RELATÓRIO DE RESOLUÇÃO DO TRABALHO 1 CORRESPONDENTE A PRIMEIRA AVALIAÇÃO DA DISCIPLINA DE ESTRUTURAS DE DADOS II

Vinícius de Sousa Carvalho
viniciussccarvalho@ufpi.edu.br
Estruturas de Dados II - Juliana Oliveira de Carvalho

Resumo

Este relatório tem como principal fundamento a resolução das questões da primeira atividade avaliativa da disciplina de Estrutura de dados II, visando apresentar todos os aspectos necessários para o entendimento do problema, bem como sua solução. Todas as questões foram resolvidas com uso da linguagem de programação C, que alinhado à estrutura de dados das árvores binária, sem balanceamento ou com o balanceamento, conhecida como AVL, pode proporcionar a criação de programas capazes de solucionar as questões.

Palavra-chave: Árvores Binárias, AVL, Estruturas de Dados.

Introdução

As árvores binárias são estruturas de dados que armazenam informações em forma hierárquica. Cada elemento da árvore é chamado de nó e cada nó pode ter, no máximo, dois filhos. A árvore é composta por um nó raiz, que é o nó principal, e pelos nós filhos, que podem ter outros nós filhos e assim por diante. Uma árvore AVL é uma árvore binária de pesquisa balanceada. Isso significa que ela mantém seus nós balanceados de maneira a garantir que a altura da árvore seja sempre a menor possível, o que resulta em operações mais rápidas.

Os tópicos seguintes deste trabalho estão divididos de modo a apresentar as questões propostas, bem como a lógica necessária para a resolução da mesma, exemplos de execução dos programas, testes de eficiência entre a árvore binária sem balanceamento e com balanceamento (AVL), conclusão dos resultados alcançados e apêndice com todos os algoritmos utilizados. As questões da atividade estão divididas de modo a apresentar a solução de um mesmo problema usando os dois tipos de árvores mencionados anteriormente, possibilitando efetuar comparações de velocidade em inserções e busca de dados.

Hardware utilizado

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

Marca	Samsung
Sistema Operacional	Linux Ubuntu 22.04
Processador	Intel i3 7º geração
Memória RAM	4GB
Tipo de memória	DDR4 2400MHz

Seções Específicas

Os tópicos seguintes apresentam os enunciados das questões, acompanhados de sua resolução e testes de velocidade. Isso permitirá que os leitores compreendam a lógica por trás da solução e avaliem a eficiência do método utilizado.

Questão 01 e Questão 02

A questão 1 e 2 propõe um algoritmo capaz de criar 1000 números aleatórios e inseri-los em uma árvore binária, imprimindo para o usuário, a folha de maior e menor profundidade, bem como a diferença entre a folha de maior e a menor profundidade, verificando o tempo de busca e de inserção desses valores. Esse processo deve ser repetido 30 vezes.

Para solucionar esse problema, primeiro foi necessário gerar os números aleatórios a partir de duas funções para garantir a aleatoriedade. Para fazer isso, foi sorteado primeiramente um valor máximo para um range, e dentro desse range, sorteado um valor. Todos os valores gerados foram armazenados em um vetor para que seja reutilizado posteriormente. Após gerado os valores, as etapas de inserção e cálculo de maior e menor nível de folha são executadas. Para a etapa de inserção, a função percorre a árvore, fazendo comparações com os valores dos nós, até encontrar o espaço vazio (NULL) de algum nó folha presente na árvore. Cabe ressaltar que para chegar nesse ponto, as comparações seguirão a estrutura da árvore, ou seja, se o valor que quero inserir for menor que o valor do nó, o próximo nó analisado será o filho da esquerda; caso contrário, será o filho da direita; e caso esse valor já exista na árvore, nada será inserido. Tendo encontrado a posição para inserir, é alocado um espaço de memória para o nó e é inserido a informação nesse nó.

Após inserir todos os valores, foram realizados o cálculo da maior e a menor profundidade da folha, para que fosse possível calcular a diferença entre elas e armazenada em um vetor, para comparar quantas vezes cada diferença ocorreu nos 30 testes. Para testar a velocidade de busca, é feito uma série de testes buscando valores na árvore, enquanto o tempo é contabilizado. Por fim, para a criação do próximo teste, o vetor que contém os valores gerados é embaralhado para gerar a próxima árvore com os mesmos valores, mas com sua estrutura diferente.

Essa solução foi utilizada tanto para a questão 01, quanto para a questão 02, variando apenas as estruturas da árvore, de binária sem balanceamento na questão 01, para binária com balanceamento (AVL) na questão 02, o que implica que no momento de inserção de um novo nó, na Árvore AVL, acontece o cálculo do fator de balanceamento, e a depender dele, a árvore poderá ser rotacionada para manter balanceada.

Para efeito de teste dos algoritmos, foram utilizados arquivos .txt contendo sequências aleatórias de números. Foram efetuados teste com inserções de 100 mil a 900 mil valores, incrementando em cada teste, 100 mil novos números aleatórios, executando assim, 9 testes de velocidade de inserção e busca. Cabe mencionar que, para cada um desses 9 testes, foram gerados 30 árvores diferentes com os mesmos valores embaralhados e ao fim contabilizado a média de tempo de inserção e busca desses testes. A figura 01 e figura 02 apresenta, respectivamente, os gráficos de comparação dos tempos de execução (em milissegundos) das inserções e buscas nas árvores binárias sem balanceamento e com balanceamento.

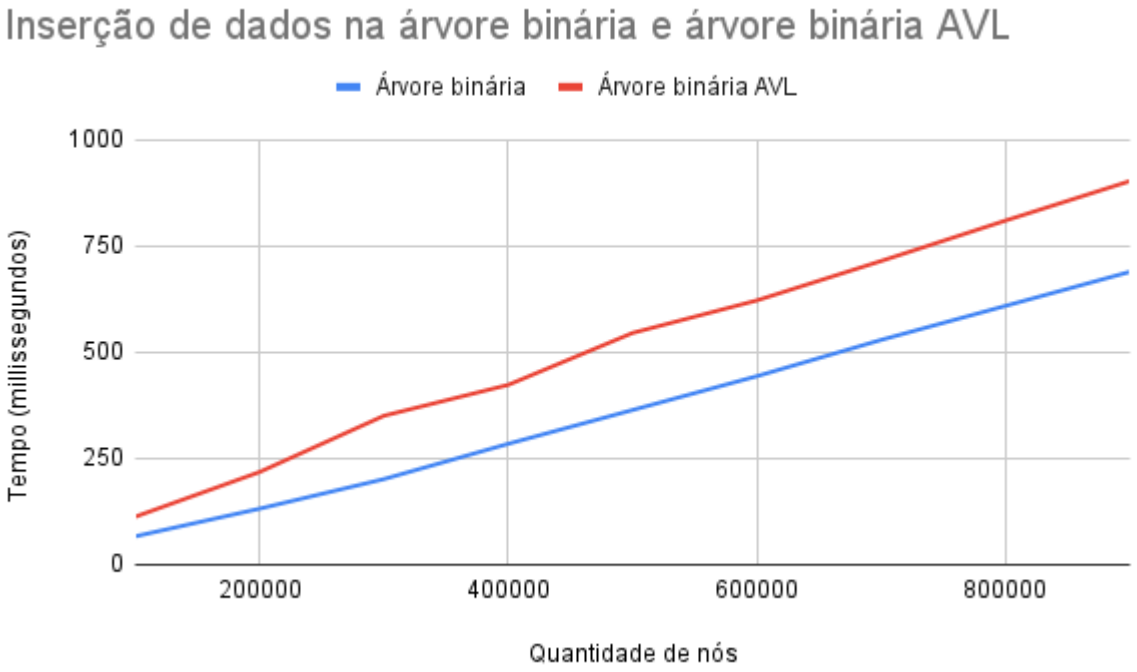

Figura 01 - Comparação do tempo de inserção das árvores binárias.

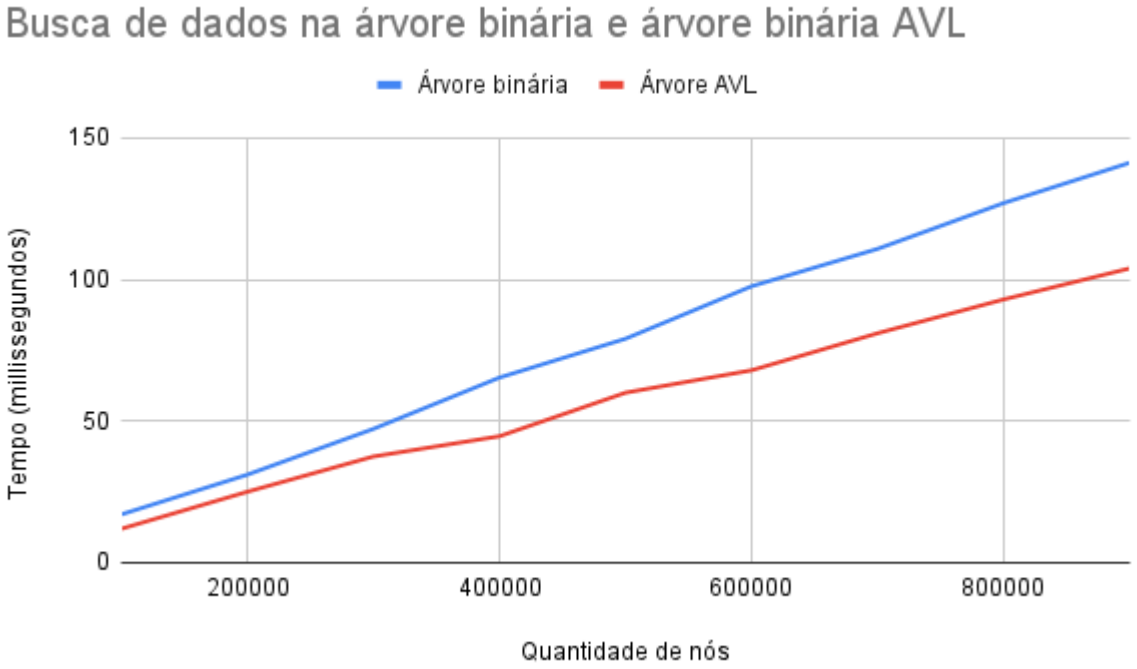

Figura 02 - Comparação do tempo de busca das árvores binárias.

A partir do gráfico da figura 01, é possível notar que o tempo de inserção na árvore binária AVL é maior do que na árvore sem balanceamento. Isso se dá pela necessidade de efetuar o balanceamento da árvore ao ponto que se insere um novo nó. Entretanto, na Figura 02, o tempo de busca da informação na árvore AVL é mais rápido do que na binária sem balanceamento. Isso ocorre devido ao balanceamento, que faz com que a árvore tenha a menor profundidade possível, fazendo com que leve menos tempo para que a busca chegue

na parte mais profunda da árvore (nós folhas de maior profundidade), tornando a árvore AVL mais rápida para buscas.

Questão 03 e Questão 04

As questões 03 e 04 propõe um algoritmo capaz de traduzir um livro-texto inglês-português para português-inglês com auxílio das árvores binárias. O dicionário é separado por unidades e cada linha possui uma palavra em inglês e suas equivalentes em protuguês. As palavras do dicionário são enviadas por meio de um arquivo .txt com a seguinte configuração:

- Os nomes das unidades são precedidos por um símbolo de porcentagem;
- Há somente uma entrada por linha;
- Uma palavra em Inglês é separada por dois pontos de sua(s) equivalentes em português. Caso haja mais de uma, serão separadas por uma vírgula.

Dessa forma, a estrutura do arquivo .txt se dá da seguinte maneira maneira:

```
% Unidade 1
salty:salgado
duck:pato
armadillo:tatu
bone:osso
residence:casa,residencia,moradia
% Unidade 2
frame:quadro
grape:uva
desktop:computador-de-mesa
cream:nata
woods:mata
home:casa
computer:computador
```

E o resultado final esperado presente na árvore binária é esse:

```
% Unidade 1
casa -> residence
moradia -> residence
osso -> bone
pato -> duck
residencia -> residence
salgado -> salty
tatu -> armadillo
% Unidade 2
casa -> home
computador -> computer
computador-de-mesa -> desktop
mata -> woods
nata -> cream
quadro -> frame
```

uva -> grape

Para solucionar esse problema, a estrutura usada foi árvores, possuindo a informação do nó, sendo uma palavra em português e uma lista encadeada para armazenar todas as palavras em inglês equivalentes a estas. Tendo a estrutura de armazenamento dos dados prontos, a próxima etapa foi criar um vetor que armazena cada raiz das árvores, já que cada árvore binária compõem uma unidade do arquivo txt. A Partir disso, é feito o processo de leitura do arquivo txt. Para fazer isso, usamos a função lerArquivo(), que itera linha a linha do arquivo e para cada linha, é chamado a função que encontra a palavra em inglês na linha (splitPalavraIngles) e a função que separa a(s) palavra(s) em português da linha (splitPalavraPort). Separadas as palavras, já é possível inseri-las em cada nó. É inserido todas as palavras em português daquela linha na árvore e, caso aquela palavra já existir na árvore, significa que possivelmente encontramos uma palavra diferente em inglês que traduz para a mesma palavra em português, portanto, para aquele nó, é inserido a palavra em inglês na lista encadeada, caso ela já não tenha sido inserido anteriormente. Dessa forma, foi possível inserir todas as palavras em português nas informações dos nós e suas equivalentes nas listas encadeadas presentes em cada nó.

O próximo passo consiste em resolver as atividades necessárias exigidas na questão, sendo elas, permitir o usuário informar uma unidade e imprimir todas as palavras desta unidade, bem como as palavra equivalentes em inglês de cada nó da unidade, imprimir todas as palavras em inglês equivalentes a uma palavra em portugues digitada pelo usuário e permitir que a remoção de palavras das unidades.

Para imprimir, bastando saber a unidade digitada, caso a mesma existisse, é chamado a função que imprime os valores da árvore in-ordem, o que garante que os nós serão apresentados em ordem alfabética, enviando como parâmetro a posição do vetor que contém a raiz da unidade desejada. Para imprimir as equivalentes de uma palavra em português, dada uma palavra, é feita a busca entre os nós da árvore e caso encontrado, é retornado o nó que contém a palavra em português e suas equivalentes. Em seguida, basta imprimir na tela. Por fim, para a remoção, é efetuado também a busca e ao encontrar, é feita a "desconexão" daquele nó da árvore e depois a liberação do espaço de memória alocado.

Para efetuar os devidos testes de performance dos algoritmos, foi necessário criarmos um exemplo de entrada com quantidade significativa de unidades e palavras. Como não há um .txt disponível para uso, usamos um arquivo que continha palavras em inglês e junto da biblioteca do google translate presente no python, criamos um programa capaz de ler as palavras presente neste arquivo e traduzi-las para um arquivo .txt no formato de entrada esperado pelo programa. Dessa forma, conseguimos criar um exemplo de teste com 28774 palavras. Cabe ressaltar, que este arquivo foi criado com uso de um tradutor e assim, existem no mesmo, palavras em inglês que possuem a mesma tradução no português, portanto, ao inserir na árvore, não haverá exatamente a mesma quantidade de linhas e nós presentes na árvore binária. Para aprimorar ainda mais o teste, foi divido esse arquivo em 9 testes, cada um deles com uma quantidade a mais de unidades. Logo, temos, teste 01 com 1 unidade, teste 02 com 2 unidades e assim sucessivamente. Isso garante que possamos avaliar a eficiente de inserção e busca das árvores.

A figura 03 apresenta a comparação do tempo de inserção da árvore binária comum com a árvore binária AVL. É possível notar, que diferente no caso anterior, dessa vez o tempo de inserção da árvore binária comum é maior do que o da árvore AVL. Normalmente, o tempo de inserção da AVL é maior, devido ao seu processo de balanceamento dos nós, entretanto, para esse caso, o arquivo txt usado, estava em ordem alfabética e por isso, cada

inserção sempre iria para o nó folha mais à direita da árvore. Por não existir o balanceamento na árvore binária comum, a mesma se tornou ineficiente no processo de inserção e busca, se tornando praticamente uma lista encadeada. Por esse fato, seu tempo de inserção se tornou muito maior do que devia ser para uma árvore. Essa questão, é um dos grandes problemas da árvore binária comum, o que foi um dos motivos para a existência de novos tipos de árvores, como a AVL, que efetua o balanceamento para a árvore não ficar desbalanceada a esse ponto.

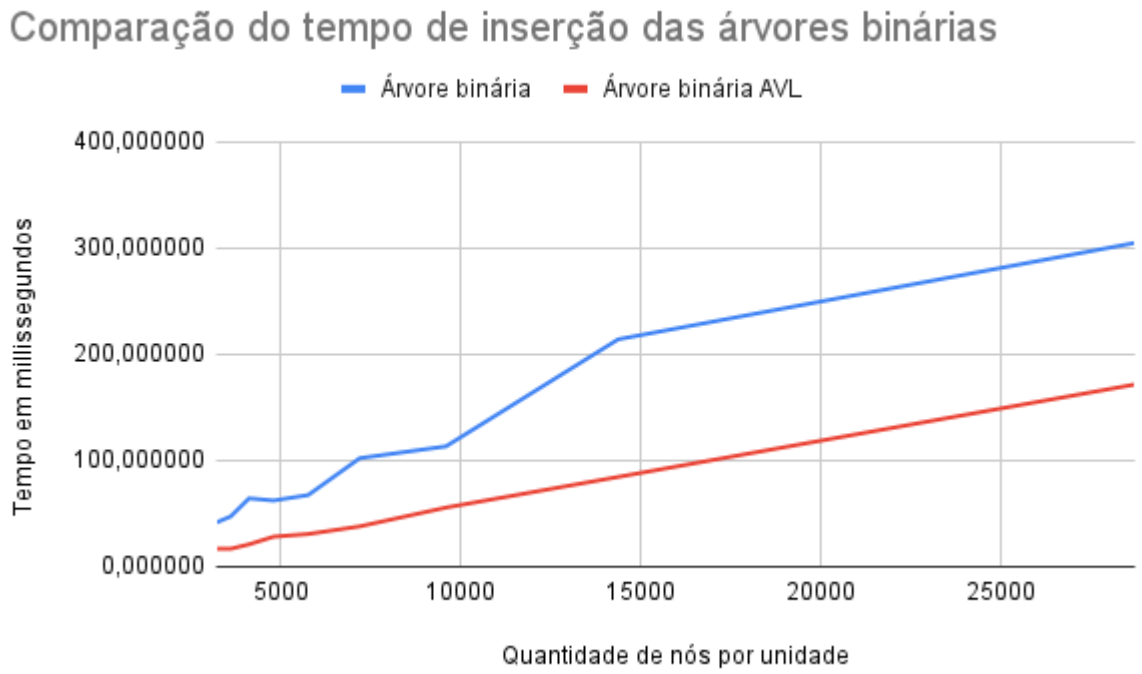

Figura 03 - Comparação do tempo de inserção das árvores binárias.

A figura 04 apresenta o tempo de busca nas árvores. Notem também, que o tempo de busca da AVL se manteve abaixo da binária sem balanceamento, o que é um comportamento esperado, já que o balanceamento reduz a profundidade da árvore para o menor valor possível.

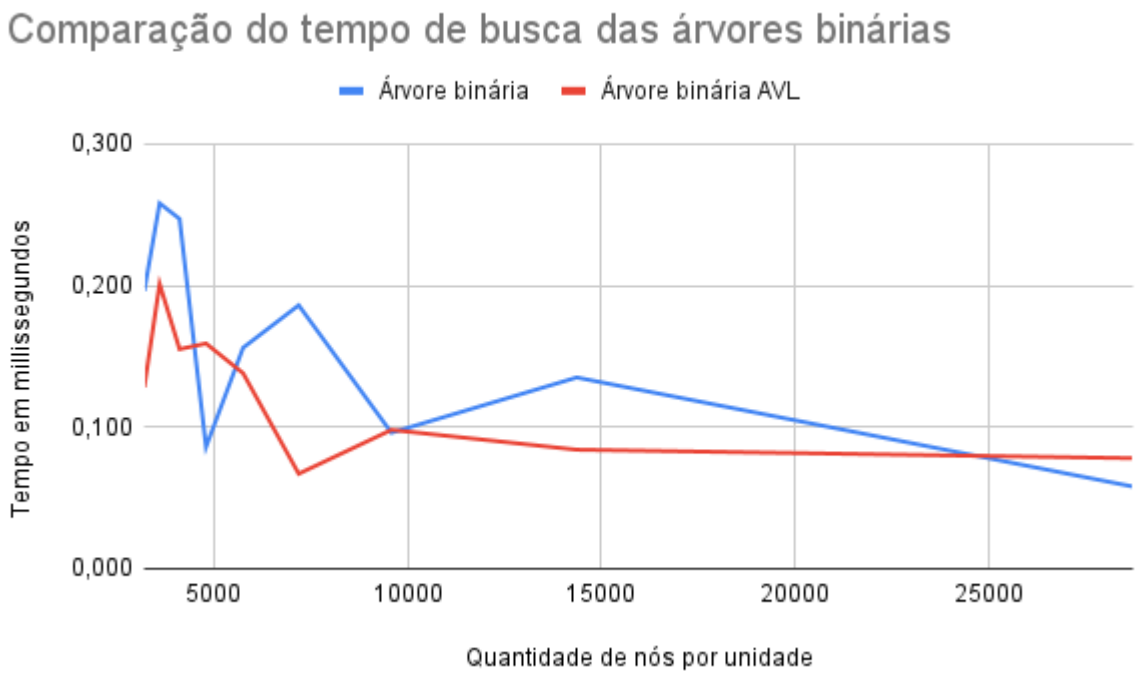

Figura 04 - Comparação do tempo de busca das árvores binárias.

Conclusão

Através dos experimentos, foi possível analisar as características e desempenho das diferentes estruturas de árvores binárias, permitindo avaliar suas vantagens e desvantagens, e assim, escolher a estrutura mais adequada para aplicações específicas. A análise dos resultados obtidos permitiu concluir que as árvores binárias são uma ferramenta valiosa para a organização e busca de dados. Os dados nos permitem concluir que a árvore AVL é mais eficiente do que a árvore binária sem balanceamento, devido ao seu alto grau de balanceamento, garantindo complexidade de tempo constante para as operações de busca, inserção e remoção de elementos, além de ser menos propensa a se tornar desequilibrada.

Apêndice

Essa seção é composta pelos algoritmos criados para a resolução das questões mencionadas nos tópicos anteriores.

Questão 01

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define qtdNOS 1000 // quantidade de nós que da arvore
#define qtdEXPERIMENTOS 30
#define qtdBUSCAS 1000
```

```c
#define qtdRANGES 1000

typedef struct ArvoreB arvoreB;
struct ArvoreB
{
    arvoreB *esq;
    arvoreB *dir;
    int valor;
};

arvoreB *alocaNo(int item);
int insereNo(arvoreB **raiz, int valor);
void preencheArvore(arvoreB **raiz, int *valoresNos);
void liberaNo(arvoreB *no);
void liberaArvore(arvoreB **raiz);
int procuraNo(arvoreB *raiz, int valor);
int rangeNumeros();
int numeroNoRange(int range);
int *geraNos();
int maiorNivelFolha(arvoreB *raiz);
int ehFolha(arvoreB *raiz);
void menorNivelFolha(arvoreB *raiz, int nivel, int
*nivelMenorFolha);
void embaralhaVetorNos(int *valoresNos);

int main()
{
    srand(time(NULL));
    arvoreB *raiz;
    raiz = NULL;

    double tempoInsercao[qtdEXPERIMENTOS];
    int diferencaNiveisAtual;
    int diferencasNiveisFolhas[qtdEXPERIMENTOS];
    int qtdNiveisFolhas[qtdEXPERIMENTOS];
    int *valoresNos = geraNos();

    int i, j, k, novoValorDifFolhas;
    for (i = 0; i < qtdEXPERIMENTOS; i++)
    {
        diferencasNiveisFolhas[i] = -1;
        qtdNiveisFolhas[i] = 0;
```

```
    }

    for (i = 0; i < qtdEXPERIMENTOS; i++)
    {
        printf("Experimento numero %d\n", i + 1);
        clock_t inicio = clock();
        preencheArvore(&raiz, valoresNos);
        clock_t fim = clock();
        tempoInsercao[i] = (double)(fim - inicio) /
CLOCKS_PER_SEC;

        printf("Tempo para inserir %d nos na arvore : %lf
segundos\n", qtdNOS, tempoInsercao[i]);
        printf("Maior nivel folha: %d\n", maiorNivelFolha(raiz) -
1);

        int nivelMenorFolha = -1;
        menorNivelFolha(raiz, 0, &nivelMenorFolha);

        printf("Nivel de menor folha: %d\n", nivelMenorFolha);
        diferencaNiveisAtual = (maiorNivelFolha(raiz) - 1) -
nivelMenorFolha;

        // contar a quantidade de vezes que cada diferenca da
maior folha pela menor aparece nos testes
        j = 0;
        novoValorDifFolhas = 1;
        // verifica se a nova diferenca encontrada já existe no
vetor de diferencas
        while (diferencasNiveisFolhas[j] != -1)
        {
            if (diferencasNiveisFolhas[j] == diferencaNiveisAtual)
            {
                //  diferenca ja existe, incrementa o vetor que
guarda
                //  a quantidade de vezes que essa diferenca
apareceu apareceu
                qtdNiveisFolhas[j] += 1;
                novoValorDifFolhas = 0;
            }

            j++;
```

```
        }

        // Verifica se o while de cima percorreu todos os valores
do vetor
        // Se sim, significa que a diferenca dessa arvore ainda
nao existe
        // no vetor de diferencas, então adicionar
        if (diferencasNiveisFolhas[j] == -1 && novoValorDifFolhas)
        {
            diferencasNiveisFolhas[j] = diferencaNiveisAtual;
            qtdNiveisFolhas[j] += 1;
        }

        // Escolher valores aleatorios para buscar
        int *valoresBusca = (int *)malloc(qtdBUSCAS *
sizeof(int));

        // escolhe aleatoriamene valores para buscar na arvore
        for (k = 0; k < qtdBUSCAS; k++)
        {
            int aux = rand() % qtdNOS;
            valoresBusca[k] = valoresNos[aux];
        }

        double tempoBusca;
        int buscas = 0, encontrou = 0;
        inicio = clock();

        // efetua varias buscas na arvore
        while (buscas < qtdBUSCAS)
        {
            encontrou = procuraNo(raiz, valoresBusca[buscas]);
            buscas++;
        }

        fim = clock();
        tempoBusca = (double)(fim - inicio) / CLOCKS_PER_SEC;
        printf("Tempo para buscar %d nos na arvore : %lf
segundos\n", qtdBUSCAS, tempoBusca);

        liberaArvore(&raiz); // libera o espaco de memoria alocado
para a arvore
```

```
        free(valoresBusca);

printf("-=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--
=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--=--=-\n");
        printf("\n");
        embaralhaVetorNos(valoresNos);
    }

    // imprime todas as diferencas encontradas e a quantidade de
vezes que cada uma apareceu
    for (i = 0; i < qtdEXPERIMENTOS; i++)
    {
        if (diferencasNiveisFolhas[i] != -1)
        {
            printf("Diferenca %d ocorreu %d vezes\n",
diferencasNiveisFolhas[i], qtdNiveisFolhas[i]);
        }
    }

    free(valoresNos);

    return 0;
}

// gera um numero aleatorio para ser o range máximo
int rangeNumeros()
{
    int range = rand() % qtdRANGES + 1;
    return range;
}

// gera um numero aleatorio dentro do range maximo gerado
int numeroNoRange(int range)
{
    int num;
    num = rand() % 100 + ((range - 1) * 100 + 1);
    return num;
};

// gera um vetor de valores aleatorios gerados com metodos de
sorteio
int *geraNos()
```

```
{
    int *valoresNos, i, range;
    valoresNos = (int *)malloc(qtdNOS * sizeof(int));
    for (i = 0; i < qtdNOS; i++)
    {
        range = rangeNumeros();
        valoresNos[i] = numeroNoRange(range);
    }
    return valoresNos;
}

// Aloca um espaco de memoria para armazenar um no da arvore
arvoreB *alocaNo(int item)
{
    arvoreB *novo = (arvoreB *)malloc(1 * sizeof(arvoreB));
    novo->esq = NULL;
    novo->dir = NULL;
    novo->valor = item;
    return novo;
}

// Insere um no na arvore
int insereNo(arvoreB **raiz, int valor)
{
    int inseriu = 0;
    if (*raiz == NULL)
    {
        *raiz = alocaNo(valor);
        inseriu = 1;
    }
    if (valor < (*raiz)->valor)
    {
        inseriu = insereNo(&((*raiz)->esq), valor);
    }
    else if (valor > (*raiz)->valor)
    {
        inseriu = insereNo(&((*raiz)->dir), valor);
    }
    return inseriu;
}

// Insere todos os valores do vetor na arvore
```

```c
void preencheArvore(arvoreB **raiz, int *valoresNos)
{
    int i, inseriu;
    for (i = 0; i < qtdNOS; i++)
        inseriu = insereNo(raiz, valoresNos[i]);
}

// libera um no da arvore
void liberaNo(arvoreB *no)
{
    if (no != NULL)
    {
        liberaNo(no->esq);
        liberaNo(no->dir);
        no = NULL;
        free(no);
    }
}

// libera a raiz da arvore
void liberaArvore(arvoreB **raiz)
{
    if (*raiz != NULL)
    {
        liberaNo(*raiz);
        *raiz = NULL;
        free(*raiz);
    }
}

// encontra o nivel da folha de maior profundidade
int maiorNivelFolha(arvoreB *raiz)
{
    int resultado;
    if (raiz == NULL)
        resultado = 0;
    else
    {
        int profundidadeEsq = maiorNivelFolha(raiz->esq);
        int profundidadeDir = maiorNivelFolha(raiz->dir);
        resultado = (profundidadeEsq > profundidadeDir) ?
profundidadeEsq + 1 : profundidadeDir + 1;
```

```
    }
    return resultado;
}

// verifica se o no passado por parametro eh uma folha
int ehFolha(arvoreB *raiz)
{
    int ehFolha = 0;
    if (raiz->esq == NULL && raiz->dir == NULL)
    {
        ehFolha = 1;
    }
    return ehFolha;
}

// encontra o nivel da folha de menor profundidade
void menorNivelFolha(arvoreB *raiz, int nivel, int
*nivelMenorFolha)
{

    if (raiz != NULL)
    {
        if (ehFolha(raiz))
        {
            if (*nivelMenorFolha > 0)
            {
                if (nivel < *nivelMenorFolha)
                {
                    *nivelMenorFolha = nivel;
                }
            }
            else
            {
                *nivelMenorFolha = nivel;
            }
        }
        else
        {
            nivel += 1;
            menorNivelFolha(raiz->esq, nivel, nivelMenorFolha);
            menorNivelFolha(raiz->dir, nivel, nivelMenorFolha);
        }
```

```
    }
}

// busca por um no na arvore
int procuraNo(arvoreB *raiz, int valor)
{
    int encontrou = 0;
    if (raiz != NULL)
    {
        if (raiz->valor == valor)
        {
            encontrou = 1;
        }
        else if (valor < raiz->valor)
        {
            encontrou = procuraNo(raiz->esq, valor);
        }
        else
        {
            encontrou = procuraNo(raiz->dir, valor);
        }
    }
    return encontrou;
}

// embaralha o vetor de numeros aleatorio para efetuar testes
diferentes na arvore
void embaralhaVetorNos(int *valoresNos)
{
    int novaPosicao = rand() % qtdNOS;
    int i, valorPosicaoAtual;
    for (i = 0; i < qtdNOS; i++)
    {
        valorPosicaoAtual = valoresNos[i];
        valoresNos[i] = valoresNos[novaPosicao];
        valoresNos[novaPosicao] = valorPosicaoAtual;
    }
}
```

Questão 02

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

```c
#include <time.h>
#include <math.h>
#define qtdNOS 1000
#define qtdEXPERIMENTOS 30
#define qtdBUSCAS 1000
#define qtdRANGES 1000

typedef struct ArvoreB arvoreB;
struct ArvoreB
{
    arvoreB *esq;
    arvoreB *dir;
    int valor;
    int altura;
};

int *geraNos();
arvoreB *alocaNo(int item);
int insereAVL(arvoreB **raiz, int valor);
int fatorBalanceamento(arvoreB *raiz);
void rotacaoEsquerda(arvoreB **raiz);
void rotacaoDireita(arvoreB **raiz);
int calculaAltura(arvoreB *raiz);
int alturaNo(arvoreB *raiz);
int maiorAltura(int a, int b);
int sotemUmFilho(arvoreB *raiz, arvoreB **noFilho);
int rangeNumeros();
int numeroNoRange(int range);
void preencheArvore(arvoreB **raiz, int *valoresNos);
void liberaNo(arvoreB *no);
void liberaArvore(arvoreB **raiz);
int maiorNivelFolha(arvoreB *raiz);
int ehFolha(arvoreB *raiz);
int procuraNo(arvoreB *raiz, int valor);
void menorNivelFolha(arvoreB *raiz, int nivel, int
*nivelMenorFolha);
void embaralhaVetorNos(int *valoresNos);

int main()
{
    printf("1...2...3...Valendo\n");
    srand(time(NULL));
```

```
    arvoreB *raiz;
    raiz = NULL;

    int *valoresNos = geraNos();

    double tempoInsercao[qtdEXPERIMENTOS];
    int diferencaNiveisAtual;
    int diferencasNiveisFolhas[qtdEXPERIMENTOS];
    int qtdNiveisFolhas[qtdEXPERIMENTOS];

    int i, j, k, novoValorDifFolhas;
    for (i = 0; i < qtdEXPERIMENTOS; i++)
    {
        diferencasNiveisFolhas[i] = -1;
        qtdNiveisFolhas[i] = 0;
    }

    for (i = 0; i < qtdEXPERIMENTOS; i++)
    {
        printf("Experimento numero %d\n", i + 1);
        clock_t inicio = clock();
        preencheArvore(&raiz, valoresNos);
        clock_t fim = clock();
        tempoInsercao[i] = (double)(fim - inicio) /
CLOCKS_PER_SEC;
        printf("Tempo para inserir %d nos na arvore : %lf
segundos\n", qtdNOS, tempoInsercao[i]);

        printf("Maior nivel folha: %d\n", maiorNivelFolha(raiz) -
1);
        int nivelMenorFolha = -1;
        menorNivelFolha(raiz, 0, &nivelMenorFolha);
        printf("Nivel de menor folha: %d\n", nivelMenorFolha);
        diferencaNiveisAtual = (maiorNivelFolha(raiz) - 1) -
nivelMenorFolha;

        j = 0;
        novoValorDifFolhas = 1;
        while (diferencasNiveisFolhas[j] != -1)
        {
            if (diferencasNiveisFolhas[j] == diferencaNiveisAtual)
            {
```

```
                qtdNiveisFolhas[j] += 1;
                novoValorDifFolhas = 0;
            }

            j++;
        }

        if (diferencasNiveisFolhas[j] == -1 && novoValorDifFolhas)
        {
            diferencasNiveisFolhas[j] = diferencaNiveisAtual;
            qtdNiveisFolhas[j] += 1;
        }

        int *valoresBusca = (int *)malloc(qtdBUSCAS *
sizeof(int));

        for (k = 0; k < qtdBUSCAS; k++)
        {
            int aux = rand() % qtdNOS + 1;
            valoresBusca[k] = valoresNos[aux];
        }

        double tempoBusca;
        int terminouBusca = 0, encontrou = 0;
        inicio = clock();

        while (terminouBusca < qtdBUSCAS)
        {
            encontrou = procuraNo(raiz, valoresBusca[i]);
            terminouBusca++;
        }

        fim = clock();
        tempoBusca = (double)(fim - inicio) / CLOCKS_PER_SEC;
        printf("Tempo para buscar %d nos na arvore : %lf
segundos\n", qtdBUSCAS, tempoBusca);

        liberaArvore(&raiz);
        free(valoresBusca);

printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
```

```
        printf("\n");
        embaralhaVetorNos(valoresNos);
    }

    for (i = 0; i < qtdEXPERIMENTOS; i++)
    {
        if (diferencasNiveisFolhas[i] != -1)
        {
            printf("Diferenca %d ocorreu %d vezes\n",
diferencasNiveisFolhas[i], qtdNiveisFolhas[i]);
        }
    }

    free(valoresNos);

    return 0;
}

// gera um numero aleatorio para ser o range máximo
int rangeNumeros()
{
    int range = rand() % qtdRANGES + 1;
    return range;
}

// gera um numero aleatorio dentro do range maximo gerado
int numeroNoRange(int range)
{
    int num;
    num = rand() % 100 + ((range - 1) * 100 + 1);
    return num;
};

// gera um vetor de valores aleatorios gerados com metodos de
sorteio
int *geraNos()
{
    int *valoresNos, i, range;
    valoresNos = (int *)malloc(qtdNOS * sizeof(int));
    for (i = 0; i < qtdNOS; i++)
    {
        range = rangeNumeros();
```

```
        valoresNos[i] = numeroNoRange(range);
    }
    return valoresNos;
}

// Aloca um espaco de memoria para armazenar um no da arvore AVL
arvoreB *alocaNo(int item)
{
    arvoreB *novo;
    novo = (arvoreB *)malloc(sizeof(arvoreB));
    if (novo)
    {
        novo->esq = NULL;
        novo->dir = NULL;
        novo->valor = item;
        novo->altura = 0;
    }
    return novo;
}

// Insere todos os valores do vetor na arvore
void preencheArvore(arvoreB **raiz, int *valoresNos)
{
    int i, inseriu;
    for (i = 0; i < qtdNOS; i++)
        inseriu = insereAVL(raiz, valoresNos[i]);
}

// libera um no da arvore
void liberaNo(arvoreB *no)
{
    if (no != NULL)
    {
        liberaNo(no->esq);
        liberaNo(no->dir);
        no = NULL;
        free(no);
    }
}

// libera a raiz da arvore
void liberaArvore(arvoreB **raiz)
```

```
{
    if (*raiz != NULL)
    {
        liberaNo(*raiz);
        *raiz = NULL;
        free(*raiz);
    }
}

// encontra o nivel da folha de menor profundidade
int maiorNivelFolha(arvoreB *raiz)
{
    int resultado;
    if (raiz == NULL)
        resultado = 0;
    else
    {
        int profundidadeEsq = maiorNivelFolha(raiz->esq);
        int profundidadeDir = maiorNivelFolha(raiz->dir);
        resultado = (profundidadeEsq > profundidadeDir) ?
profundidadeEsq + 1 : profundidadeDir + 1;
    }
    return resultado;
}

// encontra o nivel da folha de menor profundidade
void menorNivelFolha(arvoreB *raiz, int nivel, int
*nivelMenorFolha)
{

    if (raiz != NULL)
    {
        if (ehFolha(raiz))
        {
            if (*nivelMenorFolha > 0)
            {
                if (nivel < *nivelMenorFolha)
                {
                    *nivelMenorFolha = nivel;
                }
            }
            else
```

```
            {
                *nivelMenorFolha = nivel;
            }
        }
        else
        {
            nivel += 1;
            menorNivelFolha(raiz->esq, nivel, nivelMenorFolha);
            menorNivelFolha(raiz->dir, nivel, nivelMenorFolha);
        }
    }
}

// busca por um no na arvore
int procuraNo(arvoreB *raiz, int valor)
{
    int encontrou = 0;
    if (raiz != NULL)
    {
        if (raiz->valor == valor)
        {
            encontrou = 1;
        }
        else if (valor < raiz->valor)
        {
            encontrou = procuraNo(raiz->esq, valor);
        }
        else
        {
            encontrou = procuraNo(raiz->dir, valor);
        }
    }
    return encontrou;
}

// embaralha o vetor de numeros aleatorio para efetuar testes
diferentes na arvore
void embaralhaVetorNos(int *valoresNos)
{
    int novaPosicao = rand() % qtdNOS;
    int i, valorPosicaoAtual;
    for (i = 0; i < qtdNOS; i++)
```

```
    {
        valorPosicaoAtual = valoresNos[i];
        valoresNos[i] = valoresNos[novaPosicao];
        valoresNos[novaPosicao] = valorPosicaoAtual;
    }
}

// calcula o fator de balanceamento de um no
int fatorBalanceamento(arvoreB *raiz)
{
    int fb = 0;
    if (raiz != NULL)
    {
        fb = (alturaNo(raiz->esq) - alturaNo(raiz->dir));
    }
    return fb;
}

// rotaciona um no da arvore para a direita
void rotacaoDireita(arvoreB **raiz)
{
    arvoreB *aux;
    aux = (*raiz)->esq;
    (*raiz)->esq = aux->dir;
    aux->dir = *raiz;
    if (*raiz != NULL){
        (*raiz)->altura = calculaAltura(*raiz);
    }
    if (aux != NULL){
        aux->altura = calculaAltura(aux);
    }
    *raiz = aux;
}

// rotaciona um no da arvore para a esquerda
void rotacaoEsquerda(arvoreB **raiz)
{
    arvoreB *aux;
    aux = (*raiz)->dir;
    (*raiz)->dir = aux->esq;
    aux->esq = *raiz;
    if (*raiz != NULL){
```

```
        (*raiz)->altura = calculaAltura(*raiz);
    }
    if (aux != NULL){
        aux->altura = calculaAltura(aux);
    }
    *raiz = aux;
}

// retorna a altura do no passado por parametro
int alturaNo(arvoreB *raiz)
{
    int altura = -1;
    if (raiz != NULL)
        altura = raiz->altura;
    return altura;
}

// calcula a altura de um no
int calculaAltura(arvoreB *raiz)
{
    int alt_esq = alturaNo(raiz->esq);
    int alt_dir = alturaNo(raiz->dir);
    int altura_no = maiorAltura(alt_esq, alt_dir) + 1;
    return altura_no;
}

// retorna a maior altura entre oas duas altura passadas por
parâmetro
int maiorAltura(int a, int b)
{
    return (a > b) ? a : b;
}

// insere os nos na arvore AVL
int insereAVL(arvoreB **raiz, int valor)
{
    int inseriu = 0; // Que o elemento foi inserido
    if (*raiz == NULL)
    {
        *raiz = alocaNo(valor);
        inseriu = 1;
    }
```

```
    else
    {
        if (valor < (*raiz)->valor)
            inseriu = insereAVL(&((*raiz)->esq), valor);
        else if (valor > (*raiz)->valor)
            inseriu = insereAVL(&((*raiz)->dir), valor);
    }
    // balanceamento
    if (inseriu == 1)
    {
        int fb;
        fb = fatorBalanceamento(*raiz);
        if (fb == 2)
        {
            if (fatorBalanceamento((*raiz)->esq) == 1)
            {
                rotacaoDireita(raiz); // balanceamento
                //calculaAltSubArvore(*raiz);
                (*raiz)->dir->altura =
calculaAltura((*raiz)->dir);
            }
            else
            {
                rotacaoEsquerda(&((*raiz)->esq)); // ajustar galho
da esq
                rotacaoDireita(raiz);             // balanceamento
                (*raiz)->dir->altura =
calculaAltura((*raiz)->dir);
                (*raiz)->esq->altura =
calculaAltura((*raiz)->esq);
                //calculaAltSubArvore(*raiz);
            }
        }
        else if (fb == -2)
        {
            if (fatorBalanceamento((*raiz)->dir) == -1)
            {
                rotacaoEsquerda(raiz); // balanceamento
                //calculaAltSubArvore(*raiz);
                (*raiz)->esq->altura =
calculaAltura((*raiz)->esq);
            }
```

```
            else
            {
                rotacaoDireita(&((*raiz)->dir)); // ajustar galho
da dir
                rotacaoEsquerda(raiz);           // balanceamento
                //calculaAltSubArvore(*raiz);
                (*raiz)->esq->altura =
calculaAltura((*raiz)->esq);
                (*raiz)->dir->altura =
calculaAltura((*raiz)->dir);
            }
        }
        (*raiz)->altura = calculaAltura(*raiz);
    }
    return inseriu;
}

// verifica se o no passado por parametro eh uma folha
int ehFolha(arvoreB *raiz)
{
    int folha = 0;
    if (raiz->esq == NULL && raiz->dir == NULL)
    {
        folha = 1;
    }
    return folha;
}

// verifica se o no tem apenas um filho
int sotemUmFilho(arvoreB *raiz, arvoreB **noFilho)
{
    int umFilho = 0;
    if (raiz->esq != NULL && raiz->dir == NULL)
    {
        umFilho++;
        *noFilho = raiz->esq;
    }
    if (raiz->esq == NULL && raiz->dir != NULL)
    {
        umFilho++;
        *noFilho = raiz->dir;
    }
```

```
    return umFilho;
}
```

Questão 03

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define MAX_STRING 30
#define MAX_LINE 300

typedef struct Lista lista;
struct Lista{
    char palavraIngles[MAX_STRING];
    lista *prox;
};

typedef struct ArvoreB arvoreB;
struct ArvoreB
{
    char palavraPortugues[MAX_STRING];
    lista* equivalenteIngles;
    arvoreB *esq;
    arvoreB *dir;
};

arvoreB *alocaNo(char palavraPortugues[MAX_STRING],  char
palavraIngles[MAX_STRING]);
lista* lista_alocaNo(char palavraIngles[MAX_STRING]);
char** alocaMatrizChar(int linhas, int tamString);
int insereNo(arvoreB **raiz, char palavraPortugues[MAX_STRING],
char palavraIngles[MAX_STRING]);
void busca(arvoreB** unidades, int qtdUnidades, char* palavra);
arvoreB* buscaArvore(arvoreB* raiz, char* palavra);
void lerArquivo(arvoreB **vetorArv, int qtdUnidades, FILE* input,
double*temposInsercoes);
void imprimeArvore(arvoreB *raiz);
void imprimeLista(lista *raiz);
```

```
void splitPalavraIngles(char* linha, char* palavraIngles);
void splitPalavrasPort(char* linha, char** palavrasPort);
void maiusculo(char s1[], char s2[]);
int findChar(char caractere, char* linha);
void menu();
void liberaNo(arvoreB *no);
void liberaArvore(arvoreB **raiz);
int  ehfolha(arvoreB *raiz);
int sotemUmFilho(arvoreB *raiz, arvoreB **noFilho);
int removeArv(arvoreB **Raiz, char *Elem);
arvoreB * maior(arvoreB *raiz);

int main(int argc, char **argv){
    double tempo_medio_insercao = 0, tempo_busca = 0;
    int output_main = 0, opc, qtdUnidades = 0,unidadeImpressa, i,
removeu = 0;
    char linha[MAX_LINE];
    char palavra[MAX_STRING];
    FILE* input = fopen(argv[1], "r"); // txt de entrada
    if(input){
        while(!feof(input)){
            fgets(linha, MAX_LINE, input);
            qtdUnidades += findChar('%', linha);
        }
        arvoreB **unidades = (arvoreB
**)calloc(qtdUnidades,sizeof(arvoreB*));
        double *tempoInsercao;
        tempoInsercao =
(double*)calloc(qtdUnidades,sizeof(double));
        double *leituraArquivo;
        leituraArquivo = (double*)calloc(1,sizeof(double));

        clock_t inicio = clock();
        lerArquivo(unidades,qtdUnidades,input,tempoInsercao);
        clock_t fim = clock();
        *leituraArquivo = (double)((fim - inicio)* 1000) /
CLOCKS_PER_SEC;
        printf("Tempo para ler o arquivo : %lf Millissegundos\n",
*leituraArquivo);
        do
        {
            menu();
```

```
            setbuf(stdin,NULL);
            scanf("%d", &opc);
            switch (opc)
            {
            case 1:
                printf("\nDigite o numero da unidade: ");
                setbuf(stdin,NULL);
                scanf("%d", &unidadeImpressa);
                if (unidadeImpressa > 0 && unidadeImpressa <=
qtdUnidades)
                {
                    printf("\n");
                    imprimeArvore(unidades[unidadeImpressa - 1]);
                    printf("\n");
                } else {
                    printf("\nUnidade invalida!\n\n");
                }
                break;
            case 2:
                printf("Digite a palavra em portugues: ");
                setbuf(stdin, NULL);
                scanf("%[^\n]s", palavra);
                printf("\n");
                clock_t inicio = clock();
                busca(unidades, qtdUnidades, palavra);
                clock_t fim = clock();
                tempo_busca = ((double)(fim - inicio) * 1000) /
CLOCKS_PER_SEC;
                printf("Tempo de buscas (millissegundos) %lf\n\n",
tempo_busca);
                break;
            case 3:
                printf("Palavra em portugues para ser removida:
");
                setbuf(stdin, NULL);
                scanf("%[^\n]s", palavra);
                i = 0;
                while(!removeu && i < qtdUnidades){
                    removeu = removeArv(&unidades[i], palavra);
                    i++;
                }
                if(removeu){
```

```
                    printf("\nPalavra %s removida\n\n", palavra);
                }
                else
                    printf("\nPalavra %s nao cadastrada\n\n",
palavra);
                removeu = 0;
                break;
            case 4:
                printf("\n");
                for(i = 0; i < qtdUnidades; i++){
                    printf("Tempo para inserir a unidade na arvore
: %lf Milissegundos\n", tempoInsercao[i] * 1000);
                    tempo_medio_insercao += tempoInsercao[i];
                }
                printf("Tempo medio de insercao: %lf
millissegundos\n", tempo_medio_insercao / qtdUnidades);
                printf("\n");
                break;
            case 0:
                printf("\nPrograma finalizado!\n");
                for(i = 0; i < qtdUnidades; i++){
                    liberaArvore(&unidades[i]);
                }
                free(leituraArquivo);
                free(tempoInsercao);
                break;
            default:
                printf("\nOpcao invalida!\n\n");
                break;
            }
        } while (opc != 0);
    } else {
        printf("ERROR: Arquivo de entrada nao encontrado!\n");
        output_main = 1;
    }
    return output_main;
}

// efetua a busca em todas as unidades (arvores)
void busca(arvoreB** unidades, int qtdUnidades, char* palavra){
    int i = 0;
    arvoreB* palavraEncontrada;
```

```
    palavraEncontrada = NULL;

    while (palavraEncontrada == NULL && i < qtdUnidades)
    {
        palavraEncontrada = buscaArvore(unidades[i], palavra);
        i++;
    }

    if (palavraEncontrada != NULL)
    {
        printf("%s -> ",palavra);
        imprimeLista(palavraEncontrada->equivalenteIngles);
        printf("\n");
    }
    else
        printf("\nPalavra nao cadastrada\n\n");
}

// busca um valor na arvore
arvoreB* buscaArvore(arvoreB* raiz, char* palavra){
    arvoreB* encontrou;
    encontrou = NULL;
    char aux_nova_palavra[MAX_STRING],
aux_palavra_raiz[MAX_STRING];

    if (raiz != NULL)
    {
        strcpy(aux_nova_palavra, palavra);
        strcpy(aux_palavra_raiz, raiz->palavraPortugues);
        maiusculo(aux_nova_palavra, aux_nova_palavra);
        maiusculo(aux_palavra_raiz, aux_palavra_raiz);

        if (strcmp(aux_nova_palavra, aux_palavra_raiz) == 0)
        {
            encontrou = raiz;
        }
        else if (strcmp(aux_nova_palavra, aux_palavra_raiz) < 0)
        {
            encontrou = buscaArvore(raiz->esq, palavra);
        }
        else
        {
```

```
            encontrou = buscaArvore(raiz->dir, palavra);
        }
    }
    return encontrou;
}

// imprime um menu na saida padrao
void menu(){
    printf("---------MENU---------\n");
    printf("1 - Imprimir todas as palavras de uma unidade\n");
    printf("2 - Buscar palavra em portugues\n");
    printf("3 - Remover palavra\n");
    printf("4 - Visualizar o tempo de insercoes\n");
    printf("0 - Finalizar programa\n");
    printf("Digite a opcao desejada: ");
}

// itera pelo input separando a palavras ingles e as em portugues
e insere na arvore
void lerArquivo(arvoreB **vetorArv, int qtdUnidades, FILE* input,
double* temposInsercoes){
    char linha[MAX_LINE];
    int indiceUnidades = -1, inicio_arvore = 0, inseriu = 0;
    rewind(input);

    while(!feof(input)){
        fgets(linha, MAX_LINE, input);

        if (findChar('%', linha))
        {
            indiceUnidades++;
            inicio_arvore = 1;

        }else{
            if (findChar(':', linha))
            {
                char palavraIngles[100];
                splitPalavraIngles(linha, palavraIngles);
                int qtd_palavrasPort = findChar(',', linha) + 1;
                char** palavrasPort;
                palavrasPort = alocaMatrizChar(qtd_palavrasPort,
sizeof(linha));
```

```
                splitPalavrasPort(linha, palavrasPort);
                int i;
                clock_t inicio = clock();
                for(i = 0 ; i < qtd_palavrasPort; i++){
                    inseriu +=
insereNo(&vetorArv[indiceUnidades],palavrasPort[i],palavraIngles);
                }
                clock_t fim = clock();
                temposInsercoes[indiceUnidades] += (double)((fim -
inicio)* 1000) / CLOCKS_PER_SEC;
            }
        }
    }
}

// aloca matriz de caracteres
char** alocaMatrizChar(int linhas, int tamString){
    char** nova_matrizChar = (char **)calloc(linhas ,
sizeof(char*));
    int i;
    for (i = 0; i < linhas; i++)
    {
        nova_matrizChar[i] = (char
*)calloc(tamString,sizeof(tamString));
    }
    return nova_matrizChar;
}

// recorta a palavra em portugues da linha do arquivo
void splitPalavrasPort(char* linha, char** palavrasPort){
    int i_linha = 0, i_palavras = 0, i_palavraPort = 0;
    while (linha[i_linha] != ':')
    {
        i_linha++;
    }
    i_linha++;

    while (linha[i_linha] != '\0')
    {
        if (linha[i_linha] == ',')
        {
            i_palavras++;
```

```
            palavrasPort[i_palavras][i_palavraPort] = '\0';
            i_palavraPort = 0;
        } else {
            if(linha[i_linha] != '\n' && linha[i_linha] != ' '){
                palavrasPort[i_palavras][i_palavraPort] =
linha[i_linha];
                i_palavraPort++;
            }
        }
        i_linha++;
    }
}

// recorta a palavra em ingles da linha do arquivo
void splitPalavraIngles(char* linha, char* palavraIngles){
    int i = 0, palavraIngCompleta = 0, i_ingles = 0;
    while (linha[i] != '\0' && !palavraIngCompleta)
    {
        if (linha[i] == ':') {
            palavraIngCompleta = 1;
            palavraIngles[i_ingles] = '\0';
        }

        if (!palavraIngCompleta && linha[i] != ' ')
        {
            palavraIngles[i_ingles] = linha[i];
            i_ingles++;
        }
        i++;
    }
}

// Encontra um caracter em uma string
int findChar(char caractere, char* linha){
    int find = 0, i = 0;
    while (linha[i] != '\0')
    {
        if (linha[i] == caractere)
        {
            find += 1;
        }
        i++;
```

```
    }
    return find;
}

// imprime todos os valores da arvore binaria
void imprimeArvore(arvoreB *raiz){
    if (raiz != NULL)
    {
        imprimeArvore(raiz->esq);

        printf("%s", raiz->palavraPortugues);
        printf(" -> ");
        lista *aux;
        aux = raiz->equivalenteIngles;
        while (aux->prox != NULL)
        {
            printf("%s,", aux->palavraIngles);
            aux = aux->prox;
        }
        printf("%s\n", aux->palavraIngles);
        imprimeArvore(raiz->dir);
    }
}

// imprime uma lista encadeada
void imprimeLista(lista *raiz){
    if (raiz != NULL)
    {
        printf("%s", raiz->palavraIngles);
        if (raiz->prox != NULL)
        {
            printf(", ");
        }else {
            printf(".\n");
        }
        imprimeLista(raiz->prox);
    }
}

// aloca um no da arvore
arvoreB *alocaNo(char palavraPortugues[MAX_STRING], char
palavraIngles[MAX_STRING])
```

```
{
    arvoreB *novo = (arvoreB *)malloc(1 * sizeof(arvoreB));
    novo->esq = NULL;
    novo->dir = NULL;
    strcpy(novo->palavraPortugues, palavraPortugues);

    lista *novo_no;
    novo_no = lista_alocaNo(palavraIngles);

    novo->equivalenteIngles = novo_no;
    return novo;
}

// aloca um no da lista encadeada
lista* lista_alocaNo(char palavraIngles[MAX_STRING]){
  lista* no = (lista*) malloc(sizeof(lista));

  strcpy(no->palavraIngles, palavraIngles);
  no->prox = NULL;

  return no;
}

// insere um no na lista encadeada
int lista_insere(lista** raiz, char palavraIngles[MAX_STRING]){
    int inseriu = 0;

    if (*raiz == NULL){
        *raiz = lista_alocaNo(palavraIngles);
        inseriu = 1;
    } else{
        if (strcmp((*raiz)->palavraIngles, palavraIngles) != 0 )
        {
            inseriu = lista_insere(&((*raiz)->prox),
palavraIngles);
        }else {
            inseriu = 1;
        }
    }
    return inseriu;
}
```

```c
// insere um no na arvore
int insereNo(arvoreB **raiz, char palavraPortugues[MAX_STRING],
char palavraIngles[MAX_STRING])
{
    int inseriu = 0;
    char aux_nova_palavra[MAX_STRING],
aux_palavra_raiz[MAX_STRING];

    if (*raiz == NULL)
    {
        *raiz = alocaNo(palavraPortugues, palavraIngles);
        inseriu = 1;
    }else {
        strcpy(aux_nova_palavra, palavraPortugues);
        strcpy(aux_palavra_raiz, (*raiz)->palavraPortugues);

        maiusculo(aux_nova_palavra, aux_nova_palavra);
        maiusculo(aux_palavra_raiz, aux_palavra_raiz);

        if (strcmp(aux_nova_palavra, aux_palavra_raiz) < 0)
        {
            inseriu = insereNo(&((*raiz)->esq), palavraPortugues,
palavraIngles);
        }
        else if (strcmp(aux_nova_palavra, aux_palavra_raiz) > 0)
        {
            inseriu = insereNo(&((*raiz)->dir), palavraPortugues,
palavraIngles);
        }else if (strcmp(aux_nova_palavra, aux_palavra_raiz) ==
0){
            lista_insere((&(*raiz)->equivalenteIngles),
palavraIngles);
        }
    }
    return inseriu;
}

// transforma um palavra em maiusculo
void maiusculo(char s1[], char s2[]){
    int i = 0;
    while(s1[i] != '\0'){
        s2[i] = toupper(s1[i]);
```

```
        i++;
    }
    s2[i] = '\0';
}

// libera os nos de um arvore binaria
void liberaNo(arvoreB *no)
{
    if (no != NULL)
    {
        liberaNo(no->esq);
        liberaNo(no->dir);
        no = NULL;
        free(no);
    }
}

// liberar o espaco de memoria da arvore binaria
void liberaArvore(arvoreB **raiz)
{
    if (*raiz != NULL)
    {
        liberaNo(*raiz);
        *raiz = NULL;
        free(*raiz);
    }
}

// verifica se um no eh um no folha da arvore
int  ehfolha(arvoreB *raiz){
     int folha = 0;
     if(raiz->esq == NULL && raiz->dir == NULL){
          folha = 1;
     }
     return folha;
}

// verifica se o no possui apenas um filho
int sotemUmFilho(arvoreB *raiz, arvoreB **noFilho){
     int umFilho = 0;
     if(raiz->esq != NULL && raiz->dir == NULL){
          umFilho++;
```

```
            *noFilho = raiz->esq;
      }
      if(raiz->esq == NULL && raiz->dir != NULL){
            umFilho++;
            *noFilho = raiz->dir;
      }
      return umFilho;
}

// remove um no da arvore
int removeArv(arvoreB **Raiz, char *Elem)
{
      int Achou = 0;
    char aux_nova_palavra[MAX_STRING],
aux_palavra_raiz[MAX_STRING];

      if (*Raiz != NULL)
      {
         strcpy(aux_nova_palavra, Elem);
         strcpy(aux_palavra_raiz, (*Raiz)->palavraPortugues);

         maiusculo(aux_nova_palavra, aux_nova_palavra);
         maiusculo(aux_palavra_raiz, aux_palavra_raiz);

         if (strcmp(aux_nova_palavra,aux_palavra_raiz) == 0)
            {
                  arvoreB *Aux;
                  Aux = *Raiz;
                  arvoreB *NoFilho;
                  NoFilho = NULL;
                  arvoreB *NoMaior;
                  NoMaior = NULL;
                  if (ehfolha(*Raiz))
                        *Raiz = NULL;
                  else if (sotemUmFilho(*Raiz, &NoFilho) == 1)
                        *Raiz = NoFilho;
                  else
                  {
                        *Raiz = Aux->esq;
                        NoMaior = maior(*Raiz);
                        NoMaior->dir = Aux->dir;
                  }
```

```
                free(Aux);
             Achou = 1;
            }
            else if (strcmp(aux_nova_palavra,aux_palavra_raiz) < 0)
                Achou = removeArv(&((*Raiz)->esq), Elem);
            else
                Achou = removeArv(&((*Raiz)->dir), Elem);
      }

      return (Achou);
}

// busca o maior valor de uma sub-arvore
arvoreB * maior(arvoreB *raiz){
      arvoreB * aux;
      aux = raiz;
      while(aux->dir != NULL){
            aux = aux->dir;
      }
      return aux;
}
```

Questão 04

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define MAX_STRING 30
#define MAX_LINE 300

typedef struct Lista lista;
struct Lista{
    char palavraIngles[MAX_STRING];
    lista *prox;
};

typedef struct ArvoreB arvoreB;
struct ArvoreB
{
    char palavraPortugues[MAX_STRING];
```

```
    lista* equivalenteIngles;
    arvoreB *esq;
    arvoreB *dir;
    int altura;
};

arvoreB *alocaNo(char palavraPortugues[MAX_STRING],  char
palavraIngles[MAX_STRING]);
lista* lista_alocaNo(char palavraIngles[MAX_STRING]);
char** alocaMatrizChar(int linhas, int tamString);
int insereAVL(arvoreB **raiz, char palavraPortugues[MAX_STRING],
char palavraIngles[MAX_STRING]);
void busca(arvoreB** unidades, int qtdUnidades, char* palavra);
arvoreB* buscaArvore(arvoreB* raiz, char* palavra);
void lerArquivo(arvoreB **vetorArv, int qtdUnidades, FILE* input,
double*temposInsercoes);
void imprimeArvore(arvoreB *raiz);
void imprimeLista(lista *raiz);
void splitPalavraIngles(char* linha, char* palavraIngles);
void splitPalavrasPort(char* linha, char** palavrasPort);
void maiusculo(char s1[], char s2[]);
int findChar(char caractere, char* linha);
void menu();
void liberaNo(arvoreB *no);
void liberaArvore(arvoreB **raiz);
int  ehfolha(arvoreB *raiz);
int sotemUmFilho(arvoreB *raiz, arvoreB **noFilho);
int removeAVL(arvoreB **Raiz, char *Elem);
arvoreB * maior(arvoreB *raiz);
int fatorBalanceamento(arvoreB *raiz);
void rotacaoDireita(arvoreB **raiz);
void rotacaoEsquerda(arvoreB **raiz);
int alturaNo(arvoreB *raiz);
int calculaAltura(arvoreB *raiz);
int maiorAltura(int a, int b);
void calculaAltSubArvore(arvoreB *raiz);

int main(int argc, char **argv){
    int output_main = 0, opc, qtdUnidades = 0,unidadeImpressa, i,
removeu = 0;
    char linha[MAX_LINE];
    char palavra[MAX_STRING];
```

```
    FILE* input = fopen(argv[1], "r"); // txt de entrada
    if(input){
        while(!feof(input)){
            fgets(linha, MAX_LINE, input);
            qtdUnidades += findChar('%', linha);
        }
        arvoreB **unidades = (arvoreB
**)calloc(qtdUnidades,sizeof(arvoreB*));
        double *tempoInsercao;
        tempoInsercao =
(double*)calloc(qtdUnidades,sizeof(double));
        double *leituraArquivo;
        leituraArquivo = (double*)calloc(1,sizeof(double));

        clock_t inicio = clock();
        lerArquivo(unidades,qtdUnidades,input,tempoInsercao);
        clock_t fim = clock();
        *leituraArquivo = (double)(fim - inicio) / CLOCKS_PER_SEC;
        printf("Tempo para ler o arquivo : %lf segundos\n",
*leituraArquivo);
        do
        {
            menu();
            setbuf(stdin,NULL);
            scanf("%d", &opc);
            switch (opc)
            {
            case 1:
                printf("\nDigite o numero da unidade: ");
                setbuf(stdin,NULL);
                scanf("%d", &unidadeImpressa);
                if (unidadeImpressa > 0 && unidadeImpressa <=
qtdUnidades)
                {
                    printf("\n");
                    imprimeArvore(unidades[unidadeImpressa - 1]);
                    printf("\n");
                } else {
                    printf("\nUnidade invalida!\n\n");
                }
                break;
            case 2:
```

```
                printf("Digite a palavra em portugues: ");
                setbuf(stdin, NULL);
                scanf("%[^\n]s", palavra);
                busca(unidades, qtdUnidades, palavra);
                break;
            case 3:
                printf("Palavra em portugues para ser removida:
");
                setbuf(stdin, NULL);
                scanf("%[^\n]s", palavra);
                i = 0;
                while(!removeu && i < qtdUnidades){
                    removeu = removeAVL(&unidades[i], palavra);
                    i++;
                }
                if(removeu){
                    printf("\nPalavra %s removida\n\n", palavra);
                }
                else
                    printf("\nPalavra %s nao cadastrada\n\n",
palavra);
                removeu = 0;
                break;
            case 4:
                printf("\n");
                for(i = 0; i < qtdUnidades; i++){
                    printf("Tempo para inserir a unidade na arvore
: %lf segundos\n", tempoInsercao[i]);
                }
                printf("\n");
                break;
            case 0:
                printf("\nPrograma finalizado!\n");
                for(i = 0; i < qtdUnidades; i++){
                    liberaArvore(&unidades[i]);
                }
                free(leituraArquivo);
                free(tempoInsercao);
                break;
            default:
                printf("\nOpcao invalida!\n\n");
                break;
```

```
            }
        } while (opc != 0);
    } else {
        printf("ERROR: Arquivo de entrada nao encontrado!\n");
        output_main = 1;
    }
    return output_main;
}

// efetua a busca em todas as unidades (arvores)
void busca(arvoreB** unidades, int qtdUnidades, char* palavra){
    int i = 0;
    arvoreB* palavraEncontrada;
    palavraEncontrada = NULL;
    int encontrou = 0;

    for (i = 0; i < qtdUnidades; i++)
    {
        palavraEncontrada = buscaArvore(unidades[i], palavra);
        if (palavraEncontrada != NULL)
        {
            encontrou = 1;
            printf("\n%s -> ",palavra);
            imprimeLista(palavraEncontrada->equivalenteIngles);
            palavraEncontrada = NULL;
        }
    }
    if (!encontrou)
    {
        printf("\nPalavra nao cadastrada\n");
    }
    printf("\n");
}

// busca um valor na arvore
arvoreB* buscaArvore(arvoreB* raiz, char* palavra){
    arvoreB* encontrou;
    encontrou = NULL;
    char aux_nova_palavra[MAX_STRING],
aux_palavra_raiz[MAX_STRING];

    if (raiz != NULL)
```

```
    {
        strcpy(aux_nova_palavra, palavra);
        strcpy(aux_palavra_raiz, raiz->palavraPortugues);
        maiusculo(aux_nova_palavra, aux_nova_palavra);
        maiusculo(aux_palavra_raiz, aux_palavra_raiz);

        if (strcmp(aux_nova_palavra, aux_palavra_raiz) == 0)
        {
            encontrou = raiz;
        }
        else if (strcmp(aux_nova_palavra, aux_palavra_raiz) < 0)
        {
            encontrou = buscaArvore(raiz->esq, palavra);
        }
        else
        {
            encontrou = buscaArvore(raiz->dir, palavra);
        }
    }
    return encontrou;
}

// imprime um menu na saida padrao
void menu(){
    printf("---------MENU---------\n");
    printf("1 - Imprimir todas as palavras de uma unidade\n");
    printf("2 - Buscar palavra em portugues\n");
    printf("3 - Remover palavra\n");
    printf("4 - Visualizar o tempo de insercoes\n");
    printf("0 - Finalizar programa\n");
    printf("Digite a opcao desejada: ");
}

// itera pelo input separando a palavras ingles e as em portugues
e insere na arvore
void lerArquivo(arvoreB **vetorArv, int qtdUnidades, FILE* input,
double* temposInsercoes){
    char linha[MAX_LINE];
    int indiceUnidades = -1, inicio_arvore = 0;
    rewind(input);

    while(!feof(input)){
```

```
        fgets(linha, MAX_LINE, input);
        if (findChar('%', linha))
        {
            indiceUnidades++;
            inicio_arvore = 1;
        }else{
            if (findChar(':', linha))
            {
                char palavraIngles[100];
                splitPalavraIngles(linha, palavraIngles);
                int qtd_palavrasPort = findChar(',', linha) + 1;
                char** palavrasPort;
                palavrasPort = alocaMatrizChar(qtd_palavrasPort,
sizeof(linha));
                splitPalavrasPort(linha, palavrasPort);
                int i,inseriu;
                clock_t inicio = clock();
                for(i = 0 ; i < qtd_palavrasPort; i++){
                    inseriu =
insereAVL(&vetorArv[indiceUnidades],palavrasPort[i],palavraIngles)
;
                }
                clock_t fim = clock();
                temposInsercoes[indiceUnidades] += (double)(fim -
inicio) / CLOCKS_PER_SEC;
            }
        }
    }
}

// aloca matriz de caracteres
char** alocaMatrizChar(int linhas, int tamString){
    char** nova_matrizChar = (char **)calloc(linhas ,
sizeof(char*));
    int i;
    for (i = 0; i < linhas; i++)
    {
        nova_matrizChar[i] = (char
*)calloc(tamString,sizeof(tamString));
    }
    return nova_matrizChar;
}
```

```
// recorta a palavra em portugues da linha do arquivo
void splitPalavrasPort(char* linha, char** palavrasPort){
    int i_linha = 0, i_palavras = 0, i_palavraPort = 0;
    while (linha[i_linha] != ':')
    {
        i_linha++;
    }
    i_linha++;

    while (linha[i_linha] != '\0')
    {
        if (linha[i_linha] == ',')
        {
            i_palavras++;
            palavrasPort[i_palavras][i_palavraPort] = '\0';
            i_palavraPort = 0;
        } else {
            if(linha[i_linha] != '\n' && linha[i_linha] != ' '){
                palavrasPort[i_palavras][i_palavraPort] =
linha[i_linha];
                i_palavraPort++;
            }
        }
        i_linha++;
    }
}

// recorta a palavra em ingles da linha do arquivo
void splitPalavraIngles(char* linha, char* palavraIngles){
    int i = 0, palavraIngCompleta = 0, i_ingles = 0;
    while (linha[i] != '\0' && !palavraIngCompleta)
    {
        if (linha[i] == ':') {
            palavraIngCompleta = 1;
            palavraIngles[i_ingles] = '\0';
        }

        if (!palavraIngCompleta && linha[i] != ' ')
        {
            palavraIngles[i_ingles] = linha[i];
            i_ingles++;
```

```
        }
        i++;
    }
}

// Encontra um caracter em uma string
int findChar(char caractere, char* linha){
    int find = 0, i = 0;
    while (linha[i] != '\0')
    {
        if (linha[i] == caractere)
        {
            find += 1;
        }
        i++;
    }
    return find;
}

// imprime todos os valores da arvore binaria
void imprimeArvore(arvoreB *raiz){
    if (raiz != NULL)
    {
        imprimeArvore(raiz->esq);
        printf("%s", raiz->palavraPortugues);
        printf(" -> ");

        lista *aux;
        aux = raiz->equivalenteIngles;
        while (aux->prox != NULL)
        {
            printf("%s,", aux->palavraIngles);
            aux = aux->prox;
        }
        printf("%s\n", aux->palavraIngles);
        imprimeArvore(raiz->dir);
    }
}

// imprime uma lista encadeada
void imprimeLista(lista *raiz){
    if (raiz != NULL)
```

```
    {
        printf("%s", raiz->palavraIngles);
        if (raiz->prox != NULL)
        {
            printf(", ");
        }else {
            printf(".\n");
        }
        imprimeLista(raiz->prox);
    }
}

// aloca um no da arvore
arvoreB *alocaNo(char palavraPortugues[MAX_STRING], char
palavraIngles[MAX_STRING])
{
    arvoreB *novo = (arvoreB *)malloc(1 * sizeof(arvoreB));
    novo->esq = NULL;
    novo->dir = NULL;
    strcpy(novo->palavraPortugues, palavraPortugues);

    lista *novo_no;
    novo_no = lista_alocaNo(palavraIngles);

    novo->equivalenteIngles = novo_no;
    novo->altura = 0;
    return novo;
}

// aloca um no da lista encadeada
lista* lista_alocaNo(char palavraIngles[MAX_STRING]){
  lista* no = (lista*) malloc(sizeof(lista));

  strcpy(no->palavraIngles, palavraIngles);
  no->prox = NULL;

  return no;
}

// insere um no na lista encadeada
int lista_insere(lista** raiz, char palavraIngles[MAX_STRING]){
    int inseriu = 0;
```

```
    if (*raiz == NULL){
        *raiz = lista_alocaNo(palavraIngles);
        inseriu = 1;
    } else{
        if (strcmp((*raiz)->palavraIngles, palavraIngles) != 0 )
        {
            inseriu = lista_insere(&((*raiz)->prox),
palavraIngles);
        }else {
            inseriu = 1;
        }
    }
    return inseriu;
}

// insere um no na arvore AVL
int insereAVL(arvoreB **raiz, char palavraPortugues[MAX_STRING],
char palavraIngles[MAX_STRING])
{
    int inseriu = 0;
    char aux_nova_palavra[MAX_STRING],
aux_palavra_raiz[MAX_STRING];

    if (*raiz == NULL)
    {
        *raiz = alocaNo(palavraPortugues, palavraIngles);
        inseriu = 1;
    }else {
        strcpy(aux_nova_palavra, palavraPortugues);
        strcpy(aux_palavra_raiz, (*raiz)->palavraPortugues);

        maiusculo(aux_nova_palavra, aux_nova_palavra);
        maiusculo(aux_palavra_raiz, aux_palavra_raiz);

        if (strcmp(aux_nova_palavra, aux_palavra_raiz) < 0)
        {
            inseriu = insereAVL(&((*raiz)->esq), palavraPortugues,
palavraIngles);
        }
        else if (strcmp(aux_nova_palavra, aux_palavra_raiz) > 0)
        {
```

```
            inseriu = insereAVL(&((*raiz)->dir), palavraPortugues,
palavraIngles);
        }else if (strcmp(aux_nova_palavra, aux_palavra_raiz) ==
0){
            lista_insere((&(*raiz)->equivalenteIngles),
palavraIngles);
        }
    }
    if (inseriu == 1)
    {
        int fb;
        fb = fatorBalanceamento(*raiz);
        if (fb == 2)
        {
            if (fatorBalanceamento((*raiz)->esq) == 1)
            {
                rotacaoDireita(raiz); // balanceamento
                (*raiz)->dir->altura =
calculaAltura((*raiz)->dir);
            }
            else
            {
                rotacaoEsquerda(&((*raiz)->esq)); // ajustar galho
da esq
                rotacaoDireita(raiz);             // balanceamento
                (*raiz)->dir->altura =
calculaAltura((*raiz)->dir);
                (*raiz)->esq->altura =
calculaAltura((*raiz)->esq);
                //calculaAltSubArvore(*raiz);
            }
        }
        else if (fb == -2)
        {
            if (fatorBalanceamento((*raiz)->dir) == -1)
            {
                rotacaoEsquerda(raiz); // balanceamento
                //calculaAltSubArvore(*raiz);
                (*raiz)->esq->altura =
calculaAltura((*raiz)->esq);
            }
            else
```

```
            {
                rotacaoDireita(&((*raiz)->dir)); // ajustar galho
da dir
                rotacaoEsquerda(raiz);           // balanceamento
                //calculaAltSubArvore(*raiz);
                (*raiz)->esq->altura =
calculaAltura((*raiz)->esq);
                (*raiz)->dir->altura =
calculaAltura((*raiz)->dir);
            }
        }
        (*raiz)->altura = calculaAltura(*raiz);
    }
    return inseriu;
}

// transforma um palavra em maiusculo
void maiusculo(char s1[], char s2[]){
    int i = 0;
    while(s1[i] != '\0'){
        s2[i] = toupper(s1[i]);
        i++;
    }
    s2[i] = '\0';
}

// libera os nos de um arvore binaria
void liberaNo(arvoreB *no)
{
    if (no != NULL)
    {
        liberaNo(no->esq);
        liberaNo(no->dir);
        no = NULL;
        free(no);
    }
}

// liberar o espaco de memoria da arvore binaria
void liberaArvore(arvoreB **raiz)
{
    if (*raiz != NULL)
```

```
    {
        liberaNo(*raiz);
        *raiz = NULL;
        free(*raiz);
    }
}

// verifica se um no eh um no folha da arvore
int  ehfolha(arvoreB *raiz){
     int folha = 0;
     if(raiz->esq == NULL && raiz->dir == NULL){
          folha = 1;
     }
     return folha;
}

// verifica se o no possui apenas um filho
int sotemUmFilho(arvoreB *raiz, arvoreB **noFilho){
     int umFilho = 0;
     if(raiz->esq != NULL && raiz->dir == NULL){
          umFilho++;
          *noFilho = raiz->esq;
     }
     if(raiz->esq == NULL && raiz->dir != NULL){
          umFilho++;
          *noFilho = raiz->dir;
     }
     return umFilho;
}

// remove um no da arvore AVL
int removeAVL(arvoreB **Raiz, char *Elem)
{
     int Achou = 0;
    char aux_nova_palavra[MAX_STRING],
aux_palavra_raiz[MAX_STRING];

     if (*Raiz != NULL)
     {
        strcpy(aux_nova_palavra, Elem);
        strcpy(aux_palavra_raiz, (*Raiz)->palavraPortugues);
```

```
        maiusculo(aux_nova_palavra, aux_nova_palavra);
        maiusculo(aux_palavra_raiz, aux_palavra_raiz);

        if (strcmp(aux_nova_palavra,aux_palavra_raiz) == 0)
           {
                arvoreB *Aux;
                Aux = *Raiz;
                arvoreB *NoFilho;
                NoFilho = NULL;
                arvoreB *NoMaior;
                NoMaior = NULL;
                if (ehfolha(*Raiz))
                     *Raiz = NULL;
                else if (sotemUmFilho(*Raiz, &NoFilho) == 1)
                     *Raiz = NoFilho;
                else
                {
                     *Raiz = Aux->esq;
                     NoMaior = maior(*Raiz);
                     NoMaior->dir = Aux->dir;
                }
                free(Aux);
            Achou = 1;
           }
           else if (strcmp(aux_nova_palavra,aux_palavra_raiz) < 0)
                Achou = removeAVL(&((*Raiz)->esq), Elem);
           else
                Achou = removeAVL(&((*Raiz)->dir), Elem);
      }
    if (Achou == 1 && *Raiz != NULL)
        {
            int fb;
            fb = fatorBalanceamento(*Raiz);

            if (fb == 2)
            {
                if (fatorBalanceamento((*Raiz)->esq) == 1)
                {
                    rotacaoDireita(Raiz); // balanceamento
                    //calculaAltSubArvore(*Raiz);
                    (*Raiz)->dir->altura =
calculaAltura((*Raiz)->dir);
```

```
                }
                else
                {
                    rotacaoEsquerda(&((*Raiz)->esq)); // ajustar
galho da esq
                    rotacaoDireita(Raiz);             //
balanceamento
                    //calculaAltSubArvore(*Raiz);
                    (*Raiz)->dir->altura =
calculaAltura((*Raiz)->dir);
                    (*Raiz)->esq->altura =
calculaAltura((*Raiz)->esq);
                }
            }
            else if (fb == -2)
            {
                if (fatorBalanceamento((*Raiz)->dir) == -1)
                {
                    rotacaoEsquerda(Raiz); // balanceamento
                    //calculaAltSubArvore(*Raiz);
                    (*Raiz)->esq->altura =
calculaAltura((*Raiz)->esq);
                }
                else
                {
                    rotacaoDireita(&((*Raiz)->dir)); // ajustar
galho da dir
                    rotacaoEsquerda(Raiz);           //
balanceamento
                    //calculaAltSubArvore(*Raiz);
                    (*Raiz)->esq->altura =
calculaAltura((*Raiz)->esq);
                    (*Raiz)->dir->altura =
calculaAltura((*Raiz)->dir);
                }
            }
            (*Raiz)->altura = calculaAltura(*Raiz);
        }

     return (Achou);
}
```

```c
// calcula o fator de balanceamento de um no
int fatorBalanceamento(arvoreB *raiz)
{
    int fb = 0;
    if (raiz != NULL)
    {
        fb = (alturaNo(raiz->esq) - alturaNo(raiz->dir));
    }
    return fb;
}

// rotaciona um no para a direita da arvore
void rotacaoDireita(arvoreB **raiz)
{
    arvoreB *aux;
    aux = (*raiz)->esq;
    (*raiz)->esq = aux->dir;
    aux->dir = *raiz;
    if (*raiz != NULL){
        (*raiz)->altura = calculaAltura(*raiz);
    }
    if (aux != NULL){
        aux->altura = calculaAltura(aux);
    }
    *raiz = aux;
}

// rotaciona um no para a esquerda da arvore
void rotacaoEsquerda(arvoreB **raiz)
{
    arvoreB *aux;
    aux = (*raiz)->dir;
    (*raiz)->dir = aux->esq;
    aux->esq = *raiz;
    if (*raiz != NULL){
        (*raiz)->altura = calculaAltura(*raiz);
    }
    if (aux != NULL){
        aux->altura = calculaAltura(aux);
    }
    *raiz = aux;
}
```

```c
// retorna a altura atual de um no
int alturaNo(arvoreB *raiz)
{
    int altura = -1;
    if (raiz != NULL)
        altura = raiz->altura;
    return altura;
}

// calcula a altura de um no
int calculaAltura(arvoreB *raiz)
{
    int alt_esq = alturaNo(raiz->esq);
    int alt_dir = alturaNo(raiz->dir);
    int altura_no = maiorAltura(alt_esq, alt_dir) + 1;
    return altura_no;
}

// encontra a maior altura dentre duas alturas enviadas por
parametro
int maiorAltura(int a, int b)
{
    return (a > b) ? a : b;
}

// remove um no da avore
void calculaAltSubArvore(arvoreB *raiz)
{
    if (raiz != NULL)
    {
        calculaAltSubArvore(raiz->esq);
        calculaAltSubArvore(raiz->dir);
        raiz->altura = calculaAltura(raiz);
    }
}

// busca o maior valor de uma sub-arvore
arvoreB * maior(arvoreB *raiz){
     arvoreB * aux;
     aux = raiz;
     while(aux->dir != NULL){
```

```
            aux = aux->dir;
        }
        return aux;
    }
```