Tablas Hash
Práctica 3

Equipo #3

Fecha de entrega: 26.11.2021

Integrantes:

- Guillermo Arredondo Renero
- Abraham Guerrero Márquez
- Iñaki Fernández Fiscal
- Luisa Saloma Strassburger

Introducción

El presente trabajo utiliza las implementaciones de la estructura de datos conocida como Tabla de Hash (Hash Table, en inglés), en los lenguajes de programación Java y Python con el propósito de conocer a mayor profundidad su forma de manejar los datos. Asimismo, se desea analizar la eficiencia de las operaciones básicas de estructuras de datos en la Tabla de Hash. Por último, se busca generar conclusiones respecto a las diferencias entre el funcionamiento de Python y Java, sobre todo en relación a las implementaciones de la clase para la estructura de datos.

Una tabla hash, como se ha visto en clase, es una estructura de datos que utiliza llaves (*keys*) asociados a cada dato almacenado. Las llaves de dicha tabla se calculan por medio de una función de hash, lo que da por resultado que las operaciones básicas no sean dependientes de las cantidades de datos almacenados hasta el momento (idealmente). Sin embargo, al depender la posición de los datos de la configuración de la función hash, la eficiencia de la tabla dependerá de otros factores, como se verá a lo largo de esta práctica. Suponiendo una tabla de hash ideal, las operaciones de inserción, búsqueda de los valores hash correspondientes, búsqueda de datos y eliminación de datos se realizan en tiempos constantes (O(1)). Esto es dado que no depende de la cantidad de datos *n*, sino de algún parámetro *k* del que dependa la determinada función de hash.

Para la práctica se implementaron tablas de hash tanto en Java como en Python. A su vez, para la parte de experimentación, se decidió descargar una tabla de datos desde la plataforma Kaggle sobre los atletas que participan en los juegos olímpicos. A pesar de descargar una tabla sobre la información principal sobre los atletas (nombre y apellido, nacionalidad y deporte en que participa) solo utilizamos el nombre y apellido para la inserción, es decir, datos de tipo String o cadena. Debido a que los datos eran de tipo cadena, se escogió la función de hash conocida como SHA256, el cual es un método de encriptación para mensajes alfanuméricos.

Estrategia de trabajo

Debido al éxito obtenido en la práctica anterior a través de la separación de tareas por subequipos. Esta división estuvo basada en las preferencias y cualidades de cada uno de los integrantes del equipo respecto a los lenguajes de programación utilizados, es decir, aquellos que gozan de mayores facilidades para la programación en Python, se dedicaron a ello, mientras que los demás se especializaron en la implementación en Java.

Se decidió que la información sobre obstáculos, retos e ideas sobre cómo implementar las clases, diseñar el experimento y puntualizar problemas se comunicará al otro equipo, ya fuera para discutirlo o a modo de recordatorio. Además, se confirmó que al final del desarrollo de ambas implementaciones se reunieron ambos equipos para poder asemejar ambos trabajos lo más posible y así poder realizar comparaciones más precisas.

Dado a que era necesario trabajar en dos lenguajes de programación distintos, se utilizaron dos repositorios de Replit (uno para cada uno) con acceso por parte de todos los integrantes del equipo. En estos se diseñarán las clases y funciones necesarias para analizar los resultados adecuadamente y después se trasladaron a un IDE apropiado para el manejo de los grandes volúmenes de datos, como son NetBeans y Spider, evitando así el problema de consola de Replit.

Dado a que, como veremos más adelante, las implementaciones en Python involucraron una dificultad menor que el desarrollo en Java, los integrantes del subequipo asignado a Python se involucraron tanto en ambos procesos que al final no fue necesario asemejar los algoritmos.

Planteamiento del Problema

1. ¿Cómo puede generalizarse una tabla de hash?
2. ¿Es cierto que las complejidades de una tabla de hash son constantes?
3. ¿Cómo podemos comprobar que una tabla de hash está funcionando adecuadamente?
4. ¿Cómo podemos comprobar que las implementaciones en Python son, verdaderamente, más versátiles?
5. ¿Qué función de hash puede ser la mejor para asegurar un buen funcionamiento de la tabla de hash?
6. ¿Será posible determinar o aproximar el tamaño propicio para una tabla de hash de *n* datos para su óptimo funcionamiento?

Desarrollo del proyecto

Debido al antecedente teórico de la tabla hash en la clase de Estructuras de Datos, hasta cierto punto ya se conocía la organización y los requisitos de la implementación de la tabla de hash. Así que en el primer día cada uno de los subequipos se dedicó a escribir los algoritmos correspondientes a las implementaciones en Java y Python. En un principio se había diseñado la tabla de hash de forma que automáticamente modificara el tamaño de la tabla buscando regular el factor de carga y, de este modo, asegurar un funcionamiento lo más eficiente posible. Sin embargo, al intentar plantear el experimento se observó que resultaría imposible de llevar a cabo. Una vez aclarado se rediseñaron las

implementaciones (tanto en Java como en Python) para evitar el aumento de la tabla de hash, a su vez que se agregó la posibilidad de instanciar la estructura de datos con un tamaño deseado.

Es importante hacer notar que, incorrectamente, comenzamos por las implementaciones sin conocer los tipos de datos que se utilizarían para el experimento deseado. Esto resultó en la imposibilidad de concluir con el desarrollo algorítmico, ya que se desconocía el factor, en extremo importante, de qué función hash utilizar. Así, el desarrollo de las implementaciones se pausó en lo que se buscaban bases de datos que se pudieran aplicar. Una vez encontrada la base de datos sobre los deportistas en los juegos olímpicos de Tokyo[1], se procedió a utilizar el algoritmo de encriptación SHA256.

SHA-256

Es una de las funciones de encriptación hash más utilizadas, su nombre significa Algoritmo de Hash Seguro de 256 bits (Secure Hash Algorithm) es una variante del algoritmo SHA-2 desarrollado por la Agencia de Seguridad Nacional de los Estados Unidos (NSA) y el National Institute of Standards and Technology (NIST) con el objetivo de mejorar los algoritmos SHA anteriores. La idea detrás de estos algoritmos es la encriptación irreversible de mensajes para propiciar su seguridad. Como se especificó anteriormente, se eligió SHA-256 por su relación con los mensajes alfanuméricos, es decir, de tipo cadena (que es el caso de los nombres de atletas).

No obstante, pueden observarse algunas otras características que la hacen propicia para el experimento diseñado. Por un lado, el algoritmo de SHA-256 genera cadenas de caracteres (mejor dicho de bits) que siempre es de igual longitud (256 bits; 64 caracteres). Para esto se utilizó la clase de Message Digest, el cual alberga los métodos para utilizar los algoritmos de MD5, SHA-1 y SHA-256, principalmente. La idea sobre la implementación de este código se obtuvo de Geeks for geeks[2].

En cuanto a las implementaciones, en java se utilizaron **3 clases**: HashNode<T>, HashTable<T> y un Main para realizar las pruebas y el experimento.

La implementación de la clase HashTable<T> tiene por atributos la longitud deseada de la tabla, el arreglo de almacenamiento principal (en el cual se

[1] La base de datos se puede descargar desde la siguiente liga: https://www.kaggle.com/arjunprasadsarkhel/2021-olympics-in-tokyo

[2] La liga de la página web mencionada es la siguiente: https://www.geeksforgeeks.org/sha-256-hash-in-java/ dicha consulta fue de mucha ayuda para el entendimiento de la clase MessageDigest y su funcionalidad.

agregaran los datos acorde a la posición de la función hash) y un contador de los datos almacenados. En un inicio, se contaba con un cuarto atributo, el factor de carga deseado, el cual era utilizado como bandera para expandir la tabla de hash. Después del cambio en la implementación de la tabla de hash, este atributo resultaba inservible (pues además podía conseguirse su valor desde el exterior conociendo la cantidad de datos y la longitud deseada de la tabla).

Originalmente, la clase HashTable<T> establecia el arreglo denominado por *table* como un arreglo de clase HashNode<T> con la intención de asignar a cada una de las posiciones la dirección del primer nodo enlazado de la lista de encadenamiento separado. Sin embargo, Java marcaba un error en la construcción e instanciación del arreglo de tipo HashNode<T> , el cual no logramos entender. Para solucionar este problema se decidió reformular la estructura y definir el arreglo como polimórfico (de tipo Object, pues tampoco funcionó utilizando el tipo genérico <T> debido a que repetía el parámetro para el HashNode<T> y el procesador de Java se confundía). Esto implicó que cada una de los algoritmos de operación realizara un casteo de tipo las celdas del arreglo hacia un HashNode<T>. Por suerte, como la implementación era únicamente para nuestro manejo, no podía generarse un error de casteo.

Para la implementación de la clase **HashTable<T>** se crearon los siguientes métodos (además de un constructor vacío y uno que recibe el largo de la hashtable):

- size(): devuelve la cantidad de datos guardados en la tabla.
- add(T,double): añade un elemento a la hashtable utilizando la clave calculada con la función de hash (SHA-256).**
- find(double,T): regresa verdadero en el caso de que el elemento buscado se encuentre en la tabla y falso en el caso contrario. Este método no es el utilizado por las demás operaciones para encontrar la posición o nodo en el que se encuentra el dato
- findNode(double,T): regresa el nodo anterior al nodo que contiene el elemento buscado para que de esta forma se puedan eliminar los nodos mediante una reorganización de las direcciones de los nodos en la lista. Esto se debe principalmente al hecho de que los nodos HashNode<T> son lineales y no cuentan con las direcciones del nodo anterior a ellos, sino solo del siguiente.
- delete(T,double): elimina un elemento en caso de encontrarse almacenado en la tabla, dado a que llama al método findNode(double, T) es importante volver a preguntar si el nodo es nulo (caso en el que no haya encontrado al dato).

- promColisiones(): si contiene elementos, regresa el promedio de colisiones de la tabla, es decir, cuenta todas las colisiones existentes en cada una de las casillas del arreglo y las divide entre el tamaño de la tabla creada.
- conteoColisiones(int): cuenta las colisiones de una posición, que después serán utilizadas para la función promColisiones().
- getFullFactor(): devuelve el valor del factor de carga, es decir, qué tan llena está la tabla en relación al tamaño, es importante notar que este factor de carga no necesariamente es el de todos los datos que se desean almacenar, sino, únicamente, de los datos que ya están almacenados hasta ese momento.

** Es importante resaltar que la función de hash en la implementación de Java se encuentra fuera de la clase HashTable<T> debido a los problemas que se podrían generar por la utilización del método MessageDigest que puede regresar una excepción de tipo NoSuchAlgotihmException. Aunque bien esta se pudo manejar mediante un try-catch dentro de la clase, se prefirió utilizar la función de hash desde el exterior y simplemente llamarla dentro del parámetro de la operación deseada. De este modo, era necesario que las funciones add, find, findNode y delete recibieron un segundo parámetro: fnHash(elemento).

La clase **HashNode<T>** fue diseñada para el manejo de las listas por nodos enlazados que se utilizan para el encadenamiento separado. Esta clase cuenta con dos atributos, el dato a guardar dentro del nodo y la dirección de un nodo al cual se encuentra enlazado.

En la clase HashNode<T> se tienen las siguientes funciones:

- getElem(): accede al elemento guardado en el nodo.
- getNext(): regresa la dirección del siguiente nodo.
- setNext(HashNode<T>): asigna una dirección del nodo siguiente.
- setElem(T): asigna un elemento a un nodo.

Por último, la clase **main** se utilizó para leer el archivo excel (convertido a .csv) y de esta forma realizar el experimento, además en esta clase se tienen los métodos requeridos para la obtención y manejo de la función de hash. Dicha clase cuenta con las siguientes funciones:

- getSHA(String): Se llama al método digest, que calcula la dispersión de la entrada String y regresa un arreglo de bytes.
- toHexString(byte[]): convierte el arreglo de bytes a un hexadecimal.
- fnHash(): llamando a las funciones getSHA() calcula la posición que deberá tomar un elemento en la tabla hash.

- randomChoice(): devuelve aleatoriamente la primera celda de una fila escogida con la librería Random.
- createCsv(T[][],String): crea un archivo csv utilizando la matriz con los resultados de los experimentos y un nombre.
- Por último, en esta clase se lleva a cabo el experimento midiendo el tiempo que tardan la búsqueda, inserción y borrado de datos dependiendo el número de datos guardados en la tabla hash.

Para la implementación de **Python** se utiliza **1 clase**: HashTable. Esta implementación tuvo dos versiones, originalmente se consideró que lo que se pretendía con la práctica era la traducción de los algoritmos de Java a Python. Por lo tanto, se desarrollaron las clases Node para generar listas enlazadas. Sin embargo, el equipo se dio cuenta de que esta implementación desaprovechaba el potencial de Python para el manejo de datos asociados a un determinado valor clave y la existencia de listas dinámicas predefinidas. Por lo cual, se decidió reestructurar la clase HashTable de tal forma que sus atributos estuvieran dados por un diccionario de listas, el tamaño de la tabla y un contador de elementos. Esta decisión fue acertada, pues la dificultad de implementación disminuyó notablemente, además debido a la existencia de las listas dinámicas como colecciones, los métodos de conteo de colisiones y búsquedas se desenvuelven con un mínimo de líneas de código. La clase HashTable contiene los siguientes métodos:

- constructor: el cual recibe como parámetro el tamaño total de la tabla, el tamaño esperado para el diccionario, este tamaño se asegura que se mantenga debido a que la inserción de datos en el diccionario se genera siempre a través del módulo del atributo size. Matemáticamente hablando, esto asegura que el resultado de la posición de un elemento siempre sea menor que el dato referencial, en este caso el tamaño deseado de la tabla.
- obtenHash(self,mensaje): recibe como parámetro un tipo de dato, dado a que Python es más versátil, podemos acceder al funcionamiento del algoritmo de SHA-256 por medio de la librería hashlib, el cual recibe el dato en forma de bytes y obtiene su valor hexadecimal y regresa el valor casteado a un entero.
- insert(self,t): recibe como parámetro un dato, del cual se obtiene su valor hash a través de la función obtenHash() y se obtiene la posición de inserción utilizando el módulo respecto al tamaño de la tabla para establecer su posición en el diccionario y añadirlo a la posición, ya sea como primer dato o como una colisión, donde a cada dato en la misma posición añadida, se agrega a una lista referida a esa posición.

    - En el caso en que sea el primer dato mapeado a esa posición, se agrega instancia una lista para esa clave y posteriormente se procede a agregar el dato a la lista dinámica mediante el método append. Es importante notar que la utilización del método append agrega en la parte final de la lista.
- find(self,t): esta función devuelve True si el dato se encuentra almacenado en la HashTable y False en caso de no encontrarlo. En Python no es necesario recorrer expresamente la lista completa, pues es posible preguntar directamente si el elemento está contenido en la colección.
- delete(self,t): este método obtiene la función hash de su dato, se obtiene la posición asociada a dicha llave a través del módulo respecto al tamaño de la tabla. Posteriormente se busca si el elemento está contenido en la lista de dicha clave del diccionario, en caso de sí encontrarse se llama a la función remove(t), propia de las listas para eliminar el dato.
- cuentaColisiones(self): dado a que la tabla está definida por un diccionario es posible iterarlo de tal forma que se sumen las longitudes de las listas ya instanciadas en cada una de sus casillas. Nótese que aquellas posiciones que no hayan recibido datos hasta el momento, no aparecerán como casillas del diccionario y por lo tanto no se corre el riesgo de medir la longitud de un dato de tipo None.
- promedioColisiones(self): este método hace uso del método anterior (cuentaColisiones()) y divide el total de colisiones existentes entre el atributo de tamaño de la tabla. Nótese que el tamaño de la tabla es algo fijo y no depende de los datos existentes hasta ese momento en el diccionario.

Una vez desarrolladas cada una de las funcionalidades de la tabla de hash se procedió a realizar las pruebas de funcionalidad de cada uno de los métodos desde los algoritmos para la encriptación de SHA-256 hasta algunas inserciones y búsqueda de datos.

Después se diseñó el experimento de la siguiente forma:

Experimento

A pesar de ser distintos cada lenguaje de programación la esencia del experimento es la misma: instancias tablas de hash de diferentes tamaños e insertar, buscar y eliminar datos de la base de datos descargada. Se consideró que, debido a que la tabla y los tiempos de corrida serían diferentes para cuando la tabla estuviera vacía, llena o relativamente llena, se obtuviera un promedio de los tiempos de corrida de cada operación. De este modo, para cada tamaño del arreglo, se insertó el primer 30% con respecto al tamaño de la tabla de hash,

posteriormente se rellenó la tabla de hash con todos los datos y finalmente se realizaron búsquedas y eliminaciones de datos aleatoriamente escogidos.

Para la obtención de resultados, en Python se creó un DataFrame con los datos de un archivo de excel llamado Athletes.csv y se realizó el experimento insertando, borrando y buscando en tablas de hash de distintas longitudes. Después se escribió una función (regresaTiempo), que devuelve el tiempo total en segundos que tarda una operación en llevarse a cabo eligiendo, en el caso de la búsqueda y eliminación, aleatoriamente un atleta del dataframe. Una vez obtenidos los tiempos de corrida 30 veces para 500, 1000, 2000, 10000, 11000, 12000, 25000 y 30000 posiciones de la tabla de hash, se sacó el promedio de los tiempos de corrida y se añadieron a una lista para crear un dataframe y graficar los resultados.

En el caso de Java, en lugar de utilizar un DataFrame, se utilizaron matrices de tipo Double para almacenar los tiempos obtenidos a través de una función regresaTiempo similar a la de Python. Esta es la única diferencia con respecto a lo explicado anteriormente.

Asimismo, se decidió que un buen parámetro extra para el análisis de datos sería el conteo de colisiones y su promedio existente para cada uno de los tamaños de la tabla con respecto a la cantidad de datos que se deseaban almacenar.

Dificultades

Java

- Arreglo polimórfico: como se mencionó anteriormente, una de las principales dificultades en la implementación del código de tabla de hash en Java fue la forma en la que se pudiera instanciar un arreglo de clase HashNode<T>. La necesidad de dar atención a los tipos de datos que se utilizan en cada caso y el objetivo de Java de señalar cualquier posible error en la asignación de variables es una de las razones que hace que generar un arreglo polimórfico base de la tabla de hash sea de mayor complejidad.
- Optimización de código para el experimento: debido a la naturaleza estructural de Java, los diseños de experimentos resultan más laboriosos en sintaxis. Además, el registro de los datos en la matriz es un proceso laborioso que podría simplificarse en algún otro lenguaje de programación. Asimismo, el uso de la clase Scanner para la lectura de archivos también involucra un mayor número de líneas de código, es decir, es mucho más compleja.
- Desarrollo del algoritmo SHA-256 como función de hash: el desarrollo de la función de hash en Java fue todo un reto, pues no solamente implicaba la

llamada al método de MessageDigest, sino también implicaba la codificación en UTF-8 y su manejo de cantidad de bits y evitar la pérdida de información y consecuentemente evitar la pérdida de eficiencia.

  - Tamaño de los Integer para la función Hash: como se sabe, todo en la programación cuenta con un límite, la clase de Integer no es una excepción, sino al contrario, existe un tamaño máximo de entero que se puede utilizar. Es por esto que fue necesaria la utilización de la clase BigInteger para la obtención del SHA-256. Asimismo, fue necesario de la utilización de la clase Double para el manejo de los hexadecimales resultantes así como el uso de su variante primitiva para el casteo a un entero que pudiera relacionarse con el módulo del tamaño de la tabla. Esto era de suma importancia para aprovechar la eficiencia de encriptación propia de este algoritmo, pues de no haberlo manejado se habrían generado más colisiones debido a la disminución de caracteres que se podrían tomar al ser casteado a Integer.
  - UTF-8: la codificación estándar de formato en 8 bits, es decir, un byte, que permite a un String ser representado como un arreglo de bytes como se solicita por parte del método digest. Una vez más, el uso de este codificador está justificando en su eficiencia en memoria y tiempo para traducir cada uno de los caracteres del dato en un respectivo byte, lo cual lo hace ideal para la implementación del algoritmo SHA-256.

- Lectura y escritura de csv para datos: dado a que la base de datos utilizada para el experimento fue descargada desde internet, ésta fue trasladada a un archivo tipo csv que pudiera almacenarse dentro del mismo Java. La lectura, no tan complicada, fue realizada a través de la clase Scanner con base en las instrucciones de lectura de archivos en Algoritmos y Programas, donde se procedió a rellenar una matriz de tipo String en la que se almacenaran todos los datos de la base de datos para facilitar su manejo posterior. Por el contrario, la escritura de archivos csv resultó ser más complicada y fue necesario de investigación al respecto. Para esto se desarrolló la función estática createCsv(T[][], String) la cual recibiría las matrices de resultados diseñadas por el experimento y escribiría un archivo de tipo csv que podrían extraerse en forma de tablas de excel para su posterior graficación.

Python

- Dada la naturaleza de python no se encontró un gran problema para poder implementar la tabla hash en dicho lenguaje; sin embargo, si se tuvo

algunos problemas a la hora de decidir qué camino tomar, y es que se podía tanto implementar tanto con un arreglo de numpy, como con un diccionario o una lista. Como se ha explicado anteriormente

Conocimientos adicionales

Debido a las dificultades antes planteadas relacionadas con el desarrollo de la función hash para la implementación de Java fue necesario suministrar una importante cantidad de tiempo en la investigación, entendimiento y adecuación del algoritmo SHA-256 a nuestra implementación de tabla de hash. Fue necesario entender el funcionamiento de la clase Message Digest y cómo utilizarla, especialmente cuestiones relacionadas al tipo de dato que requiere como parámetro y qué regresa (arreglo de bytes).

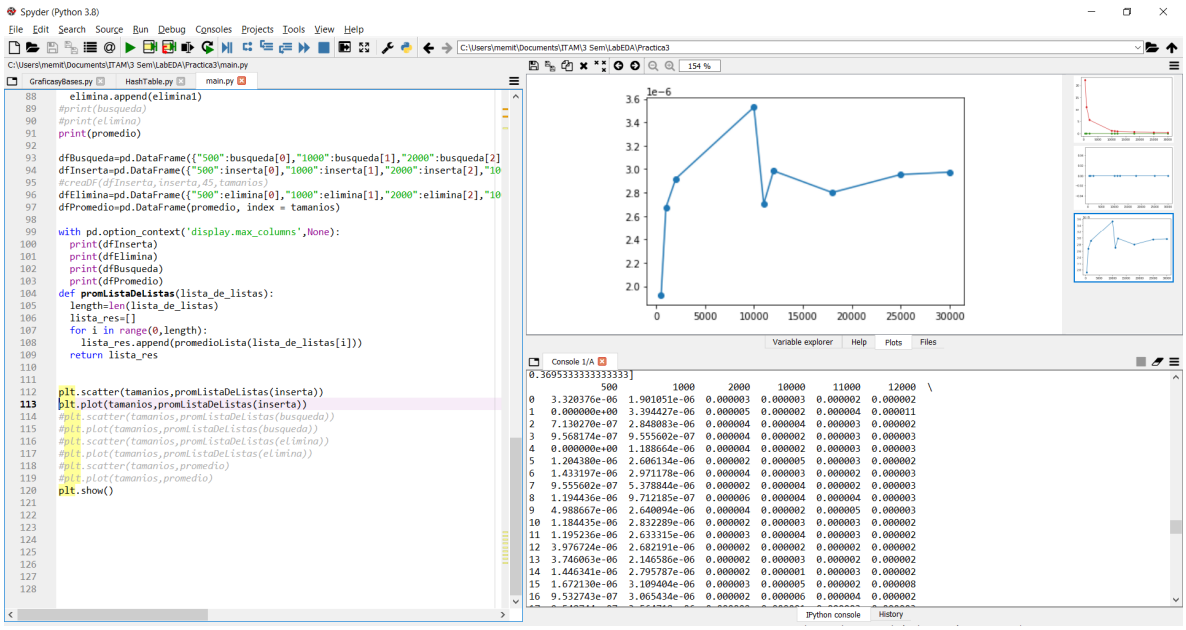
Al mismo tiempo fue necesario investigar un poco sobre el funcionamiento del algoritmo SHA-256, en parte por curiosidad, pero, sobre todo, para poder asegurar que ese era el algoritmos que necesitábamos y nos podría ayudar en evitar el mayor número de colisiones.

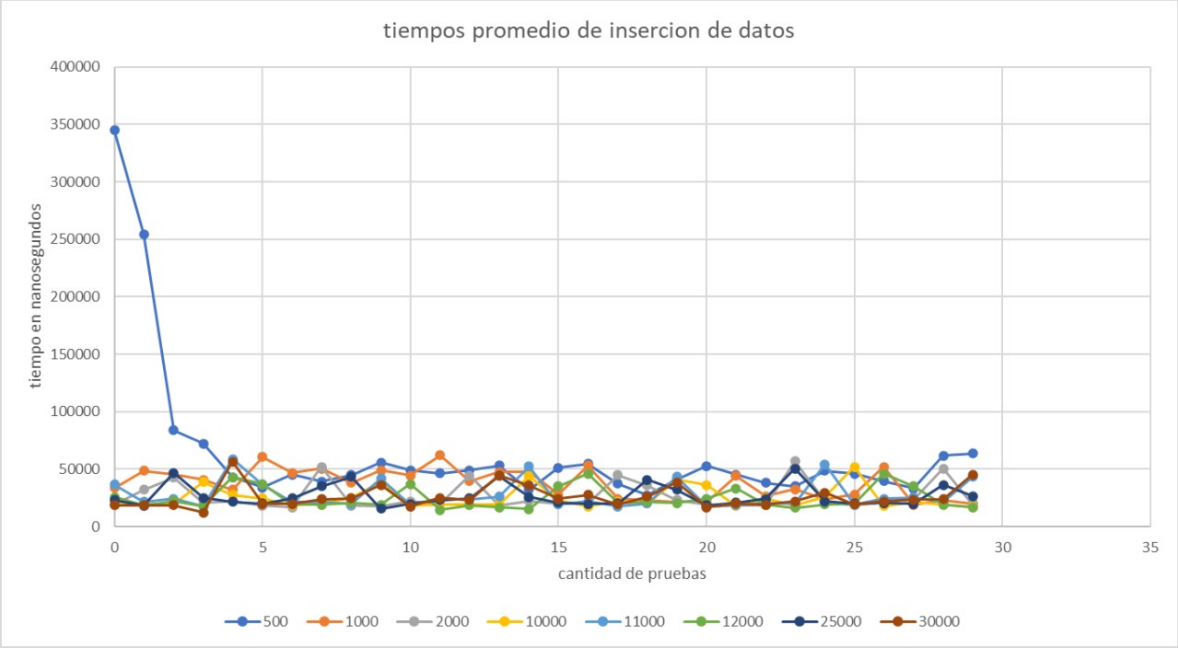
Asimismo, se investigó sobre una forma en la que se pudieran registrar los datos de los resultados obtenidos por el experimento en la implementación de Java de modo similar a lo que se realiza en Python a través de MatPlotLib o Pandas. Se investigó sobre alguna forma en que se pudieran graficar y tabular los resultados, lamentablemente sin ningún resultado. Por lo que se pensó que la más fácil y eficiente representación de los datos fuera a través de matrices y creaciones de archivos de tipo .csv. La escritura del csv es un proceso común de escritura de archivos a través de la clase FileWriter.

Resultados y análisis de resultados

Por razones de espacio y simplicidad para el lector, las tablas de resultados se encuentran anexadas en la carpeta que contiene este documento, es decir, no se incluyen dentro del presente informe.

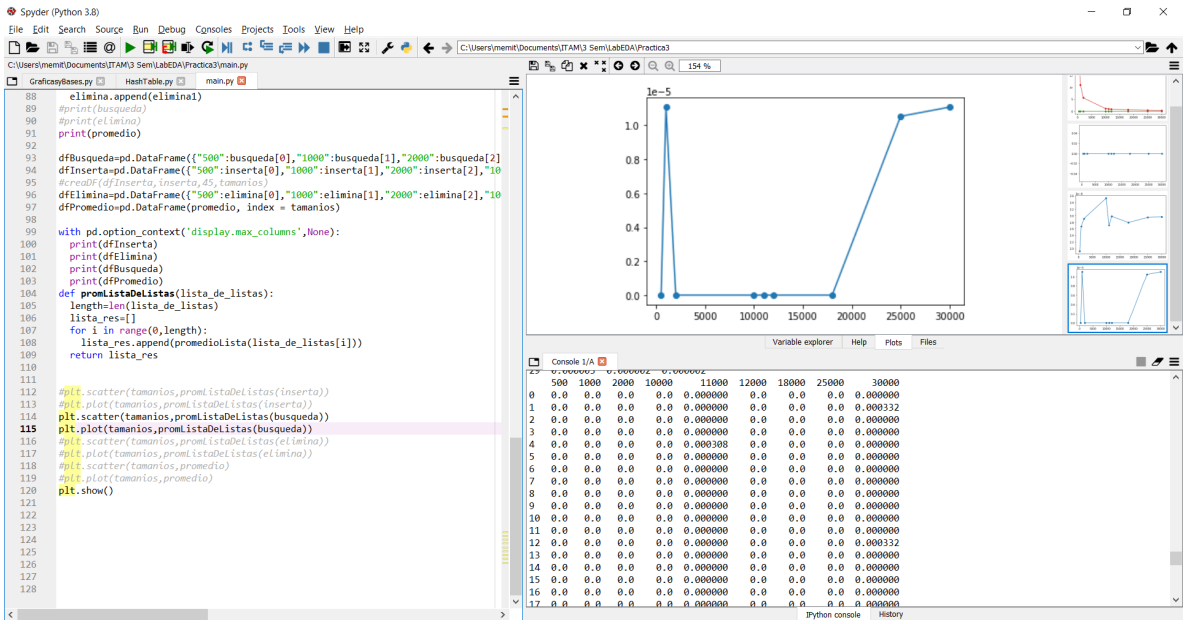
Inserción:

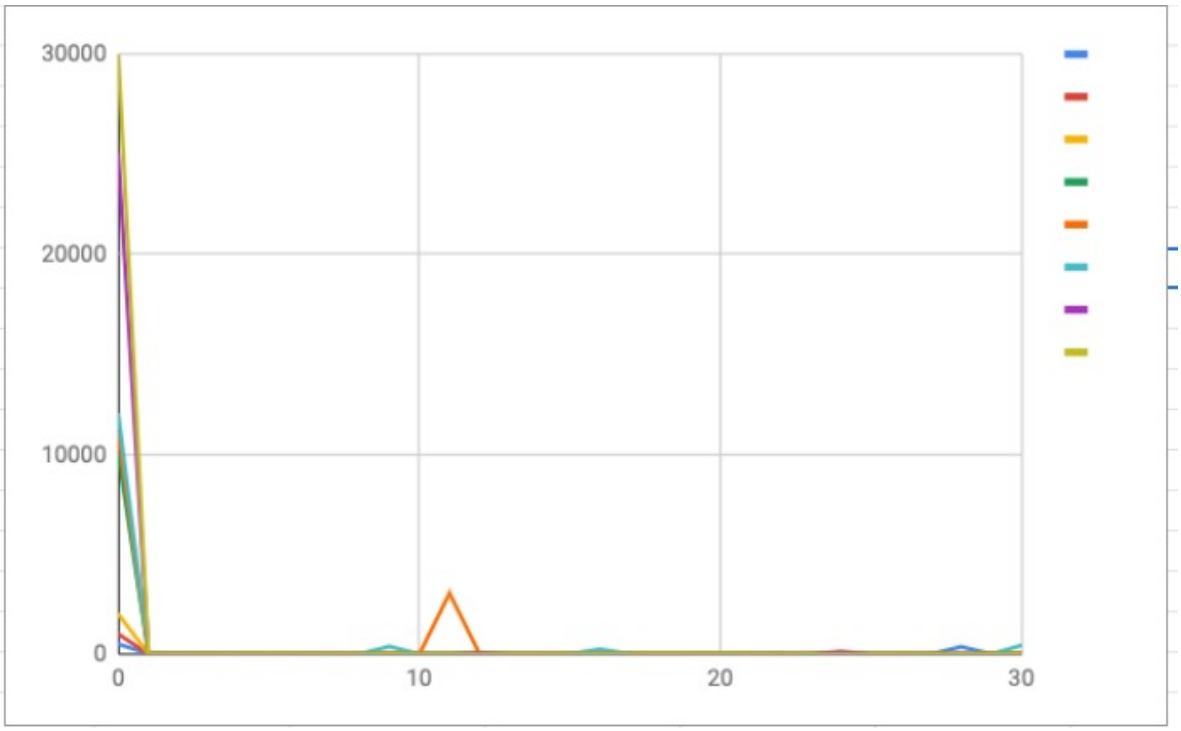




Puede observarse que los promedios de inserción son los que más varían entre ellos, esto se debe a la problemática del método append en las listas de colisiones, puesto que este a diferencia de Java ingresa datos al final de la lista. Sin embargo, nótese que en el caso de haberlos insertado al inicio, el resultado habría sido el mismo por el algoritmo detrás de los métodos de inserción en listas en Python. No obstante, obsérvese que la variación de datos es mínima (entre 2 y 3.3). Por último, nótese que el mayor tiempo se registra en los tamaños de tabla más similares a la cantidad de datos registrados (11,100).

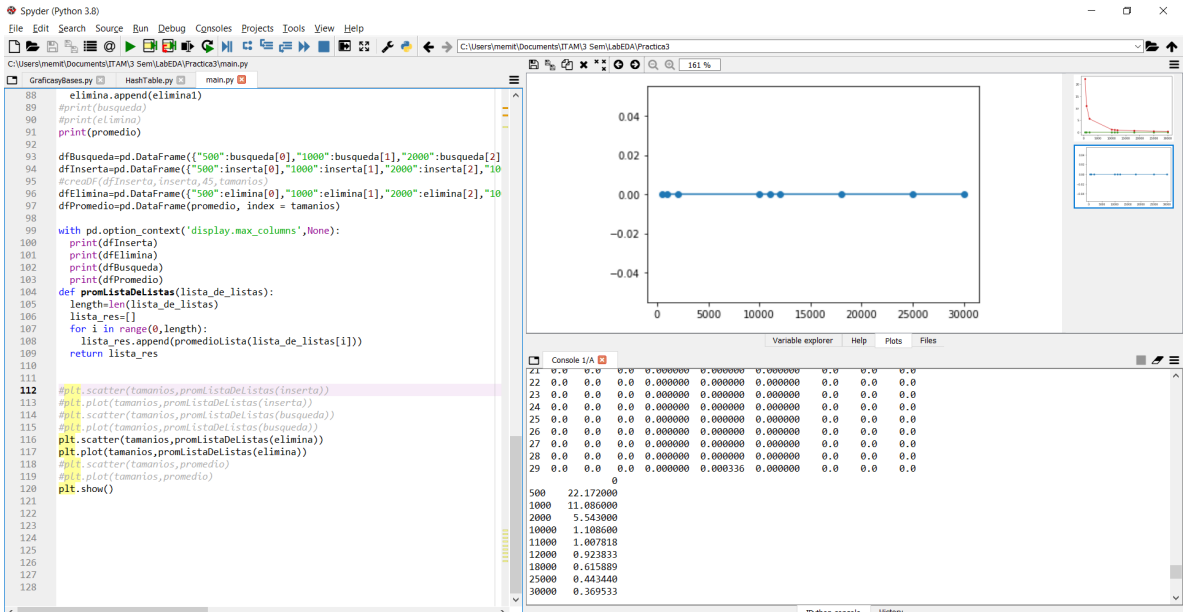
Búsqueda:

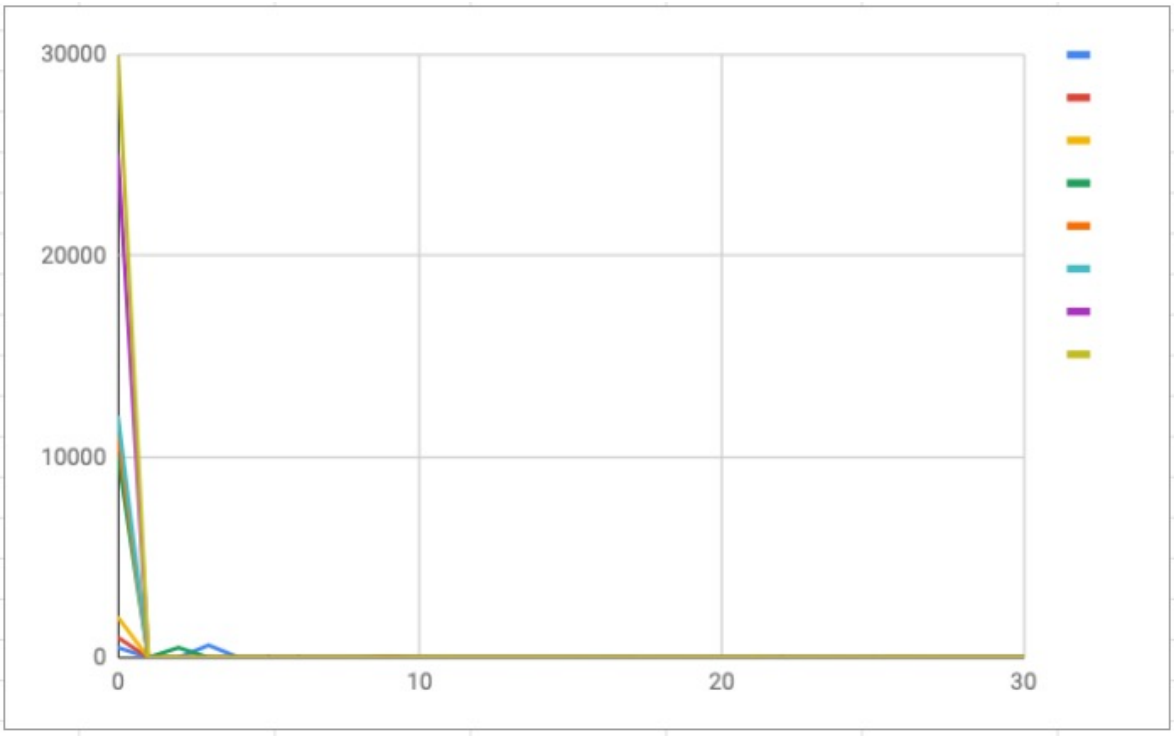




La búsqueda presenta estos “picos” en el tiempo de ejecución debido a que depende de la cantidad de colisiones existentes en cada una de las posiciones. Esto tiene una relación directa con la última tabla mostrada, la cual expresa el número de colisiones por casilla para cada tamaño.

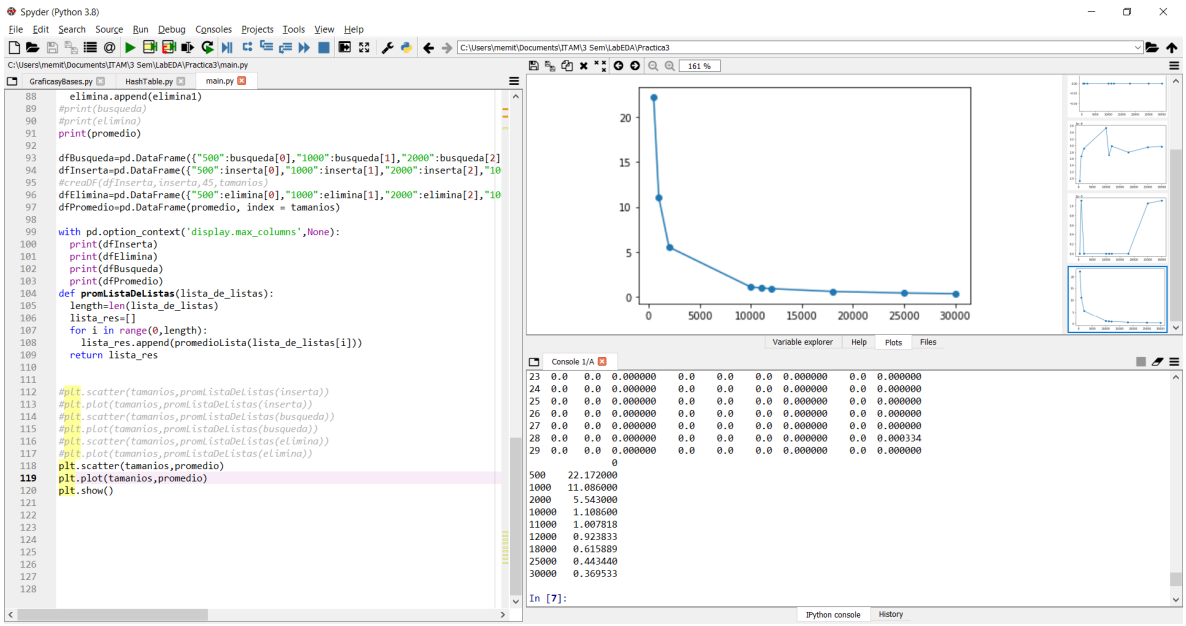
Eliminación:

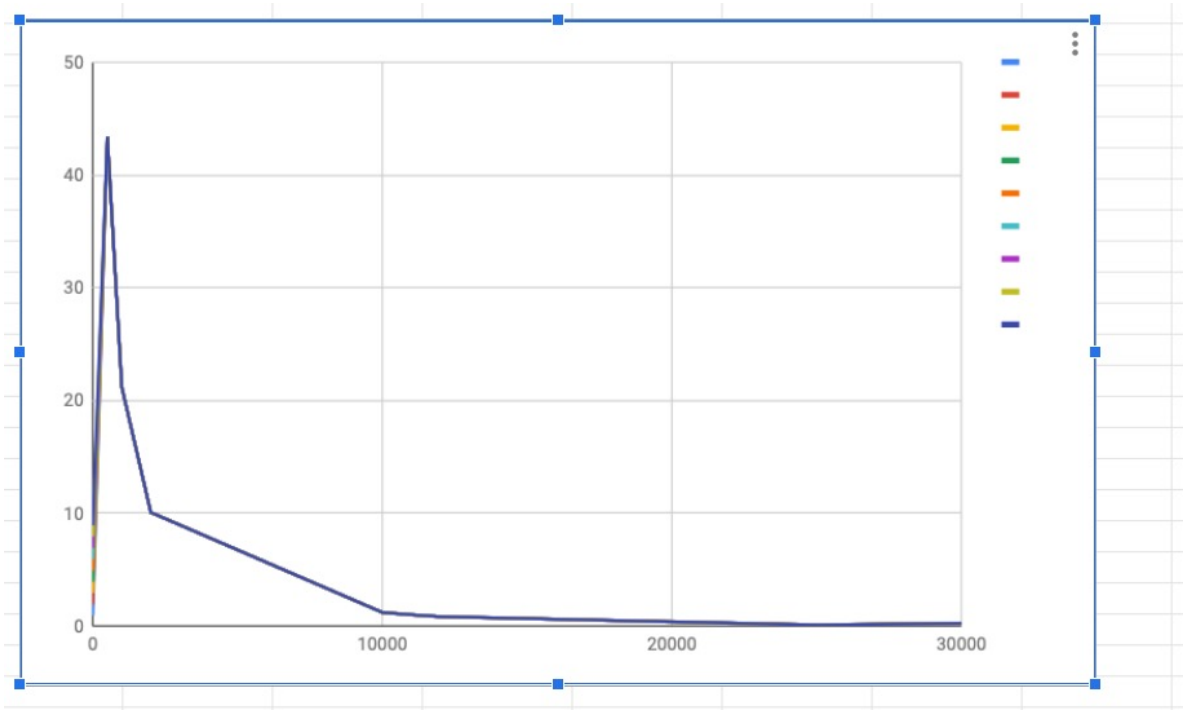




Es importante notar, que el método de eliminación es el más sencillo y directo, pues lo único que hace es quitar de la lista un dato.

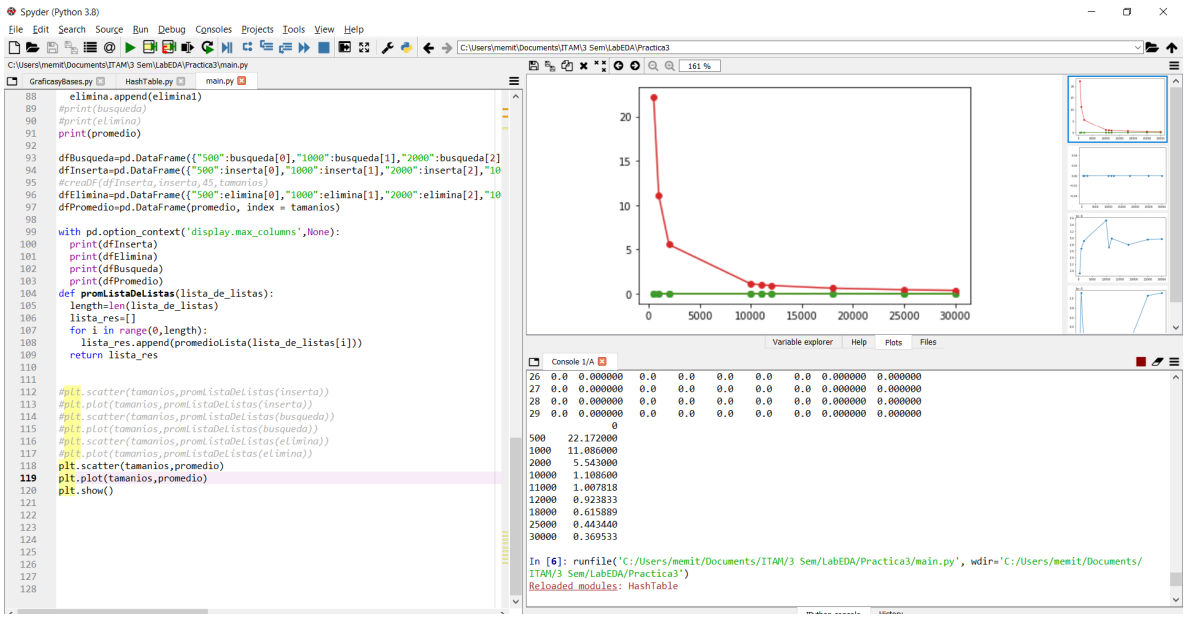
Número de colisiones:





Como puede observarse, cuanto menor sea el tamaño de la tabla con respecto a la cantidad de datos almacenado mayor será el tiempo de ejecución y, consecuentemente, mayor su ineficiencia.

Desarrollo general de las operaciones en Python:



Como se mencionaba anteriormente, a pesar de que las tablas muestran picos y variaciones que parecen muy grandes, es importante notar que esas variaciones en realidad son mínimas (si se observan bien los datos). Por eso, en esta última gráfica que presenta los números de colisiones y el tiempo de las operaciones, las últimas asemejan el comportamiento de una función constante, como se pretendía. No obstante, nótese que este promedio de colisiones se está efectuando con relación al tamaño total de casillas de la tabla, si se quisiera observar la distribución de datos entre las casillas en que se colisiona, los resultados exhibidos para tamaños mayores a la cantidad de datos utilizada serían mayores, puesto que el número de casillas utilizadas en el arreglo principal sería menor.

Contraste entre Java y Python

En este apartado se analizarán y expondrán explícitamente aquellos contrastes encontrados entre la implementación de las tablas de hash y el experimento correspondiente:

- En primera instancia y como era lógico suponer, el desarrollo de código en Python fue mucho más amigable debido a su flexibilidad y simplicidad, propias del lenguaje. Lo cual se puede observar en el número de instrucciones o líneas de código necesarias para cada una de las funciones. Por ejemplo, en el caso de la clase main, el experimento completo de Java incluía un aproximado de 220 líneas de código, 100 más de las utilizadas por Python.

- La propiedad de Python para crear listas con iteraciones directas (conocido como azúcar sintáctico) permitió el registro de resultados de una forma ordenada, sencilla y en menor cantidad de código, resultado muy diferente a Java, donde era necesario instanciar la matriz desde un inicio e iterar en el exterior.
- La implementación de diccionarios y listas como parte de Python fue de mucha ayuda, pues permitía la generación de la tabla de hash de una forma mucho más eficiente y sencilla que en Java, sobre todo, debido a las funciones predefinidas de cada una de estas subcolecciones. Dichas funciones son inexistentes en Java, y el equipo tuvo que emprender un mayor trabajo en su organización y desarrollo.
- En cuanto a la lectura de archivos, cabe destacar que la lectura de datos es incluso más accesible en Python que en Java, dado a que existen clases como DataFrame y Pandas que permiten la eficiencia de lectura y tabulación de datos. Así, la lectura del archivo csv en Python implicó un desarrollo de una función y 2 líneas de código, mientras que en Java fueron necesarias más de 13 líneas de código y el manejo de una excepción.
- El número de clases necesarias para las implementaciones resalta la simplicidad de Python, en la cual se utilizaron dos clases, pero que bien pudieron reducirse a una sola; en cambio, en Java fueron necesarias 3 clases además de una serie de 9 métodos estáticos en el main.
- En Python fue mucho más sencillo la implementación de la función de hash, tanto por la simple necesidad de importar una librería, como de la sencillez del casteo y utilización de los datos recibidos desde el método de SHA-256. También puede observarse la ventaja de seguridad que tiene el código de Python en cuanto a contener el algoritmo de la función de hash dentro de la misma clase de HashTable, característica que no se comparte con la implementación de Java debido a su dificultad de creación y múltiples casos a considerar.
- La propiedad de incertidumbre de tipo de dato en Python también es un factor importante a considerar, lo cual puede observarse, especialmente, en el obstáculo encontrado en torno al desarrollo del arreglo polimórfico. Puede criticarse que, aunque con buena intención, el procesador de Java crea mayores trabas en implementación de tablas de hash que las expuestas por Python. En realidad, este último se muestra muy accesible, pues la mera existencia de un diccionario es propiamente una tabla de hash.
- Finalmente, debe observarse que las corridas del experimento en Python son mucho más veloces de lo que son en Java. Esto puede resultar obvio debido a las características anteriormente mencionadas, desde la longitud

del código hasta su complejidad. Además de la facilidad que tiene Python de forma intrínseca para la optimización de recursos como variables, funciones, colecciones y los casteos entre los diferentes tipos de variable.

Conclusiones

Para concluir, se puede decir, que a pesar de las dificultades que se tuvieron a lo largo del desarrollo del proyecto, se llegó a los resultados deseados, pues en la mayoría de las repeticiones, como se puede observar en el análisis de datos, el tiempo de corrida de los experimentos es generalmente constante.

Puede observarse, que la eficiencia de una tabla de hash se ve disminuida y, de hecho, pierde su característica principal, cuando el número de colisiones es mayor, lo cual se presenta principalmente cuando el tamaño de la tabla es menor a la cantidad de datos que se almacenan. Sin embargo, también se observa que en los métodos de inserción y búsqueda, el tiempo tiende a aumentar cuando el tamaño de la tabla es mucho mayor, esto se debe a que al almacenar *n* datos sobre tablas que tengan mayores o menores posibles posiciones, la tabla de hash pierde su complejidad constante. Por esto, concluimos, que lo más importante para el buen funcionamiento de una tabla de hash es poder mantener en equilibrio la relación entre el tamaño de la tabla y el número total de datos, pues esto tendería a una tabla de hash uniformemente distribuida, que reduciría enormemente el número de colisiones. Para esto, sería importante el establecimiento de un método que permitiera a la tabla de hash controlar la distribución de datos manteniendo el factor de carga relativamente constante; esto solo se podría lograr a través de un método de expansión y redistribución de los valores almacenados.

Referencias

1. Cysae Legal. (febrero 22 del 2018). ¿Qué son las funciones Hash y para qué se utilizan? Recuperado 22 de noviembre de 2021, de https://www.cysae.com/funciones-hash-cadena-bloques-blockchain/

2. Guía 9 ciclo II. Tablas Hash. Recuperado 22 de noviembre de 2021, de https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-con-estructuras-de-datos/2020/i/guia-8.pdf

3. GeeksforGeeks (7 de Agosto del 2019). “SHA-259 Hash in Java”. Recuperado 22 de noviembre de 2021, de https://www.geeksforgeeks.org/sha-256-hash-in-java/

4. Java Platform. Class Message Digest. Recuperado 22 de noviembre de 2021, de https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html

5. Kaspersky daily. (10 de abril del 2014). ¿Qué es un Hash y cómo funciona? Recuperado 22 de noviembre de 2021, de https://latam.kaspersky.com/blog/que-es-un-hash-y-como-funciona/2806/

6. Academy.bit2me . ¿Qué es SHA-256? Recuperado 22 de noviembre de 2021, de https://academy.bit2me.com/sha256-algoritmo-bitcoin/

7. Hubspot. What is UTF-8 Encoding? A Guide for Non-Programmers , escrito por Jamie Juviler. Recuperado 22 de noviembre de 2021, de https://blog.hubspot.com/website/what-is-utf-8