Hash Tables I

Lecture 10

1107186 – Estrutura de Dados

Prof. Christian Azambuja Pagot
CI / UFPB

Universidade Federal da Paraíba
Centro de Informática

Let's Start with a Problem

- Suppose that you wish to implement a **dictionary** to store (*key*, *data item*) pairs where:
  - **key** = an uppercase character.
  - **data item** = anything.
- In this case, there will be up to **26 distinct** dictionary **entries**!

Which **data structure** could be used to **implement** it?

An Array-based Approach

- Each **key** (char) can be used to compute the corresponding index of the array:
  - Array index = *h*(key)

C code excerpt:

```
unsigned int h(char c)
{
    return c – 'A';
}
```

- In this case, *h* is known as **hash function**!

A more Interesting Problem

- Now, suppose that the **keys** are the **plates** of **cars** and that they present the following format:

CCCDDDD

  - where C is a uppercase character and D is a decimal digit.

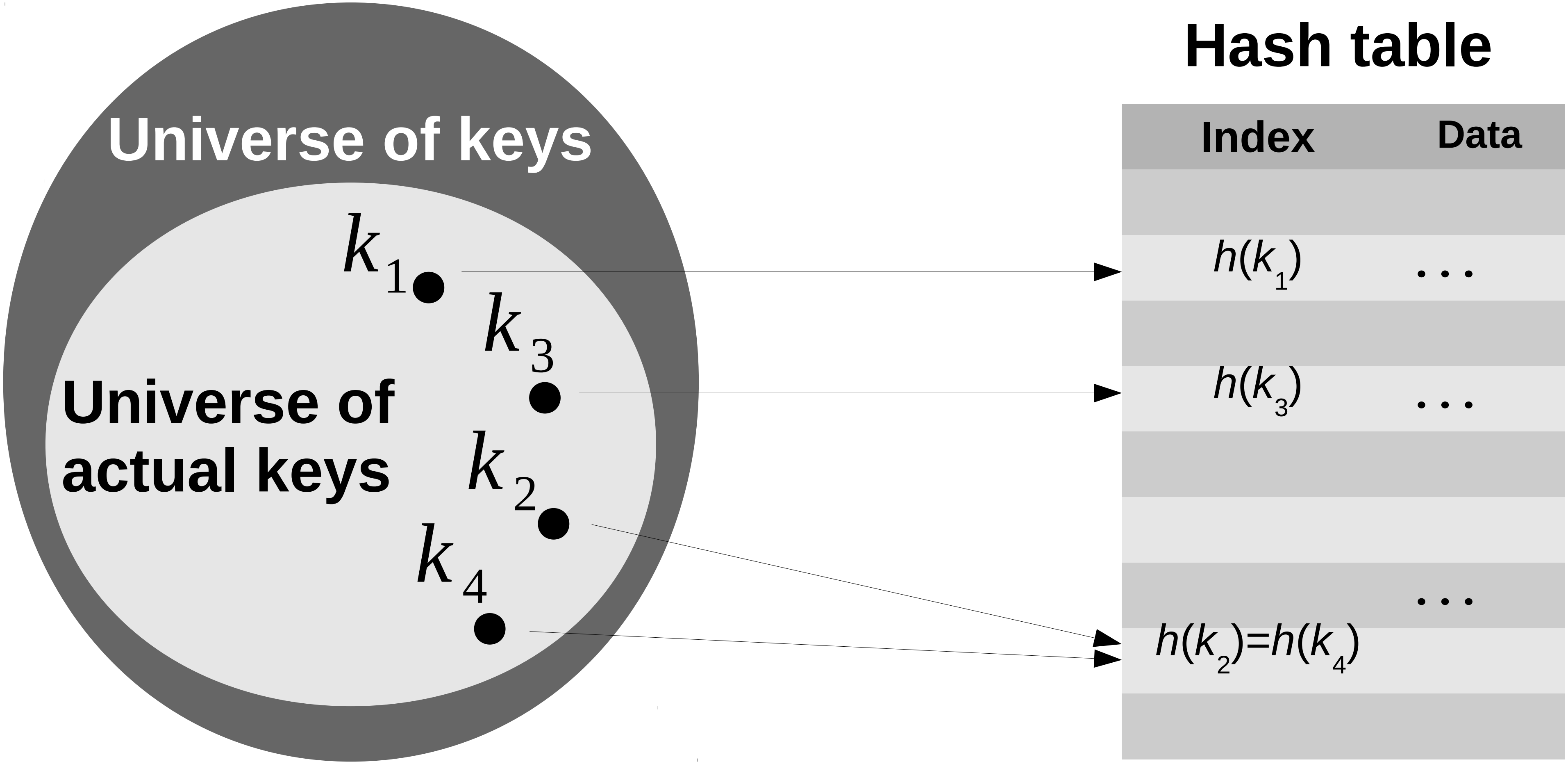
- 26*26*26*10*10*10*10 = **175.760.000 distinct possibilities**!

It can be the case that this is **too much data** for the previous **array-based implementation**!

What is a Hash Table?

- A **hash table** is a data structure used to implement **associative arrays** (also called **maps**, **dictionaries**, etc).
- Its use is effective in the case where the number of **actual keys** is **small** when compared to the **total number** of **possible keys**.

What is a Hash Table?



Applications

- It can be used when we want to keep **in-memory dictionaries**:
- **Compilers**:
  - Symbol tables (user-defined symbols).
- Fast access to records in databases.
- Web search.
- Etc.

Implementing Our Car Database with a Hash Table

- **Defining the hash function**:
  - The key (car plate) has the following format:

CCCDDDD

  - One possibility is to transform each **D** to its corresponding **digit** (with the **place value**) and each **C** to a value in the **range [0,25]** (with the **place value**).

Implementing Our Car Database with a Hash Table

- **Implementation of the hash function**:

C code excerpt:

```
unsigned int Hash(char* k)
{
    int i;
    unsigned int hash = 0;
    int place = 1;

    for (i=6; i>=3; i--) {
        hash = hash + Char2Int(k[i]) * place;
        place *= 10;
    }

    for (i=2; i>=0; i--) {
        hash = hash + (k[i] - 'A') * place;
        place *= 26;
    }

    return hash;
}
```

The **hash** may **extrapolate** the actual **size** of the **array**!

Implementing Our Car Database with a Hash Table

- **Reducing the hash to a valid index**:
  - The **hash** can be constrained to a **valid index** with the help of the **modulus operator**:

C code excerpt:

```
hash = Hash(plate);
compressed_hash = hash % HASH_TABLE_MAX_ENTRIES;
```

Size of the array

Implementing Our Car Database with a Hash Table

- **Assuming an array with 5 entries, and the following plates**:

Plate	Hash	Reduced hash
NWL5356	93715356	1
OBH2709	108492709	4
ZOW6261	172866261	1
IDD2023	54892023	3
XRJ2289	159992289	4

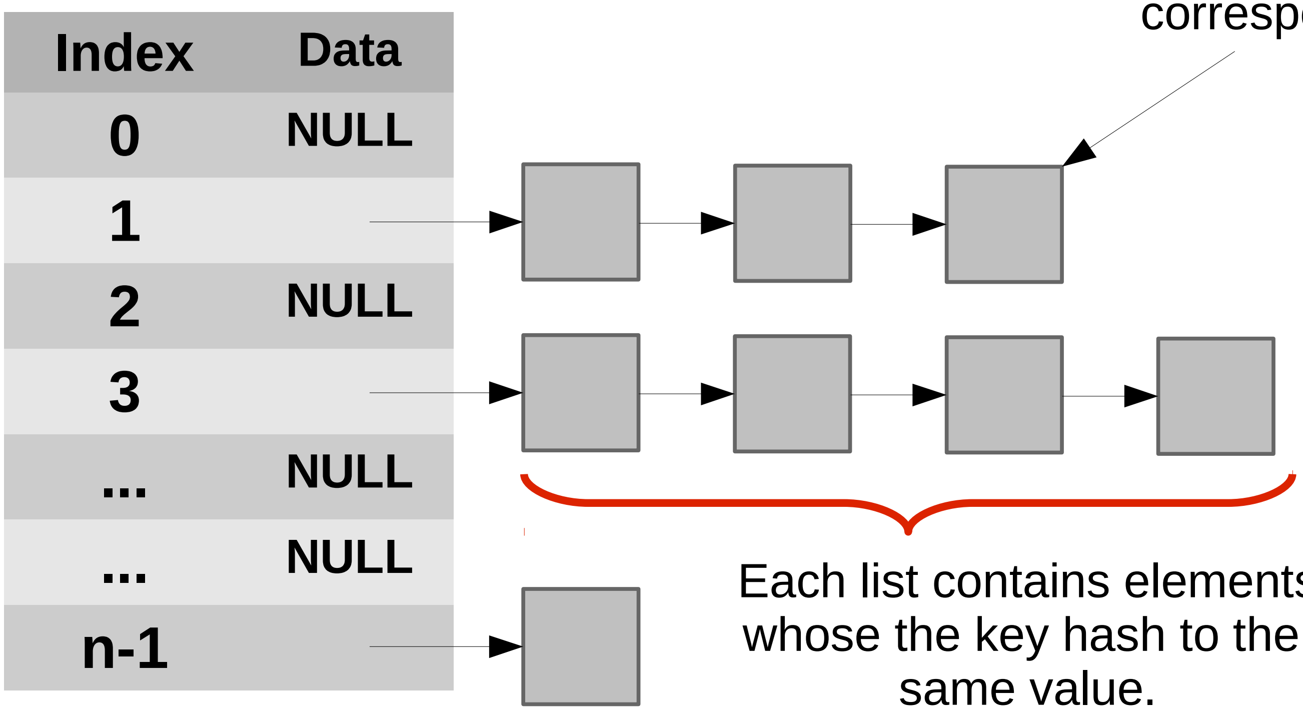
How to solve **collisions**?

Solving Collisions

- **There are several ways to solve collisions. They include:**
  - Chaining.
  - Open Addressing (linear probing).
  - Etc.

Chaining

- The **hash table** consists of an **array** whose elements are **pointers** to **lists**:



Hash table

Index	Data
0	NULL
1	
2	NULL
3	
...	NULL
...	NULL
n-1	

Each node contains the corresponding key

Each list contains elements whose the key hash to the same value.

Implementing Our Car Database with a Hash Table

- **The structure of the array and the list nodes**:

C code excerpt:

```
struct Node
{
    char plate[8];
    struct Node* next;
};
```

```
typedef struct Node* HashTable[HASH_TABLE_MAX_ENTRIES];

...
```

Implementing Our Car Database with a Hash Table

- **Creating the hash table**:

C code excerpt:

```c
void InitHashTable(HashTable h)
{
    unsigned int i;

    for (i=0; i<HASH_TABLE_MAX_ENTRIES; i++)
        h[i]= NULL;
}
```

```c
...

HashTable h;
InitHashTable(h);
...
```

Implementing Our Car Database with a Hash Table

- **Inserting data into the hash table**:

C code excerpt:

```
void InsertPlateIntoHashTable(HashTable h, char* p)
{
    ...

    hash = Hash(p);
    compressed_hash = hash % HASH_TABLE_MAX_ENTRIES;

    if (h[compressed_hash] == NULL)
    {
        h[compressed_hash] = (struct Node*) malloc (sizeof(struct Node));
        strcpy(h[compressed_hash]->plate, p);
        h[compressed_hash]->next = NULL;
    }
    else
    ...
```

Implementing Our Car Database with a Hash Table

- **Inserting data into the hash table**:

C code excerpt:

```
    ...
    else
    {
        struct Node* n = h[compressed_hash];

        while (n->next != NULL)
            n = n->next;

        n->next = (struct Node*) malloc (sizeof(struct Node));
        strcpy(n->next->plate, p);
        n->next->next = NULL;
    }

    return collision;
}
```

Open Addressing (Linear Probing)

- When collision is detected, instead of inserting the node in a list, the algorithm searches for a **free slot** in the array to **insert** the **new element**.