5.6 Árboles Binarios de Búsqueda Balaceado (AVL)

En esta sección se va a describir el comportamiento de los árboles binarios de búsqueda balanceado (AVL, por el nombre de sus autores Adelson, Velskii y Landis).

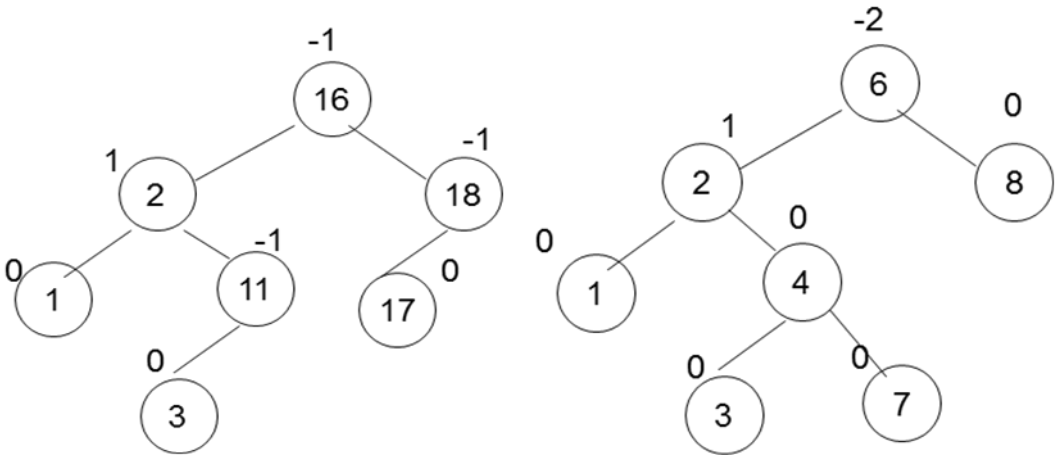
Un árbol AVL es un Árbol binario de búsqueda(ABB) al que se le añade una condición de equilibrio. Esta condición es que para todo nodo la altura de sus subárboles izquierdo y derecho pueden diferir a lo sumo en 1.

Los árboles binarios de búsqueda que hemos analizado en la sección anterior, adolecen del problema de que en el peor de los casos pueden tender parcialmente hacia un árbol similar a una lista, de manera que la búsqueda de un elemento cualquiera puede ser de un orden superior a O(log n), y tender a O(n). Este problema se resuelve con los árboles AVL debido a que estos árboles aseguran una serie de propiedades que permiten que la búsqueda de cualquier elemento quede acotada por una complejidad de orden O(log n).. El orden de las operaciones de inserción y eliminación sigue siendo O(log n), debido a la forma que se mantienen balanceados y que se verá a continuación. Por tanto, las aplicaciones de estos árboles son las mismas que las de un ABB, pero con la consideración de que las búsquedas son más eficientes.

En general podemos decir lo siguiente a cerca de un árbol balanceado de búsqueda:

- Un AVL es un ABB
- La diferencia entre las alturas de los subárboles derecho e izquierdo no debe excederse en más de 1 para cada nodo del árbol. Es decir, por cada nodo se puede decir que, la diferencia de altura es 1 si su subárbol derecho es más alto, -1 si su subárbol izquierdo es más alto y 0 si las alturas son las mismas.
- La inserción y eliminación en AVLs es la misma que en los ABBs

A continuación se describen dos ejemplos de árboles binarios de búsqueda en donde el árbol de la izquierda se encuentra balanceado y el de la derecha no, debido a que la diferencia de los subárboles izquierdo y derecho para el nodo 6 es mayor a 1.

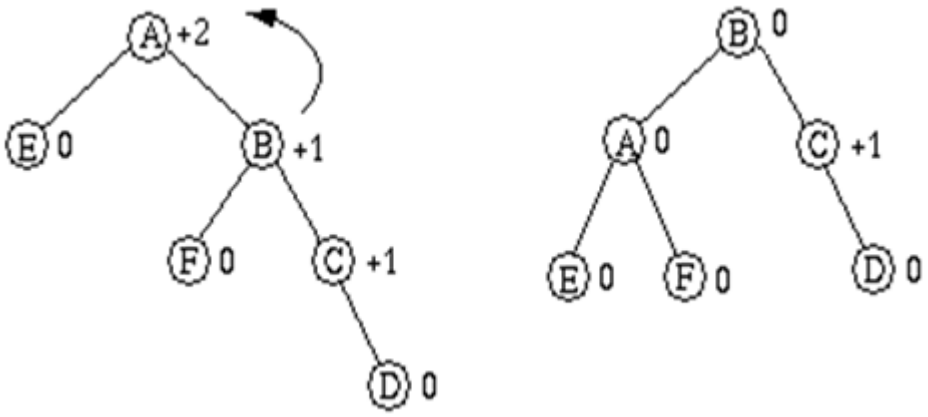


5.6.1 Rotaciones

Para mantener un árbol AVL balanceado se debe tener en cuenta que luego de cada inserción o eliminación no se pierda la propiedad que *la diferencia entre las alturas de los subárboles derecho e izquierdo no debe excederse en más de 1 para cada nodo del árbol*, por lo cual se debe verificar esta condición y corregirla en caso que suceda. Para llevar a cabo la corrección, en la diferencia de las alturas de los subarboles, existe un concepto de rotaciones en donde por medio de a lo sumo dos rotaciones entre los nodos se puede volver a balancear el árbol. A continuación se describen los casos que se pueden presentar de rotaciones, y la forma de realizarlas para devolver al árbol la condición de balanceo.

- Caso 1: Rotación simple izquierda RSI
    - Si esta desequilibrado a la izquierda y su hijo derecho tiene el mismo signo (+) hacemos rotación sencilla izquierda.

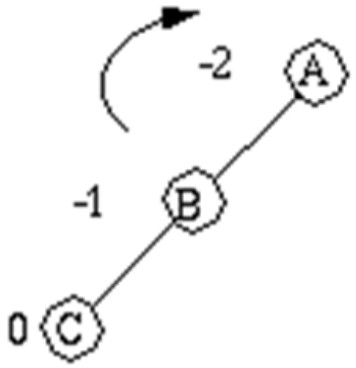
A continuación se describen los casos que se pueden presentar y la forma de rotación para resolver el problema del balanceo.

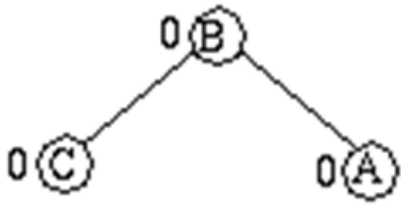


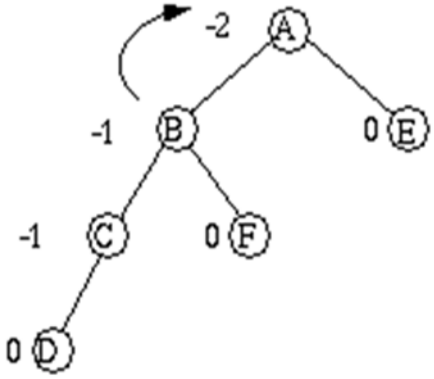
- Caso 2: Rotación simple derecha RSD

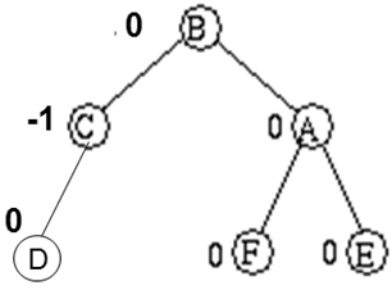
    - Si esta desequilibrado a la derecha y su hijo izquierdo tiene el mismo signo (-) hacemos rotación sencilla derecha.

  A continuación se describen los casos que se pueden presentar y la forma de rotación para resolver el problema del balanceo.





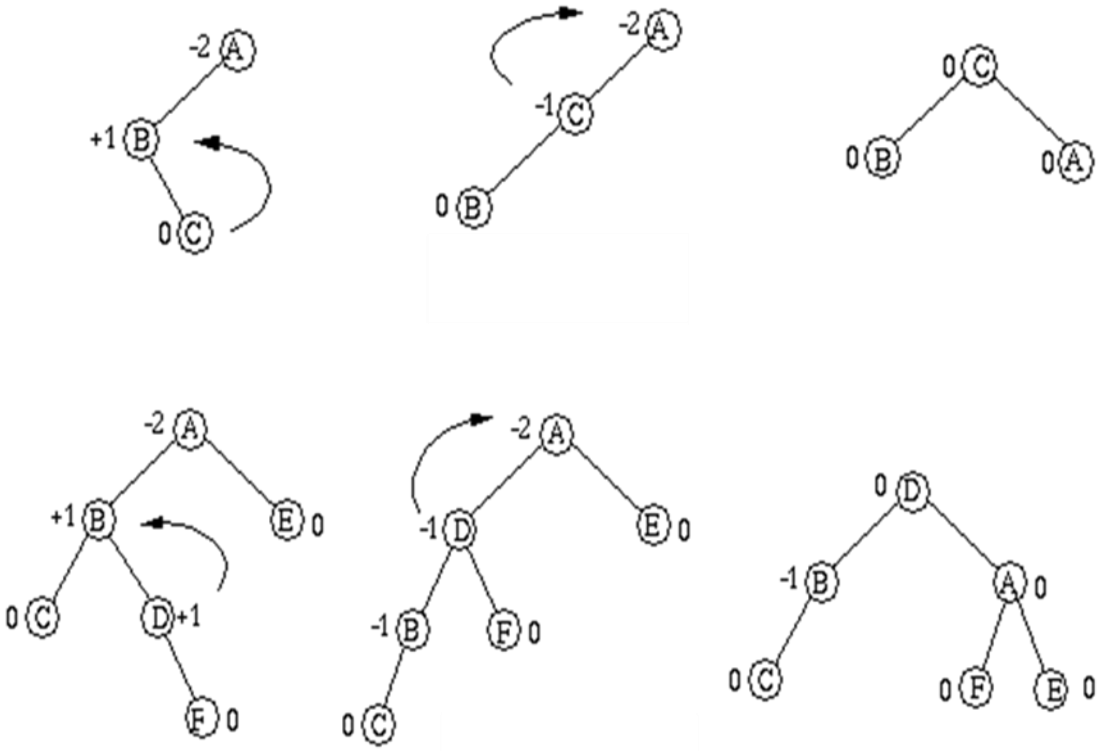




- Caso 3: Rotación doble izquierda RDI

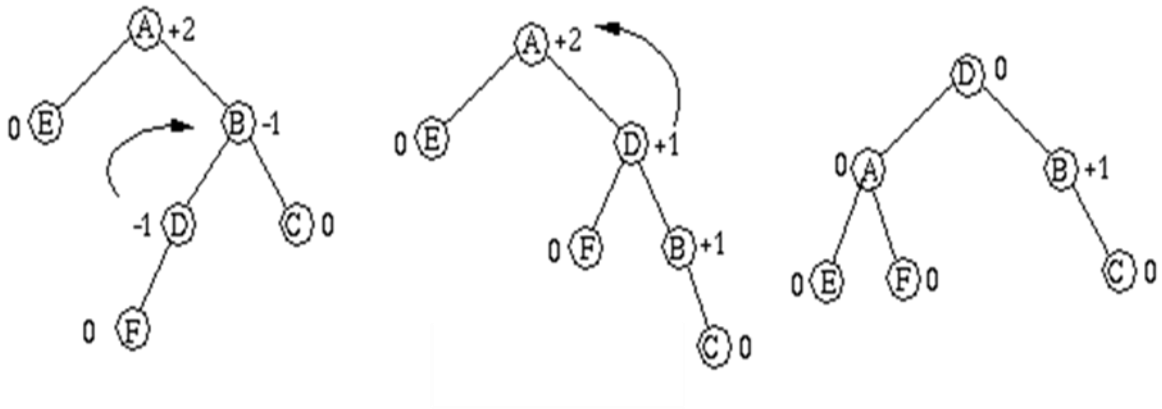
    - Si está desequilibrado a la izquierda (FE < –1), y su hijo derecho tiene distinto signo (+) hacemos rotación doble izquierda-derecha.

A continuación se describen los casos que se pueden presentar y la forma de rotación para resolver el problema del balanceo.



- Caso 4: Rotación doble derecha RDD
    - Si esta desequilibrado a la derecha y su hijo izquierdo tiene distinto signo (–) hacemos rotación doble derecha-izquierda.

A continuación se describen los casos que se pueden presentar y la forma de rotación para resolver el problema del balanceo.


A +2
0 E
B -1
-1 D
C 0
0 F
A +2
0 E
D +1
0 F
B +1
C 0
D 0
0 A
B +1
0 E
F 0
C 0

5.7 Árboles B[1]

Árboles de Búsqueda

Un árbol de búsqueda es un tipo especial de árbol que sirve para guiar la búsqueda de un valor. Un árbol de búsqueda de orden **q=2*t+1** es un árbol tal que cada nodo contiene, cuando más, **2*t** valores de búsqueda y **2*t+1** referencias a nodos en el orden <P_1, k_1, P_2, k_2, …, P_{q-1}, K_{q-1}, P_q> , donde q <= 2*t+1, cada P_i es una referencia a un nodo hijo (puede ser vacío) y cada K_i es un valor de búsqueda proveniente de algún conjunto ordenado de valores. Se supone que todos los valores de búsqueda son únicos. Un árbol de búsqueda debe cumplir, en todo momento, las siguientes restricciones:

1. Dentro de cada nodo $K_1 < K_2 < \ldots < K_{q-1}$
2. Para todos los valores de X, del subárbol al cual refiere P_i , tenemos que $K_{i-1} < X < K_i$, para $1 < i < q$; $X < K_i$, para i = 1 y $X > K_{i-1}$ para i = q.

La figura 1 muestra un nodo de un árbol de búsqueda. En ella se pueden apreciar de manera más clara, las condiciones que impone la restricción 2.

Al buscar, insertar o eliminar un valor X siempre se sigue la referencia P_i apropiada, de acuerdo con las condiciones de la segunda restricción.

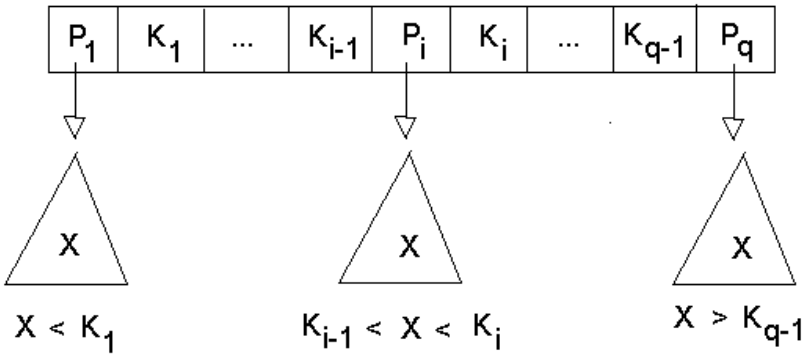


Fig 1 Nodo de un árbol de búsqueda

.

Para insertar valores de búsqueda en el árbol y eliminarlos, sin violar las restricciones anteriores, se utilizan algoritmos que no garantizan que el árbol de búsqueda esté equilibrado (que todas las hojas estén al mismo nivel).

---

[1] **Aclaración de autoría:** este texto fue adaptado de una monografía publicada originalmente por Lic. Joel Arencibia Ramírez

Es importante mantener equilibrados los árboles de búsqueda porque esto garantiza que no habrá nodos en niveles muy profundos que requieran muchas comparaciones durante una búsqueda. Además, las eliminaciones de valores pueden hacer que queden nodos casi vacíos, con lo que hay un desperdicio de espacio importante que también provoca un aumento en el número de niveles y por ende un mayor costo potencial en la búsqueda de un elemento.

Árbol B (En busca del equilibrio en el árbol y un mayor aprovechamiento del espacio)

El árbol B es un árbol de búsqueda, con algunas restricciones adicionales, que resuelve hasta cierto punto los dos problemas anteriores. Estas restricciones adicionales garantizan que el árbol siempre estará equilibrado y que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo. Los algoritmos para insertar y eliminar se hacen más complejos para poder mantener estas restricciones. No obstante, la mayor parte de las inserciones y eliminaciones son procesos simples, se complican sólo en circunstancias especiales.

De manera más formal, un Árbol B de orden q, utilizado como estructura de acceso según un valor, cumple que:

1. Cada nodo interno del árbol B tiene la forma general $<P_1, <k_1, Pr_1>, P_2, <k_2, Pr_2>, \ldots, <kq-1, Pr_{q-1}>, P_q>$ (Ver Fig. 2)

   donde q <= 2*t+1. Cada P_i es un *referencia* a otro árbol. Cada P_{ri} es una *referencia de datos*: una referencia a información adicional asociada al valor de búsqueda K_i (similar a lo visto en el TPO)

2. Dentro de cada nodo $K_1 < K_2 < \ldots < K_{q-1}$
3. Para todos los valores de X, del subárbol al que referencia P_i , tenemos que $K_{i-1} < X < K_i$, para $1 < i < q$; $X < K_i$, para $i = 1$ y $X > K_{i-1}$ para $i = q$.
4. Cada nodo tiene, cuando más, 2*t+1 referencias de árbol.
5. Cada nodo, excepto la raíz y los nodos hoja, tienen por lo menos t+1 referencias de árbol. El nodo raíz tiene, como mínimo, dos referencias a nodos del árbol, a menos que sea el único nodo del árbol.
6. Un nodo con q referencias a árbol, q <= 2*t+1, tiene q-1 valores de búsqueda y por tanto, tiene q-1 referencias de datos.
7. Todos los nodos hojas están en el mismo nivel. Los nodos hoja tienen la misma estructura que los internos, excepto que todos sus referencias de árbol, P_i son nulas (árboles vacíos).

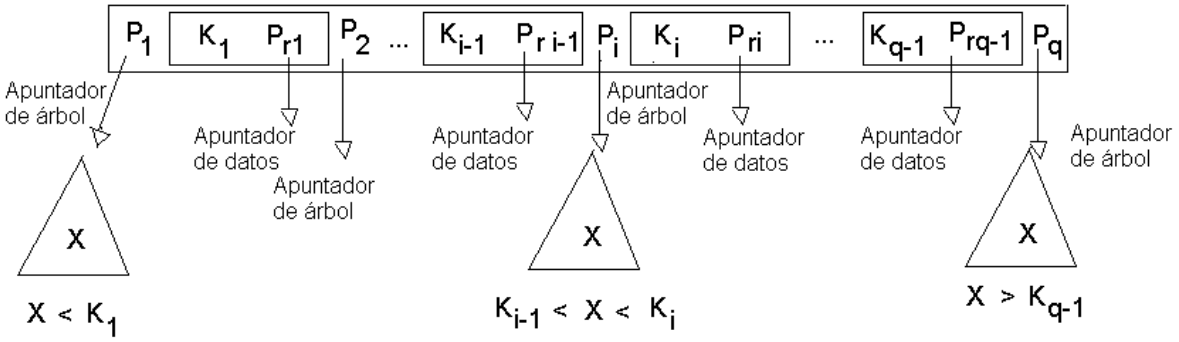


Fig 2. Estructura del árbol B. Nodo de árbol B con q-1 valores de búsqueda

Nota: Por una cuestión de simplicidad de los gráficos, en adelante sólo se representarán las referencias a árboles.

A continuación se muestran dos árboles. Uno de ellos es un árbol B, en cambio, el otro no. El árbol de la figura 3.1 es un árbol B “correcto”, pues cumple con todas las reglas en cuanto a su estructura y orden. En cambio, el de la figura 3.2 no es un árbol B, ya que a pesar de mantener el orden, hay un nodo (que no es la raíz) que tiene menos de t elementos (en este caso menos de 2 elementos). Este nodo es el que contiene al elemento 10.

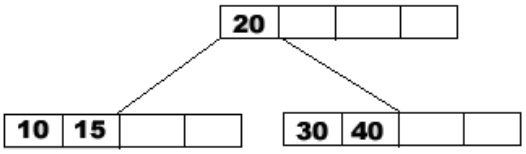


Fig 3.1 Árbol B de orden 5

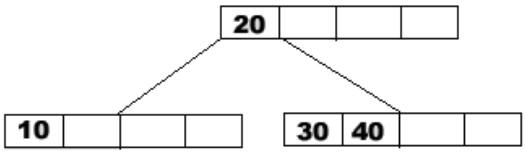
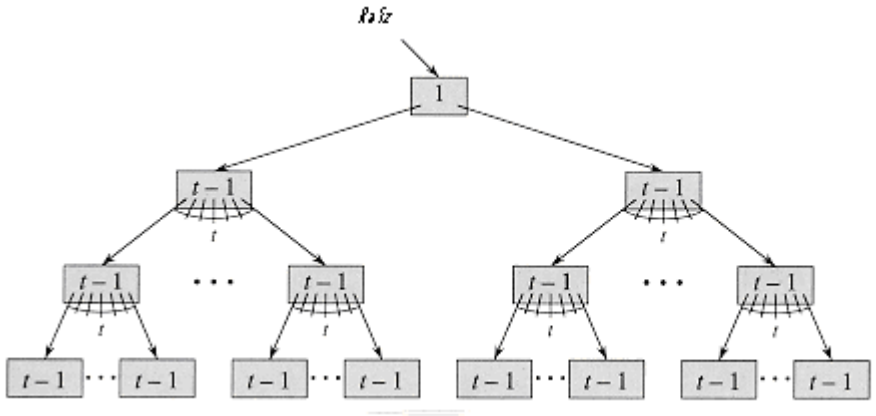


Fig. 3.2 No se corresponde con un árbol B

El número de acceso a disco en un árbol B es proporcional a la altura del árbol, el caso peor para la altura es cuando la raíz tiene un solo valor y el resto de los nodos tiene el mínimo permitido t.

La siguiente figura nos muestra la cantidad de nodos por cada nivel del árbol.



Generalizando, para un árbol de altura h, la cantidad de nodos que tiene el último nivel sería $2t^{h-1}$.

La cantidad de nodos n, que tiene un árbol B cumple con lo siguiente:

$$
\begin{aligned}
n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\
&= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\
&= 2t^h - 1 ,
\end{aligned}
$$

Obtenemos que:

$$h \leq \log_t \frac{n+1}{2} .$$

entonces el crecimiento del árbol es $O(\log_t n)$.

Operaciones básicas en árboles B

Las operaciones básicas que se pueden realizar en un árbol B son básicamente tres:

1. Buscar un valor
2. Insertar un valor
3. Eliminar un valor

Búsqueda

La búsqueda de un valor Y se realiza de manera análoga a la búsqueda en un árbol binario de búsqueda. Se comienza buscando por el nodo raíz y se compara el valor Y con las llaves K_i que se encuentran en ese nodo. Si Y es igual a algún K_i, termina la búsqueda satisfactoriamente, si no, se debe seguir la referencia adecuada hacia un nodo hijo en el que se continuará la búsqueda. La referencia seleccionada será P_1, si $Y < K_1$, P_q si $Y > K_{q-1}$ y P_i, si $K_{i-1} < X < K_i$, para $1 < i < q$. En caso que el P_i correspondiente sea nulo, la búsqueda termina insatisfactoriamente.

En la búsqueda en un árbol B el número de acceso a nodos es $\Theta(h) = \Theta(\log_t n)$ donde h es la altura del árbol, n la cantidad de valores, $n<2*t+1$. La búsqueda lineal del valor en un nodo es O(t) por lo tanto el tiempo total es $O(t*h)=O(t*\log_t n)$.

Inserción

Para realizar la inserción, lo primero que debe hacerse es un proceso de búsqueda, para localizar el nodo en el que se va a insertar. Si este proceso de búsqueda da por resultado que el elemento ya existe, no se realizará ninguna operación, pues el árbol B no permite elementos repetidos.

La búsqueda de un elemento inexistente y que, por tanto, pueda ser insertado en el árbol, terminará en el momento en que, tratando de buscar una referencia P_i que referencie al nodo donde debería estar el elemento, encontremos árbol vacío, siendo insertado el nuevo elemento en el nodo actual. Cuando se inserta en un nodo que no está lleno, sólo debemos preocuparnos por el lugar que debe ocupar la nueva llave en ese nodo y hacer un corrimiento de las restantes.

En problema surge cuando ya se han insertado varias llaves en un mismo nodo hasta que este queda completo. Una vez lleno un nodo con p-1 valores de búsqueda, si intentamos insertar una entrada más en él, debemos seguir el siguiente procedimiento:

1. Insertar el valor como si realmente tuviese sitio libre.
2. Extraer el valor que ocupa la posición del medio e insertarlo en el nodo padre. Si no hubiese nodo padre se crearía un nuevo nodo con ese valor.
3. Dividir equitativamente el nodo original en dos nuevos nodos Estos nodos serán los hijos derechos e izquierdos del nodo que promocionó al nivel superior.
4. Si el nodo al que promociona el valor mediano también se encuentra completo, se repetiría la misma operación de división y promoción.

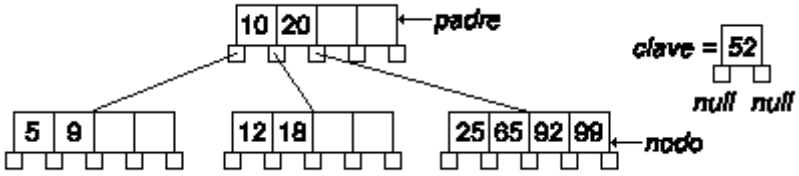
La división puede propagarse hasta llegar al nodo raíz. En caso que el nodo raíz deba dividirse también, el elemento que es promocionado para ser insertado en el padre, será insertado en un

nuevo nodo que se debe crear, que pasará a ser la nueva raíz del árbol, creando un nuevo nivel cada vez que se divida la raíz.

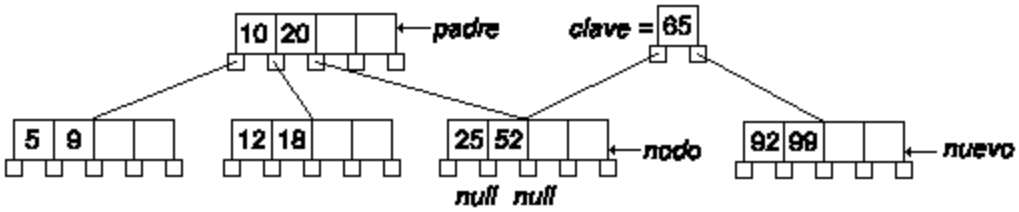
Es importante hacer notar la curiosa manera de crecer en altura que tienen los árboles B, que lo hacen hacia la raíz, a diferencia de los comúnmente utilizados, cuyas inserciones se realizan en forma de hojas y por tanto crecen en sentido inverso al árbol B. Esto es muy importante, porque garantiza que en el árbol B, todas las hojas estén en el mismo nivel y por tanto, esté balanceado.

Veamos algunos ejemplos:

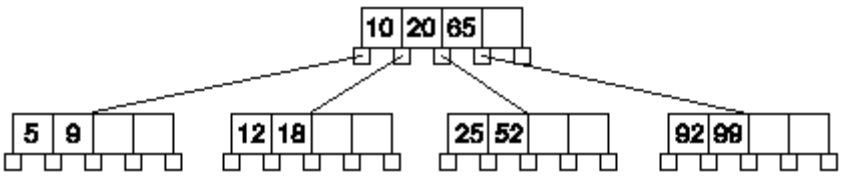
Ej. 1 Supongamos que queremos insertar el valor 52 en el árbol siguiente.



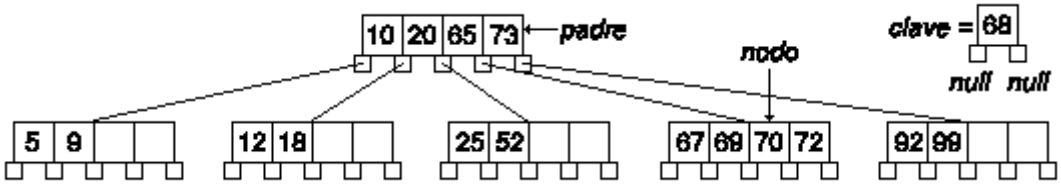
No hay espacio para almacenar el valor en nodo, por lo tanto, este nodo debe dividirse, promocionando el valor que queda en el medio, en este caso, el 65:



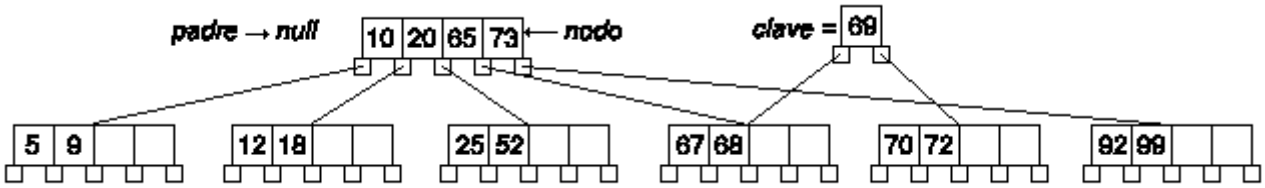
Ahora tenemos que promocionar el valor intermedio, insertándolo en el nodo padre. Como el valor promocionado, pudo insertarse sin dificultad en el nodo padre, el proceso termina.



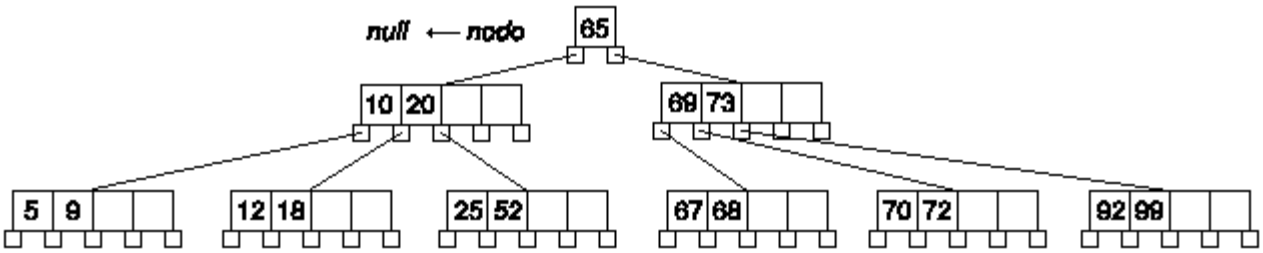
Ej. 2 Veamos otro ejemplo que implicará que aumente la altura del árbol. Supongamos que tenemos un árbol con esta estructura, y queremos insertar el valor 68:



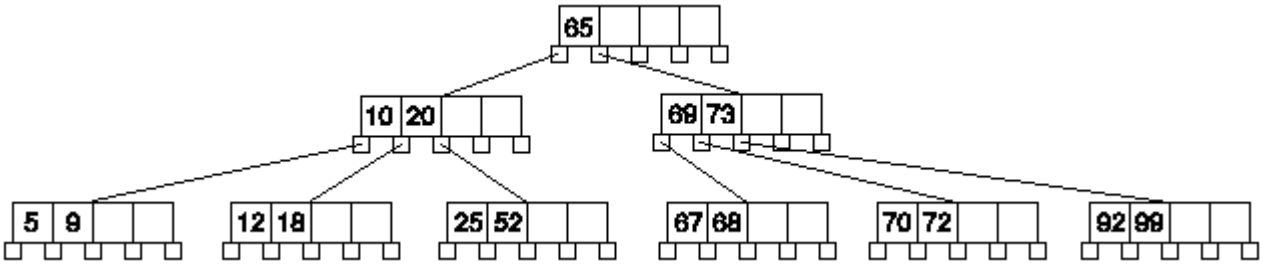
En primer lugar, dividimos *nodo* en dos, repartimos los valores y promocionamos el valor intermedio, el 69:



Ahora estamos de nuevo en la misma situación, cuando tratamos de insertar el 69 en el nodo padre, pues este no cabe en dicho nodo, luego, el anterior nodo pasa a ser el nodo padre, y el padre de éste es vacío. Ahora procedemos igual, dividimos *nodo* en dos, separamos los valores, y promocionamos el valor intermedio:



Y otra vez repetimos el proceso, al tratar de insertar el 65 en el nodo padre del nodo que se dividió, vemos que este no existe, porque el nodo que se acaba de dividir era la raíz. En este caso, se crea un nuevo nodo en el que se inserta el valor promovido y se actualizan las referencias. Note la variación en la altura del árbol y su crecimiento hacia la raíz.



En la inserción en un árbol B el número de acceso a nodos es $\Theta(h) = \Theta(\log_t n)$ donde h es la altura del árbol, n la cantidad de llaves, n<2*t+1. La búsqueda lineal del valor en un nodo es O(t) por lo tanto el tiempo total es $O(t*h)=O(t*\log_t n)$.

Eliminación

El proceso de eliminación puede llegar a ser algo más complejo que la inserción. La eliminación siempre debe realizarse en una hoja. Si después de realizarla búsqueda, el nodo a borrar no estuviese en una hoja, de la misma manera que se procede en un árbol binario de búsqueda, el nodo a borrar se sustituiría por su antecesor o sucesor, que sí debe estar en una hoja.

Al hacer la sustitución, caemos en el caso de la eliminación de un valor de un nodo hoja, la que ya fue replicada.

El caso más sencillo se produce cuando al eliminar un valor de un nodo hoja, bien porque sea el nodo a eliminar, bien porque sea un elemento que sustituyó al nodo a eliminar, el tamaño del nodo sigue siendo al menos t. De esta forma, la estructura del árbol se sigue cumpliendo y no hay que realizar ninguna reestructuración.

El problema surge cuando al eliminar un valor de la hoja, el nodo se queda con menos del mínimo número de valores permitidos. Si la eliminación de un valor hace que un nodo quede ocupado hasta menos de la mitad, este le “pedirá prestado un elemento” a uno de sus hermanos, o el izquierdo o el derecho. Si el hermano está en condiciones de ceder un elemento, o sea, si al perder dicho elemento no hace que quede ocupado a menos de la mitad, cederá el elemento más a la izquierda o más a la derecha, en dependencia de su ubicación respecto al hermano.

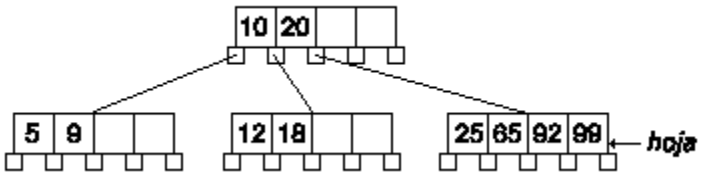
Esta “donación”, debe realizarse con participación del nodo padre. El elemento donado, debe intercambiarse con el valor K_i correspondiente en el nodo padre y es este valor K_i quien iría a suplir la falta en el hermano afectado. En este caso, pudiera adoptarse la variante de chequear primero si el hermano derecho tiene la posibilidad de donar un valor y en caso negativo, pasar a chequear si el hermano izquierdo puede, pero esto provocaría un doble costo y por tanto, es preferible chequear solamente a uno de los dos hermanos.

Aún resta un caso más. Este caso es idéntico al anterior, pero ahora el nodo vecino tiene el número mínimo de nodos y por tanto, no existe la posibilidad de suplir la falta de llave tomando un valor del hermano En esta situación se produce el efecto inverso a la división, los dos nodos se agrupan en uno solo, formando un nuevo nodo en la que todavía existe capacidad para otro valor. A este nodo se le añade el valor K_i que estaba situado en el nodo padre, entre ambos nodos. Si al fundirse los dos nodos y tomar el valor K_i de su padre, este se queda con una cantidad de llaves menor que t, el proceso se repite, propagándose hasta la raíz.

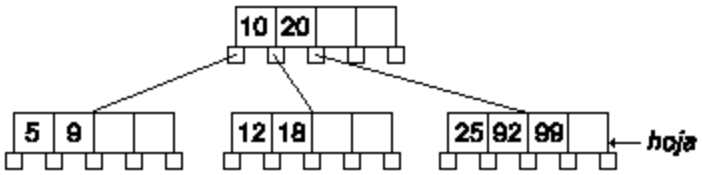
En caso de ser la raíz el nodo al que se le quitará el K_i y esta sólo tenía un valor, queda automáticamente eliminada, reduciéndose en 1 la altura del árbol. Note que, del mismo modo que el árbol B crece hacia la raíz, este tipo de árbol se acorta también desde la raíz, garantizándose así que todas las hojas, estén al mismo nivel.

Veamos algunos ejemplos de eliminación:

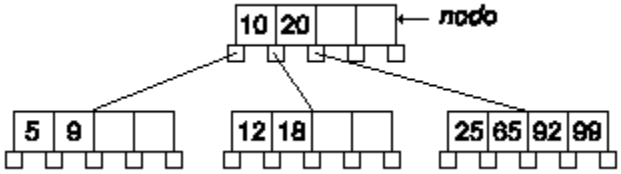
Ej. 1 Eliminemos la clave 65. Es uno de los casos más sencillos, el valor está en un nodo hoja, y el número de valores del nodo después de eliminarlo es mayor o igual al mínimo.



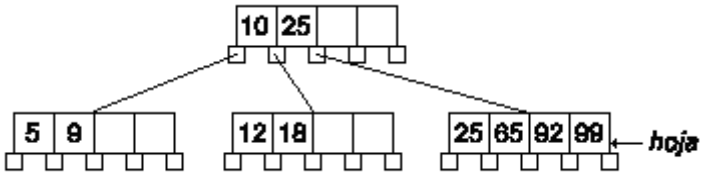
Tan sólo tendremos que eliminar el valor:



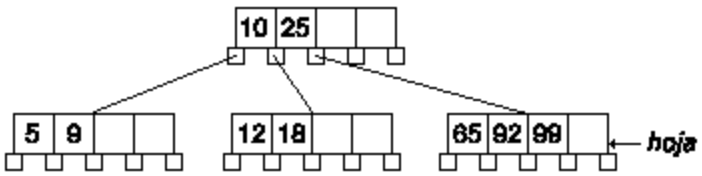
Ej. 2 Eliminaremos ahora el valor 20.



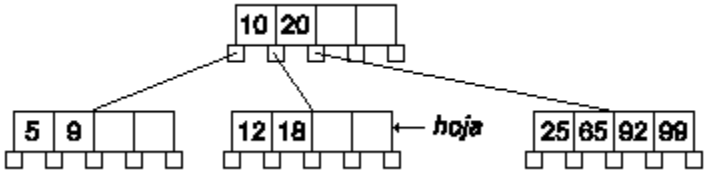
Según el algoritmo debemos sustituir el valor por el siguiente valor, es decir, el 25.



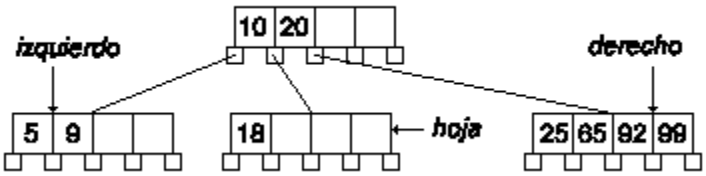
Ahora estamos en el mismo caso que antes, sólo hay que borrar el valor 25 del nodo hoja:



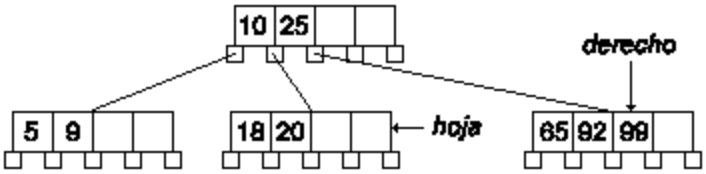
Ej. 3 Pasemos a eliminar ahora el elemento 12 del árbol:



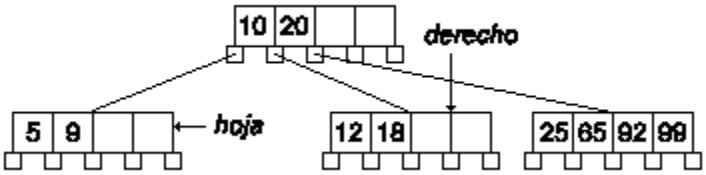
La primera parte es fácil, bastará eliminar la clave, pero el resultado es que sólo queda una clave en el nodo, y debe haber al menos dos claves en un nodo de cuatro.



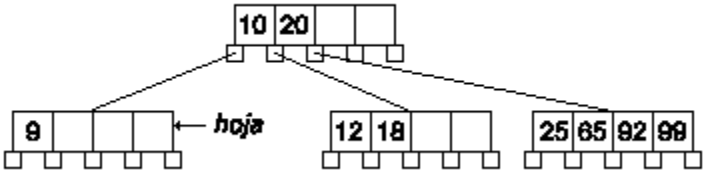
Según el algoritmo, buscaremos en el hermano derecho. Si el número de valores en el nodo derecho es mayor que el mínimo, pasaremos un valor del nodo derecho al padre y de éste al nodo hoja, en este caso, el 25 pasa al nodo padre y el 20 pasa al nodo que tenía el déficit.



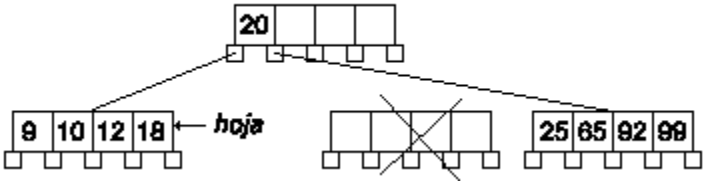
Ej. 4 Vamos a ver el caso en que el nodo derecho no tiene valores suficientes para transferir uno al nodo hoja. Eliminaremos el valor 5:



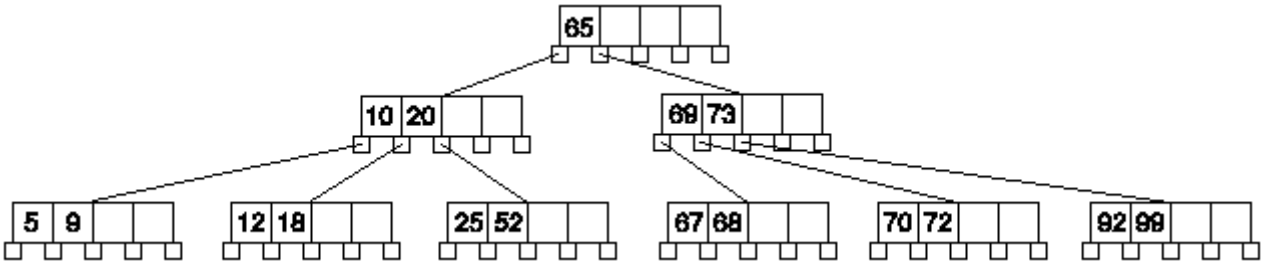
La primera parte es igual que el caso anterior, eliminamos el valor 5. Pero ahora el nodo *derecho* tiene el número mínimo de valores.



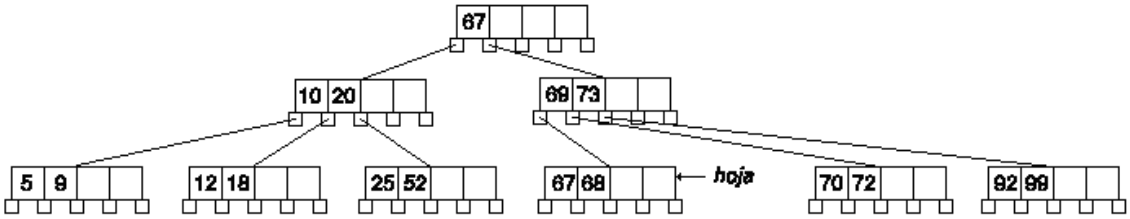
Según el algoritmo, debemos fundir en el nodo *hoja*, los valores del nodo *hoja*, el valor del *padre* y los del nodo *derecho*. Y después eliminar el nodo derecho.



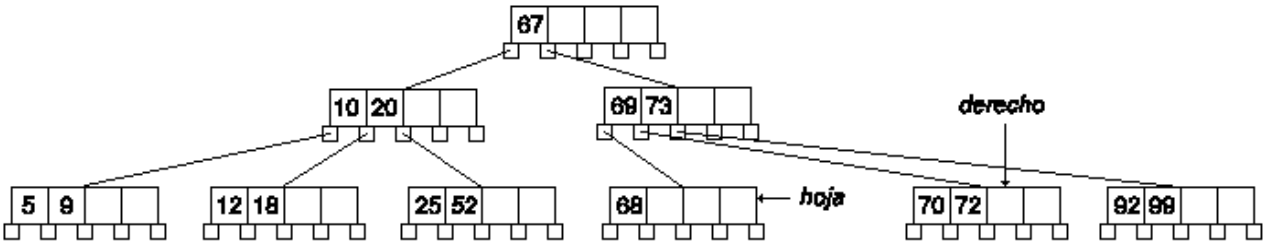
La cosa no termina ahí, ahora debemos comprobar el nodo padre, ya que también ha podido quedar con menos claves que el mínimo exigido. En este caso sucede eso, pero es un caso especial, ya que ese nodo es raíz y puede contener menos claves que número mínimo.
Ej. 5 Veamos cómo sería el proceso en un caso general, que implica la reducción de altura del árbol.
Borraremos la clave 65 de este árbol:



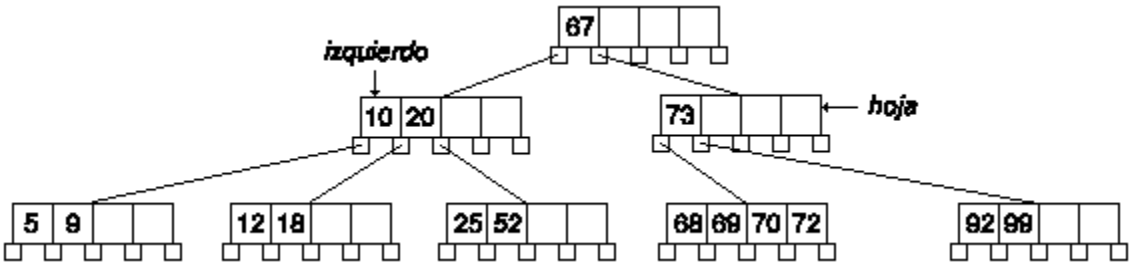
El primer paso es sustituir el valor 65 por el siguiente, el 67:



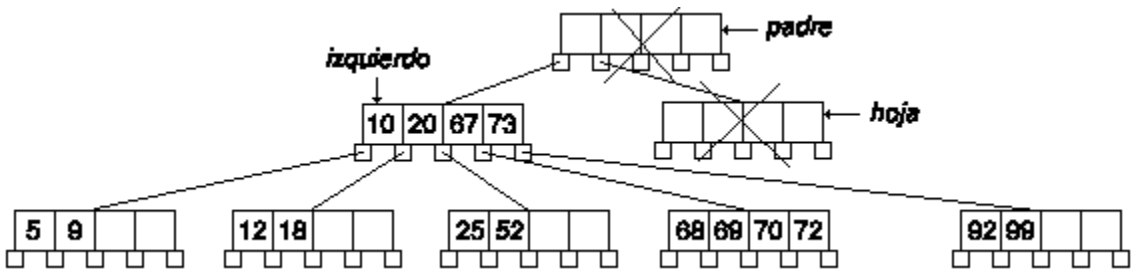
A continuación eliminamos el valor 67 del nodo *hoja*, y comprobamos que el nodo tiene menos valores del mínimo. Buscamos el nodo *derecho*



El nodo *derecho* tiene justo el número mínimo de claves, por lo tanto tenemos que fusionar *hoja* con *derecho* y con un valor de *padre*. Y eliminamos el nodo *derecho*.



Ahora el nodo *hoja* es el *padre* del anterior nodo hoja. De nuevo comprobamos que tiene menos claves que el mínimo, así que buscamos el nodo *derecho*, como no existe, buscamos el *izquierdo*. Y de nuevo comprobamos que el *izquierdo* tiene justo el número mínimo de valores.
Ahora fusionamos en el nodo *izquierdo* con el nodo *hoja* y con una clave del nodo *padre*, y eliminamos el nodo *hoja*, como además el nodo *padre* es la raíz y ha quedado vacía, lo eliminaremos y el nodo raíz será el *izquierdo*:



En la eliminación en un árbol B el número de acceso a nodo es $\Theta(h) = \Theta(\log_t n)$ donde h es la altura del árbol, n la cantidad de llaves, n<2*t+1. La búsqueda lineal del valor en un nodo es O(t) por lo tanto el tiempo total es $O(t*h)=O(t*\log_t n)$.