Árvores

Árvores Binárias de Busca: Balanceamento

Prof. Edson Alves - UnB/FGA

2018

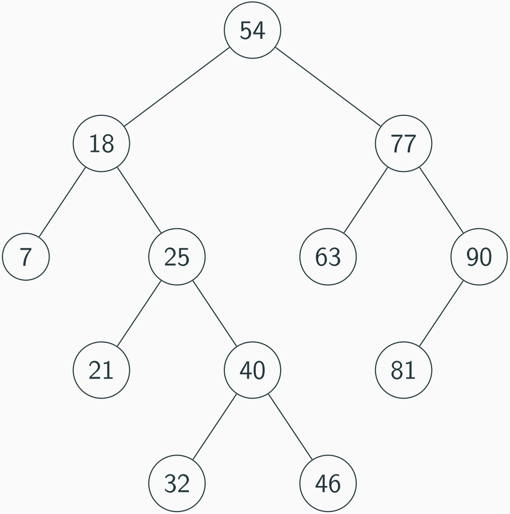
Sumário

Balanceamento de árvores binárias

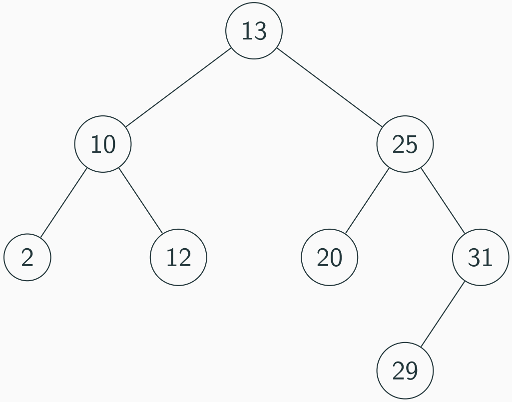
Balanceamento

- Uma árvore binária está balanceada se a diferença de altura das duas subárvores de qualquer nó da árvore é menor ou igual a 1
- Uma árvore binária está perfeitamente balanceada se ela está balanceada e todas as suas folhas se encontram em, no máximo, dois níveis distintos
- Uma árvore binária é dita completa se está perfeitamente balanceada e as folhas do seu último nível estão mais à esquerda possível
- Uma árvore binária é dita cheia se todos os seus nós tem ou zero ou dois filhos
- A altura de uma árvore é igual ao nível máximo dentre todos os nós da árvore
- Uma árvore pode estar balanceada sem estar perfeitamente balanceada

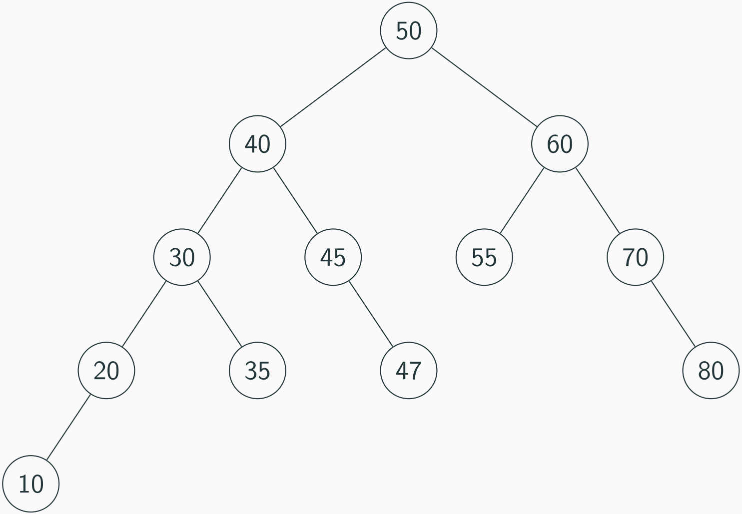
Exemplo de árvore desbalanceada



Exemplo de árvore perfeitamente balanceada



Exemplo de árvore balanceada, mas não perfeitamente balanceada



Algoritmo para verificação de balanceamento

- É possível utilizar a recursão para checar o balanceamento a cada nó, usando também a rotina recursiva de cálculo de tamanho
- O algoritmo é de fácil entendimento, mas possui maior complexidade assintótica $O(N^2)$
- É possível reduzir a complexidade para $O(N)$ através do uso de programação dinâmica
- Para verificar se uma árvore está perfeitamente balanceada, primeiro é necessário verificar se a mesma está balanceada
- Em seguida, determina-se a altura da árvore: se ela não extrapolar o menor inteiro maior do que $\log(N+1)$, a árvore está perfeitamente balanceada

Implementação do algoritmo de verificação de balanceamento

```cpp
template<typename T>
class BinaryTree {
private:
    struct Node {
        T info;
        Node *left, *right;
    };

    Node *root;

    int heigth(const Node *node) const
    {
        if (node == nullptr)
            return 0;

        return std::max(heigth(node->left), heigth(node->right)) + 1;
    }
```

Implementação do algoritmo de verificação de balanceamento

```cpp

    bool is_balanced(const Node *node) const
    {
        if (node == nullptr)
            return true;

        int L = heigth(node->left);
        int R = heigth(node->right);

        if (abs(L - R) > 1)
            return false;

        return is_balanced(node->left) and is_balanced(node->right);
    }

public:
    BinaryTree() : root(nullptr) {}

    bool is_balanced() const { return is_balanced(root); }
};
```

Algoritmos de balanceamento

Balanceamento por inserção

- Uma maneira de garantir o balanceamento de uma árvore binária de busca é utilizar um processo de inserção controlada, de modo que ao final das inserções a árvore resultante esteja balanceada
- Futuras inserções ou remoções podem resultar no desbalanceamento da árvore
- Esta estratégia é útil quando os elementos a serem inseridos são conhecidos de antemão e quando não haverão novas inserções ou remoções
- O algoritmo $O(N \log N)$ que resulta em uma árvore balanceada é
    1. armazene os N elementos a serem inseridos em um vetor v
    2. ordene v em ordem crescente
    3. insira o elemento central x do vetor na árvore
    4. continue o processo no subvetor à esquerda de x
    5. finalize o processo no subvetor à direita de x

Algoritmo de inserção balanceada

```
template<typename T>
class BST {
private:
    struct Node {
        T info;
        Node *left, *right;
    };

    Node *root;

    void BST(BinaryTree *tree, const vector<T>& vs, int a, int b)
    {
        if (a <= b)
        {
            int m = a + (b - a)/2;

            tree->insert(vs[m]);
            balanced_insertion(tree, vs, a, m - 1);
            balanced_insertion(tree, vs, m + 1, b);
        }
    }
```

Algoritmo de inserção balanceada

```cpp

public:
    BST() : root(nullptr) {}

    void insert(const T& info);

    static BinaryTree * balanced(const std::vector<T>& xs)
    {
        std::vector<T> vs(xs);
        std::sort(vs.begin(), vs.end());

        BST *tree = new BST();
        balanced_insertion(tree, vs, 0, vs.size() - 1);

        return tree;
    }
};
```

Notas sobre o algoritmo de balanceamento por inserção

- O algoritmo proposto tem duas grandes desvantagens: é necessário armazenar os elementos antes da inserção na árvore e ordenar dos elementos armazenados
- Se a árvore já tiver sido construída previamente, uma solução é preencher um vetor atráves de uma travessia em-order, destruir a árvore e reconstruí-la
- A solução apresentada remove a necessidade de ordenamento, uma vez que a travessia em-order garante que o vetor estará ordenado
- Outro ponto importante é que futuras inserções podem desfazer o balanceamento da árvore

Algoritmo DSW

- O algoritmo DSW foi proposto por Colin Day e melhorado por Quentin F. Stout e Bete L. Warren
- Ele consiste no reposicionamento dos nós através de operações de rotação do filho `C` em torno do seu pai `P`, o que pode pode envolver seu avô `G`
- A rotação pode ser feita tanto para a esquerda quanto para a direita
- Na rotação à esquerda, o filho (inicialmente à direita do pai) passa a ocupar a posição do pai, que se torna seu filho à esquerda
- Caso o filho tenha uma subárvore à esquerda antes da rotação, esta subárvore se torna o filho à direita do pai, após a rotação
- A rotação à direita faz processo simétrico, no sentido horário

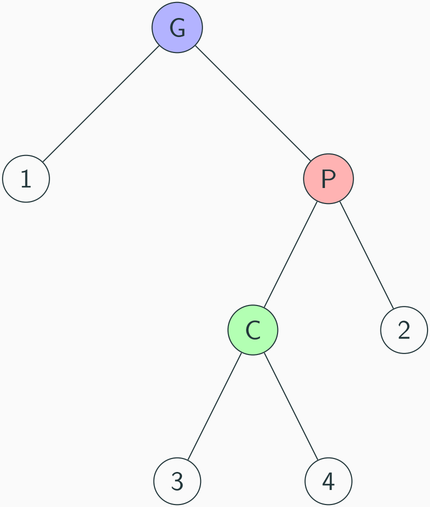
Implementação da rotação à esquerda

```
// C deve ser o filho à direita de P
void rotate_left(Node *G, Node *P, Node *C)
{
    // Se P não é a raiz da árvore
    if (G != nullptr)
    {
        // Então o avô se torna pai do neto
        if (G->left == P)
            G->left = C;
        else
            G->right = C;
    }

    // A subárvore à esquerda do filho se
    // torna a subárvore à direita do pai
    P->right = C->left;

    // O pai se torna o filho esquerdo do neto
    C->left = P;
}
```

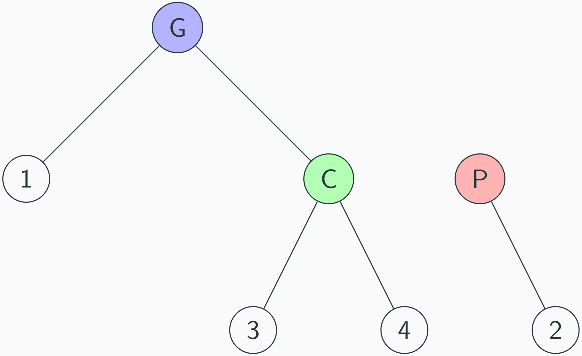
Implementação da rotação à direita

```
// C deve ser o filho à esquerda de P
void rotate_right(Node *G, Node *P, Node *C)
{
    // Se P não é a raiz da árvore
    if (G != nullptr)
    {
        // Então o avô se torna pai do neto
        if (G->left == P)
            G->left = C;
        else
            G->right = C;
    }

    // A subárvore à direita do filho
    // se torna a subárvore à esquerda do pai
    P->left = C->right;

    // O pai se torna o filho à direita do neto
    C->right = P;
}
```

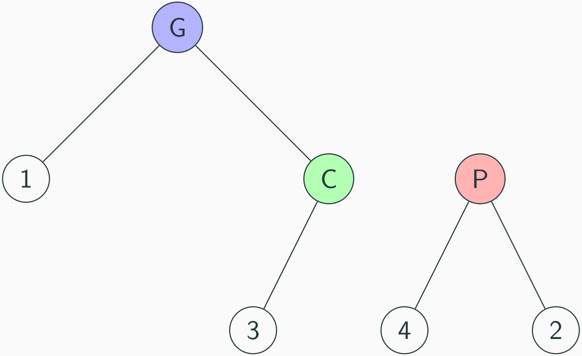
Exemplo de rotação à direita



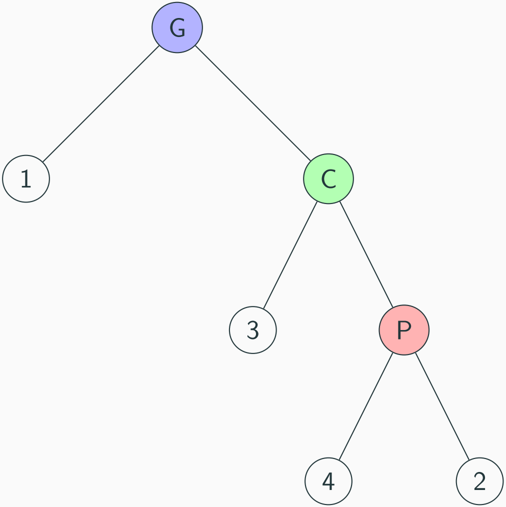
Exemplo de rotação à direita



Exemplo de rotação à direita



Exemplo de rotação à direita



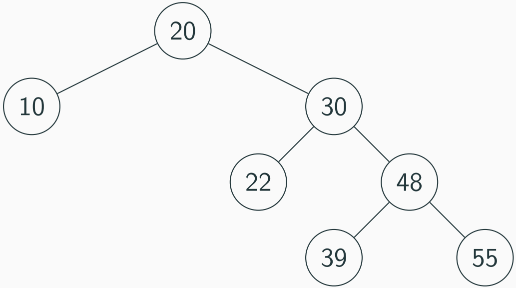
Algoritmo DSW

- O algoritmo DSW é composto por duas etapas
- A primeira delas consiste em transformar a árvore em uma lista encadeada denominada espinha dorsal (*backbone*)
- Em seguida o algoritmo transforma esta espinha dorsal numa árvore completa
- Estas duas etapas são realizada através do uso das rotações descritas anteriormente
- Cada rotação preserva a estrutura da árvore binária de busca, mudando contudo sua forma

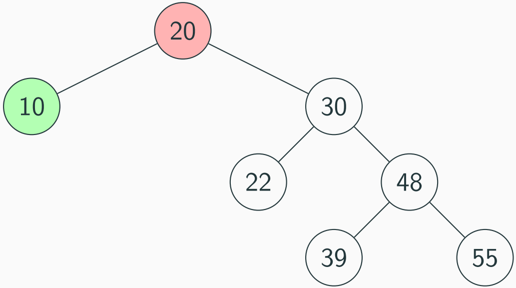
Geração da espinha dorsal

- A espinha dorsal é gerada através de sucessivas aplicações de rotações à direita
- Inicialmente se inicializa o filho `C` na raiz da árvore (naturalmente, o pai `P` e o avô `G` são inicialmente nulos)
- Se `C` tem um filho à esquerda, este passa a ser o filho e `C` passa a ser o pai, e em seguida é aplicada uma rotação à direita
- *Corner case*: se, no início do passo anterior, `C` era a raiz da árvore, a raiz deve apontar para o filho à esquerda antes da rotação
- Se `C` não tem filhos à esquerda, `G` passa a armazenar o valor de `C` e `C` passa a apontar para o seu filho à direita
- Quando `C` apontar para nulo, a árvore estará completamente degenerada, formando a espinha dorsal

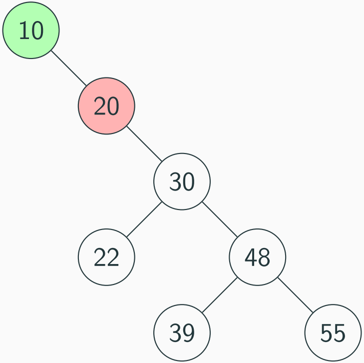
Exemplo de árvore desbalanceada



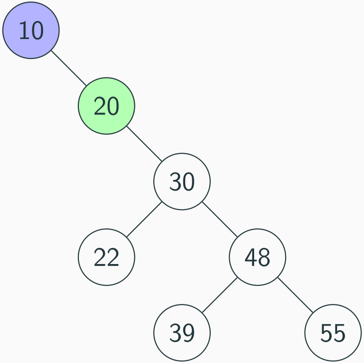
Exemplo de geração do backbone



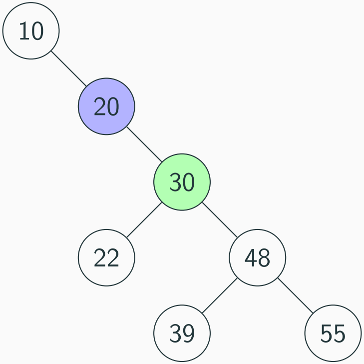
Exemplo de geração do backbone



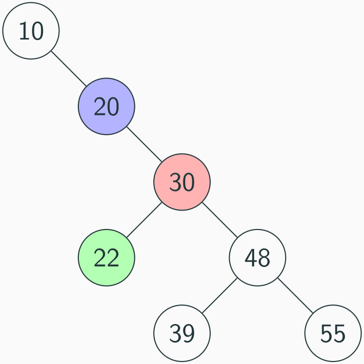
Exemplo de geração do backbone



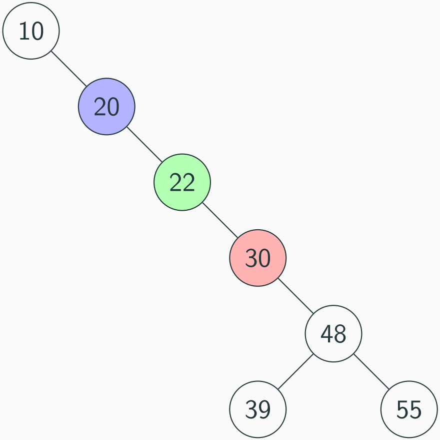
Exemplo de geração do backbone



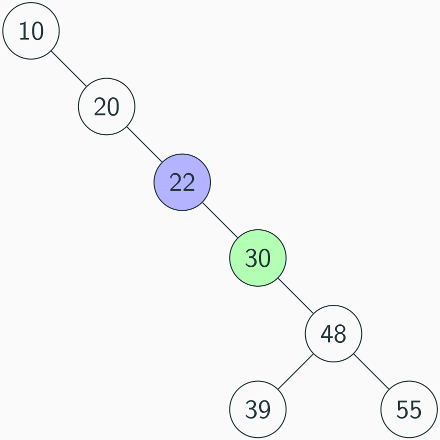
Exemplo de geração do backbone



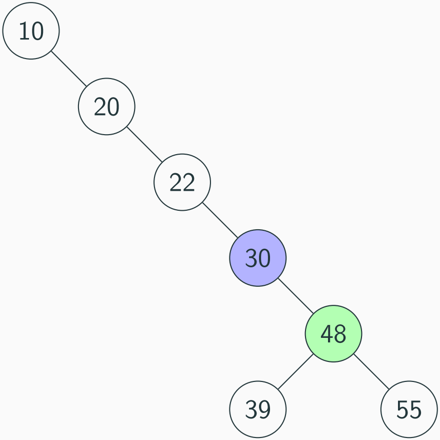
Exemplo de geração do backbone



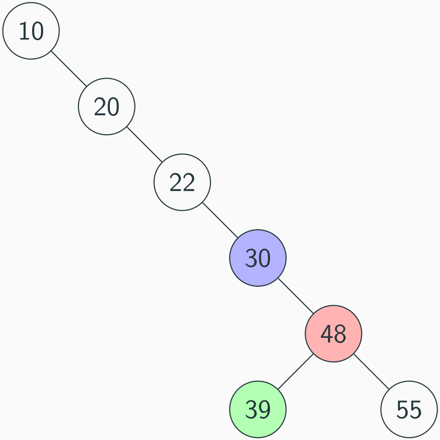
Exemplo de geração do backbone



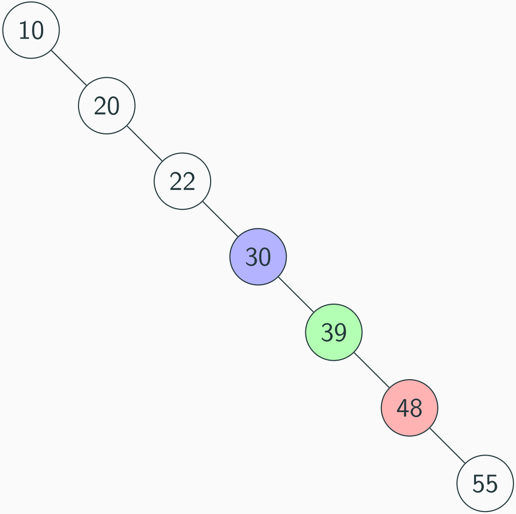
Exemplo de geração do backbone



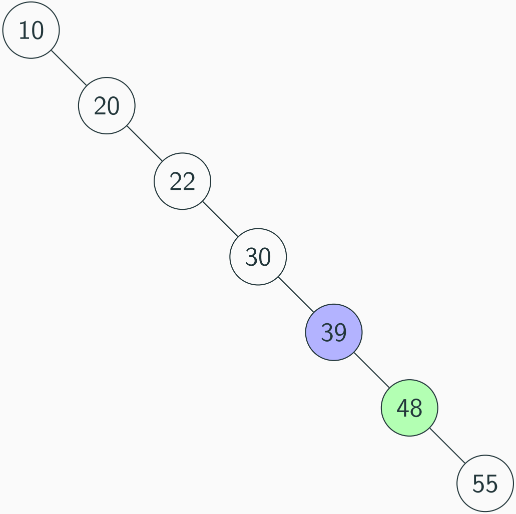
Exemplo de geração do backbone



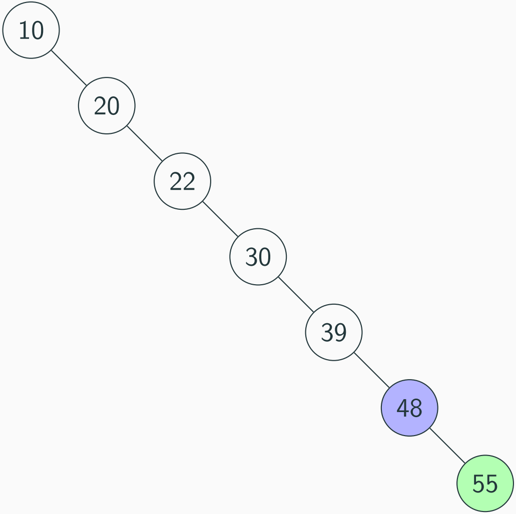
Exemplo de geração do backbone



Exemplo de geração do backbone



Exemplo de geração do backbone



Implementação da geração da espinha dorsal

```
template<typename T>
class BST {
public:
    BST() : root(nullptr) {}

private:
    struct Node {
        T info;
        Node *left, *right;
    };

    Node *root;

```

Implementação da geração da espinha dorsal

```
    void backbone()
    {
        Node *C = root, P = nullptr, G = nullptr;

        while (C != nullptr)
        {
            if (C->left) {
                if (C == root)
                    root = C->left;

                P = C;
                C = C->left;
                rotate_right(G, P, C);
            }
            else {
                G = C;
                C = C->right;
            }
        }
    }
};
```

Reorganização da espinha dorsal em uma árvore perfeitamente balanceada

- Com o objetivo de gerar uma árvore perfeitamente balanceada a partir da espinha dorsal é definida uma função de transformação
- Esta função realiza uma série de rotações à esquerda
- Antes de cada rotação, os ponteiros `G, P, C` são atualizados, movendo-se para a esquerda, dois passos por vez
- Os dois casos especiais envolvem a raiz
- Se o pai aponta para nulo, o filho será a raiz da árvore
- Se o pai for a raiz, a raiz passará a ser igual ao filho
- O número de rotações a serem feitas está relacionado com o logaritmo de N na base 2, onde N é o número de nós da árvore

Implementação do algoritmo DSW

```cpp
template<typename T>
class BST {
public:
    BST() : root(nullptr) {}

    void DSW()
    {
        backbone();

        int n = size();
        int m = (1 << ((int) floor(log(n+1)/log(2)))) - 1;

        // n - m = folhas que não estão na árvore completa
        transform(n - m);

        while (m > 1)
        {
            m = m/2;
            transform(m);
        }
    }
```

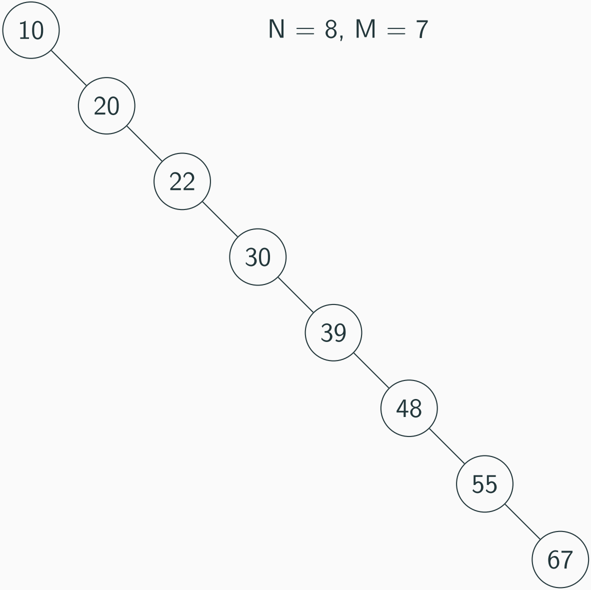
Implementação do algoritmo DSW

```

private:
    struct Node {
        T info;
        Node *left, *right;
    };

    Node *root;

    void transform(int qtd)
    {
        Node *G = nullptr, *P = nullptr, *C = nullptr;

        while (qtd--)
        {
            for (j = 0; j <= 1; j++)
            {
                // Atualiza o pai e o avô
                G = P;
                P = C;

```

Implementação do algoritmo DSW

```
                // Define o filho
                if (C)
                    C = C->right;
                else if (P == nullptr)
                    C = root;
            }

            // Atualiza o root, quando necessário
            if (P == root)
                root = C;

            rotate_left(G, P, C);
        }
    }

};
```

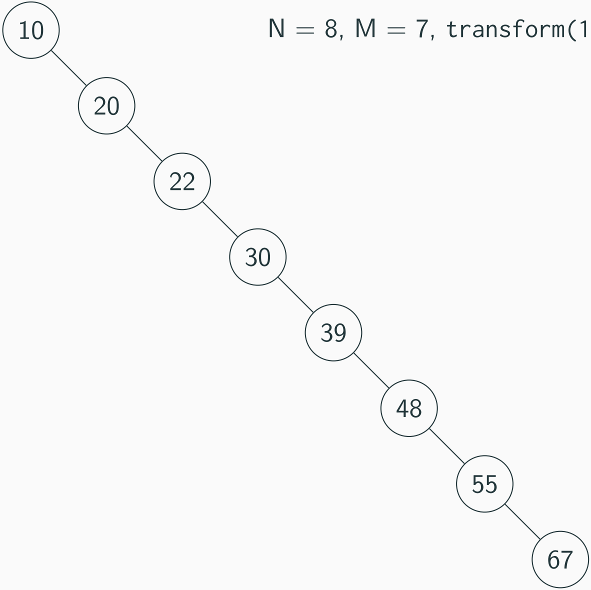
Algoritmo DSW: transformação da espinha dorsal



N = 8, M = 7

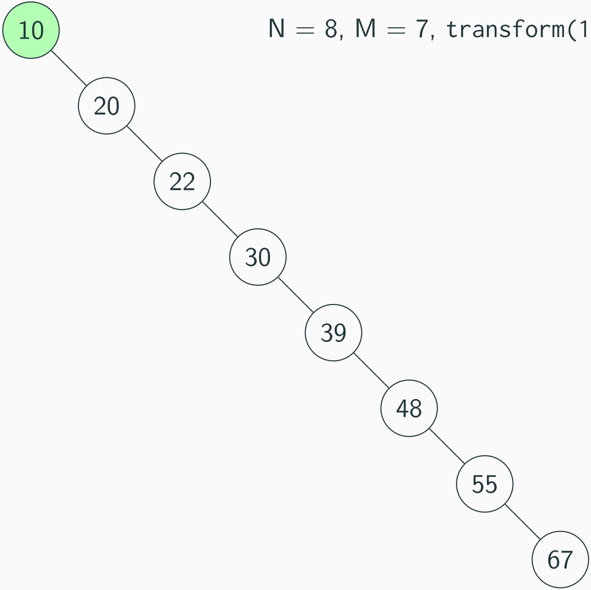
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 7$, `transform(1)`

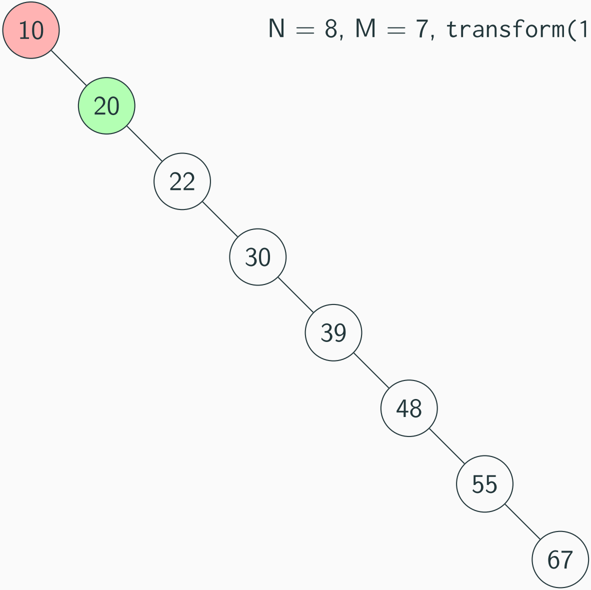


Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 7, `transform(1)`



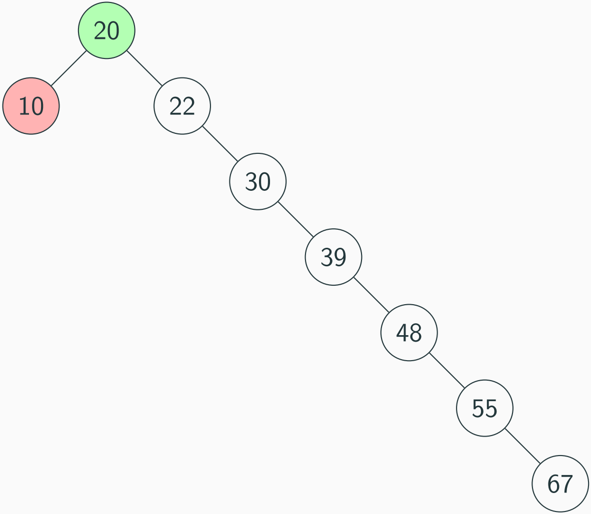
Algoritmo DSW: transformação da espinha dorsal



N = 8, M = 7, `transform(1)`

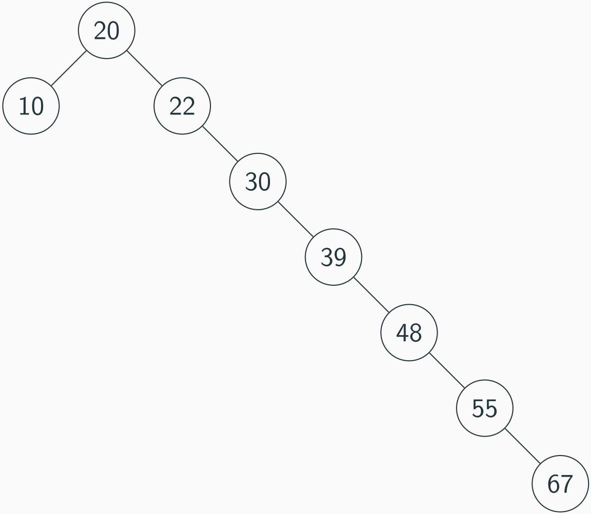
Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 7, `transform(1)`



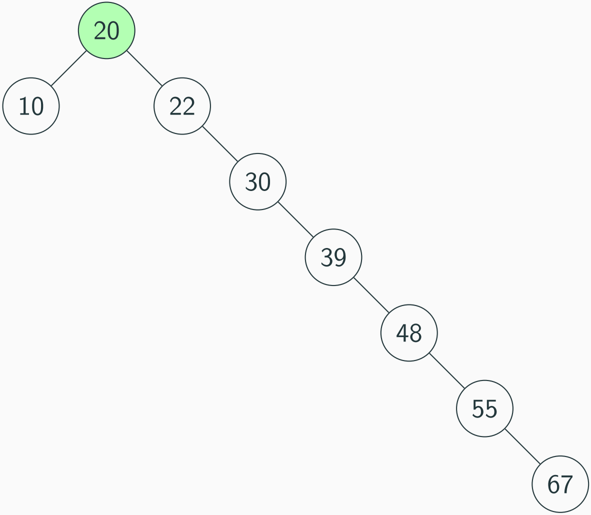
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 3$, `transform(3)`



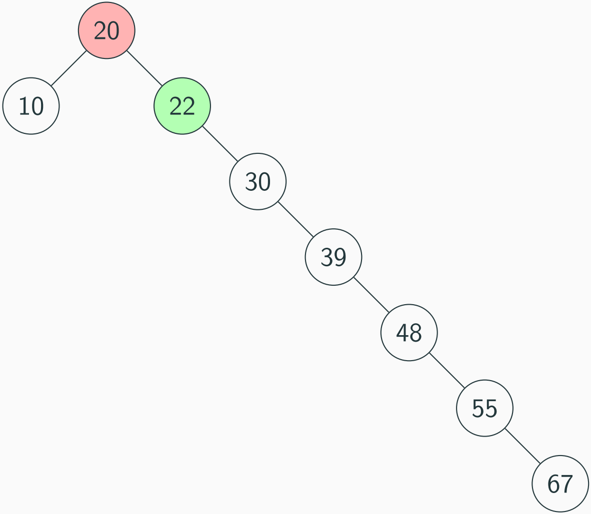
Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 3, `transform(3)`



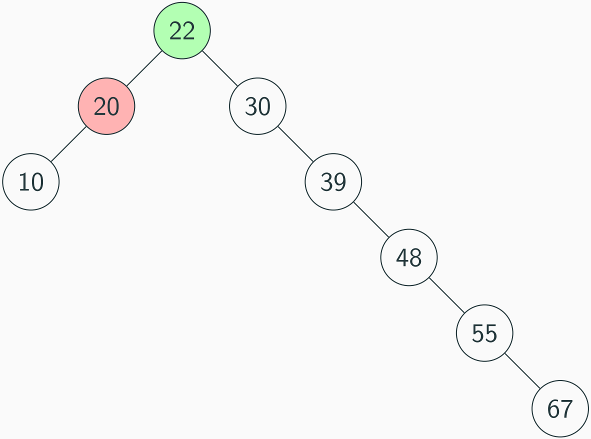
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 3$, `transform(3)`



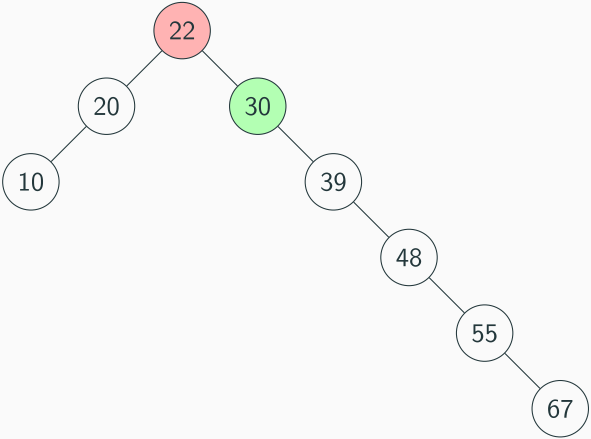
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 3$, `transform(3)`



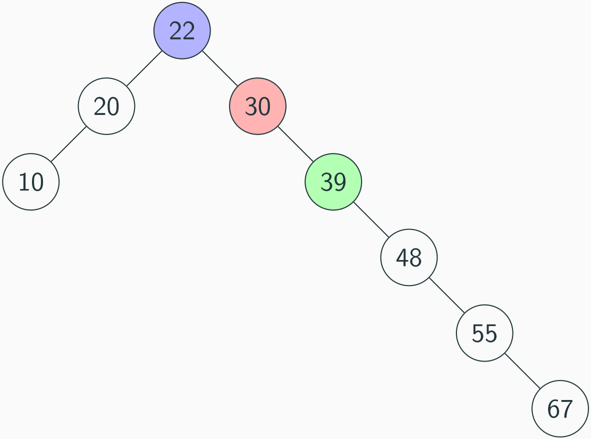
Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 3, `transform(2)`



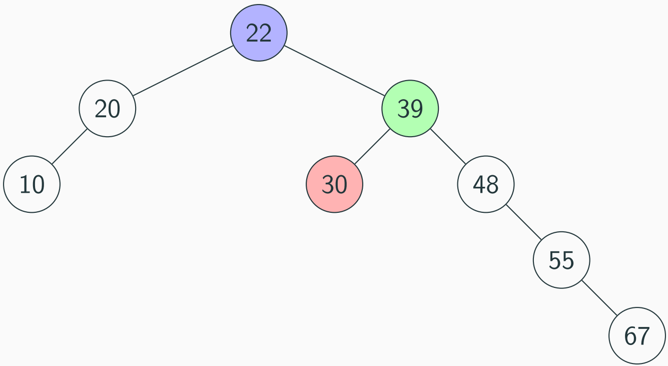
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 3$, `transform(2)`



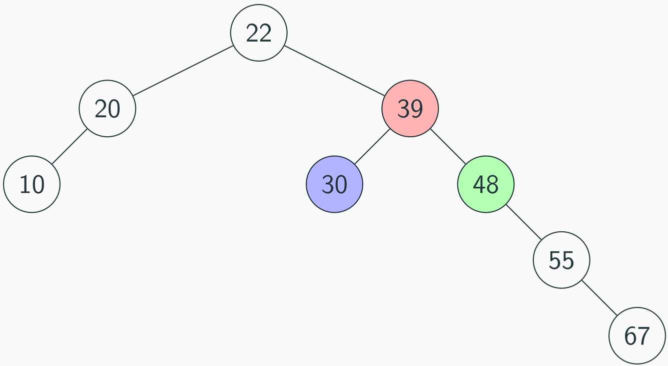
Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 3, `transform(2)`



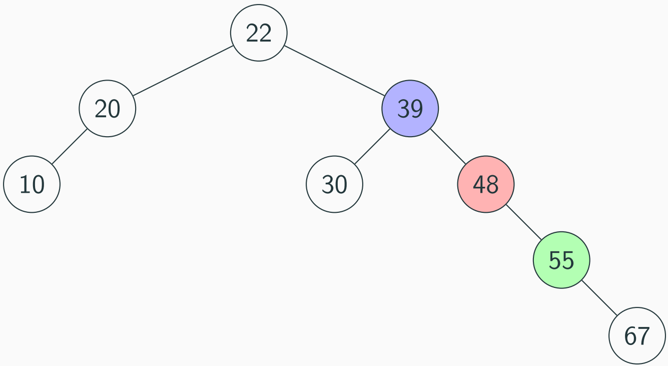
Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 3, `transform(1)`



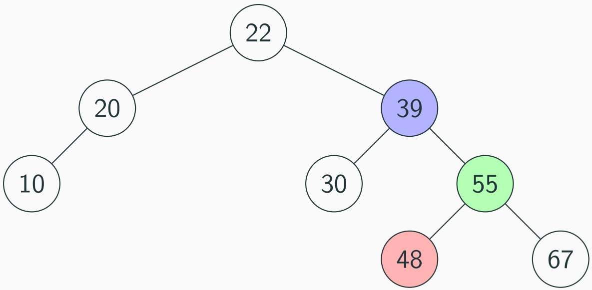
Algoritmo DSW: transformação da espinha dorsal

N = 8, M = 3, `transform(1)`



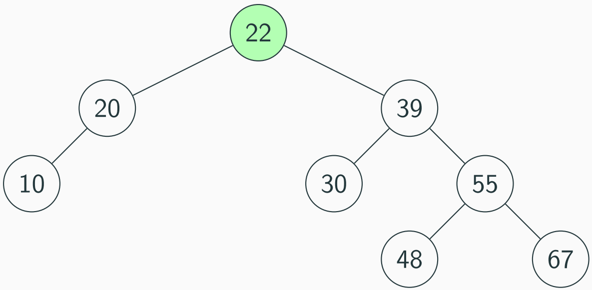
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 3$, `transform(1)`



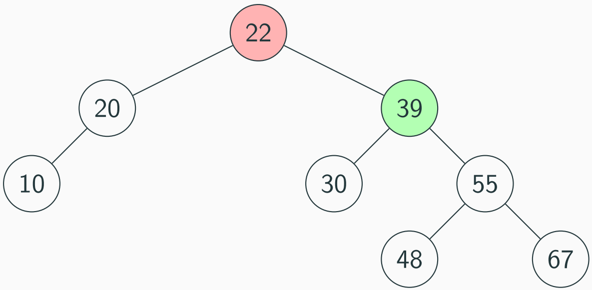
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 1$, `transform(1)`



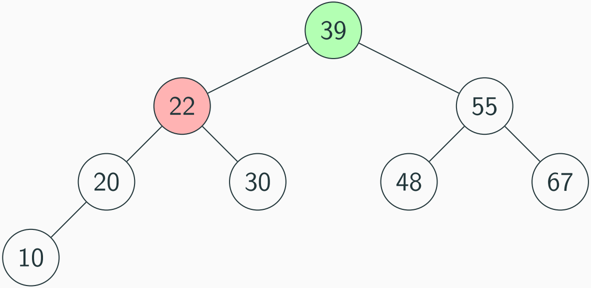
Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 1$, `transform(1)`



Algoritmo DSW: transformação da espinha dorsal

$N = 8$, $M = 1$, `transform(1)`



Referências

1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. **KERNIGHAN**, Bryan; **RITCHIE**, Dennis. *The C Programming Language*, 1978.
3. **STROUSTROUP**, Bjarne. *The C++ Programming Language*, 2013.
4. C++ Reference[1].
5. Full vs. Complete Binary Trees, acesso em 19/03/2019.

[1] https://en.cppreference.com/w/