

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
17 de março de 2023

Sumário

1	Introdução	1
1.1	Especificações do problema	1
1.2	Considerações iniciais	1
1.3	Ferramentas utilizadas	1
1.4	Especificações da máquina	1
1.5	Instruções de compilação e execução	1
2	Desenvolvimento	2
2.1	indiceInvertido.h	2
2.2	indiciInvertido.c	2
2.3	tp.c	4
3	Análise dos Resultados	5
4	Impressões Gerais e Considerações Finais	7

Lista de Figuras

1	Execução do caso de teste 1	5
2	Execução do caso de teste 2	5
3	Execução do caso de teste 3	5
4	Execução do caso de teste 4	5
5	Execução do caso de teste 5	5

Lista de Códigos Fonte

1	Código da função consulta	3
2	Código da função executaBuscaDoUsuario	4

1 Introdução

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

1.1 Especificações do problema

Utilizando a linguagem 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 L^AT_EX.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
```

¹Disponível 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 parâmetro o `IndiceInvertido`, a Chave o `NomeDocumento`. Onde, primeiramente é feita uma busca para saber se a chave já existe na tabela. Caso exista, o nome do documento será adicionado no final do vetor daquela posição.

Contudo, caso 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 ocorrerá 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.

```
1 int consulta(IndiceInvertido indiceInvertido, Chave *chave, int n,
2 NomeDocumento* documento, int *contDocumentos){
3     int *indicesChaves = (int *) malloc(n * sizeof(int));
4     for(int i = 0; i < n; i++){
5         indicesChaves[i] = busca(indiceInvertido, chave[i]);
6         if(indicesChaves[i] == -1){
7             free(indicesChaves);
8             return 0;
9         }
10    }
11    //copiando o nome dos documentos que possuem a primeira palavra(chave)
12    buscada
13    *contDocumentos = indiceInvertido[indicesChaves[0]].n;
14    for(int i=0; i < *contDocumentos; i++){
15        strcpy(documento[i], indiceInvertido[indicesChaves[0]].documentos[i]);
16    }
17    //verificando se os documentos que possuem a primeira palavra(chave),
18    //possuem as proximas palavras tambem
19    //caso nao possuam, removo ela do vetor de documentos que serao impressos
20    bool removerDoc;
21    int numInicialDocumentos = *contDocumentos;
22    for(int i=1; i < n; i++){ //Passando por todas as (chaves) buscadas
23        for(int j=0; j < numInicialDocumentos; j++){ // Passa pelos documentos
24            do vetor de documentos que serao impressos
25            removerDoc = true;
26            for(int k=0; k < indiceInvertido[indicesChaves[i]].n; k++){ //
27                Passa por todos os nomes de documentos que possuem x palavra(
28                chave)
29                if(strcmp(documento[j], indiceInvertido[indicesChaves[i]].
30                    documentos[k]) == 0){
31                    removerDoc = false;
32                    break;
33                }
34            }
35            if(removerDoc)
36                removeDocumento(documento, contDocumentos, j);
37        }
38    }
39    free(indicesChaves);
40    return *contDocumentos > 0 ? 1 : 0;
41 }
```

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 possui 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 possui 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`. Assim, a função utiliza o `strtok` para separar pelos espaços a string e salva todos os itens na posição da quantidade 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 $O(n \log(n))$ 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 inicialização das variáveis 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.

```
dados - aeds1.doc
selecao - prog.doc darwin.doc
natural - darwin.doc
algoritmo - prog.doc aeds1.doc
estrutura - aeds1.doc

===Informações da execução===
Quantidade de colisões: 0
Tempo de execução do programa: 0.000338s.
Total de memória: 5149051 bytes
```

Figura 1: Execução do caso de teste 1

```
prog.doc

===Informações da execução===
Quantidade de colisões: 0
Tempo de execução do programa: 0.000252s.
Total de memória: 5151312 bytes
```

Figura 2: Execução do caso de teste 2

```
none

===Informações da execução===
Quantidade de colisões: 0
Tempo de execução do programa: 0.000257s.
Total de memória: 5151312 bytes
```

Figura 3: Execução do caso de teste 3

```
aeds1.doc
prog.doc

===Informações da execução===
Quantidade de colisões: 0
Tempo de execução do programa: 0.000227s.
Total de memória: 5151410 bytes
```

Figura 4: Execução do caso de teste 4

```
dados - aeds1.doc
selecao - prog.doc darwin.doc
natural - darwin.doc
algoritmo - prog.doc aeds1.doc
estrutura - aeds1.doc

===Informações da execução===
Quantidade de colisões: 0
Tempo de execução do programa: 0.000161s.
Total de memória: 5149051 bytes
```

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 tem 1000 posições, sendo este responsável pela maior parte do uso de memória do código, gerando um gasto de 5128000 bytes só da parte dele.

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 338 μ segundos. Assim, fica perceptível a volatilidade do tempo de execução, uma vez que o caso de teste 1 é idêntico ao 5, mas o último possui um tempo de 161 μ 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

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

Na parte de construção do código, no sentido da sintaxe, não tivemos dificuldades. Porém, a parte que nos desafiou foi entender a lógica por trás de todo o `tp` em si, desde o momento em que o problema do programa a ser desenvolvido nos foi passado. 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.