En cuanto al tiempo de ejecución, básicamente podemos aplicar el mismo análisis que en la inserción. El tiempo total para una supresión será proporcional al nivel de la clave buscada en el AVL, puesto que el tiempo en cada nivel es constante (o limitado por una constante). Como el nivel máximo está en un $O(\log n)$, el tiempo de ejecución será también un $O(\log n)$.

Ejemplo 4.3 En un árbol AVL inicialmente vacío almacenamos números enteros. Insertamos los siguientes elementos: 7, 14, 32, 9, 40, 8, 3, 4. Después eliminamos del árbol los valores: 9, 32, 3. Vamos a ver la estructura del árbol después de aplicar cada operación, indicando los casos en los que ocurre desbalanceo y la rotación que se aplica.

El resultado aparece en la figura 4.22. Se puede ver que en la inserción se requieren en total 3 rotaciones, para 8 inserciones. Por otro lado, de las 3 eliminaciones ocurren desbalanceos en 2 de ellas.

4.4. Árboles B

Los **árboles B** [14] constituyen una categoría muy importante de estructuras de datos, que permiten una implementación eficiente de conjuntos y diccionarios, para operaciones de consulta y acceso secuencial. Existe una gran variedad de árboles B: los árboles B, B+ y B*; pero todas ellas están basadas en la misma idea, la utilización de árboles de búsqueda no binarios y con condición de balanceo.

En concreto, los árboles B+ son ampliamente utilizados en la representación de índices en bases de datos. De hecho, este tipo de árboles están diseñados específicamente para aplicaciones de bases de datos, donde la característica fundamental es la predominancia del tiempo de las operaciones de entrada/salida de disco en el tiempo de ejecución total. En consecuencia, se busca minimizar el número de operaciones de lectura o escritura de bloques de datos del disco duro o soporte físico.

4.4.1. Árboles de búsqueda no binarios

En cierto sentido, los árboles B se pueden ver como una generalización de la idea de árbol binario de búsqueda a árboles de búsqueda no binarios. Consideremos la representación de árboles de búsqueda binarios y n-arios de la figura 4.23.

En un ABB (figura 4.23a) si un nodo x tiene dos hijos, los nodos del subárbol izquierdo contienen claves menores que x y los del derecho contienen claves mayores que x. En un árbol de búsqueda no binario (figura 4.23b), cada nodo interno puede contener q claves x_1, x_2, …, x_q, y $q+1$ punteros a hijos que están "situados" entre cada par de claves y en los dos extremos. Las claves estarán ordenadas de menor a mayor: $x_1 < x_2 < \ldots < x_q$. Además, el hijo más a la izquierda contiene claves menores que x_1; el hijo entre x_1 y x_2 contiene claves mayores que x_1 y menores que x_2; y así sucesivamente hasta el hijo más a la derecha que contiene claves mayores que x_q.

[14] La B viene de *balanceados*.

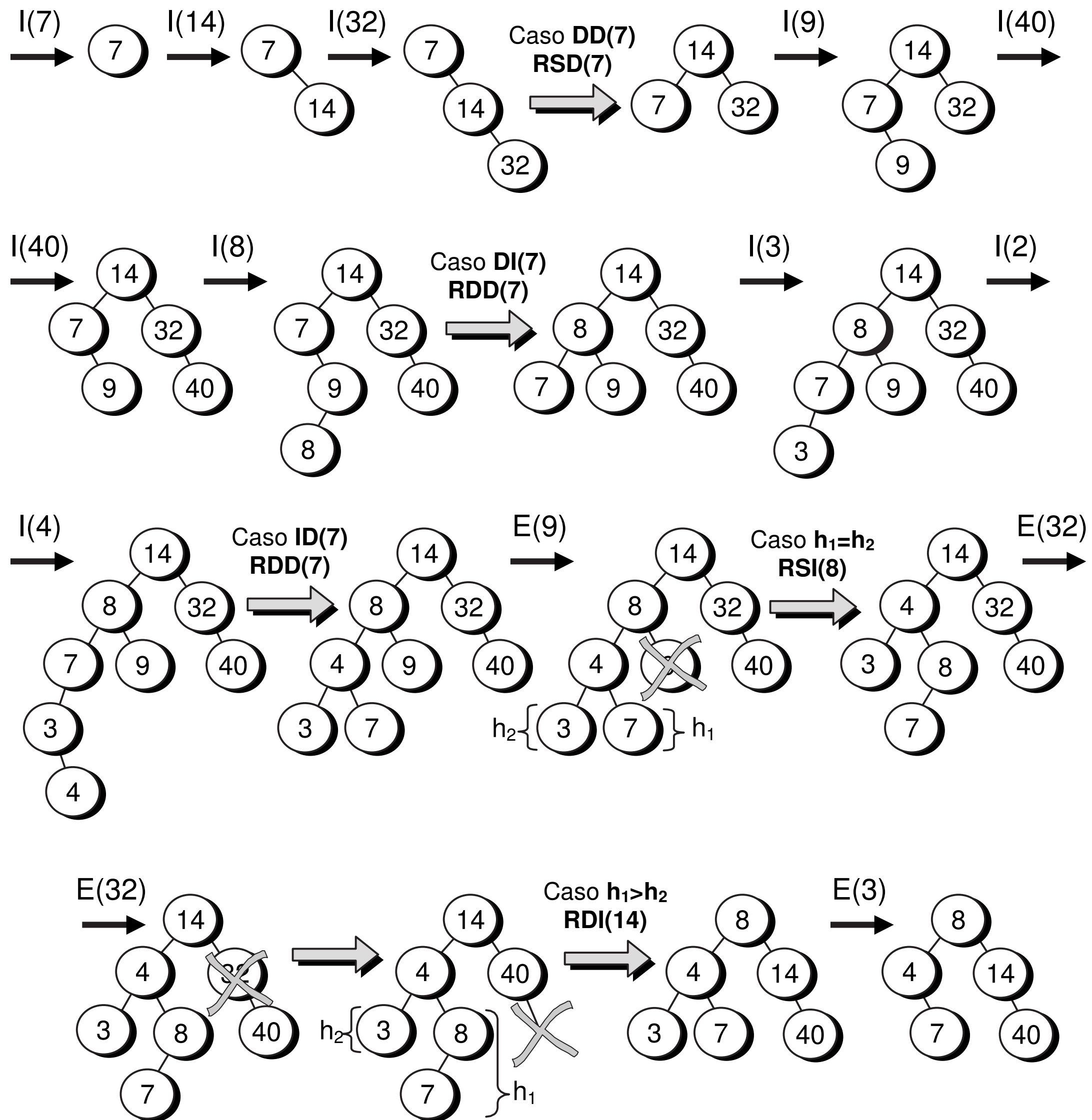

Figura 4.22: Inserción y eliminación de elementos en un AVL, según las operaciones del ejemplo 4.3.

Definición de árbol B

Un **árbol B de orden *p*** es básicamente un árbol de búsqueda n-ario donde los nodos tienen p hijos como máximo, y en el cual se añade la condición de balanceo de que todas las hojas estén al mismo nivel. La definición formal de árbol B es la siguiente.

Definición 4.6 Un **árbol B de orden** p, siendo p un entero mayor que 2, es un árbol con las siguientes características:

1. Los nodos internos son de la forma (p_1, x_1, p_2, x_2, ..., x_{q-1}, p_q), siendo p_i punteros a nodos hijos y x_i claves, o pares $(clave, valor)$ en caso de representar diccionarios.

2. Si un nodo tiene q punteros a hijos, entonces tiene $q - 1$ claves.

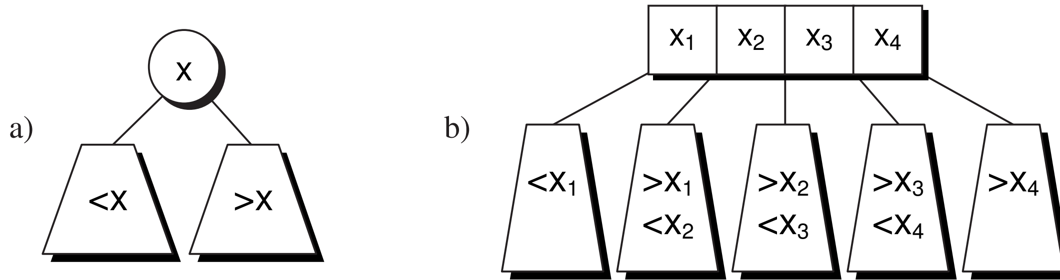

Figura 4.23: Estructura de los árboles de búsqueda. a) Árbol binario de búsqueda. b) Árbol de búsqueda n-ario.

3. Para todo nodo interno, excepto para la raíz, $\lceil p/2 \rceil \leq q \leq p$. Es decir, un nodo puede tener como mínimo $\lceil p/2 \rceil$ hijos y como máximo p. El valor $\lceil p/2 \rceil - 1$ indicaría el mínimo número de claves en un nodo interno y se suele denotar por d.

4. La raíz puede tener 0 hijos (si es el único nodo del árbol) o entre 2 y p.

5. Los nodos hoja tienen la misma estructura, pero todos sus punteros son nulos.

6. En todos los nodos se cumple: $x_1 < x_2 < \ldots < x_{q-1}$.

7. Para todas las claves k apuntadas por un puntero p_i de un nodo se cumple:
   - Si $i = 1$ entonces $k < x_1$.
   - Si $i = q$ entonces $x_{q-1} < k$.
   - En otro caso, $x_{i-1} < k < x_i$.

8. Todas los nodos hoja están al mismo nivel en el árbol.

Los **árboles B+** son una variante de los árboles B, que se utiliza en representación de diccionarios. La estructura de los nodos hoja es distinta de la de los nodos internos. En esencia, la modificación consiste en que en los nodos internos sólo aparecen claves, mientras en los nodos hoja aparecen las asociaciones $(clave, valor)$.

Por otro lado, tenemos la variante de los **árboles B***. Su principal característica es que si en los árboles B –teniendo en cuenta la anterior definición– los nodos deben estar ocupados como mínimo hasta la mitad, en los B* tienen que estar ocupados más de dos tercios del máximo. En adelante nos centraremos en el estudio de los árboles B. En la figura 4.24 se muestra un ejemplo de árbol B de orden $p = 5$.

Representación del tipo árbol B

Pasando ahora a la implementación de árboles B de cierto orden p, lo normal es reservar espacio de forma fija para p hijos y $p - 1$ elementos. La definición, suponiendo que el tipo almacenado es T, podría ser como la siguiente.

tipo

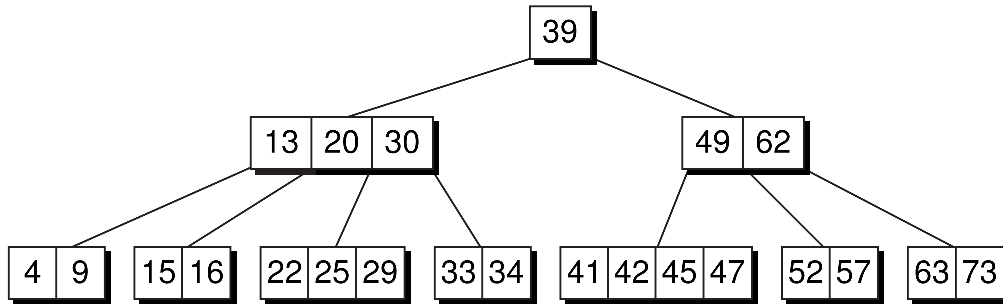

Figura 4.24: Ejemplo de árbol B de orden $p = 5$. Cada nodo interno excepto la raíz, debe tener entre 3 y 5 hijos o, equivalentemente, entre 2 y 4 entradas.

```
ArbolB[T,p] = Puntero[NodoArbolB[T,p]]
NodoArbolB[T,p] = registro
    q: entero    // Número de punteros no nulos del nodo
    x: array [1..p-1] de T
    ptr: array [1..p] de ArbolB[T,p]
finregistro
```

En aplicaciones reales de bases de datos, el valor de p se ajusta de forma que un nodo ocupe exactamente un bloque de disco. Por ejemplo, si suponemos que los enteros, punteros y valores de tipo T ocupan 4 bytes, un nodo sería de tamaño $2 * 4p$ bytes; si los bloques de disco son de 4.096 bytes, entonces usaríamos un $p = 512$.

La implementación de la operación de búsqueda consistiría simplemente en empezar por la raíz y descender hacia la rama correspondiente, según el valor de la clave buscada. Por ejemplo, la operación Miembro sobre conjuntos podría ser como la siguiente.

```
operación Miembro (b: ArbolB[T,p]; c: T): booleano
    si b = NULO entonces
        devolver false
    sino
        para i:= 1, ..., b↑.q-1 hacer
            si c = b↑.x[i] entonces
                devolver verdadero
            sino si c < b↑.x[i] entonces
                devolver Miembro(b↑.ptr[i], c)
            finsi
        finpara
        devolver Miembro(b↑.ptr[b↑.q], c)
    finsi
```

Claramente, el tiempo de ejecución depende de la altura del árbol que, como veremos adelante, crece logarítmicamente con el número de nodos. En concreto, por cada nivel de la clave buscada habrá una llamada recursiva a Miembro.

4.4.2. Inserción en un árbol B

Además de mantener la estructura de árbol de búsqueda, el procedimiento de inserción en un árbol B debe asegurar la propiedad que impone que todas las hojas estén al mismo nivel. La nueva entrada debe insertarse siempre en un nodo hoja[15]. Si hay sitio en la hoja que le corresponde, el elemento se puede insertar directamente.

En otro caso aplicamos un proceso de **partición de nodos**, que es mostrado en la figura 4.25. El proceso consiste en lo siguiente: con las $p-1$ entradas de la hoja donde se hace la inserción más la nueva entrada, se crean dos nuevas hojas con $\lceil (p-1)/2 \rceil$ y $\lfloor (p-1)/2 \rfloor$ entradas. La entrada m que está en la mediana aparece como una nueva clave de nivel superior, y tiene como hijas las dos hojas recien creadas.

El proceso se repite recursivamente en el nivel superior, en el que habrá que insertar m. Si m cabe en el nodo interno correspondiente, se inserta. En otro caso, se parte el nodo interno y se repite el proceso hacia arriba. Si se produce la partición a nivel de la raíz entonces tenemos el caso donde la profundidad del árbol aumenta en uno.

En definitiva, el esquema del algoritmo de inserción de una clave x en un árbol B de orden p sería el siguiente.

1. Buscar el nodo hoja donde se debería colocar x, usando un procedimiento parecido a la operación Miembro. Si el elemento ya está en el árbol, no se vuelve a insertar.

2. Si la hoja contiene menos de $p-1$ entradas, entonces quedan sitios libres. Insertar x en esa hoja, en la posición correspondiente.

3. Si no quedan sitios libres, cogemos las $p-1$ entradas de la hoja y x. La mediana m pasa al nodo padre, así como los punteros a sus nuevos hijos: los valores menores que m forman el nodo hijo de la izquierda y los valores mayores que m forman el nodo hijo de la derecha.

4. Si el nodo padre está completo, se dividirá a su vez en dos nodos, propagándose el proceso de partición hasta la raíz.

El algoritmo garantiza las propiedades de árbol B: todas las hojas están al mismo nivel y los nodos internos (excepto, posiblemente, la raíz) están llenos como mínimo hasta la mitad. Por otro lado, el tiempo de ejecución depende de la altura del árbol. Pero, además, el tiempo en cada nivel no es constante; el proceso de partición tardará un $O(p)$ en el peor caso. No obstante, ya hemos visto que realmente el factor a considerar es el número de nodos tratados, más que las operaciones que se realicen dentro de cada nodo.

4.4.3. Eliminación en un árbol B

Igual que la inserción en un árbol B hace uso de la partición de un nodo en dos, la eliminación se caracteriza por el proceso de **unión de dos nodos** en uno nuevo, en caso de que el nodo se vacíe hasta menos de la mitad. No obstante, hay que tener en cuenta todas las situaciones que pueden ocurrir en la supresión.

[15]Ya que recordemos que los nodos internos deben tener un puntero más que claves.

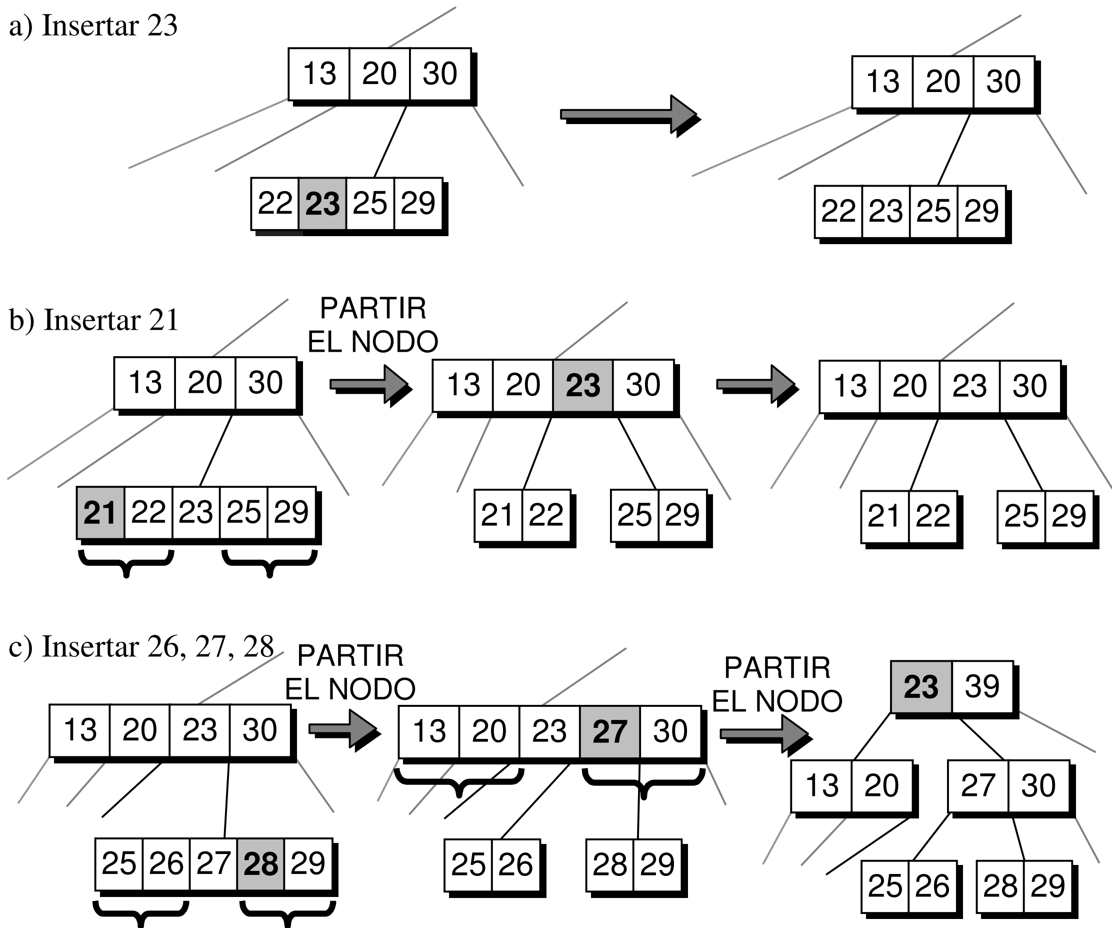

Figura 4.25: Inserción de claves en el árbol B de orden $p = 5$ de la figura 4.24. a) Inserción sin necesidad de partición. b) La inserción de 21 obliga a partir una hoja. c) Después de insertar 28, hay que partir la hoja, y después otra vez a nivel superior.

Dada una clave x a eliminar de un árbol B, en primer lugar debemos buscar el nodo donde se encuentra x. Si se encuentra dentro de un nodo interno, no se puede suprimir de forma directa. En ese caso, habrá que sustituir x en el árbol por la siguiente clave en orden –es decir, la mayor del subárbol izquierdo o la menor del subárbol derecho de x– y se continúa con la eliminación como si se hubiera producido en la posición sustituida. Por ejemplo, si en el árbol de la figura 4.24 eliminamos la clave 39, deberíamos colocar en su lugar 34 o bien 41, y seguir con el proceso de supresión a partir de la hoja correspondiente. De esta forma, el grueso del proceso de eliminación siempre parte de un nodo hoja.

Si la clave a eliminar está en una hoja –a la que llamamos la **hoja de supresión**– o se ha aplicado la sustitución explicada antes, entonces podemos encontrarnos varios casos. Recordemos que un nodo debe contener como mínimo $d = \lceil p/2 \rceil - 1$ entradas. Los casos dependen del número de entradas de la hoja de supresión, en relación con d.

- Si la hoja de supresión tiene más de d entradas, se puede eliminar la clave directamente. Acabaría la operación sin más modificaciones.

- Si la hoja contiene exactamente d entradas, entonces al eliminar la clave se queda con $d - 1$. Será necesario "añadirle" alguna entrada, pero ¿cuál? Algún nodo hermano podría "prestarle" una entrada, si tiene alguna de sobra. Pueden ocurrir dos casos.
  - Si existe algún nodo hermano, adyacente a la hoja de supresión y que tenga más de d entradas, entonces le hace un **préstamo**: la entrada del padre común pasa a la hoja de supresión y la entrada adecuada del nodo hermano[16] pasa a la posición del padre. Este proceso se muestra en la figura 4.26b).
  - Si todos los hermanos adyacentes a la hoja de supresión tienen d entradas, entonces no se puede producir el préstamo. En ese caso, la solución es **unir dos nodos en uno**. Con las $d - 1$ entradas del nodo de supresión, más las d entradas de un hermano y la entrada del padre común, se forma un nuevo nodo con $2d$ entradas. En la figura 4.26c) aparece un ejemplo de supresión que acarrea una unión de nodos.

Hay que tener en cuenta que el último caso, la unión de dos nodos, da lugar a la eliminación de una entrada a nivel superior. Por lo tanto, el proceso de eliminación debería repetirse en ese nivel superior. Es decir, si el nodo interno –el padre de la hoja de supresión– contiene más de d entradas, se puede eliminar directamente. Si tiene d entradas, entonces ocurrirá uno de los dos casos anteriores: si algún hermano tiene más de d entradas le presta una; y en otro caso se juntan dos nodos en uno. En definitiva, el proceso se iría repitiendo sucesivamente desde las hojas hasta la raíz.

Está claro que el nodo raíz no tiene hermanos, por lo que nunca podrá recibir préstamos o unirse con un hermano. Básicamente, esta es la razón por la que en los árboles B se permite que la raíz tenga menos de d entradas.

Si en el proceso de eliminación se suprime una entrada de la raíz (al juntarse dos hijos suyos) y la raíz sólo tenía esa entrada, entonces tenemos un caso donde la altura del árbol disminuye en uno. Por ejemplo, si en el árbol de la figura 4.26d) eliminamos el valor 63, habría que unir nodos, formando una hoja con 47, 52, 62 y 73. La unión se repetiría a nivel superior, dando lugar a un nuevo nodo interno con 20, 30, 39 y 45. Este nodo sería la nueva raíz, de manera que el árbol tendría ahora altura uno.

4.4.4. Análisis de eficiencia de los árboles B

En el análisis de eficiencia de los árbol B, el recurso crítico es el número de lecturas/escrituras de bloques del disco duro. Típicamente, las operaciones de acceso a disco son varios órdenes de magnitud más lentas que las que se realizan en memoria. En las implementaciones reales y eficientes de árboles B, se hace que cada nodo del árbol ocupe exactamente un bloque de disco, lo cual se consigue ajustando el parámetro p. Por lo tanto, hasta cierto límite, el tiempo de las instrucciones que se ejecutan dentro de cada nodo es despreciable y el factor importante es el número de nodos tratados[17].

[16] Es decir, la mayor o la menor, según el nodo hermano esté a la izquierda o a la derecha del nodo de supresión.

[17] Podríamos decir algo parecido de las demás estructuras de datos estudiadas en este y en otros capítulos. La diferencia es que esas otras estructuras suelen encontrarse en aplicaciones que hacen uso de disco o no, mientras que los árboles B son típicos de aplicaciones de BB.DD.

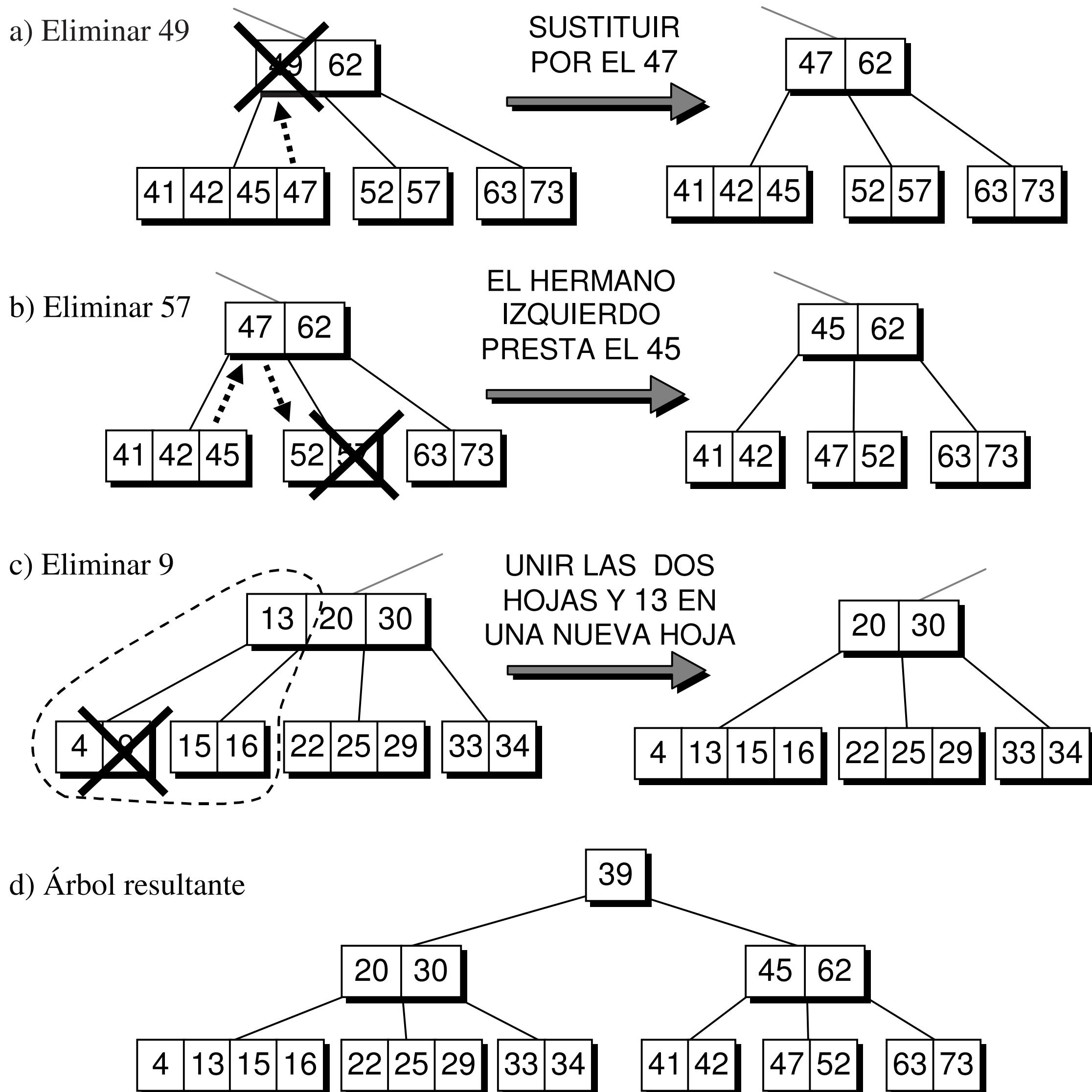

Figura 4.26: Eliminación de claves en el árbol B de orden $p = 5$ de la figura 4.24. a) Eliminación en un nodo interno. b) La eliminación de 57 requiere un préstamo del hermano. c) La eliminación de 9 produce una unión de hojas. d) Resultado final.

Análisis de tiempos de ejecución

Igual que con las restantes estructuras arbóreas, el número de nodos recorridos en las distintas operaciones es proporcional a la altura del árbol. En concreto, si la altura del árbol es h, la operación de búsqueda de una clave accederá a $h + 1$ nodos; la inserción en el peor caso tratará $2h + 1$ nodos, si se debe hacer la partición a todos los niveles; y, de forma similar, la eliminación visitará $3h + 1$ nodos en el peor caso.

Para obtener el número de nodos recorridos en función del número n de claves almacenadas en el árbol B, vamos a calcular primero la función inversa, es decir el número n de claves que caben para una cierta altura h. Podemos distinguir dos casos: en el mejor,

n_{mejor}, cada nodo interno tendrá p hijos (el máximo); y en el peor, n_{peor}, tendrá 2 ó $d+1$ si es la raíz o no (el mínimo), respectivamente. Contando el número de claves almacenadas en cada nivel, para una altura total h, tenemos lo siguiente.

Nivel	n_{mejor}	n_{peor}
0	$p-1$	1
1	$(p-1)p$	$2d$
2	$(p-1)p^2$	$2d(d+1)$
3	$(p-1)p^3$	$2d(d+1)^2$
...	...	...
h	$(p-1)p^h$	$2d(d+1)^{h-1}$

Sumando todas las claves en las distintas alturas obtenemos n_{mejor} y n_{peor}.

$$n_{mejor}(h) = (p-1)\sum_{i=0}^{h} p^i = p^{h+1} - 1 \tag{4.4}$$

$$n_{peor}(h) = 1 + 2d\sum_{i=0}^{h-1}(d+1)^i = 2(d+1)^h - 1 \tag{4.5}$$

Ahora podemos despejar la altura h y expresarla en función de n, que es lo que realmente nos interesa. Tenemos.

$$h_{mejor}(n) = \lceil \log_p(n+1) \rceil - 1 \tag{4.6}$$

$$h_{peor}(n) = \left\lceil \log_{d+1} \frac{n+1}{2} \right\rceil = \left\lceil \log_{\lceil p/2 \rceil} \frac{n+1}{2} \right\rceil \tag{4.7}$$

Como se puede ver, en todos los casos la altura está en un $O(\log n)$ y lo mismo ocurrirá con el tiempo de las operaciones sobre el árbol B, si contamos sólo la entrada/salida de disco. Pero lo que resulta realmente interesante es la base del logaritmo. No es lo mismo[18] un logaritmo en base 2 que en base 512. En la tabla 4.3 se muestran comparativamente las alturas según el número n de claves, utilizando árboles AVL y B. Se muestran también unos valores de ejemplo, suponiendo que queremos almacenar el conjunto de los DNI de todos los españoles (alrededor de 40 millones) y que p=512.

Estructura	Altura mejor caso	(ejemplo)	Altura peor caso	(ejemplo)
Árbol AVL	$\lceil \log_2(n+1) \rceil - 1$	25	$\lceil \log_{1,62}((n+1)/1{,}89) \rceil$	35
Árbol B	$\lceil \log_p(n+1) \rceil - 1$	2	$\lceil \log_{\lceil p/2 \rceil}((n+1)/2) \rceil$	4

Tabla 4.3: Alturas en el mejor y peor caso de un árbol AVL y un árbol B con n nodos. En el ejemplo, n= 40.000.000 y p= 512.

Realmente, la comparativa debería tener en cuenta que de los veinte o treinta nodos leídos en el AVL, puede que muchos de ellos estén en el mismo bloque de disco. El

[18] Aunque sí que son iguales en cuanto a órdenes de complejidad. Es decir $O(\log_2 x) = O(\log_{512} x)$.

número de E/S sería algo menor. Sin embargo, el árbol B nos garantiza que la búsqueda no requerirá nunca más de cinco E/S (una más que la altura).

Por otro lado, si suponemos que tanto el árbol B como el AVL están en memoria, deberíamos multiplicar el número de nodos recorridos por el tiempo en cada nodo. El tiempo por nodo en el AVL es constante, mientras que en el árbol B depende del número de entradas del nodo. Haciendo una búsqueda binaria en cada nodo, tendríamos $\lceil \log_2(q+1) \rceil$ comparaciones en el peor caso, si el nodo tiene q entradas. Si consideramos el mejor caso de altura, $q = p - 1$ en todos los nodos y el número de nodos tratados sería $\lceil \log_p(n+1) \rceil$. Multiplicando los dos términos tenemos:

$\lceil \log_2 p \rceil \lceil \log_p(n+1) \rceil \approx \lceil \log_2(n+1) \rceil$

En conclusión, en el árbol B el número de comparaciones es un logaritmo en base 2, ¡exactamente el mismo logaritmo que con árboles AVL[19]! La ventaja de los árboles B se encuentra, por lo tanto, cuando consideramos las E/S de disco.

Utilización de memoria

La implementación de árboles B usando un bloque de disco –de tamaño fijo– por cada nodo, implica una reserva de memoria que después puede ser utilizada o no. Esta situación es comparable a lo que puede ocurrir con tablas de dispersión, donde se reservan muchas cubetas que después pueden usarse o no. Algo parecido ocurre en los árboles AVL. Por cada clave existen dos punteros, pero todos los punteros de los hijos tendrán siempre valor nulo. En un árbol B, con un p suficientemente grande, la proporción está en torno a un puntero por clave. Por contra, en el peor caso los nodos estarán medio vacíos.

Consideremos que una entrada (clave y valor) ocupa k_1 bytes y un puntero k_2 bytes. Un árbol AVL necesitará siempre $n(k_1 + 2k_2)$ bytes, para almacenar n entradas. Por otro lado, de forma aproximada, el árbol B suponiendo que sólo se ocupan los nodos hasta la mitad, ocuparía $2n(k_1 + k_2)$. Si los nodos se llenan con más proporción, el tamaño tendería a $n(k_1 + k_2)$. En definitiva, según el porcentaje de llenado de los nodos la utilización de memoria será más o menos eficiente.

Ejemplo 4.4 En un árbol B de orden p=4 inicialmente vacío almacenamos números enteros. Insertamos los siguientes elementos: 37, 14, 60, 9, 22, 51, 10, 5, 55, 70, 1. La estructura del árbol después de cada operación se muestra en la figura 4.27.

Ejercicios resueltos

Ejercicio 4.1 En cierta aplicación, utilizamos un árbol trie para representar palabras en dos o más idiomas, por ejemplo, español e inglés. Queremos añadir a cada palabra una definición de su significado y la traducción al otro idioma. Describir la estructura de datos, mostrando las definiciones de los tipos necesarios. Hay que tener en cuenta que algunas palabras pueden tener significado en los dos idiomas, por ejemplo, can, conductor, mete, sin.

[19]Se puede comprobar que tomando el peor caso también obtenemos un logaritmo en base 2.