Hashing

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

"Hashing is used extensively in applications
and deserves recognition
as one of the cleverer inventions
of computer science."
— E.S. Roberts, *Programming Abstractions in C*

Este capítulo usa um pequeno problema de contagem como pretexto para introduzir a estrutura de dados conhecida como *tabela de dispersão* ou *hash table*. Essa estrutura é responsável por acelerar muitos algoritmos que envolvem consultas, inserções e remoções de uma tabela de dados.

Um problema de contagem

Suponha dado um fluxo de números inteiros positivos na entrada padrão `stdin`. Os números serão chamados *chaves*. As chaves estão em ordem arbitrária e há muitas chaves repetidas. Considere agora o problema de

> calcular o número de ocorrências de cada chave.

Por exemplo, o número de ocorrências de cada chave no fluxo `4998 9886 1933 1435 9886 1435 9886 7233 4998 7233 1435 1435 1004` é dado pela seguinte tabela (na primeira linha temos as diferentes chaves e na segunda o número de ocorrências de cada chave):

```
1004 1435 1933 4998 7233 9886
   1    4    1    2    2    3
```

Nosso problema de contagem tem um requisito adicional importante: a contagem deve ser feita de maneira incremental, ou seja, *online*. Cada chave recebida do fluxo de entrada deve ser imediatamente contabilizada, de modo que tenhamos, a cada passo, as contagens referentes à *parte do fluxo vista até o momento*.

O desempenho de qualquer algoritmo para o problema será medido pelo tempo consumido para contabilizar *uma* chave. Idealmente, gostaríamos que esse tempo fosse constante, ou seja, não dependesse do número de chaves já lidas (nem do número de chaves *distintas* já lidas). Mas seremos obrigados a nos contentar com algo menos que o ideal.

Quatro soluções. Discutiremos quatro diferentes algoritmos para o problema. Os dois primeiros são muito simples. Os dois seguintes, bem mais eficientes na prática, usam a técnica de hashing. Embora simples, os dois primeiros algoritmos constituem uma importante introdução aos dois outros.

Denotaremos por N o *comprimento* do fluxo de entrada, ou seja, o número total de chaves no fluxo. O valor de N pode ser muito grande (milhões ou até bilhões), mas o número de chaves distintas é tipicamente bem menor.

Suporemos que todas as chaves são menores que um número `R`. No exemplo acima, `R` vale `10000`. O conjunto **`0..R-1`** será chamado *universo das chaves*. Em geral, nem todas as chaves do universo estão presentes no fluxo de entrada.

Exercícios 1

1. Critique a seguinte proposta de algoritmo para o problema de contagem: armazene o fluxo de chaves num vetor, ordene o vetor, e depois percorra o vetor ordenado contando o número de ocorrências de cada chave.

Algoritmo 1: endereçamento direto

Começamos com um algoritmo conhecido como *endereçamento direto*. Embora muito simples, esse algoritmo contém a semente da técnica de hashing.

Suponhamos que as chaves são do tipo `int` e que `R` chaves cabem confortavelmente na memória RAM do computador. Podemos então usar uma tabela `tb[0..R-1]` para registrar os números de ocorrências.

```
int *tb;
tb = malloc (R * sizeof (int));
```

O algoritmo de endereçamento direto inicializa o vetor `tb` com zeros e repete a seguinte rotina: lê uma chave `ch` do fluxo de entrada e contabiliza a chave executando a seguinte função:

```
void contabiliza (int ch) {
   tb[ch]++;
}
```

Depois de cada execução dessa função, para cada `c` em `0..R-1`, o valor de `tb[c]` será o número de ocorrências de `c` na parte do fluxo lida até o momento.

Desempenho. Cada invocação de `contabiliza` consome tempo constante, ou seja, independente do tamanho `R` do universo e do número de chaves já lidas.

Esse algoritmo é muito rápido, mas só é prático se `R` for pequeno e conhecido explicitamente de antemão. Mesmo nesse caso, o algoritmo pode desperdiçar muito espaço. Por exemplo, se `R` vale 10 mil e o fluxo contém apenas mil chaves distintas, 90% do vetor `tb` ficará ocioso.

Exercícios 2

1. TESTES. Escreva e teste um programa que resolva o problema de contagem e imprima um relatório com as seguintes informações: o comprimento *N* do fluxo de entrada, o número de chaves distintas, a chave mais frequente, e o número de ocorrências dessa chave. Use como fluxo de entrada os arquivos `randInt1K.txt`, `randInt10K.txt`, `randInt100K.txt`, `randInt1M.txt`, que contêm mil, 10 mil, 100 mil e 1 milhão de chaves aleatórias, todas entre `0` e 9999. Use a ideia do endereçamento direto. Cronometre o seu programa (use a função `clock` da biblioteca `time`). Os resultados estão de acordo com as previsões teóricas?

Algoritmo 2: lista encadeada

Nosso segundo algoritmo armazena a contagem das chaves numa lista encadeada. As células da lista podem ter a seguinte estrutura:

```
typedef struc reg celula;
struct reg {
   int     chave, ocorr;
   celula *prox;
};
```

Se `p` é o endereço de uma célula então `p->ocorr` é o número de ocorrências da chave `p->chave`. Se `p` e `q` são endereços de duas células diferentes então `p->chave` é diferente de `q->chave`. A lista encadeada será apontada pela variável global `tb`:

```
celula *tb;
```

O algoritmo de contagem inicializa `tb` com `NULL` e repete a seguinte rotina: lê uma chave `ch` do fluxo de entrada e invoca a seguinte função para contabilizar a chave:

```
void contabiliza (int ch) {
   celula *p;
   p = tb;
   while (p != NULL && p->chave != ch)
      p = p->prox;
   if (p != NULL)
      p->ocorr += 1;
   else {
      p = malloc (sizeof (celula));
      p->chave = ch;
      p->ocorr = 1;
      p->prox = tb;
      tb = p;
   }
}
```

Desempenho. No pior caso, cada execução de `contabiliza` consome tempo proporcional ao número de chaves distintas já lidas. Portanto, a execução de `contabiliza` pode ficar cada vez mais lenta à medida que o fluxo de entrada é lido. Se todas as chaves forem distintas, as últimas execuções de `contabiliza` podem chegar a consumir tempo proporcional a N.

Mesmo no caso médio, típico de aplicações práticas, o desempenho de `contabiliza` não é bom pois consome tempo proporcional à metade do número de chaves distintas já lidas.

Por outro lado, essa solução do problema de contagem não desperdiça espaço, pois o número de células é igual ao número de chaves distintas no fluxo de entrada.

Exercícios 3

1. Lista em ordem crescente. Reescreva a função `contabiliza` acima de modo que as chaves sejam armazenadas na lista encadeada em ordem crescente. Estime o desempenho. Vale a pena mantera lista em ordem crescente?
2. Testes. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma lista encadeada para armazenar as contagens.
3. Vetor de células. Refaça a discussão da seção anterior usando um *vetor* de células no lugar de uma lista encadeada. Redimensione o vetor à medida que o número de chaves distintas aumenta. Qual o consumo de tempo dessa versão? Agora mantenha o vetor em ordem crescente de chaves e refaça a análise.

Tabelas de dispersão e funções de espalhamento

O próximos algoritmos haverão de combinar as boas qualidades dos dois algoritmos anteriores. Antes porém, precisamos introduzir o conceito de *hashing*. Começamos por estabelecer a terminologia e descrever as ideias de maneira abstrata; implementações concretas serão dadas nas seções seguintes.

Vamos supor que a contagem das chaves é registrada num vetor `tb[0..M-1]`. O valor de `M` e a natureza exata dos elementos do vetor ficarão indefinidos por enquanto. Mas você deve imaginar que, de alguma forma, cada elemento de `tb` registra o número de ocorrências de alguma chave. O vetor `tb` será chamado *tabela de dispersão* (= *hash table*). O tamanho `M` da tabela é usualmente menor que o tamanho `R` do universo de chaves. Assim, um elemento típico do vetor `tb` deverá cuidar de duas ou mais chaves.

Para determinar a posição no vetor `tb` que corresponde a uma chave `ch`, é preciso converter `ch` em um índice entre `0` e `M-1`. Qualquer função que faça a conversão, levando o universo `0..R-1` das chaves no conjunto `0..M-1` de índices, é chamada *função de espalhamento* (= *hash function*). Neste capítulo, indicaremos por

```
hash (ch, M)
```

a invocação de uma função de espalhamento para uma chave `ch`. O número `hash (ch, M)` será chamado *código hash* (= *hash code*) da chave `ch`. Uma função de espalhamento muito popular leva cada chave `ch` em `ch % M`, ou seja, no resto da divisão de ch por M. Se M vale 100, por exemplo, então ch%M consiste nos dois últimos dígitos decimais de ch.

Se a função de espalhamento levar duas chaves no mesmo índice, teremos uma *colisão*. Se M é menor que R — e mais ainda se M é menor que o número de chaves distintas — as colisões são inevitáveis. Se M vale 100, por exemplo, a função resto-da-divisão-por-M faz colidir todas as chaves que têm os mesmos dois últimos dígitos decimais.

Uma boa função de espalhamento deve espalhar bem as chaves pelo conjunto 0..M-1 de índices. Uma função que leva toda chave num índice par, por exemplo, não é boa. Uma função que só depende de alguns poucos dígitos da chave também não é boa. Infelizmente, não existe uma função de espalhamento que seja boa para todos os conjuntos de chaves extraídos do universo 0..R-1. Para começar a enfrentar essa dificuldade,

> recomenda-se que o parâmetro M seja um número primo,

pois isso tende a reduzir o número de colisões. Veja, por exemplo, o conjunto de chaves na primeira coluna da seguinte tabela (copiada do livro de Sedgewick e Wayne) e considere o resto da divisão de cada chave por 100 (um não-primo) e o resto da divisão por 97 (um primo). Observe que o número de colisões é maior no primeiro caso:

ch	ch%100	ch%97
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Em geral, encontrar uma boa função de espalhamento é mais uma arte que uma ciência…

Agora que cuidamos de espalhar as chaves pelo intervalo 0..M-1, precisamos inventar um meio de *resolver as colisões*, ou seja, de armazenar todas as chaves que a função de espalhamento leva numa mesma posição da tabela de dispersão. As seções seguintes descrevem duas maneiras de fazer isso.

Exercícios 4

1. Por que módulo primo? Suponha que ch e M são divisíveis por um inteiro k. Mostre que ch%M também será divisível por k. (Este exercício dá uma pequena indicação das vantagens de usar um número primo como valor de M.)
2. Seja d o número ⌈R/M⌉, isto é, teto de R/M. Considere a função de espalhamento que associa a cada chave ch o piso de ch/d (ou seja, o resultado da divisão inteira de ch por d). Por exemplo, se R é 10^5 e M é 10^2 então d vale 10^3 e portanto ch/d é dado pelos dois primeiros dígitos decimais de ch. Discuta a qualidade dessa função de espalhamento.
3. Paradoxo do aniversário. Aplique a função de espalhamento resto-da-divisão-por-M a uma sequência de chaves aleatórias. Depois de quantas chaves acontece a primeira colisão? Faça experimentos, com diversos valores de M, para determinar o momento da primeira colisão. (De acordo com a teoria das probabilidades clássica, a primeira colisão acontece depois de aproximadamente $\sqrt{(p\,M/2)}$ chaves, sendo p o número *pi*, igual a 3.14159…)

Algoritmo 3: hashing com encadeamento

A técnica de hashing tem dois ingredientes: uma função de espalhamento e um mecanismo de resolução de colisões. A seção anterior discutiu o primeiro ingrediente; esta seção cuida do segundo.

As colisões na tabela de dispersão podem ser resolvidas por meio de listas encadeadas: todas as chaves que têm um mesmo código hash são armazenadas numa lista encadeada. As células das listas encadeadas são iguais às do algoritmo 2:

```
typedef struc reg celula;
struct reg {
   int     chave, ocorr;
   celula *prox;
};
```

Podemos supor então que os elementos da tabela de dispersão `tb[0..M-1]` são ponteiros para listas encadeadas:

```
celula **tb;
tb = malloc (M * sizeof (celula *));
```

Para cada índice `h`, a lista encadeada `tb[h]` conterá todas as chaves que têm código hash `h`.

O algoritmo de contagem resultante é conhecido como *hashing com encadeamento* e pode ser visto como uma combinação dos algoritmos 1 e 2 discutidos acima. O algoritmo inicializa todos os elementos do vetor `tb` com `NULL` e repete a seguinte rotina: lê uma chave `ch` do fluxo de entrada e executa a seguinte função:

```
void contabiliza (int ch) {
   int h = hash (ch, M);
   celula *p = tb[h];
   while (p != NULL && p->chave != ch)
      p = p->prox;
   if (p != NULL)
      p->ocorr += 1;
   else {
      p = malloc (sizeof (celula));
      p->chave = ch;
      p->ocorr = 1;
      p->prox = tb[h];
      tb[h] = p;
   }
}
```

Desempenho. No pior caso, a função de espalhamento `hash` leva todas as chaves na mesma posição da tabela de dispersão e portanto todas as chaves ficam na mesma lista encadeada. Nesse caso, o desempenho não é melhor que o do algoritmo 2: cada execução de `contabiliza` consome tempo proporcional ao número de chaves já lidas do fluxo de entrada.

No caso médio, típico de aplicações práticas, o desempenho de `contabiliza` é muito melhor. Se a função `hash` espalhar bem as chaves pelo conjunto `0..M-1`, todas as listas encadeadas terão aproximadamente o mesmo comprimento e então podemos esperar que o consumo de tempo de `contabiliza` seja limitado por algo proporcional a

`n / M`,

onde `n` é o número de chaves lidas até o momento. Se `M` for `997`, por exemplo, podemos esperar que a função seja cerca de 1000 vezes mais rápida que aquela do algoritmo 2. É claro que devemos procurar escolher um valor de `M` que seja grande o suficiente para que as `M` listas sejam curtas (digamos algumas dezenas de elementos) mas não tão grande que muitas das listas fiquem vazias.

Exercícios 5

1. Resolva o problem da contagem para o fluxo de chaves `17 21 19 4 26 30 37` usando hashing com encadeamento. A tabela de dispersão deve ter tamanho `13` e a função de espalhamento deve ser o resto da divisão da chave por `M`. Faça uma figura do estado final da tabela de dispersão.
2. Suponha que o comprimento *N* do fluxo de entrada é aproximadamente 50000. Escolha um bom valor para o tamanho `M` da tabela de dispersão.
3. Testes. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma tabela de dispersão com colisões resolvidas por encadeamento. Experimente diferentes valores (primos e não-primos) para o tamanho `M` da tabela de dispersão. Calcule também a média e o desvio padrão dos comprimentos das listas encadeadas.

Algoritmo 4: hashing com sondagem linear

Esta seção descreve uma segunda maneira de resolver as colisões na tabela de dispersão: todas as chaves que colidem são armazenadas *na própria tabela*.

Os elementos da tabela de dispersão `tb[0..M-1]` são células que têm apenas os campos `chave` e `ocorr`:

```
typedef struc reg celula;
struct reg {
   int chave, ocorr;
};

celula *tb;
tb = malloc (M * sizeof (celula));
```

Durante a contagem, algumas das células da tabela `tb` estarão *vagas* enquanto outras estarão *ocupadas*. As células vagas terão `chave` igual a `-1`. Nas células ocupadas, a `chave` estará em `0..R-1` e `ocorr` será o correspondente número de ocorrências. Se uma célula `tb[h]` está vaga, podemos garantir que nenhuma chave na parte já lida do fluxo de entrada tem código hash igual a `h`. Mas se `tb[h]` está ocupada, não podemos concluir que o código hash de `tb[h].chave` é igual a `h`.

Cada chave `ch` do fluxo de entrada será contabilizada da seguinte maneira. Seja `h` o código hash de `ch`. Se a célula `tb[h]` estiver vaga ou tiver chave igual a `ch`, o conteúdo da célula é atualizado. Senão, algoritmo *procura a próxima célula* de `tb` que esteja vaga ou tenha chave igual a `ch`.

A implementação dessas ideias leva ao algoritmo de *hashing com sondagem linear*. O algoritmo começa por inicializar todas as células da tabela `tb` fazendo `chave = -1` e `ocorr = 0`. Depois, repete a seguinte rotina: lê uma chave `ch` e invoca a seguinte função:

```
void contabiliza (int ch) {
   int c, sonda, h;
   h = hash (ch, M);
   for (sonda = 0; sonda < M; sonda++) {
      c = tb[h].chave;
      if (c == -1 || c == ch) break; // *
      h = (h + 1) % M;
   }
   if (sonda >= M)
      exit (EXIT_FAILURE);
   if (c == -1)
      tb[h].chave = ch;
   tb[h].ocorr++;
}
```

A função faz `M` tentativas — conhecidas como *sondagens* — de encontrar uma célula "boa" (na linha * do código). A busca fracassa somente se a tabela `tb` estiver cheia. Nesse caso, a execução da função é abortada. (Teria sido melhor redimensionar a tabela `tb` e continuar trabalhando.)

Suponha, por exemplo, que o tamanho `M` da tabela de dispersão é `10` (adotamos um número não-primo para tornar o exemplo mais simples). Suponha também que `hash (ch, M)` é definido como `ch % M`. Se o fluxo de entrada é

```
333 336 1333 333 7777 446 556 999
```

então o estado final da tabela de dispersão `tb[0..9]` será o seguinte:

chave	ocorr
999	1
-1	0
-1	0
333	2
1333	1
-1	0
336	1
7777	1
446	1
556	1

Desempenho. No pior caso, a função de espalhamento `hash` leva todas as chaves na mesma posição da tabela de dispersão e portanto as chaves ocupam células consecutivas da tabela. Nesse caso, o desempenho não é melhor que o do algoritmo 2: cada execução de `contabiliza` consome tempo proporcional ao número de chaves já lidas do fluxo de entrada.

No caso médio, típico de aplicações práticas, o desempenho de `contabiliza` é muito melhor. ⚠ Se mais da metade das células da tabela de dispersão estiver vaga (como acontece se usarmos redimensionamento) e a função `hash` espalhar bem as chaves pelo conjunto `0..M-1`, uma execução da função `contabiliza` não precisará fazer mais que algumas poucas sondagens para encontrar uma célula "boa". Assim, o consumo de tempo de uma execução de `contabiliza` será praticamente independente do número de chaves já lidas.

Exercícios 6

1. Resolva o problem da contagem para o fluxo de chaves `17 21 19 4 26 30 37` usando hashing com sondagem linear. A tabela de dispersão deve ter tamanho `13` e a função de espalhamento deve ser o resto da divisão da chave por `13`. Faça uma figura do estado final da tabela de dispersão.
2. Prova de correção. Considere a função `contabiliza` do algoritmo de hashing com sondagem linear dada acima. Suponha que temos `c == -1` numa certa iteração. Prove que não existe célula `tb[h]` tal que `tb[h].chave == ch`.
3. ★ Redimensionamento da tabela de dispersão. A execução da função `contabiliza` dada acima é abortada se a tabela de dispersão estiver cheia. Escreva uma versão melhor, que redimensione a tabela escolhendo um novo valor de `M` que seja aproximadamente o dobro do anterior, alocando uma nova tabela `tb`, e reinserindo todas as chaves na nova tabela.
4. Testes. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma tabela de dispersão com colisões resolvidas por sondagem linear. Experimente diferentes valores (primos e não-primos) para o tamanho `M` da tabela. Calcule também o número máximo de sondagens que `contabiliza` faz para encontrar a posição desejada na tabela de dispersão.

Exercícios 7

1. Filtro de repetidos. Escreva um programa que leia um arquivo de texto que contém (as representações decimais de) números inteiros positivos, elimine os números repetidos, e grave uma versão do arquivo sem os repetidos. Não altere a ordem relativa dos números. O seu programa deve ter caráter de *filtro*, ou seja, deve gravar o resultado à medida que o arquivo de entrada for sendo lido.
2. Encontre o primeiro número não-repetido em um longo arquivo de texto que contém números inteiros.

Hashing de strings

Em muitas aplicações, as chaves são strings (especialmente strings ASCII) e não números. Como construir uma tabela de dispersão nesse caso? Poderíamos, por exemplo, adotar uma tabela de dispersão de tamanho `256` e usar a função de espalhamento que leva cada string no valor numérico do seu primeiro byte. (Todas as strings que começam com `'a'`, por exemplo, seriam levadas para a posição `97` da tabela.) Mas essa ideia não espalharia bem o conjunto de chaves pelo intervalo `0..255`.

Uma maneira geral de lidar com chaves que são strings envolve dois passos: o primeiro converte a string em um número inteiro; o segundo submete esse número a uma função de espalhamento. Uma conversão óbvia consiste em tratar cada string como um número inteiro na base 256. A string `"abcd"`, por exemplo, é convertida no número $97\times256^3 + 98\times256^2 + 99\times256^1 + 100\times256^0$, igual a 1633837924. Esse tipo de conversão pode facilmente produzir números maiores que `INT_MAX`, e assim levar a um overflow aritmético. Se os cálculos forem feitos com variáveis `int`, o resultado poderá ser estritamente negativo, o que seria desastroso. Por isso, faremos os cálculos com variáveis `unsigned`, garantindo assim que o resultado fique entre `0` e `UINT_MAX` mesmo que haja overflow:

```
typedef char *string;

unsigned convert (string s) {
   unsigned h = 0;
   for (int i = 0; s[i] != '\0'; i++)
      h = h * 256 + s[i];
```

```
    return h;
}
```

Para submeter o resultado da conversão à função de espalhamento basta fazer `hash (convert (s), M)`. Se adotarmos a função resto-da-divisão-por-`M`, basta fazer `convert (s) % M`.

Uma boa alternativa é fazer a conversão e o cálculo da função de espalhamento em um só movimento. O resultado pode não ser idêntico a `convert (s) % M`, mas cumpre o papel de espalhar as chaves pelo conjunto `0..M-1`:

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

Se a string é `"abcd"` e `M` é `101`, por exemplo, o índice calculado por `string_hash` é `11`:

```
( 0 * 256 +  97) % 101 = 97
(97 * 256 +  98) % 101 = 84
(84 * 256 +  99) % 101 = 90
(90 * 256 + 100) % 101 = 11
```

O uso da base `256` não é obrigatório; podemos usar uma outra base qualquer. Por razões não muito óbvias, convém que a base seja um número primo, como `31` por exemplo.

Exercícios 8

1. Suponha que as chaves são strings e `M` vale `256`. Considere a função de espalhamento que associa a cada chave o seu primeiro byte. Discuta a qualidade dessa função de espalhamento.
2. Mostre que se trocarmos "`256`" por "`1`" na função `convert`, todas as permutações da string `s` colidirão.
3. Na função `convert`, por que não trocar as duas linhas do meio pelas seguintes?

```
unsigned h = s[0];
for (i = 1; s[i] != '\0'; i++)
```

Veja minhas notas de aulas sobre hashing baseadas no livro de Sedgewick e Wayne e minhas notas baseadas no livro de Sedgewick.

Veja os verbetes *Hash table* e *Hash function* na Wikipedia.

Veja o artigo *Hash function* no MathWorld da Wolfram.

Atualizado em 2018-08-14
https://www.ime.usp.br/~pf/algoritmos/
Paulo Feofiloff
DCC-IME-USP