Universidad de Concepción
Facultad de Ingeniería
Dpto. de Ing. Informática y Cs. de la Computación

PROYECTO 2:

Compresión Re-Pair

Estructuras de Datos (2020-1)

Profesor: Diego Seco Naveiras

Carrera: Ingeniería Civil Informática

Ayudantes:

Catalina Pezo Vergara

Alexis Espinoza Rebolledo

Integrantes:

Lucas Kraemer Ananías

Cristián Pérez Vásquez

Fecha de Entrega: 26 de Julio de 2020

Descripción de la tarea

En el presente proyecto, se busca implementar una técnica de compresión de datos conocida como **Re-Pair** (*Recursive Pairing*). La compresión de datos consiste en reducir el espacio que se emplea para representar información. La técnica de **Re-Pair** es parte de la familia de compresión para datos gramáticos.

En la implementación, el programa debe recibir una secuencia de símbolos de entrada, que será una cadena de enteros, y un parámetro **σ** (sigma), donde por cada par de símbolos que se repita una mayor cantidad de veces, serán reemplazados por **σ+1**. Además, por cada llamada recursiva a la secuencia, se aumentará en 1 el valor de **σ** hasta que ya no queden pares de símbolos que se repitan en la misma.

Por ejemplo, si se recibe como entrada la secuencia w = “123233232121212”, en la primera iteración, el par que más se repite es “12”. Este par se reemplazará por **σ+1**, por lo que la secuencia ahora queda como w1 = “(**σ+1**)3233232(**σ+1**)(**σ+1**)(**σ+1**)”. Luego, para la siguiente iteración, el par que más se repite es “32”, el cual se reemplazará por **σ+2**, y así, la secuencia ahora queda como w2 = “(σ+1)(**σ+2**)3(**σ+2**)(**σ+2**)(σ+1)(σ+1)(σ+1)”. Aquí el programa termina, ya que ningún par de símbolos se repite a lo largo de la secuencia.

Finalmente, para implementar este sistema de compresión de datos, se realizarán 2 versiones, una **directa** y otra **avanzada**. Para la versión directa, en cada iteración se guardará cada par de símbolos en la estructura de datos *map*, y aquel par que tenga mayor frecuencia será reemplazado. Por otro lado, para la versión avanzada, se almacenarán punteros que referencien a las distintas ocurrencias de cada par en la secuencia, y se implementará un *MAX-Heap* que contendrá los pares de símbolos y sus respectivas frecuencias. Ambas versiones serán posteriormente detalladas en el presente informe.

Descripción de la solución propuesta

Para ambas versiones la secuencia será almacenada en una lista doblemente enlazada, y se utilizará la estructura de datos *map* de la librería estándar de C++ para facilitar la búsqueda y almacenaje de los pares de símbolos.

Como se mencionó anteriormente, existen 2 versiones del sistema de compresión de datos, las cuales son detalladas en su respectiva sección.

Versión Directa

La **versión directa** del programa se basa en almacenar cada par de símbolos, junto con su respectiva frecuencia, en un *map*. Se realizarán limitadas iteraciones sobre la secuencia de forma recursiva, hasta que no exista algún par de símbolos en el *map* que tenga una frecuencia mayor que 1.

Para implementar esta versión, se crearon clases que cumplen diferentes funciones. Además, se creó la *struct intNode* para facilitar el recorrido de los datos almacenados.

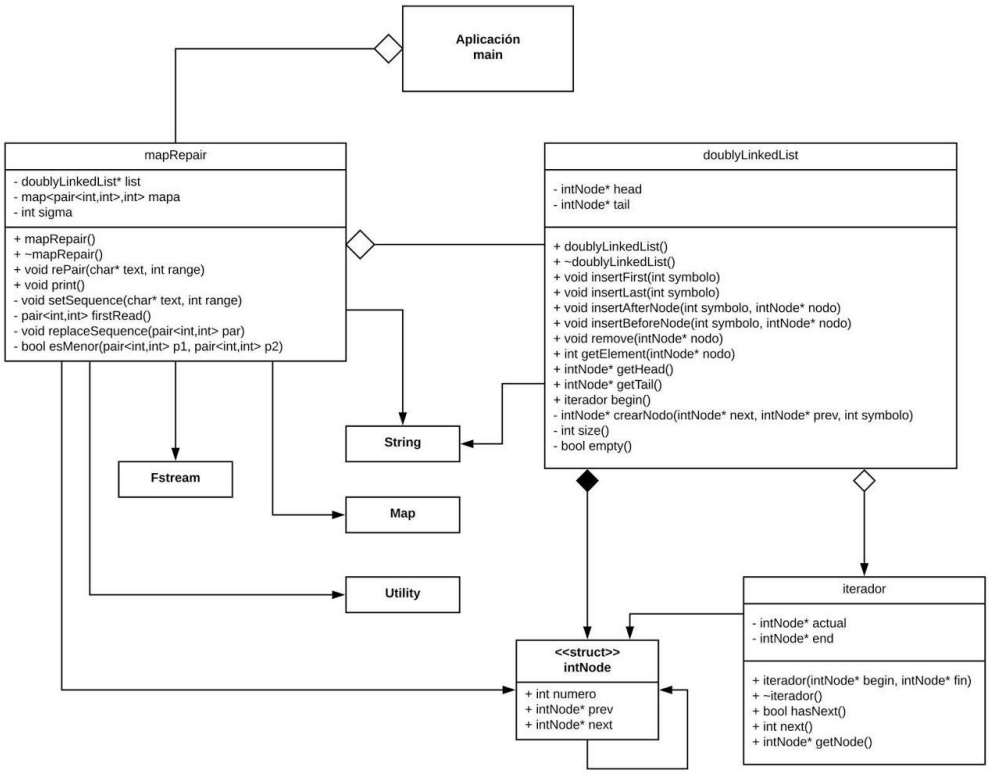
```
4  struct intNode{
5      int numero;
6      intNode* prev;
7      intNode* next;
8  };
```

- **doublyLinkedList:** Corresponde a la implementación de la lista doblemente enlazada, donde cada nodo corresponde a un *intNode* definido anteriormente. Aquí se almacenarán cada uno de los símbolos de la secuencia, incluyendo aquellos que son insertados. La lista permite insertar en la cabeza, insertar en la cola, insertar antes de un nodo o después de él, remover el nodo dado, obtener el elemento del nodo dado, retornar la cabeza, retornar la cola y comenzar el iterador.
- **iterador:** Es el iterador utilizado para recorrer la *doublyLinkedList*. Además, en esta clase está definido el *struct intNode*. Contiene métodos de consulta del siguiente elemento en la lista, y un método que retorna la referencia al *intNode* actual.
- **mapRepair:** Es la clase que realiza el trabajo de comprimir la secuencia dada inicialmente mediante un archivo de texto, y luego comprimir el resultado de cada secuencia resultante de la iteración anterior. Contiene una instancia de *doublyLinkedList* y de la clase *map* de la librería estándar de C++. Este mapa contiene un par de enteros como clave, siendo el par de símbolos consecutivos de la lista, y su respectiva frecuencia como valor; esto es, *map<pair<int,int>,int>*. Sus métodos más importantes son los siguientes:
    - **mapRepair::rePair():** Este método es el encargado de la compresión de texto, llama a *mapRepair::setSequence()* para obtener la cadena a comprimir, a *mapRepair()::firstRead()* para inicializar el map y mientras pueda comprimir, llamará al método *mapRepair::replaceSequence()* para reemplazar los pares.
    - **mapRepair::setSequence():** Este método es para obtener la información del fichero a comprimir e inicializar la *doublyLinkedList* para que la clase *mapRepair* pueda hacer la compresión.
    - **mapRepair::firstRead():** Este método lee la secuencia de pares de la *doublyLinkedList*. Además inicializa un map que se utilizará para mantener registro de la frecuencia de cada par. Una vez finalizada la

lectura de la lista, se buscará en el *map* el par de mayor frecuencia y se retornará. En caso de no existir un par que se repita se retornará un par especial (el cual no existe en el archivo a comprimir).

- **mapRepair::replaceSequence():** Este método lee par a par la *doublyLinkedList*. En caso de encontrar el par de mayor frecuencia, éste será reemplazado y se continuará con la lectura hasta el fin de la secuencia.

Así, la versión directa se resume en el siguiente diagrama de clases:



Versión Avanzada

La **versión avanzada** consiste en utilizar punteros para determinar la primera y última ocurrencia del par en el *map*, el cual estará relacionado con el *MAX-heap* que indicará la frecuencia de éste. Posteriormente del *heap* se determinará el par más frecuente, el que se procederá a reemplazar, actualizando los punteros acordes para mantener este sistema de trabajo actualizado (cuando se elimina un par y cuando se agrega otro).

Para implementar esta versión, se crearon clases que cumplen diferentes funciones. Además, en esta versión, también se crearon *structs* para organizar mejor los punteros y los datos que cada uno de los pares contiene:

```
4  struct pointerIntNode{
5      int numero;
6      pointerIntNode* prev;
7      pointerIntNode* next;
8      pointerIntNode* anteriorIgual;
9      pointerIntNode* siguienteIgual;
10 };
```

```
6  struct infoNodo{
7      pointerIntNode* primerIgual;
8      pointerIntNode* ultimoIgual;
9      int posHeap;
10 };
```

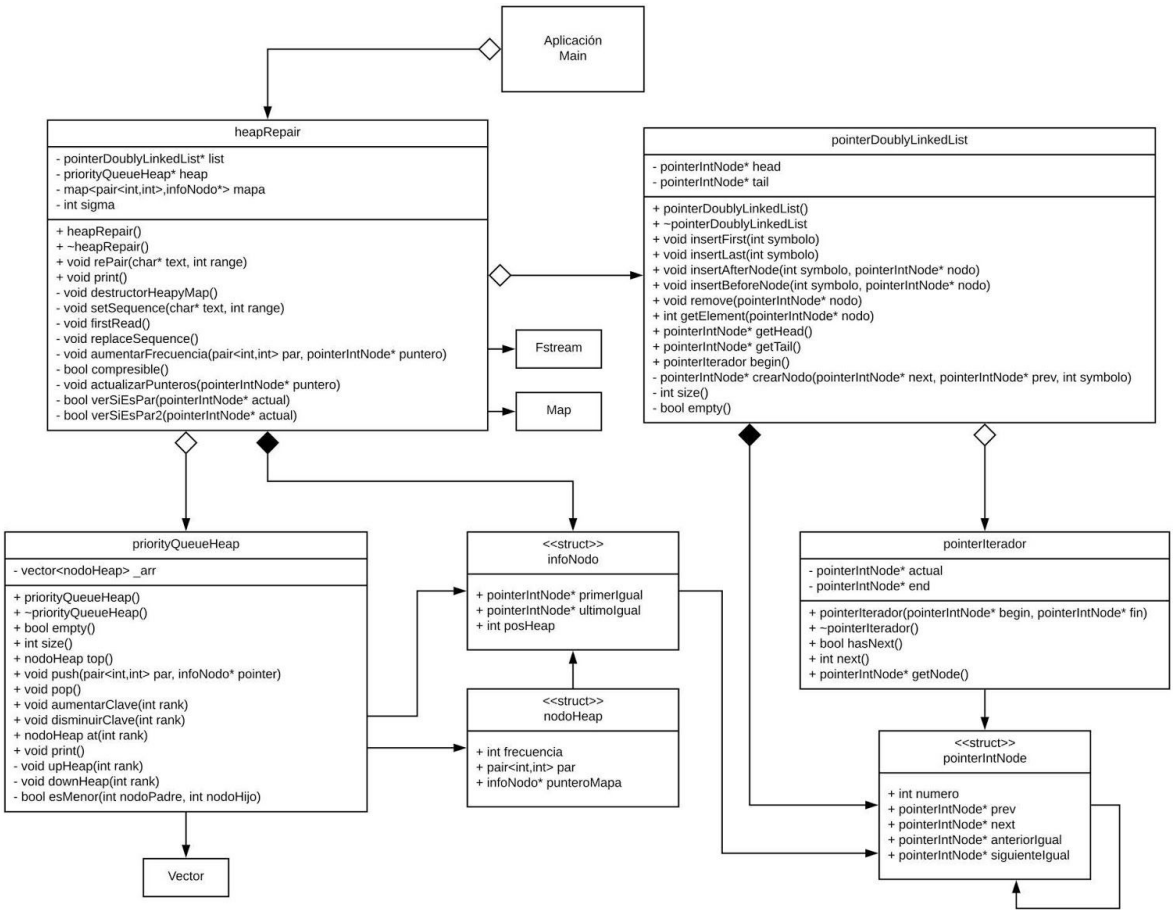
```
12 struct nodoHeap{
13     int frecuencia;
14     pair<int, int> par;
15     infoNodo* punteroMapa;
16 };
```

- **pointerDoublyLinkedList:** Corresponde a la implementación de la lista doblemente enlazada, la cual almacena todos los elementos de la secuencia dada inicialmente, y también los elementos que van siendo reemplazados. Así, la lista se actualiza cada vez que se reemplaza algún par por el sigma correspondiente. Cada nodo corresponde a un *pointerIntNode* definido anteriormente. La lista permite insertar en la cabeza, insertar en la cola, insertar antes de un nodo o después de él, remover el nodo dado, obtener el elemento del nodo dado, retornar la cabeza, retornar la cola y comenzar el iterador.
- **pointerIterador:** Es el iterador que se utiliza para recorrer la *pointerDoublyLinkedList*. Además, aquí se define el *struct pointerIntNode*. Contiene métodos de consulta del siguiente elemento en la lista, y un método que retorna la referencia al pointer*IntNode* actual.
- **priorityQueueHeap:** Es la implementación del *MAX-heap*, utilizando un vector de la librería estándar de C++ de tipo *nodoHeap* (*vector<nodoHeap>*), definido en el archivo header homónimo (mostrado anteriormente). Aquí, la clave de cada nodo es la frecuencia y, como valor, el par correspondiente a dicha frecuencia. Cada nodo del *MAX-heap* tiene su respectiva referencia al *map* que es instanciado en la clase *heapRepair*. El *heap* puede insertar, remover y consultar elementos, entre otras funciones, haciendo *upHeap* o *downHeap* según corresponda. Además, el *heap* puede modificar su clave, aumentándola o disminuyéndola dependiendo del caso.
- **heapRepair:** Es la clase que realiza el trabajo de comprimir la secuencia iterativamente. Contiene una instancia de *pointerDoublyLinkedList* y *priorityQueueHeap.* Además, crea instancias de la *struct infoNodo*, la cual almacena los pares actuales leídos en caso de que estos no se hayan almacenado anteriormente. Por cada iteración, se modifican los punteros de los nodos de la *pointerDoublyLinkedList*, los cuales apuntan a la anterior y siguiente ocurrencia del par en la lista. Además, contiene una variable privada de tipo *map* de la librería estándar de C++, que contiene un par de enteros como clave y un

puntero a *infoNodo* como valor, es decir, *map<pair<int,int>,infoNodo*>*. Por cada iteración, también se modificarán los punteros *primer*Igual y *ultimoIgual* de cada *infoNodo* del mapa en caso de ser necesario.

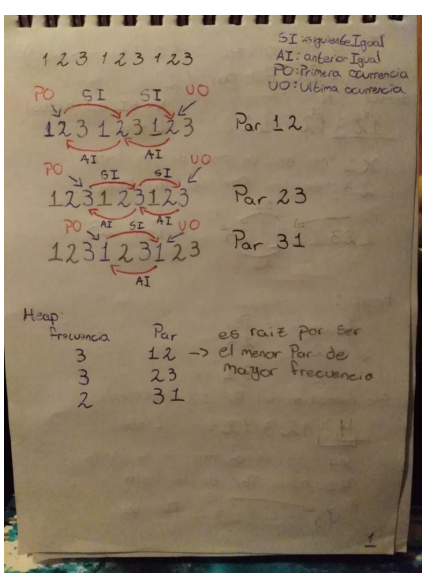
- **heapRepair::setSequence():** Este método es para obtener la información del fichero a comprimir e inicializar la *pointerDoublyLinkedList* para que la clase *heapRepair* pueda hacer la compresión.
- **heapRepair::firstRead():** Este método lee la secuencia de la *pointerDoublyLinkedList*, inicializando tanto el map como el *MAX-heap*, que se utilizarán y modificarán posteriormente en caso de que se pueda comprimir la secuencia de pares.
- **heapRepair::replaceSequence():** Este método obtiene el par más frecuente del *heap* y obtiene la primera ocurrencia de éste, por lo que se reemplazara y actualizará la información (punteros) hasta que se reemplace la última ocurrencia de este par. Durante la compresión, se actualizan los punteros de tal forma de mantener la información actualizada para poder seguir la compresión del archivo.
- **heapRepair::compresible():** verifica que la *pointerDoublyLinkedList* se pueda comprimir, incluyendo el caso en el que esta se encuentre vacía.
- **heapRepair::repair():** Este método es el encargado de la compresión de texto, llama a *heapRepair::setSequence()* para obtener la cadena a comprimir, a *heapRepair::firstRead()* para inicializar el *map* y el *MAX-heap.* Posteriormente se determinará si la secuencia de pares se puede comprimir utilizando *heapRepair::compresible()* y, en caso de ser compresible, se llamará al método *heapRepair::replaceSequence()* para reemplazar los pares respectivamente. Finalmente se borrará la información que pueda existir tanto en el *map* como en el *MAX-heap*.

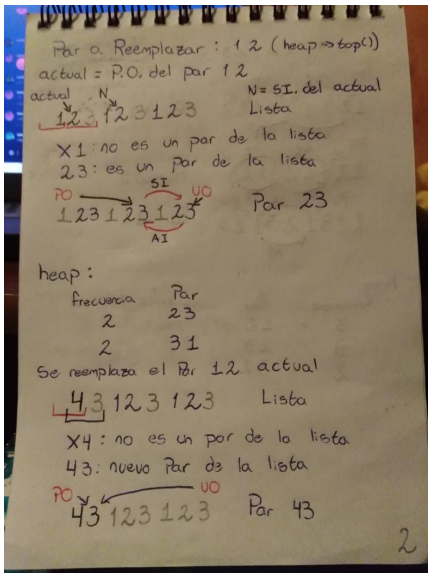
De esta forma, la versión avanzada del proyecto se resume en el siguiente diagrama de clases:

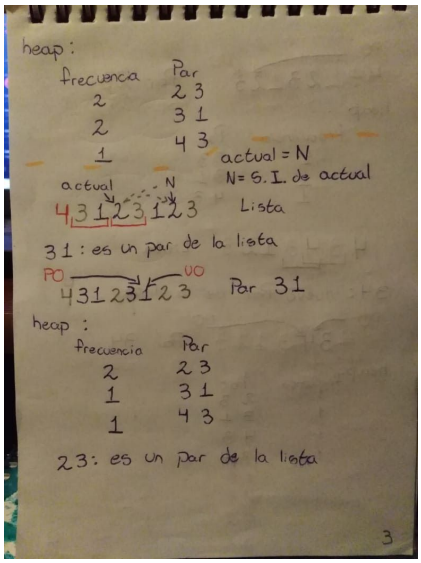


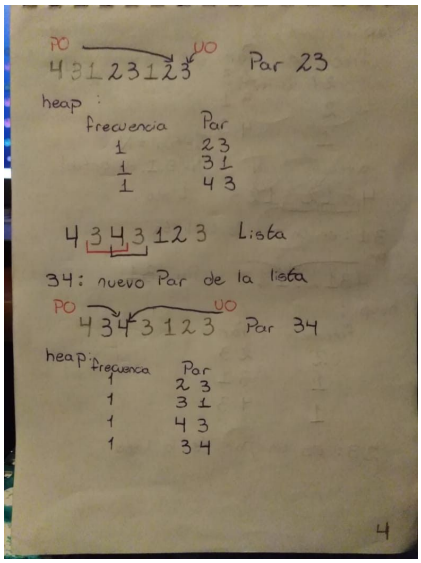
Detalles de Implementación

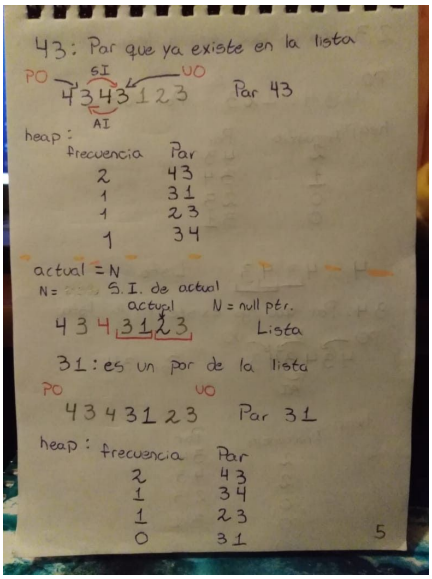
En las siguientes imágenes se muestra cómo se van actualizando las estructuras en cada paso; específicamente cómo se va modificando el *map* y el *MAX-heap,* en relación a cada par cuando este se tenga que agregar o eliminar de la lista.

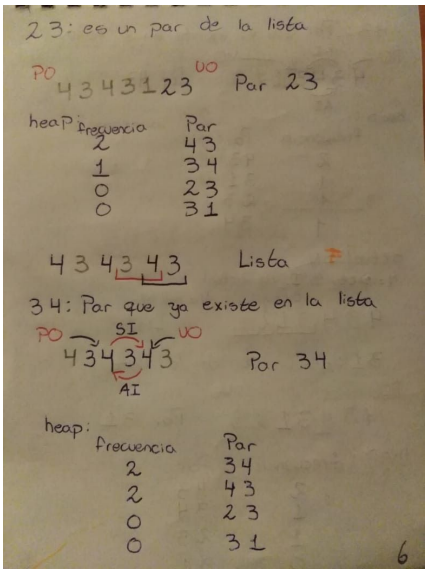


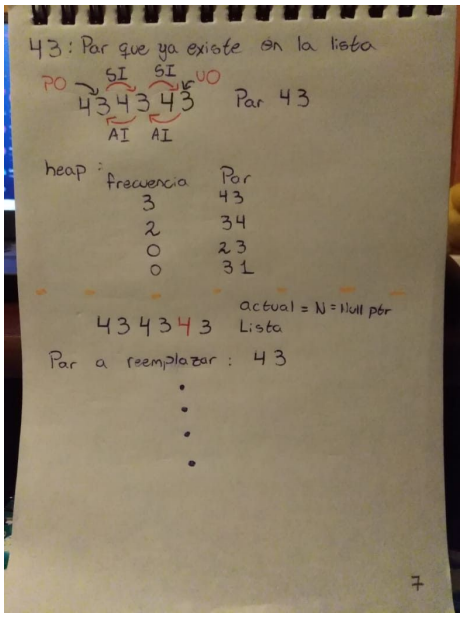








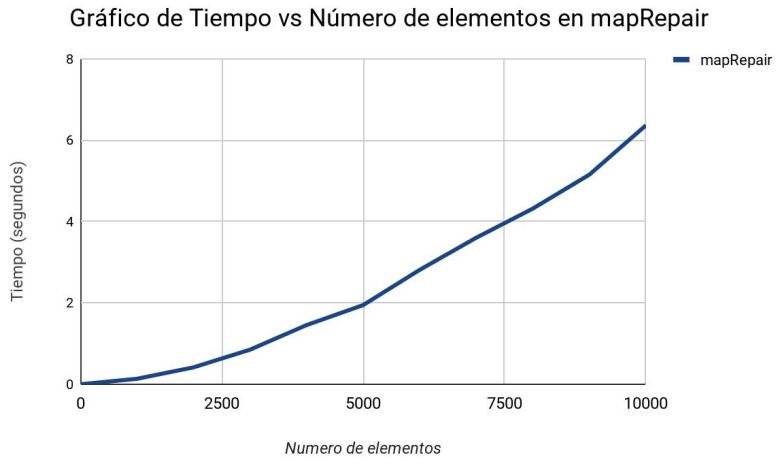


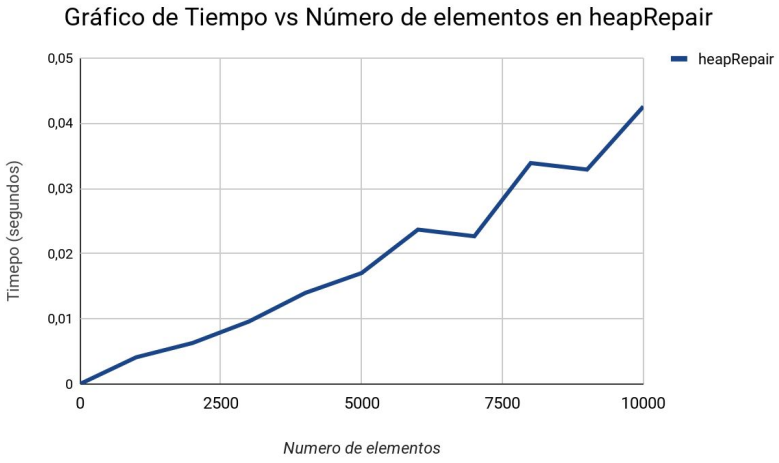


Así, continuará el proceso iterativamente hasta que ya no existan pares de símbolos con frecuencia mayor que 1 en el *heap*.

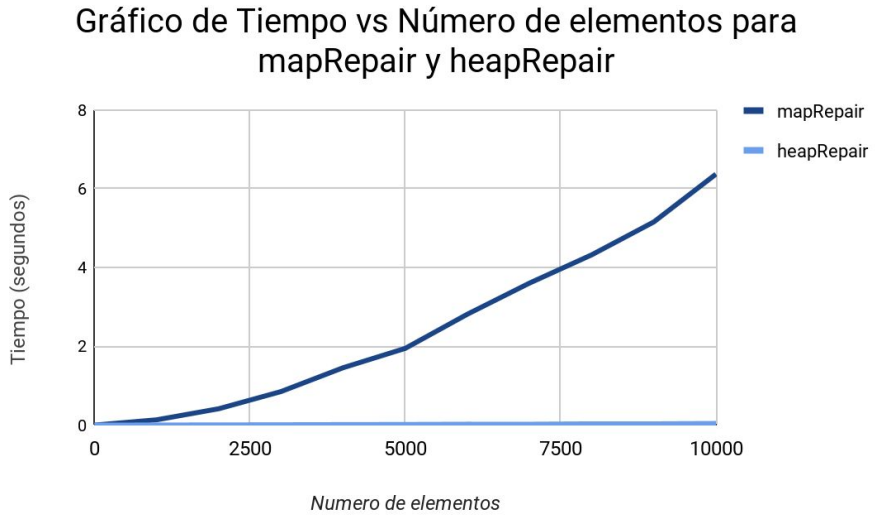
Rendimiento y conclusiones

Una vez implementadas las versiones directa y avanzada del sistema de compresión, se procedió a analizar el rendimiento de tiempo de cada uno de los algoritmos. Ambos se evaluaron con secuencias de 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 y 10000 símbolos aleatorios. Los resultados son mostrados en los siguientes gráficos:





Comparativamente, ambos algoritmos de compresión se aprecian en el siguiente gráfico:



Así, a modo de conclusión, se logró generar un sistema de compresión con dos algoritmos distintos cumpliendo el mismo objetivo, observando que el algoritmo de *heapRepair* es más eficiente que el *mapRepair* en cuanto al tiempo que utiliza para comprimir correctamente la secuencia inicial dada. Además, respecto a la forma de los gráficos, se infiere que, dada la curva de *mapRepair*, este algoritmo es de tipo **exponencial** y que el algoritmo de *heapRepair* es de tipo **n(log n)**. Esto refleja correctamente la creencia teórica que se sostenía respecto al rendimiento de estos algoritmos.

Por otra parte, estos gráficos fueron probados con secuencias de números aleatorios entre 1 y 27, sin tener 2 números iguales consecutivos, ya que al considerar varios grupos de éstos, la complejidad de desarrollo del algoritmo de *heapRepair* aumentaba mucho. Respecto a esto, se conversó con el profesor y uno de los ayudantes de la asignatura, y se llegó al acuerdo de que este caso será ignorado para su revisión. Esto indica que la compresión no funcionará para todos los tipos de entradas posibles.

Referencias

1. C++ Reference. (2014) “map - C++ Reference”, Recuperado de: http://www.cplusplus.com/reference/map/map/
2. GeeksforGeeks. (2017) “Binary Heap - GeeksforGeeks”, Recuperado de: https://www.geeksforgeeks.org/binary-heap/