

# CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

## Hashing

© 2018 Blai Bonet

## Objetivos

- Tablas de hash para implementar de forma eficiente diccionarios
- Métodos de resolución de colisiones: encadenamiento y direccionamiento abierto
- Funciones de hash

© 2018 Blai Bonet

## Introducción

Muchas aplicaciones requieren mantener un **conjunto dinámico** sobre el cual realizar las siguientes operaciones, llamadas **operaciones de diccionario**:

- **Insert**: inserta un elemento al conjunto
- **Search**: determina si un elemento dado existe en el conjunto
- **Delete**: elimina un elemento del conjunto

Una tabla de hash implementa de forma **efectiva** un diccionario

Bajo suposiciones razonables, una tabla de hash es capaz de implementar las tres operaciones en **tiempo promedio constante**

© 2018 Blai Bonet

## Direccionamiento directo

Direccionamiento directo es una técnica simple que es factible cuando el universo de claves  $U$  tiene un **tamaño relativamente pequeño**

Si el universo  $U = \{0, 1, \dots, m - 1\}$  tiene  $m$  claves ( $m$  pequeño), podemos utilizar un arreglo  $T[0 \dots m - 1]$  de dimensión  $m$  para guardar los elementos directamente en el arreglo

Cada posición o "slot" del arreglo se utiliza para guardar un elemento en el conjunto dinámico. Si el conjunto tiene un elemento con clave  $k$ ,  $T[k]$  guarda un "apuntador" a un par que contiene la clave y los **datos satélites** asociados a la clave; en otro caso,  $T[k] = \text{null}$

Con direccionamiento directo, las operaciones de diccionario se implementan en **tiempo constante**

## Direccionamiento directo

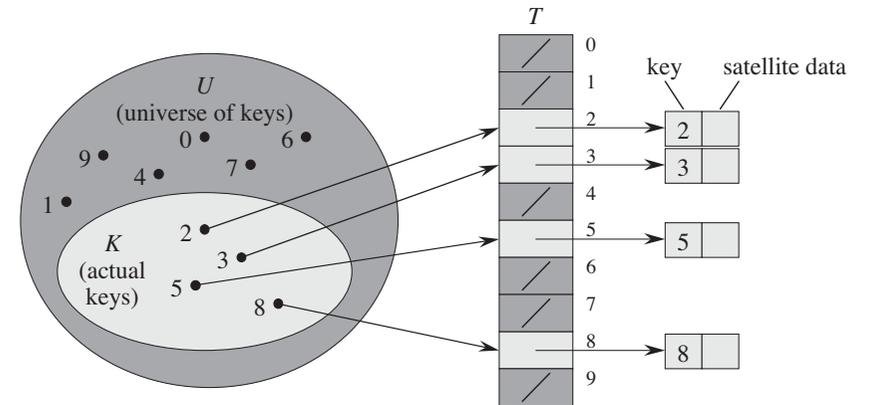


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

## Direccionamiento directo

---

```
1 Direct-Address-Insert(array T, pointer x)
2   T[x.key] = x
3
4 Direct-Address-Search(array T, int k)
5   return T[k]
6
7 Direct-Address-Delete(array T, pointer x)
8   T[x.key] = null
```

---

Cada operación requiere  $\Theta(1)$  tiempo

## Tablas de hash

Las limitaciones del esquema de direccionamiento directo son obvias:

- si el tamaño del universo es muy grande, es imposible/impráctico tener una tabla de tamaño  $|U|$
- en casos donde podemos construir la tabla, el número de objetos a guardar es típicamente mucho menor a  $|U|$  por lo que tendríamos un gran **desperdicio de espacio**

Con una **tabla de hash** podemos invertir  $\Theta(|K|)$  espacio para almacenar un conjunto  $K \subseteq U$  de claves y **garantizar** operaciones que tomen **tiempo promedio**  $\Theta(1)$  bajo condiciones razonables

## Hashing

Con direccionamiento directo, un elemento con clave  $k$  es almacenado en el slot con índice  $k$

Con hashing, el elemento es almacenado en el slot con índice  $h(k)$  donde  $h(\cdot)$  es una **función de hashing**:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

que se utiliza para indexar el arreglo  $T[0 \dots m - 1]$  con  $m \ll |U|$

## Hashing

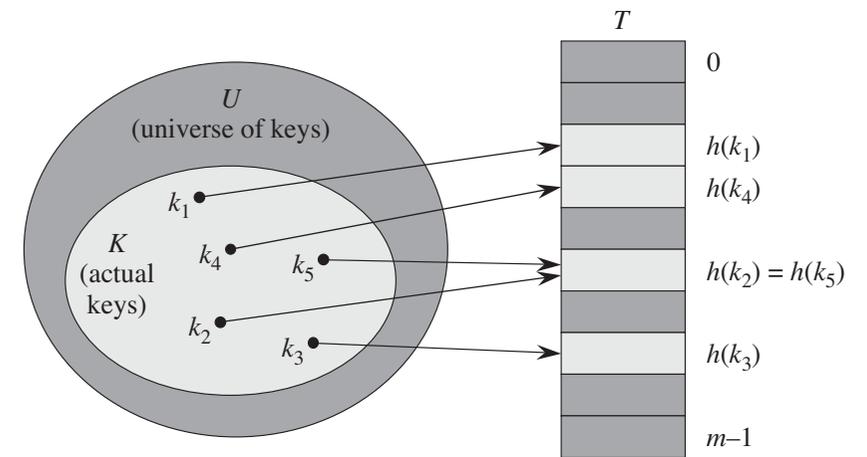


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

## Colisiones

La idea es buena excepto que pueden existir **colisiones**: dos o más claves distintas cuyo valor de hash es el mismo

Existen varias formas de resolver colisiones. Nosotros veremos dos métodos generales para la resolución de colisiones:

- encadenamiento (chaining)
- direccionamiento abierto (open addressing)

Lo ideal es que no existan o reducir al máximo las colisiones. Para esto, la función de hash debe parecer “aleatoria” lo cual significa que claves distintas tengan valores de hash distintos

## Resolución de colisiones con encadenamiento

Bajo encadenamiento todos los elementos asociados al mismo slot son mantenidos en una **lista doble enlazada**

El contenido del slot  $k$ ,  $T[k]$ , es un apuntador a una lista con los elementos en el slot  $k$ , o el apuntador **null** si no hay elementos

## Resolución de colisiones con encadenamiento

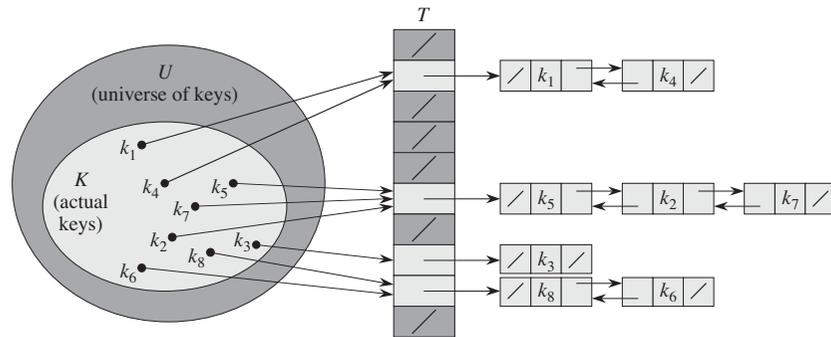


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

## Resolución de colisiones con encadenamiento

```
1 Chained-Hash-Insert(array T, pointer x)
2   k = h(x.key)
3   List-Insert(T[k], x)
4
5 Chained-Hash-Search(array T, key k)
6   return List-Search(T[h(k)], k)
7
8 Chained-Hash-Delete(array T, pointer x)
9   k = h(x.key)
10  List-Delete(T[k], x)
```

## Análisis de encadenamiento

- Insertar un elemento en la cabeza de una lista doble enlazada toma tiempo constante
- Eliminar un elemento de una lista doble enlazada toma tiempo constante (dado el apuntador al elemento)
- Buscar un elemento toma tiempo proporcional al tamaño de la lista

En lo que resta, veremos que bajo suposiciones razonables cada lista en la tabla tiene un **número promedio constante** de elementos

## Análisis peor caso de encadenamiento

Considere una tabla de hash con  $m$  slots y  $n$  elementos

Definimos el **factor de carga**  $\alpha = n/m$  como el número promedio de elementos por slot

El análisis será en función de  $\alpha$ , el cual puede ser menor, igual o mayor a 1

En el peor caso, los  $n$  elementos son mapeados al mismo slot creando una lista enlazada de  $n$  elementos y las búsquedas toman tiempo  $\Theta(n)$  (el tiempo para calcular la función de hash se asume  $O(1)$ )

## Análisis tiempo promedio de encadenamiento 1/4

Supondremos que la función de hash mapea cada elemento a uno de los  $m$  slots con **igual probabilidad** y de forma **independiente** (suposición conocida como **simple uniform hashing**)

Para  $j = 0, 1, \dots, m - 1$ , defina  $n_j$  igual a la longitud de la lista  $T[j]$ . Claramente,  $n = n_0 + n_1 + \dots + n_{m-1}$  es el total de elementos

Considere una búsqueda de la clave  $k$  en la tabla de hash  $T$ . Consideramos dos casos: (1)  $k$  no está en  $T$  y (2)  $k$  está en  $T$

**Caso 1:**  $k$  no está en la tabla  $T$ . El tiempo de búsqueda es igual al número promedio de elementos en la lista  $T[h(k)]$ . Bajo simple uniform hashing,  $\mathbb{E}[n_{h(k)}] = \alpha$  donde  $\alpha = n/m$  es el factor de carga

I.e., tiempo promedio es  $\Theta(\max\{1, \alpha\}) = \Theta(1 + \alpha)$

## Análisis tiempo promedio de encadenamiento 2/4

**Caso 2:**  $k$  está en la tabla  $T$ . Asumimos que la clave  $k$  es una de las  $n$  claves en la tabla con igual probabilidad

El tiempo necesario para buscar  $k$  es 1 + número de elementos que aparecen **antes de  $k$**  en la lista  $T[h(k)]$

Como las inserciones se hacen a la cabeza de la lista, los elementos antes de la clave  $k$  en  $T$  fueron **insertados después** de  $k$

Sean  $x_1, x_2, \dots, x_n$  los elementos en  $T$  con claves  $k_1, k_2, \dots, k_n$  en el **orden en que fueron insertados**

Definimos las variables aleatorias:

- $Z_i = \mathbb{I}\{k = k_i\}$  (1 ssi estamos buscando la clave  $k_i$ )
- $X_{ij} = \mathbb{I}\{h(k_i) = h(k_j)\}$  (1 ssi colisión entre  $k_i$  y  $k_j$ )
- $t(k) = \text{tiempo-búsqueda}(k) = \sum_{i=1}^n Z_i (1 + \sum_{j=i+1}^n X_{ij})$

## Análisis tiempo promedio de encadenamiento 3/4

Bajo simple uniform hashing:  $\mathbb{E}[X_{ij}] = 1/m$

Por la suposición sobre la clave  $k$  a buscar:  $\mathbb{E}[Z_i] = 1/n$

Calculemos  $\mathbb{E}[t(k)]$ :

$$\begin{aligned}\mathbb{E}[t(k)] &= \mathbb{E}\left[\sum_{i=1}^n Z_i \left(1 + \sum_{j=i+1}^n X_{ij}\right)\right] = \sum_{i=1}^n \mathbb{E}[Z_i] \mathbb{E}\left[1 + \sum_{j=i+1}^n X_{ij}\right] \\ &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \mathbb{E}[X_{ij}]\right) = \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) \\ &= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \\ &= \Theta(1 + \alpha)\end{aligned}$$

## Análisis tiempo promedio de encadenamiento 4/4

Conclusiones:

- Si el número  $n$  de elementos es  $O(m)$ , entonces  $\alpha = O(1)$  y la búsqueda en la tabla de hash toma tiempo promedio constante
- Si el número  $n$  de elementos es mucho mayor a  $m$ , entonces  $\alpha$  no es constante y la búsqueda toma tiempo  $\Theta(\alpha)$
- Con más trabajo (ejercicio 11-2), se muestra que la esperanza del mayor número de elementos en un slot es  $O(\log n / \log \log n)$

En la práctica, cuando  $n$  se acerca a  $m$ , la tabla se **redimensiona automáticamente (rehashing)** a una tabla de mayor tamaño (típicamente de tamaño  $2m$ )

Rehashing toma **tiempo constante amortizado** (§17, CI2613)

## Funciones de hash

Una buena función de hash satisface **simple uniform hashing**

Si las claves son números reales en  $[0, 1)$ , uniformes e independientes,  $h(k) = \lfloor mk \rfloor$  es una buena función de hash

Un principio que usualmente produce buenas funciones es evitar que los valores de hash dependan de **“patrones”** existentes en las claves

Presentamos tres métodos de hashing:

- método de división
- método multiplicativo
- funciones de hash de Jenkins

## Método de división

Muchas funciones asumen que las claves son números naturales: ya sea porque son tales números o porque pueden mapearse a ellos

El método de división define  $h(k) = (k \bmod m)$  donde  $m$  es el número de slots

**Ejemplo:** para  $m = 12$  y  $k = 100$ :  $h(k) = 4$  ya que  $100 = 12 \times 8 + 4$

No todos los valores de  $m$  son adecuados,  $m = 2^p$  y  $m = 2^p - 1$  no son buenos valores en muchos casos

En cambio,  $m = p$  (donde  $p$  es un primo “lejano” a una potencia de 2) frecuentemente resulta en una buena función de hash

## Método multiplicativo

El método multiplicativo involucra dos operaciones. Dadas constantes  $k$  (entero positivo) y  $A \in (0, 1)$ , la función de hash multiplicativa es

$$h(k) = \lfloor m \times (kA \bmod 1) \rfloor$$

donde ‘ $x \bmod 1$ ’ denota la fracción de  $x$ ; i.e.  $x \bmod 1 = x - \lfloor x \rfloor$

**Ejemplo:** para  $m = 12$ ,  $k = 100$  y  $A = 5/7$ :

$$h(k) = \lfloor 12 \times (\frac{100 \times 5}{7} \bmod 1) \rfloor = \lfloor 12 \times (71.428 \bmod 1) \rfloor = 5$$

Una ventaja de este método es que el **valor de  $m$  no es crítico**

D. E. Knuth recomienda  $A \approx (1 - \sqrt{5})/2 = 0.6180339887\dots$

## Implementación eficiente del método multiplicativo

Escogemos  $m = 2^p$  para algún entero  $p$  y suponga que la **palabra del computador** tiene  $w$  bits

Escogemos  $A \in (0, 1)$  que pueda expresarse como  $A = s/2^w$  donde  $0 < s < 2^w$

Para calcular  $h(k)$ , multiplicamos  $k$  por  $s = A \times 2^w$  para obtener el número  $r_1 \times 2^w + r_0$  de  $2w$  bits donde  $r_0$  y  $r_1$  son de  $w$  bits cada uno

El valor  $h(k)$  lo conforman los  $p$  bits **más significativos** de  $r_0$ ; i.e.  $h(k) = r_0 \gg (w - p)$

## Implementación eficiente del método multiplicativo

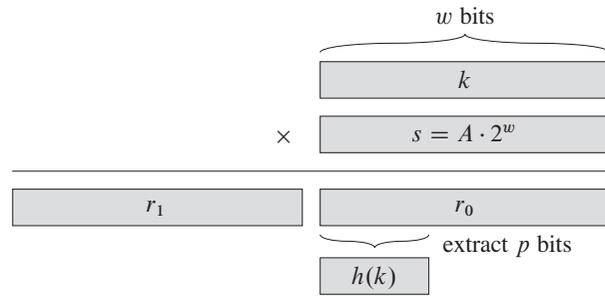


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

**Ejemplo:**  $k = 123456$ ,  $p = 14$ ,  $m = 2^{14} = 16,384$  y  $w = 32$

Escogemos  $A = 2654435769/2^{32} = 0.6180339886$  (siguiendo a Knuth)

$k \times s = 327706022297664 = 76300 \times 2^{32} + 17612864 = r_1 \times 2^{32} + r_0$

$h(k) =$  "los 14 bits más significativos de  $r_0$ " = 67

© 2018 Blai Bonet

## Funciones de hash de Jenkins

Diseñadas por Bob Jenkins para **cadenas de bytes**. Ampliamente utilizadas. Existen varias versiones

**Input:** arreglo  $key[0 \dots len - 1]$  y longitud  $len$

**Output:** valor de hash para el arreglo  $key[0 \dots len - 1]$

```
1 Jenkins-One-At-A-Time(array key)
2     int32 hash = 0
3     for i = 1 to key.length
4         hash = hash + key[i]
5         hash = hash + (hash << 10)
6         hash = hash ^ (hash >> 6)
7     hash = hash + (hash << 3)
8     hash = hash ^ (hash >> 11)
9     hash = hash + (hash << 15)
10    return hash
```

Fuente: Wikipedia

© 2018 Blai Bonet

## Direccionamiento abierto

En direccionamiento abierto todos los elementos del diccionario son guardados en la tabla sin necesidad de utilizar memoria adicional (i.e. listas enlazadas)

Cada slot de la tabla contiene un apuntador a un elemento o el apuntador **null** que indica que el slot esta vacío

Al buscar un elemento, la tabla es recorrida de **forma sistemática** hasta encontrarlo o detectar que no se encuentra en la tabla

La ventaja de direccionamiento abierto es que no utiliza apuntadores lo que se traduce en tener una mayor capacidad de almacenamiento que un esquema con encadenamiento y la misma memoria

© 2018 Blai Bonet

## Inserción con direccionamiento abierto

Para insertar un elemento  $x$  con clave  $k$  debemos **buscar un slot disponible** en donde colocar el nuevo elemento  $x$

La búsqueda se realiza con una **sonda (probe)** la cual es generada por una función de hash  $h$  de forma

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

donde  $m$  es el número de slots en la tabla

Para toda clave  $k$ , la función de hash debe cumplir con que la sonda

$$\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$$

es una **permutación de  $\{0, 1, \dots, m - 1\}$**

© 2018 Blai Bonet

## Función de hash para direccionamiento abierto

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

Esquema con  $m = 4$  slots

	$h(k, 0)$	$h(k, 1)$	$h(k, 2)$	$h(k, 3)$
$k = 73$	3	2	0	1
$k = 23$	2	1	0	3
$k = 28$	2	3	1	0
$k = 91$	0	1	2	3
...				

## Inserción con direccionamiento abierto

La idea es buscar un slot disponible a lo largo de la sonda hasta encontrarlo o hasta determinar que tal slot no existe en la tabla

```
1 Hash-Insert(hash T, pointer x)
2   i = 0
3   repeat
4     j = h(x.key, i)
5     if T[j] == null
6       T[j] = x           % éxito: el elemento x se inserta
7       return j
8     else
9       i = i + 1
10  until i == T.size
11  error "hash table overflow"
```

## Búsqueda con direccionamiento abierto

Para buscar, exploramos la tabla en el mismo orden de la inserción hasta encontrar el elemento, encontrar el apuntador `null`, o haber explorado toda la tabla

```
1 Hash-Search(hash T, key k)
2   i = 0
3   repeat
4     j = h(k, i)
5     if T[j].key == k
6       return j           % éxito: la clave k se encuentra
7     else
8       i = i + 1
9   until T[j] == null || i == T.size   % T no contiene la clave k
10  return null
```

## Eliminación con direccionamiento abierto

La eliminación es más compleja. No es correcto asignar `null` al slot  $j$  del elemento a eliminar ya que la búsqueda dejaría de ser correcta

Podemos utilizar un valor especial `deleted` y reemplazar  $T[j]$  por `deleted` al eliminar el elemento en el slot  $T[j]$

También debemos cambiar `Hash-Insert` para que inserte un elemento en un slot que contenga `null` o `deleted`. Observe que `Hash-Search` no hace falta modificarlo

Sin embargo, el desempeño ya no sólo depende del factor de carga  $\alpha$ . Por esta razón direccionamiento abierto se suele utilizar cuando **no hace falta eliminar claves**

## Análisis de direccionamiento abierto

Asumimos **uniform hashing**:  $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$  es una permutación aleatoria uniforme de las  $m!$  posibles permutaciones sobre  $\{0, 1, \dots, m - 1\}$

Uniform hashing generaliza la noción de **simple uniform hashing** sobre las sondas generadas por la función de hash

Implementar uniform hashing es difícil

Luego veremos algunas funciones de hash que aproximan esta idea

## Análisis de direccionamiento abierto

Con direccionamiento abierto cada slot de la tabla contiene a lo sumo un elemento; i.e.  $\alpha = n/m \leq 1$

Primero calculamos el número promedio de inspecciones al hacer una **búsqueda** de una clave  $k$  que **no está** en la tabla cuando  $\alpha < 1$

Defina v.a.  $X$  como el número de inspecciones a realizar al buscar  $k$ , y considere el evento

$A_i =$  "la  $i$ -ésima inspección ocurre y es sobre un slot ocupado"

Observe que el evento  $\{X \geq i\}$  es igual a  $A_1 \cap A_2 \cap \dots \cap A_{i-1}$

Acotaremos  $\mathbb{P}(X \geq i)$  acotando  $\mathbb{P}(A_1 \cap A_2 \cap \dots \cap A_{i-1})$

## Análisis de direccionamiento abierto

$$\mathbb{P}(A_1 \cap A_2 \cap \dots \cap A_{i-1}) = \mathbb{P}(A_1) \cdot \mathbb{P}(A_2|A_1) \cdot \dots \cdot \mathbb{P}(A_{i-1}|A_1, \dots, A_{i-2})$$

Como existen  $n$  elementos en la tabla y  $m$  slots:

$$\mathbb{P}(A_1) = \alpha$$

$$\mathbb{P}(A_2|A_1) = \frac{n-1}{m-1}$$

$$\mathbb{P}(A_3|A_1, A_2) = \frac{n-2}{m-2}$$

$$\mathbb{P}(A_j|A_1, \dots, A_{j-1}) = \frac{n-j+1}{m-j+1}$$

Por ejemplo, para  $\mathbb{P}(A_2|A_1)$  observe que dado  $A_1$ , existen  $m - 1$  slots y  $n - 1$  elementos restantes en la tabla. Similar para las otras probabilidades

## Análisis de direccionamiento abierto

$$\begin{aligned} \mathbb{P}(A_1 \cap A_2 \cap \dots \cap A_{i-1}) &= \frac{n}{m} \times \frac{n-1}{m-1} \times \dots \times \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1} \end{aligned}$$

Calculamos  $\mathbb{E}[X]$ :

$$\mathbb{E}[X] = \sum_{i \geq 1} \mathbb{P}(X \geq i) \leq \sum_{i \geq 1} \alpha^{i-1} = \sum_{i \geq 0} \alpha^i = \frac{1}{1-\alpha}$$

Si  $\alpha \leq$  constante, las búsquedas infructuosas toman tiempo  $O(1)$ . E.g., para  $\alpha = \frac{1}{2}$ , el número promedio de inspecciones es 2 y para  $\alpha = \frac{9}{10}$  es 10 **independientemente del número  $m$  de slots**

Al insertar se debe realizar una búsqueda. El tiempo promedio para insertar un elemento  $x$  en una tabla con  $\alpha < 1$  es  $1/(1-\alpha)$

## Análisis de direccionamiento abierto

Ahora acotamos el tiempo promedio para buscar una clave  $k$  que si está en la tabla. Suponemos que  $k$  es **cualquier clave en la tabla con igual probabilidad**

Si  $k$  es la  $(i + 1)$ -ésima clave insertada, buscarla es equivalente a hacer una búsqueda infructuosa en una tabla que solo contiene las primeras  $i$  claves. Su búsqueda toma tiempo promedio  $\frac{1}{1 - \frac{i}{m}} = \frac{m}{m-i}$

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} \sum_{i=m-n+1}^m \frac{1}{i} \\ &\leq \frac{1}{\alpha} \int_{m-n}^m \frac{dx}{x} = \frac{1}{\alpha} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

Por ejemplo, para  $\alpha = \frac{1}{2}$  se realizan 1.38 inspecciones en promedio, y para  $\alpha = \frac{9}{10}$  se realizan 2.55 inspecciones en promedio

## Resumen del análisis de direccionamiento abierto

**Factor de carga:**  $\alpha = \frac{n}{m}$  es número promedio de elementos por slot

**Búsqueda de clave que no está en la tabla:**  $O(\frac{1}{1-\alpha})$

**Búsqueda de clave que si está en la tabla:**  $O(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})$

**Inserción:**  $O(\frac{1}{1-\alpha})$

En la práctica  $\alpha$  se mantiene acotado por una constante (e.g. 0.75), y cuando se supera dicho valor se hace **rehashing**

## Funciones de hash para direccionamiento abierto

Tres métodos para construir funciones de hash para direccionamiento abierto:

- linear probing
- quadratic probing
- hashing doble

## Linear probing

Dada una función estándar de hash  $h' : U \rightarrow \{0, 1, \dots, m-1\}$ , **linear probing** define  $h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$ :

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

Linear probing es fácil de implementar pero sufre de un problema conocido como **primary clustering** donde largas secuencias de slots llenos aparecen en la tabla haciendo que las operaciones de inserción y búsqueda se hagan cada vez más lentas

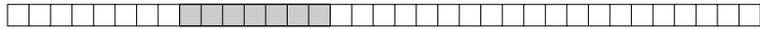
## Agrupamiento primario (primary clustering)

En linear probing,  $h(k) = (h'(k) + i) \bmod m$  define la sonda

$$\langle h'(k) \bmod m, (h'(k) + 1) \bmod m, (h'(k) + 2) \bmod m, \dots \rangle$$

Si existe un bloque contiguo de celdas ocupadas en la tabla y  $h'(k)$  "cae" dentro del bloque, al insertar la clave  $k$  el bloque "crece"

A medida que el bloque se hace más largo, las colisiones se hacen más probables, y es más probable que el bloque siga creciendo



## Quadratic probing

**Quadratic probing** también utiliza una función de hash  $h'$  auxiliar:

$$h(k, i) = (h'(k) + c_1 \times i + c_2 \times i^2) \bmod m$$

donde  $c_1$  y  $c_2$  son constantes positivas

Para poder utilizar toda la tabla (i.e., para que las sondas sean realmente permutaciones de  $\{0, 1, \dots, m-1\}$  ciertas restricciones entre los valores de  $c_1$ ,  $c_2$  y  $m$  se deben obedecer

Quadratic probing es mucho mejor que linear probing pero también sufre un problema de clustering conocido como **secondary clustering**

Observe que las sondas de linear y quadratic probing están completamente determinadas por el primer elemento, y por lo tanto **solo existen  $m$  secuencias distintas**

## Hashing doble

**Hashing doble** utiliza dos funciones de hash  $h_1$  y  $h_2$  auxiliares:

$$h(k, i) = (h_1(k) + i \times h_2(k)) \bmod m$$

El primer slot para la clave  $k$  es  $T[h_1(k)]$ , los siguientes slots se encuentran a un desplazamiento  $h_2(k)$  (módulo  $m$ ) del anterior

El valor  $h_2(k)$  debe ser **primo relativo** a  $m$  para que la secuencia recorra toda la tabla. Formas de garantizar esto son:

- $m$  es una potencia de 2 y  $h_2$  siempre retorna un número impar
- $m$  es un primo y  $h_2$  siempre retorna un número positivo menor a  $m$

En ambos casos, el número diferente de sondas es  $O(m^2)$  mejorando sustancialmente a linear y quadratic probing

## Resumen

- Una tabla de hash implementa un diccionario
- La idea es utilizar un arreglo de slots para guardar los elementos y una función de hash que mapea claves en slots
- Existen dos formas de resolver colisiones: encadenamiento y direccionamiento abierto
- En encadenamiento, las claves asociadas a un slot se mantienen en una lista doble enlazada
- En direccionamiento abierto, las claves se mantienen en la tabla y se utiliza la función de hash para obtener una sonda de búsqueda sobre la tabla
- Bajo suposiciones razonables, ambos métodos garantizan tiempo promedio constante por operación

## Ejercicios (1 de 6)

1. (11.1-1) Considere un conjunto dinámico representado por una tabla de direccionamiento directo de tamaño  $m$ . Describa un procedimiento que encuentre el elemento de mayor clave en  $S$ . ¿Cuál es la complejidad de su procedimiento en el peor caso y mejor caso?
2. (11.1-2) Un vector de bits es un arreglo de bits (de 0s y 1s). Un vector de bits de tamaño  $m$  requiere de mucho menos espacio que un arreglo de  $m$  apuntadores. Describa como utilizar un vector de bits para representar un conjunto de claves que no contienen datos satélites. Las operaciones de diccionario deben tomar tiempo  $\Theta(1)$
3. (11.1.-3) Discuta como implementar una tabla de direccionamiento directo para guardar claves que no tienen porque ser distintas y que también tienen datos satélites asociados. Las operaciones de diccionario deben correr en tiempo  $\Theta(1)$  y no olvide que la operación de eliminar una clave toma como argumento el objeto a ser eliminado y no una clave

## Ejercicios (2 de 6)

4. Mostrar que bajo simple uniform hashing resolución de colisiones por encadenamiento  $\mathbb{E}[X_{ij}] = 1/m$  y que el número promedio de elementos por slot es  $\alpha$
5. Argumentar la independencia de las v.a.  $Z_i$  y  $X_{ij}$  usada en el análisis de resolución de colisiones por encadenamiento
6. Verificar la implementación eficiente de hashing multiplicativo
7. Mostrar que si las claves se distribuyen de forma uniforme e independiente en  $[0, 1)$ , entonces  $(k) = \lfloor mk \rfloor$  es una buena función de hash
8. Probar que  $\frac{n-1}{m-1} \leq \frac{n}{m}$  cuando  $n \leq m$

## Ejercicios (3 de 6)

9. (11.1.4)\* Deseamos implementar un diccionario utilizando direccionamiento directo sobre un arreglo  $T$  de gran dimensión. Debido a su tamaño no es práctico inicializar el arreglo  $T$ . Describa como implementar dicho diccionario donde cada objeto requiera espacio  $\Theta(1)$ , cada operación de diccionario tome tiempo  $\Theta(1)$ , y la inicialización de la estructura de datos se realice en tiempo  $\Theta(1)$

**(Ayuda:** utilice dos arreglos de dimensión igual a  $T$ . El primer arreglo se maneja como un stack de dimensión igual a los elementos guardados en  $T$  y que se utiliza para validar las entradas en  $T$  que corresponden con elementos en  $T$ , el segundo arreglo guarda los elementos de  $T$ .)

10. (11.2-1) Considere una función de hash  $h$  para indexar  $n$  claves distintas en una tabla de tamaño  $m$ . Asumiendo simple uniform hashing, calcule el número promedio de colisiones; i.e. la cardinalidad esperada del conjunto  $\{k \neq \ell : k \neq \ell \wedge h(k) = h(\ell)\}$

## Ejercicios (4 de 6)

11. (11.2-2) Muestre que pasa cuando inserta las claves 5, 28, 19, 15, 20, 33, 12, 17, 10 en una tabla de hash con 9 slots y colisiones resueltas por encadenamiento. Utilice la función de hash  $h(k) = k \bmod 9$
12. (11.2-3) Implemente una tabla de hash con colisiones resueltas por encadenamiento en donde las listas para cada slot están ordenadas de forma no decreciente por las claves. ¿Cómo se ven afectados los desempeños de las operaciones? ¿Vale la pena?
13. (11.2-5) Suponga que guardamos un conjunto  $K$  de  $n$  claves en una tabla de tamaño  $m$ . Muestre que si las claves provienen de un universo  $U$  con  $|U| > nm$ , entonces existe un subconjunto  $K' \subseteq U$  de  $n$  claves que van todas al mismo slot. Por lo tanto, el peor caso para las operaciones de búsqueda sobre el hash es  $\Theta(n)$  en lugar de  $\Theta(1)$ , aún cuando la función de hash satisfaga simple uniform hashing

## Ejercicios (5 de 6)

14. (11.2-6) Suponga que tenemos  $n$  claves guardadas en un tabla de hash de tamaño  $m$  con colisiones resueltas por encadenamiento. También suponga que conocemos la longitud de cada cadena y que  $L$  es la longitud máxima. Describa un procedimiento que selecciona una clave uniformemente al azar de la tabla en tiempo esperado  $O(L(1 + 1/\alpha))$
15. (11.3-1) Suponga que debemos hacer una búsqueda sobre una lista enlazada con  $n$  elementos, donde cada elemento contiene una clave  $k$  y también el valor de hash  $h(k)$ . Las claves son cadenas largas de caracteres (strings). ¿Cómo pueden utilizarse los valores de hash para hacer la búsqueda de una clave más eficiente?
16. (11.3-2) Suponga que insertamos cadenas de  $r$  bytes en una tabla con  $m$  slots al considerar cada cadena como un número en base 128 y al utilizar el método de división. Asumiendo que el número  $m$  de slots cabe en una palabra de 32 bits (i.e.  $m < 2^{32}$ ), explique como podemos implementar el método de división para computar el valor hash de una cadena solo utilizando espacio adicional constante

## Ejercicios (6 de 6)

17. (11.3-4) Considere una tabla de hash de tamaño  $m = 1000$  con la función de hash  $h(k) = \lfloor m(kA \bmod 1) \rfloor$  con  $A = (\sqrt{5} - 1)/2$ . Calcular los valores de  $h(k)$  para las claves 61, 62, 63, 64 y 65
18. (11.4-1) Considere una tabla de hash donde las colisiones son resueltas con direccionamiento abierto. Considere la inserción de las claves 10, 22, 31, 4, 15, 28, 17, 88, 59 en una tabla con  $m = 11$  slots y con la función auxiliar de hash  $h'(k) = k$ . Mostrar el resultado de la inserción al utilizar linear probing, quadratic probing con  $c_1 = 1$  y  $c_2 = 3$ , y hashing doble con  $h_1(k) = k$  y  $h_2(k) = 1 + (k \bmod (m - 1))$
19. (11.4-2) Escriba el procedimiento **Hash-Delete** para direccionamiento abierto y modifique **Hash-Insert** para que maneje el valor **deleted**
20. (11.4-3) Considere una tabla de hash donde las colisiones son resueltas con direccionamiento abierto bajo uniform hashing. Acote el número promedio de sondas en búsquedas exitosas e infructuosas cuando el factor de carga es  $\frac{3}{4}$  y  $\frac{7}{8}$