Algoritmus Et Structex
Compendium Maximus
UT6
25-05-2025

Tablas Hash

Una tabla hash permite inserción o eleminación de cualquier elemento nominado. Se desea que:

- Las operaciones básicas sean constantes

Función hash

Es una caja mágica que recibe un objeto y lo convierte en un índice para ser almacenado.

- Es **determinista**: Siempre que le pases el mismo objeto, va a devolver el mismo resultado.
- **Una vía sola**: Recibe un key y devuelve un índice

Ejemplo de Función hash que recibe un String y en base a la suma de sus caracteres ASCII genera un índice, aplica **mod dataMap.length** para mantener los valores dentro del rango:

```
 1  private int hash(String key) {
 2          int hash = 0;
 3          char[] chars = key.toCharArray(); // convierte el string en un
 4                                            // array de chars individuales
 5
 6          // recorre cada carácter del array
 7          for (int i = 0; i < chars.length; i++) {
 8              int asciiValue = chars[i]; // obtiene el ASCII del carácter
 9              hash = (hash + asciiValue * 23) % dataMap.length; // cálculo
mágico
10          }
11          return hash; // retorna el índice
12      }
```

Mejor Implementación → Usa el método polinomial para evitar desbordamientos y distribuir mejor las claves. Se multiplica por 37 para reducir el desplazamiento de caracteres iniciales.

- Se aplica mod tableSize al final para mantener los valores manejables.
- Se ajustan resultados negativos para garantizar índices válidos.

```
 1 public static int hash(String key, int tableSize)
 2 {
 3   int hashVal = 0;
 4
 5   for (int i = 0; i < key.length(); i++)
 6     hashVal = 37 * hashVal + key.charAt(i):}
 7   hashVal %= tableSize;
 8   if (hashVal < 0)
 9     hashVal += tableSize;
10
11   return hashVal;
12 }
```

Hash code (Java)
El método hashCode() en Java devuelve un valor entero (hash) que representa un objeto. Es fundamental para estructuras basadas en hash como HashMap, HashSet y Hashtable, ya que determina dónde se almacenará el objeto en la tabla hash

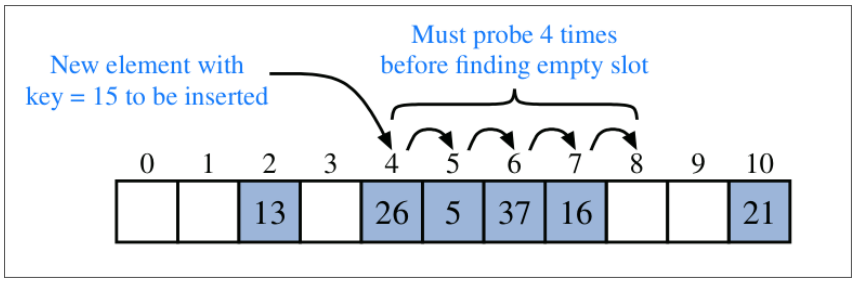
Colisiones

Ahora, tenemos un problemita → El uso de una Función Hash puede ocurrir que dos elementos se correspondan a un mismo índice. Esto se conoce como **colisión**. Hay varias formas de manejarlas:

- Sondeo lineal (lineal probing)
- Sondeo cuadrático (algo probing)
- Encadenamiento separado (Separate Chaining)

Sondeo lineal (Linear Probing)

Cuando dos claves producen el mismo índice (colisión), el sondeo lineal busca la siguiente celda disponible de manera secuencial (en orden lineal) hasta encontrar un espacio vacío. → $(\text{hash}(\text{key}) + i) \bmod N$



¿Cómo funciona?

1. **Inserción:**
   - Se calcula el índice inicial usando la función hash: índice = hash(clave) % tamaño_tabla.
   - Si la celda está ocupada, se avanza secuencialmente (índice +1, +2, etc.) hasta encontrar una celda vacía.
   - Si se llega al final de la tabla, se vuelve al inicio (comportamiento circular).
2. **Búsqueda (find):**
   - Se sigue el mismo camino que en la inserción.
   - Si se encuentra una celda vacía durante la búsqueda, la clave no existe.
   - Si se encuentra la clave, se devuelve su valor.
3. **Borrado:**
   - No se puede eliminar físicamente una clave, ya que rompería la secuencia de sondeo para otras claves.
   - Se usa borrado perezoso (lazy deletion): marcar la celda como "borrada" (sin liberarla).
   - Las inserciones pueden reutilizar celdas marcadas como borradas.
   - Las búsquedas ignoran las celdas borradas y continúan sondeando.

$$\textbf{Factor de carga}(\lambda) = \frac{\text{número de celdas ocupadas}}{\text{tamaño total de la tabla}}$$

Se recomienda mantener $\lambda < 0.7$ para evitar demasiadas colisiones. Si λ crece, redimensionar la tabla (aumentar su tamaño y rehashear las claves).

El rendimiento de una tabla hash depende de qué tan llena está.

→ La mayor desventaja del sondeo lineal es el **Agrupamiento primario:** La formación de grandes agrupaciones de celdas ocupadas que hacen que las inserciones tarden más tiempo.

Operación find

aburrido

Sondeo cuadrático (Quadratic Probing)

Elimina el temita con el agrupamiento primario. Examina las celdas que están situadas a una distancia de 1, 4, 9, etc. del punto de sondeo original. En lugar de buscar celdas secuencialmente, salta en pasos cuadráticos (1^2,2^2,3^2...) desde el índice original. $\rightarrow (\text{hash}(\text{key}) + i^2) \bmod N$

- Que el **tamaño de la tabla sea primo**:
  - Asegura que todas las celdas se sondearán **sin repeticiones** antes de agotar el espacio.
  - Ejemplo: Si M=7 (primo), las secuencias de sondeo cubren todas las posiciones posibles.
- **Factor de carga $\lambda \leq 0.5$:**
  - Si $\lambda > 0.5$, aumenta el riesgo de no encontrar celdas vacías.
  - Solución: Redimensionar la tabla (duplicar tamaño y usar un primo mayor) mediante rehashing.

"Si el tamaño de la tabla es primo y el factor de carga no es mayor a 0.5, todos los sondeos se realizarán en posiciones distintas y siempre se podrá insertar un elemento."

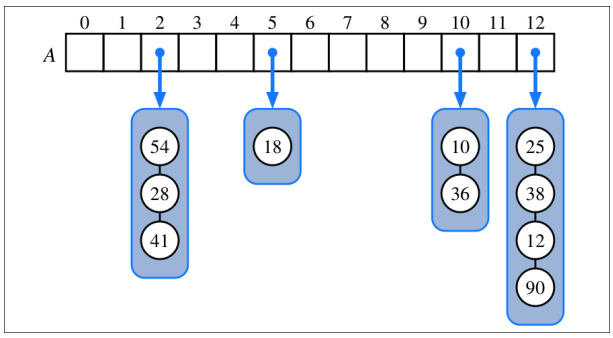
Implementación en Java

Creo que podría omitir esta bobada

Encadenamiento separado (Separate Chaining)

Una alternativa eficiente funciona de manera que mantiene una matriz de listas enlazadas. El peor caso de inserción es $O(1)$. La función hash nos dice en qué lista hay que insertar un elemento X.

- La tabla hash es un arreglo de listas enlazadas.
- Cada celda de la tabla contiene una lista de elementos con el mismo hash.



Ventajas:

- **Flexibilidad para manejar factores de carga altos**
  - El rendimiento no se degrada abruptamente cuando la tabla se llena. Las operaciones de inserción, búsqueda y borrado dependen del tamaño promedio de las listas enlazadas, con un coste esperado de $O(1 + \lambda)$, donde λ es el factor de carga.

Desventajas:

- **Overhead de memoria** debido al almacenamiento de punteros adicionales en los nodos de las listas, y su rendimiento puede verse afectado por la dispersión en el acceso a memoria.

Comparación con árboles de búsqueda binaria

- En una tabla hash no podemos encontrar el elemento mínimo de manera eficiente.
- No se puede buscar eficientemente una cadena a menos que conozcamos todos sus caracteres.

"Utilice la tabla hash en lugar de un ABB si no necesita estadísticas de orden y le preocupa la posible existencia de entradas no abstractas" (Weiss)