Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

Gestión de Datos

Árboles

Ing. Enrique Reinosa
Leandro R. Barbagallo
Septiembre 2007

Índice

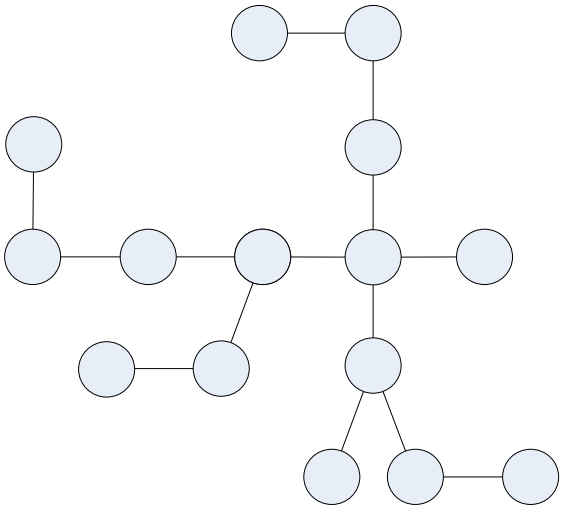
Introducción

El árbol es una estructura fundamental en las ciencias de la computación. Se lo utiliza principalmente para representar jerarquía y ser utilizado en todo lo que conlleve establecer un orden en un conjunto de valores.

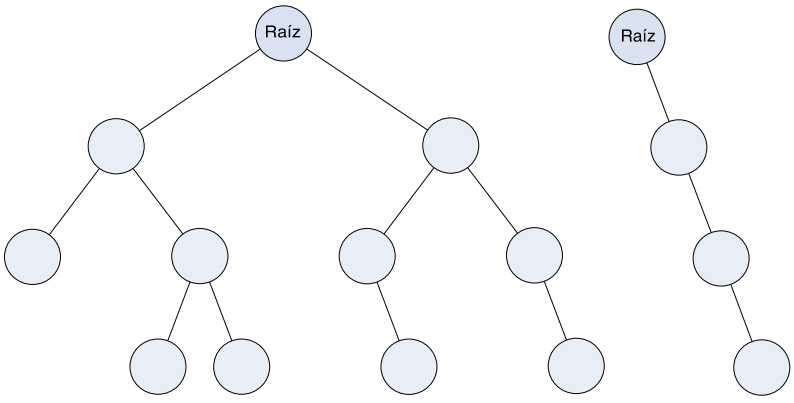
Los árboles tienen muchas aplicaciones. Casi todos los sistemas operativos utilizan estructuras de árbol para almacenar sus archivos. También son usados para el diseño de compiladores, procesamiento de textos y algoritmos de búsqueda.

Un árbol es una colección no vacía de nodos y aristas que cumplen ciertos requisitos. Un nodo (o vértice) es un objeto que puede llevar información asociada, una arista es una conexión entre dos nodos.

Un árbol con raíz es uno en el que se designa un nodo como raíz. En general se reserva el término árbol para referirse a árboles con raíz y árboles libres para aquellas estructuras de datos que cumplen con la característica de árbol pero que no tienen especificada una raíz. En este apunte solo trataremos árboles con raíz.



Árbol libre



Árboles con raices

Formalización

Hay dos formas de definir un árbol, una recursiva y otra no recursiva.

Forma no recursiva

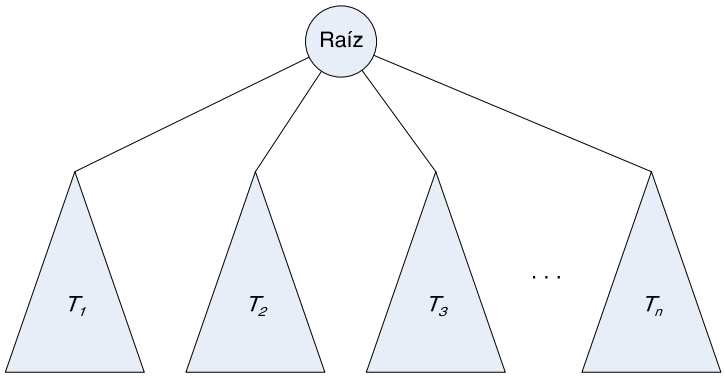
Se distingue un nodo como raíz. A cada nodo *b*, exceptuando la raíz, le llega una arista desde exactamente un nodo *a*, el cual se le llama padre de *b*. Decimos que *b* es uno de los hijos de *a*.

Hay un único camino desde la raíz hasta cada nodo.

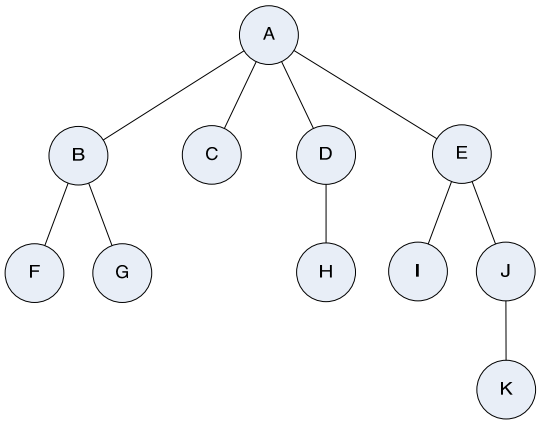
Forma recursiva

Un árbol T es un conjunto finito, no vacío de nodos $T = \{r\} \cup T_1 \cup T_2 \cup \cdots \cup T_n$ con las siguientes propiedades:

- Un nodo del conjunto es designado como raíz del árbol (*r*).
- Los nodos restantes son particionados en $N \geq 0$ subconjuntos, $T_1, T_2, \ldots, T_n$ en el que cada uno es un árbol.



Como pueden ver el árbol es definido en términos de sí mismo. No tenemos el problema de una recursividad infinita ya que un árbol tiene un número finito de nodos y además por que en el caso base un árbol tiene $N = 0$ nodos.



Nodo	Altura	Profundidad
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

En la figura se muestra un ejemplo de árbol. El nodo raíz es *A*, y los hijos de *A* son *B*, *C*, *D* y *E*. Como *A* es la raíz, no tiene padre. Todos los demás nodos tienen padre, por ejemplo, el padre de *B* es *A*. Los nodos que no tienen hijos reciben el nombre de *hojas* o *nodos terminales*. En este caso las hojas son *F, G, C, H*, *I*, y *K*.

Frecuentemente las hojas son nodos diferentes a los que no son hojas, por ejemplo, pueden no tener nombre o información asociada. Entonces en tales situaciones se denomina *nodos externos* a las hojas y *nodos internos* a los nodos que no son hojas.

A un conjunto de árboles se le denomina *bosque*, por ejemplo si se suprime la raíz del ejemplo de arriba, se obtiene un bosque formado por cuatro árboles de raíces *B*, *C*, *D* y *E*.

El *grado* de un nodo es el número de subárboles asociados a ese nodo. Por ejemplo el grado de *A* es 4, el grado de *E* es 2, el grado de G es 0.

La *longitud de camino* de *A* a *K* es tres (aristas), y la del camino de *A* a *A* es de cero aristas. La *profundidad* o *nivel* de un nodo en un árbol es la longitud del camino que va desde la raíz hasta ese nodo. Por lo tanto la profundidad de la raíz es siempre 0 y la de cualquier nodo es la de su padre más uno. La *altura de un nodo* es la longitud del camino que va desde el nodo hasta la hoja más profunda bajo él. La *altura de un árbol* se define como la altura de su raíz.

Los nodos que tienen el mismo padre son *hermanos*, luego *B*, *C*, *D* y *E* son todos hermanos. Si hay un camino del nodo *u* al nodo *v*, entonces decimos que *u* es un *ascendiente* de *v* y que *v* es un *descendiente* de *u*. El *tamaño* de un nodo es igual al número de de descendientes que tiene (incluyendo dicho nodo). El tamaño de *B* es 3 y el de *C* es 1. El tamaño de un árbol se define como el tamaño de su raíz.

El orden en que se coloca a los hijos de un nodo es a veces importante, y a veces no. Un *árbol ordenado* es aquel árbol con raíz en el que se ha sido especificado el orden de los hijos en cada nodo.

Propiedades

- Dados dos nodos cualesquiera de un árbol, existe exactamente un camino que los conecta. Una consecuencia importante de esta propiedad es que cualquier nodo puede ser raíz, ya que, dado cualquier nodo de un árbol, existe exactamente un camino que lo conecta con cualquier otro nodo del árbol.

- Un árbol con N nodos debe tener $N-1$ aristas porque a cada nodo, excepto la raíz, le llega una arista.

- La altura de cualquier nodo es uno más que la mayor altura de un hijo suyo.

- Un árbol N-ario con $n \geq 0$ nodos internos, contiene $(N-1)n+1$ nodos externos.

- Un árbol N-ario de altura $h \geq 0$ tiene como máximo N^h hojas.

Árbol binario

Un árbol binario es un árbol en el que ningún nodo puede tener más de dos hijos, es decir un árbol N-Ario donde $N = 2$. Dado que cada nodo tiene como máximo dos hijos, se los distingue como izquierdo y derecho.

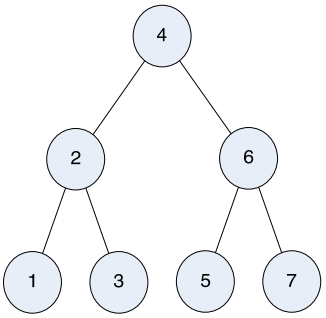
Entre las muchas aplicaciones de los árboles binarios podemos nombrar los árboles de expresión, que son la estructura fundamental en el diseño de compiladores, las árboles de codificación de Huffman, que es un algoritmo eficiente para la compresión de datos, y el uso como soporte para los árboles binarios de búsqueda.

En general, la altura promedio de un árbol binario es $O(\sqrt{N})$, pero puede legar a ser $N-1$.

Árbol binario perfecto

Un árbol binario perfecto de altura $h \geq 0$ es un árbol binario $T = \{r, T_L, T_R\}$ con las siguientes propiedades:

1. Si $h = 0$, entonces $T_L = \phi$ y $T_R = \phi$
2. Por el contrario, si $h > 0$, ambos T_L y T_R son árboles binarios perfectos de altura $h-1$.



Árbol binario perfecto

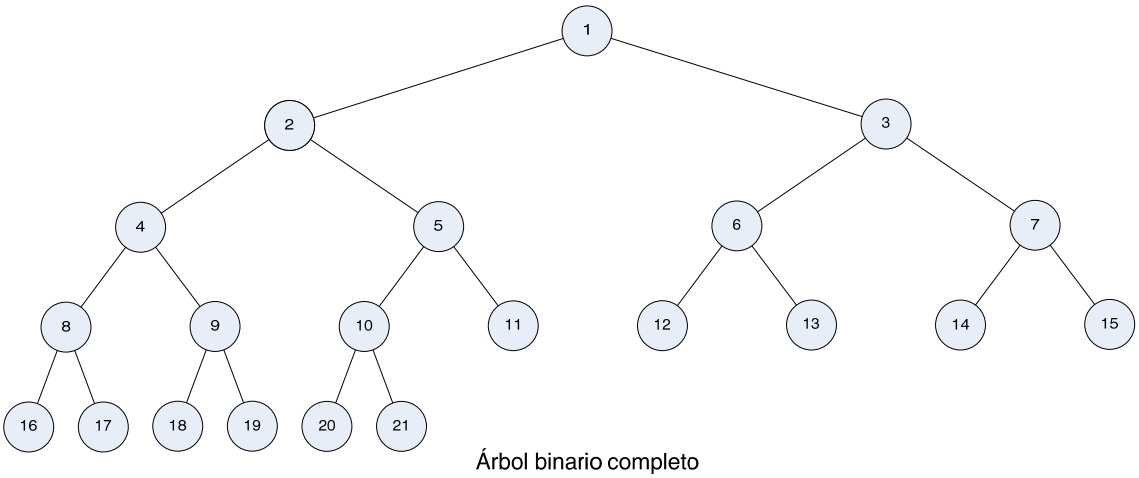
Una de las propiedades más importantes de estos árboles es que un árbol binario perfecto de altura h tiene exactamente $2^{(h+1)} - 1$ nodos internos. Por lo tanto la cantidad de nodos permitido en este tipo de árboles es 0, 1, 3, 7, 15, 31,…., $2^{(h+1)} - 1$,…

El peor caso para una búsqueda en un árbol binario perfecto es $O(\log n)$.

Árbol binario completo

El árbol binario completo esta relacionado con el árbol binario perfecto. Mientras que este último solo permite ciertos números de nodos, el árbol completo soporta un número arbitrario de nodos.

Podemos definirlo entonces como, un árbol binario en el que todos los niveles están llenos, excepto posiblemente el ultimo nivel, el cual es completado de izquierda a derecha.



Árbol binario completo

Un árbol binario completo de altura $h \geq 0$ contiene al menos 2^h nodos y a lo sumo $2^{(h+1)} - 1$ nodos.

Árbol binario balanceado

Un árbol binario esta completamente balanceado si esta vació, o ambos subárboles están completamente balanceados y tienen la misma altura. Por lo tanto cualquier camino desde la raíz a una hoja tiene la misma longitud. Los únicos árboles que están perfectamente balanceados son los árboles binarios perfectos.

Un árbol binario no vacío $T = \{r, T_L, T_R\}$ está AVL balanceado si $|H_L - H_R| \leq 1$, donde H_L es la altura de T_L y H_R es la altura de T_R. En otras palabras, en todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Debido a esta propiedad estructural, el orden de complejidad de la búsqueda en estos árboles se mantiene en $O(\log n)$

Recorrido de árboles

Hay muchas formas de hacer uso de los árboles y como resultado hay varios algoritmos diferentes para manipularlos. Sin embargo, muchos de los algoritmos que utilizan árboles tienen en común que visitan a los nodos del árbol en forma sistemática. Es decir que el algoritmo recorre la estructura de datos del árbol y realiza algún cálculo en cada uno de los nodos. El proceso de moverse entre los nodos de un árbol es llamado recorrido de árbol.

Esencialmente hay dos métodos distintos para visitar todos los nodos de un árbol, recorrido *primero en profundidad* y recorrido *primero en anchura (u orden de nivel)*. Entre los de primero en profundidad se encuentran, *preorden*, *postorden*, *en orden*.

Preorden

El recorrido en preorden puede ser definido recursivamente de la siguiente manera:

1. Visitar primero la raíz.
2. Hacer un recorrido preorden a cada uno de los subárboles de la raíz, uno por uno en un orden determinado.

En el caso de de un árbol binario, el algoritmo resulta:

1. Visitar primero la raíz
2. Recorrer el subárbol izquierdo
3. Recorrer el subárbol derecho.

Inorden

El recorrido inorden, también llamado simétrico, solo puede ser usado en árboles binarios. En este recorrido, se visita la raíz entre medio de las visitas entre el subárbol izquierdo y derecho.

1. Recorrer el subárbol izquierdo
2. Visitar la raíz
3. Recorrer el subárbol derecho.

Postorden

El último de los métodos de recorrido de árboles de primero en profundidad es el postorden. A diferencia del preorden donde se visita primero a la raíz, en el recorrido en postorden se visita la raíz a lo último.

1. Hacer un recorrido preorden a cada uno de los subárboles de la raíz, uno por uno en un orden determinado.
2. Visitar por ultimo la raíz.

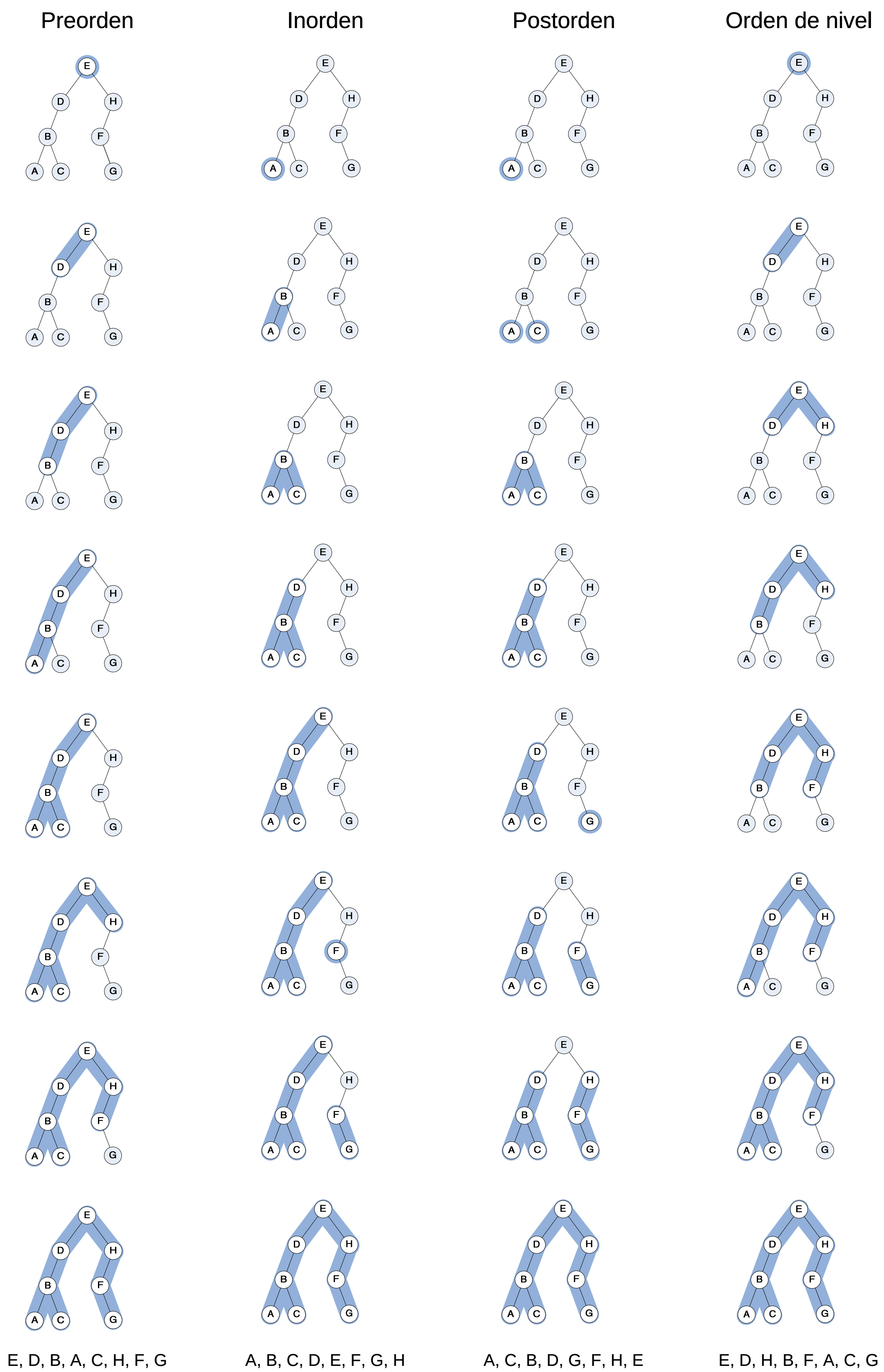
En el caso de los árboles binarios:

1. Recorrer el subárbol izquierdo
2. Recorrer el subárbol derecho.
3. Visitar por último la raíz.

Primero en anchura

Mientras que los recorridos primero en profundidad son definidos de forma recursiva, los de primero en anchura no se los considera recursivos. En primero en anchura se visitan los nodos en el orden de su profundidad de en el árbol. Primero se visita todos los nodos en la profundidad cero (la raíz), y después todos los de profundidad igual a uno, etc. Para poder realizar una búsqueda de primero en anchura es necesario valerse de una estructura de datos auxiliar, la cola.

1. Poner en cola la raíz.
2. Mientras que la cola no esté vacía
    a. Quitar primero de la cola y asignarlo a una variable auxiliar, *Nodo*.
    b. Imprimir el contenido de *Nodo*.
    c. Si *Nodo* tiene hijo izquierdo, poner al hijo izquierdo en la cola.
    d. Si *Nodo* tiene hijo derecho, poner al hijo derecho en la cola.


Preorden
Inorden
Postorden
Orden de nivel
E, D, B, A, C, H, F, G
A, B, C, D, E, F, G, H
A, C, B, D, G, F, H, E
E, D, H, B, F, A, C, G

Representación implícita de árboles binarios

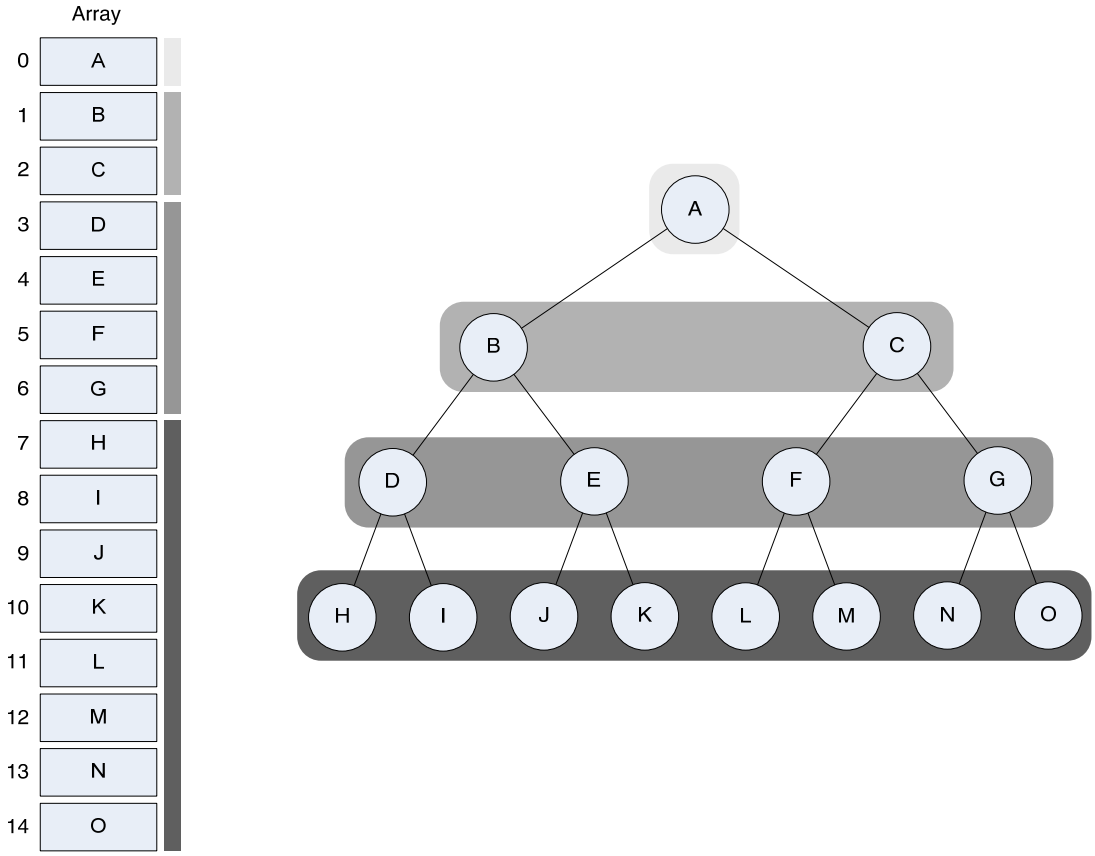
Hasta ahora la representación de los árboles, era basada en la idea de tener dos referencias a los subárboles derecho e izquierdo en cada nodo. Sin embargo hay otra forma completamente diferente de representar un árbol binario, mediante un array. En este caso los nodos están almacenados en el array en vez de estar vinculados por medio de referencias.

La posición del nodo en el array corresponde a su posición en el árbol. El nodo en la posición 0 es la raíz, el nodo en la posición 1 es el hijo izquierdo, y el nodo de la posición 2 es el hijo derecho. Se sigue con este procedimiento de izquierda a derecha en cada nivel del árbol.

En cada posición del árbol, sin importar si representa un nodo existente o no, corresponde a un elemento en el vector. Los elementos que representan posiciones del árbol sin nodos son rellenados con 0 o con *null*.

Basándose en este esquema, los hijos y el padre de un nodo puede ser encontrado haciendo cálculos en base al índice de cada nodo en el array. Si el índice de un nodo es I entonces:

El hijo izquierdo del nodo es: $2I+1$
El hijo derecho del nodo es: $2I+2$
El padre de I es: $\lfloor (i-1)/2 \rfloor$



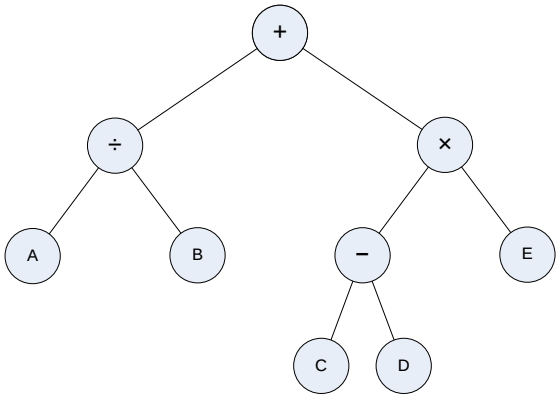
En muchas situaciones, representar un árbol con un array resulta muy eficiente. Los nodos sin llenar o nodos borrados dejan huecos en el array, desperdiciando memoria. Sin embargo si el borrado de nodos no esta permitido, puede ser una buena alternativa, y además es útil, si es muy costoso obtener memoria para cada nodo de forma dinámica.

Esta forma de representar un árbol binario mediante un array es una forma de implementar colas de prioridad mediante montículos binarios. Esta estructura de datos es muy importante y se la aplica en muchas situaciones, incluyendo el método de clasificación por montículo o Heap Sort.

Árboles de expresión

Entre los muchos usos que se le puede dar a los árboles binarios es la construcción de árboles de expresión. Las expresiones algebraicas tienen una estructura tipo árbol.

Por ejemplo, la expresión $a/b+(c-d)e$ puede ser representada por el siguiente árbol.



Los nodos terminales (hojas) de un árbol de expresión son las variables o constantes en la expresión (*a*, *b*, *c*, *d,* y *e*). Por otra parte los no terminales son los operadores (+,-× y ÷). Los paréntesis de la ecuación no aparecen en el árbol, ya que justamente las posiciones de los operadores en el árbol son los que le asignan prioridades, ya que por ejemplo la resta esta más abajo en el árbol que la multiplicación.

Para imprimir la ecuación de un árbol de expresión, generalmente se usa el recorrido inorden y se utiliza el siguiente procedimiento:

Si se encuentra un nodo terminal, se lo imprime. Cuando se encuentra un nodo no terminal, se hace lo siguiente:

1- Imprimir un paréntesis izquierdo
2- Recorrer el subárbol izquierdo
3- Imprimir la raíz
4- Recorrer el subárbol derecho.
5- Imprimir el paréntesis derecho.

Si se aplica este procedimiento se obtiene $((a/b)+((c-d)\times e))$ que a pesar de los paréntesis redundantes, representa la misma expresión que la ecuación original.

Árboles de búsqueda

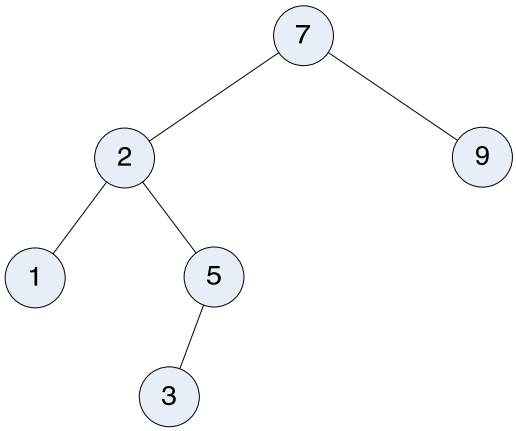
Se llama árbol de búsqueda a un árbol que soporta las operaciones de búsqueda, inserción y eliminación de forma eficiente.

Lo que hace a un árbol de búsqueda es que las claves no aparecen en los nodos de forma arbitraria, sino que hay un criterio de orden que determina donde una determinada clave puede ubicarse en el árbol en relación con las otras claves en ese árbol.

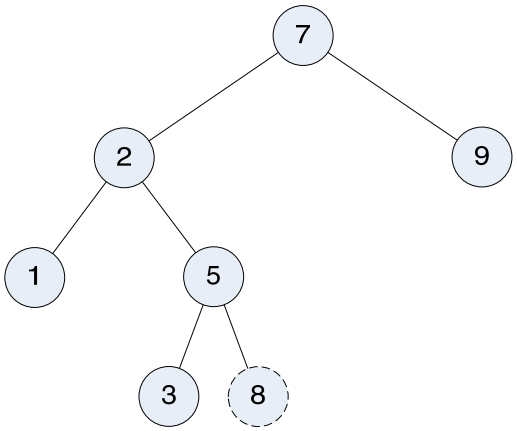
Árbol binario de búsqueda

El árbol binario de búsqueda es una estructura de datos que puede considerarse como una extensión del algoritmo de búsqueda binario que permite tanto inserciones como eliminaciones. El tiempo de ejecución de muchas de sus operaciones es de $O(\log N)$ en el caso medio, pero en el peor de los casos puede llegar a $O(N)$.

Un árbol binario de búsqueda T puede definirse como un árbol que satisface la propiedad de búsqueda ordenada. Esto significa que para cada nodo X del árbol, los valores de todas las claves de su subárbol izquierdo son menores que la clave de X y los valores de todas las claves de su subárbol derecho son mayores que la clave de X.



Árbol binario de búsqueda



Árbol binario

El árbol de la izquierda es un árbol binario de búsqueda mientras que el de la derecha no lo es ya que la clave 8 no debería pertenecer al subárbol izquierdo de la clave 7.

Si a este tipo de árbol se lo recorre en inorden se muestran los elementos ordenados de forma ascendente.

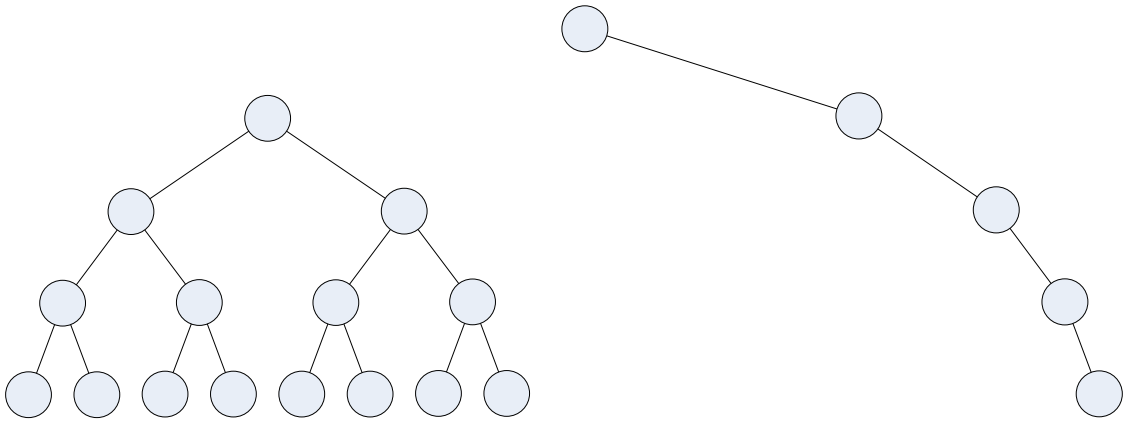
Este tipo de estructura no permite en principio la existencia de elementos duplicados, pero es posible modificarla para si se permitan. Ante la existencia de dos claves duplicadas, una de las soluciones es tener una sola clave, y guardar un contador con el número de repeticiones.

Para buscar un elemento se empieza por la raíz y desplaza repetidamente por las ramas izquierda o derecha, dependiendo del resultado de las comparaciones. Por ejemplo para encontrar el elemento 5, empezamos por el 7 y vamos por la rama izquierda. Luego en el 2 vamos a la rama derecha y ahí llegamos al 5. Si estuviésemos buscando un elemento inexistente, como por ejemplo un 6, encontraríamos una referencia null en la rama derecha de 5.

Otra operación interesante es buscar el mínimo que se obtiene recorriendo repetidamente los nodos izquierdos hasta llegar a un nodo que no tenga hijo izquierdo. Para buscar el máximo se recorre de forma similar pero haciéndolo por la derecha.

La eliminación e inserción de nodos es una tarea que requiere de cierta complejidad. En el caso de querer eliminar un nodo se debe evitar que al hacerlo el árbol no quede desconectado.

El costo de las operaciones del árbol binario de búsqueda es proporcional al número de nodos consultados durante la operación. El costo de acceso a cada nodo es 1 más su profundidad.



Árbol binario equilibrado

Árbol binario no equilibrado

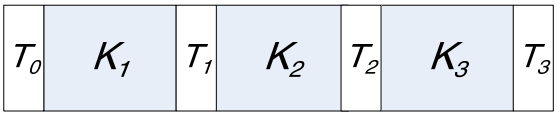
En este ejemplo se ven dos árboles binarios. El de la izquierda es un árbol bien equilibrado con 15 nodos, en donde el costo para acceder a cualquier nodo es de a lo sumo 4 unidades. Como conclusión, si el árbol esta bien equilibrado, el costo de los accesos es logarítmico.

En el árbol de la derecha es un ejemplo clásico de árbol no equilibrado. Hay N nodos en el camino para recorrer hasta el nodo de mayor profundidad, con lo que el costo de la búsqueda en el peor de los casos es $O(N)$. También es posible demostrar que en el caso medio, la mayoría de las operaciones requieren de un tiempo cercano a $O(\log N)$.

Árboles M-arios

Un árbol M-ario es consiste de n subárboles $T_0, T_1, \ldots, T_{n-1}$ y $n-1$ claves $K_1, K_2 \ldots, K_{n-1}$, $T = \{T_0, k_1, T_1, K_2, T_2, \ldots, K_{n-1}, T_{n-1}\}$ donde $2 \le n \le M$, de tal manera que las claves y nodos satisfacen las siguientes propiedades de ordenación de datos:

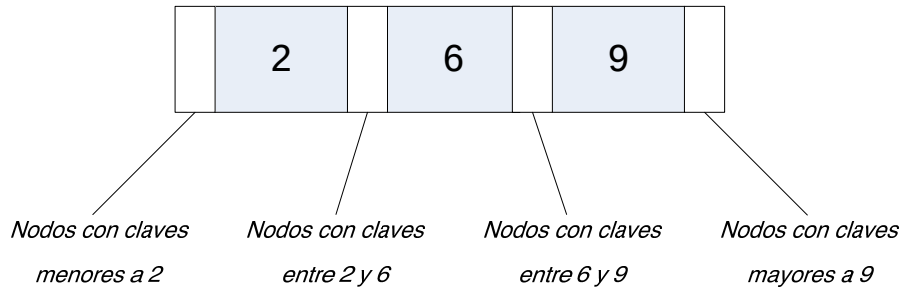
1- Las claves en cada nodo son distintas y están ordenadas, por ejemplo $K_i < K_{i+1}$ para $1 \le i \le n-1$
2- Todas las claves contenidas en el subárbol T_{i-1} son menores que K_i. Al árbol T_{i-1} se lo llama subárbol izquierdo respecto de la clave K_i.
3- Todas las claves contenidas en el subárbol T_i son mayores que K_i. Al árbol T_{i+1} se lo llama subárbol derecho respecto de la clave K_i.



Nodo de un árbol M-ario de búsqueda para M=4

A los árboles M-arios también se los llama árboles multicamino.

Los árboles binarios de búsqueda son árboles M-arios donde $M = 2$.



Árboles de búsqueda M-arios.

Si se tiene un conjunto de datos muy grande, tan grande que no podemos colocarlo en memoria principal, nos veríamos obligados implementar el árbol de búsqueda en un almacenamiento secundario, como el disco. Las características de un disco a diferencia de la memoria principal hacen que sea necesario utilizar valores de M más grandes para poder implementar estos árboles de búsqueda de forma eficiente.

El tiempo de acceso de un disco típico es de 1 a 10 ms, mientras que el tiempo acceso típico de una memoria principal es de 10 a 100 ns. Por lo tanto los accesos a memoria son entre 10.000 y 1.000.000 de veces más veloces que los accesos a disco. Para maximizar la performance es necesario minimizar a toda costa los accesos a disco.

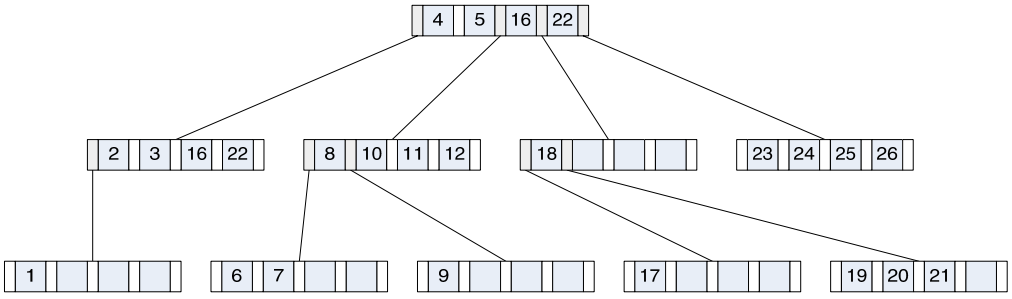
Además, los discos son dispositivos orientados a bloques. Los datos son transferidos en bloques de gran tamaño entre la memoria principal y el disco. Los tamaños de bloques típicos varían entre 512 y 4096 bytes. En consecuencia, tiene sentido tomar ventaja de esa habilidad para transferir grandes bloques de datos eficientemente.

Al elegir un valor de M grande, podemos arreglar para que un nodo de un árbol M-ario pueda ocupar un bloque de disco completo. Si cada nodo interno en el árbol M-ario tiene exactamente M hijos, podes usar el siguiente teorema:

$$h \geq \lceil \log_M ((M-1)n+1) \rceil - 1$$

Donde n es el número de nodos internos de un árbol de búsqueda. Un nodo en un árbol M-ario de búsqueda que tiene M hijos contiene exactamente $M-1$ claves. Por lo tanto, en total hay $K = (M-1)n$ claves, entonces resulta $h \geq \lceil \log_M (K-1) \rceil - 1$.

Por ejemplo, si consideramos un árbol de búsqueda que contiene $K = 2097151$ claves. Suponiendo que el tamaño de un bloque de disco es tal que podemos ajustar un nodo de tamaño $M = 128$ en él. Dado que cada nodo contiene a lo sumo 127 claves, por lo menos necesitaremos 16513 nodos. En el mejor de los casos, la altura de un árbol de búsqueda M-ario es solo dos, y a lo sumo tres accesos a disco son necesarios para recuperar una clave. Esto es una mejora significativa comparada con un árbol binario, el cual tendría una altura de al menos 20.



Árbol M-ario de búsqueda para M=5

Árboles-B

Los árboles-B fueron inventados por R. Bayer y E. McCright en 1972 (por lo tanto la "B" es por balanceado o por Bayer, pero de ninguna manera es por binario). Este tipo de árbol y sus variantes tienen aplicaciones en bases de datos y en sistemas de archivos.

Los Árboles-B son Árboles M-arios balanceados diseñados para funcionar bien en discos magnéticos o en cualquier otro tipo de almacenamiento secundario de acceso directo. El objetivo principal es minimizar las operaciones de entrada y salida hacia el disco. Al imponer la condición de balance, el árbol es restringido de manera tal que se garantice que la búsqueda, la inserción y la eliminación de sean todos de tiempo $O(\log N)$.

En una aplicación típica que utiliza árboles-B, la cantidad de datos manejados es tan grande que todos los datos no caben en memoria principal al mismo tiempo. Los algoritmos de árboles-B copian los bloques o páginas de disco a memoria principal a medida que se los necesita y se vuelve a escribir solo las páginas que han sido modificadas. Dado que estos algoritmos solo necesitan una cantidad constante de páginas presentes en memoria principal a la vez, el tamaño de la memoria principal no limita el tamaño del árbol-B que puede ser manejado.

Los árboles-B de orden M son solo definidos para $M \geq 2$. Sin embargo en la práctica se espera que M sea mucho más grande por las mismas razones que se usan los árboles M-arios, que es la utilización en grandes bases de datos en almacenamiento secundario. Estos valores de M varían entre 50 y 2000 y se determina en base al tamaño de las claves y del tamaño de la pagina del disco.

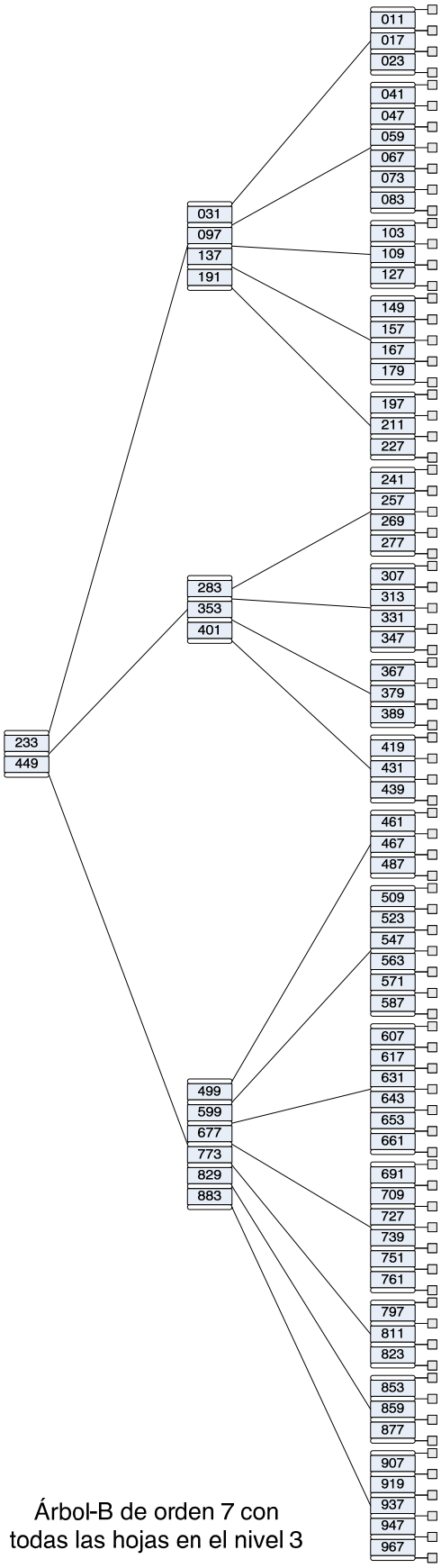
Definición

Un árbol-B de orden M es o bien un árbol vacío, o es un árbol M-ario de búsqueda T con las siguientes propiedades:

1- La raíz de T tiene al menos dos subárboles y a lo sumo M subárboles.
2- Todos los nodos internos de T (menos la raíz) tienen entre $\lceil M/2 \rceil$ y M subárboles.
3- Todos los nodos externos de T están al mismo nivel.

Los nodos deben estar, al menos, medio llenos. Esto garantiza que el árbol no degenerara en un simple árbol binario o ternario.

Hay muchas variantes del árbol-B estándar, incluyendo los árboles-B+, árboles-B*, etc. Todos están diseñados para resolver distintos aspectos de la búsqueda en almacenamiento externo. Sin embargo, cada una de estas variaciones tiene sus raíces en el árbol-B básico. Por ejemplo si solo almacenamos los datos en las hojas del árbol, se lo llama árbol-B+ y es usado en muchos sistemas de archivos como NTFS, RaiserFS, XFS, etc. También se usan este tipo de árboles en los índices de las tablas de bases de datos relacionales.



Árbol-B de orden 7 con
todas las hojas en el nivel 3

Operaciones básicas sobre un árbol-B

Búsqueda

Buscar en un árbol-B es muy parecido a buscar en un árbol binario de búsqueda, excepto que en vez de hacer una decisión binaria, o de dos caminos en cada nodo, hacemos una decisión multicamino en base al número de hijos del nodo.

Si necesitamos buscar un ítem x en un árbol-B, debemos comenzar por la raíz. Si el árbol está vacío, la búsqueda falla. De lo contrario, las claves en la nodo raíz son examinadas para determinar si el elemento que se esta buscando esta presente. Si está, la búsqueda termina exitosamente. Si no está, hay tres posibilidades a considerar:

- Si el elemento x a buscar es menor que K_1, entonces se continúa buscando en el subárbol T_0.
- Si x es más grande que K_{n-1}, se continúa buscando en el subárbol T_{n-1}.
- Si existe un i tal que $1 \le i \le n-1$ para el cual $K_i \le x \le K_{i+1}$ entonces se continúa buscando en el árbol T_i.

Cabe remarcar que cuando no se encuentra x en un nodo dado, solo se continúa buscando en uno solo de sus n subárboles. Por lo tanto no es necesario un recorrido completo del árbol. Una búsqueda exitosa comienza en la raíz y sigue el camino hacia abajo del árbol, el cual termina en un nodo que contiene el elemento de la búsqueda. El tiempo de ejecución de una búsqueda exitosa depende por la profundidad en que se encuentre el elemento a buscar dentro del árbol.

Cuando el elemento a buscar no está en el árbol, el método de búsqueda termina cuando se encuentra un subárbol vacío. En el peor de los casos, el camino de búsqueda termina en el nodo de la hoja más profunda, por lo tanto el tiempo de ejecución en el peor de los casos esta determinada por la altura del árbol-B.

Inserción

El algoritmo de inserción en un árbol-B comienza de la misma manera que los demás algoritmos de inserción en árboles de búsqueda. Para insertar un elemento x, comenzamos en la raíz y realizamos una búsqueda para el. Asumiendo que el elemento no esta previamente en el árbol, la búsqueda sin éxito terminará en un nodo hoja. Este es el punto en el árbol donde el x va a ser insertada.

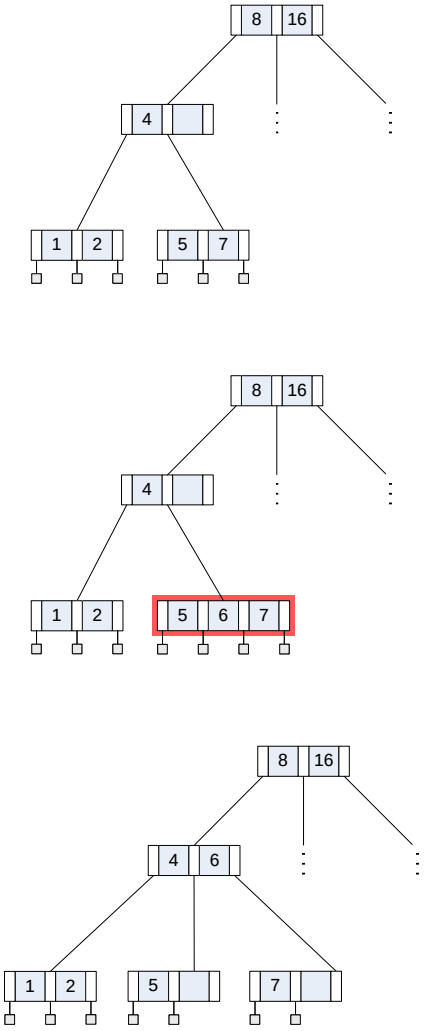
Si el nodo hoja tiene menos de $M-1$ claves en el, simplemente insertamos el elemento en el nodo hoja y damos por terminado.

Por cada clave que insertamos en el nodo, se requiere un nuevo subárbol. Si el nodo es una hoja, el subárbol a insertar está vacío. Por lo tanto cuando insertamos un elemento x, en verdad, estamos insertando el par de elementos (x, ϕ).

¿Pero que pasa si la hoja esta completa? Esto es, suponiendo que queremos insertar el par (x, ϕ), en un nodo T que ya tiene $M-1$ claves. Esto resultaría en un nodo de árbol-B de orden M no válido por que tiene $M+1$ subárboles y M claves. La solución es dividir el nodo T a la mitad, creando dos nodos, T'_L y T'_R, cada uno

conteniendo la mitad de nodos que el original, y una clave restante, llamada $k_{\lceil M/2 \rceil}$. Ahora hay dos casos para considerar, si T es la raíz o no lo es.

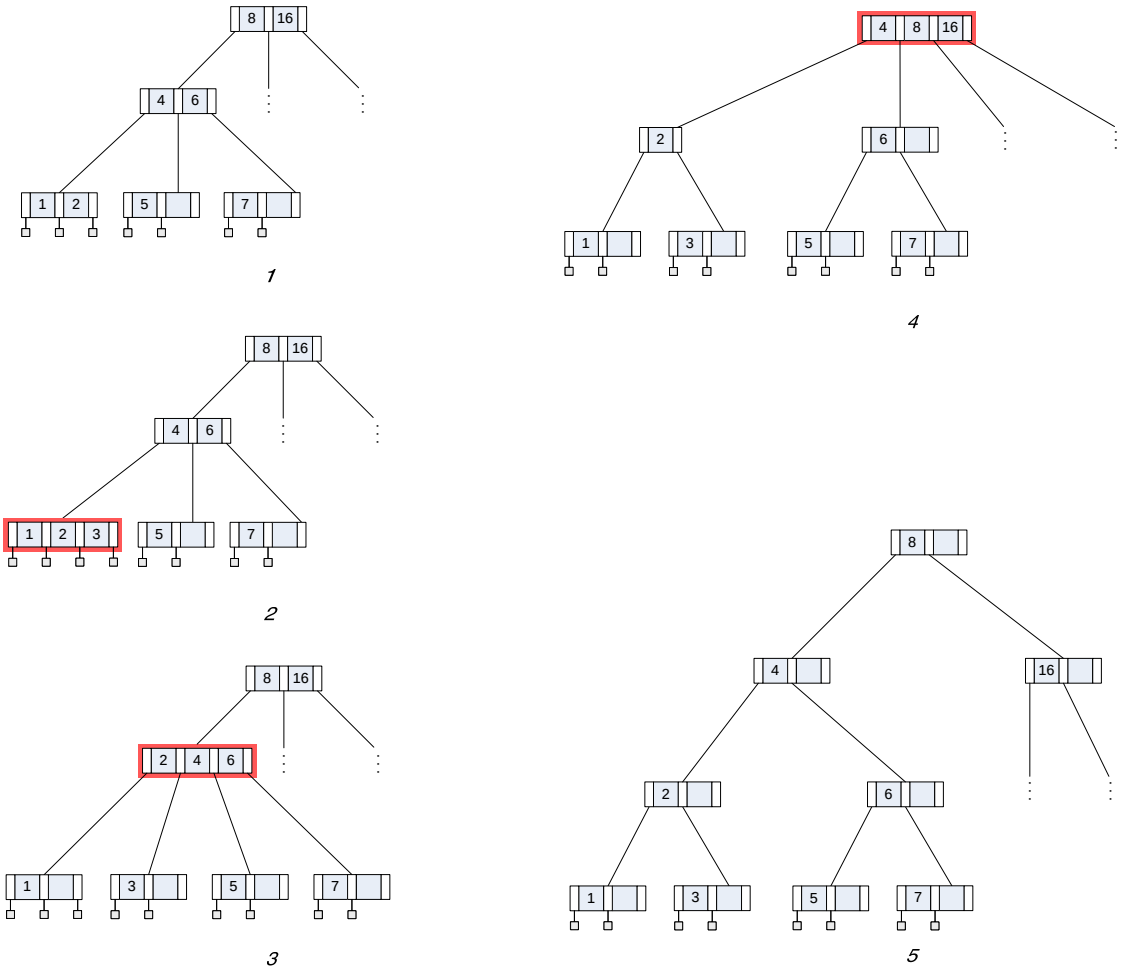
Si T no es la raíz, se hace lo siguiente: Primero, T'_L reemplaza a T en el padre de T. Luego, tomamos el par $(k_{\lceil M/2 \rceil}, T'_R)$ y lo insertamos recursivamente en el padre de T.



Inserción de items en un árbol-B (insertar 6)

En la figura vemos el caso de una inserción en un árbol-B de orden 3. Al insertar la clave 6 en el árbol causa que el nodo hoja desborde. La hoja es dividida en dos. La mitad izquierda contiene la clave 5, la derecha la clave 7, y la clave 6 es la clave restante. Las dos mitades son reconectadas al padre en el lugar apropiado con la clave restante entre medio de ellas.

Si el nodo padre se llena, entonces se divide también y los dos nuevos nodos son insertados en el abuelo. Este proceso continua todo su camino hacia arriba del árbol hasta la raíz. ¿Pero qué hacer cuando la raíz se llena también? Cuando se llena la raíz también se divide. Sin embargo, dado que no hay padre al que insertar los dos nuevos hijos, una nueva raíz es insertada arriba de la vieja raíz. La nueva raíz va a contener exactamente dos subárboles y una sola clave.



Inserción de items en un árbol-B (insertar 3)

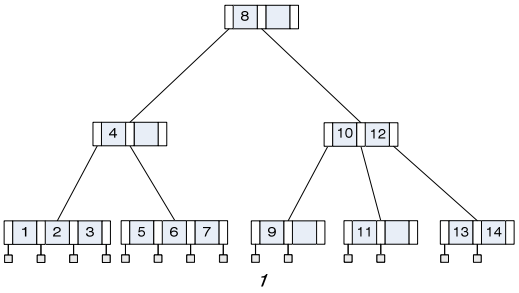
En la figura vemos como al insertar la clave 3 en el árbol causa que el nodo hoja se desborde. Dividir la hoja y readjuntarla causa que el padre se desborde. De la misma forma, dividir el padre y readjuntarla causa que el abuelo se desborde, pero el abuelo es la raíz. La raíz es dividida y la nueva raíz es adjuntada por encima de ella.

Una cosa interesante para destacar, es que la altura del árbol-B solo se incrementa cuando el nodo raíz se divide. Lo que es más, cuando el nodo raíz se divide, la dos mitades son adjuntadas a la nueva raíz, por lo tanto todos los nodos externos permanecen a la misma profundidad, cumpliendo con la definición de árbol-B.

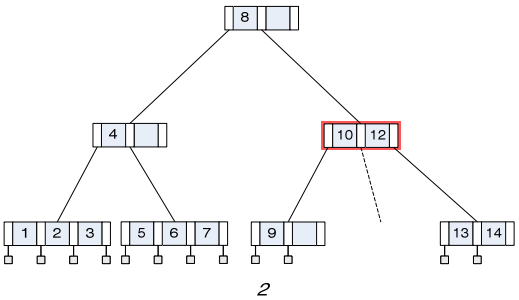
Eliminación

La eliminación de elementos en un árbol-b es más complicada que la búsqueda e inserción, debido a que involucra la fusión de nodos.

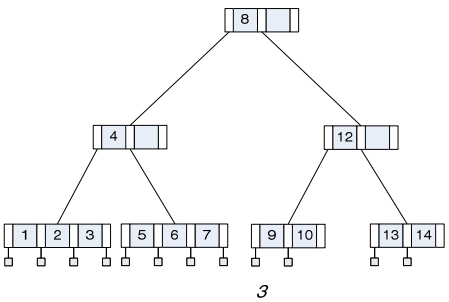
En el siguiente ejemplo, muestra que luego de eliminar la clave *11* el árbol, éste ya no es un árbol-B válido por no tener un nodo hijo entre las claves *10* y *12*. Para corregir esta situación es necesario redistribuir algunas de las claves en medio de los hijos. En este caso la clave *10* es empujada hacia abajo al nodo que contiene la clave *9*.



1



2

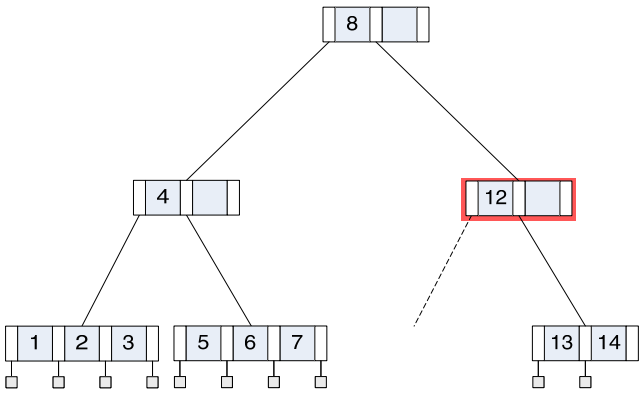


3

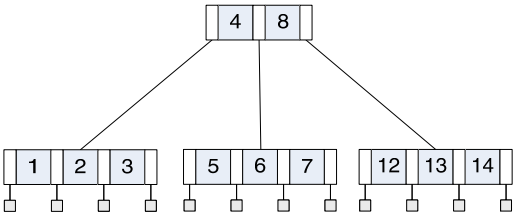
El caso anterior es uno de los más simples.

Si, por ejemplo las claves 9 y 10 fuesen eliminadas, entonces el árbol quedaría sin su primer hijo en el nodo donde esta la clave 12.

De nuevo, es necesaria la redistribución de claves para corregir el desequilibrio del árbol. Las claves de los nodos de los padres son fusionadas con los de los hijos. En algún punto el nodo raíz debe ser empujado hacia abajo (o eliminado). Cuando esto sucede, la altura del árbol, es reducida en una unidad. En este caso, se fusiona la clave 12 con su hijo y se baja el nodo raíz transformándolo en el padre.



1



2

Como se puede observar, la eliminación es un proceso un tanto complicado, e incluye un montón de casos a considerar.

Bibliografía

Robert Lafore, Data Structures & Algorithms in Java, Sams, 1998.

Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in Java, 1998.

Robert Sedgewick, Algorithms in Java, Addison Wesley, 2002.

Thomas H. Cormen, Introduction to Algorithms (Second Edition), MIT Press, 2001.

Mark A. Weiss, Data Structures and Problem Solving Using C++, Pearson, 2003.

Simon Harris, Beginning Algorithms, Wiley Publishing, 2006

The Art of Computer Programming (Second Edition) Volume 3, Donald Knuth, Addison Wesley, 1998.

Algorithms and Data Structures: The Science of Computing, Douglas Baldwin, Charles Riven Media, 2004.