ESTRUCTURAS DE DATOS

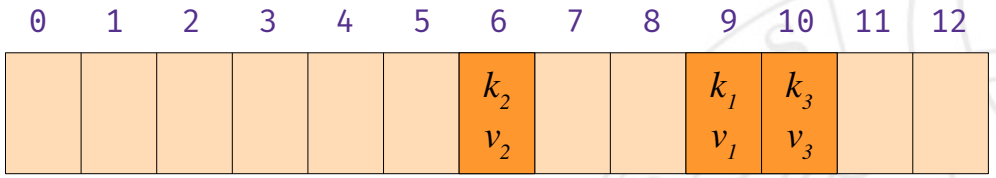
DICCIONARIOS

Tablas *hash* cerradas

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

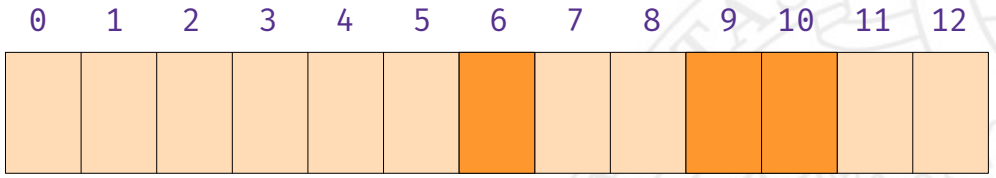
Objetivo

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



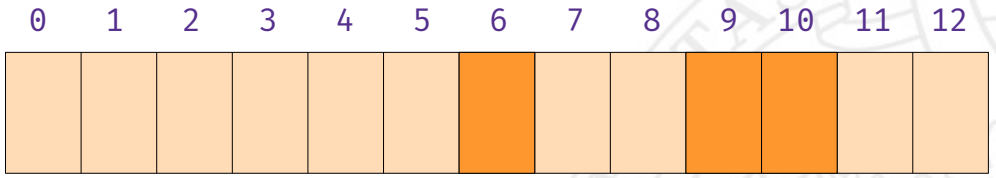
Idea de implementación

- Existen posiciones **libres** y **ocupadas**.



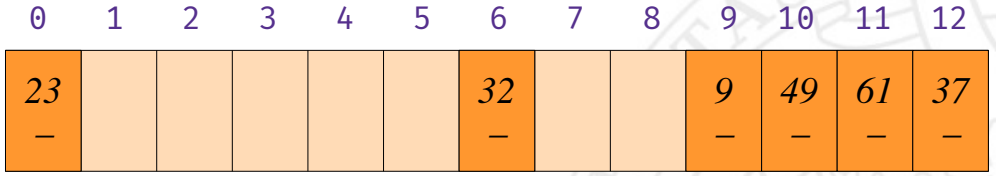
Inserción

- Si la entrada está ocupada, insertamos en la siguiente. Si también está ocupada, miramos en la siguiente, etc. hasta encontrar una posición libre.



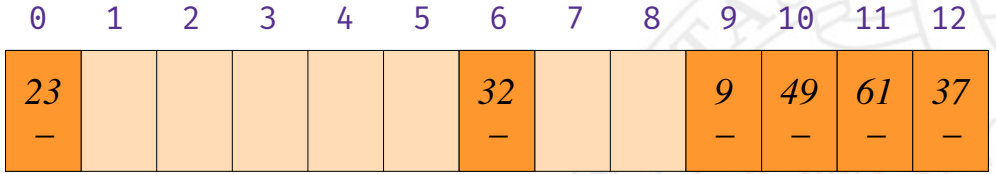
Ejemplo

- Insertamos claves 32, 49, 9, 61, 37, 23 (en este orden).



Búsqueda

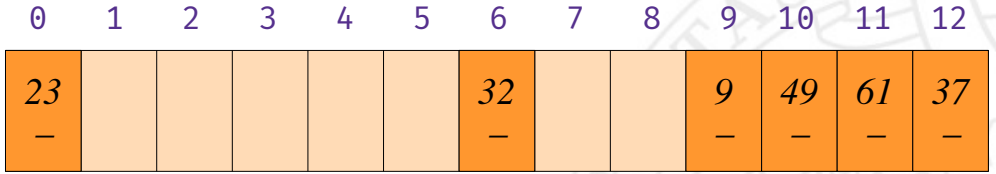
- Buscamos en el cajón $h(k)$ **mod** $CAPACITY$.
- A partir de ahí, buscamos en entradas sucesivas hasta encontrar la clave o llegar a una posición vacía.



- Ejemplo: buscamos 23 y 63.

¡Cuidado con el borrado!

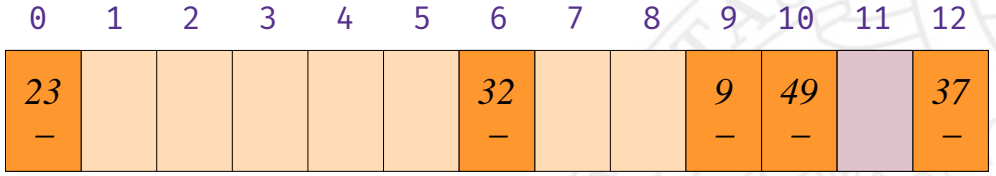
- Si eliminamos la entrada sin más, podemos imposibilitar la búsqueda de claves que vienen después.
- Ejemplo: borramos 61.



- ¿Y si ahora buscamos 23?

Solución

- Distinguir entre entradas ***libres*** y entradas ***eliminadas***.
- La búsqueda se detiene cuando llegamos a una posición libre.
- ...pero podemos escribir en entradas eliminadas.



- ¿Y si ahora buscamos 23?
- ¿Y si ahora insertamos la clave 36?

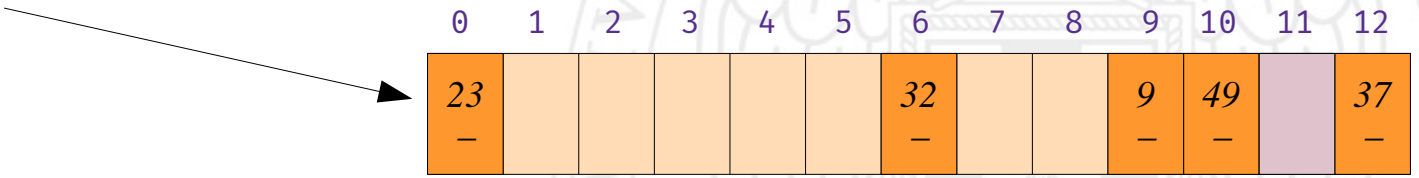
Clase MapHash: interfaz pública

```
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:
  enum class State { empty, occupied, deleted };

  struct Bucket {
    State state;
    MapEntry entry;

    Bucket(): state(State::empty) { }
  };

  Bucket *buckets;
  Hash hash;
  int num_elems;

};
```



Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key)
```

pos_to_insert

pos_found

- Busca una clave `key` recibida como parámetro en el vector `buckets`.
- Componentes del objeto `pair` de salida:
  - `pos_found`

    Posición del vector en la que se ha encontrado la clave. Si no se encuentra, es `-1`.
  - `pos_to_insert`

    Posición del vector en el que se debería insertar la clave, en caso de ser necesario (o `-1` si el vector está lleno)

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (pos_found == -1 && buckets[h].state != State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
  }

  if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (pos_found == -1 && buckets[h].state ≠ State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
  }

  if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

Repetimos mientras no hayamos encontrado la clave, o hayamos llegado a una posición vacía

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (pos_found == -1 && buckets[h].state ≠ State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
  }

  if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

Cambiamos **pos_insert** a la primera posición eliminada que encontremos

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (pos_found == -1 && buckets[h].state ≠ State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
  }

  if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

Si encontramos la clave `key` en la posición actual, establecemos `pos_found`

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (pos_found == -1 && buckets[h].state ≠ State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
  }

  if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

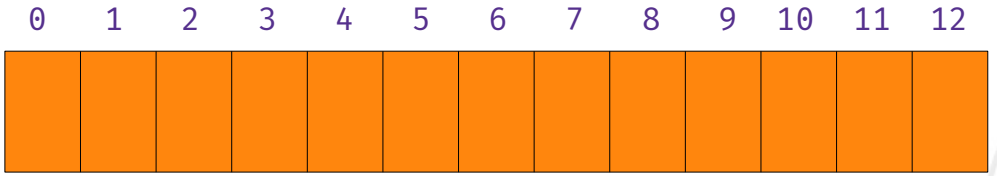
Pasamos a la posición siguiente. Si llegamos al final del vector, volvemos a la posición **0**.

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (pos_found == -1 && buckets[h].state ≠ State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
  }

  if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

Si al final hemos llegado a una posición libre, y no hemos pasado por ninguna borrada, establecemos `pos_to_insert`

¿Y si el vector está lleno?



- Se van buscando posiciones de manera circular.
- ¡La función no termina!
- Debemos acotar el número de iteraciones.

Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
  int h = hash(key) % CAPACITY;
  int cont = 0;

  int pos_to_insert = -1;
  int pos_found = -1;

  while (cont < CAPACITY && pos_found == -1 && buckets[h].state != State::empty) {
    if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
      pos_to_insert = h;
    }

    if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
      pos_found = h;
    }

    h = (h + 1) % CAPACITY;
    cont++;
  }

  if (cont < CAPACITY && pos_to_insert == -1) {
    pos_to_insert = h;
  }
  return {pos_to_insert, pos_found};
}
```

Método insert

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

  void insert(const MapEntry &entry) {
    auto [pos_to_insert, pos_found] = search_pos(entry.key);
    if (pos_found == -1) {
      assert (pos_to_insert ≠ -1);
      buckets[pos_to_insert].state = State::occupied;
      buckets[pos_to_insert].entry = entry;
      num_elems++;
    }
  }


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

Método at

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

  const V & at(const K &key) const {
    auto [pos_to_insert, pos_found] = search_pos(key);
    assert (pos_found != -1);
    return buckets[pos_found].entry.value;
  }

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

Método erase

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

  void erase(const K &key) {
    auto [pos_to_insert, pos_found] = search_pos(key);
    if (pos_found != -1) {
      buckets[pos_found].state = State::deleted;
      num_elems--;
    }
  }

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

Búsqueda de posiciones alternativas

Recordatorio

- Las tablas hash cerradas se basan en calcular una **posición inicial** p_0 en el vector y buscar una clave allí.

$$p_0 = h(k) \textbf{ mod } CAPACITY$$

- Si la clave no se encuentra, se busca en **posiciones alternativas** p_1, p_2, *etc.*

  En nuestro caso:

$$p_1 = (h(k) + 1) \textbf{ mod } CAPACITY$$

$$p_2 = (h(k) + 2) \textbf{ mod } CAPACITY$$

$$p_3 = (h(k) + 3) \textbf{ mod } CAPACITY$$

etc.

Recordatorio

- Las tablas hash cerradas se basan en calcular una **posición inicial** p_0 en el vector y buscar una clave allí.

$$p_0 = h(k) \textbf{ mod } CAPACITY$$

- Si la clave no se encuentra, se busca en **posiciones alternativas** p_1, p_2, *etc.*

  En general:

$$p_i = (h(k) + i) \textbf{ mod } CAPACITY$$

Sondeo lineal

- Existen otras posibilidades para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \textbf{ mod } CAPACITY$$

- Por ejemplo, si $c = 1$, $h(k) = 10$, $CAPACITY = 13$.

```
10  11  12  0  1  2  3 ...
```

Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \textbf{ mod } CAPACITY$$

- Por ejemplo, si $c = 2$, $h(k) = 10$, $CAPACITY = 13$.

```
10   12   1   3   5   7   9   11  ...
```

Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \textbf{ mod } CAPACITY$$

- Por ejemplo, si $c = 5$, $h(k) = 10$, $CAPACITY = 13$.

```
10   2   7   12   4   9  ...
```

Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod CAPACITY$$

- En general, si $mcd(c, CAPACITY) = 1$, el sondeo lineal garantiza el recorrido de todas las posiciones del array, en caso de ser necesario.
- En general, esto ocurre cuando $CAPACITY$ es primo.

Sondeo cuadrático

- Utiliza la siguiente fórmula:

$$p_i = (h(k) + i^2) \textbf{ mod } CAPACITY$$

- Por ejemplo, si $h(k) = 10$, $CAPACITY = 13$.

```
10  11  1  6  0  9  7  7 ...
```

Sondeo cuadrático

- Utiliza la siguiente fórmula:

$$p_i = (h(k) + i^2) \bmod CAPACITY$$

- Aquí no se garantiza el recorrido de todas las posiciones del vector, pero sí al menos la mitad de ellas.

Doble redispersión

- Utiliza una segunda función *hash* h' para la búsqueda de posiciones alternativas.

$$p_i = (h(k) + i\,h'(k)) \bmod CAPACITY$$

- Garantiza el recorrido de todas las posiciones si $CAPACITY$ es primo y $h'(k)$ nunca devuelve `0`.

Más información

- R. Peña
  Diseño de Programas. Formalismo y Abstracción (3ª edición)
  Pearson Educación (2005)
  Sección 8.1.3

- https://en.wikipedia.org/wiki/Linear_probing
- https://en.wikipedia.org/wiki/Quadratic_probing
- https://en.wikipedia.org/wiki/Double_hashing