Trabajo Practico N°1

Licenciatura en Ciencias de la Computacion

Estructuras de datos y algoritmos I

Implementacion de metodos de ordenamiento a listas enlazadas generales

Lucas Bachur, Tomas Scalbi

2020

Índice

1. Introduccion

1.1. Motivacion del trabajo.

El objetivo del trabajo es implementar algunos algoritmos de ordenacion sobre listas enlazadas generales. En particular a los algoritmos **Insertion Sort, Selection Sort y Merge Sort.**

1.2. El trabajo consiste en 2 programas en c.

El primero toma los nombres de los archivos de entradas de datos, el nombre del archivo de salida, y la cantidad de datos a generar. Y genera un nuevo archivo con el nombre que se le paso, donde almacena datos generados en forma aleatoria con la informacion de los archivos de entrada, formateados de la siguiente manera:

```
Juan Perez , 19 , Villa General Belgrano
Francisco Jose Maria Olviares , 23 , Rosario
```

El segundo, toma como argumento el nombre del archivo donde esta el juego de datos. Y genera 6 archivos donde se encuentran estos datos ordenados segun 2 criterios diferentes, y 3 metodos distintos, mostrando en la primer linea el tiempo que tardo el algoritmo en ordenar la lista en segundos, sera facil ver que algoritmo ordeno cada lista puesto que el nombre del archivo contendra primero el criterio de comparacion y luego el metodo que se utilizo. Un ejemplo:

```
Tiempo de ordenamiento: |0.037133|

FLOR MARIA GODINEZ, 1, Jerusaln Este
JORGE ENRIQUE CHAVARRIA, 1, Espaa
.
.
.
```

2. Decisiones

2.1. Criterios de comparacion.

Los criterios de comparacion que propusimos son: menor edad y menor largo de nombre. Elegimos estos criterios de comparacion frente a otros porque resultan sencillos de codear y entendemos que el objetivo del trabajo no es crear un criterio de comparacion complejo mas bien es que se aplique correctamente.

2.2. Implementacion de la lista.

Utilizamos **Listas simplemente enlazadas, con un puntero al comienzo y otro al fin** y las definimos con las siguiente estructura:

```
1  typedef struct _GNodo {
2    void *dato;
3    struct _GNodo *sig;
4  } GNodo;
5
6  typedef struct _GList {
7    GNodo *inicio;
8    GNodo *final;
9  } GList;
10
```

Elegimos trabajar con listas simplemente enlazadas pues nos parecio que es la estructura mas simple y creemos que esto nos permite concentrarnos mas en la implementacion de los algoritmos de ordenamiento, el objetivo del trabajo.

Elegimos tambien esta implementacion de listas con puntero al comienzo y al final por sobre la de puntero al inicio solamente para poder acceder rapidamente al final de la lista para aadir elementos. Y a su vez la elegimos por sobre la lista circular puesto que esta ultima ya la habiamos implementado en la practica 2 y queriamos hacer una implementacion nueva.

3. Carpeta de entrega.

En la carpeta donde se entrega este trabajo, hay 2 carpetas: *Generacion* y *Listas*. Y un archivo makefile que contendra las instrucciones de compilacion de los dos programas.

3.1. Generacion

En esta carpeta se encuentran los archivos:
generacion.h: el archivo de cabecera con los prototipos y las declaraciones de tipos de datos utilizados.
generacion.c: el archivo que contiene las implementaciones de las funciones declaradas en *generacion.h*, ademas contiene una definicion: #define BUFFER 80 que se utilizara solo en estas funciones.
generar.c: archivo que contiene el main de programa1.
nombres1.txt: archivo de entrada de datos que contiene un nombre por linea.
paises.txt: archivo de entradad de datos que contiene un pais por linea.

Cabe aclarar que el makefile sabe que los archivos de entrada estan en esta carpeta.

3.2. Listas

En esta carpeta se encuentran los arhicvos:
glist.h: archivo de cabecera que con los prototipos y las declararciones de tipos de datos utilizados con respecto a las listas simplementes enlazadas para este trabajo en particualr.
glist.c: archivo que contiene las implementaciones de las funciones declaradas en *glist.h*
listas.c: archivo que contiene el main del programa2 y las funciones especificas de la estructura de datos que se esta empleando.

Importante, cuando se ejecuten el primer programa, segun nuestro diseo, el archivo de salida con los datos de prueba se creara en esta carpeta.

3.3. makefile

Decidimos utilizar make para agilizar los tiempos que tardabamos escribiendo los comandos en consola. El uso de esta herramienta fue muy util durante la ejecucion del trabajo. Aqui dejamos el codigo que esta dentro de nuestro makefile, donde programa1 se refiere al programa que hace el archivo de juego de datos, y programa2 al que trabaja con las listas.

```
programas: generacion.o generar.o listas.o glist.o
gcc -o programa1.out generar.o generacion.o
gcc -o programa2.out listas.o glist.o

programa1: generacion.o generar.o
gcc -o programa1.out generar.o generacion.o

programa2: listas.o glist.o
gcc -o programa2.out listas.o glist.o

generar.o: Generacion/generar.c
gcc -Wall -Werror -Wextra -g -c Generacion/generar.c

generacion.o: Generacion/generacion.c Generacion/generacion.h
gcc -Wall -Werror -Wextra -g -c Generacion/generacion.c

listas.o: Listas/listas.c
gcc -Wall -Werror -Wextra -g -c Listas/listas.c

glist.o: Listas/glist.c Listas/glist.h
gcc -Wall -Werror -Wextra -g -c Listas/glist.c
```

4. Funcionamiento

El funcionamiento mas especifico de las funciones esta bien explicado cuando estas son declaradas por primera vez, y en las implementaciones de los mismos adjuntamos notas de comentario explicando un poco mas detalladamente las partes que nos parecieron necesarias para mejorar la lectura del codigo.

Caracteres especiales No tuvimos ningun problema con las letras acentuadas ni los caracteres especiales en ningun sistema operativo, asi que no hicimos nada especial en el programa.

4.1. programa1

Programa de generacion de datos de prueba Primeramente se interpretan los argumentos pasados y se declaran banderas que se utilizaran cada vez que querramos crear o abrir un archivo, e indicaran si estos lo hicieron correctamente.

Cuando se pasa un nombre de archivo, ya sea de entrada o de salida, decidimos que nuestro main lo cree o lo busque leer de un lugar especifico que se determina dentro de la funcion, especificamente, los archivos de entrada se encuentran dentro de la carpeta Generacion, y el archivo de salida se escribe en la carpeta Listas.
Si la cantidad de datos a generar es 0, directamente se crea un archivo vacio con el nombre indicado.

ARandom Para asegurar la aleatoriedad y la eficiencia, se nos ocurrio crear un nuevo tipo de dato al que llamamos ARandom, definido de la siguiente manera:

```
typedef struct _random{
  int numLinea;
  int pos;
} ARandom;

```

La estructura ARandom representa un numero generado aleatoriamente y la posicion en la que se genero. De esta manera podemos generar un array de ARandom, donde guardamos los numeros aleatorios que vamos generando al mismo tiempo que la posicion que ocupa en el array original, es decir el orden en el que este numero fue generado. De esta forma, podemos ordenar el arreglo ARandom de menor a mayor por el numLinea para que cuando leamos el archivo lo hagamos de la forma mas eficiente, y a la vez, guardar el orden en el que fueron saliendo para lograr mantener la aleatoriedad.

Generacion de numeros aleatorios Primero que nada, encontramos necesario distinguir el metodo de generacion de cada numero aleatorio segun en que sistema operativo nos encontremos (lo hicimos solo para windows y linux pero lo escribimos de tal forma que sea facil aadir un caso nuevo en el futuro), puesto que el *rand_max* de la funcion *rand()* en windows es menor a la cantidad de lineas del archivo que se nos proporciono como entrada.
Los metodos para generar los numeros aleatorios son los siguientes (copia del archivo *generacion.c*):

```
int numero_random_l (int tope){
  return rand () %tope;
}
int numero_random_w (int tope){
  // Usamos unsigned int como tipo de dato ya que los numeros generados seran
  // siempre positivos.
  unsigned int a = rand ();
  unsigned int b = rand ();
  // Multiplicamos los dos numeros generados aletoriamente, pero como puede
  // superar el tamano de un int usamos long int (tambien unsinged).
  unsigned long int c =  ((long) a * (long) b) %tope;
  return (int)c;
}

```

Basicamente, en Linux resulta de generar el numero con la funcion *rand()* y luego asignarle un tope.
Mientras que en Windows generamos dos numeros int positivos aleatorios para luego multiplicarlos. A ese resultado le hacemos el modulo por el tope que tenemos y ese resultado va a ser el numero aleatorio final.

Generacion arreglo ARandom Este arreglo es el que se comentaba anteriormente. Simplemente le asignamos un espacio en memoria y lo vamos completando a medida que se van saliendo los numeros aleatorios, complentando su campo de *numLinea* con el numero generado, y el campo *pos* con el valor del iterador que se esta utilizando.

Generacion arreglos nombres y paises Ambos arreglos son generados de la misma manera solo que cambia el archivo del cual leen.
Primero se le asigna memoria al puntero a punteros a char, y luego a cada puntero a char de este.
Luego se crea el puntero a ARandom que contendra los numeros de linea y se ordena de forma tal que se pueda leer el archivo una sola vez, y de la forma mas efetiva. Creemos que esto es muy importante y que mejora los tiempos de ejecucion.
Vamos completando el arreglo a devolver con los numeros de linea en el orden en el que salieron originalmente para mantener la aleatoriedad. Tambien notamos que se puede ser que salga varias veces un mismo numero de linea, caso que esta contemplado en nuestra implementacion.
En su momento estuvimos bastante tiempo hasta que en preguntamos por Zulip el problema que teniamos al leer los datos de los archivos de entrada, y este era que estabas haciendo el fscanf de la siguiente forma:

```
fscanf (Archivo, "%[^\n]\n", buffer);

```

Porque no sabiamos que en realidad los finales de linea en Windows estaban hechos de la forma \r\n asi que pudimos adaptarlo a:

```
fscanf (Archivo, "%[^\r\n]\r\n", buffer);

```

Escritura archivo salida Una vez que estan generados sin problemas y adecuadamente cada uno de los arreglos con los datos (el arreglo con los nombres, el arreglo con los paises), se procede a escribir cada linea de la siguiente manera:

```
edad = generar_edad ();
fprintf (archivoSalida,"%s, %d, %s\n", arregloNombres[i], edad, arregloPaises[i]);

```

Dejando el archivo de salida en las condiciones que espera el programa2.

5. programa2

Lo primero que hace el segundo programa es leer el archivo generado por el primero, interpretar su contenido y almacenar esos datos en una lista simplemente enlazada de Personas, siendo Persona el tipo de estructura que esta apuntada a utilizarse en la consigna del trabajo.

Luego, se aplica cada uno de los **3 algoritmos de ordenamiento** a la lista generada segun **2 funciones distintas de comparacion**. Volcamos las listas ordenadas junto con los tiempos de ejecucion de los algoritmos en 6 archivos distintos.

5.1. Comparaciones en tiempo de ejecución

Empezamos a comparar velocidad a partir de los 1000 elementos:

Metodo de comparacion	**Selection**	**Insertion**	**Merge**
EDAD	0,0081088	0,002385	0,0002142
LARGO NOMBRE	0,0128264	0,0066656	0,0008458

5000 elementos.

Metodo de comparacion	**Selection**	**Insertion**	**Merge**
EDAD	0,8411596	0,06091354	0,0012254
LARGO NOMBRE	0,3160378	0,1504216	0,0019852

10000 elementos.

Metodo de comparacion	**Selection**	**Insertion**	**Merge**
EDAD	0,4294468	0,295468	0,0030076
LARGO NOMBRE	1,6256668	0,7792646	0,0049134

20000 elementos.

Metodo de comparacion	**Selection**	**Insertion**	**Merge**
EDAD	2,0903696	1,525693	0,0071012
LARGO NOMBRE	7,9087686	4,1888458	0,12585

30000 elementos.

Metodo de comparacion	**Selection**	**Insertion**	**Merge**
EDAD	4,8456682	3,8137686	0,0114358
LARGO NOMBRE	18,5267306	10,484457	0,0194516

Cada uno de estos resultados es un promedio de 5 ejecuciones distintas con diferentes juegos de datos.

Llegamos a la conclusión de que el *merge sort* es el algoritmo mas rapido, lo cual era esperable ya que es el de menor complejidad. El *selection sort* y el *insertion sort* mantienen tiempos parecidos para volumenes de datos chicos, pero cuando empezamos a manejar cantidades mas significativas comenzamos a ver una diferencia.