Máquinas de Busca

André Barreto e Igor Ventorim

Universidade Federal do Espírito Santo
Vila Velha, Espírito Santo

12 de Outubro, 2015

Introdução

Máquina de Busca é um sistema de recuperação de informação com o objetivo de recuperar informações armazenadas em um sistema computacional. É implementado para buscar por palavras-chave em uma base de dados de forma eficiente, retornando ao usuário onde se encontram as informações desejadas, de forma ranqueada ou não. Para isto, deve-se criar uma estrutura de dados apropriada pra guardar as informações de tal maneira que seja rápido recuperá-las. [1]

Neste trabalho, serão implementadas duas estruturas de dados para simular uma Máquina de Busca que se atenha a definição descrita acima.

1 Implementação

Nesta seção serão apresentados as estruturas de dados utilizadas e suas implementações em linguagem C.

Para criar as estruturas, primeiro é necessário analisar e processar os documentos recebidos para a leitura. Este processamento baseia-se em ler palavra por palavra, remover pontuações, acentos e outros carácteres não convenientes para serem armazenados. Por padrão, a linguagem C trabalha apenas com a tabela ASCII, que não contém muitos símbolos presentes em português. Em decorrência disto, para realizar o pre-processamento, será utilizado uma extensão do armazenamento de carácteres em C denominado *wide char*, contido na biblioteca *wchar.h*. [2] Desta forma tornou-se mais viável a manipulação da entrada de dados do programa.

Neste procedimento as palavras processadas são armazenadas em

uma lista encadeada ordenada, onde é verificado pela repetição da mesma. Caso ela já exista, serão incluídas as informações na palavra. Se não, esta é inserida na lista.

O módulo *preProcess* é o responsável por realizar o algoritmo descrito acima.

1.1 Hash

Tabela *hash*, ou Tabela de dispersão, é um tipo de estrutura de dados especial, que associa chaves de pesquisa a valores, sendo seu principal objetivo, fazer buscas rápidas a partir de uma chave simples, desta forma conseguindo localizar o valor desejado. Na prática, a tabela de dispersão funciona da seguinte forma, informando um conteúdo através de uma função geradora simples e não muito custosa se gera um índice, ao qual será a posição em que será inserido o registro. Sendo assim muito fácil localizar onde está guardado este registro, colocando o conteúdo que foi usado para gerar a *hash key*, conseguimos encontrar a chave onde foi armazenada este registro e assim conseguindo recuperar o registro. Um problema no uso de tabelas de dispersão é no caso de colisões ao qual o conteúdo de um registro gera uma chave na qual já foi inserido um registro, onde se pode contornar este problema de diversas maneiras, três maneiras serão apresentadas neste relatório que são elas: Encadeamento, Linear e Rehashing. [3]

Para a implementação da *hash* foi utilizado um vetor de ponteiros do tipo estrutura palavra, ao qual foi criada a *hash* através da função *createHashTable*, na qual é passado para a ela o tamanho da *hash*, no qual foi adquirido pela lista de palavras que foi montada e encontrando o número primo mais próximo do dobro de palavras contidas na lista ao qual é adquirido pela função gerarSizeHash, após a *hash* criada é necessário preencher-la a função que popula é a função *fillHash*, assim que a *hash* é populada, é necessário exportá-la com a função *exportHash*, desta forma terminando a indexação. No módulo de busca a *hash* é importada através da função *buscaHash* a qual recebe os parâmetros de busca, assim chamando função *hashImport* e remontando a *hash* e chamando as subfunções de buscas.

1.1.1 Hash por Encadeamento

O tipo de tratamento de colisões com encadeamento funciona da seguinte forma: quando ocorrer uma colisão de índices na tabela, o elemento será armazenado em uma lista encadeada. Logo, para realizar a busca após encontrado o índice será necessário pesquisar o elemento na lista encadeada.

PRÓS: Cálculo do índice feito uma única vez.

CONTRA: Busca pode se tornar sequencial caso o tamanho da lista seja muito grande.

1.1.2 Hash Linear

O tipo de tratamento de colisões de modo linear funciona da seguinte forma: quando ocorrer uma colisão de índices na tabela, o elemento será armazenado no próximo índice, caso o próximo índice esteja ocupado, tentará sempre o índice mais um, até encontrar uma posição vazia. Desta forma, para realizar a busca, perguntará se o elemento de índice é o elemento que está sendo procurado, e buscando sempre o próximo sucessivamente, conforme a inserção.

PRÓS: Simplicidade de implementação.

CONTRA: Se ocorrer muitas colisões podem se agrupar muitos elementos em uma região, dificultando encontrar o elemento correto.

1.1.3 Hash por Rehashing

O tipo de tratamento de colisões de modo linear funciona de forma análoga a linear porém sendo necessário uma segunda função geradora de chave, a qual quando encontrada colisão será somada a função geradora de key e verificar se a nova posição não é uma colisão, caso seja, será somado o valor do resultado da segunda função geradora de key e assim sucessiva, até encontra uma posição vaga. De modo similar, podemos fazer a busca, tentando encontrar primeiramente o índice gerado pela função geradora de key, caso não seja, somaremos o valor gerado pela segunda função geradora de key ao índice e perguntar se o elemento é o procurado, fazendo assim sucessivamente até se encontrar o elemento.

PRÓS: Ajuda no espalhamento dos elementos na hash.

CONTRA: O índices podem está muito distantes uns dos outros podendo violar o príncipio de localidade.

1.1.4 Principais funções

A função *hashKey* é a função responsável por gerar as chaves para a tabela *hash*, sendo ela uma função com um espalhamento aceitável em relação as colisões, diferenciando palavras que são anagramas em suas *keys*.

```
1 int hashKey(char *word, const int sizeTable)
2 {
3     int codigo = 1;
4     unsigned int i;
```

```
5
6       for(i = 0; i < strlen(word);i++)
7           codigo = (31*codigo + (int)word[i])%sizeTable;
8
9       return abs(codigo%sizeTable);
10  }
```

Listing 1: Função hash key

A função *insertWordHash* é a função responsável por inserir palavras na *hash*. Após selecionado a *key* correspondente a palavra é criada e inserida na posição correta, se adequando ao padrão da *hash table*.

```
1  void insertWordHash(tpWord *word, wordList** list)
2  {
3           wordList *new = malloc(sizeof(wordList));
4           new->word = word;
5           new->prox = *list;
6           *list = new;
7  }
```

Listing 2: Função inserir palavra na Hash

A função *searchType* é a função principal de busca em relação as *hashs*, sendo ela uma função seletora de buscas, na qual seleciona as funções *searchHashE*, *searchHashL*, *searchHashR*, a qual fazem a busca de palavras de acordo com o seu tipo.

```
1  wordList* searchType(wordList **hashTable, char *word,char type,
       int sizeHash)
2  {
3    int key = hashKey(word,sizeHash);
4    wordList *aux = hashTable[key];
5    if(type == 'E')
6    {
7      aux = searchHashE(aux,word);
8      return aux;
9    }else
10   if(type == 'L')
11   {
12     aux = searchHashL(hashTable,sizeHash,word);
13     return aux;
14   }else
15   if (type == 'R')
16   {
17     aux = searchHashR(hashTable,sizeHash,word);
18     return aux;
19   }else
20     printf("Tipo informado invalido.");
21
22   return NULL;
23 }
```

Listing 3: Função de busca

1.2 Árvore B

Árvore B é uma estrutura de dados projetada para funcionar especialmente em memória secundária como um disco magnético ou outros dispositivos de armazenamento secundário. Dentre suas propriedades ela permite a inserção, remoção e busca de chaves numa complexidade de tempo logarítmica e, por esse motivo, é muito empregada em aplicações que necessitam manipular grandes quantidades de informação tais como um banco de dados ou um sistema de arquivos. [4]

Nesta implementação, trabalharemos apenas com memória, ou seja, a árvore B não estará fragmentada em memória secundária. Além disto, não ocorrerá remoções, portanto esta função não foi elaborada.

No módulo de indexação, primeiramente é chamado a função *fillHash* que recebe com parâmetro a árvore, neste ponto um ponteiro para *NULL*, e a lista de palavras do documento processado. Esta função executa um laço para cada palavra da lista inserindo-as na árvore através da função *insereArvB*. Esta, por sua vez, juntamente com a função *insere* controla em qual situação e onde deverá ser inserido a palavra recebida por parâmetro. Quando encontrado esta posição, faz-se um deslocamento do vetor de palavras no nó e insere a palavra com a função *insereWord*. Após completa a árvore, esta é exportada em um arquivo de texto indicado no momento da execução do programa através da função *geraIndex*.

No módulo de busca, é lido de um arquivo a árvore com a função *importArvB*, que analisa as palavras lidas procurando pelos carácteres que indicam o que será lido. Após recuperado a árvore, executa-se a função *buscaArvB*. Esta verifica se a busca a ser realizada é de uma palavra simples, múltiplas palavras ou de uma frase (palavras seguidas), chamando *searchArvB*, *multiSearchArvB* ou *sentenceSearchArvB*, respectivamente.

1.2.1 Estrutura

Nesta Implementação de árvore B, foi escolhido, entre vários conceitos, que a ordem M representa o número de ponteiros para subárvores na estrutura. Por conseguinte, teremos *M-1* palavras armazenadas para cada nó da árvore. Além disto, o número mínimo de palavras será de $\lfloor \frac{M}{2} \rfloor$. A variável n representa o número de palavras já inseridas no nó atual; **word* é o vetor de palavras ordenado e **p* é o vetor de ponteiros para subárvores.

```
1 #define M 1001   // Ordem da Arvore B
2
3 typedef struct arvoreB ArvoreB;
4 struct arvoreB {
5     int n;              // Numero de nos ativos
6     tpWord *word[M-1];  // Vetor de palavras
```

```
7       ArvoreB *p[M];        // Ponteiros para subarvores
8   };
```

Listing 4: Estrutura da Árvore B

1.2.2 Principais funções

Esta primeira função é a chamada quando deve-se inserir uma palavra em uma árvore. Recebendo como parâmetros um ponteiro para a raiz da árvore e um para a palavra, ela chama outra função. A *insere* é uma função recursiva que descobre a posição onde deve ser inserido a palavra e analisa em que situação a árvore se encontra. Esta foi omitida por ser muito extensa. Inicialmente, quando a árvore é nula, será retornado $h = 1$ para que esta seja criada. Para cada situação, a variável h será alterada para o controle dos casos.

```
1  ArvoreB *insereArvB(ArvoreB *raiz, tpWord *word)
2  {
3      int h;
4      ArvoreB *filho_dir, *nova_raiz;
5      tpWord *wordReturn = NULL;
6
7      filho_dir = insere(raiz, word, &h, &wordReturn);
8      if(h) { // Aumetara a altura da arvore
9          nova_raiz = criaArvB();
10         nova_raiz->n = 1;
11         nova_raiz->word[0] = wordReturn;
12         nova_raiz->p[0] = raiz;
13         nova_raiz->p[1] = filho_dir;
14
15         return(nova_raiz);
16     }
17     else return(raiz);
18 }
```

Listing 5: Função de inserção de palavras

A função *insereWord* obtém a posição certa no nó *raiz* e o insere. A verificação de um possível *split* é realizada na função *insere*.

```
1  void insereWord(ArvoreB *raiz, tpWord *word, ArvoreB *filhodir)
2  {
3      int k, pos;
4
5      k = raiz->n;
6      // Busca para obter a posicao ideal para inserir a nova
       palavra
7      pos = buscaBinaria(raiz, word->string);
8      if(raiz->word[pos] != NULL)
9          if(strcmp(raiz->word[pos]->string, word->string) < 0)
       pos++;
```

```
10
11      // Realiza o remanejamento para manter as palavras ordenadas
12      while (k > pos && strcmp(word->string, raiz->word[k-1]->
        string) < 0) {
13          raiz->word[k] = raiz->word[k-1];
14          raiz->p[k+1] = raiz->p[k];
15          k--;
16      }
17      // Insere a palavra na posicao ideal
18      raiz->word[pos] = word;
19      raiz->p[pos+1] = filhodir;
20      raiz->n++;
21  }
```

Listing 6: Função inserir palavra em um nó

Esta é a função de busca primordial. Dado um ponteiro para árvore e uma *string* correspondente a uma palavra, é realizado uma busca recursiva da mesma. Se encontrou, retorna a referência da estrutura da palavra. Caso contrário, retorna *NULL*.

```
1  tpWord *searchArvB(ArvoreB *raiz, char *word)
2  {
3      int pos;
4
5      if(raiz == NULL)
6          return NULL;
7
8      pos = buscaBinaria(raiz, word);
9
10     if(strcmp(raiz->word[pos]->string, word) == 0)
11         return raiz->word[pos]; // Palavra encontrada
12
13     if(strcmp(raiz->word[pos]->string, word) > 0)
14         return searchArvB(raiz->p[pos], word); // Buscar a
       esquerda
15     else
16         return searchArvB(raiz->p[pos+1], word); // Buscar a
       direita
17
18     return NULL;
19 }
```

Listing 7: Função de busca

Para as buscas avançadas, existem outras duas funções: *multiSearchArvB* e *sentenceSearchArvB*, que buscam por várias palavras em um mesmo arquivo e por uma frase (palavras seguidas em um mesmo arquivo), respectivamente. Estas acabam por chamar a *searchArvB*, para localizar a(s) palavra(s).

Além disto, existe a *buscaArvB* que é responsável pelo controle de qual das funções mencionadas acima devem ser convocadas.

2 Análise

Foram realizados uma série de testes para verificar a velocidade e quantidade de memória utilizada para cada uma das estruturas. Oficialmente, três diferentes testes e com esforço computacional crescente foram anotados para se estabelecer uma comparação. Estes testes foram executados em uma máquina com as seguintes configurações:

Linux Mint 17.2 Cinnamon 64-bit
Intel Core i7-3770 CPU @ 3.40GHz x 4
8GB de memória

Os experimentos foram avaliados com três diferentes conjuntos de arquivos e um mesmo arquivo de busca.

Conjunto de arquivos:

conj1: doc1.txt + doc2.txt + doc3.txt
Tamanho e total de palavras: ∼80KB , 11184 palavras

conj2: doc1.txt + doc2.txt + doc3.txt + BibliaSagrada.txt
Tamanho e total de palavras: ∼4MB , 743735 palavras

conj3: doc1.txt + doc2.txt + doc3.txt + BibliaSagrada.txt + ProjectGutenberg.txt
Tamanho e total de palavras: ∼10.2MB , 1839430 palavras

Arquivo de busca:

10	Deus vivo
teste	"olá mundo"
o	"joao ferreira"
A	"a 10"
não sei	"o não politico"
o paradigma	"o"
10 a	"o paradigma"
paradigma o	"João"
Paradigma politico	"Ferreira"
Não	visão
olá mundo	Estratégias
teste	

2.1 Hash

Veja na Tabela 1 os resultados obtidos utilizando os três tipos de Hash implementados.

Tipo	Arquivos	Tempo de indexação	Tempo de busca
Hash E	conj1	0m0.021s	0m0.006s
	conj2	1m47.454s	0m0.310s
	conj3	14m32.662s	0m0.504s
Hash L	conj1	0m0.033s	0m0.005s
	conj2	1m47.559s	0m0.317s
	conj3	19m3.836s	0m0.490s
Hash R	conj1	0m0.032s	0m0.006s
	conj2	1m47.560s	0m0.309s
	conj3	16m13.631s	0m0.516s

Tabela 1: Resultados hash

2.2 Árvore B

Veja na Tabela 2 os resultados obtidos utilizando Árvore B.

Arquivos	Tempo de indexação	Tempo de busca
conj1	0m0.038s	0m0.006s
conj2	1m40.388s	0m0.294s
conj3	31m35.121s	0m0.469s

Tabela 2: Resultados árvore B

2.3 Execução com Valgrind

Para verificar se o programa é robusto, rodamos o programa com *valgrind*. A seguir estão as saídas para a indexação e busca de alguns dos módulos implementados.

2.3.1 Hash Encadeada

Teste de execução da indexação da Hash por Encadeamento utilizando os arquivos indicados por *conj1*:

```
==12691== Memcheck, a memory error detector
==12691== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward
et al.
==12691== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for
copyright info
==12691== Command: ./trab2 -i E documentos.txt index.txt
==12691==
==12691==
```

```
==12691== HEAP SUMMARY:
==12691== in use at exit: 0 bytes in 0 blocks
==12691== total heap usage: 14,248 allocs, 14,248 frees, 247,082
bytes allocated
==12691==
==12691== All heap blocks were freed - no leaks are possible
==12691==
==12691== For counts of detected and suppressed errors, rerun with:
-v
==12691== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

Teste de execução de busca da Hash por Encadeamento utilizando o arquivo de busca indicado por previamente:

```
==13503== Memcheck, a memory error detector
==13503== Copyright (C) 20022013, and GNU GPL'd, by Julian Seward
et al.
==13503== Using Valgrind-3.10.0.SVN and LibVEX; rerun with h for
copyright info
==13503== Command: ./trab2 b E query.txt index.txt
==13503==
o
doc1.txt
doc2.txt
doc3.txt
(...)
==13503==
==13503== HEAP SUMMARY:
==13503== in use at exit: 0 bytes in 0 blocks
==13503== total heap usage: 13,864 allocs, 13,864 frees, 237,797
bytes allocated
==13503==
==13503== All heap blocks were freed  no leaks are possible
==13503==
==13503== For counts of detected and suppressed errors, rerun with:
-v
==13503== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

2.3.2 Árvore B

Teste de execução de indexação da Árvore B utilizando os arquivos indicados por *conj1*:

```
==12999== Memcheck, a memory error detector
==12999== Copyright (C) 20022013, and GNU GPL'd, by Julian Seward
et al.
==12999== Using Valgrind3.10.0.SVN and LibVEX; rerun with h for
copyright info
==12999== Command: ./trab2 i B documentos.txt indexArv.txt
==12999==
==12999==
==12999== HEAP SUMMARY:
==12999== in use at exit: 0 bytes in 0 blocks
==12999== total heap usage: 13,867 allocs, 13,867 frees, 250,362
bytes allocated
==12999==
==12999== All heap blocks were freed  no leaks are possible
==12999==
==12999== For counts of detected and suppressed errors, rerun with:
v
==12999== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

Teste de execução de busca da Árvore B utilizando o arquivo indicado por previamente:

```
==13196== Memcheck, a memory error detector
==13196== Copyright (C) 20022013, and GNU GPL'd, by Julian Seward
et al.
==13196== Using Valgrind3.10.0.SVN and LibVEX; rerun with h for
copyright info
==13196== Command: ./trab2 b B query.txt index.txt
==13196==
(...)
o paradigma
doc3.txt
doc1.txt
(...)
==13196==
==13196== HEAP SUMMARY:
==13196== in use at exit: 0 bytes in 0 blocks
```

```
==13196== total heap usage: 13,564 allocs, 13,564 frees, 248,203
bytes allocated
==13196==
==13196== All heap blocks were freed  no leaks are possible
==13196==
==13196== For counts of detected and suppressed errors, rerun with:
v
==13196== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

3 Conclusão

As estruturas implementadas são ambas apropriadas para o âmbito de buscas, porém variam em sua aplicabilidade dependendo de um conjunto de variáveis referentes à base de dados alvo para se armazenar.

A *Hash* é uma estrutura relativamente simples de ser implementada e eficiente, dependendo do método de dispersão e tratamento de colisão. Porém ela é uma estrutura dependente do conjunto de dados, pois deve ter no máximo metade do tamanho ocupada, e é fixa. De tal forma que, caso o conjunto de dados cresça mais que o esperado, a *hash* perderá a eficiência.

A Árvore B, por outro lado, independe do conjunto de dados, pois pode crescer indefinidamente. Mas é uma estrutura mais complexa e, se mal implementada, pode ser ineficiente.

Com os dados coletados, vemos que os tempos de indexação entre as *Hashs* foram próximos, assim como os tempos de busca. A Árvore B conseguiu buscas mais rápidas, porém bem pouco mais rápidas que as buscas da *Hash*. O tempo de indexação da árvore se mostrou significantemente lenta a medida que a entrada cresce. Este fato aparentemente decorre do método de processamento de dados, que são armazenados em uma lista antes de começar a inserir na árvore. Além disto, é necessário fazer deslocamentos em vetores na inserção, que é uma operação custosa.

Referências

[1] Wikipedia, *Search engine*, July 2015. 1

[2] Wikibooks, *C Reference - wchar.h*, June 2014. 1

[3] N. Ziviani, *Projeto de Algoritmos - Pesquisa em Memória Secundária*, Aug. 2010. 2

[4] N. Ziviani, *Projeto de Algoritmos - Pesquisa em Memória Primária*, Sept. 2010. 5