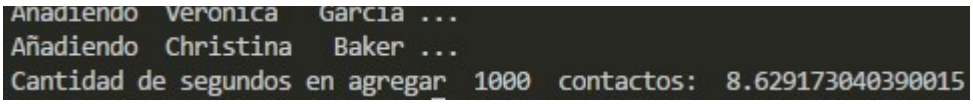
Resultados y análisis de tiempo de ejecución

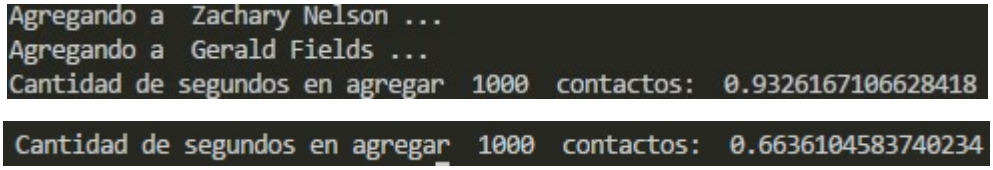
Para analizar la funciones de cada estructura, usaremos el caso al agregar 1000 datos ya que esta cantidad, puede llegar a ser más representativa.

Agregar contactos

Estos son los casos al agregar 1000 contactos a la EDD:



8,629 segundos aprox, que oscila entre 8 y 7 segundos. Para agregar 1000 contactos en una lista.



0,932 segundos aprox. Para agregar 1000 contactos en un árbol binario de búsqueda.

1,544 segundos aprox, cantidad que también puede ser menos de 1 segundo. Para agregar 1000 contactos en un hash.

Para entender lo que está ocurriendo, sabemos que los árboles (independiente del tipo estudiado en cátedra) se destacan dentro de los tipos de estructuras de datos debido a su orden de complejidad bajo, y está no es la excepción. Siendo específico su orden de complejidad promedio, para agregar datos, es de O(log(n)). Esto se puede explicar gracias al proceso de recursividad dentro de la implementación (es una de las razones).

Cabe destacar que, queda muy claro que una lista (para este caso, una lista doblemente enlazada), requiere mucho más tiempo al momento de agregar datos. Considerando el código estudiado, esto se debe a que, al agregar contactos a la libreta, estamos recorriendo toda la lista (cada nodo) para insertar en el último espacio disponible de la lista (considerando que estamos en presencia de un “arreglo dinámico”). Además, para complementar el análisis, antes de implementar el merge-sort (el cual se considera uno de los mejores métodos de ordenamiento, debido a su bajo tiempo de complejidad de su algoritmo), dentro del código, se consideró ordenar la lista mediante el método de bubble-sort. En esa ocasión, al registrar el tiempo de ejecución del método para agregar, este podía llegar muy fácil a los 200 segundos como mínimo, de esta forma queda demostrado que el merge-sort es un método óptimo.

En contraste con el Hash, este considera, un tiempo de complejidad de O(1) en el mejor de los casos y caso promedio, el cual es muy bajo, comparándolo con la lista. En el código, se consideró la primera letra de cada apellido como un valor correspondiente al valor ASCII, el cual, mediante un algoritmo descrito dentro del código, correspondió a un valor Hash, siendo específico, una posición dentro de la tabla Hash. De esta forma, esta EDD, no necesita recorrer toda la tabla Hash,

ya que se toma una key (el apellido del contacto), se codifica y se accede al valor correspondiente a esta llave, así de simple.

Cabe destacar que, el método que se usó frente a las colisiones es óptimo, ya que, al revisar el tiempo de ejecución, este arroja que: al usar una lista para cada valor Hash, dentro de la tabla, este sigue siendo bajo.

Buscar un contacto

Se analizó el tiempo de ejecución buscando el apellido de un contacto dentro de la la EDD.

Se describen a continuación:

```
Buscar Apellido: Bradley
Cantidad de segundos en buscar un contacto:  0.00099992752075195 31
```

0,0009 segundos aprox. Para buscar en una lista con 1000 contactos.

```
Buscar Apellido: Beck
Cantidad de segundos en buscar un contacto:  0.02150130271911621
```

0,021 segundos aprox. Para buscar en un árbol binario de búsqueda con 1000 contactos

```
Buscar Apellido: Lee
Cantidad de segundos en buscar un contacto:  0.0005023479461669922
```

0,0005 segundos aprox. Para buscar en un hash con 1000 contactos.

En este caso, se debe tomar en cuenta, que los tiempos de ejecución arrojados por el programa de Python, pueden variar en un buen rango de tiempo, debido a factores que tienen que ver con los procesos realizados por el dispositivo donde se ejecuta el código. Esta puede ser una explicación para el largo tiempo que toma, en este caso, buscar un contacto dentro del árbol binario de búsqueda. Es necesario explicar esto, ya que no es común, tomando en cuenta que, en el peor de los casos, el tiempo de complejidad para una búsqueda en un árbol sigue siendo de O(log(n)).

Para la búsqueda, la EDD: Hash, es óptima, y esto es por lo mencionado anteriormente. Esta estructura de datos, recibe una id, en este caso, el apellido de la persona que queremos buscar.

Esta key o id, se codifica para luego recorrer la tabla Hash, hasta cuando encontremos este apellido, pero ¿por qué se ve que el tiempo de ejecución para la búsqueda de un contacto en Hash es parecido al de la lista? (Diferencia de 0,0004 segundos = 0,4 milisegundos)

Esto se debe a que, para enfrentar las colisiones, usamos una lista. Al momento en que dentro de la tabla Hash se encuentra un espacio asignado anteriormente, se crea una lista dentro de este “slot”. Esto implica que, en el peor de los casos, en el que todos los espacios de la tabla estén ocupados, se procederá a agregar contactos en muchas listas que contengan los valores de la key y el dato correspondiente a esa key (para este caso, un nodo que contiene la info. del contacto).

Luego al buscar, en el caso menos óptimo, recorreremos en un tiempo parecido al de una lista, ósea, a O(n).

Eliminar un contacto

Tiempo de ejecución eliminando un contacto dentro de la EDD, recibiendo el nombre de este.

Para cada EDD:

```
Borrar Nombre: Eric
Cantidad de segundos en borrar un contacto:  0.0009996891021728516
```

0,0009 segundos aprox. Para borrar en una lista con 1000 contactos.

```
Borrar Nombre: Amanda
Cantidad de segundos en borrar un contacto:  0.0015141963958740234
```

0,001 segundos aprox. Para borrar en una hash con 1000 contactos.