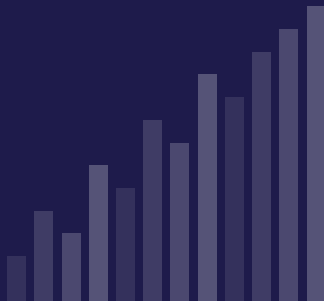

AULA 19 · BLOCO 3: TABELAS HASH E ÁRVORES

Tabelas hash com encadeamento

Seções 11.1 e 11.2

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Explicar endereçamento direto e por que ele não escala
- ▶ Resolver colisões por encadeamento
- ▶ Analisar a busca em função do fator de carga $\alpha = n/m$
- ▶ Medir α e $1 + \alpha/2$ elementos examinados por busca



Teoria

Dicionários, endereçamento direto, hash, colisões e o teorema 11.1/11.2



Prática em C

Frequência de palavras com uma tabela hash em C



Experimento

Custo de busca com e sem sucesso para α de 0,25 a 4



Conexões

Bancos de dados, compiladores, redes e cache distribuído

Dicionários e endereçamento direto

- ▶ **Dicionário:** INSERT, SEARCH, DELETE.
- ▶ Se as chaves vêm de um universo pequeno $U = \{0, \dots, m - 1\}$, use um vetor: $T[k]$ guarda a chave k .
- ▶ Tudo em $O(1)$ no pior caso.
- ▶ Problema: o universo costuma ser enorme (CPFs, URLs, palavras) e o conjunto K de chaves usadas, pequeno. Memória $|U|$ é inviável.

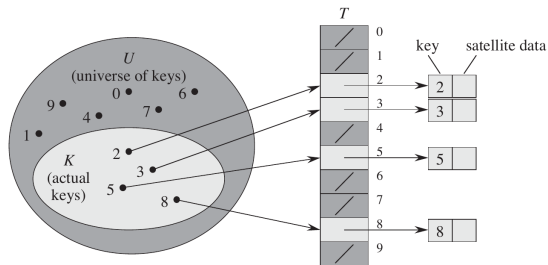


Figura 11.1 de Cormen et al. (2009) — tabela de endereçamento direto: a chave é o próprio índice.

Funções hash e colisões

- ▶ Uma **função hash**
 $h : U \rightarrow \{0, \dots, m-1\}$ diz onde guardar cada chave.
- ▶ Memória $\Theta(|K|)$ em vez de $\Theta(|U|)$.
- ▶ Como $|U| > m$, **colisões** são inevitáveis (princípio da casa dos pombos).
- ▶ Duas soluções: encadeamento (hoje) e endereçamento aberto (aula 21).

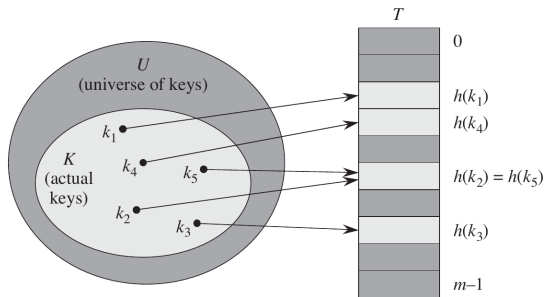


Figura 11.2 de Cormen et al. (2009) — h leva o universo U para m posições; k_2 e k_5 colidem.

Encadeamento

- ▶ $\text{INSERT}(T, x)$: insere na cabeça da lista $T[h(x.chave)]$: $O(1)$.
- ▶ $\text{SEARCH}(T, k)$: percorre a lista $T[h(k)]$.
- ▶ $\text{DELETE}(T, x)$: $O(1)$ com listas duplamente ligadas.
- ▶ Pior caso: todas as chaves na mesma lista, $\Theta(n)$.

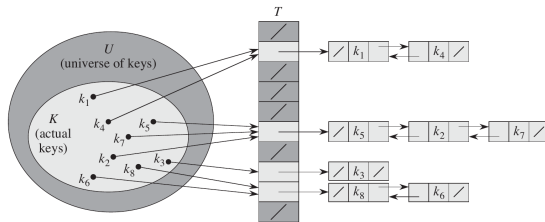


Figura 11.3 de Cormen et al. (2009) — cada posição guarda uma lista com as chaves que caíram nela.

Análise: o fator de carga

Hash uniforme simples: cada chave cai em qualquer das m posições com a mesma probabilidade, independentemente das outras. Seja $\alpha = n/m$ o **fator de carga**.

- ▶ **Teorema 11.1** (sem sucesso): a busca examina em média α elementos; tempo $\Theta(1 + \alpha)$.
- ▶ **Teorema 11.2** (com sucesso): examina em média $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$ elementos.
- ▶ Se m cresce junto com n ($n = O(m)$), então $\alpha = O(1)$ e **todas as operações são $O(1)$ em média**.

Na prática: dobre m (e reinsira tudo) quando α passar de um limite, como no vetor dinâmico da aula 13: custo amortizado $O(1)$.



Prática em C

- ▶ Hash de strings: polinômio em base 31, resto por $m = 101$ (primo)
- ▶ Listas na cabeça; busca antes de inserir para contar repetições
- ▶ Ao fim: palavras, posições usadas e a maior lista

Frequência de palavras com hash (1/4)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define M 101                                // número de posições (primo)
6
7  typedef struct No { char *palavra; int cont; struct No *prox; } No;
8  No *T[M];                                    // T[i] é a lista da posição i
9
10 /* Polinômio em base 31 com resto por M: método da divisão aplicado à string. */
11 unsigned h(const char *s) {
12     unsigned long v = 0;
13     for (; *s; s++) v = (v * 31 + (unsigned char)*s) % M;
14     return v;
15 }
```

Frequência de palavras com hash (2/4)

```
16
17 No *busca(const char *s) {                               // CHAINED-HASH-SEARCH: percorre T[h(s)]
18     for (No *x = T[h(s)]; x; x = x->prox)
19         if (strcmp(x->palavra, s) == 0) return x;
20     return NULL;
21 }
```

Frequência de palavras com hash (3/4)

```
23 void insere(const char *s) { // CHAINED-HASH-INSERT na cabeça: O(1)
24     No *x = busca(s);
25     if (x) { x->cont++; return; } // já existe: só conta
26     x = malloc(sizeof(No));
27     x->palavra = malloc(strlen(s) + 1);
28     strcpy(x->palavra, s);
29     x->cont = 1;
30     x->prox = T[h(s)];
31     T[h(s)] = x;
32 }
33
34 int main(void) {
35     char p[100];
36     int n = 0;
37     while (scanf("%99s", p) == 1) { insere(p); n++; }
```

Frequência de palavras com hash (4/4)

```
38     int usadas = 0, maior = 0;
39     for (int i = 0; i < M; i++) {
40         int tam = 0;
41         for (No *x = T[i]; x; x = x->prox) {
42             printf("%s: %d\n", x->palavra, x->cont);
43             tam++;
44         }
45         if (tam) usadas++;
46         if (tam > maior) maior = tam;
47     }
48     printf("%d palavras, %d posições usadas de %d, maior lista: %d\n",
49           n, usadas, M, maior);
50     for (int i = 0; i < M; i++)
51         while (T[i]) { No *x = T[i]; T[i] = x->prox; free(x->palavra); free(x); }
52     return 0;
53 }
```



Experimento

- ▶ Hipótese: α (sem sucesso) e $1 + \alpha/2$ (com sucesso)
- ▶ $m = 65.536$, α de 0,25 a 4
- ▶ Chaves aleatórias de 32 bits (hash uniforme)

O fator de carga decide

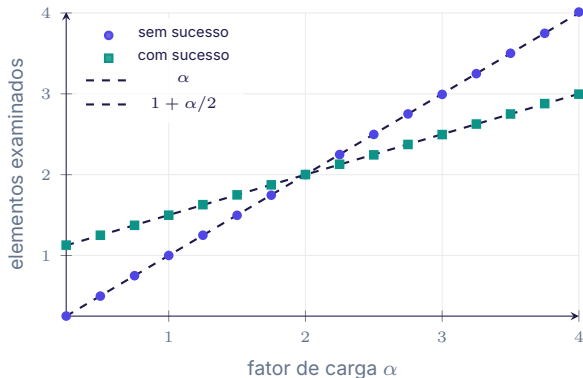
Hipótese. Com hash uniforme simples, uma busca **sem sucesso** percorre uma lista inteira: em média α elementos. Uma busca **com sucesso** pela i -ésima chave inserida examina ela mesma mais as inseridas depois na mesma lista: em média $1 + \alpha/2$.

Método

1. $m = 2^{16}$ posições, $n = \alpha m$ chaves aleatórias
2. $\alpha = 0,25, 0,5, \dots, 4$
3. Com sucesso: busca de todas as n chaves
4. Sem sucesso: 200.000 chaves sorteadas

🧪 EXPERIMENTO

Elementos examinados por busca



alfa	medido	razão
0,250	0,251	1,004
1,000	1,000	1,000
1,750	1,746	0,998
2,500	2,498	0,999
3,250	3,249	1,000
4,000	4,011	1,003

razão = medido $\div$ α

✓ desvio máx. 0,3%

As duas retas se cruzam em $\alpha = 2$:
acima disso, procurar algo que não
existe custa mais que achar.

EXPERIMENTO

O que o experimento mostrou

Os teoremas 11.1 e 11.2 se confirmam com desvio menor que 1%: α elementos sem sucesso, $1 + \alpha/2$ com sucesso. O custo depende de α , **não de** n : com α fixo, uma tabela com um bilhão de chaves busca tão rápido quanto uma com mil.

>_ Reproduza: `gcc -O2 aula19_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Bancos de dados
- ▶ Compiladores e linguagens
- ▶ Cache distribuído
- ▶ Redes

Onde isso aparece na computação

Bancos de dados

Índices hash respondem `WHERE id = ?` em $O(1)$; o *hash join* monta uma tabela hash com a menor relação e varre a maior.

Compiladores e linguagens

A tabela de símbolos guarda identificadores numa tabela hash; dicionários do Python, objetos do JavaScript e HashMap do Java também.

Cache distribuído

Memcached e Redis distribuem chaves entre servidores com hash; o *hashing consistente* minimiza as chaves movidas quando um servidor entra ou sai.

Redes

Switches mantêm a tabela de endereços MAC em hash; roteadores e firewalls classificam fluxos (5-tupla) por hash para decidir em nanossegundos.

Para casa

Exercícios

1. (11.1-1) Como achar o máximo numa tabela de endereçamento direto? Qual o custo?
2. (11.2-2) Insira 5, 28, 19, 15, 20, 33, 12, 17, 10 com $h(k) = k \bmod 9$ e encadeamento.
3. (11.2-3) E se as listas forem mantidas ordenadas? O que muda em cada operação?
4. Rode `aula19_hash_encadeamento.c` com um livro inteiro (Domínio Público) e analise a maior lista.
5. Implemente o redimensionamento: dobre m quando $\alpha > 1$ e meça o custo amortizado.

Leitura

Cormen et al., seções 11.1 e 11.2.

Código

josedihego.net/estruturas