UNIVERSIDADE DE BRASÍLIA
Instituto de Exatas
Departamento de Ciência da Computação

Disciplina 116319 - Estrutura de Dados
Professor Marcos Caetano

Alunos:

Daniel Carvalho Moreira	16/0116821
João Victor Cabral de Melo	16/0127670
Vinícius Bowen	18/0079239

Brasília, 14 de Junho de 2018

1.1-Árvores AVL

A árvore AVL consiste basicamente em ser composta de uma árvore de busca binária, porém, com propriedades adicionais que permitem que ela seja balanceada a cada interação de nó incluído ou excluído.

Por meio do Fator de balanceamento, a árvore conta com um índice criado em sua struct, quantos nós o lado esquerdo e direito possuem, a não ser que seja um nó do tipo folha(esquerda e direita equivalentes a NULL). Cada nó possui um fator de balanceamento. Esse nó de balanceamento não deve ser inferior a -1 nem superior a 1, pois valores que ultrapassem essa margem já estão nos parâmetros de uma árvore desbalanceada.

Caso estejam desbalanceadas, a correção é feita por meio de rotações, onde consiste em 4 casos(mais detalhes no slide), podendo ser elas: Rotação simples à esquerda, rotação simples à direita, rotação dupla à esquerda e Rotação dupla à direita, deste modo, fazendo com que a árvore fique balanceada.

1.2-Algoritmo

1.2.1-Struct

```c
typedef struct elemento {

    int dado;//dado da estrutura
    int fator_de_balanceamento;//fator de balanceamento da arvore
    struct elemento *esq;//estrutura elemento pra esquerda
    struct elemento *dir;//estrutura elemento pra direita

}elemento;
```

Figura 1: Struct do algoritmo

A struct utilizada tem como variáveis inteiras: “int dado” e “int fator_de_balanceamento”,para que possam armazenar o valor na árvore e verificar sua altura, respectivamente, além dos ponteiros para a direita e esquerda do tipo “struct elemento*”.

1.2.2- Alocação dinâmica.

```c
//Aloca o no para inserir
elemento *aloca_no(int dado){
    elemento *novo_no               = (elemento *) malloc(sizeof(elemento)); //Aloca novo no
    novo_no->dado                   = dado;//recebe o dado
    novo_no->fator_de_balanceamento = 0;//fator de balanceamento do no
    novo_no->esq                    = NULL;// no esquerda e setado pra NULL
    novo_no->dir                    = NULL;// no direito e setado pra NULL
    return novo_no;//retorna no
}
//Aloca ponteiro de ponteiro pra arvore
Arv_AVL *aloca_arvore(){
    Arv_AVL *raiz = (Arv_AVL *) malloc(sizeof(Arv_AVL));//faz a alocação
    if(raiz != NULL){//se a raiz não tiver nula
        *raiz = NULL;//então coloca ela como nula
    }
    return raiz;//retorna a raiz
}
```

Figura 2: funções para alocação de raiz e nós da árvore;

Funções que, ao serem chamadas, efetuam a alocação dinâmica e alocam uma região de memória para que seus valores sejam alterados. Inicialmente os valores são pré-definidos como NULL para que possam ser alterados dentro de outra função.

1.2.3-Liberação de memória

```c
//percorre a arvore e libera cada no
void libera_no(elemento *no){

    if(no == NULL){
        return;//se o no for NULL não faça nada
    }

    libera_no(no->esq);//anda pos-ordem na arvore
    libera_no(no->dir);//anda em pos-ordem na arvore
    free(no);//da free no no

    no = NULL;// coloca null no pra evitar verificações posteriores

}


//libera o ponteiro de ponteiro pra raiz
void libera_arvore(Arv_AVL *raiz){

    if(raiz == NULL){//verifica se raiz não e vazia
        return;
    }
    libera_no(*raiz);//passo a raiz para o libera os no para percorrer a arvore e liberar os nos
    free(raiz);//dou ponteiro de ponteiro da raiz
}
```

Figura 3: Funções para liberar nós e a raiz

As duas funções tem a finalidade de evitar que haja um vazamento de memória no programa. Quando chamadas, a função “libera_no”, libera nós a partir das folhas até chegar na raiz.Quando a função “Libera_raiz” é chamada, a raiz é verificada, se obter algo diferende de NULL, a funcionalidade “free()” é aplicada a ela, assim como na função que libera a memória dos nós. Com isso, os elementos são excluídos e a árvore ocupa um espaço gradativamente menor a cada vez que uma dessas funções é executada.

1.2.4-Altura

```c
//altura do no
int altura(elemento *no){//calcula a altura do no

    if (no == NULL){//se chegar no maximo da direita ou da esquerda retorna zero e vai somando mais 1 a cada recursão
        return 0;
    }

    int alt_esq = altura(no->esq);//vai ao máximo da esquerda do no
    int alt_dir = altura(no->dir);//vai ao máximo da direita do no

    if(alt_esq > alt_dir){  //verifica qual altura da subarvore e maior e acrescente um
        return alt_esq + 1;
    }
    else {
        return alt_dir + 1;//calcula maior altura do no da arvore
    }

}
```

Figura 4:Altura da árvore.

A altura da árvore é medida por meio do retorno do método recursivo chamado dentro da própria função.Deste modo, à medida que o ponteiro retorna um nível na recursão, uma unidade é acrescentada a uma das duas variáveis. No caso da figura 4, a altura da sub-árvore da direita é a condição de parada para que a sua altura seja incrementada.

1.2.5-Fator de balanceamento

```c
//calcula o fator de balanceamento do no
int calcula_fat_bal(elemento *no){

    int fat_bal = altura(no->esq) - altura(no->dir);//usa a definicação do fator de balanceamento do no

    no->fator_de_balanceamento = fat_bal;//passa pra estrutura do no

    return fat_bal;

}


int fat_bal_no(elemento *no){//retorna o fator de balanceamento do no

    if(no == NULL){//se o no for null retorna -1

        return -1;

    }
    else{

        return no->fator_de_balanceamento;//se não retorna o valor de seu fator de balanceamento

    }
}
```

Figura 5: funções de balanceamento.

A primeira função efetua o cálculo do fator de balanceamento e a segunda função retorna o valor do fator de balanceamento, onde a primeira mencionada utiliza a função altura (2.1.4), para verificar o tamanho das sub-árvores à direita e esquerda e retornar seu resultado, que ao ser analisado na sua função central, já recebe como valor o resultado que deve estar entre -1 e 1.

1.2.6-Rotações duplas

```c
//faz uma rotação dupla pra esquerda
void rotacao_LR(Arv_AVL *raiz){

    rotacao_RR(&(*raiz)->esq);//rotação simples pra esquerda
    rotacao_LL(raiz);//rotação simples pra direita
}

//faz uma rotação dupla pra direita

void rotacao_RL(Arv_AVL *raiz){

    rotacao_LL(&(*raiz)->dir);//rotação simples pra direita
    rotacao_RR(raiz);//rotação simples pra esquerda

}
```

Figura 6: Chamadas para rotações duplas.

As rotações duplas para direita e esquerda são compostas de duas rotações simples(o que muda é a ordem em que são chamadas). Nesse caso, a função chama as duas rotações por função.

1.2.7-Insere na Árvore

```
int inseri_AVL(Arv_AVL *raiz, int valor){

    int ret;

    if(*raiz == NULL){

        elemento *novo = aloca_no(valor);

        *raiz          = novo;

        return 1;
    }

        elemento *atual = *raiz;

        if(valor < atual->dado){

            if(ret = inseri_AVL(&(*raiz)->esq,valor) == 1){

                if(calcula_fat_bal(atual) >=  2 || calcula_fat_bal(atual) <= -2){

                    if(valor < (*raiz)->esq->dado){

                        rotacao_LL(raiz);
                    }
                    else {
                        rotacao_LR(raiz);
                    }

                }
            }
        }
        else if(valor > atual->dado){
            if(ret = inseri_AVL(&(*raiz)->dir,valor) == 1){
                if(calcula_fat_bal(atual) <= -2 || calcula_fat_bal(atual) >= 2){
                    if(valor > (*raiz)->dir->dado){
                        rotacao_RR(raiz);

                    }
                    else {

                        rotacao_RL(raiz);
                    }
                }
            }
        }

        else {
            printf("O valor ja existe!!\n");
```

Figura 7: inserção de elementos e chamada de funções.

De maneira geral, a função insere um novo nó ou inicia uma nova árvore e verifica seu fator de balanceamento. Caso seja maior ou igual a -2 e 2, dois casos são criados especificamente para saber quais rotações efetuar. Se não acontecer nenhum dos dois casos, uma mensagem é exibida dizendo que o número já existe.

Observação: A figura 7 não representa a função por completo por questão de espaço.(O código completo encontra-se nos arquivos da pasta zip).

1.2.8-Exibição

```c
void mostra_arvore(Arv_AVL *raiz){

    if(*raiz != NULL){
        printf("O no : %d\nTem fator de balanceamento: %d\n", (*raiz)->dado, (*raiz)->fator_de_balanceamento);
        mostra_arvore(&(*raiz)->esq);
        mostra_arvore(&(*raiz)->dir);
    }
}
```

Figura 8: Mostra árvore com seu fator de balanceamento.

Mostra no terminal a árvore em pré ordem já balanceada.

1.2.9-Main

```c
#include <stdio.h>
#include <stdlib.h>
#include "avl_2.h"


int main() {

    Arv_AVL *raiz;//ponteiro de ponteiro pra raiz

    raiz = aloca_arvore();//alocando ponteiro de ponteiro na raiz da arvore

    inseri_AVL(raiz, 9);

    inseri_AVL(raiz, 7);

    inseri_AVL(raiz, 2);

    inseri_AVL(raiz, 13);

    inseri_AVL(raiz, 15);

    mostra_arvore(raiz);

    libera_arvore(raiz);


    return 0;
}
```

Figura 9: Inserção de números

Chama a função “inseri_AVL”(1.2.7) e chama a função de exibição(1.2.8), além da função para liberar a árvore, de forma que o programa funcione proporcionalmente ao que se foi proposto.

Para uma melhor verificação do código, há 2 arquivos do tipo “.c” e um arquivo do tipo “.h” para serem executados dentro da pasta zipada. Como método alternativo, o link: https://github.com/JoaoVictorCabraldeMelo/AVL_Tree também pode ser acessado.

2- Execução do algoritmo

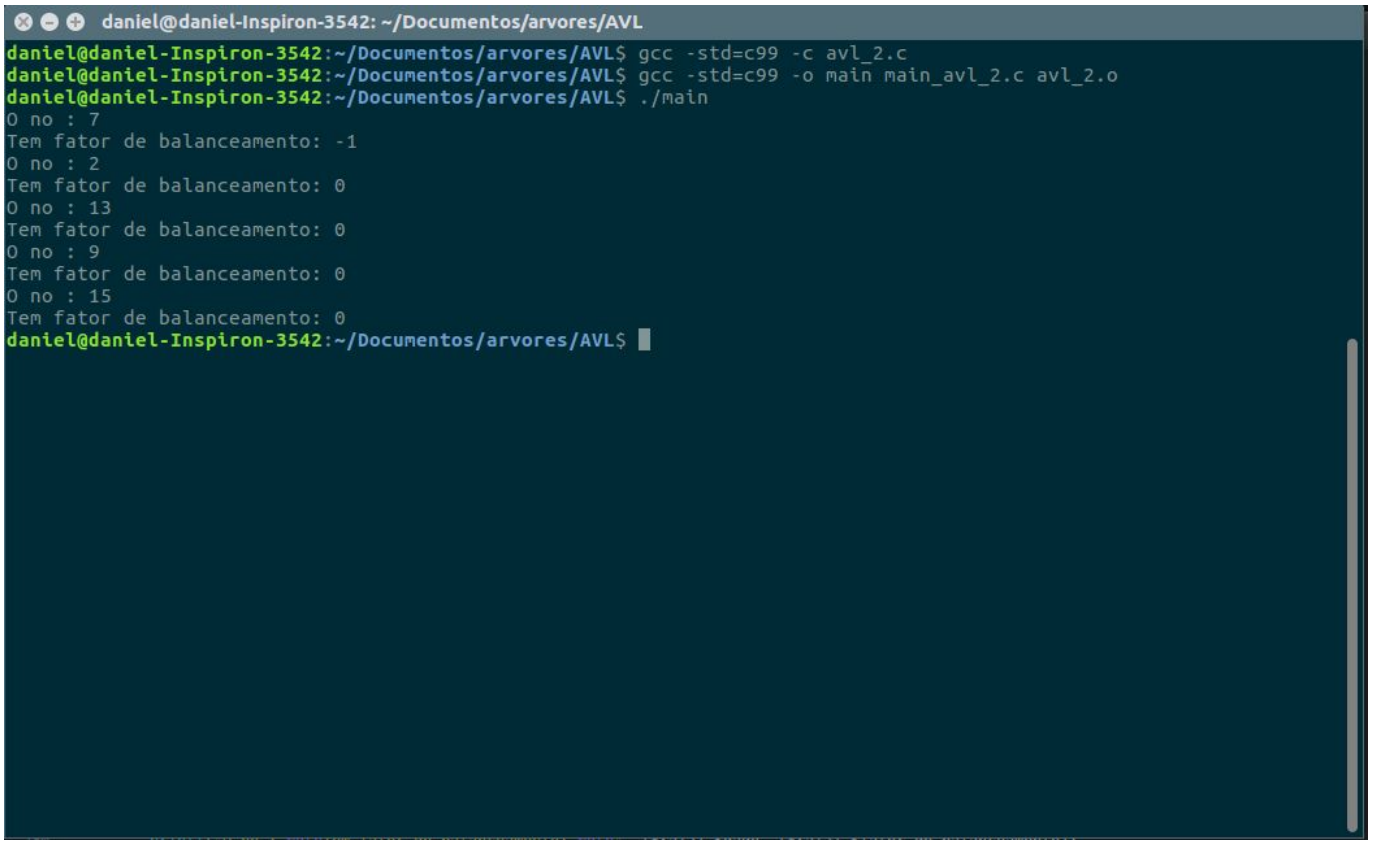

Figura 10: Árvore balanceada

Ao ser executada no terminal, o resultado final é exibido na tela com a árvore já organizada e balanceada, além de mostrar também fatores de balanceamento de cada integrante.

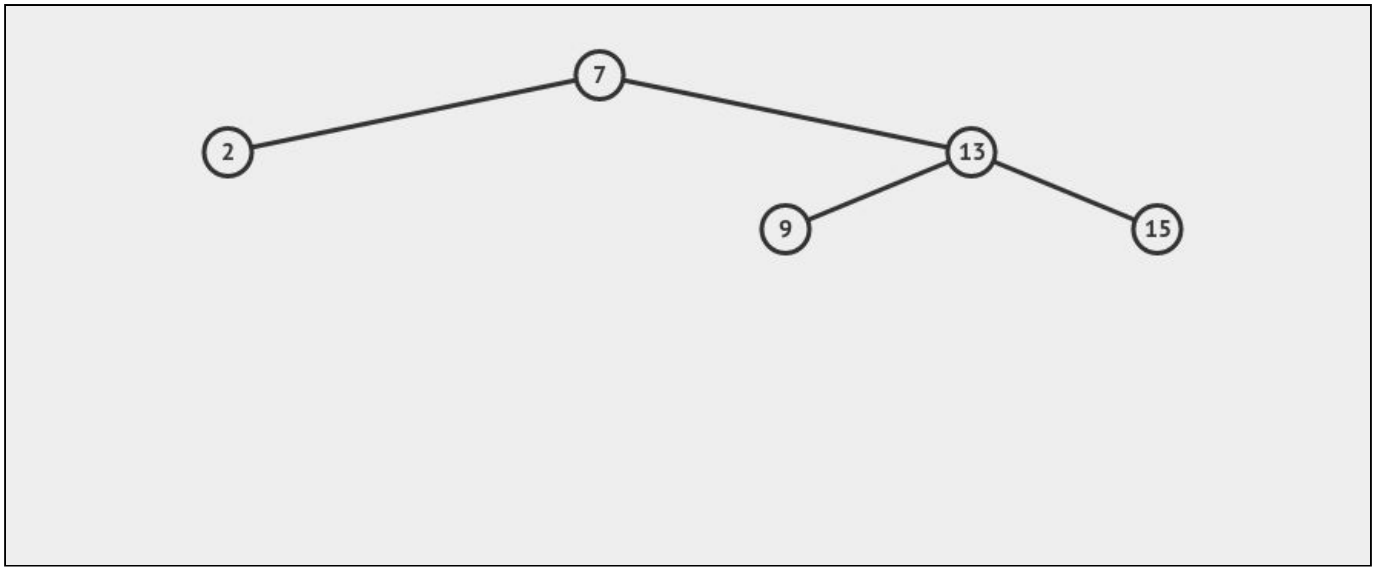

Figura 11: Árvore em formato ilustrativo(Sem fator de balanceamento).

3- Aplicações

A Árvore em AVL pode ser utilizada em dicionários e na geometria computacional (slides), além de poder ser utilizado também em elaboração de IAs que permitem ter um funcionamento menos pesado por conta de seu balanceamento dinâmico.