Algoritmos y estructuras de datos

www.u-tad.com

Grado en Ingeniería en Desarrollo de Contenidos Digitales

Introducción I

- La operación más importante de una lista (o base de datos) es la búsqueda
  - En general, hacemos muchísimas más búsquedas que inserciones/eliminaciones
    - Pensemos en Internet (una inmensa base de datos no organizada) y sus “buscadores”
  - La razón de ser de las bases de datos es para que mucha gente pueda buscar en ellas
  - Las inserciones/eliminaciones masivas se suelen hacer al principio de la vida útil de la base de datos
    - Después la base de datos se vuelve “estable” y las búsquedas superan a las inserciones/eliminaciones en un factor de 100 o incluso mayor
- Por lo tanto es muy importante optimizar todo lo que podamos el tiempo de las búsquedas

Introducción II

- Hasta ahora, las mejores búsquedas tienen una complejidad de O(logn) si se realizan sobre listas contiguas ordenadas
  - Gracias a la búsqueda binaria
- ¿Podríamos mejorar aún más este rendimiento?
  - A simple vista parece difícil, ya que tendríamos que conseguir búsquedas en O(1)
- Las tablas hash son un tipo especial de listas contiguas que nos permiten hacer búsquedas en O(1) o muy próximo a este valor
  - Por lo tanto, todas las bases de datos modernas se implementan con tablas hash
  - A cambio de conseguir O(1), las tablas hash pueden desperdiciar gran cantidad de memoria

Definición I

- Una tabla hash es una lista con las siguientes características:
  - Generalmente se almacena de forma contigua
  - No ordenada
  - Cada elemento es un par: (clave, valor)
    - La clave es una información que identifica al elemento de forma unívoca, ningún otro elemento puede tener la misma clave
      - Ej: DNI de una persona
    - El valor puede ser todo lo complicado que queramos y tener a su vez muchos campos
      - Ej: el resto de información de una persona: nombre, apellidos, fecha de nacimiento, etc.

Definición II

- Otros nombres de una tabla hash:
  - Tabla (a secas)
  - Diccionarios
    - La clave es la palabra y el valor es su definición
  - Aplicaciones
  - Maps
  - Memorias asociativas

Definición III

- Vectores vs tablas hash
  - En los vectores (listas contiguas), se accede a los elementos (“direccionamiento”) mediante su posición en el vector
  - En las tablas hash, el direccionamiento de sus elementos es mediante su clave
    - No sabemos ni nos interesa en qué posición está almacenada una persona…
    - En su lugar, accedemos a la persona mediante su clave, mediante su DNI
      - Por lo tanto, decimos: “Dame la persona cuyo DNI es XXX” en vez de “Dame la persona que está en la posición YYY”
  - Por lo tanto, en los vectores el direccionamiento (la clave) es un entero (la posición que ocupa) y el valor es lo que se guarda en esa posición, mientras que en las tablas hash la clave es cualquier tipo de dato (ej: DNI, un string, etc.) y no tiene nada que ver con la posición que se ocupa
    - Hay algunos lenguajes de programación en los cuales existen tablas hash de forma nativa (ej: PERL). No es el caso de C.

Operaciones I

- Las operaciones de una tabla hash (desde el punto de vista del usuario de la tabla, no del programador) son las siguientes:
  - Ver si un elemento existe
    - El usuario de la tabla proporciona la clave del elemento
    - La tabla busca internamente un elemento con esa clave.
    - Conseguiremos que esta búsqueda sea O(1) o muy próximo a este valor
    - Si lo encuentra, devuelve verdadero. Falso en caso contrario
  - Obtener un elemento
    - Es la operación más importante
    - Precondición: el elemento existe en la tabla
    - Se da la clave del elemento, la tabla lo busca internamente y, una vez encontrado, nos devuelve su valor asociado
    - No existe la operación de obtener un elemento por posición
      - Ni, en general, operaciones para las que sea necesario el número de posición. Por lo tanto no existe recuperar el primero, o el último, o el siguiente, o el anterior…

Operaciones II

- Insertar un nuevo elemento
  - Necesitamos dar la clave y el valor del nuevo elemento
  - Precondición: ningún elemento con esa misma clave existe ya en la tabla
  - La tabla guarda ambos valores en una posición que no conocemos ni nos interesa
- Eliminar un elemento existente
  - Se da la clave del elemento
  - Precondición: el elemento existe en la tabla
  - La tabla busca internamente el elemento a partir de su clave
  - Una vez encontrado, la tabla elimina el elemento entero (clave y valor)

Operaciones III

- Modificar un elemento existente
  - Se da la clave del elemento y su nuevo valor
  - Precondición: el elemento existe en la tabla
  - La tabla busca internamente el elemento a partir de su clave
  - Una vez encontrado, la tabla modifica el valor del elemento (pero no su clave) a su nuevo valor
- Otras operaciones secundarias:
  - Comprobar si está llena o vacía
  - Obtener el número de elementos
  - Etc…

Implementación ideal I

- Si el rango de posibles claves fuera de 1 hasta n podríamos utilizar un vector de capacidad n
  - La posición en donde se guardara cada elemento sería simplemente su clave
  - La búsqueda sería O(1) siempre
  - En cada posición del vector hay que añadir un campo (un bit) que nos permita saber si la posición está ocupada o no (si no está ocupada, tiene basura)
  - El problema es que se desperdicia mucha memoria pues no siempre vamos a guardar en la tabla hash todos los elementos con todas sus posibles claves

Implementación ideal II

- Ejemplo:
  - Una empresa necesita una tabla hash para guardar sus 500 empleados
  - La clave de cada empleado es su DNI, ya que es una persona y cada persona se identifica unívocamente por su DNI
  - El valor de cada empleado es su nombre, apellidos, sueldo, fecha de nacimiento, número de seguridad social, etc.
  - El rango de posibles DNIs que existen es 100 millones
  - Por lo tanto necesitaríamos una tabla hash de capacidad para 100 millones de objetos de la clase Empleado más 100 millones de bits de ocupado.
  - Si un empleado tiene como DNI 13032823, lo guardaríamos en esa misma posición de la tabla y activamos el bit de ocupado
  - Cuando quisiéramos obtener al empleado con DNI 13032823, iríamos a esa posición de la tabla (O(1) porque es una lista contigua) y recuperaríamos el objeto entero guardado allí, siempre que el bit de esa posición indique que esa posición está ocupada
  - Para borrar al empleado con DNI 13032823, iríamos a esa posición y pondríamos el bit de dicha posición a falso (para indicar que esa posición está vacía y sólo tiene basura)
  - Desperdiciamos 100.000.000 – 500 = 99.999.500 posiciones de memoria

Hashing I

- Para intentar desperdiciar menos memoria, hacemos “hashing”
  - Consiste en crear lo que se llama una “función hash”
- En vez de tener un vector con una capacidad que cubra todo el rango posible de claves, disminuímos esta capacidad
  - Ya no podemos mapear directamente la clave con la posición en la tabla, porque no hay posiciones suficientes para todas las posibles claves
  - Por lo tanto, necesitamos crear una función (la “función hash”) que asigne a cada clave su posición en la tabla
  - Una función hash muy típica es utilizar el resto de la división entera de la clave entre la capacidad de la tabla, para una tabla cuyas posiciones van desde 0 hasta capacidad-1
    - Si la clave no es numérica (ej: un texto), se pueden utilizar otro tipo de funciones hash que transformen dicha clave no numérica en una posición dentro de la tabla

Hashing II

- Ejemplo con empleados (actividad 6.1):
  - Recordemos que la empresa tiene aproximadamente 500 empleados
  - En vez de tener un vector con 100 millones de posiciones, tenemos un vector con 10.000 posiciones de capacidad
  - La función hash nos devolvería el resto de dividir el DNI entre 10.000
    - Ése número sería la posición en donde guardar el empleado con ese DNI
  - Si queremos insertar un empleado con DNI 13.032.823, lo guardamos en la posición 2.823 de la tabla (porque 13.032.823%10.000 = 2.823), poniendo a verdadero el bit “ocupado” de esa posición
  - Para eliminar o buscar, accederíamos también a la posición 2.823
  - Problema: contratamos a un nuevo empleado cuyo DNI es 78.502.823… ¡le correspondería la misma posición de la tabla, pero ésta ya está ocupada! Se ha producido una “colisión”

Hashing III

- Denominaremos “hashing” al proceso mediante el cual transformamos una clave en una posición
- A la función hash le exigiremos que:
  - Sea una operación rápidamente calculable, es decir con complejidad temporal O(1)
  - Distribuya de manera bastante uniforme los elementos sobre la tabla, para disminuir la probabilidad de colisiones
    - Si la función hash llevara a la misma posición muchas claves diferentes, se incrementa la posibilidad de que dos elementos distintos se intenten guardar en la misma posición…
      - … es decir, se incrementa la probabilidad de colisión
- Obviamente, si el tamaño de la tabla es menor que el número posible de claves, varias claves distintas tendrán asociada la misma posición (puede producirse una “colisión”)

Colisiones I

- Colisión:
  - Cuando, al intentar meter un nuevo elemento con una determinada clave, la función hash de esa nueva clave nos da una posición ya ocupada por otro elemento que existía previamente en la tabla
  - Es decir: a dos elementos distintos le corresponde la misma posición en la tabla

Colisiones II

- Probabilidad de colisión:
  - Si la tabla hash tiene tanta capacidad como posibles claves, nunca se producen colisiones (“implementación ideal”)
  - A medida que vamos disminuyendo la capacidad de la tabla, aumenta la probabilidad de colisiones, pues a más cantidad de claves distintas les toca la misma posición
  - Una mala función hash (que no distribuya uniformemente) también puede aumentar la probabilidad de colisiones
    - Ej: coger los primeros dígitos del DNI en vez de los últimos, sabiendo que los DNI de las personas de una misma provincia (ej: la provincia en la que reside la empresa) empiezan por los mismos dígitos.

Colisiones III

- ¿Cómo resolvemos las colisiones? El manejador de colisiones será el mecanismo encargado de hacerlo
- Dos tipos de manejador de colisiones:
  - Dispersión cerrada. También llamado direccionamiento abierto
  - Dispersión abierta. También llamado encadenamiento separado

Dispersión cerrada I

- Dispersión cerrada:
  - Si hay colisión, se busca un lugar de la tabla no ocupado y ahí se inserta el nuevo elemento
  - Se denomina:
    - Dispersión cerrada porque todos los elementos se almacenan dentro de la tabla
    - Direccionamiento abierto porque, a pesar de que asociamos una posición a cada clave, el elemento puede acabar en otra posición
  - Tendremos diversas variantes en función de cómo busquemos otra posición libre

Dispersión cerrada II

- El método más simple de la dispersión cerrada se denomina “exploración lineal”
  - Existen otros métodos más avanzados como “exploración cuadrática” o “desplazamiento cociente” (no los vemos)
- La exploración lineal consiste en mirar en la siguiente posición hasta que encontremos una libre
  - La siguiente a la última es la primera
  - Así, para buscar un elemento iremos a la posición que le toca, y hasta que le encontremos o la celda esté vacía (o hasta que hayamos recorrido toda la tabla si está llena), iremos a la siguiente

Dispersión cerrada III

- Exploración lineal (cont.):
  - Insertar: basta con encontrar un hueco a partir de la posición que le toca
  - Eliminar:
    - Ya no basta con marcar la casilla como vacía, como cuando no teníamos colisiones
    - Necesitamos también un bit que nos indique si la casilla, aunque esté vacía, ha sido previamente borrada
      - Ahora vemos ejemplos que ilustren esto
  - Buscar:
    - Iremos a la posición que, según la función hash, nos toca. Si la clave que buscamos no está ahí, iremos avanzando al igual que para insertar… pero en vez de parar en la primera casilla vacía, pararemos en la primera casilla vacía no borrada. Es decir, las casillas borradas y vacías nos las saltaremos y seguiremos buscando

Dispersión cerrada IV

- Ejemplo de exploración lineal:
  - Tenemos una función hash que calcula el módulo de la clave entre la capacidad de la tabla
  - Tenemos una tabla de capacidad 4 inicialmente vacía
  - Insertamos un elemento con clave 40
    - Su posición “ideal” (la que nos da la función hash) es 0, porque 40%4=0
    - Como la posición 0 no está ocupada (miramos el flag “ocupado”), metemos ahí el elemento. Ponemos el bit de ocupado a verdadero
  - Insertamos un elemento con clave 104
    - Su posición ideal es 0, porque 104%4=0
    - Como la posición 0 está ocupada (lo vemos en el bit “ocupado”), miramos la siguiente (la posición 1)
    - Como la posición 1 no está ocupada, lo metemos ahí. Ponemos el bit de ocupado a verdadero.

B?	Oc?	Clave	Valor
No	No	Basura	Basura
No	No	Basura	Basura
No	No	Basura	Basura
No	No	Basura	Basura

Tabla hash de capacidad 4 inicialmente vacía

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	No	Basura	Basura
No	No	Basura	Basura
No	No	Basura	Basura

Insertamos elemento con clave 40

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
No	No	Basura	Basura
No	No	Basura	Basura

Insertamos elemento con clave 104

Dispersión cerrada V

- Ejemplo (cont.):
  - ¿Qué ocurre si ahora quiero buscar el 80, ya sea para ver si está en la tabla, o para devolver su valor, o para borrarlo?
    - Su posición ideal es la 0, porque 80%4=0
    - La posición 0 está ocupada, y la clave guardada ahí no es 80, por lo tanto el elemento buscado no está en la posición 0 y tenemos que mirar en la siguiente (la posición 1)
    - En la posición 1 el elemento buscado tampoco está. Por lo tanto miramos en la posición 2.
    - La posición 2 está vacía y además nunca ha sido usada (si no, su bit de borrado estaría a verdadero). Podemos parar aquí y concluir que el elemento cuya clave es 80 no está en la lista.
      - Así nos evitamos tener que mirar todas las posiciones de la tabla

Dispersión cerrada VI

- Ejemplo (cont.): seguimos haciendo operaciones sobre la tabla
  - Insertamos un elemento con clave 121
    - Su posición ideal es 1, porque 121%4=1
    - Como la posición 1 está ocupada, miramos la siguiente (la posición 2)
    - Como la posición 2 no está ocupada, lo metemos ahí. Ponemos el bit de ocupado a verdadero
  - Insertamos un elemento con clave 20
    - Su posición ideal es 0, porque 20%4=0
    - Como la posición 0 está ocupada, miramos la siguiente (la posición 1)
    - Como la posición 1 está ocupada, miramos la siguiente (la posición 2)
    - Como la posición 2 está ocupada, miramos la siguiente (la posición 3)
    - Como la posición 3 no está ocupada, lo metemos ahí. Ponemos el bit de ocupado a verdadero
  - Borramos elemento con clave 121
    - Su posición ideal es 1, porque 121%4=1
    - Como la posición 1 está ocupada (para ello miramos el flag "ocupado"), entonces leemos la clave que hay en la posición 1
    - Como la clave guardada en la posición 1 no es 121 (es 104), entonces el elemento cuya clave es 121 no está en la posición 1. Por lo tanto, miramos la siguiente (la posición 2)
    - Como la posición 2 está ocupada, entonces leemos la clave que hay en la posición 2
    - La clave guardada en la posición 2 es 121, por lo tanto el elemento que queremos borrar está guardado en la posición 2
    - Borramos el elemento en la posición 2, poniendo el bit de ocupado a falso, y el bit de borrado a verdadero
    - Los campos de clave y valor de la posición 2 se quedan con basura

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
No	Sí	121	zzz
No	No	Basura	Basura

Insertamos elemento con clave 121

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
No	Sí	121	zzz
No	Sí	20	aaa

Insertamos elemento con clave 20

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
Sí	No	121 (basura)	zzz (basura)
No	Sí	20	aaa

Borramos elemento con clave 121

Dispersión cerrada VII

- Ejemplo (cont.):
  - ¿Qué ocurre si ahora quiero buscar el 20, ya sea para ver si está en la tabla, o para devolver su valor, o para borrarlo?
    - Su posición ideal es la 0, porque 20%4=0
    - Como en la 0 no está (porque está ocupada y además la clave guardada ahí no es 20), entonces miramos en la siguiente posición (la 1)
    - En la posición 1 tampoco está, por lo tanto miramos en la 2
    - La posición 2 está vacía… ¿nos detenemos aquí y creemos que el 20 no está? No nos deberíamos detener ante una casilla vacía que ha sido borrada, deberíamos seguir buscando y por lo tanto deberíamos pasar a la posición 3
      - Para eso sirve el bit de borrado
    - En la posición 3 encontramos el elemento buscado

Dispersión cerrada VIII

- Ejemplo (cont.):
  - ¿Qué ocurre si ahora quiero insertar el elemento cuya clave es 83?
    - Le toca idealmente la posición 3 porque 83%4=3. Pero la posición 3 está ocupada, por lo tanto miramos la siguiente. La siguiente a la posición 3 es la posición 0.
    - La posición 0 también está ocupada. Por lo tanto miramos la siguiente.
    - La posición 1 también está ocupada. Por lo tanto miramos la siguiente.
    - La posición 2 está libre. Por lo tanto lo metemos ahí y ponemos el bit de ocupado a verdadero
      - El bit de borrado no hace falta que lo toquemos, pues cuando la casilla está ocupada, el bit de borrado no se usa para nada
- Practica ahora la exploración lineal con la actividad 6.2

Dispersión cerrada IX

- Reestructuración de una tabla hash:
  - A medida que pasa el tiempo, la tabla hash va llenándose de celdas cuyo bit de borrado está activo
    - Cada vez la tabla hash es más “caótica”, más desordenada
  - Cuando hay muchas celdas borradas, la complejidad temporal de las búsquedas se va alejando de O(1) y acercándose a O(capacidad), siendo “capacidad” la capacidad de la tabla
    - … ya que recordemos que la búsqueda no para ante las celdas borradas
  - Por lo tanto, es necesario “reestructurar” la tabla hash cada cierto tiempo (ej: cada día por la noche, o los fines de semana…)
    - Consiste en “reagrupar” los datos de modo que queden lo más cerca posible de su posición “ideal”, según el método de exploración que tenga la tabla
      - Es como una “defragmentación” de un disco duro
    - Después de una reestructuración, no existirán celdas borradas

Dispersión cerrada X

- Algoritmos de reestructuración:
  - Sin gastar memoria adicional:
    - Algoritmo complicado: “subir” cada elemento que no esté en su sitio ideal a la posición más cercana a dicho sitio ideal. Similar al algoritmo de “defragmentación” de un disco duro
  - Gastando memoria adicional:
    - Se crea una nueva tabla hash vacía de igual capacidad que la tabla antigua
    - Se va recorriendo secuencialmente la tabla hash antigua. Por cada celda en la que haya un elemento en la tabla antigua, se inserta dicho elemento en la tabla nueva en la posición en que le toque en la tabla nueva, llamando al método de inserción de la nueva tabla.
      - Obviamente, es muy posible que al elemento no le toque la misma posición que tenía en la tabla antigua
    - Una vez insertados todos los elementos de la tabla antigua en la tabla nueva, se copia tal cual la nueva en la antigua, y se libera la memoria de la nueva
    - Complejidad espacial es O(capacidad)
    - Complejidad temporal:
      - O(capacidad) en el mejor caso (es “capacidad” y no “n” porque tenemos que recorrer toda la tabla para examinar si cada posición está ocupada o no). Es cuando todas las inserciones en la tabla nueva son O(1)
      - O(capacidad^2) en el peor caso, cuando todas las inserciones en la tabla nueva son O(capacidad).

Dispersión abierta I

- En cada posición tenemos una lista (ordenada o no) de elementos
  - Por lo tanto, todos los elementos van a su posición “ideal”… se van metiendo en la lista que hay en dicha posición
- Denominación:
  - Se denomina dispersión abierta porque los elementos pueden estar fuera de la tabla
  - También se llama “Encadenamiento separado” porque la lista suele ser enlazada

Dispersión abierta II

- Características:
  - El número de elementos puede ser mayor que el tamaño de la tabla
  - Coste de memoria adicional para la lista que hay en cada posición de la tabla
    - La lista además va creciendo (vamos creando nodos) a medida que vamos insertando elementos en esa posición de la tabla
  - La eliminación no da lugar a pérdida de rendimiento en la búsqueda, como sí pasaba en dispersión cerrada
    - Ahora ya no necesitamos bit de borrado
  - En vez de listas se podría tener otras estructuras (ej: ¡tablas hash!)
  - Se podría utilizar dos funciones de dispersión diferentes e insertar en la posición que tenga una lista más corta
    - Implicará también buscar en esas dos posiciones cada vez que busquemos un elemento

Complejidad temporal I

- Lo importante en una tabla hash son las búsquedas, así pues cuando hablamos de complejidad temporal de una tabla hash nos referimos a la complejidad temporal de la búsqueda en ella
- Concluimos (no lo demostramos) que la complejidad temporal dependerá únicamente de lo "cargada" que esté la tabla hash
  - Es decir, de L = n/capacidad
  - Por lo tanto, mejor complejidad cuanto más memoria sobre en la tabla

Complejidad temporal II

- Dispersión cerrada:
  - Mejor caso: O(1) cuando el elemento está en su posición ideal
    - Ocurre cuando no se producen colisiones
    - Cuanto más cargada esté la tabla, más probabilidad de colisión
  - Peor caso: O(capacidad) si llegáramos a tener que recorrer toda la tabla para llegar a encontrar (o no encontrar) el elemento
    - Ocurre cuando se producen muchas colisiones…
    - … es decir, cuando la tabla está muy cargada
  - Caso medio: más próximo a O(1) cuando menos cargada está la tabla
    - No lo demostramos

Complejidad temporal III

- Ejemplo de la influencia de la carga en el caso medio en dispersión cerrada (no lo demostramos):
  - T_E es el número medio de intentos para una búsqueda exitosa (el elemento está en la tabla)
  - T_S es el número medio de intentos para una búsqueda sin éxito (el elemento no está en la tabla)
  - L es la carga de la tabla: n/capacidad

	T_S	T_E
L =0.5	2	1.39
L =0.9	10	2.56

Complejidad temporal IV

- Dispersión abierta:
  - Aquí todos los elementos van siempre a la lista que hay en su posición ideal
  - Mejor caso: O(1)
    - Cuando en la posición del elemento hay una lista de un solo elemento…
    - … por lo tanto no hay que buscar en dicha lista
  - Peor caso: O(n)
    - Cuando todos los elementos de la tabla (n) están en la misma posición
    - Tendremos una lista de n elementos
    - Tendremos que buscar dentro de dicha lista. Si la lista es enlazada, tendremos que implementar la búsqueda secuencial, que es O(n)
  - Caso medio cuando n>capacidad: O(n/capacidad) = O(L)
    - Si los elementos están uniformemente distribuidos por la tabla hash tendremos en cada lista “n/capacidad” elementos, por lo tanto la búsqueda secuencial en dicha lista será O(n/capacidad) = O(L)

Complejidad espacial

- Dispersión cerrada:
  - O(1) en todos los métodos (excepto reestructuración) porque los elementos nunca van fuera de la tabla
  - O(capacidad) durante la reestructuración
- Dispersión abierta:
  - Suponemos que en cada celda de la tabla cabe de por sí un elemento, y si queremos añadir más ya hay que insertarlos en forma de lista enlazada
  - O(1) en el mejor caso: cada posición tiene sólo un elemento, y por lo tanto no necesitamos memoria adicional
  - O(n) en el peor caso: todos los elementos de la tabla están en la lista enlazada de una única posición, y el resto de posiciones están vacías
    - Por lo tanto, necesitamos n-1 posiciones de memoria adicional para crear dicha lista enlazada