Documentação TP1-EDII

Sumário

Introdução

O programa tem como objetivo ler vários documentos e então gravar na árvore binária e na tabela hash as palavras que estiverem nos documentos, após a gravação será feita uma consulta nas estruturas as palavras e os documentos que as contém e no final mostra quais os documentos que possuem todas as palavras pesquisadas.

O programa receberá como argumento de entrada o caminho do diretório que terá os documentos que terão suas palavras lidas e salvas na estrutura, e o nome de um arquivo contendo as palavras que serão buscadas.

No desenvolvimento do código foi usando as seguintes funções:

lista criaLista(): cria a lista do nodo da árvore ou da posição na tabela

int insereInicio(lista, Frequencia): Insere no início da lista

int insereFinal(lista, Frequencia): Insere no final da lista

int alteraFrequencia(lista, int): Incrementa a frequência de uma palavra em um documento na lista de uma palavra

void insereNodoLista(lista, lista): Cria uma lista auxiliar com todos os documentos de das palavras consultadas

int removeElemento(lista, int): Remove um elemento da lista

int alteraLista(lista, int): Verifica se os documentos possuem todas as palavras

void encerraLista(lista): Encerra uma lista que não será mais usada

lRanking criaListaRanking(): Cria a lista do ranking

int insereInicioRanking(lRanking, Ranking): Insere no início da lista do ranking

int insereFinalRanking(lRanking, Ranking): Insere no final da lista do ranking

void criaRanking(lRanking, lista, int*, int):

void ordenaLista(lRanking): Ordena os valores da lista do ranking

int verificaElementoRanking(lRanking l): Verifica os elementos do ranking e remove os documentos que não possuem todas as palavras

int removeElementoRanking(lRanking, int): Remove todos os documentos de um determinado número

void encerraListaRanking(lista): Encerra a lista de ranking

tabela criaTabela(): Cria a tabela hash

int insereTabela(tabela, char*, Frequencia): Insere a palavra na tabela

lista pesquisaTabela(tabela, char*): Pesquisa uma palavra na tabela

lRanking ordenaDocumentosHash(tabela, char *, NumeroArquivo[], int): Encontra as palavras na tabela Hash, coloca as palavras em uma lista auxiliar e depois as coloca na lista de ranking e então ordena em ordem decrescente

apontador criaArvore(): Cria uma árvore binária

lista insereNodoABP(apontador*, Tpalavras): Insere os elementos nos nodos da árvore binária

lista pesquisaABP(apontador, char*): Pesquisa uma palavra na árvore binária

lRanking ordenaDocumentosArvore(apontador, char*, NumeroArquivo[], int): Encontra as palavras na árvore binária, coloca as palavras em uma lista auxiliar e depois as coloca na lista de ranking e então ordena em ordem decrescente

int inicio(int, char*[]): Inicia o programa

int inicioArvore(int, char*[]): Inicia as operações com a árvore binária

int inicioHash(int, char*[]): Inicia as operações com a árvore binária

void retiraBarraN(char*): Retira o \n das palavras da consulta

FILE* abrirArquivo(char*): Abre os arquivos

char** alocaString(int): Aloca uma string dinâmica de acordo com a quantidade de palavras em uma consulta

As estruturas usadas são:

```
typedef struct{
   char nomeArquivo[100];
   int numArquivo;
}NumeroArquivo;
```

Tabela que vai armazenar o nome do documento e um número correspondente ao mesmo.

```
typedef struct{
   int arquivo, frequencia;
}Frequencia;

typedef struct NodoLista{
   Frequencia info;
   struct NodoLista *next;
}NodoLista;

typedef struct{
   NodoLista *first, *last;
   int tamanho;
```

```
}Lista;
```

```
typedef Lista *lista;
```

Estruturas da lista, a struct frequência vai ficar na lista dentro do nodo, mostrando o número correspondente do documento e a frequência em que aparece na palavra.

```
typedef struct{
   int arquivo;
   float ranking;
}Ranking;
```

```
typedef struct NodoRanking{
   Ranking info;
   struct NodoRanking *next;
}NodoRanking;
```

```
typedef struct{
   NodoRanking *first, *last;
   int tamanho;
}ListaRanking;
```

```
typedef ListaRanking *lRanking;
```

Estruturas da lista de ranking, a struct ranking vai ter o número do arquivo e um número de ranking para saber qual arquivo será mostrado primeiro.

```
typedef struct{
   lista elementos[M];
   int numElementos;
}Ttabela;
```

Estrutura da tabela hash, onde cada posição terá uma lista.

```
typedef struct{
   char palavra[100];
```

}TPalavras;

typedef struct Nodo{

TPalavras info;

struct Nodo *esq, *dir;

Lista *lista;

}TNodo;

typedef TNodo *apontador;

Estrutura da árvore binária, onde cada nodo terá uma lista para os números dos documentos e a frequência que as palavras aparecem nos documentos.

Implementação

int inicioArvore(int argc, char* argv[]): é a função responsável por iniciar as operações na árvore binária, sendo passado como parâmetro os argumentos de entrada que são o diretório onde estão os arquivos e o nome de um arquivo onde terá as palavras que serão consultadas.

```
dp = opendir(argv[1]);
while((d = readdir(dp)) != NULL){ //Abrindo o diretório
    if(d->d_type == DT_REG){    //Lendo os arquivos
        strcpy(NA[numVetor].nomeArquivo, d->d_name);
        NA[numVetor].numArquivo = numVetor + 1;
        quantidadeArquivo++;

        strcat(caminhoArquivo, argv[1]);    //Montando caminho para o arquivo
        strcat(caminhoArquivo, "/");
        strcat(caminhoArquivo, d->d_name);

        arq = abrirArquivo(caminhoArquivo);    //Abrindo o arquivo com as palavras que serão indexadas

        lerArquivoArvore(&raiz, arq, NA[numVetor].numArquivo);

        fclose(arq);

        numVetor++;

        memset(caminhoArquivo, '\0', 100);
    }
}
```

Esse algoritmo primeiramente abre o diretório onde estão os arquivos a partir do argumento de entrada do usuário e então ele vai abrir o diretório e vai ler os arquivos do diretório um por um, se for um arquivo então o nome do arquivo vai ser copiado para uma tabela que vai guardar o nome do arquivo e então esse arquivo vai receber um número que o representará a variável quantidade de arquivos será incrementada, depois será feito uma sequência de concatenações para abrir o arquivo e então o arquivo será enviado para uma função que irá ler as palavras nele, após isso o arquivo será fechado e a posição no vetor que guarda o nome do arquivo será incrementada e a string que guarda o caminho do arquivo será limpada para ser usada novamente na próxima iteração. A

função inicioHash(), possui uma parte semelhante, seu algoritmo é o mesmo da inicioÁrvore() com exceção que ela chama a função lerArquivoHash().

void lerArquivoArvore(apontador *p, FILE *arquivo, int numArquivo): é a função que vai ler cada palavra do arquivo e então vai salvar na estrutura da árvore, a função recebe um ponteiro para a raiz da árvore, um ponteiro para o arquivo e o número do arquivo para inserir na lista do nodo.

```
while(!feof(arquivo)){
    memset(pa.palavra, '\0', 100);  //limpa a string
    fscanf(arquivo, "%s", pa.palavra);  //Lendo as palavras do arquivo
    if(pa.palavra[0] == '\0')   //Se a string for vazia sai da função
        break;

    l = insereNodoABP(p, pa); //Inserindo palavra no nodo
    if(l->tamanho == 0){    //Se o tamanho da lista for igual a zero incrementa a frequência como 1
        f.arquivo = numArquivo;
        f.frequencia = 1;
        insereFinal(l, f);  //Inserindo o numero do arquivo e a frequência que a palavra aparece no arquivo na lista do nodo da árvore
    }

    else
        if(alteraFrequencia(l, numArquivo) == 1) //Se o número do arquivo ja estiver na lista a frequência é incrementada
            continue;

        else{       //Se o número do arquivo não estiver na lista acescenta com frequência igual a 1
            f.arquivo = numArquivo;
            f.frequencia = 1;
            insereFinal(l, f);
        }
}
```

Esse algoritmo vai ler todas as palavras do arquivo até o fim do arquivo ou até a palavra for uma string vazia, após a verificação se é uma string vazia, uma váriavel do tipo lista recebe um ponteiro para lista do nodo de uma função que insere a palavra na árvore e então retorna a lista que pertence ao nodo da palavra adicionada ou se já existir recebe a lista da palavra, então é verificado se a lista está vazia, se tiver vazia é adicionado um nodo na lista com o numero do arquivo e frequência da palavra igual a um, se já existir é verificado se a o arquivo já estava na lista e se tiver é incrementado a frequência senão é criado um novo nodo na lista.

lRanking ordenaDocumentosArvore(apontador p, char *palavraEntrada, NumeroArquivo NA[], int quantidadeArquivo): é a função responsável por receber as palavras da consulta e então retornar uma lista ordenada com os documentos onde as palavras aparecem. A função recebe um ponteiro para a raiz da árvore, as palavras que serão buscadas, a struct que possui o nome dos arquivos e a quantidade de arquivos.

Esse algoritmo primeiro verifica e retorna a quantidade de palavras que foi consultado através da quantidade de caracteres **‘ ‘(espaço)** que depois será incrementado para chegar na quantidade de palavras, depois é chamada uma função que vai dividir a frase com as palavras a serem buscadas em um vetor de string, esse vetor de strings será usado em um **loop for** para buscar as palavras na estrutura e então uma variável do tipo lista recebe a referência da lista pertencente à palavra e a lista é copiada para uma lista auxiliar que terá todos os nodos das listas das palavras consultadas.

```
quantidadePalavras = retornaQuantidadePalavras(palavraEntrada); //Recebe a quantidade de palavras que foram consultadas

quantidadePalavras++;
numDocumentos = (int*)malloc(sizeof(int) * quantidadePalavras);  //Vetor que receberá a quantidade de documentos de cada nodo das palavras

string = retornaStringPalavras(palavraEntrada, quantidadePalavras); //Vetor de string contendo as palavras que foram consultadas

for(indice = 0;indice < quantidadePalavras;indice++){
    l = pesquisaABP(p, string[indice]);
    numDocumentos[indice] = l->tamanho;
    insereNodoLista(l, aux);    //Insere os nodos em uma lista auxiliar
}

alteraLista(aux, quantidadePalavras);   //Remove os documentos que não possuem todas as palavras

criaRanking(rankingDocs, aux, numDocumentos, quantidadeArquivo);    //Cria a lista de ranking
encerraLista(aux);      //Encerra lista auxiliar
verificaElementoRanking(rankingDocs);   //Verifica se a lista possui documentos repetidos, se tiver os remove
ordenaLista(rankingDocs);   //Ordena a lista de ranking em ordem decrescente
return rankingDocs;
```

Após o fim do loop será feita a chamada da função ***alteraLista()*** *(*que recebe a lista auxiliar e a quantidade de palavras) que vai remover os documentos que não possuirem todas as palavras, isso será feito através de uma variável que vai contar quantas vezes o documento aparece na lista, se o documento aparecer menos vezes que a quantidade de palavras então quer dizer que alguma das palavras não consta no documento e então todos os nodos com esse documento são excluídos. Quando só estiver na lista os documentos que contém todas as palavras será chamada a função ***criaRanking()*** (recebe uma referência do tipo **lRanking** que terá em cada nodo o número do arquivo e o rank dele para as palavras consultadas, a lista auxiliar, a variável **numDocumentos** que contém a quantidade de documentos das palavras e a quantidade de arquivos) que vai criar um ranking com os documentos através dos cálculos, depois como não há necessidade da lista auxiliar mais ela será encerrada. Depois é chamada a função ***verificaElementoRanking()*** (recebe a lista de ranking) que vai verificar se a lista tem elementos repetidos, se tiver elementos repetidos será chamada uma função para removê-los e por fim será chamada a função ***ordenaLista()*** (recebe a lista de ranking) para ordenar em ordem decrescente através dos rankings nos nodos e a lista de rankings será retornada. Essa função é idêntica a função ***lRanking ordenaDocumentosHash(tabela t, char *palavraEntrada, NumeroArquivo NA[], int quantidadeArquivo)*** com exceção do retorno da lista na estrutura que será feita na tabela hash e não na árvore binária.

int inicioHash(int argc, char* argv[]): tem o mesmo algoritmo usado na anterior, com exceção das estruturas de armazenamento que nessa função será hash e na anterior é árvore binária, também receberá os mesmos parâmetros.

void lerArquivoHash(tabela t, FILE *arquivo, int numArquivo): tem como função ler as palavras do arquivo e salvá-las na tabela hash.

Esse algoritmo vai ler as palavras do arquivo até o final do arquivo ou a string que armazena as palavras receber um caractere **‘ ‘(vazio)**, depois será chamada a função insere tabela que vai receber a palavra e a tabela.

```c
/**
*   Função que lê as palavras no arquivo e salva na estrutura da tabela hash
*/
void lerArquivoHash(tabela t, FILE *arquivo, int numArquivo){
    Frequencia f;
    TPalavras pa;

    while(!feof(arquivo)){
        memset(pa.palavra, '\0', 100);
        fscanf(arquivo, "%s", pa.palavra);
        if(pa.palavra[0] == '\0')
            break;

        f.arquivo = numArquivo;
        f.frequencia = 1;
        insereTabela(t, pa.palavra, f);
    }
}
```

```c
/**
*   Função que insere um elemento na tabela, se o elemento já existir é alterado a frequência do elemento
*/
int insereTabela(tabela t, char *palavra, Frequencia f){
    int indice = hash(palavra);
    if(t->elementos[indice]->tamanho == 0)
        return insereFinal(t->elementos[indice], f);

    else
        if(alteraFrequencia(t->elementos[indice], f.arquivo) == 0)
            insereFinal(t->elementos[indice], f);
    return 1;
}
```

Se a palavra já existir mas o documento não estiver lá a frequência será adicionada como igual a um, se a palavra não existir será adicionado a palavra e o documento com a frequência igual a um e se a palavra existir e o documento já estiver na lista, então a frequência é incrementada.

Experimentos

Nos testes realizados na indexação a árvore binária, após três testes a média de tempo foi de 228 ms enquanto a tabela hash teve um tempo médio de 72 ms.

No teste realizado na consulta, após três consultas na árvore binária a média foi de 27 ms enquanto a tabela hash teve um tempo médio de 6 ms.

Conclusão

A árvore binária e a tabela hash são estruturas para armazenar dados que apresentam uma grande diferença de temo em suas operações.

Nos testes realizados a consulta na tabela hash foi 4,5 vezes mais rápido que a consulta na árvore binária e a indexação na tabela hash foi 3,2 vezes mais rápido que na árvore binária.

Nesse TP foi feito a implementação de um programa que tinha como objetivo salvar várias palavras em duas estruturas diferentes sendo a tabela hash e a árvore binária, depois das palavras indexadas o programa deveria retornar os documentos que tivessem as palavras requisitadas.

Referências

https://www.cplusplus.com/

https://www.gnu.org/software/libc/manual/html_node/Directory-Entries.html