

ARBOLES B

El problema original comienza con la necesidad de mantener índices en almacenamiento externo para acceso a bases de datos, es decir, con el grave problema de la lentitud de estos dispositivos se pretende aprovechar la gran capacidad de almacenamiento para mantener una cantidad de información muy alta organizada de forma que el acceso a una clave sea lo más rápido posible.

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

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

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:

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

Para evitar estos inconvenientes se usan árboles-B, sobre todo cuando se ordenan archivos, donde se ha convertido en el sistema de indexación más utilizado.

DEFINICIÓN.

Los B-árboles son árboles cuyos nodos pueden tener un número múltiple de hijos tal como muestra el esquema de uno de ellos en la figura 1.

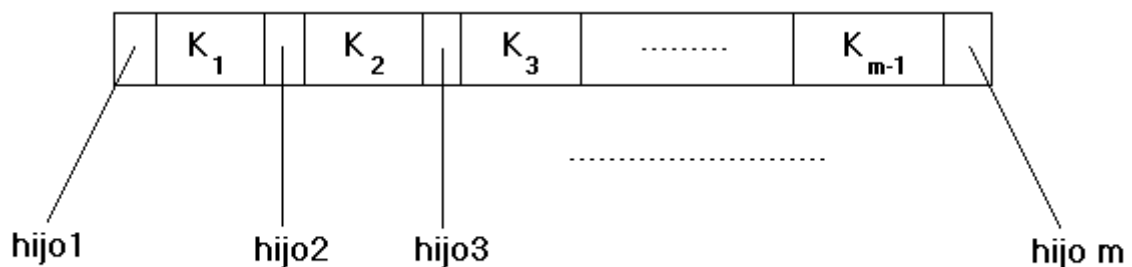


Figura 1: Esquema de un nodo de un árbol B de orden m.

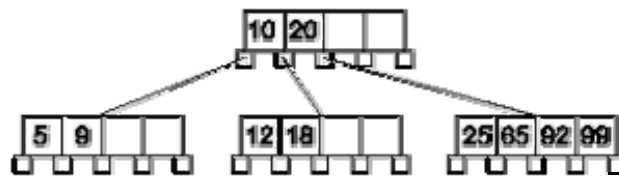
Como se puede observar en la figura 1, un B-árbol se dice que es de orden m si sus nodos pueden contener hasta un máximo de m hijos. En la literatura también aparece que si un árbol es de orden m significa que el mínimo número de hijos que puede tener es $m + 1$ (m claves).

El conjunto de claves que se sitúan en un nodo cumplen la condición:

$$K_1 < K_2 < \dots < K_{m-1}$$

De forma que los elementos que cuelgan del primer hijo tienen una clave con valor menor que K_1 , los que cuelgan del segundo tienen una clave con valor mayor que K_1 y menor que K_2 , etc. Obviamente, los que cuelgan del último hijo tienen una clave con valor mayor que la última clave (hay que tener en cuenta que el nodo puede tener menos de m hijos y por consiguiente menos de $m-1$ claves).

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



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 raíz, 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.

BÚSQUEDA EN UN B-ÁRBOL.

Localizar una clave en un B-árbol es una operación simple pues consiste en situarse en el nodo raíz del árbol, si la clave se encuentra ahí hemos terminado y si no es así seleccionamos de entre los hijos el que se encuentra entre dos valores de clave que son menor y mayor que la buscada respectivamente y repetimos el proceso hasta que la encontremos. En caso de

que se llegue a una hoja y no podamos proseguir la búsqueda la clave no se encuentra en el árbol. En definitiva, los pasos a seguir son los siguientes:

1. Seleccionar como nodo actual la raíz del árbol.
2. Comprobar si la clave se encuentra en el nodo actual:
 1. Si la clave está, fin.
 2. Si la clave no está:
 - Si estamos en una hoja, no se encuentra la clave. Fin.
 - Si no estamos en una hoja, hacer nodo actual igual al hijo que corresponde según el valor de la clave a buscar y los valores de las claves del nodo actual (i buscamos la clave K en un nodo con n claves: el hijo izquierdo si $K < K_1$, el hijo derecho si $K > K_n$ y el hijo i -ésimo si $K_i < K < K_{i+1}$) y volver al segundo paso.

INSERCIÓN EN UN B-ÁRBOL.

Para insertar una nueva clave usaremos un algoritmo que consiste en dos pasos recursivos:

1. Buscamos la hoja donde debiéramos encontrar el valor de la clave de una forma totalmente paralela a la búsqueda de ésta tal como comentábamos en la sección anterior (si en esta búsqueda encontramos en algún lugar del árbol la clave a insertar, el algoritmo no debe hacer nada más). Si la clave no se encuentra en el árbol habremos llegado a una hoja que es justamente el lugar donde debemos realizar esa inserción.
2. Situados en un nodo donde realizar la inserción si no está completo, es decir, si el número de claves que existen es menor que el orden menos 1 del árbol, el elemento puede ser insertado y el algoritmo termina. En caso de que el nodo esté completo insertamos la clave en su posición y puesto que no caben en un único nodo dividimos en dos nuevos nodos conteniendo cada uno de ellos la mitad de las claves y tomando una de éstas para insertarla en el padre (se usará la mediana). Si el padre está también completo, habrá que repetir el proceso hasta llegar a la raíz. En caso de que la raíz esté completa, la altura del árbol aumenta en uno creando un nuevo nodo raíz con una única clave.

En la figura 2 podemos observar el efecto de insertar una nueva clave en un nodo que está lleno.

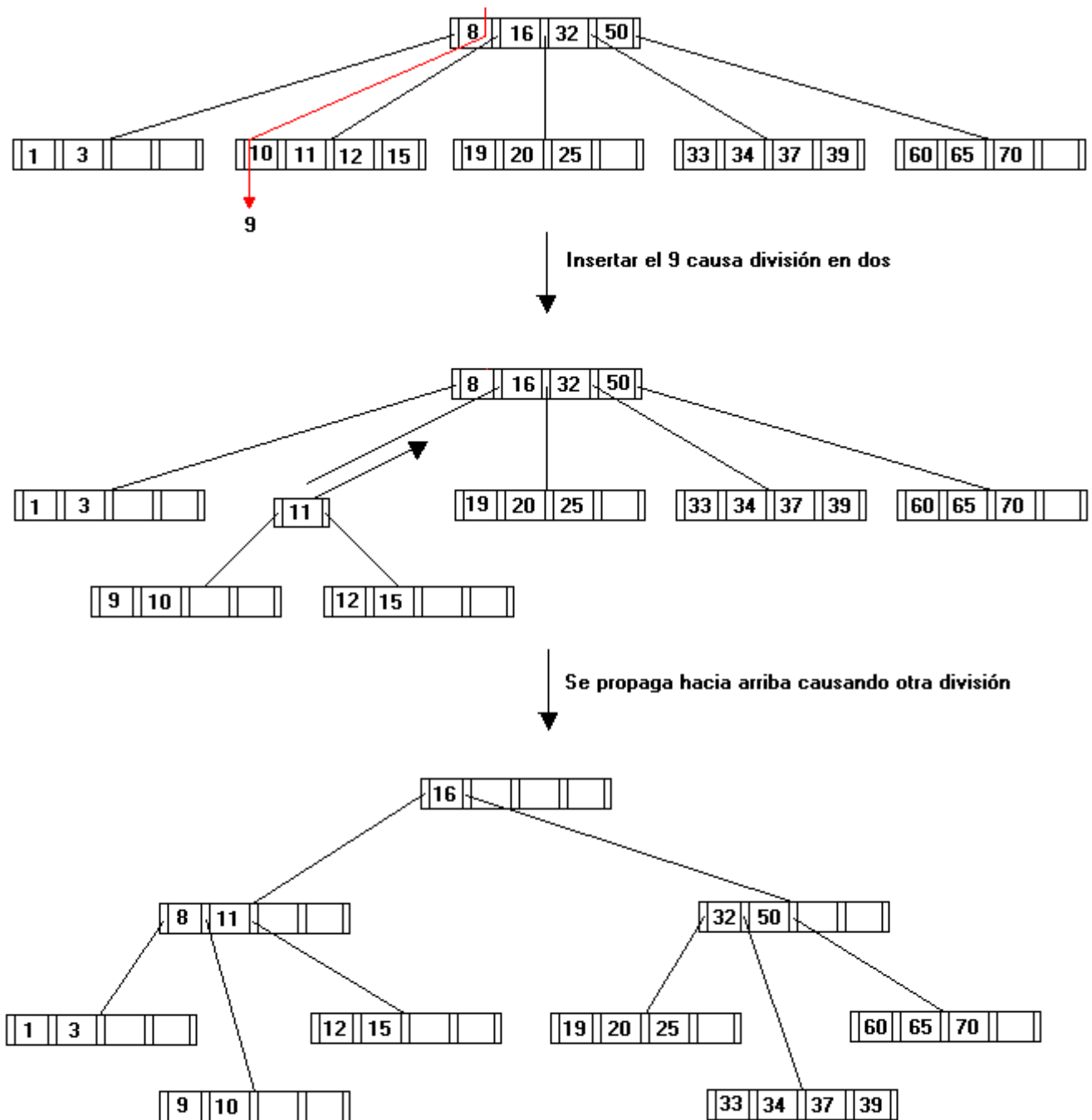


Figura 2: Inserción de un nuevo elemento en un B-árbol

BORRADO EN UN B-ÁRBOL.

La idea para realizar el borrado de una clave es similar a la inserción teniendo en cuenta que ahora, en lugar de divisiones, realizamos uniones. Existe un problema añadido, las claves a borrar pueden aparecer en cualquier lugar del árbol y por consiguiente no coincide con el caso de la inserción en la que siempre comenzamos desde una hoja y propagamos hacia arriba. La solución a esto es inmediata pues cuando borramos una clave que está en un nodo interior, lo primero que realizamos es un intercambio de este valor con el

inmediato sucesor en el árbol, es decir, el hijo más a la izquierda del hijo derecho de esa clave.

Las operaciones a realizar para poder llevar a cabo el borrado son por tanto:

1. Redistribución: la utilizaremos en el caso en que al borrar una clave el nodo se queda con un número menor que el mínimo y uno de los hermanos adyacentes tiene al menos uno más que ese mínimo, es decir, redistribuyendo podemos solucionar el problema.
2. Unión: la utilizaremos en el caso de que no sea posible la redistribución y por tanto sólo será posible unir los nodos junto con la clave que los separa y se encuentra en el padre.

En definitiva, el algoritmo nos queda como sigue:

1. Localizar el nodo donde se encuentra la clave..
2. Si el nodo localizado no es una hoja, intercambiar el valor de la clave localizada con el valor de la clave más a la izquierda del hijo a la derecha. En definitiva colocar la clave a borrar en una hoja. Hacemos nodo actual igual a esa hoja.
3. Borrar la clave.
4. Si el nodo actual contiene al menos el mínimo de claves como para seguir siendo un B-árbol, fin.
5. Si el nodo actual tiene un número menor que el mínimo:
 1. Si un hermano tiene más del mínimo de claves, redistribución y fin.
 2. Si ninguno de los hermanos tiene más del mínimo, unión de dos nodos junto con la clave del padre y vuelta al paso 4 para propagar el borrado de dicha clave (ahora en el padre).

PRIMITIVAS DE UN B-ÁRBOL.

```
AB Crear0(int ne)
{
    AB raiz;

    raiz = (AB)malloc(sizeof(struct AB));
    if (raiz == NULL)
        error("Memoria Insuficiente.");
    raiz->n_etiquetas = ne;
    for (int i=1; i<=(ne+1); i++) {
        raiz->hijos[i] = NULO;
    }
    return(raiz);
}
```

```
AB Crear(int ne, int eti[])
{
    AB raiz;

    raiz = (AB)malloc(sizeof(struct AB));
    if (raiz == NULL)
        error("Memoria Insuficiente.");
```

```

    raiz->n_etiquetas = ne;
    for (int i=1; i<=(ne+1); i++) {
        raiz->hijos[i] = NULO;
    }
    for (int i=1; i<=lenght(eti[]); i++) {
        raiz->etiquetas[i] = eti[i];
    }
    return(raiz);
}

int Buscar(int eti, int *nod, int *pos)
{
    int i,l;

    l = lenght(nod->etiquetas[]);
    for(i=0; i<nod->etiquetas[i]; i++)
        ;
    *pos = i;
    if(*posetiquetas[*pos])
        return 1;
    else;
        return 0;
}

int BuscarNodo(int eti, int *nod, int *pos)
{
    int i=0, enc;

    enc = Buscar(eti,&nod,&pos);
    if (enc == 1)
        return 1;
    do {
        if (etietiquetas[i] && nod->hijos[i]!=NULO)
            enc = BuscarNodo(eti,&nod->hijos[i],&pos);
        else
            if ((etietiquetas[i+1]||nod->etiquetas[i+1]==-1)&&nod->hijos[i+1]!=-1)
                enc = BuscarNodo(eti,&nod->hijos[i+1],&pos);
        i++;
    } while (ietiquetas[]) && enc==0);
    return (enc);
}

```