Tablas de hash (2)

mat-151

Tablas de hash: **resolviendo colisiones por encadenamiento**

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

Tablas de hash: **resolviendo colisiones por encadenamiento**

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1)$$

Tablas de hash: **resolviendo colisiones por encadenamiento**

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

Tablas de hash: **resolviendo colisiones por encadenamiento**

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

Tablas de hash: **resolviendo colisiones por encadenamiento**

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Tablas de hash: **resolviendo colisiones por encadenamiento**

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Cuándo es $E[n_j] = \Theta(1)$?

Tablas de hash: resolviendo colisiones por encadenamiento

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Cuándo es $E[n_j] = \Theta(1)$?

- cuando el número de elementos en total es proporcional al número de elementos en la tabla tenemos

Tablas de hash: resolviendo colisiones por encadenamiento

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Cuándo es $E[n_j] = \Theta(1)$?

- cuando el número de elementos en total es proporcional al número de elementos en la tabla tenemos

$$\mathrm{n} = \mathrm{O}(\mathrm{m})$$

Tablas de hash: resolviendo colisiones por encadenamiento

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Cuándo es $E[n_j] = \Theta(1)$?

- cuando el número de elementos en total es proporcional al número de elementos en la tabla tenemos

$$\mathrm{n} = \mathrm{O}(\mathrm{m})$$

$$\alpha = n/m$$

Tablas de hash: resolviendo colisiones por encadenamiento

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Cuándo es $E[n_j] = \Theta(1)$?

- cuando el número de elementos en total es proporcional al número de elementos en la tabla tenemos

$$\mathrm{n} = \mathrm{O}(\mathrm{m})$$

$$\alpha = n/m \ = O(m)/m = O(1)$$

Tablas de hash: resolviendo colisiones por encadenamiento

$$E\left[n_j\right]_{\text{no exitosa}} = \text{cálculo de } \mathcal{H} + \text{búsqueda en la lista} + \text{acceso a elemento.}$$

$$E[n_j]_{\text{no exitosa}} = \Theta(1) + \Theta(\alpha) + \Theta(1) \ = \Theta(1+\alpha)$$

$$\text{Si } \alpha > 1 \text{ entonces } \Theta(\alpha).$$

$$\text{Si } \alpha = 1 \text{ entonces } \Theta(1).$$

Cuándo es $E[n_j] = \Theta(1)$?

- cuando el número de elementos en total es proporcional al número de elementos en la tabla tenemos

$$\mathrm{n} = \mathrm{O}(\mathrm{m})$$

$$\alpha = n/m \ = O(m)/m = O(1)$$

- Tiempo de búsqueda constante en promedio.

Tablas de hash: **resolviendo colisiones por encadenamiento**

Tablas de hash: **resolviendo colisiones por encadenamiento**

- Para una búsqueda exitosa también se puede probar (libro de CLRS, p.p.227-228) que $\Theta(1+\alpha)$ en promedio.

Tablas de hash: **resolviendo colisiones por encadenamiento**

- Para una búsqueda exitosa también se puede probar (libro de CLRS, p.p.227-228) que $\Theta(1+\alpha)$ en promedio.

    - Tiempo de búsqueda constante en promedio.

Tablas de hash: **características de una buena función de hash**

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?
  - Que satisfaga la suposición de hashing simple uniforme.

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?
  - Que satisfaga la suposición de hashing simple uniforme.
  - Que la regularidad no afecte la distribución (ej. números pares): independiente de cualquier patrón que exista en las llaves.

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?
  - Que satisfaga la suposición de hashing simple uniforme.
  - Que la regularidad no afecte la distribución (ej. números pares): independiente de cualquier patrón que exista en las llaves.
  - Que nos permita minimizar el número de colisiones.

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?
  - Que satisfaga la suposición de hashing simple uniforme.
  - Que la regularidad no afecte la distribución (ej. números pares): independiente de cualquier patrón que exista en las llaves.
  - Que nos permita minimizar el número de colisiones.
- Una probabilidad no nula de tener colisión es inevitable para cualquier función de hash en la práctica.

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?
  - Que satisfaga la suposición de hashing simple uniforme.
  - Que la regularidad no afecte la distribución (ej. números pares): independiente de cualquier patrón que exista en las llaves.
  - Que nos permita minimizar el número de colisiones.
- Una probabilidad no nula de tener colisión es inevitable para cualquier función de hash en la práctica.
- La información cualitativa de la distribución de las llaves puede ser útil al momento del diseño.

Tablas de hash: **características de una buena función de hash**

- ¿Qué se espera de una buena función de hash?
  - Que satisfaga la suposición de hashing simple uniforme.
  - Que la regularidad no afecte la distribución (ej. números pares): independiente de cualquier patrón que exista en las llaves.
  - Que nos permita minimizar el número de colisiones.
- Una probabilidad no nula de tener colisión es inevitable para cualquier función de hash en la práctica.
- La información cualitativa de la distribución de las llaves puede ser útil al momento del diseño.
- Dos métodos clásicos: método de división y método de multiplicación.

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.
- por ejemplo:

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.
- por ejemplo:
- generalmente esta aproximación es buena para cualquier valor de m, pero en algunos casos se necesitará una elección más cuidadosa de m:

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.
- por ejemplo: $\text{Si } m = 12 \text{ y } k = 100, \text{ entonces } \mathcal{H}(k) = 4.$
- generalmente esta aproximación es buena para cualquier valor de m, pero en algunos casos se necesitará una elección más cuidadosa de m:
  - ¿hacer m par?

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.
- por ejemplo: $\text{Si } m = 12 \text{ y } k = 100, \text{ entonces } \mathcal{H}(k) = 4.$
- generalmente esta aproximación es buena para cualquier valor de m, pero en algunos casos se necesitará una elección más cuidadosa de m:
  - ¿hacer m par? $\mathcal{H}(k)$ es par si k es par.

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.
- por ejemplo: $\text{Si } m = 12 \text{ y } k = 100, \text{ entonces } \mathcal{H}(k) = 4.$
- generalmente esta aproximación es buena para cualquier valor de m, pero en algunos casos se necesitará una elección más cuidadosa de m:
  - ¿hacer m par? $\mathcal{H}(k)$ es par si k es par. $\mathcal{H}(k)$ es impar si k es impar.

Función de hash: **método de división**

$$\mathcal{H}(k) = k \bmod m$$

- se mapea la llave **k** en la tabla **T** de **m** campos tomando el residuo de k dividido por m.
- por ejemplo: $\text{Si } m = 12 \text{ y } k = 100, \text{ entonces } \mathcal{H}(k) = 4.$
- generalmente esta aproximación es buena para cualquier valor de m, pero en algunos casos se necesitará una elección más cuidadosa de m:
  - ¿hacer m par? $\mathcal{H}(k)$ es par si k es par. $\mathcal{H}(k)$ es impar si k es impar.
    - si las llaves son equiprobables no hay problema pero si las llaves tienen mayor probabilidad de ser pares o impares, la función no distribuirá las llaves en la tabla de manera uniforme.

Función de hash: **método de división**

si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

Función de hash: **método de división**

- ¿ hacer m una potencia de 2

si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.

$$\text{si } m = 2^p, \text{ entonces } \mathcal{H}(k) \text{ es solo los } p \text{ bits más bajos de } k.$$

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.
  - Si k=10110001110110$_2$ y p=6, entonces h(k)=011010$_2$

si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.
  - Si k=$1011000111\mathbf{011010}_2$ y p=6, entonces h(k)=011010_2

    si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.
  - Recuerda que dividir por 2^p es hacer un corrimiento a la derecha.

```
50%8 = 6*8+2
78%8 =  9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4
```

Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.
  - Si k=10110001110110$_2$ y p=6, entonces h(k)=011010$_2$

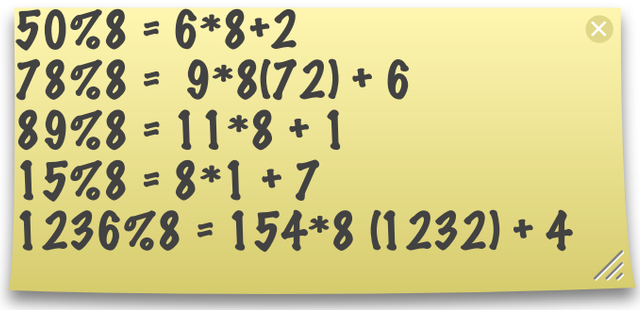
    si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.
  - Recuerda que dividir por 2^p es hacer un corrimiento a la derecha.
    - Se puede implementar x % 2^p == x & (2^p - 1)

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.
  - Si k=10110001110110$_2$ y p=6, entonces h(k)=011010$_2$

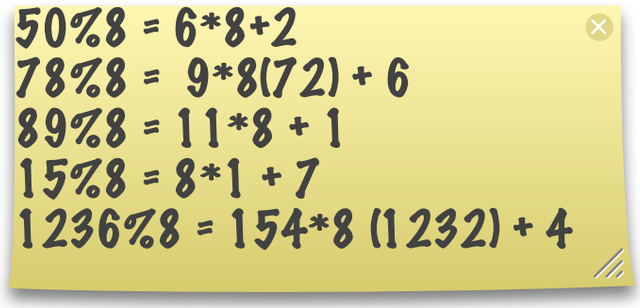
    si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.
  - Recuerda que dividir por 2^p es hacer un corrimiento a la derecha.
    - Se puede implementar x % 2^p == x & (2^p - 1)
  - los p-bits más bajos de las llaves deberían tener la misma probabilidad.



Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.
  - Si k=1011000111**011010**$_2$ y p=6, entonces h(k)=011010$_2$

    si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.
  - Recuerda que dividir por 2^p es hacer un corrimiento a la derecha.
    - Se puede implementar x % 2^p == x & (2^p - 1)
- los p-bits más bajos de las llaves deberían tener la misma probabilidad.
- Con un número cercano a una potencia de 2 es similar.



Función de hash: **método de división**

- ¿ hacer m una potencia de 2
  - el mapeo no depende de todos los bits de k.
  - Si k=10110001110110$_2$ y p=6, entonces h(k)=011010$_2$

    si $m = 2^p$, entonces $\mathcal{H}(k)$ es solo los p bits más bajos de k.
  - Recuerda que dividir por 2^p es hacer un corrimiento a la derecha.
    - Se puede implementar x % 2^p == x & (2^p - 1)
  - los p-bits más bajos de las llaves deberían tener la misma probabilidad.
  - Con un número cercano a una potencia de 2 es similar.

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

50%8 = 6*8+2
78%8 = 9*8(72) + 6
89%8 = 11*8 + 1
15%8 = 8*1 + 7
1236%8 = 154*8 (1232) + 4

Función de hash: **método de división**

Función de hash: **método de división**

- ¿hacer m un número primo?
  - Elegir un primo que no esté muy cerca de una potencia 2 o 10.
  - Ej. queremos hacer una tabla de hash que resuelva colisiones por encadenamiento y que almacene cerca de n=2000 cadenas de caracteres donde cada caracter tiene 8 bits.
    - Podemos soportar (verificar) en promedio 3 elementos en una lista de colisiones, entonces podemos poner m=701 (primo cerca de 2000/3 no muy cercano a una potencia de 2.

Función de hash: **método de división**

- ¿hacer m un número primo?
  - Elegir un primo que no esté muy cerca de una potencia 2 o 10.
  - Ej. queremos hacer una tabla de hash que resuelva colisiones por encadenamiento y que almacene cerca de n=2000 cadenas de caracteres donde cada caracter tiene 8 bits.
    - Podemos soportar (verificar) en promedio 3 elementos en una lista de colisiones, entonces podemos poner m=701 (primo cerca de 2000/3 no muy cercano a una potencia de 2.
    - ver http://srinvis.blogspot.mx/2006/07/hash-table-lengths-and-prime-numbers.html

Función de hash: **método de división**

Función de hash: **método de división**

- La función de hash sería:

Función de hash: **método de división**

- La función de hash sería:

Función de hash: **método de división**

- La función de hash sería:

- En C++:

Función de hash: **método de división**

- La función de hash sería:

- En C++:

Función de hash: **método de división**

- La función de hash sería:

- En C++:

Función de hash: **método de división**

- La función de hash sería:

- En C++:

Función de hash: **método de división**

- La función de hash sería:

- En C++:

- el cálculo de la función de hash se realiza en tiempo constante O(1).

Función de hash: **método de división**

- La función de hash sería:

- En C++:

- el cálculo de la función de hash se realiza en tiempo constante O(1).
- desventaja: llaves consecutivas se mapean a campos consecutivos.

Función de hash: **método de división**

- La función de hash sería:

$$\mathcal{H}(k) = k \bmod 701$$

- En C++:

- el cálculo de la función de hash se realiza en tiempo constante O(1).
- desventaja: llaves consecutivas se mapean a campos consecutivos.

Función de hash: **método de división**

- La función de hash sería:

$$\mathcal{H}(k) = k \bmod 701$$

- En C++:

```
unsigned int m = 701; // número primo

unsigned int h ( unsigned int k )
{
    return k % m;
}
```

- el cálculo de la función de hash se realiza en tiempo constante O(1).
- desventaja: llaves consecutivas se mapean a campos consecutivos.

Función de hash: **método de multiplicación**

Función de hash: **método de multiplicación**

- Este método funciona en dos pasos:

Función de hash: **método de multiplicación**

- Este método funciona en dos pasos:
  - Multiplicar la llave k por una constante **A** en el rango 0 < A < 1 y extraer la parte fraccionaria de kA.

Función de hash: **método de multiplicación**

- Este método funciona en dos pasos:
  - Multiplicar la llave k por una constante **A** en el rango 0 < A < 1 y extraer la parte fraccionaria de kA.
  - Multiplicar este valor por m y tomar la función floor (entero más grande menor o igual al valor) del resultado.

Función de hash: **método de multiplicación**

- Este método funciona en dos pasos:
  - Multiplicar la llave k por una constante **A** en el rango 0 < A < 1 y extraer la parte fraccionaria de kA.
  - Multiplicar este valor por m y tomar la función floor (entero más grande menor o igual al valor) del resultado.

$$\mathcal{H}(k) = \lfloor m(k\ A\ mod\ 1) \rfloor$$

Función de hash: **método de multiplicación**

- Este método funciona en dos pasos:
  - Multiplicar la llave k por una constante **A** en el rango 0 < A < I y extraer la parte fraccionaria de kA.
  - Multiplicar este valor por m y tomar la función floor (entero más grande menor o igual al valor) del resultado.

$$\mathcal{H}(k) = \lfloor m(k\ A\ mod\ 1) \rfloor$$

donde "$k\ A\ mod\ 1$" significa la parte fraccionaria de kA, que es, $kA - \lfloor kA \rfloor$.

Función de hash: **método de multiplicación**

- Este método funciona en dos pasos:
  - Multiplicar la llave k por una constante **A** en el rango 0 < A < 1 y extraer la parte fraccionaria de kA.
  - Multiplicar este valor por m y tomar la función floor (entero más grande menor o igual al valor) del resultado.

$$\mathcal{H}(k) = \lfloor m(k\ A\ mod\ 1) \rfloor$$

donde "$k\ A\ mod\ 1$" significa la parte fraccionaria de kA, que es, $kA - \lfloor kA \rfloor$.

- El valor de m no es crítico, generalmente se elige un valor potencia de 2 (2^P) por facilidad de implementación.

Función de hash: **método de multiplicación**

Función de hash: **método de multiplicación**

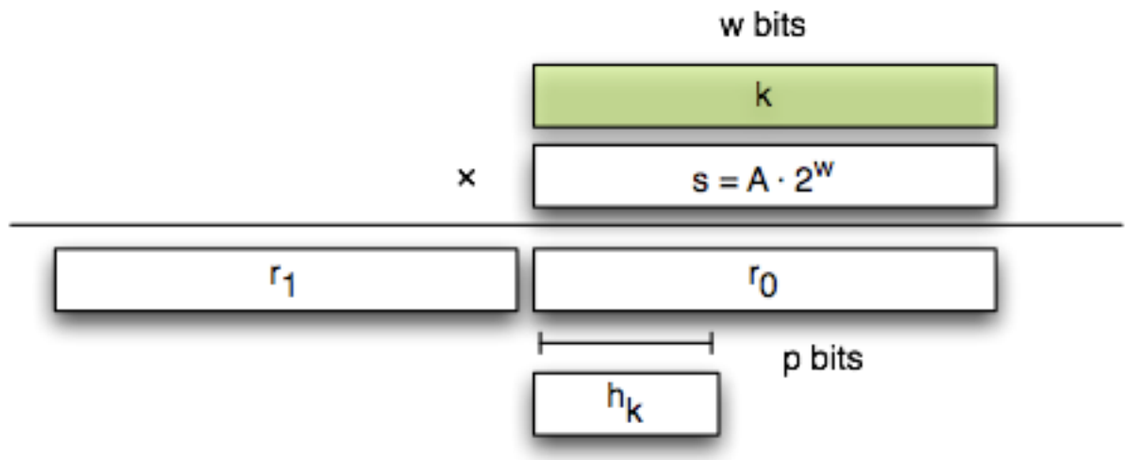
- Suponemos que el tamaño de la palabra de una computadora es w bits y k se expresa en una sola palabra.

Función de hash: **método de multiplicación**

- Suponemos que el tamaño de la palabra de una computadora es w bits y k se expresa en una sola palabra.
- Restringimos A a una fracción de la forma $s/2^w$, donde s es un entero en el rango $0 < s < 2^w$.

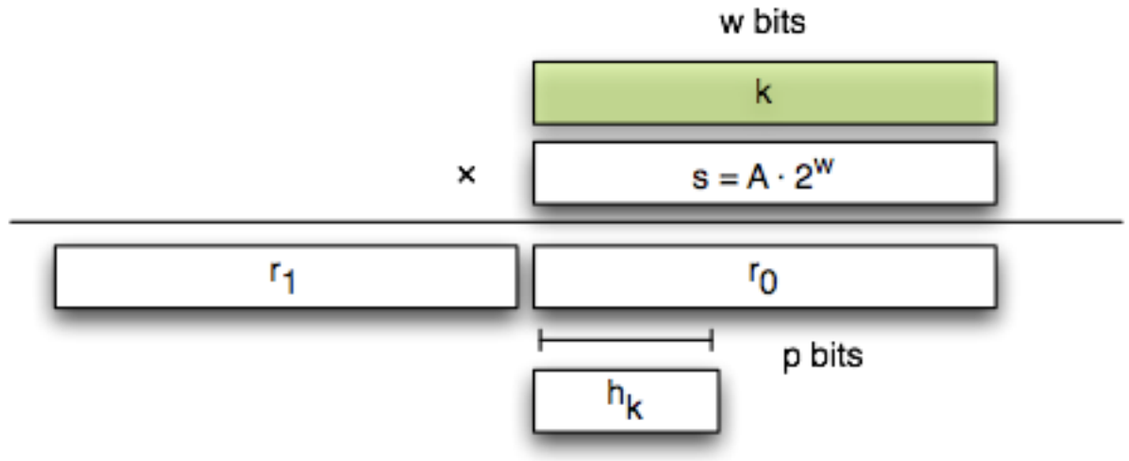
Función de hash: **método de multiplicación**

- Suponemos que el tamaño de la palabra de una computadora es w bits y k se expresa en una sola palabra.
- Restringimos A a una fracción de la forma $s/2^w$, donde s es un entero en el rango $0 < s < 2^w$.



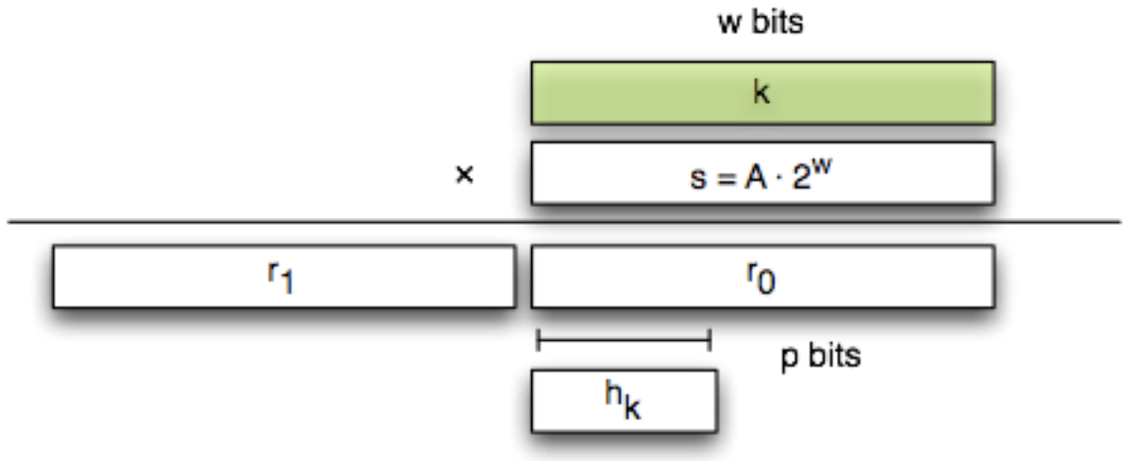
Función de hash: **método de multiplicación**

- Suponemos que el tamaño de la palabra de una computadora es w bits y k se expresa en una sola palabra.
- Restringimos A a una fracción de la forma $s/2^w$, donde s es un entero en el rango $0 < s < 2^w$.
- Multiplicamos primero k por el entero de w-bits s.



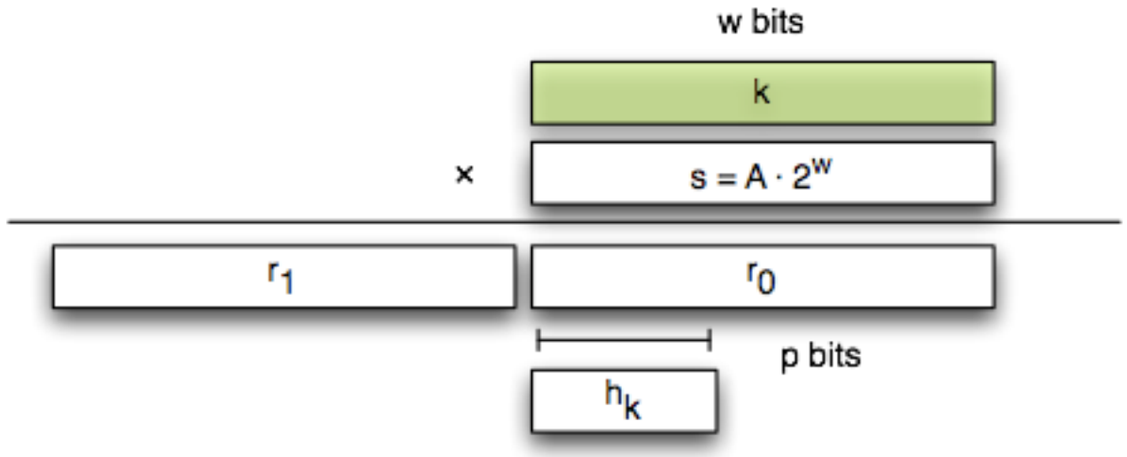
Función de hash: **método de multiplicación**

- Suponemos que el tamaño de la palabra de una computadora es w bits y k se expresa en una sola palabra.
- Restringimos A a una fracción de la forma $s/2^w$, donde s es un entero en el rango $0 < s < 2^w$.
- Multiplicamos primero k por el entero de w-bits s.
- El resultado es un valor de 2w-bits $r_1 2^w + r_0$, donde r_1 es la palabra superior del producto y r_0 la palabra inferior.



Función de hash: **método de multiplicación**

- Suponemos que el tamaño de la palabra de una computadora es w bits y k se expresa en una sola palabra.
- Restringimos A a una fracción de la forma $s/2^w$, donde s es un entero en el rango $0 < s < 2^w$.
- Multiplicamos primero k por el entero de w-bits s.
- El resultado es un valor de 2w-bits $r_1 2^w + r_0$, donde r_1 es la palabra superior del producto y r_0 la palabra inferior.
- El valor de hash de p-bits consiste en los p bits más significativos de r_0.



Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=}32.$$

67

Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=32}.$$

Tomando la sugerencia de Knuth, elegimos A como una fracción de manera que $s/2^{32}$ sea lo más cercano a $(\sqrt{5}-1)/2$, entonces $A = 2654435769/2^{32}$.

67

Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=32}.$$

Tomando la sugerencia de Knuth, elegimos A como una fracción de manera que $s/2^{32}$ sea lo más cercano a $(\sqrt{5}-1)/2$, entonces $A = 2654435769/2^{32}$.

$$\text{Entonces } k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$$

67

Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=32}.$$

Tomando la sugerencia de Knuth, elegimos A como una fracción de manera que $s/2^{32}$ sea lo más cercano a $(\sqrt{5}-1)/2$, entonces $A = 2654435769/2^{32}$.

$$\text{Entonces } k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$$

$$r_1 = 76300 \text{ y } r_0 = 17612864$$

67

Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=32}.$$

Tomando la sugerencia de Knuth, elegimos A como una fracción de manera que $s/2^{32}$ sea lo más cercano a $(\sqrt{5}-1)/2$, entonces $A = 2654435769/2^{32}$.

$$\text{Entonces } k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$$

$$r_1 = 76300 \text{ y } r_0 = 17612864$$

1001010100000011000000010000110011000000001000000

67

Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=32}.$$

Tomando la sugerencia de Knuth, elegimos A como una fracción de manera que $s/2^{32}$ sea lo más cercano a $(\sqrt{5}-1)/2$, entonces $A = 2654435769/2^{32}$.

$$\text{Entonces } k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$$

$$r_1 = 76300 \text{ y } r_0 = 17612864$$

1001010100000011 00 0000001000001100110000000001000000

Los 14 bits más significativos de r_0 nos dan el valor de $\mathcal{H}(k) =$ 67

Función de hash: **método de multiplicación**

- Ejemplo

$$k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ \text{w=32}.$$

Tomando la sugerencia de Knuth, elegimos A como una fracción de manera que $s/2^{32}$ sea lo más cercano a $(\sqrt{5}-1)/2$, entonces $A = 2654435769/2^{32}$.

$$\text{Entonces } k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$$

$$r_1 = 76300 \text{ y } r_0 = 17612864$$

1001010100000110000000100000110011000000001000000

$$\text{Los 14 bits más significativos de } r_0 \text{ nos dan el valor de } \mathcal{H}(k) = \quad 67$$

Para profundizar, ver Sección 6.4 Hashing, Art of computing programming, Donald. E. Knuth.

Función de hash

- Heurísticas para la función de hash.
- Siempre encontraremos un conjunto valores de llave que hagan que una función de hash no distribuya bien los valores provocando colisiones.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.
- La tabla se puede “llenar” de tal manera que no se puedan hacer más inserciones

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.
- La tabla se puede “llenar” de tal manera que no se puedan hacer más inserciones
- ¿Qué podemos decir sobre el factor de carga?

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.
- La tabla se puede “llenar” de tal manera que no se puedan hacer más inserciones
- ¿Qué podemos decir sobre el factor de carga?
  - α nunca puede ser más grande de UNO.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.
- La tabla se puede “llenar” de tal manera que no se puedan hacer más inserciones
- ¿Qué podemos decir sobre el factor de carga?
  - α nunca puede ser más grande de UNO.
- Cada elemento contiene un elemento o NULL, evita usar apuntadores.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.
- La tabla se puede “llenar” de tal manera que no se puedan hacer más inserciones
- ¿Qué podemos decir sobre el factor de carga?
  - α nunca puede ser más grande de UNO.
- Cada elemento contiene un elemento o NULL, evita usar apuntadores.
- La memoria liberada sirve para almacenar más elementos en la tabla (menos colisiones, acceso más rápido).

Tablas de hash: **resolviendo colisiones por direccionamiento abierto (open addressing)**

- Todos los elementos se almacenan en la misma tabla hash.
- Si el campo está lleno, vuelvo a aplicar la función de hash.
- La tabla se puede “llenar” de tal manera que no se puedan hacer más inserciones
- ¿Qué podemos decir sobre el factor de carga?
  - α nunca puede ser más grande de UNO.
- Cada elemento contiene un elemento o NULL, evita usar apuntadores.
- La memoria liberada sirve para almacenar más elementos en la tabla (menos colisiones, acceso más rápido).
- La inserción examina o prueba sistemáticamente la tabla hasta encontrar un campo vacio.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- El proceso de prueba, en lugar de tener un orden fijo 0,1,...,m-1 (cuánto tiempo tomaría una búsqueda?)

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- El proceso de prueba, en lugar de tener un orden fijo 0,1,...,m-1 (cuánto tiempo tomaría una búsqueda?)
  - la búsqueda tomaría Θ(n)

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- El proceso de prueba, en lugar de tener un orden fijo 0,1,...,m-1 (cuánto tiempo tomaría una búsqueda?)
  - la búsqueda tomaría Θ(n)
- La secuencia de prueba depende de llaves que fueron insertadas antes.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- El proceso de prueba, en lugar de tener un orden fijo 0,1,...,m-1 (cuánto tiempo tomaría una búsqueda?)
  - la búsqueda tomaría Θ(n)
- La secuencia de prueba depende de llaves que fueron insertadas antes.
- La función de hash depende ahora de ambas, de la llave y del número de prueba (empieza por 0):

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- El proceso de prueba, en lugar de tener un orden fijo 0,1,...,m-1 (cuánto tiempo tomaría una búsqueda?)
  - la búsqueda tomaría Θ(n)
- La secuencia de prueba depende de llaves que fueron insertadas antes.
- La función de hash depende ahora de ambas, de la llave y del número de prueba (empieza por 0):

$$\mathcal{H} : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}.$$

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- El proceso de prueba, en lugar de tener un orden fijo 0,1,...,m-1 (cuánto tiempo tomaría una búsqueda?)
  - la búsqueda tomaría Θ(n)
- La secuencia de prueba depende de llaves que fueron insertadas antes.
- La función de hash depende ahora de ambas, de la llave y del número de prueba (empieza por 0):

$$\mathcal{H} : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}.$$

- La secuencia de prueba $< \mathcal{H}(k,0), \mathcal{H}(k,1), \ldots, \mathcal{H}(k,m-1) >$ para una llave k, debe ser una permutación de $< 0, 1, \ldots, m-1 >$ para que cada campo de la tabla sea eventualmente considerado mientras esta se llena.

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- La eliminación es difícil: no puede solo sustituirse por el valor NULL, hay que usar otro marcador: DELETED (si no es posible es más apropiado usar chaining)

HASH-INSERT(T, k)

```
1.  i ← 0
2.  repeat j ← h(k,i)
3.          if T[ j ] = NULL
4.                  then T[ j ] ← k
5.                       return j
6.                  else i ← i+1
7.  until i = m
8.  error "hash table overflow"
```

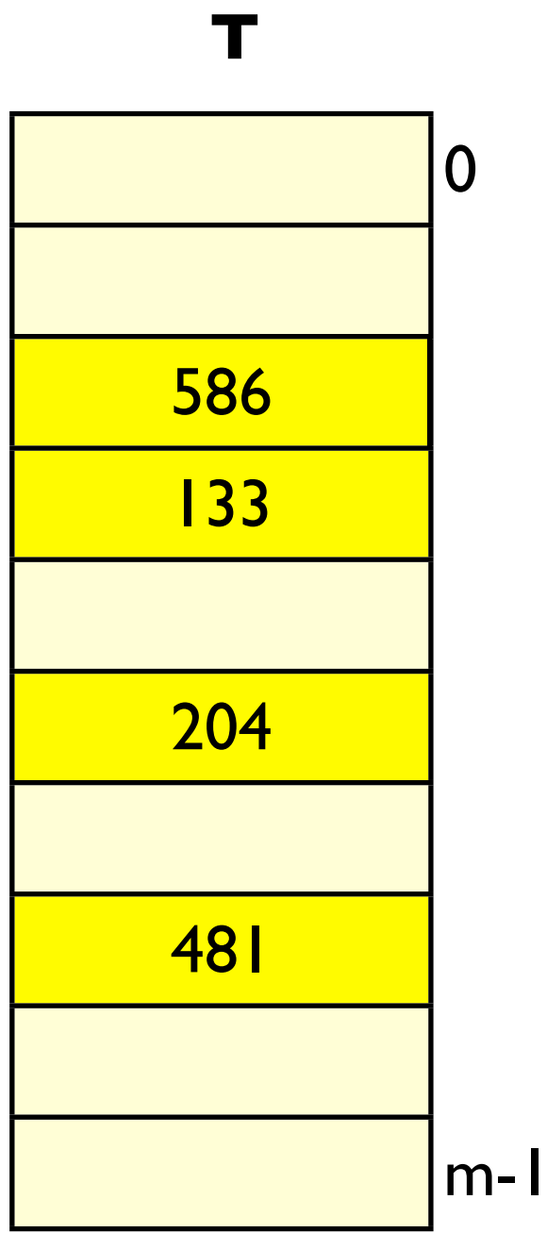
HASH-SEARCH(T, k)

```
• i ← 0
1.  repeat j ← h( k, i )
2.          if T[ j ] = k
3.                  then return j
4.                  else i ← i+1
• until T[ j ] = NULL o i = m
• return  NULL
```

Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- Ejemplo: insertar llave k=496

- La búsqueda usa la misma secuencia de prueba, terminando exitosamente si encuentra la llave y no exitosamente si encuentra una llave vacía.



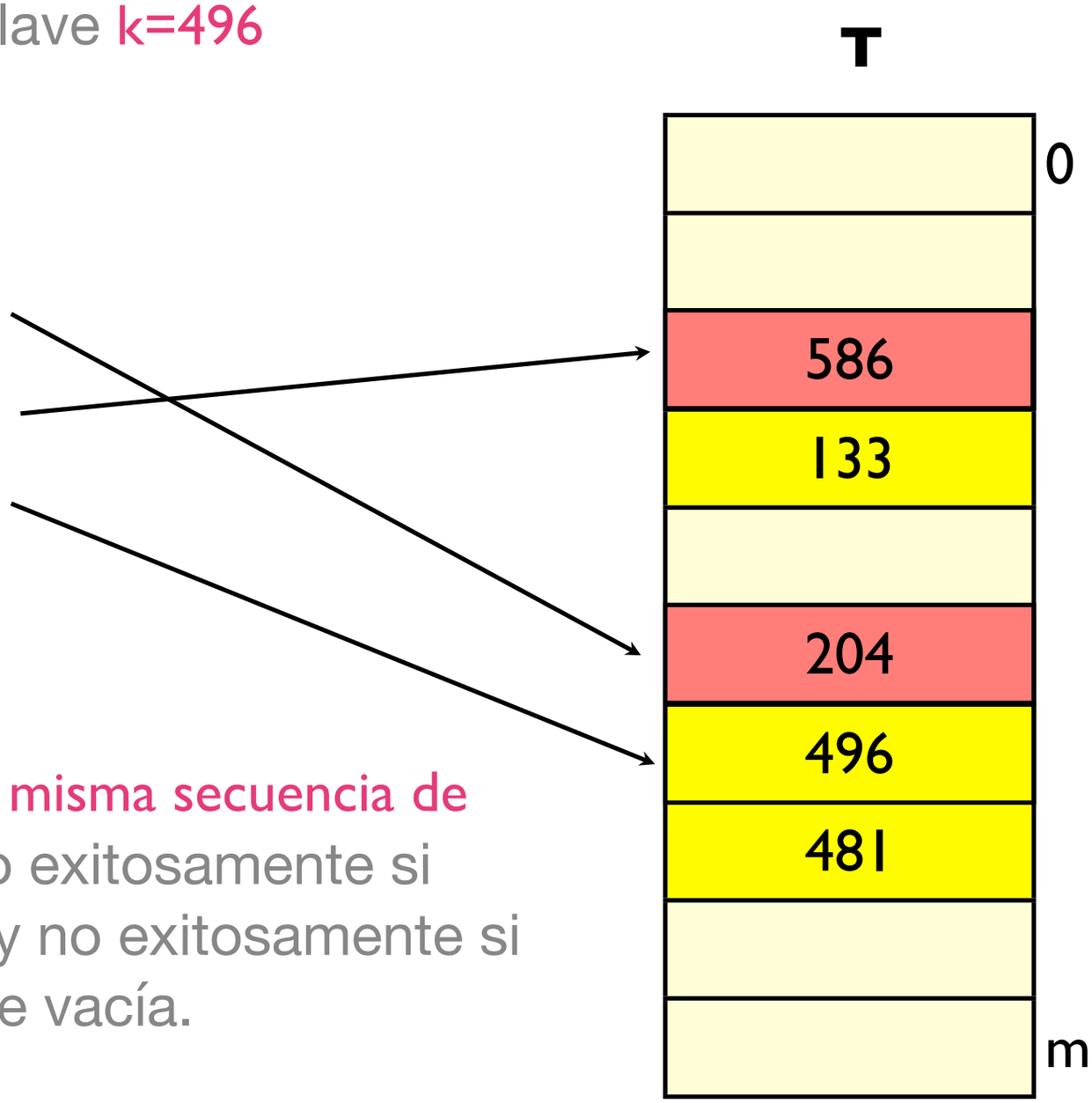
Tablas de hash: **resolviendo colisiones por direccionamiento abierto**

- Ejemplo: insertar llave k=496

Probe h(496, 0)

Probe h(496, 1)

Probe h(496, 2)



- La búsqueda usa la misma secuencia de prueba, terminando exitosamente si encuentra la llave y no exitosamente si encuentra una llave vacía.

Tablas de hash: **secuencias de prueba**

- Se usan generalmente tres técnicas para obtener una secuencia de prueba:
  - prueba lineal (linear probing)
  - prueba cuadrática (quadratic probing)
  - hashing doble (double hashing)
- Estas técnicas garantizan que la secuencia será una permutación de los indices de la tabla para cada llave k.
- Ninguna puede generar más de $\mathbf{m^2}$ secuencias de prueba, y no las **m!** que requiere un hashing uniforme.
- El hashing doble produce el mayor número de secuencias de prueba.

Tablas de hash: **prueba lineal**

Dada una función de hash ordinaria $\mathcal{H}'(k) : U \to \{0, 1, \ldots, m-1\}$, el método de prueba lineal utiliza la función de hash:

$$\mathcal{H}(k, i) = (\mathcal{H}'(k) + i) \bmod m$$

para $i = 0, 1, \ldots, m-1$.

Tablas de hash: **prueba lineal**

Dada una función de hash ordinaria $\mathcal{H}'(k) : U \to \{0, 1, \ldots, m-1\}$, el método de prueba lineal utiliza la función de hash:

$$\mathcal{H}(k, i) = (\mathcal{H}'(k) + i) \bmod m$$

para $i = 0, 1, \ldots, m-1$.

- Este método es simple pero sufre de clustering primario que causa el aumento del tiempo de búsqueda.
- La prueba inicial determina la secuencia, por lo que solo hay m secuencias distintas.

Tablas de hash: **prueba cuadrática**

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}'(k) + c_1 i + c_2 i^2) \bmod m.$$

Tablas de hash: **prueba cuadrática**

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}'(k) + c_1 i + c_2 i^2) \bmod m.$$

- Con c_1 y c_2 reales, c_2 diferente de 0.

Tablas de hash: **prueba cuadrática**

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}'(k) + c_1 i + c_2 i^2) \bmod m.$$

- Con c_1 y c_2 reales, c_2 diferente de 0.
- En lugar de agregar 1 se agrega en función de i. (proporcional a dos constantes auxiliares)

Tablas de hash: **prueba cuadrática**

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}'(k) + c_1 i + c_2 i^2) \bmod m.$$

- Con c_1 y c_2 reales, c_2 diferente de 0.
- En lugar de agregar 1 se agrega en función de i. (proporcional a dos constantes auxiliares)
- Si dos llaves tienen la misma posición inicial de prueba, sus secuencias de prueba son las mismas.

Tablas de hash: **prueba cuadrática**

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}'(k) + c_1 i + c_2 i^2) \bmod m.$$

- Con c_1 y c_2 reales, c_2 diferente de 0.
- En lugar de agregar 1 se agrega en función de i. (proporcional a dos constantes auxiliares)
- Si dos llaves tienen la misma posición inicial de prueba, sus secuencias de prueba son las mismas.
- Produce m secuencias distintas, pero aminora el clustering primario.

Tablas de hash: **hashing doble**

- Es uno de los mejores métodos disponibles porque las permutaciones producidas tienen muchas de las características de las elegidas aleatoriamente.

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}_1(k) + i\mathcal{H}_2(k)) \bmod m.$$

- Este método produce generalmente excelentes resultados cuando $\mathcal{H}_2(k)$ es primo relativo a m. Una manera de hacer esto es haciendo m una potencia de 2 y diseñando $\mathcal{H}_2(k)$ para producir solo números impares.

> Sean a, b ∈ Z, se dice que **son primos relativos (o coprimos) "a" y "b" si no tienen ningún factor primo en común**, es decir, si no tienen otro divisor común más que 1 ó -1, o cumplen que **el mcd (a, b) = 1**.

- El intervalo de "salto" depende de los datos, datos que son mapeados al mismo slot viajan por diferentes trayectorias.

Tablas de hash: **hashing doble**

- Llave = 14, tabla de hash de tamaño 13 con $h_1(k)=k \bmod 13$ y $h_2(k) = 1+ (k \bmod 11)$.
- 14 mod 13, entonce $h_1 = 1$
- 1+ 14 mod 11, entonces $h_2 = 4$

- la llave 14 se inserta en el campo vacio 9 después de visitar los campos 1 y 5, que estaban ocupados.

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}_1(k) + i\mathcal{H}_2(k)) \bmod m.$$

Tablas de hash: **hashing doble**

- Llave = 14, tabla de hash de tamaño 13 con $h_1(k)=k \bmod 13$ y $h_2(k) = 1+ (k \bmod 11)$.
- 14 mod 13, entonce $h_1 = 1$
- 1+ 14 mod 11, entonces $h_2 = 4$

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

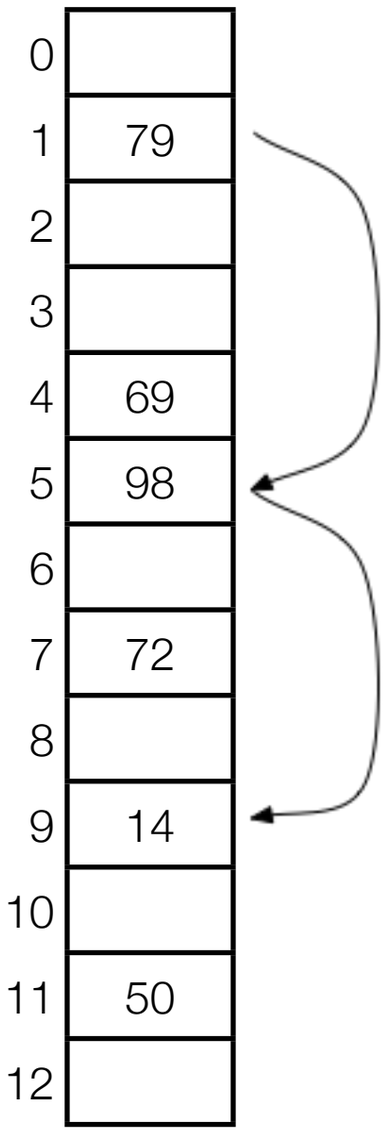
- la llave 14 se inserta en el campo vacio 9 después de visitar los campos 1 y 5, que estaban ocupados.

Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}_1(k) + i\mathcal{H}_2(k)) \bmod m.$$

Tablas de hash: **hashing doble**

- Llave = 14, tabla de hash de tamaño 13 con $h_1(k)=k \bmod 13$ y $h_2(k) = 1+ (k \bmod 11)$.
- 14 mod 13, entonce $h_1 = 1$
- 1+ 14 mod 11, entonces $h_2 = 4$
- la llave 14 se inserta en el campo vacio 9 después de visitar los campos 1 y 5, que estaban ocupados.



Usa una función de hash de la forma

$$\mathcal{H}(k,i) = (\mathcal{H}_1(k) + i\mathcal{H}_2(k)) \bmod m.$$

Tablas de hash: **análisis del direccionamiento abierto**

- Suponiendo que cada llave tiene la misma probabilidad de tener como secuencia de prueba una de las **m!** permutaciones.
- TEOREMA
  - Dada una tabla de hash con direccionamiento abierto con factor de carga $\alpha = n/m < 1$, el número esperado de pruebas en una búsqueda no exitosa es a lo más $1/(1-\alpha)$.

Tablas de hash: **análisis del direccionamiento abierto**

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

- PRUEBA

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

- PRUEBA
  - Se necesita siempre al menos una prueba.

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

- PRUEBA
  - Se necesita siempre al menos una prueba.
  - Con una probabilidad de n/m, la primera prueba encuentra un campo ocupado y requiere de una segunda prueba.

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

- PRUEBA
  - Se necesita siempre al menos una prueba.
  - Con una probabilidad de n/m, la primera prueba encuentra un campo ocupado y requiere de una segunda prueba.
  - Con una probabilidad (n-1)/(m-1), la segunda prueba encontrará un campo ocupado y una tercera prueba es necesaria.

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

- PRUEBA
  - Se necesita siempre al menos una prueba.
  - Con una probabilidad de n/m, la primera prueba encuentra un campo ocupado y requiere de una segunda prueba.
  - Con una probabilidad (n-1)/(m-1), la segunda prueba encontrará un campo ocupado y una tercera prueba es necesaria.
  - Con una probabilidad (n-2)/(m-2), la tercera prueba encontrará un campo ocupado, etc.

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

- PRUEBA
  - Se necesita siempre al menos una prueba.
  - Con una probabilidad de n/m, la primera prueba encuentra un campo ocupado y requiere de una segunda prueba.
  - Con una probabilidad (n-1)/(m-1), la segunda prueba encontrará un campo ocupado y una tercera prueba es necesaria.
  - Con una probabilidad (n-2)/(m-2), la tercera prueba encontrará un campo ocupado, etc.
  - Sabiendo que para m>n:

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ para } i = 1, 2, \ldots, n.$$

Tablas de hash: **análisis del direccionamiento abierto**

$$
\begin{aligned}
&1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)\\
&\le 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots)))\\
&\le 1+\alpha+\alpha^2+\alpha^3+\cdots\\
&=\sum_{i=0}^{\infty}\alpha^i\\
&=\frac{1}{1-\alpha}. \ \square
\end{aligned}
$$

Tablas de hash: **implicaciones del resultado del teorema**

Tablas de hash: **implicaciones del resultado del teorema**

- Si α es constante, entonces accesar una tabla de hash con direccionamiento abierto toma tiempo constante.

Tablas de hash: **implicaciones del resultado del teorema**

- Si α es constante, entonces accesar una tabla de hash con direccionamiento abierto toma tiempo constante.
- Si la tabla esta medio llena, entonces el número esperado de pruebas es 1/(1-0.5) = 2.

Tablas de hash: **implicaciones del resultado del teorema**

- Si α es constante, entonces accesar una tabla de hash con direccionamiento abierto toma tiempo constante.
- Si la tabla esta medio llena, entonces el número esperado de pruebas es 1/(1-0.5) = 2.
- Si la tabla está 90% llena, entonces el número esperado de pruebas es 1/(1-0.9) = 10.

Tablas de hash: **implicaciones del resultado del teorema**

- Si α es constante, entonces accesar una tabla de hash con direccionamiento abierto toma tiempo constante.
- Si la tabla esta medio llena, entonces el número esperado de pruebas es 1/(1-0.5) = 2.
- Si la tabla está 90% llena, entonces el número esperado de pruebas es 1/(1-0.9) = 10.

Tablas de hash: **implicaciones del resultado del teorema**

- Si α es constante, entonces accesar una tabla de hash con direccionamiento abierto toma tiempo constante.
- Si la tabla esta medio llena, entonces el número esperado de pruebas es 1/(1-0.5) = 2.
- Si la tabla está 90% llena, entonces el número esperado de pruebas es 1/(1-0.9) = 10.
- Cuando las tablas se llenan la cosa se empieza a comporta de manera lineal, por eso siempre es bueno tener una tabla hash “amplia”.

Funcion hash para caracteres codificados

```
now 6733767 1816567 55 29
for 6333762 1685490 50 20
tip 7232360 1914096 48  1
ilk 6473153 1734251 43 18
dim 6232355 1651949 45 21
tag 7230347 1913063 39 22
jot 6533764 1751028 52 24
sob 7173742 1898466 34 26
nob 6733742 1816546 34  8
sky 7172771 1897977 57  2
hut 6435364 1719028 52 16
ace 6070745 1602021 37  3
bet 6131364 1618676 52 11
men 6671356 1798894 46 26
egg 6271747 1668071 39 23
few 6331367 1684215 55 16
jay 6530371 1749241 57  4
owl 6775754 1833964 44  4
joy 6533771 1751033 57 29
rap 7130360 1880304 48 30
gig 6372347 1701095 39  1
wee 7371345 1962725 37 22
was 7370363 1962227 51 20
cab 6170342 1634530 34 24
wad 7370344 1962212 36  5
```

Cadena (3 letras de 7 bits c/u)

Octal

Decimal

Mod 64

Mod 31(primo)

Funcion hash para caracteres codificados

```
now 6733767 1816567 55 29
for 6533762 1685490 50 20
tip 7232360 1914036 48  1
ilk 6473153 1734251 43 18
dim 6232655 1651949 45 21
tag 7230347 1913063 39 22
jot 6533764 1751028 52 24
sob 7173742 1898466 34 26
nob 6733742 1816546 34  8
sky 7172771 1897977 57  2
hut 6435364 1719028 52 16
ace 6070745 1602021 37  3
bet 6131364 1618676 52 11
men 6671356 1798894 46 26
egg 6271747 1668071 39 23
few 6331367 1684215 55 16
jay 6530371 1749241 57  4
owl 6775754 1833964 44  4
joy 6533771 1751033 57 29
rap 7130360 1880304 48 30
gig 6372347 1701095 39  1
wee 7371345 1962725 37 22
was 7370363 1962227 51 20
cab 6170342 1634530 34 24
wad 7370344 1962212 36  5
```

Cadena (3 letras de 7 bits c/u)

Octal

Decimal

Mod 64

Mod 31(primo)

Tablas de #'s primos

¿Primos cercanos?

n	δ_n	$2^n - \delta_n$
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139
19	1	524287
20	3	1048573
21	9	2097143
22	3	4194301
23	15	8388593
24	3	16777213
25	39	33554393
26	5	67108859
27	39	134217689
28	57	268435399
29	3	536870909
30	35	1073741789
31	1	2147483647

Funciones Hash de cadenas largas

- Por ejemplo, si tenemos:

averylongkey?

In 7-bit ASCII, this word corresponds to the 84-bit number

$$97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^9 + 114 \cdot 128^8 + 121 \cdot 128^7$$
$$+ 108 \cdot 128^6 + 111 \cdot 128^5 + 110 \cdot 128^4 + 103 \cdot 128^3$$
$$+ 107 \cdot 128^2 + 101 \cdot 128^1 + 121 \cdot 128^0,$$

- Podemos descomponerla como (Algoritmo de Horner):

$$((((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121) \cdot 128$$
$$+ 108) \cdot 128 + 111) \cdot 128 + 110) \cdot 128 + 103) \cdot 128$$
$$+ 107) \cdot 128 + 101) \cdot 128 + 121.$$

Funciones Hash de cadenas largas

- Por ejemplo, si tenemos:

averylongkey?

In 7-bit ASCII, this word corresponds to the 84-bit number

$$97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^9 + 114 \cdot 128^8 + 121 \cdot 128^7$$
$$+ 108 \cdot 128^6 + 111 \cdot 128^5 + 110 \cdot 128^4 + 103 \cdot 128^3$$
$$+ 107 \cdot 128^2 + 101 \cdot 128^1 + 121 \cdot 128^0,$$

Necesitamos solamente guardar el modulo de esto

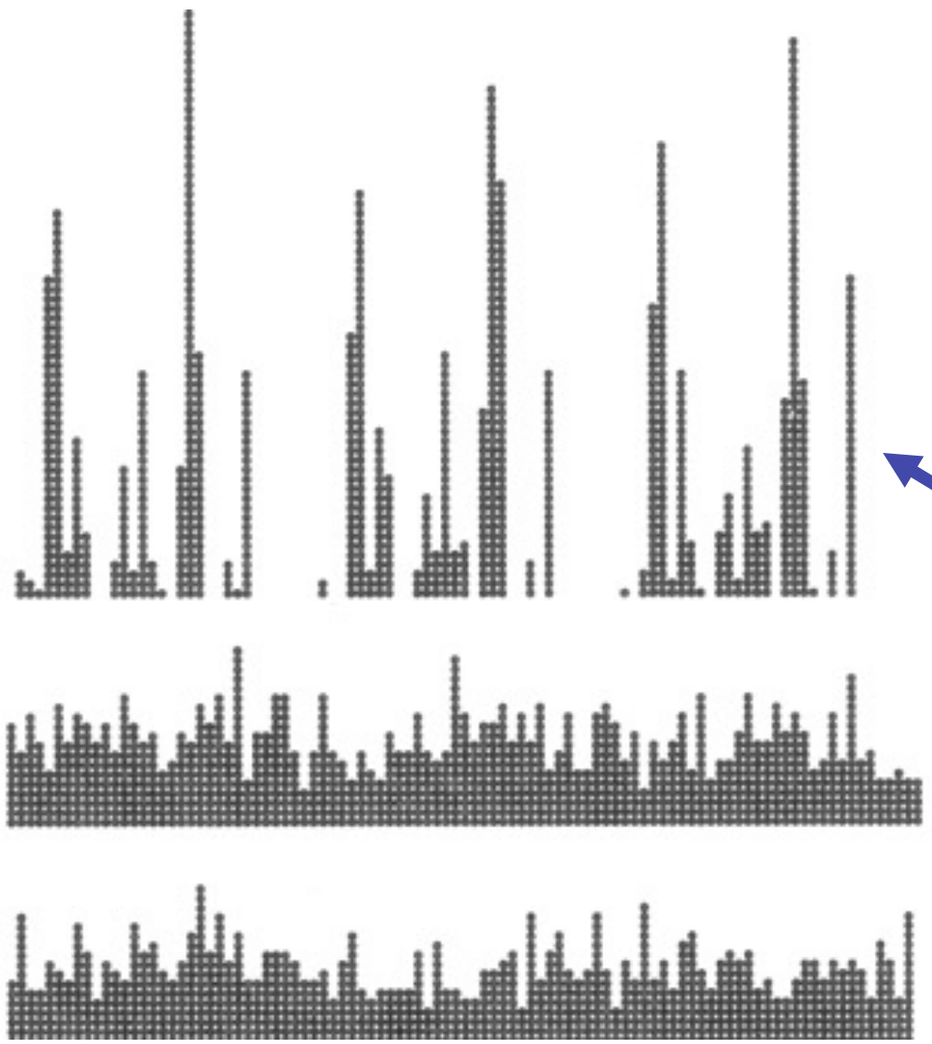
- Podemos descomponerla como (Algoritmo de Horner):

$$((((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121) \cdot 128$$
$$+ 108) \cdot 128 + 111) \cdot 128 + 110) \cdot 128 + 103) \cdot 128$$
$$+ 107) \cdot 128 + 101) \cdot 128 + 121.$$

Funciones Hash de cadenas largas

- Con este código de hash para cadenas

```
int hash(char *v, int M)
  { int h = 0, a = 127;
    for (; *v != 0; v++)
      h = (a*h + *v) % M;
    return h;
  }
```



Para las primeras 1000 palabras distintas del libro Moby Dick.

M=96, a = 128

M=97, a = 128

M=96, a = 127

Un codigo para tener Universal Hash para cadenas

- En lugar de tener un numero ***a*** fijo (radio), lo vamos moviendo de forma pseudo-aleatoria.

```
int hashU(char *v, int M)
  { int h, a = 31415, b = 27183;
    for (h = 0; *v != 0; v++, a = a*b % (M-1))
        h = (a*h + *v) % M;
    return                          h;
  }
```

Esto hace demasiadas operaciones por cada caracter, es mejor tomar pedazos de mayor tamaño (conjunto de bits) de la cadena, pero vamos a tener un máximo en el tamaño de palabra de la máquina.

Hacer un parser de # de palabras por linea

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char str[1024];

    ifstream infile;
    infile.open ("c:\\CIMAT_cable_netsh.txt");


    infile.getline(str,1024);


    char *tokenPtr =strtok(str," ");

    while (tokenPtr != NULL)
    {
        cout<<tokenPtr<<"\n";
        tokenPtr=strtok(NULL," ");
    }

}
```