Árvore rubro negra para registros de artigos científicos.

Guilherme Freitas da Silva: 2017102045
Gustavo Henrique Vago Brunetti: 2016103347 Data: 27/04/21

Introdução:

RBT (red black tree) é uma estrutura de dados do tipo, árvore binária de busca, que consegue se manter quase tão balanceada quanto a árvore AVL e possui a um conceito de cor em cada elemento, um nó pode ser vermelho ou preto, daí vem seu nome. A RBT tem suas operações de inserção, remoção e busca otimizadas podendo chegar no pior caso a uma complexidade igual a O$(log\ n)$ em comparação à uma simples árvore binária que no pior caso tem complexidade igual a O(n)se assemelhando a uma lista ligada.

Em relação a AVL a RBT possui suas operações de inserção e remoção mais eficientes já que é preciso fazer menos rotações para manter a árvore balanceada, já a busca é mais eficiente na AVL pois com ela a árvore tende a ficar mais balanceada, reduzindo assim a altura da raiz até o nó buscado, sendo a altura máxima da AVL = .$(log\ n+\ 1)$ e a altura máxima da RBT = $2.(log\ n+\ 1)$. Deve-se utilizar a RBT para algoritmos que realizam operações de inserção e remoção com maior frequência do que busca e utilizar AVL, caso contrário.

Para se manter balanceada após a remoção ou inserção de um novo elemento a RBT utiliza o conceito de cor para cada nó, 2 tipos de rotações simples, uma à direita e outra à esquerda, além de propriedades que não podem ser violadas e são descritas no próximo tópico.

A RBT implementada neste trabalho é adaptada para receber um tipo genérico de dados, isso quer dizer que o tipo de informação contida em cada elemento da árvore é desconhecido pelo programa RBT. No programa cliente (main.c) foi implementado a definição do dado como uma estrutura de artigo científico e este dado será armazenado por cada nó da RBT. Neste dado existe um valor inteiro único chamado (id) e este valor é utilizado como chave de busca e fator de comparação para ordenar a árvore.

Propriedades Obrigatórias:

1) Todo nó da árvore é vermelho ou preto.
2) A raiz é sempre preta.
3) Todo nó folha (nó externo/NULL) é preto, neste programa foi implementado o nó externo.

4) Se um nó é vermelho, então os seus filhos são pretos (Não existem nós vermelhos consecutivos)
5) Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém o mesmo número de nós pretos

Estrutura de arquivos:

(RBT.h) : Responsável por definir a estrutura de um elemento da árvore, também chamado de nó da árvore e o elemento externo que indica o final de um caminho percorrido através da árvore. Além de conter as assinaturas de todas as funções que o programa RBT.c deve implementar, cada função é acompanhada de comentários que explicam o seu propósito.

(RBT.c) : Responsável por implementar todas as funções contidas na RBT.h, cada função é acompanhada de comentários que auxiliam na descrição de seu funcionamento.

(main.c) : Programa cliente que utiliza o programa RBT para gerenciar os dados informados pelos usuários finais. Além disso, deve definir a estrutura de um artigo científico que será atribuído a cada elemento da árvore e todas as funções que requerem saber a natureza desta informação, como: criar artigo, imprimir artigo, comparar 2 artigos e decidir qual é o maior, menor ou se são iguais e outras funções relacionadas. Para receber valores do usuário final, a main.c foi implementado um menu com as seguintes opções: inserir um novo nó na RBT, deletar nó, buscar nó, imprimir Árvore completa e finalizar o programa.

Explicações do código na prática RBT.c:

Estrutura para representar um elemento da RBT.

```
1) typedef enum {black, red} colorBR;
2)
3) typedef struct no{
4)     void *inf;
5)     struct no *right, *left, *parent;
6)     colorBR color;
7) }no;
```

1) Definição de cores. black = 0 e red = 1.
3) Ponteiro para informação deve ser do tipo void para receber tipo genérico de dados.
4) Ponteiros para subárvore direita e esquerda do nó e ponteiro para o pai do nó.
5) Cor do nó.

Cria um novo elemento externo da RBT que deve ser criado apenas uma vez no programa cliente.

```
1) no *createNoExternal(){
2)     no *newNo = (no*)malloc(sizeof(no));
3)     if(!newNo)
4)         return NULL;
5)     else{
6)         newNo->inf = NULL;
7)         newNo->left = NULL;
8)         newNo->right = NULL;
9)         newNo->parent = NULL;
10)        newNo->color = black;
11)    }
12)    return newNo;
13)}
```

2) Aloca espaço de memória para comportar um novo nó.
3 - 4) Caso não tenha conseguido alocar, retorna null, indicando que a criação do novo nó falhou.
5) Caso a alocação tenha sido concluída com sucesso.
6) O atributo de informação do novo nó recebe NULL pois elemento externo não deve guardar informação.
7 - 10) Os ponteiros para nós devem apontar para valores NULL, já que são o fim do caminho e não podem apontar para outros nós.
12) Retorna o novo nó criado com sucesso.

Cria um novo elemento da RBT com informação.

```
1) no *createNo(void *inf){
2)     no *newNo = (no*)malloc(sizeof(no));
3)     if(!newNo)
4)         return NULL;
5)     else{
6)         newNo->inf = inf;
7)         newNo->left = EXTERNAL;
8)         newNo->right = EXTERNAL;
```

```
9)       newNo->parent = EXTERNAL;
10)         newNo->color = red;
11)        }
12)        return newNo;
13)    }
```

2) Aloca espaço de memória para comportar um novo nó.
3 - 4) Caso não tenha conseguido alocar, retorna null, indicando que a criação do novo nó falhou.
5) Caso a alocação tenha sido concluída com sucesso.
6) O atributo de informação do novo nó recebe a informação fornecida no parâmetro da função. No arquivo main.c este inf é definido como tipo article.
7 - 10) Valores apontam para o nó externo quando o nó é recém criado.
12) Retorna o novo nó criado com sucesso.

Informa se o nó é vermelho.

```
1) int isRed(no *rbt){
2)     if (!rbt) return 0;
3)     if (rbt->color == black) return 0;
4)     return 1;
5) }
```

1 - 2) Retorna 0 se o nó é null ou possui a cor preta.
4) Retorna 1 se a cor for vermelha.

Procura um elemento na RBT que tem a mesma informação com a chave de busca fornecida no parâmetro 'inf'.

```
1) no *search(no *rbt , void *inf, int infoComp(void *, void *)){
2)     if(!rbt || rbt == EXTERNAL)
3)         return NULL;
4)     else if(infoComp(rbt->inf, inf) == 0)
5)         return rbt;
6)     else if(infoComp(rbt->inf, inf) == 1)
7)         return search(rbt->left, inf, infoComp);
8)     else
9)         return search(rbt->right, inf, infoComp);
```

```
10)    }
```

1) O 3º parâmetro é um ponteiro para função “infoComp” e retorna um valor inteiro como resposta, ele é definido no arquivo main.c, e é responsável por comparar 2 artigos e dizer qual é o maior ou se são equivalentes.
2 - 3) Retorna NULL, caso não tenha encontrado o elemento na árvore.
4 - 5) Caso as informações sejam equivalentes, retorna o elemento atual que é o elemento buscado, ou seja busca realizada com sucesso.
6 - 7) Caso rbt>inf seja considerada maior que inf, pelo fator de comparação e se inf pertencer à um nó na RBT, ele estará na subárvore esquerda.
8- 9) Antagônico de (6-7).

Rotações

O rotacionamento troca as posições de certos elementos na RBT para rebalancear a árvore quando alguma propriedade da RBT não é seguida, e são utilizadas nas funções de balanceamento da árvore quando insere ou quando deleta.

Rotação à esquerda.

Acontece quando o nó que causou a rotação necessária está na subárvore direita, da subárvore direita do elemento pivô.

```
1)  void L_RotationRBT(no **root, no *pivo){
2)      no *y = (pivo)->right;
3)
4)      (pivo)->right = y->left;
5)      if (pivo->right != EXTERNAL)
6)          pivo->right->parent = (pivo);
7)      y->parent = (pivo)->parent;
8)
9)      if ((pivo)->parent == EXTERNAL)
10)         *root = y;
11)     else if ((pivo) == (pivo)->parent->left)
12)         (pivo)->parent->left = y;
13)     else
14)         (pivo)->parent->right = y;
15)
16)     y->left = (pivo);
```

```
17)     (pivo)->parent = y;
18) }
```

2) (y) aponta para a subárvore direita do elemento pivô.
4 - 6) Redefine a referência do elemento à direita do pivô, se o novo nó à direita de pivô não for um nó externo, isso indica que deve-se alterar e referência de seu atributo parent para apontar para pivô, já que pivô é o novo pai deste elemento, porém se for um nó externo não precisa pois todos os atributos de externo sempre são = NULL.
7) Atualiza o atributo parent do original filho à direita do pivô.
9 - 14) O original filho à direita do pivô é referenciado pelo seu novo pai.
16 - 17) O pivô se torna filho à esquerda de seu original filho à direita, ou seja (y).

Rotação à direita

Acontece quando o nó que causou a rotação necessária se encontra na subárvore esquerda, da subárvore esquerda do elemento pivô.

```
1) void R_RotationRBT(no **root, no *pivo){
2)     no *y = (pivo)->left;
3)
4)     (pivo)->left = y->right;
5)     if ((pivo)->left != EXTERNAL)
6)         pivo->left->parent = (pivo);
7)     y->parent = (pivo)->parent;
8)
9)     if ((pivo)->parent == EXTERNAL)
10)           *root = y;
11)       else if ((pivo) == (pivo)->parent->left)
12)           (pivo)->parent->left = y;
13)       else
14)           (pivo)->parent->right = y;
15)
16)       y->right = (pivo);
17)       (pivo)->parent = y;
18)   }
```

2) (y) aponta para a subárvore esquerda do elemento pivô.
4 - 6) Redefine a referência do elemento à esquerda do pivô, se o novo nó à esquerda de pivô não for um nó externo, isso indica que deve-se alterar e referência de seu atributo parent para apontar para pivô, já que pivô é o novo pai deste elemento, porém se for um nó externo não precisa pois todos os atributos de externo sempre são = NULL.

7) Atualiza o atributo parent do original filho à esquerda do pivô.
9 - 14) O original filho à esquerda do pivô é referenciado pelo seu novo pai.
16 - 17) O pivô se torna filho à direita de seu original filho à esquerda, ou seja (y).

Transfere o pai do nó rbt para o nó target.

```
1) void transferParent(no **root, no *rbt, no *target){
2)     if (rbt->parent == EXTERNAL)
3)         *root = target;
4)     else if (rbt == rbt->parent->left)
5)         rbt->parent->left = target;
6)     else
7)         rbt->parent->right = target;
8)     target->parent = rbt->parent;
9) }
```

2 - 3) Caso o rbt seja a raiz da árvore atualiza a raiz.
4 - 5) Caso rbt seja filho à esquerda de seu pai, atualiza o filho à esquerda do pai de rbt.
6 - 7) Caso rbt seja filho a direita de seu pai, atualiza o filho à direita do pai de rbt.
8) Atualiza a referência para o pai de target.

Retorna o nó considerado o menor elemento na subárvore de raiz = rbt.

```
1) no *findSmalLestNo(no *rbt){
2)     if(!rbt || rbt == EXTERNAL) return EXTERNAL;
3)     if(!(rbt->left) || rbt->left == EXTERNAL)
4)         return rbt;
5)     else
6)         return findSmalLestNo(rbt->left);
7) }
```

2) Caso não tenha elementos para procurar.
3 - 4) O menor elemento é o próprio rbt, caso ele não tenha filho a esquerda.

5 - 6) O menor elemento é o nó mais à esquerda de rbt, portanto faz uma chamada recursiva para a subárvore esquerda..

Inserir

Como explicado na introdução, o algoritmo para inserir um novo elemento na RBT deve conferir se houve um desbalanceamento a partir de uma quebra de uma propriedade obrigatória para a RBT, e caso necessário efetuar o rebalanceamento da árvore.

1ª parte para inserir um novo elemento na RBT e realizar o rebalanceamento necessário. Utiliza uma função auxiliar.

```
1)    no *insertRBT(no **root, void *inf, int infoComp(void *, void *)){
2)        no *position = *root;
3)        no *positionParent = EXTERNAL;
4)        no *newNo = createNo(inf);
5)
6)        while (position && position != EXTERNAL)
7)        {
8)            if (infoComp(inf, position->inf) == 0){
9)                free(newNo);
10)               return NULL;
11)           }
12)           positionParent = position;
13)
14)           if (infoComp(inf, position->inf) == -1)
15)               position = position->left;
16)           else
17)               position = position->right;
18)       }
19)
20)       newNo->parent = positionParent;
21)
22)       if (positionParent == EXTERNAL)
23)           *root = newNo;
24)       else if (infoComp(inf, positionParent->inf) == -1)
25)           positionParent->left = newNo;
26)       else
```

```
27)             positionParent->right = newNo;
28)
29)         balanceInsert(&(*root), newNo);
30)         return newNo;
31)     }
```

1) 1º parâmetro representa a raiz da RBT com todos os elementos, 2º um ponteiro para o elemento de informação que será utilizado para criar um novo nó e inserir na RBT caso já não exista um elemento com uma informação equivalente, o 3º parâmetro é um ponteiro para função “infoComp” e retorna um valor inteiro como resposta, ele é definido no arquivo main.c, e é responsável por comparar 2 informações e dizer qual é a maior ou se são equivalentes.
2) O position deve guardar a posição correta que o novo elemento será inserido.
3) O positionParent deve guardar a posição correta do pai do novo elemento inserido.
4) Cria um novo nó com a informação passada no parâmetro.
6) Loop para encontrar a posição que o elemento deve ser inserido.
8 - 10) Caso já exista um elemento equivalente ao novo nó, não insere e retorna NULL para indicar isso.
12) Armazena o valor do elemento que deve ser o pai do novo no.
14 - 17) Decide se deve inserir na subárvore esquerda ou direita.
20) Define a referência do valor do ponteiro para o pai do novo no.
22 - 23) Se o elemento da posição do pai do novo nó é um nó externo, isso indica que a árvore está vazia e a raiz da árvore deve apontar para o novo nó.
24 - 25) Se a informação do novo nó for considerada menor que a informação de seu pai, então a sub-árvore esquerda aponta para o novo nó.
26 - 27) Se a informação do novo nó for considerada maior que a informação de seu pai, então a sub-árvore direita aponta para o novo nó.
29) Confere se houve um desbalanceamento na árvore devido a inserção do novo nó e caso necessário faz as modificações para atender as propriedades obrigatórias da RBT e assim é garantido que a árvore permanece balanceada.

2ª parte para lidar com a lógica de inserção balanceada.

```
1) void balanceInsert(no **root, no *newNo){
2)     no *parent, *grandpa, *uncle;
3)
4)     while (isRed(newNo) && isRed((newNo)->parent))
5)     {
6)         parent = (newNo)->parent;
7)         grandpa = parent->parent;
8)
```

```
9)          if (parent == parent->parent->left)
10)             uncle = grandpa->right;
11)         else if (parent == parent->parent->right)
12)             uncle = grandpa->left;
13)
14)         if (isRed(uncle)){
15)             parent->color = black;
16)             uncle->color = black;
17)             grandpa->color = red;
18)             (newNo) = grandpa;
19)         }
20)         else if (parent == parent->parent->left){
21)             if (newNo == newNo->parent->right){
22)                 (newNo)->color = black;
23)                 grandpa->color = red;
24)                 newNo = parent;
25)                 L_RotationRBT(root, newNo);
26)                 newNo = grandpa;
27)                 R_RotationRBT(root, newNo);
28)             }
29)             else{
30)                 parent->color = black;
31)                 grandpa->color = red;
32)                 newNo = grandpa;
33)                 R_RotationRBT(root, newNo);
34)             }
35)         }
36)         else if (parent == parent->parent->right){
37)             if (newNo == newNo->parent->left){
38)                 (newNo)->color = black;
39)                 grandpa->color = red;
40)                 newNo = parent;
41)
42)                 R_RotationRBT(root, newNo);
43)                 newNo = grandpa;
44)                 L_RotationRBT(root, newNo);
45)             }
46)             else{
```

```
47)                 parent->color = black;
48)                 grandpa->color = red;
49)                 newNo = grandpa;
50)                 L_RotationRBT(root, newNo);
51)             }
52)         }
53)     }
54)     (*root)->color = black;
55) }
```

2) Representam respectivamente o pai, avô e tio do novo nó inserido na árvore.
4) Não pode haver um elemento vermelho com pai vermelho.
6 - 7) Atribui os apontamentos do pai e do avô, que foram declarados na linha 2.
9 - 12) Define o tio do novo nó, para saber quem é o tio do novo nó é necessário saber se o pai é filho à esquerda do avô do novo nó, neste caso o tio é filho à direita do avô e caso contrário, ou seja o pai é filho à direita do avô, então o tio é filho à esquerda do avô.
14 - 19) Caso 1 em que o tio do novo nó é rubro. Neste caso é necessário apenas alterar a cor do tio e do pai para vermelho e do avô para preto e o loop se repete para verificar se o avô agora está causando um desbalanceamento.
20 - 21) Caso 2 em que o tio de novo nó é Negro e o novo nó é filho a direita de seu pai.
22 - 27) Altera a cor do novo nó para preto e do avô para vermelho. Além de fazer uma rotação simples à direita e depois à esquerda no elemento pivô e o loop se repete agora com o antigo avô do novo nó, sendo considerado o avô como novo nó que pode ou não causar um desbalanceamento.
29) Caso 3 em que o tio do novo nó é negro e o novo nó é filho a direita de seu pai.
30 - 33) Redefine as cores do pai e do novo nó, além de fazer uma rotação simples à direita no elemento pivô e o loop se repete agora com o antigo avô do novo nó, sendo considerado o avô como novo nó que pode ou não causar um desbalanceamento.
36 - 52) Operações espelhadas de 20 até 35.
53) Sai do loop, ou seja, todas as propriedades da RBT foram satisfeitas e agora a árvore está balanceada, na verdade apenas a raiz pode estar causando um desbalanceamento se esta for vermelha, porém isso é resolvido fora do loop.
54) A raiz é pintada de preto, e assim garantimos que a árvore está realmente balanceada.

Deletar

Assim como a inserção, o algoritmo para deletar um elemento na RBT deve conferir se houve um desbalanceamento, e caso necessário efetuar as rotações para rebalancear a árvore.

1ª parte para deletar um elemento na RBT e realizar o rebalanceamento se necessário. Utiliza uma função auxiliar.

```
1)  no *deleteRBT(no **root, no *delete){
2)      if (!(*root) || !delete) return NULL;
3)      if ((*root) == EXTERNAL || delete == EXTERNAL) return NULL;
4)
5)      no *deleted = delete;
6)      no *auxSuccessor;
7)      no *successor = delete;
8)      colorBR originalColor = successor->color;
9)
10)     if (delete->left == EXTERNAL){
11)         auxSuccessor = (delete)->right;
12)         transferParent((root), delete, delete->right);
13)     }
14)     else if (delete->right == EXTERNAL){
15)             auxSuccessor = delete->left;
16)             transferParent((root), delete, delete->left);
17)     }
18)     else{
19)         successor = findSmalLestNo(delete->right);
20)         originalColor = successor->color;
21)         auxSuccessor = successor->right;
22)
23)         if (successor->parent == delete)
24)             auxSuccessor->parent = successor;
25)         else{
26)             transferParent((root), successor, successor->right);
27)             successor->right = delete->right;
28)             successor->right->parent = successor;
29)         }
30)
31)         transferParent((root), delete, successor);
```

```
32)         successor->left = delete->left;
33)         successor->left->parent = successor;
34)         successor->color = delete->color;
35)     }
36)     if (originalColor == black){
37)         balanceDelete((root), auxSuccessor);
38)     }
39)     return deleted;
40) }
```

1) O 1º parâmetro representa a raiz da árvore e o 2º é o elemento que deve ser deletado, este elemento tem que ser previamente apontado por uma função externa à deleteRBT(), neste projeto isso foi feito no arquivo main.c que realiza uma operação de busca antes de deletar.
2 - 3) Se a árvore for vazia ou se o nó buscado é um nó externo, não realiza a operação de deletar e o valor NULL é retornado para indicar isso.
5) Elemento que deve ser retornado desacoplado da árvore após exclusão deste elemento.
6) Indicará um possível nó que cause um desbalanceamento, ou seja, que tenha uma cor extra preta.
7) O sucessor ocupará a antiga posição do nó que for excluído.
8) Para não perder a cor original de sucessor, pois esta pode ser sobrescrita.
10) Caso o nó a ser deletado tenha filho à esquerda = externo.
11 - 12) Caso simples em que apenas é necessário alterar a referência do pai do elemento deletado para o seu filho à direita, e este filho pode causar um desbalanceamento caso sua cor seja preta.
14 - 16) Caso espelhado de (11 à 12).
18) Caso o elemento a ser deletado possua filho à direita e à esquerda.
19 - 21) Atribui os valores declarados anteriormente em (2-7). É interessante perceber que o único sucessor possível neste caso é o menor elemento na subárvore direita do elemento a ser deletado, para respeitar as regras da própria árvore de busca binária normal em que os filhos à esquerda são menores que o pai e os filhos à direita são maiores que o pai.
23) Caso o sucessor seja filho à direita do nó que deve ser deletado.
24) Atualiza a referência do pai do nó que pode estar causando um desbalanceamento.
25) Caso o filho à direita do nó deletado tenha filho à esquerda.
26 - 28) O sucessor começa a ocupar a posição do nó que deve ser deletado, neste caso transferindo o seu pai para sua subárvore direita e atualizando sua subárvore direita para receber a subárvore direita do elemento que deve ser deletado.
31 - 34) Sucessor ocupa por completo a posição do nó deletado e assumindo até mesmo a cor deste nó.
36 - 37) Se o sucessor tinha color = black, então deve ter havido um desbalanceamento. É como se auxSucessor tivesse uma cor preta e devido a propriedade (1), não pode haver um elemento com cor (duplo preto) ou (preto e vermelho) e isso é tratado na função balanceDelete().
39) O elemento desacoplado da árvore e considerado deletado, é retornado.

2ª parte para lidar com a lógica de remoção balanceada.

Utilizada na remoção, para rebalancear a árvore após o elemento ser deletado.

```
1)    void balanceDelete(no **root, no *rbt){
2)        no *siblingRbt;
3)
4)        while (rbt != (*root) && rbt->color == black)
5)        {
6)            if (rbt == rbt->parent->left){
7)                siblingRbt = rbt->parent->right;
8)                if (siblingRbt->color == red){
9)                    siblingRbt->color = black;
10)                   rbt->parent->color = red;
11)                   L_RotationRBT(root, rbt->parent);
12)                   siblingRbt = rbt->parent->right;
13)               }
14)               if (siblingRbt->left->color == black && siblingRbt->right->color ==
black){
15)                   siblingRbt->color = red;
16)                   rbt = rbt->parent;
17)               }
18)
19)               else{
20)                    if (siblingRbt->right->color == black){
21)                       siblingRbt->left->color = black;
22)                       siblingRbt->color = red;
23)                       R_RotationRBT(root, siblingRbt);
24)                       siblingRbt = rbt->parent->right;
25)                   }
26)                   siblingRbt->color = rbt->parent->color;
27)                   rbt->parent->color = black;
28)                   siblingRbt->right->color = black;
29)                   L_RotationRBT(root, rbt->parent);
30)                   rbt = (*root);
31)               }
32)           }
```

```
33)             else {
34)                 siblingRbt = rbt->parent->left;
35)                 if (siblingRbt->color == red){
36)
37)                     siblingRbt->color = black;
38)                     rbt->parent->color = red;
39)                     R_RotationRBT(root, rbt->parent);
40)                     siblingRbt = rbt->parent->left;
41)                 }
42)                 if (siblingRbt->left->color == black && siblingRbt->right->color ==
black){
43)                     siblingRbt->color = red;
44)                     rbt = rbt->parent;
45)                 }
46)
47)                 else{
48)                      if (siblingRbt->left->color == black){
49)
50)                         siblingRbt->right->color = black;
51)                         siblingRbt->color = red;
52)                         L_RotationRBT(root, siblingRbt);
53)                         siblingRbt = rbt->parent->left;
54)                     }
55)                     siblingRbt->color = rbt->parent->color;
56)                     rbt->parent->color = black;
57)                     siblingRbt->left->color = black;
58)                     R_RotationRBT(root, rbt->parent);
59)                     rbt = (*root);
60)                 }
61)             }
62)         }
63)         rbt->color = black;
64)     }
```

2) Irmão do nó (rbt), ou seja, irmão do nó que está com uma cor negra extra.
4) O loop se repete até que a cor preta extra deixa de existir.
6) Caso (rbt) seja filho a esquerda de seu pai.
7) O irmão, neste caso, será o filho à direita do pai de (rbt).

8) Caso 1: O irmão de (rbt) é vermelho.
9 - 12) Como o irmão de (rbt) deve ter filhos preto, podemos alternar as cores dele e do pai de rbt, em seguida, executar uma rotação à esquerda em no pai de rbt sem violar nenhuma das propriedades obrigatórias da RBT. O novo irmão de (rbt), que é um dos filhos do antigo irmão, ou seja, antes da rotação, agora é preto e, portanto, convertemos o caso 1 em caso 2, 3 ou 4.
14) Caso 2: O irmão de rbt é preto e os 2 filhos deste irmão também são pretos.
15 - 16) Como o irmão de (rbt e seus dois filhos são pretos tiramos uma cor preta de (rbt) e de seu irmão, deixando (rbt) com apenas uma cor preta e deixando seu irmão vermelho. Para compensar a remoção de uma cor preta de rbt e de seu irmão, adicionamos uma cor preta extra ao pai de rbt, que era originalmente vermelho ou preto. Fazemos isso repetindo o loop com o pai de rbt sendo o novo nó rbt.
19) Caso 3: O irmão de rbt eh preto e este irmao tem filho a esquerda vermelho e filho a direita negro.
21 - 24) Podemos trocar as cores do irmão de rbt e seu filho esquerdo e, em seguida, executar uma rotação à direita no irmão de rbt sem violar nenhuma das propriedades da RBT. O novo irmão de rbt agora é um nó preto com um filho vermelho à direita, portanto, transformamos o caso 3 no caso 4.
26 - 30) Caso 4: O irmao de rbt eh negro e este irmao tem filho a direita vermelho.
26 - 30) Fazendo algumas alterações de cor e executando uma rotação simples à esquerda no pai de rbt, podemos remover a cor preta extra em rbt, tornando-o um nó com apenas uma cor preta, sem violar nenhuma das propriedades da árvore RBT. Definir (rbt) como a raiz faz com que o loop termine ao testar a condição de parada.
33 - 59) Operações espelhadas de 6 até 30.
63) Define a raiz como preta e assim todas as propriedades da RBT são garantidas, e podemos afirmar que a árvore está balanceada.

Imprime a RBT, visualmente rotacionada em um ângulo de 90 graus anti-horário.

```
1)     void toStringRBT(no *rbt , int level, void *toStringInfoId(void *)){
2)         int i;
3)         if (rbt && rbt != EXTERNAL){
4)             toStringRBT (rbt->right, level + 1, toStringInfoId) ;
5)             for(i = 0; i < level; i++) printf ("\t");
6)             printf("[");
7)             toStringInfoId(rbt->inf);
8)             printf(": %d]", rbt->color);
9)             printf("\n");
10)            toStringRBT (rbt->left, level + 1, toStringInfoId) ;
11)        }
```

1) O 3º parâmetro é um ponteiro para função “toStringInfoId”, que é definido no arquivo main.c, e é responsável por imprimir a chave de busca escolhida pelo programa cliente que neste projeto é o id do artigo.
2) Contabiliza os níveis que deve imprimir.
4) Chamada recursiva para imprimir os elementos da subárvore à direita.
5) Os elementos são separados na horizontal por um espaço de tab, a quantidade de tabs é definida a partir do nível de cada elemento, a raiz da RBT tem level = 0.
6 - 7) A informação a ser impressa é definida pela chave escolhida no programa cliente. Neste programa essa informação é o id do artigo, portanto a chave impressa é a junção do id do artigo com a cor do nó que o armazena. A chave fica no formato: [id: cor], ex: [33: 1].
9) Os elementos são separados na vertical por uma quebra de linha.
8) Chamada recursiva para imprimir os elementos da subárvore à esquerda.

Explicações do código na prática main.c:

Estrutura para representar uma informação que se refere ao cadastro de um artigo.

```
1) typedef struct article{
2)     int id;
3)     int year;
4)     char *author;
5)     char *title;
6)     char *magazine;
7)     char *DOI;
8)     char *keyword;
9) }article;
10)
```

2) Valor da identidade única do artigo, utilizado como elemento de comparação na RBT.
3) Ano de publicação do artigo.
4 - 8) Vetores de caracteres para guardar informações do artigo.

Cria um elemento de informação.

```
1)    article *createArticle(int id, int year, char author[], char title[], char
magazine[], char DOI[], char keyword[]){
2)        article *newArticle = (article*)malloc(sizeof(article));
3)        char *newAuthor = (char*)malloc(sizeof(char)*(strlen(author)));
4)        char *newTitle = (char*)malloc(sizeof(char)*(strlen(title)));
5)        char *newMagazine = (char*)malloc(sizeof(char)*(strlen(magazine)));
6)        char *newDOI = (char*)malloc(sizeof(char)*(strlen(DOI)));
7)        char *newKeyword = (char*)malloc(sizeof(char)*(strlen(keyword)));
8)
9)        if(!newArticle || !newAuthor || !newTitle || !newMagazine || !newDOI ||
!newKeyword)
10)           return NULL;
11)       else{
12)           strcpy(newAuthor, author);
13)           strcpy(newTitle, title);
14)           strcpy(newMagazine, magazine);
15)           strcpy(newDOI, DOI);
16)           strcpy(newKeyword, keyword);
17)
18)           newArticle->id = id;
19)           newArticle->author = newAuthor;
20)           newArticle->title = newTitle;
21)           newArticle->magazine = newMagazine;
22)           newArticle->DOI = newDOI;
23)           newArticle->keyword = newKeyword;
24)           newArticle->year = year;
25)
26)           return newArticle;
27)       }
28)   }
```

2) Aloca espaço de memória para o novo artigo que será criado e retornado.
3 - 7) Aloca espaço de memória para os atributos que armazenam strings do novo artigo, pegando o tamanho em bytes de char e multiplicando pela quantidade de caracteres presente no atributo passado no parâmetro da função.
9) Se tiver conseguido alocar.
12 - 16) Copia o valor passado no parâmetro para o novo espaço de memória alocado.
18 - 24) Se tiver conseguido alocar, então seta os atributos do novo artigo.

26) Retorna o novo elemento de informação criado com sucesso.

Imprime todos os valores da informação de um artigo em formato JSON.

```
1) void toStringArticle(article *inf){
2)     if (!inf) return;
3)     printf("article {");
4)     printf("\n     ids : %d,", inf->id);
5)     printf("\n     author : %s,", inf->author);
6)     printf("\n     title : %s,", inf->title);
7)     printf("\n     magazine : %s,", inf->magazine);
8)     printf("\n     DOI : %s,", inf->DOI);
9)     printf("\n     keyword : %s,", inf->keyword);
10)    printf("\n     year : %d \n}", inf->year);
11) }
```

Exemplo:

article {
    id : 33,
    author : Guilherme,
    title : Fullstack,
    magazine : Super,
    DOI : ISBN,
    keyword : programacao,
    year : 2020
}

Imprime o valor do id de um artigo.

```
1) void toStringArticleid(article *inf){
2)     printf("%d", inf->id);
3) }
4) void (*toStringArticleid_ptr)(article *) = &toStringArticleid;
```

4) Ponteiro para usar “toStringArticleid” dentro das funções de RBT, que manipulam tipo genérico de dados.

Compara 2 artigos pelo fator de comparação id.

```
1) int articleComp(article *a, article *b){
2)     if (a->id < b->id) return -1;
3)     if (a->id == b->id) return 0;
4)     if (a->id > b->id) return 1;
5) }
6) int (*articleComp_ptr)(article *, article *) = &articleComp;
7)
```

2) Se “a” for menor que “b” retorna -1.
3) Se “a” for igual a b retorna 0.
4) Se “a” for maior que “b” retorna 1.
7) Ponteiro para usar articleComp dentro das funções de RBT, que manipulam tipo genérico de dados.

Limpa o buffer do teclado. Utilizado após um “scanf”.

```
1) void cleanBuffer(){
2)     int c = 0;
3)     while ((c = getchar()) != '\n' && c != EOF) {}
4) }
```

1) Essa função ajuda, por exemplo, caso o usuário digite uma string com espaço por meio de um scanf(), evita que o programa quebre.
3) Lê todos os caracteres até encontrar uma nova linha ou EOF.

Menu de opções “int main()”

Este é o menu interativo que o usuário final pode interagir para realizar operações no programa, como inserir um novo elemento na RBT, deletar um elemento existente na RBT, buscar um elemento existente na RBT, imprimir todos os elementos da RBT, e finalizar o programa. Além de receber orientações de como realizar as operações, também recebe mensagens de resposta para entender o que as suas ações causaram no sistema.

1ª parte. Início

```
1)  EXTERNAL = createNoExternal();
2)      int op;
3)      no *rbt = NULL;
4)
5)      while(op != 5){
6)          printf("\n1 INSERIR UM ARTIGO\n");
7)          printf("\n2 REMOVER UM ARTIGO\n");
8)          printf("\n3 PROCURAR UM ARTIGO \n");
9)          printf("\n4 IMPRIMIR A ARVORE \n");
10)         printf("\n5 SAIR\n");
11)         printf("\nDIGITE UM NUMERO PARA ESCOLHER SUA OPCAO: ");
12)         scanf("%d", &op);
13)         cleanBuffer();
14)
15)         switch (op){
16)             char author[200];
17)             char title[200];
18)             char magazine[200];
19)             char DOI[20];
20)             char keyword[200];
21)             int year, id;
22)             article *inf;
23)             no *deleted, *inserted, *searched;
```

1) No externo da RBT, o pai da raiz e as sub-árvores dos nós que não possuem filhos como elementos de informação à esquerda ou à direita devem apontar para ele.
2) Vai representar a opção escolhida pelo usuário final.
3) Raiz da árvore que irá armazenar os artigos.
2) Estrutura utilizada para inserir, deletar e mostrar dados do artigo.

5) O programa roda em um loop até que o usuário digite 5 como escolha de opção e encerre o programa.
6 - 11) Imprime um menu orientando quais opções o usuário pode escolher para realizar operações no sistema.
12) Captura a opção escolhida pelo usuário final.
13) Limpa o buffer do teclado.
15) Início do menu no sistema, verifica a opção digitada pelo usuário.
16 - 21) Declarações de variáveis utilizadas para guardar o valor que o usuário digitar, relativas a um elemento artigo.
22) Ponteiro para elemento de informação, relativo aos dados que o usuário digitar.
23) Ponteiros para um elemento da RBT, utilizados para retornar ao usuário se as operações de deletar, inserir ou buscar foram realizadas com sucesso.

2ª parte. Caso 1: Insere um elemento na árvore.

```
1)  case 1:
2)      printf("Digite um numero inteiro para representar o id do artigo a ser
    cadastrada: ");
3)      scanf("%d", &id);
4)      cleanBuffer();
5)
6)      printf("Digite o ano de publicacao do artigo: ");
7)      scanf("%d", &year);
8)      cleanBuffer();
9)
10)     printf("Digite o nome do autor do artigo: ");
11)     scanf("%200s", author);
12)     cleanBuffer();
13)
14)     printf("Digite o titulo do artigo: ");
15)     scanf("%200s", title);
16)     cleanBuffer();
17)
18)     printf("Digite o nome da revista que publicou o artigo: ");
19)     scanf("%200s", magazine);
20)     cleanBuffer();
21)
22)     printf("Digite DOI do artigo: ");
23)     scanf("%20s", DOI);
24)     cleanBuffer();
25)
```

```
26)     printf("Digite uma palavra-chave para o artigo: ");
27)     scanf("%200s", keyword);
28)     cleanBuffer();
29)
30)     inf = createArticle(id, year, author, title, magazine, DOI, keyword);
31)     inserted = insertRBT(&rbt, inf, articleComp_ptr);
32)
33)     if( inserted == NULL )
34)         printf("\nErro: id ja existente\n");
35)         else{
36)              printf("\n");
37)              toStringArticle(inserted->inf);
38)              printf("\n");
39)     }
40)     break;
```

1) Caso o usuário digite 1 na escolha de opções.
2 - 4) Orienta o usuário, a como inserir o id do artigo a ser cadastrado, captura o valor digitado e limpa o buffer do teclado.
6 - 8) Orienta o usuário, a como inserir o ano de publicação do artigo a ser cadastrado, captura o valor digitado e limpa o buffer do teclado.
10 - 28) Orienta o usuário, a como inserir os outros valores do artigo a ser cadastrado, captura o valor digitado limitando o tamanho da string específico de cada caso e limpa o buffer do teclado.
30) “inf” aponta para um artigo criado a partir dos dados cadastrados pelo usuário.
31) “inserted” aponta para um novo elemento inserido na RBT se a inserção tiver sido concluída com sucesso, ou aponta para NULL se não conseguir inserir.
17 - 18) Caso já exista um elemento com o “id” igual ao “id” do novo artigo que se deseja inserir na RBT, imprime uma mensagem de erro.
33 - 39) Caso a inserção tenha sido concluída com sucesso, imprime todas as informações do artigo que foi inserido em formato JSON.
40) Encerra o switch.

3ª parte. Caso 2: Remove um elemento da árvore.

```
1)  case 2:
2)      printf("Digite o id do artigo a ser deletado: ");
3)      scanf("%d", &id);
4)      cleanBuffer();
5)
```

```
6)      inf = createArticle(id, 0, "", "", "", "", "");
7)      searched = search(rbt, inf, articleComp_ptr);
8)
9)      if (!searched || searched == EXTERNAL){
10)         printf("\n Este artigo nao existe na RBT\n");
11)         break;
12)     }
13)     deleted = deleteRBT(&rbt, searched);
14)
15)     printf("\n");
16)     if (deleted && deleted != EXTERNAL)
17)         toStringArticle(deleted->inf);
18)     printf("\n");
19)
20)     break
```

1) Caso o usuário digite 2 na escolha de opções.
2 - 4) Orienta o usuário, a como inserir o id do artigo a ser deletado, captura o valor digitado e limpa o buffer do teclado.
6) “inf” aponta para um novo elemento de informação artigo criado com o mesmo id do artigo a ser deletado, para ser passado no parâmetro da função deleteRBT().
7) (searched) aponta para um elemento que deve ser deletado da RBT.
9- 11) Caso não tenha encontrado um elemento na árvore que tenha o mesmo id digitado pelo usuário, imprime uma mensagem de erro e encerra o switch.
13) “deleted” aponta para um elemento deletado da RBT, que está desacoplado se a remoção tiver sido concluída com sucesso, ou aponta para NULL se não conseguir deletar.
15- 18) Caso a remoção tenha sido concluída com sucesso, imprime todas as informações do artigo que foi deletado em formato JSON.
20) Encerra o switch.

4ª parte. Caso 3: Procura um elemento na árvore que tenha um artigo com id igual ao informado pelo usuário.

```
1)  case 3:
2)      printf("Digite o id do artigo a ser buscado: ");
3)      scanf("%d", &id);
4)      cleanBuffer();
5)
6)      inf = createArticle(id, 0, "", "", "", "", "");
7)      searched = search(rbt, inf, articleComp_ptr);
8)
```

```
9)      if ( !searched || searched == EXTERNAL){
10)         printf("\nO artigo com o id = %d, nao foi encontrado\n",
    id);
11)         break;
12)     }
13)     else{
14)         printf("\n");
15)         toStringArticle(searched->inf);
16)         printf("\n");
17)     }
18)     break;
```

1) Caso o usuário digite 3 na escolha de opções.
2 - 4) Orienta o usuário, a como inserir o id do artigo a ser buscado, captura o valor digitado e limpa o buffer do teclado.
6) “inf” aponta para um novo artigo criado com o mesmo id do artigo a ser buscado para ser passado no parâmetro da função “search”.
7) “searched” aponta para um elemento buscado na RBT, se tiver encontrado o elemento buscado, ou aponta para NULL se não encontrar.
9- 11) Caso não tenha encontrado um elemento na árvore que tenha o mesmo id digitado pelo usuário, imprime uma mensagem de erro e encerra o switch.
13- 16) Caso a busca tenha sido concluída com sucesso, imprime todas as informações do artigo que foi encontrado em formato JSON.
18) Encerra o switch.

5ª parte. Caso 4: Imprime a árvore completa.

```
1) case 4:
2)      printf("\n");
3)      toStringRBT(rbt, 0, toStringArticleid_ptr);
4)      break;
```

1) Caso o usuário digite 4 na escolha de opções.
3) Imprime todos os elementos, representados pelo id de cada artigo armazenado na RBT, acompanhado da cor de cada nó em formato de árvore a partir da raiz, visualmente rotacionada em um ângulo de 90 graus anti-horário. Cada chave possui o seguinte formato: [id :cor], exemplo: [33 :1] ou seja, este artigo tem um id = 33 e cor preta
4) Encerra o switch.