ESTRUCTURAS DE DATOS

DICCIONARIOS

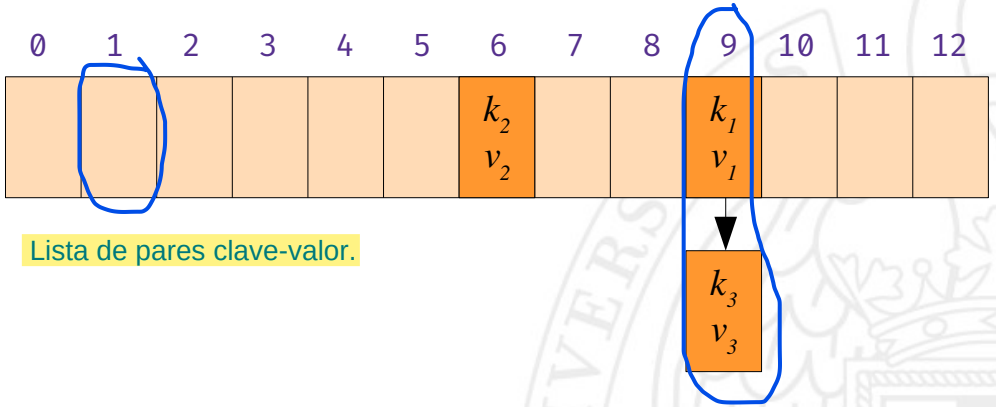
Tablas *hash* abiertas

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Objetivo

Cada cajón o bucket contiene una LISTA DE ENTRADAS ya que si había colisiones lo añadía al mismo cajón.

- Implementar el TAD Diccionario mediante una tabla *hash* abierta.



Lista de pares clave-valor.

Varias claves que apuntan a la misma posición del vector entonces las metíamos en el mismo cajón. Esto ocurre cuando al calcular su código hash y hacer el modulo de N (posición i del vector), su resultado i está ocupado. Ocurría cuando queríamos hacer una INSERCIÓN.

Recordatorio: TAD Diccionario

Del tema anterior que veíamos distintas funciones para el TAD diccionario.

- Constructoras:
  - Crear un diccionario vacío: ***create_empty***
- Mutadoras:
  - Añadir una entrada al diccionario: ***insert***
  - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
  - Saber si existe una entrada con una clave determinada: ***contains***
  - Saber el valor asociado con una clave: ***at*** Sobrecargábamos el operador [] para buscar o insertar.
  - Saber si el diccionario está vacío: ***empty*** esto era si size()==0
  - Saber el número de entradas del diccionario: ***size***

Clase MapHash: interfaz pública

En java utilizamos un Hashmap <>()

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
  MapHash();
  MapHash(const MapHash &other);
  ~MapHash();

  void insert(const MapEntry &entry);
  void erase(const K &key);

  bool contains(const K &key) const;
  const V & at(const K &key) const;
  V & at(const K &key);
  V & operator[](const K &key);

  int size() const;
  bool empty() const;

  MapHash & operator=(const MapHash &other);
  void display(std::ostream &out) const;

private:
  // ...
};
```

El que vimos nosotros anteriormente fue un MapEntry

Paramétrica respecto a la clave, el valor y el objeto función que nos define la función hash.

```
struct MapEntry {
  K key;
  V value;

  MapEntry(K key, V value);
  MapEntry(K key);
};
```

La clave y el valor.

constructores para solo clave, o clave y valor.

Sobrecarga del operador =

Clase MapHash: representación privada

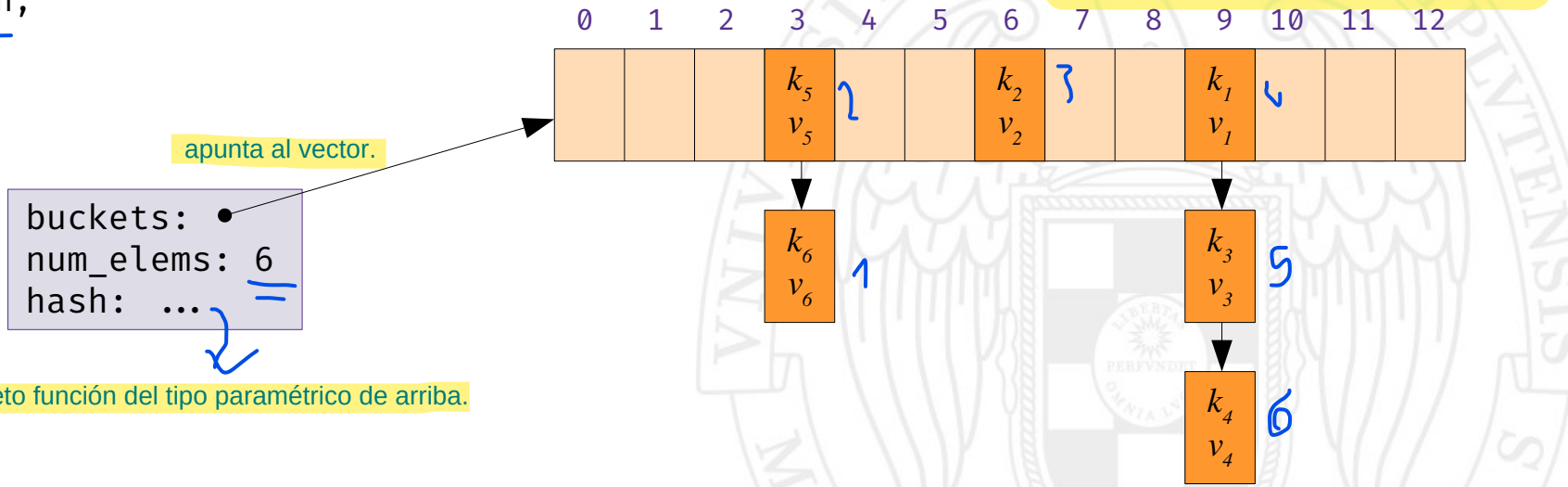
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {


private:
  using List = std::forward_list<MapEntry>;

  List *buckets;
  int num_elems;
  Hash hash;

};
```

Implementación de las LISTAS ENLAZADAS SIMPLES.

Las componentes van a ser datos de tipo MapEntry

Lo creamos en el heap y por eso lo crea mediante un puntero.

Si hay una operación de inserción insertaríamos al principio de la lista.



Clase MapHash: constructores

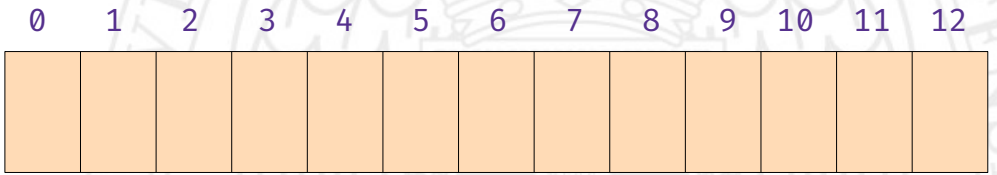
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
  MapHash(): num_elems(0), buckets(new List[CAPACITY]) { };

  MapHash(const MapHash &other): num_elems(other.num_elems),
                                 hash(other.hash),
                                 buckets(new List[CAPACITY]) {
    std::copy(other.buckets, other.buckets + CAPACITY, buckets);
  };

  ~MapHash() {
    delete[] buckets;
  }

private:
  List *buckets;
  int num_elems;
  Hash hash;
  //  ...
};
```

constante declarada anteriormente.

En este caso valdría 13

CONSTRUCTOR DE COPIA

this.buckets, creas una lista del tamaño de la lista pasada como parámetro.



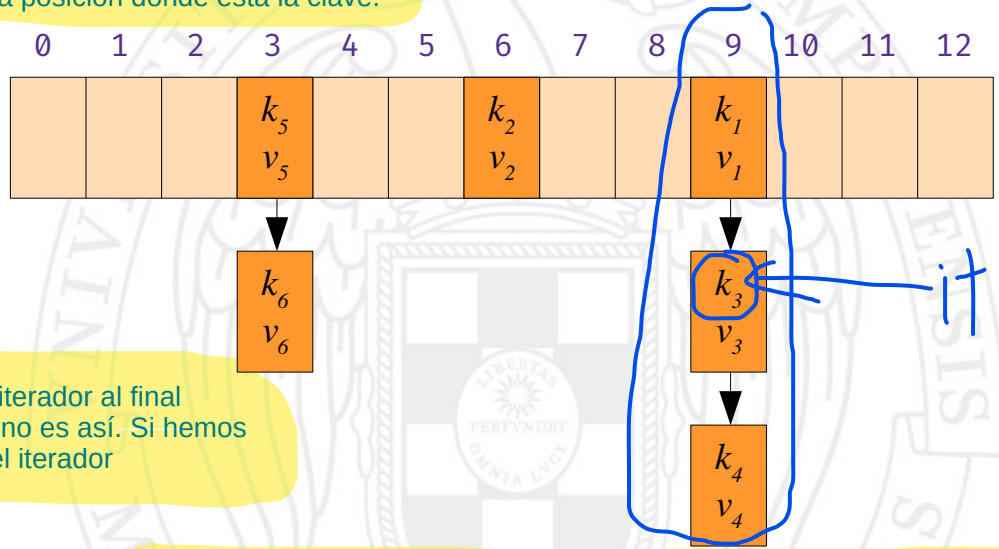
Clase MapHash: búsqueda

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  const V & at(const K &key) const {
    int h = hash(key) % CAPACITY;
    const List &list = buckets[h];

    auto it = find_in_list(list, key);

    assert (it != list.end());
    return it->value;
  }

private:
  List *buckets;
  int num_elems;
  Hash hash;
  //  ...
};
```

módulo entre el código hash y la capacity del vector.

Referencia a la lista que está en la posición 9 en este caso

iterador a la posición donde está la clave.



Si no lo encontramos devuelve un iterador al final de la lista. Comprobamos que eso no es así. Si hemos encontrado un valor, devolvemos el iterador

*it es de tipo MapEntry, entonces nosotros queremos devolver el value asociado a ese MapEntry

Es lo mismo que tener: (*it).value

Clase MapHash: búsqueda

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  const V & at(const K &key) const {
    int h = hash(key) % CAPACITY;
    const List &list = buckets[h];

    auto it = find_in_list(list, key);

    assert (it != list.end());
    return it->value;
  }

private:
  List *buckets;
  int num_elems;
  Hash hash;
  // ...
};
```

```
List::const_iterator find_in_list(const List &list, const K &key) {
  auto it = list.begin();
  while (it != list.end() && it->key != key) {
    ++it;
  }
  return it;
}
```

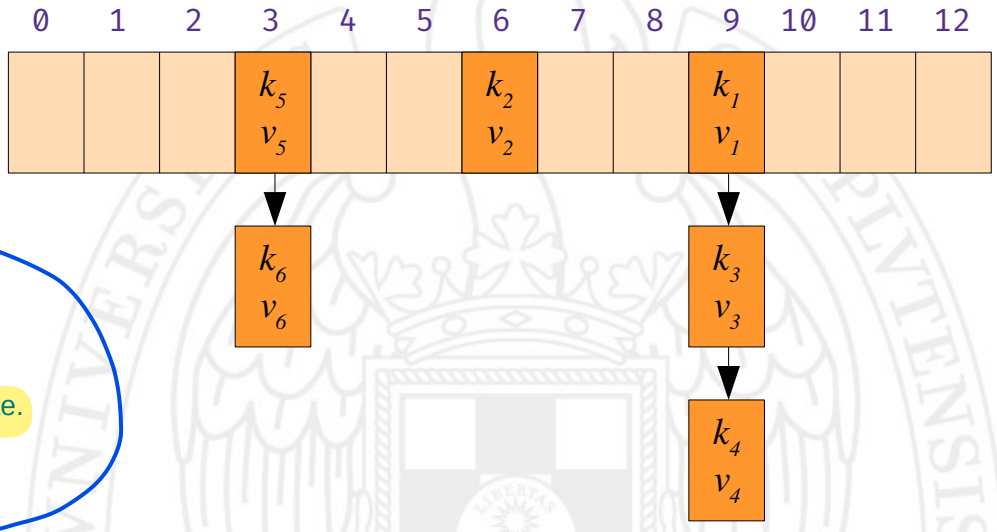
Búsqueda lineal a lo largo de una lista.

Devuelve o final o donde esté el elemento.

Clase MapHash: inserción

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  void insert(const MapEntry &entry) {
    int h = hash(entry.key) % CAPACITY;
    List &list = buckets[h];

    auto it = find_in_list(list, entry.key);

    if (it == list.end()) {
      list.push_front(entry);
      num_elems++;
    }
  }


private:
  List *buckets;
  int num_elems;
  Hash hash;
  // ...
};
```

Parecido a la búsqueda.

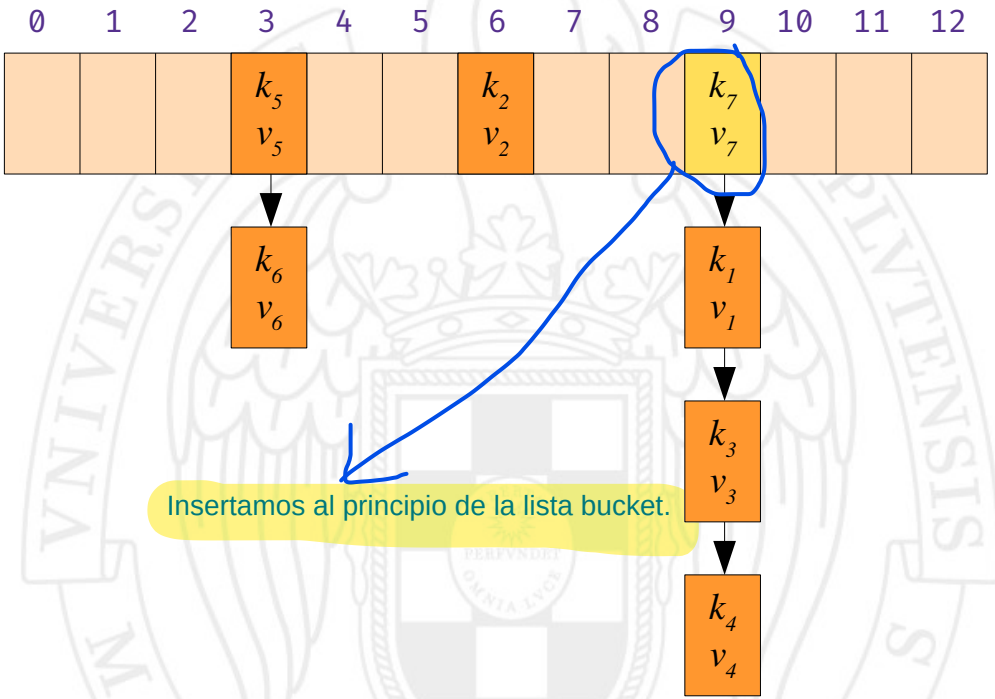


Creo que esto tenía coste constante.

Solamente insertamos si la clave no existe en esa posición.

Clase MapHash: inserción

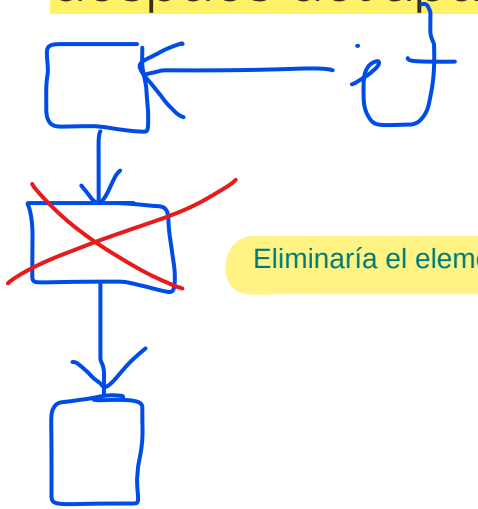
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  void insert(const MapEntry &entry) {
    int h = hash(entry.key) % CAPACITY;
    List &list = buckets[h];

    auto it = find_in_list(list, entry.key);

    if (it == list.end()) {
      list.push_front(entry);
      num_elems++;
    }
  }


private:
  List *buckets;
  int num_elems;
  Hash hash;
  // ...
};
```



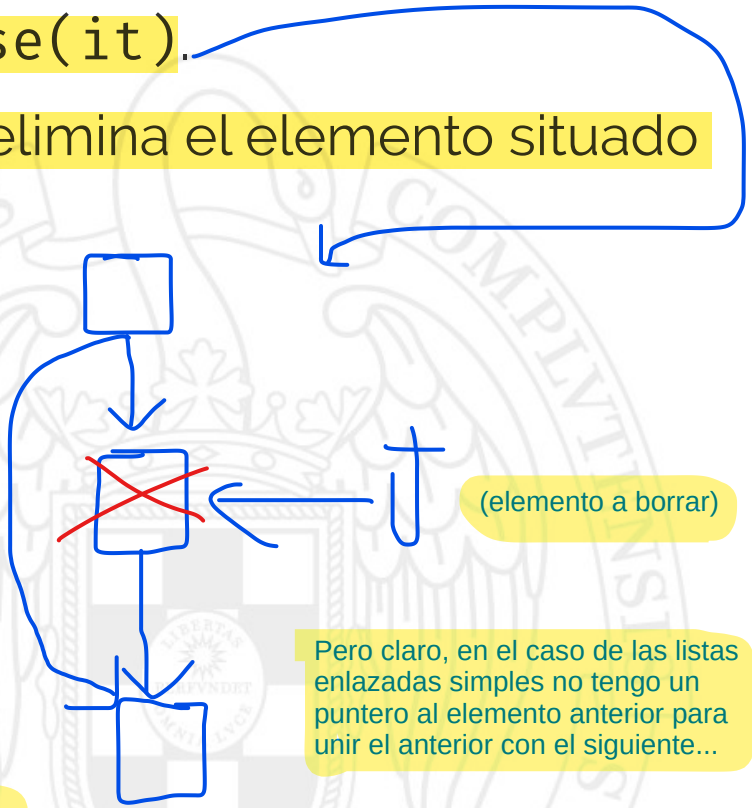
Clase MapHash: borrado

- Similar a la inserción.
- La clase `forward_list` no tiene método `erase(it)`.
- Pero sí tiene método `erase_after(it)`, que elimina el elemento situado después del apuntado por el iterador.

buscaríamos la clave a borrar, peeeero.... tenemos un pequeño problema.



Eliminaría el elemento posterior.



(elemento a borrar)

Pero claro, en el caso de las listas enlazadas simples no tengo un puntero al elemento anterior para unir el anterior con el siguiente...

Hacemos uso de este método erase_after, para poder enganchar el anterior al que hemos borrado con el posterior

Clase MapHash: borrado

```
void erase(const K &key) {
  int h = hash(key) % CAPACITY;
  List &list = buckets[h];
  if (!list.empty()) {
    if (list.front().key == key) {
      list.pop_front();
      num_elems--;
    } else {
      auto it_prev = list.begin();
      auto it_next = ++list.begin();

      while (it_next != list.end() && it_next->key != key) {
        it_prev++;
        it_next++;
      }
      if (it_next != list.end()) {
        list.erase_after(it_prev);
        num_elems--;
      }
    }
  }
}
```

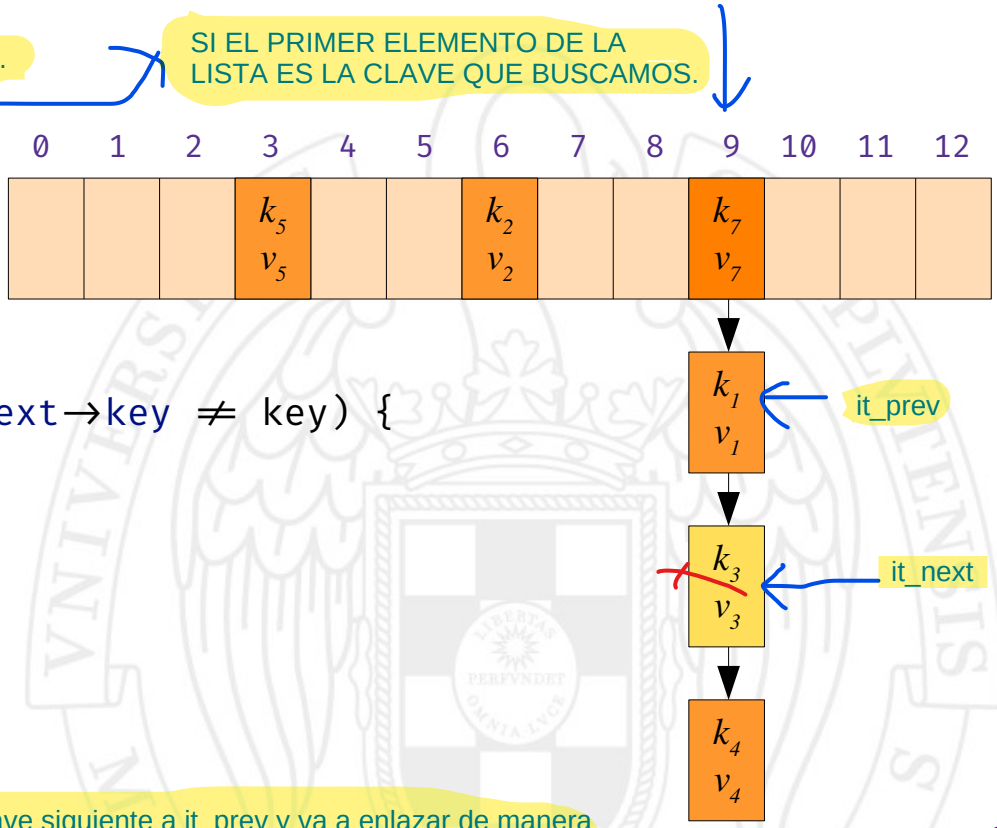
Es la que queremos eliminar.

Si la lista ES VACÍA, no haría nada.

SI EL PRIMER ELEMENTO DE LA LISTA ES LA CLAVE QUE BUSCAMOS.

SI NO LO ES

Avanzaríamos ambos iteradores.



Ese método borra la clave siguiente a it_prev y va a enlazar de manera correcta los elementos de la lista.

Clase MapHash: borrado

```
void erase(const K &key) {
  int h = hash(key) % CAPACITY;
  List &list = buckets[h];
  if (!list.empty()) {
    if (list.front().key == key) {
      list.pop_front();
      num_elems--;
    } else {
      auto it_prev = list.begin();
      auto it_next = ++list.begin();

      while (it_next != list.end() && it_next->key != key) {
        it_prev++;
        it_next++;
      }
      if (it_next != list.end()) {
        list.erase_after(it_prev);
        num_elems--;
      }
    }
  }
}
```