UNIVERSIDADE FEDERAL DO PIAUÍ – UFPI
CAMPUS SENADOR HELVÍDIO NUNES DE BARROS – PICOS
Disciplina: Estrutura de dados II
Curso: Bacharelado em Sistemas de Informação

Acadêmico: Weliton de Sousa Araujo
Dupla: Darice da Rocha Sousa

1. RESUMO

Este projeto foi totalmente desenvolvido com a linguagem de programação C, cujo a mesma se prova de extrema performance pelo fato de ser uma linguagem compilada. Este relatório tem como principal fundamento a resolução das questões da primeira atividade avaliativa. Tal qual o foco se mostra em cima de árvores binárias de busca e árvores AVL, cujo o melhor recurso utilizado para percorrimento das árvores foram funções recursivas, pois as mesmas se mostraram mais eficazes e de fácil implementação, porém as funções recursivas se mostram muito perigosas tendo em vista que a condição de parada tem que ser atendida de alguma forma, caso contrário, o código além de não ser performático, não entregará nenhum resultado por ter entrado num loop infinito.

2. INTRODUÇÃO

Um dos grandes problemas da atualidade perante o mundo da programação é realizar uma busca de algum dado de forma performática, pois com a escalabilidade dos dados, nossos algoritmos atuais se tornam cada vez mais ineficientes, tendo isso em mente, neste trabalho foram desenvolvidas dois tipos de árvores, binárias de busca e AVL, cujo as mesmas propõem uma abordagem diferente das listas lineares.

Com a árvore binária de busca podemos ordenar os dados de uma forma lógica, e não sequencial, tal qual quando precisamos de algum dado específico, essa estrutura da árvore pode ser percorrida com uma quantidade de passos muito inferior a uma lista simples com dados sequenciais.

Na árvore AVL temos uma abordagem semelhante, porém, seus galhos são totalmente balanceados, isto é, uma árvore simétrica, onde a cada novo dado inserido, esta árvore tem que ser balanceada novamente. Com isso, nesta mesma árvore temos um tempo consideravelmente maior na inserção de algum novo dado, devido a reordenação da mesma.

3. HARDWARE UTILIZADO

Todos os testes a seguir foram feitos utilizando o mesmo hardware, com as seguintes características:

Marca	Apple
Sistema Operacional	Big Sur
Processador	M1 Apple bionic
Memória Ram	8 GBs
Memória Rom	256 GBs
Tipo de memória Ram	LPDDR4X 4266 MHz

4. SEÇÕES ESPECÍFICAS

4.1 QUESTÕES 01 E 02

As questões 01 e 02 desta atividade são semelhantes, exceto que uma pede uma árvore binária de busca e a outra uma árvore AVL, tendo isto em vista, a lógica de solução de ambas as questões foram as mesmas.

O primeiro passo realizado, foi a criação de um vetor com 1.000 elementos sequenciais, isto é de 0 até 999, e isto foi feito num loop for, para que não haja números repetidos.

Após obter 1.000 números não repetidos, todos os elementos foram inseridos, sendo assim, a arvore da primeira questão fica numa configuração semelhante a de uma lista encadeada, enquanto a árvore da segunda questão fica totalmente equilibrada devido ao algoritmo de balanceamento utilizado após a inserção de cada elemento nesta mesma árvore.

Após a inserção dos números sequenciais, foram realizados os cálculos de maior e menor profundidade em ambas as árvores, e logo em seguida foi invocado um método de busca, para que o mesmo possa buscar todos os elementos contidos nessas árvores.

Logo em seguida, a memória foi limpada, para que os endereços de memória possam ser utilizados novamente por novos valores, e foi realizados os mesmos passos anteriores: inserção, cálculos de profundidade e busca,

com a diferença que os 1.000 números inseridos são aleatórios e não repetidos

4.2 QUESTÕES 03 E 04

As questões 03 e 04 também são muito semelhantes, com a diferença de que uma pede uma árvore binária de busca e a outra pede uma árvore AVL, tendo isto em vista, a lógica de solução de ambas as questões foram as mesmas, assim como nas duas primeiras questões.

O arquivo utilizado para leitura nos testes, foi o seguinte:

```
%pronomes
I:eu
you:voce
he:ele
she:ela
it:ele,ela
we:nos
they:eles,elas
%objetos
bell:sino,campainha
glasses:oculos,copos
picture:imagem,fotografia,pintura
photograph:fotografia
clock:relogio
cotton:algodao
calico:algodao,chita
eyeglasses:oculos
%cores
black:preto
blue:azul
pink:roso
```

Em seguida foi feita uma leitura do arquivo, linha a linha para poder separar as unidades de suas respectivas traduções.

Para realizar este feito, foi criado uma lista encadeada para armazenar as unidades, e uma árvore binária, que em cada galho desta mesma árvore continha um array de caracteres responsáveis por guardar a palavra em português, e uma lista encadeada para guardar as N traduções desta mesma palavra para a língua inglesa.

E em seguida, é apresentado um menu para o usuário, onde o mesmo poderá interagir com o algoritmo podendo realizar as seguintes operações:

1 - Mostrar unidades
2 - Mostrar dicionário
3 - Pesquisar unidade específica
4 - Pesquisar palavra portugues
5 - Deletar palavra
0 - Sair

5. RESULTADO DA EXECUÇÃO DOS PROGRAMAS

5.1 QUESTÃO 01

Resultado obtido com dados sequenciais:

Tempo de inserção dos dados sequenciais	5,193 clocks por milisegundos
Tempo de busca dos dados sequenciais	4,882 clocks por milisegundos

Resultado obtido com dados aleatórios:

Média de tempo de inserção	0,1237 clocks por milisegundos
Média de tempo de busca	0,0788 clocks por milisegundos

Profundidade	Repetições
18	4 vezes
16	5 vezes
19	1 vezes
17	6 vezes
15	5 vezes
14	6 vezes
22	1 vezes
30	1 vezes

30	1 vezes

5.2 QUESTÃO 02

Resultado obtido com dados sequenciais:

Tempo de inserção com dados sequenciais	0,789 clocks por milisegundos
Tempo de busca com dados sequenciais	0,136 clocks por milisegundos
Maior profundidade com dados sequenciais	9
Menor profundidade com dados sequenciais	8

Resultado obtido com dados aleatórios:

Média de tempo de inserção	0,5655 clocks por milisegundos
Média de tempo de busca	0,132067 clocks por milisegundos

Profundidade	Repetições
4	26 vezes
3	3 vezes

5.3 QUESTÕES 03 E 04

ID	Unidade	Português	Inglês
1	Pronomes	eu	I
		voce	you
		ele	he, it
		ela	she, it
		nos	we
		eles	they
		elas	they
2	Objetos	sino	bell
		campainha	bell
		óculos	glasses, eyeglasses
		copos	glasses
		imagem	picture
		fotografia	picture, photograph
		pintura	picture
		relógio	clock
		algodão	cotton, calico
		chita	calico
3	cores	preto	black
		azul	blue
		roso	pink

6. CONCLUSÃO

Com os resultados obtidos nas questões 01 e 02 desta atividade, é possível afirmar com certeza que uma árvore binária, sendo ela AVL ou de busca, tem uma performance muito superior a uma lista encadeada, exceto quando a árvore é binária, e está armazenando dados sequenciais, ou seja, quando a árvore se encontra na configuração de uma lista normal.

Esta afirmação é baseada no primeiro teste da primeira questão, onde numa lista binária que se encontra numa configuração semelhante a uma lista encadeada. O tempo de inserção desta mesma lista, foi de até 50 vezes maior se comparada com os testes seguintes onde a base de dados se encontrava ordenada de forma aleatória.

Em relação às árvores AVL, seu tempo de inserção é consideravelmente maior, devido a mesma ter uma quantidade de passos muito maior, já em relação ao tempo de busca, seu tempo de busca pode sofrer algumas variações em relação a uma árvore binária de busca.

Para testes de grandes quantidades, a árvore binária de busca provou ter uma eficiência maior, em termos de buscar algum elemento, já quando falamos de pequenas quantidades, a AVL tem uma performance maior, devido ao seu equilíbrio de distância dos termos até a raiz da árvore.

7. APÊNDICE

7.1 QUESTÃO 01

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TAM 1000
#define TESTES 30

struct arvore
{
   struct arvore *esquerda;
   int info;
   struct arvore *direita;
};
typedef struct arvore Arvore;
```

```c
/*preenche vetor de tamanho TAM com números sequenciais de 0 a TAM-1*/
void preencherVetor(int vetor[TAM])
{
   for (int i = 0; i < TAM; i++)
   {
       vetor[i] = i;
   }
}

/*embaralha os elementos entre as posições do vetor*/
void embaralharVetor(int vetor[TAM])
{
   for (int i = 0; i < TAM; i++)
   {
       int aux = vetor[i];
       int aleatorio = rand() % TAM;
       vetor[i] = vetor[aleatorio];
       vetor[aleatorio] = aux;
   }
}

/*Insire nó na arvore*/
void inserirNo(Arvore **raiz, Arvore *no)
{
   if (*raiz == NULL)
   {
       *raiz = no;
   }
   else
   {
       if (no->info < (*raiz)->info)
       {
           inserirNo(&((*raiz)->esquerda), no);
       }
       else if (no->info > (*raiz)->info)
       {
           inserirNo(&((*raiz)->direita), no);
       }
   }
```

```c
}

/*cria no com info n e retorna no*/
Arvore *criarNo(int n)
{
   Arvore *no = (Arvore *)calloc(1, sizeof(Arvore));
   no->info = n;
   no->esquerda = NULL;
   no->direita = NULL;
   return no;
}

/*libera a arvore da memória*/
void liberarArvore(Arvore *raiz)
{
   if (raiz != NULL)
   {
       liberarArvore(raiz->esquerda);
       liberarArvore(raiz->direita);
       free(raiz);
   }
}

/*insire os elementos do vetor na arvore*/
void inserirElementosArvore(Arvore **raiz, int vetor[TAM])
{
   for (int i = 0; i < TAM; i++)
   {
       Arvore *no = criarNo(vetor[i]);
       inserirNo(raiz, no);
   }
}

/*verica se no é folha, se sim, retorna 1, se não, retorna 0*/
int ehFolha(Arvore *no)
{
   int folha = 0;
   if (no != NULL)
   {
```

```
        if (no->direita == NULL && no->esquerda == NULL)
            folha = 1;
    }
    return folha;
}

/*calcula a profundidade de nó na arvore raiz e retorna*/
int profundidade(Arvore *Raiz, Arvore *No)
{
    int profNo = -1;
    if (Raiz != NULL)
    {
        if ((*No).info < (*Raiz).info)
            profNo = profundidade((*Raiz).esquerda, No) + 1;
        else if ((*No).info > (*Raiz).info)
            profNo = profundidade((*Raiz).direita, No) + 1;
        else
            profNo = 0;
    }
    return profNo;
}

/*calcula o nível da folha de menor profundidade e retorna
Raiz e No devem receber inicialmente a raiz da arvore*/
int profundidadeMenor(Arvore *Raiz, Arvore *No)
{
    int profRaiz, menorEsquerda = 0, menorDireita = 0, menorProf = TAM;
    if (No != NULL)
    {
        if (ehFolha(No))
        {
            profRaiz = profundidade(Raiz, No);
            if (profRaiz < menorProf)
                menorProf = profRaiz;
        }
        else
        {
            menorEsquerda = profundidadeMenor(Raiz, No->esquerda);
            menorDireita = profundidadeMenor(Raiz, No->direita);
```

```
            if (menorEsquerda < menorDireita)
                menorProf = menorEsquerda;
            else
                menorProf = menorDireita;
        }
    }
    return menorProf;
}

/*calcula o nível da folha de maior profundidade e retorna
Raiz e No devem receber inicialmente a raiz da arvore*/
int profundidadeMaior(Arvore *Raiz, Arvore *No)
{
    int profRaiz, maiorEsquerda = 0, maiorDireita = 0, maiorProf = -1;

    if (No != NULL)
    {
        if (ehFolha(No))
        {
            profRaiz = profundidade(Raiz, No);
            if (profRaiz > maiorProf)
                maiorProf = profRaiz;
        }
        else
        {
            maiorEsquerda = profundidadeMaior(Raiz, No->esquerda);
            maiorDireita = profundidadeMaior(Raiz, No->direita);
            if (maiorEsquerda > maiorDireita)
                maiorProf = maiorEsquerda;
            else
                maiorProf = maiorDireita;
        }
    }
    return maiorProf;
}

/*recebe o clock de inicio e o de fim e retorna
a quantidade de clocks por milisegundos*/
double calcularTempo(clock_t inicio, clock_t fim)
```

```
{
   double tempo = ((double)(fim - inicio) * 1000 / CLOCKS_PER_SEC);
   return tempo;
}

/*busca na arvore o no com info='elemento', se encontra retorna o no,
se não, retorna NULL*/
Arvore *buscar(Arvore *raiz, int elemento)
{
   Arvore *no;
   if (raiz != NULL)
   {
       if (elemento == raiz->info)
       {
           no = raiz;
       }
       else
       {
           if (elemento < raiz->info)
               no = buscar(raiz->esquerda, elemento);
           else
               no = buscar(raiz->direita, elemento);
       }
   }
   else
   {
       no = NULL;
   }
   return no;
}

/*Preenche o vetor com a diferença entre a profundidade máxima e mínima
A primeira dimensão armazena a diferença das profundidadas
A segunda dimensão armazena a quantidade de vezes que tal diferença aparece
diferença é a nova dimensão que será adicionado no vetor
quanTtermos é um ponteiro com a quantidade de elementos que o vetor possui
*/
void inserirDiferencaProfundidade(int vetor[2][TESTES], int diferenca, int
*quantTermos)
```

```
{
    for (int i = 0; i < *quantTermos && diferenca != -1; i++)
    {
        if (vetor[0][i] == diferenca)
        {
            vetor[1][i]++;
            diferenca = -1;
        }
    }
    if (diferenca != -1)
    {
        vetor[0][*quantTermos] = diferenca;
        vetor[1][*quantTermos] = 1;
        (*quantTermos)++;
    }
}

void main()
{

    int vetor[TAM];
    preencherVetor(vetor);
    Arvore *raiz;
    int diferencasProfundidade[2][TESTES];
    int quantTermos = 0;
    clock_t inicio, fim;
    double tempo, somaInsercao = 0, somaBusca = 0;
    srand(time(NULL));
    for (int i = 0; i <= TESTES; i++)
    {
        raiz = NULL;
        printf("\nTeste %d:\n", i);
        if (i != 0)
        {
            embaralharVetor(vetor);
            embaralharVetor(vetor);
        }
        inicio = clock();
        inserirElementosArvore(&raiz, vetor);
```

```
        fim = clock();
        tempo = calcularTempo(inicio, fim);
        printf("Inserção tempo: %lf\n", tempo);
        if (i != 0)
            somaInsercao += tempo;
        int maiorProfundidade = profundidadeMaior(raiz, raiz);
        printf("Maior profundidade: %d\n", maiorProfundidade);
        int menorProfundidade = profundidadeMenor(raiz, raiz);
        printf("Menor profundidade: %d\n", menorProfundidade);
        int diferenca = maiorProfundidade - menorProfundidade;
        if (i != 0)
                    inserirDiferencaProfundidade(diferencasProfundidade, diferenca,
&quantTermos);
        inicio = clock();
        for (int j = 0; j < TAM; j ++)
            buscar(raiz, j);
        fim = clock();
        tempo = calcularTempo(inicio, fim);
        printf("Busca tempo: %lf\n", tempo);
        if (i != 0)
            somaBusca += tempo;
        liberarArvore(raiz);
    }

    printf("\nDiferencas de profundidade:\n");
    for (int i = 0; i < quantTermos; i++)
                    printf("%d   (%d   vez(es))\n",   diferencasProfundidade[0][i],
diferencasProfundidade[1][i]);

    printf("media do tempo de insercao: %lf\n", somaInsercao / TESTES);
    printf("media do tempo de busca: %lf\n", somaBusca / TESTES);
}
```

7.2 QUESTÃO 02

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TAM 1000
```

```c
#define TESTES 30
struct arvore
{
   struct arvore *esquerda;
   int info;
   struct arvore *direita;
   int altura;
};
typedef struct arvore Arvore;

/*preenche vetor de tamanho TAM com números sequenciais de 0 a TAM-1*/
void preencherVetor(int vetor[TAM])
{
   for (int i = 0; i < TAM; i++)
   {
       vetor[i] = i;
   }
}

/*embaralha os elementos entre as posições do vetor*/
void embaralharVetor(int vetor[TAM])
{
   for (int i = 0; i < TAM; i++)
   {
       int aux = vetor[i];
       int aleatorio = rand() % TAM;
       vetor[i] = vetor[aleatorio];
       vetor[aleatorio] = aux;
   }
}

/*verica se no é folha, se sim, retorna 1, se não, retorna 0*/
int ehFolha(Arvore *no)
{
   int folha = 0;
   if (no != NULL)
   {
       if (no->direita == NULL && no->esquerda == NULL)
           folha = 1;
```

```
    }
    return folha;
}

/*calcula a profundidade de nó na arvore raiz e retorna*/
int profundidade(Arvore *Raiz, Arvore *No)
{
    int profNo = -1;
    if (Raiz != NULL)
    {
        if ((*No).info < (*Raiz).info)
            profNo = profundidade((*Raiz).esquerda, No) + 1;
        else if ((*No).info > (*Raiz).info)
            profNo = profundidade((*Raiz).direita, No) + 1;
        else
            profNo = 0;
    }
    return profNo;
}

/*calcula o nível da folha de maior profundidade e retorna
Raiz e No devem receber inicialmente a raiz da arvore*/
int profundidadeMaior(Arvore *Raiz, Arvore *No)
{
    int profRaiz, maiorEsquerda = 0, maiorDireita = 0, maiorProf = -1;

    if (No != NULL)
    {
        if (ehFolha(No))
        {
            profRaiz = profundidade(Raiz, No);
            if (profRaiz > maiorProf)
                maiorProf = profRaiz;
        }
        else
        {
            maiorEsquerda = profundidadeMaior(Raiz, No->esquerda);
            maiorDireita = profundidadeMaior(Raiz, No->direita);
            if (maiorEsquerda > maiorDireita)
```

```
            maiorProf = maiorEsquerda;
        else
            maiorProf = maiorDireita;
    }
  }
  return maiorProf;
}

/*calcula o nível da folha de menor profundidade e retorna
Raiz e No devem receber inicialmente a raiz da arvore*/
int profundidadeMenor(Arvore *Raiz, Arvore *No)
{
  int profRaiz, menoresquerda = 0, menordireita = 0, menorProf = 10000;

  if (No != NULL)
  {
    if (ehFolha(No))
    {
        profRaiz = profundidade(Raiz, No);
        if (profRaiz < menorProf)
            menorProf = profRaiz;
    }
    else
    {
        menoresquerda = profundidadeMenor(Raiz, No->esquerda);
        menordireita = profundidadeMenor(Raiz, No->direita);
        if (menoresquerda < menordireita)
            menorProf = menoresquerda;
        else
            menorProf = menordireita;
    }
  }
  return menorProf;
}

/*recebe o clock de inicio e o de fim e retorna
a quantidade de clocks por milisegundos*/
double calcularTempo(clock_t inicio, clock_t fim)
{
```

```
   double tempo = ((double)(fim - inicio) * 1000 / CLOCKS_PER_SEC);
   return tempo;
}

/*busca na arvore o no com info='elemento', se encontra retorna o no,
se não, retorna NULL*/
Arvore *buscar(Arvore *raiz, int elemento)
{
   Arvore *no;
   if (raiz != NULL)
   {
       if (elemento == raiz->info)
       {
           no = raiz;
       }
       else
       {
           if (elemento < raiz->info)
               no = buscar(raiz->esquerda, elemento);
           else
               no = buscar(raiz->direita, elemento);
       }
   }
   else
   {
       no = NULL;
   }
   return no;
}

/*Preenche o vetor com a diferença entre a profundidade máxima e mínima
A primeira dimensão armazena a diferença das profundidadas
A segunda dimensão armazena a quantidade de vezes que tal diferença aparece
diferença é a nova dimensão que será adicionado no vetor
quanTtermos é um ponteiro com a quantidade de elementos que o vetor possui
*/
void inserirDiferencaProfundidade(int vetor[2][TESTES], int diferenca, int
*quantTermos)
{
```

```
    for (int i = 0; i < *quantTermos && diferenca != -1; i++)
    {
        if (vetor[0][i] == diferenca)
        {
            vetor[1][i]++;
            diferenca = -1;
        }
    }
    if (diferenca != -1)
    {
        vetor[0][*quantTermos] = diferenca;
        vetor[1][*quantTermos] = 1;
        (*quantTermos)++;
    }
}

Arvore *girarHorario(Arvore *raiz)
{
    Arvore *no = raiz->esquerda;
    raiz->esquerda = no->direita;
    no->direita = raiz;
    return no;
}

Arvore *girarAntiHorario(Arvore *raiz)
{
    Arvore *no = raiz->direita;
    raiz->direita = no->esquerda;
    no->esquerda = raiz;
    return no;
}

/*calcula o fator de balanceamento do nó raiz e retorna*/
int fatorBalanceamento(Arvore *raiz)
{
    int fator = 0, esq, dir;
    if (raiz->esquerda == NULL)
        esq = -1;
    else
```

```
        esq = raiz->esquerda->altura;
    if (raiz->direita == NULL)
        dir = -1;
    else
        dir = raiz->direita->altura;
    fator = esq - dir;
    return fator;
}

/*calcula a altura do nó raiz e retorna*/
int alturaElemento(Arvore *raiz)
{
    int altura = 1;
    int esquerda = -1;
    int direita = -1;

    if (raiz->esquerda != NULL)
        esquerda = raiz->esquerda->altura;
    if (raiz->direita != NULL)
        direita = raiz->direita->altura;
    altura = 1;
    if (esquerda > direita)
        altura += esquerda;
    else
        altura += direita;

    return altura;
}

/*calcula a altura da raiz e de seus descendentes e armazena em raiz->altura*/
int calcularAlturaCadaElementoGalho(Arvore *raiz)
{
    int altura = 0;
    if (raiz == NULL)
    {
        altura = -1;
    }
    else
    {
```

```
        altura = 1;
        int direita = calcularAlturaCadaElementoGalho(raiz->direita);
        int esquerda = calcularAlturaCadaElementoGalho(raiz->esquerda);
        if (esquerda > direita)
            altura += esquerda;
        else
            altura += direita;
        raiz->altura = altura;
    }
    return altura;
}

/*balanceia raiz*/
Arvore *balancear(Arvore *raiz)
{
    if (fatorBalanceamento(raiz) > 0) //positivo
    {
        if (fatorBalanceamento(raiz->esquerda) < 0) //negativo
        {
            raiz->esquerda = girarAntiHorario(raiz->esquerda);
        }
        raiz = girarHorario(raiz);
    }
    else if (fatorBalanceamento(raiz) < 0) //negativo
    {

        if (fatorBalanceamento(raiz->direita) > 0) //positivo
        {
            raiz->direita = girarHorario(raiz->direita);
        }
        raiz = girarAntiHorario(raiz);
    }

    return raiz;
}

/*Insire nó na arvore e balanceia*/
void inserirNo(Arvore **raiz, Arvore *no)
{
```

```
    if (*raiz == NULL)
    {
        *raiz = no;
    }
    else
    {
        if (no->info < (*raiz)->info)
        {
            inserirNo(&((*raiz)->esquerda), no);
        }
        else if (no->info > (*raiz)->info)
        {
            inserirNo(&((*raiz)->direita), no);
        }
        (*raiz)->altura = alturaElemento(*raiz);
        if (abs(fatorBalanceamento(*raiz)) > 1)
        {
            *raiz = balancear(*raiz);
            calcularAlturaCadaElementoGalho(*raiz);
        }
    }
}

/*cria no com info n e retorna no*/
Arvore *criarNo(int n)
{
    Arvore *no = (Arvore *)calloc(1, sizeof(Arvore));
    no->info = n;
    no->esquerda = NULL;
    no->direita = NULL;
    no->altura = 0;
    return no;
}

/*libera a arvore da memória*/
void liberarArvore(Arvore *raiz)
{
    if (raiz != NULL)
    {
```

```
        liberarArvore(raiz->esquerda);
        liberarArvore(raiz->direita);
        free(raiz);
    }
}

/*insire os elementos do vetor na arvore*/
void inserirElementosArvore(Arvore **raiz, int vetor[TAM])
{
    for (int i = 0; i < TAM; i++)
    {
        Arvore *no = criarNo(vetor[i]);
        inserirNo(raiz, no);
    }
}

void main()
{

    int vetor[TAM];
    preencherVetor(vetor);
    Arvore *raiz = NULL;
    int diferencasProfundidade[2][TESTES];
    int quantTermos = 0;
    clock_t inicio, fim;
    double tempo, somaInsercao = 0, somaBusca = 0;
    srand(time(NULL));
    for (int i = 0; i <= TESTES; i++)
    {
        raiz = NULL;
        printf("\nTeste %d:\n", i);
        if (i != 0)
        {
            embaralharVetor(vetor);
            embaralharVetor(vetor);
        }
        inicio = clock();
        inserirElementosArvore(&raiz, vetor);
        fim = clock();
```

```
        tempo = calcularTempo(inicio, fim);
        printf("Inserção tempo: %lf\n", tempo);
        if (i != 0)
            somaInsercao += tempo;
        int maiorProfundidade = profundidadeMaior(raiz, raiz);
        printf("Maior profundidade: %d\n", maiorProfundidade);
        int menorProfundidade = profundidadeMenor(raiz, raiz);
        printf("Menor profundidade: %d\n", menorProfundidade);
        int diferenca = maiorProfundidade - menorProfundidade;
        if (i != 0)
                    inserirDiferencaProfundidade(diferencasProfundidade, diferenca,
&quantTermos);
        inicio = clock();
        for (int j = 0; j < TAM; j++)
            buscar(raiz, j);
        fim = clock();
        tempo = calcularTempo(inicio, fim);
        printf("Busca tempo: %lf\n", tempo);
        if (i != 0)
            somaBusca += tempo;
        liberarArvore(raiz);
    }

    printf("\nDiferencas de profundidade:\n");
    for (int i = 0; i < quantTermos; i++)
    {
                    printf("%d   (%d   vez(es))\n",   diferencasProfundidade[0][i],
diferencasProfundidade[1][i]);
    }

    printf("media do tempo de insercao: %lf\n", somaInsercao / TESTES);
    printf("media do tempo de busca: %lf\n", somaBusca / TESTES);
}
```

7.3 QUESTÃO 03

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <ctype.h>

#define MAX 100

struct lista
{
   char palavra[MAX];
   struct lista *prox;
};
typedef struct lista Lista;

struct arvore
{
   struct arvore *esquerda;
   char portugues[MAX];
   struct arvore *direita;
   struct lista *ingles;
};
typedef struct arvore Arvore;

struct unidade
{
   int id;
   char nome[MAX];
   struct arvore *raiz;
   struct unidade *prox;
};
typedef struct unidade Unidade;

/*cria unidade e retorna.
Recebe a string nome que será armazenada em unidade->nome
e o id que será armazenado em unidade->id */
Unidade *criarUnidade(char nome[MAX], int id)
{
   Unidade *unidade = (Unidade *)calloc(1, sizeof(Unidade));
   strcpy(unidade->nome, nome);
   unidade->id = id;
   unidade->prox = NULL;
```

```
    unidade->raiz = NULL;
    return unidade;
}

/*Inseri o nó unidade na lista dicionario*/
void inserirUnidadeNoDicionario(Unidade **dicionario, Unidade *unidade)
{
    if (*dicionario == NULL)
    {
        *dicionario = unidade;
    }
    else
    {
        Unidade *aux = *dicionario;
        while (aux->prox != NULL)
        {
            aux = aux->prox;
        }
        aux->prox = unidade;
    }
}

/*cria uma arvore, tal qual arvore->portugues recebe a string 'portugues'.
Retorna a arvore*/
Arvore *criarArvore(char portugues[MAX])
{
    Arvore *arvore = (Arvore *)calloc(1, sizeof(Arvore));
    arvore->direita = NULL;
    arvore->esquerda = NULL;
    arvore->ingles = NULL;
    strcpy(arvore->portugues, portugues);
    return arvore;
}

/*compara a palavra1 e a palavra 2 para averiguar qual vem primeiro em ordem
alfabetica.
retorna 1 se a palavra que vem primeiro for palavra1
retorna 2 se a palavra que vem primeiro for palavra2
retorna 0 se as duas palavras possuem os mesmos caracteres
```

```c
A variavel i indica a posição do caracter a ser comparado. Deve receber 0 na
chamada da função*/
int compararPalavras(char palavra1[MAX], char palavra2[MAX], int i)
{
   int palavraQueVemPrimeiro = 0;
   if (strlen(palavra1) > i && strlen(palavra2) > i)
   {
       if (tolower(palavra1[i]) == tolower(palavra2[i]))
       {
           palavraQueVemPrimeiro = compararPalavras(palavra1, palavra2, i + 1);
       }
       else
       {
           if (tolower(palavra1[i]) < tolower(palavra2[i]))
               palavraQueVemPrimeiro = 1;
           if (tolower(palavra1[i]) > tolower(palavra2[i]))
               palavraQueVemPrimeiro = 2;
       }
   }
   return palavraQueVemPrimeiro;
}

/*Insere nó em raiz*/
void inserirArvoreNaRaizDaUnidade(Arvore **raiz, Arvore *no)
{
   if (*raiz == NULL)
   {
       *raiz = no;
   }
   else
   {
       int primeiro = compararPalavras(no->portugues, (*raiz)->portugues, 0);
       if (primeiro == 1)
           inserirArvoreNaRaizDaUnidade(&((*raiz)->esquerda), no);
       else
           inserirArvoreNaRaizDaUnidade(&((*raiz)->direita), no);
   }
}
```

```c
/*cria um nó de lista que conterá a palavra em ingles
tal qual no->palavra recebe a string 'palavra'.
Retorna o no*/
Lista *criarPalavraIngles(char palavra[MAX])
{
   Lista *no = (Lista *)calloc(1, sizeof(Lista));
   strcpy(no->palavra, palavra);
   no->prox = NULL;
   return no;
}

/*insere o nó ingles  na lista*/
void inserirPalavraIngles(Lista **lista, Lista *ingles)
{
   if (*lista == NULL)
   {
       *lista = ingles;
   }
   else
   {
       Lista *aux = *lista;
       while (aux->prox != NULL)
       {
           aux = aux->prox;
       }
       aux->prox = ingles;
   }
}

/*busca uma unidade na lista dicionario pelo id. Retorna a unidade*/
Unidade *buscarUnidade(Unidade *dicionario, int id)
{
   Unidade *aux = dicionario;
   while (aux != NULL && aux->id != id)
   {
       if (aux->id != id)
       {
           aux = aux->prox;
       }
```

```
    }
    return aux;
}

/*busca um nó na árvore raiz pela string raiz.portugues. Retorna a unidade*/
Arvore *buscarPalavraPortuguesEmArvore(Arvore *raiz, char portugues[MAX])
{
    Arvore *no = NULL;
    if (raiz != NULL)
    {
        if (strcmp(raiz->portugues, portugues) == 0)
        {
            no = raiz;
        }
        else
        {
            int primeiro = compararPalavras(portugues, raiz->portugues, 0);
            if (primeiro == 1)
                no = buscarPalavraPortuguesEmArvore(raiz->esquerda, portugues);
            else
                no = buscarPalavraPortuguesEmArvore(raiz->direita, portugues);
        }
    }
    return no;
}

/*Remove o nó. Nó é uma folha.
Retorno a nova raiz*/
Arvore *removerFolha(Arvore *no)
{
    free(no);
    return NULL;
}

/*Remove o nó. Nó tem um único filho.
Retorno a nova raiz*/
Arvore *removerNoUnicoFilho(Arvore *no)
{
    Arvore *filho;
```

```
    if (no->direita == NULL)
        filho = no->esquerda;
    else
        filho = no->direita;
    free(no);
    return filho;
}

/*Retorna o nó de maior descendente de raiz*/
Arvore *noMaiorElemento(Arvore *raiz)
{
    if (raiz->direita != NULL)
        noMaiorElemento(raiz->direita);
    else
        return raiz;
}

/*Remove o nó. Nó tem dois filhos.
Retorno a nova raiz*/
Arvore *removerNoDoisfilhos(Arvore *no)
{
    Arvore *maior = noMaiorElemento(no->esquerda);
    maior->direita = no->direita;
    Arvore *aux = no;
    no = aux->esquerda;
    free(aux);
    return no;
}

/*Retorna a quantidade de filhos que nó tem*/
int quantFilhos(Arvore *no)
{
    int filhos;
    if (no->direita == NULL && no->esquerda == NULL)
        filhos = 0;
    else if (no->direita != NULL && no->esquerda != NULL)
        filhos = 2;
    else
        filhos = 1;
```

```
   return filhos;
}

/*Deleta uma arvore se raiz->protugues for igual ao conteudo da string 'portugues'.
'Encontrei' deve receber 0. Se encontrar o elemento buscado, ele recebe 1.
Retorna a nova raiz*/
Arvore  *deletarPalavraPortuguesEmArvore(Arvore  *raiz,  char  portugues[MAX],  int
*encontrei)
{
   if (raiz != NULL)
   {
       if (strcmp(raiz->portugues, portugues) == 0)
       {   *encontrei = 1;
           if (quantFilhos(raiz) == 0)
               raiz = removerFolha(raiz);
           else if (quantFilhos(raiz) == 1)
               raiz = removerNoUnicoFilho(raiz);
           else
               raiz = removerNoDoisfilhos(raiz);
       }
       else
       {
           int primeiro = compararPalavras(portugues, raiz->portugues, 0);
           if (primeiro == 1)
                   raiz->esquerda = deletarPalavraPortuguesEmArvore(raiz->esquerda,
portugues, encontrei);
           else
                    raiz->direita = deletarPalavraPortuguesEmArvore(raiz->direita,
portugues, encontrei);
       }
   }
   return raiz;
}

/*Busca o conteudo da string 'portugues' em todas as unidades da lista'.
Se encontrar, retorna a arvore, tal qual arvore->portugues for igual ao conteudo da
string.
Se não, retorna NULL*/
Arvore      *buscarPalavraPortuguesEmDicionario(Unidade      *dicionario,      char
portugues[MAX])
```

```c
{
    Unidade *aux = dicionario;
    Arvore *no = NULL;
    while (aux != NULL && no == NULL)
    {
        no = buscarPalavraPortuguesEmArvore(aux->raiz, portugues);
        aux = aux->prox;
    }
    return no;
}

/*Imprime a lista de palavras inglês*/
void imprimirListaIngles(Lista *lista)
{
    Lista *aux = lista;
    if (aux == NULL)
        printf("lista vazia\n");
    else
    {
        printf("Ingles: ");
        while (aux != NULL)
        {
            if (aux != lista)
            {
                printf(", ");
            }
            printf("%s", aux->palavra);
            aux = aux->prox;
        }
        printf("\n");
    }
}

/*Imprime as informações de um nó da arvore.
Recebe o nó*/
void imprimirNoDaArvore(Arvore *no)
{
    printf("\nPortugues: %s\n", no->portugues);
    imprimirListaIngles(no->ingles);
```

```
}

/*Imprime todos os nós da arvore*/
void imprimirArvorePortuguesIngles(Arvore *raiz)
{
   if (raiz != NULL)
   {
       imprimirNoDaArvore(raiz);
       imprimirArvorePortuguesIngles(raiz->esquerda);
       imprimirArvorePortuguesIngles(raiz->direita);
   }
}

/*Imprime todas as informações de uma unidade.
Inclusive, todos nós da sua arvore*/
void imprimirUnidade(Unidade *unidade)
{
   printf("---UNIDADE %d:  %s\n", unidade->id, unidade->nome);
   imprimirArvorePortuguesIngles(unidade->raiz);
}

/*Imprime todas as unidades da lista dicionario*/
void imprimirDicionario(Unidade *dicionario)
{
   Unidade *aux = dicionario;
   while (aux != NULL)
   {
       printf("\n");
       imprimirUnidade(aux);
       aux = aux->prox;
   }
}

/*Imprime o nome e o id de todas as unidades na lista dicionario*/
void imprimirNomeUnidades(Unidade *dicionario)
{
   Unidade *aux = dicionario;
   while (aux != NULL)
   {
```

```
        printf("\n");
        printf("---UNIDADE %d:  %s\n", aux->id, aux->nome);
        aux = aux->prox;
    }
}

/*le um arquivo no formato proposto e armazena as informações nas structs.
Retorna 1 se não é possível ler o arquivo dicionario.txt
Retorna 0 se não houve erro*/
int lerArquivo(Unidade **dicionario)
{
    int erro = 1;
    FILE *arquivo = fopen("dicionario.txt", "r");
    if (arquivo != NULL)
    {
        erro = 0;
        char *palavra;
        int id = 0;
        char *ingles;
        while (!feof(arquivo))
        {
            Unidade *unidade;
            char frase[MAX];
            fgets(frase, MAX, arquivo);
            if (frase[0] == '%')
            {
                id++;
                  palavra = strtok(frase + 1, "\n");/*Imprime o nome e o id de todas
as unidades em lista dicionario*/
                unidade = criarUnidade(palavra, id);
                inserirUnidadeNoDicionario(dicionario, unidade);
            }
            else
            {
                ingles = strtok(frase, ":");
                palavra = strtok(NULL, ",\n");
                while (palavra)
                {
                      Arvore *arvore = buscarPalavraPortuguesEmArvore(unidade->raiz,
```

```
palavra);
                    if (arvore == NULL)
                    {
                        arvore = criarArvore(palavra);
                        inserirArvoreNaRaizDaUnidade(&(unidade->raiz), arvore);
                    }
                    Lista *listaIngles = criarPalavraIngles(ingles);
                    inserirPalavraIngles(&(arvore->ingles), listaIngles);
                    palavra = strtok(NULL, ",\n");
                }
            }
        }
    }
    return erro;
}

/*Pede ao usuario para digitar o id de uma unidade, e lê*/
void lerID(int *id)
{
    printf("Digite o id da unidade: ");
    scanf("%d", id);
}

/*Pede ao usuario para digitar uma palavra em português e lê*/
void lerPalavraPortugues(char palavra[MAX])
{
    printf("Digite a palavra em portugues: ");
    scanf("%s", palavra);
}

/*Menu com as opções. Retorna a opção escolhida pelo usuario*/
int menu()
{
    printf("\n**********MENU**********\n");
    printf("1 - Mostrar unidades\n");
    printf("2 - Mostrar dicionario\n");
    printf("3 - Pesquisar unidade especifica\n");
    printf("4 - Pesquisar palavra portugues\n");
    printf("5 - Deletar palavra\n");
```

```
    printf("0 - Sair\n");
    printf("Escolha uma opção:\n");
    int op;
    scanf("%d", &op);
    return op;
}

int main()
{
    Unidade *dicionario = NULL;
    int op, id, encontrei;
    Unidade *unidade;
    Arvore *arvore;
    char palavra[MAX];

    lerArquivo(&dicionario);

    do
    {
        op = menu();
        switch (op)
        {
        case 1:
            imprimirNomeUnidades(dicionario);
            break;
        case 2:
            imprimirDicionario(dicionario);
            break;
        case 3:
            lerID(&id);
            unidade = buscarUnidade(dicionario, id);
            if (unidade == NULL)
            {
                printf("unidade não encontrada\n");
            }
            else
            {
                imprimirUnidade(unidade);
            }
```

```
            break;
        case 4:
            lerPalavraPortugues(palavra);
            arvore = buscarPalavraPortuguesEmDicionario(dicionario, palavra);
            if (arvore == NULL)
            {
                printf("Palavra não encontrada\n");
            }
            else
            {
                imprimirNoDaArvore(arvore);
            }
            break;
        case 5:
            lerID(&id);
            unidade = buscarUnidade(dicionario, id);
            if (unidade == NULL)
            {
                printf("unidade não encontrada\n");
            }
            else
            {
                encontrei = 0;
                char palavra[MAX];
                lerPalavraPortugues(palavra);
                      unidade->raiz = deletarPalavraPortuguesEmArvore(unidade->raiz,
palavra, &encontrei);
                if(encontrei){
                    printf("Deletado!\n");
                }
                else{
                     printf("Palavra não encontrada\n");
                }
                break;
            }
        }
    } while (op != 0);
    return 0;
}
```

7.4 QUESTÃO 04

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX 100

struct lista
{
   char palavra[MAX];
   struct lista *prox;
};
typedef struct lista Lista;

struct arvore
{
   int altura;
   char portugues[MAX];
   struct lista *ingles;
   struct arvore *esquerda;
   struct arvore *direita;
};
typedef struct arvore Arvore;

struct unidade
{
   int id;
   char nome[MAX];
   struct arvore *raiz;
   struct unidade *prox;
};
typedef struct unidade Unidade;

Arvore *girarHorario(Arvore *raiz)
{
   Arvore *no = raiz->esquerda;
   raiz->esquerda = no->direita;
```

```
    no->direita = raiz;
    return no;
}

Arvore *girarAntiHorario(Arvore *raiz)
{
    Arvore *no = raiz->direita;
    raiz->direita = no->esquerda;
    no->esquerda = raiz;
    return no;
}

/*calculo o fator de balanceamento do nó*/
int fatorBalanceamento(Arvore *no)
{
    int fator = 0, esq, dir;
    if (no->esquerda == NULL)
        esq = -1;
    else
        esq = no->esquerda->altura;
    if (no->direita == NULL)
        dir = -1;
    else
        dir = no->direita->altura;
    fator = esq - dir;
    return fator;
}

//Calculo a altura do elemento raiz (com base na altura dos seus filhos) e retorno
ela.
int alturaElemento(Arvore *raiz)
{
    int altura = 1;
    int esquerda = -1;
    int direita = -1;

    if (raiz->esquerda != NULL)
        esquerda = raiz->esquerda->altura;
    if (raiz->direita != NULL)
```

```
        direita = raiz->direita->altura;
    altura = 1;
    if (esquerda > direita)
        altura += esquerda;
    else
        altura += direita;

    return altura;
}

/*calcula a altura da raiz e de seus descendentes e para cada um armazena em
raiz->altura*/
int calcularAlturaCadaElementoGalho(Arvore *raiz)
{
    int altura = -1;
    int direita = 0;
    int esquerda = 0;
    if (raiz != NULL)
    {
        altura = 1;

        direita += calcularAlturaCadaElementoGalho(raiz->direita);
        esquerda += calcularAlturaCadaElementoGalho(raiz->esquerda);
        if (esquerda > direita)
        {
            altura += esquerda;
            raiz->altura = altura;
        }
        else
        {
            altura += direita;
            raiz->altura = altura;
        }
    }
    return altura;
}

/*balanceio a raiz*/
Arvore *balancear(Arvore *raiz)
```

```
{
    if (fatorBalanceamento(raiz) > 0) //positivo
    {
        if (fatorBalanceamento(raiz->esquerda) < 0) //negativo
        {
            raiz->esquerda = girarAntiHorario(raiz->esquerda);
        }
        raiz = girarHorario(raiz);
    }
    else if (fatorBalanceamento(raiz) < 0) //negativo
    {
        if (fatorBalanceamento(raiz->direita) > 0) //positivo
        {
            raiz->direita = girarHorario(raiz->direita);
        }
        raiz = girarAntiHorario(raiz);
    }

    return raiz;
}

/*cria unidade e retorna.
Recebe a string nome que será armazenada em unidade->nome
e o id que será armazenado em unidade->id */
Unidade *criarUnidade(char nome[MAX], int id)
{
    Unidade *unidade = (Unidade *)calloc(1, sizeof(Unidade));
    strcpy(unidade->nome, nome);
    unidade->id = id;
    unidade->prox = NULL;
    unidade->raiz = NULL;
    return unidade;
}

/*Insire o nó unidade na lista dicionario*/
void inserirUnidadeNoDicionario(Unidade **dicionario, Unidade *unidade)
{
    if (*dicionario == NULL)
    {
```

```
        *dicionario = unidade;
    }
    else
    {
        Unidade *aux = *dicionario;
        while (aux->prox != NULL)
        {
            aux = aux->prox;
        }
        aux->prox = unidade;
    }
}

/*cria uma arvore, tal qual arvore->portugues recebe a string 'portugues'.
Retorna a arvore*/
Arvore *criarArvore(char portugues[MAX])
{
    Arvore *arvore = (Arvore *)calloc(1, sizeof(Arvore));
    arvore->direita = NULL;
    arvore->esquerda = NULL;
    arvore->ingles = NULL;
    arvore->altura = 0;
    strcpy(arvore->portugues, portugues);
    return arvore;
}

/*compara a palavra1 e a palavra 2 para averiguar qual vem primeiro em ordem
alfabetica.
retorna 1 se a palavra que vem primeiro for palavra1
retorna 2 se a palavra que vem primeiro for palavra2
retorna 0 se as duas palavras possuem os mesmos caracteres
A variavel i indica a posição do caracter a ser comparado. Deve receber 0 na
chamada da função*/
int compararPalavras(char palavra1[MAX], char palavra2[MAX], int i)
{
    int palavraQueVemPrimeiro = 0;
    if (strlen(palavra1) > i && strlen(palavra2) > i)
    {
        if (tolower(palavra1[i]) == tolower(palavra2[i]))
```

```
        {
            palavraQueVemPrimeiro = compararPalavras(palavra1, palavra2, i + 1);
        }
        else
        {
            if (tolower(palavra1[i]) < tolower(palavra2[i]))
                palavraQueVemPrimeiro = 1;
            if (tolower(palavra1[i]) > tolower(palavra2[i]))
                palavraQueVemPrimeiro = 2;
        }
    }
    return palavraQueVemPrimeiro;
}

/*Insere nó em raiz, se necessario, balanceia a raiz*/
void inserirArvoreNaRaizDaUnidade(Arvore **raiz, Arvore *arvore)
{
    if (*raiz == NULL)
    {
        *raiz = arvore;
    }
    else
    {
        int primeiro = compararPalavras(arvore->portugues, (*raiz)->portugues, 0);
        if (primeiro == 1)
            inserirArvoreNaRaizDaUnidade(&((*raiz)->esquerda), arvore);
        else
            inserirArvoreNaRaizDaUnidade(&((*raiz)->direita), arvore);
        (*raiz)->altura = alturaElemento(*raiz);
        if (abs(fatorBalanceamento(*raiz)) > 1)
        {
            *raiz = balancear(*raiz);
            calcularAlturaCadaElementoGalho(*raiz);
        }
    }
}

/*cria um nó de lista que conterá a palavra em ingles
tal qual no->palavra recebe a string 'palavra'.
```

```c
Retorna o no*/
Lista *criarPalavraIngles(char palavra[MAX])
{
   Lista *lista = (Lista *)calloc(1, sizeof(Lista));
   strcpy(lista->palavra, palavra);
   lista->prox = NULL;
   return lista;
}

/*insere o nó ingles  na lista*/
void inserirPalavraIngles(Lista **lista, Lista *ingles)
{
   if (*lista == NULL)
   {
       *lista = ingles;
   }
   else
   {
       Lista *aux = *lista;
       while (aux->prox != NULL)
       {
           aux = aux->prox;
       }
       aux->prox = ingles;
   }
}

/*busca uma unidade na lista dicionario pelo id. Retorna a unidade*/
Unidade *buscarUnidade(Unidade *dicionario, int id)
{
   Unidade *aux = dicionario;
   while (aux != NULL && aux->id != id)
   {
       if (aux->id != id)
       {
           aux = aux->prox;
       }
   }
   return aux;
```

```
}

/*Função recursiva que busca um nó na árvore raiz pela string raiz.portugues.*/
Arvore *buscarPalavraPortuguesEmArvore(Arvore *raiz, char portugues[MAX])
{
   Arvore *no = NULL;
   if (raiz != NULL)
   {
       if (strcmp(raiz->portugues, portugues) == 0)
       {
           no = raiz;
       }
       else
       {
           int primeiro = compararPalavras(portugues, raiz->portugues, 0);
           if (primeiro == 1)
               no = buscarPalavraPortuguesEmArvore(raiz->esquerda, portugues);
           else
               no = buscarPalavraPortuguesEmArvore(raiz->direita, portugues);
       }
   }
   return no;
}

/*Remove o nó. Nó é uma folha.
Retorno a nova raiz */
Arvore *removerFolha(Arvore *no)
{
   Arvore *aux = no;
   no = NULL;
   free(aux);
   return no;
}

/*Remove o nó. Nó tem um único filho.
Retorno a nova raiz*/
Arvore *removerNoUnicoFilho(Arvore *no)
{
   Arvore *filho;
```

```
    if (no->direita == NULL)
        filho = no->esquerda;
    else
        filho = no->direita;
    free(no);
    return filho;
}

/*Retorna o nó de maior descendente de raiz*/
Arvore *noMaiorElemento(Arvore *raiz)
{
    if (raiz->direita != NULL)
        noMaiorElemento(raiz->direita);
    else
        return raiz;
}

/*Retorna o maior descendente de raiz*/
Arvore *removerNoDoisfilhos(Arvore *no)
{
    Arvore *maior = noMaiorElemento(no->esquerda);
    maior->direita = no->direita;
    Arvore *aux = no;
    no = aux->esquerda;
    free(aux);
    return no;
}

/*Retorna a quantidade de filhos que nó tem*/
int quantFilhos(Arvore *no)
{
    int filhos;
    if (no->direita == NULL && no->esquerda == NULL)
        filhos = 0;
    else if (no->direita != NULL && no->esquerda != NULL)
        filhos = 2;
    else
        filhos = 1;
    return filhos;
```

```
}

/*Deleta uma arvore se raiz->protugues for igual ao conteudo da string 'portugues'.
'Encontrei' deve receber 0. Se encontrar o elemento buscado, ele recebe 1.
Retorna a nova raiz*/
Arvore *deletarPalavraPortuguesEmArvore(Arvore *raiz, char portugues[MAX], int
*encontrei)
{
   if (raiz != NULL)
   {
       if (strcmp(raiz->portugues, portugues) == 0)
       {
           *encontrei = 1;
           if (quantFilhos(raiz) == 0)
               raiz = removerFolha(raiz);
           else if (quantFilhos(raiz) == 1)
               raiz = removerNoUnicoFilho(raiz);
           else
               raiz = removerNoDoisfilhos(raiz);
           calcularAlturaCadaElementoGalho(raiz);
       }
       else
       {
           int primeiro = compararPalavras(portugues, raiz->portugues, 0);
           if (primeiro == 1)
                   raiz->esquerda = deletarPalavraPortuguesEmArvore(raiz->esquerda,
portugues, encontrei);
           else
                    raiz->direita = deletarPalavraPortuguesEmArvore(raiz->direita,
portugues, encontrei);
           if (abs(fatorBalanceamento(raiz)) > 1)
           {
               raiz = balancear(raiz);
               calcularAlturaCadaElementoGalho(raiz);
           }
       }
   }
   return raiz;
}
```

```c
/*Busca o conteudo da string 'portugues' em todas as unidades da lista'.
Se encontrar, retorna a arvore, tal qual arvore->portugues for igual ao conteudo da
string.
Se não, retorna NULL*/
Arvore      *buscarPalavraPortuguesEmDicionario(Unidade      *dicionario,      char
portugues[MAX])
{
   Unidade *aux = dicionario;
   Arvore *no = NULL;
   while (aux != NULL && no == NULL)
   {
       no = buscarPalavraPortuguesEmArvore(aux->raiz, portugues);
       aux = aux->prox;
   }
   return no;
}

/*Imprime a lista de palavras inglês*/
void imprimirListaIngles(Lista *lista)
{
   Lista *aux = lista;
   if (aux == NULL)
       printf("lista vazia\n");
   else
   {
       printf("Ingles: ");
       while (aux != NULL)
       {
           if (aux != lista)
           {
               printf(", ");
           }
           printf("%s", aux->palavra);
           aux = aux->prox;
       }
       printf("\n");
   }
}
```

```c
/*Imprime as informações de um nó da arvore.
Recebe o nó*/
void imprimirNo(Arvore *raiz)
{
   printf("\nPortugues: %s\n", raiz->portugues);
   imprimirListaIngles(raiz->ingles);
}

/*Imprime todos os nós da arvore*/
void imprimirArvorePortuguesIngles(Arvore *raiz)
{
   if (raiz != NULL)
   {
        imprimirNo(raiz);
        imprimirArvorePortuguesIngles(raiz->esquerda);
        imprimirArvorePortuguesIngles(raiz->direita);
   }
}

/*Imprime todas as informações de uma unidade.
Inclusive, todos nós da sua arvore*/
void imprimirUnidade(Unidade *unidade)
{
   printf("---UNIDADE %d:  %s\n", unidade->id, unidade->nome);
   imprimirArvorePortuguesIngles(unidade->raiz);
}

/*Imprime todas as unidades da lista dicionario*/
void imprimirDicionario(Unidade *dicionario)
{
   Unidade *aux = dicionario;
   while (aux != NULL)
   {
        printf("\n");
        imprimirUnidade(aux);
        aux = aux->prox;
   }
}
```

```c
/*Imprime o nome e o id de todas as unidades na lista dicionario*/
void imprimirNomeUnidades(Unidade *dicionario)
{
   Unidade *aux = dicionario;
   while (aux != NULL)
   {
       printf("\n");
       printf("---UNIDADE %d:  %s\n", aux->id, aux->nome);
       aux = aux->prox;
   }
}

/*le um arquivo no formato proposto e armazena as informações nas structs.
Retorna 1 se não é possível ler o arquivo dicionario.txt
Retorna 0 se não houve erro*/
int lerArquivo(Unidade **dicionario)
{
   int erro = 1;
   FILE *arquivo = fopen("dicionario.txt", "r");
   if (arquivo != NULL)
   {
       erro = 0;
       char *palavra;
       int id = 0;
       char *ingles;
       while (!feof(arquivo))
       {
           Unidade *unidade;
           char frase[MAX];
           fgets(frase, MAX, arquivo);
           if (frase[0] == '%')
           {
               id++;
               palavra = strtok(frase + 1, "\n");
               unidade = criarUnidade(palavra, id);
               inserirUnidadeNoDicionario(dicionario, unidade);
           }
           else
```

```
            {
                ingles = strtok(frase, ":");
                palavra = strtok(NULL, ",\n");
                while (palavra)
                {
                       Arvore *arvore = buscarPalavraPortuguesEmArvore(unidade->raiz,
palavra);
                    if (arvore == NULL)
                    {
                        arvore = criarArvore(palavra);
                        inserirArvoreNaRaizDaUnidade(&(unidade->raiz), arvore);
                    }
                    Lista *listaIngles = criarPalavraIngles(ingles);
                    inserirPalavraIngles(&(arvore->ingles), listaIngles);
                    palavra = strtok(NULL, ",\n");
                }
            }
        }
    }
    return erro;
}

/*Pede ao usuario para digitar o id de uma unidade, e lê*/
void lerID(int *id)
{
    printf("Digite o id da unidade: ");
    scanf("%d", id);
}

/*Pede ao usuario para digitar uma palavra em português e lê*/
void lerPalavraPortugues(char palavra[MAX])
{
    printf("Digite a palavra em portugues: ");
    scanf("%s", palavra);
}

/*Menu com as opções. Retorna a opção escolhida pelo usuario*/
int menu()
{
```

```
    printf("\n**********MENU**********\n");
    printf("1 - Mostrar unidades\n");
    printf("2 - Mostrar dicionario\n");
    printf("3 - Pesquisar unidade especifica\n");
    printf("4 - Pesquisar palavra portugues\n");
    printf("5 - Deletar palavra\n");
    printf("0 - Sair\n");
    printf("Escolha uma opção:\n");
    int op;
    scanf("%d", &op);
    return op;
}

int main()
{
    Unidade *dicionario = NULL;
    int op, id, encontrei;
    Unidade *unidade;
    Arvore *arvore;
    char palavra[MAX];

    int erro = lerArquivo(&dicionario);
    if (erro)
    {
        printf("arquivo dicionario.txt não encontrado\n");
    }
    else
    {
        do
        {
            op = menu();
            switch (op)
            {
            case 1:
                imprimirNomeUnidades(dicionario);
                break;
            case 2:
                imprimirDicionario(dicionario);
                break;
```

```
            case 3:
                lerID(&id);
                unidade = buscarUnidade(dicionario, id);
                if (unidade == NULL)
                {
                    printf("unidade não encontrada\n");
                }
                else
                {
                    imprimirUnidade(unidade);
                }
                break;
            case 4:
                lerPalavraPortugues(palavra);
                arvore = buscarPalavraPortuguesEmDicionario(dicionario, palavra);
                if (arvore == NULL)
                {
                    printf("Palavra não encontrada\n");
                }
                else
                {
                    imprimirNo(arvore);
                }
                break;
            case 5:
                lerID(&id);
                unidade = buscarUnidade(dicionario, id);
                if (unidade == NULL)
                {
                    printf("unidade não encontrada\n");
                }
                else
                {
                    encontrei = 0;
                    char palavra[MAX];
                    lerPalavraPortugues(palavra);
                      unidade->raiz = deletarPalavraPortuguesEmArvore(unidade->raiz,
palavra, &encontrei);
                    if (encontrei)
```

```
                    {
                        printf("Deletado!\n");
                    }
                    else
                    {
                        printf("Palavra não encontrada\n");
                    }
                    break;
                }
            }
        } while (op != 0);
    }
    return 0;
}
```