**UFES / CEUNES**

**DEPARTAMENTO COMPUTAÇÃO E ELETRÔNICA**

**CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

CHRISTIAN JONAS OLIVEIRA

JOÃO VICTOR DO ROZÁRIO RECLA

# IMPLEMENTAÇÃO DA ÁRVORE AVL PARA TIPOS GENÉRICOS DE DADOS

SÃO MATEUS – ES

2021

## Sumário

Neste trabalho, o objetivo era implementar a estrutura AVL e suas funções para tipos genéricos de dados. Listaremos a seguir o TAD e todas as suas funções e as devidas explicações necessárias para a compreensão daquilo que foi programado.

## 1. Definição da estrutura AVL

```
typedef struct AVL{
    void *key;
    int balance;
    struct AVL *left, *right;
}AVL;
```

## 2. Inserção

AVL *insert_AVL(AVL *node, **void** *element, **int** *h, **int** (*compare) (**void** *, **void** *));

A inserção recebe um ponteiro para o elemento genérico a ser inserido e a função que comparará a chave do elemento para dizer em qual nó da sub-árvore ele será inserido.

Primeiramente é verificado se o nó passado é nulo, isto também nos informará se o processo recursivo já chegou na extremidade inferior de uma sub-árvore (nó folha). Ao entrar nessa verificação será alocado um novo espaço de memória para um nó na arvore que terá como chave o novo elemento a ser inserido, retornado-o caso a alocação seja bem sucedida. Atualiza-se a variável "h", para indicar crescimento da árvore.

```
if(!node){

        AVL *new_node = create_AVL(element);

        if(new_node) *h = 1;

        return new_node;
}
```

Caso não esteja em um nó folha ou a árvore não seja nula, ocorre a chamada recursiva da função afim de identificar a real posição do elemento a ser inserido na árvore.
A primeira condição trata a situação em que o elemento a ser inserido possui chave menor do que a do nó atual, realizando a chamada da função para a sub-árvore esquerda.

```
// tratamento do caso onde a chave a ser inserida é menor que a chave
do nó atual
if(compare(element, node->key) == 0){

        node->left = insert_AVL(node->left, element, h, compare);

        if(*h == 1){

                // analise de casos de crescimento
                switch(node->balance){

                        // crescimento para a direita
                        case -1: node->balance = 0; *h = 0; break;

                        // equilibrado;
                        case 0: node->balance = 1; break;

                        // crescimento para a esquerda
                        case 1: node = right_rotation(node, h); break;
                }
        }
}
```

Caso a condição anterior não seja válida, há uma nova verificação, que tratará a situação em que a chave do elemento a ser inserido é maior do que a do nó atual. Realizando assim, a chamada da função para a sub-árvore direita.

```
// tratamento do caso onde a chave a ser inserida eh maior que a chave
do node atual
else if(compare(element, node->key) == 1){

        node->right = insert_AVL(node->right, element, h, compare);

        if(*h == 1){

                // analise de casos de crescimento
                switch(node->balance){

                        // crescimento para a direita;
                        case -1: node = left_rotation(node, h); break;

                        // equilibrado
                        case 0: node->balance = -1; break;

                        // crescimento para a esquerda
                        case 1: node->balance = 0; *h = 0; break;
                }
        }
}
```

Caso a inserção provoque um desbalanceamento, o caso "-1" do *switch case*, realizará as devidas rotações (para esquerda ou para a direita) em ambas as verificações.

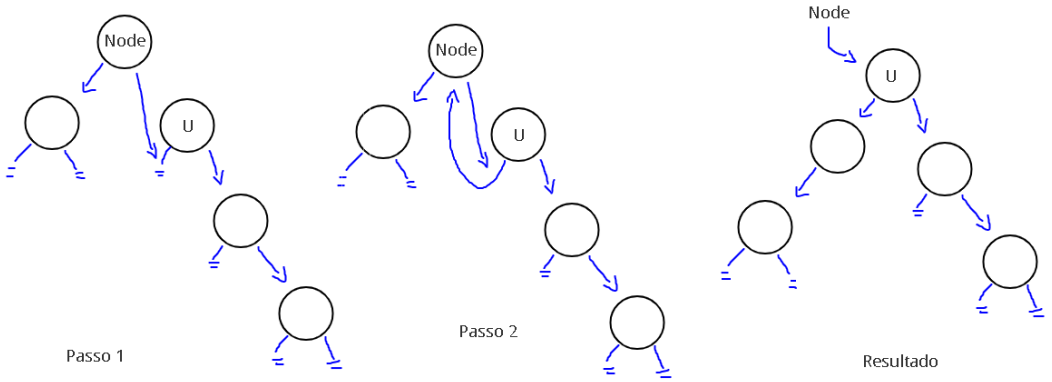
## 3. Rotações

### 3.1 Rotação para a esquerda

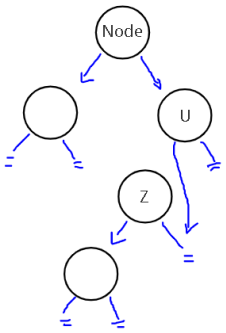
```
AVL *left_rotation(AVL *node, int *h);
```

A rotação para a esquerda, definida pela função *left_rotation*, funcionará da seguinte forma:

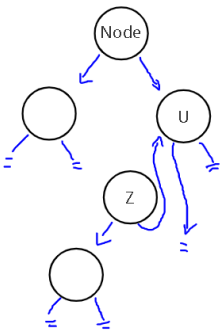
```
// caso onde a arvore esta "pesada" para a direita - (Rotação Simples
a esquerda)

AVL *u = node->right;

if(u->balance == -1){

        node->right = u->left;
        u->left = node;
        node->balance = 0;
        node = u;

}
```



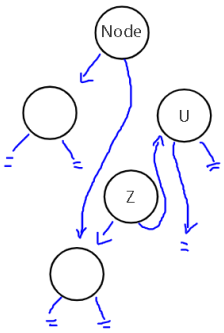
```
// caso onde a arvore esta "pesada" pra esquerda do filho a direita -
(Rotação Dupla a esquerda)

else{
        AVL *z;
        z = u->left;
        u->left = z->right;
        z->right = u;
        node->right = z->left;
        z->left = node;

        /* re-balanceamento do campo "balance" da AVL */
        if(z->balance == 1) u->balance = -1;
        else u->balance = 0;

        if(z->balance == -1) node->balance = 1;
        else node->balance = 0;

        if(z->balance == 0) u->balance = weight_AVL(u);

        node = z;
}
```



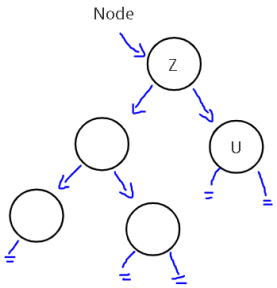

Passo 1




Passo 2




Passo 3




Resultado

### 3.2 Rotação para a direita

```
AVL *right_rotation(AVL *node, int *h);
```

A rotação para a direta, definida pela função *right_rotation*, funciona de forma análoga a função *left_rotation*, porém de forma "espelhada".

```
// caso onde a arvore esta "pesada" para a esquerda - (Rotação Simples
a direita)
AVL *u = node->left;
if(u->balance == 1){

        node->left = u->right;
        u->right = node;
        node->balance = 0;
        node = u;
}
```

Neste caso de rotação os índices do balanceamento são atualizados de maneira um pouco diferente, já que o método é "espelhado".

```
// caso onde a arvore esta "pesada" para a direita do filho a esquerda
- (Rotação Dupla a direita)
else{

        z = u->right;
        u->right = z->left;
        z->left = u;
        node->left = z->right;
        z->right = node;

        /* re-balanceamento do campo "balance" da AVL */
        if(z->balance == 1) node->balance = -1;
        else node->balance = 0;

        if(z->balance == -1) u->balance = 1;
        else u->balance = 0;

        if(z->balance == 0) u->balance = weight_AVL(u);

        node = z;
}
```

### 3.3 Algoritmos auxiliares das rotações

Em ambas as rotações, foi-se utilizada a função *weight_AVL*, responsável por calcular e retornar o peso do nó em que ela é aplicada.

```
int weight_AVL(AVL *avl){

        int left, right;

        // altura das sub-arvores da AVL
        left = height_AVL(avl->left);
        right = height_AVL(avl->right);

        // retorno do peso da AVL
        return left - right;
}
```

Que, por sua vez, utiliza a *height_AVL* para calcular a altura das sub-árvores.

```
int height_AVL(AVL *avl){

        int left, right;

        if(!avl) return 0;

        // calculo da altura das sub-arvores
        left = 1 + height_AVL(avl->left);
        right = 1 + height_AVL(avl->right);

        // retorno da altura da AVL
        if(left > right) return left;
        else return right;
}
```

## 4. Busca

```
AVL *search_AVL(AVL *node, void *element, int (*compare)(void *, void *));
```

Nessa função a ideia é buscar o nó que contém o elemento genérico à partir da análise de suas sub-árvores. O passo recursivo continuará sendo executado enquanto o elemento não for encontrado. O caso de parada se dá quando todos os elementos até a possível posição do elemento buscado, já foram analisados.

```
if(node){

        // caso onde o elemento buscado tem a chave maior
        if(compare(element, node->key) == 1)
                return search_AVL(node->right, element, compare);

        // caso onde o elemento buscado tem a chave menor
        else if(compare(element, node->key) == 0)
                return search_AVL(node->left, element, compare);

        // caso onde o elemento foi encontrado
        else return node;

}
```

## 5. Remoção

```
AVL *remove_AVL(AVL *node, void *element, int (compare) (void *, void *));
```

O algoritmo de remoção foi implementado de modo a re-apontar o nó com o elemento genérico a ser deletado, para a chave de um nó folha. Em seguida, a função de remoção é chamada recursivamente para o elemento na posição a qual o nó folha se encontra. Essa remoção se dá através da utilização de uma função auxiliar, *moreleft_AVL*, que irá segurar o nó mais a esquerda do filho à direita do nó a ser deletado (elemento seguinte da sequência das chaves). Vale ressaltar, que a chave a ser deletada está sendo guardada, através da função de busca, na função main, onde a mesma terá seu espaço de memória liberado após a impressão de seus dados.

```
// verifica-se se o nó passado é nulo (caso base)
if(!node) return node;

// tratamento do caso onde o elemento a ser removido tem a chave menor
if(compare(element, node->key) == 0){

        node->left = remove_AVL(node->left, element, compare);
        node->balance = weight_AVL(node);

        if(node->balance < -1) node = left_rotation(node, &j);
}
```

```
// tratamento do caso onde o elemento a ser removido tem a chave maior
else if(compare(element, node->key) == 1){

        node->right = remove_AVL(node->right, element, compare);
        node->balance = weight_AVL(node);

        if(node->balance > 1) node = right_rotation(node, &j);
}
```

No momento em que a chave é encontrada são realizadas verificações para saber se o nó que segura o elemento genérico possui filhos. Se o nó não possuir sub-árvore à esquerda, ou à direita, será retornado para a última chamada a sub-árvore existente liberando o ponteiro para a chave e o espaço de memória alocado para o nó.

```
// caso onde o elemento nao possui sub-arvore a esquerda
if(!node->left){

        // ponteiro para a sub-arvore direita
        AVL *tree = node->right;

        node->key = NULL;
        free(node->key);
        free(node);

        return tree;
}

// caso onde o elemento não possui sub-arvore a direita
else if(!node->right){

        // ponteiro para a sub-arvore esquerda
        AVL *tree = node->left;

        node->key = NULL;
        free(node->key);
        free(node);

        return tree;
}
```

Caso o nó, a ser deletado possua as duas sub-árvores ocorrerá um re-apontamento onde a sua chave receberá a chave resultante da busca do filho mais a esquerda de seu filho à direita. Em sequência, será chamada recursivamente a função de deleção para o nó com chave seguinte à do nó que será deletado. Em caso de desbalanceamento pós-remoção, o balanceamento se dá semelhantemente aos da inserção, chamando as devidas rotações. Vale ressaltar que há um caso em específico (tratado nas rotações) onde o peso da sub-árvore esquerda, após a remoção de um elemento, é zero. Para este caso, uma rotação simples resolverá. No entanto, o balanceamento deste nó deverá ser atualizado pela diferença entre o peso da sub-árvore esquerda pelo da direita.

```
/* caso onde o elemento possui sub-arvore a esquerda e a direita */

// ponteiro para o elemento mais a esquerda
AVL *tree = moreleft_AVL(node->right);

// apontamento para a chave do elemento mais a esquerda
//(A chave buscada é assegurada na main)
node->key = tree->key;

// chamada da remoção para a chave do elemento mais a esquerda
node->right = remove_AVL(node->right, tree->key, compare);
node->balance = weight_AVL(node);

if(node->balance > 1) node = right_rotation(node, &j);
```

### 5.1 Algoritmo auxiliar da remoção

Função citada no algoritmo da remoção.

```
AVL *moreleft_AVL(AVL *node){

        if(!node || !node->left) return node;
        else return moreleft_AVL(node->left);
}
```

## 6. Auxiliares da AVL

### 6.1 Cria AVL

Função responsável por reservar um espaço de memória dedicado a um ponteiro do tipo AVL, retornando-o caso a alocação seja bem sucedida.

```
AVL *create_AVL(void *element){

        // alocação de espaço de memória para o novo elemento
        AVL *node = (AVL *)malloc(sizeof(AVL));

        // verifica se a alocação foi bem sucedida
        if(node){

                // atribuição de valores aos seus devidos campos
                node->key = element;
                node->right = NULL;
                node->left = NULL;
                node->balance = 0;
        }

        return node;
}
```

### 6.2 Imprime AVL

Função responsável por imprimir todos os elementos de uma AVL, com o auxílio de uma função que imprimirá o conteúdo presente em suas chaves.

```
void print_AVL(AVL *node, void (*print_node) (void *), int level){

        int i;

        if(node){

                // chamada da impressão para a sub-arvore a direita
                print_AVL(node->right, print_node, level+1);

                //Tabulação dos elementos
                for(i = 0; i<level; i++) printf("\t");

                // impressão da chave de um elemento da AVL
                print_node(node->key);

                // chamada da impressão para a sub-arvore a esquerda
                print_AVL(node->left, print_node, level+1);
        }
}
```

### 6.3 Destrói AVL

Função responsável por  liberar espaço de memória antes dedicado aos nós da AVL, chamando a deleção da estrutura armazenada nas chaves.

```
void destroy_AVL(AVL *node, void (*destroy_node)(void *)){

        if(!node) return;

        // chamada da destroy_AVL para as sub-arvores esquerda e direita
        destroy_AVL(node->left, destroy_node);
        destroy_AVL(node->right, destroy_node);

        // chamada da destroy_node para a chave do elemento da AVL
        destroy_node(node->key);

        free(node);
        return;
}
```

## 7. Cliente (Main)

Afim de fazer o uso do TAD AVL, foi determinado uma estrutura denominada "Comanda" onde suas funções serão explicadas a seguir, onde a sua chave é "Ordem_de_chegada". Algumas das funções a seguir serão utilizadas como parâmetro das funções do TAD AVL, já que a mesma está implementada para tipos genéricos de dados.

### 7.1 Definição da estrutura Comanda

```
typedef struct{
    int Mesa;
    char *Nome;
    int Ordem_de_chegada;
    float Gasto;
}Comanda;
```

### 7.2 Coleta e inserção de dados na Comanda

Função responsável pela leitura e criação de um espaço do tipo Comanda.

```
Comanda *putInformation_comanda();

        int m, o;
        float g;
        char *n = (char *)malloc(50*sizeof(char));

        // nome
        printf("\n\t Digite os dados do cliente:");
        printf("\n\t Nome: ");
        getchar(); fgets(n, 50, stdin);

        // mesa
        printf("\n\t Num da Mesa: ");
        scanf("%d", &m);

        // ordem de chegada
        printf("\n\t Ordem de Chegada: ");
        scanf("%d", &o);

        // gasto
        printf("\n\t Gasto: ");
        scanf("%f", &g);

        // remove o "ENTER" do nome
        int len = strlen(n);
        n[--len] = 0;

        // retorno da Comanda com os dados inseridos
        return create_comanda(m, o, n, g);
```

### 7.3 Cria Comanda

```
Comanda *create_comanda(int m, int o, char *n, float g);
```

Função responsável por alocar espaço e retornar um ponteiro do tipo Comanda, inserindo caso a alocação seja bem sucedida, os seus dados.

```
// alocação de espaço de memória para o novo elemento
Comanda *new_comanda = (Comanda *)malloc(sizeof(Comanda));

// verifica se a alocação foi bem sucedida
if(new_comanda){

        // atribuição de valores aos seus devidos campos
        new_comanda->Mesa = m;
        new_comanda->Ordem_de_chegada = o;
        new_comanda->Nome = n;
        new_comanda->Gasto = g;
}

return new_comanda;
```

### 7.4 Compara Comandas

Função que recebe duas Comandas e compara o valor de suas chaves. Retorna -1 se as chaves forem iguais, 1 caso a chave 1 seja maior que a chave 2 ou 0 caso contrário.

```
int compare_comanda(void *cm1, void *cm2);

        // conversão das chaves genéricas para o tipo "Comanda"
        Comanda *Cm1 = (Comanda *)cm1;
        Comanda *Cm2 = (Comanda *)cm2;

        // comparacao das chaves
        if(Cm1->Ordem_de_chegada == Cm2->Ordem_de_chegada)
                return -1;
        else if(Cm1->Ordem_de_chegada > Cm2->Ordem_de_chegada)
                return 1;
        else return 0;
```

### 7.5 Imprime chave

Função responsável por imprimir uma chave dado um elemento do tipo Comanda. Nesta função, a realização do *cast* é necessária visto que a mesma recebe um ponteiro do tipo void *.

```
void print_order(void *cm);

        // Cast do ponteiro void para o tipo (Comanda *)
        Comanda *Cm = (Comanda *)cm;
        printf("%d\n", Cm->Ordem_de_chegada);
```

### 7.6 Imprime dados

Função responsável por imprimir os dados de uma Comanda.

```
void print_comanda(Comanda *cm);


        printf("\n");
        printf("\n\t Mesa: %d", cm->Mesa);
        printf("\n\t Nome do Cliente: %s", cm->Nome);
        printf("\n\t Ordem de Chegada: %d", cm->Ordem_de_chegada);
        printf("\n\t Gasto do cliente: R$ %.2f", cm->Gasto);
        printf("\n");
```

### 7.7 Destrói Comanda

Função responsável por liberar um espaço da memória antes dedicado ao campo Nome e a própria Comanda.

```
void destroy_comanda(void *cm){

        // conversão da chave genérica para o tipo "Comanda" e deleção
dos dados
        Comanda *Cm = (Comanda *)cm;
        free(Cm->Nome);
        free(Cm);
}
```

### 7.8 Imprime Menu

Função responsável por imprimir o Menu formatado.

```
void print_menu(){
        int i;

        printf("\n\n\t ");
        for (i = 0; i<40; i++) printf("-");
        printf("\n\t | 1 - Inserir um novo elemento na AVL\t|");
        printf("\n\t | 2 - Remover um elemento da AVL\t|");
        printf("\n\t | 3 - Buscar um elemento na AVL\t|");
        printf("\n\t | 4 - Imprimir a AVL\t\t\t|");
        printf("\n\t | 5 - Sair do programa\t\t\t|");
        printf("\n\t ");
        for (i = 0; i<40; i++) printf("-");
}
```