Jeferson Luiz Alves de Souza

2018103110

Essa é a função mais complexa do programa. Ela é destinada a criar uma lista de prioridade contabilizando todas as ocorrências de cada palavra, pontuação e espaço em branco. Ao final, essa lista de prioridades estará ordenada e será devolvida a função que a chamou (a função que cria uma árvore com esses dados).

List \*createlistMainSort(){

List \*listMain = NULL;

FILE \*myArchive = fopen("inicial.txt", "r");

FILE \*ptr = fopen("inicial.txt", "r");

char x[10];

char y[1];

char aux[10];

int i = 0;

int j = 0;

bool cont = false;

Agora com os arquivos de entrada abertos, iremos entrar em um laço que percorre CADA caracter do arquivo texto de entrada. (EOF significa o final de um arquivo. Obs.: não confunda com ‘\0’ nem ‘\n’).

while((aux[j] = fgetc(ptr)) != EOF){

Caso o atual caracter de *aux* for um espaço em branco ↓.

if(aux[j] == 32){

Essa variável *cont* é necessária para contabilizar quando a palavra contida em *x* já foi adicionada ao final do arquivo ou ainda está para ser adicionada. Caso entre nessa condição, significa que a palavra ainda não foi adicionada.

if(!cont){

(x, &listMain);

Sempre que eu adicionar uma palavra, preciso setar minha string para que não haja restos.

cleanString(x,10);

i = 0;

cont = true;

y[0] = aux[j];

addWord(y, &listMain);

cleanString(y,1);

cleanString(aux,10);

fgetc(myArchive);

j = -1;

}

Essa sequência de linhas acima irá se repetir por algumas vezes durante a função atual. Elas são necessárias para zerar variáveis, reiniciar strings, etc. Pois quando ocorre o ato de chegar a essa função, significa que uma palavra, uma pontuação ou um espaço em branco foi adinionado.

else{

y[0] = aux[j];

addWord(y, &listMain);

cleanString(y,1);

cleanString(aux,10);

fgetc(myArchive);

j = -1;

}

}

else{

Caso o atual caracter é um sinal de pontuação.

if((aux[j] >= 33 && aux[j] <= 47) || (aux[j] >= 58 && aux[j] <= 64) || (aux[j] >= 91 && aux[j] <= 96) || (aux[j] >= 123 && aux[j] <= 126)){

Caso a palavra ainda não tenha sido adicionada.

if(!cont){

addWord(x, &listMain);

Sempre que eu adicionar uma palavra, preciso setar minha string para que não haja restos.

cleanString(x,10);

i = 0;

cont = true;

y[0] = aux[j];

addWord(y, &listMain);

cleanString(y,1);

cleanString(aux,10);

fgetc(myArchive);

j = -1;

}

Caso já tenha sido adicionada.

else{

y[0] = aux[j];

addWord(y, &listMain);

cleanString(y,1);

cleanString(aux, 10);

fgetc(myArchive);

j = -1;

}

}

Esse é a condição que mais veremos ser passada nesse programa. Essa é a parte aonde o caracter atual é uma letra. Aqui ele é adicionada a uma variável x e contabilizada para ser posteriormente adicionada.

else{

Sempre que entrar aqui, significa q uma letra foi adicionada, ou seja, n foi adicionada a palavra atual.

cont = false;

x[i] = fgetc(myArchive);

i += 1;

}

}

j += 1;

}

fclose(myArchive);

fclose(ptr);

Essa função devolve a lista ordenada POR NÚMERO DE OCORRÊNCIA e não, por tamanho de string.

List \*listMainReturn = orderlistMain(&listMain);

return listMainReturn;

}

Função simples que remove o menor elemento (primeiro) de uma lista de prioridade (*listMain)* e o retorna o elemento da árvore.

Tree \*returnMin(List \*\*listMain){

List \*auxlistMain = NULL;

Tree \*auxTree = NULL;

if(\*listMain){

auxlistMain = \*listMain;

\*listMain = (\*listMain)->next;

auxTree = auxlistMain->ptr;

free(auxlistMain);

}

return auxTree;

}

Essa função retornar um ponteiro para a posição aonde deve ser SOMADO (caso o elemento já exista), ou retorna nulo caso n exista.

List \*ifExitInlistMain(char \*p, List \*listMain){

if(!listMain) return NULL;

List \*aux = listMain;

while(aux){

Caso tenhamos encontrado o elemento. Ou seja, o elemento já existe na lista, só precisamos somar +1 à sua quatidade.

if((strcmp(aux->ptr->chave, p) == 0)) return aux;

aux = aux->next;

}

return NULL;

}

Essa função adiciona uma string na lista de prioridade (*ListMain*) ou soma se a string já existir.

void addWord(char \*p, List \*\*listMain){

List \*aux = NULL;

List \*ptr = NULL;

Tree \*newTree = newNodeTree(1, p);

List \*position;

Caso a lista passada como parâmetro seja nula. Ou seja, iremos inserir o primeiro elemento.

if(\*listMain == NULL){

\*listMain = newNodelistMain(newTree);

return;

}

Primeiro precisamos saber se o elemento já existe na árvore.

Essa função retorna a posição aonde deve ser somado um elemento.

if((position = ifExitInlistMain(p, \*listMain)) != NULL) position->ptr->freq += 1;

else{

ptr = newNodelistMain(newTree);

aux = \*listMain;

while(aux->next != NULL) aux = aux->next; //final da listMaina

aux->next = ptr;

}

}

Função destinada a ordenar uma lista de prioridade e a retornar.

List \*orderlistMain(List \*\*listMain){

List \*newlistMain = NULL;

List \*aux = \*listMain;

List \*smaller = NULL;

List \*auxNewlistMain = NULL;

List \*ptr = NULL;

Essa variável é destinada a dizer se em certos momentos, o menor elemento é REALMENTE o primeiro elemento ou o segundo elemento, já que iremos partir da ideia de que o ponteiro auxiliar sempre será a posição estreitamente anterior a posição a ser adicionada.

int checker = 0

while(\*listMain != NULL){

smaller = \*listMain;

aux = \*listMain;

checker = 0;

while(aux){

if(checker > 0) if((aux->ptr->freq < smaller->next->ptr->freq)) smaller = ptr;

Encontramos um elemento menor que o atual menor

else if((aux->ptr->freq < smaller->ptr->freq)){

checker += 1;

smaller = ptr;

}

ptr = aux;

aux = aux->next;

}

Ou seja, se checker for > 0, significa q smaller não é o primeiro elemento.

se for o primeiro elemento ↓.

if(newlistMain == NULL){

Isso significa que smaller é ponteiro para o primeiro elemento E É EXATAMENTE o elemento a ser removido.

Caso o menor elemento esteja no inicio da lista de prioridade.

if(smaller == \*listMain && checker == 0){

newlistMain = smaller;

\*listMain = (\*listMain)->next;

}else{

newlistMain = smaller->next;

smaller->next = smaller->next->next;

}

auxNewlistMain = newlistMain;

}

Se não for o primeiro elemento.

else{

Isso significa que smaller é ponteiro para o primeiro elemento E ele É EXATAMENTE o elemento a ser removido.

Caso o menor elemento esteja no inicio da lista de prioridade

if(smaller == \*listMain && checker == 0){

auxNewlistMain->next = smaller;

\*listMain = (\*listMain)->next;

}else{

auxNewlistMain->next = smaller->next;

smaller->next = smaller->next->next;

}

auxNewlistMain = auxNewlistMain->next;

}

}

auxNewlistMain->next = NULL;

return newlistMain;

}

Função destinata a setar a string como varia. Em outras palavras, vai pegar a string e vai adicionar '\0' para todo seu tamanho.

void cleanString(char \*x, int size){

for(int i = 0; i < size; i++) x[i] = '\0';

}

Função que insere em uma arvore de prioridades em uma lista de prioridades.

void insertlistMain(List \*\*listMain, Tree \*z){

Caso a lista seja vazia, ou seja, precisamos adicionar à lista, o primeiro elemento.

if(!\*listMain){

\*listMain = newNodelistMain(z);

return;

}

bool checker = true;

List \*aux = \*listMain;

List \*ptr = NULL;

List \*x = NULL;

List \*w = NULL;

Esse laço vai rodar enquanto o novo nó da árvore não foi adicionado na lista de prioridades.

while(checker){

Pelo "teste de mesa" precisamos fazer o caso em que deve ser inserido o último elemento. Contamos que a lista de prioridades nunca será checada quando n houver nenhum elemento.

if(aux == NULL){

w = newNodelistMain(z);

ptr->next = w;

w->next = NULL

checker = false;

}

Caso o elemento atual é maior que o elemento a ser inserido.

else if(aux->ptr->freq > z->freq){

Caso o elemento será inserido no inicio da lista.

if(ptr == NULL){

x = newNodelistMain(z);

x->next = \*listMain;

\*listMain = x;

}

Caso não seja o primeiro elemento, significa que *ptr* existe, ou seja, é diferente de nulo.

else{

w = newNodelistMain(z);

ptr->next = w;

w->next = aux;

}

Ou seja, independente da condição que entrar, o novo nó será adicionado. Por isso:

checker = false;

}

Caso a variável já for falsa, significa que não precisamos mais mexer no reapontamento das variáveis locais.

if(checker){

ptr = aux;

aux = aux->next;

}

}

}

Função que cria uma árvore a partir de uma lista de prioridades encadeada.

Tree \*createTree(){

Tree \*z = NULL;

Tree \*x = NULL;

Tree \*y = NULL;

Aqui, teremos nossa lista de prioridades formada e ordenada.

List \*listMain = createlistMainSort();

int n = sizelistMain(listMain);

Esse próximo laço é destinado a criar nossa árvore de prioridades.

for(int i = 1; i < n; i++){

Usaremos em nosso código um tipo de nó interno diferente. Usaremos seu campo char como “” para identificação.

z = newNodeTree(0, "");

//printf("%d (%s) \n", z->freq, z->chave);

x = returnMin(&listMain);

z->right = x;

y = returnMin(&listMain);

z->left = y;

z->freq = x->freq + y->freq;

z->chave = "";

insertlistMain(&listMain, z);

z = NULL;

}

Retornamos nossa árvore.

return returnMin(&listMain);

}

Função que retorna o tamanho de uma lista de prioridades. Retorna quantos ELEMENTOS tem.

int sizelistMain(List \*listMainMain){

List \*aux = listMainMain;

int cont = 0;

Laço vai rodar enquanto não chegar ao final da lista.

while(aux){

cont += 1;

aux = aux->next;

}

return cont;

}

Função que cria um novo nó da arvore.

Tree \*newNodeTree(int freq, char \*chave){

Essas duas próximas linhas são necessárias pois, encontramos um problema no programa que estava criando uma cópia da *chave*, então, sempre que modificada, modificava também todas as supostas strings adicionadas anteriormente.

char \*aux = (char\*)malloc(sizeof(char));

for(int i = 0; i < 10; i++) aux[i] = chave[i];

Tree \*ptr = (Tree\*)malloc(sizeof(Tree));

if(ptr){

ptr->checker = false;

ptr->freq = freq;

ptr->chave = aux;

ptr->left = NULL;

ptr->right = NULL;

}

return ptr;

}

Função que cria um novo nó de uma lista de prioridades a partir de um nó da árvore.

List \*newNodelistMain(Tree \*z){

List \*ptr = (List \*)malloc(sizeof(List));

if(ptr){

ptr->ptr = z;

ptr->next = NULL;

}

return ptr;

}

Função que imprime a árvore com tabulação.

void printTree(Tree \*tree, int nivel){

int i;

if(tree != NULL){

printTree(tree->right, nivel+1);

for(i = 0; i < nivel; i++) printf("\t");

if(tree->chave == "") printf("(%d)\n", tree->freq);

else printf("[%s]\n", tree->chave);

printTree(tree->left, nivel+1);

}

}

Função que imprime todos os elementos de uma lista ordenada. Imprime da forma:

[string][qtd. de ocorrência].

void printlistMain(List \*heapMain){

List \*aux = heapMain;

printf("\n-----print listMain-----");

while(aux){

printf("\n[%s][%d]", aux->ptr->chave, aux->ptr->freq);

aux = aux->next;

}

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

}

Essa função será usada para percorrer e marcar o campo da struct Tree chamado Checker.

É uma função recursiva. Ela vai marcar o caminho mais rápido até o nó folha requerido.

void marcaLeaf(char \*p, Tree \*tree){

if(tree == NULL) return;

marcaLeaf(p, tree->left);

marcaLeaf(p,tree->right);

Quando ela retornar da direita, significa que encontramos um nó folha.

Significa que encontramos a chave desejada ↓.

if(strcmp(p, tree->chave) == 0){

tree->checker = true;

}

Quando retornar, preciso verificar se o esquerda ou direita já foi modificado. Pode ocorrer em alguma situação que um nó interno não tenha um filho a direita ou a esquerda então deve-se tomar cuidado.

if(tree->left) if(tree->left->checker == true) tree->checker = true;

if(tree->right) if(tree->right->checker == true) tree->checker = true;

}

É uma função semelhante à função anterior. Porém ela DESMARCA o caminho feito para encontrar a última ocorrência requerida da árvore.

void desmarcaLeaf(char \*p, Tree \*tree){

Tree \*aux = tree;

while(strcmp(p, aux->chave) != 0){

aux->checker = false;

if(aux->left->checker == true) aux = aux->left;

else aux = aux->right;

}

aux->checker = false;

}

Função importante que pega uma ocorrência de um arquivo a ser codificado. Ele adiciona ao final do arquivo de saída o código binário dessa *string*.

void addExitArchive(char \*p, FILE \*exiteArchive, Tree \*tree){

Vai marcar o caminho em que a folha se encontra agora com o caminho, iremos adicionar a sequência no arquivo de saída.

marcaLeaf(p, tree);

Tree \*aux = tree;

Laço vai rodar enquanto não foi encontrado a palavra.

while(strcmp(p, aux->chave) != 0){

if(aux->left->checker == true){

aux = aux->left;

48 = zero em ASCII.

fputc(48,exiteArchive);

}

else{

aux = aux->right;

49 = 1 em ASCII.

fputc(49,exiteArchive);

}

}

desmarcaLeaf(p, tree);

}

Função destinada a codificar um arquivo de entrada.

void codificaArchive(Tree \*tree){

Abre o arquivo de leitura ↓. Obs.: o arquivo deve existir.

FILE \*entryArchive = fopen("inicial.txt", "r");

Abre o arquivo para gravação ↓. Obs.: o arquivo não precisa existi.

FILE \*exitArchive = fopen("codificado.txt", "wb");

Abre um arquivo auxiliar ↓.

FILE \*myArchive = fopen("inicial.txt", "r");

char x[10];

char y[1];

char aux[10];

int i = 0;

int j = 0;

bool cont = false;

Percorre o arquivo enquanto não chega ao finao (EOF).

while((aux[j] = fgetc(entryArchive)) != EOF){

Caso a posição atual do arquivo seja um espaço em branco.

if(aux[j] == 32){

Caso a palavra anterior ainda não tenha sido adicionada.

if(!cont){

addExitArchive(x, exitArchive, tree);

Sempre que eu adicionar uma palavra, preciso setar minha string para que não haja restos.

cleanString(x,10);

i = 0;

cont = true;

y[0] = aux[j];

addExitArchive(y, exitArchive, tree);

cleanString(y,1);

cleanString(aux,10);

fgetc(myArchive);

j = -1;

}

Essa sequência de linhas acima irá se repetir por algumas vezes durante a função atual. Elas são necessárias para zerar variáveis, reiniciar strings, etc. Pois quando ocorre o ato de chegar a essa função, significa que uma palavra, uma pontuação ou um espaço em branco foi adinionado.

else{

y[0] = aux[j];

addExitArchive(y, exitArchive, tree);

cleanString(y,1);

cleanString(aux,10);

fgetc(myArchive);

j = -1;

}

}

else{

Caso o atual caractere seja pontuação.

if((aux[j] >= 33 && aux[j] <= 47) || (aux[j] >= 58 && aux[j] <= 64) || (aux[j] >= 91 && aux[j] <= 96) || (aux[j] >= 123 && aux[j] <= 126)){

if(!cont){

addExitArchive(x, exitArchive, tree);

cleanString(x,10);

i = 0;

cont = true;

y[0] = aux[j];

addExitArchive(y, exitArchive, tree);

cleanString(y,1);

cleanString(aux,10);

fgetc(myArchive);

j = -1;

}

else{

y[0] = aux[j];

addExitArchive(y, exitArchive, tree);

cleanString(y,1);

cleanString(aux, 10);

fgetc(myArchive);

j = -1;

}

}

else{

Sempre que entrar aqui, significa q uma letra foi adiciona, ou seja, n foi adiciona a palavra atual.

cont = false;

x[i] = fgetc(myArchive);

i += 1;

}

}

j += 1;

}

fclose(myArchive);

fclose(entryArchive);

fclose(exitArchive);

}

Função destinada a decodificar um arquivo a partir de uma árvore de ocorrências e um arquivo de entrada.

void decodificaArchive(Tree \*tree){

Essa função irá percorrer a árvore mediante cada inteiro (0 ou 1) que ler no arquivo. quando o auxiliar chegar a um nó folha, ele para, e adiciona aquele caracter na arvore.

Após isso, ele volta a ser a raiz e continua percorrendo o arquivo enquanto ele não termina.

Para leitura, o arquivo deve existir ↓.

FILE \*archive = fopen("codificado.txt", "r");

FILE \*exitArchive = fopen("decodificado.txt", "a");

char x[1];

Tree \*aux = tree;

Laço irá rodar enquanto o final do arquivo não chega.

while((x[0] = fgetc(archive)) != EOF){

if(x[0] == 49) aux = aux->right;

else aux = aux->left;

Verifica se já chegou em um nó folha (um nó com conteúdo).

if(!aux->left & !aux->right){

Então precisamos adicionar no arquivo de saída e fazer *aux* ser a raiz novamente.

fputs(aux->chave, exitArchive);

aux = tree;

}

}

fclose(archive);

fclose(exitArchive);

}