Red-Black Tree

Alumno: Josue Samuel Philco Puma

1 de octubre de 2024

Resumen

El siguiente documento contiene la implementación de la estructura llamada ***Red-Black Tree*** con un tiempo medio de accesos desde la raíz hasta un nodo aleatorio. También estará junto con el análisis de complejidad computacional del algoritmo.

1. Red-Black Tree

Red-Black Tree es un árbol binario de búsqueda autoequilibrado, en el que los nodos tienen un atributo adicional: un color, ya sea **rojo** o **negro**. El objetivo principal de estos árboles es mantener el equilibrio durante las inserciones y las eliminaciones, lo que va a garantizar una recuperación rápida y manipulación eficiente de datos. Este árbol cuenta con propiedades que debe cumplir siempre y son:

- **Color del nodo:** Este debe ser de color **rojo** o **negro**.
- **Propiedad de la raíz:** La raíz del árbol siempre es de color negro.
- **Propiedad del rojo:** Estos nodos no pueden tener hijos rojos (no puede haber dos nodos rojos consecutivos en la ruta).
- **Propiedad del negro:** Cada ruta desde un nodo hasta sus nodos nulos descendientes tienen la misma cantidad de nodos negros.
- **Propiedad de la hoja:** Todas las hijas (NULL) son negras.

1.1. Implementación de un Red-Black Tree

Implementar esta estructura es un poco tedioso por las propiedades que debe cumplir, especialmente al hacer eliminaciones donde requerimos mover unos nodos para que se mantenga el equilibrio y cumplir con sus propiedades. A continuación estará la implementación de cada función del **Red-Black Tree**.

1.1.1. Función Left Rotate

Una rotación a la izquierda en el nodo **x** mueve al nodo **x** hacia abajo a la izquierda y su hijo derecho que es el nodo **y** va hacia arriba ocupando el lugar del nodo **x**.

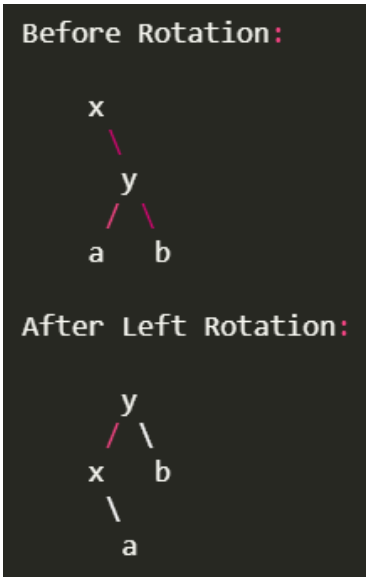

Figura 1: Visualización de la rotación

Ahora, la implementación de esta función del árbol sería de la siguiente manera:

```
void RotateLeft(Node* nodeX) {
    if (nodeX == nullptr || nodeX->right == nullptr) {
        return;
    }

    Node* nodeY = nodeX->right;
    nodeX->right = nodeY->left;

    if (nodeY->left != nullptr) {
        nodeY->left->parent = nodeX;
    }

    nodeY->parent = nodeX->parent;

    if (nodeX->parent == nullptr) {
        root = nodeY;
    }
    else if (nodeX == nodeX->parent->left) {
        nodeX->parent->left = nodeY;
    }
    else {
        nodeX->parent->right = nodeY;
    }
    nodeY->left = nodeX;
    nodeX->parent = nodeY;
}
```

1.1.2. Función Right Rotate

Una rotación a la derecha en el nodo **x** mueve al nodo **x** hacia abajo a la derecha y su hijo izquierdo que es el nodo **y** va hacia arriba ocupando el lugar del nodo **x**.

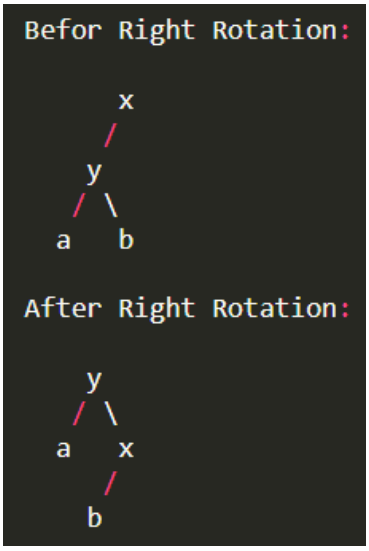

Figura 2: Visualización de la rotación

La implementación de esta función es similar a la función **Left Rotate**, solo cambiando los nodos y sería de la siguiente manera:

```
void RotateRight(Node* nodeY) {
    if (nodeY == nullptr || nodeY->left == nullptr) {
        return;
    }
```

```
5
6                       Node* nodeX = nodeY->left;
7                       nodeY->left = nodeX->right;
8
9                       if (nodeX->right != nullptr) {
10                          nodeX->right->parent = nodeY;
11                      }
12
13                      nodeX->parent = nodeY->parent;
14
15                      if (nodeY->parent == nullptr) {
16                          root = nodeX;
17                      }
18                      else if (nodeY == nodeY->parent->left) {
19                          nodeY->parent->left = nodeX;
20                      }
21                      else {
22                          nodeY->parent->right = nodeX;
23                      }
24                      nodeX->right = nodeY;
25                      nodeY->parent = nodeX;
26                  }
```

1.1.3. Función Insert

Esta función va a ser la encargada de insertar nuevas claves, este proceso implica dos pasos: Realizar la inserción como un el árbol de búsqueda binaria seguido de la reparación para evitar romper las propiedades del **Red-Black Tree**.

Ahora, implementando la función **Insert** es como hacer en un árbol binario, solo que al final debemos reparar el árbol en caso este rompiendo alguna de las propiedades, por lo que la implementación quedaría de la siguiente manera:

```
1                   void RedBlackTree::Insert(int value) {
2                       Node* newNode = new Node(value);
3                       Node* nodeY = nullptr;
4                       Node* nodeX = root;
5
6                       while (nodeX != nullptr) {
7                           nodeY = nodeX;
8                           if (newNode->data < nodeX->data) {
9                               nodeX = nodeX->left;
10                          }
11                          else {
12                              nodeX = nodeX->right;
13                          }
14                      }
15
16                      newNode->parent = nodeY;
17
18                      if (nodeY == nullptr) {
19                          root = newNode;
20                      }
21                      else if (newNode->data < nodeY->data) {
22                          nodeY->left = newNode;
23                      }
24                      else {
25                          nodeY->right = newNode;
26                      }
27
28                      InsertFix(newNode);
29                  }
```

Ahora, en la parte final encontramos una función **InsertFix**, esta se va a encargar de realizar las reparaciones necesarias al árbol garantizando que cumpla todas las propiedades mencionadas. Por lo que su implementación sería de la siguiente manera:

```
void InsertFix(Node* nodeZ) {
    while (nodeZ != root && nodeZ->parent->color == RED) {
        if (nodeZ->parent == nodeZ->parent->parent->left) {
            Node* nodeY = nodeZ->parent->parent->right;
            if (nodeY != nullptr && nodeY->color == RED) {
                nodeZ->parent->color = BLACK;
                nodeY->color = BLACK;
                nodeZ->parent->parent->color = RED;
                nodeZ = nodeZ->parent->parent;
            }
            else {
                if (nodeZ == nodeZ->right->parent) {
                    nodeZ = nodeZ->parent;
                    RotateLeft(nodeZ);
                }
                nodeZ->parent->color = BLACK;
                nodeZ->parent->parent->color = RED;
                RotateRight(nodeZ->parent->parent);
            }
        }
        else {
            Node* nodeY = nodeZ->parent->parent->left;
            if (nodeY != nullptr && nodeY->color == RED) {
                nodeZ->parent->color = BLACK;
                nodeY->color = BLACK;
                nodeZ->parent->parent->color = RED;
                nodeZ = nodeZ->parent->parent;
            }
            else {
                if (nodeZ == nodeZ->parent) {
                    nodeZ = nodeZ->parent;
                    RotateRight(nodeZ);
                }
                nodeZ->parent->color = BLACK;
                nodeZ->parent->parent->color = RED;
                RotateLeft(nodeZ->parent->parent);
            }
        }
    }
    root->color = BLACK;
}
```

1.1.4. Función Searching

Esta función como bien lo indica se encarga de buscar una clave en el árbol, aca las propiedades no juegan mucho ya que solamente buscamos la existencia de un nodo. La función quedaría implementada de la siguiente manera:

```
Node* SearchHelper(Node* node, int data, int& comparaciones) {
    comparaciones++;

    if (node == nullptr) {
        return node;
    }
```

```
        else if (data == node->data) {
            return node;
        }

        if (data < node->data) {
            return SearchHelper(node->left, data,
                comparaciones);
        }

        return SearchHelper(node->right, data,
            comparaciones);
    }
```

Esta función va a ejecutarse de forma recursiva para ir buscando en los hijos derechos o en los hijos izquierdos.

1.1.5. Función Remove

Esta función como indica su nombre se encarga de elimnar un nodo del árbol. Al igual que la función **Insert**, si eliminamos un nodo podemos romper las propiedades del árbol por lo que necita también una reparación.

Implementando la función **Remove** quedaría de la siguiente manera:

```
void RedBlackTree::remove(int value) {
    Node* nodeZ = root;

    while (nodeZ != nullptr) {
        if (value < nodeZ->data) {
            nodeZ = nodeZ->left;
        }
        else if (value > nodeZ->data) {
            nodeZ = nodeZ->right;
        }
        else {
            DeleteNode(nodeZ);
            return;
        }
    }

    std::cout << "Nodo con valor " << value << " no
        encontrado en el arbol." << std::endl;
}
```

Las condiciones inicales es para buscar el nodo que queremos eliminar buscando en los hijos derechos e izquierdos. En la última condición encontramos a la función **DeleteNode**, esta hará el proceso de la eliminación del nodo. Por lo que la implementación queda de la siguiente manera:

```
void DeleteNode(Node* nodeZ) {
    if (nodeZ == nullptr) {
        return;
    }

    Node* nodeY = nodeZ;
    Node* nodeX = nullptr;
    Color y_color_original = nodeY->color;

    if (nodeZ->left == nullptr) {
        nodeX = nodeZ->right;
        Transplant(nodeZ, nodeZ->right);
    }
    else if (nodeZ->right == nullptr) {
        nodeX = nodeZ->left;
        Transplant(nodeZ, nodeZ->left);
```

```
            }
            else {
                nodeY = Minimun(nodeZ->right);
                y_color_original = nodeY->color;
                nodeX = nodeY->right;

                if (nodeY->parent == nodeZ) {
                    if (nodeX != nullptr) {
                        nodeX->parent = nodeY;
                    }
                }
                else {
                    if (nodeX != nullptr) {
                        nodeX->parent = nodeY->parent;
                    }

                    Transplant(nodeY, nodeY->right);

                    if (nodeY->right != nullptr) {
                        nodeY->right->parent = nodeY;
                    }

                    nodeY->right = nodeZ->right;

                    if (nodeY->right != nullptr) {
                        nodeY->right->parent = nodeY;
                    }
                }

                Transplant(nodeZ, nodeY);
                nodeY->left = nodeZ->left;

                if (nodeY->left != nullptr) {
                    nodeY->left->parent = nodeY;
                }

                nodeY->color = nodeZ->color;
            }

            if (y_color_original == BLACK && nodeX != nullptr)
                {
                DeleteFix(nodeX);
            }

            delete nodeZ;
        }
```

Por último, tenemos una última función que se llama **DeleteFix**, esta solo se ejecutará cuando se compruebe que estamos las propiedades del árbol, de caso contrario se elimina el nodo de forma correcta, la función queda implementada de la siguiente manera:

```
        void DeleteFix(Node* nodeX) {
            while (nodeX != root && nodeX != nullptr && nodeX->
                color == BLACK) {
                if (nodeX == nodeX->parent->left) {
                    Node* nodeW = nodeX->parent->right;
                    if (nodeW->color == RED) {
                        nodeW->color = BLACK;
                        nodeX->parent->color = RED;
                        RotateLeft(nodeX->parent);
                        nodeW = nodeX->parent->right;
                    }

                    if ((nodeW->left == nullptr || nodeW->left
```

```
                    ->color == BLACK) && (nodeW->right ==
                    nullptr || nodeW->right->color == BLACK)
                    ) {
                    nodeW->color = RED;
                    nodeX = nodeX->parent;
                }
                else {
                    if (nodeW->right == nullptr || nodeW->
                        right->color == BLACK) {
                        if (nodeW->left != nullptr) {
                            nodeW->left->color = BLACK;
                        }
                        nodeW->color = RED;
                        RotateRight(nodeW);
                        nodeW = nodeX->parent->right;
                    }

                    nodeW->color = nodeX->parent->color;
                    nodeX->parent->color = BLACK;

                    if (nodeW->right != nullptr) {
                        nodeW->right->color = BLACK;
                    }

                    RotateLeft(nodeX->parent);
                    nodeX = root;
                }
            }
            else {
                Node* nodeW = nodeX->parent->left;
                if (nodeW->color == RED) {
                    nodeW->color = BLACK;
                    nodeX->parent->color = RED;
                    RotateRight(nodeX->parent);
                    nodeW = nodeX->parent->left;
                }

                if ((nodeW->right == nullptr || nodeW->
                    right->color == BLACK) && (nodeW->left
                    == nullptr || nodeW->left->color ==
                    BLACK)) {
                    nodeW->color = RED;
                    nodeX = nodeX->parent;
                }
                else {
                    if (nodeW->left == nullptr || nodeW->
                        left->color == BLACK) {
                        if (nodeW->right != nullptr) {
                            nodeW->right->color = BLACK;
                        }
                        nodeW->color = RED;
                        RotateLeft(nodeW);
                        nodeW = nodeX->parent->left;
                    }

                    nodeW->color = nodeX->parent->color;
                    nodeX->parent->color = BLACK;

                    if (nodeW->left != nullptr) {
                        nodeW->left->color = BLACK;
                    }

                    RotateRight(nodeX->parent);
```

```
                    nodeX = root;
                }
            }
        }
        if (nodeX != nullptr) {
            nodeX->color = BLACK;
        }
    }
```

1.1.6. Funciones Adicionales (Minimun y Transplant)

Estas funciones suelen ser para los árboles balanceados, y **Red-Black Tree** no es la excepción.

- La función **Transplant** esta encargada de reemplazar un subárbol con otro subárbol, esto es muy útil en la operación de eliminación de nodos.

```
void Transplant(Node* nodeU, Node* nodeV) {
    if (nodeU->parent == nullptr) {
        root = nodeV;
    }
    else if (nodeU == nodeU->parent->left)
        {
        nodeU->parent->left = nodeV;
    }
    else {
        nodeU->parent->right = nodeV;
    }

    if (nodeV != nullptr) {
        nodeV->parent = nodeU->parent;
    }
}
```

- La función **Minimun** va a buscar el nodo con el valor más pequeño en un subárbol donde la raíz es un parámetro.

```
Node* Minimun(Node* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}
```

1.1.7. Gráfico de medida del tiempo medio de accesos.

Como actividad de esta estructura, podemos usarlo para medir cuanto tiempo en promedio esta tomando partir desde la raíz hasta un nodo aleatorio en el árbol, por lo que acá estaría ejecutándose la función **Search** para realizar la operación, por lo que la función main estaría de la siguiente manera:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include "./RedBlackTree.cpp"

int main() {
    std::ofstream file("Resultados.csv");
    file << "Numero de Claves,Comparaciones
        Promedio\n";

    std::vector<int> keys;
```

```
    std::srand(std::time(0));

    for (int n = 100; n <= 10000; n += 100) {
        RedBlackTree RBTree;
        keys.clear();

        for (int i = 1; i <= n; i++) {
            keys.push_back(i);
            RBTree.Insert(i);
        }

        int comparaciones_totales = 0;
        int repeticiones = 100;

        for (int i = 0; i < repeticiones; i++) {
            int claves_random = keys[std::rand() %
                n];
            int comparaciones = 0;
            RBTree.Search(claves_random,
                comparaciones);
            comparaciones_totales += comparaciones;
        }

        double promedio_comparaciones =
            comparaciones_totales / (double)
            repeticiones;
        std::cout << "Numero de claves: " << n << "
             | Comparaciones promedio: " <<
            promedio_comparaciones << std::endl;
        file << n << "," << promedio_comparaciones
            << "\n";
    }

    file.close();
    return 0;
}
```

Esto simplemente el promedio que vaya obteniendo en cada comparación se pone en un archivo csv con el número de claves que han ido al árbol. Una vez teniendo el archivo pordemos generar la gráfica con **Python**:

```
import matplotlib.pyplot as plt
import pandas as pd

datos = pd.read_csv('Resultados.csv')
print(datos.columns)
datos.columns = datos.columns.str.strip()

plt.figure(figsize=(10, 6))
plt.plot(datos['Numero de Claves'], datos['
    Comparaciones Promedio'], marker='o', linestyle=
    '-', color='b', label='Comparaciones promedio')
plt.xlabel('Numero de claves')
plt.ylabel('Comparaciones promedio')
plt.title('Comparaciones promedio vs Numero de
    claves en Red-Black Tree')
plt.legend()
plt.grid(True)
plt.savefig('grafico_comparaciones.png')
plt.show()
```

Entonces, nos dará la siguiente imagen que representa el **Promedio de las Comparaciones vs Número de Claves**.

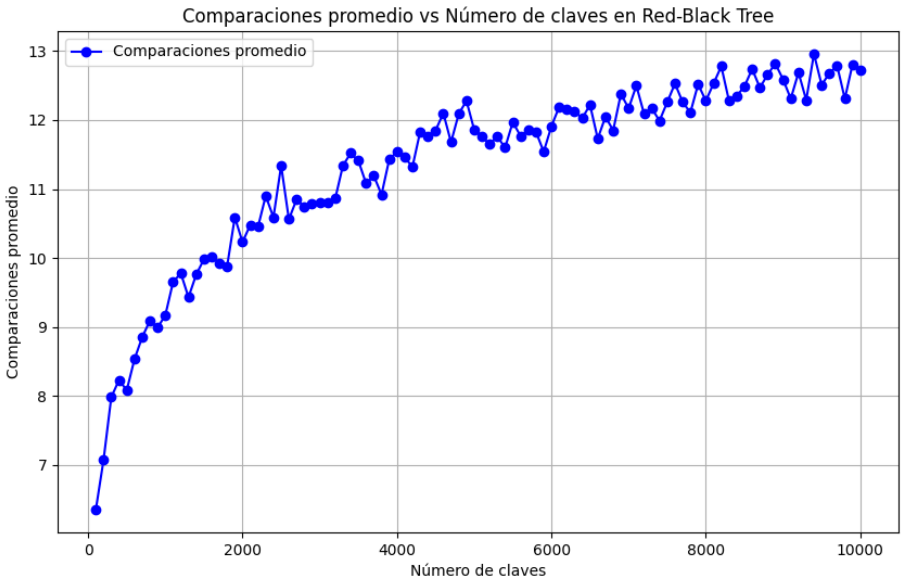

Figura 3: Gráfica generada de Promedio

2. Análisis de Complejidad Computacional

Ahora pasando con la complejidad del **Red-Black Tree** debemos considerar toda operación que influye en el árbol.

- **Altura del árbol:** Garantiza que la altura del árbol sea siempre limitada por $2\log(n+1)$ donde **n** es el número de nodos. Este balanceo permite que las operaciones principales tengan una complejidad de $O(\log n)$.
- **Inserción:** Este proceso implica varios pasos que van a contribuir a su complejidad:
  - La búsqueda para el punto de inserción sigue las propiedades de un árbol binario de búsqueda. Esto implica recorrer el árbol desde la raíz hasta una hoja, lo cual tomará una complejidad de $O(\log n)$.
  - Insertar como un nodo rojo tomará un tiempo de $O(1)$ que es constante.
  - Después de hacer la inserción, puede ser necesario hacer ajustes para restaurar las propiedades. Esto incluye **Recolorear los nodos** que toma tiempo constante $O(1)$ y realizar las rotaciones para mantener el balanceo del árbol que también toma tiempo constante $O(1)$.
  - En el peor de los casos, se podría tener un número de recoloreos y rotaciones a lo largo del camino que es proporcional a la altura $O(\log n)$.

  En conclusión, la operación de inserción considerando todas las operaciones sería un total de $O(\log n)$
- **Búsqueda:** Esta operación es muy similar al de un árbol binario de búsqueda, los pasos a realizar sería:
  - Recorrer el árbol desde la raíz y seguir el camino en función al valor que se busca. Como el árbol se encuentra balanceado tomará un tiempo de $O(\log n)$.
  - Comparar los nodos para ir por derecha o izquierda tiene un tiempo constante $O(1)$.

  En conclusión, esta operación tiene una complejidad de $O(\log n)$.
- **Eliminación:** Esta operación es más compleja debido a todos los ajustes que se deben realizar:

- Buscar el nodo que vamos a eliminar es como la operación de **Búsqueda**, por lo que toma un tiempo de $O(\log n)$.
- Eliminar físicamente el nodo tiene tiempo constante $O(1)$ si es que tiene 0 o un hijo. Si este tiene a sus dos hijos, se encuentra al sucesor inorder y se intercambia por el nodo que se va a eliminar por lo que toma un tiempo $O(\log n)$.
- Recolorear y hacer rotaciones es necesario si es que el nodo que se elimina es de color negro, los ajustes están limitados a la altura del árbol, por lo que tiene una complejidad de $O(\log n)$.

En conclusión, la operación **Eliminar** tiene un tiempo de $O(\log n)$.

- **Rotaciones:** Esta operación es utlizada para balancear el árbol, solo mueve nodos a posiciones diferentes. Por lo tanto, este paso toma tiempo constante $O(1)$.

En resumen, con todas las operaciones definidas quedaría de la siguiente manera la complejidad de cada operación:

Operación	**Complejidad en el peor caso**
Inserción	$O(\log n)$
Búsqueda	$O(\log n)$
Eliminación	$O(\log n)$
Rotaciones	$O(1)$

Cuadro 1: Complejidad de operaciones en un Árbol Red-Black

3. Conclusión del Red-Black Tree

El Red-Black Tree es una estructura de datos altamente eficiente que combina las ventajas de un árbol binario de búsqueda con un algoritmo de balanceo automático, lo que lo convierte en una opción ideal para aplicaciones donde la inserción, eliminación y búsqueda de datos deben realizarse rápidamente. Al mantener una altura $O(\log n)$ en el peor de los casos, el RB Tree garantiza un rendimiento eficiente, incluso en escenarios con grandes volúmenes de datos.

Es una herramienta poderosa para mantener el equilibrio en estructuras de datos dinámicas, permitiendo operaciones rápidas en el peor de los casos y siendo clave para garantizar un rendimiento robusto en sistemas complejos.