TAD dicionário

- O TAD dicionário modela uma coleção “buscável” de itens chave-elemento
- As principais operações em dicionários são busca, inserção e remoção de itens
- Vários itens com a mesma chave são permitidos
- Aplicações:
  - Agenda
  - Autorização de cartão de crédito
  - Mapeamento de *hosts* (e.g., cs16.net) para endereços IP (e.g., 128.148.34.101)

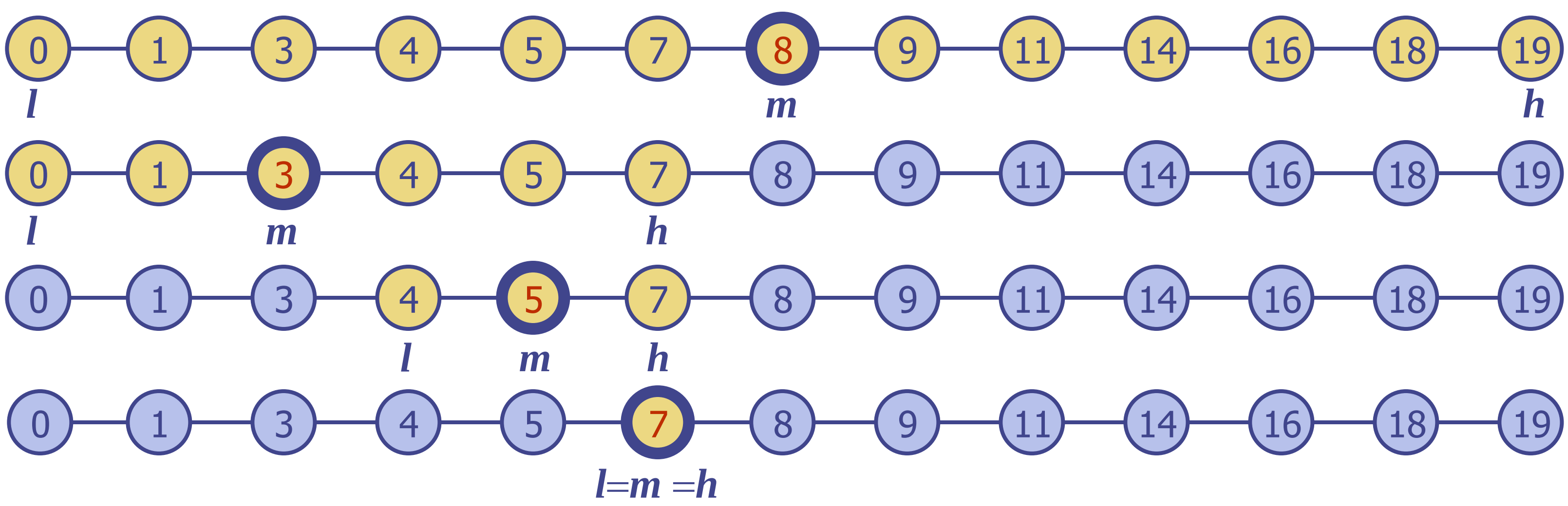
- Métodos do TAD dicionário:
  - findElement(k): se o dicionário tem um item com chave k, retorna o elemento, senão, retorna NO_SUCH_KEY
  - insertItem(k, o): insere um item no dicionário
  - removeElement(k): Se existe um item com chave k, remove e retorna, senão, retorna NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), Elements()

Arquivo de LOG

- Um arquivo de log é um dicionário implementado como um sequência não ordenada
  - Armazenamos os itens do dicionário em uma sequência em uma ordem arbitrária
- Desempenho:
  - insertItem roda em tempo $O(1)$ uma vez que podemos inserir o novo item no início ou no fim da sequência
  - findElement e removeElement roda em tempo $O(n)$ uma vez que, no pior caso, temos que percorrer toda a sequência para procurar um item com uma dada chave
- O arquivo de log é útil apenas para dicionários de tamanho pequeno ou para dicionários nos quais operações de inserção são as mais comuns, enquanto busca e remoção são raras (histórico de registros de logins em uma estação)

Busca binária

- Busca binária realiza a operação findElement(k) em um dicionário implementado com uma sequencia baseada em array, ordenada pela chave
  - a cada passo, o número de itens candidatos é dividido pela metade
  - termina após um número logaritmico de passos
- Example: findElement(7)



Busca binária

Algoritmo

```
Algoritmo BuscaBinária(A,k,min,max)
    m ←(max+min)/2
    c ← A[m]
    se min>max
        retorne NO_SUCH_KEY
    senão se c.getKey()= k
        retorne c.getElement()
    senão se(k < c.getKey()
        BuscaBinaria(A,k, min, m-1)
    senão se(k>c.getKey())
        BuscaBinaria(A,k, m +1,max)
    retorne NO_SUCH_KEY
```

Tabela de Pesquisa

- Uma tabela de pesquisa é um dicionário implementado através de uma estrutura ordenada
  - Itens são armazenados (implementada com arranjo) ordenados pela chave
  - Usamos um comparador externo para as chaves
- Desempenho:
  - findElement executa em tempo $O(\log n)$, usando busca binária
  - insertItem executa em tempo $O(n)$ uma vez que no pior caso, deve-se deslocar n itens para liberar espaço para o novo item
  - removeElement executa em tempo $O(n)$ uma vez que no pior caso, temos que deslocar $n/2$ itens para ocupar o espaço do item removido
- Uma tabela de pesquisa é eficiente apenas para dicionários de tamanho pequeno ou dicionários onde a busca é a operação mais comum, enquanto que inserções e remoções são realizadas raramente (autorizações de cartões de créditos, por exemplo)

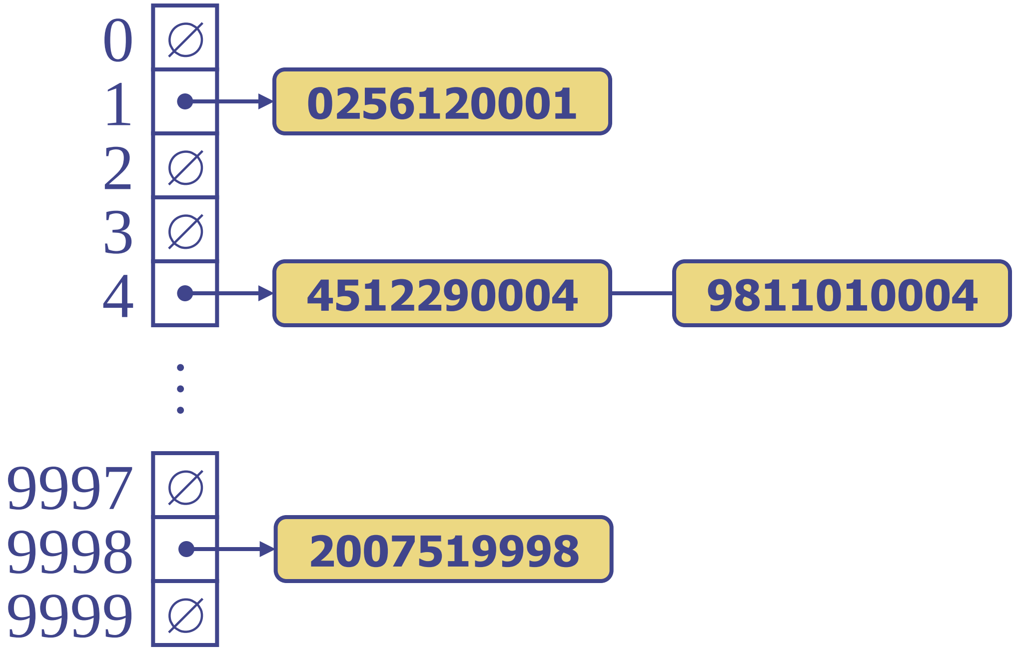
Tabelas e funções de dispersão

- Uma função de dispersão h mapeia chave de um dado tipo em inteiros num intervalo fixo $[0, N-1]$
- Exemplo:
  $$h(x) = x \bmod N$$
  é uma função de dispersão para chaves inteiras
- O inteiro $h(x)$ é chamado valor de dispersão (hash value) da chave x
- O objetivo da função de dispersão é uniformemente dispersar chaves na faixa $[0, N-1]$

- Uma tabela de dispersão para um dado tipo de chave consiste de
  - função de dispersão h
  - arranjo (chamado tabela) de tamanho N
- Quando implementando um dicionário com uma tabela de dispersão, o objetivo é armazenar itens (k, o) no índice $i = h(k)$
- Uma colisão ocorre quando duas chaves no dicionário têm o mesmo valor de dispersão
- Esquemas de tratamento de colisões:
  - encadeiamento: itens que colidem são armazenados numa sequência
  - endereçamento: o item que colide é colocado em um lugar diferente na tabela

Exemplo

- Projetamos uma tabelas de dispersão para armazenar itens (ID, nome), onde ID é um inteiro positivo de nove dígitos
- Nossa tabela de dispersão usa arranjo de tamanho $N = 10{,}000$ e a função de dispersão $\boldsymbol{h}(\boldsymbol{x})$ = quatro últimos dígitos de $\boldsymbol{x}$
- Usamos encadeiamento para tratar colisões



Funções de dispersão

- Uma função de dispersão é normalmente composta de duas funções:

  Mapa de código de dispersão:

  h_1: chaves $\rightarrow$ inteiros

  Mapa de compressão:

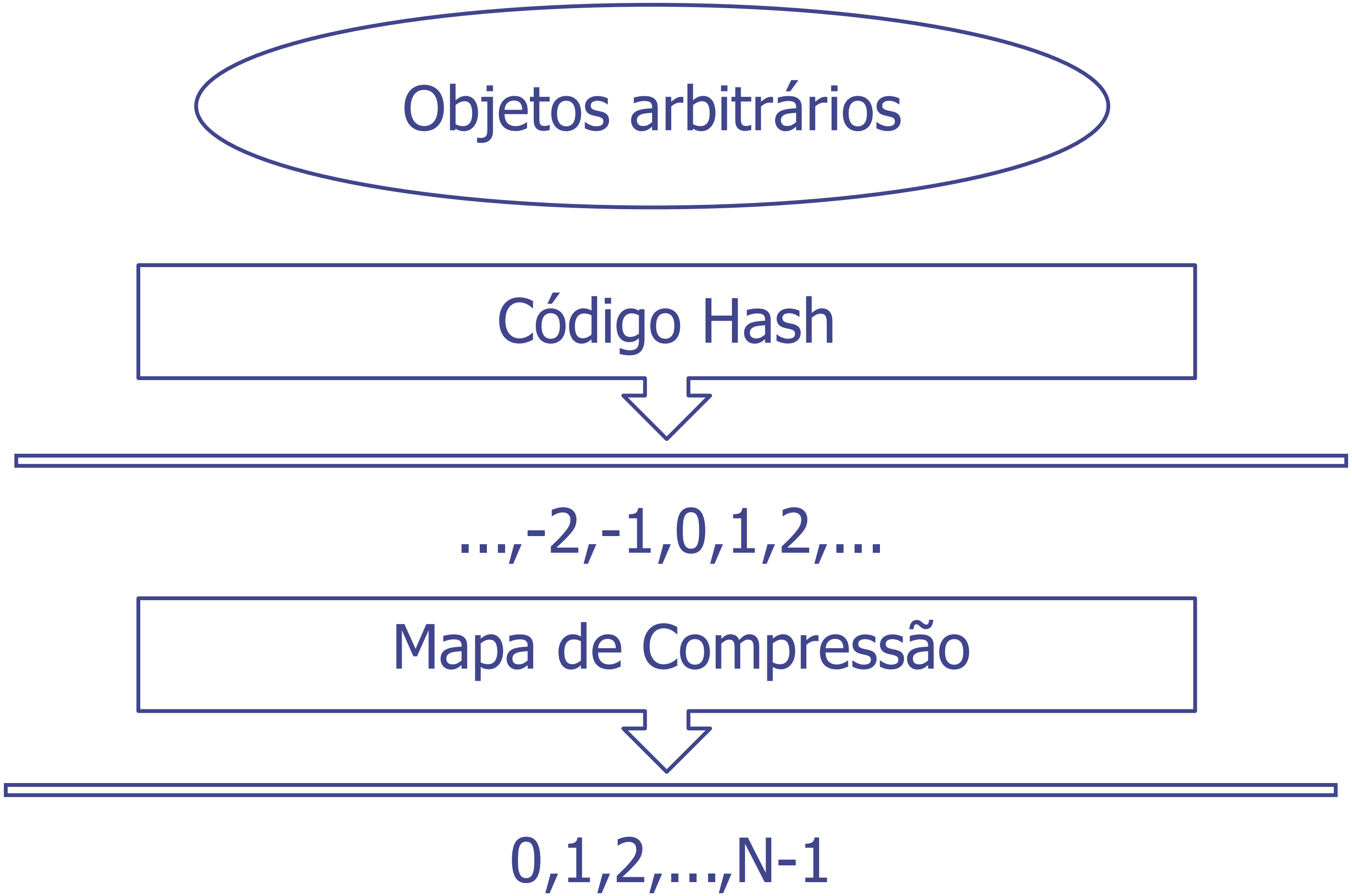
  h_2: inteiros $\rightarrow [0, N-1]$

- O mapa de código de dispersão é aplicado primeiro, e o mapa de compressão e aplica logo após no resultado

  $$h(x) = h_2(h_1(x))$$

- O objetivo da função de dispersão é "dispersar" as chaves de forma aparentemente aleatória

Funções de dispersão



Códigos de dispersão (hash code)

- Endereço de memória:
  - Interpretamos o endereço de memória da chave como inteiro (usado pelo método *hashCode* de Object)
  - Bom no geral, mas pode ter códigos diferentes para a mesma chave (String, Números, etc)
- Conversão para inteiros:
  - Interpretamos os bits da chave como um inteiro
  - Adaptável para chaves cujo tamanho é menor ou igual ao número de bits de um inteiro (byte, short, int e float em Java)
- Soma de componentes:
  - Particiona-se os bits da chave em componentes de tamanho fixo (16 oiu 32 bits) e somamos os componentes
  - Adaptável para números de tamanho fixo maior ou igual os números de bits do tipo inteiro (long e double em Java)

Códigos de dispersão (cont.)

- **Acumulação polinomial:**
  - Particiona-se os bits da chave em uma sequencia de componentes de tamanho fixo

    $$a_0\, a_1\, \ldots\, a_{n-1}$$

  - Avalia-se o polinomio

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
    $$\ldots + a_{n-1} z^{n-1}$$

    sobre uma constante z, ignorando *overflows*
  - Plenamente adaptável a String

- O polinomio $p(z)$ pode ser avaliado em tempo $O(n)$ usando a regra de Horner:
  - Os seguintes polinomios são computados sucessivamente, usando o anterior em tempo $O(1)$

    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n - 1)$$

Mapa de compressão

- Divisão:
  - $h_2(y) = y \bmod N$
  - O tamanho de N da tabela de dispersão é geralmente um número primo
  - Motivo de ser primo está no estudo da teoria dos números e não entraremos em detalhes
- Multiplicação, Adição e Divisão (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a e b são inteiros não negativos tais que
    $a \bmod N \neq 0$
  - de outra forma, todo inteiro mapearia para o mesmo valor b

“Linear Probing”

- “Linear probing” trata colisões colocando o item que colide na próxima (circular) célula disponível
- Itens que colidem ficam juntos causando uma longa sequência de “probes”

- Exemplo:
  - $h(x) = x \bmod 13$
  - Insira as chaves 18, 41, 22, 44, 59, 32, 31, 73, nessa ordem

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

busca com “Linear Probing”

- Considere uma tabela de dispersão A que usa linear probing
- findElement(k)
  - Começamos na célula $h(k)$
  - Verificamos localizações consecutivas até encontrar uma que acontece:
    - Um item com chave k é encontrado ou
    - Uma célula vazia é encontrada, ou
    - N células tenham sido verificadas

Algoritmo *findElement*(k)
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repita
    $c \leftarrow A[i]$
    se $c = \varnothing$
      retorne *NO_SUCH_KEY*
    senão se $c.key() = k$
      retorne $c.element()$
    senão
      $i \leftarrow (i + 1) \bmod N$
      $p \leftarrow p + 1$
  até $p = N$
  retorne *NO_SUCH_KEY*

Atualização com “Linear Probing”

- para manipular inserções e remoções, usamos um objeto especial, chamado ***AVAILABLE***, que substitue elementos removidos
- removeElement(***k***)
  - Procura-se por um item com chave ***k***
  - Se o item é encontrado, substitue ele com o objeto especial ***AVAILABLE*** e retorna-se o elemento ***o***
  - Senão, retorna-se ***NO_SUCH_KEY***
- insert Item(***k***, ***o***)
  - Uma exceção é disparada se a tabela está cheia
  - Começa-se na célula ***h***(***k***)
  - Procura-se em consecutivas células até que o seguinte ocorra:
    - Uma célula ***i*** é encontrada e está vazia ou armazena ***AVAILABLE***, ou
    - ***N*** células tenham sido verificadas
  - Armazena-se o item na célula ***i***

hashing duplo

- *hashing* duplo usa uma função de dispersão secundária $\boldsymbol{d}(\boldsymbol{k})$ e manipula colisões colocando o item na primeira célula disponível da série
  $$(\boldsymbol{i} + \boldsymbol{j}\boldsymbol{d}(\boldsymbol{k})) \bmod \boldsymbol{N}$$
  para $\boldsymbol{j} = 0,\ 1, \dots, \boldsymbol{N} - 1$
- A função de dispersão secundária $\boldsymbol{d}(\boldsymbol{k})$ não pode ter valores zero
- O tamanho $\boldsymbol{N}$ da tabela deve ser primo para permitir verificação de todas as células
- Uma escolha comum de mapa de compressão para a função secundária é: $\boldsymbol{d}_2(\boldsymbol{k}) = \boldsymbol{q} - \boldsymbol{k} \bmod \boldsymbol{q}$
  onde
  - $\boldsymbol{q} < \boldsymbol{N}$
  - $\boldsymbol{q}$ é primo
- valores possíveis para $\boldsymbol{d}_2(\boldsymbol{k})$ são
  $$1, 2, \dots, \boldsymbol{q}$$

Exemplo de *Hashing* duplo

- Considere uma tabela de dispersão armazenando chaves inteiras e manipulando colisões com *hashing* duplo
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insere as chaves 18, 41, 22, 44, 59, 32, 31, 73, nessa ordem

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

⇩

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

Desempenho de dispersão

- No pior caso, busca, inserção e remoção em uma tabela de dispersão roda em tempo $O(n)$
- O pior caso ocorre quando todos os itens inseridos em um dicionário colidem
- O fator de carga $\alpha = n/N$ afeta o desempenho
- Assumindo que os valores de dispersão são como números aleatórios, é possível mostrar que o número de "probes" para uma inserção com endereço aberto é
  $$1 / (1 - \alpha)$$
- O tempo de execução esperado para todas as operações de um dicionário em uma tabela de dispersão é $O(1)$
- Na prática, dispersão é muito rápida
- Aplicações de tabelas de dispersão:
  - pequenos bancos de dados
  - compiladores
  - caches de navegadores