Encadenado Separado (I)

```
private static final class Node<K> {
   K key;
   Node<K> next;

   Node(K key, Node<K> next) {
     this.key = key;
     this.next = next;
   }
 }

 private static final int DEFAULT_NUM_CHAINS = HashPrimes.primeGreaterThan(32);
 private static final double DEFAULT_MAX_LOAD_FACTOR = 5;

 private Node<K>[] table;
 private int size; // number of keys inserted in table
 private final double maxLoadFactor;
```

Encadenado Separado (III)

```
public SeparateChainingHashTable(int numCells, double maxLoadFactor) {
  if (numChains <= 0) {
    throw new IllegalArgumentException("initial number of chains must be greater than 0");
  }
  this.table = (Node<K>[]) new Node[numCells]; //Creamos el array con el tamaño dado
  this.size = 0; //Nada insertado aún.
  this.maxLoadFactor = maxLoadFactor; //Máximo factor de ocupación permitido.
}
```

Encadenado Separado (IV): clase de apoyo

```
     * Searches for a key in table and returns information about key location:
     * current will be a reference to node containing key or null if key is not in table,
     * prev will be a reference to previous node in chain or null if key is first in chain,
     * index will be the index of chain where key is or should be.
    private final class Finder {
        int index;
        Node<K> previous, current;

        Finder(K key) {
            index = hash(key);
            previous = null;
            current = table[index];

            while ((current != null) && (!current.key.equals(key))) {
                previous = current;
                current = current.next;
            }
        }
    }
```

Implementación de un iterador

```
private final class SeparateChainingHashTableIterator implements Iterator<K> {
    int index;        // índice de la cabeza de la lista enlazada de colisiones actual.
    Node<K> current;  // nodo actual que se está recorriendo.

    public SeparateChainingHashTableIterator() {
        index = 0;
        current = table[index];
        advance();              //Localizamos la primera cabeza que no es null.
    }
    private void advance() {
        while ((current == null) && (index < table.length - 1)) {
            index++;
            current = table[index];
        }
    }
    public boolean hasNext() {
        return (current != null);
    }
    public K next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        K key = current.key;
        current = current.next;  //Avanzamos dentro de la lista.
        advance();                //Llamamos para saltar a la siguiente lista si está se ha terminado.
        return key;
    }
}
```

Complejidad computacional de las operaciones de `Encadenado Separado`

Operation	Cost
`size`	O(1)
`insert`	O(1)
`search`	O(1)
`contains`	O(1)
`delete`	O(1)
`deleteOrUpdateOrInsert`	O(1)
`clear`	O(1)

Encadenado Separado vs Árboles Binarios de Búsqueda

- Test Experimental
  - Hemos medido el tiempo de ejecución para 1 millón de operaciones aleatorias (inserción, búsqueda y eliminación) en una tabla inicialmente vacía
  - Usando una CPU Intel i7 860
    - La tabla hash (SeparateChainingHashTable) fue 3.6 veces más rápida que un árbol AVL
    - y 3.3 veces más rápida que un árbol BST
- Principales desventajas de la tabla hash vs AVL y BST:
  - Si muchas claves tienen el mismo valor hash ... no podemos garantizar una complejidad logarítmica de las operaciones y éstas pueden llegar a ser lineales en el peor caso.
  - Las búsquedas vía una relación de orden (menor, mayor, predecesor, etc.) no puede implementarse eficientemente.
  - Cuando de produce el rehashing el tiempo de esa operación es bastante elevado (no apta para tiempo real).

Resolución de Colisiones: segunda parte.

- En la práctica, coincidirán los valores hash de algunas claves.
- Existen distintas formas de resolver colisiones:
  - ~~Encadenado (Separate Chaining): una lista enlazada contiene los elementos que colisionan en la misma celda.~~
  - **Direccionamiento abierto (Open addressing)**: cada elemento que colisiona es reubicado en otra celda del array
    - Prueba lineal: asignar la siguiente celda libre

Direccionamiento Abierto. Prueba Lineal

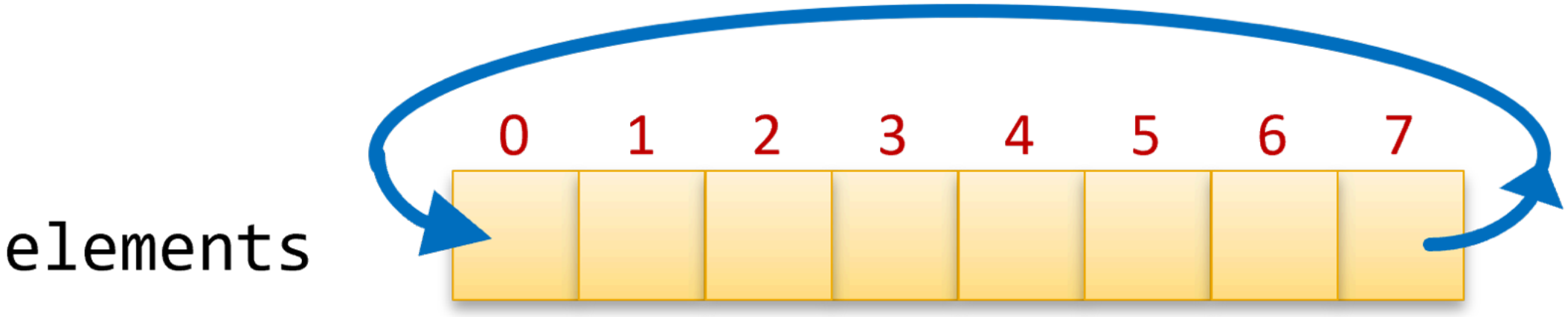
- Direccionamiento abierto:
  - No usa listas enlazadas para resolver las colisiones.
  - Resuelve cada colisión colocando el elemento en otra celda libre (el factor de carga debe ser ≤ 1).
  - La búsqueda y la eliminación de claves se complican.
- Prueba lineal:
  - Trata el array como una estructura circular y se colocan los elementos que colisionan en la siguiente celda libre.
  - El rendimiento se degrada en exceso cuando el factor de carga es > 0.5. **Solución**: aumentar el tamaño de la tabla y realizar una reubicación (rehashing)

Direccionamiento Abierto: constructor.

```
public LinearProbingHashTable(int numCells, double maxLoadFactor) {
  if (numCells <= 0) {
    throw new IllegalArgumentException("initial number of cells must be greater than 0");
  }
  keys = (K[]) new Object[numCells];
  size = 0;
  this.maxLoadFactor = maxLoadFactor;
}
```

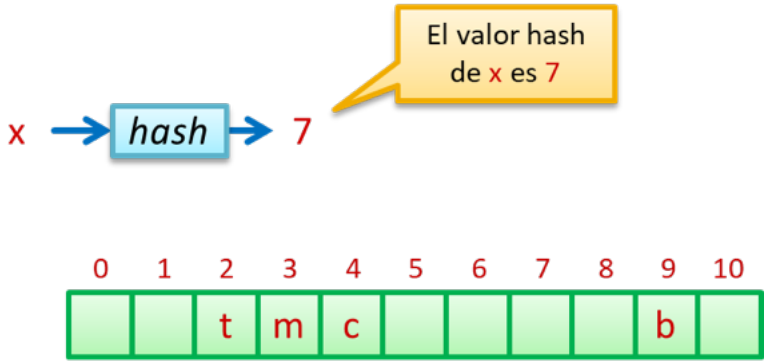
Prueba Lineal. Inserción (I)

- Inserción:
  - Sea id_x el valor hash de la clave a insertar.
  - Si la celda cell[id_x] está libre, insertarla aquí.
  - En otro caso, colocarla en la siguiente libre
    - siguiente debe interpretarse en forma circular



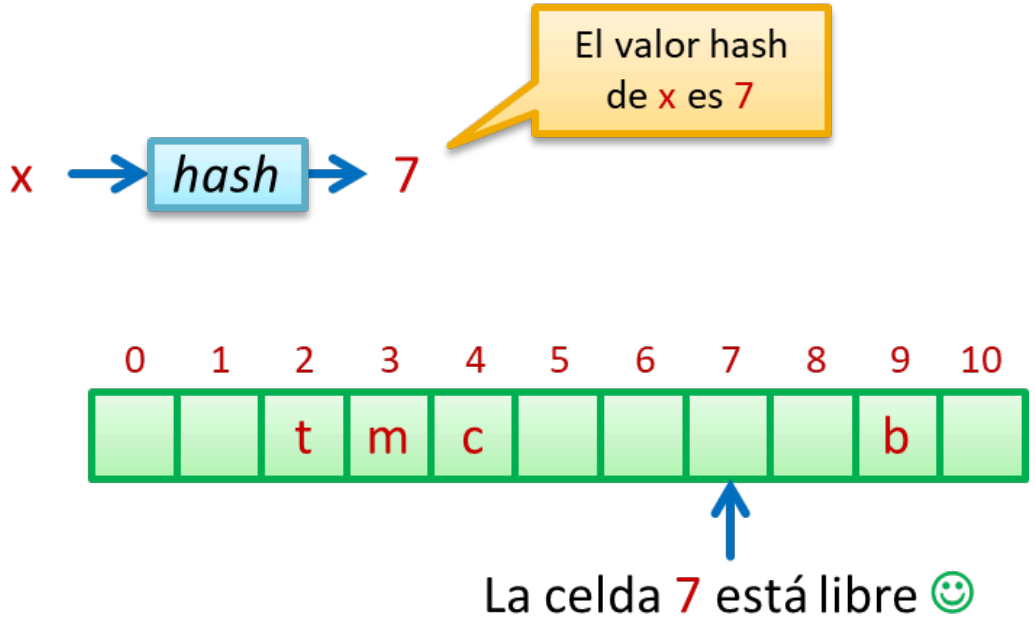
Prueba Lineal. Inserción (II)

- insert x



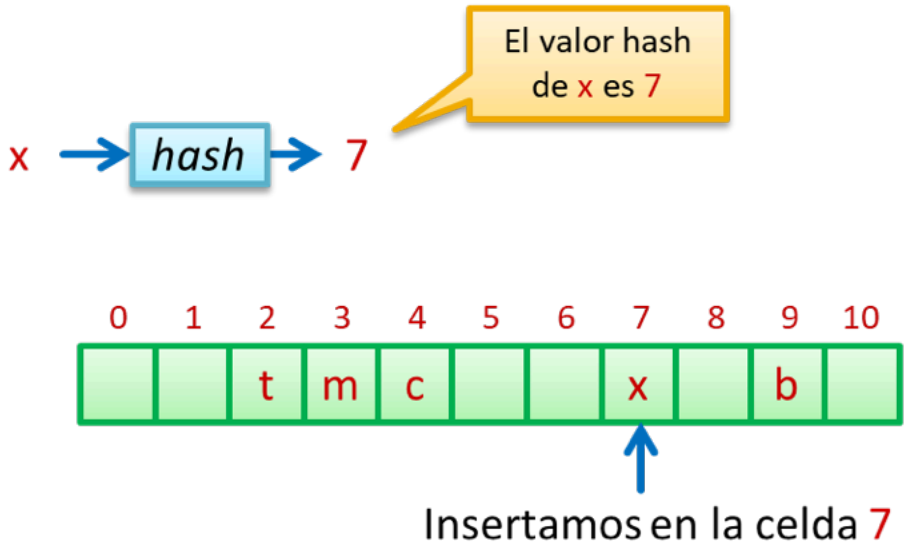
Prueba Lineal. Inserción (III)

- insert x



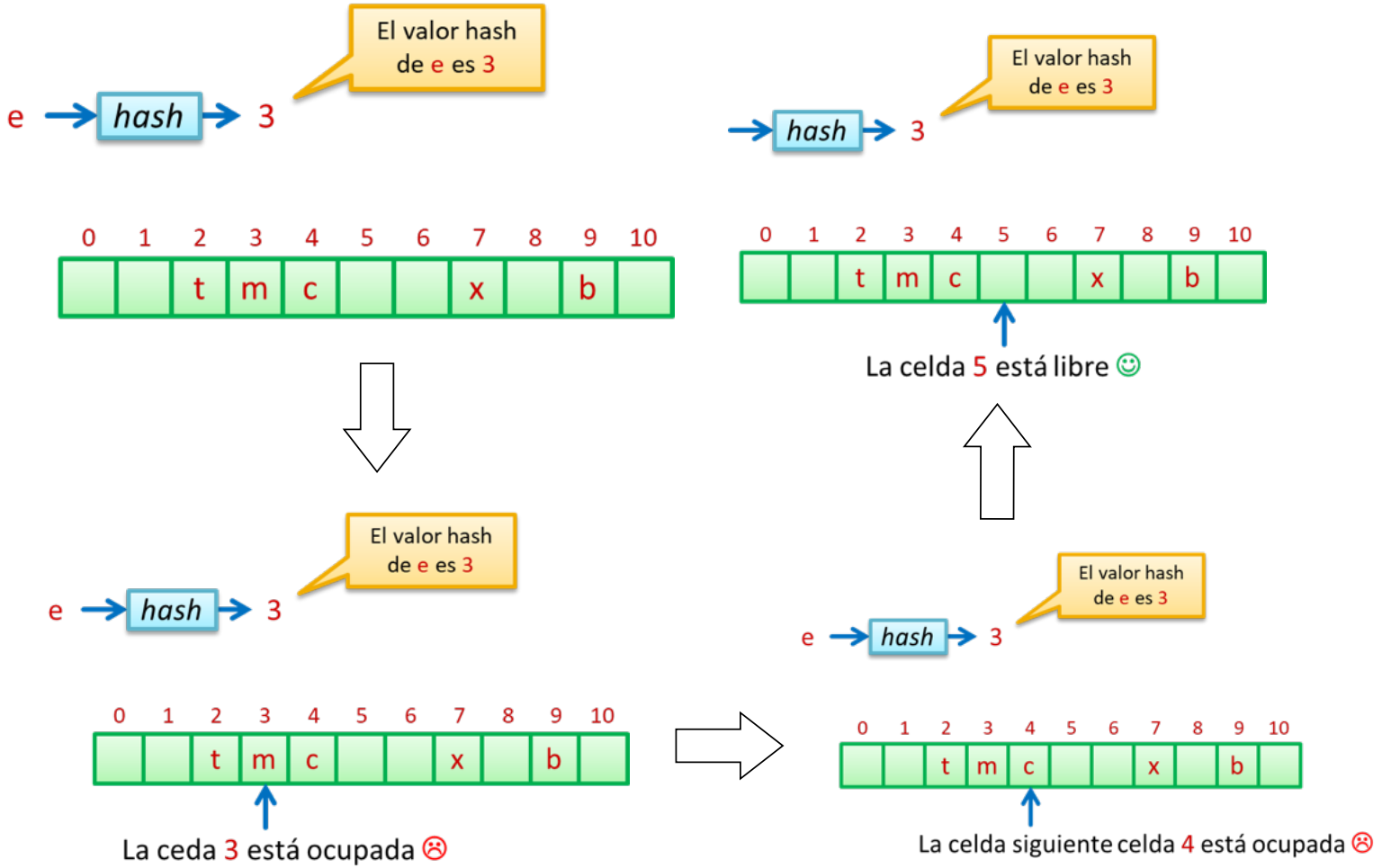
Prueba Lineal. Inserción (IV)

- insert x



Prueba Lineal. Inserción (V)

- insert e

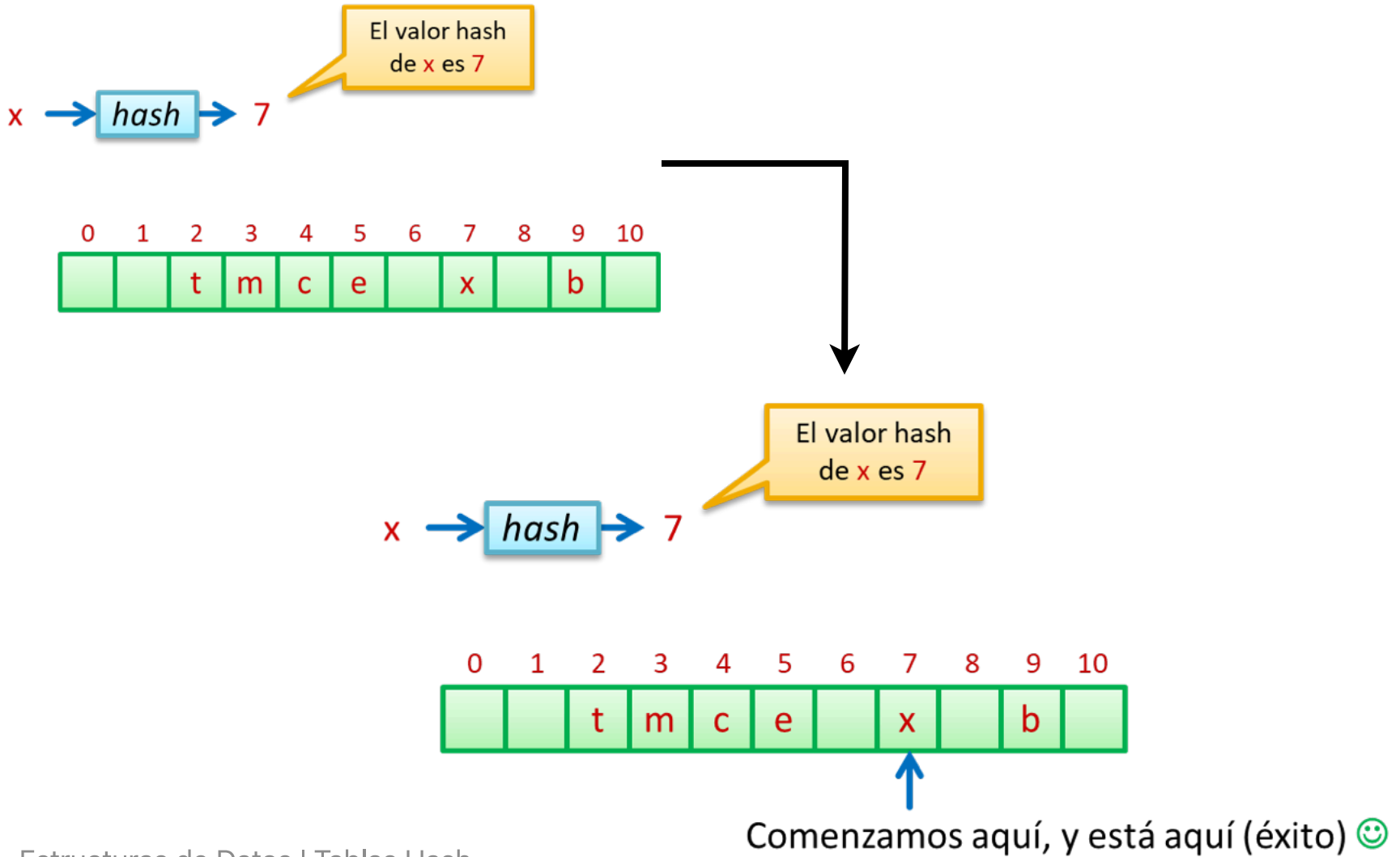


Prueba Lineal. Búsqueda

- Comenzando en la celda correspondiente al valor hash, inspeccionamos secuencialmente hasta que:
  - o bien encontramos la clave (éxito),
  - o bien encontramos en su lugar una celda libre (fracaso)

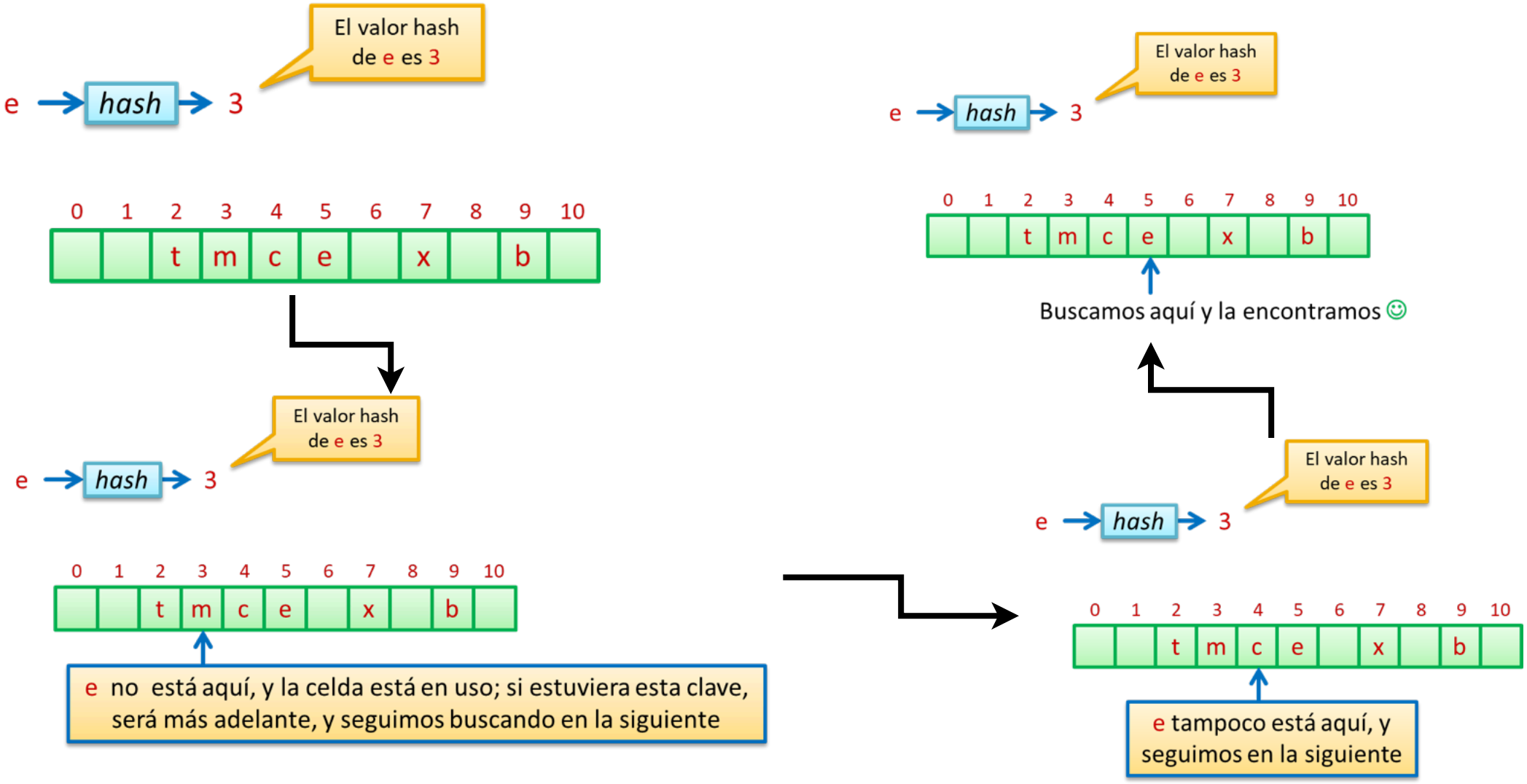
Prueba Lineal. Búsqueda con éxito

- Search x

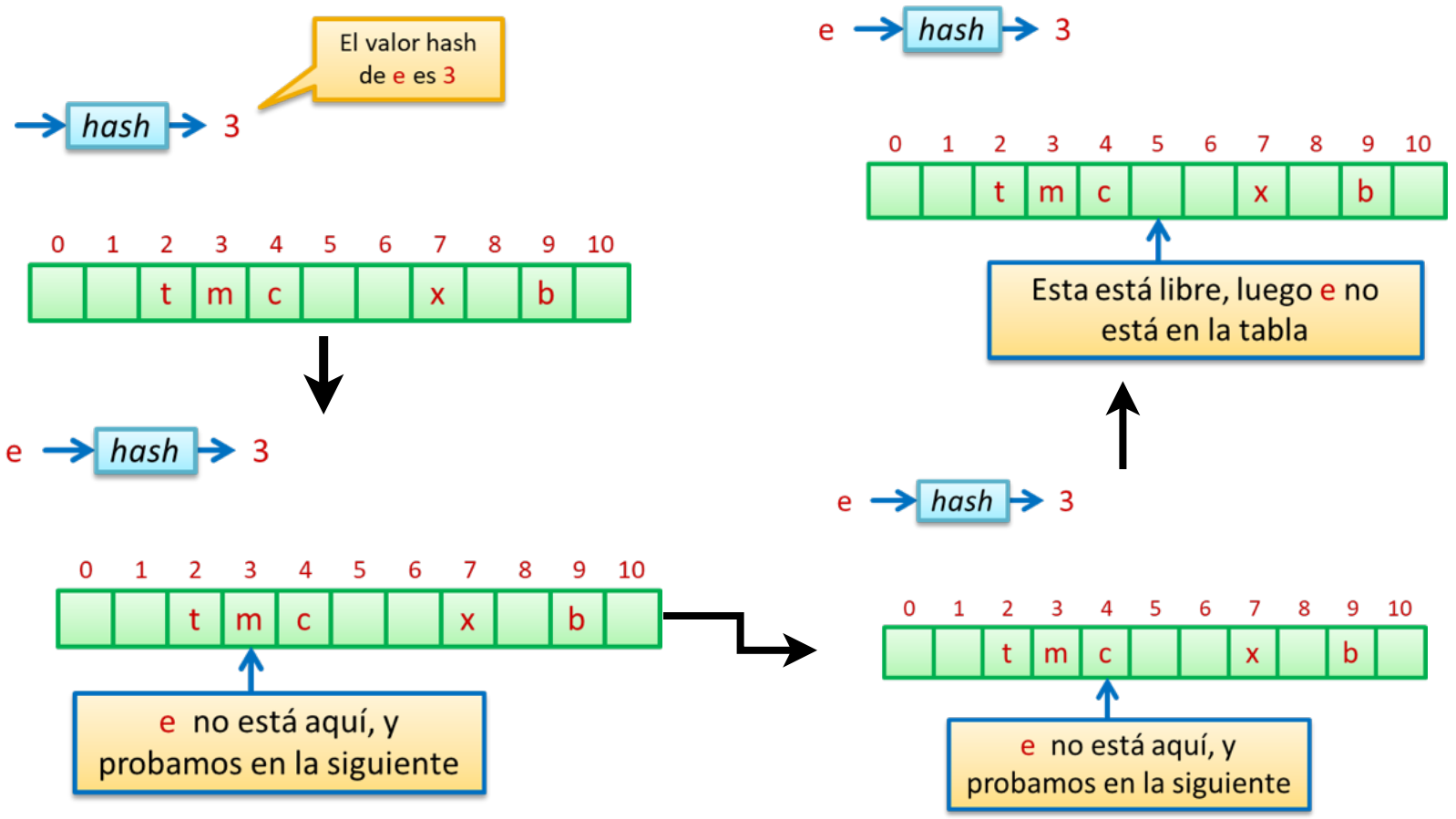


Prueba Lineal. Búsqueda con éxito

- Search e



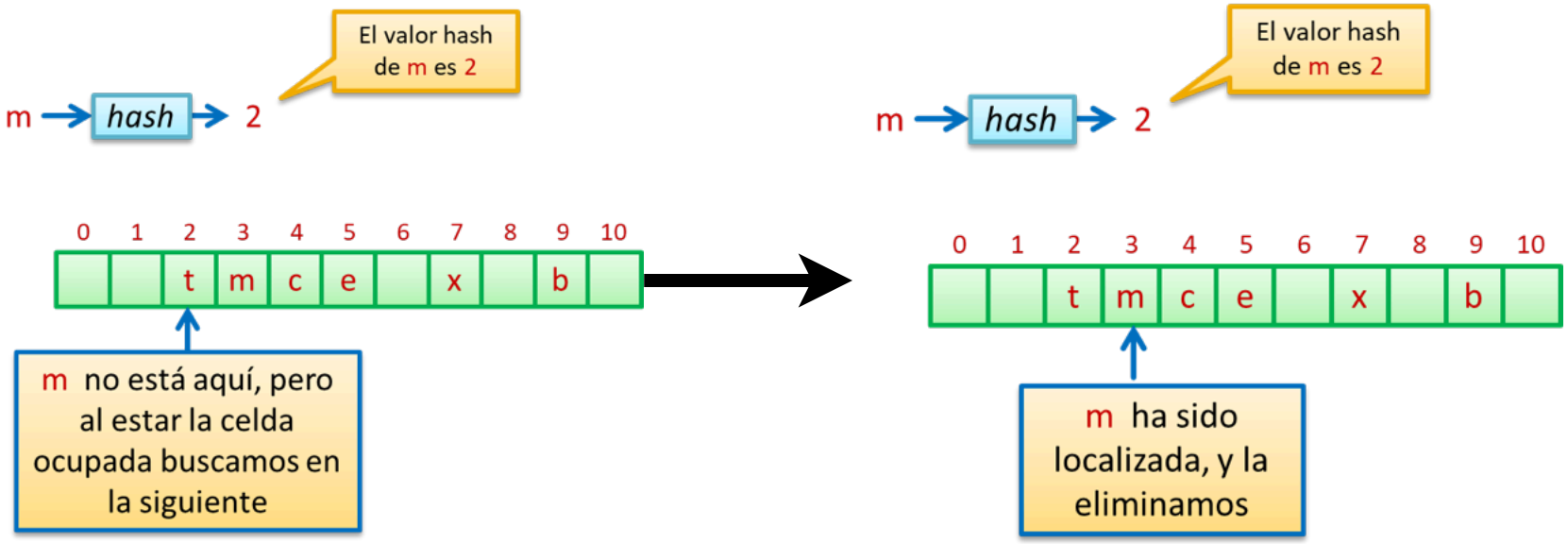
Prueba Lineal. Búsqueda sin éxito



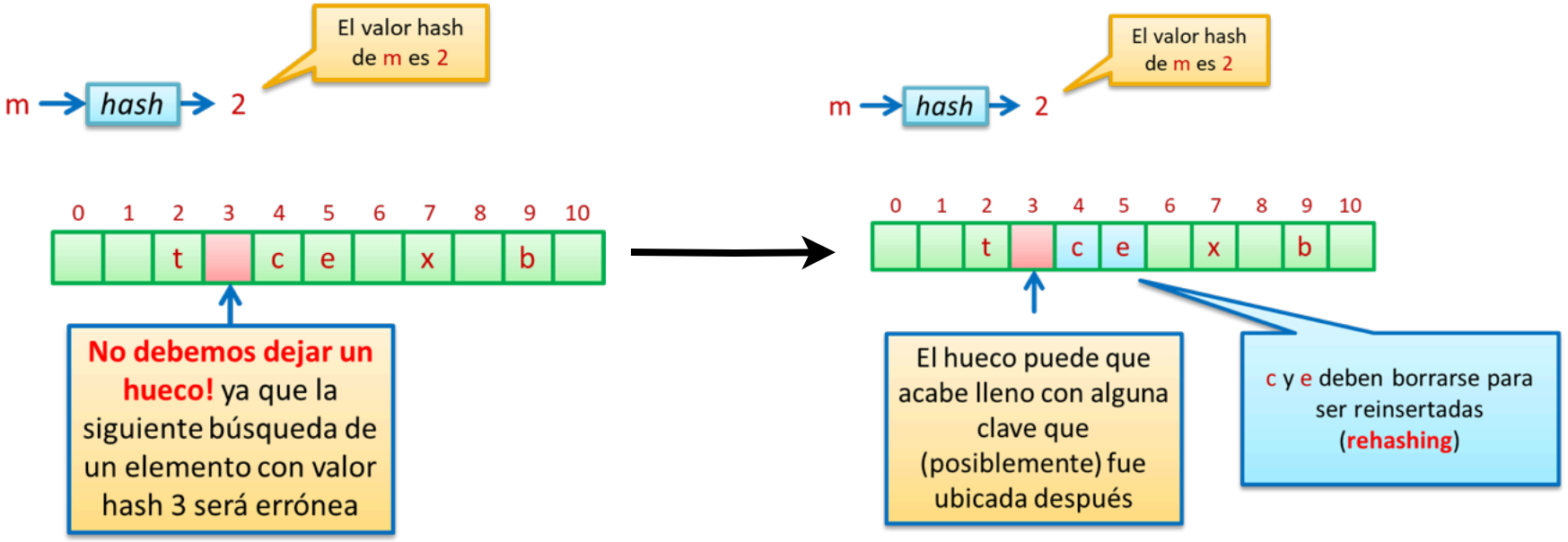
Eliminación

- Buscamos el elemento a eliminar.
- Si está, lo eliminamos liberando la celda.
- Reinsertamos los elementos de la celdas siguientes a la clave eliminada hasta encontrar un hueco (entre claves que colisionan no puede haber huecos).

Eliminación



Eliminación

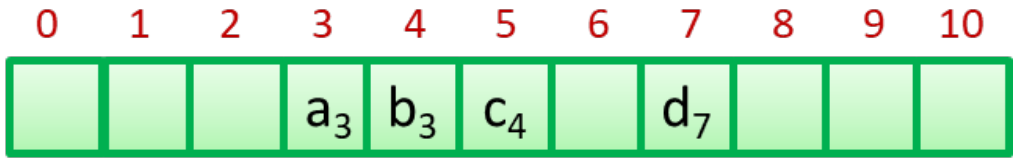


Eliminación

- Será necesario hacer un rehashig de los valores que están insertados a continuación hasta el siguiente hueco
  - Es posible que se hayan insertado en esos lugares debido a colisiones
  - Sin embargo, al eliminar la **m** se ha creado un hueco

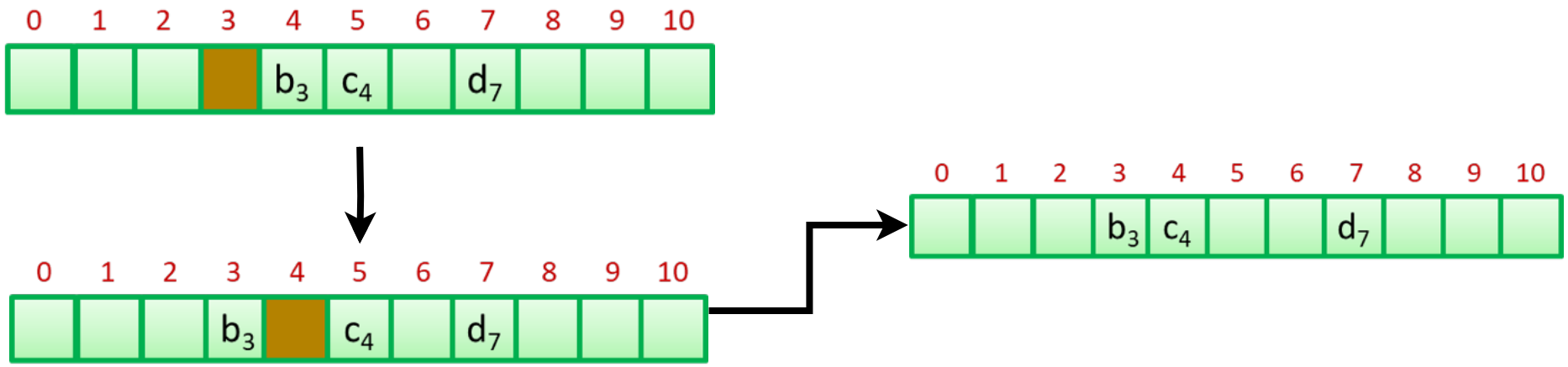
Rehashing. Ejemplo

- Insertamos sucesivamente las claves a_3 , b_3 , c_4 , d_7 (el subíndice indica el valor hash)



- Si eliminamos la **a** no sabemos si los elementos a continuación se situaron ahí debido a colisiones
- Hay que recalcular su posición (rehashing) de todos los elementos que hay a continuación.

Rehashing. Ejemplo



Rehashing. Método Java

```
private void rehashing() {
    // compute new table size
    int newCapacity = HashPrimes.primeDoubleThan(keys.length);

    K[] oldKeys = keys;

    keys = (K[]) new Object[newCapacity];

    // reinsert elements in new table
    for (K oldKey : oldKeys) {
      if (oldKey != null) {
        int newIndex = searchIndex(oldKey);
        keys[newIndex] = oldKey;
      }
    }
  }
```

Complejidad computacional de las operaciones de `Direccionamiento abierto`

Operation	Cost
`size`	O(1)
`insert`	O(n)
`search`	O(1)
`contains`	O(1)
`delete`	O(n)
`deleteOrUpdateOrInsert`	O(n)
`clear`	O(n)

x