Trabajo Práctico N°1

Licenciatura en Ciencias de la Computación

Estructuras de datos y algoritmos I

Cátedra: Federico Severino Guimpel, Mauro Lucci, Martín Ceresa, Emilio López, Valentina Bini

Implementación de métodos de ordenamiento a listas enlazadas generales

Lucas Bachur, Tomás Scalbi

2020

Índice

1. Introducción

1.1. Motivación del trabajo.

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

1.2. Breve descripción.

El primer programa al que llamamos *programa1* 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 pasó, donde almacena datos generados en forma aleatoria con la información de los archivos de entrada, formateados de la siguiente manera:

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

El segundo programa al que llamamos *programa2*, toma como argumento el nombre del archivo donde está el juego de datos. Y genera 6 archivos donde se encuentran estos datos ordenados según 2 criterios diferentes, y 3 métodos distintos, mostrando en la primer línea el tiempo que tardó el algoritmo en ordenar la lista en segundos, será fácil ver que algoritmo ordenó cada lista puesto que el nombre del archivo contendrá primero el criterio de comparación y luego el método que se utilizó. Un ejemplo:

```
Tiempo de ordenamiento: |0.037133|

FLOR MARIA GODINEZ, 1, Jerusalén Este
JORGE ENRIQUE CHAVARRIA, 1, Chad
.
.
.
```

2. Decisiones

2.1. Criterios de comparación.

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

2.2. Implementación de la lista.

Utilizamos **Listas simplemente enlazadas, con un puntero al comienzo y otro al final** 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 pareció que es la estructura más simple y creemos que esto nos permite concentrarnos más en la implementación de los algoritmos de ordenamiento, el objetivo del trabajo.

Elegimos también esta implementación de listas con puntero al comienzo y al final por sobre la de puntero al inicio solamente para poder acceder rápidamente al final de la lista para añadir elementos. Y a su vez la elegimos por sobre la lista circular puesto que esta última ya la habíamos implementado en la práctica 2 y queríamos hacer una implementación nueva.

3. Carpeta de entrega.

En la carpeta donde se entrega este trabajo, hay 2 carpetas: *Generacion* y *Listas*. Y un archivo makefile que contendrá las instrucciones de compilación 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*, además contiene una definición: #define BUFFER 80 que se utilizará 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 línea.
paises.txt: archivo de entrada de datos que contiene un país por línea.

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

3.2. Listas

En esta carpeta se encuentran los archivos:
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 particular.
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 específicas de la estructura de datos que se está empleando.
Persona: Carpeta que contiene *persona.h* y *persona.c*, archivos que contienen las funciones asociadas al tipo de dato persona con el que se desea trabajar en este proyecto.

Importante, cuando se ejecute el primer programa, según nuestro diseño, el archivo de salida con los datos de prueba se creará en esta carpeta.

3.3. makefile

Decidimos utilizar make para agilizar los tiempos que tardábamos escribiendo los comandos en consola. Al comienzo tuvimos algunas dificultades para escribir correctamente el makefile, hasta que consultamos por Zulip y lo solucionamos.
Esto nos ayudo también a entender como funcionaban las simplificaciones que estábamos haciendo cuando escribíamos en consola:

```
gcc -Wall -Werror -Wextra -g -c generacion.c
gcc -Wall -Werror -Wextra -g generar.c generacion.o
```

Donde esto es una simplificación de:

```
gcc -Wall -Werror -Wextra -g -c generacion.c
gcc -Wall -Werror -Wextra -g -c generar.c
gcc -Wall -Werror -Wextra -g generar.o generacion.o
```

El uso de esta herramienta fue muy útil durante el transcurso del trabajo.

Crear y ejecutar Solo escribiendo el comando make se compilará todo lo necesario para la ejecución del programa. Un ejemplo comple de comandos a ingresar en la consola es (linux):

```
make
./programa1.out "nombres1.txt" "paises.txt" "salidas1.txt" "20000"
./programa2.out "salidas1.txt"
make clean
```

4. Funcionamiento

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

Carácteres especiales No tuvimos ningún problema con las letras acentuadas ni los carácteres especiales en ningún sistema operativo[1], así que no hicimos nada especial en el programa.

4.1. programa1

Programa de generación de datos de prueba Primeramente se interpretan los argumentos pasados y se declaran banderas que se utilizarán cada vez que querramos crear o abrir un archivo, e indicarán 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 específico que se determina dentro de la función, específicamente, 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 vacío con el nombre indicado.

ARandom Para asegurar la aleatoriedad y la eficiencia, se nos ocurrió 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 número generado aleatoriamente y la posición en la que se generó. De esta manera podemos generar un array de ARandom, donde guardamos los números aleatorios que vamos generando al mismo tiempo que la posición que ocupa en el array original, es decir el orden en el que este número 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 más eficiente, y a la vez, guardar el orden en el que fueron saliendo para lograr mantener la aleatoriedad.

Generación de numeros aleatorios Primero que nada, encontramos necesario distinguir el método de generación de cada número aleatorio según en que sistema operativo nos encontremos (lo hicimos solo para windows y linux pero lo escribimos de tal forma que sea fácil añadir un caso nuevo en el futuro), puesto que el *rand_max* de la función *rand()* en windows es menor a la cantidad de líneas del archivo que se nos proporcionó como entrada.
Los métodos 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;
}

```

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

Generación 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 números aleatorios, complentando su campo de *numLinea* con el número generado, y el campo *pos* con el valor del iterador que se esta utilizando.

[1] Windows ni Linux

Generación arreglos nombres y países Ambos arreglos son generados de la misma manera solo que cambia el archivo del cual leen.
Primero se pide la memoria necesaria.
Luego se crea el puntero a ARandom que contendrá los números de línea y se ordena de forma tal que se pueda leer el archivo una sola vez, y de la forma más efetiva. Creemos que esto es muy importante y que mejora los tiempos de ejecución.
Vamos completando el arreglo a devolver con los números de línea en el orden en el que salieron originalmente para mantener la aleatoriedad. Tambien notamos que se puede ser que salga varias veces un mismo número de línea, caso que esta contemplado en nuestra implementación.
En su momento estuvimos bastante tiempo hasta que en preguntamos por Zulip el problema que teníamos al leer los datos de los archivos de entrada, y este era que estabas haciendo el fscanf de la siguiente forma:

```
1  fscanf (Archivo, "%[^\n]\n", buffer);
2
```

Porque no sabíamos que en realidad los finales de línea en Windows estaban hechos de la forma \r\n así que pudimos adaptarlo a:

```
1  fscanf (Archivo, "%[^\r\n]\r\n", buffer);
2
```

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 países), se procede a escribir cada línea de la siguiente manera:

```
1  edad = generar_edad ();
2  fprintf (archivoSalida,"%s, %d, %s\n", arregloNombres[i], edad, arregloPaises[i]);
3
```

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

5. programa2

Para este programa, decidimos utilizar archivos aparte para las funciones asociadas a glist, y otro para las funciones asociadas a Persona.
Primeramente se interpreta el argumento pasado al programa como el nombre del archivo donde están los datos, y se le coloca el directorio detrás, para hacerle saber al programa que este archivo esta en realidad en Listas/.
Lo segundo que hace el segundo programa es leer este archivo, 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, como se sugiere en la consigna, decidimos crear una función la cual nombramos: glist_ordenar_archivar , que toma la lista como parametro entre otros[2].
Luego, crea una copia, pero solo de los nodos, es decir, los datos siguen siendo los de la lista original por lo que si se modifican, estos cambios tambien se verán reflejados, pero como el propósito de esta función no incluye modificar estos datos decidimos hacer esto para no copiar la lista completa, mejorando el rendimiento.
Para contar el tiempo de cada algoritmo decidimos utilizar *clock()*, incluido en la librería *time.h*.

```
1  clock_t begin = clock();
2  listaAOdenar = metodo_ordenamiento (listaAOdenar, comparar);
3  clock_t end = clock();
4  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
5
```

Luego de ordenar esta copia, se vuelca en el archivo, y se libera. Nuestro programa llama un total de 6 veces a glist_ordenar_archivar con distintos parámetros antes de liberar la lista original y finalizar el programa.

5.1. Selection sort

Primer método que implementamos, nos basamos en el pseudocódigo provisto en la consigna, y la animación de wikipedia también fue de mucha ayuda. Al ver esta última, decidimos que íbamos a intercambiar los punteros void que representan datos puesto que si intercambiásemos nodos, sería bastante mas engorroso, se deberían hacer más cambios de punteros lo que empeoraría el rendimiento y la simplicidad del código sin ninguna ventaja aparente.

5.2. Insertion sort

Este fue el segundo método que implementamos, y nos basamos en la animación del link que se propuso en la consigna, también un video de udiprod[3] que dejaremos en la bibliografía donde se muestra gráficamente como funciona el algoritmo.

Lo que notamos fue que este algoritmo se haría tal cual está en las animaciones que vimos si se tuviera una lista doblemente enlazada, que no fue nuestro caso. Pero no fue difícil ver que en realidad la idea es recorrer la lista ordenada hasta encontrar la posición correcta para el elemento que queremos insertar, por lo que recorrerla del comienzo hasta este elemento sería nuestra solución.

Para la realización de este algoritmo, la principal dificultad a la cual nos enfrentamos fue la realización de las funciones de mover el elemento a la posición indicada. Para lograr esto, tomamos las representaciones de listas que se muestran en la teoria y estuvimos probando en KolourPaint y en hojas para darnos cuenta como sería el comportamiento de los punteros. Allí vimos necesario definir dos funciones diferentes, *mover al comienzo* y *mover general* y procedimos a implementarlas.

[2] Bien explicado en *glist.h*
[3] titulado *Insertion Sort vs Bubble Sort + Some analysis*

5.3. Merge sort

Este fue el último algoritmo que abordamos y nos basamos en la animación de wikipedia, de donde entendimos que este método deberíamos hacerlo en forma recursiva.
Para este algoritmo recurrimos a más material de internet, otro video de udiprod[4] y de la página Studytonight[5].

Para este algoritmo tambien decidimos ordenar la lista a partir del primer nodo, con esto nos referimos a que la función glist_merge_sort es de esta forma:

```
GList glist_merge_sort (GList lista, Compara funcion){
// Llama al algoritmo de ordenamiento y cambia la GList.
lista.inicio = merge_sort (lista.inicio, funcion);
GNodo *iterador = lista.inicio;
for (; iterador->sig != NULL; iterador = iterador->sig);
lista.final = iterador;
return lista;
}

```

Esto tiene las desventaja que debemos recorrer nuevamente la lista para agregar el final cuando se podría hacer de otra manera, pero no encontramos forma de lograrlo sin complejizar demasiado el código por lo que optamos por esta alternativa. Tomamos la idea de dividir la lista como lo hacen en Studytonight porque nos pareció muy inteligente.

Tuvimos dificultades con la función merge porque la empezamos a hacer de antes de ver ningún video ni material extra que buscamos. Así que nuevamente recurrimos a paint y hojas para representar los punteros y visualizar el comportamiento que queríamos lograr.

5.4. Comparaciones en tiempo de ejecución

Aquí se comparan las velocidades de los algoritmos cambiando el volumen de datos, y utilizando 2 métodos de ordenamiento diferentes[6]:

1000 elementos.

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

5000 elementos.

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

10000 elementos.

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

20000 elementos.

Método de comparación	**Selection**	**Insertion**	**Merge**
Edad	2,0903696	1,525693	0,0071012
LARGO NOMBRE	7,9087686	4,1888458	0,12585

30000 elementos.

Método de comparación	**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.

[4] titulado: *Merge Sort vs Quick Sort*
[5] Todos los links en la bibliografia al final
[6] Ambos métodos de comparación ordenan de *menor a mayor*, para este pequeño análisis

Cuando ya habíamos finalizado el trabajo y nos propusimos estudiar los tiempos de cada algoritmo, nos asustó la diferencia de tiempos entre *Selection sort* y *Insertion sort*, por lo tanto según nos recomendaron en Zulip, decidimos contar la cantidad de comparaciones que se hacían en cada algoritmo. Y al parecer, con un volumen de datos de 30000 elementos, *Selection sort* hace el doble de comparaciones que *Insertion sort*:

Método de comparación	**Selection**	**Insertion**
EDAD	449985000	226779961
LARGO NOMBRE	449985000	241219161

Y cuando las comparamos con sus supuestas complejidades, podríamos decir que *en nuestra implementacion*, Selection ordena en $O(n^2)/2$ mientras que insertion lo hace en $O(n^2)/4$ según los datos que recopilamos.

Llegamos a la conclusión de que el *merge sort* es el algoritmo más rápido, lo cual era esperable ya que es el de menor complejidad. El *selection sort* y el *insertion sort* mantienen tiempos parecidos para volúmenes de datos chicos, pero cuando empezamos a manejar cantidades más significativas comenzamos a ver una notable diferencia, sobre todo si el método de comparación se hace más lento como es el caso de comparar por el menor largo de nombre.

Referencias

[1] **Wikipedia** *Ordenamiento por selección*, Consultado en Abril-Mayo 2020
`https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n`

[2] **Wikipedia** *Ordenamiento por inserción*, Consultado en Abril-Mayo 2020
`https://es.wikipedia.org/wiki/Ordenamiento_por_inserci%C3%B3n`

[3] **Wikipedia** *Ordenamiento por mezcla*, Consultado en Abril-Mayo 2020
`https://es.wikipedia.org/wiki/Ordenamiento_por_mezcla`

[4] **Studytonight** *Using merge sort to sort a linked list*, Consultado en Abril-Mayo 2020
`https://www.studytonight.com/post/using-merge-sort-to-sort-a-linked-list`

[5] **Stack Overflow** *Execution time of C program*, Consultado en Abril-Mayo 2020
`https://stackoverflow.com/questions/5248915/execution-time-of-c-program`

[6] **Youtube** *Insertion Sort vs Bubble Sort + Some analysis*, Consultado en Abril-Mayo 2020
`https://www.youtube.com/watch?v=TZRWRjq2CAg`

[7] **Youtube** *Merge Sort vs Quick Sort*, Consultado en Abril-Mayo 2020
`https://www.youtube.com/watch?v=es2T6KY45cA`