Algoritmos e Estruturas de Dados 2 - UFSCar

Vinícius O. Guimarães

Prof. Mário César San Felice

Julho 2023

1 Lista 2 - Árvores AVL e rubro-negras

1. **Uma árvore é balanceada no sentido AVL se, para cada nó x, as alturas das subárvores que têm raízes x->esq e x->dir diferem de no máximo uma unidade. Escreva uma função que decida se uma dada árvore é balanceada no sentido AVL. Procure escrever sua função de modo que ela visite cada nó no máximo uma vez.**

   Como sabemos, para que uma árvore seja AVL, as alturas das subárvores de um nó qualquer tem que diferirem de no máximo uma unidade.

   Ex: Se temos um nó qualquer e minha subárvore da esquerda desse nó tiver altura 3 e a subárvore da direita tiver altura 1, então a diferença entre as alturas dessas duas subárvores é $3 - 1 = 2$, sendo portanto uma árvore **não** AVL, pois $2 > 1$.

   Para resolver o problema, podemos percorrer a árvore recursivamente, calculando as alturas das subárvores e verificando se a diferença entre elas é de no máximo uma unidade.

```c
// Considerando a árvore como nós neste formato:
typedef struct noh {
   Noh *pai;
   Noh *esq;
   Noh *dir;
   int valor;
} Noh;

// Temos a função para verificar se uma árvore é AVL
// O parâmetro result vai receber o resultado, 0 ou 1 para saber
// se a árvore é AVL ou não.
int verificarAVL(Noh *arvore, int *result) {
    if (arvore == NULL) {
        return -1;
    }

    if ((*result) == 0) {
        return (*result);
    }

    int alturaEsq = verificarAVL(arvore->esq, result);
    int alturaDir = verificarAVL(arvore->dir, result);

    // Quando result for 0 iremos parar de verificar as alturas, por isso o return.
    if ((*result) == 0) {
        return (*result);
    }
```

```c
        if (alturaEsq > alturaDir) {
            // Verificando a diferença entre as alturas
            // para que a diferença seja no máximo 1
            if ((alturaEsq - alturaDir) > 1) {
                (*result) = 0;
            }
            return alturaEsq + 1;
        }
        // Verificando a diferença entre as alturas
        // para que a diferença seja no máximo 1
        if ((alturaDir - alturaEsq) > 1) {
            (*result) = 0;
        }
        return alturaDir + 1;
    }
```

2. **Complete a função de inserção em árvore AVL das notas de aula de modo que ela também trate os casos de inserção à direita.**

   Relembrando os casos de rotação que temos:

   Rotações possíveis no caso de inserção na esquerda da raiz:

   - Rotação para a direita
   - Rotação esquerda-direita

   Rotações possíveis no caso de inserção na direita da raiz:

   - Rotação para a esquerda
   - Rotação direita-esquerda

   Para conseguir resolver os casos de inserção na esquerda e direita, foram utilizados os seguintes arquivos: AVL.h, AVL.c, main.c. O arquivo main.c foi feito para testar a inserção.

   Arquivo AVL.h:

```c
#ifndef AVL_H
#define AVL_H

typedef struct node Node;

typedef struct node {
    int bal;
    int key;
    int content;
    Node *left;
    Node *right;
    Node *parent;
} Node;

Node *create();
Node *newNode();
Node *insertAVL(Node *tree, int key, int valor, int *heightIncreased);
int getHeight(Node *tree);
```

```c

#endif
```

Arquivo AVL.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "AVL.h"

// Cria a árvore AVL
Node *create() {
    return NULL;
}


int getHeight(Node *tree) {
    if (tree == NULL) {
        return -1;
    }

    int leftHeight = getHeight(tree->left) + 1;
    int rightHeight = getHeight(tree->right) + 1;

    return leftHeight >= rightHeight ? leftHeight : rightHeight;
}

/**
 ** Caso 0
 *      Se a altura da subárvore não aumentou, então devolve que não houve um aumento da altura
 ** Caso 1
 *      Se a árvore era vazia, então crie um nó com os dois filhos sendo NULL e balanceamento 0
 *      Retorne dizendo a altura aumentou
 ** Caso 2
 *      Se inseriu na subárvore mais baixa e a altura desta aumentou
 *      Mude o balanceamento da raiz para 0 e retorne dizendo que a altura da árvore não aumentou
 ** Caso 3
 *      Se inseriu em qualquer uma das subárvores quando a altura delas eram iguais (bal da raiz sendo 0)
 *      Mude o balanceamento para 1 ou -1 dependendo do lado da inserção
 *      Devolva que a altura da árvore aumentou
 ** Caso 4
 *      Se inseriu na subárvore maior e a altura dela aumentou
 *      Realizar rotações para restaurar as propriedades AVL
 **     Caso 4.1 (Balanceamento da árvore sendo -1)
 *          Após inserir na subárvore da esquerda o balanceamento dela é -1
 *          Realizar rotação da subárvore para a direita.
 *
 *          OBS (Temos aqui também o caso de inserir na subárvore da direita e o balanceamento dela ser -1)
```

```c
    *          Realizar rotação dupla direita esquerda
    **     Caso 4.2 (Balanceamento da árvore sendo 1)
    *          Após inserir na subárvore da esquerda o balanceamento dela é 1
    *          Realizar rotação dupla esquerda direita
    *
    *          OBS (Temos aqui também o caso de inserir na subárvore da direita e o balanceamento dela ser 1)
    *          Realizar rotação da subárvore para a esquerda
    */

Node *insertAVL(Node *node, int key, int value, int *heightIncreased) {
    if (node == NULL) {
        printf("node == null\n");
        // Caso 1
        Node *new = newNode();
        new->key = key;
        new->content = value;
        *heightIncreased = 1;
        return new;
    }
    // inserção na esquerda
    else if (key <= node->key) {
        node->left = insertAVL(node->left, key, value, heightIncreased);
        node->left->parent = node;

        if (*heightIncreased == 1) {
            // Caso 2: se inseriu na menor subárvore
            if (node->bal == 1) {
                node->bal = 0;
                *heightIncreased = 0;
            }

            // Caso 3: se ambas as subárvores tinham a mesma altura
            // Se temos um árvore com apenas dois nós, um nó A raiz e um nó B subárvore de A na esquerda
            // Quando formos inserir um nó na subárvore B, então passará por aqui logo após retornar o nó criado
            else if (node->bal == 0) {
                node->bal = -1;
                *heightIncreased = 1;
            }
            // Caso 4: inseriu na maior subárvore
            else if (node->bal == -1) {
                // se o balanceamento da árvore é -1, então significa que a subárvore da esquerda está desbalanceada
                // Portanto, o balanceamento da subárvore na esquerda só poderá ser -1 ou 1, pois ela está desbalanceada

                // inseriu na esquerda da subárvore da esquerda (os valores de node->left->bal já estão atualizados)
```

```
                // Caso 4.1
                if (node->left->bal == -1) {
                    node = rightRotation(node);
                    node->right->bal = 0; // A antiga raiz da árvore agora é filho direito da raiz atual
                }
                // inseriu na direita da subárvore da esquerda
                // Caso 4.2
                else if (node->left->bal == 1) {
                    node = leftRightRotation(node);

                    // Saberemos onde que inseriu o elemento

                    if (node->bal == 0) { // Nesse caso, o próprio Z que tinha sido inserido
                        // Se sabemos que a altura aumentou e o balanceamento de Z ficou 0
                        // Então sabemos que o filho direito de Y era na verdade NULL e Z foi inserido
                        // Se a altura aumentou na inserção de Z, então o filho esquerdo de Y também era NULL

                        // Após realizar a rotação dupla, sabemos que o balanceamento da raiz Z será 0
                        // Assim, se Y agora tem ambas as subárvores sendo NULL e o balanceamento da raiz é 0;
                        // Então, sabendo que a subárvore B2 de X é NULL e que o balanceamento de X deve ser 0 para que o
                        // balanceamento da raiz seja 0, então a subárvore C de X também deverá ser NULL
                        node->left->bal = 0;
                        node->right->bal = 0;
                    } else if (node->bal == -1) { //Inseriu na esquerda de Z
                        node->left->bal = 0;
                        node->right->bal = 1;
                    } else if (node->bal == 1) {  //Inseriu na direita de Z
                        node->left->bal = -1;
                        node->right->bal = 0;
                    }
                }
                //Depois das rotações, a raiz estará balanceada
                node->bal = 0;
                // Mesmo depois de realizar todos os processos de rotações, a altura da árvore não terá aumentado
                *heightIncreased = 0;
            }

            int leftHeight = getHeight(node->left);
            int rightHeight = getHeight(node->right);

```

```
                if ((rightHeight - leftHeight) != node->bal) {
                  printf("Erro inserção esquerda! Bal: %d, Altura D: %d, Altura E: %d\n", node->bal, rightHeight, leftHeight);
                } else {
                  printf("\nAltura correta inserção esquerda.\n");
                }
            }
        }
        // Inseriu na subárvore da direita
        else {
            node->right = insertAVL(node->right, key, value, heightIncreased);
            node->right->parent = node;

            if (*heightIncreased) {
                // Inseriu na menor subárvore
                if (node->bal == -1) {
                    node->bal = 0;
                    *heightIncreased = 0;
                }
                // Ambas as subárvores tinham tamanho igual
                else if (node->bal == 0) {
                    node->bal = 1;
                    *heightIncreased = 1;
                }
                // Inseriu na maior subárvore
                else { // node->bal == 1
                    // Inseriu node na direita da subárvore da direita
                    if (node->right->bal == 1) {
                        node = leftRotation(node);
                        node->left->bal = 0;
                    }
                    // Inseriu node na esquerda da subárvore da direita
                    else if (node->right->bal == -1) {
                        node = rightLeftRotation(node);

                        // O próprio Z foi o nó inserido.
                        if (node->bal == 0) {
                            node->right->bal = 0;
                            node->left->bal = 0;
                        } else if (node->bal == -1) {
                            // Inseriu em B1 de Z (Esquerda de Z)
                            node->left->bal = 0;
                            node->right->bal = 1;
                        } else if (node->bal == 1) {
                            // Inseriu em B2 de Z (Direita de Z)
                            node->left->bal = -1;
                            node->right->bal = 0;
                        }
                    }
                    node->bal = 0;
```

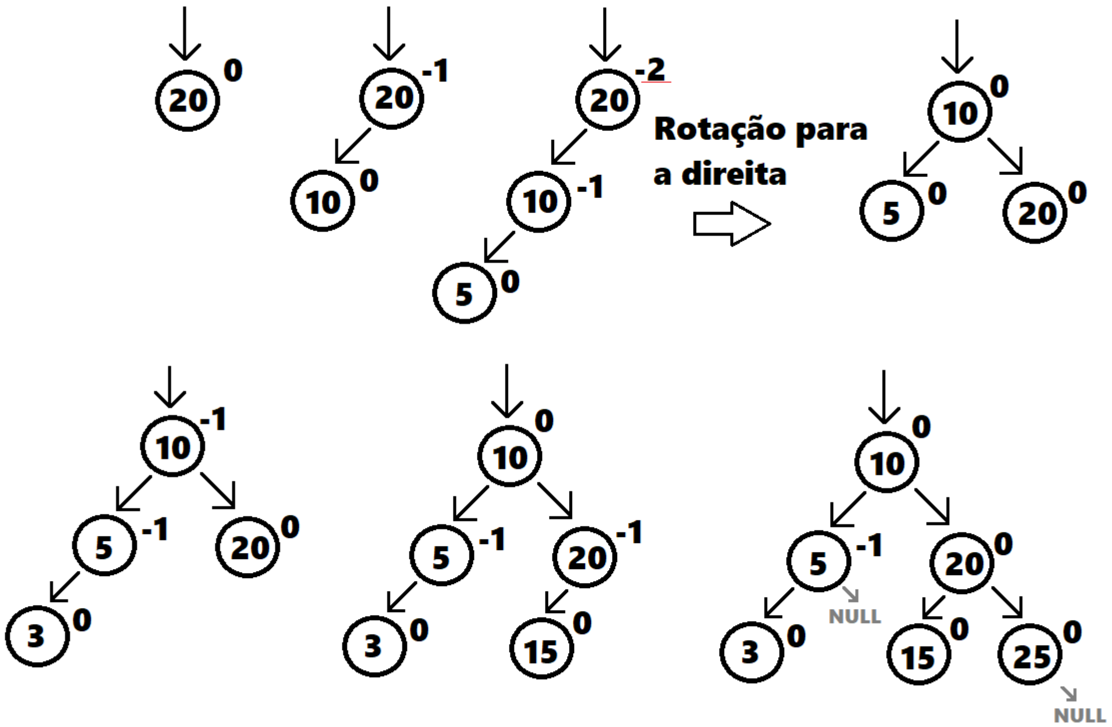
```c
                *heightIncreased = 0;
            }

            // Essa parte serve para verificar se realmente estamos organizando a árvore de forma que ela seja uma árvore AVL.
            // Caso alguma coisa errada seja feita, a altura das subárvores da esquerda e da direita irão se diferenciar em mais do que uma unidade.
            // Ou seja, como é apenas para teste, pode remover essa parte sem problemas caso queira.

            int leftHeight = getHeight(node->left);
            int rightHeight = getHeight(node->right);
            if ((rightHeight - leftHeight) != node->bal) {
              printf("Erro inserção direita! Bal: %d, Altura D: %d, Altura E: %d\n", node->bal, rightHeight, leftHeight);
            } else {
              printf("\nAltura correta inserção direita.\n");
            }
        }
    }
    return node; // Retornar o nó para que fiquem ligados
}
```

Vamos testar essa função de inserção no arquivo main.c:

```c

#include <stdio.h>
#include <stdlib.h>
#include "AVL.h"

int main() {
    Node *arvore = create();
    int heightIncreased = 0;

    /* INSERINDO NA ESQUERDA DA RAIZ */
    arvore = insertAVL(arvore, 10, 555, &heightIncreased);
    arvore = insertAVL(arvore, 5, 556, &heightIncreased);
    arvore = insertAVL(arvore, 3, 557, &heightIncreased);

    printf("Árvore: %d\n", arvore->key); // 5
    printf("Sub-esq: %d\n", arvore->left->key); //3
    printf("Sub-dir: %d\n", arvore->right->key); //10

    Node *arvore2 = create();
    heightIncreased = 0;
    arvore2 = insertAVL(arvore2, 10, 555, &heightIncreased);
    arvore2 = insertAVL(arvore2, 5, 555, &heightIncreased);
    arvore2 = insertAVL(arvore2, 7, 555, &heightIncreased);

    printf("Árvore: %d\n", arvore2->key); //7
```

```c
        printf("Sub-esq: %d\n", arvore2->left->key); //5
        printf("Sub-dir: %d\n", arvore2->right->key); //10

        /* INSERINDO NA DIREITA DA RAIZ */
        Node *arvore3 = create();
        arvore3 = insertAVL(arvore3, 10, 555, &heightIncreased);
        arvore3 = insertAVL(arvore3, 20, 556, &heightIncreased);
        arvore3 = insertAVL(arvore3, 30, 557, &heightIncreased);

        printf("Árvore: %d\n", arvore3->key); // 20
        printf("Sub-esq: %d\n", arvore3->left->key); //10
        printf("Sub-dir: %d\n", arvore3->right->key); //30

        Node *arvore4 = create();
        heightIncreased = 0;
        arvore4 = insertAVL(arvore4, 10, 555, &heightIncreased);
        arvore4 = insertAVL(arvore4, 20, 556, &heightIncreased);
        arvore4 = insertAVL(arvore4, 15, 557, &heightIncreased);

        printf("Árvore: %d\n", arvore4->key); //15
        printf("Sub-esq: %d\n", arvore4->left->key); //10
        printf("Sub-dir: %d\n", arvore4->right->key); //20
    }
```

3. **Dê um exemplo de uma árvore binária de busca cujas folhas têm todas a mesma profundidade, mas nem todo caminho da raiz até um apontador NULL passa pelo mesmo número de nós.**



Vamos considerar a árvore acima (de raiz com chave 10), com os elementos 3, 5, 10, 15, 20 e 25. Podemos observar que o primeiro ponteiro NULL na esquerda da raiz é encontrado na direita do elemento 5: ou seja, passamos por 2 nós (considerando a raiz) até chegar nesse ponteiro NULL.

Já no lado direito da raiz, temos que um dos ponteiros NULL é encontrado na direita do elemento 25: ou seja, precisamos passar por um total de 3 nós (considerando a raiz) até encontrar esse ponteiro NULL.

4. **Seja x um nó de uma árvore rubro-negra. Mostre que todos os caminho que levam de x até um apontador NULL têm o mesmo número de nós pretos**

Dados as seguintes regras que uma árvore rubro-negra deve seguir:

1 Cada nó é vermelho ou preto

2 Raiz é sempre preta

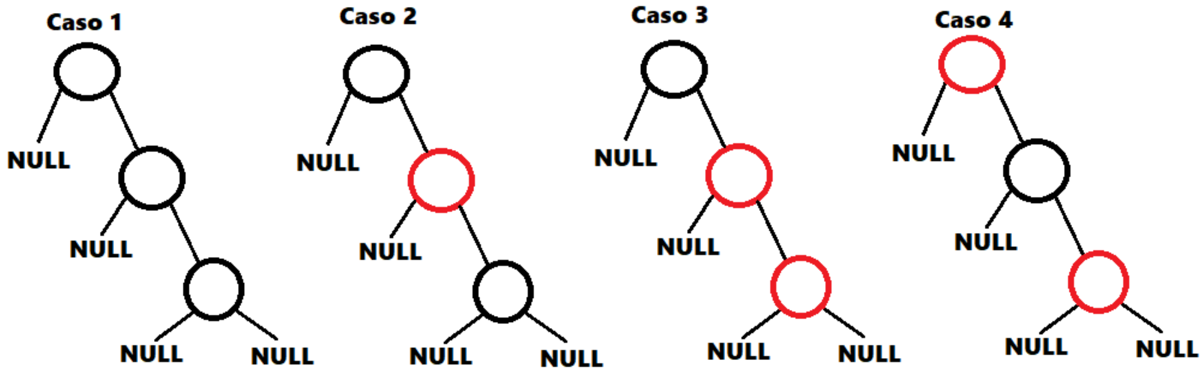
3 Dois nós vermelhos não podem ser adjacentes

- ou seja, um nó vermelho só pode ter filhos pretos

4 Todo caminho da raiz até um apontador NULL (Caminho raiz-NULL) tem o mesmo número de nós pretos

- Vamos pensar nisso como sendo buscas mal sucedidas

Para mostrar que a afirmação da questão é verdadeira temos que observar a forma com que uma árvore rubro negra se torna inválida. Com isso, vamos analisar a imagem abaixo:



- **Caso 1**

  Como visto acima, a ávore no caso 1 tem 3 nós pretos organizados de forma que a regra 4 é quebrada. Isso acontece pois existem caminhos da raiz até ponteiros nulos onde a quantidade de nós pretos é diferente.

  Ex: A esquerda da raiz tem um ponteiro NULL (Caminho de 1 nó preto até o ponteiro NULL) e o nó mais a direita da raiz tem ponteiros NULL (Caminho de 3 nós pretos até o ponteiro NULL).

  Com isso, essa árvore **não** é uma árvore rubro-negra válida.

- **Caso 2**

  No caso 2, a árvore possui dois nós pretos e um nó vermelho entre eles. Da mesma forma que o caso 1, temos que essa árvore não é válida pois existem caminhos da raiz até ponteiros NULL onde a quantidade de nós pretos não é a mesma (Caso na esquerda da raiz e no nó mais a direita da raiz)

- **Caso 3**

  No caso 3 temos dois nós adjacentes sendo vermelhos, o que quebra a regra 3, que diz que dois nós adjacentes não podem ser vermelhos.

- **Caso 4**

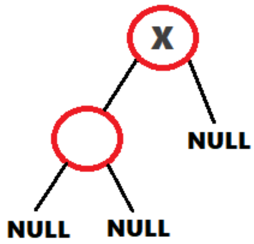
  Esse é um caso interessante e fácil de identificar o erro: A raiz é um nó vermelho, contrariando a regra de que a raiz deve ser sempre um nó preto.

Com isso, analisando todas as possibilidades de árvores rubro-negra válidas, temos que a quantidade de nós pretos até um ponteiro NULL qualquer, vai ser sempre a mesma.

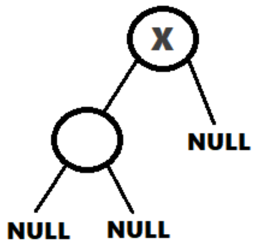
5. **Suponha que x é um nó de uma árvore rubro-negra. Suponha que x.dir == NULL mas x.esq != NULL. Prove que x.esq.vermelho == 1 e x.esq não tem filhos (ou seja, x.esq.esq == NULL e x.esq.dir == NULL).**

Para provar esta afirmação, temos que pensar em duas partes:

- Se nós temos um nó X qualquer onde o filho da esquerda é vermelho e o filho da direita é NULL, então X tem que ser obrigatoriamente preto, pois se fosse vermelho quebraria a 3º regra das árvores rubro-negras: que não permite ter nós vermelhos adjacentes.



- Da mesma forma, se o filho da esquerda desse nó X (X.esq) fosse preto e sem filhos (ou seja, X.esq.esq == NULL e X.esq.dir == NULL) também não teríamos uma árvore rubro-negra, pois a 4º regra das árvores rubro-negras seria quebrada: teríamos caminhos que levam a ponteiros NULL com quantidades diferentes de nós pretos.



Ou seja, para que seja uma árvore rubro-negra, o nó X tem que ser preto e o seu filho da esquerda (X.esq) deve ser vermelho.