Structuri de date și algoritmi

Hash Map (Hash Table)

Lect. Dr. Ing. Cristian Chilipirea – cristian.chilipirea@mta.ro

Reminder vector

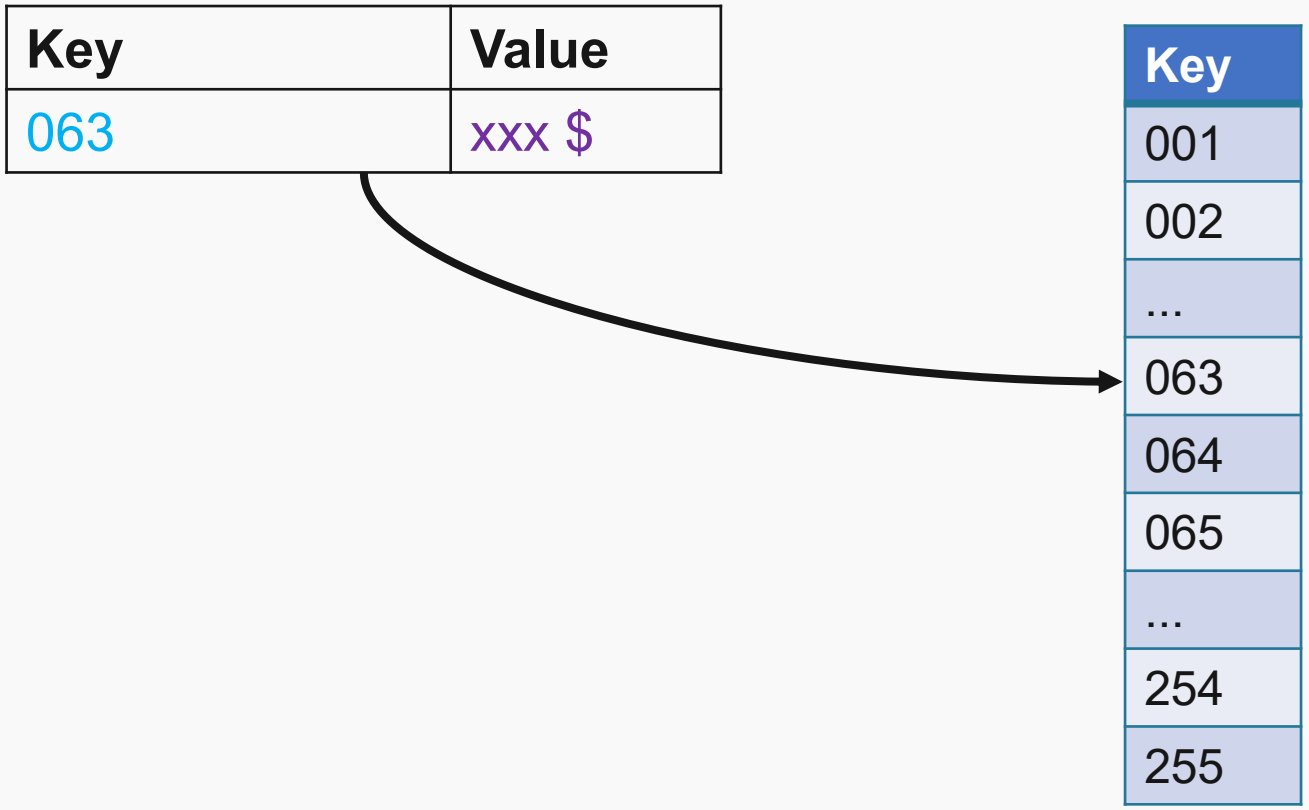
Complexitate	Vector
Acces	O(1)
Inserție	O(N)
Inserție capete	O(N)/O(1)
Ștergere	O(N)
Ștergere capete	O(N)/O(1)
Căutare ordonat/neordonat	O(log(N))/O(N)

Map pe scurt

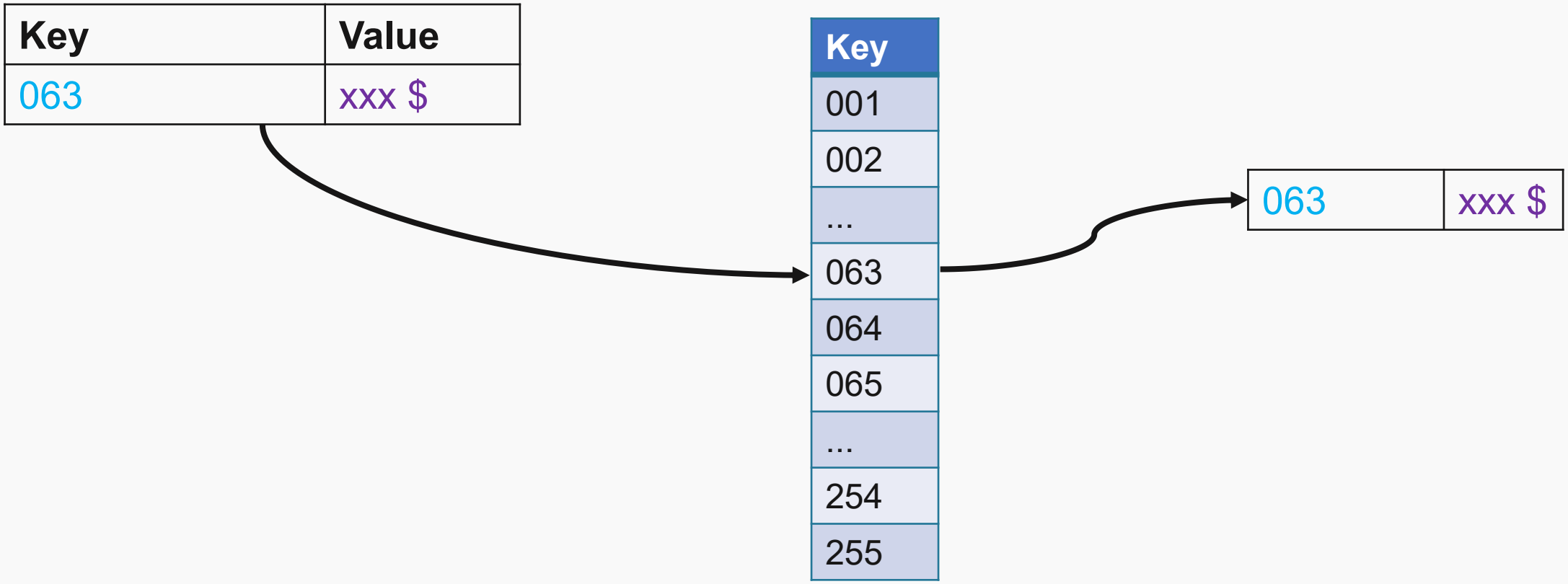
Key	Value
063	xxx $

Key
001
002
...
063
064
065
...
254
255

Map pe scurt

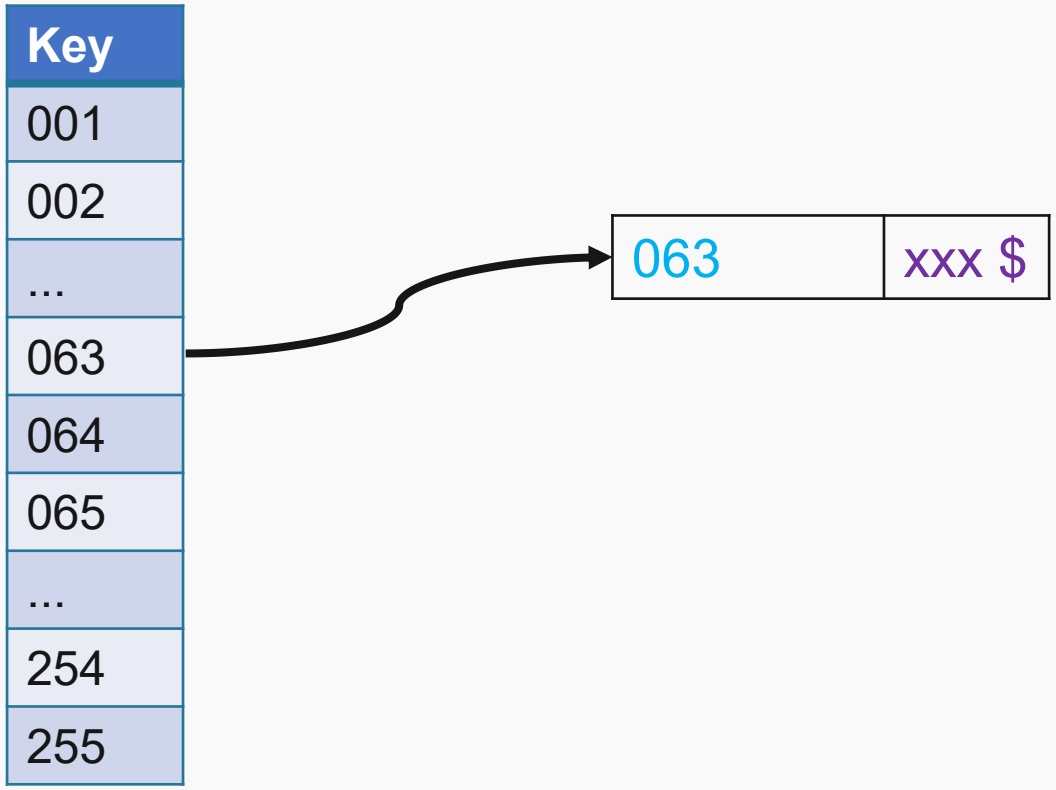


Map pe scurt

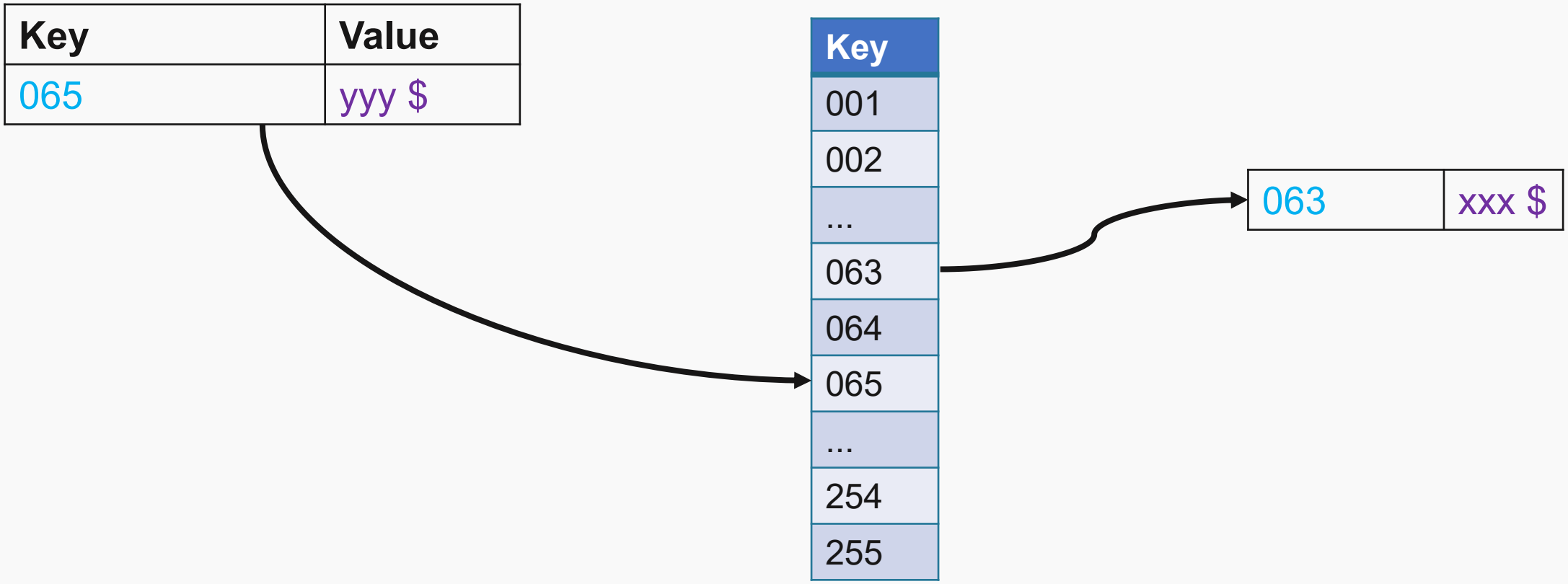


Map pe scurt

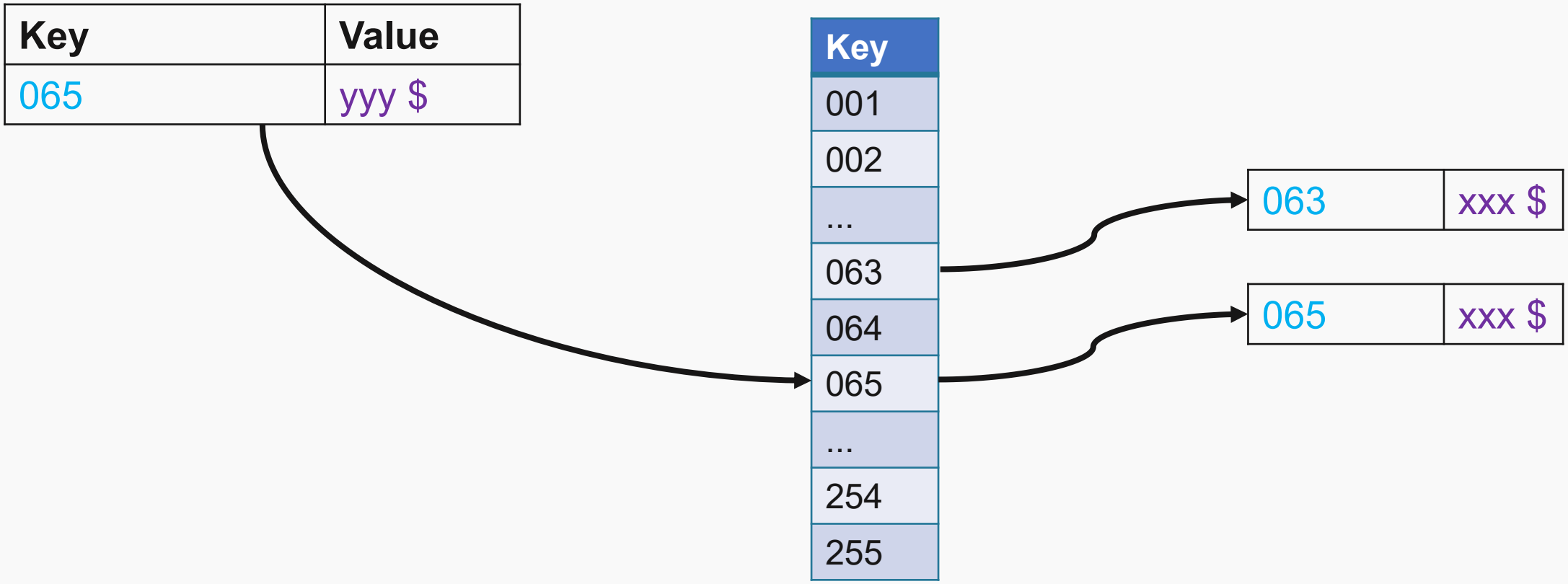
Key	Value
065	yyy $



Map pe scurt



Map pe scurt


Key
Value
065
yyy $
Key
001
002
...
063
064
065
...
254
255
063
xxx $
065
xxx $

Map – operații

```
valueType search(keyType key) {
    return T[key];
}

void insert(keyType key, valueType value) {
    T[key] = value;
}

void delete(keyType key) {
    T[key] = NULL;
}
```

Dezavantaj

- Nu sunt multe cazuri în care avem chei numere întregi.
- Dorim chei mai complexe, de exemplu string-uri.

HashMap (Tabelă hash) pe scurt

Key	Value
Nume_Prenume	xxx $

Key
001
002
...
063
064
065
...
254
255

HashMap pe scurt

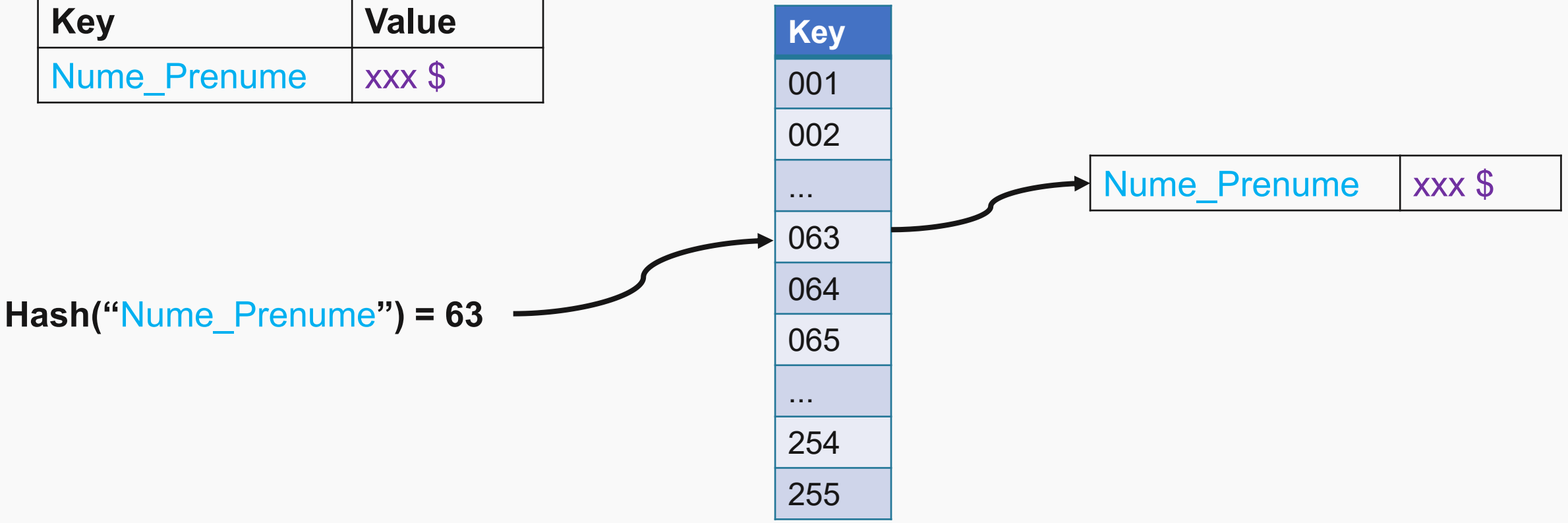
Key	Value
Nume_Prenume	xxx $

Hash("Nume_Prenume") = 63

Key
001
002
...
063
064
065
...
254
255

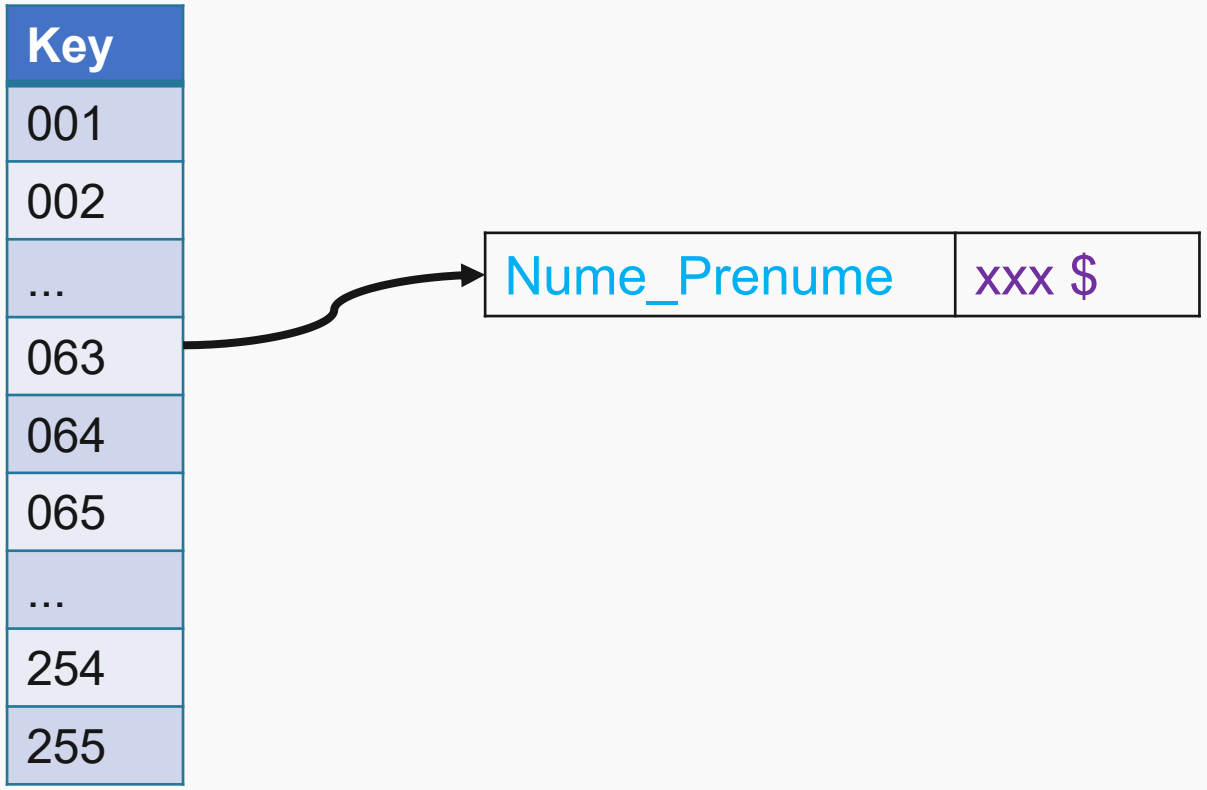
HashMap pe scurt

Key	Value
Nume_Prenume	xxx $



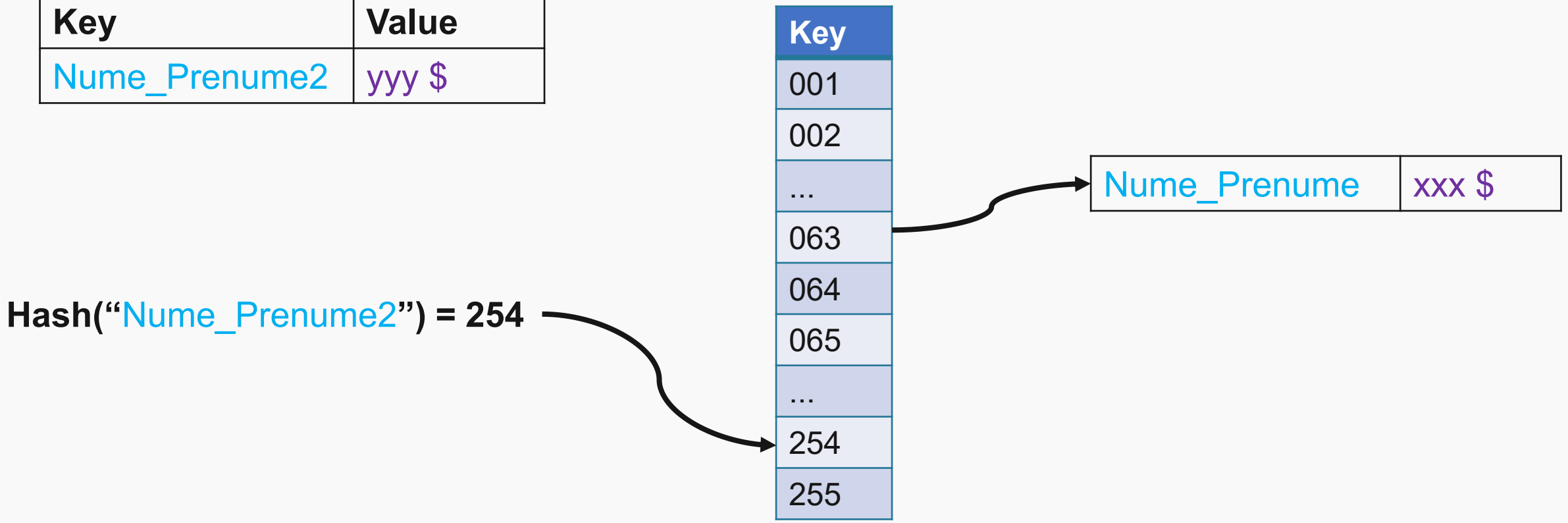
HashMap pe scurt

Key	Value
Nume_Prenume2	yyy $



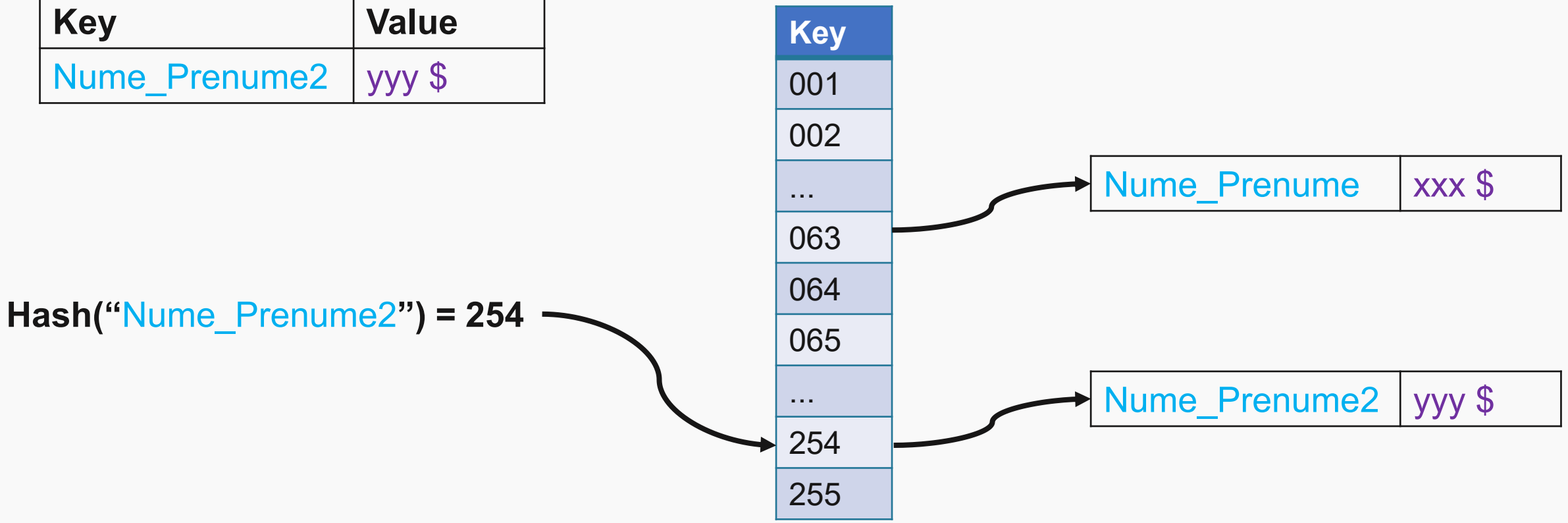
HashMap pe scurt

Key	Value
Nume_Prenume2	yyy $



HashMap pe scurt

Key	Value
Nume_Prenume2	yyy $



Complexitate HashMap

Complexitate	Vector	Hash Map
Acces	O(1)	O(1)
Inserție	O(N)	O(1)
Inserție capete	O(N)/O(1)	n/a
Ștergere	O(N)	O(1)
Ștergere capete	O(N)/O(1)	n/a
Căutare ordonat/neordonat	O(log(N))/O(N)	O(1)

Funcție hash

- Întoarce mereu aceeași valoare pentru același input.
- Pentru un input de dimensiune arbitrară returnează valori dintr-un anumit set.

Conversie simplă String -> Număr întreg

```
int textToInteger(char* text, int MAX)
{
    int l = strlen(text);
    int rez = 0;
    for (int i = 0; i < l; i++) {
        rez = rez * 256 + text[i];
        rez %= MAX;
    }
    return rez;  // numere între 0 și MAX-1
}
```

Funcții hash naivă

```
int sizeOfHashMap = 11; // M
int h(int val)
{
    return val % sizeOfHashMap;
}

int hashString(char* text)
{
    int val = textToInteger(text, 256);
    return h(val);
}
```

Funcții și tabele hash

- K este mulțimea de chei.
- M mărimea hashMap.

$$h: K -> \{0,1, \dots, M-1\}$$
$$h(x) = x \bmod M$$

Poziție	0	1	2	3	4	5	6	7	8	9	10
Cheie	-	34	-	25	48	-	-	7	96	-	87

Unde este introdus 60? Dar 59?

Coliziuni

Se numește **coliziune** momentul în care două chei au același hash

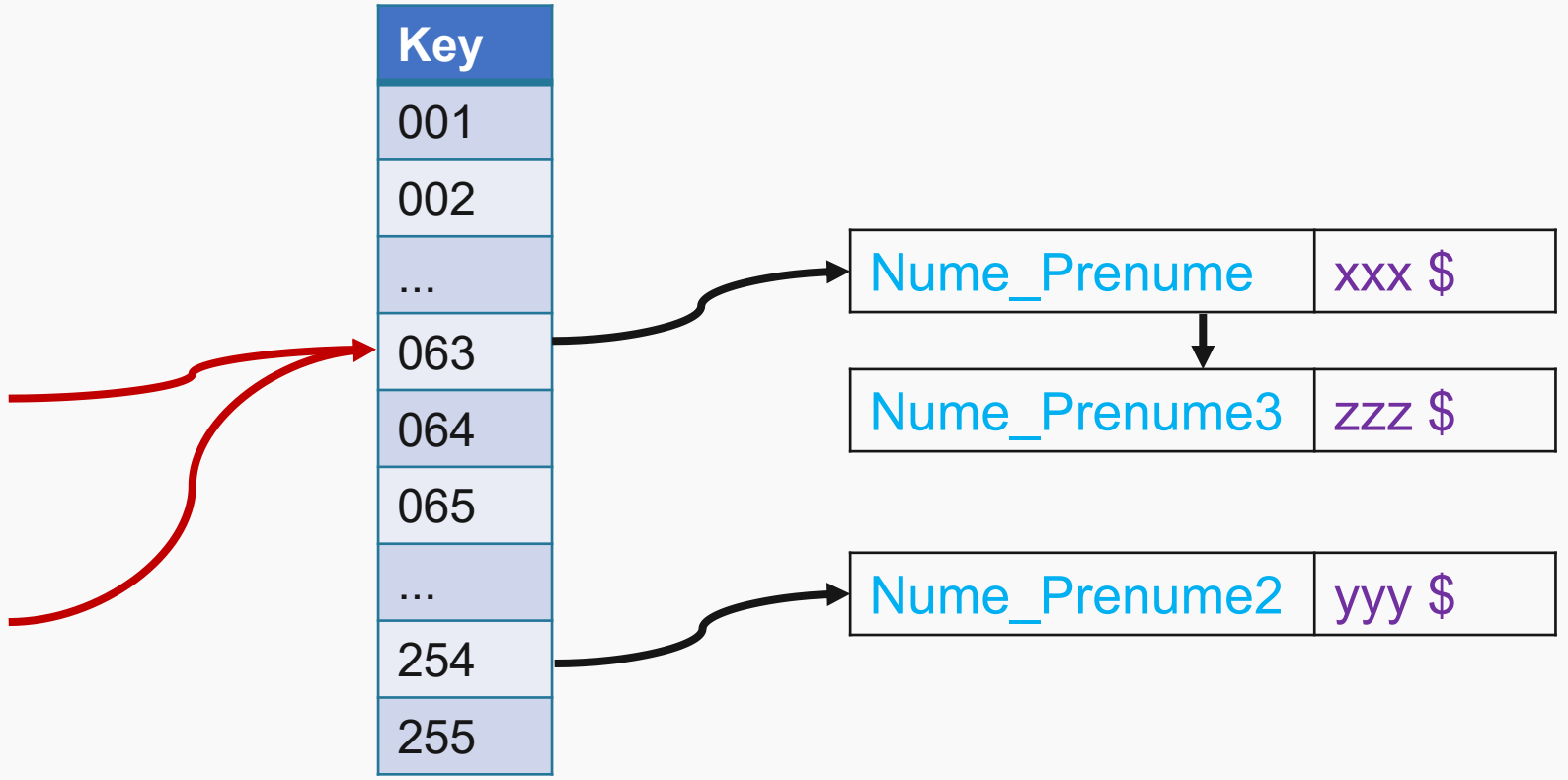
$$A \neq B$$
$$hash(A) = hash(B)$$

Rezolvare coliziunilor prin înlănțuire (chaining)

Key	Value
Nume_Prenume	xxx $
Nume_Prenume2	yyy $
Nume_Prenume3	zzz $

Hash(“Nume_Prenume”) = 063

Hash(“Nume_Prenume3”) = 063



Se folosesc liste înlănțuite (chiar dublu înlănțuite)

Rezolvare coliziunilor prin înlănțuire (chaining)

```
valueType search(keyType key)
{
    return listSearch(T[hash(key)], key);
}

void insert(keyType key, valueType value)
{
    listInsertHead(T[hash(key)], key, value);
}

void delete (keyType key)
{
    listDelete(T[hash(key)], key);
}
```

Complexitate?

Complexitate operații?

- N chei
- M sloturi în tabela hash
- Funcția hash are complexitate $O(1)$

- Cel mai defavorabil?

Complexitate operații?

- N chei
- M sloturi în tabela hash
- Funcția hash are complexitate $O(1)$

- Cel mai defavorabil?
  - Toate cheile au același hash.
  - Toate sunt coliziuni
  - Se folosește un singur slot.
  - $O(N)$

Complexitate operații?

- N chei
- M sloturi în tabela hash
- Funcția hash are complexitate $O(1)$

- Cel mai favorabil?
  - Cheile sunt distribuite uniform.
  - **Simple uniform hashing.**
  - Sunt $\alpha = \frac{N}{M}$ chei pentru fiecare slot din tabelă

Complexitate operații?

- N chei
- M sloturi în tabela hash
- Funcția hash are complexitate $O(1)$

- Cel mai favorabil?
  - Cheile sunt distribuite uniform.
  - **Simple uniform hashing.**
  - Sunt $\alpha = \frac{N}{M}$ chei pentru fiecare slot din tabelă
  - $O(\alpha)$ **(demonstrație în Cormen)**

Complexitate operații?

- N chei
- M sloturi în tabela hash
- Funcția hash are complexitate $O(1)$

- Cel mai favorabil?
  - Cheile sunt distribuite uniform.
  - **Simple uniform hashing.**
  - Sunt $\alpha = \frac{N}{M}$ chei pentru fiecare slot din tabelă
  - $O(\alpha)$ **Dar dacă N proporțional cu M, adică N=cM =>** $O\left(\frac{cM}{M}\right) = O(1)$

Funcții hash – Metoda Diviziunii

- $h(x) = x \bmod M$
- Cum alegem M?
  - Dorim **simple uniform hashing – Fiecare cheie are aceeași probabilitate să ajungă în oricare din cele M slot-uri.**

Funcții hash – Metoda Diviziunii

- $h(x) = x \, mod \, M$
- Cum alegem M?
  - Dorim **simple uniform hashing – Fiecare cheie are aceeași probabilitate să ajungă în oricare din cele M slot-uri.**
  - Dacă M are un divizor d avem multe coliziuni.
  - Cheile care sunt multipli de d vor ajunge pe poziții multipli cu d.

Funcții hash – Metoda Diviziunii

- $h(x) = x \, mod \, M$
- Cum alegem M?
  - Dorim **simple uniform hashing – Fiecare cheie are aceeași probabilitate să ajungă în oricare din cele M slot-uri.**
  - Dacă M are un divizor d avem multe coliziuni.
  - Cheile care sunt multipli de d vor ajunge pe poziții multipli cu d.
  - $x = yd$
  - $yd \, mod \, M = r \Rightarrow yd = zM + r \Rightarrow r = r'd$

Funcții hash – Metoda Diviziunii

- $h(x) = x \bmod M$
- Cum alegem M?
  - Dorim **simple uniform hashing – Fiecare cheie are aceeași probabilitate să ajungă în oricare din cele M slot-uri.**
  - Dacă $M = 2^p$ atunci toate hash-urile vor fi formate din p biți ai lui x.
  - 00000000_00000000_00000100_000000000
  - bbbbbbbb_bbbbbbbb_bbbbbb**bb_bbbbbbbb** număr binar

p

Funcții hash – Metoda Diviziunii

- $h(x) = x \, mod \, M$
- Cum alegem M?
  - Dorim **simple uniform hashing – Fiecare cheie are aceeași probabilitate să ajungă în oricare din cele M slot-uri.**
  - M în general se alege ca fiind un număr prim dar nu de forma $2^p - 1$

Funcții hash metoda multiplicării

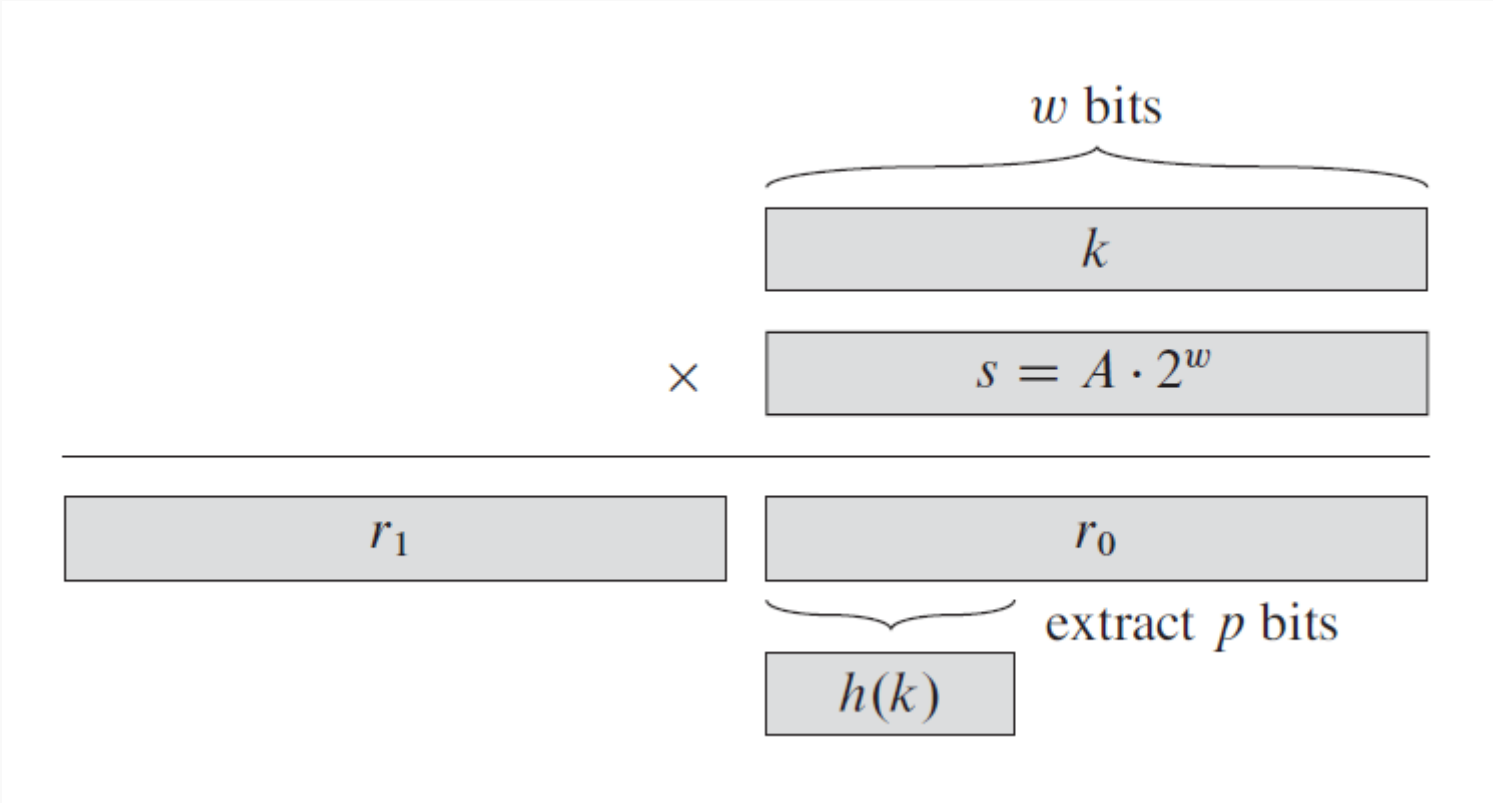
- $h(k) = \lfloor M(kA \bmod 1) \rfloor$
- $A \in (0,1)$ constantă
  - Knuth propune $A \approx \frac{(\sqrt{5}-1)}{2} = 0.618033$
- $kA \bmod 1 = kA - \lfloor kA \rfloor$
  - este partea fracționară a lui kA
- M poate lua orice valoare, de obicei 2^p

Funcții hash metoda multiplicării

- $h(k) = \lfloor M(kA \bmod 1) \rfloor$
- $A \in (0,1)$ constantă
  - Knuth propune $A \approx \frac{(\sqrt{5}-1)}{2} = 0.618033$
  - $A = \frac{s}{2^w}$. Cât poate fi s?
- $kA \bmod 1 = kA - \lfloor kA \rfloor$
  - este partea fracționară a lui kA
- M poate lua orice valoare, de obicei 2^p

Funcții hash metoda multiplicării

- $h(k) = \lfloor M(kA \bmod 1) \rfloor$
- $A = \frac{s}{2^w}$
- $M = 2^p$



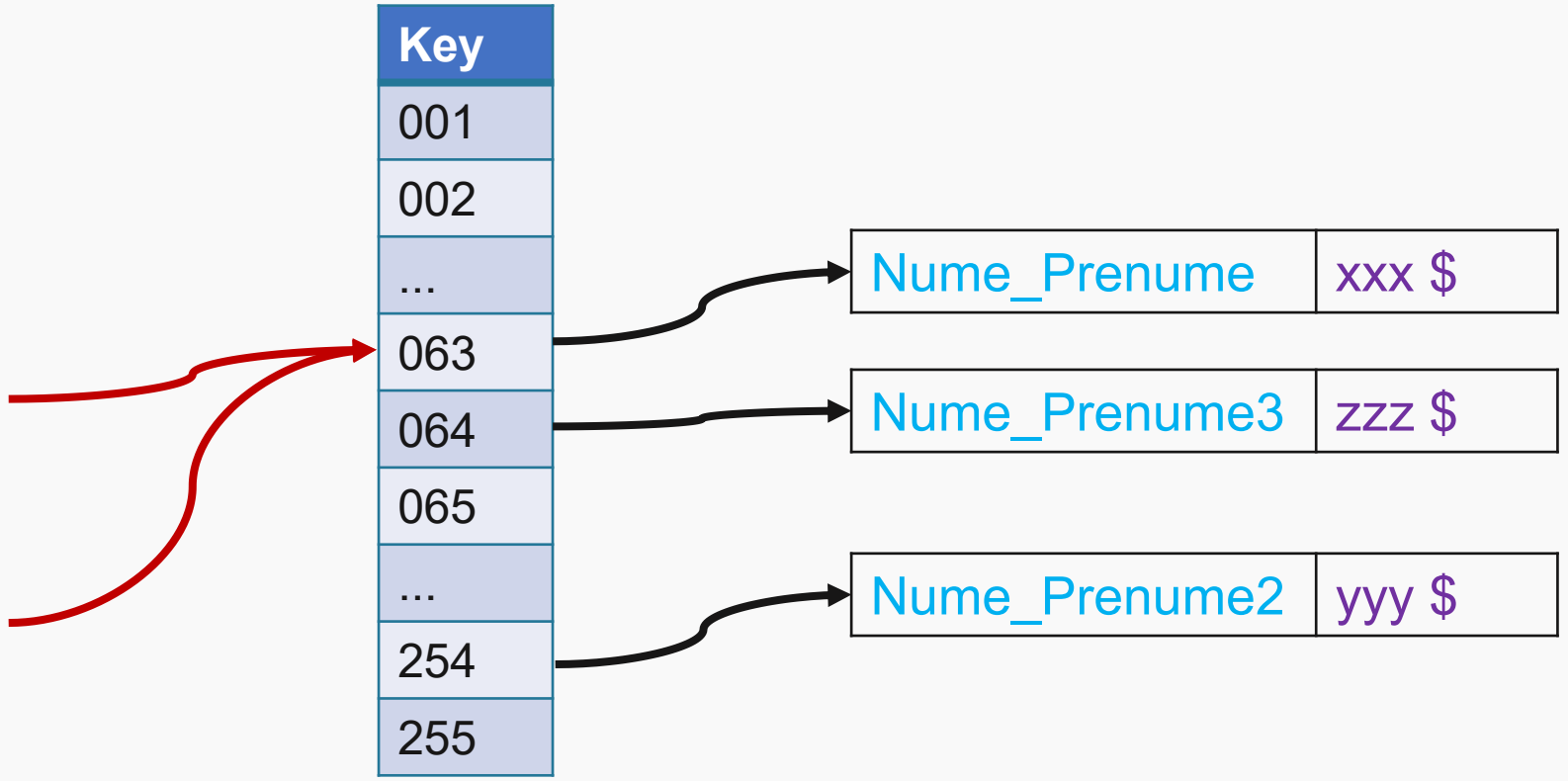
Din Cormen

Rezolvare coliziunilor prin Adresare deschisă

Key	Value
Nume_Prenume	xxx $
Nume_Prenume2	yyy $
Nume_Prenume3	zzz $

Hash(“Nume_Prenume”) = 063

Hash(“Nume_Prenume3”) = 063



Se pune pe următoarea poziție goală

Rezolvare coliziunilor prin Adresare deschisă

- Tabela poate fi complet consumată
- $\alpha = 1$
- Poziția exactă depinde de ordinea în care facem inserțiile
- Extindem funcția de hash pentru a permite mai multe încercări.
  - $h(k, i)$
  - Se va apela pe rând cu $i = 0,1, \ldots, M - 1$

HashMap – Adresare deschisă - Insert

```
void insert(keyType key, valueType value)
{
    for (int i = 0; i < sizeOfHashMap; i++) { // M == sizeOfHashMap
        int j = hash(key, i);
        if (T[j] == NULL) {
            T[j] = value;
            return;
        }
    }
    printf("hashMap full");
}
```

HashMap – Adresare deschisă - Search

```
valueType search(keyType key)
{
    for (int i = 0; i < sizeOfHashMap; i++) {  // M == sizeOfHashMap
        int j = hash(key, i);
        if (T[j] == NULL)
            return NULL;
        if (T[j] == key)
            return T[j];
    }
}
```

HashMap – Adresare deschisă - Delete

```
valueType search(keyType key)
{
    for (int i = 0; i < sizeOfHashMap; i++) {  // M == sizeOfHashMap
        int j = hash(key, i);
        if (T[j] == NULL)
            return NULL;
        if (T[j] == key)
            return T[j];
    }
}
```

HashMap – Adresare deschisă – verificare liniară

- $h(k,i) = (h(k) + i)\ mod\ M$
- Elementele sunt puse la rând.
- Problemă: Se pot crea zone mari ocupate, astfel crește timpul fiecărei noi inserții. – **primary clustering.**

HashMap – Adresare deschisă – verificare pătratică (quadatric)

- $h(k,i) = (h(k) + c_1 i + c_2 i^2)\ mod\ M$
- c_1 și c_2 sunt constante pozitive
- Elementele sunt puse dispersate quadratic.
- Dacă $h(k_1, 0) = h(k_2, 0)$ atunci $h(k_1, i) = h(k_2, i)$.
- Se numește **secondary clustering**.

HashMap – Adresare deschisă – double hashing

- $h(k,i) = (h_1(k) + ih_2(k))\ mod\ M$
- $h_1(k) = k\ mod\ M$
- $h_2(k) = 1 + (k\ mod\ M')$
  - $M' < M$ (poate fi $M - 1$)

HashMap – Adresare deschisă – double hashing

- $h(k,i) = (h_1(k) + ih_2(k))\ mod\ M$
- $h_1(k) = k\ mod\ M$
- $h_2(k) = 1 + (k\ mod\ M')$
  - $M' < M$ (poate fi $M - 1$)

Din Cormen

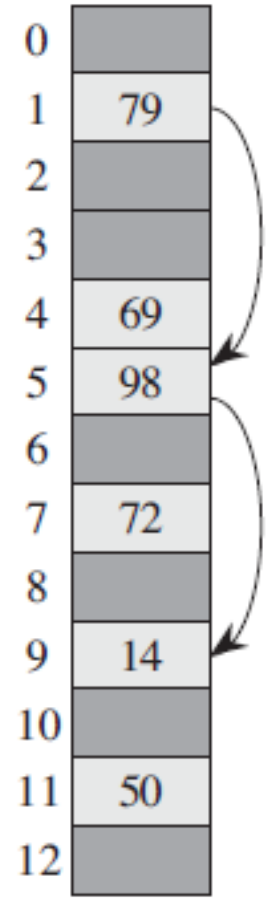


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

Hash – Stringuri - Naiv

```
int hash(char *str, int M)
{
    int sum = 0;
    for (; *str; str++) {
        sum += *str;
        sum %= M;
    }
    return sum % M;
}
```

Performanță

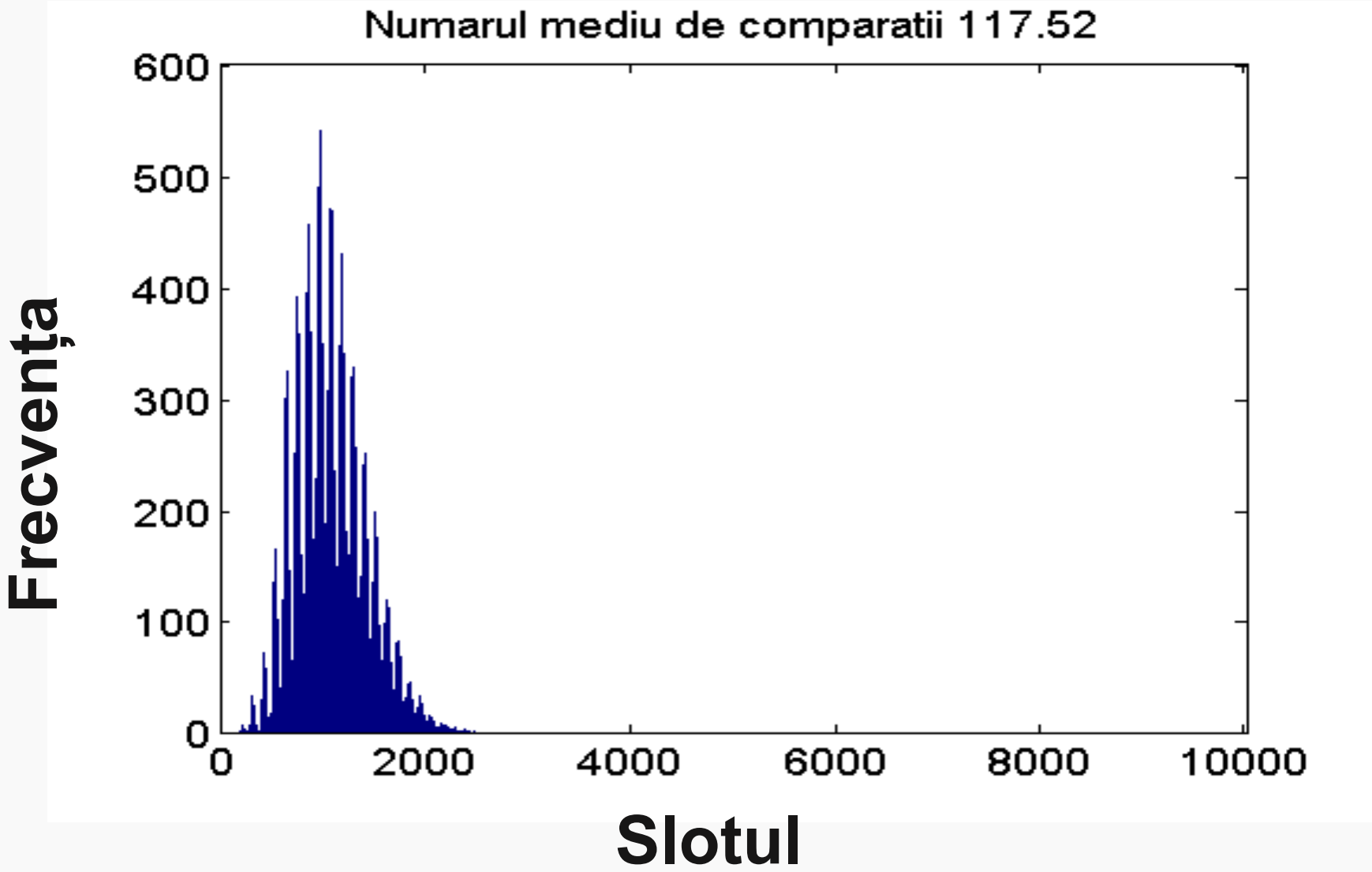
- Testăm cu dicţionar în engleză care conţine 219.154 cuvinte.
- Tabela de hash are 10.007 (număr prim) intrări.
- Încărcarea ideală (uniformă) a tabelei este

$$\frac{219.154}{10.007} \cong \mathbf{21,9}$$

- Numărul mediu de comparaţii optim este:

$$\frac{21{,}9 + 1}{2} \cong \mathbf{11,45}$$

Hash – Stringuri - Naiv



Hash – Stringuri – Berkley

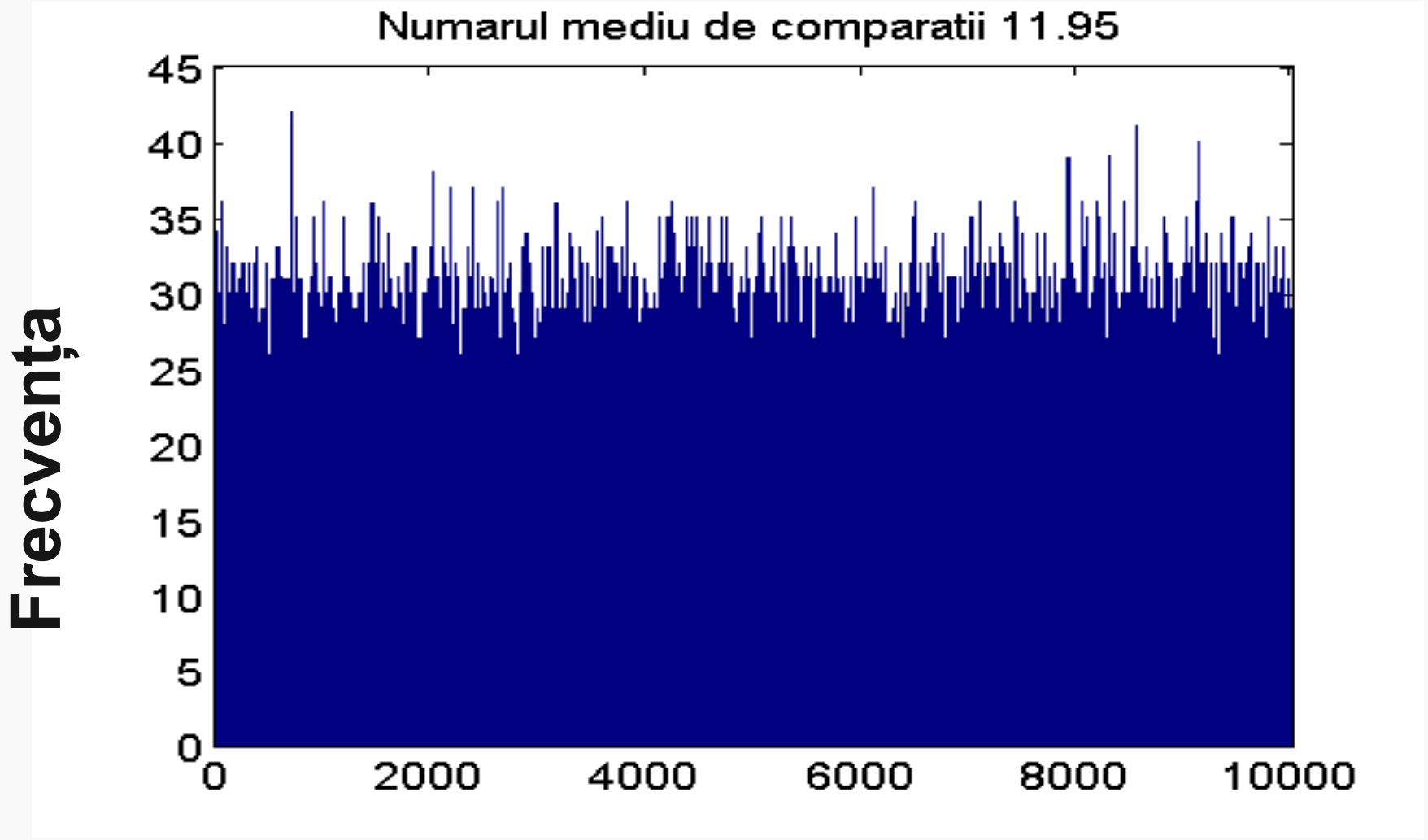
$$H(c_n c_{n-1} \dots c_0) = c_n K^n + c_{n-1} K^{n-1} + \cdots c_0 K^0 \; mod \; M$$

```
static unsigned long sdbm(unsigned char *str)
{
    unsigned long hash = 0;
    int c;

    while (c = *str++)
        hash = c + (hash << 6) + (hash << 16) - hash;

    return hash;
}
```

Hash – Stringuri – Berkley



Hash – Stringuri - Bernstein

```
unsigned long djb2(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

Hash – Stringuri - Bernstein