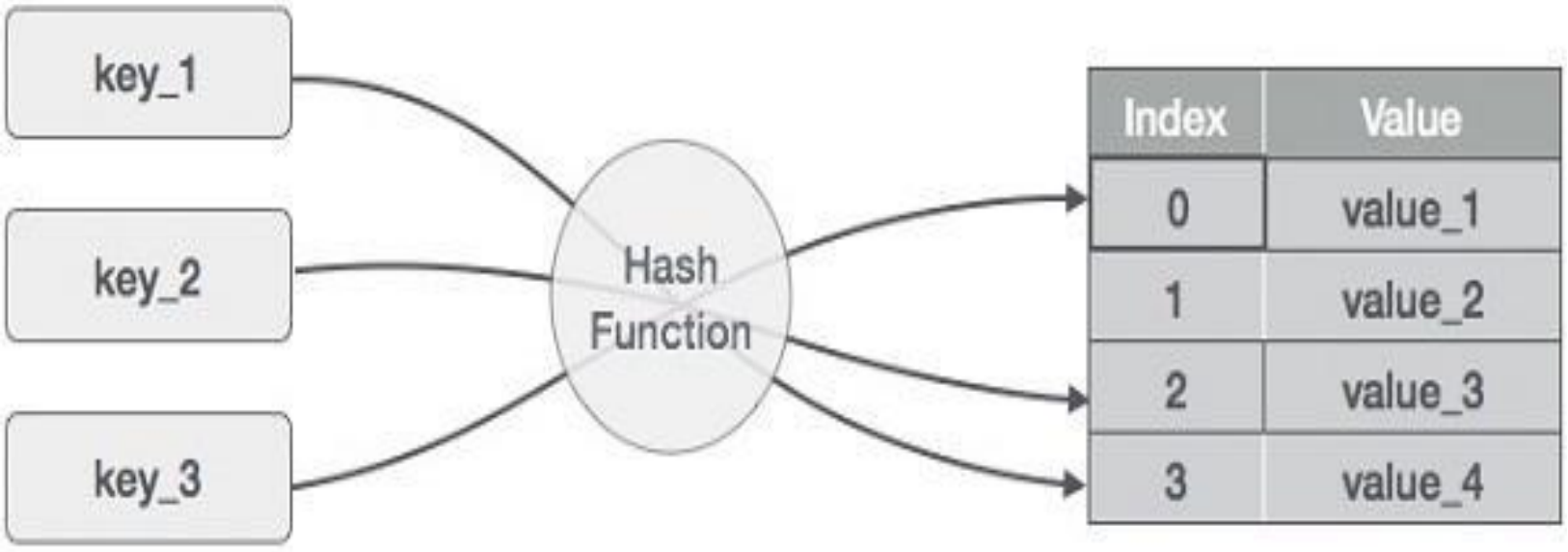
# Hashing:

- It is a technique to convert a range of key values into a range of indexes of an array.
- Hashing is a technique to convert a range of key values into a range of indexes of an array.



# Hashing:

- **Dictionaries :**
- Dictionaries stores elements so that they can be located quickly using **keys.**
- **Dictionary** = data structure that supports mainly two basic operations: insert a new item and return an item with a given key
- **For eg :** A Dictionary may hold bank accounts. In which key will be account number. And each account may stores many additional information.

# How to Implement a Dictionary?

- Different data structure to realize a key
  - Array , Linked list
  - Binary tree
  - **Hash table**
  - Red/Black tree
  - AVL Tree

# *Why Hashing?*

- The sequential search algorithm takes time proportional to the data size, i.e, **O(n)**.
- Binary search improves on liner search reducing the search time to **O(log n)**.
- With a BST, an **O(log n)** search efficiency can be obtained; but the worst-case complexity is **O(n)**.
- To guarantee the **O(log n)** search time, BST height balancing is required ( i.e., AVL trees).

# Why Hashing?

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
- A linked list implementation would take **O(n)** time.
- A height balanced tree would give **O(log n)** access time.
- Using an array of size 10,000 would give **O(1)** access time but will lead to a lot of space wastage.
- Is there some way that we could get **O(1)** access without wasting a lot of space?
- The answer is **hashing.**

# Hashing :

- Another important and widely useful technique for implementing dictionaries.
- Constant time per operation (on the average) Like an array, come up with a function to map the large range into one which we can manage.

# Applications:

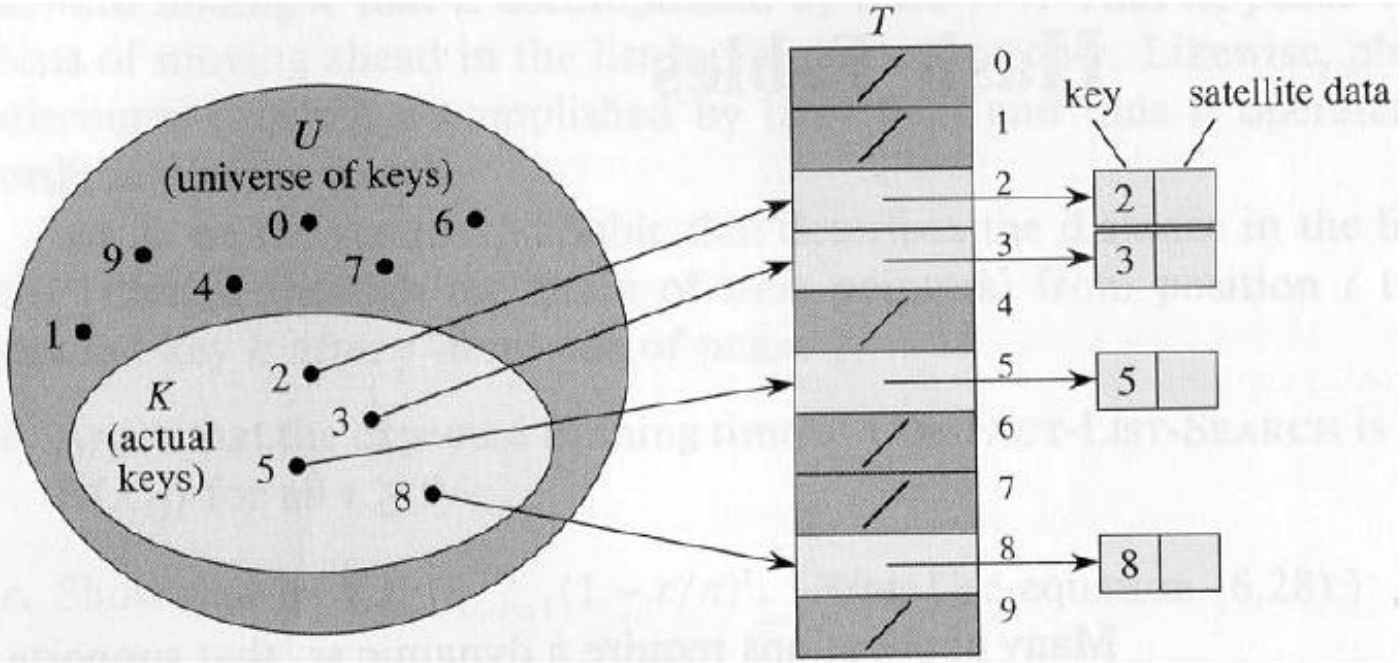
- **Keeping track of customer account information at a bank**
  - Search through records to check balances and perform transactions
- **Keep track of reservations on flights**
  - Search to find empty seats, cancel/modify reservations
- **Search engine**
  - Looks for all documents containing a given word

# Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, m - 1\}$
- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

# Direct Addressing (cont'd)




(insert/delete in O(1) time)

# Examples Using Direct Addressing

## Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array $A$ of 100 items and store the record whose key is equal to $i$ in $A[i]$

## Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

# Hash Tables

- When $K$ is much smaller than $U$, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the <u>average</u> case, not the worst case

# Hash Tables

**Idea:**

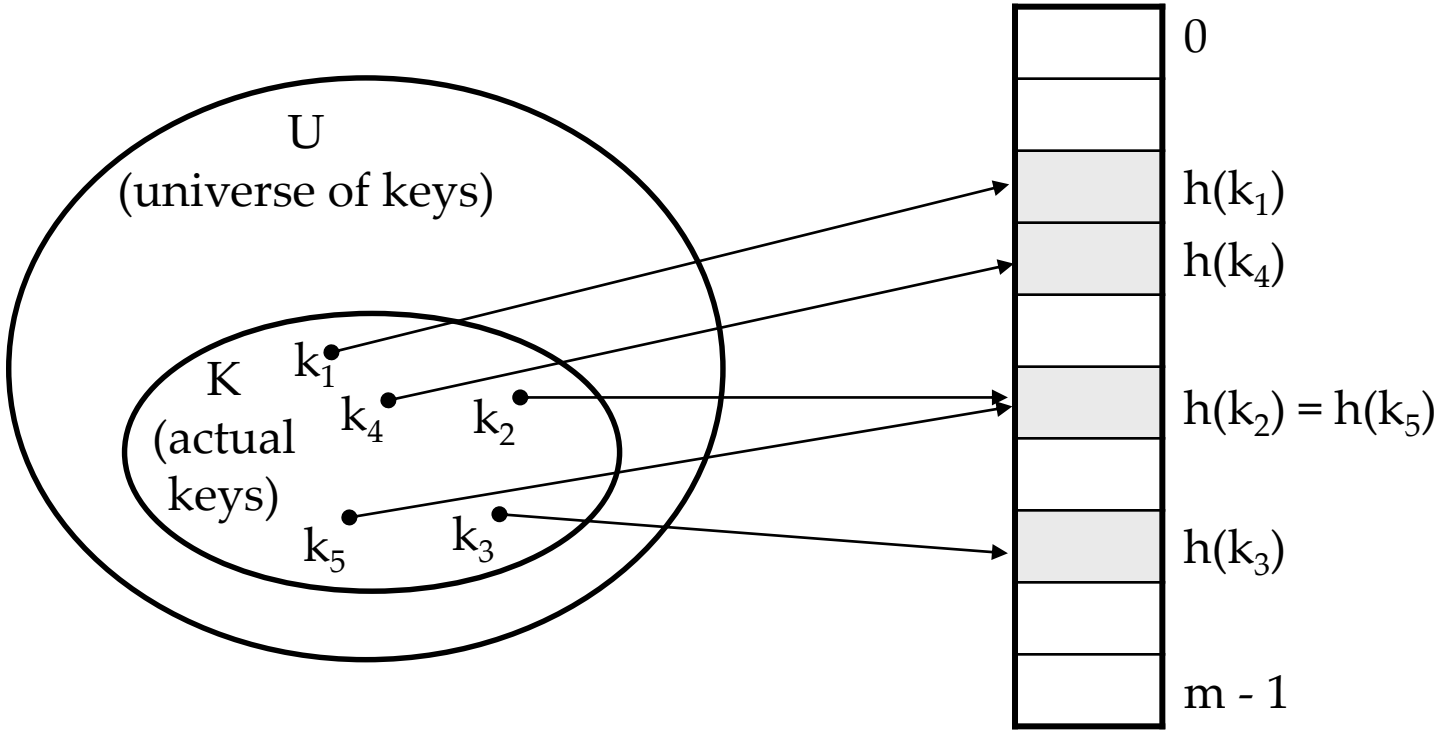
- Use a function $h$ to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** $h$ transforms a key into an index in a hash table $T[0...m-1]$:

$$h : U \rightarrow \{0, 1, \ldots, m - 1\}$$

- We say that $k$ **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: $m$ instead of $|U|$
  - Storage is also reduced

# Example: HASH TABLES



# Revisit Example 2

Suppose that the keys are nine-digit social security numbers

Possible hash function

$h(ssn) = sss \bmod 100$ (last 2 digits of ssn)

e.g., if $ssn = 10123411$ then $h(10123411) = 11$)

# *Hash Functions:*

- **Division Method:**
  - H(k) = k (mod m)  or H(k) = k (mod m) + 1
- **Midsquare Method:**
  - H(k) = l
- **Folding Method:**
  - H(k) = k1 + k2 + … +Kr

# The Division Method :

- **Idea:**
  - Map a key **k** into one of the **m** slots by taking the remainder of **k** divided by **m**

$$h(k) = k \bmod m$$

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of **m** are bad, e.g.,
    - power of 2
    - non-prime numbers

# Example - The Division Method

- If $m = 2^p$, then $h(k)$ is just the least significant $p$ bits of $k$
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$ , least significant 1 bit of $k$
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$ least significant 2 bits of $k$
- Choose $m$ to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

| | m 97 | m 100 |
|---|---|---|
| 16838 | 57 | 38 |
| 5758 | 35 | 58 |
| 10113 | 25 | 13 |
| 17515 | 55 | 15 |
| 31051 | 11 | 51 |
| 5627 | 1 | 27 |
| 23010 | 21 | 10 |
| 7419 | 47 | 19 |
| 16212 | 13 | 12 |
| 4086 | 12 | 86 |
| 2749 | 33 | 49 |
| 12767 | 60 | 67 |
| 9084 | 63 | 84 |
| 12060 | 32 | 60 |
| 32225 | 21 | 25 |
| 17543 | 83 | 43 |
| 25089 | 63 | 89 |
| 21183 | 37 | 83 |
| 25137 | 14 | 37 |
| 25566 | 55 | 66 |
| 26966 | 0 | 66 |
| 4978 | 31 | 78 |
| 20495 | 28 | 95 |
| 10311 | 29 | 11 |
| 11367 | 18 | 67 |

# Example

- Calculate hash value of keys 1234, 6236,9123,2326 and 4355 where table size m = 10

# 2.Folding method

- Algorithm: H(x) = (a + b + c) mod m

  Where a, b, and c represent the preconditioned key broken down into three parts, m is the table size, and mod stands for modulo.

  In other words: the sum of three parts of the preconditioned key is divided by the table size. The remainder is the hash key.

# Examples

- Fold the key 123456789 into a hash table of ten spaces (0 through 9).
- We are given x = 123456789 and the table size (i.e., m = 10).

  Since we can break x into three parts any way we want to, we will break it up evenly.
  Thus a = 123, b = 456, and c = 789.

  **H(x) = (a + b + c) mod m**
  H(123456789) = (123+456+789) mod 10
  = 1368 mod 10
  = 8
- 123456789 is inserted into the table at address 8.

# 3.Multiplication method

- The method that is used in the slide is the multiplication method. It multiplies all the individual digits in the key together, and takes the remainder after dividing the resulting number by the table size. In functional notation, the algorithm is:

  H(x) = (a * b * c * d *....) mod m

  Where: m is the table size, a, b, c, d, etc. are the individual digits of the item, and mod stands for modulo.

  Let's apply this algorithm to an example.

# Example

- Given a hash table of ten buckets (0 through 9), what is the hash key x = 131130

  We are given the table size (i.e., m = 10).

$$H(x) = (a * b * c * d * \ldots.) \bmod m$$
$$H(131130) = (1 * 3 * 1 * 1 * 3 * 0) \bmod 10$$
$$= 0 \bmod 10$$
$$= 0$$

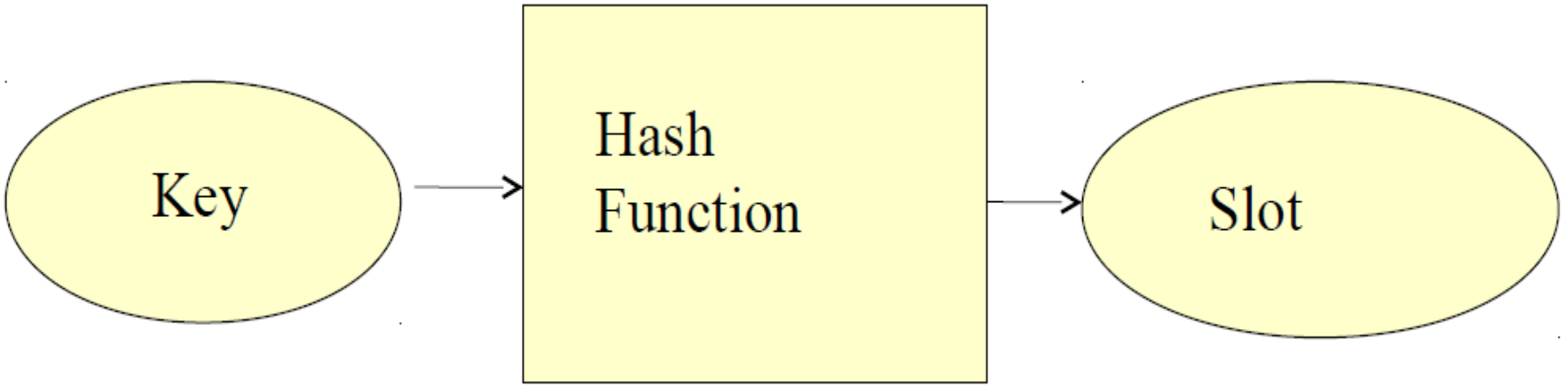
# Common Hashing Functions (cont'd)

### 4. Mid-Square

- The key is squared and the middle part of the result taken as the hash value.
- To map the key **3121** into a hash table of size **1000**, we square it $3121^2$ **= 9740641** and extract **406** as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

# Hash Functions:

- A Good Hash function is one which distribute keys evenly among the slots.
- And It is said that Hash Function is more art than a science. Because it need to analyze the data.



# *Hash Functions:*

- **Need of choose a good Hash function**
  - Quick Compute.
  - Distributes keys in uniform manner throughout the table.
- **How to deal with Hashing non integer Key???**
  - Find some way of turning keys into integer.
  - For Example: if key is in character then convert it into integer using ASCII
  - Then use standard Hash Function on the integer.

# Collisions

- Two or more keys hash to the same slot!!
- For a given set $K$ of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

# Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build "good" functions.

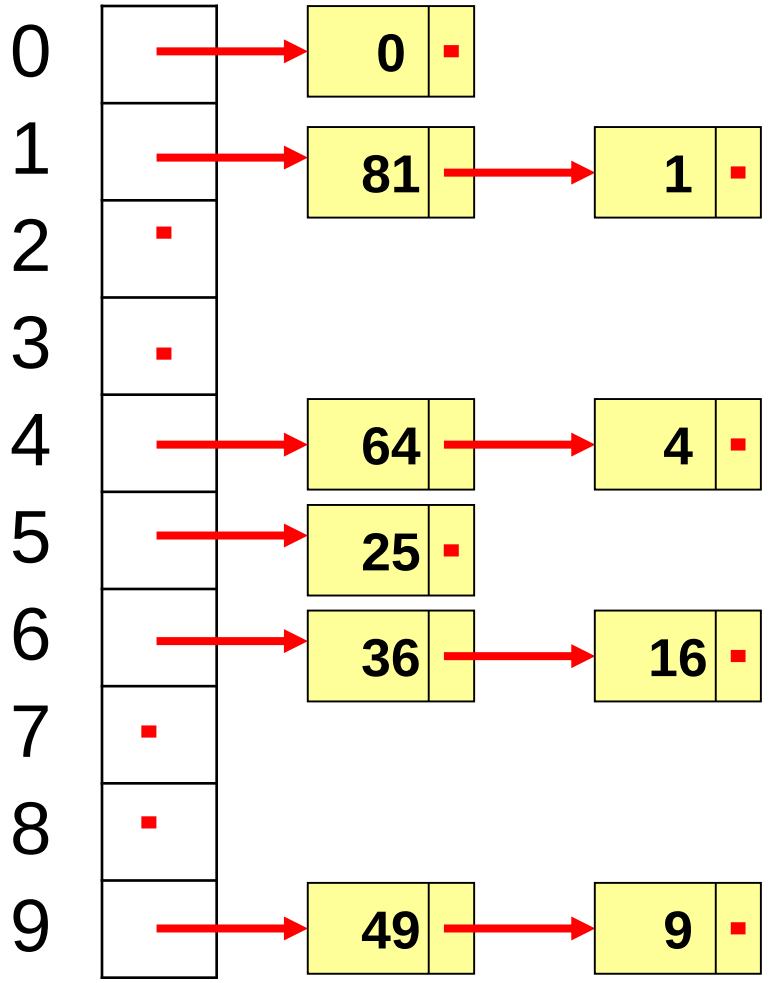
# Collision Resolution Schemes: Chaining

The hash table is an array of linked lists

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function $h(k) = k \bmod m$



# Linear Probing

Function f is linear. Typically, f(i) = i

So, h( k, i ) = ( h′(k) + i ) mod m

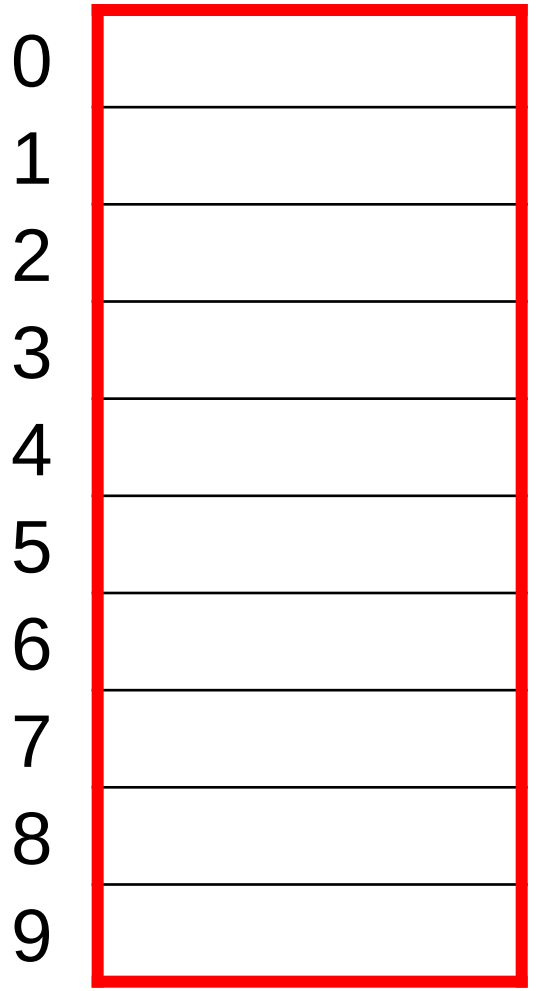
Offsets: 0, 1, 2, …, m-1

With H = h′( k ), we try the following cells with wraparound:

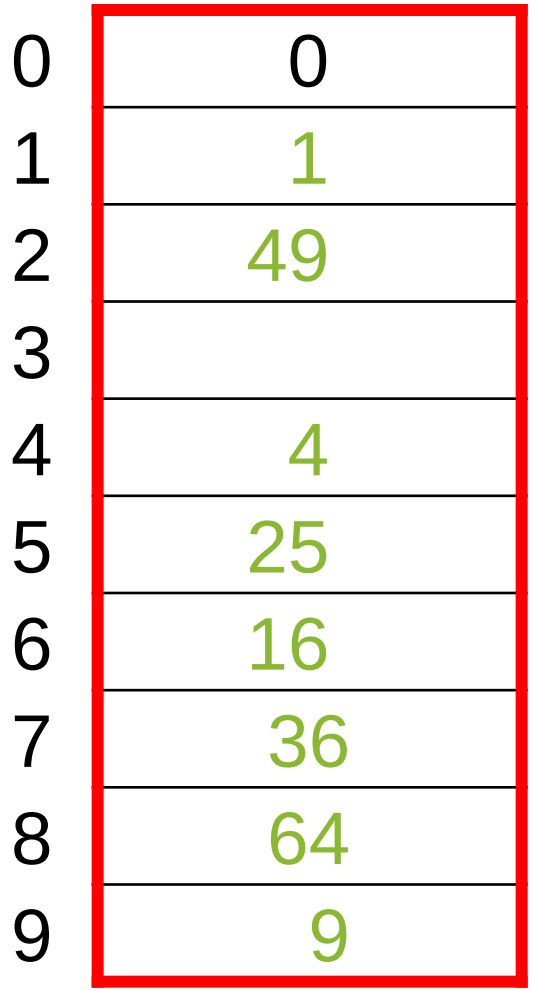
**H, H + 1, H + 2, H + 3, …**

What does the table look like after the following insertions?

**Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81**



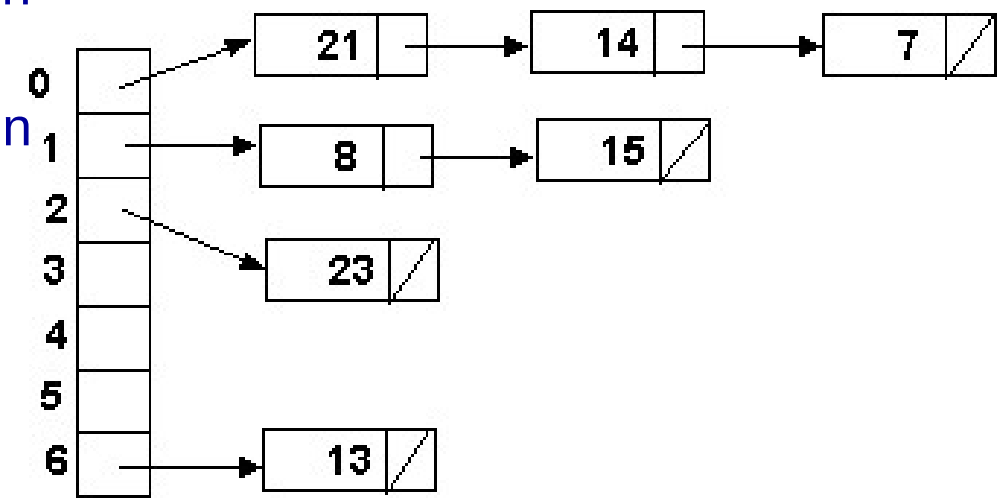
# Linear Probing :



# Separate Chaining (cont'd)

- Retrieval of an item, `r`, with hash address, `i`, is simply retrieval from the linked list at position `i`.
- Deletion of an item, `r`, with hash address, `i`, is simply deleting `r` from the linked list at position `i`.
- **Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2
h(13) = 13 % 7 = 6
h(21) = 21 % 7 = 0
h(14) = 14 % 7 = 0 collision
h(7) = 7 % 7 = 0 collision
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1 collision



# Separate Chaining with String Keys (cont'd)

- Use the hash function **hash** to load the following commodity items into a hash table of size **13** using separate chaining:

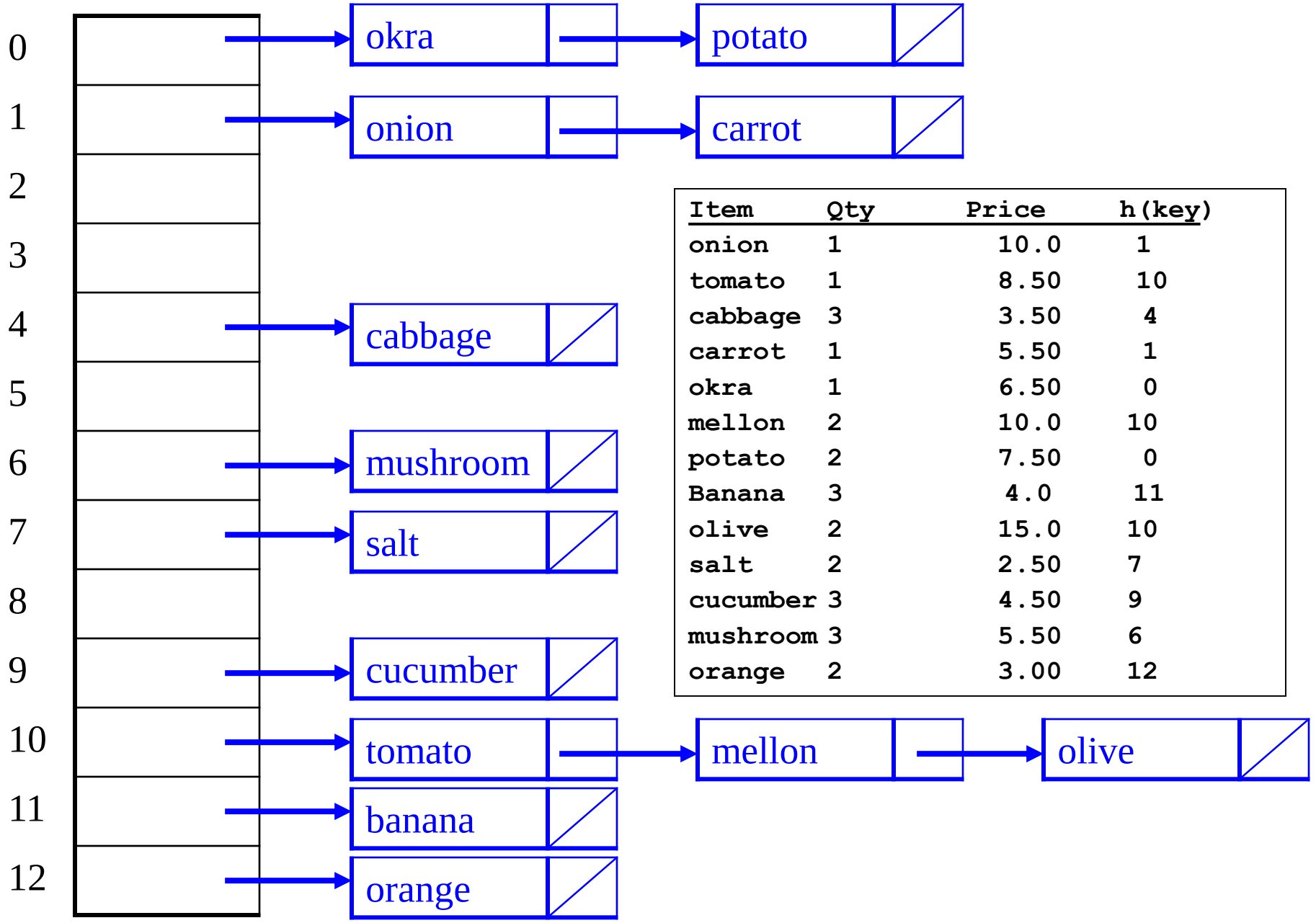
```
onion       1       10.0
tomato      1       8.50
cabbage     3       3.50
carrot      1       5.50
okra        1       6.50
mellon      2       10.0
potato      2       7.50
Banana      3        4.00
olive       2       15.0
salt        2       2.50
cucumber    3       4.50
mushroom    3       5.50
orange      2       3.00
```

- Solution:

| character | a | b | c | e | g | h | i | k | l | m | n | o | p | r | s | t | u | v |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII code | 97 | 98 | 99 | 101 | 103 | 104 | 105 | 107 | 108 | 109 | 110 | 111 | 112 | 114 | 115 | 116 | 117 | 118 |

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = 1
hash(salt) = (115 + 97 + 108 + 116) % 13 = 436 % 13 = 7
hash(orange) = (111 + 114 + 97 + 110 + 103 + 101)%13 = 636 %13 = 12

# Separate Chaining with String Keys (cont'd)

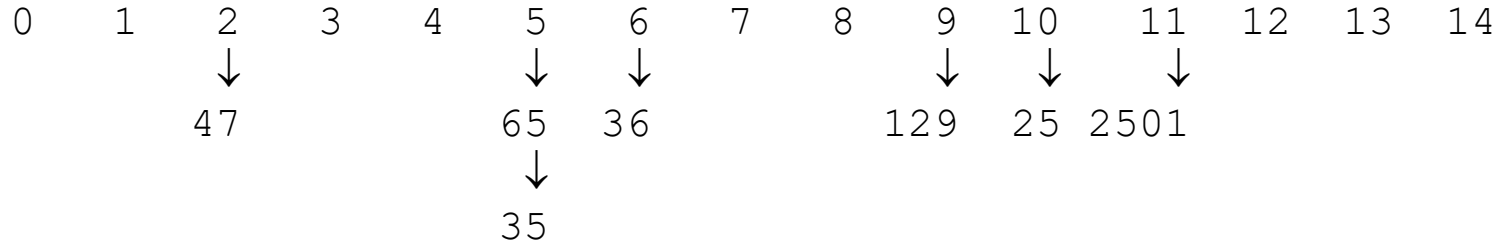



| Item | Qty | Price | h(key) |
|---|---|---|---|
| onion | 1 | 10.0 | 1 |
| tomato | 1 | 8.50 | 10 |
| cabbage | 3 | 3.50 | 4 |
| carrot | 1 | 5.50 | 1 |
| okra | 1 | 6.50 | 0 |
| mellon | 2 | 10.0 | 10 |
| potato | 2 | 7.50 | 0 |
| Banana | 3 | 4.0 | 11 |
| olive | 2 | 15.0 | 10 |
| salt | 2 | 2.50 | 7 |
| cucumber | 3 | 4.50 | 9 |
| mushroom | 3 | 5.50 | 6 |
| orange | 2 | 3.00 | 12 |

# Separate Chaining

Let each array element be the head of a chain.

Array:

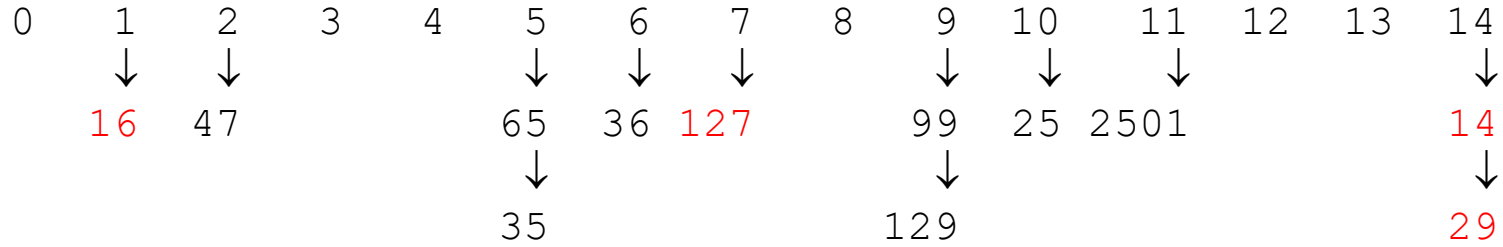



Where would you store: 29, 16, 14, 99, 127 ?

# Separate Chaining

Let each array element be the head of a chain:

Array:




Where would you store: 29, 16, 14, 99, 127 ?

Note that we use the *insertAtHead()* method when inserting new keys.

# Separate Chaining versus Open-addressing

**Separate Chaining has several advantages over open addressing:**

- Collision resolution is simple and efficient.
- The hash table can hold more elements without the large performance deterioration of open addressing.
- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

**Disadvantages of Separate Chaining:**

- It requires the implementation of a separate data structure for chains, and code to manage it.
- The main cost of chaining is the extra space required for the linked lists.
- For some languages, creating new nodes (for linked lists) is expensive and slows down the system.

# Linear Probing

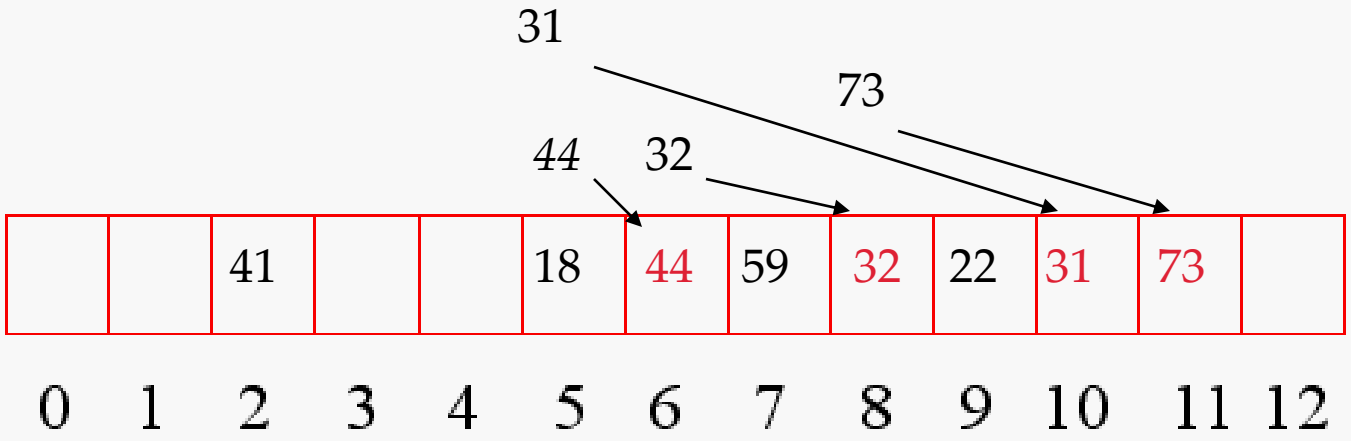
- If the current location is used, try the next table location

```
linear_probing_insert(K)
  if (table is full) error
  probe = h(K)
     while (table[probe] occupied)
    probe = (probe + 1) mod M
      table[probe] = K
```

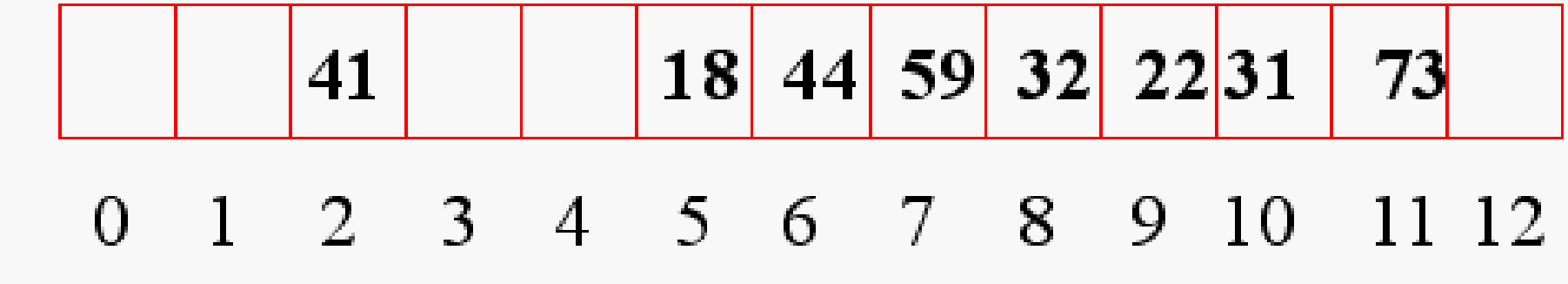
- Lookups walk along table until the key or an empty slot is found
- Uses less memory than chaining. (Don't have to store all those links)
- Slower than chaining. (May have to walk along table for a long way.)
- Deletion is more complex. (Either mark the deleted slot or fill in the slot by shifting some elements down.)

# Linear Probing Example

- h(k) = k mod 13
- Insert keys: 18 41 22 44 59 32 31 73



# Linear Probing Example (cont.)



# Linear Probing

Let key *x* be stored in element *f(x)=t* of the array

Array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 47 | | | 35 | 36 | | | 129 | 25 | 2501 | | | |
| | | | | | 65(?) | | | | | | | | | |

What do you do in case of a collision?

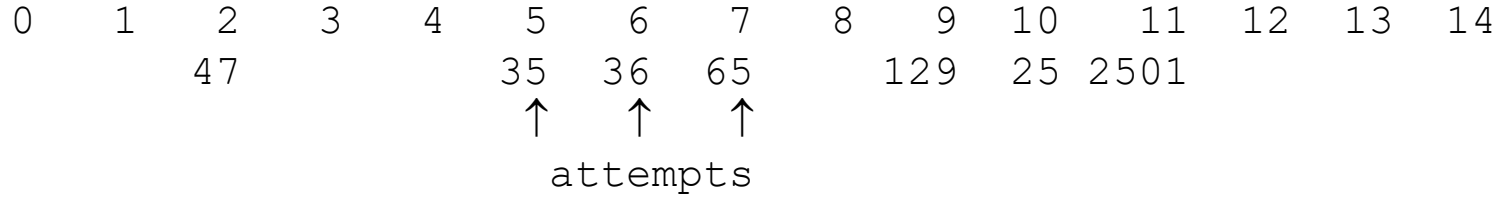
If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N*, (*t*+2)%*N*, (*t*+3)%*N* ...

until you find an empty slot.

# Linear Probing

Where do you store *65* ?

Array:



# Linear Probing

If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N*, (*t*+2)%*N*, ...

Array:

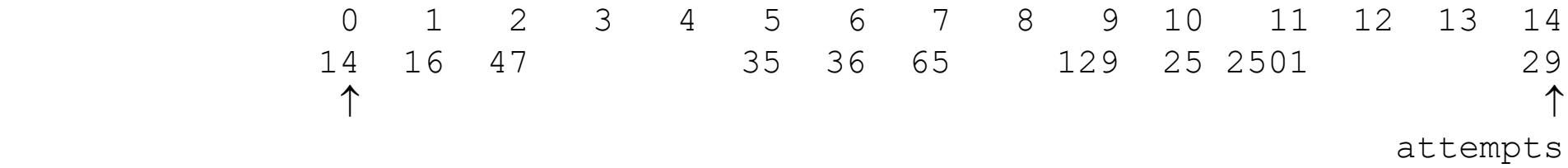
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 16 | 47 | | | 35 | 36 | 65 | | 129 | 25 | 2501 | | | 29 |
| | ↑ | | | | | | | | | | | | | |

Where would you store: 16, 14, 99, 127 ?

# Linear Probing

If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N*, (*t*+2)%*N*, ...

Array:

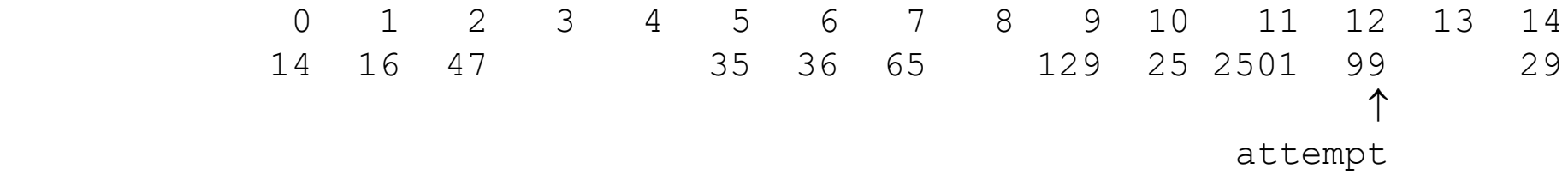



Where would you store: 14, 99, 127 ?

# Linear Probing

If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N*, (*t*+2)%*N*, ...

Array:

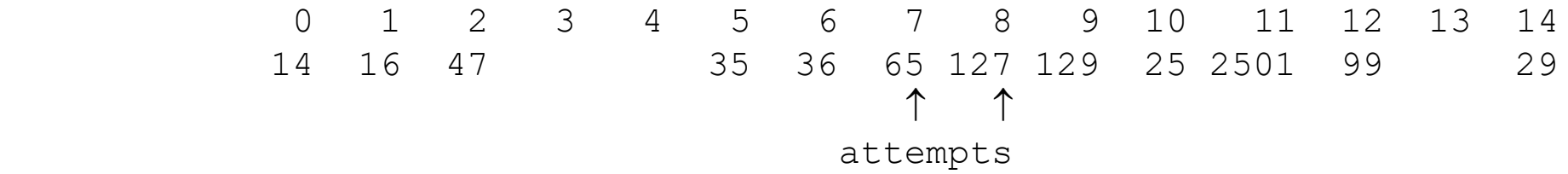



Where would you store: 99, 127 ?

# Linear Probing

If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N*, (*t*+2)%*N*, ...

Array:




Where would you store: 127 ?

# Quadratic Probing

Let key *x* be stored in element *f(x)=t* of the array

Array:

```
     0   1   2   3   4   5   6   7   8   9  10   11  12  13  14
         47          35  36          129  25 2501
                     65(?)
```

What do you do in case of a collision?

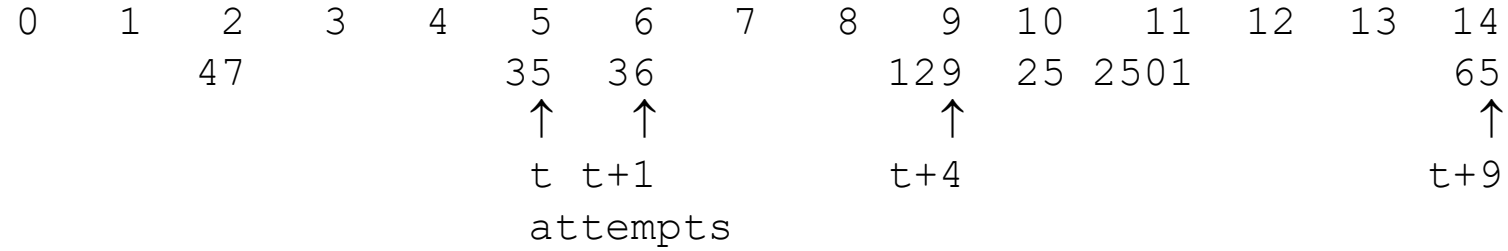
If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$, $(t+3^2)\%N$ ...

until you find an empty slot.

# Quadratic Probing

Where do you store *65* ? *f(65)=t=5*

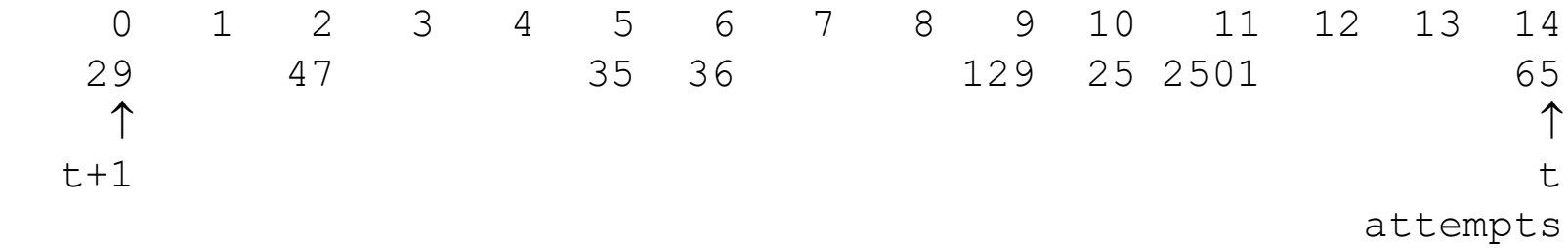
Array:



# Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

Array:

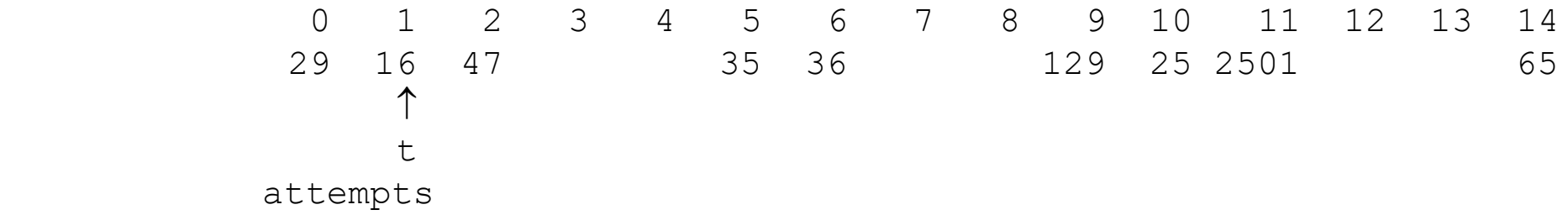



Where would you store: 29, 16, 14, 99, 127 ?

# Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

Array:

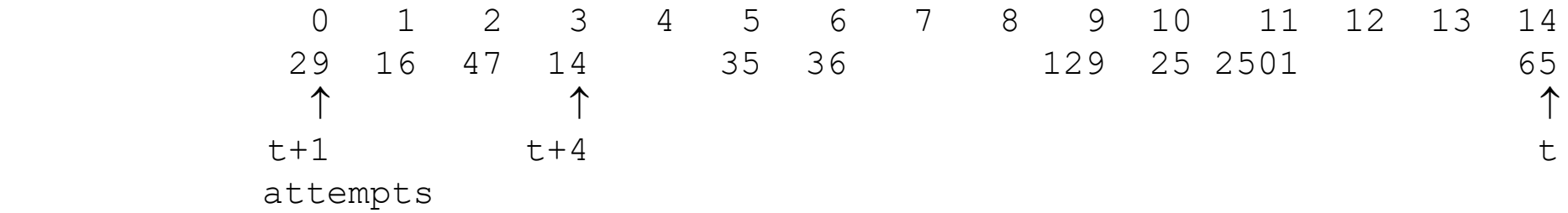



Where would you store: 16, 14, 99, 127 ?

# Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

Array:

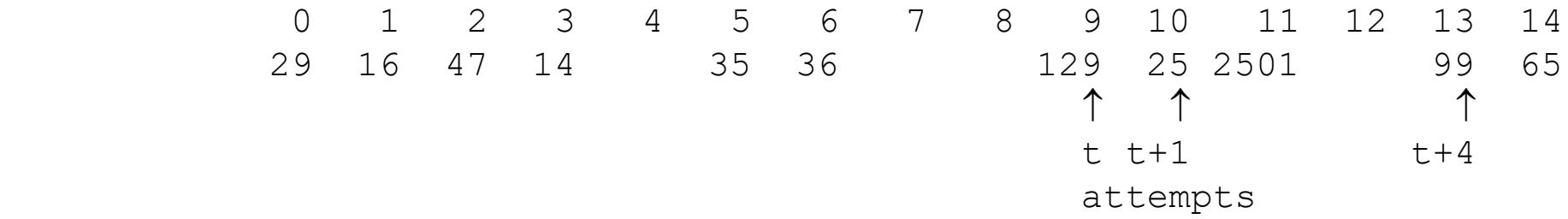



Where would you store: 14, 99, 127 ?

# Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

Array:

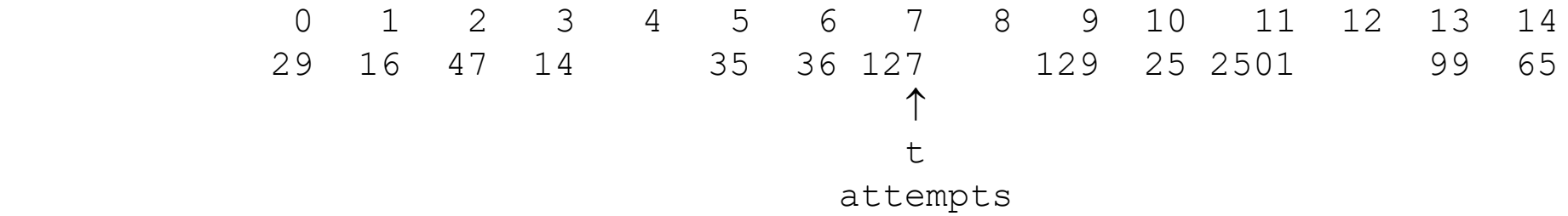



Where would you store: 99, 127 ?

# Quadratic Probing

If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

Array:




Where would you store: 127 ?

# Double Hashing

Let key $x$ be stored in element $f(x)=t$ of the array

Array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 47 | | | 35 | 36 | | | 129 | 25 | 2501 | | | |
| | | | | | 65(?) | | | | | | | | | |

What do you do in case of a collision?

Define a second hash function $f_2(x)=d$. Attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$, $(t+3d)\%N$ ...

until you find an empty slot.

# Double Hashing

- Use two hash functions
- If M is prime, eventually will examine every position in the table

```
double_hash_insert(K)
   if(table is full) error

   probe = h1(K)
   offset = h2(K)
  while (table[probe] occupied)
         probe = (probe + offset) mod M
   table[probe] = K
```

- Many of same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing does

# Double Hashing

- Typical second hash function

$$f_2(x) = R - ( x \% R )$$

where $R$ is a prime number, $R < N$

# Double Hashing

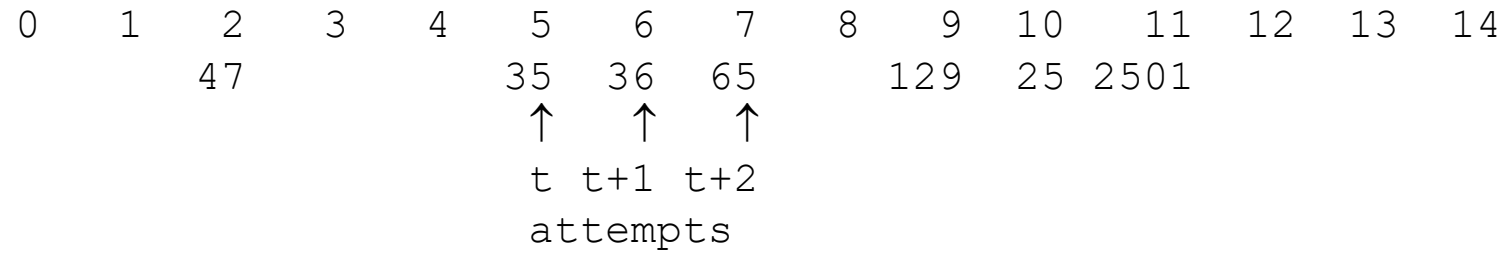
Where do you store *65* ? $f(65)=t=5$

Let $f_2(x)= 11 - (x \% 11)$ $f_2(65)=d=1$

Note: $R=11$, $N=15$

Attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$, $(t+3d)\%N$ ...

Array:



# Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$ ...

Let $f_2(x) = 11 - (x \,\%\, 11)$ $\quad$ $f_2(29)=d=4$

Array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 47 | | | 35 | 36 | 65 | | 129 | 25 | 2501 | | | 29 |
| | | | | | | | | | | | | | | ↑ |
| | | | | | | | | | | | | | | t |
| | | | | | | | | | | | | | | attempt |

Where would you store: 29, 16, 14, 99, 127 ?

# Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$ ...

Let $f_2(x) = 11 - (x \,\%\, 11)$ $\quad$ $f_2(16)=d=6$

Array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 16 | 47 | | | 35 | 36 | 65 | | 129 | 25 | 2501 | | | 29 |
| | ↑ | | | | | | | | | | | | | |
| | t | | | | | | | | | | | | | |
| attempt | | | | | | | | | | | | | | |

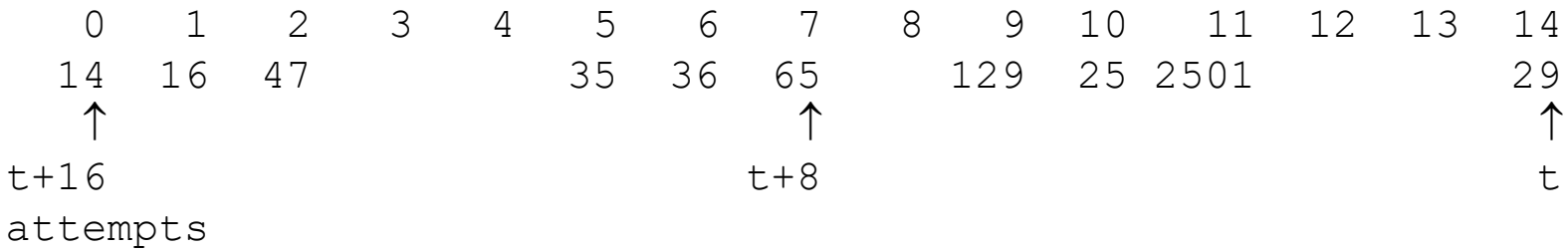
Where would you store: 16, 14, 99, 127 ?

# Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$ ...

Let $f_2(x) = 11 - (x \,\%\, 11)$ $\quad$ $f_2(14)=d=8$

Array:



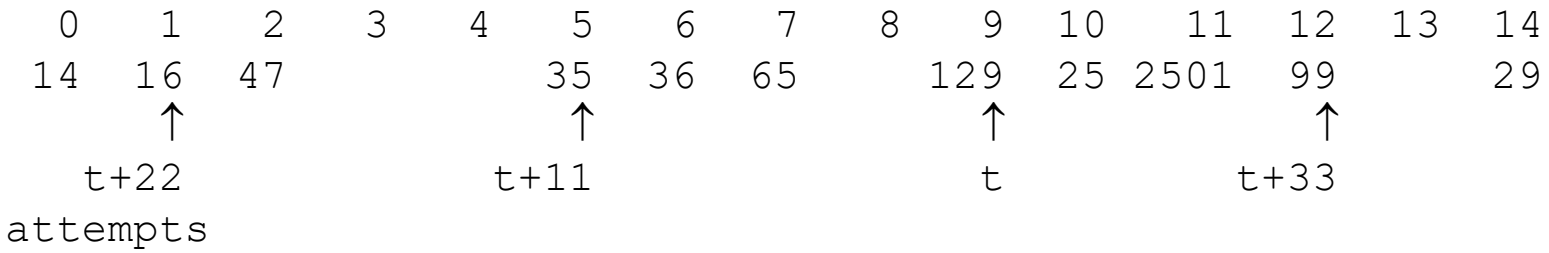

Where would you store: 14, 99, 127 ?

# Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$ ...

Let $f_2(x) = 11 - (x \,\%\, 11)$ $\quad$ $f_2(99)=d=11$

Array:




Where would you store: 99, 127 ?

# Double Hashing

If the hash table is *not full*, attempt to store key in array elements $(t+d)\%N$, $(t+2d)\%N$ ...

Let $f_2(x) = 11 - (x \,\%\, 11)$ $\quad\quad$ $f_2(127)=d=5$

Array:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 16 | 47 | | | 35 | 36 | 65 | | 129 | 25 | 2501 | 99 | | 29 |
| | | ↑ | | | | | ↑ | | | | | ↑ | | |
| | | t+10 | | | | | t | | | | | t+5 | | |

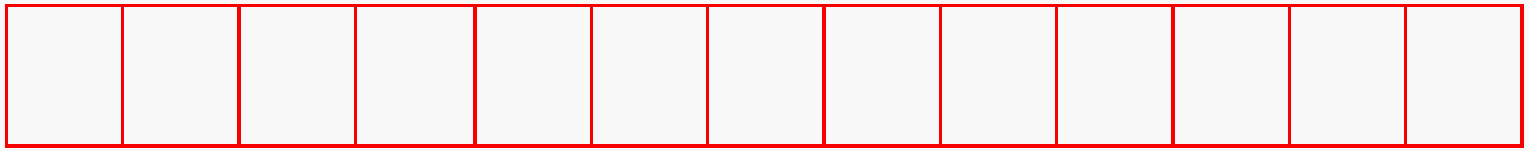
attempts

Where would you store: 127 ?

# Double Hashing Example

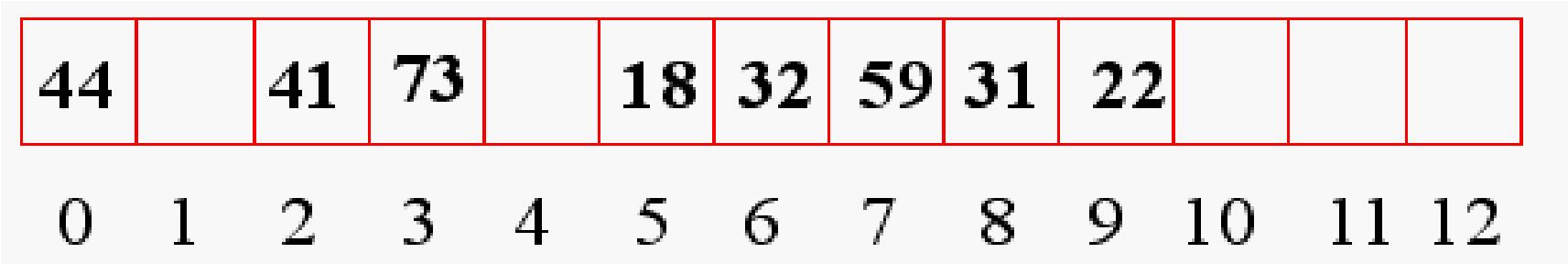
- $h_1(K) = K \bmod 13$
- $h_2(K) = 8 - K \bmod 8$
- we want $h_2$ to be an offset to a

18 41 22 44 59 32 31 73



0 1 2 3 4 5 6 7 8 9 10 11 12

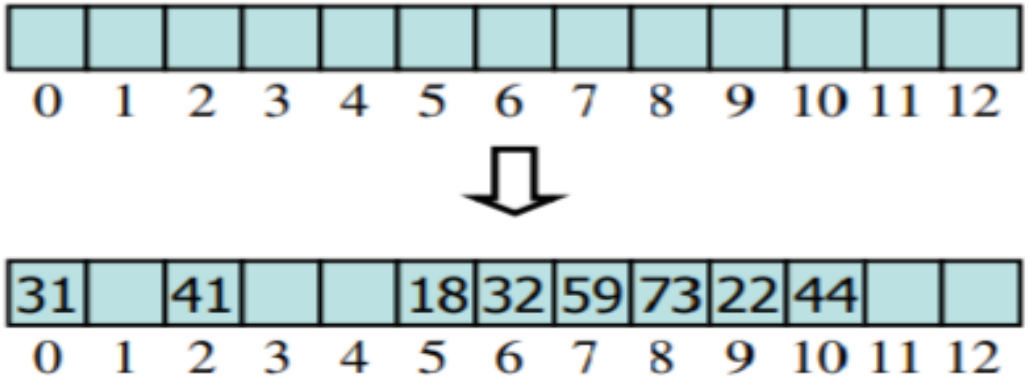
# Double Hashing Example (cont.)



# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

| $k$ | $h(k)$ | $d(k)$ | Probes | | |
|---|---|---|---|---|---|
| 18 | 5 | 3 | 5 | | |
| 41 | 2 | 1 | 2 | | |
| 22 | 9 | 6 | 9 | | |
| 44 | 5 | 5 | 5 | 10 | |
| 59 | 7 | 4 | 7 | | |
| 32 | 6 | 3 | 6 | | |
| 31 | 5 | 4 | 5 | 9 | 0 |
| 73 | 8 | 4 | 8 | | |



# Quadratic Probing

- Linear probing:
  Insert item (k, e)
  i = h(k)
  A[i] is occupied
  Try A[(i+1) mod N]: used
  Try A[(i+2) mod N]
  and so on until
  an empty bucket is found

- Quadratic probing
  A[i] is occupied
  Try A[(i+1) mod N]: used
  Try A[(i+$2^2$) mod N]: used
  Try A[(i+$3^2$) mod N]
  and so on

- May not be able to find an empty bucket if *N* is not prime, or the hash table is at least half full

# Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jd(k)) \bmod N$$
for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size $N$ must be a prime to allow probing of all the cells

Insert item (k, e)

i = h(k)

A[i] is occupied

Try A[(i+d(k))mod N]: used

Try A[(i+2d(k))mod N]: used

Try A[(i+3d(k))mod N]

and so on until

an empty bucket is found

# Comparing Collision Handling Schemes

- Separate chaining:
  – simple implementation
  – faster than open addressing in general
  – using more memory
- Open addressing:
  – using less memory
  – slower than chaining in general
  – more complex removals
- Linear probing: items are clustered into contiguous runs.
- Quadratic probing: secondary clustering.
- Double hashing: distributes keys more uniformly than linear probing does.