Árvore Binária de Busca

Uma árvore binária de busca é uma estrutura de dados que permite que os dados sejam inseridos, buscados e removidos de forma eficiente. A árvore é mantida de forma que para cada nó, o valor do nó seja maior que todos os valores dos nós descendentes à esquerda e menor que todos os valores dos nós descendentes à direita. A complexidade assintótica da árvore binária de busca é O(log n), onde *n* é o número de nós na árvore.

Árvore AVL

Uma árvore AVL é uma árvore de busca balanceada, ou seja, uma árvore binária na qual cada nó tem um campo que indica o balanceamento desse nó, que pode ser um número inteiro entre -1 e 1. O balanceamento é calculado como a diferença entre a altura da subárvore esquerda e da subárvore direita. Se o balanceamento for -1, isso significa que a subárvore esquerda é uma unidade mais alta que a subárvore direita; se o balanceamento for 0, as duas subárvores têm a mesma altura; e se o balanceamento for 1, a subárvore direita é uma unidade mais alta que a subárvore esquerda.

Árvore Rubro-Negra

Como uma árvore binária de busca, uma árvore rubro-negra mantém um conjunto de valores ordenados, mas diferentemente de uma árvore binária de busca comum, uma árvore rubro-negra insere e remove de forma inteligente para assegurar que a árvore permaneça aproximadamente balanceada. Isto é, a altura de qualquer nó na árvore rubro-negra é O(log n), onde *n* é o número de nós na árvore.

Ela segue as seguintes regras:

1. Uma árvore é rubro se e somente se o nó é rubro.
2. Um nó é rubro se e somente se os seus filhos são pretos.
3. Um nó folha (nó sem filhos) é sempre preta.
4. Se um nó é rubro, então os seus parentes são pretos.
5. A raiz é sempre preta.
6. Qualquer caminho do nó raiz até um nó folha na árvore rubro-negra tem o mesmo número de arestas pretas.

A diferença entre a árvore rubro-negra e a árvore AVL é que além de utilizar rotações ela também pode mudar a cor dos seus nós, reduzindo o próprio número de rotações, enquanto a árvore AVL permite o balanceamento apenas através das rotações dos nós.
As árvores rubro-negras são usadas para implementar as tabelas de dispersão em vários programas.

Códigos Exemplos

Árvore Binária de Busca - Insere (python)

Fonte:https://algoritmosempython.com.br/cursos/algoritmos-python/estruturas-dados/arvores/

```
class NodoArvore:
    def __init__(self, chave=None, esquerda=None, direita=None):
        self.chave = chave
        self.esquerda = esquerda
        self.direita = direita

def insere(raiz, nodo):
    """Insere um nodo em uma árvore binária de pesquisa."""
    # Nodo deve ser inserido na raiz.
    if raiz is None:
        raiz = nodo

    # Nodo deve ser inserido na subárvore direita.
    elif raiz.chave < nodo.chave:
        if raiz.direita is None:
            raiz.direita = nodo
        else:
            insere(raiz.direita, nodo)

    # Nodo deve ser inserido na subárvore esquerda.
    else:
        if raiz.esquerda is None:
            raiz.esquerda = nodo
        else:
            insere(raiz.esquerda, nodo)
```

Árvore AVL - Insere (python)

Fonte: https://www.geeksforgeeks.org/insertion-in-an-avl-tree/?ref=gcse

```
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

# AVL tree class which supports the
# Insert operation
class AVL_Tree(object):
```

```
# Recursive function to insert key in
# subtree rooted with node and returns
# new root of subtree.
def insert(self, root, key):

    # Step 1 - Perform normal BST
    if not root:
        return TreeNode(key)
    elif key < root.val:
        root.left = self.insert(root.left, key)
    else:
        root.right = self.insert(root.right, key)

    # Step 2 - Update the height of the
    # ancestor node
    root.height = 1 + max(self.getHeight(root.left),
                          self.getHeight(root.right))

    # Step 3 - Get the balance factor
    balance = self.getBalance(root)

    # Step 4 - If the node is unbalanced,
    # then try out the 4 cases
    # Case 1 - Left Left
    if balance > 1 and key < root.left.val:
        return self.rightRotate(root)

    # Case 2 - Right Right
    if balance < -1 and key > root.right.val:
        return self.leftRotate(root)

    # Case 3 - Left Right
    if balance > 1 and key > root.left.val:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)

    # Case 4 - Right Left
    if balance < -1 and key < root.right.val:
        root.right = self.rightRotate(root.right)
        return self.leftRotate(root)

    return root
```

```
def leftRotate(self, z):

    y = z.right
    T2 = y.left

    # Perform rotation
    y.left = z
    z.right = T2

    # Update heights
    z.height = 1 + max(self.getHeight(z.left),
                       self.getHeight(z.right))
    y.height = 1 + max(self.getHeight(y.left),
                       self.getHeight(y.right))

    # Return the new root
    return y

def rightRotate(self, z):

    y = z.left
    T3 = y.right

    # Perform rotation
    y.right = z
    z.left = T3

    # Update heights
    z.height = 1 + max(self.getHeight(z.left),
                      self.getHeight(z.right))
    y.height = 1 + max(self.getHeight(y.left),
                      self.getHeight(y.right))

    # Return the new root
    return y

def getHeight(self, root):
    if not root:
        return 0

    return root.height

def getBalance(self, root):
    if not root:
```

```
            return 0

        return self.getHeight(root.left) - self.getHeight(root.right)
```

Árvore Rubro-Negra - Insere (java)

Fonte: https://www.geeksforgeeks.org/red-black-tree-set-2-insert/

```
import java.io.*;
public class RedBlackTree
{
    public Node root;//root node
    public RedBlackTree()
    {
        super();
        root = null;
    }
    // node creating subclass
    class Node
    {
        int data;
        Node left;
        Node right;
        char colour;
        Node parent;

        Node(int data)
        {
            super();
            this.data = data;   // only including data. not key
            this.left = null; // left subtree
            this.right = null; // right subtree
            this.colour = 'R'; // colour . either 'R' or 'B'
            this.parent = null; // required at time of rechecking.
        }
    }
    // this function performs left rotation
    Node rotateLeft(Node node)
    {
        Node x = node.right;
        Node y = x.left;
        x.left = node;
        node.right = y;
```

```
        node.parent = x; // parent resetting is also important.
        if(y!=null)
            y.parent = node;
        return(x);
    }
    //this function performs right rotation
    Node rotateRight(Node node)
    {
        Node x = node.left;
        Node y = x.right;
        x.right = node;
        node.left = y;
        node.parent = x;
        if(y!=null)
            y.parent = node;
        return(x);
    }


    // these are some flags.
    // Respective rotations are performed during traceback.
    // rotations are done if flags are true.
    boolean ll = false;
    boolean rr = false;
    boolean lr = false;
    boolean rl = false;
    // helper function for insertion. Actually this function performs all tasks in single
pass only.
    Node insertHelp(Node root, int data)
    {
        // f is true when RED RED conflict is there.
        boolean f=false;

        //recursive calls to insert at proper position according to BST properties.
        if(root==null)
            return(new Node(data));
        else if(data<root.data)
        {
            root.left = insertHelp(root.left, data);
            root.left.parent = root;
            if(root!=this.root)
            {
                if(root.colour=='R' && root.left.colour=='R')
                    f = true;
```

```
    }
}
else
{
    root.right = insertHelp(root.right,data);
    root.right.parent = root;
    if(root!=this.root)
    {
        if(root.colour=='R' && root.right.colour=='R')
            f = true;
    }
// at the same time of insertion, we are also assigning parent nodes
// also we are checking for RED RED conflicts
}

// now lets rotate.
if(this.ll) // for left rotate.
{
    root = rotateLeft(root);
    root.colour = 'B';
    root.left.colour = 'R';
    this.ll = false;
}
else if(this.rr) // for right rotate
{
    root = rotateRight(root);
    root.colour = 'B';
    root.right.colour = 'R';
    this.rr  = false;
}
else if(this.rl)  // for right and then left
{
    root.right = rotateRight(root.right);
    root.right.parent = root;
    root = rotateLeft(root);
    root.colour = 'B';
    root.left.colour = 'R';

    this.rl = false;
}
else if(this.lr)  // for left and then right.
{
    root.left = rotateLeft(root.left);
    root.left.parent = root;
```

```
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.lr = false;
        }
        // when rotation and recolouring is done flags are reset.
        // Now lets take care of RED RED conflict
        if(f)
        {
            if(root.parent.right == root)  // to check which child is the current node of its
parent
            {
                if(root.parent.left==null || root.parent.left.colour=='B')  // case when parent's
sibling is black
                {// perform certain rotation and recolouring. This will be done while
backtracking. Hence setting up respective flags.
                    if(root.left!=null && root.left.colour=='R')
                        this.rl = true;
                    else if(root.right!=null && root.right.colour=='R')
                        this.ll = true;
                }
                else // case when parent's sibling is red
                {
                    root.parent.left.colour = 'B';
                    root.colour = 'B';
                    if(root.parent!=this.root)
                        root.parent.colour = 'R';
                }
            }
            else
            {
                if(root.parent.right==null || root.parent.right.colour=='B')
                {
                    if(root.left!=null && root.left.colour=='R')
                        this.rr = true;
                    else if(root.right!=null && root.right.colour=='R')
                        this.lr = true;
                }
                else
                {
                    root.parent.right.colour = 'B';
                    root.colour = 'B';
                    if(root.parent!=this.root)
                        root.parent.colour = 'R';
```

```
            }
        }
        f = false;
    }
    return(root);
}

// function to insert data into tree.
public void insert(int data)
{
    if(this.root==null)
    {
        this.root = new Node(data);
        this.root.colour = 'B';
    }
    else
        this.root = insertHelp(this.root,data);
}
}
```