

UNIVERSIDADE FEDERAL DO AMAZONAS – UFAM

INSTITUTO DE COMPUTAÇÃO

FT05 – ENGENHARIA DA COMPUTAÇÃO

Algoritmos e estruturas de dados II

IEC013

Manaus-AM

2020

21951799 – Vinícius Patrício Medeiros da Silva

Árvores patrícia

Relatório de algoritmo apresentado junto ao curso de Engenharia da Computação como requisito à obtenção de nota parcial para o período especial 2020.

Professor: Edson Nascimento

Manaus-AM

2020

**1-INTRODUÇÃO**

O trabalho consiste em implementar um programa que leia um texto qualquer, no caso um arquivo no formato .txt, e imprima na ordem alfabética, todas as palavras com 3 ou mais caracteres e a(s) linha(s) na(s) qual(is) elas aparecem no texto.

A leitura não deverá ser feita de qualquer jeito, é necessário que seja desprezado os espaços em branco como também sinais de pontuação. Além disso, a leitura deve converter todas as letras maiúsculas para minúsculas.

Para propor um programa com essa finalidade, é necessário a implementação a estrutura árvore patrícia. A cada palavra lida, o algoritmo deve pesquisar à árvore para verificar se a palavra já está presente. Caso já esteja presente, é necessário adicionar o numero da linha à lista de linhas dessa palavra. Se não estiver na árvore patrícia, é necessário criar um novo nó na árvore e iniciar a lista de linhas junto a palavra lida.

A árvore patrícia é uma representação mais compacta de uma trie onde os nós que teria um único filho são agrupados nos seus antecessores, tornando-a mais indicada a esse tipo de problema, sua estrutura é bastante dinâmica.

Essa estrutura de dados foi criada por Donald Morrison, nada mais é quem um algoritmo com a finalidade de buscas em árvores com conteúdos sendo palavras e nós binários, armazenando conteúdo apenas nos nós folhas.

**2-IMPLEMENTAÇÃO**

**2.1 DEFINIÇÕES E VARIÁVEIS GLOBAIS**

* **Definições**

#define true 1 // verdadeiro

#define false 0 // falso

Essas definições foram dadas apenas para ficar mais claro algumas condições que foram usadas no código, ao invés de comparar com 0 e 1, fica bem mais claro por exemplo igualarmos uma sentença a true que igualar 1, ou seja, foram utilizadas para ficar mais fácil o entendimento do código.

* **Variáveis globais**

int linha = 1; // linha vai ser incrementada assim que estivermos no final da linha lida

A variável linha irá indicar que linha está sendo efeituada a leitura, a cada quebra de linha do arquivo a variável linha será incrementada.

**2.2 ESTRUTURAS**

* **Lista**

**Lista e carga:**

typedef struct carga TELE;

struct carga{

    int carga;

    struct carga \*prox;

};

typedef struct lista TLista;

struct lista{

    int tamanho;

    TELE \*inicio;

    TELE \*fim;

};

Foi definido a estrutura lista com a finalidade de armazenar o número da linha onde a palavra aparece, como isso pode variar de texto para texto, foi necessário trabalhar exatamente com lista por tratar-se de uma estrutura dinâmica. No caso do trabalho, não foi necessário trabalhar com listas neutras, uma vez que precisamos apenas armazenar um inteiro. O struct carga vai ser um elemento da lista, ele vai armazenar um inteiro carga (no caso vamos colocar a linha) e o endereço do próximo elemento da lista.

**Criar Lista:**

TLista \*criarLista(){

    TLista \*lista = malloc(sizeof(TLista));

    lista->tamanho = 0;

    lista->inicio = NULL;

    lista->fim = NULL;

    return lista;

}

A função criarLista irá alocar na memória para estrutura do tipo struct lista.

**Criar elemento da lista:**

TELE \*criarElemento(int elemento){

    TELE \*carga\_nova = malloc(sizeof(TELE));

    carga\_nova->carga = elemento;

    carga\_nova->prox = NULL;

    return carga\_nova;

}

criarElemento irá criar o elemento da lista dado um número inteiro, no caso do trabalho, essa entrada vai ser a linha onde a palavra se encontra.

**Inserir na lista um novo elemento:**

void inserirLista(TLista \*lista, TELE \*carga){

    if(lista->tamanho == 0){

        lista->inicio = carga;

        lista->fim =  carga;

    }else{

        lista->fim->prox = carga;

        lista->fim = carga;

    }

    lista->tamanho ++;

}

Dado um elemento do tipo TELE (struct carga) e uma lista, essa função irá inserir esse elemento sempre no fim da lista.

**Verifica se a lista é vazia:**

int listaVazia(TLista \*lista){

    return lista->tamanho == 0 ? 1 : 0;

}

Essa função irá receber uma lista e retornará 1 caso ela não tenha elementos e retornará 0 caso ela tenha.

**Imprimir lista:**

void imprimirLista(TLista \*lista){

    TELE \*caminhador = lista->inicio;

    while(caminhador != NULL){

        printf("%d ",caminhador->carga);

        caminhador = caminhador->prox;

    }

}

Essa função irá imprimir a carga dos elementos da lista. Para o trabalho, ela será usada para imprimir as linhas onde a palavra se encontra no texto.

Obs: A estrutura de lista foi implementada na lista.c e foi criado a lista.h, esta vai ser usada nó código principal (patricia.c).

* **Árvore patrícia**

**Estrutura da Árvore patrícia:**

typedef struct no\_patricia PNO;

struct no\_patricia{

    int pos\_caracter;  // posição do caractere a ser comparado

    TLista \*linhas;    // uma lista que guarda em que linhas encontra-se as palavras

    char palavra[20];  // palavra que o nó/raiz irá armazenar

    PNO \*filho\_esq;

    PNO \*filho\_dir;

};

A variável pos\_caracter irá armazenar a posição onde deverá ser comparo as palavras na hora da inserção, no caso dessa implementação optou-se de utilizar um inteiro que diz a posição da comparação ao invés de usar um char que armazena o caractere. Linhas irá armazenar a linha onde a palavra ocorreu no texto, o vetor palavra irá armazenar a palavra daquele nó específico, e por se tratar de uma árvore binária, a estrutura de cada nó terá dois filhos.

**Criar raiz:**

PNO \*criarRaiz(int pos\_char, char palavra[20], int tamanho){

    PNO \*nova\_raiz = malloc(sizeof(PNO));

    nova\_raiz->pos\_caracter = pos\_char;

    nova\_raiz->linhas = criarLista();

    strncpy(nova\_raiz->palavra, palavra,tamanho);

    nova\_raiz->filho\_esq = NULL;

    nova\_raiz->filho\_dir = NULL;

    return nova\_raiz;

}

A função irá alocar na memória para um ponteiro de uma estrutura do tipo PNO, ela recebe a posição de onde deve ocorrer a comparação, recebe a palavra que deve ser inserida, e o tamanho da palavra. Por ser um novo raiz/nó seus filhos recebem NULL.

**2.3 FUNÇÕES AUXIARES**

**Tamanho da menor string:**

int tamanhoMenorString(char palavra1[20], char palavra2[20]){

    int menor = strlen(palavra1);

    if(menor > strlen(palavra2)){

        menor = strlen(palavra2);

    }

    return menor;

}

A função retorna o tamanho da menor string dada duas strings.

**Último caractere em comum:**

int ultimoCharComum(char palavra1[20], char palavra2[20]){

    int menor = tamanhoMenorString(palavra1,palavra2); // saber até onde i (contador) ser icrementado

    int i = 0; // contador de caracteres iguais

    while(palavra1[i] == palavra2[i] && (i < menor)){

        i++;

    }

    if( i == menor){ // uma das palavras é sub-palavra da outra

        return -1;

    }

    i-= 1;

    if(i < 0){ // não tem nenhum char em comum

        i = 0;

    }

    return i;

}

Retorna a posição do último caractere em comum entre duas strings, caso uma seja uma sub-palavra (sub-string) da outra, deve retornar -1, caso não tenha nada em comum, retornará 0, em nenhum desses casos, irá retornar à posição do último caractere em comum.

**Primeiro caractere distinto:**

int primeiroCharDistinto(char palavra1[20], char palavra2[20]){

    int menor = tamanhoMenorString(palavra1,palavra2); // menor foi criado para saber qual vai ser o limite do icremento do contador;

    int contador = 0; // contador de caracteres iguais

    // o contador irá ser icrementado enquanto os caracteres das strings forem iguais e ele ser menor que o tamanho da menor string

    while(palavra1[contador] == palavra2[contador] && contador < menor){

        contador ++;

    }

    return contador;

}

A função irá retorna o índice do primeiro caractere diferente entre dias strings.

**Menor string a partir de um caractere:**

char \*menorStringChar(char palavra1[20], char palavra2[20]){

    int aux = primeiroCharDistinto(palavra1,palavra2);

    return (palavra1[aux] < palavra2[aux])? palavra1 : palavra2;

}

A função irá informar a menor string entre duas strings dada a posição onde ocorre a diferença de caractere.

**Maior string a partir de um caractere:**

char \*maiorStringChar(char palavra1[20], char palavra2[20]){

    int aux = primeiroCharDistinto(palavra1,palavra2);

    return (palavra1[aux]>palavra2[aux])? palavra1 : palavra2;

}

A função irá informar a maior string entre duas strings dada a posição onde ocorre a diferença de caractere.

**Filho menor:**

PNO \*filhoMenor(PNO \*filho\_esq, PNO \*filho\_dir){

    char\* aux = menorStringChar(filho\_esq->palavra,filho\_dir->palavra);

    if(strcmp(filho\_esq->palavra,aux)==0){

        return filho\_esq;

    }else{

        return filho\_dir;

    }

}

A função irá retorna o nó com a menor string.

**Filho maior:**

PNO \*filhoMaior(PNO \*filho\_esq, PNO \*filho\_dir){

    char\* aux = maiorStringChar(filho\_dir->palavra,filho\_esq->palavra);

    return strcmp(filho\_esq->palavra,aux) == 0 ? filho\_esq : filho\_dir;

}

Será dado o nó com a maior string.

**Folha:**

int folha(PNO \*raiz){

    if( (raiz->filho\_dir==NULL) &&( raiz->filho\_esq==NULL)){ // caso  o nó não tenha filhos, então ele é uma folha

        return true;

    }else{

        return false;

    }

}

Folha irá retornar 1 caso a raiz passado para função for uma folha e 0 caso não seja.

**Imprimir Arvore:**

void imprimirArvore(PNO \*raiz){

    setlocale(LC\_ALL, "Portuguese\_Brazil");

    if(raiz != NULL){

        imprimirArvore(raiz->filho\_esq);

        if(strlen(raiz->palavra)>=3 && listaVazia(raiz->linhas) == 0){

            printf("%s ",raiz->palavra);

            imprimirLista(raiz->linhas);

            printf("\n");

        }

       imprimirArvore(raiz->filho\_dir);

    }

}

ImprimirArvore imprimi a árvore passado na chamada da função.

**Número de pontuações:**

int numeroPontuacao(char palavra[20]){

    int tamanho\_palavra = strlen(palavra);

    int contador = 0;

    for(int i = 0; i<tamanho\_palavra; i++){

        if (palavra[i] == '.' || palavra[i] == ',' || palavra[i] == '!' || palavra[i] == '?' || palavra[i] == '"'){

            contador++;

        }

    }

    return contador;

}

A função irá informar quantas pontuações a string tem.

**Caractere pontuação:**

int charPontuacao(char caracter){

    if(caracter == '.' || caracter == ',' || caracter == '!' || caracter =='?' || caracter == '"'){

        return true;

    }

    return false;

}

A função verifica se a caractere passado é uma pontuação.

**Tirar pontuação:**

char \*tirarPontuacao(char palavra[20]){

    int tamanho\_palavra = strlen(palavra);

    int i = 0;

    int num = 1;

    int numero\_pontuacao = numeroPontuacao(palavra);

    if(charPontuacao(palavra[0]) == true){  // verifica se o inicio tem aspas

        for(int j = 0;  j < tamanho\_palavra-1;j++){

            palavra[j] = palavra[j+1];

        }

        palavra[tamanho\_palavra-1] = '\0';

        i++;

        while (i < numero\_pontuacao) // irá fazer as remoções das pontuações que são armazenadas no final da string caso tenha

        {

            if(charPontuacao(palavra[strlen(palavra)-i])==true){

                palavra[strlen(palavra)-i] = '\0';

            }

            i++;

        }

    }else{ // caso não tenha pontuação no inicio, irá remover caso tenha no final

        while(i < numero\_pontuacao){

            if(charPontuacao(palavra[strlen(palavra)-i-1]) == true){

                palavra[strlen(palavra)-i-1] = '\0';

            }

            i++;

        }

    }

    return palavra;

}

tirarPontuação retira as pontuações de uma string, a primeira condição verifica se possui uma pontuação no inicio da string, por exemplo uma string dada por “carro”, ele vai verificar que essa string tem duas pontuações, vai verificar que possui uma no início, ela vai ser retirada, caso tenha mais, ele vai sair retirando as pontuações do final, caso só tenha no final, será retirada as pontuações do final.

**2.4 FUNÇÕES PRINCIPAIS**

**Lado inserção:**

int ladoInsercao(PNO \*raiz, char palavra\_nova[20]){

    int aux = primeiroCharDistinto(raiz->palavra,palavra\_nova);

    int menor = tamanhoMenorString(raiz->palavra,palavra\_nova);

    if(aux>menor-1){  // caso de uma string ser sub-string da outra

        int menorPalavra1 = primeiroCharDistinto(palavra\_nova,raiz->filho\_esq->palavra);

        int menorPalavra2 = primeiroCharDistinto(palavra\_nova,raiz->filho\_dir->palavra);

        if(menorPalavra2>menorPalavra1){ // a palavra guardada no filho direto tem mais caracteres em comum com a nova palavra a ser inserida

            return 2;

        }else if (menorPalavra2 == menorPalavra1){ // caso tenha a mesma quantidade de caracteres em comum com as palavras do os dois filhos do nó

            // irá verificar onde deve ser inserido a partir da posição onde ocorre a diferença de de caracter

            if(raiz->filho\_dir->palavra[menorPalavra2] > palavra\_nova[menorPalavra2]){

                return 1;

            }else{

                return 2;

            }

        }else{ // a palavra guardada no filho esq tem mais caracteres em comum com a nova palavra a ser inserida

            return 1;

        }

    } // caso não seja uma sub-string

    if(palavra\_nova[aux] > raiz->palavra[aux]){

        return 2;

    }else{

        return 1;

    }

}

A função irá retornar onde deve ser inserido a nova palavra dada uma raiz, para isso ela retornará 1 caso for necessário inserir no filho esquerdo do nó e retornará 2 caso for inserir no filho direito do nó.

**Busca palavra:**

int buscaPalavra(PNO \*raiz, char palavra[20], int flag){

    if(strcmp(palavra,raiz->palavra) == 0){ // casso a palavra seja igual a palavra armazenada no nó/raiz

        flag = true;

    }else{

        if( folha(raiz) == false){ // caso não seja folha, ele irá ficar percorrendo pelo filhos que estão a esquerda

            flag = buscaPalavra(raiz->filho\_esq,palavra,flag);

            if(flag == 0){ // não foi nada encontrado nos filhos da esquerda, agora irá percorrer pelos filhos a direita

                flag = buscaPalavra(raiz->filho\_dir,palavra,flag);

            }

        }else{ // caso não seja encontrado nada

            flag = 0;

        }

    }

    return flag;

}

BuscaPalavra irá busca na raiz árvore se a palavra passada no parâmetro da função está na árvore, caso esteja irá retornar 1, caso contrário irá retornar 0.

**Atualizar linha:**

PNO \*atualizarLinha(PNO \*raiz, char palavra[20]){

    if(strcmp(palavra,raiz->palavra) == 0){ // se raiz/nó tenha tenha a string, irá ser atualizado a lista de linhas

        TELE \*carga\_nova = criarElemento(linha);

        inserirLista(raiz->linhas,carga\_nova);

        return raiz;

    }else{

        if(folha(raiz) == false){ // irá percorre os filhos caso a raiz/nó não forem folhas;

           raiz->filho\_esq = atualizarLinha(raiz->filho\_esq,palavra);

           raiz->filho\_dir = atualizarLinha(raiz->filho\_dir,palavra);

        }else{ // caso a raiz/nó não for folha, não há o que se atualizar, uma vez que eles não armazenam palavras

            return raiz;

        }

    }

    return raiz;

}

A função irá verificar se a string está na árvore, caso esteja vai ser atualizado a lista de linhas, essa atualização será dada inserindo a linha atual que está sendo verificada na lista. Por isso, tornamos a linha como uma variável global.

**Inserir Árvore:**

PNO \*inserirArvore(PNO \*raiz\_pai, PNO \*raiz, char palavra\_nova[20]){

    if(palavra\_nova[0]=='\*'){ // caso seja um asterisco, é para pular a linha

        linha++;

        return raiz;

    }

    // caso a raiz seja nula;

    if(raiz == NULL){

        raiz = criarRaiz(0,palavra\_nova,strlen(palavra\_nova));

        TELE \*carga = criarElemento(linha);

        inserirLista(raiz->linhas, carga);

    // raiz != NULL

    }else{

        // primeiro será verificado se a palavra já não está na árvore

        int flag = buscaPalavra(raiz,palavra\_nova,flag);

        if(flag == 1){ // se estiver na árvore, será atualizado lista de linhas com essa palavra associada

            raiz = atualizarLinha(raiz, palavra\_nova);

            return raiz;

        // caso ela não esteja

        }else{

                if(folha(raiz) == true){ // caso se o nó/raiz for uma folha

                // sera obtido a posicao do ultimo caracter em comum com a palavra nova e com a palavra da raiz

                int pos\_caracter;

                pos\_caracter = ultimoCharComum(raiz->palavra,palavra\_nova);

                char caracterString[20]={""};

                strncpy(caracterString, palavra\_nova, pos\_caracter + 1);

                if(raiz\_pai != NULL){

                    // Se sub-string criada for igual a sub-string do pai, vai ser obtida uma nova sub-string

                    if(strcmp(caracterString,raiz\_pai->palavra) == 0){

                     pos\_caracter = primeiroCharDistinto(raiz->palavra,palavra\_nova);

                        strncpy(caracterString,maiorStringChar(raiz->palavra,palavra\_nova), pos\_caracter + 1);

                    }

                }

                PNO \*raiz\_antiga = raiz;

                int nova\_pos\_char = 0;

                if(raiz\_pai != NULL){

                    nova\_pos\_char = primeiroCharDistinto(raiz\_pai->palavra,palavra\_nova);

                }

                // criando uma nova raiz com uma sub-string em comum

                PNO \*raiz\_nova = criarRaiz(nova\_pos\_char,caracterString,1);

                // criando o novo nó/filho com a nova palavra que vai ser inserida

                PNO \*novo\_filho = criarRaiz(primeiroCharDistinto(raiz\_nova->palavra,palavra\_nova),palavra\_nova,strlen(palavra\_nova));

                TELE \*carga\_nova = criarElemento(linha);

                inserirLista(novo\_filho->linhas,carga\_nova);

                // atualizando pos\_caracter da raiz antiga

                raiz\_antiga->pos\_caracter = primeiroCharDistinto(raiz\_nova->palavra, raiz\_antiga->palavra);

                // definindo os filhos

                raiz\_nova->filho\_esq = filhoMenor(raiz\_antiga,novo\_filho);

                raiz\_nova->filho\_dir = filhoMaior(novo\_filho,raiz\_antiga);

                return raiz\_nova;

            }else{ // caso o nó/raiz não seja uma folha, será verificado onde deve ocorrer a inserção

                    if(ladoInsercao(raiz,palavra\_nova) == 1){

                    raiz->filho\_esq = inserirArvore(raiz,raiz->filho\_esq, palavra\_nova);

                }else if(ladoInsercao(raiz, palavra\_nova) == 2){

                    raiz->filho\_dir = inserirArvore(raiz, raiz->filho\_dir, palavra\_nova);

                }

            }

        }

    }

    return raiz;

}

Primeira a função irá verificar se a nova palavra que deve ser inserida é igual a string \*, caso ela seja igual, quer dizer que houve a quebra de linha, nessa situação será incrementado a variável global linha. Segundo caso irá verificar se a raiz passada na chamada da função é igual a NULL, se for igual, então estamos no caso inicial, e a palavra será inserida na raiz. Se não for nenhum desses dois casos, primeiramente será verificado se a palavra está na árvore, se estiver será chamada à função atualizaLinha, caso não esteja, será analisado se á raiz é folha, caso seja, é necessário que tenha a inserção ali, será verificados as condições e será inserido o elemento. Caso ela não seja uma folha, então irá percorrer pelos filhos, no caso será usado a função ladoInserção para saber em que filho deve ser feito a inserção.

**FUNCIONAMENTO E TESTES**

Na main() do código, serão inicializadas uma variável raiz que recebe NULL, uma string auxiliar chamada palavra, que irá receber as strings do arquivo, e também o arquivo. Será verificado se o ponteiro do arquivo é igual a NULL, caso seja, houve uma má referência do arquivo, caso contrário, a palavra irá receber as strings do arquivo, será tirada as pontuações e a string terá todos os seus caracteres em minúsculo. Depois dessas alterações, a palavra será inserida na árvore caso ela seja maior ou igual a 3 e se ela for a string \*.

if(arquivo == NULL){

        printf("Arquivo nao encontrado\n");

    }else{

        while(!feof(arquivo)){

            fscanf(arquivo,"%s",palavra); // a string palavra recebendo a string do texto

            strlwr(palavra); // deixa em letra minúscula

            strcpy(palavra, tirarPontuacao(palavra)); // tirando as pontuações da palavra

            // fará a inserção caso a palavra seja maior ou igual a 3  ou ela seja apenas \* simbolizando a leitura da outra linha

            if(strlen(palavra)>=3 || strcmp(palavra,"\*") == 0){

                raiz=inserirArvore(NULL,raiz,palavra);

            }

        }

    }

O programa utiliza como entrada um arquivo de texto .txt, ao ser executado, ele irá diretamente imprimir as letras em ordem alfabética do texto, então caso seja necessário mudar o arquivo para efetuar outro teste, é necessário que altere a referência que está no fopen na última linha do código a seguir:

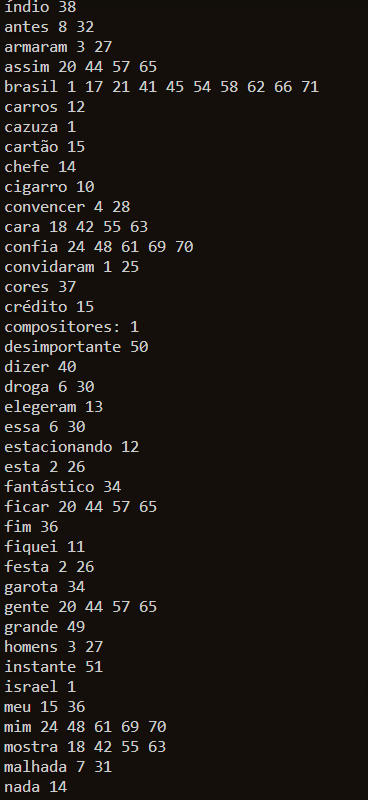
setlocale(LC\_ALL, "portuguese\_Brazil");

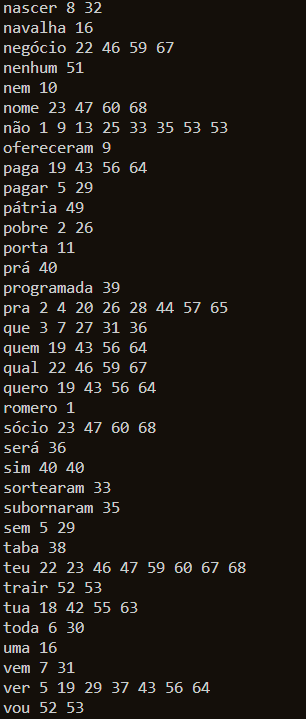
    FILE \*arquivo;

    arquivo = fopen("alterar\_aqui.txt","r");

Os testes foram feitos a partir dos documentos de textos disponibilizados no colabweb.

**1º Teste:**

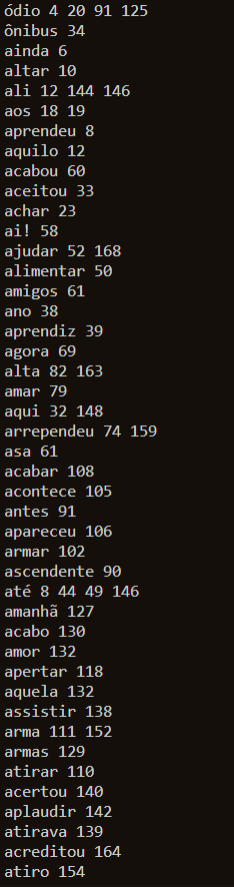
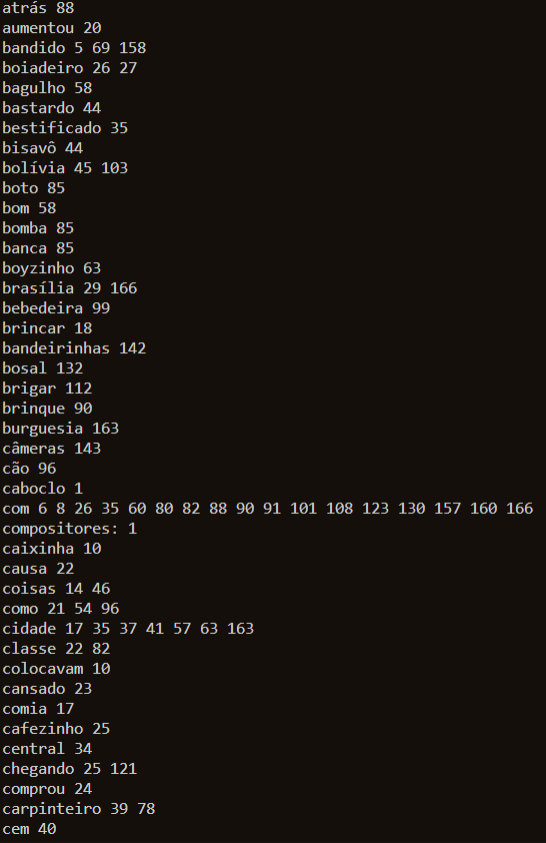
brasil.txt



Coluna II – teste 1

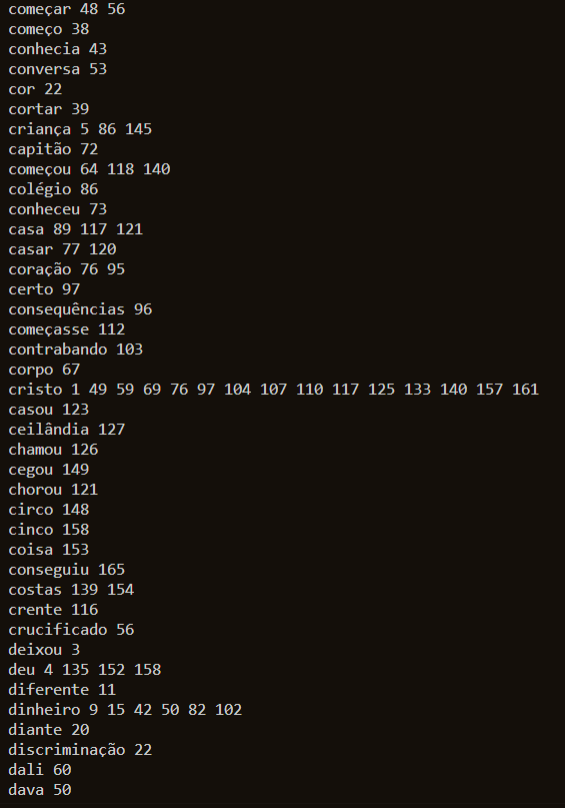
Coluna I – teste 1

**Teste 2:**

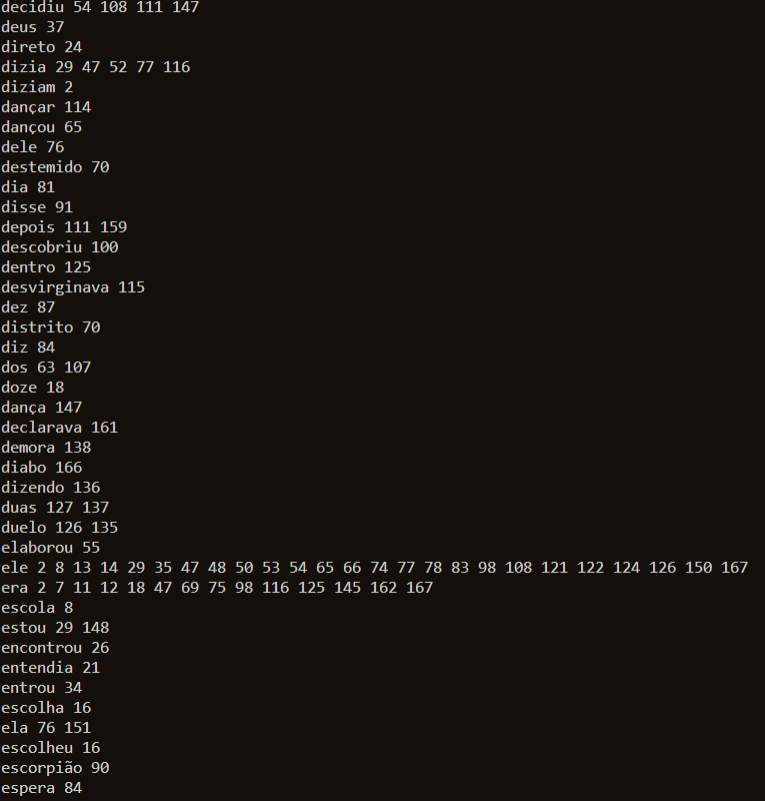
faroeste.txt

Coluna II – teste 2

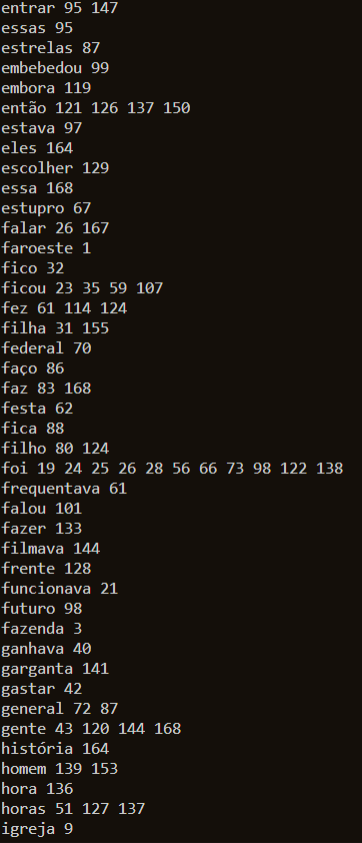
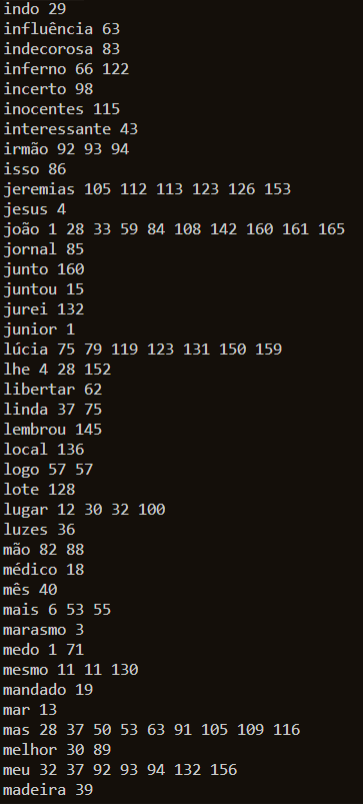
Coluna I – teste 2



Coluna III – teste 2

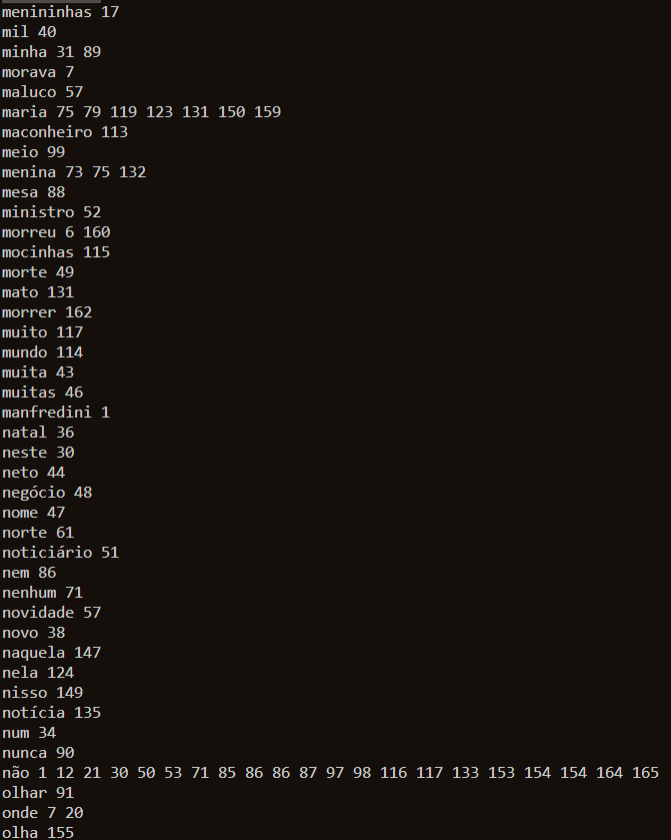


Coluna IV – teste 2

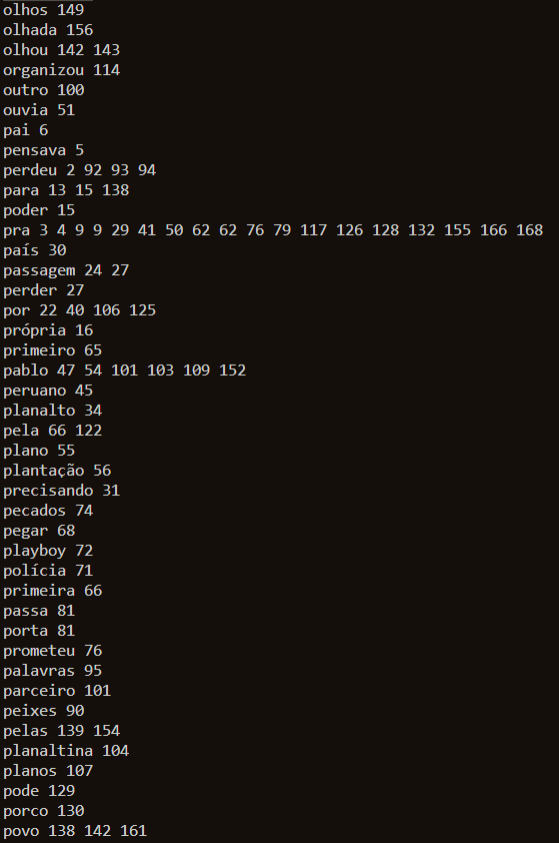


Coluna V – teste 2

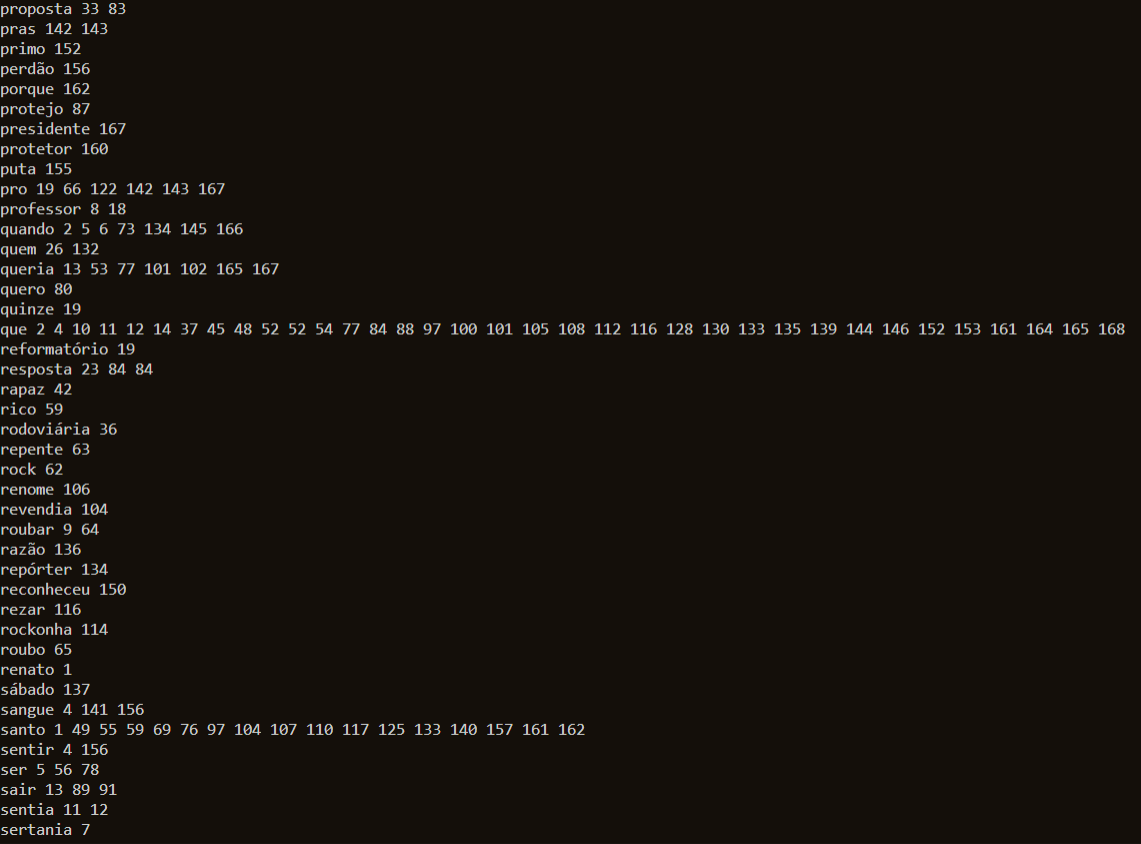
Coluna VI – teste 2



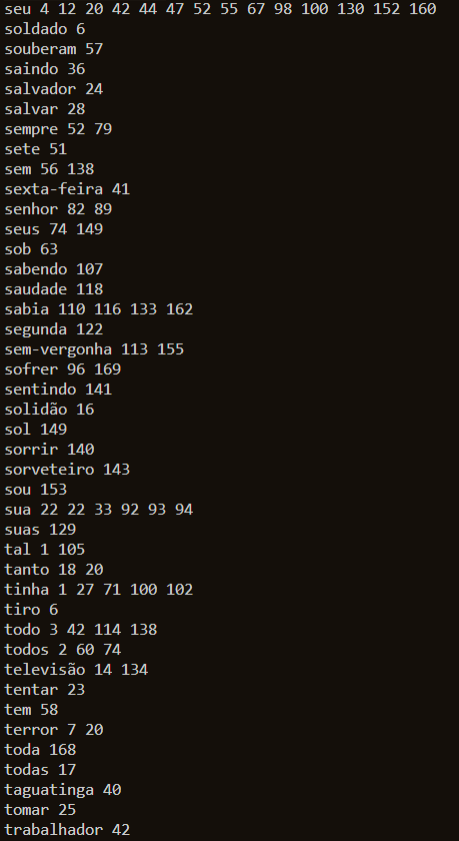
Coluna VII – teste 2



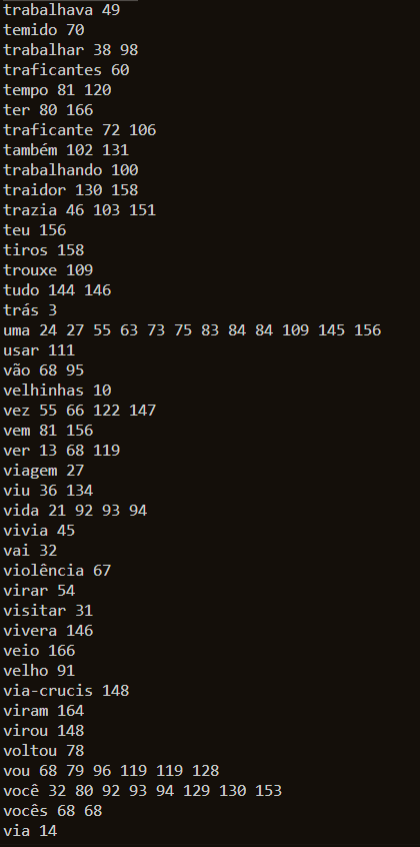
Coluna VIII – teste 2



Coluna IX – teste 2



Coluna X – teste 2



Coluna XII – teste 2

Coluna XI – teste 2

**3-CONCLUSÃO**

O objetivo do trabalho foi concluído, foi impresso alfabeticamente as palavras junto com as linhas onde elas aparecem no texto, entretanto vimos que caso o texto tenha palavras com acentos gráficos, as palavras acentuadas são consideradas como menor alfabeticamente, fazendo com que eles sejam mostrados primeiramente.

Tratando-se da construção do código, uma das maiores dificuldades do trabalho foi, primeiramente, a implementação das árvores patrícias, existe muito pouco material disponibilizado na internet que nos dá informação clara de como acontece a inserção na árvore, ficando assim difícil de efetuar a implementação. A outra dificuldade, bem menor comparada com a citada anteriormente, foi como seria para pegar as strings do arquivo, e depois veio o problema das strings terem pontuações, tendo a necessidade de criar uma função que removesse as pontuações. Apesar desses problemas que surgiram durante a implementação do código, foram encontradas soluções para cada um, de forma que fosse possível realizar o trabalho.