- Suponiendo que el valor D_{i-1} de la permutación tiene una representación binaria $(d_1, d_2, \ldots, d_n)$, el siguiente valor D_i viene dado por: si $d_1 = 0$ entonces $D_i = (d_2, d_3, \ldots, d_n, 0)$; en otro caso $D_i = (d_2, d_3, \ldots, d_n, 0)$ XOR K.

Por ejemplo, para $B = 8$ podemos tomar $K = 5$ y empezando con $D_1 = 1$ tenemos la siguiente permutación.

D_1	D_2	D_3	D_4	D_5	D_6	D_7
001	010	100	000 $\oplus$101 =101	010 $\oplus$101 =111	110 $\oplus$101 =011	110
1	2	4	5	7	3	6

De nuevo, esta función de redispersión tiene el inconveniente de producir las mismas secuencias de búsqueda para los elementos sinónimos, ya que la permutación es prefijada de antemano. Una alternativa podría ser utilizar distintos valores iniciales para los sinónimos. De esta forma, tenemos que el valor inicial D_1 debería depender de la clave, por ejemplo $D_1(x) = x\ mod\ (B-1) + 1$. Los siguientes valores serían calculados según el algoritmo anterior.

Redispersión con saltos de tamaño variable

Basándonos en la idea de la redispersión con saltos de tamaño C, podemos definir una función similar pero utilizando tamaños de paso distintos para valores de clave distintas. Los tamaños de paso válidos deberían ser primos respecto al tamaño de la tabla B. Si B es un número primo, cualquier C entre 1 y $B-1$ será válido. Con esto, la función de redispersión sería:

$h_i(x) = (h(x) + iC(x))\ mod\ B$

Donde $C(x)$ es una función cualquiera –al estilo de la función de dispersión– que devuelve un valor en el intervalo: $(1, \ldots, B-1)$. Por este motivo, esta técnica se suele conocer también como **dispersión doble**. Por ejemplo, una función válida podría ser $C(x) = 1 + (x^2/D + Ex)\ mod\ (B-1)$.

Eligiendo $C(x)$ de forma adecuada, podemos conseguir que los sinónimos produzcan secuencias de búsqueda distintas, resolviendo así el problema del agrupamiento.

3.3. Combinando estructuras de datos

Hasta ahora hemos estudiado las estructuras de datos como realizaciones en memoria de un tipo abstracto de datos. En la práctica, al diseñar una estructura, tenemos en cuenta cuestiones de eficiencia que son obviadas en el estudio del tipo abstracto. En una aplicación real necesitaremos, posiblemente, hacer adaptaciones de las estructuras estudiadas e incluso combinar varias de ellas en lo que podríamos llamar una *estructura múltiple*. Pero, ¿por qué iba a ser necesario combinar dos estructuras para los mismos datos? Y ¿cómo hacer la combinación?

La eficiencia de una estructura de datos es un concepto relativo: la eficiencia se mide en función de las operaciones que se vayan a utilizar sobre esa estructura. Una estructura

puede resultar eficiente para cierto tipo de operaciones pero muy ineficiente para otro tipo. Por ejemplo, hemos visto que las tablas de dispersión son muy eficientes para acceder a cierto elemento de forma directa según su clave. Pero ¿y si queremos acceder de forma ordenada, por ejemplo buscar el máximo o el mínimo? O ¿cómo podríamos implementar una operación que acceda por un atributo que no sea la clave? Deberíamos acceder a todas las cubetas de la tabla, ver si están ocupadas o no, y en caso de estarlo comprobar el elemento y actuar de forma adecuada. En definitiva, resultarían muy ineficientes para ese tipo de operaciones.

3.3.1. Estructuras de datos múltiples

Si sobre unos mismos datos necesitamos distintos tipos de operaciones, entonces posiblemente no exista una estructura que nos proporcione eficiencia para todas ellas. En ese caso, puede ser adecuado combinar varias estructuras –que referencien a los mismos datos– cada una de las cuales nos proporciona un acceso rápido para cierto tipo de operaciones.

Supongamos, por ejemplo, que en una aplicación de gestión empresarial almacenamos información sobre los empleados. En concreto, tenemos el nombre de cada empleado, su dirección, sueldo, DNI y teléfono (o teléfonos, si tiene más de uno). Las operaciones que más se van a utilizar serán las siguientes:

A) Dado un DNI (A1), un nombre (A2) o un teléfono (A3), devolver todos los datos del empleado correspondiente.
B) Insertar (B1), eliminar (B2) o modificar (B3) un empleado, accediendo por número de DNI.
C) Listar los empleados por sueldo (C1) o por número de DNI (C2).

Para almacenar los datos relativos a un empleado podemos utilizar un tipo de datos Empleado, definido del siguiente modo:

```
tipo
    Empleado = registro
        Nombre: cadena
        Direccion: cadena
        DNI: entero
        Telefonos: Lista[entero]
        Sueldo: entero
    finregistro
```

Pero, realmente el problema no es cómo representar un empleado, sino cómo almacenar el conjunto de todos los empleados de forma que todas las operaciones anteriores se puedan realizar de forma eficiente; es decir, cómo representar los datos.

Distintas estructuras de representación

Analicemos distintas posibles estructuras de representación para el problema.

- Para facilitar las operaciones de tipo (C1) sería interesante almacenar los registros de empleados en una lista *ListaSueldos*: Lista[Empleado], ordenada por valor del campo Sueldo. El listado de los empleados por orden de sueldo se puede hacer fácilmente

recorriendo esta lista. Pero ahora, ¿qué pasa con las operaciones (A), (B) y (C2), de consulta por DNI, nombre o número de teléfono? Por ejemplo, para la (A2) tendríamos que recorrer toda la lista hasta encontrar un elemento cuyo valor del campo Nombre coincida con el buscado. Si tenemos n empleados, necesitaríamos un $O(n)$ en el caso promedio. Pero, es más, para la operación de consulta por número de teléfono, (A3), además de recorrer toda la lista deberíamos consultar la lista Telefonos dentro de cada empleado. El tiempo de ejecución sería un $O(n\ r)$, siendo r el número promedio de teléfonos por empleado. La representación es, por lo tanto, adecuada para un tipo de operación pero ineficiente para los demás.

- Para facilitar las operaciones de consulta por nombre, sería más conveniente tener una tabla de dispersión donde las claves fueran los nombres y los valores fueran de tipo Empleado, por ejemplo *HashNombre*: TablaHash[cadena, Empleado]. La consulta por nombre sería muy rápida; usando un número de cubetas $B \approx n$ podríamos alcanzar un $O(1)$ para esas operaciones. Pero, ¿qué ocurre ahora para la consulta por teléfono, por DNI o listar según el sueldo? Los teléfonos y los sueldos de los empleados están almacenados en la tabla, pero intentar acceder por ellos resultaría muy costoso. Por ejemplo, si usamos dispersión cerrada, deberíamos recorrer todas las cubetas, comprobando las que no están vacías, y recorrer sus listas de teléfonos. El orden de complejidad sería un $O(B + n\ r)$, con $B > n$.

- Para solucionar el problema anterior, podríamos usar una tabla de dispersión abierta *HashTelefonos*: TablaHash[entero, Empleado] donde las claves fueran números de teléfono y los valores de tipo Empleado. Usando ahora una tabla de tamaño $B \approx n$ conseguiríamos implementar la operación de tipo (A3) de forma muy eficiente, en un $O(r)$ en promedio. Pero para todas las demás operaciones deberíamos recorrer la tabla completamente. Por ejemplo, para la operación de consulta por DNI tendríamos que recorrer toda la tabla y buscar un registro con ese DNI. El tiempo de ejecución sería un $O(B + n\ r)$. Además, si un empleado tiene más de un teléfono, se estaría repitiendo información en la tabla.

- Por otro lado, si consideramos las operaciones (A1) y (B), de consulta por DNI, lo más conveniente sería usar una estructura de datos que nos permita acceder rápidamente por DNI, como una tabla de dispersión. El inconveniente ahora es la operación (C2) para listar ordenadamente los DNI. Ya hemos visto que las tablas de dispersión resultan muy ineficientes para este tipo de operaciones. Una buena solución sería utilizar un árbol binario de búsqueda, ArbolDNI, que dado un conjunto de n datos nos proporciona un tiempo $O(\log n)$ para la búsqueda directa y $O(n)$ para el listado ordenado (con un recorrido en in-orden). Pero, nuevamente, optimizar las operaciones que acceden por DNI implica hacer más costosas las operaciones que acceden por otros campos.

Estructura de datos combinada

Está claro que ninguna estructura por separado nos proporciona una buena solución para todos los tipos de operaciones. Cada una está orientada a optimizar cierta clase

de operaciones. La cuestión es ¿sería posible aprovechar lo mejor de cada una de ellas, consiguiendo eficiencia en todas las operaciones? La respuesta es que sí; la solución consiste básicamente en utilizar al mismo tiempo todas las estructuras antes propuestas, haciendo que referencien al mismo conjunto de datos. Es decir, tenemos un mismo conjunto de datos almacenados, pero diferentes estructuras de acceso a los mismos. Esta idea da lugar a lo que se conoce como **estructuras de datos múltiples**. En nuestro caso, la estructura sería como la mostrada en la figura 3.6.

Para no duplicar información, tanto la lista como las tablas de dispersión y el árbol deberían contener punteros o referencias a registros de tipo Empleado. De esta forma, estos datos no están repetidos, sino que cada empleado se almacena en memoria una sola vez. La definición de los tipos sería la siguiente:

```
tipo
    Empleados = registro
        ListaSueldos: Lista[Puntero[Empleado]]
        HashNombres: TablaHash[cadena, Puntero[Empleado]]
        HashTelefonos: TablaHash[entero, Puntero[Empleado]]
        ArbolDNI: ArbolBinarioBusqueda[entero, Puntero[Empleado]]
    finregistro
```

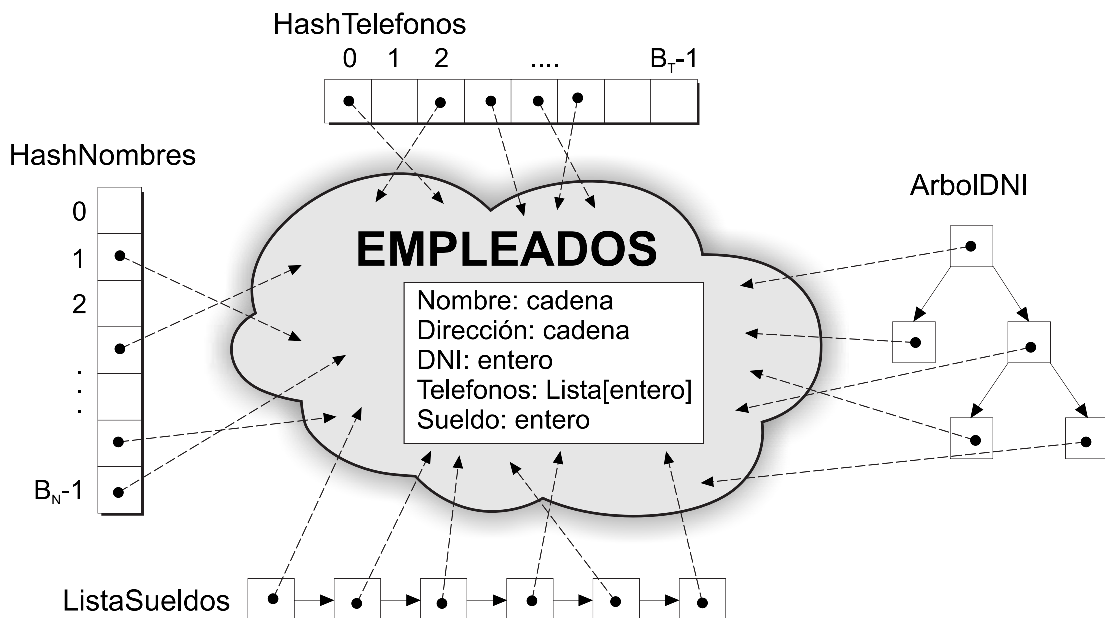

Figura 3.6: Ejemplo de estructura de datos múltiple. Varias estructuras referencian al mismo conjunto de datos almacenados.

A continuación analizamos cómo sería la implementación de las operaciones enumeradas anteriormente. Suponemos que la estructura almacena n empleados, y cada uno tiene en promedio r números de teléfono.

(A1) Consultar por DNI. Buscar el número de DNI en el árbol ArbolDNI, y devolver el puntero del empleado correspondiente. El tiempo de ejecución sería un $O(\log n)$.

(A2) Consultar por nombre. Buscar el nombre en la tabla HashNombres, y devolver el puntero del empleado correspondiente. El tiempo promedio sería un $O(1)$.
(A3) Consultar por teléfono. Buscar el teléfono en la tabla HashTelefonos, y devolver el puntero del empleado correspondiente. El tiempo de ejecución sería un $O(r)$, suponiendo que $B_T \approx n$.
(B1) Insertar un empleado por DNI. En este caso, la inserción implica hacer cuatro inserciones independientes en las cuatro estructuras, con lo que el tiempo viene dado por el tiempo de la más costosa. Deberíamos hacer 1 inserción en HashNombres, con un $O(1)$, r inserciones en HashTelefonos, con $O(r)$ cada una, 1 inserción en ArbolDNI, con $O(\log n)$, y 1 inserción en ListaSueldos, con $O(n)$.
(B2) Eliminar un empleado por DNI. Sería necesario eliminar en las cuatro estructuras, de forma independiente. Para ello, en primer lugar deberíamos acceder a ArbolDNI y obtener los datos del empleado a eliminar. Igual que antes, el tiempo lo determina el tiempo de la eliminación más costosa, que sería la eliminación en ListaSueldos, con un $O(n)$.
(B3) Modificar un empleado por DNI. La modificación sería equivalente a hacer una eliminación y una inserción, con la ventaja de que si un campo no se modifica, nos podemos ahorrar las operaciones correspondientes. Como en el caso anterior, lo primero sería acceder a ArbolDNI, obtener los datos del empleado y comprobar los que se modifican. Si se modifica el nombre tendríamos un $O(\log n)$, si se modifican los teléfonos un $O(\log n + r^2)$, y si es el sueldo (o todos los campos) un $O(n)$.
(C1) Listar ordenadamente por sueldo. El listado se podría hacer fácilmente recorriendo la lista ListaSueldos, y devolviendo los datos de los empleados apuntados. El tiempo de ejecución sería un $O(n)$.
(C2) Listar ordenadamente por DNI. Como ya hemos mencionado, la ordenación se puede conseguir con un recorrido en in-orden del árbol binario ArbolDNI. El tiempo de ejecución sería un $O(n)$.

En definitiva, este tipo de estructuras son lo que se conoce como **estructuras de datos duales** (cuando se combinan dos estructuras) o **múltiples** (cuando se combinan más de dos). La ventaja de este tipo de representaciones –si están bien diseñadas– es que se incrementa la eficiencia de las operaciones en las aplicaciones que requieran distintos modos de acceso. En el mejor caso, la estructura combinada ofrece lo mejor de cada estructura individual. No obstante, es necesario cuidar el diseño ya que podría ocurrir también todo lo contrario, que la estructura combinada tenga lo peor de cada estructura, resultando muy ineficiente.

En cualquier caso, la estructura múltiple implica una duplicación de información, por lo que siempre ocupará más memoria que cada estructura por separado. Además, es necesario mantener la coherencia de los datos almacenados. Como hemos visto, esto afecta particularmente a las operaciones de inserción, modificación o eliminación, que deben trabajar a la vez sobre todas las estructuras, actualizándolas de manera adecuada.

En conclusión, del concepto de estructura de datos se pueden diferenciar dos ideas distintas, aunque normalmente aparecen asociadas: la estructura de los datos almacenados y las estructuras de acceso a esos datos. Sobre un mismo conjunto de datos almacenados podemos tener distintas estructuras de acceso, cada una de las cuales ofrece un acceso adecuado a cierto tipo de operaciones.

3.3.2. La relación muchos a muchos

Ya hemos visto el interés de definir estructuras de datos duales o múltiples, en ciertas aplicaciones. Un tipo especialmente útil son las estructuras de listas múltiples, como por ejemplo las que surgen para representar **relaciones muchos a muchos**.

En el entorno de las bases de datos y los sistemas de información, al estudiar los datos manejados por un sistema se suele distinguir entre *objetos*, o *entidades*, y *relaciones* entre los anteriores. Por ejemplo, en la base de datos de la universidad podemos tener los objetos: estudiante, profesor y asignatura; y las relaciones: estudiante con asignatura, profesor con asignatura, asignatura con asignatura, etc. Las relaciones son siempre entre dos tipos de objetos. A su vez, dentro de estas relaciones distinguimos varias clases según el número de elementos que pueden estar relacionados. Una relación puede ser: uno a uno, uno a muchos o muchos a muchos. Supongamos que los tipos relacionados son A y B.

- En la **relación uno a uno**, un objeto de tipo A está relacionado con un y sólo un objeto de tipo B, y viceversa. Por ejemplo, un profesor (o un estudiante) tiene un DNI, y un DNI es exclusivo de un profesor (o un estudiante).

- En la **relación uno a muchos**, un objeto de tipo A puede estar relacionado con muchos objetos de tipo B, pero el B sólo está relacionado con uno de tipo A. Por ejemplo, la relación entre profesor y asignatura (que podemos llamar “imparte”) es una relación uno a muchos; una asignatura la imparte un sólo profesor (supongamos), pero un profesor puede impartir muchas asignaturas.

- En la **relación muchos a muchos**, un objeto A puede estar relacionado con muchos objetos B, y viceversa. Por ejemplo, las *matrículas* se pueden considerar como relaciones entre estudiantes y asignaturas. Un estudiante está matriculado en muchas asignaturas y una asignatura recibe las matrículas de muchos estudiantes. Por lo tanto, las matrículas son relaciones muchos a muchos.

En la figura 3.7 se muestra un ejemplo concreto de la relación *matrícula*, entre estudiantes y asignaturas. Nuestro interés es diseñar una estructura de datos adecuada a este problema. Buscamos una representación que nos proporcione un acceso eficiente a las matrículas, y al mismo tiempo haciendo un uso razonable de la memoria.

Matrículas / Notas			Asignaturas		
Estudiantes	Id	Nombre	1 Algebra	2 Física	3 AAED
	1	Agapito	5		
	2	Lucas	7	8	
	3	Jhonny		3	4
	4	Pepita		7	

Figura 3.7: Ejemplo de la relación muchos a muchos *matrícula*. Un estudiante está matriculado en muchas asignaturas y una asignatura recibe matrículas de muchos estudiantes.

Utilizando las mismas ideas básicas vistas en representación de conjuntos, llegamos a las dos soluciones directas al problema de las relaciones muchos a muchos: utilizar un

array bidimensional, o matriz, o bien una estructura de listas. En cierto sentido, ambas ideas corresponden a dos estrategias de representación opuestas: representación estática y contigua en memoria, o representación dinámica y enlazada.

Representación mediante matrices

Supongamos que de cada asignatura y estudiante almacenamos el nombre y un identificador, y para cada matrícula tenemos la nota, almacenada como un entero. La definición de los tipos sería la siguiente:

```
tipo
    Estudiante = registro
        nombre: cadena
        id: entero
    finregistro
    Asignatura = registro
        nombre: cadena
        id: entero
    finregistro
    Nota = entero
```

Para simplificar, consideramos que los identificadores de los estudiantes van de 1 a n, y las asignaturas de 1 a m, de manera que se pueden almacenar ambos en sendos arrays. En caso contrario (por ejemplo, si los identificadores de los estudiantes son números de DNI), se podrían usar tablas de dispersión, utilizando los identificadores como claves.

```
Estudiantes: array [1..n] de Estudiante
Asignaturas: array [1..m] de Asignatura
```

La forma más sencilla de representar una relación muchos a muchos es mediante un simple array bidimensional de matrículas. El tamaño de cada dimensión corresponde al número de objetos de cada uno de los dos tipos relacionados. En el caso de la relación matrícula tendremos.

```
tipo
    Matriculas = array [1..n, 1..m] de Nota
```

Si M es de tipo Matriculas, cada posición $M[i, j]$ de la matriz indica la nota del alumno i en la asignatura j, para cada $i = 1..n$ y $j = 1..m$. Podemos usar el valor 0, o algún otro valor especial, en caso de no estar matriculado. En la figura 3.8 se muestra gráficamente la disposición en memoria de los datos usando esta representación.

Estudiantes

	nombre	id
1	Agapito	1
2	Lucas	2
3	Jhonny	3
4	Pepita	4

Asignaturas

	nombre	id
1	Algebra	1
2	Física	2
3	AAED	3

Matrículas

	1	2	3
1	5		
2	7	8	
3		3	4
4		7	

Figura 3.8: Representación de la relación muchos a muchos *matrícula*, usando matrices.

Las operaciones de inserción, eliminación o consulta de una matrícula, dado un identificador de asignatura y de estudiante, son inmediatas. Las tres se consiguen fácilmente en un $O(1)$. Las operaciones de listar los estudiantes matriculados en una asignaturas j, o las asignaturas en las que está matriculado un estudiante i, se realizan mediante un simple recorrido de la columna o fila correspondiente. Por ejemplo, en el primer caso comprobaríamos todos los M[s, j], para $s = 1..n$, listando los que tengan valor no nulo. El orden de complejidad sería claramente un $O(n)$. De forma similar, para el segundo caso el orden sería un $O(m)$.

En cuanto al uso de memoria, nos encontramos con una limitación propia de las representaciones mediante arrays: es necesario conocer el tamaño que tendrán los datos almacenados. Si no lo conocemos a priori, debemos usar un tamaño suficientemente grande para permitir que se añadan nuevos estudiantes o asignaturas. Pero si el tamaño es grande, estaremos desperdiciando memoria.

Es más, puede que estemos desperdiciando memoria aun conociendo el número de asignaturas y estudiantes. Está claro que si cada entero ocupa k_1 bytes, la memoria ocupada será $n\ m\ k_1$ bytes. ¿Es mucho o poco? Supongamos la Universidad de Murcia, que tiene unos 30.000 estudiantes y, digamos, unas 2.000 asignaturas. Además, como queremos tener información de la nota de cada estudiante, la convocatoria en la que está matriculado, etc., los valores del array serán registros que ocupan 4 bytes. La memoria necesaria sería: 30.000*2.000*4 bytes = 240 Mbytes. Realmente no es mucho para toda la universidad, pero ¿cuál es el problema? Normalmente un estudiante no estará matriculado en más de 60 asignaturas. Si representamos sólo las asignaturas en las que un estudiante está matriculado usaríamos: 30.000*60*4 bytes = 7,2 Mbytes. Eso quiere decir que ¡el 97 % de la matriz estará sin utilizar! De toda la memoria reservada sólo usamos el 3 %. Esto es lo que se conoce como una matriz *dispersa* o *escasa*, donde la mayoría de las posiciones tienen valor nulo (o cero, en aplicaciones matemáticas).

Representación mediante listas

Para evitar el desperdicio de memoria, se puede utilizar una representación dinámica mediante listas, en las cuales sólo se almacenan las matrículas que realmente existan. En principio, podemos elegir entre tres posibilidades distintas: una sola lista con todas las matrículas, una lista para cada estudiante con sus matrículas o una lista para cada asignatura con sus matrículas.

```
tipo
    Mat₁ = registro
        id_est: entero
        id_asign: entero
        nota: Nota
    finregistro
    Matriculas₁ = Lista[Mat₁]    // Una lista con todas las matrículas
    Mat₂ = registro
        id_asign: entero
        nota: Nota
    finregistro
```

$\mathsf{Matriculas}_2$ = **array** [1..*n*] **de** Lista[Mat_2] *// Una lista por estudiante*
Mat_3 = **registro**
id_est: entero
nota: Nota
finregistro
$\mathsf{Matriculas}_3$ = **array** [1..*m*] **de** Lista[Mat_3] *// Una lista por asignatura*

En la figura 3.9 se muestra la estructura correspondiente a estas tres posibles representaciones, para el ejemplo del apartado anterior.

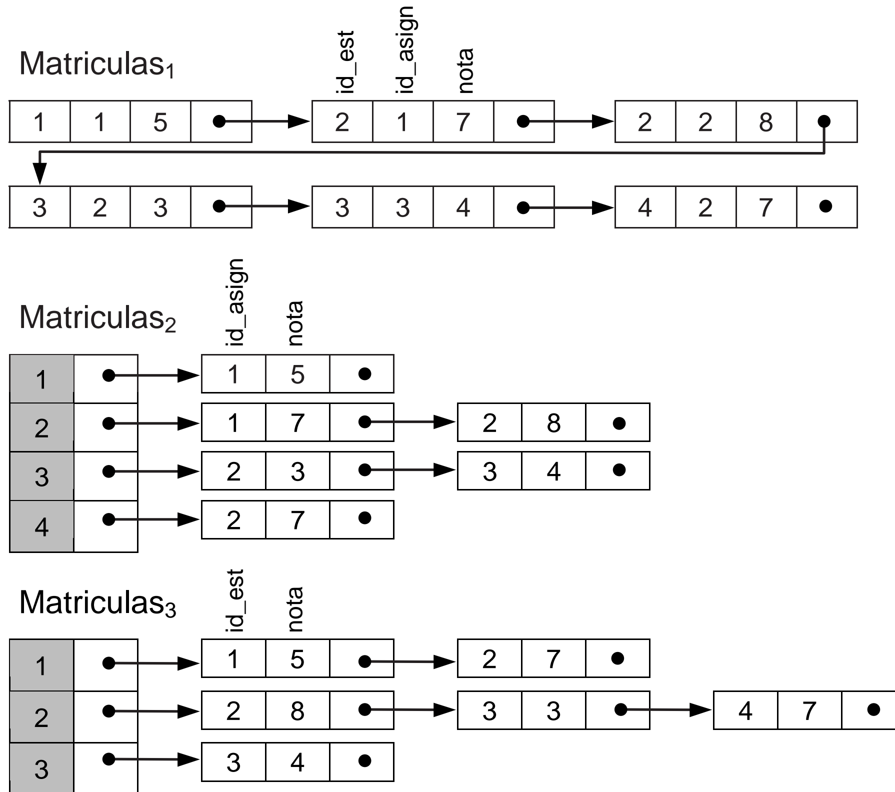

Figura 3.9: Representación de la relación muchos a muchos *matrícula*, usando listas simples. Arriba: usando una sola lista de matrículas. Centro: usando una lista de matrículas por estudiante. Abajo: usando una lista de matrículas por asignatura.

Respecto a la representación con matrices tenemos que, por un lado, se almacenan menos matrículas –únicamente las existentes–, pero por otro lado se utiliza más memoria en las listas, en los punteros y los identificadores adecuados. Si suponemos n estudiantes, m asignaturas, t matrículas, k_1 bytes por entero y k_2 bytes por puntero, en la primera estructura cada registro Mat_1 ocupa $3k_1$ bytes. Cada celda de la lista contiene, además, un puntero por lo que ocuparía $3k_1 + k_2$ bytes. En total, la lista $\mathsf{Matriculas}_1$ ocuparía $t(3k_1 + k_2)$ bytes. Sólo depende del número de matrículas, no del número de estudiantes o asignaturas.

En las otras dos estructuras, los registros Mat_2 y Mat_3 ocupan $2k_1$ bytes. La memoria utilizada por $\mathsf{Matriculas}_2$ y $\mathsf{Matriculas}_3$ sería $t(2k_1 + k_2)$ bytes en las listas, más lo

correspondiente a los arrays: $n\ k_2$ y $m\ k_2$ bytes, respectivamente. Se puede razonar sobre la obtención de estos valores observando la representación de la figura 3.9.

Por ejemplo, considerando los valores del anterior apartado, y enteros y punteros de 4 bytes, la memoria ocupada de las tres representaciones de listas sería 28,8 Mbytes, 21,72 Mbytes y 21,6 Mbytes respectivamente, unas 10 veces menos que con matrices.

El problema es ahora el alto tiempo de ejecución de las operaciones. Con la primera posibilidad, cualquier simple operación de consulta o eliminación de una matrícula implica recorrer toda la lista, dando lugar a un $O(t)$. Con la segunda, cada lista tendrá de media t/n elementos. Consultar una matrícula concreta, dado un identificador de estudiante y de asignatura, requiere un tiempo $O(t/n)$, el mismo que para listar todas las matrículas de un estudiante. Sin embargo, si queremos consultar los alumnos matriculados en una asignatura tendremos que recorrer todas las listas, consumiendo un $O(t + n)$.

Algo parecido pasa con la tercera estructura. Consultar los estudiantes de una asignatura es rápido, pero para saber las asignaturas de un estudiante dado tenemos que recorrer todas las listas.

3.3.3. Estructuras de listas múltiples

Ya hemos visto que la utilización de listas es la única opción viable –en cuanto a uso de memoria– para la representación de matrices escasas, como las que aparecen en las relaciones muchos a muchos. Pero hemos visto también que el uso de listas implica una reducción de la eficiencia de las operaciones.

En el ejemplo de las matrículas de estudiantes en asignaturas, las operaciones necesarias son del tipo: insertar, eliminar o consultar una matrícula, listar las matrículas de un estudiante y listar las matrículas de una asignatura. Si usamos listas de matrículas para cada estudiante, la consulta de matrículas para una asignatura se hace ineficiente, y lo contrario ocurre si las listas son por asignatura. ¿Cuál es la solución? Usando la idea de las estructuras duales, podemos combinar los dos tipos de listas anteriores en una estructura de listas múltiples donde cada matrícula pertenezca a la vez a dos listas: la lista de las demás matrículas de ese estudiante y la lista de las restantes matrículas de esa asignatura.

En general, una estructura de **listas múltiples** es una colección de celdas en la que algunas de ellas pueden pertenecer a más de una lista a la vez. En nuestro caso tenemos:

- Los elementos de la estructura pueden ser de tres tipos distintos: estudiante, asignatura o matrícula.

- Los registros de estudiantes forman una lista y los de asignaturas otra. Además, cada registro de estudiante apunta al primer elemento de la lista de matrículas de ese estudiante, y cada asignatura apunta al primer elemento de la lista de matrículas en esa asignatura.

- Cada elemento de matrícula, que relaciona un estudiante e con una asignatura a, pertenecen al mismo tiempo a dos listas: lista de matrículas del estudiante e y lista de matrículas de la asignatura a. Estas listas son circulares, es decir la última matrícula de un estudiante (o asignatura) apunta al registro de estudiante (o asignatura) correspondiente.

Para definir el tipo de datos podemos utilizar registros con variantes. El tipo enumerado clase_registro indica el tipo de registro en la estructura de listas múltiples. Según esa clase, las variables de tipo tipo_registro tendrán unos u otros atributos.

tipo
  clase_registro = **enumerado** (estudiante, asignatura, matricula)
  tipo_registro = **registro**
    según clase: clase_registro
      estudiante: (id: entero; nombre: cadena; pri_mat, sig_est: Puntero[tipo_registro])
      asignatura: (id: entero; nombre: cadena; pri_mat, sig_asi: Puntero[tipo_registro])
      matricula: (nota: Nota; sig_est, sig_asi: Puntero[tipo_registro])
    finsegún
  finregistro

En la figura 3.10 se muestra gráficamente la disposición en memoria de los registros en la estructura de listas múltiples.

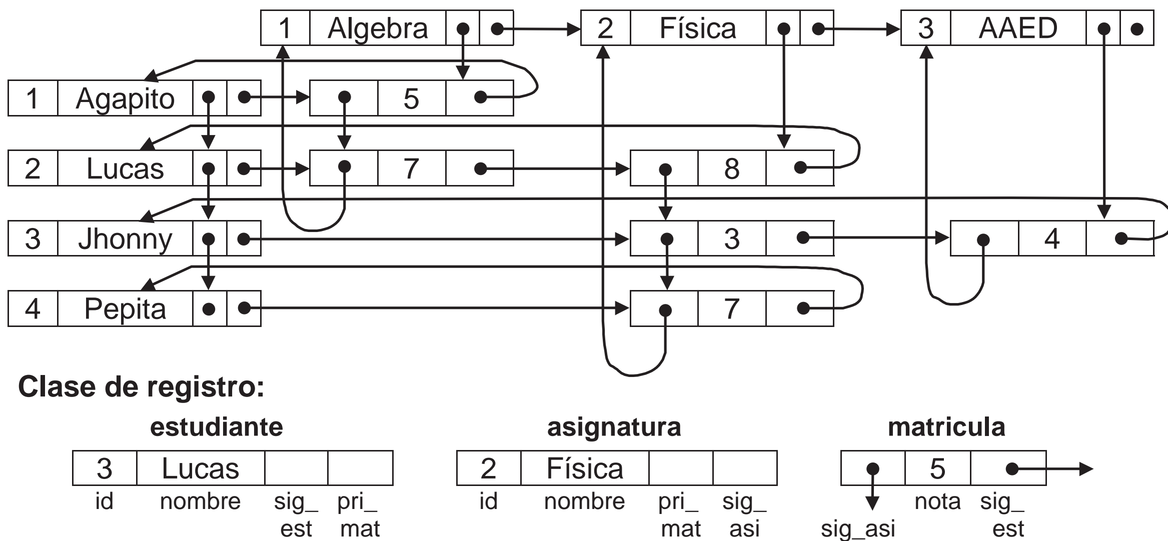

Figura 3.10: Representación de la relación muchos a muchos *matrícula*, usando listas múltiples. Se muestra abajo el significado de cada campo para cada uno de los tres tipos de registro.

Implementación de las operaciones

Supongamos que las operaciones de consulta o manipulación de matrículas reciben como parámetros punteros a los registros de estudiante o asignatura correspondientes[5]. Para buscar la matrícula asociada a un estudiante e en una asignatura a deberíamos recorrer las matrículas del estudiante e, y para cada una comprobar si está en la asignatura buscada. Para esto hacemos uso del hecho de que las listas son circulares.

[5]En caso contrario, se deberían recorrer las listas de estudiantes o de asignaturas; o bien podríamos usar sendas tablas de dispersión donde las claves sean identificadores o nombres, y los valores punteros a los registros correspondientes. La elección depende del tipo de las operaciones requeridas.

```
operación CalculaNota (e, a: Puntero[tipo_registro]): Nota
    m:= e↑.pri_mat
    mientras m ≠ e hacer
        tmp:= m↑.sig_asi
        mientras tmp↑.clase ≠ asignatura hacer
            tmp:= tmp↑.sig_asi
        finmientras
        si tmp = a entonces
            devolver m↑.nota
        finsi
        m:= m↑.sig_est
    finmientras
    error("Matrícula no existente")
```

Si nos fijamos en la figura 3.10, vemos que el anterior algoritmo utiliza la variable *m* para recorrer las listas horizontalmente, empezando por *e*, mientras que con *tmp* las recorreremos en sentido vertical. De forma similar, para listar los nombre de los estudiantes matriculados en una asignatura *a* deberíamos recorrer las matrículas de esa asignatura y encontrar el nombre del estudiante asociado a cada registro de matrícula.

```
operación ListarEstudiantes (a: Puntero[tipo_registro])
    m:= a↑.pri_mat
    mientras m ≠ a hacer
        tmp:= m↑.sig_est
        mientras tmp↑.clase ≠ estudiante hacer
            tmp:= tmp↑.sig_est
        finmientras
        escribir("Nombre del estudiante: ", tmp↑.nombre)
        m:= m↑.sig_asi
    finmientras
```

Evaluación de la eficiencia

En cuanto a la utilización de memoria, si consideramos sólo la utilizada en las matrículas (para comparar, en igualdad de condiciones, con las estructuras anteriores) tenemos que cada registro de matrícula contiene un entero y dos punteros, es decir k_1+2k_2 bytes. En total, necesitamos $t(k_1+2k_2)$ bytes para las matrículas. Para los punteros de los registros de estudiantes y asignaturas necesitaríamos adicionalmente $2k_2(n+m)$ bytes.

La utilización de memoria está dentro de los mismos valores que aparecían en las estructuras de listas simples. Para el ejemplo concreto manejado en los anteriores apartados, la memoria necesaria por los registros de matrículas sería de 21,6 Mbytes.

Por otro lado, el tiempo de ejecución de las operaciones mejora en algunos casos y empeora en otros. Por ejemplo, la operación CalculaNota debe recorrer una lista de matrículas horizontalmente. Para cada elemento recorre la lista correspondiente en vertical. Si tenemos t matrículas, entonces las listas horizontales serán, en promedio, de tamaño t/n y las verticales t/m. El tiempo de la operación CalculaNota en caso de no encontrarse la matrícula buscada estaría en $O(t^2/(nm))$. Si llamamos $p = t/(nm)$, al porcentaje de

matrículas existentes en relación al máximo posible (en el ejemplo anterior el 3 %), el orden de complejidad se puede expresar como $O(t\ p)$.

Si nos fijamos en la operación ListarEstudiantes, su funcionamiento es básicamente el mismo, cambiando únicamente el tratamiento que se hace después del bucle interno. En este caso para una lista vertical, de tamaño promedio t/m, se recorre una lista horizontal, de tamaño promedio t/n. Nuevamente, el tiempo sería un $O(t^2/(nm)) = O(t\ p)$. Lo mismo tendríamos para una operación ListarAsignaturas, para mostrar las asignaturas en las que está matriculado un estudiante, o para una operación de eliminar, insertar o modificar una matrícula dada.

Si cada alumno tiene al menos una matrícula, entonces $n < t/n < t\ p < t$, y $O(t) = O(t + n)$; si cada asignatura tiene como mínimo un estudiante, entonces $m < t/m < t\ p < t$, y $O(t) = O(t + m)$. De acuerdo con esto, la estructura de listas múltiples, con un O($t\ p$), supone un término medio entre las estructuras de listas, que oscilan entre $O(t/n)$ ó $O(t/m)$ y $O(t + n)$ ó $O(t + m)$.

Es más, es posible mejorar la eficiencia de la estructura de listas múltiples, a costa de usar más memoria. Para ello, en la definición del tipo tipo_registro cambiamos los atributos de matricula para añadir también el identificador del estudiante y de la asignatura correspondientes. La definición sería:

```
tipo
    tipo_registro = registro
        ...
        matricula: (nota: Nota; id_est, id_asi: entero; sig_est, sig_asi: Puntero[tipo_registro])
    finregistro
```

Con esta definición estamos añadiendo información redundante. Pero, en compensación, esa información nos puede permitir ahorrar muchos pasos de recorrido. Por ejemplo, para buscar una matrícula concreta del estudiante e en la asignatura a, sólo tendríamos que recorrer la lista horizontal del estudiante e y para cada matrícula comprobar el campo id_asi. El tiempo sería ahora $O(t/n)$, en lugar de $O(t\ p)$. Lo mismo pasaría con las operaciones para listar asignaturas o estudiantes.

La memoria que necesitaríamos con la versión redundante de las listas múltiples sería $t(3k_1 + 2k_2)$ bytes. En el ejemplo concreto, pasamos de 21,6 Mbytes a 36 Mbytes. No obstante, el ahorro de tiempo puede merecer la pena.

En la tabla 3.2 se muestra comparativamente la memoria y el tiempo de ejecución de algunas operaciones, para las distintas estructuras de representación de matrices escasa vistas en esta sección. Se puede concluir que las listas múltiples ofrecen unos valores de eficiencia razonables, con un uso de memoria reducido.

Ejercicios resueltos

Ejercicio 3.1 Para almacenar conjuntos de números enteros usamos tablas de dispersión. Como se espera que los conjuntos representados sean pequeños, definimos el tamaño de tabla $B = 15$. La función de dispersión es $h(x) = \lfloor x^2/10 \rfloor\ mod\ 15$. Mostrar las tablas de dispersión resultantes de insertar (en este orden) los elementos: 72, 23, 4, 5, 12, 25, 6, 2, 33, 24, 49, 74, usando las siguientes técnicas: