Búsqueda de *subtrings* en *strings*

Queremos buscar una subsecuencia X en una secuencia Y

¿Qué tan costoso es esto según el tamaño de X e Y?

¿Cómo podemos hacer para descartar subsecuencias de Y?

Ejemplo

X = AGCGATGCTATCTTGGGGCTATT

Y =

ACGTGACTGCTCCGCGCGTGAATTTCGATCGCGCGGATCTAGCTAGCTAG
CTGCTAGCTAGCTTCGCTATCGTAGTCGTCAGTATGATGTATAGAATAATTA
ATAAAAGCGCCTGCCTAGTCGTGTGTCACGTAGTCATCGAGCGGGCTCAT
ACGCAGATCAGCGATGCTATCTTGGGGCTATTATGCTAGCTATCGCCTAGC
GCGATATACGCGCGCGGATTCGCTATATGCT

Funciones de *hash*

Una función de **hash** para objetos de un dominio D es:

$$h : D \rightarrow \mathbb{N}^0$$

Decimos que la función de hash produce una **colisión** cuando

$$h(A) = h(B) \ \wedge \ A \neq B$$

¿Cómo podemos usar una función de hash para saber si $X \in Y$?

¿Qué podríamos usar como función de hash para un *string*?

¿Qué tan rápido podemos resolver el problema ahora?

Hashing incremental

Si Z es una modificación de X, y conocemos $h(X)$

La función h se dice **incremental** si permite calcular $h(Z)$ a partir de $h(X)$ y la modificación que generó Z

El costo de calcularlo debe ser lineal en el número de cambios

¿Y si usamos una función de hash incremental?

¿Cuál sería la complejidad entonces?

¿Es posible resolver el problema en menos tiempo?

Podemos “sumar” las letras

Si vemos cada letra como un número, h puede ser la suma de cada letra:

$$h(X[i:j]) = x_i + x_{i+1} + \cdots + x_{j-1} + x_j$$

Teniendo $h(X[i:j])$, ¿cómo podemos calcular $h(X[i+1:j+1])$ en $O(1)$?

Podemos interpretar los strings como números en una cierta base

¿Qué pasa si vemos la secuencia como un número?

Eso significa considerar cada letra como un dígito

¿Podemos calcular el valor de hash de manera incremental?

¿Podemos calcular *h* incrementalmente?

Interpretamos la secuencia de largo m como un número en base b:

$$h(X[i:j]) = x_i b^m + x_{i+1} b^{m-1} + \cdots + x_{j-1} b^2 + x_j\ b$$

Teniendo $h(X[i:j])$, ¿cómo podemos calcular $h(X[i+1:j+1])$ en $O(1)$?

Muchas colisiones

A mayor número de colisiones, más lento es el algoritmo

En el peor caso hay que comparar todo: $O\big((n-m)\cdot m\big)$

¿Cómo podríamos garantizar 0 colisiones?

Hashing perfecto

Una función de hash es **perfecta** si no tiene colisiones

Es decir:

$$A = B \leftrightarrow h(A) = h(B)$$

Una función puede ser **perfecta** e **incremental** a la vez

Interpretación numérica

¿Qué valores deben tener los caracteres y la base b para que h sea perfecta?

El hashing perfecto no es muy práctico en la vida real, ¿por qué?

Diccionarios

Queremos un diccionario —insertar, eliminar, chequear pertenencia— en que **no nos interesa** el orden de los datos

Esto nos debería permitir complejidades menores que $O(\log n)$

¿Podremos guardar los datos en un arreglo? ¿En qué posición?

Tablas de hash

Una **tabla de hash** es una implementación de un diccionario que:

- No tiene noción de orden
- Sus operaciones son $O(1)$ **en promedio**

Recorrido de la función

Si la tabla de hash es de tamaño m,

… ¿qué pasa con los valores de h que se salen de la tabla?

Método de la división

Simplemente, usar el módulo:

$$h'(X) = h(X) \bmod m$$

Si $h(X)$ distribuye bien, entonces $h'(X)$ distribuye bien

Pero si m es potencia de 2, o de 10, se pierde información sobre *X*: no todo el valor de *X* —o, mejor dicho, de *h*(*X*)— es usado para calcular *h'*(*X*)

Método de la multiplicación

Sea A un número entre 0 y 1:

$$h'(X) = \lfloor m \cdot (A \cdot h(X) \bmod 1) \rfloor$$

Si $h(X)$ distribuye bien, entonces $h'(X)$ distribuye bien

Es más costosa, pero no depende del valor de m

Se recomienda $A = \frac{1}{\varphi}$

En resumen

Una **buena** función de hash cumple con lo siguiente:

- Se calcula a partir de toda la información de un objeto
- Es rápida de calcular
- Distribuye de manera uniforme
- Los valores de hash de dos objetos parecidos son distintos

Colisiones

Si $A \neq B$,

… entonces decimos que una función de hash produce una **colisión** si

$$h(A) = h(B)$$

Manejo de colisiones

Eso significa que A y B caen en la misma celda de la tabla

¿Cómo podemos guardar ambos datos en la tabla?

Nos interesa poder buscarlos en el futuro

Direccionamiento abierto

Podemos buscar otra celda que esté disponible

¿Dónde la buscamos?

- debemos seguir alguna regla

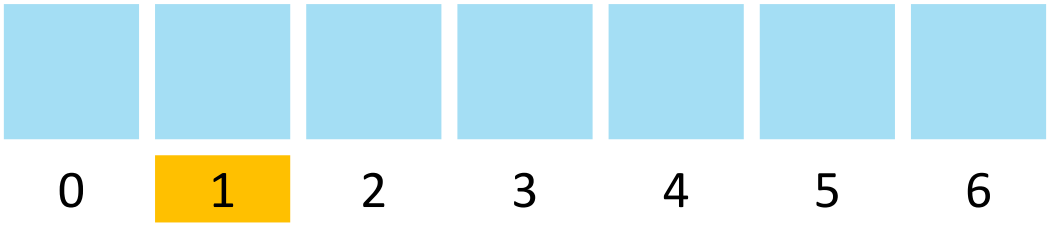
¿Qué complejidad tiene esto?

Direccionamiento abierto con **sondeo lineal**

$m = 7$

Insertemos la *A*

$h(\mathrm{A}) = 15;\ 15 \bmod 7 = 1$

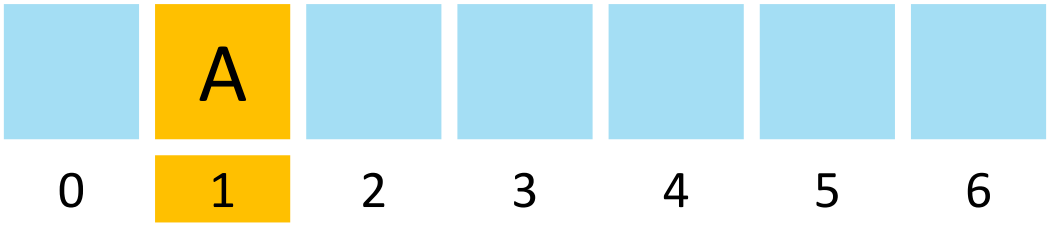


Sondeo lineal: ejemplo

$m = 7$

Insertemos la *A*

$h(\mathrm{A}) = 15;\ 15 \bmod 7 = 1$

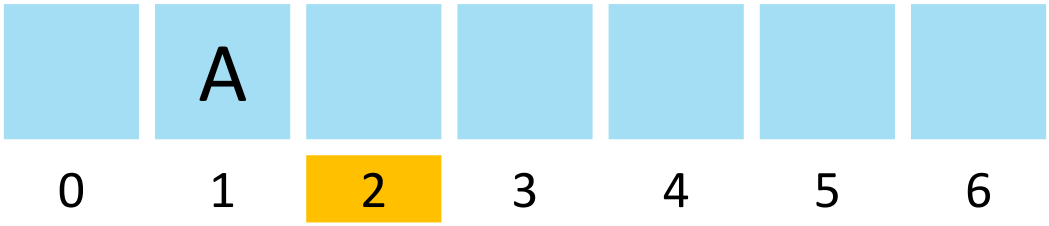


Sigue el ejemplo

$m = 7$

Insertemos la Q

$h(Q) = 37;\ 37 \bmod 7 = 2$



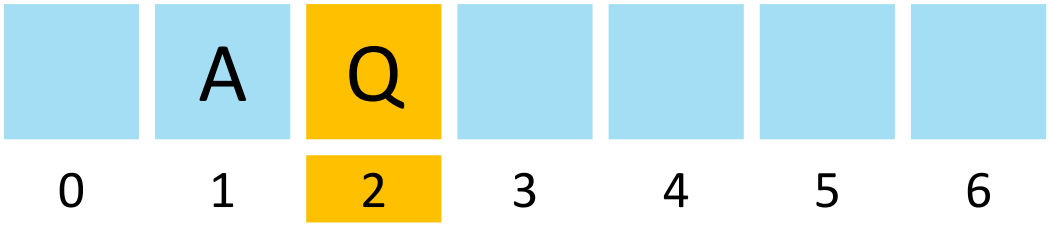
...

$m = 7$

Insertemos la Q

$$h(Q) = 37$$

$37 \bmod 7 = 2$

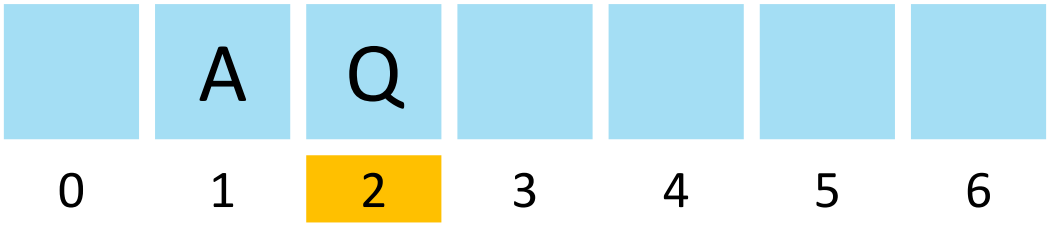


...

$m = 7$

Insertemos la *L*

$h(L) = 51;\ 51 \bmod 7\ = 2$

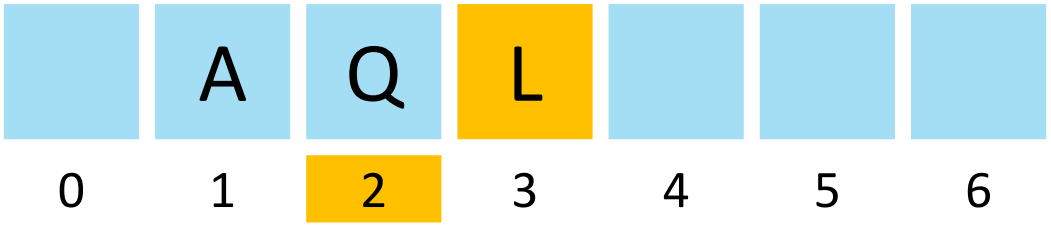


...

$m = 7$

Insertemos la *L*

$h(L) = 51;\ 51 \bmod 7\ = 2$

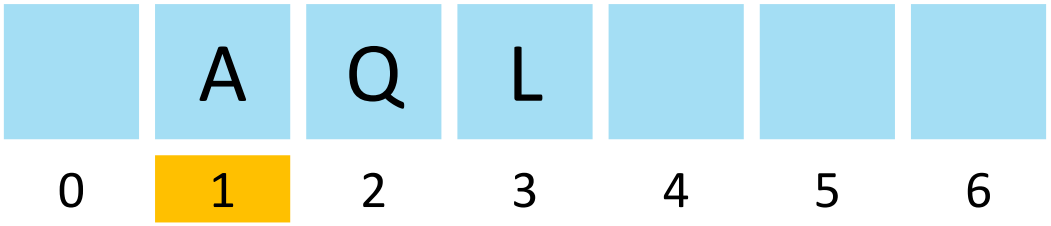


...

$m = 7$

Insertemos la *X*

$h(X) = 29;\ 29 \bmod 7\ = 1$

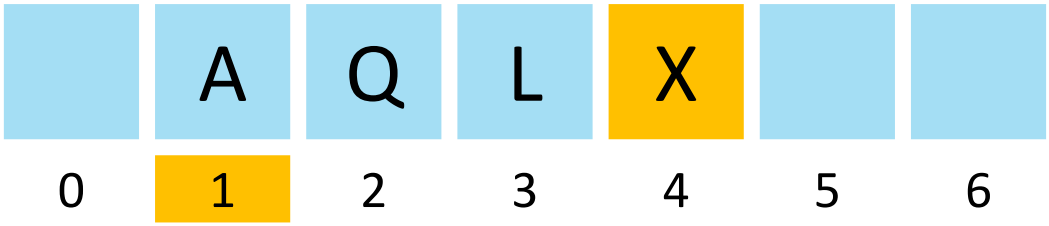


Fin del ejemplo

$m = 7$

Insertemos la *X*

$h(X) = 29;\ 29 \bmod 7\ = 1$



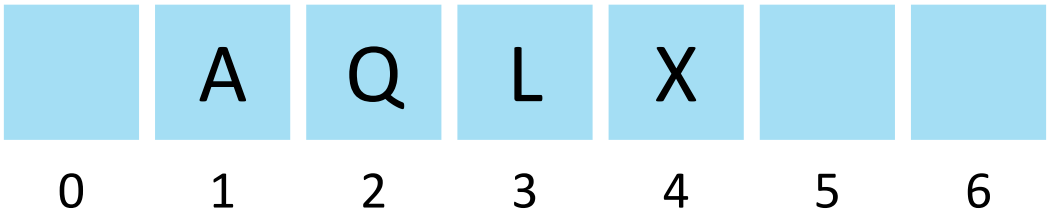
Otras posibilidades para direccionamiento abierto

Métodos populares de direccionamiento abierto son:

- sondeo lineal:
  - buscar en $H,\ H+1,\ H+2,\ H+3\ \ldots$
- sondeo cuadrático:
  - buscar en $H,\ H+1c_1+1^2c_2,\ H+2c_1+2^2c_2\ \ldots$
- *double hashing*:
  - buscar en $h_1(k), h_1(k)+h_2(k), h_1(k)+2h_2(k),\ \ldots$

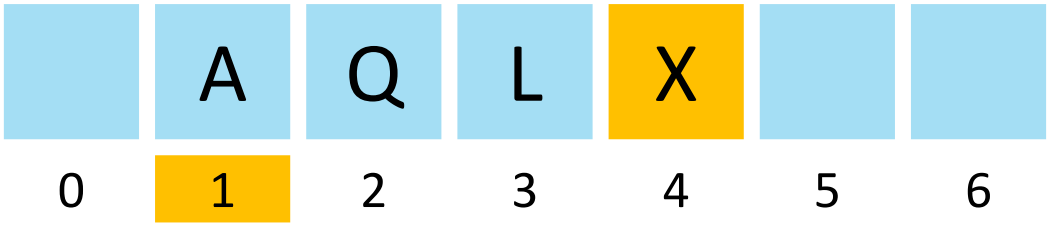
Búsqueda exitosa bajo sondeo lineal

$m = 7$



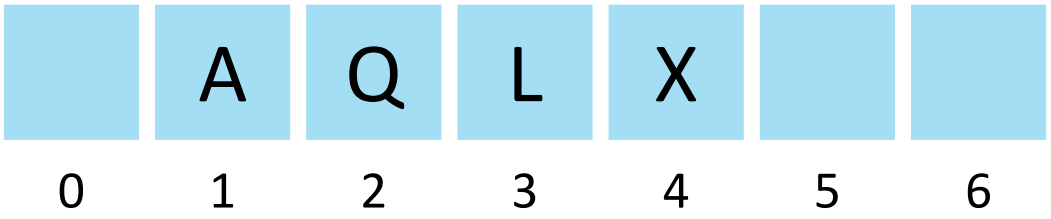
¿Cómo buscamos la *X* con sondeo lineal?

$h(X) = 29;\ 29 \bmod 7 = 1$



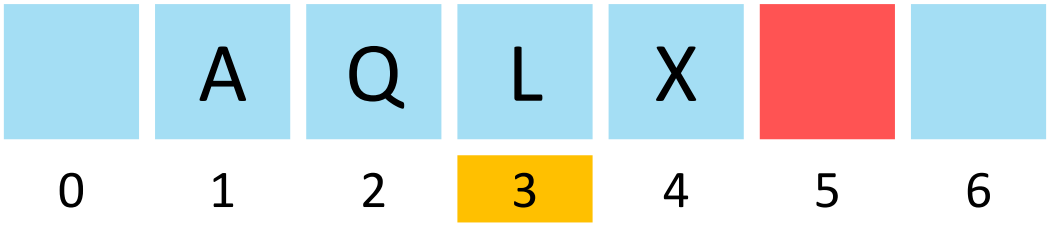
Búsqueda de un dato que no está, bajo sondeo lineal

$m = 7$



¿Cómo buscamos la *R* con sondeo lineal?

$h(R) = 10;\ 10 \bmod 7 \ = 3$



Problemas del direccionamiento abierto

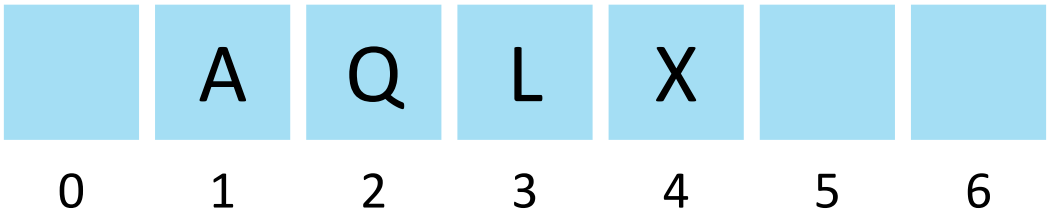
¿Qué problema tiene el sondeo lineal?

¿Qué problema tienen los otros esquemas?

¿Qué problema tiene el guardar los datos en otra celda?

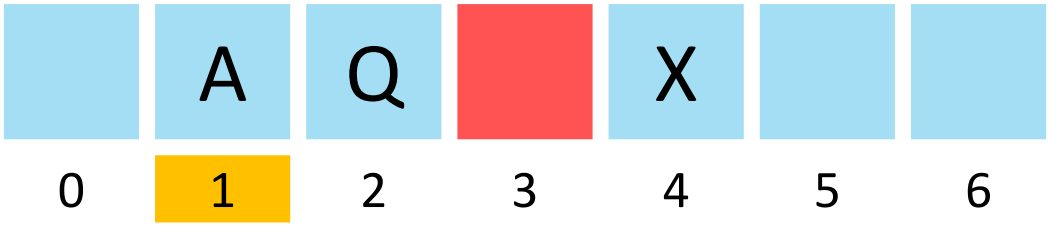
La eliminación es problemática

$m = 7$



Eliminemos la *L*. ¿Cómo buscamos la *X* con sondeo lineal?

$h(X) = 29;\ 29 \bmod 7\ = 1$



Otra posibilidad

Si A ya estaba en la tabla:

Podemos guardar A y B en la misma celda…

… pero ¡dentro de otra estructura de datos!

Encadenamiento (o listas ligadas)

Si tenemos una **lista** en cada celda de la tabla

Hemos guardado n datos, y la tabla es de tamaño m

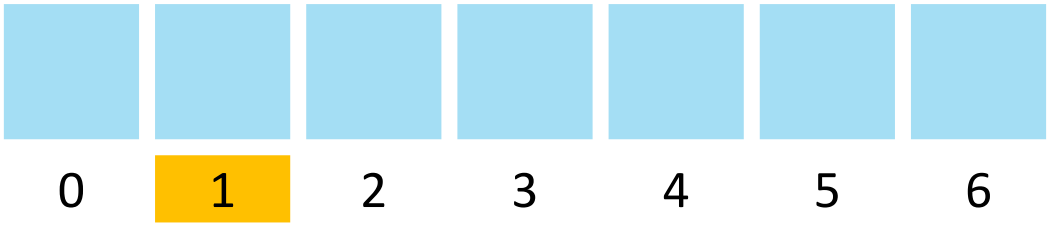
¿Cuál sería la complejidad de las operaciones de la tabla?

Encadenamiento: ejemplo

$m = 7$

Insertemos la *A*

$h(A) = 15;\ 15 \bmod 7 = 1$

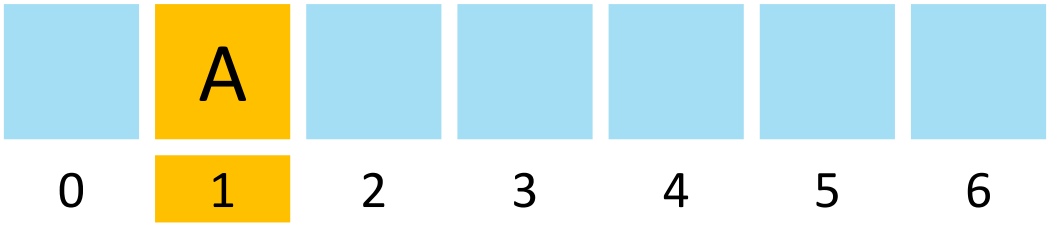


Sigue el ejemplo

$m = 7$

Insertemos la *A*

$h(A) = 15;\ 15 \bmod 7 = 1$

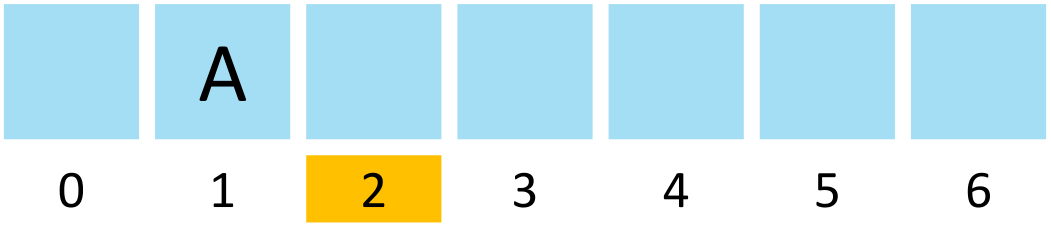


...

$m = 7$

Insertemos la Q

$h(\mathrm{Q}) = 37;\ 37 \bmod 7 \ = 2$

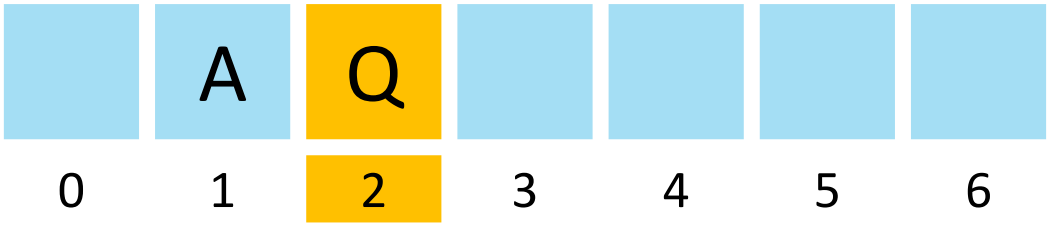


...

$m = 7$

Insertemos la Q

$h(Q) = 37;\ 37 \bmod 7\ = 2$

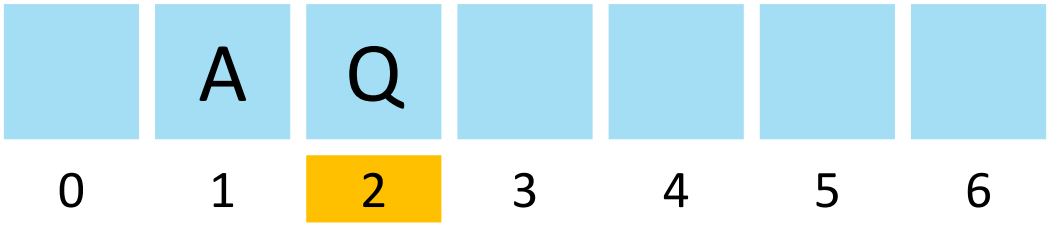


...

$m = 7$

Insertemos la *L*

$h(L) = 51;\ 51 \bmod 7\ = 2$

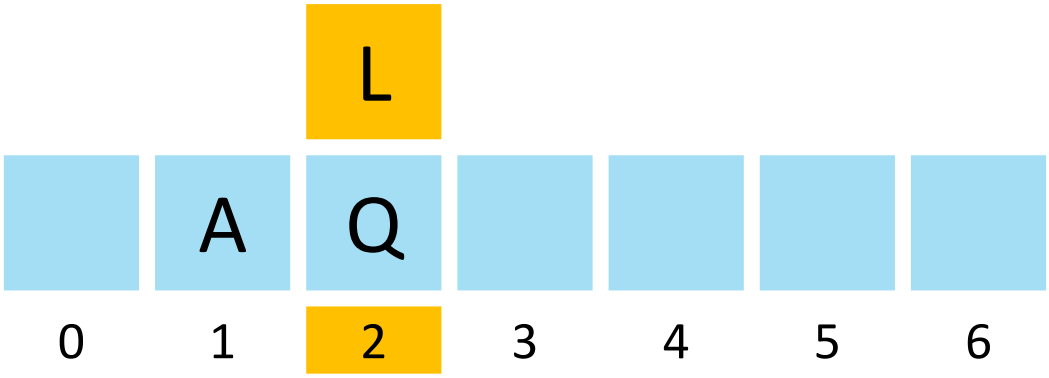


...

$m = 7$

Insertemos la *L*

$h(L) = 51;\ 51 \bmod 7\ = 2$

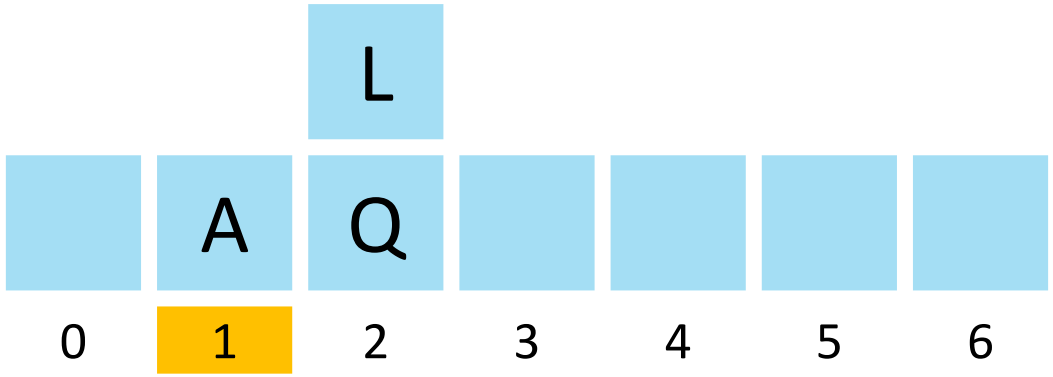


...

$m = 7$

Insertemos la *X*

$h(X) = 29;\ 29 \bmod 7\ = 1$

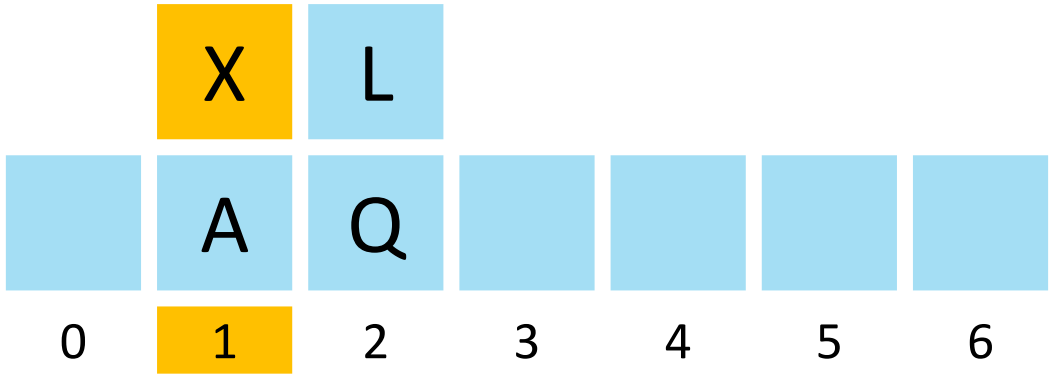


...

$m = 7$

Insertemos la *X*

$h(X) = 29;\ 29 \bmod 7\ = 1$

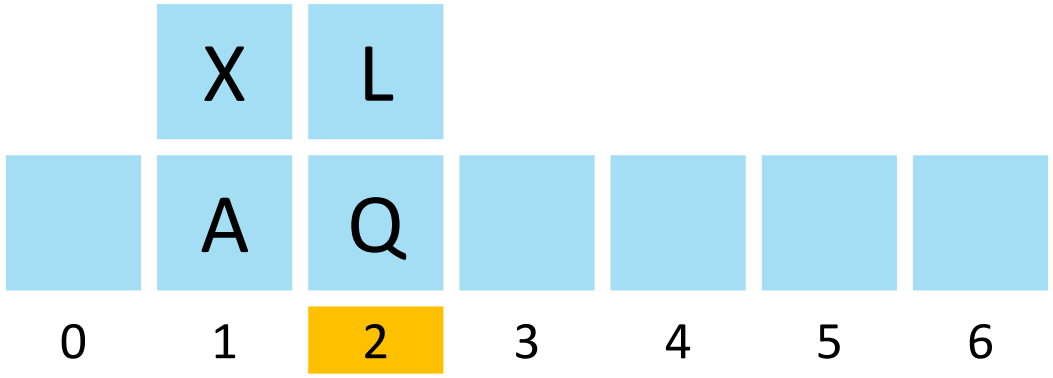


...

$m = 7$

Insertemos la *F*

$h(F) = 58;\ 58 \bmod 7\ = 2$

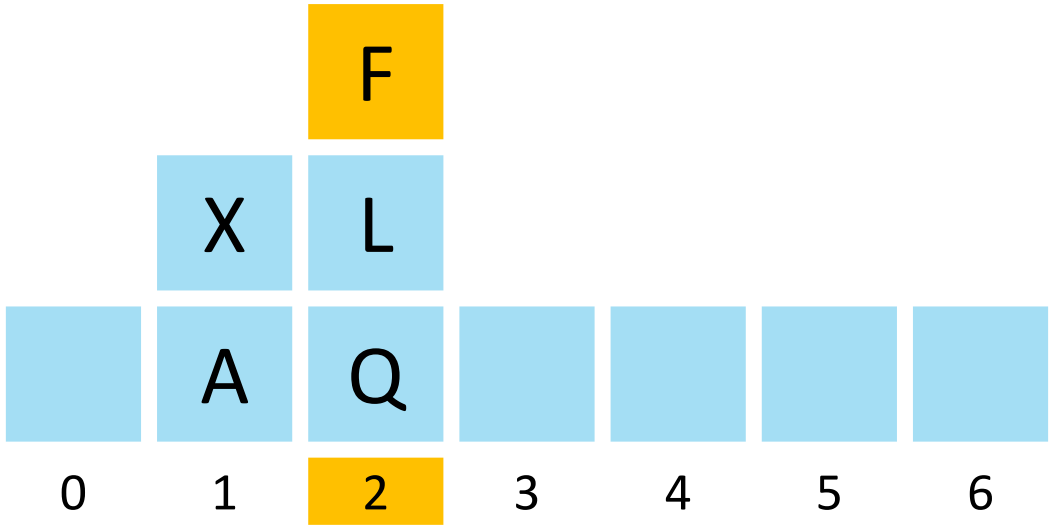


Fin del ejemplo

$m = 7$

Insertemos la *F*

$h(F) = 58;\ 58 \bmod 7\ = 2$



Factor de carga

Se define el factor de carga λ como:

$$\lambda = \frac{n}{m}$$

Podemos fijar un valor λ_{max}, y garantizar que

$$\lambda < \lambda_{max}$$

Rehashing

Si $\lambda < \lambda_{max}$, en algún momento hay que hacer crecer la tabla

A este proceso se le dice **rehashing**

¿Cuál es su complejidad?

¿Qué complejidad tendrían ahora las operaciones de la tabla?