Parte V

Estructuras de datos avanzadas

CAPÍTULO

21 Árboles de ensanchamiento

Este capítulo describe una notoria estructura de datos denominada *árbol de ensanchamiento*[1]. Los árboles de ensanchamiento soportan todas las operaciones de los árboles de búsqueda binarios, pero no garantizan un rendimiento $O(\log N)$ en el caso peor. En cambio, sus cotas son *amortizadas*, lo que significa que aunque las operaciones individuales pueden ser costosas, se garantiza que cualquier secuencia de operaciones se comportará como si cada operación en la secuencia exhibiera un comportamiento logarítmico. Como quiera que ésta es una garantía más débil que la proporcionada por los árboles de búsqueda equilibrados, para soportar cada elemento en el árbol sólo necesitaremos el dato y dos referencias por nodo. Además, las operaciones son algo más sencillas de implementar. Como veremos a lo largo de este capítulo, los árboles de ensanchamiento tienen algunas otras propiedades interesantes.

En este capítulo veremos:

- La descripción de los conceptos de amortización y auto-ajustamiento.
- El algoritmo del árbol básico ensanchado de forma ascendente y la demostración del coste amortizado logarítmico por operación.
- Cómo pueden implementarse los árboles de ensanchamiento utilizando un algoritmo descendente, con una implementación completa de los árboles de ensanchamiento (incluyendo el algoritmo de eliminación).
- Comparaciones entre los árboles de ensanchamiento y otras estructuras de datos.

21.1 Auto-ajustamiento y análisis amortizado

Aunque los árboles de búsqueda equilibrados proporcionan un tiempo de ejecución por operación logarítmico en el caso peor, tienen varias limitaciones:

- Los árboles de búsqueda equilibrados necesitan almacenar un atributo extra por nodo con información de equilibrio.

[1] *N. del T.:* En inglés, *splay tree*.

- Son complicados de implementar; como resultado, las inserciones y eliminaciones son costosas y propensas a errores.
- No ganamos nada cuando trabajamos con entradas sencillas.

El problema real es que los atributos extra añaden complicaciones de las que podemos prescindir.

Examinemos las consecuencias de cada una de estas deficiencias. En primer lugar, los árboles de búsqueda equilibrados requieren almacenar un atributo extra. Aunque en teoría éste puede ser tan pequeño como un único bit (como en los árboles rojinegros), en la práctica, para satisfacer las restricciones hardware, el atributo utilizará un entero completo. A pesar de todo, en una época en la que las memorias de los computadores son ya enormes uno puede preguntarse si debe preocuparse demasiado de la memoria. La respuesta en la mayoría de los casos es probablemente no, si no fuera porque el mantenimiento del atributo extra requiere un código más complicado y tiende a llevarnos a tiempos mayores de ejecución y a un número mayor de errores. De hecho, es difícil comprobar si la información de equilibrio de los árboles de búsqueda es correcta, ya que los errores sólo nos conducen a un árbol desequilibrado. Si sólo de forma muy esporádica cometiésemos tales errores *leves*, puede ser muy difícil encontrarlos. Por tanto, como cuestión práctica, los algoritmos que nos permitan eliminar dichas complicaciones, sin con ello sacrificar el rendimiento, se merecen una consideración seria.

La *regla del 90-10* afirma que el 90 por ciento de los accesos corresponden al 10 por ciento de los datos. Los árboles de búsqueda equilibrados no sacan partido de esta regla.

Hay una segunda razón por la cual podemos sospechar que el rendimiento de la búsqueda equilibrada podría mejorarse. Tenemos que su rendimiento en el caso peor, en promedio y en el caso mejor son esencialmente idénticos. Un ejemplo es la operación `buscar` sobre algún elemento X. Es razonable esperar que el coste de una primera ejecución de `buscar` sea logarítmico; pero un segundo acceso inmediato a X debería ser menos costoso que el primero. En un árbol rojinegro esto no es cierto. También esperaríamos que si realizamos accesos a X, Y y Z, entonces un nuevo acceso a la misma terna de elementos debería ser más rápido. Esto es importante debido a la *regla del 90-10*. Esta regla, sugerida por estudios empíricos, afirma que, en la práctica, el 90 por ciento de los accesos que se requieren corresponden sólo al 10 por ciento de los datos. En consecuencia, queremos un acceso rápido para dicho 90 por ciento.

La regla del 90-10 se ha utilizado durante muchos años en sistemas de discos de entrada salida. Una memoria de respaldo (*caché*) almacena en memoria principal el contenido de algunos bloques de disco. Nuestra esperanza es que cuando se haga una petición de acceso a disco, el bloque pueda encontrarse en la cache de memoria principal y así se ahorre el coste de un acceso a disco. Por supuesto, sólo unos pocos bloques de disco pueden almacenarse en memoria. Aún así, almacenar los bloques de disco más recientemente accedidos permite una mejora considerable en el rendimiento, pues frecuentemente los mismos bloques son accedidos una y otra vez. Los navegadores de Internet utilizan la misma idea: una caché almacena localmente las páginas Web visitadas últimamente.

21.1.1 Cotas de tiempo amortizadas

Estamos pidiendo mucho: queremos evitar la información de equilibrio y ser capaces de sacar partido de la regla del 90-10. Ello nos conduce a pensar que podríamos tener que renunciar a alguna de las propiedades de los árboles de búsqueda equilibrados.

Elegimos sacrificar el rendimiento logarítmico en el caso peor. Ya que no queremos mantener información sobre el equilibrio, este sacrificio parece inevitable. Sin embargo, no podemos admitir el rendimiento típico de un árbol de búsqueda desequilibrado. Hay, en cambio, un compromiso razonable: un tiempo $O(N)$ para un acceso aislado puede ser aceptable, siempre y cuando no ocurra muy a menudo. En particular, si M operaciones (empezando con la primera operación) tardan un tiempo total $O(M \log N)$ en el caso peor, entonces el hecho de que algunas operaciones sean costosas no tiene graves consecuencias. Cuando para una secuencia de operaciones podemos demostrar una cota, en el caso peor, mejor que la correspondiente cota obtenida considerando cada operación por separado, el tiempo de ejecución se denomina *amortizado*. En el ejemplo anterior, tenemos un coste amortizado logarítmico. Es decir, algunas operaciones pueden tardar más que un tiempo logarítmico, pero estamos seguros de que se compensan con otras operaciones menos costosas que ocurrieron anteriormente en la secuencia.

El *análisis amortizado* acota el coste de una secuencia de operaciones y distribuye equitativamente este coste entre las operaciones de la secuencia.

Sin embargo, una cota amortizada no es siempre aceptable. En concreto, si una operación mala consume demasiado tiempo, realmente necesitamos una cota en el caso peor en vez de una cota amortizada. Aún así, en muchos casos las estructuras de datos se utilizan como parte de un algoritmo y sólo es importante la cantidad total de tiempo consumida por la estructura en el transcurso de la ejecución del algoritmo.

Ya hemos visto un ejemplo de cota amortizada. Cuando implementamos la duplicación del vector en una pila o cola, el coste de una sola operación es constante cuando no es necesaria la duplicación, pero es $O(N)$ cuando sí lo es. Sin embargo, cualquier secuencia de M operaciones sobre la cola o pila garantiza un coste total $O(M)$, lo que significa un coste amortizado constante por operación. El hecho de que la duplicación del vector sea costosa no tiene consecuencias, porque su coste puede distribuirse entre muchas operaciones anteriores que no fueron costosas.

21.1.2 Una estrategia simple de auto-ajustamiento (que no funciona)

En un árbol binario de búsqueda, no podemos limitarnos a almacenar los elementos frecuentemente accedidos en una tabla. Esto sólo funcionaba en el caso del acceso a memoria externa debido a que la técnica del uso de la memoria de respaldo se beneficia de la gran diferencia entre el tiempo de un acceso a memoria y de un acceso a disco. Ya que el coste de un acceso en un árbol binario de búsqueda es proporcional a la profundidad del nodo accedido, podemos intentar reestructurar el árbol moviendo hacia la raíz los elementos más frecuentemente accedidos. Aunque esto supone un tiempo extra en la primera operación `buscar`, a la larga podría merecer la pena.

La *estrategia de rotación hacia la raíz* reorganiza un árbol de búsqueda binario después de cada acceso, de forma que los elementos frecuentemente accedidos se acercan a de la raíz.

La forma más fácil de desplazar un elemento hacia la raíz consiste en intercambiarlo continuamente con su padre hasta que llegue a la raíz. Entonces, si accedemos al elemento por segunda vez, el segundo acceso es muy barato. Incluso si se realizaran otras operaciones antes del siguiente acceso, este elemento seguirá estando cerca de la raíz y por tanto, será encontrado rápidamente. Esta estrategia se denomina *estrategia de rotación hacia la raíz*. En la Figura 21.1 se muestra la aplicación de esta estrategia al nodo 3 [2].

[2] Una inserción cuenta como un acceso. Por tanto, un elemento siempre se insertaría como una hoja e inmediatamente sería desplazado hacia la raíz. Una búsqueda sin éxito cuenta como un acceso a la hoja donde termina su búsqueda.

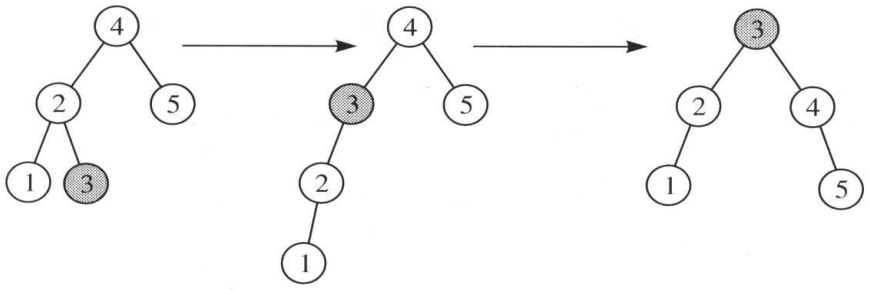


Figura 21.1 Estrategia de rotación hacia la raíz aplicada cuando se accede al nodo 3.

La estrategia de rotación hacia la raíz es buena si se puede aplicar la regla del 90-10. Puede ser muy mala cuando no se puede aplicar.

El resultado de la rotación es que un futuro acceso (durante algún tiempo) al nodo 3 será barato. Desgraciadamente, en el proceso de mover el nodo 3 hacia arriba dos niveles, los nodos 4 y 5 se mueven hacia abajo un nivel. Esto significa que si los patrones de acceso no siguen la regla del 90-10, es posible que se produzca una larga secuencia de accesos malos. Como resultado, la estrategia de rotación hacia la raíz no tendría un comportamiento amortizado logarítmico; lo cual es difícilmente aceptable. Un caso malo se presenta en el Teorema 21.1.

Teorema 21.1

Existen secuencias arbitrariamente largas para las cuales M accesos con rotación hacia la raíz utilizan un tiempo $\Theta(MN)$.

Demostración

Consideremos el árbol formado al insertar las claves 1, 2, 3, ..., N, en un árbol inicialmente vacío. Esto produce un árbol formado exclusivamente por hijos izquierdos. Esto no es de momento malo, ya que el tiempo de construcción del árbol es $O(N)$. *Como ilustra la Figura 21.2, cada nuevo nodo añadido se convierte en un hijo de la raíz, tras lo cual sólo se necesita una rotación para colocar el nuevo elemento en la raíz. La parte negativa, como ilustra la Figura 21.3, es que el acceso al nodo con clave 1 consume N unidades de tiempo. Una vez que las rotaciones han terminado, el acceso al nodo con clave 2 consume N unidades de tiempo, y el acceso al nodo 3 consume* $N - 1$ *unidades de tiempo. Acceder a las N claves en orden requiere un total* $N + \sum_{i=2}^{N} i = \Theta(N^2)$. *Después de acceder a todos los nodos, el árbol ha vuelto a su estado inicial, y podemos repetir la secuencia. Por ello tenemos un coste amortizado* $\Theta(N)$, *lo cual, ciertamente, no es una gran cosa.*

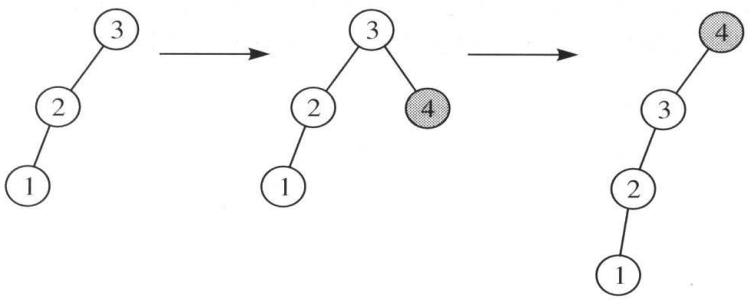


Figura 21.2 Inserción de 4 utilizando la rotación hacia la raíz.

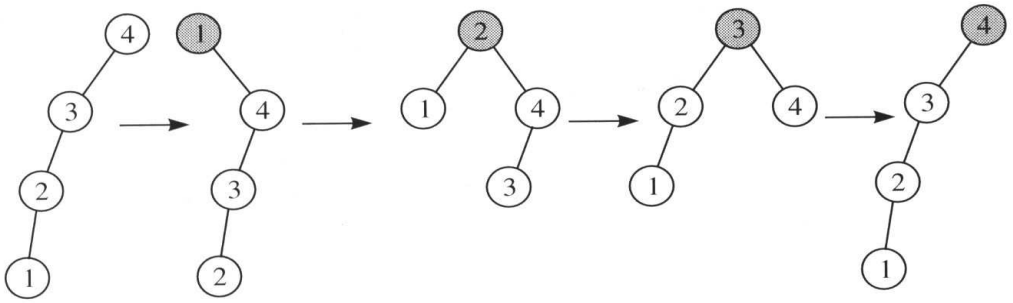


Figura 21.3 Ciertos accesos secuenciales a los elementos requieren un tiempo total cuadrático.

21.2 Árboles básicos de ensanchamiento ascendente

En los *árboles de ensanchamiento ascendente*, los elementos se rotan hacia la raíz de una forma algo más complicada que con la rotación sencilla hacia la raíz.

Lograr un coste amortizado logarítmico parece imposible, porque cuando movemos un elemento hacia la raíz vía rotaciones, otros elementos pasan a estar más profundos. Parece que si no se guarda información sobre el equilibrio siempre podría haber algunos nodos muy profundos. Sorprendentemente, podemos arreglar de forma sencilla la estrategia de rotación hacia la raíz, para conseguir una cota amortizada logarítmica. La estrategia de rotación hacia la raíz se denomina *ensanchamiento*. Su implementación nos lleva a los *árboles de ensanchamiento ascendente*.

La estrategia de ensanchamiento es parecida a la de rotación hacia la raíz, con una sutil diferencia. Ahora seguimos rotando de forma ascendente a lo largo del camino de acceso (más adelante presentaremos una estrategia descendente). Sea X un nodo, distinto de la raíz, en el camino de acceso en el que estamos rotando. Si el padre de X es la raíz del árbol, simplemente rotamos X y la raíz, como muestra la Figura 21.4. Ésta es la última rotación, y coloca X en la raíz. Observe que esto es exactamente lo que se haría en la rotación hacia la raíz. Éste es el caso *zig*.

Los casos *zig* y *zig-zag* son idénticos al caso de rotación hacia la raíz.

En otro caso, X tiene un padre P y un abuelo G, y entonces hay dos casos a considerar, junto con sus simétricos. El primer caso es el denominado *zig-zag*, que corresponde al caso interior en los árboles AVL. Aquí X es un hijo derecho y P es un hijo izquierdo (o viceversa). Realizamos una doble rotación, exactamente análoga a la doble rotación de los árboles AVL, como muestra la Figura 21.5. Observemos que ya que una doble rotación equivale a dos rotaciones ascendentes, este caso no se diferencia de lo que haríamos en la rotación hacia la raíz. En la Figura 21.1 el ensanchamiento en el nodo 3 corresponde a una simple rotación zig-zag.

El caso *zig-zig* es exclusivo de los árboles de ensanchamiento.

El último caso es el caso *zig-zig*, que corresponde al caso externo de los árboles AVL. Aquí, X y P son ambos hijos izquierdos o hijos derechos. En este caso, transformamos el árbol de la izquierda de la Figura 21.6 en el árbol de la derecha.

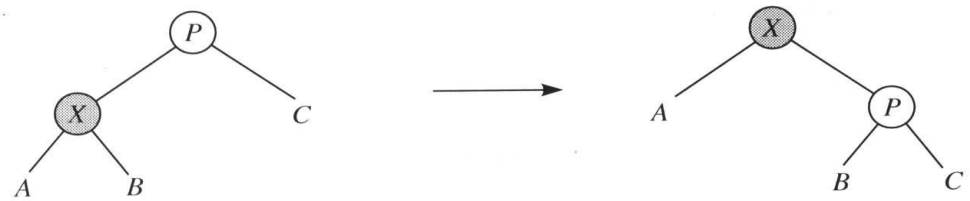


Figura 21.4 Caso zig (rotación simple normal).

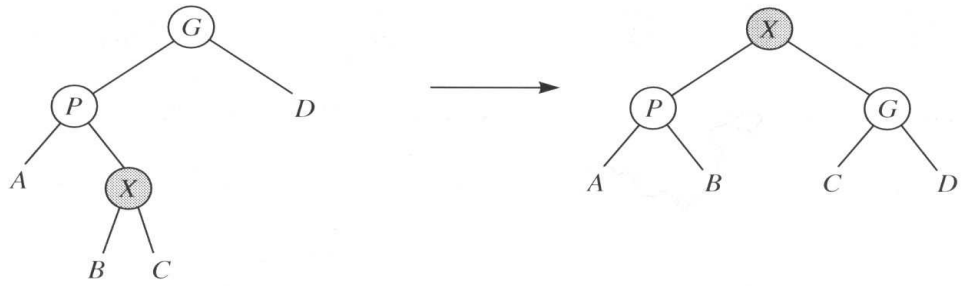


Figura 21.5 Caso zig-zag (igual que una rotación doble); se omite el caso simétrico.

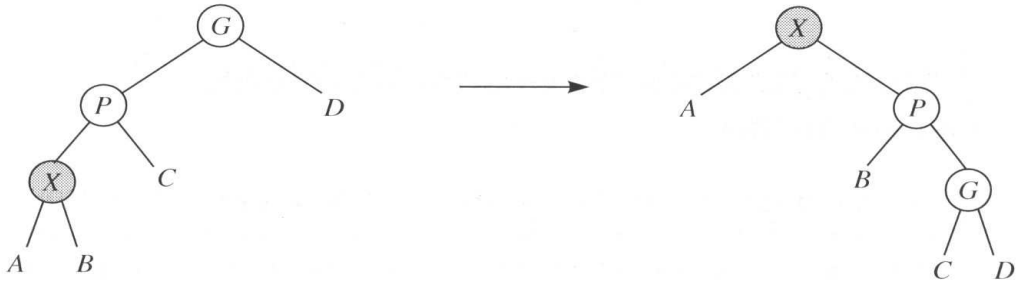


Figura 21.6 Caso zig-zig (exclusivo de los árboles de ensanchamiento); se omite el caso simétrico.

Observemos que esto difiere de lo que se hace en la rotación hacia la raíz. La rotación hacia la raíz rota entre X y P y después entre X y G. La rotación de ensanchamiento zig-zig rota entre P y G y después entre X y P.

El ensanchamiento tiene el efecto de dividir aproximadamente por la mitad la profundidad de la mayoría de los nodos en el camino de acceso, mientras que incrementa como mucho en dos la profundidad de unos pocos nodos.

El cambio en el caso zig-zig parece bastante pequeño; sorprende un tanto que tenga tanta repercusión. Para ver la diferencia entre el ensanchamiento y la rotación hacia la raíz, consideremos la secuencia que provocaba el mal resultado del Teorema 21.1. De nuevo, insertamos las claves 1, 2, 3, ..., N en un árbol inicialmente vacío en un tiempo total lineal y obtenemos un árbol desequilibrado, sólo con hijos izquierdos. Sin embargo, como muestra la Figura 21.7, el resultado de

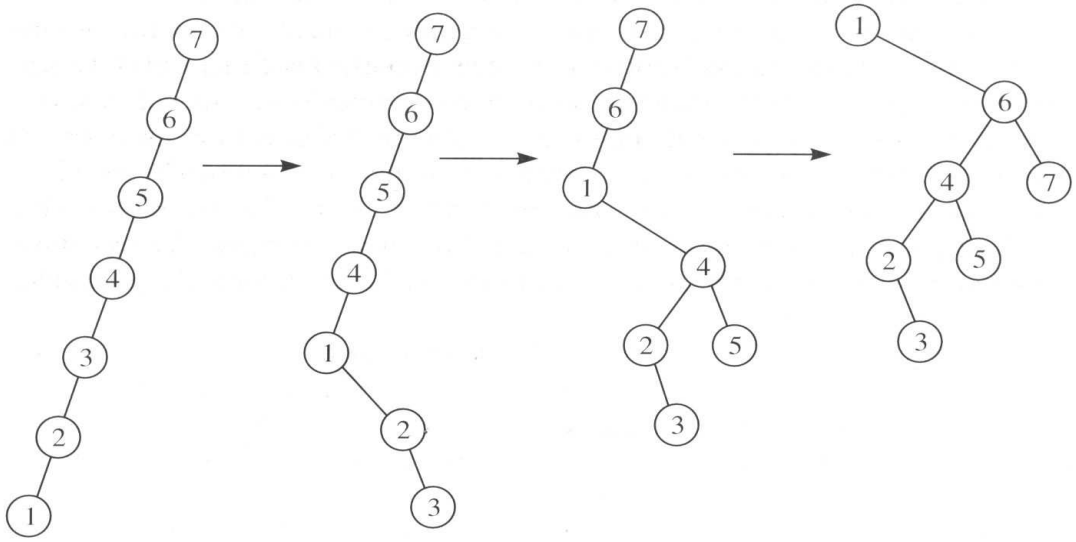


Figura 21.7 Resultado del ensanchamiento en el nodo 1 (tres zig-zig y un zig).

un ensanchamiento es algo mejor. Después del ensanchamiento en el nodo 1, que requiere N accesos a nodos, un ensanchamiento del nodo 2 consistirá aproximadamente en $N/2$ accesos, en vez de en $N - 1$ accesos. El ensanchamiento no sólo mueve el nodo accedido a la raíz, también divide aproximadamente por 2 la profundidad de la mayoría de los nodos en el camino de acceso. En contrapartida, sólo algunos nodos sombreados se profundizan, y como mucho dos niveles. Un ensanchamiento posterior en el nodo 2 llevará a los nodos del camino a $N/4$ de la raíz. Esto se repite, hasta que la profundidad llega a ser, aproximadamente, $\log N$. De hecho, un complicado análisis demuestra que lo que era un caso malo para el algoritmo de rotación hacia la raíz, es ahora un caso bueno para el ensanchamiento: el acceso secuencial de los N elementos en el árbol ensanchado precisa un tiempo total que es sólo $O(N)$. Por tanto, ganamos bastante, al menos con entradas sencillas. La Sección 11.4 demuestra, mediante cuentas sutiles, que ahora ya no hay secuencias de accesos malas.

21.3 Operaciones básicas de los árboles de ensanchamiento

Después de insertar un elemento como una hoja, éste es ensanchado hacia la raíz.

Como se mencionó anteriormente, después de cada acceso se realiza una operación de ensanchamiento. Cuando se realiza una inserción, realizamos un ensanchamiento. Como resultado, el nuevo elemento insertado se convierte en la raíz del árbol. De no hacerlo, podríamos gastar un tiempo cuadrático en la inserción de N elementos en el árbol.

Todas las operaciones de búsqueda incorporan un ensanchamiento.

En la operación `buscar`, hacemos en ensanchamiento desde el último nodo accedido durante la búsqueda. Si la búsqueda tiene éxito, entonces el elemento encontrado será ensanchado y se convertirá en la nueva raíz. Si la búsqueda no tiene éxito, será ensanchado el último nodo accedido antes de alcanzar una referencia `null` y se convertirá en la nueva raíz. Este proceso es necesario; de otra forma, si reiterásemos la búsqueda del elemento 0 en el árbol inicial de la Figura 21.7 utilizaríamos un tiempo lineal por operación. De igual manera, las operaciones `buscarMin` y `buscarMax` realizarán un ensanchamiento después de acceder al árbol.

Las operaciones interesantes son las eliminaciones. Recordemos que `eliminarMin` y `eliminarMax` son operaciones importantes de las colas de prioridad. Con árboles de ensanchamiento, estas operaciones son sencillas. Implementamos `eliminarMin` de la siguiente manera. Primero realizamos `buscarMin`, lo que hace que el menor elemento se coloque en la raíz, con lo que por las propiedades de los árboles de búsqueda, no habrá ningún hijo izquierdo. Podemos utilizar el hijo derecho como nueva raíz. De forma similar, `eliminarMax` puede implementarse llamando a `buscarMax` y haciendo que la nueva raíz sea el hijo izquierdo después del ensanchamiento.

Las operaciones de eliminación son más simples de lo habitual. También incorporan un paso de ensanchamiento (a veces dos).

Hasta la operación `eliminar` es sencilla. Para realizar la eliminación, accedemos al elemento que queremos eliminar. Esto lo coloca en la raíz. Si es borrado, obtenemos dos subárboles I y D (izquierdo y derecho). Si buscamos el mayor elemento en I, utilizando la operación `buscarMax`, entonces el mayor elemento se colocará en la raíz de I y no tendrá hijo derecho. Terminamos la operación `eliminar` haciendo que D sea el hijo derecho de la raíz de I. Un ejemplo de esta operación se muestra en la Figura 21.8.

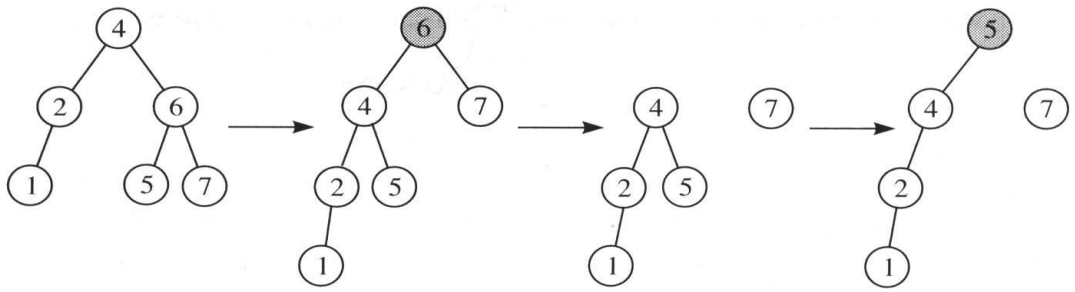


Figura 21.8 La operación `eliminar` aplicada al nodo 6; primero el 6 se ensancha hacia la raíz; la eliminación deja dos subárboles; se realiza un `buscarMax` en el subárbol izquierdo, elevando el 5 a la raíz de un árbol izquierdo; entonces se puede añadir el hijo derecho (no mostrado).

El coste de la operación `eliminar` es el de dos ensanchamientos. Todas las demás operaciones tienen el coste de un ensanchamiento. Por tanto, necesitamos analizar el coste de una serie de pasos de ensanchamiento. La próxima sección muestra que el coste amortizado de un ensanchamiento es, como mucho, el de $3 \log N + 1$ rotaciones. Entre otras cosas, esto significa que no debemos preocuparnos de que el algoritmo de eliminación descrito anteriormente pueda dar lugar a comportamientos sesgados. La cota amortizada de los árboles de ensanchamiento asegura que cualquier secuencia de M ensanchamientos necesitará, como mucho, $3\,M \log N + M$ rotaciones. En consecuencia, una secuencia de M operaciones que empiece con un árbol vacío exigirá, como mucho, un tiempo $O(M \log N)$.

21.4 Análisis del ensanchamiento ascendente

El análisis de los árboles de ensanchamiento es muy complicado y forma parte de una teoría más amplia de análisis amortizados.

El análisis de los árboles de ensanchamiento es complicado porque cada ensanchamiento puede necesitar desde unas pocas hasta $O(N)$ rotaciones. Es más, al contrario de lo que sucede con los árboles de búsqueda equilibrados, cada ensanchamiento cambia la estructura del árbol. Esta sección prueba que el coste amortizado de un ensanchamiento es, como mucho, el de $3 \log N + 1$ rotaciones. La cota amortizada de los árboles de ensanchamiento asegura que cualquier secuencia de M ensanchamientos necesitará, como mucho, $3\,M \log N + M$ rotaciones. En consecuencia, una secuencia de M operaciones que empiece con un árbol vacío exigirá, como mucho, un tiempo $O(M \log N)$.

La *función de potencial* es un mecanismo de contabilidad utilizado para establecer la cota de tiempo requerida.

Para probar esta cota, introducimos una función de contabilidad denominada *función de potencial*. La función de potencial no se calcula en absoluto por el algoritmo. Es sólo un mecanismo de contabilidad virtual para establecer la cota de tiempo requerida. Su elección no es obvia, obteniéndose generalmente tras una gran cantidad de pruebas y errores.

Para cada nodo i en el árbol de ensanchamiento, sea $S(i)$ el número de descendientes de i (incluyendo al propio i). Nuestra función de potencial será la suma, sobre todos los nodos i en el árbol T, del logaritmo de $S(i)$. Más exactamente,

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

Para simplificar la notación, tomaremos $R(i) = \log S(i)$. Esto nos conduce a

$$\Phi(T) = \sum_{i \in T} R(i).$$

El *rango* de un nodo es el logaritmo de su tamaño. Los rangos y los tamaños no se computan; son sólo herramientas de contabilidad para la demostración. Sólo el rango de los nodos cambia en el camino de ensanchamiento.

$R(i)$ representa el *rango* del nodo i. Recordemos que ni los rangos ni los tamaños son computados por los algoritmos de los árboles de ensanchamiento (a menos, por supuesto, que se necesiten estadísticas). Observemos que el rango de la raíz es $\log N$. Además, cuando se realiza una rotación zig, sólo se modifican los rangos de los dos nodos involucrados en la rotación. Cuando se realiza una rotación zig-zig o zig-zag, sólo cambia el rango de tres nodos. Y finalmente, cada ensanchamiento simple está formado por una serie de rotaciones zig-zig o zig-zag, seguidas, eventualmente, por una rotación zig. Cada rotación zig-zig o zig-zag puede contarse como dos rotaciones simples.

Sea X el nodo desde donde se ensancha el árbol y r el número total de rotaciones realizadas durante el ensanchamiento. Sea Φ_i la función de potencial del árbol inmediatamente después del i-ésimo ensanchamiento. En particular, Φ_0 es el potencial antes del primer ensanchamiento.

Teorema 21.2

Si el i-ésimo ensanchamiento utiliza r_i rotaciones, se tiene

$$\Phi_i - \Phi_{i-1} + r_i \leq 3 \log N + 1.$$

Todas las demostraciones de esta sección utilizan el concepto de sumas solapadas.

Antes de probar el Teorema 21.2, veamos lo que significa. El coste de M ensanchamientos puede tomarse como $\Sigma_{i=1}^{M} r_i$ rotaciones. Si los M ensanchamientos son consecutivos (es decir, no se entremezclan inserciones o eliminaciones), entonces el potencial del árbol después del i-ésimo ensanchamiento es el mismo que antes del $(i + 1)$-ésimo ensanchamiento. Por tanto, podemos utilizar M veces el Teorema 21.2 para obtener la secuencia de ecuaciones en la Ecuación 21.1

$$\begin{aligned}
\Phi_1 - \Phi_0 + r_1 &\leq 3 \log N + 1 \\
\Phi_2 - \Phi_1 + r_2 &\leq 3 \log N + 1 \\
\Phi_3 - \Phi_2 + r_3 &\leq 3 \log N + 1 \\
&\cdots \\
\Phi_M - \Phi_{M-1} + r_M &\leq 3 \log N + 1
\end{aligned} \tag{21.1}$$

Estas ecuaciones se solapan, de modo que si sumamos todas ellas, obtenemos

$$\Phi_M - \Phi_0 + \Sigma_{i=1}^{M} r_i \leq (3 \log N + 1)M, \tag{21.2}$$

lo que acota el número total de rotaciones de la siguiente manera

$$\Sigma_{i=1}^{M} r_i \leq (3 \log N + 1)M - (\Phi_M - \Phi_0).$$

Consideremos ahora lo que ocurre cuando se entremezclan las inserciones con las búsquedas. Merece la pena observar que el potencial de un árbol vacío es 0. Cuando se inserta un nodo en el árbol como una hoja, antes del ensanchamiento el potencial del árbol se ve incrementado en como mucho $\log N$ (esto se demostrará

en breve). Supongamos que se utilizan r_i rotaciones en la inserción y que el potencial antes de la inserción es Φ_{i-1}. Después de la inserción el potencial valdrá como mucho $\Phi_{i-1} + \log N$. Después del ensanchamiento que mueve el nodo insertado a la raíz, el nuevo potencial satisfará

$$\begin{aligned} \Phi_i - (\Phi_{i-1} + \log N) + r_i &\leqslant 3 \log N + 1 \\ \Phi_i - \Phi_{i-1} + r_i &\leqslant 4 \log N + 1. \end{aligned} \tag{21.3}$$

Supongamos que se realizan F búsquedas e I inserciones, y que Φ_i representa el potencial después de la i-ésima operación. Entonces, ya que cada búsqueda está gobernada por el Teorema 21.2 y cada inserción está gobernada por la Ecuación 21.3, el proceso de suma por solapamientos nos conduce a

$$\Sigma_{i=1}^{M} r_i \leqslant (3 \log N + 1)F + (4 \log N + 1)I - (\Phi_M - \Phi_0). \tag{21.4}$$

Por otra parte, antes de la primera operación el potencial es 0 y, ya que nunca puede ser negativo, tenemos $\Phi_M - \Phi_0 \geqslant 0$. En consecuencia, obtenemos

$$\Sigma_{i=1}^{M} r_i \leqslant (3 \log N + 1)F + (4 \log N + 1)I, \tag{21.5}$$

lo que muestra que el coste por operación de cualquier secuencia de búsquedas e inserciones es, como mucho, logarítmico. Ya que una eliminación equivale a dos ensanchamientos, ella también es logarítmica. Con todo ello podemos probar las dos afirmaciones pendientes, a saber, el Teorema 21.2 y el hecho de que una inserción de un nodo añade como mucho log N al potencial. Ambos teoremas se prueban utilizando argumentos de despliegue. Nos ocupamos primero en el Teorema 21.3 de la afirmación sobre la inserción.

Teorema 21.3 *La inserción como una hoja del N-ésimo nodo en un árbol agrega, como mucho,* log N *al potencial del árbol.*

Demostración *Los únicos nodos cuyos rangos se ven afectados son aquéllos en el camino desde la hoja insertada a la raíz. Sean* $S_1, S_2, \ldots, S_k$ *sus tamaños antes de la inserción y observemos que* $S_k = N - 1$ *y* $S_1 < S_2 < \cdots < S_k$. *Tomamos como* $S'_1, S'_2, \ldots, S'_k$ *los tamaños después de la inserción. Claramente,* $S'_i \leqslant S_{i+1}$ *para* $i < k$, *ya que* $S'_i = S_i + 1$. *En consecuencia,* $R'_i \leqslant R_{i+1}$. *Por tanto, el cambio en el potencial es*

$$\Sigma_{i=1}^{k} (R'_i - R_i) \leqslant R'_k - R_k + \Sigma_{i=1}^{k-1} (R_{i+1} - R_i) \leqslant \log N - R_1 \leqslant \log N.$$

Para probar el Teorema 21.2, descomponemos cada paso de ensanchamiento en sus componentes zig, zig-zag y zig-zig, y establecemos una cota del coste de cada tipo de rotación. Por despliegue de estas cotas, obtenemos una cota para el ensanchamiento. Antes de continuar, necesitamos un resultado técnico, el Teorema 21.4.

Teorema 21.4

Si $a + b \leq c$ y a y b son ambos enteros positivos, entonces

$$\log a + \log b \leq 2 \log c - 2.$$

Demostración

En virtud de la desigualdad aritmético-geométrica, tendremos

$$\sqrt{ab} \leq (a + b)/2.$$

Por tanto $\sqrt{ab} \leq c/2$. Elevando al cuadrado ambos lados obtenemos $ab \leq c^2/4$ y tomando logaritmos a ambos lados queda demostrado el teorema.

Ahora estamos listos para probar el Teorema 21.2.

21.4.1 Demostración de la cota de ensanchamiento

En primer lugar, si el nodo a ensanchar es la raíz, no se producirán rotaciones y no hay ningún cambio de potencial. De modo que el resultado del teorema se tendría trivialmente, por lo que podemos asumir que hay al menos una rotación. Sea X el nodo involucrado en el ensanchamiento. Necesitamos probar que si se realizan r rotaciones (un ziz-zig o zig-zag cuenta como dos rotaciones), entonces r más el cambio de potencial es, como mucho, $3 \log N + 1$. Sea Δ el cambio de potencial causado por cualquiera de los pasos del ensanchamiento, zig, zig-zag, o zig-zig. Sean $R_i(X)$ y $S_i(X)$ el rango y el tamaño de cada nodo X inmediatamente antes de un paso del ensanchamiento y sean $R_f(X)$ y $S_f(X)$ el rango y el tamaño de cada nodo X inmediatamente después del paso del ensanchamiento. A continuación se presentan las cotas que serán probadas.
En el caso de un paso zig que asciende al nodo X, $\Delta \leq 3(R_f(X) - R_i(X))$, mientras que para los otros dos pasos, $\Delta \leq 3(R_f(X) - R_i(X)) - 2$. Cuando sumamos estas cotas sobre todos los pasos que comprenden un ensanchamiento, la suma da lugar a la cota deseada. Estas cotas se prueban de forma separada en los Teoremas del 21.5 al 21.7. Entonces, la demostración del Teorema 21.2 se completa aplicando un razonamiento de sumas solapadas.

Teorema 21.5

Para un paso zig, $\Delta \leq 3(R_f(X) - R_i(X))$.

Demostración

Como se ha mencionado antes en esta sección, los únicos nodos cuyos rangos cambian en un paso zig son X y P. En consecuencia, el cambio de potencial es $R_f(X) - R_i(X) + R_f(P) - R_i(P)$. De la Figura 21.4 deducimos $S_f(P) < S_i(P)$; por lo que se sigue que $R_f(P) - R_i(P) < 0$. En consecuencia, el cambio de potencial satisface $\Delta \leq R_f(X) - R_i(X)$. Ya que $S_f(X) > S_i(X)$, se sigue que $R_f(X) - R_i(X) > 0$, por lo que $\Delta \leq 3(R_f(X) - R_i(X))$.

Los pasos zig-zag y zig-zig son más complicados, porque se ven afectados los rangos de tres nodos. Probamos primero el caso zig-zag.

Teorema 21.6 *Para un paso zig-zag,* $\Delta \leqslant 3\,(R_f(X) - R_i(X)) - 2$.

Demostración *Como antes, tenemos tres cambios, por lo que el cambio de potencial viene dado por*

$$\Delta = R_f(X) - R_i(X) + R_f(P) - R_i(P) + R_f(G) - R_i(G).$$

A partir de la Figura 21.5 deducimos $S_f(X) = S_i(G)$, *por lo que los rangos de ambos nodos deben ser iguales. Entonces obtenemos*

$$\Delta = -R_i(X) + R_f(P) - R_i(P) + R_f(G).$$

Además $S_i(P) \geqslant S_i(X)$, *por lo que,* $R_i(P) \geqslant R_i(X)$. *Realizando esta sustitución y reorganizando términos obtenemos*

$$\Delta \leqslant R_f(P) + R_f(G) - 2R_i(X). \tag{21.6}$$

A partir de la Figura 21.5, $S_f(P) + S_f(G) \leqslant S_f(X)$. *Aplicando el teorema 21.4, obtenemos* $\log S_f(P) + \log S_f(G) \leqslant 2 \log S_f(X) - 2$, *y por la definición del rango, esto se convierte en*

$$R_f(P) + R_f(G) \leqslant 2R_f(X) - 2. \tag{21.7}$$

Sustituyendo la Ecuación 21.7 en la Ecuación 21.6 obtenemos

$$\Delta \leqslant 2R_f(X) - 2R_i(X) - 2. \tag{21.8}$$

Como en el caso de la rotación zig, $R_f(X) - R_i(X) > 0$, *por lo que podemos sumarlo al lado derecho de la Ecuación 21.8, y factorizar, para obtener la desigualdad deseada*

$$\Delta \leqslant 3\,(R_f(X) - R_i(X)) - 2.$$

Finalmente, probamos la cota para el caso zig-zig.

Teorema 21.7 *Para cada paso zig-zig se tiene* $\Delta \leqslant 3(R_f(X) - R_i(X)) - 2$.

Demostración *Como antes, tenemos tres cambios, por lo que el cambio de potencial viene dado por*

$$\Delta = R_f(X) - R_i(X) + R_f(P) - R_i(P) + R_f(G) - R_i(G).$$

De la Figura 21.6 deducimos $S_f(X) = S_i(G)$, *por lo que los rangos de ambos nodos son iguales. Obtenemos entonces*

$$\Delta = -R_i(X) + R_f(P) - R_i(P) + R_f(G).$$

De forma análoga obtenemos $R_i(P) > R_i(X)$ *y* $R_f(P) < R_f(X)$. *Realizando esta sustitución y reorganizando términos obtenemos*

$$\Delta < R_f(X) + R_f(G) - 2R_i(X). \tag{21.9}$$

De la Figura 21.6, $S_i(X) + S_f(G) \leqslant S_f(X)$, por lo que al aplicar el Teorema 21.4 nos queda

$$R_i(X) + R_f(G) \leqslant 2R_f(X) - 2. \qquad \textbf{(21.10)}$$

Reorganizando la Ecuación 21.10, obtenemos

$$R_f(G) \leqslant 2R_f(X) - R_i(X) - 2. \qquad \textbf{(21.11)}$$

Cuando sustituimos en la Ecuación 21.11 obtenemos

$$\Delta \leqslant 3\,(R_f(X) - R_i(X)) - 2.$$

Una vez que hemos obtenido cotas para cada uno de los pasos del ensanchamiento, podemos finalmente completar la demostración del Teorema 21.2.

Demostración (del Teorema 21.2)

Sea $R_0(X)$ el rango de X antes del ensanchamiento, y $R_i(X)$ el rango de X después del i-ésimo paso del ensanchamiento. Antes del último paso del ensanchamiento, todos los pasos han sido zig-zag o zig-zig. Supongamos que hay k de estos pasos. Entonces el número total de rotaciones realizadas en este punto es 2k. El cambio total del potencial es

$$\Sigma_{i=1}^{k}\,(3(R_i(X) - R_{i-1}(X)) - 2).$$

Esta suma nos conduce a $3(R_k(X) - R_0(X)) - 2k$. En este punto, el número total de rotaciones más el cambio total de potencial está acotado por $3R_k(X)$, ya que 2k términos se cancelan y el rango inicial de X es no negativo. Si la última rotación es un zig-zig o un zig-zag, entonces un nuevo proceso de sumas solapadas nos conduce a un total de 3R(raíz). Observemos que aquí, el -2 en el incremento de potencial cancela dos rotaciones. Por su parte, esto no ocurre en el caso zig, por lo que obtendríamos 3R(raíz) + 1. Ya que, en el peor caso, el rango de la raíz es $\log N$*, el número total de rotaciones más el cambio de potencial durante el ensanchamiento es, como mucho,* $3 \log N + 1$.

La demostración de la cota de los árboles de ensanchamiento, aunque compleja, ilustra algunos puntos interesantes. Primero, el caso zig-zig es aparentemente el más costoso, ya que contribuye con la constante dominante tres, mientras que el caso zig-zag lo hace con dos. La demostración no funcionaría si intentáramos aplicarla al caso de la rotación hacia la raíz. Esto se debe a que, en el caso zig, el número de rotaciones más el cambio de potencial es $R_f(X) - R_i(X) + 1$. El 1 final no se elimina al realizar sumas solapadas, por lo que no podríamos probar una cota logarítmica. Esto es correcto, pues ya sabíamos que en este caso no era posible obtener una cota logarítmica.

La técnica de análisis amortizado es muy interesante, y se han desarrollado algunos principios generales para formalizar el marco. En la bibliografía se pueden encontrar más detalles al respecto.

21.5 Árboles de ensanchamiento descendente

Como con los árboles rojinegros, los *árboles de ensanchamiento descendente* son más eficientes en la práctica que sus contrapartidas ascendentes.

Una implementación directa de la estrategia de ensanchamiento ascendente requiere una pasada hacia abajo para realizar el acceso seguida de una segunda pasada hacia arriba. Esto puede hacerse, bien manteniendo referencias al padre, almacenando el camino de acceso en una pila, o utilizando un truco más inteligente para almacenar el camino, utilizando referencias disponibles en los nodos accedidos. Desgraciadamente, todos estos métodos requieren una sustancial sobrecarga, siendo necesario tratar muchos casos especiales. Recuerde, de la Sección 18.5, que es mejor implementar los árboles de búsqueda utilizando una única pasada de arriba abajo. Para evitar casos especiales podemos utilizar nodos fantasma. Esta sección describe una estructura de *árbol de ensanchamiento descendente* que preserva la cota amortizada logarítmica. El procedimiento descendente es más rápido en la práctica y utiliza sólo un espacio extra constante. Es el método recomendado por los creadores de los árboles de ensanchamiento.

La idea básica en la que se basan los árboles de ensanchamiento descendente es que al descender en el árbol durante la búsqueda de un nodo X, debemos tomar los nodos que encontramos en el camino de acceso y moverlos a ellos y a sus subárboles del camino. Además, para garantizar la cota amortizada debemos realizar algunas rotaciones.

Se mantienen tres árboles durante la pasada descendente.

En cualquier instante en medio del ensanchamiento, hay un nodo actual X que es la raíz de su subárbol; éste se representa en los diagramas como el árbol central. El árbol L almacena los nodos menores que X, mientras que el árbol D almacena los nodos que son mayores que X. Inicialmente, X es la raíz del árbol T, y L y R son vacíos. Descendiendo por el árbol dos niveles cada vez, encontramos un par de nodos. Dependiendo de si esos dos nodos son menores o mayores que X, se colocan en L o R, junto con los subárboles que no están en el camino de acceso. Así, el nodo actual en el camino de búsqueda es siempre la raíz del árbol central. Cuando finalmente alcanzamos X, podemos unir L y R al fondo del árbol central. Como resultado, X se habrá movido a la raíz. Entonces, la cuestión es cómo se colocan los nodos en L y R, y cómo se realiza la concatenación final. Esto es lo que ilustran los árboles de la Figura 21.9. Como de costumbre, se omiten los casos simétricos.

En todos los dibujos, X es el nodo actual, Y su hijo y Z su nieto (en caso de que exista un nodo aplicable; el significado preciso de «aplicable» se hará más claro en la discusión del caso zig.)

Si la rotación debiera ser un zig, entonces el árbol cuya raíz es Y se convertiría en la nueva raíz del árbol central. X y el subárbol B se enlazan dando lugar al hijo izquierdo del menor elemento de R; y el hijo izquierdo de X se hace `null` [3]. Como resultado, X es el nuevo menor elemento de R, lo que hace que nuevas adiciones sean más fáciles.

Observemos que no es necesario que Y sea una hoja para que se aplique el caso zig. Si el nodo solicitado se encuentra en Y, el caso zig se aplicará, incluso si Y tiene hijos. El caso zig también se aplica si el elemento solicitado es menor que Y e Y no tiene hijo izquierdo, aunque Y tenga hijo derecho, y también en los casos simétricos.

[3] En el código presentado aquí, el nodo más pequeño de R no tiene una referencia izquierda `null` porque no es necesaria.

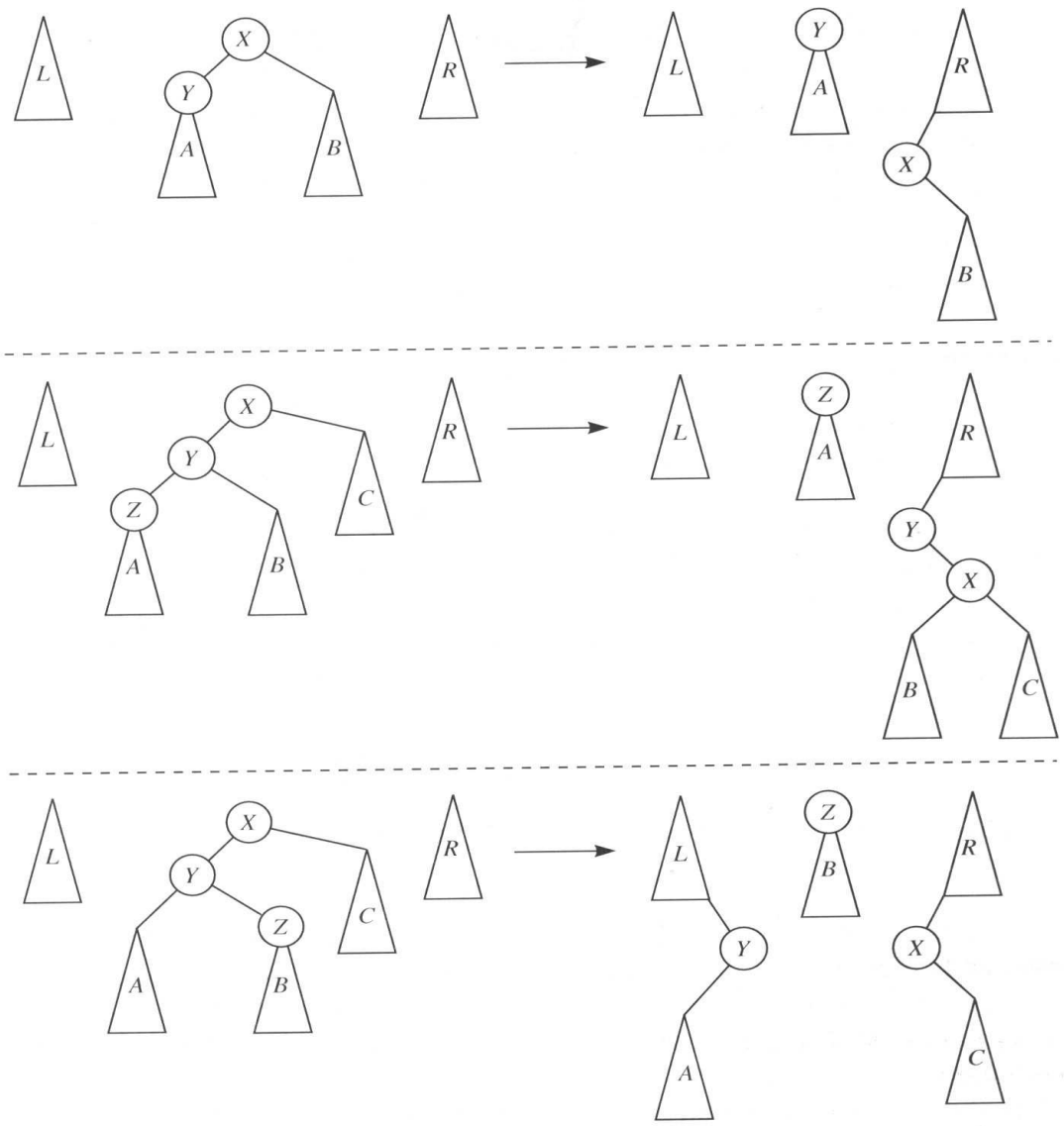


Figura 21.9 Rotaciones del ensanchamiento descendente: zig (arriba), zig-zig (en el centro) y zig-zag (abajo).

Algo similar ocurre en el caso zig-zig. El punto crucial es que se realiza una rotación entre X e Y. El caso zig-zag lleva el elemento Z del fondo arriba en el árbol central y enlaza los subárboles X e Y a R y L, respectivamente. Observemos que Y se convierte en el elemento mayor de L.

El caso zig-zag puede simplificarse algo porque no se realizan rotaciones. En vez de hacer que Z pase a ser la raíz del árbol central, tomaremos Y como raíz. Esto se muestra en la Figura 21.10. Esto simplifica el código, pues la acción en el caso zig-zag se hace idéntica al caso zig. Esto puede parecer ventajoso, ya que comprobar en qué estado estamos consume tiempo. La desventaja es que descender sólo un nodo da lugar a más iteraciones en el procedimiento de ensanchamiento.

Una vez hemos realizado el último paso del ensanchamiento, L, R y el árbol central se combinan para formar un único árbol, como muestra la Figura 21.11. Observe que el resultado es diferente al obtenido por el ensanchamiento ascendente. Sin embargo, las cotas amortizadas $O(\log N)$ se conservan (*véase* el Ejercicio 21.3).

Al final, los tres árboles se combinan en uno sólo.

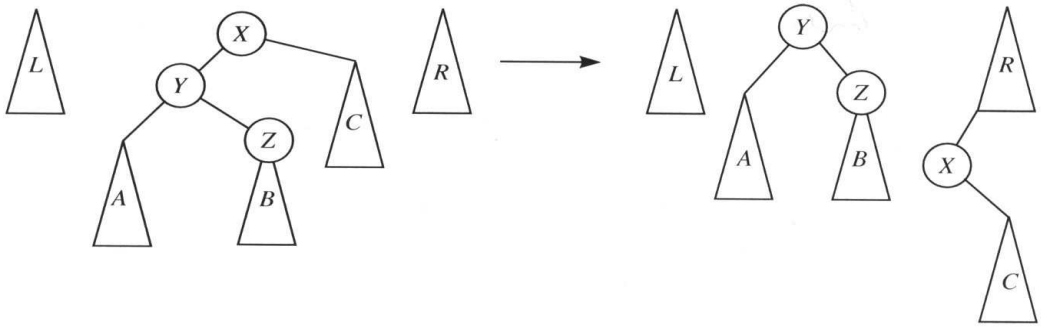


Figura 21.10 Zig-zag descendente simplificado.

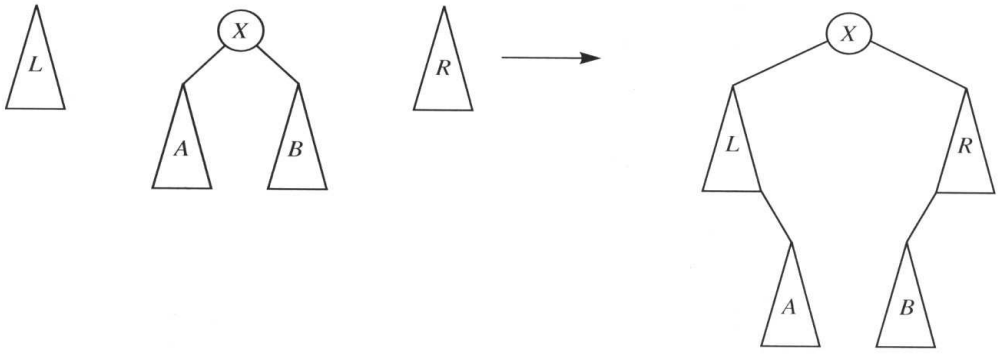


Figura 21.11 Montaje final del ensanchamiento descendente.

Un ejemplo del algoritmo de ensanchamiento descendente simplificado se muestra en la Figura 21.12. Intentamos acceder al 19 en el árbol. El primer paso es un zig-zag. De acuerdo con una versión simétrica de la Figura 21.20, traemos el subárbol con raíz 25 a la raíz del árbol central y enlazamos el 12 y su subárbol izquierdo a *L*.

A continuación, hacemos un zig-zig: el 15 se eleva a la raíz del árbol central y se realiza una rotación entre el 20 y el 25, enlazándose el árbol resultado a *R*. La búsqueda del 19 acaba con un zig. La nueva raíz del árbol central es 18, y 15 y su subárbol izquierdo se enlazan como hijo derecho del nodo mayor de *L*. El montaje, de acuerdo con la Figura 21.11, concluye el proceso de ensanchamiento.

21.6 Implementación de los árboles con ensanchamiento descendente

El esqueleto de la clase de los árboles con ensanchamiento se muestra en la Figura 21.13. Para eliminar molestos casos especiales, mantenemos un centinela estático `nodoNulo`. El centinela se coloca en memoria y se inicializa durante la carga de la clase, como muestra la Figura 21.14.

La Figura 21.15 muestra el método de inserción de un elemento `x`. Un nuevo nodo (`nodoNuevo`) se coloca en memoria y si el árbol es vacío se crea un árbol con un nodo. En otro caso, ensanchamos desde `x`. Si el elemento en la nueva raíz

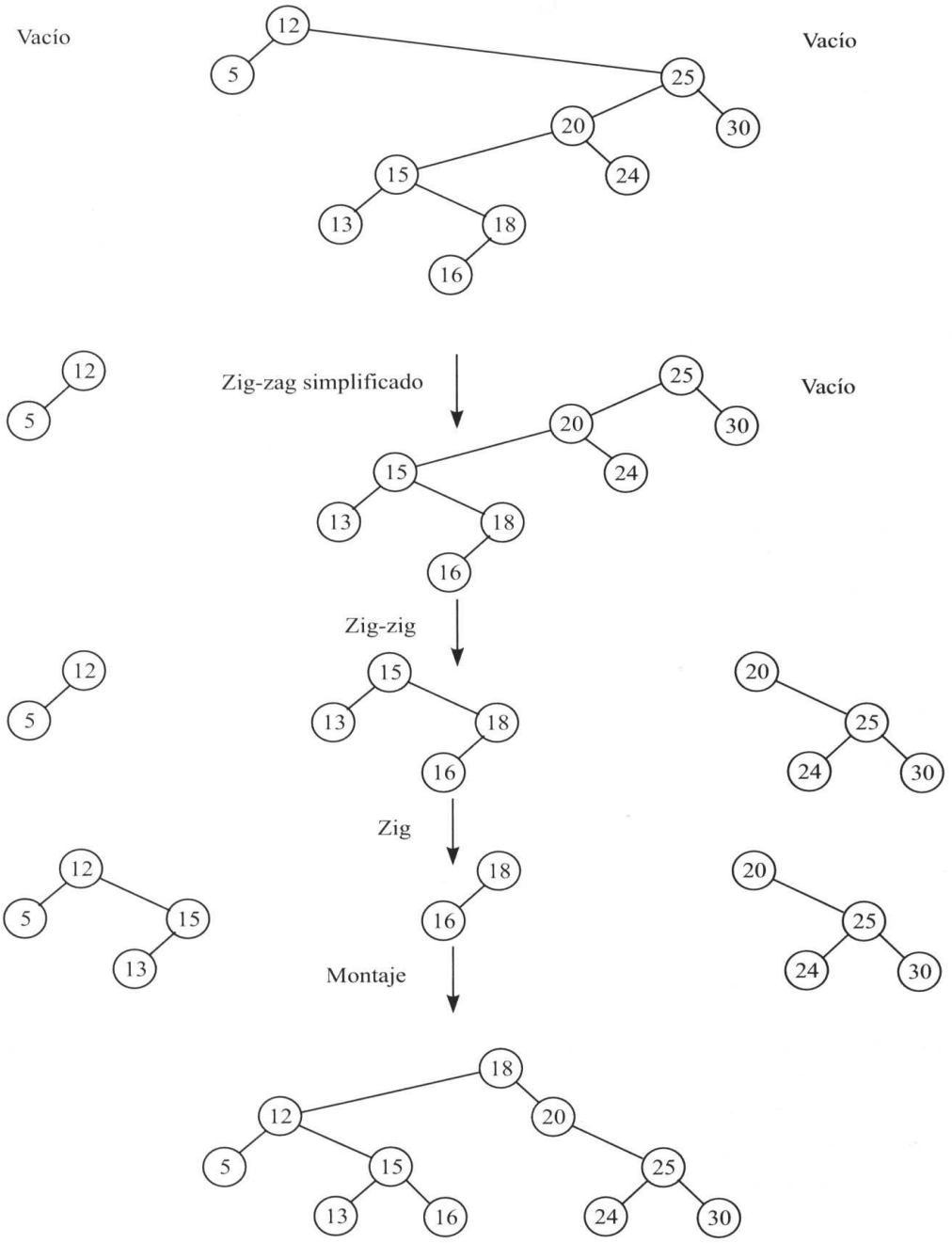


Figura 21.12 Pasos de un ensanchamiento descendente (accediendo al 19 del primer árbol).

del árbol es igual a `x`, tenemos un duplicado. En este caso, no queremos insertar `x` por lo que se lanza una excepción. Si la nueva raíz contiene un valor que es mayor que `x`, entonces la nueva raíz y su subárbol derecho se convierten en el un subárbol derecho de `nodoNuevo` y el subárbol izquierdo de la raíz se convierte en el subárbol izquierdo de `nodoNuevo`. Se aplica un razonamiento similar si la nueva raíz contiene un valor menor que `x`. En cualquier caso, `nodoNuevo` se asigna a `raiz` para indicar que es la nueva raíz.

```
1  package EstructurasDatos;
2
3  import Soporte.*; import Soporte.Comparable;
4  import Excepciones.*;
5
6  // Clase ArbolEnsanchamiento
7  //
8  // CONSTRUCCION: sin inicializador
9  //
10 // ******************OPERACIONES PÚBLICAS************************
11 // void insertar( x )       --> Inserta x
12 // void eliminar( x )       --> Elimina x
13 // void eliminarMin( )      --> Elimina el menor elemento
14 // Comparable buscar( x )   --> Devuelve el elemento que casa con x
15 // Comparable buscarMin( )  --> Devuelve el menor elemento
16 // Comparable buscarMax( )  --> Devuelve el mayor elemento
17 // boolean esVacio( )       --> Devuelve true si vacío; si no, false
18 // void vaciar( )           --> Elimina todos los elementos
19 // void imprimirArbol( )    --> Imprimir el árbol en orden
20 // ************************ERRORES*******************************
21 // Muchas rutinas lanzan ElementoNoEncontrado en casos degenerados
22 // insertar lanza ElementoDuplicado si el elemento ya está
23
24 /**
25  * Implementa un árbol con ensanchamiento descendente.
26  * Las comparaciones se basan en el método compara.
27  */
28 public class ArbolEnsanchamiento implements ArbolBusqueda
29 {
30     public ArbolEnsanchamiento ( )
31       { /* Figura 21.14 */ }
32     public void insertar( Comparable x ) throws ElementoDuplicado
33       { /* Figura 21.15 */ }
34     public void eliminar( Comparable x ) throws ElementoNoEncontrado
35       { /* Figura 21.16 */ }
36     protected NodoBinario ensanchamiento( Comparable x, NodoBinario t )
37       { /* Figura 21.18 */ }
38     public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
39       { /* Figura 21.17 */ }
40     public void eliminarMin( ) throws ElementoNoEncontrado
41       { eliminar( buscarMin( ) ); }
42     public boolean esVacio( )
43       { return raiz == nodoNulo; }
44     public void vaciar( )
45       { raiz = nodoNulo; }
46
47     // buscarMin, buscarMax e imprimirArbol están disponibles en Internet
48
49     private NodoBinario raiz;
50 }
```

Figura 21.13 Esqueleto de la clase de árboles con ensanchamiento descendente.

La Figura 21.16 muestra el algoritmo de eliminación de un árbol de ensanchamiento. Puede resultar chocante que el procedimiento de eliminación sea más corto que el correspondiente procedimiento de inserción. El método `buscar` se muestra en la Figura 21.17. Después del ensanchamiento, el elemento solicitado o bien es la raíz o no está en el árbol, por lo que el método es muy corto. Lo único

```
1  /**
2   * Construye el árbol.
3   */
4  public ArbolEnsanchamiento( )
5  {
6      raiz = nodoNulo;
7  }
8  private static NodoBinario nodoNulo;
9      static              // Inicializador estático para nodoNulo
10     {
11         nodoNulo = new NodoBinario( null );
12         nodoNulo.izquierdo = nodoNulo.derecho = nodoNulo;
13     }
```

Figura 21.14 Construcción del árbol de ensanchamiento, incluyendo la inicialización estática.

```
1  /**
2   * Inserta en el árbol.
3   * @param x el elemento a insertar.
4   * @exception ElementoDuplicado si un elemento que casa
5   *        con x ya está en el árbol.
6   */
7  public void insertar( Comparable x ) throws ElementoDuplicado
8  {
9      NodoBinario nodoNuevo = new NodoBinario( x );
10
11     if( raiz == nodoNulo )
12     {
13         nodoNuevo.izquierdo = nodoNuevo.derecho = nodoNulo;
14         raiz = nodoNuevo;
15     }
16     else
17     {
18         raiz = ensanchamiento( x, raiz );
19         if( x.menorQue( raiz.dato ) )
20         {
21             nodoNuevo.izquierdo = raiz.izquierdo;
22             nodoNuevo.derecho = raiz;
23             raiz.izquierdo = nodoNulo;
24             raiz = nodoNuevo;
25         }
26         else
27         if( raiz.dato.menorQue( x ) )
28         {
29             nodoNuevo.derecho = raiz.derecho;
30             nodoNuevo.izquierdo = raiz;
31             raiz.derecho = nodoNulo;
32             raiz = nodoNuevo;
33         }
34         else
35             throw new ElementoDuplicado( "Insertar en el árbol" );
36     }
37 }
```

Figura 21.15 Inserción en el árbol con ensanchamiento descendente.

```
1  /**
2    * Eliminación del árbol.
3    * @param x el elemento a eliminar.
4    * @exception ElementoNoEncontrado si no hay un
5    *        elemento que case con x en el árbol.
6    */
7  public void eliminar( Comparable x ) throws ElementoNoEncontrado
8  {
9      NodoBinario nuevoArbol;
10
11         // Si x se encuentra, estará en la raíz
12     raiz = ensanchamiento( x, raiz );
13     if( raiz.dato.compara( x ) != 0 )
14         throw new ElementoNoEncontrado( "Eliminar del árbol" );
15
16     if( raiz.izquierdo == nodoNulo )
17         nuevoArbol = raiz.derecho;
18     else
19     {
20         // Buscar el máximo del subárbol izquierdo
21         // Ensancharlo hacia la raiz; enlazar hijo derecho
22         nuevoArbol = raiz.izquierdo;
23         nuevoArbol = ensanchamiento( x, nuevoArbol );
24         nuevoArbol.derecho = raiz.derecho;
25     }
26     raiz = nuevoArbol;
27 }
```

Figura 21.16 Eliminación en el árbol de ensanchamiento descendente.

```
1  /**
2   * Buscar un elemento en el árbol.
3   * @param x el elemento a buscar.
4   * @return el elemento encontrado.
5   * @exception ElementoNoEncontrado si no hay un
6   *        elemento que case con x en el árbol.
7   */
8  public Comparable buscar( Comparable x ) throws ElementoNoEncontrado
9  {
10     raiz = ensanchamiento( x, raiz );
11
12     if( esVacio( ) || raiz.dato.compara( x ) != 0 )
13         throw new ElementoNoEncontrado( "Buscar en el árbol" );
14
15     return raiz.dato;
16 }
```

Figura 21.17 Método `buscar` de los árboles de ensanchamiento.

que nos falta es la rutina de ensanchamiento descendente. Se implementa como método protegido para que, si se desea, nuestra clase pueda ser extendida con un algoritmo de ensanchamiento diferente.

Nuestra implementación, mostrada en la Figura 21.18, utiliza una cabecera con referencias izquierda y derecha que a la postre contendrá las raíces de los árboles izquierdo y derecho. `cabecera` es un atributo estático, ya que queremos colocarlo en memoria sólo una vez durante la secuencia de ensanchamientos. Estos árboles

```
1  private static NodoBinario cabecera = new NodoBinario( null );
2
3  /**
4   * Método interno para realizar el ensanchamiento.
5   * El último nodo accedido se convierte en la raíz.
6   * @param x el elemento al que se refiere el ensanchamiento.
7   * @param t la raíz del subárbol que hay que ensanchar.
8   * @return el subárbol después del ensanchamiento.
9   */
10 protected NodoBinario ensanchamiento( Comparable x, NodoBinario t )
11 {
12     NodoBinario maxArbolIzq, minArbolDer;
13
14     cabecera.izquierdo = cabecera.derecho = nodoNulo;
15     maxArbolIzq = minArbolDer = cabecera;
16
17     nodoNulo.dato = x; // Garantiza un emparejamiento
18
19     for( ; ; )
20         if( x.menorQue( t.dato ) )
21         {
22             if( x.menorQue( t.izquierdo.dato ) )
23                 t = Rotaciones.conHijoIzquierdo( t );
24             if( t.izquierdo == nodoNulo )
25                 break;
26             // Enlazar derecho
27             minArbolDer.izquierdo = t;
28             minArbolDer = t;
29             t = t.izquierdo;
30         }
31         else if( t.dato.menorQue( x ) )
32         {
33             if( t.derecho.dato.menorQue( x ) )
34                 t = Rotaciones.conHijoDerecho( t );
35             if( t.derecho == nodoNulo )
36                 break;
37             // Enlazar izquierdo
38             maxArbolIzq.derecho = t;
39             maxArbolIzq = t;
40             t = t.derecho;
41         }
42         else
43             break;
44
45     maxArbolIzq.derecho = t.izquierdo;
46     minArbolDer.izquierdo = t.derecho;
47     t.izquierdo = cabecera.derecho;
48     t.derecho = cabecera.izquierdo;
49     return t;
50 }
```

Figura 21.18 Algoritmo de ensanchamiento descendente.

son inicialmente vacíos, por lo que la cabecera se utiliza para señalar al mínimo y máximo nodo del árbol izquierdo y derecho, respectivamente, en este estado inicial. De esta manera, el código puede evitar comprobaciones de árboles vacíos. La primera vez que el árbol izquierdo deja de ser vacío, la referencia derecha de la cabecera se inicializará y no cambiará en el futuro. Por tanto, contendrá la raíz

del hijo izquierdo al final de la búsqueda descendente. De forma similar, la referencia izquierda de la cabecera contendrá a la postre la raíz del árbol derecho.

Antes del montaje al final del ensanchamiento, `cabecera.izquierdo` y `cabecera.derecho` referenciarán a R y L, respectivamente (esto no es un error tipográfico, siga los enlaces). Tenga en cuenta que estamos utilizando el ensanchamiento simplificado.

21.7 Comparaciones de los árboles de ensanchamiento con otros árboles de búsqueda

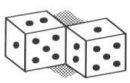
La implementación en la sección anterior sugiere que los árboles de ensanchamiento no son tan complicados como los árboles rojinegros y son casi tan simples como los AA-árboles. ¿Merece la pena utilizarlos? La respuesta definitiva está aún por obtener, pero parece que si los accesos no son aleatorios, los árboles de ensanchamiento tienen en efecto muy buen rendimiento en la práctica. Es más, algunas propiedades relativas a su buen rendimiento pueden probarse analíticamente. Los accesos no aleatorios incluyen aquéllos que siguen la regla del 90-10, así como diversos casos especiales como accesos secuenciales, accesos por los dos extremos y patrones de accesos que son típicos de las colas de prioridad durante ciertos tipos de simulación de eventos. Los ejercicios le piden estudiar esto con más detalle.

Sin embargo, los árboles de ensanchamiento no son perfectos. Un problema es que, debido al ensanchamiento, la operación `buscar` es costosa. En consecuencia, cuando las secuencias de accesos son aleatorias y uniformes, los árboles de ensanchamiento no son tan buenos como otros árboles equilibrados.

Resumen

Este capítulo describe los árboles de ensanchamiento, los cuales son una reciente alternativa a los árboles de búsqueda equilibrados. Los árboles de ensanchamiento tienen algunas propiedades notables que pueden ser demostradas, incluyendo un coste amortizado logarítmico por operación. Otras propiedades se indican en los ejercicios. Algunos estudios han sugerido que los árboles de ensanchamiento pueden utilizarse en un amplio rango de aplicaciones debido a su aparente habilidad de adaptarse a las secuencias de accesos sencillas.

El próximo capítulo describe dos estructuras de colas de prioridad que, como los árboles de ensanchamiento, tienen rendimientos pobres en el caso peor, pero buenos rendimientos amortizados. Una de éstas, el montículo de emparejamientos, parece ser una elección excelente en algunas aplicaciones.



Elementos del juego

análisis amortizado Método de análisis que acota el coste de una secuencia de operaciones y distribuye equitativamente el coste entre las operaciones de la secuencia.

árbol de ensanchamiento ascendente Árbol en el cual los elementos son rotados hacia la raíz utilizando un método algo más complicado que el utilizado en la rotación sencilla hacia la raíz.

árbol de ensanchamiento descendente Tipo de árbol de ensanchamiento más eficiente en la práctica que su contrapartida ascendente, como ocurrió en el caso de los árboles rojinegros.

ensanchamiento Estrategia de rotación hacia la raíz que permite obtener una cota amortizada logarítmica.

estrategia de rotación hacia la raíz Estrategia que reorganiza un árbol binario de búsqueda después de cada acceso de forma que desplaza los elementos más frecuentemente accedidos cerca de la raíz.

función de potencial Herramienta de contabilidad utilizada para probar una cota amortizada.

rango En el análisis de un árbol de ensanchamiento, el logaritmo del tamaño de un nodo.

regla del 90-10 Regla que afirma que el 90 por ciento de los accesos corresponden al 10 por ciento de los datos. Los árboles de búsqueda equilibrados no sacan partido de esta regla.

zig y **zig-zag** Los casos zig y zig-zag son idénticos a los casos de la rotación hacia la raíz. El caso zig se utiliza cuando *X* es un hijo de la raíz, y el zig-zag cuando *X* es un nodo más interno.

zig-zig El caso zig-zig es exclusivo de los árboles de ensanchamiento. Se utiliza cuando *X* es un nodo externo.

Errores comunes

1. Después de cada acceso se tiene que realizar un ensanchamiento, aun en el caso de los que no tengan éxito, pues si no, no son válidas las cotas de rendimiento.
2. El código es un tanto rebuscado. Sea cuidadoso.

En Internet

La clase de árboles de ensanchamiento está disponible en Internet en el directorio **DataStructures**. El nombre del fichero es el siguiente:

SplayTree.java Traducido como `ArbolEnsanchamiento.java`, contiene la implementación de la clase de los árboles de ensanchamiento.

Ejercicios

Cuestiones breves

21.1. Muestre el resultado de insertar los valores 3, 1, 4, 5, 2, 9, 6 y 8 en un

a) árbol de ensanchamiento ascendente.
b) árbol de ensanchamiento descendente.

21.2. Muestre el resultado de eliminar el 3 del árbol del Ejercicio 21.1 en ambos casos.

Problemas teóricos

21.3. Demuestre que el coste amortizado de los árboles de ensanchamiento descendente es $O(\log N)$.

21.4. Demuestre que si todos los nodos de un árbol de ensanchamiento son accedidos en orden secuencial, el árbol resultante estará formado por una cadena de hijos izquierdos.

21.5. Supongamos que, en un intento de ahorrar tiempo, ensanchamos sólo en una de cada dos operaciones. ¿Sigue siendo logarítmico el coste amortizado?

21.6. Los nodos del 1 al $N = 1.024$ forman un árbol de ensanchamiento de hijos izquierdos.

a) ¿Cuál es la longitud exacta del camino interno del árbol?
b) Calcule la longitud del camino interno después de las operaciones `buscar(1)`, `buscar(2)`, `buscar(3)`, cuando se utiliza un ensanchamiento ascendente.

21.7. Cambiando la función de potencial, se pueden obtener diferentes cotas para el ensanchamiento. Sea $P(i)$ una función peso asignada a cada nodo del árbol y $S(i)$ la suma de los pesos de todos los nodos en el árbol cuya raíz es i, incluyendo a i. El caso especial $P(i) = 1$ para todos los nodos corresponde a la función utilizada en la demostración de la cota del ensanchamiento. Demuestre los siguientes teoremas:

a) El tiempo total de acceso es $O(M + (M + N) \log N)$.
b) Si q_i es el número total de veces que se accede al nodo i y $q_i > 0$ para todo i, el tiempo de acceso total es $O(M + \Sigma_{i=1}^{N} q_i \log (M/q_i))$.

Problemas prácticos

21.8. Implemente la clase de las colas de prioridad utilizando un árbol de ensanchamiento.

21.9. Modifique los árboles de ensanchamiento para que soporten accesos indexados según el orden de los elementos en el árbol.

Prácticas de programación

21.10. Compare empíricamente los árboles de ensanchamiento descendente que se implementan en la Sección 21.6 con los árboles originales de ensanchamiento ascendente discutidos en la Sección 21.5.

21.11. Al contrario que los árboles de búsqueda equilibrados, los árboles de ensanchamiento incurren en una sobrecarga durante la operación `buscar` que es indeseable si la secuencia de acceso es suficientemente aleatoria. Experimente con una estrategia que ensanche en la operación `buscar` sólo después de rebasar cierta profundidad d en la búsqueda descendente. El ensanchamiento no mueve el elemento buscado a la raíz, sino al punto de profundidad d donde empezó el ensanchamiento.

21.12. Compare empíricamente una implementación de las colas de prioridad utilizando un árbol de ensanchamiento con un montículo binario. Utilice los siguientes modelos de entrada:

a) Inserciones y operaciones `eliminarMin` aleatorias.
b) Operaciones de `insertar` y `eliminarMin` correspondientes a una simulación dirigida por eventos.
c) Operaciones de `insertar` y `eliminarMin` correspondientes al algoritmo de Dijkstra.

21.13. Diseñe un applet que ilustre las operaciones de los árboles de ensanchamiento. Puede utilizar el ensanchamiento ascendente o descendente.

Bibliografía

Los árboles de ensanchamiento se describen en el artículo [3]. El concepto de análisis amortizado se discute en el artículo [4] y también con gran detalle en [5]. Una comparación entre los árboles de ensanchamiento y los árboles AVL se presenta en [1]. [2] muestra que los árboles de ensanchamiento se comportan bien en algunos tipos de simulación dirigida por eventos.

1. J. Bell y G. Gupta, «An Evaluation of Self-Adjusting Binary Search Tree Techniques», *Software-Practice and Experience* **23** (1993), 369-382.

2. D. W. Jones, «An Empirical Comparison of Priority-Queue and Event-Set Implementations», *Communications of the ACM* **29** (1986), 300-311.

3. D. D. Sleator y R. E. Tarjan, «Self-adjusting Binary Search Trees», *Journal of te ACM* **32** (1985), 652-686.

4. R. E. Tarjan, «Amortized Computational Complexity», *SIAM Journal on Algebraic and Discrete Methods* **6** (1985), 306-318.

5. M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, Benjamin/Cummings Publishing Co., Redwood City, Calif. (1994).