Archivos Indexados – Arboles B, B+ y B*

Consideraciones:

- La memoria RAM generalmente no es suficiente para contener todo el índice
- Los accesos a disco son muy lentos y representan la mayor parte del tiempo necesario para hacer una búsqueda por ser necesarios muchos seeks.
- Es muy costoso mantener actualizado el índice en disco
- Pueden identificarse en general dos tipos de acceso. El acceso por referencia a un registro en particular o el acceso para recorrer secuencialmente un conjunto de varios registros. Para cada tipo de acceso, la estructura óptima para el índice será diferente.

Para disminuir la cantidad de accesos a disco se utilizan estructuras de árbol, de las cuales se mantiene en memoria la parte que se está utilizando y el resto se conserva en disco.

LA idea es reducir al mínimo posible la cantidad de accesos en promedio que deben hacerse, no para una lectura en particular sino para todo el tiempo de uso de la estructura. Así, si se diseña una estructura de índice que será accedida principalmente para accesos por referencia, será preferible una estructura que reduzca la cantidad de seeks necesarios para cada búsqueda puntual. En cambio, si el tipo de búsqueda predominante será para posicionarse en un registro dado y recorrer secuencialmente a partir del mismo, será mejor una estructura que aunque requiera una mayor cantidad de seeks para un acceso por referencia (el primero), requiera pocos accesos para leer secuencialmente el resto de los datos.

Ejemplos de cantidad promedio de accesos a disco

Búsqueda binaria:

$log_2 [M/N] - 1$

M: cantidad de registros lógicos
N: Factor de bloqueo

Mantener los datos en orden para hacer las búsquedas es muy costoso.

Arbol binario balanceado:

$Piso(log_2 N) + 1$

En realidad nada garantiza que un árbol binario esté balanceado, podría degenerar a una lista enlazada si los datos se entran en orden El uso de un árbol binario evita el costo de mantener los datos en orden.

Arbol AVL paginado:

$log_{k+1}(N+1)$

k: cantidad de claves por página
N: cantidad total de claves

La reorganización de los nodos para mantener el balanceo es relativamente poco costosa pero por ser árboles binarios, su profundidad tiende a ser muy grande.

Arboles B

Son árboles que tienden a ser anchos y poco profundos.
La cantidad de seeks necesaria para un acceso por referencia a una clave en particular es baja.

$1+log_{Techo(m/2)}\ [(N+1)/2]$

Definición:

Sea m un número entero llamado Orden del árbol B, un árbol B cumple con las siguientes condiciones:

1. Todos los nodos pueden tener un máximo de *m* descendientes.
2. Cada clave en un nodo está asociada siempre con dos punteros. Uno apunta al subárbol que contiene las claves de valor < y el otro al subárbol que contiene las claves de valor >.
3. Todos los nodos que no sean hoja o raiz deben tener como mínimo Techo(*m*/2) descendientes
4. Si un nodo tiene k descendientes, tiene siempre k-1 claves.
5. La raiz puede tener como mínimo 2 descendientes, a menos que sea el único nodo en el árbol. En ese caso no tendrá descendientes.
6. Todas las ramas tienen igual profundidad.
7. Un nodo tiene un mínimo de (Techo(*m*/2) –1) claves y un máximo de m-1 claves.

Cómo se deben calcular los parámetros del árbol: Ejemplo si *m=6*

a) Cantidad máxima de descendientes que puede tener un nodo (punto *1*) = *m* = **6**
b) Cantidad mínima de descendientes que puede tener un nodo (punto *3*) = Techo(m/2)= **3**
c) Cantidad máxima de claves que puede tener un nodo (punto *7*) = m-1 = **5**
d) Cantidad mínima de claves que puede tener un nodo (punto *7*) = (Techo(*m*/2) –1) = **2**

Luego de cada alta o baja el árbol debe seguir cumpliendo con estas restricciones.
Las claves dentro de un nodo están ordenadas y como los nodos se suben a memoria para ser usados, la búsqueda de claves dentro de cada nodo en memoria puede hacerse de forma secuencial.

Underflow: nombre que se le dá a la situación en la cuál un nodo no contiene la cantidad mínima de claves necesarias.
Overflow: Situación en la cuál un nodo excede la cantidad máxima de claves.

Búsqueda:

Se busca secuencialmente la clave en el nodo raiz. SI se encuentra se devuelve. Si se llega a una clave de valor mayor a la buscada se lee el nodo apuntado por el puntero izquierdo de esa clave, si no (se llega al final del nodo) se lee el nodo apuntado por el último puntero del nodo actual.

Se continúa buscando y leyendo nodos hasta que se llega al nivel de hojas. Si en este punto se llega al final del nodo sin hallar la clave o se encuentra una clave mayor a la buscada, entonces no existe.

Inserción:

Para insertar una nueva clave K en el árbol B, se utiliza en primer lugar un algoritmo similar al de búsqueda para determinar la posición en la que se debe insertar la nueva clave.

Hay dos casos a tener en cuenta

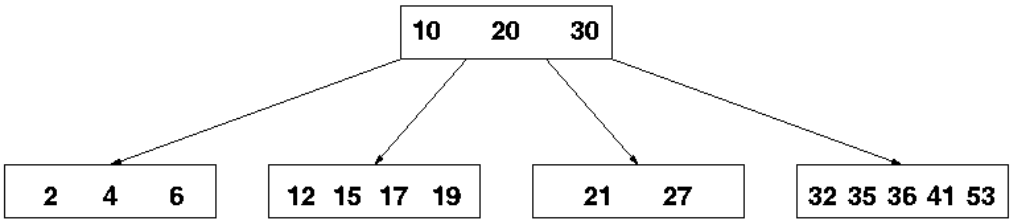
Inserción sin overflow: Si el nodo en el cual debe hacerse la inserción no está lleno, se agrega la clave en la posición correspondiente dentro del nodo y listo.

Inserción con overflow: Si el nodo está lleno (contiene m claves), estamos ante un caso de overflow. En este caso se hace un split.

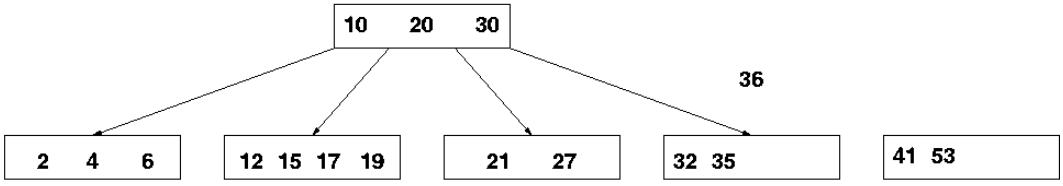
Split: consiste en tomar la clave que se encuentra en la posición central del nodo lleno (llamémosla X) y sacarla del mismo. Luego se procede a generar otro nodo y a pasar al mismo todas las claves mayores que X. Se inserta K en el nodo que corresponda y luego se inserta X en el nodo que antes era su padre (a esto se lo llama Promover a X). En este momento podría ocurrir que dicho nodo también esté lleno y se deba hacer un split del mismo. Esta situación podría propagarse hasta la raiz.

Ejemplo de inserción con overflow seguido de split en un árbol B de orden *m=6*

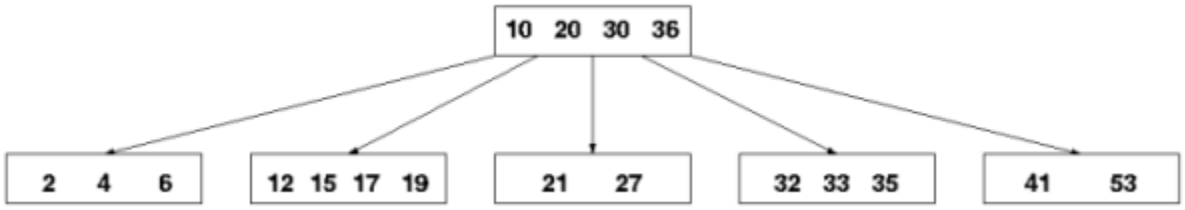
Se tiene el siguiente árbol y se desea insertar la clave *33*



En primer lugar se busca la posición el la cual se va a insertar. Es la última hoja de la derecha, entre el 32 y el 35. Como ese nodo está lleno, estamos ante una condición de overflow. Se selecciona la clave del medio para ser promovida (36), se genera un nuevo nodo vacío y se pasan al mismo las claves mayores a 36.



Por último se promueve la clave al nodo superior que en este caso es la raiz. Si este nodo también estuviera lleno debería hacerse un split.

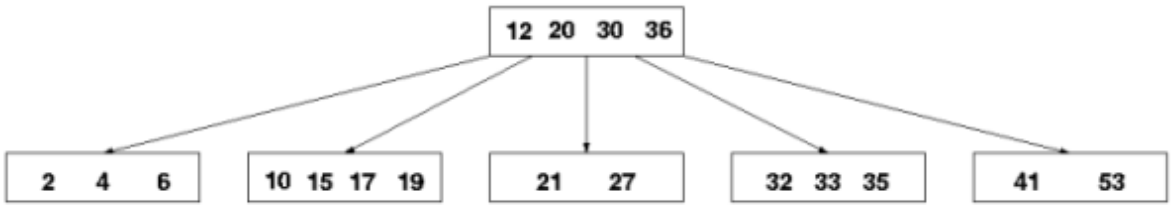


Eliminación:

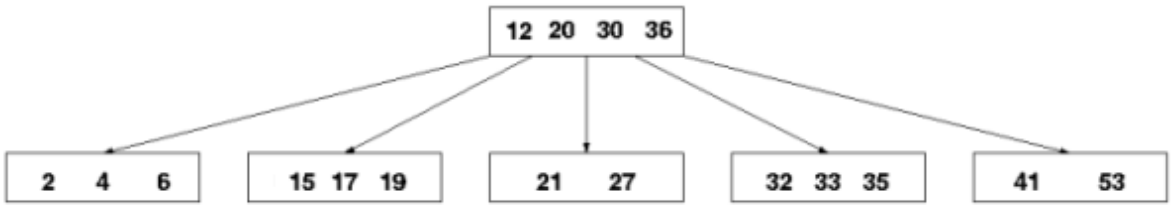
En primer lugar se busca la clave que se debe eliminar y se la borra.

Debido a que es muy complicado reorganizar el árbol si se elimina una clave que no está en una hoja, sólo se borran claves que se encuentran en una hoja . Si se desea eliminar una clave que no cumple está condición, se intercambia con la clave inmediatemente mayor que se encuentre en el árbol, la cuál se garantiza que estará en un nodo hoja (esta particularidad del árbol B deja claro que no resulta bueno para recorridas secuenciales). Un vez hecho el intercambio se procede a eliminar la clave de la hoja, lo cual puede producir cualquiera de los casos de eliminación que se explican a continuación. Cuando se debe realizar este procedimiento, se dice que es un caso de ***Eliminación con intercambio.***

Ejemplo: Si en el árbol que quedó luego del split en el ejemplo anterior se desea eliminar la clave 10, se intercambia el 10 con la menor de las claves del árbol que sea mayor a 10 que en este caso es 12.

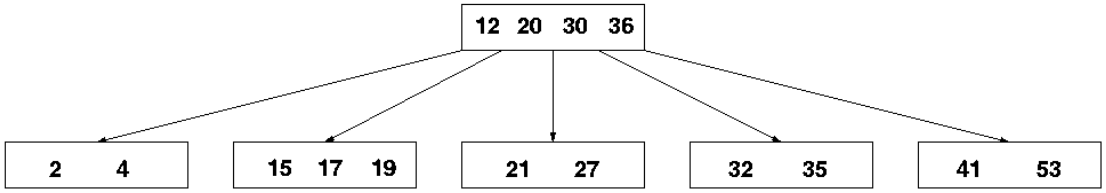


Una vez hecho el intercambio, se elimina la clave 10 de la hoja (es un caso de eliminación sin underflow).



Eliminación sin underflow: Si luego de eliminar la clave el nodo aún cumple con la cantidad de claves mínima, no se requiere ninguna acción mas.

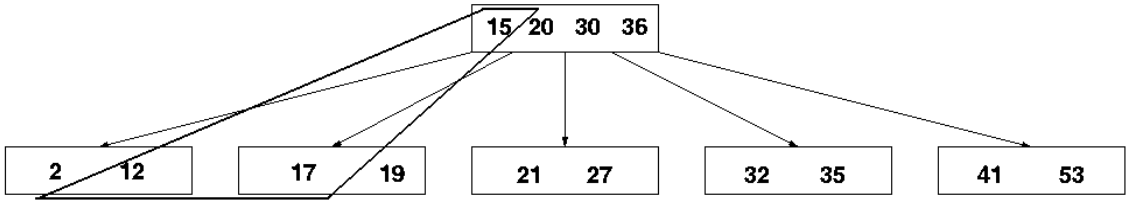
Ejemplo: Eliminación de las clave 6 y 33.



Eliminación con underflow: Si al eliminar una clave de un nodo, la cantidad de claves que quedan en este es menor al mínimo de descendientes, hay dos formas en que se puede resolver. Una es la *redistribución*, la cual es preferible porque sólo afecta localmente al árbol y no propaga la condición de underflow a niveles superiores. En casos en que no puede aplicarse la *redistribución*, se utiliza la *concatenación* que es la operación inversa del split y es mas costosa que la redistribución porque requiere eliminar nodos del árbol y puede propagar el underflow hacia arriba por el mismo e incluso hacer que la profundidad del árbol se reduzca.

Redistribución: Si al eliminar una clave de un nodo, alguno de los nodos vecinos de igual nivel (*siblings*) contiene mas claves de lo que el mínimo indica (podría decirse que "le sobran" claves), se redistribuyen las claves de forma que ninguno qude en underflow.

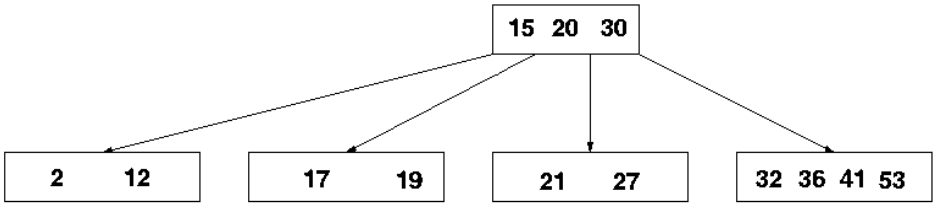
Ejemplo: Se elimina la clave 4 del árbol tal como quedó luego del último ejemplo. Al hacer esto, el nodo queda en underflow. Como el sibling de la derecha tiene una clave mas del mínimo, es posible utilizar este nodo para hacer una redistribución. El árbol resultante será:



Es muy importante tener en cuenta que al hacerse una redistribución, no se pasan claves de un sibling a otro sino que se hace una rotación que involucra a la clave que está en el padre y que actua como separador. Si esto no se hiciera y se pasaran claves de un sibling a otro, el separador dejaría de serlo y el árbol sería inválido.

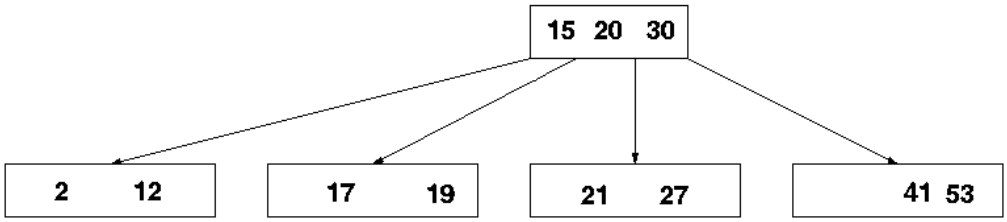
Concatenación: Si al eliminar una clave este queda en underflow y no existe ningún sibling con el cuál se pueda redistribuir, será necesario *concatenar* este nodo con alguno de sus siblings y con el separador que está en su nodo padre.

Ejemplo de concatenación sin propagación: si se desea borrar la clave 35 del árbol tal cual quedó luegodel ejemplo anterior, el nodo queda en underflow (queda sólo la clave 32) y no puede hacerse redistribución con ninguno de sus siblings puesto que ambos tienen el mínimo de claves posible. Se toman las claves 32 (la que quedó en el nodo en underflow), la 36 (la que actuaba de separador) y 41 y 53 (las del sibling) y se quedan todas en el mismo nodo. El resultado de la concatenación es :

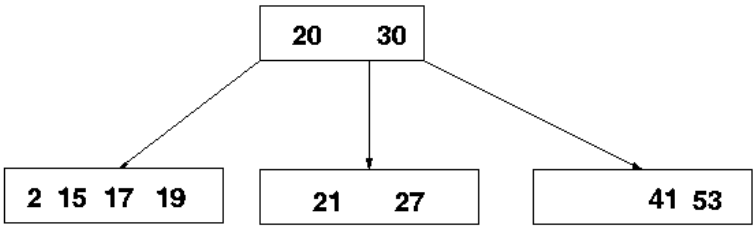


Es importante advertir que la cantidad de claves en el padre de los nodos que se concatenaron ha disminuído. Si al disminuir su cantidad de claves, este nodo quedara a su vez en underflow, estaríamos en un casod e propagación de la condición de underflow. El underflow de este nodo superior se intentará resolver en primer lugar usando redistribución. Si esto no fuera posible se usará nuevamente la concatenación, lo cual podría volver a propagar el underflow. Si el caso de underflow se propaga hasta la raiz, de forma que la raiz misma quedara en underflow, se disminuye en un nivel la profundidad del árbol, puesto que el contenido de la raiz serí absorbido por la concatenación de sus hijos.

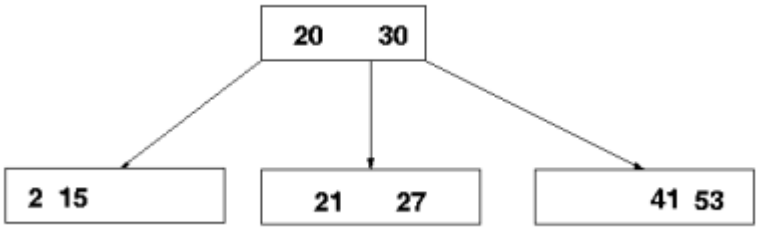
Ejemplo: Si en el árbol que quedó del ejemplo anterior se eliminan las claves 32 y 36, no se produce underflow y el árbol resultante es



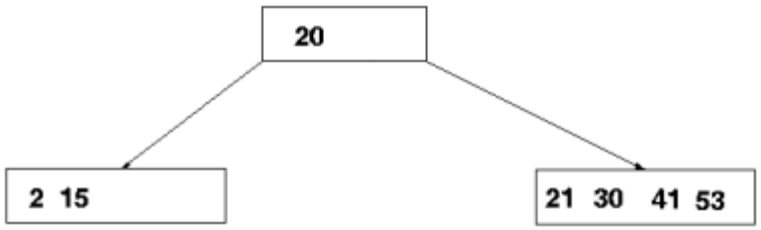
Si ahora se elimina el 12, se produce una concatenación



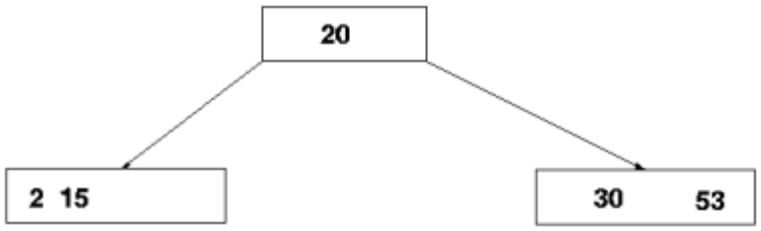
Las claves 17 y 19 pueden eliminarse sin problemas.



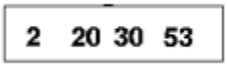
Al eliminar el 27 se produce una concatenación que deja una sola clave en el nodo raiz. Como la raiz puede tener una sola clave, el árbol queda:



La eliminación de las claves 21 y 41 se hace sin problemas



La eliminación de la clave 15 no puede resolverse por medio de una redistribución por lo que es necesaria una concatenación. Como el nodo raiz tiene una sola clave, esta es absorbida por la concatenación de sus dos hijos y la profundidad del árbol se reduce.



Arboles B*

A diferencia de un árbol B un B* hace redistribución no solo durante las bajas sino también durante las inserciones. Si durante una inserción se produce un overflow y hay algún sibling que no esté lleno, hace una redistribución. De esta forma se posterga el split hasta cuando es realmente imprescindible hacerlo. Cuando esto ocurre es porque no existe un sibling que no esté lleno y esta característica es la que lleva a la segunda diferencia de un árbol B* con un B. Un árbol B divide un nodo lleno en dos nodos que quedan con la mitad de claves ocupadas c/u, en cambio un B* basándose en que siempre que hace falta hacer un split seguro que hay por lo menos dos siblings llenos, toma los dos siblings y los divide en tres nodos que estarán 2/3 llenos lo cual lleva a un mejor aprovechamiento del espacio.
Estas dos características hacen que un árbol B* deba cumplir las siguientes condiciones

1. Todos los nodos pueden tener un máximo de *m* descendientes.

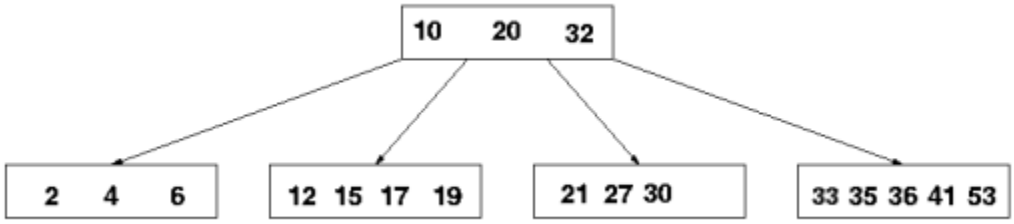
2. Cada clave en un nodo está asociada siempre con dos punteros. Uno apunta al subárbol que contiene las claves de valor < y el otro al subárbol que contiene las claves de valor >.
3. Todos los nodos que no sean hoja o raiz deben tener como mínimo Piso[(2*m-1)*/3] descendientes
4. Si un nodo tiene k descendientes, tiene siempre k-1 claves.
5. La raiz puede tener como mínimo 2 descendientes, a menos que sea el único nodo en el árbol. En ese caso no tendrá descendientes.
6. Todas las ramas tienen igual profundidad.
7. Un nodo tiene un mínimo de Piso[(2*m-1)*/3] claves y un máximo de m-1 claves.

En caso de que se produzca un overflow en el nodo raiz (que no tiene siblings) no hay posibilidad de hacer split de 2 a 3. Las dos alternativas para manejar este caso son

*Que la raiz sea mas grande que el resto de los nodos. De esta forma al hacer el split puede generar dos páginas que estén 2/3 llenas.

*La otra alternativa es manejar el nodo raiz como si fuera la de un árbol B.

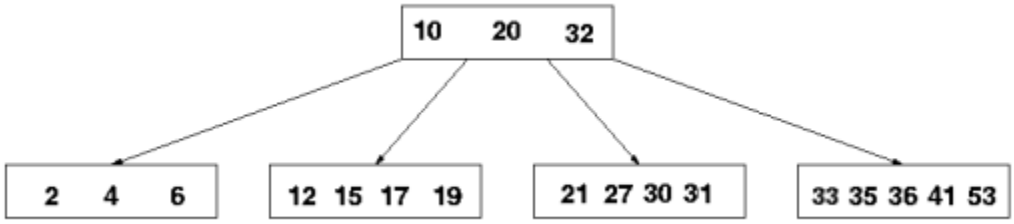
Tomemos como ejemplo el siguiente árbol B* de Orden m=6.



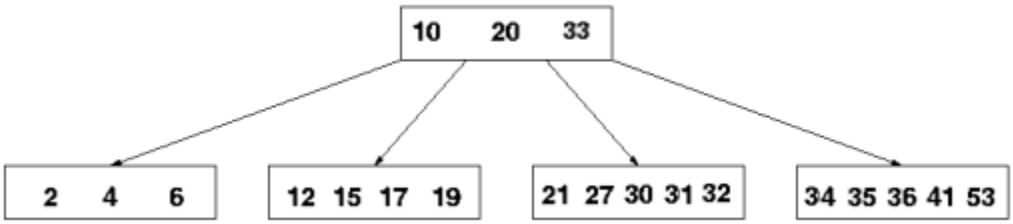
Si m= 6

Cantidad mínima de sucesores por nodo= Piso((2*6-1)/3)= **3**
Cantidad máxima de sucesores por nodo=**6**
Cantidad mínima de claves por nodo=**2**
Cantidad máxima de claves por nodo=**5**

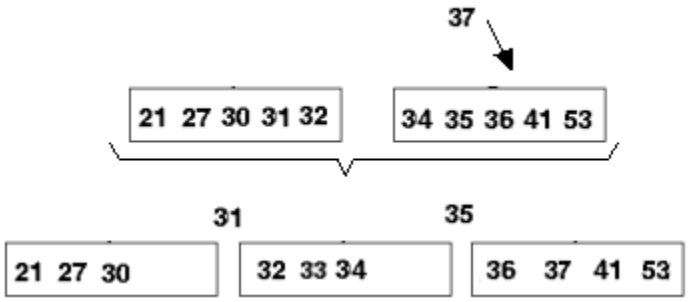
Se inserta la clave 31 sin que se produzca redistribución



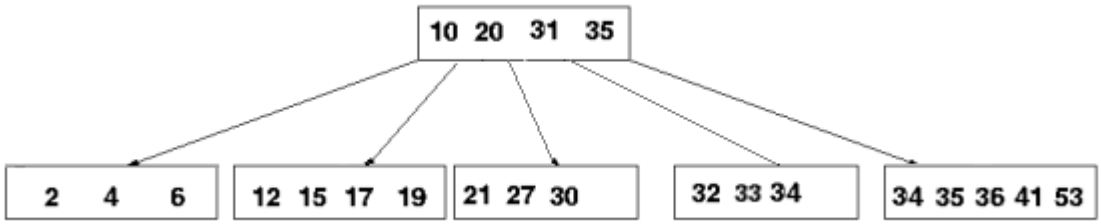
Se inserta la clave 34 y se produce una redistribución



Si se intenta insertar la clave 37 y ya no hay posibilidad de redistribución. En este caso se toman ambos siblings y se hace un split de dos a tres.



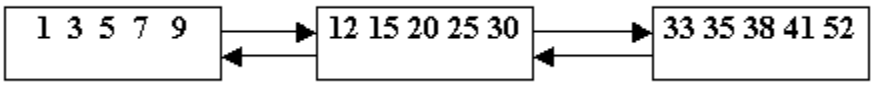
Cómo se vé en el ejemplo, las claves 31 y 35 se promueven al padre con lo cual el árbol resultante es



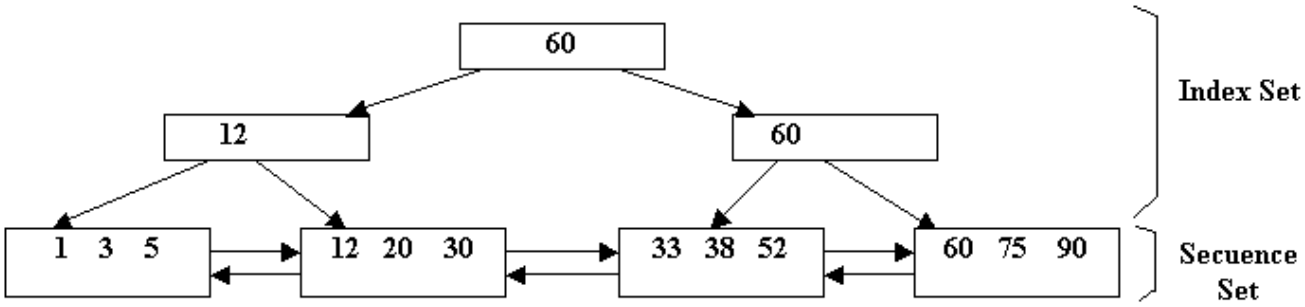
Arboles B+

Un árbol B logra reducir drásticamente la cantidad de seeks necesarios para hacer acceso indexado a un archivo. En cambio, aumenta la cantidad de seeks necesarios para una recorrida secuencial puesto que claves consecutivas se encontrarán distantes en el árbol. Para una recorrida secuencial sería mucho mas conveniente tener las claves físicamente contiguas y reordenar todo el índice en cada alta o baja pero esto requeriría regenerar todo el índice con cada actualización. Una forma de que los cambios no requieran reordenar todo el índice es usar bloques de tamaño fijo. Cada bloque contiene una cantidad de claves ordenadas y un puntero al póximo bloque y al anterior. En caso de un alta o baja sólo se debe reordenar el bloque en la mayoría de los casos o, si un alta produce un overflow o un underflow, redistribuir, hacer split, concatenar o lo que corresponda con bloques adyacentes. Esta lista doblemente enlazada de bloques es lo que se conoce como set secuencial y permite recorrer secuencialmente el índice de forma muy eficiente, alcanza leer el primer bloque, recorrer todas sus claves y cuando se llega al final leer el siguiente bloque. Si se agregan nuevas claves se intercalan bloques en la lista.

Ejemplo de set secuencial:



¿Que pasa si se desea acceder de forma indexada (por referencia) a la clave 38 por ejemplo? Habrá que recorrer todo el set secuencial hasta encontrarla lo cual no resulta muy óptimo. La solución propuesta para construir una estructura que permita acceder de forma secuencial o por referencia a una clave dada de forma rápida es el árbol B+, que consiste en tomar la estructura de set secuencial y agregarle un índice de tipo árbol B.



El contenido del secuence set son las claves y datos o punteros a los datos.El Index Set es un árbol B cuyas hojas apuntan a los bloques del set secuencial. Cada elemento contenido en un nodo de este árbol B no es una clave sino tan sólo un separador, es decir, un valor que permite saber si la clave que se busca está en el subárbol que está a la izquierda del separador o el de la derecha. Estos separadores pueden ser una copia del valor de una clave o simplemente en caso de claves que sean strings por ejemplo, algún prefijo que permita hacer la separación. A los árboles B+ que guardan prefijos para separar en vez de claves completas se los llama árboles B+ de prefijo simple y utilizan un algoritmo para determinar el prefijo mas corto que pueda usarse como separador en cada caso.

Búsqueda:

Se comienza buscando por la raiz y si la clave buscada es menor a la leída se baja al subárbol de la izquierda. En cambio, si es mayor o igual, se baja al subárbol de la derecha. Es muy importante tener en cuenta que si se busca una clave (por ejemplo 60), no importa que se la encuentre en algún nodo del árbol puesto que en tal caso estará actuando tan sólo como separador. Siempre habrá que recorrer el árbol hasta el nivel de las hojas y recién ahí se encontrará la referencia al bloque del secuence set en el cual se aloja realmente la clave. TODAS LA CLAVES EN UN ARBOL B+ APARECEN EN EL SET SECUENCIAL.

Inserciones:

Las inserciones se manejan comenzando por el set secuencial. Se identifica por medio del algoritmo de búsqueda en que bloque del secuence set se deberá insertar la clave. Si el bloque no está lleno se inserta la clave y se debe verificar que el separador siga sirviendo, si no es así se actualiza (esto será necesario cuando se inserte una clave al principio de un bloque). Si el bloque está lleno se deberá hacer un split en el cual no se promueven claves sino que todas se distribuyen en los dos bloques resultantes. Una vez distribuídas las claves se selecciona un separador y es este separador lo que se promueve al Index Set. Si al promover este separador se produce un overflow en el nodo en el que se lo debe insertar, se procede como en un árbol B común.

Bajas :

Las bajas se hacen también sobre el set secuencial. Si al hacer una baja se produce un underflow en el bloque del set secuencial, se intenta en primer lugar redistribuir las claves con los bloques vecinos y si se puede redistribuír, se actualiza el separador de bloques en el Index Set. Si no puede aplicarse redistribución porque los bloques contiguos tienen la cantidad mínima de claves se hace una concatenación y se elimina del árbol el separador que diferenciaba los bloques, lo cual podría producir un underflow en el nodo del Index Set. Dicho underflow se maneja de la forma habitual en un árbol B.

Arboles B virtuales

La organización en nodos que contienen varias claves hacen que el árbol B sea una estructura que resulte muy fácil de paginar. Se puede trabajar manteniendo en memoria uno o varios nodos por vez y el resto en disco. Cuando se necesita accedera un nodo que no está en memoria se baja a disco el que hace mas tiempo que no se accede (Least Recently Used) y se sube el que se necesita en su lugar. Cómo cualquier búsqueda que se haga comienza siempre por la raiz, esta estará siempre en memoria. En algunas pruebas hechas con un árbol B+ se ha calculado que manteniendo cerca del 15% del total de nodos del árbol en memoria, se logra dismiuir la cantidad promedio de accesos a disco por búsqueda a menos de 1. Esta reducción debería ser aún mayor en árboles B y B* puesto que estos no necesitan llegar hasta el último nivel en cada búsqueda.