



Universidade Federal do Espírito Santo
Centro Universitário Norte do Espírito Santo

Estrutura de Dados II
Avaliação 3 - Árvore rubro-negra

Arthur de Andrade Ferreira
Gabriel Alves Matos

São Mateus
2021

Sumário

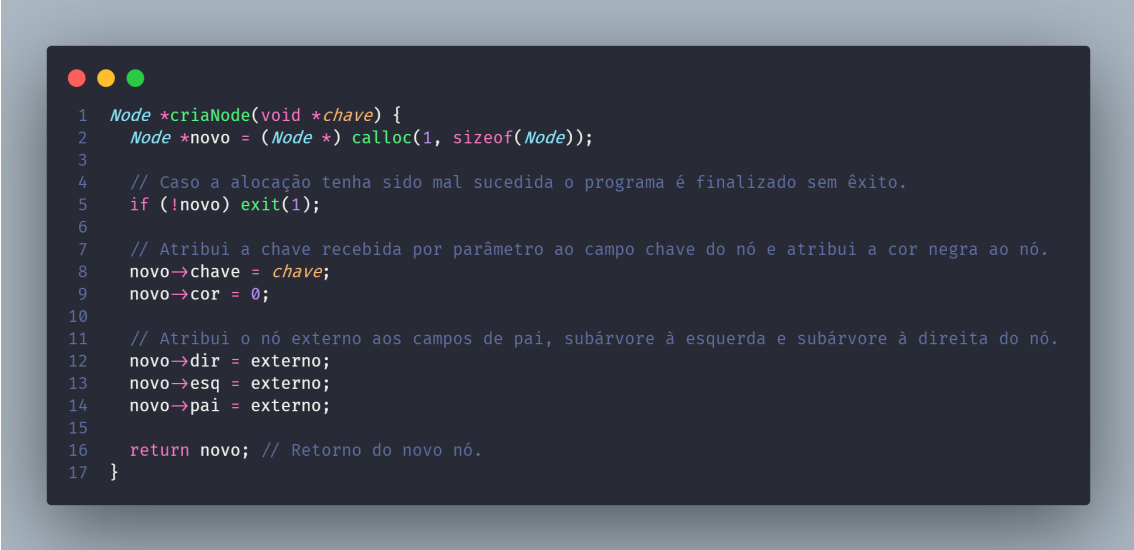
1	Criação de nó	1
1.1	Implementação em C	1
1.2	Funcionamento	1
2	Busca de um nó na árvore	2
2.1	Implementação em C	2
2.2	Funcionamento	2
3	Impressão da árvore	3
3.1	Implementação em C	3
3.2	Funcionamento	3
4	Impressão de um nó da árvore	4
4.1	Implementação em C	4
4.2	Funcionamento	4
5	Liberação da memória alocada para um nó	5
5.1	Implementação em C	5
5.2	Funcionamento	5
6	Liberação da memória alocada para a árvore	6
6.1	Implementação em C	6
6.2	Funcionamento	6
7	Rotações em árvore rubro-negra	7
7.1	Rotação à esquerda	7
7.1.1	Implementação em C	7
7.1.2	Funcionamento	7
7.2	Rotação à direita	8

7.2.1	Implementação em C	8
7.2.2	Funcionamento	8
8	Inserção em árvore rubro-negra	9
8.1	Implementação em C	9
8.2	Funcionamento	9
9	Correção para inserção	10
9.1	Implementação em C	10
9.2	Funcionamento	11
10	Remoção em árvore rubro-negra	12
10.1	Implementação em C	12
10.2	Funcionamento	12
11	Correção para remoção	14
11.1	Implementação em C	14
11.2	Funcionamento	15
12	Busca do sucessor para remoção	16
12.1	Implementação em C	16
12.2	Funcionamento	16
13	Transferência de pai entre dois nós	17
13.1	Implementação em C	17
13.2	Funcionamento	17
14	Estrutura Artigo	18
15	Comparação dos id's de 2 artigos	18
15.1	Implementação em C	18

15.2 Funcionamento	18
16 Impressão do id de um artigo	19
16.1 Implementação em C	19
16.2 Funcionamento	19
17 Impressão dos dados de um artigo	19
17.1 Implementação em C	19
17.2 Funcionamento	20
18 Liberação da memória alocada para um artigo	20
18.1 Implementação em C	20
18.2 Funcionamento	20
19 Função principal	20
19.1 Funcionamento	20

1 Criação de nó

1.1 Implementação em C



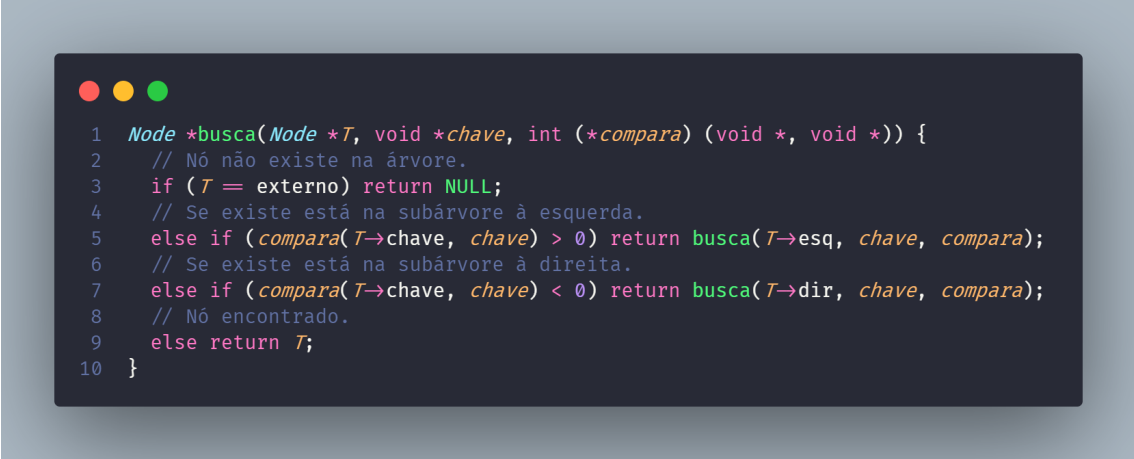
```
1 Node *criaNode(void *chave) {
2     Node *novo = (Node *) calloc(1, sizeof(Node));
3
4     // Caso a alocação tenha sido mal sucedida o programa é finalizado sem êxito.
5     if (!novo) exit(1);
6
7     // Atribui a chave recebida por parâmetro ao campo chave do nó e atribui a cor negra ao nó.
8     novo->chave = chave;
9     novo->cor = 0;
10
11    // Atribui o nó externo aos campos de pai, subárvore à esquerda e subárvore à direita do nó.
12    novo->dir = externo;
13    novo->esq = externo;
14    novo->pai = externo;
15
16    return novo; // Retorno do novo nó.
17 }
```

1.2 Funcionamento

A struct Node é utilizada como um nó da árvore, possui campos para as subárvores, um campo para o nó "pai", um campo "chave" para a chave do nó e um campo "cor" para identificar a cor do nó. A função criaNode recebe um ponteiro que aponta para um valor genérico "chave" a ser atribuído ao campo "chave" do nó a ser criado e retorna o nó criado. Primeiramente é realizada a alocação da memória necessária para um nó e, caso não tenha sido efetuada a alocação com sucesso, então é impressa uma mensagem de erro e o programa é encerrado sem êxito. Posteriormente é atribuída à chave do novo nó o valor genérico recebido como parâmetro, a cor negra é atribuída ao nó, o nó externo é atribuído aos campos de nó à esquerda, nó à direita e nó pai e é retornado o ponteiro do nó criado.

2 Busca de um nó na árvore

2.1 Implementação em C



```
1 Node *busca(Node *T, void *chave, int (*compara) (void *, void *)) {
2     // Nó não existe na árvore.
3     if (T == externo) return NULL;
4     // Se existe está na subárvore à esquerda.
5     else if (compara(T->chave, chave) > 0) return busca(T->esq, chave, compara);
6     // Se existe está na subárvore à direita.
7     else if (compara(T->chave, chave) < 0) return busca(T->dir, chave, compara);
8     // Nó encontrado.
9     else return T;
10 }
```

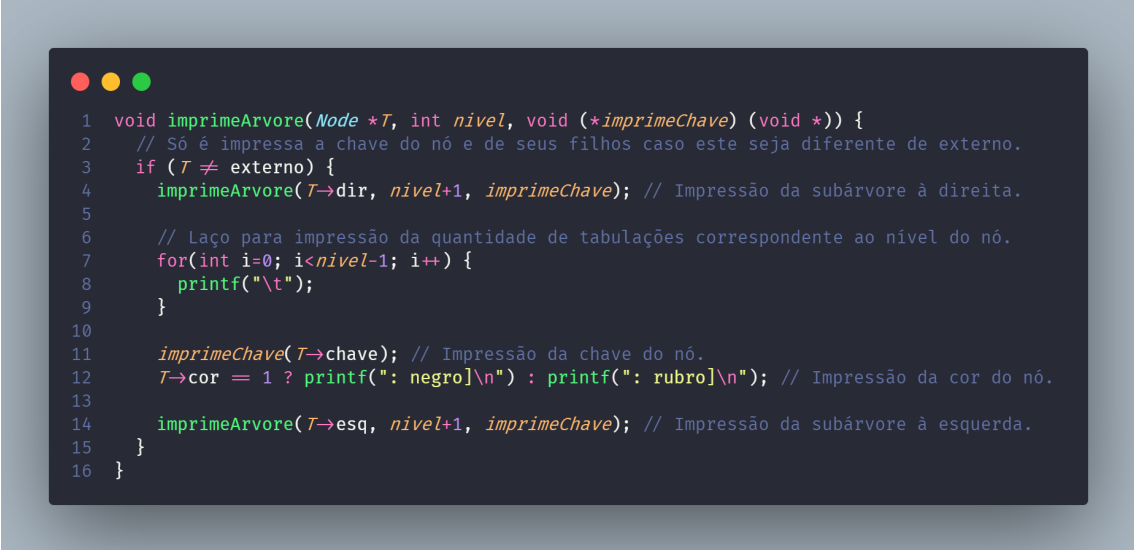
2.2 Funcionamento

A função `buscaArvore` recebe um ponteiro que aponta para o endereço de memória do nó raiz da árvore "T", um ponteiro que aponta para um valor genérico "chave" a ser buscado na árvore e uma função de callback "compara" que é uma função de comparação entre dois valores genéricos implementada pelo cliente da biblioteca, onde esta deve retornar 1 caso a primeira chave seja "maior" que a segunda, 0 caso estas sejam "iguais" e -1 caso a segunda seja "maior" que a primeira (o critério de comparação é decidido pelo cliente).

O caso base é quando foi alcançado um nó folha na árvore ou a árvore é vazia, ou seja, T é igual a externo, de forma que é retornado NULL indicando que a chave não foi encontrada. Caso a chave de T seja maior que a buscada, a função é chamada recursivamente passando como novo nó T o nó à esquerda, pois, se a chave existe na árvore este estará na subárvore à esquerda. Caso a chave de T seja menor que a buscada, a função é chamada recursivamente passando como novo nó T o nó à direita, pois, se a chave existe na árvore este estará na subárvore à direita. Por fim, caso as verificações anteriores falharam, então o nó com chave igual à procurada foi encontrado e este é retornado.

3 Impressão da árvore

3.1 Implementação em C



```
1 void imprimeArvore(Node *T, int nivel, void (*imprimeChave) (void *)) {
2     // Só é impressa a chave do nó e de seus filhos caso este seja diferente de externo.
3     if (T != externo) {
4         imprimeArvore(T->dir, nivel+1, imprimeChave); // Impressão da subárvore à direita.
5
6         // Laço para impressão da quantidade de tabulações correspondente ao nível do nó.
7         for(int i=0; i<nivel-1; i++) {
8             printf("\t");
9         }
10
11         imprimeChave(T->chave); // Impressão da chave do nó.
12         T->cor = 1 ? printf(": negro\n") : printf(": rubro\n"); // Impressão da cor do nó.
13
14         imprimeArvore(T->esq, nivel+1, imprimeChave); // Impressão da subárvore à esquerda.
15     }
16 }
```

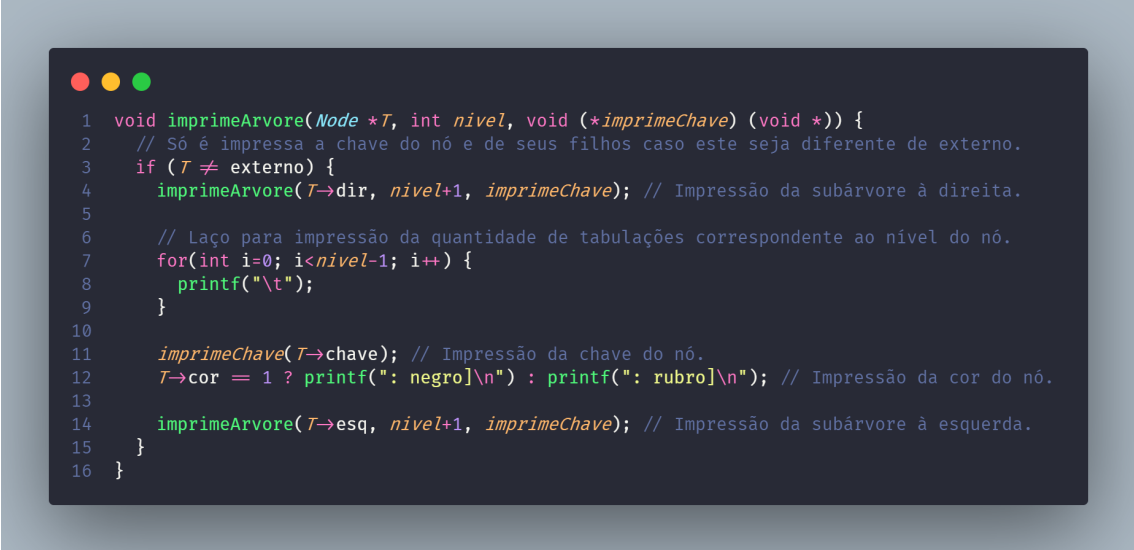
3.2 Funcionamento

A função `imprimeArvore` recebe um ponteiro "T" que aponta para o endereço de memória do nó raiz da árvore, um inteiro "nivel" que representa o nível do nó (sempre receberá 1 na primeira chamada da função) e uma função de callback "imprimeChave" que é uma função de impressão de uma chave genérica implementada pelo cliente da biblioteca. A função `imprimeArvore` é responsável por imprimir uma árvore, percorrendo ela e utilizando a função de callback "imprimeChave" para imprimir a chave de cada nó. Não há retorno de valores.

O caso base é quando foi alcançado um nó folha na árvore ou a árvore é vazia, ou seja, T é igual a externo, de forma que a pilha de recursão não é aumentada. Caso "T" seja diferente de externo, então é impressa a subárvore à direita de "T" chamando a função `imprimeArvore` recursivamente passando seu filho à direita. É impressa uma quantidade de tabulações correspondente ao nível do nó. Posteriormente, é utilizada a função de callback "imprimeChave" para imprimir a chave do nó atual. É impressa também a cor do nó. Por fim, é impressa a subárvore à esquerda de "T" chamando a função `imprimeArvore` recursivamente passando seu filho à esquerda.

4 Impressão de um nó da árvore

4.1 Implementação em C



```
1 void imprimeArvore(Node *T, int nivel, void (*imprimeChave) (void *)) {
2     // Só é impressa a chave do nó e de seus filhos caso este seja diferente de externo.
3     if (T != externo) {
4         imprimeArvore(T->dir, nivel+1, imprimeChave); // Impressão da subárvore à direita.
5
6         // Laço para impressão da quantidade de tabulações correspondente ao nível do nó.
7         for(int i=0; i<nivel-1; i++) {
8             printf("\t");
9         }
10
11         imprimeChave(T->chave); // Impressão da chave do nó.
12         T->cor = 1 ? printf(": negro\n") : printf(": rubro\n"); // Impressão da cor do nó.
13
14         imprimeArvore(T->esq, nivel+1, imprimeChave); // Impressão da subárvore à esquerda.
15     }
16 }
```

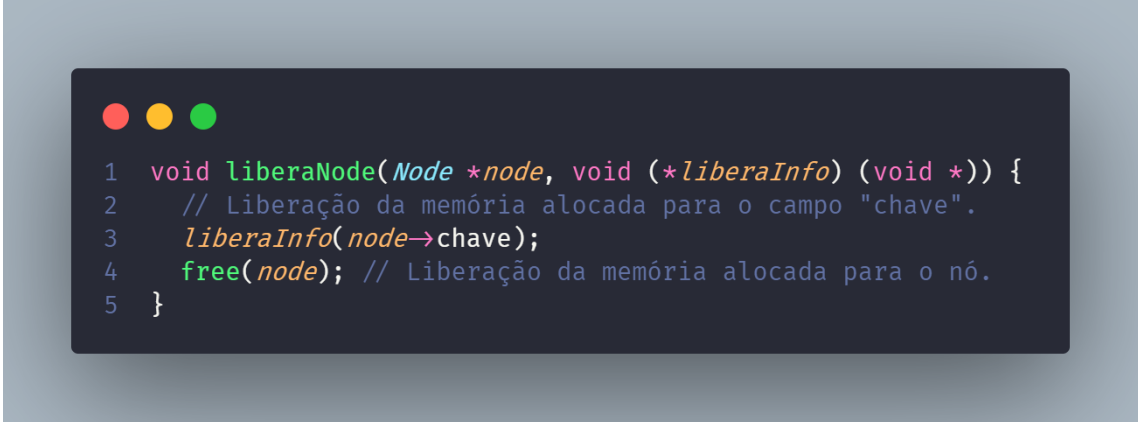
4.2 Funcionamento

A função `imprimeNode` recebe um ponteiro "T" que aponta para o endereço de memória do nó raiz da árvore, um ponteiro "valor" que aponta para um valor genérico a ser impresso, uma função de callback "compara" que é uma função de comparação entre dois valores genéricos implementada pelo cliente da biblioteca, a qual deve retornar 1 caso o primeiro parâmetro seja "maior" que o segundo, 0 caso estes sejam "iguais" e -1 caso o segundo seja "maior" que o primeiro (o critério de comparação é decidido pelo cliente) e uma função de callback "imprimeDados" que é uma função de impressão de um valor genérico implementado pelo cliente da biblioteca. A função `imprimeNode` é responsável por buscar na árvore um nó cujo campo "chave" seja "igual" ao valor genérico "valor" (recebido por parâmetro) e utilizar a função "imprimeDados" para imprimir os campos da estrutura implementada pelo cliente para o nó retornado da busca (caso exista). Não há retorno de valores.

Primeiramente, é efetuada a busca na árvore pelo valor genérico recebido como parâmetro e caso exista um nó com chave "igual" a "valor", então é utilizada a função de callback "imprimeDados" para imprimir os dados da chave genérica do nó.

5 Liberação da memória alocada para um nó

5.1 Implementação em C



```
1 void liberaNode(Node *node, void (*liberaInfo) (void *)) {  
2     // Liberação da memória alocada para o campo "chave".  
3     liberaInfo(node->chave);  
4     free(node); // Liberação da memória alocada para o nó.  
5 }
```

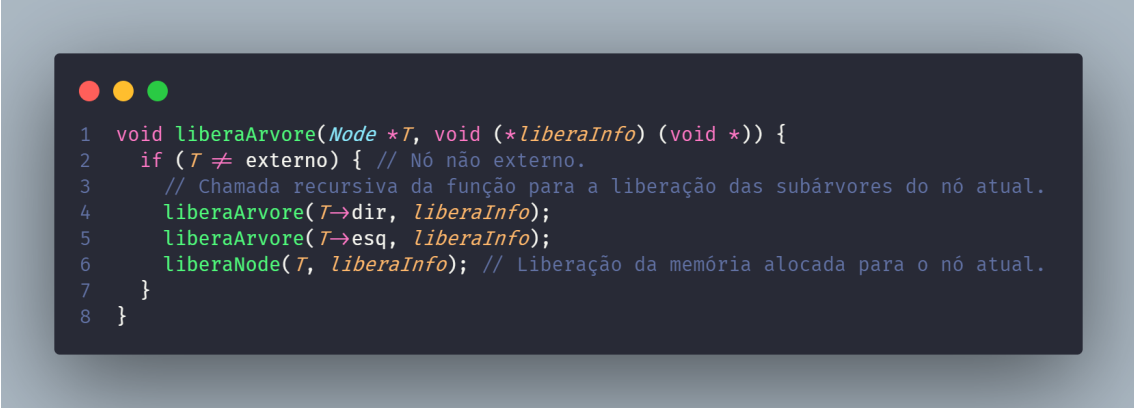
5.2 Funcionamento

A função `liberaNode` recebe um nó `"node"` e uma função de callback `"liberaInfo"` que é uma função que libera a memória alocada para o valor genérico implementado pelo cliente da biblioteca. Ela é responsável por liberar a memória alocada para um nó. Não há retorno de valores.

Primeiramente, é chamada a função de callback `"liberaInfo"` passando o campo `"chave"` de `"node"`. Por fim, é chamada a função `"free"` passando `"node"`.

6 Liberação da memória alocada para a árvore

6.1 Implementação em C



```
1 void liberaArvore(Node *T, void (*liberaInfo) (void *)) {
2     if (T != externo) { // Nó não externo.
3         // Chamada recursiva da função para a liberação das subárvores do nó atual.
4         liberaArvore(T->dir, liberaInfo);
5         liberaArvore(T->esq, liberaInfo);
6         liberaNode(T, liberaInfo); // Liberação da memória alocada para o nó atual.
7     }
8 }
```

6.2 Funcionamento

A função `liberaArvore` recebe um ponteiro que aponta para o endereço de memória do nó raiz da árvore "T" e uma função de callback "liberaInfo" que é uma função que libera a memória alocada para o valor genérico implementado pelo cliente da biblioteca. A função `liberaArvore` é responsável por liberar a memória alocada para os nós de uma árvore, percorrendo ela e utilizando a função "liberaNode" para liberar a memória alocada para cada nó. Não há retorno de valores.

Se "T" for diferente de externo (árvore não vazia) será liberada a memória alocada para os nós da árvore chamando recursivamente a função para as duas subárvores (à esquerda e à direita) e chamando a função "liberaNode" para o nó atual (T). A função de callback "liberaInfo" é passada como parâmetro para a função "liberaNode".

7 Rotações em árvore rubro-negra

7.1 Rotação à esquerda

7.1.1 Implementação em C

```

1 void rotacaoEsquerda(Node **T, Node *w) {
2     Node *v = w->dir; // v recebe o filho à direita de w.
3
4     w->dir = v->esq; // O filho à esquerda de v se torna o filho à direita de w.
5
6     // Caso o filho à esquerda de v seja diferente do nó externo, seu pai agora é w.
7     if (v->esq != externo) {
8         v->esq->pai = w;
9     }
10
11     v->pai = w->pai; // O pai de v recebe o pai de w.
12
13     // Caso o pai de w seja o nó externo (então w é raiz da árvore), v se torna a raiz da árvore.
14     if (w->pai == externo) {
15         (*T) = v;
16     } else { // Caso w não seja raiz, é preciso atualizar o pai de w para que este aponte para v.
17         if (w == w->pai->esq) { // w é filho à esquerda de seu pai, então v se torna filho à esquerda do pai de w.
18             w->pai->esq = v;
19         } else { // w é filho à direita de seu pai, então v se torna filho à direita do pai de w.
20             w->pai->dir = v;
21         }
22     }
23
24     v->esq = w; // w se torna o filho à esquerda de v.
25     w->pai = v; // O pai de w se torna v.
26 }

```

7.1.2 Funcionamento

A função `rotacaoEsquerda` recebe como parâmetros um ponteiro do tipo nó que aponta para o nó raiz da árvore e o nó em que deve ser realizada a rotação "w". O objetivo é realizar uma rotação à esquerda, alterando também o campo "pai" dos nós envolvidos na rotação. Esta função não retorna valores.

Primeiramente é criado um ponteiro auxiliar do tipo nó "v" que recebe o filho à direita de "w". O filho à direita de "w" recebe o filho à esquerda de "v". Caso o filho à esquerda de "v" seja diferente do nó externo, seu campo "pai" recebe "w". É atribuído o campo "pai" do nó "w" ao campo "pai" do nó "v".

Caso o pai de "w" seja o nó externo, ou seja, "w" é raiz da árvore, então a raiz da árvore torna-se "v". Caso contrário, é necessário atualizar o pai de "w" para que este aponte para "v". Se "w" é filho à esquerda de seu pai, "v" torna-se filho à esquerda do pai de "w" e, caso "w" seja filho à direita de seu pai, "v" torna-se filho à direita do pai de "w".

Por último, "w" torna-se filho à esquerda de "v" e "v" se torna pai de "w".

7.2 Rotação à direita

7.2.1 Implementação em C

```
1 void rotacaoDireita(Node **T, Node *w) {
2     Node *v = w->esq; // v recebe o filho à esquerda de w.
3
4     w->esq = v->dir; // O filho à direita de v se torna o filho à esquerda de w.
5
6     // Caso o filho à esquerda de v seja diferente do nó externo, seu pai agora é w.
7     if (v->dir != externo) {
8         v->dir->pai = w;
9     }
10
11     v->pai = w->pai; // O pai de v recebe o pai de w.
12
13     // Caso o pai de w seja o nó externo (então w é raiz da árvore), v se torna a raiz da árvore.
14     if (w->pai == externo) {
15         (*T) = v;
16     } else { // Caso w não seja raiz, é preciso atualizar o pai de w para que este aponte para v.
17         if (w == w->pai->esq) { // w é filho à esquerda de seu pai, então v se torna filho à esquerda do pai de w.
18             w->pai->esq = v;
19         } else { // w é filho à direita de seu pai, então v se torna filho à direita do pai de w.
20             w->pai->dir = v;
21         }
22     }
23
24     v->dir = w; // w se torna o filho à direita de v.
25     w->pai = v; // O pai de w se torna v.
26 }
```

7.2.2 Funcionamento

A função `rotacaoDireita` recebe como parâmetros um ponteiro do tipo nó que aponta para o nó raiz da árvore e o nó em que deve ser realizada a rotação "w". O objetivo é realizar uma rotação à direita, alterando também o campo "pai" dos nós envolvidos na rotação. Esta função não retorna valores.

Primeiramente é criado um ponteiro auxiliar do tipo nó "v" que recebe o filho à esquerda de "w". O filho à esquerda de "w" recebe o filho à direita de "v". Caso o filho à direita de "v" seja diferente do nó externo, seu campo "pai" recebe "w". É atribuído o campo "pai" do nó "w" ao o campo "pai" do nó "v".

Caso o pai de "w" seja o nó externo, ou seja, "w" é raiz da árvore, então a raiz da árvore torna-se "v". Caso contrário, é necessário atualizar o pai de "w" para que este aponte para "v". Se "w" é filho à esquerda de seu pai, "v" torna-se filho à esquerda do pai de "w" e, caso "w" seja filho à esquerda de seu pai, "v" torna-se filho à esquerda do pai de "w".

Por último, "w" torna-se filho à direita de "v" e "v" se torna pai de "w".

8 Inserção em árvore rubro-negra

8.1 Implementação em C

```

1 void RBInsercao(Node **T, Node *q, int (*compara) (void *, void *)) {
2     Node *x = (*T); // Nó auxiliar para percorrer a árvore.
3     Node *v = externo; // O pai do nó a ser inserido.
4
5     // Laço responsável por encontrar posição de inserção do nó na árvore.
6     while(x != externo) {
7         // Já existe nó com a chave procurada, portanto não é inserida novamente.
8         if (compara(x->chave, q->chave) == 0) return;
9
10        v = x; // Atualiza o nó auxiliar que será pai do nó inserido.
11
12        // A chave é "menor" que a chave atual, então o nó deve ser inserido na subárvore à esquerda.
13        if (compara(q->chave, x->chave) < 0) x = x->esq;
14        // A chave é "maior" que a chave atual, então o nó deve ser inserido na subárvore à direita.
15        else x = x->dir;
16    }
17
18    q->pai = v; // O pai do nó inserido será v.
19
20    if (v == externo) { // A árvore é vazia.
21        (*T) = q; // A nova raiz será o nó inserido.
22    } else {
23        if (compara(q->chave, v->chave) < 0) v->esq = q; // Nó inserido à esquerda.
24        else v->dir = q; // Nó inserido à direita.
25    }
26
27    // Atribui o nó externo aos campos de filhos à direita e à esquerda do nó inserido.
28    q->esq = externo;
29    q->dir = externo;
30    q->cor = 0; // O nó inserido recebe a cor rubro.
31    // Chamada de função que corrige possível violação das propriedades de árvore rubro-negra.
32    RBInsertFixup(T, q);
33 }

```

8.2 Funcionamento

A função `RBInsercao` recebe como parâmetros um ponteiro que aponta para a raiz da árvore "`T`", o nó a ser inserido e a função de callback para comparação das chaves implementada pelo cliente da biblioteca. A função não retorna valores.

Primeiramente são criados dois nós auxiliares, um nó "`x`" para percorrer a árvore e um nó "`v`" que será o nó pai do nó inserido. O laço (enquanto "`x`" é diferente do nó externo) é responsável por percorrer a árvore até encontrar a posição de inserção. Dentro do laço, temos as situações: se for encontrado um nó com chave igual à chave do nó a ser inserido, a função é encerrada; caso a chave do nó a ser inserido seja menor que a chave do nó atual, o laço se encaminha para a subárvore à esquerda; caso as verificações anteriores não tenham sido verdadeiras, o laço se encaminha para a subárvore à direita. A cada iteração do laço, o

nó v é atualizado, recebendo o nó atual (x).

Ao final do laço, o campo "pai" do nó inserido recebe o nó " v ". O bloco "if then else" determina a posição de inserção do nó da seguinte forma: se " v " é nó externo (árvore vazia), então a raiz (T) recebe o nó inserido; caso contrário, o nó será inserido à direita se sua chave for "maior" que a de seu pai (v) ou à esquerda se sua chave for "menor" que a de seu pai (v). Por fim, os campos de nó à esquerda e à direita recebem o nó externo, o nó é "pintado" de rubro e é chamada a função de correção para uma possível violação das propriedades, `RBInsertFixup`, explicada posteriormente.

9 Correção para inserção

9.1 Implementação em C

```

1 void RBInsertFixup(Node **T, Node *q) {
2     Node *v; // Pai do nó inserido.
3     Node *w; // Avô do nó inserido.
4     Node *t; // Tio do nó inserido.
5
6     while(q->pai->cor == 0) {
7         v = q->pai; w = v->pai;
8
9         if (v == w->esq) { // O pai do nó inserido é filho à esquerda do avô.
10             t = w->dir;
11
12             if (t->cor == 0) { // Caso 1.
13                 t->cor = 1;
14                 v->cor = 1;
15                 w->cor = 0;
16
17                 q = w; // Uma violação da propriedade de rubro seguido de rubro pode ter ocorrido.
18             } else {
19                 if (q == v->dir) { // O nó inserido é filho à direita de seu pai (caso 2).
20                     q = v; // O nó inserido é trocado com seu pai, o que afetará na mudança de cor depois da condição.
21                     rotacaoEsquerda(T, q);
22                 }
23
24                 q->pai->cor = 1; // O pai recebe a cor negra.
25                 w->cor = 0; // O avô recebe a cor rubro.
26
27                 rotacaoDireita(T, w); // Caso 3.
28             }
29         } else { // O pai do nó inserido é filho à direita do avô.
30             t = w->esq;
31
32             if (t->cor == 0) { // Caso 1.
33                 t->cor = 1;
34                 v->cor = 1;
35                 w->cor = 0;
36
37                 q = w; // Uma violação da propriedade de rubro seguido de rubro pode ter ocorrido.
38             } else {
39                 if (q == v->esq) { // O nó inserido é filho à esquerda de seu pai (caso 4).
40                     q = v; // O nó inserido é trocado com seu pai, o que afetará na mudança de cor depois da condição.
41                     rotacaoDireita(T, v);
42                 }
43
44                 q->pai->cor = 1; // O antigo pai recebe a cor negra.
45                 w->cor = 0; // O antigo avô recebe a cor rubro.
46
47                 rotacaoEsquerda(T, w); // Caso 5.
48             }
49         }
50     }
51
52     (*T)->cor = 1; // A raiz recebe a cor negra.
53 }

```

9.2 Funcionamento

A função `RBInsertFixup` recebe como parâmetros um ponteiro que aponta para a raiz da árvore "T" e o nó que foi inserido "q". A função não retorna valores.

Primeiramente são criados 3 ponteiros auxiliares do tipo nó, para o pai do nó inserido (v), o avô do mesmo (w) e para seu tio (t). Sabemos que a propriedade 5 não foi violada pois o nó inserido é sempre de rubro.

Seguindo, o laço é responsável por tratar a violação da propriedade 4 de árvores rubro-negras (se um nó é vermelho, então os seus filhos são pretos), pois se o pai do nó inserido é rubro temos um nó rubro com filho rubro (lembrando que sempre "pintamos" um nó inserido de rubro).

Dentro do laço, "v" recebe o nó pai de "q" e "w" recebe o nó avô de "q". Caso o pai (v) seja filho à esquerda de "w", então o tio de "q" é o filho à direita de "w" ($t = w \rightarrow \text{dir}$). Sendo assim, são tratados os casos como segue.

Para o caso 1 a cor do nó tio (t) é rubro e a "altura negra" de "v" é igual à de "t", portanto, podemos simplesmente "colorir" o avô (w) de rubro e os irmãos "v" e "t" de negro. Tendo em vista que a alteração da cor do nó "w" pode ter ocasionado a violação da propriedade 4 (o pai de "w" pode ser rubro), fazemos o nó a ser verificado no laço (q) ser agora o nó "w".

Para o caso 2, a cor do nó tio (t) é preto e q é filho à direita de seu pai, portanto, fazemos uma rotação à esquerda em "v" para obtermos o caso 3. O nó "v" vira filho à esquerda do nó "q" e a violação da propriedade 4 se mantém.

Para o caso 3, a cor do nó tio (t) é preto e "q" é filho à esquerda de seu pai. Colorindo o pai de "q" de negro, seu avô de rubro e realizando uma rotação à direita, podemos reestabelecer a propriedade 4.

É importante notar que, quando ocorre o caso 2 é necessário realizar a atribuição do nó "v" ao nó "q", uma vez que na próxima iteração do laço a verificação deve ser falsa (sempre colorimos a nova raiz da subárvore de preto, sendo impossível uma violação na propriedade 4). Se essa atribuição não fosse realizada, temos que "q" se tornaria a raiz da subárvore e a verificação do laço poderia ser verdadeira (a cor o pai da nova raiz pode ser rubro), porém, como a raiz é sempre "pintada" de preto o tratamento realizado estaria equivocado.

Caso o pai (v) seja filho à direita de "w", o tratamento é análogo ao visto para quando "v" é filho à esquerda de "w", porém, simétrico.

10 Remoção em árvore rubro-negra

10.1 Implementação em C

```

1 void RBRemocao(Node **T, Node *z, void (*liberaInfo) (void *)) {
2     Node *x; // Nó que possivelmente será corrigido.
3     Node *y; // Nó que irá ocupar o vértice z.
4     int corDeY;
5
6     y = z;
7     corDeY = y->cor;
8
9     if (z->esq == externo) { // Caso A.
10        x = z->dir;
11        RBTransferePai(T, z, z->dir);
12    } else {
13        if (z->dir == externo) { // Caso B.
14            x = z->esq;
15            RBTransferePai(T, z, z->esq);
16        } else { // Casos C e D (dois filhos).
17            y = RBSucessor(z);
18            corDeY = y->cor;
19            x = y->dir;
20
21            if (y->pai == z) { // y é filho a direita de z.
22                x->pai = y;
23            } else { // y tem um pai diferente de z.
24                RBTransferePai(T, y, y->dir);
25                y->dir = z->dir;
26                y->dir->pai = y;
27            }
28
29            // Troca o pai de y por z.
30            RBTransferePai(T, z, y);
31            y->esq = z->esq;
32            y->esq->pai = y;
33            y->cor = z->cor;
34        }
35    }
36
37    if (corDeY == 1) { // Caso a cor do nó removido era negra.
38        // Correção de violação da propriedade de altura negra, pois foi removido um nó negro.
39        RBDeleteFixup(T, x);
40    }
41
42    liberaNode(z, liberaInfo); // Liberação da memória alocada para o nó removido.
43 }

```

10.2 Funcionamento

A função `RBRemocao` recebe como parâmetros um ponteiro que aponta para a raiz da árvore "`T`", o nó a ser removido "`z`" e uma função de callback que libera a memória alocada para o campo "chave" do nó (implementada pelo cliente). A função não retorna valores.

Primeiramente são criados dois nós auxiliares, um nó "`x`" que guarda o endereço de uma

subárvore de interesse e um nó "y" que ocupará o vértice antes ocupado pelo nó "z", além de uma variável do tipo inteiro que guarda a cor do nó "y" ("corDeY"). No início, o nó "y" recebe o nó "z" e "corDeY", a cor de "y" (cor de "z"). A função trata os casos a partir de condições iniciais da árvore, como segue.

Se o nó a ser removido tem somente filho à direita, basta trocar o nó removido pelo seu filho à direita, fazendo os devidos reapontamentos. Para isso utilizamos a função `RBTransferePai`, explicada em momento oportuno. Para fins de conhecer o nó em que ocorre violação de alguma (ou várias) propriedade(s) de árvore rubro-negra, atribuímos à "x" o filho à direita do nó removido "z". Do contrário, se o nó a ser removido tem somente filho à esquerda, fazemos o mesmo tratamento porém substituindo o nó por seu filho à esquerda.

Para o caso em que o nó tem dois filhos (à esquerda e à direita), é atribuído a "y" o seu sucessor, obtido através da função `RBSucessor` (explicada em outro momento). A variável "corDeY" recebe a cor do sucessor e o ponteiro "x" recebe o filho à direita do sucessor (nó em que pode ter ocorrido violação). Posteriormente, caso o sucessor seja filho à direita do nó removido ($y \rightarrow \text{pai} == z$), basta atribuir y como o pai de "x", pois não há um caminho de nós que precisa ser reapontado partindo do filho à direita do nó a ser removido ao sucessor. Na situação em que o sucessor não é filho à direita de "z", haverá um caminho do filho à direita do nó a ser removido ao sucessor, desta forma, é necessário trocar o pai do sucessor com o seu filho à direita (utilizando a função `RBTransferePai`) e fazer a subárvore à direita do sucessor ser a subárvore à direita do nó removido. Por fim, para os dois casos é necessário trocar o pai do nó à esquerda do nó removido com o pai do nó removido (novamente utilizando a função `RBTransferePai`), fazendo também a subárvore à esquerda do sucessor ser a subárvore à esquerda do nó removido e "colorir" o sucessor com a cor do nó removido (evitando maiores violações, uma vez que as propriedades da árvore eram atendidas naquela região).

Por último, é verificado se a cor do nó removido (`corDeY`) era negra e sendo verdadeiro é necessário tratar a violação da propriedade 5 (altura negra). Também, a memória alocada para o nó é liberada.

É importante observar que, caso o nó a ser removido tenha somente 1 filho, o tratamento será no vértice em que ele estava. Porém, quando o nó tem 2 filhos, o tratamento será no vértice em que o sucessor se encontrava. Isso se deve pois, quando o nó possui somente 1 filho, a cor do vértice em que ele se encontrava é alterada e no caso em que o nó possui 2 filhos a cor do vértice em que se encontrava o sucessor é a alterada. A alteração da cor do vértice em que se encontrava o sucessor e não alteração do vértice em que o nó que foi removido se encontrava é de suma importância, pois, evita violações na região da árvore em volta do vértice em que houve a remoção, trazendo a violação somente para o vértice do sucessor.

11 Correção para remoção

11.1 Implementação em C

```

1 void RBDeleteFixup(Node **T, Node *x) {
2     Node *w; // Irmão de x.
3
4     // Laço responsável por tratar a violação da propriedade de altura negra (no laço, x é sempre um nó duplo negro).
5     while(x != (*T) && x->cor == 1) {
6
7         if (x == x->pai->esq) { // x é filho à esquerda.
8             w = x->pai->dir; // O irmão x é o filho à direita de seu pai.
9
10            if (w->cor == 0) { // Caso 1 (obrigatoriamente os filhos de w são negros).
11                w->cor = 1;
12                x->pai->cor = 0;
13                rotacaoEsquerda(T, x->pai);
14                w = x->pai->dir;
15            }
16
17            if (w->dir->cor == 1 && w->esq->cor == 1) { // Caso 2 (w é negro e os dois filhos são negros).
18                w->cor = 0;
19                x = x->pai;
20            } else {
21
22                if (w->dir->cor == 1) { // Caso 3 (w é negro e somente o filho à direita é negro).
23                    // Fazemos se tornar o caso 4.
24                    w->esq->cor = 1;
25                    w->cor = 0;
26                    rotacaoDireita(T, w);
27                    w = x->pai->dir;
28                }
29
30                // Caso 4 (w é negro e o filho à direita é rubro).
31                w->cor = x->pai->cor;
32                x->pai->cor = 1;
33                w->dir->cor = 1;
34                rotacaoEsquerda(T, x->pai);
35                x = (*T);
36            }
37        } else {
38
39            w = x->pai->esq; // O irmão de x é o filho à esquerda de seu pai.
40
41            if (w->cor == 0) { // Caso 1 (obrigatoriamente os filhos de w são negros).
42                w->cor = 1;
43                x->pai->cor = 0;
44                rotacaoDireita(T, x->pai);
45                w = x->pai->esq;
46            }
47
48            if (w->dir->cor == 1 && w->esq->cor == 1) { // Caso 2 (w é negro e os dois filhos são negros).
49                w->cor = 0;
50                x = x->pai;
51            } else {
52
53                if (w->esq->cor == 1) { // Caso 3 (w é negro e somente o filho à direita é negro).
54                    // Fazemos se tornar o caso 4
55                    w->dir->cor = 1;
56                    w->cor = 0;
57                    rotacaoEsquerda(T, w);
58                    w = x->pai->esq;
59                }
60
61                // Caso 4 (w é negro e o filho à direita é rubro).
62                w->cor = x->pai->cor;
63                x->pai->cor = 1;
64                w->esq->cor = 1;
65                rotacaoDireita(T, x->pai);
66                x = (*T);
67            }
68        }
69    }
70
71    x->cor = 1; // O nó x é pintado de negro.
72 }

```

11.2 Funcionamento

A função `RBDeleteFixup` recebe como parâmetros um ponteiro que aponta para raiz da árvore "T" e o nó a ser corrigida a violação. A função não retorna valores.

Primeiramente, é criado um ponteiro auxiliar que aponta para o nó irmão de "x". Esta função consiste basicamente em um laço responsável por tratar a violação da propriedade 5 (altura negra), sendo que no laço, "x" é sempre um nó duplo negro (com duas cores negras). A condição do laço é que o nó "x" seja diferente da raiz da árvore e sua cor seja negra, uma vez que ao chegar na raiz da árvore basta "pintá-la" de negro obedecendo à propriedade 2 (a raiz é sempre preta).

Dentro do laço, a primeira verificação é se "x" é o filho à esquerda de seu pai, seguindo assim com o tratamento para esse caso. Quando a verificação é verdadeira, fazemos "w" receber o filho à direita do pai de "x".

Se a cor do irmão de "x" é rubro, temos o caso 1. Sabemos que os filhos de "w" são negros. Sendo assim, convertemos este no caso 2 realizando a seguinte sequência: "pintamos" o nó "w" de negro, "pintamos" o pai de "x" de rubro e realizamos uma rotação à esquerda na árvore em relação ao nó pai de x. Fazemos, por último, "w" receber o filho à direita do pai de "x".

Se os filhos de "w" têm a cor negra, temos o caso 2. Basta "pintar" o nó "w", fazendo x receber seu pai (também pai de "w") para que seja tratada a violação agora neste nó na próxima iteração do laço.

Para a situação em que "w" é negro e somente seu filho à direita é negro, temos o caso 3. Para corrigir a violação, seguimos a sequência de mudanças: "pintamos" o filho à esquerda de "w" de negro, "w" de rubro e realizamos uma rotação à direita na árvore em relação ao nó w. Por fim, atribuímos a "w" o filho à direita do pai de "x". Sendo assim, transformamos a situação no caso 4.

O caso 4 ocorre quando "w" é negro e somente seu filho à esquerda é negro. Desta forma, resolvemos a violação seguindo os passos: "pintamos" o nó "w" com a cor do pai de "x", "pintamos" o pai de "x" de negro, "pintamos" o filho à direita de "w" de negro e, por último, realizamos uma rotação à esquerda na árvore em relação ao nó pai de "x". Ao final do caso 4 fazemos "x" receber a raiz da árvore, pois, ao corrigir este caso a altura negra da árvore rubro-negra foi reestabelecida totalmente.

Quando o "x" não é filho à esquerda de seu pai (e sim à direita) o tratamento é análogo ao explicado nos parágrafos anteriores, porém, agora os apontamentos serão "espelhados",

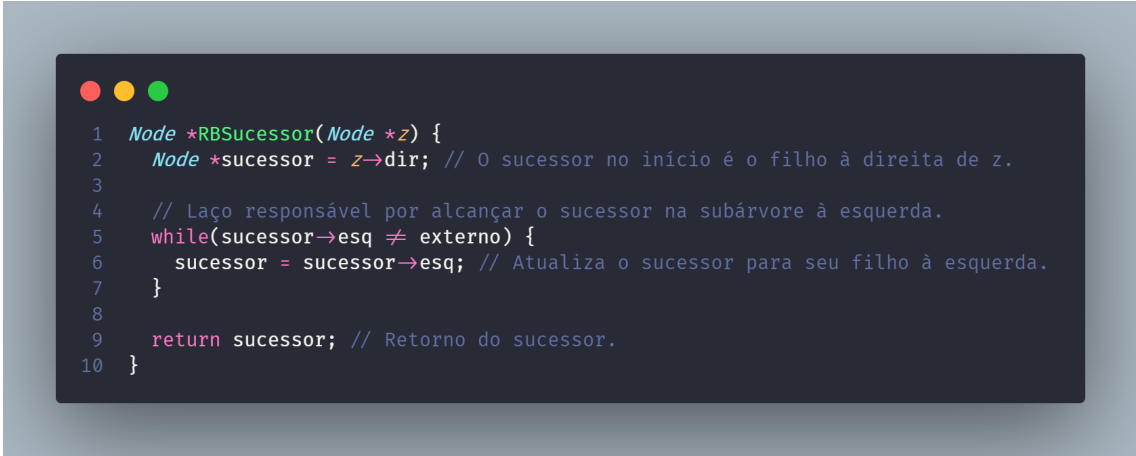
assim como as rotações.

Por fim, após o laço, fazemos a cor de "x" ser negro, isto pois existem duas situações em que o laço termina: o nó "x" corrente é rubro ou a raiz da árvore foi alcançada. Nas duas situações é necessário colorir o nó "x" de negro, primeiramente porque é feita uma consideração de cores pseudo duplas logo, "pintando" o nó de negro reestabelecemos a altura negra, para o segundo caso, o nó "x" é "pintado" de negro pois a propriedade 2 precisa ser mantida (a raiz é sempre preta).

Após todas as iterações possíveis do laço e o tratamento para cada caso encontrado, subindo na árvore em direção à raiz, obtemos a árvore com suas propriedades novamente obedecidas.

12 Busca do sucessor para remoção

12.1 Implementação em C



```
1 Node *RBSucessor(Node *z) {
2     Node *sucessor = z->dir; // O sucessor no início é o filho à direita de z.
3
4     // Laço responsável por alcançar o sucessor na subárvore à esquerda.
5     while(sucessor->esq != externo) {
6         sucessor = sucessor->esq; // Atualiza o sucessor para seu filho à esquerda.
7     }
8
9     return sucessor; // Retorno do sucessor.
10 }
```

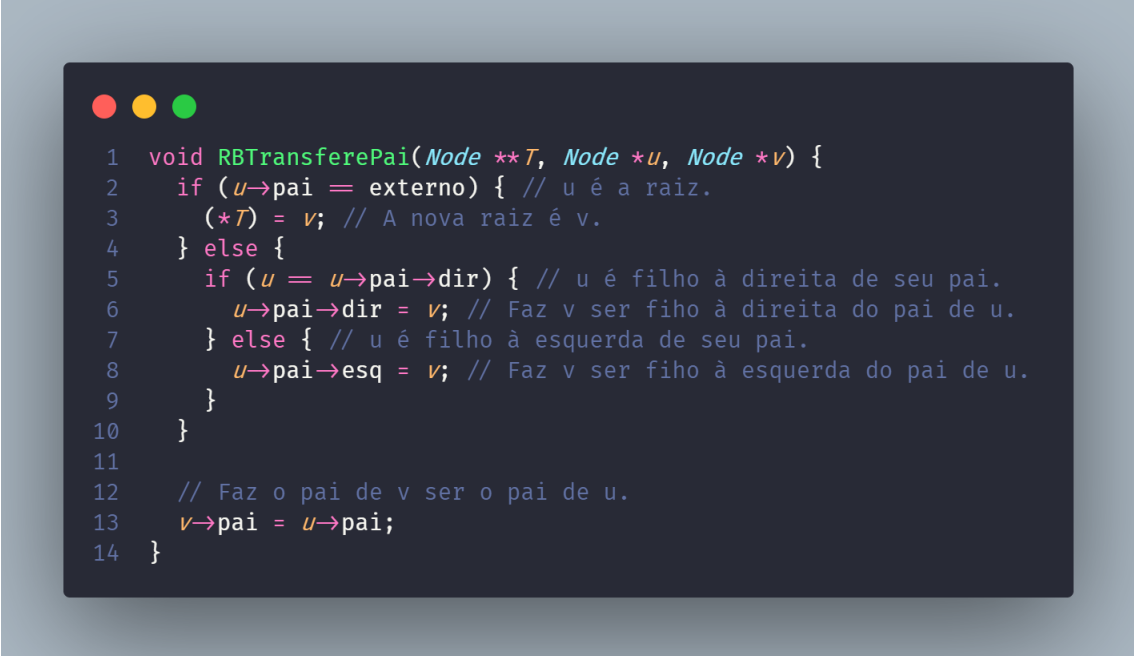
12.2 Funcionamento

A função RBSucessor recebe como parâmetro o nó o qual deve ser encontrado o sucessor. É retornado o nó do sucessor.

Primeiramente, é criado um ponteiro auxiliar "sucessor", do tipo nó, para guardar o nó sucessor buscado. O ponteiro "sucessor" recebe inicialmente o filho à direita do nó recebido por parâmetro. Posteriormente é utilizado um laço "while" para alcançar o sucessor efetivo para o nó que foi recebido. Ao fim do laço, é retornado o nó sucessor.

13 Transferência de pai entre dois nós

13.1 Implementação em C



```
1 void RBTransferePai(Node **T, Node *u, Node *v) {
2     if (u->pai == externo) { // u é a raiz.
3         (*T) = v; // A nova raiz é v.
4     } else {
5         if (u == u->pai->dir) { // u é filho à direita de seu pai.
6             u->pai->dir = v; // Faz v ser fiho à direita do pai de u.
7         } else { // u é filho à esquerda de seu pai.
8             u->pai->esq = v; // Faz v ser fiho à esquerda do pai de u.
9         }
10    }
11
12    // Faz o pai de v ser o pai de u.
13    v->pai = u->pai;
14 }
```

13.2 Funcionamento

A função `RBTransferePai` recebe como parâmetros a raiz da árvore `"T"` e os nós que deseja-se "trocar" os pais, `"u"` e `"v"`. Sendo assim, a função troca o nó `"u"` pelo nó `"v"` na árvore. Não há retorno de valores.

Primeiramente, é verificado se o pai do nó `"u"` é o nó externo e, caso a verificação seja verdadeira, o nó `"v"` se torna a nova raiz da árvore (se o pai do nó é o nó externo, este é raiz da árvore). Caso contrário (`"u"` não é raiz da árvore) é realizado o reapontamento do pai de `"u"` para `"v"`, como segue.

Se `"u"` é filho à direita de seu pai, o filho à direita do pai de `"u"` passa a ser `"v"`. Se não (`"u"` é filho à esquerda de seu pai), o filho à esquerda do pai de `"u"` passa a ser `"v"`.

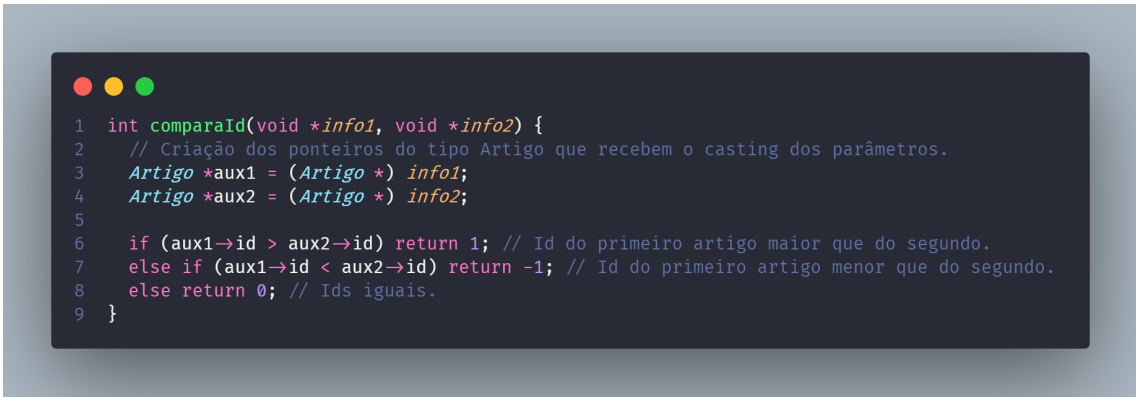
Por último, é feito o reapontamento do campo `"pai"` do nó `"v"` para o nó pai de `"u"`.

14 Estrutura Artigo

A struct `Artigo` representa a entidade "artigo" no sistema. Esta estrutura possui os campos: `id`, que corresponde ao identificador do artigo no sistema; `ano`, que é o ano de publicação do artigo; `autor`, que é o nome do autor; `titulo`, que é o título do artigo; `revista`, que é a revista em que o artigo foi publicado; `DOI`, que é o identificador único para artigos (Digital Object Identifier); `palavraChave`, que corresponde à palavra chave do artigo. É utilizada no código cliente e corresponde ao campo "chave" da estrutura "Node" da biblioteca.

15 Comparação dos id's de 2 artigos

15.1 Implementação em C



```
1 int comparaId(void *info1, void *info2) {
2     // Criação dos ponteiros do tipo Artigo que recebem o casting dos parâmetros.
3     Artigo *aux1 = (Artigo *) info1;
4     Artigo *aux2 = (Artigo *) info2;
5
6     if (aux1->id > aux2->id) return 1; // Id do primeiro artigo maior que do segundo.
7     else if (aux1->id < aux2->id) return -1; // Id do primeiro artigo menor que do segundo.
8     else return 0; // Ids iguais.
9 }
```

15.2 Funcionamento

A função `comparaId` recebe dois ponteiros que apontam para um valor genérico, "info1" e "info2". Esta função retorna 1 caso o id do primeiro artigo seja maior que o do segundo, 0 caso os ids sejam iguais e -1 caso o id do primeiro artigo seja menor que o do segundo.

Primeiramente são criados ponteiros auxiliares do tipo `Artigo` que recebem o resultado do casting dos parâmetros. Se o id do primeiro artigo for maior que o do segundo é retornado 1. Se não se o id do primeiro artigo for menor que o do segundo é retornado -1. Se não (os ids forem iguais) é retornado 0.

16 Impressão do id de um artigo

16.1 Implementação em C

```
1 void imprimeArtigo(void *info) {
2     // Criação do ponteiro do tipo Artigo que recebe o casting do ponteiro do parâmetro.
3     Artigo *aux = (Artigo *) info;
4
5     printf("[id %d", aux->id);
6 }
```

16.2 Funcionamento

A função `imprimeArtigo` recebe um ponteiro que aponta para um valor genérico "info". Não há retorno.

Primeiramente é criado um ponteiro auxiliar do tipo `Artigo` que recebe o resultado do casting de "info". Posteriormente é efetuada a impressão do id do artigo.

17 Impressão dos dados de um artigo

17.1 Implementação em C

```
1 void imprimeDadosArtigo(void *artigo) {
2     // Criação do ponteiro do tipo Artigo que recebe o casting do parâmetro.
3     Artigo *artigoResultado = (Artigo *) artigo;
4
5     printf("\n\tDados: ");
6     printf("\nId: %d", artigoResultado->id);
7     printf("\nAno: %d", artigoResultado->ano);
8     printf("\nAutor: %s", artigoResultado->autor);
9     printf("\nTitulo: %s", artigoResultado->titulo);
10    printf("\nRevista: %s", artigoResultado->revista);
11    printf("\nDOI: %s", artigoResultado->DOI);
12    printf("\nPalavra chave: %s\n", artigoResultado->palavraChave);
13 }
```

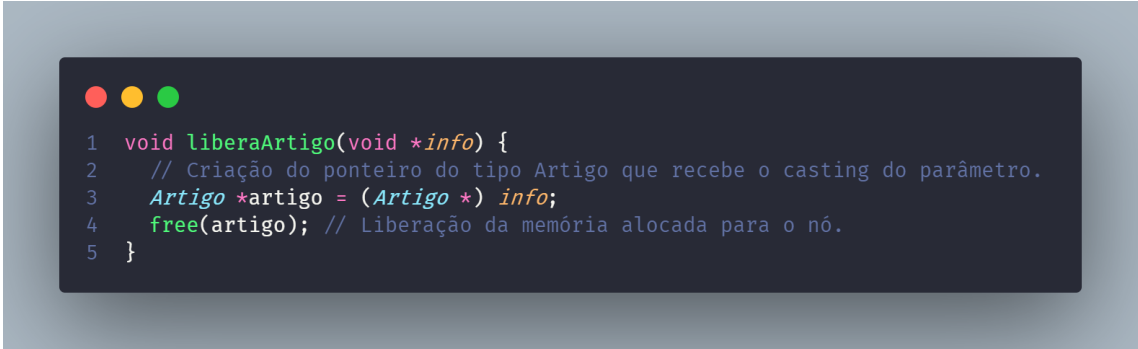
17.2 Funcionamento

A função `imprimeDadosArtigo` recebe um ponteiro que aponta para um valor genérico "artigo". Não há retorno.

Primeiramente é criado um ponteiro auxiliar do tipo `Artigo` que recebe o resultado do casting de "artigo". Posteriormente é efetuada a impressão dos campos do artigo.

18 Liberação da memória alocada para um artigo

18.1 Implementação em C



```
1 void liberaArtigo(void *info) {  
2     // Criação do ponteiro do tipo Artigo que recebe o casting do parâmetro.  
3     Artigo *artigo = (Artigo *) info;  
4     free(artigo); // Liberação da memória alocada para o nó.  
5 }
```

18.2 Funcionamento

A função `liberaArtigo` recebe um ponteiro que aponta para um valor genérico "info". Não há retorno.

Primeiramente é criado um ponteiro auxiliar "artigo" do tipo `Artigo` que recebe o resultado do casting de "info". Posteriormente é efetuada a liberação da memória alocada para o nó, chamando a função "free" passando como parâmetro o ponteiro "artigo".

19 Função principal

19.1 Funcionamento

Na função principal (`main`), primeiramente é efetuada a criação de variáveis auxiliares para as informações do novo artigo (no caso de inserção) e para a escolha do usuário (inserção,

remoção, etc). Posteriormente é realizada a alocação do nó externo (criado como variável global), a atribuição da cor negra ao nó externo e, também, é feito com que a raiz da árvore receba o nó externo.

O principal objetivo da função principal é apresentar ao usuário um menu amigável que utilize as funcionalidades da biblioteca, como inserção, remoção, busca e impressão. Desta forma, é utilizado um laço de repetição "do while" que mostra na tela as opções e realiza a ação de acordo com a escolha do usuário utilizando uma estrutura condicional "switch case". O laço de repetição é finalizado quando o usuário insere o número correspondente à opção "Sair".

As opções disponíveis e os fluxos são os seguintes: caso o usuário escolha inserir um artigo, são pedidos os dados do novo artigo e caso não exista na árvore um artigo com o id informado então é inserido através da função `RBInsercao` (caso já exista um artigo com o id informado, é liberada a memória alocada para o novo artigo e impressa uma mensagem de erro); para a escolha da busca é pedido o id do artigo e caso exista na árvore um artigo com o id informado, os dados dele são impressos utilizando a função `imprimeNode` (caso não exista é impressa uma mensagem de erro); se a escolha for remoção é pedido ao usuário o id do artigo a ser removido e, caso exista na árvore um artigo com o id informado, este é removido utilizando a função `RBRemocao` (caso não exista é impressa uma mensagem de erro); caso a escolha seja a impressão da árvore, se a raiz for vazia (igual a externo) então é impresso que a árvore é vazia, caso contrário é utilizada a função `imprimeArvore` para mostrar a árvore na tela; para a escolha de sair somente é encerrado o laço.

Após o fim do laço que possibilita o usuário utilizar de forma indireta as funcionalidades da biblioteca, é realizada a liberação da memória alocada para todos os nós e para os "artigos" da árvore através da função `"liberaArvore"`. Por fim é liberada a memória alocada para o nó externo utilizando a função `"liberaNode"`, encerrando assim a função principal.