Universidade Federal
do Espírito Santo

Centro Tecnológico

Departamento de Informática

Máquina de Busca

Autor:
Vinicius Arruda

Professora:
Mariella Berger

12 de outubro de 2015

Resumo

Trabalho da disciplina de Estrutura de Dados II, que consiste na implementação de uma máquina de busca utilizando duas estruturas de dados distintas. As estruturas são Tabela de Dispersão e Árvore B. Para o tratamento de colisões da Tabela de Dispersão foram empregados três estratégias: Hashing Linear, Encadeamento e Rehashing.

1 Introdução

Uma máquina de busca é um programa projetado para procurar palavras-chave em documentos e outras base de dados. O google por exemplo, é uma máquina de busca que tem como base de dados a internet para procurar as palavras-chave.

As máquinas de busca surgiram logo após o aparecimento da Internet, com a intenção de prestar um serviço extremamente importante: a busca de qualquer informação na rede, apresentando os resultados de uma forma organizada, e também com a proposta de fazer isto de uma maneira rápida e eficiente.

A proposta deste trabalho é mostrar diferentes implementações de estrutura de dados para armazenar e buscar palavras-chave de maneira rápida e eficiente, mostrando uma análise e resultados destas implementações.

2 Objetivo

Aplicar o conhecimento adquirido na disciplina de Estruturas de Dados II para implementar estruturas de dados para armazenamento de grande quantidade de informação e acesso rápido à qualquer informação contida nestas estruturas.

3 Ferramentas

Os algoritmos foram implementados na linguagem de programação C. Para a compilação foi utilizado o compilador GCC versão 4.7.2 em uma máquina com o sistema operacional Debian GNU/Linux 7.8 (wheezy). O código foi escrito utilizando o editor de texto gedit versão 3.4.2. Para a depuração do programa, foi utilizado a ferramenta Valgrind versão 3.7.0.

4 Estruturas

Para a manipulação dos dados foram utilizado quatro estruturas principais:

4.1 Documents

Estrutura que representa os documentos a serem indexados. Contém o número de arquivos a serem lidos e indexados e um vetor de *strings* que é o

nome de cada arquivo.

```
typedef struct
{
  size_t size;
  char** documents;
} Documents;
```

Listing 1: Estrutura Documents

4.2 Doc

Estrutura que representa o documento que uma palavra-chave está. Contém um identificador do arquivo e uma lista de indices que representa a posição de cada palavra-chave no documento.

```
typedef struct
{
  size_t docID;
  List* index;
} Doc;
```

Listing 2: Estrutura Doc

4.3 Word

Estrutura que representa uma palavra-chave. Contém a palavra-chave em si e uma lista de documentos do tipo *Doc* no qual esta palavra-chave se encontra.

```
typedef struct
{
  char* word;
  List* doc;
} Word;
```

Listing 3: Estrutura Word

4.4 Query

Estrutura que representa uma query. Contém o tipo da query, se possui áspas duplas (sinaliza uma pesquisa sequencial) ou se não possui áspas duplas. Contém a query original no formato *wchar* e uma lista de palavras-chave, geradas após o processamento da query original. A lista query tem como informação as palavras-chave no formato ASCII (*char*).

```
typedef enum {NONQUOTES, QUOTES} Q_TYPE;

typedef struct
{
  Q_TYPE type;
  wchar_t* original;
  List* query;
} Query;
```

Listing 4: Estrutura Query e QTYPE

5 Estruturas de Dados

O tamanho das tabelas de dispersão foi calculado com base em um fator de carga (*load factor*) de 70 por cento, e em cima deste valor, para diminuir o número de colisões, é gerado o próximo número primo e este é utilizado como tamanho da tabela.

A função de hash utilizada é a *djb2*, criada por Dan Bernstein. Sua implementação se encontra a seguir:

```
size_t hash_function(char* str)
{
  unsigned long hash = 5381;
  int c;

  c = *str;
  str++;
  while (c)
  {
    hash = hash * 33 + (unsigned long) c;
    c = *str;
    str++;
  }

  return hash;
}
```

Listing 5: Função de hash DJB2

5.1 Hash Linear

O tratamento de colisão por hashing linear se baseia em encontrar o próximo cesto vazio da tabela. A estrutura da hash linear é a seguinte:

```
typedef struct
{
  size_t size;
  Word** table;
} Hash_l;
```

Listing 6: Estrutura da Hash Linear

5.2 Hash Encadeada

O tratamento de colisão por encadeamento se resume à uma lista encadeada em cada cesto da tabela de dispersão. Quando houver colisão, a informação deverá ser encadeada na lista existente. A estrutura da hash encadeada é a seguinte:

```
typedef struct
{
  size_t size;
  List** table;
} Hash_e;
```

Listing 7: Estrutura da Hash Encadeada

5.3 Rehash

O tratamento de colisão por rehashing se resume à realizar uma segunda função de hash, e enquanto não encontrar um cesto vazio, esta segunda função de hash é aplicada, gerando uma nova posição de cesto.

A implementação da segunda função de hash se encontra a seguir, e implementa a soma dos valores dos caracteres referentes à tabela ASCII.

```
size_t hash_2_function(char* str)
{
  size_t hash = 0;

  if(str == NULL || *str == '\0')
  {
    fprintf(stderr, "String cannot be NULL.\n");
    exit(EXIT_FAILURE);
  }

  while(*str != '\0')
```

```
12   {
13     hash += (size_t) *str;
14     str++;
15   }
16
17   return hash;
18 }
```

Listing 8: Segunda função de Hash

A estrutura da rehash é identica à estrutura da hash linear.

5.4 Árvore B

A árvore B foi implementada para servir tanto como uma estrutura de busca quanto para ajudar a indexação das palavras. No tipo modelo B, o arquivo de palavras-chave é indexado diretamente na árvore B, sem passar por nenhuma outra estrutura intermediária. Já nos tipos modelo L, E e R, a árvore B é uma estrutura intermediária para auxiliar o pré-processamento e indexação das palavras-chave. A estrutura da árvore B se encontra a seguir:

```
1 struct btree
2 {
3   int leaf;     // Flag indicando se o no e uma folha ou nao.
4   size_t n;     // Numero de chaves atual do no.
5   void* keys[MAX];  // Ponteiros para as chaves (informacao da
      arvore).
6   struct btree* children[MAX + 1]; // Ponteiros para os nos
      filhos.
7 };
```

Listing 9: Estrutura da Árvore B

Esta implementação utiliza 101 como valor máximo de chaves por nó e 50 como valor mínimo de chaves por nó.

6 Módulo de Indexação

O módulo de indexação das palavras foram divididas em três etapas: tratamento de caracteres, pré-indexação armazenando na árvore B e a conversão dos dados da árvore B para a hash definida pelo tipo modelo. Caso o tipo modelo for a árvore B, esta última etapa não é realizada, pois a árvore já estará pronta na segunda etapa.

6.1 Tratamento de Caracteres

Inicialmente, é feito um tratamento dos caracteres lidos dos arquivos, convertendo todos os caracteres com acento para caracteres sem acento. Os caracteres também são convertidos para maiúsculo. As bibliotecas *wchart.h*, *wctype.h* e *stddef.h* foram amplamente utilizadas para auxiliar a conversão e tratamento de caracteres UTF-8 para caracteres da tabela ASCII.

6.2 Pré-Indexação

Para o processo de pré-indexação, foi utilizado como estrutura de dados uma Árvore B. Esta escolha foi baseada na eficiência que esta estrutura possui para armazenar a informação e na velocidade de busca da informação.

O processo de pré-indexação se baseou em pegar a palavra-chave lida do arquivo (já convertida para ASCII maiúsculo) e verificar se já existe a palavra-chave na árvore, se não, a palavra-chave é inserida de acordo com a estrutura *Word*, contendo o documento em que foi encontrada e a posição dela no documento. Caso a palavra já exista na árvore, é verificado se o identificador do documento da palavra-chave a ser indexada se encontra na lista de documentos da palavra chave da árvore. Se não, este identificador é incluído na lista. Caso contrário a lista de indices daquele documento é atualizada com a adição do índice da palavra-chave a ser indexada. A função que faz este tratamento é a *handle_word()*, e sua implementação é vista a seguir:

```
1  size_t handle_word(Btree** btree, char* word, size_t doc_id,
       size_t index)
2  {
3    Word* word_in_list;
4    Doc* doc_in_list;
5
6    if((word_in_list = (Word *) binarySearch(*btree, word,
       cmpStrKey)) == NULL)
7    {
8      word = strdup(word);
9
10     if(word == NULL)
11     {
12       fprintf(stderr, "Out of memory.\n");
13       exit(EXIT_FAILURE);
14     }
15
16     insert(btree, create_word(word, create_doc(doc_id,
       create_index(index))), cmpKey);
17
```

```
18      return 1;
19    }
20    else
21    {
22      if((doc_in_list = searchInfo(word_in_list->doc, &doc_id,
       compare_doc)) == NULL)
23      {
24        insertBegList(&word_in_list->doc, create_doc(doc_id,
       create_index(index)));
25      }
26      else
27      {
28        insertBegList(&doc_in_list->index, create_index(index));
29      }
30    }
31
32    return 0;
33  }
```

Listing 10: Função handle_word

6.3 Conversão da estrutura

Após as palavras serem indexadas na árvore B, os dados devem ser armazenados na tabela de dispersão descrita pelo tipo modelo. Para esta conversão, o processo foi simplesmente desmontar a árvore B inserindo cada ponteiro para informação que foi retirado da árvore na tabela de dispersão. A função responsável por este processo é a função recursiva *convertBtree()* e sua implementação se encotra a seguir:

```
1  // data: Ponteiro para a tabela de dispersao.
2  // insert: Funcao para a insercao da informacao na tabela de
       dispersao.
3
4  void convertBtree(Btree* btree, void* data, void (* insert) (
       void *, void *))
5  {
6    size_t i;
7
8    for(i = 0; i < btree->n; i++)
9      insert(data, btree->keys[i]);
10
11   if(btree->leaf == FALSE)
12   {
13     for(i = 0; i < btree->n + 1; i++)
14       convertBtree(btree->children[i], data, insert);
15   }
16
```

```
17    free(btree);
18 }
```

Listing 11: Função convertBtree

7 Módulo de busca

O módulo de busca se baseou em pesquisar cada *query*, composta de uma ou mais palavras-chave na estrutura de dados e retornar a lista de documentos que contém esta *query*. Caso a *query* esteja entre áspas duplas, além de ser verificado a lista de documentos que a palavra-chave está, é também verificado a lista de indices de cada documento, e se a ordem das palavras na *query* for igual à ordem da lista de indices, então a *query* existe naquele documento.

A função central para a busca é a função *intersection()*, que faz a interseção dos dados das listas.

```
1  // Faz a intersecao dos dados de duas listas.
2  // Ex: A = 0 2 4 6
3  //     B = 1 3 6 9 10 2
4  //     Lista apos a intersecao de A e B = 6 2
5
6  void intersection(List** a, List* b, int (* equals) (void *,
       void *), void (* freeInfo) (void *))
7  {
8    List* end;
9    void* info;
10   List* new = createEmptyList();
11
12   while((info = removeBegList(a)) != NULL)
13   {
14     if(isInList(b, info, equals))
15     {
16       insertEndList(&new, &end, info);
17     }
18     else
19     {
20       freeInfo(info);
21     }
22   }
23
24   *a = new;
25 }
```

Listing 12: Função de interseção

8 Avaliação e Resultados

Para a avaliação dos algoritmos foram indexados os seguintes documentos:

Tabela 1: Documentos para indexação

Número de palavras	Nome	Tamanho
781258	biblia.txt	4.4 MB
30752	dataminer.txt	208.7 kB
154442	dicionario1.txt	1.02 MB
164162	dicionario2.txt	1.11 MB
246	ia.txt	1.80 kB
187982	losrofrings.txt	1.0 MB
10205	matematica.txt	63.7 kB

A avaliação do tempo de indexação foi feita utilizando a função *time* disponível por linha de comando nos sitemas linux.

Tabela 2: Tempo de indexação

Tipo Modelo	Tempo
Hash Linear	1.471s
Hash Encadeada	1.477s
Rehash	1.483s
Árvore B	1.431s

Para a avaliação da busca, diversas *query* de comprimentos variados foram elaboradas, sendo 18 com àspas duplas e 19 sem àspas duplas, somando um total 216 palavras. O tempo de busca de cada tipo modelo se encontra a seguir:

Tabela 3: Tempo de busca

Tipo Modelo	Tempo
Hash Linear	0.728s
Hash Encadeada	0.738s
Rehash	0.716s
Árvore B	0.727s

9 Referências Bibliográficas

1. https://www.cs.usfca.edu/~galles/visualization/BTree.html
2. http://www.cse.yorku.ca/~oz/hash.html
3. http://www.aquaphoenix.com/ref/gnu_c_library/libc_311.html
4. http://pubs.opengroup.org/onlinepubs/007908775/xsh/wchar.h.html
5. http://www.gnu.org/software/libc/manual/html_node/Setting-the-Locale.html#Setting-the-Locale
6. http://tex.stackexchange.com
7. http://www.inf.ufes.br/~mberger/Disciplinas/2015_2/EDII/