Capítulo de Demostración

CAPÍTULO 4
RESOLUCIÓN DE PROBLEMAS

Llegó el momento de resolver problemas integradores. Para esto, estudiaremos diferentes técnicas que nos permitirán abordar una estrategia de solución particular según cuál sea el tipo de problema. Y veremos también cómo clasificar y gestionar eficientemente el acceso a los datos que persisten en los archivos de registros.

Trabajaremos sobre un extenso caso testigo que, dividido en 15 versiones, nos ayudará a comprender, profundizar y aplicar las técnicas mencionadas.

Para probar el funcionamiento de los programas que desarrollamos, presentaremos una herramienta que permite crear archivos de registros, y generar automáticamente todas las funciones auxiliares que requiere el TAD `Coll`: *tToString*, *tFromString*, funciones de inicialización de estructuras, etcétera.

En síntesis, en este capítulo veremos cómo analizar un problema, cómo acceder a los datos que persisten en los archivos, conoceremos una herramienta que aumentará nuestra productividad, y aplicaremos todos los conocimientos adquiridos para resolver las diferentes situaciones que el caso testigo pondrá en evidencia.

4.1. Cómo analizar un problema

Por supuesto que el primer paso para analizar un problema será leer detenidamente el enunciado que lo describe. Como ya hemos comentado en varias oportunidades, difícilmente podamos resolver un problema que no llegamos a comprender.

Debemos identificar cuáles son los datos de entrada, si provienen de archivos o serán ingresados por el usuario a través de la consola, cuáles son los datos de salida que nos piden generar, y hacia dónde debemos dirigirlos: a la consola o a un archivo. También, determinar cuáles son los procesos de transformación que, a partir de los datos de entrada y de contexto, permiten generar los datos de salida.

4.1.1. Contexto, relevamiento y enunciado del problema

Los problemas existen dentro de un determinado contexto. Por ejemplo: la reserva de pasajes para los vuelos de una compañía de aviación, la tabla de posiciones de un torneo de fútbol, o el pago de las infracciones de tránsito labradas por los agentes de una municipalidad. Todos estos son contextos complejos y disímiles entre sí. Poder solucionar cualquier situación problemática inmersa dentro de alguno de estos contextos requeriría, tácitamente, llegar a conocerlo en profundidad.

El estudio del contexto, así como la identificación del problema y su delimitación, es responsabilidad del *analista de sistemas*, quien luego de analizarlo detalladamente debe confeccionar un documento llamado *Informe de relevamiento* (o simplemente *relevamiento*) describiendo el contexto, su operatoria y la situación problemática por resolver. Por supuesto que el informe de relevamiento es un documento extenso y su elaboración requiere tiempo y dedicación.

En nuestro caso, el informe de relevamiento será el enunciado del problema. Si faltaran datos o la descripción de algún proceso pudiese originar dudas o cuestionamientos, debemos aclararlo antes de avanzar con la propuesta de solución. De ningún modo podemos suponer que algo que no está descripto en el enunciado debe hacerse de una u otra manera. Lo que no está escrito en el enunciado no existe; por eso, si es necesario debe ser aclarado antes de continuar.

4.1.2. Datos persistentes

Sea cual fuere el contexto, siempre vamos a encontrar datos persistentes guardados en dispositivos de almacenamiento, que tendremos que procesar para obtener las salidas que el enunciado del problema nos pide generar.

Más adelante, fuera del alcance de este curso, sabremos que la gestión de datos será administrada por motores de bases de datos, como Oracle, DB2, SQL Server y MySQL, entre otros. Pero aquí y ahora, los datos persistirán en archivos de registros de longitud fija, y la gestión de éstos será nuestra propia responsabilidad.

Generalmente, vamos a establecer una distinción entre *archivos de novedades* o *movimientos*, y *archivos de consulta*. Veremos que los resultados que nos piden obtener surgirán luego de leer y procesar los registros del archivo de novedades.

Los archivos de consulta proporcionan información extra que, además de complementar las novedades, aporta parámetros en función de los cuales nuestros algoritmos podrán tomar decisiones.

4.1.2.1. Archivos de consulta

Los archivos de consulta se caracterizan por ser relativamente estáticos, es decir, sus datos no varían; y si lo hacen, sucede muy esporádicamente.

Por ejemplo: en una compañía de aviación, el archivo de CIUDADES es un archivo de consulta. Sus registros siempre serán los mismos, a no ser que la compañía incorpore nuevas rutas, conectando más ciudades.

En un torneo de fútbol, el archivo de EQUIPOS no registrará variaciones, pues generalmente son los mismos equipos los que compiten en los diferentes campeonatos.

4.1.2.2. Archivo de novedades

En el contexto de una compañía aérea, el archivo de RESERVAS, que contiene las reservas que realizaron los clientes para conseguir lugar en los próximos vuelos, es el archivo de movimientos. Su contenido es totalmente dinámico y contiene las novedades que debemos procesar. Las reservas realizadas para los vuelos del presente mes son totalmente diferentes de las reservas del mes que viene: son las novedades.

En el caso del torneo de fútbol, el archivo RESULTADOS, que contiene los resultados de los partidos que se jugaron en una fecha, es el archivo de novedades. Es-

te archivo tendrá registros totalmente diferentes luego de que se juegue cada una de las fechas del torneo, y seguramente será el archivo que tendremos que procesar.

Identificar cuál es el archivo de novedades y cuáles son los de consulta es clave, pues generalmente tendremos que recorrer y procesar el archivo de novedades; y por cada uno de sus registros, acceder a los diferentes archivos de consulta para complementar su contenido con información adicional.

4.1.3. Estrategia

La estrategia es una narración que describe nuestra propuesta de solución. El texto debe ser breve (a lo sumo 10 líneas) y no redundar en detalles técnicos. Simplemente se trata de señalar una dirección, es decir, el plan de acción que implementará el algoritmo para dar solución al problema planteado.

4.2. Tipos de problema

Básicamente, podremos distinguir entre tres tipos de problema, que llamaremos: de *corte de control*, de *apareo de archivos*, o de *procesamiento directo*.

Identificar el tipo de problema es muy importante, pues nos permitirá estructurar un esquema de solución. Una especie de receta de cocina que, si la aplicamos correctamente, nos garantizará desplazarnos adecuadamente a través de los datos, despejando el camino para que nos concentremos en cómo los vamos a procesar.

4.2.1. Problemas de corte de control

En el capítulo 1 estudiamos los problemas de corte de control, caracterizándolos como aquellos en los que el conjunto de datos (ahora archivo de novedades) podía verse dividido en subconjuntos de registros, identificados por tener un valor en común.

Por lo general, este tipo de problemas requiere informar resultados referidos a cada subconjunto, y a la totalidad de los datos. Por ejemplo: el archivo LLAMADAS, que contiene las llamadas realizadas por los abonados de una compañía de telefonía celular, tiene la estructura que vemos a continuación, y sus registros se encuentran ordenados por `idAbonado`, como se observa en la imagen de la derecha.

```
struct Llamada
{
   int idAbonado;
   long long fechaHora;
   int duracion;
   char nroDestino[30];
};
```

idAbonado	fechaHora	...
123	...	
123	...	
123	...	
123	...	
567	...	
567	...	
876	...	
876	...	
1098	...	
1098	...	
1098	...	
1255	...	
1255	...	
:	:	

Figura 4.1, Corte de control

Al estar ordenado por `idAbonado`, todos los registros que describen las llamadas realizadas por un mismo abonado estarán agrupados, estableciendo así subconjuntos de registros.

Si el enunciado pidiera (por ejemplo) informar, por cada abonado, el total de minutos consumidos, estaríamos frente a un típico problema de corte de control. Pues, informaremos un resultado por cada uno de estos subconjuntos.

Adicionalmente, se podría pedir algún dato referido al conjunto completo, como ser: porcentaje de llamadas realizadas (de todos los abonados) discriminado por hora del día. Es decir, de la totalidad de las llamadas, qué proporción se realizó de 0 a 0:59, de 1 a 1:59, etcétera.

4.2.2. Problemas de apareo de archivos

En ocasiones, tendremos que intercalar el contenido de dos o más archivos que se encuentran ordenados por una misma clave. Por ejemplo, en una escuela, los archivos ALUMNOS y NOTAS, ambos ordenados por `legajo` y con las siguientes estructuras.

```
struct Alumno
{
   int legajo;
   char nombre[50];
};
```

```
struct Nota
{
   int legajo;
   int nota;
};
```

Podría suceder que algunos alumnos no hayan rendido examen, en tal situación existirán registros en ALUMNOS que no tendrán correspondencia entre los registros de NOTAS. Otro caso que podría darse es que, por error, hubiera notas asignadas a legajos incorrectos, lo que ocasionaría la existencia de registros en NOTAS sin correspondencia entre los registros de ALUMNOS.

Así, un enunciado característico de apareo de archivos podría requerir:

- Emitir un listado indicando, por cada alumno, su legajo y nota, o "ausente" si el estudiante no rindió el examen.
- Informar también, el listado de las notas que, por error, fueron asignadas a legajos inexistentes.

La figura 4.2. ilustra un ejemplo del contenido de ambos archivos, destacando los casos que acabamos de mencionar: Juan (legajo 10) y Marta (legajo 50) no rindieron examen. Pedro (legajo 20), Pablo (legajo 30) y Carlos (legajo 40) sí lo hicieron. Además, hubo dos calificaciones que fueron asignadas a legajos incorrectos: 35 y 60.

ALUMNOS

nombre	legajo
Juan	10
Pedro	20
Pablo	30
Carlos	40
Marta	50

NOTAS

legajo	nota
20	7
30	7
35	8
40	5
60	6

Figura 4.2. Apareo de archivos

El hecho de que los archivos se encuentren ordenados por la misma clave (legajo) nos permitirá recorrerlos *a la par*, de ahí el nombre *apareo de archivos*.

Comenzaremos leyendo la primera fila de ambos archivos para comparar los valores de sus campos clave (legajo). Llamaremos *a* al valor del legajo contenido en el registro de ALUMNOS y *n* al valor del legajo de NOTAS.

La relación que exista entre *a* y *n* nos permitirá tomar una determinación sobre la fila que leímos en uno u otro archivo; luego de procesar la fila, leeremos la siguiente, y volveremos a evaluar ambos valores para tomar una nueva determinación.

Al inicio, en la primera iteración (según el ejemplo) *a* será 10, y *n* 20.

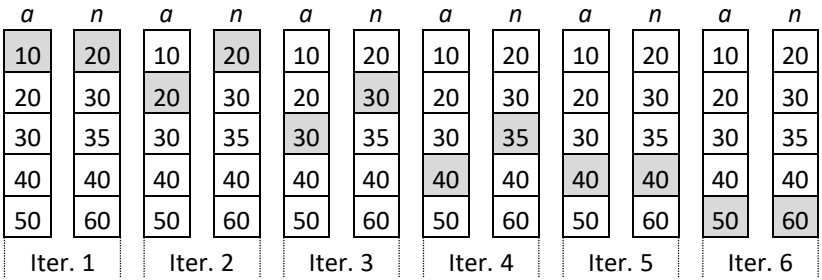

Figura 4.3. Valores de a y n en cada iteración

1. Si *a*<*n*, significa que la fila de ALUMNOS no tiene correspondencia entre las filas de NOTAS, lo que nos permite concluir que el estudiante no se presentó a rendir el examen. En este caso leeremos una nueva fila sobre el archivo cuyo registro hemos podido procesar: ALUMNOS. Ahora, en la segunda iteración, el valor de ambos legajos serán 20.
2. Si *a*==*n*, refleja que el estudiante sí tiene asignada una calificación. En tal caso volveremos a leer ambos archivos, pasando a la tercera iteración que expone un caso similar: *a* y *n* valen 30. Al volver a leer ambos archivos, el valor de *a* será 40, y el de *n*, 35.
3. Si *a*>*n*, sabremos que la calificación reflejada en NOTAS se asignó a un legajo inexistente, tal como vemos en la cuarta iteración. En este caso volvemos a leer un registro del archivo de NOTAS.

Si al finalizar el archivo ALUMNOS aún quedan notas por procesar, todas serán calificaciones mal asignadas. Al contrario, si el primer archivo en llegar al final fuera NOTAS, todas las filas pendientes en ALUMNOS representarán estudiantes que estuvieron ausentes el día del examen.

4.2.3. Problemas de procesamiento directo

Si el problema que vamos a resolver no encaja en ninguno de los casos anteriores, lo consideraremos como un problema de procesamiento directo. Y la estructura de solución simplemente consistirá en recorrer el archivo de novedades procesando uno a uno sus registros.

Más adelante, cuando analicemos el caso testigo, veremos ejemplos de cada uno de estos tipos de problema, incluyendo subtipos y casos particulares.

4.3. Gestión de archivos y persistencia de datos

Cuando contratamos el servicio de una compañía de telefonía celular, o cualquier otro, nos piden nuestros datos personales para registrarnos como clientes: nombres, número de documento, dirección, email, fecha de nacimiento, etcétera.

Los datos que proporcionamos persisten más allá de la duración de los programas que los procesan. Por esto, si posteriormente nos comunicamos con la compañía para realizar algún tipo de trámite o gestión, sólo nos pedirán que indiquemos nuestro número de documento o número de cliente, y eso bastará para que nos identifiquen y puedan tener acceso a los otros datos, que quedaron almacenados (persistiendo) cuando nos registramos como clientes.

Como ya mencionamos, en la vida real (profesional), la gestión de datos la implementan los motores de bases de datos. Pero en este curso será nuestra responsabilidad. Por esto, para garantizar una gestión eficiente, acordaremos respetar una serie de restricciones.

4.3.1. Restricciones

De ningún modo aceptaremos recorrer un mismo archivo más de una vez. Sólo será válido en determinadas circunstancias, que por lo general estarán relacionadas con la actualización de los valores de sus campos.

Si por cada registro del archivo de novedades buscamos información complementaria en los archivos de consulta, tendremos que agilizar el modo en que accederemos a sus registros; por ejemplo, subiéndolos a colecciones en memoria. Esto será válido siempre y cuando el archivo de consulta sea relativamente pequeño. Por ejemplo, el archivo EQUIPOS que mencionamos previamente. ¿Cuántos equipos de fútbol podrían participar de un campeonato? Por contraste, podemos pensar en el archivo ABONADOS, que contiene la información de los abonados de una compañía de telefonía celular. Evidentemente este archivo tendrá varios millones de registros, razón por la cual subirlos a memoria no será una opción válida.

4.3.2. Búsqueda sobre archivos

Analizaremos tres métodos de búsqueda que nos permitirán determinar si dentro de un archivo de consulta existe un registro que contenga un valor especificado; y, por supuesto, poder acceder a sus datos.

4.3.2.1. Subir el archivo a memoria, en una colección de objetos

Este método es muy fácil de implementar, y consiste en disponer de copia de la información del archivo en la memoria de la computadora.

Cada registro será un elemento dentro de una colección de estructuras. Luego, podremos buscar y acceder a cualquiera de los registros de la colección a través de las funciones de búsqueda y acceso, como `collFind`, `collGetAt` u otras que desarrollaremos más adelante.

Trabajaremos sobre la estructura `Ciudad`, que describe los registros del archivo `CIUDADES.dat` antes mencionado. Este archivo formará parte de uno de los problemas propuestos para la práctica.

```
struct Ciudad
{
   int idCiu;
   char descr[20]; // descripcion o nombre de la ciudad
   int millas;
};
```

Subir archivo

Para subir el archivo debemos recorrerlo y agregar cada uno de sus registros a una colección previamente creada. Esto lo haremos dentro de una función que, en este caso, llamaremos `subirCiudades`.

La función crea una colección, abre el archivo, lo recorre, carga sus registros en la colección, cierra el archivo y retorna la colección con todos los datos cargados.

```
Coll<Ciudad> subirCiudades()
{
   // abrimos el archivo
   FILE* f = fopen("CIUDADES.dat","r+b");

   // creamos una coleccion vacia
   Coll<Ciudad> c = coll<Ciudad>();

   // recorremos el archivo
   Ciudad r = read<Ciudad>(f);
```

```
   while( !feof(f) )
   {
      // agregamos el registro a la coleccion
      collAdd<Ciudad>(c,r,ciudadToString);

      r = read<Ciudad>(f);
   }

   // cerramos el archivo y retornamos la coleccion
   fclose(f);
   return c;
}
```

Buscar registro

Invocando a `collFind` podemos buscar entre los registros de la colección y determinar si existe una ciudad identificada por un `idCiu`. Esta búsqueda la vamos a implementar dentro de una función que, en este caso, llamaremos `buscarCiudad`.

```
Ciudad buscarCiudad(Coll<Ciudad> c,int id,bool& encontrado)
{
   int pos = collFind<Ciudad,int>(c
                                  ,id
                                  ,cmpCiudadId
                                  ,ciudadFromString);

   // asignamos true o false al parametro encontrado
   encontrado = pos>=0;

   Ciudad ret;
   if( encontrado )
   {
      ret = collGetAt<Ciudad>(c,pos,ciudadFromString);
   }

   return ret;
}
```

Ejemplo de uso

A continuación accedemos al archivo CIUDADES y realizamos una consulta.

```
// subimos el archivo a memoria
Coll<Ciudad> ciudades = subirCiudades();

// ciudad a buscar
int idCiu = 3;
bool encontrado;
Ciudad ciudad = buscarCiudad(ciudades,idCiu,encontrado);

if( encontrado )
{
   cout << ciudadToString(ciudad) << endl;
}
```

Las funciones de conversión `ciudadToString` y `ciudadFromString` las desarrollaremos automáticamente usando la herramienta *Algoritmos Tools*, que estudiaremos en los próximos apartados. Por esta razón omitiremos codificarlas. El código de la función de comparación `cmpCiudadId`, que utilizamos para invocar a `collFind`, es el siguiente:

```
int cmpCiudadId(Ciudad c,int id)
{
   return c.idCiu-id;
}
```

4.3.2.2. Búsqueda binaria sobre un archivo

Si el archivo de consulta es extenso, probablemente no corresponda subirlo a memoria. Sin embargo, si se encuentra ordenado por el campo de búsqueda, o *campo clave* por el cual queremos buscar, la búsqueda binaria sería una buena opción.

Por ejemplo: el archivo ABONADOS, que contiene la información de los abonados de una compañía de telefonía celular, tiene la siguiente estructura:

```
struct Abonado
{
```

```
   int idAbo;
   char nombre[50];
   char direccion[200];
   int fechaAlta; // aaaammdd
};
```

Buscar registro

Suponiendo que el archivo ABONADOS está ordenado por `idAbo`, desarrollaremos la función `buscarAbonado` dentro de la cual vamos a implementar el algoritmo de la búsqueda binaria.

```
Abonado buscarAbonado(FILE* f,int id,bool& encontrado)
{
   encontrado = false;

   int i = 0;              // primer registro
   int j = fileSize(f)-1; // ultimo registro

   Abonado ret;
   while( i<=j && !encontrado )
   {
      // posicion intermedia entre i y j
      int k = (i+j)/2;

      // accedo al archivo y veo si encuentro lo que busco
      seek<Abonado>(f,k);
      ret = read<Abonado>(f);

      // actualizo los indices
      i = id>a.idAbo?k+1:i;
      j = id<a.idAbo?k-1:j;
      encontrado = i>=j;
   }

   return ret;
}
```

Si el archivo no estuviese ordenado por el campo clave `idAbo`, la búsqueda binaria no será una opción viable.

Ejemplo de uso

En el siguiente programa, buscamos un abonado dentro del archivo `ABONADOS.dat`:

```
// abrimos el archivo
FILE* f = fopen("ABONADOS.dat","r+b");

// id del abonado que vamos a buscar
int id = 6342;

bool encontrado;
Abonado a = buscarAbonado(f,id,encontrado);

if( encontrado )
{
   cout << abonadoToString(a) << endl;
}

fclose(f);
```

4.3.2.3. Indexar un archivo

Si el archivo de consulta es extenso y no está ordenado por el campo clave, nuestra única opción será *indexarlo*. Esta técnica consiste en crear una especie de índice, como si se tratase de un libro; y así vincular el valor de búsqueda de cada registro con la posición que dicho registro ocupa dentro del archivo.

El índice lo implementaremos sobre una colección de estructuras que sólo tendrán dos campos: el campo clave (que generalmente será un identificador), y una posición o número de registro. Trabajaremos sobre el archivo ABONADOS aceptando que no está ordenado por el campo `idAbo`.

Estructura del índice

Necesitaremos disponer de una estructura que vincule una clave con la posición que ocupa, dentro del archivo, el registro que la contiene. La llamaremos `AbonadoIdx`.

```
struct AbonadoIdx
{
   int idAbo; // clave
   int pos;   // posicion
};
```

Indexar

El índice es, en sí mismo, una colección tipo `Coll<AbonadoIdx>`, la cual crearemos recorriendo íntegramente el archivo ABONADOS. La función `indexarAbonados`, cuyo código vemos a continuación, realiza esta tarea.

```
Coll<AbonadoIdx> indexarAbonados(FILE* f)
{
   Coll<AbonadoIdx> c = coll<AbonadoIdx>();

   // nos movemos al inicio del archivo
   seek<Abonado>(f,0);

   Abonado a = read<Abonado>(f);
   while( !feof(f) )
   {
      // creo el elemento con el id y la posicion
      AbonadoIdx elm = {a.idAbo,filePos<Abonado>(f)-1};

      // lo agrego a la coleccion
      collAdd<AbonadoIdx>(c,elm,abonadoIdxToString);

      a = read<Abonado>(f);
   }

   return c;
}
```

Buscar un registro

Ahora, podemos crear una función para buscar un abonado dentro del índice. Y en caso de encontrarlo acceder al archivo para obtener todos sus datos.

```
Abonado buscarAbonadoIdx(FILE* f
                        ,Coll<AbonadoIdx> c
                        ,int id
                        ,bool encontrado)
{
   int pos = collFind<AbonadoIdx,int>(c
                                      ,id
                                      ,cmpAbonadoIdxId
                                      ,abonadoidxFromString);

   encontrado = false;
   Abonado ret;
   if( pos>=0 )
   {
      encontrado = true;

      // obtengo el elemento de la coleccion
      AbonadoIdx elm = collGetAt<AbonadoIdx>(
                                      c
                                     ,pos
                                     ,abonadoIdxFromString);

      // accedo al archivo y veo si encuentro lo que busco
      seek<Abonado>(f,elm.pos);
      ret = read<Abonado>(f);
   }

   return ret;
}
```

Ejemplo de uso

En el siguiente programa, indexamos el archivo ABONADOS, y buscamos entre sus registros la existencia de alguno que concuerde con un *id* especificado.

```
// abrimos el archivo
FILE* f = fopen("ABONADOS.dat","r+b");

// lo indexamos
Coll<AbonadoIdx> idx = indexarAbonados(f);
```

```
int id = 6342;
bool encontrado;

// buscamos un abonado
Abonado a = buscarAbonadoIdx(f,idx,id,encontrado);

if( encontrado )
{
   cout << abonadoToString(a) << endl;
}
```

A esta técnica de indexación, donde por cada clave también guardamos la posición del registro que la contiene, la llamaremos: *indexación indirecta*. Y, como veremos en el siguiente apartado, nos permitirá acceder en orden a los registros del archivo. Cuando analicemos los diferentes versiones del caso testigo, veremos también la técnica de indexación directa.

4.3.3. Ordenar archivos

Los archivos no se ordenan, porque ordenarlos requeriría reescribirlos por completo. Sin embargo, aplicando las técnicas que estudiamos en el apartado anterior, podríamos usar colecciones para ver y/o acceder a los registros de un archivo de consulta como si éstos estuviesen ordenados.

4.3.3.1. Ordenamiento en memoria

Si el archivo es relativamente pequeño podemos subirlo a una colección, ordenarla y acceder en orden a su contenido.

```
Coll<Ciudad> c = subirCiudades();
collSort<Ciudad>(c
                ,cmpCiudadDescr // criterio de ordenamiento
                ,ciudadFromString
                ,ciudadToString);

collReset<Ciudad>(c);
while( collHasNext<Ciudad>(c) )
{
```

```
   Ciudad ciu = collNext<Ciudad>(c,ciudadFromString);
   cout << ciudadToString(ciu) << endl;
}
```

La función `cmpCiudadDescr`, que determina el criterio de ordenamiento que utilizará `collSort`, la programaremos de modo tal que permita establecer un orden de precedencia alfabéticamente ascendente, según la descripción de cada ciudad.

```
int cmpCiudadDescr(Ciudad a,Ciudad b)
{
   string aDescr = a.descr;
   string bDescr = b.descr;
   return aDescr<bDescr?-1:aDescr>bDescr?1:0;
}
```

4.3.3.2. Ordenamiento por indexación

Si el archivo es extenso podemos indexarlo y ordenar su índice. De este modo tendremos acceso directo a sus registros, que aparecerán ordenados en función del criterio de precedencia establecido para el campo clave guardado en la estructura del índice.

```
// abrimos el archivo
FILE* f = fopen("ABONADOS.dat","r+b");

// lo indexamos
Coll<AbonadoIdx> idx = indexarAbonados(f);

// ordenamos el indice
collSort<AbonadoIdx>(idx
                     ,cmpAbonadoId // criterio de ordenamiento
                     ,abonadoIdxFromString
                     ,abonadoIdxToString);

// recorremos el indice y accedemos en
// orden a los registros del archivo
collReset<AbonadoIdx>(idx);
while( collHasNext<AbonadoIdx>(idx) )
{
   AbonadoIdx x=collNext<AbonadoIdx>(idx
                                     ,abonadoIdxFromString);
```

```
   // accedo al archivo
   seek<Abonado>(f,x.pos);
   Abonado a = read<Abonado>(f);

   // muestro
   cout << abonadoToString(a) << endl;
}

fclose(f);
```

La función de comparación que establece el ordenamiento por `idAbo` es:

```
int cmpAbonadoId(Abonado a,Abonado b)
{
   return a.idAbo-b.idAbo;
}
```

4.4. Algoritmos Tools

Algoritmos Tools es una herramienta que genera archivos de registros a partir de datos que cargamos en una planilla Excel. Disponer de los archivos nos permitirá comprobar si los programas que desarrollamos funcionan correctamente o no.

Además, la herramienta permite generar automáticamente todas las funciones auxiliares requeridas para usar el TAD `Coll`: *tToString*, *tFromString*, y las funciones de inicialización para las estructuras, entre otras.

A través de estos QR podremos descargar y aprender a usar Algoritmos Tools.

Descargar Algoritmos Tools *Instalación y uso*

4.5. Caso testigo

Para poner blanco sobre negro y despejar cualquier duda sobre cómo o cuándo aplicar los conocimientos que estudiamos previamente, analizaremos un caso testigo. Un problema que, dividido en 15 versiones, nos permitirá abordar cada situación particular, que podremos resolver mediante la técnica más apropiada. Todas las variantes del problema compartirán el siguiente fragmento de enunciado.

Caso testigo

Enunciado: Una escuela prepara informes estadísticos sobre el rendimiento que tuvieron sus estudiantes en las diferentes asignaturas. Se dispone del archivo `CALIFICACIONES.dat`, cuya estructura de registro es la siguiente:

```
struct Calificacion
{
   int idAsig; // identificador de la asignatura
   int idEst;  // identificador del estudiante
   int calif;  // calificación obtenida durante el bimestre
};
```

Cada uno de los siguientes apartados completa el enunciado anterior proponiendo una variación del problema, y su título describe la técnica que utilizaremos para resolverla.

El código fuente de los programas, así como la planilla Excel con los *scripts* de Algoritmos Tools para generar los archivos de registros con los datos de prueba, están disponibles para ser descargados a través del código QR de la derecha.

Scripts y código fuente

4.5.1. Corte de control (versión 1)

Se dispone del archivo `CALIFICACIONES.dat` cuyos registros se encuentran ordenados (agrupados) por asignatura (`idAsig`). Se pide informar: para cada asignatura, la calificación promedio obtenida por los estudiantes.

Análisis: como los registros del archivo se encuentran ordenados (o agrupados) por `idAsig`, y justamente es por cada asignatura que debemos informar un resultado (la calificación promedio obtenida), estamos ante un problema de corte de control. Esto lo podemos verificar fácilmente analizando un lote de datos de ejemplo, como vemos a la derecha.

CALIFICACIONES.dat

idAsig	idEst	calif	
3	1	7	
3	4	6	
:	:	:	
3	5	9	7.32
6	1	3	
6	4	10	
:	:	:	
6	5	6	6.74
14	1	3	
14	4	10	
:	:	:	
14	5	6	4.98
:	:	:	

Figura 4.4. Lote de datos

La estrategia que nos permite resolver un problema de corte de control consiste en recorrer el archivo de movimientos controlando que la clave de la fila leída (`idAsig`) sea igual a la clave de la fila anterior. Cuando esta igualdad se rompa, sabremos que terminamos de leer y procesar los registros de un subconjunto, y tendremos que informar los resultados que obtuvimos (calificación promedio de la asignatura).

El corte de control lo podemos implementar usando un doble `while` anidado. El primer ciclo controlará que no lleguemos el final del archivo, el segundo controlará que el registro que acabamos de leer pertenezca al mismo grupo (o subconjunto) de registros que el anterior.

En un corte de control siempre identificaremos tres secciones:

1. *Sección de inicialización*: se ubica antes de ingresar al segundo `while` (o ciclo interno), y se ejecuta justo antes de comenzar a procesar los registros de cada subconjunto. Esto la convierte en el lugar indicado para dar valor inicial a las variables que utilizaremos para procesar dichos registros; que en nuestro serán: un contador (`cont`) y un acumulador (`acum`). Y, por supuesto, una variable de control para controlar que el registro leído corresponda al mismo subconjunto que estamos procesando (`idAsigAnt`).
2. *Sección de resultados*: se ubica entre el cierre del ciclo interno y el cierre del ciclo exterior, y se ejecuta cada vez que hayamos finalizado de leer y procesar los registros de un subconjunto. Por lo tanto, esta es la sección donde debemos mostrar los resultados relativos al subconjunto, que en nuestro caso son: la calificación promedio que obtuvieron los estudiantes para la asignatura cuyos registros acabamos de procesar.

3. *Sección de procesamiento*: se ubica dentro del ciclo interno, el cual nos asegura que cada registro leído pertenece al mismo subconjunto que el registro anterior. Por esto, en aquí donde procesaremos los registros del subconjunto.

```
int main()
{
   // abrimos el archivo
   FILE* f = fopen("CALIFICACIONES.dat","r+b");

   // leemos el primer registro del archivo
   Calificacion reg = read<Calificacion>(f);

   // controlamos que no llegue el fin del archivo
   while( !feof(f) )
   {
      // ***SECCION DE INICIALIZACION***
      int cont = 0;
      int acum = 0;

      // guardamos la asignatura anterior
      int idAsigAnt = reg.idAsig;
      while( !feof(f) && idAsigAnt == reg.idAsig )
      {
         // ***SECCION DE PROCESAMIENTO***
         cont++;
         acum += reg.calif;

         // leemos el siguiente registro del archivo
         reg = read<Calificacion>(f);
      }

      // ***SECCION DE RESULTADOS***
      // mostramos la calificacion promedio obtenida
      // para la asignatura cuyos registros acabamos
      // de procesar
      double prom = acum/(double)cont;
      cout << idAsigAnt << ": " << prom << endl;
   }

   // cerramos el archivo
   fclose(f);
   return 0;
}
```

4.5.2. Corte de control, con salida bufferizada (versión 2)

Se dispone del archivo `CALIFICACIONES.dat`, cuya estructura recordamos a continuación, ordenado por `idAsig`. Se pide informar, por cada asignatura, la lista de estudiantes aprobados (`calif>=4`), ordenada decrecientemente por calificación.

```
struct Calificacion
{
   int idAsig; // identificador de la asignatura
   int idEst;  // identificador del estudiante
   int calif;  // calificación obtenida durante el bimestre
};
```

Análisis: Si no fuera por el orden en que nos piden que aparezcan los estudiantes dentro de cada asignatura (decrecientemente según su calificación), el problema no presentaría ninguna complicación adicional, y lo resolveríamos recorriendo el archivo con corte de control por `idAsig`, mostrando por pantalla cada uno de los registros, tal que su valor en `calif` sea mayor o igual a 4 (cuatro).

Por supuesto que esta solución no resuelve el problema aquí planteado, pues los estudiantes aparecerán en el listado, en el mismo orden en que sus registros aparecen dentro del archivo; no en el orden solicitado.

La solución será no mostrar por pantalla los estudiantes aprobados. En lugar de esto, los guardaremos en una colección que, luego, en la sección de resultados, podremos ordenar, iterar y mostrar.

```
int main()
{
   // abrimos el archivo
   FILE* f = fopen("CALIFICACIONES.dat","r+b");

   // leemos el primer registro del archivo
   Calificacion reg = read<Calificacion>(f);

   // controlamos que no llegue el fin del archivo
   while( !feof(f) )
   {
```

```
      // SECCION DE INICIALIZACION
      Coll<Calificacion> buff = coll<Calificacion>();

      // guardamos la asignatura anterior
      int idAsigAnt = reg.idAsig;
      while( !feof(f) && idAsigAnt == reg.idAsig )
      {
         // SECCION DE PROCESAMIENTO
         if( reg.calif>=4 )
         {
            collAdd<Calificacion>(buff
                                  ,reg
                                  ,calificacionToString);
         }

         // leemos el siguiente registro del archivo
         reg = read<Calificacion>(f);
      }

      // SECCION DE RESULTADOS
      mostrarEstudiantesAprobados(idAsigAnt,buff);
   }

   // cerramos el archivo
   fclose(f);
   return 0;
}
```

A la colección `buff` la llamaremos *buffer*. Pues la utilizamos como almacenamiento temporal para guardar los datos a medida fueron llegando. Datos que posteriormente ordenamos, y mostramos en el orden requerido por el enunciado. Diremos que *bufferizamos* las calificaciones aprobadas.

Veamos la función que ordena y muestra la colección.

```
void mostrarEstudiantesAprobados(int idAsig
                                 ,Coll<Calificacion> buff)
{
   // mostramos la asignatura
   cout << "Asignatura: " << idAsig << endl;
```

```
   // ordenamos la coleccion
   collSort<Calificacion>(buff
                          ,cmpCalificacion
                          ,calificacionFromString
                          ,calificacionToString);

   // la iteramos
   collReset<Calificacion>(buff);
   while( collHasNext<Calificacion>(buff) )
   {
      Calificacion elm = collNext<Calificacion>(
                                    buff
                                   ,calificacionFromString);

      // mostramos los estudiantes aprobados
      cout << elm.idEst << ", " << elm.calif << endl;
   }
}
```

Finalmente, el código de la función `cmpCalificacion`, que establece el criterio de ordenamiento que utlizará `collSort`, es el siguiente:

```
int cmpCalificacion(Calificacion a,Calificacion b)
{
   return b.calif-a.calif;
}
```

4.5.3. Descubrimiento (versión 3)

Se dispone del archivo `CALIFICACIONES.dat`, cuyos registros no respetan ningún orden en particular. Se pide informar: para cada asignatura, la calificación promedio obtenida por los estudiantes.

Recordemos la estructura de los registros del archivo:

```
struct Calificacion
{
   int idAsig;
```

```
   int idEst;
   int calif;
};
```

Análisis: como los registros del archivo no están ordenados, recorrerlo con corte de control ya no es una opción.

Supongamos que sólo nos piden obtener la calificación promedio para la asignatura 3 (`idAsig`=3). De ser así, podríamos recorrer el archivo verificando por cada registro si corresponde a dicha asignatura. En tal caso, necesitaremos disponer de un contador y un acumulador para contar una calificación más, y acumular dicha calificación.

Luego de haber recorrido todo el archivo, el promedio de las calificaciones de la asignatura 3 lo calcularemos como el cociente entre el acumulador y el contador.

Supongamos ahora que nos piden obtener las calificaciones promedio de las asignaturas 3 y 6. Siendo así, la estrategia anterior continúa siendo válida. Sólo que necesitaremos disponer de dos contadores y dos acumuladores: uno para cada una de las asignaturas cuyas calificaciones debemos procesar. Además, por cada registro leído tendremos que verificar si corresponde a alguna de estas dos asignaturas. Si corresponde a la asignatura 3, contaremos y acumularemos en el contador y acumulador destinados para esta asignatura. Si el registro corresponde a la asignatura 6, haremos lo propio con su contador y acumulador. Si el registro no corresponde a ninguna de estas asignaturas, lo ignoraremos.

CALIFICACIONES.dat

idAsig	idEst	calif
6	5	6
3	4	6
:	:	:
6	1	3
14	4	10
14	5	6
:	:	:
3	1	7
6	4	10
14	1	3
:	:	:
3	5	9
:	:	:

Figura 4.5. Lote de datos

Según el razonamiento anterior, necesitaremos un contador y un acumulador por cada asignatura cuyo promedio de calificaciones queramos procesar.

El enunciado del problema nos pide calcular y mostrar la calificación promedio de cada una de las asignaturas, y no sabemos cuáles ni cuántas son. La solución será utilizar una colección de contadores y acumuladores. Un par {`cont`,`acum`} por cada asignatura. Es decir: una colección de estructuras.

```
struct Estad // estadistica
{
   int id;    // identificador
   int cont; // contador
   int acum; // acumulador
};
```

La estructura `Estad` contiene también el campo `id`, pues será necesario para identificar a qué asignatura corresponden el contador y el acumulador. Esto nos permitirá apoyar la estrategia de solución en una colección de estructuras `Estad`, como la que declaramos a continuación:

```
Coll<Estad> collEstad = coll<Estad>();
```

Ya mencionamos que, a priori, no conocemos cuáles ni cuántas asignaturas diferentes existen. Las iremos *descubriendo* a medida que vamos leyendo cada uno de los registros del archivo.

La estrategia será la siguiente:

1. Recorremos íntegramente el archivo de calificaciones.
2. Por cada registro leído utilizamos `idAsig` para buscar en `collEstad`.
    a. Si no existe en `collEstad` ningún elemento vinculado a `idAsig`, lo agregamos, inicializándolo con `idAsig` y cero en `cont` y `acum`.
3. Ahora que disponemos del contador y el acumulador que corresponden a la asignatura `idAsig` (ya sea porque lo encontramos, o porque lo acabamos de agregar), incrementamos `cont` y acumulamos `calif` en `acum`.
4. Finalmente, `collEstad` tendrá tantos contadores y acumuladores como asignaturas diferentes existen. Para mostrar los resultados recorreremos la colección y, por cada iteración:
    a. Calculamos el promedio.
    b. Mostramos el resultado.

A la técnica de buscar si existe un elemento dentro de una colección y agregarlo en caso de que no exista la llamaremos *descubrimiento*, y la veremos implementada en la función `descubrirElemento`. que invocaremos dentro de `main`.

Veamos el código del programa principal que resuelve esta versión.

```
int main()
{
   FILE* f = fopen("CALIFICACIONES.dat","r+b");

   // coleccion de contadores y acumuladores
   Coll<Estad> collEstad = coll<Estad>();

   // leemos el primer registro del archivo
   Calificacion reg = read<Calificacion>(f);
   while( !feof(f) )
   {
      // buscamos (y eventualmente agragamos) en la coll
      int pos = descubrirElemento(collEstad,reg.idAsig);

      // procesamos
      Estad elm = collGetAt<Estad>(collEstad
                                   ,pos
                                   ,estadFromString);

      elm.cont++;
      elm.acum += reg.calif;
      collSetAt<Estad>(collEstad,elm,pos,estadToString);

      // leemos el siguiente registro
      reg = read<Calificacion>(f);
   }

   // resultados
   mostrarResultados(collEstad);

   fclose(f);
   return 0;
}
```

La función `descubrirElemento` lleva a cabo el descubrimiento de la asignatura identificada por `idAsig`. Busca dentro de `collEstad` un elemento cuyo `id` coindida con `idAsig`. Si no lo encuentra, lo agrega. En cualquier caso retorna la posición del elemento que representa a la asignatura en cuestión.

```
int descubrirElemento(Coll<Estad>& collEstad,int id)
{
   // buscamos
   int pos = collFind<Estad,int>(collEstad
                                 ,id
                                 ,cmpEstadId
                                 ,estadFromString);

   // si no se encontro el elemento, lo agregamos
   if( pos<0 )
   {
      Estad x = estad(id,0,0);
      pos = collAdd<Estad>(collEstad,x,estadToString);
   }

   return pos;
}
```

La función `cmpEstadId` compara un elemento `Estad` con un `idAsig`.

```
int cmpEstadId(Estad e,int id)
{
   return e.id-id;
}
```

`mostrarResultados` muestra el listado que solicita el enunciado del problema. La lógica es tan simple que se limita a iterar la colección, calcular un promedio por cada iteración, y mostrar el resultado en la pantalla.

```
void mostrarResultados(Coll<Estad> c)
{
   // iteramos
   collReset<Estad>(c);
   while( collHasNext<Estad>(c) )
   {
      Estad e = collNext<Estad>(c,estadFromString);
      double prom = e.acum/(double)e.cont;
      cout << e.id << ": " << prom << endl;
```

```
    }
}
```

4.5.4. Archivo de consultas en memoria (versión 4)

Se dispone de los archivos `CALIFICACIONES.dat` y `ASIGNATURAS.dat`, ambos sin orden, y con las estructuras de registro que se describen a continuación:

CALIFICACIONES

```
struct Calificacion
{
   int idAsig;
   int idEst;
   int calif;
};
```

ASIGNATURAS

```
struct Asignatura
{
   int idAsig;
   char nomAsig[30];
   char maestroACargo[50];
};
```

Se pide: emitir dos listados, ambos detallando la calificación promedio obtenida por los estudiantes de cada asignatura. El primero debe estar ordenado alfabéticamente por asignatura (`nomAsig`), tal como se muestra a la izquierda. El segundo listado debe estar ordenado decrecientemente según la calificación promedio, tal como se ilustra a la derecha.

Asignatura	*Promedio*
Arte	7.88
Geografía	6.09
Historia	8.23
Matemáticas	7.34
:	:

Asignatura	*Promedio*
Historia	8.23
Arte	7.88
Matemáticas	7.34
Geografía	6.09
:	:

Análisis: comenzaremos por identificar cuál es el archivo de novedades y cuál es el de consulta. En este caso, CALIFICACIONES es el archivo que trae las novedades, pues contiene las calificaciones que obtuvieron los estudiantes durante el presente período. En cambio, el archivo de ASIGNATURAS contiene información que complementa los datos del archivo de novedades. Por ejemplo, conociendo a el `idAsig` de una asignatura podremos conocer su nombre y quién es el maestro a cargo.

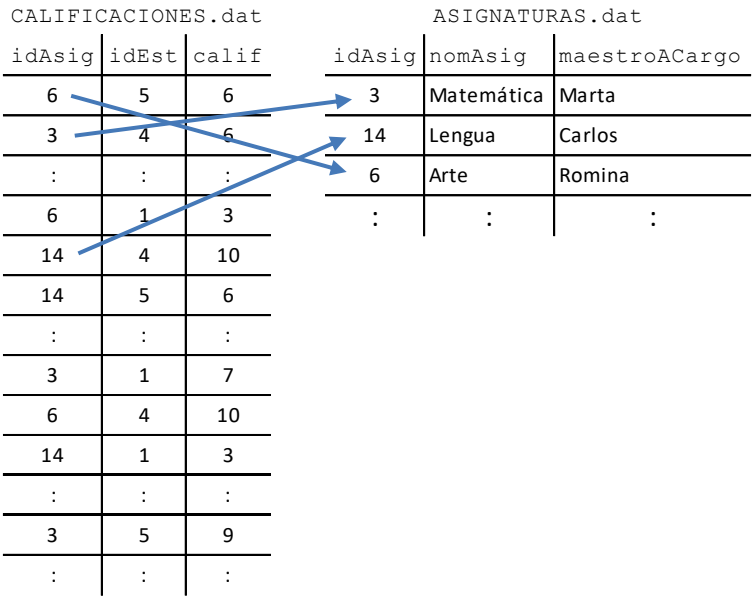

Figura 4.6. Lote de datos complementados por el archivo de consulta

Al disponer del archivo de asignaturas ya no tendremos la necesidad de descubrirlas (mientras vamos procesando las novedades), pues todas las asignaturas que existen se encuentran descriptas en dicho archivo.

Como el archivo de consultas es relativamente pequeño (¿cuántas asignaturas diferentes se podrían dictar en una escuela?), lo mantendremos en memoria, en una colección; y veremos cómo hacer para que cada uno de sus elementos (cada asignatura) tenga asociados un contador y un acumulador.

RAsignatura				
asig			estad	
idAsig	nomAsig	maestroACargo	cont	acum
3	Matematica	Marta	0	0
14	Lengua	Carlos	0	0
6	Arte	Romina	0	0
:	:	:	:	:

Figura 4.7. Estructura de datos de la colección

La imagen describe una colección de estructuras `RAsignatura`, cuyo código veremos enseguida, y deja en evidencia que los elementos de la colección serán pares: {`Asignatura`, `Estad`}.

```
struct RAsignatura
{
   Asignatura asig;
   Estad estad;
};
```

```
struct Estad
{
   int cont; // contador
   int acum; // acumulador
};
```

Observemos que ya no necesitamos el campo `id` en la estructura `Estad`, pues un elemento tipo `RAsignatura` tiene dicho *id* en el campo `asig.idAsig`.

A esta técnica, que permite agregarle campos a una estructura mediante la creación de otra estructura de mayor nivel, la llamaremos *envoltorio* o *wrapping*. Entonces diremos que `RAsignatura` es un *wrapper* de `Asignatura`.

La siguiente función, que invocaremos en el programa principal, lee el archivo de consulta y lo sube a una colección en memoria.

```
Coll<RAsignatura> subirAsignaturas()
{
   // coleccion que contendra todas las asignaturas
   Coll<RAsignatura> ret = coll<RAsignatura>();

   FILE* f = fopen("ASIGNATURAS.dat","r+b");

   // leemos el primer registro del archivo
   Asignatura reg = read<Asignatura>(f);

   while( !feof(f) )
   {
      // creamos un RAsignatura con las funciones de inic
      RAsignatura ra = rAsignatura(reg,estad(0,0));

      // lo agregamos a la coleccion
      collAdd<RAsignatura>(ret,ra,rAsignaturaToString);

      // leemos el siguiente registro
      reg = read<Asignatura>(f);
```

```
   }

   fclose(f);
   return ret;
}
```

La estrategia para resolver el problema será la siguiente:

1. Subimos el archivo de consultas a memoria, usando la función anterior.
2. Recorremos el archivo de novedades.
3. Por cada registro, lo buscamos en la colección, incrementamos su contador y acumulamos la calificación.

Luego de esto, en la colección tendremos todos los datos necesarios para mostrar los dos listados solicitados, que sólo difieren en cómo debemos ordenar la colección antes de iterarla. Por esto, en la función `mostrarResultados` recibiremos, como parámetro, una función de comparación, de modo que al invocarla podamos indicar diferentes criterios de comparación (y por lo tanto, de ordenamiento).

```
void mostrarResultados(
             Coll<RAsignatura> c
            ,int cmpRAsignatura(RAsignatura,RAsignatura))
{
   // ordenamos la coleccion
   collSort<RAsignatura>(c
                        ,cmpRAsignatura
                        ,rAsignaturaFromString
                        ,rAsignaturaToString);

   // iteramos
   collReset<RAsignatura>(c);
   while( collHasNext<RAsignatura>(c) )
   {
      RAsignatura ra = collNext<RAsignatura>(
                                      c
                                     ,rAsignaturaFromString);

      // calculamos el promedio y mostramos
      double prom = ra.estad.acum/(double)ra.estad.cont;
```

```
      string sNomAsig = ra.asig.nomAsig;
      cout << sNomAsig << ": " << prom << endl;
   }
}
```

Las funciones que establecen los criterios de ordenamiento que requiere el enunciado son: `cmpRAsigAlfabetico`, que indica que una asignatura precede a otra si su nombre es alfabéticamente menor, y `cmpRAsigPromedio`, que calcula la calificación promedio de ambas asignaturas, y determina que una precede a la otra si dicho promedio es menor.

Veamos el código de ambas funciones.

```
// funcion de comparacion, compara alfabeticamente
int cmpRAsigAlfabetico(RAsignatura a,RAsignatura b)
{
   string sA = a.asig.nomAsig;
   string sB = b.asig.nomAsig;
   return sA<sB?-1:sA>sB?1:0;
}

// funcion de comparacion, compara por promedio descendente
int cmpRAsigPromedio(RAsignatura a,RAsignatura b)
{
   double pA = a.estad.acum/(double)a.estad.cont;
   double pB = b.estad.acum/(double)b.estad.cont;
   return pA>pB?-1:pA<pB?1:0;
}
```

Ahora sí, podemos ver el programa principal.

```
int main()
{
   // subimos el archivo de consultas a memoria
   Coll<RAsignatura> collAsig = subirAsignaturas();
   FILE* f = fopen("CALIFICACIONES.dat","r+b");
```

```
   // leemos el primer registro del archivo
   Calificacion reg = read<Calificacion>(f);
   while( !feof(f) )
   {
      // buscamos la asignatura en la coleccion
      int pos = collFind<RAsignatura,int>(
                                    collAsig
                                   ,reg.idAsig
                                   ,cmpRAsigId
                                   ,rAsignaturaFromString);

      // obtenemos el elemento encontrado en la posicion pos
      RAsignatura elm = collGetAt<RAsignatura>(
                                    collAsig
                                   ,pos
                                   ,rAsignaturaFromString);

      // procesamos
      elm.estad.cont++;
      elm.estad.acum += reg.calif;

      // aplicamos los cambios en la coleccion
      collSetAt<RAsignatura>(collAsig
                            ,elm
                            ,pos
                            ,rAsignaturaToString);

      // leemos el siguiente registro
      reg = read<Calificacion>(f);
   }

   // mostramos el listado 1 (ordenado alfabeticamente)
   mostrarResultados(collAsig,cmpRAsigAlfabetico);

   // mostramos el listado 2 (ordenado por prom descend)
   mostrarResultados(collAsig,cmpRAsigPromedio);

   fclose(f);
   return 0;
}
```

La función `cmpRAsigId`, que le pasamos a `collFind`, es la siguiente:

```
int cmpRAsigId(RAsignatura ra,int id)
{
   return ra.asig.idAsig-id;
}
```

4.5.5. Descubrimiento, otro caso (versión 5)

Se dispone del archivo `CALIFICACIONES.dat`, sin orden y con la estructura de registro que vemos más abajo, que describe las calificaciones que obtuvieron los estudiantes al cursar las diferentes asignaturas. Se pide emitir un listado detallando, para cada estudiante, cuál fue su calificación promedio.

```
struct Calificacion
{
   int idAsig; // asignatura
   int idEst;  // estudiante
   int calif;  // calificacion
};
```

Análisis: Si el archivo de calificaciones estuviese ordenado por `idEst`, podríamos resolver el problema mediante un corte de control. Sin embargo, como el archivo no está ordenado (y las restricciones nos impiden ordenarlo) debemos implementar una solución similar a la que utilizamos para resolver la versión 3.

En este caso tendremos que descubrir los diferentes `idEst` a medida que vayamos leyendo los registros del archivo de novedades. Como necesitaremos un contador y un acumulador por cada estudiante, reutilizaremos la estructura `Estad`, cuyo código recordamos a continuación.

```
struct Estad
{
   int id;
   int cont; // contador
   int acum; // acumulador
};
```

Observemos que volvimos a agregar el campo `id` a la estructura `Estad`. Esto se debe a la necesidad de identificar al estudiante, cuyas calificaciones estaremos contando y acumulando en los campos `cont` y `acum` respectivamente.

La solución es prácticamente idéntica a la que implementamos para resolver la tercera versión del problema. La única diferencia es que aquí utilizaremos `reg.idEst` en lugar de `reg.idAsig`, para buscar dentro de la colección.

```
int main()
{
   FILE* f = fopen("CALIFICACIONES.dat","r+b");

   // coleccion de contadores y acumuladores
   Coll<Estad> collEstad = coll<Estad>();

   // leemos el primer registro del archivo
   Calificacion reg = read<Calificacion>(f);
   while( !feof(f) )
   {
      // buscamos (y eventualmente agragamos) en la coll
      int pos = buscarElemento(collEstad,reg.idEst);

      // procesamos
      Estad elm = collGetAt<Estad>(collEstad
                                   ,pos
                                   ,estadFromString);

      elm.cont++;
      elm.acum += reg.calif;
      collSetAt<Estad>(collEstad,elm,pos,estadToString);

      // leemos el siguiente registro
      reg = read<Calificacion>(f);
   }

   // resultados
   mostrarResultados(collEstad);

   fclose(f);
   return 0;
}
```

La función `buscarElemento` busca dentro de `collEstad` un elemento cuyo `id` coincida con `reg.idEst`. Si no lo encuentra, lo agrega. Finalmente retorna la posición del elemento buscado, sea que lo encontró o lo agregó.

La función `mostrarResultados` itera la colección. Por cada iteración, calcula la calificación promedio obtenida del estudiante y la muestra por pantalla.

4.5.6. Actualizar registros (versión 6)

Se dispone de los archivos `CALIFICACIONES.dat` y `ASIGNATURAS.dat`, ambos sin orden y con las estructuras de registro que vemos a continuación.

CALIFICACIONES

```
struct Calificacion
{
   int idAsig;
   int idEst;
   int calif;
};
```

ASIGNATURAS

```
struct Asignatura
{
   int idAsig;
   char nomAsig[30];
   char maestroACargo[50];
   double califProm;
};
```

Se pide:

1. Por cada asignatura, mostrar la calificación promedio anterior, y la nueva calificación promedio que surge de procesar las calificaciones del archivo.

Asignatura	*Promedio anterior*	*Promedio actual*
Arte	7.88	6.25
Geografía	6.09	6,57
Historia	8.23	7.34
Matemáticas	7.34	5.48
:	:	:

2. Actualizar el archivo de asignaturas con los nuevos promedios calculados.

Análisis: La estrategia de solución que utilizaremos para resolver esta versión del problema es idéntica a la que diseñamos para la versión 4, y se basa en mantener en memoria en una colección de `RAsignatura`, con los registros del archivo

de consulta, más un contador y un acumulador para recopilar los datos estadísticos a medida que vayamos leyendo los registros del archivo de novedades.

Recordemos las estructuras `RAsignatura` y `Estad`.

```
struct RAsignatura
{
   Asignatura asig;
   Estad estad;
};
```

```
struct Estad
{
   int cont; // contador
   int acum; // acumulador
};
```

Por cada registro leído, buscamos `idAsig` en `collAsig` y actualizamos su contador y acumulador. Luego de procesar todo el archivo de novedades tendremos, por cada asignatura, el promedio anterior (en el campo `califProm`), y los datos necesarios para calcular el nuevo promedio (en los campos `cont` y `acum`).

La única diferencia que existe entre el listado del punto 1 de esta versión y la solución que desarrollamos para la cuarta versión del problema, es que aquí no tendremos que ordenar la colección. Por esta razón, `mostrarResultados` ya no necesitará recibir la función de comparación entre sus parámetros.

```
void mostrarResultados(Coll<RAsignatura> c)
{
   // iteramos
   collReset<RAsignatura>(c);
   while( collHasNext<RAsignatura>(c) )
   {
      RAsignatura ra = collNext<RAsignatura>(
                                    c
                                   ,rAsignaturaFromString);

      // promedios anterior y actual
      double promAct = ra.estad.acum/(double)ra.estad.cont;
      double promAnt = ra.asig.califProm;
      string sNomAsig = ra.asig.nomAsig;

      cout<<sNomAsig<<": "<<promAnt<<", "<<promAct << endl;
   }
}
```

Demás está decir que al invocar a `mostrarResultados` en el programa principal no tendremos que pasarle ninguna función como argumento.

El motivo por el cual estamos analizando esta versión del problema no es el punto 1, sino el punto 2; que nos pide actualizar el campo `califProm` del archivo de asignaturas. Dicha actualización la lleva a cabo la función `actualizar`, que invocaremos al final del programa principal. Su código lo veremos más adelante.

```
int main()
{
   // subimos el archivo de consultas a memoria
   Coll<RAsignatura> collAsig = subirAsignaturas();

   FILE* f = fopen("CALIFICACIONES.dat","r+b");

   // leemos el primer registro del archivo
   Calificacion reg = read<Calificacion>(f);
   while( !feof(f) )
   {
      // buscamos la asignatura en la coleccion
      int pos = collFind<RAsignatura,int>(
                                    collAsig
                                   ,reg.idAsig
                                   ,cmpRAsigId
                                   ,rAsignaturaFromString);

      // obtenemos el elemento encontrado en la posicion pos
      RAsignatura elm = collGetAt<RAsignatura>(
                                    collAsig
                                   ,pos
                                   ,rAsignaturaFromString);

      // procesamos
      elm.estad.cont++;
      elm.estad.acum += reg.calif;

      // aplicamos los cambios en la coleccion
      collSetAt<RAsignatura>(collAsig
                            ,elm
                            ,pos
                            ,rAsignaturaToString);
```

```
      // leemos el siguiente registro
      reg = read<Calificacion>(f);
   }

   // mostramos el listado requerido en el punto 1
   mostrarResultados(collAsig);

   // actualizamos el archivo de consultas
   actualizar(collAsig);

   fclose(f);
   return 0;
}
```

Ahora veremos cómo usar los datos de la colección `collAsig`, para actualizar el archivo de consultas. La estrategia será la siguiente:

1. Recorreremos el archivo de consultas.
2. Por cada registro leído, buscamos por `idAsig` en la colección `collAsig`.
3. Calculamos el nuevo valor promedio como: la suma del promedio anterior más el nuevo promedio dividido dos.
4. Actualizamos el registro del archivo.

Debemos tener presente que luego de leer un registro del archivo, su indicador de posición quedará apuntando al siguiente registro. Por tal razón será necesario retroceder una posición antes de grabar los datos actualizados. De no hacerlo, estaremos modificando el siguiente registro, no el que queremos modificar.

```
void actualizar(Coll<RAsignatura> c)
{
   // "r+b" permite modificar registros y agregar al final
   FILE* f = fopen("ASIGNATURAS.dat","r+b");

   Asignatura reg = read<Asignatura>(f);
   while( !feof(f) )
   {
      int pos = collFind<RAsignatura,int>(
                                    collAsig
                                   ,reg.idAsig
                                   ,cmpRAsigId
                                   ,rAsignaturaFromString);
```

```
      // obtenemos el elemento encontrado en la posicion pos
      RAsignatura elm = collGetAt<RAsignatura>(
                                      collAsig
                                     ,pos
                                     ,rAsignaturaFromString);

      double promAnt = elm.asig.califProm;
      double promAct = elm.estad.acum/(double)elm.estad.cont;
      double promNuevo = (promAnt+promAct)/2;

      // actualizamos el registro
      reg.califProm = promNuevo;

      // grabamos
      seek<Asignatura>(f,filePos<Asignatura>(f)-1);
      write<Asignatura>(f,reg);

      // leemos el siguiente registro
      reg = read<Asignatura>(f);
   }

   fclose(f);
}
```

Otra estrategia para implementar la función `actualizar` consistiría en borrar el archivo de asignaturas y reescribirlo con los registros de la colección. Si optamos por esta solución, los pasos que deberíamos ejecutar serían los siguientes:

1. Abrir el archivo con "w+b", para borrarlo y volver a crearlo vacío.
2. Iterar la colección, y por cada elemento:
    a. Calcular el nuevo promedio.
    b. Grabar el registro en el archivo.

Veamos esta implementación de la función `actualizar`.

```
void actualizar(Coll<RAsignatura> c)
{
   // "r+w" borra el archivo y lo crea vacio
   FILE* f = fopen("ASIGNATURAS","w+b");
```

```
   collReset<RAsignatura>(c);
   while( collHasNext<RAsignatura>(c) )
   {
      RAsignatura elm = collNext<RAsignatura>(
                                        c
                                       ,rAsignaturaFromString);

      double promAnt = elm.asig.califProm;
      double promAct = elm.estad.acum/(double)elm.estad.cont;
      double promNuevo = promAnt+promAct/2;

      // actualizamos el registro
      elm.asig.califProm = promNuevo;

      // grabamos
      write<Asignatura>(f,elm.asig);
   }

   fclose(f);
}
```

4.5.7. Colección de colecciones (versión 7)

Se dispone de los archivos `CALIFICACIONES.dat` y `ASIGNATURAS.dat`, ambos sin orden, y con las estructuras que vemos a continuación:

CALIFICACIONES

```
struct Calificacion
{
   int idAsig;
   int idEst;
   int calif;
};
```

ASIGNATURAS

```
struct Asignatura
{
   int idAsig;
   char nomAsig[30];
   char maestroACargo[50];
};
```

Se pide mostrar un listado indicando, por cada asignatura, el maestro a cargo y la lista de estudiantes cuya calificación es inferior a 4 (cuatro).

Por ejemplo:

Asignatura	*Maestro*	*Estudiantes aplazados*
Arte	Juan	2, 6, 9, 10, 15
Geografía	Pedro	1, 2, 8, 14
Historia	Juan	3, 5, 6, 10, 17, 23
Matemáticas	Marta	2, 4, 5, 14, 15, 16, 19, 21, 23
:	:	

Análisis: como ambos archivos se encuentran desordenados podemos descartar cualquier posibilidad de corte de control. Apoyaremos la estrategia de solución en una estructura de datos compuesta por una colección de estructuras, cada una de las cuales tendrá una colección de `idEst`; es decir: un `Coll<int>`.

```
struct RAsignatura
{
   Asignatura asig;
   Coll<int> collEst;
};
```

Subiremos el archivo de consulta a un `Coll<RAsignatura>` y recorreremos el archivo de novedades. Por cada registro que leamos, si `calif<4` agregaremos el valor de `idEst` al final de la colección de estudiantes aplazados correspondiente a la asignatura `idAsig`.

Finalmente, recorreremos la colección de asignaturas, y por cada una, recorreremos la colección de estudiantes aplazados para mostrarlos por pantalla.

```
int main()
{
   // coleccion de asignaturas
   Coll<RAsignatura> collAsig = subirAsignaturas();

   FILE* f = fopen("CALIFICACIONES.dat", "r+b");

   // recorremos el archivo
   Calificacion reg = read<Calificacion>(f);
   while( !feof(f) )
   {
```

```
      // procesamos el registro leido
      procesarCalificacion(reg,collAsig);

      reg = read<Calificacion>(f);
   }

   // mostramos los resultados
   mostrarResultados(collAsig);

   fclose(f);
   return 0;
}
```

La función `procesarCalificacion` verifica si la calificación corresponde a un aplazo (`calif`<4). De ser así, busca la asignatura y agrega el `idEst` al final de la colección de estudiantes aplazados.

```
void procesarCalificacion(Calificacion reg
                          ,Coll<RAsignatura>& collAsig)
{
   if( reg.calif<4 )
   {
      int pos = collFind<RAsignatura,int>(
                                    collAsig
                                   ,reg.idAsig
                                   ,cmpRAsignaturaId
                                   ,rAsignaturaFromString);

      // obtenemos el elemento encontrado en la posicion pos
      RAsignatura elm = collGetAt<RAsignatura>(
                                    collAsig
                                   ,pos
                                   ,rAsignaturaFromString);

      // agregamos el estudiante a la coleccion
      collAdd<int>(elm.collEst,reg.idEst,intToString);

      // actualizamos la coleccion de asignaturas
      collSetAt<RAsignatura>(collAsig
                             ,elm
                             ,pos
                             ,rAsignaturaToString);
```

```
    }
}
```

Para mostrar los resultados requeridos tendremos que recorrer la colección principal (la de asignaturas), y por cada elemento recorreremos la colección secundaria (la de estudiantes aplazados).

```
void mostrarResultados(Coll<RAsignatura> collAsig)
{
   // recorremos la coleccion de asignaturas
   collReset<RAsignatura>(collAsig);
   while( collHasNext<RAsignatura>(collAsig) )
   {
      RAsignatura ra = collNext<RAsignatura>(
                                    collAsig
                                   ,rAsignaturaFromString);

      // mostramos la asignatura
      string nomAsig = ra.asig.nomAsig;
      cout << nomAsig << endl;

      // recorremos la coleccion de estudiantes aplazados
      Coll<int> collEst = ra.collEst;
      collReset<int>(collEst);
      while( collHasNext<int>(collEst) )
      {
         int idEst = collNext<int>(collEst,stringToInt);

         // mostramos los estudiantes aplazados
         cout << idEst << endl;
      }
   }
}
```

Veamos la función que sube a memoria el archivo de asignaturas.

```
Coll<RAsignatura> subirAsignaturas()
{
```

```
   Coll<RAsignatura> ret = coll<RAsignatura>();
   FILE* f = fopen("ASIGNATURAS.dat","r+b");

   // leemos el primer registro del archivo
   Asignatura reg = read<Asignatura>(f);
   while( !feof(f) )
   {
      // creamos un RAsignatura
      RAsignatura ra = rAsignatura(reg,coll<int>(','));

      // lo agregamos a la coleccion
      collAdd<RAsignatura>(ret,ra,rAsignaturaToString);

      // leemos el siguiente registro
      reg = read<Asignatura>(f);
   }

   fclose(f);
   return ret;
}
```

4.5.8. Colección de colecciones, con descubrimiento (versión 8)

Se dispone de los archivos `CALIFICACIONES.dat` y `ASIGNATURAS.dat`, ambos sin orden y con las siguientes estructuras de registro.

CALIFICACIONES

```
struct Calificacion
{
   int idAsig;
   int idEst;
   int calif;
};
```

ASIGNATURAS

```
struct Asignatura
{
   int idAsig;
   char nomAsig[30];
   char maestroACargo[50];
};
```

Asumiendo que cada maestro podría tener varias asignaturas a cargo, se pide mostrar un listado de los estudiantes aplazados por maestro.

```
Maestro         Estudiantes aplazados
Juan            2, 3, 5, 6, 9, 10, 15, 17, 23
Pedro           1, 2, 8, 14
Marta           2, 4, 5, 14, 15, 16, 19, 21, 23
  :             :
```

Análisis: la estructura de datos en la que nos apoyaremos para resolver esta versión será una colección de maestros, y por cada uno, una colección de estudiantes aplazados. La variable `collMaes`, declarada como `Coll<RMaestro>`, representa dicha estructura de datos.

```
struct RMaestro
{
  string maestro;      // nombre del maestro
  Coll<int> collEst;   // coleccion de estudiantes aplazados
};
```

Como no conocemos de antemano cuáles son los maestros, tendremos que descubrirlos a medida que procesemos los registros del archivo de calificaciones, previa búsqueda en el archivo de asignaturas. En síntesis, el algoritmo será el siguiente:

1. Subimos el archivo de asignaturas a una colección.
2. Creamos una colección vacía: `Coll<RMaestro> collMaes`.
3. Recorremos el archivo de calificaciones, y por cada calificación:
    a. Buscamos por `idAsig` en la colección de asignaturas para obtener el maestro a cargo.
    b. Buscamos al maestro en la colección de maestros, y si no lo encontramos, lo agregamos (descubrimos los maestros).
    c. Si la calificación del estudiante es menor que 4, lo agregamos a la colección de estudiantes aplazados por el maestro.
4. Para mostrar los resultados, recorremos la colección de maestros, y por cada elemento recorremos la colección de estudiantes aplazados.

```
int main()
{
```

```
   // coleccion de asignaturas
   Coll<Asignatura> collAsig = subirAsignaturas();

   // coleccion de maestros
   Coll<RMaestro> collMaes = coll<RMaestro>();

   FILE* f = fopen("CALIFICACIONES.dat", "r+b");

   // recorremos el archivo
   Calificacion reg = read<Calificacion>(f);
   while( !feof(f) )
   {
      // procesamos
      procesarCalificacion(reg,collAsig,collMaes);

      // leemos el siguiente registro
      reg = read<Calificacion>(f);
   }

   // resultlados
   mostrarResultados(collMaes);

   fclose(f);
   return 0;
}
```

`procesarCalificación` busca la asignatura `reg.idAsig` en la colección `collAsig` para obtener el nombre del maestro a cargo. Luego busca al maestro en la colección `collMaes` y si no lo encuentra lo agrega (lo descubre). Todo esto, siempre y cuando `reg.calf`<4; es decir: que el estudiate `reg.idEst` haya reprobado. Siendo así, lo agrega a la colección de estudiantes aplazados del maestro.

```
void procesarCalificacion(Calificacion reg
                         ,Coll<Asignatura> collAsig
                         ,Coll<RMaestro>& collMaes)
{
   if( reg.calif<4 )
   {
```

```
      // obtenemos el elemento de la coleccion
      Asignatura asig = buscarAsignatura(reg.idAsig
                                         ,collAsig);

      // buscamos, y si corresponde agregamos al maestro
      string maestro = asig.maestroACargo;
      int pos = buscarMaestro(collMaes,maestro);

      // obtenemos el elemento de la coleccion
      RMaestro rm = collGetAt<RMaestro>(
                                       collMaes
                                      ,pos
                                      ,rMaestroFromString);

      collAdd<int>(rm.collEst,reg.idEst,intToString);
      collSetAt<RMaestro>(collMaes,rm,pos,rMaestroToString);
   }
}
```

La función `buscarAsignatura` simplemente busca por `id` dentro de `collAsig`, y retorna la asignatura.

```
Asignatura buscarAsignatura(int id,Coll<Asignatura> collAsig)
{
   // buscamos la asignatura
   int pos = collFind<Asignatura,int>(collAsig
                                      ,id
                                      ,cmpAsignaturaId
                                      ,asignaturaFromString);

   // obtenemos el elemento de la coleccion
   Asignatura asig = collGetAt<Asignatura>(
                                       collAsig
                                      ,pos
                                      ,asignaturaFromString);

   return asig;
}
```

Notemos la diferencia entre las funciones `buscarAsignatura`, cuyo código acabamos de ver, y `buscarMaestro`, que analizaremos a continuación. La primera retorna un registro (`Asignatura`), la segunda retorna la posición donde se ubica el registro dentro de la colección.

Esto se debe a que en `procesarCalificacion`, donde invocamos ambas funciones, no vamos a modificar la asignatura que buscamos. Sólo la utilizaremos para obtener el maestro a cargo. Sin embargo, sí modificaremos el elemento de la colección de maestros, pues le agregaremos un estudiante aplazado. Por tal movivo necesitamos la posición de dicho elemento, para poder hacer el `collSetAt`.

```
int buscarMaestro(Coll<RMaestro>& collMaes,string maestro)
{
   int pos = collFind<RMaestro,string>(collMaes
                                       ,maestro
                                       ,cmpRMaestroMaestro
                                       ,rMaestroFromString);

   if( pos<0 )
   {
      RMaestro rm = rMaestro(maestro,coll<int>(','));
      pos = collAdd<RMaestro>(collMaes,rm,rMaestroToString);
   }

   return pos;
}
```

La función `cmpRMaestroMaestro`, que compara un `RMaestro` con un `string` (nombre del maestro), necesaria para invocar a `collFind`, es la siguiente:

```
int cmpRMaestroMaestro(RMaestro rm,string m)
{
   return rm.maestro<m?-1:rm.maestro>m?1:0;
}
```

Finalmente, veamos la función que muestra los resultados requeridos:

```
void mostrarResultados(Coll<RMaestro> collMaes)
{
   // recorremos la coleccion de maestros
   collReset<RMaestro>(collMaes);
   while( collHasNext<RMaestro>(collMaes) )
   {
      RMaestro rm = collNext<RMaestro>(
                                    collMaes
                                   ,rMaestroFromString);

      // mostramos el nombre del maestro
      string nomMaes = rm.maestro;
      cout << nomMaes << endl;

      // recorremos la coleccion de estudiantes aplazados
      Coll<int> collEst = rm.collEst;

      collRest<int>(collEst);
      while( collHasNext<int>(collEst) )
      {
         int idEst = collNext<int>(collEst,stringToInt);

         // mostramos los estudiantes aplazados
         cout << idEst << endl;
      }
   }
}
```

4.5.9. Apareo de archivos (versión 9)

Se dispone de los archivos `ASIGNATURAS19.dat` y `ASIGNATURAS20.dat`, ambos con el mismo formato de registro, describiendo respectivamente las asignaturas y sus calificaciones promedio durante los años 2019 y 2020. Los dos archivos se encuentran ordenados por `idAsig`.

```
struct Asignatura
{
   int idAsig;
   char nomAsig[30];
   char maestroACargo[50];
   double califProm;
```

```
};
```


De un año a otro podrían aparecer nuevas asignaturas, o dejar de dictarse algunas. Tambien podría haber variaciones respecto de los maestros que las dictan.

Se pide:

1. Un listado de las asignaturas que sólo se dictaron en 2019.
2. Un listado de las asignaturas que comenzaron a dictarse en 2020.
3. Un listado de las asignaturas que se dictaron durante ambos años, pero estuvieron a cargo de diferentes maestros.

Todos los listados deben estar ordenados alfabéticamente por `nomAsig`.

Análisis: comenzaremos por analizar un ejemplo de cómo podrían presentarse los datos contenidos en ambos archivos.

ASIGNATURAS19.dat

idAsig	nomAsig	maestroACargo	califProm
3	Matemática	Marta	6.98
5	Geografía	Carlos	5.34
6	Arte	Romina	6.43
7	Ed. Física	José	8.31
10	Tecnología	Marta	7.56
11	Ed. Cívica	Carlos	5.47
14	Lengua	Romina	6.88
15	Historia	María	4.75

ASIGNATURAS20.dat

idAsig	nomAsig	maestroACargo	califProm
3	Matemática	Marta	7.02
5	Geografía	Pablo	7.88
7	Ed. Física	José	6.79
10	Tecnología	Iván	8.61
12	Plástica	Romina	6.03
14	Lengua	María	6.88
15	Historia	María	3.99
17	Inglés	Carla	6.35
19	Música	Romina	9.26

Figura 4.8. Ejemplo de datos

Es fundamental que ambos archivos se encuentren ordenados por `idAsig`. Esto nos permitirá comparar los valores que uno y otro archivo tienen en dicho campo.

Para simplificar el análisis llamaremos A19 y A20, respectivamente, a los archivos ASIGNATURAS19 y ASIGNATURAS20, y `a1` y `a2` a los `idAsig` leidos desde los archivos mencionados.

Comenzaremos leyendo la primera fila de A19 y A20, y compararemos los valores de `a1` y `a2`. Si `a1<a2` será porque `a1` corresponde a una asignatura que sólo se dictó durante 2019. Por el contrario, si `a1>a2`, el valor de `a2` será el identi-

ficador de una asignatura que se incorporó en 2020. Si ambos valores son iguales sabremos que la asignatura se dictó durante los dos años en cuestión.

Cada vez que determinemos uno u otro caso, volveremos a leer el archivo cuyo registro nos proporcionó información. Es decir: si `a1<a2`, luego de procesar, volveremos a leer A1. Si `a1>a2`, procesaremos y leeremos A2. Si `a1==a2`, luego de procesar leeremos ambos archivos.

Podría ocurrir que alguno de los dos archivos finalice primero. Si finalizó A1, todos los registros de A2 representarán asignaturas que aparecieron en 2020. Por el contrario, si primero finalizó A2 entonces los registros remanentes de A1 corresponderán a asignaturas que sólo se dictaron en 2019.

Para finalizar, como los listados deben aparecer ordenados por `nomAsig`, a medida que vayamos detectando cada caso tendremos que cargarlo en una colección, que ordenaremos al momento de mostrar los resultados.

```
int main()
{
   Coll<Asignatura> coll1 = coll<Asignatura>(); // solo 19
   Coll<Asignatura> coll2 = coll<Asignatura>(); // solo 20
   Coll<Asignatura> collA = coll<Asignatura>(); // ambos

   FILE* f1 = fopen("ASIGNATURAS19.dat","r+b");
   FILE* f2 = fopen("ASIGNATURAS20.dat","r+b");

   Asignatura a1 = read<Asignatura>(f1);
   Asignatura a2 = read<Asignatura>(f2);

   while( !feof(f1) && !feof(f2) )
   {
      if( a1.idAsig<a2.idAsig )
      {
         // la asignatura solo se dicto en 2019
         collAdd<Asignatura>(coll1,a1,asignaturaToString);
         a1 = read<Asignatura>(f1);
      }
      else
      {
         if( a1.idAsig>a2.idAsig )
         {
```

```
            // la asignatura aparece en 2020
            collAdd<Asignatura>(coll2,a2,asignaturaToString);
            a2 = read<Asignatura>(f2);
         }
         else
         {
            // la asignatura se dicto ambos anios
            procesarPunto3(a1,a2,collA);
            a1 = read<Asignatura>(f1);
            a2 = read<Asignatura>(f2);
         }
      }
   }

   while( !feof(f1) )
   {
      // asignaturas que solo aparecen en 2019
      collAdd<Asignatura>(coll1,a1,asignaturaToString);
      a1 = read<Asignatura>(f1);
   }

   while( !feof(f2) )
   {
      // asignaturas que solo se dictaron en 2020
      collAdd<Asignatura>(coll2,a2,asignaturaToString);
      a2 = read<Asignatura>(f2);
   }

   // resultados
   mostrarResultados("Solo en 2019",coll1);
   mostrarResultados("Comenzo en 2020",coll2);
   mostrarResultados("Ambos anios, dif. maestros",collA);

   fclose(f2);
   fclose(f1);
   return 0;
}
```

La función `procesarPunto3` agrega un elemento a la colección `collA` siempre y cuando el docente a cargo haya cambiado de un año al otro.

```
void procesarPunto3(Asignatura a1
                   ,Asignatura a2
                   ,Coll<Asignatura>& collA)
{
   string maes1 = a1.maestroACargo;
   string maes2 = a2.maestroACargo;

   if( maes1 != maes2 )
   {
      collAdd<Asignatura>(collA,a1,asignaturaToString);
   }
}
```

La función `mostrarResultados` es idéntica para los tres casos. Recibe un mensaje para mostrar un título, y la colección de asignaturas, que ordena y muestra.

```
void mostrarResultados(string msg,Coll<Asignatura> c)
{
   collSort<Asignatura>(c
                       ,cmpAsignatura
                       ,asignaturaFromString
                       ,asignaturaToString);

   cout << msg << endl;
   while( collHasNext<Asignatura>(c) )
   {
      Asignatura a = collNext<Asignatura>(
                                      c
                                     ,asignaturaFromString);

      cout << asignaturaToString(a) << endl;
   }
}
```

Para finalizar, la función que establece el orden de precedencia alfabético:

```
int cmpAsignatura(Asignatura x,Asignatura y)
{
```

```
   string m1 = x.maestroACargo;
   string m2 = y.maestroACargo;
   return m1<m2?-1:m1>m2?1:0;
}
```

4.5.10. Apareo de archivos, con corte de control (versión 10)

Se dispone de los archivos `RENDI19.dat` y `RENDI20.dat`, que contienen datos estadísticos sobre el rendimiento de los estudiantes que cursaron asignaturas en las diferentes comisiones de una escuela. Ambos archivos tienen el mismo formato de registro, y se encuentran ordenados por `idAsig`.

```
struct Rendi
{
   int idAsig;        // asignatura
   char comision;     // comision (ej. A, B,...)
   double califProm; // calificacion promedio
};
```

Podrían existir asignaturas que sólo se dictaron en 2019, otras que comenzaron a dictarse en 2020, y otras que se dictaron ambos años.

Se pide:

1. Para las asignaturas que sólo se dictaron en 2019, indicar cuál fue la mejor, y cuál la peor calificación promedio lograda por una comisión.
2. Para las asignaturas que sólo se dictaron durante 2020, indicar la calificación promedio general (promedio de los promedios de las comisiones).
3. Para las asignaturas que se dictaron ambos años, un listado indicando la calificación promedio 2020, y en qué proporción aumentó o disminuyó este valor respecto a 2019.

Análisis: La figura 4.9 ilustra cómo podrían presentarse los datos. Como ambos archivos están ordenados por `idAsig`, podemos verificar que la asignatura 1 sólo se dictó durante 2019. Para determinar cuál fue la mejor comisión y cuál la peor, tendremos que recorrer RENDI19, con corte de control por `idAsig`, buscando el valor

máximo y mínimo de `califProm`. Según el lote de datos, la mejor comisión fue la B, y la peor fue la C.

La asignatura 2 aparece a partir de 2020. Habrá que recorrer RENDI20 con corte de control por `idAsig` para calcular el promedio de los valores `califProm`. Según el ejemplo, dicho promedio será: 7.1.

RENDI19.dat

idAsig	comision	califProm
1	A	7,45
1	B	7,68
1	C	3,98
3	A	6,65
3	B	8,53
3	C	4,23
5	A	6,89
5	B	9,23
5	C	9,02
7	A	7,34
7	B	8,56
7	C	5,56
8	A	7,93
8	B	2,78
8	C	5,21

RENDI20.dat

idAsig	comision	califProm
2	A	5,32
2	B	7,55
2	C	8,43
3	A	9,54
3	B	2,67
3	C	8,53
4	A	7,23
4	B	8,74
4	C	9,44
6	A	2,35
6	B	7,52
6	C	8,37

Figura 4.9. Ejemplo de datos

La asignatura 3 se dictó durante 2019 y 2020. Haremos un corte de control por `idAsig` sobre ambos archivos para calcular el promedio logrado cada año. Mostraremos el promedio de 2020, y cómo este valor mejoró o empeoró respecto a 2019. Según los datos del ejemplo: el promedio de 2020 es: 6.91, y representa una mejora del 6.85% respecto al año anterior.

Utilizaremos tres colecciones que llamaremos: `coll1`, `coll2` y `collA`, donde guardaremos los elementos del listado 1 (sólo 2019), listado 2 (sólo 2020), y el listado 3 (Ambos años), cuyos tipos serán `Lst1`, `Lst2`, y `LstA` respectivamente.

```
struct Lst1
{
   int idAsig;
   char comMax;
   double max;
   char comMin;
   double min;


};

Coll<Lst1> coll1;
```

```
struct Lst2
{
   int idAsig;
   double promGral;
};


Coll<Lst2> coll2;
```

```
struct LstA
{
   int idAsig;
   double prom2;
   double porc;
};


Coll<LstA> collA;
```

La estrategia será recorrer ambos archivos a la par (apareo de archivos) para determinar ante cuál de los tres casos estamos posicionados.

- Si estamos ante un grupo de registros que representan estadísticas sobre una asignatura que sólo se dictó en 2019, recorreremos el archivo RENDI19 con corte de control por `idAsig` buscando cuál es la comisión que tuvo mejor rendimiento, y cuál la peor. Luego agregaremos una fila a `coll1`.
- Si estamos ante un grupo de registros que detallan estadísticas sobre una asignatura que comenzó en 2020, recorreremos RENDI20 con corte de control promediando los valores de `califProm`. Luego agregaremos una fila a `coll2`.
- Si los registros representan estadísticas de una asignatura que se dictó ambos años, promediaremos los promedios de 2019, los de 2020, y los compararemos para calcular la proporción en que las comisiones mejoraron o empeoraron de un año a otro. Finalmente, agregaremos una fila a `collA`.

```
int main()
{
   Coll<Lst1> coll1 = coll<Lst1>();
   Coll<Lst2> coll2 = coll<Lst2>();
   Coll<LstA> collA = coll<LstA>();

   FILE* f1 = fopen("RENDI19.dat","r+b");
   FILE* f2 = fopen("RENDI20.dat","r+b");
```

```
   Rendi r1 = read<Rendi>(f1);
   Rendi r2 = read<Rendi>(f2);
   while( !feof(f1) && !feof(f2) )
   {
      if( r1.idAsig<r2.idAsig )
      {
         // solo 2019
         procesar1(f1,r1,coll1);
      }
      else
      {
         // solo 2020
         if( r1.idAsig>r2.idAsig )
         {
            procesar2(f2,r2,coll2);
         }
         else
         {
            // ambos anios
            procesarA(f1,r1,f2,r2,collA);
         }
      }
   }

   while( !feof(f1) )
   {
      procesar1(f1,r1,coll1);
   }

   while( !feof(f2) )
   {
      procesar2(f2,r2,coll2);
   }

   mostrarListado1(coll1);
   mostrarListado2(coll2);
   mostrarListadoA(collA);

   fclose(f2);
   fclose(f1);

   return 0;
}
```

Cada vez que entremos a `procesar1`, RENDI19 estará posicionado al inicio de un grupo de registros que corresponden a una asignatura que sólo se dictó en 2019.

Lo que haremos será recorrer el archivo con corte de control, buscando el mínimo y máximo valor de `califProm` para agregarlos al final de la colección `coll1`.

```
void procesar1(FILE* f,Rendi& r,Coll<Lst1>& coll1)
{
   // el primer registro se maximo y minimo
   char comMax = r.comision;
   double max = r.califProm;

   // el primer registro es maximo y minimo
   char comMin = r.comision;
   double min = r.califProm;

   int idAsigAnt = r.idAsig;
   while( !feof(f) && r.idAsig==idAsigAnt )
   {
      if( r.califProm>max )
      {
         comMax = r.comision;
         max = r.califProm;
      }
      else
      {
         if( r.califProm<min )
         {
            comMin = r.comision;
            min = r.califProm;
         }
      }

      r = read<Rendi>(f);
   }

   // agregamos un elemento tipo Lst1 a la collecion
   Lst1 elm = lst1(idAsigAnt,comMax,max,comMin,min);
   collAdd<Lst1>(coll1,elm,lst1ToString);
}
```

Cada vez que entremos a `procesar2`, RENDI20 estará posicionado al inicio de un grupo de registros que corresponden a una asignatura que sólo se dictó en 2020.

Vamos a recorrer el archivo con corte de control para calcular el promedio de los valores de `califProm`, y agregarlo al final de la colección `coll2`.

La recorrida con corte de control, así como el cálculo del valor promedio de `califProm`, lo delegaremos en la función `promCC`, que podremos reutilizar más adelante; pues resolver el punto 3 requiere realizar exactamente el mismo proceso.

```
void procesar2(FILE* f,Rendi& r,Coll<Lst2>& coll2)
{
   int idAsig = r.idAsig;
   double prom = promCC(f,r);
   Lst2 elm = lst2(idAsig,prom);
   collAdd<Lst2>(coll2,elm,lst2ToString);
}
```

El código de la función `promCC` lo vemos a continuación.

```
double promCC(FILE* f,Rendi& r)
{
   int cont = 0;
   double sum = 0;

   int idAsigAnt = r.idAsig;
   while( !feof(f) && r.idAsig==idAsigAnt )
   {
      sum+=r.califProm;
      cont++;

      r = read<Rendi>(f);
   }

   double prom = sum/cont;
   return prom;
}
```

Finalmente, cada vez que entremos a `procesarA` recibiremos ambos archivos posicionados al inicio de un grupo de registros que representan estadísticas de una asignatura que se dictó durante 2019 y 2020.

Recorreremos RENDI19 y RENDI20 con corte de control, calculando sobre cada uno el valor promedio de `califProm`, para establecer cuánto varió dicho indicador entre 2020 y 2019. Finalmente, agregaremos una fila en `collA`.

Como podremos observar, la función `promCC` nos será de gran ayuda.

```
void procesarA(FILE* f1
               ,Rendi& r1
               ,FILE* f2
               ,Rendi& r2
               ,Coll<LstA>& collA)
{
   int idAsig = r1.idAsig; // <-- ATENCION
   double prom1 = promCC(f1,r1); // corte de control y prom
   double prom2 = promCC(f2,r2); // corte de control y prom

   double porc = 100-prom1/prom2*100;

   LstA elm = lstA(idAsig,prom2,porc);
   collAdd<LstA>(collA,elm,lstAToString);
}
```

Antes de invocar a `promCC` debemos resguardar el valor de `idAsig`. Esto es necesario porque que dicha función, luego de emprender el recorrido con corte de control, dejará al archivo posicionado al inicio del siguiente grupo de registros, por lo cual `r1` (y luego `r2`) tendrá el `idAsig` de otra asignatura.

`mostrarListado1`, `mostrarListado2` y `mostrarListadoA`, simplemente iteran la colección y muestran sus elementos por pantalla.

4.5.11. Búsqueda binaria sobre el archivo de consulta (versión 11)

Se dispone del archivo `PADRON.dat`, que contiene el padrón de los estudiantes matriculados en todas las escuelas de la municipalidad. Este archivo se encuentra ordenado por `idEst`, y su estructura de registro es la siguiente:

```
struct Padron
{
   int idEst;                  // identificador de estudiante
   int dni;                    // documento de identidad
   char nombre[20];
   char telefono[20];
   char direccion[20];
   int codigoPostal;
   int fechaNacimiento;        // aaaammdd
   int idEscuela;              // escuela donde estudia
   int fechaMatriculacion;     // cuando se matriculo
   int idSeguroSocial;         // cobertura medica
};
```

Se cuenta también con el archivo `INSCRIPCIONES.dat`, sin orden, con las inscripciones de los estudiantes en los diferentes establecimientos educativos.

```
struct Inscripcion
{
   int idEst;
   int idEscuela;
   int fecha;    // fecha de la inscripcion (aaaammdd)
};
```

Se pide generar el archivo `INCONSISTENCIAS.dat` con las inscripciones inconsistentes. Detallando el tipo de problema (1, 2 o 3), el nombre del estudiante involucrado, y el identificador de la escuela (`idEscuela`).

Tipos de problema:

1. El estudiante no figura en el padrón (tipo de problema: 1).
2. El estudiante figura matriculado en una escuela diferente a la que se inscribió (tipo de problema: 2).
3. La fecha de inscripción es anterior a la de matriculación (tipo de problema 3).

Análisis: la estrategia consiste en recorrer las inscripciones (archivo de novedades), y por cada una realizar una consulta en el padrón (archivo de consulta). Pero a diferencia de los casos anteriores, donde manejamos las consultas en memoria, aquí

el archivo de consulta es demasiado extenso, pues contiene un registro por cada uno de los estudiantes de la municipalidad. Por esto, no podremos subirlo a una colección.

Como el archivo de consulta está ordenado por `idEst`, que justamente es el campo por el cual nos interesa buscar, lo accederemos mediante el algoritmo de la búsqueda binaria.

Para grabar el archivo de inconsistencias utilizaremos la siguiente estructura:

```
struct Inconsistencia
{
   int idTipoProblema; // 1, 2 o 3
   char nombreEstudiante[20];
   int idEscuela;
};
```

Con esto, la estrategia de solución será:

1. Creamos el archivo de salida: `fopen("INCONSISTENCIAS.dat","w+b")`.
2. Recorremos el archivo de inscripciones.
3. Por cada inscripción, buscamos al estudiante en el padrón.
    a. Si no existe, grabamos un registro con `idTipoProblema`=1.
    b. Si existe, pero no coinciden los valores de `idEscuela` grabamos un registro con `idTipoProblema`=2.
    c. Si existe, pero las fechas son inconsistentes, `idTipoProblema`=3.

```
// constantes
#define ERR_ESTINEXIST 1; // error tipo 1
#define ERR_ESCUELADIF 2; // error tipo 2
#define ERR_FECHAANT   3; // error tipo 3

int main()
{
   // abrimos los archivos
   FILE* fIns = fopen("INSCRIPCIONES.dat","r+b");
   FILE* fPad = fopen("PADRON.dat","r+b");
   FILE* fOut = fopen("INCONSISTENCIAS.dat","w+b");

   Inscripcion ins = read<Inscripcion>(fIns);
   while( !feof(fIns) )
   {
```

```
      // buscamos el estudiante y asignamos true o false
      // a enc segun encuentre o no
      bool enc;
      Padron pad = buscarEstudiante(ins.idEst,fPad,enc);

      // procesamos el registro
      procesarInscripcion(ins,pad,enc,fOut);

      // leemos el siguiente registro
      ins = read<Inscripcion>(fIns);
   }

   // cerramos los archivos
   fclose(fOut);
   fclose(fPad);
   fclose(fIns);

   return 0;
}
```

Comenzaremos analizando la función `procesarInscripcion`, que debe verificar si existe alguna de las inconsistencias mencionadas, y en tal caso grabar un registro en el archivo de salida.

```
void procesarInscripcion(Inscripcion ins
                         ,Padron pad
                         ,bool& enc
                         ,FILE* fOut)
{
   Inconsistencia out;

   // asignamos la escuela involucrada
   out.idEscuela = ins.idEscuela;

   if( !enc )
   {
      strcpy(out.nombreEstudiante,"desconocido!!");
      out.idTipoProblema = ERR_ESTINEXIST;
      write<Inconsistencia>(fOut,out);
   }
   else
   {
```

```
      // asignamos pad.nombre a out.nombreEstudiante
      strcpy(out.nombreEstudiante,pad.nombre);
      if( pad.idEscuela!=ins.idEscuela )
      {
         out.idTipoProblema = ERR_ESCUELADIF;
         write<Inconsistencia>(fOut,out);
      }
      else
      {
         if( fechaCmp(ins.fecha,pad.fechaMatriculacion)<0 )
         {
            out.idTipoProblema = ERR_FECHAANT;
            write<Inconsistencia>(fOut,out);
         }
      }
   }
}
```

Observemos que utilizamos la función `strcpy` (de la biblioteca de C) para copiar en `out.nombreEstudiante` el contenido de `pad.nombre`. Esto se debe a que ambas cadenas son tipo `char[]`, y el único modo asignar una en otra es hacerlo copiando carácter por carácter. Eso es lo que hace `strcpy`, cuyo nombre es una abreviatura de *string copy*.

La función `buscarEstudiante` realiza una búsqueda binaria sobre el archivo de consultas.

```
Padron buscarEstudiante(int id,FILE* f,bool& enc)
{
   Padron ret;

   int i = 0;
   int j = fileSize<Padron>(f)-1;

   enc = false;
   while( i<=j && !enc)
   {
      int k = (i+j)/2;
```

```
      // posiciono y leo
      seek<Padron>(f,k);
      ret = read<Padron>(f);

      if( ret.idEst<id )
      {
         i = k+1;
      }
      else
      {
         if( ret.idEst>id )
         {
            j = k-1;
         }
         else
         {
            enc = true;
         }
      }
   }

   return ret;
}
```

4.5.12. Indexación directa (versión 12)

Ídem anterior, considerando que PADRON e INSCRIPCIONES están desordenados.

PADRON

```
struct Padron
{
   int idEst;
   int dni;
   char nombre[20];
   char telefono[20];
   char direccion[20];
   int codigoPostal;
   int fechaNacimiento;
   int idEscuela;
   int fechaMatriculacion;
   int idSeguroSocial;
```

INSCRIPCIONES - INCONSISTENCIAS

```
struct Inscripcion
{
   int idEst;
   int idEscuela;
   int fecha; // aaaammdd
};
```

INCONSISTENCIAS

```
struct Inconsistencia
{
```

```
};
```

```
   int idTipoProblema;
   char nombreEstudiante[20];
   int idEscuela;
};
```

Análisis: Como el archivo de consultas es extenso, no podemos subirlo a memoria. La opción de accederlo mediante una búsqueda binaria ya no es válida, porque dicho algoritmo de búsqueda requiere que el archivo esté ordenado. La alternativa será indexar el archivo.

La técnica de indexación lineal o directa consiste en subir a memoria sólo el campo clave del registro (campo de búsqueda). De este modo, la búsqueda la realizaremos sobre una colección, y la posición que dicha clave ocupa dentro de la colección coincidirá con la posición que el registro completo ocupa dentro del archivo.

Desarrollaremos dos funciones: `idxCrear`, para generar la colección con los valores de búsqueda, e `idxLeer`, que usaremos para leer un registro del archivo, previo paso por el índice para buscar su posición.

Es importante observar que ambas funciones reciben abierto el archivo sobre el que van a trabajar, y no lo deben cerrar. Pues, será necesario utilizarlo durante el resto del programa.

`idxCrear` es tan simple que casi no requiere explicación. Recorre el archivo subiendo el `idEst` (campo clave) de cada registro a una colección de enteros.

```
Coll<int> idxCrear(FILE* f)
{
   Coll<int> idx = coll<int>();

   // posicionamos al archivo en el primer registro
   seek<Padron>(f,0);

   Padron p = read<Padron>(f);
   while( !feof(f) )
   {
      collAdd<int>(idx,p.idEst,intToString);
      p = read<Padron>(f);
   }
```

```
   return idx;
}
```

idxBuscar recibe el archivo, el id a buscar, y la colección de índices. Busca el id dentro de la colección y utiliza la posición donde lo encontró para posicionarse en el archivo. Luego, lee y retorna el registro leído. Si el id buscado no existe dentro de la colección asignará false al parámetro enc.

```
Padron idxBuscar(int id, FILE* f,bool& enc,Coll<int> idx)
{
   Padron ret;

   int pos = collFind<int,int>(idx,id,cmpIntId,stringToInt);
   if( pos>=0 )
   {
      seek<Padron>(f,pos);
      ret = read<Padron>(f);
   }

   enc = pos>=0; //true o false

   return ret;
}
```

Con estas dos funciones podemos modificar la función buscarEstudiante que desarrollamos durante la anterior versión del problema.

```
Padron buscarEstudiante(int id,FILE* f,bool& enc,Coll<int> idx)
{
   return idxBuscar(id,f,enc,idx);
}
```

Ahora el programa principal de la versión anterior es prácticamente el mismo que necesitamos para esta versión. Sólo que, al inicio, debemos crear el índice.

```
// constantes
#define ERR_ESTINEXIST 1; // error tipo 1
#define ERR_ESCUELADIF 2; // error tipo 2
#define ERR_FECHAANT   3; // error tipo 3

int main()
{
   // abrimos los archivos
   FILE* fIns = fopen("INSCRIPCIONES.dat","r+b");
   FILE* fPad = fopen("PADRON.dat","r+b");
   FILE* fOut = fopen("INCONSISTENCIAS.dat","w+b");

   // creamos el indice
   Coll<int> idx = idxCrear(fPad);

   Inscripcion ins = read<Inscripcion>(fIns);
   while( !feof(fIns) )
   {
      // buscamos el estudiante, asigna true o false a enc
      // segun encuentre o no lo que busca
      bool enc;
      Padron pad = buscarEstudiante(ins.idEst,fPad,enc,idx);

      // procesamos el registro
      procesarInscripcion(ins,pad,enc,fOut);

      // leemos el siguiente registro
      ins = read<Inscripcion>(fIns);
   }

   // cerramos los archivos
   fclose(fOut);
   fclose(fPad);
   fclose(fIns);
   return 0;
}
```

4.5.13. Indexación indirecta, con corte de control (versión 13)

Se dispone del archivo `PADRON.dat`, sin orden, cuya estructura de registro vemos a continuación:

```
struct Padron
{
   int idEst;                  // identificador de estudiante
   int dni;                    // documento de identidad
   char nombre[20];
   char telefono[20];
   char direccion[20];
   int codigoPostal;
   int fechaNacimiento;        // aaaammdd
   int idEscuela;              // escuela donde estudia
   int fechaMatriculacion;     // aaaammdd
   int idSeguroSocial;         // cobertura medica
};
```

Se pide un listado de los estudiantes empadronados, según su código postal.

Codigo postal: 999

Nombre	*Direccion*	...
XXXXXXXXXX	XXXXXXXXXXX	...
XXXXXXXXXX	XXXXXXXXXXX	...
XXXXXXXXXX	XXXXXXXXXXX	...

:

Análisis: Si PADRON estuviera ordenado por código postal podríamos resolver este problema recorriéndolo con corte de control. Pues bien, esta continúa siendo una opción válida; sólo que primero tendremos que indexarlo por dicho campo.

La estrategia será: indexar PADRON guardando en el índice la posición de cada registro, tal como lo hicimos previamente en el apartado de ordenamiento. Luego ordenaremos el índice, lo recorreremos con corte de control, y por cada elemento realizaremos un acceso directo al archivo para obtener todos los datos del registro.

Utilizaremos la estructura `Idx`, cuyo código vemos a continuación.

```
struct Idx
{
   int codigoPostal;
   int pos;
```

```
};
```

La función `indexarPadron` indexa el archivo.

```
Coll<Idx> indexarPadron(FILE* f)
{
   Coll<Idx> ret = coll<Idx>();

   // reseteamos el archivo
   seek<Padron>(f,0);

   // recorremos el archivo
   Padron p = read<Padron>(f);
   while( !feof(f) )
   {
      Idx i = idx(p.codigoPostal,filePos<Padron>(f)-1);
      collAdd<Idx>(ret,i,idxToString);
      p = read<Padron>(f);
   }

   // ordenamos el indice
   collSort<Idx>(ret,cmpIdx,idxFromString,idxToString);

   return ret;
}
```

Para ordenar el índice usamos la función `cmpIdx`.

```
int cmpIdx(Idx a,Idx b)
{
   return a.codigoPostal-b.codigoPostal;
}
```

Ahora resolvemos el problema recorriendo el índice con corte de control, para lo cual usaremos la versión sobrecargada de `collNext`.

```
int main()
{
   // abrimos el archivo e indexamos
   FILE* f = fopen("PADRON.dat","r+b");
   Coll<Idx> cIdx = indexarPadron(f);
   collReset<Idx>(cIdx);

   // recorremos con corte de control
   bool endOfColl;
   Idx i = collNext<Idx>(cIdx,endOfColl,idxFromString);
   while( !endOfColl )
   {
      // mostramos el codigo postal
      cout << "Codigo postal: " << i.codigoPostal << endl;

      // variable de control
      int codigoPostalAnt = i.codigoPostal;
      while( !endOfColl && codigoPostalAnt==i.codigoPostal)
      {
         // leemos el registro en la posicion pos
         Padron p = leerPadron(f,i.pos);

         // mosotramos los datos del estudiante
         cout << padronToString(p) << endl;

         // leemos el siguientes
         i = collNext<Idx>(cIdx,endOfColl,idxFromString);
      }
   }

   return 0;
}
```

Finalmente, el código de la función `leerPadron` es el siguiente.

```
Padron leerPadron(FILE* f,int pos)
{
   seek<Padron>(f,pos);
   return read<Padron>(f);
}
```

4.5.14. Indexación indirecta múltiple (versión 14)

Se dispone del archivo `PADRON.dat`, sin orden, cuya estructura de registro vemos a continuación. Podrían existir, por error, estudiantes con el mismo `idEst`.

```
struct Padron
{
   int idEst;                  // identificador de estudiante
   int dni;                    // documento de identidad
   char nombre[20];
   char telefono[20];
   char direccion[20];
   int codigoPostal;
   int fechaNacimiento;        // aaaammdd
   int idEscuela;              // escuela donde estudia
   int fechaMatriculacion;     // aaaammdd
   int idSeguroSocial;         // cobertura medica
};
```

Con el objetivo de depurar el padrón de estudiantes, se pide:

1. Un listado de todos los estudiantes empadronados correctamente (sin error), ordenado por `idEst`.
2. Para aquellos estudiantes con `idEst` duplicado, emitir el siguiente listado, ordenado por `idEst`, y con el siguiente formato:

```
Id. Estudiante: 999
      Nombre          DNI     Telefono        Direccion
      XXXXXXXXXXX     99999   99999999999     XXXXXXXXXXXXXXXXXX
      XXXXXXXXXXX     99999   99999999999     XXXXXXXXXXXXXXXXXX
            :           :          :               :
Id. Estudiante: 999
      Nombre          DNI     Telefono        Direccion
            :           :          :               :
  :
```

Análisis: Indexaremos el archivo en una colección, cuyos elementos guardarán: para cada `idEst`, las posiciones del archivo en que aparece dicho identificador.

```
struct Idx
{
   int idEst;           // clave de busqueda
   Coll<int> collPos;   // posiciones con este idEst
};
```

Para esto, vamos a recorrer el padrón buscando, por cada registro leído, si existe o no una entrada en el índice (colección de `Idx`). Si dicha entrada no existe, la creamos. Luego agregamos la posición del registro a la colección `collPos`.

Cuando terminemos de leer el archivo, el índice tendrá tantas entradas como `idEst` diferentes existen en el padrón. Aquellas entradas cuya colección (`collPos`) tenga más de un elemento, corresponderán a `idEst` repetidos.

Según los datos de la derecha, el índice quedará como vemos a la izquierda.

`Coll<Idx> cIdx`

idAsig	collPos
5	0, 2
3	1, 6, 11
8	3
1	4, 7, 10
2	5, 8
4	9
6	12
:	:

`PADRON.dat`

	idAsig	dni	nombre	...
0	5	233	Pedro	...
1	3	632	Pablo	...
2	5	412	Carlos	...
3	8	472	José	...
4	1	342	María	
5	2	834	Analía	...
6	3	262	Nélida	...
7	1	682	Roberto	...
8	2	461	Carolina	...
9	4	735	Diego	...
10	1	271	Alejandro	...
11	3	132	Martín	...
12	6	426	Darío	...
:	:	:	:	:

Figura 4.10. Ejemplo de datos

Al igual que en la versión anterior, no existe una relación directa entre las posiciones del índice y las posiciones de los registros del archivo. Cada `idEst`, identificado por una entrada en el índice, tiene asociada la posición que ocupa (dentro del archivo) el registro que representa (o una lista de posiciones si hubiese error). Esto nos permitirá alterar el orden del índice, por ejemplo, para ordenarlo.

```
int main()
{
   // indice
   Coll<Idx> cIdx = coll<Idx>();

   FILE* f = fopen("PADRON.dat","r+b");

   Padron r = read<Padron>(f);
   while( !feof(f) )
   {
      // buscamos por idEst y (si corresponde) agregamos
      int pos = descubrirEnIdx(r.idEst,cIdx);
      Idx x = collGetAt<Idx>(cIdx,pos,idxFromString);

      // agregamos la posicion del archivo a la coleccion
      int posArch = filePos<Padron>(f)-1;
      collAdd<int>(x.collPos,posArch,intToString);
      collSetAt<Idx>(cIdx,x,pos,idxToString);

      r = read<Padron>(f);
   }

   // ordenamos el indice
   collSort<Idx>(cIdx,cmpIdx,idxFromString,idxToString);

   // resultados
   mostrarListadoPunto1(cIdx,f);
   mostrarListadoPunto2(cIdx,f);

   fclose(f);
   return 0;
}
```

La función de comparación que nos permitirá ordenar el índice por `idEst` es:

```
int cmpIdx(Idx a,Idx b)
{
   return a.idEst-b.idEst;
}
```

La función `descubrirEnIdx` busca y/o agrega según corresponda.

```
int descubrirEnIdx(int idEst,Coll<Idx>& cIdx)
{
   int pos = collFind<Idx,int>(cIdx
                              ,idEst
                              ,cmpIdxId
                              ,idxFromString);

   if( pos<0 )
   {
      Idx x = idx(idEst,coll<int>(','));
      pos = collAdd<Idx>(cIdx,x,idxToString);
   }

   return pos;
}
```

Veamos la función de comparación que usamos para invocar a `collFind`:

```
int cmpIdxId(Idx x,int id)
{
   return x.idEst-id;
}
```

Para emitir el listado solicitado en el punto 1, recorreremos el índice ignorando aquellas entradas cuya colección `collPos` tenga más de un elemento, porque corresponden a registros con `idEst` duplicados.

De este modo, por cada una de las entradas del índice cuya colección tenga un único elemento, éste indicará en qué posición del archivo se ubica el registro que debemos mostrar por pantalla. Entonces: nos posicionamos, leemos e imprimimos.

```
void mostrarListadoPunto1(Coll<Idx> cIdx,FILE* f)
{
   cout << "--ESTUDIANTES EMPADRONADOS--" << endl;

   // iteramos la coleccion
   collReset<Idx>(cIdx);
   while( collHasNext<Idx>(cIdx) )
   {
      Idx x = collNext<Idx>(cIdx,idxFromString);

      // verificamos que no haya idEst duplicados
      if( collSize<int>(x.collPos)==1 )
      {
         // nos posicionamos y leemos
         int posArch = collGetAt<int>(x.collPos
                                      ,0
                                      ,stringToInt);
         seek<Padron>(f,posArch);
         Padron p = read<Padron>(f);

         // mostramos
         cout << padronToString(p) << endl;
      }
   }
}
```

`mostrarListadoPunto2` muestra, para cada `idEst` duplicado, la lista de estudiantes que tienen, por error, el mismo identificador.

```
void mostrarListadoPunto2(Coll<Idx> cIdx,FILE* f)
{
   cout << "--ESTUDIANTES CON ERROR--" << endl;

   // iteramos la coleccion
   collReset<Idx>(cIdx);
   while( collHasNext<Idx>(cIdx) )
   {
```

```
      Idx x = collNext<Idx>(cIdx,idxFromString);

      // verificamos que no haya idEst duplicados
      if( collSize<int>(x.collPos)>1 )
      {
         // mostramos
         cout << "Id. Estudiante: " << x.idEst << endl;

         // posiciones de registros con error
         Coll<int> collPos = x.collPos;

         // iteramos la coleccion
         collReset<int>(collPos);
         while( collHasNext<int>(collPos) )
         {
            int z = collNext<int>(collPos,stringToInt);

            // nos posicionamos y leemos
            seek<Padron>(f,z);
            Padron p = read<Padron>(f);

            // mostramos
            cout << padronToString(p) << endl;
         }
      }
   }
}
```

4.5.15. Ordenar y depurar un archivo (versión 15)

Se dispone del archivo `PADRON.dat`, sin orden, cuya estructura de registro vemos a continuación. Podrían existir, por error, estudiantes con el mismo `idEst`.

```
struct Padron
{
   int idEst;                  // identificador de estudiante
   int dni;                    // documento de identidad
   char nombre[20];
   char telefono[20];
   char direccion[20];
   int codigoPostal;
```

```
   int fechaNacimiento;       // aaaammdd
   int idEscuela;             // escuela donde estudia
   int fechaMatriculacion;    // aaaammdd
   int idSeguroSocial;        // cobertura medica
};
```

1. Se pide generar el archivo `PADRONFIX.dat`, con la misma estructura que PADRON, asignando nuevos `idEst` a aquellos estudiantes cuyo identificador está duplicado. El nuevo archivo debe estar ordenado por `idEst`.
2. ¿Cómo cambiarían la estructura de datos, estrategia y algoritmo, si el listado del punto anterior debiera estar ordenado por `idEscuela+idEst`?

Análisis: la estrategia de solución, así como la estructura de datos, son idénticas a las del problema anterior. La diferencia que debemos analizar es cómo vamos a generar el archivo solicitado.

Primero ordenaremos el índice por `idEst`. Luego lo recorreremos omitiendo las entradas cuyo `collPos` tenga más de un elemento. Todo lo que tendremos que hacer será acceder al registro del padrón, y grabarlo en PADRONFIX.

Luego, volveremos a iterar el índice para grabar, al final de PADRONFIX, los registros de los estudiantes con `idEst` duplicado. A cada uno de estos registros les asignaremos nuevos `idEst`, generados como una secuencia consecutiva a partir del mayor `idEst` (el último que aparece en el índice).

El programa principal es igual al anterior. La diferencia está al final, donde reemplazamos `mostrarListadoPunto1` y `mostrarListadoPunto2` por la función `generarArchivo`.

```
int main()
{
   // indice
   Coll<Idx> cIdx= coll<Idx>();

   FILE* f = fopen("PADRON.dat","r+b");

   Padron r = read<Padron>(f);
   while( !feof(f) )
   {
```

```
      // buscamos por idEst y (si corresponde) agregamos
      int pos = descubrirEnIdx(r.idEst,cIdx);
      Idx x = collGetAt<Idx>(cIdx,pos,idxFromString);

      // agregamos la posicion del archivo a la coleccion
      int posArch = filePos<Padron>(f)-1;
      collAdd<int>(x.collPos,posArch,intToString);
      collSetAt<Idx>(cIdx,x,pos,idxToString);

      r = read<Padron>(f);
   }

   // ordenamos y generamos el archivo
   generarArchivo(cIdx,f);

   fclose(f);
   return 0;
}
```

Veamos la función `generarArchivo`.

```
void generarArchivo(Coll<Idx> cIdx,FILE* f)
{
   FILE* fOut = fopen("PADRONFIX.dat","w+b");

   // ordenamos el indice
   collSort<Idx>(cIdx,cmpIdx,idxFromString,idxToString);

   // grabamos registros sin error
   collReset<Idx>(cIdx);
   while( collHasNext<Idx>(cIdx) )
   {
      Idx x = collNext<Idx>(cIdx,idxFromString);
      Coll<int> collPos = x.collPos;
      if( collSize<int>(collPos)==1 )
      {
         // grabamos
         int pos = collGetAt<int>(collPos,0,stringToInt);
         Padron p = leerPadron(f,pos);
         write<Padron>(fOut,p);
```

```
      }
   }

   // el ultimo idEst de la coleccion + 1 es el proximo
   // idEst a partir del cual asignaremos a los registros
   Idx elm = collGetAt<Idx>(cIdx
                           ,collSize<int>(cIdx)-1
                           ,idxFromString);

   // nuevos idEst para los estudiantes con error
   int idEst = eml.idEst+1

   // grabamos registros con error
   collReset<Idx>(cIdx);
   while( collHasNext<Idx>(cIdx) )
   {
      Idx x = collNext<Idx>(cIdx,idxFromString);

      Coll<int> collPos = x.collPos;
      if( collSize<int>(collPos)>1 )
      {
         collReset<int>(collPos);
         while( collHasNext<int>(collPos) )
         {
            int pos = collNext<int>(collPos,stringToInt);
            Padron p = leerPadron(f,pos);
            p.idEst = idEst;
            write<Padron>(fOut,p);
            idEst++;
         }
      }
   }
}
```

La función `leerPadron` lee y retorna el registro ubicado en la posición `pos`.

```
Padron leerPadron(FILE* fPad,int pos)
{
   // posicionamos y leemos
   seek<Padron>(fPad,pos);
   Padron p = read<Padron>(fPad);
```

```
   return p;
}
```

Finalmente, en respuesta a la pregunta planteada en el punto 2, para que el archivo PADRONFIX pudiera quedar ordenado por `idEscuela+idEst` deberíamos guardar en el índice, al momento de crearlo, el identificador de escuela. La estructura `Idx` quedaría así:

```
struct Idx
{
   int idEst;           // clave de busqueda
   int idEscuela;
   Coll<int> collPos;   // posiciones con este idEst
};
```

La función `descubrirEnIdx` se modificaría así:

```
int descubrirEnIdx(int idEst,int idEscuela,Coll<Idx>& cIdx)
{
   int pos = collFind<Idx,int>(cIdx
                               ,idEst
                               ,cmpIdxId
                               ,idxFromString);

   if( pos<0 )
   {
      Idx x = idx(idEst,idEscuela,coll<int>(','));
      pos = collAdd<Idx>(cIdx,x,idxToString);
   }

   return pos;
}
```

Finalmente, en `generarArchivo` cambiaríamos el criterio de ordenamiento invocando a `collSort` con esta nueva función de comparación:

```
// :
collSort<Idx>(cIdx,cmpIdx2,idxFromString,idxToString);
// :
```

Donde `cmpIdx2` tendrá el siguiente código:

```
int cmpIdx2(Idx a,Idx b)
{
   int aEsc = a.idEscuela;
   int bEsc = b.idEscuela;
   int aEst = a.idEst;
   int bEst = b.idEst;

   return aEsc==bEsc?aEst-bEst:aEsc-bEsc;
}
```

4.6. Autoevaluación y ejercicios

Los ejercicios propuestos para este capítulo son problemas contextualizados, que resolveremos dos veces. La primera vez será justo ahora. Pues, además de las técnicas que analizamos con el caso testigo, disponemos del TAD `Coll`, que nos permite mantener en memoria colecciones de objetos.

Al finalizar el capítulo 5 contaremos con más y mejores recursos: *arrays*, mapas, listas, pilas, colas. Será entonces el momento de volver, para resolver nuevamente cada uno de estos problemas y aplicar los nuevos recursos que adquirimos.

Autoevaluación

Problemas

4.7. ¿Qué sigue?

En el próximo capítulo estudiaremos otros tipos de estructuras de datos, mucho más flexibles y funcionales que el TAD `Coll` con el que hemos venido trabajando.

Para implementar estas estructuras, primero tendremos que adquirir conocimientos sobre la gestión dinámica de memoria. Justamente, al tratarse de estructuras dinámicas, ocuparán más memoria a medida que incorporemos más elementos, y menos memoria si removemos elementos de la colección.

Finalmente, y tal como mencionamos, luego de programar y encapsular en TAD cada una de estas estructuras de datos, volveremos a este capítulo para resolver, una vez más, los mismos problemas, pero con mejores herramientas.

La figura 4.11 describe este flujo de trabajo, y detalla los recursos con los que podremos contar luego de finalizar la lectura de los capítulos 4 y 5.

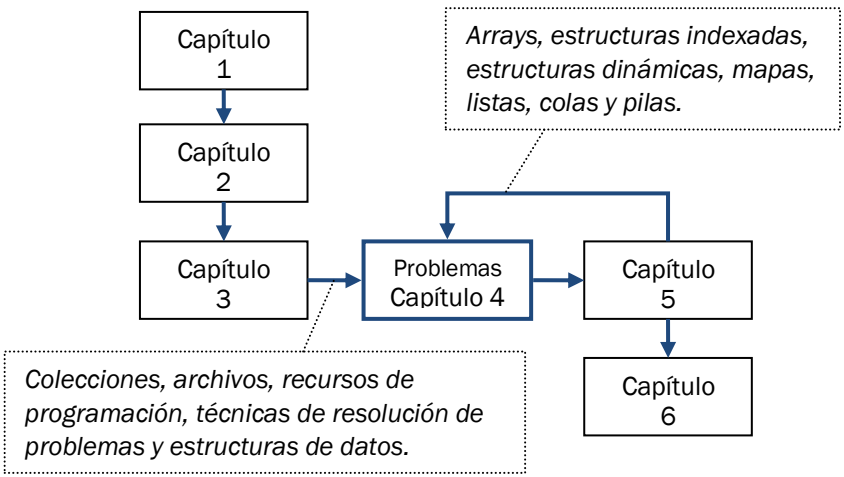

Figura 4.11. Flujo de trabajo y recursos disponibles para utilizar en la resolución de problemas