3 CONJUNTOS, BOLSAS Y FUNCIONES. FUNCIONES DE DISPERSIÓN

3.1 EL TAD CONJUNTO.

El TAD Conjunto es una colección de elementos distintos (todos del mismo tipo) junto con una serie de procedimientos de acceso.

Veamos la diferencia entre esta definición y la del TAD Lista. Los elementos en una lista no tienen porqué ser distintos, y tienen un orden lineal impuesto que refleja el orden en que fueron insertados en la lista. Por el contrario, en un conjunto no se permiten elementos duplicados, y el orden en que los elementos son insertados no es significativo.

Por otro lado, los elementos que forman parte de un conjunto, pueden especificarse de dos formas:

a) Por enumeración explícita, que se hará como lo haremos nosotros, aunque sólo sirve para conjuntos finitos.

b) Para conjuntos infinitos -que no vamos a tratar directamente- se puede dar una propiedad o predicado P, tal que:

$$\forall a\ P(a) \Leftrightarrow a \in Cjto$$

Aunque no lo parezca, en computación hay casos en los que se emplea el método b) de especificación de conjuntos, como p.ej. en una consulta SQL, en la que se retorna el cjto. de tuplas que verifican una cierta propiedad -si se usa la cláusula DISTINCT-, o una bolsa o multiconjunto -en caso contrario-.

3.1.1 ESPECIFICACIÓN ALGEBRAICA

Como en los TADs lineales, tenemos dos operadores constructores, Crear e Incluir,que son usados para crear un conjunto vacío e insertar un elemento en un conjunto respectivamente. Tenemos dos predicados , uno que comprueba si el conjunto es vacío (Es_Vacío) y otro que comprueba si contiene un determinado elemento (Pertenece). Tenemos una función selectora que devuelve un conjunto del que ha sido extraído un determinado elemento (Eliminar). Hemos añadido otras funciones, como Unión, Intersección o Diferencia. Se ha incluído asimismo una operación Elemento que no debe confundirse con el tipo base Elemento.

tipo Conjunto
dominios Elemento, Lógico, $\mathbb{N}$
generadores
Crear: $\longrightarrow$ Conjunto
Agregar: Elemento $\times$ Conjunto $\longrightarrow$ Conjunto
constructores
Eliminar: Elemento $\times$ Conjunto $\longrightarrow$ Conjunto
Union: Conjunto $\times$ Conjunto $\longrightarrow$ Conjunto
Interseccion: Conjunto $\times$ Conjunto $\longrightarrow$ Conjunto

Diferencia: Conjunto × Conjunto ⟶ Conjunto

selectores

Pertenece: Elemento × Conjunto ⟶ Lógico
⊆: Conjunto × Conjunto ⟶ Lógico
Es_Vacio: Conjunto ⟶ Lógico
Cardinal: Conjunto ⟶ ℕ
Elemento: Conjunto ⇸ Elemento

precondiciones c:Conjunto

Elemento(c) : not Es_Vacio(c)

ecuaciones i, j:Elemento; c:Conjunto

Agregar(i, Agregar(j, c)) == Agregar(j, Agregar(i, c))
Agregar(i, Agregar(i, c)) == Agregar(i, c)
Es_Vacio(Crear) == V
Es_Vacio(Agregar(i, c)) == F
Pertenece(i, Crear) == F
Pertenece(i, Agregar(j, c)) == SI i = j ENTONCES
V
SI NO
Pertenece(i, c)
⊆(Crear, c) == V
⊆(Agregar(i, c_1), c_2) == SI Pertenece(i, c_2) ENTONCES
⊆(c_1, c_2)
SI NO
F
Eliminar(i, Crear) == Crear
Eliminar(i, Agregar(j, c)) == SI i = j ENTONCES
Eliminar(i, c)
SI NO
Agregar(j, Eliminar(i, c))
Cardinal(Crear) == 0
Cardinal(Agregar(i, c)) = 1 + Cardinal(Eliminar(i, c))
Pertenece(Elemento(c), c) == V
Union(Crear, c) == c
Union(Agregar(i, c_1), c_2) == Agregar(i, Union(c_1, c_2)
Interseccion(Crear, c) == Crear
Interseccion(Agregar(i, c_1), c_2) == SI Pertenece(i, c_2) ENTONCES
Agregar(i, Interseccion(c_1, c_2))
SI NO
Interseccion(c_1, c_2)
Diferencia(c, Crear) == c
Diferencia(c_1, Agregar(i, c_2)) == Diferencia(Eliminar(i, c_1), c_2)

fin

De fundamental importancia es la operación Elemento; se trata de una función parcial que retorna un elemento de un conjunto. Sin embargo, el elemento que retorna puede ser cualquiera,

de ahí que sea una operación no especificada completamente; esta inespecificación queda reflejada en que un término que comienza por la operación Elemento(), no podrá ser reducido jamás a una forma canónica del tipo Elemento. No obstante, aplicada al mismo conjunto, devuelve siempre el mismo elemento, ya que de otro modo, si pudiese devolver elementos distintos en cada llamada, podríamos llegar a la conclusión a través de ecuaciones, de que elementos conceptualmente distintos son iguales; de ahí que, aunque el resultado de su primera "ejecución" es no determinista, "ejecuciones" posteriores sí lo serán, considerándose la operación como no determinista. Por tanto, la inespecificación viene dada sólo por el hecho de que *no se sabe qué elemento es el que se retorna, aunque* por tratarse de una especificación algebraica, *se tiene que para un mismo conjunto, el elemento retornado es siempre el mismo*. De esta forma, si queremos recorrer todos los elementos de un conjunto será imprescindible tomarlos de uno en uno con Elemento(), e ir extrayendo los elementos consultados del conjunto, para que, de esta forma, la operación Elemento no repita los valores indefinidamente.

3.1.1.1 Reutilización de especificaciones.

En este punto se va a incluir una nueva forma de hacer especificaciones. De hecho, vamos a representar un conjunto haciendo uso a su vez del TAD Lista. Para ello, usaremos como generador una operación auxiliar oculta Conversor, que toma una lista como entrada (sin necesidad de la restricción de que no posea elementos repetidos), y la considera un conjunto. El resto de operaciones se hacen en base a las operaciones propias de las listas. En este caso, no podemos demostrar algebraicamente que dos conjuntos son iguales por el mero hecho de ser listas de permutaciones de la misma secuencia de elementos. Por ello, es necesario incluir una operación de igualdad que solucione la cuestión.

Este método permite definir TADes nuevos en base a la especificación de otro TAD cuyo comportamiento sea más conocido.

tipo Conjunto
dominios Lista, Elemento, Lógico, $\mathbb{N}$
generadores auxiliares
Conversor: Lista $\longrightarrow$ Conjunto
constructores
Crear: $\longrightarrow$ Conjunto
Agregar: Elemento × Conjunto $\longrightarrow$ Conjunto
Eliminar: Elemento × Conjunto $\longrightarrow$ Conjunto
Union: Conjunto × Conjunto $\longrightarrow$ Conjunto
Interseccion: Conjunto × Conjunto $\longrightarrow$ Conjunto
Diferencia: Conjunto × Conjunto $\longrightarrow$ Conjunto
selectores
Pertenece: Elemento × Conjunto $\longrightarrow$ Lógico
$\subseteq$: Conjunto × Conjunto $\longrightarrow$ Lógico
=: Conjunto × Conjunto $\longrightarrow$ Lógico
Es_Vacio: Conjunto $\longrightarrow$ Lógico

Cardinal: Conjunto $\longrightarrow \mathbb{N}$
Elemento: Conjunto $\not\longrightarrow$ Elemento

auxiliares

Pertenece$_l$: Elemento × Lista $\longrightarrow$ Lógico

precondiciones c:Conjunto

Elemento(c) : not Es_Vacio(c)

ecuaciones i, j:Elemento; c, c_1, c_2:Conjunto l:Lista

Pertenece$_l$(i, Crear) == F
Pertenece$_l$(i, Construir(j, l)) == (i = j) or (Pertenece$_l$(i, l))
Conversor(Construir(i, Construir(j, l))) ==
SI i = j ENTONCES
Conversor(Construir(i, l))
SI NO
Conversor(Construir(j, Construir(i, l)))
Crear == Conversor(Crear)
Agregar(i, Conversor(l)) == Conversor(Construir(i, l))
Es_Vacio(Conversor(l)) == Es_Vacía(l)
Pertenece(i, Conversor(l)) == Pertenece$_l$(i, l)
$\subseteq$(Conversor(Crear), c) == V
$\subseteq$(Conversor(Construir(i, l), c) == Pertenece(i, c) and $\subseteq$(Conversor(l), c)
=(c_1, c_2) == $\subseteq$(c_1, c_2) and $\subseteq$(c_2, c_1)
Eliminar(i, Conversor(Crear)) == Conversor(Crear)
Eliminar(i, Conversor(Construir(j, l))) == SI i = j ENTONCES
Eliminar(i, Conversor(l))
SI NO
Agregar(j, Eliminar(i, Conversor(l)))
Cardinal(Conversor(Crear)) == 0
Cardinal(Conversor(Construir(i, l))) == 1 + Cardinal(Eliminar(i, Conversor(l)))
Pertenece(Elemento(c), c) == V
Union(Conversor(Crear), c) == c
Union(Conversor(Construir(i, l)), c) == Agregar(i, Union(Conversor(l), c))
Interseccion(Conversor(Crear), c) == Conversor(Crear)
Interseccion(Conversor(Construir(i, l)), c) ==
SI Pertenece(i, c) ENTONCES
Agregar(i, Interseccion(Conversor(l), c))
SI NO
Interseccion(Conversor(l), c)
Diferencia(c, Conversor(Crear)) == c
Diferencia(c, Conversor(Construir(i, l))) == Diferencia(Eliminar(i, c), Conversor(l))

fin

3.1.2 APLICACIONES

Al no tener impuesta una estructura determinada, el TAD Conjunto es más general y tiene muchas más aplicaciones. Por otra parte, esta generalidad implica una manipulación más

complicada. Los conjuntos suelen ser usados para representar atributos de determinados objetos; son usados también para comprobar un valor está dentro de un conjunto de posibles valores. Otra aplicación es la implementación del TAD Grafo.

3.2 EL TAD BOLSA.

El TAD Bolsa es una colección de un número arbitrario de elementos (todos del mismo tipo) que no son necesariamente distintos junto con una serie de procedimientos de acceso. La *multiplicidad* de un elemento es el número de veces que dicho elemento aparece en la bolsa.

Como un ejemplo de un problema que hace uso del TAD Bolsa (supongamos bolsas de números cardinales), tomemos tres números naturales, encontremos las bolsas de sus factores primos, su máximo común divisor y mínimo común múltiplo. Por ejemplo, dados los números 30, 78 y 84, denotemos las bolsas de sus factores primos como X, Y y Z respectivamente; así pues tenemos: X = {2, 3, 5}, Y = {2, 3, 13} y Z = {2, 2, 3, 7}. Los factores del m.c.d. vienen dados por la intersección de las tres bolsas, es decir:

m.c.d. = $X \cap Y \cap Z$
= $\{2, 3, 5\} \cap \{2, 3, 13\} \cap \{2, 2, 3, 7\}$
= $\{2, 3\} \cap \{2, 2, 3, 7\}$
= $\{2, 3\}$
= 6

Los factores del m.c.m. vienen dados por la diferencia simétrica de la unión de las tres bolsas con su intersección.

m.c.m. = $(X \cup Y \cup Z) - (X \cap Y \cap Z)$
= $(((X \cup Y) - (X \cap Y)) \cup Z) - (((X \cup Y) - (X \cap Y)) \cap Z)$

$(((X \cup Y) - (X \cap Y)) \cup Z)$ =
= $((((\{2, 3, 5\} \cup \{2, 3, 13\}) - (\{2, 3, 5\} \cap \{2, 3, 13\})) \cup \{2, 2, 3, 7\})$
= $((((\{2, 2, 3, 3, 5, 13\}) - (\{2, 3, 5\} \cap \{2, 3, 13\})) \cup \{2, 2, 3, 7\})$
= $((((\{2, 2, 3, 3, 5, 13\}) - (\{2, 3\})) \cup \{2, 2, 3, 7\})$
= $(\{2, 3, 5, 13\} \cup \{2, 2, 3, 7\})$
= $(\{2, 2, 2, 3, 3, 5, 7, 13\}$

$(((X \cup Y) - (X \cap Y)) \cap Z)$ =
= $((((\{2, 3, 5\} \cup \{2, 3, 13\}) - (\{2, 3, 5\} \cap \{2, 3, 13\})) \cap \{2, 2, 3, 7\})$
= $(\{2, 3, 5, 13\} \cap \{2, 2, 3, 7\})$
= $\{2, 3\}$

m.c.m. =
= $\{2, 2, 2, 3, 3, 5, 7, 13\} - \{2, 3\}$
= $\{2, 2, 3, 5, 7, 13\}$
= 5460

3.2.1 ESPECIFICACIÓN ALGEBRAICA DE BOLSA O MULTICONJUNTO.

La especificación es muy parecida a la del TAD Conjunto. Entre las diferencias podemos destacar que:

- Si hay elementos repetidos, permanecen todas las ocurrencias. (Recordemos que en los conjuntos sólo quedaba una de las ocurrencias, a través de la segunda ecuación en los conjuntos).
- La operación Eliminar sólo elimina una ocurrencia del elemento que sea.

- La operación Cardinal se define de forma recursiva, simplemente contando el número de veces que aparece la operación Agregar.

tipo Bolsa;
dominios Elemento, Lógico, $\mathbb{N}$
generadores
Crear: $\longrightarrow$ Bolsa
Agregar: Elemento × Bolsa $\longrightarrow$ Bolsa
constructores
Eliminar: Elemento × Bolsa $\longrightarrow$ Bolsa
selectores
Elemento: Bolsa $\not\longrightarrow$ Elemento
Pertenece: Elemento × Bolsa $\longrightarrow$ Lógico
Es_Vacia: Bolsa $\longrightarrow$ Lógico
Cardinal: Bolsa $\longrightarrow$ $\mathbb{N}$
NumRep: Elemento × Bolsa $\longrightarrow$ $\mathbb{N}$
precondiciones b:Bolsa
pre Elemento(b) : not Es_Vacia(b)
ecuaciones i, j:Elemento; b:Bolsa
Agregar(i, Agregar(j, b)) == Agregar(j, Agregar(i, b))
Es_Vacia(Crear) == V
Es_Vacia(Agregar(i, b)) == F
Pertenece(i, Crear) == F
Pertenece(i, Agregar(j, b)) == SI i=j ENTONCES
V
SI NO
Pertenece(i, b)
Eliminar(i, Crear) == Crear
Eliminar(i, Agregar(j, b)) == SI i=j ENTONCES
b
SI NO
Agregar(j, Eliminar(i, b))
Pertenece(Elemento(b), b) == V
Cardinal(Crear) == 0
Cardinal(Agregar(i, b)) == 1 + Cardinal(b)
NumRep(i, Crear) == 0

```
NumRep(i, agregar(j, b)) ==  SI i=j ENTONCES
                                  1 + NumRep(i, b)
                             SI NO
                                  NumRep(i, b)
```

fin

3.3 IMPLEMENTACIÓN DE CONJUNTOS EN MODULA 2.

Podemos implementar los conjuntos de 5 formas principales, a saber:

- Tipo SET de Modula 2.
- Tabla de elementos de tipo BOOLEAN.
- Tabla de bits.
- Listas de elementos.
- Listas ordenadas de elementos.

3.3.1 TIPO SET EN MODULA 2.

Consiste en hacer uso del tipo conjunto que ya trae incorporado el propio lenguaje Modula 2. Esta utilización nos obliga a que el tipo base del conjunto sea un tipo cuyo número de valores o estados sea inferior a MAX(INTEGER) + 1, lo que equivale a decir cualquier ordinal, como pueda ser un enumerado, o un subrango.

La definición viene a sería algo así como:

```
TYPE
        TIPO_BASE = [0..20];
        CONJUNTO = SET OF TIPO_BASE;
```

Y la asignación, algo como:

```
VAR
        C1 : CONJUNTO;
BEGIN
        C1 := CONJUNTO{1, 3, 8, 2, 14};
```

Las operaciones entre conjuntos vienen dadas por:

+	UNIÓN.
-	DIFERENCIA
*	INTERSECCIÓN.
/	DIFERENCIA SIMÉTRICA.

Los operadores relacionales son:

=	IGUAL.
#	DIFERENTE.
<=	INCLUIDO EN.
>=	CONTIENE A.
IN	ES MIEMBRO DE.

Además, el operador Agregar() viene dado por: INCL(Conjunto, Elemento), y el operador Eliminar, por EXCL(Conjunto, Elemento).

Veremos como gestiona Modula 2 este tipo de construcciones.

3.3.1.1 ¿CÓMO IMPLEMENTA MODULA-2 LOS CONJUNTOS?. VECTORES DE BITS.

En el Modula-2 de TopSpeed, los conjuntos son representados como mapas de bits empaquetados (packed bit-maps) con un bit para cada miembro potencial en el tipo elemento, por ejemplo [0..N]. El número de bytes requerido para representar el conjunto viene dado por ⌈N / 8⌉. Por ejemplo, si tenemos un conjunto que puede tener hasta cien elementos, necesitamos 13 bytes -es decir, ⌈100 / 8⌉- para representarlo. El elemento **e** del conjunto, siendo $0 \leq e \leq N$, es representado por el bit e MOD 8 en el byte número e DIV 8. La fórmula supone que empezamos contando con el byte 0 y con el bit 0.

En el Modula-2 de TopSpeed, el tipo base de un conjunto puede ser CARDINAL, SHORTCARDINAL, CHAR, un tipo enumerado, o un subrango de cualquiera de estos tipos. Un conjunto puede tener hasta 65.536 elementos.

Es la forma más simple de implementar conjuntos. Si todos los conjuntos son subconjuntos de un conjunto pequeño de elementos, podemos representar el conjunto por un vector de bits donde tenemos un bit por cada uno de los elementos pertenecientes al conjunto; si el elemento pertenece al conjunto su bit correspondiente estará puesto a 1, y si no estará a 0. Para que nuestra definición sea uniforme supongamos que el conjunto universal contiene elementos 1, 2, 3, ..., N-1, N; es decir, enteros positivos entre 1 y N, siendo por tanto N el tamaño del conjunto universal. Así pues, representamos el conjunto como un vector de bits (secuencia de bits) donde el i-ésimo bit es un 1 si el elemento i pertenece al conjunto, y 0 en caso contrario. Si N=10, tenemos las siguientes representaciones:

S = {1, 3, 5}	< 1 0 1 0 1 0 0 0 0 0 >
T = {3, 8, 10, 9}	< 0 0 1 0 0 0 0 1 1 1 >
U = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	< 1 1 1 1 1 1 1 1 1 1 >
V = {}	< 0 0 0 0 0 0 0 0 0 0 >

3.3.2 TABLAS DE ELEMENTOS DE TIPO BOOLEAN.

Este caso es muy simple. Se trata de definir una tabla de valores lógicos, en la que los índices vienen dados por los posibles valores que pueden incluirse en el conjunto. Si un elemento está o no en el conjunto vendrá determinado por el valor lógico que contenga en su casilla. Cualquier implementación en base a tablas como ésta, nos obligará a que el tipo base de nuestro conjunto tenga un número de valores finito, y a ser posible, pequeño, y además debe ser un tipo tal que permita indexar un array. Con esta técnica se pretende tener un valor Verdadero o Falso asociado a cada posible elemento del tipo base, que nos dirá si ese elemento forma parte o no del conjunto. Así, tendremos tantos valores lógicos como elementos puedan formar parte del conjunto.

P. ej., y siguiendo el caso anterior, la definición sería:

```
TYPE
        TIPO_BASE = [0..20];
        CONJUNTO = ARRAY TIPO_BASE OF BOOLEAN;
```

Y la asignación, algo como:

```
VAR
        C1 : CONJUNTO;
BEGIN
        CONJUNTO[1] := TRUE;
        CONJUNTO[3] := TRUE;
        CONJUNTO[8] := TRUE;
        CONJUNTO[2] := TRUE;
        CONJUNTO[14] := TRUE;
```

Dado que se trata de una definición propia, las operaciones tendremos que definirlas por nuestra cuenta. Por supuesto, el numero de elementos que pueda contener un conjunto vendrá dado por las limitaciones que Modula 2 impone en cuanto a la longitud de matrices. Estas restricciones son las mismas que en el caso anterior.

Como puede observarse, el tamaño de un conjunto del mismo tipo base es siempre el mismo, tanto si contiene muchos como pocos o ningún elemento. Esto mismo ocurre en el caso de los conjuntos tal y como los suministra Modula 2.

3.3.3 LISTAS DE ELEMENTOS.

Este es el modelo de representación más versátil en el sentido de que permite representar conjuntos de cualquier tipo base, como puedan ser incluso registros, matrices, incluso punteros a estructuras complejas.

Se trata simplemente de mantener una lista con todos los elementos que componen al conjunto en un momento determinado. En esta lista no importará el orden en que se almacenen los elementos, y tendrá las restricciones propias de los conjuntos: no repetición de elementos, y existencia del conjunto vacío.

Así, el tipo se definiría tal como:

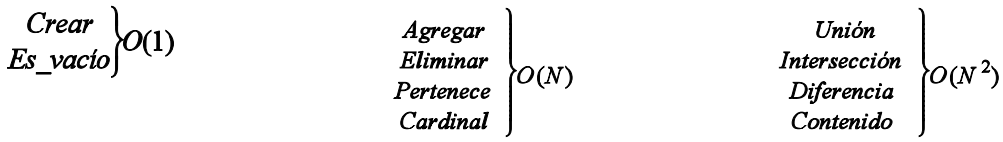
```
TYPE
        CONJUNTO = LISTA;
```

y las operaciones del conjunto vendrían dadas en función de las de la lista.

Otra ventaja, además de que abarca a cualquier tipo base, es que su tamaño no es constante, sino que depende del número de elementos que el conjunto contenga en un momento dado, (siempre y cuando la lista esté implementada de forma dinámica, y su tamaño sólo esté acotado por el de la memoria disponible).

Así, las listas ordenadas se usan en caso de que los conjuntos sean pequeños, pues en caso contrario, cualquier operación sería costosa de ejecutar y poco eficiente.

Los órdenes de cada operación serían:



3.3.4 LISTAS ORDENADAS DE ELEMENTOS.

Cuando entre los elementos que van a formar parte del conjunto existe una relación de orden total, (o bien el tipo base es un registro, y esta relación existe para algún campo de éste), es posible representar un conjunto como una lista ordenada de elementos del tipo base.

Así, el costo de cada operación vendrá determinado por la representación interna que se haya escogido para la lista. P.ej., en el caso de añadir un elemento, sería más fácil hacerlo con una búsqueda dicotómica, que a través de una inserción directa; sin embargo, las listas dinámicas con punteros impiden las búsquedas dicotómicas, ya que no sabemos en un momento dado la posición de todos y cada uno de los elementos que conforman la lista. Así, dependiendo de la estructura interna, podemos tener implementaciones de O(N) u O(lg N), ($lg \equiv lg_2$).

Las operaciones entre conjuntos, pueden pasar a un orden de O(N), mediante el uso de mezclas de las listas que representan a los conjuntos. P. ej., en el caso de la unión, el procedimiento sería así:

```
TYPE
        CONJUNTO = POINTER TO NODO;
        NODO = RECORD
                        Contenido : INTEGER;
                        Siguiente : POINTER TO NODO;
                END;
PROCEDURE Unión(c1, c2 : CONJUNTO) : CONJUNTO;
VAR
        Resultado : CONJUNTO;
        Último : POINTER TO NODO;
        Nuevo : POINTER TO NODO;
BEGIN
        Resultado := NIL;
        Último := NIL;
        WHILE (c1 # NIL) OR (c2 # NIL) DO
                NEW(Nuevo);
                Nuevo^.Siguiente := NIL;
                IF (c2 = NIL) OR ((c1 # NIL) AND (c1^.Contenido < c2^.Contenido)) THEN
                        Nuevo^.Contenido := c1^.Contenido;
                        c1 := c1^.Siguiente;
                ELSE
                        Nuevo^.Contenido := c2^.Contenido;
                        c2 := c2^.Siguiente;
                END;
                IF Último = NIL THEN
                        Resultado := Nuevo;
                        Último := Nuevo;
                ELSIF Último^.Contenido < Nuevo^.Contenido THEN
                        Último^.Siguiente := Nuevo;
                        Último := Nuevo;
                ELSE
                        DISPOSE(Nuevo);
                END;
        END;
END Unión;
```

Las operaciones de intersección y diferencia se pueden hacer de forma similar.

3.3.5 IMPLEMENTACIONES.

a continuación vemos una implementación de conjuntos, usando para ello una lista, y las funcionalidades que ésta suministra.

Nótese que se ha incluído el procedimiento Elemento, para conseguir de alguna forma saber cuáles son los elementos que componen en conjunto. Para ello, se supone que cada elemento tiene una etiqueta numérica, donde los valores de las etiquetas van del 1..Cardinal(j). Esta etiqueta es precisamente el segundo parámetro que se le pasa a Elemento(). Si suponemos que dicho segundo parámetro es más bien una posición que una etiqueta, estamos cometiendo una incorrección, ya que entre los elementos de un conjunto no existe orden alguno.

Se deja al lector la inclusión de los errores.

```
DEFINITION MODULE Conjuntos;

        TYPE
                CONJUNTO;
                ITEM = SHORTINT;

        PROCEDURE Crear(): CONJUNTO;
        PROCEDURE Incluir(VAR j: CONJUNTO; x: ITEM);
        PROCEDURE Vacio(j: CONJUNTO): BOOLEAN;
        PROCEDURE Excluir(VAR j: CONJUNTO; x: ITEM);
        PROCEDURE Esta(j: CONJUNTO; x: ITEM): BOOLEAN;
        PROCEDURE Cardinal(j: CONJUNTO): CARDINAL;
        PROCEDURE Elemento(j: CONJUNTO; i: CARDINAL): ITEM;
        PROCEDURE Destruir(VAR j: CONJUNTO);

END Conjuntos.

IMPLEMENTATION MODULE Conjuntos;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

        TYPE
                CONJUNTO = POINTER TO ARRAY ITEM OF BOOLEAN;

PROCEDURE Crear(): CONJUNTO;
VAR
        Retorno : CONJUNTO;
        Cont : ITEM;
BEGIN
        NEW(Retorno);
        FOR Cont := MIN(ITEM) TO MAX(ITEM) DO
                Retorno^[Cont] := FALSE;
        END;
        RETURN Retorno;
END Crear;

PROCEDURE Incluir(VAR j: CONJUNTO; x: ITEM);
BEGIN
```

```
        j^[x] := TRUE;
END Incluir;

PROCEDURE Vacio(j: CONJUNTO): BOOLEAN;
VAR
        Cont : ITEM;
BEGIN
        Cont := MIN(ITEM);
        WHILE (j^[Cont] = FALSE) AND (Cont < MAX(ITEM)) DO
                INC(Cont)
        END;
        RETURN NOT(j^[Cont]);
END Vacio;

PROCEDURE Excluir(VAR j: CONJUNTO; x: ITEM);
BEGIN
        j^[x] := FALSE;
END Excluir;

PROCEDURE Esta(j: CONJUNTO; x: ITEM): BOOLEAN;
BEGIN
        RETURN j^[x];
END Esta;

PROCEDURE Cardinal(j: CONJUNTO): CARDINAL;
VAR
        Retorno : CARDINAL;
        Cont : ITEM;
BEGIN
        Retorno := 0;
        FOR Cont := MIN(ITEM) TO MAX(ITEM) DO
                IF j^[Cont] THEN
                        INC(Retorno);
                END;
        END;
        RETURN Retorno;
END Cardinal;

PROCEDURE Elemento(j: CONJUNTO; i: CARDINAL): ITEM;
VAR
        Cont : ITEM;
        Por_buscar : CARDINAL;
BEGIN
        IF (i > 0) AND (i <= Cardinal(j)) THEN
                Por_buscar := i;
                Cont := MIN(ITEM);
                LOOP
                        IF j^[Cont] THEN
                                DEC(Por_buscar);
                        END;
                        IF Por_buscar = 0 THEN EXIT; END;
                        INC(Cont);
                END
        END;
        RETURN Cont;
END Elemento;
```

```
PROCEDURE Destruir(VAR j: CONJUNTO);
BEGIN
        DISPOSE(j);
END Destruir;

END Conjuntos.
```

3.4 EL TAD FUNCIÓN.

Hay algunas ocasiones en las que es necesario tener un conjunto de información relacionada (p.ej., un registro de un cliente), asociada a una clave por la cual se desea acceder. Esto se soluciona a través de funciones o correspondencias. En éstas, se trata de almacenar un conjunto de pares (índice-valor), donde el *índice* representa la clave mediante la cual se desea acceder a *valor*.

A pesar de que la especificación de este TAD es muy similar a la de Conjunto (con la excepción de que no se pueden repetir los índices de cada registro, en lugar de los registros en sí), vamos a ver su especificación para estudiar su comportamiento particular y sus operaciones.

Una función *f* es un concepto matemático que se describe a través de un dominio D, y un rango R, y se denota por *f*: D → R, donde D y R pueden ser dos conjuntos cualesquiera. La función *f* asocia valores de D y R. Cuando una *f* asocia un elemento r∈R a un valor d∈D, se dice que r es la imagen de d, o que d queda proyectado sobre r, y se denota por *f*(d) = r. Cuando hablamos de funciones, todo elemento de D posee a lo sumo una imagen en R. Si para una *f* determinada, todo elemento de D posee una imagen en R, decimos que *f* es una **función total**, y en caso contrario *f* será una **función parcial**, y de los valores de D que no poseen imagen, decimos que no están definidos. P.ej.:

Edad: Persona → [0..120], es total.

Cónyuge: Persona → Persona, es parcial.

Otra definición que nos conviene más es decir que una función *f* define un subconjunto de pares del producto cartesiano de D×R, con la restricción de que un elemento d∈D puede aparecer a lo sumo en un par de *f*, o sea:

$$f = \{(d, r) / (d \in D) \cap (r \in R) \cap \nexists\, (d, r')_{r' \neq r} \in f\} \subseteq D \times R$$

Así, podemos definir una función, por mera enumeración de los pares que relacionan elementos del dominio y elementos del rango. P.ej.:

Edad = {(Lenin, 40), (Mao Tse Tung, 67), (Ho Chih Min, 58)}

Edad = {(Franklin, 70), (Lincoln, 56), (Reagan, 80), (Franklin, 50)} no es una función válida porque el elemento **Franklin** tiene dos valores asociados.

Podemos decir que estas funciones son parciales, ya que hay elementos que no tienen valor asociado. P.ej., ¿qué es Edad(Castro)?: no está definido.

De esta forma, la especificación de una función quedará como un mero conjunto de pares, en la vertiente **función total**, y como un conjunto de pares en el que se asocia un valor por defecto a todo elemento del dominio, en el caso de la **función total**.

Recordemos asimismo, que una función *f* puede clasificarse también en:

- **Inyectiva**: Cuando elementos diferentes d∈D, tienen imágenes diferentes r∈R.
- **Suprayectiva**: Cuando todo elemento r∈R es imagen de al menos un d∈D.
- **Biyectiva**: Es aquella función inyectiva y suprayectiva.

3.4.1 El TAD Función parcial.

Aquí, disponemos de dos generadores, a saber:

Crear: $\longrightarrow$ Función_p

Insertar: $\text{Función}_p \times \text{Dominio} \times \text{Rango} \longrightarrow \text{Función}_p$

donde Crear() devuelve una tabla vacía, e insertar devuelve la tabla que se pasa como entrada, añadiendo el par (DOMINIO, RANGO). La función Valor será parcial, ya que no está definida para aquellos elementos del Dominio que no hayan sido explícitamente emparejados con otro del Rango. Así, la especificación completa quedaría:

tipo Función_p
dominios Dominio, Rango, Lógico
generadores
Crear: $\longrightarrow$ Función_p
Insertar: $\text{Función}_p \times \text{Dominio} \times \text{Rango} \longrightarrow \text{Función}_p$
constructores
Eliminar: $\text{Función}_p \times \text{Dominio} \longrightarrow \text{Función}_p$
selectores
Valor: $\text{Función}_p \times \text{Dominio} \not\longrightarrow \text{Rango}$
Está_Definido: $\text{Función}_p \times \text{Dominio} \longrightarrow$ Lógico
precondiciones t:Función_p d:Dominio
pre Valor(t, d) : Está_Definido(t, d)
ecuaciones t:Función_p d, d_1, d_2:Dominio r_1, r_2:Rango

1) Insertar(Insertar(t, d_1, r_1), d_2, r_2) == SI $d_1 = d_2$ ENTONCES
Insertar(t, d_2, r_2)
SI NO
Insertar(Insertar(t, d_2, r_2), d_1, r_1)

2) Eliminar(Crear, d) == Crear

3) Eliminar(Insertar(t, d_1, r_1), d) == SI $d = d_1$ ENTONCES
Eliminar(t, d)
SI NO
Insertar(Eliminar(t, d), d_1, r_1)

4) Valor(Insertar(t, d_1, r_1), d) == SI $d = d_1$ ENTONCES
r_1
SI NO
Valor(t, d)

5) Está_Definido(Crear, d) == F

6) Está_Definido(Insertar(t, d_1, r_1), d) == SI $d = d_1$ ENTONCES
V
SI NO
Está_Definido(t, d)

fin

No se ha incluído el predicado Está_Vacía() por ser impropio del tipo que estamos

definiendo.

Nótese que las ecuaciones que relacionan entre sí a las operaciones Insertar, hacen que el conjunto de generadores sea no libre. O sea, permitiremos que en los términos formados sólo por generadores haya pares cuyo valor del dominio sea idéntico, aunque en un término canónico se habrán hecho las reducciones y permutaciones necesarias (merced a la primera ecuación), para evitar esta situación. El significado de esta primera ecuación es que, si en más de un par, coincide el valor del dominio, sólo tiene sentido la última vinculación efectuada. Como no podemos asegurar que la función de entrada al resto de las operaciones sea un término en forma canónica, se deberá andar con cuidado. P.ej., en la operación Eliminar no basta con eliminar el primer par en el que el valor del dominio coincida con el del elemento que se desea borrar; es necesario seguir aplicando la operación Eliminar para acabar con todos los posibles pares; definir la ecuación 3) como:

3') Eliminar(Insertar(t, d_1, r_1), d) == SI d = d_1 ENTONCES
t
SI NO
Insertar(Eliminar(t, d), d_1, r_1)

sería incorrecto. El siguiente caso hará una idea de la situación.

Sea el término

t = Insertar(Insertar(Insertar(Crear, 'a', 65), 'b', 70), 'c', 80)

donde el Dominio es un alfabeto de caracteres, y el rango es $\mathbb{N}$. Nadie nos impide hacer un Insertar(t, 'b', 4), lo que daría lugar a:

t' = Insertar(Insertar(Insertar(Insertar(Crear, 'a', 65), 'b', 70), 'c', 80), 'b', 4)

E igualmente podemos hacer algo así como Eliminar(t, 'b'), quedando:

t'' = Eliminar(Insertar(Insertar(Insertar(Insertar(Crear, 'a', 65), 'b', 70), 'c', 80), 'b', 4), 'b')

Si en lugar de aplicar la ecuación 1) a t', aplicamos 3') a t'', obtendremos nuevamente t, lo cual no es correcto, ya que en t, el valor 'b' sigue definido, en contraposición del objetivo de la operación Eliminar, que es desvincular a 'b' de todo elemento del rango. Por contra, si antes de aplicar 3'), aplicamos 1) dos veces, tendremos:

Eliminar(Insertar(**Insertar(Insertar(Insertar(Crear, 'a', 65), 'b', 70), 'c', 80)**, 'b', 4), 'b') =
Eliminar(**Insertar(Insertar(Insertar(Insertar(Crear, 'a', 65), 'c', 80), 'b', 70), 'b', 4)**, 'b') =
Eliminar(Insertar(Insertar(Insertar(Crear, 'a', 65), 'c', 80), 'b', 4), 'b') =

Y aplicando ahora 3') llegaríamos a:

Insertar(Insertar(Crear, 'a', 65), 'c', 80)

que es diferente a t, y en la que *no está definda la función para el valor 'b'*, por lo que observamos que, dependiendo del orden en que se aplique la secuencia de ecuaciones, podemos llegar a un resultado u otro.

Si en lugar de aplicar 3'), aplicamos 3), sea cual sea el orden de aplicación de las ecuaciones, llegaremos al mismo término, como puede comprobar el lector. Este hecho debe tenerse en cuenta en cualquier otra operación que se quiera añadir al TAD.

3.4.2 El TAD Función total.

Este caso difiere del anterior en que aquellos valores del dominio a los que no se ha asociado explíctamente un valor del rango, se les asigna uno por defecto. El valor que se asigna

por defecto, puede venir dado por la propia estructura del TAD, o puede ser especificado por el usuario en el momento de la creación. Veamos la implementación de este último caso, en el que es necesario incluir un parámetro a la operación Crear.

tipo Función_T
dominios Dominio, Rango, Lógico
generadores
Crear: Rango $\longrightarrow \text{Función}_T$
Insertar: $\text{Función}_P \times$ Dominio $\times$ Rango $\longrightarrow \text{Función}_T$
constructores
Eliminar: $\text{Función}_T \times$ Dominio $\longrightarrow \text{Función}_T$
selectores
Valor: $\text{Función}_T \times$ Dominio $\longrightarrow$ Rango
ecuaciones t:Función_T d, d_1, d_2:Dominio r, r_1, r_2:Rango

Insertar(Insertar(t, d_1, r_1), d_2, r_2) == SI $d_1 = d_2$ ENTONCES
Insertar(t, d_2, r_2)
SI NO
Insertar(Insertar(t, d_2, r_2), d_1, r_1)

Eliminar(Crear(r), d) == Crear

Eliminar(Insertar(t, d_1, r_1), d) == SI $d = d_1$ ENTONCES
Eliminar(t, d)
SI NO
Insertar(Eliminar(t, d), d_1, r_1)

Valor(Crear(r), d) == r

Valor(Insertar(t, d_1, r_1), d) == SI $d = d_1$ ENTONCES
r_1
SI NO
Valor(t, d)

fin

Aquí, el objetivo de la operación Eliminar es desvincular el valor del dominio de la vinculación explícita hecha por el usuario del TAD, aunque en cualquier caso, el objeto desvinculado, seguirá estando asociado al valor por defecto.

Véase también que, en este caso, no es necesaria la operación Está_Definido, ya que todo elemento del dominio está definido por tratarse de funciones totales.

3.4.3 IMPLEMENTACIONES.

Pueden ser mediante:

- Listas lineales: Se ponen los pares (índice-valor) uno detrás de otro, como si de una lista desordenada se tratase.

El tipo sería:

```
TYPE
    PAR = RECORD
```

```
            indice : DOMINIO;
            valor : RANGO;
    END;
```

La operación Elemento buscará el par cuyo índice sea x, y devolverá el campo valor asociado.

- Ordenada: Es igual a la anterior, con la salvedad de que los pares están ordenados por el índice. A su vez, la búsqueda puede ser:
  - Dicotómica: En el caso de que se pueda acceder a cualquier posición de la lista, en el mismo tiempo (p.ej., cuando se usan estructuras estáticas de zona compacta), resulta conveniente efectuar las búsquedas de la clave mediante un proceso dicotómico.
  - Interpolación: Es igual a la anterior, pero en lugar de buscar en el punto intermedio, se busca en el punto interpolado en el que debiera estar el elemento buscado.
  Si I y F son las posiciones inicial y final, y Cl es el valor clave buscado, la posición en que mirar vendría dada por:

$$(F - I)\cdot\left(\frac{Cl - a[I]}{a[F] - a[I]}\right) + I$$

- Una tabla que tenga una entrada por cada índice, y que contenga los valores asociados.

- Si el Dominio es muy grande, la aproximación anterior puede ser prohibitiva. Para solucionarlo, en el caso de que sólo unos pocos elementos del dominio estén definidos, podemos crear una tabla de longitud suficiente, y almacenar en sus celdas los pares deseados, de forma parecida a una lista desordenada, pero en una tabla, en lugar de una lista. La ventaja de este método es que pueden emplearse ciertos criterios (además del puramente secuencial), para decidir en qué posición se almacena un par determinado, y es más, estos criterios pueden aumentar espectacularmente la eficiencia a la hora de encontrar dónde se encuentra un par, o dónde se debería encontrar.

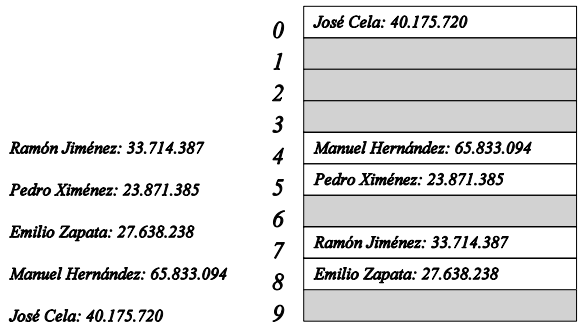
Dado que aquí interesa enormemente la rapidez en el acceso a un par de nuestra lista, vamos a usar el último método, usando como criterio las funciones de dispersión (Hash).

3.4.4 FUNCIÓN DE DISPERSIÓN (HASH).

Es una función cuyo dominio son los índices, y cuyo rango son los números naturales o posiciones de que se dispone en una tabla.

Se usarán cuando la cantidad potencial de índices sea enorme, pero cuando se vayan a guardar pocos valores (p.ej. los DNI de nuestros clientes).

Cualquier función F:D → R se puede transformar fácilmente en H:D → P, donde P es un conjunto de posiciones. Así, para un elemento d∈D, H(d) será la posición en que se almacena F(d), (d es la clave, y F(d) es el valor asociado). P.ej. si tenemos D = {nº's de D.N.I} y R = {Nombre y Apellidos}, y sólo vamos a guardar los datos de unos pocos empleados de una empresa, podemos hacer una función hash como p.ej.: H(nº D.N.I.) = último dígito del D.N.I.. En la figura se observan las posiciones en las que se guardarían los datos de diferentes empleados. En cada celda de la tabla habrá un valor enumerado que nos dirá si la celda está ocupada o no.



Cuando el dominio es un conjunto muy grande, la implementación mediante arrays simples no es satisfactoria. La técnica hashing asigna una función pseudoaleatoria H, de manera que si F(d)=r, entonces H(d)=p, y (d, r) se almacena en la posición *p* del array de almacenamiento.
Las funciones hash deben tener las siguientes características:

- Distribución uniforme.
- Pequeños cambios en la clave han de resultar en cambios aleatorios en la función de dispersión.
- Toda posición de la tabla debe ser destino de una clave.
- Los algoritmos han de ser sencillos.

Una condición fundamental para conseguir esto es que la función hash debe depender de todos sus argumentos.

Antes de pasar a ver algunas funciones hash, veamos que son las colisiones. Resulta claro que como el número de posiciones en la tabla puede ser menor que el número de elementos a almacenar, llegará un momento en que se llenará, y cuando intentemos introducir algún elementos más, se producirá una colisión. Pero no sólo habrá colisiones en tales casos, sino que como las funciones hash son funciones muchos a uno (el universo de claves tiene más elementos que posiciones hay en la tabla), pues es posible que dos claves den lugar a la misma posición en la tabla, incluso cuando ésta esté poco llena. En el caso anterior se producirá una colisión en el momento en que se intente guardar los datos de dos empleados cuyo nº del D.N.I. coincida en su último dígito.

P.ej. supóngase que vamos a almacenar información relacionada con un individuo, y que la clave será su nombre; supondremos además que toda persona tiene un nombre único y diferente del de los demás. Supongamos que vamos a trabajar con no más de 28 clientes. Habrá que tener una H:Nombre → [1..28]; aprovechando que sólo hay 28 letras en el alfabeto, podemos hacer H(Nombre) = ((Código ASCII de la última letra del nombre) MOD 28) + 1

Sin embargo, en el momento que queramos guardar datos de Mario y Antonio, se

producirá una colisión, aunque no haya nada más en la tabla.

Así, una función de dispersión es una función **pseudoaleatoria** que debe ser fácil y rápida de calcular. El problema radica en que H no puede ser inyectiva y se producen colisiones. Es necesario un mecanismo adecuado de manejo de colisiones.

Para que el hashing sea una solución práctica, es preciso resolver dos aspectos fundamentales:

- Tratamiento de colisiones (hashing abierto y cerrado).
- Buena elección de H.

3.4.5 HASHING PERFECTO.

Cuando la tabla es estática, y las claves se conocen a priori, hay técnicas que permiten encontrar funciones de dispersión perfectas, en las que no existen colisiones, y en las que se puede encontrar cualquier elemento de forma inmediata, o sea, con complejidad O(1) siempre.

3.4.6 ALGUNAS FUNCIONES HASH.

Podemos dividirlas en varios bloques, suponiendo siempre que la clave es numérica; si no lo es, entonces se transforma a numérica:

- **Extracción**: Consiste en tomar ciertos bits de una transformación de la clave (también se llama *fold & unfold*): El ejemplo más típico es tomar los *k* bits centrales del resultado de elevar la clave al cuadrado. En tal caso, el número de posiciones en la tabla debe ser 2^k.
- **Compresión**: Consiste en dividir la clave en trozos, y operar con ellos, p.ej., haciendo XOR con tales trozos:

$$Ej.: DAVID \rightarrow D \oplus A \oplus V \oplus I \oplus D$$

Sin embargo, permutaciones de las palabras pueden dar lugar a resultados iguales. Así, podemos tomar en lugar de caracteres completos, trozos de 9 bits de las representaciones completas.

- **División**: Estas funciones son de la forma H(k) = k MOD r. Es fundamental escoger la r adecuada. En general es suficiente con escoger un valor de r tal que no posea divisores menores que 20, siendo lo ideal que r sea primo.

3.4.7 COLISIONES.

Cuando se producen colisiones, hay varias soluciones:

- Hashing abierto. Aquí, a cada entrada de la tabla hay asociada una lista que contiene los elementos colisionados.
- Hashing cerrado. En este caso se tienen en reserva varias funciones Hash secundarias que nos permiten buscar posiciones alternativas.

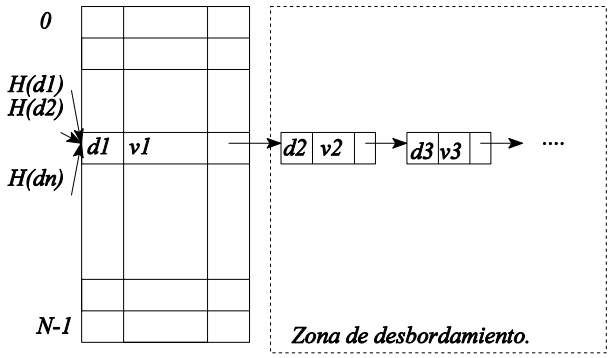
Veamos estos dos métodos.

3.4.7.1 Hashing abierto.

Supongamos que hay una colisión entre los elementos d_1, d_2, ...d_n de D, o sea:

$$H(d_1) = H(d_2) = ... = H(d_n)$$

Mediante el hashing abierto, esto se resuelve mediante una lista de pares en la posición $H(d_i)$, de manera que la tabla contiene un par, y un puntero a la lista de desbordamiento, también llamada área de desbordamiento. También, para mayor uniformidad, podemos hacer que la tabla contenga sólo el puntero al elemento en cuestión.



3.4.7.2 Hashing cerrado.

En este caso, tras la aplicación de la función Hash, si se ha producido una colisión, aplicamos una nueva función Hash para obtener una alternativa; si en ésta se produce otra colisión, aplicamos una tercera función hash, y así sucesivamente. Por tanto, tendremos como mucho tantas funciones de dispersión como casillas haya en nuestra tabla. Así, los pares a almacenar se guardan siempre en la tabla.

Podemos definir un cluster (piña) inicial como los elementos que van a parar a la misma posición. Cuando se producen varias entradas de tales índices, éstas se extienden a través de una o varias funciones rehash. Definimos un cluster general como todos los elementos entre los que se producen colisiones en una tabla hash con algunos elementos ya introducidos. También podemos definirlo como las posiciones por las que se extienden estos elementos.

Un dato muy importante a tener en cuenta es como borrar los elementos. Cuando se elimina un elemento de una posición de la tabla, es necesario marcar la casilla como *liberada*, en lugar de como *desocupada*, ya que si no, una búsqueda de un elemento existente en la tabla, pero situado en una posición posterior, debido a una colisión con el elemento que se borra, no será positiva.

Así, podemos tener las siguientes estrategias rehash, en las que consideraremos la tabla como una estructura circular, o sea, la celda siguiente al último elemento es la primera:

- Rehash lineal: $H_i(Cl) = H_{i-1}(Cl) + k$, donde k es una constante.

  Esta estrategia tiene el problema de que no depende de *i*, ni tampoco de la clave Cl, en el sentido de que si $H_0(Cl_1) = H_0(Cl_2)$, entonces las secuencias de posiciones de rehash para Cl_1 y para Cl_2 son la misma. El valor k debe ser tal que se comprueben todas las posiciones de la tabla. Se suele tomar k = 1.

- Rehash aleatorio: $H_i(Cl) = H_{i-1}(Cl) + k_{Cl}(i)$.

  Aquí, en lugar de usar k para todos los H_i, empleamos una $k_{Cl}(i)$ diferente para cada H_i;

así los k_i forman una secuencia de n-1 números aleatorios, diferente para cada Cl, y la secuencia k_i debe ser una permutación aleatoria de los naturales 1..n-1, que asegure que se recorren todas las posiciones de la tabla

- Hash con doble función: $H_i = H_{i-1} + f(i, Cl)$.
Esta estrategia tiene el problema de que se tarda mucho en recalcular las funciones hash.

Por último, decir que cuando el factor de carga de una tabla hash alcanza el 90%, la tabla debe ser redimensionada al doble del número de posiciones que poseía inicialmente. El *factor de carga* se define como M/N, donde M es el número de elementos guardados en la tabla, y N el número de posiciones que tiene la tabla.

3.4.8 IMPLEMENTACIONES.

A continuación vamos a ver la implementación del TAD Función parcial mediante una tabla de dispersión cerrada (sin área de desbordamiento), y con función hash por división entre 23 (nótese que 23 es primo), y funciones de redispersión lineales con constante igual a 1. Asimismo, se hace el tratamiento de errores con variable global.

```
DEFINITION MODULE FuncionParcial;
TYPE
        FUNCIONP;
        RANGO = ARRAY [0..20] OF CHAR;
        DOMINIO = CARDINAL;
        ERRORFUNCION = (SinError, NoEncontrado, SinMemoria, FuncionNoCreada, OtroError);
        (* El error OtroError se añade para posibles ampliaciones. *)
VAR
        error: ERRORFUNCION;

PROCEDURE Crear(): FUNCIONP;
PROCEDURE Vacia(f : FUNCIONP): BOOLEAN;
PROCEDURE Insertar(VAR f : FUNCIONP; r:RANGO; d: DOMINIO);
PROCEDURE Suprimir(VAR f : FUNCIONP; d: DOMINIO);
PROCEDURE Elemento(f : FUNCIONP; d: DOMINIO): RANGO;
PROCEDURE Esta(f : FUNCIONP; d: DOMINIO): BOOLEAN;
PROCEDURE Destruir(VAR f : FUNCIONP);

END FuncionParcial.


IMPLEMENTATION MODULE FuncionParcial;
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

CONST MAX = 23;
TYPE
        POSICION = [0..MAX-1];
        MARCA = (VACIO, SUPRIMIDO, OCUPADO);
        CELDA = RECORD
                clave: DOMINIO;
                cont: RANGO;
                marca: MARCA
```

```
        END;
        ESTRUCTURA = ARRAY POSICION OF CELDA;
        FUNCIONP = POINTER TO ESTRUCTURA;

PROCEDURE Hash(d: DOMINIO): POSICION;
BEGIN
        RETURN d MOD MAX;
END Hash;

(*
Esta es una función de dispersión por extracción, que tomaría los bits 7, 8, 9, 10 y 11 del cuadrado de la
clave, dando un resultado entre 0 y 31. Se podría utilizar si la tabla tuviese 32 elementos indexados del
0 al 31. En nuestro caso habría que hace finalmente un MOD 28. En cualquier caso, hay que tener cuidado
de que el producto de d*d no supere el MAX(DOMINIO) en este caso.
PROCEDURE Hash(d: DOMINIO): POSICION;
BEGIN
        RETURN (((d*d) MOD 2^11) DIV 2^7);
END Hash;
*)

PROCEDURE Crear(): FUNCIONP;
VAR
        f: FUNCIONP; i: POSICION;
BEGIN
        error := SinError;
        IF NOT (Available(SIZE(ESTRUCTURA)) THEN error := SinMemoria; RETURN NIL; END;
        NEW(f);
        FOR i := 0 TO MAX-1 DO
                f^[i].marca := VACIO
        END;
        RETURN f;
END Crear;

PROCEDURE Vacia(f : FUNCIONP): BOOLEAN;
VAR
        i: POSICION;
BEGIN
        IF f = NIL THEN error := FuncionNoCreada; RETURN FALSE; END;
        error := SinError;
        FOR i := 0 TO MAX-1 DO
                IF f^[i].marca = OCUPADO THEN
                        RETURN FALSE
                END
        END;
        RETURN TRUE
END Vacia;

PROCEDURE Insertar(VAR f : FUNCIONP; r:RANGO; d: DOMINIO);
VAR
        h,h1: POSICION;
        encontrado : BOOLEAN;
BEGIN
        IF f = NIL THEN error := FuncionNoCreada; RETURN; END;
        error := SinError;
        encontrado := FALSE;
        h := Hash(d);
```

```
        IF (Esta(f, d)) THEN
                h1 := h;
                WHILE (f^[h1].clave <> d) DO
                        h1 := (h1 + 1) MOD MAX;
                END;
                encontrado := TRUE;
        ELSE
                h1 := h;
                LOOP
                        IF (f^[h1].marca <> OCUPADO) THEN encontrado := TRUE; EXIT; END;
                        h1 := (h1 + 1) MOD MAX;
                        IF h1 = h THEN error := SinMemoria; EXIT; END;
                END;
        END;
        IF encontrado THEN
                WITH f^[h1] DO
                        clave := d;
                        cont := r;
                        marca := OCUPADO;
                END;
        END;
END Insertar;

PROCEDURE Suprimir(VAR f : FUNCIONP; d: DOMINIO);
VAR
        h,h1: POSICION;
BEGIN
        IF f = NIL THEN error := FuncionNoCreada; RETURN; END;
        error := SinError;
        h := Hash(d);
        IF (f^[h].marca=OCUPADO) AND (f^[h].clave=d) THEN
                f^[h].marca := SUPRIMIDO;
        ELSIF f^[h].marca = VACIO THEN
                RETURN;
        ELSE
                h1 := (h + 1) MOD MAX;
                WHILE (f^[h1].marca<>VACIO) AND (f^[h1].clave<>d) AND (h1<>h) DO
                        h1 := (h1 + 1) MOD MAX
                END;
                IF (f^[h1].marca=OCUPADO) AND (f^[h1].clave=d) THEN
                        f^[h1].marca := SUPRIMIDO
                END
        END
END Suprimir;

PROCEDURE Elemento(f : FUNCIONP; d: DOMINIO): RANGO;
VAR
        h,h1: POSICION;
        basura : RANGO;
BEGIN
        IF f = NIL THEN error := FuncionNoCreada; RETURN basura; END;
        error := SinError;
        h := Hash(d);
        IF (f^[h].marca=OCUPADO) AND (f^[h].clave=d) THEN
                RETURN f^[h].cont;
        ELSE
```

```
                h1 := (h + 1) MOD MAX;
                WHILE (f^[h1].marca<>VACIO) AND (f^[h1].clave<>d) AND (h1<>h) DO
                        h1 := (h1 + 1) MOD MAX
                END;
                IF (f^[h1].marca <> OCUPADO) OR (f^[h1].clave <> d) THEN
                        error := noEncontrado;
                        RETURN basura;
                ELSE
                        RETURN f^[h1].cont;
                END;
        END
END Elemento;

PROCEDURE Esta(f : FUNCIONP; d: DOMINIO): BOOLEAN;
VAR
        h,h1: POSICION;
BEGIN
        IF f = NIL THEN error := FuncionNoCreada; RETURN FALSE; END;
        error := SinError;
        h := Hash(d);
        IF (f^[h].marca=OCUPADO) AND (f^[h].clave=d) THEN
                RETURN TRUE;
        ELSE
                h1 := (h + 1) MOD MAX;
                WHILE (f^[h1].marca<>VACIO) AND (f^[h1].clave<>d) AND (h1<>h) DO
                        h1 := (h1 + 1) MOD MAX;
                END;
                IF (f^[h].marca=OCUPADO) AND (f^[h].clave=d) THEN
                        RETURN TRUE
                ELSE
                        RETURN FALSE
                END
        END
END Esta;

PROCEDURE Destruir(VAR f: FUNCIONP);
BEGIN
        IF f = NIL THEN error := FuncionNoCreada; RETURN; END;
        error := SinError;
        DISPOSE(f)
END Destruir;

BEGIN
        error := SinError;
END FuncionParcial.
```

Ejercicios.

1.- Se quiere organizar una tabla de dispersión de siete posiciones con estrategia de direccionamiento cerrado, y redispersión lineal. Sea la función *H* de dispersión que da los siguientes valores para las siguientes claves:

H(Sonia) = 3
H(Gema) = 5
H(Paula) = 2
H(Ana) = 3
H(Rogelia) = 3
H(Cristina) = 2

a) Insertar todas estas claves en el siguiente orden: Paula, Ana, Cristina, Rogelia, Sonia, y Gema. Mostrar claramente como va evolucionando la tabla e indicar las colisiones que se producen. A continuación mostrar el camino generado para buscar la clave de Rogelia.

b) Repetir el proceso cuando el orden de inserción sea: Sonia, Ana, Rogelia, Gema, Paula, y Cristina. Comprobar que las claves ocupan, en general, posiciones diferentes a las de antes dentro de la tabla. A continuación, mostrar el camino generado para buscar la clave Rogelia

c) ¿En cuál de las dos configuraciones obtenidas hay menos comparaciones entre claves en el peor caso de búsqueda de un elemento en la tabla?.

d) ¿Hay algún orden de inserción que distribuya mejor las claves según el criterio del apartado anterior?¿Y peor?¿Por qué?.

2.- Queremos almacenar en una tabla de dispersión 10.000 elementos de X bits de tamaño, de modo que no exista ningún valor especial. Suponiendo una función de dispersión que distribuya los elementos de manera uniforme, y sabiendo que un booleano ocupa un bit, y un entero, o un puntero ocupa 32 bits, determinar el espacio que se ocuparía para almacenar la tabla en en el caso más favorable, cuando se utiliza:

- Hashing abierto.
- Hashing abierto, pero en la tabla no se guardan registros, sino punteros a registros.
- Hashing cerrado.

si queremos que el número de comparaciones entre claves en una búsqueda con éxito, sea como media de 2' 5.

3.- Hacer en Modula-2 la implementación funcional de la operación Unión entre bolsas, suponiendo que se hallan representadas por listas ordenadas:

```
PROCEDURE Union(b1, b2 : BOLSA) : BOLSA;
```

4.- El ejercicio anterior se puede hacer de dos formas diferentes:

a) Suponiendo que podemos acceder a la estructura interna de la lista.

b) Haciendo uso exclusivamente de las operaciones que nos suministra el TAD Lista.

Suponiendo que las listas se hallan implementadas con estructuras dinámicas simplemente encadenadas sin cabecera, ¿qué ventajas y desventajas tiene cada una de las formas de operar anteriores?.

5.- ¿Qué hace el siguiente trozo de código?:

```
Crear(l3); (* Inicializa la lista l3 *)
p := 1;
WHILE p <= Longitud(l1) DO
        q := 1;
        WHILE q <= Longitud(l2) DO
                IF Elemento(l1, p) = Elemento(l2, q) THEN
                        Insertar(l3, 1, Elemento(l1, p));
                END;
                INC(q);
        END;
        INC(p);
END;
```

6.- La operación Mezcla : Conjunto × Conjunto $\not\longrightarrow$ Conjunto, es similar a la operación Unión, excepto en que se tiene la seguridad (mediante precondiciones), de que los argumentos son conjuntos disjuntos, o sea, no tienen elementos en común. Escribir en Modula-2 la operación Mezcla de dos formas distintas, una suponiendo que los conjuntos se representan por listas desordenadas, y otra si se representan por listas ordenadas. En cualquier caso, suponer que se tiene acceso a la estructura de la lista.

7.- Partiendo de las ecuaciones dadas en clase, y añadiendo:
Unión(c, Crear) == c
Unión(c_1, Agregar(i, c_2)) == Agregar(i, Unión(c_1, c_2)),
demostrar que:
Unión(c_1, c_2) == Unión(c_2, c_1).

8.- En el TAD Función , para evitar que aparezcan pares con el mismo valor de índice, es necesario incluir ecuaciones que pueden impedir el cálculo automático de reducciones. Para evitar esto en la medida de los posible, podemos hacer que Insertar (que es un generador), no esté disponible para el usuario, y en su lugar incluir una función sustituta Incluir(c, v, f), cuyo objetivo sea:
1) Eliminar los pares (c, _), donde _ representa cualquier cosa, de la función f, y
2) Por último hacer un Insertar en el resultado.
Reescribir las ecuaciones de la especificación anterior teniendo esto en cuenta.

9.- Vamos a definir un TAD Conjunto generalizado, de manera que se permitan conjuntos de conjuntos, a cualquier nivel. Obligaremos que todo elemento del tipo base esté metido en un conjunto del que será el único miembro; por tanto, no trabajaremos con elementos sueltos, sino con conjuntos unitarios. Así, nuestros generadores serán:

generadores

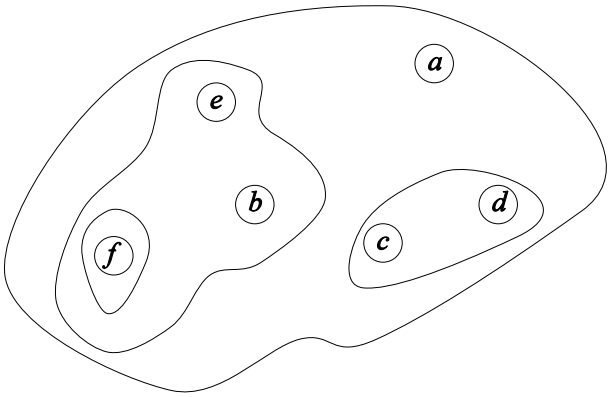
Crear: $\longrightarrow$ Conjunto
Unitario: Elemento $\longrightarrow$ Conjunto
Agregar: Conjunto × Conjunto $\longrightarrow$ Conjunto

precondiciones

pre Agregar(c_1, c_2) : not Es_Unitario(c_2)

de manera que Agregar(c_1, c_2), hace que el conjunto c_1 sea un elemento más del conjunto c_2.

P.ej., el conjunto de la figura se representaría por:
c_1 = Agregar(Unitario(b), Agregar(Unitario(e), Agregar(Agregar(Unitario(f), Crear), Crear)))
c_2 = Unitario(a)
c_3 = Agregar(Unitario(d), Agregar(Unitario(c), Crear))
quedando el conjunto final como:
Agregar(c_1, Agregar(c_2, Agregar(c_3, Crear)))



Con estos generadores, dar la especificación algebraica de las siguientes operaciones:
=: Conjunto × Conjunto ⟶ Lógico
⊆: Conjunto × Conjunto ⟶ Lógico
∈: Conjunto × Conjunto ⟶ Lógico
en el sentido propio que ya sabemos.

Especificar además la operación:
Elim_En_Profundidad: Conjunto × Conjunto ⟶ Conjunto
cuyo objetivo es eliminar el conjunto primero en el segundo, tanto si aparece en él como miembro, como si aparece en alguno de sus miembros a cualquier profundidad.

10.- Reescribir la especificación del TAD Función Total, reutilizando el TAD Lista.

11.- Se pretende especificar un sistema de cambio de moneda en una máquina de refrescos. La máquina de refrescos posee un número concreto de monedas de 1, 5, 10, 25, 50, 100 y 500 pesetas. Establecer el conjunto de operaciones generadoras que permitan representar cuántas monedas de cada tipo posee una máquina.

12.- Siguiendo con el ejercicio anterior, se desea que dada una máquina (con unas monedas para cambio dadas), y un importe, la máquina devuelva una lista con las monedas fraccionarias cuya suma dan lugar al importe dado. P.ej., si se da un importe de 133 pesetas, la máquina dirá que se puede convertir en 1 moneda de 100 pesetas, 1 moneda de 25 pesetas, 1 moneda de 5 pesetas, y 3 monedas de 3 pesetas. no obstante, el resultado devuelto será éste si la máquina dispone de las monedas suficientes de cada tipo. Por ejemplo, si en el caso anterior, la máquina no posee monedas de 25 pesetas, el resultado sería 1 moneda de 100 pesetas, 3 monedas de 10 pesetas, y 3 monedas de 3 pesetas. Se procurará dar siempre el menor número posible de monedas. Si la máquina no posee monedas suficientes se producirá un error.