Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático III

BCC202 - Estruturas de Dados I

Vitor Oliveira Diniz
Maria Luiza Aragão
Jéssica Machado
Professor: Pedro Silva

Ouro Preto
22 de março de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Neste trabalho foi necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é a construção de um índice invertido utilizando tabela hash como uma forma de pesquisar um ou mais documentos em um banco de dados textual.

A codificação foi feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais.

1.1 Especificações do problema

Um banco de dados textual é uma coleção de documentos onde cada documento é constituído por uma lista de palavras. Uma estratégia de pesquisa de documentos é comparar a lista de palavras da consulta com a lista de palavras dos documentos, mas no âmbito computacional, a comparação direta é custosa. Uma forma de diminuir esse custo é pela construção de um índice invertido para esses banco de documentos e, neste trabalho, utilizaremos a tabela hash como estrutura de dados.

1.2 Considerações Iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Visual Studio Code LaTeXWorkshop.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *CLANG*: ferramentas de análise estática do código.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: Ryzen 7-5800H.
- Memória RAM: 16 Gb.
- Sistema Operacional: Arch Linux x86_64.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc -c indiceInvertido.c -Wall
gcc -c hash.c -Wall
gcc -c tp.c -Wall
gcc -g indiceInvertido.o hash.o tp.o -o exe
```

Usou-se para a compilação as seguintes opções:

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.
- *-Wall*: para mostrar todos os possível *warnings* do código.

- *-c*: Para compilar o arquivo sem linkar os arquivos para obtermos um arquivo do tipo objeto.

Para a execução do programa basta digitar:

```
./exe < caminho_até_o_arquivo_de_entrada
```

Onde "caminho-até-o-arquivo-de-entrada" pode ser: "1.in" para realizar o primeiro caso de teste e "2.in" para realizar o segundo.

2 Desenvolvimento

Para este trabalho, não precisamos implementar TADs já que os que iríamos necessitar já nos foi dado. De acordo com o pedido, e para uma melhor organização, o nosso código foi modularizado em vários arquivos: tp.c, hash.h, hash.c, indiceInvertido.c e indiceInvertido.h. O arquivo tp.c deve apenas invocar e tratar as respostas das funções e procedimentos definidos nos arquivos hash.h e indiceInvertido.h.

2.1 TAD

Para começar a resolução do problema proposto, seguimos os TADs fornecidos pelo professor.

```
1
2       typedef struct {
3           int n; // numero de documentos
4           Chave chave;
5           NomeDocumento documentos[ND];
6       } Item;
7
8       typedef Item IndiceInvertido[M];
```

Código 1: TADs

2.2 Funções

A seguir entraremos em detalhe sobre as principais funções utilizadas no programa.

2.2.1 inicia

Para começar a resolução do problema proposto, primeiro iniciamos cada chave do nosso indiceInvertido como VAZIO, utilizando a função memcpy (memory copy).

```
1
2   void inicia(IndiceInvertido indiceInvertido){
3
4       for (int i = 0; i < M; i++)
5           memcpy(indiceInvertido[i].chave, VAZIO, N);
6
7   }
```

Código 2: Função inicia

2.2.2 insereDocumentos

Essa função insere um documento baseado na chave do TAD IndiceInvertido. Com base na chave passada como parâmetro, criamos a variável indice que recebe o hash.

```
1   bool insereDocumento(IndiceInvertido indiceInvertido, Chave chave,
        NomeDocumento nomeDocumento){
2
3       int indice = h(chave, M); //indice recebe o hash baseado na chave passada
            por parametro e tamanho da tabela
4       if (chaveEhVazia(indiceInvertido[indice].chave))
5           return false;
6
7       while(strcmp(indiceInvertido[indice % M].chave, chave)){
8       /*faz a comparacao entre a chave passada por parametro e a chave presente
            na tabela do indiceInvertido. Enquanto as chaves forem iguais o indice
             vai mudando.*/
9           indice++;
10      }
11
```

```
12      strcpy(indiceInvertido[indice].documentos[indiceInvertido[indice % M].n
            ++], nomeDocumento);
13      /*quando achar um lugar vazio na tabela/lista vai copiar o nome do
            documento passado como parametro para o documento presente no
            indiceInvertido*/
14
15      return true;
16  }
```

Código 3: Função insereDocumento

2.2.3 busca

Esta função retorna o índice de uma chave passada por parâmetro presente no TAD IndiceInvertido, caso não tenha retorna -1.

```
1   int busca(IndiceInvertido indiceInvertido, Chave chave){
2       int j = 0;
3       int ini = h(chave, M);
4
5       while (strcmp(indiceInvertido[(ini + j) % M].chave, VAZIO) != 0 &&
6               strcmp(indiceInvertido[(ini + j) % M].chave, chave) != 0 && //
                    compara se a chave do parametro e a presente na tabela sao
                    diferentes
7               j < M) //enquanto j for menor que o tamanho da tabela
8           j++;
9
10      if (strcmp(indiceInvertido[(ini + j) % M].chave, chave) == 0) //se as
            chaves forem iguais retorna a posicao na tabela
11          return (ini + j) % M;
12
13      return -1; //caso nao haja correspondencia da chaves retorna -1
14  }
```

Código 4: Função busca

2.2.4 consulta

Essa função, baseada em uma ou mais chaves, retorna o nome dos documentos que contêm todas as chaves no índice invertido presente no TAD IndiceInvertido.

```
1   int consulta(IndiceInvertido indiceInvertido, Chave* chave, int numero_chaves,
        NomeDocumento* nomeDocumento){
2
3       int documentos_encontrados = 0;
4       if(!numero_chaves)
5           return 0;
6
7       Item item_base;
8       int indice_item_base = busca(indiceInvertido, chave[0]); //recebe a
            posicao da tabela que a chave se escontra
9
10      Item comparar;
11      int indice_comparar;
12
13      if(indice_item_base == -1){ //se for -1, condicao que diz que nao ha essa
            chave a tabela, retorna 0
14          return 0;
15      }
16      item_base = indiceInvertido[indice_item_base]; //se estiver na tabela,
            essa variavel do tipo Item recebe as informacoes presentes no
            indiceInvertido na posicao do indice_item_base
```

```
17
18
19      int *vetor_int1 = inicializa_um(item_base.n); //inicializa com 1 o vetor
            que vai ter tamanho do numero de documentos
20
21      for(int i = 1; i < numero_chaves; i++){
22
23          int *vetor_int2 = calloc(item_base.n, sizeof(int)); //inicia com 0 o
                vetor que vai ter tamanho do numero de documentos
24
25          indice_comparar = busca(indiceInvertido, chave[i]); //recebe a posicao
                 em que a chave[i] esta
26
27          if(indice_comparar == -1) //se for -1 passa pro proximo loop
28              continue;
29
30          comparar = indiceInvertido[indice_comparar]; //se houver, a variavel
                do tipo Item recebe o que tem no indiceInvertido nessa posicao
31
32          for(int j = 0 ; j < item_base.n; j++){
33              for(int k = 0; k < comparar.n; k++){
34                  if(!strcmp(item_base.documentos[j], comparar.documentos[k])){
                        //
35                      vetor_int2[j] = 1;
36
37                  }
38              }
39          }
40
41          for(int j = 0; j < item_base.n;j++)
42              vetor_int1[j] = vetor_int2[j];
43
44          free(vetor_int2);
45
46          if(vetor_tudo_zero(vetor_int1, item_base.n)){ //passa tudo que tem no
                vetor1 para 0
47              free(vetor_int1); //desaloca
48              return 0;
49          }
50
51      }
52
53      for(int i = 0 ; i < item_base.n; i++){
54
55          if(vetor_int1[i]){
56              strcpy(nomeDocumento[documentos_encontrados++], item_base.
                    documentos[i]); //copia o que tem no item_base para o nome do
                    documentos na posicao documentos_encontrados
57          }
58
59          //que inicialmente era 0
60      }
61
62      return documentos_encontrados; //retorna a quantidade de documentos
            encontrados que tem a chave que foi buscada
63
64  }
```

Código 5: Função consulta

2.2.5 adicionarChaves

A função adicionarChaves adicona chaves na tabela.

```
void adicionarChaves(IndiceInvertido indiceInvertido, Chave *chaves, int
    qtdChaves){

    for (int i = 0; i < qtdChaves; i++){

        bool chaveDuplicada = false; //variavel pra verificar se a chave ja
            foi inserida anteriormente

        int indice = h(chaves[i], M);

        while(!chaveEhVazia(indiceInvertido[indice % M].chave)){

            if(!strcmp(indiceInvertido[indice % M].chave, chaves[i])){
                chaveDuplicada = true;
                break;
            }

            indice++;
        }

        if(chaveDuplicada)
            continue;

        strcpy(indiceInvertido[indice % M].chave, chaves[i]);
    }

}
```

Código 6: Função adicionarChaves

2.2.6 chaveEhVazia

Nesta função a verificação para confirmar se a chave é vazia é realizada.

```
int chaveEhVazia(Chave chave){

    if(!strcmp(chave, VAZIO))
        return 1;

    return 0;
}
```

Código 7: Função chaveEhVazia

2.2.7 inicializa_um

Inicializa todas as posições do vetor com o valor 1.

```
int *inicializa_um(int n){

    int *vetor = malloc(n * sizeof(int));

    for(int i = 0; i < n; i++)
        vetor[i] = 1;

    return vetor;
}
```

Código 8: Função inicializa_um

2.2.8 vetor_tudo_zero

Para poder checar se não há mais elementos possiveis na nossa busca, vericamos se a soma do vetor de incidência é zero, ou seja só tem elementos nulos;

```
int vetor_tudo_zero(int *vetor, int n){
    int soma = 0;

    for(int i = 0; i < n; i++){
        soma += vetor[i];
        if(soma >= 1)
            return 0;
    }

    return 1;
}
```

Código 9: Função vetor_tudo_zero

2.2.9 mergeSort

Para ordenação, optamos por utilizar o método mergeSort, que faz uma chamada recursiva dela mesma no passo da divisão, para depois chamarmos a merge no passo da conquista e fazermos a ordenação.

```
void mergeSort(NomeDocumento *documentos, int l, int r){
    if (l < r)
    {
        int m = (l+r)/2;
        mergeSort(documentos,l,m);
        mergeSort(documentos, m+1, r);
        merge(documentos, l, m, r);
    }
}
```

Código 10: Função mergeSort

2.2.10 merge

Implementamos a função merge, que será responsável pela parte da conquista, em que após a divisão irá ordenar os subvetores em um maior, até que nosso vetor esteja completamente ordenado.

```
void merge(NomeDocumento *documentos, int l, int m, int r){
    int size_l = (m - l + 1);
    int size_r = r - m;
    int i, j;

    NomeDocumento *vet_l =  malloc(size_l * sizeof(NomeDocumento));
    NomeDocumento *vet_r =  malloc(size_r * sizeof(NomeDocumento));

    for (i = 0; i < size_l; i++)
    {
       strcpy(vet_l[i], documentos[i+l]);
    }
    for (j = 0; j < size_r; j++)
    {
        strcpy(vet_r[j], documentos[m + j + 1]);
    }
    i = 0; j = 0;
    for (int k = l; k <= r; k++)
    {
        if (i == size_l)
        {
            strcpy(documentos[k], vet_r[j++]);
```

```
            //documentos[k] = vet_r[j++];
        }
        else if (j == size_r)
        {
            strcpy(documentos[k], vet_l[i++]);
            //documentos[k] = vet_l[i++];
        }
        else if (compare(vet_l[i],vet_r[j]))
        {
            strcpy(documentos[k], vet_l[i++]);
            //documentos[k] = vet_l[i++];
        }
        else{
            strcpy(documentos[k], vet_r[j++]);
            //documentos[k] = vet_r[j++];
        }
    }
    free(vet_l);
    free(vet_r);
}
```

Código 11: Função merge

2.2.11 compare

Nesta função ocorre a comparação dos nomes dos documentos.

```
int compare(const NomeDocumento documento1, const NomeDocumento documento2) {

    if(strcmp(documento1, documento2) < 0)
        return 1;
    else
        return 0;
}
```

Código 12: Função compare

2.2.12 h

Aqui se encontra a função hash, onde utilizamos a variável m passada por parâmetro que indica o tamanho da tabela hash. Ela retorna o resto da soma pelo tamanho da tabela.

```
int h(char * chave, int m) {
    float p[] = {0.8326030060567271, 0.3224428884580177,
                 0.6964223353369197, 0.1966079596929834,
                 0.8949283476433433, 0.4587297824155836,
                 0.5100785238948532, 0.05356055934904358,
                 0.9157270141062215, 0.7278472432221632};
    int tamP = 10;
    unsigned int soma = 0;
    for (int i=0;i<strlen ( chave );i++)
        soma += (unsigned int) chave [i] * p[i % tamP];
    return soma % m; //
}
```

Código 13: Função h

2.2.13 pegarChaves

Essa função pega a entrada e divide em documento e várias chaves e retorna a quantidade de chaves. O nome do documento não está sendo salvo no vetor de chaves.

```c
int pegarChaves(Chave* chaves) {
    int i = 0;
    char* token;
    char str[MAX_STR];

    // tokenizacao da string original, divide em strings delimitadas por
       espaco em branco
    fgets (str, MAX_STR, stdin);
    str[strcspn(str, "\n")] = 0;

    token = strtok(str, " ");

    while (token != NULL) {
        strcpy(chaves[i++], token);
        //inserir o item na lista adequada
        token = strtok(NULL, " ");
    }
    return i;
}
```

Código 14: Função pegarChaves

2.3 main

Na função main, além de utilizarmos o scanf para obter o nobj e npontos, invocamos as funções necessárias para a realização dos procedimentos, sendo eles: a alocação dos objetos, a leitura deles, os cálculos necessários, a ordenação da lista, sua impressão e por último, desalocação.

```
1
2  #include <stdio.h>
3  #include "indiceInvertido.h"
4  #include "hash.h"
5
6
7  int main() {
8
9      int numeroDocumentos;
10     scanf("%d", &numeroDocumentos);
11     Chave chaves[NN];
12     IndiceInvertido indiceInvertido;
13     int numeroChaves;
14     inicia(indiceInvertido);
15
16
17     for(int i = 0; i < numeroDocumentos; i++){
18
19         NomeDocumento nomeDocumento;
20         scanf("%s ", nomeDocumento);
21
22         numeroChaves = pegarChaves(chaves);
23         adicionarChaves(indiceInvertido, chaves, numeroChaves);
24
25
26         for(int j = 0; j < numeroChaves; j++)
27             insereDocumento(indiceInvertido, chaves[j], nomeDocumento);
28
29     }
30
31
32     char opcao;
33     scanf("%c", &opcao);
34
35     int documentos_encontrados;
36     NomeDocumento documentos[ND];
37
38     switch (opcao){
39         case 'I':
40             imprime(indiceInvertido);
41             break;
42
43         case 'B':
44
45
46             numeroChaves = pegarChaves(chaves);
47             documentos_encontrados = consulta(indiceInvertido, chaves,
                   numeroChaves,documentos);
48
49             mergeSort(documentos, 0, documentos_encontrados-1);
50             // printf("---------------\n");
51             if( documentos_encontrados == 0){
52                 printf("none\n");
53             }
54
55             else{
```

```
                for(int i = 0; i < documentos_encontrados; i++){
                    printf("%s\n", documentos[i]);
                }
            // printf("---------------\n");
            }

            break;

        }

    return 0;
}
```

Código 15: Função main

3 Impressões Gerais

Com as funções já pré definidas foi muito mais fácil construir a lógica para o desenvolvimento modular do código. O nosso grupo então se reuniu e pensou coletivamente sobre a ordem de execução das funções e suas utilidades. Outro conhecimento já adquirido anteriormente em outros trabalhos e posto em prática foi o uso do mergeSort, método de ordenação, para a solução de parte do nosso problema. Houve também o desenvolvimento de um código bem modularizado, com uma excelente ajuda das instruções contidas no documento que nos foi disponibilizado como exemplo.

4 Análise

Após o desenvolvimento do programa, a primeira análise feita foi atráves dos casos de teste disponibilizados na página do trabalho prático do run.codes, com simples exemplos de entrada e saída, executamos o programa com um dos exemplos de entrada e assim, foi possível fazer uma simples análise se o programa se comportava corretamente. Obtivemos algumas dificuldades que serão abordadas na conclusão. As próximas realizações de testes apresentaram resultados iguais ao exemplo de saída disponibilizado. Depois dos testes iniciais para verificar um funcionamento inicial do programa, utilizamos o valgrind, uma ferramenta de análise dinâmica de código para conferir se há algum memory leak ou warning referente a manipulação de memória.

5 Conclusão

Com este trabalho ampliamos os nossos conhecimentos referente a tabela Hash e como aplicá-lo para encontrar a solução de um problema. Como uma dificuldade inicial, tivemos a construção da lógica para a realização da busca de arquivos, em que deveriamos pensar em como realizar essa bsuca. A solução que decidimos utilizar foi um vetor de inciência, o numero máximo de elementos na interseção de dois conjuntos é o número máximo de elementos. Assim usamos como base a primeira chave passada na busca e a partir dela vamos verificando a incidência dos documentos em cada chave. Outra dificuldade foi entender o conceito de Hash, que é intrisicamente um conceito complicado de se entender.