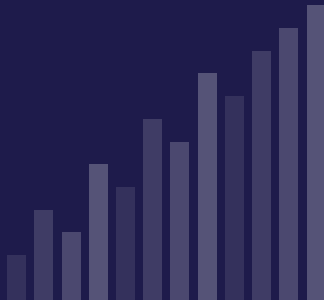

AULA 20 · BLOCO 3: TABELAS HASH E ÁRVORES

Funções hash e hashing universal

Seção 11.3

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Construir funções hash pelos métodos da divisão e da multiplicação
- ▶ Reconhecer escolhas ruins de m (potências de 2 com chaves estruturadas)
- ▶ Definir hashing universal e o que ele garante
- ▶ Medir o efeito da função hash em chaves com padrão



Teoria

O que é uma boa função hash; divisão, multiplicação, família universal



Prática em C

Quantas posições cada função usa com chaves múltiplas de 64



Experimento

Quatro funções, chaves com padrão, fator de carga 1



Conexões

Segurança, sistemas distribuídos, big data e criptografia

O que é uma boa função hash

- ▶ Ideal: **hash uniforme simples**, cada chave igualmente provável em cada posição, independente das demais.
- ▶ Na prática não conhecemos a distribuição das chaves, e elas têm **padrões**: endereços alinhados, IDs sequenciais, nomes com mesmo prefixo.
- ▶ Interpretar chaves como números naturais: uma string vira $\sum c_i b^i$ (base b).
- ▶ Uma boa função usa **todos os bits** da chave e espalha padrões regulares.

Método da divisão

$$h(k) = k \bmod m$$

- ▶ Rápido: uma divisão.
- ▶ **Evite** $m = 2^p$: $h(k)$ seria só os p bits menos significativos. Chaves múltiplas de 64 cairiam em apenas $m/64$ posições.
- ▶ Escolha m **primo** longe de potências de 2. Exemplo: $n = 2000$ strings, até 3 por lista: $m = 701$.
- ▶ Chaves em progressão aritmética $s, 2s, 3s, \dots$ com m primo (e s não múltiplo de m) ocupam **todas** as posições.

Método da multiplicação

$$h(k) = \lfloor m (kA \bmod 1) \rfloor, \quad 0 < A < 1$$

- ▶ O valor de m não é crítico: pode ser 2^p .
- ▶ Implementação com palavras de w bits: $s = \lfloor A \cdot 2^w \rfloor$; $h(k)$ = os p bits mais altos de $(k s) \bmod 2^w$. Uma multiplicação e um deslocamento.
- ▶ Knuth sugere $A \approx (\sqrt{5} - 1)/2 = 0,618\dots$ (razão áurea): *hashing de Fibonacci*.

Em C, com unsigned de 32 bits: `(k * 2654435769u) >> (32 - p)`.

Hashing universal

Qualquer função fixa tem um conjunto de n chaves que cai todo na mesma posição. Um adversário pode escolhê-lo.

Solução: sortear a função, no início da execução, de uma família $\mathcal{H}$.

- ▶ $\mathcal{H}$ é **universal** se, para $k \neq l$, no máximo $|\mathcal{H}|/m$ funções fazem $h(k) = h(l)$: probabilidade de colisão $\leq 1/m$.
- ▶ **Teorema 11.3:** com encadeamento, a lista esperada de uma chave tem tamanho $\leq 1 + \alpha$, para **qualquer** conjunto de chaves.
- ▶ Família do livro: $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$, p primo maior que as chaves, $a \in [1, p - 1]$, $b \in [0, p - 1]$.



Prática em C

- ▶ Chaves $s, 2s, \dots, ms$ (padrão comum)
- ▶ Divisão por m , divisão por um primo, multiplicação
- ▶ Conta quantas posições cada função ocupa

Comparando funções hash (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Lê o passo s e m; insere as chaves s, 2s, 3s, ..., m*s (um padrão comum: endereços
5     alinhados, IDs múltiplos de 10) e conta quantas posições cada função usa. */
6  unsigned divisao(unsigned k, unsigned m) { return k % m; }
7
8  unsigned multiplicacao(unsigned k, int p) {           // m = 2^p; A = √(5 - 1)/2 em 32 bits
9     unsigned s = 2654435769u;                          // [A · ]2^32, sugestão de Knuth
10    return (k * s) >> (32 - p);                        // os p bits mais altos de k·s mod 2^32
11 }
12
13 typedef unsigned (*Hash)(unsigned, unsigned);
```

Comparando funções hash (2/3)

```
15  √[]
16  int usadas(Hash f, unsigned passo, unsigned m, unsigned param) {
17      char *ocupada = calloc(m, 1);
18      int cont = 0;
19      for (unsigned i = 1; i <= m; i++) {
20          unsigned pos = f(i * passo, param);
21          if (!ocupada[pos]) { ocupada[pos] = 1; cont++; }
22      }
23      free(ocupada);
24      return cont;
25  }
26
27  unsigned mult_aux(unsigned k, unsigned p) { return multiplicacao(k, (int)p); }
28
29  int main(void) {
30      unsigned passo, m;
```

Comparando funções hash (3/3)

```
31  √[]
32      unsigned primo = m - 1;                // maior primo menor que m
33      for (;;) primo--;
34          int ok = 1;
35          for (unsigned d = 2; d * d <= primo; d++) if (primo % d == 0) { ok = 0; break; }
36          if (ok) break;
37      }
38      int p = 0;
39      while ((1u << p) < m) p++;
40      printf("divisão por %u: %d posições usadas\n", m, usadas(divisao, passo, m, m));
41      printf("divisão por %u: %d posições usadas\n", primo,
42             usadas(divisao, passo, primo, primo));
43      printf("multiplicação (m = 2^%d): %d posições usadas\n", p,
44             usadas(mult_aux, passo, 1u << p, (unsigned)p));
45      return 0;
46  }
```



Experimento

- ▶ Chaves $64, 128, \dots, 64m$ e $n = m$ (fator de carga 1)
- ▶ Divisão por 2^p , divisão por primo, multiplicação, universal
- ▶ Universal: média sobre muitas funções sorteadas

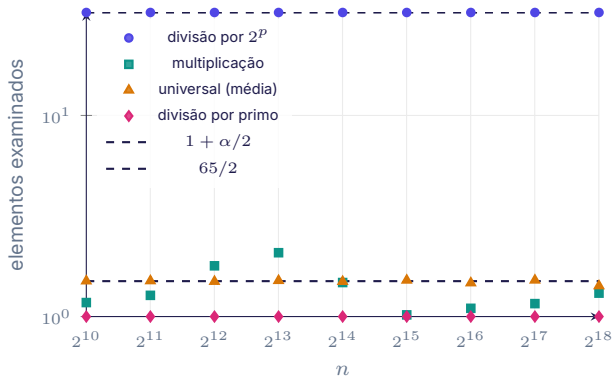
Chaves com padrão

Hipótese. Com $m = 2^p$, o método da divisão só vê os bits baixos: múltiplos de 64 caem em $m/64$ posições, listas de 64, custo médio $(64 + 1)/2 = 32,5$. Com m primo, a progressão ocupa todas as posições: custo 1. O hashing universal garante em média no máximo $1 + \alpha/2 = 1,5$, para **qualquer** conjunto de chaves.

Método

1. $m = 2^{10}, \dots, 2^{18}$ e $n = m$ chaves 64i
2. Custo médio de busca com sucesso:
 $\sum L(L + 1)/2 / n$
3. Primo: o menor primo maior que m
4. Universal: média de $2^{26}/n$ funções h_{ab} sorteadas

Custo médio de busca com sucesso



n	medido	razão
1.024	1,502	1,002
2.048	1,505	1,003
4.096	1,494	0,996
8.192	1,509	1,006
16.384	1,493	0,995
262.144	1,421	0,947

razão = medido $\div$ $1 + \alpha/2$

✓ desvio máx. 5,3%

Divisão por 2^p : 32,5 em todos os tamanhos. Primo: 1 (perfeito). Universal: 1,5 em média. Multiplicação: entre 1 e 2, conforme m .

EXPERIMENTO

O que o experimento mostrou

A escolha da função muda o custo em mais de **30 vezes** com as mesmas chaves: dividir por potência de 2 dá exatamente 32,5; dividir por um primo dá 1. O hashing universal fica em 1,5 em média, como garante o teorema, **sem depender** das chaves. Uma função isolada sorteada, porém, pode ter azar: a garantia é sobre a média da família.

➤ Reproduza: `gcc -O2 aula20_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Segurança
- ▶ Sistemas distribuídos
- ▶ Big data
- ▶ Criptografia

Onde isso aparece na computação

 Segurança

Em 2011, ataques de *hash flooding* derrubaram servidores web enviando parâmetros que colidiam; Python, Ruby e Java passaram a sortear a função (SipHash).

 Sistemas distribuídos

Hashing consistente (Dynamo, Cassandra) e *rendezvous hashing* decidem qual servidor guarda cada chave; uma função ruim desequilibra o cluster.

 Big data

Contagem aproximada (HyperLogLog), filtros de Bloom e *count-min sketch* dependem de funções hash independentes e bem espalhadas.

 Criptografia

Hashes criptográficos (SHA-256) vão além: resistência a colisões e a pré-imagem. Senhas, assinaturas e blockchain dependem disso.

Para casa

Exercícios

1. (11.3-1) Busca numa lista de strings longas com hash armazenado: como usar o hash para acelerar?
2. (11.3-3) Mostre que, com $m = 2^p - 1$ e strings em base 2^p , permutar caracteres não muda o hash.
3. (11.3-4) Calcule as posições de 61, 62, 63, 64, 65 com $m = 1000$ e $A = (\sqrt{5} - 1)/2$.
4. Rode `aula20_funcoes_hash.c` com passos 1, 10, 64 e 1000. Qual função é robusta?
5. Meça o custo de **uma** função universal sorteada várias vezes: qual a pior que você encontra?

Leitura

Cormen et al., seção 11.3.

Código

josedihego.net/estruturas