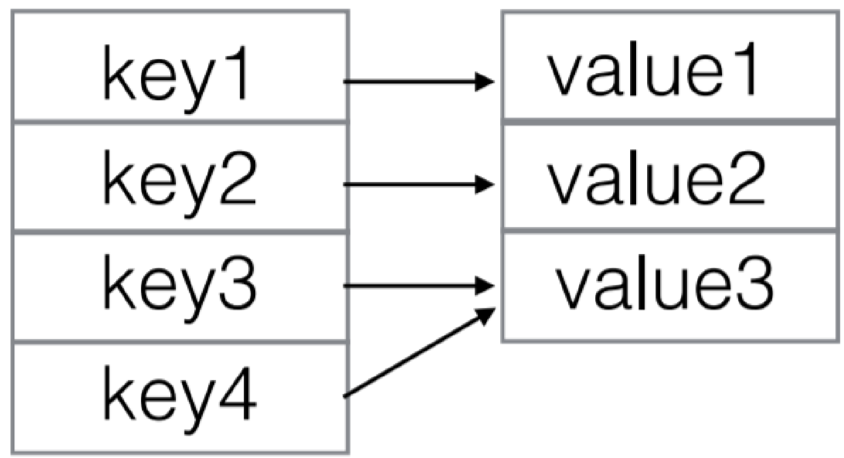
Hashing

Slides adapted from Daniel Bauer

Map ADT

- A *map* is a collection of *(key, value)* pairs
- Keys are unique, values may not be
- Two operations:
  - `get(key)`
  - `put(key, value)`

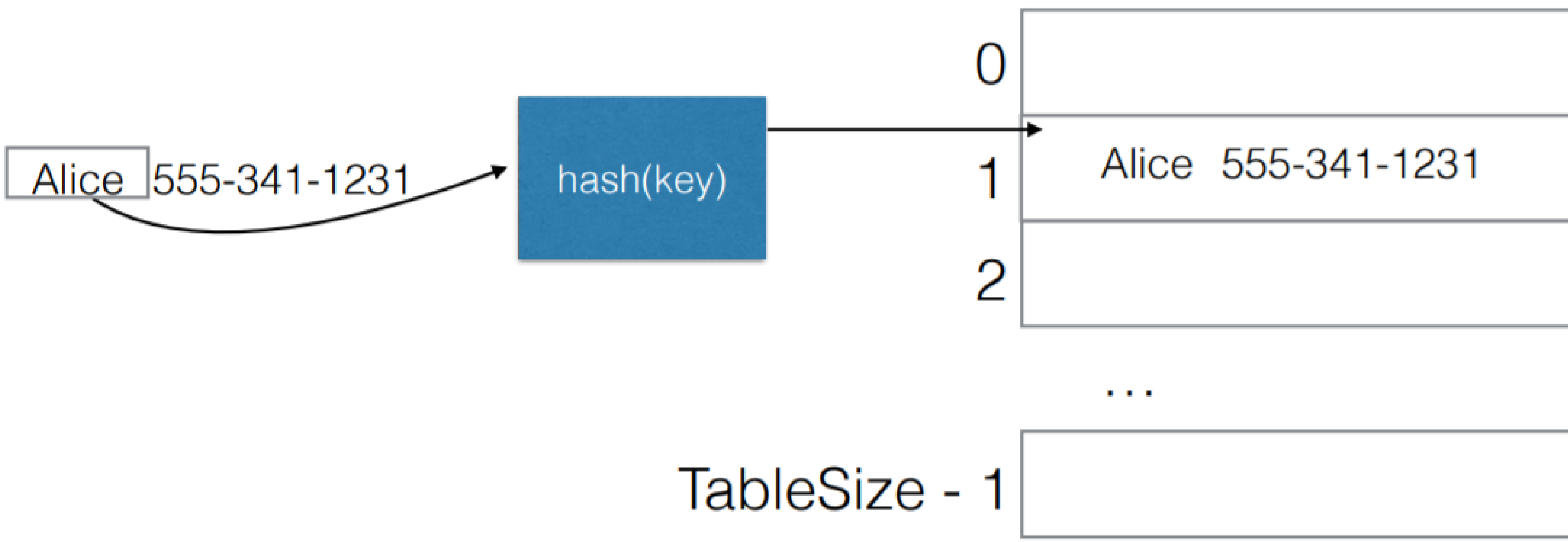


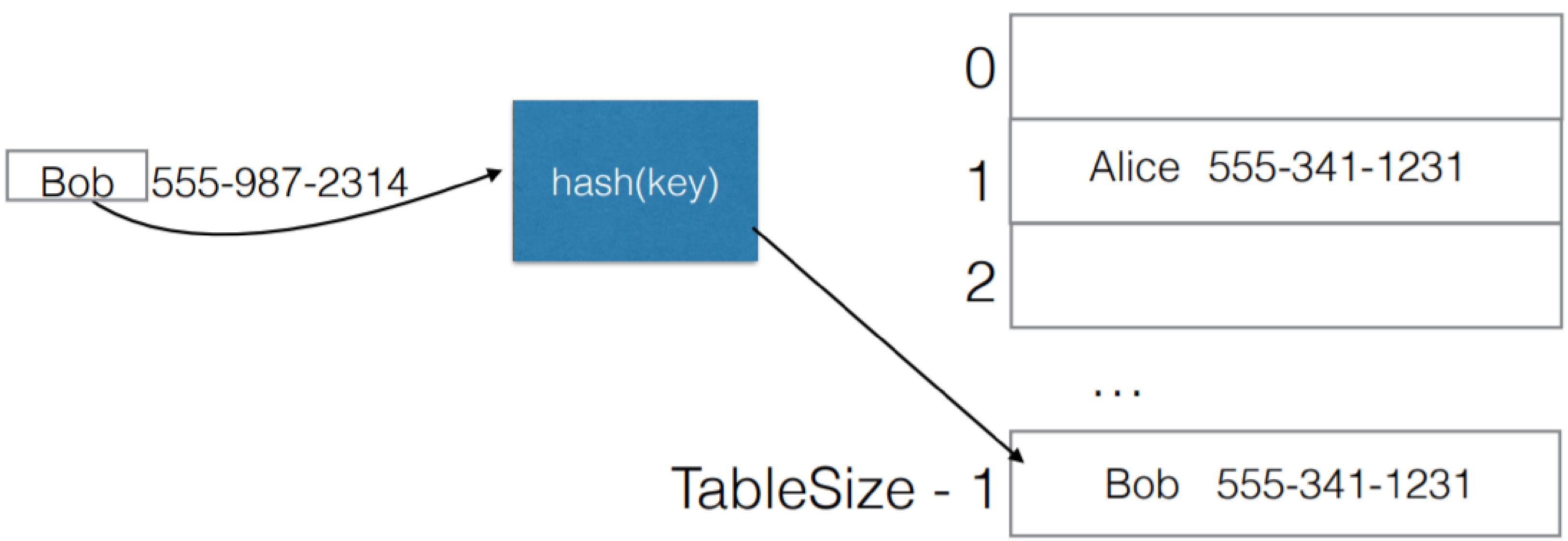
Implementing Maps

- Store *(key, value)* pairs in a Set
  - For example, store them in an AVL Tree, with the comparator comparing keys
  - $O(\log(n))$ `get` and `put`
- Use an array of values
  - Only integer keys permitted, the array may need to be large
  - $O(1)$ `get` and `put`
- Use hash tables
  - An extension of the array idea, but without needing very large arrays

Hash Tables

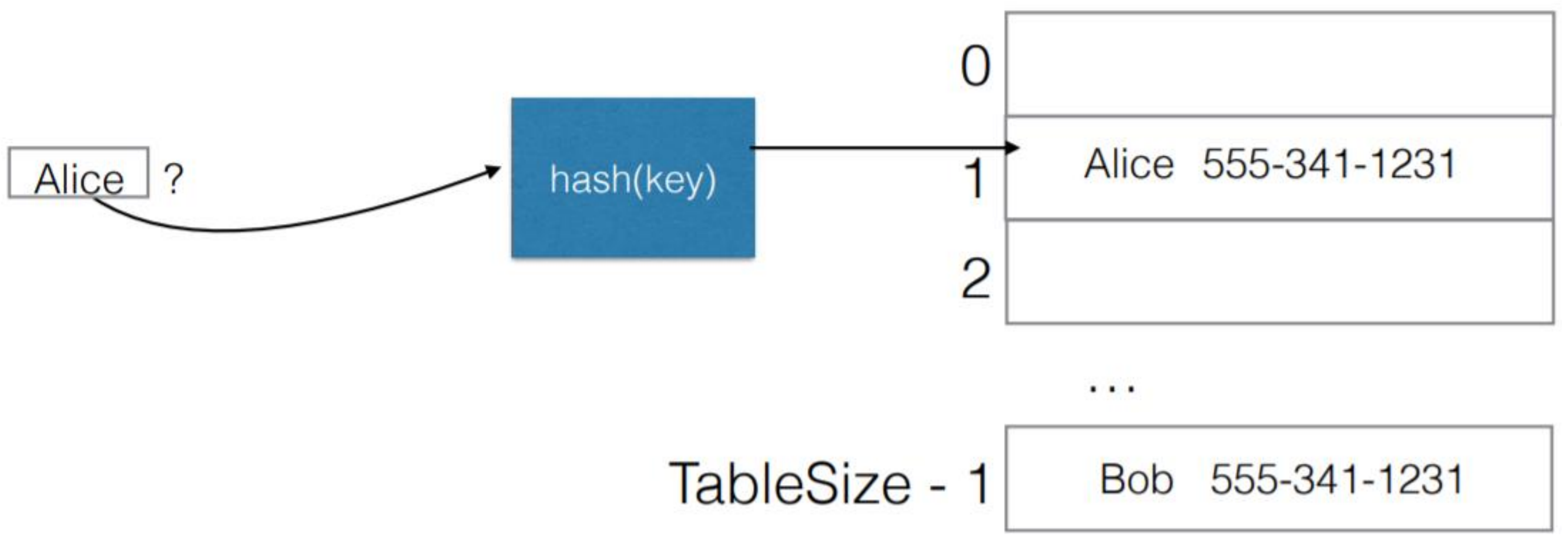
- Define a table (array) of length *TableSize*
- Define a function `hash(key)` that maps keys to integer indices in the range *0…TableSize-1*




Bob
555-987-2314
hash(key)
0
1
Alice 555-341-1231
2
...
TableSize - 1
Bob 555-341-1231

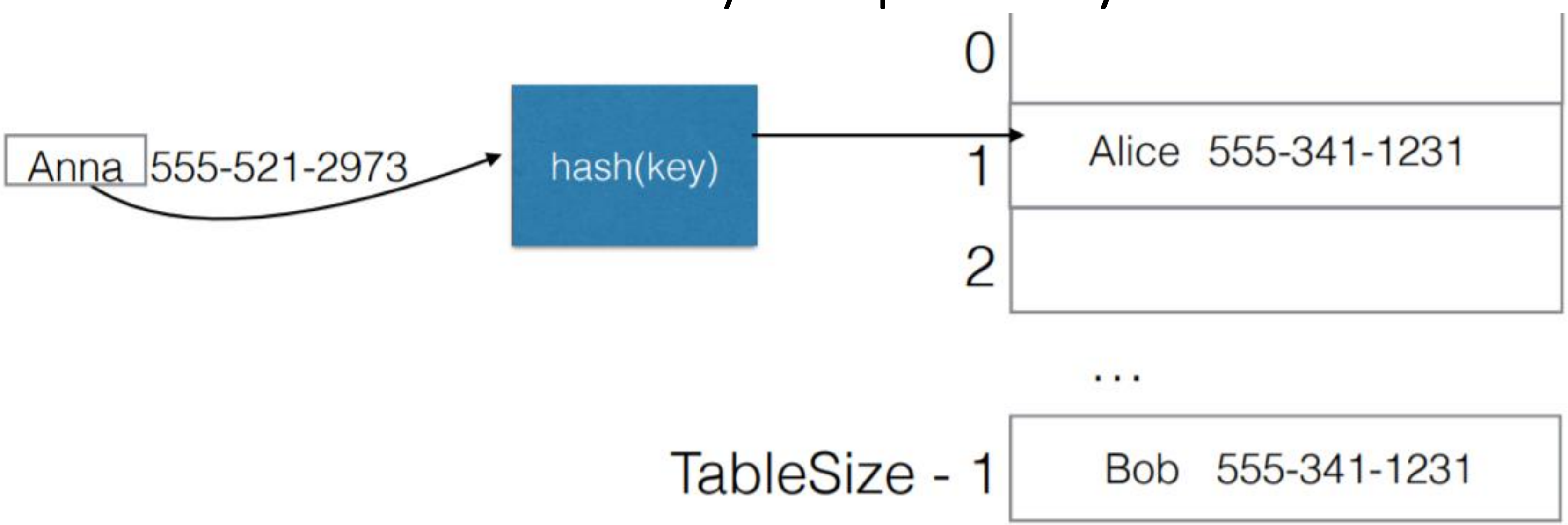
Hash Tables

- Assuming `hash(key)` takes constant time, `get` and `put` run in $O(1)$



Hash Tables Collisions

- Problem: there can be an infinite number of keys, but only *TableSize* entries in the array
  - Want to use a hash function that distributes items in the array evenly
  - Need to deal with *collisions* – situations where a new item hashes to an already-occupied array cell



Choosing a Hash Function

- Need to
  - Spread out the keys as much as possible in the table
    - If the function maps a lot of the keys to the same area of the table, there will be collisions
  - Make use of all table cells
    - Otherwise we are wasting space

Choosing a Hash Function

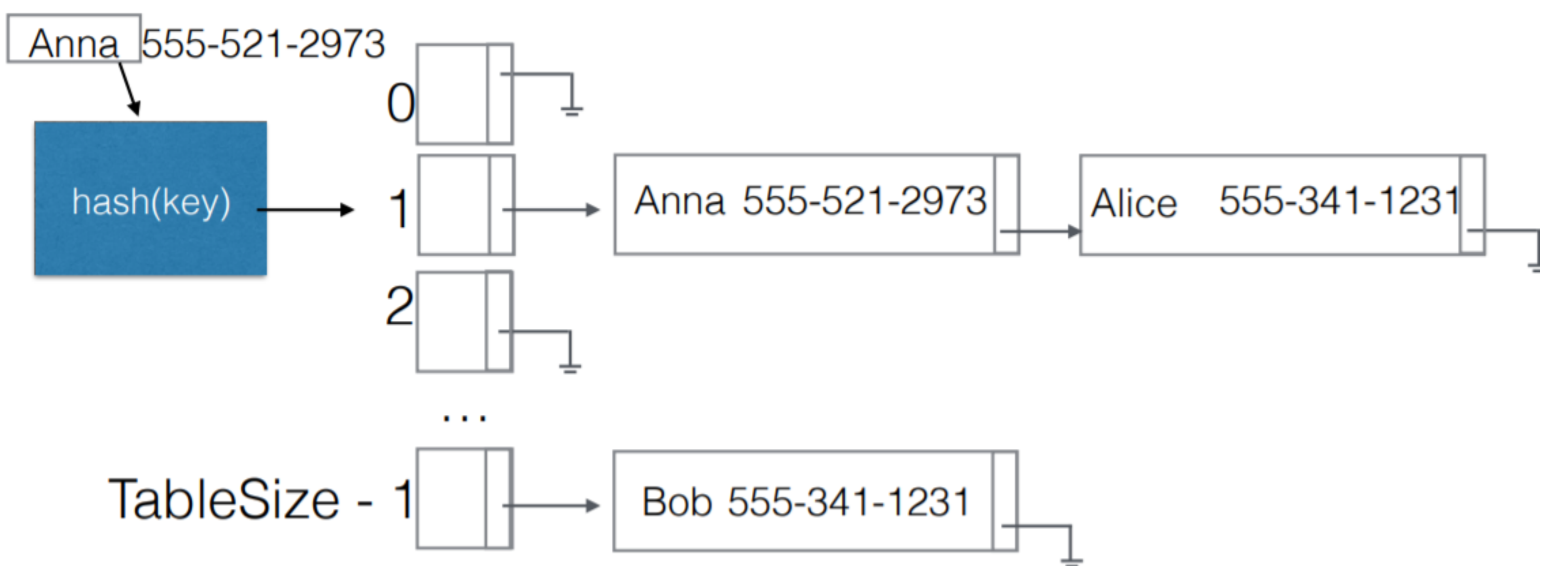
- If we can assume that the possible integer keys *x* are distributed evenly, we can use `hash(x) = x % TableSize`
- For strings, can use the ASCII value of the characters
  - E.g. $(str[n-1] \times 37^{n-1} + str[n-2] \times 37^{n-1} + \cdots + str[0])$ % TableSize

Choosing a Hash Function

- For a compound object (e.g., a string and an integer), can combine hash functions using
  $$hash(s,x) = (hash_1(s) \times p_1 + hash_2(x) \times p_2)\%TableSize$$
- p_1, p_2 prime
  - Don't want to be able to factor the expression
  - $hash(s,x) = 2 \times (hash_1(s) \times p_1 + hash_2(x) \times p_2)\%TableSize$ means we are skipping the odd cells in the table

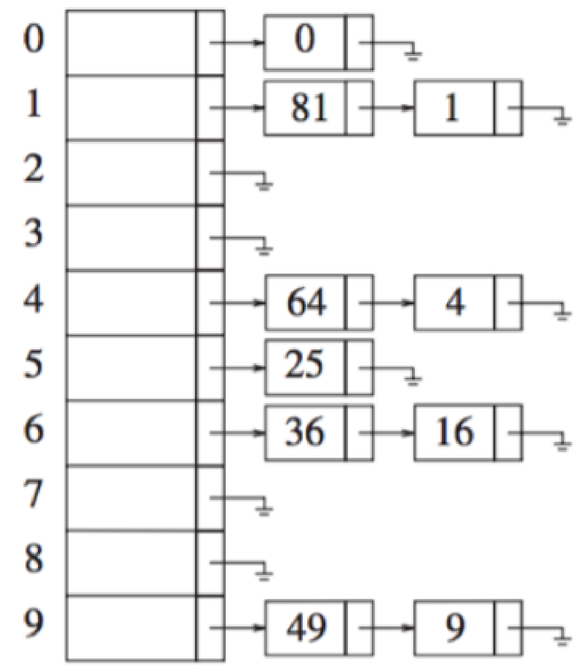
Collisions: Separate Chaining

- 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



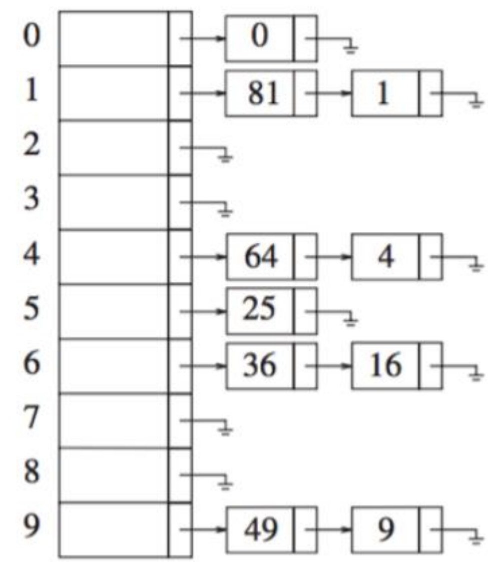
Runtime for Separate Chaining

- Runtime depends on the number of elements in a list on average
- Load factor $\lambda = N/TableSize$
- N is the total number of entries



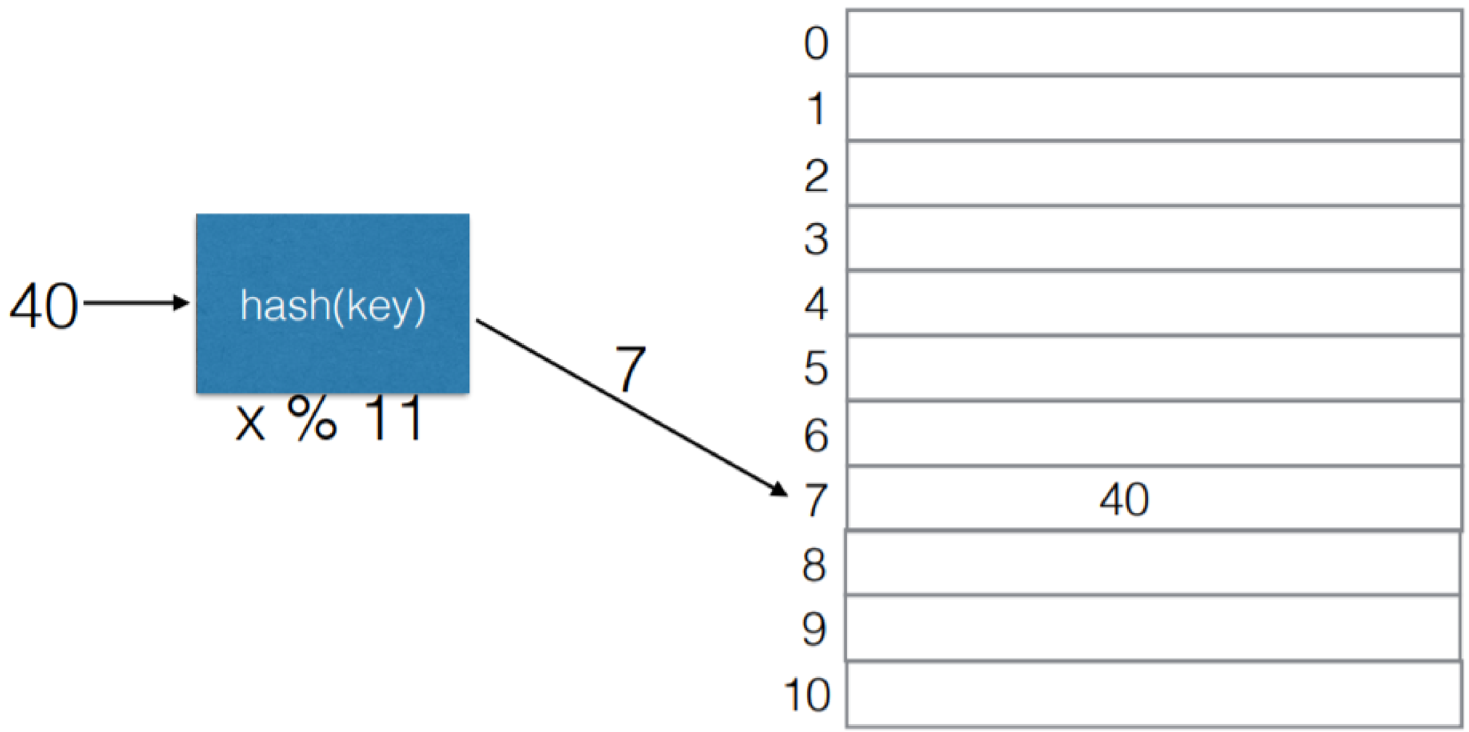
Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed.

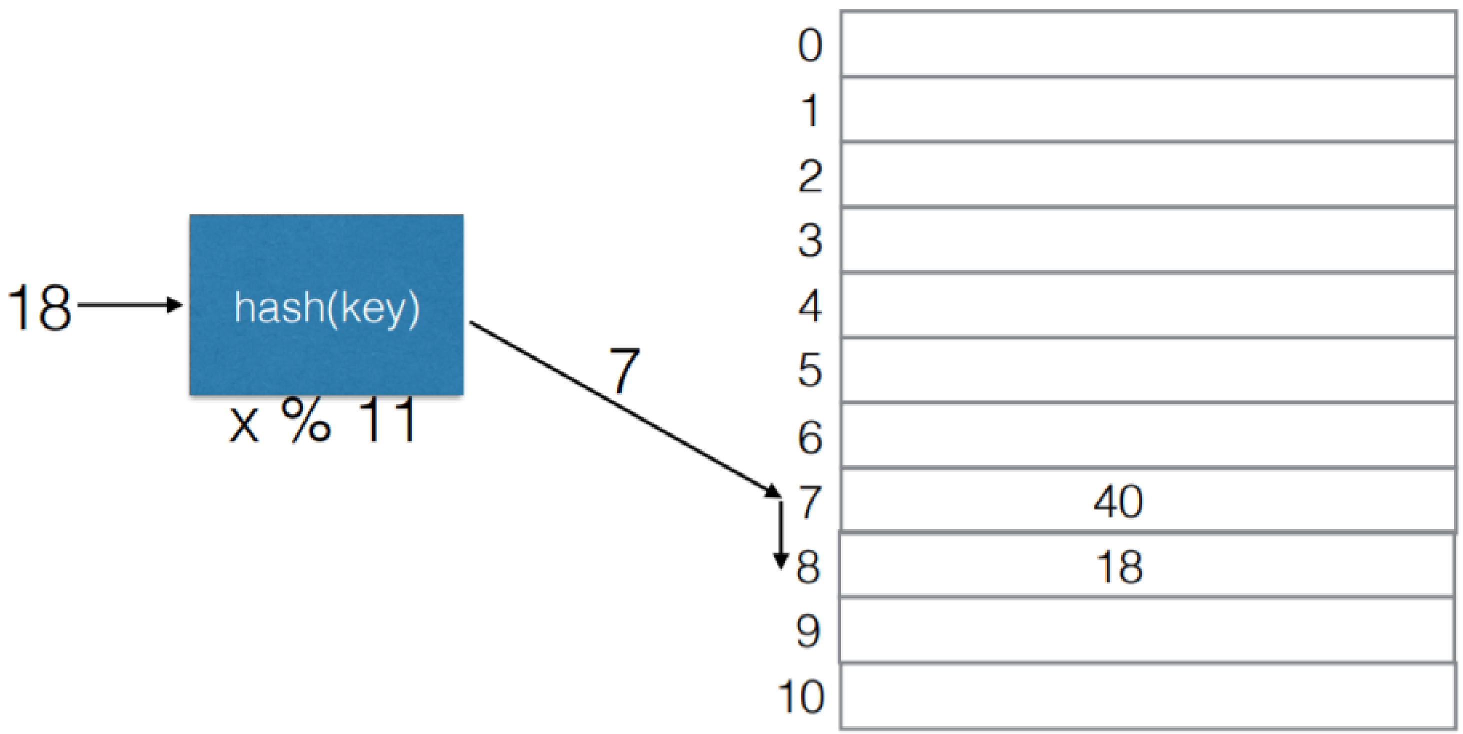
- If lookup fails (key is not in the hash table), need to search all (on average) λ nodes in the list for the hash bucket
- If lookup succeeds (key is in the table), need to search (on average) $\lambda/2$ +1 nodes
- Keep $\lambda \approx 1$
  - Increase table size as needed

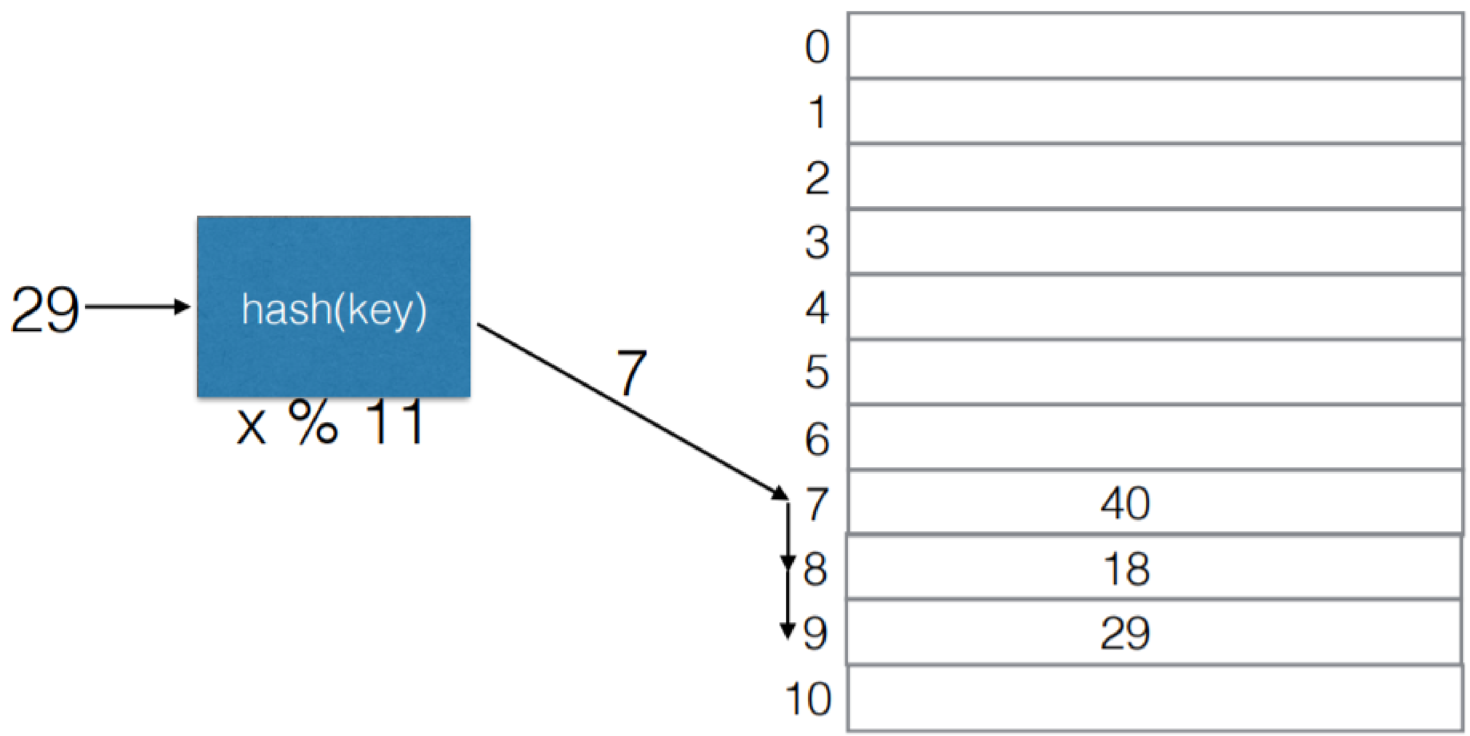


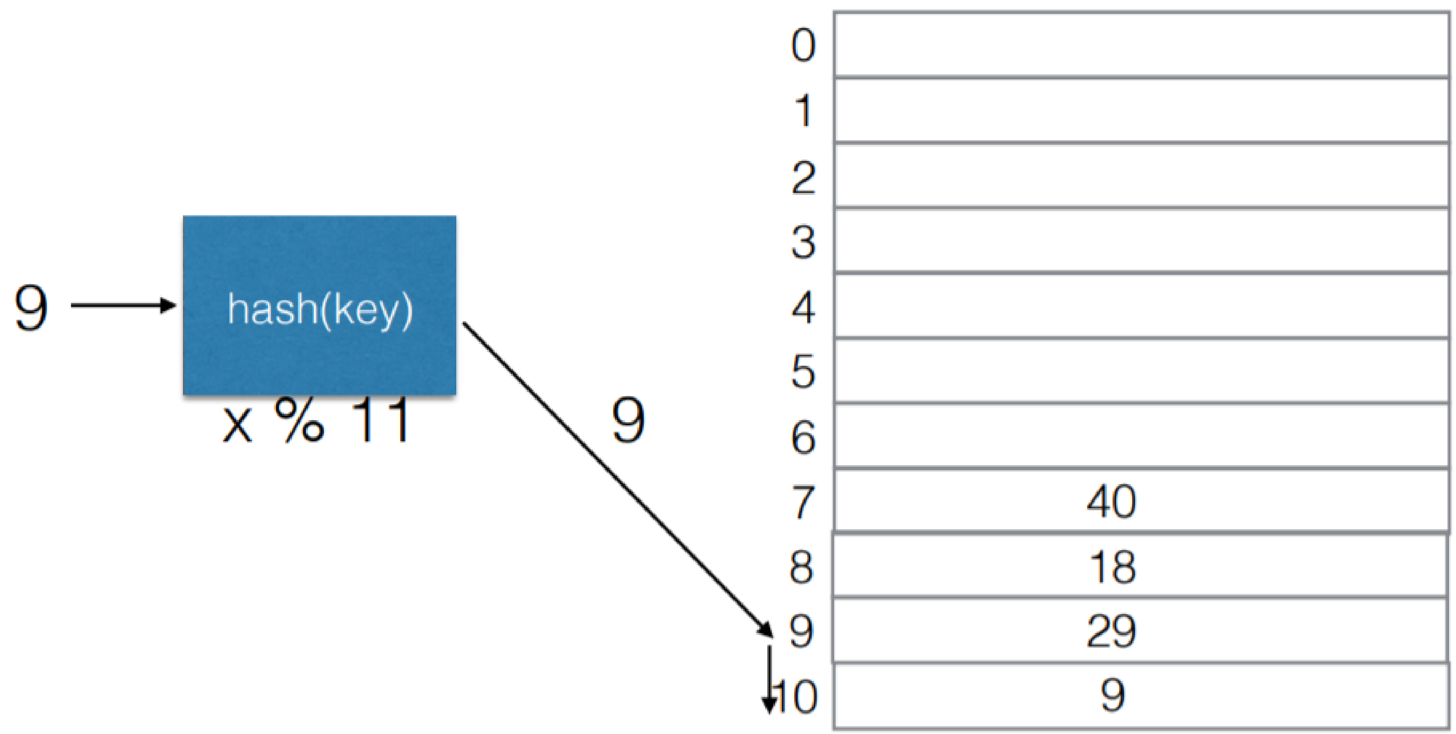
Probing

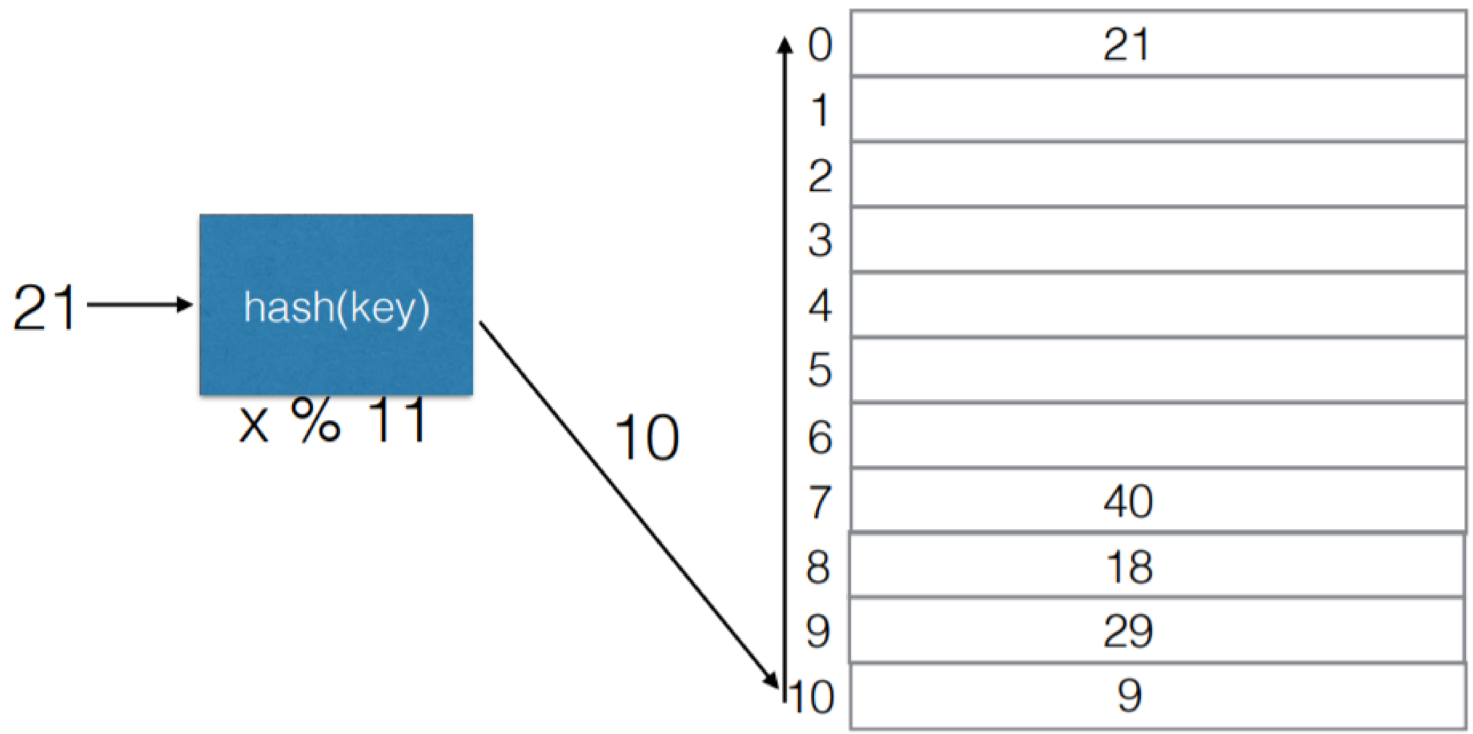
- When a collision occurs, put item in an empty cell of the had table




18
hash(key)
x % 11
7
0
1
2
3
4
5
6
7
40
8
18
9
10

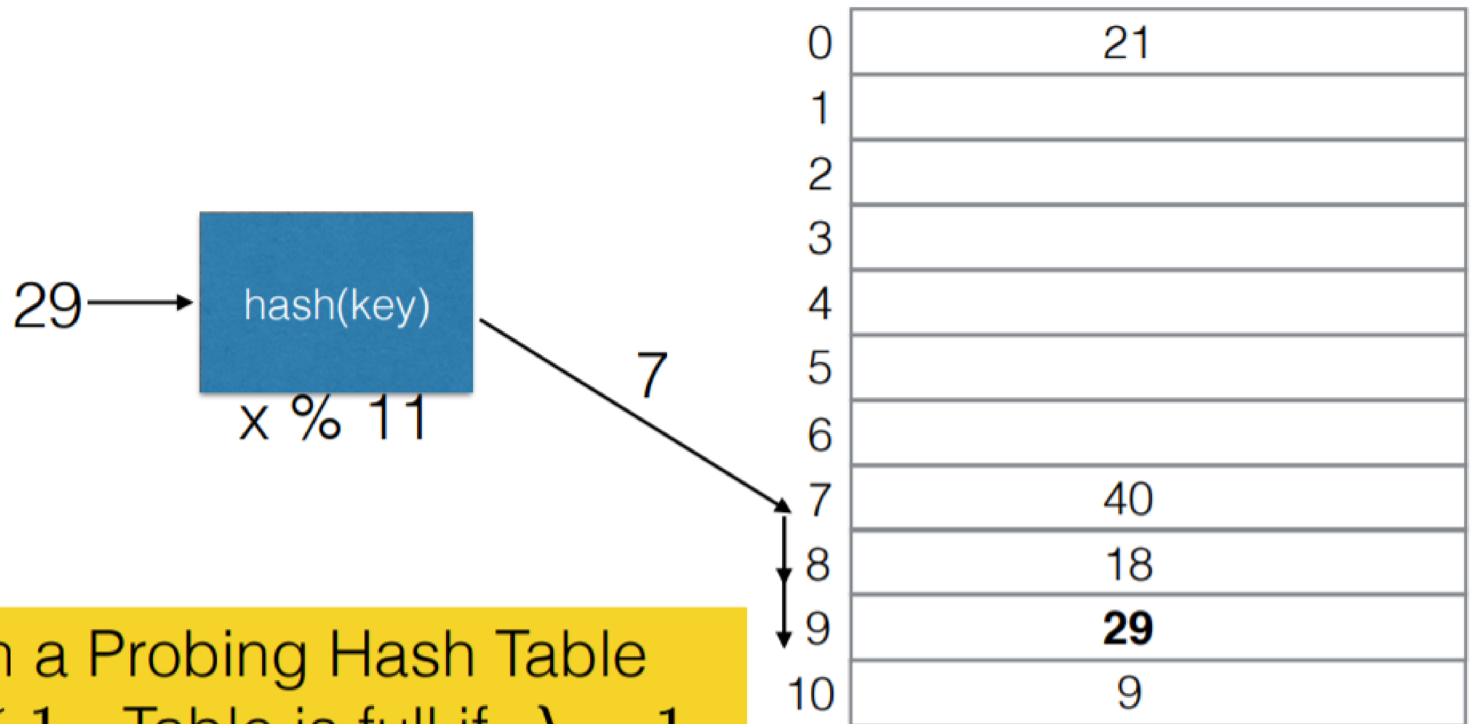

29
hash(key)
x % 11
7
0
1
2
3
4
5
6
7
8
9
10
40
18
29


9
hash(key)
x % 11
9
0
1
2
3
4
5
6
7
8
9
10
40
18
29
9



21
hash(key)
x % 11
10
0
1
2
3
4
5
6
7
8
9
10
21
40
18
29
9

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



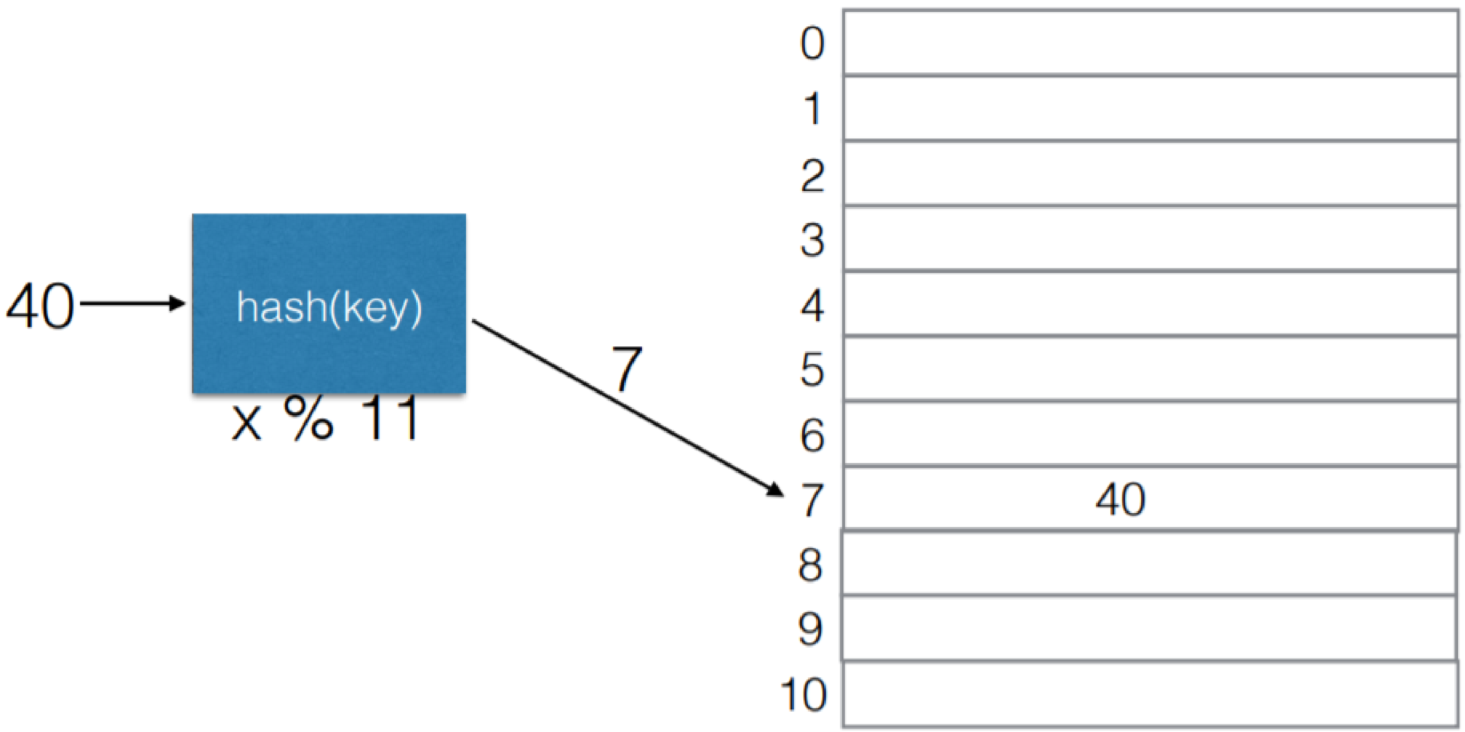
With a Probing Hash Table
$\lambda \leq 1$. Table is full if $\lambda = 1$.

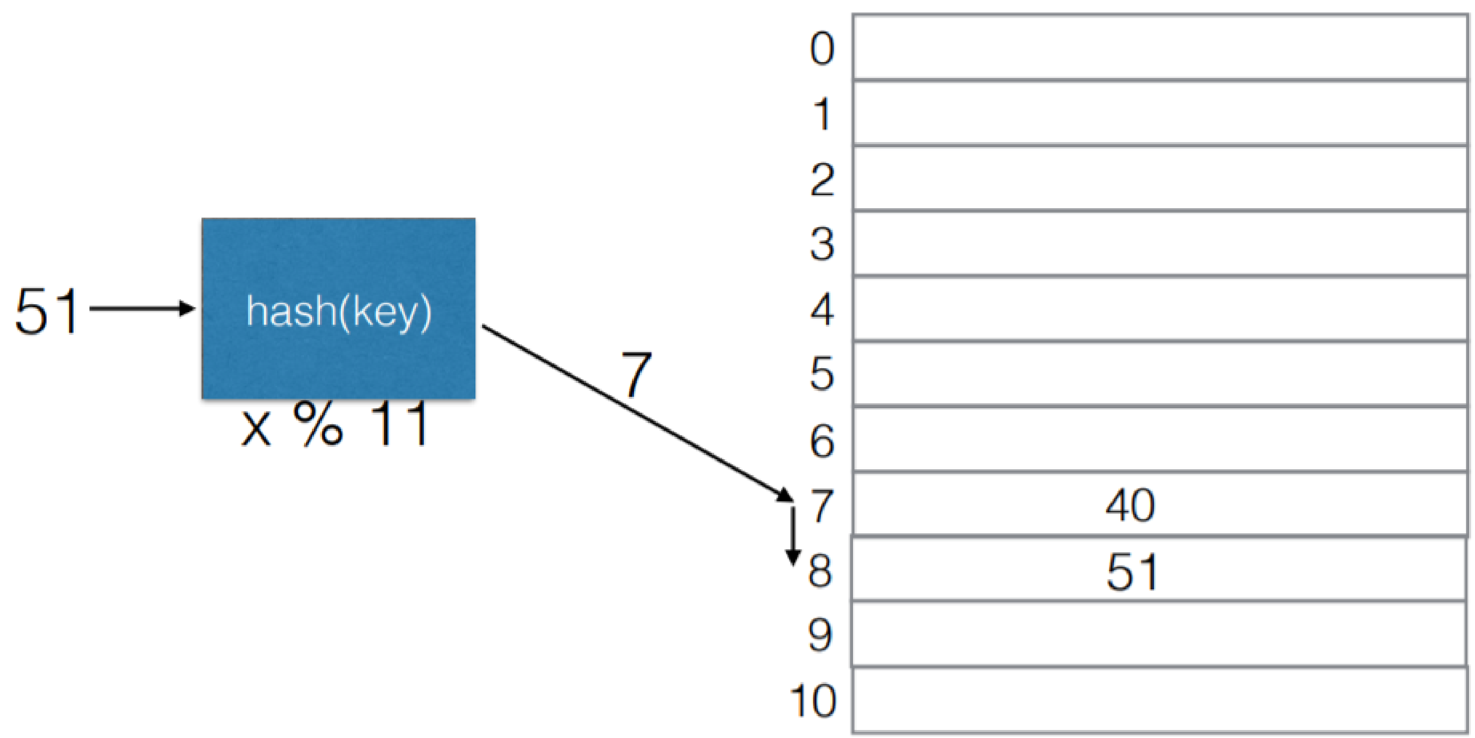
Probing: Collision Resolution Strategies (1)

- To insert an item, we probe other table cells systematically until an empty cell is found
- To look up a key, we probe until the key is found
- Different strategies to determine the next cell
  - In general, try $(hash(x) + f(i))\%TableSize$ after cell i
  - Linear probing: try the next cell, $f(i) = i$
  - Quadratic probing: $f(i) = i^2$
  - Double hashing: $f(i) = i \times hash_2(x)$

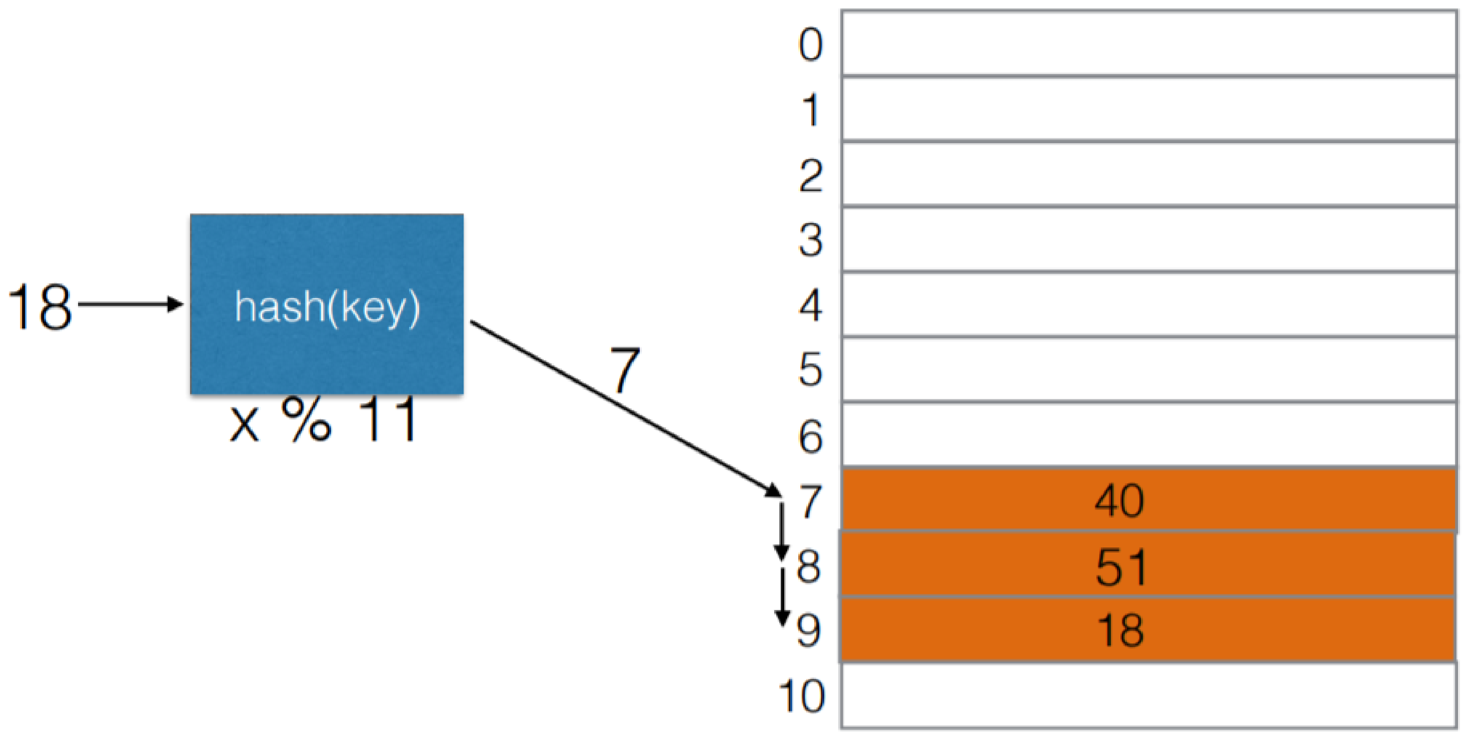
Linear probing

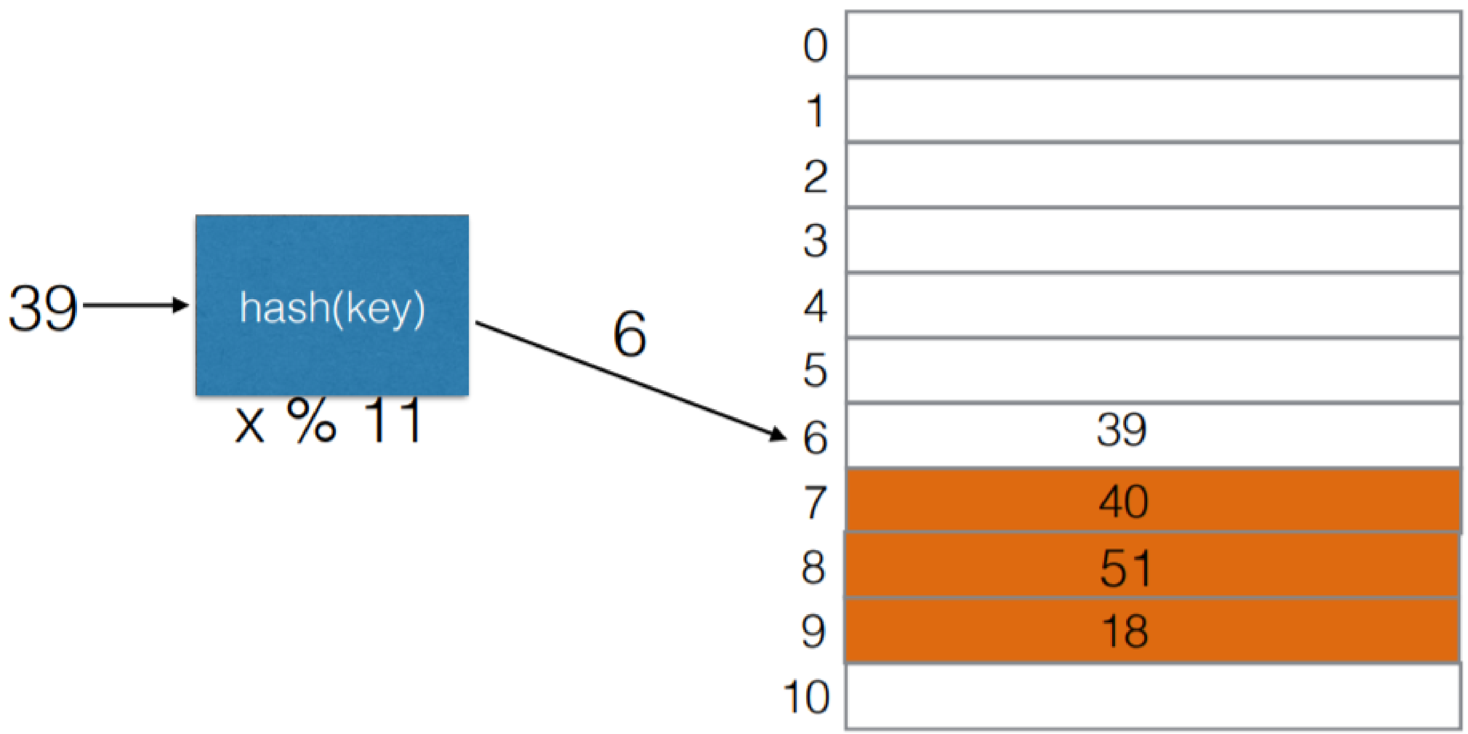
- If cell is occupied, try the next cell
- Problem: primary clustering
  - Full cells tend to cluster, with no free cells in between
  - Time required to find an empty cell can become very large if the table is almost full (λ is close to 1)

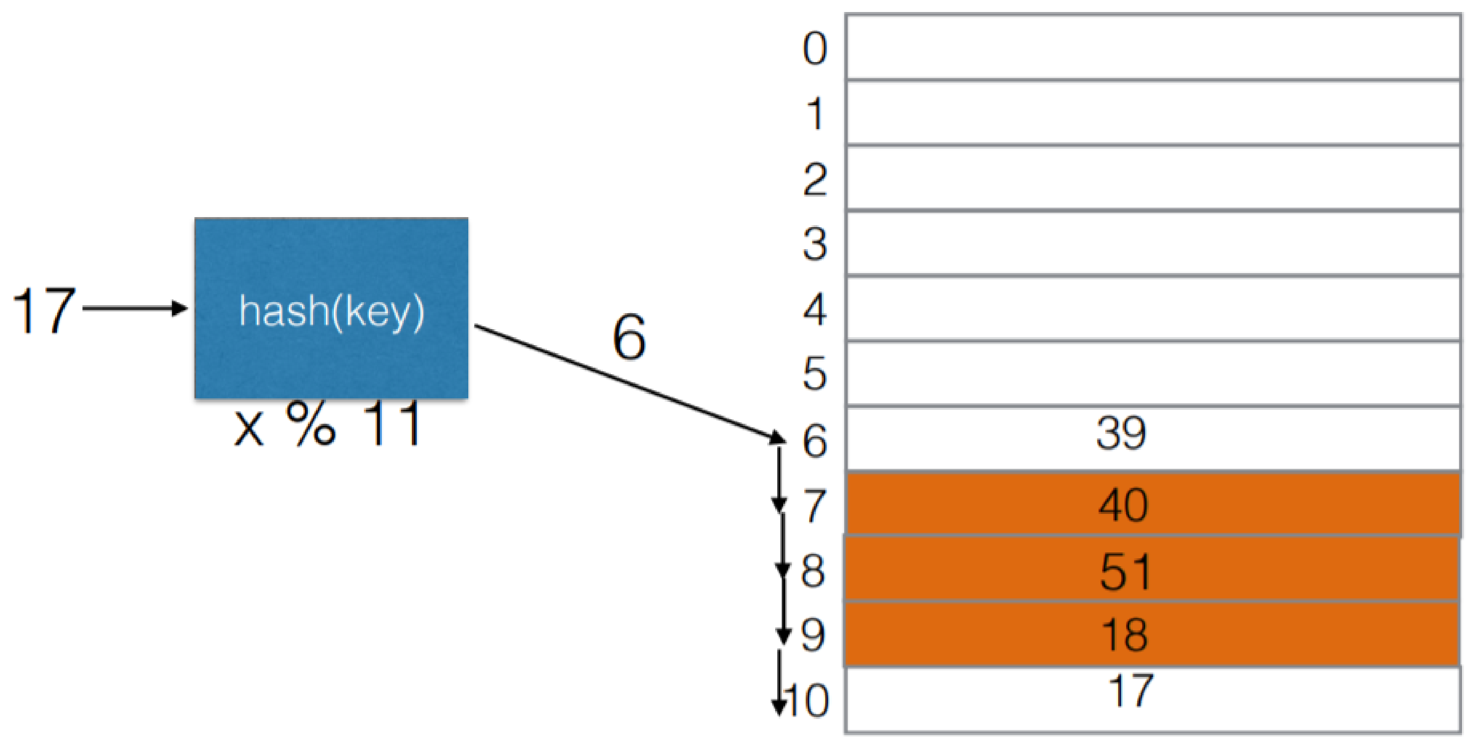

40
hash(key)
x % 11
7
0
1
2
3
4
5
6
7
8
9
10
40


51
hash(key)
x % 11
7
0
1
2
3
4
5
6
7
40
8
51
9
10

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

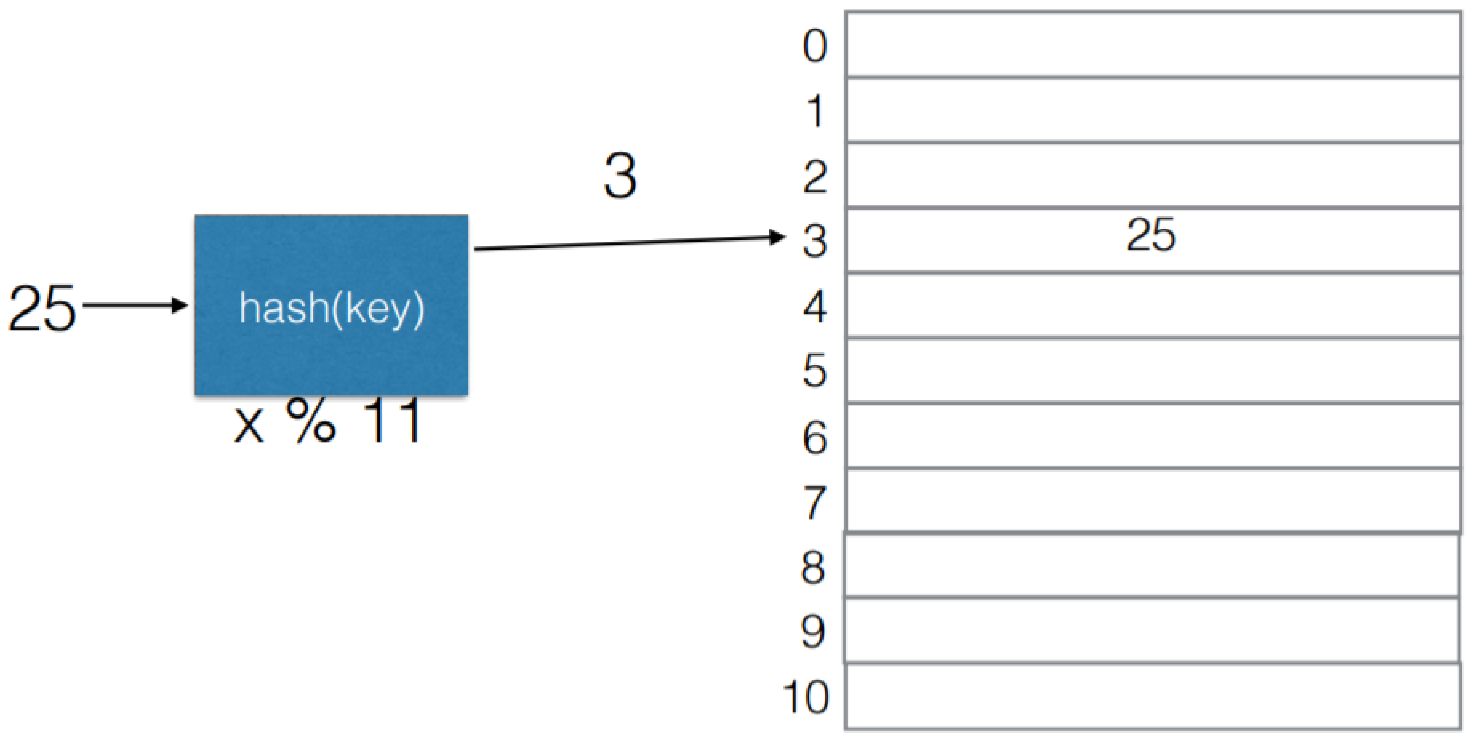


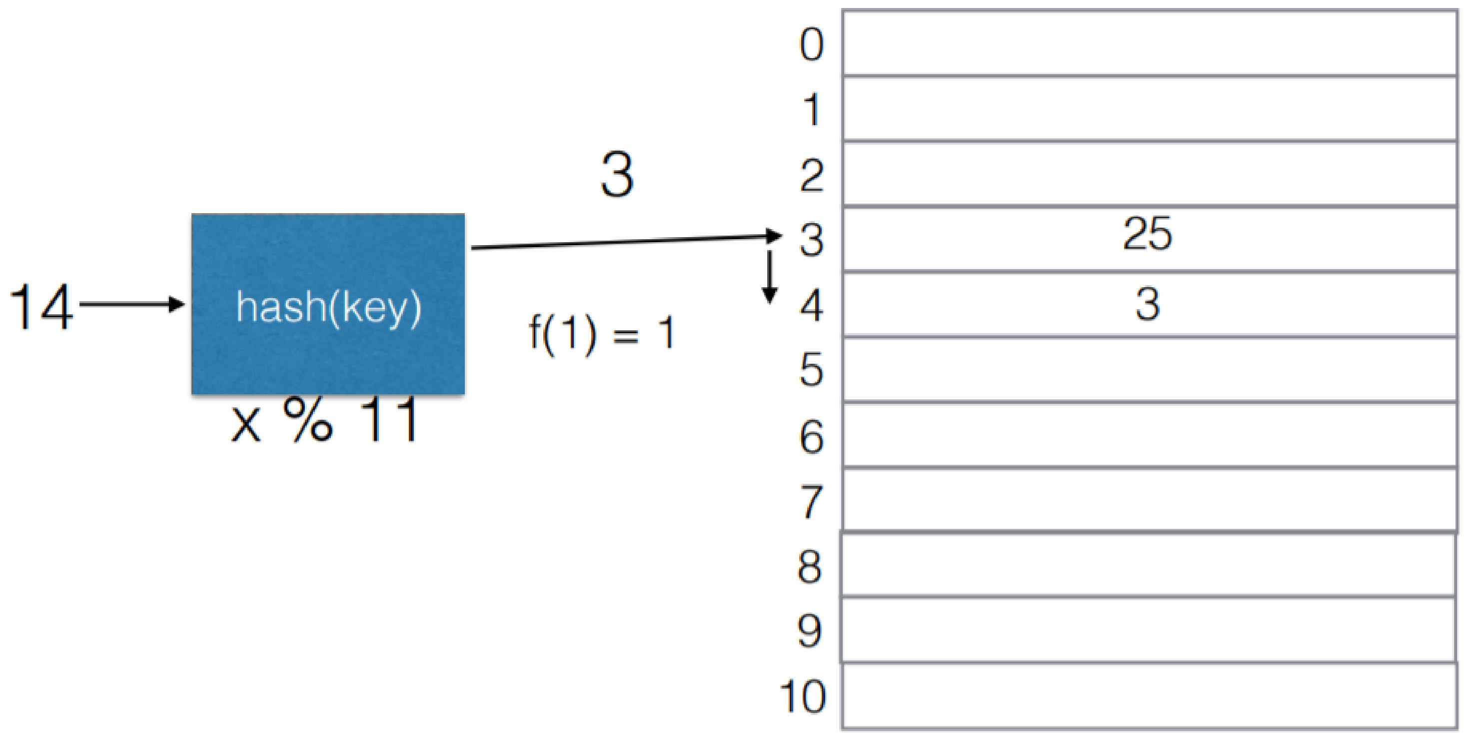

39
hash(key)
x % 11
6
0
1
2
3
4
5
6
39
7
40
8
51
9
18
10

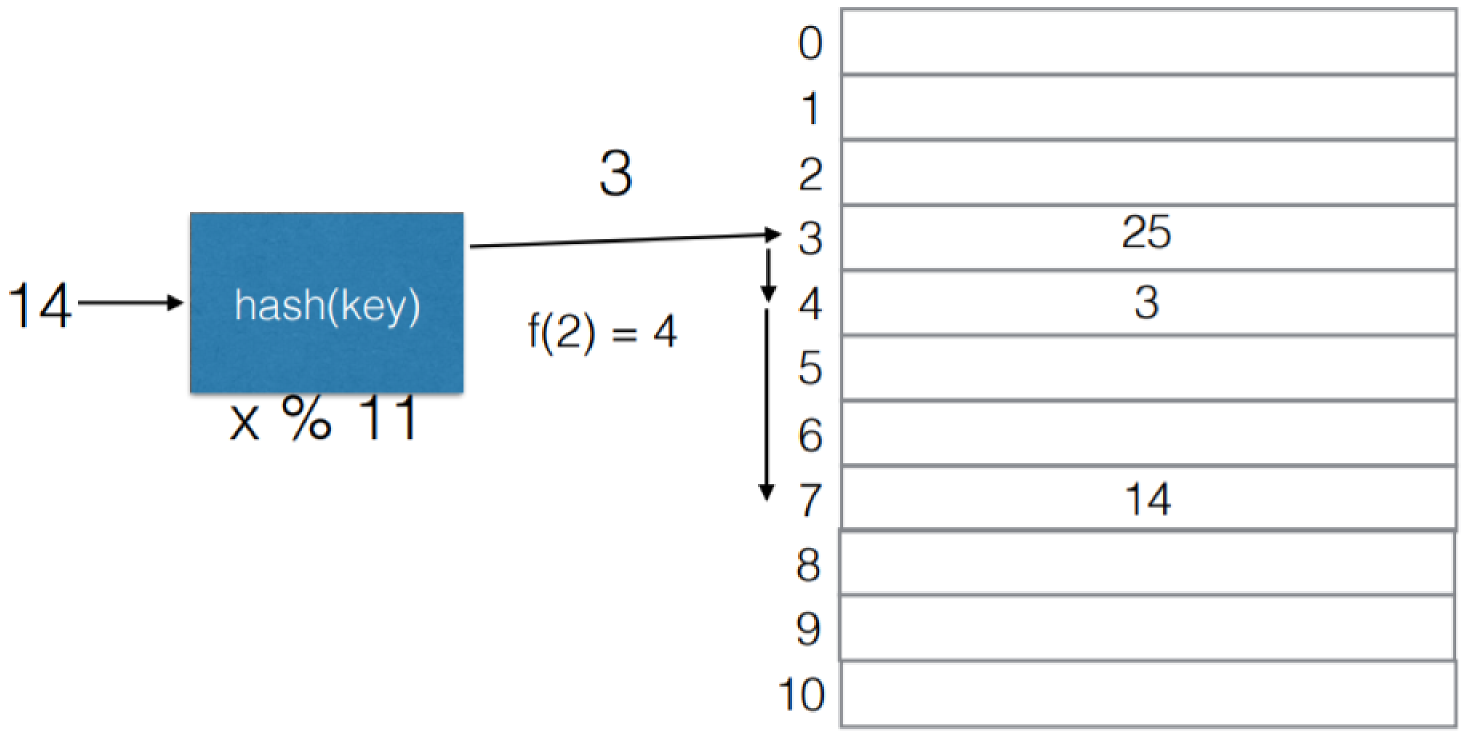

17
hash(key)
x % 11
6
0
1
2
3
4
5
6
7
8
9
10
39
40
51
18
17

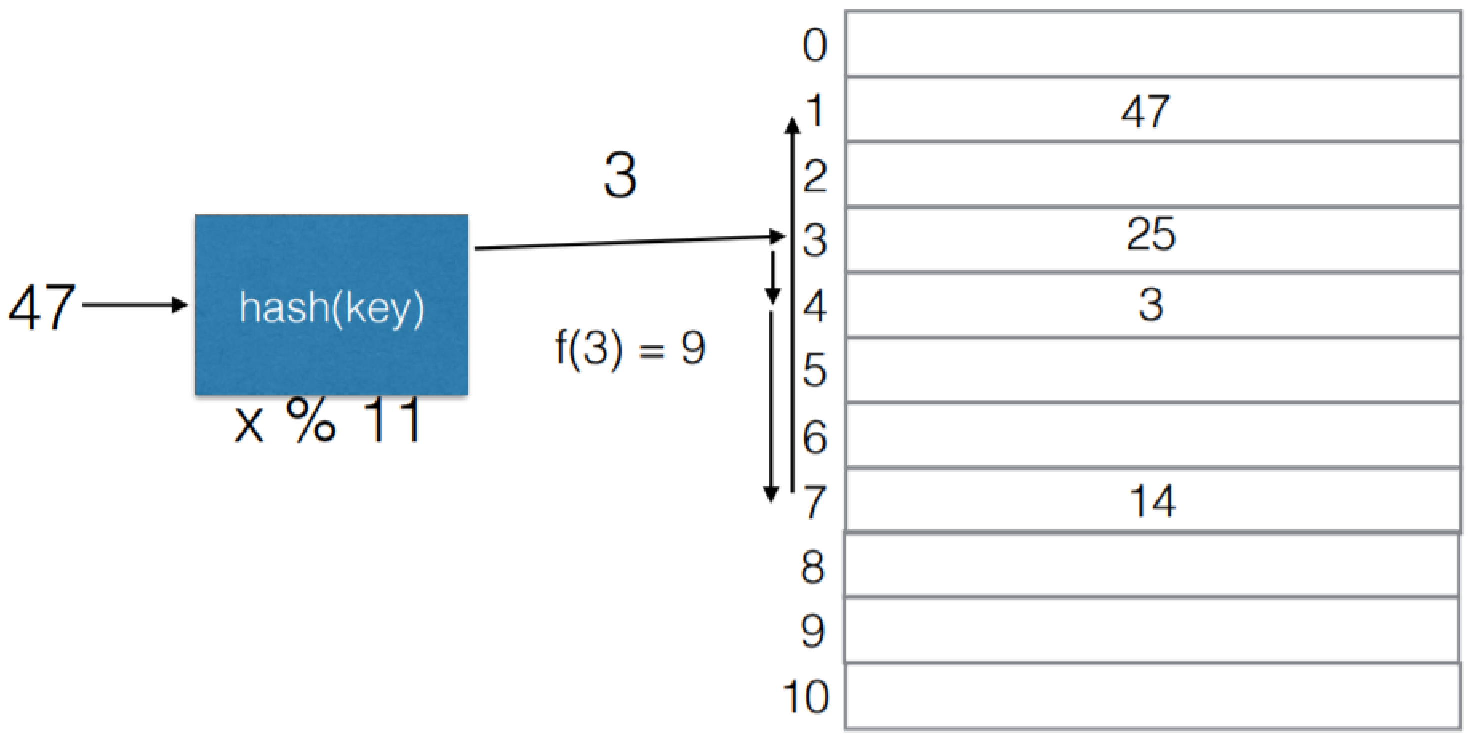
Quadratic Problem

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



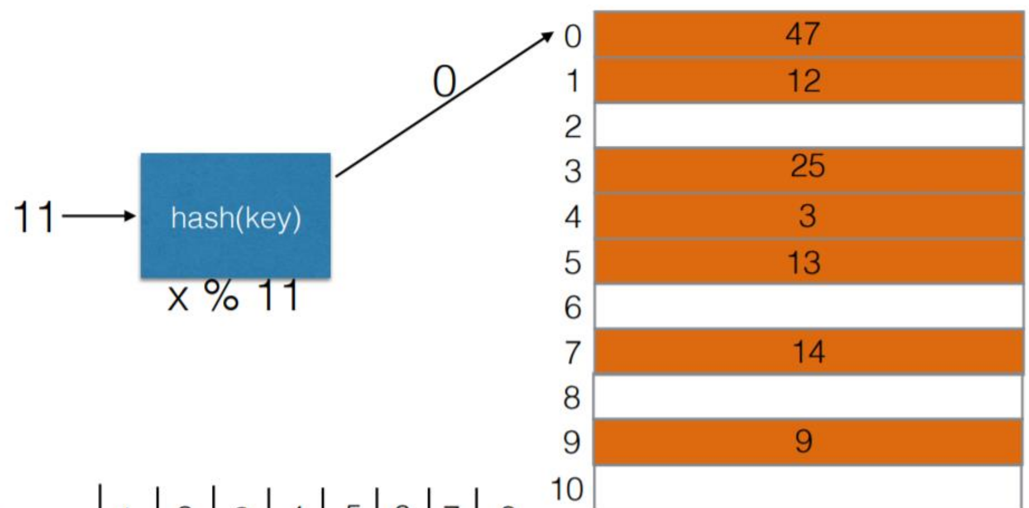

14
hash(key)
x % 11
3
f(1) = 1
0
1
2
3
4
5
6
7
8
9
10
25
3


14
hash(key)
x % 11
3
f(2) = 4
0
1
2
3
25
4
3
5
6
7
14
8
9
10


47
hash(key)
x % 11
3
f(3) = 9
0
1
2
3
4
5
6
7
8
9
10
47
25
3
14

Quadratic Probing

- Primary clustering is not a problem
- If the table gets too full ($\lambda > 0.5$), it is possible that empty cells become unreachable



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 Problem Theorem

- If *TableSize* is prime, then the first *TableSize/2* cells visited by quadratic probing are distinct
  - So it's always possible to find an empty cell if the table is at most half-full
- Proof sketch
  Suppose there is a repetition. Then
  $(h + i^2)\%T = (h + j^2)\%T$, so
  $h + i^2 = h + j^2 + kT$ for some integer *k*, so
  $kT = (i + j)(i - j)$
  T is prime, so either (i+j) or (i-j) is divisible by T. But i < j< M, which means that both (i-j) or (i+j) are too small