Laboratório 2: Árvores Balanceadas

Caio Graça Gomes

13 de Maio de 2020

1 Descrição da estrutura utilizada

Considerando as propriedades requeridas para atingir complexidade de tempo satisfatória, decidiu-se implementar uma árvore rubro-negra, pois ela permite realizar operações de inserção e busca em $O(log(n))$ tempo. Utilizou-se o livro Cormen 3ª edição [1] como base para implementação da árvore.

1.1 Propriedades das árvores rubro-negras

A ideia principal da árvore rubro-negra é se comportar como uma árvore de busca binária, com armazenamento de mais uma propriedade: a cor do nó, que pode ser vermelho ou preto, o que dá a ela o seu nome. Ao restringir as cores dos nós qualquer caminho descente da raíz até uma folha não será maior que duas vezes qualquer outro, o que faz da árvore aproximadamente balanceada.

Dessa forma, cada nó da árvore possui os atributos *cor*, *chave*, *filho esquerdo*, *filho direito* e *pai*. Em caso de inexistência de filho ou pai de um nó, o respectivo ponteiro aponta para um nó especial que contém o valor *NIL*. Esses valores devem ser tratados como ponteiros para as folhas da árvore.

A árvore rubro-negra satisfaz as seguintes propriedades:

1. Nós são vermelhos ou pretos;

2. A raíz é preta;

3. Toda folha (*NIL*) é preta;

4. Nós vermelhos tem filhos pretos;

5. Para todo nó, todos os caminhos descentes partindo deste até uma folha qualquer contém o mesmo número de nós pretos.

A Figura 1 apresenta um exemplo de uma árvore rubro-negra. Porém, por uma questão de mais simplicidade nos códigos, todos os ponteiros para *NIL* são substituídos para um único *T.nil* (Figura 2). Contudo, nas próximas representações será omitido o ponteiro *T.nil*, conforme a Figura 3.

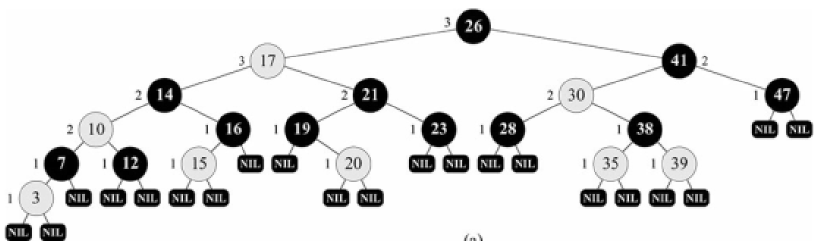

Figure 1: Representação da árvore rubro-negra com folhas *NIL*.

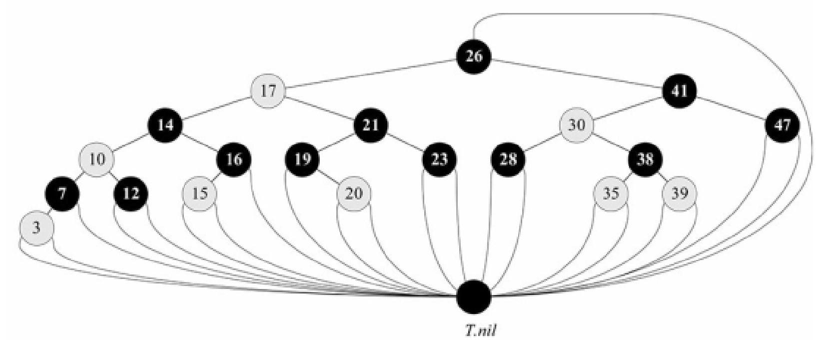

Figure 2: Representação da árvore rubro-negra com único ponteiro *T.nil*.

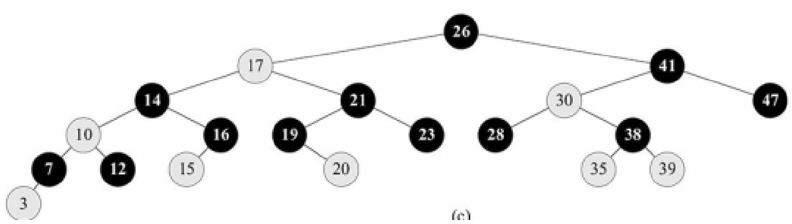

Figure 3: Representação da árvore rubro-negra omitindo o ponteiro *T.nil*.

Denomina-se o número de nós pretos em qualquer caminho descente partindo de um nó x por $bh(x)$. A partir dessa definição e das propriedades cumpridas pela árvore rubro-negra, pode-se demonstrar que sua altura, supondo que possui n nós, é no máximo $2 * log(n + 1)$ [1]. Isso garante que a operação de busca na árvore é $O(log(n))$.

Assim, representa-se o construtor da classe árvore rubro-negra em $C++$ da seguinte maneira:

```cpp
#include<bits/stdc++.h>

struct node {
    float key;
    long idx;
    struct node* left = nullptr;
    struct node* right = nullptr;
    struct node* parent = nullptr;
    std::string color;
};

class RBtree{

    struct node* nil;
    struct node* root;
    long counter;

    public:

        RBtree(){
            nil = new struct node;
            nil->color = "BLACK";
            nil->left = nullptr;
            nil->right = nullptr;
            root = nil;
            counter = 0;
        };

        long get_counter();

        struct node* get_root();

        void left_rotate(struct node* oldroot);

        void right_rotate(struct node* oldroot);

        void insert_node(float key, long idx);

        void insert_fixup(struct node* newnode);

        void find(struct node* startnode,
        std::vector<long> &res, float first, float last);

};
```

Figure 4: Construtor da classe árvore rubro-negra

1.2 Rotações

Quando executadas, as operações de inserção e deleção em árvores rubro-negras modificam a estrutura da árvore, de modo que ela pode vir a violar as propriedades enumeradas. Para restaurar essas propriedades, é necessário alterar as cores de alguns nós e mudar os apontamentos de alguns ponteiros. Mudar a estrutura dos ponteiros por meio de uma rotação é uma operação que preserva a propriedade de árvore de busca binária.

Há dois tipos de rotações, para a esquerda e para a direita, que são ilustradas na Figura 5:

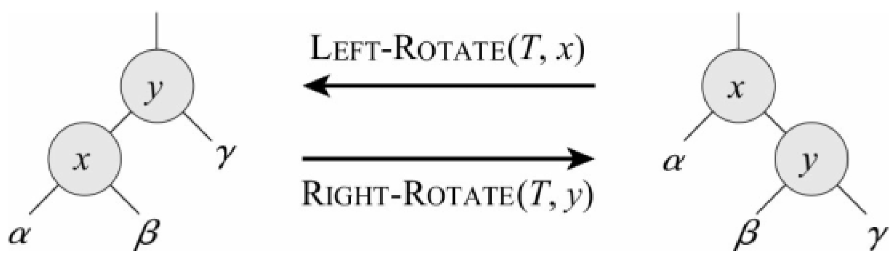

Figure 5: Operação de rotação em árvore rubro-negra. Cada qual é realizada em $O(1)$ tempo.

Assim sendo, o código em $C++$ para a operação *left_rotate()* é:

```
void RBtree::left_rotate(struct node* oldroot){
    struct node* newroot = oldroot->right;
    oldroot->right = newroot->left;
    if(newroot->left != nil)
        newroot->left->parent = oldroot;
    newroot->parent = oldroot->parent;
    if(oldroot->parent == nullptr) this->root = newroot;
    else if(oldroot == oldroot->parent->left)
        oldroot->parent->left = newroot;
    else oldroot->parent->right = newroot;
    newroot->left = oldroot;
    oldroot->parent = newroot;
}
```

Figure 6: Implementação em $C++$ da função *left_rotate()*

A implementação de *right_rotate()* é simétrica, trocando *left* por *right* e vice-versa.

1.3 Inserções

Para a operação de inserção em uma árvore rubro-negra, é utilizado um algoritmo semelhante à inserção em árvore de busca binária, com a diferença de ter que, após adicionar um nó a árvore, fazer operações de modo a restaurar as propriedades das árvores rubro-negras.

```cpp
void RBtree::insert_node(float key, long idx){
    this->counter++;

    struct node* aux = this->root;
    struct node* aux2 = nullptr;

    struct node* newnode = new struct node;
    newnode->key = key;
    newnode->idx = idx;
    newnode->parent = nullptr;
    newnode->left = nil;
    newnode->right = nil;
    newnode->color = "RED";

    while(aux != nil){
        aux2 = aux;
        if(newnode->key < aux->key)
            aux = aux->left;
        else aux = aux->right;
    }
    newnode->parent = aux2;
    if(aux2 == nullptr) this->root = newnode;
    else if (newnode->key < aux2->key)
        aux2->left = newnode;
    else aux2->right = newnode;

    if(newnode->parent == nullptr){
        newnode->color = "BLACK";
        return;
    }

    if(newnode->parent->parent == nullptr){
        return;
    }

    insert_fixup(newnode);
}
```

Figure 7: Função para inserção de nó na árvore rubro-negra

Como visto na linha 75 da Figura 7, é necessária a implementação de uma função *insert_fixup()* que visa a restaurar as propiedades da árvore rubro-negra, cuja lógica está explicitada em [1]. Assim, seu código em $C++$ está implementado na Figura 8:

```cpp
void RBtree::insert_fixup(struct node* newnode){
    while(newnode->parent->color == "RED"){
        if(newnode->parent == newnode->parent->parent->right){
            struct node* aux = newnode->parent->parent->left;
            if(aux->color == "RED"){
                newnode->parent->color = "BLACK";
                aux->color = "BLACK";
                newnode->parent->parent->color = "RED";
                newnode = newnode->parent->parent;
            }
            else{
                if(newnode == newnode->parent->left){
                    newnode = newnode->parent;
                    right_rotate(newnode);
                }
                newnode->parent->color = "BLACK";
                newnode->parent->parent->color = "RED";
                left_rotate(newnode->parent->parent);
            }
        }
        else{
            struct node* aux = newnode->parent->parent->right;
            if(aux->color == "RED"){
                newnode->parent->color = "BLACK";
                aux->color = "BLACK";
                newnode->parent->parent->color = "RED";
                newnode = newnode->parent->parent;
            }
            else{
                if(newnode == newnode->parent->right){
                    newnode = newnode->parent;
                    left_rotate(newnode);
                }
                newnode->parent->color = "BLACK";
                newnode->parent->parent->color = "RED";
                right_rotate(newnode->parent->parent);
            }
        }
        if(newnode == root) break;
    }
    root->color = "BLACK";
}
```

Figure 8: Função para restaurar propriesdades da árvore rubro-negra após uma inserção

2 Análise da complexidade do tempo de execução das operações

Analisando os resultados obtidos a partir dos casos testes propostos, pode-se computar gráficos a respeito do tempo de execução das operações em função do tamanho da árvore rubro-negra.

Com o uso do $MATLAB$, gerou-se gráficos a partir dos dados obtidos e os sobrepuseram com curvas da forma $y = a * log(x) + b$ para verificação de que as operações são realmente $O(log(n))$.

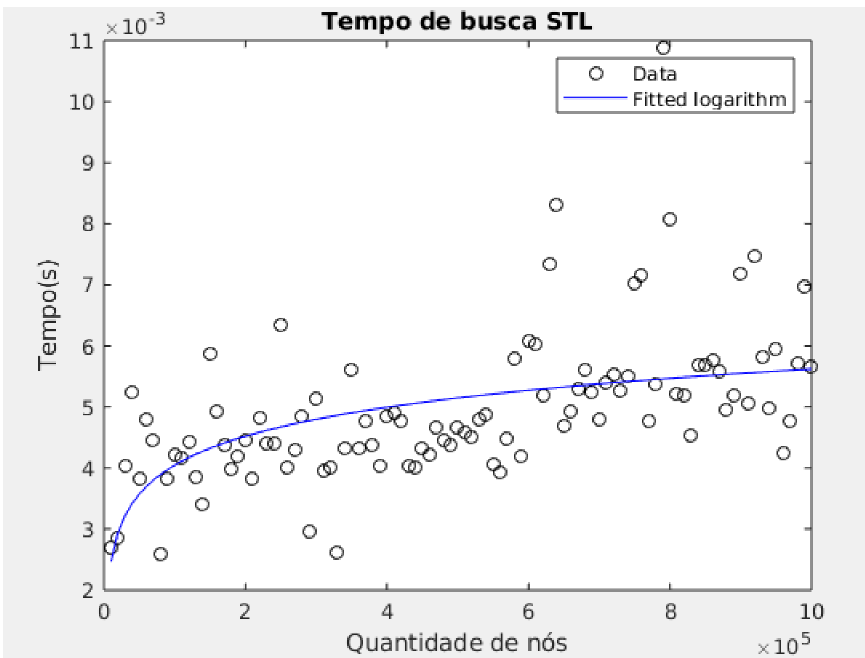

Figure 9: Tempo de execução da busca em função da quantidade de entradas usando a implementação STL

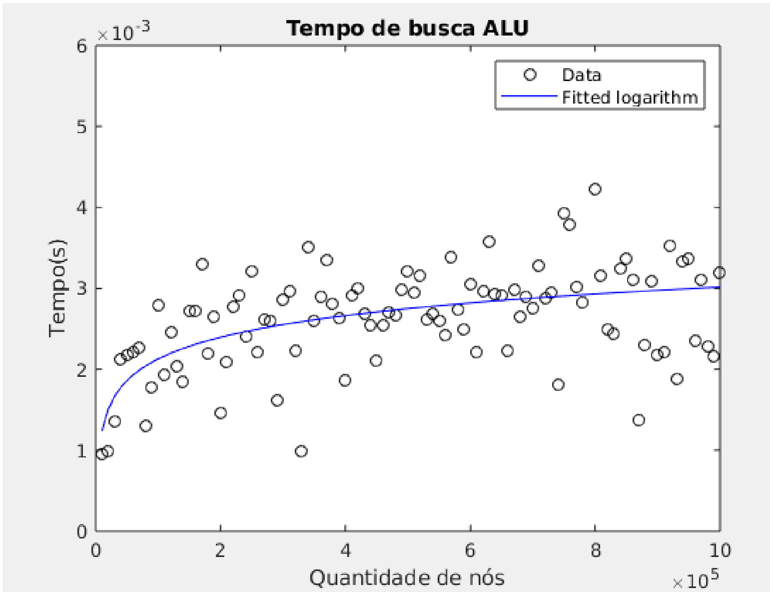

Figure 10: Tempo de execução da busca em função da quantidade de entradas usando a implementação de árvore rubrp-negra

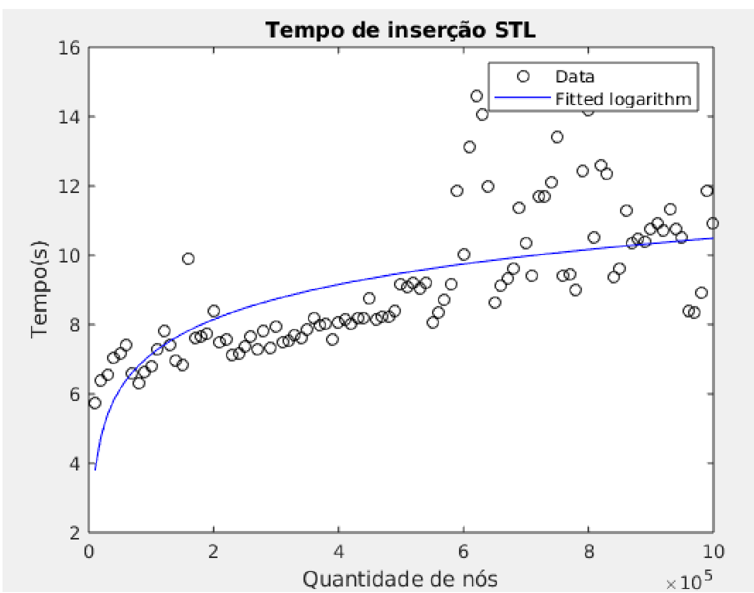

Figure 11: Tempo de execução de inserção em função da quantidade de entradas usando a implementação STL

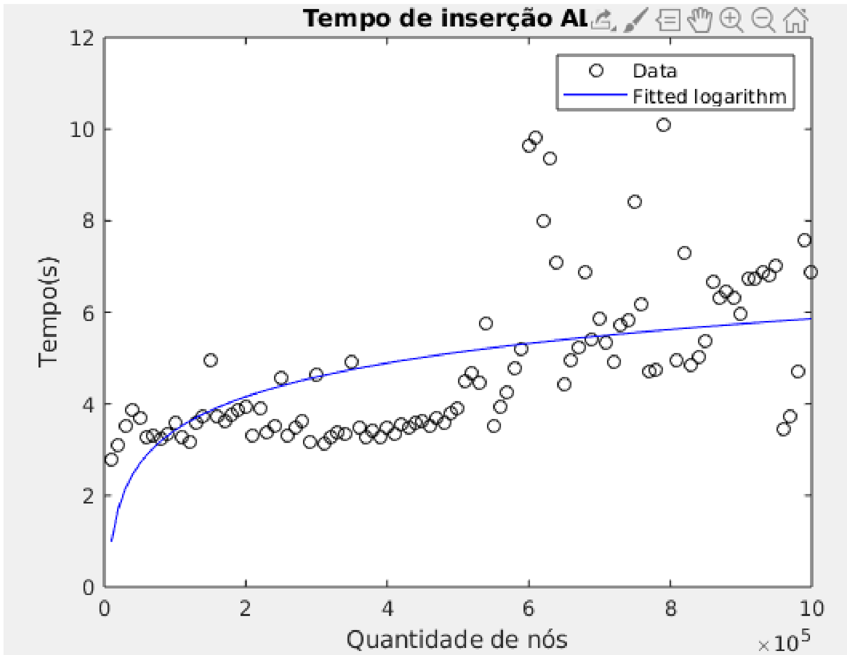

Figure 12: Tempo de execução de inserção em função da quantidade de entradas usando a implementação de árvore rubro-negra

Primeiramente, é evidente que a implementação de árvore rubro-negra foi mais eficiente que a implementação com a biblioteca padrão do $C++$ STL. Isso ocorre pois a STL é mais genérica, há uma maior hierarquia de classes.

Nota-se ainda que as operações em ambas implementações apresentam uma tendência a ser $O(log(n))$ mas ainda variam muito dependendo da variável de entrada. Uma quantidade relevante de pontos fogem consideravelmente à tendência da curva que melhor ajusta. Isso é esperado na implementação de árvore rubro-negra pois, dado um nó, um caminho descente deste nó até uma folha pode ser até duas vezes o tamanho do caminho deste nó para outra folha, então é esperado que existam tempos de execução até duas vezes maior que o esperado pela curva de melhor ajuste quadrático.

3 Referências

1 T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.