Sistemas de Informação | AMF
Classificação e Pesquisa de Dados

Aula 11 - *Hash tables*

Cristiano Santos

Baseado no material de proposto por Rhauani Fazul

Conteúdo Programático

1. Métodos de Classificação de Dados

2. Pesquisa de Dados

1. Famílias de métodos de pesquisa de dados
2. Pesquisa sequencial (pesquisa linear)
3. Pesquisa binária
4. Pesquisa digital
5. Árvores de busca
    1. Árvores binárias de pesquisa sem balanceamento
    2. Árvores binárias de pesquisa com balanceamento
6. Tabelas de dispersão (*hash tables*)
    1. Funções de transformação de chave (*hashing*)
    2. Cálculo de endereços e tratamento de colisões
    3. Endereçamento aberto
    4. Listas encadeadas
    5. Hashing perfeito
7. Pesquisa de dados em memória secundária
    1. Acesso sequencial indexado
    2. Árvores de pesquisa
    3. Árvores-B
    4. Árvores-B*

3. Introdução à Análise da Complexidade de Algoritmos

Agenda

- Contextualização
- *Hash tables*
- Tratamento de colisões
- *Hands-on*

Contextualização

- Até agora vimos alguns métodos de pesquisa baseados na **comparação da chave** de busca com as chaves já armazenadas na ED ou em **processamento feito sob a chave**:
  - sequencial → O(n); busca binária → O(log n), com o vetor já ordenado;
  - árvores (em geral) → O(log n), ED permite inserir/adicionar elementos de forma eficiente, mas pode ficar desbalanceada com o passar do tempo.
- Podemos dizer que **algoritmos eficientes** armazenam os elementos ordenados e tiram proveito dessa ordenação:
  - Limite inferior da complexidade de tempo dos algoritmos que vimos até o momento → O(log n)

- É possível fazer melhor?

Contextualização

- Vetores utilizam índices para acessar as informações armazenadas;
- Através do índice, as operações são realizadas em tempo constante O(1);
- Porém, vetores não fornecem mecanismos para calcular o índice a partir de uma informação armazenada. Logo, a pesquisa não é O(1)

0	1	2	3	4	5
Ana	Bruna	Bruno	Carlos	João	Sabrina

Como descobrir que a chave "Sabrina" está no índice 5?

Contextualização

- Vetores utilizam índices para acessar as informações armazenadas;
- Através do índice, as operações são realizadas em tempo constante O(1);
- Porém, vetores não fornecem mecanismos para calcular o índice a partir de uma informação armazenada. Logo, a pesquisa não é O(1)

0	1	2	3	4	5
Ana	Bruna	Bruno	Carlos	João	Sabrina

Como descobrir que a chave "Sabrina" está no índice 5?

- **Ideal:** (parte da) chave poderia ser utilizada para recuperar diretamente a informação! Mas como fazer isso?
  - Queremos evitar "desperdício" de posições (*gaps*).

Agenda

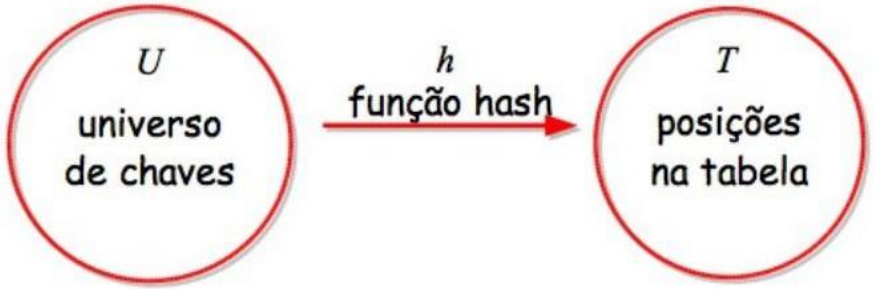
- Contextualização
- *Hash tables*
- Tratamento de colisões
- *Hands-on*

Hash tables

- Tabelas *hash* (ou tabelas de dispersão/espelhamento) são uma ED onde os registros armazenados são endereçados a partir de uma **transformação** aritmética sobre a chave de pesquisa.
- **Objetivo:** permitir o acesso a dados de maneira extremamente **eficiente**, utilizando uma função de hashing para converter a chave em um índice no *array* de armazenamento:
  - complexidade constante O(1) nas operações de busca, inserção e remoção;
  - encontrar a chave com apenas uma única operação de leitura.
- Estrutura especializada em prover operações de inserir, pesquisar e remover, amplamente utilizada nos mais variados contextos:
  - diversas aplicações, como em bancos de dados para realizar buscas rápidas, em caches para armazenar e recuperar dados eficientemente, compiladores para gerenciar as tabelas de símbolos, criptografia, armazenamento de senhas, compactação de dados, transações financeiras, ...

Hash tables

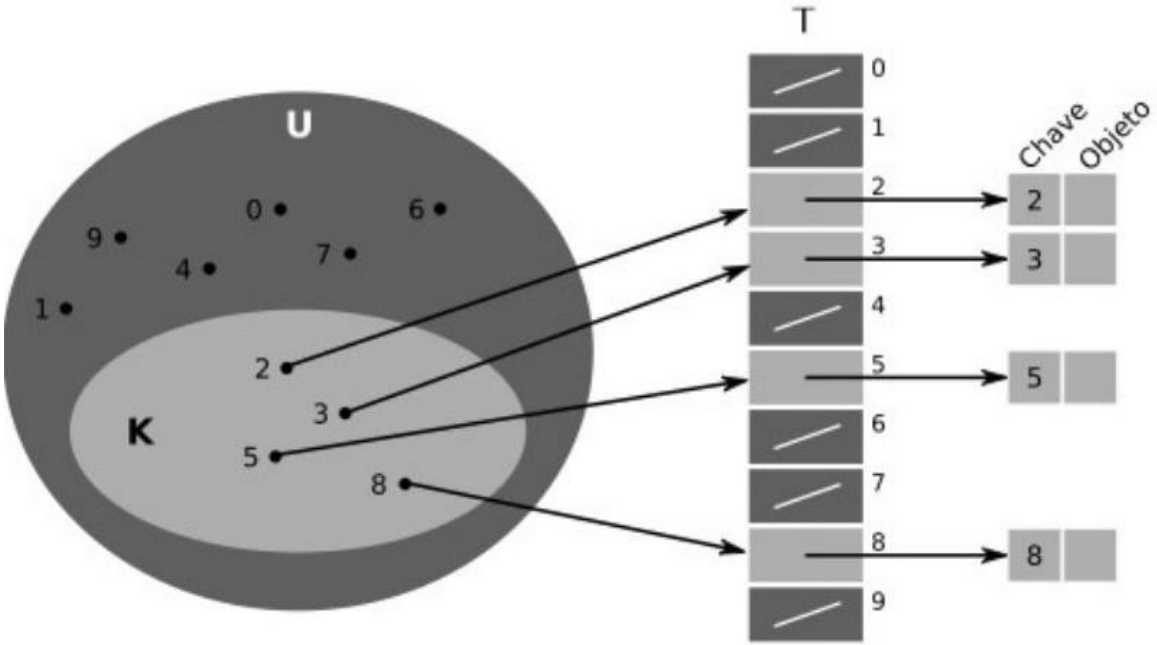
- A ED mapeia **chaves** → **valores**: a ideia central é utilizar uma função (aplicada sobre parte da chave) para retornar o índice onde a informação deve (ou deveria) estar armazenada:
  - ***hash function*** é a função que mapeia a chave para um índice (*hashing*).
- Cada entrada possui uma chave e um valor associado. A chave é única e mapeada para um valor específico;
- Precisamos computar o valor da função de dispersão (método de cálculo de endereço), a qual transforma a chave de pesquisa em um endereço da tabela.



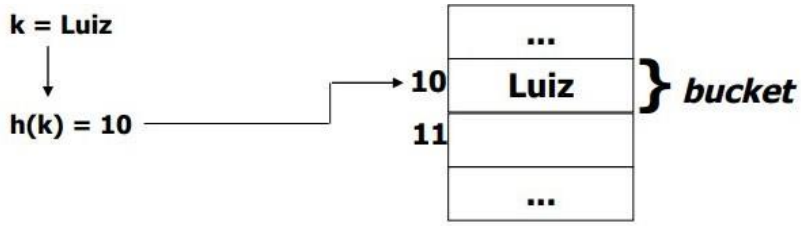
Leiam sobre **entropia** no contexto da informação e seu impacto na criptografia.

Endereçamento direto

- Quando o universo de chaves U é pequeno, podemos alocar uma tabela com uma posição para cada chave, ou seja, |T| = |U|
- Cada posição da tabela, que pode ser implementada como um vetor, representa uma chave de U e armazena um elemento ou um ponteiro para o elemento.



Pensamento inicial...



- Imagine um pequeno país (bem menos que 100 mil cidadãos) onde os números de CPF têm apenas 5 dígitos decimais;
- Considere a tabela que leva CPFs em nomes:

chave	valor associado
01555	Ronaldo
01567	Pelé
...	...
80114	Maradona
80320	Dunga
95222	Romário

Para aquecer, vamos implementar essa lógica em Python

- Como armazenar a tabela? Resposta: vetor de 100 mil posições:
  - Use a própria chave como índice do vetor!

- O vetor é conhecido com tabela de *hash* e terá muitas posições vagas (desperdício de espaço), mas as operações de busca e inserção serão extremamente rápidas.

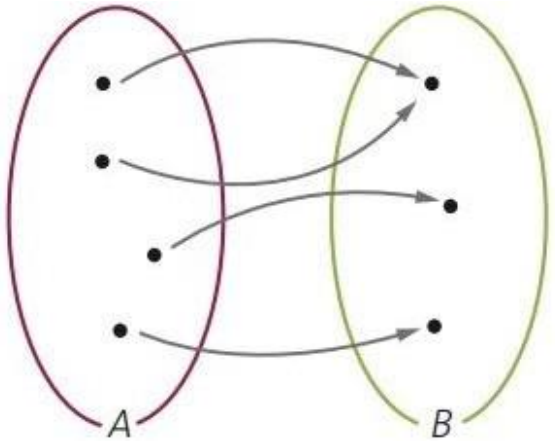
Endereçamento direto

- **Problema:** nem sempre U é pequeno:
  - Imagine um identificador de 7 dígitos;
  - São 10.000.000 chaves (ou elementos) → tamanho do vetor;
  - Se cada posição da tabela ocupar 127 bytes, precisamos de mais de 1 Gbyte de memória apenas para a tabela... mesmo que ela não esteja cheia.

- Sendo K o conjunto de chaves que serão efetivamente armazenadas na tabela, a tabela deveria ter dimensão |K|, mais que isso seria desperdício de memória;
- Na prática, os elementos de K não são conhecidos e |U| >> |K|
- Então, como podemos fazer isso?

Hashing imperfeito

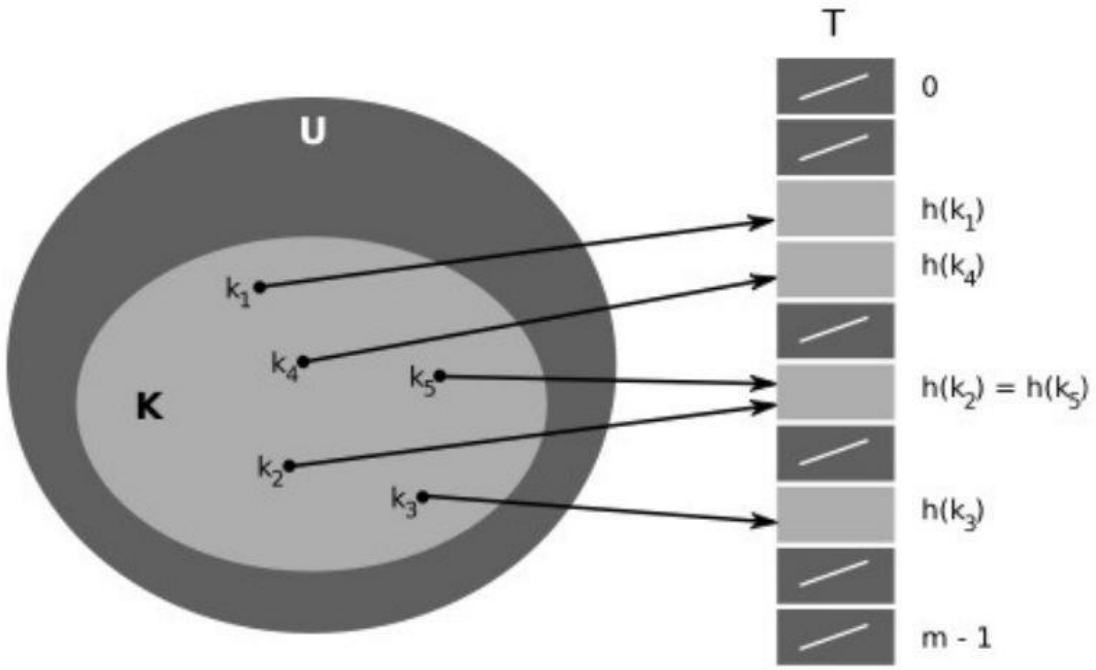
- Existem chaves x e y, diferentes e pertencentes a A, onde a função *hash* utilizada fornece saídas iguais.



Função Sobrejetora

Funções de *hashing*

- Podemos utilizar uma função hash h para mapear chaves em inteiros dentro do intervalo [0..m – 1], no qual m é o tamanho da tabela.
- A tabela é implementada como um vetor em que cada posição armazena um subconjunto de U.



Funções de *hashing*

- **Exemplo:** Se as chaves são inteiros positivos, podemos usar a função modular (resto da divisão por M):

```
def hash(key, M):
        return key % M;
```

- Exemplos com M = 100 e com M = 97:

key	hash (M = 100)	hash (M = 97)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Modular hashing

Funções de *hashing*

- Em *hashing* modular, é bom que M seja primo (por algum motivo não óbvio).
- No caso de strings, podemos iterar hashing modular sobre os caracteres:

```
def string_hash(s, M):
    h = 0
    for char in s:
        h = (31 * h + ord(char)) % M
    return h
```

No lugar do multiplicador 31, poderia usar qualquer outro inteiro R, de preferência primo, mas suficientemente pequeno para que os cálculos não produzam *overflow*).

Funções de *hashing*

- **Pergunta:** supondo que se deseja armazenar **n** elementos em uma tabela de **m** posições, qual o número esperado de elementos por posição na tabela?

Funções de *hashing*

- **Pergunta:** supondo que se deseja armazenar **n** elementos em uma tabela de **m** posições, qual o número esperado de elementos por posição na tabela?

Fator de carga $\alpha = n/m$

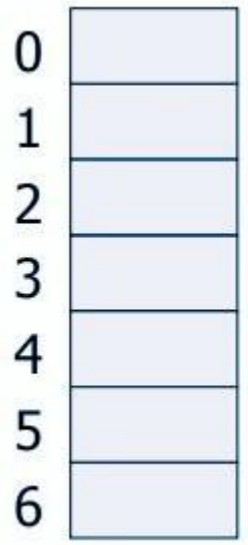
- A biblioteca padrão de Java usa 0.75 como fator de carga para decidir quando fazer o *resize* da tabela*. Isso significa que quando a tabela atinge 75% de ocupação o *resize* é executado. **Por quê?**

*é um tipo de *hashing* **dinâmico** (não é o foco na disciplina), onde o espaço de endereçamento pode aumentar.

Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números 1, 5, 10, 20, 25, 24

```
def hash(key, M):
        return key % M;
```

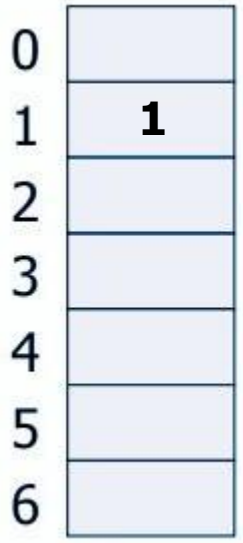


Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números **1**, 5, 10, 20, 25, 24

```
def hash(key, M):
        return key % M;
```

1 % 7 = 1



Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números 1, **5**, 10, 20, 25, 24

```
def hash(key, M):
        return key % M;
```

5 % 7 = 5

0	
1	1
2	
3	
4	
5	**5**
6	

Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números 1, 5, **10**, 20, 25, 24

```
def hash(key, M):
        return key % M;
```

10 % 7 = 3

0	
1	1
2	
3	**10**
4	
5	5
6	

Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números 1, 5, 10, **20**, 25, 24

```
def hash(key, M):
        return key % M;
```

20 % 7 = 6

0	
1	1
2	
3	10
4	
5	5
6	**20**

Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números 1, 5, 10, 20, **25**, 24

```
def hash(key, M):
        return key % M;
```

25 % 7 = 4

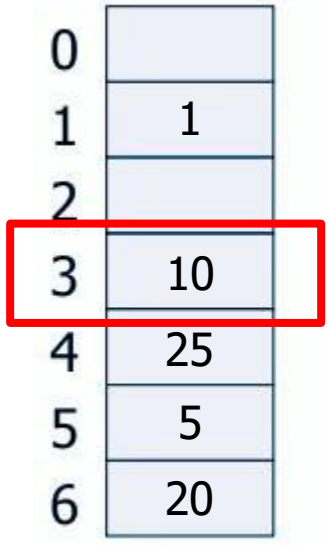
0	
1	1
2	
3	10
4	**25**
5	5
6	20

Exemplo prático

- Seja B um arranjo de 7 elementos:
  - Inserção dos números 1, 5, 10, 20, 25, **24**

```
def hash(key, M):
        return key % M;
```

24 % 7 = 3



Colisão!

Agenda

- Contextualização
- *Hash tables*
- Tratamento de colisões
- *Hands-on*

Colisões

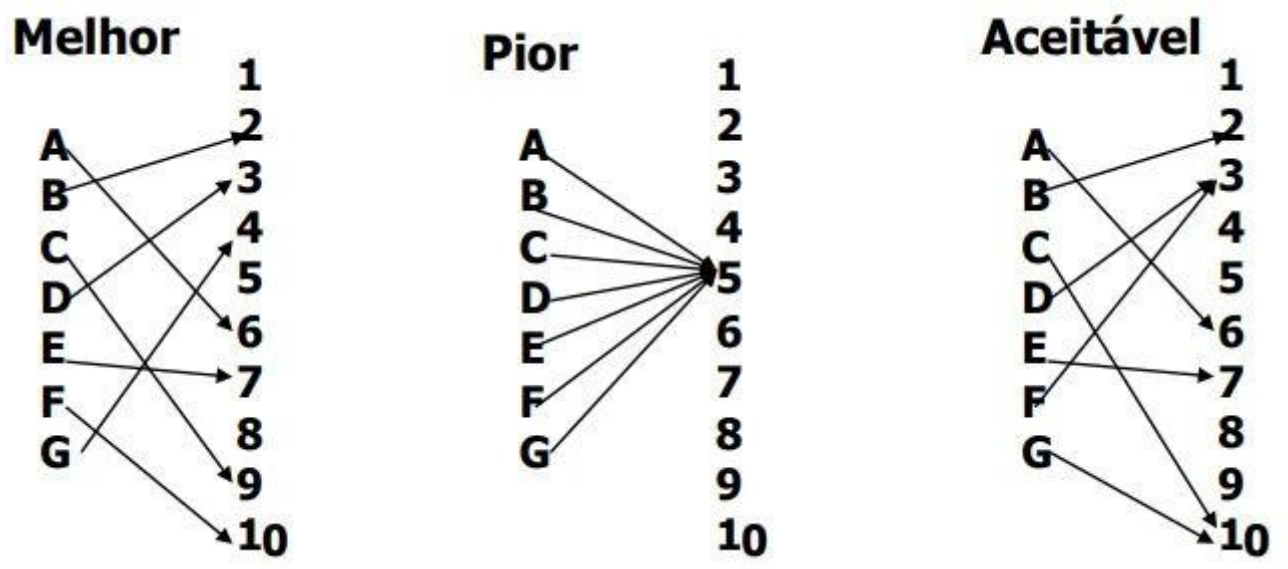
- É possível que mais de uma chave seja mapeada em uma única posição da tabela, o que resulta no que chamamos de **colisão**:
  - Ocorre quando duas chaves diferentes são mapeadas para o mesmo índice no *array*. O tratamento adequado de colisões é crucial para manter a eficiência das operações. Por quê?

Colisões

- É possível que mais de uma chave seja mapeada em uma única posição da tabela, o que resulta no que chamaremos de **colisão**:
  - Ocorre quando duas chaves diferentes são mapeadas para o mesmo índice no *array*. O tratamento adequado de colisões é crucial para manter a eficiência das operações.

Colisões

- Quando duas ou mais chaves geram o mesmo endereço (índice) na tabela *hash* temos uma colisão. E isso é comum. Principais causas:
  - número N de chaves é **muito maior** que o número de entradas na tabela;
  - falta de garantias que a função de hashing possua um bom potencial de distribuição (espalhamento), dado a natureza das chave.
- Um bom método de r**esolução de colisões** é **essencial**, não importando a qualidade da função de *hashing.*



Chaves com mesmo endereço são conhecidas como "sinônimos"

Colisões

- **Segredos** para um bom *hashing*:
  - Escolher uma boa função *hash* (em função dos dados):
    - Distribuia uniformemente os dados, na medida do possível;
    - Que evita colisões;
    - Seja fácil/rápida de computar.
  - Estabelecer uma boa estratégia para tratamento de colisões.
- **Diversas técnicas para resolução de colisão**, sendo as mais comuns para *hashing* **estático** (foco na disciplina), onde o espaço de endereçamento não muda, enquadradas em duas categorias:
  - **Fechado:**
    - Técnicas de *rehash* (endereçamento aberto) para tratamento de colisões.
  - **Aberto:**
    - Encadeamento de elementos para tratamento de colisões.

Hashing fechado

- Com o endereçamento aberto (*rehash*), a informação é armazenada na própria tabela *hash*;
- Exemplos:
  - *Rehash* linear (verificar o próximo slot disponível em uma sequência linear para encontrar uma posição aberta para inserir um novo elemento);
  - *Rehash* quadrático (procura o próximo slot usando uma função quadrática, por exemplo: $p + 1^2$, $p + 2^2$, $p + 3^2$, ...);
  - *Rehash* duplo (usa uma segunda função de *hash* para sondar e encontrar o próximo slot disponível na tabela hash).

Rehash linear

k = João

h(k) = 10

0	
1	
...	...
10	Maria
11	Carlos
12	

primeira tentativa: *bucket* ocupado

Rehash linear

k = João

h(k) = 10

0	
1	
...	...
10	Maria
11	Carlos
12	

primeira tentativa: *bucket* ocupado

segunda tentativa: *bucket* ocupado

Rehash linear

k = João

h(k) = 10

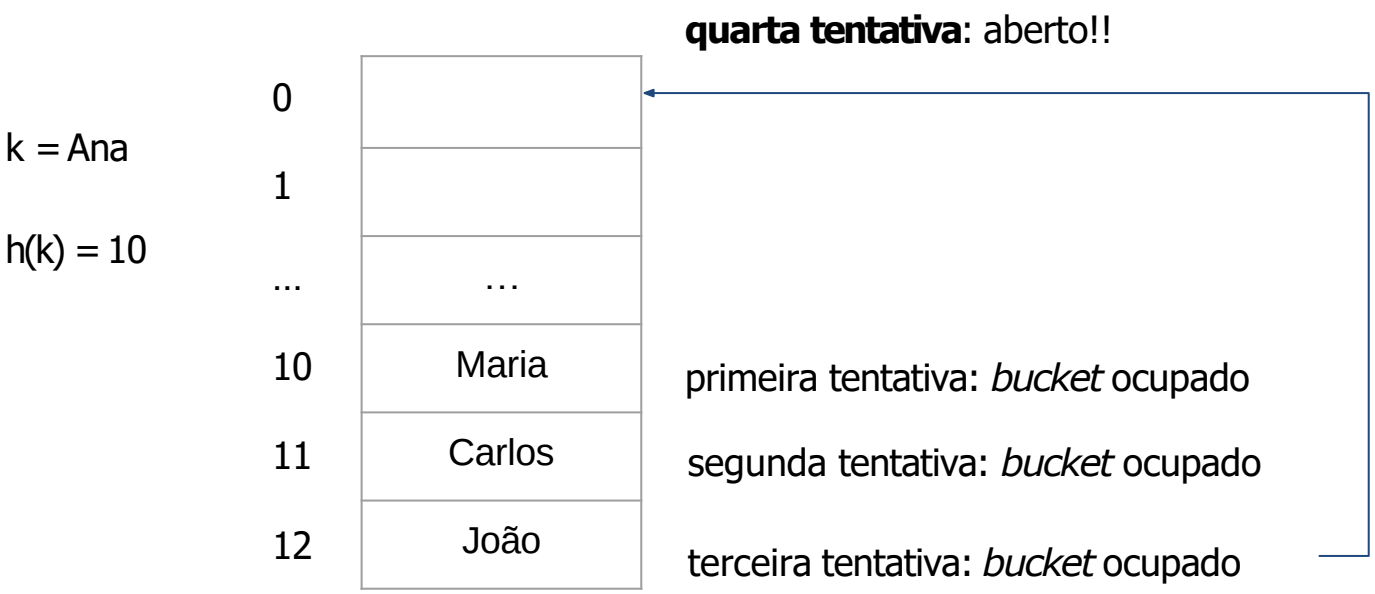
0	
1	
...	...
10	Maria
11	Carlos
12	João

primeira tentativa: *bucket* ocupado

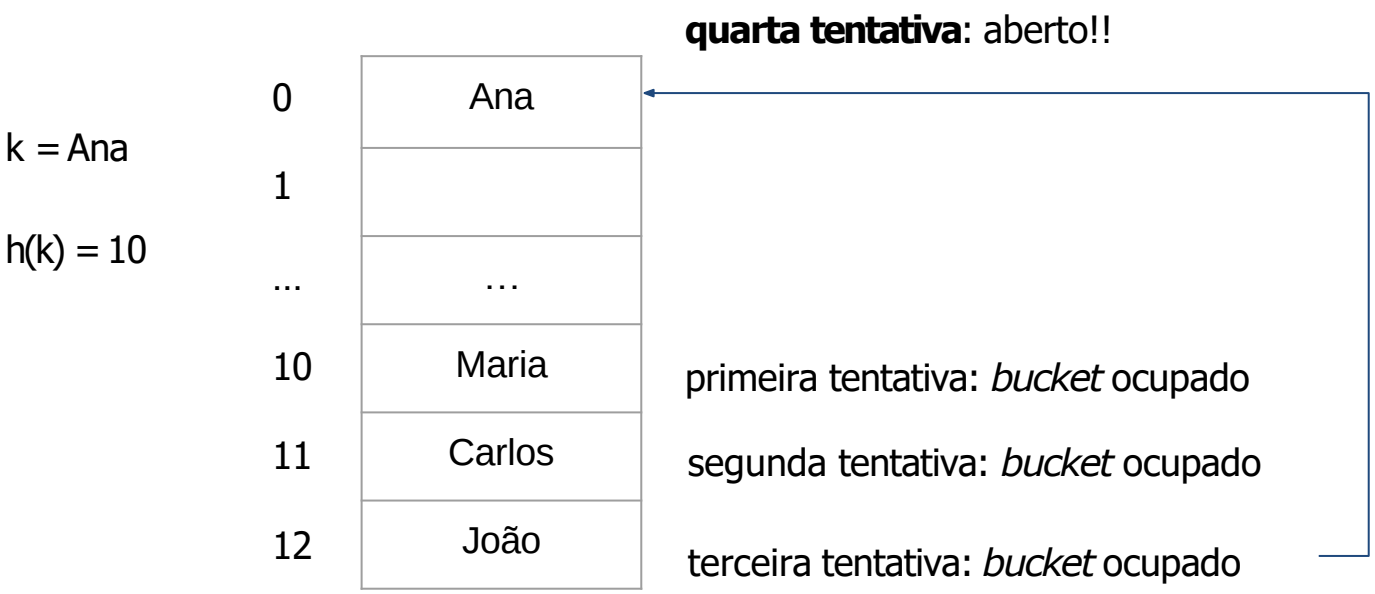
segunda tentativa: *bucket* ocupado

terceira tentativa: aberto!!

Rehash linear



Rehash linear

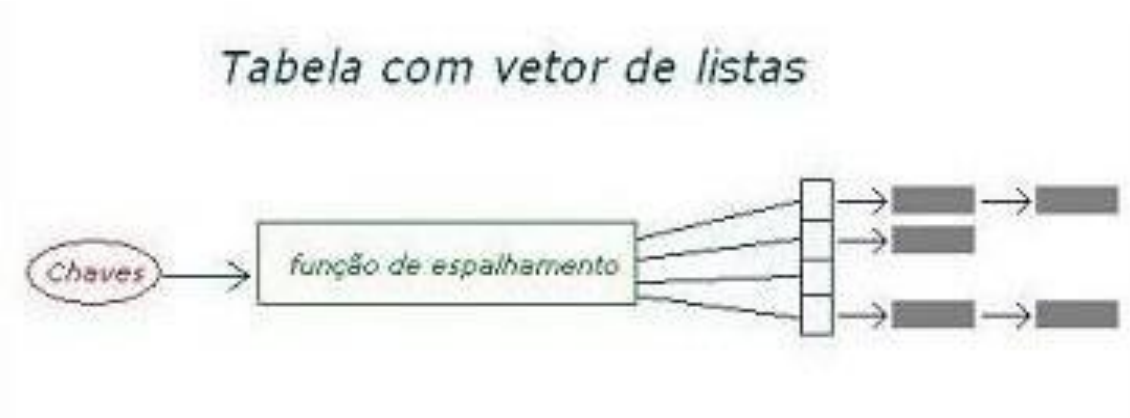


Hashing fechado

- ***Rehash* linear:**
  - **Vantagem:** simples de implementar;
  - **Desvantagens:**
    - Agrupamento de dados (causado por colisões);
    - Com estrutura cheia, a busca fica lenta;
    - Dificulta inserções e remoções.
- ***Rehash* duplo:**
  - **Vantagem:** evita agrupamento de dados, em geral;
  - **Desvantagens:**
    - Difícil achar funções *hash* que, ao mesmo tempo, satisfaçam os critérios de cobrir o máximo de índices da tabela e evitem agrupamento de dados;
    - Operações de buscas, inserções e remoções são mais difíceis.

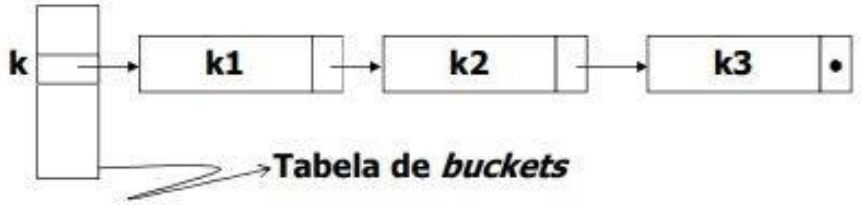
Hashing aberto

- A informação é armazenada em estruturas encadeadas fora da tabela *hash*;
- A tabela de buckets, contém apenas ponteiros para uma lista de elementos;
- Quando há colisão, o elemento é inserido no bucket como um novo nó da lista;
- Busca deve percorrer a lista.

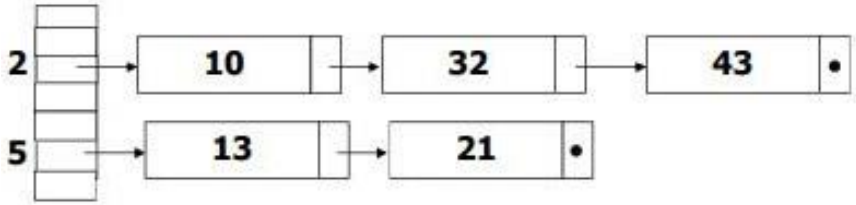


Hashing aberto

- Ideia geral:



- Se as listas estiverem ordenadas, reduz-se o tempo de busca.



Hashing aberto

- **Vantagens**:
  - A tabela pode receber mais itens mesmo quando um *bucket* já foi ocupado;
  - Permite percorrer a tabela por ordem de valor *hash.*

- **Desvantagens:**
  - Espaço extra para as listas;
  - Listas longas implicam em muito tempo gasto na busca;
  - Se as listas estiverem ordenadas, reduz-se o tempo de busca:
    - Mas há custo extra com a ordenação.

Tabelas *hash* frequentemente se beneficiam do conhecimento sobre o domínio (características, premissas, regras, inferências, ...) dos dados que serão *hashados*.

Busca sequencial, binária, digital, ABBs, AVLs, hashing... o que usar?

- Em geral, critérios para se eleger um (ou mais) método(s) variam:
  - Eficiência da busca;
  - Eficiência de outras operações:
    - Inserção e remoção;
    - Listagem e ordenação de elementos;
    - Outras?
  - Frequência das operações realizadas;
  - Dificuldade de implementação;
  - Consumo de memória (interna);
  - Outros?

Agenda

- Contextualização
- *Hash tables*
- Tratamento de colisões
- *Hands-on*

Atividade Individual

1. Implemente uma *hash table* utilizando **dicionários** em Python:
   a. versão onde cada chave do dicińario refere-se a um vetor simples:

      htable = { key1: [ ], key2: [ ] }

   b. versão com lista encadeadas.