Capítulo 4 Tablas

Se quiere estudiar el TAD de las funciones $f : K \rightarrow V$ que se aplican sobre un dominio K y dan un resultado sobre un codominio V; a los valores del dominio los llamamos claves (ing., key; también denominados índices o identificadores) y a los valores del codominio, información o simplemente valores (ing., value); si $f(k) = v$, diremos que v es la información asociada o asignada a k. En el caso general, el dominio K tendrá una cardinalidad muy grande y nuestras implementaciones deberán tratar adecuadamente esta característica. Las funciones que consideramos en este capítulo pueden ser totales o parciales; en el segundo caso, llamaremos clave indefinida a toda clave que no esté en el dominio de la función. Dado este modelo matemático queda claro que, así como las estructuras lineales están orientadas al acceso consecutivo a todos sus elementos (ya sea ordenadamente o no), las tablas están orientadas a su acceso individual: interesa definir la función en un punto del dominio, dejar este punto indefinido y aplicar la función sobre él.

El TAD de las funciones es conocido en el ámbito de las estructuras de la información con el nombre de tabla (ing., lookup table o symbol table, aunque la notación no es totalmente estándar; algunos textos también lo denominan estructura funcional o diccionario). En este contexto, es frecuente considerar las tablas como conjuntos de pares de clave y de valor, con la restricción de que no haya dos pares con la misma clave, de manera que la clave identifica unívocamente el par. A los pares los llamaremos elementos definidos o, simplemente, elementos de la tabla.

La aplicación del tipo en el campo de la programación es diversa; por ejemplo, se emplea para implementar tablas de símbolos de compiladores o de sistemas operativos: la clave es el identificador de los símbolos que se almacenan y la información puede ser muy diversa (direcciones de memoria, dimensión, etc.).

En el resto del capítulo se estudia la especificación y la implementación del TAD; de manera especial, se busca una implementación con el objetivo de que las operaciones tengan un coste temporal $\Theta(1)$, lo que conseguiremos con las denominadas tablas de dispersión.

4.1 Especificación

Sean K y V dos dominios tales que V presenta un valor indefinido que denotaremos por $\perp$, sea $f : K \rightarrow V$ una tabla y sean $k \in K$ una clave y $v \in V$ un valor. Las operaciones del TAD son:

- Crear la tabla vacía: crea, devuelve la función Ø definida como dom(Ø) = Ø.
- Asociar información a una clave: asigna(f, k, v), devuelve la función g definida como:
  - ◊ $dom(g) = dom(f) \cup \{k\} \wedge g(k) = v$.
  - ◊ $\forall k' : k' \in dom(g) - \{k\} : g(k') = f(k')$.
- Separar la información correspondiente a una clave: borra(f, k), devuelve la función g definida como:
  - ◊ $dom(g) = dom(f) - \{k\}$.
  - ◊ $\forall k' : k' \in dom(g) : g(k') = f(k')$.
- Consultar el valor asociado a una clave: consulta(f, k), devuelve f(k) si $k \in dom(f)$, y devuelve $\perp$ si $k \notin dom(f)$.

Notemos que, en realidad, el modelo aquí presentado se corresponde con las funciones totales, ya que toda clave tiene un valor asociado. En caso de que el valor indefinido no se pueda identificar en V, se puede forzar que la función consulta devuelva un booleano adicional indicando si la clave estaba o no definida, o bien se puede añadir una operación más, definida?, que lo averigüe, de manera que consulta o borra sobre una clave no definida dé error, y entonces el modelo de las tablas corresponda realmente a las funciones parciales.

En la fig. 4.1 se presenta una especificación parametrizada para el modelo de las funciones totales con la signatura introducida arriba. Sus parámetros formales son: los géneros para las claves y los valores, la igualdad de las claves y el valor indefinido. Por ello, usamos los universos de caracterización ELEM_= y ELEM_ESP: el primero define las claves y el segundo los valores; algunos de estos símbolos se renombran, por motivos de legibilidad[1]. Por lo que al resto del universo se refiere, notemos la semejanza con la especificación típica de los conjuntos. Destaquemos que la función borra deja la tabla inalterada si la clave no existe, que la operación consulta devuelve el valor indefinido si la clave buscada está indefinida y que la operación asigna, si la clave ya está definida, le asigna un nuevo valor (otra opción sería dejarla inalterada, o bien que la función asigna devolviera un segundo valor tal que, si la clave estuviera definida, devolviera el valor asociado a la clave y dejara la tabla inalterada, o bien asociara a la clave el nuevo valor). Notemos también la ecuación de error que evita la inserción del valor indefinido como pareja de una clave.

En una instancia de las tablas es necesario asociar símbolos a todos estos parámetros formales; por ejemplo, podemos definir una tabla cuyas claves sean enteros, la información cadenas de caracteres y el valor indefinido una cadena especial, no válida como información normal (algo así como "*&%$#!", por ejemplo).

[1] Los renombramientos de los parámetros formales sólo tienen vigencia en el universo que los efectúa.

universo TABLA (ELEM_=, ELEM_ESP) es

renombra ELEM_=.elem por clave
ELEM_ESP.elem por valor, ELEM_ESP.esp por indef

usa BOOL

tipo tabla

ops crea: → tabla
asigna: tabla clave valor → tabla
borra: tabla clave → tabla
consulta: tabla clave → valor

error ∀k∈clave; ∀t∈taula: asigna(t, k, indef)

ecns ∀k,k_1,k_2∈clave; ∀v,v_1,v_2∈valor; ∀t∈taula

Ecuaciones impurificadoras:
1) asigna(asigna(t, k, v_1), k, v_2) = asigna(t, k, v_2)
2) [$k_1 \neq k_2$] ⇒ asigna(asigna(t, k_1, v_1), k_2, v_2) =
asigna(asigna(t, k_2, v_2), k_1, v_1)

Axiomas para borra:
1) borra(crea, k) = crea
2) borra(asigna(t, k, v), k) = borra(t, k)
3) [$k_1 \neq k_2$] ⇒ borra(asigna(t, k_1, v_1), k_2) = asigna(borra(t, k_2), k_1, v_1)

Axiomas para consulta:
1) consulta(crea, k) = indef
2) consulta(asigna(t, k, v), k) = v
3) [$k_1 \neq k_2$] ⇒ consulta(asigna(t, k_1, v_1), k_2) = consulta(t, k_2)

funiverso

universo ELEM_ESP caracteriza
tipo elem
ops esp: → elem
funiverso

Fig. 4.1: especificación del TAD tabla.

Una variante del TAD tabla bastante usada consiste en considerar el modelo de los conjuntos de elementos de V, P(V), con operaciones de inserción y supresión individual de elementos y de comprobación de pertenencia, siendo V un dominio que presenta una función id : V → K, de manera que cada elemento del conjunto esté identificado por una clave que sirva para hacerle referencia; a veces, el identificador será el mismo elemento y entonces V y K serán el mismo género e id será la función identidad, y el resultado será equivalente al universo CJT_$\in$ de la fig. 1.29 con el añadido de la supresión. La signatura

propuesta para este modelo aparece en la fig. 4.2 y su especificación queda como ejercicio para el lector. Las implementaciones y los algoritmos sobre tablas que aparecen en el resto del capítulo se pueden adaptar a este TAD con muy poco esfuerzo.

```
universo CONJUNTO (ELEM_CJT) es
    usa BOOL
    tipo conjunto
    ops
        crea: → conjunto
        añade: conjunto elem → conjunto
        borra: conjunto clave → conjunto
        consulta: conjunto clave → elem
        está?: conjunto clave → bool
funiverso
```

```
universo ELEM_CJT caracteriza
    usa ELEM_ESP
    tipo clave
    ops
        id: elem → clave
        _=_, _≠_: clave clave → bool
    ecns ... las de _=_ y _≠_
funiverso
```

Fig. 4.2: una signatura para el TAD de los conjuntos.

4.2 Implementación

Estudiamos en esta sección las diferentes posibilidades para implementar el TAD de las tablas. Comenzamos usando diversas variantes de listas hasta llegar a las tablas de dispersión, que son de la máxima eficiencia para las operaciones del tipo. A lo largo de la sección usaremos dos medidas de eficiencia que nos permitan comparar las organizaciones:

- $S_n(k)$: número necesario de comparaciones entre claves en una tabla de n elementos para encontrar el elemento de clave k (ing., successful search).
- $U_n(k)$: número necesario de comparaciones entre claves en una tabla de n elementos en una búsqueda sin éxito del elemento de clave k (ing., unsuccessful search).

Notemos que $S_n(k)$ es un factor importante en el cálculo del coste de la modificación y la supresión de elementos ya existentes y en la consulta de claves definidas, mientras que $U_n(k)$ determina el coste de la inserción de nuevos elementos dentro de la tabla, de la supresión de elementos dando como referencia una clave indefinida y de la consulta de claves indefinidas. Generalmente, a partir de los valores de $S_n(k)$ y $U_n(k)$ se puede determinar el coste de las operaciones del TAD; ahora bien, a veces puede haber otros factores que influyan en la eficiencia (por ejemplo, seguimiento de encadenamientos, movimiento de elementos, etc.), en cuyo caso se destacará explícitamente.

A veces, estas magnitudes serán variables aleatorias, en cuyo caso $\varepsilon[S_n(k)]$ y $\varepsilon[U_n(k)]$ serán sus esperanzas[2].

[2] Pese a que sería interesante, en este libro no se estudian otras medidas estadísticas más que a nivel muy superficial y en algún caso concreto; para profundizar en el tema, consultar [GoB91] y [Knu73].

4.2.1 Implementación por listas desordenadas

Consiste en organizar una lista no ordenada cuyos elementos sean las claves definidas con su valor asociado. Claramente se cumple que $\varepsilon[S_n(k)] = (n+1)/2$ y $U_n(k) = n$. Así, el coste asintótico de las operaciones del TAD tabla es $\Theta(n)$, frente a otros esquemas logarítmicos o constantes que se introducen a continuación. No obstante, su simplicidad hace posible su elección, para n pequeña, o si no hay restricciones de eficiencia.

Es interesante conocer un par de estrategias útiles en determinadas circunstancias (aunque no reducen el coste asintótico):

- Si se dispone a priori de las probabilidades de consulta de cada clave y, a partir de un momento dado, las actualizaciones son relativamente irrelevantes respecto a las consultas, se pueden mantener ordenadas las claves según estas probabilidades dentro de la lista, y se obtiene como resultado $\varepsilon[S_n(k)] = \Sigma x : 1 \leq x \leq n : x.p_x$, donde p_x representa la probabilidad de consulta de la x-ésima clave más probable.
- En las búsquedas con éxito puede reorganizarse la lista para que los elementos que se consulten queden cerca del inicio, con la suposición de que las claves no son equiprobablemente consultadas y que las más consultadas hasta al momento son las que se consultarán más en el futuro; es decir, se tiende precisamente a una lista ordenada por probabilidades. Esta estrategia se conoce con el nombre de búsqueda secuencial autoorganizativa (ing., self-organizing sequential search) y hay varias implementaciones posibles entre las que destacamos la denominada búsqueda con movimiento al inicio (ing., move to front), en la que el elemento consultado se lleva al inicio de la lista, y la denominada búsqueda por transposición (ing., transpose), donde se intercambia el elemento consultado con su predecesor. En general, el segundo método da mejores resultados que el primero cuando las probabilidades de consulta de las diversas claves no varían en el tiempo (los saltos de los elementos dentro la lista son más cortos, con lo que su distribución es más estable); por otro lado, si la distribución es muy sesgada el primer método organiza los elementos más rápidamente. Notemos también que, en el caso de representar las listas secuencialmente, los desplazamientos exigen copias de elementos con los problemas mencionados en el apartado 3.3.2.a).

4.2.2 Implementación por listas ordenadas

Aplicable sólo si las claves presentan una operación < de comparación que defina un orden total, es una organización útil cuando las consultas predominan sobre las actualizaciones, porque aunque estas últimas queden $\Theta(n)$, las primeras reducen su coste. También pueden usarse si la tabla tiene dos estados bien diferenciados en el tiempo, el primero en el que predominen las actualizaciones y el segundo donde predominen las consultas; entonces se

puede mantener la lista desordenada durante el primer estado y aplicar un algoritmo de ordenación de coste $\Theta(n\log n)$ antes de comenzar el segundo. En todo caso, aparte del coste temporal, es necesario recordar que, si la implementación elegida para las listas es secuencial, las actualizaciones exigen movimientos de elementos. A continuación, citamos los tres esquemas de búsqueda habituales.

a) Lineal (secuencial)

Ya presentado en la sección 3.3, consiste en recorrer la lista desde el principio hasta encontrar el elemento buscado o llegar al final. La búsqueda sin éxito acaba antes que en el caso desordenado, $\varepsilon[U_n(k)] = \varepsilon[S_n(k)] = (n+1)/2$, aunque el coste asintótico sigue siendo lineal.[3]

b) Dicotómica

También conocida como búsqueda binaria (ing., dichotomic o binary search), es aplicable sólo cuando la lista se representa secuencialmente dentro de un vector y consiste en descartar sucesivamente la mitad de las posiciones del vector hasta encontrar el elemento buscado, o bien acabar. Para ello, a cada paso se compara el elemento buscado con el elemento contenido en la posición central del trozo de vector en examen; si es más pequeño, se descartan todos los elementos de la derecha, si es más grande, los de la izquierda, y si es igual, ya se ha encontrado.

En la fig. 4.3 se da una versión iterativa del algoritmo de búsqueda dicotómica. Suponemos que las tablas se representan por un vector A dimensionado de uno a un máximo predeterminado y un apuntador sl de sitio libre; suponemos también que cada posición del vector es una tupla de clave y valor. Los apuntadores izq y der delimitan el trozo del vector en examen, tal como establece el invariante; debemos destacar, sin embargo, que para que el invariante sea correcto es necesario imaginar que las posiciones 0 y sl de la tabla están ocupadas por dos elementos fantasma tales que sus claves son, respectivamente, la clave más pequeña posible y la clave más grande posible dentro del dominio de las claves. El análisis del algoritmo revela que $S_n(k)$ y $U_n(k)$ valen aproximadamente $\log_2 n$; el coste de la consulta es, pues, $\Theta(\log n)$.

[3] Hay un algoritmo que asegura un coste $O(\sqrt{n})$ en el caso medio, pero que queda $\Omega(n)$ en el caso peor, que se basa en la inserción de nuevos encadenamientos, que permiten efectuar saltos más largos. En [BrB87, pp. 248-250] se presenta una versión probabilistica de este algoritmo, que consigue reducir el coste del caso peor al caso medio.

```
función consulta (t es tabla; k es clave) devuelve valor es
var izq, der, med son nat; encontrado es bool; v es valor fvar
    izq := 0; der := t.sl; encontrado := falso
    mientras (der - izq > 1) ∧ ¬ encontrado hacer
            { I ≅ 0 ≤ izq < der ≤ t.sl ∧ t.A[izq].k < k < t.A[der].k
                ∧ encontrado ⇒ t.A[med].k = k }
        med := (izq + der) / 2
        opción
            caso t.A[med].k = k hacer encontrado := cierto
            caso k < t.A[med].k hacer der := med
            caso t.A[med].k < k hacer izq := med
        fopción
    fmientras
    si encontrado entonces v := t.A[med] si no v := indefinido fsi
devuelve v
```

Fig. 4.3: algoritmo de búsqueda dicotómica.

c) Por interpolación

También para listas representadas secuencialmente, la búsqueda por interpolación (ing., interpolation search; también conocida como estimated entry search) sigue la misma estrategia que la anterior con la diferencia de que, en vez de consultar la posición central, se consulta una posición que se interpola a partir del valor de la clave que se quiere buscar y de los valores de la primera y la última clave del trozo del vector en examen. Si suponemos que las claves se distribuyen uniformemente entre sus valores mínimo y máximo, el resultado es que tanto $\varepsilon[S_n(k)]$ como $\varepsilon[U_n(k)]$ valen $\Theta(\log\log n)$, con la función de interpolación F:

$$F(A, k, i, j) = \lfloor \{ (k-A[i]) \ / \ (A[j]-A[i]) \} \ . \ (j-i) \rfloor + i$$

Si la distribución no es uniforme, aunque teóricamente se podría corregir la desviación para mantener este coste bajo (conociendo a priori la distribución real de las claves), en la práctica sería complicado; por ejemplo, si $\forall p: 1 \le p \le n-1$: $A[p] = p$ y $A[n] = \infty$, en el caso de buscar el elemento n-1, el coste resultante es lineal. Para evitar este riesgo, en [GoB91, pp. 42-43] se propone combinar un primer paso de acceso a la tabla usando el algoritmo de interpolación y, a continuación, buscar secuencialmente en la dirección adecuada.

Notemos que la búsqueda por interpolación dentro de una lista sólo funciona cuando las claves son numéricas (o tienen una interpretación numérica), y no están repetidas (como es el caso que aquí nos ocupa, en el que la lista representa una tabla).

4.2.3 Implementación por vectores de acceso directo

Una mejora evidente del coste de las operaciones del TAD es implementar la tabla sobre un vector de manera que cada clave tenga asociada una y sólo una posición del vector. En otras palabras, dada una función g: K $\rightarrow$ Z_n, donde Z_n representa el intervalo cerrado [0, n-1], y siendo n la cardinalidad del conjunto de claves, la tabla se puede representar con un vector A tal que, dada una clave k, en A[g(k)] esté el valor asociado a k; si la clave está indefinida, en A[g(k)] residirá el valor indefinido o, si no existe ningún valor que pueda desempeñar este papel, en cada posición añadiremos un campo booleano adicional que marque la indefinición. Con esta representación, es obvio que todas las operaciones se implementan en orden constante.

La función g ha de ser inyectiva como mínimo; mejor aún si es biyectiva, porque en caso contrario habrá posiciones del vector que siempre quedarán sin usar. Incluso en el último caso, sin embargo, el esquema es impracticable si n es muy grande y el porcentaje de claves definidas es muy pequeño (lo que implica que la mayoría de las posiciones del vector estarán sin usar), que es lo más habitual.

4.2.4 Implementación por tablas de dispersión

Las tablas de dispersión (ing., hashing table o también scatter table; to hash se traduce por "desmenuzar"), también conocidas como tablas de direccionamiento calculado, se parecen al esquema anterior en que utilizan una función h : K $\rightarrow$ Z_r que asigna un entero a una clave, pero ahora sin requerir que h sea inyectiva, lo que divide el dominio de las claves en r clases de equivalencia y quedan dentro de cada clase todas las claves tales que, al aplicarles h, dan el mismo valor. A h la llamamos función de dispersión (ing., hashing function), y los valores de Z_r son los valores de dispersión (ing., hashing values). Lo más importante es que, a pesar de la no-inyectividad de h (que permite aprovechar mucho más el espacio, porque sólo se guardan los elementos definidos de la tabla), el coste de las operaciones se puede mantener constante con gran probabilidad.

Como antes, se usa h(k) para acceder a un vector donde residan las claves definidas junto con su valor (más otra información de gestión de la tabla que ya introduciremos). Como no se requiere que h sea inyectiva, se define el vector con una dimensión más pequeña que en el esquema anterior y se asigna cada clase de equivalencia a una posición del vector. A las posiciones del vector algunos autores las llaman cubetas (ing., bucket); esta terminología se usa sobre todo al hablar de dispersión sobre memoria secundaria, donde en lugar de vectores se usan ficheros.

Quedan dos cuestiones por resolver:

- ¿Qué forma toma la función de dispersión? Ha de cumplir ciertas propiedades, como distribuir bien las claves y ser relativamente rápida de calcular.
- ¿Qué pasa cuando en la tabla se quieren insertar claves con idéntico valor de dispersión? Si primero se inserta una clave k, tal que h(k) = i, y después otra k' , tal que h(k') = i, se dice que se produce una colisión (ing., collision), porque k' encuentra ocupada la posición del vector que le corresponde; a k y k' los llamamos sinónimos (ing., synonym). El tratamiento de las colisiones da lugar a varias estrategias de implementación de las tablas de dispersión.

4.3 Funciones de dispersión

En esta sección se determinan algunos algoritmos que pueden aplicarse sobre una clave para obtener valores de dispersión. Comencemos por establecer las propiedades que debe cumplir una buena función de dispersión:

- Distribución uniforme: todos los valores de dispersión deben tener aproximadamente el mismo número de claves asociadas dentro de K, es decir, la partición inducida por la función da lugar a clases de equivalencia de tamaño parecido.
- Independencia de la apariencia de la clave: pequeños cambios en la clave han de resultar en cambios absolutamente aleatorios del valor de dispersión; además, las variaciones en una parte de la clave no han de ser compensables con variaciones fácilmente calculables en otra parte.
- Exhaustividad: todo valor de dispersión debe tener como mínimo una clave asociada, de manera que ninguna posición del vector quede desaprovechada a priori.
- Rapidez de cálculo: los algoritmos han de ser sencillos y, si es necesario, programados directamente en lenguaje máquina o bien en lenguajes como C, que permiten manipulación directa de bits; las operaciones necesarias para ejecutar los algoritmos deben ser lo más rápidas posible (idealmente, desplazamientos o rotaciones de bits, operaciones lógicas y similares).

Es necesario señalar que la construcción de computadores cada vez más potentes relativiza la importancia del último criterio al diseñar la función; ahora bien, sea cual sea la potencia de cálculo del computador parece seguro que operaciones como la suma, los desplazamientos y los operadores lógicos serán siempre más rápidas que el producto y la división, por ejemplo, y por esto seguimos considerando la rapidez como un parámetro de diseño.

Por otro lado, y este es un hecho fundamental para entender la estrategia de dispersión, las dos primeras propiedades se refieren al dominio potencial de claves K, pero no aseguran el buen comportamiento de una función de dispersión para un subconjunto A de claves,

siendo $A \subseteq K$, porque puede ocurrir que muchas de las claves de A tengan el mismo valor de dispersión y que algunos de estos valores, por el contrario, no tengan ninguna antiimagen en A. Precisamente por esto, una función de dispersión teóricamente buena puede dar resultados malos en un contexto de uso determinado. Si se considera que este problema es grave, en las implementaciones de tablas que se introducen en la próxima sección se puede añadir código para controlar que el número de colisiones no sobrepase una cota máxima. En todo caso, esta propiedad indeseable de la dispersión siempre debe ser tenida en cuenta al diseñar estructuras de datos.

Es natural preguntarse, antes de empezar a estudiar estrategias concretas, qué posibilidades existen de encontrar buenas funciones de dispersión. Si se quiere almacenar n claves en una tabla de dimensión r, hay r^n posibles configuraciones, cada una de ellas obtenida mediante una hipotética función de dispersión; si n < r, sólo r ! / (r - n)! de estas funciones no provocan ninguna colisión. Por ejemplo, D.E. Knuth presenta la "paradoja del cumpleaños": si acuden veintitrés personas a una fiesta, es más probable que haya dos de ellas que celebren su cumpleaños el mismo día que no lo contrario [Knu73, pp. 506-507]. Por ello, lo que pretendemos no es buscar funciones que no provoquen ninguna colisión sino funciones que, en el caso medio, no provoquen muchas[4]. Debe tenerse en cuenta, además, que es teóricamente imposible generar datos aleatorios a partir de claves no aleatorias, pero lo que se busca aquí es una buena simulación, lo que sí que es posible; incluso una función de dispersión puede provocar menos colisiones que una auténtica función aleatoria.

Analizaremos las funciones de dispersión según dos enfoques diferentes:

- Considerando directamente las funciones $h : K \to Z_r$.
- Considerando las funciones $h : K \to Z_r$ como la composición de dos funciones diferentes, $f : K \to Z$ y $g : Z \to Z_r$.

Asimismo, a partir de ahora consideraremos que las claves son, o bien enteros dentro de un intervalo cualquiera, o bien cadenas de caracteres (las cadenas de caracteres las denotaremos por C^*, siendo C el conjunto válido de caracteres de la máquina), porque es el caso usual. A veces, no obstante, nos encontramos con que las claves han de considerarse como la unión o composición de diversos campos, y es necesario adaptar los algoritmos aquí vistos a este caso, sin que generalmente haya mayores problemas. Cuando las claves sean numéricas, la función h será directamente igual a g.

[4] Si la tabla es estática y las claves se conocen a priori, hay técnicas que permiten buscar funciones que efectivamente no generen sinónimos; son las denominadas técnicas de dispersión perfecta (ing., perfect hashing), que no estudiamos aquí. Se puede consultar el articulo de T.G. Lewis y C.R. Cook "Hashing for Dynamic and Static Internal Tables" (publicado en el Computer IEEE, 21(10), 1988, es un resumen de la técnica de dispersión en general) para una introducción al tema y [Meh84] para un estudio en profundidad.

4.3.1 Funciones de traducción de cadenas a enteros

Normalmente, las funciones $f: C^* \rightarrow Z$ interpretan cada carácter de la cadena como un número, y se combinan todos los números así obtenidos mediante algún algoritmo; normalmente, estos números son naturales entre 0 y 127 ó 255, según el código concreto de la máquina.

El primer problema que se presenta es que seguramente las claves no usarán todos los caracteres del código. Por ejemplo, imaginemos el caso habitual en que las cadenas de caracteres representan nombres (de personas, de lugares, etc.); es evidente que algunos caracteres válidos del código de la máquina no aparecerán nunca (como '&', '%', etc.), por lo que su codificación numérica no se usará. Por ejemplo, si el código de la máquina es ASCII estándar (con valores comprendidos en el intervalo [0, 127]) y en los nombres sólo aparecen letras y algunos caracteres especiales más (como '.' y '-'), las cadenas estarán formadas por un conjunto de 54 caracteres del código (suponiendo que se distingan mayúsculas de minúsculas); es decir, más de la mitad del código quedará desaprovechado y por ello los valores generados no serán equiprobables (es más, puede haber valores inalcanzables). Para evitar una primera pérdida de uniformidad en el tratamiento de la clave es conveniente efectuar una traducción de códigos mediante una función $\phi: C \rightarrow [1, b]$, siendo b el número diferente de caracteres que pueden formar parte de las claves, de manera que la función de conversión f se define finalmente como $f(k) = f'(\Phi(k))$, siendo Φ la extensión de ϕ a todos los caracteres de la clave y f' una función $f': [1, b]^* \rightarrow Z$. Esta conversión será más o menos laboriosa según la distribución de los caracteres dentro del código usado por la máquina.

A continuación, se definen algunes funciones f aplicables sobre una clave $k = c_n \ldots c_1$:

- Suma de todos los caracteres: $f(k) = \Sigma i: 1 \leq i \leq n: \phi(c_i)$. Empíricamente se observa que los resultados de esta función son poco satisfactorios, fundamentalmente porque el valor de un carácter es el mismo independientemente de la posición en que aparezca.
- Suma ponderada de todos los caracteres: modifica la fórmula anterior considerando que los caracteres tienen un peso asociado que depende de la posición donde aparecen y queda: $f(k) = \Sigma i: 1 \leq i \leq n: \phi(c_i).b^{i-1}$. Los resultados son espectacularmente mejores que en el método anterior y por ello esta función es muy usada, aunque presenta dos problemas que es necesario conocer:
  - ◊ Es más ineficiente, a causa del cálculo de una potencia y del posterior producto a cada paso del sumatorio. La potencia se puede ahorrar introduciendo una tabla auxiliar T, tal que $T[i] = b^{i-1}$, que se inicializará una única vez al principio del programa y que generalmente será lo suficientemente pequeña dentro del contexto de los datos del programa como para despreciar el espacio que ocupa. Otra opción consiste en elevar, no b, sino la menor potencia de 2 más grande que b, de manera que los productos y las potencias puedan implementarse como operaciones de desplazamiento de bits; los resultados de esta variante no son tan

malos como se podría temer (dependiendo de la diferencia entre b y la potencia de 2). Es necesario controlar, no obstante, que los desplazamientos no provoquen apariciones reiteradas de ceros dentro de los sumandos parciales ni apariciones cíclicas de la misma secuencia de números.

◊ Se puede producir un desbordamiento en el cálculo, que es necesario detectar antes que se produzca; por ejemplo, si b vale 26, en una máquina que use 32 bits para representar los enteros puede producirse desbordamiento a partir del octavo carácter. Una solución consiste en ignorar los bits que se van generando fuera del tamaño de la palabra del computador, pero esta estrategia ignora cierta cantidad de información reduciendo pues la aleatoriedad de la clave; además, no siempre es posible y/o eficiente detectar e ignorar este desbordamiento.

Una alternativa es dividir la clave de entrada en m trozos de tamaño k bits (excepto uno de ellos si el resto de la división m /k es diferente de cero) aplicando entonces la fórmula de la suma ponderada sobre cada uno de estos trozos y combinando los resultados parciales mediante sumas. El tamaño k debe verificar que el cálculo de cada trozo por separado no provoque desbordamiento, y tampoco su suma final. Es decir, k es el mayor entero que cumple $m.b^k \leq$ maxent, siendo maxent el mayor entero representable en la máquina.

- Cualquier otra variante del método anterior hecha a partir de estudios empíricos, cuando se tiene una idea aproximada de las distribuciones de los datos. Por ejemplo, en el artículo "Selecting a Hashing Algorithm", de B.J. McKenzie, R. Harries y T. Bell, Software-Practice and Experience, 20(2), 1990, se presentan algunas funciones de dispersión usadas en la construcción de compiladores, que utilizan otros valores para ponderar los caracteres (potencias de dos, que son rápidas de calcular o números primos, que distribuyen mejor).

4.3.2 Funciones de restricción de un entero en un intervalo

En la literatura sobre el tema se encuentran gran cantidad de funciones; destacamos tres:

- División (ing., division): definida como g(z) = z mod r, es muy sencilla y rápida de calcular. Evidentemente, el valor r es crucial para una buena distribución. Así, por ejemplo, una potencia de 2 favorece la eficiencia, pero la distribución resultante será defectuosa, porque simplemente tomaría los $\log_2 r$ bits menos significativos de la clave; tampoco es bueno que r sea par, porque la paridad de z se conservaría en el resultado; ni que sea un múltiplo de 3, porque dos valores z y z' que sólo se diferencien en la posición de dos bytes tendrían el mismo valor. En general, es necesario evitar los valores de r tales que $(b^s \pm a) \bmod r = 0$, porque un módulo con este valor tiende a ser una superposición de los bytes de z, siendo lo más adecuado un r primo tal que $b^s \bmod r \neq \pm a$, para s y a pequeños. En la práctica, es suficiente que r no tenga divisores más pequeños que 20. Se puede encontrar un ejemplo de elección de r en [Meh84, pp. 121-122].

- Desplegamiento-plegamiento (ing., folding-unfolding): se divide la representación binaria de la clave numérica z en m partes de la misma longitud k (excepto posiblemente la última), normalmente un byte. A continuación, se combinan las partes según una estrategia determinada (por ejemplo, con sumas o o-exclusivos) obteniéndose un valor $r = 2^k+\varepsilon$, dependiente ε del método concreto de combinación (por ejemplo, si usamos o-exclusivos, $\varepsilon = 0$).
- Cuadrado: se define g(z) como la interpretación numérica de los $\lceil \log_2 r \rceil$ bits centrales de z^2. Conviene que r sea potencia de 2. Esta función distribuye bien porque los bits centrales de un cuadrado no dependen de ninguna parte concreta de la clave, siempre que no haya ceros de relleno al inicio o al final del número. Debe controlarse el posible desbordamiento que pueda provocar el cálculo del cuadrado. En [AHU83, p.131] se presenta una variante interesante.

El problema de todos estos métodos es que pueden distribuir mal una secuencia dada de claves, en cuyo caso es necesario encontrar una nueva función, pero, ¿con qué criterios? ¿Y si esta nueva función tampoco se comporta correctamente? Un enfoque diferente consiste en no trabajar con funciones individuales sino con familias de funciones de modo que, si una de estas funciones se comporta mal para una distribución dada de la entrada, se elija aleatoriamente otra de la misma familia; son las denominadas familias universales (ing., universal class) de funciones de dispersión, definidas por J.L. Carter y M.N. Wegman el año 1979 en "Universal Classes of Hash Functions", Journal of Computer and System Sciences, 18. De manera más formal, dada una familia de funciones H, $H \subseteq \{g : Z_a \rightarrow Z_r\}$, siendo a el valor más grande posible después de aplicar f a la clave, diremos que H es una familia 2-universal si: $\forall x,y : x,y \in Z_a : ||\{h \in H / h(x) = h(y)\}|| \leq ||H|| / r$ o, en otras palabras, si ningún par de claves colisiona en exceso sobre el conjunto de todas las funciones de H (precisamente, el prefijo "2" de la definición pone el énfasis en el estudio del comportamiento sobre pares de elementos), de manera que si se toma aleatoriamente una función de H y se aplica sobre un conjunto cualquiera de datos $A, A \subseteq K$, es altamente probable que esta función sea tan buena (es decir, que distribuya las claves igual de bien) como otra función diseñada específicamente para K. Informalmente, se está afirmando que dentro de H hay un número suficiente de "buenas" funciones para esperar que aleatoriamente se elija alguna de ellas.

Una consecuencia de la definición de 2-universal es que, dado un conjunto A cualquiera de claves y dada $k \in K$ una clave cualquiera tal que $k \notin A$, el número esperado de colisiones de k con las claves de A usando una función f de 2-universal está acotado por $||A|| / r$, que es una magnitud que asegura el buen comportamiento esperado de la función. Es más, la variancia de este valor está acotada para cada consulta individual, lo que permite forzar que dichas consultas cumplan restricciones temporales (v. "Analysis of a universal class of functions", G. Markowsky, J.L. Carter y M.N. Wegman, en los Proceedings 7th Workshop of Mathematical Foundations of Computer Science, LNCS 64, Springer-Verlag).

Carter y Wegman presentan tres clases de 2-universal, adoptadas también por otros autores (principalmente las dos primeras, posiblemente con pequeñas variaciones), que exhiben el mismo comportamiento y que pueden evaluarse de modo eficiente:

- H_1: sea p un número primo tal que $p \geq a$, sea g_1: $Z_p \rightarrow Z_r$ una función uniforme cualquiera (por ejemplo, $g_1(z) = z \bmod r$), sea $g_2^{m,n}$: $Z_a \rightarrow Z_p$ una familia de funciones tales que $m,n \in Z_a$, $m \neq 0$, definida como $g_2^{m,n}(z) = (m.z + n) \bmod p$ y, finalmente, sea $g_{m,n}$: $Z_a \rightarrow Z_r$ una familia de funciones definida como $g_{m,n}(z) = g_1(g_2^{m,n}(z))$. Entonces se cumple que la clase $H_1 = \{g_{m,n} / m,n \in Z_a \text{ con } m \neq 0\}$ es 2-universal. El artículo original de Carter y Wegman da una propiedad sobre p que, si se cumple, reduce el número de divisiones del algoritmo de dispersión.
- H_3: en este caso, es necesario que r sea potencia de 2, $r = 2^s$. Sea $j = \lceil \log_2 a \rceil$ (es decir, j es el número mínimo de bits necesarios para codificar cualquier valor de Z_a), sea Ψ el conjunto de matrices M_{jxs} de componentes 0 ó 1, sea g_M: $Z_a \rightarrow Z_r$ una familia de funciones definida como $g_M(z) = [z_j \otimes_s (m_{j1}...m_{js})] \oplus_s ... \oplus_s [z_1 \otimes_s (m_{11}...m_{1s})]$, donde $z_j...z_1$ es la representación binaria de z, M es una matriz de Ψ tal que sus filas son de la forma $(m_{i1}...m_{is})$, $\oplus_s$ es la operación o-exclusivo sobre operandos de s bits y $z_i \otimes_s (m_{i1}...m_{is})$ es igual a s ceros si $z_i = 0$ o a $(m_{i1}...m_{is})$ si $z_i = 1$. Entonces se cumple que la clase $H_3 = \{g_M / M \in \Psi\}$ es 2-universal.
- H_2: variante de H_3 que permite ganar tiempo a costa de espacio, consiste en interpretar el número $z = z_j...z_1$ como un número en base α que se puede expandir a otro número binario $z' = z_j \alpha z_{(j\alpha)-1}...z_1$, en el que habrá exactamente $\lceil r / \alpha \rceil$ unos; si esta magnitud no es muy grande, el cálculo exige menos o-exclusivos.

4.3.3 Funciones de traducción de cadenas a enteros en un intervalo

La estrategia más sencilla consiste en combinar dos métodos de las familias que acabamos de estudiar. El más usual consiste en aplicar la función de suma ponderada a la cadena de caracteres que forma la clave y operar el resultado con una familia universal o un método particular, generalmente el módulo; en este último caso, se puede alternar el módulo con la suma ponderada de manera que se evite el problema del desbordamiento.

También se pueden diseñar funciones que manipulen directamente la representación binaria de la clave. Si la clave es pequeña, puede interpretarse su representación como un número binario, si no, es necesario combinar sus bytes con ciertas operaciones (usualmente, sumas y o-exclusivos) de manera parecida al método de desplegamiento visto en el punto anterior. Presentamos aquí uno de estos métodos propuesto por P.K. Pearson en "Fast Hashing of Variable-Length Text Strings", Communications ACM, 33(6), 1990, que tiene un funcionamiento parecido a la familia H_3 de funciones universales.

Sea $r = 2^s$ la mínima potencia de 2 más grande que la base b de los caracteres, y sea un vector T [de 0 a r -1] que contenga una permutación aleatoria (sin repeticiones) de los elementos de Z_r (sus elementos, pues, se representan con s bits); la función de dispersión $h: C^* \rightarrow Z_r$ aplicada sobre una clave $k = c_n...c_1$ se define como:

$$A_0(k) = 0$$
$$A_i(k) = T[A_{i-1}(k) \oplus_s \phi(c_i)]$$
$$h(k) = A_n(k)$$

siendo $\oplus_s$ la operación o-exclusivo sobre operandos de s bits.

Este esquema se adapta perfectamente al caso habitual de claves de longitud variable o desconocida a priori; además, no impone ninguna restricción sobre su longitud y no necesita preocuparse de posibles desbordamientos. Ahora bien, la definición dada presenta un grave inconveniente: la dimensión de la tabla viene determinada por s, lo que normalmente resulta en un valor muy pequeño. Si suponemos el caso inverso, en el cual se diseña la función de dispersión a partir de r, es necesario reformular la definición. Sea t el entero más pequeño tal que $r - 1 < 2^t$, siendo t el número de bits necesario para representar valores dentro de Z_r; entonces, se aplica la función A_n un total de $p = \lceil t/s \rceil$ veces sobre k de la siguiente forma:

- Se define la familia de funciones $A_n^i(k) = A_n(k^i)$, donde $k^i = c_n...c_1{}^i$, siendo c^i el carácter tal que $\phi(c^i) = (\phi(c)+i)$ mod r.
- Se define la función de dispersión como $h(k) = A_n{}^{p-1}(k) \cdot ... \cdot A_n{}^0(k)$, siendo · la concatenación de bits. Notemos que, efectivamente, h(k) da valores en Z_r.

Se puede demostrar que las p secuencias de valores de $A_n^i(k)$ son independientes entre sí. Si r no es potencia de 2, la última vez que se aplique la función es necesario obtener menos de s bits, por ejemplo, tomando el resto de la división t / s.

4.3.4 Caracterización e implementación de las funciones de dispersión

En la siguiente sección se presentan varias implementaciones de tablas de dispersión. Todas ellas serán independientes de la función de dispersión elegida, que se convertirá, pues, en un parámetro formal de la implementación. Por ello, es necesario encapsular la definición de las funciones de dispersión en un universo de caracterización.

a) Caracterización de las funciones $f: C^* \rightarrow Z$

En el universo de caracterización ELEM_DISP_CONV se define el género elem que representa las claves tal como se ha requerido en la especificación. Además, para poder aplicarles una función de dispersión, estas claves se consideran como una tira de elementos individuales de tipo ítem (que generalmente serán caracteres, como ya se ha dicho), y presentan dos operaciones, una para averiguar el número de ítems de una clave y otra para obtener el ítem i-ésimo. Por último, es necesario definir también como parámetros formales la

cardinalidad b del dominio de los ítems y la función ϕ de conversión de un ítem a un natural dentro del intervalo apropiado, que denotamos respectivamente por base y conv. El universo resultante contiene, pues, los parámetros formales necesarios para definir las funciones de dispersión f.

```
universo ELEM_DISP_CONV caracteriza
    usa ELEM_=, NAT, BOOL
    tipo ítem
    ops  i_ésimo: elem nat → ítem
         nítems: elem → nat
         conv: ítem → nat
         base: → nat
    errores ∀v∈ elem; ∀i∈ nat: [NAT.ig(i, 0) ∨ (i > nítems(v))] ⇒ i_ésimo(v, i)
    ecns ∀v∈ ítem: (conv(v) > 0) ∧ (conv(v) ≤ base) = cierto
funiverso
```

Fig. 4.4: caracterización de las claves de las funciones de dispersión f.

A continuación, se puede enriquecer este universo formando uno nuevo, FUNCIONES_F, que añade la caracterización de las funciones f : $C^* \rightarrow Z$. El resultado aparecerá en la cabecera de todos aquellos universos parametrizados por la función de dispersión.

```
universo FUNCIONES_F caracteriza
    usa ELEM_DISP_CONV, NAT
    ops f: elem → nat
funiverso
```

Fig. 4.5: enriquecimiento de ELEM_DISP_CONV añadiendo la función f.

Los diferentes métodos particulares se pueden definir dentro de universos de implementación convenientemente parametrizados por ELEM_DISP_CONV ; los algoritmos resultantes son parámetros reales potenciales en aquellos contextos donde se requiera una función f (es decir, en universos parametrizados por FUNCIONES_F). Por ejemplo, en la fig. 4.6 se muestra el método de la suma ponderada aplicando la regla de Horner[5]; se supone que mult_desb(x, y, z) efectua la operación x*y + z sin que se llegue a producir realmente desbordamiento en ninguna operación.

[5] Opcionalmente, y para reducir el número de universos de las bibliotecas de módulos al mínimo, se podrían encapsular todos los métodos de dispersión en un único universo; la estructuración aquí presentada se adapta fácilmente a este caso.

```
universo SUMA_POND (ELEM_DISP_CONV) es[6]
    usa NAT
    función suma_pond (v es elem) devuelve nat es
    var acum, i son nat fvar
        acum := 0
        para todo i desde nítems(v) bajando hasta 1 hacer
            acum := mult_desb(acum, base, conv(i_ésimo(v, i)))
        fpara todo
    devuelve acum
funiverso
```

Fig. 4.6: implementación del método de la suma ponderada.

b) Caracterización de las funciones g: $Z \to Z_r$

En el universo de caracterización FUNCIONES_G se definen el método de dispersión y el valor r (v. fig. 4.7). A continuación, se pueden implementar los diversos algoritmos de forma similar a las funciones f, parametrizados tan solo por el valor r (natural caracterizado en VAL_NAT); por ejemplo, en la fig. 4.8 se implementa el método del módulo.

```
universo FUNCIONES_G caracteriza
    usa NAT, BOOL
    ops r: → nat
        g: nat → nat
    ecns ∀z∈ nat: g(z) < r = cierto
funiverso
```

Fig. 4.7: caracterización de las claves de las funciones de dispersión g.

```
universo DIVISIÓN (VAL_NAT) es
    usa NAT
    función división (z es nat) devuelve nat es
    devuelve z mod val
funiverso
```

Fig. 4.8: implementación del método de la división.

[6] En esta situación no es útil definir la especificación inicial de la función, porque de hecho se estaría dando su algoritmo, y por ello se implementa la función directamente. En todo caso, se podría considerar la posibilidad de especificarla en otro marco semántico.

c) Caracterización de las funciones h: $C^* \to Z_r$

De manera similar, se define un universo de caracterización parametrizado tanto por el tipo de las claves como por el valor r; este universo caracteriza todos los símbolos que el usuario de una tabla de dispersión ha de determinar, y por ello es el universo que finalmente aparecerá en la cabecera de las diversas implementaciones de las tablas.

```
universo CLAVE_DISPERSIÓN caracteriza
    usa ELEM_=, VAL_NAT, NAT, BOOL
    ops h: elem → nat
    ecns ∀v∈ elem: h(v) < val = cierto
funiverso
```

Fig. 4.9: caracterización de las claves en el contexto de las funciones de dispersión h.

En este punto ya es posible definir funciones de dispersión. La primera opción consiste en componer dos funciones f y g, tal como se muestra en la fig. 4.10. Por ejemplo, en la fig. 4.11 se da la composición de la suma ponderada y la división; las dos primeras instancias, privadas, construyen las funciones f y g en el contexto requerido por el tipo de las claves, el valor del módulo, etc. y, a continuación, se efectúa una tercera instancia que define la función h simplemente como composición de las anteriores, con un renombramiento final.

```
universo COMPOSICIÓN_F_Y_G (FUNCIONES_F, FUNCIONES_G) es
    función g_de_f (v es elem) devuelve nat es
    devuelve g(f(v))
funiverso
```

Fig. 4.10: composición de funciones de dispersión.

La segunda manera de definir funciones de dispersión consiste en trabajar directamente sobre la clave sin pasos intermedios. Por ejemplo, en la fig. 4.12 se muestra la función que alterna la suma ponderada y la división.

Cabe destacar que los universos resultantes de estos pasos deben crearse una única vez y que, a partir de ese momento, el usuario los tendrá disponibles en una biblioteca de funciones de dispersión, y se limitará simplemente a efectuar las instancias adecuadas en aquellos contextos que lo exijan (sin necesidad de conocer la estructuración interna en universos). Estas instancias son muy simples. Por ejemplo, supongamos que se quiere implementar una tabla donde las claves son cadenas de letras mayúsculas, los valores son

enteros y la dimensión de la tabla es 1017 y que, como función de dispersión, se componen la suma ponderada y la división; el resultado podría ser la instancia:

```
instancia SUMA_POND_Y_DIV(ELEM_DISP_CONV, VAL_NAT) donde
    elem es cadena, ítem es carácter
    = es CADENA.=, nítems es CADENA.long, i_ésimo es CADENA.i_ésimo
    conv es conv_mayúsc, base es 26, val es 1017
```

donde conv_mayúsc debería estar implementada en la biblioteca de funciones de conversión, y CADENA presenta las operaciones definidas en el apartado 1.5.1.

```
universo SUMA_POND_Y_DIV(E es ELEM_DISP_CONV, V1 es VAL_NAT) es
    instancia privada SUMA_POND(F es ELEM_DISP_CONV) donde
        F.elem es E.elem, F.ítem es E.ítem
        F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
        F.conv es E.conv, F.base es E.base
    instancia privada DIVISIÓN(V2 es VAL_NAT) donde V2.val es V1.val
    instancia COMPOSICIÓN_F_Y_G(F es FUNCIONES_F, G es FUNCIONES_G) donde
        F.elem es E.elem, F.ítem es E.ítem
        F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
        F.conv es E.conv, F.base es E.base, F.f es suma_pond
        G.r es V1.val, G.g es división
    renombra f_de_g por suma_pond_y_div
funiverso
```

Fig. 4.11: composición de la suma ponderada y la división.

```
universo SUMA_POND_Y_DIV_SIMULT(ELEM_DISP_CONV, VAL_NAT) es
    usa NAT
    función suma_pond_y_mod (v es elem) devuelve nat es
    var acum, i son nat fvar
        acum := 0
        para todo i desde nítems(v) bajando hasta 1 hacer
            acum := mult_desb(acum, base, conv(i_ésimo(v, i))) mod val
        fpara todo
    devuelve acum
funiverso
```

Fig. 4.12: aplicación simultánea de la suma ponderada y la división.

4.4 Organizaciones de las tablas de dispersión

Las tablas de dispersión pueden organizarse de varias maneras según el modo en que se gestionen las colisiones. Hay dos grandes familias de tablas dependiendo de si se encadenan o no los sinónimos; además, existen diversas organizaciones que son combinaciones de estas dos, de las que se presentará una. Algunos autores también clasifican las tablas como abiertas o cerradas, según exista una zona diferenciada para todas o parte de las claves, o se almacenen en un único vector directamente indexado por la función de dispersión. A veces, los nombres que dan diversos autores pueden causar confusión; por ejemplo, algunos textos denominan closed hashing a una organización que otros llaman open addressing.

A lo largo de la sección, definiremos las diferentes organizaciones como universos parametrizados por los géneros de las claves y los valores, la igualdad de las claves, el valor indefinido, la función de dispersión y el número de valores de dispersión diferentes que hay.

4.4.1 Tablas encadenadas

En las tablas de dispersión encadenadas (ing., chaining) se forman estructuras lineales con los sinónimos; estudiaremos dos esquemas:

- Tablas encadenadas indirectas: se encadenan los sinónimos de un mismo valor de dispersión, y el primer elemento de la lista es accesible a partir de un vector índice.
- Tablas encadenadas directas: se guarda la primera clave de cada valor de dispersión, con su valor asociado, en una zona diferenciada, y los respectivos sinónimos se encadenan en otra zona, y son accesibles a partir del que reside en la zona diferenciada.

a) Tablas encadenadas indirectas

También conocidas con el nombre de tablas encadenadas abiertas (ing., separate chaining), asocian la lista de los sinónimos de un valor de dispersión i a la posición i de un vector índice. En la fig. 4.13 se muestra el esquema de esta representación. Queda claro que si la función distribuye correctamente, las listas de sinónimos serán de longitud similar, concretamente $\lceil n/r \rceil$, donde n es el número de elementos en la tabla. En consecuencia, tenemos que $\varepsilon[S_n(k)] = \lceil (n+r)/2r \rceil$ y $\varepsilon[S_n(k)] = \lceil n/r \rceil$, de manera que las operaciones quedan $\Theta(n/r)$. Si aproximadamente hay tantos elementos como valores de dispersión, este cociente será muy pequeño (es decir, las listas de sinónimos serán muy cortas) y las funciones del TAD se podrán considerar $\Theta(1)$, dado que la inserción y la supresión consisten simplemente en buscar el elemento y modificar un par de encadenamientos.

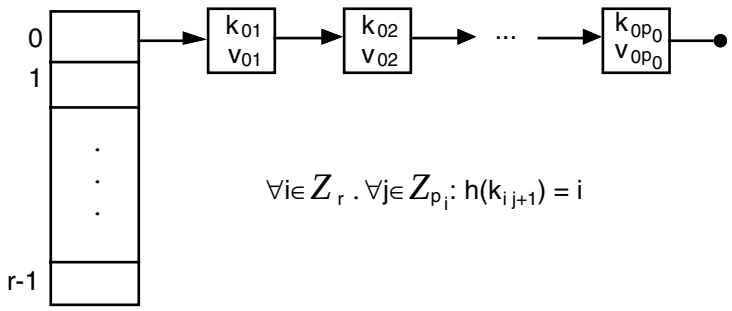


Fig. 4.13: organización de una tabla de dispersión encadenada indirecta.

En la fig. 4.14 se presenta una posible implementación para el tipo tabla representando las listas en memoria dinámica (otra alternativa sería guardar los elementos dentro de un único vector compartido por todas las listas). Observemos que el invariante establece que los elementos de la lista que cuelga de la posición i del vector índice son sinónimos del valor de dispersión i, usando la conocida función que devuelve la cadena que cuelga de un elemento inicial (v. fig. 3.23); cualquier otra propiedad de la representación se deduce de ésta. Los universos de caracterización definen los parámetros necesarios para aplicar las funciones de dispersión tal como se ha explicado en la sección anterior; notemos que, en particular, todos los parámetros que aparecían en la especificación también están en la implementación, aunque el encapsulamiento en universos de caracterización haya cambiado. Tanto en esta implementación como en posteriores, no se incluye código para controlar el buen comportamiento de la función de dispersión, quedando como ejercicio para el lector.

Los algoritmos son claros: se accede por la función de dispersión al índice y se recorre la lista de sinónimos; las inserciones se efectúan siempre por el inicio (es lo más sencillo) y en las supresiones es necesario controlar si el elemento borrado es el primero, para actualizar el índice (una alternativa sería utilizar elementos fantasmas, pero a costa de aumentar considerablemente el espacio empleado, siendo necesario un estudio cuidadoso sobre su conveniencia). Se usa una función auxiliar, busca, que busca una clave en la tabla. Podríamos considerar la posibilidad de ordenar los sinónimos por clave o de disponer de listas autoorganizativas; estas dos variantes reducen el coste de algunas búsquedas a costa de las inserciones. Otra variante consiste en no buscar las claves cuando se insertan sino añadirlas directamente al inicio de la lista, de manera que la consulta siempre encontrará primero la última ocurrencia de la clave; obviamente, si las claves se redefinen con frecuencia puede llegar a desaprovecharse mucha memoria.

Una alternativa a la implementación del tipo consiste en usar una instancia de alguna implementación de las listas escrita en algún universo ya existente. En la sección 6.2.2 se introduce una representación del TAD de los grafos usando listas con punto de interés que sigue esta estrategia, de manera que el lector puede comparar los resultados.

```
universo TABLA_IND_PUNTEROS(CLAVE_DISPERSIÓN, ELEM_ESP) es
    implementa TABLA(ELEM_=, ELEM_ESP)
    usa ENTERO, BOOL
    renombra CLAVE_DISPERSIÓN.elem por clave, CLAVE_DISPERSIÓN.val por r
             ELEM_ESP.elem por valor, ELEM_ESP.esp por indef
    tipo tabla es vector [de 0 a r-1] de ^nodo ftipo
    tipo privado nodo es
        tupla
            k es clave; v es valor; enc es ^nodo
        ftupla
    ftipo
    invariante (T es tabla): ∀i: 0 ≤ i ≤r-1:
            NULO∈ cadena(T[i]) ∧ (∀p: p∈ cadena(T[i])-{NULO}: h(p^.k) = i)
    función crea devuelve tabla es
    var i es nat; t es tabla fvar
        para todo i desde 0 hasta r-1 hacer t[i] := NULO fpara todo
    devuelve t
    función asigna (t es tabla; k es clave; v es valor) devuelve tabla es
    var i es nat; encontrado es booleano; ant, p son ^nodo fvar
        i := h(k); <encontrado, ant, p> := busca(t[i], k)
        si encontrado entonces p^.v := v     {cambiamos la información asociada}
        si no {es necesario crear un nuevo nodo}
            p := obtener_espacio
            si p = NULO entonces error     {no hay espacio}
            si no p^ := <k, v, t[i]>; t[i] := p     {inserción por el inicio}
            fsi
        fsi
    devuelve t
    función borra (t es tabla; k es clave) devuelve tabla es
    var i es nat; encontrado es booleano; ant, p son ^nodo fvar
        i := h(k); <encontrado, ant, p> := busca(t[i], k)
        si encontrado entonces {es necesario distinguir si es el primero o no}
            si ant = NULO entonces t[i] := p^.enc si no ant^.enc := p^.enc fsi
            liberar_espacio(p)
        fsi
    devuelve t
```

Fig. 4.14: implementación de las tablas de dispersión encadenadas indirectas.

```
función consulta (t es tabla; k es clave) devuelve valor es
var encontrado es booleano; res es valor; ant, p son ^nodo fvar
    <encontrado, ant, p> := busca(t[h(k)], k)
    si encontrado entonces res := p^.v si no res := indef fsi
devuelve res

{Función busca(q, k) → <encontrado, ant, p>: busca el elemento de clave k a
 partir de q. Devuelve un booleano indicando el resultado de la búsqueda y, en
 caso de éxito, un apuntador p a la posición que ocupa y otro ant a su
 predecesor, que valdrá NULO si p es el primero
      P ≅ cierto
      Q ≅ encontrado ≡ ∃r: r∈cadena(q)-{NULO}: r^.k = k ∧
            encontrado ⇒ p^.k = k ∧ (p = q ⇒ ant = NULO ∧ p ≠ q ⇒ ant^.enc = p) }

función privada busca (q es ^nodo; k es clave) ret <bool, ^nodo, ^nodo> es
var encontrado es booleano; res es valor; ant, p es ^nodo fvar
    ant := NULO; p := q; encontrado := falso
    mientras (p ≠ NULO) ∧ ¬encontrado hacer
        si p^.k = k entonces encontrado := cierto si no ant := p; p := p^.enc fsi
    fmientras
devuelve <encontrado, ant, p>
funiverso
```

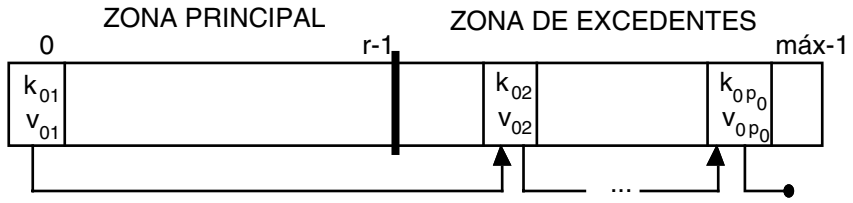
Fig. 4.14: implementación de las tablas de dispersión encadenadas indirectas (cont.).

Por último, es necesario destacar un problema que surge en todas las organizaciones de dispersión: la progresiva degeneración que sufren a medida que crece su tasa de ocupación. Para evitarla se puede incorporar a la representación un contador de elementos en la tabla, de forma que el cálculo de la tasa de ocupación sea inmediato; al insertar nuevos elementos es necesario comprobar previamente que la tasa no sobrepase una cota máxima determinada (valor que sería un parámetro más, tanto de la especificación como de la implementación), en cuyo caso se da un error conocido con el nombre de desbordamiento (ing., overflow; evidentmente, la especificación de las tablas debería modificarse para reflejar este comportamiento). La solución más simple posible del desbordamiento (y también la más ineficiente) consiste en redimensionar la tabla, lo cual obliga a definir una nueva función de dispersión y, a continuación reinsertar todos los identificadores presentes en la tabla usando esta nueva función. Alternativamente, hay métodos que aumentan selectivamente el tamaño de la tabla, de manera que sólo es necesario reinsertar un subconjunto de los elementos; estas estrategias, denominadas incrementales, difieren en la frecuencia de aplicación y en la parte de la tabla tratada. Entre ellas destacan la dispersión extensible y la dispersión lineal; ambos métodos son especialmente útiles para tablas implementadas en disco y no se estudian aquí. En el apartado 4.4.6 se presentan unas fórmulas que determinan

exactamente como afecta la tasa de ocupación a la eficiencia de las operaciones.

b) Tablas encadenadas directas

También conocidas con el nombre de tablas con zona de excedentes (ing., hashing with cellar), precisamente por la distinción de la tabla en dos zonas: la zona principal de r posiciones, donde la posición i, o bien está vacía, o bien contiene un par <k, v> que cumple h(k) = i, y la zona de excedentes, que ocupa el resto de la tabla y guarda las claves sinónimas. Cuando al insertar un par se produce una colisión, el nuevo sinónimo se almacena en esta zona; todos los sinónimos de un mismo valor están encadenados y el primer sinónimo residente en la zona de excedentes es accesible a partir del que reside en la zona principal. Alternativamente, la zona de excedentes se puede implementar por punteros sin que el esquema general varíe sustancialmente. Las consideraciones sobre la longitud de las listas y el coste de las operaciones son idénticas al caso de tablas encadenadas indirectas; sólo es necesario notar el ahorro de un acceso a causa de la inexistencia del vector índice, que evidentmente no afecta al coste asintótico de las operaciones[7].



$$\forall i \in Z_r \,.\, \forall j \in Z_{p_i} : h(k_{i\,j+1}) = i$$

Fig. 4.15: organización de una tabla de dispersión encadenada directa.

En la fig. 4.16 se ofrece una implementación del tipo; en la cabecera aparece un universo de caracterización adicional, VAL_NAT, para determinar la dimensión de la zona de excedentes (diversos estudios empíricos apuntan a que esta zona ha de representar aproximadamente el 14% de la tabla). Vuelve a haber una función auxiliar para buscar una clave. Observamos que la codificación de las funciones es un poco más complicada que en el caso indirecto, porque es necesario distinguir si una posición está libre o no y también el acceso a la zona principal del acceso a la zona de excedentes. Para solucionar el primer punto, se establece una convención para saber si las posiciones de la zona principal están o no vacías, y en caso de que estén vacías, se marcan (es decir, se da un valor especial a algún campo; en la fig. 4.16, se asigna el valor -2 al campo de encadenamiento, y queda el -1 para el último elemento de cada lista). Las posiciones libres de la zona de excedentes se gestionan en forma de pila.

[7] Pero que es importante al considerar la implementación sobre ficheros; de hecho, las tablas con zona de excedentes son útiles sobre todo en memoria secundaria, donde en cada cubeta se colocan tantos elementos como se puedan leer del / escribir al disco en una única operación de entrada/salida.

La supresión de los elementos residentes en la zona de excedentes requiere tan sólo modificar encadenamientos e insertar la posición liberada en la pila de sitios libres; ahora bien, al borrar un elemento residente en la zona principal no se puede simplemente marcar como libre la posición correspondiente, porque el algoritmo de búsqueda no encontraría los hipotéticos sinónimos residentes en la zona de excedentes. Así, se podría introducir otra marca para distinguir las posiciones borradas, de manera que las posiciones marcadas como borradas se interpreten como ocupadas al buscar y como libres al insertar una nueva clave. El mantenimiento de posiciones borradas puede provocar que en la zona principal existan demasiados "agujeros", que la tabla degenere (es decir, que empeore el tiempo de acceso) y que, eventualmente, se llene la zona de excedentes, quedando posiciones borradas en la zona principal y provocando error en la siguiente colisión. Una alternativa consiste en mover un sinónimo de la zona de excedentes a la zona principal, lo que es costoso si el tamaño de los elementos es grande o si los elementos de la tabla son apuntados por otras estructuras de datos; en la fig. 4.16 se implementa esta última opción y la primera queda como ejercicio. Notemos que el invariante es ciertamente exhaustivo y comprueba los valores de los encadenamientos, la ocupación de las diferentes posiciones de la tabla y que todos elementos que cuelgan de la misma cadena sean sinónimos del valor correspondiente de dispersión; se usa la función cadena de la fig. 3.20 aplicada sobre tablas.

```
universo TABLA_DIRECTA(CLAVE_DISPERSIÓN, ELEM_ESP, VAL_NAT) es
    implementa TABLA(ELEM_=, ELEM_ESP)
    usa ENTERO, BOOL
    renombra CLAVE_DISPERSIÓN.elem por clave, CLAVE_DISPERSIÓN.val por r
             ELEM_ESP.elem por valor, ELEM_ESP.esp por indef
    const máx vale r+VAL_NAT.val
    tipo tabla es tupla A es vector [de 0 a máx-1] de
                                        tupla k es clave; v es valor; enc es entero ftupla
                    sl es entero
                ftupla
    ftipo
    invariante (T es tabla): (T.sl = -1) ∨ (r ≤ T.sl < máx)
            ∧ ∀i: 0 ≤ i ≤ r-1: (T.A[i].enc = -1) ∨ (T.A[i].enc = -2) ∨ (r ≤ T.A[i].enc < máx)
            ∧ ∀i: r ≤ i ≤ máx-1: (T.A[i].enc = -1) ∨ (r ≤ T.A[i].enc < máx)
            ∧ ∀i: 0 ≤ i ≤ r-1 ∧ T.A[i].enc ≠ -2:   {es decir, para toda posición no vacía}
                    ∀p: p∈ (cadena(T.A, i) - {-1}): h(T.A[p].k) = i ∧
                    ∀j: 0 ≤ j ≤ r-1 ∧ j≠i ∧ A[j].enc ≠ -2: cadena(T.A, i) ∩ cadena(T.A, j) = {-1} ∧
                    cadena(T.A, i) ∩ cadena(T.A, T.sl) = {-1}
            ∧ (∪i: 0 ≤ i ≤ r-1 ∧ T.A[i].enc ≠ -2: cadena(T.A, T.A[i].enc) )
                                        ∪ cadena(T.A, T.sl) = [r, máx-1] ∪ {-1}
```

Fig. 4.16: implementación de las tablas de dispersión encadenadas directas.

```
función crea devuelve tabla es
var i es nat; t es tabla fvar
        {zona principal vacía}
    para todo i desde 0 hasta r-1 hacer t.A[i].enc := -2 fpara todo
        {pila de sitios libres}
    para todo i desde r hasta máx-2 hacer t.A[i].enc := i+1 fpara todo
    t.A[máx-1].enc := -1; t.sl := r
devuelve t

función asigna (t es tabla; k es clave; v es valor) devuelve tabla es
var ant, i, p son enteros; encontrado es booleano fvar
    i := h(k)
    si t.A[i].enc = -2 entonces t.A[i] := <k, v, -1>  {posición libre en la zona principal}
    si no  {hay sinónimos}
        <encontrado, ant, p> := busca(t, k, i)
        si encontrado entonces t.A[p].v := v   {cambio de la información asociada}
        si no  si t.sl = -1 entonces error  {no hay espacio}
                si no {se deposita en la zona de excedentes}
                    p := t.sl; t.sl := t.A[t.sl].enc
                    t.A[p] := <k, v, t.A[i].enc>; t.A[i].enc := p
                fsi
        fsi
    fsi
devuelve t

función borra (t es tabla; k es clave) devuelve tabla es
var ant, p, nuevo_sl son enteros; encontrado es booleano fvar
    <encontrado, ant, p> := busca(t, k, h(k))
    si encontrado entonces
        si ant = -1 entonces {reside en la zona principal}
            si t.A[p].enc = -1 entonces t.A[p].enc := -2  {no tiene sinónimos}
            si no {se mueve el primer sinónimo a la zona principal}
                nuevo_sl := t.A[p].enc; t.A[p] := t.A[t.A[p].enc]
                t.A[nuevo_sl].enc := t.sl; t.sl := nuevo_sl
            fsi
        si no {reside en la zona de excedentes}
            t.A[ant].enc := t.A[p].enc; t.A[p].enc := t.sl; t.sl := p
        fsi
    fsi
devuelve t
```

Fig. 4.16: implementación de las tablas de dispersión encadenadas directas (cont.).

```
    función consulta (t es tabla; k es clave) devuelve valor es
    var encontrado es booleano; res es valor; ant, p son entero fvar
        <encontrado, ant, p> := busca(t, k, h(k))
        si encontrado entonces res := t.A[p].v si no res := indef fsi
    devuelve res

    {Función auxiliar busca(t, k, i): v. fig. 4.14}
    función privada busca (t es tabla; k es clave; i es entero) dev <bool, entero, entero> es
    var encontrado es booleano; ant es entero fvar
        encontrado := falso
        si t.A[i].enc ≠ -2 entonces  {si no, no hay claves con valor de dispersión i }
            ant := -1
            mientras (i ≠ -1) ∧ ¬encontrado hacer
                si t.A[i].k = k entonces encontrado := cierto si no ant := i; i := t.A[i].enc fsi
            fmientras
        fsi
    devuelve <encontrado, ant, i>
funiverso
```

Fig. 4.16: implementación de las tablas de dispersión encadenadas directas (cont.).

4.4.2 Tablas de direccionamiento abierto

En las tablas de direccionamiento abierto (ing., open addresing) se dispone de un vector con tantas posiciones como valores de dispersión; dentro del vector, no existe una zona diferenciada para las colisiones ni se encadenan los sinónimos, sino que para cada clave se define una secuencia de posiciones que determina el lugar donde irá. Concretamente, al insertar el par <k, v> en la tabla, tal que $h(k) = p_0$, se sigue una secuencia p_0, ..., p_{r-1} asociada a k hasta que:

- Se encuentra una posición p_s tal que su clave es k: se sustituye su información asociada por v.
- Se encuentra una posición p_s que está libre: se coloca el par <k, v>. Si $s \neq 0$, a la clave se la llama invasora (ing., invader), por razones obvias.
- Se explora la secuencia sin encontrar ninguna posición que cumpla alguna de las dos condiciones anteriores: significa que la tabla está llena y no se puede insertar el par.

El esquema al borrar y consultar es parecido: se siguen las p_0, …, p_{r-1} hasta que se encuentra la clave buscada o una posición libre. El punto clave de esta estrategia es que la secuencia p_0, …, p_{r-1} asociada a una clave k es fija durante toda la existencia de la tabla de manera que, cuando se añade, se borra o se consulta una clave determinada siempre se examinan las

mismas posiciones del vector en el mismo orden; idealmente, claves diferentes tendrán asociadas secuencias diferentes. Este proceso de generación de valores se denomina redispersión (ing., rehashing), la secuencia generada por una clave se denomina camino (ing., path o probe sequence) de la clave, y cada acceso a la tabla mediante un valor del camino se denomina ensayo (ing., probe). En otras palabras, el concepto de redispersión consiste en considerar que, en vez de una única función h de dispersión, disponemos de una familia $\{h_i\}$ de funciones, denominadas funciones de redispersión (ing., rehashing function), tales que, si la aplicación de $h_i(k)$ no lleva al resultado esperado, se aplica $h_{i+1}(k)$; en este esquema, la función h de dispersión se define como h_0.

Una vez más es necesario estudiar cuidadosamente el problema de la supresión de elementos. La situación es parecida a la organización encadenada directa: cuando una clave k colisiona y se deposita finalmente en la posición determinada por un valor $h_j(k)$ se debe a que todas las posiciones determinadas por los valores $h_i(k)$, para todo $i < j$, están ocupadas; al suprimir una clave que ocupa la posición $h_s(k)$, para algún $s < j$, no se puede simplemente marcar como libre esta posición, porque una búsqueda posterior de k dentro la tabla siguiendo la estrategia antes expuesta fracasaría. En consecuencia, además de las posiciones "ocupadas" o "libres", se distingue un tercer estado que identifica las posiciones "borradas", las cuales se tratan como libres al buscar sitio para insertar un nuevo par y como ocupadas al buscar una clave; al igual que en el esquema encadenado directo ya estudiado, las posiciones borradas provocan la degeneración de la tabla.

En la fig. 4.17 se muestra una implementación de las tablas de direccionamiento abierto sin determinar la familia de funciones de dispersión, que queda como parámetro formal de la implementación sustituyendo a la tradicional función de dispersión; los universos de caracterización de los parámetros formales aparecen en la fig. 4.18. En la definición de los valores se añade una constante como parámetro formal, que permite implementar el concepto de posición borrada sin emplear ningún campo adicional (las posiciones libres contendrán el valor indefinido, que ya es un parámetro formal). En caso de no poderse identificar este nuevo valor especial, se podría identificar una clave especial y, si tampoco fuera posible, sería imprescindible entonces un campo booleano en las posiciones del vector que codificara el estado. Por lo que respecta al invariante, se usa una función auxiliar predecesores que, para una clave k residente en la posición p_i de su camino, devuelve el conjunto de posiciones $\{p_0, ..., p_{i-1}\}$, las cuales, dada la estrategia de redispersión, no pueden estar libres. Por último, la función auxiliar busca da la posición donde se encuentra una clave determinada o, en caso de no encontrarse, la primera posición libre o borrada donde se insertaría, y da prioridad a las segundas para reaprovecharlas; el valor booleano devuelto indica si la clave se ha encontrado o no. Como siempre, se podría incorporar un contador para controlar el factor de ocupación de la tabla, considerando que una posición borrada vale lo mismo que una posición ocupada.

```
universo TABLA_ABIERTA(CLAVE_REDISPERSIÓN, ELEM_2_ESP_=)
    implementa TABLA(ELEM_=, ELEM_ESP)
    renombra
        CLAVE_REDISPERSIÓN.elem por clave, CLAVE_REDISPERSIÓN.val por r
        ELEM_2_ESP_=.elem por valor
        ELEM_2_ESP_=.esp1 por indef, ELEM_2_ESP_=.esp2 por borrada
    usa ENTERO, BOOL
    tipo tabla es vector [de 0 a r-1] de tupla k es clave; v es valor ftupla ftipo
    invariante (T es tabla):
        ∀i: 0 ≤ i ≤ r-1 ∧ T[i].v ≠ indef ∧ T[i].v ≠ borrada:
            ∀j: j∈predecesores(T, T[i].k, 0): T[j].v ≠ indef
        donde se define predecesores: tabla clave nat → P(nat) como:
            T[h(i, k)].k = k ⇒ predecesores(T, k, i) = Ø
            T[h(i, k)].k ≠ k ⇒ predecesores(T, k, i) = {h(i, k)} ∪ predecesores(T, k, i+1)

    función crea devuelve tabla es
    var i es nat; t es tabla fvar
        para todo i desde 0 hasta r-1 hacer t[i].v := indef fpara todo
    devuelve t

    función asigna (t es tabla; k es clave; v es valor) devuelve tabla es
    var p es entero; encontrado es booleano fvar
        <p, encontrado> := busca(t, k)
        si encontrado entonces t[p].v := v   {se cambia la información asociada}
        si no {se inserta en la posición que le corresponde}
            si p = -1 entonces error {no hay espacio} si no t[p] := <k, v> fsi
        fsi
    devuelve t

    función borra (t es tabla; k es clave) devuelve tabla es
    var p es nat; encontrado es booleano fvar
        <p, encontrado> := busca(t, k)
        si encontrado entonces t[p].v := borrada fsi
    devuelve t

    función consulta (t es tabla; k es clave) devuelve valor es
    var encontrado es booleano; res es valor; p es nat fvar
        <p, encontrado> := busca(t, k)
        si encontrado entonces res := t[p].v si no res := indef fsi
    devuelve res
```

Fig. 4.17: implementación de las tablas de dispersión abiertas.

```
{Función auxiliar busca(t, k) → <j, encontrado>: busca la clave k dentro de t ; si la
 encuentra, devuelve cierto y la posición que ocupa, si no, devuelve falso y la
 posición donde insertarla si es el caso (o -1, si no hay ninguna)
     P ≅ cierto
     Q ≅ encontrado ≡ ∃i: 0 ≤ i ≤ r-1: (t[i].k = k ∧ t[i].v ≠ indef ∧ t[i].v ≠ borrada)
         ∧ (encontrado ⇒ t[j].k = k)
         ∧ (¬encontrado ⇒ (t[j].v = indef ∨ t[j].v = borrada) ∧
                          t[j].v = indef ⇒ (¬ ∃s: s∈pos(T, k, j): T[s].v = borrada)
  donde pos devuelve las posiciones del camino asociado a k entre T[h(0, k)] y j }
función privada busca (t es tabla; k es clave) devuelve <nat, bool> es
var i, p, j son enteros; encontrado, final son booleano fvar
   i := 0; final := falso; j := -1; encontrado := falso
   mientras (i ≤ r) ∧ ¬ final hacer
      p := h(i, k) {se aplica la función i-ésima de la familia de redispersión}
      opción
         caso t[p].v = indef hacer {la clave no está}
            final := cierto; si j = -1 entonces j := p fsi
         caso t[p].v = borrada hacer i := i + 1; j := p {sigue la búsqueda}
         en cualquier otro caso {la posición contiene una clave definida}
            si t[p].k = k entonces encontrado := cierto; final := cierto si no i := i + 1 fsi
      fopción
   fmientras
devuelve <j, encontrado>
```

Fig. 4.17: implementación de las tablas de dispersión abiertas (cont.).

```
universo ELEM_2_ESP_= caracteriza
   usa BOOL
   tipo elem
   ops esp1, esp2: → elem
       _=_, _≠_: elem elem → bool
funiverso
```

```
universo CLAVE_REDISPERSIÓN caracteriza
   usa ELEM, VAL_NAT, NAT, BOOL
   ops h: nat elem → nat {h(i, k) ≅ h_i(k)}
   errores ∀k∈elem; ∀i∈nat: [i > val] ⇒ h(i, k)
   ecns ∀k∈elem; ∀i∈nat: h(i, k) < val = cierto
funiverso
```

Fig. 4.18: caracterización de los parámetros formales de las tablas de dispersión abiertas.

En la fig. 4.19 se presenta la evolución de una tabla de direccionamiento abierto siguiendo la estrategia de la fig. 4.17, suponiendo que las funciones de redispersión $\{h_i\}$ simplemente toman el último dígito de la cadena y le suman i (¡es un buen ejemplo de estrategia que nunca debe emplearse!); la disposición de los elementos dentro de la tabla depende de la historia y de los algoritmos concretos de inserción y de supresión, de manera que podríamos encontrar otros configuraciones igualmente válidas al cambiar cualquiera de los dos factores.

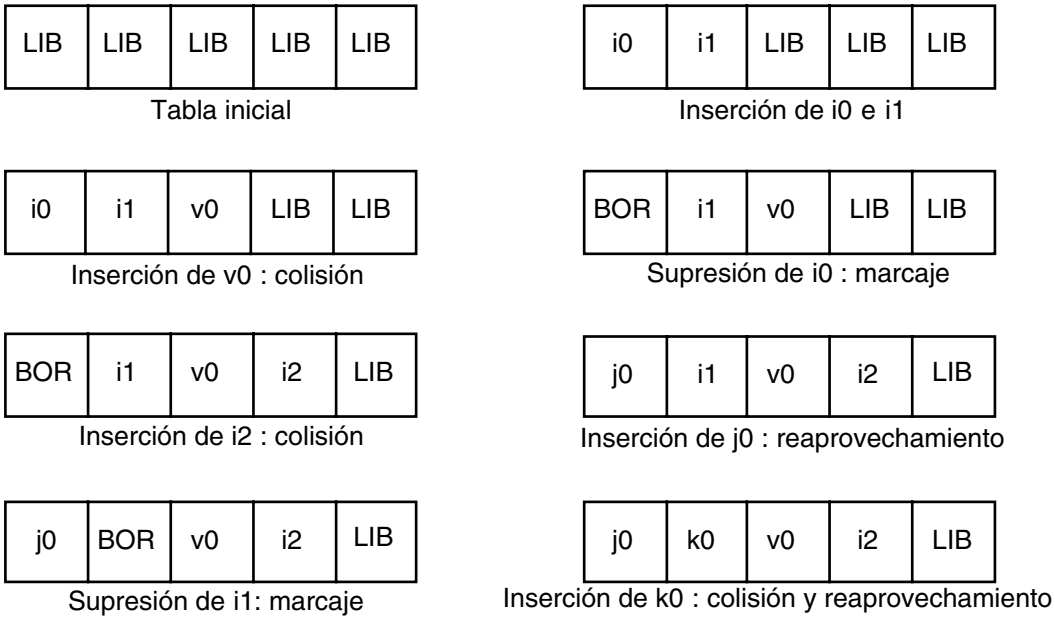


Fig. 4.19: evolución de una tabla de direccionamiento abierto
(LIB: posición libre; BOR: posición borrada).

Una vez formulada la implementación genérica de las tablas de dispersión de direccionamiento abierto, es necesario estudiar los diferentes métodos de redispersión existentes; todos ellos se definen a partir de una o dos funciones de redispersión primarias que denotaremos por h y h'. Sea cual sea la familia de funciones de redispersión elegida, hay tres propiedades que deberían cumplirse; la primera de ellas, sobre todo, es ineludible:

- Para una clave dada, su camino es siempre el mismo (las mismas posiciones en el mismo orden).
- Para una clave dada, su camino pasa por todas las posiciones de la tabla.
- Para dos claves diferentes k y k', si bien es prácticamente inevitable que a veces compartan parte de sus caminos, es necesario evitar que estos caminos sean idénticos a partir de un punto determinado.

Examinemos atentamente esta última propiedad. De entrada parece factible que se cumpla, porque en una tabla de dimensión r hay $r!$ caminos diferentes de longitud r; ahora bien, la mayoría de esquemas de redispersión que estudiamos a continuación usan bastante menos de $r!$ caminos y, por ello, presentan normalmente el fenómeno conocido como apiñamiento (ing., clustering), que consiste en la formación de largas cadenas de claves en la tabla (denominadas cadenas de reubicación) que la hacen degenerar rápidamente. Imaginemos el caso extremo en que $\forall k,k' \in K: h_i(k) = h_j(k') \Rightarrow h_{i+d}(k) = h_{j+d}(k')$, y sea una secuencia $p_1, \ldots, p_s$

de posiciones ocupadas por claves $k_1, \ldots, k_s$, tal que $\forall i: 2 \leq i \leq s: \exists j: 1 \leq j \leq i: h(k_i) = p_j$ y $h(k_1) = p_1$; en esta situación, la inserción de una nueva clave k, tal que $\exists i: 1 \leq i \leq s: h(k) = p_i$, implica que la posición destino de k es $h_{s+1}(k_1)$; es decir, la posición $h_{s+1}(k_1)$ es el destino de gran cantidad de claves, por lo que su posibilidad de ocupación es muy grande; además, el efecto es progresivo puesto que, cuando se ocupe, la primera posición $h_{s+1+x}(k_1)$ libre tendrá todavía más probabilidad de ocupación que la que tenía $h_{s+1}(k_1)$.

A continuación, enumeramos los principales métodos de redispersión.

a) Redispersión uniforme

Método de interés principalmente teórico, que asocia a cada clave un camino distinto, que es una permutación de los elementos de Z_r; los r ! caminos posibles son equiprobables. Evidentemente, es el mejor método posible; ahora bien, ¿cómo implementarlo?

b) Redispersión aleatoria

De manera parecida, si disponemos de una función capaz de generar aleatoriamente una secuencia de números a partir de una clave, tendremos una familia de funciones de redispersión casi tan buena como la uniforme aunque, a diferencia del esquema anterior, una misma posición puede generarse más de una vez en el camino de una clave.

De nuevo tenemos el problema de definir esta función aleatoria. Normalmente, hemos de conformarnos con un generador de números pseudoaleatorio. Otra opción consiste en construir una función ad hoc que manipule con contundencia la clave; por ejemplo, en [AHU83, pp.134-135] se muestra una simulación de la redispersión aleatoria mediante una manipulación de la clave con sumas, desplazamientos y o-exclusivos.

c) Redispersión lineal

Es el método de redispersión más intuitivo y fácil de implementar, se caracteriza porque la distancia entre $h_i(k)$ y $h_{i+1}(k)$ es constante para cualquier i.

$$h_i(k) = (h(k) + c \,.\, i) \bmod r$$

Si c y r son primos entre sí, se van ensayando sucesivamente todas las posiciones de la tabla: h(k), h(k)+c mod r, h(k)+2c mod r, …, y si hay alguna posición vacía finalmente se encuentra; por esto, normalmente se toma c = 1 (como en el ejemplo de la fig. 4.19).

Esta estrategia sencilla presenta, no obstante, el llamado apiñamiento primario: si hay dos claves k y k' tales que h(k) = h(k'), entonces la secuencia de posiciones generadas es la misma para k y para k'. Observamos que, cuanto más largas son las cadenas, más probable es que crezcan; además, la fusión de cadenas agrava el problema porque las hace crecer de golpe. El apiñamiento primario provoca una variancia alta del número esperado de ensayos al acceder a la tabla.

No obstante, la redispersión lineal presenta una propiedad interesante: la supresión no obliga a degenerar la tabla, sino que se pueden mover elementos para llenar espacios vacíos. Concretamente, al borrar el elemento que ocupa la posición p_i dentro de una cadena $p_1, ..., p_n, i < n$, se pueden mover los elementos k_j que ocupan las posiciones $p_j, i < j \leq n$, a posiciones p_s, $s < j$, siempre que se cumpla que $s \geq t$, siendo $p_t = h(k_j)$. Es decir, al borrar un elemento se examina su cadena desde la posición que ocupa hasta al final y se mueven hacia adelante tantos elementos como sea posible; en concreto, un elemento se mueve siempre que la posición destino no sea anterior a su valor de dispersión (v. fig. 4.20). Esta técnica no sólo evita la degeneración espacial de la tabla sino que acerca los elementos a su valor de dispersión y, además, según la distribución de los elementos, puede partir las cadenas en dos. Ahora bien, la conveniencia del movimiento de elementos ha de ser estudiada cuidadosamente porque puede acarrear los problemes ya conocidos.

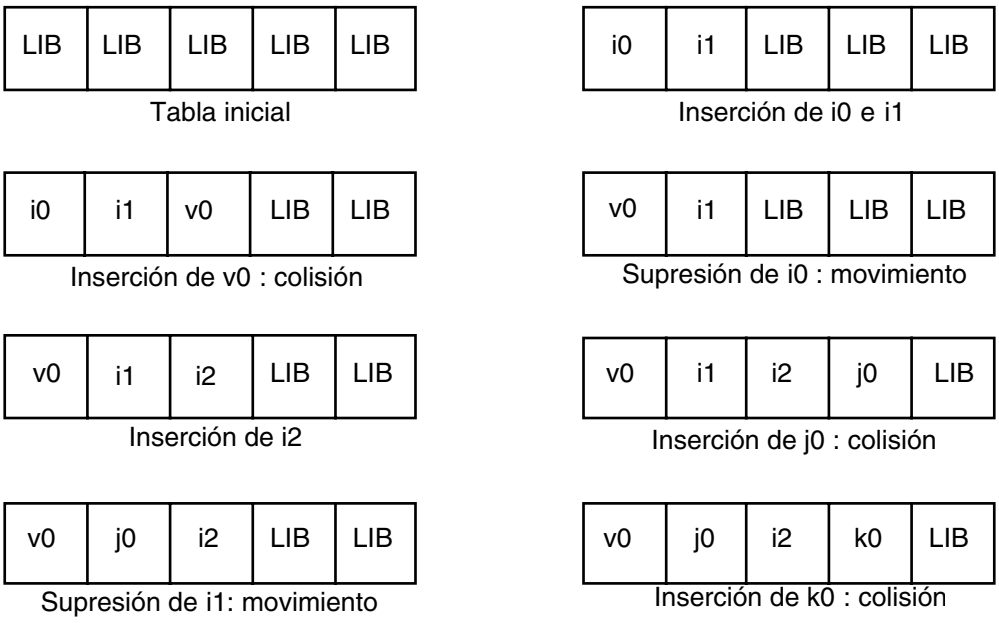


Fig. 4.20: id. fig. 4.19, pero con reubicación de elementos en la supresión.

d) Redispersión doble

La familia de redispersión se define a partir de una segunda función primaria de dispersión.

$$h_i(k) = (h(k) + i \,.\, h'(k)) \underline{\text{mod}}\ r$$

Si se quiere explorar toda la tabla, es necesario que h'(k) produzca un valor entre 0 y r -1 que sea primo con r (lo más sencillo es elegir r primo, o bien $r = 2^s$, y que h' sólo genere números impares). La ventaja respecto el método anterior es que, como h' depende de la clave, es muy improbable que h y h' colisionen a la vez; ahora bien, si esto sucede, vuelve a formarse

una cadena de reubicación, aunque en este caso el apiñamiento no es tan grave como en el caso lineal. Los resultados de este método se acercan a la redispersión uniforme o aleatoria de manera que, a efectos prácticos, son indistinguibles.

Tradicionalmente, la forma concreta de h' depende de la elección de h; así, por ejemplo, si $h(k) = k \underline{\text{mod}}\ r$, puede definirse $h'(k) = 1+(k \underline{\text{mod}}\ (r-2))$; idealmente, r y r -2 deberían ser primos (por ejemplo, 1021 y 1019). Otra posibilidad es tomar h' que no obligue a efectuar nuevas divisiones, como $h(k) = k \underline{\text{mod}}\ r$ y $h'(k) = k / r$ (obligando que el resultado sea diferente de cero). Es importante, eso sí, que h y h' sean independientes entre sí para que la probabilidad de que h(k) = h(k') y a la vez h'(k) = h'(k') sea cuadrática respecto a r, no lineal.

Notemos que, a diferencia del método lineal, no se pueden mover elementos, porque al borrarlos no se sabe trivialmente de qué cadenas forman parte.

e) Redispersión cuadrática

Es una variante del método lineal que usa el cuadrado del ensayo:

$$h_i(k) = (h(k) + i^2) \underline{\text{mod}}\ r$$

Dado que la distancia entre dos valores consecutivos depende del número de ensayos, una colisión secundaria no implica la formación de cadenas siempre que $h(k) \neq h(k')$; ahora bien, si h(k) = h(k'), entonces las cadenas son idénticas.

La rapidez de cálculo puede mejorarse transformando el producto en sumas con las siguientes relaciones de recurrencia:

$$h_{i+1}(k) = (h_i(k) + d_i) \underline{\text{mod}}\ r, \text{ con } h_0(k) = 0$$
$$d_{i+1} = d_i + 2, \text{ con } d_0 = 1$$

Tal como se ha definido, no se recorren todas las posiciones de la tabla; como máximo, si r es primo, la secuencia $h_i(k)$ puede recorrer $\lceil r/2 \rceil$ posiciones diferentes (dado que el ensayo i es igual al ensayo r-i, módulo r en ambos casos), por lo que la tabla puede tener sitios libres y la técnica de reubicación no los encuentre. En la práctica, este hecho no es demasiado importante, porque si después de $\lceil r/2 \rceil$ ensayos no se encuentra ningún sitio libre significa que la tabla está casi llena y debe regenerarse, pero en todo caso se puede solucionar el problema obligando a r, además de ser primo, a tomar la forma 4s + 3 y redefiniendo la función como $h_i(k) = (h(k) + (-1)^i\, i^2) \underline{\text{mod}}\ r$.

f) Otras

Siempre se pueden definir otras familias ad hoc, sobre todo aprovechando la forma concreta que tiene la función de dispersión primaria. Por ejemplo, dada la función descrita en el apartado 4.3.3, podemos definir una familia de redispersión tal que $h_i(k)$ se define como la aplicación de h sobre repr(k)+i, donde repr(k) es la representación binaria de k (de manera similar al tratamiento de claves largas); esta estrategia genera todas las posiciones de la tabla.

4.4.3 Caracterización e implementación de los métodos de redispersión

Como se hizo con las funciones, es necesario encapsular en universos los diferentes métodos de redispersión que se han presentado. La caracterización de las funciones de redispersión ya ha sido definida en el universo CLAVE_REDISPERSIÓN ; el siguiente paso consiste en caracterizar los métodos. Como ejemplos, estudiamos las familias lineal y doble.

El método lineal puede considerarse como un algoritmo parametrizado por una función de dispersión, tal como está definida en CLAVE_DISPERSIÓN, más la constante multiplicadora (v. fig. 4.21). Para definir una familia de redispersión lineal basta con concretar la función primaria de dispersión; en la fig. 4.22, tomamos la función definida en SUMA_POND_Y_DIV.

```
universo REDISPERSIÓN_LINEAL(CLAVE_REDISPERSIÓN_LINEAL) es
    usa NAT
    función redispersión_lineal (i es nat; v es elem) devuelve nat es
    devuelve (h(v) + ct*i) mod val
funiverso

universo CLAVE_REDISPERSIÓN_LINEAL caracteriza
    usa CLAVE_DISPERSIÓN, NAT, BOOL
    ops ct: → nat
    ecns (ct = 0) = falso
funiverso
```

Fig. 4.21: caracterización de la redispersión lineal.

```
universo REDISP_LINEAL_SUMA_POND_Y_DIV(E es ELEM_DISP_CONV;
                                       V1, CT son VAL_NAT) es
  instancia priv SUMA_POND_Y_DIV(F es ELEM_DISP_CONV; V2 es VAL_NAT) donde
      F.elem es E.elem, F.ítem es E.ítem
      F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
      F.conv es E.conv, F.base es E.base; V2.val es V1.val
  instancia REDISPERSIÓN_LINEAL(K es CLAVE_REDISPERSIÓN_LINEAL) donde
      K.elem es E.elem, K.ítem es E.ítem
      K.= es E.=, K.nítems es E.ítems, K.i_ésimo es G.i_ésimo
      K.conv es E.conv, K.base es E.base
      K.val es V1.val, K.h es suma_pond, K.ct es CT.val
  renombra redispersión_lineal por redisp_lin_suma_pond_y_div
funiverso
```

Fig. 4.22: una instancia de la redispersión lineal.

Por lo que respecta al método doble, es necesario definir dos funciones de dispersión como parámetros formales (v. fig. 4.23). A continuación, se puede instanciar el universo parametrizado, por ejemplo con la función de suma ponderada y división, y separando el módulo de ambas funciones por dos (v. fig. 4.24).

```
universo REDISPERSIÓN_DOBLE (CLAVE_REDISPERSIÓN_DOBLE) es
    usa NAT
    función redispersión_doble (i es nat; v es elem) devuelve nat es
    devuelve (h(v) + h2(v)*i) mod val
funiverso

universo CLAVE_REDISPERSIÓN_DOBLE caracteriza
    usa CLAVE_DISPERSIÓN, NAT, BOOL
    ops h2: elem → nat
    ecns ∀v∈ elem: h2(v) < val = cierto
funiverso
```

Fig. 4.23: caracterización de la redispersión doble.

```
universo REDISP_DOBLE_SUMA_POND_Y_DIV(E es ELEM_DISP_CONV;
                                      V1, CT son VAL_NAT) es
  instancia priv SUMA_POND_Y_DIV(F es ELEM_DISP_CONV; V2 es VAL_NAT) donde
      F.elem es E.elem, F.ítem es E.ítem
      F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
      F.conv es E.conv, F.base es E.base; V2.val es V1.val
  renombra suma_pond por hprim
  instancia priv SUMA_POND_Y_DIV(F es ELEM_DISP_CONV, V2 es VAL_NAT) donde
      F.elem es E.elem, F.ítem es E.ítem
      F.= es E.=, F.nítems es E.ítems, F.i_ésimo es G.i_ésimo
      F.conv es E.conv, F.base es E.base; V2.val es V1.val+2
  renombra suma_pond por hsec
  instancia REDISPERSIÓN_DOBLE(K es CLAVE_REDISPERSIÓN_DOBLE) donde
      K.elem es E.elem, K.ítem es E.ítem
      K.= es E.=, K.nítems es E.ítems, K.i_ésimo es E.i_ésimo
      K.conv es E.conv, K.base es E.base
      K.val es V1.val, K.h es hprim, K.h2 es hsec
  renombra redispersión_doble por redisp_doble_suma_pond_y_div
funiverso
```

Fig. 4.24: una instancia de la redispersión doble.

4.4.4 Variantes de las tablas de direccionamiento abierto

Hay algunas variantes de la estrategia de direccionamiento abierto implementada en la fig. 4.17, que permiten reducir el coste de las búsquedas a costa de mover elementos al insertar. Así, estas variantes se pueden usar para satisfacer mejor los criterios de eficiencia de una aplicación dada, siempre que el coste de mover los elementos no sea elevado (es decir, que su dimensión no sea demasiado grande y que la modificación de los posibles apuntadores a los elementos que se mueven no sea prohibitiva dados los requerimientos de eficiencia del problema). La implementación de los algoritmos resultantes queda como ejercicio para el lector; se puede consultar, por ejemplo, [TeA86, pp. 531-535].

a) Algoritmo de Brent

Estrategia útil cuando el número de consultas con éxito es significativamente más grande que el número de inserciones, de manera que en estas últimas se trabaja más para favorecer a las primeras; su comportamiento es comparativamente mejor, sobre todo a medida que la tabla está más llena. El algoritmo de Brent ha sido tradicionalmente asociado a la estrategia de redispersión doble, porque es cuando presenta los mejores resultados.

Su funcionamiento es el siguiente: si al insertar un nuevo par <k, v> se ha generado el camino $p_0, \ldots, p_s$, definiendo el ensayo $p_i = (h(k) + i.h'(k)) \underline{\text{mod}}\ r$ y tal que p_s es la primera posición vacía de la secuencia, encontramos dos casos:

- Si $s \leq 1$, se actúa como siempre.
- Si no, sea k' la clave que reside en p_0 y sea $c_0 = h'(k')$.
  - ◊ Si la posición $(p_0+c_0) \underline{\text{mod}}\ r$ (que es la siguiente a p_0 en el camino asociado a k') está vacía, entonces se mueve k' a esta posición (con su información asociada) y se inserta el par <k, v> en p_0. Notemos que, después de la inserción, el algoritmo de Brent determina z+1 ensayos para encontrar k', pero sólo un ensayo para encontrar k, gracias al desalojo del invasor que residía en la posición de dispersión, mientras que sin usar el algoritmo el número de ensayos es z para encontrar k' y s+1 para encontrar k, $s \geq 2$. Así, pues, se gana como mínimo un acceso con esta estrategia.
  - ◊ Si la posición $(p_0+c_0) \underline{\text{mod}}\ r$ está llena, se aplica recursivamente la misma estrategia, considerando p_1 como el nuevo p_0, p_2 como el nuevo p_1, etc.

b) Dispersión ordenada

Si las claves de dispersión presentan una operación de comparación (que sería un parámetro formal más del universo de implementación), se pueden ordenar todos los elementos que residen en una misma cadena de reubicación de manera que las búsquedas sin éxito acaben antes. Igual que en el anterior método, esta variante se acostumbra a implementar con redispersión doble. Concretamente, al insertar el par <k, v> tal que h(k) = i puede ocurrir que:

- la posición i esté vacía: se almacena el par en ella;
- la posición i contenga el par <k, v'>: se sustituye v' por v;

- si no, sea <k', v'> el par que reside en la posición i. Si k' < k, se repite el proceso examinando la siguiente posición en el camino de k ; si k' > k, se guarda <k, v> en la posición i y se repite el proceso sobre el par <k', v'> a partir del sucesor de i en el camino de k'.

c) Método Robin Hood

Ideado por P. Celis, P.-Å. Larso y J.I. Munro y publicado en Proceedings 26th Foundations of Computer Science (1985), está orientado a solucionar dos problemas que presentan los métodos de direccionamiento abierto vistos hasta ahora: por un lado, todos ellos aseguran el buen comportamiento esperado de las operaciones sobre tablas, pero no el de una operación individual; por el otro, pierden eficiencia cuando la tabla se llena. Una estrategia de inserción que recuerda vagamente a la variante de Brent ofrece ventajas en estos aspectos.

Consideremos el siguiente algoritmo de inserción: supongamos que los j-1 primeros ensayos de la inserción de una clave k no tienen éxito y que se consulta la posición p correspondiente al j-ésimo ensayo. Si esta posición p está vacía o contiene un par <k, v>, se actúa como siempre; si no, está ocupada por un elemento k' que se depositó en la tabla en el i-ésimo ensayo, y en este caso se sigue la siguiente casuística:

- Si i > j, entonces k es rechazada por p y se sigue con el j+1-ésimo ensayo.
- Si i < j, entonces k desplaza a k' de la posición p y es necesario insertar k' a partir del i+1-ésimo ensayo.
- Si i = j, no importa.

La segunda de estas reglas da nombre al método, porque favorece la clave k más "pobre" (es decir, que exige más ensayos al buscarla) sobre la clave k' más "rica". El resultado es que, aunque el número medio de ensayos necesario para encontrar los elementos en la tabla no varía (al contrario que en el esquema de Brent), no hay claves extraordinariamente favorecidas ni tampoco perjudicadas (en términos estadísticos, la variancia de la variable correspondiente queda $\Theta(1)$), ni siquiera cuando la tabla se llena. Eso sí, el método requiere que se sepa el número de ensayo de todo elemento de la tabla (ya sea con un campo adicional, ya sea implícitamente dada la família de redispersión).

4.4.5 Tablas coalescentes

Las tablas coalescentes (ing., coalesced hashing) son una organización intermedia entre las encadenadas y el direccionamiento abierto. Por un lado, se encadenan los sinónimos y, opcionalmente, los sitios libres; por el otro, todos los elementos ocupan la misma zona dentro de un vector dimensionado de 0 a r - 1, y la supresión presenta los problemes típicos del direccionamiento abierto.

En concreto, al insertar una clave k en la tabla se siguen los encadenamientos que salen de la posición h(k) hasta que se encuentra la clave (si ya existía), o bien se llega al final de la cadena. En el primer caso se sustituye la información que había por la nueva, mientras que en el segundo se obtiene una posición libre, se inserta el par <k, v> y se encadena esta posición con la última de la cadena. La estrategia de inserciones provoca que las cadenas estén formadas por elementos con diferentes valores de dispersión; eso sí, todos los sinónimos de un mismo valor de dispersión forman parte de la misma cadena. Esta propiedad no provoca ninguna situación incorrecta, porque la longitud de las cadenas se mantiene corta incluso cuando los algoritmos provocan fusiones de cadenas, y asegura buenos resultados en las búsquedas. De hecho, los algoritmos podrían modificarse para evitar las fusiones y así tener cadenas sin sinónimos de diferentes claves, pero el beneficio sería tan escaso que no se justifica.

La organización coalescente presenta apiñamiento, porque todas las claves que tienen como valor de dispersión cualquier posición que forme parte de una cadena irán a parar a ella, y las cadenas largas tienen más probabilidad de crecer que las cortas.

Por lo que respecta a la supresión, se puede optar por marcar las posiciones borradas o por reubicar los sinónimos. En el primer caso, conviene reciclar las posiciones borradas en la inserción antes de ocupar nuevas posiciones libres. En la segunda opción, si hay sinónimos de diversos valores de dispersión dentro de una misma cadena, puede que no baste mover uno solo, porque este sinónimo puede dejar otro agujero dentro de la cadena que esconda otras claves posteriores. En consecuencia, en el caso general es necesario efectuar varios movimentos hasta dejar la cadena en buen estado. A veces, estos movimientos acaban fraccionando la cadena original en dos o más diferentes, y reducen el grado de apiñamiento que presenta la tabla. Eso sí, es necesario evitar mover una clave a una posición de la cadena anterior a su valor de dispersión.

Tal como se ha dicho, la gestión del espacio libre se puede hacer o no encadenada. En caso de formar la típica pila de sitios libres, es necesario encadenar doblemente los elementos, porque en todo momento se ha de poder ocupar de manera rápida cualquier posición libre. En caso de gestionarse secuencialmente, se dispondrá de un apuntador que inicialmente valga, por ejemplo, r - 1 y que se actualize a medida que se inserten y borren elementos. Esta estrategia, aunque pueda parecer más ineficiente, no perjudica significativamente el tiempo de las inserciones o las supresiones.

Destaquemos que este método presenta una variancia muy baja en las búsquedas (v. [GoB91, p.62]; en este texto, se presenta también una variante que implementa las ideas de la dispersión coalescente usando una zona de excedentes para los sinónimos).

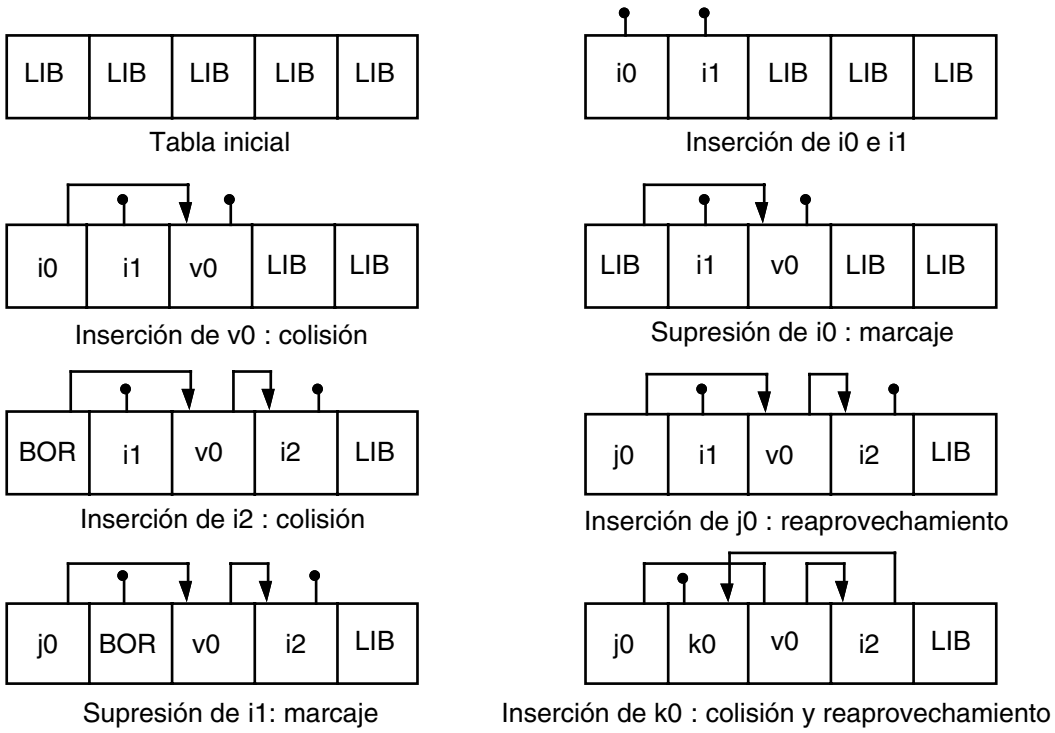


Fig. 4.25: ídem fig. 4.19 en una tabla coalescente con marcaje en la supresión.

4.4.6 Evaluación de las diferentes organizaciones

En la fig. 4.26 se presentan las fórmulas que permiten comparar las diferentes estrategias comentadas. En concreto, se estudian las esperanzas de las magnitudes $S_n(k)$ y $U_n(k)$, que a veces pueden variar si la tabla está (casi) llena. Podríamos haber considerado otros tipo de medida, por ejemplo, incorporar el seguimiento de encadenamientos al estudio, o bien las comparaciones entre apuntadores. Sea como sea, las variaciones respecto a los resultados que se ofrecen serían muy pequeñas y, de todas maneras, el comportamiento general de los métodos queda bien reflejado. Por motivos de espacio, no se incluye la deducción de las fórmulas, pero hay varios libros y artículos que sí las presentan, entre los que destacan [Knu73] y [GoB91].

Los resultados obtenidos dependen de la tasa de ocupación de la tabla, α, denominada factor de carga (ing., load factor), definido como el cociente n/r, siendo n el número de elementos en la tabla más las posiciones marcadas como borradas (si la estrategia tiene), y Z_r el codominio de la función de dispersión. Cuanto más grande es α, peor se comporta la tabla, y la mayoría de las organizaciones presentan un punto a partir del cual la degeneración es demasiado acusada para que la tabla valga la pena.

Dispersión encadenada indirecta:

$\varepsilon[S_n(k)] \approx \alpha/2$

$\varepsilon[U_n(k)] \approx \alpha$

Dispersión encadenada con zona de excedentes:

$\varepsilon[S_n(k)] \approx \alpha/2$

$\varepsilon[U_n(k)] \approx \alpha + e^{-\alpha}$

Redispersión uniforme:

$\varepsilon[S_n(k)] \approx -\alpha^{-1}\ln(1-\alpha)$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma - 1 + o(1)$ $(\gamma = 0.577...)$

$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$

Redispersión aleatoria:

$\varepsilon[S_n(k)] \approx -\alpha^{-1}\ln(1-\alpha)+O((r-n)^{-1})$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma + O(1/r)$

$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$

Redispersión lineal:

$\varepsilon[S_n(k)] \approx 0.5(1+(1-\alpha)^{-1})$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx (\pi r/8)^{1/2}$

$\varepsilon[U_n(k)] \approx 0.5(1+(1-\alpha)^{-2})$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$

Redispersión doble:

$\varepsilon[S_n(k)] \approx -\alpha^{-1}\ln(1-\alpha)+O(1)$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma + O(1)$

$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}+o(1)$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$

Redispersión cuadrática:

$\varepsilon[S_n(k)] \approx 1 - \ln(1-\alpha) - \alpha/2$

$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1} - \ln(1-\alpha) - \alpha$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$

Algoritmo de Brent:

$\varepsilon[S_n(k)] \approx 1 + \alpha/2 + \alpha^3/3 + \alpha^4/15 - \alpha^5/18 + ...$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx 2.49$

$\varepsilon[U_n(k)] \approx (1-\alpha)^{-1}$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx r$

Dispersión ordenada:

$\varepsilon[S_n(k)] \approx -\alpha^{-1}\ln(1-\alpha)+o(1)$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] \approx \ln r + \gamma + O(1)$

$\varepsilon[U_n(k)] \approx -\alpha^{-1}\ln(1-\alpha)$; $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx \ln(r+1) + \gamma + O(1/(r+1))$

Variante Robin Hood:

$\varepsilon[S_n(k)] = O(1)$; $\alpha \approx 1.0 \Rightarrow \varepsilon[S_n(k)] < 2.57$ (usando búsquedas no tradicionales)

$\varepsilon[U_n(k)] \approx$ desconocido (pero $< \log r$); $\alpha \approx 1.0 \Rightarrow \varepsilon[U_n(k)] \approx \log r$ (ídem)

Dispersión coalescente:

$\varepsilon[S_n(k)] \approx 1 + (e^{2\alpha} - 1 - 2\alpha)/4 + O(1/r)$

$\varepsilon[U_n(k)] \approx 1 + (e^{2\alpha} - 1 - 2\alpha)/8\alpha + \alpha/4$

Fig. 4.26: fórmulas de los diversos métodos de dispersión.

Se ha supuesto que las funciones de dispersión distribuyen uniformemente el dominio de las claves en el intervalo Z_r; ahora bien, es necesario señalar que todas las organizaciones pueden comportarse tan mal como se pueda temer, para un subconjunto de claves determinado que provoque un mal funcionamiento de la función de dispersión.

4.4.7 Elección de una organización de dispersión

Una vez conocidas las características de las diferentes organizaciones de tablas de dispersión, es lógico preguntarse cuáles son los criterios que conducen a la elección de una u otra en un contexto determinado. Destacan los siguientes puntos:

- Tiempo. A la vista de las fórmulas de la fig. 4.26 está claro que, a igual factor de carga, las estrategias encadenadas dan mejores resultados que las otras. Ahora bien, en algún momento puede merecer la pena incrementar la dimensión de una tabla no encadenada para disminuir este factor y, consecuentemente, el tiempo de acceso. Eso sí, en cualquier caso el coste asintótico será siempre $\Theta(1)$ para factores de carga razonables, suponiendo que la función de dispersión distribuya las claves de manera uniforme.
- Espacio. Si se desconoce el número esperado de elementos, es recomendable usar la estrategia encadenada indirecta implementada con punteros, donde sólo se desaprovecha a priori el espacio del índice y, posteriormente, los encadenamientos; además, las estrategias encadenadas soportan factores de carga superiores al 100% sin que la eficiencia temporal se resienta[8]. Ahora bien, si se conoce con cierta exactitud el número de elementos esperado en la tabla, es necesario recordar que las estrategias de direccionamiento abierto no precisan encadenamientos; eso sí, siempre se deberá dimensionarlas en exceso para evitar las casi-asíntotas existentes a partir de factores de carga del orden del 60%.
- Frecuencia de las supresiones. Si hay muchas supresiones la política encadenada indirecta generalmente es mejor, dado que cualquier otra exige, o bien movimiento de elementos, o bien marcaje de posiciones. Si los movimentos son desdeñables (si no hay apuntadores externos a la estructura y los elementos no son de un tamaño demasiado grande), o bien si las supresiones son escasas o inexistentes, este factor no influye en la elección del tipo de tabla.

A la vista de los resultados, parece claro que la mayoría de las veces podemos decantarnos por la estrategia indirecta, que es rápida, soporta perfectamente las supresiones, no exige un conocimiento exacto del número de elementos y soporta desviaciones en la previsión de

[8] Si el desconocimiento es total, puede ser desaconsejable el uso de dispersión, pues esta técnica exige conjeturar el valor de r, y una mala elección lleva a desperdiciar mucho espacio en caso de sobredimensionar vectores, o bien a regenerar la tabla en caso de llegar a un factor de carga prohibitivo. En la sección 5.6 se muestra una alternativa a la dispersión, los árboles de búsqueda, que se adaptan mejor a esta situación, sacrificando algo de eficiencia temporal.

ocupación; además, su programación es trivial. Sólo queda por decidir si las listas se implementan en memoria dinámica o, por el contrario, en un único vector. Ahora bien, bajo ciertos requerimientos pueden ser más indicadas otras estrategias; veamos un ejemplo.

Se quiere implementar los conjuntos de naturales con operaciones de conjunto vacío, añadir un elemento y pertenencia, y con la propiedad de que, a la larga, se espera que vayan a almacenarse n naturales en el conjunto; este tipo de conjuntos puede emplearse para registrar los elementos a los que se ha aplicado un tratamiento determinado. Dado que no hay operación de supresión, no se descarta inicialmente ningún método. Por otro lado, como el tamaño de los elementos es muy pequeño y el número que habrá es conocido, la estrategia de direccionamiento abierto puede ser la indicada, puesto que evita el espacio requerido por los encadenamientos. Ahora bien, sería incorrecto dimensionar la tabla de n posiciones porque, al llenarse, el tiempo de acceso a los conjuntos sería muy grande. Consultando las fórmulas, vemos que un factor de carga del 60% da resultados razonables siempre que la función de dispersión distribuya bien.

Para confirmar que la elección es adecuada, es necesario comparar la sobredimensión de la tabla con el ahorro respecto a las otras estrategias. Supongamos que un encadenamiento ocupa el mismo espacio z que un natural; en este caso, las diferentes estrategias necesitan el siguiente espacio:

- Direccionamiento abierto: la tabla tiene (100/60)n posiciones y cada posición ocupa z.
- Coalescente: la tabla tiene, como mínimo, n posiciones y cada posición ocupa 2z (natural + encadenamiento).
- Encadenado indirecto: el vector índice ocupa del orden de n posiciones de espacio z cada una de ellas, y cada uno de los n elementos del conjunto ocupará 2z.
- Encadenado directo: el vector ocupa del orden de (100/86)n posiciones y cada posición ocupa 2z.

Comparando estas magnitudes puede confirmarse que el direccionamiento abierto presenta mejores resultados; incluso se podría aumentar la dimensión del vector hasta 2n para asegurar un buen comportamiento si la función no distribuye del todo bien. Si el contexto de uso de la tabla lo exige, se puede optar por una de las variantes presentadas en el apartado 4.4.4; no hay ningún problema en mover elementos, dado que son pequeños y no hay apuntadores externos hacia la tabla.

4.5 Tablas recorribles

Frecuentemente, es necesario que los elementos de una tabla no sólo sean accesibles individualmente, sino también que se puedan obtener todos ellos, ya sea ordenadamente o no. Con este objetivo, definimos un nuevo tipo abstracto de datos, las tablas recorribles, que puede considerarse como un híbrido entre una tabla y una secuencia. Para llevar la discusión a conceptos ya conocidos, y aunque hay otras opciones igualmente posibles, la estrategia que adoptaremos será añadir las operaciones de las listas con punto de interés sobre el TAD de las tablas.

Para empezar, es necesario decidir cuál es el modelo del TAD. Una manera sencilla de describir las operaciones consiste en tomar como modelo los pares de secuencias de pares de clave y valor, $(K \times V)^* \times (K \times V)^*$, con el significado habitual de las listas con punto de interés tales que no hay ninguna clave que aparezca repetida considerando los pares de ambas secuencias; también se puede tomar el modelo de los pares de funciones de claves a valores, $(f : K \to V) \times (g : K \to V)$, tales que $dom(f) \cap dom(g) = \emptyset$, e interpretar entonces que f representa los pares a la izquierda del punto de interés y g los pares de la derecha. Si se quiere que el recorrido sea ordenado, es necesario requerir que haya una operación de comparación de claves, que denotaremos por $<$, en K. En el primer modelo exigiremos que los elementos de las secuencias estén ordenados según $<$, mientras que en el segundo la ordenación conducirá a que el máximo del dominio de la primera función sea menor que el mínimo del dominio de la segunda función. Sea cual sea el modelo elegido, se definen las operaciones crea, asigna, borra y consulta propias de las tablas y las operaciones principio, actual, avanza y final?, propias de las listas con punto de interés. La signatura en el caso ordenado se presenta en la fig. 4.27, izquierda, sin detallar su comportamiento exacto, pues hay diversas decisiones que dependen bastante del contexto de uso (por ejemplo, qué sucede si se inserta un elemento a la izquierda del punto de interés durante un recorrido); las claves se definen en el universo ELEM_<_= de la fig. 4.28 y los valores en ELEM_ESP.

En la fig. 4.27, derecha, se muestra también una signatura para los conjuntos recorribles considerando que la clave es el elemento entero. Notemos que la signatura es muy parecida a la de las listas con punto de interés; de hecho, el conjunto recorrible se puede considerar como una lista con punto de interés sin repeticiones, borrando los elementos mediante su clave en vez de suprimir el elemento actual, y con operación de pertenencia.

Por lo que respecta a la implementación, estudiamos por separado el caso ordenado y el caso no ordenado para tablas, siendo similar el caso de los conjuntos.

```
universo TABLA_ORDENADA_RECORRIBLE (ELEM_<_=, ELEM_ESP) es
    usa BOOL
    renombra ELEM_<_=.elem por clave, ELEM_ESP.elem por valor
    instancia PAR(A, B son ELEM) donde A.elem es clave, B.elem es valor
    renombra par por elem_tabla
    tipo tabla
    ops crea: → tabla
        asigna: tabla clave valor → tabla
        borra: tabla clave → tabla
        consulta: tabla clave → valor
        principio, avanza: tabla → tabla
        actual: tabla → elem_tabla
        final?: tabla → bool
funiverso
```

```
universo CJT_RECORRIBLE (ELEM_=) es
    usa BOOL
    tipo cjt
    ops crea: → cjt
        añade, borra: cjt elem → cjt
        está?: cjt elem → bool
        principio, avanza: cjt → cjt
        actual: cjt → elem
        final?: cjt → bool
funiverso
```

Fig. 4.27: signatura de las tablas ordenadas (izq.) y de los conjuntos recorribles (der.).

```
universo ELEM_<_= caracteriza
    usa BOOL
    tipo elem
    ops
        _=_, _≠_, _<_: elem elem → bool
    ecns ... las habituales
funiverso
```

Fig. 4.28: universo de caracterización de los elementos de las tablas ordenadas recorribles.

a) Recorridos desordenados

Entendemos por recorrido desordenado aquel que no requiere ningún orden de visita de los elementos. La estrategia más sencilla consiste en no modificar la representación más que por el añadido del apuntador al elemento actual dentro del recorrido, de manera que el recorrido sea simplemente un examen secuencial de la tabla y quede de orden $\Theta(t+n)$, siendo t la dimensión del vector que aparece en la tabla (que vale r en todos los casos excepto en la dispersión encadenada directa); el factor n es necesario para cubrir el caso indirecto, donde el número de elementos puede ser superior a t. En esta opción, no obstante, las inserciones y/o supresiones incontroladas pueden provocar que un recorrido no examine todos los elementos de la tabla. Ahora bien, la mayoría de las veces que se usan las operaciones de recorrido no se alternan con las actualizaciones y, por ello, este problema raramente es relevante.

Esta implementación, pese a ser conceptualmente muy simple, no se corresponde con la especificación de una estructura con punto de interés. De hecho, el concepto de recorrido desordenado no se puede especificar con semántica inicial; siempre es necesario identificar algún criterio de recorrido de los elementos para evitar la aparición de inconsistencias en el modelo. Una implementación que sí cumple la especificación consiste en encadenar los elementos tal como es habitual en las estructuras con punto de interés. Con esta estrategia, la inserción queda $\Theta(1)$ mientras que en la supresión existen dos posibilidades: encadenar los elementos en los dos sentidos, de manera que la operación queda constante, o bien dejar encadenamientos simples, y en este caso la operación queda de orden lineal porque, si bien el elemento a borrar puede localizarse por dispersión, su predecesor en la lista no.

b) Recorridos ordenados

Es evidente que, para que el recorrido no sea excesivamente lento, es necesario que los elementos de la tabla estén ordenados según su clave. Existen dos opciones:

- Se puede tener la tabla desordenada mientras se insertan y se borran elementos y ordenarla justo antes de comenzar el recorrido. La ventaja de este esquema es que las operaciones propias de las tablas quedan $\Theta(1)$, aplicando dispersión pura, y también quedan eficientes el resto de operaciones excepto principio, que ha de ordenar los elementos de la tabla (con un buen algoritmo de ordenación, quedaría $\Theta(n\log n)$, siendo n el número de elementos de la estructura; evidentemente, el algoritmo ha de evitar movimientos de elementos para no destruir la estructura de dispersión).
- Como alternativa, se pueden mantener los elementos de la tabla siempre ordenados; esto implica que, para cada inserción, es necesario buscar secuencialmente la posición donde ha de ir el elemento y que resultará un orden lineal, pero las operaciones de recorrido quedarán constantes. Por lo que respecta a la supresión, queda igual que en el caso desordenado.

La elección de una alternativa concreta depende de la relación esperada entre altas, bajas y recorridos ordenados de la tabla. En ambos casos, sin embargo, la búsqueda de un elemento dada su clave es inmediata por dispersión.

Un modelo igualmente válido de las tablas recorribles consiste en sustituir las operaciones de punto de interés por una única operación que devuelva directamente la lista de pares clave-valor (ordenada o no). Evidentemente esta opción necesita un espacio adicional para la lista que no es necesario en el modelo anterior; además, la lista es una estructura diferenciada que se ha de construir a partir de la tabla y ello requiere un tiempo que no era necesario antes. No obstante, en algunas circunstancias puede ser una alternativa adecuada y, por ello, en la fig. 4.29 se introduce la signatura correspondiente a las tablas recorribles ordenadas. Notemos las instancias de los universos de los pares y las listas (públicas, porque los usuarios del tipo puedan servirse de éllas).

La discusión se puede adaptar al TAD de los conjuntos de manera obvia.

```
universo TABLA_ORDENADA (ELEM_<_=, ELEM_ESP) es
    renombra ELEM_<_=.elem por clave, ELEM_ESP.elem por valor
    instancia PAR(A, B son ELEM) donde A.elem es clave, B.elem es valor
    renombra par por elem_tabla, _.c1 por _.clave, _.c2 por _.valor
    instancia LISTA(C es ELEM) donde C.elem es elem_tabla
    renombra lista por lista_elem_tabla
    tipo tabla
    ops crea: → tabla
        asigna: tabla clave valor → tabla
        borra: tabla clave → tabla
        consulta: tabla clave → valor
        todos_ordenados: tabla → lista_elem_tabla
funiverso
```

Fig. 4.29: signatura del TAD de las tablas ordenadas recorribles que devuelve una lista.

Ejercicios

4.1 Modificar el modelo y la especificación de las tablas para el caso en que no haya elemento indefinido en el conjunto de valores asociados a las claves.

4.2 Implementar las tablas con todas las variantes de listas que aparecen en la sección 4.2.

4.3 Sean las funciones de dispersión h(k) = k mod 512 y h'(k) = k mod 557 aplicadas sobre claves naturales. Enumerar las ventajas y desventajas que presentan.

4.4 Programar en C, al máximo detalle y con la mayor eficiencia posible, las funciones de dispersión presentadas en la sección 4.3, suponiendo una tabla de dimensión de aproximadamente 5.000 elementos (determinar con exactitud la que sea apropiada en cada caso) y siendo las claves cadenas de caracteres compuestas sólo de letras mayúsculas y minúsculas (distinguibles entre ellas).

4.5 a) Se quiere organizar una tabla de dispersión de siete posiciones con estrategia de direccionamiento abierto y redispersión lineal. Sea la función h de dispersión que da los siguientes valores para las siguientes claves: h(sonia) = 3, h(gemma) = 5, h(paula) = 2, h(anna) = 3, h(ruth) = 3, h(cris) = 2.

i) Insertar todas estas claves en el siguiente orden: paula, anna, cris, ruth, sonia, gemma. Mostrar claramente cómo va evolucionando la tabla y indicar las colisiones que se producen. A continuación, mostrar el camino generado para buscar la clave ruth.

ii) Repetir el proceso cuando el orden de inserción sea: sonia, anna, ruth, gemma, paula,

cris. 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 ruth.

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

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

b) Repetir el apartado a) con una tabla de dispersión de siete posiciones con estrategia coalescente, suponiendo que los sitios libres se obtienen comenzando a buscarlos desde la última posición de la tabla en dirección a la primera.

4.6 Queremos almacenar en una tabla de dispersión 10.000 elementos de X bits de tamaño y tales que no existe 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 puntero 32 bits, determinar el espacio (en bits y en función de X) que ocuparía cada una de las representaciones habituales de dispersión, si se queremos que el número esperado de comparaciones entre claves en una búsqueda con éxito sea como mucho de 2.5.

4.7 Implementar la estrategia de direccionamiento abierto con redispersión lineal y con movimiento de elementos en la supresión.

4.8 En el año 1974, O. Amble y D.E. Knuth propusieron en el artículo "Ordered Hash Tables", publicado en The Computer Journal, 17, un método de dispersión similar a la estrategia de direccionamiento abierto con redispersión lineal, con la particularidad de que los elementos se podrían obtener ordenados sin necesidad de encadenar los elementos o reorganizar la tabla. La única (y muy restrictiva) condición era que la función de dispersión fuera monotónica, de manera que la gestión lineal de los sinónimos garantizara que los elementos quedaban realmente ordenados dentro del vector.

a) Pensar alguna situación donde sea posible definir una función de dispersión monotónica y definirla. (D. Gries muestra un ejemplo en Communicationes ACM, 30(4), 1987.)

b) Implementar la estrategia.