Proyecto #1: Métodos de ordenamiento externo
Estructura de datos y algoritmos II

Cabello, Sofía López, Ricardo Núñez, Diego

15 de noviembre de 2020

1. Objetivo

Que el alumno implemente los algoritmosde ordenamiento externo,que conozca elementos para el manejo de archivos,aplique los conceptos generales deprogramacióny desarrolle sus habilidades de trabajo en equipo.

2. Introducción

Los algoritmos de ordenamiento nos permiten, como lo indica el nombre, ordenar o reorganizar un conjunto de datos. Esto es una operación fundamental no sólo en la computación sino en la vida cotidiana ya que facilita la búsqueda, agiliza procesos, permite manipular mejor la información, eliminar duplicados, entre otros.

Cuando se va a ordenar algo, es necesario considerar cuál es el algortimo de ordenamiento más adecuado, esto se puede determinar dependiendo del contexto, el tamaño del problema, el tiempo, verificació, pero nos vamos a enfocar en la memoria. A pesar de que cada vez los ordenadores cuentan con más memoria interna, existen situaciones en la que no se puede realizar la operación enteramente ahí, ya sea por la capacidad del dispositivo o por la cantidad de elementos a ordenar, en ese caso los datos deben de ser guardados en almacenamiento secundario como tarjetas o discos duros,y se deben de usar algoritos de ordenmiento externo.

El enfoque tradicional de estos algoritmos tiene dos partes importantes.

- En la primera se divide el problema por bloques que se ordenan con algún ordenamiento interno.

- En la segunda se mezclan los bloques hasta tener el archivo ordenado.

Para poder acceder a la mmoria externa se tuvieron que usar herramientas par el manejor de archivos. La entrada y salida de Java se organiza mediante objetos llamados Streams, que funcionan como intermediarios entre el programa y el origen o destino de la información generalizan a un fichero, es una secuencia de bytes en un dispositivo de almacenamiento. Los representa la clase java.io.

En el presente proyecto se busca aplicar los conocimientos aprendidos del tema para simular tres diferentes algortimos en lenguaje de programación Java en los que se utilizarán archivos para ordenar los elementos.

3. Mezcla Equilibrada

Para poder hablar de Mezcla Equilibrada es necesario hablar antes del algoritmo de ordenamiento externo llamado Mezcla Directa, ya que Mezcla Equilibrada es una mejora de este.

Mezcla Directa es un algoritmo muy fácil de comprender y por ende es uno de los utilizados en cuanto el ordenamiento externo. Su funcionamiento consiste principalmente en la realización sucesiva de una partición y una fusión que produce secuencias ordenadas de longitud cada vez mayor. Este proceso se repite hasta que la longitud de la secuencia para la partición sea: ((n+1)/2), que es cuando ya no es posible generar un bloque de mayor tamaño y todas las claves ya están ordenadas.

Como se menciona, Mezcla Equilibrada es una optimización de Mezcla Directa. La lógica que sigue es realizar las particiones tomando secuencias ordenadas de máxima longitud en lugar de secuencias de tamaño fijo. Después se mezclan estas secuencias ordenadas alteradamente sobre dos archivos auxiliares. Al hacer esto de forma repetida se podrá obtener al archivo original ordenado.

4. Polifase

Polifase es un algoritmo de ordenamiento externo que requiere de un ordenamiento interno y de 3 archivos auxiliares. El algoritmo básicamente consiste separar la colección inicial de datos en pequeños grupos ordenados, y después, a través de merge, unirlos en uno solo.

Es necesario un algoritmo de ordenamiento interno para ordenar los pequeños grupos de de datos. Los grupos de datos deben caber en memoria interna por esta razón, así que el tamaño de los grupos se suele elegir en función del tamaño del la memoria interna.

El proceso de generación de estos bloques es simple. Primero, se lee el grupo de datos del archivo de entrada(F0) y se ordenan los datos con un algoritmo de ordenamiento interno. Para este proyecto se eligió Insertion Sort, porque el tamaño de los bloques se fijó en 10 datos y porque es un algoritmo sencillo y eficiente para cantidades de datos pequeñas. Después, el grupo ordenado se escribe en un archivo auxiliar (F1) y se repite el proceso para el siguiente grupo, solo que en esta ocasión se escribe en otro archivo auxiliar (F2).

Después de tener los pequeños grupos ordenados, se hace merge entre pares de ellos para generar uno más grande. Este proceso se repite hasta que quede solo un grupo con todos los datos ordenados.

Para hacer el merge, los archivos F1 y F2 se convierten en los archivos de lectura. Luego, el primer bloque de F1 se intercala con el primer bloque de F2 y se escribe el bloque resultante en F0. Luego, se repite el proceso con el segundo bloque de cada archivo, pero esta vez el resultado se escribe en F3. Cuando se terminan los bloques de F1 y F2, F0 y F3 se vuelven los archivos de lectura y F1 y F2 en los de salida. Se repite el proceso hasta que todos los datos estén en un solo archivo.

5. Radix

Radix sort es un algoritmo de distribución que itera sobre claves en dígitos o carácteres específicos para luego separarlos en colas, de tal manera que se agrupan en ellas si tienen el mismo valor en la misma posición, por lo tanto requiere w número de iteraciones.

Hay dos variantes importantes para este algoritmo:

- LDS Dígito menos significativo.
- MDS Dígito más significativo.

La diferencia principal entre estos es por dónde comienza el ordenamiento, en LSD se consideran los carácteres de derecha a izquierda, en MSD se parte el archivo en R partes iguales de acuerdo con los primeros carácteres y se ordenan recursivamente las cadenas por los carácteres con los que empiezan.

En este proyecto se considero la variante LSD porque a diferencia de la otra, es estable, es decir, al ordenar elementos por un criterio se conserva el orden de entrada y si se quiere ordenar por otro criterio, se mantiene el ordenamiento anterior, y no produce tantos archivos pequeños por la recursividad.

5.1. Pseudocódigo

RadixSort(List A)

Inicio

w = número de caracteres de elementos de A

Para i = n hasta i = 0

Ordenar a A en pos i

Fin

6. Clases para el manejo de archivos de Java

6.1. Clase File

Todos los sistemas operativos requieren de pathnames para nombrar archivos y directorios. Esta clase es una vista abstracta e independiente del sistema operativo de los pathnames. Los pathnames consisten en un prefijo específico del sistema y una secuencia de cero o más directorios separados por un carácter también determinado por el sistema(En sistemas unix-like "/ " y en Windows "\"). Además, los pathnames pueden ser absolutos o relativos. Los absolutos son las rutas completas desde la raíz del sistema y los relativos son solo parte de ellas. File por defecto resuelve las rutas relativas al directorio en el que se invocó a la JVM.

Las instancias de esta clase no necesariamente refieren a archivos o directorios que realmente existen. Si representan a un archivo o directorio real, entonces ese archivo o directorio debe existir en el sistema de archivos. Además, las instancia de esta clase no son mutables, por lo que no pueden ser modificadas después de su creación.

El sistema de archivos puede administrar los permisos de archivos y directorios de los cuales esta clase es dependiente. Estos permisos pueden hace que algunos métodos fallen.

Algunos de los métodos que utilizamos en el proyecto son:

- createNewFile(): Crea un nuevo archivo si y solo si el archivo aun no existe.
- delate(): Elimina el archivo si es que existe. Si la instancia refiere a un directorio entonces este debe estar vacío.
- exists(): Verifica que el archivo denotado por el pathname exista.
- getAbsolutePath(): regresa una cadena con la ruta absoluta del archivo.
- isDirectory(): Verifica si la instancia refiere a un directorio.
- listFiles(): Devuelve un arreglo de objetos File de todos los archivos dentro de un directorio.
- mkdir(): Crea un directorio en la ruta de la instancia.

6.2. FileReader

Hereda a la clase InputStreamReader. La clase Java FileReader, java. io. FileReader hace posible leer el contenido de un archivo como una secuencia de caracteres. Funciona de manera muy similar a FileInputStream, excepto que FileInputStream lee bytes, mientras que FileReader lee caracteres. Puede crear objetos que pueden leer archivos de texto.

Constructores

- FileReader(File file) Crea un lector de archivo con una archivo del cuál leer.
- FileReader(FileDescriptor fd) Crea un lector de archivo con una FileDescriptor del cual leer.
- FileReader(String fileName) Crea un lector de archivo con un nombre de achiivo del cuál leer.

6.3. BufferedReader

Hereda a la clase InputStreamReader. Cuando se necesita leer la información almacenada en un fichero que contiene caracteres podems utilizar la clase BufferedReader, en la que se lee el texto almacenando los caracteres leídos. Proporciona un buffer de almacenamiento temporal. BufferedReader es síncrono, resulta útil usarse si estamos trabajando con varios subprocesos,tiene una memoria de búfer significativamente más grande que Scanner.

Constructores

- BufferedReader(Reader in) Crea el buffer de tamaño estándar.
- BufferedReader(Reader in,int sz) Crea el buffer de tamaño especificado.

Métodos

- void close() Cierra el stream.
- void mark(inr readLimit) Marca la posición actual del stream.
- int read() Lee un carácter.
- String readLine() Lee una linea de texto.
- void reset() Regresa stream a marca más reciente.
- long skip (long n) Se salta carácteres.

6.4. FileWriter

FileWriter es una clase de Java hecha para escribir en archivos de textos que fue añadida en la versión 1.1 del JDK. Los principales constructores de esta clase son:

- FileWriter(File file) : Construye un objeto de tipo FileWriter dado un objeto File.
- FileWriter(File file, boolean append) : Construye un objeto de tipo FileWriter dado un objeto File con un booleano que indica si añadir o sobreescribir los caracteres escritos.

- FileWriter(String fileName) : Construye un objeto de tipo FileWriter asociado con el String del nombre del archivo en donde se quiere escribir.

- FileWriter(String fileName, boolean append) : Construye un objeto de tipo FileWriter asociado con el String del nombre del archivo en donde se quiere escribir junto con un booleano que indica si añadir o sobreescribir los caracteres escritos.

En este proyecto se utilizo un método llamado write, el cual puede recibir una cadena de caracteres o un número entero que represente el código ASCII de un carácter que serán escritos en el archivo de texto correspondiente. Además de este método, se utilizó close para poder cerrar el stream y de esta forma asegurar que se guarde lo escrito.

Ejemplo de la utilización de FileWriter en el proyecto:

En este ejemplo se escribe una cadena de texto en un archivo indicado. Como se puede observar, se utiliza el constructor que recibe un objeto de tipo file y un booleano con valor true porque se quiere agregar la cadena y no sobrescribir el archivo.

7. Análisis del programa

7.1. Análisis de Alumno

La clase Alumno es la representación de un alumno cuyos atributos son su nombre, apellido y número de cuenta. Su constructor recibe una cadena de caracteres para el nombre y otra para el apellido. Además de un número entero para el número de cuenta. Por supuesto, dentro de la clase están definidos los setters y getters de cada atributo.

La principal razón de ser de esta clase es facilitar el ordenamiento en todos los algoritmos. Esto es debido a que los algoritmos son capaces de ordenar por cada atributo del alumno y es más fácil compararlos volviéndolos objetos.

7.2. Análisis de Dato

Esta clase es la parte fundamental del proyecto, ya que todos los programas utilizan los métodos que esta clase provee. Dato se encarga del manejo de los archivos de texto y sus funciones son capaces de leer y escribir datos específicos. Contiene 8 métodos, los cuales se describen a continuación:

- leerDato(int inicio, int fin, String archivoE)
  Este método lee varias líneas de un archivo de texto y obtiene un arreglo de objetos de tipo alumno. Recibe como parámetros dos números enteros, el primero representa el inicio de la línea a leer y el segundo la última línea a leer. Además, recibe una cadena de caracteres que indica el nombre del archivo de donde se obtendrán los datos. Primero se empieza por definir una lista de alumnos de la clase ArrayList, esta lista será llenada y posteriormente devuelta. Para hacer esto se crea un objeto de tipo File y su respectivo objeto BufferedReader y FileReader para leer datos del archivo. Después, con un ciclo for, se lleva al BufferedReader a la línea en donde se quiere iniciar a leer y con un ciclo while se van leyendo las líneas e instanciando objetos de tipo Alumno para agregarlos a la lista de alumnos. Para poder obtener el objeto Alumno de una cadena leída se utiliza el método obtenerDato, este método es analizado más adelante.

- escribirDatos(ArrayList<Alumno> alumnos, String archivoD)
  Este método escribe una lista de objetos de tipo alumno en un archivo de texto. Recibe como parámetro una lista de la clase ArrayList de tipo Alumno y el String del nombre del archivo en donde se escribirá la lista. Primero se define un objeto de tipo File y su respectivo FileWriter para poder escribir en él. Posteriormente, con ayuda de un ciclo for, se va recorriendo la lista y obteniendo cada atributo de sus elementos para concatenarlos y formar una línea de texto con el formato: “Nombre, Apellido, NoCuenta”.

- escribirDatos2(ArrayList¡Alumno¿alumnos, String archivoD)
  Este método es igual que el anterior, con la única diferencia de que se agrega un salto de línea al final del archivo.

- obtenerDato(String cadena)
  Este método recibe una cadena con formato ”Nombre, Apellido, NoCuentaz la convierte en un objeto de tipo alumno. Para hacer esto se ocupa una función llamada split que ayuda a dividir una cadena de texto dependiendo del token especificado. En este contexto en especifico se busca dividir la cadena por el token “, “, ya que con esto se puede obtener un arreglo en donde el primer elemento es el nombre, el segundo el apellido y el tercero el número de cuenta. Una vez obtenido estos datos sólo se tienen que enviar por medio del constructor de Alumno y por último devolverlo.

- addString(String archivoD, String string)
  Este método recibe una cadena y la escribe en el archivo especificado. Esto lo hace con ayuda de un objeto de tipo FileWriter instanciado para poder añadir texto. El principal algoritmo que usa este método es Polifase y lo utiliza para dividir las iteraciones en el archivo de texto.

- addHash(String archivoD)
  Este método escribe el símbolo "#.$^{\text{en}}$ un archivo especificado. Su funcionamiento es igual al método addString y el principal algoritmo que lo utiliza es el de Mezcla Equilibrada para dividir las iteraciones en el archivo de texto.

- escribirDato(Alumno alumno, String archivoD)
  Este método recibe un objeto de tipo alumno y lo escribe en un archivo de texto cuyo nombre está especificado en el parámetro de tipo String. Para que este método pueda funcionar se instancia un objeto de tipo File y su respectivo FileWriter para poder escribir en él. El objeto FileWriter está instanciado para que pueda añadir texto.

  Para transformar el Alumno en una cadena de texto se van obteniendo los atributos de ese y se van concatenando para que quede con el formato "Nombre, Apellido, NoCuenta".

- escribirDato2(Alumno alumno, String archivoD)
  Este método es igual que el anterior con el añadido de que se agrega un salto de línea al final de cada archivo.

7.3. Análisis de Mezcla Equilibrada

La implementación de Mezcla Equlibrada se divide en cuatro clases agrupadas en el paquete mezclaEquilibrada, y utiliza a las clases dato y alumno del paquete dato.

Los nombres de las clases son MezclaEquilibrada, MezclaNombre, MezclaApellido y MezclaCuenta.

7.3.1. Clase MezclaEquilibrada

Esta clase es la encargada de crear el directorio y los archivos auxiliares para el ordenamiento. Además, se encarga de ejecutar los métodos necesarios hasta que se haya podido cumplir con el objetivo de ordenar.

MezclaEquilibrada es una clase abstracta debido a que dos de sus métodos son igualmente abstractos. Se compone de sólo tres métodos, los cuales se describen a continuación:

- mezcla(String archivo)
  Este método es el principal de la clase. Aquí se instancian todos los archivos y se mandan a llamar a los métodos descritos más adelante.

  Primero se empieza por definir una variable booleana llamada isSorted, la cual ayuda al método a saber cuando el archivo ya está ordenado y se crea el directorio en donde se guardarán todos los archivos que genere el programa.

  Posteriormente se definen todos los objetos File correspondientes a todos los archivos que se utilizarán durante la ejecución del algoritmo. Además de esto se instancian los objetos FileReader y BufferedReader respectivos a cada archivo para poder comenzar con la lectura del archivo.

  Después, se manda a llamar el método mezclaE (este es definido más adelante) que se encarga de la primera parte del algoritmo de Mezcla Equilibrada y se iguala a la variable booleana, ya que dentro de mezclaE se comprueba si el algoritmo está ordenado y retorna un valor booleano dependiendo de esto. Luego se entra a un ciclo do-while en donde se manda a llamar al método mezclaD (también definido posteriormente) que hace la segunda parte del algoritmo de ordenamiento. Al finalizar este algoritmo se vuelve a llamar a mezclaE e igual se iguala a isSorted. Este ciclo se repetirá mientras isSorted sea false.

  Por último, se imprime en pantalla el número de iteraciones realizadas y las direcciones en donde se encuentran el archivo ordenado y los archivos auxiliares.

- mezclaE(BufferedReader reader, String carpetaPath)
  Este es un método abstracto y se encarga de la primera parte del algoritmo de Mezcla Equilibrada, el cual consiste en realizar particiones tomando secuencias ordenadas de máxima longitud. Recibe un objeto de tipo BufferedReader correspondiente al archivo que se desea particionar, debido a que este método se ejecuta varias veces es necesario tener guardadas las líneas en donde se va quedando el algoritmo de ordenamiento. Además de esto se recibe un String correspondiente a la dirección de la carpeta en donde se encuentran los archivos para ordenar.

  La razón de que este sea un método abstracto es debido a que existen tres clases llamadas MezclaNombre, MezclaApellido y MezclaCuenta. Cada una corresponde a un criterio para ordenar el archivo por lo que cada clase implementa estos métodos de una manera un poco diferente.

El funcionamiento de este método sigue la misma lógica para todos los casos:

Primero se instancian todos las variables y objetos a utilizar:

- boolean b : bandera que ayuda en la escritura las particiones.
- Boolean isSorted : bandera que indica cuando el archivo está ordenado.
- Alumno alumnoI y Alumno alumnoD : objetos que se irán comparando para particionar el archivo.
- Dato dato : objeto para los métodos de la clase Dato.
- String cadena : arreglo de caracteres que irá almacenando lo leído en el archivo.

Una vez hecho esto se procede a leer la primera línea del archivo. Si esta cadena es nula, significa que el archivo está vacío y por ende "ordenado" y se termina el programa. Por otro lado, si la cadena es igual al carácter "#", significa que está al inicio de un nuevo bloque y por lo tanto debe leer la siguiente línea. Si esta siguiente línea está vacía significa que se llegó al final del programa y se termina.

Ya que se aseguró de que esta cadena contiene el texto esperado se entra a un ciclo do-while. Con este ciclo se hará la partición del archivo. Primero se instancia el objeto alumnoI con ayuda del método obtenerDato y con la cadena leída anteriormente. Luego, se lee la siguiente cadena y se comprueba que no esté vacía ni sea igual a "#" para así poder instanciar a alumnoD. En caso de que no se cumpla con esto significa que estamos en el último elemento del archivo y por lo tanto se procede a escribirlo en alguno de los archivos auxiliares f1 o f2 dependiendo de la bandera b.

En caso de contar con los dos alumnos se procede a compararlos. Esta comparación se hace de acuerdo con el atributo del alumno con el que se quiere ordenar y para cada caso hay una clase diferente.

Si el atributo elegido del alumnoI es menor que el del alumnoD se procede a escribir al alumnoI en el archivo f1 si la bandera b es true y f2 si es false. Esto se hace con ayuda del método escribirDato. En caso de que alumnoD sea mayor que alumnoI significa que estamos al final de un bloque, por lo que se procede a escribir alumnoI en el archivo f1 o f2 dependiendo de la bandera b. En este momento se cambia el valor de la bandera b, ya que al terminar un bloque de la partición se

tiene que empezar a escribir el siguiente bloque en el otro archivo. La escritura de este alumno se hace con el método escribirDato2 ya que al ser el final de un bloque se tiene que dejar un espacio para representar el final de un bloque.

Todo este procedimiento se repetirá mientras que la cadena leída sea diferente del valor nulo, ya que en cuanto se llegue a este momento se habrá llegado al final del archivo. Para terminar, se escribe en los archivos f1 y f2 el carácter "#" con ayuda del método addHash para indicar la finalización de una iteración.

Por último, se devuelve el valor booleano isSorted. Esta última variable empieza teniendo un valor true y cambia a false en el momento en el que termina un bloque ordenado y empieza otro. Esto es debido a que cuando el archivo ya está ordenado es como si se estuviera queriendo particionar un archivo con un solo bloque todo ordenado.

Al principio, este método recibe al archivo original, pero posteriormente recibirá un archivo f0 que contiene las mezclas de las particiones generadas en f1 y f2 para así poder volverlo a particionar ahora con bloques más grandes hasta que sólo quede uno solo.

- mezclaD(boolean isSorted,BufferedReader lectura_f1, BufferedReader lectura_f2, String carpetaPath)
  Este es un método abstracto y se encarga de mezclar las particiones contenidas en los dos archivos auxiliares editados en el método anterior para producir secuencias ordenadas escritas en otro archivo llamado f0. Al igual que mezclaE, este método es implementado de diferentes formas dependiendo de la clase del criterio a ordenar, de ahí que sea abstracto.

  Los parámetros que recibe este método son los siguientes:

  - boolean isSorted : bandera que indica si el archivo está ordenado.
  - BufferedReader lectura_f1 : BufferedReader del archivo f1.
  - BufferedReader lectura_f2 : BufferedReader del archivo f2.
  - String carpetaPath : dirección del directorio donde están almacenados los archivos.

  Lo primero que hace este programa es verificar que el archivo no este ordenado con ayuda de isSorted. En caso de estar ordenado ya se termina el método.

  Después de eso se declaran las variables a utilizar:

- String cadenaf1 y cadena f2 : cadenas de texto que irán almacenando lo leído en el archivo en f1 y f2 respectivamente.
- Alumno alumnof1 y alumno f2 : objetos que se irán comparando para mezclar las particiones.
- Dato dato : objeto para los métodos de la clase Dato.

Luego de declarar las variables y objetos, se comienzan por leer las primeras líneas de ambos archivos auxiliares. En caso de que alguno de los dos sea nulo quiere decir que nos encontramos al final del archivo y ya no hay nada que leer, entonces se termina el método. Si cadenaf2 es igual a "#" significa que no hay ningún bloque en f2 y sólo se procede a escribir todo el otro bloque de f1 y se termina el programa. Esto sólo ocurre cuando el usuario ingresa un archivo completamente ordenado.

Una vez asegurado todo lo anterior se procede a instanciar los objetos alumnof1 y alumnof2 con las respectivas cadenas de sus archivos y con el método obtenerDato. Después de eso se entra a un ciclo do-while en el que se irán mezclando todas las particiones.

Si las cadenas de f1 y f2 no están vacías se comparan alumnof1 y alumno f2 (de nuevo, cada clase los compara de manera diferente). En caso de que alumnof1 sea menor que alumnof2 se escribe a alumnof1 en el archivo f0 y se lee la siguiente línea de f1 para obtener al siguiente Alumno en caso de que esta no sea un salto de línea o un "#". Por otro lado, si alumnof2 es menor, se escribe en f0 y se lee la siguiente línea para obtener al siguiente alumno de ese mismo archivo.

Si las cadenas de f1 y f2 están vacías quiere decir que nos encontramos en el final de un bloque, por lo que se proceden a leer las siguientes líneas mientras que sigan vacías y sean diferentes a "#". Todo esto se hace con dos ciclos while para cada cadena y dentro de ellos se van leyendo las líneas e instanciando a los objetos. En el momento en el que abas cadenas sean iguales a "#" quiere decir que nos encontramos en el final de los dos archivos por lo que se procede a salir del ciclo.

Después de hacer esta condición se procede a comprobar otra. El propósito de esta nueva condición es escribir los elementos sobrantes que no fueron escritos debido a que se finalizó la escritura de un bloque de mayor tamaño.

Primero se empieza comprobando que la cadenaf2 sea igual a "#" o este vacía, en caso de cumplirse esto se puede decir que se termino con

la lectura de un bloque en f2 y que aún hay elementos en f1. Por lo tanto, se entra a un ciclo do-while en el que se va escribiendo a alumnof1 para luego leer una nueva línea y volver a instanciar a alumnof1 para volverlo a escribir, todo esto se hace mientras que cadenaf1 sea diferente de nulo, diferente de un salto de línea y diferente de "#".

En caso de que cadenaf1 sea igual a "#" o este vacía quiere decir que se acabó de leer un bloque en f1 y aún quedan elementos en el otro bloque de f2. Aquí se hace lo mismo que con cadenaf2 sólo que se escriben los elementos sobrantes de f2.

Todo este procedimiento se irá repitiendo mientras que ninguna de las dos cadenas sea nula o igual a "º#".

Una vez fuera del ciclo se comprueba que ninguna de las dos cadenas haya terminado en un salto de línea y en caso de ser así se lee la siguiente línea por si se repitiese el método se inicie a leer desde la nueva iteración.

Por último, se añade un "#" al archivo f0 para indicar la finalización de una iteración.

7.3.2. Clase MezclaNombre

Esta clase se encarga de ordenar las claves del nombre de un alumno en un archivo de texto. Hereda de MezclaEquilibrada y sobrescribe los métodos mezclaE y mezclaD para poder comparar a los objetos de la siguiente forma:
if(alumnoI.getNombre().compareTo(alumnoD.getNombre())<= 0)

7.3.3. Clase MezclaApellido

Esta clase se encarga de ordenar las claves del apellido de un alumno en un archivo de texto. Hereda de MezclaEquilibrada y sobrescribe los métodos mezclaE y mezclaD para poder comparar a los objetos de la siguiente forma:
if(alumnoI.getApellido().compareTo(alumnoD.getApellido())¡=0)

7.3.4. Clase MezclaCuenta

Esta clase se encarga de ordenar las claves del numero de cuenta de un alumno en un archivo de texto. Hereda de MezclaEquilibrada y sobrescribe los métodos mezclaE y mezclaD para poder comparar a los objetos de la siguiente forma:
if(alumnoI.getNoCuenta()¡=alumnoD.getNoCuenta())

7.4. Análisis de Polifase

La implementación de polifase se divide en cuatro clases agrupadas en el paquete polifase, y utiliza a las clases dato y alumno del paquete dato. Los nombres de las clases son Polifase, Merge, InsertionSort y FilesDirect.

7.4.1. Clase Polifase

La clase polifase es el método principal de polifase. Dentro de ella se pueden encontrar tres variantes del método sort, una para cada criterio de ordenamiento. Lo mismo sucede con las clases Merge e InsertionSort. Para este análisis y explicación del programa, se tomara en cuenta solo la versión para número de cuenta y se mencionarán las diferencias con las variantes.

El método sortNum recibe como parámetro el nombre del archivo que se va a ordenar. Algo importante que notar es que recibe únicamente el nombre del archivo y no su ruta completa, ya que esta se genera más adelante.

Lo primero que hace este método es instanciar algunas de las clases necesarias, como FilesDirect, File, FileReader, BufferedReader, Merge y Dato. Con el método rutaFolder de FilesDirect se obtiene la ruta del directorio donde se guardarán los archivos. En seguida, se crea dicho directorio junto a los archivos.

Luego, el método comienza con el proceso de separar el archivo de entra en bloques. Dentro de un do while, el programa verifica que haya al menos 10 elementos para crear el primer bloque, si no es así, cuenta cuantos elementos realmente hay. Luego, el programa llama al método leer dato de la clase dato. Este método recibe desde que linea debe empezar a leer los datos, hasta que otra linea debe dejar de leerlos y devuelve una lista con objetos Alumno.

Después, esa lista es ordenada con el ordenamiento interno, en este caso con la versión para números de cuenta llamada inSortNum de la clase InsetionSort. Aquí es donde esta la única diferencia entre las diferentes versiones del método polifase, ya que la versión para nombres llama a inSortNom y la versión para apellidos a inSortApe.

posteriormente, se escriben los datos ordenados en el archivo auxiliar F1. Finalmente, el programa verifica que el archivo aun no esté vacío. Si esta vacío, sale del ciclo do-while, si no,vuelve a contar 10 elementos, los ordena y los escribe, esta vez en el archivo F2 (La siguiente escritura será en F1, la siguiente en F2 y así sucesivamente). Este proceso se repite hasta que no haya más datos que leer en el archivo original.

Una vez que terminó de separar los datos del archivo original entre los archivos F1 y F2, el método llama al método MergeNum de la clase Merge.

7.4.2. Clase Merge

La clase merge tienen tres métodos, mergeNum, mergeApe y mergeNom, cada una para un criterio de ordenamiento. Como ya se dijo, solo se explicará la versión para número de cuenta y se harán notar las pequeñas diferencias entre las variantes.

Cunado se instancía a esta clase, el constructor obtiene la ruta al directorio donde se guardarán los archivos con una instancia de la clase FilesDirect, que se almacena como un atributo del objeto.

Los métodos de Merge reciben 5 parámetros, la ruta de los dos archivos de los que se va a leer, la de los dos archivos a los que se va a escribir y un parámetro para tener un recuento de cuántas llamadas recursivas se han hecho.

Lo primero que se hace es crear una instancia de la clases File, FileReader y BufferedReader para cada archivo de entrada. En la primera iteración los archivos de entrada serán F1 y F2 que son los archivos donde se colocaron los bloque ordenados internamente. Después, con ayuda del parámetro que nos dice en que llamada recursiva se está, se calcula la cantidad de arrobas que se tienen que contar para llegar a la iteración correcta en los archivos. Las diferentes iteraciones de los archivos se separan por arrobas, por esta razón, es necesario hacer lo anterior.

Luego,con lo obtenido, se posiciona a los lectores de ambos archivo en la primera linea de la iteración. En seguida, se le la primera linea; si está vacía, se deduce que la iteración también lo está, lo que significa que el archivo ya está ordenado porque todos los datos ya están en un solo archivo. Si sucede esto, se despliega un mensaje indicando el archivo donde están los datos ordenados y se detienen las llamadas recursivas. Si las primeras lineas si tienen contenido, se procede a combinar los datos de ambos archivos.

En un while que se repite mientras no se llegue al final de archivo, se ejecuta otro while mientras no se llegue al final del archivo o del bloque de datos. El primer while se repite mientras siga habiendo datos en la iteración y el segundo solo mientras haya datos en el bloque de datos actual.

Dentro del segundo while, se convierten las lineas leídas a objetos de tipo Alumno con ayuda del método obtenerDato de la clase dato. Luego, se compara el atributo de número de cuenta de ambos objetos. Los objetos están nombrados como alumL y alumR (izquierda y derecha) donde en el caso de la primera iteración alumR proviene del archivo F1 y alumL de F2. Si el número de cuenta de alumR es mayor al de alumL, se escribe en uno de los archivos, en caso contrario se escribe a alumL. Finalmente, se lee la siguiente linea del archivo cuyo alumno tuvo un número de cuenta menor.

Los archivos de escritura se van intercambiando, es decir, en la primero iteración del while se escribe en F4, la segunda en F3 la tercera en F4 y así sucesivamente hasta que se llegue al final del bloque de datos de uno de los dos archivos.

En las comparaciones anteriores es donde se pueden encontrar las diferencias entre las versiones del método merge. Para nombres, se compara el atributo de nombre de los dos alumnos con el método compareTo justo después de haberles aplicado el método toUpperCase de las cadenas. Si pudo haber utilizado el método compareToIgnoreCase, pero al momento de programar esto no se conocía su existencia. Lo mismo ocurre en la versión para apellidos, pero en lugar de ocupar el atributo de nombre, se utiliza el de apellido.

Con este proceso, se logran unir dos de los bloque de una iteración de los archivos, pero hay un problema. Como en el proceso anterior el ciclo se detiene cuando se alcanzaba un espacio en blanco en solo uno de los archivos, pueden quedar alumnos en el bloque de datos del otro archivo. Por esta razón, en dos ciclos while(Uno para el caso en que los elementos hayan quedado en F1 y otro para el caso en que hayan quedado en F2) se leen y escriben los datos sobrantes. Este proceso se repite para todos los bloques presentes en la iteración.

al final de este ciclo, se lee la siguiente linea de ambos archivos. Si está vacía o es null en alguno de los dos archivos, significa que no hay un bloque más, por lo que se sale del ciclo. Si tienen contenido se repite el ciclo.

Después de terminar el ciclo anterior, se terminaron de unir todos los pares de archivos, pero ¿Qué pasa si alguno de los archivos tiene un bloque más que el otro? Ese último bloque es ignorado. Por esta razón, después de salir del ciclo anterior, otro par de ciclos (Uno para el caso en que el bloque extra hayan quedado en F1 y otro para el caso en que haya quedado en F2) se leen los datos del archivo de entrada y se escriben en el correspondiente de salida.

Luego de terminar el proceso, se hace una llamada recursiva al mismo método, donde los archivos de lectura se vuelven los de escritura y los de escritura los de lectura. Finalmente, se utiliza el método close de las instancias de bufferedReader y FileReader.

7.4.3. Clase InsertionSort

Como su nombre indica, esta clase almacena métodos para realizar Insertion Sort con los diferentes criterios de ordenamiento. Se explicará la versión para número de cuenta y se expondrán las diferencias entre las versiones.

Este método recibe un ArrayList de objetos Alumno.

Después, con el método size de ArratList se obtiene la cantidad de elemento de la lista. Luego, se recorre la lista desde 1 hasta $n - 1$ y cada elemento se compara con los que están antes de él (que están ordenados) hasta que se encuentre uno que sea menor o se llegue al comienzo de la lista. En ese momento, se inserta al elemento justo después del elemento que es menor que él, o al inicio de la lista, si es que es el menor de todos. Por último, el método devuelve la lista ordenada.

Para hacer las comparaciones entre elementos se utiliza el getter getNoCuenta() de los alumnos y el comparador de relación <= de java. Aquí es donde esta la diferencia entre las versiones. para nombres y apellidos se utiliza el método compareTo de las cadenas. Para evitar errores, se utiliza el método toUpperCase justo antes del anterior.

7.5. Radix

7.5.1. RadixFiles

En esta clase se crean los archivos necesarios para el funcionamiento del programa. Con el fin de tener todo más ordenado, se crea un directorio llamado radixFiles en el que se almacenará una carpeta del mismno nombre del archivo, dentro de ella estarán todas las iteraciones y el archivo ordenado, el camino a esta carpeta estará dado por el atributo folderpath.

public RadixFiles(String archivoD) Crea el camino de la carpeta de iteraciones, su parámetro es el nombre del archivo original , puesto que con él se creará el nombre de la carpeta para las iteraciones.

public String getPath() Método que obtiene la ruta al folder de las iteraciones, regresa esta ruta.

public File newFile(String nombre) Crea un nuevo archivo en la carpeta creada para el archivo a ordenar, su paramétro es el nombre del archivo que se va a crear, que es lo que regresa.

public BufferedReader files(String filename,String archivo) Su parámetro es el nombre de un archivo del cuál se creará el lector, así como el nombre del archivo a ordenar, de nuevo, para saber en qué carpeta crear los archivos. Crea un archivo en la carpeta y regresa su buffer de lectura.

7.5.2. Clase RadixSort

En esta clase se encuentran los algoritmos de ordenamiento por Radix, ya sea por nombre, apellido o número de cuenta.

readWrite(int n, BufferedReader file) Sus parámetros son un entero n, que es el número de líneas y el BufferedReader del archivo que se quiere leer. Lee un archivo por un cierto número de líneas y escribe todo su contenido en el archivo list.

getMax(Alumno alumno, int max) Sus parámetros son un alumno, del que obtiene el número de cuenta, y un entero, que es el máximo actual. Obtiene el número máximo de dígitos en los números de cuenta.

getMaxLength(Alumno alumno, int max, int met) Obtiene la longitud máxima de una cadena, ya sea un apellido o un nombre, si met es igual a 1, la de apellidos, si no, se obtiene la longitud del nombre.

printIte(String archivoD, int ite) Recibe como parámetro el archivo en el que se quiere escribir y el número de iteración. Imprime una cadena que indica qué iteración está ocurriendo en los archivos auxiliares.

RadixSortC (String archivo) En esta clase se ordenan números de cuenta por radix, recibe cómo parámetro el archivo que va a ordenar.

Se crea el directorio para los archivos y posteriormente se crean 11 BufferedReaders, uno para cada archivo auxiliar que se necesita en el programa, en este caso es uno para cada dígito y uno para la lista de elementos tras las iteraciones.

Se copia el contenido del archivo original al de la lista y se encuentra el número máximo de números de cuenta, a partir del último dígito de este número se va buscando el dígito menos significativo para después colocarlo en el archivo auxiliar correspondiente. Por ejemplo, si estuviésemos evaluando unidades y se tiene el número 546, 6 sería el dígito significativo y se escribiría en el archivo auxiliar f6. Una vez que se recorrieron todos los datos, se leen los alumnos que están en cada uno de los archivos de dígito y se van colocando en la lista par realizar otra iteración si es que es posible. Al final se cierran los BufferedReaders y se escriben los datos ordenados en un archivo.

RadixSortA(String archivo) Recibe cómo parámetro el archivo que va a ordenar, este método implementa RadixSort para apellidos. Al igual que en el anterior método, se crea el directorio para todos los archivos de texto con sus lectores y se crean 27 de ellos, ahora uno por cada letra del abecedario, más la lista en la que se copiaran después de cada iteración.

Se lee cada linea del archivo original y se copia en el archivo de lista, en este proceso también se encuentra cuál apellido es el de más tamáño usando el método getMaxLength.

Desde la posición más grande de las cadenas, hasta la más pequeñas, se realiza el siguiente proceso, Se lee cada línea de la lista, si es lo suficientemente larga se obtiene el carácter significativo y se coloca en el archivo auxiliar que le toque, si no es así, se volverá a escribir en la lista.

Cuando se acaben de recorrer, se leen todos los archivos auxiliares y se vuelven a escribir en la lista, esto se repite hasta tener datos ordenados.

Una vez que estén ordenados, serán copiados en otro archivo dentro del directorio creado antteriormente, finalmente se cierran los BufferedReaders.

RadixSortN(String archivo) Este método recibe cómo parámetro al archivo original, igual que los dos anteriores, y luego se realiza exactamente lo mismo que en radix para apellidos, el proceso es idéntico, la única diferencia es que en vez de usar el apellido como criterio, se utiliza el nombre para obtener los dígitos significativos.

7.5.3. Análisis

Con esta implementación se puede ver que cómo funcionaría en memoria externa al usar archivos como colas, en ningún momento se tienen todos los datos en la memoria interna. Cada iteración de radix es de complejidad O(n), puesto que recorre todos los datos al sacarlos de las colas y colocarlos en la lista, si se tiene una longitud w de las claves a ordenar, la complejidad temporal del algoritmo es O(wn) para todos los casos.

Radix no es comúnmente usado para ordenar archivos, a diferencia de Polifase o Mezcla Equilibrada. Es uno de los algoritmos lineares más efectivos, resulta útil si se quieren ordenar claves largas de manera estable, puesto

que las recorre en tiempo O(n), sus desventajas son que puede ser lento al compararse con otros como Merge y QuickSort, se tiene que adaptar dependiendo de si se quiere trabajar con cadenas o números, además de que utiliza memoria adicional del mismo tamaño que la del archivo a ordenar.

7.5.4. Consideraciones

Para que pueda funcionar correctamente se necesita que no haya un salto de línea adicional al final del archivo ordenar y que este exista.

7.6. Main

La clase Main contiene al método main del del programa. EL método main sirve únicamente para desplegar el menú y para llamar a los métodos de los diferentes ordenamientos. Primero, pide al usuario el nombre del archivo a ordenar y lo guarda como cadena. Con una instancia de File, el programa verifica que el archivo exista. Si es así, continua con la ejecución, si no, se termina prematuramente la ejecución.

Luego, despliega un menú con los diferentes ordenamientos. Cuando el usuario elige una opción, el programa la almacena como una cadena. Después, en una serie de if-else, se utiliza el método equals de las cadenas para comparar lo ingresado por el usuario con las diferentes opciones. Se eligió esta aproximación debido a que evita que el programa se cierre de manera inesperada por el ingreso de tipos de datos diferentes a los esperados.

Si el usuario elige cualquiera de los algoritmos, se despliega otro menú que le indica elegir por que criterio quiere ordenar los datos. Al igual que antes, se recibe la opción del usuario como cadena para evitar errores relacionados con el tipo de dato. Después, esa cadena se convierte en un entero con el método valueOf y en un switch se valida la opción del usuario.

En caso de quiera ordenar con Mezcla equilibrada, el programa utiliza polimorfismo para ordenar por el criterio correcto. A una variable de la clase Mezcla equilibrada, se le asigna la referencia a una instancia de su clases hijas MezclaNombre, MezclaApellido o MezclaCuenta dependiendo del criterio de ordenamiento elegido. Por último, se llama al método mezcla de la clase padre.

En caso de elegir polifase, se creará una instancia de la clase de mismo nombre. Dependiendo del criterio de ordenamiento, se llamará la al método sortNum, sortNom o sortApe para ordenar por número de cuenta, nombre o apellido respectivamente.

Por último, en caso de elegir radix, se creará una instancia de RadixSort

y se llamará a los métodos RadixSortN, RadixSortA o RadixC para ordenar por nombre, apellido o numero de cuenta respectivamente, dependiendo de la opción elegida por el usuario.

8. Conclusiones

8.1. Ricardo López Becerra

En este proyecto se pudieron practicar los algoritmos de ordenamiento externo, el manejo de archivos, la programación orientada a objetos y el trabajo en equipo. El tema principal del proyecto fueron los algoritmos de ordenamiento externo, por lo que para poder programarlos debíamos tener un entendimiento completo de los algoritmos. De esta manera, cumplimos el objetivo de implementar los algoritmos de ordenamiento. La programación orientada a objetos fue muy importante para el proyecto debido a que es considerablemente más grande que otros programas que se han hecho en prácticas o actividades. Con la ayuda de herramientas que proporciona el paradigma, se pudo dividir el trabajo más fácilmente. Finalmente, aprender a trabajar en equipo a distancia fue esencial para lograr terminar el programa compartiendo ideas y código.

8.2. Diego Ignacio Núñez Hernández

Definitivamente los objetivos propuestos al inicio del proyecto fueron logrados. Lo más claro es que su pudieron implementar los algoritmos de ordenamiento externo, Polifase, Mezcla Equilibrada y RadixSort sin ninguna complicación. Para poder hacer esto fue necesario aprender a utilizar las herramientas necesarias para manipular archivos de texto, específicamente en Java. Por último, y lo más importante para cumplir estos objetivos fue el trabajo en equipo. Gracias a la buena organización y comunicación que siempre se tuvo desde el principio fue que el proyecto salió bien, cumpliendo con todos los requerimientos e incluso con algunas cosas extra.

8.3. Sofía Elizabeth Cabello Díaz

Para concluir, este proyecto me hizo llevar a la práctica lo visto en teoría acerca de algoritmos de ordenamiento externo, ahora entiendo más cómo funcionan al usar la memoria externa para ir guardando el proceso hasta terminar de ordenar las claves. Considero que fue muy útil para conocer las clases en Java relevantes para el manejo y trabajo con archivos, así

como otros conceptos, como paquetería y documentación usando Javadoc. Adicionalmente, fomentó el trabajo en equipo y el compañerismo para poder terminar de buena forma el proyecto.

8.4. Conclusión general

Hoy en día los algoritmos de ordenamiento externo son muy importantes por la gran cantidad de información que se genera en el mundo. Esta información no puede ser almacenada todo en memoria principal, por lo que es necesario utilizar algoritmos capaces de trabajar con archivos cuando se requiere ordenarlos. Este proyecto nos ayudo a comprender mejor el funcionamiento de este tipo de algoritmos ya que al programarlos se tuvo que comprender su lógica para poder implementarlos.

Una de las partes fundamentales para cumplir con los objetivos propuestos fue el aprender a manejar archivos de texto en el lenguaje de programación Java. Esto es muy importante porque los archivos nos permiten almacenar información de forma persistente y en mayor cantidad. El manejo de archivos en Java no es tan difícil como puede llegar a ser en otros lenguajes, resultando ser más fácil de lo que pensábamos.

La principal razón de que el proyecto haya salido como fue planeado es que se tuvo una buena organización y comunicación entre los integrantes del equipo. Además, aprender el manejo de herramientas de control de versiones como GitHub ayudaron a agilizar el trabajo.

Por todo esto es que tenemos la confianza de afirmar que se cumplieron los objetivos del proyecto además de que pudimos agregar un pequeño extra al programa.

Referencias

[1] Cairó, O., Guardati, S.(2006). *Estructuras De Datos*, tercera edición. D.F., México: McGrawHill.

[2] Oracle.(2018).. *Java Platform Standard Edition & Java Development Kit Version 11 API Specification*

[3] Zhaoxing L.(2019). *Leyenda: An Adaptive, Hybrid Sorting Algorithm for Large Scale Data withLimited Memory.*

[4] Harbour M.,Aldea,M.(2015) *Programación en Lenguaje Java. Tema 10.Entrada/Salida con ficheros)*,

https://ocw.unican.es/pluginfile.php/293/course/section/228/cap10-ficheros.pdf.

[5] Krulis, M. *Critical Evaluation of Existing External SortingMethods in the Perspective of ModernHardware?*, Algorithms Research Group-Faculty of Mathematics and Physics.

[6] Zave, D.(1976) *Optimal Polyphase Sorting*,Computer Science Department, School of Humanities and Sciences, Stanford University.