DCA0204, Módulo 5
Hashing

Daniel Aloise

baseado em slides do prof. Leo Liberti, École Polytechnique, França

DCA, UFRN

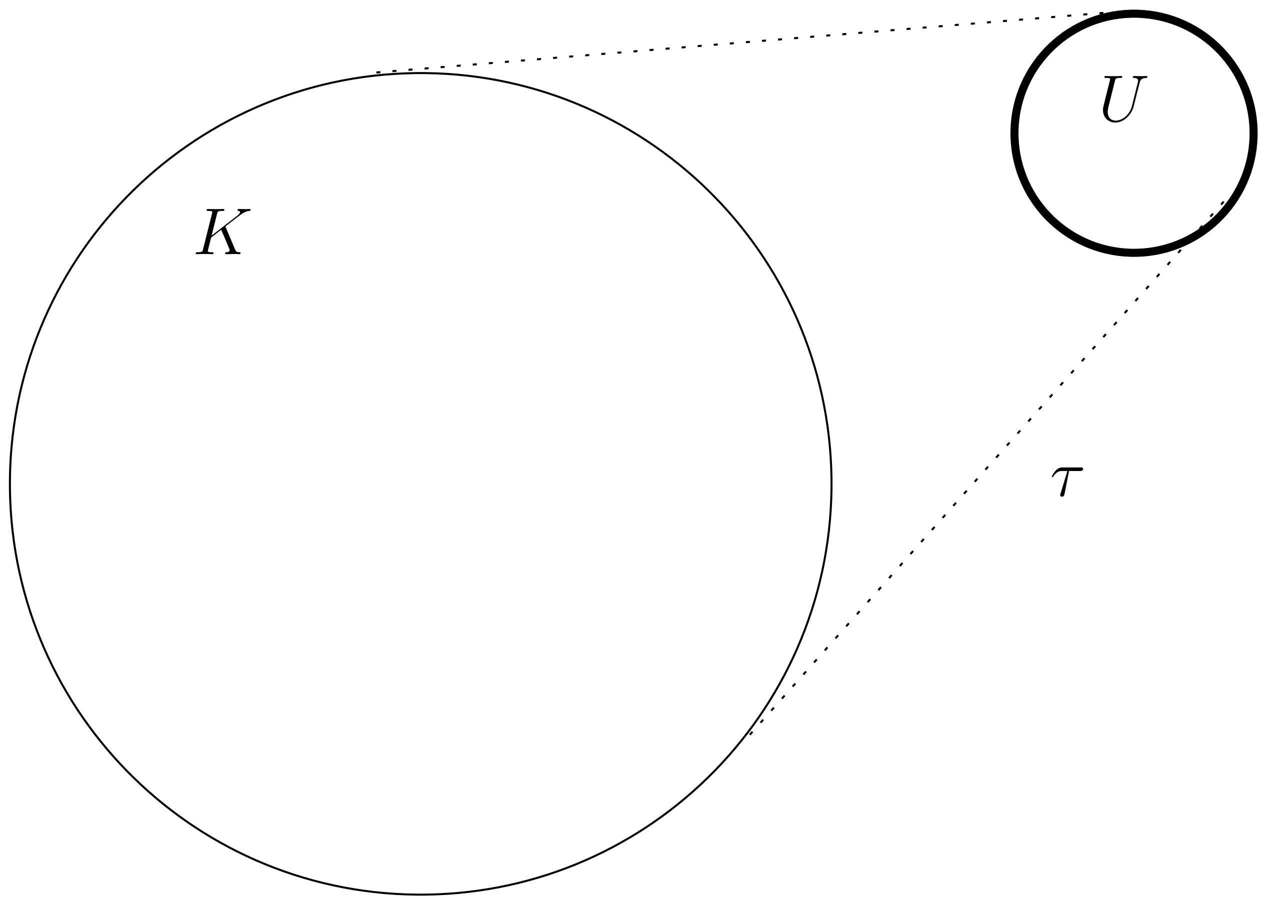
Sumário

- Busca
- Tabelas
- Hashing
- Colisões
- Implementação

Motivação

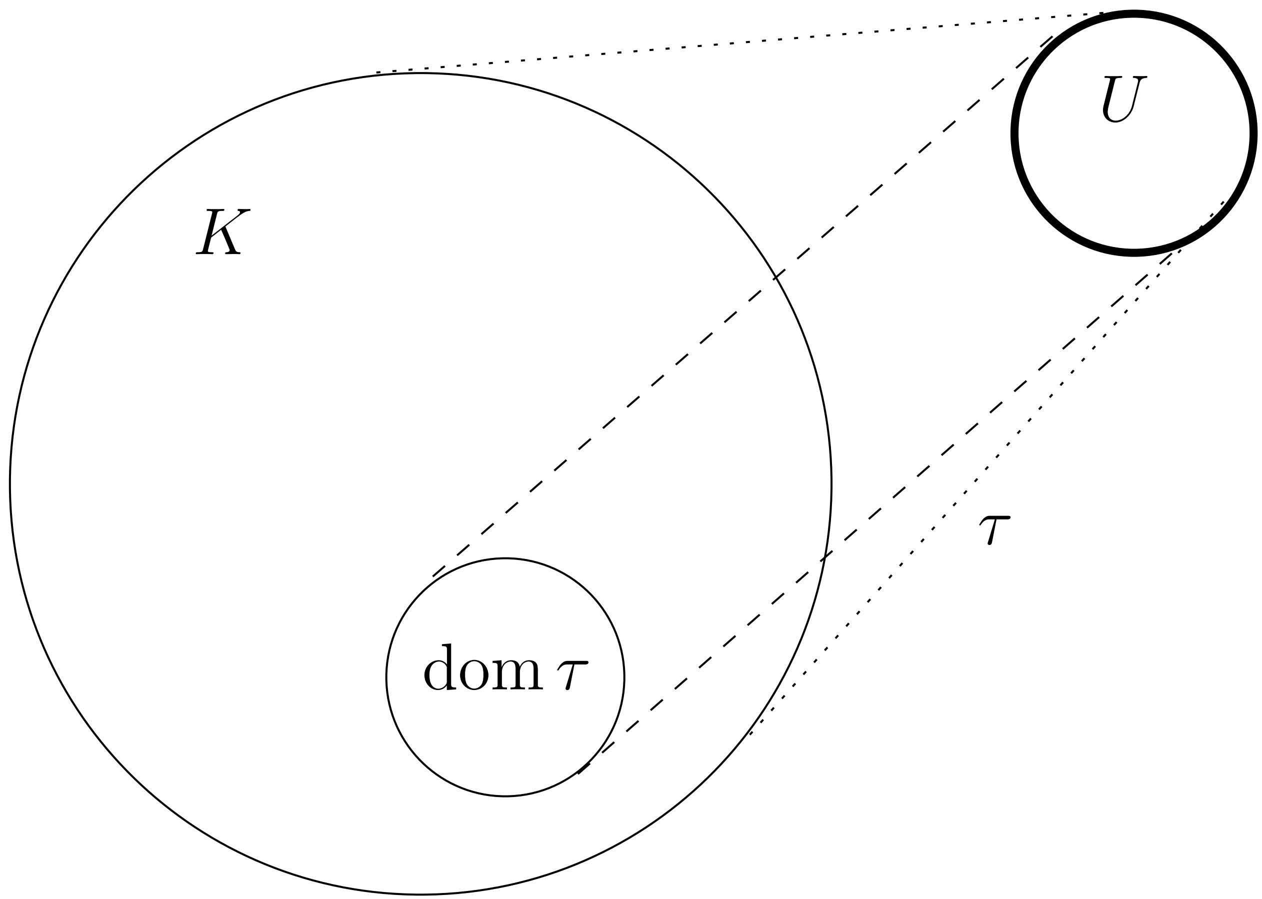
- Uma agenda “física” de contatos
- Cada página corresponde a um caracter
- A página com o caracter k contem todos os nomes começando com k
- Fácil de buscar:
  - 26 caracteres no alfabeto: $O(1)$
  - L linhas por página, L não depende de n: $O(1)$
- A busca na agenda de contatos é $O(1)$.

Conhecimento básico



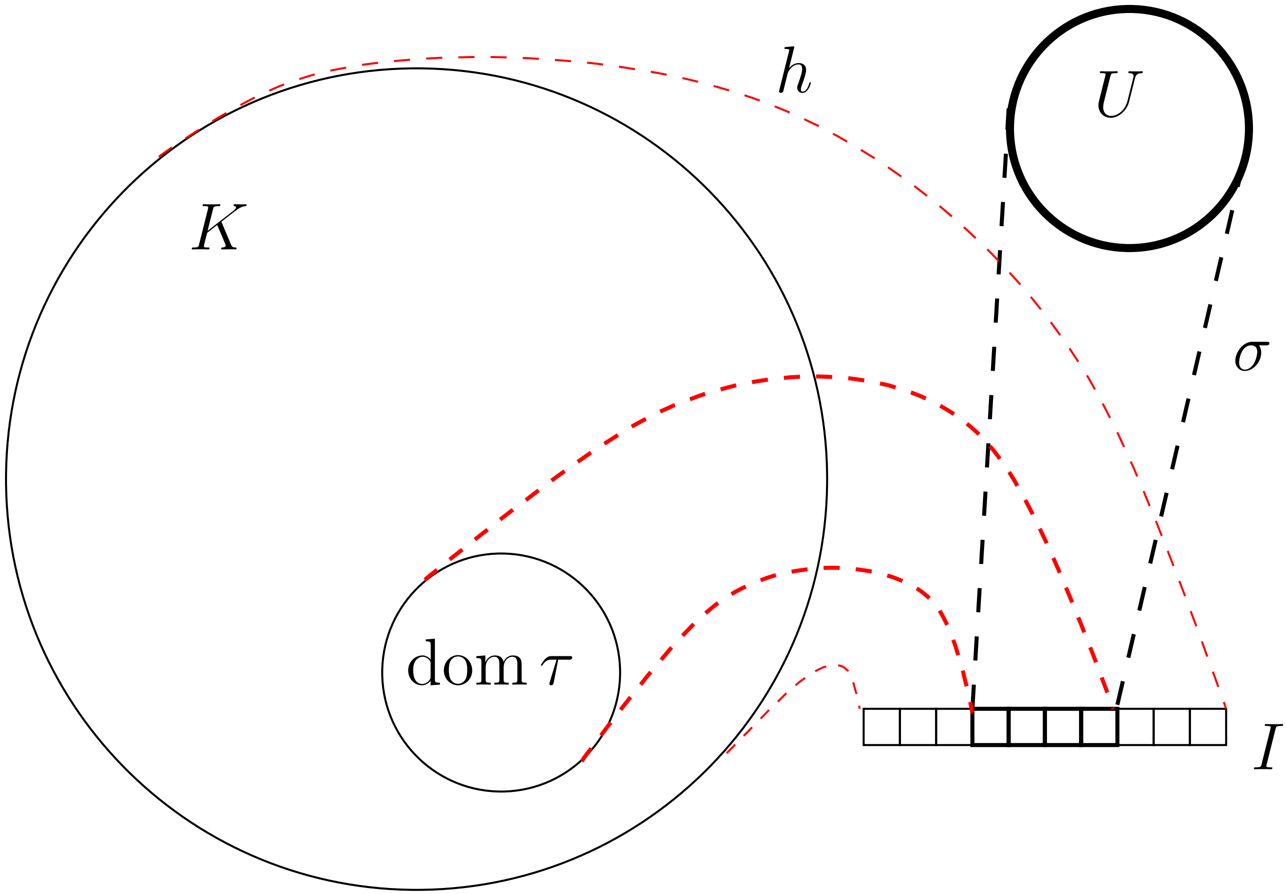
- K um conjunto muito grande de chaves; U: um conjunto de objetos; $\tau : K \to U$: uma tabela
- Assuma K muito grande para ser armazenado, mas $\operatorname{dom} \tau$ é pequeno
- Encontre uma função $h : K \to I$ com $I = \{0, 1, \ldots, p-1\}$ e $|I| \approx |U|$, então armazene $u = \tau(k)$ em $\sigma(i)$ onde $i = h(k)$

Conhecimento básico



- K um conjunto muito grande de chaves; U: um conjunto de objetos; $\tau : K \to U$: uma tabela
- Assuma K muito grande para ser armazenado, mas $\operatorname{dom}\tau$ é pequeno
- Encontre uma função $h : K \to I$ com $I = \{0, 1, \ldots, p-1\}$ e $|I| \approx |U|$, então armazene $u = \tau(k)$ em $\sigma(i)$ onde $i = h(k)$

Conhecimento básico



- K um conjunto muito grande de chaves; U: um conjunto de objetos; $\tau : K \to U$: uma tabela
- Assuma K muito grande para ser armazenado, mas $\mathrm{dom}\,\tau$ é pequeno
- Encontre uma função $h : K \to I$ com $I = \{0, 1, \ldots, p-1\}$ e $|I| \approx |U|$, então armazene $u = \tau(k)$ no elemento $\sigma(i)$ do array onde $i = h(k)$

Por que não uma lista?

- Considere uma lista de pares (chave, dado)
- Procurar é $O(n)$
- Não é eficiente em termos de tempo

Por que não um array?

- Considere um array de dados indexados por chaves
- Procurar é $O(1)$
- Suponha que as chaves são $\{1, 16, 1643, 1094382\}$
- Precisa alocar espaço para 1094382 dados, mas só 4 são necessários
- Não é eficiente em termos de espaço

Definição do problema

- $K =$*chaves*, $U =$*dados*
- Considere um array de dados indexados por chaves
- Associe algumas chaves com dados
- Obtenha uma *tabela* injetiva $\tau : K \to U$, com $\operatorname{dom} \tau \subsetneq K$
- **Problema**

  Dada uma chave $k \in K$, determine se $k \in$ dom τ

Definição do problema

- Considere um **conjunto de índices** I tal que $|I| \approx |\operatorname{dom} \tau| \ll |K|$
- **Tabela Hash:** função $\sigma : I \to U$
- **Função Hash** $h : K \to I$ tal $\tau = \sigma \circ h$

$$\begin{tikzcd} K \arrow[rr, "\tau"] \arrow[dr, "h"'] & & U \\ & I \arrow[ur, "\sigma"'] & \end{tikzcd}$$

- $\Rightarrow u$ é armazenado na posição $h(k)$
- Temos então $\sigma(h(k)) = \tau(k) = u$

Colisões

- Pelo esquema acima, $k \in \mathrm{dom}\,\tau \Leftrightarrow h(k) \in I$
- Este esquema só funciona se h for uma função injetiva
- Caso contrário temos *colisões* (pense na nossa agenda de contatos)
- Se temos colisões, $\sigma(h(k)) =$ todos os dados u com o mesmo valor de $h(k)$

Última dúvida incômoda

- Precisamos armazenar $h : K \rightarrow I$ em algum lugar
- Lista não é eficiente em termos de tempo
- Array não é eficiente em termos de espaço

Estamos apenas adiando o problema?

A mágica

Não é preciso armazenar h explicitamente

- Defina $h(k)$ usando uma “pequena descrição”
- Uma fórmula aplicada à descrição da chave k
- Ex. agenda:
  - Seja $k =$ Leo
  - código ASCII: L = 76, e=101, o = 111
  - $h(k) = 76 + 101 + 111 = 288$

    colisões, $h(HHHH) = 288$ também
  - $h(k) = 76 \times 113^2 + 101 \times 113 + 111 = 981968$

    colisão desaparece

Colisões são prováveis

- Um fato triste da vida: a maioria das funções hash *não* são injetivas

Colisões são prováveis

- Um fato triste da vida: a maioria das funções hash *não* são injetivas
- Existem $|I|^{|K|}$ funções de $K \to I$, todas potenciais funções hash

Colisões são prováveis

- Um fato triste da vida: a maioria das funções hash *não* são injetivas
- Existem $|I|^{|K|}$ funções de $K \to I$, todas potenciais funções hash
- If $|I| < |K|$, nenhuma é injetiva

Colisões são prováveis

- Um fato triste da vida: a maioria das funções hash *não* são injetivas
- Existem $|I|^{|K|}$ funções de $K \to I$, todas potenciais funções hash
- If $|I| < |K|$, nenhuma é injetiva
- If $|I| \geq |K|$:
  - Existem $|I|$ maneiras de se escolher a imagem do primeiro elemento de K,
  - $|I| - 1$ maneiras pro segundo, e assim por diante
  - obtem-se $P(|I|, |K|)$ funções injetivas $K \to I$

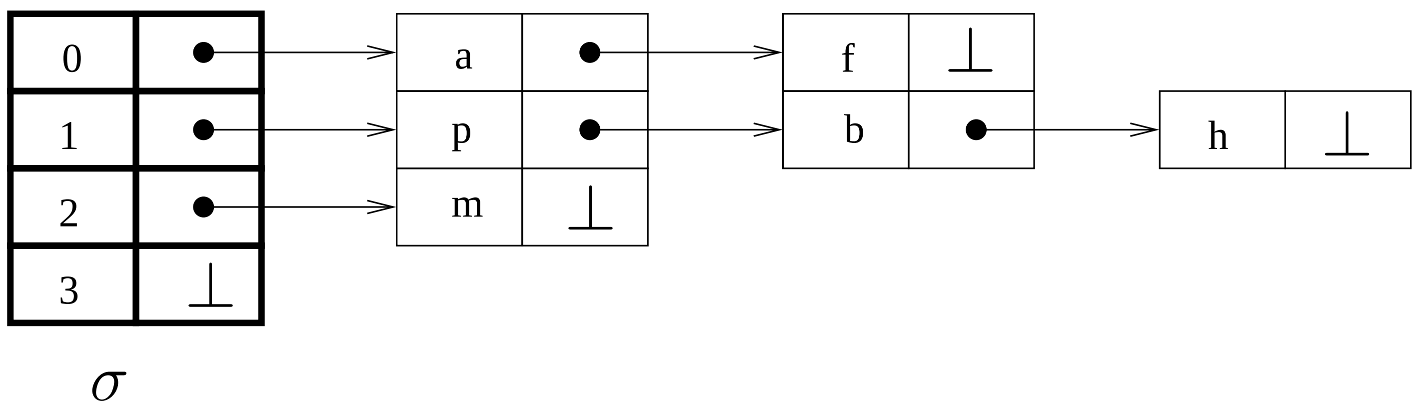
Colisões são prováveis

- Um fato triste da vida: a maioria das funções hash *não* são injetivas
- Existem $|I|^{|K|}$ funções de $K \to I$, todas potenciais funções hash
- If $|I| < |K|$, nenhuma é injetiva
- If $|I| \geq |K|$:
  - Existem $|I|$ maneiras de se escolher a imagem do primeiro elemento de K,
  - $|I| - 1$ maneiras pro segundo, e assim por diante
  - obtem-se $P(|I|, |K|)$ funções injetivas $K \to I$
- Se $|K| = 31$ e $|I| = 41$, existem +- 10^{50} funções, apenas 10^{43} delas são injetivas (*uma em dez milhões: raro*)

Obrigado a D. Knuth por este cálculo

Resolvendo colisões: encadeamento

- Lida com colisões
- armazena todos os dados com o mesmo código hash em uma lista
- Armazena lista em $h(k)$



$h(\mathtt{a}) = h(\mathtt{f}) = 0$

$h(\mathtt{p}) = h(\mathtt{b}) = h(\mathtt{h}) = 1$

$h(\mathtt{m}) = 2$

$\perp$ se refere a referência `null`

Implementação: `find`

- Método `find` de uma tabela hash com colisões

  `find`(k) $\{$

  if $|\sigma(h(k))| = 0$ **then**
    return NOT_FOUND
  else
    if $|\sigma(h(k))| = 1$ **then**
      return $\sigma(h(k)$
    else
      return $\sigma(h(k))$.`find`(k);
    end if
  end if

  $\}$

- Reduza colisões

  Use funções hash injetivas ou "quase injetivas"

Boas funções hash

- Todos os dados computacionais podem ser escritos como sequências de bytes de vários tamanhos
- Cada byte guarda um inteiro entre 0 e 255.
- Assumimos que todas as sequências em $k = (k_1, \ldots, k_\ell) \in K$ têm o mesmo comprimento ℓ (c.c., insira sequências de zeros à esquerda)
- p: menor primo $\geq |U|$
- A seguinte família de funções hash é quase injetiva:

$$h_a(k) = \sum_{j \leq \ell} a_j k_j (\mathrm{mod} p)$$

- A escolha de a faz diferença

Complexidade de pior caso da função `find`(k)

- Assuma:
  - o tamanho da chave k é $O(1)$ wrt a $|\operatorname{dom}\tau|$
  - a função hash é avaliada em tempo $O(1)$
- Pior caso:
  - $\exists i \in I, \forall k \in K \quad h(k) = i$
  - todas os dados são armazenados na mesma posição $\sigma_i \Rightarrow O(n)$

Complexidade de caso médio da função `find`(k)

- Assuma:
  - a probabilidade que $h(k) = h(k')$ para $k \neq k'$: $\frac{1}{|I|}$
  - esta probabilidade é independente de k e k'
- L_k: variável randômica para $|\sigma(h(k))|$
- Percorrer $\sigma(h(k))$: $O(E(L_k))$
- $X_{k\ell}$: variável randômica que vale 1, se $h(k) = h(\ell)$, 0 c.c.

$$
\begin{aligned}
L_k &= \sum_{\ell \in \operatorname{dom} \tau} X_{k\ell} \\
E(L_k) &= E[\sum_{\ell \in \operatorname{dom} \tau} X_{k\ell}] = \sum_{\ell \in \operatorname{dom} \tau} E[X_{k\ell}] \\
&= \sum_{\ell \in \operatorname{dom} \tau} \frac{1}{|I|} = \frac{|\operatorname{dom} \tau|}{|I|} = \alpha
\end{aligned}
$$

`find`(k) leva tempo $O(1 + \alpha)$

Aplicação: Comparando objetos C++/Java

- Um objeto pode ocupar um pedaço grande da memória (ex. uma tabela de banco de dados)
- Algumas vezes queremos testar quando dois objetos a, b são iguais
- Requer comparação bit-a-bit: $O(\min(|\mathsf{a}|, |\mathsf{b}|))$: ineficiente
- Ao invés, teste `a.hashCode() == b.hashCode()`, tempo $O(1)$
- A função `hashCode()` do Java é boa
- Pouca probabilidade de colisão (mas ainda sim existe!)
- Você pode também fazer uma implementação ad-hoc da sua própria função hash

Objetos diferentes podem ter códigos hash iguais

Se os códigos hash são diferentes, os objetos são diferentes

Linear Probing

- Implementado com array: pode ser circular!
- A implementação da função `insert`(k) é trivial

  Se $\sigma(h(k))$ estiver ocupado, verificamos se podemos inserir o dado na posição $\sigma(h(k)+1)$ da tabela. Se esta posição estiver ocupada, verificamos $\sigma(h(k)+2)$, e assim por diante...

- A implementação da função `find`(k) também é trivial

  Percorremos o array a partir de $\sigma(h(k))$, até que o elemento seja encontrado. A busca é abortada quando uma posição vazia da tabela é encontrada

  Invariante: Se k está armazenado em $\sigma(t)$ tal que $t > h(k)$, então os índices de σ de $h(k)$ até $t-1$ estão ocupados

Exemplo

- h(k) = *k* mod 13
- Insira chaves:
- 18 41 22 44 59 32 31 73

0	1	2	3	4	5	6	7	8	9	10	11	12

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

- Usa menos memória do que o encadeamento: não precisa armazenar todos os ponteiros!
- Mais lento do que o encadeamento: pode ser que tenhamos de percorrer a tabela σ por vários índices
- A operação `remove`(k) é mais complexa
  - ou marcamos com um *flag* o índice da tabela referente ao elemento removido (ineficiente em termos de performance para a busca)
  - ou restauramos o invariante deslocando elementos à esquerda até que uma nova posição livre da tabela seja encontrada

Fim do módulo 5