ESTRUCTURAS DE DATOS

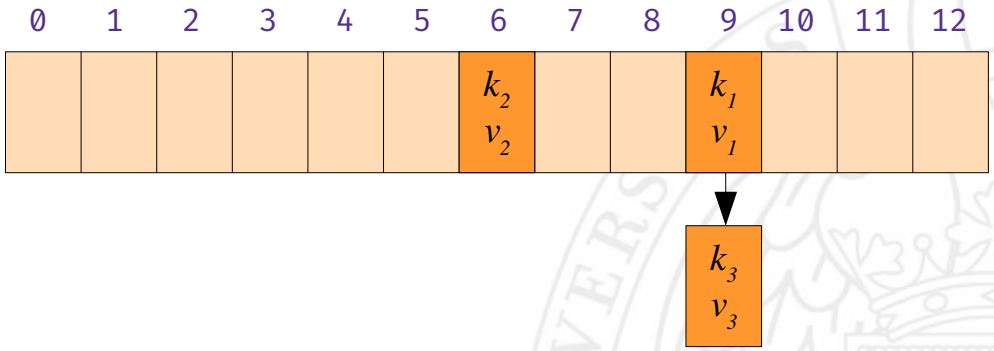
DICCIONARIOS

Tablas *hash* abiertas

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

Objetivo

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



Recordatorio: 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***
  - Saber si el diccionario está vacío: ***empty***
  - Saber el número de entradas del diccionario: ***size***

Clase MapHash: interfaz pública

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

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

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

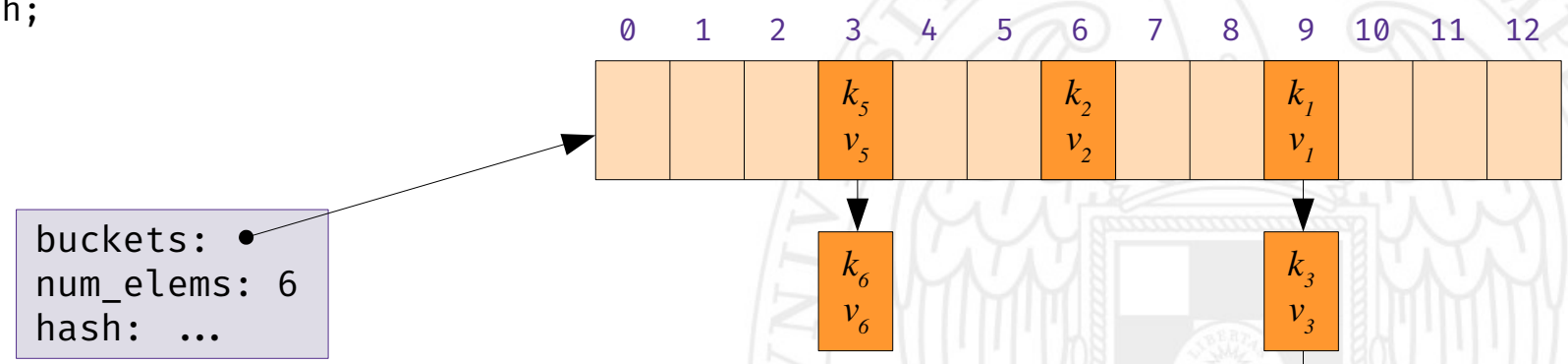
Clase MapHash: representación privada

```
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;

};
```



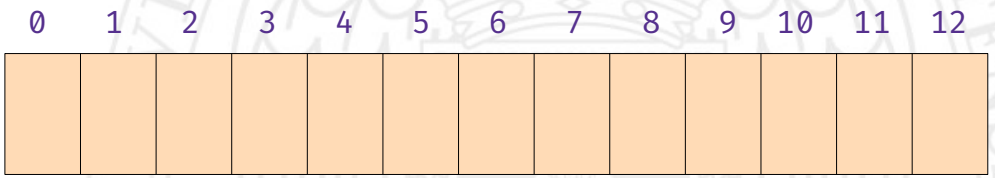
Clase MapHash: constructores

```
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;
  // ...
};
```



Clase MapHash: búsqueda

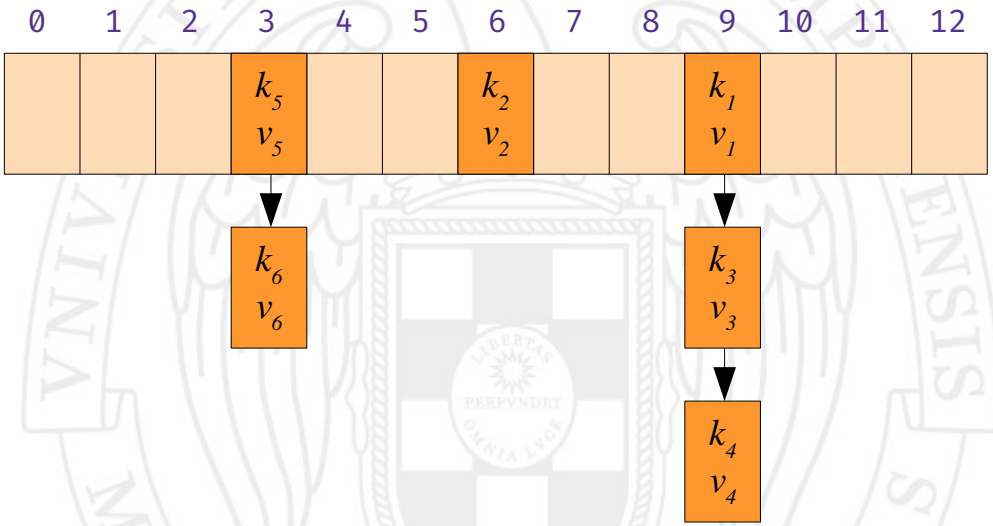
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  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;
  // ...
};
```



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;
}
```

Clase MapHash: inserción

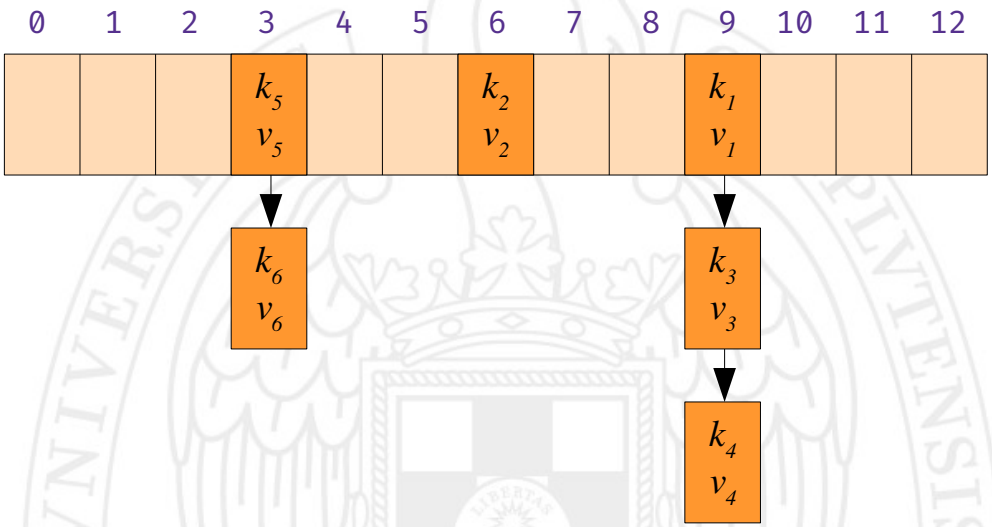
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  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: inserción

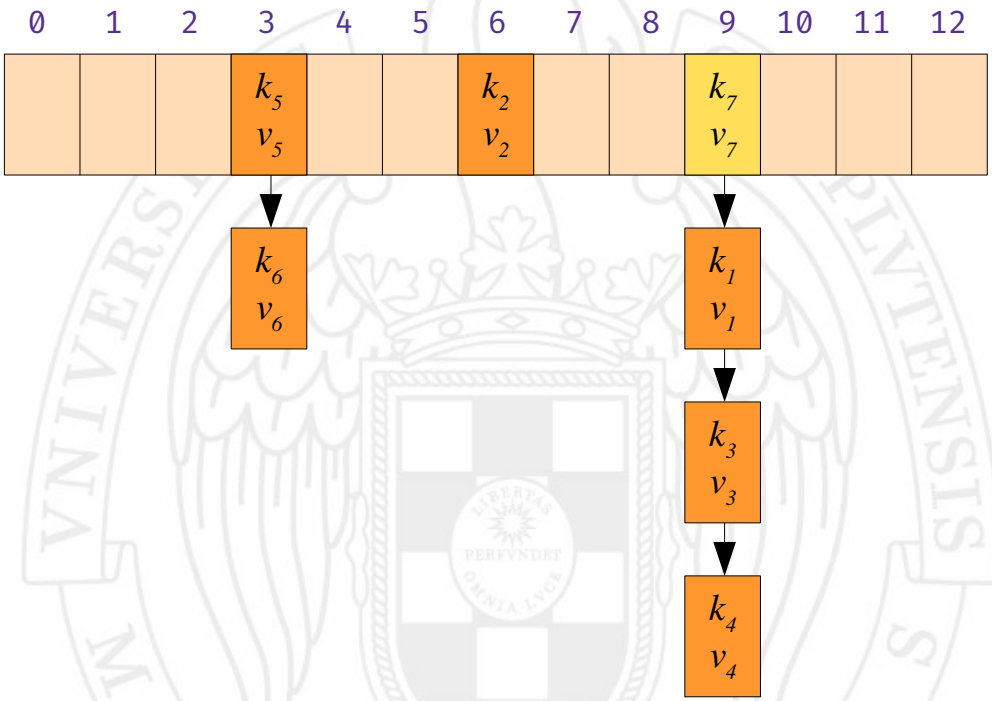
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

  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;
  // ...
};
```



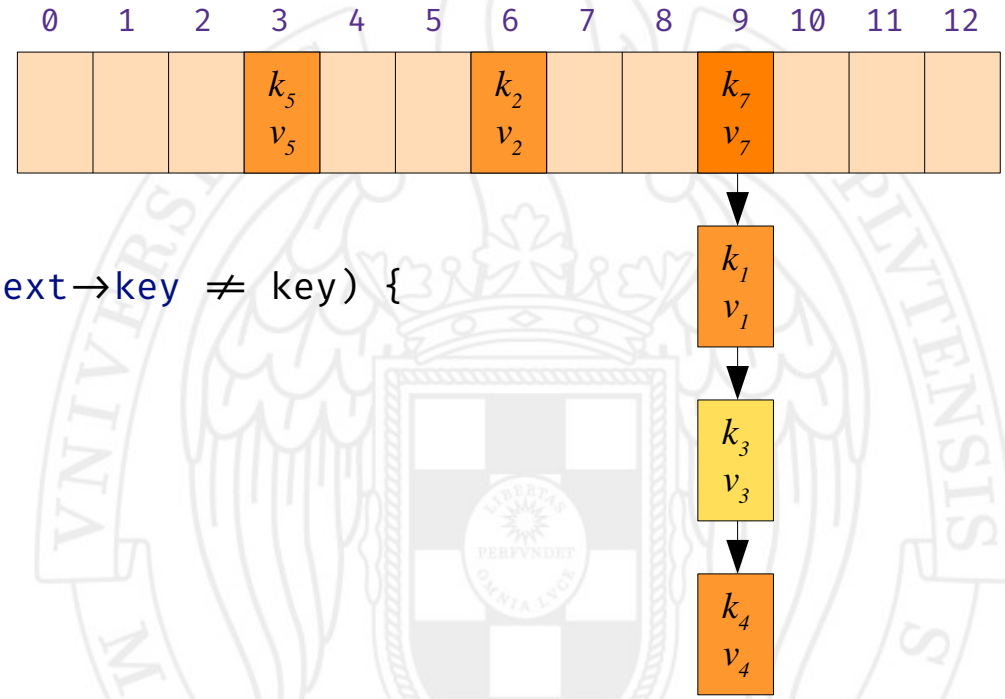
Clase `MapHash`: borrado

- 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.

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--;
      }
    }
  }
}
```



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--;
      }
    }
  }
}
```