

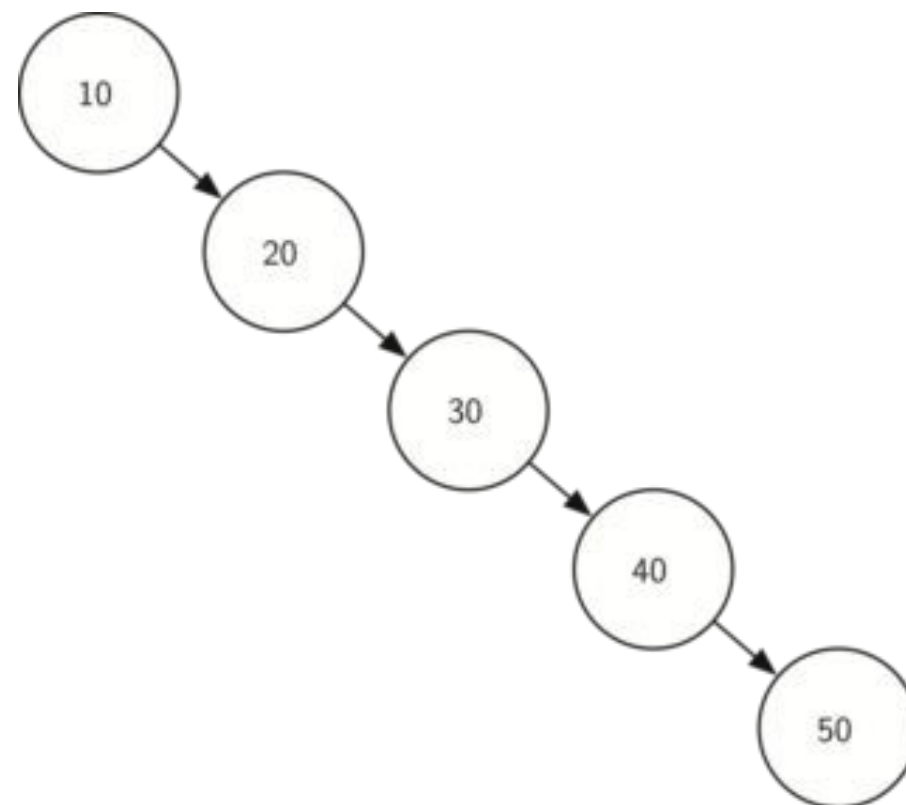
Árvore Rubro-Negra

**Vinicius Monteiro, Luis Antônio, Lucas Montenegro, João
Messias**

https://github.com/lucas-montenegro/projeto_p2

Motivação

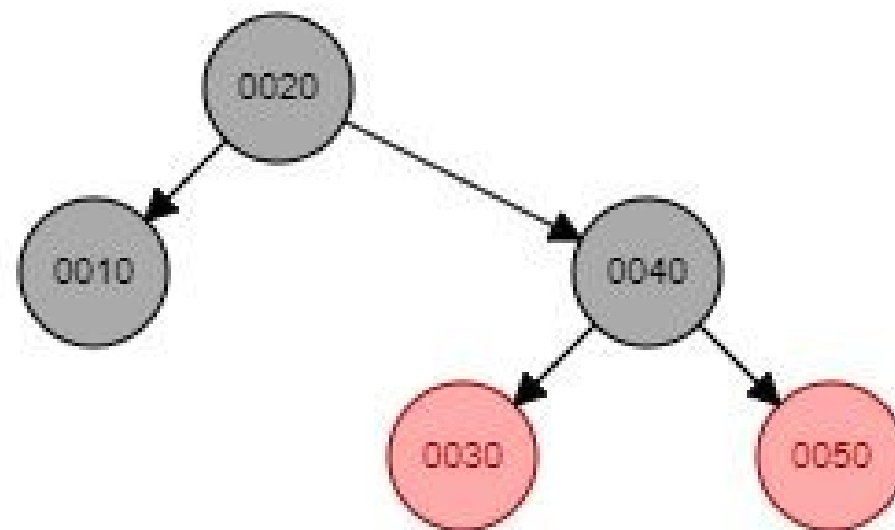
Queremos buscas com garantia de $O(\log n)$



É possível fazer busca em $O(\log n)$ nessa

Motivação

**É possível resolver esse problema com
Árvore Rubro-Negra !**



Árvore Rubro-Negra

- É uma Árvore de Busca Binária balanceada.
- É muito utilizada para fazer buscas, inserções e remoções em $O(\log n)$.

Definições

- Assim como uma Árvore de Busca Binária, uma Árvore Rubro-Negra possui raíz, nós internos e folhas.
- Propriedades da Árvore Rubro-Negra:
 - . Cada nó possui uma cor: vermelho ou preto.
 - . Sua raíz é preta.
 - . Todos os nós nulos são pretos.
 - . Todos os filhos de um nó vermelho são pretos.
 - . Todos os caminhos de um dado nó até o nó nulo têm mesma quantidade de nós pretos.

Árvore Rubro-Negra ADT

```
rbt *create_empty_red_black_tree();
```

```
rbt *create_red_black_tree(  
int item, rbt *left_child, rbt *right_child, rbt *parent);
```

```
rbt *left_rotation(rbt **root, rbt *r_b_t);
```

```
rbt *right_rotation(rbt **root, rbt *r_b_t);
```

```
rbt* add(rbt **r_b_t, rbt *parent, int item);
```

Árvore Rubro-Negra ADT

rbt* search(rbt *r_b_t, **int** item);

void red_uncle(rbt *parent_node, rbt *uncle);

void black_uncle(
rbt **root, rbt *r_b_t, rbt *parent_node, rbt *uncle);

void fix(rbt **root, rbt *r_b_t);

rbt *add_and_fix(rbt *root, **int** item);

void print_tree_pre_order(rbt *r_b_t);

Struct

```
struct red_black_tree {  
    bool color;  
    int item;  
    rbt *parent;  
    rbt *right_child;  
    rbt *left_child;  
};
```


Inserção

```
rbt *add_and_fix(rbt *root, int item) {  
    rbt *added = NULL;  
    added = add(&root, root, item);  
  
    fix(&root, added);  
  
    return root;  
}
```

```

rbt* add(rbt **r_b_t, rbt *parent, int item) {
    if(*r_b_t == NULL){
        *r_b_t = create_red_black_tree(item, NULL,
parent);
        return *r_b_t;
    }
    else if((*r_b_t) -> item > item) {
        add(&(*r_b_t) -> left_child, *r_b_t, item);
    }
    else {
        add(&(*r_b_t) -> right_child, *r_b_t, item);
    }
}

```

```

void fix(rbt **root, rbt *r_b_t)
{
    rbt *parent_node = r_b_t -> parent;
    if(parent_node != NULL && parent_node -> color == false && parent_node -> parent != NULL) {
        rbt *uncle;

        if(parent_node == parent_node -> parent -> left_child) {
            uncle = parent_node -> parent -> right_child;
        }
        else if (parent_node == parent_node -> parent -> right_child) {
            uncle = parent_node -> parent -> left_child;
        }

        if(uncle != NULL && uncle -> color == false) {
            red_uncle(parent_node, uncle);
            fix(root, parent_node -> parent);
        }
        else if(uncle == NULL || uncle -> color == true) {
            black_uncle(root, r_b_t, parent_node, uncle);
        }
    }
    (*root) -> color = true;
}

```

```
void red_uncle(rbt *parent_node, rbt *uncle) {  
    parent_node -> color = true;  
    uncle -> color = true;  
    parent_node -> parent -> color = false;  
}
```

```

void black_uncle(rbt **root, rbt *r_b_t, rbt *parent_node, rbt *uncle) {
    if(parent_node == parent_node -> parent -> left_child && r_b_t == parent_node -> left_child) {
        parent_node = right_rotation(root, parent_node -> parent);
        parent_node -> color = true;
        parent_node -> right_child -> color = false;
    }
    else if(parent_node == parent_node -> parent -> left_child && r_b_t == parent_node -> right_child) {
        parent_node = left_rotation(root, parent_node);
        parent_node = right_rotation(root, parent_node -> parent);
        parent_node -> color = true;
        parent_node -> right_child -> color = false;
    }
    else if(parent_node == parent_node -> parent -> right_child && r_b_t == parent_node -> right_child) {
        parent_node = left_rotation(root, parent_node -> parent);
        parent_node -> color = true;
        parent_node -> left_child -> color = false;
    }
    else if(parent_node == parent_node -> parent -> right_child && r_b_t == parent_node -> left_child) {
        parent_node = right_rotation(root, parent_node);
        parent_node = left_rotation(root, parent_node -> parent);
        parent_node -> color = true;
        parent_node -> left_child -> color = false;
    }
}

```

```

rbt *left_rotation(rbt **root, rbt *r_b_t) {
    rbt *rbt_right_child = NULL;

    if(r_b_t != NULL && r_b_t -> right_child != NULL) {
        rbt_right_child = r_b_t -> right_child;
        r_b_t -> right_child = rbt_right_child -> left_child;
        rbt_right_child -> left_child = r_b_t;
    }

    if(r_b_t -> parent != NULL){
        if(r_b_t == r_b_t -> parent -> left_child)
            r_b_t -> parent -> left_child = rbt_right_child;
        else
            r_b_t -> parent -> right_child = rbt_right_child;
    }

    else
        *root = rbt_right_child;

    rbt_right_child -> parent = r_b_t -> parent;
    r_b_t -> parent = rbt_right_child;
    if(r_b_t -> right_child != NULL)
        r_b_t -> right_child -> parent = r_b_t;

    return rbt_right_child;
}

```

```

rbt *right_rotation(rbt **root, rbt *r_b_t) {
    rbt *rbt_left_child = NULL;

    if(r_b_t != NULL && r_b_t -> left_child != NULL) {
        rbt_left_child = r_b_t -> left_child;
        r_b_t -> left_child = rbt_left_child -> right_child;
        rbt_left_child -> right_child = r_b_t;
    }

    if(r_b_t -> parent != NULL){
        if(r_b_t == r_b_t -> parent -> left_child)
            r_b_t -> parent -> left_child = rbt_left_child;
        else
            r_b_t -> parent -> right_child = rbt_left_child;
    }

    else
        *root = rbt_left_child;

    rbt_left_child -> parent = r_b_t -> parent;
    r_b_t -> parent = rbt_left_child;
    if(r_b_t -> left_child != NULL)
        r_b_t -> left_child -> parent = r_b_t;

    return rbt_left_child;
}

```

Busca

```
rbt* search(rbt *r_b_t, int item) {  
    if(r_b_t == NULL)  
        return NULL;  
  
    else if(r_b_t -> item == item)  
        return r_b_t;  
  
    else if(r_b_t -> item > item)  
        search(r_b_t -> left_child, item);  
  
    else  
        search(r_b_t -> right_child, item);  
}
```


Animação

De volta à Motivação

- Conseguimos realizar operações com garantia de $O(\log n)$ de acordo com as rotações e mudanças de cores que realizávamos ao decorrer da montagem da árvore.
- Por que não usar AVL ?
- Em uma aplicação que envolve muitas inserções e remoções na árvore, a AVL tem uma performance pior que a Árvore Rubro-Negra devido à maior quantidade de rotações necessárias para balanceá-la.

Referências

<https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>