Reporte de Resultados

- **Nombre**: Murillo García Sebastián

- **Práctica #3**: Viernes 7 de mayo, 2021

- **Aspectos prácticos adicionales**.

Un aspecto práctico que se tuvo que investigar estuvo relacionado con la lectura de archivos (.csv o .txt). Específicamente se investigó sobre las funciones **open(***directorio del texto***)** y **close()**. Si bien la investigación y la implementación en el código de estas dos funciones resultaron bastante rápidas y sencillas, al principio fue un problema que se sabía cómo resolver, puesto que se había olvidado cuál era la manera más eficiente para hacerlo en Java. En este sentido, es menester dejar en claro que se debe de poner el directorio exacto donde se encuentre el archivo a leer (en este caso se llama **movie_titles.txt**), ya que puede arrojar el típico error *FileNotFoundError*. Debido a que se implementó la lectura de este archivo, fue que se decidió crear una clase Película para extraer la información, crear instancias y poder ocuparlas en las colecciones. Además, el archivo resultó bastante útil para el desarrollo de esta práctica debido a que en este se encuentra una gran cantidad de películas con más de un atributo, pero se hablará de esto más adelante. El código implementado de la clase Película y la lectura de archivos se recicló de otras tareas y prácticas entregadas por uno de los integrantes del equipo. La traducción de los métodos de lectura a Java fue lo que se investigó.

Otro aspecto que se investigó fue algunas de las funcionalidades de la biblioteca **pyplot** de **matplotlib**. Si bien ya se conocía el cómo utilizar algunas de las funciones (bastante sencillas) como **pyplot.plot(**componente x, componente y**)**, **pyplot.title(**título**)**, **pyplot.xlabel(**nombre**)**, etcétera, en la lluvia de ideas que se generó cuando se propuso representar los resultados en gráficas se presentaron varias preguntas relacionadas con el tipo de gráfica que se podía ocupar. Se ocupó en investigar sobre los tipos de gráficas que nos ofrecía esta famosa biblioteca y que fueran las más adecuadas para los objetivos propuestos. Después de navegar en la página oficial de la biblioteca y de discutir mentalmente una vez más acerca de los resultados que se esperaban obtener, se llegó a la conclusión de que se podía implementar un solo tipo de gráfica nueva y que era la más adecuada: una gráfica de barras. Esta implementación se logra con un simple paso: cambiar ‘plot’ por ‘bar’ en la primera función antes descrita, la cual quedó de la siguiente manera: **pyplot.bar(**componente x, componente y**)**. Si bien la extensa búsqueda de una nueva gráfica dio a parar en un cambio tan sencillo, se considera que esa

misma búsqueda rindió frutos importantes que pueden ser utilizados en otro momento académico. En otras palabras, el haber buscado tanto tiempo una gráfica ideal para los objetivos ambiciosos que se presentaron, brindó la posibilidad de que se conocieran más funcionalidades de la biblioteca e incluso dieron soluciones a otras tareas de programación, no solo para esta práctica. Otros pequeños detalles que se implementaron fue el establecimiento de un color específico para las gráficas y también la determinación de la "anchura" de las barras en la gráfica donde se ocuparon, un ejemplo es el siguiente: *pyplot.bar(numCasillas, datosMult,* ***1.0****,* ***color*** *=' mediumpurple')*.

- **Aspectos conceptuales adicionales**.

Un tema adicional que se tuvo que investigar estuvo relacionado con un error arrojado mientras se hacía la lectura del archivo de texto que se ocupó para la práctica. En el momento de estar haciendo las pruebas de los métodos independientes para cerciorarse de que funcionaran en su totalidad, cuando se intentaba leer una cantidad grande de líneas del archivo Python arrojaba el error ***UnicodeDecodeError:*** *'utf-8' codec can't decode byte 0xe9 in position 4255: invalid continuation byte*. Al investigar un poco en internet se encontraron algunas respuestas confusas como aquella que dice que este error ocurre cuando existe un separador problemático en la línea o que sucede cuando se trata de leer un caracter que no estaba codificado en utf-8. Se decidió no investigar más a profundidad sobre este problema debido a que el límite de líneas que se podían leer del archivo bastaban para obtener los resultados esperados de la práctica; además, si en algún momento se necesitaba leer más de esa cantidad, simplemente se podrían agregar algunas líneas más de código para saltarnos justo esa línea problemática al momento de leer del archivo, pero no creímos que fuera necesario para los objetivos de la práctica. El límite que se pudo leer del archivo fue de 12,094 líneas. Incluso por motivos relacionados a una buena visibilidad de las gráficas generadas para el análisis e interpretación de los resultados, se decidió ocupar una cantidad menor de líneas para crear instancias de Películas en la práctica, por lo que con más razón no se vio necesaria la investigación profunda del error arrojado al intentar leer más de 12,094 líneas.

Otro tema que se tuvo que investigar dentro del aspecto conceptual (aunque también puede ser considerado en el aspecto práctico) fue el de la búsqueda de otros métodos para la función de hash. ¿Por qué se investigó esto si el profesor ya había indicado dos métodos completamente funcionales? Porque se decidió implementar algunos otros métodos para poder comparar su eficiencia con aquellos dados por el profesor. Las preguntas que se querían resolver eran las siguientes: ¿por qué la sugerencia de ambos métodos para el desarrollo de la práctica? ¿Cuántas otras funciones de hash habrá que

cumplan con las propiedades deseables de las funciones de hash? Por lo tanto, se emprendió la búsqueda de distintas funciones de hash para poder comparar su desempeño con las otorgadas en la práctica por el profesor. Aquí se debe de mencionar algo importante: los resultados esperados no fueron lo más satisfactorios; tan poco satisfactorios fueron que se decidió eliminar del código. Entonces, ¿por qué se hace una mención de esta búsqueda por nuevas funciones si al final no se implementaron? Porque esta búsqueda amplió el espectro visual de los métodos que pueden ser ocupados para la estructura de datos de Tabla de Hash. Fue tan ambiciosa la propuesta de buscar otros métodos que hasta se llegó a pensar en la posibilidad de crear uno nuevo o modificar uno existente (se topó muy pronto con pared en este asunto). Más que una funcionalidad práctica (por eso se puso en el apartado de *Aspectos conceptuales adicionales*) se piensa que el valor de esta búsqueda radicó más en el sentido puro de la investigación y es por ello que se considera digno de mencionar en la práctica.

- **Retos**.

Uno de los retos más grandes que se presentó durante la realización de esta práctica fue el establecimiento de los objetivos y la forma por la cual se iba a poder llegar a los resultados deseados, puesto que inicialmente se estaba teniendo un enfoque erróneo de los medios para llegar al objetivo de la práctica. Por un lado, no se sabía exactamente qué se iba a medir y cómo es que el análisis de esas mediciones nos iban a dar un entendimiento de la estructura de datos tratada en esta práctica. ¿Debíamos mantener un tamaño de la tabla de hash fijo y variar la cantidad de datos que iban a ser operados en ella?, ¿convendría mejor mantener la cantidad de datos fija y variar el tamaño de la tabla de hash?, ¿cómo íbamos a entender el funcionamiento general de la estructura si se implementaba de una manera u otra? Por otro lado, las hipótesis se estaban haciendo mucho antes de conocer, en su base, la teoría de esta estructura de datos. Como todo en la vida, se tuvo que empezar a caminar desde un punto inicial, así que primero se ocupó el esfuerzo por establecer las clases necesarias básicas y fundamentales (como lo fue la clase Película y claramente las clases HashNode y HashTable). Al desarrollar este código base fue que se tuvo un entendimiento más claro de la teoría que está detrás del desempeño de una tabla de hash y se pudo asentar una idea más clara sobre los experimentos y medios para llevar a cabo el objetivo de la práctica. Así fue como se empezaron a gestar los diferentes códigos en distintas capas (bases, pruebas y gráficas) para dar lugar a los resultados de la práctica.

Un segundo reto que se presentó fue la articulación de los diferentes métodos para obtener las gráficas que se tenían propuestas como objetivos. Esto se debió a que el código de los “prototipos” de pruebas primeramente se hacían al final de todo el *script* para tener lo más a la mano posible cada

detalle de la prueba y gráfica que se estaba intentando implementar. En esta sección se hacían todas las modificaciones que se creían necesarias al código mientras que se bombardeaba con casos "de esquina" para conocer las limitaciones y puntos débiles del código. Una vez que se tenía probado en su totalidad a la prueba (o al menos eso era lo que se esperaba), se procedía a separar las partes del código en sus diferentes capas: la capa más básica (clases Pelicula, HashNode y HashTable), la capa de pruebas (la medición de tiempos y la obtención de los datos necesarios para las gráficas) y la capa de gráficas (donde se insertan los datos para poder visualizarlos en un producto final). Esta separación del código resultó un poco problemática, pero brindó bastante apoyo cuando se deseaba cambiar alguna variable o se deseaba hacer una implementación más profunda. Si bien la articulación del código en pruebas "unificadas" fue un proceso complicado, la flexibilidad de estas mismas fue una ventaja en el momento de querer explorar nuevas variantes del código.

- **Detalles experimentos y otras cuestiones a considerar**.

En primer lugar, debe hacerse mención sobre la elección del archivo "movie_titles.txt" como el *dataset* de la práctica implementada. Como primer parámetro de elección fue la disponibilidad del archivo, puesto que este documento fue proporcionado por el maestro Fernando Esponda Darlington para una tarea de la materia Estructura de Datos Avanzadas y en principio no se pensó como necesaria una búsqueda en internet de otros datos funcionales para la práctica. Además, una vez que ya se había trabajado con ese archivo, se tenía una idea más o menos clara de cómo manejar la información contenida en él. Este aspecto de disponibilidad no fue el único factor a tomar en cuenta. También tuvo mucho que ver el hecho de que el archivo brindaba la posibilidad de trabajar con una cantidad mayor a 10,000 datos, pero menor a 20,000. Cuando se buscó en internet otros *datasets* para la implementación, muchas veces se encontró con algunos bastante pequeños (entre 500 y 1,000 datos) o extremadamente grandes (millones de datos) basándonos en los gustos e intereses personales. Esta cantidad justa entre 10,000 y 20,000 se veía prometedora para cumplir con los objetivos de la práctica sin llegar a extremos notorios. Además, otro problema que se presentó fue que en muchos de estos *datasets* algunos atributos de las instancias de los objetos que trataban se presentaban como nulos, cosa que hacía difícil la lectura y la interpretación de ese atributo en el archivo. Como último factor, el archivo "movie_titles.txt" brindaba la posibilidad de crear una clase Película para poder instanciar objetos más complicados o profundos que un simple entero.

Ahora bien, una vez que se determinó qué tipo de dato se iba a manejar en la tabla de hash, fue que se procedió a tomar una decisión sobre la función de hash más adecuada con este tipo de dato. La clase

Película tomó en consideración un atributo llamado “id” (numérico), mismo que venía como primer dato en el archivo de las películas. Este dato fue el que se eligió como el valor llave del Objeto. Si se observa el archivo con detenimiento, estos valores no están en orden y están dentro de un rango entre 1 y 16,000 sin repetición (esto se observó en la tarea de Algoritmos de Ordenamiento de la materia antes mencionada). Por lo tanto, una vez leído el archivo de la práctica 3, se tomó la decisión de que la función de hash por el método de la división era el más adecuado con este tipo de dato. ¿Por qué? Se menciona que si *k1* y *k2* son valores clave adyacentes, entonces *h(k1)* y *h(k2)* probablemente también sean adyacentes, aunque no siempre se iba a cumplir esta condición. La disponibilidad de los datos en el archivo parece ser la adecuada para funcionar en armonía con esta función de hash porque el valor clave de los datos sí son adyacentes. En principio los datos no están ordenados, pero se consideró que lo más prudente era intentar mantener esa posibilidad de adyacencia de los valores clave en la tabla; en otras palabras, a pesar de que el valor clave 1 y el valor clave 2 se encuentran alejados entre sí en el archivo, se pensó que la mejor idea fuera que estos valores fueran adyacentes en la tabla.

Ya en un aspecto más relacionado al diseño de los experimentos, se abordaron tres grandes “espectros” para analizar. En primer lugar, se consideró contemplar una prueba en la cual se pudiera analizar en su generalidad el comportamiento de las operaciones (inserción, búsqueda y borrado) en tiempo en la tabla de hash en relación con la cantidad de datos operados en ella. Para ello, se desarrollaron dos métodos que tuvieron un enfoque distinto para este mismo objetivo: **pruebaTamanioFijo** y **pruebaEntradaFija**. el primer método mencionado mantiene un tamaño de tabla de hash fijo (exactamente en 5,000 casillas) mientras que la entrada de datos es la que varía (entre 1,000 y 1,000 datos en 7 diferentes entradas); es decir, la variable fija es **m** y **n** es la que cambia. Por otro lado, el segundo método mantiene una entrada de datos fija (exactamente en 5,000) mientras que el tamaño de la tabla de hash es la que varía (entre 1,000 y 1,000 datos en 7 diferentes entradas); es decir, la variable fija es **n** y **m** es quien cambia. Ambos métodos ocupan un **caso general** (un método) para conocer el comportamiento cuando ‘n’ es menor ‘m’, cuando ‘n’ es aproximadamente igual a ‘m’ y cuando ‘n’ es mayor a ‘m’ en una misma gráfica. Como hipótesis inicial, ambos casos deberían de arrojar los mismos resultados, pero más adelante se comprobará. Este método de casos puede ser cambiado por el usuario por otro caso en el cual se contemple solamente cuando ‘n’ es menor a ‘m’, uno más cuando ‘n’ es aproximadamente igual a ‘m’ y uno final cuando ‘n’ es mayor a ‘m’ para analizar más a fondo qué sucede en esas zonas (si es que se desea). En total se tienen 4 casos que se pueden llamar para ambos métodos. Para fines de esta práctica solo se expondrán las dos gráficas de los casos generales, pues son

los más ilustrativos para los objetivos de la práctica. Los tiempos que se consideran en estas gráficas son aquellos que representan la **operación por cada tamaño de entrada de datos**.

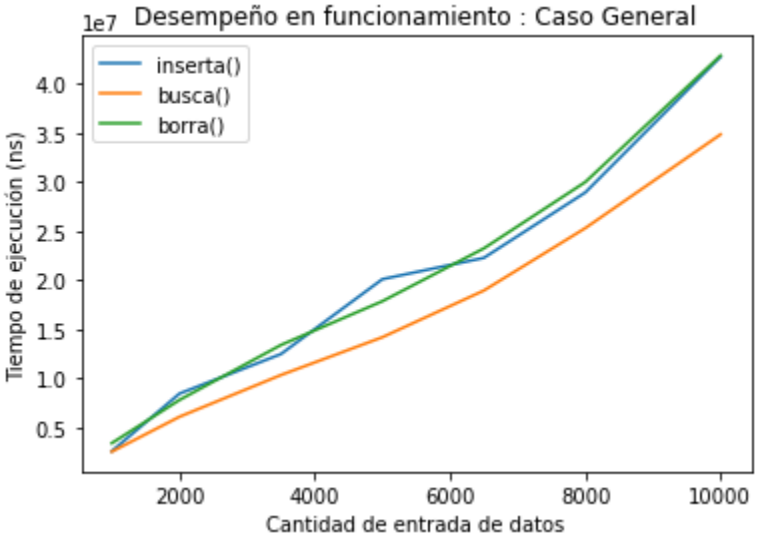
Como segundo espectro a analizar, se consideró contemplar un método que analice las mediciones de tiempo por dato para los tres casos antes mencionados: **n < m** (con 'n' igual a 100 datos y 'm' igual a 1,000 casillas), **n = m** (con 'n' igual a 1,000 datos y 'm' igual a 1,000 casillas) y **n > m** (con 'n' igual a 1,000 datos y 'm' igual a 100 casillas). El método es llamado **pruebaParticular**. Estos casos se pueden considerar para una cantidad de datos pequeña (aproximadamente en 1000) o grande (aproximadamente en 5000); particularmente para esta práctica se decidió conveniente analizar una cantidad de datos pequeña por la visualización de las gráficas, aunque si el usuario lo desea, puede cambiar ese parámetro. Aquí se tiene un análisis más exacto sobre el desempeño de las operaciones de inserción, de búsqueda y de borrado en la tabla de hash con valores fijos y el tiempo que se promedia es el de la **operación por cada dato**. También en este segundo espectro se considera una gráfica que representa al número de datos por casilla de la tabla de hash como guía para entender el porqué del funcionamiento. Los datos para esta gráfica se obtienen del método **pruebasDatos**.

Como tercer y último espectro se considera un método que compara el desempeño de las funciones de hash: **pruebaComparacion**. Este método busca comparar dos aspectos: el desempeño en tiempo y el desempeño en acomodo. Como la función de hash se ocupa para las tres operaciones principales de la tabla de hash ya que se ocupa para conocer la posición en la que un elemento debe estar, se consideró suficiente mostrar el desempeño de tiempos al momento de la inserción, puesto que para las otras dos operaciones daría resultados análogos. En relación al acomodo, es interesante conocer qué tanto distribuye los datos de entrada una función con respecto a la otra. Por esto mismo se decidió mantener el caso cuando 'n' es igual a 'm' para conocer qué tan distribuidos están los datos en las tablas y el promedio de tiempo de ejecución de la misma operación.

Debe mencionarse que para los espectros 2 y 3 de los experimentos se consideró un "tope" en tiempos, pues facilita la visualización de las gráficas y "acota" el margen error causado por la propia máquina al intentar hacer una operación en la tabla, ya que como se está haciendo la medición de la operación por dato, estos valores suelen ser muy fluctuantes.

- **Análisis de los resultados**.

I. *Gráfica sobre el desempeño cuando el tamaño de la tabla de hash es fijo (m = 5000).*



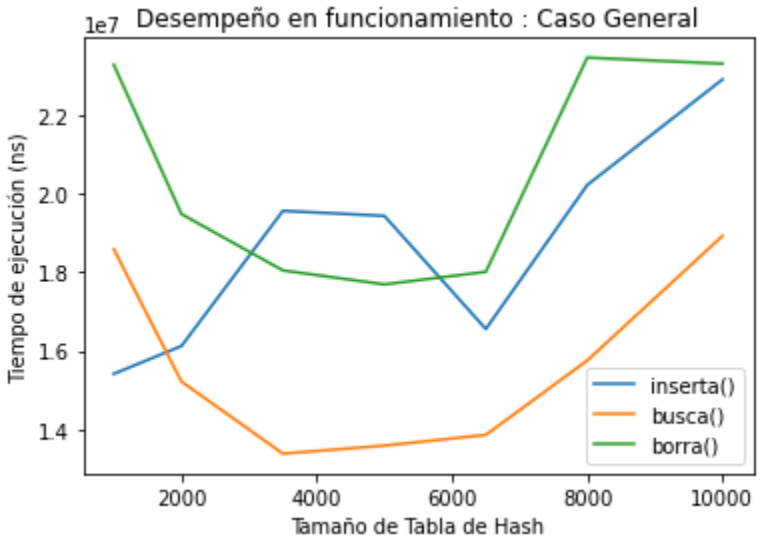
Si el tamaño de la tabla de hash es fijo con 'm' = 5000 y la entrada de datos 'n' es la que cambia, podemos leer la gráfica de izquierda a derecha como un cambio que va de n < m en la zona lateral izquierda, con n = m en la zona central, hasta n > m en la zona lateral derecha. Aquí podemos hacer unas observaciones interesantes. Para empezar, la operación de búsqueda parece ser la más rápida entre las tres operaciones no importa el caso en el que nos encontremos. ¿Por qué hace de ser así? Recordemos que en estas gráficas la diferencia de tiempos está en relación al tamaño de entrada, es decir, la primera medición se hace antes de insertar los 'n' datos en la tabla y la segunda se hace después de insertar esos 'n' datos. Entonces como primera respuesta se puede decir que se debe a que esta operación trabaja con una variable booleana en su base y que no necesita de otra cosa más que de esa característica de ser verdadera o falsa. Por otro lado, las operaciones de inserción y de borrado trabajan "más a profundidad" en la tabla, ya que tienen que modificar las listas contenidas en ella y este proceso de "modificación" de los apuntadores claramente puede llevar un poco más de tiempo.

Después de haber dicho esto es interesante analizar el caso de la operación de inserción y la operación de borrado. La intuición (relacionada a la implementación del código) diría que la operación para insertar 'n' elementos en la tabla de hash debería de ser más rápida que la de borrado de esos mismos 'n' datos debido a que en el caso de la inserción los elementos se insertan justo al inicio de las listas ligadas y que en el caso del borrado de los elementos se tienen que buscar entre las listas. Estamos seguros de que ambos tienen que modificar los apuntadores, entonces lo único que diferencia ambas operaciones es que una inserta al inicio (y esto suena bastante rápido) y otra busca en las listas

(y esto suena lento). La gráfica muestra que entre el caso cuando ‘n’ es menor a ‘m’ y cuando ‘n’ es aproximadamente igual a ‘m’ estos promedios de tiempo son fluctuantes y no podemos decir con seguridad cuál es el que tarda menos; pero cuando se trata del caso cuando ‘n’ es mayor a ‘m’, podemos observar que claramente la inserción es menos tardada. Con respecto a este último caso todo tiene sentido, ya que entre más datos se tenga en la tabla, más elementos van a estar en cada lista de las casillas; al seguir esta lógica, se va a tardar más en buscar a un elemento que insertarlo. ¿Y qué pasa cuando nos encontramos en los otros dos casos? Por más que se analizó el asunto, no se llegó a una conclusión concreta. Cuando ‘n’ es menor a ‘m’ habría que esperar que existieran muchas casillas vacías y que las que están ocupadas por una lista, estas deberían de contener muy pocos elementos; además, cuando ‘n’ es aproximadamente igual a ‘m’, se esperaría que en promedio cada casilla contuviera a un solo elemento en su interior. Tal vez en esto radica la fluctuación: no hay suficientes datos en las listas como para determinar que la inserción es más rápida (pues borrar a un elemento de una lista que contiene a un solo elemento no debe de ser más tardado, por ejemplo) y esa medición de tiempo puede deberse a procesos concretos de la computadora.

Otro aspecto importante a tener en cuenta es que los tiempos de ejecución son crecientes con respecto al tamaño de entrada de datos.

II. Gráfica sobre el desempeño cuando la entrada de datos es fija (n = 5000).



Esta gráfica resulta curiosa a primera vista. Si la cantidad de datos de entrada es fija con ‘n’ = 5000 y el tamaño de la tabla de hash ‘m’ es el que cambia, podemos leer la gráfica de izquierda a derecha como un cambio que va de n > m en la zona lateral izquierda, con n = m en la zona central, hasta n < m en la zona lateral derecha. Desde aquí podemos partir para empezar a analizar.

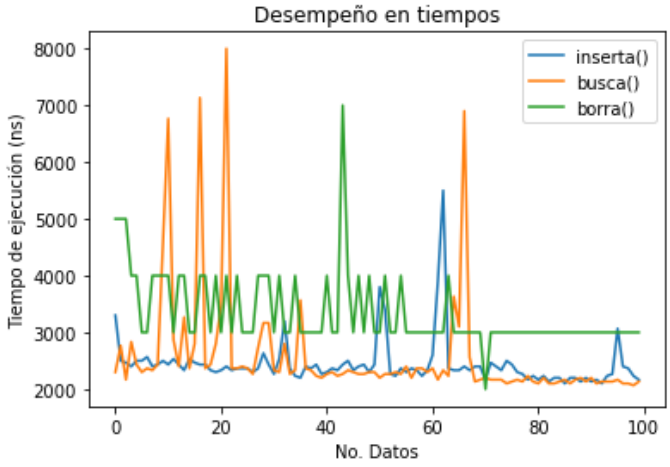
Al igual que la gráfica anterior se puede ver un comportamiento general del orden de ejecución para la operación de búsqueda muy parecido al de la gráfica anterior (es más rápido buscar que otra operación), pero podemos observar otras cosas. Para empezar, podemos notar que al inicio de la gráfica insertar es más rápido que buscar, cosa que no se había visto en la gráfica pasada. ¿A qué se debe? Tenemos que considerar el hecho de que se están intentando buscar e insertar 5000 datos en una tabla que solamente tiene 1500 casillas aproximadamente; he aquí la respuesta. Las listas están atascadas de datos y la operación de búsqueda no es la más rápida porque tiene que recorrer listas, mientras que la operación de inserción es rápida porque simplemente se insertan al inicio de las listas. Este comportamiento empieza a cambiar conforme la tabla de hash va aumentando hasta que llegamos al caso en que la operación de búsqueda es más rápida que la de inserción como habíamos visto en la gráfica pasada.

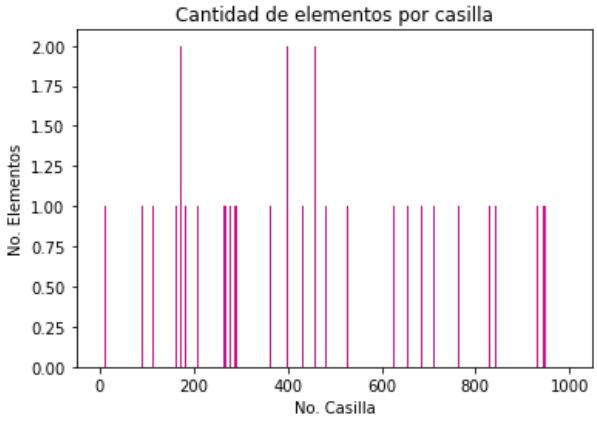
Ahora bien, podemos ver un comportamiento curioso y muy notorio: la función que representa a la operación de borrado es muy (demasiado) parecida a la de búsqueda, solo que parece tardar más. Esto es completamente razonable, ya que ambas operaciones tienen que recorrer de la misma manera las listas, solo que la operación de búsqueda trabaja con una variable booleana que finaliza cuando se encuentra al método mientras que la operación de borrado tiene que modificar los apuntadores de las listas para eliminar al elemento.

Algo curioso que debemos notar es que nuevamente se presenta la fluctuación en tiempos cuando se presenta el caso de ‘n’ aproximadamente igual a ‘m’. Podemos dar la misma respuesta de esta fluctuación que se mencionó en la gráfica anterior, puesto que a pesar de analizar y meditar el porqué de los resultados, no se llegó a nada concreto.

Ahora bien, ¿por qué la gráfica parece una “u” en los tiempos de borrado y de búsqueda? La mitad lateral izquierda se puede explicar a que entre más casillas se tienen para insertar a los 5000 datos, cada vez habrán menos elementos en las listas y será más sencilla la búsqueda y el borrado. Si seguimos esta lógica, ¿por qué la mitad derecha es creciente en tiempos? Sinceramente es otra laguna que no se pudo resolver al momento de analizar estas gráficas, pues no tiene mucha lógica que el tiempo tarde más cuando se está insertando, buscando y borrando la misma cantidad de datos a una tabla que cada vez es más y más grande. Por lo tanto, es una cuestión que se deja sin respuesta, pero con una gran interrogante.

III. *Gráfica sobre el desempeño en tiempos de las operaciones por dato y gráfica sobre la distribución de los elementos en la tabla de hash cuando $n < m$.*





Estas gráficas representan la inserción de 100 datos en una tabla de hash de 1000 casillas. Aquí se está considerando de manera particular el caso cuando **'n' es mucho menor a 'm'** (la cantidad de entrada de datos representa el 10% de la capacidad de la tabla de hash). Como datos importantes a tomar en cuenta que son arrojados en la consola es que el número de casillas vacías de la tabla es de 905, es decir, solo se ocuparon 95 casillas del arreglo y podemos observar claramente en la gráfica que aquellas listas que albergan la cantidad mayor de elementos solo guardan 2. Estos datos son bastante importantes para interpretar la gráfica de tiempos. Recordemos que, de ahora en adelante, estas gráficas representan el promedio de tiempo de operación por cada dato, es decir, la primera medición de tiempo se toma antes de la inserción de un elemento y la segunda medición de tiempo se toma justo después de la inserción, por ejemplo.

Los tiempos promedio de las operaciones son los siguientes:

- inserción: 2441.66 ns
- búsqueda: 2583.66 ns
- borrado: 3340.0 ns

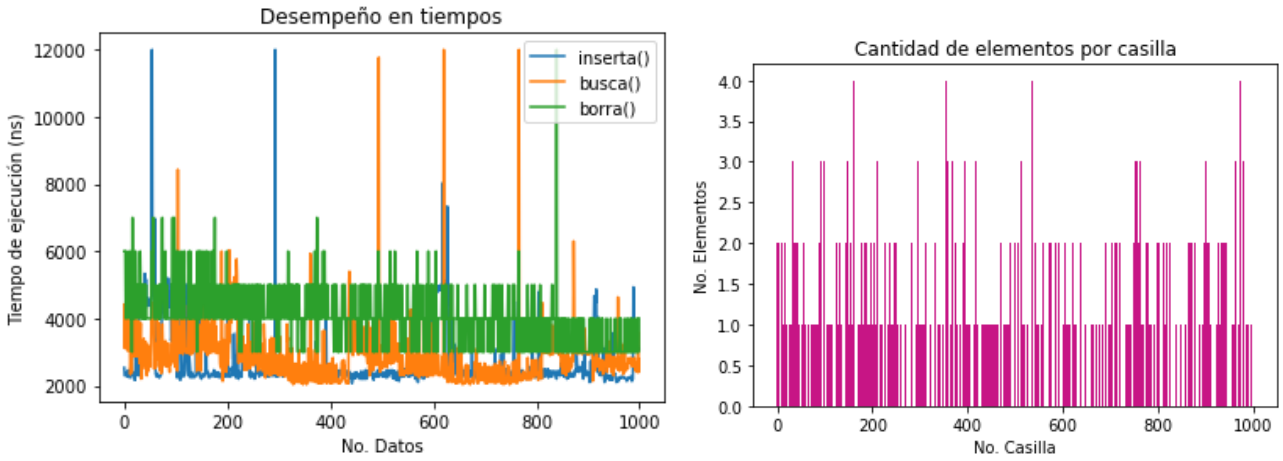
Algo curioso a observar es que el promedio de tiempos de inserción y el de búsqueda es muy cercano para toda 'n' operada. ¿A qué se debe esto? Al igual que se explicó anteriormente en las otras gráficas, la cantidad de datos insertados en la tabla no es la suficiente como para que se tengan listas muy grandes, por lo que hacer la operación de inserción es aproximadamente igual en tiempo a la de búsqueda porque no se tienen que recorrer listas. Si seguimos esta lógica, entonces ¿por qué la de

borrado es más tardada? Si bien no tiene que recorrer listas demasiado largas (la más extensa de ellas es de 2 elementos), esto se puede deber a la implementación del código de borrado, ya que este tiene que recordar a un nodo previo para que se pueda hacer la correcta eliminación del dato. Este *reminder* del nodo anterior puede ser costoso en tiempo y es por ello que existe una notable diferenciación en los tiempos.

Otra cosa bastante importante a considerar es que aquí vemos claramente la propiedad que nos dice la teoría de que las operaciones en una tabla de hash son ***O(1)***, pues podemos observar que no importa el tamaño de entrada de datos, la operación se tarda en un tiempo constante (hasta parece ser una línea recta un poco fluctuante). Por lo tanto, esta prueba refleja claramente la teoría de las tablas de hash en un primer inicio cuando **n < m**.

Como conclusiones de esta prueba tenemos que el promedio de tiempos de inserción de datos y el de búsqueda parecen ser muy cercanos para el caso considerado, pero que el de tiempo de borrado es el más tardado. A pesar de ello, los tres cumplen con la característica de ser constantes.

IV. Gráfica sobre el desempeño en tiempos de las operaciones por dato y gráfica sobre la distribución de los elementos en la tabla de hash cuando $n \approx m$.



Estas gráficas representan la inserción de 1000 datos en una tabla de hash de 1000 casillas. Aquí se está considerando de manera particular el caso cuando **'n' es igual a 'm'** (la cantidad de entrada de datos representa el 100% de la capacidad de la tabla de hash). Como datos importantes a tomar en cuenta son que el número de casillas vacías de la tabla es de 348, es decir, todavía el 34.8% de la tabla no contiene ni a un solo elemento en su interior. Se puede observar en la gráfica de la derecha que aquellas listas que almacenan la mayor cantidad de elementos solo guardan hasta 4 elementos. Recordemos que estos datos son importantes para el análisis de la gráfica de tiempos.

Los tiempos promedio de las operaciones son los siguientes:

- inserción: 2697.6 ns
- búsqueda: 2875.90 ns
- borrado: 4086.0 ns

Podemos ver muchas cosas interesantes al analizar esta gráfica de tiempos. En primer lugar, podemos observar que los tiempos de borrado al inicio de la gráfica son mayores y que cada vez disminuyen un poco conforme se acerca al final de la gráfica. ¿A qué se debe esto? Esto tiene una relación clara con la cantidad de datos que se desean operar. Al principio tenemos que considerar que se están intentando borrar los primeros elementos de la tabla de hash, es decir, la tabla comienza con todos los datos insertados. Conforme se avanza en el componente de las 'x', se están eliminando cada vez más datos y la tabla de hash queda más vacía. Entonces al principio de la gráfica se tarda más en borrar porque se tienen que recorrer más elementos en las listas y como cada vez hay menos elementos mientras nos movemos en el eje de las abscisas, menos tiempo se tarda en borrar porque ya no se tienen que recorrer tantos elementos en las listas. A pesar de este comportamiento singular, podemos decir que en general sí presenta un desempeño muy cercano a ***O(1)***, pues todavía se puede trazar una línea horizontal en tiempo (que me perdonen los grandes matemáticos por haber interpretado de esta manera esa característica).

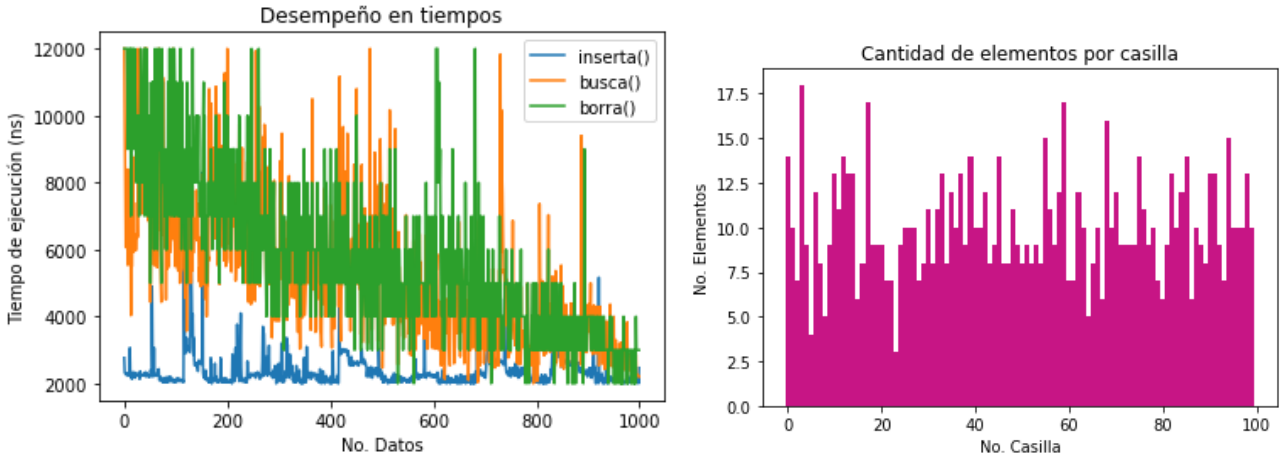
Así mismo, podemos observar que la operación de búsqueda tiene sus complicaciones a lo largo de todo el componente de las abscisas, aunque claramente tiene algunas fluctuaciones. Este comportamiento podemos explicarlo debido a la característica de las listas. Como en el método de búsqueda no está alterando a la tabla ni a sus elementos, podemos esperar que en general se tarde más que la operación de inserción porque ahora sí existen elementos que recorrer en las listas. También por eso podemos ver que en la gráfica existen fluctuaciones: puede deberse a que justo en la búsqueda de esos elementos en la tabla se encuentre con listas que están bastante llenas y por eso la demora. De todos modos, podemos afirmar que el tiempo promedio sigue siendo muy cercano al de inserción. De la misma manera, podemos afirmar que el desempeño general sigue siendo muy cercano a ***O(1)*** a pesar de tener esas fluctuaciones.

Lo que sí se debe de mencionar ahora es que la operación de inserción sí mantiene un desempeño general de ***O(1)*** y que no se vio tan afectado por el caso considerado (n = m). Esto se explica enteramente por la implementación del código. Como simplemente se tiene que encontrar la posición en la tabla e insertar inmediatamente al inicio de la lista, no se encuentra con ninguna complicación de

recorrido que pueda afectar su desempeño. Esto es bastante interesante y demuestra una vez más lo visto de la teoría de las tablas de hash. Es por este mismo desempeño que se puede considerar que la operación de búsqueda y de borrado siguen siendo muy cercanos a *O(1)*. ¿Por qué? Porque claramente se puede visualizar que los tiempos no están tan alejados del comportamiento de la operación de inserción.

Como conclusiones de esta prueba tenemos que el promedio de tiempos de inserción de datos es el más rápido y que el de búsqueda y de borrado cada vez se alejan más de este promedio, pues se encuentran con complicaciones en el momento de recorrer las listas. A pesar de ello, los tres cumplen en gran medida con la característica de ser constantes.

V. *Gráfica sobre el desempeño en tiempos de las operaciones por dato y gráfica sobre la distribución de los elementos en la tabla de hash cuando n > m.*



Estas gráficas representan la inserción de 1000 datos en una tabla de hash de 100 casillas. Aquí se está considerando de manera particular el caso cuando **'n' es mucho mayor a 'm'** (la cantidad de entrada de datos representa el 1000% de la capacidad de la tabla de hash). Como datos importantes: el número de casillas vacías de la tabla es de 0, es decir, todas las casillas de la tabla están llenas. Además, las listas que almacenan la mayor cantidad de datos son de aproximadamente 18 elementos en su interior. No es necesario recordar la importancia de estos datos.

Los tiempos promedio de las operaciones son los siguientes:

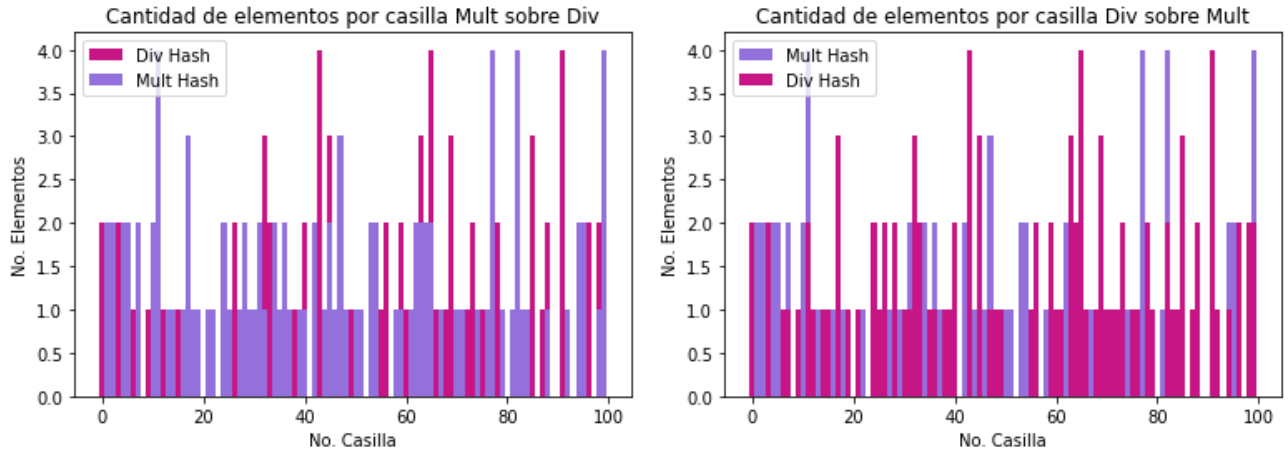
- inserción: 2349.76 ns
- búsqueda: 5181.83 ns
- borrado: 5733.0 ns

Para empezar, es destacable hacer mención de la eficiencia de la operación de inserción en este caso. Podemos observar que su desempeño sigue siendo de ***O(1)*** a pesar de que se esté insertando una cantidad mucho mayor a la capacidad de la tabla. Claro, nos encontramos de vez en cuando con algunas fluctuaciones, pero el promedio sigue siendo muy parecido al de las otras dos gráficas. Esto resulta ser maravilloso y refleja la teoría del desempeño ideal buscado en esta estructura de datos.

¿Qué sucede con el desempeño de la operación de borrado y la de búsqueda? Ambos tienen un desempeño bastante alejado del esperado (***O(1)***). El desempeño en tiempos de la operación de borrado tiene la misma lógica explicada en el análisis de la gráfica anterior. Al inicio la tabla está completamente llena y poco a poco se van vaciando las listas conforme se borran los elementos. Al inicio es más tardado borrar porque se tienen que recorrer más elementos en las listas y al final es menos tardado porque las listas a recorrer contienen pocos elementos en su interior. Esto es bastante simple. Pero ¿por qué el comportamiento de tiempo de la operación de búsqueda se parece tanto al de borrado si simplemente se está buscando al elemento y no se modifica nunca la extensión de las listas o algo parecido? Esto se explica por una lógica bastante sencilla: si la inserción de los datos se hace al principio de la lista correspondiente, entonces resulta lógico pensar que los primeros datos insertados van a ser los últimos datos de las listas cuando se termine el proceso de inserción de los 'n' datos y que los últimos datos insertados van a ser los que se encuentren primero en las listas. Aquí está la explicación más sencilla. El comportamiento en tiempos de búsqueda se parece bastante al de borrado porque se tienen que buscar a los elementos que se encuentran más al final de las listas dado el orden de inserción de los elementos. Esta misma lógica afecta a la operación de borrado de los elementos de la tabla de hash.

Por lo tanto, ¿cuál es la conclusión? Para la operación de inserción, el desempeño no se ve afectado por la cantidad de datos que se desea insertar en la tabla de hash y parece ser de complejidad ***O(1)***. El problema ocurre con las operaciones de borrado y búsqueda **en este método de encadenamiento** (esto es bastante importante, no podemos perderlo de vista), puesto que parece ser que conforme se exceda la capacidad de de la tabla, el desempeño se va alejando del ideal ***O(1)*** para acercarse a un desempeño de ***O(n)***. Tampoco hay que ser tan drásticos, los promedios de tiempo de estas últimas dos operaciones no son tan distantes con respecto al de inserción, pero también debemos de ser conscientes de que se ocupó una cantidad bastante pequeña de datos, con 1,000 solamente. Este problema puede agravarse cuando se trata de una cantidad mucho mayor de datos.

VI. *Gráfica sobre la comparación de la distribución de los datos por dos funciones de hash distintas: la de división y la de multiplicación cuando $n = m$.*



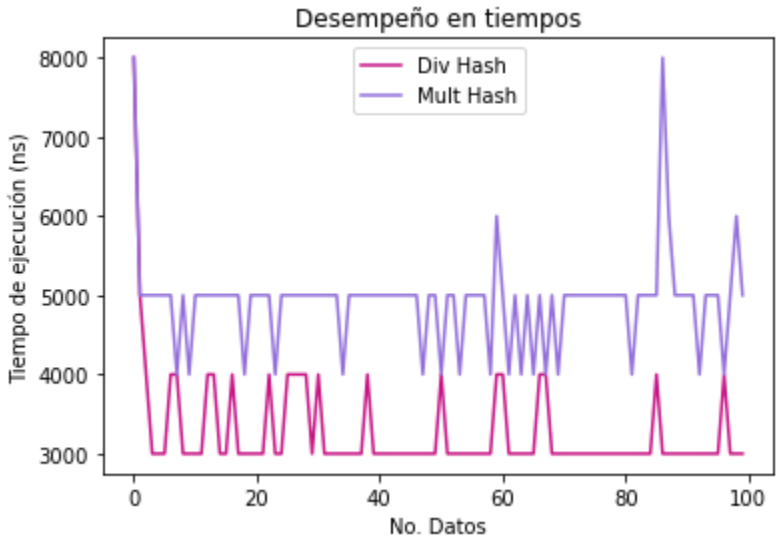
Se insertaron 100 datos en una tabla de hash de tamaño 100. Algunos datos importantes a tomar en cuenta para esta comparación son los siguientes:

- El número de casillas vacías en la tabla donde se utilizó la función de hash por el método de la **división** es de **39**.
- El número de casillas vacías en la tabla donde se utilizó la función de hash por el método de la **multiplicación** es de **37**.

Aquí es cuando se pensó que se podía probar si la función de hash considerada durante todo el desarrollo de la práctica (por el método de división) fue la más adecuada para el tipo de dato que consideramos. De entrada podemos comprobar que el número de casillas vacías que dejó la inserción por el método de la división es mayor que aquel que dejó la inserción por el método de la multiplicación. ¿Que existan más casillas vacías es un indicador de que es mejor? No precisamente. Por un lado tenemos una ventaja: se tienen más lugares disponibles (aunque sean pocos) para seguir insertando datos que puedan llenar esos vacíos en vez de alimentar listas que más tarde perjudicarán el desempeño de las operaciones. Por otro lado, tenemos una desventaja: esas dos casillas vacías representan dos datos que alimentaron alguna lista que ya contenía elementos y esto puede perjudicar el desempeño general. Esta misma lógica puede deducirse del caso contrario. Entonces, ¿podemos decir que una función es mejor que otra? De entrada, no. En las mismas gráficas podemos observar que a simple vista no hay una distribución muy clara de un sesgo en la distribución de los elementos. Por lo tanto, la cantidad de casillas vacías en la tabla no es el indicador ideal para saber si una función de hash es mejor que otra en la implementación de esta práctica. Tendríamos que adentrarnos más a fondo en la distribución de los elementos en la tabla o hacer muchas más mediciones de esta cantidad de casillas vacías para conocer con un poco más de exactitud cuál es la mejor, pero por los tiempos en que se

desarrolló el código no se pudieron hacer más comparaciones y mediciones. También se debe tener en cuenta que las funciones de hash son más efectivas para cierto enfoque dado al tipo de dato. En la misma práctica se menciona algo al respecto en el momento en que se hace referencia a la adyacencia de los datos. Por lo tanto, con más razón es difícil determinar cuál función de hash es mejor para esta práctica. De todas maneras fue interesante la implementación del código y la extensa búsqueda de otros métodos de hash para comparar

VII. Gráfica sobre la comparación del desempeño en tiempos de inserción por dos funciones de hash distintas: la de división y la de multiplicación cuando n = m.



Lo que sí podemos medir es el tiempo de ejecución de inserción para los 'n' datos en la tabla de tamaño 'm' (con n = m = 100) y también es interesante conocer este aspecto para que sea considerado en futuras implementaciones. Podemos ver claramente que el tiempo de inserción de los datos en la tabla por el método de la división fue más veloz que la inserción de los datos por el método de la multiplicación.

- El tiempo promedio de inserción por el método de la **división** es de **3270.0 ns**
- El tiempo promedio de inserción por el método de la **multiplicación** es de **4920.0 ns**

Esta diferencia en tiempos se puede deber a que el método de la multiplicación debe de realizar más operaciones que el método de la división, pues este último solo hace el cálculo ***h(k) = k % m***, mientras que el primer método tiene que hacer **h(k) = math.floor(m*(k*A-math.floor(k*A)))**. Además de que el valor de A se calcula de la siguiente manera: **A = 1/((1+math.sqrt(5))/2)**, donde en el denominador hace el cálculo de la relación áurea. Debe decirse que este puede ser el factor que haga que el tiempo promedio de inserción de los datos por el método de la multiplicación sea mayor.

- **Conclusiones**.

Las tablas de hash son una estructura de datos bastante eficiente si no se desea algo más que almacenar grandes cantidades de datos, puesto que otras operaciones como ordenar, encontrar el mínimo elemento o el máximo no se pueden hacer. En general su desempeño es bastante bueno, pues se mantiene muy cercano a una complejidad de ***O(1)*** si el diseño de la tabla de hash con respecto a la cantidad de datos que se piensan operar en ella. Esta característica se mostró bastante segura: un diseño bueno de la tabla de hash va a mantener muy constantes los tiempos de ejecución para las operaciones. ¿Cuál es el caso más favorable para un buen diseño para la tabla de hash? Cuando el tamaño de entrada de datos es menor o aproximadamente igual al tamaño de la tabla, es decir, en los casos cuando 'n' < 'm' o cuando 'n' ≈ 'm'.

Fue interesante investigar, contemplar y analizar las situaciones consideradas y los resultados obtenidos en esta práctica para obtener un entendimiento del funcionamiento de las tablas de hash en sus generalidades y en sus casos particulares. Si bien en algunos aspectos se investigó más de lo pensado o no se llegó a resultados concretos interesantes para implementarlos de manera formidable en el código, estas investigaciones brindaron un apoyo bastante fuerte para complementar el conocimiento adquirido en esta práctica y utilizar otros conceptos fuera de la misma.

- **Bibliografía**.

Fleck, Margaret. (2008). *Hash Functions*. Recuperado el 4 de Mayo, 2021 de cs.hmc.edu. Sitio web: https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html

Sedgewick, Robert and Wayne, Kevin. (2016). *3.4 Hash Tables*. Recuperado el 24 de abril, 2021 del sitio web: https://algs4.cs.princeton.edu/34hash/#:~:text=The%20most%20commonly%20used%20method,between%200%20and%20M%2D1.

(S/A). (S/A). *Obtengo el error "UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf3 in position 30: invalid continuation byte"*. Recuperado el 20 de abril, 2021 de Stack Overflow. Sitio web: https://es.stackoverflow.com/questions/382281/obtengo-el-error-unicodedecodeerror-utf-8-codec-cant-decode-byte-0xf3-in-po