30 DE OCTUBRE DE 2025

TALLER DE ALGORITMOS Y ESTRUCTURAS DE DATO S II

TRABAJO PRÁCTICO 2

VALENTIN RAGUSA
UES21

Alumno: Valentin Ragusa
Legajo: VINF016492
Fecha: 30/10/2025

---

1. Introducción teórica

Las tablas hash constituyen una de las estructuras de datos más eficientes para el almacenamiento y recuperación de información. Su principal ventaja radica en que permiten acceder a los elementos en un tiempo promedio cercano a **O(1)**, es decir, constante, independientemente del tamaño de colección de datos.

El principio fundamental detrás de una tabla hash es la utilización de una **función hash**, la cual transforma una clave (key) en un valor numérico (hash-value) que actúa como índice dentro de una tabla. De esta manera, la función hash determina la posición donde se almacenará o buscara el elemento asociado a la clave. En este trabajo se implementó la función $\boldsymbol{H(x) = x \bmod 10}$, siguiendo la consigna del canvas, la cual distribuye los valores en base al resto de la división de la clave por el tamaño de la tabla.

El trabajo tiene como objetivo analizar, implementar y evaluar diferentes métodos de resolución de colisiones dentro de una tabla hash, comprendiendo su funcionamiento interno y su impacto en el rendimiento.

2. Desarrollo y resolución

Para la resolución del trabajo se implementó una estructura modular basada en POO, donde cada clase cumple una función específica dentro del sistema de almacenamiento hash.

2.1. Interfaz HashTable<K, V>

Define las operaciones fundamentales que debe cumplir cualquier tabla hash. Establece los métodos de inserción (put), búsqueda (get), eliminación (remove), verificar la existencia (containsKey), limpieza (clear) y consulta del tamaño (size, isEmpty).

2.2. Clase abstracta AbstractHashTable<K, V>

Actua como clase base para las implementaciones concretas de la interfaz HashTable<K, V>. Proporciona los atributos y comportamientos comunes a todas las tablas hash:

- Capacity: capacidad actual del array de almacenamiento
- Size: cantidad de elementos almacenados.
- loadFactorThreshold: factor de carga límite (0.75 por defecto), que determina cuando debe redimensionarse la tabla.

Incluye además los métodos auxiliares:

- hash(K key), que delega el cálculo de índices a la clase HashFunctions.
- ensureCapacity(), que controla el crecimiento de la tabla y ejecuta el método abstracto rehash() cuando es necesario.

2.3. Clase Entry<K, V>

Representa el par **clave-valor** almacenado en la tabla.
Cada objeto Entry contiene:

- La clave (key),
- El valor asociado (value),
- Y un indicador lógico (deleted) que permite marcar una posición como eliminada sin perder la secuencia de búsqueda mediante un *boolean*.

Esta estrategia de "borrado lógico" evita la ruptura de cadenas de sondeo en las implementaciones con direccionamiento abierto.
La clase incluye métodos getter y setter, y redefine toString() para representar su contenido de manera legible.

2.4. Clase LinearProbingHashTable<K, V>

Implementa la tabla hash utilizando el método de sondeo lineal como estrategia para resolver colisiones.
Cuando las dos claves general el mismo índice, la nueva se ubica en la **siguiente posición disponible**, recorriendo el arreglo de manera secuencial.

Sus principales métodos son:

- probe(int hash, int i) que calcula el desplazamiento lineal (hash +i) % capacity.
- put(K key, V value), que inserta o actualiza elementos, gestionando rehash cuando se supera el factor de carga.
- get(K key), recupera el valor correspondiente a la clave.
- remove(K key), hace el borrado lógico del elemento.

El sondeo lineal es simple de implementar pero puede producir clustering primario, agrupamiento de claves contiguas, afectando el rendimiento en grandes volúmenes de datos.

2.5. Clase QuadraticProbingHashTable<K, V>

En cuanto a código, es una variante exactamente igual a la clase LinearProbingHashTable<K, V>, solo que a diferencia de esta, en el método de sondeo cuadrático dispersamos mejor las colisiones mediante saltos crecientes según el cuadrado de la iteración $(hash + i^2)\% \, capacity$.

Esta estrategia disminuye significativamente el clustering primario, mejorando la distribución de elementos dentro del arreglo.

2.6. Clase HashFunctions

Contiene las funciones hash utilizadas en el proyecto.
La principal es hMod(int x, int m), que implementa la función $H(x) = x \ mod \ m$ solicitada en la consigna, devolviendo un valor de índice comprendido entre 0 y $m - 1$. Se incluye además una versión específica para hMod10(int x) que aplica el módulo 10 directamente.

3. Variables utilizadas

Variable	Tipo	Clase donde se utiliza	Descripción y función dentro del programa
capacity	int	AbstractHashTable	Representa la capacidad actual del array interno donde se almacenan las entradas de la tabla hash. Se utiliza para calcular el índice de inserción y determinar cuándo es necesario redimensionar la tabla.
size	Int	AbstractHashTable	Indica la cantidad de elementos efectivamente almacenados en la tabla. Se actualiza en cada inserción o eliminación.
loadFactorThreshold	double	AbstractHashTable	Define un límite máximo del factor de carga antes de ejecutar un **rehash**. Su valor por defecto es 0.75.
tabla	Entry<K, V>[]	LinearProbingHashTable/QuadraticProbingHashTable	Arreglo principal donde se almacenan los pares clave-valor.
key	K	Entry	Clave asociada al valor dentro de cada entrada. Se utiliza para calcular el hash y realizar búsquedas o comparaciones.
value	V	Entry	Valor almacenado correspondiente a la clave. Puede ser cualquier tipo genérico.
deleted	boolean	Entry	Indica si la entrada fue eliminada lógicamente. Permite mantener la secuencia de búsqueda sin romper la cadena de sondeo.
hash	int	LinearProbingHashTable/QuadraticProbingHashTable	Valor numerico calculado a partir de la clave mediante la función hash.

			Determina la posición inicial del sondeo
index	int	LinearProbingHashTable/QuadraticProbingHashTable	Indice actual evaluado durante el proceso de inserción, búsqueda o eliminación. Se modifica en cada iteración del sondeo.
firstDeletedIndex	int	QuadraticProbingHashTable	Almacena la primera posición marcada como eliminada encontrada durante el sondeo cuadrático, para poder reutilizarla si la clave no existe.
tablaVieja	Entry<K, V>	LinearProbingHashTable/QuadraticProbingHashTable	Referencia temporal utilizada durante el proceso de **rehash** para volver a insertar las entradas válidas en la nueva tabla.
r	int	HashFunctions	Resto resultante de aplicar la operación módulo sobre la clave, se utiliza para asegurar que el hash sea siempre positivo.

4. Resultados obtenidos

Durante las pruebas, se ejecutaron distintos casos de inserción, eliminación y redimensionamiento para ambas implementaciones de pruebas lineales y cuadráticas.

Al iniciar las pruebas con una capacidad base de cinco posiciones, se verificó que tanto el sondeo lineal como el cuadrático resolvían bien las colisiones generadas por claves con índices equivalentes (por ejemplo 1 y 11). Sin embargo, se evidenció que el **sondeo lineal tiende a agrupar las claves colisionadas de manera secuencial**, provocando la formación de zonas densas o clusters primarios. Este efecto incrementa la cantidad de pasos necesarios en operaciones de búsqueda y reduce la eficiencia conforme crece la tabla.

Por otra parte, la versión **cuadrática mostró una mejor dispersión de las claves**, evitando la concentración de elementos consecutivos. Gracias a los saltos crecientes (1, 4, 9, 16, …), las colisiones se resolvieron en posiciones más alejadas entre sí. Este comportamiento se traduce en una búsqueda más eficiente bajo altos factores de carga.

En ambos casos, el método ensureCapacity() activó el proceso de **rehash** cuando el factor de carga superó el 75%. Durante el proceso, la capacidad de la tabla se duplicó y las entradas válidas fueron reinsertadas mediante la función hash, garantizando una redistribución bien implementada.

5. Diagrama de clases UML

A continuación dejo adjunto el diagrama de clases UML que hice para evidenciar mi enfoque hacia el problema y cómo lo resolví partiendo de su base;