

Clase 31

Hashing (Dispersión)

Motivación

- ¿Podemos realizar una búsqueda en un tiempo mejor que $O(\lg n)$?
- La operación de memoria de un ordenador lo realiza en mucho menos tiempo: toma una clave (la dirección de la memoria) para insertar o recuperar elementos de datos (la palabra de la memoria) en un tiempo constante ($O(1)$).
- Los tiempos de acceso para la memoria de un ordenador no aumentan con el tamaño o la proporción de la memoria.

Direccionamiento directo

- El acceso a la memoria del ordenador es un caso especial de una técnica denominada *direccionamiento directo* en la que la clave conduce directamente al elemento de datos.
- El almacenamiento de datos en *arrays* también es otro ejemplo de este método, en el que el índice del *array* juega el papel de la clave.
- El problema de los esquemas de direccionamiento directo reside en que requieren un almacenamiento equivalente al rango de todas las posibles claves y no es proporcional al número de elementos que realmente se almacenan.

3

Ejemplo de direccionamiento directo

- Utilicemos el ejemplo de los números de la seguridad social.
- Un esquema de direccionamiento directo para almacenar la información de ingresos de los contribuyentes estadounidenses requeriría una tabla de 1.000.000.000 de entradas, puesto que el número de la seguridad social es de 9 dígitos.
- No importa si esperamos almacenar datos de 100 o de 100.000.000 contribuyentes.
- El esquema de direccionamiento directo necesitará una tabla que pueda incluir mil millones de entradas potenciales.

4

Hashing

- El *hashing* es una técnica que ofrece una velocidad comparable a la del direccionamiento directo ($O(1)$) con requisitos de memoria ($O(n)$) mucho más gestionables, donde n es el número de entradas almacenadas en la tabla.
- El *hashing* utiliza una función para generar un código *hash* pseudoaleatorio a partir de la clave del objeto y luego utiliza este código (~dirección directa) para indexar en la tabla *hash*.

5

Ejemplo de *hashing*

- Suponga que queremos una pequeña tabla *hash* con capacidad de 16 entradas para almacenar palabras en inglés.
- Necesitaremos una función *hash* que asigne las palabras a los enteros 0, 1, ..., 15.
- Normalmente, la tarea se divide en crear una función *hash* en dos partes:
 1. Asignar la clave a un entero.
 2. Asignar el entero "aleatoriamente" o en un rango bien distribuido de enteros ({ 0, ..., $m-1$ }, donde m es la capacidad o el número de entradas que se utilizarán para indexar en la tabla *hash*.

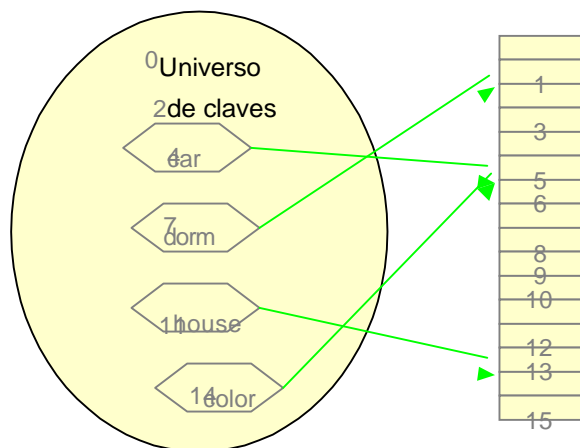
6

Ejemplo de código *hash*

- Como ejemplo, considere un código *hash* que toma el valor numérico del primer carácter de la palabra y lo agrega al valor numérico del último carácter (*paso 1*) para, después, tomar el resto *mod 16* (*paso 2*).
- Por ejemplo, el valor numérico de una "c" es 99 y el de una "r" es 114. Por lo que, "car" se dispersaría en $(99 + 114) \bmod 16 = 5$.

7

Diagrama de código *hash*



8

Colisiones

- "car" y "color" se dispersan al mismo valor utilizando esta función *hash*, ya que tienen la misma letra inicial y final. Nuestra función *hash* podría no ser tan "aleatoria" como debiera.
- Pero si $n > m$, los códigos *hash* duplicados, también conocidos como *colisiones*, son inevitables.
- De hecho, incluso con $n < m$, las colisiones pueden seguir siendo consecuencia del argumento *von Mises* (también conocido como la paradoja del cumpleaños: si hay 23 personas en una habitación, la probabilidad de que al menos dos de ellas cumplan los años el mismo día supera el 50%).

9

Tareas de *hashing*

1. Diseñar una función *hash* adecuada para asignar códigos *hash* a las claves, de tal forma que un conjunto no aleatorio de claves genere un conjunto equilibrado "aleatorio" de código *hash*;
Si los códigos *hash* no son aleatorios, el número excesivo de colisiones "agolpará" las claves en la misma dirección directa.
2. Tratar las posibles colisiones que aparezcan después del *hashing*.

10

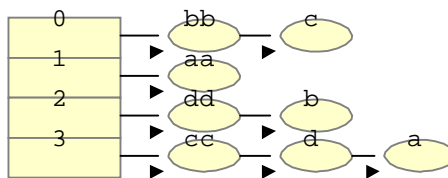
Encadenamiento para evitar colisiones

- El *encadenamiento* es un enfoque sencillo y eficaz para gestionar las colisiones.
- En una tabla *hash* con encadenamiento, las entradas de la tabla (normalmente llamadas *slots* o *buckets*, no contienen los propios objetos almacenados, sino listas enlazadas de los objetos.
- Los objetos con claves colisionadas se insertan en la misma lista.
- La inserción, la búsqueda y la eliminación se convierten en procesos de 2 pasos:
 1. Utilizar la función *hash* para seleccionar el *slot* adecuado.
 2. Realizar la operación necesaria en la lista enlazada a la que se hace referencia en dicho *slot*.

11

Ilustración de encadenamiento

claves = { a, b, c, d, aa, bb, cc, dd }



12

Factor de carga y rendimiento

- La relación entre el número de elementos almacenados, n , y el número de *slots* de la tabla, m , n/m , recibe el nombre de *factor de carga*.
- Como las listas enlazadas a las que se hace referencia en los *slots hash* pueden contener un número arbitrario de elementos, no existe límite en la capacidad de la tabla *hash* que utiliza el encadenamiento.
- Si la función *hash* empleada no distribuye bien las claves, el rendimiento de la tabla disminuirá.
- El peor caso para una tabla *hash* y para un árbol binario es el de la lista enlazada. Esto ocurre cuando todas las claves se dispersan al mismo *slot*.
- Sin embargo, dada una buena función *hash*, se puede demostrar que una tabla *hash* con encadenamiento y un factor de carga L puede realizar las operaciones básicas de inserción, búsqueda y eliminación en un tiempo $O(1 + L)$.
- Para una mayor eficacia, el factor de carga debe ser ≤ 0.75

Iteradores de la tabla *hash*

- La iteración en una tabla *hash* también es un proceso de dos pasos.
Para cada slot con una list enlazada, hlist
Para cada elemento, i, en hlist
Devolver i
- Como los códigos *hash* asignan elementos a *slots* "aleatoriamente" y no estamos ordenando las listas enlazadas, dicha iteración no está ordenada.
- El orden y la localidad son una de las características buscadas para la velocidad de acceso en las tablas *hash*.

Eficacia del iterador

- El tiempo que se tarda en iterar una tabla con pocas colisiones es proporcional a la capacidad de la tabla o al número de *slots*, no al número de elementos almacenados.
- El tiempo aproximado que tardará en iterarse una tabla de gran capacidad con pocos elementos será equivalente al de una tabla en la que el número de elementos almacenados se asemeje al número de *slots*.
- Las aplicaciones tradicionales de tablas *hash* no requieren acceso ordenado ni la posibilidad de iterar.

15

Aplicaciones típicas de tablas *hash*

- Uno de los usos clásicos de las tablas *hash* es gestionar la tabla de símbolos de intérpretes y compiladores. Los nombres de símbolos (p.ej, nombres de variables) son la clave y los datos de símbolos (p.ej., tipo, ubicación, etc.) están contenidos en el objeto almacenado. La necesidad principal es la búsqueda rápida.
- Si se requiere una lista ordenada de la tabla de símbolos (y ahora es más difícil que nunca) puede ofrecerse en un segundo análisis.
- Otros usos típicos de tablas *hash* suelen incluir la gestión de una clase de datos por una clave arbitraria, como el número de la seguridad social, el número de cuenta o el número de DNI.

16

Funciones *hash*

- Las funciones *hash* eficaces son cruciales para una buena implementación de tablas *hash*.
- Solemos querer almacenar claves que son cualquier cosa excepto aleatorias. Piense en almacenar nombres de personas con el apellido en primer lugar. En EE.UU. habrá muchas claves que comiencen por "Smith, ..." y que terminen por "..., John". Necesitaríamos que la función de *hashing* para los nombres obtuviese como resultado una distribución de las entradas "Smith" en *slots* dispersos independientes y que distinguiese "Smith, John" de, por ejemplo, "Harward, Judson".

17

Funciones *hash*₁ y *hash*₂

- Asumamos que queremos almacenar n objetos de un universo U de N objetos distintos con N claves distintas en una tabla *hash* T , con m *slots*. Asumamos de momento que $n \ll N$ y $n \leq m$.
- Formalmente, necesitamos una función *hash*, *hash*(k), que asigne cada objeto $k \in U$ a un entero h , $0 \leq h < m$, desde un punto de vista menos formal, *hash*(k) debe asignar cada k a su *slot*. Ya hemos mencionado que una función *hash* suele considerarse una composición de dos funciones: $hash_1: U \rightarrow I$ y $hash_2: I \rightarrow \{ h \in I \mid 0 \leq h < m \}$, donde I es el conjunto de enteros.

18

hashCode ()

- En un lenguaje orientado a objetos como es Java, la primera fase de *hashing*, la función *hash*, es responsabilidad de la clase de la clave, no de la clase de la tabla *hash*.
- La tabla *hash* almacenará las entradas como `Object`. No tiene información suficiente para generar un código *hash* desde `Object`, que podría ser un `String`, un `Integer` o un objeto personalizado.
- Java confirma esto mediante el método `hashCode ()` en `Object`. Todas las clases de Java amplían implícita o explícitamente a `Object`. Y `Object` posee un método `hashCode ()` que devuelve un `int`.
- Cuidado: el método `hashCode ()` puede devolver un entero negativo en Java; si queremos un entero no negativo (y solemos quererlo) deberemos tomar el valor absoluto de `hashCode ()`.

19

¿Qué propiedades debe tener un código *hash*?

- Como utilizamos códigos *hash* para indexar en la tabla *hash* y encontrar objetos, el código *hash* del objeto debe permanecer fijo a lo largo del programa.
- Un número realmente aleatorio no sería válido para un código *hash* aceptable, ya que generaría un valor distinto cada vez que dispersásemos.
- Si dos objetos son equivalentes (pero no necesariamente idénticos), como dos copias de la misma cadena, los códigos *hash* de los dos objetos también serían iguales, ya que deben recuperar el mismo objeto de la tabla *hash*.
- Formalmente, si `o1` y `o2` son `Object` y `o1.equals(o2)`, entonces `o1.hashCode ()` debería ser `== o2.hashCode ()`.
- Si ignora los métodos `equals ()` de una clase, probablemente también debería ignorar el método `hashCode ()`.

20

Diseño del código *hash*

- El *hashing* tiene mucho más de arte que de ciencia, especialmente en el diseño de funciones *hash*₁.
- La última comprobación de un buen código *hash* es ver si distribuye sus claves de un modo “aleatorio” correcto.
- Se debe seguir algunos principios:
 1. Un código *hash* debe depender tanto como sea posible de la clave.
 2. Un código *hash* debería asumir que sufrirá manipulaciones posteriores para adaptarse al tamaño concreto de la tabla: la fase *hash*₂.

21

hashCode () de la clase String

- La clase String de Java ignora Object() y debe ignorar, por tanto, hashCode ().
- La representación interna de una cadena en Java es un *array* de caracteres:

```
// almacenamiento de caracteres
private char value[];
// offset es el índice de la primera posición utilizada
private int offset;
// count es el número de caracteres de la cadena
private int count;
// Almacena en caché el código hash de la cadena
private int hash = 0;
```

22

hashCode() de la clase String, 2

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++)
            h = 31*h + val[off++];
        hash = h;
    }
    return h;
}
```

23

La función $hash_2$

- Una vez que el método `hashCode()` devuelve un `int`, todavía nos queda distribuirlo (éste es el papel de $hash_2$) en uno de los m slots, h , $0 \leq h < m$. La forma más sencilla de hacerlo es tomar el valor absoluto del módulo del código $hash$ dividido por el tamaño de la tabla, m : `k = Math.abs(o.hashCode() % m);`
- Este método puede no distribuir bien las claves pero, dependiendo del tamaño m (en concreto, si m es una potencia de 2, 2^p), este $hash_2$ simplemente extraerá los bits p de menor orden del $hash$ de entrada.
- Si puede confiar en la aleatoriedad del $hash_1$ de entrada, este método probablemente sea adecuado. Si no puede, se recomienda que utilice un esquema más elaborado para realizar un $hash$ adicional con el $hash$ de entrada como clave.

24

Hashing de enteros

Un buen método para dispersar un entero (incluyendo nuestros códigos *hash*) es multiplicar el entero por un número A , $0 < A < 1$, extraer la parte fraccional, multiplicarla por el número de *slots* de la tabla, m , y truncarlo a un entero. En Java, si `hcode` es el entero que debe redispersarse, esto se convierte en:

```
private int hashCode( int n ) {  
    double t = Math.abs( n ) * A;  
    return ( (int) ( ( t - (int)t ) * m ) );  
}
```

Dejando a un lado la intuición, ciertos valores de A parecen funcionar mucho mejor que otros. Los estudios sugieren que el valor recíproco de la relación áurea ($\text{sqrt}(5.0) - 1.0$) / 2.0) obtiene resultados especialmente buenos.

25

Implementación de la tabla *hash*

- Vamos a implementar nuestra tabla *hash* (`HashMap`) como un `Map` con claves y valores.
- Un `HashMap` es un `Map`, no un `OrderedMap`, ya que no está ordenado. `Map` no tiene métodos `firstKey()` y `lastKey()`.
- Vamos a utilizar *listas enlazadas sencillas* para solucionar las colisiones.
- Dado que la implementación de la lista enlazada sencilla de la clase 25 no es un mapa y no contiene valores y claves, hemos incluido aquí una implementación reducida de un mapa de lista enlazada sencilla en la clase `HashMap` misma.

26

Miembros de HashMap

```
public class HashMap
    implements Map
{
    private int length = 0;
    // encabezados de cadenas para slots
    private Entry [] table = null;
    private static final double golden =
        (Math.sqrt(5.0) - 1.0)/2.0;
    public static final int DEFAULT_SLOTS = 64;

    public HashMap( int slots ) {
        table = new Entry[ slots ];
        clear();
    }
}
```

27

Entrada de clase interna estática

```
private static class Entry {
    final Object key;
    Object value;
    Entry next;

    Entry( Object k, Object v, Entry n )
    { key = k; value = v; next = n; }

    Entry( Object k, Object v )
    { key = k; value = v; next = null; }

    Entry( Object k )
    { key = k; value = null; next = null; }
}
```

28

Método clear()

```
public void clear()
{
    length = 0;
    for ( int i = 0; i < table.length; i++ )
        table[ i ] = null;
}
```

29

Método put ()

```
public Object put( Object k, Object v )
{
    if ( k == null )
        throw new IllegalArgumentException(
            "Clave null no permitida"
        ); int idx = index( k.hashCode() );
    Entry current = table[ idx ];
```

30

Método put (), 2

```
// Si la clave existe en el mapa, salir apuntando
// hacia ella; si no valor actual == null
while ( current != null )
{
    if ( current.key.equals( k ) )
        break;
    current = current.next;
}
```

31

Método put (), 3

```
if ( current == null ) { // ¿Lo encontramos?
    // No, insertar un nuevo elemento en encabezado de cadena
    length++;
    // Nuevo encabezado apunta al antiguo
    table[ idx ] = new Entry( k, v, table[ idx ] );
    return null;
} else { // Sí, lo encontramos
    // Reemplazar valor y devolver antiguo
    Object ret = current.value; current.value =
    v;
    return ret;
}
}
```

32

Método index()

```
// Volver a dispersar para garantizar buena distribución
// hash
private int index( int hcode )
{
    double t = Math.abs( hcode ) * GOLDEN;
    return ((int) ((t - (int)t) * table.length ));
}
```

33

Método get ()

```
public Object get( Object k ) {
    if ( k == null )
        throw new IllegalArgumentException(
            "Clave null no permitida" );
    Entry current = table[ index( k.hashCode()) ];
    // buscar coincidencia en este slot;
    // si el slot está vacío, devolver inmediatamente
    while ( current != null ) {
        if ( current.key.equals( k ) )
            return current.value;
        current = current.next;
    }
    return null;
}
```

34