## Árvore Binária de Busca:

Uma árvore binária é uma estrutura que pode não ter elemento algum (chamada de árvore vazia). Ou ela possui um elemento diferente, que será chamada de raiz. Essa raiz aponta para duas sub-árvores chamadas de sub-árvore esquerda e sub-árvore direita. As árvores possuem nós, e quando se trata de nós de uma árvore binária eles podem ter grau 0, 1 ou 2. Quando ele tem grau 0, ele é uma folha. Caso cada nó da árvore tenha grau 0 ou 2, ela será chamada de estritamente binária.

A distância de um nó até a raiz é chamada de profundidade de um nó. Quando temos um conjunto de nós que possuem profundidades iguais, temos um nível de uma árvore. A altura de uma árvore é a maior profundidade existente. Uma árvore será completa caso todas as folhas da árvore estejam no mesmo nível.

Em grafos, uma árvore binária é um grafo acíclico, conexo, dirigido e que cada nó não tem grau maior que 2. Assim sendo, só existe um caminho entre dois nós distintos. E cada ramo da árvore é um vértice dirigido, sem peso, que parte do pai e vai o filho.

## CÓDIGO EM JAVA:

```
public class ArvBinBusca<Chave extends Comparable<Chave>, Valor>
{
	private No raiz;
	private class No
	{
		private Chave chave;
		private Valor valor;
		private No esq, dir;

		public No(Chave chave, Valor valor)
		{
			this.chave = chave;
			this.valor = valor;
			this.esq = null;
			this.dir = null;
		}

		public No(Chave chave, Valor valor, No esq, No dir)
		{
			this.chave = chave;
			this.valor = valor;
			this.esq = esq;
			this.dir = dir;
		}
	}

	public ArvBinBusca()
	{
		raiz = null;
	}


	public int tamanho()
	{
		return tamanho(raiz);
	 }
```

```
 private int tamanho(No x)
{
	if(x == null)
		return 0;

	return 1 + tamanho(x.esq) + tamanho(x.dir);
}

public void insere(Chave chave, Valor valor)
{
	if(chave == null)
		throw new IllegalArgumentException("A chave fornecida é null!");

	if(valor == null) {
		delete(chave);
		return;
	}

	raiz = insere(raiz, chave, valor);
}

private No insere(No x, Chave chave, Valor valor)
{

	if (x == null)
		return new No(chave, valor);

	int cmp = chave.compareTo(x.chave);

	if(cmp < 0) /* Deve-se ir para a esquerda. */
		x.esq = insere(x.esq, chave, valor);
	else if(cmp > 0) /* Deve-se ir para a direita. */
		x.dir = insere(x.dir, chave, valor);
	else /* Caso tenha encontrado nó de mesma chave. */
		x.valor = valor;

	return x;
}
```

**Árvore AVL:**

É uma árvore binária de busca que se auto-balanceia. Na AVL, a altura de dois nós que são folha difere em no máximo 1. Algumas operações como busca, inserção e deleção possuem complexidade O(log n), no qual n é o número de elementos da árvore. Após uma inserção ou deleção pode ser necessário que a árvore seja rebalanceada, exigindo rotações.

Uma árvore AVL é balanceada quando a diferença entre as alturas das sub-árvores não é maior do que 1. Caso não esteja balanceada é necessário realizar a rotação simples ou rotação dupla para balanceá-la. Para definir o balanceamento é utilizado um fator. O fator de balanceamento de um nó é dado pelo seu peso em relação a sua sub-árvore. Um nó com fator balanceado pode conter 1, 0, ou -1 em seu fator. Quando um nó possui fator de balanceamento -2 ou 2 ele é considerado uma árvore não-AVL e com isso, é necessário um balanceamento por rotação ou dupla-rotação.

As operações básicas em uma árvore AVL normalmente possuem os mesmos algoritmos de uma BST desbalanceada. A rotação ocorre devido ao desbalanceamento, uma rotação simples ocorre

quando um nó está desbalanceado e seu filho estiver no mesmo sentido da inclinação. Uma rotação-dupla ocorre quando um nó estiver desbalanceado e seu filho estiver inclinado no sentido inverso ao do pai.

**Rotação à esquerda:**
Basta empurrar o nodo N para baixo e para a esquerda. O filho à direita de N o substitui, e o filho à esquerda do filho à direita vem a ser o novo filho à direita de N.

**Rotação à direita:**
Basta empurrar o nodo N para baixo e para a direita. O filho à esquerda de N o substitui, e o filho à direita do filho à esquerda vem a ser o novo filho à esquerda de N.

As rotações são duas rotações simples seguidas, independentes se *à* direita ou *à* esquerda.

**CÓDIGO EM PYTHON:**

```
class treeNode(object):
    def __init__(self, value):
        self.value = value
        self.l = None
        self.r = None
        self.h = 1

class AVLTree(object):

    def insert(self, root, key):

        if not root:
            return treeNode(key)
        elif key < root.value:
            root.l = self.insert(root.l, key)
        else:
            root.r = self.insert(root.r, key)

        root.h = 1 + max(self.getHeight(root.l),
                                        self.getHeight(root.r))

        b = self.getBal(root)

        if b > 1 and key < root.l.value:
            return self.rRotate(root)

        if b < -1 and key > root.r.value:
            return self.lRotate(root)

        if b > 1 and key > root.l.value:
            root.l = self.lRotate(root.l)
            return self.rRotate(root)

        if b < -1 and key < root.r.value:
            root.r = self.rRotate(root.r)
            return self.lRotate(root)

        return root
```

```
    def lRotate(self, z):

        y = z.r
        T2 = y.l

        y.l = z
        z.r = T2

        z.h = 1 + max(self.getHeight(z.l),
                      self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),
                      self.getHeight(y.r))

        return y

    def rRotate(self, z):

        y = z.l
        T3 = y.r

        y.r = z
        z.l = T3

        z.h = 1 + max(self.getHeight(z.l),
                      self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),
                      self.getHeight(y.r))

        return y

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

        return root.h

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

        return self.getHeight(root.l) - self.getHeight(root.r)

    def preOrder(self, root):

        if not root:
            return

        print("{0} ".format(root.value), end="")
        self.preOrder(root.l)
        self.preOrder(root.r)
```

**Árvore Rubro-Negra:**

São árvores binárias simétricas, elas também possuem as operações de inserção, remoção, busca, porém são mais eficientes devido ao fato de sempre estarem balanceadas. Isso acontece devido a característica que essa árvore possui, essa peculiaridade vem de um bit extra em cada nó que determina se esta é "vermelha" ou "preta" dentro do conjunto de regras que rege a árvore. Além desse bit, cada nó também conta com os dados do nó, filho esquerdo do nó, filho direito do nó e pai do nó.

Esta árvore está sempre balanceada pois ela segue essas regras:

- Cada nó da árvore possui um valor (regra 1)
- A cada novo nó inserido na árvore obedecerá o esquema de menor para o lado esquerdo e maior para o lado direito. (regra 2)
- A cada nó é associada uma cor: vermelha ou preta. (regra 3)
- A raiz é sempre preto. (regra 4)
- Nós vermelhos que não são folhas possuem apenas filhos pretos. (regra 5)
- Todos os caminhos a partir da raiz até qualquer folha passa pelo mesmo número de nós pretos. (regra 6)

A cada vez que uma operação é realizada na árvore, testa-se este conjunto de propriedades e são efetuadas rotações e ajuste de cores até que a árvore satisfaça todas estas regras.

**Rotação:**

É uma operação realizada na árvore a fim de garantir seu balanceamento. Pode ser feita a direita e a esquerda, onde são alterados os nós rotacionados.

**Inserção:**

Ao inserir um elemento na árvore rubro-negra, este é comparado com os outros elementos e é alocado em sua posição conforme a regra 2. Ao inserir um elemento ele é sempre da cor vermelha (exceto se for o nó raiz). A seguir a árvore analisa o antecessor da folha. Se este for vermelho será necessário alterar as cores para garantir a regra 6.

**Remoção:**

**Remoção efetiva:**
Com as operações de rotação e alteração de cor, remove-se o nó e estabelece-se as propriedades da árvore.

**Remoção preguiçosa:**
Esta remoção marca um nó como removido, mas efetivamente não o remove. Sendo desta maneira nenhuma alteração é efetuada na árvore, porém são necessários novos mecanismos de busca e inserção para que reconheçam o nó como "ausente".

**CÓDIGO EM PYTHON:**

```
# Define Node
class Node():
    def __init__(self,val):
        self.val = val                  # Value of Node
        self.parent = None                  # Parent of Node
        self.left = None                # Left Child of Node
        self.right = None                # Right Child of Node
        self.color = 1

# Define R-B Tree
class RBTree():
    def __init__(self):
        self.NULL = Node ( 0 )
        self.NULL.color = 0
```

```
    self.NULL.left = None
    self.NULL.right = None
    self.root = self.NULL


# Insert New Node
def insertNode(self, key):
    node = Node(key)
    node.parent = None
    node.val = key
    node.left = self.NULL
    node.right = self.NULL
    node.color = 1                          # Set root colour as Red

    y = None
    x = self.root

    while x != self.NULL :                  # Find position for new node
        y = x
        if node.val < x.val :
            x = x.left
        else :
            x = x.right

    node.parent = y                         # Set parent of Node as y
    if y == None :                          # If parent i.e, is none then it is root node
        self.root = node
    elif node.val < y.val :
        y.left = node
    else :
        y.right = node

    if node.parent == None :                # Root node is always Black
        node.color = 0
        return

    if node.parent.parent == None :         # If parent of node is Root Node
        return

    self.fixInsert ( node )
```