ESTRUTURAS DE DADOS

Conceitos de Tabela Hash

Roteiro

- Motivação
- Tipo Abstrato de Dados
- Detalhes de Implementação
- Funções de Hash

Motivação

- **Sabemos que:**
  - **Busca sequencial executa em tempo O(n).**
  - **Busca binária executa em tempo O(log(n)).**
  - **Busca binária exige arranjo ordenado.**

- **Seria possível efetuar uma busca em tempo melhor do que O(log(n))?**

- **Quais restrições devem existir sobre os dados?**

- **Tabelas de Hash (ou Tabelas Hash) permitem buscas em tempo constante, satisfeitas algumas restrições.**

- **Essa estrutura pode ter vários nomes como: dicionários, mapas, arrays associativos, e assim por diante.**

- **A princípio, a chave de busca pode ser de qualquer tipo.**

Tipo Abstrato de Dados

- **retrieveItem(*k*): retorna uma entrada com chave igual a *k*, se ela existir. Caso contrário, retorna nulo.**

- **insertItem(*k*, *v*): insere uma entrada *v* na chave *k* se a chave não existir. Caso contrário, atualiza o valor associado a *k*.**

- **deleteItem(*k*): remove a chave *k* e o valor associado a ela.**

Outros Métodos:

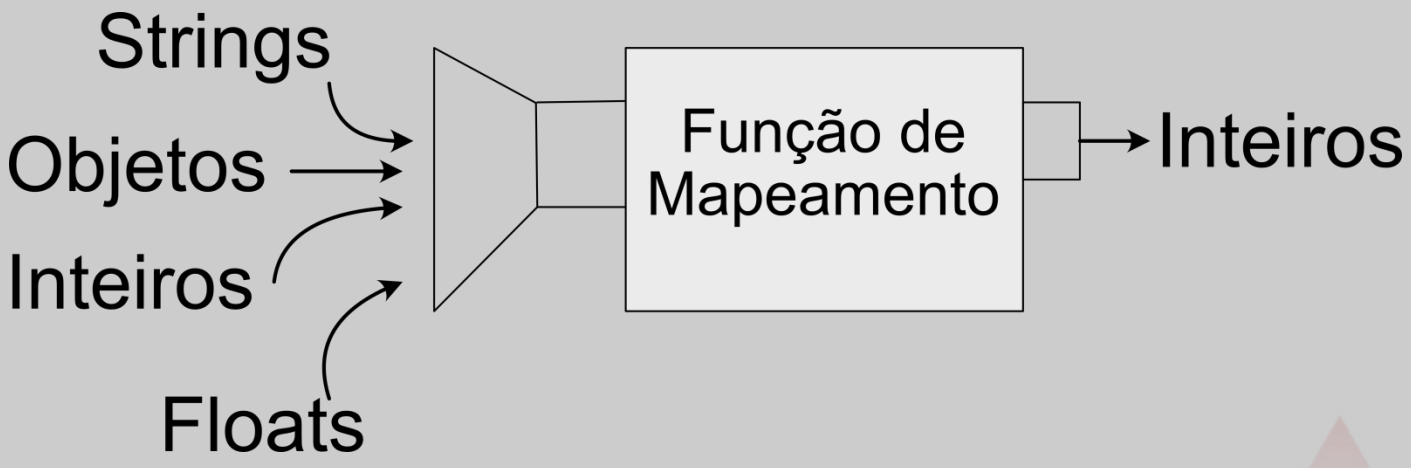
- **size(): retorna o número de entradas.**
- **keySet(): retorna uma lista encadeada de todas as chaves armazenadas na tabela.**
- **values(): retorna uma coleção contendo todos os valores associados com as chaves armazenadas na tabela.**
- **entrySet(): retorna uma coleção contendo todas as entradas (chave-valor) da tabela.**

Implementação

- **A própria chave deve ser usada para organizar os dados em memória.**
- **Cada entrada da estrutura é composta por um par "chave-valor" (*k*, *v*). A associação entre *k* e *v* define o mapeamento.**
- **A chave é um identificador único e deve ser vista como um "endereço" para seu valor.**

- **A tabela pode ser organizada em memória como um arranjo, dado que este permite acesso em tempo constante.**

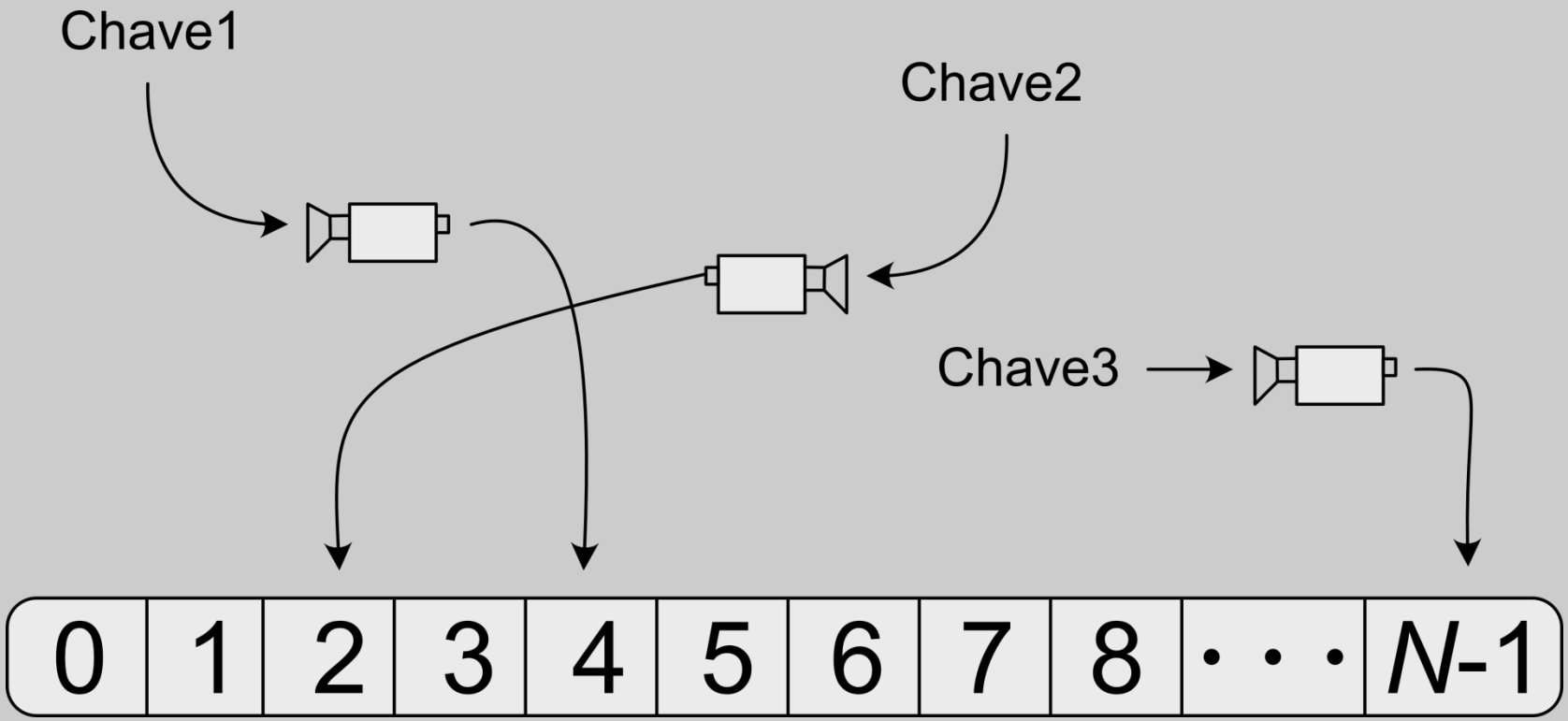
- **As chaves podem ser de qualquer tipo de dados, mas para efetuarmos a busca no arranjo, precisaremos de uma função que mapeie chaves em números inteiros.**



- **Seja *h* a função que faz o mapeamento (também chamada de função de espalhamento) e *k* a chave, o endereço de memória será dado por *h*(*k*).**

- **Se os valores retornados por *h*(*k*) forem bem distribuídos em um intervalo entre 0 e *N*-1, então precisamos de um array de capacidade *N*.**

- **Assumindo ausência de colisões, essa estrutura básica seria suficiente.**


Chave1
Chave2
Chave3
0
1
2
3
4
5
6
7
8
• • •
N-1

Funções de Hash

- **A função de hash *h* mapeia cada chave em um intervalo de 0 a *N*-1, onde *N* é a capacidade do arranjo.**
- **É possível tratar colisões, mas a melhor estratégia é evitá-las.**
- **Uma função de hash é boa se minimiza a ocorrência de colisões.**

- **A primeira tarefa da função será transformar chaves de tipos arbitrários em inteiros.**

- **Vamos assumir que queremos armazenar informações de funcionários de uma empresa e indexar essas informações pelo *login* único da pessoa.**

- **O login pode ser o primeiro nome da pessoa, mas se este já foi escolhido por alguém, então outro deve ser selecionado pelo funcionário.**

- **Uma função de hash pode primeiramente mapear os caracteres para inteiros.**

A	65		a	97		0	48
B	66		b	98		1	49
C	67		c	99		2	50
D	68		d	100		3	51
E	69		e	101		4	52
F	70		f	102		5	53
G	71		g	103		6	54

ulisses: u + l + i + e + 3* s

117 + 108+ 105 + 101 + 3*115 = 776

- **Podemos mapear qualquer *login* em inteiro**

ulisses	776
danielle	830
amanda	610
cleópatra	1218

- **O valor inteiro encontrado pode ser o índice da entrada em um arranjo.**

- **Essa ideia ilustra uma implementação básica da tabela hash.**

- **Essa estratégia gera colisões:**

orlando	751
odnalro	751
adriana	720
ariadna	720

- **Uma função de hash melhor levaria em conta a posição dos caracteres c_i na cadeia $C = (c_0,c_1,c_2,\ldots,c_{k-1})$.**

$$c_0 a^{k-1} + c_1 a^{k-2} + c_2 a^{k-3} + \ldots + c_{k-2} a^1 + c_{k-1}$$

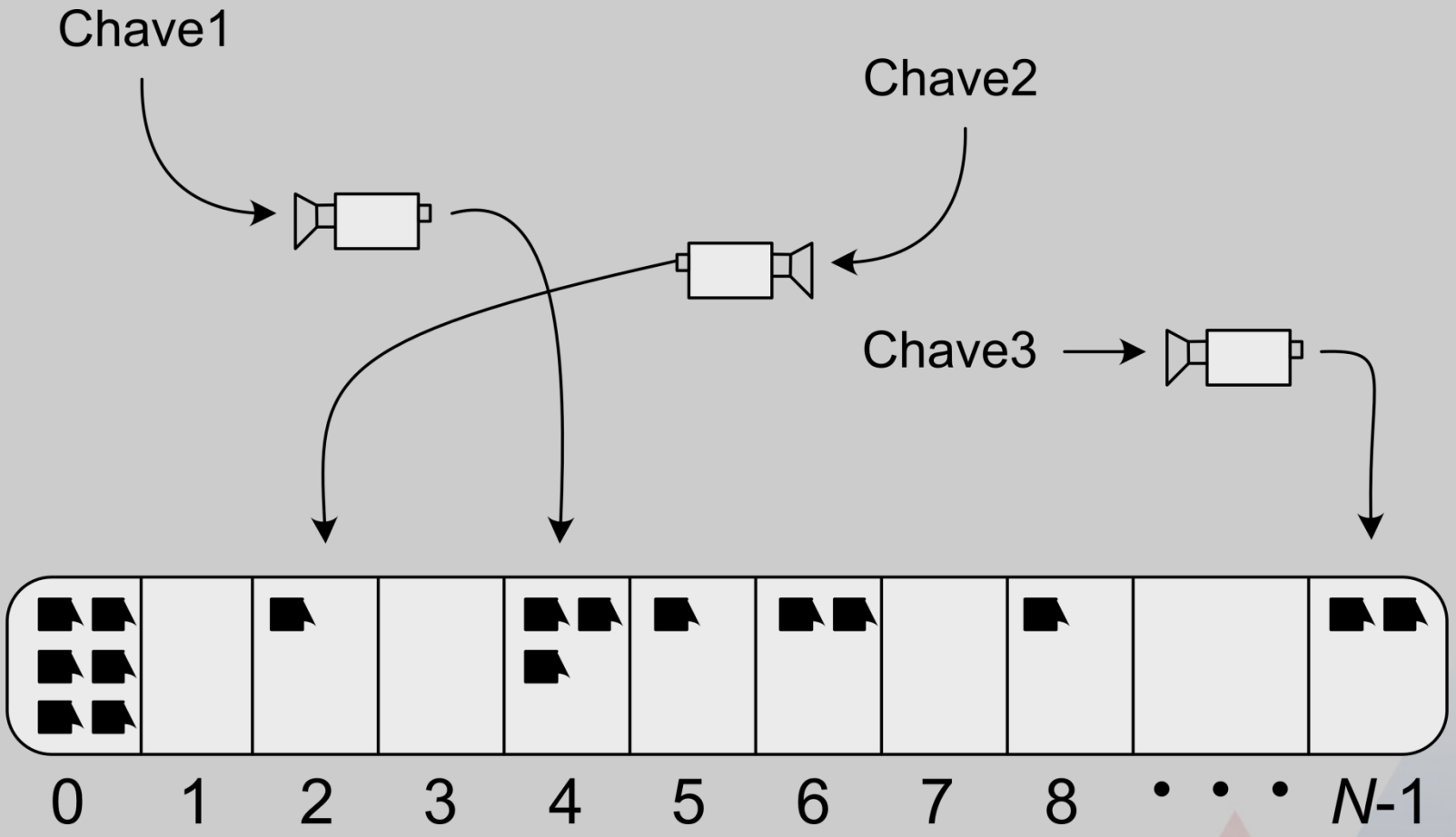
Para algum *a* diferente de 0 ou 1.

- **Por exemplo, com *a* = 3, teríamos os seguintes valores para orlando e odnalro:**

o	111	x	729	+		o	111	x	729	+	
r	114	x	243	+		d	100	x	243	+	
l	108	x	81	+		n	110	x	81	+	
a	97	x	27	+		a	97	x	27	+	
n	110	x	9	+		l	108	x	9	+	
d	100	x	3	+		r	114	x	3	+	
o	111	x	1	=	121389	o	111	x	1	=	118173

- **Um valor de *a* alto (33, 37, 39 ou 41) tende a diminuir o número de colisões para alguns poucos.**

- **Nesse caso, é possível fazer com que cada endereço tenha espaço para mais de uma entrada.**



- **Um valor de *a* alto (33, 37, 39 ou 41) pode levar a um *overflow* do intervalo dos inteiros.**

- **A compressão dos valores pode fazer parte da função. O resto da divisão por *N* estabiliza os valores em um intervalo [0 .. *N*-1].**

$$i \bmod N$$

- **O tamanho do arranjo *N* pode aumentar ou diminuir o número de colisões:**
  - **Se usarmos *N* = 1000, teremos muito menos colisões do que com *N* = 100.**

- **Para ajudar o espalhamento das chaves, é interessante usar um número primo para *N*. Isso diminui a chance de ocorrer padrões na distribuição de dados.**

- **Por exemplo, se temos as chaves {200, 205, 210, 215, 220,..., 600}.**
  - **Com *N* = 100, cada chave irá colidir com três outras chaves.**
  - **Com *N* = 101 não teremos colisões.**

- **Na próxima aula, implementaremos esses conceitos em C++. Em seguida, veremos técnicas de tratamento de colisão.**

ESTRUTURAS DE DADOS

Conceitos de Tabela Hash