Trabajo Práctico № 1
Algoritmos de ordenamiento

Spoletini Bruno, Poma Lucas 02/05/2020

Introducción

En este trabajo utilizamos una lista doblemente enlazada no circular, sobre la cual aplicamos tres algoritmos de ordenamiento: Selection Sort, Insertion Sort, Merge Sort. Esta lista contenía personas generadas aleatoriamente en la primera parte del trabajo.

Compilación

El programa cuenta con un archivo makefile, que se encarga de compilar los archivos correspondientes al funcionamiento del programa.
Para ejecutarlo, basta con utilizar el comando make en la terminal, estando situado en el directorio principal.
Esto generará dos ejecutables "ejecutablePersona" "ejecutableLista".

Ejecución del programa

El primero, corresponde al programa GenerarPersonas, el cual recibe como parámetros un archivo de nombres, otro de países, el nombre del archivo de salida, y la cantidad de datos a generar, y se ejecuta de la siguiente manera:

./ejecutablePersona ./primeraParte/nombres1.txt ./primeraParte/paises.txt ./primeraParte/output.txt 1000

Esto generará un archivo "output.txt" con los datos aleatorios generados

El archivo ejecutableLista recibe como parámetro el path del archivo generado por el programa anterior, y crea seis archivos en la carpeta ./segundaParte/sortedLists, en los cuales estará la lista pasada como parámetro ordenada según tres algoritmos de ordenamientos distintos, y con dos criterios de comparacion distintos.
Y se ejecuta de la siguiente manera, situado en el directorio principal:

./ejecutableListas ./primeraParte/output.txt

Estructura del programa

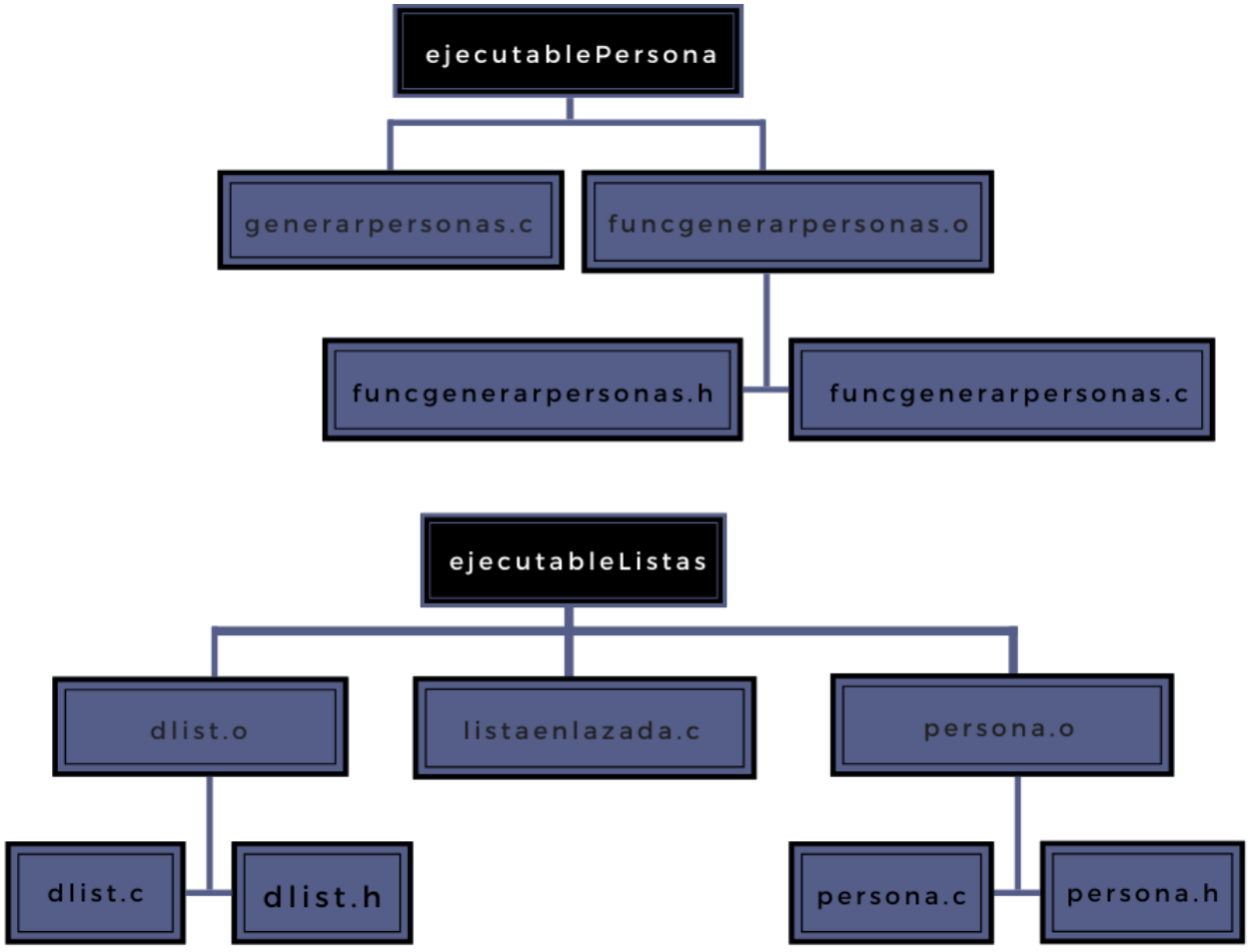
Primera Parte

- generarpersonas.c: contiene la función main
- funcgenerarpersonas.c: contiene las funciones necesarias para el funcionamiento del programa
- funcgenerarpersonas.h: archivo cabecera de funcgenerarpersonas.c

Segunda Parte

- listaenlazada.c: contiene la función main
- persona.c: contiene las fun. respectivas al manejo de la estructura de datos "Persona"
- persona.h: archivo cabecera de persona.h, contiene la estructura de datos "Persona"
- dlist.c: contiene las funciones respectivas a manejo de listas
- dlist.h: archivo cabecera de dlist.c, contiene la declaracion de las estructuras de listas

Árbol de dependencias



Caracteres Especiales

Colocamos $setlocate$ a es_ar.utf8 para utilizar la codificación correcta.
Para corregir el ordenamiento cuando ordenamos por lugar de nacimiento queríamos utilizar la función $strcoll$ que compara las palabras según la localidad seleccionada por $setlocate$, pero no logramos encontrar ningún lenguaje que ordenara de manera correcta las vocales con tilde. Ademas esta función demora mucho mas en comparar que $strcmp$
También, pensamos en antes de colocar el dato en la linked list, convertir las localidades en palabras sin caracteres especiales a la hora de leer el archivo de las personas, ordenar la lista con estas palabras mediante $strcmp$, y luego antes de escribirlas en el archivo de salida, volverlas a su estado original.
Nos pareció difícil de implementar, ya que variaría según el archivo de localidades, pero en contraparte, es seria algo mejor a la hora de mantener la velocidad de los algoritmos de ordenamiento, no creando nuestra propia función de comparación de strings.

Estructuras de datos

Para este trabajo decidimos trabajar con la estructura de lista doblemente enlazada no circular, con una estructura que posee dos punteros, los cuales apuntan al inicio y al final de la lista.

Listing 1: Estructura de datos utilizada.

```
1  typedef struct _DNodo{
2    void *dato ;
3    struct _DNodo *ant;
4    struct _DNodo *sig;
5  } DNodo;
6
7  typedef struct {
8    DNodo *primero;
9    DNodo *ultimo;
10 } DList;
```

Esta estructura nos ofrece las siguientes ventajas frente al resto de implementaciones de listas enlazadas:

- La lista puede recorrerse de forma bidireccional
- La estructura que apunta al inicio y al final de la lista nos permite insertar un elemento al final de esta sin necesidad de recorrerla por completo.
- Las listas cirulares no ofrecen una ventaja significativa respecto de nuestra implementación.

También utilizamos la estructura "Persona", provista por la cátedra.

Comparaciones con ejemplos

Caso 40000 personas:

Ordenamiento por edad:

- **Selection Sort:** 3.892 segundos
- **Insertion Sort:** 4.526 segundos
- **Merge Sort:** 0.080 segundos

Ordenamiento por lugar de nacimiento:

- **Selection Sort:** 7.679 segundos
- **Insertion Sort:** 5.914 segundos
- **Merge Sort:** 0.091 segundos

Caso 20000 personas:

Ordenamiento por edad:

- **Selection Sort:** 0.978 segundos
- **Insertion Sort:** 0.941 segundos
- **Merge Sort:** 0.037 segundos

Ordenamiento por lugar de nacimiento:

- **Selection Sort:** 1.831 segundos
- **Insertion Sort:** 1.293 segundos
- **Merge Sort:** 0.039 segundos

Caso 4000 personas:

Ordenamiento por edad:

- **Selection Sort:** 0.037 segundos
- **Insertion Sort:** 0.024 segundos
- **Merge Sort:** 0.006 segundos

Ordenamiento por lugar de nacimiento:

- **Selection Sort:** 0.062 segundos
- **Insertion Sort:** 0.033 segundos
- **Merge Sort:** 0.006 segundos

Análisis de los resultados

Mediante estas pruebas podemos observar como a medida que crecen la cantidad de elementos en la lista, el tiempo que tardan los algoritmos en ordenarlas es distinto. Por ejemplo, se nota la clara diferencia de como **Merge Sort** crece en el tiempo como $n * \log n$, mientras que **Selection Sort** e **Insertion Sort** demoran mas tiempo, ya que crecen en n^2.
Podemos también ver como ordenar por lugar de nacimiento toma mas tiempo que ordenar por edad, con la misma cantidad de personas, ya que una comparación de $strings$ toma mas operaciones que una comparación de $ints$.
Donde mas se nota esta diferencia es cuando ordenamos por edad, el **Selection Sort** suele tener tiempos cercanos al **Insertion Sort**, pero este ultimo, al realizar menos comparaciones, sufre menos en el tiempo que tarda a la hora de ejecutar el ordenamiento por el lugar de nacimiento.

Conclusiones

El algoritmo más fácil de implementar es **Selection Sort**, pero requiere una mayor cantidad de comparaciones que los otros algoritmos, lo cual ralentiza bastante el tiempo de ejecución.
El siguiente algoritmo es **Insertion Sort** un poco mas eficiente en la mayoria de casos en comparación al anterior.
Merge Sort resulta ser el mas rápido de todos, pero su implementación es considerablemente mas difícil que la de los dos anteriores.

Fuentes consultadas

https://es.wikipedia.org/wiki/Ordenamiento_por_selección
https://es.wikipedia.org/wiki/Ordenamiento_por_inserción
https://es.wikipedia.org/wiki/Ordenamiento_por_mezcla
https://www.geeksforgeeks.org/how-to-measure-time-taken-by-a-program-in-c/
https://www.tutorialspoint.com/c_standard_library/c_function_setlocale.htm
https://www.tutorialspoint.com/c_standard_library/c_function_strcoll.htm