Análisis de rendimiento de Tablas Hash: *Separate Chaining* vs. *Open Addresing*

Alvarado Contreras Rebeca.
Universidad Autónoma de San Luis Potosí, Facultad de Ciencias.
Ingeniería Física, Cómputo de alto rendimiento.
Proyecto 1: Tablas Hash.

***Resumen*—Se hizo una análisis de rendimiento (tiempo y memoria empleada) para la comparación de las diferentes implementaciones de Tablas Hash: *separate chaining* y *open addressing*. En esta última, *open addressing*, se comparan los diferentes tipos de sondeo: lineal, cuadrático y double hash. También se revisa la expansión dinámica de la estructura para mejorar su rendimiento.**

***Palabras clave*—Tablas Hash, Separate chaining, Open addressing, probing.**

I. Introducción

Se entiende como *hashing*, un par de llave - valor, donde la llave se asocia a la posición en un arreglo donde el valor correspondiente está almacenado, mediante operaciones aritméricas para transformar la llave a la posición, esto es, una función hash. [1]

Los algoritmos de búsqueda que emplean *hashing* consisten en dos partes. La primera de ellas es calcular una función hash. Idealmente, diferentes llaves se mapearían a diferentes posiciones, pero esto no siempre es cierto; por lo que se debe de considerar las colisiones, es decir, la situación donde dos llaves diferentes se mapean a una misma posición. La segunda parte consiste en el proceso de resolución de colisiones. [1]

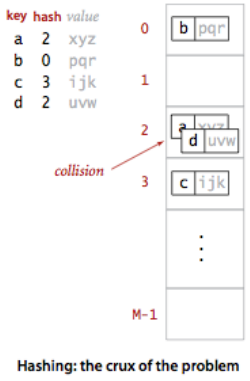

Figura 1: Consideraciones al implementar una Tablas Hash.

En el trabajo presente, se exploran las diferentes implementaciones de tablas hash: *separate chaining* (SC) y *open addressing* (OA), con los diferentes tipos de sondeo para OA: lineal, cuadrático y double hash.

II. Metodología

En este aparatado se habla sobre consideraciones generales, y una breve descripción del programa implementado.

II-A. Consideraciones generales

En una sola librería `hash.h` se implementaron los dos tipos de Tablas Hash: *open addressing* y *separate chaining*. Por lo que se cuenta con funciones generales como:

- Crear nueva tabla.
- Búsqueda.
- Inserción.
- Remover registro.
- Expandir tabla.

En donde, en las mismas funciones se verifica el tipo de Tablas Hash, y usa las funciones correspondientes.

Se usaron estructuras `struct`:

- `hash_table`
- `hash_item`
- `hash_item_root`, especifica para *separate chaining*.

Como función hash $h(k)$ se usó la siguiente expresión:

$$h(k) = k\,\%s \tag{1}$$

Donde k es la llave, y s el tamaño de la tabla.

Para calcular una llave `key` dado un registro `record`, se emplea el algoritmo Adler-32. [2]

```
const uint32_t MOD_ADLER = 65521;
uint32_t compute_key(char *data, size_t
↪  data_len_bytes)
{
    uint32_t a = 1, b = 0;
    size_t index;

    for (index = 0; index < data_len_bytes;
    ↪  ++index)
    {
        a = (a + data[index]) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }
```

```
    return (b << 16) | a;
}
```

Código 1: Código para algoritmo Adler-32.

Sin embargo, como se discutirá más a detalle en la sección de resultados, este algoritmo para calcular llaves no es ideal, ya que diferentes registros pueden generar la misma llave. Por este asunto, se debieron hacer varias consideraciones para la función de búsqueda y por tanto, las funciones de inserción y remover, que dependen en gran parte de la función de búsqueda.

II-A1. Búsqueda

La función de búsqueda, solo toma en cuenta la llave generada según el registro/valor, y regresa la dirección de la estructura del registro si lo encontró. Al regresar la dirección, las funciones de insertar y remover pueden acceder al elemento dado y modificar los atributo, ahorrándose así, una nueva búsqueda.

II-A2. Insertar

Primero se usa la función de búsqueda para saber si la llave del valor está registrada. Si sí existe en la tabla y son valores diferentes con la misma llave, se reemplaza el valor anterior por el nuevo registro solicitado; de esta manera no hay llaves duplicadas. Aunque, también se podría optar por no sobre escribir.

II-A3. Remover

Al igual que en insertar, primero se usa la función de búsqueda para saber si la llave del valor está registrada. Si sí existe la llave en la tabla, pero son valores diferentes con la misma llave, no se remueve el registro solicitado. Debido a que se accesa a la dirección del registro dado por la función de búsqueda, no es necesario hacer búsquedas extras, es suficiente con actualizar los valores.

II-A4. Expansión de tabla

Tanto en *open addressing* como en *separate chaining* se lleva a cabo una expansión dinámica, duplicando la capcidad total de la tabla, disparado por el hecho que el número de registros supera el 50 % de la capacidad total de la tabla.

Es importante que el tamaño de la tabla sea un número primo para disminuir la cantidad de colisiones. Por esto, se tiene un arreglo de números primos, que aproximadamente van duplicándose: 503, 1009, 2017, 4027, 8053, 16103, 32203, 64403, 128813.

II-B. Separate chaining

Para implementar la inserción lateral de un elemento de la Tablas Hash, se empleó un arreglo con tamaño inicial de 4. Los registros se van agregando, en el primer lugar disponible del arreglo. Si es necesario, este arreglo se expande duplicando de tamaño.

Anteriormente, esta inserción lateral se había implementado usando punteros que unieran a los diferentes registros; pero, se debía de llevar cuenta del puntero hacia el registro anterior y el siguiente, lo cual causaba problemas al momento de eliminar registros. Con un arreglo, naturalmente se lleva cuenta del anterior y siguiente, y el remover registros se hace de una manera muy sencilla, marcándolos como no válidos.

II-C. Open addresing

Este tipo de tabla radica en el método de sondeo: lineal, cuadrática, o double hash.

En realidad, solo se ven afectadas las funciones de inserción y búsqueda, específicamente dentro el ciclo, en el intento i, para acceder al elemento j del arreglo de la tabla con tamaño s:

```
j=(h(k)+probingFunction(probing, key, i) ) % s
```

Dentro de la misma función `probingFunction` se elige el j respectivo al probing. En específico:

- Lineal: $j = (h(k) + i)\,\%s$
- Cuadrático: $j = (h(k) + i^2)\,\%s$
- Double hash: $j = (h(k) + i\ h_2(k))\,\%s$
  - Donde $h_2(k) = p - k\,\%s$ y p es un número primo $p < s$.
  - Se uso $p = 499$

II-D. Pruebas de tiempo de ejecución

Se midió el tiempo que le tomaba hacer n operaciones, desde 1k hasta 5M operaciones para las diferentes implementaciones de Tablas Hash. Para esto se empleo un programa como el del Código 2. En donde, se lee un archivo de texto con diferentes cadenas de caracteres y cada una de ellas con el comando de insertar o remover de la tabla.

```
comandos_max=1000;
for(uint8_t pr=0; pr<no_pruebas; pr++){
    size_t ii=0;
    hash_table *ht = ht_new(open_addr); //crar
    ↪  nueva tabla
    ht->probing=ptobing_type;

    clock_t tic = clock();
    while( fgets(command, 100, pFile) != NULL){
        //...
        //Hacer operaciones:insertar o remover
        if( strcmp(operation, "I") == 0)
            ht_insertRecord(fullname, ht);
        if( strcmp(operation, "R") == 0)
            ht_removeRecord(fullname, ht);
```

```
        if(ii>=comandos_max)
            break;
        ii++;

    }
    clock_t toc = clock();
    double tictoc = (double)(toc - tic) /
    ↪  CLOCKS_PER_SEC;
    tiempos[pr]=tictoc;
}
```

Código 2: Código para medir tiempos para un diferente número de operaciones.

II-E. Medición de memoria

Se tenía la intención de medir la memoria usada por el programa; la primera opción era hacerlo manualmente revisando el administrador de tareas, pero no se tuvo éxito con ese métodos. Otras opciones sugeridas, se centran en el sistema operativo de Linux, y para Windows, es un proceso complicado. Por esto, desafortunadamente, no se pudo hacer comparación de uso de memoria.

III. Resultados y discusión

Ha de ser notado que las pruebas de rendimiento se llevaron a cabo en una computadora "ineficiente", tareas que a una computadora le pueden tomar minutos, a esta le puede tomar horas. A pesar de tal situación, terminó jugando a favor, ya que se pudieron apreciar mejor las diferencias entre las distintas implementaciones.

III-A. Tablas hash resultantes

A continuación, en las Figuras 2 y 3, se muestran parte de una tabla hash con 1000 operaciones para todas las implementaciones de Tablas Hash.

```
Hash table: SC - 0
Load factor: 669 / 2017

[ 0]: 4-
[100]: 4-
[200]: 4-
[300]: 4-Xaxu Layomu,
[400]: 4-
[500]: 4-Vo Vibe, Kacije Jici,
[600]: 4-
[700]: 4-
[800]: 4-Xiso Fakupi,
[900]: 4-
[1000]: 4-
[1100]: 4-
[1200]: 4-
[1300]: 4-Laze Piye,
[1400]: 4-Gi Hozu, Yuno Wasoti,
[1500]: 4-Liki Zanime,
[1600]: 4-
[1700]: 4-Ku Zemi, Pu Miwima,
[1800]: 4-
[1900]: 4-
[2000]: 4-
```

Figura 2: Tabla Hash - *separate chaining* representativa.

III-B. Generación de llaves.

Como ya se ha mencionado, la función generadora d llaves Adler-32 está lejos de ser óptima, ya que no era del todo extraño que dos registros con diferentes valores se mapearan

```
Hash table: OA - 1
Load factor: 669 / 2017

[ 0]: (null) - 0
[100]: (null) - 0
[200]: (null) - 0
[300]: Xaxu Layomu - 0
[400]: (null) - 0
[500]: Vo Vibe - 0
[600]: (null) - 0
[700]: (null) - 0
[800]: Xiso Fakupi - 0
[900]: (null) - 0
[1000]: (null) - 0
[1100]: (null) - 0
[1200]: (null) - 0
[1300]: Laze Piye - 0
[1400]: Gi Hozu - 0
[1500]: Liki Zanime - 0
[1600]: (null) - 0
[1700]: Ku Zemi - 0
[1800]: (null) - 0
[1900]: (null) - 0
[2000]: (null) - 0
```

(a) Sondeo lineal.

```
Hash table: OA - 2
Load factor: 669 / 2017

[ 0]: (null) - 0
[100]: (null) - 0
[200]: (null) - 0
[300]: Xaxu Layomu - 0
[400]: (null) - 0
[500]: Vo Vibe - 0
[600]: (null) - 0
[700]: (null) - 0
[800]: Xiso Fakupi - 0
[900]: (null) - 0
[1000]: (null) - 0
[1100]: (null) - 0
[1200]: (null) - 0
[1300]: Laze Piye - 0
[1400]: Gi Hozu - 0
[1500]: Liki Zanime - 0
[1600]: (null) - 0
[1700]: Ku Zemi - 0
[1800]: (null) - 0
[1900]: (null) - 0
[2000]: (null) - 0
```

(b) Sondeo cuadrático.

```
Hash table: OA - 3
Load factor: 669 / 2017

[ 0]: (null) - 0
[100]: (null) - 0
[200]: (null) - 0
[300]: Xaxu Layomu - 0
[400]: (null) - 0
[500]: Vo Vibe - 0
[600]: (null) - 0
[700]: (null) - 0
[800]: Xiso Fakupi - 0
[900]: (null) - 0
[1000]: Wi Yuguyo - 1
[1100]: (null) - 0
[1200]: (null) - 0
[1300]: Laze Piye - 0
[1400]: Gi Hozu - 0
[1500]: Liki Zanime - 0
[1600]: (null) - 0
[1700]: Ku Zemi - 0
[1800]: (null) - 0
[1900]: (null) - 0
[2000]: (null) - 0
```

(c) Double hash.

Figura 3: Tablas Hash - *open addressing* representativas.

a una misma llave. Provocando que ciertos registros no se inserten, borrando registros deseados, o buscando valores incorrectos.

Por ejemplo, los valores "Zi Jaqalu", "Te Vihawiz "Du Qorebo". Incluso se dio el caso en el que se inserta va a Zi, después, sin haber insertado a Te, se pedía removerlo; provocando que Zi se removiera; luego, se insertaba a Du. Finalmente, aun cuando nunca se pidió remover a Zi, en el lugar de esta se encontraba a Du.

III-C. Tiempo de ejecución

Con el método de medición del tiempo de ejecución ya mencionado, y con el parámetro de expandir tabla cuando esta tenga un factor de carga (número de registros / capacidad total) mayor al 50 %; más que nada, fue para asegurarse que el sondeo cuadrático no se ciclara, y prevenir un gran número de colisiones en general. Se tomaron 5 tiempos para un mismo número de operaciones y se promediaron.

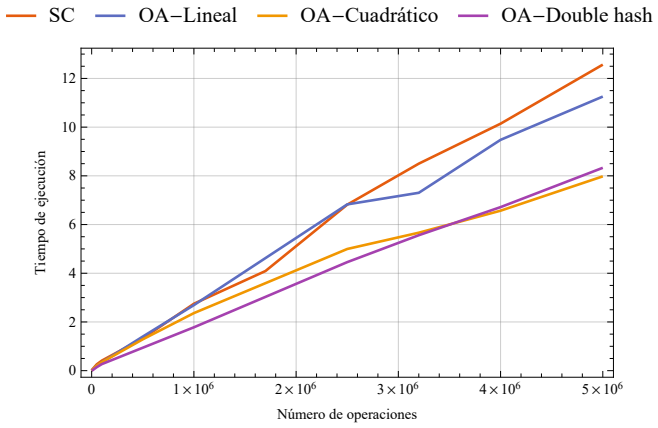

Figura 4: Tiempos de ejecución para diferente número de operaciones, con expansión dinámica en 50 %.

Como se observa en la gráfica de la Figura 4, *open addressing* tuvo un mejor desempeño, exceptuando por el

sondeo lineal, el cual es comparable con *separate chaining*.

Se estimaba que tanto el sondeo cuadrático como double hash tuvieran el mejor desempeño, ya que distribuyen las llaves de una manera más uniforme, sin crear bloques de registros. Además, el proceso de expansión de *open addressing*, aun cuando es costoso, no lo es tanto comparado con el de *separate chaining*.

Era esperado que el sondeo lineal fuera el que peor se desempeñara, debido a que no distribuye uniformemente los registros, creando un tipo de bloques de registros. Sin embargo, no se estimaba que *separate chaining* tuviera el peor desempeño, dado que en general, los arreglos laterales no eran muy grandes; su pobre desempeño se atribuye a que el proceso de expansión es mucho más lento que en *open addressing*, dado que involucra varios ciclos anidados.

También se probó para diferentes factores de carga máximo, es decir, en qué porcentaje de capacidad se dispara la expansión. Para 90 %, (véase Figura 5) hubo grandes diferencias cuando el número de operaciones aumenta considerablemente, sobre todo para *open addressing*. Con esta carga máxima se expandió a 32203, comparado con 64403 para una carga máxima de 50 %.

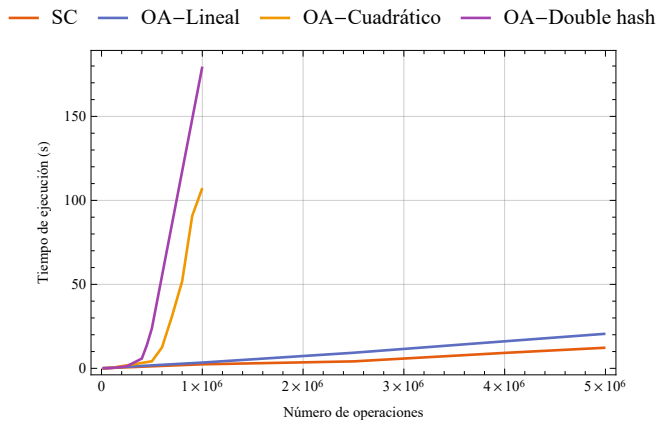

Figura 5: Tiempos de ejecución para diferente número de operaciones, con expansión dinámica en 90 %.

Dado que para garantizar que se pueda insertar un registro efectivamente en el sondeo cuadrático, la capacidad máxima (o el factor de carga) debía ser menor al 50 %. Se estimaba que el cuadrático tardara mucho con respecto la de 50 % (véase Figura 6), debido a que se podía ciclar en le programa o que inclusive no se lograran insertar (se hacen s intentos para insertarlo); pero no se recibió advertencia alguna de que no se encontró un lugar disponible.

Por otro lado, no se esperaba que double hash tuviera tan pobre desempeño, aún cuando fue el mejor método cuando la carga máxima era de 50 % (véase Figura 7), implicando que también hay posibilidad que se cicle. Aunque esto puede ser atribuido a que en cada iteración se calcula la $h_2(k)$, la

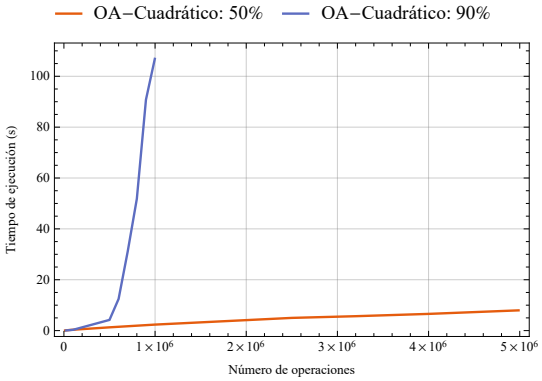

Figura 6: OA-cuadrático: Comparación de tiempos de ejecución para diferente número de operaciones, con expansión dinámica en 50 % vs. 90 %.

cual tiene un módulo, que puede llegar a ser una operación relativamente costosa. No fue posible calcular los tiempos para más de 1M de operaciones, ya que estos tardaban demasiado y la operación tenía que ser abortada.

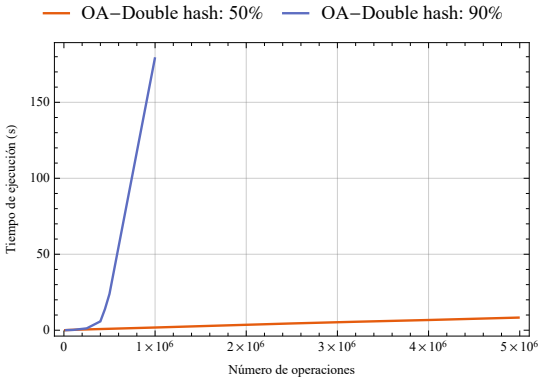

Figura 7: OA-double hash: Comparación de tiempos de ejecución para diferente número de operaciones, con expansión dinámica en 50 % vs. 90 %.

Tampoco se preveía que el sondeo lineal tuviera un desempeño razonable con respecto a la de 50 % (véase Figura **??**). Una posible razón es que al tener un menor tamaño, aunque hubiera mayores colisiones, se hacían menores búsquedas que con respecto la de tamaño 64k. Pero lo mismo puede aplicar para el sondeo cuadrático y lineal, lo cual no es el caso.

Finalmente, en cuanto a *separate chaining*, era de suponerse que tuviera un desempeño comparable con la de 50 % (véase Figura 9), ya que aunque se tenía un 90 % de carga; gracias a las estructuras laterales, se tiene por lo menos el cuádruple de espacio disponible. Ahora, la razón por la que se desempeñó mejor con una carga máxima de 90 % que con la de 50 %, es porque se tenían que hacer menos expansiones, lo cual es una operación demasiado costosa, sobre todo para *separate chaining*.

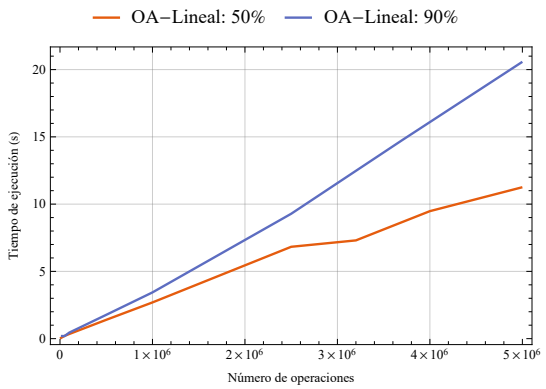

Figura 8: OA-lineal: Comparación de tiempos de ejecución para diferente número de operaciones, con expansión dinámica en 50 % vs. 90 %.

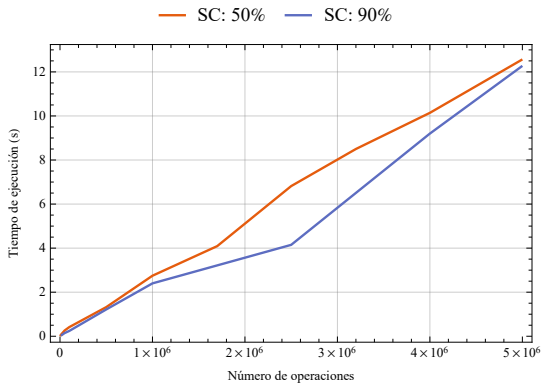

Figura 9: SC: Comparación de tiempos de ejecución para diferente número de operaciones, con expansión dinámica en 50 % vs. 90 %.

III-D. Uso de memoria

Como ya se explicó previamente, no fue posible medir el uso de memoria de la aplicación. No obstante, es fácil ver o deducir que *separate chaining* ocupa más memoria, debido a los arreglos laterales, que en este caso en específico es de mínimo 4. Por lo que se reservaría 4 veces la memoria que la de *open addressing*. Y en cuanto, a los sondeos de *open addressing*, el uso de memoria sería el mismo.

De igual manera, es fácil ver que la cantidad de memoria - tiempo de ejecución tienen una relación inversa, sobre todo para *open addressing*; si hay más espacio disponible, hay menor probabilidad de colisión.

III-E. Otros

Se tuvieron problemas al hacer un barrido con el número de operaciones a realizar; al crear una tabla varias veces en una sola ejecución del programa, se obtenían resultados incorrectos (no se hacía una inserción correcta); sin embargo, al crear solo una tabla en cada ejecución, se obtenían resultados satisfactorios. Este problema se atribuye a la incorrecta liberación de la memoria.

IV. Conclusión

En general, la gran ventaja de las Tablas Hash es que tienen un tiempo de búsqueda constante $\mathcal{O}(1)$ sin ordenar; a comparación de otros algoritmo que suelen tener mínimo $\mathcal{O}(\log n)$, pero se tiene el asunto que debe estar ordenado, por lo que en cada inserción se debe ordenar el arreglo. No obstante, la mayor desventaja es el considerable desaprovechamiento de memoria y la necesidad de ampliar la capacidad de la Tablas Hash, un proceso muy costoso.

Sin embargo, para que la tabla siga siendo viable en cuestión de eficiencia (tiempo de ejecución). Se deben hacer las siguientes consideraciones, centradas en disminuir el número de colisiones:

- El tamaño de la tabla debe ser un número primo.
- No debe de estar muy llena, idealmente 50 % para OA, para evitar un gran número de colisiones; si se llega a dar el caso, se debe expandir la tabla, lo cual es una operación costosa.
- La función hash debe de distribuir las llaves a lo largo de la tabla.
- La función generadora de llaves debe de generar llaves únicas para evitar inconsistencias.

En cuanto a la comparación de las diferentes implementaciones, se encontró que si la carga máxima es de 50 %, *open addressing* (el sondeo cuadrático y double hash) son los más eficientes, debido a que distribuyen mejor los registros. Mientras que, cuando su capacidad de carga aumentó (90 %), su desempeño fue más que deplorable en tanto el número de operaciones aumentaba.

Este asunto es de esperarse, dado que solo se puede garantizar que, si la capacidad de la tabla es menor al 50 %, se evita el riesgo de ciclaje con el sondeo cuadrático (este riesgo aumenta en cuanto la capacidad es mayor al 50 %). Por esto, es necesario expandir la tabla, por lo que podría decirse que su capacidad máxima efectiva es en realidad el 50 % de la tabla. Se encontró que ocurre una situación similar para double hash.

Aunque no se esperaba el “pobre”desempeño de *separate chaining* en el caso de factor de carga máxima, este se atribuye a que el proceso de copias es mucho más costoso que *separate chaining* (por lo menos 4 veces).

Por otro lado, *separate chaining* fue la que tuvo un desempeño más consistente y estable (50 % vs. 90 % de carga máxima) a comparación de *open addressing*(double hashy cuadrático). Fácilmente soportaba cargas máximas mayores al 90 % (incluso mayores a 100 %), y en principio, no es estrictamente necesario expandir la tabla. Lo cual es de esperarse, puesto que crece naturalmente hacia los lados.

Con respecto al uso de memoria, aunque no fue posible medirlo, no es difícil deducir que todos los tipos de sondeo de *open addressing* tienen un uso de memoria similar (si no es que igual); mientras que en *separate chaining*, si bien,

soporta una carga máxima mayor al 100 %, gracias a los arreglos laterales, se usa por lo menos 4 veces más memoria que en *open addressing*.

Algunos "hiperparámetros"que podrían explorarse para mejorar la eficiencia son: la relación del número primo p en double hash con respecto al tamaño de la tabla, factor de carga máximo para disparar expansión.

En general, la mejor implementación será según su aplicación final. Por ejemplo, si no se sabe el número de registros a insertar, se considera que *separate chaining* es la más viable (o en general, la opción más consistente con diferentes cargas máximas), gracias a que naturalmente se expande lateralmente. Mientras que si el número de registros a insertar es conocido, la mejor opción sería *open addressing* (solo si es posible que la carga máxima sea del 50 %).

Como nota, debida la naturaleza de las tablas hash o estructuras de datos abstractas en general, sería conveniente (para el programador) una implementación en un lenguaje orientado a objetos.

Referencias

[1] Robert Sedgewick y Kevin Wayne. (2016). "Hash Tables," Algorithms, 4th Edition, dirección: https://algs4.cs.princeton.edu/34hash/ (visitado 18-03-2021).

[2] *Adler-32*, en *Wikipedia*, Page Version ID: 984590893, 20 de oct. de 2020. dirección: https://en.wikipedia.org/w/index.php?title=Adler-32&oldid=984590893 (visitado 18-03-2021).