Práctica 7. TAD aplicación

Objetivos.

Se trata de construir el TAD APLICACION con una implementación acotada.

Se introducen la *dispersión* (hashing) y las *cachés* como técnicas de implementación para mejorar la eficiencia.

Enunciado.

Construir el TAD APLICACION según el siguiente módulo de definición:

```
DEFINITION MODULE Aplicacion;

    FROM Dominio IMPORT DOMINIO;
    FROM Rango IMPORT RANGO;

    TYPE   APLICACION;

    PROCEDURE Crear(): APLICACION;
    PROCEDURE Destruir(VAR a: APLICACION);

    PROCEDURE EsVacia(a: APLICACION): BOOLEAN;
    PROCEDURE Enlazado(a: APLICACION; d: DOMINIO): BOOLEAN;
    PROCEDURE Extension(a: APLICACION): CARDINAL;
    (* Pre: Enlazado(a,d) *)
    PROCEDURE RangoDe(a: APLICACION; d: DOMINIO): RANGO;

    (* Pre: NOT Enlazado(a,d) *)
    PROCEDURE Enlazar(VAR a: APLICACION; d: DOMINIO; r: RANGO);
    (* Pre: Enlazado(a,d) *)
    PROCEDURE Desenlazar(VAR a: APLICACION; d: DOMINIO);

    PROCEDURE Copiar(VAR a1: APLICACION; a2: APLICACION);

    TYPE   TIPO_OPERACION=PROCEDURE(DOMINIO, RANGO);
    PROCEDURE Aplicar(a: APLICACION; op: TIPO_OPERACION);

    TYPE   TIPO_ERROR = <<a definir>>;

    PROCEDURE Error():  TIPO_ERROR;

END Aplicacion.
```

El TAD Aplicación.

Una APLICACION (o función) es una correspondencia entre dos conjuntos, DOMINIO y RANGO, tal que los elementos de DOMINIO tienen a lo más una imagen en RANGO. Otra forma de definir una APLICACION es como un conjunto de pares ordenados *(d,r)*, tal que *d* es un elemento de DOMINIO, *r* es un elemento de RANGO, y no existen dos pares que coincidan en la primera componente.

Puesto que el TAD APLICACION no es más que un conjunto, parece razonable representar sus valores mediante una secuencia de nodos enlazados mediante punteros (implementación no acotada) o bien como un array de nodos (implementación acotada). Tal fue la decisión que tomamos al representar los TADs CONJUNTO y BOLSA. Este tipo

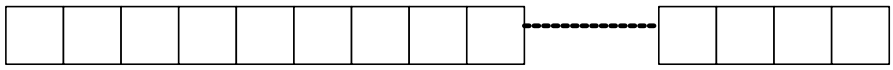
de representación *secuencial* es adecuada cuando se implementan TADs lineales (pilas, colas, listas), pues la abstracción no es más que una secuencia de elementos a la que se accede de acuerdo con cierta disciplina. Sin embargo, en TADs no lineales (conjuntos, bolsas, aplicaciones,...) esta representación puede no resultar ventajosa. Ciertas operaciones requieren recorrer la secuencia para comprobar si un valor dado se encuentra almacenado en la misma, lo que lleva un tiempo proporcional a la longitud de la secuencia (es decir, O(*n*)).

Mejora de la eficiencia mediante dispersión.

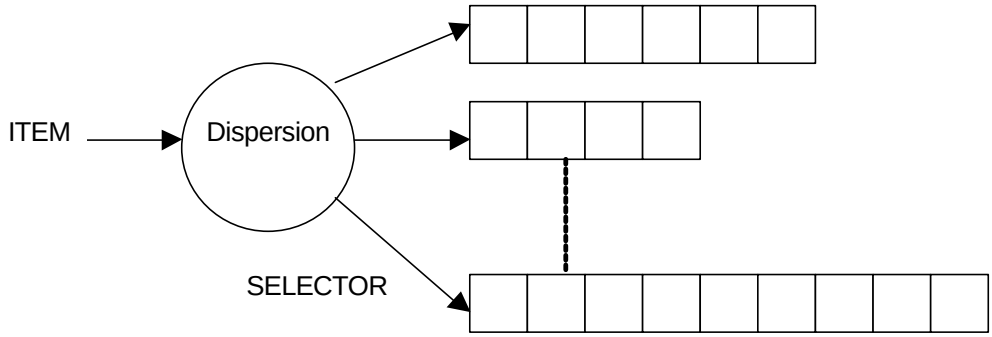
Una forma de reducir el coste de estas operaciones es aplicar una técnica de dispersión (*hashing*) para reemplazar la secuencia original por varias secuencias de menor longitud, a las que nos referiremos con el nombre de *particiones*. La idea consiste en definir una función de dispersión que calcule para cada valor del tipo base de la secuencia ITEM otro valor de un tipo SELECTOR:

```
PROCEDURE Dispersion(x: ITEM): SELECTOR;
```

El valor Dispersion(x) nos permitirá seleccionar la secuencia (i.e. partición) adecuada para almacenar el elemento x. Gráficamente, la dispersión consiste en sustituir la secuencia única:



por el conjunto de secuencias:



Para que la dispersión mejore la eficiencia, deben satisfacerse las siguientes condiciones:

- la función Dispersion debe ser simple de calcular y, a ser posible, debe distribuir uniformemente los valores del tipo ITEM sobre los valores del tipo SELECTOR.
- un valor del tipo SELECTOR debe permitirnos seleccionar la partición adecuada en tiempo constante O(1). Típicamente, el tipo SELECTOR será el índice de un array, aunque nada impide que se trate de un puntero, o un número de bloque en un dispositivo de almacenamiento secundario.

Aplicando dispersión, el coste de acceso a un elemento pasa de ser O(*n*) a ser O(Dispersion) + O(1) + O(*n/s*), donde *s* es el número de valores posibles del tipo SELECTOR; es decir, el número de particiones en que queda distribuida la secuencia original.

Cabe destacar que este esquema puede aplicarse para dispersar cualquier estructura de datos. Podríamos, por ejemplo, aplicar dispersión para sustituir un árbol por un conjunto de árboles.

Colisiones, Dispersión Abierta y Dispersión Cerrada.

Se dice que se produce una *colisión* entre los valores x1 y x2 cuando

Dispersion(x1) = Dispersion(x2)

es decir, cuando x1 y x2 deben almacenarse en la misma partición de la estructura dispersa. La función de dispersión debe diseñarse de manera que se minimice el número de colisiones.

Se dice que se tiene una dispersión *abierta* cuando las particiones de la estructura dispersa son estructuras no acotadas. En tal caso, las colisiones se resuelven añadiendo un nuevo nodo a la partición seleccionada por la función Dispersion.

Se dice que se tiene una dispersión *cerrada* cuando las particiones de la estructura dispersa son estructuras acotadas. Si el tamaño de cada partición es N, resulta claro que pueden resolverse sin problemas las primeras N colisiones. Si se produjeran más de N colisiones, éstas pueden resolverse empleando:

- un área de desbordamiento común
- el espacio disponible en otras particiones de acuerdo con alguna política de asignación concreta
- otra función de dispersión

En esta práctica implementaremos el TAD APLICACION mediante dispersión cerrada mínima (es decir, el tamaño de las particiones es N=1), y empleando el espacio disponible de otras particiones cuando se produzcan colisiones, siguiendo para ello una estrategia de asignación lineal. Los detalles de implementación se exponen en el siguiente párrafo.

Dispersión cerrada mínima con resolución lineal de colisiones.

La implementación más simple (e incorrecta) de una dispersión cerrada mínima emplea un array de tipo ITEM indizado por el tipo SELECTOR:

```
TYPE TABLA_DISPERSION =  ARRAY SELECTOR OF ITEM;
```

Un dato x de tipo ITEM se almacenará en la casilla [Dispersion(x)]. Esta representación es incorrecta, pues no tenemos manera de averiguar si una determinada casilla del array está o no en uso. Esto puede solucionarse si en cada casilla del array almacenamos un registro con dos campos: el contenido de tipo ITEM y un valor de tipo ESTADO que nos indique el estado en que se encuentra tal casilla (OCUPADA, VACIA). Así obtenemos las declaraciones:

```
TYPE    ESTADO = (OCUPADA, VACIA);

        NODO= RECORD
                        Cont: ITEM;
                        Estado: ESTADO;
                END;

        TABLA_DISPERSION = ARRAY SELECTOR OF NODO;
```

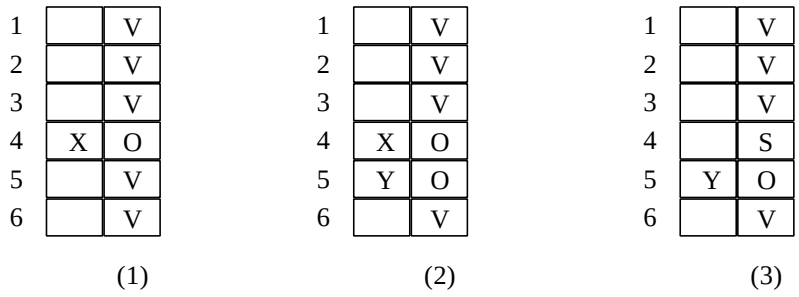
Se produce una colisión cuando la función de dispersión nos devuelve el índice de una casilla que está OCUPADA. Esto se resuelve intentando almacenar el elemento en la siguiente posición VACIA del array, recorriendo éste circularmente. A esta estrategia se la denomina resolución *lineal* de colisiones.

La representación anterior no es suficiente aún para implementar adecuadamente la dispersión, pues es necesario distinguir entre celdas *vacías* (aquéllas que nunca han sido empleadas) y *suprimidas* (aquéllas que fueron empleadas pero cuyo contenido ha sido eliminado). Para comprender esta necesidad, considérese el siguiente ejemplo:

(1) Se inserta un elemento X en la posición que indique la función de dispersión, por ejemplo la 4

(2) se inserta otro elemento Y cuya posición por la función de dispersión también es la 4. Como esta celda está OCUPADA, se produce una colisión que se resuelve linealmente, colocándose Y en la primera posición VACIA, en este caso en la 5

(3) posteriormente, se elimina el elemento X, marcando la casilla 4 como SUPRIMIDA, para indicar que quizá hay elementos que deberían ocupar esta posición pero al producirse colisión se han colocado en otra posición



Si no distinguiéramos entre celdas VACIAS y celdas SUPRIMIDAS, habríamos marcado la posición 4 como VACIA. En tal caso, en una posible búsqueda de Y al inspeccionar la posición 4 no se produciría ni una colisión ni se encontraría la Y. Sin embargo, no se puede asegurar que el elemento Y no está en la tabla, lo que nos obligaría a recorrer la tabla completamente. Sin embargo, al distinguir entre los dos estados (VACIA y SUPRIMIDA), si encontramos una celda marcada como VACIA sabemos que podemos detener la búsqueda, mientras que si encontramos una celda SUPRIMIDA hay que continuar la búsqueda circularmente hasta llegar otra vez a la posición inicial, encontrar una celda VACIA o el elemento buscado.

Mejora de la eficiencia mediante cachés.

Otra forma de mejorar el rendimiento de las estructuras de datos es incorporándoles una *caché*. La caché de una estructura de datos es similar en espíritu a las cachés disponibles en los sistemas de memoria o en los procesadores actuales: almacena el dato *más recientemente accedido* de la estructura de datos para tratar de acelerar futuras referencias al mismo.

Una caché básica está compuesta por dos campos:

- El dato cacheado propiamente dicho
- Un campo BOOLEANO que indica la validez del dato cacheado

Cuando se incorpora una caché a una estructura de datos debe tenerse especial cuidado en no alterar la semántica del TAD. En particular, es fundamental estudiar con detalle cómo afecta cada operación a la caché y viceversa. Básicamente, un procedimiento puede actuar como *lector* de la caché, como *escritor*, o bien no afectar ni verse afectado por la caché.

La tabla siguiente muestra la forma en que interactúa cada procedimiento del TAD APLICACION con la caché:

Procedimiento	Papel	Psedocódigo de la interacción con la caché
Crear()	Escritor	invalidar(caché)
Destruir(a)	No afecta	

EsVacia(a)	Lector	si la caché contiene un dato válido entonces
		  devolver FALSE
		si no
		  inspeccionar(a)

Enlazado(a, d)	Lector/Escritor	si la caché contiene d entonces
		  devuelve TRUE
		si no
		  buscar(a, d, r)
		  actualizar(caché)

Extension(a)	No afecta	

RangoDe(a, d)	Lector/Escritor	si la caché contiene d entonces
		  devolver r
		si no
		  buscar(d, r)
		  actualizar(caché)

Enlazar(a, d, r)	Escritor	insertar(d, r) actualizar(caché)
Desenlazar(a, d)	Escritor	si la caché contiene d entonces invalidar(caché) eliminar(d, r)
Copiar(a1,a2)	No afecta	
Aplicar(a, op)	No afecta	

Como se desprende de la tabla anterior, no todas las operaciones se ven afectadas por la caché en la misma medida. Hay operaciones cuya eficiencia no se ve en absoluto afectada (Crear, Destruir, Copiar,...); mientras que otras operaciones, especialmente los selectores (Enlazado, RangoDe), pueden mejorar considerablemente su rendimiento.

Representación del TAD Aplicación.

Todos los TADs genéricos que hemos estudiado hasta el momento importaban un solo tipo: el tipo base del TAD al que denominábamos tipo ITEM. El TAD APLICACION es un TAD genérico que importa dos tipos: DOMINIO y RANGO. Estos tipos son definidos por el usuario del TAD APLICACION en los módulos de definición Dominio y Rango, respectivamente, que además deben exportar ciertos procedimientos necesarios para la implementación del TAD APLICACION. En particular, el módulo Rango debe exportar la función ValorPorDefecto, pues el procedimiento RangoDe devuelve un valor de tipo RANGO.

El TAD APLICACION se implementará de forma acotada mediante dispersión cerrada mínima, con resolución de colisiones lineal y caché. La dispersión se aplicará al valor de tipo DOMINIO del par ordenado. Puesto que vamos a aplicar dispersión sobre el tipo DOMINIO, nos hará falta definir la función:

```
PROCEDURE Dispersion(d: DOMINIO): SELECTOR;
```

Aquí nos encontramos ante una paradoja. El tipo DOMINIO sólo es conocido por el usuario del TAD, así que debe ser él quien defina la función Dispersion. Por otro lado, el tipo SELECTOR es conocido solamente por el implementador del TAD, así que debería ser él quien definiera la función Dispersion. Para resolver este problema, dejaremos que sea el usuario del TAD que defina la función de Dispersion devolviendo un tipo CARDINAL en lugar de SELECTOR, por lo que en el módulo de definición Dominio queda como sigue:

```
DEFINITION MODULE Dominio;

    TYPE DOMINIO = << a definir >>;

    PROCEDURE Dispersion(d: DOMINIO): CARDINAL;

END Dominio.
```

La representación de la APLICACION, sin embargo, se basa en el tipo SELECTOR:

```
CONST MAX_SELECTOR = <<a definir>>;

TYPE  SELECTOR= [1.. MAX_SELECTOR];

      ESTADO   = (OCUPADA, VACIA, SUPRIMIDA);

      CELDA  =  RECORD
                    dom: DOMINIO;
                    rang: RANGO;
                    est:  ESTADO;
                END;

      TABLA  = ARRAY SELECTOR OF CELDA;


      CACHE = RECORD
                    dom: DOMINIO;
                    rang: RANGO;
                    valido: BOOLEAN;
              END;

      APLICACION = POINTER TO ESTRUCTURA;

      ESTRUCTURA = RECORD
                          mem_tab: TABLA;
                          mem_cache: CACHE;
                   END;
```

Esto significa que los valores devueltos por la función Dispersion definida en el módulo Dominio deben ser normalizados en la implementación del TAD APLICACION, aplicando la siguiente conversión:

```
1 +  (Dispersion(d)  MOD MAX_SELECTOR)
```

Para facilitar la implementación, resulta conveniente definir un procedimiento interno que permita localizar un enlace en la tabla a partir del valor del dominio d:

```
PROCEDURE Buscar(t: TABLA; d: DOMINIO): CARDINAL;
```

Si d está enlazado, devuelve su posición en la tabla. Si d no está enlazado y la tabla no está llena, devuelve el índice de la primera celda disponible en la tabla, y si d no está en la tabla y la tabla está llena, devuelve 0. Para efectuar esta búsqueda, se debe comenzar en la posición de la tabla devuelta por Dispersion(d) y continuar circularmente.

Finalmente, construir un programa de prueba para este TAD.

Práctica Suplementaria

Construir el TAD APLICACION con una implementación acotada, mediante dispersión cerrada con N= *k* (para *k*>1), resolución lineal de colisiones y con caché.