CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Hashing cuco y filtro de Bloom

Objetivos

- Hashing cuco (cuckoo hashing)
- Filtro de Bloom

Hashing cuco

Inventado por Ramus Pagh y Flemming F. Rodl en 2001

Tratamos de mantener una **tabla de hash sin colisiones** en donde todas las operaciones tomen tiempo constante

Asociamos a cada clave x **dos posiciones** posibles, determinadas por dos funciones de hash $h_1, h_2 : U \to \{0, \ldots, m-1\}$

Alternativamente podemos pensar que tenemos dos tablas de hash T_1 y T_2 tal que cada elemento se almacena en la primera o segunda tabla (pero no en ambas)

Hashing cuco

Al insertar una nueva clave x, la clave se coloca en una de sus dos posiciones, **desplazando** cualquier otra clave que pueda estar ahí

La clave desplazada es insertada en su posición alternativa, pudiendo desplazar otra clave

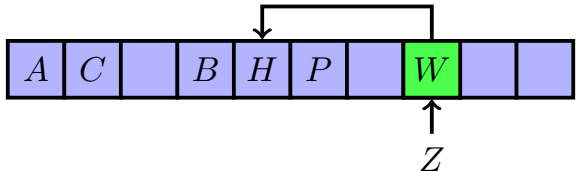
El procedimiento se repite hasta que todas las claves se estabilizan o hasta caer en un lazo infinito (en la práctica hasta un número máximo de iteraciones) cuando la tabla es redimensionada y/o las funciones de hash son cambiadas

Nota: su nombre lo debe al **pájaro cuco** que cuando necesita espacio dentro del nido (de otro pájaro), reemplaza el contenido del nido con su huevo

Hashing cuco

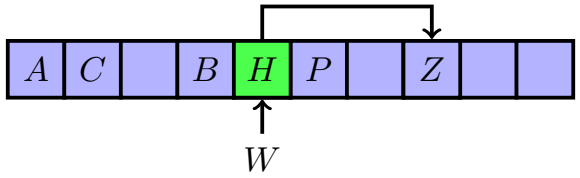
```
1  Insert(array T, pointer x)
2      pos = h1(x.key)
3      for i = 1 to N do
4          if T[pos] == null              % éxito: encontramos un lugar para x
5              T[pos] = x
6              return
7          else
8              Intercambiar x con T[pos]              % hacemos espacio para x
9              if pos == h1(x.key)
10                 pos = h2(x.key)
11             else
12                 pos = h1(x.key)
13
14     % después de N iteraciones no hemos podido insertar x
15     Rehash(T)
16     Insert(T, x)
17
18 Search(array T, key k)
19     if T[h1(k)] != null && T[h1(k)].key == k then return T[h1(k)]
20     if T[h2(k)] != null && T[h2(k)].key == k then return T[h2(k)]
21     return null
```


Ejemplo de inserción en un hash cuco



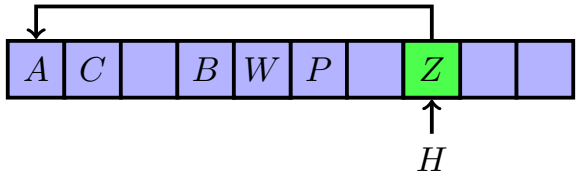
clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Ejemplo de inserción en un hash cuco



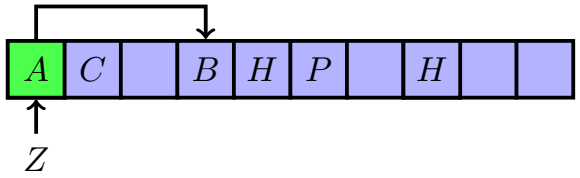
clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Ejemplo de inserción en un hash cuco



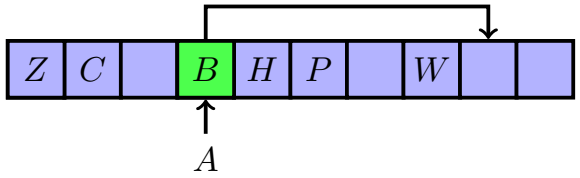
clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Ejemplo de inserción en un hash cuco



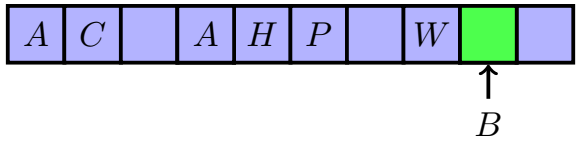
clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Ejemplo de inserción en un hash cuco



clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Ejemplo de inserción en un hash cuco



clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Ejemplo de inserción en un hash cuco

clave	$h_1(\cdot)$	$h_2(\cdot)$
A	0	3
B	3	8
C	1	2
H	4	7
P	5	8
W	7	4
Z	7	0

Análisis de hash cuco (resultados)

Considere un hash con m slots y dos funciones de hash h_1 y h_2

Asumimos:

- cota superior $n \leq m$ de elementos que pueden entrar al diccionario
- h_1 y h_2 producen índices de forma equiprobable e independiente

Resultados:

- por la cota n, el factor de carga $\alpha \leq 1$
- las operaciones de búsqueda y eliminación toman tiempo constante
- las inserciones tomán tiempo esperado constante (si rehashing es necesario, tenemos **tiempo amortizado constante** por inserción)
- si redimensionamiento es necesario, tenemos **tiempo amortizado constante** (la tabla debe ser redimensionada por factor constante)

Variaciones sobre hashing cuco

- Usar mas de dos funciones de hash
- Usar un "cache" de espacio constante para guardar colisiones de forma temporal (stash)
- Usar buckets de tamaño constante > 1

Filtro de Bloom

Filtro de Bloom

Inventado por en 1970 por Burton H. Bloom

Un filtro de Bloom es una estructura de datos que implementa de forma eficiente las operaciones: insertar y buscar clave

Se puede pensar como una **ED probabilística** tipo Monte Carlo donde las búsquedas pueden cometer **errores**

En el filtro de Bloom **no existen datos satélites** asociados a las claves. La búsqueda solo retorna un booleano que indica cuando la clave existe o no dentro del diccionario

Estructura del filtro de Bloom

El filtro se implementa con un arreglo de bits de dimensión m

Al igual que en hashing, para acceder al arreglo se utilizan funciones de hash. Sin embargo, el filtro de Bloom utiliza k **funciones de hash** en lugar de una sola función:

- Al inicio, todos los bits en el filtro B están "apagados" (i.e. son 0)
- Para insertar/buscar una clave x, se utilizan las funciones de hash para calcular k índices $i_1, \ldots, i_k$ donde $i_j = h_j(x)$
- La inserción se implementa realizando las asignaciones $B[i_j] = 1$ para cada índice i_j
- La búsqueda retorna **true** ssi **todas** las entradas $B[i_j]$ son 1

Filtro de Bloom

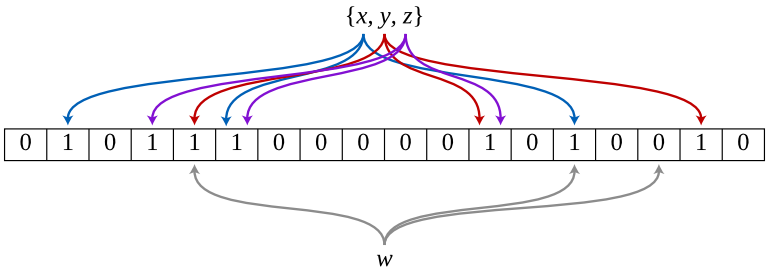


Imagen de https://en.wikipedia.org/wiki/Bloom_filter

Filtro de Bloom con $m = 18$ bits y $k = 3$ funciones de hash. Las claves x, y, z han sido insertadas. La clave w no está en el filtro puesto que $B[h_j(w)] = 0$ para un $j \in \{1, 2, 3\}$

Errores en la búsqueda

Por diseño solo existe un posible **tipo de error** al realizar la búsqueda de una clave x en el filtro de Bloom:

- Si x está en el filtro, la búsqueda de x retorna **true**
- Si x no está en el filtro, la búsqueda puede retornar **true** o **false**

El error en que puede incurrir la búsqueda se llama un **falso positivo**

Bajo suposiciones razonables, veremos cómo escoger m y k para **garantizar cierta probabilidad de error**

Uso típico del filtro de Bloom

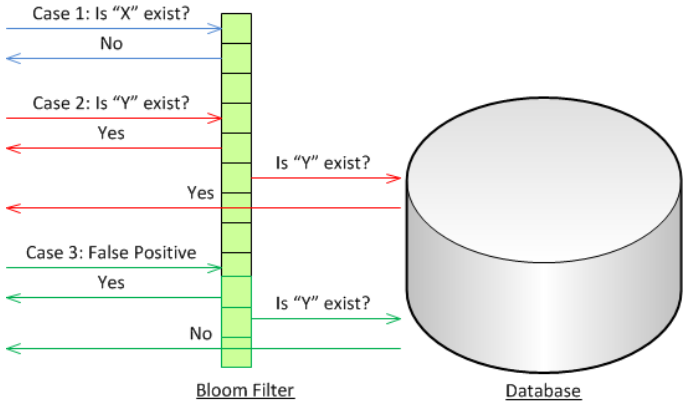


Imagen de http://crzyjcky.com/2013/01/03/the-magical-bloom-filter

Ejemplos

- Akamai (Content Delivery Network o CDN) utiliza filtros de Bloom para evitar insertar en el cache los "one-hit-wonders" que representan 3/4 de los objetos. Un "one-hit-wonder" es un objeto que se solicita una sola vez y por lo tanto no hace falta insertarlo en el cache
- Google BigTable, Apache HBase y Apache Cassandra utilizan filtros de Bloom para reducir las búsquedas en disco de filas/columnas inexistentes
- Bitcoin utiliza los filtros de Bloom durante la sincronización de la cartera
- SPIN "model checker" utiliza filtros de Bloom para monitoriear el espacio de estado al verificar grandes sistemas
- ...

Compromiso espacio vs. tiempo

Aunque puedan existir falsos positivos, un filtro de Bloom ofrece ventajas sobresalientes sobre otras ED para conjuntos dinámicos

Un filtro diseñado para tener una **probabilidad de error del 1%**, utilizando un valor óptimo de k, solo requiere de aproximadamente **9,6 bits por elemento almacenado** (sin importar la naturaleza y tamaño de los elementos)

Otras ED almacenan las claves y en algunos casos mantienen apuntadores por cada elemento almacenado

Los filtros de Bloom tienen la propiedad especial que el tiempo requerido para agregar y buscar elementos es constante, $O(k)$, que es **independiente de número de elementos almacenados**

Análisis del filtro de Bloom

Asumimos que para toda clave x, cada función de hash h_j selecciona un bit de los m bits en el filtro con igual probabilidad y de forma independiente de las otras funciones

Al insertar una clave x, la probabilidad que un bit específico **no sea seleccionado** por la j-ésima función de hash es $1 - \frac{1}{m}$, y que no sea seleccionado por ninguna función es $\left(1 - \frac{1}{m}\right)^k$

Después de n **inserciones**, el bit es 0 con probabilidad $\left(1 - \frac{1}{m}\right)^{kn}$ y es 1 con probabilidad

$$p_{m,n} \;=\; 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Análisis del filtro de Bloom

Considere la búsqueda de una clave x que **no está en el filtro**

La probabilidad que las k funciones de hash accesen bits con valor igual a 1 es

$$p_{m,n}^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

La probabilidad de error:

- **decrece** con el número m de bits en el filtro
- **decrece** con el número k de funciones de hash
- **incrementa** con el número n de elementos en el filtro

Probabilidad de falso positivo

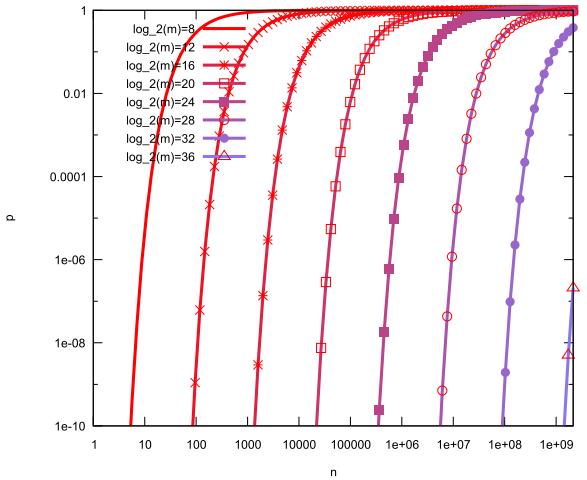


Imagen de `https://en.wikipedia.org/wiki/Bloom_filter`

El número k de funciones de hash es $k = (m/n) \ln 2$

Probabilidad de falso positivo

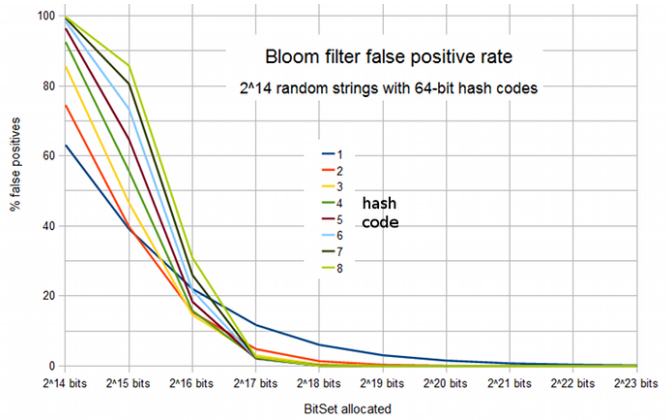


Imagen de `http://www.javamex.com/tutorials/collections/bloom_filter.shtml`

Tamaño del filtro

Para valores de m (tamaño del filtro) y n (elementos en el filtro) dados, el **número óptimo** k de funciones de hash que **minimizan la probabilidad de error** es $k^*_{m,n} = \frac{m}{n} \ln 2$

El tamaño requerido m (número de bits) para garantizar una probabilidad de error p cuando n elementos son insertados, utilizando $k^*_{m,n}$ funciones de hash, es $m = -\frac{n \ln p}{(\ln 2)^2}$

Para p y n fijos, el tamaño requerido es proporcional a n. El resultado es un **número constante de bits por elemento almacenado**

Resumen

- Hashing cuco: una forma alternativa de resolver colisiones tipo direccionamiento abierto
- Filtro de Bloom: ED probabilística tipo Monte Carlo para implementar un diccionario que sólo soporta inserciones y búsquedas
- Ambas ED tienen garantías de tiempo constante en el peor caso sobre las búsquedas y sobre la eficiencia del uso de espacio, pero a cierto costo

Ejercicios

1. Escriba el pseudocódigo para la eliminación de elementos en hashing cuco
2. Implemente los esquemas de hashing con resolución de colisiones por encadenamiento, direccionamiento abierto con linear probing, y hashing cuco, y realice una evaluación experimental
3. Escriba el pseudocódigo para las operaciones de inserción y búsqueda en el filtro de Bloom
4. Implemente un filtro de Bloom