TDA Árboles

Definición

Un árbol es una estructura no lineal en la que cada nodo puede apuntar a uno o varios nodos. En teoría de grafos se define como un GRAFO ACÍCLICO, CONEXO y NO DIRIGIDO en el que existe exactamente un camino entre todo par de nodos.

También se suele dar una definición recursiva como:

- Una estructura vacía
- Un elemento o clave de información (nodo) más un número finito de estructuras tipo árbol, disjuntos, llamados subárboles.

Formas de representación:

Hay varias formas de representar un árbol:

- Conjuntos incluidos
- Paréntesis incluidos
- Jerarquización de márgenes
- Grafos

Estudiaremos solo la representación por grafos

Conceptos:

Definiremos varios conceptos:

Nodo	Un elemento o clave de la información
Rama	Arista entre dos nodos
Antecesor o Precedente o Nodo Padre	Un nodo x es antecesor de un nodo y si por alguna de las ramas de x se puede llegar a y
Sucesor o Descendiente o Nodo Hijo	Un nodo x es sucesor de un nodo y si por alguna de las ramas de y se puede llegar a x
Terminal u Hoja	Un elemento o nodo que no tiene descendientes (grado 0)
Raíz	El único nodo que no tiene antecesores propios
Nivel	Número de ramas que hay que recorrer para llegar de la raíz a un nodo. En otras palabras, se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero y el de sus hijos uno y así sucesivamente.
Nodo interior	Un nodo que no es un terminal, es decir, tiene al menos un descendiente.

Grado u Orden	Número de descendientes de un nodo interior. Es el número potencial de hijos que puede tener cada elemento de árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos es de *orden dos*, si puede apuntar a tres será de *orden tres*, etc.
Grado del árbol	El máximo de los grados de los nodos, en otras palabras, el número de hijos que tiene el elemento con más hijos dentro del árbol.
Longitud de camino de x	Número de arcos o aristas que deben ser recorridos para llegar a un nodo x partiendo de la raíz. La raíz tiene longitud 1. (#nodos del camino – 1)
Longitud de camino interno o longitud del árbol	$\sum$ longitudes de todos sus nodos de camino La media sería $C_i = (1/n) \sum n_i * i$ n_i = # nodos en el nivel i
Nodo especial	Extensión ficticia de nodos, suponiendo mismo grado
Longitud de camino externo	$\sum$ longitudes de todos sus nodos especiales
Altura de un nodo	La longitud del camino más largo de ese nodo a una hoja
Altura del árbol	Es la altura de la raíz. Es el nivel más alto del árbol. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas.
Anchura	Es el mayor valor del número de nodos que hay en un nivel
Profundidad de un nodo	Es la longitud del camino único desde la raíz a ese nodo.
Árbol completo	Un árbol en el que en cada nodo o bien todos o ninguno de los hijos existe

Características:

- ✓ Cada nodo sólo puede ser apuntado por otro nodo, es decir, cada nodo sólo tendrá un padre. Esto hace que estos árboles estén fuertemente jerarquizados, y es lo que en realidad les da la apariencia de árboles.
- ✓ Todos los nodos contengan el mismo número de apuntadores, es decir, usaremos la misma estructura para todos los nodos del árbol. Esto hace que la estructura sea más sencilla, y por lo tanto también los programas para trabajar con ellos.
- ✓ No es necesario que todos los nodos hijos de un nodo concreto existan. Es decir, que pueden usarse todos, algunos o ninguno de los apuntadores de cada nodo.
- ✓ Los árboles se parecen al resto de las estructuras que hemos visto: dado un nodo cualquiera de la estructura, podemos considerarlo como una estructura independiente. Es decir, un nodo cualquiera puede ser considerado como la raíz de un árbol completo.
- ✓ Es importante conservar siempre el nodo *raíz* ya que es el nodo a partir del cual se desarrolla el árbol, si perdemos este nodo, perderemos el acceso a todo el árbol. Para hacer más fácil moverse a través del árbol, podríamos añadir un apuntador al nodo padre en cada nodo. De este modo podremos avanzar en dirección a la raíz, y no sólo hacia las hojas.
- ✓ Los árboles de orden dos son bastante especiales, se conocen también como **árboles binarios**.

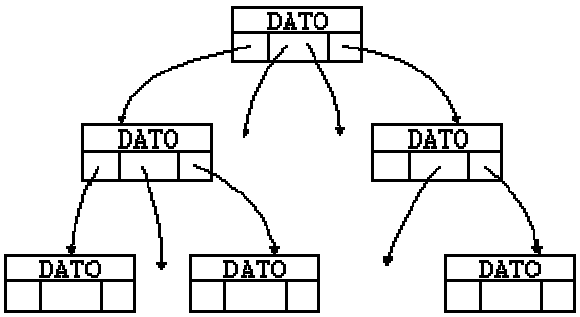
Declaración en C:

```
typedef struct _nodo {
  int dato;
  struct _nodo *rama[ORDEN];
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;
```

Declaramos un tipo *tipoNodo* para declarar nodos, y un tipo *pNodo* que es el tipo para declarar apuntador a un nodo.

Arbol es el tipo para declarar árboles de orden *ORDEN*.



El movimiento a través de árboles, salvo que implementemos apuntadores al nodo padre, será siempre partiendo del nodo raíz hacia un nodo hoja. Cada vez que lleguemos a un nuevo nodo podremos optar por cualquiera de los nodos a los que apunta para avanzar al siguiente nodo.

Orden y recorrido de los nodos:

El modo evidente de moverse a través de las ramas de un árbol es siguiendo los apuntadores, del mismo modo en que nos movíamos a través de las listas.

Esos recorridos dependen en gran medida del tipo y propósito del árbol, pero hay ciertos recorridos que usaremos frecuentemente. Se trata de aquellos recorridos que incluyen todo el árbol.

Hay tres formas de recorrer un árbol completo, y las tres se suelen implementar mediante recursividad. En los tres casos se sigue siempre a partir de cada nodo todas las ramas una por una.

Supongamos que tenemos un árbol de orden tres, y queremos recorrerlo por completo.

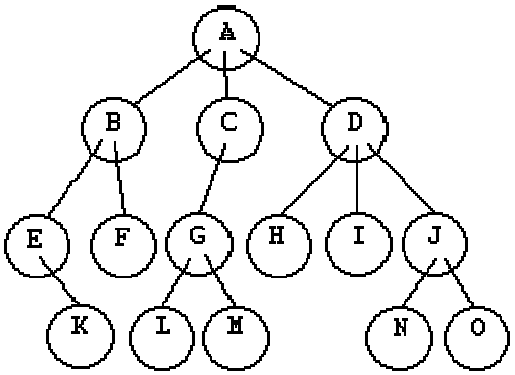
Partiremos del nodo raíz:

```
RecorrerArbol(raiz);
```

La función *RecorrerArbol*, aplicando recursividad, será tan sencilla como invocar de nuevo a la función *RecorrerArbol* para cada una de las ramas:

```
void RecorrerArbol(arbol a)
{
  if(a == NULL) return;
  RecorrerArbol(a->rama[0]);
  RecorrerArbol(a->rama[1]);
  RecorrerArbol(a->rama[2]);
}
```

Lo que diferencia los distintos métodos de recorrer el árbol no es el sistema de hacerlo, sino el momento que elegimos para procesar el valor de cada nodo con relación a los recorridos de cada una de las ramas.



Los tres tipos son:

Pre-orden:

En este tipo de recorrido, el valor del nodo se procesa antes de recorrer las ramas:

```
void PreOrden(arbol a)
{
  if(a == NULL) return;
  Procesar(dato);
  RecorrerArbol(a->rama[0]);
  RecorrerArbol(a->rama[1]);
  RecorrerArbol(a->rama[2]);
}
```

Si seguimos el árbol del ejemplo en pre-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así: A B E K F C G L M D H I J N O

In-orden:

En este tipo de recorrido, el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última. Esto tiene más sentido en el caso de árboles binarios, y también cuando existen ORDEN-1 datos, en cuyo caso procesaremos cada dato entre el recorrido de cada dos ramas (este es el caso de los árboles-b):

```
void InOrden(arbol a)
{
  if(a == NULL) return;
  RecorrerArbol(a->rama[0]);
  Procesar(dato);
  RecorrerArbol(a->rama[1]);
  RecorrerArbol(a->rama[2]);
}
```

Si seguimos el árbol del ejemplo en in-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así: K E B F A L G M C H D I N J O

Post-orden:

En este tipo de recorrido, el valor del nodo se procesa después de recorrer todas las ramas:

```
void PostOrden(arbol a)
{
  if(a == NULL) return;
  RecorrerArbol(a->rama[0]);
  RecorrerArbol(a->rama[1]);
  RecorrerArbol(a->rama[2]);
  Procesar(dato);
}
```

Si seguimos el árbol del ejemplo en post-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así: K E F B L M G C H I N O J D A

Eliminación de nodos:

El proceso general es muy sencillo en este caso, pero con una importante limitación, sólo podemos borrar nodos hoja:

El proceso sería el siguiente:

1. Buscar el nodo padre del que queremos eliminar.
2. Buscar el apuntador del nodo padre que apunta al nodo que queremos borrar.
3. Liberar el nodo.
4. padre->nodo[i] = NULL;.

Cuando el nodo a borrar no sea un nodo hoja, diremos que hacemos una "poda", y en ese caso eliminaremos el árbol cuya raíz es el nodo a borrar. Se trata de un procedimiento recursivo, aplicamos el recorrido PostOrden, y el proceso será borrar el nodo.

El procedimiento es similar al de borrado de un nodo:

1. Buscar el nodo padre del que queremos eliminar.
2. Buscar el apuntador del nodo padre que apunta al nodo que queremos borrar.
3. Podar el árbol cuyo padre es nodo.
4. padre->nodo[i] = NULL;.

En el árbol del ejemplo, para podar la rama 'B', recorreremos el subárbol 'B' en postorden, eliminando cada nodo cuando se procese, de este modo no perdemos los apuntadores a las ramas apuntadas por cada nodo, ya que esas ramas se borrarán antes de eliminar el nodo.

De modo que el orden en que se borrarán los nodos será:

K E F y B

ÁRBOLES BINARIOS

Son árboles de grado 2. Se utilizan frecuentemente para representar conjuntos de datos cuyos elementos se identifican con una clave única.

```
typedef struct _nodo {
  int dato;
  struct _nodo *izq;
  struct _nodo *der;

} tipoNodo;
```

ARBOLES ORDENADOS

Un árbol ordenado, en general, es aquel a partir del cual se puede obtener una secuencia ordenada siguiendo uno de los recorridos posibles del árbol: inorden, preorden o postorden.

En estos árboles es importante que la secuencia se mantenga ordenada aunque se añadan o se eliminen nodos.

Existen varios tipos de árboles ordenados, que veremos en clase. Algunos son:

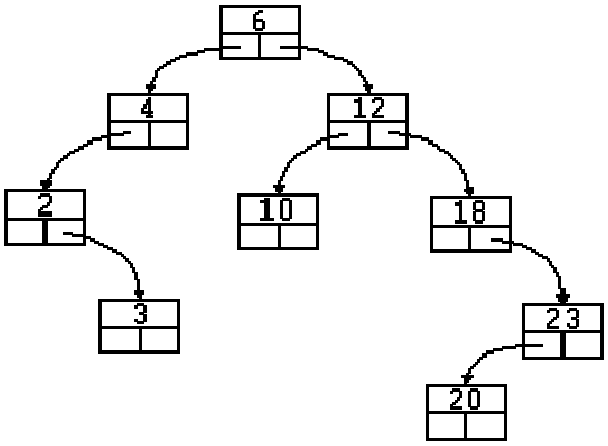
- Árboles binarios de búsqueda (ABB): son árboles de orden 2 que mantienen una secuencia ordenada si se recorren en inorden.
- Árboles AVL: son árboles binarios de búsqueda equilibrados, es decir, para todo nodo, la altura de sus subárboles izquierdo y derecho pueden diferir a lo sumo en 1.
- Árboles perfectamente equilibrados: son árboles binarios de búsqueda en los que el número de nodos de cada rama para cualquier nodo no difieren en más de 1. Son por lo tanto árboles AVL también.
- Árboles 2-3: son árboles de orden 3, que contienen dos claves en cada nodo y que están también equilibrados. También generan secuencias ordenadas al recorrerlos en inorden.

- Árboles-B: caso general de árboles 2-3, que para un orden M, contienen M-1 claves.

ARBOLES BINARIOS DE BÚSQUEDA (ABB)

Se define como:

1) Los nodos están etiquetados con elementos de un conjunto, por lo tanto no contiene elementos repetidos (clave única).
2) Un árbol binario, es decir, el número máximo de hijos de cada nodo es dos (2)
3) Todos los nodos situados a su izquierda son menores que él.
4) Todos los nodos situados a su derecha son mayores que él.
5) Mantienen una secuencia ordenada si se recorren en inorden.



Operaciones sobre ABB:

El repertorio de operaciones que se pueden realizar sobre un ABB es parecido al que realizábamos sobre otras estructuras de datos, más alguna otra propia de árboles:

- Buscar un elemento.

  Partiendo siempre del nodo raíz, el modo de buscar un elemento se define de forma recursiva:

  - Si el árbol está vacío, terminamos la búsqueda: el elemento no está en el árbol.
  - Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda con éxito.
  - Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
  - Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

El valor de retorno de una función de búsqueda en un ABB puede ser un puntero al nodo encontrado, o NULL, si no se ha encontrado.

➢ Insertar un elemento.

Para insertar un elemento nos basamos en el algoritmo de búsqueda. Si el elemento está en el árbol no lo insertaremos. Si no lo está, lo insertaremos a continuación del último nodo visitado.

Necesitamos un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

- Padre = NULL
- nodo = Raiz
- Bucle: mientras actual no sea un árbol vacío o hasta que se encuentre el elemento.
    - Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo: Padre=nodo, nodo=nodo->izquierdo.
    - Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho: Padre=nodo, nodo=nodo->derecho.
- Si nodo no es NULL, el elemento está en el árbol, por lo tanto salimos.
- Si Padre es NULL, el árbol estaba vacío, por lo tanto, el nuevo árbol sólo contendrá el nuevo elemento, que será la raíz del árbol.
- Si el elemento es menor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol izquierdo de Padre.
- Si el elemento es mayor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol derecho de Padre.

Este modo de actuar asegura que el árbol sigue siendo ABB.

➢ Borrar un elemento.

Para borrar un elemento también nos basamos en el algoritmo de búsqueda. Si el elemento no está en el árbol no lo podremos borrar. Si está, hay dos casos posibles:

1. Se trata de un nodo hoja: en ese caso lo borraremos directamente.
2. Se trata de un nodo rama: en ese caso no podemos eliminarlo, puesto que perderíamos todos los elementos del árbol de que el nodo actual es padre. En su lugar buscamos el nodo más a la izquierda del subárbol derecho, o el más a la derecha del subárbol izquierdo e intercambiamos sus valores. A continuación eliminamos el nodo hoja.

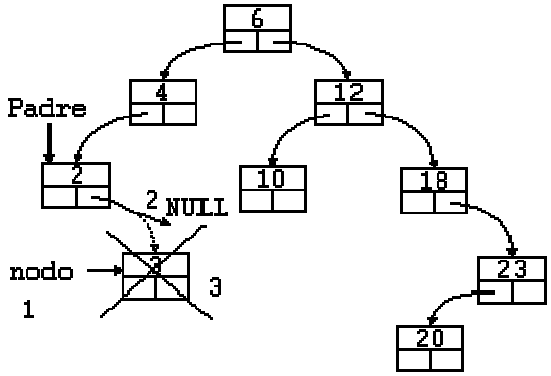
Necesitamos un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

- Padre = NULL
- Si el árbol está vacío: el elemento no está en el árbol, por lo tanto salimos sin eliminar ningún elemento.
- (1) Si el valor del nodo raíz es igual que el del elemento que buscamos, estamos ante uno de los siguientes casos:
    - El nodo raíz es un nodo hoja:
        - Si 'Padre' es NULL, el nodo raíz es el único del árbol, por lo tanto el puntero al árbol debe ser NULL.
        - Si raíz es la rama derecha de 'Padre', hacemos que esa rama apunte a NULL.
        - Si raíz es la rama izquierda de 'Padre', hacemos que esa rama apunte a NULL.
        - Eliminamos el nodo, y salimos.
    - El nodo no es un nodo hoja:
        - Buscamos el 'nodo' más a la izquierda del árbol derecho de raíz o el más a la derecha del árbol izquierdo. Hay que tener en cuenta que puede que sólo exista uno de esos árboles. Al mismo tiempo, actualizamos 'Padre' para que apunte al padre de 'nodo'.
        - Intercambiamos los elementos de los nodos raíz y 'nodo'.
        - Borramos el nodo 'nodo'. Esto significa volver a (1), ya que puede suceder que 'nodo' no sea un nodo hoja. (Ver ejemplo 3)
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

Ejemplo 1: Borrar un nodo hoja

En el árbol de ejemplo, borrar el nodo 3.

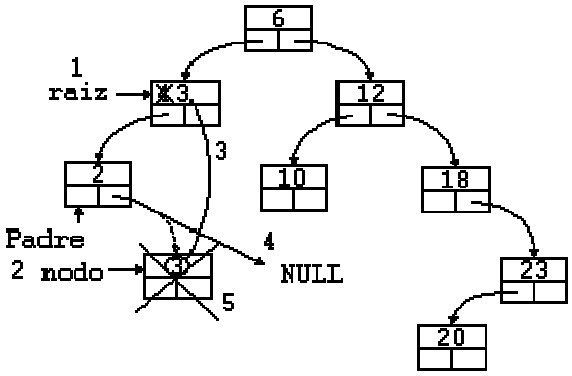
1. Localizamos el nodo a borrar, al tiempo que mantenemos un puntero a 'Padre'.
2. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
3. Borramos el 'nodo'.



Ejemplo 2: Borrar un nodo rama con intercambio de un nodo hoja.

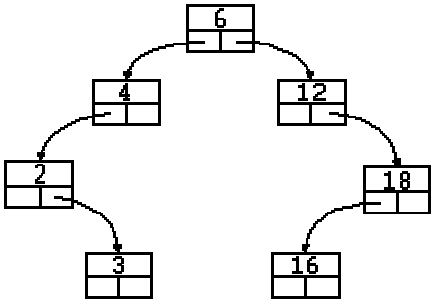
En el árbol de ejemplo, borrar el nodo 4.

1. Localizamos el nodo a borrar ('raíz').
2. Buscamos el nodo más a la derecha del árbol izquierdo de 'raíz', en este caso el 3, al tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
3. Intercambiamos los elementos 3 y 4.
4. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
5. Borramos el 'nodo'.

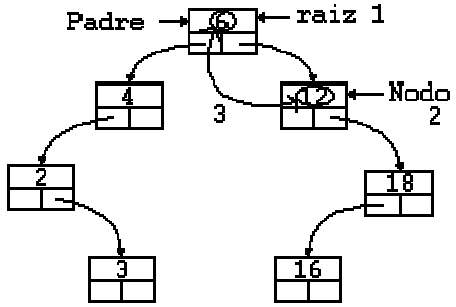


Ejemplo 3: Borrar un nodo rama con intercambio de un nodo rama.

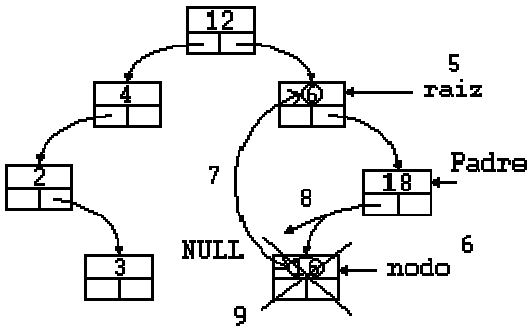
Para este ejemplo usaremos otro árbol. En éste borraremos el elemento 6.



1. Localizamos el nodo a borrar ('raíz').
2. Buscamos el nodo más a la izquierda del árbol derecho de 'raíz', en este caso el 12, ya que el árbol derecho no tiene nodos a su izquierda, si optamos por la rama izquierda, estaremos en un caso análogo. Al mismo tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
3. Intercambiamos los elementos 6 y 12.
4. Ahora tenemos que repetir el bucle para el nodo 6 de nuevo, ya que no podemos eliminarlo.



5. Localizamos de nuevo el nodo a borrar ('raíz').
6. Buscamos el nodo más a la izquierda del árbol derecho de 'raíz', en este caso el 16, al mismo tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
7. Intercambiamos los elementos 6 y 16.
8. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
9. Borramos el 'nodo'.



Este modo de actuar asegura que el árbol sigue siendo ABB.

- Movimientos a través del árbol:

Nuestra estructura se referenciará siempre mediante un puntero al nodo Raíz, este puntero no debe perderse nunca.

Para movernos a través del árbol usaremos punteros auxiliares, de modo que desde cualquier puntero los movimientos posibles serán: moverse al nodo raíz de la rama izquierda, moverse al nodo raíz de la rama derecha o moverse al nodo Raíz del árbol.

- Información:

Hay varios parámetros que podemos calcular o medir dentro de un árbol. Algunos de ellos nos darán idea de lo eficientemente que está organizado o el modo en que funciona.

Comprobar si un árbol está vacío.

Un árbol está vacío si su raíz es NULL.

Calcular el número de nodos.

Tenemos dos opciones para hacer esto, una es llevar siempre la cuenta de nodos en el árbol al mismo tiempo que se añaden o eliminan elementos. La otra es, sencillamente, contarlos.

Para contar los nodos podemos recurrir a cualquiera de los tres modos de recorrer el árbol: inorden, preorden o postorden, como acción sencillamente incrementamos el contador.

Comprobar si el nodo es hoja.

Esto es muy sencillo, basta con comprobar si tanto el árbol izquierdo como el derecho están vacíos. Si ambos lo están, se trata de un nodo hoja.

Calcular la altura de un nodo.

No hay un modo directo de hacer esto, ya que no nos es posible recorrer el árbol en la dirección de la raíz. De modo que tendremos que recurrir a otra técnica para calcular la altura.

Lo que haremos es buscar el elemento del nodo de que queremos averiguar la altura. Cada vez que avancemos un nodo incrementamos la variable que contendrá la altura del nodo.

- Empezamos con el nodo raíz apuntando a Raiz, y la 'Altura' igual a cero.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda y el valor de la altura es 'Altura'.
- Incrementamos 'Altura'.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

Calcular la altura de un árbol.

La altura del árbol es la altura del nodo de mayor altura. Para buscar este valor tendremos que recorrer todo el árbol, de nuevo es indiferente el tipo de recorrido que hagamos, cada vez que cambiemos de nivel incrementamos la variable que contiene la altura del nodo actual, cuando lleguemos a un nodo hoja compararemos su altura con la variable que contiene la altura del árbol si es mayor, actualizamos la altura del árbol.

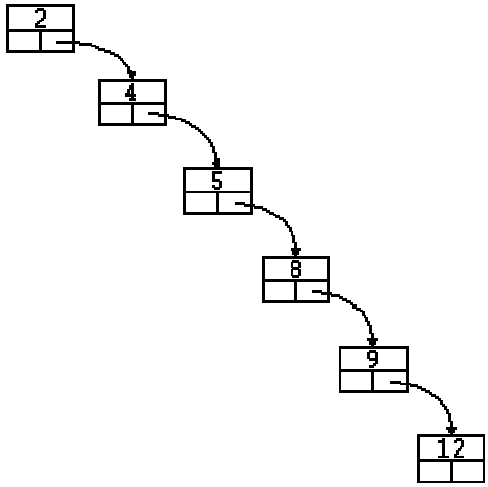
- Iniciamos un recorrido del árbol en postorden, con la variable de altura igual a cero.
- Cada vez que empecemos a recorrer una nueva rama, incrementamos la altura para ese nodo.
- Después de procesar las dos ramas, verificamos si la altura del nodo es mayor que la variable que almacena la altura actual del árbol, si es así, actualizamos esa variable.

ÁRBOLES DEGENERADOS:

Los árboles binarios de búsqueda tienen un gran inconveniente. Por ejemplo, supongamos que creamos un ABB a partir de una lista de valores ordenada:

2, 4, 5, 8, 9, 12

Difícilmente podremos llamar a la estructura resultante un árbol:



Esto es lo que llamamos un árbol binario de búsqueda degenerado, por lo que veremos una nueva estructura, el árbol AVL, que resuelve este problema, generando árboles de búsqueda equilibrados.

ÁRBOLES EQUILIBRADOS AVL

Un árbol AVL (llamado así por las iniciales de sus inventores: Adelson-Velskii y Landis) es un árbol binario de búsqueda en el que para cada nodo, las alturas de sus subárboles izquierdo y derecho no difieren en más de 1.

No se trata de árboles perfectamente equilibrados, pero sí son lo suficientemente equilibrados como para que su comportamiento sea lo bastante bueno como para usarlos donde los ABB no garantizan tiempos de búsqueda óptimos.

El algoritmo para mantener un árbol AVL equilibrado se basa en reequilibrados locales, de modo que no es necesario explorar todo el árbol después de cada inserción o borrado.

Operaciones en AVL:

Los AVL son también ABB, de modo que mantienen todas las operaciones que poseen éstos. Las nuevas operaciones son las de equilibrar el árbol, pero eso se hace como parte de las operaciones de insertado y borrado.

Factor de Balance o Equilibrio:

Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los ABB, y además un miembro nuevo: el factor de equilibrio. El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

FE = altura subárbol derecho - altura subárbol izquierdo;

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

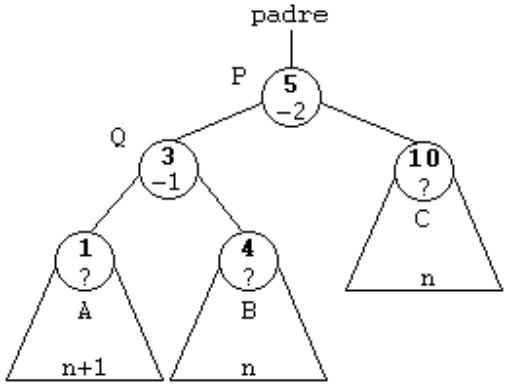
Rotaciones de nodos:

El balanceo se realiza mediante rotaciones, en el siguiente punto veremos cada caso, ahora vamos a ver las cuatro posibles rotaciones que podemos aplicar.

Rotaciones simples:

Rotación simple a la derecha (RSD):

Esta rotación se usará cuando el factor de equilibrio del subárbol izquierdo de un nodo sea 2 unidades más alto que el derecho, es decir, cuando su FE sea de -2. Y además, la raíz del subárbol izquierdo tenga una FE de -1, es decir, que esté cargado a la izquierda.

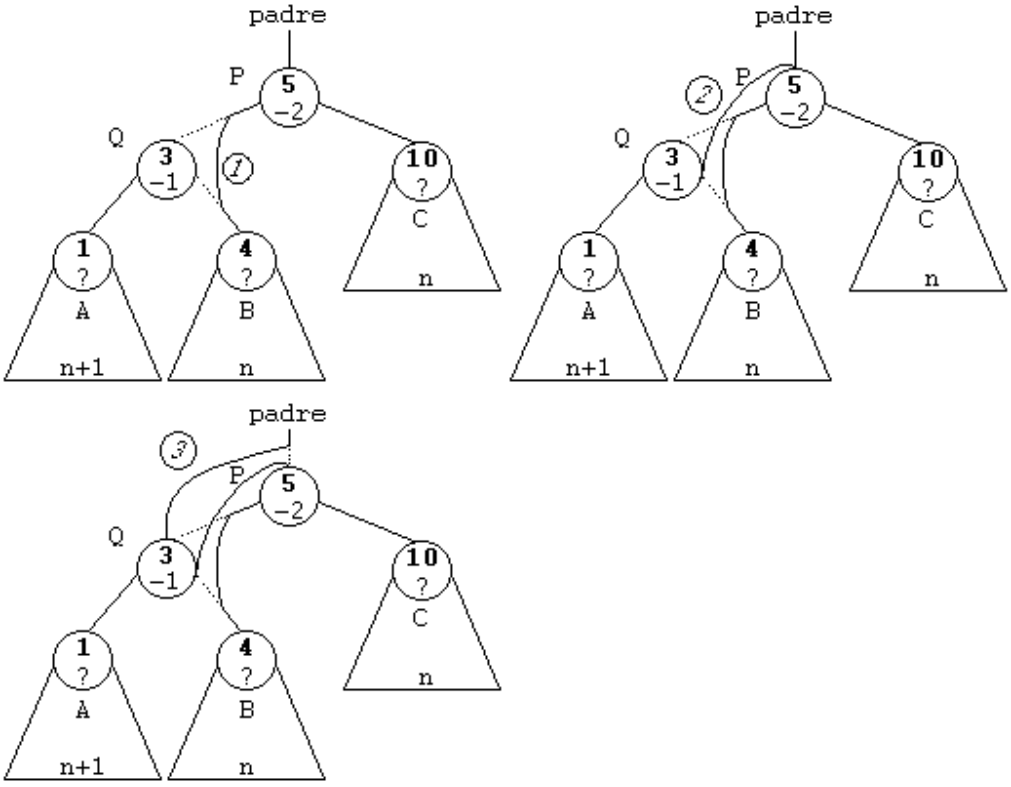


Procederemos del siguiente modo:

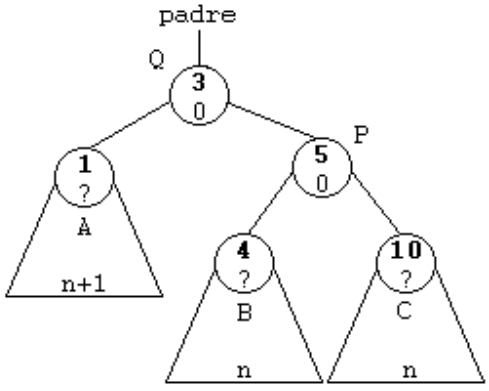
Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de -2. Y llamaremos Q al nodo raíz del subárbol izquierdo de P. Además, llamaremos A al subárbol izquierdo de Q, B al subárbol derecho de Q y C al subárbol derecho de P.

En el gráfico que puede observar que tanto B como C tienen la misma altura (n), y A es una unidad mayor (n+1). Esto hace que el FE de Q sea -1, la altura del subárbol que tiene Q como raíz es (n+2) y por lo tanto el FE de P es -2.

1. Pasamos el subárbol derecho del nodo Q como subárbol izquierdo de P. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de Q siguen estando a la izquierda de P.
2. El árbol P pasa a ser el subárbol derecho del nodo Q.
3. Ahora, el nodo Q pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo Q, en lugar del nodo P. Previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de mayor altura.

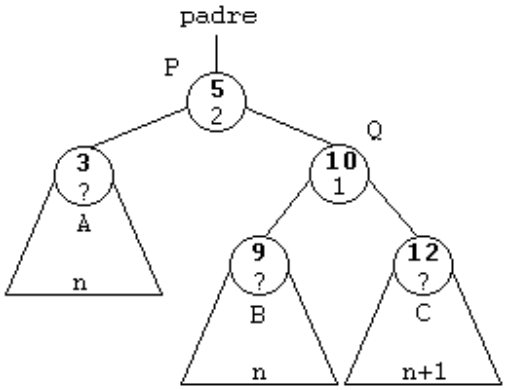


En el árbol resultante se puede ver que tanto P como Q quedan equilibrados en cuanto altura. En el caso de P porque sus dos subárboles tienen la misma altura (n), en el caso de Q, porque su subárbol izquierdo A tiene una altura (n+1) y su subárbol derecho también, ya que a P se añade la altura de cualquiera de sus subárboles.



Rotación simple a la izquierda (RSI):

Se trata del caso simétrico del anterior. Esta rotación se usará cuando el subárbol derecho de un nodo sea 2 unidades más alto que el izquierdo, es decir, cuando su FE sea de 2. Y además, la raíz del subárbol derecho tenga una FE de 1, es decir, que esté cargado a la derecha.

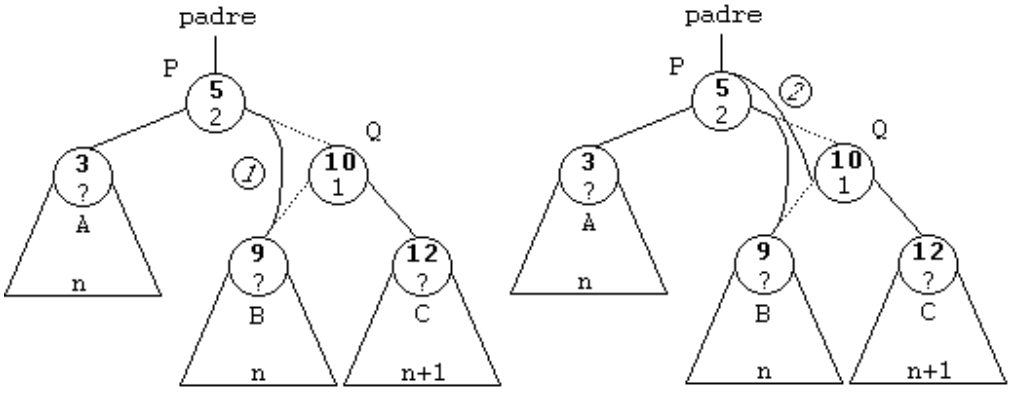


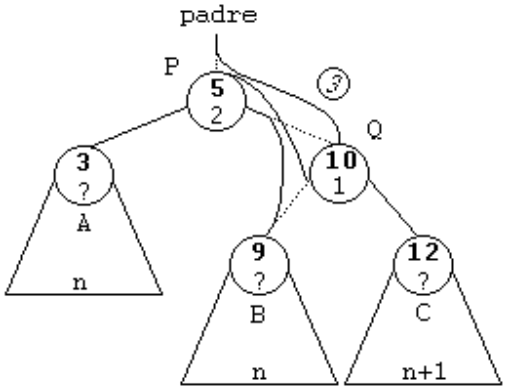
Procederemos del siguiente modo:

Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de 2. Y llamaremos Q al nodo raíz del subárbol derecho de P. Además, llamaremos A al subárbol izquierdo de P, B al subárbol izquierdo de Q y C al subárbol derecho de Q.

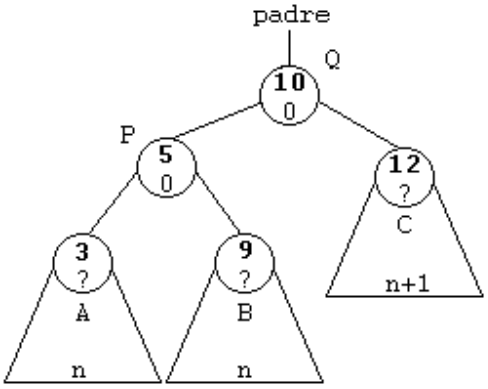
En el gráfico que puede observar que tanto A como B tienen la misma altura (n), y C es una unidad mayor (n+1). Esto hace que el FE de Q sea 1, la altura del subárbol que tiene Q como raíz es (n+2) y por lo tanto el FE de P es 2.

1. Pasamos el subárbol izquierdo del nodo Q como subárbol derecho de P. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de Q siguen estando a la derecha de P.
2. El árbol P pasa a ser el subárbol izquierdo del nodo Q.
3. Ahora, el nodo Q pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo Q, en lugar del nodo P. Previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.





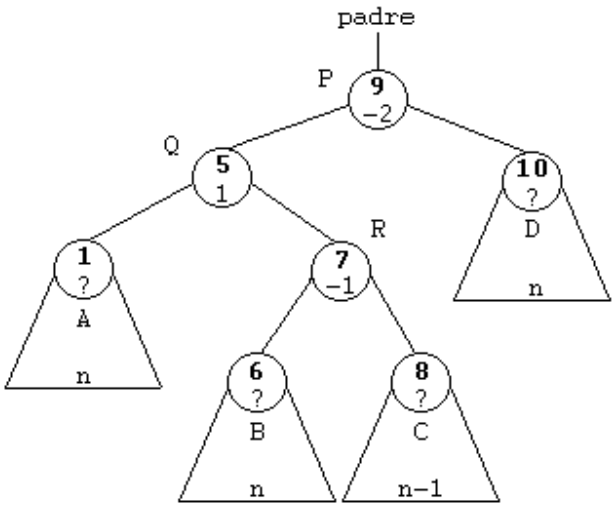
En el árbol resultante se puede ver que tanto P como Q quedan equilibrados en cuanto altura. En el caso de P porque sus dos subárboles tienen la misma altura (n), en el caso de Q, porque su subárbol izquierdo A tiene una altura (n+1) y su subárbol derecho también, ya que a P se añade la altura de cualquiera de sus subárboles.



Rotaciones dobles:

Rotación doble a la derecha (RDD):

Esta rotación se usará cuando el subárbol izquierdo de un nodo sea 2 unidades más alto que el derecho, es decir, cuando su FE sea de -2. Y además, la raíz del subárbol izquierdo tenga una FE de 1, es decir, que esté cargado a la derecha.



Este es uno de los posibles árboles que pueden presentar esta estructura, pero hay otras dos posibilidades. El nodo R puede tener una FE de -1, 0 ó 1. En cada uno de esos casos los árboles izquierdo y derecho de R (B y C) pueden tener alturas de n y n-1, n y n, o n-1 y n, respectivamente.

El modo de realizar la rotación es independiente de la estructura del árbol R, cualquiera de las tres produce resultados equivalentes. Haremos el análisis para el caso en que FE sea -1.

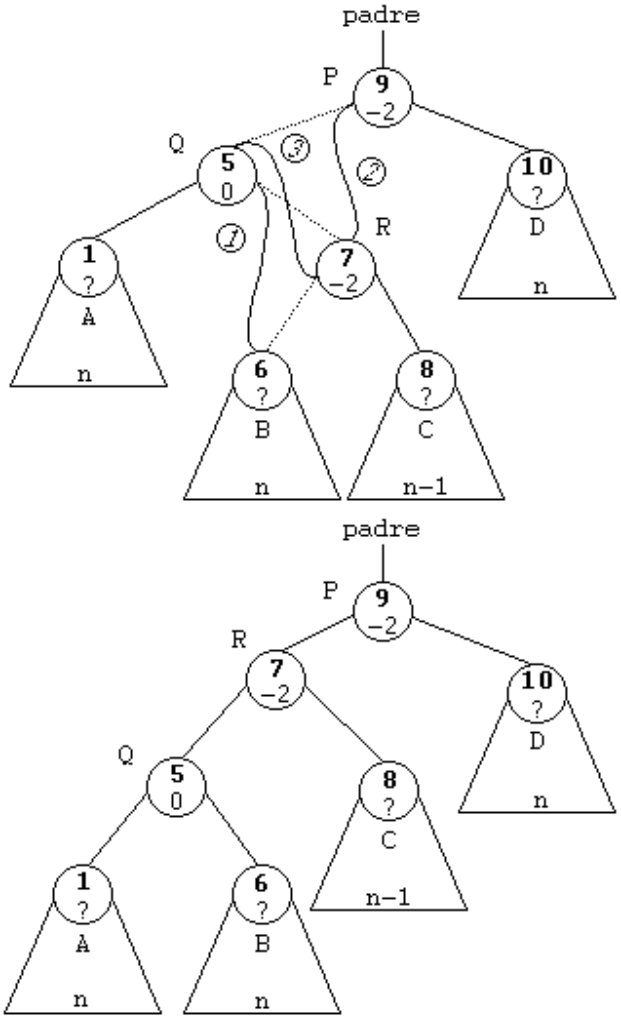
En este caso tendremos que realizar dos rotaciones.

Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de -2. Llamaremos Q al nodo raíz del subárbol izquierdo de P, y R al nodo raíz del subárbol derecho de Q.

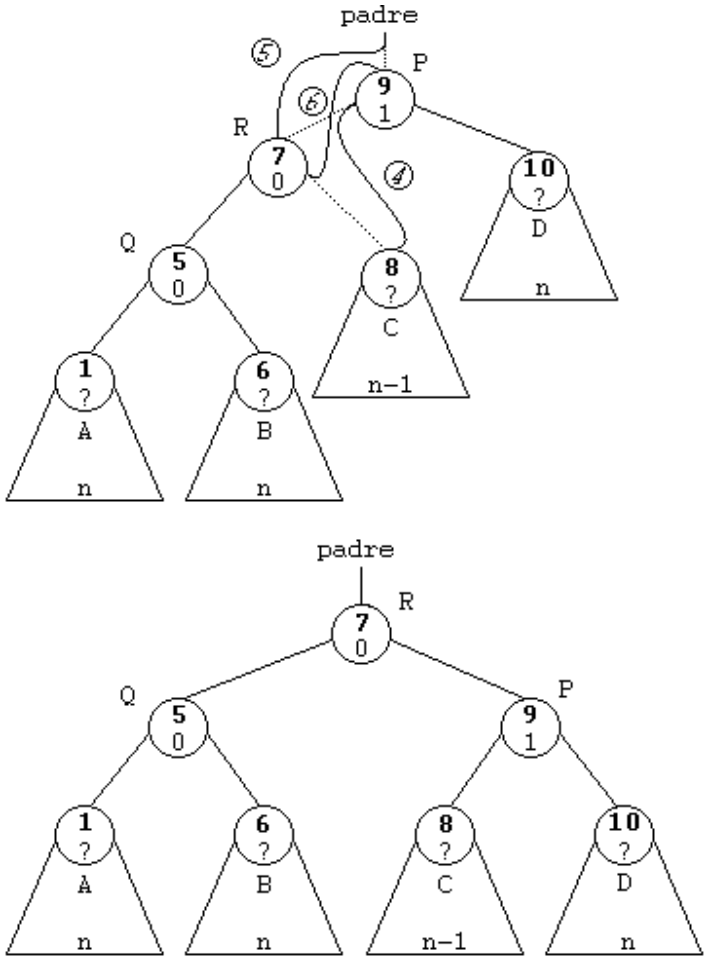
1. Haremos una rotación simple de Q a la izquierda.
2. Después, haremos una rotación simple de P a la derecha.

Con más detalle, procederemos del siguiente modo:

1. Pasamos el subárbol izquierdo del nodo R como subárbol derecho de Q. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de R siguen estando a la derecha de Q.
2. Ahora, el nodo R pasa a tomar la posición del nodo Q, es decir, hacemos que la raíz del subárbol izquierdo de P sea el nodo R en lugar de Q.
3. El árbol Q pasa a ser el subárbol izquierdo del nodo R.

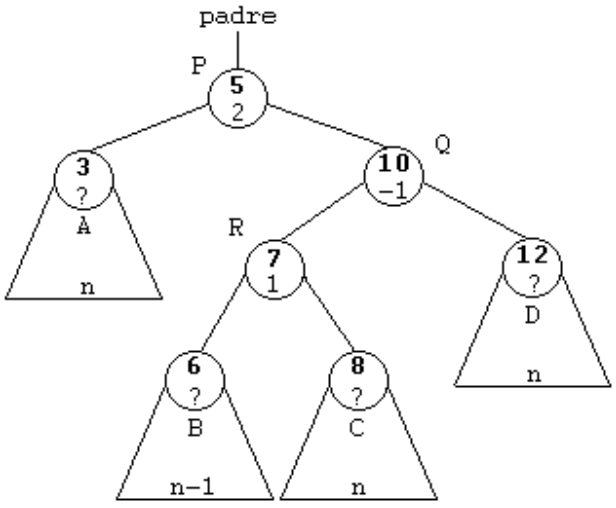


4. Pasamos el subárbol derecho del nodo R como subárbol izquierdo de P. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de R siguen estando a la izquierda de P.
5. Ahora, el nodo R pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo R, en lugar del nodo P. Como en los casos anteriores, previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.
6. El árbol P pasa a ser el subárbol derecho del nodo R.



Rotación doble a la izquierda (RDI):

Esta rotación se usará cuando el subárbol derecho de un nodo sea 2 unidades más alto que el izquierdo, es decir, cuando su FE sea de 2. Y además, la raíz del subárbol derecho tenga una FE de -1, es decir, que esté cargado a la izquierda. Se trata del caso simétrico del anterior.



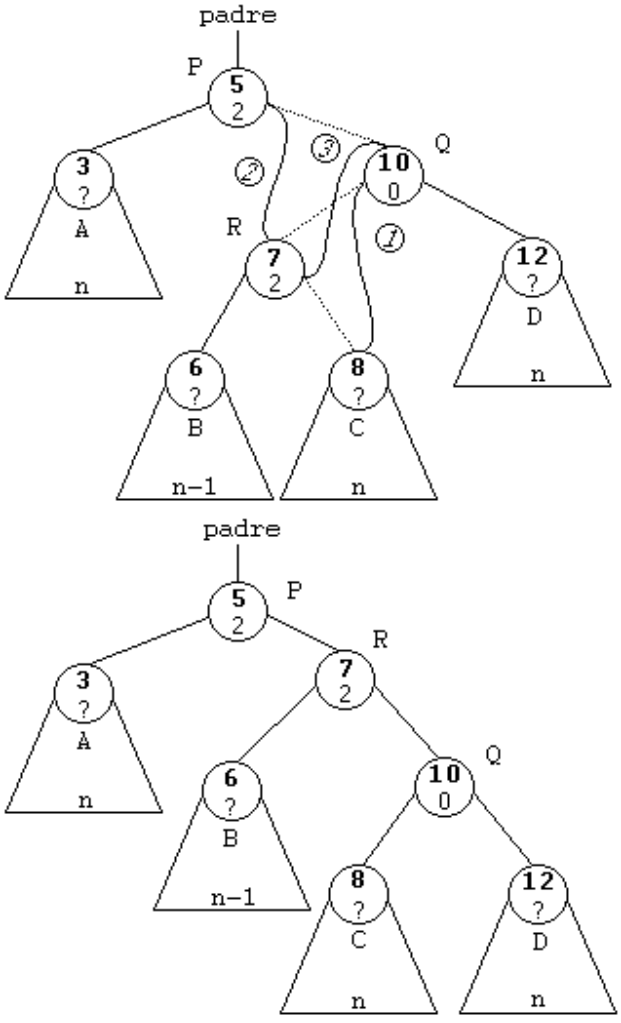
En este caso también tendremos que realizar dos rotaciones.

Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de 2. Llamaremos Q al nodo raíz del subárbol derecho de P, y R al nodo raíz del subárbol izquierdo de Q.

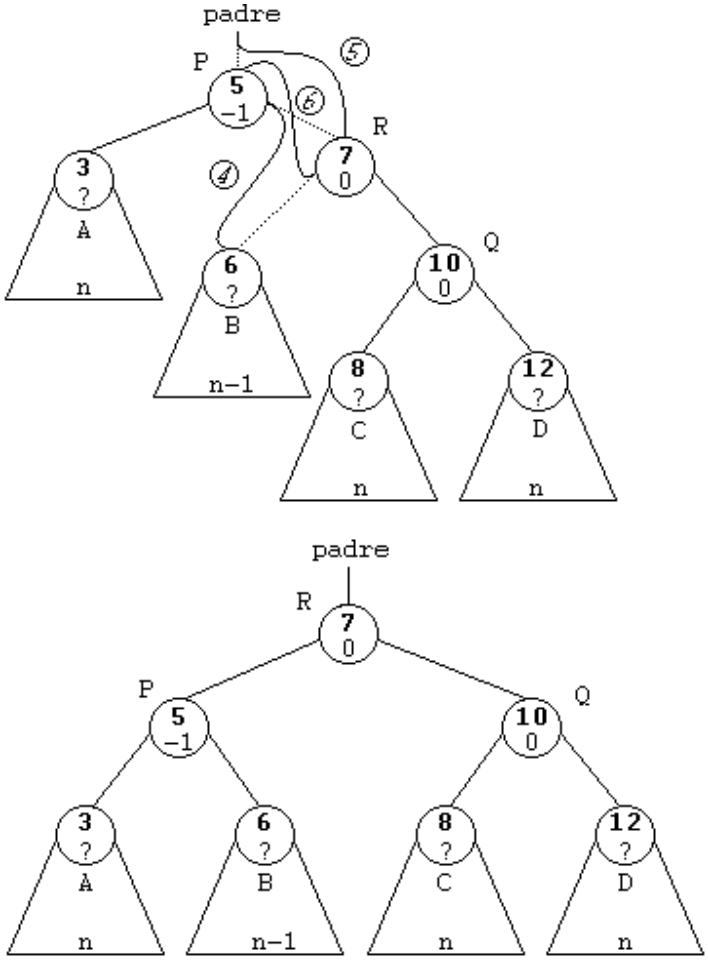
1. Haremos una rotación simple de Q a la derecha.
2. Después, haremos una rotación simple de P a la izquierda.

Con más detalle, procederemos del siguiente modo:

1. Pasamos el subárbol derecho del nodo R como subárbol izquierdo de Q. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de R siguen estando a la izquierda de Q.
2. Ahora, el nodo R pasa a tomar la posición del nodo Q, es decir, hacemos que la raíz del subárbol derecho de P sea el nodo R en lugar de Q.
3. El árbol Q pasa a ser el subárbol derecho del nodo R.



4. Pasamos el subárbol izquierdo del nodo R como subárbol derecho de P. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de R siguen estando a la derecha de P.
5. Ahora, el nodo R pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo R, en lugar del nodo P. Como en los casos anteriores, previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.
6. El árbol P pasa a ser el subárbol izquierdo del nodo R.



Reequilibrio o Balanceo de un árbol AVL:

Cada vez que insertemos o eliminemos un nodo en un árbol AVL pueden suceder dos cosas: que el árbol se mantenga como AVL o que pierda esta propiedad. En el segundo caso siempre estaremos en uno de los explicados anteriormente, y recuperaremos el estado AVL aplicando la rotación adecuada.

Ya comentamos que necesitamos añadir un nuevo miembro a cada nodo del árbol para averiguar si el árbol sigue siendo AVL, el Factor de Equilibrio. Cada vez que insertemos o eliminemos un nodo deberemos recorrer el camino desde ese nodo hacia el nodo raíz actualizando los valores de FE de cada nodo. Cuando uno de esos valores sea 2 ó -2 aplicaremos la rotación correspondiente.

Debido a que debemos ser capaces de recorrer el árbol en dirección a la raíz, añadiremos un nuevo puntero a cada nodo que apunte al nodo padre. Esto complicará algo las operaciones de inserción, borrado y rotación, pero facilita y agiliza mucho el cálculo del FE, y veremos que las complicaciones se compensan en gran parte por las facilidades obtenidas al disponer de este puntero.

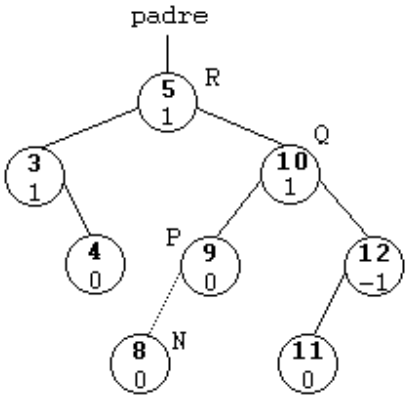
Nota: En rigor, no es necesario ese puntero, podemos almacenar el camino que recorremos para localizar un nodo concreto usando una pila, y después podemos usar la pila para recuperar el camino en orden inverso. Pero esto nos obliga a introducir otra estructura dinámica, y según mi opinión, complica en exceso el algoritmo.

Cuando estemos actualizando los valores de FE no necesitamos calcular las alturas de las dos ramas de cada nodo, sabiendo en valor anterior de FE, y sabiendo en qué rama hemos añadido o eliminado el nodo, es fácil calcular el nuevo valor de FE. Si el nodo ha sido añadido en la rama derecha o eliminado en la izquierda, y ha habido un cambio de altura en la rama, se incrementa el valor de FE; si el nodo ha sido añadido en la rama izquierda o eliminado en la derecha, y ha habido un cambio de altura en la rama, se decrementa el valor de FE.

Los cambios de altura en una rama se producen sólo cuando el FE del nodo raíz de esa rama ha cambiado de 0 a 1 ó de 0 a -1. En caso contrario, cuando el FE cambia de 1 a 0 ó de -1 a 0, no se produce cambio de altura.

Si no hay cambio de altura, los valores de FE del resto de los nodos hasta el raíz no pueden cambiar, recordemos que el factor de equilibrio se define como la diferencia de altura entre las ramas derecha e izquierda de un nodo, la altura de la rama que no pertenece al camino no puede cambiar, puesto que sigue teniendo los mismos nodos que antes, de modo que si la altura de la rama que pertenece al camino no cambia, tampoco puede cambiar el valor de FE.

Por ejemplo, supongamos que en siguiente árbol AVL insertamos el nodo de valor 8:

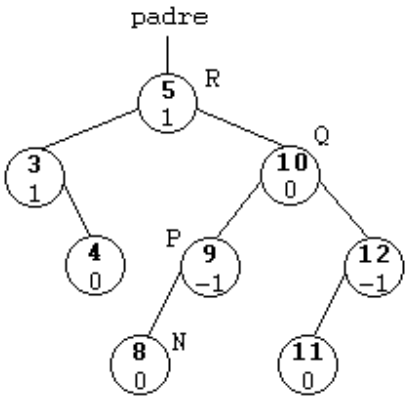


Para empezar, cualquier nodo nuevo será un nodo hoja, de modo que su FE será siempre 0.

Ahora actualizamos el valor de FE del nodo padre del que acabamos de insertar (P). El valor previo es 0, y hemos añadido un nodo en su rama izquierda, por lo tanto, el nuevo valor es -1. Esto implica un cambio de altura, por lo tanto, continuamos camino hacia la raíz.

A continuación tomamos el nodo padre de P (Q), cuyo valor previo de FE era 1, y al que también hemos añadido un nodo en su rama izquierda, por lo tanto decrementamos ese valor, y el nuevo será 0. En este caso no ha incremento de altura, la altura del árbol cuya raíz es Q sigue siendo la misma, por lo tanto, ninguno de los valores de FE de los nodos hasta el raíz puede haber cambiado. Es decir, no necesitamos seguir recorriendo el camino.

Si verificamos el valor de FE del nodo R vemos que efectivamente se mantiene, puesto que tanto la altura del subárbol derecho como del izquierdo, siguen siendo las mismas.



Pero algunas veces, el valor de FE del nodo es -2 ó 2, son los casos en los que perdemos la propiedad AVL del árbol, y por lo tanto tendremos que recuperarla.

Reequilibrados en árboles AVL por inserción de un nodo

En ese caso, cuando el valor de FE de un nodo tome el valor -2 ó 2, no seguiremos el camino, sino que, con el valor de FE de el nodo actual y el del nodo derecho si FE es 2 o el del nodo izquierdo si es -2, determinaremos qué tipo de rotación debemos hacer.

FE nodo actual	FE del nodo derecho	FE del nodo izquierdo	Rotación
-2	No importa	-1	RSD
-2	No importa	1	RDD
2	-1	No importa	RDI
2	1	No importa	RSI

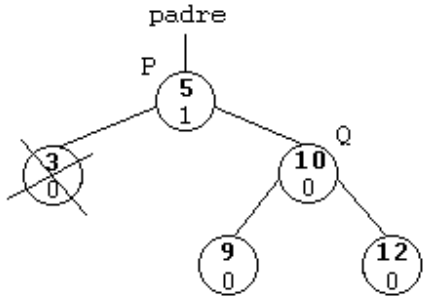
El resto de los casos no nos interesan. Esto es porque en nodos desequilibrados hacia la derecha, con valores de FE positivos, siempre buscaremos el equilibrio mediante rotaciones a la izquierda, y viceversa, con nodos desequilibrados hacia la izquierda, con valores de FE negativos, buscaremos el equilibrio mediante rotaciones a la derecha.

Supongamos que el valor de FE del nodo ha pasado de -1 a -2, debido a que se ha añadido un nodo. Esto implica que el nodo añadido lo ha sido en la rama izquierda, si lo hubiéramos añadido en la derecha el valor de FE nunca podría decrecer.

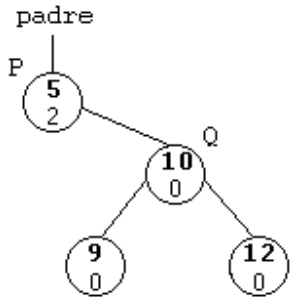
Reequilibrados en árboles AVL por borrado de un nodo

Cuando el desequilibrio se debe a la eliminación de un nodo la cosa puede ser algo diferente, pero veremos que siempre se puede llegar a uno de los casos anteriores.

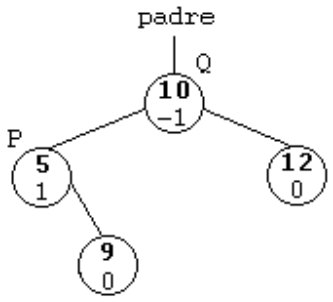
Supongamos el siguiente ejemplo, en el árbol AVL eliminaremos el nodo de valor 3:



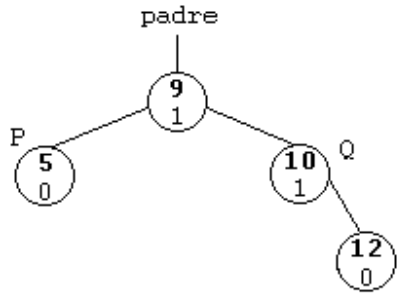
El valor de FE del nodo P pasa de 1 a 2, sabemos que cuando el valor de FE de un nodo es 2 siempre tenemos que aplicar una rotación a izquierdas. Para saber cual de las dos rotaciones debemos aplicar miramos el valor de FE del nodo derecho. Pero en este caso, el valor de FE de ese nodo es 0. Esto no quiere decir que no podamos aplicar ninguna de las rotaciones, por el contrario, podremos aplicar cualquiera de ellas. Aunque por economía, lo razonable es aplicar la rotación simple.



Si aplicamos la rotación simple, el resultado es:



Y aplicando la rotación doble:



Del mismo modo, el valor de FE del nodo derecho podría haber sido 1 ó -1, en ese caso sí está determinado el tipo de rotación a realizar.

El razonamiento es similar cuando se eliminan nodos y el resultado es que se obtiene un nodo con FE de -2, en este caso se realizará una rotación a derechas, y la rotación dependerá del valor de FE del nodo izquierdo al que muestra el desequilibrio. Si es 0 ó -1 haremos una rotación simple, si es 1, haremos una rotación doble.

Tendremos entonces una tabla más general para decidir la rotación a aplicar:

FE nodo actual	FE del nodo derecho	FE del nodo izquierdo	Rotación
-2	No importa	-1	RSD
-2	No importa	0	RSD
-2	No importa	1	RDD
2	-1	No importa	RDI
2	0	No importa	RSI
2	1	No importa	RSI

Los árboles AVL siempre quedan equilibrados después de una rotación.

Esto puede comprobarse analizando los métodos de rotación que hemos estudiado, después de efectuada la rotación, la altura del árbol cuya raíz es el nodo rotado se mantiene, por lo tanto, no necesitamos continuar el camino hacia la raíz: sabemos que el árbol es AVL.

De inserción de nodo

En general, la inserción de nodos en un árbol AVL es igual que en un árbol ABB, la diferencia es que en un árbol AVL, después de insertar el nodo debemos recorrer el árbol en sentido hacia la raíz, recalculando los valores de FE, hasta que se cumpla una de estas condiciones: que lleguemos a la raíz, que se encuentre un nodo con valor de FE de 2, ó -2, o que se llegue a un nodo cuyo FE no cambie o decrezca en valor absoluto, es decir, que cambie de 1 a 0 ó de -1 a 0.

Podemos considerar que el algoritmo de inserción de nodos en árboles AVL es una ampliación del que vimos para árboles ABB.

De borrado de nodo

Lo mismo pasa cuando se eliminan nodos, el algoritmo es el mismo que en árboles ABB, pero después de eliminar el nodo debemos recorrer el camino hacia la raíz recalculando los valores de FE, y equilibrando el árbol si es necesario.

De recalcular FE

Ya comentamos más atrás que para seguir el camino desde el nodo insertado o borrado hasta el nodo raíz tenemos dos alternativas:

1. Guardar en una pila los punteros a los nodos por los que hemos pasado para llegar al nodo insertado o borrado, es decir, almacenar el camino.
2. Añadir un nuevo puntero a cada nodo que apunte al padre del nodo actual. Esto nos permite recorrer el árbol en el sentido contrario al normal, es decir, en dirección a la raíz.

Para calcular los nuevos valores de FE de los nodos del camino hay que tener en cuenta los siguientes hechos:

- El valor de FE de un nodo insertado es cero, ya que siempre insertaremos nodos hoja.
- Si el nuevo valor de FE para cualquiera de los siguientes nodos del camino es cero, habremos terminado de actualizar los valores de FE, ya que la rama mantiene su altura, la inserción o borrado del nodo no puede influir en los valores de FE de los siguientes nodos del camino.
- Cuando se elimine un nodo pueden pasar dos cosas. Siempre eliminamos un nodo hoja, ya que cuando no lo es, lo intercambiamos con un nodo hoja antes de eliminarlo. Pero algunas veces, el nodo padre del nodo eliminado se convertirá a su vez en nodo hoja, y en ese caso no siempre hay que dar por terminada la actualización del FE del camino. Por lo tanto, cuando eliminemos un nodo, actualizaremos el valor de FE del nodo padre y continuaremos el camino, independientemente del valor de FE calculado.
- A la hora de actualizar el valor de FE de un nodo, tenemos que distinguir cuando el equilibrado sea consecuencia de una inserción o lo sea de una eliminación. Incrementaremos el valor de FE del nodo si la inserción fue en la rama derecha o si la eliminación fue en la rama izquierda, decrementaremos si la inserción fue en la izquierda o la eliminación en la derecha.
- Si en valor de FE es -2, haremos una rotación doble a la derecha su el valor de FE del nodo izquierdo es 1, y simple si es 1 ó 0.
- Si en valor de FE es 2, haremos una rotación doble a la izquierda su el valor de FE del nodo izquierdo es -1, y simple si es -1 ó 0.
- En cualquiera de los dos casos, podremos dar por terminado el recorrido del camino, ya que la altura del árbol cuya raíz es un nodo rotado no cambia.
- En cualquier otro caso, seguiremos actualizando hasta llegar al nodo raíz.

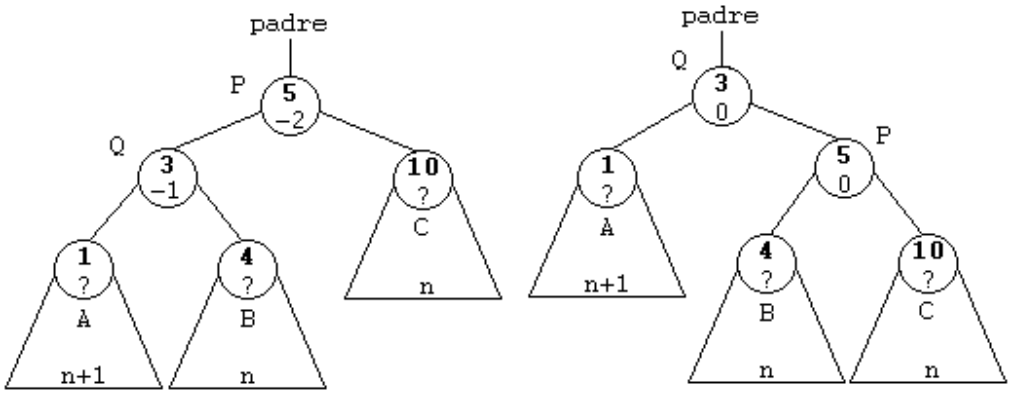
De rotación simple

A la hora de implementar los algoritmos que hemos visto para rotaciones simples tenemos dos opciones: seguir literalmente los pasos de los gráficos, o tomar un atajo, y hacerlo

mediante asignaciones. Nosotros lo haremos del segundo modo, ya que resulta mucho más rápido y sencillo.

Primero haremos las reasignaciones de punteros, de modo que el árbol resultante responda a la estructura después de la rotación. Después actualizaremos los punteros al nodo padre para los nodos que han cambiado de posición. Por último actualizaremos los valores de FE de esos mismos nodos.

Para la primera fase usaremos punteros auxiliares a nodo, que en el caso de rotación a la derecha necesitamos un puntero P al nodo con FE igual a -2. Ese será el parámetro de entrada, otro puntero al nodo izquierdo de P: Q. Y tres punteros más a los árboles A, B y C.



En realidad, si nos fijamos en los gráficos, los punteros a A y C no son necesarios, ya que ambos conservan sus posiciones, A sigue siendo el subárbol izquierdo de Q y C el subárbol derecho de P.

Usaremos otro puntero más: Padre, que apunte al padre de P. Disponiendo de los punteros Padre, P, Q y B, realizar la rotación es muy sencillo:

```
if(Padre)
  if(Padre->derecho == P) Padre->derecho = Q;
  else Padre->izquierdo = Q;
else raíz = Q;

// Reconstruir árbol:
P->izquierdo = B;
Q->derecho = P;
```

Hay que tener en cuenta que P puede ser la raíz de un subárbol derecho o izquierdo de otro nodo, o incluso la raíz del árbol completo. Por eso comprobamos si P tiene padre, y si lo tiene, cual de sus ramas apunta a P, cuando lo sabemos, hacemos que esa rama apunte a Q. Si Padre es NULL, entonces P era la raíz del árbol, así que hacemos que la nueva raíz sea Q.

Sólo nos queda trasladar el subárbol B a la rama izquierda de P, y Q a la rama derecha de P.

La segunda fase consiste en actualizar los punteros padre de los nodos que hemos cambiado de posición: P, B y Q.

```
  P->padre = Q;
  if(B) B->padre = P;
  Q->padre = Padre;
```

El padre de P es ahora Q, el de Q es Padre, y el de B, si existe es P.

La tercera fase consiste en ajustar los valores de FE de los nodos para los que puede haber cambiado.

Esto es muy sencillo, después de una rotación simple, los únicos valores de FE que cambian son los de P y Q, y ambos valen 0.

```
// Rotación simple a derechas
void RSD(Nodo* nodo)
{
  Nodo *Padre = nodo->padre;
  Nodo *P = nodo;
  Nodo *Q = P->izquierdo;
  Nodo *B = Q->derecho;

  if(Padre)
    if(Padre->derecho == P) Padre->derecho = Q;
    else Padre->izquierdo = Q;
  else raíz = Q;

  // Reconstruir árbol:
  P->izquierdo = B;
  Q->derecho = P;

  // Reasignar padres:
  P->padre = Q;
  if(B) B->padre = P;
  Q->padre = Padre;

  // Ajustar valores de FE:
  P->FE = 0;
  Q->FE = 0;
}
```

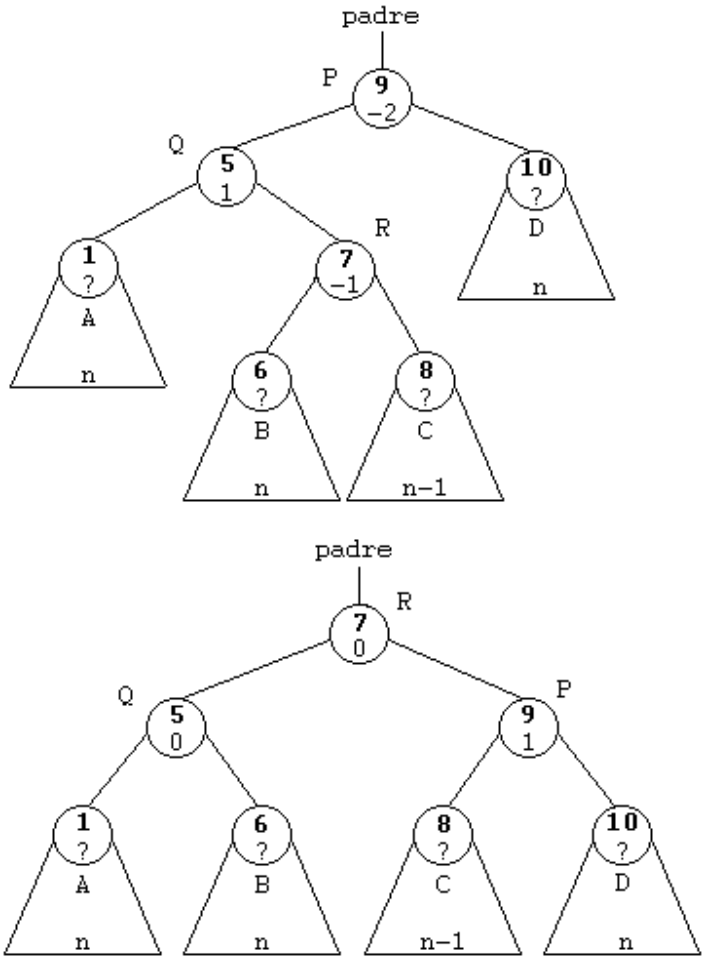
La rotación a izquierdas es simétrica.

De rotación doble

Para implementar las rotaciones dobles trabajaremos de forma análoga.

Primero haremos las reasignaciones de punteros, de modo que el árbol resultante responda a la estructura después de la rotación. Después actualizaremos los punteros al nodo padre para los nodos que han cambiado de posición. Por último actualizaremos los valores de FE de esos mismos nodos.

Para la primera fase usaremos punteros auxiliares a nodo, que en el caso de rotación a la derecha necesitamos un puntero P al nodo con FE igual a -2. Ese será el parámetro de entrada, otro puntero al nodo izquierdo de P: Q. Un tercero al nodo derecho de Q: R. Y cuatro punteros más a los árboles A, B, C y D.



En realidad, si nos fijamos en los gráficos, los punteros a A y D no son necesarios, ya que ambos conservan sus posiciones, A sigue siendo el subárbol izquierdo de Q y D el subárbol derecho de P.

También en este caso usaremos otro puntero más: Padre, que apunte al padre de P. Disponiendo de los punteros Padre, P, Q, R, B y C, realizar la rotación es muy sencillo:

```
if(Padre)
  if(Padre->derecho == nodo) Padre->derecho = R;
  else Padre->izquierdo = R;
else raíz = R;
```

```
// Reconstruir árbol:
Q->derecho = B;
P->izquierdo = C;
R->izquierdo = Q;
R->derecho = P;
```

Ahora también hay que tener en cuenta que P puede ser la raíz de un subárbol derecho o izquierdo de otro nodo, o incluso la raíz del árbol completo. Por eso comprobamos si P tiene padre, y si lo tiene, cual de sus ramas apunta a P, cuando lo sabemos, hacemos que esa rama apunte a R. Si Padre es NULL, entonces P era la raíz del árbol, así que hacemos que la nueva raíz sea R.

Sólo nos queda trasladar el subárbol B a la rama derecha de Q, C a la rama izquierda de P, Q a la rama izquierda de R y P a la rama derecha de R.

La segunda fase consiste en actualizar los punteros padre de los nodos que hemos cambiado de posición: P, Q, R, B y C.

```
R->>padre = Padre;
P->padre = Q->padre = R;
if(B) B->padre = Q;
if(C) C->padre = P;
```

El padre de R es ahora Padre, el de P y Q es R, y el de B, si existe es Q, y el de C, si existe, es P.

La tercera fase consiste en ajustar los valores de FE de los nodos para los que puede haber cambiado.

En las rotaciones dobles esto se complica un poco ya que puede suceder que el valor de FE de R antes de la rotación sea -1, 0 o 1. En cada caso, los valores de FE de P y Q después de la rotación serán diferentes.

```
// Ajustar valores de FE:
  switch(R->FE) {
     case -1: Q->FE = 0; P->FE = 1; break;
     case 0:  Q->FE = 0; P->FE = 0; break;
     case 1:  Q->FE = -1; P->FE = 0; break;
  }
  R->FE = 0;
```

Si la altura de B es n-1 y la de C es n, el valor de FE de R es 1. Después de la rotación, la rama B pasa a ser el subárbol derecho de Q, por lo tanto, la FE de Q, dado que la altura de su rama izquierda es n, será 0. La rama C pasa a ser el subárbol izquierdo de P, y dado que la altura de la rama derecha es n, la FE de P será -1.

Si la altura de B es n y la de C es n-1, el valor de FE de R es -1. Después de la rotación, la rama B pasa a ser el subárbol derecho de Q, por lo tanto, la FE de Q, dado que la altura de su rama izquierda es n, será 0. La rama C pasa a ser el subárbol izquierdo de P, y dado que la altura de la rama derecha es n, la FE de P será 0.

Por último, si la altura de B y C es n, el valor de FE de R es 0. Después de la rotación, la rama B pasa a ser el subárbol derecho de Q, por lo tanto, la FE de Q, dado que la altura de su rama izquierda es n, será 0. La rama C pasa a ser el subárbol izquierdo de P, y dado que la altura de la rama derecha es n, la FE de P será 0.

```
// Rotación doble a derechas
void RDD(Nodo* nodo)
{
  Nodo *Padre = nodo->padre;
  Nodo *P = nodo;
  Nodo *Q = P->izquierdo;
  Nodo *R = Q->derecho;
  Nodo *B = R->izquierdo;
  Nodo *C = R->derecho;

  if(Padre)
    if(Padre->derecho == nodo) Padre->derecho = R;
    else Padre->izquierdo = R;
  else raíz = R;
  // Reconstruir árbol:
  Q->derecho = B;
  P->izquierdo = C;
  R->izquierdo = Q;
  R->derecho = P;
  // Reasignar padres:
  R->padre = Padre;
  P->padre = Q->padre = R;
  if(B) B->padre = Q;
  if(C) C->padre = P;
  // Ajustar valores de FE:
  switch(R->FE) {
     case -1: Q->FE = 0; P->FE = 1; break;
     case 0:  Q->FE = 0; P->FE = 0; break;
     case 1:  Q->FE = -1; P->FE = 0; break;
  }
  R->FE = 0;
}
```

Arboles-B

Características

Los árboles-B son árboles de búsqueda.

La "B" probablemente se debe a que el algoritmo fue desarrollado por "Rudolf Bayer" y "Eduard M. McCreight", que trabajan para la empresa "Boeing" aunque parece que "Karl Unterauer" desarrolló un algoritmo semejante en la misma época.

Lo que si es cierto es que la letra B no significa "binario", ya que:

a. Los árboles-B nunca son binarios.
b. Y tampoco es porque sean árboles de búsqueda, ya que en inglés se denominan B-trees.
c. Tampoco es porque sean balanceados, ya que no suelen serlo.

En cualquier caso, tampoco es demasiado importante el significado de la "B", si es que lo tiene, lo interesante realmente es el algoritmo.

A menudo se usan árboles binarios de búsqueda para ordenar listas de valores, minimizando el número de lecturas, y evitando tener que ordenar dichas listas.

Pero este tipo de árboles tienen varias desventajas:

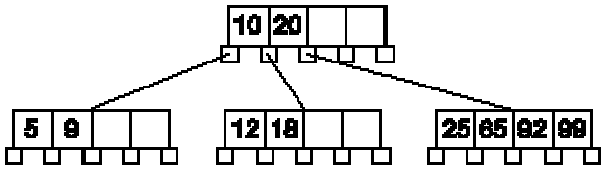
a. Es difícil construir un árbol binario de búsqueda perfectamente equilibrado.
b. El número de consultas en el árbol no equilibrado es impredecible.
c. Y además el número de consultas aumenta rápidamente con el número de registros a ordenar.

Los árboles-B son árboles de búsqueda de m ramas, y cada nodo puede almacenar un máximo de m-1 claves.

Las características que debe cumplir un árbol-B son:

- Un parámetro muy importante en los árboles-B es el ORDEN (m). El orden de un árbol-B es el número máximo de ramas que pueden partir de un nodo.
- Si n es el número de ramas que parten de un nodo de un árbol-b, el nodo contendrá n-1 claves.
- El árbol está ordenado.
- Todos los nodos terminales, (nodos hoja), están en el mismo nivel.
- Todos los nodos intermedios, excepto el raiz, deben tener entre m/2 y m ramas no nulas.
- El máximo número de claves por nodo es m-1.
- El mínimo número de claves por nodo es (m/2)-1.
- La profundidad (h) es el número máximo de consultas para encontrar una clave.

Ejemplo de un árbol-B de ORDEN 4 y de profundidad 2.



Existen varios criterios para dimensionar el ORDEN de un árbol-B.

Normalmente se intenta limitar la profundidad del árbol, de modo que el número máximo de consultas sea pequeño. Cuanto mayor sea el ORDEN, menor será la profundidad.

Cuando se crean árboles-B para indexar archivos de datos en disco, intervienen otras condiciones. En ese caso interesa que el número de accesos a disco sea lo más pequeño posible. Para calcular el ORDEN se usa como dato el tamaño del cluster. Un cluster es el bloque de disco más pequeño que se lee o se escribe en una operación de acceso a disco, su tamaño suele ser distinto según el tamaño de disco y el tipo de formato que tenga, puede variar entre 512 bytes y múltiplos de esa cantidad. El ORDEN se ajusta de modo que el tamaño del nodo sea lo más próximo posible, menor o igual, al tamaño del cluster.

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

- Insertar una clave.
- Eliminar una clave.
- Buscar una clave.

Estructura de un nodo:

Para ilustrar este artículo haremos un programa que indexará una tabla de enteros usando un árbol-B.

Implementaremos dos clases, una para el tratamiento de los nodos y otra para el árbol.

Cada clave estará compuesta por una pareja de valores, uno de esos valores es la propia clave, en nuestro ejemplo será un entero, pero en general será del mismo tipo que el campo por el que ordenaremos la tabla o archivo, el otro campo será una referencia al registro al que corresponda dicha clave. Por ejemplo, si tenemos un archivo con la siguiente estructura de registro:

```
struct registro {
  char nombre[32];
  int edad;
  long telefono;
};
```

Y queremos construir un árbol-B para indicar una tabla de 1000 registros por el campo nombre.

Nuestra clave será una estructura como ésta:

```
struct stclave {
  char nombre[32];
  long registro; // número índice correspondiente a la clave "nombre"
};
```

Si lo que tenemos que ordenar es un archivo, el valor de registro sería la posición del registro de datos asociado a la clave dentro del archivo.

En rigor, un nodo sólo necesita almacenar un máximo de m-1 claves y m punteros, pero para facilitar el manejo de los árboles-B, añadiremos algunos datos extra.

Para el nodo, definiremos la clase bnodo como sigue:

```
class bnodo {
 public:
  bnodo(int nClaves); // Constructor
  ~bnodo();          // Destructor

 private:
  int clavesUsadas;   // Claves usadas en el nodo
  stclave *clave;     // Array de claves del nodo
  bnodo **puntero;    // Array de punteros a bnodo
  bnodo *padre;       // Puntero a nodo padre

 friend class btree;
};
```

Además de un array de claves y de punteros, que es lo que se necesita para definir un nodo según el gráfico que hemos visto antes, hemos añadido un par de datos más que nos facilitarán en manejo del árbol.

Uno es el número de claves usadas en el nodo, ya sabemos que no tienen por qué estar todas usadas. El otro es un puntero al nodo padre, este puntero nos será muy útil para insertar y eliminar claves.

Clase para manejar la estructura del árbol:

```
class btree {
  public:
   btree(int nClv);            // Constructor
   ~btree();                   // Destructor
   long Buscar(int clave);     // Buscar un valor de clave, devuelve la posición en el array
   bool Insertar(stclave clave); // Insertar una clave
   void Borrar(int clave);     // Borrar la clave correspondiente a un valor
   void Mostrar();             // (Depuración) Mostrar el árbol por pantalla

  private:
   stclave *lista;             // Auxiliar para insertar claves
   pbnodo *listapunt;          // Auxiliar para insertar claves
   // Funciones auxiliares internas de la clase:
   void Inserta(stclave clave, pbnodo nodo, pbnodo hijo1, pbnodo hijo2);
   void BorrarClave(pbnodo nodo, int valor);
   void BorrarNodo(pbnodo nodo);
   void PasarClaveDerecha(pbnodo derecha, pbnodo padre, pbnodo nodo, int posClavePadre);
   void PasarClaveIzquierda(pbnodo izquierda, pbnodo padre, pbnodo nodo, int
posClavePadre);
   void FundirNodo(pbnodo izquierda, pbnodo &padre, pbnodo derecha, int posClavePadre);
   void Ver(pbnodo nodo);

   int nClaves;                // Número de claves por nodo
   int nodosMinimos;           // Número de punteros mínimos para cada nodo que no sea hoja
   pbnodo Entrada;             // Puntero a nodo de entrada en el árbol
};
```

Se han declarado varias funciones auxiliares privadas para facilitar la inserción y borrado de nodos. Estas funciones sólo se usarán internamente por la clase.

Algoritmo de Inserción de claves:

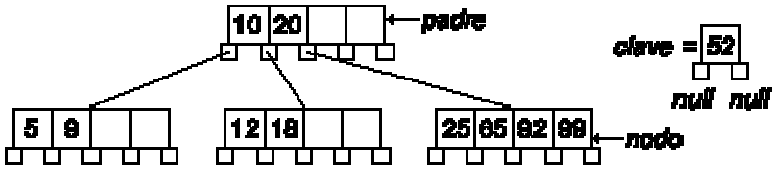
Se trata de un algoritmo iterativo, aunque también puede implementarse recursivamente.

- Buscar la *clave* en el árbol. (Nuestro algoritmo no permite almacenar claves duplicadas).
- Si la *clave* ya está en el árbol, salimos sin hacer nada.
- Buscamos el nodo donde debería añadirse la *clave*. (*nodo*)
- *derecho* = *izquierdo* = NULL
- **Bucle**:
    - Si *nodo* es NULL crear un nodo nuevo y hacer que *Entrada* sea el nuevo nodo.
    - Guardar el valor del padre de *nodo* en *padre*.
    - ¿Hay espacio libre en *nodo* para la nueva *clave*?
        - **SI**:
            - Almacenar la *clave* en *nodo*.
            - Puntero izquierdo = *izquierdo*.
            - Puntero derecho = *derecho*.
            - salir de bucle.
        - **NO**:
            - Crear un nodo *nuevo*
            - Segregar la mitad de las claves menores en *nodo* y la mitad de las mayores en *nuevo*.
            - Promocionar el valor intermedio, es decir insertarlo en el nodo *padre*.
                - *izquierdo* = *nodo*
                - *derecho* = *nuevo*
                - *nodo* = *padre*
                - *clave* = clave intermedia
            - cerrar bucle

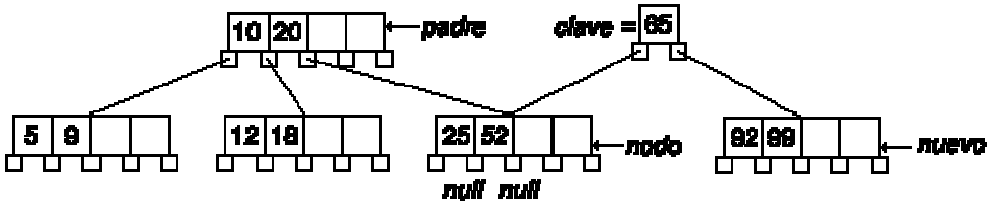
Veremos ahora algunos ejemplos.

Ejemplo con inserción de nodo:

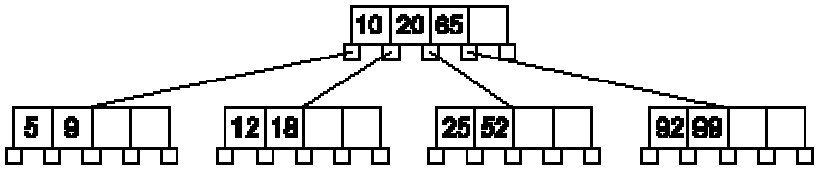
Veamos un ejemplo. Supongamos que queremos insertar la clave 52 en el árbol del ejemplo.



No hay espacio para almacenar la clave en nodo, por lo tanto creamos un nuevo nodo:

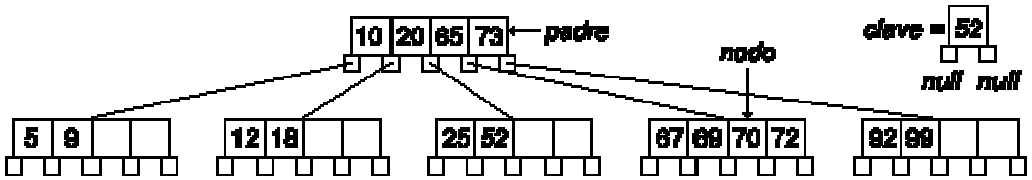


Ahora tenemos que promocionar la clave intermedia, insertándola en el nodo padre. El proceso es recursivo, aunque el algoritmo que hemos diseñado es iterativo.

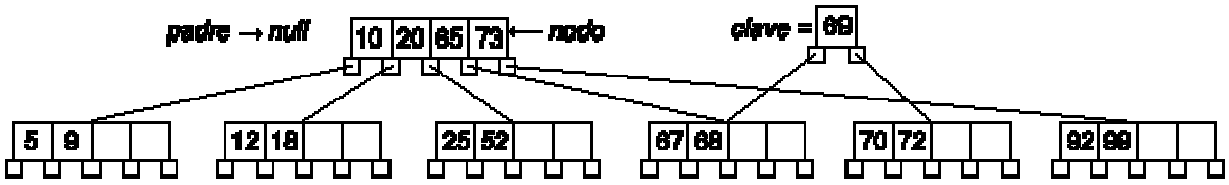


Ejemplo con aumento de altura:

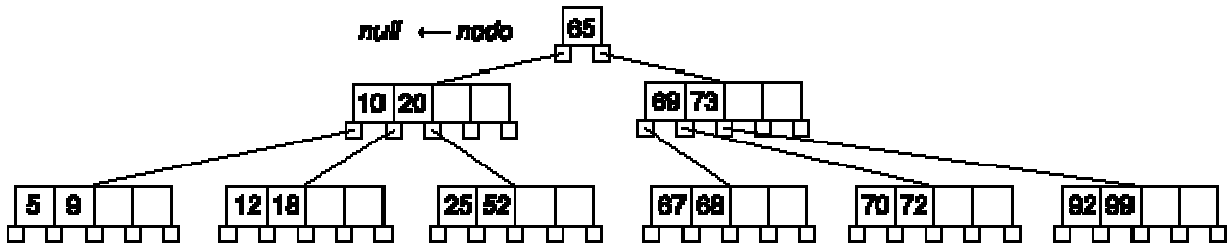
Veamos otro ejemplo que implicará que aumente la altura del árbol. Supongamos que tenemos un árbol con esta estructura, y queremos insertar la clave 52:



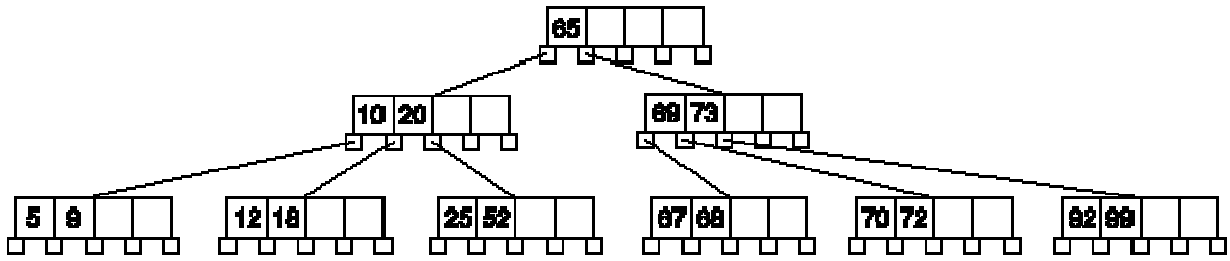
En primer lugar, dividimos *nodo* en dos, repartimos las claves y promocionamos el valor intermedio, el 69:



Ahora estamos de nuevo en la misma situación, el anterior nodo pasa a ser el nodo padre, y el padre de éste es null. Ahora procedemos igual, dividimos *nodo* en dos, separamos las claves, y promocionamos el valor intermedio:



Y otra vez repetimos el proceso, el algoritmo funciona de forma recursiva, pero en este caso *nodo* es null, por lo tanto estamos en un caso diferente, según el algoritmo debemos crear un nuevo nodo, hacer que ese nodo sea el de entrada, insertar la clave y actualizar los punteros:



Algoritmo de Borrado de claves:

Borrar nodos es algo más complicado, como veremos ahora. También se trata de un algoritmo iterativo, aunque puede implementarse recursivamente.

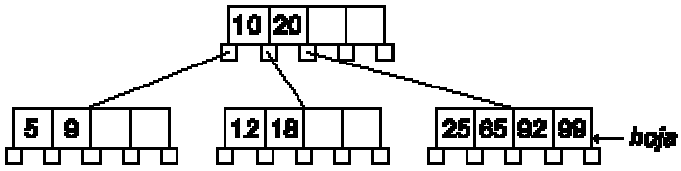
- Buscar el nodo donde está la *clave* que queremos borrar.
- Si la *clave* no está en el árbol, salimos sin hacer nada.
- Buscamos la *posición* que ocupa la *clave* dentro del nodo.
- ¿Se trata de un nodo hoja o terminal?
    - **SI**:
        - *hoja* = *nodo*
    - **NO**:
        - Intercambiar *clave* con la clave siguiente. Se trata de la primera clave del nodo hoja obtenido al seguir la rama izquierda a partir del nodo derecho de la *clave* actual.
        - *hoja* = nodo de la siguiente clave
- Eliminar clave, se consigue rotando las claves siguientes dentro del *nodo* y decrementando el número de claves usadas del *nodo*.
- Si el número de claves usadas es cero y *hoja* es el nodo de *entrada:*
    - **SI**:
        - *Entrada* = NULL
        - Borrar *hoja*
        - Salir.
- Hoja es el nodo de entrada o el número de claves usadas es mayor del número mínimo de claves por nodo:
    - **SI**:
        - Salir.
- **Bucle**:
    - Buscar el puntero que apunta a *hoja* en nodo *padre*: posClavePadre.
    - Localizar los nodos *izquierdo* y *derecho* del nodo *hoja*.
    - Si existe el nodo *derecho* y contiene más claves que el número mínimo.
        - **SI**:
            - Pasar una clave del nodo *derecho* a *padre*, en *posClavePadre* y la clave que había allí al nodo *hoja*.
            - Salir.
    - Si existe el nodo *izquierdo* y contiene más claves que el número mínimo.
        - **SI**:
            - Pasar una clave del nodo *izquierdo* a *padre*, en *posClavePadre* y la clave que había allí al nodo *hoja*.
            - Salir.
    - Si existe el nodo *derecho*.
        - **SI**:
            - Fundir en *hoja* los nodos *hoja* y *derecho* junto con la clave del nodo *padre posClavePadre*. Borrar nodo *derecho*.
            - Si *padre* es el nodo de *entrada* y no tiene claves.
                - **SI**:
                    - *entrada* = *hoja*
                    - borrar nodo *padre*

                - salir
            - **NO**: (Entonces debe existir un nodo *izquierdo*)
                - Fundir en *izquierdo* los nodos *hoja* y *izquierdo* junto con la clave del nodo *padre posClavePadre*. Borrar nodo *hoja*.
                - Si *padre* es el nodo de *entrada* y no tiene claves.
                    - **SI**:
                        - *entrada* = *izquierdo*
                        - borrar nodo *padre*
                        - salir
    - *hoja* = *padre*
    - Si hoja no es NULL, ni hoja es el nodo de entrada ni el número de claves en hoja es menos que el mínimo.
        - **SI**:
            - cerrar **bucle**.
        - **NO**:
            - Salir.

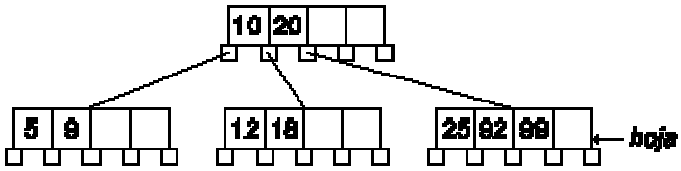
Veremos ahora algunos ejemplos.

Clave en nodo hoja y número de claves mayor que el mínimo:

Borraremos la clave 65. Es uno de los casos más sencillos, la clave está en un nodo hoja, y el número de claves del nodo después de eliminarla es mayor del mínimo.

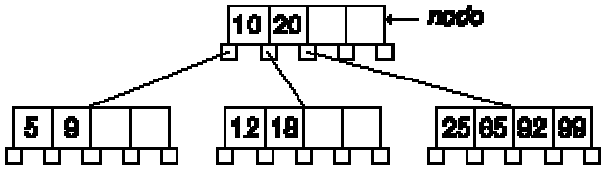


Tan sólo tendremos que eliminar la clave:

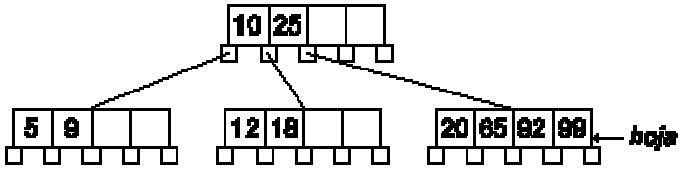


Clave en nodo intermedio:

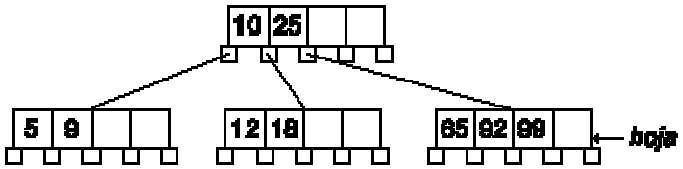
Eliminaremos ahora la clave 20.



Según el algoritmo debemos intercambiar la clave por el siguiente valor, es decir, el 25. En general será la primera clave del nodo más a la izquierda a de la rama derecha de la clave a borrar.

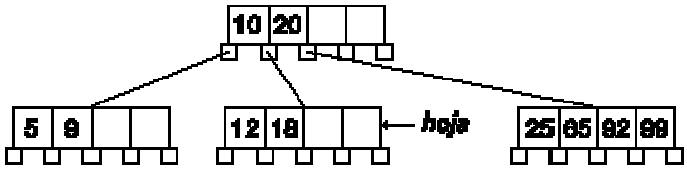


Ahora estamos en el mismo caso que antes, sólo hay que borrar la clave 20 del nodo hoja:

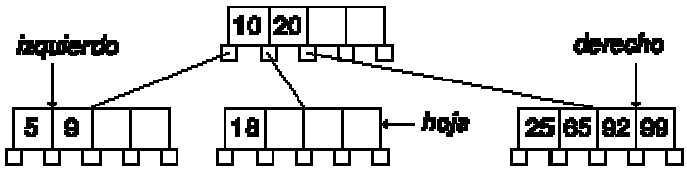


Borrar clave cuando el número de claves es menor que el mínimo:

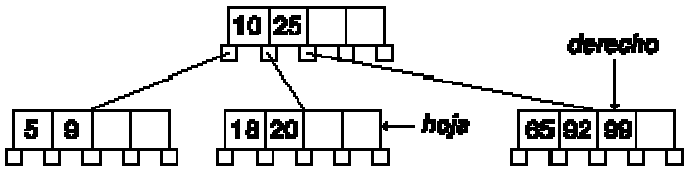
Eliminaremos el nodo 12 del árbol:



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.

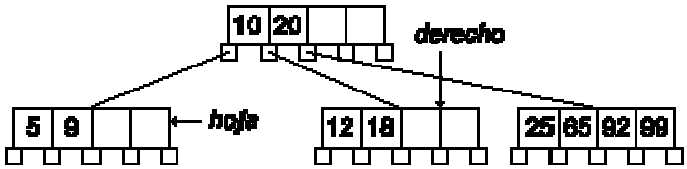


Según el algoritmo, buscaremos los nodos izquierdo y derecho. Si el número de claves en el nodo derecho es mayor que el mínimo, pasaremos una clave del nodo derecho al padre y de éste al nodo hoja. Si no, lo intentaremos con el izquierdo. En este caso nos vale el nodo derecho:

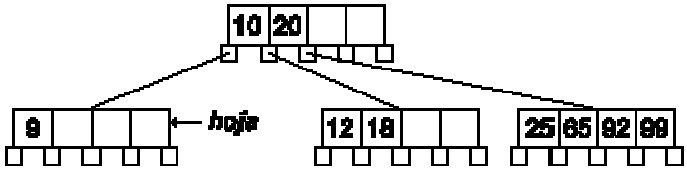


Borrar clave cuando implica borrar nodos:

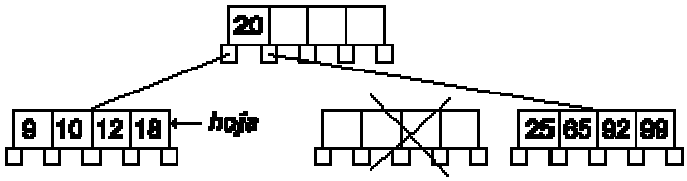
Vamos a ver el caso en que tanto el nodo derecho como el izquierdo no tienen claves suficientes para transferir una al nodo *hoja*. Borraremos la clave 5:



La primera parte es igual que el caso anterior, eliminamos la clave 5. Pero ahora el nodo *derecho* tiene el número mínimo de claves, y el *izquierdo* no existe.



Según el algoritmo, debemos fundir en el nodo *hoja*, las claves del nodo *hoja*, la clave del *padre* y las del nodo *derecho*. Y después eliminar el nodo derecho.

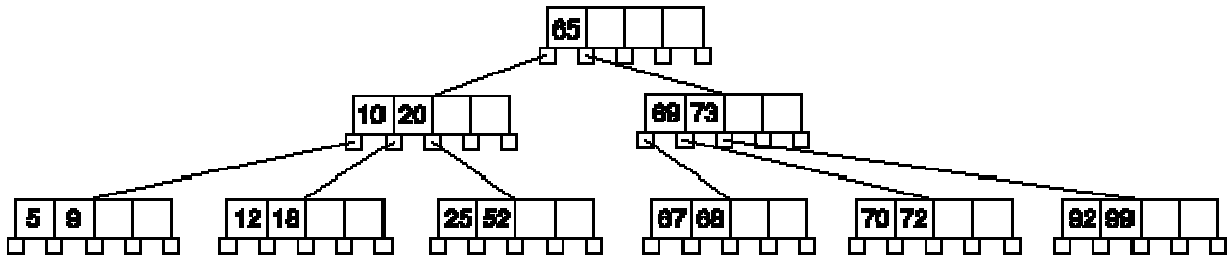


La cosa no termina ahí, ahora debemos comprobar el nodo padre, ya que también ha podido quedar con menos claves que el mínimo exigido. En este caso sucede eso, pero es un caso especial, ya que ese nodo es el de entrada y puede contener menos claves que número mínimo.

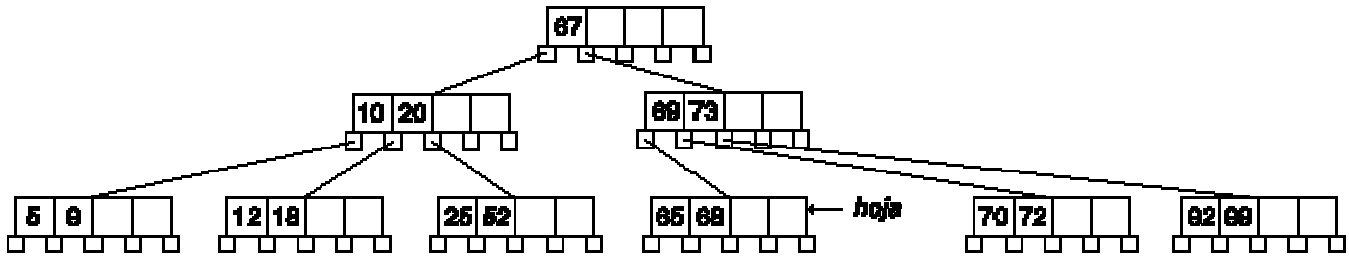
Caso de reducción de altura:

Veamos cómo sería el proceso en un caso general, que implica la reducción de altura del árbol.

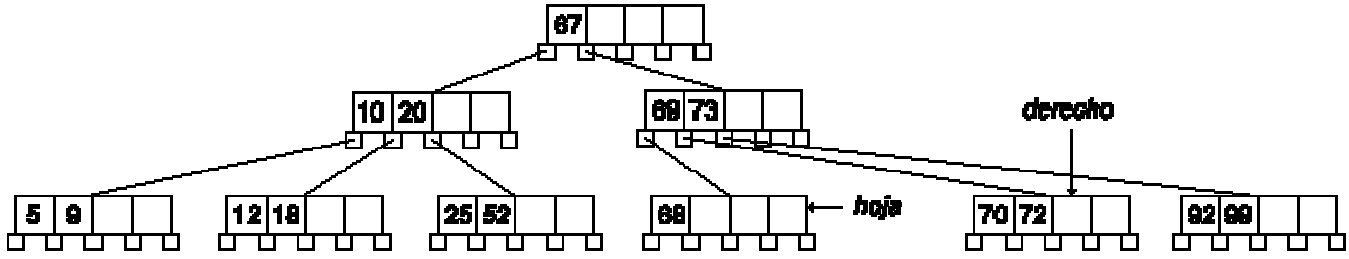
Borraremos la clave 65 de este árbol:



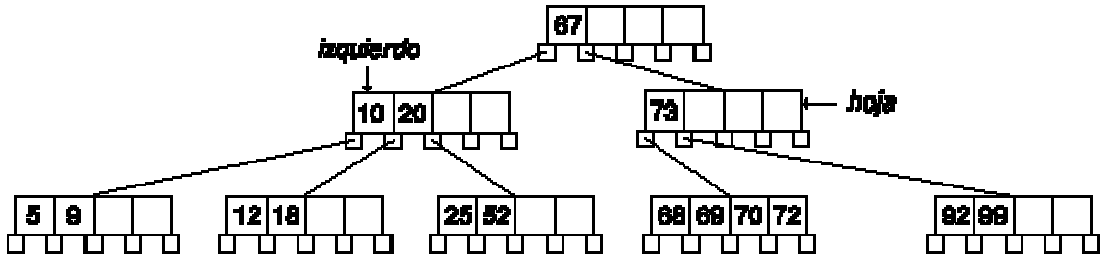
El primer paso es intercambiar la clave 65 por la siguiente, la 67. Y hacer que el nodo al que pertenece sea el nodo *hoja*.



A continuación eliminamos la clave 65 del nodo *hoja*, y comprobamos que el nodo tiene menos claves del mínimo. Buscamos el nodo *derecho* e *izquierdo*.



El nodo *izquierdo* no existe, y el *derecho* tiene justo el número mínimo de claves, por lo tanto tenemos que fusionar *hoja* con *derecho* y con una clave 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 los nodos *derecho* e *izquierdo*. Y de nuevo comprobamos que el nodo *derecho* no existe y que el *izquierdo* tiene justo el número mínimo de claves.

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 el de *entrada* y ha quedado vacío, lo eliminaremos y el nodo de *entrada* será el *izquierdo*: