Capítulo 5 Árboles

Los árboles (ing., tree) son estructuras que organizan sus elementos, denominados nodos, formando jerarquías. Su definición inductiva es:

- Un único nodo es un árbol; a este nodo se le llama raíz del árbol.
- Dados n árboles a_1, ..., a_n, se puede construir uno nuevo como resultado de enraizar un nuevo nodo con los n árboles a_i. Generalmente, el orden de los a_i es significante y así lo consideramos en el resto del capítulo; algunos autores lo subrayan llamándolos árboles ordenados (ing., ordered trees). Los árboles a_i pasan a ser subárboles del nuevo árbol y el nuevo nodo se convierte en raíz del nuevo árbol.

La representación gráfica de los árboles refleja claramente la jerarquía resultante de su definición.

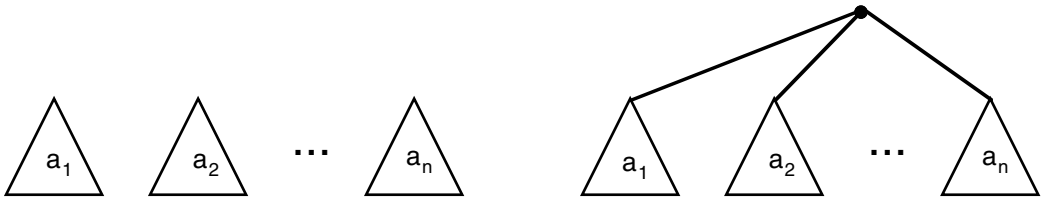


Fig. 5.1: n árboles (a la izquierda) y su enraizamiento formando uno nuevo (a la derecha).

El concepto matemático de árbol data de mediados del siglo XIX y su uso en el mundo de la programación es habitual. Ya ha sido introducido en la sección 1.2 como una representación intuitiva de la idea de término; en general, los árboles permiten almacenar cómodamente cualquier tipo de expresión dado que la parentización, la asociatividad y la prioridad de los operadores quedan implícitos y su evaluación es la simple aplicación de una estrategia de recorrido. Suelen utilizarse para representar jerarquías de todo tipo: taxonomías, relaciones entre módulos, etc. Igualmente es destacable su contribución a diversos campos de la informática: árboles de decisión en inteligencia artificial, representación de gramáticas y programas en el ámbito de la compilación, ayuda en técnicas de programación

© Los autores, 1998; © Edicions UPC, 1998.

(transformación de programas recursivos en iterativos, etc.), índices en ficheros de acceso por clave, bases de datos jerárquicas, etc.

5.1 Modelo y especificación

En esta sección introducimos el modelo formal asociado a un árbol, que emplearemos para formular diversas definiciones útiles, y también su especificación ecuacional. Hablando con propiedad, hay varios modelos de árboles que surgen a partir de todas las combinaciones posibles de las características siguientes:

- El proceso de enraizar puede involucrar un número indeterminado de subárboles o bien un número fijo n. En el primer caso hablamos de árboles generales (ing., general tree) y en el segundo caso de árboles n-arios (ing., n-ary tree)[1]; en estos últimos destaca el caso de n = 2, los denominados árboles binarios (ing., binary tree), sobre el cual nos centraremos a lo largo del texto dada su importancia, siendo inmediata la extensión a cualquier otro n. En los árboles n-arios se definirá el concepto de árbol vacío que ampliará el conjunto de valores del tipo.
- Los nodos del árbol pueden contener información o no; el primer caso es el habitual y responde al uso de árbol como almacén de información, pero también se puede dar la situación en que sólo interese reflejar la jerarquía, sin necesidad de guardar ningún dato adicional. En el resto del capítulo nos centraremos en el primer tipo de árbol, porque generaliza el segundo. La información en los nodos se denominará etiqueta (ing., label) y los árboles con información árboles etiquetados (ing., labelled tree).
- La signatura definida en los árboles puede ser muy variada. Destacamos dos familias: en la primera se utilizan árboles enteros tal como se ha explicado en la introducción, con operaciones de enraizar diversos árboles, obtener un subárbol y, si el árbol es etiquetado, obtener la etiqueta de la raíz; la segunda familia define un nodo de referencia para las actualizaciones y consultas, que puede cambiarse con otras operaciones. Llamaremos al segundo tipo árboles con punto de interés, por su similitud respecto a las listas de las mismas características.

5.1.1 Modelo de árbol general

Un árbol general se caracteriza por su forma, que determina una relación jerárquica, y por el valor de la etiqueta asociada a cada uno de los nodos del árbol. La forma del árbol se puede describir como un conjunto de cadenas de los naturales sin el cero, $P(N_0^*)$, de manera que cada cadena del conjunto identifica un nodo del árbol según una numeración consecutiva de izquierda a derecha comenzando por el uno, tal como se muestra en la fig. 5.2. Hay que

[1] La nomenclatura no es estándar; es más, diferentes textos pueden dar los mismos nombres a diferentes modelos y contradecirse entre sí.

© Los autores, 1998; © Edicions UPC, 1998.

resaltar que este conjunto es cerrado por prefijo y compacto; cerrado por prefijo, porque si la cadena $\alpha.\beta$ está dentro del conjunto que denota la forma de un árbol, para $\alpha,\beta \in N_0^*$, también lo está la cadena α; y compacto, porque si la cadena α.k está dentro del conjunto, para $\alpha \in N_0^*$, k$\in N_0$, también lo están las cadenas α.1, α.2, ..., α.(k-1).

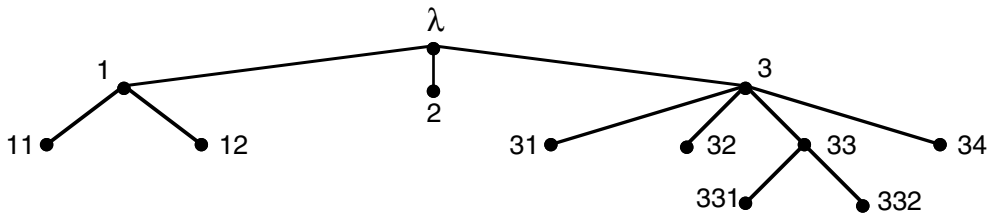


Fig. 5.2: representación del árbol de forma {λ, 1, 2, 3, 11, 12, 31, 32, 33, 34, 331, 332}.

Por lo que respecta a los valores de las etiquetas, se puede definir una función que asocie a una cadena un valor del conjunto de base de las etiquetas. Evidentemente, esta función es parcial y su dominio es justamente el conjunto de cadenas que da la forma del árbol.

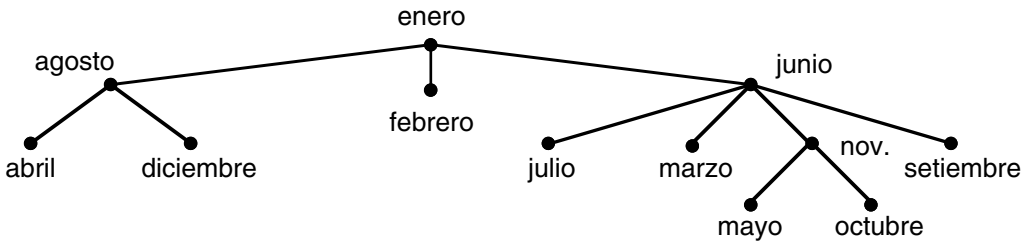


Fig. 5.3: el mismo árbol que en la fig. 5.2, asignando etiquetas a sus elementos.

Así, dado el conjunto base de las etiquetas V, definimos el modelo A_V de los árboles generales y etiquetados como las funciones que tienen como dominio las cadenas de naturales N_0^* y como codominio las etiquetas V, f : $N_0^* \rightarrow$ V, tales que su dominio dom(f) cumple que no es vacío, es cerrado por prefijo y compacto; a dom(f) le llamaremos la forma o contorno del árbol. A partir de este modelo se formulan una serie de definiciones útiles para el resto del capítulo, cuya especificación ecuacional queda como ejercicio para el lector.

a) Definiciones sobre los nodos

Sea a$\in A_V$ un árbol general y etiquetado cualquiera.

- Un nodo del árbol (ing., node) es un componente determinado por la posición que

© Los autores, 1998; © Edicions UPC, 1998.

ocupa y por la etiqueta; es decir, n=<s, v> es un nodo de a si $s \in dom(a)$ y $a(s) = v$. En la fig. 5.3, <11, abril> es un nodo pero no lo son ni <7, julio> ni <11, setiembre>. El conjunto $N_a = \{n / n \text{ es un nodo de a}\}$ es el conjunto de todos los nodos del árbol a.

- La raíz del árbol (ing., root) es el nodo que está más arriba, $<\lambda, a(\lambda)>$. Por ejemplo, la raíz del árbol de la fig. 5.3 es el nodo <λ, enero>. Notemos que, a causa de las propiedades del modelo, todo árbol a presenta una raíz. A veces, por comodidad y cuando quede claro por el contexto, denominaremos raíz simplemente a la etiqueta del nodo raíz.

- Una hoja del árbol (ing., leaf) es un nodo que está abajo de todo; es decir, el nodo $n=<s, v> \in N_a$ es hoja dentro de a si no existe $\alpha \in N_0^+$ tal que $s.\alpha \in dom(a)$. En el árbol de la fig. 5.3, son hojas los nodos <2, febrero> y <331, mayo>, pero no lo es <1, agosto>. El conjunto $F_a = \{n / n \text{ es hoja dentro de a}\}$ es el conjunto de todas las hojas del árbol a.

- Un camino dentro del árbol (ing., path) es una secuencia de nodos conectados dentro del árbol; o sea, la secuencia $c = n_1 \ldots n_r$, $\forall i: 1 \le i \le r: n_i = <s_i, v_i> \in N_a$, es camino dentro de a si cada nodo cuelga del anterior, es decir, si $\forall i: 1 \le i \le r-1: (\exists k: k \in N_0: s_{i+1} = s_i.k)$. La longitud del camino (ing., length) es el número de nodos menos uno, r - 1; si r > 1, se dice que el camino es propio (ing., proper). En el árbol de la fig. 5.3, la secuencia formada por los nodos <λ, enero><3, junio><32, marzo> es un camino propio de longitud 2. El conjunto $C_a = \{c / c \text{ es camino dentro de a}\}$ es el conjunto de todos los caminos del árbol a.

- La altura de un nodo del árbol (ing., height) es uno más que la longitud del camino desde el nodo a la hoja más lejana que sea alcanzable desde él[2]; es decir, dado el nodo $n \in N_a$ y el conjunto de caminos $C_a{}^n$ que salen de un nodo n y van a parar a una hoja, $C_a{}^n = \{c = n_1 \ldots n_k / c \in C_a \wedge n_1 = n \wedge n_k \in F_a\}$, definimos la altura de n dentro de a como la longitud más uno del camino más largo de $C_a{}^n$. En el árbol de la fig. 5.3, el nodo <3, junio> tiene altura 3. La altura de un árbol se define como la altura de su raíz; así, la altura del árbol de la fig. 5.3 es 4.

- El nivel o profundidad de un nodo del árbol (ing., level o depth) es el número de nodos que se encuentran entre él y la raíz; es decir, dado el nodo $n=<s, v> \in N_a$ definimos el nivel de n dentro de a como $||s||+1$, siendo $||s||$ la longitud de s; también se dice que n está en el nivel $||s||+1$. En el árbol de la fig. 5.3, el nodo <3, junio> está en el nivel 2. Por extensión, cuando se habla del nivel i de un árbol (también primer nivel, segundo nivel, y así sucesivamente) se hace referencia al conjunto de nodos que están en el nivel i. El número de niveles de un árbol a se define como el nivel de la hoja más profunda; así, el número de niveles del árbol de la fig. 5.3 es 4. Notemos que, por

[2] Las definiciones de altura, profundidad y nivel son también contradictorias en algunos textos. Hay autores que no definen alguno de estos conceptos, o los definen sólo para un árbol y no para sus nodos, o bien empiezan a numerar a partir del cero y no del uno, etc. En nuestro caso concreto, preferimos numerar a partir del uno para que el árbol vacío (que aparece en el caso n-ario) tenga una altura diferente al árbol con un único nodo.

© Los autores, 1998; © Edicions UPC, 1998.

definición, el número de niveles de un árbol es igual a la altura, por lo que pueden usarse ambas magnitudes indistintamente.

b) Definiciones sobre las relaciones entre nodos

A partir del concepto de camino, damos diversas definiciones para establecer relaciones entre nodos. Sean $a \in A_V$ un árbol general y etiquetado y $n,n' \in N_a$ dos nodos del árbol, $n = <s, v>$ y $n' = <s', v'>$; si hay un camino $c = n_1 \ldots n_r \in C_a$ tal que $n_1 = n$ y $n_r = n'$, decimos que:

- El nodo n' es descendiente (ing., descendant) de n y el nodo n es antecesor (ing., ancestor) de n'. Si, además, r > 1, los llamamos descendiente o antecesor propio.
- Si, además, r = 2 (es decir, si n' cuelga directamente de n), entonces:
  - ◊ El nodo n' es hijo (ing., child) de n; en concreto, si $s' = s.k$, $k \in N_0$, decimos que n' es hijo k-ésimo de n (o también primer hijo, segundo hijo, y así hasta el último hijo).
  - ◊ n es padre (ing., parent) de n'.
  - ◊ Hay una rama (ing., branch) entre n y n'.
- El grado o aridad (ing., degree o arity) de n es el número de hijos del nodo; es decir, se define $\text{grado}_n = ||\{\alpha \in \text{dom}(a) / \exists k,s: k \in N_0 \wedge s \in N_0^*: \alpha = s.k\}||$. La aridad de un árbol se define como el máximo de la aridad de sus nodos.

Además, dos nodos n y n' son hermanos (ing., sibling) si tienen el mismo padre, es decir, si se cumple que $s = \alpha.k$ y $s' = \alpha.k'$, para $k,k' \in N_0$ y $\alpha \in N_0^*$. Si, además, $k = k'+1$, a n' lo llamaremos hermano izquierdo de n y a n, hermano derecho de n'. En el árbol de la fig. 5.3, el nodo $<\lambda$, enero> es antecesor propio del nodo <3, junio>, también del <32, marzo> y del <1, agosto> (y, consecuentemente, estos tres nodos son descendientes propios del primero). Además, <3, junio> y <1, agosto> son hijos de $<\lambda$, enero> (y, en consecuencia, $<\lambda$, enero> es padre de los nodos <3, junio> y <1, agosto>, que son hermanos, y hay una rama entre $<\lambda$, enero> y <3, junio> y entre $<\lambda$, enero> y <1, agosto>). El grado del nodo raíz es 3 porque tiene tres hijos, mientras que el grado de todas las hojas es 0 porque no tienen ninguno, siendo ésta una propiedad común a todos los árboles del modelo.

c) Definiciones sobre subárboles

Sean $a,a' \in A_V$ dos árboles generales y etiquetados. Decimos que a es subárbol hijo de a', o simplemente subárbol (ing., subtree), si está contenido en él, o sea, si $\exists \alpha \in N_0^*$ tal que:

- Se superponen: $\alpha \,\#\, \text{dom}(a) = \{\gamma / \gamma \in N_0^* \wedge \alpha.\gamma \in \text{dom}(a')\}$, con $\#: N_0^* \times P(N_0^*) \to P(N_0^*)$ definida como: $\alpha \,\#\, \{s_1, \ldots, s_n\} = \{\alpha.s_1, \ldots, \alpha.s_n\}$.
- Las etiquetas de los nodos se mantienen: $\forall <s, v>: <s, v> \in N_a: a'(\alpha.s) = v$.
- No hay nodos por debajo de la superposición: $\neg(\exists <s, v> \in N_a: \exists \beta \in N_0^+: \alpha.s.\beta \in \text{dom}(a'))$.

Concretamente, en este caso decimos que a es α-subárbol de a'; si, además, $||\alpha|| = 1$, a se denomina primer subárbol de a' si $\alpha = 1$, segundo subárbol de a' si $\alpha = 2$, etc., hasta el último subárbol. Por ejemplo, en la fig. 5.4 se presentan un árbol que es subárbol del árbol de la fig. 5.3 (a la izquierda, en concreto, su tercer subárbol) y otro que no lo es (a la derecha).

© Los autores, 1998; © Edicions UPC, 1998.

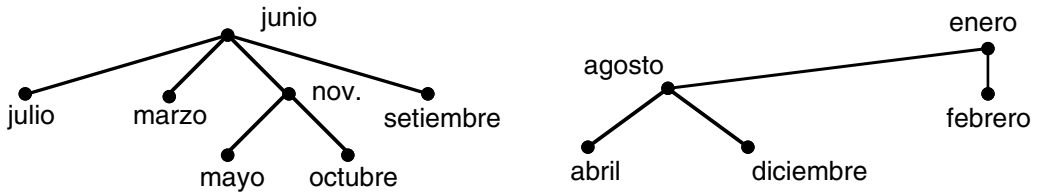


Fig. 5.4: un subárbol del árbol de la fig. 5.3 (a la izquierda) y otro que no lo es (a la derecha).

Una vez determinado completamente el modelo y las definiciones necesarias para hablar con propiedad de los árboles generales, vamos a determinar su signatura y especificación. Tal como se ha dicho previamente, la operación básica de la signatura de este modelo es enraizar un número indeterminado de árboles para formar uno nuevo, que tendrá en la raíz una etiqueta dada, enraiza: $A_V{}^* \times V \rightarrow A_V$. Ahora bien, la sintaxis de las especificaciones de Merlí (y, en general, de ningún lenguaje de especificación o implementación) no permite declarar un número indeterminado de parámetros como parece requerir esta signatura y, por tanto, adoptamos un enfoque diferente. En concreto, introducimos el concepto de bosque (ing., forest), definido como una secuencia[3] de árboles, de manera que enraiza tendrá dos parámetros, el bosque que contiene los árboles para enraizar y la etiqueta de la raíz[4]. Así, dados el árbol $a \in A_V$, el bosque $b \in A_V{}^*$, $b = a_1 ... a_n$, y la etiqueta $v \in V$, la signatura es:

- Crear el bosque vacío: crea, devuelve el bosque λ sin árboles.
- Añadir un árbol al final del bosque: inserta(b, a), devuelve el bosque b.a.
- Averiguar el número de árboles que hay en el bosque: cuántos(b), devuelve n.
- Obtener el árbol i-ésimo del bosque: i_ésimo(b, i), devuelve a_i; da error si i > n o vale 0.
- Enraizar diversos árboles para formar uno nuevo: enraiza(b, v), devuelve un nuevo árbol a tal que la etiqueta de su raíz es v y cada uno de los a_i es el subárbol i-ésimo de a, es decir, devuelve el árbol a tal que:
  - ◊ $dom(a) = \{ \cup i : 1 \le i \le n : i \# dom(a_i) \} \cup \{\lambda\}$.
  - ◊ $a(\lambda) = v$.
  - ◊ $\forall i : 1 \le i \le n : \forall \alpha : \alpha \in dom(a_i) : a(i.\alpha) = a_i(\alpha)$.
- Averiguar la aridad de la raíz del árbol: nhijos(a), devuelve $grado_{<\lambda, a(\lambda)>}$.
- Obtener el subárbol i-ésimo del árbol: subárbol(a, i) devuelve el subárbol i-ésimo de a o da error si i es mayor que la aridad de la raíz o cero; es decir, devuelve el árbol a' tal que:
  - ◊ $dom(a') = \{\alpha / i.\alpha \in dom(a)\}$.
  - ◊ $\forall \alpha : \alpha \in dom(a') : a'(\alpha) = a(i.\alpha)$.
- Consultar el valor de la etiqueta de la raíz: raíz(a), devuelve $a(\lambda)$.

[3] No un conjunto, porque el orden de los elementos es significativo.

[4] También podríamos haber optado por un conjunto de operaciones $enraiza_1$, ..., $enraiza_n$, donde n fuera un número lo bastante grande, de manera que $enraiza_i$ enraizase i árboles con una etiqueta.

© Los autores, 1998; © Edicions UPC, 1998.

En la fig. 5.5 se muestra una especificación para este modelo. Notemos que el bosque se crea fácilmente a partir de una instancia de las secuencias tal como fueron definidas en el apartado 1.5.1, pero considerando el género del alfabeto como parámetro formal (definido en el universo de caracterización ELEM). La instancia va seguida de varios renombramientos para hacer corresponder los identificadores de un tipo con los del otro, y de la ocultación de los símbolos no usables. La especificación del tipo árbol es trivial tomando {enraiza} como conjunto de constructoras generadoras (puro) y usando las operaciones sobre bosques. Como hay dos apariciones diferentes del universo ELEM, se prefija cada una de ellas con un identificador. Es conveniente que los géneros del bosque y del árbol residan en el mismo universo, porque se necesitan recíprocamente.

```
universo ÁRBOL_GENERAL (A es ELEM) es
    usa NAT, BOOL
    tipo árbol
    instancia CADENA (B es ELEM) donde B.elem es árbol
    renombra _._ por inserta, ||_|| por cuántos, cadena por bosque
    esconde _ . _, [_], etc.
    ops enraiza: bosque A.elem → árbol
        nhijos: árbol → nat
        subárbol: árbol nat → árbol
        raíz: árbol → A.elem
    error ∀a∈ árbol; ∀i∈ nat: [i > nhijos(a) ∨ NAT.ig(i, 0)] ⇒ subárbol(a, i)
    ecns ∀b∈ bosque; ∀i∈ nat; ∀v∈ A.elem
        nhijos(enraiza(b, v)) = cuántos(b)
        subárbol(enraiza(b, v), i) = CADENA.i_ésimo(b, i)
        raíz(enraiza(b, v)) = v
funiverso
```

Fig. 5.5: especificación del TAD de los árboles generales.

5.1.2 Modelo de árbol binario

El modelo de árbol binario, denotado por $A^{2}{}_{V}$, son las funciones f : {1, 2}* → V, dado que un nodo puede tener, como mucho, dos hijos. Como en el caso de los árboles generales, el dominio de la función ha de ser cerrado por prefijo, pero, en cambio, se permite la existencia del árbol vacío y, como consecuencia, el árbol binario presentará discontinuidades cuando la operación de enraizamiento involucre un árbol vacío como primer hijo y un árbol no vacío como segundo hijo. En la fig. 5.6 se muestra algunos árboles binarios válidos; notemos que si interpretamos los árboles como generales, el de la derecha y el del medio son idénticos.

© Los autores, 1998; © Edicions UPC, 1998.

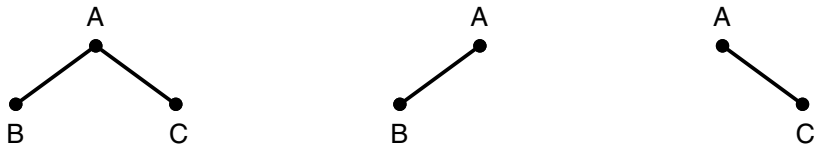


Fig. 5.6: tres árboles binarios tales que la raíz tiene: primer y segundo hijo (izquierda), primer hijo, pero no segundo (medio), y segundo hijo, pero no primero (derecha).

El resto del modelo no varía, siempre que se recuerde que el grado de todo nodo del árbol es, como mucho, igual a dos. Las definiciones dadas también son válidas respetando esta regla y considerando que la altura y la profundidad de un árbol vacío valen cero. Para mayor claridad, el primer hijo de un nodo de un árbol binario se llama hijo izquierdo (ing., left child) y el segundo se llama hijo derecho (ing., right child). De un árbol vacío también se puede decir que no existe; refiriéndose a un nodo, si un hijo (o hermano o cualquier otra relación de parentesco) de un nodo es vacío también podemos decir que el nodo no tiene este hijo (o hermano, etc.). Todas estas definiciones se pueden aplicar al caso de los subárboles.

Dados los árboles $a, a_1, a_2 \in A^2{}_V$ y la etiqueta $v \in V$, definimos las siguientes operaciones[5]:

- Crear el árbol vacío: crea, devuelve la función Ø de dominio vacío.
- Enraizar dos árboles y una etiqueta para formar un árbol: enraiza(a_1, v, a_2), devuelve el árbol a tal que $a(\lambda) = v$, a_1 es el subárbol izquierdo de a y a_2 es su subárbol derecho.
- Obtener los subárboles izquierdo y derecho de un árbol: izq(a) y der(a), respectivamente; si a es el árbol vacío, dan error.
- Obtener la etiqueta de la raíz: raíz(a), devuelve $a(\lambda)$ o da error si a es vacío.
- Averiguar si un árbol es vacío: vacío?(a), devuelve cierto si a es el árbol vacío y falso si no.

La aplicación en el caso de árboles n-arios cualesquiera, $A^n{}_V$, es inmediata, considerando el modelo como las funciones $f : [1, n]^* \rightarrow V$, y queda como ejercicio para el lector.

5.1.3 Modelo de árbol con punto de interés

Los árboles con punto de interés, ya sea generales o n-arios, se caracterizan por la existencia de un nodo privilegiado identificado por el punto de interés, que sirve de referencia para diversas operaciones de actualización. La etiqueta de este nodo es la única que puede consultarse y, en consecuencia, también habrá diversas operaciones de modificación del punto de interés. Vamos a estudiar el caso de árboles generales (el caso n-ario es parecido).

[5] La descripción formal es similar a los árboles generales y queda como ejercicio para el lector.

© Los autores, 1998; © Edicions UPC, 1998.

universo ÁRBOL_BINARIO (ELEM) es
  usa BOOL
  tipo árbol
  ops
    crea: → árbol
    enraiza: árbol elem árbol → árbol
    izq, der: árbol → árbol
    raíz: árbol → elem
    vacío?: árbol → bool
  errores izq(crea); der(crea); raíz(crea)
  ecns ∀a,a_1,a_2∈ árbol; ∀v∈ elem
    izq(enraiza(a_1, v, a_2)) = a_1
    der(enraiza(a_1, v, a_2)) = a_2
    raíz(enraiza(a_1, v, a_2)) = v
    vacío?(crea) = cierto; vacío?(enraiza(a_1, v, a_2)) = falso
funiverso

Fig. 5.7: especificación del TAD de los árboles binarios.

El modelo de este tipo de árboles ha de recoger la posición del punto de interés y, por tanto, podemos considerarlo un par ordenado <f : N_0* → V, N_0>, donde V es el dominio de las etiquetas y se cumple que el segundo componente, que identifica la posición actual, está dentro del dominio de la función o bien tiene un valor nulo que denota la inexistencia de elemento distinguido. En cuanto a la signatura, el abanico es muy amplio tanto en lo que se refiere a las operaciones de actualización como a las de recorrido, y citamos a continuación algunas posibilidades sin entrar en detalle.

- Actualización. En las inserciones podemos plantearnos dos situaciones: insertar un nodo como hijo i-ésimo del punto de interés, o colgar todo un árbol a partir del punto de interés. En cualquier caso, parece lógico no cambiar el nodo distinguido. En cuanto a las supresiones, se presenta la misma disyuntiva: borrar nodos individualmente (que deberían ser hojas e hijos del nodo distinguido, o bien el mismo nodo distinguido) o bien subárboles enteros. Las diversas opciones no tienen por qué ser mutuamente exclusivas, ya que un mismo universo puede ofrecer los dos tipos de operaciones.
- Recorrido. También distinguimos dos situaciones: la primera consiste en ofrecer un conjunto de operaciones de interés general para mover el punto de interés al hijo i-ésimo, al padre o al hermano izquierdo o derecho. También se puede disponer de diversas operaciones de más alto nivel que incorporen las diversas estrategias de recorrido que se introducen en la sección 5.3. En cualquier caso, las operaciones de recorrido tendrán que permitir que se examinen los n nodos del árbol.

© Los autores, 1998; © Edicions UPC, 1998.

5.2 Implementación

En esta sección se presentan diversas estrategias de implementación de los árboles. Nos centramos en las representaciones más habituales para los modelos binario y general; en [Knu68, pp. 376-388] se presentan algunas variantes adicionales que pueden ser útiles en casos muy concretos. En el último apartado de la sección, se hace una breve referencia a la extensión de binario a n-ario y a los árboles con punto de interés.

5.2.1 Implementación de los árboles binarios

Como en las estructuras lineales, las representaciones de árboles binarios se subdividen en dos grandes familias: encadenadas (por punteros y vectores) y secuenciales.

a) Representación encadenada

En la fig. 5.8 se muestra una representación encadenada por punteros. Hay un simple apuntador de cada nodo a sus dos hijos; si un hijo no existe, el encadenamiento correspondiente vale NULO. Las operaciones quedan de orden constante, mientras que el espacio utilizado para guardar el árbol es lineal respecto al número de nodos que lo forman. El invariante prohíbe que haya más de un encadenamiento apuntando al mismo nodo (lo cual conduciría a una estructura en forma de grafo, v. capítulo 6) mediante el uso de una función auxiliar correcto (necesaria para hacer comprobaciones recursivas siguiendo la estructura del árbol) que se define sobre otra, nodos, que obtiene el conjunto de punteros a nodos de un árbol, similar a la operación cadena ya conocida (v. fig. 3.23).

Esta representación presenta una característica importante. Recordemos que, tal como se expuso en el apartado 3.3.4, la asignación a_3 := enraiza(a_1, v, a_2) es en realidad una abreviatura de duplica(enraiza(a_1, v, a_2), a_3), por lo que la instrucción tiene un coste lineal debido a la duplicación del árbol. Para evitar dichas duplicaciones, es necesario codificar la función en forma de acción definiendo, por ejemplo, dos parámetros de entrada de tipo árbol y uno de salida que almacene el resultado (v. fig. 5.9, arriba). Ahora bien, con esta estrategia, varios árboles pueden compartir físicamente algunos o todos los nodos que los forman, de manera que la modificación de uno de los árboles tenga como efecto lateral la modificación de los otros. Por ejemplo, dados los árboles a_1, a_2 y a_3 y dada la invocación enraiza(a_1, v, a_2, a_3), es evidente que a_1 y el subárbol izquierdo de a_3 apuntan al mismo lugar, situación en la que, si se ejecuta destruye(a_1), se está modificando el valor del árbol a_3, lo cual probablemente sea incorrecto[6]. Si se considera que esta peculiaridad del funcionamento de la implementación de los árboles binarios es perniciosa, es necesario duplicar los árboles a enraizar, ya sea desde el programa que usa el TAD, ya sea sustituyendo la simple asignación

[6] En realidad, éste no es un problema exclusivo de los árboles, sino en general de las representaciones por punteros. No lo hemos encontrado antes simplemente por la signatura de las operaciones definidas sobre secuencias y tablas. Por ejemplo, si considerásemos la operación de concatenación de dos listas, surgiría la misma problemática.

© Los autores, 1998; © Edicions UPC, 1998.

```
universo ARBOL_BINARIO_ENC_PUNTEROS (ELEM) es
    implementa ARBOL_BINARIO (ELEM)
    usa BOOL
    tipo árbol es ^nodo ftipo
    tipo privado nodo es
        tupla
            etiq es elem; hizq, hder son ^nodo
        ftupla
    ftipo
    invariante (a es árbol): correcto(a), donde correcto: ^nodo → bool se define:
            correcto(NULO) = cierto
            p ≠ NULO ⇒ correcto(p) = correcto(p^.hizq) ∧ correcto(p^.hder) ∧
                                    nodos(p^.hizq) ∩ nodos(p^.hder) = {NULO} ∧
                                    p ∉ nodos(p^.hizq) ∪ nodos(p^.hder)
        y donde nodos: ^nodo → P(^nodo) se define como:
            nodos(NULO) = {NULO}
            p ≠ NULO ⇒ nodos(p) = {p} ∪ nodos(p^.hizq) ∪ nodos(p^.hder)
    función crea devuelve árbol es
    devuelve NULO
    función enraiza (a1 es árbol; v es elem; a2 es árbol) devuelve árbol es
    var p es ^nodo fvar
        p := obtener_espacio
        si p = NULO entonces error si no p^.v := v; p^.hizq := a1; p^.hder := a2 fsi
    devuelve p
    función izq (a es árbol) devuelve árbol es
        si a = NULO entonces error si no a := a^.hizq fsi
    devuelve a
    función der (a es árbol) devuelve árbol es
        si a = NULO entonces error si no a := a^.hder fsi
    devuelve a
    función raíz (a es árbol) devuelve elem es
    var v es elem fvar
        si a = NULO entonces error si no v := a^.v fsi
    devuelve v
    función vacío? (a es árbol) devuelve bool es
    devuelve a = NULO
funiverso
```

Fig. 5.8: implementación por punteros del TAD de los árboles binarios.

© Los autores, 1998; © Edicions UPC, 1998.

de punteros de enraiza, izq y der por duplicaciones de los árboles implicados (v. fig. 5.9, abajo). Obviamente, la segunda alternativa resulta en un coste lineal incondicional de las operaciones que duplican árboles. Una opción diferente consistiría en poner a NULO los apuntadores a árboles que actúen como parámetros de entrada (que dejarían de ser sólo de entrada, obviamente) para evitar manipulaciones posteriores.

```
acción enraiza (ent a1 es árbol; ent v es elem; ent a2 es árbol; sal a3 es árbol) es
    a3 := obtener_espacio
    si a3 = NULO entonces error
    si no a3^.v := v; a3^.hizq := a1; a3^.hder := a2
    fsi
facción

acción enraiza (ent a1 es árbol; ent v es elem; ent a2 es árbol; sal a3 es árbol) es
var p es ^nodo fvar
    a3 := obtener_espacio
    si a3 = NULO entonces error
    si no a3^.v := v; duplica(a1, a3^.hizq); duplica(a2, a3^.hder)
    fsi
facción
```

Fig. 5.9: implementación mediante una acción de enraiza, sin duplicación (arriba) y con duplicación (abajo) de los árboles.

La representación sin copia presenta un riesgo añadido: la posible generación de basura. Este problema aparece porque las acciones izq y dr no aliberan espacio, de manera que al ejecutar izq(a, a) o dr(a, a) el espacio del árbol que no aparece en el resultado quedará inaccessible si no hay otros apuntadores a él. Si, incluso con este problema, se quieren mantener las operaciones de coste constante, será necesario controlar la situación desde el programa que usa los árboles.

La representación encadenada con vectores sigue una estrategia similar. Ahora bien, en el caso de representar cada árbol en un vector, tal como se muestra en la fig. 5.10, las operaciones de enraizar y obtener los subárboles izquierdo y derecho exigen duplicar todos los nodos del resultado para pasar de un vector a otro y, por ello, quedan lineales sobre el número de nodos del árbol[7]. Para evitar este coste se debe permitir que diferentes árboles compartan los nodos y, así, usar un único vector que los almacene todos y que gestione las posiciones libres en forma de pila, de manera que los árboles serán apuntadores (de tipo entero o natural) a la raíz que reside en el vector.

[7] El coste será lineal aunque se transformen las funciones en acciones con parámetros de entrada y de salida según las reglas del apartado 3.3.4.

© Los autores, 1998; © Edicions UPC, 1998.

```
tipo árbol es tupla
                A es vector [de 0 a máx-1] de nodo
                sl es nat
            ftupla
ftipo

tipo privado nodo es tupla etiq es elem; hizq, hder son entero ftupla ftipo
```

Fig. 5.10: representación encadenada de los árboles usando un vector para cada árbol.

En la fig. 5.12 se presenta una implementación posible usando un único vector para todos los árboles. Notemos que es necesario introducir el vector compartido como variable global y externa al universo de definición de los árboles, dado que es una estructura que utilizarán muchos árboles diferentes y que no es propiedad de uno solo. Esta solución viola todas las reglas de modularidad dadas hasta ahora y se puede pensar que es totalmente descartable. No obstante, es necesario tener en cuenta que el vector desempeña exactamente el mismo papel que la memoria dinámica en el caso de los punteros; la única diferencia es que en el segundo caso la memoria (que también es un objeto global y externo a cualquier módulo que lo usa) es un objeto anónimo y implícitamente referible desde cualquier punto de un programa. Una opción diferente (descartada por los motivos que se exponen más adelante) consiste en declarar el vector como parámetro de las diferentes funciones de los árboles de manera que se encaje el esquema dentro del método de desarrollo modular habitual. Ahora bien, como contrapartida, la signatura de las operaciones es diferente en la especificación y en la implementación y, por tanto, se viola el principio de transparencia de la representación, porque un universo que use esta implementación ha de seguir unas convenciones totalmente particulares y no extrapolables a cualquier otra situación. Notemos, eso sí, que el segundo enfoque permite disponer de más de un vector para almacenar árboles.

El invariante de la memoria es exhaustivo. En la primera línea se afirma que todo árbol individual que reside en la memoria está bien formado (en el mismo sentido que la representación de la fig. 5.8) y no incluye ninguna posición de la pila de sitios libres; a continuación, se asegura que toda posición del vector está dentro de algún árbol, o bien en la pila de sitios libres y, por último, se impide la compartición de nodos entre los diferentes árboles. Para escribir comódamente el predicado se usan diversas funciones auxiliares: las ya conocidas correcto y nodos, simplemente adaptadas a la notación vectorial; una función cadena_der que devuelve la cadena de posiciones que cuelga a partir de una posición por los encadenamientos derechos (que se usan en las posiciones libres para formar la pila), y una cuarta función, raíces, que devuelve el conjunto de posiciones del vector que son raíces de árboles, caracterizadas por el hecho de que no son apuntadas por ninguna otra posición ni están en la pila de sitios libres.

En la implementación se incluye también una rutina de inicialización de la memoria que ha de

© Los autores, 1998; © Edicions UPC, 1998.

ser invocada antes de cualquier manipulación y que se limita a crear la pila de sitios libres. Además, se puede incluir otra función para averiguar si queda espacio en el vector.

```
tipo memoria es tupla
                A es vector [de 0 a máx-1] de nodo
                sl es nat
               ftupla
ftipo

tipo privado nodo es tupla etiq es elem; hizq, hder son entero ftupla ftipo

invariante (M es memoria):
    ∀i: i∈raíces(M): correcto(M, i)  ∧  nodos(M, i) ∩ cadena_der(M, M.sl) = {-1}  ∧
    { ∪i: i∈raíces(M): nodos(M, i) }  ∪  cadena_der(M, M.sl) = [-1, máx-1]  ∧
    { ∩i: i∈raíces(M): nodos(M, i) } = {-1}
donde correcto: memoria entero → bool se define:
    correcto(M, -1) = cierto
    i≠-1 ⇒ correcto(M, i) = correcto(M, M[i].hizq) ∧ correcto(M, M[i].hder) ∧
                            nodos(M, M[i].hizq) ∩ nodos(M, M[i].hder) = {-1} ∧
                            i ∉ nodos(M, M[i].hizq) ∩ nodos(M, M[i].hder),
donde nodos: memoria entero → P(entero) se define como:
    nodos(M, -1) = {-1}
    i≠-1 ⇒ nodos(M, i) = {i}  ∪ nodos(M, M[i].hizq) ∪ nodos(M, M[i].hder),
donde cadena_der: memoria entero → P(entero) se define como:
    cadena_der(M, -1) = {-1}
    i≠-1 ⇒ cadena_der(M, i) = {i}  ∪ cadena_der(M, M[i].hder)
y donde raíces: memoria → P(entero) se define como:
    raíces(M) = {i∈[0, máx-1]-nodos(M, M.sl)  /
                 ¬( ∃j: j∈[0, máx-1]-nodos(M, M.sl): M.A[j].hizq = i ∨ M.A[j].hder = i)}

{inicializa(M): acción que es necesario invocar una única vez antes de crear
  cualquier árbol que use la memoria M, para crear la pila de sitios libres
     P ≅ cierto
     Q ≅ cadena_der(M, M.sl) = [-1, máx-1]    }
acción inicializa (sal M es memoria) es
var i es nat fvar
      {simplemente, se forma la pila de sitios libres usando el campo hder}
    para todo i desde 0 hasta máx-2 hacer M.A[i].hder := i+1 fpara todo
    M.A[máx-1].hder := -1; M.sl := 0
facción
```

(a) definición de la memoria y rutina de inicialización.

Fig. 5.12: implementación encadenada por vector compartido de los árboles binarios.

© Los autores, 1998; © Edicions UPC, 1998.

```
universo ÁRBOL_BINARIO_ENC_1_VECTOR (ELEM) es
    implementa ÁRBOL_BINARIO (ELEM)
    usa ENTERO, NAT, BOOL
    tipo árbol es entero ftipo
    invariante (a es árbol): -1 ≤ a ≤ máx-1
    función crea devuelve árbol es
    devuelve -1
    función enraiza (a1 es árbol; v es elem; a2 es árbol) devuelve árbol es
    var i es entero fvar
        si M.sl = -1 entonces error    {el vector es lleno}
        si no {se obtiene un sitio libre y se deposita el nuevo nodo en él}
            i := M.sl; M.sl := M.A[M.sl].hder
            M.A[i]  := <v, a1, a2>
        fsi
    devuelve i
    función izq (a es árbol) devuelve árbol es
        si a = -1 entonces error   {árbol vacío}
        si no a := M[a].hizq
        fsi
    devuelve a
    función der (a es árbol) devuelve árbol es
        si a = -1 entonces error   {árbol vacío}
        si no a := M[a].hder
        fsi
    devuelve a
    función raíz (a es árbol) devuelve elem es
    var v es elem fvar
        si a = -1 entonces error   {árbol vacío}
        si no v := M[a].v
        fsi
    devuelve v
    función vacío? (a es árbol) devuelve bool es
    devuelve a = -1
funiverso
```

(b) universo de los árboles.

Fig. 5.12: implementación encadenada por vector compartido de los árboles binarios (cont.).

© Los autores, 1998; © Edicions UPC, 1998.

b) Representación secuencial

Siguiendo la filosofía de la implementación de estructuras lineales, una representación secuencial ha de almacenar los nodos del árbol sin usar campos adicionales de encadenamientos. Ahora bien, así como en las secuencias había una ordenación clara de los elementos, en los árboles no ocurre lo mismo. Por ejemplo, en el árbol de la fig. 5.3, ¿qué relación se puede establecer entre los nodos <2, febrero> y <332, octubre>? Intuitivamente, podríamos intentar guardar todos los nodos en el orden lexicográfico dado por las secuencias del dominio del árbol. Sin embargo, esta estrategia es totalmente ambigua porque una única configuración del vector se puede corresponder con diversos árboles (v. fig. 5.11). Otra posibilidad consiste en deducir una fórmula que asigne a cada posible nodo del árbol una posición dentro del vector; en el caso de nodos que no existan, la posición correspondiente del vector estará marcada como desocupada. Estudiemos este enfoque.

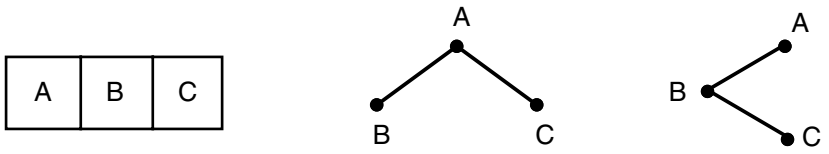


Fig. 5.11: una distribución de elementos dentro de un vector (a la izquierda) y dos posibles árboles que se pueden asociar según el orden lexicográfico (al medio y a la derecha).

La estrategia adoptada consiste en usar la cadena que identifica la posición del nodo, de manera que la raíz vaya a la primera posición, el hijo izquierdo de la raíz a la segunda, el hijo derecho de la raíz a la tercera, el hijo izquierdo del hijo izquierdo de la raíz a la cuarta, etc. Concretamente, dado un nodo $n = <s, v>$ ubicado en el nivel $k+1$ del árbol, $s = s_1...s_k$, se puede definir la función Ψ que determina la posición del vector a la que va a parar cada uno de los nodos del árbol como:

$$\Psi(s) = (1+\sum_{i=0}^{k-1} 2^i) + \sum_{i=1}^{k} (s_i - 1).2^{k-i}$$

El primer factor da la posición que ocupa el primer nodo de nivel k+1-ésimo dentro del árbol[8] y cada uno de los sumandos del segundo término calcula el desplazamiento producido en cada nivel.

A partir de esta fórmula se pueden derivar las diversas relaciones entre la posición que ocupa un nodo y la que ocupan sus hijos, hermano y padre, y que se muestran a continuación. Su cálculo por inducción queda como ejercicio para el lector, pero podemos comprobar su validez en el ejemplo dado en la fig. 5.13 de forma intuitiva.

[8] Se puede demostrar por inducción que el número máximo de nodos en el nivel i-ésimo es 2^{i-1}.

© Los autores, 1998; © Edicions UPC, 1998.

- Hijos: sea el nodo n = <s, v> tal que $\Psi(s) = i$; su hijo izquierdo $n_1 = <s.1, v_1>$ ocupa la posición $\Psi(s.1) = 2i$ y su hijo derecho $n_2 = <s.2, v_2>$ la posición $\Psi(s.2) = 2i+1$.
- Padre: simétricamente, dado el nodo n = <s.k, v> tal que $k \in [1, 2]$ y $\Psi(sk) = i$, $i > 1$, la posición que ocupa su padre <s, v'> es $\Psi(s) = \lfloor i/2 \rfloor$.
- Hermanos: dado que el nodo n = <s, v> es hijo izquierdo si $s = \alpha.1$, e hijo derecho si $s = \alpha.2$, y que en el primer caso la función $\Psi(s)$ da un resultado par y en el segundo uno impar, se cumple que, si $\Psi(s) = 2i$, el hermano derecho de n ocupa la posición 2i +1, y si $\Psi(s) = 2i+1$, el hermano izquierdo de n ocupa la posición 2i.

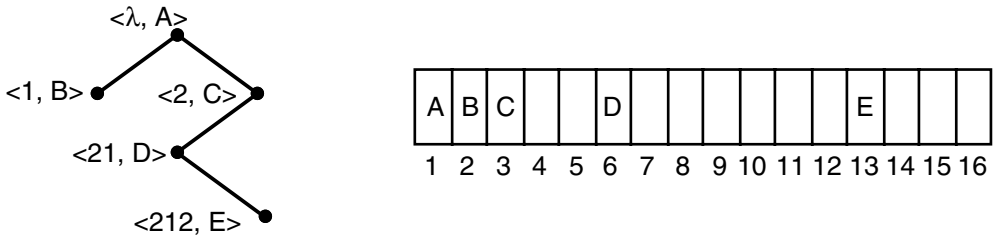


Fig. 5.13: un árbol binario (izquierda) y su representación secuencial (derecha); de cada nodo se da su posición y etiqueta; las posiciones del vector sin etiquetas están vacías.

La representación secuencial aquí presentada se denomina montículo (ing., heap) y exige que cada árbol se implemente con un único vector, de manera que todas las operaciones de enraizar y obtener los subárboles izquierdo y derecho queden lineales sobre el número de nodos implicados. Por este motivo, su utilidad principal surge en los árboles con punto de interés con operaciones individuales sobre nodos. No obstante, incluso en este caso es necesario estudiar cuidadosamente si el espacio para almacenar el árbol es realmente menor que usando encadenamientos: si el árbol tiene pocos nodos pero muchos niveles, probablemente el espacio que se ahorra por la ausencia de apuntadores no compensa las posiciones del vector desocupadas, necesarias para guardar los nodos de los niveles inferiores en la posición dada por Ψ. El caso ideal se da al almacenar árboles completos: un árbol es completo (ing., full) cuando sus niveles contienen todos los nodos posibles; también es bueno el caso de árboles casi completos, que son aquellos que no presentan ninguna discontinuidad en su dominio considerado en orden lexicográfico (el árbol completo es un caso particular de casi completo). Si se puede conjeturar el número de nodos, en ambos casos se aprovechan todas las posiciones del vector; si sólo se puede conjeturar el número de niveles, el aprovechamiento será óptimo en el caso de árboles completos, pero habrá algunas posiciones libres (en la parte derecha del vector) en los árboles casi completos.

© Los autores, 1998; © Edicions UPC, 1998.

5.2.2 Implementación de los árboles generales

Por lo que respecta a la representación encadenada, independientemente de si usamos punteros o vectores, la primera opción puede ser fijar la aridad del árbol y reservar tantos campos de encadenamiento como valga ésta. Esta estrategia, no obstante, presenta el inconveniente de acotar el número de hijos de los nodos (cosa a veces imposible) y, además, incluso cuando esto es posible, el espacio resultante es infrautilizado. Veamos porqué.

Sea n la aridad de un árbol general a, N_a^i el conjunto de nodos de a que tienen i hijos, y X e Y el número de encadenamientos nulos y no nulos de a, respectivamente. Obviamente, se cumplen:

$$X = \Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i|| \ . \ (n\text{-}i) \quad (5.1)$$

$$||N_a|| = \Sigma i: 0 \le i \le n: \ ||N_a^i|| \quad (5.2)$$

$$X + Y = n \ . \ ||N_a|| \quad (5.3)$$

Ahora, sea B el número de ramas que hay en a. Como que en cada nodo del árbol, exceptuando la raíz, llega una y sólo una rama, queda claro que:

$$||N_a|| = B + 1 \quad (5.4)$$

y como que de un nodo con i hijos salen i ramas, entonces:

$$B = \Sigma i: 1 \le i \le n: \ ||N_a^i|| \ . \ i \quad (5.5)$$

Combinando (5.4) i (5.5), obtenemos:

$$||N_a|| = \Sigma i: 1 \le i \le n: \ ||N_a^i|| \ . \ i + 1 \quad (5.6)$$

A continuación, desarrollamos la parte derecha de (5.1) desdoblando el sumatorio en dos, manipulando sus cotas y usando (5.2) y (5.6) convenientemente:

$$\Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i|| \ . \ (n\text{-}i) =$$

$$(\Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i|| \ . \ n) - (\Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i|| \ . \ i) =$$

$$(n \ . \ \Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i||) - (\Sigma i: 1 \le i \le n\text{-}1: \ ||N_a^i|| \ . \ i) =$$

$$(n \ . \ \Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i||) - (\Sigma i: 1 \le i \le n: \ ||N_a^i|| \ . \ i + \ ||N_a^n|| \ . \ n) =$$

$$(n \ . \ \Sigma i: 0 \le i \le n\text{-}1: \ ||N_a^i|| + \ ||N_a^n|| \ . \ n) - (\Sigma i: 1 \le i \le n: \ ||N_a^i|| \ . \ i) =$$

$$(n \ . \ \Sigma i: 0 \le i \le n: \ ||N_a^i||) - (\Sigma i: 1 \le i \le n: \ ||N_a^i|| \ . \ i) = \quad \text{(usando (5.2) y (5.6))}$$

$$(n \ . \ ||N_a||) - (||N_a|| + 1) = (n\text{-}1) \ . \ ||N_a|| + 1$$

Resumiendo, hemos expresado el número X de encadenamientos nulos en función del número total de nodos, $X = (n\text{-}1) \ . \ ||N_a|| + 1$, y usando (5.3) podemos comprobar que X es mayor que el número de encadenamientos no nulos en un factor n-1, de manera que la representación encadenada propuesta desaprovecha mucho espacio.

© Los autores, 1998; © Edicions UPC, 1998.

La solución intuitiva a este problema consiste en crear, para cada nodo, una lista de apuntadores a sus hijos. Desde el nodo se accede a la celda que contiene el apuntador al primer hijo y, desde cada celda, se puede acceder a la celda que contiene el apuntador al hijo correspondiente y a la que contiene el apuntador al hermano derecho. Ahora bien, esta solución es claramente ineficiente porque introduce muchos apuntadores adicionales. La alternativa más eficiente surge al observar que, dado un número fijo de nodos, mientras más grande es la aridad del árbol que los contiene, mayor es el número de encadenamientos desaprovechados, porque el número total crece mientras que el de ramas no varía. Por esto, buscamos una estrategia que convierta el árbol general en binario, para optimizar el espacio. Tomando como punto de partida la lista de apuntadores a los hijos que acabamos de proponer, se puede modificar encadenando directamente los nodos del árbol de manera que cada nodo tenga dos apuntadores, uno a su primer hijo y otro al hermano derecho. Esta implementación se denomina hijo izquierdo, hermano derecho (ing., leftmost-child, right-sibling) y, rotando el árbol resultante, se puede considerar como un árbol binario tal que el hijo izquierdo del árbol binario represente al hijo izquierdo del árbol general, pero el hijo derecho del árbol binario represente al hermano derecho del árbol general (v. fig. 5.14); por esto, la representación también se denomina representación por árbol binario.

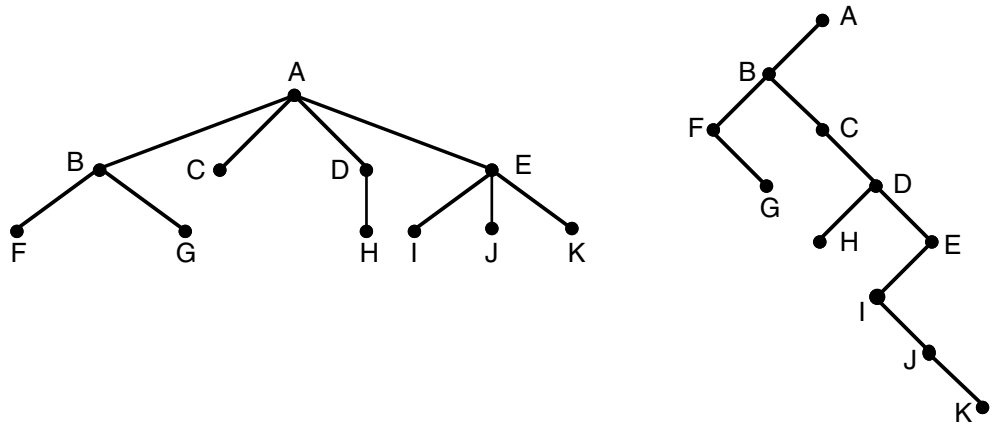


Fig. 5.14: un árbol general (izquierda) y su representación por árbol binario (derecha).

Notemos que el hijo derecho de la raíz será siempre el árbol vacío, porque la raíz no tiene hermano, por definición, de modo que se puede aprovechar para encadenar todos los árboles que forman un bosque y, así, un bosque también tendrá la apariencia de árbol binario, tal como se muestra en la fig. 5.15.

© Los autores, 1998; © Edicions UPC, 1998.

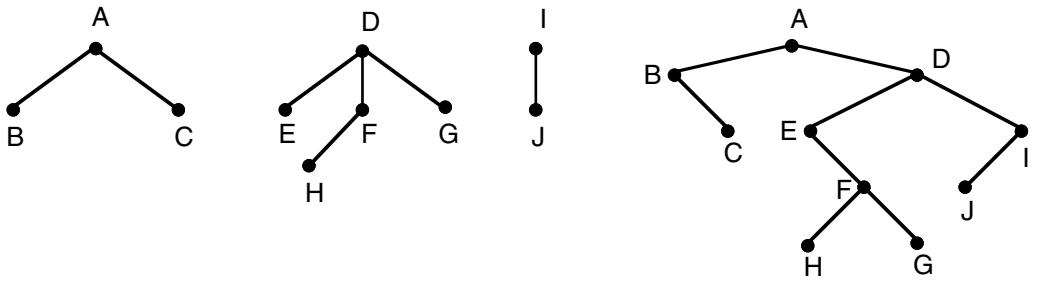


Fig. 5.15: un bosque con tres árboles (izq.) y su representación por árbol binario (der.).

En la fig. 5.16 se muestra la implementación de esta estrategia por punteros. El bosque se representa como una cola de árboles encadenados, dado que se necesitan apuntadores al primer y último árboles; ahora bien, no se efectúa una instancia de las colas para optimizar el espacio y aprovechar el encadenamiento a hermano derecho tal como se ha explicado. En la cola se podría poner un elemento fantasma para no distinguir el caso de cola vacía y también un contador, pero no se hace porque probablemente las colas serán cortas en el caso general; de lo contrario, sería necesario reconsiderar ambas decisiones. En lo que respecta al invariante, es necesario dar propiedades para ambos géneros; como los bosques se pueden considerar como árboles, ambos se ayudan de un predicado auxiliar correcto: ^nodo $\rightarrow$ bool, que comprueba las propiedades habituales de los árboles binarios. En los árboles no se puede decir que el encadenamiento derecho sea nulo, porque un árbol que pase a formar parte del bosque no lo cumple.

La representación secuencial de los árboles generales sigue la misma idea que el caso binario, pero comenzando a numerar por 0 y no por 1 para simplificar las fórmulas de obtención del padre y de los hijos. Es necesario, antes que nada, limitar a un máximo r la aridad mayor permitida en un nodo y también el número máximo de niveles (para poder dimensionar el vector). Entonces, definimos Ψ_r como la función que asigna a cada nodo una posición en el vector tal que, dado $s = s_1 \ldots s_k$:

$$\Psi_r(s) = \sum_{i=0}^{k-1} r^i + \sum_{i=1}^{k} (s_i - 1).r^{k-i}$$

El resto de fórmulas son también una extensión del caso binario y su deducción queda igualmente como ejercicio:

- Hijos: sea el nodo n = <s, v> tal que $\Psi_r(s) = i$; la posición que ocupa su hijo k-ésimo n'=<s.k, v'> es $\Psi_r(s.k) = ir+k$. El primer sumando representa la posición anterior al primer hijo de n, y el sumando k es el desplazamiento dentro de los hermanos.
- Padre: simétricamente, dado el nodo n = <s, v> tal que $\Psi_r(s) = i$, $i > 1$, la posición que ocupa su padre es $\lfloor (i-1) / r \rfloor$.

© Los autores, 1998; © Edicions UPC, 1998.

- Hermanos: sólo es necesario sumar o restar uno a la posición del nodo en cuestión, según se quiera obtener el hermano derecho o izquierdo, respectivamente.

5.2.3 Variaciones en los otros modelos de árboles

Por lo que respecta a la extensión de los árboles binarios a los árboles n-arios, en la representación encadenada se puede optar por poner tantos campos como aridad tenga el árbol, o bien usar la estrategia hijo izquierdo, hermano derecho. En cuanto a la representación secuencial, no hay ninguna diferencia en las fórmulas del caso general tomando como r la aridad del árbol.

El modelo de punto de interés presenta ciertas peculiaridades respecto al anterior. En la representación encadenada, puede ser útil tener apuntadores al padre para navegar convenientemente dentro del árbol. Opcionalmente, si la representación es por hijo izquierdo, hermano derecho, el apuntador al hermano derecho del hijo de la derecha del todo (que es nulo) se puede reciclar como apuntador al padre y, así, todo nodo puede acceder sin necesidad de utilizar más espacio adicional que un campo booleano de marca (en caso de usar punteros), a costa de un recorrido que probablemente será rápido. La representación secuencial queda igual y es necesario notar que es precisamente en este modelo donde és más útil ya que, por un lado, el acceso al padre (y, en consecuencia, a cualquier hermano) se consigue sin necesidad de usar más encadenamientos y, por lo tanto, se puede implementar fácilmente cualquier estrategia de recorrido; por el otro, las operaciones individuales sobre nodos se pueden hacer en tiempo constante, porque consisten simplemente en modificar una posición del vector calculada inmediatamente.

5.2.4 Estudio de eficiencia espacial

La cuestión primordial consiste en determinar cuando se comporta mejor una representación encadenada que una secuencial. Un árbol k-ario de n nodos y k campos de encadenamiento ocupa un espacio (k+X)n, siendo X la dimensión de las etiquetas y tomando como unidad el espacio ocupado por un encadenamiento. El mismo árbol representado con la estrategia hijo izquierdo, hermano derecho ocupa sólo (X+2)n; el precio a pagar por este ahorro son las búsquedas (que generalmente serán cortas) de los subárboles y, por ello, generalmente la segunda implementación será preferible a la primera. Si se usa una representación secuencial, el árbol ocupará un espacio que dependerá de su forma: si los árboles con los cuales se trabaja son casi completos habrá pocas posiciones desaprovechadas. En general, dada una aridad máxima r y un número máximo de niveles k, el espacio necesario para una representación secuencial es $(\Sigma i: 1 \leq i \leq k: r^i)X = [(r^k-1) / (r-1)]X$; esta cifra es muy alta incluso para r y k no demasiado grandes. Por ejemplo, un árbol 4-ario de 10 niveles como máximo, exige dimensionar el vector de 1.398.101 posiciones; suponiendo que las etiquetas sean

© Los autores, 1998; © Edicions UPC, 1998.

enteros (caso usual), los árboles deben tener del orden de medio millón de nodos para que la representación secuencial sea mejor. Por ello, sólo se usa con árboles casi completos.

Para ser más precisos, calculamos a continuación cuál es la relación que debe haber entre el número de nodos y el número de niveles para que la representación secuencial realmente supere la encadenada (implementada por punteros). Supongamos que la aridad del árbol es r y fijemos el número k de niveles del árbol. Definimos el factor de carga α de un árbol como el cociente entre el número n de nodos y el número total de nodos que puede haber en un árbol de aridad r y altura k, $\alpha = n(r-1) / (r^k-1)$, que cumple $0 \leq \alpha \leq 1$. Sustituyendo, el espacio de la representación por árbol binario queda $\alpha(X+2)(r^k-1) / (r-1)$; si comparamos este valor con el espacio de la representación secuencial, $[(r^k-1) / (r-1)]X$, puede verse que el factor de carga debe superar X/(X+2) para que la representación secuencial supere la encadenada. Es decir, mientras más grande sea el espacio necesario para almacenar la etiqueta, menos posiciones vacías pueden quedar en el vector propio de la representación secuencial.

```
universo ÁRBOL_GENERAL_POR_BINARIO (ELEM) es
    implementa ÁRBOL_GENERAL (ELEM)
    usa BOOL
    tipo bosque es tupla
                       prim, ult son ^nodo
                   ftupla
    ftipo
    tipo árbol es ^nodo ftipo
    tipo privado nodo es tupla
                           etiq es elem
                           hizq, hder son ^nodo
                       ftupla
    ftipo
    invariante (b es bosque): correcto(b.prim) ∧ b.ult∈ cadena_der(b.prim)
               (a es árbol): a ≠ NULO ∧ correcto(a)
        donde correcto: ^nodo → bool se define como en la fig. 5.8 y
        siendo cadena_der: ^nodo → P(^nodo) como en el caso binario (v. fig. 5.11)

    función crea devuelve bosque es
    devuelve <NULO, NULO>

    función cuántos (b es bosque) devuelve bosque es
    devuelve cuenta(b.prim)
```

Fig. 5.16: implementación hijo izquierdo, hermano derecho de los árboles generales.

© Los autores, 1998; © Edicions UPC, 1998.

```
función inserta (b es bosque; a es árbol) devuelve bosque es
    si b.ult = NULO entonces b.prim := a si no b.ult^.hder := a fsi
    b.ult := a
devuelve b

función i_ésimo (b es bosque; i es nat) devuelve árbol es
devuelve hermano_derecho_i_ésimo(b.prim, i)

función enraiza (b es bosque; v es elem) devuelve árbol es
var p es ^nodo fvar
   p := obtener_espacio
   si p = NULO entonces error  {falta espacio} si no p^ := <v, b.prim, NULO> fsi
devuelve p

{Precondición (asegurada por el invariante): el puntero a no puede ser NULO}
función nhijos (a es árbol) devuelve nat es
devuelve cuenta(a^.hizq)

{Precondición (asegurada por el invariante): el puntero a no puede ser NULO}
función subárbol (a es árbol; i es nat) devuelve árbol es
devuelve hermano_derecho_i_ésimo(a^.hizq, i)

{Precondición (asegurada por el invariante): el puntero a no puede ser NULO}
función raíz (a es árbol) devuelve elem es
devuelve a^.v

{Función auxiliar cuenta(p): devuelve la longitud de la cadena derecha que
 cuelga de p. }
función privada cuenta (p es ^nodo) devuelve nat es
var cnt es nat fvar
    cnt := 0
    mientras p ≠ NULO hacer cnt := cnt + 1; p := p^.hder fmientras
devuelve cnt

{Función auxiliar hermano_derecho_i_ésimo(p, i): avanza i encadenamientos
 a la derecha a partir de p; da error si no hay suficientes encadenamientos}
función priv hermano_derecho_i_ésimo (p es ^nodo; i es nat) devuelve árbol es
var cnt es nat fvar
    cnt := 1
    mientras (p ≠ NULO) ∧ (cnt < i) hacer p := p^.hder; cnt := cnt + 1 fmientras
    si p = NULO entonces error  fsi
devuelve p

funiverso
```

Fig. 5.16: implementación hijo izquierdo, hermano derecho de los árboles generales (cont.).

© Los autores, 1998; © Edicions UPC, 1998.

5.3 Recorridos

Usualmente, los programas que trabajan con árboles necesitan aplicar sistemáticamente un tratamiento a todos sus nodos en un orden dado por la forma del árbol; es lo que se denomina visitar todos los nodos del árbol. Se distinguen dos categorías básicas de recorridos: los que se basan en las relaciones padre-hijo de los nodos, que denominaremos recorridos en profundidad, y los que se basan en la distancia del nodo a la raíz, conocidos como recorridos en anchura o por niveles. Por lo que se refiere a la signatura, de momento los consideramos como funciones recorrido: árbol $\rightarrow$ lista_etiq, donde lista_etiq es el resultado de instanciar las listas con punto de interés con el tipo de las etiquetas.

En esta sección estudiaremos detalladamente el caso de árboles binarios; la extensión al resto de modelos queda como ejercicio para el lector.

5.3.1 Recorridos en profundidad de los árboles binarios

Dado el árbol binario $a \in A^2_V$, se definen tres recorridos en profundidad (v. fig. 5.17 y 5.18):

- Recorrido en preorden (ing., preorder traversal): si a es el árbol vacío, termina el recorrido; si no lo es, primero se visita la raíz de a y, a continuación, se recorren en preorden los subárboles izquierdo y derecho.
- Recorrido en inorden (ing., inorder traversal): si a es el árbol vacío, termina el recorrido; si no lo es, primero se recorre en inorden el subárbol izquierdo de a, a continuación, se visita su raíz y, por último, se recorre en inorden el subárbol derecho de a.
- Recorrido en postorden (ing., postorder traversal): si a es el árbol vacío, termina el recorrido; si no lo es, primero se recorren en postorden sus subárboles izquierdo y derecho y, a continuación, se visita su raíz.

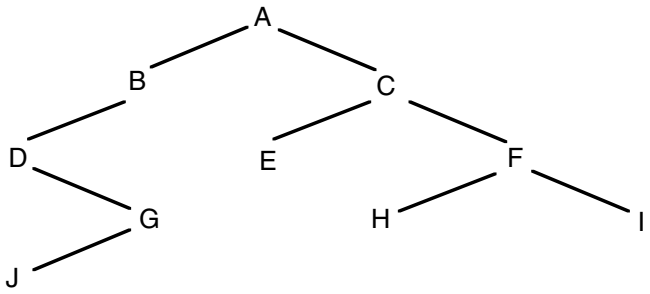
preorden(crea) = crea
preorden(enraiza(a_1, v, a_2)) =
concatena(concatena(inserta(crea, v), preorden(a_1)), preorden(a_2))

inorden(crea) = crea
inorden(enraiza(a_1, v, a_2)) = concatena(inserta(inorden(a_1), v), inorden(a_2))

postorden(crea) = crea
postorden(enraiza(a_1, v, a_2)) = inserta(concatena(postorden(a_1), postorden(a_2)), v)

Fig. 5.17: especificación ecuacional de los recorridos de árboles binarios, donde concatena es la concatenación de dos listas que deja el punto de interés a la derecha del todo.

© Los autores, 1998; © Edicions UPC, 1998.



Recorrido preorden: A, B, D, G, J, C, E, F, H, I
Recorrido inorden: D, J, G, B, A, E, C, H, F, I
Recorrido postorden: J, G, D, B, E, H, I, F, C, A

Fig. 5.18: un árbol binario y sus recorridos preorden, inorden y postorden.

Notemos que el recorrido postorden se puede definir como el inverso especular del recorrido preorden, es decir, para obtener el recorrido postorden de un árbol a se puede recorrer a en preorden, pero visitando siempre el subárbol derecho antes que el izquierdo (especular), considerando la lista de etiquetas resultante como el inverso de la solución. Esta visión será útil posteriormente en la formulación de los algoritmos de recorrido postorden.

Estos tres recorridos se usan en diferentes contextos de la informática. Por ejemplo, el recorrido preorden permite el cálculo de atributos heredados de una gramática. Precisamente, el calificativo "heredados" significa que el hijo de una estructura arborescente hereda (y, posiblemente, modifica) el valor del atributo en el padre. El recorrido inorden de un árbol que cumple ciertas relaciones de orden entre los nodos los obtiene ordenados; en concreto, si todos los nodos del subárbol de la izquierda presentan una etiqueta menor que la raíz y todos los del subárbol de la derecha una etiqueta mayor, y los subárboles cumplen recursivamente esta propiedad, el árbol es un árbol de búsqueda que se caracteriza precisamente por la ordenación de los elementos en un recorrido inorden (v. la sección 5.6). Por último, el recorrido postorden de un árbol que represente una expresión es equivalente a su evaluación; en las hojas residen los valores (que pueden ser literales o variables) y en los nodos intermedios los símbolos de operación que se pueden aplicar sobre los operandos, cuando éstos ya han sido evaluados (i.e., visitados).

La implementación más inmediata de los recorridos consiste en construir funciones recursivas a partir de la especificación, aplicando técnicas de derivación de programas (en este caso, incluso, por traducción directa). Estas funciones pueden provocar problemas en la ejecución aplicadas sobre árboles grandes y con un soporte hardware no demasiado potente, por ello, es bueno disponer también de versiones iterativas, que pueden obtenerse a partir de la aplicación de técnicas de transformación de las versiones recursivas, que se ayudan de una pila que simula los bloques de activación del ordenador durante la ejecución de estas últimas.

© Los autores, 1998; © Edicions UPC, 1998.

En la fig. 5.19 se muestra la transformación de recursivo a iterativo del recorrido en preorden. Se supone que tanto éste como otros recorridos residen en un universo parametrizado por el tipo de las etiquetas, ÁRBOL_BINARIO_CON_RECORRIDOS(ELEM), que enriquece el universo de definición de los árboles binarios, ÁRBOL_BINARIO(ELEM), razón por la que no se accede a la representación del tipo, sino que se utilizan las operaciones que ofrece su especificación. Notemos que los árboles se van guardando dentro de la pila siempre que no estén vacíos, y se empila el hijo izquierdo después del derecho para que salga antes. Queda claro que un nodo siempre se inserta en la lista resultado antes de que se inserten sus descendientes.

```
función preorden (a es árbol) devuelve lista_etiq es
var p es pila_árbol; l es lista_etiq; aaux es árbol  fvar
    p := PILA.crea; l := LISTA_INTERÉS.crea
    si ¬vacío?(a) entonces p := empila(p, a) fsi
    mientras ¬vacía?(p) hacer
        aaux := cima(p); p := desempila(p)   {aaux no puede ser vacío}
        l := inserta(l, raíz(aaux))
        si ¬ vacío?(hder(aaux)) entonces p := empila(p, der(aaux)) fsi
        si ¬ vacío?(hizq(aaux)) entonces p := empila(p, izq(aaux)) fsi
    fmientras
devuelve l
```

Fig. 5.19: implementación del recorrido preorden con la ayuda de una pila.

En lo que respecta a la eficiencia, dado el coste constante de las operaciones sobre pilas y listas, es obvio que el recorrido de un árbol de n nodos es $\Theta(n)$, siempre que las operaciones hizq y hder no dupliquen árboles (parece lógico no hacerlo, porque los árboles no se modifican), puesto que a cada paso se inserta una y sólo una etiqueta en la lista y, además, una misma etiqueta se inserta sólo una vez. En cuanto al espacio, el crecimiento de la pila auxiliar depende de la forma del árbol; como sus elementos serán apuntadores (si realmente estamos trabajando sin copiar los árboles), el espacio total tiene como coste asintótico el número esperado de elementos en la pila[9]. El cálculo de la dimensión de la pila sigue el razonamiento siguiente:

- Inicialmente, hay un único árbol en la pila.
- En la ejecución de un paso del bucle, la longitud de la pila puede:
  - ◊ disminuir en una unidad, si el árbol que se desempila sólo tiene un nodo;
  - ◊ quedar igual, si el árbol que se desempila sólo tiene subárbol izquierdo o derecho;
  - ◊ incrementarse en una unidad, si el árbol que se desempila tiene ambos subárboles.

[9] En realidad esta pila estaba implícita en la versión recursiva y, por ello, no se puede considerar un empeoramiento respecto a ésta.

© Los autores, 1998; © Edicions UPC, 1998.

El caso peor, pues, es el árbol sin nodos de grado uno en el que todo nodo que es hijo derecho es a la vez una hoja. Así, al empilar el árbol formado por el único hijo izquierdo que también es hoja, dentro de la pila habrá este árbol y todos los subárboles derechos encontrados en el camino, cada uno de ellos formado por un único nodo (es decir, lo más pequeños posible); para un árbol de n nodos, la pila puede crecer hasta $\lceil n/2 \rceil$ elementos. En cambio, el caso mejor es el árbol en el que todo nodo tiene exactamente un subárbol hijo de manera que la pila nunca guarda más de un elemento. En el caso de un árbol completo la pila crece hasta $\lceil \log_2 n \rceil$ al visitar las hojas del último nivel.

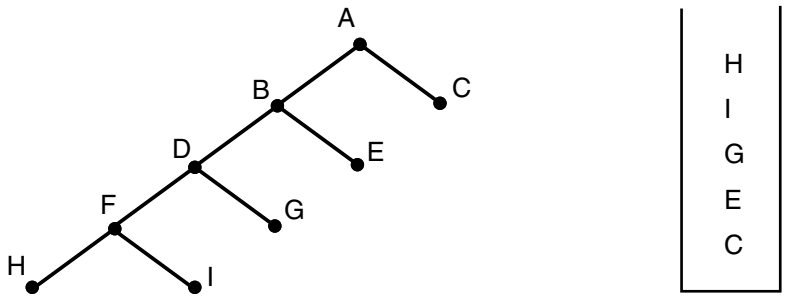


Fig. 5.20: caso peor en el recorrido preorden en cuanto al tamaño de la pila (se identifican los árboles por su raíz dentro de la pila).

El recorrido en postorden sigue el mismo esquema, considerado como un recorrido preorden inverso especular (v. fig. 5.21). Para implementar la noción de especular se empila antes el hijo derecho que el hijo izquierdo, y para obtener la lista en orden inverso se coloca el punto de interés al principio siempre que se inserta un nuevo elemento, de manera que la inserción sea por la izquierda y no por la derecha (se podría haber requerido un modelo de listas con operación de insertar por el principio). De acuerdo con la especificación, al final del recorrido es necesario mover el punto de interés a la derecha del todo, para dejar el resultado en el mismo estado que los otros algoritmos vistos hasta ahora. El coste del algoritmo es totalmente equivalente al recorrido preorden.

Por último, el recorrido en inorden se obtiene fácilmente a partir del algoritmo de la fig. 5.19: cuando se desempila un árbol, su raíz no se inserta directamente en la lista sino que se vuelve a empilar, en forma de árbol con un único nodo, entre los dos hijos, para obtenerla en el momento adecuado, tal como se presenta en la fig. 5.22. Consecuentemente, la pila puede crecer todavía más que en los dos recorridos anteriores. En el caso peor, que es el mismo árbol que el de la fig. 5.20, puede llegar a guardar tantos elementos como nodos tiene el árbol. Además, el algoritmo es más lento, porque en ese caso no todas las iteraciones añadirán nodos a la lista solución (eso sí, no se empeorará el coste asintótico). En el caso peor de la fig. 5.20, primero se empilan n subárboles de un nodo en $\lfloor n/2 \rfloor$ iteraciones y, a continuación, se visitan sus raíces en n vueltas adicionales del bucle.

© Los autores, 1998; © Edicions UPC, 1998.

```
función postorden (a es árbol) devuelve lista_etiq es
var p es pila_árbol; l es lista_etiq; aaux es árbol  fvar
    p := PILA.crea; l := LISTA_INTERÉS.crea
    si ¬vacío?(a) entonces p := empila(p, a) fsi
    mientras ¬vacía?(p) hacer
        aaux := cima(p); p := desempila(p)
        l := principio(inserta(l, raíz(aaux)))
        si ¬ vacío?(hizq(aaux)) entonces p := empila(p, hizq(aaux)) fsi
        si ¬ vacío?(hder(aaux)) entonces p := empila(p, hder(aaux)) fsi
    fmientras
    mientras ¬ final?(l) hacer l := avanza(l) fmientras
devuelve l
```

Fig. 5.21: implementación del recorrido postorden con la ayuda de una pila.

```
función inorden (a es árbol) devuelve lista_etiq es
var p es pila_árbol; l es lista_etiq; aaux es árbol  fvar
    p := PILA.crea; l := LISTA_INTERÉS.crea
    si ¬ vacío?(a) entonces p := empila(p, a) fsi
    mientras ¬ vacía?(p) hacer
        aaux := cima(p); p := desempila(p)  {aaux no puede ser vacío}
        si vacío?(hizq(aaux)) ∧ vacío?(hder(aaux)) entonces
            l := inserta(l, raíz(aaux))   {hay un único nodo, que se visita}
        si no {se  empila la raíz entre los dos hijos}
            si ¬ vacío?(hder(aaux)) entonces p := empila(p, hder(aaux)) fsi
            p := empila(p, enraiza(crea, raíz(aaux), crea)))
            si ¬ vacío?(hizq(aaux)) entonces p := empila(p, hizq(aaux)) fsi
        fsi
    fmientras
devuelve l
```

Fig. 5.22: implementación del recorrido inorden con la ayuda de una pila.

Una variante habitual presente en diversos textos consiste en incluir la visita de los vértices dentro del recorrido, de manera que no sea necesario construir una lista primero y después recorrerla. Evidentemente, el aumento de eficiencia de esta opción choca frontalmente con la modularidad del enfoque anterior, porque es necesario construir un nuevo algoritmo de recorrido para cada tratamiento diferente que surja, a no ser que el lenguaje permitiera funciones de orden superior de manera que el tratamiento de los nodos fuera un parámetro más de los recorridos[10]. Una opción diferente que apunta en el mismo sentido, pero sin

[10] E incluso así hay restricciones, porque sería necesario declarar la signatura de este parámetro y esto limitaría el conjunto de acciones válidas

© Los autores, 1998; © Edicions UPC, 1998.

perder modularidad, es adoptar el modelo de los árboles binarios con punto de interés para obtener los nodos uno a uno según las diferentes estrategias.

5.3.2 Árboles binarios enhebrados

Los árboles enhebrados (ing., threaded tree) son una representación ideada por A.J. Perlis y C. Thornton en el año 1960 (Communications ACM, 3), que permite implementar los recorridos sin necesidad de espacio adicional (como mucho, un par de booleanos de marca en cada nodo), y se basa en la propiedad ya conocida de que una representación encadenada normal de los árboles desaprovecha muchos apuntadores que no apuntan a ningún nodo. En el caso binario, y a partir de la fórmula calculada en el apartado 5.2.2, de los 2n encadenamientos existentes en un árbol de n nodos hay exactamente n+1 sin usar. Por ello, es bueno plantearse si se pueden reciclar estos apuntadores para llevar a cabo cualquier otra misión, y de ahí surge el concepto de hebra (ing., thread): cuando un nodo no tiene hijo derecho, se sustituye el valor nulo del encadenamiento derecho por su sucesor en inorden, y cuando no tiene hijo izquierdo, se sustituye el valor nulo del encadenamiento izquierdo por su predecesor en inorden (v. fig. 5.23); de esta manera, se favorece la implementación de los recorridos sobre el árbol.

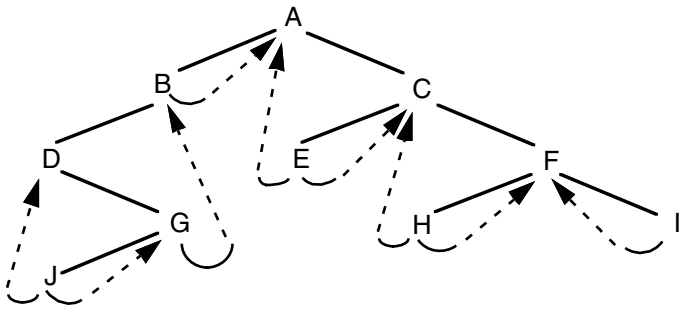


Fig. 5.23: el árbol binario de la fig. 5.18, enhebrado en inorden (las hebras son las rayas discontinuas).

La elección del enhebrado inorden y no de otro se justifica por la fácil implementación de los diferentes recorridos. El punto clave consiste en definir cuál es el primer nodo del árbol que se ha de visitar y como se calcula el siguiente nodo dentro del recorrido:

- Preorden: el primer nodo es la raíz del árbol (si la hay). Dado un nodo n, su sucesor en preorden es el hijo izquierdo, si tiene; si no, el hijo derecho, si tiene. Si es una hoja, es necesario seguir las hebras derechas (que llevan a nodos antecesores de n, ya visitados por definición de recorrido preorden, tales que n forma parte de su subárbol izquierdo) hasta llegar a un nodo que, en lugar de hebra derecha, tenga hijo derecho,

© Los autores, 1998; © Edicions UPC, 1998.

que pasa a ser el sucesor en preorden.

- Inorden: el primer nodo (si lo hay) se localiza bajando recursivamente por la rama más izquierda del árbol. Dado un nodo n (que cumple que los nodos de su árbol izquierdo ya han sido visitados, por definición de recorrido inorden), su sucesor en inorden es el primero en inorden del hijo derecho, si tiene; si no, simplemente se sigue la hebra (que recordamos que apunta precisamente al sucesor en inorden de n).
- Postorden: es inmediato a partir de la definición como preorden inverso especular.

En la fig. 5.26 se presenta la implementación de estos recorridos. Como se está explotando una característica de la representación de los árboles, los algoritmos se incluyen dentro del mismo universo de definición del tipo porque así pueden acceder a ella. En la figura se ha optado por una representación con punteros (queda como ejercicio la representación por vectores). El esquema de los tres recorridos es casi idéntico y sólo cambia la implementación de la estrategia; notemos el uso de un puntero auxiliar que desempeña el papel de elemento actual en el recorrido. El invariante incorpora el concepto de hebra y garantiza, por un lado, que tanto el hijo izquierdo del primero en inorden como el hijo derecho del último en inorden valen NULO y, por otro, que las hebras apuntan a los nodos correctos según la estrategia adoptada. Los predicados correcto, cadena_izq y cadena_der varían respecto a la fig. 5.11, porque el caso trivial deja de ser el puntero NULO y pasa a ser la existencia de hebra. El predicado nodos, en cambio, queda igual, porque la inserción reiterada de algunos nodos no afecta al resultado. Las pre y postcondiciones de las funciones auxiliares, así como los invariantes de los bucles, son inmediatas y quedan como ejercicio para el lector.

Dado un árbol enhebrado de n nodos, su recorrido siguiendo cualquier estrategia queda $\Theta(n)$. En concreto, dado que cada hebra se sigue una única vez y, o bien sólo las hebras de la izquierda, o bien sólo las de la derecha, el número total de accesos adicionales a los nodos del árbol durante el recorrido es $\lfloor (n+1) / 2 \rfloor$.

El último paso consiste en modificar las operaciones habituales de la signatura de los árboles binarios para adaptarlas a la estrategia del enhebrado. En la fig. 5.24 se presenta el esquema general de la operación de enraizar (la obtención de los hijos es la inversa); notemos que el mantenimiento del enhebrado exige buscar el primero y el último en inorden del árbol y, por tanto, la representación clásica de los árboles binarios no asegura el coste constante de las operaciones que, en el caso peor, pueden llegar a ser incluso lineales. Si este inconveniente se considera importante, se pueden añadir a la representación de los árboles dos apuntadores adicionales a estos elementos. En la fig. 5.25 se muestra la nueva representación del tipo y la codificación de enraiza con esta estrategia; sería necesario retocar los algoritmos de la fig. 5.26 para adaptarlos a la nueva situación. Si la formación del árbol es previa a cualquier recorrido y se prevé recorrerlo múltiples veces, se puede optar por mantener la representación no enhebrada durante los enraizamientos y enhebrar el árbol antes del primer recorrido.

© Los autores, 1998; © Edicions UPC, 1998.

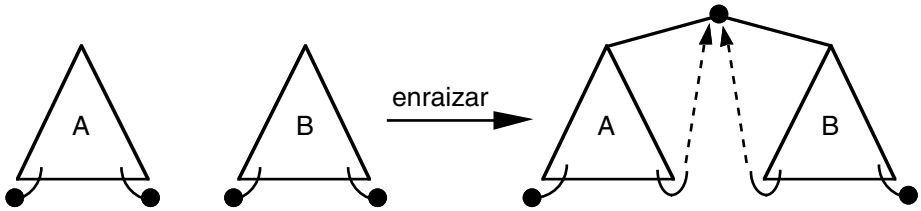


Fig. 5.24: esquema general del enraizamiento de árboles binarios enhebrados inorden.

```
tipo árbol es
    tupla
        prim, ult, raíz son ^nodo
    ftupla
ftipo
tipo privado nodo es
    tupla
        v es etiq
        ∃hizq, ∃hder son bool
        hizq, hder son ^nodo
    ftupla
ftipo
invariante (a es árbol): el mismo que la fig. 5.24 y
        (a.raíz = NULO ⇔ a.prim = NULO) ∧ (a.prim = NULO ⇔ a.ult = NULO) ∧
        a.raíz ≠ NULO ⇒ (a.prim^.hizq = NULO) ∧ (a.ult^.hder = NULO)

función enraiza (a1 es árbol; v es elem; a2 es árbol) devuelve árbol es
var a es árbol fvar
    a.raíz := obtener_espacio
    si a.raíz = NULO entonces error   {no hay espacio}
    si no     {primero enraizamos normalmente}
        a.raíz^.v := v; a.raíz^.hizq := a1.raíz; a.raíz^.hder:= a2.raíz
        a.raíz^.∃hizq := (a1.raíz ≠ NULO); a.raíz^.∃hder := (a2.raíz ≠ NULO)
            {a continuación, se enhebra el árbol en inorden}
        si a1.prim ≠ NUL entonces a.prim := a1.prim si no a.prim := a fsi
        si a2.ult ≠ NUL entonces a.ult := a2.ult si no a.ult := a fsi
        si a1.ult ≠ NULO entonces a1.ult^.hder := a.raíz fsi
        si a2.prim ≠ NULO entonces a2.prim^.hizq := a.raíz fsi
    fsi
devuelve a
```

Fig. 5.25: algoritmo de enraizamiento de árboles binarios enhebrados inorden.

© Los autores, 1998; © Edicions UPC, 1998.

```
tipo árbol es ^nodo ftipo
tipo privado nodo es
    tupla
        v es etiq; hizq, hder son ^nodo
        ∃hizq, ∃hder son bool[11]  {indican si los encadenamientos son hebras}
    ftupla
ftipo
invariante (a es árbol): correcto(a) ∧ ∀p: p∈nodos(a)-{NULO}:
        p^.hizq = NULO ⇔ (p∈cadena_izq(a) ∧ ¬ p^. ∃hizq) ∧
        p^.hder = NULO ⇔ (p∈cadena_der(a) ∧ ¬ p^. ∃hder) ∧
        (¬ p^. ∃hizq ∧ p^.hizq ≠ NULO) ⇒ p∈cadena_izq(p^.hizq^.hder) ∧
        (¬ p^. ∃hder ∧ p^.hder ≠ NULO) ⇒ p∈cadena_der(p^.hder^.hizq)
   donde correcto: ^nodo → bool se define como:
        1) correcto(NULO) = cierto  {caso trivial}
        2) p ≠ NULO ∧ ¬ p^. ∃hizq ∧ ¬ p^. ∃hder ⇒ correcto(p) = cierto  {hoja}
        3) p ≠ NULO ∧ p^. hizq ∧ p^. ∃hder ⇒ correcto(p) =     {tiene los dos hijos}
                correcto(p^.hizq) ∧ correcto(p^.hder) ∧
                nodos(p^.hizq) ∩ nodos(p^.hder) = Ø ∧
                p∉nodos(p^.hizq) ∪ nodos(p^.hder)
        4) p ≠ NULO ∧ ¬ p^. ∃hizq ∧ p^. ∃hder ⇒     {sólo tiene hijo derecho}
                correcto(p) = correcto(p^.hder) ∧ p∉nodos(p^.hder)
        5) p ≠ NULO ∧ p^. ∃hizq ∧ ¬ p^. ∃hder ⇒    {sólo tiene hijo izquierdo}
                correcto(p) = correcto(p^.hizq) ∧ p∉nodos(p^.hizq),
   donde cadena_izq: ^nodo → P(^nodo) se define (y cadena_der simétricamente):
            ¬ p^. ∃hizq ⇒ cadena_izq(p) = Ø
            p^. ∃hizq ⇒ cadena_izq(p) = {p} ∪ cadena_izq(p^.hizq)
    y donde nodos: ^nodo → P(^nodo) se define de la manera habitual

función privada siguiente_preorden (act es árbol) devuelve árbol es
    si act^. ∃hizq entonces act := act^.hizq   {si hay hijo izquierdo, ya lo tenemos}
    si no {es necesario remontar por las hebras hasta llegar a un nodo con hijo derecho}
            mientras ¬ act^. ∃hder hacer act := act^.hder fmientras
            act := act^.hder
    fsi
devuelve act
```

Fig. 5.26: implementación de los recorridos en profundidad con árboles enhebrados.

[11] Estos dos booleanos se pueden obviar en caso de representación por vectores y puede codificarse la existencia o no de hijo con el signo de los encadenamientos: un valor negativo significa que el encadenamiento representa un hijo y es necesario tomar su valor absoluto para acceder a él.

© Los autores, 1998; © Edicions UPC, 1998.

```
función privada primero_inorden (act es ^nodo) devuelve ^nodo es
    si act ≠ NULO entonces {es necesario bajar por la rama izquierda, recursivamente}
        mientras act^. ∃hizq hacer act := act^.hizq fmientras
    fsi
devuelve act

función privada siguiente_inorden (act es ^nodo) devuelve ^nodo es
    si ¬act^. ∃hder entonces act := act^.hder    {se sigue la hebra}
    si no act := primero_inorden(act^.hder)  {busca primero en inorden del hijo derecho}
    fsi
devuelve act

función privada siguiente_preorden_especular (act es ^nodo) devuelve ^nodo es
    si act^. ∃hder entonces act := act^.hder    {si hay hijo derecho, ya lo tenemos}
    si no {es necesario remontar por las hebras hasta llegar a un nodo con hijo izquierdo}
            mientras ¬ act^. ∃hizq hacer act := act^.hizq fmientras
            act := act^.hizq
    fsi
devuelve act

función preorden (a es árbol) devuelve lista_etiq es
var l es lista_etiq; act es ^nodo fvar
    l := LLISTA_INTERÉS.crea; act := a
    mientras act ≠ NULO hacer l := inserta(l, act^.v); act := siguiente_preorden(act) fmientras
devuelve l
```

función inorden (a es árbol) devuelve lista_etiq: como preorden, pero inicializando act al primero inorden, act := primero_inorden(act), y avanzando con siguiente_inorden

función postorden (a es árbol) devuelve lista_etiq: como preorden, pero avanzando con siguiente_preorden_especular, colocando la lista al inicio justo antes de insertar el elemento actual y, al acabar, colocarla al final

Fig. 5.26: implementación de los recorridos en profundidad con árboles enhebrados (cont.).

5.3.3 Recorrido por niveles de los árboles binarios

El recorrido por niveles o por anchura (ing., level order o breadth traversal) de un árbol binario consiste en visitar primero la raíz del árbol, después los nodos que están en el nivel 2, después los que están en el nivel 3, etc., hasta visitar los nodos del último nivel del árbol; para cada nivel, los nodos se visitan de izquierda a derecha. Como resultado, el orden de visita se corresponde con la numeración presentada en el punto 5.2.1 b, ya que se basa en la misma idea. Por ejemplo, en el árbol de la fig. 5.18 se obtiene el recorrido A, B, C, D, E, F, G, H, I, J. Esta política de visita por distancia a la raíz es útil en determinados contextos donde las ramas tienen algún significado especialmente significativo.

© Los autores, 1998; © Edicions UPC, 1998.

La especificación del recorrido por niveles no es tan sencilla como la de los recorridos en profundidad dado que, en este caso, no importa tanto la estructura recursiva del árbol como la distribución de los nodos en los diferentes niveles. En la fig. 5.27 se muestra una propuesta que usa una cola para guardar los subárboles pendientes de recorrer en el orden adecuado. Notemos que la cola mantiene la distribución por niveles de los nodos y obliga a la introducción de una operación auxiliar niveles_con_cola. Otra opción es formar listas de pares etiqueta-nivel que, correctamente combinadas, ordenen los nodos según la estrategia. Sea como sea, el resultado peca de una sobreespecificación causada por la adopción de la semántica inicial como marco de trabajo.

```
niveles(a) = niveles_con_cola(encola(COLA.crea, a))
niveles_con_cola(crea) = LISTA_INTERÉS.crea
[vacío?(cabeza(encola(c, a)))] ⇒
      niveles_con_cola(encola(c, a)) = niveles_con_cola(desencola(encola(c, a)))
[¬vacío?(cabeza(encola(c, a)))] ⇒
      niveles_con_cola(encola(c, a)) =
            concatena(inserta(crea, raíz(cabeza(encola(c, a)))),
                  niveles_con_cola(encola(encola(desencola(encola(c, a),
                                                hizq(cabeza(encola(c, a))),
                                          hder(cabeza(encola(c, a)))))
```

Fig. 5.27: especificación del recorrido por niveles con la ayuda de una cola.

En la fig. 5.28 se presenta la implementación del algoritmo siguiendo la misma estrategia que la especificación, procurando simplemente no encolar árboles vacíos. Observemos la similitud con el algoritmo iterativo del recorrido postorden de la fig. 5.21, cambiando la pila por una cola. Así mismo, su coste asintótico es también $\Theta(n)$ para un árbol de n nodos.

```
función niveles (a es árbol) devuelve lista_etiq es
var c es cola_árbol; l es lista_etiq; aaux es árbol  fvar
    c := COLA.crea; l := LISTA_INTERÉS.crea
    si ¬ vacío?(a) entonces c := encola(p, a) fsi
    mientras ¬ vacía?(c) hacer
        aaux := cabeza(c); c := desencola(c)  {aaux no puede ser vacío}
        l := inserta(l, raíz(aaux))
        si ¬ vacío?(hizq(aaux)) entonces c := encola(c, hizq(aaux)) fsi
        si ¬ vacío?(hder(aaux)) entonces c := encola(c, hder(aaux)) fsi
    fmientras
devuelve l
```

Fig. 5.28: implementación del recorrido por niveles con la ayuda de una cola.

© Los autores, 1998; © Edicions UPC, 1998.

En lo que respecta a la dimensión de la cola, es necesario notar que, al visitar el nodo $n \in N_a$ residente en el nivel k del árbol a, en la cola sólo habrá los subárboles tales que su raíz x esté en el nivel k de a (porque x estará más a la derecha que n dentro de a), o bien en el nivel k+1 de a (porque x será hijo de un nodo ya visitado en el nivel k). Por este motivo, el caso peor es el árbol en que, después de visitar el último nodo del penúltimo nivel, en la cola hay todos los subárboles posibles cuya raíz esté en el último nivel. Es decir, el árbol peor es el árbol completo, donde hay $n = 2^k-1$ nodos dentro del árbol y, así, en el último nivel hay exactamente 2^{k-1} nodos.

5.4 Relaciones de equivalencia

En el resto del capítulo se estudia la aplicación de los árboles en la implementación eficiente de diversos tipos de datos. Para empezar, en esta sección se quieren implementar los conjuntos compuestos de conjuntos de elementos, con la particularidad de que ningún elemento puede aparecer en más de un conjunto componente. A esta clase de conjuntos los denominamos relaciones de equivalencia dado que, normalmente, los conjuntos componentes representan clases formadas a partir de una relación reflexiva, simétrica y transitiva. No obstante, en la literatura del tema los podemos encontrar bajo otras denominaciones: estructuras de partición, MFsets, union-find sets y otras que, aun cuando reflejan las operaciones que ofrecen, no denotan tan fielmente su significado.

Hay diversos modelos posibles para el TAD de las relaciones de equivalencia que se pueden adaptar los unos mejor que los otros a determinados contextos. En esta sección definimos un modelo que asocia un identificador a cada clase de equivalencia para poder referirse a ellas desde otras estructuras externas; para simplificar, consideramos que los identificadores son naturales. Dado V el conjunto base de los elementos de la relación, podemos definir las relaciones de equivalencia R_V como funciones de los elementos a los naturales, $f: V \to N$, siendo f(v) el identificador de la clase a la que pertenece el elemento v, $v \in V$.

Para $R \in R_V$, $v \in V$ e $i_1, i_2 \in N$, definimos las siguientes operaciones sobre el TAD:

- Crear la relación vacía: crea, devuelve la relación tal que cada elemento forma una clase por sí mismo; es decir, devuelve la relación I que cumple: $\forall v,w \in V : v \neq w \Rightarrow I(v) \neq I(w)$.
- Determinar la clase en la que se halla un elemento: clase(R, v), devuelve el natural que identifica la clase de equivalencia dentro de R que contiene v; es decir, devuelve R(v).
- Establecer la congruencia entre dos clases: fusiona(R, i_1, i_2), devuelve la relación resultado de fusionar todos los elementos de las dos clases identificadas por i_1 e i_2 en una misma clase, o deja la relación inalterada si i_1 o i_2 no identifican ninguna clase dentro de la relación, o bien si $i_1 = i_2$; es decir, devuelve la relación R' que cumple:

© Los autores, 1998; © Edicions UPC, 1998.

$$\Diamond \{ \forall v: v \in V: (R(v) = i_1 \vee R(v) = i_2) \Rightarrow R'(v) = i_1 \} \vee$$
$$\{ \forall v: v \in V: (R(v) = i_1 \vee R(v) = i_2) \Rightarrow R'(v) = i_2 \}$$
$$\Diamond \forall v: v \in V: (R(v) \neq i_1 \wedge R(v) \neq i_2) \Rightarrow R'(v) = R(v).$$

- Averiguar el número de clases de la relación: cuántos?(R), devuelve el número de naturales que son imagen de algún elemento, $||\{i / i \in codom(R)\}||$.

Notemos que el modelo no determina cuáles son los identificadores iniciales de las clases, porque no afecta al funcionamiento del tipo, sólo obliga a que sean diferentes entre ellos. Se puede decir lo mismo por lo que se refiere al identificador de la clase resultante de hacer una fusión y, por ello, no se obliga a que sea uno u otro, sino que simplemente se establece que todos los elementos de una de las dos clases fusionadas vayan a parar a la otra.

En la fig. 5.29 se muestra el contexto de uso habitual de este TAD de las relaciones. Primero, se forma la relación con tantas clases como elementos, a continuación, se organiza un bucle que selecciona dos elementos según un criterio dependiente del algoritmo concreto y, si están en clases diferentes, las fusiona, y el proceso se repite hasta que la relación consta de una única clase. En el apartado 6.5.2 se muestra un algoritmo sobre grafos, el algoritmo de Kruskal para el cálculo de árboles de expansión minimales, que se basa en este esquema. Notemos que el número máximo de fusiones que se pueden efectuar sobre una relación es n -1, siendo $n = ||V||$, pues cada fusión junta dos clases en una, por lo que n-1 fusiones conducen a una relación con una única clase. En cambio, el número de ejecuciones de clase y cuántos no puede determinarse con exactitud, ya que el criterio de selección puede generar pares de elementos congruentes un número elevado de veces. Eso sí, como mínimo el algoritmo ejecutará clase 2(n-1) veces y cuántos, n veces. Además, si la selección asegura que no puede generarse el mismo par de elementos más de una vez, la cota superior del número de ejecuciones de estas operaciones es n(n-1) para clase y n(n-1)/2 + 1 para cuántos, porque el número diferente de pares (no ordenados) es $\Sigma k: 1 \leq k \leq n: k$.

```
R := crea
mientras cuántos?(R) > 1 hacer
        seleccionar dos elementos v1 y v2 según cierto criterio
        i1 := clase(R, v1); i2 := clase(R, v2)
        si i1 ≠ i2 entonces R := fusiona(R, i1, i2) fsi
fmientras
```

Fig. 5.29: algoritmo que usa las relaciones de equivalencia.

A veces este modelo de las relaciones de equivalencia no se adapta totalmente a los requerimientos de una aplicación concreta; las variantes más habituales son:

© Los autores, 1998; © Edicions UPC, 1998.

- Considerar la relación como una función parcial, de manera que haya elementos del conjunto de base que no estén en ninguna clase, en cuyo caso, se acostumbra a sustituir la operación crea por otras dos: la primera, para crear la relación "vacía" (es decir, sin ninguna clase), y una segunda para añadir a la relación una nueva clase que contenga un único elemento que se explicita como parámetro. A veces, el identificador de la clase vendrá asignado por el contexto (es decir, será un parámetro más).
- Identificar las clases no por naturales, sino por los elementos que la forman. Según este esquema, la operación clase se sustituye por otra que, dados dos elementos, comprueba si son congruentes o no, mientras que la operación fusiona tiene como parámetros dos elementos que representan las clases que es necesario fusionar.

En la fig. 5.31 se muestra la especificación del tipo releq de las relaciones de equivalencia. Para dotar al TAD de un significado dentro de la semántica inicial (es decir, para construir el álgebra cociente de términos asociada al tipo con la cual se establece un isomorfismo) es necesario resolver las dos indeterminaciones citadas al describir el modelo. Por lo que respecta al identificador de la fusión, se elige arbitrariamente el último parámetro de la función como identificador de la clase resultado. En lo que se refiere a los identificadores iniciales, se toma la opción de numerar las clases de uno al número de elementos. Para poder escribir la especificación con estas convenciones, se requieren diversas operaciones y propiedades sobre los elementos: han de ser un tipo con un orden total definido que permita identificar cuál es el primer elemento, prim, cuál es el último, ult, y cómo se obtiene el sucesor a un elemento dado, suc ; así, se podrán generar todos los elementos en la creación uno tras otro. También se dispondrá de las operaciones de igualdad. Además, para uso futuro, requerimos una constante n que dé el número de elementos dentro del género y una operación de comparación < que representa el orden total. En la fig. 5.30 se muestra el universo de caracterización que responde a esta descripción.

```
universo ELEM_ORDENADO caracteriza
    usa NAT, BOOL
    tipo elem
    ops prim, ult: → elem
        suc: elem → elem
        _=_, _≠_, _<_: elem elem → bool
        n: → nat
    error suc(ult)
    ecns n > 0 = cierto
        ...reflexividad, simetría y transitividad de _=_
        (v ≠ w) = ¬ (v = w)
        ...antireflexividad, antisimetría y transitividad de _<_
    funiverso
```

Fig. 5.30: caracterización de los elementos de las relaciones de equivalencia.

© Los autores, 1998; © Edicions UPC, 1998.

Por lo que respecta a la estrategia de la especificación, se introducen dos operaciones privadas, vacía (que representa la relación sin ninguna clase) e inserta (que añade un elemento a una clase dada, que puede no existir), que forman un conjunto de constructoras generadoras impuras que facilitan considerablemente la especificación. También surgen otras operaciones auxiliares: vacía?, que comprueba si el identificador dado denota una clase existente, y pon_todos, que se encarga de construir una a una las clases iniciales añadiendo los elementos desde el primero hasta el último. Los errores que aparecen se escriben por la aplicación estricta del método general de especificación presentado en la sección 1.3, pero es imposible que se produzcan, dado el uso que se hace de las operaciones.

universo RELACIÓN_DE_EQUIVALENCIA(ELEM_ORDENADO) es
  usa NAT, BOOL
  tipo releq
  ops privada vacía: → releq
    privada inserta: releq elem nat → releq
    crea: → releq
    clase: releq elem → nat
    fusiona: releq nat nat → releq
    cuántos?: releq → nat
    privada vacía?: releq nat → bool
    privada pon_todos: elem nat → releq
  errores ∀r∈ releq; ∀i_1,i_2∈ nat; ∀v∈ elem
    inserta(inserta(r, v, i_1), v, i_2); clase(vacía); [i > n ∨ NAT.ig(i, cero)] ⇒ inserta(R, v, i)
  ecns ∀r∈ releq; ∀i,i_1,i_2∈ nat; ∀v,v_1,v_2∈ elem
    inserta(inserta(r, v_1, i_1), v_2, i_2) = inserta(inserta(r, v_2, i_2), v_1, i_1)
    crea = pon_todos(prim, 1)
    clase(inserta(r, v, i), v) = i
    [$v_1 \neq v_2$] ⇒ clase(inserta(r, v_1, i), v_2) = clase(r, v_2)
    fusiona(vacía, i_1, i_2) = vacía
    [NAT.ig(i, i_1)] ⇒ fusiona(inserta(r, v, i), i_1, i_2) = inserta(fusiona(r, i_1, i_2), v, i_2)
    [¬NAT.ig(i, i_1)] ⇒ fusiona(inserta(r, v, i), i_1, i_2) = inserta(fusiona(r, i_1, i_2), v, i)
    cuántos?(vacía) = 0
    [vacía?(r, i)] ⇒ cuántos?(inserta(r, v, i)) = cuántos?(r) + 1
    [¬ vacía?(r, i)] ⇒ cuántos?(inserta(r, v, i)) = cuántos?(r)
    vacía?(vacía, i) = cierto; vacía?(inserta(c, v, i_1), i_2) = vacía?(c, i_2) ∧ NAT.ig(i_1, i_2)
    pon_todos(ult, n) = inserta(vacía, ult, n)
    [v ≠ ult] ⇒ pon_todos(v, i) = inserta(pon_todos(suc(v), i+1), v, i)
  funiverso

Fig. 5.31: especificación del TAD de las relaciones de equivalencia.

© Los autores, 1998; © Edicions UPC, 1998.

En el resto del apartado nos ocupamos de la implementación del TAD, partiendo de dos representaciones lineales diferentes y llegando a una arborescente que asegura un coste asintótico medio constante en todas las operaciones. Supondremos que el conjunto de base es un tipo escalar que permite indexar directamente un vector[12]; en caso contrario, sería necesario organizar una tabla de dispersión para mantener los costes que aquí se darán o, en el caso que la dispersión no sea aconsejable (v. sección 5.6), cualquier otra estructura que incrementará el coste de las operaciones.

5.4.1 Implementaciones lineales

Dado que el algoritmo de la fig. 5.29 ejecuta más veces clase que fusiona, nos planteamos su optimización. Una representación intuitiva consiste en organizar un vector indexado por los elementos, tal que en cada posición se almacene el identificador de la clase en la que resida el elemento. Así, el coste temporal de clase queda constante (acceso directo a una posición del vector), mientras que fusiona es lineal, porque su ejecución exige modificar el identificador de todos aquellos elementos que cambien de clase mediante un recorrido del vector; el coste de n -1 ejecuciones de fusiona es, pues, $\Theta(n^2)$. Por otro lado, el coste de crea es lineal (suficientemente bueno, puesto que con toda probabilidad sólo se ejecutará una vez) y el de cuántos? será constante si añadimos un contador.

En la fig. 5.32 se muestra una variante de este esquema que puede llegar a mejorar el coste asintótico de fusiona sin perjudicar clase. Por un lado, se encadenan todos los elementos que están en la misma clase; así, al fusionar dos clases no es necesario recorrer todo el vector para buscar los elementos que cambian de clase. Por otro lado, se introduce un nuevo vector indexado por identificadores que da acceso al primer elemento de la cadena de cada clase.

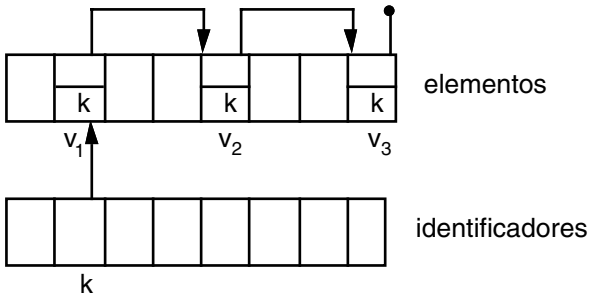


Fig. 5.32: implementación lineal del TAD de las relaciones de equivalencia.

[12] Esta suposición es lógica, dados los requerimientos sobre los elementos establecidos en la especificación.

© Los autores, 1998; © Edicions UPC, 1998.

Está claro que la única operación que varía de coste es fusiona; concretamente, la ejecución de fusiona(R, i_1, i_2) exige modificar sólo las posiciones del vector de elementos que estén dentro de la cadena correspondiente a la clase i_2: cada posición ha de tomar el valor i_1 y, además, el último elemento de la cadena se encadena al primero de la cadena correspondiente a la clase i_1. Es necesario preguntarse, pues, cuántas posiciones pueden formar parte de la cadena asociada a i_2.

Como siempre, buscamos el caso peor. Supongamos que la k-ésima fusión del algoritmo de la fig. 5.29 implica a dos clases i_1 e i_2 de la relación R, tales que en la clase identificada por i_1 sólo hay un elemento, $\|\{v \in V / R(v) = i_1\}\| = 1$, y en la clase identificada por i_2 hay k elementos, $k = \|\{v \in V / R(v) = i_2\}\|$; entonces, el número de posiciones del vector de elementos que es necesario modificar es k. Si esta situación se repite en las n-1 fusiones del algoritmo, $n = \|V\|$, el número total de modificaciones del vector de elementos es $\Sigma k : 1 \le k \le n\text{-}1 : k = n\,(n-1)\,/\,2$, es decir, $\Theta(n^2)$. Dicho de otra manera, el coste asintótico de fusiona después de las n-1 ejecuciones posibles no varía en el caso peor.

La solución es sencilla. Como ya se ha dicho, el identificador de la clase resultante de la fusión no es significativo y, por ello, se puede redefinir de manera que fusiona(R, i_1, i_2) mueva los elementos de la clase más pequeña a la más grande (siempre referido a las clases identificadas por i_1 y i_2). Es decir, siendo $k_1 = \|\{v \in V / R(v) = i_1\}\|$ y $k_2 = \|\{v \in V / R(v) = i_2\}\|$, fusiona(R, i_1, i_2) devuelve la relación R' que cumple[13]:

- Si $k_2 \ge k_1$, entonces $\forall v : v \in V : \{ R(v) = i_1 \Rightarrow R'(v) = i_2 \} \wedge \{ R(v) \neq i_1 \Rightarrow R'(v) = R(v) \}$
- Si $k_2 < k_1$, entonces $\forall v : v \in V : \{ R(v) = i_2 \Rightarrow R'(v) = i_1 \} \wedge \{ R(v) \neq i_2 \Rightarrow R'(v) = R(v) \}$

Según esta estrategia, no hay ningún elemento que cambie de clase más de $\lceil \log_2 n \rceil$ veces después de las n-1 fusiones posibles, puesto que la clase resultante de una fusión tiene una dimensión como mínimo el doble que la más pequeña de las clases que en ella intervienen. Como el coste total de fusiona depende exclusivamente del número de cambios de clase de los elementos (el resto de tareas de la función son comparativamente desdeñables), el coste total de las n-1 fusiones es $\Theta(n \log n)$. Notemos, no obstante, que no se puede asegurar que el coste individual de una fusión sea $\Theta(\log n)$, pero en el caso general (y en el algoritmo de la fig. 5.29 en particular) esta característica no es negativa, porque no será habitual tener fusiones aisladas.

Por lo que respecta a la eficiencia espacial de este esquema, para implementar la nueva estrategia de fusiones es necesario añadir contadores de elementos al vector de los identificadores de las clases. El espacio resultante es, pues, más voluminoso que la primera estructura presentada, aunque el coste asintótico no varía.

En la fig. 5.33 se muestra la implementación del tipo. Se dispone de dos vectores idents y

[13] También se debe adaptar la especificación ecuacional del tipo (queda como ejercicio para el lector).

© Los autores, 1998; © Edicions UPC, 1998.

elems que implementan la estrategia descrita y del típico contador de clases. Se adopta la convención de que, si un identificador no denota ninguna clase válida (porque la clase ha desaparecido en alguna fusión), su contador de elementos vale cero. Esta propiedad es fundamental al establecer el invariante (concretamente, para fijar el valor del contador de clases y para identificar el comienzo de las cadenes válidas dentro del vector de elementos). El final de la cadena correspondiente a una clase de equivalencia se identifica por un elemento que se apunta a sí mismo (de esta manera no es necesario definir un elemento especial que sirva de marca, ni emplear un campo booleano adicional). En el invariante, las cadenas se definen de la manera habitual usando esta convención, y no es necesario exigir explícitamente que la intersección sea vacía, ya que se puede deducir de la fórmula dada. La implementación de las operaciones es inmediata: se introduce una función auxiliar cambia para cambiar todos los elementos de una clase a otra tal como se ha dicho en el modelo.

5.4.2 Implementación arborescente

Las implementaciones lineales que acabamos de ver son suficientemente buenas si no se formulan requerimientos de eficiencia sobre el tipo, o bien si el número de elementos dentro de la relación es pequeño. Igualmente, hay usos del tipo diferentes del esquema general presentado en la fig. 5.29 en los cuales la ineficiencia de fusiona no es tan importante. Por ejemplo, puede que el objetivo final no sea tener una única clase, sino simplemente procesar una colección de equivalencias para agrupar los elementos bajo determinado criterio y, en este caso, seguramente el coste inicial de construir la relación con la aplicación sucesiva de fusiona será irrelevante comparado con el coste posterior de las consultas con clase. Es aconsejable, no obstante, buscar una representación que optimice al máximo el coste del algoritmo de la fig. 5.29, y éste es el objetivo del resto de la sección.

La idea básica para mejorar la fusión consiste en evitar cualquier recorrido de listas, por lo que no todos los elementos guardarán directamente el identificador de la clase a la cual pertenecen y no será preciso tratarlos todos en cada fusión en que intervengan. Para ser más exactos, para cada clase habrá un único elemento, que denominamos representante de la clase, que guardará el identificador correspondiente; los otros elementos que sean congruentes accederán al representante siguiendo una cadena de apuntadores.

El funcionamiento de la representación es el siguiente:

- Al crear la relación, se forman las n clases diferentes; cada elemento es el representante de su clase y guarda el identificador.
- Al fusionar, se encadena el representante de la clase menor con el representante de la clase mayor, el cual conserva su condición de representante de la nueva clase. Será necesario, pues, asegurar el acceso rápido a los representantes de las clases mediante un vector indexado por identificador de clase.

© Los autores, 1998; © Edicions UPC, 1998.

```
universo RELACIÓN_DE_EQUIVALENCIA_LINEAL(ELEM_ORDENADO) es
  implementa RELACIÓN_DE_EQUIVALENCIA(ELEM_ORDENADO)
  usa ENTERO, BOOL
  tipo releq es
    tupla
      elems es vector [elem] de tupla id es nat; enc es elem ftupla
      idents es vector [de 1 a n] de tupla cnt es nat; prim es elem ftupla
      nclases es nat
    ftupla
  ftipo
  invariante (R es releq):
        R.nclases = || { i / R.idents[i].cnt ≠ 0 } || ∧
        ∀i: i∈{ i / R.idents[i].cnt ≠ 0 }:
            R.idents[i].cnt = || cadena(R, R.idents[i].prim) || ∧
            ∀v: v∈cadena(R, R.idents[i].prim): R.elems[v].id = i ∧
        { ∪i: i∈{ k / R.idents[k].cnt ≠ 0}: cadena(R, R.idents[i].prim) } = elem
      donde cadena: releq nat → P(elem) se define:
        R.elems[v].enc = v ⇒ cadena(R, v) = {v}
        R.elems[v].enc ≠ v ⇒ cadena(R, v) = {v} ∪ cadena(R, R.elems[v].enc)
  función crea devuelve releq es
  var R es releq; v es elem; i es nat fvar
    i := 1
    para todo v dentro de elem hacer
        {se forma la clase i únicamente con v}
      R.elems[v].id := i; R.elems[v].enc := v
      R.idents[i].prim := v; R.idents[i].cnt := 1
      i := i + 1
    fpara todo
    R.nclases := n
  devuelve R
  función clase (R es releq; v es elem) devuelve nat es
  devuelve R.elems[v].id
  función cuántos? (R es releq) devuelve nat es
  devuelve R.nclases
```

Fig. 5.33: implementación lineal de las relaciones de equivalencia.

© Los autores, 1998; © Edicions UPC, 1998.

```
función fusiona (R es releq; i1, i2 son nat) devuelve releq es
   si (i1 = 0) ∨ (i1 > n) ∨ (i2 = 0) ∨ (i2 > n) ∨ (R.idents[i1].cnt = 0) ∨ (R.idents[i2].cnt = 0)
      entonces error
      sino si (i1 ≠ i2) entonces
                {se comprueba qué clase tiene más elementos}
                si R.idents[i1].cnt ≥ R.idents[i2].cnt entonces R := cambia(R, i2, i1)
                                                       sino R := cambia(R, i1, i2)
                fsi
                R.nclases := R.nclases - 1   {en cualquier caso desaparece una clase}
             fsi
   fsi
devuelve R

{Función auxiliar cambia: dados dos identificadores de clase, fuente y destino,
 implementa el trasvase de elementos de la primera a la segunda. Como
 precondición, los identificadores denotan dos clases válidas y diferentes
 P ≅ (fuente ≠ destino) ∧ (fuente > 0) ∧ (fuente ≤ n) ∧ (destino > 0) ∧
     (destino ≤ n) ∧ (R.idents[fuente].cnt ≤ R.idents[destino].cnt) ∧ (R = R0)
 Q ≅ ∀i: i∈cadena(R0, R0.idents[fuente].prim): R.elems[i].id = destino ∧
     ∀k: k∈[1, n]-{fuente}: ∀i: i∈cadena(R0, R0.idents[k].prim): R.elems[i].id = k ∧
     R.idents[fuente].cnt = 0}
función privada cambia (R es releq; fuente, destino son nat) devuelve releq es
var v es elem fvar
      {primero cambiamos los identificadores de los elementos de la clase fuente}
   v := R.idents[fuente].prim
   repetir
      {I ≅ ∀i: i∈cadena(R0, R0.idents[fuente].prim)-cadena(R0, R0.idents[v].prim):
                                R.elems[i].id = destino}
      R.elems[v].id := destino; v := R.elems[v].enc
   hasta que R.elems[v].enc = v  {condición de parada: siguiente de v = v}
   R.elems[v].id := destino
      {luego se encadenan las secuencias correspondientes a las dos clases}
   R.elems[v].enc := R.idents[destino].prim
   R.idents[destino].prim := R.idents[fuente].prim
      {por último se actualizan los contadores de elementos}
   R.idents[destino].cnt := R.idents[destino].cnt + R.idents[fuente].cnt
   R.idents[fuente].cnt := 0   {marca de clase vacía}
devuelve R
funiverso
```

Fig. 5.33: implementación lineal de las relaciones de equivalencia (cont.).

© Los autores, 1998; © Edicions UPC, 1998.

- Para consultar el identificador de la clase en la que reside un elemento dado, es necesario buscar su representante siguiendo los encadenamientos.

La repetición del proceso de fusión puede hacer que diversos elementos apunten a un mismo representante, dando lugar a una estructura arborescente. Concretamente, cada clase forma un árbol, donde la raíz es el representante y el resto de elementos apuntan cada uno su padre. Un elemento v es padre de otro elemento w, porque en algún momento se han fusionado dos clases A y B, tales que v era el representante de A, w el representante de B y la clase A tenía más elementos que la clase B.

En la fig. 5.34 se muestra un ejemplo. Se define V = {a, b, ... , h} tal que inicialmente la clase [a] se identifica con el uno, la clase [b] con el dos, etc., y a continuación se realizan diversas fusiones hasta obtener una única clase. En caso de clases igual de grandes, se elige arbitrariamente el identificador de la segunda clase para denotar el resultado. Para mayor claridad, se dibujan directamente los árboles sin mostrar la distribución de los elementos dentro del vector correspondiente. En el vector de identificadores, una cruz denota clase vacía.

En la fig. 5.35 se muestra finalmente la implementación del universo correspondiente. En el vector de elementos, los nodos que no representan ninguna clase se caracterizan por el valor cero en su campo identificador. Igualmente, los naturales que no identifican ninguna clase tienen el contador de elementos a cero en la posición correspondiente del vector de identificadores; opcionalmente, podríamos haber usado tuplas variantes. La codificación de las operaciones es muy parecida: la fusión sigue el mismo esquema pero ahora la función auxiliar da forma arborescente a las clases tal como se ha explicado. Por lo que respecta a clase, hay una búsqueda desde el nodo consultado hasta la raíz del árbol. El invariante es muy similar al caso anterior, usando una función que devuelve el representante de la clase correspondiente a un elemento dado.

La eficiencia espacial es idéntica al enfoque lineal. En cuestión de tiempo, fusiona queda efectivamente constante, mientras que clase pasa a ser función de la profundidad del nodo buscado. Notemos que, cada vez que se hace una fusión, los elementos que cambian de clase aumentan su profundidad en uno; dado que un elemento cambia $\lceil \log_2 n \rceil$ veces de clase como máximo, la altura de los árboles está acotada por este valor y, en consecuencia, la operación queda $\Theta(\log n)$.

© Los autores, 1998; © Edicions UPC, 1998.

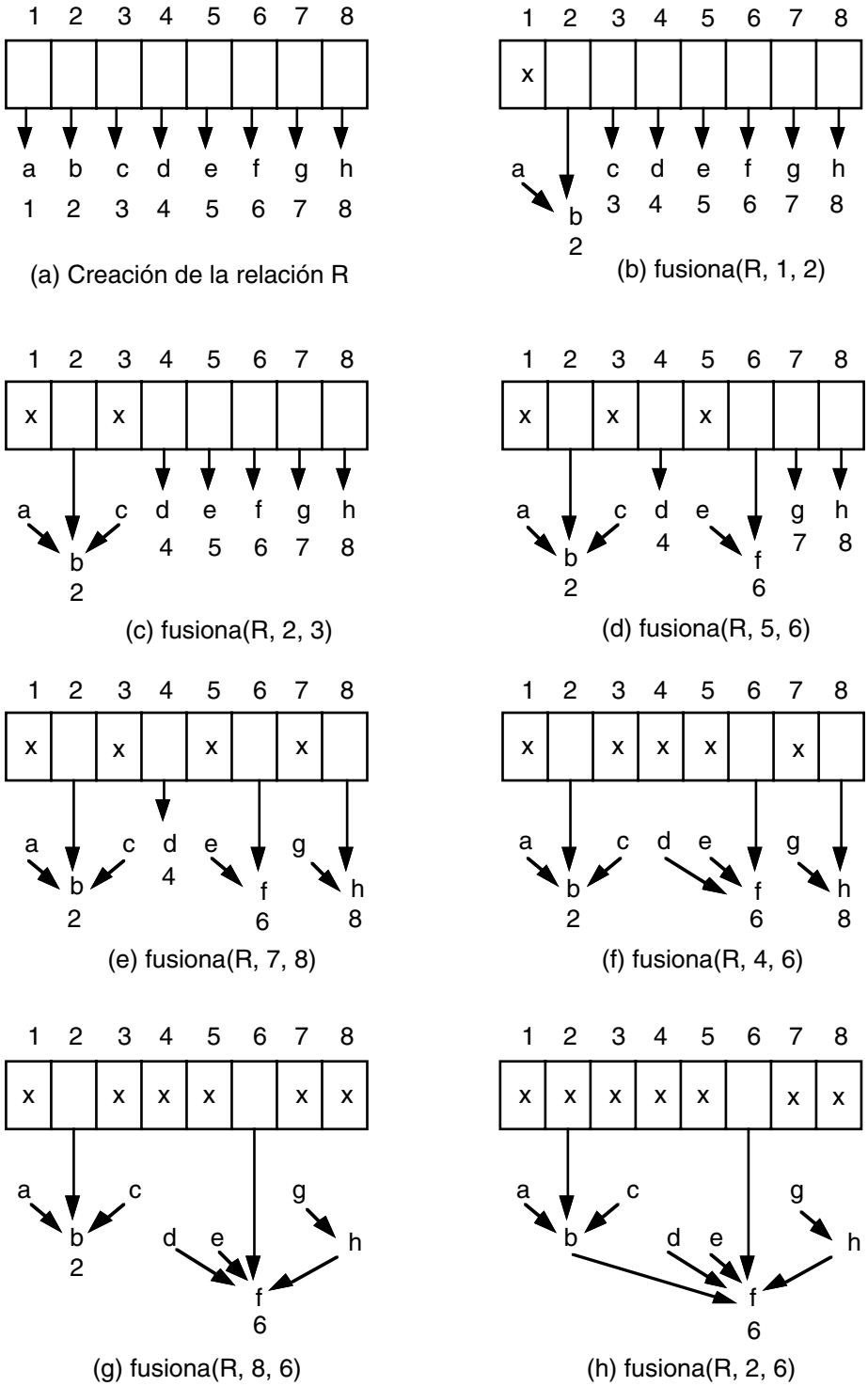


Fig. 5.34: ejemplo de funcionamiento de la representación arborescente del tipo releq.

© Los autores, 1998; © Edicions UPC, 1998.

```
universo RELACIÓN_DE_EQUIVALENCIA_ARBORESCENTE(ELEM_ORDENADO) es
    implementa RELACIÓN_DE_EQUIVALENCIA(ELEM_ORDENADO)
    usa ENTERO, BOOL

    tipo releq es
        tupla
            elems es vector [elem] de tupla id es nat; padre es elem ftupla
            idents es vector [de 1 a n] de tupla cnt es nat; raíz es elem ftupla
            nclases es nat
        ftupla
    ftipo
    invariante (R es releq):
        R.nclases = || { i / R.idents[i].cnt ≠ 0} || ∧
        ∀i: i∈{ i / R.idents[i].cnt ≠ 0 }:
                R.idents[i].cnt = || { v / representante(R, v) = R.idents[i].raíz } || ∧
                R.elems[R.idents[i].raíz].id = i ∧
      donde representante: releq elem →elem se define:
        R.elems[v].id = 0 ⇒ representante(R, v) = representante(R, R.elems[v].padre)
        R.elems[v].id ≠ 0 ⇒ representante(R, v) = v

    función crea devuelve releq es
    var R es releq; v es elem; i es nat fvar
      i := 1
      para todo v dentro de elem hacer
            {se forma la clase i únicamente con v}
        R.elems[v].id := i
        R.idents[i].raíz := v; R.idents[i].cnt := 1
        i := i + 1
      fpara todo
      R.nclases := n
    devuelve R

    función clase (R es releq; v es elem) devuelve nat es
      mientras R.elems[v].id = 0 hacer v := R.elems[v].padre fmientras  {se busca la raíz}
    devuelve R.elems[v].id

    función cuántos? (R es releq) devuelve nat es
    devuelve R.nclases
```

Fig. 5.35: implementación arborescente de las relaciones de equivalencia.

© Los autores, 1998; © Edicions UPC, 1998.

```
función fusiona (R es releq; i1, i2 son nat) devuelve releq es
   si (i1 = 0) ∨ (i1 > n) ∨ (i2 = 0) ∨ (i2 > n) ∨ (R.idents[i1].cnt = 0) ∨ (R.idents[i2].cnt = 0)
      entonces error
      sino si (i1 ≠ i2) entonces
               {se comprueba qué clase tiene más elementos}
               si R.idents[i1].cnt ≥ R.idents[i2].cnt llavors R := cambia(R, i2, i1)
                                                     si no R := cambia(R, i1, i2)
               fsi
               R.nclases := R.nclases - 1  {en cualquier caso desaparece una clase}
            fsi
   fsi
devuelve R

{Función auxiliar cambia: dados dos identificadores de clase, fuente y destino,
 cuelga el árbol asociado a fuente como hijo del árbol asociado a destino
 P ≅ (fuente ≠ destino) ∧ (fuente > 0) ∧ (fuente ≤ n) ∧ (destino > 0) ∧
     (destino ≤ n) ∧ (R.idents[fuente].cnt ≤ R.idents[destino].cnt) ∧ (R = R0)
 Q ≅ ∀v: representante(R0, v) = fuente: representante(R, v) = R.idents[destino].raíz
   ∧ ∀v: representante(R0, v) ≠ fuente: representante(R, v) = representante(R0, v)
   ∧ R.idents[fuente].cnt = 0}
función privada cambia (R es releq; fuente, destino son nat) devuelve releq es
var v es elem fvar
      {primero se cuelga el árbol fuente del árbol destino}
   R.elems[R.idents[fuente].raíz].padre := R.idents[destino].raíz
   R.elems[R.idents[fuente].raíz].id := 0  {marca que no es raíz}
      {a continuación se actualizan los contadores de elementos}
   R.idents[destino].cnt := R.idents[destino].cnt + R.idents[fuente].cnt
   R.idents[fuente].cnt := 0   {marca de clase vacía}
devuelve R
funiverso
```

Fig. 5.35: implementación arborescente de las relaciones de equivalencia (cont.).

5.4.3 Compresión de caminos

Con la estrategia arborescente tal como se ha explicado, el algoritmo sobre relaciones de la fig. 5.29 no es más eficiente que con el uso de la implementación lineal (incluso empeora), porque no sirve de nada mejorar las fusiones sin que las consultas de identificadores sean igualmente rápidas. Por este motivo, introducimos finalmente una técnica llamada compresión de caminos (ing., path compression) que reduce el coste asintótico de una secuencia de operaciones fusiona y clase a lo largo del tiempo, aunque alguna ejecución individual pueda salir perjudicada.

© Los autores, 1998; © Edicions UPC, 1998.

En la implementación de la fig. 5.35, al ejecutar clase(R, v) se recorre el camino C que hay entre v y la raíz de su árbol, donde se encuentra el identificador de la clase. Si posteriormente se repite el proceso para un nodo w antecesor de v dentro del árbol correspondiente, se examinan otra vez todos los nodos entre w y la raíz que ya se habían visitado antes, y que pueden ser muchos. Para evitarlo, se puede aprovechar la primera búsqueda para llevar todos los nodos del camino C al segundo nivel del árbol (es decir, se cuelgan directamente de la raíz); posteriores búsquedas del representante de cualquier descendente de los nodos de C (incluídos los propios) serán más rápidas que antes de la reorganización; concretamente, si el nodo v situado en el nivel k_v es descendiente de un nodo $u \in C$ situado en el nivel k_u, la búsqueda de v pasa de seguir k_v encadenamientos a seguir sólo $k_v - k_u + 2$.

Esta idea presenta un problema: la raíz del árbol correspondiente a la clase no es accesible hasta que no finaliza la búsqueda del representante, de manera que los nodos de C no se puede subir durante la búsqueda misma. Por esto, se recorre C dos veces: en la primera vez, se localiza la raíz (se averigua así el resultado de la consulta) y, durante la segunda, realmente se cuelgan los nodos de C de la raíz. El algoritmo resultante se presenta en la fig. 5.36.

```
función clase (R es releq; v es elem) devuelve nat es
var temp, raíz son elem fvar
        {primero se busca la raíz del árbol}
    raíz := v
    mientras R.elems[raíz].id = 0 hacer raíz := R.elems[raíz].padre fmientras
        {luego se cuelgan de la raíz todos los nodos del camino entre v y la raíz}
    mientras R.elems[v].id = 0 hacer
        temp := v; v := R.elems[v].padre; R.elems[temp].padre := raíz
    fmientras
devuelve R.elems[raíz].id
```

Fig. 5.36: codificación de clase con compresión de caminos.

Es difícil analizar el coste resultante del algoritmo. Si bien fusiona es $\Theta(1)$, es evidente que puede haber ejecuciones individuales de clase costosas (eso sí, nunca más ineficientes que $\Theta(\log n)$), pero es necesario destacar que, precisamente, cuanto peor es una ejecución individual, mejor organizado queda el árbol y su uso futuro se optimiza, porque acerca el máximo número posible de nodos a la raíz (v. fig. 5.37). Se puede demostrar que el coste de una secuencia de n-1 ejecuciones de fusiona y k de clase, $k \geq n$, es $\Theta(k\alpha(k, n))$, siendo α una función no decreciente casi inversa de la llamada función de Ackerman, que cumple que $1 \leq \alpha(k, n) \leq 4$ para todo k y n razonables (v. [CLR90, pp. 450-458] y [BrB87, pp. 63-67]); a efectos prácticos, el coste global de la secuencia de k operaciones es, pues, equivalente a $\Theta(k)$ (aunque aplicando estrictamente la definición de Θ no es así, no obstante el crecimiento

© Los autores, 1998; © Edicions UPC, 1998.

lento de α). Este estudio global del coste se denomina coste amortizado (ing., amortized complexity) y, en nuestro caso, nos permite asegurar que la secuencia de k+n-1 operaciones se comporta como si cada una de ellas fuera $\Theta(1)$; este resultado es más preciso que el análisis habitual del caso peor, que comporta multiplicar el coste individual de las operaciones en el caso peor por la cota superior de su número de ejecuciones, lo cuál resultaría en $\Theta(k \log n)$ en nuestro contexto.

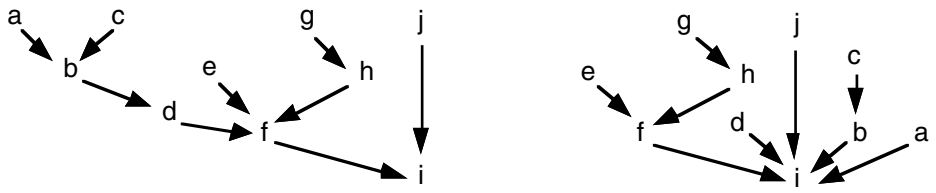


Fig. 5.37: a la derecha, compresión de caminos tras buscar a en el árbol de la izquierda.

5.5 Colas prioritarias

Las colas organizadas por prioridades o, abreviadamente, colas prioritarias (ing., priority queue) son un tipo especial de conjunto donde las operaciones de supresión y consulta afectan siempre al elemento más prioritario (al cual denominaremos elemento menor).

Dado un dominio de elementos V y una operación de comparación según la prioridad, $\leq$ (a partir de la cual se definen automáticamente las operaciones =, $\geq$, < y > que, por ello, pueden ser usadas libremente como abreviatura en el resto de la sección), el modelo de las colas prioritarias de elementos de V son los conjuntos de elementos de V, $P(V)$, con las operaciones siguientes, para $s \in P(V)$ y $v \in V$:

- Crear la cola vacía: crea, devuelve el conjunto Ø.
- Añadir un elemento a la cola: inserta(c, v), devuelve el conjunto $c \cup \{v\}$.
- Obtener el elemento menor de la cola: menor(c), devuelve el elemento $v \in c$, que cumple: $\forall w: w \in c : v \leq w$; da error si c es vacía.
- Borrar el elemento menor de la cola: borra(c), devuelve el conjunto $c - \{v\}$, siendo v el elemento menor(c) de la cola; da error si c es vacía.
- Consultar si la cola está vacía: vacía?(c), devuelve cierto si c es Ø, o falso en caso contrario.

En la fig. 5.38 se muestra la especificación del tipo colapr de las colas prioritarias; los parámetros formales se definen en el universo de caracterización ELEM_$\leq$, similar a ELEM_< (v. fig. 1.36). Recordemos que la comparación se refiere siempre a la igualdad de prioridades. El comportamiento de las operaciones es evidente. Ahora bien, es necesario

© Los autores, 1998; © Edicions UPC, 1998.

señalar que la especificación presenta un problema de consistencia, pues en caso de haber varios elementos con la prioridad más pequeña en la cola, su consulta y supresión depende del orden de aplicación de las ecuaciones. Para solucionar dicho problema puede añadirse la condición $v_1 \neq v_2$ a la ecuación purificadora, con lo cual el elemento afectado por las operaciones será el primero que entró en la cola con dicha prioridad. Este es un nuevo ejemplo que muestra la necesidad de sobreespecificar los TAD para obtener modelos iniciales correctos.

universo COLA_PRIORITARIA (ELEM_≤) es
  usa BOOL
  tipo colapr
  ops crea: → colapr
    inserta: colapr elem → colapr
    menor: colapr → elem
    borra: colapr → colapr
    vacía?: colapr → bool
  errores menor(crea); borra(crea)
  ecns ∀c∈ cuapr; ∀v,v_1,v_2∈ elem
    inserta(inserta(c, v_1), v_2) = inserta(inserta(c, v_2), v_1)
    menor(inserta(crea, v)) = v
    [menor(inserta(c, v_1)) ≤ v_2] ⇒
    menor(inserta(inserta(c, v_1), v_2)) = menor(inserta(c, v_1))
    [¬ (menor(inserta(c, v_1)) ≤ v_2)] ⇒ menor(inserta(inserta(c, v_1), v_2)) = v_2
    borra(inserta(crea, v)) = crea
    [menor(inserta(c, v_1)) ≤ v_2] ⇒
      borra(inserta(inserta(c, v_1), v_2)) = inserta(borra(inserta(c, v_1)), v_2)
    [¬ (menor(inserta(c, v_1)) ≤ v_2)] ⇒ borra(inserta(inserta(c, v_1), v_2)) = inserta(c, v_1)
    vacía?(crea) = cierto; vacía?(inserta(c, v)) = falso
  funiverso

Fig. 5.38: especificación del TAD de las colas prioritarias.

La implementación de las colas prioritarias usando estructuras lineales es costosa en alguna operación. Si mantenemos la lista desordenada, la inserción queda constante, pero la supresión y la consulta exigen una búsqueda lineal; si escogemos ordenar la lista, entonces es la inserción la operación que requiere una búsqueda lineal a cambio del coste constante de la consulta, mientras que la supresión depende de la representación concreta de la lista (si es secuencial, es necesario mover elementos y entonces queda lineal). Una opción intermedia consiste en mantener la lista desordenada durante las inserciones y ordenarla a

© Los autores, 1998; © Edicions UPC, 1998.

continuación, siempre que todas las inserciones se hagan al principio y las consultas y supresiones a continuación, como pasa con cierta frecuencia. En este caso, el coste total de mantener una cola de n elementos (es decir, primero insertar los n elementos y después consultarlos y obtenerlos de uno en uno) queda $\Theta(n\log n)$ con un buen algoritmo de ordenación en comparación con el coste $\Theta(n^2)$ de las dos anteriores.

5.5.1 Implementación por árboles parcialmente ordenados y casi completos

A continuación, introducimos una representación que garantiza el coste logarítmico de las operaciones modificadoras de las colas prioritarias manteniendo el coste constante de la consultora. Esta representación usa una variante de árbol conocida con el nombre de árbol parcialmente ordenado (ing., partially ordered tree), que cumple que todo nodo es menor que sus hijos, si tiene. Además, nos interesará que el árbol sea casi completo (o sea, que no presente discontinuidades en un recorrido por niveles) para asegurar una buena eficiencia espacial y temporal. Es obvio que en esta clase de árbol el elemento menor reside en la raíz y, por lo tanto, la ejecución de menor es constante si disponemos de acceso directo a la raíz desde la representación del tipo. Es necesario ver, pues, cómo quedan las operaciones de inserción y de supresión. Para hacer este estudio trataremos árboles parcialmente ordenados binarios; v. ejercicio 5.13 para el estudio de la generalización a cualquier aridad[14].

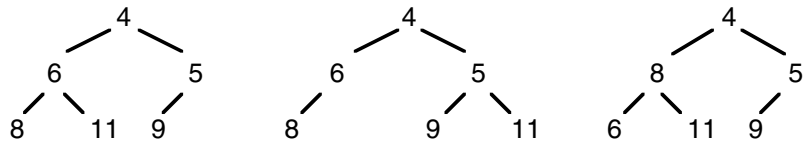


Fig. 5.39: tres árboles binarios con los mismos elementos: parcialmente ordenado y completo (izquierda), sólo parcialmente ordenado (medio) y sólo completo (derecha).

a) Inserción en un árbol parcialmente ordenado y casi completo

Para que el árbol resultante de una inserción sea completo, insertamos el nuevo elemento v en la primera posición libre en un recorrido por niveles del árbol. Ahora bien, en el caso general, esta inserción no da como resultado el cumplimiento de la propiedad de los árboles parcialmente ordenados, porque v puede tener una prioridad más pequeña que su padre. Debido a ello, es necesario comenzar un proceso de reestructuración del árbol, que consiste en ir intercambiando v con su padre hasta que la relación de orden se cumpla, o bien hasta que v llegue a la raíz; cualquiera de las dos condiciones de parada implica que el árbol vuelve a cumplir la propiedad de ordenación parcial.

[14] Los árboles parcialmente ordenados n-arios, n > 2, pueden ser interesantes en algunos contextos, porque, cuanto mayor es la n, menos comparaciones entre elementos es necesario hacer en las inserciones.

© Los autores, 1998; © Edicions UPC, 1998.

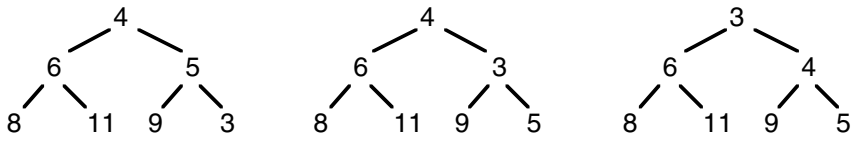


Fig. 5.40: inserción del 3 paso a paso en el árbol de la fig. 5.39, izquierda.

Notemos que, en el caso peor, el número de intercambios de elementos es igual al número de niveles del árbol y, como el árbol es casi completo, el número de niveles es del orden del logaritmo en base 2 del número n de elementos del árbol, exactamente $\lceil \log_2(n+1) \rceil$ (recordemos que en el nivel i caben hasta 2^{i-1} nodos); si la representación de este tipo de árbol permite realizar cada intercambio en tiempo constante, el coste de la operación será efectivamente $\Theta(\log n)$.

b) Supresión en un árbol parcialmente ordenado y casi completo

Dado que el elemento menor reside en la raíz, su supresión consiste en eliminar ésta; ahora bien, el resultado no presenta una estructura arborescente y, por ello, es necesario reestructurar el árbol. Para formar un nuevo árbol casi completo simplemente se mueve el último elemento v del árbol en un recorrido por niveles hasta la raíz. Sin embargo, normalmente esta nueva raíz no es más pequeña que sus hijos, lo cual obliga también aquí a reestructurar el árbol con una estrategia muy similar: se va intercambiando v con uno de sus hijos hasta que cumple la relación de orden, o vuelve a ser una hoja. Sólo es necesario notar que, en caso de que los dos hijos sean más pequeños que v, en el intercambio intervendrá el menor de ellos; de lo contrario, el nuevo padre no cumpliría la buena relación de orden con el hijo no movido. De la misma manera que en la inserción, y por el mismo razonamiento, el coste asintótico de la operación es $\Theta(\log n)$.

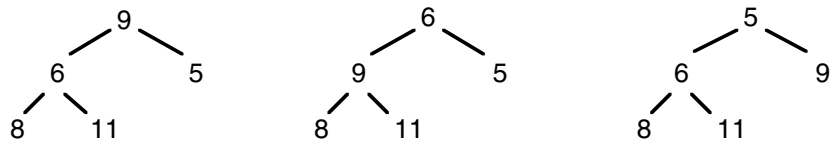


Fig. 5.41: supresión del menor en el árbol de la fig. 5.39, izquierda: se sube el 9 a la raíz (izquierda); se intercambia, erróneamente, el 9 con el 6, violando la relación de orden (medio). En realidad, pues, se intercambia con el menor de los hijos (derecha).

Como queda claro a partir de la explicación de los algoritmos de las operaciones sobre colas prioritarias, dado un árbol parcialmente ordenado es necesario acceder rápidamente a la raíz y al último elemento en un recorrido por niveles; a la raíz, porque en ella reside el elemento menor y al último, porque es el elemento afectado al insertar o borrar. Además, también

© Los autores, 1998; © Edicions UPC, 1998.

queda claro que, a causa de los procesos de reestructuración del árbol, a partir de un nodo es necesario acceder a sus hijos y a su padre. Finalmente, recordemos que el árbol parcialmente ordenado siempre será casi completo. Precisamente, los dos últimos hechos apuntan a una representación secuencial del árbol mediante un montículo. Recordemos que los montículos (v. el apartado 5.2.1) organizan los nodos del árbol dentro de un vector, lo que permite el acceso a los hijos y al padre de un nodo determinado sin necesidad de mantener encadenamientos, sino aplicando una fórmula. En el caso de los árboles binarios, dado un nodo que resida en la posición i del montículo, el padre residirá en la posición $\lfloor i/2 \rfloor$, el hijo izquierdo en la $2i$ y el hijo derecho en la $2i + 1$. Además, el aprovechamiento óptimo del espacio en un montículo se da cuando el árbol que se quiere representar es casi completo, como es el caso que nos ocupa, porque así no hay posiciones intermedias vacías dentro del vector. No obstante, en alguna situación concreta puede considerarse el uso de representaciones encadenadas. En el resto del apartado se estudia la representación por montículo; la representación encadenada se propone en el ejercicio 5.14.

El universo se presenta en la fig. 5.42. Notemos que en la cabecera aparece el universo de caracterización VAL_NAT (v. fig. 1.32), que define una constante natural que representa el número máximo de elementos que caben en la cola, motivo por el que la especificación de las colas no es exactamente la que aparece en la fig. 5.38, sino que también sería necesario controlar el tamaño (de manera similar a todos los TAD ya estudiados). En la representación sólo se necesita el vector para guardar los elementos y un apuntador al primer sitio libre; la raíz estará siempre en la primera posición del vector y así no es necesario ningún apuntador explícito. El invariante de la representación garantiza que la relación de orden entre los nodos se cumple. Por lo que respecta a la codificación de las operaciones del tipo, la creación y la obtención del menor son inmediatas, y la inserción y la supresión del menor siguen la casuística que se acaba de dar, tal como establecen los invariantes esbozados de manera informal. Notemos que, efectivamente, el coste de estas dos últimas operaciones queda logarítmico, dado que los intercambios son de orden constante, y que, además, en realidad no se intercambian los elementos a cada paso, sino que el elemento en proceso de ubicación no se mueve hasta conocer la posición destino, lo que ahorra la mitad de los movimientos (aunque no se mejora el coste asintótico).

© Los autores, 1998; © Edicions UPC, 1998.

```
universo COLA_PRIORITARIA_POR_MONTÍCULO(ELEM_≤, VAL_NAT) es
    implementa COLA_PRIORITARIA(ELEM_≤, VAL_NAT)
    usa BOOL
    tipo colapr es
        tupla
            A es vector [de 1 a val] de elem
            sl es nat
        ftupla
    ftipo
    invariante (C es colapr): 1 ≤ C.sl ≤ val+1 ∧ es_cola(C.A, 1, C.sl-1),
        siendo el predicado es_cola(A, i, j) definido como:
            [2r > s] ⇒ es_cola(A, r, s) = cierto
            [2r = s] ⇒ es_cola(A, r, s) = A[r] ≥ A[2r]
            [2r < s] ⇒ es_cola(A, r, s) = (A[r] ≥ A[2r]) ∧ (A[r] ≥ A[2r+1]) ∧
                                          es_cola(A, 2r, s) ∧ es_cola(A, 2r+1, s)
    función crea devuelve colapr es
    var C es colapr fvar
        C.sl := 1
    devuelve C
    función inserta (C es colapr; v es elem) devuelve colapr es
    var k es nat; fin? es bool fvar
        si C.sl = val+1 entonces error  {cola llena}
        sino {se busca la posición destino de v y se mueven los elementos afectados}
            C.sl := C.sl+1  {sitio para el nuevo elemento}
            k := C.sl-1; fin? := falso
            mientras ¬ fin? ∧ (k > 1) hacer
                    {I ≅ es_cola(C.A, 2k, C.sl-1)}
                si v < C.A[k/2] entonces C.A[k] := C.A[k/2]; k := k/2 si no fin? := cierto fsi
            fmientras
            C.A[k] := v   {inserción del nuevo elemento en su sitio}
        fsi
    devuelve C
    función menor (C es colapr) devuelve elem es
    var res es elem fvar
        si C.sl = 1 entonces error si no res := C.A[1] fsi
    devuelve res
    función vacía? (C es colapr) devuelve bool es
    devuelve C.sl = 1
```

Fig. 5.42: implementación de las colas prioritarias usando un montículo.

© Los autores, 1998; © Edicions UPC, 1998.

```
función borra (C es colapr) devuelve colapr es
var k, hmp son nat; éxito? es bool fvar
    si C.sl = 1 entonces error  {cola vacía}
    si no {se busca la posición destino del último y se mueven los elementos afectados}
        k := 1; éxito? := falso
        mientras (k*2 < C.sl-1) ∧ ¬ éxito? hacer
              {I ≅ despreciando el subárbol de raíz k, el árbol es parcialmente ordenado}
            hmp :=menor_hijo(A, k)
            si C.A[hmp] < C.A[C.sl-1] entonces C.A[k] := C.A[hmp]; k := hmp
                                          si no éxito? := cierto
            fsi
        fmientras
        C.A[k] := C.A[C.sl-1]; C.sl := C.sl-1     {inserción del elemento en su sitio}
    fsi
devuelve C

{Función auxiliar menor_hijo: dado un nodo, devuelve la posición de su hijo
 menor; si sólo tiene uno, devuelve la posición de este único hijo. Como
 precondición, el nodo tiene como mínimo hijo izquierdo}
función privada menor_hijo (C es colapr; k es nat) devuelve nat es
var i es nat fvar
    si k*2+1 = C.sl-1 entonces i := k*2     {sólo tiene hijo izquierdo}
    si no si C.A[k*2] < C.A[k*2+1] entonces i := k*2  si no i := k*2+1 fsi
    fsi
devuelve i
funiverso
```

Fig. 5.42: implementación de las colas prioritarias usando un montículo (cont.).

5.5.2 Aplicación: un algoritmo de ordenación

Los algoritmos de ordenación son una de las familias más clásicas de esquemas de programación y han dado lugar a resoluciones realmente brillantes e ingeniosas. Una de ellas es el algoritmo de ordenación por montículo (ing., heapsort), presentado en 1964 por J.W.J. Williams en "Heapsort (Algorithm 232)", Communications ACM, 7(6), que se basa en el tipo de las colas prioritarias. En él, se recorre la lista a ordenar y sus elementos se insertan en una cola prioritaria; una vez se han insertados todos, se obtienen uno a uno hasta que la cola queda vacía y cada elemento obtenido, que será el menor de los que queden en la cola en aquel momento, se inserta en la lista resultado y se borra de la cola (v. fig. 5.43; en el invariante, la función elems devuelve el conjunto de elementos de una lista o cola, y máximo el elemento mayor de la lista).

© Los autores, 1998; © Edicions UPC, 1998.

```
función heapsort (l es lista) devuelve lista es
var C es colapr; v es elem; lres es lista fvar
        {primer paso: se construye la cola con los elementos de l }
    C := COLA_PRIORITARIA.crea
    para todo v dentro de l hacer C := COLA_PRIORITARIA.inserta(C, v) fpara todo
        {segundo paso: se construye la lista ordenada con los elementos de C }
    lres := LISTA_INTERÉS.crea
    mientras ¬ COLA_PRIORITARIA.vacía?(C) hacer
        {I ≅ elems(lres) ∪ elems(C) = elems(l) ∧ menor(C) ≥ máximo(lres) ∧ ordenada(lres)}
        lres := LISTA_INTERÉS.inserta(lres, COLA_PRIORITARIA.menor(C))
        C := COLA_PRIORITARIA.borra(C)
    fmientras
devuelve lres
```

Fig. 5.43: algoritmo de ordenación por montículo.

Al analizar el coste del algoritmo para una lista de n elementos, observamos que cada uno de los bucles se ejecuta n veces, porque en cada vuelta se trata un elemento. El paso i-ésimo del primer bucle y el paso (n-i+1)-ésimo del segundo tienen un coste $\Theta(\log i)$, de manera que el coste total es igual a $\Sigma i: 1 \leq i \leq n: \Theta(\log i) = \Theta(n \log n)$, que es la cota inferior de los algoritmos de ordenación, asintóticamente hablando; en este caso, el coste así determinado coincide con el cálculo hecho con las reglas habituales, determinado como el producto del número n de iteraciones multiplicado por el coste individual de la operación en el caso peor, $\Theta(\log n)$. No obstante, la cola exige un espacio adicional lineal, lo cual es inconveniente. Para mejorar este resultado, R.W. Floyd formuló en el mismo año 1964 una variante del método ("Treesort (Algorithm 243)", Communications ACM, 7(12)), aplicable en el caso de que la lista esté representada por un vector directamente manipulable desde el algoritmo (sería el típico caso de ordenar un vector y no una lista). El truco consiste en dividir el vector en dos trozos, uno dedicado a simular la cola y el otro a contener parte de la solución. Dentro de la cola, los elementos se ordenarán según la relación > (inversa de la ordenación resultado), de manera que en la raíz siempre esté el elemento mayor. Es necesario notar que, en la primera fase del algoritmo, en realidad se dispone de varias colas que se van fusionando para obtener la cola (única) final. El algoritmo no exige ninguna estructura voluminosa auxiliar, pero, en cambio, no es en absoluto modular, pues sólo es aplicable sobre representaciones secuenciales.

Fig. 5.44: heapsort: construcción de la cola (izq.) y ordenación de los elementos (der.); la flecha indica el sentido del desplazamiento de la frontera entre las partes del vector.

© Los autores, 1998; © Edicions UPC, 1998.

Para escribir una versión sencilla y fácilmente legible del algoritmo de ordenación, introducimos una acción auxiliar hunde (denominada heapify en algunos textos en lengua inglesa), por la que hunde(A, pos, máx) reubica el elemento v residente en la posición pos del vector A, intercambiando reiteradamente con el mayor de sus hijos, hasta que v sea mayor que sus hijos, o bien hasta que v sea una hoja; para saber que un elemento es hoja basta con conocer la posición del último elemento por niveles, que será el parámetro máx. Como precondición, se supone que pos no es hoja y que sus dos hijos (o sólo uno, si no tiene hijo derecho) son subárboles parcialmente ordenados. La acción correspondiente se presenta en la fig. 5.45; notemos su similitud con la función borra de las colas con prioridad, que es una particularización. En las pre y postcondiciones se usa un predicado, elems, que da como resultado el conjunto de elementos entre dos posiciones dadas del vector, y que se usa para establecer que los elementos del vector son los mismos y, en particular, que los que había en el trozo afectado por la ordenación también se mantienen. Por lo que respecta a es_cola, se considera el criterio de ordenación inverso que la fig. 5.42.

```
{P ≅ 1 ≤ pos ≤ ⌊máx/2⌋ ≤ n ∧ A = A0 ∧ es_cola(A, 2pos, máx) ∧ es_cola(A, 2pos+1, máx) }
acción privada hunde (ent/sal A es vector [de 1 a n] de elem; ent pos, máx son nat) es
var hmp es nat; temp es elem; éxito? es bool fvar
        {se busca la posición que ocupará el nuevo elemento y, mientras, se mueven
         los elementos afectados}
    temp := pos
    mientras (pos*2 < máx) ∧ ¬éxito? hacer
    {I ≅ despreciando los nodos que cuelgan de pos, el árbol es parcialmente ordenado}
        hmp := mayor_hijo(A, pos) {inverso de menor_hijo, v. fig. 5.42}
        si A[hmp] > temp entonces A[pos] := A[hmp]; pos := hmp si no éxito? := cierto fsi
    fmientras
    C.A[pos] := temp   {inserción del elemento que se está reubicando en su sitio}
facción
{Q ≅ elems(A, pos, máx) = elems(A0, pos, máx) ∧ elems(A, 1, máx) = elems(A0, 1, máx)
            ∧ es_cola(A, pos, máx) }, donde se define elems(A, r, s) ≅ {A[i] / r ≤ i ≤ s}
```

Fig. 5.45: algoritmo de ubicación de un elemento en un montículo.

Finalmente, en la fig. 5.47 se codifica el algoritmo de ordenación y en la fig. 5.46 se muestra un ejemplo de funcionamiento. Como en la versión anterior, distinguimos dos partes:

- Formación de la cola: se van ubicando los nodos en un recorrido inverso por niveles del árbol, de manera que dentro del vector residan diversos árboles parcialmente ordenados, y a cada paso se fusionan dos árboles y un elemento en un único árbol. Notemos que este recorrido no trata las hojas, porque ya son árboles parcialmente ordenados.

© Los autores, 1998; © Edicions UPC, 1998.

- Construcción de la solución: a cada paso se selecciona el elemento mayor de la cola (que reside en la primera posición) y se coloca en la posición i que divide el vector en dos partes, de manera que todos los elementos de su izquierda sean más pequeños y todos los de su derecha más grandes; estos últimos, además, habrán sido ordenados en pasos anteriores. A continuación, el elemento que ocupaba previamente la posición i se sitúa como nueva raíz y se reorganiza el árbol resultante.

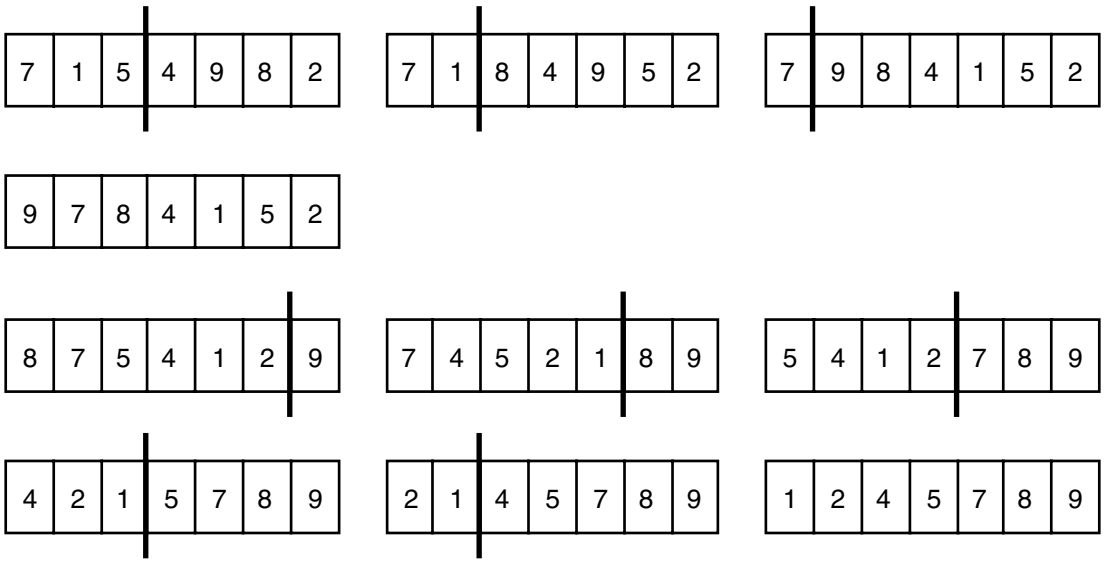


Fig. 5.46: ejemplo de ordenación de un vector con el método del montículo; en las dos filas superiores se muestra la formación de la cola y en las dos inferiores la extracción ordenada de elementos; la barra vertical más gruesa indica la partición lógica del vector.

La versión resultante, efectivamente, no precisa de espacio auxiliar y su coste temporal asintótico se mantiene $\Theta(n\log n)$. Ahora bien, si examinamos detenidamente la primera parte del algoritmo, veremos que su coste no es en realidad $\Theta(n\log n)$, sino simplemente $\Theta(n)$. Ello se debe al hecho de que, al dar una vuelta al bucle que hay dentro de hunde, pos vale como mínimo el doble que en el paso anterior. Por lo tanto, para pos entre n/2 y n/4+1 el bucle (que tiene un cuerpo de coste constante) se ejecuta como mucho una vez, entre n/4 y n/8+1 como mucho dos, etc. La suma de estos factores queda, para un árbol de k niveles, siendo $k = \lceil \log_2(n+1) \rceil$, igual a $\Sigma i: 1 \leq i \leq k-1: 2^{i-1}(k-i)$, donde cada factor del sumatorio es un nivel, el valor absoluto de la potencia de 2 es el número de nodos del nivel, y k-i es el número máximo de movimientos de un nodo dentro del nivel. Esta cantidad está acotada por 2n y así el coste asintótico resultante queda $\Theta(n)$. Si bien en el algoritmo de ordenación el coste asintótico no queda afectado, puede ser útil que las colas prioritarias ofrezcan una función organiza que implemente esta conversión rápida de un vector en una cola, porque puede

© Los autores, 1998; © Edicions UPC, 1998.

reducir el coste de otros algoritmos (esta función tendría que trabajar sobre la representación del tipo y por ello formaría parte del modelo de las colas). Un ejemplo es la modificación de heapsort para que, en lugar de ordenar todo el vector, sólo obtenga los k elementos más grandes, $k < n$, en cuyo caso, el coste del algoritmo modificado sería $\Theta(n+k \log n)$, que podría llegar a quedar $\Theta(n)$ si $k \leq n/\log n$. También el algoritmo de Kruskal sobre grafos (v. sección 6.5) se beneficia de esta reducción de coste.

```
{P ≅ n ≥ 1 ∧ A = A0}
acción heapsort (ent/sal A es vector [de 1 a n] de elem) es
var i es nat; temp es elem fvar
    {primero, se forma la cola}
  para todo i desde n/2 bajando hasta 1 hacer
      {I ≅ ∀k: i+1 ≤ k ≤ n: es_cola(A, k, n) ∧ elems(A, 1, máx) = elems(A0, 1, máx)}
    hunde(A, i, n)
  fpara todo
    {a continuación, se extraen ordenadamente los elementos}
  para todo idesde n bajando hasta 2 hacer
      {I ≅ es_cola(A, 1, i) ∧ ordenado(A, i+1, n) ∧
          ∧ elems(A, 1, máx) = elems(A0, 1, máx) ∧ (i < n ⇒ A[1] ≤ A[i+1])}
    temp := A[1]; A[1] := A[i]; A[i] := temp      {intercambio de los elementos}
    hunde(A, 1, i-1)     {reorganización del árbol}
  fpara todo
facción
{Q ≅ elems(A, 1, n) = elems(A0, 1, n) ∧ ordenado(A, 1, n)}
```

Fig. 5.47: algoritmo de ordenación por montículo de un vector.

5.6 Tablas ordenadas

El TAD de las tablas ordenadas ha sido introducido en la sección 4.5 como una estructura que combina el acceso individual a los elementos de una tabla y su obtención ordenada. También se han citado diversas posibilidades de implementación, entre la que destaca el uso de tablas de dispersión. Ahora bien, la organización por dispersión presenta algunas características negativas que es necesario comentar:

- El acceso a los elementos de la tabla puede dejar de ser constante si la función de dispersión no los distribuye bien en las diferentes cubetas. Para controlar este mal funcionamiento es necesario añadir código de control a la implementación de la tabla y, si se detecta, se ha de redefinir la función y volver a insertar los elementos.
- Es necesario realizar un estudio cuidadoso para decidir cuál es la organización de dispersión adecuada para el contexto concreto de uso, tal como se muestra en el

© Los autores, 1998; © Edicions UPC, 1998.

apartado 4.4.6.

- Se ha de determinar el número aproximado de elementos que se espera guardar en la tabla, aun cuando no se tenga la más remota idea. Si, con el tiempo, la tabla queda pequeña, es necesario agrandarla, redefinir la función de dispersión y reinsertar los elementos (a no ser que se usen métodos incrementales que no se han tratado en este texto); si se revela excesiva, se desperdiciará espacio de la tabla.
- En el contexto de las tablas ordenadas hay alguna operación forzosamente lineal: o bien la inserción, si se mantiene siempre una estructura encadenada ordenada de los elementos, o bien la preparación del recorrido, si se mantiene la estructura sin ningún orden y se ordena previamente al recorrido (en concreto, el coste del último caso es casi lineal).

A continuación se introduce una técnica de representación de las tablas mediante árboles que permite recorridos ordenados eficientes sin que ninguna operación quede lineal. Cada elemento tendrá asociados como mínimo dos encadenamientos y un campo adicional de control, de manera que ocupará más en esta nueva estructura, pero como no será necesario dimensionar a priori ningún vector, es posible que el espacio total resultante no sea demasiado diferente (v. ejercicio 5.25). Estos árboles se denominan árboles binarios de búsqueda.

5.6.1 Árboles binarios de búsqueda

Sea V un dominio de elementos y sea < la operación de comparación que define un orden total (para simplificar las explicaciones, usaremos también el resto de operadores relacionales). Un árbol binario de búsqueda (ing., binary search tree) es un árbol binario etiquetado con los elementos de V tal que, o bien es el árbol vacío, o bien su raíz es mayor que todos los elementos de su subárbol izquierdo (si tiene) y menor que todos los elementos de su subárbol derecho (si tiene) y, además, sus subárboles izquierdo y derecho son también árboles de búsqueda (si existen). De manera más formal, definimos $A^{C}{}_{V}$ como los árboles binarios de búsqueda sobre V, $A^{C}{}_{V} \subseteq A^{2}{}_{V}$:

- $\emptyset \in A^{C}{}_{V}$.
- $\forall a, a' : a, a' \in A^{C}{}_{V} : dom(a) \neq \emptyset \wedge dom(a') \neq \emptyset$:
  $\forall v; V \in V :$ máx $\{n / <s, n> \in N_a\} < V \wedge v <$ mín $\{n / <s, n> \in N_{a'}\}$:
  enraiza($\emptyset$, v, $\emptyset$), enraiza($\emptyset$, v, a'), enraiza(a, v, $\emptyset$), enraiza(a, v, a') $\in A^{C}{}_{V}$.

En la fig. 5.48 se muestran tres árboles binarios con etiquetas naturales; el de la izquierda y el del medio son de búsqueda, pero no el de la derecha, ya que su subárbol izquierdo contiene un elemento, el 9, mayor que la raíz. Los dos árboles de búsqueda contienen los mismos naturales. En general, se pueden formar muchos árboles de búsqueda con los mismos elementos (v. ejercicio 5.17).

© Los autores, 1998; © Edicions UPC, 1998.

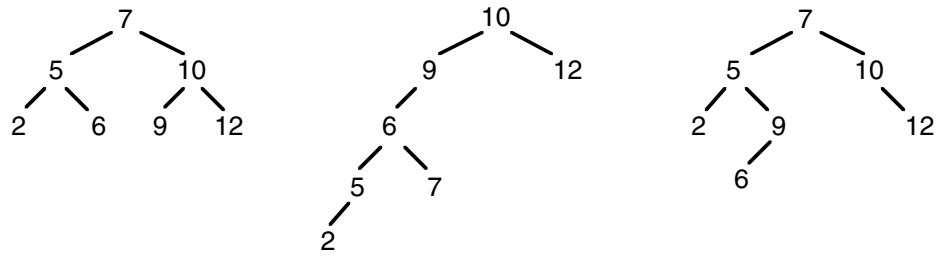


Fig. 5.48: dos árboles de búsqueda con los mismos elementos y uno que no lo es (der.).

La propiedad que caracteriza los árboles de búsqueda es que, independientemente de su forma, su recorrido en inorden proporciona los elementos ordenados precisamente por la relación exigida entre la raíz del árbol y las raíces de sus subárboles. Esta propiedad se podría comprobar en los dos árboles de búsqueda de la fig. 5.48, y se podría demostrar fácilmente por inducción sobre la forma de los árboles a partir de su definición recursiva (queda como ejercicio para el lector). La adecuación de los árboles de búsqueda como implementación de las tablas ordenadas se basa en este hecho, que permite obtener los n elementos de la tabla en $\Theta(n)$, ya sea con árboles enhebrados o no. A continuación, es necesario estudiar el coste de las operaciones de acceso individual a la tabla para acabar de determinar la eficiencia temporal de la estructura.

Las operaciones de acceso individual se basan en la búsqueda de un elemento en el árbol y se rigen por un esquema bastante evidente. Sea $a \in A^{C_V}$ un árbol de búsqueda y sea $v \in V$ el elemento a buscar:

- Si a es el árbol vacío Ø, se puede afirmar que v no está dentro del árbol.
- En caso contrario, se comparará v con la raíz de a y habrá tres resultados posibles:
  - ◊ v = raíz(a): el elemento ha sido encontrado en el árbol.
  - ◊ v < raíz(a): se repite el proceso dentro del subárbol izquierdo de a.
  - ◊ v > raíz(a): se repite el proceso dentro del subárbol derecho de a.

a) Inserción en un árbol binario de búsqueda

Para que el árbol resultante de una inserción en un árbol de búsqueda sea también de búsqueda, se aplica la casuística descrita para localizar el elemento. Si se encuentra, no es necesario hacer nada, si no, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en caso de existir). En la fig. 5.49 se muestra la inserción de dos elementos dentro de un árbol de búsqueda; en los dos casos y como siempre sucede, el nuevo elemento se inserta como una hoja, dado que precisamente la búsqueda acaba sin éxito cuando se accede a un subárbol izquierdo o derecho que está vacío.

© Los autores, 1998; © Edicions UPC, 1998.

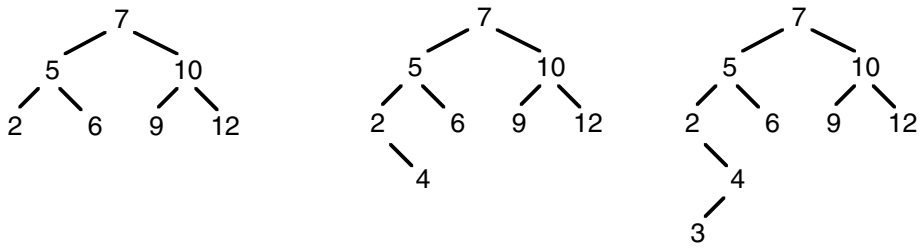


Fig. 5.49: inserción del 4 (medio) y del 3 (derecha) en el árbol de búsqueda de la izquierda.

Si el elemento a insertar no está en el árbol, el algoritmo de inserción recorre un camino desde la raíz hasta una hoja; obviamente, el camino más largo tiene como longitud el número de niveles del árbol. Si tenemos suerte y el árbol es casi completo, el número de niveles es logarítmico. No obstante, en el caso peor el número de niveles de un árbol binario de búsqueda de n elementos es n, ya que no hay ninguna propiedad que restrinja la forma del árbol y, por ello, el coste resultante puede llegar a ser lineal; este caso peor se da en la inserción ordenada de los elementos que resulta en un árbol completamente degenerado, donde cada nodo tiene un hijo y sólo uno. Se puede demostrar que el número esperado de niveles de un árbol binario de búsqueda después de insertar n nodos es asintóticamente logarítmico, aproximadamente $2(\ln n + \gamma + 1)$, siendo $\gamma = 0.577...$ la constante de Euler y suponiendo que el orden de inserción de los n nodos es equiprobable [Wir86, pp. 214-217].

b) Supresión en un árbol binario de búsqueda

Para localizar un elemento v dentro del árbol se aplica el algoritmo de búsqueda habitual. Si el elemento no se encuentra, la supresión acaba; de lo contrario, el comportamiento exacto depende del número de hijos que tiene el nodo n = <s, v> que contiene el elemento:

- Si n es una hoja, simplemente desaparece.
- Si n tiene un único subárbol, ya sea izquierdo o derecho, se sube a la posición que ocupa n.
- Si n tiene dos hijos, ninguno de los dos comportamientos descritos asegura la obtención de un árbol binario de búsqueda, sino que es necesario mover otro nodo del árbol a la posición s. ¿Cuál? Para conservar la propiedad de los árboles de búsqueda, se mueve el mayor de los elementos más pequeños que v (que está dentro del subárbol izquierdo del árbol que tiene n como raíz), o bien el menor de los elementos más grandes que v (que está dentro del subárbol derecho del árbol que tiene n como raíz). En cualquier caso, el nodo n' que contiene el elemento que responde a esta descripción es, o bien una hoja, o bien un nodo con un único hijo, de manera que a continuación se le aplica el tratamiento correspondiente en estos casos.

En la fig. 5.50 se muestran las supresiones de los elementos 6, 2 y 7 en el árbol de la fig.

© Los autores, 1998; © Edicions UPC, 1998.

5.49, derecha. Cada una de las supresiones se corresponde con los tres casos citados; en particular, la supresión del 7 lleva al menor de los mayores al nodo que lo contiene. Por lo que respecta a la eficiencia temporal, pueden repetirse los razonamientos hechos en la inserción.

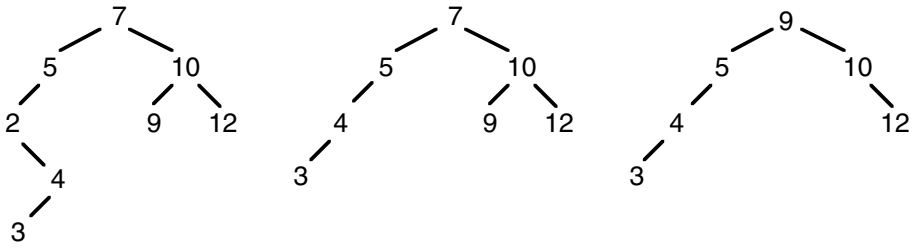


Fig. 5.50: supresión del 6 (izq.), 2 (medio) y 7 (der.) en el árbol de la fig. 5.49, derecha.

En la fig. 5.51 se muestra una implementación por árboles de búsqueda de las tablas ordenadas que sigue la estrategia dada. Para simplificar la discusión, se ha escogido la especificación de las tablas ordenadas con una única operación que devuelve la lista ordenada de los elementos (v. fig. 4.29); queda como ejercicio para el lector una versión usando árboles con punto de interés. Observemos que no es necesario manipular directamente la representación de los árboles binarios, sino que simplemente se instancia un universo adecuado que ofrezca operaciones de recorrido y, a continuación, se restringen los valores posibles estableciendo un invariante, que caracteriza la propiedad de los árboles de búsqueda con el uso de dos funciones auxiliares para obtener el máximo y el mínimo de un árbol de búsqueda; notemos que las propiedades sobre la forma del árbol binario (v. fig. 5.8, por ejemplo) son implicadas por las propiedades exigidas a los árboles de búsqueda. La instancia es privada, porque las operaciones propias de los árboles binarios no han de ser accesibles al usuario del tipo. Las instancias de los universos para definir pares y listas ya se hicieron en la especificación y no es necesario repetirlas. Por lo que respecta a las operaciones, se implementan recursivamente, dado que es la manera natural de traducir el comportamiento que se acaba de describir; no obstante, el algoritmo de supresión presenta un par de ineficiencias al borrar un nodo que tiene dos hijos, que no se eliminan del algoritmo para mayor claridad en la exposición: por un lado, para obtener el menor de los mayores se hace un recorrido en inorden del subárbol derecho, cuando se podría simplemente bajar por la rama correspondiente; por otro, para borrar este elemento se vuelve a localizar recursivamente con borrar ; la solución de estas ineficiencias (que no afectan al coste asintótico) queda como ejercicio para el lector. Notemos que las repetidas invocaciones a enraizar provocan cambios en la posición física de las etiquetas (aunque la forma del árbol no varíe), lo cual puede ser un inconveniente; la modificación de la codificación para evitar este movimiento exige manipular las direcciones físicas de los nodos como mínimo en la supresión de nodos con dos hijos, por lo que debe escribirse la representación del tipo explícitamente en el universo.

© Los autores, 1998; © Edicions UPC, 1998.

```
universo ÁRBOL_BINARIO_DE_BÚSQUEDA (A es ELEM_<_= , B es ELEM_ESP) es
    implementa TABLA_ORDENADA (A es ELEM_<_=, B es ELEM_ESP)
    renombra A.elem por clave, B.elem por valor, B.esp por indef
    usa BOOL
    instancia privada ÁRBOL_BINARIO_CON_RECORRIDOS (C es ELEM)
        donde C.elem es elem_tabla
    renombra árbol por árbol_búsqueda
    tipo tabla es árbol_búsqueda ftipo
    invariante (T es tabla): correcto(T)
        donde el predicado correcto: árbol_búsqueda → bool se define:
            correcto(crea) = cierto
            correcto(enraiza(a1, v, a2)) = correcto(a1) ∧ correcto(a2) ∧
                                           { ¬ vacío?(a1) ⇒ v.clave > máximo(a1) } ∧
                                           { ¬ vacío?(a2) ⇒ v.clave < mínimo(a2) }
        y la función máximo: árbol_búsqueda → elem (y simétricamente mínimo) es:
            vacío?(hder(a)) ⇒ máximo(a) = raíz(a).clave
            ¬vacío?(hder(a)) ⇒ máximo(a) = máximo(hder(a))

    función crea devuelve tabla es
    devuelve ÁRBOL_BINARIO_CON_RECORRIDOS.crea

    función asigna (a es tabla; k es clave; v es valor) devuelve tabla es
        si vacío?(a) entonces {se crea un árbol de un único nodo}
            a := enraiza(crea, <k, v>, crea)
        si no
            opción
                caso raíz(a).clave = k hacer {se sustituye la información asociada}
                    a := enraiza(hizq(a), <k, v>, hder(a))
                caso raíz(a).clave < k hacer {debe insertarse en el subárbol derecho}
                    a := enraiza(hizq(a), raíz(a), asigna(hder(a), k, v))
                caso k < raíz(a).clave hacer {debe insertarse en el subárbol izquierdo}
                    a := enraiza(asigna(hizq(a), k, v), raíz(a), hder(a))
            fopción
        fsi
    devuelve a
```

Fig. 5.51: implementación de las tablas ordenadas usando un árbol binario de búsqueda.

© Los autores, 1998; © Edicions UPC, 1998.

```
    función borra (a es tabla; k es clave) devuelve tabla es
    var temp es elem_tabla fvar
        si ¬vacío?(a) entonces {si el árbol es vacío, no es necesario hacer nada}
            opción
                caso raíz(a).clave = k hacer {hay cuatro casos posibles}
                    opción
                        caso vacío?(hizq(a)) ∧ vacío?(hder(a)) hacer {el nodo es una hoja}
                            a := crea
                        caso ¬vacío?(hizq(a)) ∧ vacío?(hder(a)) hacer {se sube el hijo izquierdo}
                            a := hizq(a)
                        caso vacío?(hizq(a)) ∧ ¬vacío?(hder(a)) hacer {se sube el hijo derecho}
                            a := hder(a)
                        caso ¬vacío?(hizq(a)) ∧ ¬vacío?(hder(a)) hacer {el nodo tiene dos hijos}
                                {se busca el menor de los mayores, temp}
                            temp := actual(principio(inorden(hder(a)))
                                {se sube a la raíz y se borra del subárbol donde reside}
                            a := enraiza(hizq(a), temp, borra(hder(a), temp.clave))
                    fopción
                caso raíz(a).clave < k hacer {es necesario suprimir en el subárbol derecho}
                    a := enraiza(hizq(a), raíz(a), borra(hder(a), k))
                caso k < raíz(a).clave hacer {es necesario suprimir en el subárbol izquierdo}
                    a := enraiza(borra(hizq(a), k), raíz(a), hder(a))
            fopción
        fsi
    devuelve a
    función consulta (a es tabla; k es clave) devuelve valor es
    var v es valor fvar
        si vacío?(a) entonces v := indef {la clave no está definida}
        si no
            opción
                caso raíz(a).clave = k hacer v := raíz(a).valor
                caso raíz(a).clave < k hacer v := consulta(hder(a), k)
                caso raíz(a).clave > k hacer v := consulta(hizq(a), k)
            fopción
        fsi
    devuelve v
    función todos_ordenados (T es tabla) devuelve lista_elem_tabla es
    devuelve ÁRBOL_BINARIO_CON_RECORRIDOS.inorden(T)
funiverso
```

Fig. 5.51: implementación de las tablas ordenadas usando un árbol binario de búsqueda.

© Los autores, 1998; © Edicions UPC, 1998.

5.6.2 Árboles AVL

Como ya se ha explicado, la eficiencia temporal de las operaciones de acceso individual a los elementos del árbol depende exclusivamente de su altura y, en caso de mala suerte, puede llegar a ser lineal. En este apartado se introduce una variante de árbol que asegura un coste logarítmico, ya que reduce el número de niveles de un árbol binario de n nodos a $\Theta(\log n)$, el mínimo posible.

Hay diversas técnicas que aseguran este coste logarítmico sin exigir que el árbol sea casi completo (lo que llevaría a algoritmos demasiado complicados y costosos). Por ejemplo, los denominados árboles 2-3 son una clase de árboles no binarios, sino ternarios, que obligan a que todas las hojas se encuentren al mismo nivel; su extensión a una aridad cualquiera son los árboles B y sus sucesores B^* y B^+, muy empleados en la implementación de ficheros indexados. En este texto, no obstante, estudiamos otra clase de árboles (estos sí, binarios) que se definen a partir de los conceptos y algoritmos hasta ahora introducidos: los árboles AVL (iniciales de sus creadores, G.M. Adel'son-Vel'skii y E.M. Landis, que los presentaron en el año 1962 en una publicación soviética), que son árboles de búsqueda equilibrados.

Diremos que un árbol binario está equilibrado (ing., height balanced o, simplemente, balanced) si el valor absoluto de la diferencia de alturas de sus subárboles es menor o igual que uno y sus subárboles también están equilibrados. Por ejemplo, los árboles izquierdo y derecho de la fig. 5.48 son equilibrados; el de la izquierda, porque es completo, y el de la derecha, porque sus desequilibrios no son lo bastante acusados como para romper la definición dada. En cambio, el árbol central de la misma figura es desequilibrado: por ejemplo, su subárbol izquierdo tiene altura 4 y el derecho 1.

Los árboles AVL aseguran realmente un coste logarítmico a sus operaciones de acceso individual; en [HoS94, p. 520-521] y [Wir86, pp. 218-219] se deduce este coste a partir de una formulación recursiva del número mínimo de nodos de un árbol AVL de altura h, que puede asociarse a la sucesión de Fibonacci. Los mismos Adel'son-Vel'skii y Landis demuestran que la altura máxima de un árbol AVL de n nodos está acotada por $\lceil 1.4404\log_2(n+2)-0.328 \rceil$ (aproximadamente 1.5logn), es decir, orden logarítmico; los árboles que presentan esta configuración sesgada se denominan árboles de Fibonacci, también debido al parecido con la sucesión correspondiente. Por lo que respecta al caso medio (el caso mejor es, obviamente, el árbol perfectamente completo), estudios empíricos apuntan a que la altura de un árbol AVL de n nodos, usando el algoritmo de inserción que se presenta a continuación, es del orden de $\lceil (\log n)+0.25 \rceil$, suponiendo que el orden de inserción de los elementos sea aleatorio [Wir86, pp. 223-224].

En la fig. 5.52 se muestra una representación de las tablas con árboles AVL implementados con punteros. Observamos que en este caso no reaprovechamos el TAD de los árboles binarios dado que es necesario guardar información sobre el equilibrio del árbol dentro de los

© Los autores, 1998; © Edicions UPC, 1998.

nodos. Concretamente se introduce un tipo por enumeración, que implementa el concepto de factor de equilibrio (ing., balance factor) que registra si un árbol AVL tiene el subárbol izquierdo con una altura superior en uno al subárbol derecho, si está perfectamente equilibrado o si el subárbol derecho tiene una altura superior en uno al subárbol izquierdo. El uso del factor de equilibrio en lugar de la altura simplifica la codificación de los algoritmos, porque evita distinguir subárboles vacíos. El invariante del tipo simplemente refuerza el invariante de los árboles de búsqueda con la condición de equilibrio, que tendrá que ser efectivamente mantenida por los algoritmos de inserción y de supresión; para definirla, se introduce una operación auxiliar que calcula la altura de un nodo según se definió en la sección 5.1. Notemos que no se incluyen las comprobaciones sobre los apuntadores que aparecen en los árboles binarios de la fig. 5.8, porque se pueden deducir de la condición de árbol de búsqueda. Para simplificar algoritmos posteriores, el árbol no se enhebra; este caso queda como ejercicio para el lector.

```
tipo tabla es árbol_AVL ftipo
tipo privado árbol_AVL es ^nodo ftipo
tipo privado equilibrio es (IZQ, PERFECTO, DER) ftipo
tipo privado nodo es
   tupla
      k es clave; v es valor
      hizq, hder son ^nodo
      equib es equilibrio  {factor de equilibrio del nodo}
   ftupla
ftipo
invariante (T es tabla): correcto(T),
   donde correcto: árbol_AVL → bool se define como:
      correcto(NULO) = cierto
      p ≠ NULO ⇒ correcto(p) = correcto(p^.hizq) ∧ correcto(p^.hder) ∧
         p^.hizq ≠ NULO ⇒ máximo(p^.hizq) < p^.k ∧
         p^.hder ≠ NULO ⇒ mínimo(p^.hder) > p^.k  ∧
         | altura(p^.hizq) - altura(p^.hder) | ≤ 1  ∧
         p^.equib = IZQ ⇔ altura(p^.hizq) > altura(p^.hder)  ∧
         p^.equib = PERFECTO ⇔ altura(p^.hizq) = altura(p^.hder)  ∧
         p^.equib = DER ⇔ altura(p^.hizq) < altura(p^.hder)
   y donde la función altura: árbol_AVL → nat se define como:
      altura(NULO) = 0
      p ≠ NULO ⇒ altura(p) = máx(altura(p^.hizq), altura(p^.hder)) + 1,
   y máximo y mínimo se definen como en la fig. 5.51
```

Fig. 5.52: representación del tipo de las tablas ordenadas con árboles AVL.

© Los autores, 1998; © Edicions UPC, 1998.

a) Inserción en un árbol AVL

La inserción en un árbol AVL consta de dos etapas diferenciadas: por un lado, es necesario aplicar la casuística de la inserción "normal" para conservar la propiedad de los árboles de búsqueda y, por el otro, es necesario asegurar que el árbol queda equilibrado, reestructurándolo si es necesario. La primera tarea ya ha sido descrita; nos centramos, pues, en la segunda.

Sean a un árbol AVL y <k, v> el par clave-valor a insertar; si la clave k ya estaba dentro de a, o bien el nodo que se añade según el algoritmo de inserción no provoca ningún desequilibrio, el proceso acaba sin más problemas. El desequilibrio se produce cuando existe un subárbol a' de a, que se encuentra en cualquiera de los dos casos siguientes:

- El subárbol derecho de a' tiene una altura superior en uno[15] al subárbol izquierdo de a', y el nodo correspondiente al par <k, v> se inserta en el subárbol derecho de a' y, además, provoca un incremento en uno[16] de su altura (v. fig. 5.53, izquierda).
- El subárbol izquierdo de a' tiene una altura superior en una unidad al subárbol derecho de a', y el nodo correspondiente al par <k, v> se inserta en el subárbol izquierdo de a' y, además, provoca un incremento en uno de su altura (v. fig. 5.53, derecha).

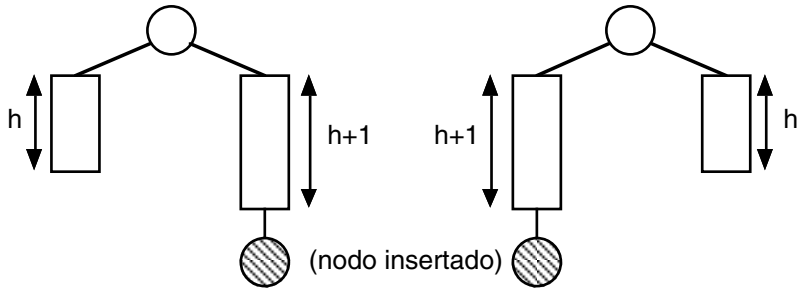


Fig. 5.53: los dos casos posibles de desequilibrio de la inserción en un árbol AVL.

A continuación se estudian las medidas que es necesario tomar para reequilibrar el árbol en estas situaciones. Notemos que ambas son simétricas, razón por la que nos centraremos sólo en la primera, y queda como ejercicio la extensión al segundo caso. Hay dos subcasos:

- Caso DD (abreviatura de Derecha-Derecha; del inglés RR, abreviatura de Right-Right): el nodo se inserta en el subárbol derecho del subárbol derecho de a'. En la fig. 5.54 se muestra este caso (las alturas de β y γ son las únicas posibles, una vez fijada la altura de α) y su solución, que es muy simple: la raíz B del subárbol derecho de a' pasa a ser la nueva raíz del subárbol y conserva su hijo derecho, que es el que ha provocado el

[15] Nunca puede ser superior en más de una unidad, porque a es un árbol AVL antes de la inserción.

[16] El incremento no puede ser superior a una unidad, porque se inserta un único nodo.

© Los autores, 1998; © Edicions UPC, 1998.

desequilibrio con su incremento de altura y que, con este movimiento, queda a un nivel más cerca de la raíz del árbol, compensando el aumento; la antigua raíz A de a' pasa a ser hijo izquierdo de B y conserva su subárbol izquierdo; finalmente, el anterior subárbol izquierdo de B pasa a ser subárbol derecho de A para conservar la propiedad de ordenación de los árboles de búsqueda. Algunos autores denominan rotaciones a estos movimentos de subárboles. Se puede comprobar que la rotación mantiene la ordenación correcta recorriendo inorden el árbol antes y después del proceso. En la fig. 5.54 confirmamos que, efectivamente, el recorrido es idéntico, αAβBγ, tanto en el árbol de la izquierda como en el de la derecha.

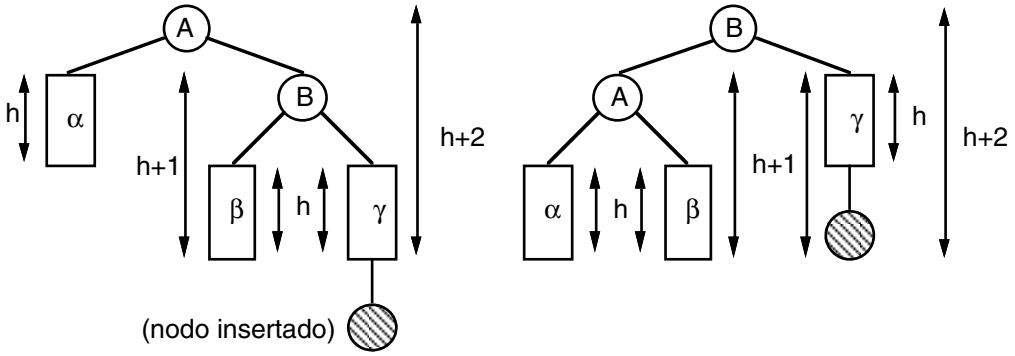


Fig. 5.54: árbol con desequilibrio DD (a la izquierda) y su resolución (a la derecha).

Notemos que la altura del árbol resultante es la misma que tenía el árbol antes de la inserción. Esta propiedad es importantísima, porque asegura que basta con un único reequilibrio del árbol para obtener un árbol AVL después de la inserción, siempre que la búsqueda del primer subárbol que se desequilibra se haga siguiendo el camino que va de la nueva hoja a la raíz. En el momento en que se reequilibra este subárbol, el resto del árbol queda automáticamente equilibrado, porque ya lo estaba antes de la inserción y su altura no varía. El árbol resultante queda incluso "más equilibrado" que antes, en el sentido de que el proceso de reequilibrio iguala las alturas de los dos subárboles de a'.

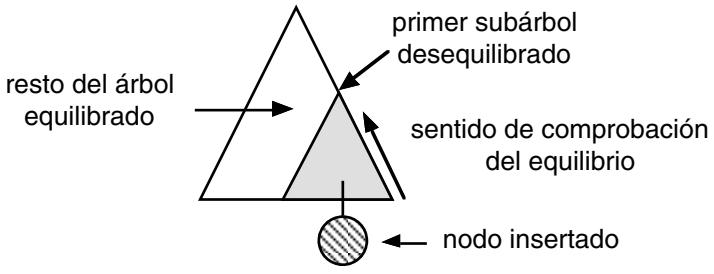


Fig. 5.55: proceso de equilibrado del árbol.

© Los autores, 1998; © Edicions UPC, 1998.

- Caso DI (abreviatura de Derecha-Izquierda; del inglés RL, abreviatura de Right-Left): el nodo se inserta en el subárbol izquierdo del subárbol derecho de a'. En la fig. 5.57 se muestra este caso (las alturas de los subárboles de la derecha de la raíz son fijadas a partir de la altura de α) y su solución, que no es tan evidente como antes, porque aplicando las mismas rotaciones de subárboles no se solucionaría el desequilibrio, por lo que es necesario descomponer también el 21-subárbol de a'. Precisamente por este motivo es necesario distinguir el caso trivial en que el 21-subárbol de a' sea vacío y no se pueda descomponer (v. fig. 5.56). La rotación DI se puede considerar como la composición de dos rotaciones: la primera, una rotación II (simétrica de la DD) del subárbol derecho de a' y la segunda, una rotación DD del subárbol resultante. Notemos que el nuevo nodo puede ir a parar indistintamente a cualquiera de los dos subárboles que cuelgan del 21-subárbol de a' sin que afecte a las rotaciones definidas. También aquí la altura después de la rotación es igual a la altura previa a la inserción.

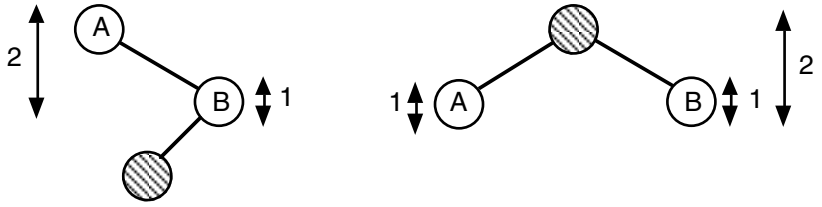


Fig. 5.56: caso trivial de desequilibrio DI (a la izquierda) y su resolución (a la derecha).

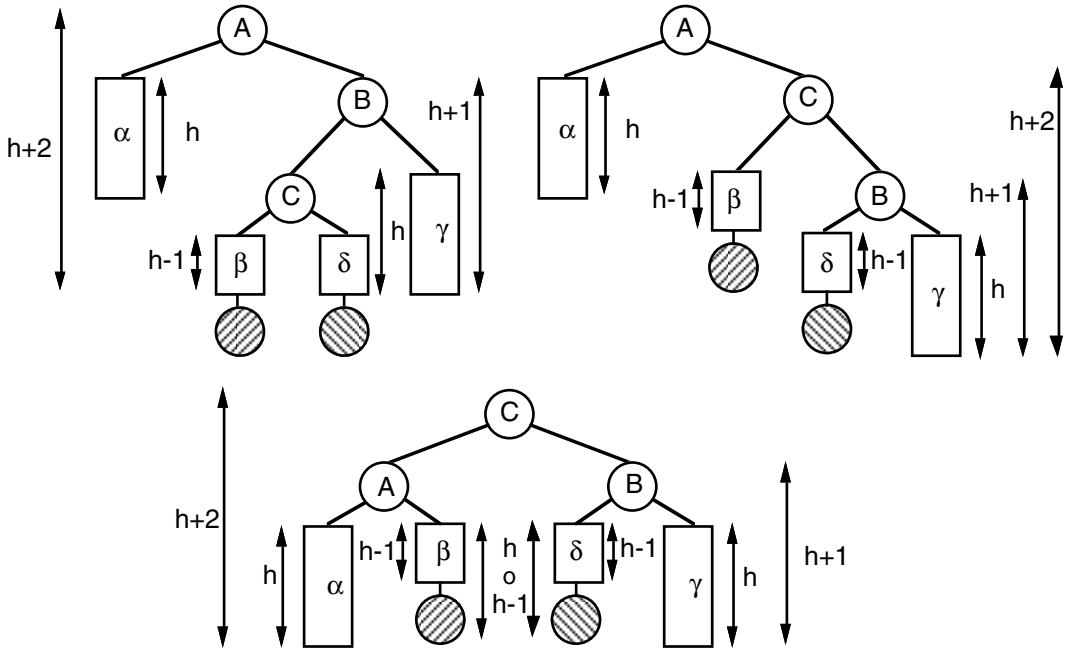


Fig. 5.57: árbol con desequilibrio DI (arriba, izquierda), rotación II sobre el subárbol izquierdo (arriba, derecha) y rotación DD sobre el árbol entero (abajo).

© Los autores, 1998; © Edicions UPC, 1998.

Existen diversos estudios empíricos que intentan establecer la relación entre el número de rotaciones exigidas para una secuencia dada de inserciones [Wir86, p. 227]. Los resultados apuntan que, dada una secuencia aleatoria de inserciones en un árbol AVL, se necesita una rotación para cada dos inserciones, siendo los dos tipos de rotaciones casi equiprobables.

En la fig. 5.59 se muestra la codificación recursiva del algoritmo; la versión iterativa queda como ejercicio para el lector. La tarea más importante recae sobre una función recursiva auxiliar, inserta_AVL, y los casos terminales de la recursividad son crear una nueva hoja o bien encontrar algún nodo en el árbol que contenga la clave. En el primer caso, y después de hacer la inserción del nodo siguiendo la misma estrategia que en la fig. 5.51, se estudia el equilibrio desde la hoja, avanzando a la raíz hasta que se llega a ella, o se encuentra algún subárbol que no crece, o bien se encuentra algún subárbol que se desequilibra; este estudio es sencillo usando el factor de equilibrio del nodo. Para equilibrar el árbol, se usan dos operaciones auxiliares más, una de las cuales también se codifica en la figura, que no hacen más que implementar las rotaciones que se acaban de describir como modificaciones del valor de los encadenamientos. Destaquemos que el árbol es un parámetro de entrada y de salida para asegurar que los encadenamientos queden realmente actualizados.

b) Supresión en un árbol AVL

Los dos casos posibles de desequilibrio en la supresión son idénticos al proceso de inserción, pero ahora el desequilibrio se produce porque la altura de un subárbol disminuye por debajo del máximo tolerado. Una vez más, nos centramos en el desequilibrio provocado por la supresión en el subárbol izquierdo; la otra situación es simétrica. Los diferentes algoritmos de rotación que se necesitan dependen exclusivamente de la relación de las alturas de los dos subárboles del subárbol derecho de la raíz (que, a diferencia de lo que ocurría al insertar, nunca pueden ser vacíos):

- Si son iguales, se produce el desequilibrio DD del caso de la inserción, que se resuelve de la misma forma (v. fig. 5.58). El árbol resultante tiene la misma altura antes y después de la supresión, por lo que basta con esta rotación para reestablecer el equilibrio.

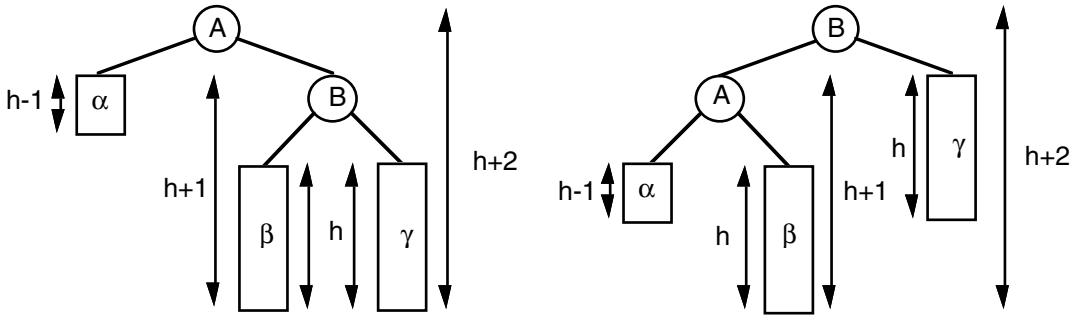


Fig. 5.58: árbol con desequilibrio DD (a la izquierda) y su resolución (a la derecha).

© Los autores, 1998; © Edicions UPC, 1998.

```
función inserta  (a es tabla; k es clave; v es valor) devuelve tabla es
var res es bool fvar
   inserta_AVL(a, k, v, res)
devuelve a

{Función auxiliar inserta_AVL(a, k, v, crece?): dado un árbol AVL a,  inserta el par <k, v>, de
 manera que se conserva el invariante del tipo; crece? indica si la altura del árbol aumenta}
acción privada inserta_AVL (ent/sal a es árbol_AVL; ent k es clave; ent v es valor;
                            sal crece? es bool) es
   si a = NULO entonces {se crea un árbol de un único nodo}
      a := obtener_espacio
      si a = NULO entonces error
      si no
         a := <k, v, NULO, NULO, PERFECTO>
         crece? := cierto
      fsi
   si no
      opción
         caso a^.k = k hacer  a^.v := v; crece? := falso
         caso a^.k > k hacer
            inserta_AVL(a^.hizq, k, v, crece?)   {inserción recursiva por la izquierda}
            si crece? entonces   {estudio del equilibrio y rotaciones si es necesario}
               opción
                  caso a^.equib = DER hacer a^.equib := PERFECTO; crece? := falso
                  caso a^.equib = PERFECTO hacer a^.equib := IZQ; crece? := cierto
                  caso a^.equib = IZQ hacer a := rotación_izq(a); crece? := falso
               fopción
            fsi
         caso a^.k < k hacer  {simétrico al anterior}
            inserta_AVL(a^.hder, k, v, crece?)
            si crece? entonces
               opción
                  caso a^.equib = IZQ hacer a^.equib := PERFECTO; crece? := falso
                  caso a^.equib = PERFECTO hacer a^.equib := DER; crece? := cierto
                  caso a^.equib = DER hacer a := rotación_der(a); crece? := falso
               fopción
            fsi
      fopción
   fsi
facción
```

Fig. 5.59: algoritmo de inserción en un árbol AVL.

© Los autores, 1998; © Edicions UPC, 1998.

```
{Función auxiliar rotación_derecha: dado un subárbol a donde todos sus
 subárboles son AVL, pero a está desequilibrado por la derecha, investiga la razón
 del desequilibrio y efectúa las rotaciones oportunas}
función privada rotación_derecha (a es árbol_AVL) devuelve árbol_AVL es
var raíz, b, beta, delta son ^nodo ftupla
        {raíz apuntará a la nueva raíz del árbol}
    si (a^.hizq = NULO) ∧ (a^.hder^.hder = NULO) entonces {caso trivial de la fig. 5.56}
        b := a^.hder; raíz := b^.hizq
        raíz^.hizq := a; raíz^.hder  := b; b^.hizq := NULO; a^.hder := NULO
        a^.equib := PERFECTO; b^.equib := PERFECTO
    si no  {es necesario distinguir tipo de desequilibrio y rotar en consecuencia}
        si a^.hder^.equib = DER entonces {desequilibrio DD, v. fig. 5.54}
            raíz := a^.hder; beta := raíz^.hizq
            raíz^.hizq := a; a^.hder := beta; a^.equib := PERFECTO
        si no {desequilibrio DI, v. fig. 5.57}
            b := a^.hder; raíz := b^.hizq; beta := raíz^.hizq; delta := raíz^.hder
            a^.hder := beta; raíz^.hizq := a; b^.hizq := delta; raíz^.hder := b
            si raíz^.equib = IZQ entonces  {el nuevo elemento está dentro de beta}
                a^.equib := PERFECTO; b^.equib := DER
            si no  {el nuevo elemento es dentro de delta}
                a^.equib := IZQ; b^.equib := PERFECTO
            fsi
        fsi
    fsi
    raíz^.equib := PERFECTO
devuelve raíz
```

Fig. 5.59: algoritmo de inserción en un árbol AVL (cont.).

© Los autores, 1998; © Edicions UPC, 1998.

- Si la altura del subárbol izquierdo es menor que la altura del subárbol derecho, la rotación es exactamente la misma; ahora bien, la altura del árbol resultante es una unidad más pequeña que antes de la supresión. Este hecho es significativo, porque obliga a examinar si algún subárbol que lo engloba también se desequilibra.

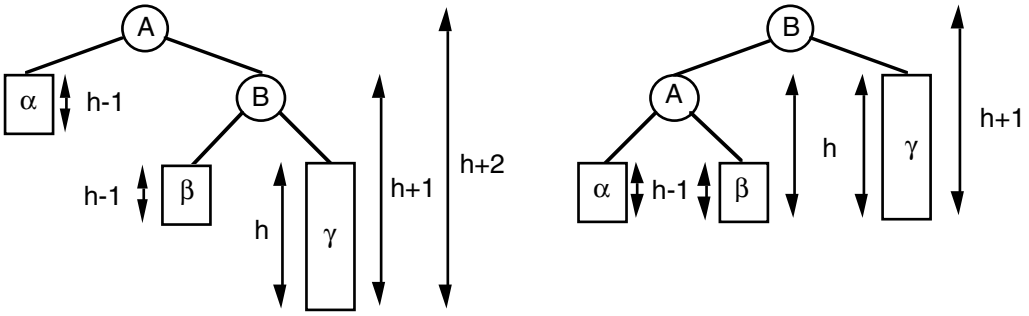


Fig. 5.60: árbol con desequilibrio DD (izq.) y su resolución (der.) con altura variable.

- Si la altura del subárbol izquierdo es mayor que la altura del subárbol derecho, la rotación es similar al caso DI (v. fig. 5.61); también aquí la altura del árbol resultante es una unidad más pequeña que antes de la supresión. Los árboles β y γ de la figura pueden tener altura h-1 ó h-2, pero al menos uno de ellos tiene altura h-1.

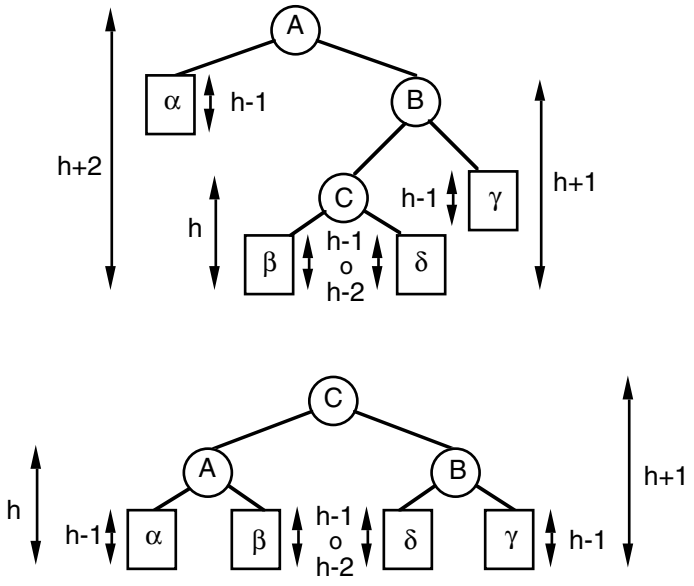


Fig. 5.61: árbol con desequilibrio DI (arriba) y su resolución (abajo).

© Los autores, 1998; © Edicions UPC, 1998.

Igual que sucedía en la inserción, se han hecho estudios empíricos sobre el número de rotaciones exigidas durante una secuencia de supresiones y, curiosamente, los resultados indican que sólo es necesaria una rotación por cada cinco supresiones [Wir86, p. 227].

El algoritmo de supresión se implementa recursivamente en la fig. 5.62. Su similitud con la operación de inserción es evidente, por lo que no se comenta más que aquello estrictamente imprescindible. Queda claro que la supresión de un elemento puede exigir tantas rotaciones como nodos haya en el camino provinente de la raíz, porque después de cada vuelta de una llamada recursiva puede haber reorganizaciones. Se usa una función auxiliar para borrar el elemento que aplica la casuística de la supresión en un árbol de búsqueda y comprueba el equilibrio en el caso general; esta función se podría escribir más compacta (v. [Wir86, pp.226-227]), pero se ha expandido por motivos de legibilidad. Una vez más, hay una ineficiencia temporal, porque en el caso de mover elementos se recorre un mismo camino dos veces; de todos modos, el coste asintótico es igualmente logarítmico; también, la codificación cambia la dirección física de los datos en las mismas condiciones que la supresión en árboles binarios de búsqueda.

```
función borra  (a es tabla; k es clave) devuelve tabla es
var res es bool fvar
   borra_AVL(a, k, res)
devuelve a

{Función borra_AVL(a, k, v, encoge?): dado el árbol AVL a, borra el nodo que tiene como
 clave k, conservando el invariante del tipo; encoge? indica si la altura disminuye}
acción priv borra_AVL (ent/sal a es árbol_AVL; ent k es clave; sal encoge? es bool) es
   si a = NULO entonces encoge? := falso  {no hay ningún nodo de clave k}
   si no opción
            caso a^.k = k hacer    {se borra el nodo, controlando posibles desequilibrios}
               <a, encoge?> := borra_nodo(a)
            caso a^.k > k hacer
               borra_AVL(a^.hizq, k, encoge?)  {se sigue la rama adecuada}
                    {estudio del equilibrio y rotaciones si es necesario}
               si encoge? entonces <a, encoge?> := equilibra_derecha(a) fsi
            caso a^.k < k hacer  {simétrico al anterior}
               borra_AVL(a^.hder, k, encoge?)
               si encoge? entonces <a, encoge?> := equilibra_izquierda(a) fsi
         fopción
   fsi
facción
```

Fig. 5.62: algoritmo de supresión en un árbol AVL.

© Los autores, 1998; © Edicions UPC, 1998.

```
{Función auxiliar equilibra_derecha(a): dado un árbol AVL a  donde se ha borrado un
 nodo del subárbol derecho provocando una disminución de altura, averigua si el árbol
 se ha desequilibrado y, en caso afirmativo, efectúa las rotaciones oportunas. Además,
 devuelve un booleano que indica si la altura del árbol ha disminuido}
función privada equilibra_derecha (a es árbol_AVL) devuelve <árbol_AVL, bool> es
var encoge? es bool fvar
    opción
        caso a^.equib = IZQ hacer a^.equib := PERFECTO; encoge? := cierto
        caso a^.equib = PERFECTO hacer a^.equib := DER; encoge? := falso
        caso a^.equib = DER hacer <a, encoge?> := rotación_derecha(a)
    fopción
devuelve <a, encoge?>

{Función auxiliar rotación_derecha: dado un subárbol a  en que todos los subárboles
 son AVL, pero a está desequilibrado por la derecha, investiga la razón del desequilibrio
 y efectúa las rotaciones oportunas. Además, devuelve un booleano que indica si la
 altura del árbol ha disminuido}

función privada rotación_derecha (a es árbol_AVL) devuelve <árbol_AVL, bool> es
var raíz, b, beta, delta son ^nodo; encoge? es bool fvar
            {raíz  apuntará a la nueva raíz del árbol}
   si a^.hder^.equib = IZQ entonces {desequilibrio DI, v. fig. 5.61}
        b := a^.hder; raíz := b^.hizq; beta := raíz^.hizq; delta := raíz^.hder
        a^.hder := beta; raíz^.hizq := a; b^.hizq := delta; raíz^.hder := b
        opción   {se actualizan los factores de equilibrio según las alturas de beta  y  delta}
            caso raíz^.equib = IZQ hacer a^.equib := PERFECTO; b^.equib := DER
            caso raíz^.equib = PERFECTO hacer
                        a^.equib := PERFECTO; b^.equib := PERFECTO
            caso raíz^.equib = DER hacer a^.equib := IZQ; b^.equib := PERFECTO
        fopción
        raíz^.equib := PERFECTO; encoge? := cierto
   si no {desequilibrio DD, v. fig. 5.58 y 5.60, de idéntico tratamiento}
        raíz := a^.hder; beta := raíz^.hizq; raíz^.hizq := a; a^.hder := beta
           {a continuación, se actualizan los factores de equilibrio según las alturas de beta
            y el hijo derecho de la nueva raíz, y se indaga si el árbol ha encogido}
        si b^.equib = PERFECTO
            entonces a^.equib := DER; raíz^.equib := IZQ; encoge? := falso
            si no a^.equib := PERFECTO; raíz^.equib := PERFECTO; encoge? := cierto
        fsi
   fsi
devuelve raíz
```

Fig. 5.62: algoritmo de supresión en un árbol AVL (cont.).

© Los autores, 1998; © Edicions UPC, 1998.

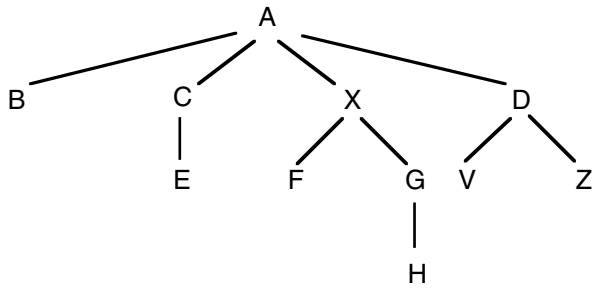
```
{Función auxiliar borra_nodo(a): dado un árbol a en que la raíz contiene el elemento
 que se quiere borrar, lo suprime según la casuística vista en el primer apartado y libera
 espacio; en caso de que existan los dos subárboles de a, controla el equilibrio y rota
 si es necesario. Además, devuelve un booleano que indica si la altura del árbol ha
 disminuido}

función privada borra_nodo (a es árbol_AVL) devuelve <árbol_AVL, bool> es
var raíz, min son ^nodo; encoge? es bool fvar
        {raíz apunta a la raíz del subárbol resultante}
    opción
        caso (a^.hizq = NULO) ∧ (a^.hder = NULO) hacer
                {es una hoja; la altura disminuye}
            raíz := NULO; encoge? := cierto
            liberar_espacio(a)
        caso (a^.hizq ≠ NULO) ∧ (a^.hder = NULO) hacer
                {le cuelga un único nodo por la izquierda, que sube; la altura disminuye}
            raíz := a^.hizq; encoge? := cierto
            liberar_espacio(a)
        caso (a^.hizq = NULO) ∧ (a^.hder ≠ NULO) hacer
                {le cuelga un único nodo por la derecha, que sube; la altura disminuye}
            raíz := a^.hder; encoge? := cierto
            liberar_espacio(a)
        caso (a^.hizq ≠ NULO) ∧ (a^.hder ≠ NULO) hacer
                {obtiene y copia el mínimo del subárbol derecho a la raíz}
            min := a^.hder
            mientras min^.hizq ≠ NULO hacer min := min^.hizq fmientras
            a^.k := min^.k; a^.v := min^.v
                {a continuación, borra el mínimo y controla el equilibrio}
            borra_AVL(a^.hder, min^.k, encoge?)
            si encoge? entonces <a, encoge?> := equilibra_derecha(a) fsi
    fopción
devuelve <raíz, encoge?>
```

Fig. 5.62: algoritmo de supresión en un árbol AVL (cont.).

© Los autores, 1998; © Edicions UPC, 1998.

Ejercicios

5.1 Dado el árbol siguiente:



a) describirlo según el modelo asociado a los árboles generales; **b)** decir cuál es el nodo raíz y cuáles son las hojas; **c)** decir qué nodos son padres, hijos, antecesores y descendientes del nodo X ; **d)** calcular el nivel y la altura del nodo X; **e)** recorrerlo en preorden, inorden, postorden y por niveles; **f)** transformarlo en un árbol binario que lo represente según la estrategia "hijo izquierdo, hermano derecho".

5.2 Especificar ecuacionalmente el modelo de los árboles con punto de interés tanto binarios como generales, con un conjunto apropiado de operaciones.

5.3 Usando la signatura de los árboles binarios, especificar e implementar una operación que cuente el número de hojas.

5.4 Una expresión aritmética puede representarse como un árbol donde los nodos que son hojas representen operandos y el resto de nodos representen operadores.

a) Dibujar un árbol representando la expresión (a+b)*c/d*(a-5).

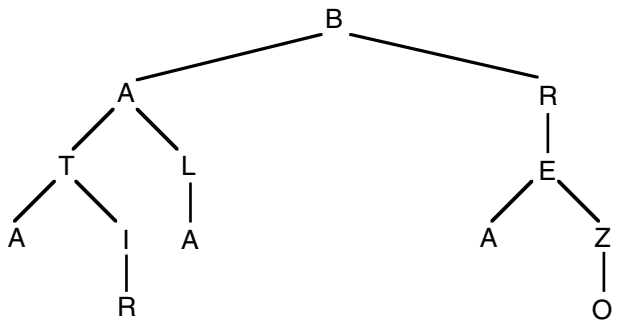
b) ¿Cuál sería el algoritmo de evaluación de la expresión? ¿Se corresponde a algún algoritmo conocido sobre árboles?

c) Escribir un algoritmo que transforme una expresión representada mediante una cola de símbolos en una expresión representada mediante un árbol. Suponer que los símbolos tienen una operación para averiguar si son operandos, operadores, paréntesis de abrir o paréntesis de cerrar. Considerar las prioridades habituales y la asociatividad por la izquierda. (Una variante de este problema está resuelta en el apartado 7.1.1, usando estructuras lineales.)

5.5 Escribir un algoritmo que transforme un árbol general implementado con apuntadores a los hijos en un árbol binario por la estrategia hijo izquierdo, hermano derecho. Hacer también el algoritmo inverso.

© Los autores, 1998; © Edicions UPC, 1998.

5.6 Sean los árboles generales con la signatura habitual. Implementar un procedimiento que escriba todos los caminos que van de la raíz a las hojas de un árbol de letras. Por ejemplo, del árbol:



han de salir los caminos BATA, BATIR, BALA, BREA, BREZO.

5.7 Sea un árbol general A y un árbol binario A' resultado de representar A bajo la estrategia hijo izquierdo, hermano derecho. Decir si hay alguna relación entre los recorridos de A y A'. Justificar la conveniencia de enhebrar o no el árbol binario (v. [Knu68, pp. 361-363]).

5.8 a) ¿Es posible generar un árbol binario a partir únicamente de uno de sus recorridos preorden, inorden o postorden? ¿Y a partir de dos recorridos diferentes (examinar todos los pares posibles)? ¿Por qué?

b) Reconstruir un árbol binario a partir de los recorridos siguientes:

i) preorden: 2, 1, 4, 7, 8, 9, 3, 6, 5
inorden: 7, 4, 9, 8, 1, 2, 6, 5, 3

ii) inorden: 4, 6, 5, 1, 2, 12, 7, 3, 9, 8, 11, 10
postorden: 6, 5, 4, 12, 7, 2, 8, 9, 10, 11, 3, 1

c) Diseñar el algoritmo que reconstruya un árbol binario dados sus recorridos preorden e inorden. Hacer lo mismo a partir de los recorridos postorden e inorden. En ambos casos usar el tipo lista_nodos para representar los recorridos, defininiendo claramente su signatura. Si se necesita alguna operación muy particular de este ejercicio, definirla claramente, especificarla e implementarla.
d) Escribir un algoritmo que, dados dos recorridos preorden, inorden o postorden de un árbol binario y dos nodos suyos cualesquiera, decida si el primero es antecesor del segundo.

5.9 Escribir un algoritmo que transforme un árbol binario representado con apuntadores a los hijos en un árbol binario representado secuencialmente en preorden con apuntador al hijo derecho (es decir, los nodos dentro de un vector almacenados en el orden dado por un recorrido preordren y, para cada uno de ellos, un apuntador adicional al hijo derecho).

5.10 Escribir algoritmos para añadir un nodo como hijo izquierdo o derecho de otro dentro

© Los autores, 1998; © Edicions UPC, 1998.

de un árbol binario enhebrado inorden, y borrar el hijo izquierdo o derecho de un nodo.

5.11 Modificar la implementación de las relaciones de equivalencia de la fig. 5.35 para el caso particular en que los elementos sean los n naturales del intervalo [1, n].

5.12 Construir un árbol parcialmente ordenado, insertando sucesivamente los valores 64, 41, 10, 3, 9, 1 y 2, y usando < como relación de orden. A continuación, borrar tres veces el mínimo. Mostrar claramente la evolución del árbol paso a paso en cada operación.

5.13 Sea una nueva función sobre las colas prioritarias que borre un elemento cualquiera dada una clave que lo identifique. Implementarla de manera que tenga un coste logarítmico sin empeorar las otras (si es necesario, modificar la representación habitual del tipo).

5.14 Se quieren guardar 10.000 elementos dentro de una cola prioritaria implementada con un montículo que representa el árbol parcialmente ordenado correspondiente. Determinar en los casos siguientes si es mejor un árbol binario o uno cuaternario:
a) Minimizando el número máximo de comparaciones entre elementos al insertar uno nuevo.
b) Minimizando el número máximo de comparaciones entre elementos al borrar el mínimo.
c) Minimizando el número máximo de movimientos de elementos al insertar uno nuevo.
d) Minimizando el número máximo de movimientos de elementos al borrar el mínimo.

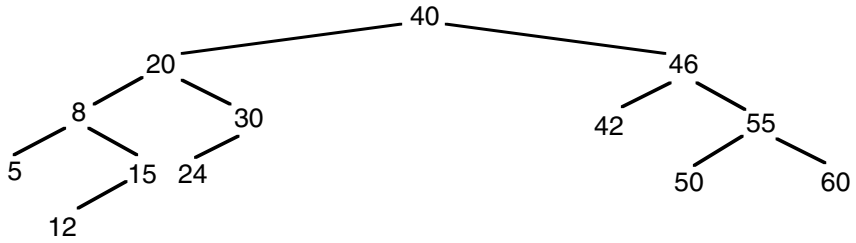
5.15 Implementar un árbol parcialmente ordenado siguiendo la estrategia encadenada.

5.16 Espacio intencionadamente en blanco.

5.17 Dibujar todos los árboles de búsqueda posibles que contengan los naturales 1, 2, 3 y 4 (sin repeticiones). ¿Cuáles de estos árboles son equilibrados? En general, ¿cuántos árboles de búsqueda se pueden formar con n elementos diferentes?

5.18 Dado un árbol de búsqueda inicialmente vacío, insertar los elementos 7, 2, 9, 0, 5, 6, 8 y 1, equilibrando a cada paso.

5.19 Dado el árbol de búsqueda de la figura:



insertar sucesivamente los valores 64, 41, 10, 3, 9, 1 y 2. A continuación, borrar

© Los autores, 1998; © Edicions UPC, 1998.

sucesivamente los valores 9, 15, 10 y 55; en caso de borrar un nodo con dos hijos, sustituirlo por el menor de su subárbol derecho. Repetir el proceso equilibrando el árbol después de cada modificación (notar que, inicialmente, ya está equilibrado). Mostrar claramente la evolución del árbol paso a paso en cada operación.

5.20 Proponer una representación de los árboles de búsqueda equilibrados que, además de las operaciones habituales, permita encontrar el k-ésimo elemento menor del árbol en tiempo logarítmico, sin afectar el coste de las operaciones habituales y empleando el mínimo espacio adicional que sea posible. Codificar la operación.

5.21 Proponer una estructura de datos que tenga tiempo logarítmico en la inserción y supresión de elementos y tiempo constante en la consulta de elementos, y que permita listar todos los elementos ordenados en un tiempo lineal.

5.22 Implementar el TAD de las colas prioritarias usando árboles de búsqueda.

5.23 En lógica, se dice que un predicado (que es una expresión formada por variables y operaciones booleanas; por ejemplo, cierto, falso, $\wedge$, $\vee$ y $\neg$) se satisface si, para alguna asignación de sus variables, el predicado evalúa cierto. A las variables se les pueden asignar los valores cierto y falso y la evaluación de un predicado es la habitual. Pensar una buena representación de los predicados y un algoritmo que decida si un predicado se satisface. Calcular el coste del algoritmo en tiempo y espacio.

5.24 Interesa encontrar una representación de los conjuntos que, aparte de las típicas operaciones de añadir y sacar elementos y de comprobar si un elemento está dentro de un conjunto, ofrezca otras tres que obtengan una lista con la intersección, la unión y la diferencia de dos conjuntos. ¿Cuál es la mejor estructura para favorecer las tres últimas operaciones? ¿Y si, además, queremos que las tres primeras también sean rápidas? ¿Qué ocurre si las tres últimas, en vez de devolver una lista de elementos, han de devolver también un conjunto? Justificar las respuestas en función del coste de las operaciones.

5.25 Se quiere implementar un diccionario con operaciones de acceso directo por palabra y de recorrido alfabético. Suponer que el espacio reservado para cada entrada del diccionario es X, un valor fijo. Suponer que el número esperado de entradas es N. Razonar qué estructura de datos es mejor en los siguientes supuestos: **a)** minimizando el espacio que ocupa el diccionario; **b)** favoreciendo el coste temporal de las operaciones de recorrido alfabético y de consulta directa por palabra; **c)** favoreciendo el coste temporal de las operaciones de recorrido alfabético y de actualización del diccionario (inserciones y supresiones). En cada caso determinar exactamente el coste temporal de las diferentes operaciones del diccionario, así como también el espacio usado (en bits y como función de X y N), suponiendo que los enteros y los punteros ocupan 32 bits.

© Los autores, 1998; © Edicions UPC, 1998.

5.26 Dado un fichero con n números enteros, n muy grande (por ejemplo, $n = 10^8$), construir un algoritmo que obtenga los k nombres más grandes, $k << n$ (por ejemplo, $k = 1000$) con la mayor eficiencia posible tanto en tiempo como en espacio. Calcular cuidadosamente su coste.

5.27 La Federación Local de Ajedrez de CAtalunya (FLACA) ha decidido informatizarse. La FLACA tiene registrados los nombres de los diferentes clubes y jugadores de ajedrez de Catalunya, y sabe que un jugador sólo pertenece a un club y que, a causa de la incipiente crisis económica, los clubes se fusionan con cierta frecuencia para formar otros nuevos. El nombre del club resultante es igual al nombre del club que tenga más ajedrecistas y los jugadores de los clubes que se fusionan pasan a ser automáticamente del nuevo club. Determinar la signatura y la implementación de la estructura necesaria para que se puedan fusionar clubes, y a qué club pertenece un jugador concreto con cierta rapidez. ¿Cuál es el coste resultante de las operaciones? ¿Y el espacio empleado?

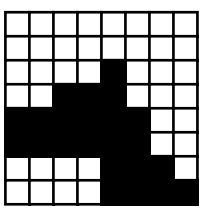
5.28 A causa de una intensa campaña institucional y de las exenciones fiscales, el número de clubes de ajedrez en Catalunya se ha disparado, es muy grande y continúa creciendo. Para organizar futuros campeonatos, la FLACA clasifica los clubes por comarcas (considerar que hay un número pequeño y fijo de comarcas) de manera que, al dar de alta un nuevo club, se dice de qué comarca es. Diseñar una estructura de datos para que sea lo más eficiente posible al sacar los listados, ordenados alfabéticamente, de todos los clubes de Catalunya y de todos los clubes de una comarca dada, y que la operación de añadir un nuevo club no sea excesivamente lenta. En todos los casos, calcular el coste de las operaciones y justificar que la solución expuesta no es mejorable. ¿Cómo se implementaría la operación de listar ordenadamente todos los clubes de una comarca?

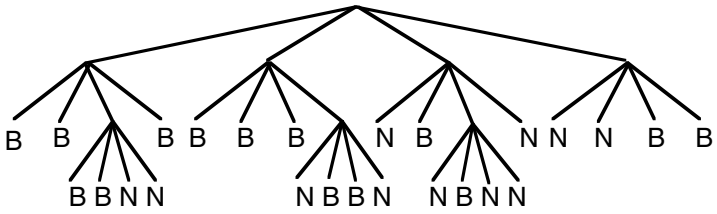
5.29 Nos ocupamos de la construcción de una familia de códigos de compresión de cadenas denominados códigos de Huffman. Suponer que tenemos mensajes que consisten en una secuencia de caracteres. En cada mensaje los caracteres aparecen con una probabilidad conocida, independiente de la posición concreta del carácter dentro del mensaje. Por ejemplo, suponer que los mensajes se componen de los caracteres a, b, c, d y e, que aparecen con probabilidades 0.12, 0.40, 0.15, 0.08 y 0.25, respectivamente (obviamente, la suma da 1). Queremos codificar cada carácter en una secuencia de ceros y unos de manera que ningún código de un carácter sea prefijo del código de otro; esta propiedad permite, dado un mensaje, codificarlo y descodificarlo de manera no ambigua. Además, queremos que esta codificación minimice la extensión media esperada de los mensajes, de manera que su compactación sea óptima. En el ejemplo anterior, un código óptimo es a = 1111, b = 0, c = 110, d = 1110 y e = 10. En concreto:
a) Diseñar un algoritmo tal que, dado el conjunto V de caracteres y dada la función de probabilidad $pr : V \rightarrow [0,1]$, construya un código de Huffman para V.
b) Escribir los algoritmos de codificación y descodificación de mensajes suponiendo que el tipo mensaje es una instancia adecuada del tipo cola.

© Los autores, 1998; © Edicions UPC, 1998.

5.30 La estructura de datos quad-tree se usa en informática gráfica para representar figuras planas en blanco y negro. Se trata de un árbol en el cual cada nodo, o bien tiene exactamente cuatro hijos, o bien es una hoja. En este último caso, puede ser una hoja blanca o una hoja negra.

El árbol asociado a una figura dibujada dentro de un plano (que, para simplificar, podemos suponer un cuadrado de lado 2^k) se construye de la forma siguiente: se subdivide el plano en cuatro cuadrantes; los cuadrantes que estén completamente dentro de la figura corresponderán a hojas negras, los que esten completamente fuera de la región, a hojas blancas y los que estén parcialmente dentro y parcialmente fuera, a nodos internos; para estos últimos se aplica recursivamente el mismo algoritmo. Como ejemplo, se muestra una figura en blanco y negro y su árbol asociado (considerando los cuadrantes en el sentido de las agujas del reloj, a partir del cuadrante superior izquierdo):





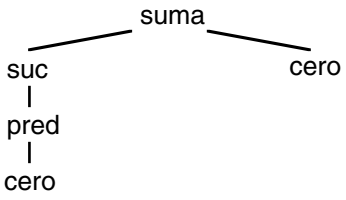
a) Determinar la signatura necesaria sobre el tipo quad-tree que permita algoritmos para representar una figura como un quad-tree y para recuperar la figura a partir de un quad-tree. Suponer que existe un tipo figura con las operaciones que más convengan.

b) Escribir una representación para los quad-trees.

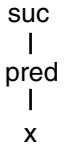
c) Implementar las operaciones del tipo obtenidas en a) sobre la representación de b).

d) Modificar la representación de b) para implementar una nueva operación que obtenga el negativo de un quad-tree. El resultado ha de ser $\Theta(1)$.

5.31 Ya sabemos que, en el contexto de las especificaciones algebraicas, se define un término como la aplicación sucesiva de símbolos de operación de una signatura, que puede tener variables o no tenerlas; gráficamente, un término se puede representar mediante un árbol con símbolos dentro de los nodos.



suma(suc(pred(cero)), cero)
Término sin variables



suc(pred(x))
Término con variables

© Los autores, 1998; © Edicions UPC, 1998.

Consideramos los símbolos definidos dentro de un universo SÍMBOLO :

universo SÍMBOLO es
  usa BOOL
  tipo símbolo
  ops op_1, ..., op_n: → símbolo
    x_1, ..., x_k: → símbolo
    var?: símbolo → bool
    _=_, _≠_: símbolo símbolo → bool
  ecns
    var?(op_1) = falso; ...; var?(x_1) = cierto...
    (op_1 = op_1) = cierto; (op_2 = op_2) = falso; ...
    (x ≠ y) = ¬ (x = y)
funiverso

donde var? es una función que indica si un símbolo representa una operación.

a) Definir los términos como instancia de árboles. Si, para simplificar, sólo se consideran términos formados con operaciones de aridad 0, 1 ó 2, estos árboles serán binarios.

b) Sean un término t sin variables y un término r con variables, de modo que en r no haya variables repetidas. Diremos que r se superpone con t mediante α si existe alguna asignación α de las variables de r que iguale r y t. Así, los términos r = suma(x, cero) y t = suma(mult(cero, cero), cero) se superponen mediante la asignación x = mult(cero, cero). En cambio, los términos t = suma(mult(cero, cero), cero) y r = mult(cero, cero) no se superponen. Implementar las operaciones:

superponen?: término término → bool: siendo r un término con variables no repetidas y t sin variables, superponen?(t, r) comprueba si r se superpone con t mediante alguna asignación de sus variables.

as: término término → lista_pares_variables_y_términos: para t y r definidos como antes, as(t, r) devuelve la lista de asignaciones que es necesario hacer a las variables del término r, para que r se superponga con t; si r no se superpone con t, da error.

c) Sean un término t sin variables y dos términos r y s con variables de modo que ni en r ni en s haya variables repetidas y las variables de s sean un subconjunto de las de r. Si r se superpone con t mediante una asignación α, podemos definir la transformación de t usando una ecuación r=s como el resultado de aplicar la asignación α sobre s. Así, se puede transformar el término t = suma(mult(cero, cero), cero) usando la ecuación suma(x, cero) = x, siendo α = mult(cero, cero), con lo que se obtiene el término mult(cero, cero). Implementar la operación transforma: término término término → término, que realiza la operación descrita, o da error si el segundo término no se superpone con el primero.

d) Sea un término t sin variables y dos términos r y s con variables de modo que ni en r ni en s haya variables repetidas y las variables de s sean un subconjunto de las de r. Ahora queremos generalizar el apartado c) para que la transformación se pueda realizar sobre cualquier subtérmino t' de t. Así, dado el término t = suma(mult(cero, cero), cero) y la ecuación r = s: mult(cero, x) = cero, podemos transformar el subtérmino t' = mult(cero, cero) de t aplicando r = s , y obtener como resultado el término suma(cero, cero). Se puede

© Los autores, 1998; © Edicions UPC, 1998.

considerar que todo término es subtérmino de sí mismo. Implementar las operaciones:

transformable?: término término término → bool: siendo r y s términos con variables y t sin variables, transformable?(t, r, s) comprueba si algún subtérmino de t puede transformarse mediante la ecuación r = s (es decir, si r se superpone con algún subtérmino de t).

transfsubt: término término término → término: para t, r y s definidos como antes, transfsubt(t, r, s) transforma un subtérmino de t mediante la ecuación r = s; si no se puede transformar, da error.

e) A continuación se quiere implementar una operación que, dados un término t sin variables y una lista L de ecuaciones de la forma r = s, compruebe si puede transformarse algún subtérmino t' de t mediante alguna ecuación de L, según la mecánica descrita en el apartado d). Implementar las operaciones.

se_puede_transformar?: lista_pares_términos término → bool: siendo L una lista de ecuaciones y t un término sin variables, se_puede_transformar?(L, t) comprueba si algún subtérmino de t puede transformarse mediante alguna ecuación r = s de L.

un_paso: lista_pares_términos término → término: para L y t como antes, un_paso(L, t) transforma un subtérmino de t mediante alguna ecuación r = s de L; si no hay ningún subtérmino transformable, da error. Si hay más de una transformación posible, aplica una cualquiera.

Se pueden usar las especificaciones genéricas de pares y de listas (v. fig. 1.31 y 3.13).

f) Implementar la operación forma_normal: lista_pares_términos término → término que, dado un término t sin variables y una lista L de ecuaciones de la forma r = s, transforme t en su forma normal, aplicando sucesivamente ecuaciones de L. Notar que, si se consideran las ecuaciones r = s como reglas de reescritura r → s, el resultado de este apartado será la implementación del proceso de reescritura.

© Los autores, 1998; © Edicions UPC, 1998.