Arquitectura de sistemas

Abelardo Pardo

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
Autor principal del curso de 2009 a 2012

Iria Estévez Ayres

Damaris Fuentes Lorenzo

Pablo Basanta Val

Pedro J. Muñoz Merino

Hugo A. Parada

Derick Leony

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

© Universidad Carlos III de Madrid | Licencia Creative Commons 

Capítulo 7. Tablas Hash

Tabla de contenidos

Las tablas hash son estructuras de datos que se utilizan para almacenar un número elevado de datos sobre los que se necesitan operaciones de búsqueda e inserción muy eficientes. Una tabla hash almacena un conjunto de pares "(clave, valor)". La clave es única para cada elemento de la tabla y es el dato que se utiliza para buscar un determinado valor.

Un diccionario es un ejemplo de estructura que se puede implementar mediante una tabla hash. Para cada par, la clave es la palabra a buscar, y el valor contiene su significado. El uso de esta estructura de datos es tan común en el desarrollo de aplicaciones que algunos lenguajes las incluyen como tipos básicos.

7.1. Contexto de uso de una tabla hash

Supongamos que en una aplicación se deben manipular un número muy elevado de pares `(clave, valor)`. Estos pares son creados bajo demanda por la aplicación, se manipulan, y se destruyen. La manipulación consiste principalmente en buscar si la aplicación contiene ya un par `(clave, valor)` dado, y si no es así, se añade a la tabla.

Por ejemplo, se dispone de una aplicación un teléfono móvil que es capaz de reconocer mediante la cámara los números de un pasaporte. Para cada pasaporte se almacena un mensaje de texto con un comentario. La aplicación debe realizar las siguientes operaciones:

- Crear una tabla de pasaportes vacía.
- Dado un número de pasaporte, buscar si está almacenado ya en la tabla.
- Dado un número de pasaporte y un comentario, almacenar este par `(clave, valor)` en la tabla. Se asume que no hay información previa sobre este número de pasaporte.

7.2. Posibles implementaciones

Como el número de pares a manipular es indeterminado, se necesita una estructura de datos dinámica. Como primera solución podemos considerar una lista encadenada de pares. La operación de inserción es sencilla. Dado un número de pasaporte y una cadena, se crea un elemento nuevo en la tabla y se inserta. La operación de búsqueda necesita atravesar la lista hasta que, o se encuentra la clave dada (número de pasaporte), o se llega al final (la clave no está en la tabla). Cuando el número de pares es muy elevado, esta búsqueda es muy ineficiente, pues hay que procesar un gran número de elementos.

Una segunda opción para tener una búsqueda muy rápida podría ser almacenar todos los datos en una tabla y utilizar el número de pasaporte como su índice. En este caso, ver si un número está en la tabla consiste simplemente en ver si tiene una cadena asociada. El insertar un nuevo par (pasaporte,

mensaje) también sería muy rápido pues consistiría en guardar el mensaje dado en la posición correspondiente al número. Pero el problema con esta solución es que el número de pasaportes puede ser un entero muy largo y por tanto la tabla que se necesita definir está más allá de lo aceptable.

La tabla hash ofrece un compromiso para esta situación. Los pares (clave, valor) se guardan en una tabla, pero con un tamaño menor del ideal. Para nuestro ejemplo, se utiliza una tabla, pero no tiene como tamaño el número máximo de pasaportes posibles, sino un número más pequeño.

7.3. Tablas hash

Las tablas hash son uno de los mecanismos más utilizados en el desarrollo de aplicaciones (haz una búsqueda en internet del término *hash table* y mira número de enlaces que se devuelven). Existen múltiples librerías en casi todos los lenguajes de programación que proporcionan implementaciones muy eficientes de estas tablas (por ejemplo la clase `java.util.Hashtable` de las librerías del lenguaje Java).

La implementación de una tabla hash está basada en los siguientes elementos:

- Una tabla de un tamaño razonable para almacenar los pares (clave, valor).
- Una función "hash" que recibe la clave y devuelve un índice para acceder a una posición de la tabla.
- Un procedimiento para tratar los casos en los que la función anterior devuelve el mismo índice para dos claves distintas. Esta situación se conoce con el nombre de ***colisión***.

Las posibles implementaciones de cada uno de estos tres elementos se traducen en una infinidad de formas de implementar una tabla hash. A continuación se detalla una solución concreta para el problema de las colisiones.

Responde a las siguientes preguntas para ver si has entendido lo que se explica en este documento:

1. Una función `hash` es:

   - Una función que, dada una clave de un elemento, devuelve el índice de la tabla `hash` donde se encuentra almacenado (o donde hay que almacenar) dicho elemento.
   - Una función que, dada una clave de un elemento, devuelve el elemento almacenado en la tabla `hash`.
   - Una función que, dado un elemento, devuelve su clave en la tabla `hash`.

2. En una tabla `hash` una colisión se produce cuando:

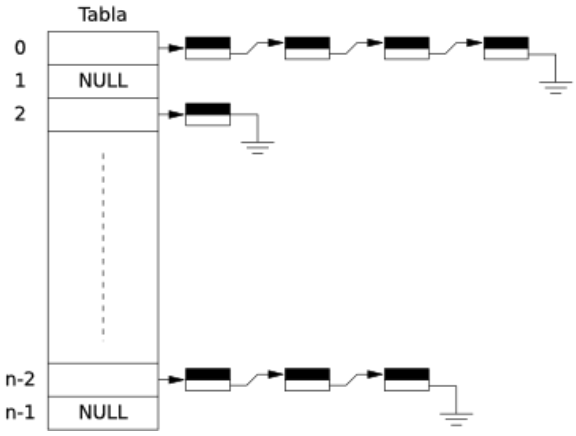
   - En las tablas `hash`, si la función `hash` está bien diseñada no pueden ocurrir.
   - No hay espacio suficiente para guardar todos los elementos en la tabla `hash`. En este caso, la tabla debe redimensionarse.
   - La función `hash` devuelve el mismo índice para dos o más claves diferentes.

7.4. Gestión de colisiones mediante listas encadenadas

La solución propuesta para implementar la tabla hash combina la estructura de tabla con la de lista encadenada. Cada posición de la tabla no almacena un único elemento sino la cabeza de una lista encadenada que a su vez contiene todos aquellos elementos cuya función de hash ha devuelto idéntico resultado. Una posición de la tabla en la que no se haya insertado ningún elemento, contiene un puntero a NULL. La siguiente figura muestra una tabla de tamaño **n** y los elementos almacenados

en las posiciones 0, 1, 2, n-2 y n-1.

Figura 7.1.



Fíjate que diferentes posiciones de la tabla pueden tener listas de diferente longitud, o incluso vacías. Dada una clave, el proceso de búsqueda consiste en calcular primero su índice mediante la función de hash, y a continuación buscar esa clave en la lista de colisiones.

7.5. La función de hash

Como se puede observar en la figura 7.1, la ganancia en eficiencia de una tabla hash con respecto a una lista encadenada que contenga todos los elementos se basa en que la longitud de las listas de colisiones es ***sensiblemente menor*** que el número total de elementos. Si una tabla tiene todas sus cadenas de colisión de igual longitud, el tiempo de búsqueda se reduce sensiblemente. Sin embargo, en el caso extremo en el que ***todos los elementos están en una única cadena de colisión***, la tabla tiene la misma eficiencia que una lista encadenada.

Esta observación sugiere que la función de hash no sólo debe producir un índice en la tabla, sino que para un número elevado de elementos, ***debe producir estos índices lo más dispersos posible***. La especificación de una función hash es muy simple. En el caso de estudio es una función que dado un número de pasaporte devuelve un índice válido de la tabla, y este índice debe ser el mismo para cadenas idénticas. Además, se requiere que el cálculo realizado por esta función sea lo más eficiente posible puesto que se utiliza en todo acceso a la tabla.

7.6. El tamaño de la tabla

El tamaño de una tabla hash es el otro parámetro que debe ser ajustado con cierto cuidado. Al igual que en el caso de la función de hash, se puede analizar cuales son las consecuencias extremas de la elección de un tamaño de tabla inadecuado. Por ejemplo, si el tamaño es demasiado pequeño, digamos 1, de nuevo la tabla se comporta como una lista encadenada. En cambio, si el tamaño es enorme, el malgasto de memoria es evidente, puesto que habrá un elevado número de posiciones cuya lista de colisiones está vacía.

Cuando se utilizan listas de colisión se suele utilizar una técnica basada en una "densidad". Por densidad se entiende el valor al dividir el número de elementos en la tabla y su número de entradas. Si la función de hash produce una distribución uniforme de los elementos, un valor elevado de la densidad denota una longitud excesiva de las listas de colisiones. Un valor de densidad muy reducido denota una longitud muy corta de las cadenas de colisión y por tanto una infrautilización de memoria.

Las implementaciones más complejas de tablas hash incluyen dos umbrales para el valor de la densidad. Cada operación de inserción o borrado de un elemento consulta estos valores. Si la densidad supera el valor máximo, se reserva espacio para una tabla mayor, se borran ***todos los elementos de la tabla antigua*** y se insertan en la nueva (con densidad obviamente menor). Análogamente, si al borrar un elemento la densidad es menor que el umbral inferior, se realiza una operación similar pero con una nueva tabla más pequeña. De esta forma se consigue mantener de

operación similar pero con una nueva tabla más pequeña. De esta forma se consigue mantener de una forma transparente al exterior, el valor de la densidad en límites razonables. La operación de redimensionado de la tabla es cara en tiempo de CPU, pero se supone que las operaciones futuras de búsqueda, al encontrar listas de colisión de la dimensión adecuada, rentabilizan su ejecución.

Comprueba con estas preguntas si has entendido este documento.

Responde a las siguientes preguntas para ver si has entendido lo que se explica en este documento:

1. Para una tabla `hash` su densidad es:

   - El número máximo de índices que puede devolver la función `hash` asociada a dicha tabla.
   - El ratio entre el tamaño de la tabla `hash` (número de índices) y el número de elementos que se guardan en la estructura de datos.
   - El ratio entre el número de elementos que se guardan en la estructura de datos y el tamaño de la tabla `hash` (número de índices).

7.7. Bibliografía de apoyo

- ***Funciones Hash (teoría y ejemplo)***: "Programación en C: Metodología, algoritmos y estructura de datos", páginas 540-541.

7.8. Preguntas de autoevaluación

Comprueba con estas preguntas si has entendido este documento.

1. Este es el código incompleto de definición de la estructura de datos de una tabla `hash` con colisiones gestionadas a través de estructuras enlazadas.

```
1  struct datos {
2   char *name;
3   int nia;
4  };
5  struct elemento_t {
6   struct datos valor;
7   struct elemento_t *next;
8  };
9  struct tabla_type {
10  int tamanho;
11  int num_elementos;
12  double densidad_deseada;
13  ??????????????????????
14 };
```

   Indique cuál es el trozo de código que falta

   - `struct elemento_t *tabla;`
   - `struct elemento_t **tabla;`
   - `struct elemento_t tabla;`

2. Este es el código incompleto de inicialización de la tabla `hash` del ejercicio anterior.

```
1 tabla_type tabla;
```

```
2 int i;
3 tabla.tamanho=20;
4 tabla.num_elementos=0;
5 tabla.densidad_deseada=0.25;
6 tabla.tabla = (elemento_type **) malloc (sizeof(elemento_type *)*tabla.tamanho);
7 ????????????????
```

Indique cuál es el trozo de código que falta

-
```
for (i=0;i<tabla.tamanho;i++)
{
 tabla.tabla[i]=NULL;
}
```
-
```
for (i=0;i<tabla.tamanho;i++)
{
 tabla.tabla=NULL;
}
```
- El código está completo.

3. Este es el código incompleto de la función `hash` y de las funciones necesarias para insertar un nuevo elemento.

```
1  int hash(int clave,tabla_type *tabla)
2  {
3  /*fmod returns the module of the division*/
4  return (int ) fmod((float)clave,(float)tabla->tamanho);
5  }
6
7  elemento_type *new_element(int clave, datos valor, elemento_type *next){
8   elemento_type *new_el = (elemento_type *)malloc(sizeof (elemento_type));
9   new_el->datos=valor;
10  new_el->clave=clave;
11  new_el->next=next;
12  return new_el;
13 }
14
15 void insert_element(int clave, datos valor, tabla_type *tabla){
16  int index;
17  index = hash(clave,tabla);
18 ????????????????????????????????
19 }
```

Indique cuál es el trozo de código que falta

- `new_element(clave,valor,tabla);`
- `tabla=new_element(clave,valor,tabla[index]);`
- `tabla[index]=new element(clave,valor,tabla[index]);`