

Arboles

Algoritmos y Estructuras de Datos (CB100) - FIUBA
Martin Klöckner - mklockner@fi.uba.ar

Un árbol es una estructura de datos que posee ramificaciones pero no sigue con una estructura lineal, como las listas enlazadas por ejemplo (salvo en casos particulares) en esta estructura es posible poseer mas de una posición siguiente, aunque típicamente se tienen dos (árboles binarios). Un árbol obligatoriamente debe cumplir que cada nodo tenga un solo padre.

Dentro de la estructura se asigna diferentes nombres a los nodos con diferentes características, como la **raíz**, el cual es el nodo que no tiene “ancestros” y la **hoja** que es un nodo que no tiene hijos. Otras definiciones incluyen el **grado** que es el numero de hijos máximo que puede tener un subárbol o nodo, y la **altura** de un nodo que es la longitud del camino más largo desde el nodo a la raíz (por convención la raíz tiene altura 1).

En la figura 1.1 se muestra un ejemplo de un árbol binario, en el cual la raíz es el nodo A, las hojas los nodos D, E, F y G, el grado 2 (ya que es binario) y la altura 3.

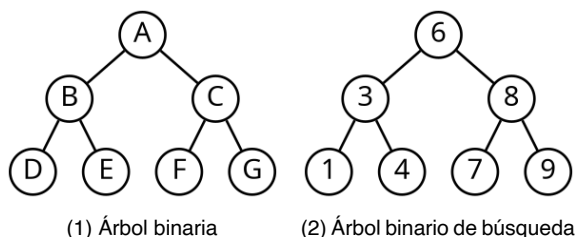


Figura 1: Ejemplos de arboles binarios

En la figura 2 se ve un ejemplo de la estructura típica de un árbol binario de altura 2 junto con el contenido de los nodos, cada nodo tiene un dato asociado que almacena y a su vez una referencia al nodo izquierdo y derecho que le preceden.

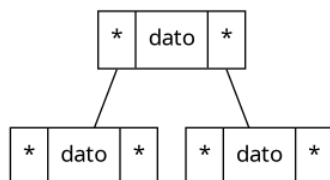


Figura 2: Estructura de nodo de un árbol binario

Para insertar o eliminar un elemento no hay un orden particular y depende la implementación, lo general es **insertar** elementos en las hojas, creando un nuevo nodo, y para **eliminar** un elemento, en el peor caso caso (cuando un nodo tiene dos hijos) insertar en el espacio el ultimo nodo insertado (nodo hoja).

Arboles de búsqueda binaria

Un árbol de búsqueda binaria o ABB por sus siglas, es un árbol binario, es decir que tiene un máximo de dos hijos por cada nodo, pero que además está ordenado, esto es, el nodo izquierdo contiene un dato menor en comparación con el nodo actual, y el nodo derecho un dato mayor. Un ejemplo de un árbol de búsqueda binaria se puede ver en la figura 1.2.

Tener un árbol binario ordenado permite reducir el numero de pasos requeridos para encontrar un dato almacenado en el árbol. Los ABB deben cumplir que el dato de la raíz sea mayor al dato de todos los valores almacenados en los nodos hijos del lado izquierdo y menor a todos los datos almacenados en los hijos del lado derecho. En el mejor de los casos la altura de un ABB es $\log(n)$ y en el peor de los casos la altura es n , esto ultimo en este tipo de arboles ABB determina el tiempo mínimo y máximo para acceder a un nodo^[1] (el peor de los casos que es cuando el dato esta en una hoja).

Cuando se **inserta** o **elimina** un elemento en el árbol se lo hace siguiendo un orden. El caso en que el nodo a eliminar tiene uno o ningún hijo es trivial, cuando el nodo tiene dos hijos, se debe reemplazar el nodo eliminado por el sucesor inorden, esto es, el nodo que se debería visitar si se acabara de visitar el nodo eliminado (con recorrido inorden) esto resulta en el nodo más a la izquierda del subárbol derecho del nodo eliminado, otras implementaciones utilizan el predecesor inorden, lo que resulta en el nodo mas a la derecha del subárbol izquierdo.

Maneras de recorrer un árbol binario

Existen varias formas de recorrer un árbol binario siendo las más típicas Depth-First Search (DFS o recorrido en profundidad) y Breath-First Search (BFS o recorrido en anchura). En la primera (recorrido en profundidad) se puede realizar en preorden, inorden o postorden; para la segunda se recorren los nodos por niveles.

Para el método en profundidad, en el recorrido en preorden, primero se accede a la raíz, luego al nodo izquierdo y luego al derecho, si alguno de los nodos es un subárbol entonces se realiza el mismo procedimiento. Para el recorrido inorden, primero se accede al nodo izquierdo, luego a la raíz, y por ultimo al nodo derecho, este recorrido típicamente se usa en arboles de búsqueda binaria (ABB). Por ultimo en el recorrido postorden, primero se accede a ambos nodos, izquierdo y derecho, y luego a la raíz.

Para el método en anchura, se recorre el nodo en orden por niveles de arriba hacia abajo y de izquier-

^[1] Abdul Bari. (2018, Marzo 16). 10.1 AVL Tree - Insertion and Rotations.
https://www.youtube.com/watch?v=jDM6_TnYlqE&t=239s.

da a derecha, primero la raíz, luego todos los nodos, luego los nodos de los nodos, etc.

Por ejemplo, en el árbol de la figura 1.1, el recorrido utilizando los 4 métodos mencionados resultan como se ve en la tabla 1.

Recorrido	Nodos
Preorden	A - B - D - E - C - F - G
Inorden	D - B - E - A - C - F - G
Postorden	D - E - B - F - G - C - A
Nivel por nivel	A - B - C - D - E - F - G

Tabla 1: Recorridos de un árbol binario

Características de los arboles

Árbol lleno

Se dice que un árbol esta lleno si todas las hojas tienen el mismo nivel y todos los nodos anteriores tienen el número máximo de hijos (en un árbol binario 2) los arboles de la figura 1 están llenos, si en alguno de los dos faltase alguna hoja, no lo estarían.

Árbol completo

Se dice que un árbol esta completo, si todas sus hojas están llenas excepto el ultimo nivel, que debe estar casi-completo de izquierda a derecha, es decir, puede haber hojas vacías pero deben estar lo más a la derecha posible y solo en el ultimo nivel. En la figura 1 ambos arboles están completos. A un árbol incompleto también se le dice **desequilibrado**.

Árbol degenerado o patológico

Un árbol degenerado (también llamado árbol patológico) es un tipo especial de árbol en el que cada nodo tiene a lo sumo un hijo. Es decir, se comporta como una lista enlazada en lugar de un árbol ramificado. Esto ocurre por ejemplo al insertar datos ordenados en un árbol de búsqueda binaria

Árbol balanceado

Un árbol se dice balanceado si para cada nodo la diferencia de alturas entre el subárbol izquierdo y derecho (el factor de equilibrio) es pequeña, típicamente se toma menor o igual a 1 en valor absoluto.

Factor de equilibrio

El factor de equilibrio se define para cada nodo como la diferencia entre las alturas del subárbol derecho e izquierdo, como se muestra a continuación:

$$F_{\text{equilibrio}} = h_{\text{derecho}} - h_{\text{izquierdo}}$$

El reciproco, es decir, la diferencia de alturas del subárbol izquierdo y derecho de un nodo también vale, pero se debe respetar la convención elegida para todo el árbol, o tomar el valor absoluto.

Árbol AVL (Adelson-Velski y Landis)

Un árbol AVL, es una caso particular de un árbol de búsqueda binaria en el que cada vez que se inserta o elimina un nuevo elemento se lo hace de manera que el árbol resulte balanceado, esta es una mejora a los ABB ya que si se insertan los datos ordenados en el árbol ABB se degenera. Para que el árbol AVL resulte balanceado al insertar o eliminar un elemento se comprueba si el árbol resulta balanceado mediante el calculo del factor de equilibrio para cada nodo, en caso de que el factor de equilibrio no pertenezca a $[-1, 0, 1]$, se lo balancea.

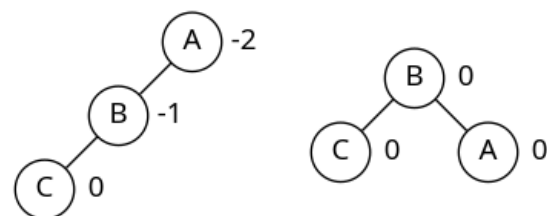
Si bien los arboles AVL son mucho más eficientes que los arboles ABB, tienen la particularidad de que por cada nivel pueden tener una mínima cantidad de nodos (en el peor de los casos) que no es la máxima, y en particular se verifica que para un árbol AVL de altura h (o h niveles) la cantidad mínima de nodos es $F(n)$, siendo F la serie de Fibonacci de n términos, es por esto que también a los arboles AVL se los conoce como **arboles de Fibonacci**.

Algoritmos de rotación

Las rotaciones se realizan para balancear arboles AVL. El punto clave es entender que las rotaciones se aplican solo a 3 nodos y modifican los nodos hijos. Se determina que un nodo debe ser balanceado calculando el factor de equilibrio de cada nodo, el primer nodo de abajo hacia arriba que esta desbalanceado debe ser balanceado.

Rotación simple a derecha (o LL ^[2])

En las rotaciones simples a derecha sobre un nodo, se rota ese nodo a su hijo izquierdo y en su lugar se toma el hijo derecho del nodo previo; en el lugar del nodo previo "asciende" su hijo derecho. En la figura 3.1 se muestra un árbol desbalanceado, cuyo primer nodo desbalanceado de abajo hacia arriba es A, ya que tiene un factor de equilibrio -2 lo cual en valor absoluto es mayor estricto que 1, en la figura 3.2 se muestra el resultado luego de realizar una rotación simple a derecha sobre el nodo A.



(1) Arbol desbalanceado

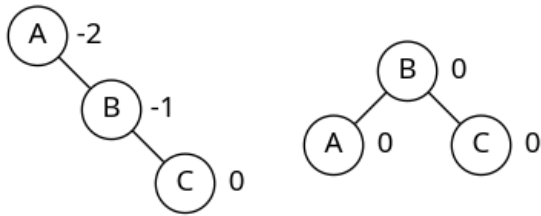
(2) Luego de RSD

Figura 3: Rotación simple a derecha

^[2] LL (por left-left en ingles) es porque el que causa el desbalanceo del nodo esta en el hijo izquierdo del hijo izquierdo.

Rotación simple a izquierda (o RR)

Las rotaciones simples a izquierda son análogas a las rotaciones simples a derecha pero en sentido contrario, en la figura 4.1 se muestra un ejemplo de un árbol desbalanceado y en la figura 4.2 se muestra el mismo árbol luego de realizar una rotación simple a derecha en el nodo A.

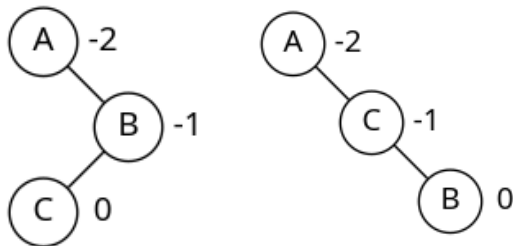


(1) Árbol desbalanceado (2) Luego de RSI

Figura 4: Rotación simple a izquierda

Rotación doble derecha (derecha-izquierda, RL)

En las rotaciones dobles a derecha se deben realizar dos rotaciones simples, en primer lugar a derecha y por último a izquierda. En la figura 5.1 se muestra un árbol desbalanceado, para balancearlo en primer lugar se realiza una rotación simple a derecha sobre el nodo B, resultando como en la figura 5.2.



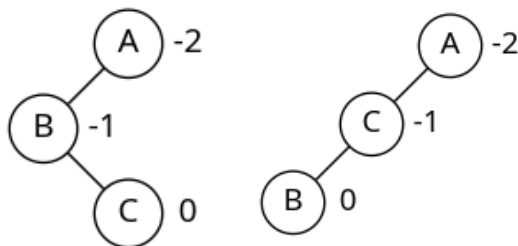
(1) Árbol desbalanceado (2) Luego de RSD sobre B

Figura 5: Rotación doble a derecha

Luego de la primer rotación se obtiene un nuevo árbol desbalanceado, pero que se balancea fácilmente mediante una rotación simple.

Rotación doble izquierda (izquierda-derecha, LR)

Estas rotaciones son similares a las dobles a derecha pero las rotaciones ocurren en sentido opuesto, la primera a izquierda y la segunda a derecha.



(1) Árbol desbalanceado (2) Luego de RSI sobre B

Figura 6: Rotación doble a izquierda

Árbol B

Los árboles B surgen como una forma de hacer más eficiente la lectura y escritura en disco de datos, en particular de decrementar el tiempo y la cantidad de bloques leídos de un disco a la hora de buscar un dato.

Lo particular de los árboles B en comparación a los árboles binarios vistos anteriormente es que tienen un número mayor de datos almacenados y un número mayor de hijos, pero se sigue la convención de los árboles de búsqueda binaria en que los datos a la izquierda son menores y a la derecha mayores, tanto en los nodos como en las claves. Esto se ve en la estructura de los nodos, un ejemplo se muestra en la figura 7, en el cual se ve una estructura de un árbol B de orden 3.

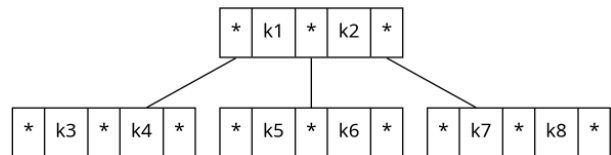


Figura 7: Estructura de un árbol B

El **orden** de un árbol B (se denota con la letra m) determina la cantidad máxima de hijos que pueden tener los nodos (y por consiguiente la cantidad máxima de claves, ya que es $m - 1$). Por ejemplo, en el árbol de la figura 7, el orden del árbol es 3, por lo que cada nodo puede tener un máximo de 3 hijos, y almacenar 2 claves.

Por definición, en los árboles B se deben cumplir las siguientes reglas:

1. Todos los nodos, salvo la raíz, deben tener al menos $m/2$ hijos o $(m/2) - 1$ claves.
2. Todas las hojas están en el mismo nivel.
3. El proceso de creación es de abajo hacia arriba.

Operaciones

Los árboles B tienen las mismas operaciones que los árboles binarios y aparte la operación de dividir y fusionar (las cuales son operaciones de balanceo). Al igual que los árboles AVL, al momento de insertar y eliminar datos del árbol ocurre un proceso de balanceo, pero en este caso siendo la operación más sencilla, ya que se trata de un procedimiento de un solo paso.

Búsqueda

La búsqueda es similar al árbol binario ya que las claves están ordenadas en cada nodo $k1 < k2 < \dots < kn$, en caso de que el dato a buscar este entre el rango de dos claves, por ejemplo el dato a buscar k , cumple $k2 < k < k3$, entonces se debe buscar en el hijo entre $k2$ y $k3$, luego se realiza el mismo procedimiento hasta llegar a las hojas.

Inserción

Para agregar un elemento, de manera análoga a arboles binarios, se busca la posición mediante comparación con las claves hasta llegar a la hoja, en caso de que haya lugar en la hoja simplemente se agrega, si no hay lugar se tiene que dividir la hoja y promover el elemento $((m-1)/2) + 1$ a un nivel superior, de manera que en la nueva hoja queden el número mínimo de elementos. En caso de que no haya más lugar en el nodo superior donde se promueve el elemento se realiza el mismo procedimiento.

Eliminación

Cuando se elimina un elemento, a diferencia de cuando se inserta, puede ocurrir en cualquier nodo, sea el nodo raíz, un nodo interno o un nodo hoja. Si el nodo del cual se quiere eliminar el elemento es una hoja, y al eliminar el elemento el número de claves sigue siendo mayor a la mínima, entonces el proceso de eliminación se detiene allí. Si el número es menor al número mínimo de elementos por nodo entonces se pide "prestado" al hermano izquierdo o derecho (dependiendo la implementación del algoritmo) en caso de que ninguno de los dos tenga o que tengan el número mínimo de elementos, se pide al de un nivel superior, en caso de que ocurra lo mismo con el del nivel superior, que no tiene elementos para prestar, entonces se pide al de un nivel superior, y así sucesivamente hasta la raíz, si la raíz tiene un solo elemento entonces se baja un nivel del árbol.

Árbol B+

Los arboles B+ son muy similares a los arboles B pero con la condición que los datos deben estar en las hojas, para esto se hace que las raíces sean referencias al dato verdadero que se encuentra en las hojas. Además se impone la condición de que todas las hojas deben estar conectadas, de esta forma todas las hojas formen una especie de lista enlazada, y como todos los datos se almacenan en las hojas, esta lista enlazada dispone de todos los datos almacenados en el árbol, esto permite recorrer secuencialmente todos los nodos, lo cual es una mejora con respecto a los arboles B, ya que se deben recorrerse recursivamente, nodo por nodo.

En la figura 8 se muestra un ejemplo de una estructura de un árbol B+, se puede ver la conexión entre las hojas y las copia de los nodos intermedios (k_1 y k_2).

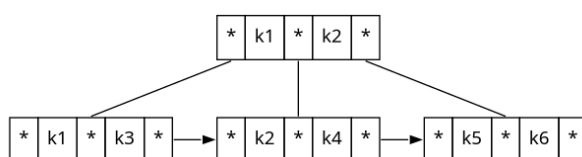


Figura 8: Estructura de un árbol B+

Operaciones

Las operaciones en general son similares a los arboles B, pero con algunas particularidades.

Búsqueda

En el caso de la búsqueda en un árbol B+, es similar a la búsqueda del árbol B, pero no se debe detener cuando se encuentre la clave en la pagina raíz o en un pagina interior, si no que se debe seguir por la rama derecha apuntada por esa clave, ya que la clave apuntada en primera instancia era una copia de la clave real almacenada en la hoja.

Inserción

El proceso de inserción en un árbol B+ es muy similar al de un árbol B, pero se diferencia en que cuando se inserta una nueva clave en un nodo lleno, esta se divide en dos, con la primera teniendo el número mínimo de claves y la segunda el número mínimo de claves sumada la nueva clave, el elemento que promociona a un nuevo nivel es una copia de la clave central.

Eliminación

En el caso de eliminar un elemento una vez encontrado, se elimina directamente de la hoja (todos los datos están en las hojas), si luego de eliminarlo la hoja queda con menos del mínimo de elementos por hoja ($m/2$) entonces hay que eliminar la hoja y redistribuir los elementos con las hojas hermanas, la clave copia en el nodo padre se debe eliminar también, si el nodo padre también queda con menos del mínimo de claves por hoja, entonces hay que realizar el mismo procedimiento.

Árbol heap

Los arboles heap son arboles con la particularidad de que se busca que siempre estén completos o casi completos y parcialmente ordenados, esto se hace insertando los elementos de izquierda a derecha por niveles, es por esto que también se le dicen casi-completos ya que los subárboles de las hojas se van completando a medida que ingresan los elementos.

Los arboles heap, tienen un orden diferente al de los arboles binarios, se ordenan por nivel. En los arboles heap de máximo la raíz es siempre mayor a los hijos, y entre hijos (o hermanos) no hay una relación particular como si pasa en los arboles binarios, en los cuales el izquierdo es siempre menor al hijo derecho, además, existen también los arboles heap de mínimo, en los cuales la raíz es siempre menor a los hijos.

En la figura 9 se ven dos ejemplos de arboles heap, en la figura 9.1, se ve un árbol heap de máximo, mientras que en la figura 9.2 se ve un árbol heap de mínimo.

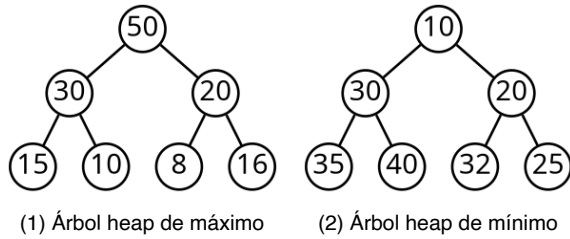


Figura 9: Tipos de árbol heap

Los árboles heap típicamente se almacenan en vectores, almacenando los valores por nivel, contando k desde 0, para un padre en posición k sus hijos se almacenan en la posición $(2*k)+1$ y $(2*k)+2$ del vector para el hijo izquierdo y derecho respectivamente, análogamente, si el hijo está en la posición h del vector, el padre estará en la posición $\text{floor}((h-1)/2)$. Almacenando los árboles en vectores permite acceder a los datos de forma directa. En la figura 10 se muestra como se almacena el árbol heap de máximo de la figura 9.1, como particularidad se ve que el primer elemento es la raíz, lo cual es el elemento más grande por ser heap de máximo, si fuera de mínimo este sería el menor elemento.

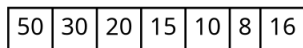


Figura 10: Árbol heap almacenado en vector

Para **insertar** elementos en un árbol heap se agrega siempre en la primera hoja vacía de izquierda a derecha en el último nivel del árbol, una vez insertado el elemento se compara e intercambia con el padre hasta cumplir con la definición de árbol heap (que la raíz sea mayor o menor a los hijos, es decir que los hijos sean árboles heap de por sí).

Para **eliminar** elementos de un árbol heap típicamente se elimina la raíz, para esto en primer lugar se intercambia con la última hoja (último nivel, última hoja contando de izquierda a derecha) y luego se compara la nueva raíz con los hijos intercambiando en cada caso con el mayor (o menor) hasta que el árbol vuelva a ser un heap, por último se elimina el dato a eliminar, es decir la última hoja, o en caso de estar almacenado en un vector se decrementa la longitud de este.

Como los árboles heap se completan por niveles, la altura del árbol es $\log(n)$, siendo n la cantidad de nodos en el árbol, esto es una mejora contra los árboles AVL que tienen una altura $\text{Fib}(n)$.

Algoritmo heapify

Heapify es un algoritmo utilizado para ordenar un árbol heap si la raíz del árbol viola la propiedad principal de un árbol heap (que sea mínimo o máximo a todos los hijos, dependiendo el tipo de heap, y que a su vez cada subárbol sea un árbol heap de por sí).

El algoritmo también se utiliza para convertir un árbol binario en un árbol heap, para esto se comienza en las hojas y se verifica que todos los subárboles cumplan que sean heaps, en caso de no serlo intercambia los valores hacia abajo hasta que se cumpla la condición de heap, estos pasos se repiten sucesivamente con todos los nodos hasta llegar a la raíz. En este caso el algoritmo tiene una complejidad de $O(n)$

A continuación se muestra una posible implementación recursiva del algoritmo heapify para un árbol heap de máximo almacenado en un vector, en este caso denominado `arr`, el lenguaje utilizado es python

```
def heapify(arr, n, i):
    # Se asume que la raíz es máxima
    max = i

    # posición de hijo izquierdo y derecho
    left = (2*i) + 1
    right = (2*i) + 2

    # se comprueba si hay hijo izquierdo y es
    # mayor a la raíz
    if left < n and arr[left] > arr[max]:
        max = left

    # se comprueba si hay hijo derecho y es
    # mayor a la raíz o al hijo izquierdo
    if right < n and arr[right] > arr[max]:
        max = right

    # si la raíz no es máxima se intercambia
    # con el máximo hijo, y se llama
    # recursivamente a heapify para reordenar
    # los hijos restantes
    if max != i:
        arr[i], arr[max] = arr[max], arr[i]
        heapify(arr, n, max)
```

Heapsort

El algoritmo de ordenamiento heapsort es un algoritmo que se basa en el algoritmo heapify, ya que para ordenar un vector, arma un árbol heap de máximo (o mínimo según la implementación) e intercambia el valor de la raíz con el último valor del vector, decrementando la capacidad del vector en un elemento, luego vuelve a armar un árbol heap con los elementos restantes e intercambia nuevamente el último elemento ahora con el ante-último del vector, y decrementando la cantidad de elementos en el vector, de esta forma al final del vector se van colocando los valores máximos ordenados, el algoritmo sigue hasta llegar al primer elemento del vector.

Colas con prioridad

Las colas con prioridad son una estructura de datos de tipo cola (o queue) pero tienen la particularidad que al momento de insertar un elemento se puede asignar una prioridad, de esta forma no se inserta siempre en un mismo lugar si no que se puede insertar con un orden en particular dependiendo de la prioridad que se asigne a el elemento a insertar.

Existen varias maneras de implementar una cola con prioridad, por ejemplo un vector, una lista enlazada o un árbol heap. El caso de la implementación con árbol heap es de las más eficientes, ya que en un árbol heap de máximo la raíz es la que tiene el valor máximo, y se puede imponer que este número sea la prioridad, de esta forma eliminar un elemento, es decir, desencolar el elemento con mayor prioridad es eliminar la raíz del árbol.