ÁRVORES AVL

AVLTree

Documentação da biblioteca “AVLTree.h”, escrita por Wallace Luiz Carvalho de Andrade, do curso de Engenharia de Computação da Universidade Federal do Espírito Santo - Campus CEUNES.

SUMÁRIO:

PROPÓSITO

Essa biblioteca foi escrita com o propósito de criar e gerenciar *árvores AVL*, de forma que os dados armazenados são de tipo genérico, ou seja, qualquer tipo de dado, desde os tipos padrões da linguagem C até estruturas criadas pelo usuário.

A criação dessa biblioteca faz parte da avaliação 2 da matéria de Estrutura de Dados II, dos cursos de Engenharia da Computação e Ciência da Computação da UFES (Universidade Federal do Espírito Santo) - campus CEUNES, matéria ministrada pela professora Dra. Luciana Lee.

O QUE SÃO ÁRVORES AVL

Árvore AVL criada no ano de 1962 por Georgy Adelson-Velsky e Yevgeniy Landis, é uma *árvore binária de busca balanceada,* ou seja, ela busca através de rotações, manter as alturas das sub-árvores à direita e à esquerda parecidas, com uma diferença máxima de 1.

Dessa forma, as operações executadas na árvore como inserção, remoção e principalmente a busca possuem velocidade otimizada e tendo como complexidade de tempo (notação big O) O(log n) nessas 3 operações no pior e no melhor caso, já que não é necessário percorrer todos os elementos da árvore para encontrar um. O nome Árvore AVL vem dos nomes de seus criadores, Adelson Velsky e Landis.

ARQUIVOS

Os arquivos incluídos nesta biblioteca são “AVLTree.h” e “AVLTree.c”, onde o arquivo terminado “.h” é o header (cabeçalho) da biblioteca, e nele possui a inclusão de todas as outras bibliotecas necessárias (stdio.h, stdlib.h string.h), juntamente com a estrutura da árvore e o protótipo de todas as funções da biblioteca.

```c
typedef struct ArvoreAVL
{
    void *info;
    int altura;
    struct ArvoreAVL *dir;
    struct ArvoreAVL *esq;
} noAVL;
```

*Estrutura da árvore

FUNÇÕES E UTILIZAÇÃO

Para a utilização desta biblioteca primeiro faz-se necessária a inclusão da mesma no seu arquivo principal:

```c
#include "AVLTree.h"
```

Também é necessário que o usuário crie o tipo de dado a ser usado, seja ele padrão da linguagem C ou uma estrutura criada.

```c
typedef struct itemInfo
{
    char nome[50];
    float preco;
    float peso;
} item;
```

*Exemplo de estrutura

Juntamente com a estrutura, o usuário precisa criar uma função de comparação que retorna “**1**” caso o *primeiro* parâmetro seja **maior** que o *segundo*, “**-1**” caso o *primeiro* parâmetro seja **menor** que o *segundo*, “**0**” caso *ambos* parâmetros sejam **iguais**.

```c
int compara(void *info_1, void *info_2)
{
    item *x = (item *)info_1;
    item *y = (item *)info_2;

    int res;
    res = strcmp(x->nome, y->nome);

    if (res > 0)
        return 1;
    else if (res < 0)
        return -1;
    else
        return 0;
}
```

*Exemplo de função compara, comparando o campo nome da estrutura apresentada anteriormente

Além da função de comparação, caso o usuário queira visualizar a árvore e os dados inseridos, ele deve implementar funções de impressão, uma para imprimir a chave que está sendo usada nas comparações e outra para imprimir toda a estrutura.

```c
void printInfo(void *info)
{
    item *x = (item *)info;
    printf("%s", x->nome);
}

void printFullInfo(void *info)
{
    item *x = (item *)info;
    printf("\nnome: %s", x->nome);
    printf("\npreco R$: %.2f", x->preco);
    printf("\npeso: %.4f", x->peso);
}
```

*Exemplos de funções de impressão, baseadas na estrutura apresentada anteriormente.

Com todas essas funções prontas, basta inicializar as variáveis da árvore e da estrutura usada, para a árvore são necessárias inicializar duas variáveis, uma para as operações de inserção, remoção e impressão, e outra para a operação de busca:

```c
noAVL *ArvoreAVL = NULL;
noAVL *aux = NULL;
```

*No exemplo, a variável arvoreAVL é utilizada como a árvore principal, e aux é utilizada para pegar o retorno da busca.

Já para a estrutura do usuário, é necessário inicializar ela dentro do bloco de repetição onde as operações são executadas, para evitar que toda alteração nesta variável reflita dentro do que já foi guardado na árvore.

```c
int main()
{
    noAVL *ArvoreAVL = NULL;
    noAVL *aux = NULL;
    int op = 0;

    while (op ≠ 5)
    {
        item *produto = (item *)malloc(sizeof(item));
```

IMPRESSÃO DA ÁRVORE

```c
void imprimeArvore(noAVL *arvore, int cont, void (*printInfo)(void *))
{
    int i;

    if (arvore ≠ NULL)
    {
        imprimeArvore(arvore→dir, cont + 1, printInfo);
        for (i = 0; i < cont; i++)
            printf("\t");
        printInfo(arvore→info);
        printf("\n");
        imprimeArvore(arvore→esq, cont + 1, printInfo);
    }
}
```

A função “imprimeArvore” é uma função muito simples e recebe como parâmetro a função recebe a árvore a ser impressa, um inteiro representando a quantidade de tabs básicos (tabs na raiz) que terá (valor recomendado: 0) e também recebe a função que imprime a chave usada na comparação.

Ela percorre recursivamente a árvore começando pela direita e imprimindo a informação, a impressão respeita as alturas dos nós em relação à raiz dando forma a uma árvore.

Exemplo de chamada:

```c
printf("\n\n———————————————————————————————\n\n");
imprimeArvore(ArvoreAVL, 0, printInfo);
printf("\n\n———————————————————————————————\n\n");
```

IMPRESSÃO DE UM NÓ

```
void imprimeNo(noAVL *arvore, void (*printInfo)(void *))
{
    if (arvore ≠ NULL)
    {
        printf("\nInfo:\n");
        printInfo(arvore→info);
        printf("\n");
    }
    else
        printf("O valor nao existe na arvore");
}
```

A função “imprimeNo” é feita para ser utilizada com o resultado da função busca, caso seja utilizada com a árvore principal irá imprimir as informações da raiz da árvore e não do nó desejado. recebe por parâmetro o nó e a função que imprime todos os campos da estrutura.

Exemplo de chamada:

```
aux = buscaNo(ArvoreAVL, produto, compara);
imprimeNo(aux, printFullInfo);
```

BUSCANDO UM NÓ NA ÁRVORE

```c
noAVL *buscaNo(noAVL *ArvoreAVL, void *info, int (*compara)(void *, void *))
{
    if (ArvoreAVL == NULL)
        return ArvoreAVL;
    if (compara(info, ArvoreAVL→info) == -1)
        ArvoreAVL = buscaNo(ArvoreAVL→esq, info, compara);

    else if (compara(info, ArvoreAVL→info) == 1)
        ArvoreAVL = buscaNo(ArvoreAVL→dir, info, compara);
    else
        return ArvoreAVL;

    return ArvoreAVL;
}
```

A função “buscaNo” recebe como parâmetro a árvore a ser analisada, a informação a ser buscada (se for uma estrutura, deve passar uma variavel do tipo da estrutura), e a função de comparação criada pelo usuário. O seu retorno é o nó contendo a informação ou um nó nulo caso a informação não seja encontrada.

A função começa com o caso base, a árvore inteira vazia ou o nó buscando sendo nulo, depois ela percorre recursivamente a árvore verificando se a informação está nas sub-árvores à esquerda ou à direita, por fim caso a informação esteja na árvore entrará no “else” e irá retornar o nó desejado através de todas as chamadas recursivas até o fim da função.

Exemplo de chamada:

```c
aux = buscaNo(ArvoreAVL, produto, compara);
```

INSERINDO UM NÓ NA ÁRVORE

```c
noAVL *insereNo(noAVL *ArvoreAVL, void *info, int (*compara)(void *, void *))
{
    int FatorBalanceamento;

    if (ArvoreAVL == NULL)
        return criaNo(info);

    if (compara(info, ArvoreAVL->info) == -1)
        ArvoreAVL->esq = insereNo(ArvoreAVL->esq, info, compara);

    else if (compara(info, ArvoreAVL->info) == 1)
        ArvoreAVL->dir = insereNo(ArvoreAVL->dir, info, compara);

    else
        return ArvoreAVL;

    ArvoreAVL->altura = retornaMaior(retornaAltura(ArvoreAVL->esq), retornaAltura(ArvoreAVL->dir)) + 1;

    FatorBalanceamento = retornaFatorBalanceamento(ArvoreAVL);

    if (FatorBalanceamento >= 2 && compara(info, ArvoreAVL->esq->info) == -1)
        return rotacaoSimplesDir(ArvoreAVL);

    if (FatorBalanceamento <= -2 && compara(info, ArvoreAVL->dir->info) == 1)
        return rotacaoSimplesEsq(ArvoreAVL);

    if (FatorBalanceamento >= 2 && compara(info, ArvoreAVL->esq->info) == 1)
    {
        ArvoreAVL->esq = rotacaoSimplesEsq(ArvoreAVL->esq);
        return rotacaoSimplesDir(ArvoreAVL);
    }

    if (FatorBalanceamento <= -2 && compara(info, ArvoreAVL->dir->info) == -1)
    {
        ArvoreAVL->dir = rotacaoSimplesDir(ArvoreAVL->dir);
        return rotacaoSimplesEsq(ArvoreAVL);
    }

    return ArvoreAVL;
}
```

A função “insereNo” recebe como parâmetro a árvore em que o nó será inserido, a informação a ser inserida, e a função de comparação, ela retorna a nova árvore com a informação inserida.

A função começa com o caso base, a árvore sendo vazia ou a recursão chegou na exata posição em que a informação deve ser inserida, depois do caso base, ela percorre recursivamente a árvore verificando se deve ser inserido à esquerda ou à direita dos nós subsequentes, o else representa o caso da informação já existir na árvore.

após o final dessas verificações e chamadas recursivas o nó já estará inserido na árvore, a função agora volta em todas as chamadas atualizando a altura de cada nó no caminho, e fazendo o balanceamento caso necessário. as verificações de balanceamento são feitas de forma bem simples:

- Caso o fator de balanceamento do nó seja maior ou igual a 2 e o nó que foi inserido está à esquerda do filho à esquerda, é efetuada uma rotação simples à direita (Rotação LL).
- Caso o fator de balanceamento do nó seja menor ou igual a -2 e o nó que foi inserido está à direita do filho à direita, é efetuada uma rotação simples à esquerda (Rotação RR).
- Caso o fator de balanceamento do nó seja maior ou igual a 2 e o nó que foi inserido está à direita do filho à esquerda, é efetuada uma rotação dupla à direita (Rotação LR), ou seja, uma rotação RR no filho à esquerda e uma rotação LL no nó atual.
- Caso o fator de balanceamento do nó seja menor ou igual a -2 e o nó que foi inserido está à esquerda do filho à direita, é efetuada uma rotação dupla à esquerda (Rotação RL), ou seja, uma rotação LL no filho à direita e uma rotação RR no nó atual.

Após os balanceamentos de todos os nós voltando através da recursão é retornado, para cada chamada recursiva o nó balanceado com altura atualizada e no final da função é retornado o nó raiz da árvore.

Exemplo de chamada:

```
ArvoreAVL = insereNo(ArvoreAVL, produto, compara);
```

REMOVENDO UM NÓ DA ÁRVORE

```c
noAVL *removeNo(noAVL *ArvoreAVL, void *info, int (*compara)(void *, void *))
{
    int FatorBalanceamento;

    if (ArvoreAVL == NULL)
        return ArvoreAVL;

    if (compara(info, ArvoreAVL->info) == -1)
        ArvoreAVL->esq = removeNo(ArvoreAVL->esq, info, compara);

    else if (compara(info, ArvoreAVL->info) == 1)
        ArvoreAVL->dir = removeNo(ArvoreAVL->dir, info, compara);

    else
    {
        if (ArvoreAVL->dir == NULL && ArvoreAVL->esq == NULL)
        {
            free(ArvoreAVL);
            ArvoreAVL = NULL;
        }

        else if (ArvoreAVL->esq != NULL && ArvoreAVL->dir == NULL)
        {
            ArvoreAVL = ArvoreAVL->esq;
        }

        else if (ArvoreAVL->dir != NULL && ArvoreAVL->esq == NULL)
        {
            ArvoreAVL = ArvoreAVL->dir;
        }

        else
        {
            noAVL *novo = achaMenor(ArvoreAVL->dir);
            ArvoreAVL->info = novo->info;
            ArvoreAVL->dir = removeNo(ArvoreAVL->dir, novo->info, compara);
        }
    }
    if (ArvoreAVL == NULL)
        return ArvoreAVL;

    ArvoreAVL->altura = 1 + retornaMaior(retornaAltura(ArvoreAVL->esq), retornaAltura(ArvoreAVL->dir));

    FatorBalanceamento = retornaFatorBalanceamento(ArvoreAVL);

    if (FatorBalanceamento >= 2 && retornaFatorBalanceamento(ArvoreAVL->esq) >= 0)
        return rotacaoSimplesDir(ArvoreAVL);

    if (FatorBalanceamento <= -2 && retornaFatorBalanceamento(ArvoreAVL->dir) <= 0)
        return rotacaoSimplesEsq(ArvoreAVL);

    if (FatorBalanceamento >= 2 && retornaFatorBalanceamento(ArvoreAVL->esq) < 0)
    {
        ArvoreAVL->esq = rotacaoSimplesEsq(ArvoreAVL->esq);
        return rotacaoSimplesDir(ArvoreAVL);
    }

    if (FatorBalanceamento <= -2 && retornaFatorBalanceamento(ArvoreAVL->dir) > 0)
    {
        ArvoreAVL->dir = rotacaoSimplesDir(ArvoreAVL->dir);
        return rotacaoSimplesEsq(ArvoreAVL);
    }
    return ArvoreAVL;
}
```

A função “removeNo” recebe como parâmetro a árvore em que o nó a ser removido está, a informação a ser removida e a função de comparação, e retorna a árvore resultante da operação.

A função começa com o caso base, a árvore vazia ou a informação a ser removida não está nela, em seguida parte para as chamadas recursivas verificando se a informação está à esquerda ou à direita, e caso seja encontrada entrará no else. Dentro do else é verificado primeiro se o nó a ser removido é um nó folha, e caso seja ele é apenas liberado da memória. Depois é verificado se o nó possui apenas filhos a esquerda, e caso sim, o nó a ser removido recebe as informações do filho a esquerda, sem a necessidade de liberar a memória já que as informações nele foram sobrescritas, também é verificado se o nó possui apenas filhos à direita, caso sim, o nó a ser removido recebe as informações do filho à direita, e por último é verificado se o nó possui filhos de ambos os lados, tornando necessário encontrar um sucessor para assumir o lugar desse nó, e em seguida é chamada a função de remoção para remover o sucessor.

Caso o resultado desse processo seja uma uma árvore nula, ela será retornada sem a necessidade de balanceamentos e atualizações na altura.

Em seguida é feita a atualização da altura e o balanceamento dos nós enquanto volta pelo caminho feito através da recursão:

- Caso o fator de balanceamento do nó seja maior ou igual a 2 e o fator de balanceamento do filho à esquerda for positivo, é efetuada uma rotação simples à direita (Rotação LL).
- Caso o fator de balanceamento do nó seja menor ou igual a -2 e o fator de balanceamento do filho à direita for negativo, é efetuada uma rotação simples à esquerda (Rotação RR).
- Caso o fator de balanceamento do nó seja maior ou igual a 2 e o fator de balanceamento do filho à esquerda for negativo, é efetuada uma rotação dupla à direita (Rotação LR), ou seja, uma rotação RR no filho à esquerda e uma rotação LL no nó atual.
- Caso o fator de balanceamento do nó seja menor ou igual a -2 e o fator de balanceamento do filho à direita

for positivo, é efetuada uma rotação dupla à esquerda (Rotação RL), ou seja, uma rotação LL no filho à direita e uma rotação RR no nó atual.

Após os balanceamentos de todos os nós voltando através da recursão é retornado, para cada chamada recursiva o nó balanceado com altura atualizada e no final da função é retornado o nó raiz da árvore.

Exemplo de chamada:

```
ArvoreAVL = removeNo(ArvoreAVL, produto, compara);
```

ROTAÇÃO SIMPLES À ESQUERDA

```c
noAVL *rotacaoSimplesEsq(noAVL *no_A)
{
    noAVL *no_B = no_A->dir;
    noAVL *aux = no_B->esq;

    no_B->esq = no_A;
    no_A->dir = aux;

    no_A->altura = retornaMaior(retornaAltura(no_A->esq), retornaAltura(no_A->dir)) + 1;
    no_B->altura = retornaMaior(retornaAltura(no_B->esq), retornaAltura(no_B->dir)) + 1;

    return no_B;
}
```

NOTA: ESSA FUNÇÃO NÃO DEVE SER UTILIZADA PELO USUÁRIO EM HIPÓTESE ALGUMA.

A função “rotacaoSimplesEsq” recebe como parâmetro o nó a ser balanceado e retorna o nó já balanceado.

A função começa criando um nó auxiliar “B” que recebe o filho à direita do nó “A” (passado como parâmetro), e um nó auxiliar “aux” que recebe o filho à esquerda do nó B (o filho à esquerda do filho à direita de A).

Em seguida, faz o filho à esquerda do nó B apontar para o nó A, e o filho à direita do nó A apontar para o aux(Antigo filho à esquerda do nó B).

Após esse processo são atualizadas as alturas dos nós A e B e o nó B é retornado como nova raiz da subárvore.

ROTAÇÃO SIMPLES À DIREITA

```c
noAVL *rotacaoSimplesDir(noAVL *no_A)
{
    noAVL *no_B = no_A->esq;
    noAVL *aux = no_B->dir;

    no_B->dir = no_A;
    no_A->esq = aux;

    no_A->altura = retornaMaior(retornaAltura(no_A->esq), retornaAltura(no_A->dir)) + 1;
    no_B->altura = retornaMaior(retornaAltura(no_B->esq), retornaAltura(no_B->dir)) + 1;

    return no_B;
}
```

NOTA: ESSA FUNÇÃO NÃO DEVE SER UTILIZADA PELO USUÁRIO EM HIPÓTESE ALGUMA.

A função “rotacaoSimplesDir” recebe como parâmetro o nó a ser balanceado e retorna o nó já balanceado.

A função começa criando um nó auxiliar “B” que recebe o filho à esquerda do nó “A” (passado como parâmetro), e um nó auxiliar “aux” que recebe o filho à direita do nó B (o filho à direita do filho à esquerda de A).

Em seguida, faz o filho à direita do nó B apontar para o nó A, e o filho à esquerda do nó A apontar para o aux(Antigo filho à direita do nó B).

Após esse processo são atualizadas as alturas dos nós A e B e o nó B é retornado como nova raiz da subárvore.

ACHANDO O MENOR ELEMENTO

```
noAVL *achaMenor(noAVL *ArvoreAVL)
{
    while (ArvoreAVL->esq != NULL)
        ArvoreAVL = ArvoreAVL->esq;

    return ArvoreAVL;
}
```

A função “achaMenor” é uma função utilizada na remoção para encontrar o sucessor, e pode ser utilizada pelo usuário caso queira encontrar o menor elemento da árvore para imprimi-lo.

A função recebe como parâmetro a árvore a ser analisada, e retorna o nó desta árvore com o menor elemento. ATENÇÃO: ASSIM COMO NA FUNÇÃO BUSCA VOCÊ DEVE UTILIZAR UMA OUTRA VARIÁVEL SEM SER A ÁRVORE PRINCIPAL PARA RECEBER O RETORNO DESSA FUNÇÃO;