*Jeferson Luiz Alves de Souza*

Documentação

Aqui teremos a implementação das bibliotecas usadas, como a “AVL.h” que seria nosso cabeçalho.

//autor: Jeferson Luiz Alves de Souza

#include <stdio.h>

#include <stdlib.h>

#include "AVL.h"

Função destinada a comparar uma chave genérica descrita pelo usuário e uma chave atual da arvore passada como parâmetro.

Perceba o uso da *cast* nas comparações. Isso é usado pois essa função é uma das que o cliente irá reformular caso o tipo de dado não seja inteiro.

int comparaChaves(void\* chave1, void\* chave2){

Aqui comparamos se a *chave1* é maior que a *chave2* e retornando 1 caso positivo.

if (\*(int\*)chave1 > \*(int\*)chave2)

return 1;

Se não, compara se a *chave1* é menor que a *chave2* retornando -1 caso positivo.

else if (\*(int\*)chave1 < \*(int\*)chave2)

return -1;

Se nenhuma das duas comparações forem verdadeira, significa que ambas as chaves são iguais. Nesse caso, a função retorna 0.

else

return 0;

}

Função destinada a imprimir uma chave genérica passada pelo usuário.

void printaChave(No \*root){

printf("%d\n", \*(int\*)root->chave);

}

Função destinada a alocar memória para um tipo de dados que o usuário (cliente) irá definir. Vemos que o retorno dessa função é um inteiro. Porque a priori, estamos supondo que o dado que será usado pelo cliente será inteiro. Porém essa função é implementada no arquivo *main* justamente para ser modificado caso o tipo de dado não seja inteiro.

int \*alocaInt(){

int\* a = NULL;

Aloca memória para o tipo de dado que for usado. Isso ocorre porque estamos trabalhando com tipo genérico de dado. Isso significa que cada tipo de dado tem um tamanho diferente. Precisando especificar o tamanho antes de retornar.

a = (int\*) malloc(sizeof(int));

Pega o tipo de dado usado (int nesse exemplo) e associa a variável que já foi alocada. E a retorna.

scanf("%d", a);

return a;

}

Função destinada a criar um nó e o retornar à função chamada.

No \*createNo(void \*x){

No \*root = NULL;

Caso o nó seja alocado com sucesso, a instrução retornará true e assim, entrará na condição, que será responsável por atribuir valores aos campos do nó criado.

if(root = (No\*)malloc(sizeof(No))){

root->bal = 0;

Nessa linha de código, devemos tomar cuidado pois, não sabemos o tipo de dado que o cliente está usando.

//root->chave = x;

root->left = NULL;

root->right = NULL;

return root;

}

Caso a alocação não tenha sucedido, retorne NULO.

return NULL;

}

Função destinada a inserir, fazer as chamadas das rotações mediante cada caso.

Temos aqui um ponteiro para ponteiro pois possivelmente, mudamos apontamentos dentro da estrutura usada. O segundo parâmetro é o valor que será usado para marcar as rotações. Ou seja, quando uma rotação ocorrer, significa que o balanço da árvore foi restituído e, então, esse valor deve ser mudado como forma de marcação. O valor x sucede o valor da chave a ser inserido

void insertAVL(No \*\*root, int \*h, void \*x){

Caso em que a raiz ainda não existe. Ou seja, primeiro elemento a ser inserido. A raiz.

if(\*root == NULL){

Chama a função para alocar um nó (citada a cima).

\*root = createNo(x);

return;

}

Se o nó já existir, vai entrar nas próximas condições do próximo bloco.

else{

Segundo caso trivial, em que o nó a se inserir já existe na estrutura. Isso significa que o cliente deverá IMPLEMENTAR a função da condição (“comparaChave”).

if(comparaChave(x, (\*root)->chave) == 0){

printf("\nElement already exists.");

Ou seja, o como o nó já existe, a variável h deve ser mudada para 1. Assim, todos os retornos saberão se a mudança foi feita ou não.

\*h = 1;

return;

}

Caso em que a chave é “menor” que a raiz sendo comparada. Se isso acontecer, significa que a chave a ser inserida deverá ficar à esquerda da árvore em questão. Essa função compara e retorna 1 caso a chave esteja à esquerda da árvore. (função implementada pelo cliente).

if(comparaChave(x, (\*root)->chave) == 1){

Aqui ocorre a chamada recursiva. Como dito, a chave deve ser adicionada a sub árvore esquerda da atual. Porquanto, a chamada recursiva da função deve ser passada como parâmetro a sub árvore esquerda como se sucede.

insertAVL(&(\*root)->left, h, x);

Quando a linha de código acima retornar, devemos considerar as questões de balanceamento já que um novo nó possa ser que tenha sido inserido. Nesse ponto a variável h faz seu papel. Se h = 0, então significa que a rotação ainda não foi feita OU nenhum novo nó foi inserido. Se h = 1, então significa que a rotação foi feita OU o novo suposto nó já existe, o que significa que não precisa entrar nessa condicional, nem fazer nenhuma rotação.

if(\*h == 0){

Devemos considerar o fato de que estamos no caso em que o nó foi inserido na sub árvore esquerda. Isso significa que o valor do balanceamento faz diferença.

Caso o nó pai antes da inserção era -1, significa que o nó estava desbalanceado pela direita. Com a adição desse novo nó na sub árvore esquerda, a raiz atual fica balanceada.

switch((\*root)->bal){

Caso o nó pai antes da inserção era -1, significa que o nó estava desbalanceado pela direita. Com a adição desse novo nó na sub árvore esquerda, a raiz atual fica balanceada (= 0).

case -1:

(\*root)->bal = 0;

break;

Caso o nó pai antes da inserção era 0, significa que o nó ou era folha, ou tinha filhos balanceados pela esquerda e direita. Com a adição desse novo nó na sub árvore esquerda, a raiz atual fica desbalanceada pela ESQUERDA (=-1).

case 0:

(\*root)->bal = 1;

break;

Caso o nó pai antes da inserção era 1, significa que o nó estava desbalanceado pela esquerda. Com a adição desse novo nó na sub árvore esquerda, a raiz atual fica desbalanceada pela ESQUERDA (=-2). E é esse caso em que a rotação deve ser feita. A função “caso1” é chamada, que significa o caso em que é feito as rotações a direita.

case 1:

caso1(root);

\*h = 1;

break;

}

}

}

Se a chave a ser adicionada não for menor nem igual a raiz atual, ela é maior. E esse bloco trata desse caso. Em que o novo nó deve ser inserido na sub árvore direita.

else{

Aqui ocorre a chamada recursiva. Como dito, a chave deve ser adicionada a sub árvore direita da atual. Porquanto, a chamada recursiva da função deve ser passada como parâmetro a sub árvore esquerda como se sucede.

insertAVL(&(\*root)->right, h, x);

Quando a linha de código acima retornar, devemos considerar as questões de balanceamento já que um novo nó possa ser que tenha sido inserido. Nesse ponto a variável h faz seu papel. Se h = 0, então significa que a rotação ainda não foi feita OU nenhum novo nó foi inserido. Se h = 1, então significa que a rotação foi feita OU o novo suposto nó já existe, o que significa que não precisa entrar nessa condicional, nem fazer nenhuma rotação.

if(\*h == 0){

Devemos considerar o fato de que estamos no caso em que o nó foi inserido na sub árvore direita. Isso significa que o valor do balanceamento faz diferença.

switch((\*root)->bal){

Caso o nó pai antes da inserção era 1, significa que a árvore estava desbalanceada pela esquerda. Com a inserção de um nó a direita, a árvore fica balanceada (=0).

case 1:

(\*root)->bal = 0;

\*h = 1;

break;

Caso o nó pai antes da inserção era 0, significa que o nó era folha OU estava balanceado tanto pela direita quanto pela esquerda. Nesse caso, vai ficar desbalanceado pela direita já que o nó foi adicionado em sua sub árvore direita (=-1).

case 0:

(\*root)->bal = -1;

break;

Caso o nó pai antes da inserção era -1, significa que o nó estava desbalanceado pela direita. Com a adição desse novo nó na sub árvore direita, a raiz atual fica desbalanceada (=-2) e, é aqui que entra a função de balanceamento.

case -1:

Função de rotações pela esquerda.

caso2(root);

\*h = 1;

break;

}

}

}

}

}

Perceba que nos casos em que a árvore recebe o balanceamento igual a zero (bal = 0) ou ocorre uma rotação ou o nó já existe, a variável h recebe valor 1. Que significa que as recursões não precisam mais analisar o fator de balanceamento.

Função destinada a deletar um nó e fazer as chamadas das rotações mediante cada caso.

Temos aqui um ponteiro para ponteiro pois possivelmente, mudamos apontamentos dentro da estrutura usada. O segundo parâmetro é o valor que será usado para marcar as rotações. Ou seja, quando uma rotação ocorrer, significa que o balanço da árvore foi restituído e, então, esse valor deve ser mudado como forma de marcação. O valor x sucede o valor da chave a ser inserido.

No \*deleteAVL(No \*root, int \*i, void \*x){

De antemão devemos considerar 4 casos possíveis para a remoção de um elemento da árvore: 1 - elemento não tem filhos (nó folha); 2 - elemento possui apenas filhos a esquerda; 3 - elemento possui apenas filhos a direita; 4 - elemento possui filho a esquerda e a direita.

No \*aux, \*aux2;

Caso trivial. Valor não existe, então, não pode ser removido.

if(root == NULL){

printf("\nthe value not exist.");

return NULL;

}else{

Caso em que o valor é encontrado. Ou seja, é o nó atual a ser removido.

if(comparaChave(x, (\*root)->chave) == 0){

1 – Caso em que o elemento não tem filhos em nenhuma das duas sub árvores (nó folha). Nesse caso não precisamos se preocupar com a mudança de ponteiros já que é um nó folha. Porquanto, basta liberar a memória e retornar NULO.

if(root->left == NULL && root->right == NULL){

printf("\nsheet");

free(root);

return NULL;

2 – Caso tenha filhos apenas pela esquerda. Nesse caso, é fácil o entendimento quando se entende o caso 1. Se existe apenas sub árvore pela esquerda, basta quer retornemos seu filho a esquerda como se segue.

}else if(root->left != NULL && root->right == NULL){

printf("\children just left");

aux = root->left;

free(root);

return aux;

3 – Caso tenha filhos apenas pela direita. Mesma ideia do caso 2.

}else if(root->left == NULL && root->right != NULL){//caso 3: filho apenas à direita

printf("\children just right");

aux = root->right;

free(root);

return aux;

4 – Caso tenha filhos tanto pela esquerda quando pela direita.

}else{

printf("\nchildren left and right");

Nesse caso, podemos fazer de algumas maneiras. As duas mais populares é trocar esse elemento por um elemento maior mais próximo ou menor mais próximo. Em outras palavras, podemos pegar o maior elemento da sub árvore esquerda (“elemento mais a direita da sub árvore a esquerda”), ou pegarmos o menor elemento da sub árvore direita (“elemento mais a esquerda da sub árvore a direita”). Nesse caso, usamos a sub árvore direita, como se segue.

int x = auxDeleteAVL(root->right);

Criamos um auxiliar para ajudar no reapontamento.

No \*ptr = createNo(x);

Aqui irá remover o elemento folha mais à esquerda da sub árvore direita como mencionamos. Daí em diante, ele não existirá mais na árvore, apenas uma “copia” dele no ponteiro “ptr” que usaremos para realizar o reapontamento.

root = deleteAVL(root, i, x);

aux = root->left;

aux2 = root->right;

free(root);

ptr->left = aux;

ptr->right = aux2;

return ptr;

}

Caso o elemento ainda não foi encontrado, iremos continuar com a recursividade.

}else{

Elemento está na sub árvore esquerda (é menor que a raiz).

if(comparaChave(x, (\*root)->chave) == 1){

root->left = deleteAVL(root->left, i, x);

Mesmo caso da função de inserção. A variável i faz o mesmo papel da variável h na função de inserção. Devemos ter em consideração que haverá uma mudança no balanceamento cada vez que um elemento for removido.

if(\*i == 0){

Só entrará aqui caso não tenha ocorrido qualquer remoção da árvore OU tenha entendido que o elemento NÃO existe na árvore.

Não esqueça que aqui, estamos na sub árvore ESQUERDA.

switch(root->bal){

Se o fator de balanceamento for igual a -1, significa que o elemento estava desbalanceado pela direita e, removendo um elemento da esquerda, o fator de balanceamento fica -2. Necessitando fazer uma rotação à esquerda (“caso2”).

case -1:

caso2(&root);

\*i = 1;

break;

Se o fator de balanceamento for igual a 0, significa que a árvore estava balanceada. Remover um elemento à esquerda, significa ficar desbalanceada para a direita (=-1).

case 0:

root->bal = -1;

break;

Se o fator de balanceamento for igual a 1, significa que a árvore estava desbalanceada pela esquerda. Removendo um elemento a esquerda, faz com que a árvore, naturalmente volte a ser balanceada (=0).

case 1:

root->bal = 0;

break;

}

}

}

Elemento está na sub árvore direita (é maior que a raiz).

else{

root->right = deleteAVL(root->right, i, x);

if(\*i == 0){

Só entrará aqui caso não tenha ocorrido qualquer remoção da árvore OU tenha entendido que o elemento NÃO existe na árvore.

Não esqueça que aqui, estamos na sub árvore DIREITA.

switch(root->bal){

Se o fator de balanceamento for igual a -1, significa que a árvore estava desbalanceada pela direita. Removendo um elemento a direita, faz com que a árvore, naturalmente volte a ser balanceada (=0).

case -1:

root->bal = 0;

break;

Se o fator de balanceamento for igual a 0, significa que a árvore estava balanceada. Remover um elemento à direita, significa ficar desbalanceada para a direita (=1).

case 0:

root->bal = 1;

break;

Se o fator de balanceamento for igual a 1, significa que o elemento estava desbalanceado pela esquerda e, removendo um elemento da direita, o fator de balanceamento fica -2. Necessitando fazer uma rotação à direita (“caso1”).

case 1:

caso1(&root);

\*i = 1;

break;

}

}

}

}

}

}

Função auxiliar a delete. Ela entra para suprir a necessidade do caso 4 em que existe tanto filho a esquerda e direita. Essa função procura o maior elemento mais a esquerda de uma árvore e o retorna.

int auxDeleteAVL(No \*root){

No \*aux = root;

Caso a arvore não seja NULA, ou seja, caso a árvore exista, iremos percorrer até encontrar o nó folha mais a esquerda e p retornar.

if(root != NULL){

while(aux->left != NULL){

aux = aux->left;

}

return aux->chave;

}

Se a árvore não existir, retorne -1.

return -1;

}

Função que faz a rotação do caso 1, ou seja, a rotação para a direita quando é chamada.

void caso1(No \*\*root){

No \*u = (\*root)->left;

No \*z;

Se o fator de balanceamento for igual 1, significa uma rotação simples à direita.

if(u->bal = 1){

printf("\nright rotation");

(\*root)->left = u->right;

u->right = (\*root);

Considerando a rotação, nós fazemos o fator de balanceamento ser zero já que ficará balanceado após essa rotação.

(\*root)->bal = 0;

(\*root) = u;

}

Se o fator de balanceamento for diferente de 1, significa uma dupla rotação à direita.

else{

printf("\ndouble right rotation");

z = u->right;

u->right = z->left;

z->left = u;

(\*root)->left = z->right;

z->right = (\*root);

Aqui iremos reorganizar o fator de balanceamento dos filhos que foram rotacionados dependendo de seus valores de balanceamento.

if(z->bal == 1) (\*root)->bal = -1;

else (\*root)->bal = 0;

if(z->bal == -1) u->bal = 1;

else u->bal = 0;

(\*root) = z;

}

Aqui apenas zeramos o balanceamento da “nova” raiz.

(\*root)->bal = 0;

}

Função que faz a rotação do caso 1, ou seja, a rotação para a direita quando é chamada.

void caso2(No \*\*root){

No \*u, \*z;

u = (\*root)->right;

Se o fator de balanceamento for igual -1, significa uma rotação simples à esquerda.

if(u->bal = -1){

printf("\nleft rotation");

(\*root)->right = u->left;

u->left = (\*root);

Considerando a rotação, nós fazemos o fator de balanceamento ser zero já que ficará balanceado após essa rotação.

(\*root)->bal = 0;

(\*root) = u;

}

Se o fator de balanceamento for diferente de 1, significa uma dupla rotação à esquerda.

else{

printf("\ndouble left rotation");

z = u->left;

u->left = z->right;

z->right = u;

(\*root)->right = z->left;

z->left = (\*root);

Aqui iremos reorganizar o fator de balanceamento dos filhos que foram rotacionados dependendo de seus valores de balanceamento.

if(z->bal == 1) u->bal = -1;

else u->bal = 0;

if(z->bal == -1) (\*root)->bal = 1;

else (\*root)->bal = 0;

(\*root) = z;

}

Aqui apenas zeramos o balanceamento da “nova” raiz.

(\*root)->bal = 0;

}

Função destinada a imprimir a árvore.

void printAVL(No \*root, int nivel){

Caso a árvore seja NULA, ou chegue ao final da árvore, retorne.

if(!root) return;

Percorre a sub árvore direita.

printAVL(root->right, nivel+1);

linha responsável por imprimir o espaçamento adequado para os nós.

for(i = 0; i < nivel; i++) printf("\t");

printa o elemento baseado na função “printaChave” incrementada pelo cliente.

printaChave(root);

Percorre a sub árvore esquerda.

printAVL(root->left, nivel+1);

}

Função destinada a percorrer a árvore para encontrar um elemento existente. Seu primeiro parâmetro é a raiz dessa árvore. Seu segundo argumento é o elemento a ser inserido.

No \*searchAVL(No \*root, void \*x){

Se a árvore não existir (NULA) retorna NULL à função que a chamou.

if(!root) return NULL;

Aqui, chama a função incrementada pelo usuário para comparar o tipo do elemento utilizado. Se retornar zero, significa que é o elemento buscado.

if(comparaChave(x, (\*root)->chave) == 0) return root;

Aqui, chama a função incrementada pelo usuário para comparar o tipo do elemento utilizado. Se retornar 2, significa que está à DIREITA da raiz atual.

else if((comparaChave(x, (\*root)->chave) == 2) return searchAVL(root->left, x);

Caso contrário, significa que está a ESQUERDA da raiz atual.

else return searchAVL(root->right, x);

}