Es más, en la aplicación del corrector ortográfico interactivo los tries resultan especialmente interesantes. A medida que el usuario escribe, el programa iría moviéndose por el árbol, de manera que cada pulsación de una tecla correspondería a bajar un nivel en el árbol. Al acabar una palabra, se comprobaría si existe un valor para la etiqueta $. Todas las operaciones tardarían un tiempo constante y muy reducido. El corrector podría incluir otras utilidades, como por ejemplo completar una palabra a medio escribir si a partir de ese prefijo sólo existe una palabra correcta.

4.2. Relaciones de equivalencia

Las relaciones de equivalencia son un concepto matemático definido sobre un conjunto dado cualquiera. Como tantos otros conceptos matemáticos, está basado en una idea intuitiva, la representación de relaciones del tipo: ciudades en una misma región, alumnos de la misma clase, instrucciones dentro del mismo bloque de código, enteros con el mismo valor de módulo P, etc.

Definición 4.2 Una **relación de equivalencia** sobre un conjunto C es una relación R que cumple las siguientes propiedades[7]:

- Reflexiva. $\forall a \in C;\ a\ R\ a$
- Simétrica. $\forall a, b \in C;\ a\ R\ b \Leftrightarrow b\ R\ a$
- Transitiva. $\forall a, b, c \in C;\ (a\ R\ b) \wedge (b\ R\ c) \Rightarrow (a\ R\ c)$

Es fácil comprobar estas propiedades para los ejemplos anteriores. Por ejemplo, la propiedad reflexiva significa que una ciudad está en la misma región que ella misma, ¡obviamente!; la simétrica diría que si la ciudad a está en la misma región que b, entonces b está en la misma región que a; y la transitiva, que si a está en misma región que b, y esta en la misma que c, entonces a y c están en la misma región. Las tres se cumplen de manera trivial. En la figura 4.7 se muestra un ejemplo particular de relación de equivalencia, definida sobre un conjunto de nueve elementos.

Las relaciones de equivalencia dan lugar al concepto de **clase de equivalencia**. La clase de equivalencia de un elemento a, según una relación R sobre un conjunto C, es el subconjunto de todos los elementos c de C tales que $c\ R\ a$. Por ejemplo, en los casos anteriores tendríamos: la clase de las ciudades de la Región de Murcia, la clase de los alumnos de segundo curso, etc. El conjunto de todas las clases de equivalencia definidas sobre C según la relación R, forma una **partición** de ese conjunto, es decir, son un conjunto de subconjuntos disjuntos y la unión de todos ellos es el conjunto de partida C.

Nuestro interés, como programadores, es estudiar la implementación eficiente de relaciones de equivalencia definidas sobre conjuntos finitos. Además, vamos a considerar especialmente relaciones que no pueden ser calculadas mediante una fórmula (como ocurre, por ejemplo, con la igualdad de enteros módulo P) y que pueden variar en tiempo de ejecución. En otro caso, la implementación sería inmediata. Pero, en primer lugar, vamos a definir un tipo abstracto de datos para el uso de relaciones de equivalencia.

[7] La expresión "$a\ R\ b$" se lee "a está relacionado con b".

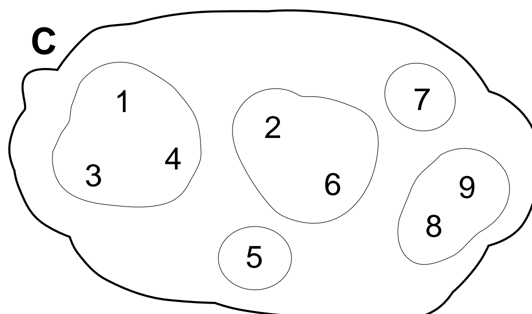

Figura 4.7: Ejemplo de relación de equivalencia y clases de equivalencia, definidos sobre el conjunto de los enteros del 1 a 9. Los elementos que están en la misma clase de equivalencia aparecen dibujados dentro del mismo subconjunto del conjunto C.

El TAD relación de equivalencia

La definición del tipo abstracto debe enumerar las operaciones que se pueden aplicar sobre relaciones de equivalencia y su significado. Existen varias maneras de plantear la utilización del tipo. Una primera forma sería empezar con una relación mínima, vacía, al crearla y tener una operación para unir varias clases de equivalencia en una. Otra posibilidad sería empezar una sola clase de equivalencia al crear e incluir operaciones para partir una clase en dos o más. A lo largo de toda esta sección, trabajaremos exclusivamente con la primera posibilidad.

Además de los constructores del tipo –que, en este caso, serían la operación Crear para formar una nueva relación vacía y Union para juntar dos clases– añadimos una operación Encuentra para consultar la clase de equivalencia de un elemento de C cualquiera. La especificación informal del tipo abstracto es la siguiente.

TAD RelacionEquiv[T: tipo] **es** Crear, Union, Encuentra

Requiere

El tipo T debe tener definida una operación de comparación *Igual (ent T, T; sal boolean)*.

Descripción

Los valores de tipo RelacionEquiv[T] son relaciones de equivalencia modificables, definidas sobre conjuntos de elementos del tipo T. En particular, el conjunto de elementos que pueden participar en la relación de equivalencia es indicado en la operación Crear. Para añadir relaciones entre elementos se debe utilizar la operación Union, y para consulta la clase de un elemento se debe usar Encuentra.

Operaciones

Operación Crear (**ent** *C*: Conjunto[T]; **sal** RelacionEquiv[T])

Requiere: El conjunto *C* debe ser no vacío.

Calcula: Devuelve una relación de equivalencia mínima sobre el conjunto *C*, donde las clases de equivalencia son los elementos en sí mismos, es decir, para todo *a* y *b* de *C*, *a* está relacionado con *b* sí y sólo si *a*=*b*.

Operación Union (**ent** *R*: RelacionEquiv[T]; *a*, *b*: T)

Requiere: Los elementos *a* y *b* deben pertenecer al conjunto *C* sobre el que se ha definido la relación de equivalencia *R*.

Modifica: R
Calcula: En la nueva relación de equivalencia R, los elementos a y b pertenecen a la misma clase de equivalencia. Además, se mantienen las demás relaciones definidas previamente sobre R.

Operación Encuentra (**ent** R: RelacionEquiv[T]; a: T; **sal** T)
Requiere: El elemento a deben pertenecer al conjunto C sobre el que se ha definido la relación de equivalencia R.
Calcula: Devuelve el nombre de la clase de equivalencia a la que pertenece el elemento a dentro de la relación R. Este nombre es un elemento representativo dentro de dicha clase.

Fin RelacionEquiv.

Para saber si dos elementos x e y están en la misma clase, simplemente comprobamos si Encuentra(x) = Encuentra(y). Como se documenta en la especificación de Encuentra, el resultado es un elemento de los que pertenecen a la clase de x. Este elemento puede ser seleccionado arbitrariamente entre los de dicha clase. Solamente necesitamos que al aplicar Encuentra para dos elementos relacionados, devuelva siempre el mismo valor.

Por ejemplo, en la relación de la figura 4.7, a la clase formada por $\{1, 3, 4\}$ se le puede asociar como nombre representativo cualquiera de esos tres valores; pero una vez asociado uno, no puede variar. En definitiva, lo importante no es el valor concreto, sino que Encuentra(1) = Encuentra(3) = Encuentra(4). Obviamente, el nombre sí que puede cambiar después de una operación Union.

Ejemplo de uso de relaciones de equivalencia

Un ejemplo de utilización intensiva de las relaciones de equivalencia lo podemos encontrar en el procesamiento de imágenes digitales. Supongamos una imagen en escala de grises, como la de la figura 4.8, que está compuesta por una matriz de píxeles. En este caso, las clases de equivalencia son las regiones contiguas y del mismo color. Estas clases surgen de la siguiente relación de equivalencia: dos píxeles a y b están relacionados si tienen el mismo color (o nivel de gris) y además son adyacentes en la imagen[8].

En esta aplicación, el cálculo y la modificación de las clases de equivalencia puede resultar interesante para diversos propósitos. Por ejemplo, si queremos aplicar la operación "rellenar una región de color", tendremos que buscar la clase de equivalencia del punto sobre el que se aplica el rellenado. Tras aplicar la operación, pueden ocurrir cambios en las clases de equivalencia, que se deberán tener en cuenta posteriormente.

También podrían resultar interesantes operaciones para contar el número de regiones existentes en la imagen, o saber cuál contiene más píxeles. Por ejemplo, un posible algoritmo de detección de caras humanas en una imagen podría estar basado en buscar una región contigua de color de piel, y que contenga por lo menos dos regiones oscuras correspondientes a los ojos.

En todas estas aplicaciones, la relación está definida sobre un conjunto finito, y puede variar dinámicamente a lo largo del tiempo. Si consideremos imágenes con una resolución de 800×600 píxeles, entonces el conjunto C sobre el que está definida la relación tendrá 480.000 elementos. Por lo tanto, conseguir una implementación muy eficiente para

[8]Es decir, uno se encuentra inmediatamente encima, debajo, a la izquierda o a la derecha del otro.

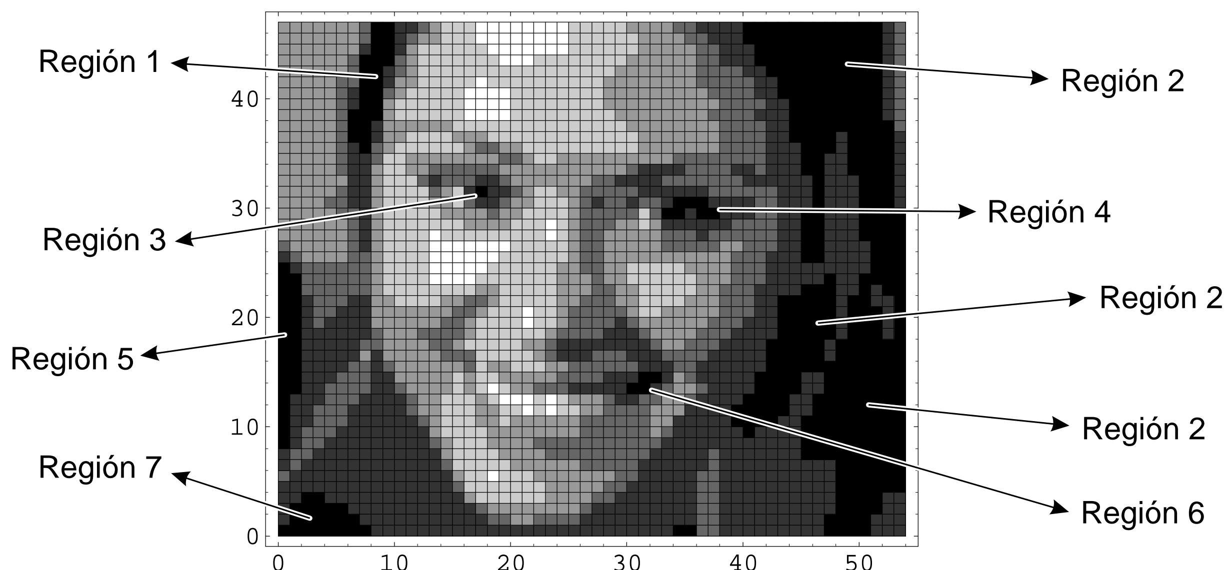

Figura 4.8: Ejemplo de aplicación de relaciones y clases de equivalencia, en procesamiento de imágenes. Aparecen señaladas algunas clases (regiones) cuyos píxeles tienen color negro.

conocer y manejar las clases de equivalencia resulta fundamental en este caso.

4.2.1. Representaciones sencillas de relaciones de equivalencia

La representación de relaciones de equivalencia es un ejemplo típico de problema donde se intuye fácilmente una solución simple y directa. Pero un pequeño e irritante contratiempo hace que esta solución se comporte peor de lo que se esperaba, lo cual obliga a tomar un largo rodeo para alcanzar una representación eficiente.

Representación mediante arrays

Supongamos, para simplificar el problema, que las relaciones de equivalencia están definidas sobre un tipo enumerado. Es más, supongamos que los elementos del conjunto C sobre el que se definen están numerados desde 1 hasta cierto límite N. Por ejemplo, en el caso de las imágenes, podemos asignar un número consecutivo a cada píxel, de arriba abajo y de izquierda a derecha. Para imágenes de resolución 800×600, $N = 480.000$.

La representación más sencilla podría consistir en un array de tamaño N, donde en cada posición se almacena el nombre de la clase de equivalencia del elemento correspondiente. La definición del tipo sería la siguiente.

tipo
  RelacionEquiv[N] = **array** [1..N] **de** entero

En la figura 4.9a) se muestra un ejemplo de la disposición en memoria de la estructura, para la relación de equivalencia definida en la figura 4.7.

La implementación de las operaciones Crear y Encuentra es inmediata.

operación Crear (**var** *R*: RelacionEquiv[N])

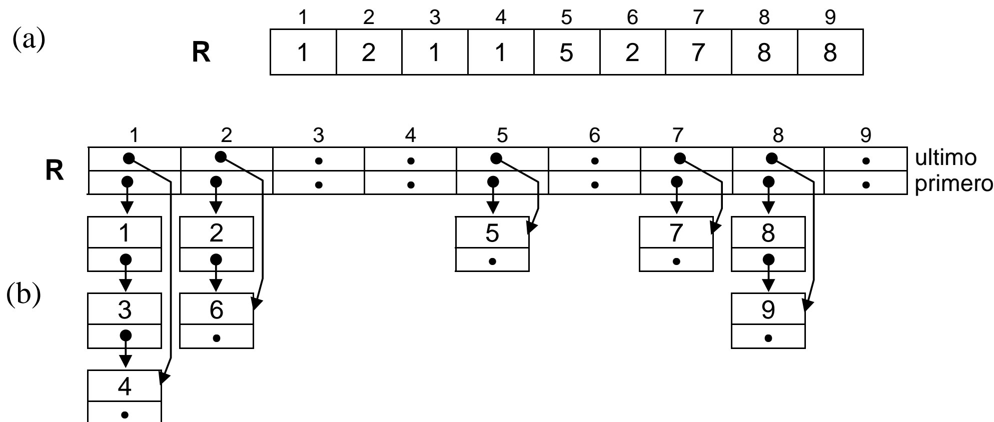

Figura 4.9: Ejemplo de representaciones sencillas de relaciones de equivalencia, para la relación de la figura 4.7. a) Representación con arrays. b) Representación con listas.

para *i*:= 1, ..., N **hacer**
    R[*i*]:= *i*
finpara
operación Encuentra (*R*: RelacionEquiv[N]; *a*: entero): entero
    devolver *R*[*a*]

La operación Encuentra está claramente en un $O(1)$, lo cual resulta muy interesante. Por otro lado, Crear tarda un $O(N)$, lo cual no es muy preocupante puesto que sólo se aplicará una vez. El problema es, sin embargo, la operación Union que es previsible que aparezca con más frecuencia. ¿Cómo hacer la unión de dos clases? Simplemente cambiando en el array el nombre de una de las clases por la otra.

Supongamos que los parámetros que recibe la operación son nombres de clases (y no valores de elementos particulares). La implementación podría ser la siguiente.

operación Union (**var** *R*: RelacionEquiv[N]; *a*, *b*: entero)
    para *i*:= 1, ..., N **hacer**
        si *R*[*i*] = *b* **entonces**
            R[*i*]:= *a*
        finsi
    finpara

Es fácil ver que la operación Union tarda un $O(N)$, en todos los casos. Este coste puede resultar excesivo para muchas aplicaciones. En el ejemplo del procesamiento de imágenes, se aplicará la unión siempre que un píxel tenga a un lado algún otro píxel del mismo color –lo cual suele ser muy frecuente–. Si tenemos que aplicar Union para cada píxel de una imagen de 800×600, el número total de comparaciones dentro del bucle **para** sería: $(800 * 600)^2 =$ ¡230 mil millones de comparaciones!

Representación mediante listas

Para solucionar el problema anterior podemos intentar una representación dinámica mediante listas, al estilo de las representaciones básicas de conjuntos estudiadas en el

capítulo anterior. Para cada clase de equivalencia usamos una lista de elementos, en la que estarán los elementos que pertenecen a la misma. La definición del tipo sería.

```
tipo
    RelacionEquiv[N] = array [1..N] de Lista[entero]
```

Siendo suficientemente cuidadosos, deberíamos usar una estructura que permita concatenar dos listas de forma rápida. Esto se puede conseguir, por ejemplo, usando listas circulares y doblemente enlazadas; o simplemente almacenando en la cabecera de las listas un puntero al primer y al último elemento, como se muestra en la figura 4.9b).

Usando la segunda opción, la operación Union se puede conseguir en un tiempo constante, $O(1)$, de la siguiente forma. Se supone que el tipo Lista incluye los atributos primero y ultimo, y los nodos de lista tienen siguiente, para enlazar los elementos.

```
operación Union (var R: RelacionEquiv[N]; a, b: entero)
    R[a].ultimo.siguiente:= R[b].primero
    R[a].ultimo:= R[b].ultimo
    R[b]:= NULO
```

El problema surge ahora con la operación Encuentra. ¿Cómo podemos saber la clase a la que pertenece un elemento concreto? Deberíamos recorrer todas las listas hasta encontrarlo en alguna de ellas.

```
operación Encuentra (R: RelacionEquiv[N]; a: entero): entero
    para i:= 1, ..., N hacer
        si BuscaLista (R[i], a) entonces
            devolver i
        finsi
    finpara
    error("El elemento", a, "no está en ninguna clase")
```

Damos por supuesta la operación BuscaLista, que realiza un simple recorrido secuencial en una lista. No siempre tardará el mismo tiempo, pero sabemos que en el peor caso se recorrerán en total todos los elementos de todas las listas, es decir N. En promedio, la operación Encuentra recorrerá la mitad de las listas, tardando un $O(N)$.

Si resulta malo que la operación Union esté en un $O(N)$, como ocurría usando arrays, el hecho de que Encuentra tarde un $O(N)$ puede ser aun más desastroso en aplicaciones que requieren un uso intensivo de las relaciones de equivalencia.

4.2.2. Representación mediante punteros al padre

El inconveniente último de las dos anteriores representaciones es que utilizan estructuras lineales –ya sean enlazadas (listas) o contiguas en memoria (arrays)– lo cual ocasiona tiempos lineales, bien para la búsqueda o para la unión de clases de equivalencia. En su lugar vamos a estudiar el uso de estructuras arbóreas. Vamos a disponer de un árbol para cada clase, donde la raíz del árbol será por definición el nombre de esa clase.

Básicamente, la unión de dos árboles se puede conseguir en un tiempo constante, colocando un árbol como subárbol del otro. En cuanto a la búsqueda, está claro que necesitamos una operación que dado un elemento del árbol encuentre cuál es la raíz. Por esta razón, usamos una representación de árboles con punteros al nodo padre.

Árboles de punteros al padre

La clave de la representación de árboles con punteros al padre es que para cada nodo del árbol sólo necesitamos conocer un valor: un puntero o referencia a su padre dentro del árbol. Para la raíz del árbol ese valor será nulo o algún otro valor especial.

Suponiendo que el número de elementos del árbol es fijo, o limitado por un tamaño reducido, la estructura de punteros al padre se puede almacenar en simple array.

tipo
  ArbolEnteros[N] = **array** [1..N] **de** entero

Si R es un árbol de enteros que usa esta estructura, entonces la posición $R[i]$ indica el padre del nodo i. Un ejemplo de representación de árboles con esta definición se puede ver en la figura 4.10. En este caso, se usa el valor 0 para indicar que el nodo es la raíz de un árbol.

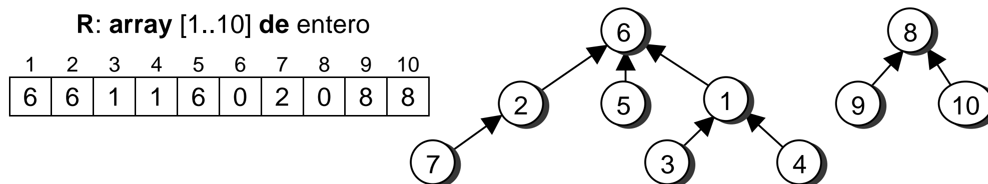

Figura 4.10: Ejemplo de representación de árboles mediante punteros al padre.

Bosques de relaciones de equivalencia

La anterior representación de árboles con punteros al padre se ajusta perfectamente a nuestras necesidades: el número de elementos de la relación de equivalencia es fijo, se define al crear la relación. Realmente, la estructura no almacena un solo árbol sino un conjunto de árboles, lo que normalmente se conoce como un **bosque**. La definición del tipo sería la siguiente[9].

tipo
  RelacionEquiv[N] = **array** [1..N] **de** entero

Como ya hemos avanzado, cada árbol del bosque representa una clase de equivalencia, en la cual la raíz es el nombre de esa clase. La implementación de las operaciones es sencilla. La operación Crear crea una relación de equivalencia mínima, donde cada elemento sólo está relacionado consigo mismo. Equivalentemente, cada elemento del conjunto será la raíz de un árbol. Usando el valor 0 para representar las raíces, tenemos.

operación Crear (**var** *R*: RelacionEquiv[N])
  para *i*:= 1, ..., N **hacer**
    R[*i*]:= 0

[9]El lector podrá apreciar un curioso hecho, ¡la definición es exactamente la misma que en la representación mediante arrays! La diferencia es la forma en la que se maneja el array. Mientras que antes lo usábamos para almacenar las clases de cada elemento, ahora lo usamos para representar árboles. Esto demuestra, una vez más, la estrecha relación existente entre algoritmos y estructuras de datos. La estructura de datos no determina unívocamente un tipo de datos, sino que se deben tener en cuenta también los algoritmos que manipulan esa estructura.

finpara

Aunque su tiempo es claramente un $O(N)$, ya hemos visto que en este caso no supone un inconveniente, ya que sólo se hará al principio. Por otro lado, la operación Union simplemente coloca un árbol como subárbol del otro. Igual que antes, suponemos que los parámetros no son elementos sino nombres de clases de equivalencia[10].

operación Union (**var** *R*: RelacionEquiv[N]; *a*, *b*: entero)
R[*a*]:= *b* // *a se coloca como hijo de b*

Así pues, la operación tarda un interesante $O(1)$. Finalmente, la operación Encuentra debe subir por el árbol, hacia sus antecesores, hasta llegar a la raíz de su árbol. Una posible implementación recursiva sería la siguiente.

operación Encuentra (*R*: RelacionEquiv[N]; *a*: entero): entero
si *R*[*a*] = 0 **entonces**
devolver *a*
sino
devolver Encuentra(*R*[*a*])
finsi

El tiempo de la operación Encuentra depende de la profundidad del nodo buscado dentro del árbol. En principio, es de esperar que un árbol de N nodos tenga una profundidad de un orden logarítmico en función de N, con lo cual el tiempo estaría en un $O(\log n)$. Sin embargo, en el peor caso el árbol puede adoptar una forma lineal: cualquier nodo sólo tiene un descendiente como máximo. En tal caso, un árbol de N nodos tendrá profundidad N y las operaciones de búsqueda estarán en promedio en $O(N)$. Vamos a ver un ejemplo ilustrativo de una situación de este tipo.

Ejemplo 4.1 Utilizando la representación de relaciones de equivalencia con punteros al padre, definimos una variable R de tipo RelacionEquiv[9], para almacenar relaciones en el conjunto de los naturales de 1 a 9. Sobre esta estructura aplicamos las siguientes operaciones: Crear(*R*), Union(*R*, 3, 4), Union(*R*, 9, 8), Union(*R*, 4, 1), Union(*R*, 5, 6), Union(*R*, 1, 6) y Encuentra(*R*, 3). El resultado se muestra en la figura 4.11.

4.2.3. Equilibrado y compresión de caminos

Podemos extraer una conclusión práctica del anterior ejemplo: si queremos conseguir eficiencia en una representación de árboles debemos garantizar que el árbol crecerá a lo ancho, más que a lo largo. Un árbol poco profundo significa pocos pasos en la búsqueda, lo cual se traduce en operaciones más rápidas. Entonces, ¿cómo podemos hacer que el árbol crezca a lo ancho?

Supongamos, en primer lugar, que tenemos un árbol a de profundidad 40 y otro árbol b de profundidad 2. Tenemos dos opciones –en principio igual de válidas–, colocar a como hijo de b o viceversa. Si hacemos que a sea hijo de b, entonces el nuevo árbol tendrá profundidad 41. En cambio, si colocamos b como hijo de a, la profundidad del árbol no varía, seguiría siendo 40. El resultado es bastante claro: si queremos que el árbol no crezca en profundidad, debemos colocar el árbol menos profundo como subárbol

[10]En caso contrario, habría que aplicar las operaciones de búsqueda correspondientes.

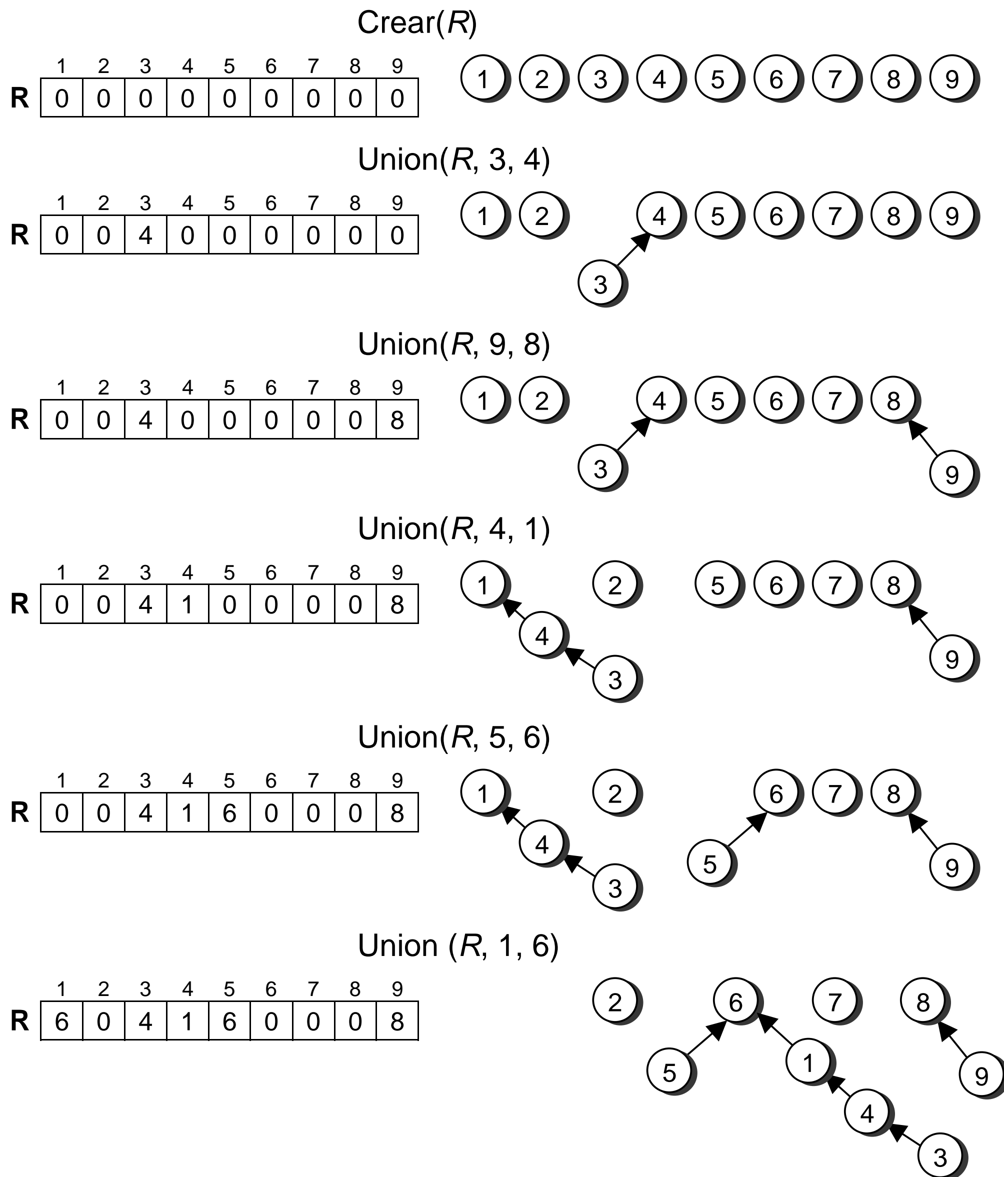

Figura 4.11: Evolución de la estructura de relaciones de equivalencia, para las operaciones del ejemplo 4.1.

del más profundo. Si ambos son igual de profundos, entonces necesariamente el árbol resultante crecerá en profundidad.

Calcular la profundidad de los árboles puede resultar de por sí muy costoso, y especialmente con el tipo de representación usado. Pero si se almacena la profundidad y se va actualizando a medida que crece el árbol, podemos lograrlo eficientemente.

Para no incurrir en un uso adicional de memoria, vamos a usar la misma definición del tipo que en el anterior apartado, pero cambiando la forma de manejar las operaciones.

En particular, si antes almacenábamos un 0 en el array para las raíces, ahora podemos tener 0 o un número negativo indicando la profundidad de ese árbol. De esta manera, en la operación Union podemos comprobar fácilmente cuál es el árbol más profundo. En caso de que sean iguales, da lo mismo cuál se coloque como hijo y además sabemos que el árbol resultante aumentará de nivel.

```
operación Union (var R: RelacionEquiv[N]; a, b: entero)
    si R[b] < R[a] entonces    // El árbol b es más profundo
        R[a]:= b
    sino        // El árbol a es más o igual de profundo
        si R[a] = R[b] entonces
            R[a]:= R[a]-1
        finsi
        R[b]:= a
    finsi
```

Por otro lado, en la operación Encuentra es posible hacer otra modificación interesante conocida como **compresión de caminos**. Supongamos que sobre una relación de equivalencia aplicamos la operación Encuentra(x), obteniendo como resultado una clase y cualquiera. La operación habrá necesitado un número de pasos que depende de su profundidad en el árbol. Ahora bien, puesto que después de ejecutar la operación sabemos que la clase de x es y, es posible hacer que x apunte directamente a y, esto es $R[x] := y$. De esta forma, la próxima vez que se ejecute Encuentra(x) sólo se necesitará un paso para llegar a la raíz del árbol.

Es más, la idea de apuntar directamente a la raíz del árbol se puede aplicar a todos los antecesores del nodo x, que se recorren al aplicar la operación de búsqueda. En definitiva, la operación Encuentra con compresión del árbol sería la siguiente.

```
operación Encuentra (var R: RelacionEquiv[N]; a: entero): entero
    si R[a] ≤ 0 entonces
        devolver a
    sino
        R[a]:= Encuentra(R, R[a])
        devolver R[a]
    finsi
```

El objetivo del proceso de compresión de caminos es reducir la profundidad de los nodos sobre los que se aplica la búsqueda. En algún caso, esta reducción puede dar lugar a una disminución en la profundidad total del árbol.

Sin embargo, se puede ver que la implementación de la operación Encuentra no actualiza esa profundidad, que es almacenada en las raíces con valores negativos. Esta omisión no es casual, recalcular la profundidad del árbol supondría un coste computacional de $O(N)$ que echaría a perder la eficiencia de tipo sin producir una mejora sustancial posterior. Así pues, podemos decir que la profundidad almacenada en los nodos raíces es realmente un valor de profundidad máxima o profundidad suponiendo que no se aplican búsquedas.

Ejemplo 4.2 Vamos a analizar la ejecución de las operaciones del ejemplo 4.1, con las modificaciones propuestas para Union y Encuentra. Las operaciones aplicadas son :

Crear(*R*), Union(*R*, 3, 4), Union(*R*, 9, 8), Union(*R*, 4, 1), Union(*R*, 5, 6), Union(*R*, 4, 6) y Encuentra(*R*, 3). El resultado se muestra en la figura 4.12.

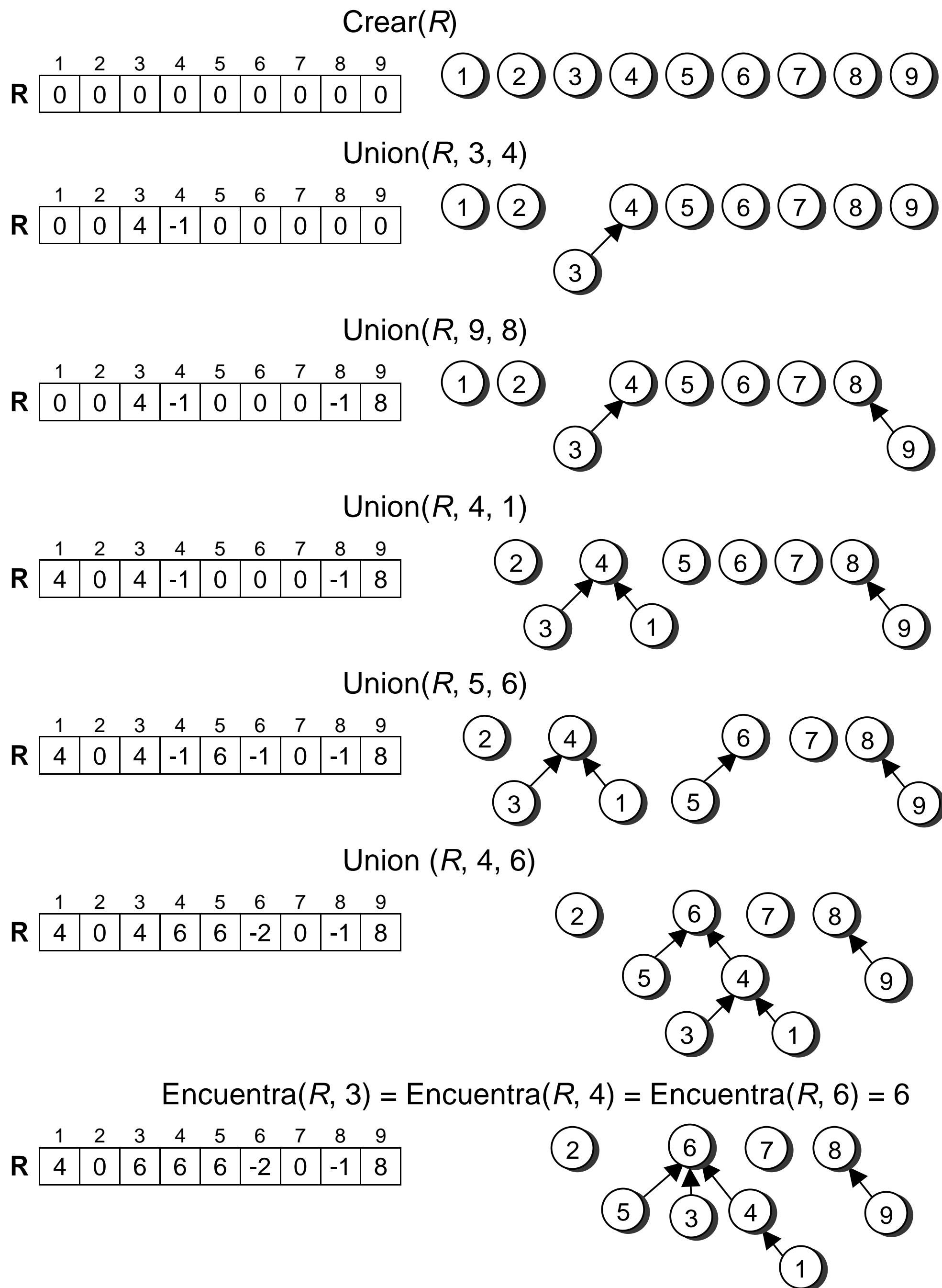

Figura 4.12: Evolución de la estructura de relaciones de equivalencia, para las operaciones del ejemplo 4.1, usando la representación de punteros al padre, con equilibrado y compresión de caminos.

Como cabía esperar, la profundidad de los árboles obtenidos es menor aplicando

equilibrado y compresión de camino. Esto significa un menor tiempo de ejecución para las operaciones del tipo.

Se puede ver que, en el ejemplo, la operación Encuentra(*R*, 1, 6) ha sido sustituida por Encuentra(*R*, 4, 6). Recordemos que la operación Encuentra requiere que se aplique sobre raíces de los árboles. Mientras que el nodo 1 es raíz en el ejemplo 4.1, en el ejemplo 4.2 no lo es. En su lugar, usamos la clase del nodo 1, es decir 4.

Evaluación de la eficiencia

Es evidente que la estructura utilizada para representar relaciones de equivalencia necesita una cantidad de memoria proporcional al tamaño N del conjunto sobre el que se aplica la relación. En el caso de las imágenes, por ejemplo, la memoria podría ser tanta como la utilizada por la imagen en sí. No obstante, este uso es razonable teniendo en cuenta que la relación no puede ser calculada mediante una fórmula.

En lo que se refiere al tiempo de ejecución, tenemos por un lado la operación Union que ejecuta siempre como máximo un número fijo de instrucciones, con lo que su tiempo es un $O(1)$. Por otro lado, el tiempo de la operación Encuentra es proporcional al nivel del nodo buscado en el árbol, como ya hemos visto. El tiempo necesario dependerá del caso.

Mejor caso. Si se aplican las operaciones Union de forma adecuada, el árbol no crece en profundidad más que una cantidad fija y reducida. En esta situación el tiempo de Encuentra sería un $O(1)$.

Peor caso. El peor caso se dará cuando el árbol crezca en profundidad lo más rápidamente posible. Teniendo en cuenta la estructura del procedimiento Union, el árbol sólo crecerá de nivel cuando los dos árboles unidos tengan la misma altura. Se puede demostrar por inducción[11] que un árbol de profundidad h tiene como mínimo 2^h nodos. Por lo tanto, para N nodos la profundidad máxima sería $\log_2 N$, y el tiempo de ejecución de Encuentra en el peor caso es un $O(\log N)$.

Caso promedio. El tiempo promedio de Encuentra, incluyendo equilibrado y compresión de caminos, resulta difícil de calcular. Se puede ver intuitivamente que el tiempo será mucho menor que un tiempo logarítmico. En término medio, será muy difícil que el árbol alcance profundidad 4. Teniendo en cuenta que el árbol de profundidad 5 se forma uniendo dos árboles de profundidad 4, será mucho más difícil obtener el árbol de profundidad 5, y así sucesivamente.

En concreto, el tiempo de ejecución de Encuentra en el caso promedio es un $O(\alpha(N))$; donde $\alpha(N)$ es una función de crecimiento muy lento, situado entre un $O(1)$ y un $O(\log n)$. La función $\alpha(N)$ se define como la pseudo-inversa de una función $A(x)$ de crecimiento muy rápido, conocida como **función de Ackerman**. Por ejemplo, los valores iniciales de esta función son: $A(1) = 2$, $A(2) = 2^2$, $A(3) = 2^{2^2}$, $A(4) = 2^{2^{\cdot^{\cdot^{2}}}}$ donde los puntos significan repetir la operación de elevación 65.536 veces. $A(5)$ es un valor tan grande que no existe una notación estándar para expresarlo de forma expandida. A efectos prácticos, el tiempo promedio de Encuentra se puede considerar *casi* un $O(1)$.

[11] Base de la inducción: un árbol con altura 1 tiene como mínimo 2 nodos. Paso de inducción: suponiendo que los árboles de altura $h-1$ tienen como mínimo 2^{h-1} nodos, un árbol de altura h tendrá como mínimo 2^h nodos. La demostración se deja como ejercicio. Tener en cuenta que un árbol de altura h se forma uniendo dos árboles de altura $h-1$.