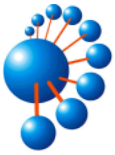


Universidad de Concepción
Facultad de Ingeniería
Departamento de Ing. Informática y
Cs. de la Computación

Informe Proyecto 2: Re-Pair

Estructuras de Datos (503220-1)
Dr. Diego Seco

Alumnos:

Nicolás Araya

Martina Cádiz

Ayudantes:

Alexis Espinoza
Catalina Pezo

Fecha:

26/07/20

Descripción de la tarea

El proyecto consiste en llevar a cabo dos técnicas de compresión de datos, en otras palabras es poder reducir el espacio empleado para lograr representar grandes volúmenes de información. El objetivo es poder demostrar que existen técnicas más eficientes que otras, esto depende de las estructuras de datos que se implementen en cada solución y los casos que se analicen. El criterio que se utilizará para evaluar la eficiencia será: el tiempo de ejecución utilizado por cada solución para hacer la compresión.
La primera será llamada "Repairv1" y esta deberá ser implementada con la que la solución avanzada, la cual será catalogada como "Repairv2".

Las cadenas a reducir serán de números enteros y para poder llevar a cabo la compresión se considerarán los pares de números, será necesario que estos aparezcan mínimo dos veces en la cadena a reducir. Es decir, la frecuencia de aparición de cada par debe ser superior a 1.

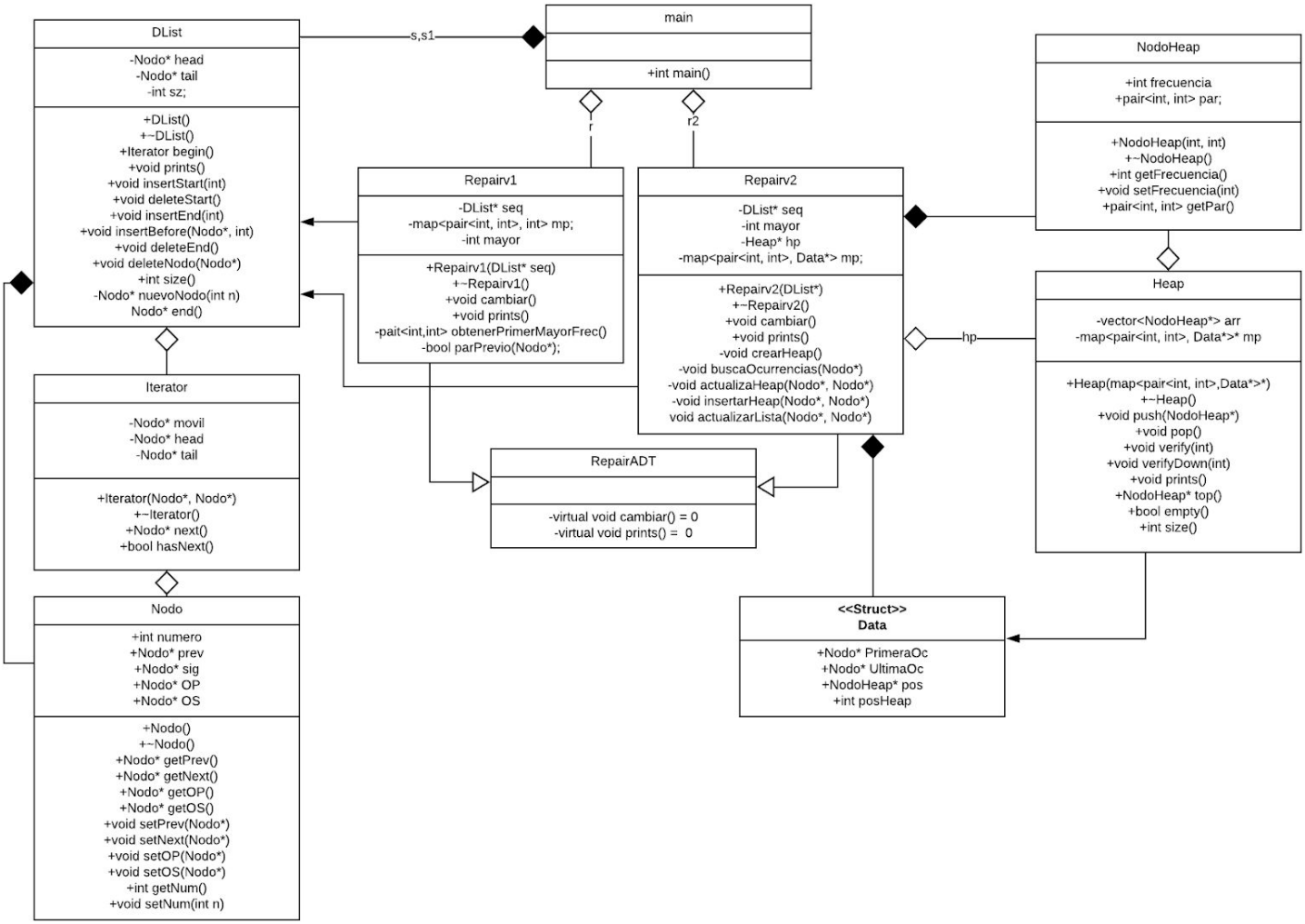
Descripción de las soluciones

Ambas soluciones están basadas en la recursividad, ya que se realizarán varias pasadas sobre la secuencia para poder entregar la respuesta final. Además la cadena será almacenada en una lista doblemente enlazada, a raíz de que esta brinda flexibilidad para acceder a los elementos que posee. Las siguientes descripciones son a grandes rasgos:

Repairv1, primero realiza una lectura lineal de la lista, con la finalidad de obtener el par con mayor frecuencia. El valor del mapa corresponderá a la frecuencia de cada par. Luego realiza una segunda lectura a la lista completa, para modificarla y así ir cambiando el par anteriormente seleccionado como el mayor. Este ciclo deberá ser repetido múltiples veces, ya que al finalizar la modificación del par escogido se limpiará el mapa. La solución finalizará cuando el par mayor corresponda a uno con frecuencia menor o igual a 1.

Repairv2, lo primero que hace es una lectura lineal de la lista con la finalidad de construir el heap. El orden del heap será determinado por la frecuencia de cada par y almacenará datos de tipo NodoHeap. En esta solución, el valor del mapa corresponderá a una struct llamada "Data" y entregará información extra de cada par. Luego se llamará a una función recursiva, que accederá directamente a todas las ocurrencias que posea el par, cabe destacar que no se volverá a recorrer la lista ya que se utilizarán punteros a las ocurrencias previas, siguientes, última y primera. A medida que se modifica y actualiza la lista, el heap, el mapa y los punteros de cada nodo también sufrirán cambios. El par a modificar será siempre el que se encuentre en el top del Heap. La solución termina cuando el par asociado al top del heap posea frecuencia menor o igual a 1.

Diagrama de clase



Descripción de los métodos de cada clase
(*): métodos privados

1. **Clase Nodo:** Esta clase sirve para la implementación de la clase DList. Para ello cada objeto de la clase Nodo contiene como atributos privados, una entero "numero", el cual lo representa en la lista, una referencia al Nodo previo "prev", al Nodo siguiente "sig". Además las referencias a la ocurrencia previa "OP" y la ocurrencia siguiente "OS", las cuales son utilizadas solo por la clase Repairv2.
   a) **Nodo():** Método constructor de la clase.
   b) **~Nodo():** Método destructor de la clase.
   c) **Nodo* getPrev():** Retorna la referencia al Nodo que le antecede en la DList.
   d) **Nodo* getNext():** Retorna la referencia al Nodo que le sigue en la DList.
   e) **Nodo* getOP():** Retorna la referencia al Nodo del primer par de la ocurrencia previa.
   f) **Nodo* getOS():** Retorna la referencia al Nodo del primer par de la ocurrencia siguiente.
   g) **int getNum():** Retorna el valor del número entero que lo representa en la lista.
   h) **void setPrev(Nodo*):** Permite modificar la referencia del Nodo previo.
   i) **void setNext(Nodo*):** Permite modificar la referencia del Nodo siguiente.
   j) **void setOP(Nodo*):** Permite modificar la referencia a la ocurrencia previa.
   k) **void setOS(Nodo*):** Permite modificar la referencia a la ocurrencia siguiente.
   l) **void setNum(int):** Permite modificar el número que lo identifica.

2. **Clase DList:** Esta clase es utilizada para la implementación de una lista doblemente enlazada. Sus atributos privados son un puntero al principio de la lista "head", uno al final de la lista "tail" y un entero "sz" que corresponderá al tamaño de la lista.
   a) **DList():** Método constructor de la clase. Inicializa los punteros "head" y "tail".
   b) **~DList():** Método destructor de la clase.
   c) **void insertStart(int):** Permite insertar en la primera posición de la lista.
   d) **void deleteStart():** Elimina el primer elemento de la lista.
   e) **void insertEnd(int):** Permite insertar al final de la lista.
   f) **void deleteEnd():** Elimina el último elemento de la lista.
   g) **void insertBefore(Nodo*, int):** Permite insertar en la posición anterior al Nodo dado.
   h) **void deleteNodo(Nodo*):** Elimina el Nodo entregado.
   i) **int size():** Retorna el tamaño de la lista.
   j) **Iterator begin():** Retorna un Iterador iniciado en la primera posición de la lista.
   k) **void prints():** Imprime cada elemento de la lista en orden.
   l) **Nodo* nuevoNodo(int)(*):** Es un método privado llamado por los métodos c), e) y g). Retorna un nuevo nodo con el número asignado y todos sus punteros en NULL, para que luego cada método los asigne según corresponda.
   m) **Nodo* end()(*):** Retorna la referencia a la cola de la lista. Es un método privado y sólo es utilizado para instanciar el iterador.

3. **Clase Iterator:** La finalidad de esta clase es recorrer los elementos de tipo DList mediante un puntero. Será utilizada por ambas soluciones. Sus atributos privados son 3 punteros de tipo Nodo llamados “movil”, “head” y “tail”.
   a) **Iterator(Nodo*, Nodo*):** Método constructor de la Clase. Se inicia la posición inicial del iterador en la primera posición de la lista. y guarda la referencia al principio y al final de la lista
   b) **~Iterator():** Método destructor de la Clase.
   c) **Nodo* next():** Retorna el Nodo en el que se encuentra el iterador.
   d) **bool hasNext():** Retorna verdadero en caso de que se pueda seguir avanzando en la lista y falso si es que se ha llegado al final de la lista.

4. **Clase NodoHeap:** Esta clase será utilizada por la clase Heap. Sus atributos privados son “par” de tipo un par<int, int>, este almacenará un par de la lista. y un entero “frecuencia” que corresponderá a la frecuencia del par.
   a) **NodoHeap(int, int):** Método constructor de la clase. Le asignará los valores al par y su frecuencia comenzará en 1.
   b) **~NodoHeap():** Método destructor de la Clase
   c) **int getFrecuencia():** Retorna el valor asociado a la frecuencia del par.
   d) **void setFrecuencia(int):** Permite asignar la frecuencia del par.
   e) **pair<int,int> getPar():** Retorna el par correspondiente del Nodo.

5. **Struct Data:** Su función es entregar más información de las ocurrencias de cada par. Almacenará un puntero de la primera y última ocurrencia de tipo Nodo, una referencia de tipo NodoHeap y esta corresponderá a la información del par almacenado en el Heap y un entero que entregará la posición específica de este en el Heap.

6. **Clase Heap:** Esta clase implementa un MAXHeap mediante un arreglo de tipo NodoHeap y están ordenados según su frecuencia. Permite a Repairv2 acceder de manera más directa al par más frecuente de la lista, cabe destacar que esto es gracias a la referencia de tipo map que recibe.
   a) **Heap(map<pair<int, int> Data*>*):** Es el constructor de la clase, recibe una referencia al map que contiene Repairv2 para así poder modificar directamente.
   b) **~Heap():** Destructor de la clase.
   c) **void push(NodoHeap*):** Permite insertar un par en el heap, con frecuencia 1. Debido a la implementación de Repairv2 push no hace UpHeap, ya que no se insertan elementos de frecuencia mayor a 1.
   d) **void pop():** Permite eliminar el elemento con mayor frecuencia del Heap, posteriormente llama a i), que corresponde a un down heap, para equilibrar.
   e) **NodoHeap* top():** Retorna una referencia al elemento con mayor frecuencia del Heap.
   f) **bool empty():** Permite consultar si hay elementos en el heap
   g) **int size():** Retorna el número de elementos del heap
   h) **void verify(int):** Permite hacer up heap en el Heap. No es llamado directamente por la clase, sino que es llamado por Repairv2. Cuando un elemento ya fue

añadido al Heap, no se vuelve a insertar, simplemente se accede al NodoHeap que contiene el par, se le incrementa la frecuencia y se le hace up heap. Luego accede a la referencia al map para actualizar la nueva posición en el arreglo de los NodoHeap que fueron afectados por este método.

i) **void verifyDown(int):** Corresponde al down heap, por lo tanto al igual que el método h) influye en el ordenamiento del heap. Sin embargo es llamado por la clase Repairv2 cuando se disminuye la frecuencia.

j) **void prints():** Imprime el heap, recorriendo el vector con el que fue implementado e imprimiendo tanto el par como su frecuencia.

7. **Clase RepairADT:** Es una clase abstracta que sirve para la implementación de Repairv1 y Repairv2. Contiene las librerías que necesita cada una y los métodos básicos.
   a) **virtual void cambiar():** Método principal de las clases implementadas, se encarga se realizar la solución en cada clase.
   b) **virtual void prints():** En cada clase imprime la Lista.

8. **Clase Repairv1:** Sus atributos privados son: un puntero "seq" de tipo DList, un entero "mayor" y un map "mp".
   a) **Repairv1(DList*):** Método constructor de la clase, recibe como parámetro una lista enlazada y asigna el valor al entero "mayor".
   b) **~Repairv1():** Método destructor de la clase.
   c) **pair<int, int> ObtenerPrimerMayorFrec()(*):** Este método recorre la lista asignando frecuencias a los pares utilizando un map. Almacena el par con mayor frecuencia en "num" y lo retorna.
   d) **void Cambiar():** Es el método principal de la solución, es llamado recursivamente. Lo primero que hace es consultar al método c) cuál es el par con mayor frecuencia y lo almacena en "dato", si este retorna un par con frecuencia igual o menor a 1, Cambiar() termina. En caso contrario recorre, utilizando un iterador, la lista doblemente enlazada y busca los nodos que coincidan con los números en "dato" y modifica la lista. Los elimina y añade el nuevo número dado por la variable "mayor". Luego elimina todos elementos del mapa "mp" y se llama a sí misma.
   e) **void prints():** Imprime la lista almacenada en la clase.

9. **Clase Repairv2:** Corresponde a la solución avanzada. Sus atributos privados son: un puntero "seq" de tipo DList, un entero "mayor", un puntero "hp" de tipo Heap y un map "mp". La clave del map es el par de enteros y su valor es una referencia de tipo Data.
   a) **Repairv2(DList*):** Es el método constructor de la clase, recibe como parámetro la referencia a la lista que se analizará. Se encarga de inicializar el Heap enviando como referencia el map que contiene la clase. Luego llama al método CrearHeap().
   b) **~Repairv2():** Método destructor de la clase, se encarga de llamar al destructor del Heap.
   c) **void crearHeap()(*):** Este método es llamado una única vez por el constructor.

Se encarga de recorrer la lista e introducir cada par en el Heap. Utilizando un map puede reconocer si es que un par fue añadido o no. En caso de ya haber estado en el Heap, se encarga de modificar la ocurrencia siguiente del último par que había sido añadido y de actualizar la última ocurrencia del map asociado al par.

d) void buscaOcurrencias(Nodo*)(*): Este método es recursivo y se encarga de encontrar las ocurrencias del par que se le entrega. Lo primero que hace es verificar lo que se encuentra a los costados del par en la lista doblemente enlazada y con el método e) reduce la frecuencia de los que se encuentran al costado, es decir con los que el par comparte nodo. Luego con el método e) reduce su propia frecuencia. Finalmente llama a g) para actualizar el estado de la lista. Si la primera o la última ocurrencia del par es nula, el método no vuelve a llamarse a sí mismo.

e) void actualizaHeap(Nodo*, Nodo*): Se encarga principalmente de reducir la frecuencia de los nodos entregados y actualiza el ordena del Heap.

f) void actualizarLista(Nodo*, Nodo*)(*): Método que modifica el estado de la lista doblemente enlazada. Este analiza 3 casos, si el nodo a modificar está en la cabeza, en la cola o en el medio de la lista. Si entra en uno de los casos, se analizan y se actualizan los punteros de las ocurrencias previas, siguientes, la primera y última correspondiente, se elimina el nodo que ingresa y se inserta el valor que está almacenado en "mayor" a la lista, además llama a h) que sirve para insertar los/el nuevo nodo/s que se formen en el Heap.

g) void cambiar(): Este método contiene un ciclo while, se termina cuando el top del Heap posee frecuencia menor o igual a 1. El ciclo llama a d) y le entrega de parámetro la primera ocurrencia, almacenada gracias a la struct Data, correspondiente al par que está en el top del Heap. Luego aumenta el valor de "mayor".

h) void insertarHeap(Nodo*,Nodo*)(*): Método que añade al heap los pares nuevos que se forman, este es similar a c), sin embargo difiere cuando se debe reasignar la primera y la última ocurrencia de un par ya existente. Ya que pueden ocurrir casos donde las ocurrencias previas o siguientes son eliminadas y esto puede provocar cambios con las primera y última ocurrencia. Para evitar esto reajusta los punteros de sí mismo .

i) void prints(): Imprime la lista doblemente enlazada.

Ejemplo Repairv2:

A continuación se describirá un ejemplo que demuestra detalladamente el funcionamiento de la solución avanzada. La secuencia a reducir será **{2 3 1 2 3 1 2 3}** y será entregada como parámetro del constructor de un objeto de tipo Repairv2. En el momento en que es llamado constructor de la clase, se llama al método *crearHeap()*, que se encargará de analizar cada par de la lista e introducirlos en la estructura de datos *Heap*.

La lectura comienza con el primer par (2,3) **{2 3 1 2 3 1 2 3}**. Como este par no se encuentra en la estructura map, se sabe que no está presente en el *Heap*, por lo que se introduce con frecuencia 1 y el valor de su llave en el map se inicia guardando como primera y última ocurrencia éste mismo par. Luego se sigue con el siguiente par (3, 1) **{2 3 1 2 3 1 2 3}**, de igual manera, el par no se encuentra en el *Heap*, y se introduce con frecuencia 1. Lo mismo con el par siguiente (1,2) **{2 3 1 2 3 1 2 3}**, que ocurre lo mismo que en los casos anteriores. Seguido de esto viene el par (2,3) **{2 3 1 2 3 1 2 3}**, el cual al buscar la clave en el map se reconoce su existencia, demostrando que que ya había aparecido anteriormente en la lista. Por lo tanto, no es necesario añadirlo al *Heap*, en lugar de eso se accede al *NodoHeap*, que fue almacenado en el valor del map está asociado a ese par, para incrementar su frecuencia y luego se realiza un *UpHeap*. Causando que el nuevo top del *Heap* corresponda al par (2,3) con frecuencia 2. Además en el map del par se actualizan los punteros correspondientes. Con más detalle ocurre de la siguiente manera: primero se accede a la última ocurrencia que estaba almacenada correspondiente al par **{2 3 1 2 3 1 2 3}** y se le asigna su ocurrencia siguiente el par actual **{2 3 1 2 3 1 2 3}**. y viceversa, el par previamente almacenado pasa a ser la ocurrencia previa del par actual. Y luego se asigna como última ocurrencia del map el par actual **{2 3 1 2 3 1 2 3}**.
El *Heap* hasta este momento está así:

```
(2,3)->2
(3,1)->1
(1,2)->1
```

Luego para el par (3,1) **{2 3 1 2 3 1 2 3}** y (1,2) **{2 3 1 2 3 1 2 3}** ocurre lo mismo descrito anteriormente, ya que sus claves son reconocidas en el map. Luego de ese proceso, el *Heap* queda:

```
(2,3)->2
(3,1)->2
(1,2)->2
```

Ahora con el último par (2,3) **{2 3 1 2 3 1 2 3}**, como es un par que ya está en el *Heap*, se accede nuevamente al *NodoHeap* del par para incrementar su frecuencia y hacer *UpHeap*. Luego se actualizan las referencias a las ocurrencias, en concreto el par **{2 3 1 2 3 1 2 3}** pasa a ser la ocurrencia previa de **{2 3 1 2 3 1 2 3}** y viceversa. Finalmente se actualiza la última ocurrencia del map que ahora corresponde al par **{2 3 1 2 3 1 2 3}.**
El método *crearHeap()* finaliza, ya que el iterador se encuentra al final de la lista. El map y el *Heap* resultante quedan de la siguiente manera:

Map:

```
mp[{2,3}]->PrimeraOcurrencia = {2 3 1 2 3 1 2 3}
         ->UltimaOcurrencia  = {2 3 1 2 3 1 2 3}

mp[{3,1}]->PrimeraOcurrencia = {2 3 1 2 3 1 2 3}
         ->UltimaOcurrencia  = {2 3 1 2 3 1 2 3}

mp[{1,2}]->PrimeraOcurrencia = {2 3 1 2 3 1 2 3}
         ->UltimaOcurrencia  = {2 3 1 2 3 1 2 3}
```

Heap:

```
(2,3) -> 3
(3,1) -> 2
(1,2) -> 2
```

Al llamar al método *cambiar()* este se ejecutará mientras el elemento top del *Heap* tenga una frecuencia mayor que 1. En este caso recibirá el par (2,3), ya que posee frecuencia 3. *cambiar()*, llama al método *buscaOcurrencias()* con la primera ocurrencia del par, obtenida por el map, es decir el par **{2 3 1 2 3 1 2 3}**.
Una vez es llamado *buscaOcurrencias()* éste se encargará primero de los pares que saldrán afectados tras reemplazar el par, Para este caso como se trata del primer par de la lista, el único para afectado tras el reemplazo es el par **{2 3 1 2 3 1 2 3}**. Por lo que se llama a *actualizarHeap()* con el par para que se disminuya su frecuencia en su *NodoHeap,* y posterior se haga *DownHeap* a este mismo.
Luego, *buscaOcurrencias()* llama al método *actualizaHeap()* con el par **{2 3 1 2 3 1 2 3}**, para que se le disminuya su frecuencia y haga *DownHeap* de él mismo. Como resultado el *Heap* queda de la siguiente manera:

```
(2,3) -> 2
(1,2) -> 2
(3,1) -> 1
```

Finalmente se llama al método *actualizarLista()* con el par **{2 3 1 2 3 1 2 3}**, el cual se encargará de realizar el reemplazo mediante *insertarHeap()* y de asignar los punteros a las ocurrencias correspondientes.
El método *actualizarLista()* primero va a tomar el par afectado **{2 3 1 2 3 1 2 3}** y hará que la que era su ocurrencia siguiente **{2 3 1 2 3 1 2 3}** sea ahora la primera ocurrencia del map[{3,1}], y luego al par **{2 3 1 2 3 1 2 3}** le asignará su ocurrencia previa en *NULL*.
Luego de eso, el método accedera a la ocurrencia siguiente del par **{2 3 1 2 3 1 2 3}**, o sea el par **{2 3 1 2 3 1 2 3}**, y este par pasará a ser la primera ocurrencia del map[{2,3}]. Y al par **{2 3 1 2 3 1 2 3}** ahora se le asignará su ocurrencia previa en *NULL*.
Después de esto sí se hará el reemplazo del par **{2 3 1 2 3 1 2 3}**. Y quedará así **{28 1 2 3 1 2 3}** . Finalmente se procede a añadir, llamando a insertarHeap(), el nuevo par(28, 1) al *Heap*, Aquí termina la llamada al método *actualizarLista().*

El método *buscaOcurrencia()* se vuelve a llamar a sí mismo. esta vez usando la nueva primera ocurrencia del par (2,3), **{28 1 2 3 1 2 3}** y así hasta que la primera ocurrencia del par sea *NULL*.

En ésta llamada con el par **{28 1 2 3 1 2 3}** , se llama *actualizaHeap()* con los pares **{28 1 2 3 1 2 3}** , **{28 1 2 3 1 2 3}** y **{28 1 2 3 1 2 3}**, el resultado del *Heap* es el siguiente:

```
(2,3) -> 1
(1,2) -> 1
(28,1) -> 1
(3,1) -> 0
```

Ahora es llamado *actualizarLista()* con el par **{28 1 2 3 1 2 3}**, aquí al par (1,2) se modificará su primera ocurrencia en el map, la cual será ahora el par **{28 1 2 3 1 2 3}**, y ahora a este par se le asignará su ocurrencia previa en *NULL*. De igual modo con el par (3,1) el cual la primera ocurrencia del map se asignará en *NULL,* ya que el par **{28 1 2 3 1 2 3}** no tiene ocurrencia siguiente.
Finalmente se hace el reemplazo del par **{28 1 2 3 1 2 3}**, dejando la lista de la siguiente manera: **{28 1 28 1 2 3}.** Para terminar el método *actualizarLista()* se encarga de asignar como primera ocurrencia del par (2,3) a **{28 1 28 1 2 3}** y además de añadir mediante *insertarHeap()* los nuevos pares creados al *Heap*, estos son los pares (1,28) y (28,1) como el par (28,1) ya está presente en el *Heap* ocurre lo anteriormente descrito y se hace *UpHeap* sobre el, quedando como resultado el siguiente *Heap:*

```
(28,1) -> 2
(2,3) -> 1
(1,2) -> 1
(1,28) -> 1
(3,1) -> 0
```

Nuevamente se vuelve a llamar al método *buscaOcurrencias()*, esta vez con el par**{28 1 28 1 2 3}** que es ahora la primera ocurrencia del par(2,3).
Al llamar al método *actualizaHeap()* con el par **{28 1 28 1 2 3}** y **{28 1 28 1 2 3}**, el *Heap* queda así:

```
(28,1) -> 2
(1,28) -> 1
(2,3) -> 0
(1,2) -> 0
(3,1) -> 0
```

Ahora *actualizarLista()* con el par **{28 1 28 1 2 3}** como se trata del par a la cola de la lista solo tiene que modificar el par (1,2) y asignar una nueva ultima ocurrencia. Sin embargo, como el par **{28 1 28 1 2 3}** no tiene ocurrencia previa, al map[{1,2]} se le asigna su primera y última ocurrencia como *NULL*. Finalmente se reemplaza el par **{28 1 28 1 2 3}** resultando **{28 1 28 1 28}** , al par(2,3) se le asigna como primera ocurrencia un *NULL* porque el par **{28 1 28 1 2 3}** no tenía ocurrencia siguiente. Y finalmente se añade el par(1,28) al *Heap* y terminando la llamada al método.
Volviendo a *buscaOcurrencia()* el cual termina definitivamente su llamada ya que ahora la primera ocurrencia del par(2,3) en *NULL*.

El Heap resultante es:

```
(28,1) -> 2
(1,28) -> 2
(2,3) -> 0
(1,2) -> 0
(3,1) -> 0
```

y el Map:

```
mp[{28,1}]->PrimeraOcurrencia = {28 1 28 1 28}
          ->UltimaOcurrencia  = {28 1 28 1 28}
mp[{1,28}]->PrimeraOcurrencia = {28 1 28 1 28}
          ->UltimaOcurrencia  = {28 1 28 1 28}
```

Ahora el programa se encuentra nuevamente en el método *cambiar()*, el cual llama al top del *Heap*, el par retornado es(28,1). Como éste tiene frecuencia 2, se llama a *buscaOcurrencias()* con la primera ocurrencia de este par: **{28 1 28 1 28}**.
El método *buscaOcurrencias()* llama a *actualizarHeap()* con los pares **{28 1 28 1 28} y {28 1 28 1 28}**, el resultado del *Heap* tras esto es:

```
(28,1) -> 1
(1,28) -> 1
(2,3) -> 0
(1,2) -> 0
(3,1) -> 0
```

Ahora se llama a *actualizarLista()* con el par **{28 1 28 1 28}**, el cual actualiza las primeras ocurrencias de los pares(1,28) y (28,1). Reemplaza el par quedando tal que así: **{29 28 1 28}**, y añade el nuevo par (29,28) al *Heap*.

```
(28,1) -> 1
(1,28) -> 1
(29,28) -> 1
(2,3) -> 0
(1,2) -> 0
(3,1) -> 0
```

Luego es llamado *buscaOcurrencias()* con la nueva primera ocurrencia del par (28,1), **{29 28 1 28}**, se llama a *actualizaHeap()* con el par **{29 28 1 28}**, **{29 28 1 28}**, y **{29 28 1 28}**, quedando el *Heap así:*

```
(28,1) -> 0
(1,28) -> 0
(29,28) -> 0
(2,3) -> 0
(1,2) -> 0
(3,1) -> 0
```

El método *actualizaLista()* actualiza la lista de la siguiente manera **{29 29 28}**, y añade el par(29, 29) y modifica el par (29,28) y finaliza el método.

Finalmente *buscaOcurrencias()* finaliza ya que ya no existen más ocurrencias del par(28,1). El *Heap* resultante es:

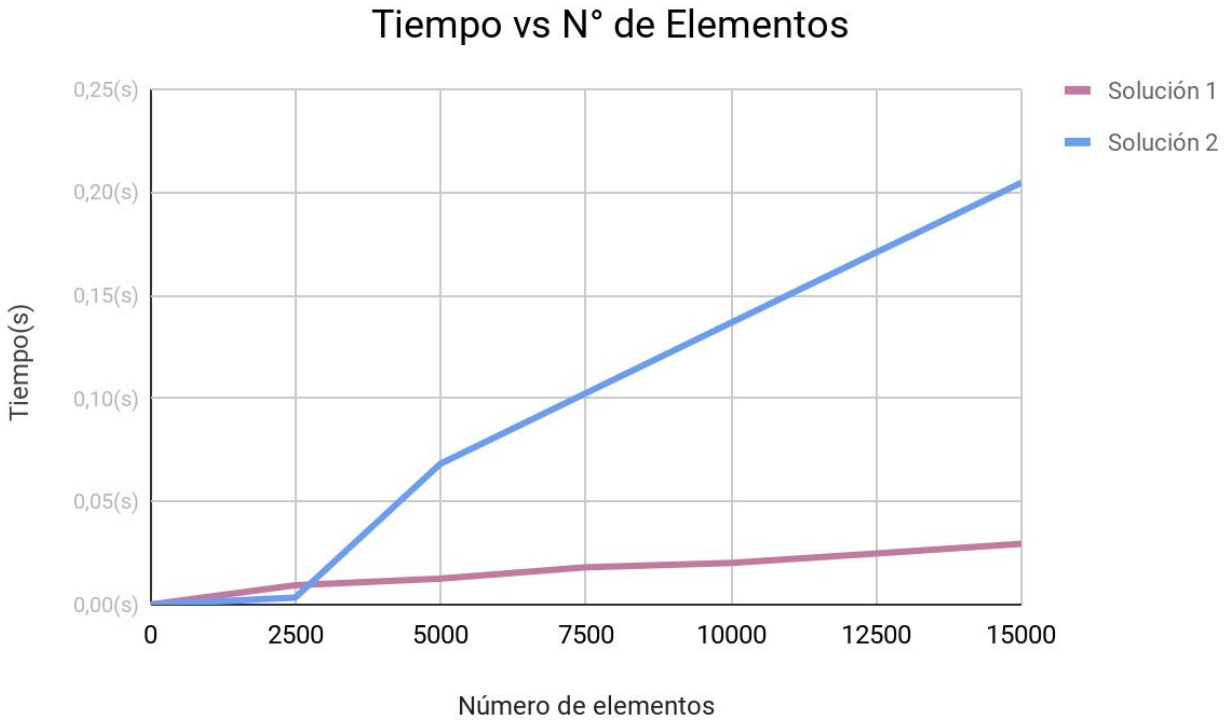
```
(29,29) -> 1
(29,28) -> 1
(28,1) -> 1
(1,28) -> 0
(2,3) -> 0
(1,2) -> 0
(3,1) -> 0
```

Devuelta en el método *cambiar()* ahora el top del *Heap* tiene frecuencia 1, por lo que también finaliza dejando como resultado la lista **{29 29 28}**

Evaluación del tiempo de ejecución en cada solución:

CASO 1: Números iguales

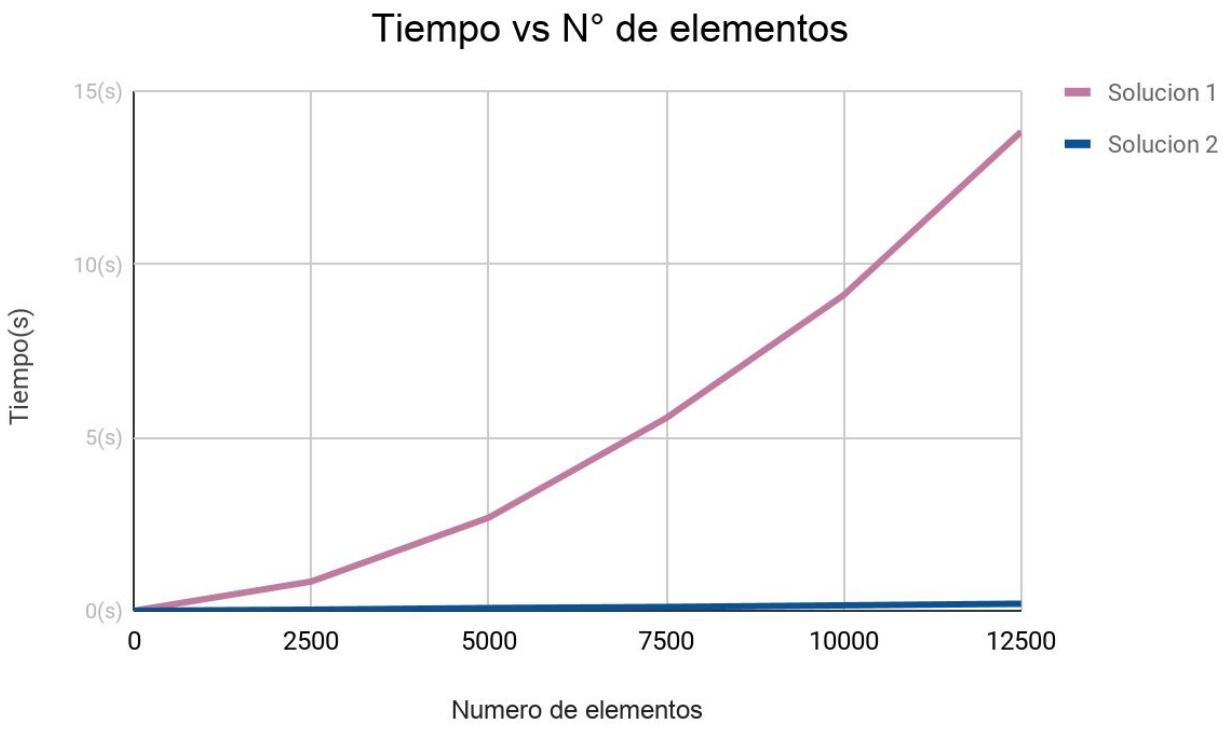
Número de elementos	Repairv1/Solución 1 Tiempo (s)	Repairv2/Solución 2 Tiempo (s)
0	0	0
2500	0,0094	0,0034
5000	0,01261	0,0683
7500	0,0181	0,1025
10000	0,0202	0,1367
12500	0,0248	0,1708
15000	0,0295	0,2046



En este caso se puede observar que los tiempos de las soluciones difieren levemente. La solución 1 entrega menor tiempo que la 2 y esto puede ser porque al tener que cambiar todos los datos linealmente, el cambio es más directo. Por otro lado en la solución 2 puede influir la actualización de punteros, ya que en este caso realizará un cambio lineal en la lista doblemente enlazada, pero tendrá un trabajo extra al tener que actualizar el orden del heap y los punteros correspondientes.

CASO 2: Números aleatorios

Número de elementos	Repairv1/Solución 1 Tiempo (s)	Repairv2/Solución 2 Tiempo (s)
0	0	0
2500	0,8438	0,0312
5000	2,6875	0,0781
7500	5,5625	0,1094
10000	9,1095	0,1562
12500	13,8281	0,2031
15000	20,7188	0,2812



En este caso se puede observar que la solución 1 entrega tiempos muy elevados y esto se debe a que realiza, en todo los casos, múltiples lecturas sobre la lista. En cambio la solución avanzada solo realiza una lectura y si en esta lectura no encuentra ocurrencias coincidentes, no accederá a esa posición. Al ser números azarosos del 1 al 27, la frecuencia de cada par puede tender a ser menor.

Conclusión

El proyecto nos permitió entender el funcionamiento de diversas estructuras de datos y las posibles aplicaciones de estas. Al realizar un análisis a los tiempos de ejecución de cada solución se pudo concluir que la eficiencia depende de los casos seleccionados, ya que en caso promedio era mucho mejor la solución avanzada que la básica. Sin embargo, no fue posible asegurar una solución óptima para todos los casos. Ya que se pudo observar que en peor caso era más directa la solución básica. Cabe destacar que con grandes volúmenes de datos, los tiempos de ejecución eran muy elevados, esto influyó en los resultados entregados y pueden diferir con los teóricos.