Tablas asociativas
Tablas de dispersión

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

July 7, 2025

Descripción de la unidad

En este capítulo se considera el concepto de dispersión y su aplicación para la construcción de tablas de símbolos no ordenadas que permiten búsquedas muy eficientes.

Contents

1 Tablas de dispersión (Hash Tables) 2
1.1 Funciones de dispersión (Hash Functions) . . . 2
1.1.1 Propiedades deseables de una función de dispersión: . . . 2
1.1.2 Ejemplos de funciones de dispersión comunes: . . . 2
1.2 Tablas de dispersión con encadenamiento (Hash Tables with Chaining) . . . 3
1.3 Direccionamiento abierto (Open Addressing) . . . 4
1.4 Cuckoo Hashing . . . 8
1.5 Rendimiento de las tablas de dispersión . . . 8
1.6 Aplicaciones de las tablas de dispersión . . . 8
1.7 Ejercicios y Actividades Prácticas . . . 10

1 Tablas de dispersión (Hash Tables)

Las tablas de dispersión, también conocidas como tablas hash (*hash tables*), son estructuras de datos que implementan el Abstract Data Type (ADT) tabla de símbolos (o diccionario) de manera eficiente. A diferencia de las implementaciones basadas en árboles de búsqueda, las tablas de dispersión no requieren que las llaves sean comparables, solo permiten implementar tablas de símbolos no ordenadas. Son ampliamente utilizadas en diversas aplicaciones, desde bases de datos hasta compiladores, debido a su eficiencia en la búsqueda, inserción y eliminación de elementos.

La idea central detrás de las tablas de dispersión es utilizar una *función hash* para mapear las llaves a los índices de un arreglo. Idealmente, esta función debería asignar cada llave a una posición única en el arreglo, permitiendo el acceso en tiempo casi constante. Sin embargo, en la práctica, es posible que diferentes llaves sean mapeadas a la misma posición, lo que se conoce como *colisión*. La gestión eficiente de estas colisiones es crucial para el rendimiento de la tabla de dispersión.

La clave (*key*) utilizada en una tabla hash debe ser inmutable, es decir, su valor no debe cambiar después de ser insertada en la tabla. Esto es fundamental para garantizar que la función hash siempre produzca el mismo índice para la misma clave, permitiendo la correcta recuperación del valor asociado.

1.1 Funciones de dispersión (Hash Functions)

Una función de dispersión (o función hash) es una función que toma una llave como entrada y devuelve un valor entero, que se utiliza como índice en la tabla de dispersión. La elección de una buena función hash es crucial para el rendimiento de la tabla.

1.1.1 Propiedades deseables de una función de dispersión:

- **Uniformidad:** Debería distribuir las llaves de manera uniforme a lo largo de la tabla, minimizando las colisiones. Una función hash ideal asigna cada llave a una posición única en la tabla, pero esto rara vez es posible en la práctica.
- **Eficiencia:** Debería ser rápida de calcular, ya que se invoca en cada operación de inserción, búsqueda y eliminación. La eficiencia es especialmente importante en aplicaciones donde se realizan muchas operaciones en la tabla hash.
- **Determinismo:** Para una misma llave, la función hash siempre debe devolver el mismo valor. Esto es esencial para garantizar que la búsqueda de una llave siempre la encuentre en la misma posición.

1.1.2 Ejemplos de funciones de dispersión comunes:

- **Función de división (Division Method):** Esta función calcula el hash como el residuo de la división de la llave por el tamaño de la tabla: $h(k) = k \mod m$, donde k es la llave y m es el tamaño de la tabla. Es simple y rápida, pero puede generar muchas colisiones si el tamaño de la tabla es un múltiplo de un factor común de las llaves. Es recomendable que m sea un número primo. Esto ayuda a que todos los bits de k influyan en el resultado.
- **Función de multiplicación (Multiplication Method):** Esta función calcula el hash multiplicando la llave por una constante A en el rango $0 < A < 1$, extrayendo la parte fraccionaria del resultado, y multiplicándola por el tamaño de la tabla: $h(k) = \lfloor m(kA \mod 1) \rfloor$, donde k es la llave, m es el tamaño de la tabla, y A es una constante. La elección de A es importante

para la uniformidad de la distribución, un valor comúnmente usado de A es el conjugado de la razón aurea, $A = \frac{\sqrt{5}-1}{2}$.

- **Funciones para cadenas de caracteres:** Para cadenas de caracteres, se pueden utilizar funciones que consideran cada carácter como un "digito" (*e.g.* su código ASCII) de un número base R. El valor decimal equivalente módulo m seria el hash de la cadena:

$$h(k) = \left(c_0 R^0 + c_1 R^1 + \ldots + c_{n-1} R^{n-1}\right) \mod m$$

  Funciones más avanzadas, como las que utilizan operaciones de desplazamiento de bits y XOR, pueden mejorar la distribución.

Ejemplo de función de dispersión no válida: Una función que siempre devuelve el mismo valor (e.g., `return 0;`) es una función de dispersión no válida, ya que causa que todas las llaves colisionen en la misma posición. Asimismo, una función de dispersión que para una llave de entrada no devuelva el mismo valor de salida, tampoco es válida. Por esta razón se exige que las funciones de dispersión sean consistentes con la igualdad: Llaves de igual valor siempre deben retornar el mismo hash.

Funciones de dispersión para otros tipos de datos: Las funciones de dispersión pueden diseñarse para diferentes tipos de datos, como cadenas de caracteres, audios, imágenes, etc. Para tipos de datos más complejos, es común combinar los valores hash de sus componentes. Cuando se implementa un ADT es importante sobre-escribir el método `hashCode` con el fin de implementar el cálculo de la función hash **con base en el valor** del ADT.

Consideraciones importantes al implementar hash functions para valores double:

- **Precisión:** Los valores double tienen limitaciones de precisión que pueden causar que valores que deberían ser iguales tengan diferentes representaciones binarias. Por esto, es importante definir un epsilon para comparaciones de igualdad.
- **Consistencia:** La implementación del hash debe ser consistente con la igualdad. Si dos puntos son considerados iguales según `equals()`, deben tener el mismo valor hash.
- **Redondeo:** En algunos casos, puede ser deseable redondear los valores a una precisión específica antes de calcular el hash. Esto puede ser útil cuando se trabaja con datos que tienen errores de medición o cuando la precisión exacta no es necesaria.
- **Conversión a bits:** El uso de `Double.doubleToLongBits()` es importante para preservar la representación exacta del valor double, incluyendo casos especiales como NaN e infinito.

Funciones de dispersión vs. hash criptográficos: Es importante distinguir entre las funciones de dispersión utilizadas en las tablas de dispersión y las funciones hash criptográficas. Las funciones hash criptográficas están diseñadas para ser resistentes a colisiones y preimágenes, y se utilizan en aplicaciones de seguridad. Las funciones de dispersión para tablas de dispersión priorizan la eficiencia y la uniformidad, y no necesitan ser resistentes a ataques.

Algunos ejemplos de funciones hash criptográficas son: MD5, SHA.

1.2 Tablas de dispersión con encadenamiento (Hash Tables with Chaining)

El encadenamiento, también conocido como *separate chaining* (encadenamiento separado), es una técnica común para resolver colisiones en las tablas de dispersión. En esta técnica, cada posición

del arreglo contiene una lista enlazada de todas las llaves que se mapean a esa posición. Es decir, en lugar de almacenar directamente el valor en la tabla, se almacena una lista de pares llave-valor que tienen el mismo índice hash.

Implementación de la tabla de dispersión con encadenamiento: La tabla de dispersión se implementa como un arreglo de listas enlazadas. Cuando se inserta una nueva llave, se calcula su valor hash, y se agrega la llave al principio de la lista enlazada en esa posición. La búsqueda y eliminación se realizan de manera similar, recorriendo la lista enlazada en la posición correspondiente.

Complejidad en el peor caso: En el peor caso, todas las llaves se mapean a la misma posición, y la tabla de dispersión se degenera en una lista enlazada. En este caso, las operaciones de inserción, búsqueda y eliminación tienen una complejidad de $O(n)$, donde n es el número de llaves en la tabla.

Factor de carga y redispersión: El factor de carga (load factor) de una tabla de dispersión se define como el número de llaves almacenadas n dividido por el tamaño del arreglo m.

$$factor\_carga = \frac{n}{m}$$

Asumiendo una función de dispersión uniforme, el factor de carga corresponde a la longitud promedio de las listas enlazadas.

Un factor de carga alto indica que la tabla está "llena" y que las colisiones son más frecuentes. Para mantener un buen rendimiento, es común realizar una redispersión (rehashing) cuando el factor de carga excede un cierto umbral. La redispersión implica crear una nueva tabla de dispersión más grande, y reinsertar todas las llaves en la nueva tabla.

Complejidad con factor de carga acotado: Si el factor de carga se mantiene acotado (e.g., menor que una constante), la complejidad promedio de las operaciones de inserción, búsqueda y eliminación es $O(1)$. Esto se debe a que la longitud promedio de las listas enlazadas es constante.

Ejemplo de implementación completa de una tabla hash con encadenamiento:

1.3 Direccionamiento abierto (Open Addressing)

Existen otras soluciones para el problema de las colisiones en las tablas de dispersión, además del encadenamiento. Una de las más comunes es el *direccionamiento abierto* (open addressing).

En esta técnica, todas las llaves se almacenan directamente en el arreglo. Cuando ocurre una colisión, se busca una posición vacía en el arreglo utilizando una función de *probing*. Algunas variantes comunes de direccionamiento abierto son:

- *Linear Probing:* Se busca la siguiente posición vacía en el arreglo de forma lineal. Si la posición está ocupada, se prueba la siguiente, y así sucesivamente, hasta encontrar una posición vacía.
- *Quadratic Probing:* Se busca una posición vacía utilizando una función cuadrática. Esto ayuda a evitar el agrupamiento primario que puede ocurrir con el linear probing.
- *Double Hashing:* Se utiliza una segunda función hash para determinar el intervalo entre las posiciones que se exploran. Esto proporciona una mejor distribución de las llaves que el *linear* y *quadratic probing*.

El algoritmo de *rehashing* es el mismo que para la tabla hash con encadenamiento.

Algorithm 1 Hash Table with Chaining - Insertion

Require: *key*: Key to insert, *value*: Value to insert, *table*: Hash table (array of linked lists), *hash*(*key*): Hash function, *m*: Table size, *loadFactor*: Load factor threshold

Ensure: Key-value pair inserted into the hash table

```
1: function INSERT(key, value)
2:     index ← hash(key) mod m
3:     current ← table[index]
4:     while current ≠ null do
5:         if current.key == key then
6:             current.value ← value return
7:         end if
8:         current ← current.next
9:     end while
10:    Create new node newNode with key, value
11:    newNode.next ← table[index]
12:    table[index] ← newNode
13:    size ← size + 1
14:    if size/m > loadFactor then
15:        REHASH
16:    end if
17: end function
```

Algorithm 2 Hash Table with Chaining - Search

Require: *key*: Key to search, *table*: Hash table (array of linked lists), *hash*(*key*): Hash function, *m*: Table size

Ensure: Value associated with the key, or null if not found

```
1: function SEARCH(key)
2:     index ← hash(key) mod m
3:     current ← table[index]
4:     while current ≠ null do
5:         if current.key == key then
6:             return current.value
7:         end if
8:         current ← current.next
9:     end while
10:    return null
11: end function
```

Algorithm 3 Hash Table with Chaining - Delete

Require: *key*: Key to delete, *table*: Hash table (array of linked lists), $hash(key)$: Hash function, m: Table size

Ensure: Key-value pair removed from the hash table

```
1: function DELETE(key)
2:     index ← hash(key) mod m
3:     current ← table[index]
4:     prev ← null
5:     while current ≠ null do
6:         if current.key == key then
7:             if prev == null then
8:                 table[index] ← current.next
9:             else
10:                prev.next ← current.next
11:            end if
12:            size ← size − 1 return
13:        end if
14:        prev ← current
15:        current ← current.next
16:    end while
17: end function
```

Algorithm 4 Hash Table with Chaining - Rehash

Require: *table*: Hash table, m: Table size, $hash(key)$: Hash function

Ensure: Hash table rehashed with a new size (e.g., doubled)

```
1: function REHASH
2:     oldTable ← table
3:     m ← m * 2 // Double the table size
4:     table ← new array of size m
5:     size ← 0
6:     for each entry in oldTable do
7:         while entry != null do
8:             INSERT(entry.key, entry.value)
9:             entry = entry.next
10:        end while
11:    end for
12: end function
```

Algorithm 5 Hash Table with Linear Probing - Insertion

Require: key: Key to insert, $value$: Value to insert, $table$: Hash table (array), $hash(key)$: Hash function, m: Table size, $loadFactor$: Load factor threshold
Ensure: Key-value pair inserted into the hash table

```
1: function INSERT(key, value)
2:     if size/m ≥ loadFactor then
3:         REHASH
4:     end if
5:     index ← hash(key) mod m
6:     originalIndex ← index
7:     repeat
8:         if table[index] == null or table[index].isDeleted == true then
9:             table[index] ← new Entry(key, value)
10:            size ← size + 1 return
11:        end if
12:        if table[index].key == key then
13:            table[index].value ← value return
14:        end if
15:        index ← (index + 1) mod m
16:    until index == originalIndex
17:    REHASH // Table full
18:    INSERT(key, value) // Retry insertion after rehashing
19: end function
```

Algorithm 6 Hash Table with Linear Probing - Search

Require: key: Key to search, $table$: Hash table (array), $hash(key)$: Hash function, m: Table size
Ensure: Value associated with the key, or null if not found

```
1: function SEARCH(key)
2:     index ← hash(key) mod m
3:     originalIndex ← index
4:     repeat
5:         if table[index] == null then
6:             return null
7:         end if
8:         if table[index].key == key and table[index].isDeleted == false then
9:             return table[index].value
10:        end if
11:        index ← (index + 1) mod m
12:    until index == originalIndex
13:    return null
14: end function
```

Algorithm 7 Hash Table with Linear Probing - Delete

Require: key: Key to delete, $table$: Hash table (array), $hash(key)$: Hash function, m: Table size
Ensure: Key-value pair removed from the hash table

```
1: function DELETE(key)
2:     index ← hash(key) mod m
3:     originalIndex ← index
4:     repeat
5:         if table[index] == null then
6:             return
7:         end if
8:         if table[index].key == key and table[index].isDeleted == false then
9:             table[index].isDeleted ← true
10:            size ← size − 1 return
11:        end if
12:        index ← (index + 1) mod m
13:    until index == originalIndex
14: end function
```

1.4 Cuckoo Hashing

Cuckoo Hashing es una técnica que utiliza dos funciones hash diferentes. Cuando se inserta una nueva llave, se calcula su posición utilizando la primera función hash. Si la posición está ocupada, la llave que estaba en esa posición se "desaloja" y se reinserta utilizando la segunda función hash. Este proceso se repite hasta que se encuentra una posición vacía o se alcanza un límite máximo de iteraciones. Si se alcanza el límite máximo de iteraciones, la tabla se redispersa.

1.5 Rendimiento de las tablas de dispersión

El rendimiento de una tabla de dispersión depende de varios factores, incluyendo la calidad de la función hash, la técnica de manejo de colisiones, y el factor de carga.

En general, las tablas de dispersión con encadenamiento y direccionamiento abierto tienen un rendimiento promedio de $O(1)$ para las operaciones de inserción, búsqueda y eliminación, siempre y cuando el factor de carga se mantenga acotado. Sin embargo, en el peor caso, el rendimiento puede ser de $O(n)$, donde n es el número de llaves en la tabla.

Cuckoo hashing puede ofrecer un rendimiento aún mejor en la práctica, pero su rendimiento en el peor caso es difícil de predecir.

1.6 Aplicaciones de las tablas de dispersión

Las tablas de dispersión tienen una amplia variedad de aplicaciones, incluyendo:

- **Bases de datos:** Las tablas de dispersión se utilizan para indexar las tablas de las bases de datos, permitiendo la búsqueda rápida de registros.
  B-Tree vs Hash index en MySQL
- **Compiladores:** Las tablas de dispersión se utilizan para implementar las tablas de símbolos de los compiladores, que almacenan información sobre las variables, funciones y otros elementos del programa.

- **Cachés:** Las tablas de dispersión se utilizan para implementar las cachés, que almacenan datos que se acceden con frecuencia para acelerar el acceso a los mismos.
  Ejemplos de servicios de caché implementados bajo esta filosofía son Redis y Memcached.

- **Diccionarios:** Las tablas de dispersión se utilizan para implementar los diccionarios, que son estructuras de datos que asocian claves con valores.

Importancia de la consistencia entre equals() y hashCode():
La consistencia entre los métodos `equals()` y `hashCode()` es fundamental para el correcto funcionamiento de las tablas de dispersión. Esta relación debe cumplir el siguiente contrato:

- Si `a.equals(b)` es `true`, entonces `a.hashCode() == b.hashCode()` debe ser `true`.
- Si `a.hashCode() != b.hashCode()`, entonces `a.equals(b)` debe ser `false`.

¿Por qué es importante?

- **Correctitud:** Si dos objetos son considerados iguales según `equals()`, pero tienen diferentes valores hash, la tabla de dispersión podría almacenar múltiples copias del mismo objeto, lo que viola la semántica de la estructura de datos.
- **Rendimiento:** Si dos objetos tienen el mismo hash pero no son iguales según `equals()`, se producirán colisiones innecesarias, degradando el rendimiento de la tabla de dispersión.
- **Comportamiento inesperado:** La violación de este contrato puede llevar a comportamientos inesperados en colecciones que dependen de hash, como `HashSet` o `HashMap`.

Consecuencias de la violación:

- **Duplicación de datos:** En un `HashSet`, podrían existir múltiples objetos que son considerados iguales según `equals()`.
- **Pérdida de datos:** En un `HashMap`, al usar un objeto como clave, podríamos no encontrar el valor asociado aunque la clave exista en el mapa.
- **Comportamiento no determinista:** El comportamiento de las colecciones basadas en hash podría variar dependiendo de cómo se calculen los hash codes.

Buenas prácticas:

- Siempre implementar `equals()` y `hashCode()` juntos.
- Asegurarse de que los mismos campos que se usan en `equals()` se consideren en `hashCode()`.
- Para objetos con campos de punto flotante, usar la misma lógica de comparación en ambos métodos (como se hizo en la implementación correcta de `Point`).
- Considerar el uso de `Objects.hash()` o `Objects.equals()` para implementaciones más seguras.
- Documentar claramente la lógica de igualdad y hash para futuros desarrolladores.

1.7 Ejercicios y Actividades Prácticas

Ejercicio 1: Implementación de una Tabla Hash Implementa una tabla hash que almacene números de teléfono (como String) y nombres de personas. Debes:

- Implementar la función hash para Strings
- Manejar colisiones usando encadenamiento
- Implementar las operaciones básicas: insertar, buscar y eliminar
- Probar tu implementación con diferentes casos

Ejercicio 2: Análisis de Funciones Hash Analiza las siguientes funciones hash y determina si son buenas o malas. Justifica tu respuesta: Sample code

Ejercicio 3: Implementación de equals() y hashCode() Implementa los métodos equals() y hashCode() para la siguiente clase: Sample code

Actividad Práctica: Análisis de Rendimiento Implementa un programa que compare el rendimiento de diferentes estrategias de manejo de colisiones:

Ejercicio 4: Diseño de Función Hash Diseña una función hash para la siguiente clase que represente coordenadas geográficas:

Actividad de Investigación Investiga y responde las siguientes preguntas:

1. ¿Por qué se usa el número 31 en muchas implementaciones de hashCode()?
2. ¿Cuál es la diferencia entre HashMap y TreeMap en Java?
3. ¿Cómo maneja Python las colisiones en sus diccionarios?
4. ¿Qué es el "hash flooding" y cómo se puede prevenir?

Ejercicio 5: Debugging Encuentra y corrige los errores en la siguiente implementación:

```
public class BuggyHashTable<K, V> {
    private Entry<K, V>[] table;
    private int size;

    public void put(K key, V value) {
        int index = key.hashCode() % table.length;
        table[index] = new Entry<>(key, value);
        size++;
    }

    public V get(K key) {
        int index = key.hashCode() % table.length;
        return table[index].value;
    }

    public void remove(K key) {
        int index = key.hashCode() % table.length;
        table[index] = null;
        size--;
    }
}
```

Listing 1: Implementación con errores

Solución de Problemas Resuelve los siguientes problemas:

1. Implementa una tabla hash que pueda almacenar múltiples valores para la misma clave.
2. Diseña una función hash para una cadena de ADN (secuencia de A, T, G, C).
3. Implementa una tabla hash que pueda redimensionarse automáticamente.

Bibliografía

[1] Aditya Bhargava. *Grokking Algorithms*. Manning Publications, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy*. CareerMonk Publications, 2011.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2005.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.