



Índice

1. Objetivos	2
2. Introducción	3
3. Marco Teórico	4
3.1. Polifase	4
3.2. Mezcla Equilibrada	6
3.3. Radix Externo	7
4. Metodología de desarrollo	9
4.1. Entorno inicial de trabajo	9
4.2. Lectura y escritura de archivos en Java	9
4.3. Generación de archivos	11
4.4. Lectura inicial de las claves	11
5. Análisis de las implementaciones	13
5.1. Simulación de Polifase	13
5.1.1. Polifase numérica	13
5.1.2. Polifase no numérica	20
5.2. Simulación de Radix Externo	22
5.3. Simulación de la Mezcla Natural	28
6. Mezcla Natural no numérica	34
7. Pruebas de funcionamiento y rendimiento	34
8. Conclusiones	36



1. Objetivos

- Que el alumno implemente los algoritmos de ordenamiento externos y que conozca los elementos para el manejo de archivos.
- Que el alumno aplique los conceptos generales de programación.
- Que el alumno desarrolle sus habilidades de trabajo en equipo.



2. Introducción

De manera constante en la solución de problemas en programación es necesario manejar grandes cantidades de datos de forma eficiente, Dichas cantidades de información crecen con el paso del tiempo, como claro ejemplo tenemos el Big Data. Las técnicas necesarias para lograrlo se han ido desarrollando conforme las necesidades surgen en la solución de problemas, una de las técnicas para lidiar con la gran cantidad de información, son los algoritmos de ordenamiento.

Se sabe que el ordenamiento permite una mejor gestión de la información, pero, ¿qué es el ordenamiento? Es una operación que consiste en reacomodar colecciones de datos de tal manera que sigan un orden lógico o una secuencia, siguiendo algún criterio, ascendente o descendente, por ejemplo.

Para efectos del presente proyecto, se tratarán tres algoritmos de ordenamiento externo: Polifase, Mezcla equilibrada y Radix externo. Se tratarán por separado con la intención de dejar lo más claro posible cada uno de los algoritmos y de esta manera permitir que los estudiantes adquieran de manera significativa conocimientos sobre estos algoritmos.

3. Marco Teórico

A largo del desarrollo de las bases de datos, solución de problemas en programación y otras cuestiones que tengan que ver con el uso de grandes cantidades de datos, se han implementado algoritmos de ordenamiento.

Los algoritmos de ordenamiento más conocidos son: Insertion sort, Selection Sort, Heap Sort, Bubble Sort, Quick Sort, Counting Sort, Radix Sort y Merge Sort. Los algoritmos de ordenamiento tienen 4 operaciones fundamentales:

- Comparación
- Intercambio
- Intercalación
- Inserción

Ahora, si bien todos los algoritmos son capaces, en teoría, de ordenar una colección de N elementos, en la práctica se tiene un inconveniente que se presenta cuando la colección es muy grande y la memoria interna (RAM) no es capaz de soportar o manejar todos los datos al mismo tiempo, ya que la memoria no es infinita. Antes se trabajaba solo de manera interna y los datos se tenían que manejar en una sola ejecución, por lo que cuando la memoria interna es incapaz de mantener todos los datos al mismo tiempo, el algoritmo inevitablemente fallaría en estos casos.

A medida que pasa el tiempo la cantidad de datos que deben ser ordenados crece con rapidez mayor a la que se desarrollan computadoras con mayor capacidad de almacenamiento, es en este punto donde surge la necesidad de emplear algoritmos de ordenamiento externo. Los cuales se auxilian de componentes externos para almacenar los elementos a ordenar y de esta manera optimizar la manera en la que se manejan los datos a ordenar.

Los algoritmos de ordenamiento externo se basan en la estrategia divide y vencerás, la cual permite la construcción de subproblemas menos complejos a partir del problema general con el objetivo de facilitar el manejo de la información de el problema y de esta forma hacer más eficiente la solución de el problema general.

Los algoritmos de ordenamiento que se tratarán en el presente documento son todos externos, es decir, es necesario emplear memoria adicional y archivos, los cuales funcionarán como apoyo para aplicar la estrategia con la que están diseñados estos algoritmos.

3.1. Polifase

Este algoritmo utiliza tres archivos adicionales al archivo que contiene la colección de datos a ordenar.

La manera en que realiza las particiones se basa en bloques de un tamaño fijo, la longitud de los bloques queda predefinida según lo que se requiera en el programa. Los bloques de longitud M serán leídos de un archivo F_0 , posteriormente cada bloque se reescribirá de forma intercalada en los archivos auxiliares: el primer bloque irá a F_1 y el segundo a F_2 , y así sucesivamente.

Una vez que no haya más bloques que reescribir, empezarán a intercalarse entre ellos, para dar lugar a bloques del doble de tamaño, dichos bloques se escribirán en los archivos F_0 y F_3 de manera intercalada, de forma similar a como se escribieron en F_1 y F_2 .

Las acciones antes mencionadas se repetirán hasta que ya no sea posible general bloques de mayor tamaño. Cuando esto suceda, la colección de datos quedará ordenada.

Ejemplo:

Dado el archivo $F_0 = 12, 34, 87, 56, 19, 43, 65, 78, 12, 34, 23, 45, 67, 89, 76, 12, 56, 43, 78, 89$ y $M=3$, se realizarán las siguientes acciones:

$$F_1 = [12, 34, 87], [12, 65, 78], [67, 76, 89], [78, 89]$$

$$F_2 = [19, 43, 56], [23, 34, 45], [12, 43, 56]$$

Como se puede ver el F_1 , no siempre se logran hacer bloque del tamaño especificado, el último bloque con frecuencia es de menor tamaño.

Una vez obtenidos esos archivos, empiezan a mezclarse los bloques de manera que se vayan ordenando al momento de escribirse en el archivo correspondiente, se escribe en F_0 y F_3 de la siguiente forma.

$$F_3 = [12, 23, 34, 45, 65, 78], [78, 89]$$

En caso de no haber otro bloque con el cual combinar el último bloque esté pasa tal cual al archivo que corresponda.

Ser repite el proceso las veces necesarias, hasta que ya no sea posible formar bloques de mayor tamaño.

$$F_1 = [12, 12, 19, 23, 34, 34, 43, 45, 56, 65, 78, 87]$$

$$F_2 = [12, 43, 56, 67, 76, 78, 89, 89]$$

$$F_0 = [12, 12, 12, 19, 23, 34, 34, 43, 43, 45, 56, 56, 65, 67, 76, 78, 78, 87, 89, 89]$$

La colección ha sido ordenada.

3.2. Mezcla Equilibrada

Este algoritmo también recibe el nombre de Mezcla Natural, trabaja de manera similar a Polifase, las diferencias entre estos dos algoritmos son:

- Polifase trabaja con bloque de tamaño fijo y predefinido, mientras Mezcla Equilibrada lo hace con bloques de tamaño variable. Esto debido a que al trabajar con mezcla equilibrada los bloques que se tomen serán de tamaño máximo, tomando como criterio que los elementos de cada bloque estarán ordenados según el criterio deseado para toda la colección de datos. De ahí surge otra diferencia con Polifase, no es necesario emplear un algoritmo de ordenamiento interno, pues los bloques para Mezcla Equilibrada ya están ordenados desde el momento en el que fueron formados.
- Polifase requiere tres archivos adicionales, por su parte Mezcla Equilibrada requiere únicamente dos.

Ejemplo:

Si utilizamos la misma colección del ejemplo de Polifase:

$$F_0 = 12, 34, 87, 56, 19, 43, 65, 78, 12, 34, 23, 45, 67, 89, 76, 12, 56, 43, 78, 89$$

obtenemos:

$$F_1 [12, 34, 87], [19, 43, 65, 78], [23, 45, 67, 89], [12, 56]$$

$$F_2 = [56], [12, 34], [76], [43, 78, 89]$$

Una vez repartidos los bloques entre esos dos archivos, se hace la combinación de los elementos en el archivo F0 de la siguiente forma.

$$F_0 = [12, 34, 56, 87], [12, 19, 34, 43, 65, 78], [23, 45, 67, 78, 89, 89], [12, 43, 56, 78, 89]$$

Aquí termina la primera iteración y a partir de estos nuevos bloques se generarán otros bloques de mayor tamaño.

$$F_1 = [12, 34, 56, 87], [23, 45, 67, 78, 89, 89]$$

$$F_2 = [12, 19, 34, 43, 65, 78], [12, 43, 56, 78, 89]$$

$$F_0 = [12, 12, 19, 34, 34, 43, 56, 78, 87], [12, 23, 43, 45, 56, 67, 78, 78, 89, 89, 89]$$

Fin de la segunda iteración.

$$F_1 = [12, 12, 19, 34, 34, 43, 56, 78, 87]$$

$$F_2 = [12, 23, 43, 45, 56, 67, 78, 78, 89, 89, 89]$$

$$F_0 = [12, 12, 12, 19, 23, 34, 34, 43, 43, 45, 56, 56, 67, 78, 78, 78, 87, 89, 89, 89]$$

Fin de la tercera iteración y fin del algoritmo, ya que no es posible formar bloques de mayor tamaño y la colección ya está ordenada.

3.3. Radix Externo

En el caso de este algoritmo la forma en que se realizan las particiones cambia con respecto a cómo se han hecho en Polifase y Mezcla Equilibrada.

Esta vez las particiones se hacen en base a los dígitos significativos de cada uno de los elementos de que deben ser ordenados.

Por lo general este algoritmo requiere de más archivos auxiliares en comparación con los algoritmos antes mencionados, la cantidad de estos dependen de los posibles dígitos distintos que tengan las claves a ordenar. Es necesario un archivo por cada uno de los posibles dígitos que contengan las claves.

Es necesario aclarar que cuando se habla de dígitos se contemplan los números, las letras y otros caracteres que puedan componer a las claves.

Ejemplo:

Si se tiene el archivo $F_* = 234, 241, 421, 342, 32, 443, 112, 34, 432, 13$

Los archivos necesarios serán: $F_0 = , F_1 = , F_2 = , F_3 = , F_4 =$

Un archivo por cada dígito diferente: 0, 1, 2, 3 y 4.

Una vez teniendo esto claro, se inicia tomando el dígito menos significativo, que en este caso como son números, se toman las unidades:

$$F_0 =$$

$$F_1 = 241, 421$$

$$F_2 = 342, 32, 112, 432 \quad F_3 = 443, 13$$

$$F_4 = 234, 34$$

En este caso, como se ordenarán de forma ascendente, primero se “vacía” el archivo correspondiente al dígito con menor valor y el último que se vacía será el del dígito con mayor valor.

Se sobrescribe en el archivo que tenía las claves en el orden original de manera que se respete la regla con la que se desea ordenar la colección completa en este caso de manera ascendente.

Es importante resaltar que los archivos que se usarán para dividir las claves se comportan de manera similar a una estructura FIFO (First In First Out).

$$F_* = 241, 421, 32, 112, 342, 432, 13, 443, 34, 234$$

Se sigue con la división de las claves en los archivos, pero ahora se toma en cuenta el siguiente dígito más significativo que el anterior: las decenas. De la siguiente manera.

$$F_0 =$$

$$F_1 = 112, 13$$

$$F_2 = 421$$

$$F_3 = 32, 432, 34, 234$$

$$F_4 = 241, 342, 443$$

$$F_* = 112, 13, 421, 32, 432, 234, 34, 241, 342, 443$$

Se sigue con la división de las claves en los archivos, pero ahora se toma en cuenta el siguiente dígito más significativo que el anterior: las centenas. De la siguiente manera.

$$F_0 = 13, 32, 34$$

$$F_1 = 112$$

$$F_2 = 234, 241$$

$$F_3 = 342$$

$$F_4 = 421, 432, 443$$

$$F_* = 13, 32, 34, 112, 234, 241, 342, 421, 432, 443$$

Dado que ya no hay más dígitos significativos que clasificar, el algoritmo llega a su fin y da como resultado la colección ya ordenada.

4. Metodología de desarrollo

4.1. Entorno inicial de trabajo

La programación de una solución al problema del ordenamiento de claves con 3 campos requiere abstraer y dividir el problema inicial en 3 sub-problemas:

- Almacenar cada una de las claves en estructuras de datos auxiliares.
- Leer archivos y escribir sobre ellos.
- Generar nuevos archivos o eliminarlos.

4.2. Lectura y escritura de archivos en Java

Java posee diversas clases que permiten la apertura, así como la lectura y escritura en archivos existentes:

- La clase *File* permite crear un objeto que tiene asociado al archivo que se desea manipular; estableciendo la ruta de acceso al archivo en el constructor del objeto.

```
File(String pathname)  
Creates a new File instance by converting the given pathname string into an abstract pathname.
```

Figura 1: Constructor básico de *File*.

- La clase *FileReader* permite inicializar un proceso de lectura sobre un objeto de tipo *File*; introduciendo en el constructor un objeto inicializado de tipo *File*, respectivamente. Su uso está pensado específicamente para la lectura de archivos con caracteres.

```
FileReader(File file)  
Creates a new FileReader, given the File to read from.
```

Figura 2: Constructor básico de *FileReader*.

- Las herramientas proporcionadas por la clase *BufferedReader* permiten una lectura eficiente de caracteres, arreglos y líneas generando un *buffer* que almacena los datos temporalmente y agiliza el proceso de lectura de archivos externos. Al crear un objeto de esta clase, su constructor debe incluir como parámetro un objeto de tipo *FileReader*.

```
BufferedReader(Reader in)  
Creates a buffering character-input stream that uses a default-sized input buffer.
```

Figura 3: Constructor básico de *BufferedReader*.

- La clase *FileWriter* permite inicializar un proceso de escritura sobre un objeto de tipo *File*; introduciendo en el constructor un objeto inicializado de tipo *File*, respectivamente. Su uso está pensado específicamente para la escritura de caracteres.

```
FileWriter(File file)  
Constructs a FileWriter object given a File object.
```

Figura 4: Constructor básico de *FileWriter*.

- Las herramientas proporcionadas por la clase *BufferedWriter* permiten una escritura eficiente de caracteres, arreglos y líneas generando un *buffer* que almacena los datos temporalmente y agiliza el proceso de escritura de caracteres sobre archivos externos. Al crear un objeto de esta clase, su constructor debe incluir como parámetro un objeto de tipo *FileWriter*.

```
BufferedWriter(Writer out)  
Creates a buffered character-output stream that uses a default-sized output buffer.
```

Figura 5: Constructor básico de *BufferedWriter*.

El uso de cada una de las clases anteriores requiere el manejo de **Excepciones**: mensajes de error asociados a entradas incorrectas, defectos en el código de un programa o errores de diseño. Cuando un error de esta naturaleza se presente, **Java** terminará inmediatamente la ejecución del programa e imprimirá en pantalla un mensaje de error.

Para manejar internamente las excepciones, se requiere el uso de las líneas **try - catch**:

- La declaración **try** permite definir un bloque de código que será puesto a prueba, verificando que no existan errores mientras se ejecuta.
- La declaración **catch** permite definir un bloque de código que se ejecutará únicamente en caso de que se detecte un error en el bloque de código definido por **try**.

```
try {  
    // Bloque de código a intentar ejecutar.  
} catch (Exception e) {  
    // Bloque de código para manejar errores.  
}
```

Figura 6: Estructura *try-catch*.

4.3. Generación de archivos

Crear nuevos archivos a través de la clase *File* es una tarea sencilla. Al momento de inicializar un objeto de tipo *File* con una ruta de acceso especificada y un nombre, la clase permite utilizar 2 métodos: **exists()** y **createNewFile()**. A través de estos 2 métodos se puede verificar si existe o no un archivo y, en caso de que no exista, crearlo en la ruta establecida en el objeto *File*.

```
File archivoMod = new File(this.rutaArchivo + nombreDelArchivo + "Sorted.txt");  
  
if (!archivoMod.exists()) {  
    archivoMod.createNewFile();  
}
```

Figura 7: Generación de un archivo no existente.

4.4. Lectura inicial de las claves

Una de las partes vitales de la implementación del ordenamiento externo reside en obtener una solución eficaz a la lectura y carga de las claves solicitadas, las cuales están compuestas cada una de 3 campos: Nombre, Apellido y Número de Cuenta. Con fin de identificar internamente las claves a manipular en el programa, las claves con sus 3 campos serán denominadas **superclaves** y cada campo se denominará como una **subclave**.

La solución al problema, por tanto, requiere del uso de un arreglo dinámico anidado que contenga cada **superclave** en cada uno de sus índices, y a su vez, cada superclave represente otro arreglo que contenga cada una de las **sub-claves** que la componen.

Uno de los métodos creados para el proyecto denominado **scanSuperKeys** brinda una solución a tal problema, generando un arreglo multidimensional anidado que guardará cada una de las superclaves y sus respectivas subclaves. El resultado final, por tanto, se asemeja a los ejemplos teóricos analizados en el curso teórico de EDA II y permite un manejo más natural e intuitivo sobre cada una de las claves.

Los pasos que sigue este método son los siguientes:

1. Inicializa un arreglo anidado de cadenas de caracteres llamado **lecturaDF0** de tipo *ArrayList*.
2. Carga por medio de la clase *File* el archivo a leer con las claves deseadas y, posteriormente, se inicializa un objeto *Scanner* introduciendo como parámetro en su constructor el objeto *File* creado.
3. Mientras el escáner detecte líneas de texto en el archivo, se definirá a cada línea con el nombre **saltoClave** (Ya que cada línea de texto en el archivo contiene 1 sola superclave) y se definirá como el **identificador** de cada superclave un espacio vacío existente entre ellas por medio de la clase *StringTokenizer*, que recibe una cadena y la separa al momento de leer un carácter definido como el **delimitador**.

4. A continuación, se recupera cada superclave y se inicializa otro arreglo dinámico de cadenas denominado **subClaves**.
5. Se inicializa otro objeto de tipo *StringTokenizer* llamado **separador** que recibirá como parámetro cada superclave y usará como delimitador de cada subclave el carácter ”,”. Posteriormente, se añadirá cada subclave a la lista utilizando el método **nextToken()** por medio de un ciclo **for**.
6. Finalmente, se añade el arreglo con las 3 subclaves en el arreglo **lecturaDF0** y se imprime el resultado en pantalla.

```
String superClave = identificador.nextToken();
System.out.println("\nEsta es una superclave: \n");
System.out.println(superClave);

System.out.println("\nEsta es la superclave separada en un arreglo con sus subclaves.");
subClaves = new ArrayList<>();

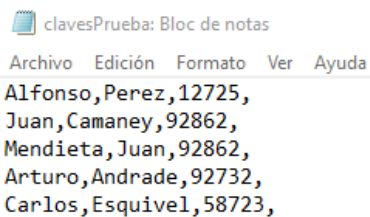
StringTokenizer separador = new StringTokenizer(superClave, ",");

for (int i = 0; i < 3; i++) {
    subClaves.add(separador.nextToken());
}

System.out.println(subClaves);
lecturaDF0.add(subClaves);
```

Figura 8: *Proceso de lectura y almacenamiento de cada superclave.*

Con fines de prueba, se ha utilizado un archivo *.txt* que contiene 5 superclaves con el formato especificado. Este archivo será leído por **scanSuperKeys** y lo almacenará en un arreglo dinámico anidado:



```
clavesPrueba: Bloc de notas
Archivo Edición Formato Ver Ayuda
Alfonso,Perez,12725,
Juan,Camane,92862,
Mendieta,Juan,92862,
Arturo,Andrade,92732,
Carlos,Esquivel,58723,
```

Figura 9: *Archivo con 5 superclaves.*

Al término de su ejecución, el método devuelve el arreglo anidado para que cualquier método de ordenamiento que lo invoque pueda hacer uso del mismo; teniendo un acceso sencillo a cada una de las subclaves de acuerdo al campo de ordenamiento que se desee utilizar (Ordenamiento por Nombre, Apellido o Número de cuenta).

```
Esta es una clave completa:
Carlos,Esquivel,58723,

Esta es la clave separada en un arreglo.
[Carlos, Esquivel, 58723]

Finalmente, el arreglo inicial F0 es el siguiente:
[[Alfonso, Perez, 12725], [Juan, Camaney, 92862], [Mendieta, Juan, 92862], [Arturo, Andrade, 92732], [Carlos, Esquivel, 58723]]
```

Figura 10: Arreglo anidado resultante.

5. Análisis de las implementaciones

5.1. Simulación de Polifase

5.1.1. Polifase numérica

El primer método de ordenamiento tiene entre sus atributos las rutas de acceso a los archivos iniciales, así como los archivos auxiliares que utilizará durante su ejecución. Además, se tienen declaradas referencias a objetos de tipo *BufferedReader*, *BufferedWriter*, *StringTokenizer* y *FileWriter* y finalmente, se añaden referencias a objetos *ArrayList* que serán utilizados para recuperar la lista de superclaves y manipularla.

```
//Rutas de acceso a las carpetas.
static String rutaArchivo = "./Archivos/";
static String rutaAuxiliar = "./Archivos/Polifase/";

//Buffers de lectura para los archivos.
BufferedReader lectorF23;
BufferedReader lectorF01;

//Buffers de escritura para los archivos.
BufferedWriter escritorF01;
BufferedWriter escritorF23;
BufferedWriter bufferClave;

//FileWriter para la inserción de caracteres en F0.
FileWriter escritorDeClaves;

//Arreglos auxiliares en la recuperación de claves.
ArrayList<ArrayList<String>> lecturaDF0;
ArrayList<Integer> arrayF0;

//Identificador para identificar y separar claves.
StringTokenizer identificador;
```

Figura 11: Atributos de la clase Polifase.

A continuación, se listarán los métodos secundarios creados para la clase *Polifase*. La documentación completa de su funcionamiento se encuentra en el código fuente del proyecto creado.

- **hubReader** y **hubWriter** son métodos dedicados a la creación de buffers de lectura y escritura para los archivos F0, F1, F2 y F3 durante la ejecución del programa. Su uso tiene como ventaja proporcionar una apertura rápida de flujos de lectura/escritura sobre los archivos auxiliares y, por tanto, tener un control simplificado de los buffers. Así mismo, permite obtener un código fuente más claro y limpio.

```
if (opDeseada == 0) {  
    //Buffer de lectura dedicado a F0 y F1.  
    this.lectorF01 = new BufferedReader(lectorArchivo);  
} else if (opDeseada == 1) {  
    //Buffer de lectura dedicado a F2 y F3.  
    this.lectorF23 = new BufferedReader(lectorArchivo);
```

Figura 12: *Buffers de lectura para los archivos de Polifase.*

```
switch (opDeseada) {  
    case 0:  
        escritorArchivo = new FileWriter(archivoTarget, false);  
        this.escriptorF23 = new BufferedWriter(escriptorArchivo);  
        break;  
    case 1:  
        escritorArchivo = new FileWriter(archivoTarget, true);  
        this.escriptorF01 = new BufferedWriter(escriptorArchivo);  
        break;  
    case 2:  
        escritorArchivo = new FileWriter(archivoTarget, true);  
        this.escriptorF23 = new BufferedWriter(escriptorArchivo);  
        break;
```

Figura 13: *Buffers de escritura para los archivos de Polifase.*

- El método **appendKeyToFile** es utilizado durante la ejecución de Polifase para añadir las subclaves recuperadas del arreglo **lecturaDF0** a un archivo "F0" inicial y comenzar el ordenamiento a partir del mismo; sin alterar el archivo original con las superclaves.

```
escriptorDeClaves = new FileWriter(archivoFile, true);  
bufferClave = new BufferedWriter(escriptorDeClaves);  
bufferClave.append(elementoClave + ",");  
bufferClave.close();  
escriptorDeClaves.close();
```

Figura 14: *Uso de buffers de escritura para añadir claves a un archivo.*

- **internalSortingArray** es utilizado durante la generación de los bloques de subclaves generados por Polifase. Cada bloque, recibido como una cadena de caracteres, es transformado a un arreglo de subclaves utilizando el método **split** definido para la clase *String*; el cual funciona de forma equivalente a *StringTokenizer*. Utilizando como delimitador el carácter “,” se añade cada clave a un arreglo dinámico por medio de un ciclo **for-each**; transformando las subclaves a enteros por medio del método **parseInt** definido para el envoltorio *Integer*.

Posteriormente, se utiliza el método de ordenamiento interno *Insertion-Sort* definido en la clase *Utilidades* sobre el arreglo dinámico. Con el arreglo ordenado, se realiza el proceso inverso, extrayendo del arreglo cada subclave ordenada y creando una cadena que, finalmente, es devuelta por el método.

```
dynamicAux = Utilidades.insertionSorting(dynamicAux, tipoDeOrdenamiento);  
cadenaDeInsercion = "";  
  
for (Integer clave : dynamicAux) {  
    cadenaDeInsercion = cadenaDeInsercion + clave + ",";  
}  
  
return cadenaDeInsercion;
```

Figura 15: *Ordenamiento de bloques de claves.*

- El método **lecturaLinea** permite leer y recuperar como una cadena de caracteres el contenido presente en una línea establecida como parámetro. Este método fue generado debido a que, entre cada iteración de Polifase, se escriben 2 saltos de línea para mantener una separación entre los bloques de claves que son escritos y recuperados de forma sucesiva.

```
case 0:

    for (int i = 0; i < numLineaDeLectura; i++) {
        this.lectorF01.readLine();
    }
    String cadenaLeidaF01 = this.lectorF01.readLine();
    return cadenaLeidaF01;

case 1:

    for (int i = 0; i < numLineaDeLectura; i++) {
        this.lectorF23.readLine();
    }
    String cadenaLeidaF23 = this.lectorF23.readLine();
    return cadenaLeidaF23;
```

Figura 16: *Lectura y recuperación de líneas transformadas a cadenas de caracteres.*

Una vez establecidos los métodos secundarios, se enlistarán los 2 métodos principales de la implementación de Polifase realizada:

- **ordenamientoPolifase** es el primer método inicial y es el que se invoca en el menú principal del programa. Recibiendo el nombre del archivo con las claves, el tamaño de los bloques de claves (Para efectos del proyecto, se ha establecido en 4) y el criterio de ordenamiento seleccionado, inicia invocando un recolector de basura denominado **resetArchivoAuxPolyphase** que reinializa los archivos de Polifase provenientes de anteriores ejecuciones.

```
Utilidades.resetArchivoAuxPolyphase("F0");
Utilidades.resetArchivoAuxPolyphase("F1");
Utilidades.resetArchivoAuxPolyphase("F2");
Utilidades.resetArchivoAuxPolyphase("F3");
```

Figura 17: *Invocación del recolector de basura.*

A continuación, realiza los siguientes pasos:

1. Recupera el arreglo anidado con las superclaves y, posteriormente, recupera la subclave 'Número de cuenta' de cada una de las superclaves y las almacena en otro arreglo dinámico; transformando en el proceso los datos a *Integers* para su correcta manipulación.
2. Por medio del método **appendKeyToFile**, se escriben cada una de las subclaves recuperadas en un archivo llamado F_0 , y se inicializa un proceso de lectura sobre el archivo F_0 , así como una escritura sobre los auxiliares F_1 y F_2 .


```
this.hubReader("F0", 0);  
this.hubWriter("F1", 1);  
this.hubWriter("F2", 2);
```

Figura 18: Invocación de los hub de lectura y escritura.

- Se establece un primer ciclo **while** tal que, mientras el lector establecido para F_0 encuentre una cadena de claves que no sea nula, establezca un identificador de claves por medio de *StringTokenizer* y un delimitador ", ". Luego, se establece por un ciclo **while** anidado que, por cada clave individual que sea recuperada, esta se almacene en un arreglo auxiliar. Mientras esta operación es realizada, un contador interno mantiene la cuenta de las claves almacenadas: cuando este valor llegue a 4, se imprime el bloque leído y se reinicializa el contador.

```
if (numElementosProcesados == tamañoDelBloque) {  
    bloqueProcesado++;  
  
    System.out.println("\nSe ha cargado un bloque.");  
    System.out.println("Estado actual : " + arrayAuxiliarF0);  
    numElementosProcesados = 0;  
  
    System.out.println("Bloques procesados: " + bloqueProcesado);
```

Figura 19: Generación de los bloques de subclaves.

- El contador **bloqueProcesado** se utiliza para enviar a F_1 o a F_2 el bloque cargado. Por ejemplo, para el primer bloque: $\text{bloqueProcesado} = 1$, usando la operación $\text{bloqueProcesado} \bmod 2$, si el resultado $\neq 0$, eso indica la presencia de un bloque impar que deberá ser enviado al archivo F_1 . En cambio, si $\text{bloqueProcesado} \bmod 2 = 0$, el bloque cargado es par y será enviado al archivo F_2 ; previamente transformado a una cadena de caracteres por medio de un ciclo **for-each** y ordenado por el método **internalSortingArray**.

```
//Impresión del bloque convertido a cadena y ordenado. Posteriormente, se escribe en F2.  
System.out.println("Bloque almacenado en F2: " + this.internalSortingArray(cadenaDeInsercion, criterioDeOrdenamiento));  
this.escriptorF23.write(this.internalSortingArray(cadenaDeInsercion, criterioDeOrdenamiento));  
this.escriptorF23.write(" ");
```

Figura 20: Inserción de un bloque impar sobre .

- Posteriormente, se verifica que sobre el arreglo auxiliar no existan subclaves adicionales que no alcancen el tamaño del bloque establecido. En caso de que suceda tal situación, el proceso anterior se repite, con la diferencia de que, por ejemplo, si la lectura de bloques con el tamaño

establecido terminó en F_1 , el contador **bloqueProcesado** tendrá con un valor impar y, por tanto, el bloque restante será enviado a F_2 .

```
//Si la lectura de bloques terminó en F1 con bloqueProcesado impar, enviará el bloque restante a F2.
} else {

    String cadenaDeInsercion = "";
    for (Integer clave : arrayAuxiliarF0) {
        cadenaDeInsercion = cadenaDeInsercion + clave + ",";
    }
    System.out.println("Bloque almacenado en F2: " + this.internalSortingArray(cadenaDeInsercion, criterioDeOrdenamiento));
    this.escriptorF23.write(this.internalSortingArray(cadenaDeInsercion, criterioDeOrdenamiento));
    this.escriptorF23.write(" ");
}
```

Figura 21: *Inserción de bloques con tamaño menor al establecido.*

6. Finalmente, se cierran cada uno de los buffers de lectura/escritura utilizados y se inicializa un proceso de lectura sobre F_0 para proceder al segundo método principal del ordenamiento: **mergingPolyphase**.
- El método **mergingPolyphase** únicamente recibe como parámetro el criterio de ordenamiento seleccionado por el usuario. A continuación, realiza los siguientes pasos:
 1. Inicializa por medio del método **hubReader** y **hubWriter** un proceso de lectura sobre los bloques de claves presentes en los archivos F_1 y F_2 y prepara un proceso de escritura en los archivos F_0 y F_3 . Por medio de un ciclo **do-while**, se recuperan las claves presentes en la línea de lectura actual de la iteración (Recordando que, entre cada iteración de bloques escritos en los archivos, se añaden 2 espacios en blanco) y se imprimen en pantalla. Posteriormente, los bloques de claves son separados por medio del método **split** usando como identificador el espacio presente entre cada bloque de claves.

```
System.out.println("Bloque 1: " + cadenaAMezclar1);
System.out.println("Bloque 2: " + cadenaAMezclar2);

/*Arreglos auxiliares utilizados para almacenar y fusionar
los bloques de claves.*/
String bloqueDeClavesF1[] = cadenaAMezclar1.split(" ");
String bloqueDeClavesF2[] = cadenaAMezclar2.split(" ");
```

Figura 22: *Recuperación de bloques de clave.*

2. Debido a la naturaleza de la generación de particiones de Polifase, existe la posibilidad de que el segundo arreglo generado en [22] sea de menor tamaño (Contiene menor cantidad de bloques que). Por tanto, implica que su manipulación y mezcla por medio de un ciclo **for** inevitablemente resultaría en una excepción de índice inexistente. Para evitar tal situación, se añade un

bloque *placeholder* al segundo arreglo en caso de que este sea de menor tamaño que el primer arreglo. De esta forma, se asegura que ambos arreglos sean siempre del mismo tamaño.

```
System.out.println("\nAlerta: Se ha detectado que uno de los archivos contiene más bloques que el otro.");
cadenaAMezclar2 = cadenaAMezclar2 + "!";
System.out.println("Se ha añadido un bloque adicional:\nBloque 2: " + cadenaAMezclar2);
bloqueDeClavesF2 = cadenaAMezclar2.split(" ");
```

Figura 23: Verificación y modificación sobre el segundo arreglo.

Durante el proceso de intercalado de ambos bloques, el bloque *placeholder* es eliminado por el método **internalSortingArray**.

3. Hecho lo anterior, se inicializa un ciclo **for** que, considerando el valor de la variable **bloquesProcesados** (la cuál indica el número de bloques que han sido intercalados), escribirá sucesivamente sobre F_0/F_3 o F_1/F_2 , según corresponda.

```
if (bloquesProcesados % 2 == 0) { //Cuenta las veces que se realiza el intercalado de bloques

    //Escritura sobre F0.
    if (i % 2 == 0) {

        this.escriptorF01.write(this.internalSortingArray((bloqueDeClavesF1[i] + bloqueDeClavesF2[i]));
        System.out.println("\nBloque fusionado y añadido a F0: " + this.internalSortingArray((bloqueDeClavesF1[i] + bloqueDeClavesF2[i]));
        this.escriptorF01.write(" ");

        //Escritura sobre F3.
    } else {

        this.escriptorF23.write(this.internalSortingArray((bloqueDeClavesF1[i] + bloqueDeClavesF2[i]));
        System.out.println("Bloque fusionado y añadido a F3: " + this.internalSortingArray((bloqueDeClavesF1[i] + bloqueDeClavesF2[i]));
        this.escriptorF23.write(" ");

    }

} else {
```

Figura 24: Escritura de bloques intercalados y ordenados por medio de Insertion-Sort.

4. Una vez hecho la escritura intercalada de los bloques, se añaden 2 saltos de línea en los archivos y se cierran los escritores utilizados. Posteriormente, se verifica si los bloques procesados se escribieron sobre F_0/F_3 o F_1/F_2 para inicializar su lectura y repetir el proceso de intercalado sobre el otro par de archivos auxiliares.

En caso de que se detecte que alguna de las cadenas de bloques recuperadas no contiene elementos, se rompe el ciclo **while** y se indica en pantalla que los datos han sido finalmente ordenados.

```
//Impresiones en pantalla del estado actual de la ejecución.
if (bloquesProcesados % 2 == 0) {

    //Detecta si una de las cadenas de bloques recuperadas ya no contienen más claves por fusionar y detiene la ejecución.
    if (cadenaAMezclar1.equals("") || cadenaAMezclar2.equals("")) {
        System.out.println("¡Felicidades! Los datos han sido ordenados.\n\nSaliendo de la opción...");
        this.lectorF01.close();
        this.lectorF23.close();
        this.escriptorF01.close();
        this.escriptorF23.close();
        break;
    }

    System.out.println("\n\nLos archivos F0 y F3 contienen los siguientes bloques: ");
    //Inicializa escritura para F1 y F2.
    this.hubWriter("F1", 1);
    this.hubWriter("F2", 2);

    //Inicializa lectura para F0 y F3.
    this.hubReader("F0", 0);
    this.hubReader("F3", 1);
}
```

Figura 25: Condición de ruptura del ciclo while e inicialización de escritura sobre F₁/F₂.

5.1.2. Polifase no numérica

El ordenamiento de claves no numéricas contiene diferencias con respecto al enfoque numérico, por tanto, su implementación se encuentre contenida sobre otra clase que hereda directamente de *Polifase*: a esta clase se le ha denominado *PolifaseStrings*. A continuación se mostrarán las diferencias con respecto a la implementación anterior.

Se crea un arreglo el cual contendrá nuestras de claves de tipo String para almacenar claves no numéricas, como los nombres o apellidos.

```
public class PolifaseStrings extends Polifase {

    //Arreglo auxiliar en la recuperación de claves no numéricas.
    ArrayList<String> arrayF0S;
```

Figura 26: Arreglo auxiliar en la recuperación de claves no numericas.

La recuperación de las claves presenta modificaciones: debido a que es posible seleccionar entre Nombre o Apellido, con valor (1) o (2) respectivamente durante la selección del campo en el menú principal, se obtendrán las claves de acuerdo al campo que fue escogido, restando -1 tomando en consideración el índice inicial del arreglo siendo 0.

El ciclo de lectura y generación de bloques es modificado para adaptarse a la lectura de claves no numéricas. El resto del procedimiento se mantiene sin cambios.

```
//Se obtiene el arreglo dinámico con las claves del archivo introducido.
lecturaDF0 = Utilidades.scanSuperKeys(nombreDelArchivo);

//Arreglo dinámico utilizado para seleccionar y almacenar las claves no numéricas (Nombre y apellido).
arrayFOS = new ArrayList<>();

for (int i = 0; i < lecturaDF0.size(); i++) {

    arrayFOS.add(i, (lecturaDF0.get(i).get(campoSeleccionado - 1)));

}
```

Figura 27: Recuperación de claves no numéricas.

```
StringTokenizer identificadorDeClaves = new StringTokenizer(cadenaDeClaves, ",");

while (identificadorDeClaves.hasMoreTokens()) {

    //Recuperación de las claves almacenadas en el archivo F0.
    arrayAuxiliarFOS.add(identificadorDeClaves.nextToken());
    numElementosProcesados++;

    if (numElementosProcesados == tamañoDelBloque) {
        bloqueProcesado++;

        System.out.println("\nSe ha cargado un bloque.");
        System.out.println("Estado actual : " + arrayAuxiliarFOS);
        numElementosProcesados = 0;

        System.out.println("Bloques procesados: " + bloqueProcesado);

        //Bloques par.
        if (bloqueProcesado % 2 == 0) {

            String cadenaDeInsercion = "";

            //Se almacenan las claves de cada bloque en una cadena.
            for (String clave : arrayAuxiliarFOS) {
                cadenaDeInsercion = cadenaDeInsercion + clave + ",";
            }

        }

    }

}
```

Figura 28: Ciclo de lectura de claves y almacenado en un arreglo auxiliar.

El método **internalSortingArray** es modificado para admitir el ordenamiento de claves no numéricas. El resto del procedimiento, sigue sin cambios adicionales; más allá de las modificaciones a los ciclos **for-each** implementados que iterarán sobre Strings.

```
//Impresión del bloque convertido a cadena y ordenado. Posteriormente, se escribe en F2.  
System.out.println("Bloque almacenado en F2: " + this.internalSortingArrayStrings(cadenaDeInsercion,  
this.escriptorF23.write(this.internalSortingArrayStrings(cadenaDeInsercion, criterioDeOrdenamiento));  
this.escriptorF23.write(" ");
```

Figura 29: Ciclo para lectura de claves y guardarlas en los arreglos dinámicos.

El método **mergingPolyphaseString** es una modificación de **mergingPolyphase** adaptado al procesamiento de claves no numéricas. En general, el cambio más significativo reside en el uso del método modificado **internalSortingArrayStrings** que permite ordenar claves no numéricas. Así mismo, el resto de métodos han sido adaptados para procesar claves de tipo String.

5.2. Simulación de Radix Externo

Los atributos de la clase que se utilizará para este algoritmo incluyen:

- Las rutas donde se encontrarán los archivos utilizados para la implementación de este algoritmo de ordenamiento externo.
- Elementos para escribir las claves a ordenar.
- Elementos para leer las claves a ordenar.
- Un arreglo auxiliar para leer las claves de archivo F_0 , el cual será de tipo ArrayList con elementos tipo ArrayList de cadenas de caracteres.

```
19 public class RadixExterno {  
20  
21     static String rutaArchivo = "./Archivos/";  
22     static String rutaAuxiliar = "./Archivos/RadixExterno/";  
23  
24     FileWriter escritorDeClaves;  
25     BufferedWriter bufferClave;  
26  
27     StringTokenizer identificador;  
28     Scanner lectura;  
29  
30     static ArrayList<ArrayList<String>> lecturaDF0;  
31
```

Figura 30: Atributos de Radix Externo.

Así mismo la clase cuenta con varios métodos para la implementación de este algoritmo. A continuación, se dará una vista general de ellos; la documentación completa puede ser consultada en el código de la clase RadixExterno.java

■ ordenamientoRadix

Este primer método es el método principal de Radix, recibe 2 parametros, el primero es el nombre del archivo que contiene la claves que deben ordenarse el cual no es el original, es más bien una copia, esto se hizo con el propósito de no modificar el archivo original para poder hacer más ejecuciones posteriormente. El segundo parámetro es el criterio que se utilizará para hacer el ordenamiento, ya sea ascendente o descendente, según lo indique el usuario..

Este método posee un contador de iteraciones para que el usuario pueda hacer el seguimiento de los pasos que hace el algoritmo.

Contempla posibles errores de escritura, los evita borrando algunos archivos o limpiándolos para que no haya problemas por corridas anteriores.

Así mismo, crea e indica las posiciones significativas que el algoritmo va tomando en cuenta conforme realiza el ordenamiento.

Uno de los aspectos relevante es el uso de simulaciones de estructuras de tipo FIFO que contribuyen al ordenamiento mediante este algoritmo, se habla de una simulación debido a que no se declara como tal una FIFO, pero la manera en la que se leen las claves es similar a como se haría en una estructura FIFO, los detalles sobre la creación de estas se explicará más adelante en **queueBuilder1k** y **queueBuilder**

```
60 impresionArchivo(nombreDelArchivo + "Sorted");
61
62 for (int i = 10; i <= 100000; i *= 10) {
63
64     contadorIteraciones++;
65     System.out.println("-----Iteración actual del ordenamiento: " + contadorIteraciones);
66     recolectorDeBasura();
67     System.out.println("\nSe ha actualizado la posición significativa a: " + i);
68     System.out.println("");
69     queueBuilder1K(nombreDelArchivo + "Sorted", i);
70     Utilidades.eliminarArchivo(nombreDelArchivo + "Sorted");
71     criterioSeleccionado(nombreDelArchivo, tipoDeOrdenamiento);
72
73     impresionArchivo(nombreDelArchivo + "Sorted");
74 }
```

Figura 31: Método principal.

■ criterioSeleccionado

Para este método es necesario obtener dos parámetros: nombre del archivo donde se encuentran las claves ordenar para leer las claves y el tipo de ordenamiento para establecer el orden de los archivos auxiliares.

Hace una llamada al método **sortAux** que se explicará más adelante.

```
88 public void criterioSeleccionado(String nombreDelArchivo, int tipoDeOrdenamiento) {
89
90     if (tipoDeOrdenamiento == 1) {
91
92         sortRadix("archivoN0", nombreDelArchivo);
93         sortRadix("archivoN1", nombreDelArchivo);
94         sortRadix("archivoN2", nombreDelArchivo);
95         sortRadix("archivoN3", nombreDelArchivo);
96         sortRadix("archivoN4", nombreDelArchivo);
97         sortRadix("archivoN5", nombreDelArchivo);
98         sortRadix("archivoN6", nombreDelArchivo);
99         sortRadix("archivoN7", nombreDelArchivo);
100        sortRadix("archivoN8", nombreDelArchivo);
101        sortRadix("archivoN9", nombreDelArchivo);
102    } else {
103        sortRadix("archivoN9", nombreDelArchivo);
104        sortRadix("archivoN8", nombreDelArchivo);
105        sortRadix("archivoN7", nombreDelArchivo);
106        sortRadix("archivoN6", nombreDelArchivo);
107        sortRadix("archivoN5", nombreDelArchivo);
108        sortRadix("archivoN4", nombreDelArchivo);
109        sortRadix("archivoN3", nombreDelArchivo);
110        sortRadix("archivoN2", nombreDelArchivo);
111        sortRadix("archivoN1", nombreDelArchivo);
112        sortRadix("archivoN0", nombreDelArchivo);
113    }
114 }
115
116 }
```

Figura 32: Establece el orden de creación de los archivos auxiliares dependiendo del criterio de ordenamiento que el usuario seleccionó.

■ queueBuilder1K

Clasifica las claves para para asignarla al archivo auxiliar correspondiente, lo logra haciendo la lectura del archivo donde se encuentran las claves a ordenar. Es necesario utilizar otro método de la misma clase **RadixExterno.java** llamado **escrituraAuxiliar** para escribir las claves obtenidas en los archivos auxiliares según corresponda. Puede diferenciar cuando una clave termina porque toma en cuenta cuando encuentra comas “,”

Este método se usa para leer claves de los archivo que se van generando conforme se avanza en los dígitos significativos, hay un método similar a el llamado **queueBuilder**, descrito más adelante.


```

127 public void queueBuilder1K(String nombreDelArchivo, double digitoSignificativo) {
128
129     try {
130
131         File archivoLectura = new File(this.rutaArchivo + nombreDelArchivo + ".txt");
132         lectura = new Scanner(archivoLectura);
133
134         while (lectura.hasNextLine()) {
135
136             String saltoLinea = lectura.nextLine();
137             identificador = new StringTokenizer(saltoLinea, ",");
138
139             while (identificador.hasMoreTokens()) {
140
141                 int elementoClave = (int) Double.parseDouble(identificador.nextToken());
142                 System.out.println("Número de cuenta: " + elementoClave);
143
144                 int posicionOrdenamiento = (int) ((elementoClave % (digitoSignificativo * 10)) / digitoSignificativo);
145                 System.out.println("Posición de ordenamiento significativa: " + posicionOrdenamiento);
146
147                 switch (posicionOrdenamiento) {

```

Figura 33: *Clasifica las claves desde la iteración segunda en adelante.*

■ queueBuilder

A diferencia de método anterior **queueBuilder1K** este método sólo se utiliza en la primera iteración, por ello recibe como primer parámetro el arreglo en el que estarán las claves al leer el archivo original.

Como segundo parámetro recibe el dígito significativo, con base a esto clasifica las claves que va obteniendo del archivo que recibe como primer parámetro. Los escribe con ayuda del método **escrituraAuxiliar**.

```

246 public void queueBuilder(ArrayList<ArrayList<String>> archivoF0, double digitoSignificativo) {
247
248     for (int i = 0; i < archivoF0.size(); i++) {
249
250         int elementoClave = (int) Double.parseDouble(archivoF0.get(i).get(2));
251         System.out.println("Número de cuenta: " + elementoClave);
252
253         int posicionOrdenamiento = (int) ((elementoClave % (digitoSignificativo * 10)) / digitoSignificativo);
254         System.out.println("Posición de ordenamiento significativa: " + posicionOrdenamiento);
255
256         switch (posicionOrdenamiento) {

```

Figura 34: *Clasifica las claves en la primera iteración.*

■ appendKeyToFile

Recibe como parámetros una cadena de caracteres que indica el nombre del archivo auxiliar donde escribirá la clave que recibe como segundo parámetro.

Hace la verificación de existencia del archivo, en caso de que el archivo no haya sido creado, lo crea para poder hacer la escritura de la clave correspondiente. Los archivos son verificados en la subcarpeta de `.archivos` llamada `RadixExterno`.

```
352 | | | | | escritorDeClaves = new FileWriter(archivoAuxiliar, true);
353 | | | | | bufferClave = new BufferedWriter(escritorDeClaves);
354 | | | | | bufferClave.append(elementoClave + ",");
355 | | | | | bufferClave.close();
356 | | | | | escritorDeClaves.close();
```

Figura 35: *Escribe las claves en sus respectivo archivo auxiliar.*

■ sortAux

Con este método es posible finalizar el proceso de ordenamiento en cada iteración, pues este método el que lee las claves de los archivos auxiliares que recibe como primer parámetro y las escribe en las copia el archivo original creada desde un inicio, es decir el archivo que contenía las claves en desorden, es por eso que necesita recibir como parámetro el nombre de este archivo.

```
while (identificador.hasMoreTokens()) {

    String elementoClave = identificador.nextToken();
    bufferClave.append(elementoClave + ",");

}
```

Figura 36: *Método que concluye las iteraciones.*

■ recolectorDeBasura

La única función de este método es eliminar archivos auxiliares en caso de que contengan claves, estos archivos son de ejecuciones anteriores y es necesario eliminarlos para evitar problemas con la ejecución actual.

```
422 public void recolectorDeBasura() {  
423  
424     String ubicacionAuxiliares = RadixExterno.rutaAuxiliar;  
425     File archivosAEliminar = new File(ubicacionAuxiliares);  
426     File[] archivosAuxiliares = archivosAEliminar.listFiles();  
427  
428     if (archivosAuxiliares.length != 0) {  
429  
430         for (File archivosAuxiliare : archivosAuxiliares) {  
431             File archivoActual = new File(archivosAuxiliare.toString());  
432             /*Eliminando esta línea se pueden observar los archivos auxiliares... aunque de una forma muy fea  
433             que afecta al resultado final. */  
434             archivoActual.delete();  
435         }  
436     }  
437 }  
438  
439 }
```

Figura 37: *Elimina archivos auxiliares de ejecuciones anteriores.*

■ impresionArchivo

Con apoyo de este método es posible escribir las claves correspondientes en el archivo que recibe como parámetro, este archivo será el que tenía las claves desordenadas y lo que se obtiene finalmente es un el archivo original pero con las claves ya ordenadas, es decir que con este método se finaliza el algoritmo.

Es importante mencionar que cuando este método se utiliza en la primera iteración del algoritmo, obtiene las claves que han pasado por la primera iteración pero no necesariamente el archivo que recibe como parámetro este método tendrá la claves totalmente ordenadas.

```
454 while (lectura.hasNextLine()) {  
455  
456     String saltoLinea = lectura.nextLine();  
457     identificador = new StringTokenizer(saltoLinea, ",");  
458  
459     while (identificador.hasMoreTokens()) {  
460  
461         System.out.println(identificador.nextToken() + ",");  
462     }  
463 }
```

Figura 38: *Vacía las claves ya ordenadas a la copia del archivo original.*

5.3. Simulación de la Mezcla Natural

La programación de este método requiere consideraciones adicionales respecto a la generación de bloques:

1. Los bloques no son fijos y son de máximo tamaño, siguiendo un criterio ascendente o descendente.
2. Se requiere una comparación entre los elementos de cada arreglo; operación no realizable si no se dispone de arreglos auxiliares que guarden los bloques contenidos en cada archivo.
3. Un mayor control de los bloques leídos y procesados es necesario para implementar efectivamente la escritura sobre los archivos.

Con lo anterior en mente, se han añadido los siguientes atributos a la clase *NaturalMix*:

```
//Rutas de acceso a las carpetas y archivos.
static String rutaArchivo = "./Archivos/";
static String rutaAuxiliar = "./Archivos/MezclaNatural/";
File archivoF0 = new File(NaturalMix.rutaAuxiliar + "F0" + ".txt");
File archivoF1 = new File(NaturalMix.rutaAuxiliar + "F1" + ".txt");
File archivoF2 = new File(NaturalMix.rutaAuxiliar + "F2" + ".txt");

//Buffer de lectura para los archivos.
BufferedReader bufferLector;

//Buffers de escritura para los archivos.
BufferedWriter bufferEscritor;

//Declaraciones a objetos FileReader y FileWriter.
FileWriter escritorDeClaves;
FileWriter escritor;
FileReader lectorArchivo;

//Arreglos auxiliares en la recuperación inicial de claves.
ArrayList<ArrayList<String>> lecturaDF0;
ArrayList<Integer> arrayF0;

//Arreglos contenedores de los bloques generados tras realizar particiones.
ArrayList<ArrayList> arrayBlocksF0 = new ArrayList<>();
ArrayList<ArrayList> arrayBlocksF1 = new ArrayList<>();
ArrayList<ArrayList> arrayBlocksF2 = new ArrayList<>();

//Arreglos auxiliares con las claves presentes en cada bloque.
ArrayList<Integer> arrayAuxiliarF0;
ArrayList<Integer> arrayAuxiliarF1;
ArrayList<Integer> arrayAuxiliarF2;

//Identificador para separar y manipular claves.
StringTokenizer identificador;

//Iterador de línea de operación actual.
int lineaDeOperacion = 0;

//Contador de iteraciones del programa sobre el archivo F0.
int contadorIteraciones = 0;
```

Figura 39: Atributos de *NaturalMix*.

Los arreglos anidados **arrayBlocksFX** serán utilizados para el procesamiento de los bloques de claves presentes en los archivos, donde cada bloque es un arreglo con sus respectivas claves en orden ascendente o descendente, según corresponda. Así mismo, se inicializan archivo especiales para F_0 , F_1 Y F_2 .

A continuación se dará un listado de los métodos de la clase. La documentación completa se encuentre en *NaturalMix.java*

- **blockBuilder** es el método encargado de realizar las particiones sucesivas sobre las claves presentes en el archivo F_0 , previamente leídas y almacenadas en un arreglo, y realiza las particiones de máximo tamaño, de acuerdo al criterio seleccionado. El proceso se puede resumir en los siguientes pasos:
 1. Se obtiene el primer elemento del arreglo y se coloca sobre un arreglo auxiliar que representa un bloque a insertar sobre F_1 . Por medio de 2 variable de control, se comienza a iterar sobre el arreglo recibido y se añaden a la lista auxiliar los elementos que cumplan la condición máxima para cada partición.
 2. Cuando se detecta que un elemento no cumple el criterio establecido, se establece una primera partición que es cargada, en el caso particular del primer bloque procesado, sobre el arreglo **arrayBlocksF2** por medio del método **addBlock** y por medio de la variable **bloqueActual**, se inicia una nueva partición que será escrita sobre F_2 .
 3. Finalmente, cualquier bloque que no haya sido tratado se añade a su respectivo arreglo, en caso de existir alguno. Finalmente, se imprimen las particiones almacenadas en **arrayBlocksF1** y **arrayBlocksF2** y se retorna la variable booleana **particionCreada**; que tendrá un valor *true* en caso de que se haya realizado una partición válida sobre el arreglo recibido.

```
claveAnterior = claveActual;
particionCreada = true;

/*Se añade la clave que no cumple el criterio ascendente
a F2 y el bloque anterior se carga sobre F1.*/
if (bloqueActual == 1) {

    arrayAuxiliarF2 = new ArrayList<>();
    arrayAuxiliarF2.add(claveActual);
    addBlock(arrayAuxiliarF1, bloqueActual);
    //Se procede a generar una partición para F2.
    bloqueActual = 2;
```

Figura 40: Carga de los bloques de máximo tamaño.

- El método **addBlock** se encarga de añadir un bloque de claves recibido a los arreglos **arrayBlocksF1** y **arrayBlocksF2**, según corresponda.

```
if (bloqueActual == 1) {
    this.arrayBlocksF1.add(arrayAuxiliar);
} else {
    this.arrayBlocksF2.add(arrayAuxiliar);
}
```

Figura 41: Estructura del método.

- El método **saltoLinea** añade 2 saltos de línea sobre el archivo introducido como parámetro. Este método es utilizado para separar las iteraciones almacenadas en cada uno de los archivos que ocupa *MezclaNatural* durante su ejecución.

```
try {
    escritor = new FileWriter(archivoTarget, opDeseada);
    bufferEscritor = new BufferedWriter(escritor);
    bufferEscritor.append("\n\n");
    bufferEscritor.close();
}
```

Figura 42: Saltos de línea sobre el archivo introducido como parámetro.

- El método **appendBlockToFile** se encarga de escribir un bloque de claves recibido como parámetro sobre el archivo seleccionado. Por defecto en la implementación realizado, la variable booleana **opDeseada** siempre tiene un valor *true*. También se añade el método **appendKeyToFile** de Polifase como auxiliar en la generación inicial del arreglo contenedor de claves en F_0 .

```
escritor = new FileWriter(archivoTarget, opDeseada);
bufferEscritor = new BufferedWriter(escritor, 1);

for (int i = 0; i < bloqueClave.size(); i++) {
    bufferEscritor.append(bloqueClave.get(i) + ",");
}
bufferEscritor.close();
escritor.close();
```

Figura 43: Escritura de bloques sobre archivos.

- **fileToArray** es una modificación del método **scanSuperKeys** adaptada exclusivamente para su uso sobre la implementación de la Mezcla Natural. A diferencia del método anterior, este recibe directamente el archivo F_0 y transforma la lectura de claves en un arreglo almacenado en **arrayBlocksF0**; el cual contiene las lecturas sucesivas de claves recuperadas tras cada iteración del programa.

```

while (identificador.hasMoreTokens()) {
    claveLeida = Integer.parseInt(identificador.nextToken());

    arrayAuxiliarF0.add(claveLeida);
}
this.arrayBlocksF0.add(arrayAuxiliarF0);
}

bufferLector.close();
lectorArchivo.close();
System.out.println("Claves procesadas en F0: ");
System.out.println(arrayAuxiliarF0);

```

Figura 44: Lecturas de claves en F_0 almacenada en el arreglo de control de F_0 .

Los métodos principales de la implementación son los siguientes:

- El método **mezclaNatural** recibe el nombre del archivo contenedor de claves y el criterio de ordenamiento seleccionado. Tras realizar un proceso de limpieza de los archivos de ejecuciones anteriores, se recupera el arreglo anidado con las superclaves y, finalmente, se obtiene el arreglo con las claves, en este caso particular, numéricas para insertarlas en un archivo llamado **F0** en la carpeta *Archivos* – \rightarrow *MezclaNatural* por medio del método **appendKeyToFile**. Posteriormente, se añade al arreglo **arrayBlocksF0** el primer arreglo de claves leído en el archivo F_0 .

Hecho lo anterior, se implementa un **switch** que, de acuerdo al criterio de ordenamiento seleccionado, realizará particiones sucesivas del contenido en F_0 almacenado; de acuerdo a la línea de operación actual del programa, y ejecutará el método **mixSort** encargado de procesar las particiones realizadas. Cuando el método **blockBuilder** informe que no se han realizado más particiones en F_0 , el ciclo **while** implementado para repetir las operaciones anteriores se romperá y terminará la ejecución del ordenamiento.

```

//Mientras se detecte la presencia de una partición válida sobre F0, se realizará el proceso de Mezcla.
while ((blockBuilder((ArrayList<Integer>) arrayBlocksF0.get(lineaDeOperacion), criterioDeOrdenamiento)) == true) {
    mixSort(archivoF0, criterioDeOrdenamiento);
}
System.out.println("\nNo se han detectado más particiones sobre F0.");
System.out.println("¡Felicidades! Los datos han sido ordenados.\n\nSaliendo de la opción...\n");
break;

```

Figura 45: Lecturas de claves en F_0 almacenada en el arreglo de control de F_0 .

- **blockMerge** es el método encargado de realizar la mezcla o intercalado de los bloques presentes en los archivos F_1 Y F_2 . En primer lugar, se inicializan listas auxiliares que representan un bloque de claves extraído sobre los archivos auxiliares. A continuación, se imprimen los bloques de claves

presentes en F_1 Y F_2 y se declaran 2 variables booleanas que indican si existe un bloque para ser mezclado en los archivos auxiliares. A continuación, se recupera un bloque de cada arreglo representativo de los bloques presentes en F_1 Y F_2 y su contenido se añade a los arreglos secundarios declarados anteriormente.

```
//Por medio de un ciclo for anidado, se extrae un bloque de claves en F1 con sus respectivas subclaves.
for (ArrayList blockKeysF1 : arrayBlocksF1) {

    if (flagF1 == true) {

        for (int i = 0; i < blockKeysF1.size(); i++) {

            elementoActual = (int) blockKeysF1.get(i);
            mixKeysF1.add(elementoActual);
            flagF1 = false;
        }
    }
}
```

Figura 46: Nótese que únicamente se recupera un único bloque para ser mezclado.

Realizado el procesamiento anterior, por medio de 2 ciclos **for-each** implementados, se recuperan las claves obtenidas por los arreglos secundarios y se almacenan en un arreglo llamado **mixedArray**. Finalmente, el arreglo obtenido es ordenado por medio del método **insertionSorting** de la clase *Utilidades*, se imprime el resultado y se devuelve el arreglo obtenido.

```
//Se procede a mezclar los arreglos auxiliares con las claves recuperadas.
ArrayList<Integer> mixedArray = new ArrayList<>();

for (Integer clave : mixKeysF1) {
    mixedArray.add(clave);
}

for (Integer clave : mixKeysF2) {
    mixedArray.add(clave);
}
```

Figura 47: Generación del arreglo mezclado que contiene 2 bloques de claves intercaladas.

- El método **mixSort** se encarga de escribir sobre los archivos F_1 Y F_2 las particiones almacenadas en los arreglos de control **arrayBlocksF1** y **arrayBlocksF2**, respectivamente. Lo anterior es realizado por medio del método **forEach** definido para la clase *ArrayList*.

Una vez hecha la escritura, por medio de un ciclo **while** se proceden a añadir al archivo F_0 cada uno de los bloques intercalados por medio del método **blockMerge**. De esta forma, se realiza el intercalado de cada uno de los bloques y se realiza su escritura sobre F_0 .


```

/*Por medio de un ciclo for-each se recupera cada bloque almacenado
en el arreglo de bloques correspondiente a F1 y se escriben sobre
el archivo. */
arrayBlocksF1.forEach(
    (i) -> {
        System.out.println("Bloque cargado exitosamente en F1: " + i);
        appendBlockToFile(this.archivoF1, i, true);
    }
);
//Se escriben 2 saltos de línea en F1 para separar las iteraciones.
saltoLinea(this.archivoF1, true);

```

Figura 48: Escritura de los particiones actuales sobre F_1 y F_2 . Se insertan 2 saltos de línea tras cada escritura.

```

while (!arrayBlocksF1.isEmpty() && !arrayBlocksF2.isEmpty()) {
    appendBlockToFile(archivoF0, blockMerge(criterioDeOrdenamiento), true);
}

```

Figura 49: Proceso de escritura de los bloques mezclados sobre F_0 , siempre y cuando ambos arreglos tengan bloques válidos por intercalar.

Finalmente, cualquier bloque que no hubiese sido añadido a F_0 , es añadido haciendo una verificación sobre los arreglos de control de F_1 y F_2 . Luego, se incrementa en 2 unidades la línea de operación actual de la ejecución y se incrementa el contador de iteraciones realizadas. El método termine imprimiendo un mensaje de estado, limpia el arreglo de control de F_0 e inicia una nueva lectura sobre F_0 . Es importante señalar el papel que la variable **líneaDeOperacion** tiene debido a que, tras inspeccionar nuevamente F_0 , el arreglo **arrayBlocksF0** tendrá 3 índices en su interior: la iteración original, los saltos de línea agregados, y escritura realizada de las claves mezcladas. Por medio del incremento en 2 unidades de esta variable, se accede correctamente a las escrituras sucesivas que se realizan en F_0 ; como se muestra en [50].

```

/*Se aumenta la línea de operación; tomando en cuenta los saltos de línea
que separan las iteraciones del programa.*/
this.líneaDeOperacion = líneaDeOperacion + 2;
this.contadorIteraciones++;

System.out.println("\nSe han añadido los bloques mezclados a F0 exitosamente.");
System.out.println("Iteración realizada: " + (this.contadorIteraciones));
//Limpieza del arreglo con los bloques en F0 con el de iniciar una nueva inspección a F0.
arrayBlocksF0.clear();
System.out.println("");
fileToArray(archivoF0);

```

Figura 50: Procedimiento final del método.

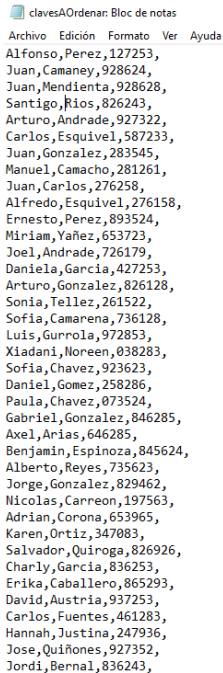
6. Mezcla Natural no numérica

El ordenamiento de claves no numéricas contiene diferencias con respecto al enfoque numérico, en un caso semejante al de Polifase. Su implementación se encuentre contenida sobre otra clase que hereda directamente de *NaturalMix*: a esta clase se le ha denominado *NaturalMixStrings*. A continuación se enlistan los cambios realizados:

- En el método **mezclaNatural** ahora se recuperan claves no numéricas de acuerdo al campo seleccionado: nombre o apellido.
- Los métodos **blockBuilder**, **blockMerge**, **appendBlockToFile** y **appendKeyToFile** han sido adaptados para recibir y procesar claves de tipo String.
- Se han sobrescrito los métodos **mixSort** y **fileToArray** para que operen adecuadamente al leer claves String y el proceso de intercalado y ordenamiento interno funcione adecuadamente.

7. Pruebas de funcionamiento y rendimiento

A continuación se mostrarán capturas de pantalla del funcionamiento del programa haciendo uso de diversos criterios y utilizando un archivo de 100 claves, como puede verse a continuación:

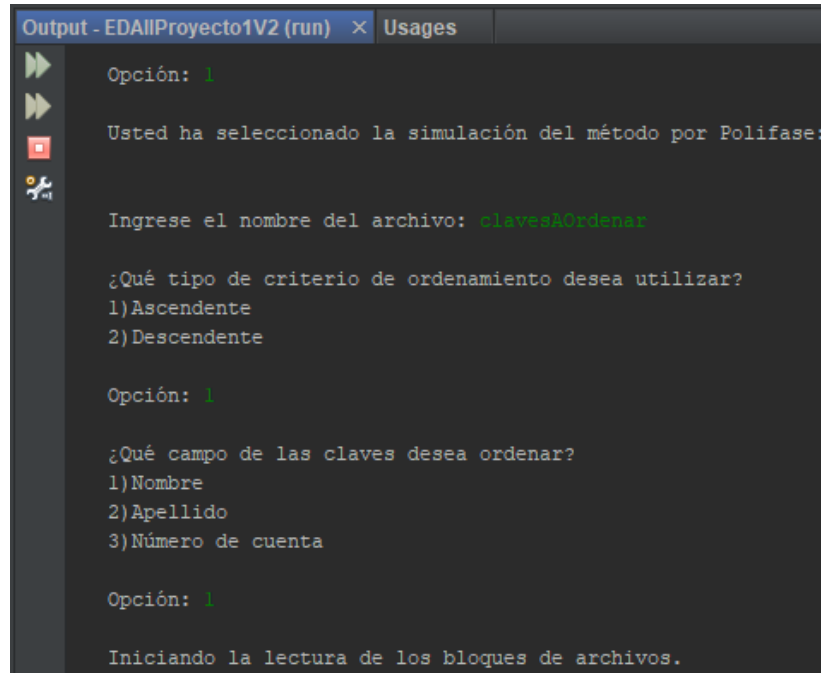


```

clavesAOOrdenar: Bloc de notas
Archivo Edición Formato Ver Ayuda
Alfonso, Perez, 127253,
Juan, Camaney, 928624,
Juan, Mendianta, 928628,
Santiago, Rios, 826243,
Arturo, Andrade, 927322,
Carlos, Esquivel, 587233,
Juan, Gonzalez, 283545,
Manuel, Camacho, 281261,
Juan, Carlos, 276258,
Alfredo, Esquivel, 276158,
Ernesto, Perez, 893524,
Miriam, Yañez, 653723,
Joel, Andrade, 726179,
Daniela, Garcia, 427253,
Arturo, Gonzalez, 826128,
Sonia, Tellez, 261522,
Sofia, Camarena, 736128,
Luis, Gurrola, 972853,
Xiadani, Noreen, 038283,
Sofia, Chavez, 923623,
Daniel, Gomez, 258286,
Paula, Chavez, 073524,
Gabriel, Gonzalez, 846285,
Axel, Arias, 646285,
Benjamin, Espinoza, 845624,
Alberto, Reyes, 735623,
Jorge, Gonzalez, 829462,
Nicolas, Carreon, 197563,
Adrian, Corona, 653965,
Karen, Ortiz, 347083,
Salvador, Quiroga, 826926,
Charly, Garcia, 836253,
Erika, Caballero, 865293,
David, Austria, 937253,
Carlos, Fuentes, 461283,
Hannah, Justina, 247936,
Jose, Quiñones, 927352,
Jordi, Bernal, 836243,

```

Figura 51: Vista previo del archivo con 100 claves a utilizar.



```

Output - EDAlProyecto1V2 (run) x Usages

Opción: 1

Usted ha seleccionado la simulación del método por Polifase:

Ingrese el nombre del archivo: clavesAOrdenar

¿Qué tipo de criterio de ordenamiento desea utilizar?
1)Ascendente
2)Descendente

Opción: 1

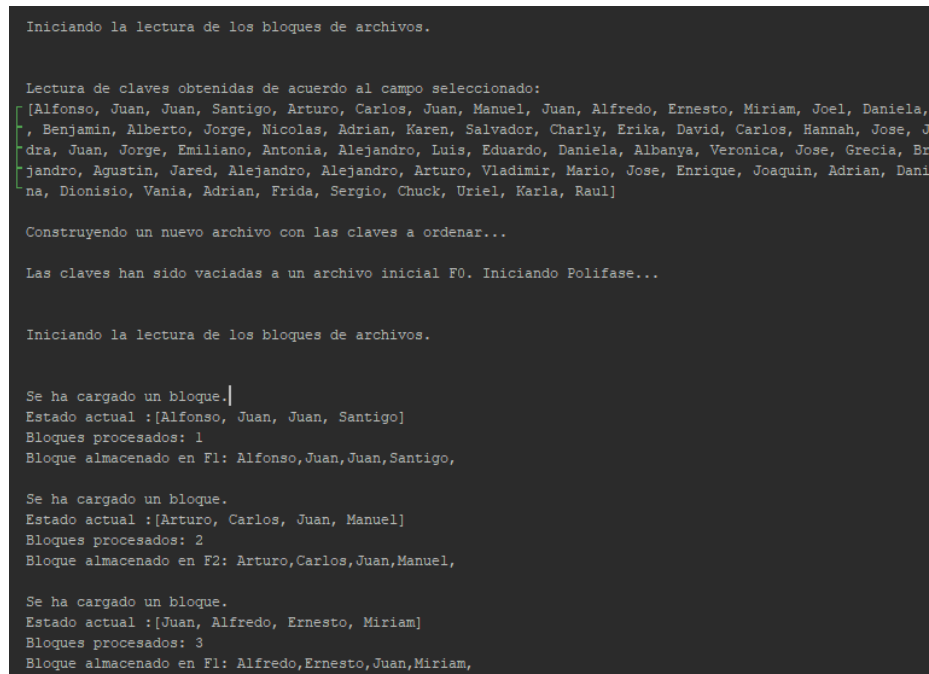
¿Qué campo de las claves desea ordenar?
1)Nombre
2)Apellido
3)Número de cuenta

Opción: 1

Iniciando la lectura de los bloques de archivos.

```

Figura 52: Ejecución de Polifase con las opciones mostradas en la captura.



```

Iniciando la lectura de los bloques de archivos.

Lectura de claves obtenidas de acuerdo al campo seleccionado:
[Alfonso, Juan, Juan, Santiago, Arturo, Carlos, Juan, Manuel, Juan, Alfredo, Ernesto, Miriam, Joel, Daniela,
Benjamin, Alberto, Jorge, Nicolas, Adrian, Karen, Salvador, Charly, Erika, David, Carlos, Hannah, Jose, U
dra, Juan, Jorge, Emiliano, Antonia, Alejandro, Luis, Eduardo, Daniela, Albanya, Veronica, Jose, Grecia, Br
jandro, Agustin, Jared, Alejandro, Alejandro, Arturo, Vladimir, Mario, Jose, Enrique, Joaquin, Adrian, Dani
na, Dionisio, Vania, Adrian, Frida, Sergio, Chuck, Uriel, Karla, Raul]

Construyendo un nuevo archivo con las claves a ordenar...

Las claves han sido vaciadas a un archivo inicial F0. Iniciando Polifase...

Iniciando la lectura de los bloques de archivos.

Se ha cargado un bloque.
Estado actual :[Alfonso, Juan, Juan, Santiago]
Bloques procesados: 1
Bloque almacenado en F1: Alfonso,Juan,Juan,Santiago,

Se ha cargado un bloque.
Estado actual :[Arturo, Carlos, Juan, Manuel]
Bloques procesados: 2
Bloque almacenado en F2: Arturo,Carlos,Juan,Manuel,

Se ha cargado un bloque.
Estado actual :[Juan, Alfredo, Ernesto, Miriam]
Bloques procesados: 3
Bloque almacenado en F1: Alfredo,Ernesto,Juan,Miriam,

```

Figura 53: Generación de los bloques de tamaño fijo.

```

Iniciando el proceso de intercalación...

[Bloque 1: Alfonso,Juan,Juan,Santiago, Alfredo,Ernesto,Juan,Miriam, Luis,Sofia,Sofia,Xiadani, Alberto,Benjamin,Jorge,Nicolas,
Antonia,Emiliano,Jorge,Juan, Albanya,Grecia,Jose,Veronica, Aldo,Jorge,Monica,Rodrigo, Alejandro,Alejandro,Arturo,Vladimir,
Victoria, Chuck,Karla,Raul,Uriel,
Bloque 2: Arturo,Carlos,Juan,Manuel, Arturo,Daniela,Joel,Sonia, Axel,Daniel,Gabriel,Paula, Adrian,Charly,Karen,Salvador, Ana
Alejandro,Daniela,Eduardo,Luis, Aurelio,Brigitte,Jacob,Javier, Agustin,Alejandro,Jared,Yvette, Enrique,Joaquin,Jose,Mario, Cri
,
Alerta: Se ha detectado que archivo F1 contiene más bloques que el archivo F2.
Se ha añadido un bloque adicional:
Bloque 2: Arturo,Carlos,Juan,Manuel, Arturo,Daniela,Joel,Sonia, Axel,Daniel,Gabriel,Paula, Adrian,Charly,Karen,Salvador, Ana
Alejandro,Daniela,Eduardo,Luis, Aurelio,Brigitte,Jacob,Javier, Agustin,Alejandro,Jared,Yvette, Enrique,Joaquin,Jose,Mario, Cri
, !

Bloque fusionado y añadido a F0: Alfonso,Arturo,Carlos,Juan,Juan,Juan,Manuel,Santiago,
Bloque fusionado y añadido a F3: Alfredo,Arturo,Daniela,Ernesto,Joel,Juan,Miriam,Sonia,

Bloque fusionado y añadido a F0: Axel,Daniel,Gabriel,Luis,Paula,Sofia,Sofia,Xiadani,
Bloque fusionado y añadido a F3: Adrian,Alberto,Benjamin,Charly,Jorge,Karen,Nicolas,Salvador,

Bloque fusionado y añadido a F0: Ana,Carlos,David,Erika,Hannah,Jordi,Jose,Rosario,
Bloque fusionado y añadido a F3: Andres,Brenda,Diego,Edgar,Isabel,Jose,Rosa,Sandra,

Bloque fusionado y añadido a F0: Alejandro,Antonia,Daniela,Eduardo,Emiliano,Jorge,Juan,Luis,
Bloque fusionado y añadido a F3: Albanya,Aurelio,Brigitte,Grecia,Jacob,Javier,Jose,Veronica,

Bloque fusionado y añadido a F0: Agustin,Aldo,Alejandro,Jared,Jorge,Monica,Rodrigo,Yvette,
Bloque fusionado y añadido a F3: Alejandro,Alejandro,Arturo,Enrique,Joaquin,Jose,Mario,Vladimir,

Bloque fusionado y añadido a F0: Adrian,Cristian,Daniel,Enrique,Nancy,Rodrigo,Saul,Valeria,
Bloque fusionado y añadido a F3: Adrian,Celia,Dionisio,Frida,Gina,Sergio,Vania,Victoria,

Bloque fusionado y añadido a F0: Chuck,Karla,Raul,Uriel,

Los archivos F0 y F3 contienen los siguientes bloques:

```

Figura 54: *Proceso de intercalado realizado con los bloques recuperados.*

```

Los archivos F1 y F2 contienen los siguientes bloques:
[Bloque 1: Adrian,Adrian,Adrian,Agustin,Albanya,Alberto,Aldo,Alejandro,Alejandro,Alejandro,Alejandro,Alfonso,Alfredo,Ana,Andres,Antonia,Arturo,
Arturo,Arturo,Aurelio,Axel,Benjamin,Brenda,Brigitte,Carlos,Carlos,Celia,Charly,Chuck,Cristian,Daniel,Daniel,Daniela,Daniela,David,Diego,Dionis
io,Edgar,Eduardo,Emiliano,Enrique,Enrique,Erika,Ernesto,Frida,Gabriel,Gina,Grecia,Hannah,Isabel,Jacob,Jared,Javier,Joaquin,Joel,Jordi,Jorge,Jo
rge,Jorge,Jose,Jose,Jose,Jose,Juan,Juan,Juan,Juan,Juan,Juan,Karen,Karla,Luis,Luis,Manuel,Mario,Miriam,Monica,Nancy,Nicolas,Paula,Raul,Rodrigo,Rodri
go,Rosa,Rosario,Salvador,Sandra,Santiago,Saul,Sergio,Sofia,Sofia,Sonia,Uriel,Valeria,Vania,Veronica,Victoria,Vladimir,Xiadani,Yvette,
Bloque 2:

Bloque fusionado y añadido a F0: Adrian,Adrian,Adrian,Agustin,Albanya,Alberto,Aldo,Alejandro,Alejandro,Alejandro,Alejandro,Alfonso,Alfredo,Ana
Andres,Antonia,Arturo,Arturo,Arturo,Aurelio,Axel,Benjamin,Brenda,Brigitte,Carlos,Carlos,Celia,Charly,Chuck,Cristian,Daniel,Daniel,Daniela,Dan
iela,David,Diego,Dionisio,Edgar,Eduardo,Emiliano,Enrique,Enrique,Erika,Ernesto,Frida,Gabriel,Gina,Grecia,Hannah,Isabel,Jacob,Jared,Javier,Joaq
uin,Joel,Jordi,Jorge,Jorge,Jorge,Jose,Jose,Jose,Jose,Juan,Juan,Juan,Juan,Juan,Juan,Karen,Karla,Luis,Luis,Manuel,Mario,Miriam,Monica,Nancy,Nicolas,P
aula,Raul,Rodrigo,Rodrigo,Rosa,Rosario,Salvador,Sandra,Santiago,Saul,Sergio,Sofia,Sofia,Sonia,Uriel,Valeria,Vania,Veronica,Victoria,Vladimir,Xi
adani,Yvette,
;Felicitades! Los datos han sido ordenados.

Saliendo de la opción...

```

Figura 55: *Resultado obtenido.*

8. Conclusiones

■ López Lara Marco Antonio

Debido a que algunas partes esenciales de nuestro código no funcionaban de la manera correcta se



hizo necesaria una investigación de estructura de datos en específico del ordenamiento de datos, en este proyecto nuestra prioridad era el manejo de datos o archivos .TXT en java debido a nuestro equipo tenía experiencia con el uso de dichos archivos pero muy poca en el lenguaje de programación JAVA.

El uso de archivos es muy relevante en el proyecto debido a su importancia ya que en ellos se guardarán nuestras claves, y en algunas cuestiones será necesario de otros archivos para almacenar nuestras claves que serán ordenadas y guardadas o sobrescribir en el archivo.

En el proyecto vi el uso de lo anteriormente aprendido en la clase, también fue de mucha ayuda para poder entender mejor la polifase, mezcla natural, radix externo ya que los tres yo no los había entendido hasta que con la ayuda de del proyecto y de mis compañeros lo entendí.

Me llevo nuevos conocimientos de algunas entre ellas es el uso de Try para las excepciones de nuestras claves y siendo que si existe una excepción no cause que nuestro programa no siga avanzando.

■ **Olivera Martínez Jenyffer Brigitte**

A medida que se avanzaba en el proyecto era necesario ir investigando distintos aspectos en relación con el manejo de archivos en java. Aspecto completamente nuevo en el lenguaje que se está aprendiendo y el paradigma orientado a objetos. Si bien este aspecto representó un primer acercamiento al manejo de archivos en Java, crea una base para el desarrollo de conocimientos a futuro, pues al momento en el que en el curso deban tratarse archivos no se llegará con la mente en blanco.

El manejo de archivos es relevante no solo para la cuestión de ordenamiento externo, muchas veces al momento de resolver problemas en programación los archivos serán la primera fuente de información para la solución de algún problema específico, además de optimizar la gestión de información, aspecto que hace a los archivos una herramienta fundamental para cualquier programador.

Al igual que con el manejo de archivos, se pusieron en práctica muchos de los conocimientos adquiridos a partir de los temas vistos en clase y se profundizó en ellos, lo que contribuye a aclarar cualquier aspecto que no haya sido entendido.

Otro aspecto relevante en cuanto a los elementos utilizados en el proyecto son las excepciones, pues se debe ser consciente de que pueden llegar a aparecer una corrida de algún programa y es necesario saber las acciones que deben tomarse ante ellas.

De hicieron notar las diferencias entre los algoritmos que se emplearon, cada uno tiene similitudes y diferencias, por ejemplo, el hecho de que todos estos algoritmos están basados en la estrategia “divide y vencerás”, no significa que hagan la división de la información de la misma manera, pues cada algoritmo emplea sus propios métodos. Así mismo los criterios que utiliza cada algoritmo son distintos: Polifase establece bloques de tamaño fijo y emplea algún algoritmo de ordenamiento interno, Mezcla Equilibrada establece bloques ordenados y Radix externo emplea un criterio de ordenamiento distinto, pues toma en cuenta dígitos significativos, cosa que no para en ninguno de los otros dos



algoritmos.

Sin lugar a duda este proyecto contribuye de manera significativa a el desarrollo de nuevos conocimientos e implementaciones de los algoritmos y antecedentes. En cuanto al trabajo en equipo, el proyecto hace notar las diferencias en cuanto al ritmo de trabajo de cada individuo y resalta la importancia de las contribuciones de equipo, pues pese a las diferencias que puedan existir, el trabajo en equipo reúne las habilidades de cada integrante para un fin común y de esta manera desarrollar más habilidades y apreciar las aportaciones que cada integrante puede hacer.

Para concluir en mi experiencia los objetivos del proyecto fueron los correctos y fueron llevados a cabo ya que pienso que nuestro equipo se lleva bastantes conocimientos extras así como el uso de latex y algunas cuestiones de java, para mi el proyecto es de gran ayuda en mi formación como ingeniero ya que con ello puedo ver las utilidades de los conocimientos que voy adquiriendo y que verdaderamente son usados y utiles.

■ Téllez González Jorge Luis

El primer proyecto realizado para el curso de EDA II representa una culminación del desarrollo de habilidades que cada uno de los integrantes ha tenido que desarrollar a lo largo de las prácticas de laboratorio, debido a que muchos de los conceptos usados en las implementaciones programadas requieren el manejo, por citar algunos ejemplos, de clases como *ArrayList*. Por otra parte, el desarrollo de habilidades de manejo de archivos externos fue una componente interesante de estudiar y aplicar; ya que en mi experiencia previa de Java únicamente llegué a leer contenido de un archivo. En cambio, en el proyecto me vi forzado a llevar más allá esta habilidad y aprender a hacer un manejo eficiente de los procesos de entrada/salida sobre archivos.

Entre los puntos más relevantes que puedo comentar de la experiencia obtenida se encuentran los siguientes:

- La primer gran dificultad fue, evidentemente, aprender a manejar los archivos *File*, así como todas las clases que operan sobre ellos. En un inicio fue una experiencia tremendamente frustrante y sin demasiados avances; especialmente al experimentar con la escritura sobre archivos. Me tomó una cantidad considerable de tiempo leer la documentación proporcionada por Oracle y experimentar en **NetBeans**. Por supuesto, el esfuerzo rindió sus frutos y el manejo de los procesos de lectura/escritura se volvió una tarea mucho más sencilla e intuitiva.
- El primer método programado fue la simulación de Radix-Externo. Sin embargo, fue escrito considerando únicamente claves numéricas simples. Idear e implementar la solución a la lectura de las superclaves con 3 campos fue un obstáculo que no pudimos sortear efectivamente hasta que, experimentando y estudiando los ejemplos de la clase teórica escribí el método **scanSuperKeys**, el cual es vital en todas las implementaciones escritas y representa, por su importancia en la lectura inicial, uno de los métodos más importantes del proyecto.



- El método de ordenamiento que, considero, fue el más complicado de implementar fue la Mezcla Natural. El enfoque de particiones máximas requirió un mayor tiempo de abstracción a diferencia de Polifase, así como un control más elevado de los bloques escritos en cada archivo: por ello, se implementaron arreglos de control para tener un control efectivo de los datos entrantes y salientes dentro de los archivos utilizados por el programa.

La programación de cada uno de los métodos, sin duda, fue una experiencia muy retadora y frustrante en ocasiones. Después de sortear una gran cantidad de excepciones *nullPointer* o relacionadas con los índices de los arreglos, el resultado final en cada implementación refleja un proceso de *sudor* y *sangre*: una satisfacción que solo una persona del área puede comprender.

La experiencia final es una gran cantidad de conocimiento adquirido. Por supuesto, el trabajo en equipo es importante y la conclusión del proyecto no hubiese sido posible con una única persona. Esto es un importante recordatorio de la importancia del trabajo en equipo en la Ingeniería en cualquiera de sus campos de acción. Tengo la certeza que toda la experiencia obtenida será invaluable durante el desarrollo del resto del curso; pese a que la situación actual en México ha afectado la dinámica de toda la **UNAM**. Sin embargo, no nos detenemos y seguimos trabajando pese a todas las adversidades a las que nos enfrentemos.

Referencias

- [1] Radix-Sort. Recuperado de: <https://www.geeksforgeeks.org/radix-sort/>. Fecha de consulta: 9/03/2020.
- [2] Aguilar, L. J. and Zahonero, I. (2008). *Estructuras de datos en Java*. Mc Graw Hill, 1st edition.
- [3] Cairo, O. and Guardati, S. (1993). *Estructuras de Datos*. Mc Graw Hill, 3rd edition.
- [4] Ceballos, F. J. (2010). *Java 2. Curso de programación*. Madrid: Ra-Ma S.A., 4th edition.
- [5] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). *Introduction to Algorithms*. Massachusetts Institute of Technology, 3rd edition.
- [6] McConnell, J. (2001). *Analysis of Algorithms: An Active Learning Approach*. Jones and Bartlett Publishers., 1st edition.
- [7] Skiena, S. (2008). *The Algorithm Design Manual*. Springer, 2nd edition.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©