CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

ANTONIELLY BERGAMI RIBEIRO
WILLIAN PACHECO SILVA

RELATÓRIO: ÁRVORE AVL PARA TIPOS GENÉRICOS DE DADOS

SÃO MATEUS
2021

ANTONIELLY BERGAMI RIBEIRO
WILLIAN PACHECO SILVA

RELATÓRIO: ÁRVORE AVL PARA TIPOS GENÉRICOS DE DADOS

Relatório apresentado à Disciplina de Estrutura de dados II dos cursos de bacharelado em Ciência e Engenharia da Computação 2020/2 do Centro Universitário Norte do Espírito Santo, como requisito parcial para avaliação .
Orientador: Luciana Lee.

SÃO MATEUS
2021

RESUMO

Este relatório faz parte da avaliação parcial da disciplina de Estrutura de dados II dos cursos de Bacharelado em Ciência e Engenharia da Computação do Centro Universitário Norte do Espírito Santo.

Sua finalidade é apresentar os resultados das implementações das funções de uma árvore AVL para um tipo de dado genérico. Além das funções padrões da árvore: inserir, remover e buscar, também foram implementadas funções auxiliares para realizar as rotações. Além disso, foi criado um arquivo main em que foi definida uma estrutura para testar o código da biblioteca AVL. Nesse arquivo também foi feito um menu para auxiliar o usuário a cadastrar, remover e buscar elementos na árvore e imprimir os elementos dela por níveis. Também foi criado o arquivo .h para conter o cabeçalho das funções e o makefile para que seja possível realizar a compilação do programa por meio do make.

SUMÁRIO

1 INTRODUÇÃO

Uma árvore AVL nada mais é do que uma árvore de busca binária (estrutura de dados baseada em nós) balanceada. Uma árvore balanceada diminui o número de comparações no pior caso para busca de chaves com ocorrências iguais. Esse tipo de estrutura foi criada em 1962 por soviéticos e possui complexidade O(log n) para todas as suas operações (busca, inserção e remoção).

Este relatório apresenta os resultados encontrados após a implementação da árvore AVL com as funções de busca, inserção e remoção, além de outras funções como, por exemplo, a função de impressão da árvore e a função para liberar a memória alocada.

2 OBJETIVOS

O objetivo do trabalho é apresentar os conhecimentos adquiridos nas aulas de estruturas de dados sendo aplicados na implementação da árvore de acordo com o conteúdo aprendido e com pesquisas complementares demonstrando os métodos utilizados e a ordem de serviço para a perfeita execução da atividade proposta.

3 METODOLOGIA

Os resultados encontrados no procedimento da realização do estudo sobre uma árvore AVL foram feitos através de um programa desenvolvido pelos alunos na linguagem de programação C.

3.1 LINGUAGEM C

C é uma linguagem de programação compilada de propósito geral, estruturada, imperativa, procedural, padronizada pela Organização Internacional para Padronização (ISO), criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs para desenvolvimento do sistema operacional Unix (originalmente escrito em Assembly). C é uma das linguagens de programação mais populares e existem poucas arquiteturas para as quais não existem compiladores para C. C tem influenciado muitas outras linguagens de programação (por exemplo, a linguagem Java), mais notavelmente C++, que originalmente começou como uma extensão para C.

4 APRESENTAÇÃO DOS RESULTADOS E DISCUSSÕES

Para alcançarmos o objetivo proposto, que é a implementação da árvore AVL para um tipo de dado genérico, foram implementados alguns arquivos para melhor organização e execução do programa.

4.1 AVL.H

Um dos arquivos criados para a melhor organização e execução do programa foi o “AVL.h”. Esse arquivo permite que o cliente tenha uma ideia geral das funções que foram implementadas na biblioteca. Ele contém a declaração da estrutura, inclusão das bibliotecas, definição dos protótipos das funções, a descrição de cada uma delas e o seu retorno. Após sua criação, arquivo AVL.h passa a ser uma biblioteca que depois será incluída no arquivo AVL.c e no arquivo do cliente que for utilizar a biblioteca criada. No caso deste trabalho, o arquivo cliente é o “main.c”. As imagens abaixo demonstram como foi implementado o arquivo AVL.h

```c
// Este arquivo trata-se de uma biblioteca para uma árvore AVL com tipo de dados genérico.
#ifndef __AVL_H__
#define __AVL_H__

//Inclusão das bibliotecas necessárias.
#include <stdio.h>
#include <stdlib.h>

/*A estrutura abaixo é a estrutura da arvore AVL. Cada nó da árvore é declarado com esse tipo de dado.
Na estrutura da árvore AVL, cada nó tem os campos de chave, esq, dir e fb.
esq: campo do tipo struct avl * que armazena um ponteiro para o filho da esquerda do nó.
dir: campo do tipo struct avl * que armazena um ponteiro para o filho da direita do nó.
chave: campo do tipo void * que armazena um dado do tipo genérico.
fb: campo do tipo int que armazena o fator de balanceamento de cada nó.*/
typedef struct avl
{
    struct avl *esq;
    struct avl *dir;
    void *chave;
    int fb;
} AVL;

/* Função que cria um nó da árvore. Ela recebe como parâmetro a chave do tipo void * e retorna
o nó da árvore que possui como chave o elemento recebido.*/
AVL *criaNo(void *chave);

/*Função recursiva auxiliar utilizada para realizar as rotações na inserção. Ela recebe como parâmetro um
ponteiro para o nó da árvore que está desbalanceado e um ponteiro para uma variável que controla
se a altura da árvore foi alterada ou não. Com base nisso, ela realiza as devidas rotações (simples
ou dupla) para a direita e depois retorna um ponteiro para o novo nó raiz da árvore recebida.*/
AVL *rodaDireita(AVL *v, int *h);

/*Função recursiva auxiliar utilizada para realizar as rotações na inserção. Ela recebe como parâmetro um
ponteiro para o nó da árvore que está desbalanceado e um ponteiro para uma variável que controla
se a altura da árvore foi alterada ou não. Com base nisso, ela realiza as devidas rotações (simples
ou dupla) para a esquerda e depois retorna um ponteiro para o novo nó raiz da árvore recebida.*/
AVL *rodaEsquerda(AVL *v, int *h);
```

IMAGEM 1.1: Implementação do arquivo AVL.h

```c
/*Função recursiva auxiliar utilizada para realizar as rotações na remoção. Ela recebe como parâmetro um
ponteiro para o nó da árvore que está desbalanceado e um ponteiro para uma variável que controla
se a altura da árvore foi alterada ou não. Com base nisso, ela realiza as devidas rotações (simples
ou dupla) para a direita e depois retorna um ponteiro para o novo nó raiz da árvore recebida.*/
AVL *rodaDireitaRemover(AVL *v, int *h);

/*Função recursiva auxiliar utilizada para realizar as rotações na remoção. Ela recebe como parâmetro um
ponteiro para o nó da árvore que está desbalanceado e um ponteiro para uma variável que controla
se a altura da árvore foi alterada ou não. Com base nisso, ela realiza as devidas rotações (simples
ou dupla) para a esquerda e depois retorna um ponteiro para o novo nó raiz da árvore recebida.*/
AVL *rodaEsquerdaRemover(AVL *v, int *h);

/*Função recursiva para inserir dados na árvore. Ela recebe como parâmetro um ponteiro para o nó
raiz da árvore, um ponteiro para uma variável que controla se a altura da árvore foi alterada ou
não, um ponteiro para a chave a ser inserida na árvore, uma função de callback para realizar a
comparação entre as chaves para que seja possível inserir os dados na posição correta da árvore.
Além disso, ela recebe uma função de callback que realiza a liberação da memória alocada para o caso
da chave recebida já existir na árvore. Essa função retorna um ponteiro para a nova raiz da árvore.*/
AVL *insereAVL(AVL *v, int *h, void *chave, int (*compara)(void *, void *), void (*liberaChave)(void *));

/*Função recursiva para remover dados da árvore. Ela recebe como parâmetro um ponteiro para o nó
raiz da árvore, um ponteiro para uma variável que controla se a altura da árvore foi alterada ou
não, um ponteiro para a chave a ser removida da árvore, uma função de callback para realizar a
comparação entre as chaves para que seja possível encontrar o nó a ser removido e removê-lo da
árvore. Além disso, ela recebe uma função de callback que realiza a liberação da memória alocada
para o nó removido. Essa função retorna um ponteiro para a nova raiz da árvore.*/
AVL *remover(AVL *a, int *h, void *chave, int (*compara)(void *, void *), void (*liberachave)(void *));
```

IMAGEM 1.2: Implementação do arquivo AVL.h.

```c
/*Função recursiva utilizada para buscar uma chave na árvore. Essa função recebe como parâmetro
um ponteiro para o nó raiz da árvore, um ponteiro para a chave a ser buscada da árvore e uma função
de callback para realizar a comparação entre as chaves para que seja possível encontrar o nó. O retorno
dela é a chave buscada caso o elemento seja encontrado e NULL caso o elemento não esteja na árvore.*/
void *buscaChave(AVL *a, void *chave, int (*compara)(void *, void *));

/*Função recursiva que libera a memória alocada para todos os nós da árvore e das respectivas chaves.
Ela recebe como parâmetro um ponteiro para o nó raiz da árvore e uma função de callback que realiza
a liberação da memória alocada para cada nó. Essa função não possui retorno.*/
void liberaArvore(AVL *a, void (*liberaChave)(void *));

/*Função recursiva utilizada para imprimir os nós da árvore evidenciando o nível e o fator de
balanceamento de cada nó. Ela recebe como parâmetro um ponteiro para o nó raiz da árvore, uma
variável para controlar o nível da árvore em que o nó se encontra e uma função de callback que
realiza a impressão das chaves do tipo de dado do código do cliente. Essa função não possui retorno.*/
void imprimeArvore(AVL *a, int cont, void (*imprimeChave)(void *));
```

IMAGEM 1.3: Implementação do arquivo AVL.h.

No AVL.h é possível notar que foram incluídas as bibliotecas necessárias para que o programa funcione. Logo após isso, a estrutura utilizada para a árvore AVL foi implementada e descrita. Depois, os cabeçalhos das funções foram listados. Vale ressaltar que esse arquivo foi bem comentado e possui descrição do que será necessário para o funcionamento de cada função. Isso é importante pois são esses comentários que irão orientar o cliente sobre como utilizar a biblioteca da forma correta.

4.1.1 Estrutura do nó

Um ponto importante que merece destaque nesse arquivo é a implementação da estrutura do nó da árvore. Essa estrutura é a seguinte:

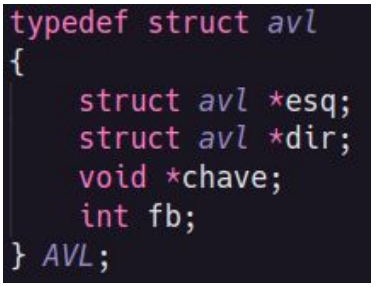


IMAGEM 2: Declaração da estrutura do nó.

Como é possível perceber, a estrutura utilizada como base para a árvore AVL contém alguns campos para que a árvore funcione bem.

Os dois primeiros campos da estrutura são campos do tipo ponteiro para ela mesma. Isso quer dizer que esses campos são ponteiros e que irão apontar para outras estruturas do tipo AVL. Isso é importante pois eles são os ponteiros que irão apontar para as subárvores esquerdas e direitas de cada nó.

O terceiro campo é o campo que vai armazenar a chave do nó. Como a implementação da árvore AVL foi para um tipo de dados genérico, a chave deve ser do tipo void * para que o nó possa receber como chave um ponteiro para qualquer tipo de dado.

Por fim, o último campo é um campo do tipo inteiro que armazena o fator de balanceamento de cada nó. Esse campo é muito importante pois é ele que será utilizado ao verificar o balanceamento da árvore.

4.2 MAIN.C

Outro arquivo implementado foi o main.c, ou também chamado de arquivo de cliente. Nesse arquivo estão as funções de callback, o menu de opções e a função principal do programa. Além disso, também foi implementada uma estrutura para que todas as chaves armazenadas na árvore sejam desse tipo.

4.2.1 Estrutura da mesa

A estrutura criada no arquivo main.c foi denominada mesa. Ela foi implementada da seguinte forma:

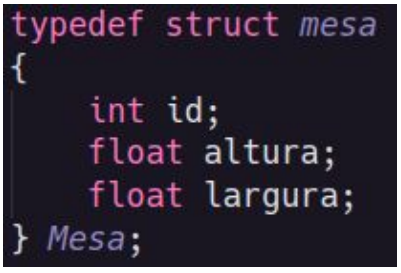


IMAGEM 3: Declaração da estrutura mesa.

A estrutura acima foi criada no arquivo main.c de forma que todo o nó inserido na árvore possuísse a chave do tipo Mesa.

Para a criação dessa estrutura foi necessário abstrair 3 atributos que definem uma mesa, e chegou-se ao consenso de que os três parâmetros que melhor definiriam uma mesa seriam o id, a altura e a largura dela. Como o id é uma chave identificadora, foi considerado que ele seria do tipo int. Já a altura e a largura podem ter casas decimais, por isso, são do tipo float.

4.2.2 Funções de Callback

No arquivo main.c também foram implementadas as funções de callback. Essas funções são passadas por parâmetro nas chamadas de algumas funções do arquivo AVL.c.

4.2.2.1 Imprime Chave

A função “imprimeChave” é a função de callback utilizada para imprimir as chaves através do ID da mesa. Ela recebe um ponteiro do tipo void * e não retorna nada. Nela é feito um casting do tipo void * para o tipo Mesa * e, caso a chave esteja vazia, a função imprime uma mensagem de erro. Caso contrário, ela imprime o id da chave.

Essa função é importante pois ela é passada como parâmetro para a função do arquivo AVL.c que realiza a impressão da árvore.

```
//Função de callback que imprime o id da mesa
void imprimeChave(void *chave)
{
    //Realizando o casting de void * para Mesa *
    Mesa *m = (Mesa *)chave;

    if (m) //Caso a mesa não seja nula
        printf("[%d]", m->id);
    else //Caso a mesa seja nula
        printf("Chave nao encontrada.");
}
```

IMAGEM 4: Implementação da função imprimeChave

4.2.2.2 Libera Chave

A função “liberaChave” é a função de callback utilizada para liberar a memória alocada para uma chave do tipo Mesa. Para isso, é necessário realizar um casting de void * para Mesa * e depois executar o comando free( ). Essa função recebe como parâmetro uma chave do tipo void * e não retorna nada. Além disso, ela é passada como parâmetro para as funções de inserção, remoção e também para a “liberaArvore”. A implementação dessa função está na imagem abaixo:

```c
//Função de callback que libera a memória alocada para a mesa
void liberaChave(void *chave)
{
    //Realizando o casting de void * para Mesa *
    Mesa *m = (Mesa *)chave;
    //Liberando a memória alocada
    free(m);
}
```

IMAGEM 5: Implementação da função liberaChave.

4.2.2.3 Compara

A função “compara” é a função de callback usada para comparar duas mesas através do id de cada uma delas.

Inicialmente, essa função recebe dois ponteiros do tipo void *. Caso um dos ponteiros seja NULL, não é possível realizar a comparação, então é apresentada uma mensagem de erro. Caso contrário, a função irá realizar o casting de void * para Mesa * e comparar os ids. Com base nessa comparação, ela irá retornar 1 caso o id da primeira mesa seja maior, -1 caso o id da segunda mesa seja maior e 0 caso os ids sejam iguais. A partir desses retornos, a função inserir (do arquivo AVL.c), por exemplo, pode percorrer os nós da árvore e inserir o novo nó no lugar certo. As funções de remoção e de busca também utilizam essa função de callback para realizar as comparações. A implementação dessa função está na imagem abaixo:

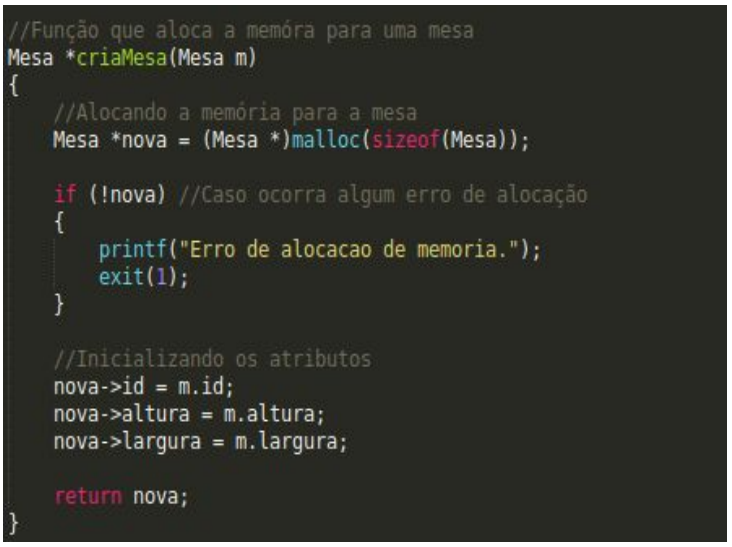
```c
//Função de callback utilizada para comparar duas mesas
int compara(void *mesa1, void *mesa2)
{
    if (mesa1 && mesa2) //Caso nenhuma das duas mesas seja nula
    {
        //Realizando o casting de void * para Mesa *
        Mesa *m1 = (Mesa *)mesa1;
        Mesa *m2 = (Mesa *)mesa2;

        if (m1->id > m2->id) //Caso a chave da primeira mesa seja maior do que a da segunda
            return 1;
        else if (m1->id < m2->id) //Caso a chave da primeira mesa seja menor do que a da segunda
            return -1;
        else //Caso a chave da primeira mesa seja igual a da segunda
            return 0;
    }
    else //Caso alguma mesa seja nula, a comparação é inválida
        printf("\nComparacao invalida!\n");

    return 0;
}
```

IMAGEM 6: Implementação da função compara.

4.2.3 Cria mesa

Outra função implementada é a “criaMesa”. Essa função é usada para alocar memória para uma nova “Mesa”. Ela irá imprimir uma mensagem de erro caso

ocorra algum erro na alocação, mas caso ocorra tudo certo, ela irá inicializar os atributos e retornar o novo nó criado com os atributos inicializados. A imagem a seguir demonstra como foi implementada essa função.



```c
//Função que aloca a memória para uma mesa
Mesa *criaMesa(Mesa m)
{
    //Alocando a memória para a mesa
    Mesa *nova = (Mesa *)malloc(sizeof(Mesa));

    if (!nova) //Caso ocorra algum erro de alocação
    {
        printf("Erro de alocacao de memoria.");
        exit(1);
    }

    //Inicializando os atributos
    nova->id = m.id;
    nova->altura = m.altura;
    nova->largura = m.largura;

    return nova;
}
```

IMAGEM 7: Implementação da função para alocação de memória de uma mesa.

4.2.4 Menu

Outra função muito importante que está presente no arquivo “main.c” é a função de menu. Essa função recebe como parâmetro um ponteiro para o nó raiz da árvore e uma variável que indica qual opção foi escolhida pelo usuário e retorna um ponteiro para a nova raiz da árvore para o caso da raiz ser alterada por alguma função.

As operações a serem realizadas na árvore são escolhidas através desse menu que, no programa feito, ficou na seguinte ordem:

1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Árvore
5 - Sair

Durante a execução do programa, o usuário receberá as opções na tela e irá escolher a ação desejada. A opção escolhida será passada para a função menu e ela tratará cada caso.

Essa função possui um switch case que irá chamar a função referente a ação desejada. Caso o usuário escolha uma opção que não existe no menu, é exibida uma mensagem de erro e ocorre um looping das opções até que ele escolha uma opção válida. A imagem abaixo mostra o que acontece quando o usuário escolhe uma opção que não estava presente no menu.

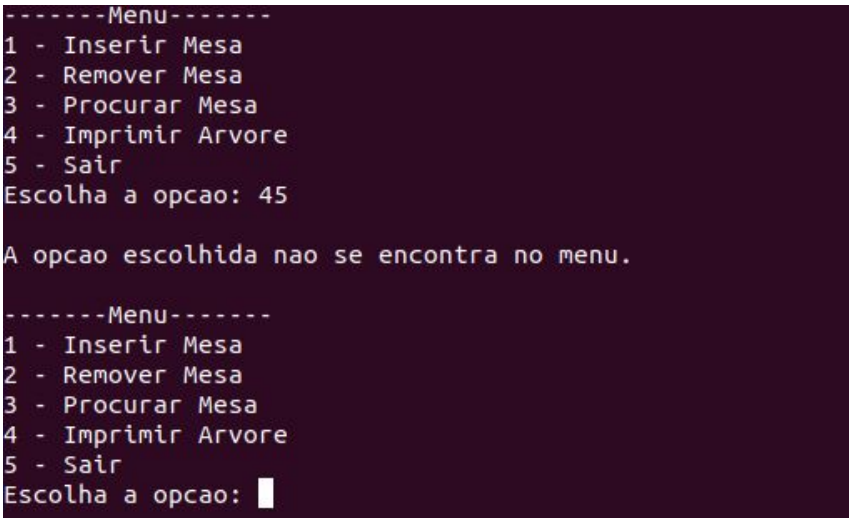


IMAGEM 8: Saída do menu caso a opção escolhida seja inválida.

O caso da opção escolhida não estar disponível no menu foi tratado no caso default do switch case. O código desse caso está na imagem abaixo.

```
default: //Caso a opção escolhida não esteja no menu
    printf("\nA opcao escolhida nao se encontra no menu.\n");
    break;
```

IMAGEM 9: Trecho de código que trata o caso da opção escolhida não está no menu.

Além de chamar as funções, quando a opção de sair é selecionada no menu, toda a memória alocada durante o programa é liberada e o programa é encerrado como mostra a figura abaixo.

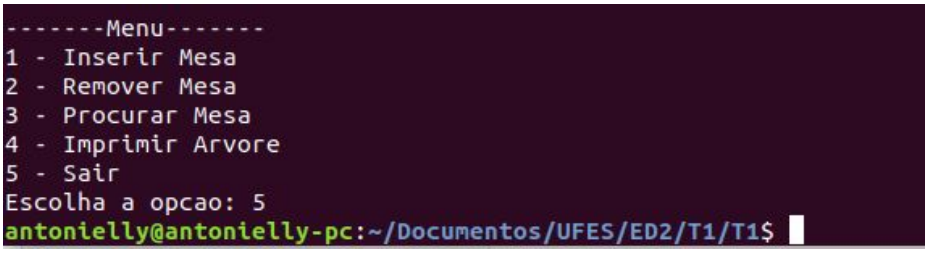


IMAGEM 10: Saída do programa caso a opção escolhida seja sair.

4.2.4.1 Inserir

Quando for selecionada a opção de inserir uma nova mesa no menu, inicialmente será feita a leitura dos dados da nova mesa como é demonstrado na imagem abaixo.

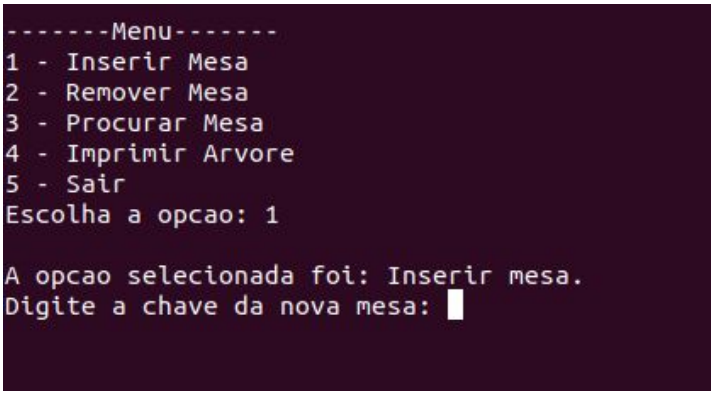


IMAGEM 11: Saída do programa caso a opção seja inserir.

Depois disso, será feita uma busca na árvore para verificar se a chave a ser inserida já foi cadastrada anteriormente na árvore. Isso é feito para garantir que não seja possível inserir chaves repetidas na árvore. Caso seja encontrada outra mesa com a mesma chave, uma mensagem de erro é mostrada na tela. Caso não haja nenhum elemento com a mesma chave, a função “insereAVL” é chamada para realizar a inserção da nova mesa na árvore. Como essa função de inserção recebe como um dos parâmetros um ponteiro para a chave a ser inserida na árvore, e essa chave é a própria mesa, foi utilizada a função “criaMesa” para alocar a memória para uma estrutura desse tipo, já inicializando os valores e depois disso o nó alocado já é passado para a função de inserção como sendo a chave a ser inserida na árvore. Depois de realizar a inserção, os dados da mesa inserida são impressos na tela. A implementação desse caso está na imagem abaixo:

```c
case 1: //Inserir elemento
    //Leitura dos dados
    printf("\nA opcao selecionada foi: Inserir mesa.\n\n");
    printf("Digite a chave da nova mesa: ");
    scanf("%d", &m.id);
    printf("Digite a altura da nova mesa: ");
    scanf("%f", &m.altura);
    printf("Digite a largura da nova mesa: ");
    scanf("%f", &m.largura);

    //Realizando a busca da chave a ser inserida
    mesa = buscaChave(a, &m, compara);
    mesaBuscada = (Mesa *)mesa;

    if (!mesaBuscada) //Caso a chave ainda não exista na árvore
    {
        //Realiza a inserção do elemento na árvore
        a = insereAVL(a, &h, criaMesa(m), compara, liberaChave);

        //Imprime o elemento inserido
        printf("\nA seguinte mesa foi cadastrada:\n");
        printf("\nChave: %d\n", m.id);
        printf("Altura: %.2f\n", m.altura);
        printf("Largura: %.2f\n", m.largura);
    }
    else //Caso a chave já exista na árvore
        printf("\nNao foi possivel cadastrar. A mesa com chave %d ja foi cadastrada.\n", m.id);
    break;
```

IMAGEM 12: Implementação do código para a chamada da opção inserir.

4.2.4.2 Remover

Caso o usuário queira remover algum elemento da árvore, será necessário que ele informe o elemento a ser removido pelo número da chave desse elemento, como mostrado na figura abaixo.

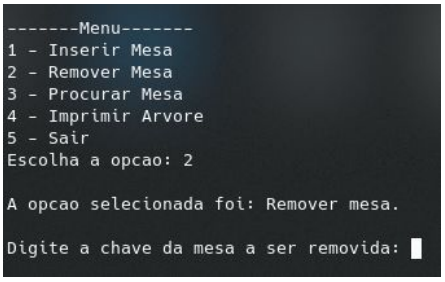


IMAGEM 13: Saída do programa caso a opção escolhida seja remover.

Depois de realizar a leitura da chave, é feita uma busca na árvore para verificar se a chave está na árvore ou não. Caso ela não esteja na árvore, uma mensagem de erro é mostrada para o usuário. Caso a chave seja encontrada, a função de remoção é chamada para remover o nó que possui aquela chave. Um detalhe importante sobre esse caso é que a função de remoção recebe um ponteiro do tipo void * que aponta para uma mesa que possui a chave a ser removida. Nesse caso, não é necessário utilizar a função “criaMesa” para alocar a memória para a nova mesa com a chave a ser buscada, basta criar uma variável do tipo Mesa, definir o campo id dela como sendo o id a ser removido e passar o endereço de memória dessa variável para a função. Por fim, depois de realizar a remoção, as informações da mesa removida são impressas na tela. A implementação desse caso está apresentada na imagem abaixo.

```c
case 2: //Remover elemento
    //Realizando a leitura dos dados
    printf("\nA opcao selecionada foi: Remover mesa.\n\n");
    printf("Digite a chave da mesa a ser removida: ");
    scanf("%d", &m.id);

    //Buscando pela chave a ser removida
    mesa = buscaChave(a, &m, compara);
    mesaBuscada = (Mesa *)mesa;

    if (mesaBuscada) //Caso a chave já exista na árvore
    {
        Mesa mesaRemovida;
        mesaRemovida.id = mesaBuscada->id;
        mesaRemovida.altura = mesaBuscada->altura;
        mesaRemovida.largura = mesaBuscada->largura;

        //Realiza a remoção
        a = remover(a, &h, &m, compara, liberaChave);

        //Imprime a mesa removida
        printf("\nA seguinte mesa foi removida:\n");
        printf("\nChave: %d\n", mesaRemovida.id);
        printf("Altura: %.2f\n", mesaRemovida.altura);
        printf("Largura: %.2f\n", mesaRemovida.largura);
    }
    else //Caso a chave não esteja na árvore
        printf("\nNao foi possivel remover. A mesa com chave %d nao foi encontrada.\n", m.id);
    break;
```

IMAGEM 14: Implementação do código para o caso de o menu chamar a opção de remover

4.2.4.3 Procurar

Caso o usuário escolha a opção de buscar uma mesa, assim como na remoção, será solicitado que ele informe a chave da mesa que está sendo buscada como é mostrado na imagem abaixo.

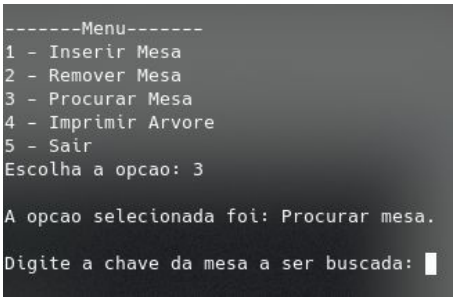


IMAGEM 15: Saída do programa caso a opção escolhida seja procurar.

Após pedir que o usuário digite a chave da mesa a ser buscada, a função de busca será chamada para buscar a chave na árvore. Caso a chave seja encontrada, as informações dela serão impressas. Caso ela não seja encontrada, será impressa uma mensagem de erro indicando que não foi possível encontrar a chave buscada. A implementação dessa função está na imagem a seguir.

```
case 3: //Buscar elemento
    //Leitura dos dados
    printf("\nA opcao selecionada foi: Procurar mesa.\n\n");
    printf("Digite a chave da mesa a ser buscada: ");
    scanf("%d", &m.id);

    //Realizando a busca da chave
    mesa = buscaChave(a, &m, compara);
    mesaBuscada = (Mesa *)mesa;

    if (mesaBuscada) //Caso a chave seja encontrada
    {
        printf("\nChave: %d\n", mesaBuscada->id);
        printf("Altura: %.2f\n", mesaBuscada->altura);
        printf("Largura: %.2f\n", mesaBuscada->largura);
    }
    else //Caso a chave não esteja na árvore
        printf("\nA mesa buscada nao foi encontrada.\n");
    break;
```

IMAGEM 16: Implementação do código para o caso de a opção escolhida no menu for procurar.

4.2.4.4 Imprimir

O caso de imprimir a árvore é bem simples. A única coisa que ele faz é verificar se a árvore está vazia. Caso a árvore esteja vazia, uma mensagem é impressa dizendo que ela está vazia. Caso contrário, a função de impressão da árvore é chamada. A implementação desse caso está na imagem a seguir.

```c
case 4: //Imprimir a árvore

    printf("\nA opcao selecionada foi: Imprimir arvore.\n\n");

    if (!a) //Caso a árvore esteja vazia
        printf("A arvore esta vazia.\n");
    else //Caso a árvore possua algum elemento
        imprimeArvore(a, 1, imprimeChave);
    break;
```

IMAGEM 17: Implementação do código para o caso de a opção escolhida no menu for imprimir.

4.2.5 Main

Além dessas funções já citadas, existe no arquivo main.c a função principal que é justamente a “main”. Essa função inicializa uma árvore vazia para realização das operações disponíveis no menu. A função “main” também é utilizada para a realização do laço de repetição (do-while) das opções do menu, funcionando da seguinte forma: Assim que o usuário escolher a opção desejada, a função “menu” será chamada recebendo como parâmetros a árvore que foi inicializada e o número referente a opção escolhida para assim realizar a operação. Além disso, a função de menu só é chamada caso a opção escolhida não seja a opção de sair. No fim, ao escolher a opção de sair, o programa irá sair do loop e a função “liberaArvore” irá liberar a memória que foi alocada para a árvore que foi utilizada. A implementação da função main está na imagem abaixo.

```c
//Função principal do programa
int main()
{
    //Inicializando uma árvore vazia
    AVL *a = NULL;

    int op = 0;
    do
    {
        //Opções do menu
        printf("\n-------Menu-------\n");
        printf("1 - Inserir Mesa\n");
        printf("2 - Remover Mesa\n");
        printf("3 - Procurar Mesa\n");
        printf("4 - Imprimir Arvore\n");
        printf("5 - Sair\n");

        //Leitura da opção
        printf("Escolha a opcao: ");
        scanf("%d", &op);

        //Se a opção selecionada não for a de sair
        if (op != 5)
            a = menu(a, op);

        //Permanece no menu enquanto não for selecionada a opção de sair
    } while (op != 5);

    //Libera a memória alocada para a árvore
    liberaArvore(a, liberaChave);

    return 0;
}
```

IMAGEM 18: Implementação da função main.

4.3 MAKEFILE

Também foi implementado o arquivo “makefile” contendo um conjunto de diretivas usadas para a compilação dos arquivos. Esse arquivo permite que todo o código seja compilado apenas com o comando make, e permite que todos os arquivos gerados pela compilação sejam removidos com o comando make clean. Esse arquivo torna a compilação do programa bem mais simples e prática. A implementação do arquivo makefile se encontra na imagem abaixo.

```
all: AVL

AVL: main.o AVL.o
    gcc -o AVL main.o AVL.o

main.o: main.c AVL.h
    gcc -c main.c -Wall

AVL.o: AVL.c AVL.h
    gcc -c AVL.c -Wall

clean:
    rm AVL main.o AVL.o
```

IMAGEM 19: Arquivo makefile

4.4 AVL.C

Outro arquivo implementado foi o arquivo AVL.c. Esse arquivo é muito importante pois nele se encontram as implementações das funções relativas à árvore AVL utilizadas no programa cliente. Essas funções são explicadas a seguir.

4.4.1 Cria nó

A função “criaNo”, é a função usada para alocar memória para um novo nó na árvore. Ela imprime uma mensagem de erro caso ocorra algum erro durante a alocação e, caso não haja nenhum erro, os campos desse nó são inicializados. Por fim a função retorna o novo nó.

```
// Função que aloca a memória para um novo nó da árvore
AVL *criaNo(void *chave)
{
    //Alocação de memória
    AVL *novo = (AVL *)malloc(sizeof(AVL));

    if (!novo) //Caso ocorra algum erro de alocação
    {
        printf("Erro de alocacao de memoria!");
        exit(1);
    }

    //Inicialização das variáveis
    novo->chave = chave;
    novo->fb = 0;
    novo->dir = NULL;
    novo->esq = NULL;

    return novo;
}
```

IMAGEM 20: Implementação da função criaNo.

4.4.2 Funções de rotação

Foram implementadas também as funções de rotação, essas funções são utilizadas para caso haja desbalanceamento na árvore, garantindo assim que a principal propriedade de uma árvore AVL seja respeitada.

4.4.2.1 Roda direita

A função “rodaDireita” é utilizada para realizar as rotações na inserção, seja essa rotação simples ou dupla. Ela funciona através de condicionais onde caso o fator de balanceamento da árvore seja igual a 1 (mesmo sinal do fator de V) será realizada a rotação simples para a direita, dessa forma a função realizará os apontamentos para a rotação. Da mesma forma irá acontecer caso ocorra uma rotação dupla, entretanto a condição do fator de balanceamento é que ele seja diferente de um. Após realizar a rotação propícia, o fator de balanceamento é colocado como zero, assim como a variável “h” para indicar que não existem mais alterações nela. O retorno dessa função será um ponteiro para o novo nó da árvore. A implementação se encontra na imagem abaixo.

```c
//Função que realiza as rotações simples e dupla para a direita ao inserir na árvore
AVL *rodaDireita(AVL *v, int *h)
{
    //Declaração das variáveis
    AVL *u, *z;

    //Filho a esquerda do nó desbalanceado
    u = v->esq;

    if (u->fb == 1) // Rotação simples a direita
    {
        //Realizando os reapontamentos
        v->esq = u->dir;
        u->dir = v;

        //Alterando o fator de balanceamento
        v->fb = 0;
        //Atualizando a raiz
        v = u;
    }
    else //Rotação dupla a direita
    {
        //Filho a direita do filho a esquerda do nó desbalanceado
        z = u->dir;

        //Realizando os reapontamentos
        u->dir = z->esq;
        z->esq = u;
        v->esq = z->dir;
        z->dir = v;

        //Alterando o fator de balanceamento para cada caso
        if (z->fb == 1)
            v->fb = -1;
        else
            v->fb = 0;

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

        //Atualizando a raiz
        v = z;
    }

    //Atualizando o fator de balanceamento do nó desbalanceado para 0
    v->fb = 0;
    //Indicando que não há mais alteração na altura
    *h = 0;
    return v;
}
```

IMAGEM 21: Implementação da função rodaDireita.

Como é possível perceber, essa função recebe um ponteiro para o nó da árvore que está desbalanceado (denominado de V) e um ponteiro para uma variável que contabiliza se a altura da árvore foi alterada ou não.

A função “rodaDireita” lida tanto com a rotação simples à direita, quanto com a rotação dupla à direita. Ela realiza cada tipo de rotação dependendo de cada caso. Esses casos serão explicados a seguir.

4.4.2.1.1 Rotação simples à direita

O primeiro caso é o de rotação simples à direita. Esse caso foi representado abaixo onde as letras em preto representam a altura da árvore e os números em vermelho representam os fatores de balanceamento de cada nó.

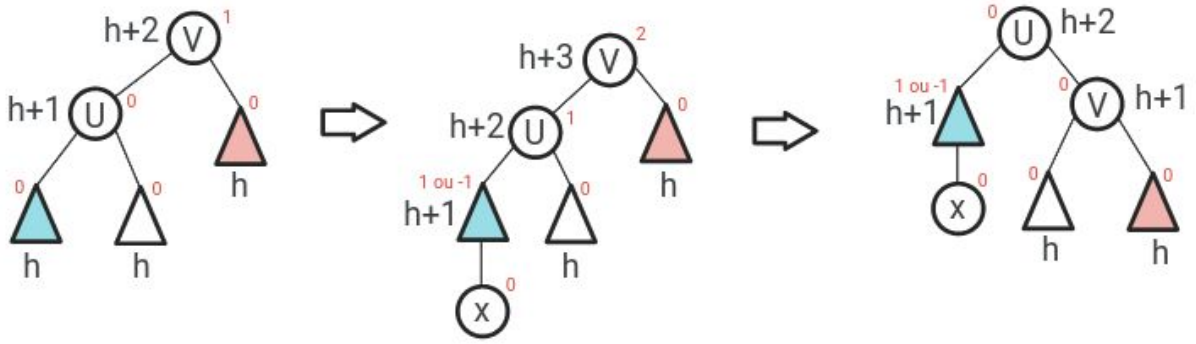


IMAGEM 22: Caso de rotação simples.

Ao analisar a imagem, é possível notar que a altura da árvore azul, depois da inserção do nó x se tornou h+1, e o fator de balanceamento da raiz dessa árvore pode ter se tornado tanto 1 quanto -1, dependendo de qual subárvore da árvore azul o no x foi inserido. Além disso, o fator de balanceamento do nó V após a inserção do elemento x se torna 2. Isso viola o balanceamento de uma árvore AVL, pois nesse tipo de árvore o fator de balanceamento pode ser apenas 1, 0 ou -1. Para resolver esse desbalanceamento, nesse caso foi necessário realizar apenas uma rotação simples à direita em V.

Em uma rotação simples à direita, o filho à esquerda de V recebe o filho à direita de U e o filho à direita de U recebe V. Além disso, após a rotação, o fator de balanceamento de V e de U se tornam 0 de acordo com a análise feita na imagem. Por fim, o novo nó raiz da árvore se torna o nó U.

De acordo com o desenho feito, é possível perceber que a rotação simples à direita ocorre quando o filho à esquerda do nó desbalanceado tem fator de balanceamento igual a 1 depois da inserção. Isso irá se comprovar à medida que forem apresentados os outros casos de rotação.

A implementação da rotação simples dentro da função de rotação à direita foi feita no seguinte trecho de código:

```
if (u->fb == 1) // Rotação simples a direita
{
    //Realizando os reapontamentos
    v->esq = u->dir;
    u->dir = v;

    //Alterando o fator de balanceamento
    v->fb = 0;
    //Atualizando a raiz
    v = u;
}
```

IMAGEM 23: Implementação do código para rotação simples à direita.

Como foi analisado no desenho, o caso de rotação simples à direita na inserção ocorre quando o fator de balanceamento de U for igual a 1. Isso é tratado no código por meio da estrutura condicional if. Depois disso, os apontamentos são feitos como foi descrito anteriormente de forma que o resultado final esteja de acordo com o desenho. Por fim, o fator de balanceamento de V é alterado para 0 e U se torna a nova raiz da árvore. Além disso, é possível perceber no fim da função de rotação que o fator de balanceamento da nova raiz também recebe 0. Assim, tanto o fator de balanceamento de U como o fator de balanceamento de V recebem 0. Dessa forma, o caso ilustrado de inserção que gera uma rotação simples à direita foi tratado pelo código. Esse é o único caso de rotação simples à direita pois ao inserir nas outras subárvores, será necessária uma rotação dupla ou nenhuma rotação.

4.4.2.1.2 Rotação dupla à direita

Caso a rotação simples não seja suficiente para resolver o desbalanceamento da árvore que foi causado pela inserção, outro caso de rotação pode ocorrer: a rotação dupla à direita. Os casos que geram esse tipo de rotação serão analisados a seguir.

O primeiro caso foi ilustrado na imagem abaixo. Nessa imagem, os triângulos representam possíveis subárvores, os círculos representam os nós, as letras em preto descrevem a altura de cada subárvore e as letras em vermelho descrevem o fator de balanceamento de cada nó.

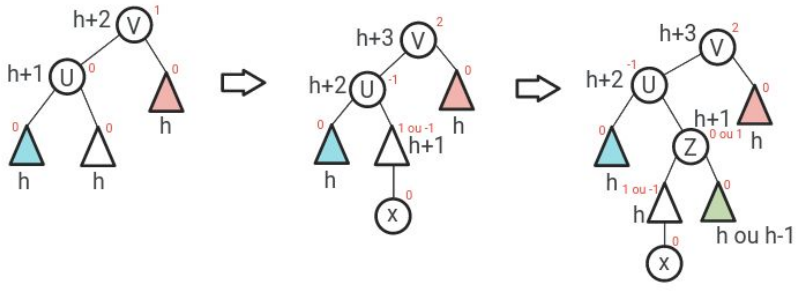


IMAGEM 24: Caso de rotação dupla.

Como é possível observar nessa imagem, ao inserir o nó na subárvore branca, a altura dela passou de h para h+1. Além disso, o fator de balanceamento da raiz daquela árvore se torna 1 ou -1, dependendo de qual subárvore o nó x for inserido. Outro ponto importante é que o fator de balanceamento do nó V foi alterado para 2 com a inserção de x. Isso viola o balanceamento da árvore AVL que só permite que um nó tenha fator de balanceamento 1, -1 ou 0. Para resolver isso, é necessário realizar uma rotação dupla para a direita.

Para compreender melhor a rotação dupla, foi feita uma expansão da subárvore branca em duas subárvores e foi dado o nome de Z para a raiz da antiga subárvore branca.

Como a inserção de x provocou a alteração da altura da subárvore branca, a altura dela passou a ser h+1. Consequentemente, a altura de Z também é h+1. Para que Z tenha altura h+1, é necessário que a altura de pelo menos uma das suas subárvores seja h. Se isso ocorrer, Z terá altura h+1. Isso implica que pelo menos uma das subárvores de Z deve ter a altura h, e a outra pode ter a altura h ou h-1.

Para o primeiro caso, será considerado que o nó x foi inserido na subárvore esquerda de Z. Como as subárvores de Z podem ter altura h ou h-1, e essa é a altura delas após a inserção de x, não é possível que após a inserção de x, um nó tenha altura h-1, pois ele deveria ter altura h-2 antes disso. Caso ele tivesse altura h-2 antes da inserção, haveria um desbalanceamento antes da inserção, pois o fator de balanceamento de Z seria (h-2) - h ou h - (h-2). Como a inserção ocorreu em uma árvore AVL balanceada, é mais plausível considerar que o nó que sofreu a inserção do nó x tinha altura h-1 e agora tem altura h. Como a subárvore em que foi inserido o x tem altura h e a outra pode ter altura h-1 ou h, entende-se que o fator de balanceamento de Z pode ser 0 ou 1 de acordo com o que foi ilustrado na figura. Esse fator de balanceamento também pode ser -1, mas inicialmente serão tratados os casos para 0 e 1. Caso o fator de balanceamento de Z assuma os valores de 0 ou 1, temos o seguinte caso:

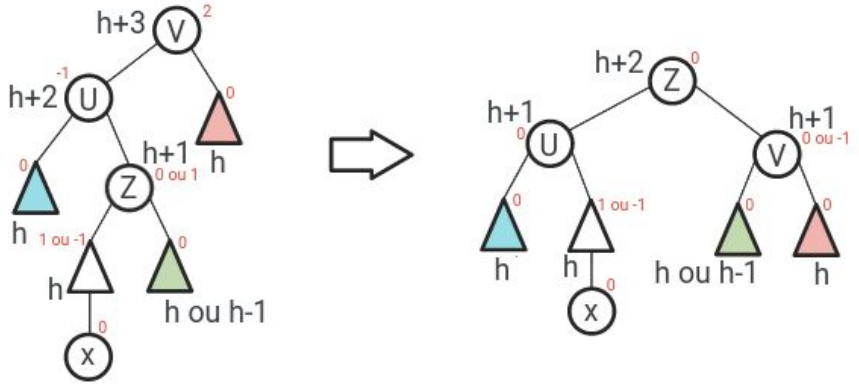


IMAGEM 25: Caso o fator de balanceamento de Z seja 0 ou 1.

Como é possível perceber, o fator de balanceamento de Z pode ser 0 ou 1 dependendo de qual subárvore o nó x for inserido e de qual for a altura da subárvore de Z em que x não foi inserido.

Para que a árvore fique na disposição desenhada na última imagem, é necessário realizar algumas modificações nos ponteiros. Inicialmente a subárvore direita de U recebe a subárvore esquerda de Z, depois a subárvore esquerda de Z recebe U, então a subárvore esquerda de V recebe a subárvore direita de Z e a subárvore direita de Z recebe V. Após realizar esses apontamentos, a árvore estará em uma distribuição de nós balanceada, restando apenas alterar os fatores de balanceamento.

Voltando para o caso em que o fator de balanceamento de Z pode ser 0 ou 1, é possível perceber que ele é 0 quando a altura da subárvore verde é h e é 1 quando a altura da subárvore verde é h-1.

Ao analisar a árvore final, é possível compreender que, nesse caso, o fator de balanceamento de V depende do fator de balanceamento de Z. Caso o fator de balanceamento de Z seja 1, o fator de balanceamento de V se torna -1, e caso ele seja 0, o fator de balanceamento de V se torna 0.

Outro caso ocorre quando o nó x é inserido na subárvore verde. Nesse caso, temos a seguinte árvore:

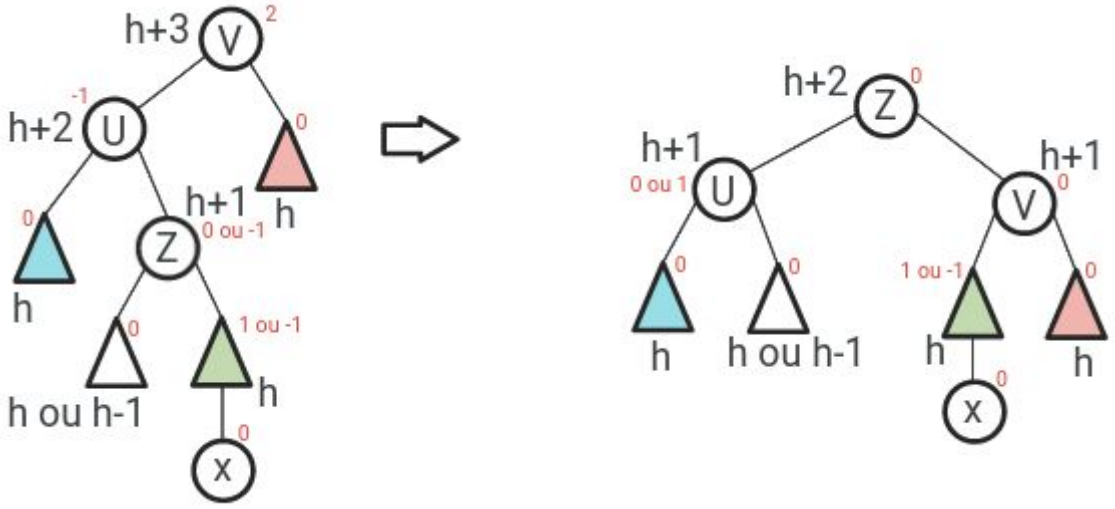


IMAGEM 26: Caso o fator de balanceamento de Z seja 0 ou -1.

Na árvore desenhada acima, o nó x foi inserido na subárvore verde em vez da branca. Com isso, o fator de balanceamento de Z só pode ser 0 ou -1 pois a altura da subárvore branca só pode ser h ou h-1, já que a altura da subárvore em que foi inserido o nó x já é h pelo mesmo motivo explicado para o caso anterior. Dessa forma, é possível perceber que agora o U tem o fator de balanceamento alterado de acordo com o fator de balanceamento de Z. Caso a altura da subárvore branca seja h, o fator de balanceamento de Z é 0 e o fator de balanceamento de U também deve ser 0. Caso a altura da subárvore branca seja h-1, o fator de balanceamento de Z é 1, e isso implica que o fator de balanceamento de U será 1 após a rotação.

Ao analisar os casos, é possível compreender que em todos os casos, o fator de balanceamento de Z após a rotação é sempre 0. Além disso, entende-se que tanto o fator de balanceamento de U quanto o de V dependem do fator de balanceamento de Z. Se o fator de balanceamento de Z for 0, os fatores de balanceamento de U e de V são 0. Se o fator de balanceamento de Z for 1, o fator de balanceamento de U é 0 e o de V é -1. Por fim, se o fator de balanceamento de Z for -1, o fator de balanceamento de U é 1 e o de V é 0. O trecho de código abaixo mostra como são tratadas as rotações duplas à direita dentro da função “rodaDireita”. Nele é possível perceber que os apontamentos foram feitos para reorganizar a árvore de forma balanceada e os casos descritos dos fatores de balanceamento de U e V com base no fator de balanceamento de Z também são tratados no código.

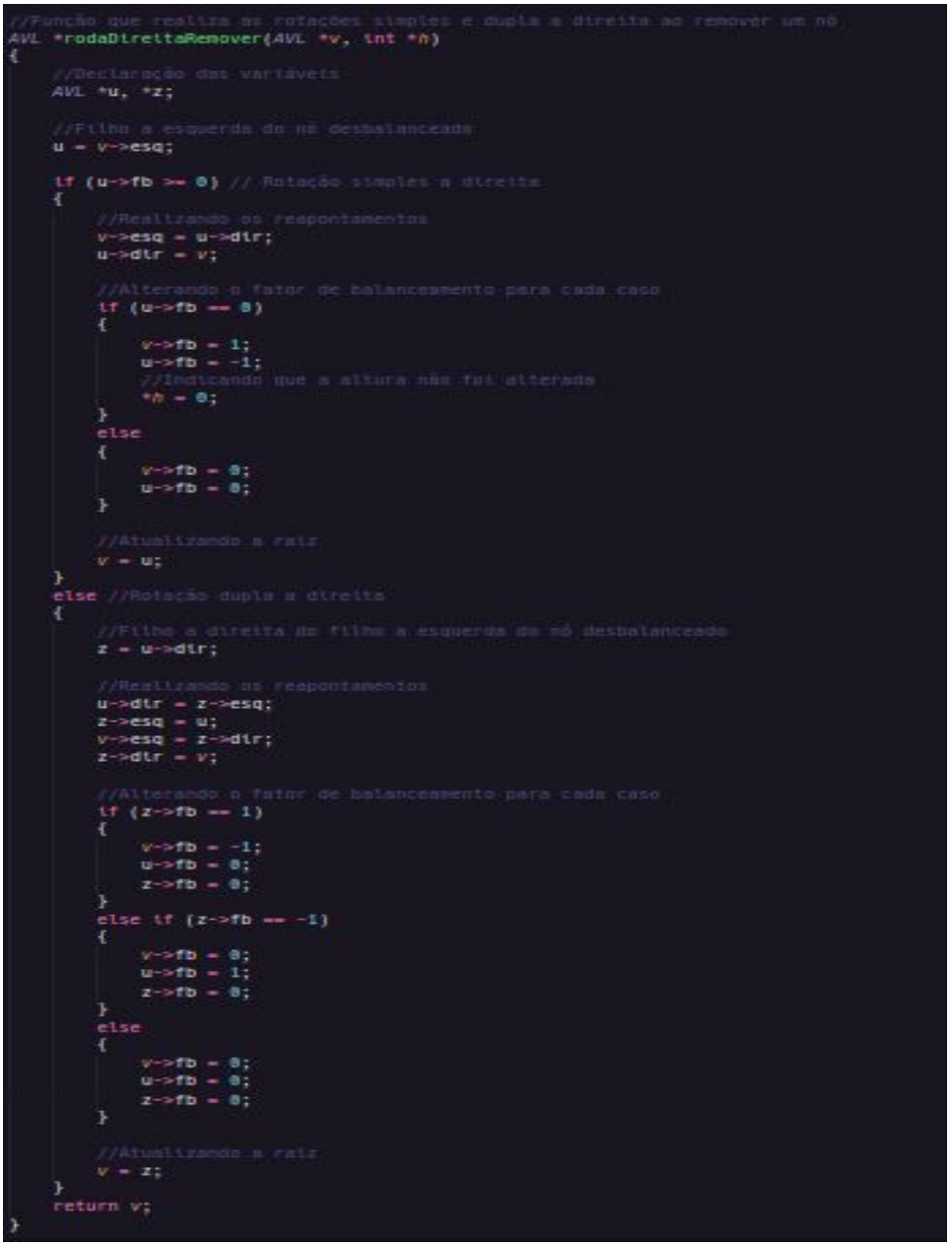
```
else //Rotação dupla a direita
{
    //Filho a direita do filho a esquerda do nó desbalanceado
    z = u->dir;

    //Realizando os reapontamentos
    u->dir = z->esq;
    z->esq = u;
    v->esq = z->dir;
    z->dir = v;

    //Alterando o fator de balanceamento para cada caso
    if (z->fb == 1)
        v->fb = -1;
    else
        v->fb = 0;

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

    //Atualizando a raiz
    v = z;
}
```

IMAGEM 27: Implementação do código para rotação dupla à direita.

4.4.2.2 Roda direita remover

A função rodaDireitaRemover é utilizada para realizar as rotações na remoção, seja essa rotação simples ou dupla. Essa função possui condições que funcionam assim: Caso o fator de balanceamento seja maior ou igual a 0, a árvore

vai realizar uma rotação simples para a direta, dessa forma a função realizará os apontamentos de forma que a árvore realize essa rotação. Dessa forma, caso o fator de balanceamento seja menor do que zero, a função realiza uma rotação dupla para a direita e mais uma vez a função realiza os apontamentos necessários. Após realizar a rotação propícia, são tratados os casos para a alteração do fator de balanceamento. O retorno dessa função será um ponteiro para o novo nó da árvore. A implementação dessa função se encontra na imagem abaixo.



```c
//Função que realiza as rotações simples e dupla a direita ao remover um nó
AVL *rodaDireitaRemover(AVL *v, int *h)
{
    //Declaração das variaveis
    AVL *u, *z;

    //Filho a esquerda do nó desbalanceado
    u = v->esq;

    if (u->fb >= 0) // Rotação simples a direita
    {
        //Realizando os reapontamentos
        v->esq = u->dir;
        u->dir = v;

        //Alterando o fator de balanceamento para cada caso
        if (u->fb == 0)
        {
            v->fb = 1;
            u->fb = -1;
            //Indicando que a altura não foi alterada
            *h = 0;
        }
        else
        {
            v->fb = 0;
            u->fb = 0;
        }

        //Atualizando a raiz
        v = u;
    }
    else //Rotação dupla a direita
    {
        //Filho a direita do filho a esquerda do nó desbalanceado
        z = u->dir;

        //Realizando os reapontamentos
        u->dir = z->esq;
        z->esq = u;
        v->esq = z->dir;
        z->dir = v;

        //Alterando o fator de balanceamento para cada caso
        if (z->fb == 1)
        {
            v->fb = -1;
            u->fb = 0;
            z->fb = 0;
        }
        else if (z->fb == -1)
        {
            v->fb = 0;
            u->fb = 1;
            z->fb = 0;
        }
        else
        {
            v->fb = 0;
            u->fb = 0;
            z->fb = 0;
        }

        //Atualizando a raiz
        v = z;
    }
    return v;
}
```

IMAGEM 28: Implementação da função rodaDireitaRemover.

A função descrita acima é utilizada na função de remoção para tratar os casos de balanceamento que podem ocorrer após a remoção de algum nó. Assim como a função de rotação utilizada para a inserção, essa função aborda os casos de rotação simples e dupla à direita. Esses casos serão mostrados a seguir.

4.4.2.2.1 Rotação simples à direita

O primeiro caso a ser tratado é o de rotação simples à direita. Essa rotação é descrita na imagem abaixo. Nessa imagem, os círculos representam os nós da árvore, os triângulos representam as possíveis subárvores que esses nós possuem, e os números em vermelho representam os fatores de balanceamento de cada nó. Além disso, foi indicada a altura de cada nó por meio das letras em preto.

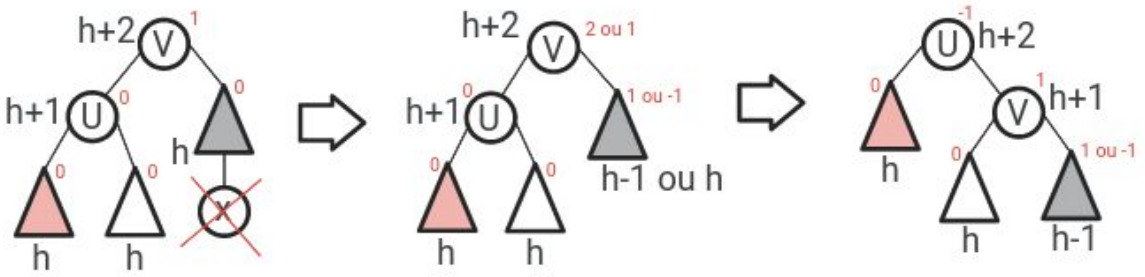


IMAGEM 29: caso a ser tratado é o de rotação simples à direita.

Ao remover um nó da direita, pode ser que a subárvore à direita fique com uma altura menor do que deveria, para resolver esse problema ocorreria uma rotação à direita.

No caso apresentado, a árvore inicialmente tem um nó x que é removido da subárvore cinza. Como a subárvore cinza tem altura h antes da remoção, após a remoção ela terá altura h-1, ou h. Isso se dá pois pode acontecer algum caso em que a remoção não altera a altura da árvore e assim, não é necessário realizar uma rotação. Esse caso será mostrado mais adiante no código da remoção. Entretanto, para o presente caso, será considerado que a altura após a remoção se tornou h-1 para que seja necessário realizar a rotação.

Como é possível perceber, após remover o nó x, os fatores de balanceamento mudaram e isso fez com que o fator de balanceamento do nó V fosse 2 para o caso da subárvore cinza ter altura h-1, e 1 para o caso dela ter altura h. Para o caso da rotação mencionada acima, o nó V terá fator de balanceamento 2. Isso desrespeita o balanceamento de uma árvore AVL e requer uma rotação na árvore para recuperar o balanceamento. Nesse caso, uma rotação simples basta e para realizar a rotação, basta fazer o filho à esquerda de V receber o filho à direita de U, e o filho à direita de U receber V. Isso resolve o problema do posicionamento dos nós, mas o fator de balanceamento requer uma análise mais apurada.

Para esse caso, em que o fator de balanceamento de U é 0, é possível notar que após a remoção e a rotação, o fator de balanceamento de U é -1 e o de V é 1. Por isso, esse caso foi tratado no código no seguinte trecho:

```
if (u->fb == 0)
{
    v->fb = 1;
    u->fb = -1;
    //Indicando que a altura não foi alterada
    *h = 0;
}
```

IMAGEM 30: Trecho do código que trata o caso do fator de balanceamento de u seja 0.

Nesse trecho o fator de balanceamento de U é verificado e os fatores de balanceamento de U e de V são definidos de acordo com o estudo de caso feito.

Outro caso importante ocorre quando o fator de balanceamento de U é igual a 1. Nesse caso, temos o seguinte processo:

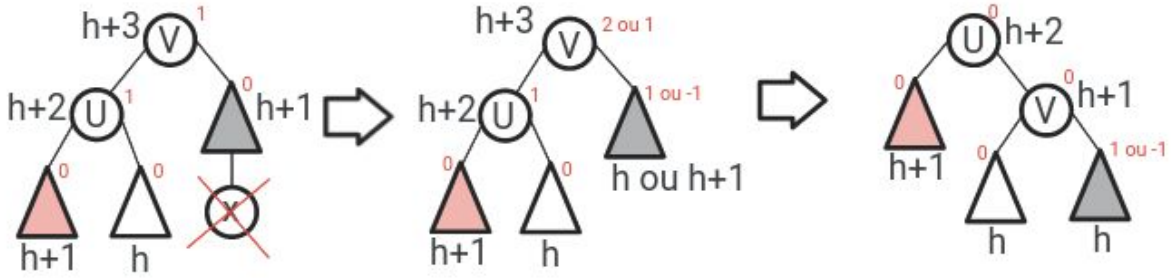


IMAGEM 31: Caso o fator de balanceamento de U seja 1.

Como é possível perceber, o processo realizado foi o mesmo, e o nó V continua com o mesmo problema de ter o fator de balanceamento 2, mas nesse caso o fator de balanceamento de U é 1. Para esse caso, após realizar a remoção do nó x e a rotação simples à direita, percebe-se que os nós U e V agora têm fator de balanceamento igual a 0. Isso se dá pois a diferença entre as alturas h e h, e as alturas h+1 e h+1 é zero. Esse caso também foi tratado no código no seguinte trecho:

```
else
{
    v->fb = 0;
    u->fb = 0;
}
```

IMAGEM 32: Caso o fator de balanceamento de V seja 2.

Esse trecho contendo o else é o caso contrário do condicional tratado anteriormente. Isso significa que quando o fator de balanceamento de U não for igual a 0, os fatores de balanceamento de U e V serão definidos como 0 pois entram no caso mostrado.

Vale ressaltar que quando o fator de balanceamento de U é igual a -1, é necessário realizar uma rotação dupla, que será explicada em breve. Por causa disso, foi necessário limitar a rotação simples para quando o fator de balanceamento de U for apenas maior ou igual a zero. Assim, essa rotação irá abranger apenas os casos do fator de balanceamento de U ser 0 ou 1. Reunindo todas essas informações é possível obter o código da rotação simples à direita. Esse código está exposto na imagem a seguir.

```
if (u->fb >= 0) // Rotação simples a direita
{
    //Realizando os reapontamentos
    v->esq = u->dir;
    u->dir = v;

    //Alterando o fator de balanceamento para cada caso
    if (u->fb == 0)
    {
        v->fb = 1;
        u->fb = -1;
        //Indicando que a altura não foi alterada
        *h = 0;
    }
    else
    {
        v->fb = 0;
        u->fb = 0;
    }

    //Atualizando a raiz
    v = u;
}
```

IMAGEM 33: Trecho de código para caso ocorra rotação simples à direita na remoção.

Nesse trecho de código, os casos serão aplicados apenas para fatores de balanceamento de U maiores ou iguais a zero. Além disso, é possível notar os apontamentos descritos no código, e os casos sendo tratados. Por fim, a raiz da árvore é atualizada para ser o nó U.

4.4.2.2.2 Rotação dupla à direita

O outro caso estudado foi o caso de rotação dupla à direita. Essa rotação, como será possível perceber ao longo dos exemplos, só ocorre quando o fator de balanceamento de U for -1, que possui sinal oposto ao fator de balanceamento de V. Assim sendo, os casos serão analisados a seguir.

Como foi dito anteriormente, a rotação dupla ocorre quando U tem fator de balanceamento -1. Nesse caso, uma rotação simples não resolveria o problema. Assim sendo, os casos de rotação dupla serão explicados a seguir.

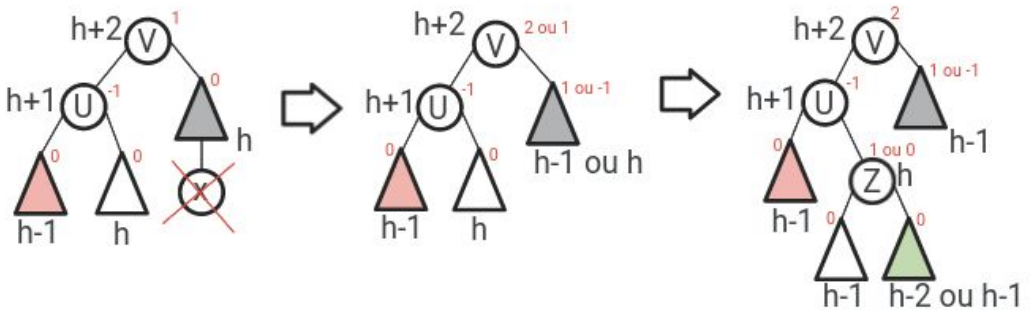


IMAGEM 34: Caso de rotação dupla à direita.

Como é possível observar no desenho acima, ao remover o nó x, ocorreu um desbalanceamento no nó V, que fica com fator de balanceamento 2 ou 1 dependendo da altura da subárvore cinza. Se ao remover o nó x, a altura da árvore não for alterada, ela continuará com altura h e não haverá necessidade de uma rotação. Para que seja possível demonstrar a rotação, será considerado o caso em que a altura da subárvore cinza é h-1 e o fator de balanceamento de V é 2.

Como o fator de balanceamento de U é -1, trata-se de uma rotação dupla. Sendo assim, é necessário expandir a subárvore branca em duas outras subárvores e chamar a raiz da subárvore branca que foi expandida de Z.

Ao analisar a subárvore com raiz em Z, é possível notar que para Z ter altura h, é necessário que pelo menos uma de suas subárvores tenha altura h-1. Esse caso foi indicado na figura e ele afeta o fator de balanceamento de Z, que pode ser -1, 1 ou 0. Para o caso do fator de balanceamento de Z ser 1, temos o seguinte caso:

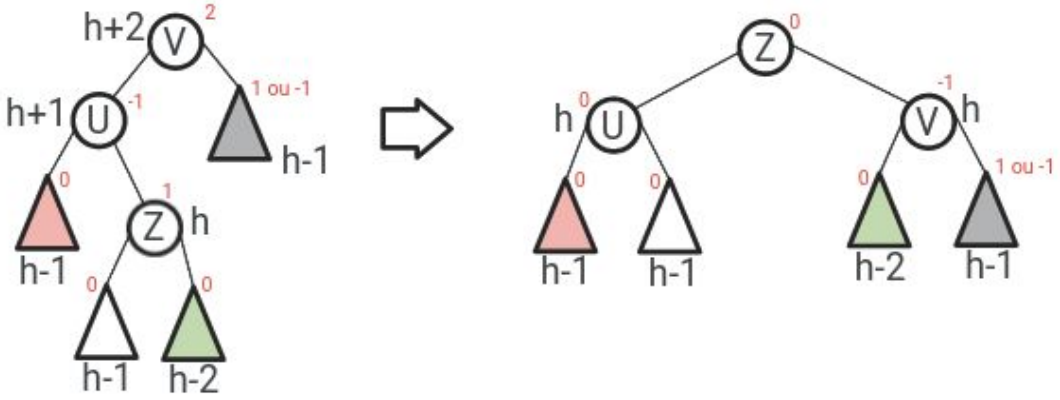


IMAGEM 35: Caso do fator de balanceamento de Z seja 1.

Nesse caso, o fator de balanceamento de Z é 1 pois foi considerado que a subárvore branca tem altura h-1 e a subárvore verde tem altura h-2. Para esse caso, é possível notar que caso o fator de balanceamento de Z seja 1, o fator de

balanceamento de U é 0 e o de V é -1. Esse caso está presente no código no seguinte trecho:

```
if (z->fb == 1)
{
    v->fb = -1;
    u->fb = 0;
    z->fb = 0;
}
```

IMAGEM 36: Trecho de código para o caso do fator de balanceamento de Z ser 1.

Nesse trecho, foi tratado o caso em que o fator de balanceamento de Z é 1, e os fatores de balanceamento de U e V foram definidos para esse caso. Além disso, como é possível observar no desenho, o fator de balanceamento de Z se torna 0.

Outro caso acontece quando o fator de balanceamento de Z é -1. Nesse caso, temos a seguinte rotação:

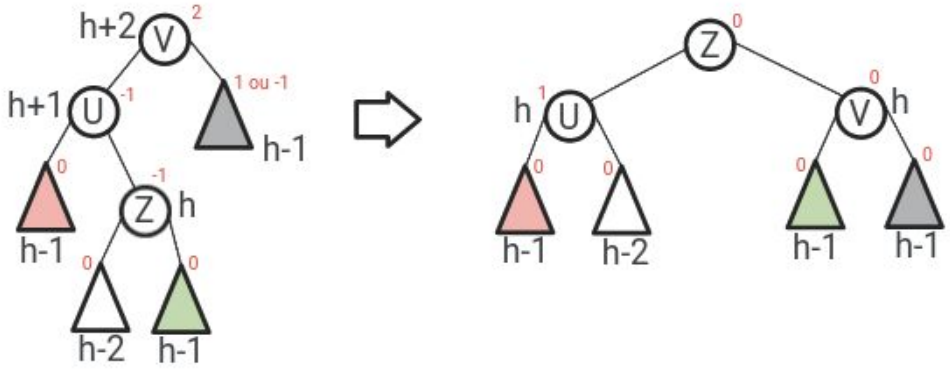


IMAGEM 37: Caso o fator de balanceamento de Z seja -1.

Como é possível perceber, o fator de balanceamento de Z foi -1 pois a altura da subárvore branca foi considerada h-2 e a altura da subárvore verde foi considerada h-1. Isso ocasiona um fator de balanceamento de -1 em Z. Dessa forma, após a rotação para esse caso, o fator de balanceamento de V é 0 e o de U é 1. Esse caso está presente no seguinte trecho de código:

```
else if (z->fb == -1)
{
    v->fb = 0;
    u->fb = 1;
    z->fb = 0;
}
```

IMAGEM 38: Trecho de código que trata o caso do fator de balanceamento de zer ser -1.

Como é possível perceber, o caso do fator de balanceamento de Z ser igual a -1 foi tratado e os fatores de balanceamento de U e de V foram definidos para o caso. Além disso, é possível observar que o fator de balanceamento de Z recebe 0.

O último caso ocorre quando o fator de balanceamento de Z é 0. Esse caso foi ilustrado a seguir:

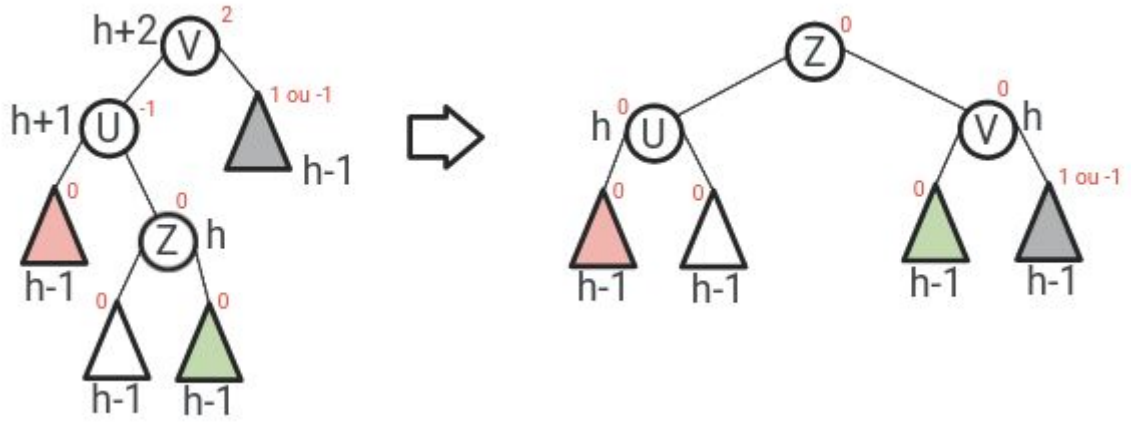


IMAGEM 39: Caso o fator de balanceamento de Z seja 0.

Nesse caso, para que o fator de balanceamento de Z fosse zero, a altura das árvores branca e verde foram definidas como h-1. Isso resulta em um fator de balanceamento 0 em Z.

Como é possível observar, após a remoção, nesse caso, tanto U como V possuem fator de balanceamento 0. Sendo assim, quando o fator de balanceamento de Z é 0, os de U e V também são 0. Esse caso foi tratado no seguinte código:

```
else
{
    v->fb = 0;
    u->fb = 0;
    z->fb = 0;
}
```

IMAGEM 40: Trecho de código para caso do fator de balanceamento de Z ser 0.

Na organização do código, esse else vem depois dos ifs dos casos anteriores, então o programa só entra nele caso o fator de balanceamento não seja -1 nem 1. Dessa forma, o else representa o caso do fator de balanceamento de Z ser 0, e ele define o fator de balanceamento de U, V e Z como sendo 0, como ocorre no caso apresentado.

Uma coisa que há em comum entre os casos é que todos eles utilizam a mesma sequência de apontamentos e o fator de balanceamento de Z recebe sempre 0. Essa sequência de apontamentos para balancear a posição dos nós é feita tomando o filho a direita de U como sendo o filho a esquerda de Z, o filho a

esquerda de Z como sendo U, o filho a esquerda de V como sendo o filho à direita de Z, e o filho à direita de Z como sendo V.

Reunindo todos os casos e os apontamentos, é possível obter a parte da função que trata das rotações duplas, que é a seguinte:

```
else //Rotação dupla a direita
{
    //Filho a direita do filho a esquerda do nó desbalanceado
    z = u->dir;

    //Realizando os reapontamentos
    u->dir = z->esq;
    z->esq = u;
    v->esq = z->dir;
    z->dir = v;

    //Alterando o fator de balanceamento para cada caso
    if (z->fb == 1)
    {
        v->fb = -1;
        u->fb = 0;
        z->fb = 0;
    }
    else if (z->fb == -1)
    {
        v->fb = 0;
        u->fb = 1;
        z->fb = 0;
    }
    else
    {
        v->fb = 0;
        u->fb = 0;
        z->fb = 0;
    }

    //Atualizando a raiz
    v = z;
}
```

IMAGEM 41: Trecho de código para a rotação dupla à direita.

Nesse código, o primeiro else indica que o fator de balanceamento de U não é maior ou igual a 0, ou seja, ele é igual a -1 e indica o caso de uma rotação dupla pois tem sinal oposto ao de V. Depois disso, dentro do código apenas foram feitos os apontamentos, o tratamento do fator de balanceamento para cada caso e a atualização da raiz da árvore que agora passa a ser Z.

4.4.2.3 Roda esquerda

A função rodaEsquerda é utilizada para realizar as rotações na inserção, seja essa rotação simples ou dupla. Nessa função, caso o fator de balanceamento da árvore seja igual a -1 será realizada a rotação simples pela esquerda, dessa forma a função realizará os apontamentos de forma que a árvore tenha realizado a rotação. O mesmo irá acontecer caso ocorra uma rotação dupla, entretanto a condição do

fator de balanceamento é que ele seja diferente de -1. Após realizar a rotação adequada, o fator de balanceamento é colocado como zero, assim como a variável que controla se houve alteração de altura, para indicar que não existem mais alterações nela. O retorno dessa função será um ponteiro para o novo nó da árvore.

```c
//Função que realiza as rotações simples e dupla para a esquerda ao inserir na árvore
AVL *rodaEsquerda(AVL *v, int *h)
{
    AVL *u, *z;

    //Filho a direita do nó desbalanceado
    u = v->dir;

    if (u->fb == -1) //Rotação simples a esquerda
    {
        //Realizando os reapontamentos
        v->dir = u->esq;
        u->esq = v;

        //Atualizando o fator de balanceamento
        v->fb = 0;
        //Atualizando a raiz
        v = u;
    }
    else //Rotação dupla a esquerda
    {
        //Filho a esquerda do filho a direita do nó desbalanceado
        z = u->esq;

        //Realizando os reapontamentos para a rotação
        u->esq = z->dir;
        z->dir = u;
        v->dir = z->esq;
        z->esq = v;

        //Alterando o fator de balanceamento para cada caso
        if (z->fb == 1)
            u->fb = -1;
        else
            u->fb = 0;

        if (z->fb == -1)
            v->fb = 1;
        else
            v->fb = 0;

        //Atualizando a raiz
        v = z;
    }

    //Zerando o fator de balanceamento da raiz
    v->fb = 0;
    //Indicando que não houve mais alteração de altura
    *h = 0;
    return v;
}
```

IMAGEM 42: Implementação da função rodaEsquerda.

Ao analisar a implementação dessa função inserida acima é possível perceber que ela é muito semelhante a função rodaDireita já explicada anteriormente, porém com algumas pequenas diferenças. Os casos tratados são análogos, mas para uma rotação no sentido oposto. A rotação à esquerda também possui casos de rotação simples e dupla. A seguir serão explicados cada um deles.

4.4.2.3.1 Rotação simples à esquerda

O primeiro caso é o de rotação simples à esquerda. Nesse caso, apenas uma rotação simples basta para que a árvore esteja balanceada novamente.

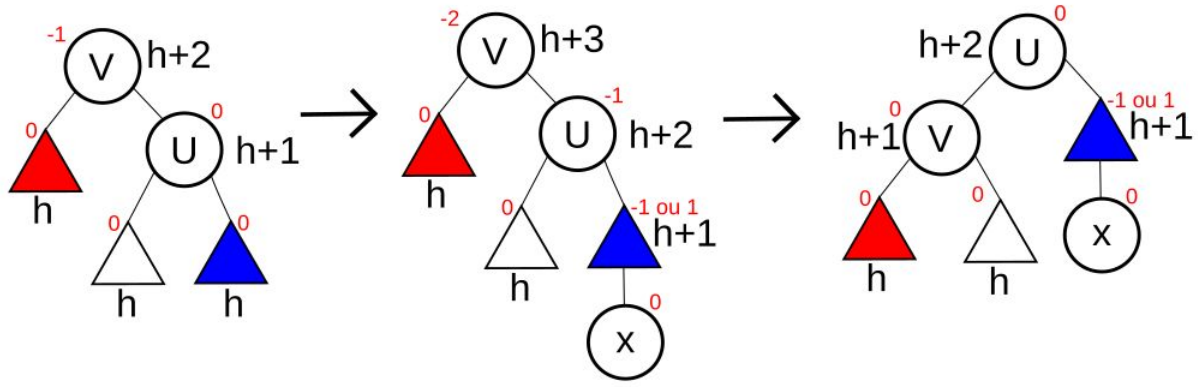


IMAGEM 43: Demonstração de uma rotação simples à esquerda ao inserir um elemento.

Como é possível perceber, o processo de rotação simples à esquerda é análogo ao processo de rotação simples à direita. Ao inserir um novo nó na subárvore azul de forma que altere a altura dela, é necessário realizar uma rotação para a esquerda, pois o nó V ficou com fator de balanceamento -2 depois da inserção, o que viola o balanceamento da árvore AVL. Sendo assim, ao analisar o caso, é possível compreender que uma rotação simples já soluciona o problema. Nessa rotação, para reorganizar os nós são feitos alguns apontamentos: o filho à direita de V recebe o filho à esquerda de U e o filho à esquerda de U recebe V.

Outra informação importante é que o fator de balanceamento tanto de U quanto de V são 0 na rotação simples da inserção. Além disso, como uma rotação simples solucionou o problema, é possível concluir que quando o fator de balanceamento de U é -1 (mesmo sinal do fator de V) depois da inserção, é um caso de rotação simples. Com base nisso, é possível implementar o código.

```
if (u->fb == -1) //Rotação simples a esquerda
{
    //Realizando os reapontamentos
    v->dir = u->esq;
    u->esq = v;

    //Atualizando o fator de balanceamento
    v->fb = 0;
    //Atualizando a raiz
    v = u;
}
```

IMAGEM 44: Implementação do código de rotação simples para esquerda (inserção).

Como é possível observar, o código verifica se o fator de balanceamento de U é -1 para saber se é um caso de rotação simples à direita ou não. Depois disso, ele realiza as devidas mudanças nos ponteiros que foram descritas anteriormente e atualiza o fator de balanceamento de V para 0. Depois disso, a nova raiz da árvore passa a ser U. Vale ressaltar que no final da função o fator de balanceamento da raiz também é definido como 0.

4.4.2.3.2 Rotação dupla à esquerda

O outro caso presente na função rodaEsquerda é o caso da rotação dupla. Esse caso ocorre quando o fator de balanceamento de U não é -1.

Em geral, é necessário realizar uma rotação dupla quando apenas uma rotação simples não resolve o problema do desbalanceamento e o fator de balanceamento de U não tem o mesmo sinal do de V. Para a rotação dupla à direita, foram observados os seguintes casos:

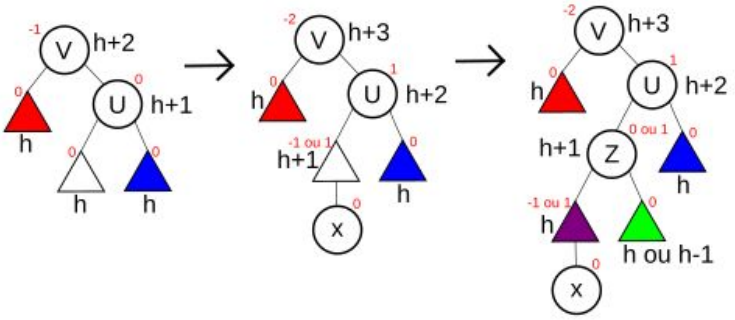


IMAGEM 45: Caso de rotação dupla à esquerda.

Durante o processo de inserção na subárvore branca, é possível perceber que houve o desbalanceamento do nó V, pois ele está com fator de balanceamento -2. Isso viola o balanceamento de uma árvore AVL, pois ela permite apenas os valores 1, -1 e 0.

Como já foi dito, será necessário realizar uma rotação dupla para a esquerda para que a árvore fique balanceada. Assim, será tratada a rotação para quando Z apresentar os fatores de balanceamento 1, -1 e 0.

O primeiro caso é quando o fator de balanceamento de Z é 1.

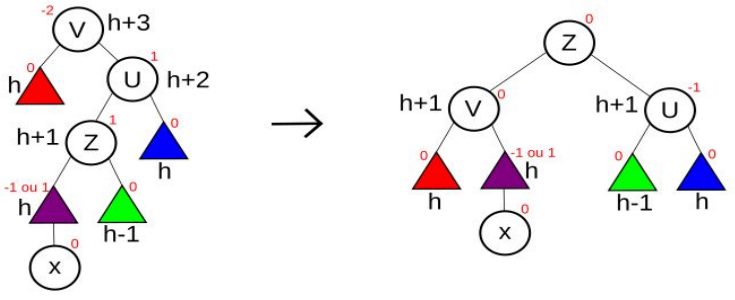


IMAGEM 46: Quando o fator da balanceamento de Z for 1.

Como é possível observar, Z possui fator de balanceamento 1 quando o nó x é inserido na subárvore roxa e a subárvore verde tem altura h-1. De acordo com o desenho, ao realizar a rotação quando o fator de balanceamento de Z é 1, o fator de balanceamento de V é 0 e o de U é -1.

Outro caso é quando o fator de balanceamento de Z é -1. Nesse caso, é possível obter o seguinte resultado:

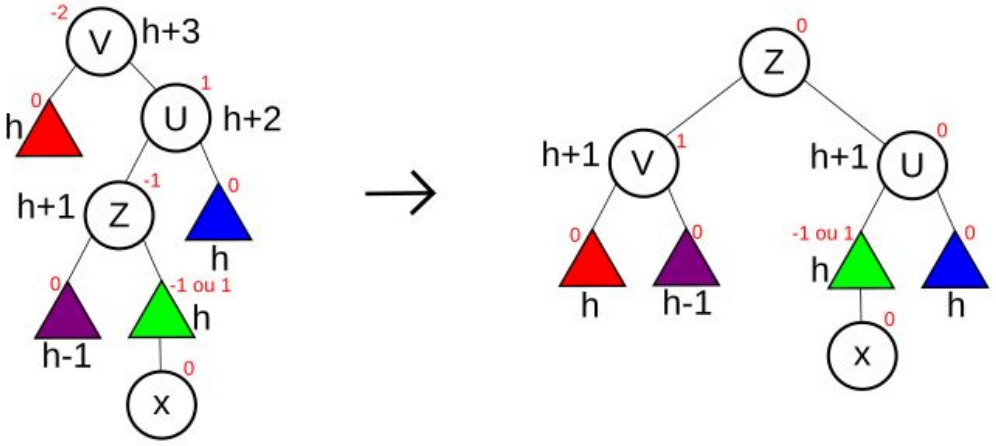


IMAGEM 47: Quando o fator de balanceamento de z for -1.

Como é possível observar, o fator de balanceamento de Z é -1 quando o nó x é inserido na subárvore verde e a subárvore roxa tem altura h-1. Nesse caso, após realizar a rotação, o fator de balanceamento de V é 1 e o de U é 0.

O último caso ocorre quando o fator de balanceamento de Z é 0.

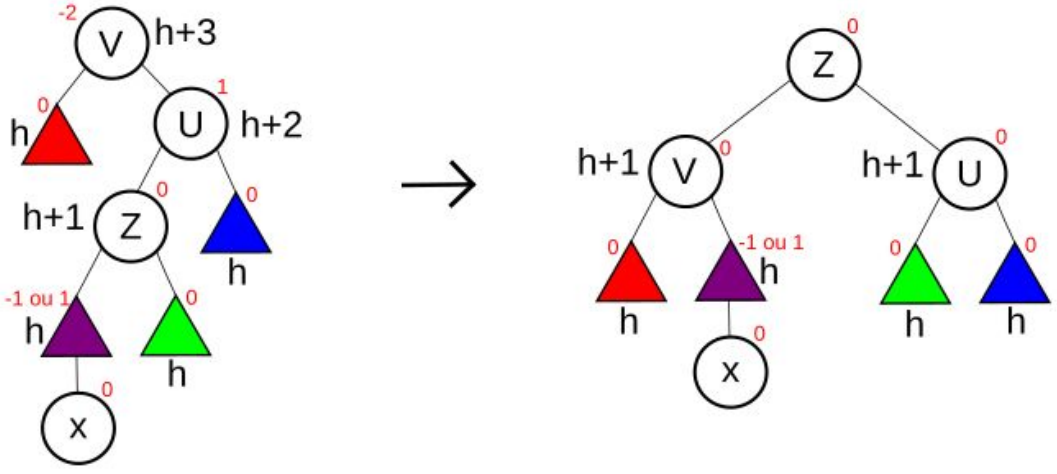


IMAGEM 48: Quando o fator de balanceamento de Z for 0.

Para que o fator de balanceamento de Z seja 0, é necessário que o nó x seja inserido em alguma subárvore de Z e que a outra subárvore tenha altura h. Nesse caso, é possível notar que o fator de balanceamento de V e de U são 0 após inserir.

Após analisar todos os casos, é possível perceber que nesse tipo de rotação, Z sempre será a nova raiz da árvore e terá fator de balanceamento 0 após a inserção. Além disso, as alterações que foram feitas nos ponteiros para realizar a rotação na árvore são sempre as mesmas em todos os casos. Essas alterações consistem em fazer o filho à esquerda de U receber o filho à direita de Z, o filho à direita de Z receber U, o filho à direita de V receber o filho à esquerda de Z e o filho à esquerda de Z receber V. Esse processo de rotação e esses casos foram tratados na função “rodaEsquerda” no processo de rotação dupla à esquerda a seguir.

```
else //Rotação dupla a esquerda
{
    //Filho a esquerda do filho a direita do nó desbalanceado
    z = u->esq;

    //Realizando os reapontamentos para a rotação
    u->esq = z->dir;
    z->dir = u;
    v->dir = z->esq;
    z->esq = v;

    //Alterando o fator de balanceamento para cada caso
    if (z->fb == 1)
        u->fb = -1;
    else
        u->fb = 0;

    if (z->fb == -1)
        v->fb = 1;
    else
        v->fb = 0;

    //Atualizando a raiz
    v = z;
}
```

IMAGEM 49: Trecho do código que trata a rotação dupla à esquerda na inserção.

.

Nesse código, o primeiro else diz respeito ao if que verifica se a rotação é simples à esquerda. Dessa forma, caso a rotação não seja um caso de rotação simples, será executada uma rotação dupla.

Como é possível observar, foram feitos os apontamentos como descrito anteriormente, e os fatores de balanceamento foram alterados obedecendo os casos descritos. Por fim, a nova raiz da árvore se torna o Z.

4.4.2.4 Roda esquerda Remover

A função “rodaEsquerdaRemover” é utilizada para realizar as rotações à esquerda na remoção, seja essa rotação simples ou dupla. Essa função possui condições que funcionam assim: caso o fator de balanceamento seja menor ou igual a 0, a árvore vai realizar uma rotação simples para a esquerda, dessa forma a função irá realizar os apontamentos de forma que a árvore tenha realizado essa rotação. Caso o fator de balanceamento seja maior que zero, a função realiza uma rotação dupla para a esquerda e, mais uma vez, a função realiza os apontamentos necessários. Após realizar a rotação propícia, todos os casos para a alteração do fato de balanceamento são tratados. Além disso, o retorno dessa função será um ponteiro para o novo nó raiz da árvore. Essa função se encontra na imagem abaixo.

```
//Função que realiza as rotações simples e dupla a esquerda ao remover um nó
AVL *rodaEsquerdaRemover(AVL *v, int *h)
{
    AVL *u, *z;

    //Filho a direita do nó desbalanceado
    u = v->dir;

    if (u->fb <= 0) //Rotação simples a esquerda
    {
        //Realizando os reapontamentos
        v->dir = u->esq;
        u->esq = v;

        //Atualizando o fator de balanceamento pra cada caso
        if (u->fb == 0)
        {
            v->fb = -1;
            u->fb = 1;
            //Indicando que não houve alteração na altura
            *h = 0;
        }
        else
        {
            u->fb = 0;
            v->fb = 0;
        }

        //Atualizando a raiz da árvore
        v = u;
    }
```

IMAGEM 50.1: Implementação da função rodaEsquerdaRemover.

```
    }
    else //Rotação dupla a esquerda
    {
        //Filho a esquerda do filho a direita do nó removido
        z = u->esq;

        //Realizando os reapontamentos
        u->esq = z->dir;
        z->dir = u;
        v->dir = z->esq;
        z->esq = v;
```

IMAGEM 50.2: Implementação da função rodaEsquerdaRemover.

```
        //Atualizando os fatores de balanceamento para cada caso
        if (z->fb == 1)
        {
            v->fb = 0;
            u->fb = -1;
            z->fb = 0;
        }
        else if (z->fb == -1)
        {
            v->fb = 1;
            u->fb = 0;
            z->fb = 0;
        }
        else
        {
            v->fb = 0;
            u->fb = 0;
            z->fb = 0;
        }

        //Atualizando a raiz da árvore
        v = z;
    }
    return v;
}
```

IMAGEM 50.3: Implementação da função rodaEsquerdaRemover.

A função rodaEsquerdaRemover apresentada acima lida com todos os casos de rotação para a remoção de uma chave da árvore, tanto os casos de rotação simples quanto os casos de rotação dupla. Esses casos serão abordados a seguir.

4.4.2.4.1 Rotação simples à esquerda

Inicialmente serão considerados os casos de rotação simples à esquerda para a remoção. O primeiro caso de rotação simples é o seguinte.

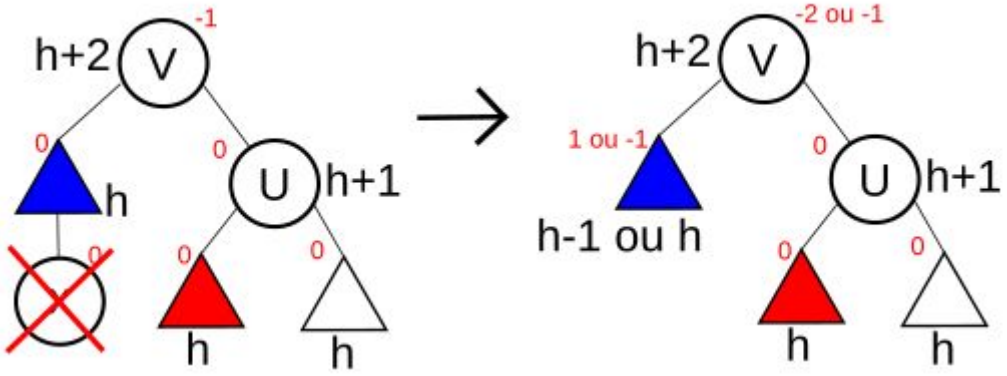


IMAGEM 51: Caso de rotação simples à esquerda.

Como é possível perceber, ao realizar a remoção do nó, o fator de balanceamento da subárvore azul pode ser tanto h-1 quanto h. Isso vai depender se, ao remover o nó, houve alteração na altura da subárvore azul ou não. Caso não haja alteração da altura, não é necessário realizar uma rotação, apenas alterar os fatores de balanceamento que estiverem errados. Para explicar o caso de rotação, será

considerado o caso em que a altura da subárvore azul foi alterada e é h-1. Assim sendo, a rotação ocorre da seguinte forma:

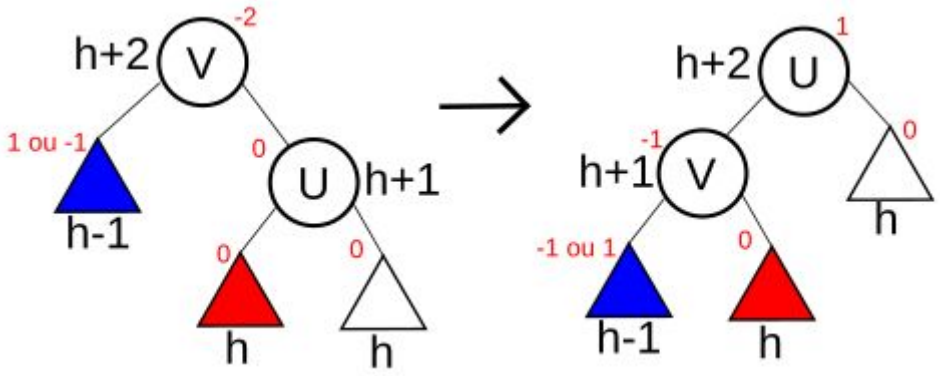


IMAGEM 52: Caso o fator de balanceamento de U seja 0.

Como é possível perceber, ao realizar a rotação, para o caso do fator de balanceamento de U ser 0, o fator de balanceamento de V é -1 e o fator de balanceamento de U é 1. Esse caso foi tratado na função “rodaEsquerdaRemover” no código abaixo, em que é verificado se o fator de balanceamento de U é zero e, se for, são atribuídos os devidos fatores de balanceamento para U e V.

```
if (u->fb == 0)
{
    v->fb = -1;
    u->fb = 1;
    //Indicando que não houve alteração na altura
    *h = 0;
}
```

IMAGEM 53: Trecho de código para o caso do fator de balanceamento de U ser 0.

Outro caso de rotação simples ocorre quando o fator de balanceamento de U é -1. Nesse caso, para obter esse fator de balanceamento em U, é necessário ter a seguinte árvore:

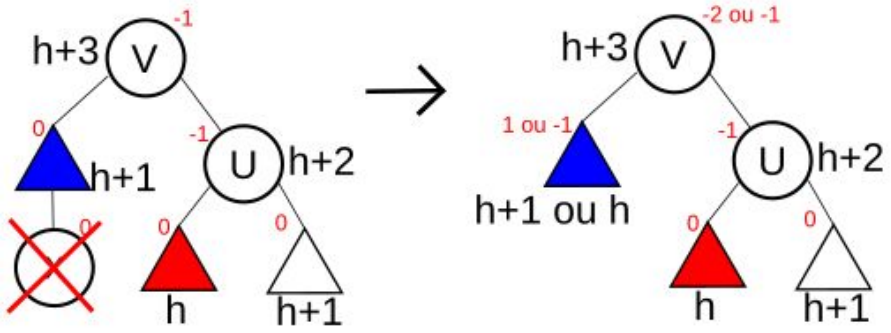


Imagem 54: Caso o fator de balanceamento de U seja -1.

Ao realizar a remoção do nó da subárvore azul, pode ser que a altura seja alterada ou não dependendo de onde o nó foi removido. Isso indica que a nova altura da subárvore pode ser h+1 ou h. Considerando que a remoção do nó causou uma diminuição da altura, será considerado que a altura da subárvore azul após a remoção é h, para que seja necessário realizar a rotação. Nesse caso, a rotação será da seguinte forma:

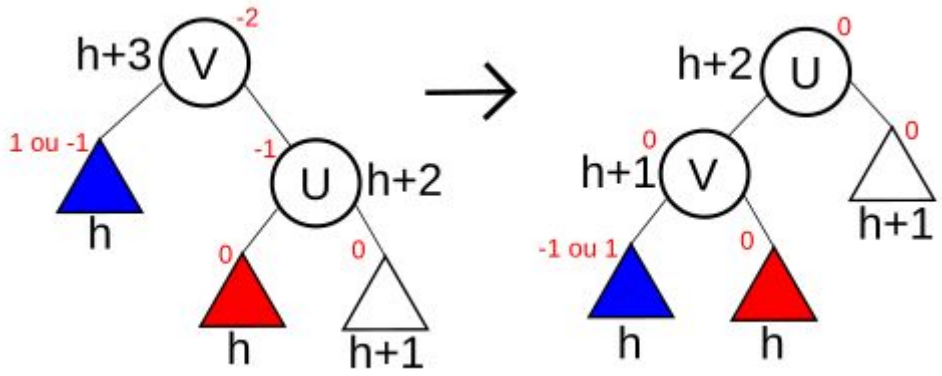


IMAGEM 55: Caso o fator de balanceamento de V seja -2.

Como é possível perceber, ao remover o nó de forma que ele altere a altura da subárvore azul, o fator de balanceamento de V se torna -2. Isso desbalanceia a árvore pois em uma árvore AVL o fator de balanceamento só pode ser 1, -1 ou 0. Para resolver isso, novamente basta realizar uma rotação simples à esquerda. Após realizar essa rotação, quando o fator de balanceamento de U é -1, é possível perceber que os fatores de balanceamento de U e de V são 0. Esse caso também foi tratado na função “rodaEsquerdaRemover” no seguinte trecho de código:

```
else
{
    u->fb = 0;
    v->fb = 0;
}
```

IMAGEM 56: Trecho do código para o caso de U ser -1.

Esse else veio logo após o if do caso anterior tratando o caso do fator de balanceamento de U ser 0. Nesse caso, os fatores de balanceamento de U e de V são definidos como 0.

Após analisar os casos de rotação simples, é possível notar que são feitos os mesmos apontamentos para rotacionar em cada caso. Esses apontamentos são: o filho à direita de V recebe o filho à esquerda de U e o filho à esquerda de U recebe

V. Além disso, nos casos abordados, após a rotação simples à direita, a nova raiz da árvore se torna U.

Outro ponto importante é que a rotação simples ocorre apenas quando o fator de balanceamento de U é 0 ou -1. Caso o fator de balanceamento de U seja 1, (sinal oposto ao de V) já é um caso de rotação dupla à esquerda que será mostrado adiante.

```
if (u->fb <= 0) //Rotação simples a esquerda
{
    //Realizando os reapontamentos
    v->dir = u->esq;
    u->esq = v;

    //Atualizando o fator de balanceamento pra cada caso
    if (u->fb == 0)
    {
        v->fb = -1;
        u->fb = 1;
        //Indicando que não houve alteração na altura
        *h = 0;
    }
    else
    {
        u->fb = 0;
        v->fb = 0;
    }

    //Atualizando a raiz da árvore
    v = u;
}
```

IMAGEM 57: Trecho do código para a rotação simples à esquerda (remoção).

Ao analisar o código, é possível perceber que foi verificada a condição do fator de balanceamento de U ser menor ou igual a 0 para ser uma rotação simples, depois foram feitos os apontamentos descritos anteriormente e foram tratados os casos para quando o fator de balanceamento de U for 0 ou -1. Por fim, a nova raiz da árvore foi definida como U.

4.4.2.4.2 Rotação dupla à esquerda

O outro caso de rotação abordado na função “rodaEsquerdaRemover” foi o caso da rotação dupla à esquerda. Nesses casos de rotação, uma rotação simples não resolve o problema, e é necessário realizar uma rotação dupla. Os casos da rotação dupla serão analisados a seguir.

Como já foi dito anteriormente, uma rotação dupla à esquerda vai acontecer quando se remove um elemento da subárvore esquerda de V e o fator de balanceamento de U é 1, apresentando sinal oposto ao fator de balanceamento de V. Nesse caso, temos a seguinte situação:

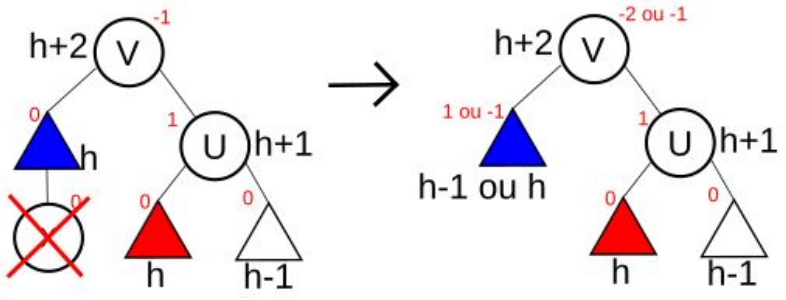


IMAGEM 58: Caso o fator de balanceamento de U seja 1.

Para que o fator de balanceamento de U seja 1, é necessário que a subárvore vermelha tenha altura h e a branca tenha altura h-1. Isso é mostrado no desenho. Além disso, é possível notar que ao remover o nó da subárvore azul, pode ser que a altura dela seja alterada ou não. Caso a altura dela seja alterada, o fator de balanceamento dela será h-1, caso contrário a altura continuará h. Vale ressaltar que quando a altura da árvore é alterada, pode ser necessário rotacionar a árvore. Assim sendo, para que seja possível apresentar os casos de rotação, será considerado que a remoção do nó x alterou a altura da subárvore azul de h para h-1. Com base nisso, é possível obter a seguinte árvore:

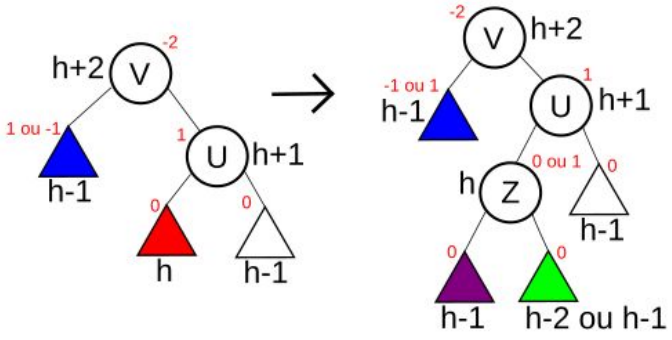


IMAGEM 59: Análise do nó Z.

Para que seja possível analisar melhor a subárvore vermelha, ela foi expandida. Nesse caso, o nó Z representa a raiz da subárvore vermelha e as subárvores roxa e verde representam as subárvores esquerda e direita de Z respectivamente.

Como, após a remoção, a altura de Z é h, é possível concluir que pelo menos uma das subárvores de Z deve ter altura h-1, e a outra pode ter altura h-2 ou h-1. Com base nessa variação possível do fator de balanceamento de Z, é possível

compreender que ele pode ser 1, -1 ou 0 dependendo das alturas das subárvores de Z. Abaixo será feita a análise de cada caso. Inicialmente será considerado o caso em que o fator de balanceamento de Z é 1.

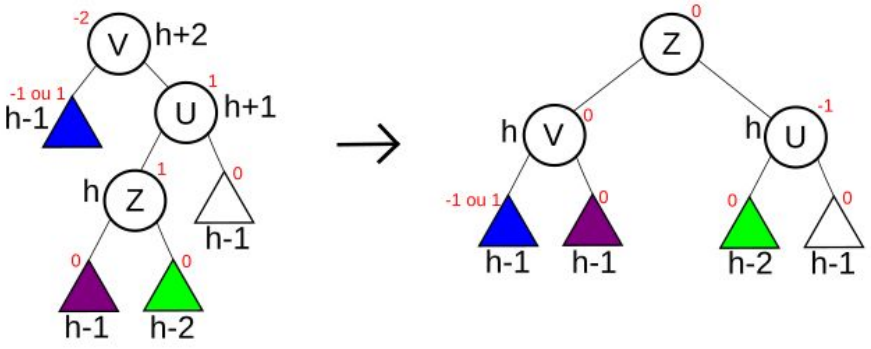


IMAGEM 60: Caso o fator de balanceamento de Z seja 1.

Como é possível perceber na imagem acima, para que o fator de balanceamento de Z seja 1, é necessário que a altura da subárvore roxa seja h-1, e a altura da subárvore verde seja h-2. Ao realizar a rotação dupla à esquerda para esse caso, é possível observar que o fator de balanceamento de V é 0 e o fator de balanceamento de U é -1.

```
if (z->fb == 1)
{
    v->fb = 0;
    u->fb = -1;
    z->fb = 0;
}
```

IMAGEM 61: Implementação do código para o caso do fator de balanceamento de Z ser 1.

No código apresentado, é verificado se o fator de balanceamento de Z é 1, e são feitas as devidas alterações nos fatores de balanceamento de U e de V. Além disso, o fator de balanceamento de Z se torna zero de acordo com o desenho.

Outro caso acontece quando o fator de balanceamento de Z é -1. Esse caso foi apresentado na imagem abaixo.

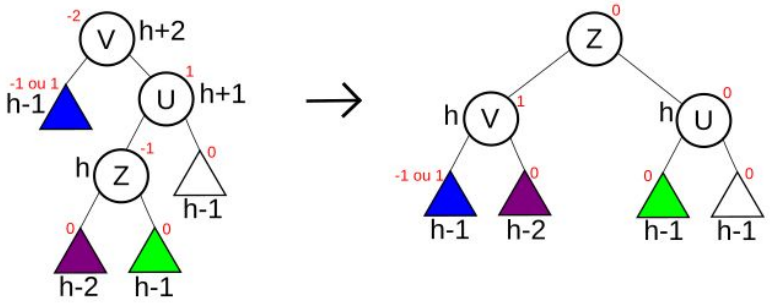


IMAGEM 62: Caso o fator de balanceamento de Z seja -1.

De acordo com o desenho, é possível perceber que o fator de balanceamento de Z é -1 pois a altura da subárvore roxa é h-2 e a altura da subárvore verde é h-1. Além disso, é possível perceber que após a rotação, quando o fator de balanceamento de Z é -1, o fator de balanceamento de V é 1 e o de U é 0.

```
else if (z->fb == -1)
{
    v->fb = 1;
    u->fb = 0;
    z->fb = 0;
}
```

IMAGEM 63: Implementação do código para o caso do fator de balanceamento de Z ser -1.

Como é possível perceber no código acima, foi verificado se o fator de balanceamento de Z é -1, e depois foram feitas as devidas alterações no fator de balanceamento de U, V e Z para esse caso.

O último caso ocorre quando o fator de balanceamento de Z for 0. Esse caso é ilustrado a seguir.

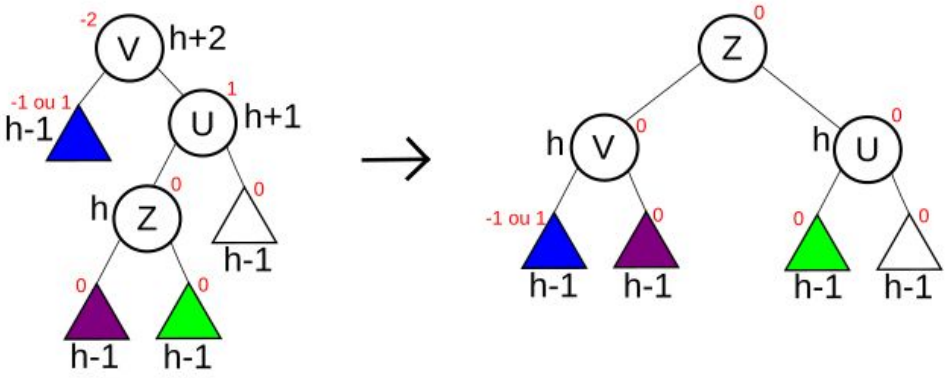


IMAGEM 64: Caso o fator de balanceamento de Z seja 0.

De acordo com a imagem, o fator de balanceamento de Z é 0 quando a altura das subárvores roxa e verde são h-1. Além disso, é possível observar que, para esse caso, após realizar a rotação dupla à esquerda, os fatores de balanceamento de U e de V são 0.

```
else
{
    v->fb = 0;
    u->fb = 0;
    z->fb = 0;
}
```

IMAGEM 65: Implementação do código para o caso do fator de balanceamento de U e V serem 0.

O else apresentado na imagem está após os dois ifs que verificam se o fator de balanceamento de Z é 1 ou -1. Caso ele não seja nenhum desses casos, entra no else, trata o caso que está sendo analisado agora e define o fator de balanceamento de U, V e Z para 0.

Ao analisar todos os casos, é possível perceber que o processo de rotação é o mesmo para todos os casos, e ele consiste apenas em algumas atualizações de ponteiros. Essas atualizações são: o filho à esquerda de U recebe o filho à direita de Z, o filho à direita de Z recebe U, o filho à direita de V recebe o filho à esquerda de Z e o filho à esquerda de Z recebe V. Além disso, ao analisar os casos é possível perceber que o fator de balanceamento de Z é sempre 0 depois de realizar a rotação e a nova raiz da árvore depois da remoção é Z.

```
else //Rotação dupla à esquerda
{
    //Filho à esquerda do filho à direita do nó removido
    z = u->esq;

    //Realizando os reapontamentos
    u->esq = z->dir;
    z->dir = u;
    v->dir = z->esq;
    z->esq = v;

    //Atualizando os fatores de balanceamento para cada caso
    if (z->fb == 1)
    {
        v->fb = 0;
        u->fb = -1;
        z->fb = 0;
    }
    else if (z->fb == -1)
    {
        v->fb = 1;
        u->fb = 0;
        z->fb = 0;
    }
    else
    {
        v->fb = 0;
        u->fb = 0;
        z->fb = 0;
    }

    //Atualizando a raiz da árvore
    v = z;
}
```

IMAGEM 66: Trecho de código que trata das rotações duplas à esquerda para a remoção.

Nesse código, o primeiro else está relacionado com o if que verifica se é um caso de rotação simples. Caso não for, entra no caso atual e é realizada a rotação dupla. Depois disso, é definida a variável Z como sendo o filho à esquerda de U, e os fatores de balanceamento de U e V são alterados de acordo com o fator de balanceamento de Z. Além disso, em todos os casos o fator de balanceamento de Z é alterado para 0 após a rotação. Por fim, a nova raiz da árvore se torna o nó Z.

4.4.3 Busca chave

A função “buscaChave” é utilizada para buscar uma chave na árvore. Ela verifica se a árvore está vazia e caso não esteja, a função irá começar a verificação através da função de callback “compara” e de recursão para encontrar a chave. O primeiro condicional é usado para comparar se o id da chave a ser buscada é menor do que o da chave atual, se for, chama novamente a função “buscaChave” mas agora percorrendo a subárvore da esquerda, o mesmo acontece com o condicional para percorrer a subárvore da direita, porém agora na função “compara” a chave deverá ser maior do que a chave atual. Com isso, a função irá percorrendo a árvore até encontrar a chave. Quando essa chave for encontrada, o retorno será a própria chave. Caso a árvore esteja vazia ou a chave não seja encontrada o retorno será NULL. A imagem abaixo demonstra como foi implementada a função.

```c
//Função recursiva que busca uma chave na árvore
void *buscaChave(AVL *a, void *chave, int (*compara)(void *, void *))
{
    if (a) //Caso a árvore não esteja vazia
    {
        if (compara(chave, a->chave) == -1) //Se a chave buscada for menor do que a chave atual
            //Realiza a busca na subárvore da esquerda
            return buscaChave(a->esq, chave, compara);
        else if (compara(chave, a->chave) == 1) //Se a chave buscada for maior do que a chave atual
            //Realiza a busca na subárvore da direita
            return buscaChave(a->dir, chave, compara);
        else //Caso a chave for encontrada
            //Retorna a chave encontrada
            return a->chave;
    }
    else //Se a árvore está vazia ou o elemento não está na árvore
        return NULL;
}
```

IMAGEM 67: Implementação da função buscaChave.

Essa função de busca foi utilizada no menu para procurar uma chave. As imagens abaixo mostram como o menu utiliza ela para verificar se uma chave foi encontrada ou não.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 3

A opcao selecionada foi: Procurar mesa.

Digite a chave da mesa a ser buscada: 1

Chave: 1
Altura: 1.00
Largura: 1.00
```

IMAGEM 68: Saída do programa caso a chave seja encontrada.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 3

A opcao selecionada foi: Procurar mesa.

Digite a chave da mesa a ser buscada: 2

A mesa buscada nao foi encontrada.
```

IMAGEM 69: Saída do programa caso a chave não seja encontrada.

4.4.4 Libera Árvore

A função “liberaArvore” é usada para liberar a memória alocada para cada nó de uma árvore e de suas respectivas chaves. A função irá verificar se a árvore está vazia e caso não esteja irá realizar recursão para que ocorra a liberação de memória nó por nó. Assim, primeiramente a recursão será com a subárvore à esquerda, depois para subárvore à direita, assim percorrendo toda a árvore. Depois disso, será utilizada a função de callback “liberaChave” para liberar a memória alocada para cada chave de cada nó da árvore. O comando utilizado para liberar a memória alocada para o nó é o comando free( ).

```c
//Função recursiva que libera a memória alocada para cada nó da árvore e suas respectivas chaves
void liberaArvore(AVL *a, void (*liberaChave)(void *))
{
    if (a) //Caso a árvore não esteja vazia
    {
        //Chama a função recursivamente para liberar a memória na subárvore esquerda
        liberaArvore(a->esq, liberaChave);
        //Chama a função recursivamente para liberar a memória na subárvore direita
        liberaArvore(a->dir, liberaChave);
        //Utiliza a função de callback para liberar a memória alocada para a chave
        liberaChave(a->chave);
        //Libera a memória alocada para o nó da árvore.
        free(a);
    }
}
```

IMAGEM 70: Implementação da função liberaArvore.

4.4.5 Imprime Árvore

A função "imprime Árvore" é usada para imprimir cada nó de uma árvore e o fator de balanceamento dele. Caso a árvore não esteja vazia, é aplicada a tabulação por nível do nó através de um print e um for. Depois disso, é chamada a função

recursivamente para imprimir a subárvore da esquerda, depois a callback “imprimeChave” será chamada para imprimir a chave daquele nó e o fator de balanceamento dela após os dois pontos. Depois disso, ela é chamada de forma recursiva para imprimir a subárvore da direita e imprimir todas as chaves da árvore. Por fim, a impressão ocorrerá de forma simétrica. O código está na imagem abaixo.

```c
/*Função recursiva que realiza a impressão de cada nó da árvore e do seu fator de balanceamento
aplicando a tabulação de acordo com o nível de cada nó*/
void imprimeArvore(AVL *a, int cont, void (*imprimeChave)(void *))
{
    if (a) //Caso a árvore não esteja vazia
    {

        //Chama a função recursivamente para imprimir a subarvore esquerda
        imprimeArvore(a->esq, cont + 1, imprimeChave);

        //Realiza a tabulação de acordo com o nível
        for (int i = 0; i < cont; i++)
            printf("\t");

        //Utiliza a função de callback para imprimir a chave
        imprimeChave(a->chave);
        //Imprime o fator de balanceamento
        printf(":%d\n", a->fb);

        //Chama a função recursivamente para imprimir a subarvore direita
        imprimeArvore(a->dir, cont + 1, imprimeChave);
    }
}
```

IMAGEM 71: Implementação da função imprimeArvore.

A função imprimeChave é utilizada pelo menu para realizar a impressão da árvore. As imagens a seguir mostram como os casos de árvore vazia ou não são tratados.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 4

A opcao selecionada foi: Imprimir arvore.

A arvore esta vazia.
```

IMAGEM 72: Saída do programa para a opção imprimir caso a árvore esteja vazia.

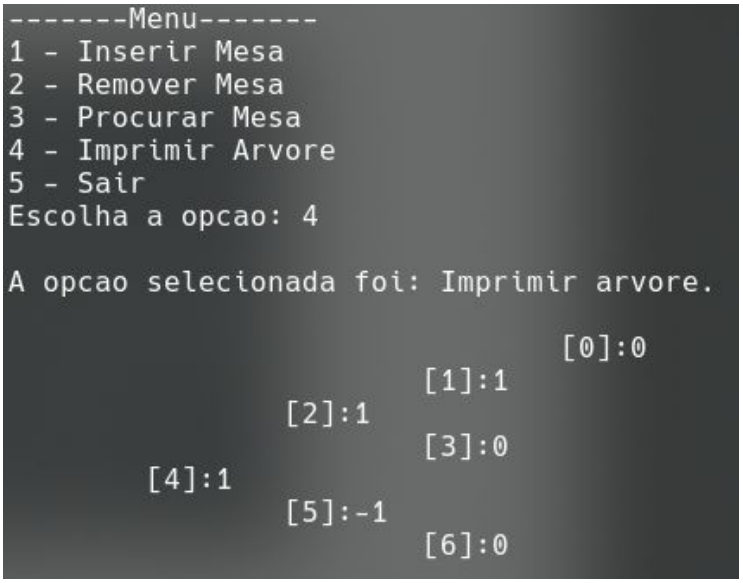


IMAGEM 73: Saída do programa para a opção imprimir.

4.4.6 Inserir

A função “insereAVL” é usada para realizar as inserções na árvore. Ela inicia com um condicional para verificar se a árvore está vazia. Caso esteja, ela realiza a alocação de um novo nó através da função criaNo, define que a altura foi alterada e retorna o novo nó criado.

Caso essa árvore não esteja vazia, primeiro irá passar por um condicional que verifica (através da função de callback “compara”) se a chave a ser inserida já está nessa árvore. Caso esteja, é utilizada a função de callback “liberaChave” para liberar a memória que havia sido alocada para essa chave e ela não é inserida para evitar chaves repetidas na árvore. Depois disso, são realizados os condicionais para verificar se é necessário percorrer a subaŕvore direita ou esquerda e inserir o novo nó no local correto e já de forma balanceada.

O primeiro condicional para essa situação utiliza a função de callback “compara”. Caso a nova chave seja menor do que a chave do elemento atual, chama a função “insereAVL” mas agora com a subárvore da esquerda. Após realizar a inserção na subárvore esquerda, pode ser que tenha ocorrido mudança na altura dessa árvore, então também tratamos os casos para alteração do fator de balanceamento através de um condicional que verifica se a altura não foi alterada (h = 0) seguido de um switch case para tratar cada caso do fator de balanceamento caso a altura tenha sido alterada (h = 1). O switch possui os seguintes casos.

No primeiro caso, se o fator de balanceamento de V (que é o ponteiro da árvore que foi passada para a função) for igual a -1, o fator de balanceamento passa a ser zero e a variável que controla se a altura foi alterada ou não também. Com isso, acontece o seguinte caso:

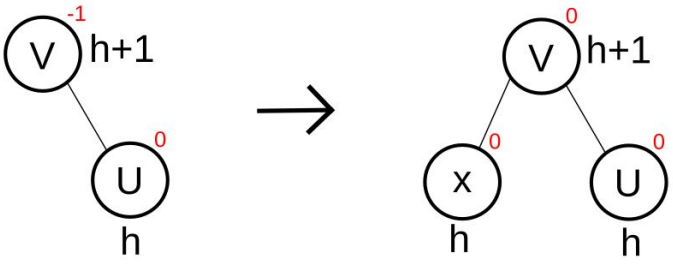


IMAGEM 74: Caso o fator de balanceamento de V seja -1.

Como é possível observar, o fator de balanceamento de V era -1 e depois da inserção do nó x, na subárvore esquerda, ele passa a ser 0 e não há alteração na altura do nó V. Esse é o primeiro caso tratado no código, e o código dele está abaixo.

```
case -1:
    v->fb = 0;
    //Inserção nesse caso não altera a altura
    *h = 0;
    break;
```

IMAGEM 75: Trecho do código para o caso do fator de balanceamento de Z ser -1.

Outro caso acontece quando o fator de balanceamento de V for igual a 0. Nesse caso, o fator de balanceamento passa a ser 1 e não haverá mudança na altura. Nesse caso, acontece a seguinte situação:

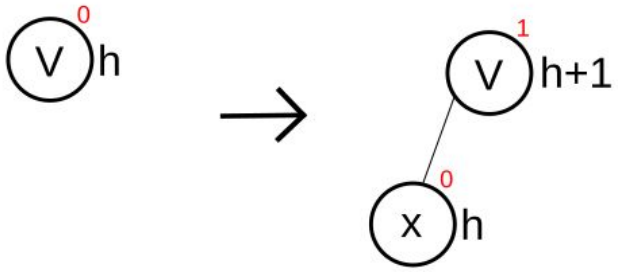


IMAGEM 76: Caso o fator de balanceamento de V seja 0.

Como é possível perceber, a inserção provocou a alteração na altura, mas não houve nenhum desbalanceamento. Como a altura já tinha sido alterada antes, o “h” não precisa ser alterado pois ele já é 1. O código desse caso está abaixo.

```
case 0:
    v->fb = 1;
    break;
```

IMAGEM 77: Trecho do código para caso a inserção não tenha alterado a altura.

No último caso, quando o fator de balanceamento for 1 existe a necessidade de uma rotação, então aqui é chamada a função "rodaDireita" tendo como parâmetro a árvore e a variável que controla se a altura foi alterada ou não. Um exemplo desse caso é ilustrado a seguir.

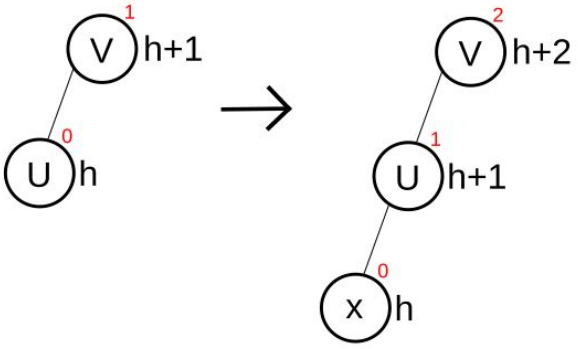


IMAGEM 78: Caso o fator de balanceamento de V seja 1.

Como é possível perceber, houve um desbalanceamento e foi necessário realizar uma rotação para a direita. Esse é o último caso e o código dele está abaixo.

```
case 1:
    //Houve necessidade de rotação
    v = rodaDireita(v, h);
    break;
```

IMAGEM 79: Trecho do código para o caso de precisar ocorrer uma rotação.

Uma situação análoga acontece quando a inserção é na subárvore da direita, mas, nesse caso, na função “compara” será verificado se a nova chave é maior do que o elemento atual. Além disso, será feita a mesma verificação se a altura da árvore foi alterada e se for, será executado o switch case. Entretanto, aqui o switch case terá outros casos.

O primeiro caso é quando o fator de balanceamento de V é 1. Nesse caso, o fator e a altura passam a ser zero. Isso pode ser observado no exemplo e no código abaixo:

IMAGEM 80: Caso o fator de balanceamento de V seja 1.

```
case 1:
    v->fb = 0;
    //Nesse caso não houve alteração na altura
    *h = 0;
    break;
```

IMAGE 81: Trecho do código para o caso de o fator de balanceamento de V ser igual a 1.

Outro caso acontece se o fator de balanceamento for 0. Se isso ocorrer, ele vai passar a ser -1. Isso é exemplificado no desenho e no código abaixo.

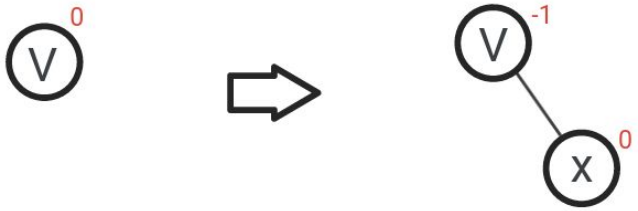


IMAGEM 82: Caso o fator de balanceamento de v seja igual a 0.

```
case 0:
    v->fb = -1;
    break;
```

IMAGEM 83: Trecho do código para o caso o fator de balanceamento de v seja igual a 0.

Por fim, se o fator de balanceamento de V for -1 irá acontecer a rotação através da função “rodaEsquerda”, que recebe como parâmetro o nó da árvore e a variável que contabiliza se a altura foi alterada ou não. Esse caso é exemplificado no exemplo e no código abaixo.

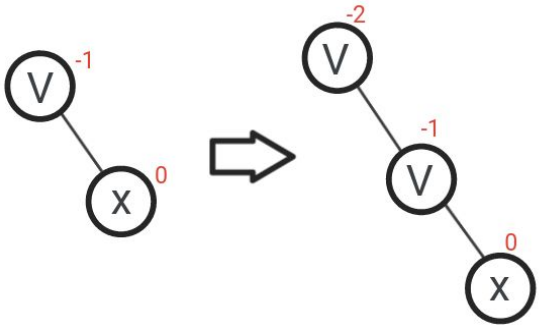


IMAGEM 84: Caso o fator de balanceamento de V seja -1.

```
case -1:
    //Houve necessidade de rotação
    v = rodaEsquerda(v, h);
    break;
```

IMAGEM 85: Trecho do código para o caso do fator de balanceamento de V ser -1.

Após inserir o elemento, a função irá retornar um ponteiro para a nova raiz da árvore. A implementação completa da função “insereAVL” é apresentada a seguir.

```
//Função recursiva que realiza a inserção em uma árvore AVL mantendo o balanceamento
AVL *insereAVL(AVL *v, int *h, void *chave, int (*compara)(void *, void *), void (*liberaChave)(void *))
{
    if (!v) //Caso a árvore esteja vazia
    {
        //Aloca memória para um novo nó
        v = criaNo(chave);
        //Indica que a altura foi alterada
        *h = 1;
    }
```

IMAGEM 86.1: Implementação da função insereAVL.

```
    else //Caso a árvore não seja vazia
    {
        if (compara(chave, v->chave) == 0) //Se o elemento a ser inserido já existe na árvore
        {
            //Libera a memória alocada para o elemento
            liberaChave(chave);
            return v;
        }
        if (compara(chave, v->chave) == -1) //Se o elemento a ser inserido é menor do que o elemento atual
        {
            //Chama a inserção recursivamente na subarvore esquerda
            v->esq = insereAVL(v->esq, h, chave, compara, liberaChave);

            if (*h != 0) //Caso haja alteração na altura durante a inserção
            {
                switch (v->fb) //Altera o fator de balanceamento de acordo com cada caso
                {
                case -1:
                    v->fb = 0;
                    //Inserção nesse caso não altera a altura
                    *h = 0;
                    break;

                case 0:
                    v->fb = 1;
                    break;

                case 1:
                    //Houve necessidade de rotação
                    v = rodaDireita(v, h);
                    break;

                default:
                    break;
                }
            }
        }
```

IMAGEM 86.2: Implementação da função insereAVL.

```
        else //Se o elemento a ser inserido é maior do que o elemento atual
        {
            //Chama a função de inserção recursivamente na subarvore direita
            v->dir = insereAVL(v->dir, h, chave, compara, liberaChave);

            if (*h != 0) //Caso ocorra alteração na altura após a inserção
            {
                switch (v->fb) //Altera o fator de balanceamento de acordo com cada caso
                {
                case 1:
                    v->fb = 0;
                    //Nesse caso não houve alteração na altura
                    *h = 0;
                    break;

                case 0:
                    v->fb = -1;
                    break;

                case -1:
                    //Houve necessidade de rotação
                    v = rodaEsquerda(v, h);
                    break;

                default:
                    break;
                }
            }
        }
    }
    return v;
}
```

IMAGEM 86.3: Implementação da função insereAVL.

A função “insereAVL” também é utilizada no menu e as imagens abaixo demonstram como são tratados os casos de inserir um elemento que não existe, e inserir um elemento já existente na árvore.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 1

A opcao selecionada foi: Inserir mesa.

Digite a chave da nova mesa: 1
Digite a altura da nova mesa: 1
Digite a largura da nova mesa: 1

A seguinte mesa foi cadastrada:

Chave: 1
Altura: 1.00
Largura: 1.00
```

IMAGEM 87: Saída do programa caso a opção escolhida seja inserir.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 1

A opcao selecionada foi: Inserir mesa.

Digite a chave da nova mesa: 1
Digite a altura da nova mesa: 1
Digite a largura da nova mesa: 1

Nao foi possivel cadastrar. A mesa com chave 1 ja foi cadastrada.
```

IMAGEM 88: Saída do programa caso a chave inserida já exista na árvore

4.4.7 Remover

A função remover é a função utilizada para remover um nó da árvore. A primeira verificação a ser realizada nessa função é se a árvore está vazia. Caso ela esteja, é indicado que a altura não sofreu alterações (h=0) e a função retorna NULL, isso também acontece para o caso do elemento a ser removido não ser encontrado. Se a árvore não estiver vazia, começam então os condicionais para que ocorra a busca para a remoção. Para saber se a chave a ser encontrada é maior ou menor do que a chave atual ou se ela já foi encontrada foram utilizadas três condicionais (if, else if e else) que utilizam a função callback “compara”.

A primeira condicional verifica se a chave a ser removida é maior do que a chave atual. Se isso acontecer, a função de remoção será chamada percorrendo a subárvore da direita. Depois de realizar a remoção na subárvore direita, é verificado se a altura da árvore sofreu alguma alteração e, dependendo do caso, a função realiza a alteração do fator de balanceamento ou realiza uma rotação.

A verificação de cada caso é feita através de um switch case. Nessa estrutura, o primeiro caso é quando o fator de balanceamento for -1. Esse caso é ilustrado a seguir:

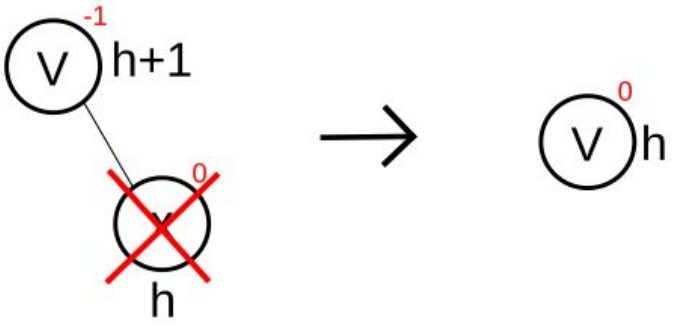


IMAGEM 89: Caso o fator de balanceamento seja -1.

.

Como é possível perceber, a altura foi alterada com essa remoção, então a variável que contabiliza se a altura foi alterada ou não vai receber 1 para indicar que ela foi alterada. Além disso, o fator de balanceamento de V será 0. A imagem a seguir demonstra o código para tratar esse caso.

```
case -1:
    a->fb = 0;
    //Houve alteração na altura
    *h = 1;
    break;
```

IMAGEM 90: Trecho de código para caso o fator de balanceamento ser -1.

Outro caso que pode acontecer ao remover um nó da subárvore direita é quando o fator de balanceamento da raiz for 0. Esse caso é ilustrado a seguir:

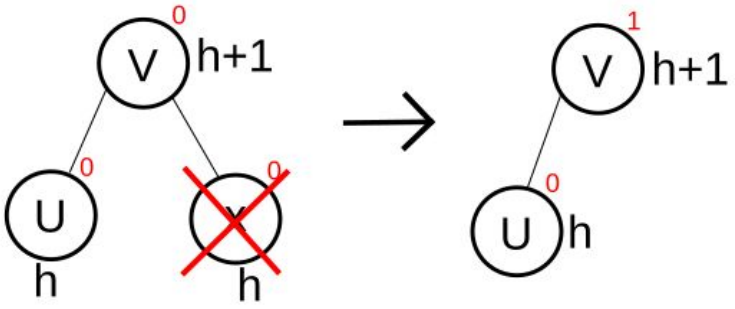


IMAGEM 91: Caso o fator de balanceamento da raiz seja 0.

Como é possível perceber, a altura não sofre alteração ao realizar esse tipo de remoção e o fator de balanceamento da raiz passa a ser 1. A figura a seguir apresenta como esse caso foi tratado no código.

```
case 0:
    a->fb = 1;
    //Não houve alteração na altura
    *h = 0;
    break;
```

IMAGEM 92: Trecho de código para o caso de o fator de balanceamento da raiz ser 0.

Por fim, o último caso é quando o fator de balanceamento da raiz é 1. Nesse caso, vai existir a necessidade de uma rotação através da função “rodaDireitaRemover”. Para essa função, serão passadas a árvore e a variável que

controla se a altura da árvore foi alterada ou não. A imagem a seguir descreve esse caso:

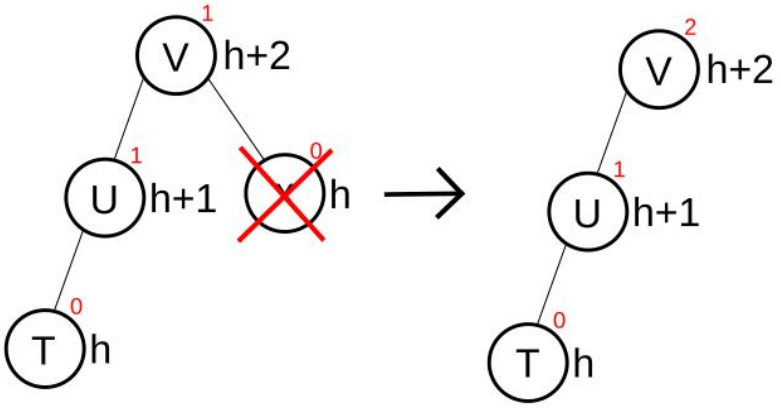


IMAGEM 93: Caso exista a necessidade de uma rotação.

Depois do elemento ser removido (caso ele exista na árvore) e da árvore ser balanceada por esses casos, a função retorna a nova raiz da árvore após o elemento ser removido. A imagem a seguir mostra o código para esse caso.

```
case 1:
    //Houve necessidade de rotação
    a = rodaDireitaRemover(a, h);
    break;
```

IMAGEM 94: Trecho de código para caso o fator de balanceamento da raiz ser 1.

A segunda condicional utiliza a função de callback “compara” para verificar se o elemento a ser removido é menor do que o elemento atual. Se isso acontecer, a função de remoção é chamada novamente na subárvore esquerda.

Depois da remoção ser feita, será verificado se a altura da árvore foi alterada pela remoção. Se isso acontecer, o switch case irá tratar cada caso de remoção na subárvore esquerda.

O primeiro caso acontece se o fator de balanceamento da raiz for 1. Esse caso é ilustrado a seguir:

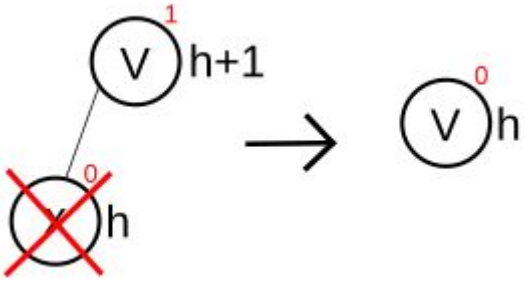


IMAGEM 95: Caso o fator de balanceamento da raiz seja 1.

Nesse caso, ao remover, é possível perceber que houve alteração na altura e o fator de balanceamento passará a ser 0. A imagem a seguir demonstra como esse caso foi tratado no código:

```
case 1:
    a->fb = 0;
    //Houve alteração na altura
    *h = 1;
    break;
```

IMAGEM 96: Trecho de código para o caso do fator de balanceamento da raiz ser 1.

.

Outro caso acontece quando o fator de balanceamento da raiz é 0. Esse caso é ilustrado a seguir:

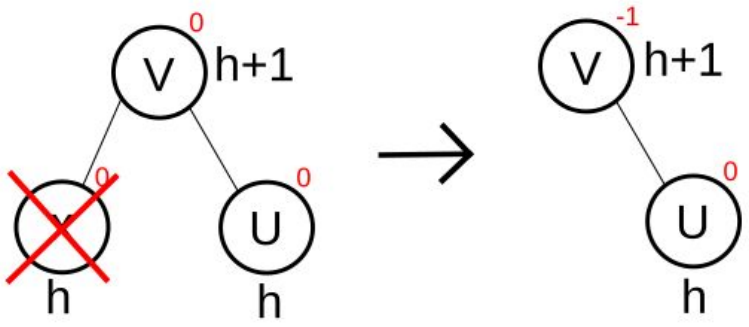


IMAGEM 97: Caso o fator de balanceamento da raiz seja 0.

Como é possível observar no desenho, não houve alteração na altura e o fator de balanceamento da raiz será -1 após a remoção da esquerda. A imagem a seguir mostra como esse caso foi tratado no código:

```
case 0:
    a->fb = -1;
    //Não houve alteração na altura
    *h = 0;
    break;
```

IMAGEM 98: Trecho de código para o caso do fator de balanceamento da raiz ser 0.

Por fim, o último caso ocorre quando o fator de balanceamento é -1. Esse caso foi ilustrado a seguir:

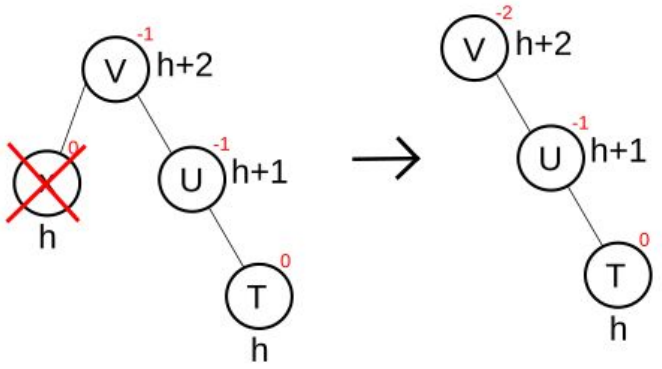


IMAGEM 99:Caso o fator de balanceamento da raíz seja -1.

Como é possível perceber, após a remoção do nó da subárvore esquerda de V, o fator de balanceamento de V é -2. Nesse caso, existe a necessidade de rotação através da função “rodaEsquerdaRemover”. Para essa função serão passadas a árvore e a variável que controla se a altura foi alterada ou não. A imagem a seguir mostra como esse caso foi tratado no código:

```
case -1:
    //Houve necessidade de rotação
    a = rodaEsquerdaRemover(a, h);
    break;
```

IMAGEM 100: Trecho de código para o caso do fator de balanceamento da raíz ser -1.

Depois do elemento ser removido e a árvore balanceada por esses casos, a função retorna a nova raiz da árvore. Através da busca na subárvore esquerda ou direita, a árvore vai sendo percorrida para buscar o elemento até encontrá-lo ou concluir que ele não está na árvore.

Caso o elemento a ser removido seja um nó folha, a memória alocada para a chave dele é liberada através da função callback “liberaChave”, a memória do nó também é liberada através do comando free( ) e a variável que controla se a altura foi alterada ou não recebe 1 para indicar que houve alteração na altura.

Outro caso ocorre quando o elemento possui uma subárvore na esquerda e na direita. Nesse caso, é feita a busca do elemento predecessor daquele nó para que seja possível realizar a troca da chave que foi removida pela chave do seu predecessor. Vale ressaltar que o predecessor é definido como sendo o elemento mais à direita da subárvore esquerda do nó.

Depois que a troca for realizada, a função de remoção é chamada na subárvore esquerda passando agora a chave do predecessor. Assim que o elemento for removido, os fatores de balanceamento serão modificados de acordo com cada caso para recuperar o balanceamento da árvore.

O primeiro caso ocorre quando o fator de balanceamento for 1. Esse caso foi ilustrado abaixo:

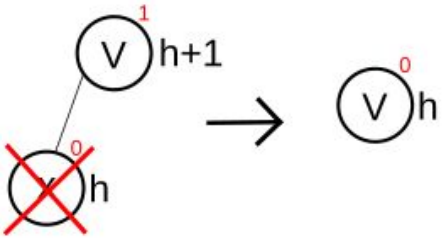


IMAGEM 101: Caso o fator de balanceamento seja 1.

Nesse caso, houve alteração na altura e o fator de balanceamento de V após a remoção é 0.

Outro caso ocorre quando o fator de balanceamento for 0. A imagem abaixo ilustra esse caso.

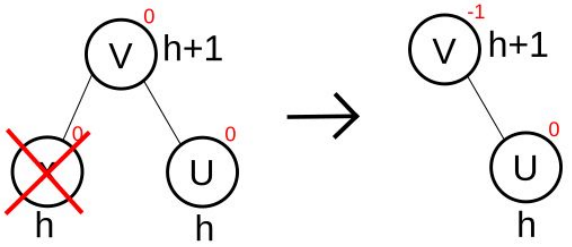


IMAGEM 102: Caso o fator de balanceamento seja 0.

.

Nesse caso, o fator de balanceamento de V passará a ser -1 e não houve alteração na altura.

Por fim, o último caso ocorre quando o fator de balanceamento for -1. Ele foi ilustrado abaixo.

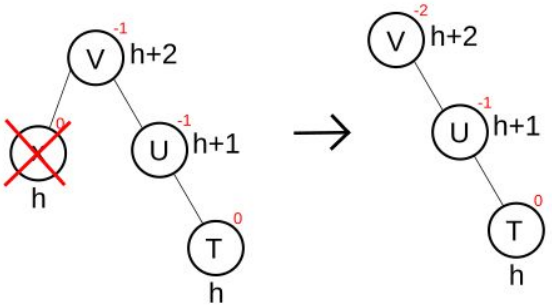


IMAGEM 103: Caso o fator de balanceamento seja -1.

Nesse caso, existe a necessidade de rotação através da função “rodaEsquerdaRemover” pois a árvore foi desbalanceada e o nó V tem fator de balanceamento -2.

Depois da rotação ser feita e do fator de balanceamento ser alterado, a função retorna a nova raiz da árvore.

As outras duas condicionais da função tratam os casos em que a árvore tem apenas a subárvore da direita ou da esquerda. Nesses casos, a variável “proximo” irá armazenar a subárvore que o elemento removido possuir. Depois, será liberada a memória que foi alocada para a chave através da função callback “liberaChave”. Depois disso, será liberada a memória alocada para o nó através do comando free() e a variável que controla se a altura foi alterada ou não recebe 1 para indicar que houve a alteração na altura. Nesses dois casos o retorno será o “proximo” já que ele agora será o nó que ficará no lugar do elemento removido da árvore. Por fim, se a árvore está vazia ou se o elemento não foi encontrado, não vai haver alteração na altura nem na árvore, e o retorno será NULL. O código está nas imagens abaixo.

```c
//Função recursiva que realiza a remoção de um elemento da árvore mantendo o balanceamento
AVL *remover(AVL *a, int *h, void *chave, int (*compara)(void *, void *), void (*liberachave)(void *))
{
    if (a) //Se a árvore não está vazia
    {
        if (compara(chave, a->chave) > 0) //Se a chave a ser removida é maior do que a chave atual
        {
            //Chama a função de remoção recursivamente na subarvore direita
            a->dir = remover(a->dir, h, chave, compara, liberachave);

            if (*h != 0) //Caso a altura da árvore seja alterada após a remoção
            {
                switch (a->fb) //Altera o fator de balanceamento de acordo com cada caso
                {
                case -1:
                    a->fb = 0;
                    //Houve alteração na altura
                    *h = 1;
                    break;

                case 0:
                    a->fb = 1;
                    //Não houve alteração na altura
                    *h = 0;
                    break;

                case 1:
                    //Houve necessidade de rotação
                    a = rodaDireitaRemover(a, h);
                    break;

                default:
                    break;
                }
            }
            return a;
        }
```

IMAGEM 104.1: Implementação da função remover.

```c
        else if (compara(chave, a->chave) < 0) //Se o elemento a ser removido é menor do que a chave atual
        {
            //Chama a função de remoção recursivamente na subarvore esquerda
            a->esq = remover(a->esq, h, chave, compara, liberachave);

            if (*h != 0) //Se houve alteração na altura após a remoção
            {
                switch (a->fb) //Altera o fator de balanceamento de acordo com cada caso
                {
                case 1:
                    a->fb = 0;
                    //Houve alteração na altura
                    *h = 1;
                    break;

                case 0:
                    a->fb = -1;
                    //Não houve alteração na altura
                    *h = 0;
                    break;

                case -1:
                    //Houve necessidade de rotação
                    a = rodaEsquerdaRemover(a, h);
                    break;

                default:
                    break;
                }
            }

            return a;
        }
```

IMAGEM 104.2: Implementação da função remover.

```
{
    if (!a->dir && !a->esq) //Se o elemento for uma folha
    {
        //Libera a memória alocada para a chave
        liberachave(a->chave);
        //Libera a memória alocada para o nó
        free(a);
        //Indica que houve alteração na altura
        *h = 1;
        //Retorna NULL para a chamada recursiva anterior
        return NULL;
    }
    else if (a->dir && a->esq) //Se o elemento tiver a subarvore esquerda e a direita
    {
        //Buscando o predecessor (elemento mais a direita da subarvore esquerda)
        AVL *predecessor = a->esq;

        while (predecessor->dir)
            predecessor = predecessor->dir;

        //Realizando a troca da chave a ser removida pela chave do predecessor
        void *aux = predecessor->chave;
        predecessor->chave = a->chave;
        a->chave = aux;

        //Chama a função recursivamente na subarvore esquerda
        a->esq = remover(a->esq, h, predecessor->chave, compara, liberachave);
```

IMAGEM 104.3: Implementação da função remover.

```
        if (*h != 0) //Caso haja alteração na altura após a remoção
        {
            switch (a->fb) //Altera o fator de balanceamento de acordo com cada caso
            {
            case 1:
                a->fb = 0;
                //Houve alteração na altura
                *h = 1;
                break;

            case 0:
                a->fb = -1;
                //Não houve alteração na altura
                *h = 0;
                break;

            case -1:
                //Houve necessidade de rotação
                a = rodaEsquerdaRemover(a, h);
                break;

            default:
                break;
            }
        }
        return a;
    }
```

IMAGEM 104.4: Implementação da função remover.

```
    else if (a->dir) //Se a árvore tem apenas subarvore direita
    {
        AVL *proximo = a->dir;
        //Libera a memória alocada para a chave
        liberachave(a->chave);
        //Libera a memória alocada para o nó
        free(a);
        //Indica que houve alteração na altura
        *h = 1;
        return proximo;
    }
```

IMAGEM 104.5: Implementação da função remover.

```
            }
            else //Se a árvore tem apenas subarvore esquerda
            {
                AVL *proximo = a->esq;
                //Libera a memória alocada para a chave
                liberachave(a->chave);
                //Libera a memória alocada para o nó
                free(a);
                //Indica que a altura foi alterada
                *h = 1;
                return proximo;
            }
        }
    }
    else //Se a árvore está vazia ou não encontrou o elemento
    {
        //Não houve alteração na árvore nem na altura
        *h = 0;
        return NULL;
    }
}
```

IMAGEM 104.6: Implementação da função remover.

A função de remoção também foi utilizada no menu para remover um elemento da árvore. As imagens abaixo ilustram como ela foi utilizada e como foram tratados os casos da mesa ser removida ou não ser encontrada.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 2

A opcao selecionada foi: Remover mesa.

Digite a chave da mesa a ser removida: 1

A seguinte mesa foi removida:

Chave: 1
Altura: 1.00
Largura: 1.00
```

IMAGEM 105: Saída do programa caso seja escolhida a função remover.

```
-------Menu-------
1 - Inserir Mesa
2 - Remover Mesa
3 - Procurar Mesa
4 - Imprimir Arvore
5 - Sair
Escolha a opcao: 2

A opcao selecionada foi: Remover mesa.

Digite a chave da mesa a ser removida: 2

Nao foi possivel remover. A mesa com chave 2 nao foi encontrada.
```

IMAGEM 106: Saída do programa caso não seja possível remover o elemento.

5 CONCLUSÃO

Portanto, é possível concluir que o que permite que a árvore AVL apresente um bom desempenho constante nas operações de busca, inserção e remoção é o fato dela ser balanceada. Entretanto, para garantir o balanceamento dela é preciso realizar algumas operações de rotação e alterar os fatores de balanceamento para cada caso como foi descrito no decorrer do trabalho. Além disso, vale ressaltar que os apontamentos feitos para uma rotação à direita da inserção são os mesmos apontamentos feitos para a rotação à direita da remoção. O mesmo acontece para a rotação à esquerda. Porém, como optou-se por alterar os fatores de balanceamento dentro das funções de rotação, foi necessário criar funções separadas de rotação para a inserção e para a remoção pois os fatores de balanceamento e os casos de rotação são um pouco diferentes na inserção e remoção, o que impossibilita de tratar tudo em uma única função que abrange tanto as rotações da inserção e remoção e as alterações nos fatores de balanceamento para cada caso.

6 REFERÊNCIAS

https://pt.wikipedia.org/wiki/%C3%81rvore_AVL#Rota%C3%A7%C3%A3o
https://docs.microsoft.com/pt-br/cpp/cpp/header-files-cpp?view=msvc-160
https://stackoverflow.com/questions/19278236/avl-tree-balance
https://www.youtube.com/watch?v=ZK4LgQELy9U&ab_channel=MarceloManzato
https://pt.wikipedia.org/wiki/C_(linguagem_de_programa%C3%A7%C3%A3o)