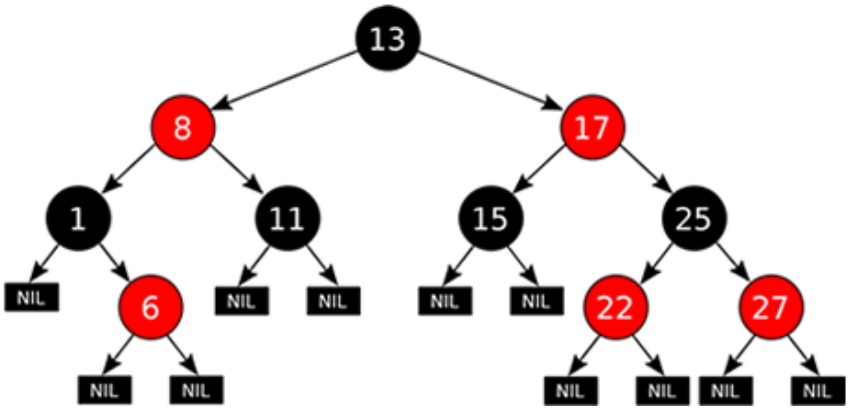
Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Estrutura de Dados
Professor: Bruno Motta
Aluno: Iaslan Nascimento Paulo da Silva

Implementação de Árvore Rubro Negra em C/C++ para a criação de um dicionário

Árvore Rubro Negra

Segundo Cormen (data livro e edição) Árvores Rubro Negras são árvores de busca binaria com um bit extra em cada nó para guardar a informação de cor que pode ser Vermelha ou Preta. Cada nó da arvore contém as seguintes informações: Key ou a informação contida no nó, um apontador para a esquerda, um para a direta e um para o seu pai, além do atributo de cor.



Além das características das árvores binárias de busca as árvores rubro negras possuem características próprias sendo elas:

1 → um nó é vermelho ou preto;
2 → A raiz é preta;
3 → Todas as folhas são pretas;
4 → Ambos os filhos do nó vermelho são pretos; e
5 → Todo caminho de um dado nó para qualquer de seus nós descendentes contém o mesmo número de nós pretos.

Aplicação

Vocês deverão implementar em C/C++ uma árvore rubro-negra que será utilizada para buscar palavras de um dicionário. Você pode assumir que o tamanho máximo das palavras a serem armazenadas é de 20 caracteres. Seu programa deverá ler um arquivo texto cujo nome será especificado na linha de comando, montar a árvore rubro-negra correspondente e iniciar um laço que lê palavras a serem buscadas na árvore. Um exemplo da chamada do programa pode ser visto abaixo.

Arquivos usados no projeto

main.cpp

Arquivo que contém a classe principal.

RBTree.cpp

Arquivo com todas as funções da árvore.

RBElement.cpp

Arquivo que contém os métodos do nó.

Principais funções usadas no projeto

Leitura do Arquivo:

No trecho de código abaixo, vemos uma parte importante do nosso projeto que é a leitura do arquivo contendo as informações que serão inseridas e/ou removidas da nossa árvore. O laço de repetição é iniciado e vai rodar até que o arquivo não tenha mais palavras. Na sequência são iniciadas uma sequência de testes condicionais sendo eles:

1 → Se a opção é 1 e a não está no dicionário adicione usando a função RBInsert;
2 → Se a opção é 1 e a palavra está no dicionário informe ao usuário que a palavra já foi adicionada e pule para a próxima;
3 → Se a opção é 0 e a palavra não está no dicionário informe ao usuário que a palavra não está no dicionário;
4 → Se a opção é 0 e a palavra está no dicionário delete-a usando a função RBDelete;

RBInsert ();

A função RBInsert tem por finalidade a inserção de palavras na árvore. O primeiro passo é verificar se a árvore está vazia, se não a árvore vai ser percorrida até o local disponível para a inserção da palavra, caso não aja nenhum nó ocupado o nó atual se torna o nó raiz. O próximo passo é a coloração do nó que é colorido com a cor Vermelha. Na sequência é chamada a função RBInsertFixup();

RBInsertFixup();

A função *RBInsertFixup* tem por finalidade ajustar as propriedades da árvore rubro negra. Temos 3 casos que são solucionados com o acionamento desta função, caso 1 quando o tio é vermelho, caso 2 Quando o tio do elemento z é preto e elemento z é o filho da direita nesse caso a rotação a esquerda é chamada. O caso 3 quando o tio do elemento z é preto e o elemento é o filho da esquerda, nesse caso a rotação a direita é chamada.

RBDelete();

A função RBDelete tem por finalidade deletar nós da árvore. Um dos passos para a remoção de elementos é a busca, nessa etapa se verifica se o elemento está presente na árvore e assim ele é removido. Na sequência a função RBDeleteFixup é chamada

RBDeleteFixup();

A função RBDeleteFixup tem por finalidade ajustar a estrutura da nossa árvore após as remoções feitas através da função RBDelete, assim mantendo as propriedades da árvore rubro negra. Assim como na função RBInsertFixup esta função resolve alguns casos que mantém as propriedades da árvore. Caso 1 quando o irmão do elemento x é vermelho, caso 2 quando o irmão de x é preto e tem dois filhos pretos, caso 3 quando o irmão de x é preto, o filho da esquerda do irmão é vermelho e o filho da direita do irmão é preto e por fim o caso 4 quando o irmão do x é preto e o filho da direita de

seu irmão é vermelho. A função rotação a esquerda é chamada para os casos 1 e 4 enquanto a função rotação a direita é chamada no caso 3.

LeftRotate();

A função LeftRotate tem por finalidade aplicar uma rotação a esquerda na árvore. Os procedimentos de rotação servem para corrigir e/ou ajustar elementos que podem vir a estar violando as propriedades da árvore.

RightRotate();

A função RightRotate tem por finalidade aplicar uma rotação a direita na árvore. Os procedimentos de rotação servem para corrigir e/ou ajustar elementos que podem vir a estar violando as propriedades da árvore.

TreeSearch();

A função TreeSearch tem por finalidade fazer buscas na árvore atrás de um determinado elemento de interesse do programa.

RBCheck();

A função tem por finalidade imprimir cada nó da árvore com informações como sua chave, chave de seu pai, sua cor , sua altura negra, a chave de seus filhos.

RBPrint();

A função RBPrint tem por finalidade exibir a estrutura final da árvore com seus nós de forma ordenada.

Resultados

Resultados para a base disponibilizada.

->teste
----------palavra nao existe----------

->abuso
----------Inserindo a palavra no dicionario --------

->carro
----------Inserindo a palavra no dicionario --------

->doce
----------Inserindo a palavra no dicionario --------

->gola
----------palavra nao existe----------

->gola
----------Inserindo a palavra no dicionario --------

->palha ├º o
----------Inserindo a palavra no dicionario --------

->taturana
----------Inserindo a palavra no dicionario --------

->pacote
----------Inserindo a palavra no dicionario --------

->bolha
----------Inserindo a palavra no dicionario --------

->fussura
----------Inserindo a palavra no dicionario --------

->batata
----------Inserindo a palavra no dicionario --------

->estrela
----------Inserindo a palavra no dicionario --------

->taturana

----------Deletando a palavra do dicionario -------

abuso batata bolha carro doce estrela fussura gola pacote palha ├ º o (palha ├ º o,taturana,RED,1,NIL,NIL)
->cataplana
----------Inserindo a palavra no dicionario --------

->cerveja
----------Inserindo a palavra no dicionario --------

->zebra
----------Inserindo a palavra no dicionario --------

->lis
----------Inserindo a palavra no dicionario --------

->almirante
----------Inserindo a palavra no dicionario --------

->elefante
----------Inserindo a palavra no dicionario --------

->espa ├º o
----------Inserindo a palavra no dicionario --------

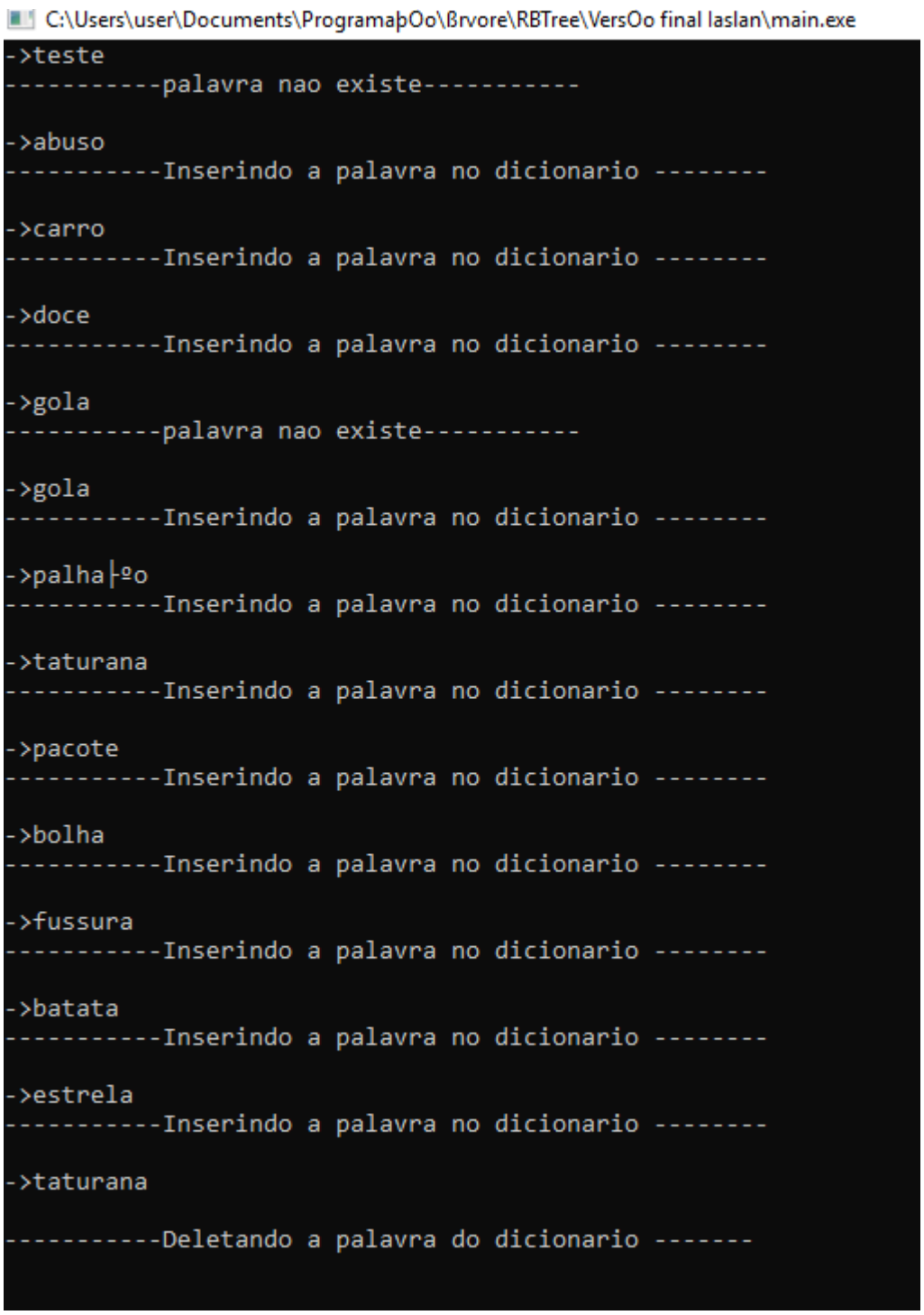
->estrela
----------Palavra ja existente no dicionario---------

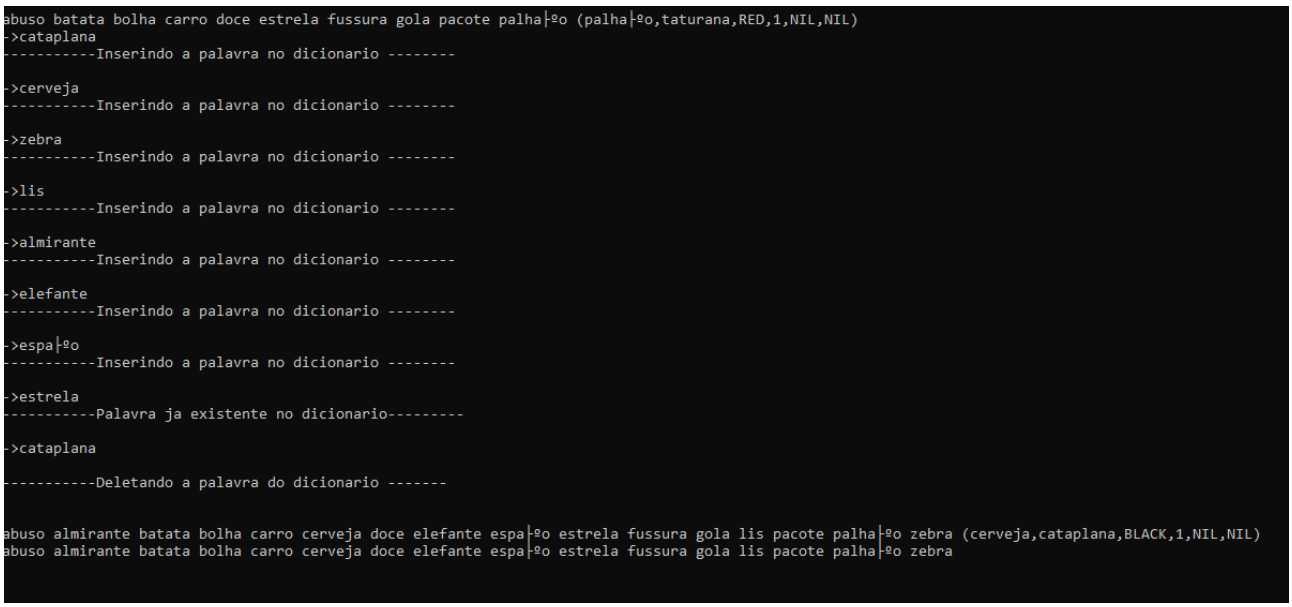
->cataplana

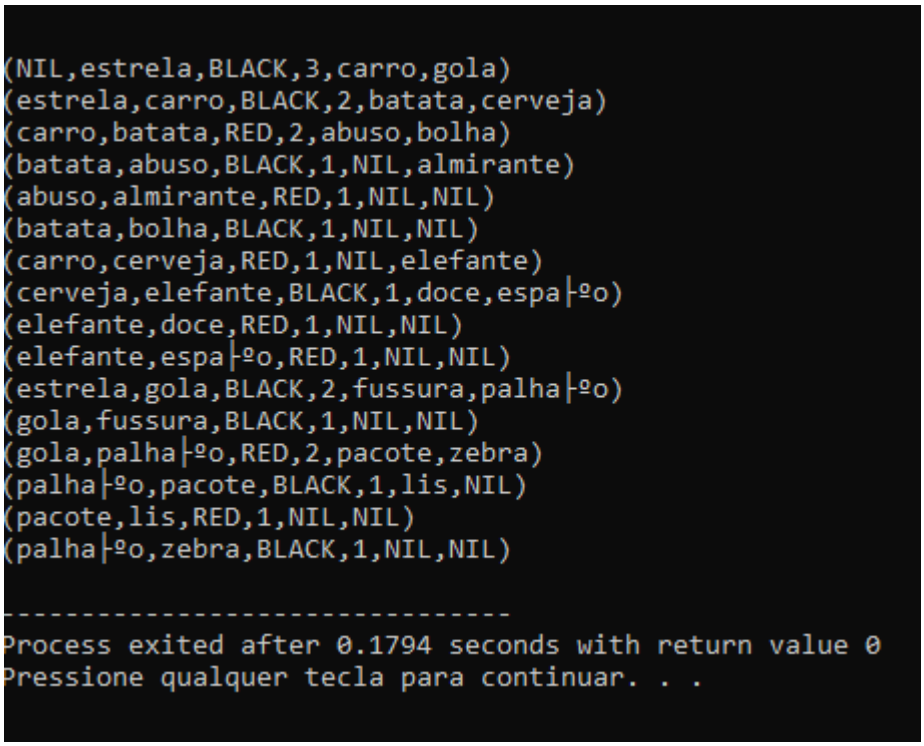
----------Deletando a palavra do dicionario -------

abuso almirante batata bolha carro cerveja doce elefante espa ├º o estrela fussura gola lis pacote palha ├º o zebra (cerveja,cataplana,BLACK,1,NIL,NIL)
abuso almirante batata bolha carro cerveja doce elefante espa ├º o estrela fussura gola lis pacote palha ├º o zebra

(NIL,estrela,BLACK,3,carro,gola)
(estrela,carro,BLACK,2,batata,cerveja)
(carro,batata,RED,2,abuso,bolha)
(batata,abuso,BLACK,1,NIL,almirante)
(abuso,almirante,RED,1,NIL,NIL)
(batata,bolha,BLACK,1,NIL,NIL)
(carro,cerveja,RED,1,NIL,elefante)
(cerveja,elefante,BLACK,1,doce,espa ├º o)
(elefante,doce,RED,1,NIL,NIL)
(elefante,espa ├º o,RED,1,NIL,NIL)
(estrela,gola,BLACK,2,fussura,palha ├º o)
(gola,fussura,BLACK,1,NIL,NIL)
(gola,palha ├º o,RED,2,pacote,zebra)
(palha ├º o,pacote,BLACK,1,lis,NIL)
(pacote,lis,RED,1,NIL,NIL)
(palha ├º o,zebra,BLACK,1,NIL,NIL)


C:\Users\user\Documents\ProgramaþOo\ßrvore\RBTree\VersOo final Iaslan\main.exe
->teste
----------palavra nao existe----------
->abuso
----------Inserindo a palavra no dicionario --------
->carro
----------Inserindo a palavra no dicionario --------
->doce
----------Inserindo a palavra no dicionario --------
->gola
----------palavra nao existe----------
->gola
----------Inserindo a palavra no dicionario --------
->palha├ºo
----------Inserindo a palavra no dicionario --------
->taturana
----------Inserindo a palavra no dicionario --------
->pacote
----------Inserindo a palavra no dicionario --------
->bolha
----------Inserindo a palavra no dicionario --------
->fussura
----------Inserindo a palavra no dicionario --------
->batata
----------Inserindo a palavra no dicionario --------
->estrela
----------Inserindo a palavra no dicionario --------
->taturana
----------Deletando a palavra do dicionario -------





Conclusão

Com este trabalhou provou-se que a utilização de Árvores Rubro Negras é uma ótima estrutura da dados para a criação de um dicionário.

Apêndice

Arquivos do projeto

RBElement.cpp

main.cpp | RBTree.cpp | RBElement.cpp

```cpp
#ifndef _RBElement_
#define _RBElement_
#include<iostream>
#include <string>

using namespace std;

class RBElement{

    //struct
    public:
        string key;
        string cor;
        RBElement* pai;
        RBElement* esq;
        RBElement* dir;

        //construtor
        RBElement(){
            key ="NIL";
            pai = NULL;
            esq = NULL;
            dir = NULL;
            cor = "BLACK";
        }


};
#endif
```

RBTree.cpp

main.cpp | RBTree.cpp | RBElement.cpp

```
10      public :
11
12          RBElement* root;
13
14          RBTree(){
15              this->root = new RBElement();
16          }
17
18          //inserção ---------------------------------
19          void RBIsert(RBTree* T, RBElement *z){
20              RBElement* y = new RBElement();
21              RBElement* x = T->root;
22
23              while(!(x->key=="NIL")){
24                  y=x;
25                  if((z->key.compare(x->key))<0){
26                      x =x->esq;
27                  }else{
28                      x=x->dir;
29                  }
30              }
31              z->pai =y;
32
33              if(y->key=="NIL"){
34                  T->root = z;
35              }else if(z->key.compare(y->key)<0){
36                  y->esq =z;
37              }else{
38                  y->dir=z;
39              }
40
41              z->esq = new RBElement();
42              z->dir =new RBElement();
43              z->cor = "RED";
44              //chamando RBinserFixup
```

```
49      //Insert Fixup ----------------------------------------------
50      void RBInsertFixup(RBTree* T, RBElement* z){
51          //enquanto o pai for vermelho
52          while(z->pai->cor == "RED"){
53              if(z->pai->key == (z->pai->pai->esq->key)){
54                  RBElement* y= z->pai->pai->dir;
55                  //caso 1
56                  if(y->cor=="RED"){
57                      z->pai->cor = "BLACK";
58                      y->cor = "BLACK";
59                      z->pai->pai->cor="RED";
60                      z = z->pai->pai;
61                  }else{
62                      if((z->key)==(z->pai->dir->key)){
63                          //caso 2
64                          z =z->pai;
65                          LeftRotate(T,z);
66                      }
67                      //caso 3
68                      z->pai->cor = "BLACK";
69                      z->pai->pai->cor = "RED";
70                      RightRotate(T, z->pai->pai);
71                  }
72              }else{
73                  RBElement* y = z->pai->pai->esq;
74                  if(y->cor =="RED"){
75                      z->pai->cor = "BLACK";
76                      y->cor = "BLACK";
77                      z->pai->pai->cor = "RED";
78                      z = z->pai->pai;
79                  }else{
80                      if(z->key == (z->pai->esq->key)){
81                          z =z->pai;
82                          RightRotate(T,z);
83                      }
```

```
71                  }
72              }else{
73                  RBElement* y = z->pai->pai->esq;
74                  if(y->cor =="RED"){
75                      z->pai->cor = "BLACK";
76                      y->cor = "BLACK";
77                      z->pai->pai->cor = "RED";
78                      z = z->pai->pai;
79                  }else{
80                      if(z->key == (z->pai->esq->key)){
81                          z =z->pai;
82                          RightRotate(T,z);
83                      }
84                      z->pai->cor="BLACK";
85                      z->pai->pai->cor="RED";
86                      LeftRotate(T, z->pai->pai);
87                  }
88              }
89          }
90          T->root->cor ="BLACK";
91      }
```

```
92          //Delecao ----------------------------------------------------
93          void RBDelete(RBTree* T, RBElement* z){
94              RBElement* y =z;
95              string OriginalColor = y->cor;
96              if(z->esq->key =="NIL"){
97                  RBElement* x = z->dir;
98                  RBTransplant(T,z,z->dir);
99                  if(OriginalColor=="BLACK"){
100                 }
101             }else if(z->dir->key =="NIL"){
102                 RBElement* x=z->esq;
103                 RBTransplant(T,z,z->esq);
104                 if(OriginalColor =="BLACK"){
105                     RBDeleteFixup(T,x);
106                 }
107             }else{
108                 y = TreeMinimun(z->dir);
109                 OriginalColor = y->cor;
110                 RBElement* x=y->dir;
111                 if(y->pai->key == z->key){
112                     x->pai=y;
113                 }else{
114                     RBTransplant(T,y,y->dir);
115                     y->dir = x->dir;
116                     y->dir->pai =y;
117                 }
118                 RBTransplant(T,z,y);
119                 y->esq = z->esq;
120                 y->esq->pai = y;
121                 y->cor =z->cor;
122                 if(OriginalColor =="BLACK"){
123                     RBDeleteFixup(T,x);
124                 }
125             }
```

```
127         //delete fixup-------------------------------------------------
128         void RBDeleteFixup(RBTree* T, RBElement* x){
129             while((!(x->key==(T->root->key)))&&x->cor=="BLACK"){
130                 if(x->key ==(x->pai->esq->key)){
131                     RBElement* w =x->pai->dir;
132                     //caso 1
133                     if(w->cor=="RED"){
134                         w->cor=="BLACK";
135                         x->pai->cor="RED";
136                         LeftRotate(T,x->pai);
137                         w = x->pai->dir;
138                     }
139                     //caso2
140                     if((w->esq->cor=="BLACK")&& w->dir->cor=="BLACK"){
141                         w->cor ="RED";
142                         x=x->pai;
143                     //caso 3
144                     }else{//caso 3
145                         if(w->dir->cor =="BLACK"){
146                             w->esq->cor = "BLACK";
147                             w->cor="RED";
148                             RightRotate(T,w);
149                             w = x->pai->dir;
150                         }
151                         //caso 4
152                         w->cor = x->pai->cor;
153                         x->pai->cor = "BLACK";
154                         w->dir->cor="BLACK";
155                         LeftRotate(T,x->pai);
156                         x = T->root;
157                     }
158                 }else{
159                     RBElement* w =x->pai->esq;
160                     if(w->cor=="RED"){
161                         w->cor ="BLACK";
```

```
157                 }
158             }else{
159                 RBElement* w =x->pai->esq;
160                 if(w->cor=="RED"){
161                     w->cor ="BLACK";
162                     x->pai->cor ="RED";
163                     RightRotate(T,x->pai);
164                     w=x->pai->esq;
165                 }
166                 if((w->esq->cor=="BLACK")&&(w->dir->cor=="BLACK")){
167                     w->cor = "RED";
168                     x =x->pai;
169                 }else{
170                     if(w->esq->cor=="BLACK"){
171                         w->dir->cor="BLACK";
172                         w->cor="RED";
173                         LeftRotate(T,w);
174                         w=x->pai->esq;
175                     }
176                     w->cor = x->pai->cor;
177                     x->pai->cor ="BLACK";
178                     w->esq->cor = "BLACK";
179                     RightRotate(T,x->pai);
180                     x=T->root;
181 
182                 }
183             }
184         }
185         x->cor ="BLACK";
186     }
187     //rotação a esquerda
188     void LeftRotate(RBTree* T, RBElement* x){
189         RBElement* y= x->dir;
190         x->dir = y->esq;
191 
192         if(!(y->esq->key == "NIL")){
193             y->esq->pai =x;
194         }
195         y->pai = x->pai;
196         if(x->pai->key =="NIL"){
197             T->root = y;
198         }else if(x->key == (x->pai->esq->key)){
199             x->pai->esq = y;
200         }else{
201             x->pai->dir =y;
202         }
203 
204         y->esq =x;
205         x->pai =y;
206     }
207 
```

```
208     //rotação a direita
209     void RightRotate(RBTree* T, RBElement* x){
210         RBElement* y =x->esq;
211         x->esq = y->dir;
212         if(!(y->dir->key=="NIL")){
213             y->dir->pai =x;
214         }
215         y->pai = x->pai;
216 
217         if(x->pai->key =="NIL"){
218             T->root = y;
219         }else if(x->key == (x->pai->dir->key)){
220             x->pai->dir = y;
221         }else{
222             x->pai->esq = y;
223         }
224 
225         y->dir =x;
226         x->pai=y;
227     }
228 
229     //transplant
230     void RBTransplant(RBTree* T, RBElement* u, RBElement* v){
231         if(u->pai->key=="NIL"){
232             T->root = v;
233         }else if(u->key == (u->pai->esq->key)){
234             u->pai->esq =v;
235         }else{
236             u->pai->dir = v;
237         }
238         v->pai = u->pai;
239     }
240 
241     //busca o menor valor da arvore
242     RBElement* TreeMinimun(RBElement* x){
243         while(!(x->esq->key == "NIL")){
244             x=x->esq;
245         }
246         return x;
247     }
248 
249     //busca o maior
250     RBElement* TreeMaximun(RBElement* x){
251         while(!(x->dir->dir->key =="NIL")){
252             x=x->dir;
253         }
254         return x;
255     }
256     //busca
257     RBElement* TreeSearch(RBElement* x, string info){
258 
259         while((x->key!="NIL")&&(info!=x->key)){
260             if(info.compare(x->key)<0){
261                 x = x->esq;
262             }else{
263                 x = x->dir;
264             }
265         }
266         return x;
267 
268     }
```

```
    //profundida negra da arvore

    int RBAlturaNegra(RBElement* x){
        if(x->key =="NIL"){
            return 1;
        }else if(x->esq->cor =="BLACK"){
            return RBAlturaNegra(x->esq)+1;
        }else{
            return RBAlturaNegra(x->esq);
        }
    }

    //RBCheck
    void RBCheck(RBElement* x){
        if(!(x->key == "NIL")){
            int altura = RBAlturaNegra(x)-1;
            cout << "(" <<x->pai->key << ","<<    x->key  << "," << x->cor << "," << altura << "," << x->esq->key << "," << x->dir->key + ")\n";
            RBCheck(x->esq);
            RBCheck(x->dir);
        }
    }

    //RBPrint
    void RBPrint(RBTree* T){

        RBElement* x = T->root;
        InOrderTree(x);

    }
```

```
290
291         //RBPrint
292         void RBPrint(RBTree* T){
293
294             RBElement* x = T->root;
295             InOrderTree(x);
296
297         }
298
299 //Exibição em ordenação pela raiz
300     void InOrderTree(RBElement* x){
301         if(!(x->key =="NIL")) {
302             InOrderTree(x->esq);
303             cout << x->key + " ";
304             InOrderTree(x->dir);
305         }
306
307     }
308
309 };
310 #endif
```

Main.cpp

main.cpp | RBTree.cpp | RBElement.cpp

```
1   //autor Iaslan Nascimento
2   //árvore rubro negra dicionario
3   #include "RBTree.cpp"
4   #include "RBElement.cpp"
5   #include<iostream>
6   #include<fstream>
7   #include <string>
8
9   using namespace std;
10
11  int main(int argc, char*argv[]){
12      //leitura de arquivo
13      ifstream arq;
14      arq.open("dicionario1.txt");
15      string info;
16      int op;
17
18      RBTree* T = new RBTree();
19      RBElement* no;
20
```

```
    while(arq>>info){
        arq >> op;
        //verificando se a opção é 1 ou 0  para decidir se vamos
        //inserir ou deletar
        //verificar se a opção é 1 e a palavra não existe
        if((op ==1)&&(T->TreeSearch(T->root,info)->key =="NIL")){
            cout <<"->"+info+"\n";
            cout << "-----------Inserindo a palavra no dicionario --------\n\n";

            no = new RBElement();
            no->key = info;
            T->RBIsert(T,no);
        }else if((op == 1)&&(T->TreeSearch(T->root,info)->key!="NIL")){
            cout <<"->"+info+"\n";
            cout<<"-----------Palavra ja existente no dicionario---------\n\n";

        }else if((op == 0)&&(T->TreeSearch(T->root,info)->key!="NIL")){
            RBElement* no = T->TreeSearch(T->root,info);
            cout <<"->"+info+"\n";
            cout <<"\n-----------Deletando a palavra do dicionario -------\n\n\n";
            T->RBDelete(T,no);
            T->RBPrint(T);
            T->RBCheck(no);
        }else{
                cout <<"->"+info+"\n";
                cout<<"-----------palavra nao existe----------- \n\n";

        }
    }


    T->RBPrint(T);
50
51
52      T->RBPrint(T);
53
54      cout <<"\n \n \n \n";
55      T->RBCheck(T->root);
56      return 0;
57
58  }
59
```