UNIVERSIDADE FEDERAL DO TOCANTINS
CAMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ÁRVORE B E ÁRVORE RUBRO NEGRA

Suayder Milhomem Costa

Palmas
Novembro de 2016

Sumário

1 Introdução

Árvores são uma forma de organizar dados e representa-los de maneira hierárquica. O objetivo de representar dados dessa maneira é uma forma de otimizar problemas de busca. Devido a isso o nosso foco aqui serão as Árvores B e Árvores Rubro Negras,que são dois métodos para resolução desses problemas. Neste artigo serão apresentados conceitos das mesmas, bem como as funções básicas de manipulação que serão as inserções, exclusões, pesquisa e balanceamento, esta ultima é uma função interna da árvore que deve ser implementada para garantir a máxima eficiência possível nas operações.

2 Árvores Rubro-Negras

As árvores rubro-negras (Red-Black Tree) são árvores binárias de busca, ou seja, é balanceada, para manter esse balanceamento ela utiliza o critério de cores, onde há regras para manter a mesma colorida de forma que esteja balanceada. Tais regras são:

1. Todos os nós devem possuir uma cor;
2. A raiz sempre será preta;
3. Todos os nós folhas Nulos são pretos;
4. Um nó vermelho jamais poderá ter um filho vermelho;
5. Partindo de um determinado nó, todos os caminhos até chegar nas folhas possuem um número igual de nós pretos;

Essas propriedades asseguram que a árvore esteja sempre balanceada, pois sempre na inserção ou remoção a árvore é alterada, assim pode ser necessário fazer o ajuste de cores novamente ou fazer as rotações, pois assim pode ser corrigido possíveis violações das propriedades da árvore. Devido a simplicidade em nossos códigos utilizaremos uma variação das árvores rubro-negras que são as árvores rubro-negras caídas à esquerda, ela possui apenas uma propriedade a mais, diz que: Se o nó é vermelho ele sempre será filho esquerdo do pai.

Em relação as árvores AVL, as árvores rubro-negras são mais lentas na operação de busca, contudo mais rápidas nas operações de inserção e remoção. Isso ocorre devido a árvore AVL ser mais balanceada que a rubro-negra, porém para ser tão balanceada é necessário um maior número de rotações nas operações de inserção e remoção, o que já é o motivo das rubro-negras serem mais rápidas nesses processos. Nas structs do código o que aumenta é só uma variável para cor.

```
1        typedef struct no{
2            int info;
3            struct no *right;
4            struct no *left;
5            char collor; //foi escolhido um char para determinar as
                 cores
6        }Tree;
```

2.1 Rotações

As rotações são operações básicas de balanceamento de uma árvore (lembrando que há outras funções para balanceamento) há dois tipos de rotações na árvore rubro-negra, a direita e a esquerda, elas são tem a mesma ideia das rotações que conhecemos das árvores AVL, a diferença é que alteramos a cor dos nós.

```
1     Tree *left-rotation(Tree *root){
2         Tree *tree=root->dir;
3         root->right=tree->left
4         tree->left=root;
5         tree->collor=root->collor; // aqui a cor da nova raiz ser  a
              cor da raiz
6         root->collor='r';//Agora a raiz(que    filho esquerdo) recebe a
              cor vermelha
7         return tree;
8     }
9
10    Tree *right-rotation(Tree *root){
11        Tree *tree=root->left;
12        root->left=tree->right;
13        tree->right=root;
14        tree->collor=root->collor;// aqui a cor da nova raiz ser  a cor
              da raiz
15        root->collor='r';//Agora a raiz(que    filho direito) recebe a
              cor vermelha
16        return tree;
17    }
```

É importante observar que as rotações podem transgredir algumas das propriedades da árvore (as 6 citadas ate o momento), porem nós não estamos preocupados com isso na rotação, outras funções serão adicionadas para corrigir a árvore, e automaticamente corrigir as transgressões feitas pela rotação.

2.2 Demais funções de balanceamento de árvores rubro-negras

Além das funções de rotação, que como ditas anteriormente são funções básicas de balanceamento, temos algumas outras funções que são para auxiliar no balanceamento, essas funções ajudam a manter a integridade da árvore, no quesito da árvore sempre respeitar as regras suas regras de existência, pois, ao inserir ou remover pode ser violada alguma propriedade da árvore, para recupera-las temos as funções abaixo:

```
1     Tree *move_redToleft(Tree *root){
2         swapCollor(&root);
3         //Ao mudar a cor caso apare a dois vermelhos seguidos
4         if(root->right && root->right->left && returnCollor(root->right
              ->left)=='r'){
5             root->right=right_rotation(root->right);
6             root=left_rotation(root);
7             swapCollor(&root);
8         }
9         return root;
10    }
11
12    Tree *move_redToRight(Tree *root){
13        swapCollor(&root);
14        //Mesmo que o caso passado, ao mudar a cor se aparecer dois
              vermelhos seguidos
15        if(root->left && root->left->left && returnCollor(root->left->
              left)=='r'){
16            root=right_rotation(root->right);
17            swapCollor(&root);
18        }
19        return root;
20    }
21
22    Tree balance(Tree **root){
23        //Condi  o garante que um n  vermelho nunca seja filho
              esquerdo
24        if(returnCollor((*root)->right)=='r'){
25            (*root)=left_rotation((*root));
26        }
27        //Caso o filho da direita e neto da esquerda sejam vermelhos
28        if((*root)->left && returnCollor((*root)->right)=='r' &&
29            returnCollor((*root)->left->left)=='r')
30        {
31            (*root)=right_rotation(*root);
32        }
33        //Se os dois filhos forem vermelhos a cor ser  trocada
34        if(returnCollor((*root)->left)=='r' && returnCollor((*root)->
              right)='r'){
35            swapCollor(root);
36        }
37    }
```

Dependendo do caso pode-se usar qualquer uma dessas funções apresentadas ou usar todas, porque há situações que se ele entrar em um determinado if pode ser que bagunce a árvore sendo corrigida no próximo if.

2.3 Inserção

A operação de inserção em uma árvore rubro-negra precisamos analisar os mesmos casos das árvores binárias. O primeiro caso, se a raiz é null, somente inserimos o nó na raiz. O segundo, se o valor a ser inserido é menor que a raiz, aí vamos à sub-árvore a esquerda. O terceiro, se o valor é maior que a raiz, caminhamos para a sub-árvore a direita. Esta função de inserção será implementada recursivamente, isso é feito devido ao fato de quando estiver voltando da recursão poder verificar as propriedades das árvores, o que já é uma grande vantagem. Caso algum nó esteja violando as leis da árvore aí serão aplicadas as funções de correção, mencionadas em tópicos anteriores.

Sempre quando vamos inserir devemos observar que o nó que inserimos sempre iniciará vermelho, e nas correções que ele pode mudar ou não de cor. E também devemos observar os casos de inserções para ser tratados que são a partir de quando já há a raiz na árvore. O primeiro caso é se o pai é preto, como se quer inserir um vermelho basta inserir normalmente, não precisa tratar neste caso. Já o segundo caso verifica se o pai é vermelho, é necessário verificar o tio, se o tio é vermelho somente modificar as cores do avô do pai e do tio, neste caso, se a raiz for avô é necessário voltar sua cor para preto, porém se o tio é preto é necessário fazer uma rotação à direita e trocar as cores deixando o pai preto e os dois filhos vermelhos. O código da inserção a seguir já faz todas estas análises, lembrando que o mesmo é para árvore rubro-negra caída para a esquerda.

```
1     void insert_tree(Tree *root, int info){
2         insert(root, info);
3         if(root!= NULL)
4         root->collor='b';
5     }
6
7     void *insert(Tree *root, int info){
8         if(root==NULL){
9         Tree *tree=(Tree*) malloc(sizeof(Tree));
10        tree->left=NULL;
11        tree->right=NULL;
12        tree->info=info;
13        tree->collor='r';
14        root=tree;
15        return (root);
16    }
17    else{
18        if(root->info>info) //move to left
19            root->left=insert(root->left,info);
20        else//move to right
21            root->right=insert(root->right,info);
```

```
22      }
23      //N o posso ter filho a direita vermelho
24      if(returnCollor(root->right)=='r' && returnCollor(root->left)=='b')
25          root=left_rotation(root);
26      //N o posso ter dois filhos vermelho seguidos
27      if(returnCollor(root->left)=='r' && returnCollor(root->left->left)
            =='r')
28          root=right_rotation(root);
29      if(returnCollor(root->left)=='r' && returnCollor(root->right)=='r')
30          swapCollor(&root);
31      return root;
32  }
```

No código anterior pode ser observado que há duas funções, a primeira é uma função de controle, ela é que deve ser chamada na main, e ela (a primeira) é que faz a chamada da segunda função onde realmente é feira a inserção.

2.4 Remoção

Na remoção devemos analisar os mesmos três casos que analisamos quando vamos fazer a remoção na AVL, a saber:

1. Quando é nó folha;
2. Nó possui apenas 1 filho;
3. Nó possui os dois filhos;

Para tratarmos cada caso desses devemos primeiramente ter atenção com alguns fatores que podem estar sobre a árvore, o primeiro é caso a árvore esteja vazia, porque logicamente não podemos remover em uma árvore que não tenha nada. Outro ponto a se observar é caso seja feita a remoção do último (único) nó da árvore, pois caso isso ocorra devemos lembrar de atribuir o valor null em nossa estrutura. Devemos observar também o balanceamento pois, do mesmo jeito que a cada vez que inserimos em nossa árvore pode ocorrer o desbalanceamento, a medida que removemos também pode ocorrer o mesmo problema, para corrigir isso devemos analisar da mesma maneira que analisamos o balanceamento na inserção, ou seja, a melhor maneira é fazer uma função recursiva para pesquisar o nó que é pra ser movido, pois ao acharmos apenas aplicamos a remoção e retornamos a recursividade verificando as propriedades.

```
1
2       int remove_tree(Tree *root, int info){
3           if(search(root, info)){
4               root=removal(root, info);
```

```
5             if(root!=NULL)
6                 root->collor='b';
7             return 1;
8         }
9         return 0;
10    }
11
12    Tree *removal(Tree *root, int info){
13        if(info<root->info){
14            if(returnCollor(root->left)=='b' && returnCollor(root->left
                  ->left)=='b')
15                root=move_redToleft(root);
16
17            root->left=removal(root->left, info);
18        }
19        else{
20            if(returnCollor(root->left)=='r')
21            root=right_rotation(root);
22            if(info==root->info && root->right==NULL){
23                free(root);
24                return NULL;
25            }
26            if(returnCollor(root->right)=='b' && returnCollor(root->
                  right->left)=='b')
27            root=move_redToRight(root);
28            if(info == root->info){
29                Tree *aux= search_minor(root->right);
30                root->info=aux->info;
31                root->right=remove_minor(root->right);
32            }
33            else
34                root->right=removal(root->right, info);
35        }
36        return balance(root);
37    }
```

Como já mencionado o código de remoção por si só já balanceia a árvore, pois quando é feita a pesquisa para encontrar o nó ela já é feita recursiva para quando retornar já retornar devolvendo as propriedades da árvore. Observe que quando é falado em pesquisa não é a função de pesquisa da primeira função, pois, a mesma serve somente para verificar se o que queremos remover realmente existe.

Na primeira função de remoção apresentada segue o mesmo princípio da inserção, no quesito de gerenciamento da remoção, pois ela primeiro verifica se a função existe, caso a resposta seja afirmativa então ele chamará a função de remoção, a qual Realizará uma pesquisa novamente na árvore, porém esta é a pesquisa recursiva que já foi mencionada

anteriormente, que ao voltar da recursão irá verificar se há algum desbalanceamento. Na função removal quando o nó é encontrado ela chama duas funções desconhecidas ate o momento, a seguir as funções que também são necessárias para conseguirmos fazer uma remoção com sucesso na árvore rubro-negra.

```
1         Tree *search_minor(Tree *root){
2             Tree *aux1=root;
3             Tree *aux2=root->right;
4             while (aux2!=NULL) {
5                 aux1=aux2;
6                 aux2=aux2->left;
7             }
8             return aux1;
9             }
10
11        Tree *remove_minor(Tree *root){
12            if(root->left == NULL){
13                free(root);
14                return NULL;
15            }
16            if(returnCollor(root->left)=='b' && returnCollor(root->left
                  ->left)=='b')
17                root=move_redToleft(root);
18            root->left=remove_minor(root->left);
19            return balance(root);
20        }
```

3 Árvore B

Árvores B são projetadas para trabalhar com dispositivos de armazenamento secundário que contém um alto custo de acesso aos dados. Considerando que a aplicação seja muito grande e que não possa ser guardada somente na memória principal (RAM), estas árvores visam otimizar a entrada e saída de dados. Pois ao trabalhar com armazenamento secundário, enquanto menos acessos ao disco forem feitos melhor será o desempenho do sistema.

Ao implementarmos a árvore b devemos nos atentar a algumas propriedades que a mesma possui.

1. Em uma árvore b a raiz deve ter no mínimo duas e no máximo n sub árvores, onde n é o grau da árvore;
2. Se uma árvore tem grau n, significa que ela vai ter até n-1 elementos na raiz;
3. Caso o nó não seja raiz ou nó folha, o numero minimo de sub árvores que um nó pode ter é $\lceil \frac{n}{2} \rceil - 1$;
4. Todas as folhas sempre estarão no mesmo nível;
5. Os elementos em um nó devem estar todos ordenados;

A declaração da estrutura de uma árvore binária é um pouco diferente das árvores que conhecemos, pois ela contém um vetor de link para os outros elementos encontrados nas árvores.

```
1     typedef struct No{
2             int key[MAX+1], count;
3             struct No *pointer[MAX+1];
4     }T_tree;
```

3.1 Busca

Para buscar uma chave em uma árvore b a operação é bastante semelhante ao da árvore binária de busca, a mesma forma de caminhar na árvore é utilizada, porém ele terá o opção de navegar dentro de um nó, onde há vários elementos.

```
1
2     void search(int info, int *pos, T_tree *root){
```

```
3           if(!root){
4               return;
5           }
6
7           if(info < root->key[1]){
8               *pos = 0;
9           }
10          else{
11              for(*pos = root->count;(key < root->key[*pos] && *pos > 1);
                    (*pos)--);
12              if(info == root->key[*pos]){
13                  printf("value: %d \n", info);
14                  return;
15              }
16          }
17          search(info, pos, root->pointer[*pos]);
18          return;
19      }
```

3.2 Inserção

Como nas demais árvores a inserção de um elemento sempre será em um nó folha, porém por esta ser uma árvore balanceada também, dependendo da inserção a árvore pode desbalancear, para balancearmos ela diferencia das árvores AVL's, pois na mesma utilizamos rotações, a direita e a esquerda, dependendo da situação, nas árvores b nós temos que fazer um processo chamado redistribuição, que consiste basicamente em verificarmos se a folha que queremos inserir está completa, caso esteja a folha será dividida o seu elemento central irá para o nó do pai, caso o nó esteja cheio será redistribuida novamente, e assim recursivamente ate finalizar a inserção, se não estiver basta inserirmos normalmente. A seguir o código de inserção e suas funções.

```
1  /* Adiciona o valor na posi   o apropriada */
2  void addkeyToNo(int key, int pos, T_tree *No,T_tree *filho){
3      int i = No->count;
4      while (i > pos){
5          No->key[i+1] = No->key[i];
6          No->pointer[i+1] = No->pointer[i];
7          i--;
8      }
9      No->key[i+1] = key;
10     No->pointer[i+1] = filho;
11     No->count++;
12 }
13
```

```
14 /* Separa o N  */
15 void spliT_treede (int key, int *pval, int pos, T_tree *no,T_tree *filho
       , T_tree **newNo){
16     int mediana, i;
17
18     if(pos > MIN)
19         mediana = MIN + 1;
20     else
21         mediana = MIN;
22
23     *newNo = (T_tree *)malloc(sizeof(T_tree));
24     i = mediana + 1;
25     while (i <= MAX){
26         (*newNo)->key[i - mediana] = no->key[i];
27         (*newNo)->pointer[i - mediana] = no->pointer[i];
28         i++;
29     }
30     no->count = mediana;
31     (*newNo)->count = MAX - mediana;
32
33     if(pos <= MIN){
34         addkeyToNo(key, pos, no, filho);
35     }
36         else{
37         addkeyToNo(key, pos - mediana, *newNo, filho);
38     }
39     *pval = no->key[no->count];
40     (*newNo)->pointer[0] = no->pointer[no->count];
41     no->count--;
42 }
43
44 /* Coloca o valor solicitado dentro do n  */
45 int setValueInNode(int key, int *pval,T_tree *root, T_tree **filho){
46     int pos;
47     if(!root){
48         *pval = key;
49         *filho = NULL;
50         return 1;
51     }
52
53     if(key < root->key[1]){
54        pos = 0;
55     }
56         else{
57         for(pos = root->count;(key < root->key[pos] && pos > 1); pos--);
58         if(key == root->key[pos]){
59             printf("Duplicacoes nao sao permitidas\n");
```

```
60              return 0;
61          }
62      }
63      if(setValueInNode(key, pval, root->pointer[pos], filho)){
64          if(root->count < MAX){
65              addkeyToNo(*pval, pos, root, *filho);
66          } else{
67              spliT_treede(*pval, pval, pos, root, *filho, filho);
68              return 1;
69          }
70      }
71      return 0;
72  }
73
74  /* Insere um valor na  rvore  */
75  void insert(T_tree **root, int info){
76      int flag, i;
77      T_tree *filho;
78
79      flag = setValueInNode(info, &i, *root, &filho);
80      if(flag)
81          *root = createNo(i, *root, filho);
82  }
```

Nas funções acima a função insert recebe a raiz e o valor que é para ser inserido e chama as demais funções para inserir, em cada função há os comentários sobre o que cada uma faz. Na árvore b, uma árvore só aumenta sua altura quando a raiz é dividida, que é o pior caso que pode ocorrer ao inserir um nó.

3.3 Remoção

A remoção é um caso mais complicado que encontramos nas árvores, e nesta não é diferente, o caso é um pouco mais complicado. Para remover inicialmente devemos realizar uma busca, para encontrarmos o que desejamos remover. O primiro caso, é o caso mais simples, se o nó é folha somente removemos, como acostumados a remover. O segundo o caso é igual ao caso de remoção da AVL, removemos e substituímos o nó pelo imediatamente menor ou o imediatamente maior, então verificamos se o nó contém o mínimo de elementos que um nó pode ter, se os dois irmãos adjacentes (estão na mesma altura) contém mais do que o mínimo que um nó pode ter, devemos fazer uma redistribuição, caso contrario fazer uma concatenação. Se for feita uma concatenação deve-se novamente verificar as propriedades da árvore, caso ainda estejam erradas realizar uma concatenação ou uma redistribuição novamente.

```
1  /* Faz a c pia do sucessor do valor a ser deletado */
2  void copySuccessor(T_tree *root, int pos){
3      T_tree *aux;
4      aux = root->pointer[pos];
5
6      while(aux->pointer[0] != NULL)
7          aux = aux->pointer[0];
8      root->key[pos] = aux->key[1];
9  }
10
11 /* remove o valor de um n  e reorganiza os valores */
12 void removeKey(T_tree *root, int pos){
13     int i = pos + 1;
14     while (i <= root->count){
15         root->key[i-1] = root->key[i];
16         root->pointer[i-1] = root->pointer[i];
17         i++;
18     }
19     root->count--;
20 }
21
22 /* Troca o valor do pai com o do filho da direita */
23 void doRightShift(T_tree *root, int pos){
24     T_tree *aux = root->pointer[pos];
25     int i = aux->count;
26
27     while (i > 0){
28         aux->key[i+1] = aux->key[i];
29         aux->pointer[i+1] = aux->pointer[i];
30     }
31     aux->key[1] = root->key[pos];
32     aux->pointer[1] = aux->pointer[0];
33     aux->count++;
34
35     aux = root->pointer[pos-1];
36     root->key[pos] = aux->key[aux->count];
37     //root->pointer[pos] = x->pointer[x->count];
38     aux->count--;
39     return;
40 }
41
42 /* Troca o valor do pai para com o do filho da esquerda */
43 void doLeftShift(T_tree *root, int pos){
44     int i = 1;
45     T_tree *aux = root->pointer[pos - 1];
46
```

```
47      aux->count++;
48      aux->key[aux->count] = root->key[pos];
49      aux->pointer[aux->count] = root->pointer[pos]->pointer[0];
50
51      aux = root->pointer[pos];
52      root->key[pos] = aux->key[1];
53      aux->pointer[0] = aux->pointer[1];
54      aux->count--;
55
56      while (i <= aux->count){
57          aux->key[i] = aux->key[i + 1];
58          aux->pointer[i] = aux->pointer[i + 1];
59          i++;
60      }
61      return;
62  }
63
64  /* Jun  o de n s, para quando h  um n  vazio ap s o processo de
        remo  o, faz-se a jun  o entre o valor no n  pai
65  mais pr ximo do valor removido com o valor mais pr ximo de um n
        irm o do valor removido. */
66  void mergeNodes(T_tree *root, int pos){
67      int i = 1;
68      T_tree *aux1 = root->pointer[pos], *aux2 = root->pointer[pos - 1];
69
70      aux2->count++;
71      aux2->key[aux2->count] = root->key[pos];
72      aux2->pointer[aux2->count] = aux1->pointer[0];
73
74      while (i <= aux1->count){
75          aux2->count++;
76          aux2->key[aux2->count] = aux1->key[i];
77          aux2->pointer[aux2->count] = aux1->pointer[i];
78          i++;
79      }
80
81      i = pos;
82      while (i < root->count){
83          root->key[i] = root->key[i + 1];
84          root->pointer[i] = root->pointer[i + 1];
85          i++;
86      }
87      root->count--;
88      free(aux1);
89  }
90
91  /* Fun  o para os ajustes necess rios ap s a remo  o de um valor.
```

```
92 faremos a fun   o necess ria de ajuste dependendo da posi   o do
       filho que teve um de seus valores
93 removidos */
94 void adjusT_treede(T_tree *root, int pos){
95     if(!pos){
96         if(root->pointer[1]->count > MIN){
97             doLeftShift(root, 1);
98         }
99                 else{
100            mergeNodes(root, 1);
101        }
102    }
103        else{
104        if(root->count != pos){
105            if(root->pointer[pos - 1]->count > MIN){
106                doRightShift(root, pos);
107            }
108                        else{
109                if(root->pointer[pos + 1]->count > MIN){
110                    doLeftShift(root, pos + 1);
111                } else{
112                    mergeNodes(root, pos);
113                }
114            }
115        }
116                else{
117            if(root->pointer[pos - 1]->count > MIN)
118                doRightShift(root, pos);
119            else
120                mergeNodes(root, pos);
121        }
122    }
123 }
124
125 /* Fun   o usada para descer a  rvore  recursivamente e encontrar o
       valor a ser removido
126 e especificar o processo de remo   o a ser utilizado dependendo da
       quantidade de filhos que o n
127 tem e de onde se encontra o valor a ser removido*/
128 int delKeyFromNo(int key, T_tree *root){
129     int pos, flag = 0;
130     if(root){
131         if(key < root->key[1]){
132             pos = 0;
133         }
134                 else{
135             for(pos = root->count;(key <= root->key[pos] && pos > 0);
```

```
                pos--){
136                 if(key == root->key[pos]){
137                     //flag indica que o valor a ser removido foi
                            encontrado
138                     flag = 1;
139                     break;
140                 }
141                         else{
142                     flag = 0;
143                 }
144             }
145         }
146         if(flag){
147                 /* caso valor a ser removido n o seja de um n  folha
                        */
148            if(root->pointer[pos-1]){
149                 copySuccessor(root, pos);
150                 /* remo  o recursiva do valor no n  sucessor (
                        troca-se o valor a ser removido
151                 pelo valor do n  sucessor, logo o programa busca o
                        valor a ser removido no n  sucessor)*/
152                 flag = delKeyFromNo(root->key[pos], root->pointer[
                        pos]);
153             }
154                     else{
155             removeKey(root, pos);
156             }
157         }
158             else{
159                     /* continua a busca na  rvore  pelo valor dado*/
160          flag = delKeyFromNo(key, root->pointer[pos]);
161         }
162         /* caso o n  onde o valor foi removido tenha um ponteiro a
            direita do valor removido    necess rio checar
163         se n o h  ajustes a serem feitos.*/
164         if(root->pointer[pos]){
165          if(root->pointer[pos]->count < MIN)
166             adjusT_treede(root, pos);
167         }
168     }
169     return flag;
170 }
171
172 /* Fun  o inicial de remo  o de valor da    rvore    B */
173 void deletion(int key,T_tree **search){
174     T_tree *temp;
175     if(!delKeyFromNo(key, *search)){
```

```
176             printf("Valor nao esta presente na arvore\n");
177             return;
178         }
179             else{
180             if((*search)->count == 0){
181                 temp = *search;
182                 *search = (*search)->pointer[0];
183                 free(temp);
184             }
185         }
186         return;
187 }
```

O código tem várias funções, cada função possui o comentário falando qual o papel da cada uma na remoção, mas são as funções da remoção e auxiliares da remoção, que são as de concatenação, ou redistribuição.

4 Conclusão

As árvores apresentadas são mais uma das estrutura de dados que são de suma importância para uma boa base computacional. Não podemos classificar cada árvore apresentada e dizer qual é melhor ou qual é pior, pois cada uma tem sua vantagem dependendo da aplicação, por exemplo as árvores AVL tem vantagens nas árvores rubro-negras na operação de busca, no entanto as rubro-negras são mais vantajosas nas operações de inserção e remoção, ou seja, ao fazer-mos nossa aplicação devemos analisar se a mesma realizará mais operações de inserção e remoção ou se realizará mais operações de busca, ou ainda em questão de árvores b, se ela trabalhará com muito ou poucos dados, então assim podemos escolher uma árvore adequada dependendo da necessidade.

Referências Bibliográficas

[1] L. M. Jayme Luiz Szwarcfiter. Árvores balanceadas. In *ESTRTURA DE DADOS E SEUS ALGORITMOS*, pages 160–169. JC Editora, May 1994.

[2] P. Neubauer. *B-Trees: Balanced Tree Data Structures*, volume 5. 1999.

[3] N. Ziviani. *Projetos de Algorítmos*. Cengage Learning, 2010.