UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA
CIÊNCIA DA COMPUTAÇÃO
ESTRUTURAS DE DADOS II

MÁQUINAS DE BUSCA

Alunos: Daniel Favoreto e Rafael Athaydes

Professora: Mariella Berger

Vitória
12 de Outubro de 2015

Introdução

Este trabalho tem como finalidade a implementação de máquinas de busca baseadas em diferentes estruturas de dados, utilizando duas estruturas de dados principais: Lista e Tabela Hash. O trabalho foi dividido em duas vertentes (módulos), sendo essas: Indexação e Busca.

No módulo indexação (Parte I), houve a implementação de tabela hash com 3 tratamentos de colisão diferentes. Foram utilizados tratamento de colisão por encadeamento, hashing linear e rehashing. E portanto, diferentes desempenhos foram obtidos de acordo com o tratamento de colisão.

Neste mesmo módulo, o usuário pode especificar o modelo para a criação do índice, podendo ser tratamento de colisão por encadeamento, hashing linear e rehashing. Em seguida o usuário especifica um arquivo (podendo ter qualquer nome) contendo os nomes dos documentos a serem indexados. E por final, o usuário especifica o arquivo (podendo ter qualquer nome) que conterá os índices gerados pela indexação seguidos da palavra.

O trabalho desenvolvido também permite a busca por palavras exatamente na posição em que são apresentadas nos documentos de entrada, através do módulo de Busca (Parte II). Neste módulo, o usuário especifica o modelo usado para a criação do índice, podendo ser tratamento de colisão por encadeamento, hashing linear e rehashing. Em seguida o usuário especifica o arquivo resultante do módulo anterior contendo os índices gerados seguidos da palavra. Por final, o usuário deve especificar o arquivo com as buscas a serem feitas podendo ser palavras ou frases completas. Neste módulo a saída na tela do usuário é a palavra ou frase buscada seguida de enter e os nomes dos documentos em que ela aparece.

Vale notar que não houve a implementação de Árvore B neste trabalho.

Implementação

O trabalho é implementado na linguagem C e compilado no GCC. A figura 1 representa as estruturas de dados usadas:

```
 9 typedef char* string;
10
11 typedef struct listaDocumentos{
12         string nome;
13         int *pos,nPos;
14         struct listaDocumentos *prox;
15 }ListaDocumentos;
16
17 typedef struct listaString{
18         string palavra;
19         ListaDocumentos *docs;
20         struct listaString *prox;
21 }ListaString;
22
```

Figura 1: Estruturas de dados utilizadas

As estruturas utilizadas assim como a assinatura das funções estão no arquivo **Functions-Trab.h**.

Para a simplificação do uso convencionou-se criar o tipo string. A estrutura `ListaDocumentos` utiliza um campo para o nome do documento (`nome`), campo para o número de ocorrências da string no documento (`npos`), campo para a posição da string no documento (note que é a posição em relação às palavras e não em relação às linhas) (`pos`) e ponteiro para o próximo documento em que a string ocorre (`*prox`). A estrutura foi útil para todas as hashs e diferentes tratamentos de colisão (Encadeamento, Hashing Linear e Rehashing).

A estrutura `ListaString` utiliza um campo para o nome da string (`nome`), ponteiro para a lista de documentos em que a palavra ocorre (`*docs`) e ponteiro para a próxima string (`*prox`). A estrutura foi idealizada primeiramente para a hash encadeada, para que as strings em que ocorressem colisão pudessem ser conectadas entre si, mas a dupla achou prático utilizá-la para as demais tabelas hash.

No arquivo **Trabalho2.c** está a função `main` principal que recebe os parâmetros de entrada e faz a verificação se o número de argumentos corresponde com a especificação. Depois disso a função se encarrega de direcionar de acordo com o módulo de entrada (-i ou -b) a indexação ou a busca de palavras, assim como se encarrega de identificar o tipo do tratamento de colisão (E,L,R), caso o parâmetro (módulo) seja -i, passa para a função **indexarHash** que tem como parâmetros um dos tipos (E,L,R), os argumentos `argv[3]` e `argv[4]` que são os arquivos a serem indexados e o arquivo de saída que possuirá o índice gerado respectivamente. Caso seja o módulo -b como parâmetro, o fluxo se dirige para a função **buscarPalavra** que tem como parâmetros um dos tipos (E,L,R) e o argumento **argv[4]** que representa o arquivo que contém as palavras a serem buscadas.

Funções utilizadas:

Indexação: As funções principais para este módulo estão representadas na figura abaixo.

A função **indexarHash** faz a leitura de string em string do arquivo de entrada e faz um cálculo

```
24 void indexarHash(char tipo,string entradaDocs, string indiceGerado);
25 void hashEncadeada(string palavra, int indice, int posicao,string nomeDoc);
26 void reHash(string palavra, int indice, int posicao, string nomeDoc);
27 void hashLinear(string palavra, int indice, int posicao, string nomeDoc);
```

aproximado do tamanho do arquivo de entrada. Essa função também é responsável por fazer a retirada de caracteres indesejados do texto. Em seguida faz um cálculo para o índice capturando cada caractere da palavra, e caso a posição do caractere na palavra seja par, utiliza a posição na tabela ASCII vezes a posição na palavra mais 7 e soma ao índice calculado anteriormente. Caso a posição do caractere na palavra seja ímpar, utiliza a posição na tabela ASCII vezes o índice calculado anteriormente mais a posição na palavra somado a 11. Esse índice da string é recalculado pegando-se o resto da divisão pelo tamanho da hash. De acordo com o tipo passado como parâmetro (E,L,R) chama-se uma das seguintes funções: **hashEncadeada**,**reHash** ou **hashLinear**.

A função **hashEncadeada** tem o papel de alocar e inicializar uma hash Encadeada caso seja nula e chamar a função addString. Essa função tem o papel de verificar se no índice calculado anteriormente já possui alguma string, caso não haja insere a string na hash na posição especificada pela variável índice e adiciona o nome do documento na lista de documentos da string. Caso uma string já se encontra na posição, a função se encarrega de verificar se a string que já se encontra na hash é a mesma que se deseja inserir e apenas adiciona o documento na lista de documentos da palavra. Se for uma string diferente, tem-se uma lista de strings na mesma posição, portanto, a função adiciona a nova string no final da lista de strings dessa posição e adiciona também o documento em que se encontra além de inserir a posição em que a palavra se encontra no campo `pos`.

A função **reHash** tem o papel de alocar o dobro do tamanho atual da hash mais um e inicializar uma hash caso seja nula e verificar se no índice calculado anteriormente já possui alguma string, caso não haja insere a string na hash na posição especificada pela variável índice e adiciona o nome do documento na lista de documentos da string. Caso uma string já se encontra na posição, a função se encarrega de verificar se a string que já se encontra na hash é a mesma que se deseja inserir e apenas adiciona o documento na lista de documentos da palavra. Caso contrário, a função utiliza uma segunda hash que será utilizada com o resto da divisão pelo tamanho da hash. Enquanto houver colisão esse processo de rehash é feito até que se encontre uma posição vazia da hash para inserir a string.

A função **hashLinear** tem o papel de alocar o dobro do tamanho atual da hash mais um e inicializar uma hash caso seja nula e verificar se no índice calculado anteriormente já possui alguma string, caso não haja insere a string na hash na posição especificada pela variável índice e adiciona o nome do documento na lista de documentos da string. Caso uma string já se encontra na posição, a função se encarrega de verificar se a string que já se encontra na hash é a mesma que se deseja inserir e apenas adiciona o documento na lista de documentos da palavra. Caso contrário, a função vai iterando nas posições da hash sequencialmente de posição em posição até que encontre ou uma posição livre ou já tenha iterado em todas as posições possíveis da hash para a partir daí inserir a string.

Tem-se também que ao final da indexação a função **indexarHash** chama a função **gerarIndice** para imprimir no arquivo indiceGerado as palavras indexadas seguidas do número de ocorrências e a posição que se encontram no texto.

Ainda na função main principal invoca-se a função **liberarPonteiros** para fazer a desalocação da hash e de memória e os respectivos "frees".

Busca: As funções principais para este módulo estão representadas na figura abaixo. A função **buscarPalavra** faz a leitura do arquivo que contém as palavras a serem buscadas e

```
36 void buscarPalavra(char tipo,string arqPalavras);
37 void buscarHashEncad (string palavra,ListaDocumentos **docs);
38 void buscarHashLin(string palavra, ListaDocumentos **docs);
39 void buscarReHash(string palavra,ListaDocumentos **docs);
```

direciona de acordo com o tipo (E,L,R) uma das 3 funções: **buscarHashEncad, buscarReHash,buscarHashLin**

A função **buscarHashEncad** faz o mesmo cálculo de índice da função **indexarHash** e acessa a posição especificada pelo índice na tabela hash. A função verifica os documentos de cada palavra e posteriormente imprime na tela os documentos em que a palavra se encontra.

A função **buscarReHash** executa o mesmo cálculo do índice que a função **buscarHashEncad** executa, com a diferença que caso a posição acessada na tabela hash esteja vazia, a função utiliza o mesmo processo de recálculo da função hash que a função **reHash** executa, caso encontre uma posição não nula, executa os mesmos passos de verificação e impressão que a função **buscarHashEncad** executa e caso encontre uma posição não vazia, verifica se a palavra buscada coincide com a palavra na posição da tabela hash, caso verdadeiro, verifica os documentos de cada palavra e posteriormente imprime na tela os documentos em que a palavra se encontra, caso contrário, faz o recálculo da hash até encontrar a palavra buscada.

A função **buscarHashLin** executa o mesmo cálculo do índice que a função **buscarHashEncad** executa, com a diferença que caso a posição acessada na tabela hash esteja vazia, a função procura a palavra nas posições seguintes até encontrar a palavra buscada e verifica os documentos da palavra buscada e os imprime na tela. Caso a posição esteja não vazia, a função apenas faz o processo de verificar os documentos e os imprimir na tela.

Ainda na função main principal invoca-se a função **liberarPonteiros** para fazer a desalocação da hash e de memória e os respectivos "frees".

Análise

Elaboramos uma entrada para comparativo entre os 3 principais tipos de tratamento de colisão sendo hash encadeada, hash linear e rehash com os respectivos tempos de execução:

```
1 escada
2 divórcio
3 céu
4 branco
5 casa
```

```
daniel@ubuntu:~/Downloads/Trabalho2EDII$
daniel@ubuntu:~/Downloads/Trabalho2EDII$ ./trab2 -i E entradaDocs indiceGerado
6.160000
```

```
daniel@ubuntu:~/Downloads/Trabalho2EDII$ ./trab2 -i L entradaDocs indiceGerado
25.000000
```

```
daniel@ubuntu:~/Downloads/Trabalho2EDII$ ./trab2 -i R entradaDocs indiceGerado
35.610000
```

```
daniel@ubuntu:~/Downloads/Trabalho2EDII$ ./trab2 -b E indiceGerado arqBuscas
escada
biblia.txt
div♦rcio
biblia.txt
c♦u
biblia.txt
branco
biblia.txt
casa
biblia.txt
0.010000
```

```
daniel@ubuntu:~/Downloads/Trabalho2EDII$ ./trab2 -b L indiceGerado arqBuscas
escada
biblia.txt
div♦rcio
biblia.txt
c♦u
biblia.txt
branco
biblia.txt
casa
biblia.txt
0.020000
```

```
daniel@ubuntu:~/Downloads/Trabalho2EDII$ ./trab2 -b R indiceGerado arqBuscas
escada
biblia.txt
div♦rcio
biblia.txt
c♦u
biblia.txt
branco
biblia.txt
casa
biblia.txt
0.010000
```

Como pode-se constatar, o tratamento de colisão por encadeamento torna-se mais eficiente em relação aos demais tratamentos em virtude da simplicidade de adicionar as palavras com colisão ao final de uma lista. Vale lembrar de que foi usada a Bíblia para a indexação.

O desempenho da rehash foi o pior entre os 3 como já esperado, uma vez que calcula várias vezes os índices das palavras quando sofrem colisão na tabela, o que leva a um desempenho pior em relação aos demais tratamentos.

A hash linear apesar de apresentar uma simplicidade para tratar colisões, a procura por uma próxima posição livre faz com que o algoritmo torne-se pouco eficiente principalmente tratando-se da indexação de um arquivo extenso como a bíblia.

Conclusão

Como demonstrado na implementação, os tratamentos de colisão por Hash Encadeada, Hash Linear e ReHash exigiram esforço por parte da dupla para compreender a teoria e implementar na prática para que fossem feitas com êxito. Obteve-se um grande acúmulo de conhecimento com relação à como implementar e qual tratamento utilizar dependendo do uso.

Devido à pesquisas de outros tratamentos de colisão, a dupla compreendeu que existem algoritmos mais sofisticados que podem comprometer um pouco mais o desempenho de indexação mas que para buscas na tabela podem oferecer menos trabalho e melhor espalhamento.

Constatou-se que a função reHash ofereceu um melhor espalhamento na tabela hash.

As aulas dadas em sala de aula a respeito do assunto também forneceram uma boa base para que fossem implementados os algoritmos, embora não houve a implementação de Árvore B, a dupla compreendeu seu uso e importância principalmente nas buscas.

Bibliografia

Sites da web:

stackoverflow.com

en.wikipedia.org

Livros:

ZIVIANI, N. Projeto de Algoritmos, Cengage Learning.