**Documentação – Arvore Rubro Negra**

Jeferson Luiz Alves de Souza – 2018103110

Também conhecida como árvore vermelho-preto, a Rubro-Negra é um tipo de árvore binária balanceada. Essa estrutura de dados foi originalmente criada por Rudolf Bayer em 1972. Era chamada de Árvores Binárias Simétricas.

Ela é uma alternativa da árvore AVL. Porém, ambas têm suas especialidades. Se a operação de busca for mais usada, utilize a AVL. Se a operação de inserção e remoção for mais usada, utilize a Rubro-Negra.

Função destinada a criar um nó sentinela e retornar no novo nó.

Esse nó sentinela é um nó de referência. A estrutura Rubro-Negra, não utiliza de apontamentos para *NULL*, mas despõe de um nó diferenciado dos nós internos. Esse nó tem seus apontamentos para si mesmo. Sempre existirá apenas um nó sentinela durante todo o programa.

Artigo \*createSentinel(){

Alocação de memória para o novo nó.

Artigo \*sent = (Artigo\*)malloc(sizeof(Artigo));

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(sent){

Aqui fazemos com que o os apontamentos sejam apontados para o próprio nó.

sent->color = SENTINEL;

sent->left = sent;

sent->right = sent;

sent->dad = sent;

return sent;

}

Caso não haja êxito na alocação do nó, a função retorna NULL.

return NULL;

}

Função destinada a criar um nó interno da árvore.

Seu primeiro parâmetro é o a *identificação* da chave e o segundo é o nó *sentinela*.

Esse é o tipo de nó que é usado para o funcionamento da árvore. São os nós internos.

Artigo \*createArtigo(int id, Artigo \*sent){

Alocação de memória para o novo nó.

Artigo \*ptr = (Artigo\*)malloc(sizeof(Artigo));

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(ptr){

ptr->id = id;

printf("ano : ");

scanf("%d", &ptr->ano);

Necessário para que ocorra tudo bem com a função *fgets*.

getchar();

printf("autor : ");

fgets(ptr->autor, 200, stdin);

printf("titulo : ");

fgets(ptr->titulo, 200, stdin);

printf("revista : ");

fgets(ptr->revista, 200, stdin);

printf("DOI : ");

fgets(ptr->DOI, 200, stdin);

printf("palavra chave: ");

fgets(ptr->palavraChave, 200, stdin);

Todo nó inicialmente da arvore começa sendo *rubro*.

ptr->color = RUBRO;

ptr->dad = sent;

ptr->left = sent;

ptr->right = sent;

return ptr;

}

Caso não haja êxito na alocação do nó, a função retorna *NULL*.

return NULL;

}

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. Seu terceiro argumento é o nó sentinela. Seu quarto parâmetro é um nó para saber se precisa exibir uma mensagem ou não.

Artigo \*searchArtigo(Artigo \*mainArtigo, int key, Artigo \*sent, int n){

Artigo \*ptr = mainArtigo;

Laço vai rodar enquanto o nó não encontrar o nó procurado ou chegar ao nó sentinela (nó folha).

while(ptr != sent){

if(ptr->id == key) return ptr;

else if(ptr->id > key) ptr = ptr->left;

else ptr = ptr->right;

}

Aqui entra o papel do quarto parâmetro. Ele se ele for igual a zero, ele não imprime a mensagem. Isso ocorre porque essa função também é usada em outras funções para verificação e a impressão dessa mensagem pode poluir o designe dependendo da ocasião.

if(n != 0) printf("\nchave nao existe");

return NULL;

}

Função destinada a inserir um novo elemento da árvore.

Seu primeiro parâmetro é a raiz da árvore, seu segundo é um novo *identificador* de chave. O terceiro é o nó *sentinela*.

void insertRB(Artigo \*\*mainArtigo, int id, Artigo \*sent){

Cria um novo nó a ser inserido.

Artigo \*ptr = createArtigo(id, sent);

Artigo \*v = sent;

Artigo \*x = \*mainArtigo;

O laço roda enquanto o auxilia x não encontra a posição ao novo nó ser inserido.

while(x != sent){

if(x->id == ptr->id){

printf("\nError: chave existe.");

}

Aqui v é exatamente o pai de x. Já que x vai descer um nível nas três próximas linhas que se segue.

v = x;

if(ptr->id < x->id) x = x->left;

else x = x->right;

}

Nesse momento, o ponteiro x é a posição aonde o nó deve ser inserido e o ponteiro v é seu pai.

ptr->dad = v;

Ou seja, se a arvore *mainArtigo* passada for *NULA*, muda-se o apontamento do ponteiro para a raiz já que o novo nó será a raiz.

if(v == sent) \*mainArtigo = ptr;

Casos em que verifica se é na direita ou à esquerda que será adicionado.

else{

if(ptr->id < v->id) v->left = ptr;

else v->right = ptr;

}

Essa função auxiliar é chamada porque pode ter ocorrido uma violação na árvore.

insertAuxRB(\*mainArtigo, ptr, sent);

}

Função que auxilia nos apontamentos do novo nó inserido e verifica se ouve alguma

violação da árvore Rubro-Negra.

Primeiro argumento é a raiz da árvore, o segundo é o ponteiro que foi inserido e o terceiro é o nó sentinela.

void insertAuxRB(Artigo \*mainArtigo, Artigo \*ptr, Artigo \*sent){

Artigo \*v, \*w, \*t;

O objetivo do laço *enquanto* é deslocar para cima a cor negra extra.

Como estamos usando um nó sentinela como externo, então esse nó normalmente é negro por regra. Mas, para associação aqui no código, usamos como se ele n fosse nem uma, nem outra cor para n haver erros de sintaxe, como na comparação a baixo.

↓ ↓

while(ptr->dad->color == RUBRO){

v = ptr->dad;

w = v->dad;

Trocando direita por esquerda

if(v == w->left){

t = w->right;

Caso 1. Aqui o tio é rubro e o nó é o filho a esquerda.

if(t->color == RUBRO){

v->color = NEGRO;

t->color = NEGRO;

w->color = NEGRO;

Atualizando o nó problemático

ptr = w;

}

else{

Caso em que o tio não é rubro e o ponteiro é filho a direita do nó observado.

if(ptr == v->right){

Atualizando o nó problemático.

ptr = v;

No caso 2, precisamos chamar a função porque precisa ser realizado uma rotação a esquerda.

rotationLeft(&mainArtigo, ptr, sent);

}

Como o caso 3 é uma parte do caso 2, então se segue.

ptr->dad->color = NEGRO;

w->color = RUBRO;

No caso 3, precisamos chamar a função porque precisa ser realizado uma rotação a direita.

rotationRight(&mainArtigo, w, sent);

}

}

Essa parte do código trata simetricamente das mesmas modificações acima, mas agora, pelo lado direito.

else{

t = w->left;

if(t->color == RUBRO){

v->color = NEGRO;

t->color = NEGRO;

w->color = RUBRO;

ptr = w;

}

else{

if(ptr == v->left){

ptr = v;

rotationRight(&mainArtigo, ptr, sent);

}

ptr->dad->color = NEGRO;

w->color = RUBRO;

rotationLeft(&mainArtigo, w, sent);

}

}

}

Como o laço joga as cores negras para cima, essa linha de código garante que a cor da raiz foi mudada.

mainArtigo->color = NEGRO;

}

Função que auxilia nos reapontamentos em relação ao nó passado como parâmetro.

Precisa ser ponteiro para ponteiro caso queiramos remover a raiz.

Modifica os apontamentos de *z* para *y*.

void transferedadRB(Artigo \*\*mainArtigo, Artigo \*z, Artigo \*y, Artigo \*sent){

Se *z* for a raiz.

if(z->dad == sent){

\*mainArtigo = y;

}

Caso *z* não seja a raiz.

else{

Se *z* for filho a esquerda.

if(z == z->dad->left){

z->dad->left = y;

}

Se *z* for filho a direita.

else{

z->dad->right = y;

}

}

y->dad = z->dad;

}

Função que procura o nó sucessor ao nó passado como parâmetro.

Artigo \*sucessor(Artigo \*x, Artigo \*sent){

Ou seja, o laço que se segue, vai procurar o menor nó da subarvore a direita.

while(x->left != sent) x = x->left;

return x;

}

Função que remove um elemento da árvore passado como parâmetro.

O diferencial dessa função é que ela já recebe o nó a ser removido, diferente da função de inserção que recebe um identificador e assim, cria um novo nó. Nessa função, a função já recebe uma referência do nó, caso ele exista. Isso significa que a função de remoção não deve se preocupar com a procura do nó.

void removeRB(Artigo \*\*mainArtigo, Artigo \*z, Artigo \*sent){

Artigo \*x, \*y;

int colorY;

y = z;

Deveremos saber se a remoção desse nó e sua color, afetará a árvore. Por esse fator, guardamos a cor do nó.

colorY = y->color;

Se não existir filho a esquerda, pressupomos que tenha filho a direita já que estamos trabalhando com nó sentinela. Em outras palavras, a diferença de se trabalhar com nó sentinela é que temos a liberdade de usa-lo em condicionais sem nos preocupar com erro te sintaxe ou algo do tipo, já que apesar de tudo, o nó sentinela é um nó que existe (há memória alocada para esse nó).

if(z->left == sent){

x = z->right;

Função auxiliar para os reapontamentos de z, para seu filho a direita.

transferedadRB(&(\*mainArtigo), z, z->right, sent);

}

else{

Aqui temos a certeza que existe um filho a esquerda, diferente da condição anterior. E, então, verificamos se existe filho a direita também.

if(z->right == sent){

x = z->left;

Função auxiliar para os reapontamentos de *z*, para seu filho a esquerda.

transferedadRB(&(\*mainArtigo), z, z->left, sent);

}

Possui filho à esquerda e à direita.

else{

y = sucessor(z->right, sent);

Precisamos antes de qualquer coisa guardar a color de *y*. pois, não podemos comprometer as outras sub árvores do outro lado.

colorY = y->color;

x = y->right;

Significa q y é filho a direita de z.

if(y->dad == z){

Tudo bem se *x* for sentinela e mudar o apontamento para o dad!

x->dad = y;

}

*y* é filho a esquerda de *z*.

else{

Função auxiliar para os reapontamentos de *y*, para seu filho a direita

transferedadRB(&(\*mainArtigo), y, y->right, sent);

y->right = z->right;

y->right->dad = y;

}

Função auxiliar para os reapontamentos de *z*, para *y.*

transferedadRB(&(\*mainArtigo), z, y, sent);

y->left = z->left;

y->left->dad = y;

y->color = z->color;

}

}

Devolve *z* a memória.

free(z);

Caso a cor do nó y, que no caso, ficou no lugar do nó removido seja negro, significa que alguma regra da árvore Rubro-Negra está comprometida.

if(colorY == NEGRO) removeFixUpRB(\*mainArtigo, x, sent); //correção de violação

}

Função auxiliar à função de remoção. Auxilia quando uma regra é quebrada.

Essa função recebe como parâmetro a raiz da árvore, o nó problemático e o sentinela.

void removeFixUpRB(Artigo \*mainArtigo, Artigo \*x, Artigo \*sent){

Artigo \*w;

Se x não for a raiz e sua cor for negra.

while((x != mainArtigo) && (x->color == NEGRO)){

Se x for filho a esquerda de seu pai.

if(x == x->dad->left){

w = x->dad->right;

Se a cor do tio for rubro.

Caso 1.

if(w->color == RUBRO){

printf("\nCaso 1.1");

w->color= NEGRO;

x->dad->color= RUBRO;

Função chamada para realizar rotação a esquerda.

rotationLeft(&mainArtigo, x->dad, sent);

w= x->dad->right;

}

Ou seja, o caso 1, ele não se encerra. Ele é usado para modificar o estado da árvore para que caia no caso 2.

Se o tio tiver filhos negros, ele será modificado para rubro.

Caso 2.

if((w->left->color == NEGRO)&&(w->right->color == NEGRO)){

printf("Caso 1.2");

w->color= RUBRO;

x= x->dad;

}

Se não, vai chegar mais dois casos.

else{

Se o filho a direita do tio for negro.

Caso 3.

if(w->right->color == NEGRO){

printf("\nCaso 1.3");

w->left->color= NEGRO;

w->color= RUBRO;

rotationRight(&mainArtigo, w, sent);

w= x->dad->right;

}

Então o problema se resume ao caso em que a mudança já ocorreu, e, então, deve-se modificar a cor de w e do pai de x. E então, tudo se resume ao caso 4.

printf("\nCaso 1.4");

w->color= x->dad->color;

x->dad->color= NEGRO;

w->right->color= NEGRO;

Faz uma rotação a esquerda.

rotationLeft(&mainArtigo, x->dad, sent);

x= mainArtigo;

}

}

Aqui vai ocorrer a mesma coisa do bloco a cima, porém, pela esquerda.

else{

w = x->dad->left;

if(w->color == RUBRO){

printf("\nCaso 2.1");

w->color= NEGRO;

x->dad->color= RUBRO;

rotationRight(&mainArtigo, x->dad, sent);

w= x->dad->left;

}

if((w->left->color == NEGRO)&&(w->right->color == NEGRO)){

printf("\nCaso 2.2");

w->color= RUBRO;

x= x->dad;

}

else{

if(w->right->color == NEGRO){

printf("\nCaso 2.3");

w->left->color= NEGRO;

w->color= RUBRO;

rotationLeft(&mainArtigo, w, sent);

w= x->dad->left;

}

printf("\nCaso 2.4");

w->color = x->dad->color;

x->dad->color= NEGRO;

w->right->color= NEGRO;

rotationRight(&mainArtigo, x->dad, sent);

x= mainArtigo;

}

}

}

Ao final, *x* deve ter a cor modificada para negra. Isso ocorre pelo fato de que: toda modificação já tenha sido realizada na árvore para o rebalanceamento da árvore em nós e em cores. Porém, o pai nesse momento, precisa que sua cor seja modificada para negra.

if(x != sent) x->color= NEGRO;

}

Função que rotaciona à esquerda a árvore em relação a *w*.

Recebe a raiz da árvore, o nó referência da rotação e o nó sentinela.

void rotationLeft(Artigo \*\*mainArtigo, Artigo \*w, Artigo \*sent){

Artigo \*v = w->right;

w->right = v->left;

Caso o filho a esquerda do filho a direita de w seja um sentinela.

if(v->left != sent) v->left->dad = w;

v->dad = w->dad;

Caso w seja raiz da arvore.

if(w->dad == sent) \*mainArtigo = v;

Aqui trata os casos em que w tem pai.

else{

Caso *w* seja filho a esquerda de seu pai.

if(w == w->dad->left) w->dad->left = v;

Caso *w* seja filho a direita de seu pai

else w->dad->right = v;

}

v->left = w;

w->dad = v;

}

Função que rotaciona à direita a árvore em relação a *w*.

Recebe a raiz da árvore, o nó referência da rotação e o nó sentinela.

void rotationRight(Artigo \*\*mainArtigo, Artigo \*w, Artigo \*sent){

Artigo \*v = w->left;

w->left = v->right;

Se o filho a direita do filho a esquerda de *w* for sentinela.

if(v->right != sent) v->right->dad = w;

v->dad = w->dad;

Se w for raiz da árvore.

if(w->dad == sent) \*mainArtigo = v;

Trata os casos em que w não seja a raiz da árvore.

else{

Caso seja filho a direita de seu pai.

if(w == w->dad->right) w->dad->right = v;

Se w for fiho a esquerda de seu pai.

else w->dad->left = v;

}

v->right = w;

w->dad = v;

}

Função que imprime a árvore com tabulação.

Essa função usa recursividade.

Recebe o nó de um nível, o nível e o sentinela.

void printRB(Artigo \*mainArtigo, int nivel, Artigo \*sent){

int i;

if(mainArtigo != sent){

printRB(mainArtigo->right, nivel+1, sent);

Tabbed printer.

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

if(mainArtigo->color == 0) printf("[%d:R]\n", mainArtigo->id);

else printf("[%d:N]\n", mainArtigo->id);

printRB(mainArtigo->left, nivel+1, sent);

}

}