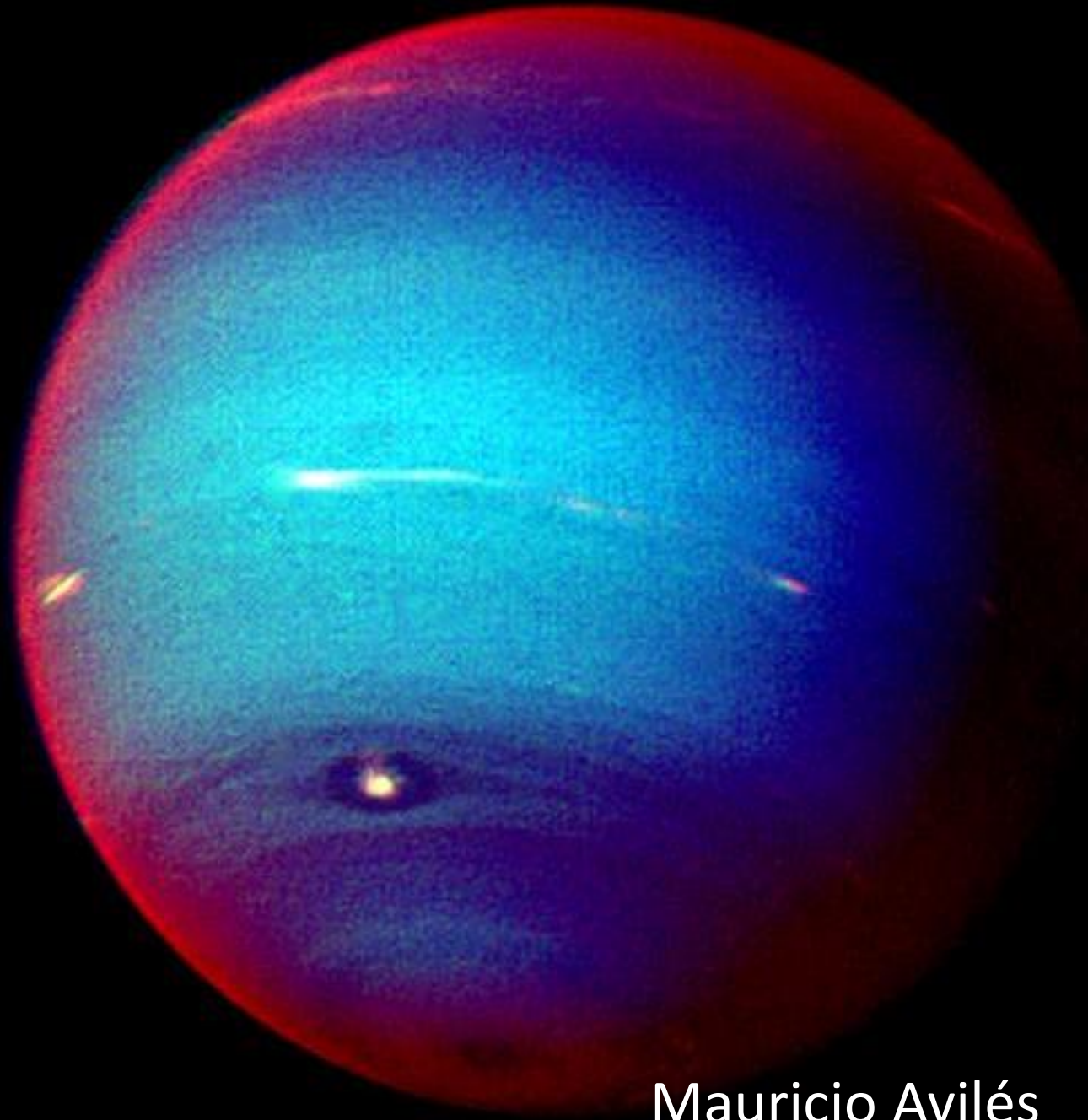


Tablas Hash

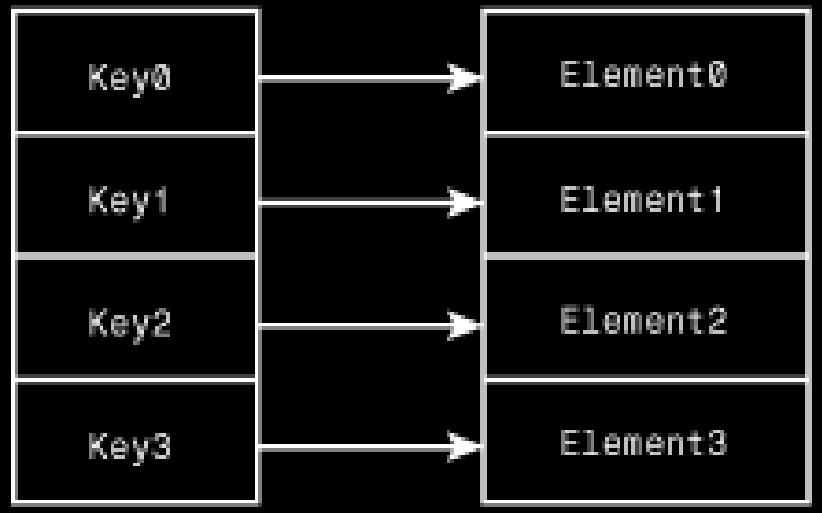
Mauricio Avilés

Contenido

- Definición
- Función hash
- Código hash
  - Mapeo a enteros
  - Mapeo a enteros con sumas
  - Polinomial
  - Corrimiento cíclico
- Función de compresión
  - División
  - Multiplicación, suma y división
- Manejo de colisiones
  - Encadenamiento
  - Sondeo
- Implementación

Tablas Hash

- El problema de asociar diferentes llaves con sus respectivos valores
- La estructura de diccionario puede implementarse de diferentes formas

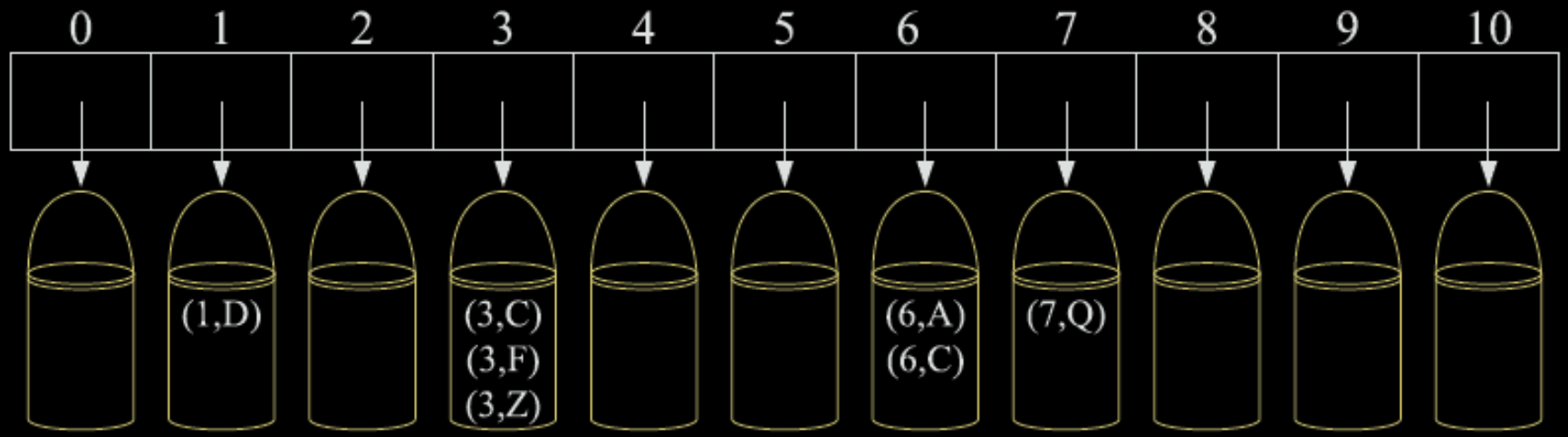


Tablas Hash

- Las tablas de hash son una de las formas más eficientes de resolver este problema
- Consta de dos componentes principales, un arreglo por casilleros y una función hash

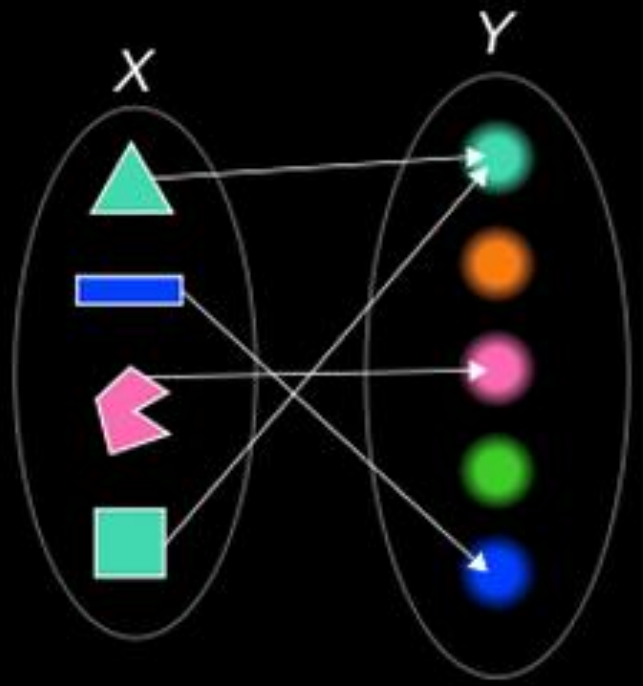
Arreglo por casilleros

- Es un arreglo *A* de tamaño *N*
- Cada celda puede contener una colección de pares llave-valor
- Similar al utilizado en el algoritmo de ordenamiento *binsort*



- Si las llaves se encuentran distribuidas en *[0, N-1]*, entonces cada casillero guardaría un valor
- Pero:
  - Se requiere memoria proporcional al tamaño de N, si N es grande el arreglo crece mucho
  - Las llaves no necesariamente van a estar en el rango especificado, puede ser que no sean enteros

- Se requiere una función que haga un buen mapeo entre las llaves y los enteros en el rango *[0, N-1]*



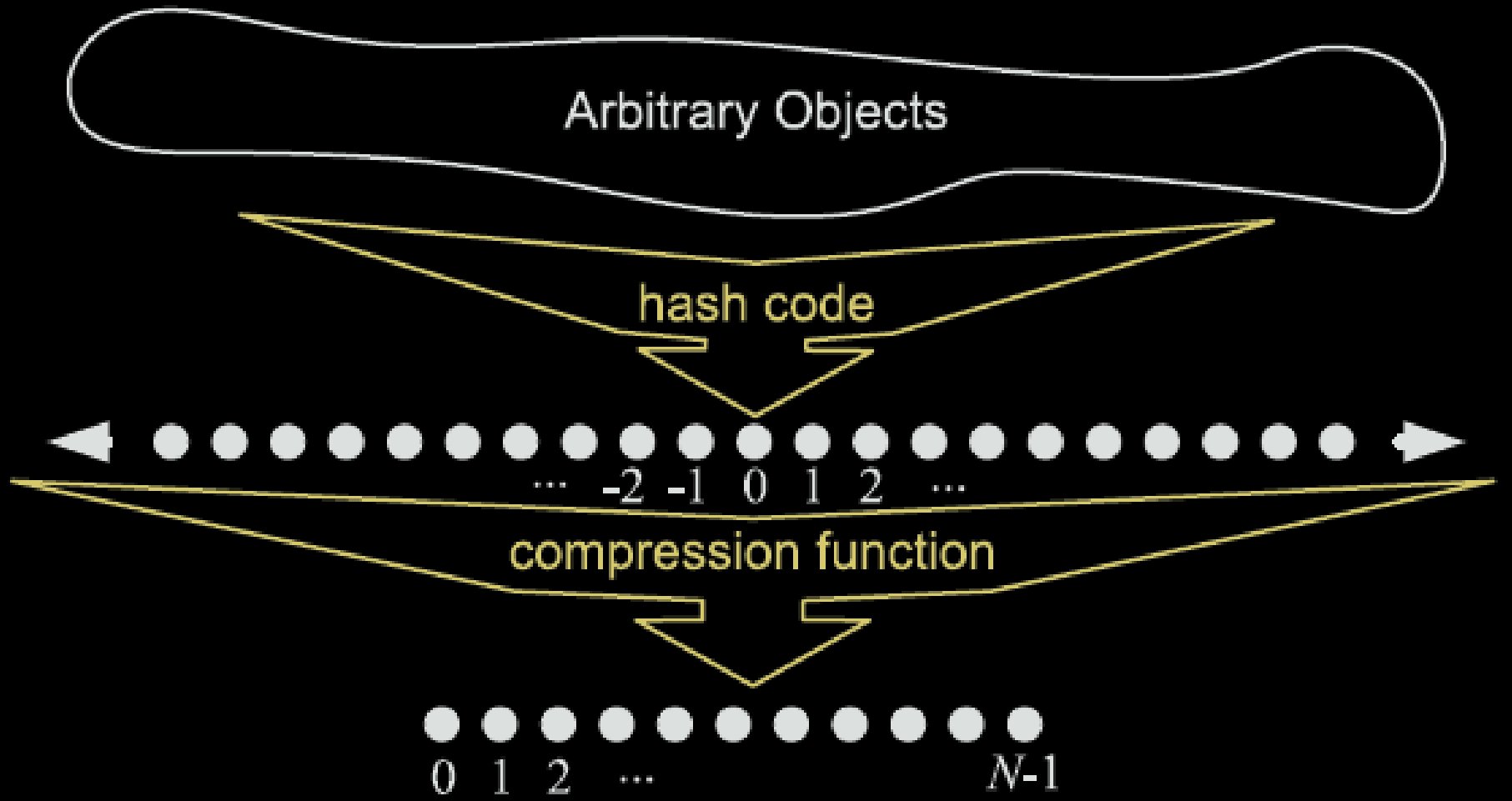
Función hash

- Es la otra parte importante de una tabla hash
- Se encarga de asignar a cada llave *k* un entero en el rango *[0, N-1]*
- La idea es usar el resultado de la función *h* aplicada a cada llave para usar el resultado como índice en el arreglo

$$A[h(k)]$$

- Dos llaves diferentes puede ser que queden mapeadas al mismo casillero de *A*
- Esto es conocido como colisión
- Si cada casilla del arreglo almacena sólo un elemento, entonces las colisiones son un problema
- Existen diferentes formas de resolver las colisiones
- En general: una buena función hash minimiza colisiones y es rápida y fácil de calcular

- La función hash está compuesta por dos partes:
  - Mapeo de la llave *k* a un número entero, conocido como código hash
  - Mapeo del código hash a un entero que se encuentre entre el rango de índices *[0, N-1]*, conocido como función de compresión


Arbitrary Objects
hash code
... -2 -1 0 1 2 ...
compression function
0 1 2 ... N-1

Código hash

- Tomar cualquier llave *k* y asignarle un valor entero
- No es necesario que se encuentre en *[0, N-1]*, puede ser negativo
- Debe evitar colisiones, si la función hash genera colisiones, la función de compresión no puede evitarlas

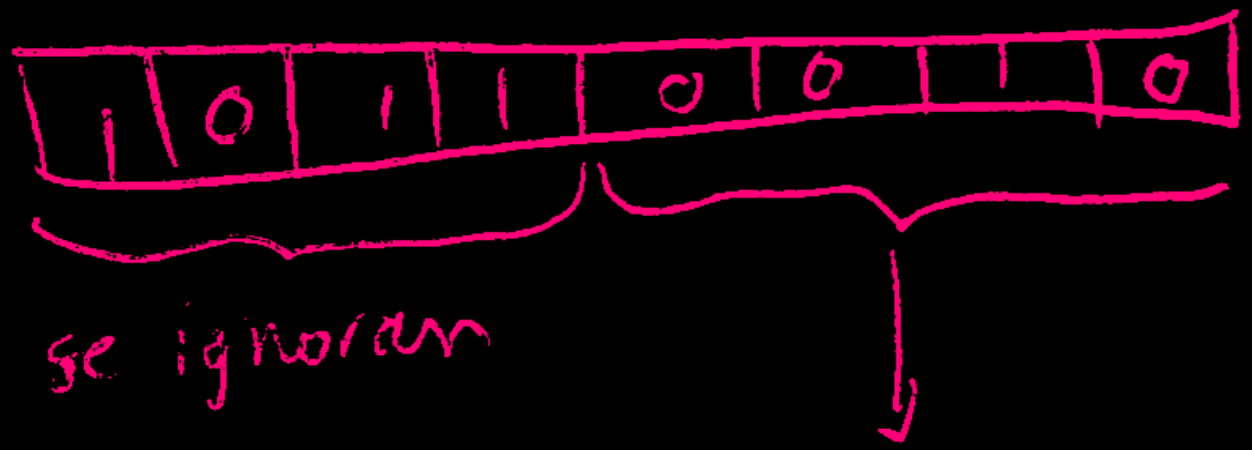
Código hash en C++

- Para calcular el código hash de una llave, se hace necesario conocer cuántos bits utiliza en memoria el tipo utilizado en la llave
- Si la llave es de tipo *char*, *int* o *float*, este dato puede conocerse con *numeric_limits<T>.digits*

Mapeo a un número entero

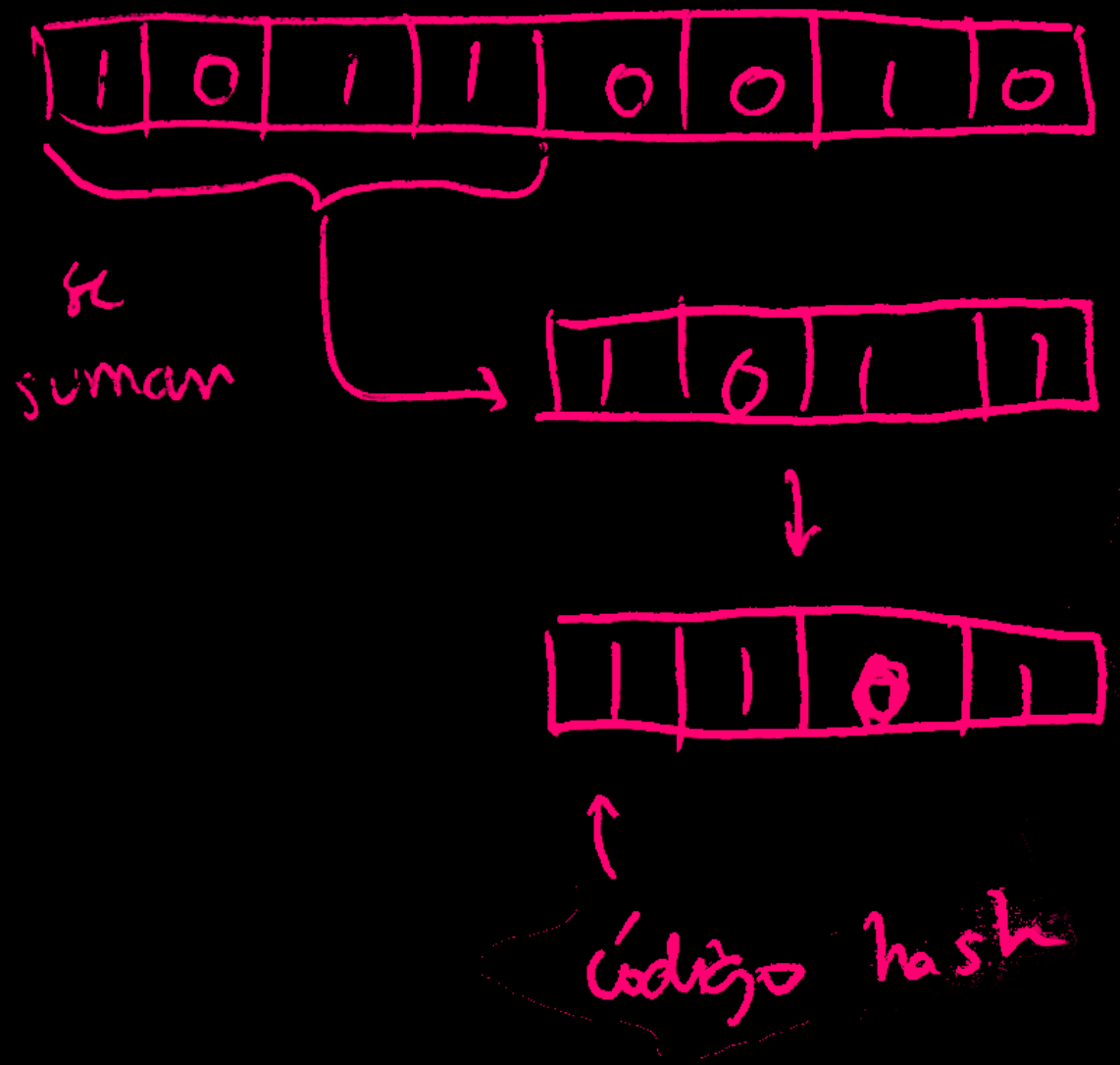
- Para algunos tipos de datos lo más simple es interpretar el valor como un entero (*char*, *short*, *int*)
- Otros tipos como el *long* usan más bits que el *int*; se pueden ignorar los bits que sobran, pero puede causar colisiones

valor original de la llave



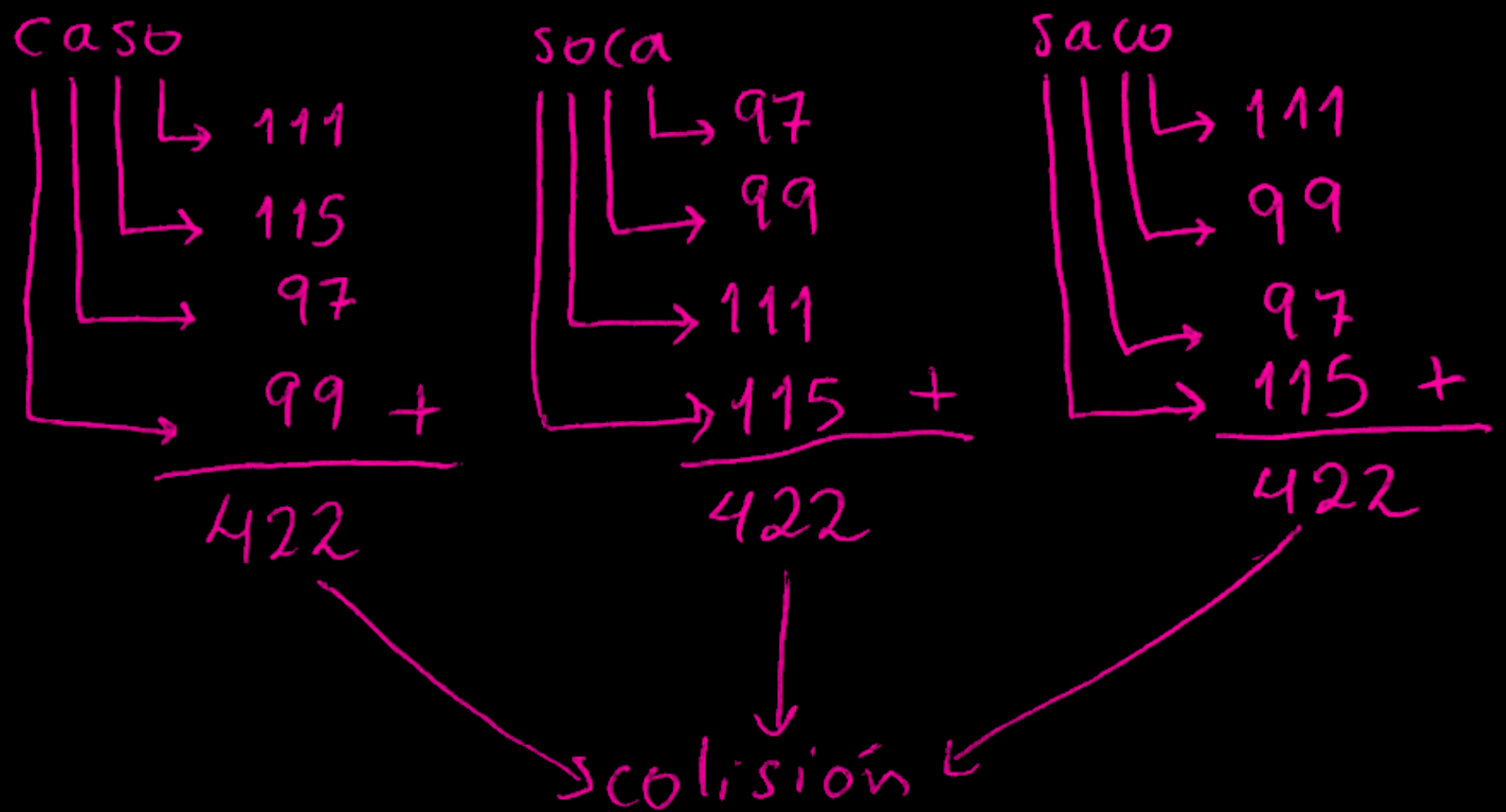
Mapeo a un número entero

- Una forma de no ignorarlos puede ser sumar los bits que sobran con la parte inferior y así evitar colisiones


1 0 1 1 0 0 1 0
se suman
1 0 1 1
1 1 0 1
código hash

Códigos hash polinomiales

- El código generado por medio de sumas no es apto para llaves tipo string
- Los strings “caso”, “soca”, “saco” y “cosa” darían como resultado el mismo código hash


caso
111
115
97
99 +
422
soca
97
99
111
115 +
422
saco
111
99
97
115 +
422
colisión

Códigos hash polinomiales

- El problema de las colisiones soluciona multiplicando cada carácter por un valor (a) según su posición, similar a un sistema posicional

Caso

$111 \times a^0$

$115 \times a^1$

$97 \times a^2$

$99 \times a^3$ +

valor_hash

- Matemáticamente hablando, esto es un polinomio, donde *a* toma los componentes del string como coeficientes
- Es posible que la suma de los productos produzca un *overflow* en el valor entero, pero esto se ignora porque el objetivo es esparcir las llaves y evitar colisiones
- Según estudios experimentales, los valores que menos colisiones causan al trabajar con strings son:
  - 33, 37, 39, 41

Código hash de corrimiento cíclico

- Esta variante consiste en tomar el valor de cada carácter y hacer dos corrimientos de bits
- Hacia la izquierda y hacia la derecha con dos valores diferentes
- Se unen por medio de un *or* o *xor* de bits
- El resultado obtenido se utiliza como código hash

Funciones de compresión

- El código hash no puede usarse como índice
- Debe comprimirse para estar en *[0..N-1]*
- Esta es la tarea de la función de compresión
- Métodos:
  - Por división
  - Multiplicación, suma y división

Compresión por división

$$|k| \bmod N$$

- Distribuye todos los valores hash dentro del rango disponible, donde *N* es el tamaño del arreglo
- Este método funciona mejor si *N* es un número primo

Multiplicación, suma y división

$$|ak + b| \bmod N$$

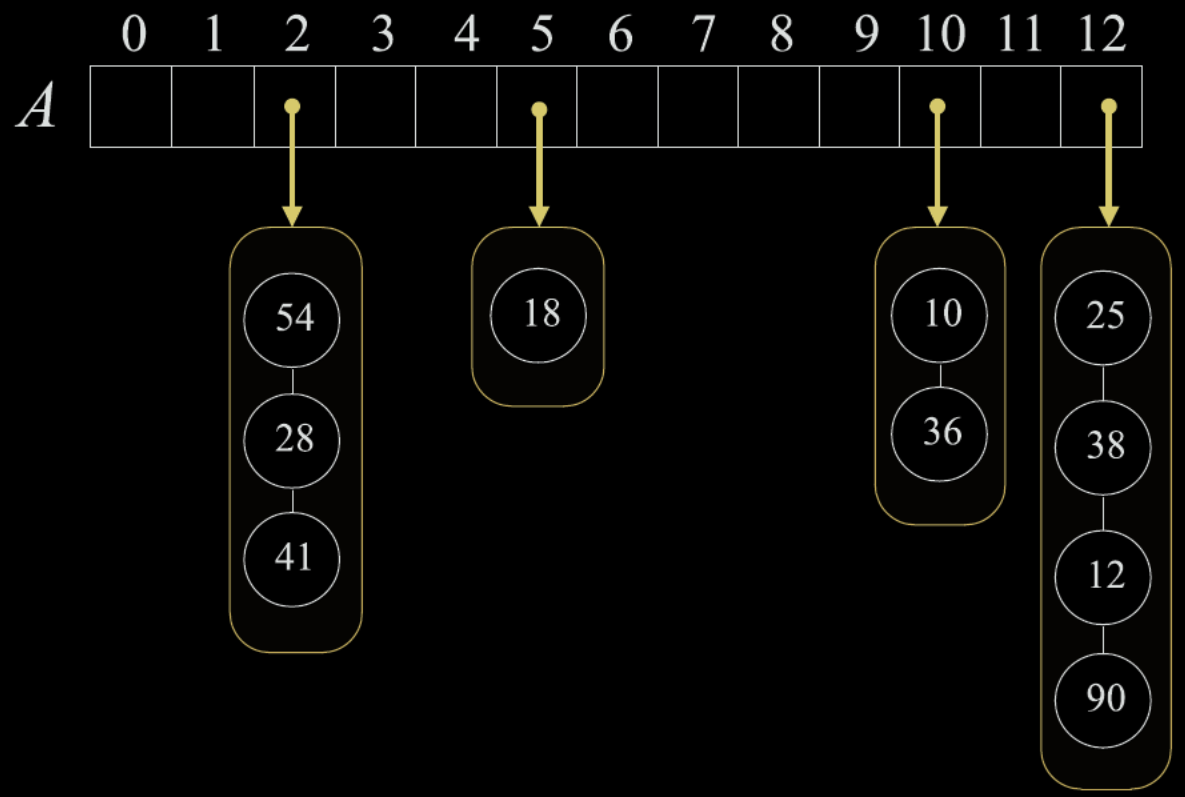
- Ayuda a distribuir posibles patrones en las llaves
- *a* y *b* se escogen aleatoriamente, deben ser positivos y *a* diferentes de cero
- *N* también debe ser primo

Manejo de colisiones

- Dos estrategias principales
  - Por encadenamiento
  - Por sondeo

Encadenamiento

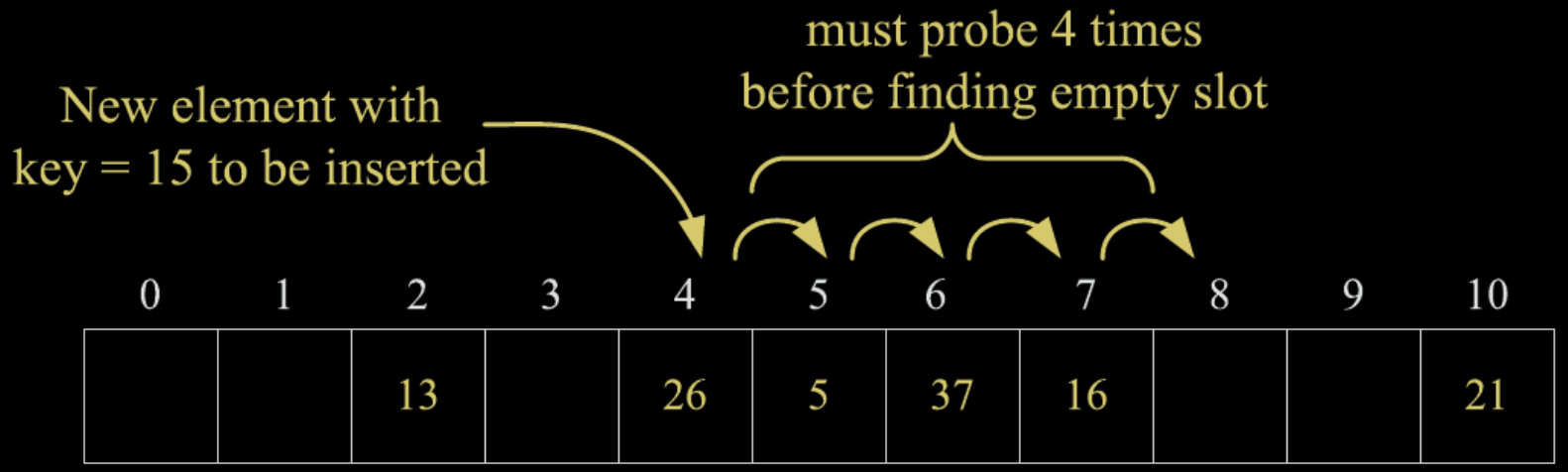
- Consiste en mantener una lista en cada espacio del arreglo, que contiene las llaves que generan un mismo código hash



- *find(k)*
  - Retorna el valor asociado a una llave o un valor nulo en caso de no encontrarla
  - *return A[h(k)].find(k)*
    - Se delega la búsqueda a la lista ubicada en *A[h(k)]*
- *put(k, v)*
  - Agrega un nuevo par llave-valor a la estructura
  - *A[h(k)].put(k, v)*
    - Se delega la inserción a la lista ubicada en *A[h(k)]*
- *erase(k)*
  - Elimina el elemento con la llave *k* de la estructura
  - *A[h(k)].erase(k)*
    - Se delega el borrado a la lista ubicada en *A[h(k)]*

Por sondeo

- Consiste en utilizar un arreglo plano
- Si el código hash genera una colisión, entonces se busca una celda disponible a partir de la actual



- Tipos:
  - Lineal: búsqueda de espacio libre de 1 en 1
  - Cuadrático: hacer saltos en intervalos j^2 con $j=0,1,2…$
  - Doble hashing: utilizar otra función de hash $h'(k)$ para aplicarla sobre códigos que causen colisiones
- Se complican los métodos de búsqueda y borrado
- Si no existen restricciones de memoria, la opción de encadenamiento es mejor que el sondeo

Implementación

- La implementación a continuación tiene las siguientes características
  - Manejo de colisiones por encadenamiento
  - Código hash polinomial y corrimiento cíclico (se usa uno a la vez)
  - Compresión por multiplicación, suma y división

```cpp
#include "KVPair.h"
#include "LinkedList.h"
#include <string>
#include <stdlib.h>
#include <stdio.h>

using namespace std;

template <typename K, typename V>
class HashTable : public Dictionary<K, V> {
private:
    LinkedList<KVPair<K, V> > *buckets;
    int nbuckets;
```

```
int h(K key) {
    return compress(hashCodeCyclicShift(key));
}
```

```
int hashCodePolynomial(K pKey) {
    int a = 33;
    int result = 0;
    char* bytes = reinterpret_cast<char*>(&pKey);
    for (unsigned int i = 0; i < sizeof(K); i++) {
        result += bytes[i] * pow(a, i);
    }
    return result;
}
```

```
template <typename T>
int hashCodeCyclicShift(T pKey) {
    int result = 0;
    char* bytes = reinterpret_cast<char*>(&pKey);
    for (unsigned int i = 0; i < sizeof(pKey); i++) {
        result = (result << (7)) ^ (result >> (25));
        result += (int) bytes[i];
    }
    return result;
}
int hashCodeCyclicShift(string pKey) {
    int result = 0;
    for (unsigned int i = 0; i < pKey.length(); i++) {
        result = (result << (7)) ^ (result >> (25));
        result += (int) pKey.at(i);
    }
    return result;
}
```

```
int compress(int pHashCode) {
    int a = 1097;
    int b = 1279;
    return abs(a * pHashCode + b) % nbuckets;
}
```

```
void checkNotExisting(K key) throw (runtime_error) {
    int pos = h(key);
    if (buckets[pos].contains(key))
        throw runtime_error("Key not found.");
}
void checkExisting(K key) throw (runtime_error) {
    int pos = h(key);
    if (!buckets[pos].contains(key)) {
        throw runtime_error("Duplicated key.");
    }
}
```

```
public:
    HashTable(int nbuckets = 1021) {
        buckets = new   LinkedList<KVPair<K, V> >[nbuckets];
        this->nbuckets = nbuckets;
    }
    ~HashTable() {
        delete [] buckets;
    }
```

```
void insert(K key, V value) {
    checkNotExisting(key);
    KVPair<K, V> p(key, value);
    buckets[h(key)].append(p);
}
```

```
V remove(K key) {
    checkExisting(key);
    KVPair<K, V> p(key);
    int pos = h(key);
    int listPos = buckets[pos].indexOf(p);
    buckets[pos].goToPos(listPos);
    cout << buckets[pos].getElement().getKey() << endl;
    p = buckets[pos].remove();
    return p.getValue();
}
```

```
V getValue(K key) {
    checkExisting(key);
    int pos = h(key);
    KVPair<K, V> p(key);
    int listPos = buckets[pos].indexOf(p);
    buckets[pos].goToPos(listPos);
    p = buckets[pos].getElement();
    return p.getValue();
}
```

```
void setValue(K key, V value) {
    checkExisting(key);
    int pos = h(key);
    KVPair<K, V> p(key);
    int listPos = buckets[pos].indexOf(p);
    buckets[pos].goToPos(listPos);
    buckets[pos].remove();
    p.setValue(value);
    buckets[pos].insert(p);
}
```

```
void clear() {
    for (int i = 0; i < nbuckets; i++) {
        buckets[i].clear();
    }
}
bool contains(K key) {
    int pos = h(key);
    return buckets[pos].contains(key);
}
```

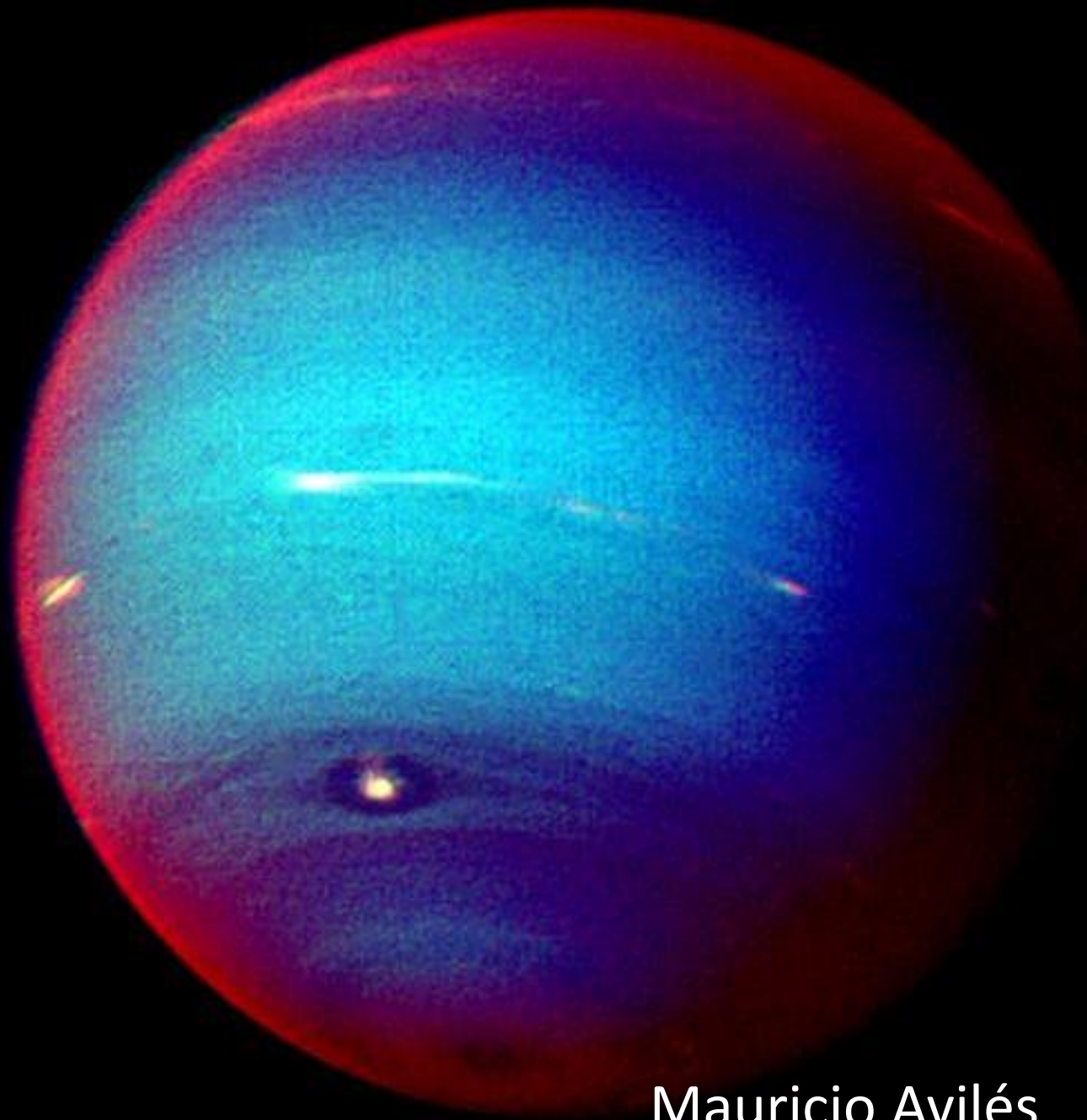
```
List<K>* getKeys() {
    List<K> *keys = new DLinkedList<K>();
    for (int i = 0; i < nbuckets; i++) {
        buckets[i].goToStart();
        while (!buckets[i].atEnd()) {
            KVPair<K, V> p;
            p = buckets[i].getElement();
            keys->append(p.getKey());
            buckets[i].next();
        }
    }
    return keys;
}
```

```
List<V>* getValues() {
    List<V> *values = new DLinkedList<V>();
    for (int i = 0; i < nbuckets; i++) {
        buckets[i].goToStart();
        while (!buckets[i].atEnd()) {
            KVPair<K, V> p;
            p = buckets[i].getElement();
            values->append(p.getValue());
            buckets[i].next();
        }
    }
    return values;
}
```

```
    int getSize() {
        int size = 0;
        for (int i = 0; i < nbuckets; i++) {
            size += buckets[i].getSize();
        }
        return size;
    }
};
```

Lecturas

- Sección 9.4
  - Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.
- Sección 9.2.7
  - Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.
- Capítulo 14
  - Joyanes, Aguilar, & Martínez. Estructura de datos en C ++. Madrid: McGraw-Hill Interamericana. 2007.



Tablas Hash

Mauricio Avilés