Árboles Rojo-Negro

Por Rodrigo Plauchú y Ángel Reyes

Un árbol rojo-negro es una estructura de datos abstracta. Específicamente es un tipo de árbol binario de búsqueda equilibrado. El creador de la estructura original fue Rudolf Bayer en 1972, posteriormente Leo J. Guibas y Robert Sedgewick realizaron un trabajo en el cual adquirió el nombre moderno.

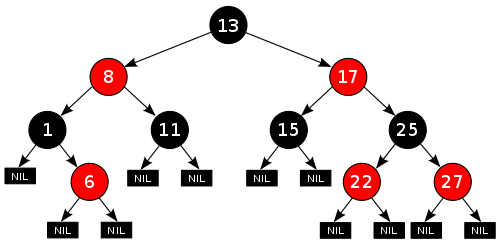
Estos árboles buscan mantener su altura o número de niveles lo más bajo posible. Algo que se consigue por medio de inserciones y rotaciones con las cuales se deben cumplir las siguientes características: en primer lugar, todo nodo es rojo o negro; en segundo lugar, la raíz siempre es negra; en tercer lugar, todas las hojas null son negras; en cuarto lugar, todo nodo rojo debe tener dos nodos hijos negros; en quinto y último lugar, cualquier camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros (Observe fig. 1). 

Fig.1

Al ser un caso especial de un árbol binario de búsqueda las operaciones que se pueden realizar son las mismas (inserciones, eliminaciones y búsquedas, estas últimas no requieren modificación alguna). Sin embargo, para insertar y eliminar se deben contemplar distintos casos que serán comentados más adelante. La complejidad algorítmica para estos tres métodos es O (log n), es importante mencionar que en cada inserción o borrado se requiere un espacio adicional del mismo tamaño O (log n). La búsqueda de información se realiza de la misma forma que un árbol binario de búsqueda normal. Recorre el árbol comparándose con el dato de los nodos. Si lo encuentra, lo regresa. Si llega a un null, el dato no está en el árbol. La inserción es parecida, pero en el árbol rojo-negro las propiedades que se tienen que respetar hacen que haya complicaciones. Para insertar un nuevo nodo lo haremos como se haría un en árbol binario de búsqueda convencional y se pinta de rojo, las modificaciones dependen del color de otros nodos a su alrededor. Procederemos a explicar las cinco formas o casos que se presentan al momento de insertar un nodo en un árbol rojo-negro.

Nota: Al hermano del padre de un nodo se podrá referir como tío nodo y al padre de ambos nodos como abuelo nodo.

Caso 1: Cuando el nodo insertado es la raíz. En este caso el nodo es pintado de negro (por propiedad uno) y como esto añade 2 hijos negros a cada lado se cumplen todas las propiedades.

Caso 2: Cuando el padre del nodo nuevo es negro. En este caso (dado que el nodo insertado es rojo) el árbol sigue siendo válido.

Caso 3: Cuando el padre y el tío del nodo nuevo son rojos. En este caso los padres son pintados de negro y el abuelo de rojo. De esta manera no se viola la propiedad cinco. Sin embargo, podría darse el caso en que el abuelo sea raíz y la propiedad dos sea violada o incluso la propiedad cuatro. En estos casos, (dado que el método es recursivo) en la siguiente iteración se comprueba que el abuelo sea válido. Si existe algún problema con la raíz se regresa al caso 1 y si hay algún problema con la propiedad cuatro, al caso 2 (Observe figura 2).

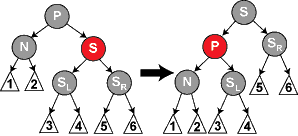
Fig. 2

Caso 4: Cuando el padre del nodo nuevo es rojo y el tío negro. Además, el nuevo nodo es el hijo derecho del padre que a su vez es el hijo izquierdo del abuelo. En este caso, se requiere realizar una rotación a la izquierda sobre el nodo nuevo, sin embargo, esto no resuelve el problema ya que ahora el hijo izquierdo (antes padre) debe ser evaluado. Es por eso que se en la siguiente iteración caiga en el caso 5 (Observe figura 3).

Fig. 3

Caso 5: Cuando el padre del nodo nuevo es rojo y el tío negro. Además, el nuevo nodo es el hijo izquierdo del padre que a su vez es el hijo izquierdo del abuelo. En este caso, se realiza una rotación a la derecha sobre el padre. El resultado es que el padre ahora tiene como hijos el nodo nuevo y además al abuelo inicial. El color del abuelo debe ser negro y el del padre rojo por lo que se intercambian los colores por lo que se satisfacen las propiedades cuatro y cinco (Observar figura 4).

Fig. 4

Ya explicado el proceso de inserción y sus casos procederemos a explicar el proceso de eliminación y sus seis casos.

Caso 1: Cuando se elimina un nodo padre rojo con un hijo. En este caso simplemente se elimina y se ajusta. Si el nodo fuera negro y el hijo rojo habría que ajustar el árbol y además realizar un cambio de color a negro.

Fig. 5

Caso 2: Cuando el hermano del nodo a eliminar es rojo y el padre negro. En este caso se invierten los colores del hermano y el padre, posteriormente se realiza una rotación a la izquierda del padre. Ahora el hermano pasa a ser el abuelo del nodo a eliminar y además ahora tiene un hermano negro. Todavía no se realiza la eliminación, por lo que en la siguiente iteración caerá en alguno de los casos 4, 5 o 6 (Observar figura 5).

Fig. 6

Caso 3: Cuando el padre del nodo a eliminar, el hermano y sus hijos son negros. En este caso simplemente se cambia el color del hermano a rojo. El problema es que ahora hay un desequilibrio en el árbol y violación de propiedades, para resolverlo se vuelven a llamar a los casos de eliminación comenzando desde el caso 1 (Observar figura 6).

Fig. 7

Caso 4: Cuando el padre del nodo a eliminar es rojo y su hermano e hijos negros. En este caso simplemente se cambia el color de el padre y el hermano, de esta manera no se desequilibra el árbol al momento de eliminar el nodo y sigue siendo válido (Observar figura 7).

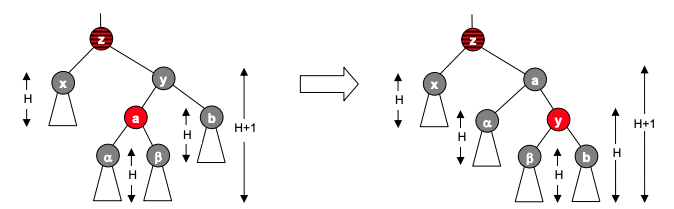
Caso 5: Cuando el hermano del nodo a eliminar es negro y este tiene un hijo izquierdo rojo y un hijo derecho negro, el padre puede ser de cualquier color. En este caso se realiza una rotación a la derecha sobre el hijo rojo del hermano y se intercambian sus colores. Ahora el hijo izquierdo se convierte en padre del hermano del nodo a eliminar, por lo que ahora se vuelve su hermano. El árbol está equilibrado, pero ahora el hermano tiene un hijo derecho rojo, por lo que cae en el caso 6 (Observar figura 8).

Fig. 8

Caso 6: Cuando el hermano del nodo a eliminar es negro, su hijo izquierdo de cualquier color y su hijo izquierdo rojo, el padre puede ser de cualquier color. En este caso se realiza una rotación a la izquierda sobre el hermano y se intercambian los colores del padre y el hermano, así como del hijo derecho del hermano que pasa a ser negro. De esta manera el árbol se encuentra equilibrado y cumpliendo todas las condiciones por lo que el ciclo termina (Observar figura 9).

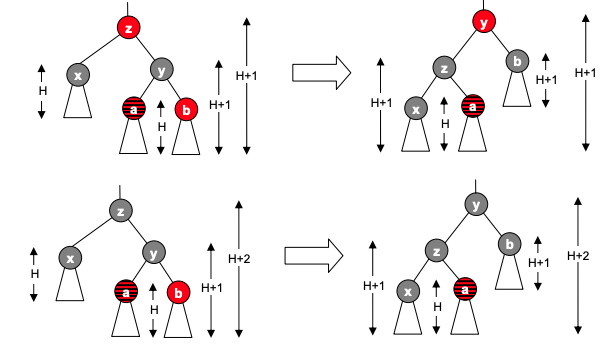
 Después de haber mencionado todos los posibles casos de inserción y eliminación de los árboles rojo-negro, procederemos a comparar ciertas características como complejidad, similitudes y diferencias con otros árboles. Como fue mencionado con anterioridad el árbol estudiado es de tipo binario y concretamente de búsqueda. Por lo anterior, se sabe que es posible realizar búsquedas, inserciones y eliminaciones, sin embargo, no comparten la misma complejidad en operaciones. Un árbol binario normalmente mantiene una complejidad de O (log n) al igual que el rojo-negro, la diferencia es que en el primero existen diferentes constantes que hacen que varíe como puede ser que no sea completo. En el caso del árbol binario de búsqueda, dado que no son árboles equilibrados puesto que no se necesita, su complejidad puede ser O (log2 n) ó O(n). Si hablamos de árboles equilibrados inevitablemente tendremos a los árboles AVL y los rojo-negro. Evidentemente estos comparten ciertas características como las rotaciones y la complejidad, ya que, en ambos casos hay una complejidad de O (log n). La principal diferencia que se encuentra entre estos es que en los AVL hay un factor de equilibrio, lo anterior para mantener la diferencia entre los subárboles izquierdo y derecho en no más de una unidad, y en el rojo-negro como hemos visto se busca mantener las cinco propiedades. Ahora, en los siguientes árboles probablemente encontremos mayores diferencias. El árbol 2-3 ya no es un árbol binario y mantiene una complejidad en promedio de O (log3 n) aunque para las búsquedas de O (log n), en estos también hay inserción y borrado, mas no rotaciones. Los montículos o heaps existen en diferentes representaciones, algo importante de estos es que son árboles binarios completos y por tanto en promedio tiene una complejidad de O (n log2 n), hay inserciones y eliminaciones bajo ciertas propiedades, mas no rotaciones. Para el segment tree o árbol de segmentos es importante recalcar que son árboles que específicamente guardan segmentos y permiten consultas, mas no modificaciones (inserciones, borrado y rotaciones), por lo que son estructuras estáticas y mantienen una complejidad de búsqueda de O (log n + k). Con conocimientos de lo que es un árbol binario de búsqueda y un heap pasamos a estudiar el treap, composición de un árbol con un heap, se definen como un tipo de árbol de búsqueda aleatorizado. En este árbol no solo están permitidas las operaciones ordinarias como la búsqueda, inserciones, rotaciones y eliminaciones, sino que también se permiten unas “operaciones masivas” que se refieren a la unión, intersección y diferencia de conjunto, a pesar de esto, se mantiene una complejidad de O (log n). Ahora comparemos un skip list o lista de saltos con los árboles, una lista de saltos es una estructura de datos basada en listas enlazadas que es posible ser comparada con árboles dada su eficiencia, ya que, la complejidad que mantiene es de O (log n) en sus operaciones de búsqueda, inserción y eliminación, más por ser listas enlazadas no existe la rotación. Pasamos a introducir al trie, árbol que permite la recuperación de claves (por retrieval en inglés) como si fuera un diccionario, permite inserciones, eliminaciones y evidentemente búsquedas, en donde es muy eficiente y presenta una complejidad de O(n). Si hablamos ahora de los árboles B, que se dividen en árboles B+, B\*, e incluso el counted B tree, también son árboles de búsqueda, son balanceados, pero no binarios, se pueden realizar operaciones de búsqueda, inserción y eliminación, y aunque se es posible que se tenga que volver a equilibrar en algún momento no hay rotaciones, hay rebalanceo, su complejidad es de O (log n). Finalmente, compararemos a los árboles R, que se dividen a su vez en árboles R+ y R\*, son árboles muy parecidos a los B, concretamente se enfocan en almacenar información multidimensional como coordenadas, permite inserciones, eliminaciones y búsquedas, no hay rotaciones, en su mejor caso su complejidad es de O (log n) y en promedio es de O (M \* logM n).

Fig. 9

Podemos concluir que los árboles rojo-negro son una estructura de datos eficiente y con gran utilidad dentro de la computación, de hecho, tiene muchas aplicaciones, entre ellas están: el desarrollo de aplicaciones para uso en tiempo real, creación de nuevas estructuras de datos, particularmente en la geometría computacional, y grandes ventajas en la programación funcional. Por lo cual, podemos comentar que su única desventaja podría ser la complejidad al implementarse o entenderse, sin embargo, es una gran opción para el manejo de datos digna de estudiarse.

Bibliografía

Todas las imágenes fueron tomadas de Wikipedia.

Árbol rojo-negro. (2019, October 22). Retrieved from [https://es.wikipedia.org/wiki/Árbol\_rojo-negro#Usos\_y\_ventajas](https://es.wikipedia.org/wiki/%C3%81rbol_rojo-negro#Usos_y_ventajas).

Definición de un árbol Rojinegro[PDF file]. Retrieved from

<https://www.infor.uva.es/~cvaca/asigs/doceda/rojonegro.pdf>