Tercera entrega

Juan Esteban Murcia y David Moreno

Red Black Trees

Un árbol rojo-negro o RBT (por sus siglas en inglés) es una estructura abstracta de datos que amplía los arboles binarios de búsqueda, con el objetivo de lograr un balanceo óptimo dentro del árbol, para lograr las tres funciones elementales de los árboles búsqueda, inserción y remoción de elementos en una complejidad algorítmica de O(log N), siendo N el número de nodos en el árbol, para esto se le agrega una característica nueva a los nodos del árbol, el color y se plantean 5 reglas para lograr el objetivo principal de esta estructura de datos:

1. Todo nodo es o rojo o negro.
2. La raíz del árbol es negra
3. Un nodo rojo no puede tener padre rojo o hijos rojos
4. Cualquier camino desde la raíz hasta un hijo de una hoja (Nil) tiene la misma cantidad de nodos negros
5. Los hijos de las hojas (Nil) son negros

Estas propiedades nos dan ciertos lemas, que ayudan al análisis algorítmico de esta estructura de datos, estos son:

1. La cantidad de nodos negros en el camino de un nodo arbitrario N, a un Nil, denotada por Bh(N) es mayor o igual a la mitad de la altura del árbol h. Bh(N) >= h/2
2. Un RBT con N nodos tiene altura h <= 2\*Log(N+1)

Para la correcta implementación de un RBT, se tiene poner en consideración las tres funciones elementales.

**Búsqueda:**

Como los RBT tienen las mismas características de un árbol de búsqueda binaria y las nuevas restricciones no violan la propiedad de que a la derecha de un nodo con un elemento k, hay elementos menores a k, y a la derecha hay elementos mayores a k, entonces la búsqueda en un RBT es igual a la búsqueda estándar en un árbol de búsqueda binario.

**Inserción:**

Aquí es donde el procedimiento cambia radicalmente con respecto a un árbol de búsqueda binario estándar, .ya que un nodo será insertado de forma estándar con el color rojo, si el padre del nodo recién insertado es negro, la inserción habrá acabado, ya que no se estará violando ninguna propiedad y si el nodo insertado es la raíz, su color se cambiara a negro y ya, así que el problema es cuando el nodo insertado tenga padre rojo, esto se va a solucionar con rotaciones y re-coloreado, entraremos a analizar varios casos:

Caso 1: si el tío del nodo insertado (x) es rojo, nótese que el abuelo de x ha de ser negro por la propiedad (3), lo que se va a hacer es cambiar el color del padre y tío de x a negro, y del abuelo de x a rojo y se realizara recursión sobre el abuelo de x.

caso 2: si el tío de x es negro, hay cuatro configuraciones dentro de este caso, (I) que el padre de x, p, sea hijo izquierdo del abuelo de x, g, y que x sea hijo izquierdo de p. (II) que p sea hijo izquierdo de g y x sea hijo derecho de p. (III) que p sea hijo derecho de g y x sea hijo derecho de p. (IV) que p sea hijo derecho de g y x sea hijo izquierdo de p.

para (I) se realizará una rotación a la derecha con respecto a g y se cambiaran los colores de g y p. Para (II) se realizará una rotación a la izquierda con respecto a p y se aplicará (I) sobre x. Para (III) se realizará una rotación a la izquierda con respecto a g y se cambiaran los colores de g y p. Y para (IV) se realizará una rotación a la derecha con respecto a p, y luego se aplicará (III) sobre x.

**Eliminación:**

Lógicamente la propiedad que se puede afectar a la hora de insertar un elemento es la propiedad (3), pero a la hora de eliminar un elemento, el problema es afectar la propiedad (4), así que la solución para mantener (4) es que al eliminar un nodo negro vamos a marcar uno de sus hijos negros como doble negro (así se mantiene parcialmente la propiedad (4)) y el problema es ahora convertir ese nodo doble negro a un nodo negro.

Lo primero es realizar una eliminación estándar, nótese que siempre vamos a terminar eliminando un nodo con un solo hijo o sin hijos, ya que cuando se va a eliminar un nodo con 2 hijos, su valor es cambiado con el valor del predecesor (el valor más grande de los valores más pequeños a p) o por el sucesor (el valor más pequeño de los valores más grandes a p) y esto no viola ninguna propiedad de los RBT, y se elimina el nodo del predecesor/sucesor teniendo como máximo un hijo, Así que solo consideremos un hjo como máximo.

Sean v el nodo que se va a eliminar, y u el hijo de v que lo va a reemplazar. Si u es rojo o v es rojo, convertimos u en negro (si u ya es negro no estamos violando ninguna propiedad, y si u es rojo, pues v ha de ser negro y al ser eliminado se restaura la propiedad (4)) se elimina v, y u pasa a ser el nuevo hijo del padre de v.

Si tanto v como u son negros, (si u es Nil, por (5) u es negro) eliminamos v, y u se convierte en doble negro (para mantener trivialmente (4)) y realizaremos el siguiente proceso mientras u siga siendo doble negro y no sea la raíz.

Si el hermano de v, s, es negro y al menos uno de sus hijos es rojo realizamos una rotación sobre el padre de s (rotación a izquierda si s es hijo derecho y viceversa) esto nos genera 3 situaciones. (I) si s es el hijo izquierdo de su padre, y el hijo rojo de s, r, es hijo izquierdo de s, o ambos hijos de s son rojos. (II) si s es el hijo izquierdo de su padre, y r es hijo derecho de s. (III) s es hijo derecho de su padre y r es hijo derecho de s, o ambos hijos de s son rojos. Y (IV) s es hijo derecho de su padre, y r es hijo izquierdo de s. Para (I) realizamos una rotación a derecha sobre el padre de s. Para (II), realizamos una rotación a izquierda sobre s, y una rotación a derecha sobre el antiguo padre de s. Para (III) realizamos una rotación a izquierda sobre el padre de s. Y para (IV) realizamos una rotación a derecha sobre s, y luego una rotación a izquierda sobre el antiguo padre de s.

Si s es negro y los dos hijos de s son negros, cambiaremos el color de s a rojo, quitaremos la marca de doble negro a u, si el padre de s es negro llamaremos a recursión sobre el padre de s, si es rojo, cambiaremos el color del padre de s a negro.

Si s es rojo, realizaremos una rotación sobre el padre de s (rotación izquierda si s es hijo derecho, y viceversa) cambiaremos el color de s a negro, el color del antiguo padre de s a rojo y llamamos a recursión sobre el nodo que sigue marcado como doble negro (ya que esta rotación nos trasladara al primer o al segundo caso).

Y el último caso es que, u sea la nueva raíz, en lo que convertiremos u en negro y terminaremos.

Durante la realización de este proyecto nos enfrentamos a diversos inconvenientes, ya que son demasiados casos lo que toca considerar tanto para la eliminación, como para la inserción de elementos en el árbol, sin embargo, fue interesante realizar esta proyecto, ya que nos permitió ver más a fondo las miles de formas en las que dado un problema, en este caso como realizar un balances óptimo sobre un árbol, se pueden obtener soluciones creativas como esta, ya que difiere de otras técnicas de balanceo como AVL.

Bibliografía:

***Introduction to Algorithms*** by [Thomas H. Cormen](https://en.wikipedia.org/wiki/Thomas_H._Cormen), [Charles E. Leiserson](https://en.wikipedia.org/wiki/Charles_E._Leiserson), [Ronald L. Rivest](https://en.wikipedia.org/wiki/Ron_Rivest), and [Clifford Stein](https://en.wikipedia.org/wiki/Clifford_Stein).

ChiragAcharya. (N/A). Red Black Trees | Set 2 (insert). GeeksByGeeks, https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2 /