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

Trabalho prático 1

BCC202 - Estrutura de Dados

João Victor Ramalho, Maria Eduarda Bessa, Gabriel Henrique Rocha
Professor: Pedro Henrique Lopes Silva

Ouro Preto
22 de março de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Para este trabalho foi necessário implementar um código em C utilizando a Tabela Hash para auxiliar na busca em um banco de dados textual e um relatório referente ao que foi desenvolvido. O algoritmo desenvolvido é um programa que permite ler documentos e palavras que são associadas a cada documento. A partir disso, essas informações são associadas pela hash para conseguirmos efetuar buscas com eficiência.

1.1 Especificações do problema

Em grandes coleções de informações, a comparação direta entre elas é computacionalmente cara. Por esse motivo, é necessário traçar uma estratégia para tornar essa busca mais eficiente.

Utilizar a Tabela Hash é uma boa opção pois, a partir de um único valor gerado a partir de um conjunto de caracteres, podemos associar uma posição em um tabela a um conjunto de informações de um banco de dados.

Porém, podemos encontrar barreiras para esse armazenamento, uma vez que há a possibilidade de ocorrerem colisões entre os valores gerados pelo conjunto de caracteres, resultando em conflitos na tabela. Assim, dois ou mais conjuntos diferentes de caracteres tem a possibilidade de gerar o mesmo valor de hash. Isso pode levar a resultados imprecisos na pesquisa, a menos que sejam implementadas técnicas para lidar com as colisões. Nesse trabalho utilizamos a técnica de *Endereçamento Aberto* para tratar isso.

1.2 Considerações iniciais

1.2.1 Ferramentas utilizadas durante a criação deste projeto

- Ambiente de desenvolvimento do código fonte: Visual Studio Code e GitHub. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.2.2 Ferramentas utilizadas para testar a implementação deste projeto

- -*Wall*: indica potenciais problemas ou erros no código fonte do programa;
- *Valgrind*: ferramenta para análise de memória e detecção de erros em programas.

1.3 Especificações da máquina

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

- Processador: i7 (11 geração).
- Memória RAM: 16Gb.
- Sistema Operacional: Linux Ubuntu.

1.4 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

[1] GitHub está disponível em https://www.github.com
[2] Disponível em https://www.overleaf.com/

Compilando o projeto

```
gcc tp.c -c
gcc indiceInvertido.c -c
gcc hash.c -c
gcc tp.o indiceInvertido.o hash.o -o exe
rm -r *.o
```

Para a execução do programa basta digitar:

```
./exe
número de documentos que serão inseridos
nomeDoDocumento Palavras-chave1 Palavras-chave2 Palavras-chave3 ...
.
.
.
Opção O
```

Sendo possível inserir até 100 documentos, os quais podem ter até 1000 palavras-chave de no máximo 20 caracteres associadas

A opção O é a opção escolhida pelo usuário do que o programa deve realizar, sendo possivel digitar 'I' caso queira imprimir todas as palavras-chave com os documentos que apresentam ela como associação ou digitar 'B', e em seguida colocar a(s) palavra(as) que queira buscar para que seja impresso todos os documentos relacionados a elas.

No caso de escolha da opção B, se digitado mais de uma palavra é apresentado apenas os documentos que tenham todas as palavras digitadas como suas palavras-chave. Além disso, se caso houver mais de um documento associados às palavras, esses são impressos em ordem alfabética.

Exemplos de entrada e saída com a opção 'I':

Entrada

```
3
ed1.txt TAD recursividade filas algoritmos
pesquisa.txt sequencial binaria arvores hash
ordenacao.txt mergeSort recursividade algoritmos heapSort
I
```

Saída

```
TAD - ed1.txt
hash - pesquisa.txt
filas - ed1.txt
heapSort - ordenacao.txt
binaria - pesquisa.txt
arvores - pesquisa.txt
mergeSort - ordenacao.txt
sequencial - pesquisa.txt
algoritmos - ed1.txt ordenacao.txt
recursividade - ed1.txt ordenacao.txt
```

Exemplos de entrada e saída com a opção 'B':

Entrada
3
ed1.txt TAD recursividade filas algoritmos
pesquisa.txt sequencial binaria arvores hash
ordenacao.txt mergeSort recursividade algoritmos heapSort
B

Saída
ed1.txt
ordenacao.txt

2 Desenvolvimento

2.1 Tipo Abstrato de Dados (TAD)

Para atingir nosso objetivo, implementamos um Tipo Abstrato de Dados (TAD) IndiceInvertido para representar um índice invertido implementado com Hash.

```
typedef struct {
    int n; // numero de documentos
    Chave chave;
    NomeDocumento documentos[ND];
} Item;

typedef Item IndiceInvertido[M]; // M    o tamanho da hash
```

Código 1: TAD IndiceInvertido

Então, para cada palavra é calculada a sua posição n na tabela hash. A partir disso, adicionamos na posição n da hash a palavra, seus documentos vinculados e a quantidade de documentos vinculados.

2.2 As funções

2.2.1 inicia()

Essa função é utilizada para inicializar todas as chaves das posições da tabela hash como VAZIO e com o número de documentos igual a zero.

2.2.2 h()

Essa função é utilizada para que a partir de um conjunto de caracteres (nesse caso, a palvra-chave) seja calculado uma posição da hash para ser inserida.

2.2.3 inserePalavras()

Essa função é utilizada para lê os documentos e palavras inseridas pelo usuário e fazer a chaamada das funções para armazena-los.

```
void inserePalavras(IndiceInvertido indiceInvertido, int
    numeroDocumentoInseridos){
    Chave aux[MAX_STR];
    char nomeDocumento[D];
    for(int i = 0; i < numeroDocumentoInseridos; i++){
```

```
        int numeroPalavrasInseridas = pegarChaves(aux); //aux[0]    o
            documento e aux[1]-aux[numeroPalavrasInseridas - 1] s o as
            palavras-chave
        strcpy(nomeDocumento, aux[0]);
        for(int j = 1; j < numeroPalavrasInseridas; j++){
            indiceInvertidoInsere(indiceInvertido, aux[j], nomeDocumento);
        }
    }
}
```

Código 2: Função inserePalavras()

2.2.4 indiceInvertidoPesquisa()

Essa função é utilizada para verificar se uma palavra está presente na hash. A partir da posição gerada pela função h(), ela verifica na posição e em suas próximas se a chave está presente. Ela retorna -1 se encontrar uma célula vazia antes de achar a palavra ou se passar por todas as celulas da tabela e não achar uma celula vazia ou a palavra. Encontrando a palavra, ele retorna seu índice.

2.2.5 indiceInvertidoInsere()

Essa função é utilizada para inserir palavras e documentos a tabela.

A partir dela, que utilizamos a técnica de endereçamento aberto, pois ela analisa se o indice retornado pela função h() está vazia para que se insira a novaa palavra. Porém há a possibilidade de este indice já estar oucupado por outra palavra, e assim, será necessário, verificar as próximas posições até que seja encontrada uma vazia para que aconteça a inserção.

Se a palavra já estiver presente na tabela, ela apenas adicionará o documento novo ao vetor de documentos da palavra. Se não estiver presente na tabela, a partir do indice retornado pela função h(), ela irá verificar essa posição e as proximas até que seja encontrada alguma que está disponível (ou seja, está vazia). Após inserir a nova palavra, também adiciona o documento associado a ela ao vetor de documentos desta palavra.

2.2.6 interseção()

Essa função recebe dois vetores e retorna um vetor com os elementos em comum entre os dois vetores.

```
int indiceInvertidoInsere(IndiceInvertido indiceInvertido, Chave chave,
    NomeDocumento documento){
    int i = indiceInvertidoPesquisa(indiceInvertido, chave);
    int chaveJaExiste = 0;
    if (i >= 0)
        chaveJaExiste = 1;  // chave j   existe na hash
    int j = 0;
    int ini = h(chave);
    if (chaveJaExiste){
        strcpy(indiceInvertido[i].documentos[indiceInvertido[i].n], documento)
            ;
        indiceInvertido[i].n++;
        return 1;
    }
    while(strcmp(indiceInvertido[(ini + j) % M].chave, VAZIO) != 0 && j < M ){
        j++;
    }if(j < M && !chaveJaExiste){
        strcpy(indiceInvertido[(ini + j) % M].chave, chave);
        strcpy(indiceInvertido[(ini + j) % M].documentos[indiceInvertido[(ini
            + j) % M].n], documento);
        indiceInvertido[(ini + j) % M].n++;
    }
    return 1;
```

```
}
```

Código 3: Função indiceInvertidoInsere()

2.2.7 busca()

Essa função verifica quais são os documentos que tem associação com todas as palavras inseridas. Para isso, se for inserida mais de uma palavra para a busca, ela utiliza a função interseção() para achar os documentos que apresentam todas as palavras.

Nessa função, utilizamos a variável "tamanho"para controlar o tamanho do array de documentos e não fazer comparações desnecessárias.

```
NomeDocumento * busca(IndiceInvertido indiceInvertido, Chave *aux, int
    numPalavrasBuscadas, int *tamanho){
    int posicao = indiceInvertidoPesquisa(indiceInvertido, aux[1]);
    (*tamanho) = indiceInvertido[posicao].n;
    NomeDocumento *auxArray;
    NomeDocumento *documentosEmComum = malloc(sizeof(NomeDocumento) * (*
        tamanho));
    for(int i = 0; i < (*tamanho); i++){
        strcpy(documentosEmComum[i], indiceInvertido[posicao].documentos[i]);
    }
    if (numPalavrasBuscadas == 2){
        return documentosEmComum;
    }else{
        for(int i = 2; i < numPalavrasBuscadas; i++){
            posicao = indiceInvertidoPesquisa(indiceInvertido, aux[i]);
            auxArray = documentosEmComum;
            documentosEmComum = intersecao(documentosEmComum, indiceInvertido[
                posicao].documentos, tamanho, indiceInvertido[posicao].n);
            free(auxArray);
        }
    }
    return documentosEmComum;
}
```

Código 4: Função busca()

2.2.8 Outras

Além dessas, temos outras funções auxiliares. Como a imprimeBusca() e a imprimeIndiceInvertido() que emprime dados para o usuário. Ademais, temos funções para a implementação do MergeSort para ordenação do vetor de documentos.

3 Impressões Gerais

Para implementação deste trabalho, divimos o processo em duas partes:

- 1° parte: Implementar a tabela Hash, inserir os dados e implementar a opção 'I' no programa;
- 2° parte: Implemetar a opção de busca no programa;

Tivemos algumas dificuldades na implementação do código por ser nosso primeiro contato com a Tabela Hash. Porém, o trio desenvolveu bem durante os encontros e conseguiu entender todos os pontos para assim desenvolver o programa em conjunto com bastante tranquilidade.

4 Análise

Para todas as análises de desempenho a seguir, utilizamos sempre o mesmo caso de teste para rodar:

```
3
prog.doc algoritmo selecao
aeds1.doc algoritmo estrutura dados
darwin.doc selecao natural
B algoritmo
```

4.1 Vazamento de memória e acesso a posições inválidas de memória

Ao finalizar nosso programa, rodamos o Valgrind para verificar ocasionais erros no código que poderiam estar resultando em vazamentos de memória e acesso a posições inválidas de memória. Porém, ficamos satisfeitos pois tivemos o resultado:

```
== xxxxx == ERROR SUMMARY : 0 errors from 0 contexts ( suppressed : 0 from 0)
```

4.2 Consumo de memória

Em relação o consumo de memória, o Valgrind indicou:

```
total heap usage: 5 allocs, 5 frees, 5,324 bytes allocated
```

Pela nossa implementação da opção 'B' temos um custo de memória maior que é justificado por procurar reduzir comparações desnecessárias. Assim, temos algumas vantagens:

- Melhor desempenho: Ao fazer menos comparações desnecessárias, o programa pode executar mais rapidamente e eficientemente. Isso pode ser especialmente importante para este programa pois podemos lidar com grandes quantidades de dados.
- Menor uso de CPU: Com menos comparações desnecessárias, permitimos que outros processos sejam executados simultaneamente sem afetar significativamente o desempenho do sistema.

4.3 Tempo de execução

Para esse caso de teste nosso programa levou 0.0054 seguntos para execução.

4.4 Colisões

Utilizamos contadores para calcular o número de colisões ocorridas. Nesse teste, não ocorreu nenhuma colisão. Isso se deve ao tamanho grande da tabela em comparação com a quantidade de índices inseridos.

Nos interessamos em saber o número de colisões caso houvesse uma quantidade próxima entre o tamanho da tabela e a quantidade de índices inseridos. Por isso, reduzimos o tamanho da tabela para 10 e tivemos 1 conflito entre os 5 índices invertidos inseridos. Dessa forma, podemos observar que se não houvesse o tratamento de colisões em nosso programa, teríamos problemas graves na estrutura da tabela hash e na eficiência do algoritmo de hash quando inserida uma grande quantidade de dados.

4.5 Método de Ordenação

Fizemos a escolha do Merge Sort como nosso método de ordenação neste trabalho por diversos motivos:

- Desempenho consistente;
- Fácil implementação;
- Excelente adequação para grandes conjuntos de dados.

5 Conclusão

Conclui-se então que o trabalho prático foi de suma importância ao desenvolvimento do grupo na disciplina por abordar tópicos como algoritmos de pesquisa, tabela hash, interseção entre vetores, método de ordenação, entre outros.

O trabalho foi realizado através de encontros com o grupo e desenvolvimento com a ajuda do Git e GitHub. O relatório foi realizado com práticidade após a nossa progressão no aprendizado do uso do Overleaf.