# CSC 311

## Lectures on
# Data Structures

by

Dr. Marek A. Suchenek ©

Computer Science
CSUDH

# Copyrighted material

# CSC 311

## Lecture 12
# Hashing

**ADT TABLE, Applications, Implementations, Analysis**

| Key<br>*K* = Airport Code | Associated Information<br>*I* = City |
| --- | --- |
| AKL | Auckland, New Zealand |
| DCA | Washington, D.C. |
| FRA | Frankfurt, Germany |
| GCM | Grand Cayman, Cayman Islands |
| GLA | Glasgow, Scotland |
| HKG | Hong Kong, China |
| LAX | Los Angeles, California |
| ORY | Paris, France |
| PHL | Philadelphia, Pennsylvania |

# ADT TABLE

1. Construct(N) an empty table T of size N
2. Insert(K, I) an entry with key K and info I into T
3. Delete(K) an entry with key K from T
4. Find(K) an entry with key K in T and, if found, return info I of that entry.

# ADT TABLE

Keys – the set of all keys

N – size of T

The size #(Keys) of the set Keys is much larger than N

Hash function h

h: Keys → {0, … N-1}

Hash function h assigns to every key K an index h(N)

in table T, with even distribution of probability.

# Construct(N)

1. Construct an array T of size N.
2. for (int i = 0; i < N; i++)

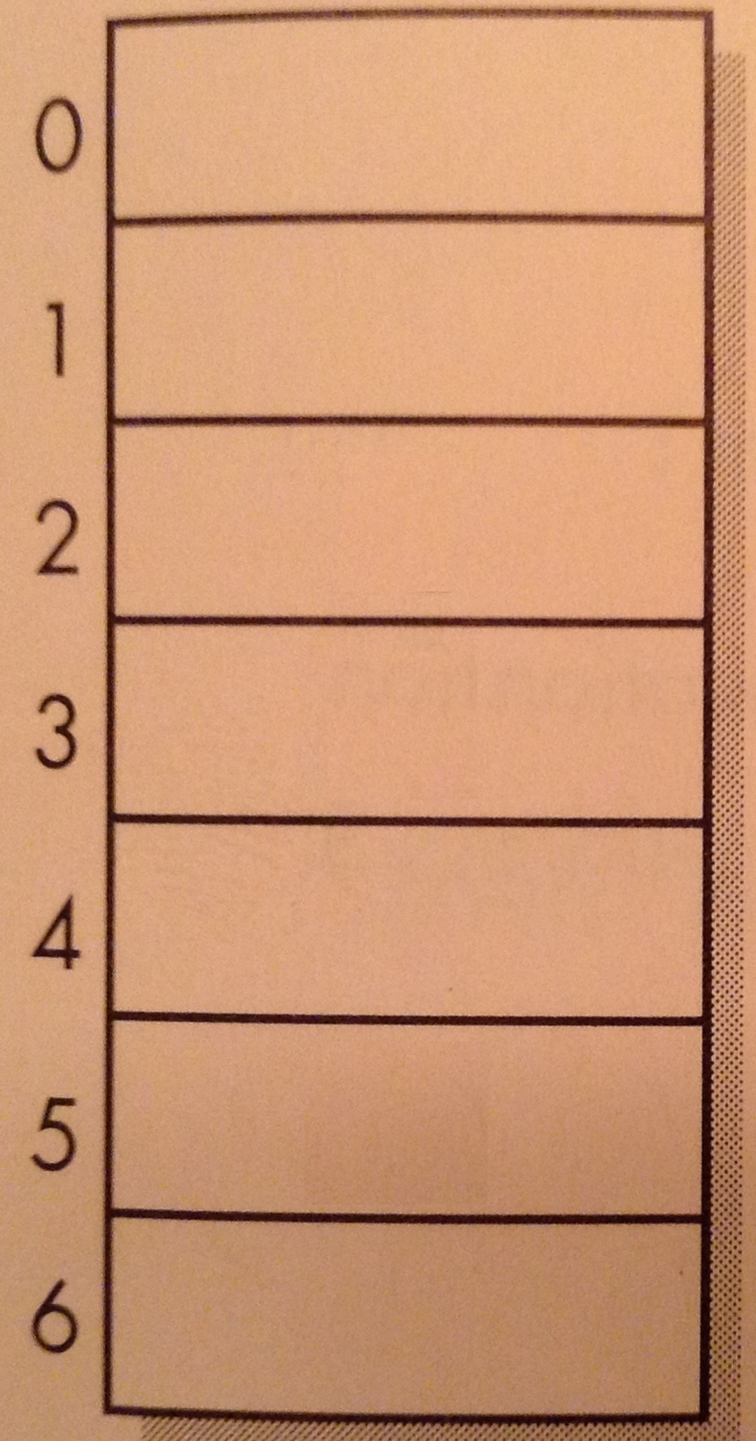
   T(i) = "empty"

# Insert(K, I)

Example:

$$h(L_n) = n \% N$$

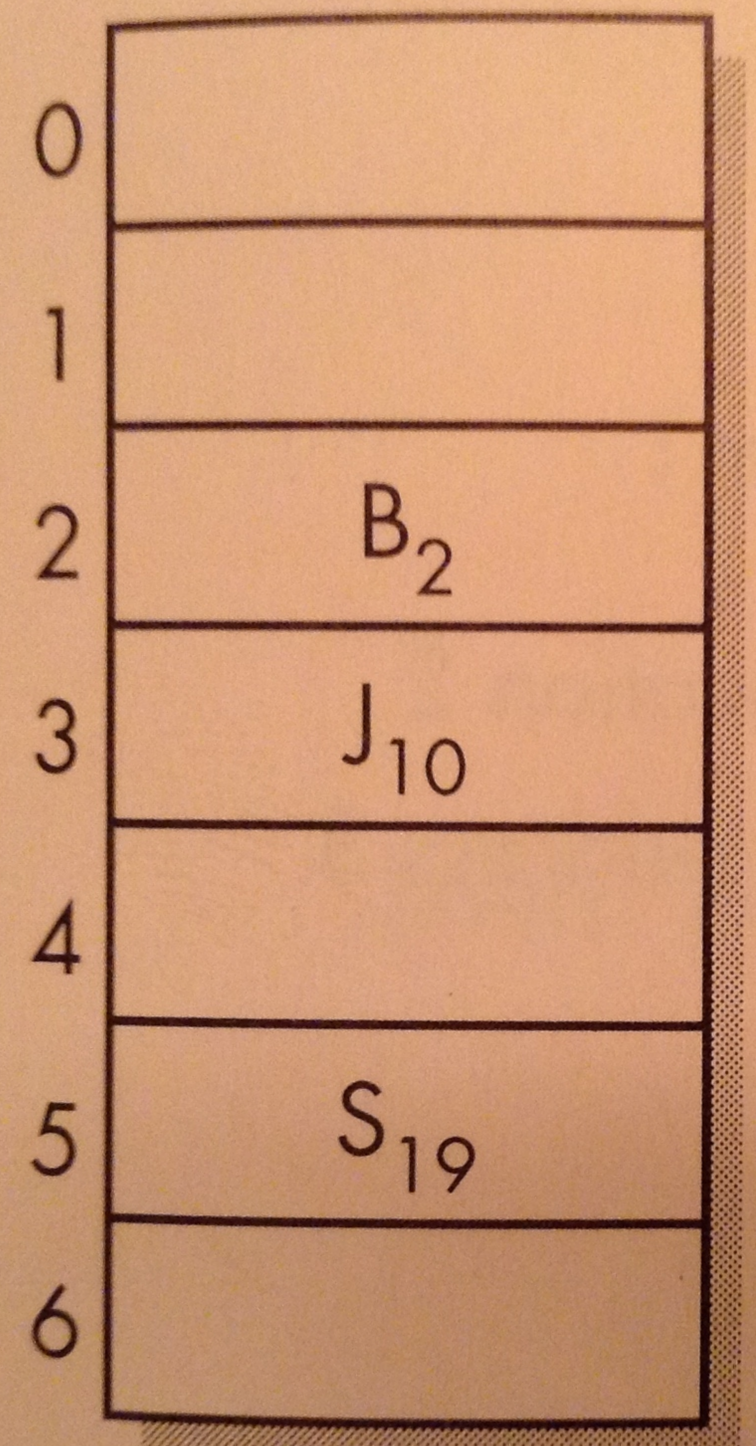
$$N = 7$$


0
1
2
3
4
5
6

# Insert(K, I)

If no deletions were made in the past:

1. Compute m = h(K).
2. If T(m) is empty, store (K, I) in T(m).
3. If T(m) is not empty, collision occurs and needs to be resolved.


0
1
2
B2
3
J10
4
5
S19
6

# Insert(K, I)

The following collision resolution methods will be considered:

1. linear probing,
2. double hashing, and
3. separate chaining.

# Linear probing

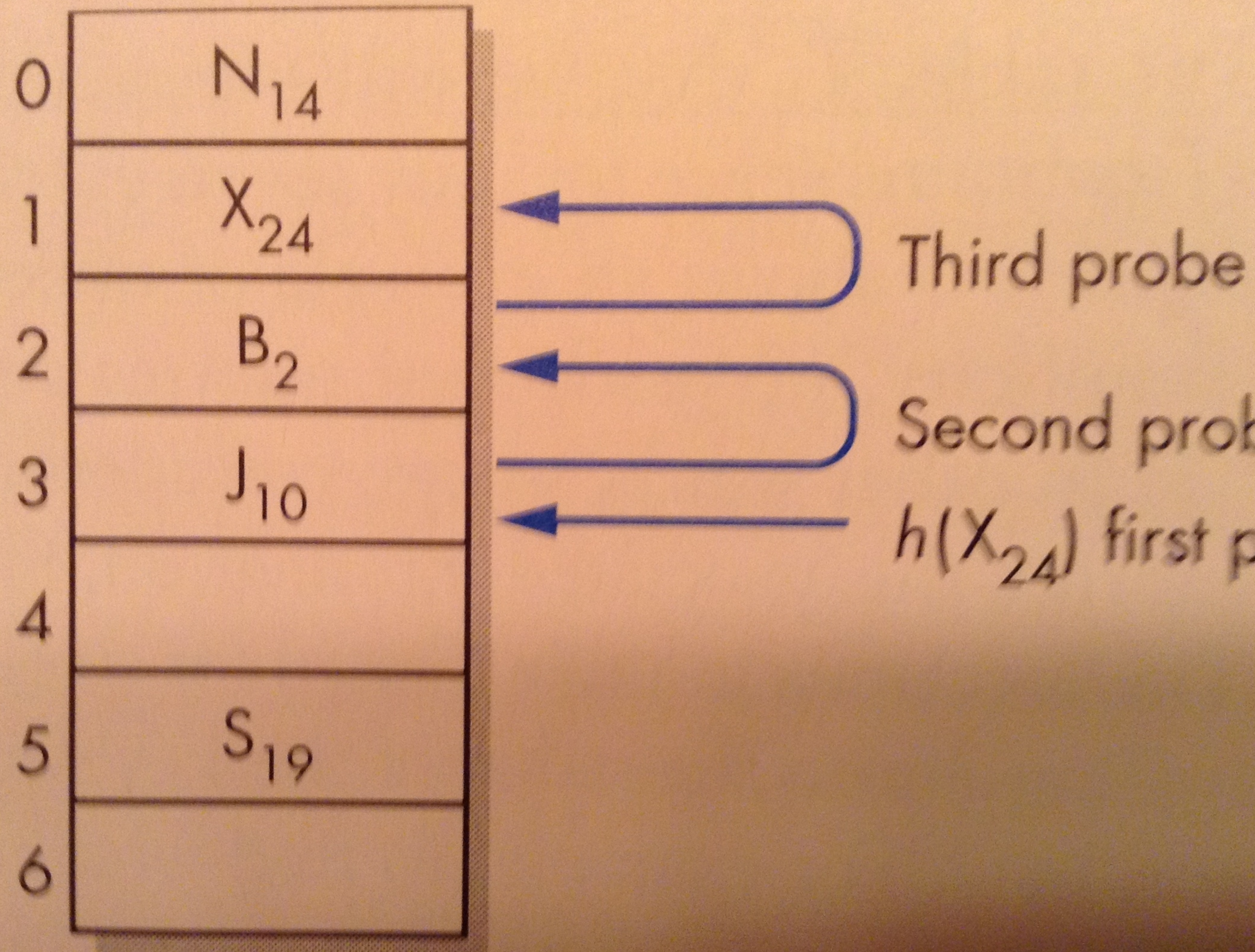
This is slightly different version of linear probing than described in the textbook.

We know that T(m) is not free, that is a collision occurred.

for (int i = 1; i < N; i++)

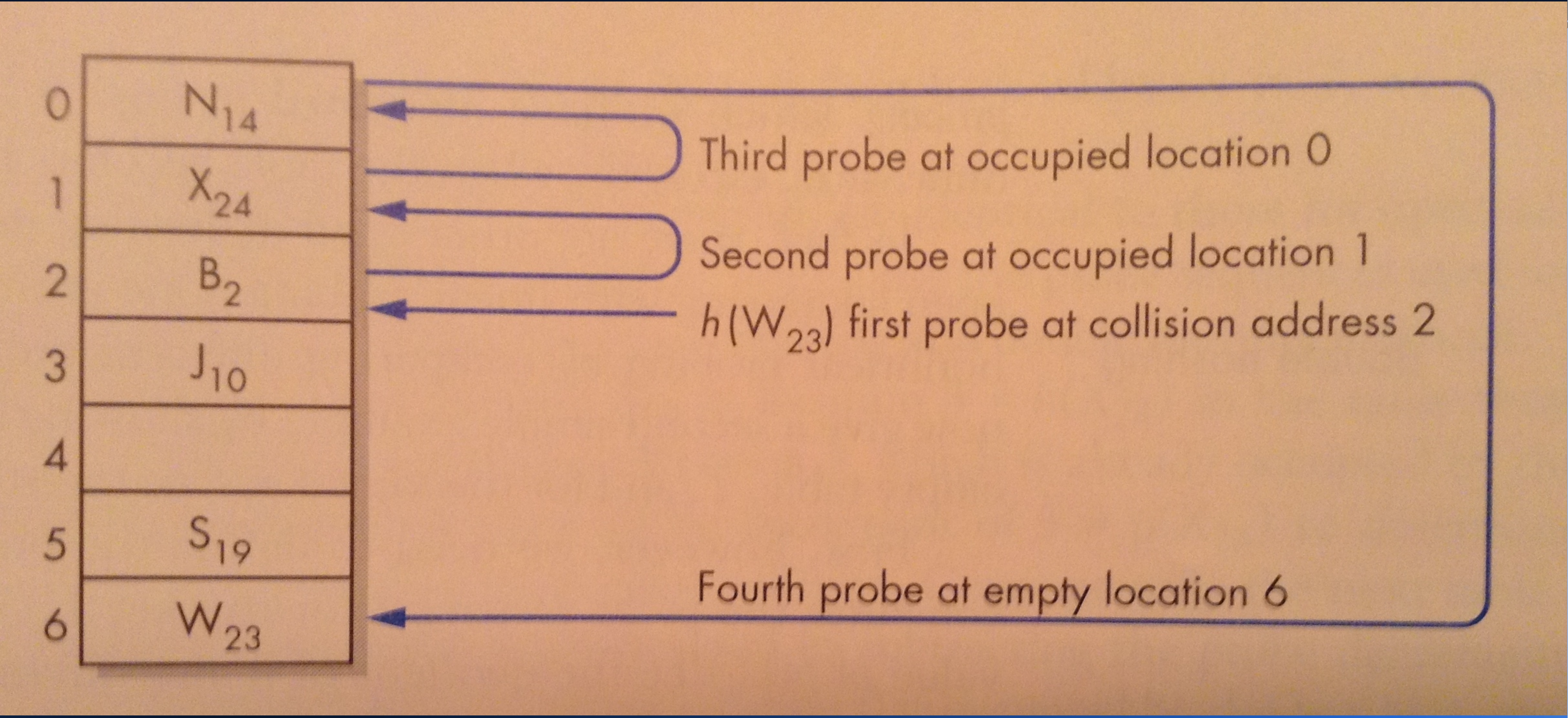
- {m = (m+1) % N; // for wrap-around effect
- If (T(m) is free)
  - {store (K, I) in T(m);
  - break;} }

// no room for insertion – the hash table needs to be stretched



0
1
2
3
4
5
6
$N_{14}$
$X_{24}$
$B_2$
$J_{10}$
$S_{19}$
Third probe
Second prob
$h(X_{24})$ first p

# Linear probing



# Double hashing

This is slightly different version of double hashing than described in the textbook.

We know that T(m) is not free, that is a collision occurred.

int incr = p(K) // the second hash function, p(K) > 0

for (int i = 1; i < N; i++)

- {m = (m+incr) % N; // for wrap-around effect
- If (T(m) is free)
  - {store (K, I) in T(m);
  - break;} }

// no room for insertion – the hash table needs to be stretched

# Double hashing

In order to not waste storage,

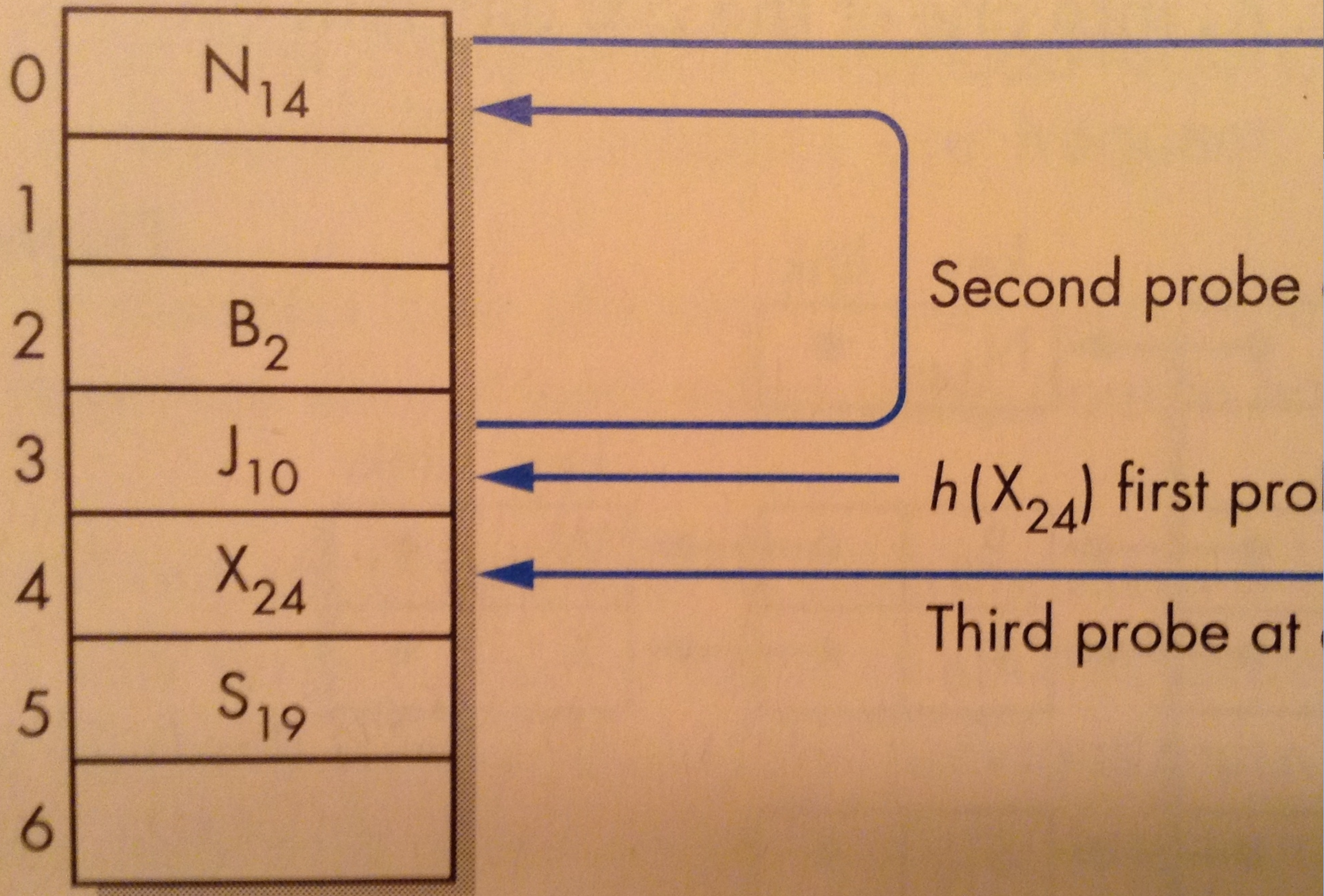
p(K) and N must have no common divisor > 1.

(i.e., p(K) and N must be relatively prime).

For example, if $N = 2^k$ then p(K) must be even.

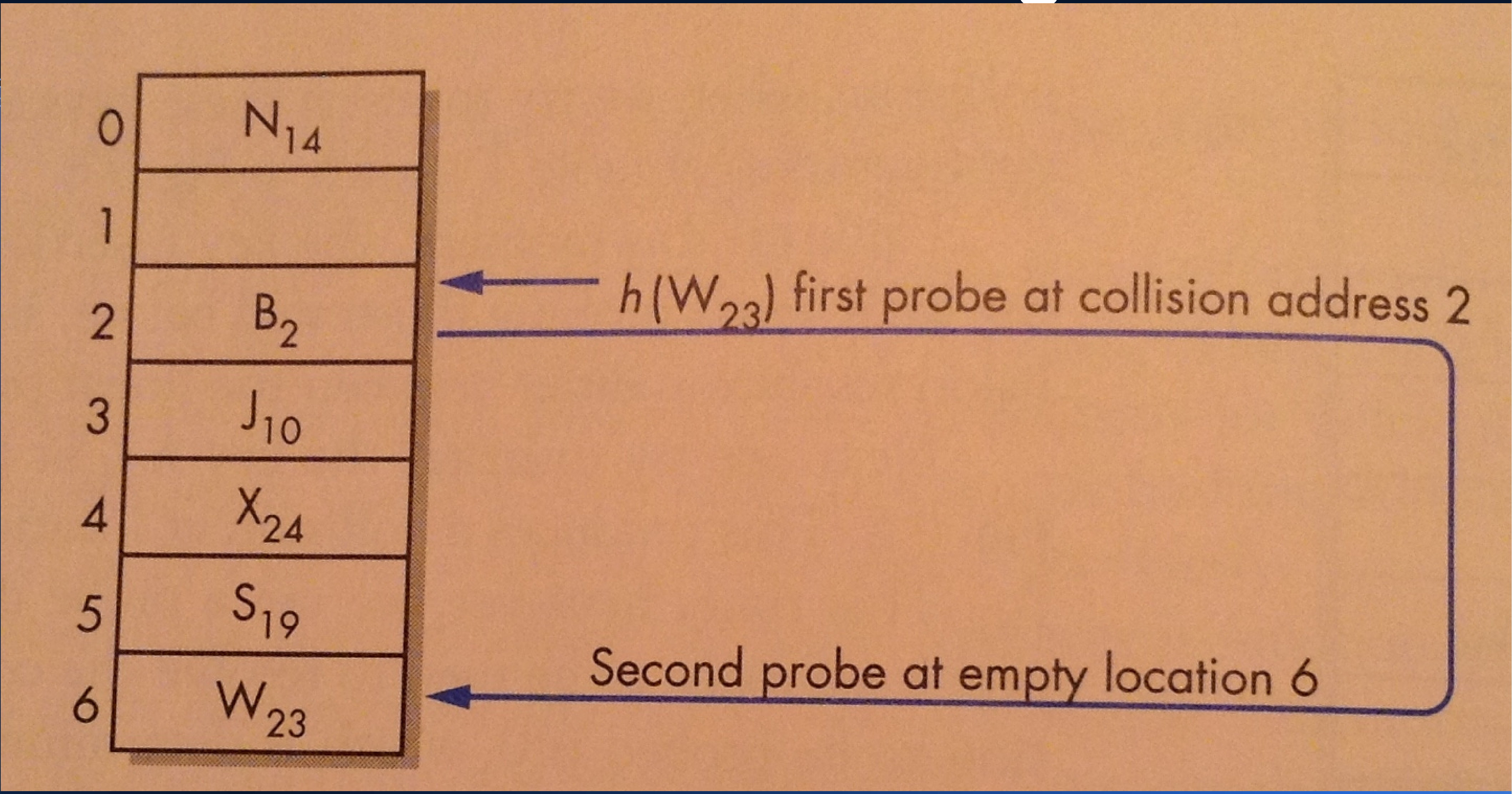
If N is prime then p(K) > 1.

| Key = $L_n$ | $h(L_n)$ | $p(L_n)$ |
| --- | --- | --- |
| $J_{10}$ | 3 | 1 |
| $B_2$ | 2 | 1 |
| $S_{19}$ | 5 | 2 |
| $N_{14}$ | 0 | 2 |
| $X_{24}$ | 3 | 3 |
| $W_{23}$ | 2 | 3 |



0
1
2
3
4
5
6
N14
B2
J10
X24
S19
Second probe
h(X24) first pro
Third probe at

# Double hashing



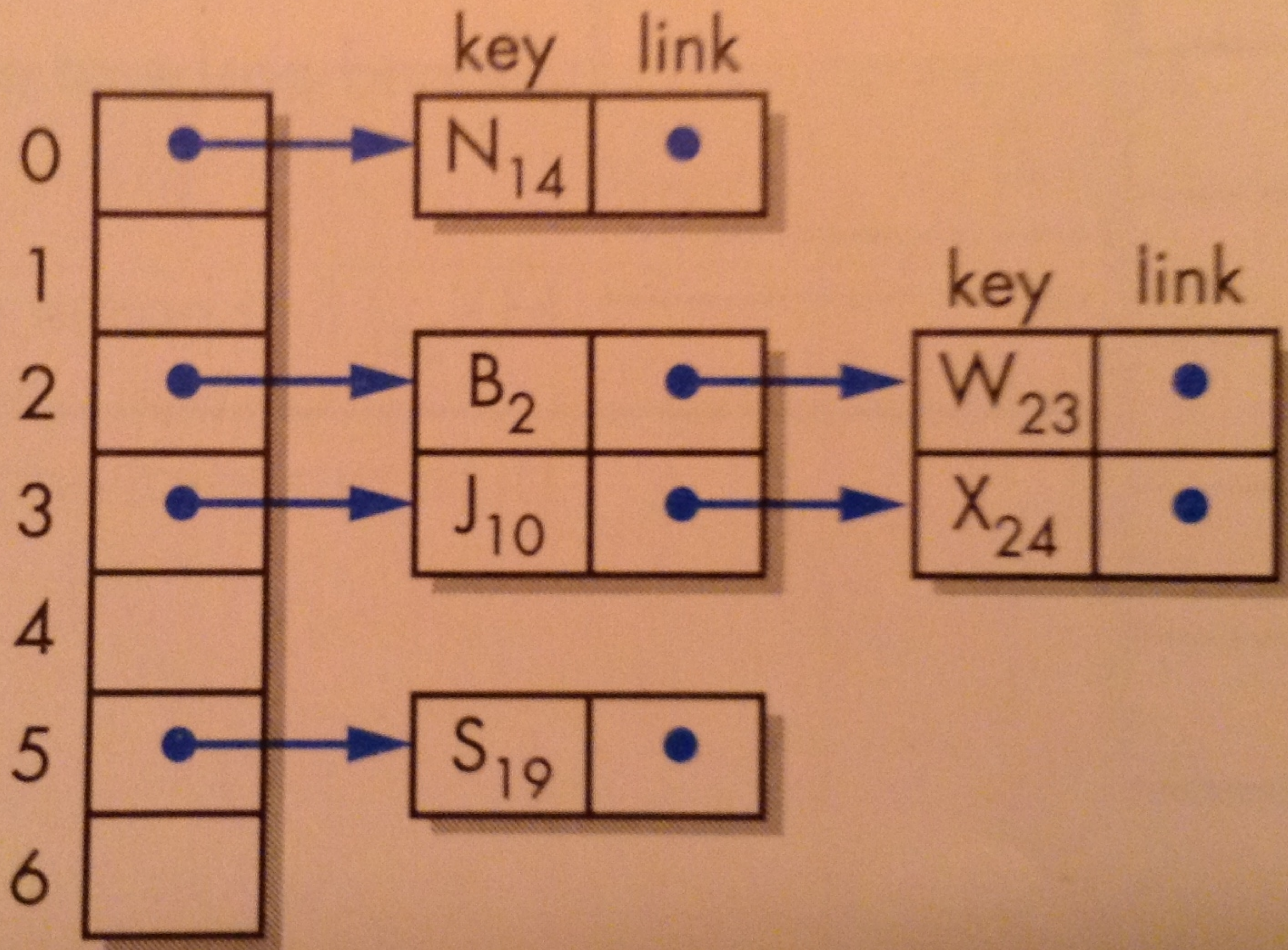
# Separate chaining

This is slightly different version of linear probing than described in the textbook.

We know that T(m) is not free, that is a collision occurred.

Add (K, I) to the beginning of a linked list $L_m$ of all entries (M, J) such that

$$h(M) = m$$


key
link
0
1
2
3
4
5
6
$N_{14}$
key
link
$B_2$
$W_{23}$
$J_{10}$
$X_{24}$
$S_{19}$

# Find(K)

Case of linear probing or double hashing

1. Compute m = h(K).
2. while (T(m) != “empty”)

if (T(m) != “deleted” && T(m) == K) return info from T(m);

else update m following the collision resolution method used by Insert;

// the entry (K, I) not found in T

# Find(K)

Case of linear probing or double hashing

1. Compute m = h(K).
2. while (T(m) != "empty" && T(m) != "deleted")

   if (T(m) == K) return info from T(m);

   else update m following the collision resolution method used by Insert;

// the entry (K, I) not found in T

# Find(K)

Case of separate chaining

1. Compute m = h(K).
2. if (T(m) != "empty")

    {Find K on list $L_m$;

    - if found return it;

// the entry (K, I) not found in T

# Find(K)

Case of separate chaining

1. Compute m = h(K).
2. if (T(m) != “empty”)

   {f = Find.$L_m$(K);

   - if found return info(K);

// the entry (K, I) not found in T

# Delete(K)

Case of linear probing or double hashing

1. Follow the same steps as in Find(K);
2. if (found)

        {T(m) = "deleted";
        return;}

    // else do nothing;

    // the entry (K, I) not found in T

# Delete(K)

Case of separate chaining

1. Compute m = h(K).
2. if (T(m) != "empty")

   {Delete.$L_m$(K);

   - if ($L_{m.}$empty()) T(m) = "empty";

# Insert(K, I)

General case (a deletion could have been made in the past):

1. Execute Find(K), remembering location n of the first free indicator.
2. If not found then store (K, I) in T(n).

“free” means “empty” or “deleted”

# To be continued ...

in Lecture Notes ...