Universidad Nacional Autonoma de Mexico

Facultad de Ingeniería

Estructuras de Datos y Algoritmos II

Análisis del Proyecto #1
Ordenamiento Externo

Profesor:
Tista García Edgar

Equipo 2

Integrantes:
Andrade López Lesly Beatriz
Gómez Luna Alejandro
Leonardo Said Cruz Rangel

6 de Octubre del 2019

Análisis

El proyecto se realizó en el lenguaje de programación Java, debido a que tiene bastantes aspectos que nos facilitaron la realización del proyecto, como lo es el manejo de archivos, el manejo de excepciones, el uso de la herencia para la reutilización de cóodigo, entre otras cuestiones.
En cuanto a la parte teórica del proyecto, relacionado con lo visto en clase, se utilizaron los conceptos respecto a los métodos de ordenamiento interno y externo.
En el ordenamiento interno para polifase, se implementó el algoritmo de Quicksort visto en la práctica de laboratorio. Se determinó utilizarlo ya que nos ofrece un buen tiempo de complejidad para entradas de listas con bastantes elementos.
De igual forma, se utilizó parte de Mergesort, en la parte donde se combinan los bloques de llaves leídas. Esto se tratará con mayor detenimiento más adelante en el análisis.
En cuanto a los algoritmos de ordenamiento externo, se implementaron polifase, mezcla equilibrada o natural y mezcla por distribución.
Para cada ordenamiento externo se utilizó una clase distinta.
Ahora se tratará la manera en como se implementó cada ordenamiento externo.

Implementación de polifase.

Para llevar a acabo este ordenamiento, es necesario usar 3 archivos auxiliares, además del archivo con las claves a ordenar. Todas las operaciones a realizar para el ordenamiento de los archivos es implementado en una sola clase llamada Polifase. Dicha clase tiene como atributos:

- **File to:** Almacena el archivo original.
- **Lista de números float F1:** Se considera para llevar a cabo el conteo de la longitud de los bloques y para facilitar el ordenamiento de los mismos.
- **Lista de números float F2:** Se implementa a la par de la lista F1 y tiene la misma función que la lista anterior.
- Variable entera llamada clave donde su valor es 50: Este valor indica el tamaño de cada uno de los bloques que se van a generar.
- **Variable entera llamado nCl con valor inicial 1:** Como el código va a ser implementado en archivos de diferentes tamaños, es necesario que cada vez que se concatenan algunos bloques, ya no se tenga que considerar un tamaño inicial de 50 sino que este comenzara a incrementar el doble de lo que valía anteriormente, por ello es que se utiliza esta variable.
- **Variable booleana llamada formord:** Nos indicará si el usuario ha elegido una de las formas de ordenar los valores, así el programa tendrá certeza de cuál método debe usar.

Inicialmente se tiene un constructor de la clase, en donde recibe como parámetros el nombre del archivo a ordenar, más la variable booleana, donde se determina la forma de ordenar el archivo.

Teniendo ya estos atributos y considerando que el algoritmo está dividido en 2 fases, de igual manera se implementan 2 métodos fundamentales, dados que ahí se engloban otros métodos y operaciones.

FASE 1:

- **Método partition():** Este método ha sido usado en alguna de las prácticas de laboratorio para implementar el algoritmo de ordenamiento interno QuickSort, junto con el método sort().
  Quicksort toma un elemento llamado pivote perteneciente a la lista de elementos para que con el, se comparen los valores que se encuentran tanto a su derecha, como a su izquierda. Los valores que se encuentran a su izquierda deben ser menores o iguales que el valor del pivote, mientras que los de la derecha deben ser mayores a dicho valor (suele tomarse la opción inversa a lo mencionado con anterioridad si es que se trata de ordenamiento de elementos de forma descendente). Como ahora se ha tomado un pivote, por lo regular queda dividida la lista original en 2 partes, a cada parte de ella se le va a aplicar nuevamente el método de ordenamiento Quicksort, hasta que la lista de elementos quede totalmente ordenada.
- **Método sort():** Este método, como ya ha sido mencionado, trabaja junto al método partition(), y su función es realizar Quicksort en las sublistas generadas, mediante la recursividad.
- **Método printArray():** Este método imprime en pantalla las iteraciones realizadas, mediante las sublistas generadas.
- **Método guardar():** Inicialmente recibe como parámetros el archivo en el cual se guardará el bloque y la lista donde se encuentran contenidos los valores del bloque, como ya han sido previamente ordenados, solo los guardará.
  Para poder guardar los datos en el archivo, se implementa FileWriter y BufferedWriter, el cual ayuda a guardar los datos necesarios en cada archivo. Se emplea un ciclo for, para recorrer cada valor perteneciente a la lista. Cada valor es convertido a String, para escribirlo en el archivo correspondiente agregando una coma después de cada elemento. Al momento de terminar el ciclo for se limpia la lista de valores y se cierra el archivo (con la finalidad de no perder información o causar errores en el código).Las acciones que necesitan implementación de un archivo, deben ir contenidas en un try-catch, para tratar alguna excepción que se pueda tener al interactuar con archivos.
- **Método Fase1():** En esta primera fase, se recibe el archivo original para ser distribuido mediante bloques en 2 archivos auxiliares, por ello es que se necesitan los parámetros de los 2 archivos auxiliares correspondientes.
  En un bloque try-catch se encuentran contenidas las siguientes acciones: Se hace uso de un Scanner para poder leer los datos almacenados en el archivo, por ello en el parámetro del objeto Scanner debe de estar escrito como parámetro el nombre del archivo el cual va a ser leído.

Posteriormente, se hace uso del método useDelimiter() perteneciente a la clase Scanner, el cual nos permite indicar la manera en cómo leer cada uno de los valores, y, para este caso, se ha estipulado que cada valor debe estar separado por una coma (“,”). Una vez definido este limitante, se usa un ciclo while, donde se evaluará que exista un dato siguiente en el archivo. Si esto se cumple, se procede a guardar el dato leído del archivo en una variable de tipo flotante, convirtiendo el valor a flotante, pues se lee como si fuera una cadena.

Después, se evalúa que el tamaño de la primera lista no sea igual que el tamaño del bloque, si esto se cumple, se guardará el valor en dicha lista. En caso contrario, se utiliza otra vez la misma condición anterior, pero ahora tomando en cuenta el tamaño de la segunda lista. Si esta lista tiene el mismo tamaño del bloque, finaliza la condición, en caso contrario, será almacenado el valor que ha sido leído. Es necesario tener otra condición tomando en cuenta si el tamaño de ambas listas es igual al tamaño del bloque,en caso de que la condición se cumpla, se utilizan los métodos sort(), printArray(), guardar() a fin de que sean ordenadas ambas listas, las cuales se imprimen y finalmente se guardan en su respectivo archivo.

Se toma en cuenta que al momento de que ya no haya un elemento siguiente en la lista, ya no se cumplira el ciclo while, es por ello que se verifica si quedaron guardado datos en la lista F1, si esto es verdadero, se usarán los métodos sort(), printArray(), guardar() .Esta consideración es tomada debido a que puede existir el caso de que las 2 listas no se llenen por completo debido al tamaño del bloque y se considera evaluar solo la primera lista, porque si la primera lista tiene algún valor guardado, hay una posibilidad de que la segunda tenga algún elemento en ella.

FASE 2:

- **Método acomodarnum():** Es utilizado para corroborar en qué posición de la lista se debe acomodar el valor previamente leído. Este se implementa sólo en la fase 2. La razón por la cual no es utilizado nuevamente el método sort() y se decide crear este nuevo método, es debido a que los bloques pequeños a concatenar para originar uno más grande, ya se encuentran ordenados, solo se necesitan revisar los elementos en cada uno para determinar en qué posición deben irse insertando.
- **Método copia():** Al momento de terminar el acomodo de bloques en la fase 2, se sabe que aún no se tienen todos los bloques almacenados en un solo archivo, por lo que se debe volver a repetir la fase 2. Para ello, toda la información que se encuentra en el archivo auxiliar 3 y original debe ser copiada a los archivos auxiliares 2 y 1 respectivamente. Primero es necesario borrar la información que se encuentre en los archivos 1 y 2. Posteriormente, se copia la información de un archivo a otro, creando un Filein para el archivo que proporciona su información y un Fileout para el archivo que va a recibir información. Como en la fase 2, se vuelve a escribir en el archivo 3 y auxiliar respectivamente.

Es importante remarcar que estos archivos deben de limpiarse también para no generan algún problema en el código.

- **Método printArray():** Aunque ya ha sido explicado el método en la parte de Fase 1, en esta fase es de suma utilidad para mostrar en pantalla los bloques que se van a ir generando.
- **Método guardar():** Método que también es necesario en las 2 Fases, debido a que almacena los bloques previamente ordenados en el archivo correspondiente.
- **Método fase 2():** Necesita como parámetros los archivos auxiliares 1,2 y 3. El archivo original no es necesario pasarlo como parámetro, pues se debe recordar que es un atributo de la clase.

  Se hace uso de la clase Scanner para leer los elementos del archivo.

  Las acciones de este método se realizarán mientras el primer archivo o el segundo tengan un elemento siguiente, si esto se cumple, en el interior de este ciclo se encuentra una condición donde se cuestiona si el Scanner encargado de leer los datos del archivo 2 contenga un elemento siguiente, si es verdadero, se seguirán trabajando con los datos proporcionado, en caso contrario, sólo se leerán datos del primer archivo y la variable ord tendrá un valor de 3.. Esta acción se tomó en cuenta porque puede llegar el caso de que un bloque no se encuentre lleno hasta el límite establecido.Se debe considerar que la evaluación del ciclo while se cumple siempre que se encuentre un elemento siguiente en alguno de los 2 archivos auxiliares, a consecuencia de ello, se puede tener el caso de que un archivo sea mayor que el otro, por consiguiente, se seguirán leyendo datos de los 2 archivos, sin importar que alguno de ellos se encuentre vacío. Para evitar lo antes mencionado, a la variable ord se le da el valor de 3, para que al momento de usar el método acomodarnum() no se escriba ningún valor a la lista, en este caso, la lista que pertenece al archivo con el menor número de valores en su interior.

  Los datos comenzarán a almacenarse primero en el archivo original, y luego en el archivo 3. Si estos archivos se encuentran llenos, se procederá a imprimir en pantalla lo que se tiene guardado en las listas, para después guardar los datos en el archivo correspondiente. Cuando ya no se cumpla el ciclo de repetición, puede llegar a suceder que alguna lista contenga valores de un bloque (por lo regular siempre suele ser la lista perteneciente al archivo 1),entonces se presenta el cuestionamiento de si hay valores en dicha lista y si esto es verdadero, imprime en pantalla los valores que ahí se encuentren, para después almacenarlos en el archivo que corresponde. Para concluir, se copian los datos del archivo original y del auxiliar 3 en los archivos auxiliares 1 y 2 por si se necesita volver a repetir este método.

  Un dato importante de este método es que devuelve un valor entero. Devuelve un -1 si es que uno de los archivos se encuentra vacío. Cuando se cumple esto, se sabe que todos los datos se encuentran en un solo archivo y por tanto ya se han concatenado todos los bloques. Devuelve un 0 si es que aún se encuentran bloques en los 2 archivos.

- **Método OrdP():** Puede considerarse el método principal de esta clase, pues se encarga de englobar todos los métodos necesarios para llevar a cabo el ordenamiento Polifase. En este método son creados los 3 archivos auxiliares con los que se estarán trabajando al momento de ejecutar el código. El método fase1() sólo se realiza una sola vez, mientras que el método fase2() es el que debe repetirse hasta que el archivo se encuentre totalmente ordenado. Para llevar a cabo esta acción, se utilizó un ciclo do-while que se repita hasta que el método retorna el valor -1, entonces terminará el ciclo, teniendo la certeza de que el archivo se ha ordenado.

Implementación de mezcla natural.

El programa se divide en 4 métodos: partición, mergeAscendente, mergeDescendente y ordenarPorMezclaNatural. El método partición recibe como parámetro un booleano, éste nos indicará si vamos a ordenar de forma descendente o ascendente. El método pretende dividir a un archivo en dos archivos auxiliares que contendrán los bloques que ya se encuentran ordenados y regresa un valor booleano el cual indicará si el archivo auxiliar 2 tiene elementos. La forma en que se hace lo anterior es muy sencilla, primeramente declaramos un objeto de la clase Scanner, éste lo usaremos para leer el archivo a ordenar, sin embargo, para poder leer los números del archivo tenemos que definir un delimitador que en nuestro caso será la coma. Seguido a esto declaramos los BufferedWriter de los dos archivos auxiliares que usaremos. A continuación leeremos el primer número del archivo a ordenar y lo escribiremos en el archivo auxiliar 1, inicializamos una variable booleana(band) en true y en una variable auxiliar(aux) de tipo float guardamos el primer número que escribimos. En caso de que el orden sea ascendente entonces mientras el archivo a ordenar tenga elementos recuperaremos su siguiente elemento en una variable float(r). Si r es mayor o igual que aux procederemos a actualizar el valor de aux a r y si band es true escribimos en auxiliar 1 de lo contrario escribiremos en el auxiliar 2; si r no es mayor o igual que aux entonces actualizamos el valor de aux a r y si band es true escribimos en el auxiliar 2 y cambiaremos el valor de band a falso, de ésta forma cuando el ciclo while vuelva a empezar como band es igual a falso entonces escribiremos en el auxiliar 2 en caso de que r sea mayor o igual que aux; si band es falso entonces escribiremos en el auxiliar 1 y cambiaremos el valor de band a verdadero. En caso de que el orden sea descendente se hace exactamente lo mismo, únicamente cambiamos la condición de que r sea mayor o igual que aux a que r sea menor o igual que aux. Para finalizar cerramos el flujo de datos entre nuestro programa y los archivos y por medio de otro Scanner para lectura del auxiliar 2 determinamos si tiene elementos, en caso de que no tenga retornamos falso en caso contrario retornamos verdadero.
El método mergeAscendente no recibe parámetros ni regresa algún valor, éste únicamente hará la mezcla de los bloques ordenados de los archivos auxiliares y el resultado lo escribirá en el archivo auxiliar a ordenar, es decir, el original.

Para lograr lo anterior, creamos los Scanner para lectura del auxiliar 1 y 2, borramos el contenido del archivo original y declaramos un bufferedWriter para poder escribir en él. Si los dos archivos tienen elementos entonces leemos el primero de cada uno y encontramos el de menor valor, asignamos su valor a una variable aux, lo escribimos en el archivo original a band le asignamos true y la variable en donde almacenamos ese valor (r1 o r2) la igualamos a cero. Mientras alguno de los auxiliares tenga elementos entonces si band es true y auxiliar 1 tiene elementos leeremos el siguiente número y lo guardaremos en r1; si aux 1 ya no tiene elementos entonces verificamos que r1 o r2 no tengan elementos que aún no han sido escritos, es decir, que su valor sea diferente de cero, en caso de que sí escribimos el elemento y posterior a esto escribimos todos los números que haya en al archivo auxiliar 2. Si band es false entonces hacemos lo mismo pero con el auxiliar 2.
Si r1 es menor que r2 entonces entramos en un if-else anidado en donde evaluamos que r1 sea mayor o igual que aux, ésto indicará que efectivamente estamos colocando los números del menor al mayor, actualizamos el valor de aux a r1, escribimos el número en el original, a band le asignamos true para leer otro número del auxiliar 1 y a r1 le asignamos 0; en caso de que no se cumpla la condición entonces evaluamos que r2 sea mayor o igual que aux para poder escribirlo en el original, actualizar el valor de aux a r2, band ponerlo en false para leer otro número del auxiliar 2 y a r2 le asignamos cero; en caso de que las dos condiciones anteriores no se cumplan entonces actualizamos el valor de aux a r1 lo escribimos en el original, a band le asignamos true y r1 le asignamos cero. En caso de que r1 no sea menor que r2 entonces entramos a otro if-else anidado con las mismas condiciones que el anterior pero cambiando los r1 por r2. Al final verificamos que no hayan quedado números en r1 y r2 sin escribir, es decir, que sus valores sean diferentes de cero, en caso de que si entonces los esribímos hasta el final. Al final del método borramos el contenido de los auxiliares 1 y 2.
En pocas palabras lo que se hace es leer dos números de los auxiliares, tomamos el menor de ellos y lo escribimos en el archivo original, leemos otro número del auxiliar del que resultó el menor y lo comparamos con el número que leímos con anterioridad del auxiliar 2, nuevamente escribimos el de menor valor y así sucesivamente hasta acabar con un bloque y continuamos con los demás. El método mergeDescendente hace casi lo mismo que mergeAscendente, únicamente cambiamos las condiciones que hay entre r1, r2 y aux para que consgia mezclar los números y ordenarlos de forma descendente.
En el método ordenarPorMezclaNatural repetimos particion y el merge para el caso de ordenamiento que nos encontremos(ascendente o descente) hasta que el archivo auxiliar 2 ya no tenga elementos, esto indicará que el archivo original ya está ordenado. Para hacer esto usamos un ciclo while que se repetirá hasta que particion arroje false, mientras sea true haremos el merge correspondiente.

Implementación de mezcla por distribución.

Se comenzó por importar las clases necesarias para poder escribir y leer archivos, así como otras clases necesarias para el manejo de excepciones.
Ya importado esto, se procedió a crear la clase MezclaPorDistribucion, la cual contendrá todo lo necesario para realizar el ordenamiento.
Cuenta con un atributo File, el cual se inicializa en el constructor y almacena el archivo que se ordenará.
En el constructor de la clase, además de inicializar el único miembro, también se verifica que no existan archivos auxiliares residuales de alguna ejecución pasada del programa, por lo que, si existen, se tratan de eliminar, o en su defecto, se vacían.
El ordenamiento se lleva a cabo en el método ordenarPorDistribucion, el cual solo toma como parámetro un boolean, el cual indica si se quiere realizar un ordenamiento ascendente o descendente.
Dentro de este método se comienza por crear un Scanner para leer el contenido del archivo que ordenaremos, indicándole que debe generar cada token siempre que encuentre una coma. Ya con este delimitante, se utiliza el método hasNext() junto con el método next() para ir leyendo los elementos dentro del archivo y transformarlos a números de punto flotante. En toda esta pasada por los elementos del archivo se almacenará el número más grande, al cual se le obtendrá la cantidad de dígitos que tiene después de ser multiplicado por cien, para saber la cantidad de iteraciones que realizaremos para poder ordenar todos los números. Se multiplica por cien este número ya que, dentro del proceso de ordenamiento, todos los números se multiplican por cien para no lidiar con la parte decimal temporalmente.
Ya que conocemos la cantidad de veces que repetiremos el proceso, utilizamos un for externo y un ciclo while interno en primera instancia. El ciclo for externo llevará el control de la cantidad de dígitos del número más grande obtenido anteriormente, mientras que el while interno llevará el control de la cantidad de números en el archivo a ordenar.
Una vez más, para leer el contenido del archivo a ordenar, se utiliza un Scanner junto con sus métodos hasNext() y next(). Siempre que se lee un número, se multiplica por cien y se redondea para manejarlo con entero, pues resulta más sencillo trabajar los números de esta forma. A este número entero, se le van a ir tomando los dígitos de derecha a izquierda. Este dígito se va a ir tomando en cuenta con respecto al for externo que va conforme a los dígitos del número más grande, por lo que, si ya no tiene más dígitos el número actual, se generará una excepción y esta se tratará diciendo que el dígito actual es cero.
Ya que se obtuvo el dígito actual del número actual, se procede a escribirlo en su archivo correspondiente, llamado como cola[valorDigitoActual].txt.
Después de haber escrito todos los números en su archivo correspondiente, se procederá a ir retirando los números de cada archivo generado.Es importante remarcar que los números almacenados en cada archivo auxiliar sobrescribirán a los números en el archivo original.

Para realizar esto, se utiliza un for que irá recorriendo cada archivo generado, y existirá un Scanner que leerá todos los elementos en este archivo, donde cada número volverá a ser almacenado en el archivo original. En este caso, si el ordenamiento es ascendente, se recorrerán los archivos desde aquel con nombre cola0.txt hasta cola9.txt. En caso contrario, los archivos se recorrerán desde cola9.txt hasta cola0.txt.
Estos archivos nombrados como cola son eliminados o en su defecto son vaciados para poder ser utilizados en futuras iteraciones, sin embargo, para que el usuario pueda visualizar lo que se está realizando no solamente en pantalla, se crean otros archivos auxiliares nombrados como colaAux[valorDigito].txt y es aquí donde se muestra el número de iteración y los números almacenados en el archivo para dicha iteración.
Todo el proceso se repite hasta que se agoten los dígitos del número más grande, dando como resultado que los números estén en su correspondiente orden especificado.
Cabe agregar que, para escribir en un archivo se utilizó un BufferedWriter junto con un FileWriter, en el modo donde no se sobrescriba nada. Por otro lado, para vaciar el contenido del archivo, dado que no existe un método en particular para hacerlo, solamente se creó un PrintWriter asociado al archivo que queramos vaciar, y, como lo abrimos en modo sobrescritura, el archivo se vacía.

Marco Teórico

Algoritmos de ordenamiento externo.

Al momento de realizar ordenamientos grandes de información, se ve afectada en la mayoría de las ocasiones la memoria principal, por lo que se ha dado la tarea de usar los dispositivos de almacenamiento secundario.
Al procedimiento de ordenar el contenido de información mediante un archivo almacenado en memoria secundaria se le conoce como externo. Esto implica leer el archivo, procesar la información encontrada para ordenarla y posteriormente reescribirlo nuevamente en dicho archivo.
Para llevar a cabo dicha acción, se ha encontrado un patrón en los métodos externos de ordenamiento, los pasos a realizar son:

1. Dividir el archivo, el cual debe ser ordenado en bloques de un tamaño considerable para poder ser procesado lo más eficiente posible.
2. Tratar a los bloques que se han generado como si se tratase de la única información a ordenar
3. Realizar una escritura nueva obtenida de la mezcla de los bloques que previamente han sido tratados.

Para conocer la eficiencia del ordenamiento externo, se debe tomar en cuenta que no depende solo de la forma que está planteado el algoritmo, sino que también del tiempo de acceso al dispositivo externo en donde se encuentra dicho archivo.
Los algoritmos de ordenamiento externo pueden variar en la forma en que dividen y mezclan cada uno de los bloques, la cantidad de uso de archivos auxiliares y el resultado final del ordenamiento.

Para este proyecto se llevaron a cabo los algoritmos de ordenamiento polifase, mezcla natural y mezcla por distribución, los cuales se describen a continuación:

- Polifase:
  Al momento de realizar la lectura del archivo original, se van generando los bloques de n claves, donde n es un valor arbitrario que es considerado a conveniencia del programador. Se implementan, además del archivo original, 3 archivos auxiliares, los cuales fomentan la rapidez del algoritmo en cada una de las iteraciones, a su vez, estas iteraciones se encuentran divididas en 2 fases, las cuales son:
  -Fase 1: Se lee una cantidad n de claves provenientes del archivo original hasta que no se encuentren valores por leer en el archivo. Después, utilizar un algoritmo interno para ordenarlas.
  Las n claves leídas generan un bloque que posteriormente sera guardardado en el primer archivo auxiliar F1 (se denota con dicho nombre para facilitar la explicación del algoritmo). Se repiten los pasos 1 a 2 donde el bloque resultante será guardado en un segundo archivo auxiliar F2 . Los siguientes bloques generados deberan intercalarse entre los archivo auxiliares F1 y F2.
  -Fase 2: Se debe intercalar el primer bloque del archivo auxiliar F1 con el primer bloque del archivo auxiliar F2, para escribir el resultado en el archivo original debe considerarse que al guardar ambos bloques, estos deben de seguir el ordenamiento. Posteriormente se lee el siguiente bloque de cada archivo, se intercalan y para finalizar se escriben en el archivo auxiliar F3. Los pasos 1 y 2 deben repetirse hasta que no haya mas claves por procesar. Si el total de claves no se encuentran contenidas en solo un archivo, se deben repetir las fases hasta que se genere un bloque del tamaño de la cantidad n inicial.

- Mezcla Equilibrada:
  También es conocida como mezcla natural.
  Contempla dos archivos auxiliares y se basa en:
  1. Realizar bloques tomando como consideración la secuencia de valores ordenados según el criterio elegido (principalmente se realiza de forma ascendente o descendente). Hasta que ya no se cumpla la condición, será formado el primer bloque.
  2. El primer bloque generado se guarda en el archivo auxiliar F1.
  3. Se vuelve a repetir la secuencia de valores ordenados, hasta que la condición no se cumpla y el bloque será guardado en el archivo auxiliar F2.
     Los tres pasos anteriores se repiten hasta que todos los elementos del archivo se encuentren distribuidos en los archivos auxiliares.
  4. Se toma el primer bloque del archivo auxiliar 1 con el primer bloque del archivo auxiliar 2 y se concatenan para que posteriormente sean guardados en el archivo original. Esto se repite hasta que ya no queden bloques que concatenar.

Los pasos mencionados con anterioridad son repetidos hasta que el archivo se ordene.

- Mezcla por distribución:
  Basado en el algoritmo de ordenamiento interno radix.
  Se crean archivos según los dígitos que se encuentren, las cuales pueden pertenecer desde el dígito 0 hasta el dígito 9.
  Se toma dígito por dígito, de derecha a izquierda. Según el dígito que se encuentre en esa posición, se insertará en su archivo correspondiente.
  Cuando el archivo original ya esté distribuida en todas las archivos correspondientes a cada dígito, procede a tomar de nuevo todos los valores y regresarlos al archivo original, tomando en cuenta el criterio que se ha decidido para ordenarlos (ascendente o descendente).
  Este proceso se repite hasta que ya se haya revisado el último dígito.

Manejo de archivos

Los archivos dentro de Java se pueden trabajar de distintas maneras, depende de la forma en como nosotros leamos los datos dentro del archivo, pues se pueden utilizar distintos flujos que trabajan la información del archivo de distintas maneras.
Para este caso, se utilizó la manera más fácil de utilizar archivos, la cual es mediante la clase File en java, además de utilizar la clase Scanner para poder leer el archivo y la clase BufferedWriter para escribir en el archivo.
La clase Scanner se ha utilizado para leer los datos ingresados por el usuario, el cual se puede observar como un flujo de información entre el teclado y el programa.
No obstante, la clase Scanner también se puede utilizar para los archivos, en donde ahora el flujo de información se da entre un archivo y el programa. En otras palabras, la clase Scanner nos permite trabajar con flujos de información. La manera en como Scanner trabaja un archivo es dividiendo los elementos que se encuentran en este en «tokens». Para saber la manera en como originar dichos tokens, se utiliza el método useDelimiter(), el cual por defecto toma los espacios en blanco como delimitante para ir creando los tokens, sin embargo, este método se puede modificar para que nosotros definamos el delimitante que queramos.
Una vez que ya se han creado los tokens, utilizaremos otros métodos dentro de Scanner, los cuales son hasNext() y algun metodo con la palabra next(). El método hasNext() nos permite ir verificando si todavía existe otro token por leer, es decir, si aún hay elementos en el archivo. Alguno de los métodos con next() nos va retornando el tipo de dato que estamos leyendo del archivo, ya sean enteros, flotantes, caracteres, etc.

Por otro lado, para la escritura de elementos en el archivo, se utilizó BufferedWriter, el cual va a ir escribiendo los elementos que nosotros le indiquemos en el archivo, a través del metodo write(). La ventaja de utilizar BufferedWriter y no alguna otra clase, es que tiene soporte para hilos y permite escribir bastantes cantidades de información sin generar errores. Esta clase no funciona de manera independiente, pues también utiliza una instancia de la clase FileWriter, la cual es la encargada de abrir el fllujo de información entre el archivo y el programa. Es importante remarcar que siempre que se abre un flujo de información, se tiene que cerrar, ya que se puede quedar en memoria el espacio reservado para dicho flujo, además de que puede generar errores.

Manejo de excepciones

En la programación se producen excpeciones con frecuencia, los cuales se pueden visualizar como un tipo de error pero que pueden ser «tratados» para evitar que el programa finalice, por lo que es necesario gestionarlos y tratarlos correctamente.

Java nos proporciona un mecanismo para manejar excepciones, éste consiste principalmente el uso de bloques try/catch. Dentro del bloque try colocamos las instrucciones que podrían llegar a generar alguna excepción. En caso de ocurrir la excpeción, entonces utilizaremos el bloque catch, el cual contendrá las instrucciones necesarias para poder tratarla.

El manejo de archivos puede generar excepciones tales como: no encontrar el archivo especificado durante el tiempo de ejecución, que se interrumpa la comunicación entre el archivo y nuestro programa,que intentemos acceder a un elemento que no exista dentro del archivo, entre otros. Para evitar que durante la ejecución del programa, el programa no se detenga o se vea severamente afectado, es necesario tratar las excepciones generadas.