Estructura de datos y de la información
Boletín de problemas - Tema 4

Ficheros de texto secuenciales

1. Escribir un programa que lea un fichero de texto y cree una copia del mismo. El fichero original puede crearse con cualquier editor.

2. Escribir un programa que lea una nueva palabra por teclado y la añada por teclado al final del fichero del ejercicio anterior.

3. Escribir un programa que concatene dos ficheros en un tercero.

4. Escribir un programa que busque una palabra en un fichero de texto y devuelva el número de palabras que hay antes que ella.

5. Escribir un programa que nos diga cuantas veces aparece una palabra leída por teclado en un fichero.

6. Escribir un programa que, a partir de un fichero de texto, cree otro en el que se sustituyan todas las ocurrencias de una palabra dada por otra.

7. Escribir un programa que lea un fichero de texto en castellano de suficiente longitud y calcule el porcentaje de veces que aparece cada letra minúscula. Repetir la ejecución del programa sobre un fichero de texto en inglés y comparar los porcentajes de ambos idiomas. Puedes obtener los ficheros por ejemplo a partir de cualquier página de Internet.

Ficheros binarios de registros

8. Supongamos que queremos guardar en un fichero binario los datos relativos a nuestra colección de cómics. Por cada cómic guardamos un registro con la siguiente estructura:

```
struct Tcomic {
  char codigo[4];
  char titulo[100];
  char autor[100];
  int numero;
};
```

   Como podrás comprobar, todos los registros tienen un tamaño fijo.

   a) Escribir una función que lea los datos de varios cómics por teclado y los guarde en un fichero binario de registros denominado *comics.dat*

b) Escribir una función que dado un código de cómic lo busque en el fichero, escriba su contenido por pantalla y devuelva su desplazamiento en bytes respecto al inicio del fichero. Dado que los registros son de tamaño fijo, y podemos saber cuántos contiene el fichero, si estuviesen ordenados por el campo código, podríamos implementar la operación anterior como una búsqueda binaria.

c) Escribir una función que dado el fichero de cómics construya un segundo fichero *comics.idx* conteniendo la siguiente información por cada cómic:

```
struct Tindice {
  char codigo[4];
  int posicion;
};
```

donde el código coincidirá con el de los cómics almacenados en *comics.dat* y en posición se guardará el desplazamiento del registro correspondiente con respecto al origen en *comics.dat*.

d) Escribir una función que dado un código de cómic lo busque en el fichero *comics.idx* y, utilizando la posición asociada acceda al mismo en *comics.dat* y escriba la información correspondiente por pantalla. De nuevo podemos utilizar el algoritmo de búsqueda dinámica para implementar la operación anterior.

e) Queremos cambiar la información sobre el autor de un determinado cómic. Dado el código del mismo, localizarlo en *comics.dat* utilizando el fichero *comics.idx*. Una vez localizado, solicitar la nueva información y modificar la anterior.

9. Supongamos que queremos guardar la misma información sobre nuestra colección de cómics que en el ejercicio anterior. Sin embargo, en este caso, con el fin de ahorrar espacio en el fichero de datos no la guardaremos en campos de tamaño fijo, sino en campos de tamaño variable conteniendo tan sólo los caracteres necesarios para almacenar el título y autor de cada cómic en particular. Para ello utilizaremos el siguiente tipo para guardar la información de cada cómic:

```
struct Tcomic {
  string codigo, titulo, autor;
  int numero;
};
```

Los distintos registros estarán separados mediante el carácter especial `#`. Dentro de cada registro, los distintos campos estarán separados por el carácter especial `|`.

Pista: Recordar que la función *getline* es capaz de leer textos hasta encontrar un determinado término.

a) Escribir una función que lea por teclado los datos de varios cómics y los guarde en un fichero *comicsv.dat* siguiendo las especificaciones anteriores. Comparar el tamaño del fichero resultante con el del ejercicio anterior si ambos contienen los datos de los mismos cómics.

b) Escribir una función que dado el fichero anterior, lea la información de los cómics que contiene y los escriba por pantalla.

c) Escribir una función que genere un fichero *indexv.idx* con las mismas características que en el ejercicio anterior. En este caso, sin embargo, las posiciones de cada cómic no estarán separadas por un tamaño fijo.

d) Escribir una función que dado el código de un cómic, utilice la búsqueda binaria para encontrarlo en el fichero índice y devuelva su ubicación en el fichero de datos

e) Escribir una función que utilice la del apartado anterior para encontrar y devolver y cómic en el fichero de datos dado su código.

f) Queremos copiar la información del fichero con registros de longitud variable como los de este ejercicio en otro con registros de longitud fija como los del ejercicio anterior. Escribir una función que lleve a cabo este proceso.

10. Supongamos ahora que guardamos la misma información que en el ejercicio anterior. Sin embargo, en esta ocasión cada campo de tipo string va precedido por su tamaño en bytes. No es necesario por tanto ningún carácter especial para separar los campos o los registros.

    a) Escribir una función que lea por teclado los datos de varios cómics y los guarde en un fichero *comicst.dat* siguiendo las especificaciones anteriores.

    b) Escribir una función que dado el fichero anterior, lea la información de los cómics que contiene y los escriba por pantalla.

Tablas de dispersión

11. Implementar una función hash de plegado que pueda aplicarse a palabras de una longitud dada (p.e. 10 caracteres). La función debe devolver un número entero entre 0 y 100. Aplicar la función a distintos apellidos y comprobar el número que devuelve.

12. Queremos guardar en un vector una agenda de teléfonos de 100 personas como máximo. Por cada persona guardaremos la siguiente información:

```
struct Tpersona{
  char apellido1[15];
  char apellido2[15];
  char nombre[15];
  char telefono[9];
};
```

Supondremos por simplicidad que no hay dos personas con el mismo primer apellido.

a) Escribir una función que lea los datos de varias personas por teclado y las guarde en un fichero *agenda.dat*. Los datos de este fichero nos servirán para hacer los apartados posteriores sin tener que volver a teclear todos los datos cada vez.

b) Escribir una función que lea del fichero anterior los datos de una persona. La información se guardará en un vector de personas, en la posición del vector resultante de aplicar la función hash del ejercicio anterior al primer apellido. Las posiciones vacías del vector se indicarán con el campo apellido1 con valor `#`. Ejecutar el programa con diversos nombres e indicar cuando se produce una colisión con un mensaje por pantalla.

c) Mejorar la función del apartado anterior implementando una política de manejo de colisiones mediante direccionamiento abierto. Cuando se llegue al final del vector, se comenzará de nuevo por el principio. Si el vector está lleno, se indicará con un mensaje de error.

d) Escribir una función que dado un apellido, busque a la persona en el vector y escriba sus datos por pantalla si la encuentra, o un mensaje de error en caso contrario. La función devolverá la posición del vector en que se encontró a la persona o -1 en caso de no haberla encontrado.

e) Escribir una función que dado un apellido, busque a la persona en el vector y borre sus datos. Utilizar la función del apartado anterior. Para evitar problemas en búsquedas posteriores se marcará la posición como borrada rellenando el campo apellido1 con el valor `@`.

f) Modificar la función de inserción del tercer apartado para que pueda insertar los nuevos apellidos en posiciones marcadas como borradas.

13. Queremos guardar la misma información que en el ejercicio anterior en un fichero. Sin embargo, en esta ocasión no copiaremos toda la información en la tabla hash, sino tan sólo la necesaria para poder acceder a los registros en el fichero a partir del primer apellido. De este modo, en cada posición de la tabla guardaremos un dato del siguiente tipo:

```
struct Tdato{
  char apellido1[15];
  int pos;
};
```

donde el campo *pos* contendrá la posición en el fichero del registro con los datos de la persona con *apellido1*.

a) Definir el tipo de datos para guardar la tabla hash.

b) Supongamos implementada la función de plegado y la función que lee los registros por teclado y los almacena en posiciones consecutivas del fichero. Implementar una función que lea los datos del fichero y construya la tabla hash asociada. Para ello, por cada registro leído, se aplicará la función de plegado sobre su primer apellido y se guardará en la ubicación obtenida de la tabla hash, el primer apellido y la posición del registro en el fichero.

A la hora de rellenar la tabla hash, aplicaremos la técnica de resolución de colisiones vista en el ejercicio anterior. Además, tendremos también en cuenta la posibilidad de insertar en posiciones marcadas como borradas.

c) Escribir una función que dado el primer apellido de una persona, calcule su ubicación en la tabla hash, obtenga la posición del registro en el fichero, lo lea y lo escriba por pantalla.