Universidade Federal do Espírito Santo

Uarlley do Nascimento Amorim - Matricula: 2018205346
Uálaci dos Anjos Júnior - Matricula: 2017102015
29 de março de 2021

Estrutura de Dados II

Professora: Luciana Lee

Ciência/Engenharia da computação

Universidade Federal do Espírito Santo
DCEL

Relatório

Relatório do código TAD Árvore AVL

Aluno(a): Uarlley Nascimento amorim,
Uálaci dos Anjos Júnior

Professor(a) orientador(a): Luciana Lee

29 de março de 2021

Sumário

1 Introdução

A árvore AVL é um tipo mais sofisticado da árvore de busca binária. Sua diferenciação, se encontra no fato de que esta árvore possui sempre um balanceamento estável, dessa forma, ela permite um preenchimento mais adequado para operações de busca, aumentando seu desempenho em situações em que haja uma necessidade frequente de buscas. Nesse relatório iremos apresentar o código de implementação de um TAD (tipo abstrato de dados) de uma árvore AVL, assim como um exemplo de aplicação, com uma análise razoavelmente minuciosa sobre seu código.

2 Estruturas

Nessa seção serão apresentadas as estruturas criadas para a implementação tanto do TAD AVL quanto das funções do cliente.

2.1 Estrutura do cliente

```
typedef struct Info_{
    char* categoria;
    char* fabricante;
    long long numeroSerie;
    char* modelo;
    float preco;
    int peso;
}Info;
```

Para representar o cliente, resolvemos utilizar a árvore para a aplicação de estocagem de produtos de uma loja, chamada $Info$. Dessa forma. a estrutura de dados do cliente possui três campos do tipo ponteiro para caractere, cada um deles representando a *categoria*, $fabricante$ e o *modelo* de um produto. Há ainda um campo do tipo *long long* para representar o *número de série*, visto que é comum esse número ser consideravelmente grande. Por fim, um inteiro para representar o *peso*, este que é responsável por posicionar os elementos na árvore. Ele será mais explicado posteriormente no item4.2.

2.2 Estrutura do TAD AVL

```
typedef struct tnode{
    struct tnode *left;
    struct tnode *right;
    int height;
    void* info;
}AVLNode;
```

Essa é a estrutura utilizada em cada chave da árvore, chamada $AVLNODE$. Ela possui dois ponteiros, chamados $left$ e $right$, do seu próprio tipo, que representam os ponteiros para o filho a esquerda e o filho a direita, respectivamente. A estrutura também possui um campo inteiro chamado $height$, que representa a altura da chave correspondente, e um ponteiro do tipo vazio chamado $info$, que é o ponteiro que irá apontar para o elemento de informação implementado pelo cliente. É importante ressaltar que a escolha do tipo *void* para esse ponteiro é justamente para satisfazer uma relação comercial, na qual o programador do $TADAVL$ não tem conhecimento da estrutura do cliente, tampouco o cliente tem conhecimento da estrutura do $TADAVL$, e simplesmente a utiliza sem conhecimento profundo das linhas de código.

3 Funções de suporte

Nesta seção, serão apresentadas cada uma das funções auxiliares utilizadas para a implementação do *TAD árvore AVL*. Entre essas funções se encontram as rotações, funções de impressão, balanceamento e altura.

3.1 Direita AVL

```
AVLNode* rightAVL(AVLNode* root){
    if(root) return root->right;
    else return NULL;
}
```

Essa função tem como objetivo retornar o filho a direita de um nó. É retornado o filho a direita do nó caso o mesmo exista, do contrário, a função retorna vazio.

3.2 Esquerda AVL

```
AVLNode* leftAVL(AVLNode* root){
    if(root) return root->left;
    else return NULL;
}
```

Essa função tem como objetivo retornar o filho a esquerda de um nó. É retornado o filho a esquerda do nó caso o mesmo exista, do contrário, a função retorna vazio.

3.3 Raiz AVL

```
void* rootAVL(AVLNode* root){
  if(root) return root->info;
  else return NULL;
}
```

Esta função retorna o elemento de informação de um nó caso o mesmo exista. Do contrário, ela retorna vazio.

3.4 Criação de um nó

```
AVLNode* createAVLNode(void* info){
    AVLNode* newnode;
    if(info!=NULL){
        newnode = (AVLNode *)malloc(sizeof(AVLNode));
        if(newnode!=NULL){
            newnode->info = info;
            newnode->left = newnode->right = NULL;
            newnode->height = 1;
        }
    }
    return newnode;
}
```

Essa função é responsável por criar um nó da árvore. É recebido como parâmetro um ponteiro genérico do tipo *void*, o qual irá apontar para um elemento de informação, e então irá alocar dinamicamente um espaço para o novo nó, que por sua vez terá um campo que irá apontar para o conteúdo do ponteiro passado como

parâmetro para a função, assim como outros campos necessários para a implementação no formato de árvore, como por exemplo um ponteiro para apontar para o filho a direita e outro para apontar para o filho a esquerda. Por fim, essa função retorna o nó alocado. Se a alocação de memoria não for bem sucedida, é retornado vazio.

3.5 Rotação simples à Direita

```
void rotationLL(AVLNode **root){
    AVLNode* aux;
    aux = leftAVL(*root);
    (*root)->left = aux->right;
    aux->right = *root;

    (*root)->height = max(height((*root)->left), height((*root)->right))+1;
    aux->height = max(height(aux->left), height(aux->right))+1;
    (*root) = aux;
}
```

Uma rotação simples à direita, ou LL, ocorre quando há um desbalanceamento pela esquerda de uma determinada subárvore/árvore, ou seja, quando o filho à esquerda da subárvore esquerda de uma dada subárvore for o nó que esteja causando um desbalanceamento. Essa função recebe como parâmetro um ponteiro que aponta para o endereço da raiz da árvore/subárvore, e, então, realiza a rotação tornando o nó a esquerda da raiz o novo pivô da subárvore/árvore.Essa rotação pode ser visualizada na figura 1.

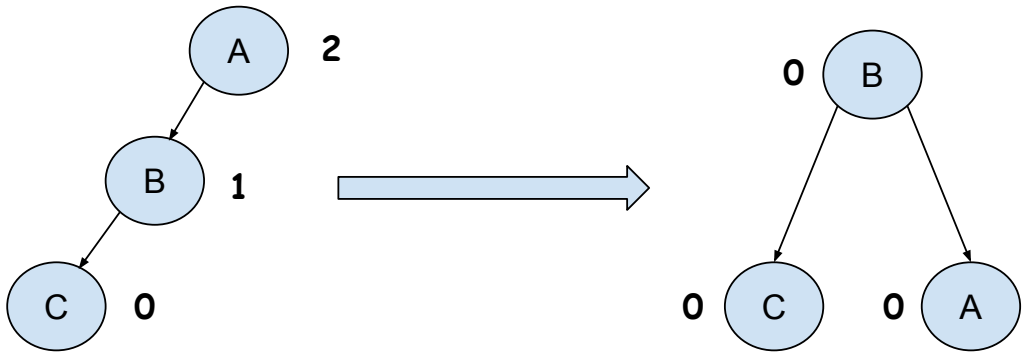

Figura 1: Aplicação de uma rotação simples à direita em uma subárvore desbalanceada a esquerda.

3.6 Rotação simples à Esquerda

```
void rotationRR(AVLNode **root){
    AVLNode* aux;
    aux = rightAVL(*root);
    (*root)->right = aux->left;
    aux->left = *root;

    (*root)->height = max(height((*root)->left), height((*root)->right))+1;
    aux->height = max(height(aux->left), height(aux->right))+1;
    (*root) = aux;
}
```

Uma rotação simples à esquerda, ou rotação RR, ocorre de forma análoga a anterior. Quando há um desbalanceamento pela direita de uma determinada subárvore/árvore, ou seja, quando o filho à direita da subárvore direita de uma dada subárvore/árvore, for o nó que esteja causando um desbalanceamento. Essa função recebe como parâmetro um ponteiro que aponta para o endereço da raiz de uma subárvore/árvore, e, então, realiza a rotação tornando o nó a direita da raiz o novo pivô. Essa rotação pode ser visualizada na figura 2.

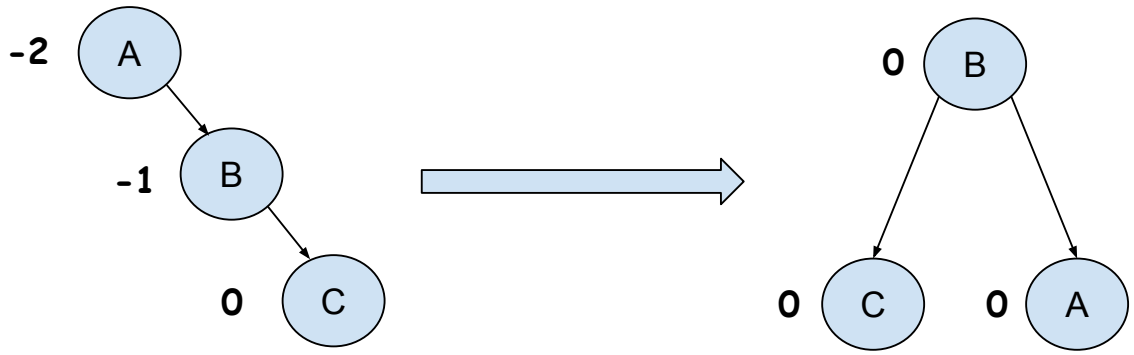

Figura 2: Aplicação de uma rotação simples à esquerda em uma subárvore desbalanceada a direita.

3.7 Rotação dupla à Esquerda

```
void rotationRL(AVLNode **root){
    rotationLL(&(*root)->right);
    rotationRR(root);
}
```

Uma rotação dupla à esquerda, ou rotação RL, ocorre quando há um desbalanceamento causado pelo filho à esquerda do filho a direita de uma dada subárvore/árvore. Essa função recebe como parâmetro um ponteiro que aponta para o endereço da raiz de uma subárvore/árvore, e, então, são aplicadas duas rotações simples . Primeiramente, é realizada uma rotação simples à direita no filho à direita da raiz da subárvore/árvore, e em seguida, é feita uma rotação simples à esquerda na raiz da subárvore/árvore.Essa rotação pode ser visualizada na figura 3.

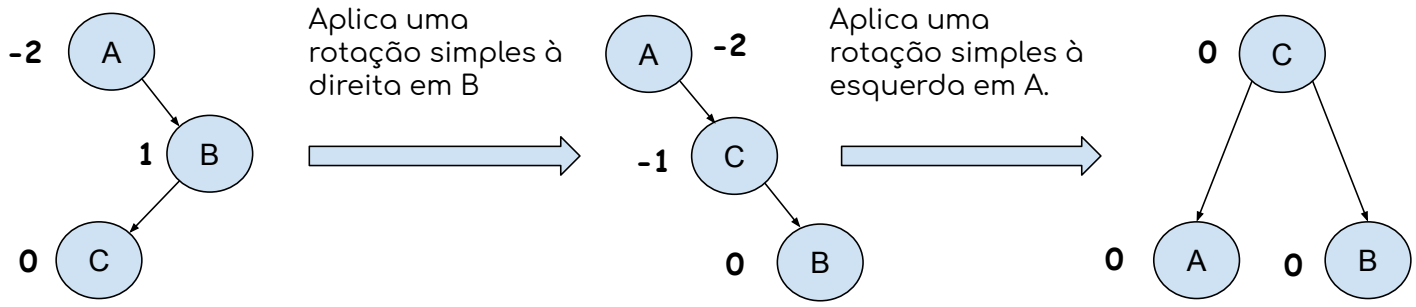

Figura 3: Rotação dupla a esquerda aplicada em uma subárvore desbalanceada.

3.8 Rotação dupla à Direita

```
void rotationLR(AVLNode **root){
    rotationRR(&(*root)->left);
    rotationLL(root);
}
```

A rotação dupla à direita, ou LR, ocorre quando há um desbalanceamento causado pelo filho a esquerda do filho a direita de uma subárvore/árvore. Essa função recebe como parâmetro um ponteiro que aponta para o endereço da raiz de uma subárvore/árvore, e, então, aplica duas rotações simples. Primeiramente, é realizada uma rotação simples à esquerda no filho à direita da raiz da subárvore/árvore. Em seguida, é aplicada uma rotação simples à direita na raiz da subárvore/árvore.Essa rotação pode ser visualizada na figura 4.

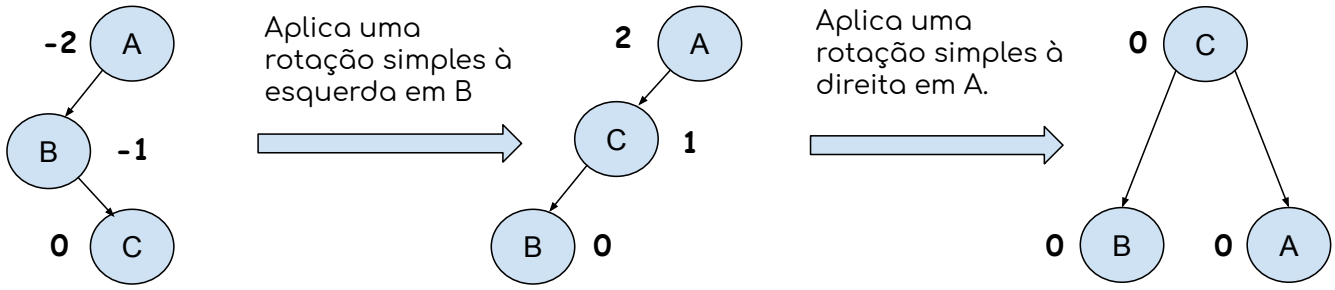

Figura 4: Rotação dupla à direita aplicada em uma árvore/subárvore desbalanceada.

3.9 Altura

```
int height(AVLNode* root){
    if(root == NULL) return 0;
    return root->height;
}
```

Essa função simplesmente retorna a altura que se encontra no campo altura de um nó. Se o nó não existir, a função retorna 0.

3.10 Altura AVL

```
int heightAVL(AVLNode* root){
    int lH = 0,rH = 0;
    if(root==NULL) return 0;
    lH = 1 + heightAVL(leftAVL(root));
    rH = 1 + heightAVL(rightAVL(root));
    if(rH>lH) return rH;
    else return lH;
}
```

Essa função calcula recursivamente a altura de uma árvore/subárvore e retorna sua altura. É recebido como parâmetro um nó da árvore. Se a chave não existir, o valor 0 é retornado. Em seguida, a função se chama recursivamente para a esquerda, e depois para a direita, somando em um o valor do retorno dessa recursão e atribuindo o resultado a uma variável. Por fim, é retornado o maior valor entre a altura da variável que recebeu a atribuição da recursão a esquerda e a que recebeu a atribuição da recursão a direita.

3.11 Recebe Balanço

```
int getBalance(AVLNode* root){
    if(root==NULL) return 0;
    return height(root->left) - height(root->right);
}
```

Essa função têm como objetivo calcular o balanço de um nó. É recebido como parâmetro um nó, e então é retornada a diferença da altura do seu filho a esquerda e seu filho a direita. Caso o nó não exista, a função retorna 0.

3.12 Imprime AVL

```
void printAVL(AVLNode* root, void (*imprime)(void* info)){
    if(root==NULL) return;
    printAVL(root->left,imprime);
    imprime(root->info);
    printAVL(root->right,imprime);

}
```

A função *imprimeAVL*, imprime os dados presentes em uma AVL de forma ordenada utilizando o caminho em-ordem. O caminho em-ordem consiste em, de forma recursiva, obter o elemento de informação do filho á esquerda da raiz, obter o elemento de informação da raiz, e, então, obter o elemento de informação do filho à direita da raiz da árvore/subárvore. Este processo faz com que a obtenção das informações saia de forma ordenada.

Para realizar a impressão, é necessário passar como parâmetros um ponteiro para a raiz da árvore/subárvore e uma função de impressão, essa que é responsável por imprimir os dados de um elemento de informação.

3.13 Balanceia AVL

```
void BalanceAVL(AVLNode** root){
    if(getBalance(*root)==2){
        if((*root)->left!=NULL && getBalance((*root)->left) <=-1) rotationLR(root);
        else rotationLL(root);
    }
    else if(getBalance(*root)==-2){
        if((*root)->right!=NULL && getBalance((*root)->right)>=1) rotationRL(root);
        else rotationRR(root);
    }
    return;
}
```

Essa função é responsável por determinar qual rotação deverá ser aplicada em um nó desbalanceado. Para identificar qual rotação deve ser aplicada, é utilizada a função *getBalance*(tópico 3.11), a qual determina o fator de balanceamento do nó desbalanceado.

Como mostrado na *figura*(5), caso o fator de balanceamento do nó seja igual a 2, é verificado o fator de balanceado de seu filho à esquerda, e dependendo do valor, são realizadas duas possíveis rotações (casos 2 e 4 da *figura*(5)). Analogamente, se o fator de balanceamento do nó for igual a -2, é analisado é fator de balanceamento de seu filho à direita, e dependendo do valor, duas possíveis rotações são realizadas (casos 1 e 3 da *figura*(5)).

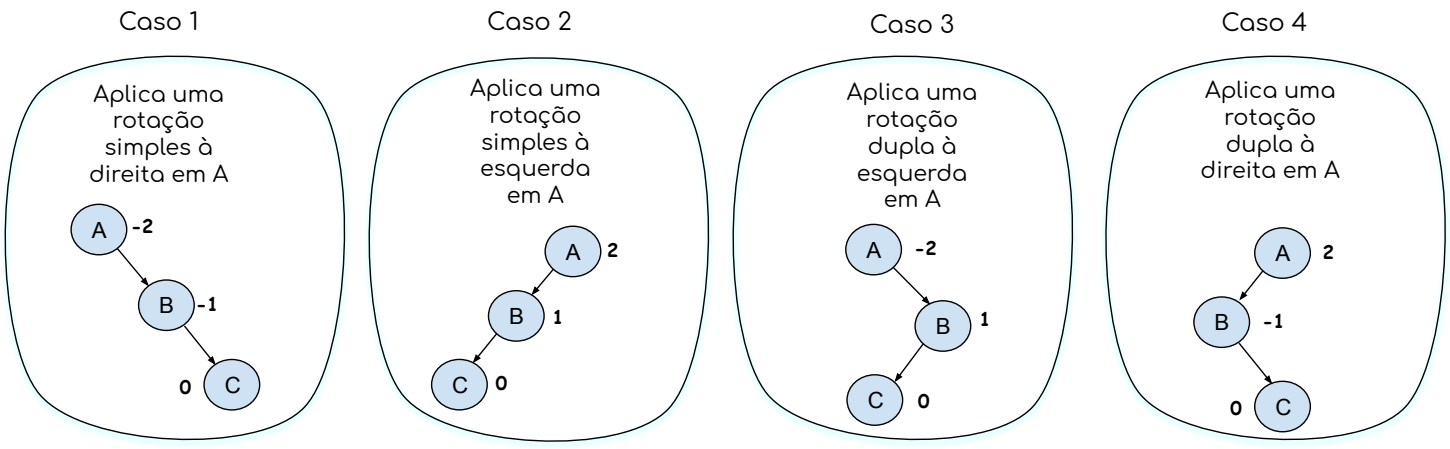

Figura 5: Todos os casos de desbalanceamento de uma AVL, e as rotações que devem ser aplicadas.

4 Funções Principais

Nessa seção, serão analisados os código das funções principais, que são as funções responsáveis por realizar as tarefas tradicionais da árvore, tais como inserção, remoção e busca.

4.1 Destrói AVL

```
void destroyAVL(AVLNode* root){
    if(root==NULL) return;
    destroyAVL(rightAVL(root));
    destroyAVL(leftAVL(root));
    free(root->info);
    free(root);
}
```

Essa função é responsável por excluir toda a árvore. É recebido como parâmetro um nó, que é a raiz da árvore. Caso a árvore não exista, a função retorna vazio. Então, ela se chama recursivamente para seu filho a esquerda, e para seu filho a direita em seguida. Ela limpa então o elemento de informação da chave e o ponteiro da chave.

4.2 Inserção AVL

```
int insertAVL(AVLNode **root,void* key, int (*comp)(void *a, void *b)){
    int s = 1;
    if(*root==NULL) {
        *root = createAVLNode(key);
        return 1;
    }
    else if(comp(key,(*root)->info) < 0) s = insertAVL(&(*root)->left,key,comp);
    else if (comp(key,(*root)->info) > 0)  s = insertAVL(&(*root)->right,key,comp);
    else return 0;
    if(*root && s){
        (*root)->height = 1 + max(height((*root)->left),height((*root)->right));
        if(abs(getBalance(*root))==2) BalanceAVL(root);
    }
    return s;
}
```

Essa é a função que realiza a inserção na árvore É possível separar essa função em duas partes, a inserção propriamente dita, e o rebalanceamento dos nós afetados pela inserção.
a função inicialmente atribui a variável *s* o valor 1, *s* representa o resultado da inserção. No primeiro caso, a função verifica se a nó da recursão atual não existe, se isso for verdade, cria-se um elemento de informação e o atribui à este nó retornando 1. Do contrário, o peso do elemento de informação é comparado com o elemento de informação do nó atual através da função de comparação. Se o resultado da comparação for menor que 1, ou seja, se a informação a ser inserida for menor do que a informação contida no nó atual, *s* recebe a chamada recursiva da função com o filho a esquerda do nó atual. Analogamente, se o resultado da comparação for maior que 1, *s* recebe a chamada recursiva da função com o filho a esquerda do nó atual. Caso nenhum dos casos anteriores ocorra, significa que o elemento a ser inserido já existe na árvore e portanto não faz sentido realizar a inserção do mesmo. Dessa forma, é retornado 0, o que significa uma falha de inserção.

Agora indo para a segunda parte da função. Após o procedimento anterior, a função atualiza a altura do nó atual da recursão, como sendo a soma de 1 mais a maior das alturas entre o filho da esquerda e o filho da direita da chave atual. Após isso, é calculado to balanço da chave atual através da função *getbalance* (tópico 3.11). Caso o valor absoluto do balanço do nó atual seja 2, isso significa que houve um desbalanceamento na

árvore. Dessa forma. a função realiza o balanço da árvore utilizando a função *BalanceAVL* (tópico 3.14), e por fim, retorna s.

4.3 Delete AVL

```
int deleteAVL(AVLNode **root, void* key, int (*comp)(void* a, void* b), void (*freeInfo)(void* info), void (*
    copy)(void* a,void* b)){
    int s = 1;
    if(*root==NULL) return 0;
    if(comp(key,(*root)->info) > 0)  s = deleteAVL(&(*root)->right,key,comp,freeInfo,copy);
    else if(comp(key,(*root)->info) < 0)   s = deleteAVL(&(*root)->left,key,comp,freeInfo,copy);
    else{

        AVLNode* temp;

        if((*root)->left==NULL){
            temp = (*root)->right;
            freeInfo((*root)->info);
            free((*root)->info);
            free(*root);
            *root = temp;
        }

        else if((*root)->right==NULL){
            temp = (*root)->left;
            freeInfo((*root)->info);
            free((*root)->info);
            free(*root);
            *root = temp;
        }

        else{
            temp = findLargerstElement((*root)->left);
            copy((*root)->info,temp->info);
            s = deleteAVL(&(*root)->left,(*root)->info,comp,freeInfo,copy);
        }
    }
    if(*root && s){
        (*root)->height = 1 + max(height((*root)->left), height((*root)->right));
        if(abs(getBalance(*root))==2) BalanceAVL(root);
    }
    return s;
}
```

Essa função pode ser separada em duas etapas, a remoção propriamente dita, e o rebalanceamento da árvore caso a remoção cause um desbalanceamento na mesma. Primeiramente, a função atribui 1 a uma variável s, que representa o resultado da remoção. Caso a raiz do nó atual não exista, a função retorna 0, indicando uma falha na remoção.
Então, é feita uma comparação entre a chave a ser removida e a chave presente no elemento de informação do nó atual através da função de comparação passada como parâmetro. Caso o resultado dessa comparação seja positivo, s recebe o resultado da chamada recursiva da função com o endereço do ponteiro do filho a direita do nó atual. Se o resultado da comparação for negativo, s recebe o resultado da chamada recursiva da função com o endereço do ponteiro do filho a esquerda do nó atual. Se nenhum dos caso anteriores forem verdade, significa que a chave a ser removida foi encontrada, e portanto, o processo de remoção é iniciado.

Caso o filho a esquerda do nó a ser removido não exista, a remoção precisa tratar os apontamentos referentes somente ao filho a direita e ao pai deste nó, portanto o processo de remoção se da como na figura (6).

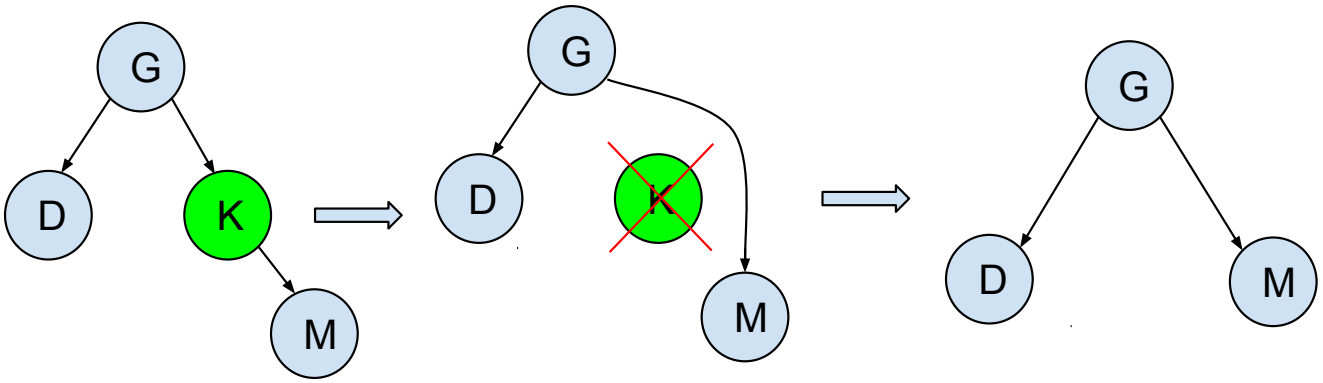

Figura 6: remoção do elemento "K"da árvore.

Caso o filho a direita do nó a ser removido não exista, a remoção precisa tratar os apontamentos referentes somente ao filho a esquerda e ao pai deste nó, portanto o processo de remoção se da como na figura (7).

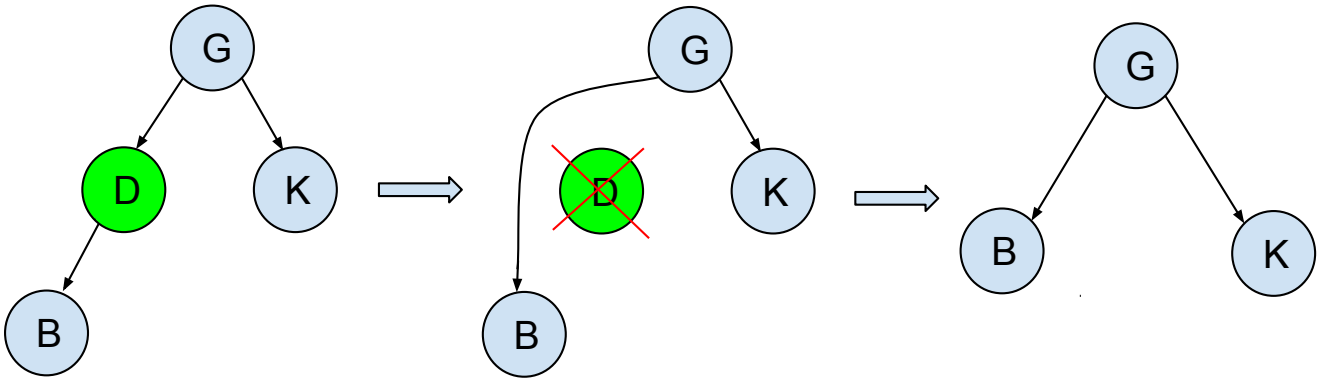

Figura 7: remoção do elemento "D"da árvore.

No entanto, se a chave a ser removida possuir dois filhos, a remoção é realizada conforme ilustrada na figura (8). É realizada uma busca do maior elemento da subárvore esquerda do nó, e, então, o conteúdo do maior elemento da subárvore esquerda é copiado no nó a ser removido, e em seguida, s recebe o resultado da chamada recursiva da função com o filho a esquerda do nó, porém, o elemento a ser removido torna-se o maior elemento da subárvore esquerda do nó. Isso fará com que no próximo passo da recursão, esse nó seja removido por meio de um dos casos anteriores.

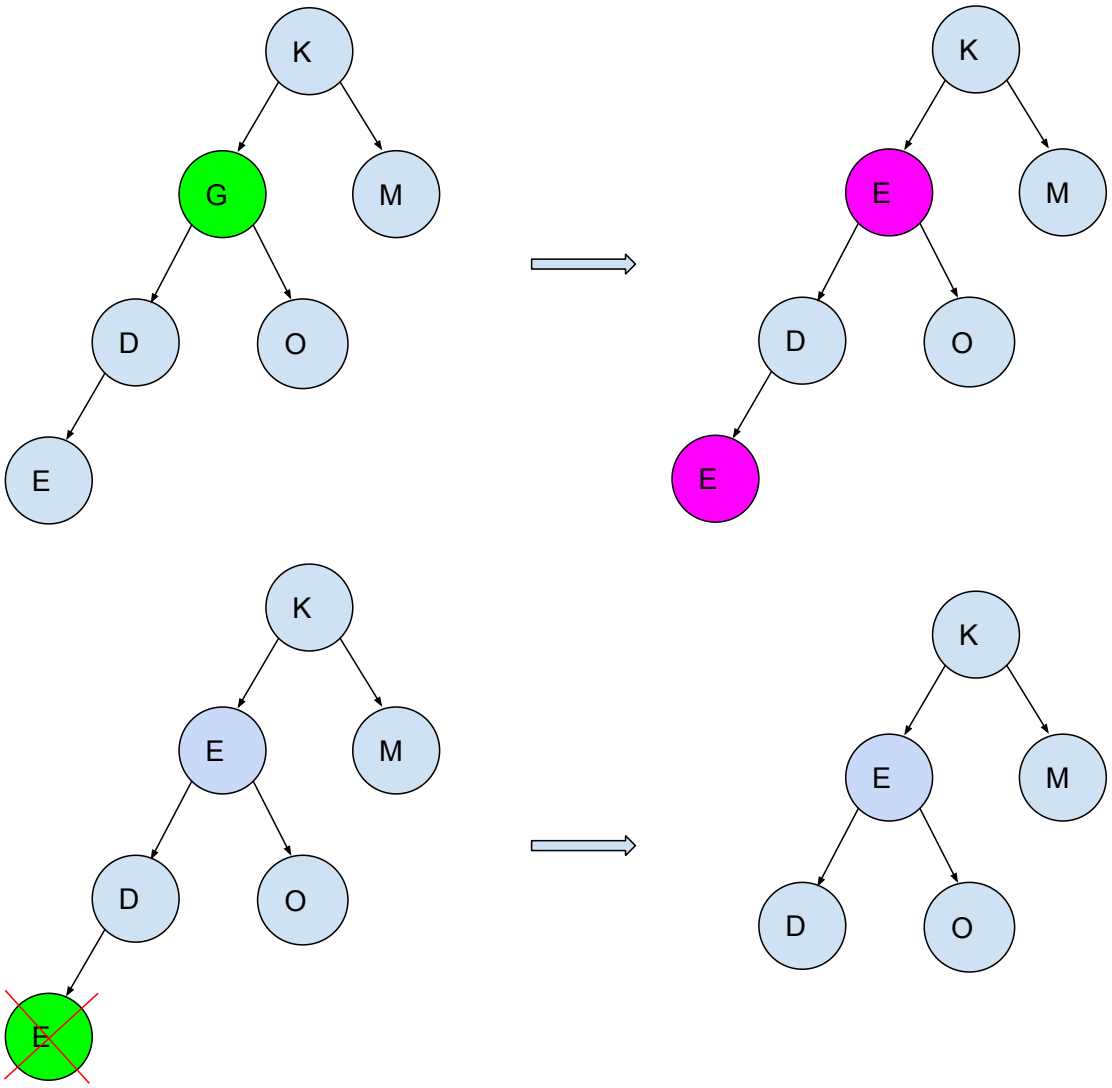

Figura 8: remoção do elemento "G"da árvore.

Após feita a remoção, se inicia o processo de balanceamento. A função verifica se a chave existe e se s existe, caso ambos forem verdade, a altura da chave atual é calculada, como sendo a soma de 1 mais o maior valor entre a altura do seu filho a esquerda e do seu filho a direita. Caso o balanço dessa chave for igual a 2, significa que houve um desbalanceamento na árvore, então a função $BalanceAVL$ (3.13) é chamada para realizar o balanceamento.

5 Funções do cliente, auxiliares para o TAD

Nessa seção, serão apresentados os códigos de funções auxiliares do cliente. Os códigos do cliente visam interpretar um cliente utilizando o TAD AVL. Dessa forma, as funções aqui apresentadas são funções auxiliares que o cliente deve interpretar para ajustar o TAD AVL para a sua aplicação.

5.1 Valor de palavra

```
1 int wordValue(char* word){
2     int soma=0,i;
```

```
    for(i=0;i<strlen(word);i++) soma+=(int)(word[i]);
    return soma;
}
```

Essa função calcula o valor em inteiro de uma palavra. A função recebe como parâmetro uma *string*, que é a palavra, e então é iniciado um loop indo do começo ao fim dessa palavra, e vai adicionando o valor em inteiro correspondente ao número da tabela ASCII do caractere no qual a iteração atual do loop está lendo e vai sempre somando esses valores dentro da variável *soma*. Ao fim do loop, a função retorna o valor final de *soma*.

5.2 Calcula peso

```
int calculaPeso(char* modelo, long numeroSerie){
    return wordValue(modelo) + numeroSerie%1000;
}
```

Essa função recebe como parâmetro uma *string* que representa o modelo de um produto, e o número de série desse produto. Como numa árvore AVL é necessário sempre ter um valor para utilizar como parâmetro de comparação, o cliente aqui utiliza o campo peso da $structInfo$ criada por ele como o parâmetro de comparação. Por questão de conveniência, para que os produtos apareçam de forma mais organizada na árvore, um cálculo que associa o valor do número de série e o modelo é efetuado para a determinação deste peso. A função retorna o Valor de palavra do modelo somado com o valor do resto da divisão do número de série por 1000, visto que a ideia é organizar as chaves dando uma importância maior ao modelo e em seguida para o número de série.

5.3 Cria informação

```
Info* createInfo(char* categoria, long long numeroSerie, char* modelo, float preco){
    Info* info = (Info*)malloc(sizeof(Info));
    if(info){
        info->categoria = (char*)malloc(53*sizeof(char));
        info->modelo = (char*)malloc(53*sizeof(char));
        info->numeroSerie = numeroSerie;
        strcpy(info->categoria,categoria);
        strcpy(info->modelo, modelo);
        info->preco = preco;
        info->peso = calculaPeso(modelo, numeroSerie);
    }

    return info;
}
```

Essa função simplesmente cria um elemento de informação que contêm as informações do produto, esse elemento será posteriormente alocado a um nó da árvore. A função recebe as strings *categoria* e *modelo*, um inteiro longo que representa o *numerodeserie*, e um $float$ que representa o *preço*. Inicialmente, a função aloca um espaço para um ponteiro do tipo $Info$, e, caso esse ponteiro tenha sido bem sucedido na alocação, são alocados cada um dos campos de *string*, possuindo também a validação da alocação. Após isso, são copiadas as informações das entradas em *string* para seus respectivos campos, o mesmo acontece com o preço e o numero de série, porém sem a necessidade de alocação visto que não são ponteiros. Por fim, a função retorna o elemento de informação $info$ devidamente preenchido.

5.4 Compara chave

```
int compKey(void* key , void* info){
    int* peso = (int*) key;
    Info* auxInfo = (Info*)info;
    if (*peso == auxInfo->peso) return 0;
    if(*peso == auxInfo->peso) return 1;
    return -1;
}
```

Essa função é responsável por comparar dois peso. Ela recebe como parâmetro dois ponteiros do tipo *void*. Um ponteiro é peso, que é do tipo *int*, e o outro um elemento de informação, que é do tipo $Info$. Então, a função copia o valor do ponteiro do peso para uma variável chamada *peso* fazendo o casting para o tipo $int*$, e o outro para $auxInfo$, fazendo o casting para o tipo $Info*$. É feita então uma comparação entre o *peso* e o campo que representa o peso do $auxInfo$, caso *peso* for maior, a função retorna 1, caso *peso* seja menor, o retorno é -1, e caso ambos tenham o mesmo valor, o retorno é 0.

5.5 Compara Informação

```
int compInfo(void* A, void* B){
    Info* a = (Info*)A;
    Info* b = (Info*)B;
    if (a->peso > b->peso) return 1;
    if(a->peso < b->peso) return -1;
    return 0;
}
```

Essa função é responsável por comparar dois elementos de informação. No caso do cliente no qual desenvolvemos, essa comparação é feita pelo peso do produto. Caso o peso do produto a for maior que o do produto b, a função retorna 1, do contrário, ela retorna -1, e caso forem iguais, o retorno é 0.

5.6 Copia

```
void copy(void* A, void* B){
    Info* a = (Info*)A;
    Info* b = (Info*)B;
    a->peso = b->peso;
    a->preco = b->preco;
    a->numeroSerie =  b->numeroSerie;
    strcpy(a->modelo, a->modelo);
    strcpy(a->categoria, a->categoria);
}
```

Essa função basicamente copia um elemento de informação para outro. Ela tem como parâmetro de entrada dois ponteiros do tipo vazio. Primeiramente, a função cria dois ponteiros, a e b, e faz o casting de ambos para o tipo $Info*$ e os aponta, cada um para o elemento de informação de um dos ponteiros de entrada, respectivamente. Após isso, cópia de cada campo do elemento de informação de a, que é do tipo $Info$, para o elemento de informação do ponteiro b de mesmo tipo, e finaliza.

5.7 Libera Dados

```
void liberaDados(void* informacao){
    /*Casting do parametro de tipo generico para o tipo informacao de um produto*/
    Info* info = (Info*)informacao;
    /*Liberacao da memoria alocada dinamicamente pelas variaveis de tipo char*/
    free(info->categoria);
    free(info->modelo);
    free(info->fabricante);
}
```

Essa função é responsável por liberar os espaços alocados pelos campos ponteiro de caractere de um elemento do tipo $Info$ alocado anteriormente. Ela simplesmente libera o espaço alocado no qual esses ponteiros apontam.

5.8 Imprime dados

```c
void imprimeDados(void* informacao){
    /*Casting do parametro de tipo generico para o tipo informacao de um produto*/
    Info* info = (Info*)informacao;
    printf("\n-------------------------\n");
    printf("Categoria: %s\n", info->categoria);
    printf("Fabricante: %s\n", info->fabricante);
    printf("Modelo: %s\n", info->modelo);
    printf("Numero de S rie: %lld \n",info->numeroSerie);
    printf("Pre o: R$ %.2f \n", info->preco);
    printf("-------------------------\n");
}
```

Essa função imprime todos os dados contidos no elemento de informação de um nó da árvore.

6 Resultados e conclusões

Como mencionado anteriormente, uma AVL tem como objetivo manter uma árvore binária de busca balanceada, para que a busca seja sempre eficiente. E como esperado, os resultados obtidos foram estes, uma estrutura de dados que possui complexidade $\log_2 n$ na busca, inserção e remoção de um elemento.

As maiores dificuldades da implementação foi a utilização de tipos genéricos de dados e a utilização da ferramenta $Makefile$, pois são conteúdos completamente novos e, de certa forma, difíceis de aplicar. Um dos pontos que foram considerados importantes na implementação do trabalho por completo, foi avaliar a liberação de memória de toda a estrutura. Para isso, foi utilizado o Valgrind, uma ferramenta que identifica vazamentos de memória durante e após o uso do programa. E, após realizar vários testes, foi comprovado que esta implementação não possui nenhum vazamento de memória. Além disso, foi implementado um sistema automatizado de inserção e remoção, para avaliar o balanceamento após cada uma dessas operações. E mesmo após a inserção de mais de 5 mil elementos, a AVL não apresentou nenhum desbalanceamento, comprovando a confiabilidade da implementação.

No geral, muito foi aprendido, não só como funciona uma AVL e suas utilidades, mas também as partes novas como os tipos genéricos de dados e a ferramenta $Makefile$.

7 Bibliografia

Aulas síncronas; conteúdo da disciplina de Estrutura de Dados I.

8 Apêndices

8.1 Apêndice 1: Makefile

```
1
2 SYSTEM  = x86-64_linux
3 LIBFORMAT = static_pic
4
5 OPTIONS = -O3
6 COMP = g++ -c
7
8
9 # ------------------------------------------------------------
10 # Compiler selection
11 # ------------------------------------------------------------
12 CCC = gcc
13 SRCDIR =
14 OBJDIR = obj
15
16 OBJ_PRJ =
17
```

```
CCFLAGS = $(CCOPT)

OBJ_LIB =

Projeto :  $(OBJ_PRJ) $(OBJ_LIB)
  $(CCC) $(CCFLAGS) -o programa.exe main.c Teste.c AVL.c $(OBJ_PRJ) $(OBJ_LIB)

clean:
  rm -f *.o
  rm -f *~
  rm Projeto
```

8.2 Apêndice 2: Main

```
//Programa principal para realizar os testes na AVL implementada.
#include "AVL.h"//Inclusao dos cabe alhos das funcoes e estruturas da AVL.
#include "Teste.h"
/*Este programa simula um inventario de produtos diversificados.
Tais produtos devem possuir:
    -categoria;
    -fabricante;
    -modelo;
    -n mero de serie;
-   -pre o em reais.
A posi  o dos produtos na AVL eh determinada pelo campo peso presente na estrutura.
Para a busca e remo  o, eh utilizado como elemento de busca o n mero de serie do produto.
Nao sao inseridos produtos com um mesmo numero de serie.*/

int main(){
    int op,ok; /*Op: variavel que define qual operacao sera realizada.
                 ok: variavel que define se a operacao teve sucesso ou nao.*/
    long long serie; //Variavel que recebe um numero de serie do produto.
    char mod[53]; // Variavel que recebe um numero de serie do produto.
    AVLNode* root = NULL; //Definicao da raiz da AVL;

    while(1){ //Loop para manter o menu de operacoes
        system("clear");
        printf("\n ----------------------------------------\n");
        printf("| 1 - Dar entrada em um produto     |\n");
        printf("| 2 - Buscar um produto             |\n");
        printf("| 3 - Remover um produto do estoque |\n");
        printf("| 4 - Imprimir lista de produtos    |\n");
        printf("| 5 - Sair                          |\n");
        printf(" ----------------------------------------\n");

        //Leitura da operacao a ser realizada
        if(!scanf("%d",&op)){
            printf("Nao foi possivel concluir a leitura!\n");
            exit(1);
        }
        getchar();

        //Inicia a operacao de insercao
        if(op==1) {
            system("clear");
            //Variaveis que receberao os dados do novo produto
            char categoria[53];
            float preco;
            char fabricante[53];

            //Dados do elemento novo elemento:
            printf("Preencha as informacoes do produto:\n");
            printf("\tCategoria: ");
            if(!fgets(categoria, 53, stdin)) {
                printf("Nao foi poss vel concluir a leitura!\n");
                exit(1);
            }
            categoria[strlen(categoria) - 1] = 0;

```

```
56              printf("\tFabricante: ");
57              if(!fgets(fabricante, 53, stdin)) {
58                  printf("Nao foi poss vel concluir a leitura!\n");
59                  exit(1);
60              }
61              fabricante[strlen(fabricante) - 1] = 0;
62
63              printf("\tModelo: ");
64              if(!fgets(mod, 53, stdin)){
65                  printf("Nao foi poss vel concluir a leitura!\n");
66                  exit(1);
67              }
68              mod[strlen(mod) - 1] = 0;
69
70              printf("\tNumero de Serie: ");
71              if(!scanf("%lld",&serie)){
72                  printf("Nao foi poss vel concluir a leitura!\n");
73                  exit(1);
74              }
75
76              printf("\tPreco: ");
77              if(!scanf("%f", &preco)){
78                  printf("Nao foi possivel concluir a leitura!\n");
79                  exit(1);
80              }
81              //Criacao do elemento de informacao de um produto para finalmente ser inserido na arvore
82              Info *info = createInfo(categoria, fabricante, serie, mod, preco);
83              if(insertAVL(&root, info, &compInfo)) printf("\nProduto inserido com sucesso!\n");
84              else printf("Nao foi possivel inserir!\nERRO: Numero de serie duplicado.\n");
85              printf("Pressione enter para continuar...");
86              getchar();
87              getchar();
88          }
89          //Inicia a operacao de busca
90          else if(op ==2){
91              system("clear");
92              AVLNode* ref;//Criacao do no temporario para receber o resultado da busca;
93              printf("\tDigite o modelo do produto: ");
94              if(!fgets(mod, 53, stdin)){
95                  printf("Nao foi possivel concluir a leitura!\n");
96                  exit(1);
97              }
98
99              mod[strlen(mod) - 1] = 0;
100
101             printf("\tDigite o numero de Serie do produto:");
102             if(!scanf("%lld",&serie)){
103                 printf("Nao foi possivel concluir a leitura!\n");
104                 exit(1);
105             }
106             //Realiza a busca na AVL
107             int peso = calculaPeso(mod, serie);
108             ref = searchAVL(root, &peso,&compKey);
109
110             //Se o elemento nao for encontrado e impresso a mensagem abaixo.
111             if(ref==NULL) printf("Produto nao encontrado!\n");
112
113             //Se o elemento for encontrado, seus dados sao impressos.
114             else imprimeDados((Info*)ref->info);
115             printf("Pressione enter para continuar...");
116             getchar();
117             getchar();
118         }
119         //Inicia a operacao de remocao
120         else if(op==3){
121             system("clear");
122             //Realiza a leitura do modelo do produto a ser removido
123             printf("\tDigite o modelo do produto: ");
124             if(!fgets(mod, 53, stdin)){
125                 printf("Nao foi possivel concluir a leitura!\n");
126                 exit(1);
```

```
127             }
128
129             mod[strlen(mod) - 1] = 0;
130
131             //Realiza a leitura do numero de serie do produto a ser removido
132             printf("\tDigite o n mero de S rie do produto:");
133             if(!scanf("%lld",&serie)){
134                 printf("Nao foi possivel concluir a leitura!\n");
135                 exit(1);
136             }
137
138             int peso = calculaPeso(mod, serie);
139
140             //Realiza e verifica se a remocao do elemento foi efetuada ou nao
141             if(deleteAVL(&root, &peso, &compKey , &liberaDados, &copy)) printf("Produto removido!\n");
142             else printf("Nao foi possivel remover o produto!\nERRO: Numero de serie nao encontrado.\n");
143             printf("Pressione enter para continuar...");
144             getchar();
145             getchar();
146         }
147         //Realiza a impressao da AVL utilizando o caminho em ordem
148         else if(op==4){
149             system("clear");
150             printAVL(root,&imprimeDados);
151             printf("Pressione enter para continuar...");
152             getchar();
153         }
154         /*Caso o usuario escolha sair do programa, a AVL eh destruida liberando toda a memoria alocada*/
155         else{
156           destroyAVL(root,&liberaDados);
157           return 0;
158         }
159     }
160 }
```