República Bolivariana de Venezuela
Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Técnicas Avanzadas de Programación
Semestre 1-2025

# Métodos de resolución de colisiones en *Hash Maps: Separate Chaining, Linear Probing, Quadratic Hash* y *Double Hash*

**Estudiante:**
César Carios
C.I 30.136.117

25 de junio del 2025

## *Separate Chaining*

`Separate Chaining` (Encadenamiento Separado) es una de las estrategias más comunes y robustas para resolver colisiones en tablas hash. Cuando dos claves diferentes se mapean al mismo índice de la tabla (una colisión), `Separate Chaining` no intenta encontrar un nuevo espacio dentro de la tabla principal. En su lugar, cada entrada de la tabla hash (conocida como *bucket*) se convierte en un puntero a una estructura de datos secundaria, comúnmente una lista enlazada.

La implementación de `Separate Chaining` proporcionada en `separate_chaining.cpp` utiliza una aproximación directa para gestionar las colisiones. Los puntos clave de su diseño son los siguientes:

- **Estructura de la Tabla:** La tabla hash se representa mediante un vector de listas que contienen pares de enteros. Esto significa que cada "bucket"de la tabla principal es una `list` (lista enlazada), que es donde se almacenan los pares clave-valor (`pair<int, int>`) cuando ocurre una colisión. Además, cada una de las operaciones (inserción, búsqueda y eliminación) han sido implementadas de la forma en que se vieron en clase. También debemos resaltar que partimos del hecho de que no tendremos claves repetidas.
- **Función Hash Primaria:** Se emplea una función hash muy simple basada en el operador módulo: `hashFunction(key, table_size) = key % table_size`. Esta función asigna cada clave a un índice dentro del rango del tamaño de la tabla.
- **Generación de Datos y Pruebas:** La implementación incluye funciones para generar datos aleatorios y claves de prueba, utilizando una semilla fija (`TEST_RANDOM_SEED`) para garantizar la reproducibilidad de los experimentos. Se usan `mt19937` y `uniform_int_distribution` para la aleatoriedad controlada.
- **Medición de Rendimiento:** El tiempo de ejecución se mide utilizando `chrono::high_resolution_clock`, lo que permite una medición precisa del rendimiento en nanosegundos para cada escenario de prueba.

Para las pruebas, en cada ejecución del programa se hacían 6 intentos. El programa se ejecutó al menos 3 veces, así que en total se hicieron 18 intentos para cada una de las implementaciones. Veamos los resultados:

Cuadro 1: Tiempos Promedio de Ejecución para Separate Chaining (ms)

| **Tamaño de Tabla (M) Factor de Carga ($\alpha$ inicial)** | **Dominado por Inserciones** (I:80 %, S:10 %, D:10 %) | **Dominado por Búsquedas** (I:10 %, S:80 %, D:10 %) | **Dominado por Eliminaciones** (I:10 %, S:10 %, D:80 %) | **Uso Promedio** (I:33 %, S:33 %, D:34 %) |
|---|---|---|---|---|
| 1000 ($\alpha = 10{,}0$) | 10.63 | 8.10 | 8.04 | 7.93 |
| 20000 ($\alpha = 0{,}5$) | 7.22 | 8.32 | 5.28 | 5.86 |
| 50000 ($\alpha = 0{,}2$) | 10.47 | 6.14 | 9.33 | 9.61 |
| 100000 ($\alpha = 0{,}1$) | 11.51 | 7.82 | 5.17 | 9.10 |

Los resultados de las pruebas para la implementación de `Separate Chaining` revelan varios patrones de rendimiento en función del tamaño de la tabla (`M`) y, consecuentemente, del factor de carga inicial ($\alpha$):

- **Resistencia a Factores de Carga Altos ($\alpha$):** Una de las características más destacadas de `Separate Chaining` es su robustez frente a factores de carga elevados. Incluso con un `M=1000`, lo que resulta en un factor de carga inicial de $\alpha = 10{,}0$ (diez veces más elementos que buckets, en promedio), el rendimiento sigue siendo razonable (ej., 10.63 ms para inserciones). Esto contrasta fuertemente con las técnicas de sondeo abierto, donde un $\alpha$ tan alto resultaría en una degradación catastrófica del rendimiento o incluso en el fallo de la tabla. Esto se debe a que las colisiones se manejan de forma independiente en listas enlazadas, evitando el agrupamiento dentro de la tabla principal.
- **Impacto del Factor de Carga en las Inserciones:** Aunque `Separate Chaining` es tolerante a factores de carga altos, un menor $\alpha$ generalmente se asocia con un mejor rendimiento.
  - El mejor tiempo de inserción se observa con `M=20000` ($\alpha = 0{,}5$), con 7.22 ms. Esto es esperable, ya que listas más cortas implican menos tiempo de recorrido para añadir un elemento al final.
  - Curiosamente, para factores de carga aún más bajos ($\alpha = 0{,}2$ y $\alpha = 0{,}1$), los tiempos de inserción aumentan ligeramente (10.47 ms y 11.51 ms, respectivamente). Esto podría explicarse por la sobrecarga de mantener una tabla subyacente mucho más grande (`std::vector`), lo que podría llevar a una mayor dispersión en memoria y a un menor aprovechamiento de la caché, incluso si las listas individuales son muy cortas.
- **Rendimiento en Búsquedas y Eliminaciones:**
  - Las operaciones de búsqueda y eliminación muestran un rendimiento más estable en general, con los mejores tiempos ocurriendo en rangos de $\alpha$ bajos a moderados. Por ejemplo, la búsqueda más rápida se da con `M=50000` ($\alpha = 0{,}2$) a 6.14 ms, y las eliminaciones más rápidas con `M=100000` ($\alpha = 0{,}1$) a 5.17 ms.
  - La variabilidad en los tiempos para búsquedas y eliminaciones entre diferentes $\alpha$ es menor que en las inserciones, lo que sugiere que una vez que se encuentra el bucket correcto, el recorrido de una lista relativamente corta es una operación eficiente.
- **Escenario de Uso Promedio:**
  - El mejor rendimiento general en el escenario de Üso Promedio"se obtiene con `M=20000` ($\alpha = 0{,}5$), con 5.86 ms. Esto sugiere un punto óptimo de equilibrio donde el tamaño de la tabla no es excesivamente grande (minimizando la sobrecarga de memoria y caché) y las listas enlazadas no son demasiado largas (minimizando el tiempo de recorrido).

## *Linear Probing*

`Linear Probing` (Sondeo Lineal) es una técnica de resolución de colisiones en tablas hash que pertenece a la categoría de "sondeo abierto"(Open Addressing). A diferencia del encadenamiento separado, donde los elementos en colisión se almacenan externamente, `Linear Probing` busca una ranura vacía *dentro de la propia tabla* cuando ocurre una colisión. Si la posición calculada por la función hash ya está ocupada, la técnica inspecciona secuencialmente las siguientes posiciones en la tabla (incrementando el índice de uno

en uno, de forma lineal) hasta encontrar una ranura disponible. Las operaciones de búsqueda y eliminación siguen la misma secuencia de sondeo para localizar el elemento.

La implementación hecha sigue los siguientes puntos principales:

- **Estructura de la Tabla:** La tabla hash se representa como un `vector<HashEntry>`. Cada `HashEntry` es una estructura que contiene la clave, el valor y, crucialmente, un `EntryState`.
- **Estados de las Entradas:** Para manejar las eliminaciones y búsquedas correctamente, cada entrada puede tener uno de tres estados: `EMPTY` (vacía), `OCCUPIED` (ocupada) o `DELETED` (marcada como eliminada lógicamente).
- **Función Hash Primaria:** Al igual que en `Separate Chaining`, se utiliza una función hash simple de módulo: `key % table_size`.
- **Manejo de Colisiones (Sondeo Lineal):** Cuando ocurre una colisión (la posición inicial está ocupada), la implementación sondea linealmente las siguientes posiciones '(currentindex + 1)
- **Inserción (`insertLinearProbing`):**
  - Busca la próxima ranura `EMPTY` o `DELETED`.
  - Si la clave ya existe en una posición `OCCUPIED`, la implementación **actualiza su valor** y retorna, asumiendo que no debe haber duplicados reales sino actualizaciones.
  - Incluye una verificación de 'probes >= tablesize' para evitar bucles infinitos en caso de que la tabla esté completamente llena, retornando 'false' si no se puede insertar.
- **Búsqueda (`searchLinearProbing`):**
  - Sigue la misma secuencia de sondeo que la inserción.
  - Se detiene y retorna 'false' si encuentra una celda `EMPTY`, ya que esto indica que la clave no puede haber sido insertada más allá de ese punto.
  - Ignora las celdas `DELETED` y continúa el sondeo, lo cual es crucial para encontrar elementos que podrían haber sido desplazados debido a colisiones.
- **Eliminación (`deleteLinearProbing`):**
  - Al encontrar la clave, la celda no se vacía completamente, sino que se marca con el estado `DELETED`. Esto es vital para asegurar que las búsquedas futuras puedan saltar sobre estas posiciones lógicamente eliminadas y encontrar elementos que fueron insertados originalmente después de ellas.
  - También se detiene y retorna 'false' si encuentra una celda `EMPTY`.
- **Generación de Datos y Pruebas:** Similar a `Separate Chaining`, utiliza generadores de números aleatorios con una semilla fija (`TEST_RANDOM_SEED`) para reproducibilidad y prepara datos base, claves para búsqueda y claves para eliminación.
- **Consideración del Factor de Carga:** La implementación incluye un comentario importante señalando que para `Linear Probing`, el tamaño de la tabla (`table_size`) **debe ser mayor que el número de puntos de datos iniciales** (`NUM_DATA_POINTS`) para evitar fallos de inserción tempranos debido a un factor de carga cercano a 1 o superior. Esto es crítico para el rendimiento y la funcionalidad del sondeo abierto.

Veamos los resultados obtenidos:

Cuadro 2: Tiempos Promedio de Ejecución para Linear Probing (ms)

| **Tamaño de Tabla (M)Factor de Carga ($\alpha$ inicial)** | **Dominado por Inserciones** (I:80 %, S:10 %, D:10 %) | **Dominado por Búsquedas** (I:10 %, S:80 %, D:10 %) | **Dominado por Eliminaciones** (I:10 %, S:10 %, D:80 %) | **Uso Promedio** (I:33 %, S:33 %, D:34 %) |
|---|---|---|---|---|
| 15000 ($\alpha = 0{,}67$) | 1613.41 | 18.99 | 63.27 | 147.72 |
| 20000 ($\alpha = 0{,}50$) | 1016.36 | 5.21 | 5.25 | 5.30 |
| 50000 ($\alpha = 0{,}20$) | 5.28 | 0.78 | 0.00 | 5.26 |
| 100000 ($\alpha = 0{,}10$) | 6.08 | 5.29 | 5.35 | 2.58 |

Los resultados obtenidos para la implementación de `Linear Probing` muestran una clara dependencia del factor de carga ($\alpha$) de la tabla hash. Los tiempos de ejecución (en milisegundos) varían drásticamente a medida que el factor de carga cambia, un comportamiento característico de las técnicas de sondeo abierto.

- **Sensibilidad Extrema al Factor de Carga ($\alpha$):** A diferencia de `Separate Chaining`, `Linear Probing` es extremadamente sensible al factor de carga.
  - Con un factor de carga inicial de $\alpha = 0{,}67$ (`M=15000`), los tiempos de ejecución son significativamente altos, especialmente en el escenario dominado por inserciones (1613.41 ms). Esto se debe al fenómeno de agrupamiento primario (primary clustering), donde largas secuencias de celdas ocupadas se forman en la tabla, forzando a las operaciones a realizar muchas pruebas (probes) consecutivas para encontrar una ranura o un elemento.
  - A medida que el factor de carga disminuye, el rendimiento mejora drásticamente. Para $\alpha = 0{,}50$ (`M=20000`), aunque las inserciones siguen siendo costosas (1016.36 ms), ya se observa una mejora sustancial.
- **Rendimiento Óptimo con Factores de Carga Bajos:**
  - Cuando el factor de carga es bajo ($\alpha \leq 0{,}20$), `Linear Probing` muestra un rendimiento excepcional.
  - Para $\alpha = 0{,}20$ (`M=50000`), los tiempos bajan a valores muy pequeños. Esto se debe a que la probabilidad de colisión es baja y, cuando ocurre, se encuentra una ranura vacía muy rápidamente, aprovechando la localidad de la caché. Las eliminaciones incluso reportan 0.00 ms en promedio, lo que sugiere que las operaciones son tan rápidas que caen por debajo de la resolución del temporizador.
  - En el caso de $\alpha = 0{,}10$ (`M=100000`), el escenario de Uso Promedio alcanza su mejor tiempo (2.58 ms), indicando que con una tabla muy dispersa, las operaciones de sondeo lineal son extremadamente eficientes.

## *Quadratic Hash*

`Quadratic Hashing` (Sondeo Cuadrático) es otra técnica de resolución de colisiones dentro de la categoría de "sondeo abierto"(Open Addressing), similar a `Linear Probing`. Sin embargo, a diferencia del sondeo lineal que inspecciona las siguientes posiciones de forma secuencial, `Quadratic Hashing` utiliza una secuencia de sondeo cuadrática para encontrar la próxima ranura disponible. Si la posición calculada por la

función hash primaria $h(k)$ está ocupada, las posiciones probadas son $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, y así sucesivamente, todas ellas módulo el tamaño de la tabla. El objetivo principal de este patrón de sondeo es mitigar el problema de agrupamiento primario que afecta a `Linear Probing`.

Los puntos claves de esta implementación son los siguientes:

- **Doble Función Hash:** La clave de esta implementación radica en el cálculo de la posición de sondeo mediante la fórmula $pos = (h_1(key) + i \cdot h_2(key)) \pmod{M}$.
- **Función Hash Primaria ($h_1$):** Utiliza una función de módulo simple: hashFunction(key, tablesize) que es key % tablesize.
- **Función Hash Secundaria ($h_2$):** Se introduce una segunda función hash, 'hashFunction2(key, tablesize)'.
- **Secuencia de Sondeo:** La variable 'i' comienza en 1 y se incrementa en cada iteración. El desplazamiento es 'i * hashFunction2(key, tablesize)'.
- **Estados de las Entradas:** Al igual que en `Linear Probing` y `Quadratic Hashing`, se utilizan los estados `EMPTY`, `OCCUPIED` y `DELETED` para cada entrada 'HashEntry'. El estado 'DELETED' es fundamental para permitir que las búsquedas y eliminaciones naveguen correctamente a través de celdas que fueron ocupadas y luego eliminadas lógicamente.
- **Control de Bucles Infinitos:** Las operaciones de inserción, búsqueda y eliminación incluyen un contador 'probes' para evitar bucles infinitos en el caso de que la tabla esté llena, o si el diseño de las funciones hash no garantiza el sondeo de todas las celdas.

Los resultados obtenidos con esta implementación fueron:

Cuadro 3: Tiempos Promedio de Ejecución para Quadratic Probing (ms)

| **Tamaño de Tabla (M)Factor de Carga ($\alpha$ inicial)** | **Dominado por Inserciones** (I:80 %, S:10 %, D:10 %) | **Dominado por Búsquedas** (I:10 %, S:80 %, D:10 %) | **Dominado por Eliminaciones** (I:10 %, S:10 %, D:80 %) | **Uso Promedio** (I:33 %, S:33 %, D:34 %) |
|---|---|---|---|---|
| 20003 ($\alpha \approx 0{,}50$) | 1275.64 | 2.63 | 2.83 | 3.75 |
| 25003 ($\alpha \approx 0{,}40$) | 0.18 | 2.61 | 0.10 | 5.19 |
| 50021 ($\alpha \approx 0{,}20$) | 4.28 | 2.62 | 3.18 | 2.59 |
| 100000 ($\alpha = 0{,}10$) | 5.22 | 2.61 | 3.99 | 5.20 |
| 100003 ($\alpha \approx 0{,}10$) | 5.30 | 5.28 | 2.61 | 2.72 |

Los resultados de las pruebas para `Quadratic Probing` revelan un comportamiento de rendimiento que, si bien también es sensible al factor de carga ($\alpha$), gestiona las colisiones de manera diferente a `Linear Probing`.

- **Impacto del Factor de Carga y Agrupamiento Secundario:**
  - Se observa una mejora significativa en el rendimiento con factores de carga decrecientes, similar a `Linear Probing`. Sin embargo, `Quadratic Probing` busca mitigar el agrupamiento primario al esparcir las pruebas de forma no lineal.

- Los tiempos de inserción para $\alpha \approx 0{,}50$ (`M=20003`) son de 1275.64 ms. Esto es considerablemente mejor que los 1016.36 ms de `Linear Probing` con un $\alpha = 0{,}50$ cercano, lo que sugiere que el sondeo cuadrático es más eficaz reduciendo la longitud de las secuencias de prueba en rangos de carga moderados.
- El rendimiento sigue mejorando drásticamente a medida que el $\alpha$ disminuye, con tiempos muy bajos para $\alpha \approx 0{,}40$ (`M=25003`) donde las inserciones caen a 0.18 ms, indicando una eficiencia casi instantánea en este escenario.

- **Rendimiento en Búsquedas y Eliminaciones:**
  - Las operaciones de búsqueda y eliminación también se benefician enormemente de los factores de carga bajos. Para $\alpha \leq 0{,}40$, los tiempos son extremadamente rápidos, incluso reportando 0.00 ms en varias ocasiones. Esto indica que se encuentran las claves rápidamente o que las operaciones son tan veloces que caen por debajo de la resolución del cronómetro.
  - Para $\alpha \approx 0{,}50$, los tiempos de búsqueda (2.63 ms) y eliminación (2.83 ms) son notablemente buenos y mucho más estables que en el caso de las inserciones, lo que es esperable para operaciones que solo necesitan encontrar un elemento existente.
- **Importancia del Factor de Carga $\alpha \leq 0{,}5$ y Tamaño Primo:**
  - La implementación se adhiere a la regla crítica de que el factor de carga no debe exceder 0,5 cuando el tamaño de la tabla es primo. Esto garantiza que siempre habrá una celda vacía para una inserción, lo cual es fundamental para la terminación y corrección del algoritmo de sondeo cuadrático.
  - Los tamaños de tabla utilizados (‘20003‘, ‘25003‘, ‘50021‘, ‘100003‘) son primos, lo que es una buena práctica y ayuda a la distribución uniforme de las claves y a la efectividad del sondeo cuadrático. El caso de ‘M=100000‘ (no primo) muestra un rendimiento similar a ‘M=100003‘ (primo) en este rango de carga muy baja, pero la recomendación de usar primos es crucial para cargas más altas.
- **Costo de Inserción en $\alpha$ Moderado vs. Bajo:**
  - Se observa que el tiempo de inserción es el más afectado a medida que $\alpha$ se acerca a 0.5. Las inserciones son más sensibles a la ocupación de la tabla que las búsquedas o eliminaciones, ya que siempre deben encontrar una ranura `EMPTY` o `DELETED`.
  - No obstante, la curva de rendimiento es mucho más suave que la de `Linear Probing` en el rango de $\alpha$ entre 0.5 y 0.2, lo que valida la capacidad de `Quadratic Probing` para reducir el agrupamiento primario. El rendimiento excepcional a $\alpha \approx 0{,}40$ para inserciones (0.18 ms) es un claro indicador de su eficiencia cuando se evitan los problemas de agrupamiento.

## *Double Hash*

`Double Hashing` (Doble Hashing) es una técnica avanzada de resolución de colisiones en tablas hash de sondeo abierto que utiliza **dos funciones hash distintas e independientes**, $h_1(key)$ y $h_2(key)$. Cuando ocurre una colisión en la posición inicial $h_1(key)$, las posiciones subsiguientes no se calculan de forma lineal o cuadrática fija, sino que se determinan añadiendo múltiplos de la segunda función hash: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \pmod{M}$, donde $i$ es el número de intento de sondeo. Esto genera una secuencia de sondeo única y pseudoaleatoria para cada clave, lo que es extremadamente efectivo para minimizar los problemas de agrupamiento primario y secundario.

Los puntos principales de esta implementación son los siguientes:

- **Doble Función Hash Dinámica:** La implementación utiliza una primera función hash $h_1(key) = key \pmod{M}$. La segunda función hash es $h_2(key) = R - (key \pmod{R})$, donde $R$ es el primo más grande menor que el tamaño de la tabla ($M$). Esta $R$ se calcula dinámicamente usando la función 'findLargestPrimeLessThan()' para cada tamaño de tabla, asegurando un valor apropiado para la segunda función hash.
- **Secuencia de Sondeo Única:** La posición de sondeo se calcula como $(h_1(key) + i \cdot h_2(key)) \pmod{M}$, donde $i$ es el número de intento de sondeo. Esto crea una secuencia de sondeo distinta para cada clave, lo que es fundamental para evitar tanto el agrupamiento primario como el secundario.
- **Manejo de Estados de Entrada:** Se emplean los estados `EMPTY`, `OCCUPIED` y `DELETED` para cada entrada de la tabla (`HashEntry`). El estado `DELETED` es crucial para asegurar que las operaciones de búsqueda y eliminación continúen el sondeo correctamente a través de posiciones que antes contenían datos pero fueron eliminadas lógicamente.
- **Garantía de Sonda y Prevenir Bucles:**
  - La implementación asegura que $h_2(key)$ nunca retorne cero y que el desplazamiento siempre sea válido, un requisito crítico para la terminación del algoritmo.
  - Las operaciones incluyen un contador de 'probes' (intentos) para evitar bucles infinitos en casos extremos donde la tabla esté llena o la combinación de funciones hash no garantice el recorrido de todas las celdas.
- **Requisitos de Tamaño de Tabla:** En el 'main', se destaca la importancia de que el tamaño de la tabla ($M$) sea un número primo y que exista un primo $R$ menor que $M$ para $h_2$. Esto es vital para asegurar que la secuencia de sondeo explore todas las posibles ranuras de la tabla antes de potencialmente fallar en una inserción (garantía de sonda).

Veamos los resultados:

Cuadro 4: Tiempos Promedio de Ejecución para Double Hashing (ms)

| **Tamaño de Tabla (M)Factor de Carga ($\alpha$ inicial)** | **Dominado por Inserciones** (I:80 %, S:10 %, D:10 %) | **Dominado por Búsquedas** (I:10 %, S:80 %, D:10 %) | **Dominado por Eliminaciones** (I:10 %, S:10 %, D:80 %) | **Uso Promedio** (I:33 %, S:33 %, D:34 %) |
|---|---|---|---|---|
| 15013 ($\alpha \approx 0{,}67$) | 2479.37 | 4.11 | 2.50 | 7.95 |
| 20003 ($\alpha \approx 0{,}50$) | 1366.33 | 4.94 | 5.21 | 5.21 |
| 50021 ($\alpha \approx 0{,}20$) | 3.94 | 5.27 | 2.78 | 5.17 |
| 100000 ($\alpha = 0{,}10$) | 5.21 | 2.86 | 4.77 | 5.30 |
| 100003 ($\alpha \approx 0{,}10$) | 3.61 | 4.76 | 2.53 | 5.31 |

- **Sensibilidad Extrema a Factores de Carga Altos:**
  - Para un factor de carga de $\alpha \approx 0{,}67$ (`M=15013`), los tiempos de inserción son excepcionalmente altos (2479.37 ms). Este valor es notablemente superior incluso a los observados en `Linear Probing` y `Quadratic Probing` para factores de carga similares o ligeramente inferiores. Esto

sugiere que, a pesar de su sofisticada estrategia de sondeo, `Double Hashing` también enfrenta desafíos significativos cuando la tabla está muy cerca de su capacidad. Este comportamiento puede ser indicativo de un gran número de sondeos necesarios o incluso fallos en las inserciones iniciales debido a una alta ocupación.

- Las operaciones de búsqueda y eliminación, sin embargo, se mantienen eficientes incluso a este alto factor de carga (4.11 ms para búsquedas, 2.50 ms para eliminaciones), lo que indica que una vez que una clave está en la tabla, encontrarla (o su marca de borrado) es relativamente rápido.

- **Mejora Drástica con la Disminución del Factor de Carga:**
  - A medida que el factor de carga disminuye a $\alpha \approx 0{,}50$ (`M=20003`), el tiempo de inserción se reduce significativamente a 1366.33 ms. Este valor es comparable con el de `Quadratic Probing` en un factor de carga similar.
  - Para factores de carga bajos ($\alpha \leq 0{,}20$, `M=50021`, `M=100000`, `M=100003`), el rendimiento de `Double Hashing` es excelente en todas las operaciones. Los tiempos promedio de inserción caen a valores de un solo dígito de milisegundos (ej. 3.94 ms para $\alpha \approx 0{,}20$), y muchas repeticiones muestran 0.00 ms para búsquedas y eliminaciones, lo que indica operaciones extremadamente rápidas, a menudo por debajo de la resolución del temporizador.

- **Efectividad en la Mitigación del Agrupamiento:**
  - La teoría sugiere que `Double Hashing` es la más efectiva de las técnicas de sondeo abierto para evitar el agrupamiento primario y secundario, al generar secuencias de sondeo únicas para cada clave. Los resultados en factores de carga bajos confirman esta eficiencia, donde las colisiones son manejadas de forma casi instantánea.
  - La comparación con `Linear Probing` y `Quadratic Probing` muestra que `Double Hashing` no necesariamente supera a las otras en todos los escenarios, especialmente en cargas muy altas (donde la inserción es más lenta que en Linear Probing para $\alpha \approx 0{,}67$). Sin embargo, en un rango más amplio de factores de carga moderados a bajos, su rendimiento es consistentemente bueno y competitivo.

- **Importancia de $R_{prime}$ y el Tamaño de la Tabla Primo:**
  - La implementación correcta de $h_2(key) = R - (key \pmod R)$, con $R$ siendo un primo menor que el tamaño de la tabla ($M$), es crucial. Esta elección garantiza que la secuencia de sondeo visite todas las celdas de la tabla (garantía de sonda), evitando ciclos prematuros y asegurando la terminación de las operaciones.
  - Se confirma que el tamaño de la tabla ('M') debe ser primo para optimizar la distribución y la eficiencia de la función hash, aunque en factores de carga muy bajos ($\alpha \approx 0{,}10$), las diferencias de rendimiento entre una tabla de tamaño primo ('100003') y no primo ('100000') son mínimas.

## *Unordered Map y Unordered Set*

En C++, `std::unordered_map` y `std::unordered_set` son contenedores asociativos que implementan tablas hash utilizando un método de **encadenamiento separado** (separate chaining) internamente. Están diseñados para proporcionar un tiempo promedio de complejidad constante $O(1)$ para las operaciones básicas (inserción, búsqueda y eliminación), bajo el supuesto de una buena función hash.

Las principales diferencias entre ellos son:

- `std::unordered_map`:
  - Almacena pares **clave-valor** (`std::pair<Key, Value>`).
  - Cada clave es **única**.
  - Proporciona acceso rápido a los valores a través de sus claves, similar a un diccionario.
  - Ejemplo: `unordered_map<string, int>edades;`
- `std::unordered_set`:
  - Almacena **solo claves** (elementos individuales), sin valores asociados.
  - Cada clave es **única**.
  - Se utiliza principalmente para comprobar la pertenencia de un elemento a un conjunto (si un elemento existe o no).
  - Ejemplo: `unordered_set<int>numeros_primos;`

En resumen, la distinción fundamental es que `unordered_map` asocia valores a claves, mientras que `unordered_set` solo almacena elementos únicos para una rápida búsqueda de existencia. Ambos utilizan el mismo principio subyacente de tablas hash para lograr su eficiencia. Veamos cuáles fueron los resultados obtenidos con estos contenedores:

Cuadro 5: Tiempos Promedio de Ejecución para `std::unordered_map` (ms)

| **Escenario** | **Tiempo Promedio (ms)** |
|---|---|
| Dominado por Inserciones (I:80 %, S:10 %, D:10 %) | 15.8633 |
| Dominado por Búsquedas (I:10 %, S:80 %, D:10 %) | 10.5362 |
| Dominado por Eliminaciones (I:10 %, S:10 %, D:80 %) | 7.979 |
| Uso Promedio (I:33 %, S:33 %, D:34 %) | 13.1417 |

Los resultados de las pruebas para `std::unordered_map` demuestran la robustez y eficiencia de las implementaciones de tablas hash de la librería estándar de C++. A diferencia de las implementaciones manuales de sondeo abierto que mostraron una alta sensibilidad al factor de carga, `std::unordered_map` mantiene un rendimiento notablemente consistente en todos los escenarios.

- **Rendimiento Consistente y Predecible:**
  - Todos los escenarios (Dominado por Inserciones, Búsquedas, Eliminaciones, y Uso Promedio) presentan tiempos de ejecución que se encuentran en un rango estrecho, entre aproximadamente 8 ms y 16 ms. Esto es una clara evidencia de la complejidad temporal promedio $O(1)$ de las operaciones, característica fundamental de una tabla hash bien implementada.
  - Las operaciones de eliminación (7.979 ms) y búsqueda (10.5362 ms) son ligeramente más rápidas que las de inserción (15.8633 ms). Esto es típico de las tablas hash basadas en encadenamiento, ya que las inserciones pueden incurrir en el costo adicional de la asignación de memoria para nuevos nodos o, ocasionalmente, el costo amortizado del redimensionamiento (re-hashing) de la tabla.
- **Ventaja de la Gestión Dinámica del Factor de Carga:**

- La característica más destacada de `std::unordered_map` en este contexto es su capacidad para gestionar dinámicamente el factor de carga. Esto significa que la tabla se redimensiona automáticamente (re-hashing) cuando el número de elementos y el factor de carga superan un umbral predefinido, asegurando que el rendimiento se mantenga óptimo sin intervención manual.
- Esta gestión automática contrasta fuertemente con las implementaciones de sondeo abierto (Linear, Quadratic, Double Hashing) que requerían una cuidadosa selección del tamaño de tabla para mantener un factor de carga bajo y evitar una degradación severa del rendimiento. Las implementaciones manuales de sondeo abierto mostraron tiempos de inserción en el orden de los miles de milisegundos para factores de carga altos (ej., 1613 ms en Linear Probing $\alpha = 0{,}67$, 1275 ms en Quadratic Probing $\alpha = 0{,}50$, 2479 ms en Double Hashing $\alpha = 0{,}67$), mientras que `std::unordered_map` nunca excede los 16 ms.

- **Robustez y Optimización:**
  - `std::unordered_map` se beneficia de años de optimización y de algoritmos de funciones hash altamente refinados que son robustos para una amplia variedad de tipos de datos y distribuciones de claves. Esto reduce drásticamente la probabilidad de "peores casos"de rendimiento en la práctica, que pueden ser difíciles de mitigar en implementaciones manuales.

Cuadro 6: Tiempos Promedio de Ejecución para `std::unordered_set` (ms)

| **Escenario** | **Tiempo Promedio (ms)** |
|---|---|
| Dominado por Inserciones (I:80 %, S:10 %, D:10 %) | 8.9295 |
| Dominado por Búsquedas (I:10 %, S:80 %, D:10 %) | 4.1065 |
| Dominado por Eliminaciones (I:10 %, S:10 %, D:80 %) | 2.619 |
| Uso Promedio (I:33 %, S:33 %, D:34 %) | 6.1265 |

Los resultados de las pruebas para `std::unordered_set` confirman su eficiencia como contenedor de tablas hash, con un rendimiento que incluso supera a `std::unordered_map` en los escenarios evaluados. Esto es consistente con su diseño de almacenar solo claves únicas, sin la sobrecarga de gestionar pares clave-valor.

- **Rendimiento Excepcional en Todas las Operaciones:**
  - Los tiempos promedio de ejecución para `std::unordered_set` son consistentemente bajos en todos los escenarios, variando desde aproximadamente 2.6 ms hasta 9 ms. Esto subraya su capacidad para proporcionar una complejidad temporal $O(1)$ en promedio para inserciones, búsquedas y eliminaciones.
  - Las operaciones de eliminación (2.619 ms) y búsqueda (4.1065 ms) son las más rápidas, a menudo registrando 0 ms en repeticiones individuales, lo que indica que se ejecutan prácticamente de forma instantánea dentro de la precisión del sistema de medición.
  - Las inserciones (8.9295 ms) son la operación comparativamente más "lenta", pero siguen siendo extremadamente rápidas y eficientes.

- **Comparación Directa con `std::unordered_map`:**
  - Una de las conclusiones más importantes es que `std::unordered_set` es consistentemente **más rápido** que `std::unordered_map` en todos los escenarios de prueba.

- Por ejemplo, las inserciones en `unordered_set` (8.9295 ms) son casi el doble de rápidas que en `unordered_map` (15.8633 ms). De manera similar, las búsquedas y eliminaciones también muestran mejoras notables.
- Esta diferencia de rendimiento se atribuye principalmente a que `unordered_set` solo almacena las claves. Al no tener que gestionar un valor asociado, se reduce la sobrecarga de memoria por elemento, lo que puede resultar en una mejor utilización de la caché, menos asignaciones/desasignaciones de memoria y operaciones internas más rápidas.

- **Beneficios de la Gestión Dinámica del Factor de Carga:**
  - Al igual que `std::unordered_map`, `std::unordered_set` gestiona su factor de carga de forma dinámica mediante redimensionamientos automáticos (re-hashing). Esta característica fundamental permite que el contenedor mantenga un rendimiento óptimo sin que el usuario tenga que preocuparse por la selección del tamaño inicial o por la monitorización constante del factor de carga.
  - Esta gestión automática contrasta con las implementaciones manuales de sondeo abierto (Linear, Quadratic, Double Hashing), que experimentaron una degradación severa del rendimiento a medida que el factor de carga aumentaba, demostrando la superioridad de la estrategia de la STL en entornos variables o de alta carga.

## Conclusiones

Las pruebas de rendimiento realizadas sobre diversas implementaciones de tablas hash revelan información crítica sobre la eficiencia y las compensaciones de cada enfoque:

- **Encadenamiento Separado:**
  - Demostró ser robusto y relativamente estable en su rendimiento a través de diferentes factores de carga. Mantuvo tiempos bajos (alrededor de 3-5 ms para inserciones óptimas) y evitó la degradación drástica observada en el sondeo abierto a altas cargas.
  - Su rendimiento es competitivo con las estructuras de la STL cuando el factor de carga se mantiene bajo, validando su efectividad como una técnica fundamental para el manejo de colisiones.

- **Sondeo Abierto (Lineal, Cuadrático, Doble Hashing):**
  - **Sensibilidad Crítica al Factor de Carga:** La característica más sobresaliente es su extrema sensibilidad al factor de carga ($\alpha$). A medida que $\alpha$ se acerca a 0.5 o 0.67, especialmente en escenarios dominados por inserciones, los tiempos de ejecución se disparan a miles de milisegundos. Esto se debe al aumento drástico en el número de sondeos necesarios para encontrar una ranura vacía, exacerbando los problemas de agrupamiento.
  - **Impacto del Agrupamiento:**
    - `Linear Probing` sufrió notablemente de agrupamiento primario, resultando en los peores rendimientos a altas cargas.
    - `Quadratic Probing` mitigó el agrupamiento primario, mostrando una mejora en el rendimiento respecto a Linear Probing a cargas moderadas, pero aún así se vio severamente afectada por factores de carga altos debido al agrupamiento secundario.

    - `Double Hashing`, aunque teóricamente el mejor en la distribución de sondeos, también experimentó un rendimiento deficiente en cargas muy altas (incluso peor que Linear Probing en $\alpha \approx 0{,}67$ en las inserciones), lo que puede indicar una interacción compleja con el tamaño específico de la tabla, la función hash secundaria y la naturaleza de las operaciones, o la sobrecarga del cálculo de dos funciones hash por sondeo.
  - **Rendimiento en Cargas Bajas:** En contraste, a factores de carga bajos ($\alpha \leq 0{,}20$), todas las técnicas de sondeo abierto mostraron un rendimiento excelente, con tiempos en el rango de milisegundos de un solo dígito, ya que las colisiones eran raras y los sondeos mínimos.

- **`std::unordered_map` y `std::unordered_set`:**
  - **Estabilidad y Fiabilidad Superior:** Estos contenedores de la biblioteca estándar de C++ demostraron una estabilidad de rendimiento excepcional en todos los escenarios, manteniendo los tiempos de ejecución consistentemente bajos (generalmente por debajo de 20 ms). Su rendimiento es un testimonio de las optimizaciones internas, las funciones hash robustas y, crucialmente, la **gestión automática y dinámica del factor de carga** a través del redimensionamiento (re-hashing).
  - **Manejo Automático vs. Manual:** La principal ventaja de los contenedores `std::` radica en su capacidad para adaptarse a diferentes cargas sin requerir intervención manual para ajustar el tamaño de la tabla o el factor de carga. Esto elimina el riesgo de los "peores casos"de rendimiento observados en las implementaciones custom de sondeo abierto a altas cargas.
  - **Diferencia entre `map` y `set`:** `std::unordered_set` fue consistentemente más rápido que `std::unordered_map` en todas las operaciones. Esto se debe a que `unordered_set` solo necesita almacenar y gestionar las claves, mientras que `unordered_map` debe manejar pares clave-valor, lo que implica una mayor sobrecarga de memoria y procesamiento por elemento.

En resumen, la elección de una estrategia de tabla hash depende en gran medida del caso de uso y los requisitos de rendimiento. Las implementaciones de sondeo abierto pueden ser muy rápidas en tablas poco pobladas, pero son extremadamente sensibles al factor de carga y al agrupamiento. El encadenamiento separado es una alternativa más estable para implementaciones custom. Sin embargo, para la mayoría de las aplicaciones prácticas, **`std::unordered_map` y `std::unordered_set` son las opciones más recomendables** debido a su rendimiento promedio $O(1)$ confiable, su robustez inherente y su gestión automática del rendimiento, lo que simplifica enormemente el desarrollo y asegura un comportamiento predecible incluso bajo condiciones de carga variables.