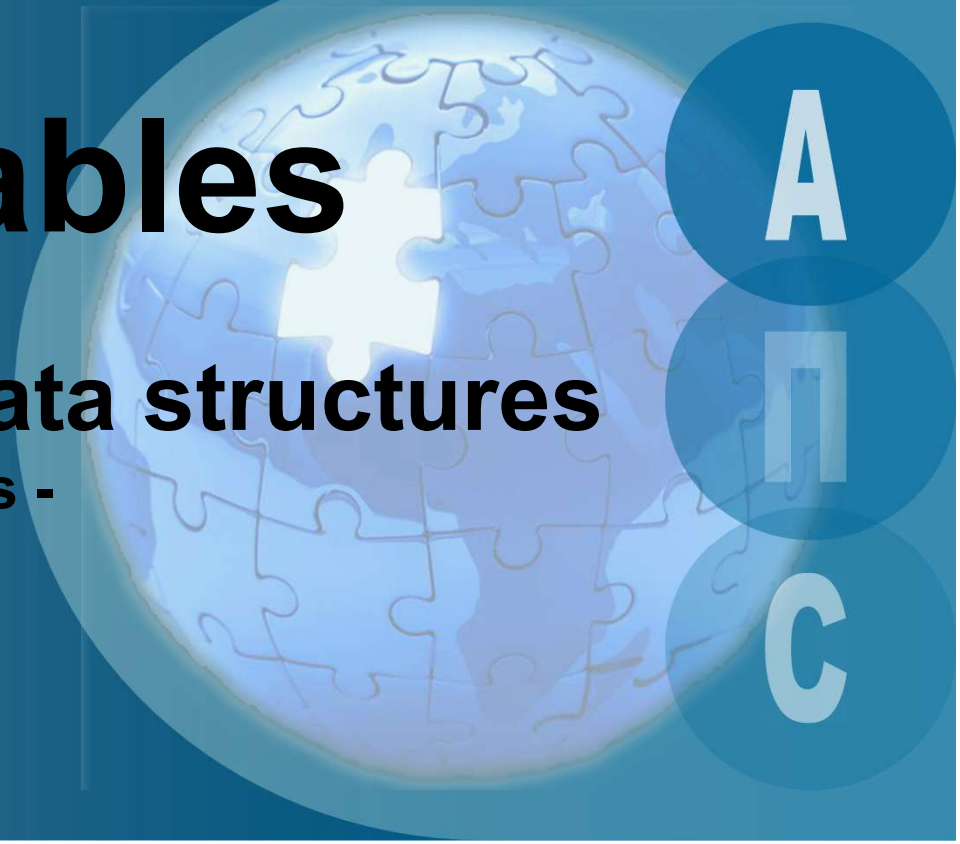
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Hash tables

Algorithms and data structures

- lectures -



Содржина

- Hash tables
- Hashing methods
- Hash functions
- Hash applications

Hash table abstract data type

- Provide efficient insertion and find operations
- No ordering required
- No efficient removal required
- In the ideal case, insert and find have O(1) in the worst case
  - But will have to settle for O(1) in the average case

Implementation alternatives

- Array based implementation could be the only option
- Single linked lists do not provide access in O(1) and search in O(1)

Hashing principles

- Consider a (key, value) **map**
- If a map's keys are small integers, we can represent the map by a key-indexed array.
  - Search, insertion, and deletion then have time complexity $O(1)$.
- The idea is we can extend this approach with keys of arbitrary types

Hashing principles

- **Hashing:** translate each key to a small integer, and use that integer to index an array

- **Hash table:** is an array of *m* **buckets**, together with a **hash function** *hash*(*k*) that translates each key *k* to a bucket index (in the range 0…*m*–1).

Illustration

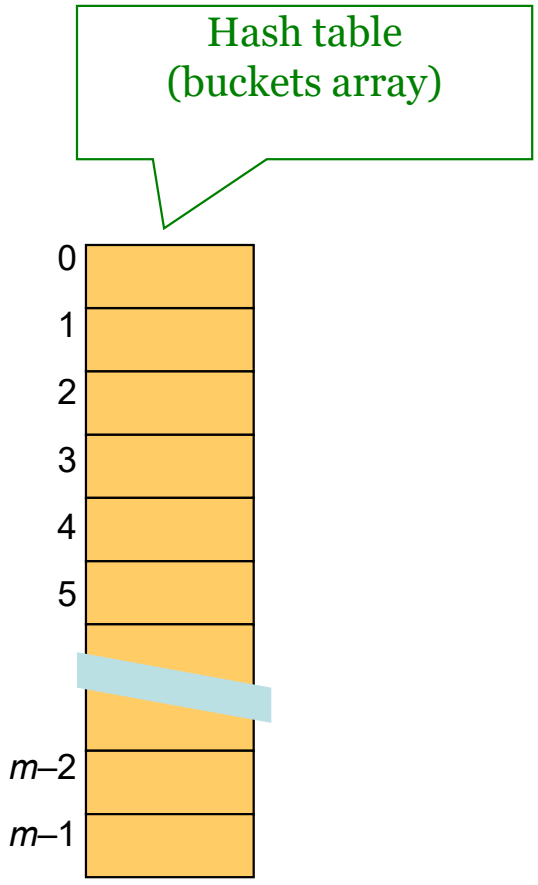
- Hashing

key	value
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_n	v_n

Illustration

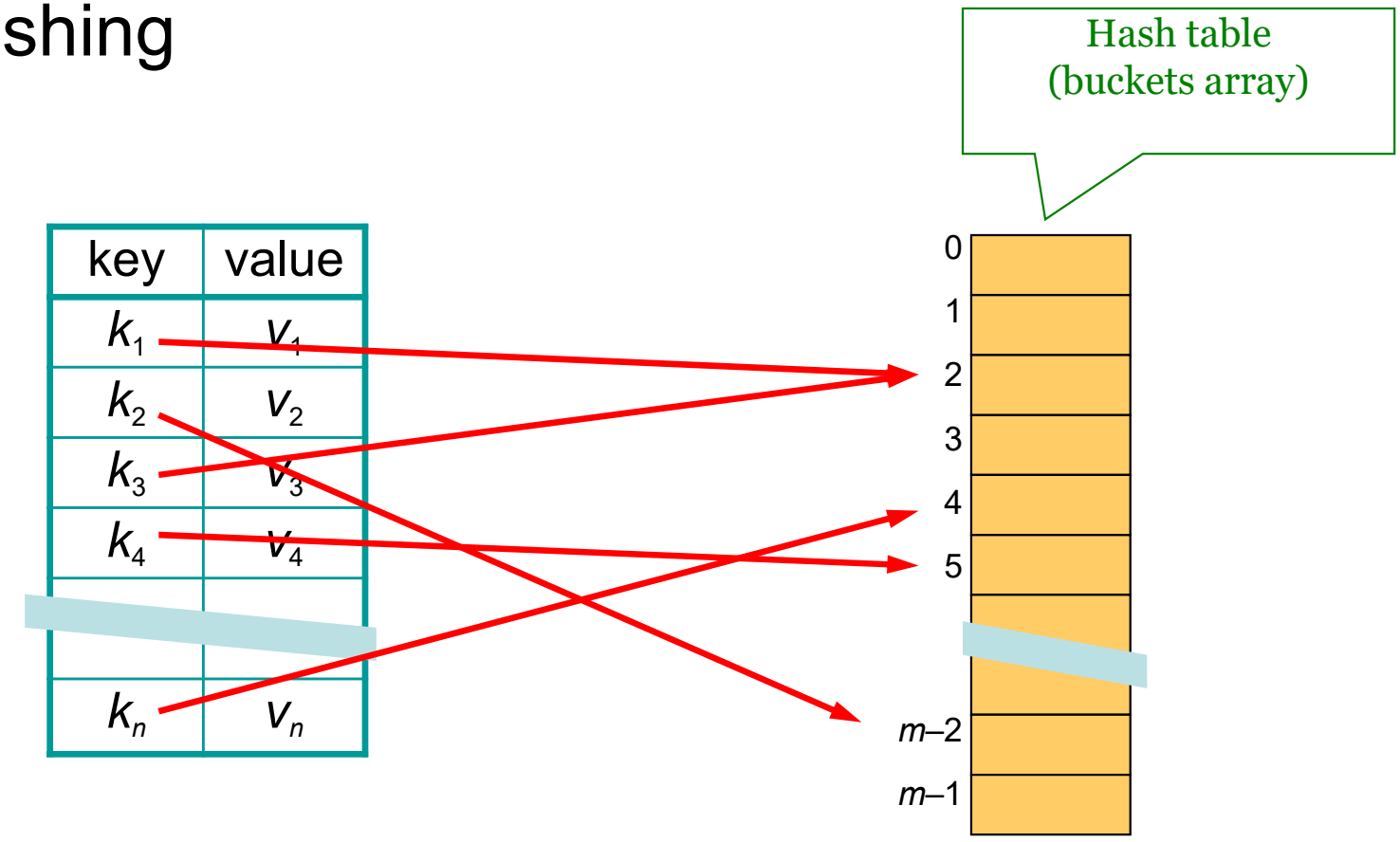
- Hashing

key	value
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_n	v_n



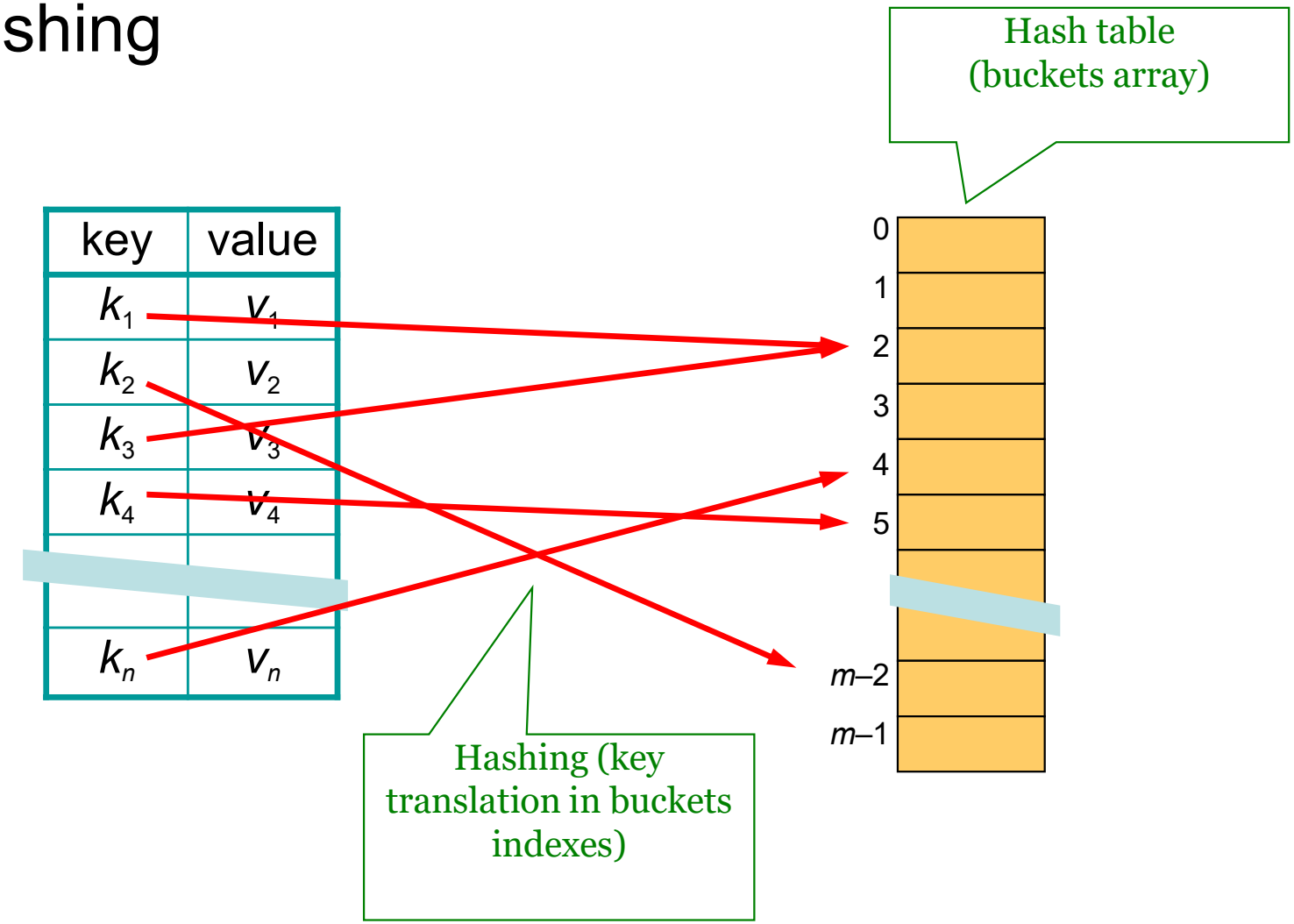
Illustration

- Hashing



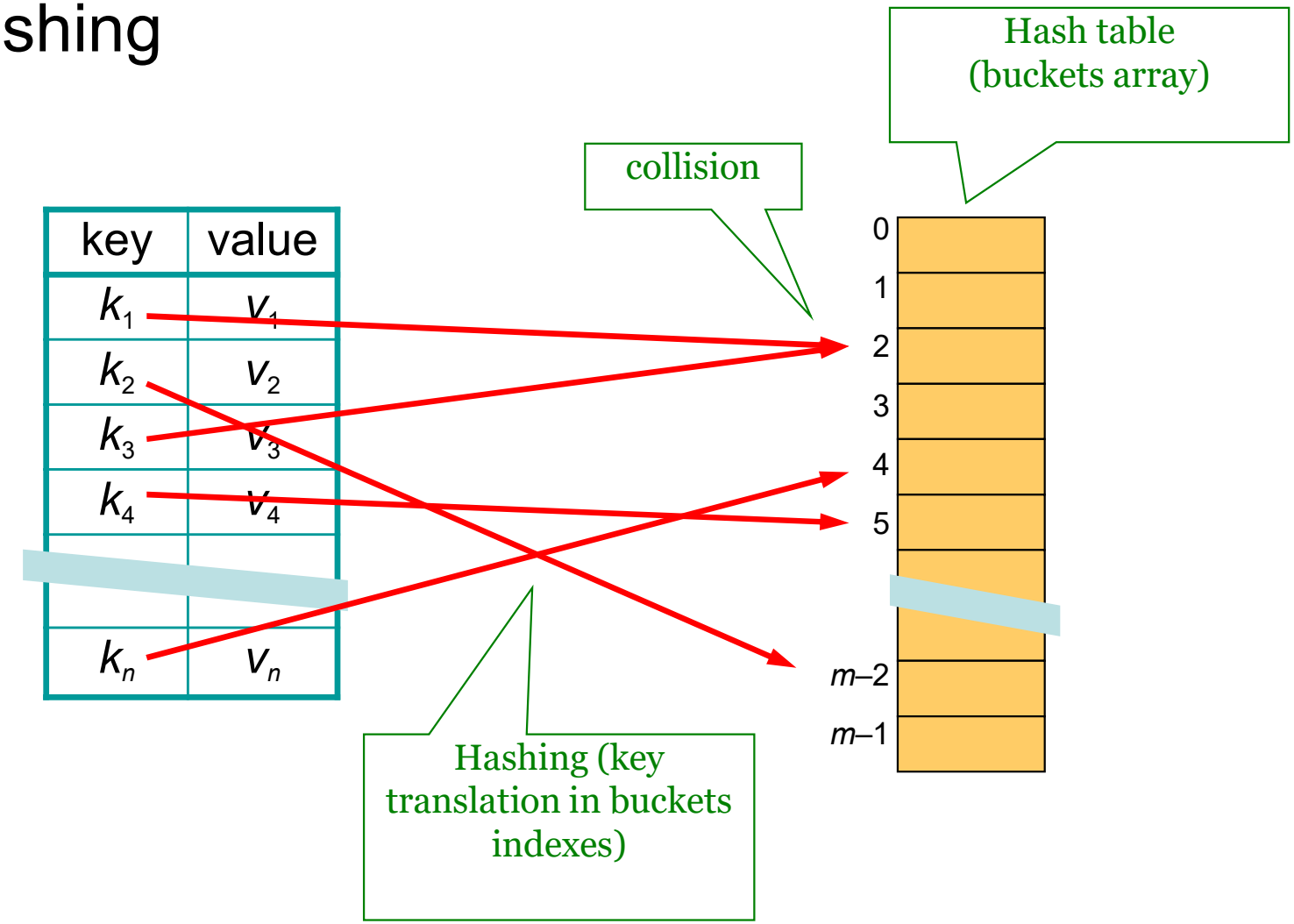
Illustration

- Hashing

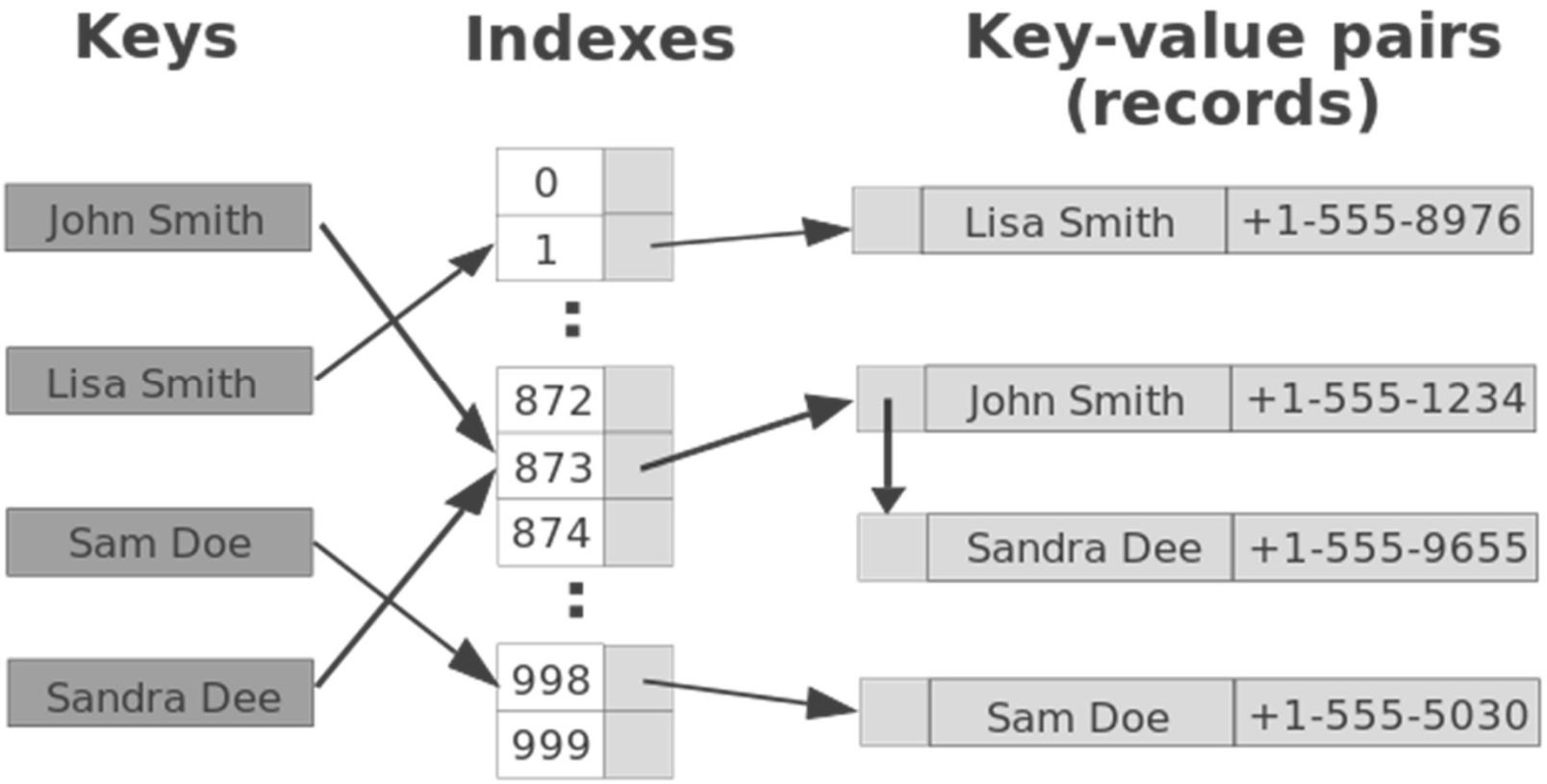


Illustration

- Hashing



Illustration



Hashing principles

- Each key *k* has a **home bucket** in the hash table, namely the bucket with index *hash*(*k*).
- To **insert** a new entry with key *k* into the hash table, assign that entry to *k*'s home bucket.
- To **search** for an entry with key *k* in the hash table, look in *k*'s home bucket.
- To **delete** an entry with key *k* from the hash table, look in *k*'s home bucket.

Hashing principles

- The hash function must be consistent:

  $k_1 = k_2$, hence $hash(k_1) = hash(k_2)$.

- In general case, this function does the mapping many-to-one
- It means that for different keys the same home bucket can be obtained :

  $k_1 \neq k_2$, but $hash(k_1) = hash(k_2)$.

  This is called a **collision**.

Hashing principles

- The hash function must be consistent:

  $k_1 = k_2$, hence $hash(k_1) = hash(k_2)$.

- In general case, this function does the mapping many-to-one
- It means that for different keys the same home bucket can be obtained :

  $k_1 \neq k_2$, but $hash(k_1) = hash(k_2)$.

  This is called a **collision**.

Always prefer a hash function that makes collisions relatively infrequent.

Example: Words hashing

- Suppose that the keys are English words.
- Possible hash function:

  m = 26

  $hash(w)$ = (initial letter of w) – ‘A’
- All words with initial letter ‘A’ share bucket 0;

  …

  all words with initial letter ‘Z’ share bucket 25.

Example: Words hashing

- Suppose that the keys are English words.
- Possible hash function:

  m = 26

  $hash(w)$ = (initial letter of w) – ‘A’
- All words with initial letter ‘A’ share bucket 0;

  …

  all words with initial letter ‘Z’ share bucket 25.

This is a convenient choice for illustrative purposes.
But it is a poor choice for practical purposes: collisions are likely to be frequent in some buckets.

General implementations

- **Closed-bucket hash table**:
  - Each bucket may be occupied by several entries.
  - Buckets are completely separate.
- **Open-bucket hash table**:
  - Each bucket may be occupied by at most one entry.
  - Whenever there is a collision, displace the new entry to another bucket.

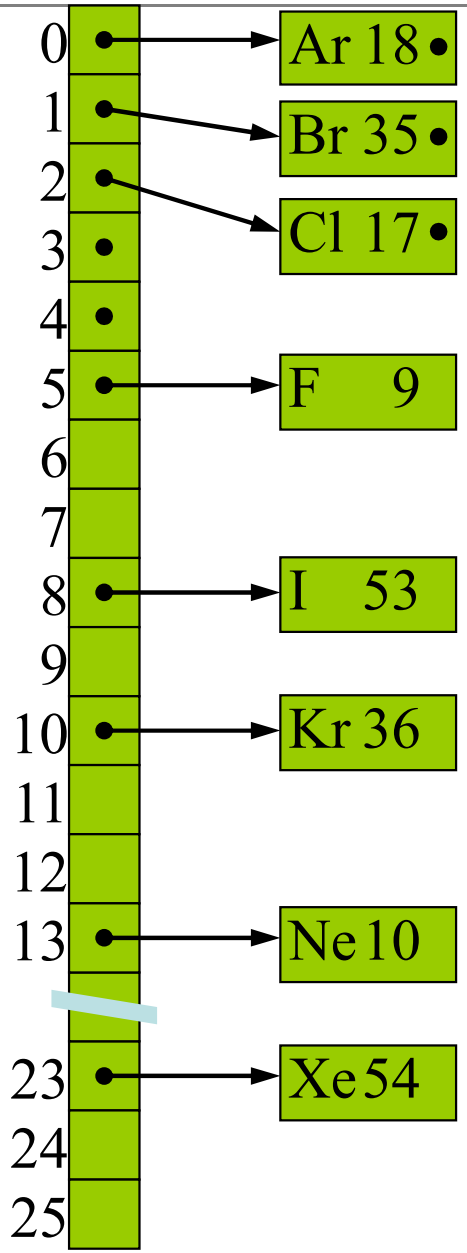
Closed-bucket hash table

- **Closed-bucket hash table** (**CBHT**):
  - Each bucket may be occupied by several entries.
  - Buckets are completely separate. No overflow, that is why they are called closed.
- Simplest implementation:
  - each bucket is an SLL,
  - there is and array of single linked lists

Closed-bucket hash table

- Illustration (no collisions):

element	number
F	9
Ne	10
Cl	17
Ar	18
Br	35
Kr	36
I	53
Xe	54



Chemical elements hashing:

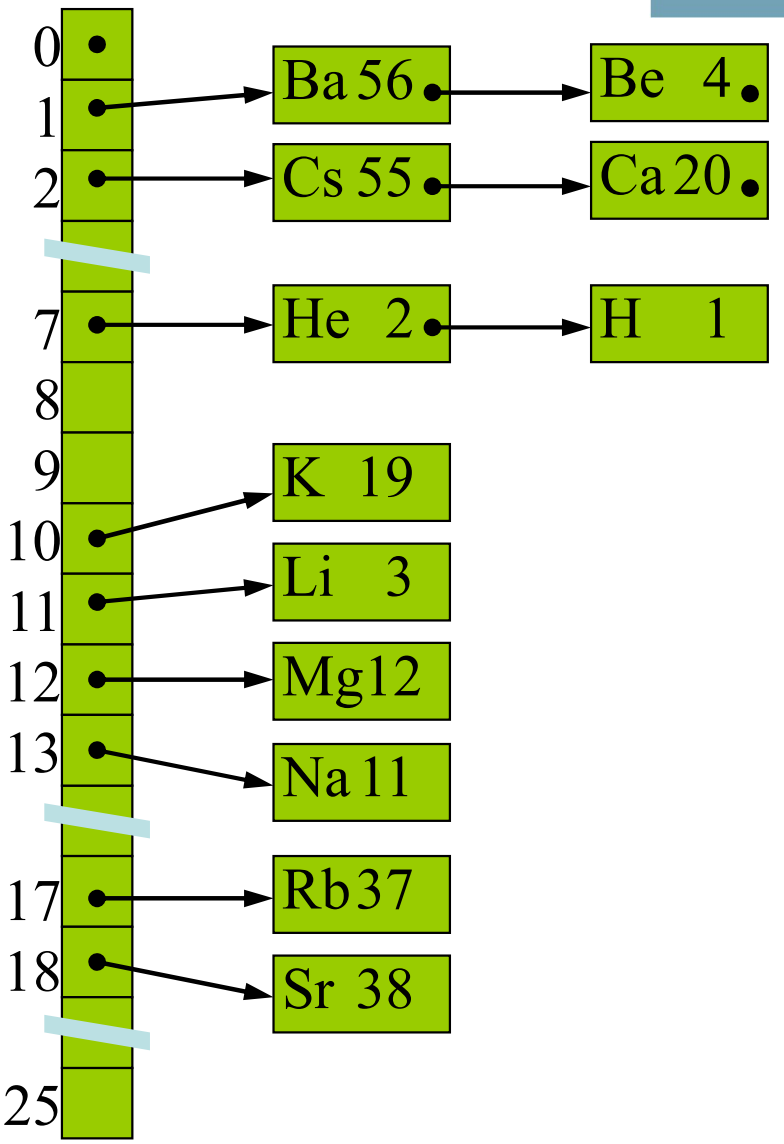
m = 26

hash(*e*) = (the first letter is) – 'A'

Closed-bucket hash table

- Illustration (with collisions):

element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



CBHT search

- To find which if any node of a CBHT contains an entry whose key is equal to target-key:
  - 1. Set b to hash(target-key)
  - 2. Find which if any node of the SLL of bucket b contains an entry whose key is equal to *target-key*, and terminate with that node as answer.

CBHT insertion

- To insert the entry (key, val) into a CBHT:
  - 1. Set b to hash(key).
  - 2. Insert the entry (key, val) into the SLL of bucket b, replacing any existing entry whose key is key.
  - 3. Terminate.

CBHT deletion

- To delete the entry (if any) whose key is equal to *key* from a CBHT:
  - 1. Set *b* to *hash*(*key*).
  - 2. Delete the entry (if any) whose key is equal to *key* from the SLL of bucket *b*.
  - 3. Terminate.

CBHT analysis

- Let the number of entries be n.
- In the best case, no bucket contains more than (say) 2 entries:
  - Max. no. of comparisons = 2
  - Best-case time complexity is O(1).
- In the worst case, one bucket contains all n entries:
  - Max. no. of comparisons = n
  - Worst-case time complexity is O(n).

CBHT design

- CBHT design consists of:
  - choosing the number of buckets *m*
  - choosing the hash function *hash*.
- Design aims:
  - collisions should be infrequent
  - entries should be distributed evenly among the buckets, such that few buckets contain more than about 2 entries.

CBHT choosing the number of buckets

- The **load factor** of a hash table is the average number of entries per bucket, n/m.
- If n is (roughly) predictable, choose m such that the load factor is likely to be between 0.5 and 0.75.
  - A low load factor wastes space.
  - A high load factor tends to cause some buckets to have many entries.
- Choose m to be a prime number.
  - Typically, the hash function performs modulo-m arithmetic.
  - If m is prime, the entries are more likely to be distributed evenly over the buckets, regardless of any pattern in the keys.

CHBT choosing the hash function

- The hash function should be efficient (performing few simple and fast arithmetic operations).
- The hash function should distribute the entries evenly among the buckets, regardless of any patterns in the keys.
- Possible trade-off:
  - Speed up the hash function by using only part of the key.
  - But beware of collisions because of any repeating patterns in that part of the key.

Example: words hash table

- Suppose that a hash table will contain about 1000 common English words.
- Known patterns in the keys:
  - Letters vary in frequency:
    - A, E, I, N, S, T are common
    - Q, X, Z are uncommon.
  - Word lengths vary in frequency:
    - word lengths 4–8 are common
    - other word lengths are less common.

Example: words hash table

- Suppose that a hash table will contain about 1000 common English words.
- Known patterns in the keys:
  - Letters vary in frequency:
    - A, E, I, N, S, T are common
    - Q, X, Z are uncommon.
  - Word lengths vary in frequency:
    - word lengths 4–8 are common
    - other word lengths are less common.

How to choose the most appropriate values for m and the hash function?

Example: words hash table

- *hash*(*w*) can depend on any of *w*'s letters and/or length.
- Consider *m* = 20, *hash*(*w*) = length of *w* – 1.
  - Far too few buckets. Load factor = 1000/20 = 50.
  - Very uneven distribution.
- Consider *m* = 26, *hash*(*w*) = initial letter of *w* – 'A'.
  - Far too few buckets.
  - Very uneven distribution.

Example: words hash table

- Consider $m = 520$, $hash(w) = 26 \times$ (length of w – 1) + (initial letter of w – 'A').
  - – Too few buckets. Load factor = 1000/520 ≈ 1.9.
  - – Very uneven distribution. Since few words have length 0–2, buckets 0–51 will be sparsely populated. Since initial letter Z is uncommon, buckets 25, 51, 77, 103, … will be sparsely populated. And so on.
- Consider $m = 1499$, $hash(w)$ = (weighted sum of letters of w) modulo m

  i.e., ($c_1 \times$ 1st letter of w + $c_2 \times$ 2nd letter of w + …) modulo m
  - \+ Good number of buckets. Load factor ≈ 0.67.
  - \+ Reasonably even distribution.

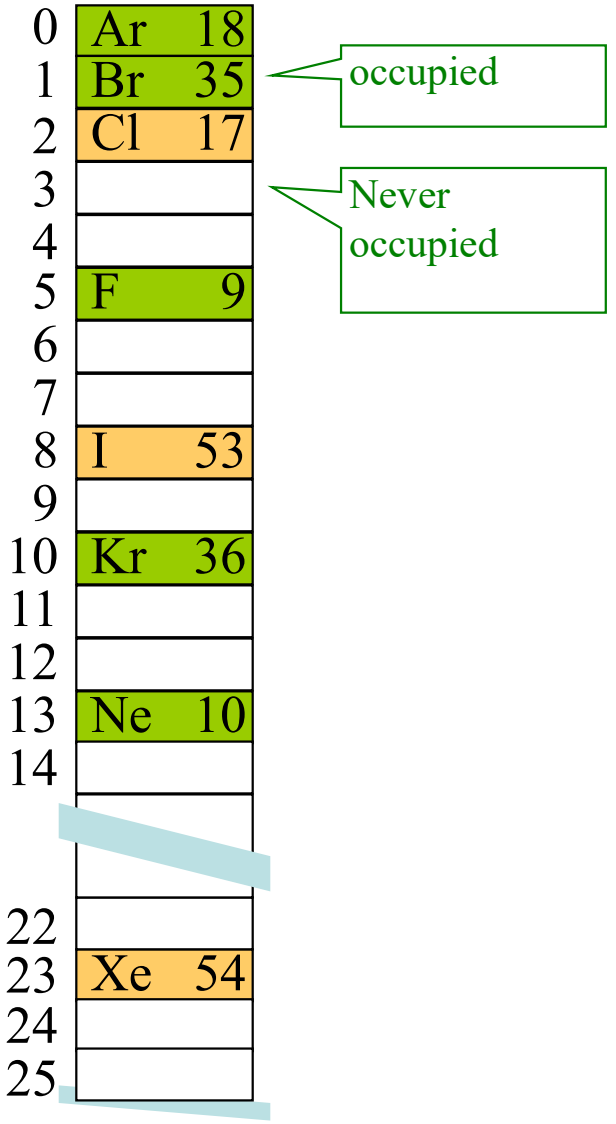
Open-bucket hash table

- **Open-bucket hash table** (OBHT):
  - Each bucket may be occupied by at most one entry.
  - Whenever there is a collision, displace the new entry to another bucket.
- Each bucket has three possible states:
  - **never-occupied** (has never contained an entry)
  - **occupied** (currently contains an entry)
  - **formerly-occupied** (previously contained an entry, which has been deleted and not yet replaced).

Open-bucket hash table

- Illustration (no collisions):

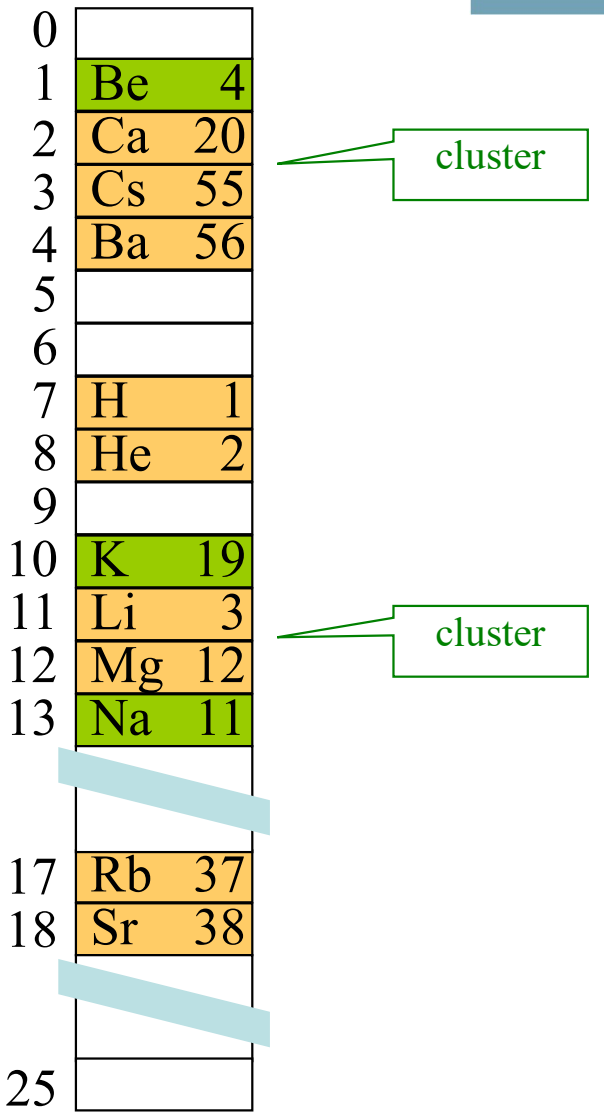
element	number
F	9
Ne	10
Cl	17
Ar	18
Br	35
Kr	36
I	53
Xe	54



Open-bucket hash table

- Illustration (with collisions):

element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

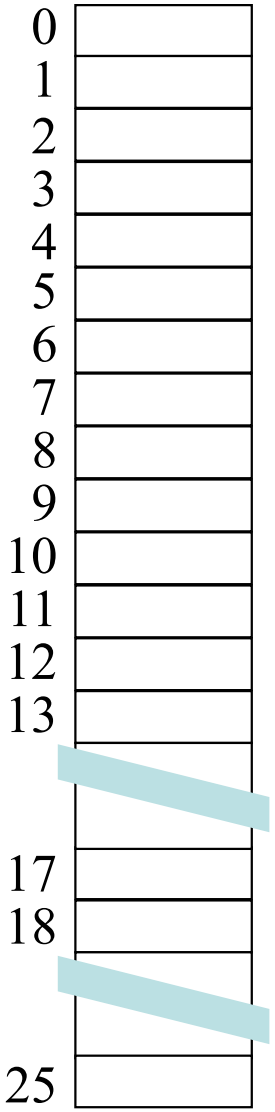
- Animation:

element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56

Populating an OBHT

- Animation:

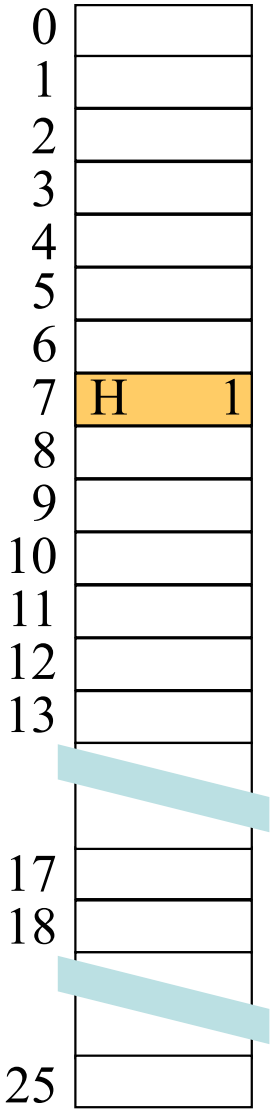
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

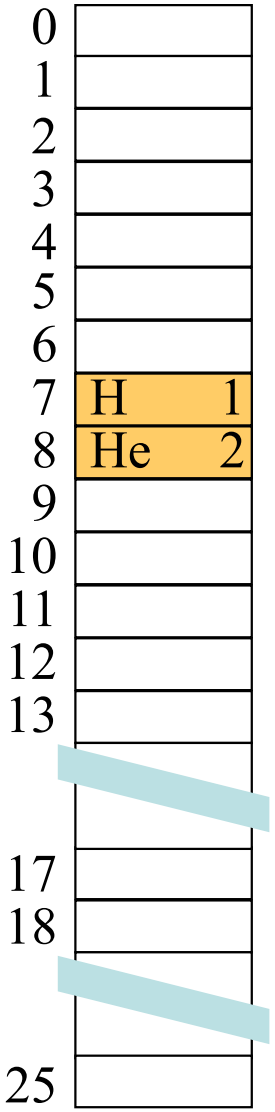
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

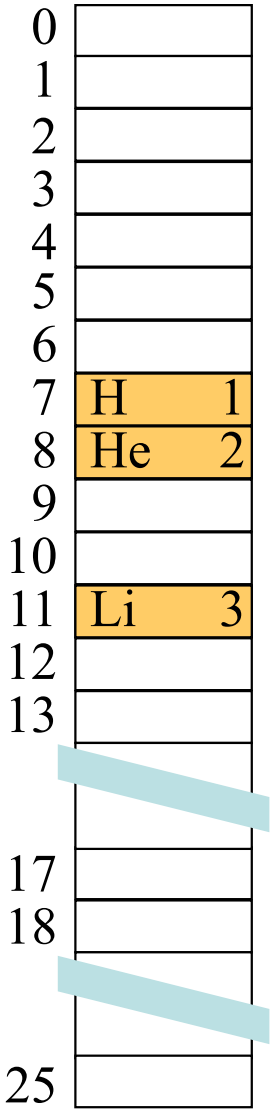
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

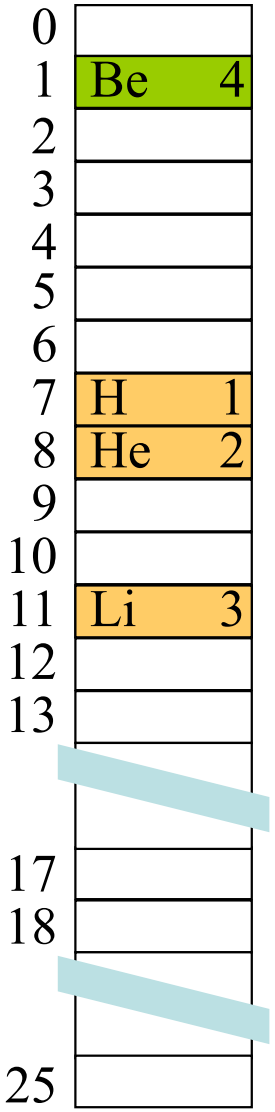
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

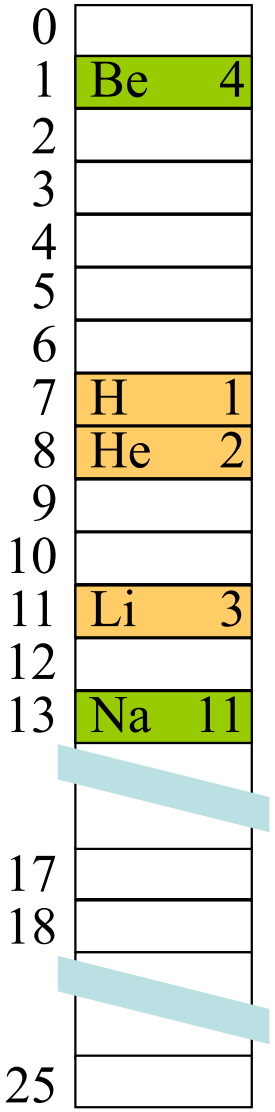
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

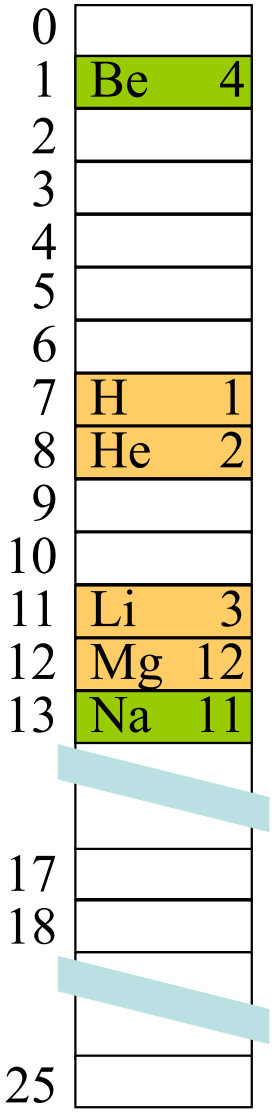
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

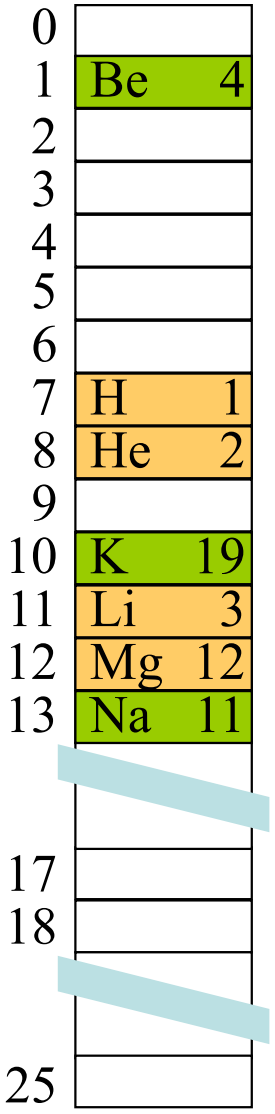
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

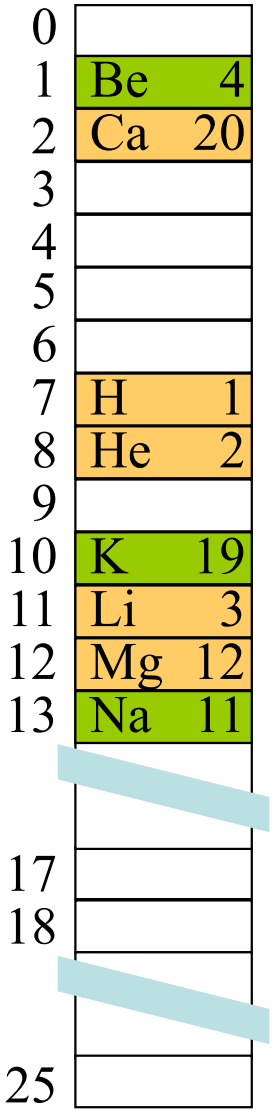
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

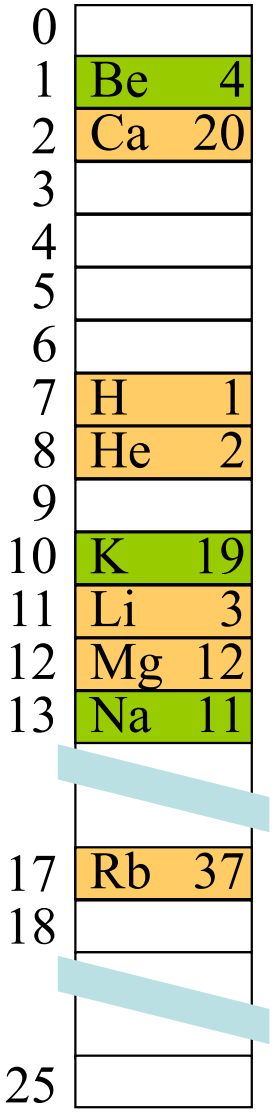
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

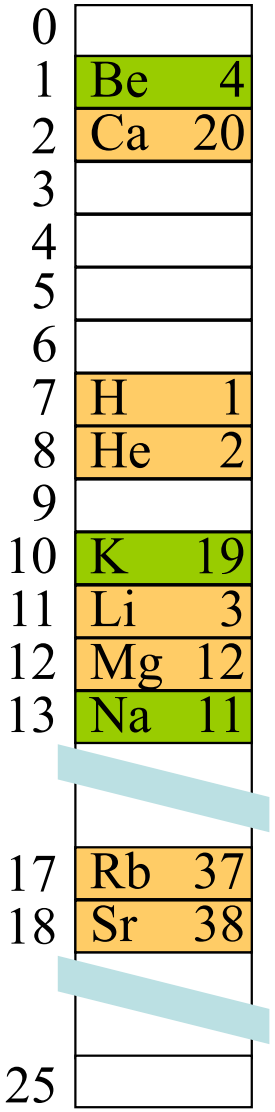
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

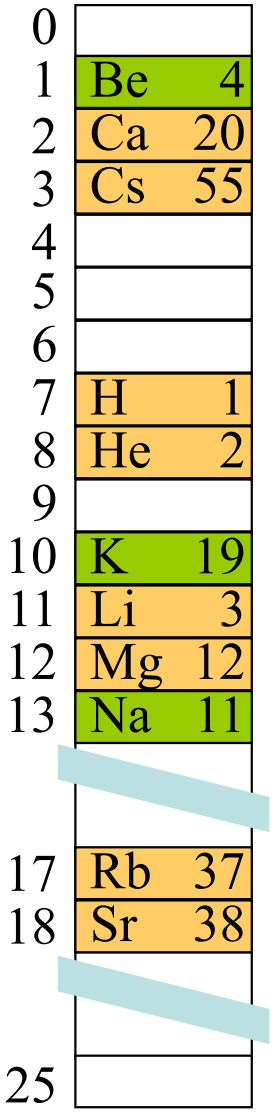
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

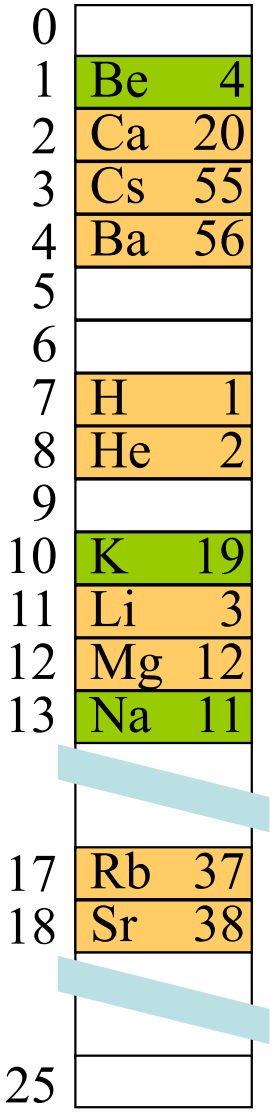
element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



Populating an OBHT

- Animation:

element	number
H	1
He	2
Li	3
Be	4
Na	11
Mg	12
K	19
Ca	20
Rb	37
Sr	38
Cs	55
Ba	56



OBHT search

- To find which if any bucket of an OBHT is occupied by an entry whose key is equal to *target-key*:

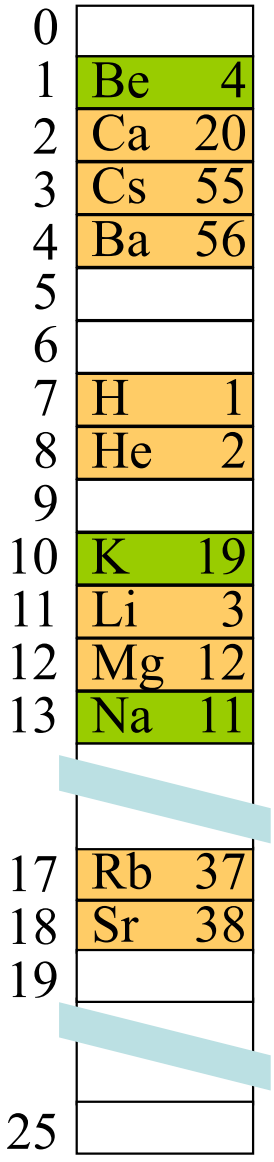
1. Set *b* to *hash*(*target-key*).
2. Repeat:
    2.1. If bucket *b* is never-occupied:
        2.1.1. Terminate with answer *none*.
    2.2. If bucket *b* is occupied by an entry whose key is equal to *target-key*:
        2.2.1. Terminate with answer *b*.
    2.3. If bucket *b* is formerly-occupied, or is occupied by an entry whose key is not equal to *target-key*:
        2.3.1. Increment *b* modulo *m*.

OBHT search

- Illustration:

OBHT search

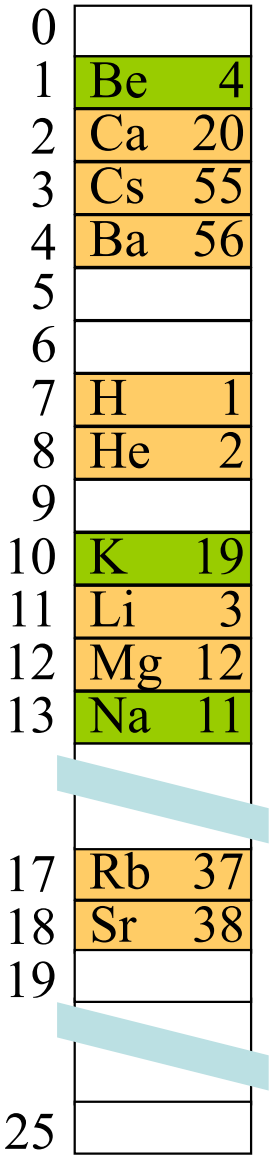
- Illustration:



OBHT search

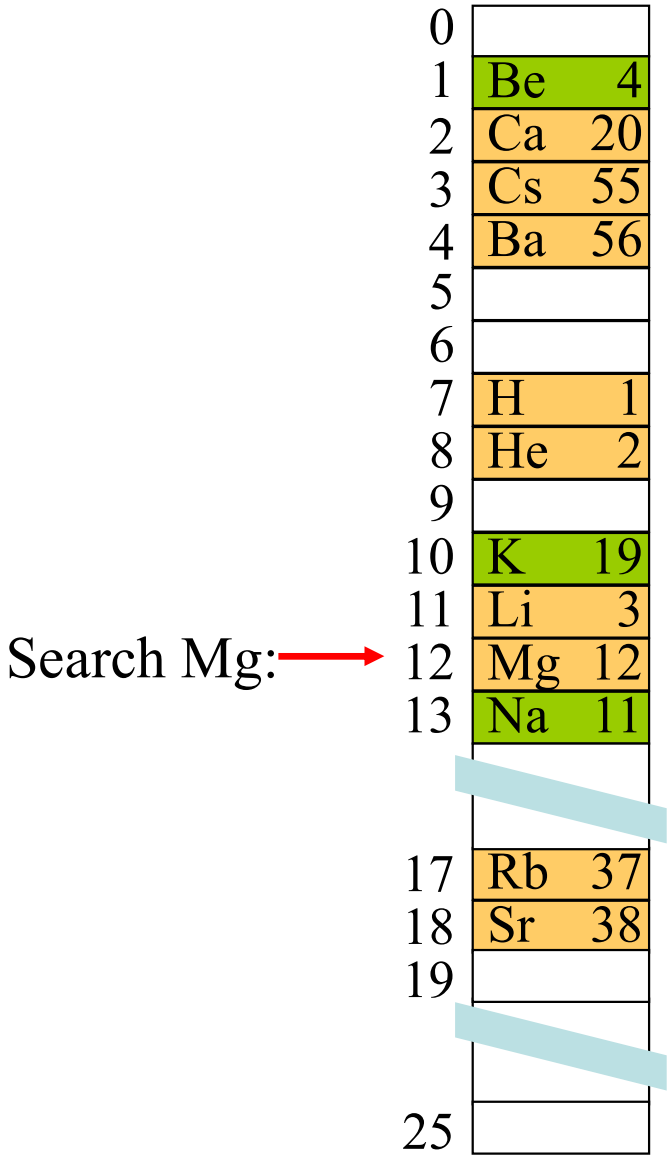
- Illustration:

Search Mg:



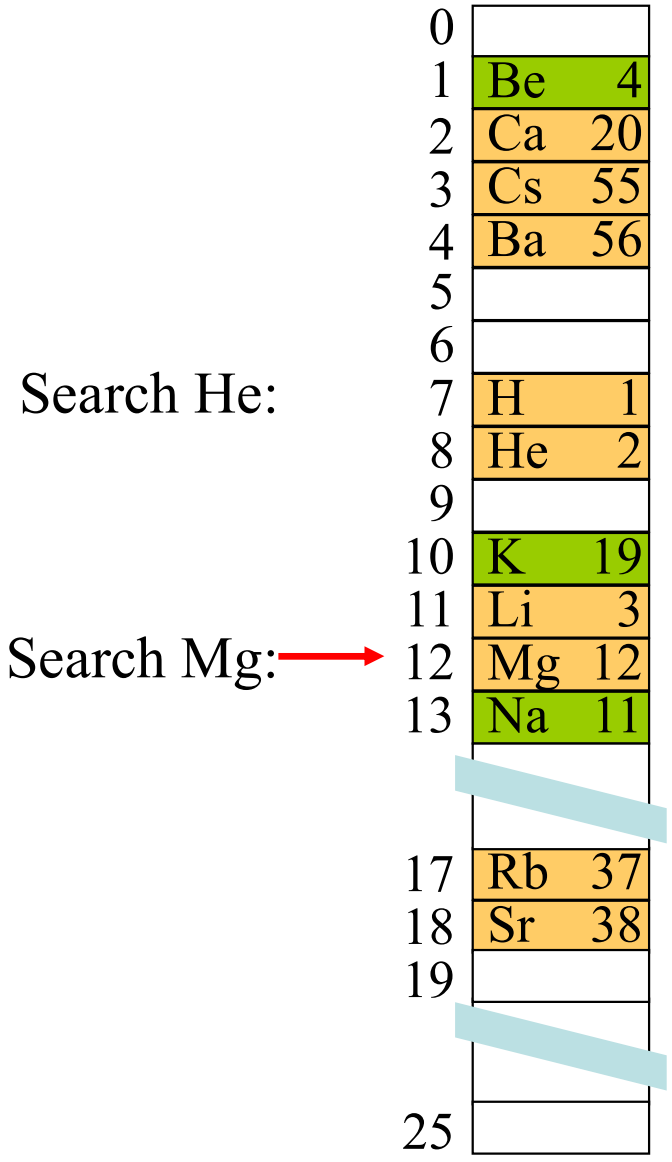
OBHT search

- Illustration:



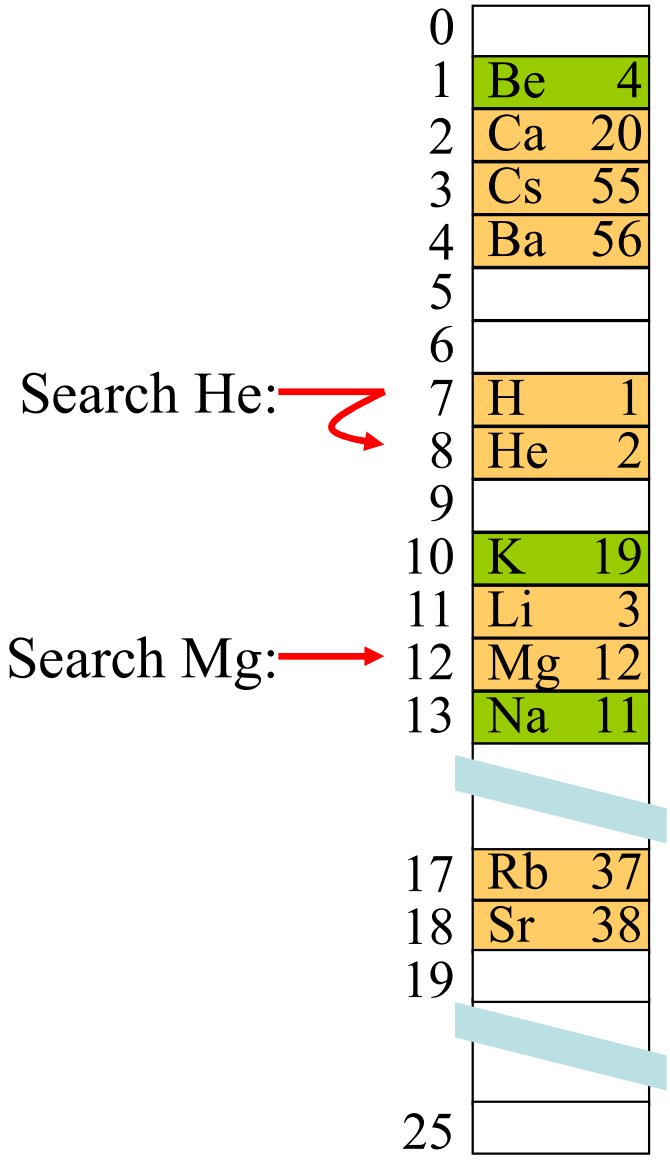
OBHT search

- Illustration:



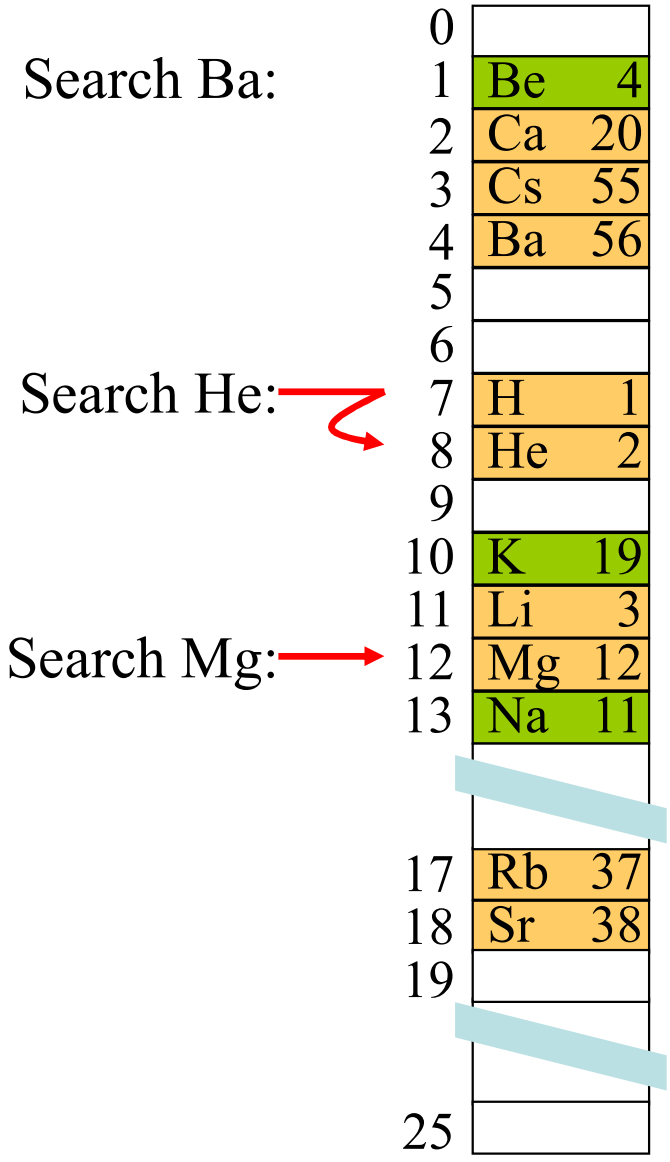
OBHT search

- Illustration:



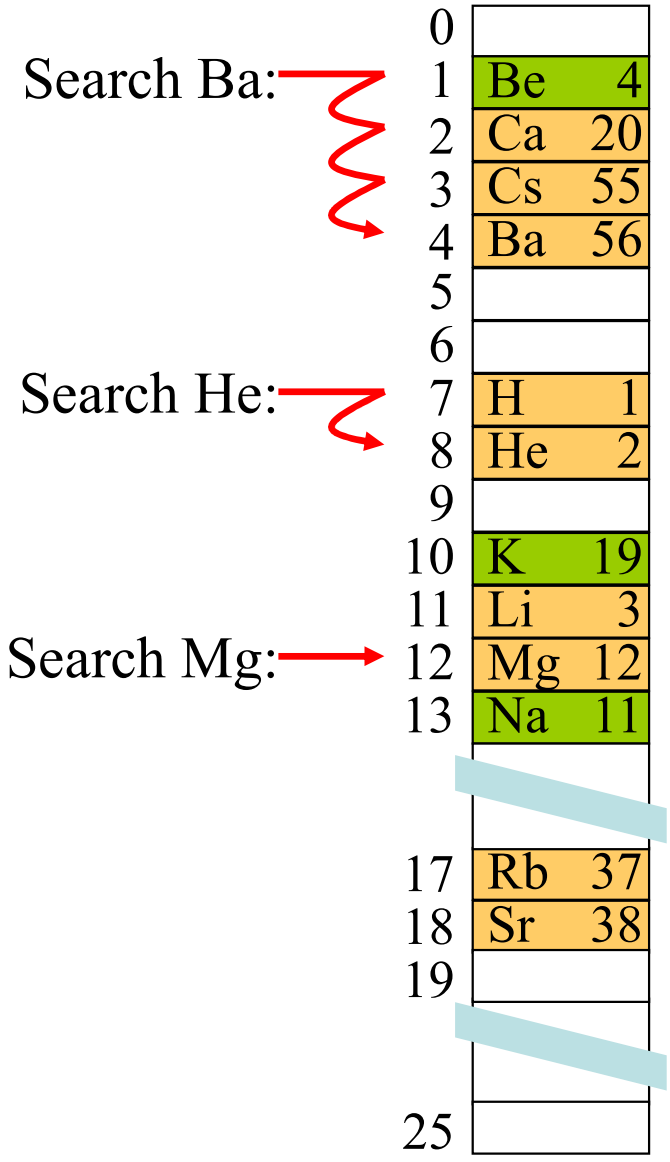
OBHT search

- Illustration:



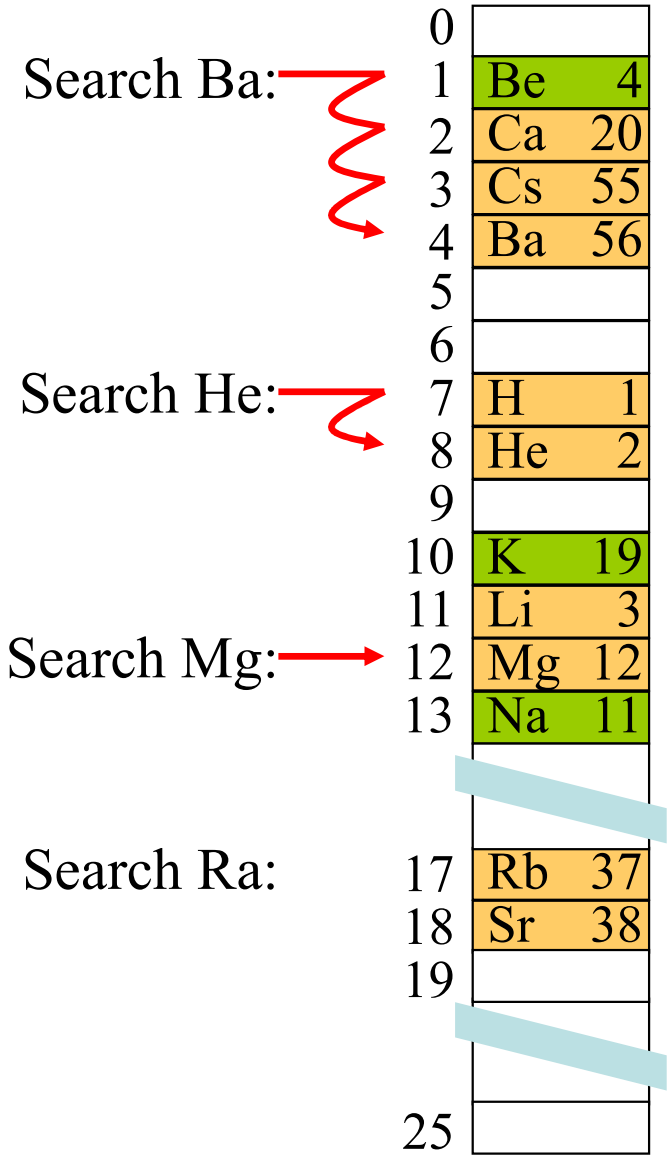
OBHT search

- Illustration:



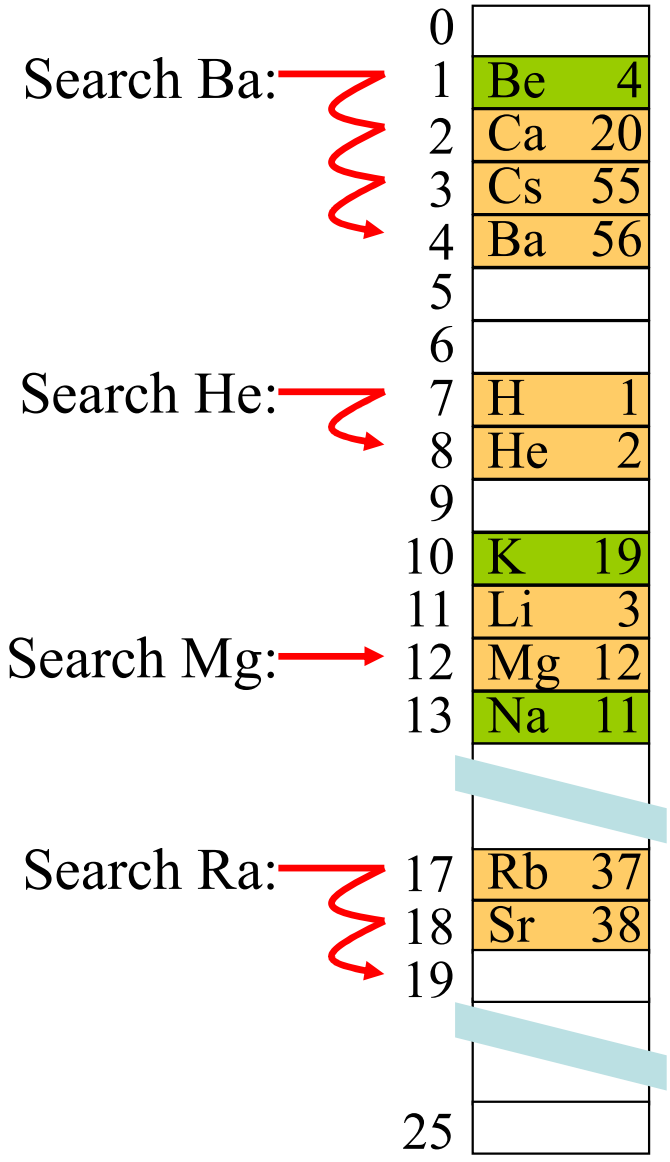
OBHT search

- Illustration:



OBHT search

- Illustration:



OBHT insertion

- To insert the entry (*key*, *val*) into an OBHT:

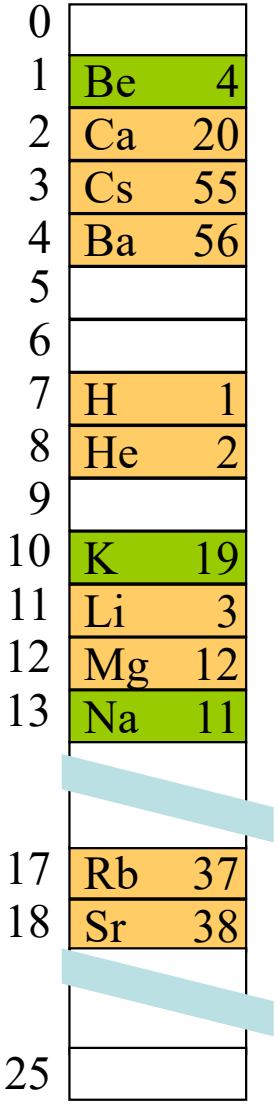
1. Set *b* to *hash*(*key*).
2. Repeat:
    2.1. If bucket *b* is never-occupied:
        2.1.1. If bucket *b* is the last never-occupied bucket, treat the OBHT as full.
        2.1.2. Make bucket *b* occupied by (*key*, *val*).
        2.1.3. Terminate.
    2.2. If bucket *b* is formerly-occupied, or is occupied by an entry whose key is equal to *key*:
        2.2.1. Make bucket *b* occupied by (*key*, *val*).
        2.2.2. Terminate.
    2.3. If bucket *b* is occupied by an entry whose key is not equal to *key*:
        2.3.1. Increment *b* modulo *m*.

OBHT insertion

- Illustration:

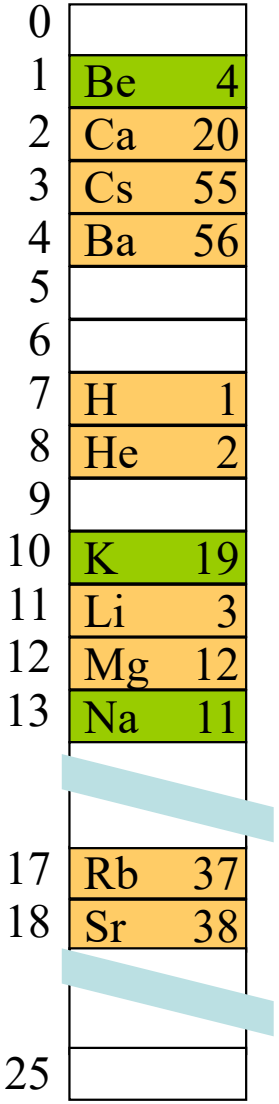
OBHT insertion

- Illustration:



OBHT insertion

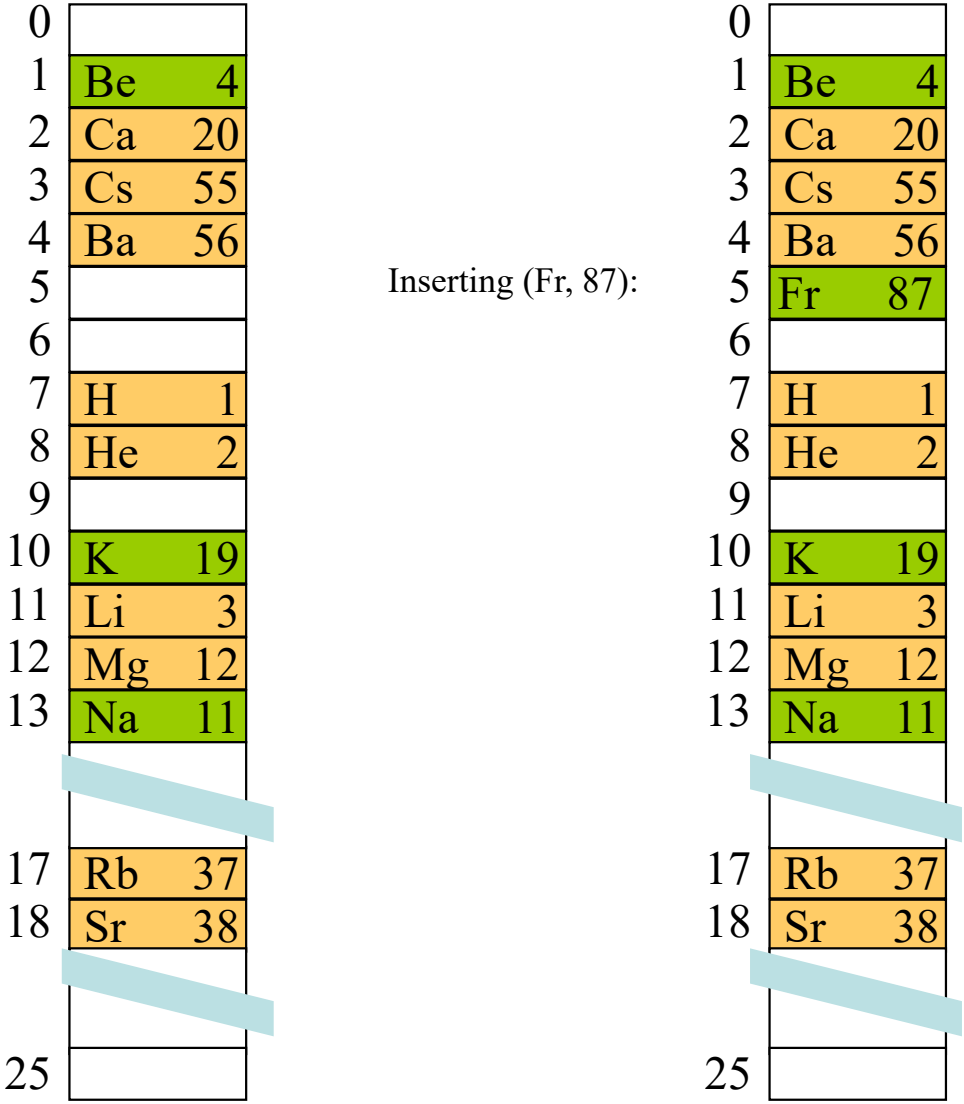
- Illustration :



Inserting (Fr, 87):

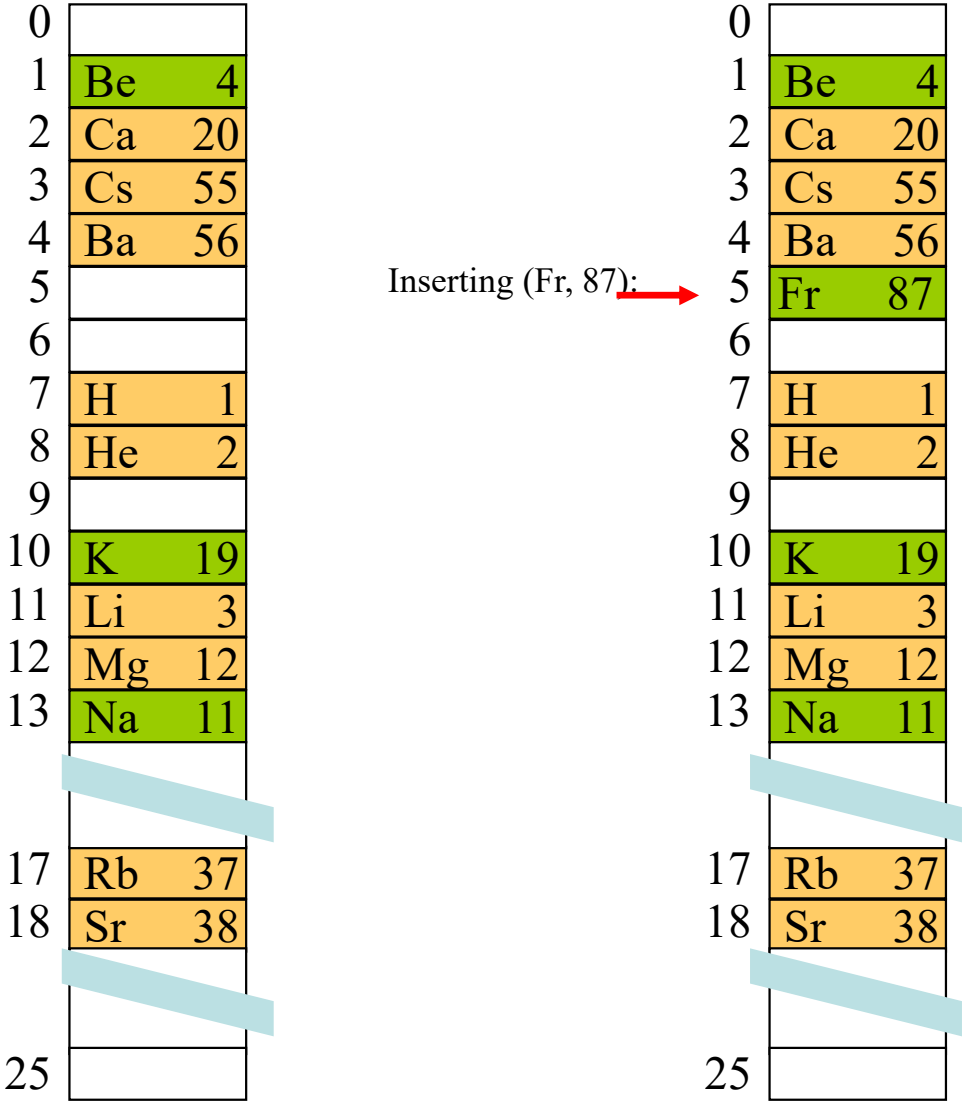
OBHT insertion

- Illustration:



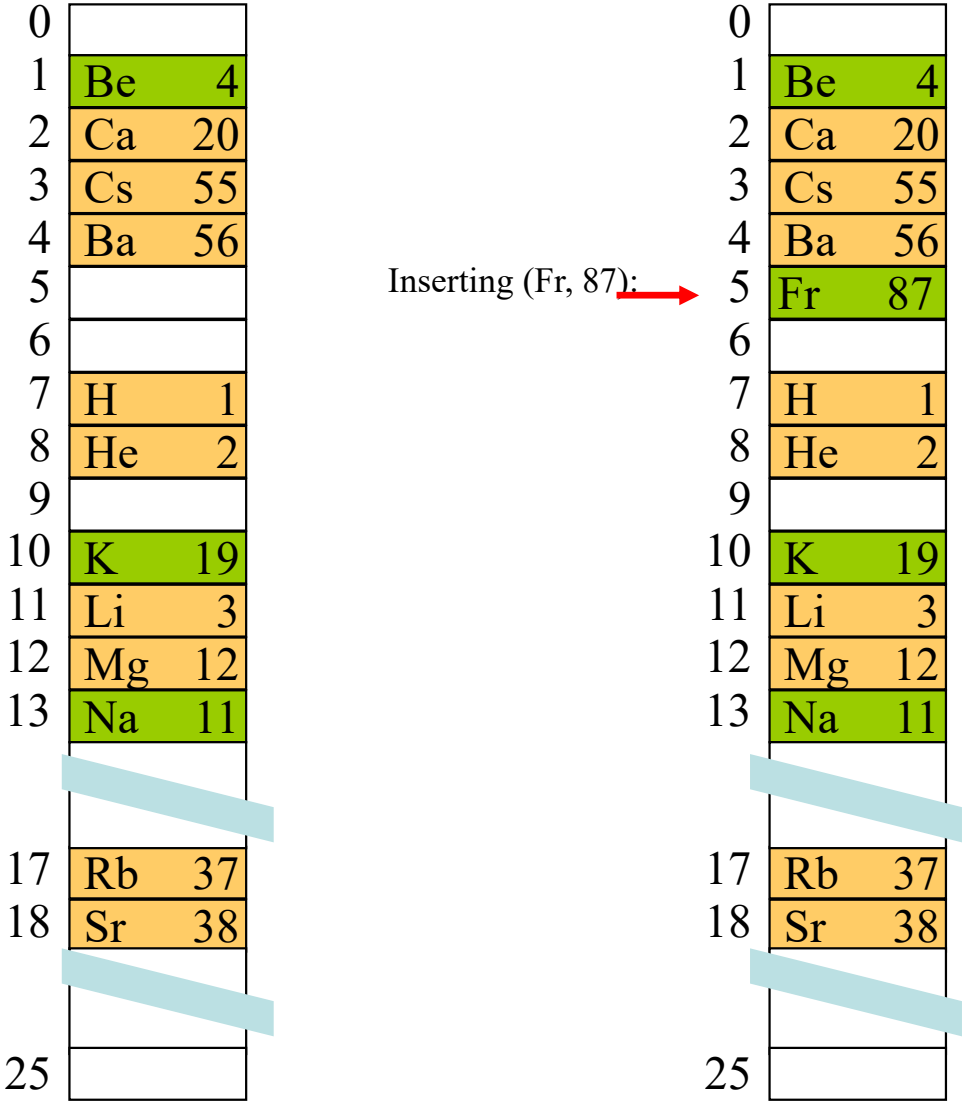
OBHT insertion

- Illustration:



OBHT insertion

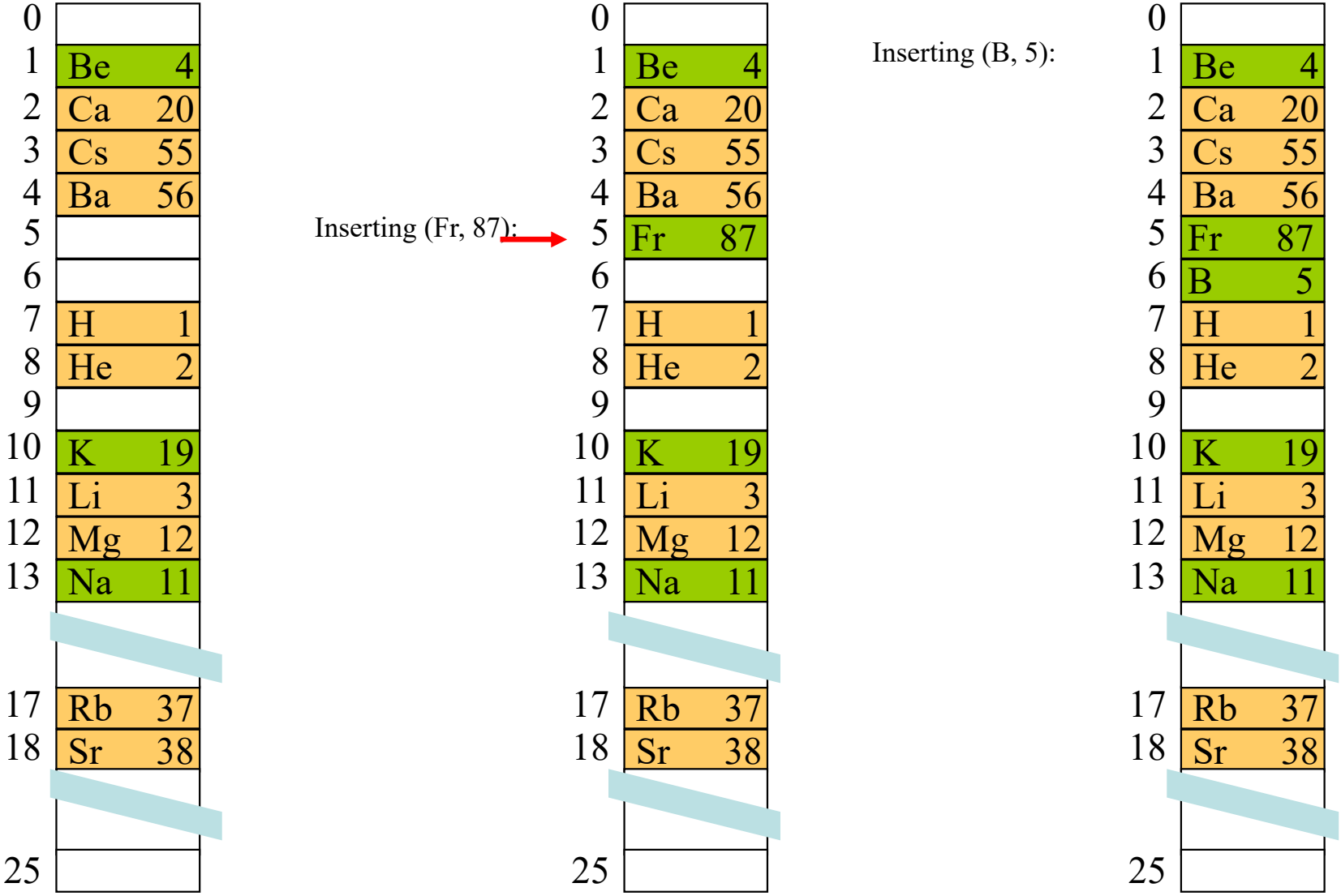
- Illustration:



Inserting (B, 5):

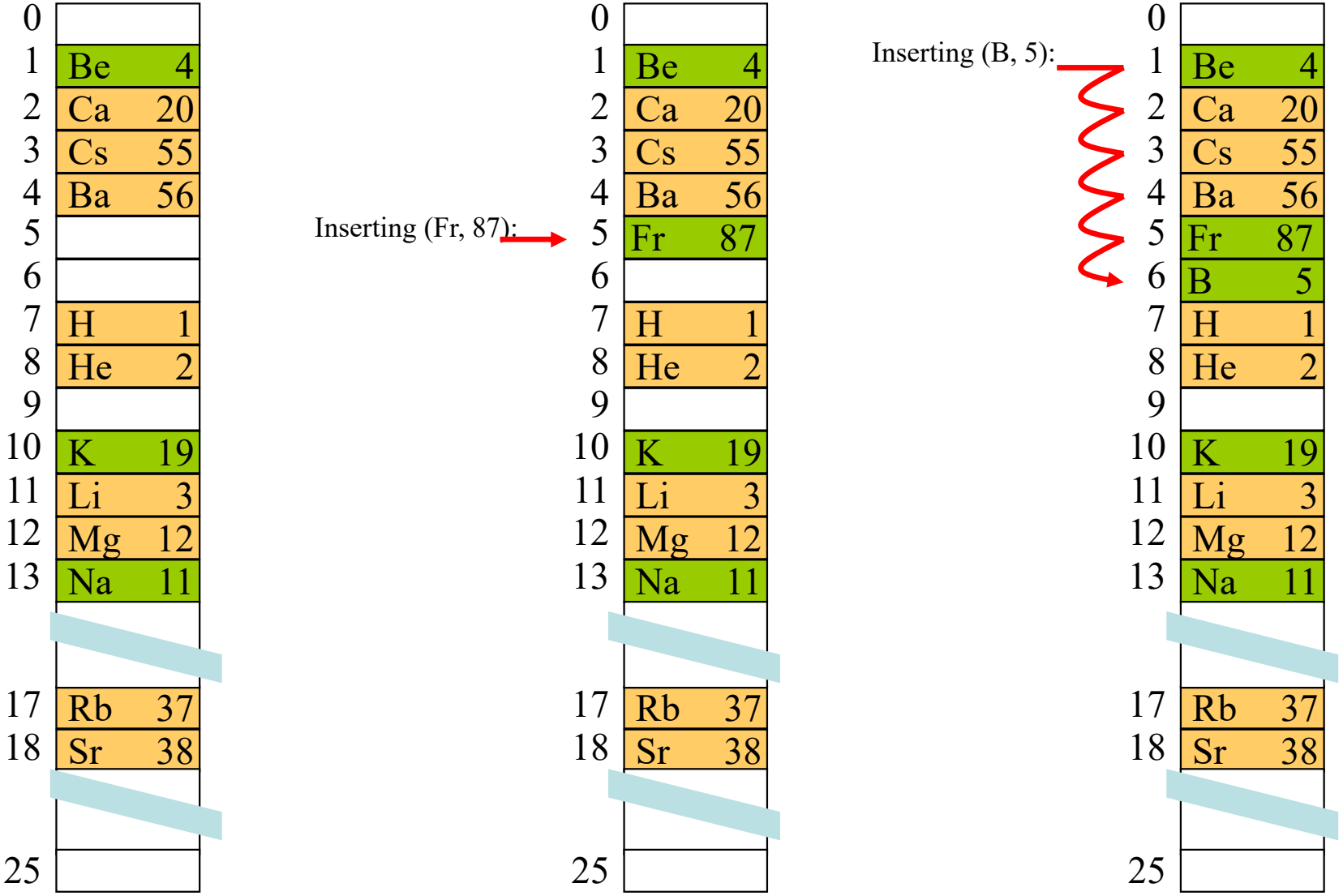
OBHT insertion

- Illustration:



OBHT insertion

- Illustration:



OBHT deletion

- To delete the entry (if any) whose key is equal to *key* from an OBHT:

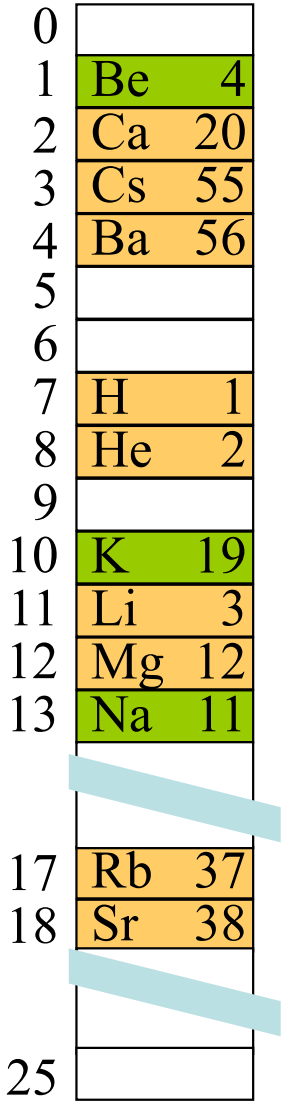
1. Set *b* to *hash*(*key*).
2. Repeat:
    2.1. If bucket *b* is never-occupied:
        2.1.1. Terminate.
    2.2. If bucket *b* is occupied by an entry whose key is equal to *key*:
        2.2.1. Make bucket *b* formerly-occupied.
        2.2.2. Terminate.
    2.3. If bucket *b* is formerly-occupied, or is occupied by an entry whose key is not equal to *key*:
        2.3.1. Increment *b* modulo *m*.

OBHT deletion

- Illustration:

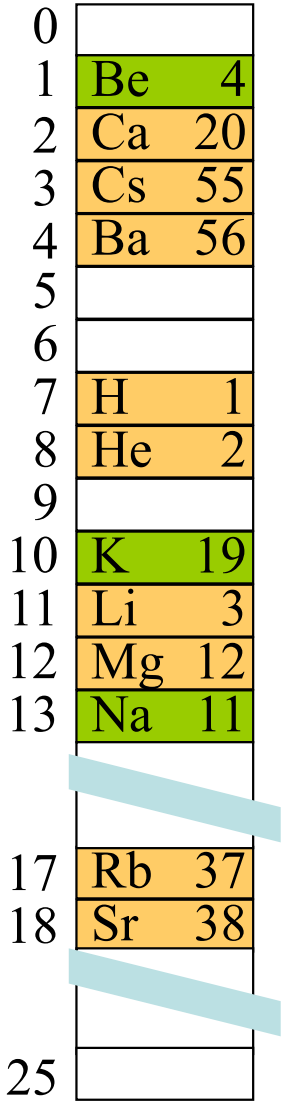
OBHT deletion

- Illustration:



OBHT deletion

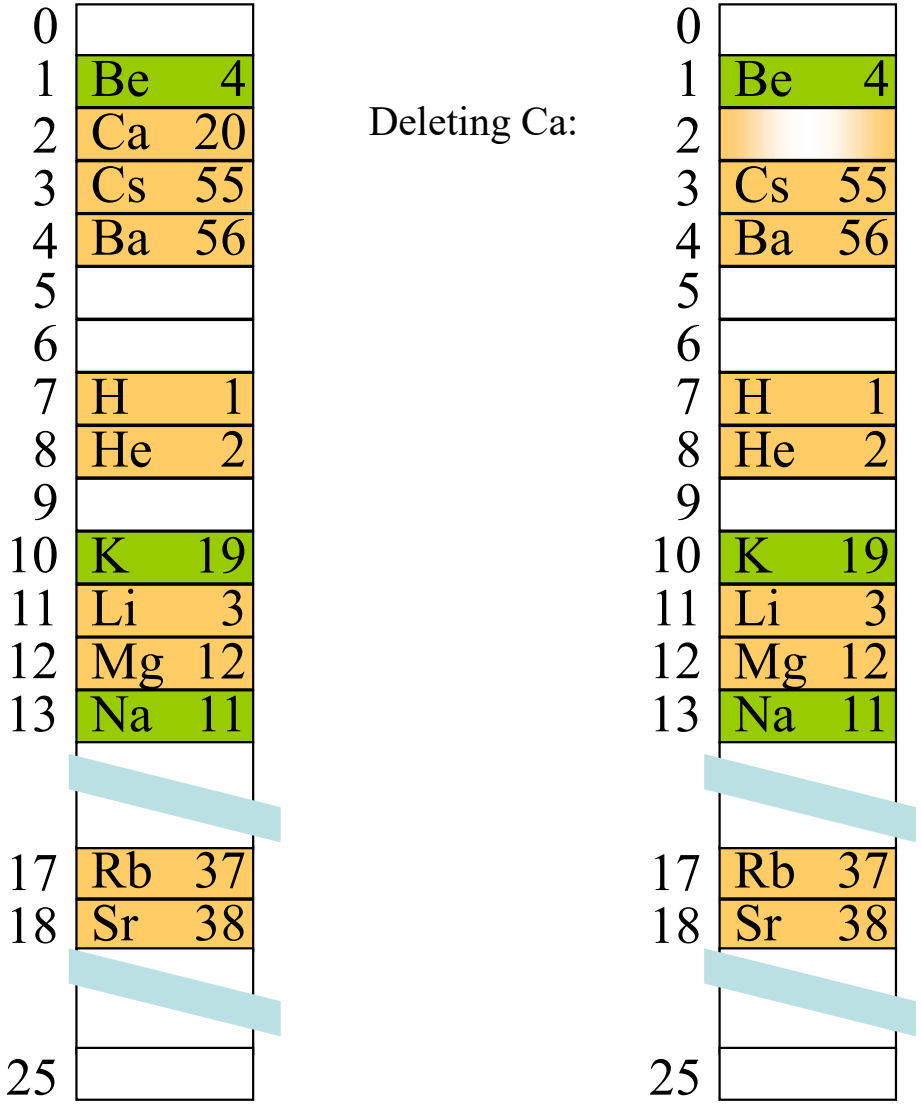
- Illustration:



Deleting Ca:

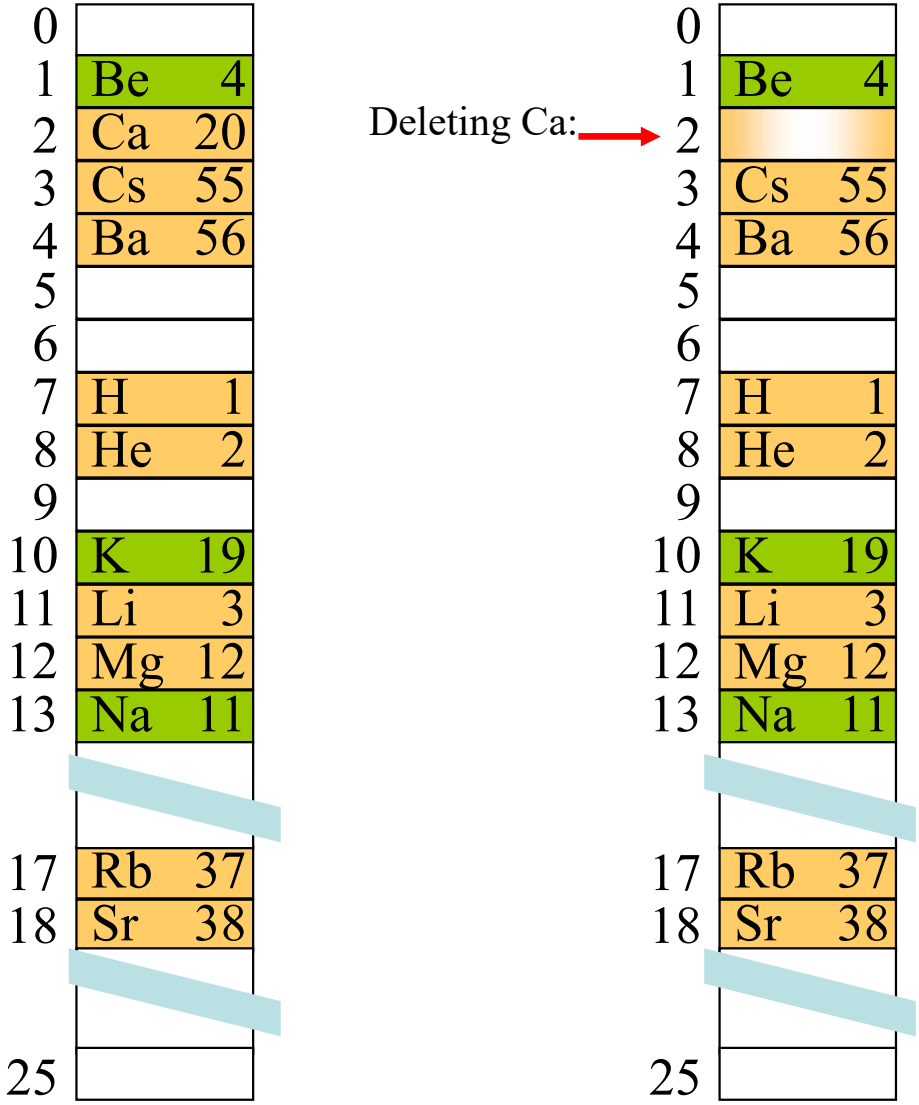
OBHT deletion

- Illustration:



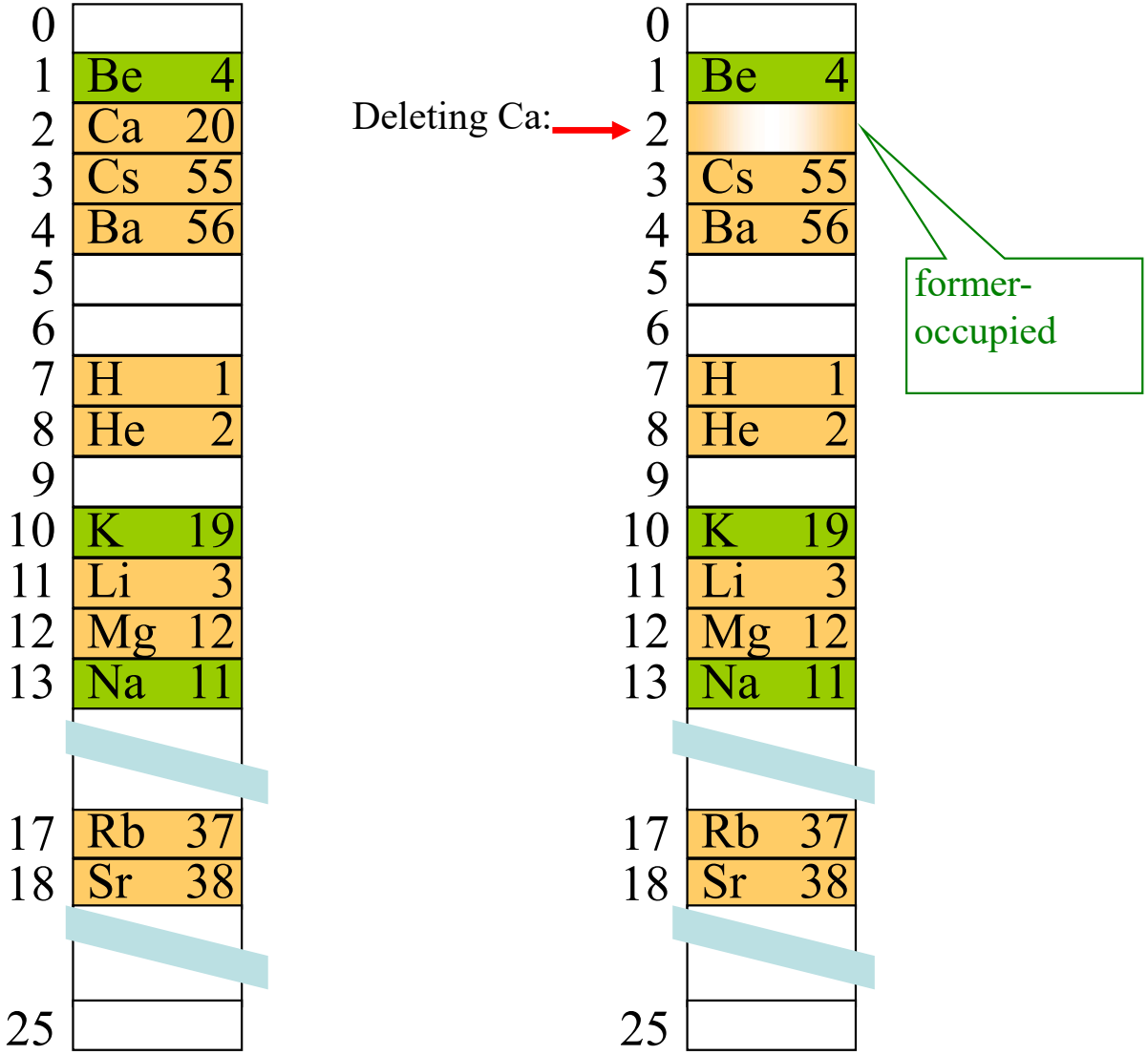
OBHT deletion

- Illustration:



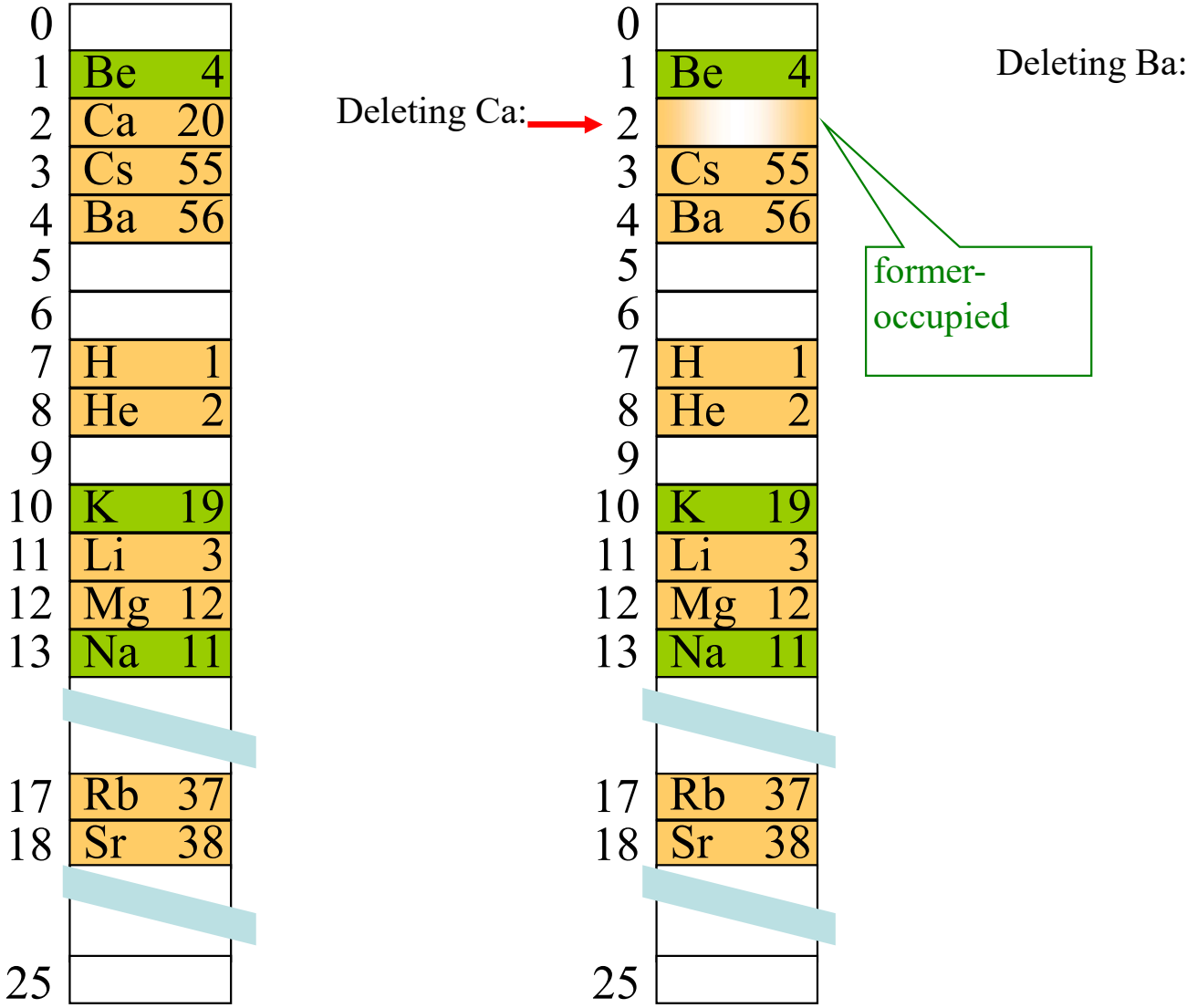
OBHT deletion

- Illustration:



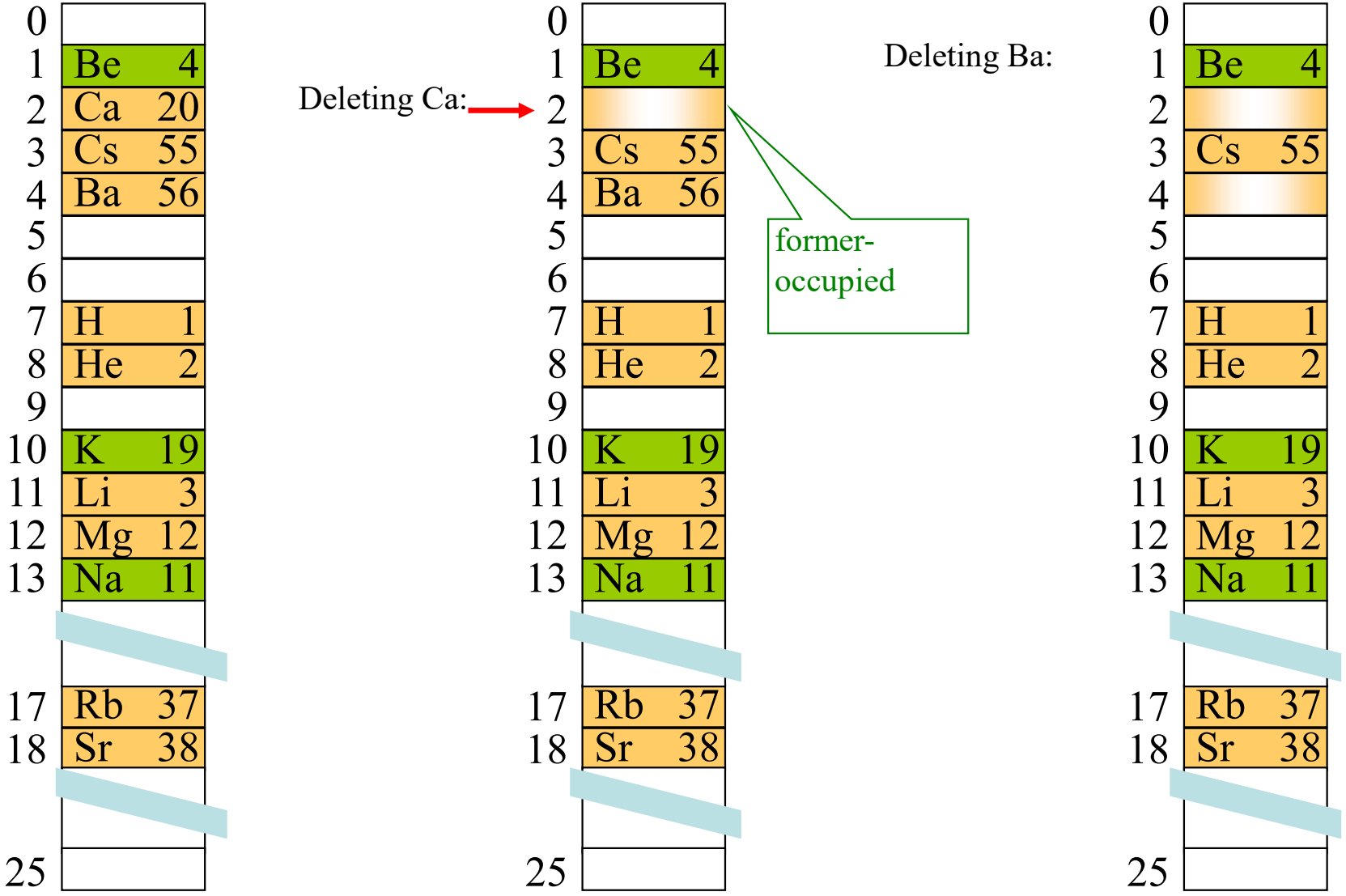
OBHT deletion

- Illustration:



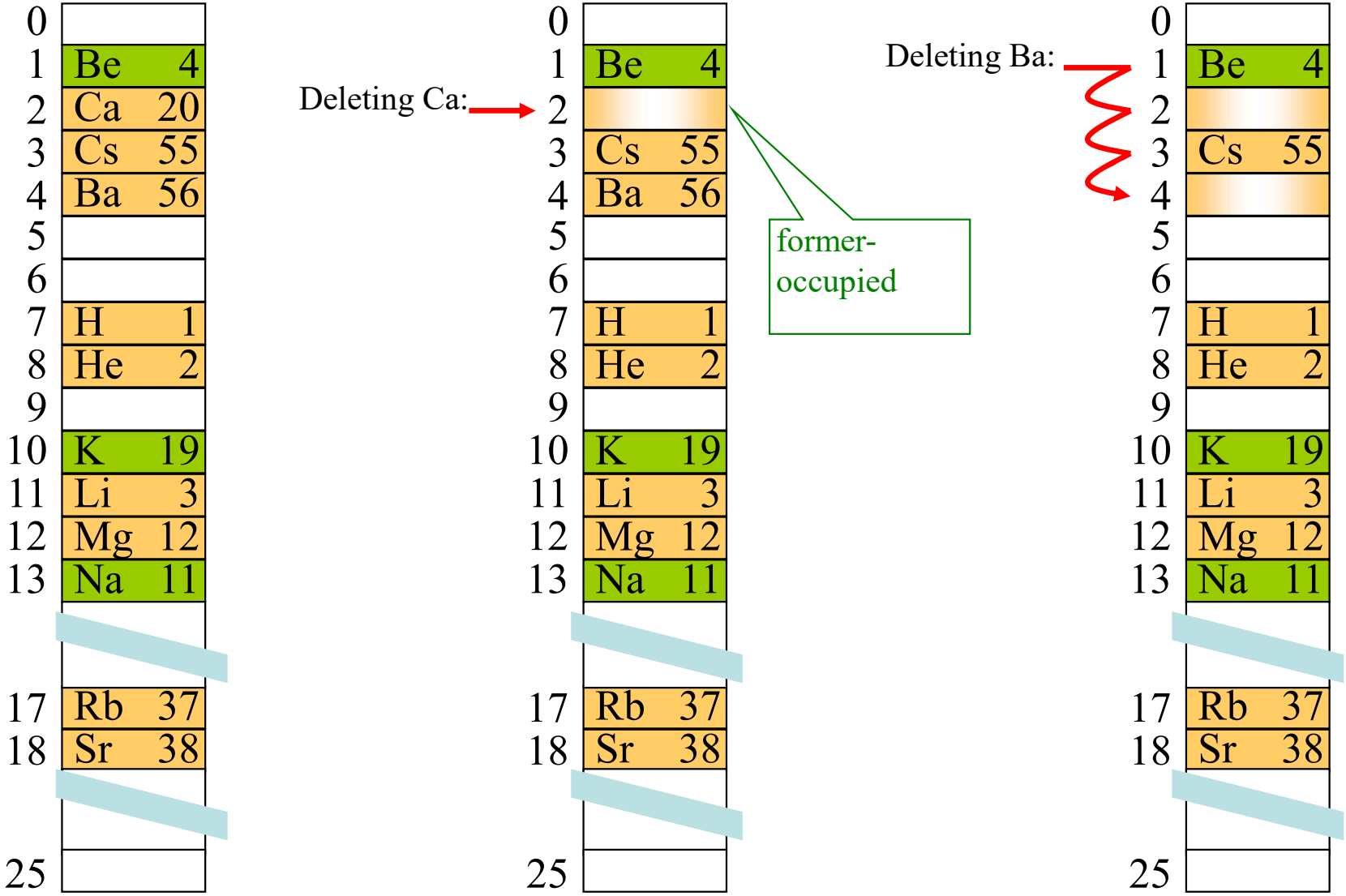
OBHT deletion

- Illustration:



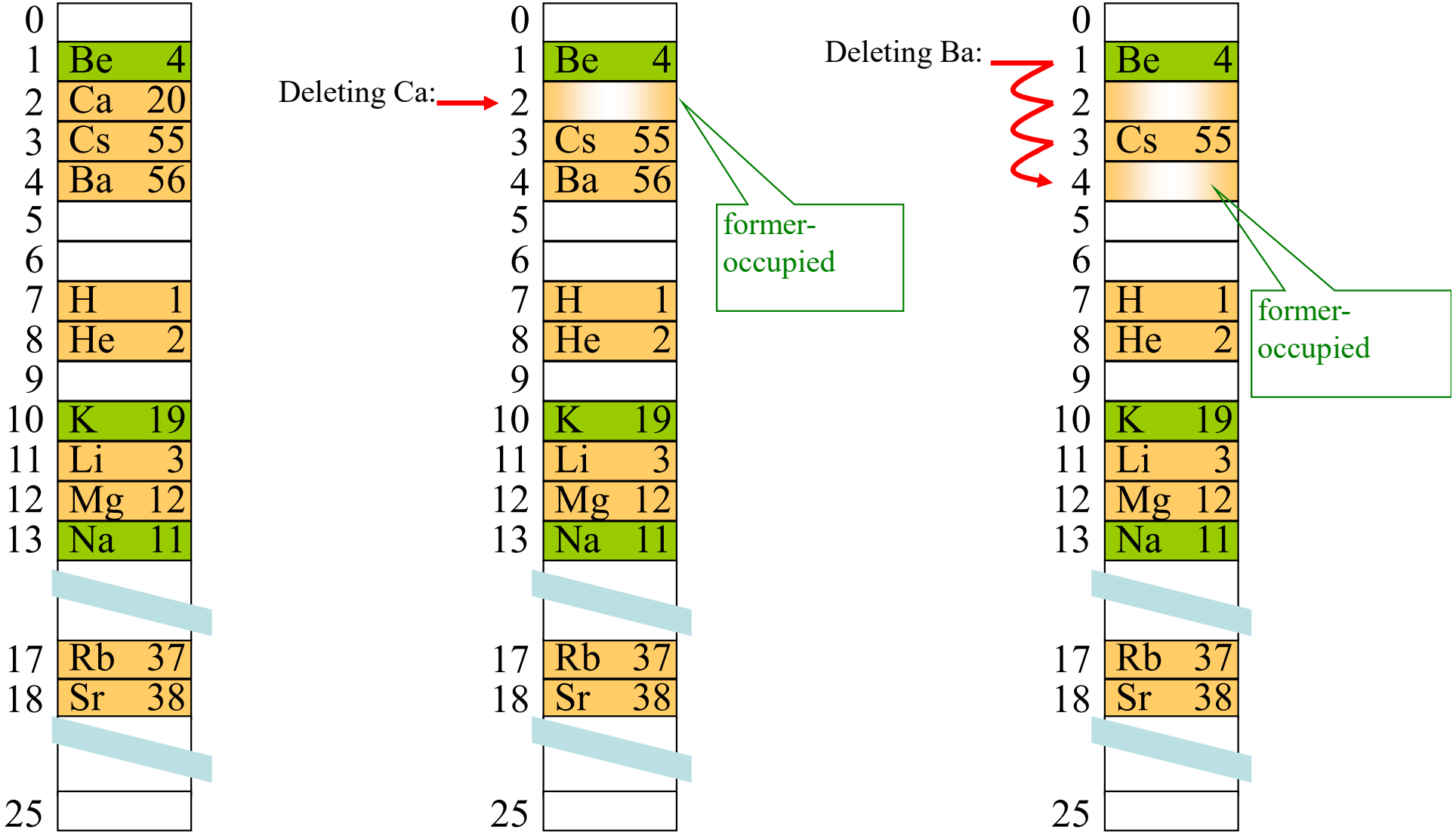
OBHT deletion

- Illustration:



OBHT deletion

- Illustration:



OBHT analysis

- Analysis of OBHT search/insertion/deletion algorithm (counting comparisons):

  Let the number of entries be *n*.
- In the **best case**, no cluster contains more than (say) 4 entries:

  Max. no. of comparisons = 4

  Best-case time complexity is $O(1)$.
- In the **worst case**, one cluster contains all *n* entries:

  Max. no. of comparisons = *n*

  Worst-case time complexity is $O(n)$.

OBHT design

- OBHT design consists of:
  - choosing the number of buckets *m*
  - choosing the hash function *hash*
  - choosing the step length *s* (explained later).
- Design aims:
  - collisions should be infrequent
  - entries should be distributed evenly over the hash table, such that few clusters contain more than about 4 entries.

OBHT: choosing the number of buckets

- Recall: The **load factor** of a hash table is the average number of entries per bucket, n/m.
- If n is (roughly) predictable, choose m such that the load factor is likely to be between 0.5 and 0.75.
  - A low load factor wastes space.
  - A high load factor tends to result in long clusters.
- Choose m to be a prime number.

OBHT: choosing the hash function

- The hash function should be efficient.
- The hash function should distribute the entries evenly over the buckets, with few long clusters.
  - In an OHBT with $s = 1$, a cluster will form when several entries fall into the same **or adjacent** buckets.

OBHT: choosing the step length

- To resolve a collision, the search/insertion/deletion algorithm increments the bucket index and tries again.
- The **step length**, *s*, is the amount by which the bucket index is incremented.

OBHT: choosing the step length

- So far we have assumed $s = 1$.
- Alternatively, we can use a fixed $s > 1$.
- Choose m to be prime, and choose s to be in the range $2 \ldots m-1$.
  - This ensures that s and m have no common factors.
  - Otherwise, if (say) $m = 10$ and $s = 2$, a typical search path would be 6–8–0–2–4, never reaching the remaining buckets!

OBHT: double hashing

- Better still, let different keys have different step lengths. (But each key always has the same step length.)
- **Double hashing**: To search/insert/delete key *k*, compute *s* from *k*, using a second hash function *s* = *step*(*k*).
- In the following illustration, keys are names of chemical elements. Assume:

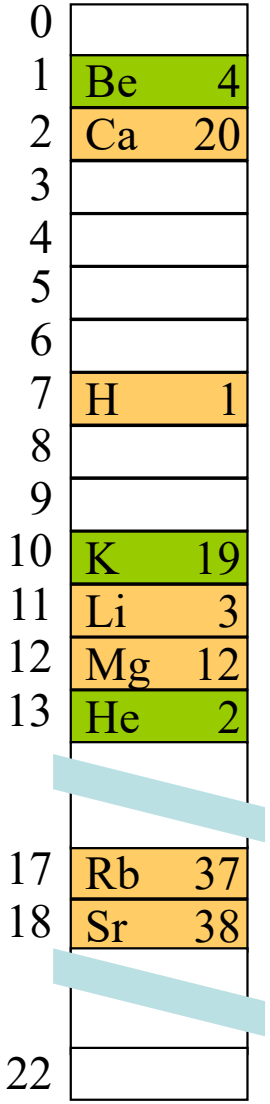
m	=	23
hash(*e*)	=	(initial letter of *e* – ‘A’) modulo *m*
step(*e*)	=	1, if *e* has a single letter, otherwise 2 + (second letter of *e* – ‘a’) modulo 21

Double hashing

- Illustration:

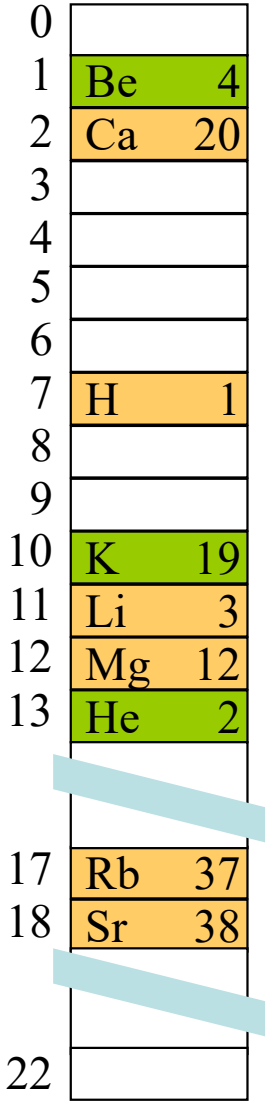
Double hashing

- Illustration:



Double hashing

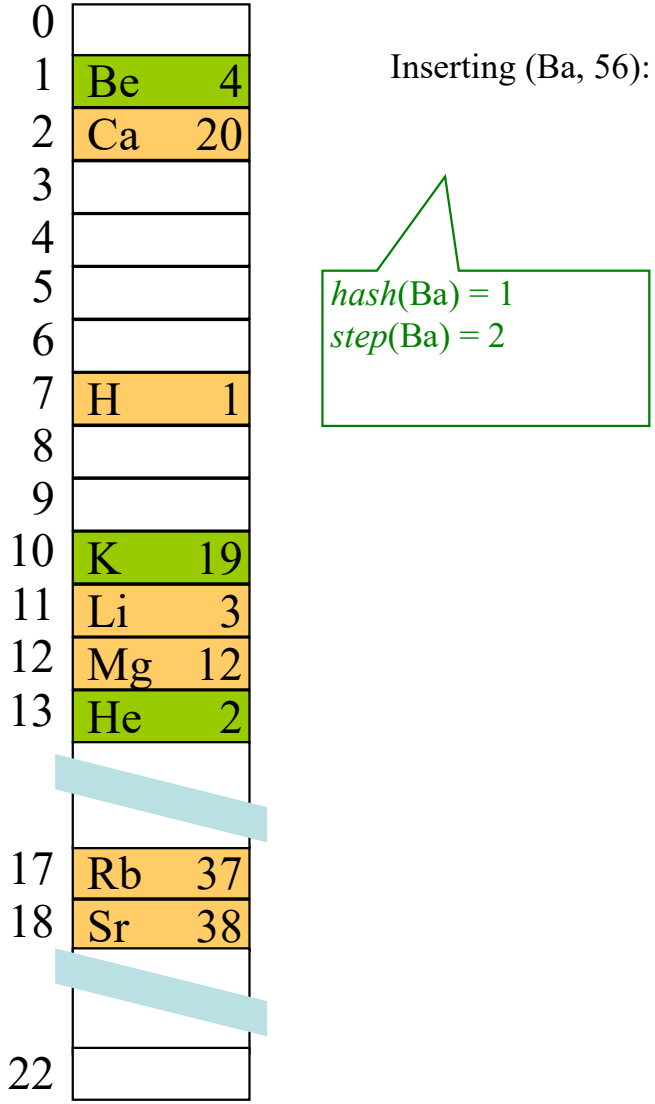
- Illustration:



Inserting (Ba, 56):

Double hashing

- Illustration:

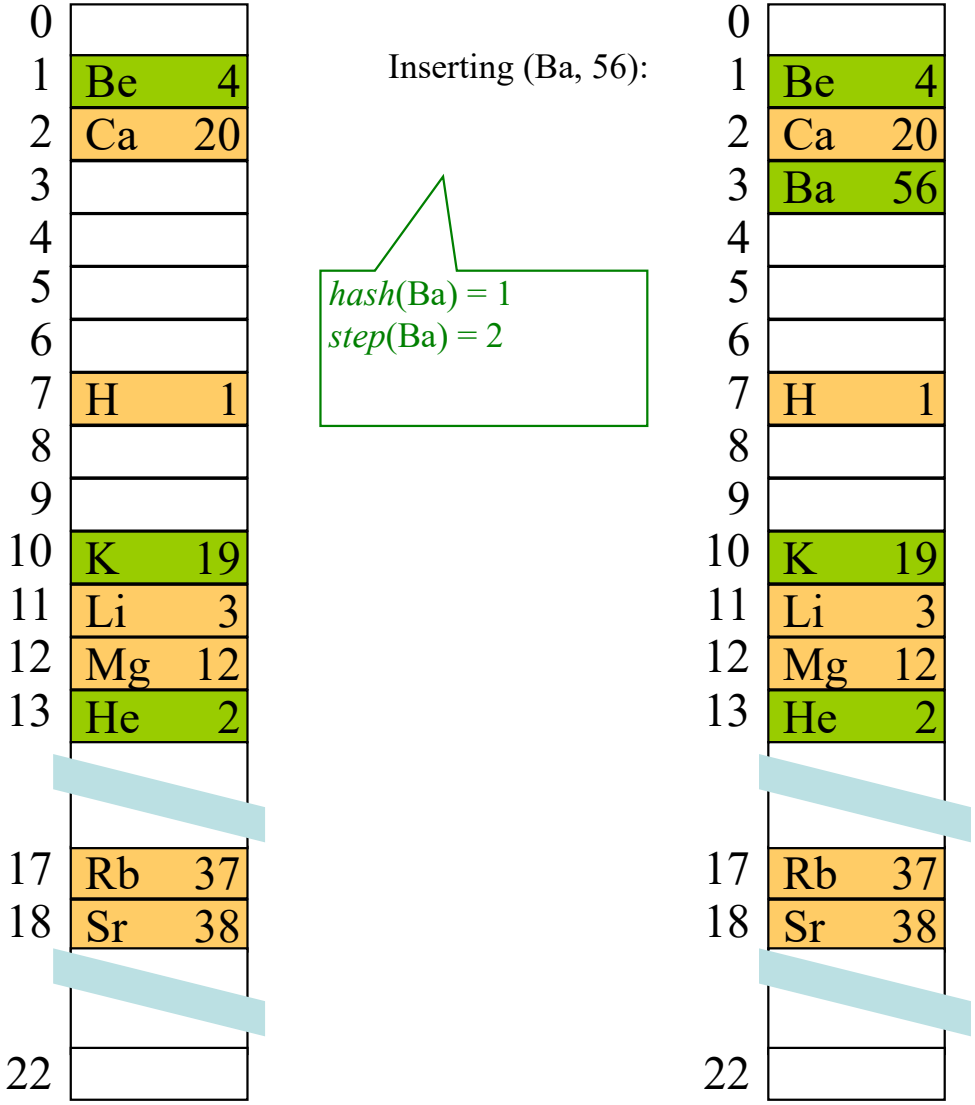


Inserting (Ba, 56):

hash(Ba) = 1
step(Ba) = 2

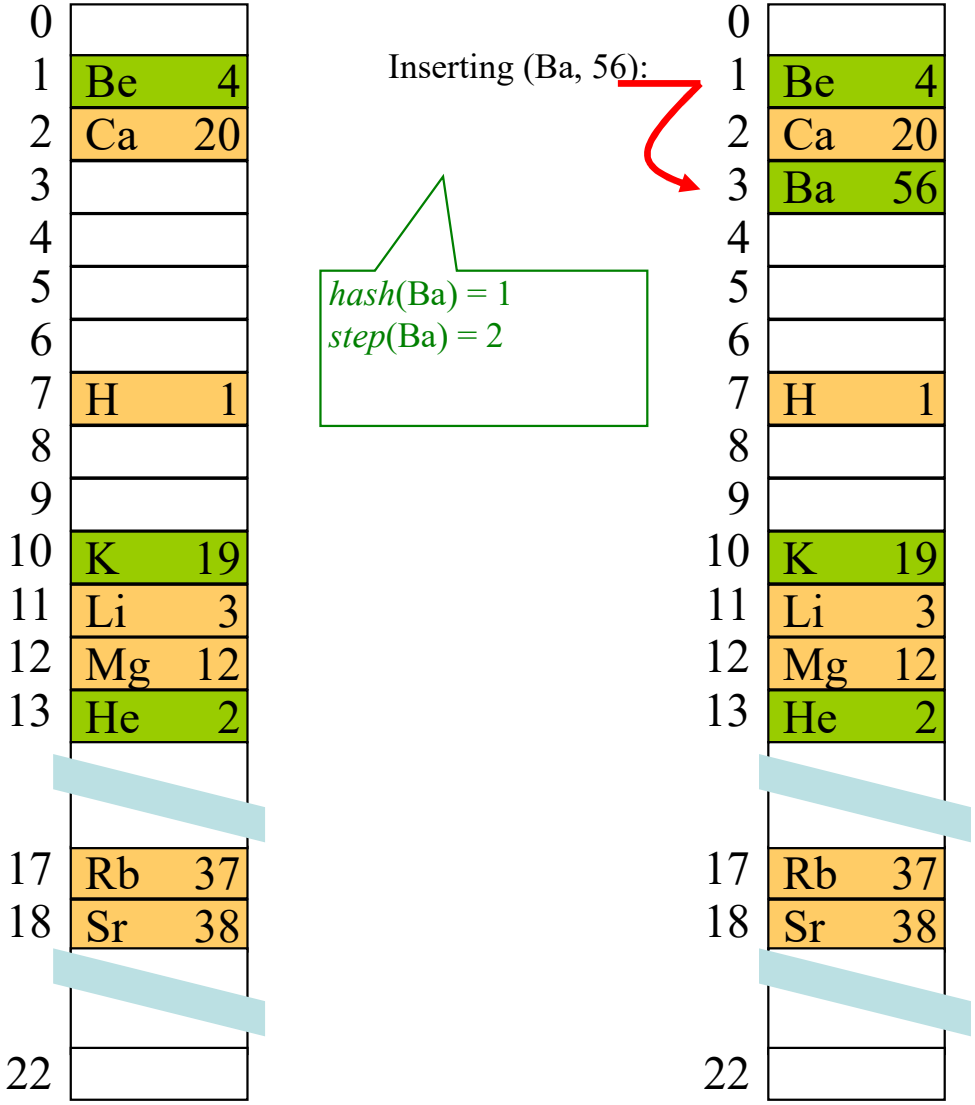
Double hashing

- Illustration:



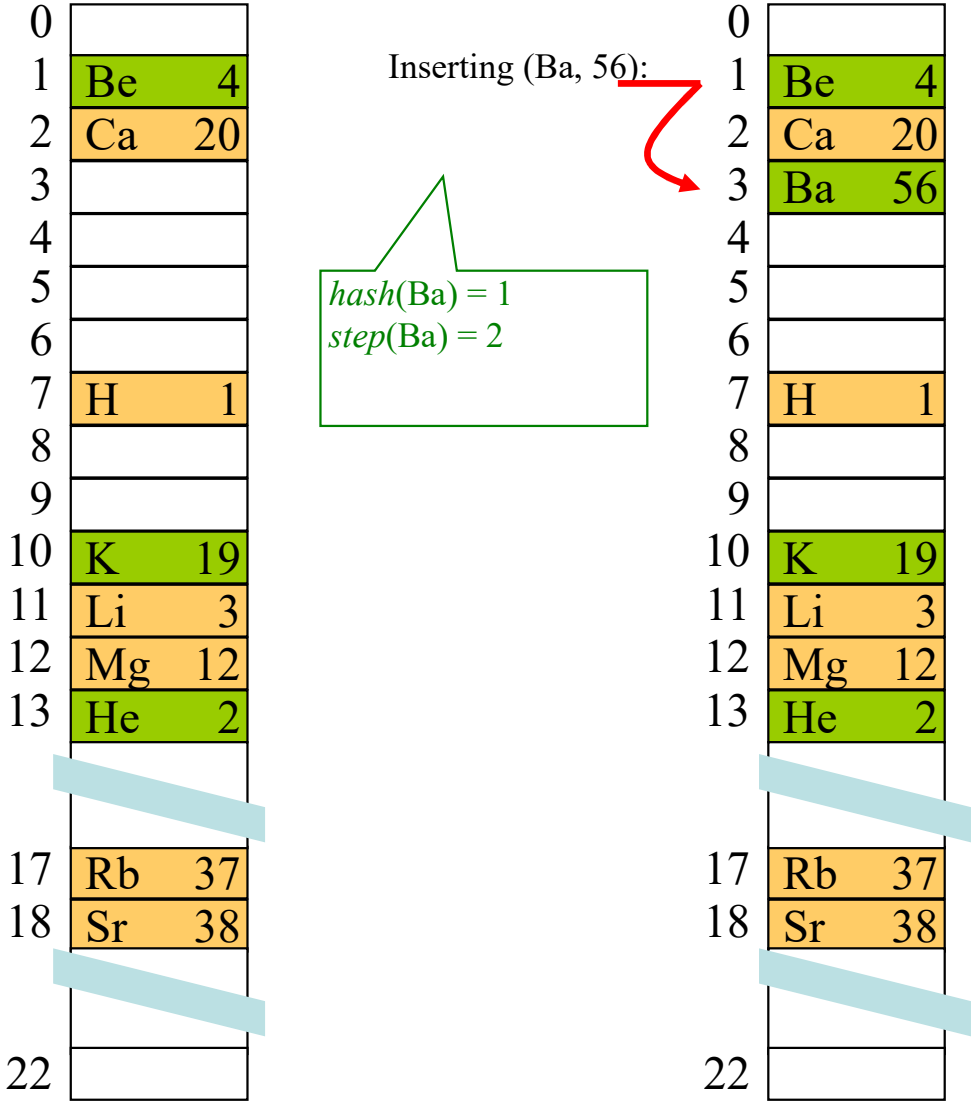
Double hashing

- Illustration:



Double hashing

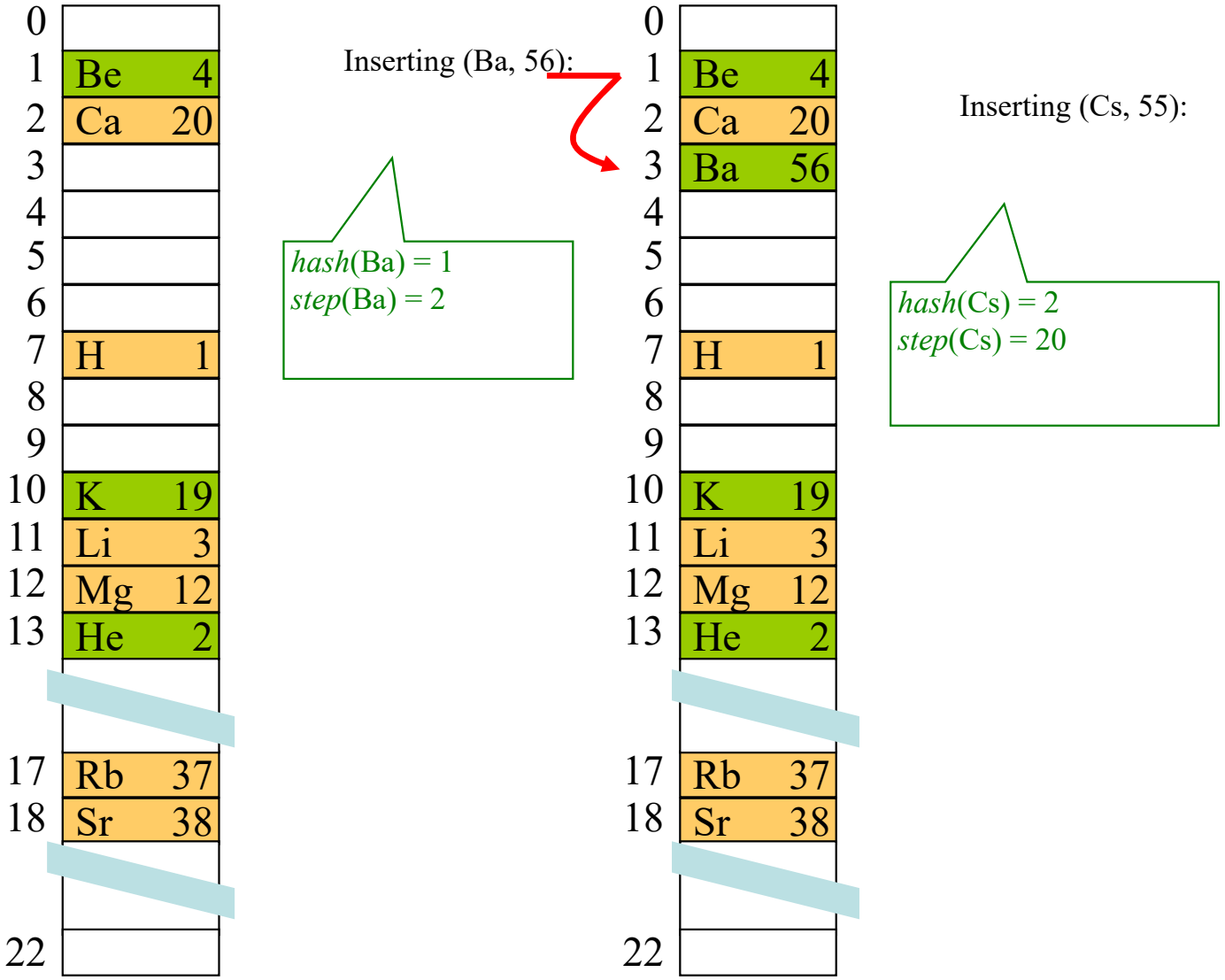
- Illustration:



Inserting (Cs, 55):

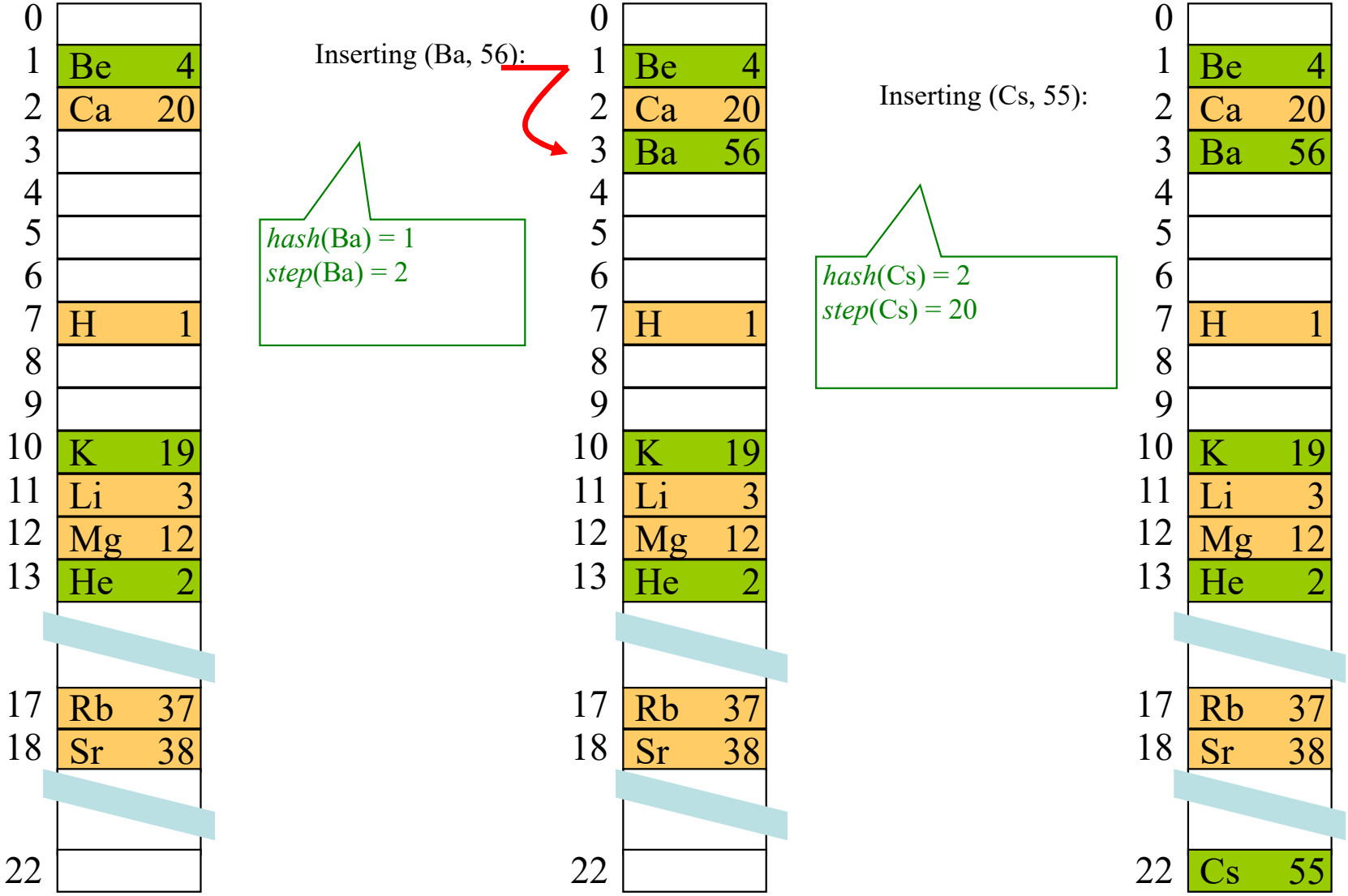
Double hashing

- Illustration:



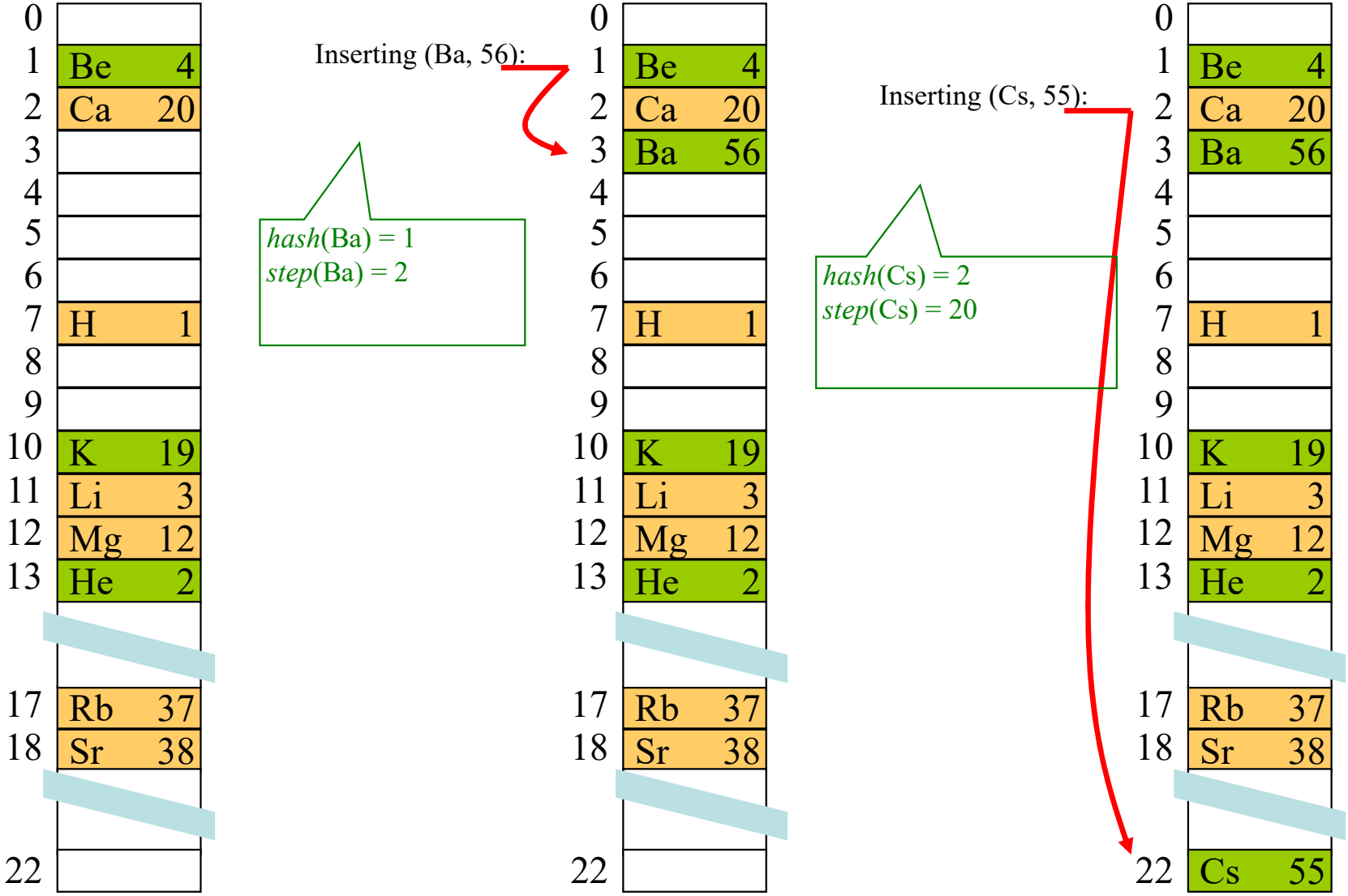
Double hashing

- Illustration:



Double hashing

- Illustration:



Double hashing insertion

- To insert the entry (*key*, *val*) into an OBHT:
  1. Set *b* to *hash*(*key*), and set *s* to *step*(*key*).
  2. Repeat:
     - 2.1. If bucket *b* is never-occupied:
       …
     - 2.2. If bucket *b* is formerly-occupied, or is occupied by an entry whose key is equal to *key*:
       …
     - 2.3. If bucket *b* is occupied by an entry whose key is not equal to *key*:
       - 2.3.1. Increment *b* by *s*, modulo *m*.

Hash tables in practice

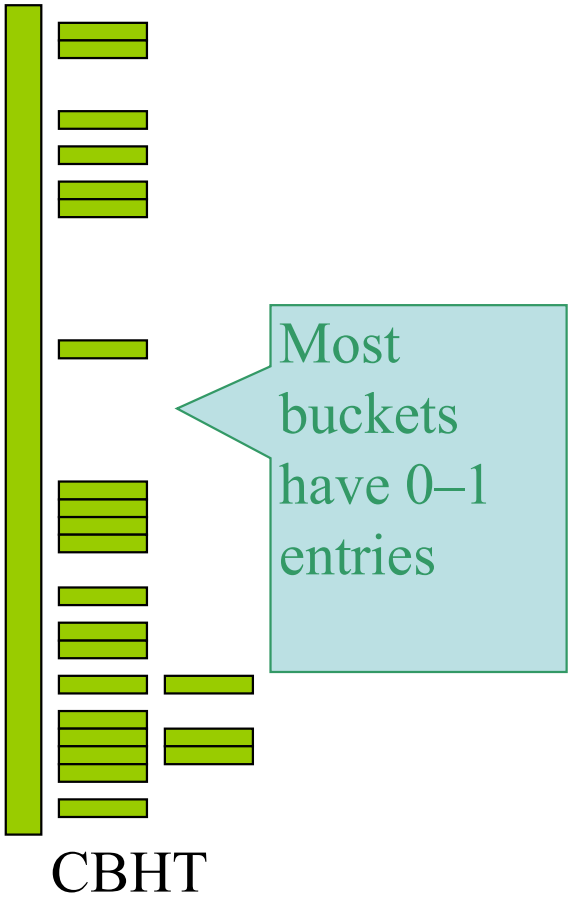
- The following trials compared the performance of three hash tables, each with *m* buckets:
  - a CBHT
  - an OHBT with step length *s* = 1
  - an OHBT with step length *s* determined by double hashing.
- The hash function was chosen to distribute keys uniformly among the *m* buckets (i.e., the probability that a key is mapped to any particular bucket was 1/*m*).
- In each trial, all three hash tables were loaded with the same set of *n* randomly-generated keys.

Hash tables in practice

- Test with $m = 47$ and $n = 23$ (load factor ≈ 0.5):

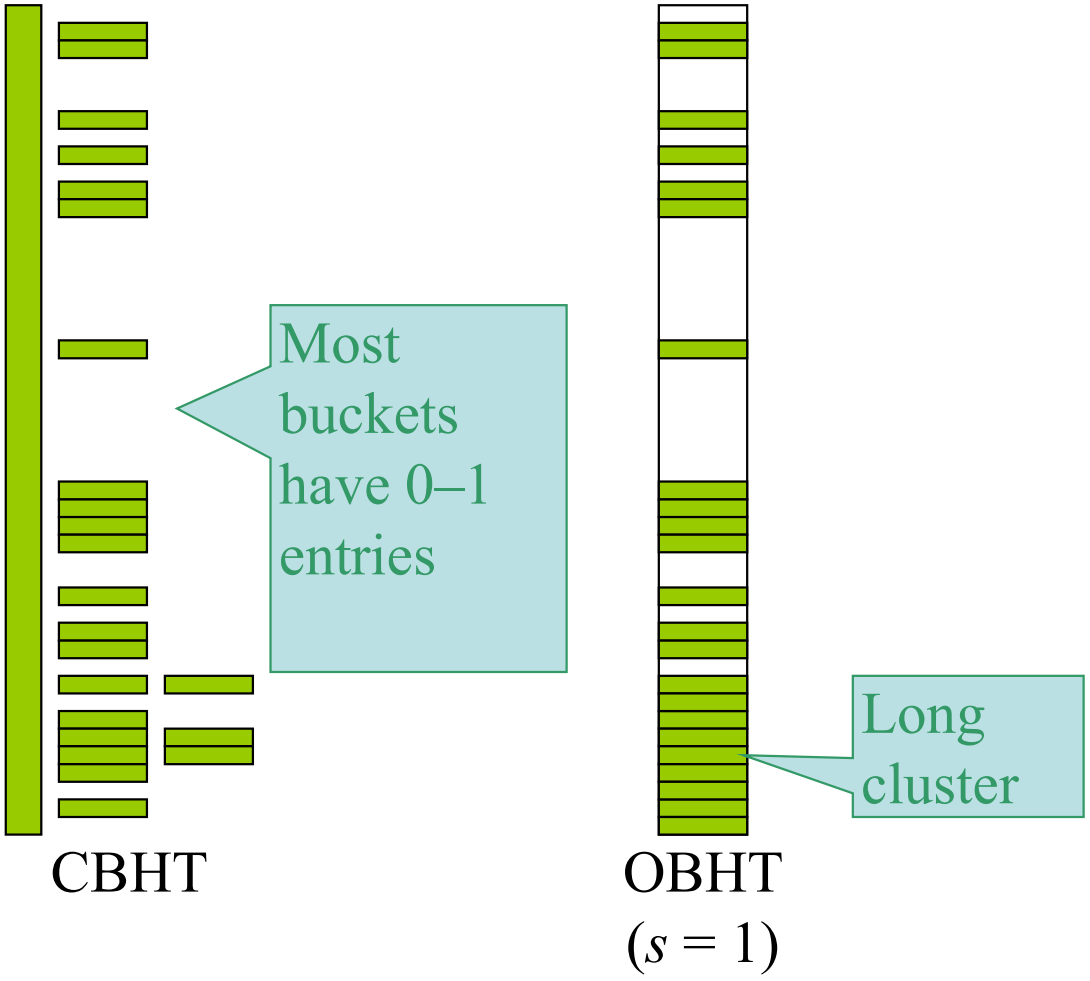
Hash tables in practice

- Test with $m = 47$ and $n = 23$ (load factor ≈ 0.5):



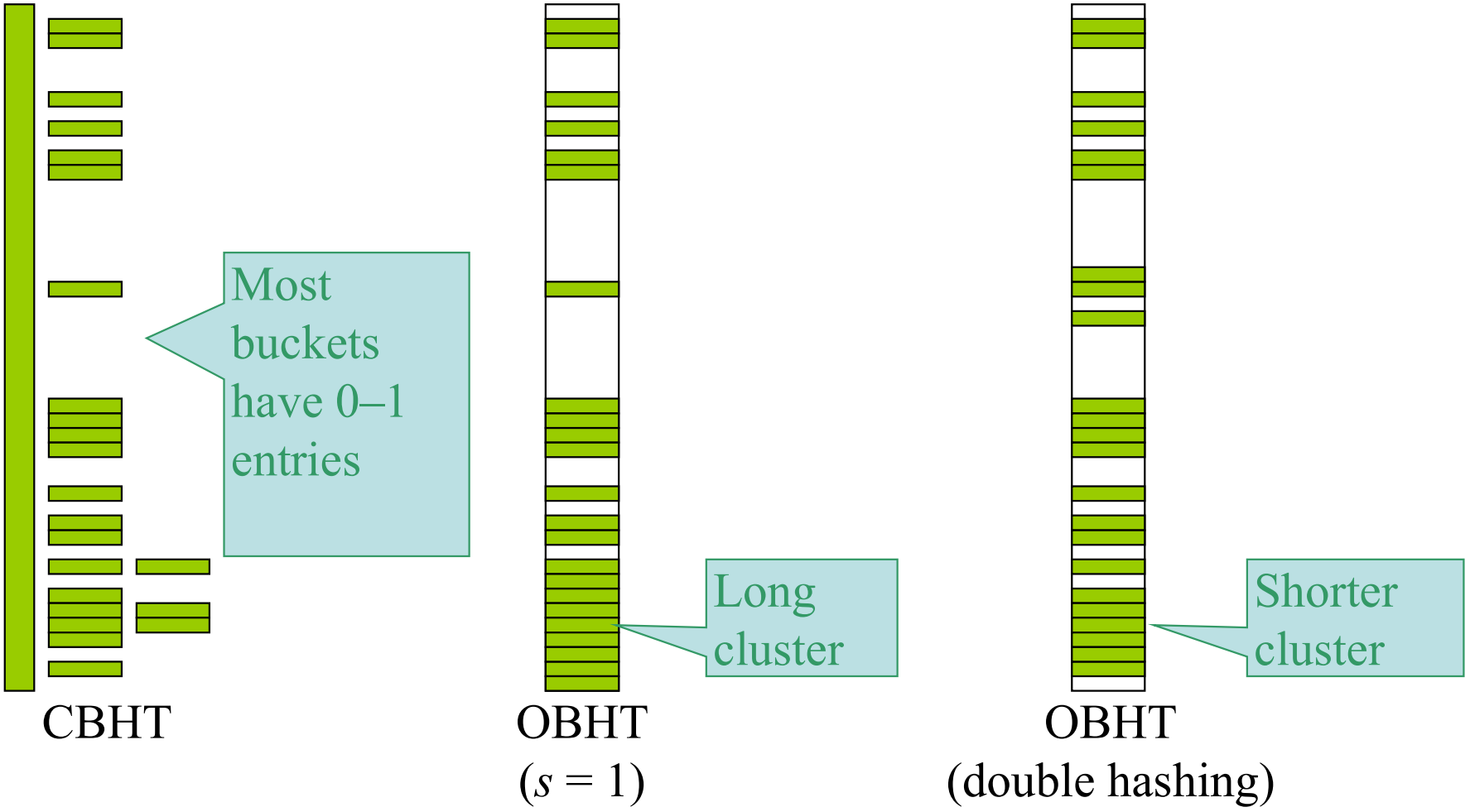
Hash tables in practice

- Test with $m = 47$ and $n = 23$ (load factor ≈ 0.5):



Hash tables in practice

- Test with $m = 47$ and $n = 23$ (load factor ≈ 0.5):

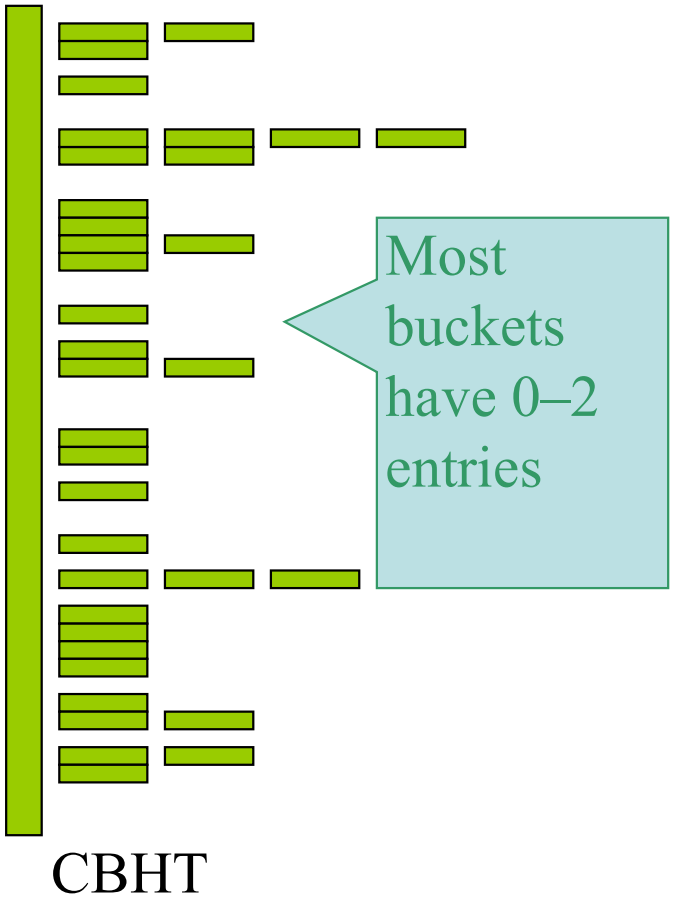


Hash tables in practice

- Test with $m = 47$ and $n = 36$ (load factor ≈ 0.75):

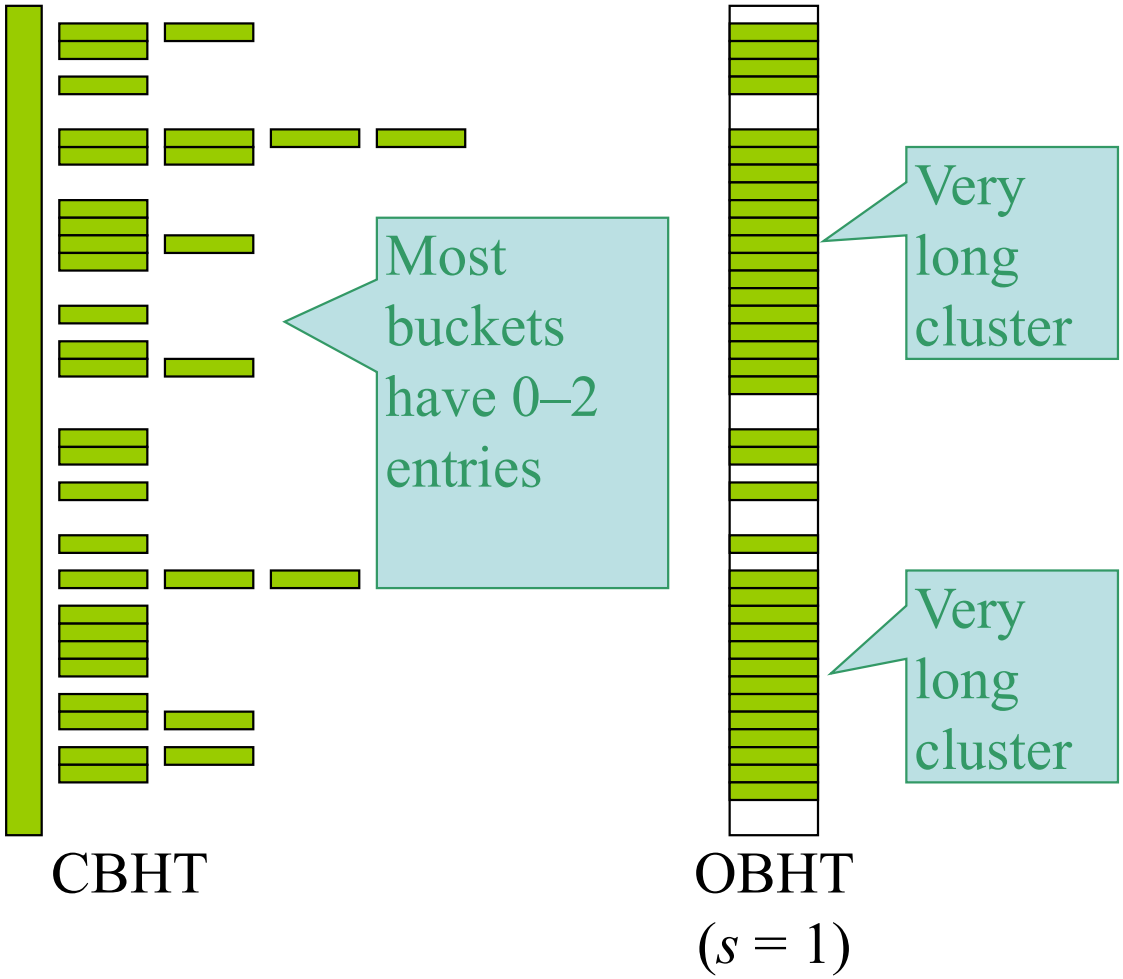
Hash tables in practice

- Test with $m = 47$ and $n = 36$ (load factor ≈ 0.75):



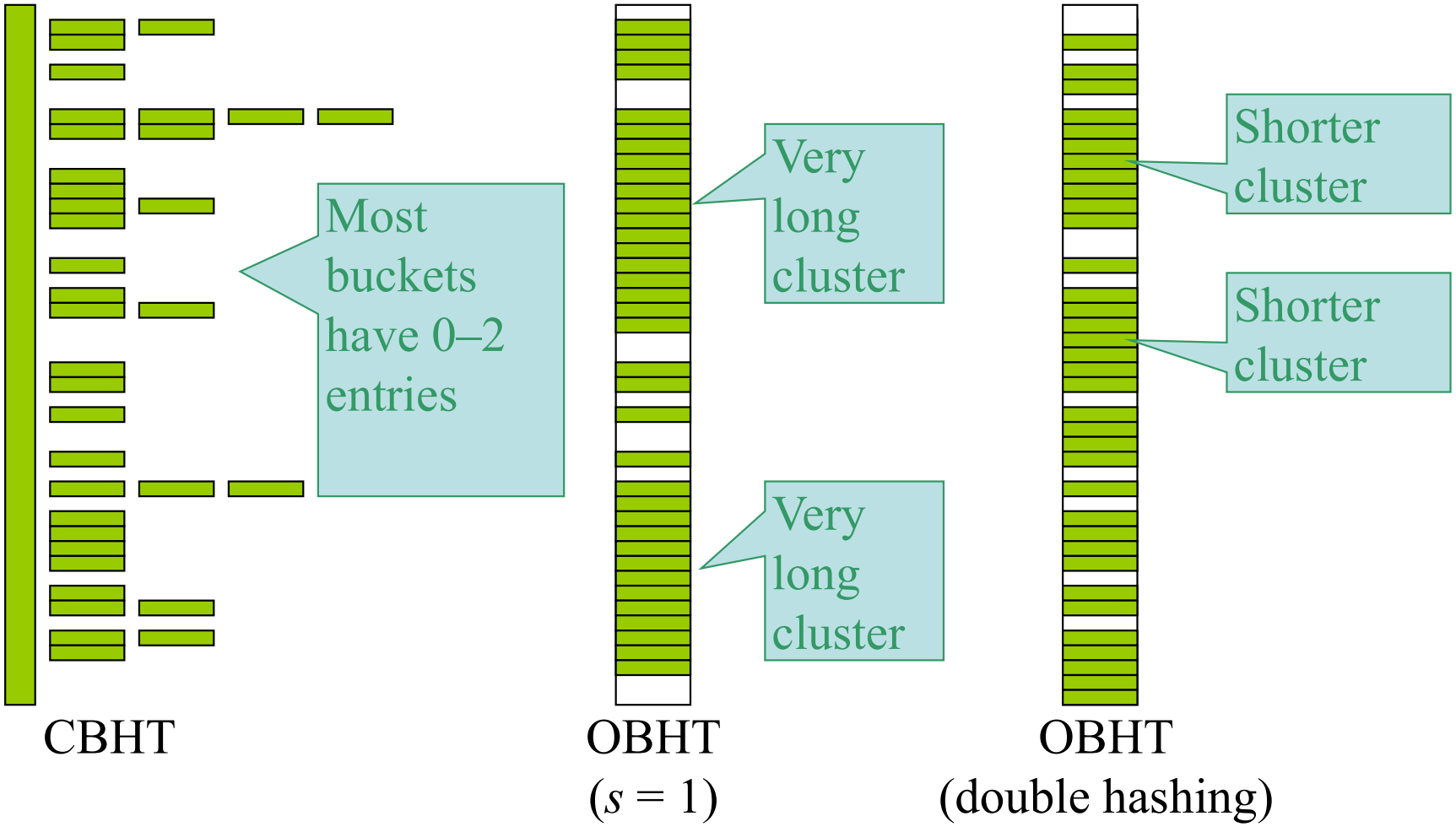
Hash tables in practice

- Test with $m = 47$ and $n = 36$ (load factor ≈ 0.75):



Hash tables in practice

- Test with $m = 47$ and $n = 36$ (load factor ≈ 0.75):



Example: student records

- Consider a hypothetical university's student records. About 1500 new students register each year, and most students stay for 4 years.
- Each student has a unique id, of the form *yydddd* (where *yy* are the last two digits of the year of first registration, and where *dddd* is a serial number).
- Suppose that the student records will be held in a hash table.
- *hash*(*id*) can depend on any or all of *id*'s digits.

Example: student records

- Consider $m = 100$, *hash*(*id*) = first two digits of *id*.
  - – Far too few buckets. Load factor ≈ 6000/100 ≈ 60.
  - – Very uneven distribution. E.g., in academic year 2001–02, most ids start with 98, 99, 00, or 01.
- Consider $m = 10000$, *hash*(*id*) = last four digits of *id*.
  - + Good number of buckets. Load factor ≈ 6000/10000 ≈ 0.6.
  - – Uneven distribution. Most ids end with 0000…1500.
- Consider $m = 9997$, *hash*(*id*) = *id* modulo *m*.
  - + Good number of buckets. Load factor ≈ 6000/9997 ≈ 0.6.
  - + Even distribution (since *m* is prime).

Example: student records

- Consider OBHT with $s = 1$.
  - Four clusters of about 1500 entries.
- Consider OBHT with $s = 2000$.
  - \+ Should avoid clustering.
- Consider OBHT with double hashing.
  - \+ Should avoid clustering.

Implementation of sets using CBHTs

- Complexity:

Operation	Algorithm	Time complexity
`search`	CBHT search	$O(1)$ best $O(n)$ worst
`insertion`	CBHT insertion	$O(1)$ best $O(n)$ worst
`deletion`	CBHT deletion	$O(1)$ best $O(n)$ worst