MAC0122 Princípios de Desenvolvimento de Algoritmos

Tabelas de dispersão (hash tables)

hash = picadinho
to hash = picar, fazer picadinho, misturar, confundir

Este é um resumo de parte do capítulo 14 do livro de Sedgewick e da seção 11.2 do livro de Roberts.

Uma tabela de dispersão (= *hash table*) é uma maneira muito popular de organizar uma tabela de símbolos. Pode-se dizer que tabelas de dispersão são uma generalização do ideia de endereçamento direto. Elas foram inventadas para funcionar bem *em média*, ou seja, na maioria dos casos; seu desempenho no pior caso é lamentável.

A implementação de uma tabela de dispersão envolve muitas sutilezas que afetam sua eficiência, embora não afetem sua correção. Nosso objetivo nesta página é apenas chamar a atenção para essas sutilezas; não pretendemos entrar em detalhes sobre os truques e heurísticas que se usam para enfrentadas as dificuldades.

Hashing

Toda tabela de símbolos tem um **universo de chaves**, que é o conjunto de todas as possíveis chaves. O conjunto das chaves efetivamente usadas em uma determinada aplicação é, em geral, apenas uma pequena parte do universo de chaves. Nessas circunstâncias, faz sentido usar uma tabela de tamanho bem menor que o tamanho do universo de chaves.

Denotaremos o tamanho de nossa tabela por `M`. A tabela terá a forma `tab[0..M-1]`. Temos que inventar agora uma maneira de *mapear* o universo de chaves no conjunto de índices da tabela. Esse é o papel da **função de espalhamento** (= *hash function*). A função de espalhamento

recebe uma chave `v` e devolve um número inteiro `h` no intervalo `0..M-1`.

O número `h` é o **código de espalhamento** (= *hash code*) da chave `v`. É fundamental que a função de espalhamento seja uma função no sentido matemático do termo, isto é, que para cada chave `v` a função devolva sempre *o mesmo* código de espalhamento. Além disso, uma boa função de espalhamento "espalha" as chaves uniformemente pelo conjunto de índices.

Como o número de chaves é em geral maior que `M`, é inevitável que a função de espalhamento leve várias chaves diferentes no mesmo índice. Dizemos que há uma **colisão** quando duas chaves diferentes são levadas no mesmo índice.

EXEMPLO. Suponha que o universo de chaves é o conjunto dos números inteiros que vai de 100001 a 9999999 (veja o exemplo no capítulo de tabelas de símbolos). Suponha que `M = 100`. Vamos adotar os dois últimos dígitos da chave como código de espalhamento. Em outras palavras, o código de `v` é o resto da divisão de `v` por `100`:

chave	código
123456	56
7531	31
3677756	56

(Poderíamos igualmente bem ter adotado os dois *primeiros* dígitos da chave, ou talvez o dígito dos milhares junto com o digito das centenas.)

O exemplo ilustra um função de espalhamento **modular**: a função leva cada chave `v` no resto da divisão de `v` por `M`. Em notação C, a função pode ser definida assim:

```
hash(v, M) = v % M
```

Resta saber o que fazer com as colisões. Há várias maneiras de lidar com essa questão, como veremos adiante.

Boas e más funções de espalhamento

Qualquer função que leva qualquer chave no intervalo `0..M-1` de índices serve como função de espalhamento. Mas uma tal função só é *eficiente* se "espalha" as chaves pelo intervalo de índices de maneira razoavelmente uniforme. Por exemplo, se as chaves são números inteiros e todas são múltiplos de 100 (ou seja, todas terminam com "00") então `v % 100` é uma *péssima* função de espalhamento. Mesmo que as chaves não sejam múltiplas de 100, a função `v % 100` não é muito boa: basta imaginar o que acontece se, por algum motivo, o penúltimo dígito decimal da chave é sempre inferior a 5.

Vamos examinar a seguir alguns casos mais sutis de funções de espalhamento inadequadas. Suponha que nossas chaves são números inteiros pares, ou seja, que todas são divisíveis por 2. Suponha que `M` também é par. É fácil verificar então que todos os índices produzidos pela função modular

```
hash(v, M) = v % M
```

serão pares. Assim, as posições `1`, `3`, `5`, `. . .`, `M-2` jamais serão usadas! O fenômeno existe mesmo em situações mais sutis: nem todas as chaves são pares, mas há muito mais chaves pares que ímpares.

De maneira mais geral: se `M` for divisível por `k` (isto é, se `M % k == 0`) então, o número `v % M` será divisível por `k` sempre que `v` for divisível por `k`. Em vista dessa observação,

é recomendável que `M` seja um número primo.

Sedgewick sugere escolher `M` da seguinte maneira. Comece por escolher uma potência de 2 que esteja próxima do valor desejado de `M` (ou seja, de um valor que seja apropriado para os seus dados). Depois, adote para `M` o número primo que esteja logo abaixo da potência escolhida.

k	2^k	M
7	128	127
8	256	251
9	512	509
10	1024	1021
11	2048	2039
12	4096	4093
13	8192	8191
14	16384	16381
15	32768	32749
16	65536	65521
17	131072	131071
18	262144	262139

Exercícios

1. Suponha que `v` e `M` são divisíveis por `k`. Mostre que `v % M` também é divisível por `k`.

Resolvendo colisões por meio de listas encadeadas

Uma solução popular para resolver colisões é conhecida como *separate chaining*: para cada índice `h` da tabela há uma lista encadeada que armazena todos os objetos que a função de espalhamento leva em `h`. Essa solução é muito boa se cada uma das "listas de colisão" resultar curta. Se o número total de objetos for `N`, o comprimento de cada lista deveria, idealmente, estar

próximo de `N/M`.

De acordo com Sedgewick, uma boa regra prática é escolher `M` de modo que o valor de `N/M` fique entre 5 e 10.

Nas condições do exemplo acima, suponha que nossa tabela deve estar preparada para armazenar até 50000 objetos. Parece razoável, então, adotar um número primo entre 5000 e 10000 para valor de `M`. Adotemos, pois, o número 8191. A implementação a seguir (compare com o programa 14.3 de Sedgewick) usa *separate chaining* para resolver colisões:

```
#define M 8191
// Tamanho da tabela.

#define hash(v, M) (v % M)
// Transforma uma chave v em um índice no intervalo 0..M-1.

typedef struct {
   int    chave;
   string valor;
} tipoObjeto;

tipoObjeto objetonulo;
objetonulo.chave = 0;
// Todas as chaves "válidas" são estritamente positivas.

// Definição de um nó das listas de colisões.
typedef struct STnode *link;
struct STnode {
   tipoObjeto obj;
   link       next;
} ;

// Tabela que aponta para as M listas de colisões.
link *tab;



// Inicializa uma tabela de símbolos que, espera-se, armazenará
// cerca de 50000 objetos. A espinha dorsal da tabela será um
// vetor tab[0..M-1].
//
void STinit()
{
   int h;
   tab = malloc(M * sizeof (link));
   for (h = 0; h < M; h++)
      tab[h] = NULL;
}

// Insere obj na tabela de símbolos.
//
void STinsert(tipoObjeto obj)
{
   int h, v;
   v = obj.chave;
   h = hash(v, M);
   link novo = malloc(sizeof (STnode));
   novo->obj = obj;
   novo->next = tab[h];
   tab[h] = novo;
}

// Devolve um objeto cuja chave é v. Se tal objeto não existe,
```

```
// a função devolve um objeto fictício com chave nula.
//
tipoObjeto STsearch(int v)
{
   link t;
   int h;
   h = hash(v, M);
   for (t = tab[h]; t != NULL; t = t->next)
      if (t->obj.chave == v) break;
   if (t != NULL) return t->obj;
   return objetonulo;
}
```

Exercícios

2. [Roberts 11.3] Suponha dada uma tabela de dispersão como a descrita acima, com colisões resolvidas por listas encadeadas. Escreva uma função que calcule e devolva a média e o desvio padrão dos comprimentos das listas.

3. [Roberts 11.4] Implemente uma função que remova da tabela de símbolos todas os objetos que têm uma determinada chave. A função terá protótipo

   ```
   STdelete(int v);
   ```

4. [Sedg 14.19] Escreva um programa que insira N inteiros aleatórios em uma tabela de símbolos com `M` = $N/100$ posições. Use a função de espalhamento `v % M` e resolva colisões por meio de listas encadeadas. Em seguida, determine o comprimento da lista mais curta e o da lista mais longa. Repita o experimento para $N = 10^3$, $N = 10^4$, $N = 10^5$, $N = 10^6$.

Funções de espalhamento modulares para cadeias de caracteres

Até aqui, todos os exemplos usaram chaves numéricas. O que fazer se a chave é uma cadeia de caracteres? Como calcular uma função de espalhamento nesse caso?

É muito fácil. Cada caracter é um número entre 0 e 255. Portanto, uma cadeia não vazia pode ser interpretada como a representação em base 256 de um número. Suponha que `s` é uma cadeia de comprimento 2. Então o número correspondente é

`s[0]` $*$ 256 $+$ `s[1]` .

Por exemplo, se `s` é `"AB"` então o número correspondente é `65` $*$ 256 $+$ `66` , ou seja, 16706.

Para fazer os cálculos de maneira eficiente, basta usar o método de Horner:

```
int hash(string v, int M) {
   int i, h = v[0];
   for (i = 1; v[i] != '\0'; i++)
      h = h * 256 + v[i];
   return h % M;
}
```

A base da representação nem precisa ser igual ao número, 256, de valores possíveis de cada caracter. Para que o espalhamento seja melhor, recomenda-se (veja o programa 14.1 de Sedgewick) usar um número primo como base:

```
int hash(string v, int M) {
   int i, h = v[0];
   for (i = 1; v[i] != '\0'; i++)
      h = h * 251 + v[i];
   return h % M;
}
```

Um último detalhe: para evitar *overflow*, convém reescrever a função da seguinte maneira, inteiramente equivalente:

```
int hash(string v, int M) {
   int i, h = v[0];
   for (i = 1; v[i] != '\0'; i++)
      h = (h * 251 + v[i]) % M;
   return h;
}
```

(Estamos supondo que `M` não é muito grande. Caso contrário, `h` pode se aproximar de `INT_MAX` e o valor da expressão `h*251 + v[i]` pode ser *negativo* em virtude de *overflow*, o que seria desastroso. Se `M` for muito grande, é melhor declarar `h` como `unsigned` e não `int`.)

Exercícios

5. [Sedg 14.10] Suponha que `a`, `b` e `x` são inteiros não negativos e `M` é um inteiro positivo. Mostre que

   `((a % M) * x + b) % M  ==  (a*x + b) % M .`

6. Considere o polinômio $a[0]x^0 + a[1]x^1 + \ldots + a[n]x^n$. Imagine um algoritmo que calcula o valor do polinômio num dado ponto `x` seguindo literalmente a forma da expressão acima. Quantas multiplicações e quantas somas o algoritmo fará? Agora imagine um algoritmo que calcula o valor do polinômio usando o método de Horner. Quantas multiplicações e quantas somas o algoritmo fará?

Exemplo

Digamos que meus objetos são as palavras de um livro (veja tarefa 7). Para cada palavra, quero saber quantas vezes ela aparece no livro. Esse tipo de objeto poderia ser representado assim:

```
typedef char *string;

typedef struct {
   string chave;
   int    ocorrencias;
} tipoObjeto;
```

Cada chave é uma palavra (não vazia) do livro. A função de inserção nesse caso é ligeiramente mais complicada que o usual: se a palavra a ser inserida já está na tabela de símbolos, não é necessário inseri-la novamente mas é preciso incrementar o seu contador de ocorrências.

Suponha que o livro tem cerca de 10000 palavras. Se usarmos um valor de `M` próximo de 1000 as listas de colisão terão comprimento 10 em média. Adotemos pois o número primo 1021 como valor de `M`. As funções de inserção e busca podem ser implementadas assim:

```
char stringvazia[1];
stringvazia[0] = '\0';
tipoObjeto objetonulo;
objetonulo.chave = stringvazia;

// Definição de um nó das listas de colisões.
typedef struct STnode *link;
struct STnode {
   tipoObjeto obj;
   link       next;
} ;

#define M 1021
// Tamanho da tabela.
```

```
// A tabela tab[0..M-1] apontará para as M listas de colisões.
link tab[M];

// Função de espalhamento: transforma uma chave não vazia v em um
// número no intervalo 0..M-1.
//
int hash(string v, int M) {
   int i, h = v[0];
   for (i = 1; v[i] != '\0'; i++)
      h = (h * 251 + v[i]) % M;
   return h;
}



// Inicializa uma tabela que apontará as M listas de colisões.
//
void STinit() {
   int h;
   for (h = 0; h < M; h++)
      tab[h] = NULL;
}

// Se o objeto obj já está na tabela de símbolos, a função
// insert incrementa o campo ocorrencias de obj. Senão,
// obj é inserido e seu contador é inicializado com 1.
//
void STinsert(tipoObjeto obj)
{
   string v = obj.chave;
   int h = hash(v, M);
   link t = tab[h];
   for (t = tab[h]; t != NULL; t = t->next)
      if (strcmp(t->obj.chave, v) == 0) break;
   if (t != NULL)
      t->obj.ocorrencias++;
   else {
      obj.ocorrencias = 1;
      link novo = malloc(sizeof (STnode));
      novo->obj = obj;
      novo->next = tab[h];
      tab[h] = novo;
   }
}

// A função search devolve um objeto obj que tenha chave v.
// Se tal objeto não existe, a função devolve um objeto cuja
// chave é a string vazia (ou seja, chave[0] == '\0').
//
tipoObjeto STsearch(string v)
{
   link t;
   int h = hash(v, M);
   for (t = tab[h]; t != NULL; t = t->next)
      if (strcmp(t->obj.chave, v) == 0) break;
   if (t != NULL) return t->obj;
   return objetonulo;
}
```

Resolvendo colisões por "linear probing"

Um outro método de resolução de colisões é conhecido como *linear probing*. Todos os objetos são armazenados em um vetor `tab[0..M-1]`. Quando ocorre uma colisão, procuramos a próxima posição vaga do vetor.

O seguinte exemplo (versão modificada do programa 14.4 de Sedgewick) ilustra o conceito:

```
int M;
// Tamanho da tabela (a ser definido durante a inicialização).

#define hash(v, M) (v % M)
// Transforma uma chave v em um índice no intervalo 0..M-1.

typedef struct {
   int       chave;
   string valor;
} tipoObjeto;

tipoObjeto objetonulo;
objetonulo.chave = 0;
// Todas as chaves "válidas" são estritamente positivas.

// A tabela tab[0..M-1] conterá todos os objetos.
tipoObjeto *tab;



// Inicializa uma tabela de símbolos que, espera-se, armazenará
// no máximo max objetos. A tabela residirá no vetor tab[0..M-1].
//
void STinit(int max)
{
   int h;
   M = 2 * max;
   tab = malloc(M * sizeof (tipoObjeto));
   for (h = 0; h < M; h++)
      tab[h] = objetonulo;
}

// A função insere obj na tabela de símbolos. Ela supõe
// que o número N de objetos na tabela de símbolos não é
// maior que M (ou seja, que N <= M).
//
void STinsert(tipoObjeto obj)
{
   int v = obj.chave;
   int h = hash(v, M);
   while (tab[h].chave != objetonulo.chave)
      h = (h + 1) % M;
   tab[h] = obj;
}

// Devolve um objeto cuja chave é v. Se tal objeto não existe,
// a função devolve um objeto fictício com chave nula.
// A função supõe que o número de objetos na tabela de símbolos
// não é maior que M.
//
tipoObjeto STsearch(int v)
{
   int h = hash(v, M);
   while (tab[h].chave != objetonulo.chave)
      if (tab[h].chave == v) return tab[h];
      else h = (h + 1) % M;
   return objetonulo;
}
```

Digamos que a tabela tem `M` posições e contém `N` objetos num dados instante. O número

`N/M`

é o **fator de carga** (= *load factor*) da tabela. É claro que esse número é menor ou igual a 1. Quanto maior o fator de carga, mais tempo as funções de busca e inserção vão consumir.

Exercícios

7. Qual o número médio de comparações entre chaves em uma execução da função `STinsert` acima?

URL of this site: www.ime.usp.br/~pf/mac0122-2002/
Last modified: Mon Oct 9 08:03:35 BRT 2017
Paulo Feofiloff
IME-USP