17. Tabelas de dispersão

W. Celes e J. L. Rangel

No capítulo anterior, discutimos diferentes estruturas e algoritmos para buscar um determinado elemento num conjunto de dados. Para obtermos algoritmos eficientes, armazenamos os elementos ordenados e tiramos proveito dessa ordenação para alcançar eficientemente o elemento procurado. Chegamos a conclusão que os algoritmos eficientes de busca demandam um esforço computacional de *O(log n)*. Neste capítulo, vamos estudar as estruturas de dados conhecidas como tabelas de dispersão (*hash tables*), que, se bem projetadas, podem ser usadas para buscar um elemento da tabela em ordem constante: *O(1)*. O preço pago por essa eficiência será um uso maior de memória, mas, como veremos, esse uso excedente não precisa ser tão grande, e é proporcional ao número de elementos armazenados.

Para apresentar a idéia das tabelas de dispersão, vamos considerar um exemplo onde desejamos armazenar os dados referentes aos alunos de uma disciplina. Cada aluno é individualmente identificado pelo seu número de matrícula. Podemos então usar o número de matrícula como chave de busca do conjunto de alunos armazenados. Na PUC-Rio, o número de matrícula dos alunos é dado por uma seqüência de oito dígitos, sendo que o último representa um dígito de controle, não sendo portanto parte efetiva do número de matrícula. Por exemplo, na matricula `9711234-4`, o ultimo dígito `4`, após o hífen, representa o dígito de controle. O número de matrícula efetivo nesse caso é composto pelos primeiros sete dígitos: `9711234`.

Para permitir um acesso a qualquer aluno em ordem constante, podemos usar o número de matrícula do aluno como índice de um vetor – `vet`. Se isso for possível, acessamos os dados do aluno cuja matrícula é dado por `mat` indexando o vetor – `vet[mat]`. Dessa forma, o acesso ao elemento se dá em ordem constante, imediata. O problema que encontramos é que, nesse caso, o preço pago para se ter esse acesso rápido é muito grande.

Vamos considerar que a informação associada a cada aluno seja representada pela estrutura abaixo:

```
struct aluno {
   int mat;
   char nome[81];
   char email[41];
   char turma;
};
typedef struct aluno Aluno;
```

Como a matrícula é composta por sete dígitos, o número inteiro que conceitualmente representa uma matrícula varia de 0000000 a 9999999. Portanto, precisamos dimensionar nosso vetor com dez milhões (10.000.000) de elementos. Isso pode ser feito por:

```
#define MAX 10000000
Aluno vet[MAX];
```

Dessa forma, o nome do aluno com matrícula `mat` é acessado simplesmente por: `vet[mat].nome`. Temos um acesso rápido, mas pagamos um preço em uso de

memória proibitivo. Como a estrutura de cada aluno, no nosso exemplo, ocupa pelo menos 127 bytes, estamos falando num gasto de 1.270.000.000 bytes, ou seja, acima de 1 Gbyte de memória. Como na prática teremos, digamos, em torno de 50 alunos cadastrados, precisaríamos apenas de algo em torno de 6.350 (=127*50) bytes.

Para amenizar o problema, já vimos que podemos ter um vetor de ponteiros, em vez de um vetor de estruturas. Desta forma, as posições do vetor que não correspondem a alunos cadastrados teriam valores `NULL`. Para cada aluno cadastrado, alocaríamos dinamicamente a estrutura de aluno e armazenaríamos um ponteiro para essa estrutura no vetor. Neste caso, acessaríamos o nome do aluno de matrícula `mat` por `vet[mat]->nome`. Assim, considerando que cada ponteiro ocupe 4 bytes, o gasto excedente de memória seria, no máximo, aproximadamente 40 Mbytes. Apesar de menor, esse gasto de memória ainda é proibitivo.

A forma de resolver o problema de gasto excessivo de memória, mas ainda garantindo um acesso rápido, é através do uso de tabelas de dispersão (*hash table*) que discutiremos a seguir.

17.1. Idéia central

A idéia central por trás de uma tabela de dispersão é identificar, na chave de busca, quais as partes significativas. Na PUC-Rio, por exemplo, além do dígito de controle, alguns outros dígitos do número de matrícula têm significados especiais, conforme ilustra a Figura 17.1.

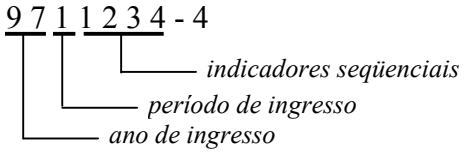

Figura 17.1: Significado dos dígitos do número da matrícula.

Numa turma de aluno, é comum existirem vários alunos com o mesmo ano e período de ingresso. Portanto, esses três primeiros dígitos não são bons candidatos para identificar individualmente cada aluno. Reduzimos nosso problema para uma chave com os quatro dígitos seqüenciais. Podemos ir além e constatar que os números seqüenciais mais significativos são os últimos, pois num universo de uma turma de alunos, o dígito que representa a unidade varia mais do que o dígito que representa o milhar.

Desta forma, podemos usar um número de matrícula parcial, de acordo com a dimensão que queremos que tenha nossa tabela (ou nosso vetor). Por exemplo, para dimensionarmos nossa tabela com apenas 100 elementos, podemos usar apenas os últimos dois dígitos seqüenciais do número de matrícula. A tabela pode então ser declarada por: `Aluno* tab[100]`.

Para acessarmos o nome do aluno cujo número de matrícula é dado por `mat`, usamos como índice da tabela apenas os dois últimos dígitos. Isso pode ser conseguido aplicando-se o operador modulo (`%`): `vet[mat%100]->nome`.

Desta forma, o uso de memória excedente é pequeno e o acesso a um determinado aluno, a partir do número de matrícula, continua imediato. O problema que surge é que provavelmente existirão dois ou mais alunos da turma que apresentarão os mesmos últimos dois dígitos no número de matrícula. Dizemos que há uma *colisão*, pois alunos diferentes são mapeados para o mesmo índice da tabela. Para que a estrutura funcione de maneira adequada, temos que resolver esse problema, tratando as colisões.

Existem diferentes métodos para tratarmos as colisões em tabelas de dispersão, e estudaremos esses métodos mais adiante. No momento, vale salientar que não há como eliminar completamente a ocorrência de colisões em tabelas de dispersão. Devemos minimizar as colisões e usar um método que, mesmo com colisões, saibamos identificar cada elemento da tabela individualmente.

17.2. Função de dispersão

A função de dispersão (função de *hash*) mapeia uma chave de busca num índice da tabela. Por exemplo, no caso exemplificado acima, adotamos como função de *hash* a utilização dos dois últimos dígitos do número de matrícula. A implementação dessa função recebe como parâmetro de entrada a chave de busca e retorna um índice da tabela. No caso d a chave de busca ser um inteiro representando o número de matrícula, essa função pode ser dada por.

```
int hash (int mat)
{
   return (mat%100);
}
```

Podemos generalizar essa função para tabelas de dispersão com dimensão `N`. Basta avaliar o modulo do número de matrícula por `N`:

```
int hash (int mat)
{
   return (mat%N);
}
```

Uma função de *hash* deve, sempre que possível, apresentar as seguintes propriedades:

- Ser eficientemente avaliada: isto é necessário para termos acesso rápido, pois temos que avaliar a função de *hash* para determinarmos a posição onde o elemento se encontra armazenado na tabela.
- Espalhar bem as chaves de busca: isto é necessário para minimizarmos as ocorrências de colisões. Como veremos, o tratamento de colisões requer um procedimento adicional para encontrarmos o elemento. Se a função de *hash* resulta em muitas colisões, perdemos o acesso rápido aos elementos. Um exemplo de função de *hash* ruim seria usar, como índice da tabela, os dois dígitos iniciais do número de matrícula – todos os alunos iriam ser mapeados para apenas três ou quatro índices da tabela.

Ainda para minimizarmos o número de colisões, a dimensão da tabela deve guardar uma folga em relação ao número de elementos efetivamente armazenados. Como regra empírica, não devemos permitir que a tabela tenha uma taxa de ocupação superior a

75%. Uma taxa de 50% em geral traz bons resultados. Uma taxa menor que 25% pode representar um gasto excessivo de memória.

17.3. Tratamento de colisão

Existem diversas estratégias para tratarmos as eventuais colisões que surgem quando duas ou mais chaves de busca são mapeadas para um mesmo índice da tabela de *hash*. Nesta seção, vamos apresentar algumas dessas estratégias comumente usadas. Para cada uma das estratégias, vamos apresentar as duas principais funções de manipulação de tabelas de dispersão: a função que busca um elemento na tabela e a função que insere ou modifica um elemento. Nessas implementações, vamos considerar a existência da função de dispersão que mapeia o número de matrícula num índice da tabela, vista na seção anterior.

Em todas as estratégias, a tabela de dispersão em si é representada por um vetor de ponteiros para a estrutura que representa a informação a ser armazenada, no caso `Aluno`. Podemos definir um tipo que representa a tabela por:

```
#define N 100
typedef Aluno* Hash[N];
```

Uso da posição consecutiva livre

Nas duas primeiras estratégias que discutiremos, os elementos que colidem são armazenados em outros índices, ainda não ocupados, da própria tabela. A escolha da posição ainda não ocupada para armazenar um elemento que colide diferencia as estratégias que iremos discutir. Numa primeira estratégia, se a função de dispersão mapeia para um índice já ocupado, procuramos o próximo (usando incremento circular) índice livre da tabela para armazenar o novo elemento. A Figura 17.2 ilustra essa estratégia. Nessa figura, os índices da tabela que não têm elementos associados são preenchidos com o valor `NULL`.

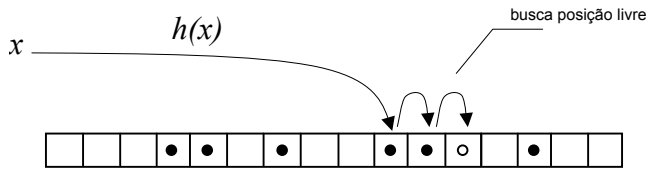

Figura 17.2: Tratamento de colisões usando próxima posição livre.

Vale lembrar que uma tabela de dispersão nunca terá todos os elementos preenchidos (já mencionamos que uma ocupação acima de 75% eleva o número de colisões, descaracterizando a idéia central da estrutura). Portanto, podemos garantir que sempre existirá uma posição livre na tabela.

Na operação de busca, considerando a existência de uma tabela já construída, se uma chave x for mapeada pela função de dispersão (função de $hash - h$) para um determinado índice *h(x)*, procuramos a ocorrência do elemento a partir desse índice, até que o elemento seja encontrado ou que uma posição vazia seja encontrada. Uma possível implementação é mostrada a seguir. Essa função de busca recebe, além da

tabela, a chave de busca do elemento que se busca, e tem como valor de retorno o ponteiro do elemento, se encontrado, ou `NULL` no caso do elemento não estar presente na tabela.

```
Aluno* busca (Hash tab, int mat)
{
   int h = hash(mat);
   while (tab[h] != NULL) {
      if (tab[h]->mat == mat)
         return tab[h];
      h = (h+1) % N;
   }
   return NULL;
}
```

Devemos notar que a existência de algum elemento mapeado para o mesmo índice não garante que o elemento que buscamos esteja presente. A partir do índice mapeado, temos que buscar o elemento utilizando, como chave de comparação, a real chave de busca, isto é, o número de matrícula completo.

A função que insere ou modifica um determinado elemento também é simples. Fazemos o mapeamento da chave de busca (no caso, número de matrícula) através da função de dispersão e verificamos se o elemento já existe na tabela. Se o elemento existir, modificamos o seu conteúdo; se não existir, inserimos um novo na primeira posição que encontrarmos na tabela, a partir do índice mapeado. Uma possível implementação dessa função é mostrada a seguir. Essa função recebe como parâmetros a tabela e os dados do elemento sendo inserido (ou os novos dados de um elemento já existente). A função tem como valor de retorno o ponteiro do aluno modificado ou do novo aluno inserido.

```
Aluno* insere (Hash tab, int mat, char* nome, char* email, char turma)
{
   int h = hash(mat);
   while (tab[h] != NULL) {
      if (tab[h]->mat == mat)
         break;
      h = (h+1) % N;
   }
   if (tab[h]==NULL) {        /* não encontrou o elemento */
      tab[h] = (Aluno*) malloc(sizeof(Aluno));
      tab[h]->mat = mat;
   }
   /* atribui informação */
   strcpy(tab[h]->nome,nome);
   strcpy(tab[h]->email,email);
   tab[h]->turma = turma;
   return tab[h];
}
```

Apesar de bastante simples, essa estratégia tende a concentrar os lugares ocupados na tabela, enquanto que o ideal seria dispersar. Uma estratégia que visa melhorar essa concentração é conhecida como “dispersão dupla” (*double hash*) e será apresentada a seguir.

Uso de uma segunda função de dispersão

Para evitar a concentração de posições ocupadas na tabela, essa segunda estratégia faz uma variação na forma de procurar uma posição livre a fim armazenar o elemento que

colidiu. Aqui, usamos uma segunda função de dispersão, *h'*. Para chaves de busca dadas por números inteiros, uma possível segunda função de dispersão é definida por:

$$h'(x) = N - 2 - x\%(N - 2)$$

Nesta fórmula, *x* representa a chave de busca e *N* a dimensão da tabela. De posse dessa segunda função, procuramos uma posição livre na tabela com incrementos, ainda circulares, dados por *h'(x)*. Isto é, em vez de tentarmos *(h(x)+1)%N*, tentamos *(h(x)+h'(x))%N*. Dois cuidados devem ser tomados na escolha dessa segunda função de dispersão: primeiro, ela nunca pode retornar zero, pois isso não varia com que o índice fosse incrementado; segundo, de preferência, ela não pode retornar um número divisor da dimensão da tabela, pois isso nos limitaria a procurar uma posição livre num sub-conjunto restrito dos índices da tabela.

A implementação da função de busca com essa estratégia é uma pequena variação da função de busca apresentada para a estratégia anterior.

```
int hash2 (int mat)
{
   return N - 2 - mat%(N-2);
}

Aluno* busca (Hash tab, int mat)
{
   int h = hash(mat);
   int h2 = hash2(mat);
   while (tab[h] != NULL) {
      if (tab[h]->mat == mat)
         return tab[h];
      h = (h+h2) % N;
   }
   return NULL;
}
```

Exercício: Implemente a função para inserir (ou modificar) um elemento usando a estratégia de uma segunda função de dispersão.

Uso de listas encadeadas

Uma estratégia diferente, mas ainda simples, consiste em fazer com que cada elemento da tabela *hash* represente um ponteiro para uma lista encadeada. Todos os elementos mapeados para um mesmo índice seriam armazenados na lista encadeada. A Figura 17.1 ilustra essa estratégia. Nessa figura, os índices da tabela que não têm elementos associados representam listas vazias.

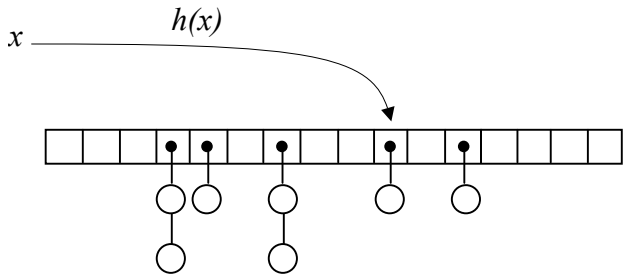

Figura 17.3: Tratamento de colisões com lista encadeada.

Com essa estratégia, cada elemento armazenado na tabela será um elemento de uma lista encadeada. Portanto, devemos prever, na estrutura da informação, um ponteiro adicional para o próximo elemento da lista. Nossa estrutura de aluno passa a ser dada por:

```
struct aluno {
   int mat;
   char nome[81];
   char turma;
   char email[41];
   struct aluno* prox;   /* encadeamento na lista de colisão */
};
typedef struct aluno Aluno;
```

Na operação de busca, procuramos a ocorrência do elemento na lista representada no índice mapeado pela função de dispersão. Uma possível implementação é mostrada a seguir.

```
Aluno* busca (Hash tab, int mat)
{
   int h = hash(mat);
   Aluno* a = tab[h];
   while (a != NULL) {
      if (a->mat == mat)
         return a;
      a = a->prox;
   }
   return NULL;
}
```

A função que insere ou modifica um determinado elemento também é simples e pode ser dada por:

```
Aluno* insere (Hash tab, int mat, char* nome, char turma)
{
   int h = hash(mat);
   Aluno* p = NULL;       /* ponteiro para anterior */
   Aluno* a = tab[h];
   while (a != NULL) {
      if (a->mat == mat)
         break;
      p = a;
      a = a->prox;
   }
   if (a==NULL) {        /* não encontrou o elemento */
      a = (Aluno*) malloc(sizeof(Aluno));
      a->mat = mat;
      a->prox = NULL;
      if (p==NULL)
         tab[h] = a;
      else
         p->prox = a;
   }
   /* atribui informação */
   strcpy(a->nome,nome);
   a->turma = turma;
   return a;
}
```

Exercício: Faça um programa que utilize as funções de tabelas de dispersão vistas acima.

17.4. Exemplo: Número de Ocorrências de Palavras

Para exemplificar o uso de tabelas de dispersão, vamos considerar o desenvolvimento de um programa para exibir quantas vezes cada palavra foi utilizada em um dado texto. A saída do programa será uma lista de palavras, em ordem decrescente do número de vezes que cada palavra ocorre no texto de entrada. Para simplificar, não consideraremos caracteres acentuados.

Projeto: "Dividir para conquistar"

A melhor estratégia para desenvolvermos programas é dividirmos um problema grande em diversos problemas menores. Uma aplicação deve ser construída através de módulos independentes. Cada módulo é projetado para a realização de tarefas específicas. Um segundo módulo, que é cliente, não precisa conhecer detalhes de como o primeiro foi implementado; o cliente precisa apenas saber a funcionalidade oferecida pelo módulo que oferece os serviços. Dentro de cada módulo, a realização da tarefa é dividida entre várias pequenas funções. Mais uma vez, vale a mesma regra de encapsulamento: funções clientes não precisam conhecer detalhes de implementação das funções que oferecem os serviços. Dessa forma, aumentamos o potencial de re-uso do código e facilitamos o entendimento e a manutenção do programa.

O programa para contar o uso das palavras é um programa relativamente simples, que não precisa ser subdividido em módulos para ser construído. Aqui, vamos projetar o programa identificando as diversas funções necessárias para a construção do programa como um todo. Cada função tem sua finalidade específica e o programa principal (a função `main`) fará uso dessas funções.

Vamos considerar que uma palavra se caracteriza por uma seqüência de uma ou mais letras (maiúsculas ou minúsculas). Para contar o número de ocorrências de cada palavra, podemos armazenar as palavras lidas numa tabela de dispersão, usando a própria palavra como chave de busca. Guardaremos na estrutura de dados quantas vezes cada palavra foi encontrada. Para isso, podemos prever a construção de uma função que acessa uma palavra armazenada na tabela; se a palavra ainda não existir, a função armazena uma nova palavra na tabela. Dessa forma, para cada palavra lida, conseguiremos incrementar o número de ocorrências. Para exibir as ocorrências em ordem decrescente, criaremos um vetor e armazenaremos todas as palavras que existem na tabela de dispersão no vetor. Esse vetor pode então ser ordenado e seu conteúdo exibido.

Tipo dos dados

Conforme discutido acima, usaremos uma tabela de dispersão para contar o número de ocorrências de cada palavra no texto. Vamos optar por empregar a estratégia que usa lista encadeada para o tratamento de colisões. Dessa forma, a dimensão da tabela de dispersão não compromete o número máximo de palavras distintas (no entanto, a dimensão da tabela não pode ser muito justa em relação ao número de elementos armazenados, pois aumentaria o número de colisões, degradando o desempenho). A estrutura que define a tabela de dispersão pode ser dada por:

```
#define NPAL  64   /* dimensão máxima de cada palavra */
#define NTAB 127   /* dimensão da tabela de dispersão */

/* tipo que representa cada palavra */
struct palavra {
   char pal[NPAL];
   int  n;
   struct palavra* prox;    /* tratamento de colisao com listas */
};
typedef struct palavra Palavra;

/* tipo que representa a tabela de dispersão */
typedef Palavra* Hash[NTAB];
```

Leitura de palavras

A primeira função que vamos discutir é responsável por capturar a próxima seqüência de letras do arquivo texto. Essa função receberá como parâmetros o ponteiro para o arquivo de entrada e a cadeia de caracteres que armazenará a palavra capturada. A função tem como valor de retorno um inteiro que indica se a leitura foi bem sucedida (`1`) ou não (`0`). A próxima palavra é capturada pulando os caracteres que não são letras e, então, capturando a seqüência de letras do arquivo. Para identificar se um caractere é ou não letra, usaremos a função `isalpha` disponibilizada pela interface `ctype.h`.

```
int le_palavra (FILE* fp, char* s)
{
   int i = 0;
   int c;

   /* pula caracteres que nao sao letras */
   while ((c = fgetc(fp)) != EOF) {
      if (isalpha(c))
         break;
   };

   if (c == EOF)
      return 0;
   else
      s[i++] = c;      /* primeira letra já foi capturada */

   /* lê os próximos caracteres que são letras */
   while ( i<NPAL-1 && (c = fgetc(fp)) != EOF && isalpha(c))
      s[i++] = c;
   s[i] = '\0';

   return 1;
}
```

Tabela de dispersão com cadeia de caracteres

Devemos implementar as funções responsáveis por construir e manipular a tabela de dispersão. A primeira função que precisamos é responsável por inicializar a tabela, atribuindo o valor `NULL` para cada elemento.

```
void inicializa (Hash tab)
{
   int i;
   for (i=0; i<NTAB; i++)
      tab[i] = NULL;
}
```

Também precisamos definir uma função de dispersão, responsável por mapear a chave de busca, uma cadeia de caracteres, em um índice da tabela. Uma função de dispersão simples para cadeia de caracteres consiste em somar os código dos caracteres que compõem a cadeia e tirar o módulo dessa soma para se obter o índice da tabela. A implementação abaixo ilustra essa função.

```
int hash (char* s)
{
   int i;
   int total = 0;
   for (i=0; s[i]!='\0'; i++)
      total += s[i];
   return total % NTAB;
}
```

Precisamos ainda da função que acessa os elementos armazenados na tabela. Criaremos uma função que, dada uma palavra (chave de busca), fornece como valor de retorno o ponteiro da estrutura `Palavra` associada. Se a palavra ainda não existir na tabela, essa função cria uma nova palavra e fornece como retorno essa nova palavra criada.

```
Palavra *acessa (Hash tab, char* s)
{
   int h = hash(s);
   Palavra* p;
   for (p=tab[h]; p!=NULL; p=p->prox) {
      if (strcmp(p->pal,s) == 0)
         return p;
   }
   /* insere nova palavra no inicio da lista */
   p = (Palavra*) malloc(sizeof(Palavra));
   strcpy(p->pal,s);
   p->n = 0;
   p->prox = tab[h];
   tab[h] = p;
   return p;
}
```

Dessa forma, a função cliente será responsável por acessar cada palavra e incrementar o seu número de ocorrências. Transcrevemos abaixo o trecho da função principal reponsável por fazer essa contagem (a função completa será mostrada mais adiante).

```
...
inicializa(tab);
while (le_palavra(fp,s)) {
   Palavra* p = acessa(tab,s);
   p->n++;
}
...
```

Com a execução desse trecho de código, cada palavra encontrada no texto de entrada será armazenada na tabela, associada ao número de vezes de sua ocorrência. Resta-nos arrumar o resultado obtido para podermos exibir as palavras em ordem decrescente do número de ocorrências.

Exibição do resultado ordenado

Para colocarmos o resultado na ordem desejada, criaremos dinamicamente um vetor para armazenar as palavras. Optaremos por construir um vetor de ponteiros para a estrutura `Palavra`. Esse vetor será então ordenado em ordem decrescente do número de ocorrências de cada palavra; se duas palavras tiverem o mesmo número de ocorrências, usaremos a ordem alfabética como critério de desempate.

Para criar o vetor, precisamos conhecer o número de palavras armazenadas na tabela de dispersão. Podemos implementar uma função que percorre a tabela e conta o número de palavras existentes. Essa função pode ser dada por:

```
int conta_elems (Hash tab)
{
   int i;
   int total = 0;
   Palavra* p;
   for (i=0; i<NTAB; i++) {
      for (p=tab[i]; p!=NULL; p=p->prox)
         total++;
   }
   return total;
}
```

Podemos agora implementar a função que cria dinamicamente o vetor de ponteiros. Em seguida, a função percorre os elementos da tabela e preenche o conteúdo do vetor. Essa

função recebe como parâmetros de entrada o número de elementos e a tabela de dispersão.

```
Palavra** cria_vetor (int n, Hash tab)
{
   int i, j=0;
   Palavra* p;
   Palavra** vet = (Palavra**) malloc(n*sizeof(Palavra*));

   /* percorre tabela preenchendo vetor */
   for (i=0; i<NTAB; i++) {
      for (p=tab[i]; p!=NULL; p=p->prox)
         vet[j++] = p;
   }
   return vet;
}
```

Para ordenar o vetor (de poteiros para a estrutura `Palavra`) utilizaremos a função `qsort` da biblioteca padrão. Precisamos então definir a função de comparação, que é mostrada abaixo.

```
int compara (const void* v1, const void* v2)
{
   Palavra** pp1 = (Palavra**)v1;
   Palavra** pp2 = (Palavra**)v2;
   Palavra* p1 = *pp1;
   Palavra* p2 = *pp2;

   if (p1->n > p2->n) return -1;
   else if (p1->n < p2->n) return 1;
   else return strcmp(p1->pal,p2->pal);
}
```

Por fim, podemos escrever a função que, dada a tabela de dispersão já preenchida e utilizando as funções mostradas acima, conta o número de elementos, cria o vetor, ordena-o e exibe o resultado na ordem desejada. Ao final, a função libera o vetor criado dinamicamente.

```
void imprime (Hash tab)
{
   int i;
   int n;
   Palavra** vet;

   /* cria e ordena vetor */
   n = conta_elems(tab);
   vet = cria_vetor(n,tab);
   qsort(vet,n,sizeof(Palavra*),compara);

   /* imprime ocorrencias */
   for (i=0; i<n; i++)
      printf("%s = %d\n",vet[i]->pal,vet[i]->n);

   /* libera vetor */
   free(vet);
}
```

Função principal

Uma possível função principal desse programa é mostrada a seguir. Esse programa espera receber como dado de entrada o nome do arquivo cujas palavras queremos contar o número de ocorrências. Para exemplificar a utilização dos parâmetros da função

principal, utilizamos esses parâmetros para receber o nome do arquivo de entrada (para detalhes, veja seção 6.3).

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

...  /* funções auxiliares mostradas acima */

int main (int argc, char** argv)
{
   FILE* fp;
   Hash tab;
   char s[NPAL];

   if (argc != 2) {
      printf("Arquivo de entrada nao fornecido.\n");
    return 0;
   }

   /* abre arquivo para leitura */
   fp = fopen(argv[1],"rt");
   if (fp == NULL) {
      printf("Erro na abertura do arquivo.\n");
    return 0;
   }

   /* conta ocorrencia das palavras */
   inicializa(tab);
   while (le_palavra(fp,s)) {
      Palavra* p = acessa(tab,s);
      p->n++;
   }

   /* imprime ordenado */
   imprime (tab);

   return 0;
}
```

17.5. Uso de callbacks **

No programa da seção anterior, precisamos implementar duas funções que percorrem os elementos da tabela de dispersão: uma para contar o número de elementos e outra para preencher o vetor. Em todas as estruturas de dados, é muito comum necessitarmos de funções que percorrem os elementos, executando uma ação específica para cada elemento. Como um exemplo adicional, podemos imaginar uma função para imprimir os elementos na ordem em que eles aparecem na tabela de dispersão. Ainda usando o exemplo da seção anterior, poderíamos ter:

```
void imprime_tabela (Hash tab)
{
   int i;
   Palavra* p;

   for (i=0; i<NTAB; i++) {
       for (p=tab[i]; p!=NULL; p=p->prox)
          printf("%s = %d\n",p->pal,p->n);
   }
}
```

Podemos observar que as estruturas dessas funções são as mesmas. Como também teriam as mesmas estruturas funções para percorrer os elementos de uma lista encadeada, de uma árvore, etc.

Nesses casos, podemos separar a função que percorre os elementos da ação que realizamos a cada elemento. Assim, a função que percorre os elementos é única e pode ser usada para diversos fins. A ação que deve ser executada é passada como parâmetro, via um ponteiro para uma função. Essa função é usualmente chamada de *callback* pois é uma função do cliente (quem usa a função que percorre os elementos) que é "chamada de volta" a cada elemento encontrado na estrutura de dados. Usualmente essa função *callback* recebe como parâmetro o elemento encontrado na estrutura. No nosso exemplo, como os elementos são ponteiros para a estrutura `Palavra`, a função receberia o ponteiro para cada palavra encontrada na tabela.

Uma função genérica para percorrer os elementos da tabela de dispersão do nosso exemplo pode ser dada por:

```
void percorre (Hash tab, void (*cb)(Palavra*) )
{
   int i;
   Palavra* p;

   for (i=0; i<NTAB; i++) {
       for (p=tab[i]; p!=NULL; p=p->prox)
          cb(p);
   }
}
```

Para ilustrar sua utlização, podemos usar essa função para imprimir os elementos. Para tanto, devemos escrever a função que executa a ação de imprimir cada elemento. Essa função pode ser dada por:

```
void imprime_elemento (Palavra* p)
{
   printf("%s = %d\n",p->pal,p->n);
}
```

Assim, para imprimir os elementos da tabela bastaria chamar a função `percorre` com a ação acima passada como parâmetro.

```
...
percorre(tab,imprime_elemento);
...
```

Essa mesma função `percorre` pode ser usada para, por exemplo, contar o número de elementos que existe armazenado na tabela. A ação associada aqui precisa apenas incrementar um contador do número de vezes que a *callback* é chamada. Para tanto, devemos usar uma variável global que representa esse contador e fazer a *callback* incrementar esse contador cada vez que for chamada. Nesse caso, o ponteiro do elemento passado como parâmetro para a *callback* não é utilizado, pois o incremento ao contador independe do elemento. Assumindo que `Total` é uma variável global inicializada com o valor zero, a ação para contar o número de elementos é dada simplesmente por:

```
void conta_elemento (Palavra* p)
{
   Total++;
}
```

Já mencionamos que o uso de variáveis globais deve, sempre que possível, ser evitado, pois seu uso indiscriminado torna um programa ilegível e difícil de ser mantido. Para evitar o uso de variáveis globais nessas funções *callbacks* devemos arrumar um meio de transferir, para a função *callback*, um dado do cliente. A função que percorre os elementos não manipula esse dado, apenas o transfere para a função *callback*. Como não sabemos *a priori* o tipo de dado que será necessário, definimos a *callback* recebendo dois parâmetros: o elemento sendo visitado e um ponteiro genérico (`void*`). O cliente chama a função que percorre os elementos passando como parâmetros a função *callback* e um ponteiro a ser repassado para essa mesma *callback*.

Vamos exemplificar o uso dessa estratégia re-implementando a função que percorre os elementos.

```
void percorre (Hash tab, void (*cb)(Palavra*, void*), void* dado)
{
   int i;
   Palavra* p;

   for (i=0; i<NTAB; i++) {
       for (p=tab[i]; p!=NULL; p=p->prox)
          cb(p,dado);   /* passa para a callback o ponteiro recebido */
   }
}
```

Agora, podemos usar essa nova versão da função para contar o número de elementos, sem usar variável global. Primeiro temos que definir a *callback*, que, nesse caso, receberá um ponteiro para um inteiro que representa o contador.

```
void conta_elemento (Palavra* p, void* dado)
{
   int *contador = (int*)dado;
   (*contador)++;
}
```

Por fim, uma função que conta o número de elemento, usando as funções acima, é mostrada a seguir.

```
int total_elementos (Hash tab)
{
   int total = 0;
   percorre(tab,conta_elemento,&total);
   return total;
}
```