UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO - UFES
CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO - CEUNES
CIÊNCIA DA COMPUTAÇÃO

JOÃO PAULO SOUZA FERRETE
RAMON PEZZIN TON

RELATÓRIO

IMPLEMENTAÇÃO DE ÁRVORE AVL

SÃO MATEUS
23 DE MARÇO DE 2021

INTRODUÇÃO

Neste relatório temos como objetivo a implementação do tipo abstrato de dados (TAD) AVL, que é uma árvore binária de busca balanceada. Desta forma, serão apresentadas as funções feitas e as lógicas utilizadas para a implementação e funcionamento do programa, bem como os conceitos aplicados e como funcionam cada função. A codificação foi feita na linguagem C, com a compilação feita a partir de um arquivo Makefile.

IMPLEMENTAÇÕES

Arquivo ‘avl.h’:

Neste arquivo, temos a inclusão das bibliotecas a serem utilizadas no programa, a definição da estrutura AVL e o protótipo das funções que serão implementadas no arquivo ‘avl.c’.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

void * mallocSafe(size_t size);
AVL * encontraMenor(AVL * ptr);
AVL * criaArvore(void * inf);
AVL * procuraAVL(AVL * ptr, void * inf, int (*compara)(void*, void*));
void apagaAVL(AVL * ptr);
int alturaAVL (AVL * ptr);
void balanceiaAVL(AVL * avl);
AVL * SimplesDireita(AVL * avl);
AVL * DuplaDireita(AVL * avl);
AVL * SimplesEsquerda(AVL * avl);
AVL * DuplaEsquerda(AVL * avl);
AVL * inserirAVL(AVL * ptr, void * inf, int (*compara)(void*, void*));
AVL * removeAVL(AVL * ptr, void * inf, int (*compara)(void*, void*), void
(*copia)(void**, void**)); void imprimeArvore(AVL * ptr, void (*imprime)(void *));
```

A estrutura ‘AVL’ conta com dois ponteiros para AVL, o ‘`*left`’ que corresponde ao filho esquerdo da raiz e o ‘`*right`’ que corresponde ao filho direito da raiz. Também possui um ponteiro do tipo void (‘`*info`’) que é um ponteiro para tipo genérico de dados, que guardará o ponteiro para o elemento de informação daquela raiz. E, também, possui uma variável do tipo ‘`int`’ inteiro para armazenar o grau de balanceamento.

No restante do código, temos o protótipo das funções que serão implementadas no arquivo que utilizará essa biblioteca e serão explicadas a seguir.

Arquivo ‘avl.c’

Neste arquivo, temos a inclusão da biblioteca definida anteriormente (‘avl.h’) , e então temos as implementações das funções.

```c
#include "avl.h"

void * mallocSafe(size_t size){
    void * space = malloc(size);
    if(!space){
        printf("ERRO: SEM MEMÓRIA!\n");
        exit(1);
    }
    else return space;
}
```

A primeira função deste arquivo é a ‘mallocSafe’, que tem como objetivo impedir que o programa, com problemas de alocação de memória, continue executando. Ela recebe por parâmetro uma variável do tipo ‘`size_t`’ chamada size que representa quantidade de bytes a serem alocados. Durante sua execução, ela aloca em um ponteiro para tipo genérico um espaço de memória, do tamanho passado por parâmetro, a partir da função ‘`malloc`’, e então é feito a verificação de se o espaço foi alocado corretamente. Em caso negativo, é impresso uma mensagem de erro e então o programa pára de executar. Já, em caso positivo, um ponteiro para o espaço alocado é retornado.

```c
AVL * encontraMenor(AVL * ptr){
    if(!ptr) return NULL;
    if(ptr && !ptr->left) return ptr;
    return encontraMenor(ptr->left);
}
```

A função ‘encontraMenor’ retorna um ponteiro para raiz com o menor valor-chave, e recebe como parâmetro um ponteiro para uma árvore. Seu funcionamento depende de um dos princípios da AVL de que um elemento menor que a raiz será inserido sempre à sua esquerda, de forma que o menor elemento inserido será o elemento mais à esquerda. Portanto, é verificado se a árvore existe, em caso positivo é verificado se a árvore em questão é uma árvore válida e não possui filho a esquerda (ou seja, é a sub-árvore mais à esquerda), em caso positivo é retornado um ponteiro para essa raiz, caso contrário é chamada a função, de forma recursiva, passando por parâmetro a sub-árvore esquerda para verificação.

```c
AVL * criaArvore(void * inf){
    AVL * raiz = (AVL *) mallocSafe(sizeof(AVL));
    raiz->balance = 0;
    raiz->left = NULL;
    raiz->right = NULL;
    raiz->info = inf;
    return raiz;
}
```

A função ‘criaArvore’ retorna um ponteiro para uma nova árvore, já atribuindo a ela um elemento de informação, de tipo genérico, que foi passado por parâmetro. Primeiro se aloca um espaço de memória do tamanho da estrutura AVL e o ponteiro para esse espaço é guardado na variável ‘raiz’. Então, são inicializados os campos desta nova árvore, com o campo de balanço igual a 0 e os campos ‘raiz->left’ e ‘raiz->right’ apontando para ‘NULL’, já que a raiz em questão não possui filhos. Então é atribuído ao elemento de informação o ponteiro passado por parâmetro e se retorna o ponteiro para essa raiz.

```
AVL * procuraAVL(AVL * ptr, void * inf, int (*compara)(void*, void*)){
    if(!ptr) return NULL;
    if(compara(ptr->info, inf)==0) return ptr;
    else if(compara(ptr->info, inf)>0) return procuraAVL(ptr->right, inf,
compara);
    return procuraAVL(ptr->left, inf, compara);
}
```

Esta função é responsável pela busca de uma raiz na árvore. Ela retorna um ponteiro para o nó buscado, e recebe por parâmetro o ponteiro para a árvore, um elemento de informação de tipo genérico e uma função de comparação entre dois elementos de informação. Na execução, primeiro é verificado se a árvore existe, em caso negativo isso significa que o elemento também não existe na árvore, retornando ‘NULL’. Já, caso a árvore seja válida, verifica-se se o elemento buscado é igual ao elemento da raiz atual, em caso positivo eu retorno o ponteiro dessa árvore, mas em caso negativo é verificado se o elemento é maior que o elemento da árvore atual. Caso seja, eu chamo recursivamente a função passando por parâmetro a sub-árvore direita (já que o elemento buscado é maior), caso contrário faço a mesma chamada recursiva passando a sub-árvore esquerda.

```
void apagaAVL(AVL * ptr){
    if(!ptr) return;
    apagaAVL(ptr->left);
    apagaAVL(ptr->right);
    free(ptr->info);
    free(ptr);
}
```

A função ‘apagaAVL’ libera todo o espaço de memória utilizado por uma árvore AVL que é recebida por parâmetro. A primeira verificação é se a raiz está vazia. Em caso positivo se encerra a execução da função, já que não tem o que se liberar. Caso negativo, é chamada a função ‘apagaAVL’ para a sub-árvore esquerda e direita, e então é liberado o espaço utilizado pelo elemento de informação e pela raiz da árvore.

```
int alturaAVL (AVL * ptr){
    int r, l;
    if(!ptr) return 0;
    r=1+alturaAVL(ptr->right);
    l=1+alturaAVL(ptr->left);
    if(r>l) return r;
    return l;
}
```

Esta função é responsável por calcular a altura de uma árvore AVL. Ela recebe por parâmetro um ponteiro para uma raiz e, inicialmente, verifica se esse ponteiro é válido. Em caso negativo, é retornado 0, visto que ela não possui altura, caso contrário a variável '`int`' 'r' recebe 1 (a altura de uma nó sozinho é 1) mais a altura da subárvore direita, que é calculada com uma chamada recursiva da função, e a variável 'l' recebe a mesma chamada, mas com a sub-árvore esquerda. Ao fim, corresponte à altura da árvore o maior valor, então caso 'r' seja maior que 'l' 'r' é retornado, caso contrário 'l' é retornado.

```
void balanceiaAVL(AVL * avl){
    if(!avl) return;
    int a = alturaAVL(avl->left)- alturaAVL(avl->right);
    avl->balance=a;
}
```

A função 'balanceiaAVL' calcula e atribui o balanço de uma árvore passada por parâmetro. O balanceamento de uma árvore AVL é dado pela altura da subárvore esquerda subtraída da altura da subárvore direita. Desta forma, primeiramente é verificada se a árvore está vazia, em caso positivo a função se encerra. Mas caso a árvore seja válida, então a variável 'a' recebe o valor da altura da subárvore esquerda subtraída da subárvore direita, e então esse valor é atribuído ao campo 'balance' da raiz da árvore.

```
void imprimeArvore(AVL * ptr, void (*imprime)(void *)){
    if(!ptr) return;
    int i, nivel = alturaAVL(ptr);
    if(ptr){
        imprimeArvore(ptr->left, imprime);
        for(i=0; i<nivel; i++) printf("\t");
        imprime(ptr->info);
        imprimeArvore(ptr->right, imprime);
    }
}
```

A função de impressão de árvore recebe por parâmetro um ponteiro para a árvore e uma função responsável por imprimir a chave do elemento de informação de tipo genérico. A impressão é feita a partir do número de tabulações correspondentes ao nível. Primeiramente,

a variável ‘nivel’ recebe a altura da árvore. Então, chamamos a função para que, primeiramente, seja impresso a subárvore esquerda, para que a impressão seja simétrica, e então imprimimos, de acordo com o nível da árvore, a tabulação, para facilitar a visualização, então imprime-se a chave da raiz atual e se chama a função de impressão para a subárvore direita.

```
AVL * SimplesDireita(AVL * avl){
    AVL * a = avl->left;
    avl->left=a->right;
    a->right=avl;
    balanceiaAVL(avl);
    balanceiaAVL(a);
    return a;
}
```

A função ‘SimplesDireita’ é a função responsável por fazer a rotação simples à direita. Essa rotação ocorre quando uma raiz está desbalanceada para a esquerda, e seu filho esquerdo está desbalanceado, também, para a esquerda. A rotação é feita fazendo com que o filho à esquerda do nó desbalanceado seja o filho à direita do nó apontado por ‘a’(que era filho esquerdo do nó desbalanceado), e o filho à direita do nó ‘a’ passa a ser o nó desbalanceado. Após essa rotação, é garantido que o nó não está mais desbalanceado, já que o desbalanceamento era para esquerda e foi feita uma rotação para a direita. Portanto, é feito novamente o cálculo do balanceamento para a nova árvore e é retornada a raiz apontada por ‘a’ como nova raiz daquela subárvore.

```
AVL * DuplaDireita(AVL * avl){
    AVL * a=avl->left, * b = avl->left->right;
    avl->left=b->right;
    a->right=b->left;'
    b->left = a;
    b->right = avl;
    balanceiaAVL(avl);
    balanceiaAVL(a);
    balanceiaAVL(b);
    return b;
}
```

Essa função é responsável pela rotação dupla à direita, que é feita quando um nó está desbalanceado para a esquerda e o seu filho esquerdo está pesado para a direita. Inicialmente atribuímos o filho à esquerda da raiz para a variável ‘a’ e o filho à direita de ‘a’ para a variável ‘b’. Essencialmente, nesta rotação, fazemos com que ‘b’ seja a raiz da subárvore, fazendo com que ‘a’ seja seu filho à esquerda e a raiz desbalanceada seja seu filho à direita, sendo seus antigos filhos à esquerda passados para filhos à direita de ‘a’ e seus filhos à direita

passados como filhos à esquerda do nó desbalanceado. Após feita a rotação, são recalculados os balanços das subárvores.

```
AVL * SimplesEsquerda(AVL * avl){
    AVL *a = avl->right;
    avl->right=a->left;
    a->left = avl;
    balanceiaAVL(avl);
    balanceiaAVL(a);
    return a;
}
```

A função que faz a rotação simples à esquerda possui a mesma ideia de rotação que a função 'SimplesDireita', na qual o nó passado por parâmetro está desbalanceado para a direita e o seu filho à direita, também, está pesado para a direita. Desta forma, a variável 'a' recebe o filho à direita da raiz passada por parâmetro. O filho à esquerda da variável 'a' passa a ser filho à direita da raiz da árvore, e o novo filho à esquerda da variável passa a ser a árvore desbalanceada. É feito novo balanceamento e a variável 'a' é retornada como nova raiz da árvore.

```
AVL * DuplaEsquerda(AVL * avl){
    AVL * a = avl->right, * b=avl->right->left;
    a->left = b->right;
    avl->right = b->left;
    b->left=avl;
    b->right=a;
    balanceiaAVL(avl);
    balanceiaAVL(a);
    balanceiaAVL(b);
    return b;
}
```

A função 'DuplaEsquerda', também, tem funcionamento parecido ao da função 'DuplaDireita', mas, neste caso, a raiz está desbalanceada para a direita e o seu filho direito está pesado para a esquerda. Neste caso, a variável 'a' recebe o filho direito da raiz 'avl' e a variável 'b' recebe o filho esquerdo da variável 'a'. Desta forma, o re-apontamento é feito para que 'b' seja a nova raiz, seus filhos a direita passem a ser filhos a esquerda de 'a', e seus filhos a esquerda passem a ser filhos à direita de 'avl'. E então, 'avl' passa a ser filho à esquerda de 'b' e 'a' filho à direita. E então se refaz o balanceamento e retorna a variável 'b' como raiz.

FUNÇÃO DE INSERÇÃO

A função de inserção numa árvore será explicada em partes, para facilitar o entendimento do código.

```
AVL * inserirAVL(AVL * ptr, void * inf, int (*compara)(void*, void*)){

    if(!ptr) return criaArvore(inf);
```

A função ‘inserirAVL’ recebe por parâmetro um ponteiro para a árvore, onde o elemento deve ser inserido, o elemento de informação que deve ser inserido, e uma função de comparação entre elementos de informação, para que os elementos sejam inseridos no lugar correto. Ela retorna um ponteiro para a árvore com o elemento inserido, para caso de haver necessidade de rotações.

A função de comparação, tanto nesta como em outras que a utilizam, deve retornar um número menor que zero caso o primeiro parâmetro seja maior que o segundo, retornar zero, caso os elementos sejam iguais, e retornar um número maior que zero caso o segundo elemento seja maior que o primeiro.

A primeira verificação é se a árvore atual, passada por parâmetro, existe, caso não exista, o elemento a ser inserido não existe na árvore, então é criada uma nova raiz com o elemento de informação, a partir da função ‘criaArvore’, e então o ponteiro para essa nova raiz é retornado.

```
if(compara(ptr->info, inf)>0){
        ptr->right=inserirAVL(ptr->right, inf, compara);
        balanceiaAVL(ptr->right);
        balanceiaAVL(ptr);
        if(ptr->right->balance < 0 && ptr->balance < -1) ptr = SimplesEsquerda(ptr);
        else if (ptr->right->balance > 0 && ptr->balance < -1) ptr = DuplaEsquerda(ptr);
    }
```

O segundo caso é se a árvore existe, e o elemento a ser inserido é maior que o elemento da raiz atual. Neste caso o elemento deve ser inserido na subárvore direita àquele nó, portanto é chamada a função de inserção para a subárvore direita, e a nova árvore resultante deve ser retornada para o filho direito do nó atual. Após a inserção à direita, é feito novamente o cálculo do balanceamento e, se caso o nó atual ficar desbalanceado para a direita e o filho à direita desse nó estiver pesado à direita, deverá ser feita uma rotação simples à esquerda. Já, caso o nó atual esteja desbalanceado para a direita mas o seu filho direito esteja pesado para a esquerda, será necessária uma rotação dupla à esquerda.

```
else if (compara(ptr->info, inf)<0){
        ptr->left=inserirAVL(ptr->left, inf, compara);
        balanceiaAVL(ptr->left);
        balanceiaAVL(ptr);
        if(ptr->left->balance > 0 && ptr->balance > 1) ptr = SimplesDireita(ptr);
        else if (ptr->left->balance < 0 && ptr->balance > 1) ptr = DuplaDireita(ptr);
    }
```

O terceiro, e último, caso é se a árvore existe, e o elemento a ser inserido é menor que o elemento de informação da raiz atual. Neste caso, a inserção deve ocorrer na subárvore esquerda, que receberá o retorno da chamada da função também. Após a inserção, é feito o rebalanceamento e é verificada a necessidade de rotação.

Caso a raiz atual esteja desbalanceada para a esquerda e seu filho esquerdo esteja pesado para a esquerda, deve-se realizar a rotação simples à direita, mas caso a raiz esteja desbalanceada para a esquerda mas seu filho esquerdo esteja pesado para a direita, deverá ser feita a rotação dupla para a direita.

```
return ptr;
```

Ao fim das inserções, a árvore deverá ser retornada. Este retorno também funciona para o caso em que um elemento a ser inserido na árvore já está inserido. Neste último caso, nenhuma das opções anteriores será satisfeita, fazendo com que a árvore atual, que contém o elemento já inserido, seja retornada, sem a necessidade de alterações.

FUNÇÃO DE REMOÇÃO

De forma análoga à função anterior, essa função será explicada em partes, de modo a fazer-se mais didática.

```
AVL * removeAVL(AVL * ptr, void * inf, int (*compara)(void*, void*), void
(*copia)(void**, void**)){
      if(!ptr) return ptr;

    if(!procuraAVL(ptr, inf, compara)) return ptr;
```

A função ‘removeAVL’ retorna um ponteiro para a árvore com o elemento removido, e recebe por parâmetro o ponteiro para a árvore em que se encontra o elemento a ser removido, o elemento de informação, de tipo genérico, que contém a chave do elemento a ser removido, uma função para comparação, e uma função para fazer a troca dos valores de elementos de informação.

A primeira verificação a ser feita é se a árvore existe. Em caso negativo é retornado o próprio ponteiro vazio da árvore. Mas caso a árvore exista, é verificado se o elemento a ser removido existe na árvore, utilizando a função ‘procuraAVL’. Caso o elemento não esteja

inserido na árvore a função retorna o próprio ponteiro para a árvore, encerrando a execução da função. Mas caso o elemento exista, a função prossegue.

```
if(compara(ptr->info, inf)==0){
        if(!ptr->right){
            AVL * aux=ptr->left;
            free(ptr->info);
            free(ptr);
            balanceiaAVL(aux);
            return aux;
```

A primeira opção possível, caso o elemento esteja presente na árvore, é o elemento buscado ser a raiz atual, opção que podemos dividir em três casos.

O primeiro deles é caso o a raiz atual, que possui o elemento buscado, não possua filhos à direita. Se isso acontecer, é guardado na variável ‘aux’ um ponteiro para o filho esquerdo dessa raiz, e então liberamos todo o espaço utilizado por ela, refazemos o balanceamento, e retornando, depois, o ponteiro para sua filha à esquerda, que será recebido por quem chamou a função.

```
        }else if(!ptr->left){
            AVL * aux = ptr->right;
            free (ptr->info);
            free(ptr);
            balanceiaAVL(aux);
            return aux;
```

O segundo caso é se a raiz não possuir filhos à esquerda. Neste caso, da mesma forma do caso anterior, a variável ‘aux’ guarda o ponteiro para o filho direito do elemento, e então é liberado o espaço de memória utilizado por ele, refeito o balanceamento e retornando a variável ‘aux’.

```
        }else{
            AVL * aux = encontraMenor(ptr->right);
            copia(&ptr->info, &aux->info);
            ptr->right = removeAVL(ptr->right, inf, compara, copia);
            balanceiaAVL(ptr->left);
            balanceiaAVL(ptr);
            if(ptr->left && ptr->left->balance >= 0 && ptr->balance > 1)
ptr = SimplesDireita(ptr);
            else if (ptr->left && ptr->left->balance < 0 && ptr->balance
> 1) ptr = DuplaDireita(ptr);
        }
```

O último caso é se a raiz possuir dois filhos. Neste caso, a variável ‘aux’ irá guardar o menor elemento da subárvore direita, já que esse elemento continuará satisfazendo a condição de que todos os filhos à esquerda são menores que ele e, todos à direita, são maiores que ele. Então, é trocado os dados do elemento de informação entre esses dois elementos e chamamos, novamente, a função de remoção, mas passando por parâmetro a subárvore

direita, que, então, cairá em um dos dois casos anteriores.

Após a remoção do elemento, é refeito o balanceamento dos elementos. O balanceamento da subárvore direita é feita na chamada recursiva, então só é solicitado o rebalanceamento da subárvore esquerda e da raiz atual. Após isso, é analisada a necessidade de rebalanceamento. Como foi removido um elemento da subárvore direita, a unica possibilidade de desbalanceamento é para a esquerda. Então verificamos se a raiz está desbalanceada para a esquerda e se seu filho esquerdo está pesado para a esquerda, caso positivo então é feita a rotação simples à direita. E se a raiz estiver desbalanceada para a esquerda mas seu filho esquerdo estiver pesado para a direita é feita uma rotação dupla à direita.

Acabando os casos possíveis de se a raiz atual possui o elemento a ser removido, temos os casos de quando a raiz não possui o elemento a ser buscado. Neste contexto, existem duas opções: Caso o elemento a ser removido seja menor que o elemento atual e caso o elemento seja maior que a raiz atual.

```
}else if(compara(ptr->info, inf)<0) {
        ptr->left=removeAVL(ptr->left, inf, compara, copia);
        balanceiaAVL(ptr->left);
        balanceiaAVL(ptr);
        if(ptr->right && ptr->right->balance <= 0 && ptr->balance < -1)
ptr = SimplesEsquerda(ptr);
        else if (ptr->right && ptr->right->balance > 0 && ptr->balance <
-1) ptr = DuplaEsquerda(ptr);
    }
```

Neste primeiro caso, temos caso o elemento a ser removido seja menor que o elemento na raiz atual. Assim, o elemento a ser removido pertence à subárvore esquerda, então chamamos a função de remover passando por parâmetro a subárvore esquerda, que também irá retornar um ponteiro para a subárvore esquerda.

Após a remoção, atualizamos os balanços e verificamos se existe a necessidade de rotação. Como o elemento foi removido na subárvore esquerda, então a árvore só pode se desbalancear para a direita. Neste caso, verificamos se a árvore está desbalanceada para a direita e se a subárvore direita está pesada, também, para a direita. Em caso positivo é feita a rotação simples à esquerda. Já, caso a árvore esteja desbalanceada para a direita e a subárvore direita esteja pesada para a esquerda, então é feita a rotação dupla à esquerda.

```
  else if(compara(ptr->info, inf)>0) {
        ptr->right = removeAVL(ptr->right, inf, compara, copia);
        balanceiaAVL(ptr);
        balanceiaAVL(ptr->left);
        if(ptr->left && ptr->left->balance >= 0 && ptr->balance > 1)
ptr=SimplesDireita(ptr);
        else if (ptr->left && ptr->left->balance < 0 && ptr->balance > 1)
ptr=DuplaDireita(ptr);
    }

    return ptr;

      }
```

O último caso é se o elemento a ser removido é maior que o elemento atual. Neste caso, o procedimento é parecido ao efetuado no caso anterior, chamando-se a função de remover para a subárvore direita, balanceando os nós e então verificando a necessidade de rotação.

Neste caso, como a remoção foi feita à direita, então a árvore poderá ficar desbalanceada para a esquerda. Então, é verificado se a raiz atual está desbalanceada para a esquerda e se a subárvore esquerda está pesada para a esquerda, se caso positivo é feita a rotação simples à direita. Mas, caso a árvore esteja desbalanceada para a esquerda mas a subárvore esquerda esteja pesada para a direita, então é feita a rotação dupla à direita.

Ao fim da execução, é retornado o ponteiro para a árvore, já com o elemento removido.

Arquivo ‘main.c’

Neste arquivo, estão implementadas as funções ‘cliente’ que utilizarão a árvore AVL implementada nos arquivos anteriores, bem como a estrutura dos elementos de informação.

```
#include "avl.h"

typedef struct produto{
    long long codigo;
    char produto[50];
    int quant;
}info;
```

Primeiramente, temos a inclusão da biblioteca ‘avl.h’ para possibilitar o uso da estrutura de árvore. Então, temos a estrutura do elemento de informação, que são as informações de um produto em estoque, que são: o código de identificação do produto (que será utilizado como chave de busca e inserção na árvore), o nome do produto e a quantidade

deles em estoque.

```
info *criarInfo();
void printElemento(void * elem);
info *criarInfo2();
int compara(void * it1, void * it2);
void copia(void ** it1, void ** it2);
void imprimeInfo(void *inf);
```

Depois, temos os protótipos das funções implementadas que serão explicadas a seguir.

```
info *criarInfo(){

    info *elemento;
    elemento= (info*)mallocSafe(sizeof(info));
    printf("Digite o codigo: ");
    scanf("%lld",&elemento->codigo);
    printf("Digite o nome do produto: ");
    scanf("%s", elemento -> produto);
    printf("Digite a quantidade do estoque: ");
    scanf("%d",&elemento -> quant);

    return elemento;
}
```

A função ‘criarInfo’ é uma função que aloca o espaço de memória para um elemento de informação do tipo ‘info’, utilizando a função mallocSafe, e então pede para que o usuário insira os campos deste elemento. Depois de preencher os campos, é retornado um ponteiro para este elemento criado. Essa função é utilizada ao criar um elemento que será inserido na árvore.

```
void printElemento(void * elem){
    info* inf = (info *) elem;

    printf("\n---SOBRE O PRODUTO---\n");
    printf("Código do produto: %lld\n",inf->codigo);
    printf("Nome produto: %s\n", inf->produto);
    printf("Quantidade do Produto: %d\n", inf->quant);
}
```

A função ‘printElemento’ é responsável por imprimir todos os dados de um elemento. Ela é utilizada sempre antes de inserir ou apagar um elemento, e também após a busca de um elemento que exista na árvore. Ela recebe por parâmetro um ponteiro para um tipo genérico, que é utilizado para para fazer um casting para um elemento do tipo ‘info’ e então é feita a impressão de todos os dados.

```
int compara(void * it1, void * it2){

    info * item1 = (info *) it1;
    info * item2 = (info*) it2;

    if(!item1 && !item2) return 0;
    else if(!item1) return 1;
    else if(item1->codigo > item2->codigo) return -1;
    else if(item2->codigo > item1->codigo) return 1;
    else return 0;
}
```

A função ‘compara’ é a função de comparação para os elementos de informação passada por parâmetro para as funções de busca, inserção e remoção. Ela recebe dois ponteiros para elementos de tipo genérico de dados, e faz casting para elementos do tipo ‘info’. A função retorna 0 caso os elementos sejam iguais (ou ambos estejam vazios), -1 se o primeiro item for maior que o segundo (ou caso apenas o segundo ítem seja vazio) e 1 caso o segundo elemento seja maior (ou caso apenas o primeiro elemento esteja vazio).

```
void copia(void ** it1, void ** it2){

    info * item1 = (info *) *it1;
    info * item2 = (info*) *it2;
    info * aux = mallocSafe(sizeof(info));

    aux->codigo=item2->codigo;
    item2->codigo=item1->codigo;
    item1->codigo=aux->codigo;

    strcpy(aux->produto, item2->produto);
    strcpy(item2->produto, item1->produto);
    strcpy(item1->produto, aux->produto);

    aux->quant=item2->quant;
    item2->quant = item1->quant;
    item1->quant=aux->quant;

    *it1 = item1;
    *it2 = item2;

    free(aux);
}
```

A função ‘copia’ é passada por parâmetro para a função de remoção para trocar os elementos de informação de duas raízes de árvores. Ela recebe por parâmetro dois ponteiros para ponteiros de elementos de informação, que permite que esses elementos sejam alterados diretamente na posição de memória deles. Primeiramente fazemos casting dos elementos recebidos para o tipo ‘info’ e então alocamos o espaço necessário para uma variável auxiliar.

Com isso é feita a troca dos elementos de informação, atribuímos os novos valores às posições de memória original e então liberamos o espaço utilizado pela variável auxiliar.

```
info *criarInfo2(){
    info *elemento = (info*)mallocSafe(sizeof(info));
    printf("Digite o codigo: ");
    scanf("%lld",&elemento->codigo);

    return elemento;
}
```

A função ‘criaInfo2’ é responsável por criar um elemento de informação para fazer uma busca ou uma remoção. Ela aloca o espaço de memória para um elemento de informação, pede que o usuário insira o código do produto, que é a chave de busca na árvore, e retorna um ponteiro para o elemento criado.

```
void imprimeInfo(void *inf){
    info * elemento = (info *) inf;
    printf("%lld\n\n", elemento->codigo);
}
```

A função ‘imprimeInfo’ é a função passada por parâmetro para a função de imprimir árvore, ela recebe um elemento de informação de tipo genérico, faz o casting para o tipo ‘info’ e imprime a chave do elemento.

FUNÇÃO MAIN

A função main é a função principal do programa, utilizada como cliente para utilizar a estrutura de árvore implementada. Será mostrada em partes para facilitar o entendimento.

```
int main(){
    int l, p=4;
    info *elemento = NULL;
    AVL * ptr = NULL, * aux  = NULL;

    while( p != 0){
        printf("\n\n      ---- MENU ----\n");
        printf("Escolha uma das opções abaixo: \n");
        printf("1 - Inserir um elemento na arvore!\n");
        printf("2 - Remover um elemento da arvore!\n");
        printf("3 - Procurar um elemento na arvore!\n");
        printf("4 - Imprimir a arvore!\n");
        printf("0 - Sair do programa!\n\n");
        scanf("%d",&l);
```

Após a declaração de variáveis, temos um laço, que garante a impressão do menu e a solicitação, ao usuário, da escolha de uma das opções do menu.

```
switch (l)
        {
        case 1:
            printf("\nQual elemento deseja inserir: \n");
            elemento = criarInfo();
            printElemento(elemento);

            printf("\nAperte ENTER para continuar");
            getchar();
            getchar();

            ptr = inserirAVL(ptr,elemento, &compara);

            printf("\nElemento adicionado! \n");
            getchar();
            break;
```

Após o usuário inserir a opção desejada, entra-se num switch-case. Caso o usuário insira um número inválido nada acontece, até que seja inserido uma das opções do menu. Se o usuário inserir a opção de inserir um elemento, pergunta-se ao usuário qual elemento ele deseja inserir e então chama-se a função 'criarInfo', que cria um elemento de informação, e depois é impresso as informações inseridas. Então, é solicitado que o usuário aperte ENTER (que será lido pelo 'getchar') para continuar com a inserção, que é a chamada da função de inserção com o ponteiro para árvore criado anteriormente, após a inserção, espera-se mais um ENTER para que se volte ao menu .

```
case 2:
            printf("\nQual elemento deseja remover: \n");
            elemento=criarInfo2();
            aux=procuraAVL(ptr, elemento, &compara);

            if(aux) {
                printElemento(aux->info);
                printf("\nAperte ENTER para continuar");
                getchar();
                getchar();

            }else {
                printf("\n---ELEMENTO NÃO ENCONTRADO---\n");
                if(elemento) free(elemento);
                break;
            }

            ptr = removeAVL(ptr, elemento, &compara, &copia);

            if(elemento) free(elemento);
            printf("\nELEMENTO REMOVIDO!");
            printf("\nAperte ENTER para continuar");
            getchar();
            break;
```

Caso seja inserida a opção de remoção de elemento, é criado um elemento que apenas contém a chave de remoção, utilizando a função ‘criarInfo2’, e então verificamos se o elemento existe na árvore. Se a função de busca retornar nulo temos que o elemento não existe na árvore, então é impresso que o elemento não foi encontrado e voltamos para o menu. Mas caso o elemento exista, é impresso as informações deste elemento é solicitado que o usuário aperte ENTER para continuar. Então, é chamada a função de remoção e após isto é liberado o espaço da variável de elemento de informação para a busca e é impresso que o elemento foi removido, esperando o ENTER do usuário para voltar ao menu.

```
case 3:
            printf("\nQual elemento deseja procurar: \n");
            elemento=criarInfo2();
            aux=procuraAVL(ptr, elemento, &compara);

            if(aux) {
                printElemento(aux->info);
                printf("\nAperte ENTER para continuar");
                getchar();
                getchar();

            }else printf("\n---ELEMENTO NÃO ENCONTRADO---\n");
            if(elemento) free(elemento);
            break;
```

Caso a opção escolhida seja a de busca, é criado um elemento com a chave de busca e ele é passado por parâmetro para a função de buscar. Caso o elemento não esteja na árvore é impresso que o elemento não foi encontrado. Mas caso a função retorne um ponteiro válido, é impresso as informações desse elemento e é solicitado um ENTER para voltar ao menu. Então é liberado o espaço do elemento de busca e a função volta ao menu.

```
case 4:
            printf("\n---IMPRIMINDO ARVORE---\n");
            if(ptr)imprimeArvore(ptr, &imprimeInfo);

            else printf("ARVORE VAZIA\n");

            printf("\nAperte ENTER para continuar");
            getchar();
            getchar();
            break;
```

Já, caso seja inserida a opção de imprimir a árvore, é verificada se a árvore possui elementos. Em caso positivo, é chamada a função de impressão, mas caso contrário é impresso que a árvore está vazia. Então é solicitado um ENTER para voltar ao menu.

```
        case 0:
                    if(ptr) apagaAVL(ptr);
                    printf("\n----Programa Finalizado!!----\n");
                    p = 0;
                    break;
            }
        }
    }
```

Por último, caso seja inserida a opção de saída, é chamada a função ‘apagaAVL’ que libera todo o espaço utilizado pela árvore, é impresso que o programa foi finalizado e a variável ‘p’ recebe o valor 0, que faz com que seja interrompido o loop.

CONCLUSÃO

Neste relatório foi possível descrever todas as funções implementadas, aplicando os conceitos aprendidos em aula e exercitando os conhecimentos adquiridos anteriormente, o que ocasionou uma melhora nas práticas de programação e de entendimento de códigos.