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

Índice Invertido

BCC202 - Estrutura de Dados I

Felipe Braz Marques
Matheus Peixoto Ribeiro Vieira
Pedro Henrique Rabelo Leão de Oliveira

Professor: Pedro Henrique Lopes Silva

Ouro Preto
15 de março de 2023

Sumário

Lista de Figuras

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar um código em C para realizar a pesquisa de palavras chaves através de um indíce invertido usando uma tabela hash.

1.1 Especificações do problema

Utilizando a liguagem de programação C, é necessário armazenar os nomes dos documentos e suas palavras chaves em uma tabela hash. E para realizar a pesquisa dessas palavras chaves, fica menos custoso usando um índice invertido. O índice invertido pode ser explicado como tendo duas partes principais: um vocabulário, contendo as diferentes palavras-chave, e para cada termo, uma lista que armazena os identificadores dos documentos que o contém.

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: Overleaf LaTeX. [1]

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *GCC*: versão 11.3.0.
- *Valgrind*: versão 3.18.1.

1.4 Especificações da máquina

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

- Processador: Intel i5-9300H.
- Memória RAM: 16Gb.
- Sistema Operacional: Ubuntu 22.04.1 LTS.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

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

Usou-se para a compilação as seguintes opções:

- *-Wall*: para mostrar todos os possível *warnings* do código.

Para a execução do programa basta digitar:

```
./exe caminho_até_o_arquivo_de_entrada
```

[1] Disponível em https://www.overleaf.com/

2 Desenvolvimento

2.1 indiceInvertido.h

Primeiramente, o grupo começou a desenvolver a interface do TAD no arquivo indiceInvertido.h, a fim de ter, desde o princípio, bem estruturado as principais funções do programa, sendo elas "inicia", "busca", "consulta", "imprimeIndiceInvertido"e "imprimeDocumentos", "sort", além da Struct "Item", que possui um inteiro "n"que armazena o número de documentos, um tipo Chave "chave", onde o tipo Chave é definido como um vetor de char com 21 posições. E por fim um vetor do tipo NomeDocumento com no máximo 100 documentos. O tipo NomeDocumento também é definido como um vetor de 51 posições, que é o tamanho do nome do documento. E para ordenar usamos o método MergeSort.

Vale ressaltar que há a presença de um define nomeado como ANALISE_RELATORIO que, quando definido permite a exibição de dados para a análise dos resultados, como o número de colisões e o tempo de execução do programa.

2.2 indiciInvertido.c

Dentro do arquivo indiceInvertido.c foi desenvolvido o corpo das funções prototipadas no indiceInvertido.h

A função **"inicia"** recebe, por parâmetro, indiceInvertido, nela é inicializado todas as chaves do indiceInvertido como VAZIO, e o inteiro n como 0.

A função **"insereDocumento"** será a responsável por encontrar a posição onde cada elemento irá ficar salvo na tabela *hash*. Ela recebe como parâmentro o IndiceInvertido, a Chave o NomeDocumento. Onde, primeiramente é feita uma busca para saber se a chave já existe na tabela. Caso exita, o nome do documento será adicionado no final do vetor daquela posição.

Contudo, casa a chave ainda não esteja na tabela, será chamada a função **h** para encontrar a posição onde ela deverá ficar. Caso o local não esteja vazio, e ocorrá uma colisão, tentará adicionar no próximo espaço livre.

A função **"busca"** recebe, por parâmetro, um indiceInvertido e uma chave. A partir da posição inicial que é possível a chave buscada estar presente no índice invertido, é feito um while para varrer o vetor, parando caso encontre a posição do vetor vazia, o que significa que aquela chave não existe no índice invertido, quando encontra a chave buscada, ou então quando já tiver varrido toda a tabela e ainda não tiver encontrado a chave. Dessa forma, caso a busca tenha tido sucesso, é retornado o índice daquela chave no vetor de índice invertido, caso contrário é retornado -1. Sendo que o código é muito similar ao que foi passado em sala.

A função **"removeDocumento"** é uma função auxiliar da função de consulta, que recebendo um vetor de nome de documentos, a quantidade de documentos e a posição para remover, ele copia todos os próximos dados e apaga a posição atual.

A função **"consulta"** recebe o IndiceInvertido, um conjunto de chaves, a quantidade de chaves, o vetor que receberá o nome dos documentos que possuem as chaves buscadas e um ponteiro com a quantidade dos mesmos. Primeiramente, é criado um vetor de inteiro que armazenará os índices no indiceInvertido de todas as chaves procuradas, por meio de chamadas da função busca. Caso seja retornado -1 em alguma chamada da função busca, a função consulta será interrompida e já retornará 0, tendo em vista que alguma das chaves procuradas não existe na tabela de índice invertido, e, portanto, não existirá nenhum documento que possuirá todas as chaves.

Caso todas as chaves forem encontradas no vetor, utilizando os índices das chaves no indiceInvertido, todos os documentos que possuem a primeira chave procurada é copiado para o vetor de documentos.

Posteriormente, é feita uma verificação para checar se os documentos também possuem as próximas palavras. Assim, caso não possua, ele será removido do vetor de nomes de documentos. E, chegando ao fim da função, será retornado um inteiro que indica se há documentos ou não para serem exibidos. Caso

tenha, retornará 1, caso contrário, zero. E é retornado também, os documentos à serem impressos, através desse vetor de nomes de documentos recebido como parâmetro.

```
int consulta(IndiceInvertido indiceInvertido, Chave *chave, int n,
    NomeDocumento* documento, int *contDocumentos){
    int *indicesChaves = (int *) malloc(n * sizeof(int));
    for(int i = 0; i < n; i++){
        indicesChaves[i] = busca(indiceInvertido, chave[i]);
        if(indicesChaves[i] == -1){
            free(indicesChaves);
            return 0;
        }
    }
    //copiando o nome dos documentos que possuem a primeira palavra(chave)
       buscada
    *contDocumentos = indiceInvertido[indicesChaves[0]].n;
    for(int i=0; i < *contDocumentos; i++){
        strcpy(documento[i], indiceInvertido[indicesChaves[0]].documentos[i]);
    }
    //verificando se os documentos que possuem a primeira palavra(chave),
    //possuem as proximas palavras tambem
    //caso nao possuam, removo ela do vetor de documentos que serao impressos
    bool removerDoc;
    int numInicialDocumentos = *contDocumentos;
    for(int i=1; i < n; i++){ //Passando por todas as (chaves) buscadas
        for(int j=0; j < numInicialDocumentos; j++){ // Passa pelos documentos
             do vetor de documentos que serao impressos
            removerDoc = true;
            for(int k=0; k < indiceInvertido[indicesChaves[i]].n; k++){ //
               Passa por todos os nomes de documentos que possuem x palavra(
               chave)
                if(strcmp(documento[j], indiceInvertido[indicesChaves[i]].
                    documentos[k]) == 0){
                    removerDoc = false;
                    break;
                }
            }
            if(removerDoc)
                removeDocumento(documento, contDocumentos, j);
        }
    }
    free(indicesChaves);
    return *contDocumentos > 0 ? 1 : 0;
}
```

Código 1: Código da função consulta

A função **"leEntrada"** recebe como parâmetro um vetor onde os dados de entrada serão salvos, e por referência uma variável que armazenará o número de documentos. Como o nome da função é bem sugestivo, a função lê todos os dados de entrada, através de um for que roda de 0 até o número de documentos, armazenando o nome do documento e suas palavras chaves no vetor. Dentro desse for, em cada linha de entrada, é separado o nome do documento de suas palavras chaves através do strtok e feita chamadas da função insereDocumento, para que seja feita a inserção no vetor indiceInvertido de cada palavra com o documento que a possui.

A função **"leOpcao"** recebe por parametro o vetor onde os dados estão salvos, e também o numero de documentos. Esta função lê o que o usuário deseja. Se for "I", imprime o índice invertido. E, se a opção "B", faz a busca das palavras que o usuário passou, e imprime quais arquivos contém todas palavras. Se não for encontrado nenhum documento que tenha todas as chaves, é impresso "none".

A função **"executaBuscaDoUsuario"** recebe como parâmetros o vetor indiceInvertido, o número de documentos e um vetor de char que possue todas as palavras que o usuário deseja buscar. É feito uma

chamada da função copiaPalavrasBuscadas, possuindo, então, um vetor de chaves, onde cada posição será uma chave a ser procurada no indiceInvertido. Posteriormente, é criado um vetor de documentos, no qual ficará salvo os documentos a serem impressos, ou seja, os documentos que possuem todas as palavras do vetor palavrasChave. Depois, a função consulta é chamada, onde será feita a busca por essas palavras, e será retornado por meio do vetor documentos passado como parâmetro os documentos. Caso a função consulta retorne 1, o vetor documentos será ordenado em ordem alfabética, e depois será impresso na saída do programa. Caso seja retornado 0, é impresso na saída "none", significando que não existe nenhum documento que possue todas as palavras procuradas pelo usuário.

```
1  void executaBuscaDoUsuario(IndiceInvertido indiceInvertido, int nDocumentos,
       char *palavrasBuscadas){
2      //Salvando todas as palavras de pesquisa em um vetor
3      Chave palavrasChave[100];
4      int qtdPalavrasChave = 0;
5      copiaPalavrasBuscadas(palavrasChave, &qtdPalavrasChave, palavrasBuscadas);
6      NomeDocumento *documentos = (NomeDocumento *) malloc(nDocumentos * sizeof(
           NomeDocumento));
7      int contDocumentos;
8      if(consulta(indiceInvertido, palavrasChave, qtdPalavrasChave, documentos,
           &contDocumentos)){
9          sort(documentos, contDocumentos);
10         imprimeDocumentos(documentos, contDocumentos);
11     }
12     else
13         printf("none\n");
14     free(documentos);
15 }
```

Código 2: Código da função executaBuscaDoUsuario

A função **"copiaPalavrasBuscadas"** recebe por parametro um vetor que receberá as chaves, um ponteiro para inteiro da quantidade de palavras-chave e uma string com todas as palavras, obtida a partir do fgets. Asism, a função utiliza o strtok para separar pelos espaços a string e salva todos os itens na posição da qquantidade de palavras chave menos um no vetor palavrasChave.

A função **"sort"** irá receber o vetor de documentos e a quantidade de itens que esse arranjo possui e irá ordená-lo por meio do MergeSort, que foi escolhido por ter um custo de complexidade O(nlog(n)) para todos os seus casos, até mesmo para o pior. Assim, a sua ordenação será pela ordem lexicográfica dos documentos.

2.3 tp.c

Já o arquivo tp.c contém a inclusão do indiceInvertido.h e a main, na qual está organizado a execução do programa.

Na main, ainda temos a incialização das váriaveis de controle de tempo de execução e quantidade de colisões. Além das variáveis de indiceInvertido e número de documentos, indispensáveis para o funcionamento do programa.

3 Análise dos Resultados

Com o fim do trabalho prático, conseguimos obter os resultados esperados com os casos de teste disponibilizados, como pode-se observar nas figuras 1 a 5.

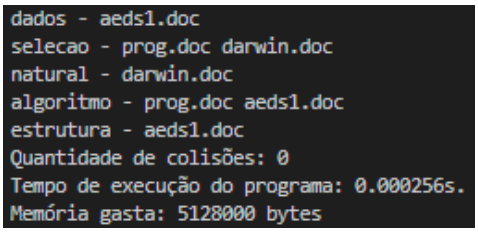

Figura 1: Execução do caso de teste 1

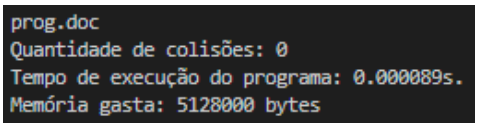

Figura 2: Execução do caso de teste 2

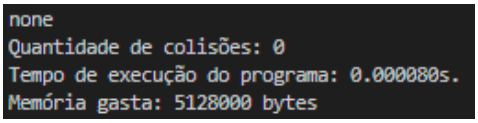

Figura 3: Execução do caso de teste 3

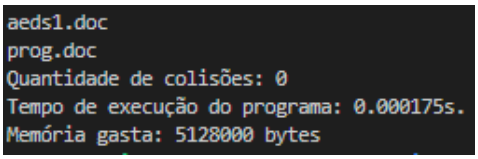

Figura 4: Execução do caso de teste 4

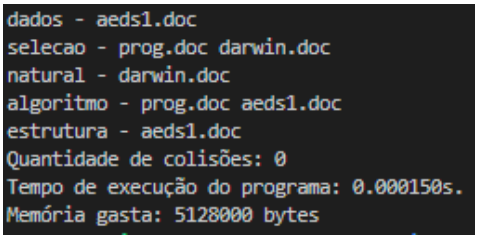

Figura 5: Execução do caso de teste 5

Observa-se que, em todos os casos, o número de colisões foi zero. Isso pode ser entendido devido à quantidade de espaços que a tabela possui, que foi definido como 1000. Dessa forma, pelo fato dos casos de teste terem poucas entradas, a chance de ocorrer uma colisão é muito baixa.

Em relação à memória gasta, destaca-se o grande uso do vetor IndiceInvertido, que 1000 posições, sendo este responsável pela maior parte do uso de memória do código, gerando um gasto de 5128000 bytes.

Por fim, analisando o tempo de execução, mesmo que seja um valor muito volátil, pode-se observar que o primeiro caso de teste foi o que mais demorou, levando 2,56µ segundos. Assim, fica perceptível

a volatidade do tempo de execução, uma vez que o caso de teste 1 é idêntico ao 5, mas o último possui um tempo de 1,50µ segundos.

Percebe-se também que o caso de teste três foi o mais rápido, levando somente cerca de 0,8µ segundos, uma diferença de quase um microsegundo mais rápido que o caso de tete 2, que levou 0,89µ segundos.

Vale ressaltar que estes resultados foram obtidos em um computador com processador Intel i5-9300H, 16Gb de memória RAM e Ubuntu versão 22.04.1 LTS

4 Impressões Gerais e Considerações Finais

Fazndo uma análise geral sobre o trabalho, consideramos que ele é bem completo por abrager uma grande parte do conteúdo do semestre, como por exemplo ordenação, tabelas hash, alocação dinâmica e TAD.

A parte de construção do código no sentido da sintaxe não tivemos dificuladades. Porém a parte que nos desafiou foi entender a lógica por trás de todo o tp em si. A partir do momento que entendemos a forma de fazer, foi bem tranquilo. Tivemos um pequeno problema com a função memcpy, que estava causando uma leitura de posição inválida de memória, perceptível através do valgrind. Para contornar esse problema tivemos que usar o strcpy para atribuir vazio à chave.