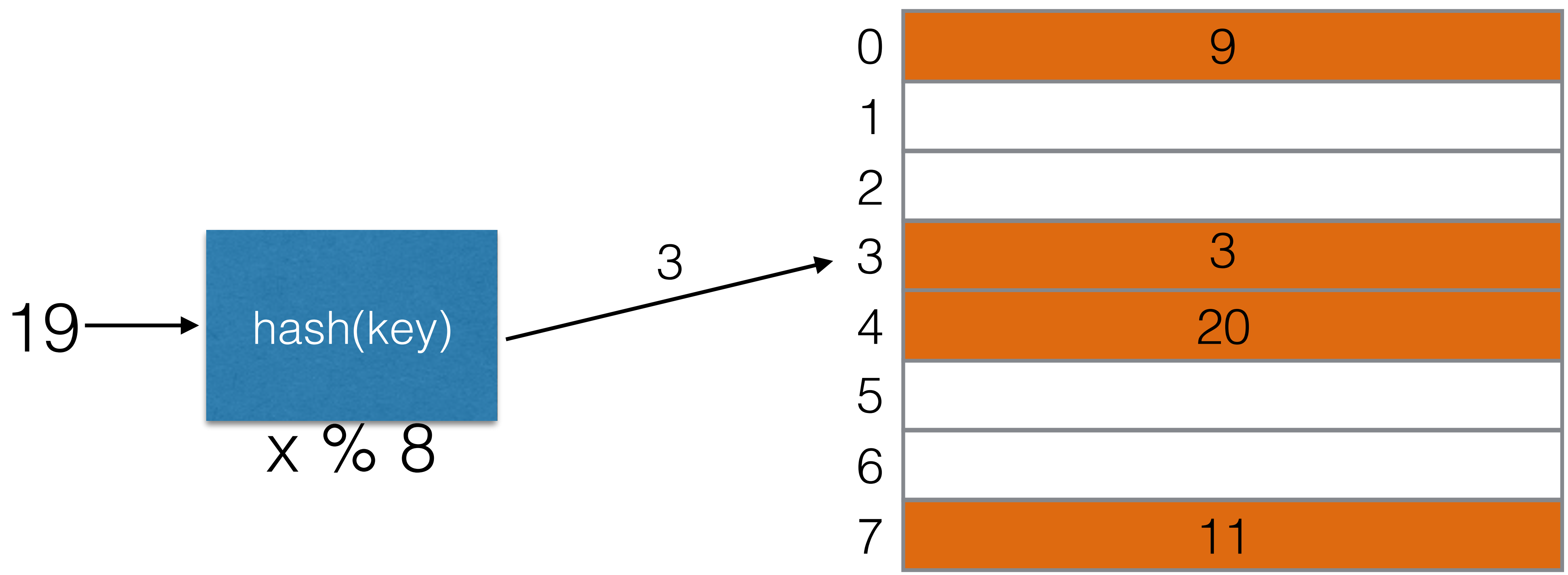
$$(hash(x) + f(i)) \ \% \ TableSize \qquad\qquad f(i) = i^2$$

- Primary clustering is not a problem.



Quadratic Probing

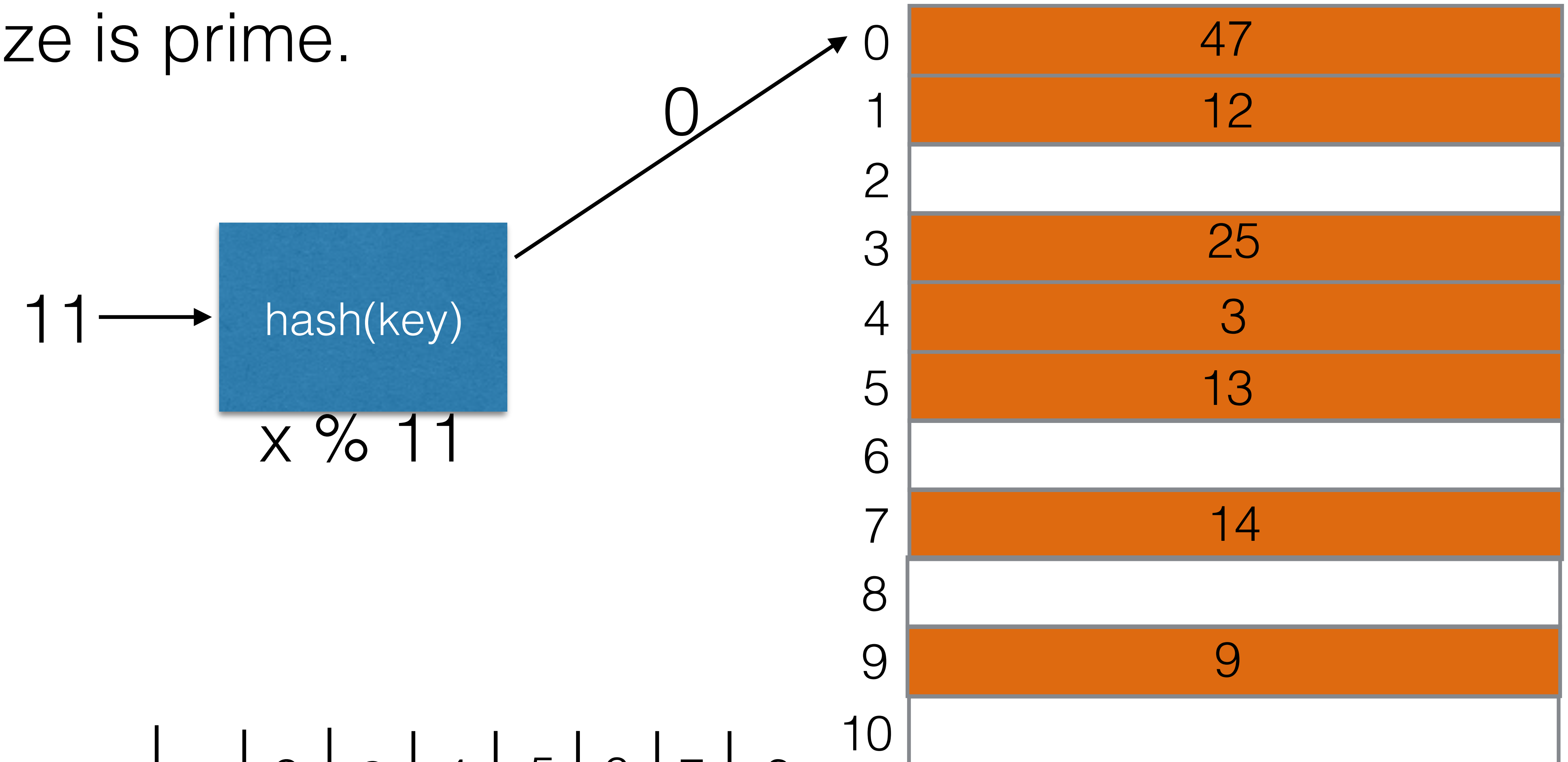
- Important: With quadratic probing, *TableSize* should be a prime number! Otherwise it is possible that we won't find an empty cell, even if there is plenty of space.



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

Quadratic Probing

- 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

…

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

Quadratic Probing Theorem

If *TableSize* is prime, then the first $\frac{TableSize}{2}$ cells visited by quadratic probing are distinct.
Therefore we can always find an empty cell if the table is at most half full.

Quadratic Probing Theorem

If *TableSize* is prime, then the first $\frac{TableSize}{2}$ cells visited by quadratic probing are distinct.
Therefore we can always find an empty cell if the table is at most half full.

- Let *TableSize* be some prime greater than 3.
- Let hash(x) = h
- If there was a slot visited twice during the first $\frac{TableSize}{2}$ probing steps, then there must be two numbers

$$0 \leq i < j \leq \frac{TableSize}{2} \quad \text{such that}$$

$$(h + i^2) \,\%\, TableSize = (h + j^2) \,\%\, TableSize$$

Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first $\frac{TableSize}{2}$ probing steps, then there must be two numbers

$$0 \leq i < j \leq \frac{TableSize}{2}$$ such that

$$(h + i^2) \,\%\, TableSize = (h + j^2) \,\%\, TableSize$$

$$h + i^2 = h + j^2$$

$$i^2 = j^2$$

$$i^2 - j^2 = 0$$

$$(i - j)(i + j) = 0$$

Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first $\frac{TableSize}{2}$ probing steps, then there must be two numbers

$$0 \leq i < j \leq \frac{TableSize}{2}$$ such that

$$(h + i^2)\ \%\ TableSize = (h + j^2)\ \%\ TableSize$$

$$h + i^2 = h + j^2$$

$$i^2 = j^2$$

$$i^2 - j^2 = 0$$

$$(i - j)(i + j) = 0$$

either $(i - j)(i + j) = TableSize$

or $(i - j) = 0$ or $(i + j) = 0$

Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first $\frac{TableSize}{2}$ probing steps, then there must be two numbers

$$0 \le i < j \le \frac{TableSize}{2} \quad \text{such that}$$

$$(h + i^2)\ \%\ TableSize = (h + j^2)\ \%\ TableSize$$

$$h + i^2 = h + j^2$$

$$i^2 = j^2$$

$$i^2 - j^2 = 0$$

$$(i - j)(i + j) = 0$$

either ~~$(i - j)(i + j) = TableSize$~~

or ~~$(i - j) = 0$~~ or ~~$(i + j) = 0$~~

impossible because TableSize is prime

impossible because i < j

impossible because i<j≤TableSize/2

Quadratic Probing Theorem (2)

Proof by contradiction:

If there is an index visited twice during the first $\frac{TableSize}{2}$ probing steps, then there must be two numbers

$$0 \leq i < j \leq \frac{TableSize}{2} \quad \text{such that}$$

$$(h + i^2)\ \%\ TableSize = (h + j^2)\ \%\ TableSize$$

$$h + i^2 = h + j^2$$

$$i^2 = j^2$$

$$i^2 - j^2 = 0$$

$$(i - j)(i + j) = 0$$

impossible because TableSize is prime

Contradiction!

The assumption must be false!

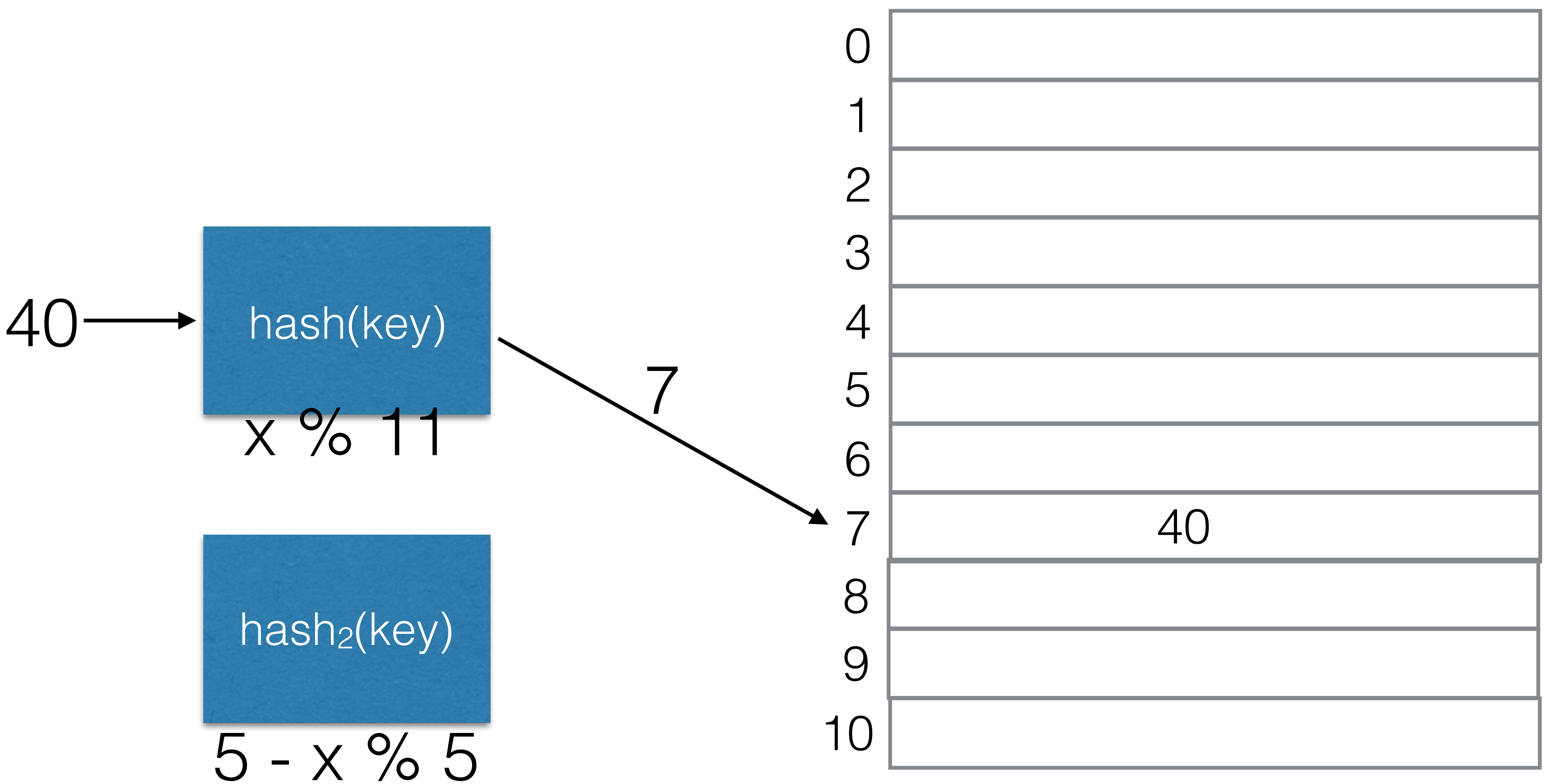
impossible because i < j

impossible because i<j≤TableSize/2

Double Hashing

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

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

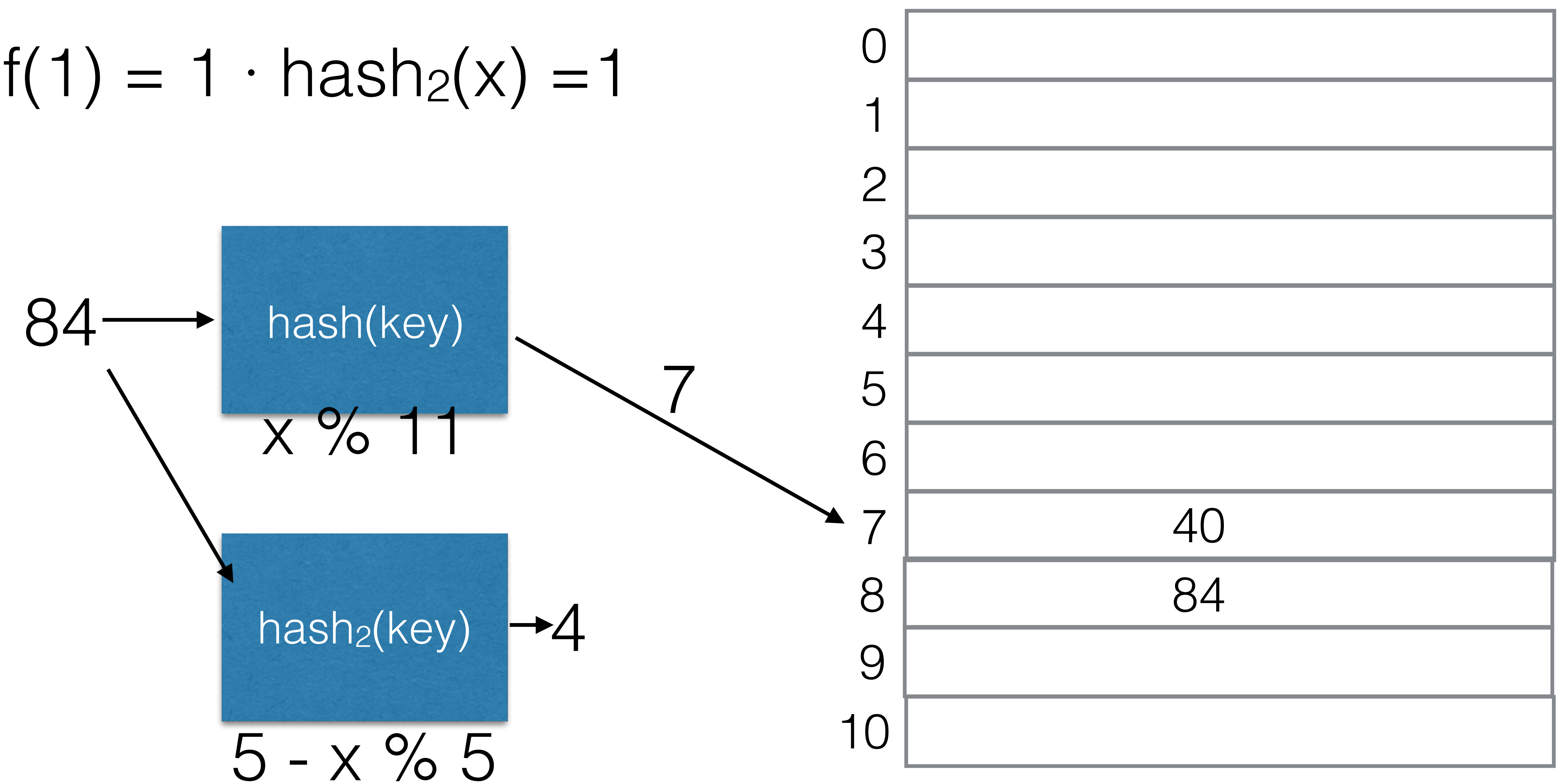


Double Hashing

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

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

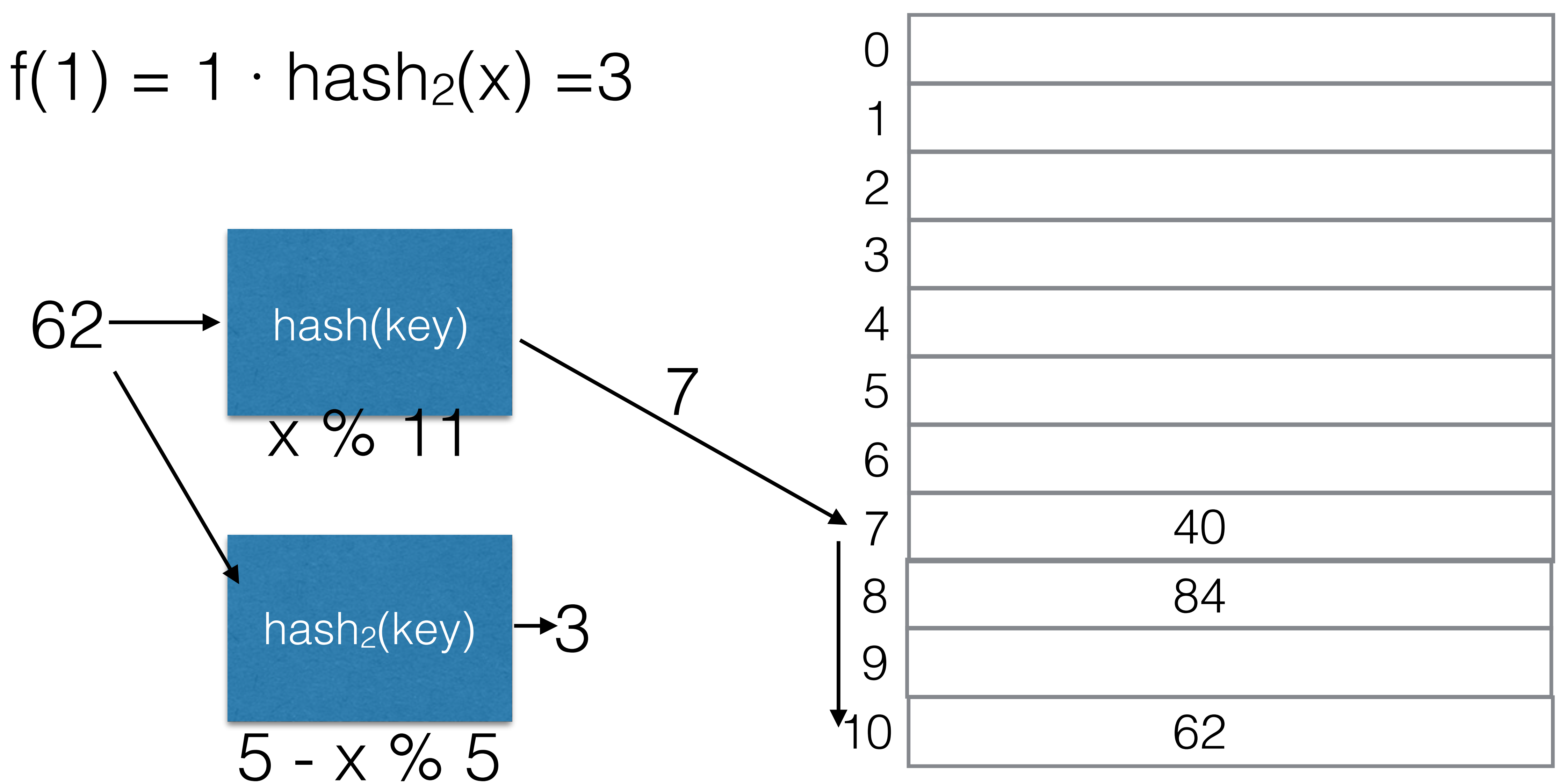
$f(1) = 1 \cdot hash_2(x) = 1$



Double Hashing

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

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



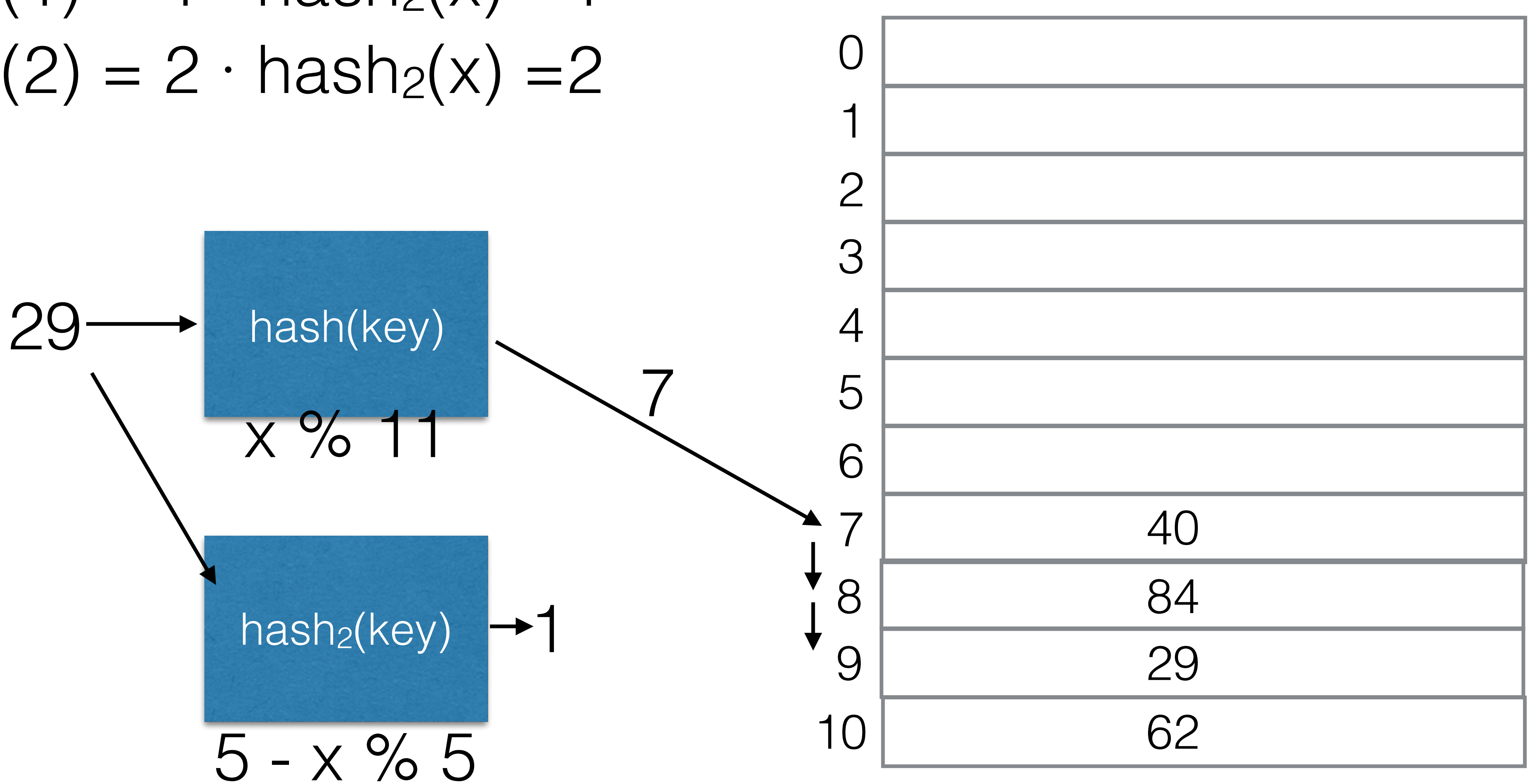
Double Hashing

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

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

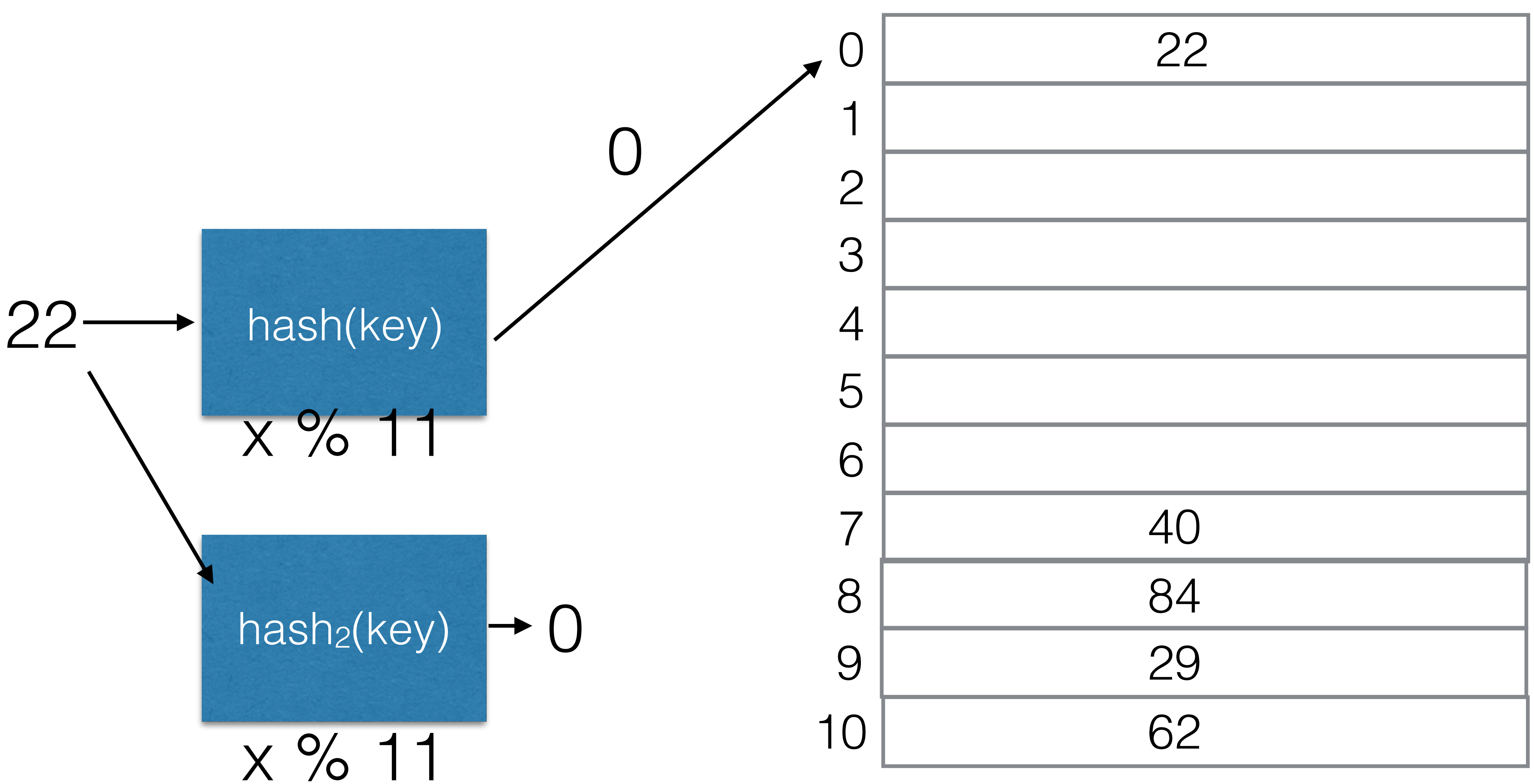
$f(1) = 1 \cdot hash_2(x) = 1$

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



Choosing a Secondary Hash Function

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



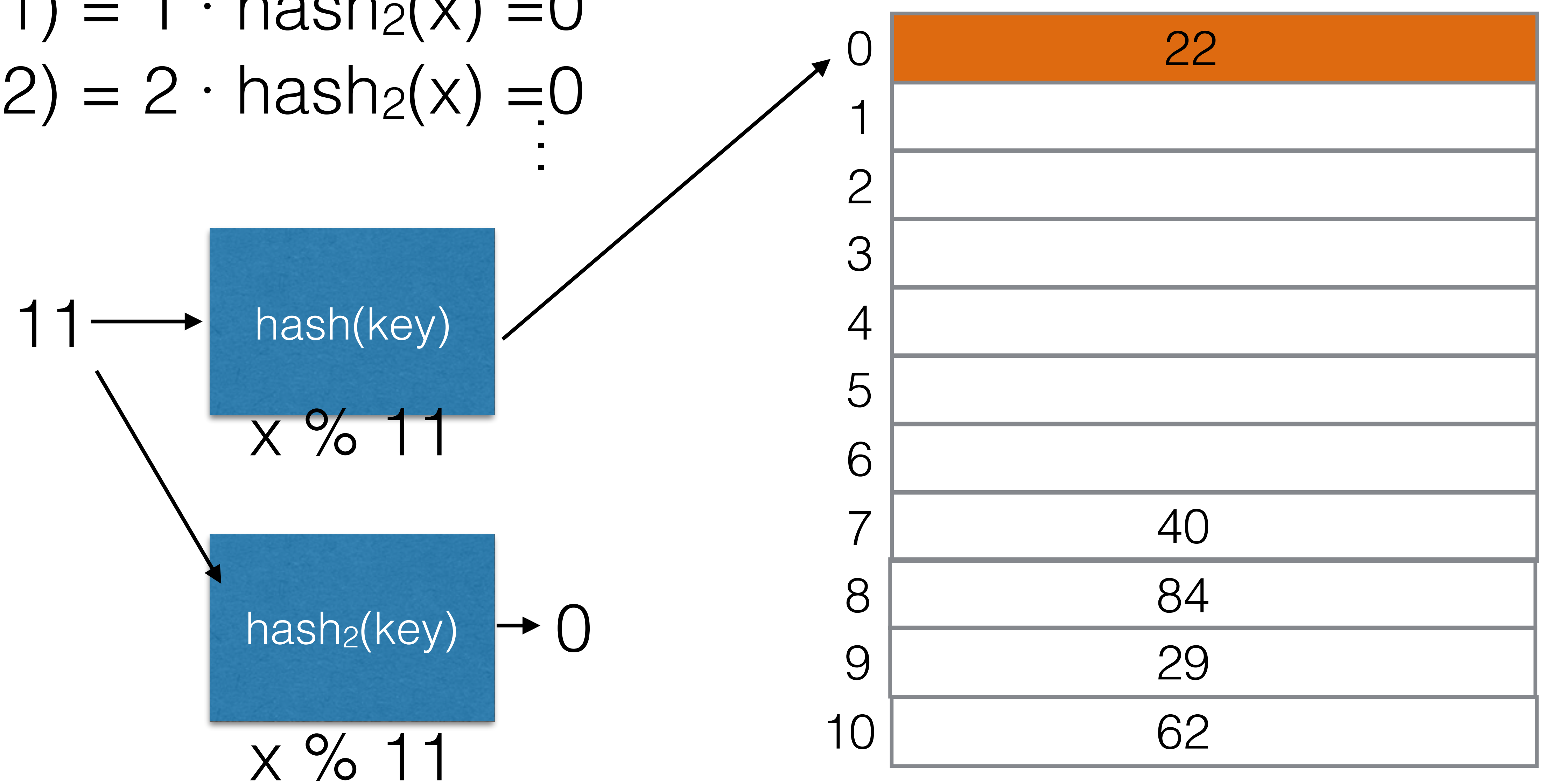
Choosing a Secondary Hash Function

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

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

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

$\vdots$



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.

Rehashing

- 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.
- 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.

Rehashing Running Time

- Remove all N items and re-insert into the new table.
- Every insert is O(1), so rehashing takes O(N).
- But rehashing is relatively rare, we need to do it only after every TableSize/2 inserts.