

# Hash em arquivos



Prof. André Backes | @progdescomplicada

# Problema

- Princípio de funcionamento dos métodos de busca
  - Procurar a informação desejada com base na comparação de suas chaves
- Problema
  - Algoritmos eficientes necessitam que os elementos estejam armazenados de forma ordenada
  - Custo ordenação melhor caso é  **$O(N \log N)$**
  - Custo da busca melhor caso é  **$O(\log N)$**

# Problema

- Custo da comparação de chaves é alto
- O que seria uma operação de **busca ideal**?
  - Seria aquela que permitisse o acesso direto ao elemento procurado, sem nenhuma etapa de comparação de chaves
  - Nesse caso, teríamos um custo  **$O(1)$** 
    - Tempo sempre constante de acesso

# Problema

- Uma saída é usar **arrays**
  - São estruturas que utilizam índices para armazenar informações
  - Permite acessar uma determinada posição com custo  **$O(1)$**
- Problema
  - Arrays não possuem nenhum mecanismo que permita calcular a posição onde uma informação está armazenada
  - A operação de busca não é  **$O(1)$**

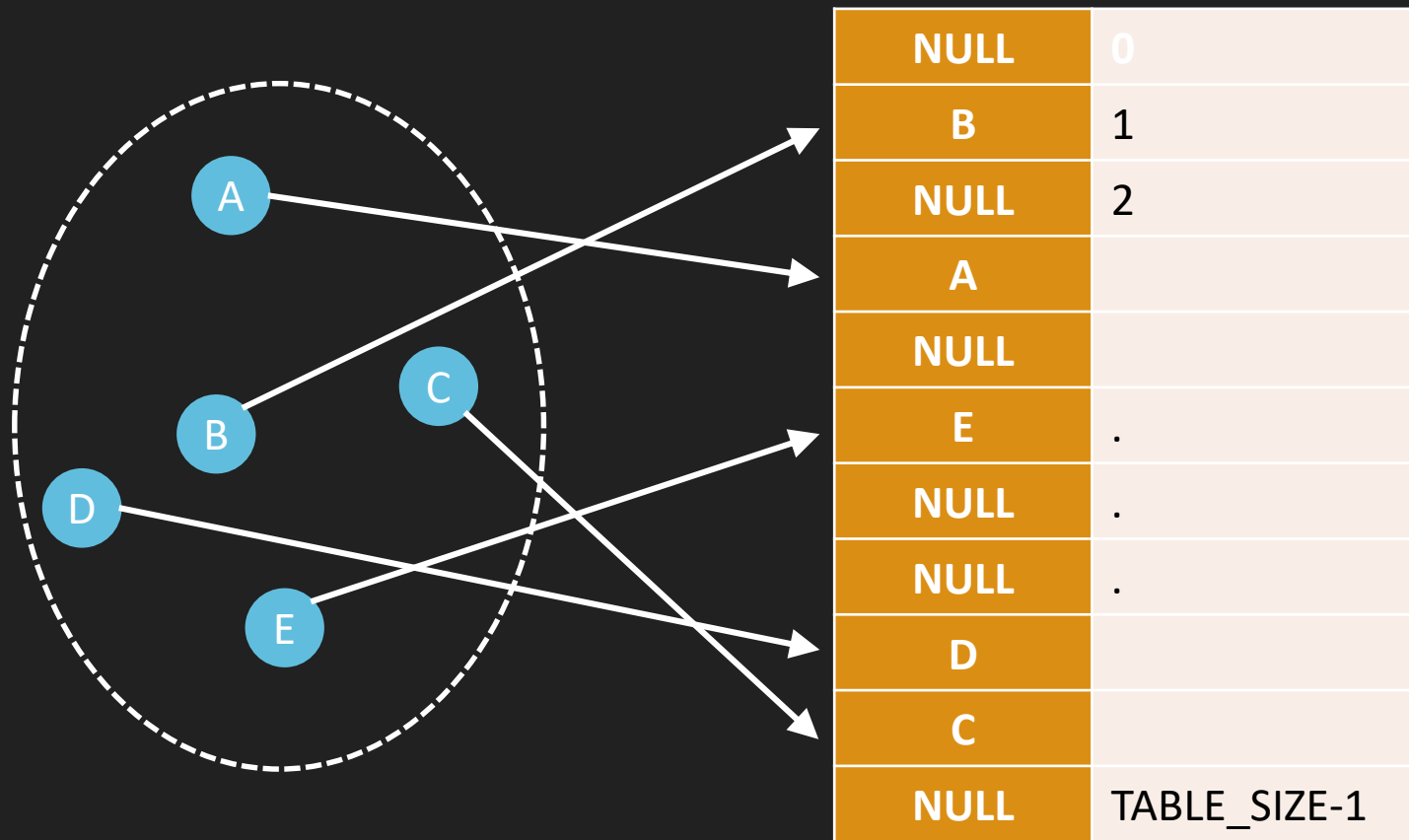
# Problema

- Precisamos do tempo de acesso do array juntamente com a capacidade de busca um elemento em tempo constante
- Solução: usar uma **tabela hash**

# Tabela Hash

- Também conhecidas como tabelas de indexação ou de espalhamento
  - É uma generalização da ideia de array
- Ideia central
  - Utilizar uma função, chamada de **função de hashing**, para espalhar os elementos que queremos armazenar na tabela
  - Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do array que define a tabela

# Tabela Hash | Exemplo



# Tabela Hash

- Por que espalhar os elementos melhora a busca?
  - A tabela permite a associar valores a chaves
    - **chave**: parte da informação que compõe o elemento a ser inserido ou buscado na tabela
    - **valor**: é a posição (índice) onde o elemento se encontra no array que define a tabela
  - Assim, a partir de uma **chave** podemos acessar de forma rápida uma **posição** do array
    - Na média, essa operação tem custo  **$O(1)$**



# Função de Hashing

- Inserção e busca: é necessário calcular a posição dos dados dentro da tabela.
- A função de hashing calcula a posição a partir de uma chave escolhida a partir dos dados manipulados

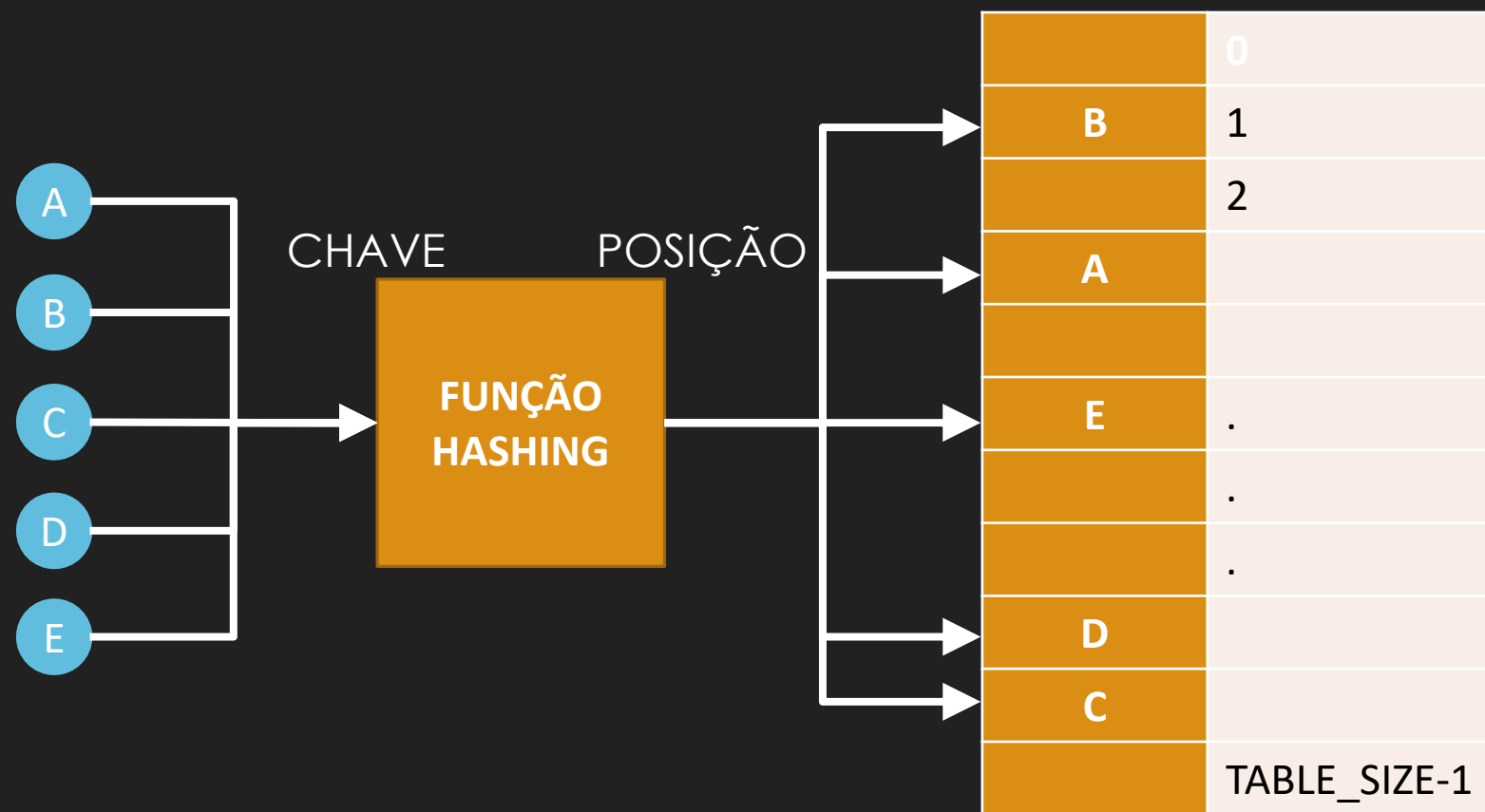


# Função de Hashing

- É extremamente importante para o bom desempenho da tabela
- Ela é responsável por distribuir as informações de forma equilibrada pela tabela hash



# Função de Hashing | Exemplo



# Tabela Hash

- Vantagens
  - Alta eficiência na operação de busca
    - Caso médio é  $O(1)$  enquanto o da busca linear é  $O(N)$  e a da busca binária é  $O(\log_2 N)$
  - Tempo de busca é praticamente independente do número de chaves armazenadas na tabela
  - Implementação simples

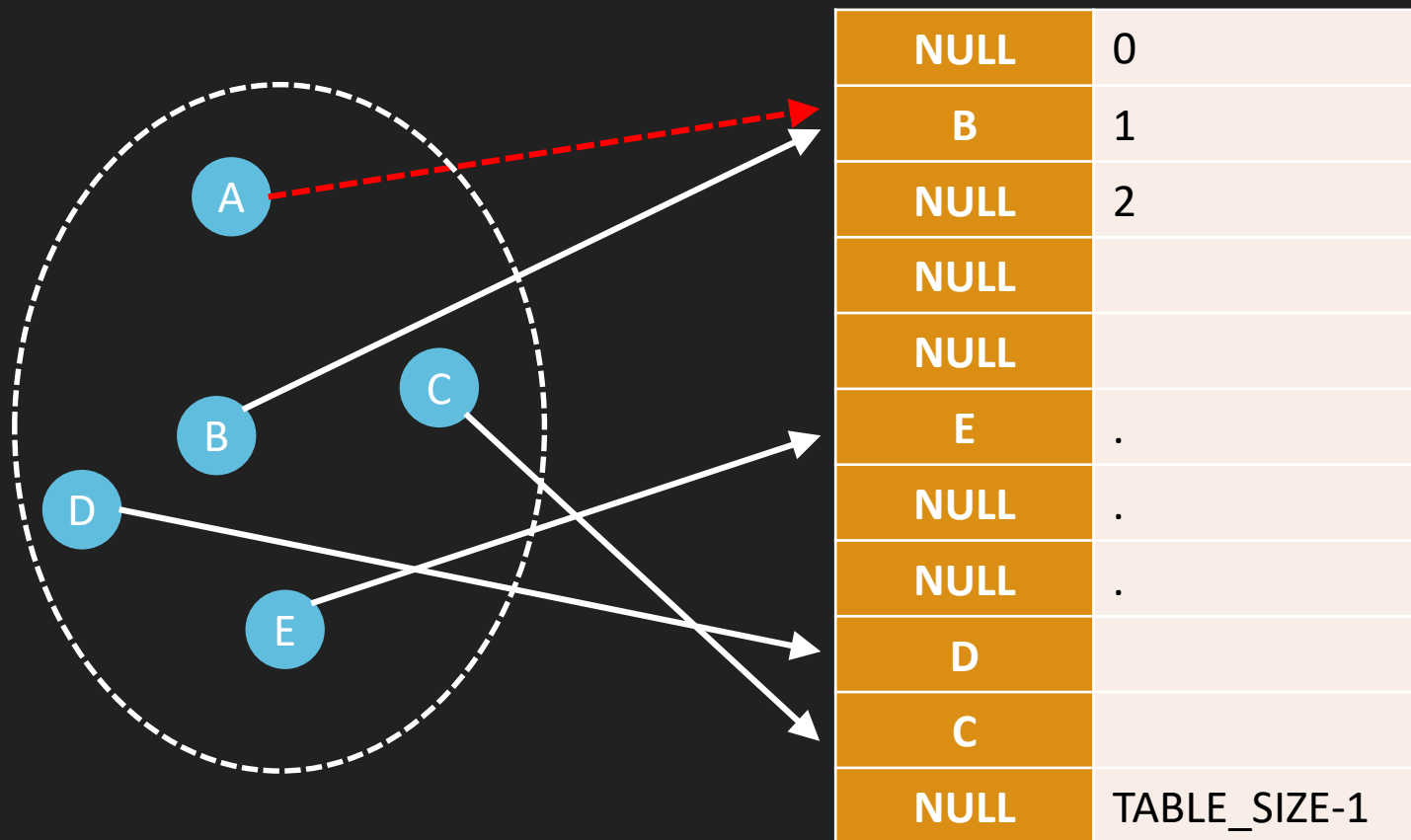
# Tabela Hash

- Infelizmente, esse tipo de implementação também tem suas desvantagens
  - Alto custo para recuperar os elementos da tabela ordenados pela chave
    - Nesse caso, é preciso ordenar a tabela
  - O pior caso é  $O(N)$ , sendo  $N$  o tamanho da tabela
    - Alto número de **colisões**

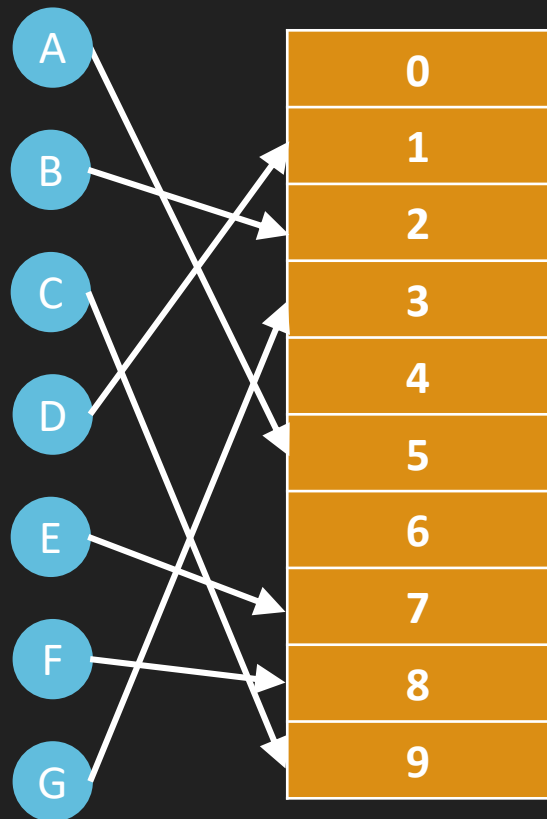
# Tabela Hash

- O que é uma **colisão**?
  - Uma colisão ocorre quando duas (ou mais) chaves diferentes tentam ocupar a mesma posição na tabela hash
  - A colisão de chaves não é algo exatamente ruim, é apenas algo indesejável pois diminui o desempenho do sistema

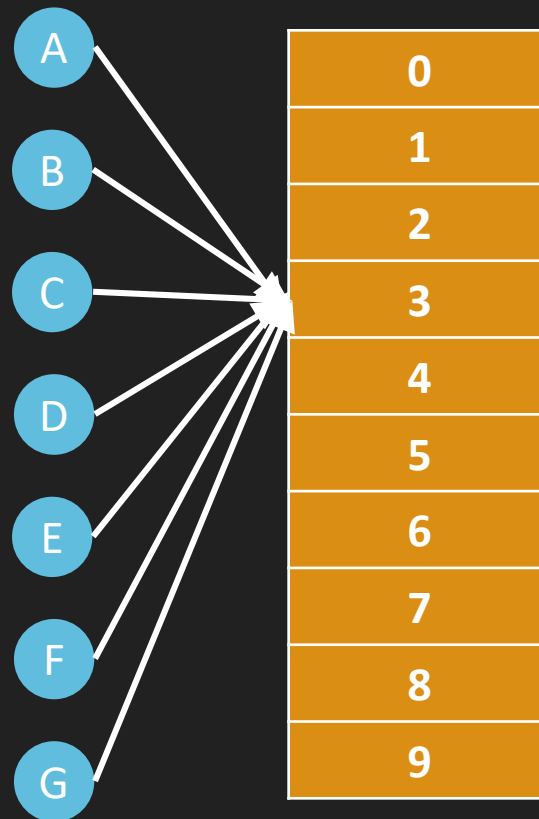
# Tabela Hash | Colisão



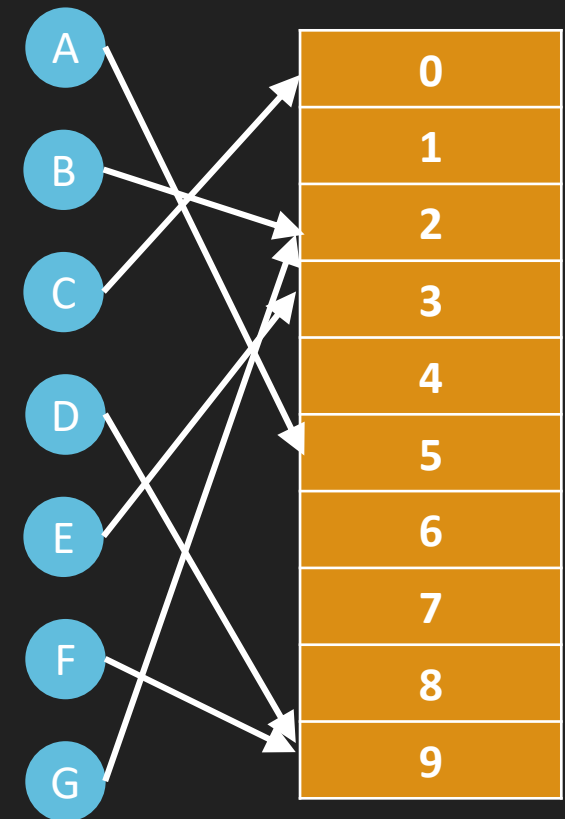
# Tabela Hash | Colisão



Melhor caso



Pior caso



Caso aceitável



# Tabela Hash

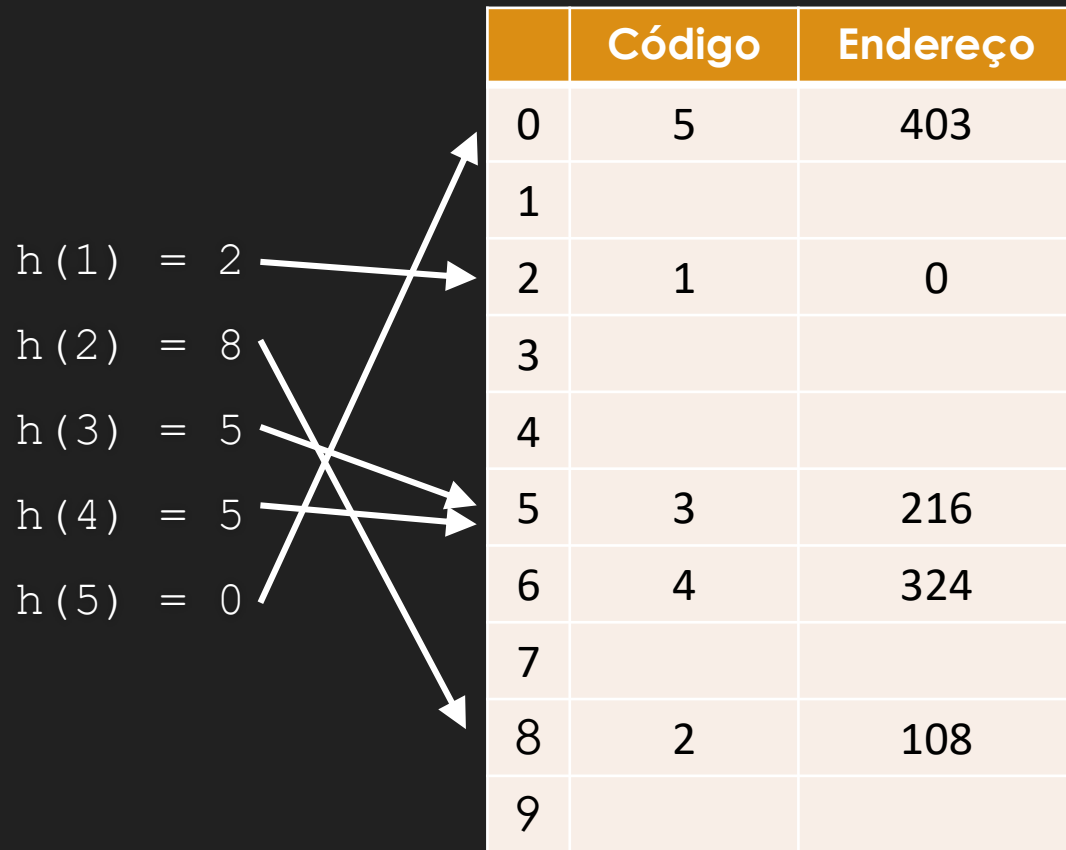
- Mundo ideal: hashing perfeito
  - Função de hashing irá sempre fornecer posições diferentes para cada uma das chaves inseridas
- Mundo real: independente da função de hashing utilizada, a mesma **posição** vai ser calculada para duas **chaves** diferentes
  - **Colisão!**
- A criação de uma tabela hash consiste de duas coisas
  - uma **função de hashing**
  - uma **abordagem para o tratamento de colisões**
    - Endereçamento aberto
    - Encadeamento separado

# Hash em Arquivos

# Hashing Externo

- O conceito de hashing pode ser estendido para memória secundária
  - Armazenamento e recuperação em disco
  - Funcionamento é parecido com a Tabela Hash em memória principal
- Em arquivos
  - Resultado da função hash,  $h(chave)$ , direciona ao RRN
  - As operações de inserção, remoção e busca seguem o mesmo padrão

# Hashing Externo | Endereçamento aberto



	Código	Endereço
0	5	403
1		
2	1	0
3		
4		
5	3	216
6	4	324
7		
8	2	108
9		

$h(1) = 2$

$h(2) = 8$

$h(3) = 5$

$h(4) = 5$

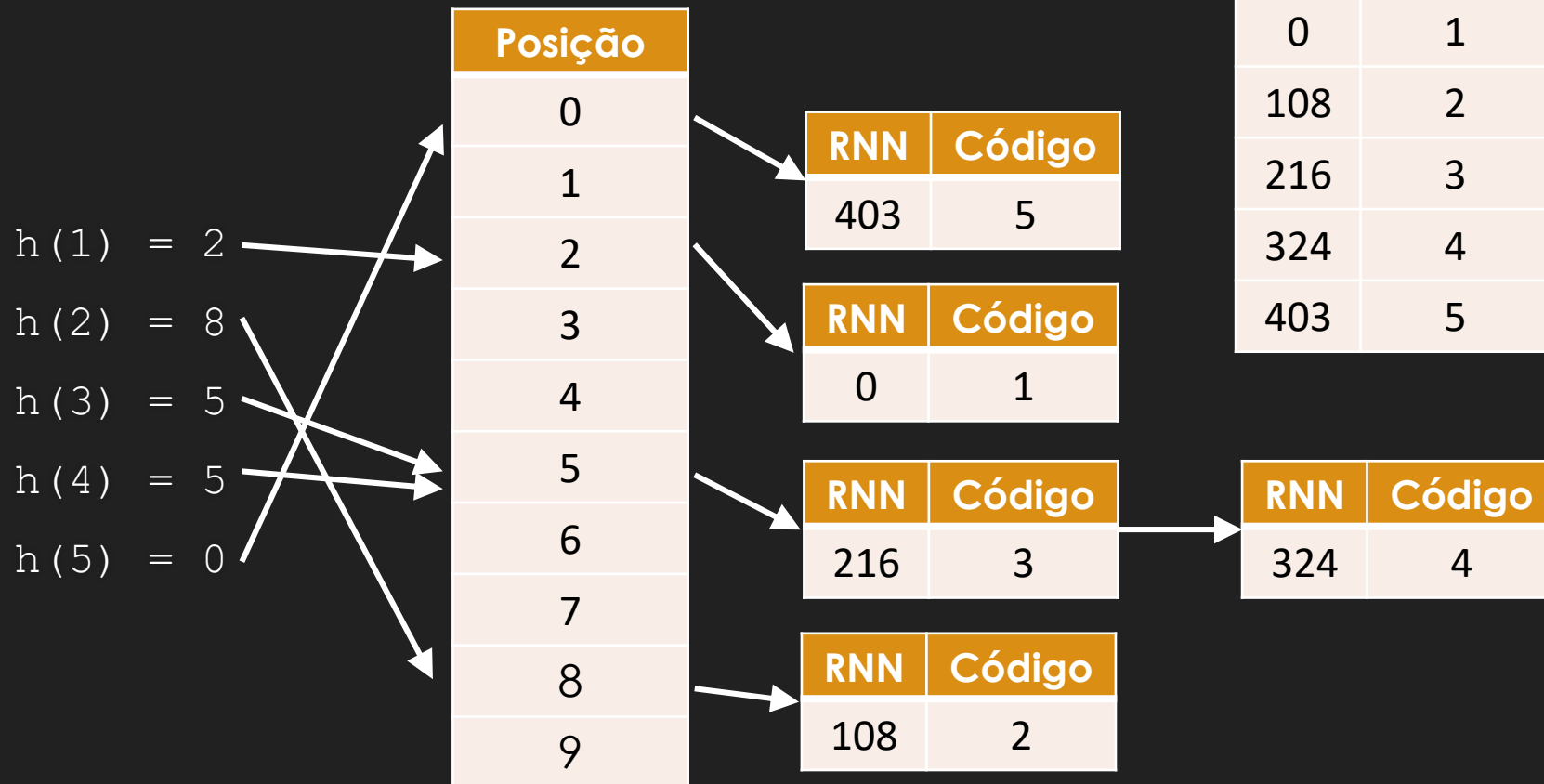
$h(5) = 0$

RNN	Código	Nome	Curso	Ano
0	1	André	BSI	2020
108	2	Carlos	Elétrica	2021
216	3	Eduardo	Materiais	2021
324	4	Pedro	Medicina	2022
403	5	Jéssica	Enfermagem	2021

# Hashing Externo

- Tratamento de colisões: usar endereçamento aberto aumenta a chance de novas colisões e compromete o desempenho em arquivos
- O ideal é usar encadeamento separado
  - Não procura por posições vagas dentro da tabela
  - Armazena dentro de cada posição da tabela o início de uma lista encadeada de registros
  - É dentro dessa lista que serão armazenadas as colisões (elementos com chaves iguais) para aquela posição da tabela

# Hashing Externo | Encadeamento separado



RNN	Código	Nome	Curso	Ano
0	1	André	BSI	2020
108	2	Carlos	Elétrica	2021
216	3	Eduardo	Materiais	2021
324	4	Pedro	Medicina	2022
403	5	Jéssica	Enfermagem	2021

# Hashing Externo | Encadeamento separado

- Essa abordagem ignora as particularidades do armazenamento na memória secundária
  - A gravação e a leitura são feitas em blocos
- Se for usado encadeamento, pode ocorrer de cada elemento de uma lista estar em um bloco diferente
  - Necessidade de mais acessos ao dispositivo externo
  - Isso impacta negativamente no desempenho

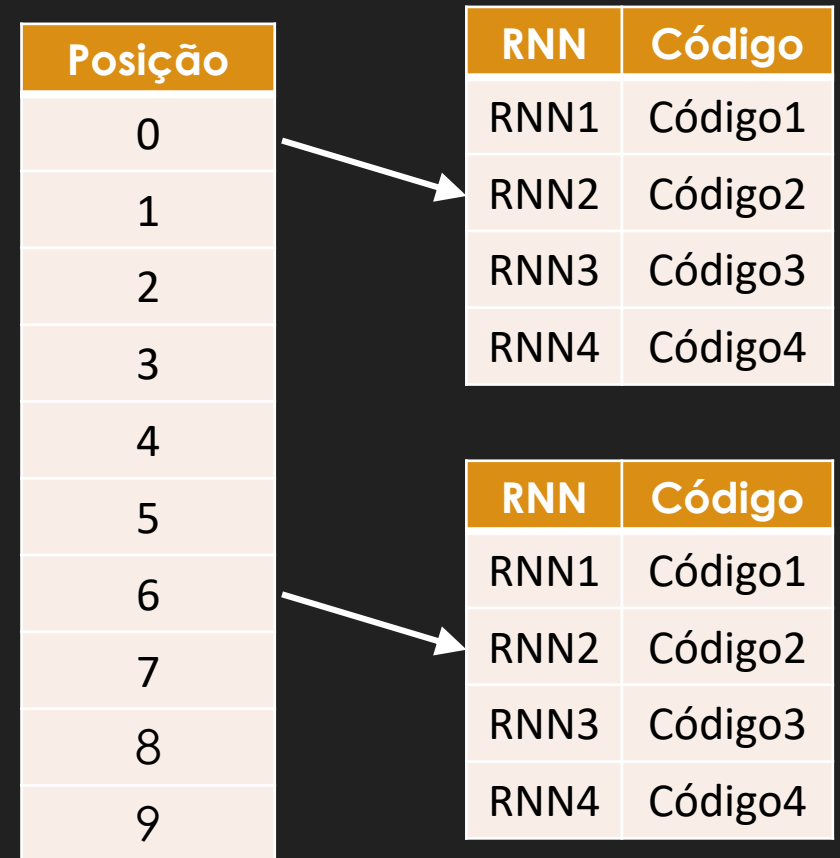
# Hashing Externo

- Uma alternativa é fazer uso de baldes (*buckets*)
  - Um balde corresponde a um grupo de registros. Se o tamanho de um balde corresponde à dimensão de um bloco, a função de espalhamento determina em qual bloco o registro será inserido
  - Melhora o acesso a blocos
  - Tende a minimizar o número de acessos externos



# Hashing Externo | Buckets

- Cada posição da tabela pode armazenar mais de uma entrada (registro)
  - Colisões são colocadas dentro do bucket
  - Agora, um acesso ao disco permite retornar os dados de vários registros e não apenas um como na lista encadeada



# Hashing Externo | Organização

- Organização de índices hashing
- Único arquivo
  - Os dados e o índice hashing ficam no mesmo arquivo
- Dois arquivos
  - Os dados ficam em um arquivo e o índice hashing das chaves fica em outro

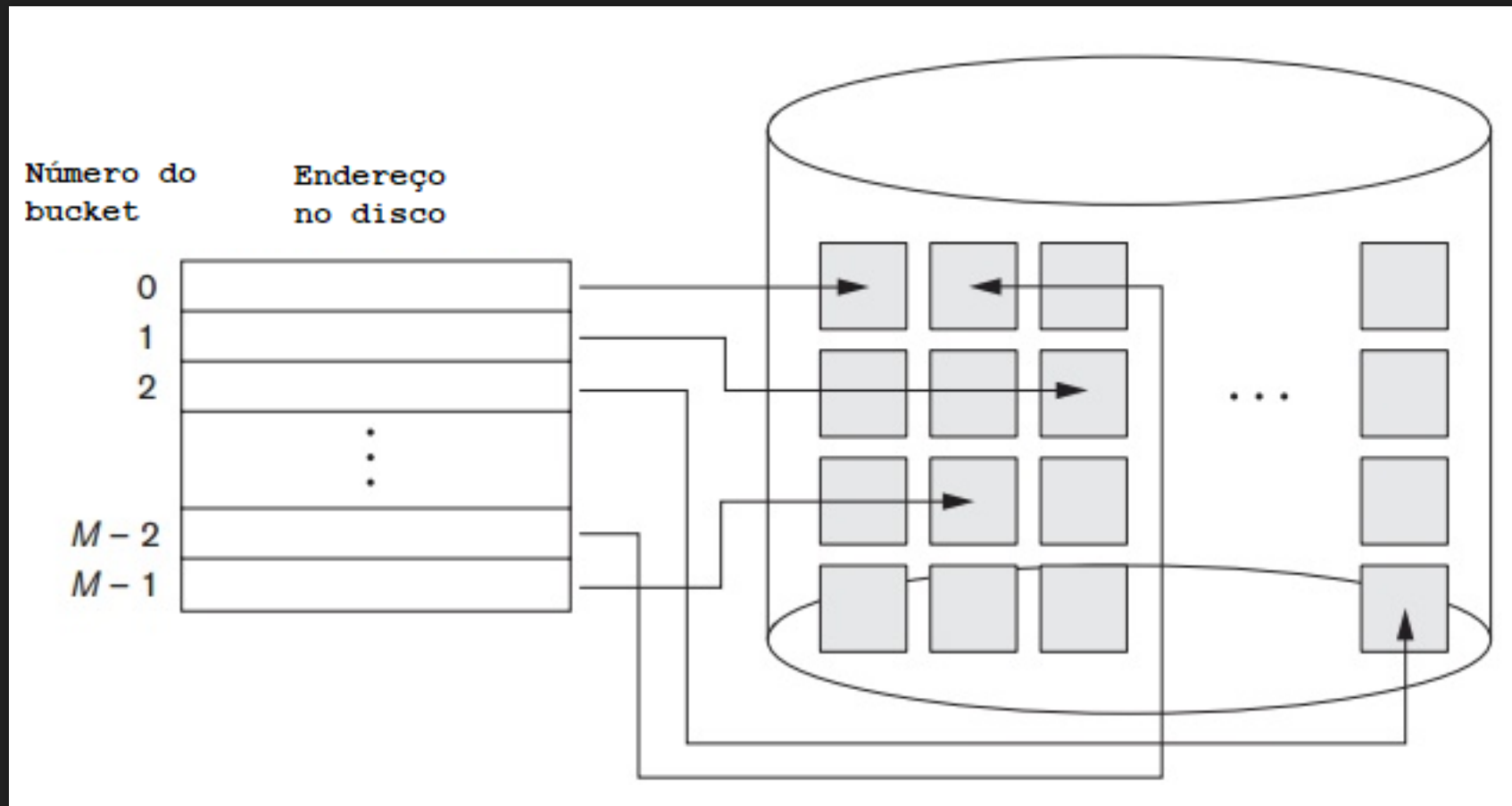
# Hashing Externo | Tipos

- Hashing estático
  - Garante acesso  $O(1)$ , para arquivos estáticos
  - Espaço de endereçamento não muda (número de *buckets* é fixo)
- Hashing dinâmico
  - O tamanho do espaço de endereçamento (número de *buckets*) pode aumentar
  - Extensão do hashing estático para tratar arquivos dinâmicos, ou seja, arquivos que sofrem muitas inserções e remoções
  - Estratégias: extensível e linear (mais comuns)

# Hashing Estático

- A função hash mapeia as chave de busca em um conjunto fixo de endereços de *bucket*
  - Retorna o número de um *bucket* ao invés de uma posição no array
  - Um *bucket* é uma unidade de armazenamento (normalmente é um bloco de disco)
  - Um *bucket* pode conter um ou mais registros do arquivo
  - Podem haver  $m$  registros por *bucket*
- Uma tabela associa o número do *bucket* ao endereço do primeiro (ou único) registro
- Colisão ocorre quando inserimos o  $(m+1)$ -ésimo registro

# Hashing Estático



# Hashing Estático | Exemplo

- Organização de arquivo de hash do arquivo **conta**, usando **nome-agência** como chave
  - Existem 10 buckets
  - A representação binária do  $i$ -ésimo caractere é considerada como o inteiro  $i$

# Hashing Estático | Exemplo

- A função de hash retorna a soma das representações binárias dos caracteres módulo 10

- $h(\text{Perryridge}) = 5$

- $h(\text{Round Hill}) = 3$

- $h(\text{Brighton}) = 3$

bucket 0			
bucket 1			
bucket 2			
bucket 3	A-217	Brighton	750
	A-305	Round Hill	350
bucket 4	A-222	Redwood	700
bucket 5	A-102	Perryridge	400
	A-201	Perryridge	900
	A-218	Perryridge	700
bucket 6			
bucket 7	A-215	Mianus	700
bucket 8	A-101	Downtown	500
	A-110	Downtown	600
bucket 9			

# Hashing Estático | Funções de Hash

- A pior função de hash mapeia todos os valores de chave para o mesmo *bucket*
  - Isso torna o tempo de acesso proporcional ao número de valores de chave de busca no arquivo
- Uma função de hash ideal é uniforme
  - Cada *bucket* recebe o mesmo número de valores de chave de busca do conjunto de valores possíveis
- A função de hash ideal é aleatória
  - Cada *bucket* terá o mesmo número de registros atribuídos a ele, independente da distribuição real dos valores de chave de busca no arquivo



# Hashing Estático | Funções de Hash

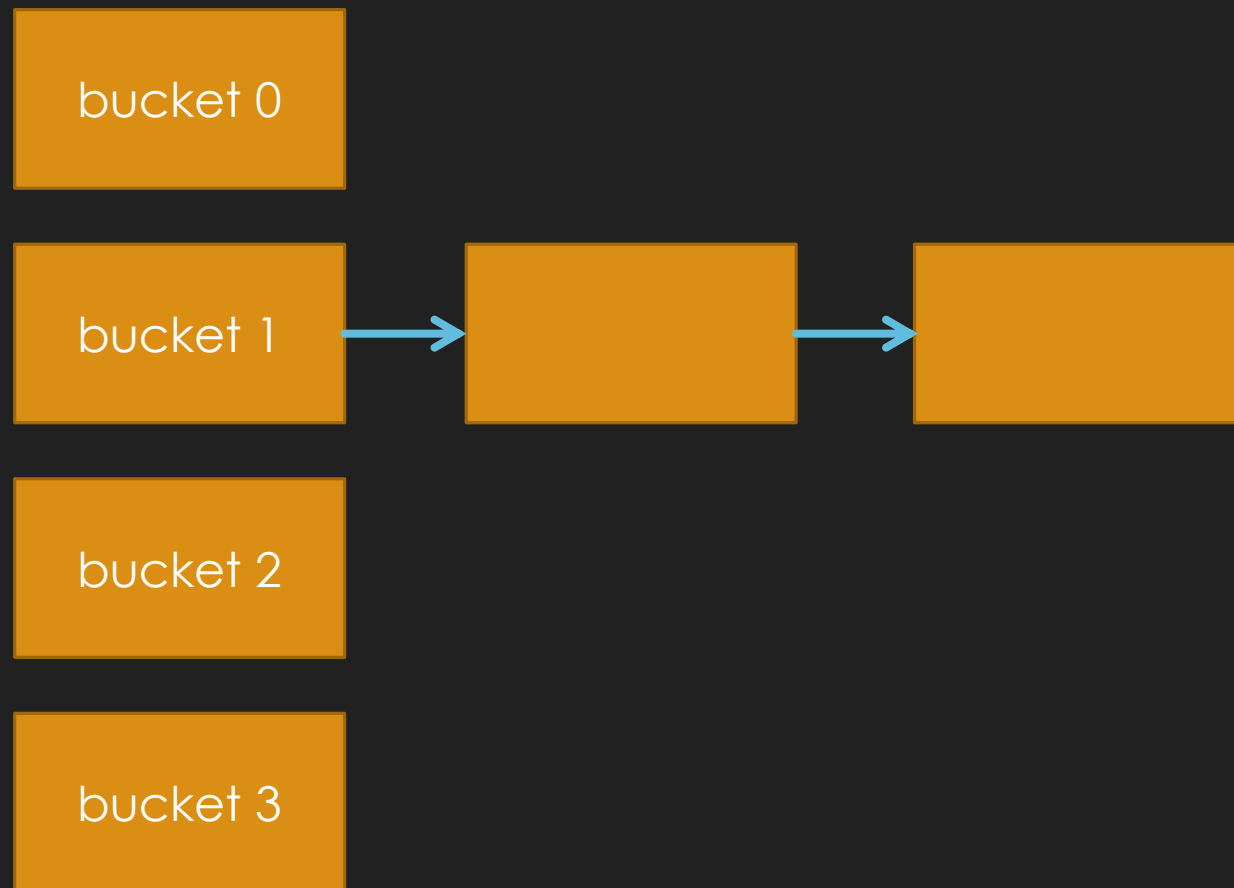
- As funções de hash típicas realizam seu cálculo sobre a representação binária interna da chave de busca
  - Por exemplo, para uma chave de busca de *string*, as representações binárias de todos os caracteres na *string* poderiam ser somadas e a soma módulo número de *buckets* poderia ser retornada

# Hashing Estático | Estouro de *bucket*

- O estouro (*overflow*) de *bucket* pode ocorrer devido a
  - *Buckets* insuficientes
  - Distorção na distribuição de registros. Isso pode ocorrer por dois motivos:
    - Vários registros possuem o mesmo valor de chave de busca
    - A função de hash escolhida produz uma distribuição não uniforme das chaves
- Embora a probabilidade de estouro de *bucket* possa ser reduzida, ela não pode ser eliminada
  - Ele é tratado pelo uso de *buckets* de estouro

# Hashing Estático | Tratamento de Estouro

- Encadeamento de estouro
  - Os *buckets* de estouro para um *bucket* são encadeados em uma lista ligada
  - Esse esquema é chamado de hashing fechado
  - Outra alternativa: hashing aberto
    - Não usa buckets de estouro
    - Não é apropriada para aplicações de banco de dados



# Hashing Estático | Inserção

- Dada a chave, calcula o endereço do *bucket* com a função hashing, lê o *bucket*, verifica se há espaço nele
- Se houver espaço, insere na posição correta dentro do *bucket*, e grava o *bucket* atualizado no disco
- Se não houver espaço, busca / lê novo *bucket* na lista encadeada de *bucket*, até encontrar um com espaço e/ou o último da lista
  - Se encontrar na lista um *bucket* com espaço, insere e grava o *bucket* atualizado no disco
  - Caso não houver espaço em nenhum deles, cria um novo *bucket*, insere e grava o novo *bucket*, atualizando a lista encadeada

# Hashing Estático | Deficiências

- Os bancos de dados crescem com o tempo. Se o quantidade de *buckets* for muito pequena, o desempenho degradará devido a muitos estouros (*overflow*)
- Se o tamanho do arquivo for antecipado e o número de *buckets* for alocado de acordo com isso, uma quantidade de espaço significativa será desperdiçada inicialmente
- Se o banco de dados diminuir, novamente o espaço será desperdiçado

# Hashing Estático | Deficiências

- Podemos fazer uma reorganização periódica do arquivo com uma nova função de hash, mas isso é muito custoso
- Esses problemas podem ser evitados usando técnicas que permitem que o número de *buckets* seja modificado dinamicamente

# Manipulando arquivos dinâmicos

- A função hashing tem que se adaptar de alguma forma ao novo tamanho do arquivo
  - Se a função simplesmente muda, ela se torna inútil
  - Não é possível localizar os registros inseridos anteriormente

# Hashing Dinâmico

- Extensão do hashing estático para tratar arquivos dinâmicos
  - Arquivos que sofrem muitas inserções e remoções
- Permite um auto-ajuste do espaço de endereçamento do espalhamento
  - O tamanho do espaço de endereçamento (número de *buckets*) pode aumentar
- Duas abordagens possíveis
  - Hash extensível, número de *buckets*,  $B$ , cresce dobrando-o
  - Hashing linear, número de *buckets*,  $B$ , cresce em 1 unidade



# Hashing Extensível

- Consiste em organizar os dados em *buckets*, porém usando uma estrutura intermediária mantida em memória primária
  - Essa estrutura intermediária é chamada de **diretório**
  - Basicamente, é uma tabela contendo endereços de *buckets*

# Hashing Extensível

- O propósito do **diretório** é aumentar ou diminuir com o tempo, de acordo com a quantidade de registros existentes no arquivo dados
  - Ele varia de acordo com a quantidade de *buckets*
  - O controle da variação do tamanho do diretório é feita por sua **profundidade**

# Hashing Extensível

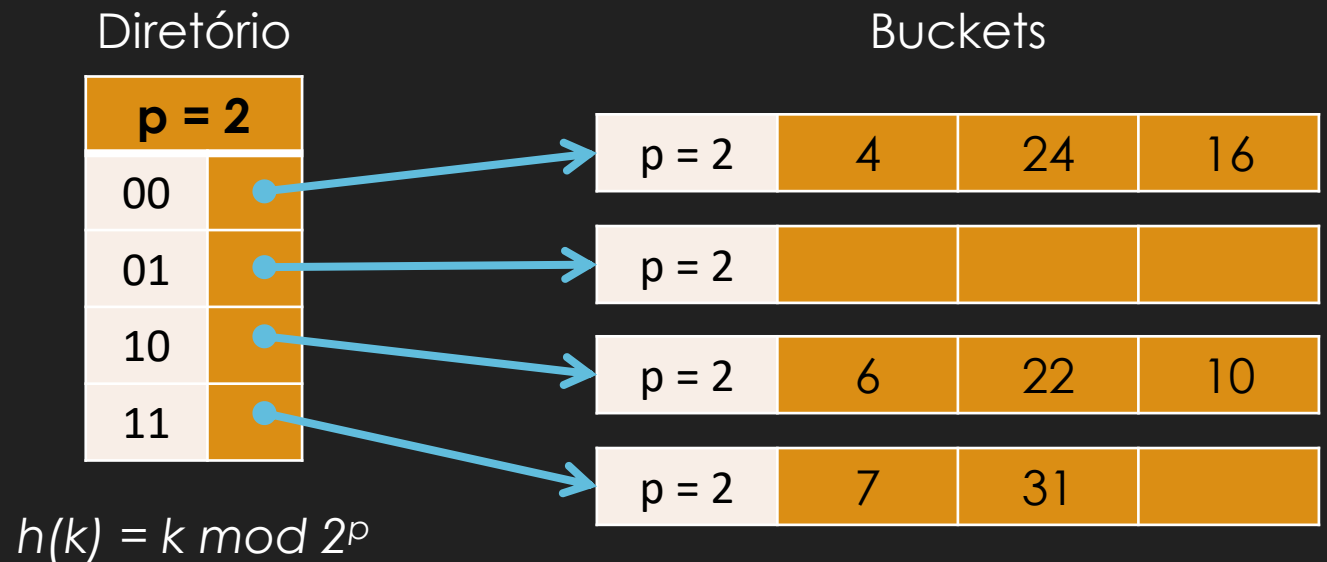
- Estratégia geral: usar uma única função hash, mas não todo o seu resultado de uma vez
  - A função hash computa uma sequência de **m** bits para uma chave **k**
  - Apenas os **p** primeiro bits (**p ≤ m**) do início da sequência são usados como endereço.
  - Se **p** é o número de bits usados, a tabela de diretórios terá **2<sup>p</sup>** entradas
  - A quantidade de bits usados é a **profundidade do diretório**
  - O tamanho da tabela sempre cresce como potência de 2

# Hashing Extensível

- Em resumo, usamos pedaços menores da chave para gerar um endereçamento quando o arquivo está pequeno, e partes cada vez maiores da chave para gerar endereçamento a um arquivo maior
  - Junto ao *bucket* também guardamos a informação da sua profundidade, i.e., uma indicação do número de bits da chave necessários para determinar quais registros ele contém
  - Inicialmente, a profundidade é a mesma para todos os *buckets*, e é igual a profundidade do **diretório**

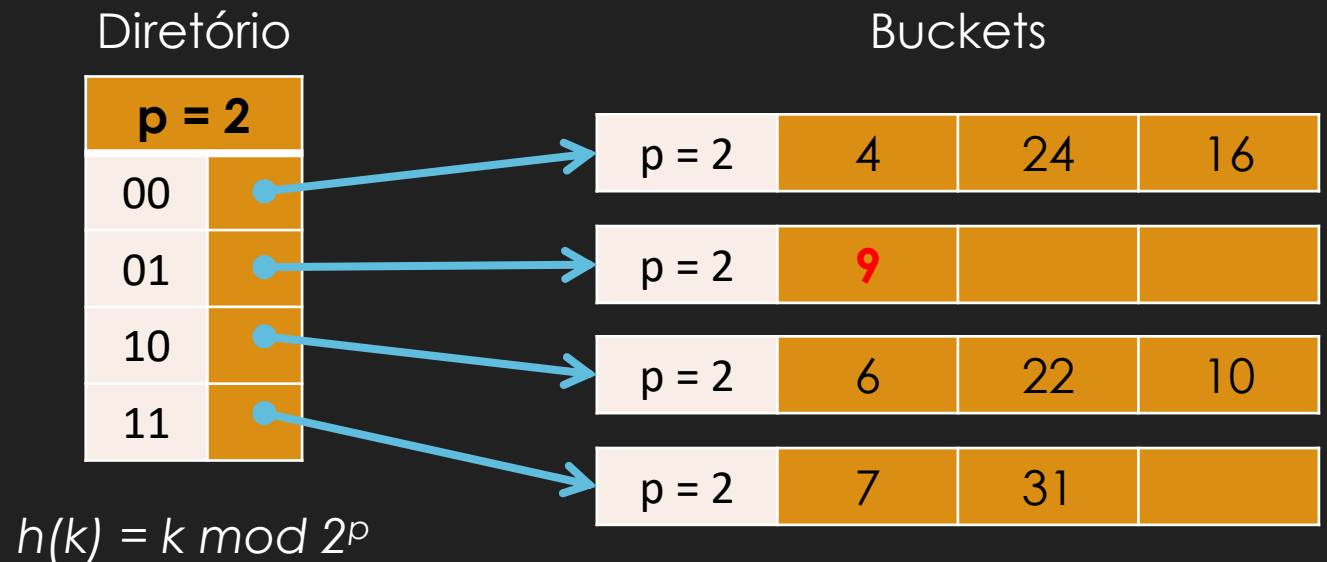
# Hashing Extensível | Exemplo

- Profundidade do diretório:  $p = 2$
- Isso dá um total de 4 entradas, cada uma apontando para um *bucket*



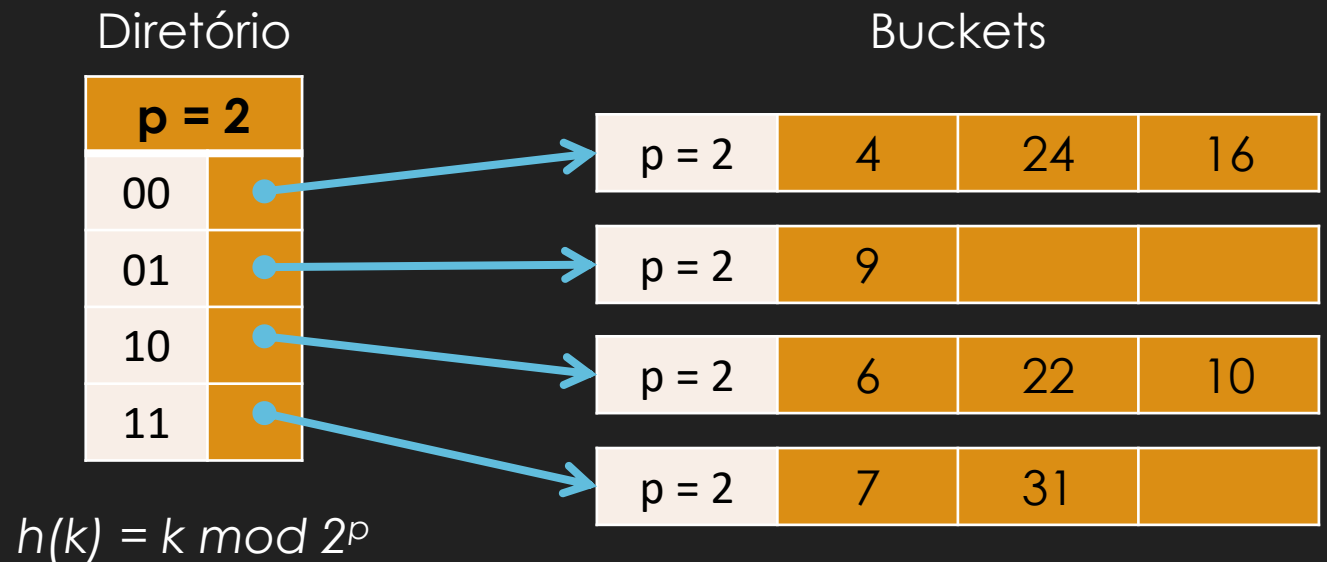
# Hashing Extensível | Exemplo

- Inserção da chave 9
  - Profundidade do diretório:  $p = 2$
  - $h(9) = 9 \bmod 2^2 = 1$
  - Tem espaço no *bucket*



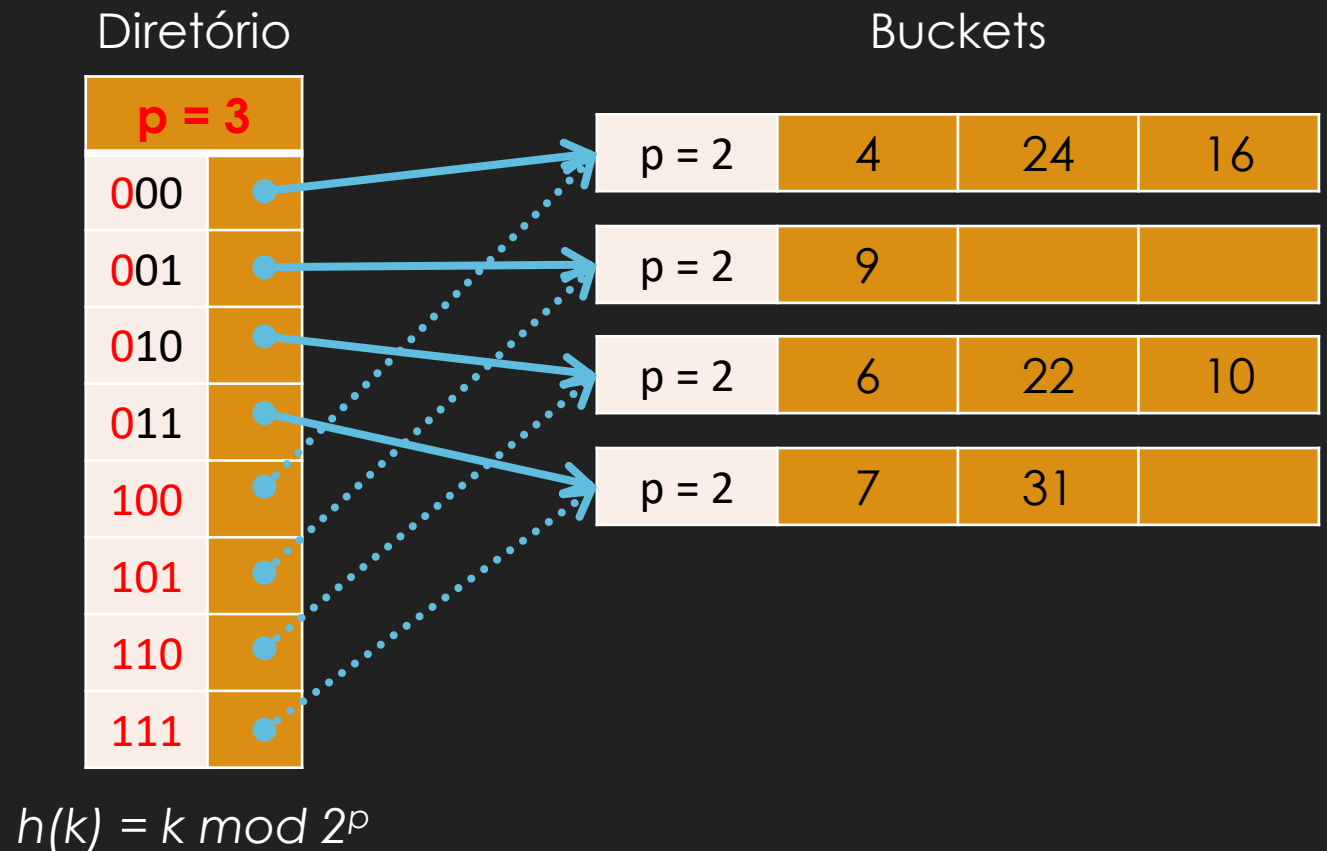
# Hashing Extensível | Exemplo

- Inserção da chave 20
  - Profundidade do diretório:  $p = 2$
  - $h(20) = 20 \bmod 2^2 = 0$
  - *Bucket* está cheio



# Hashing Extensível | Exemplo

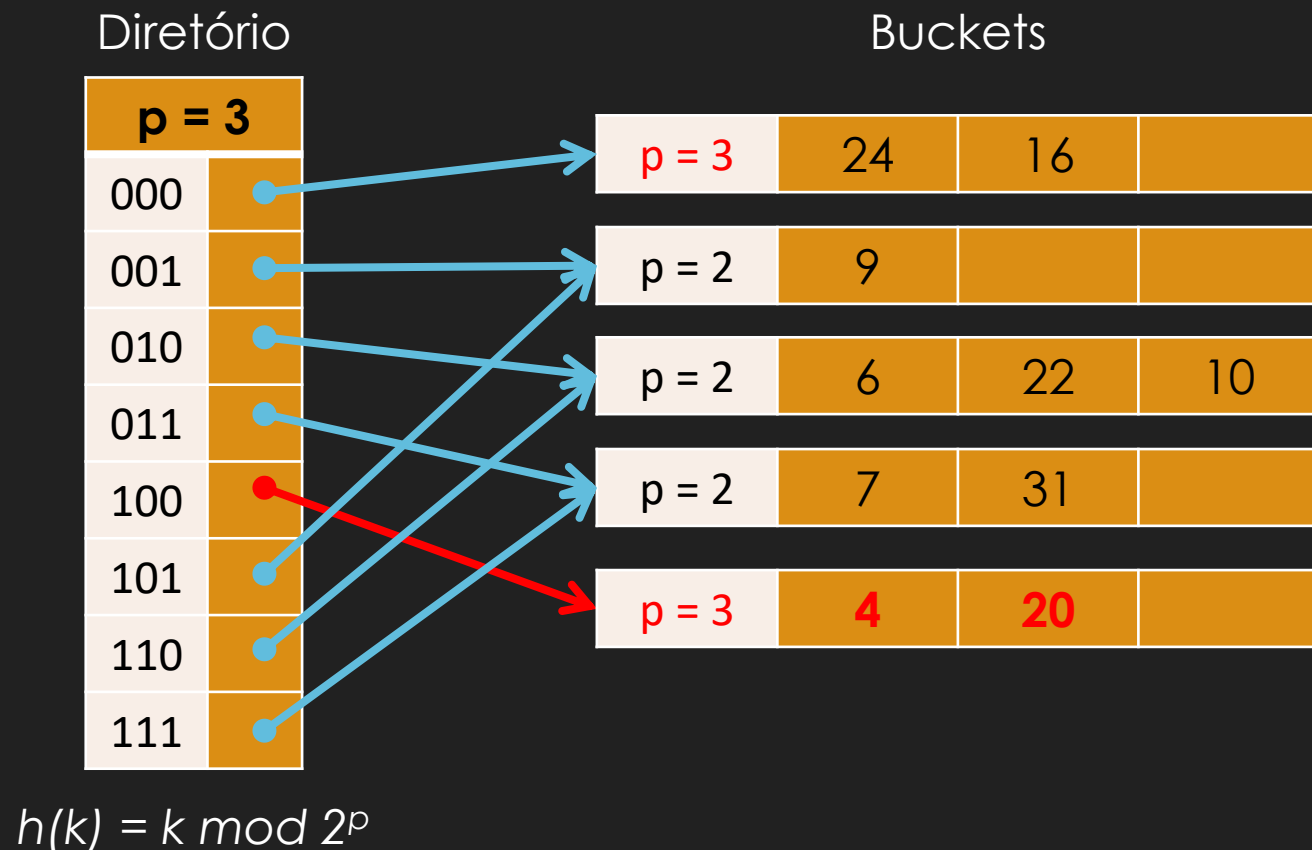
- Solução: dobrar tamanho do diretório
  - Isso é feito considerando mais um **bit** na chave
  - Novas entradas do diretório mantém os ponteiros para os dados originais





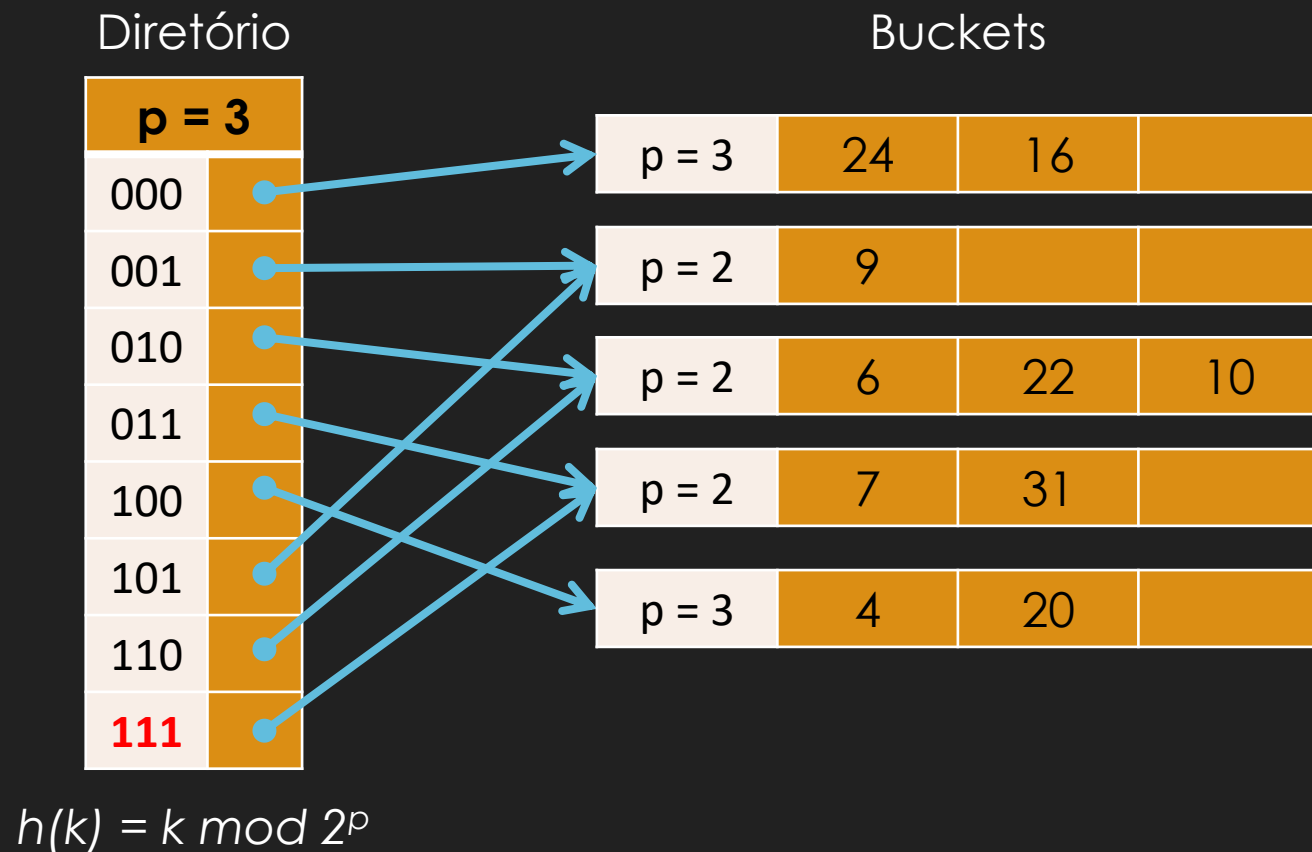
# Hashing Extensível | Exemplo

- Criar um novo *bucket*
  - Usar a nova entrada do diretório que aponta para o *bucket* cheio para apontar para o novo *bucket*
  - Redistribuir as chaves do *bucket* original. Algumas ficam no *bucket* original, outras vão para o novo *bucket*
  - Inserir a chave 20



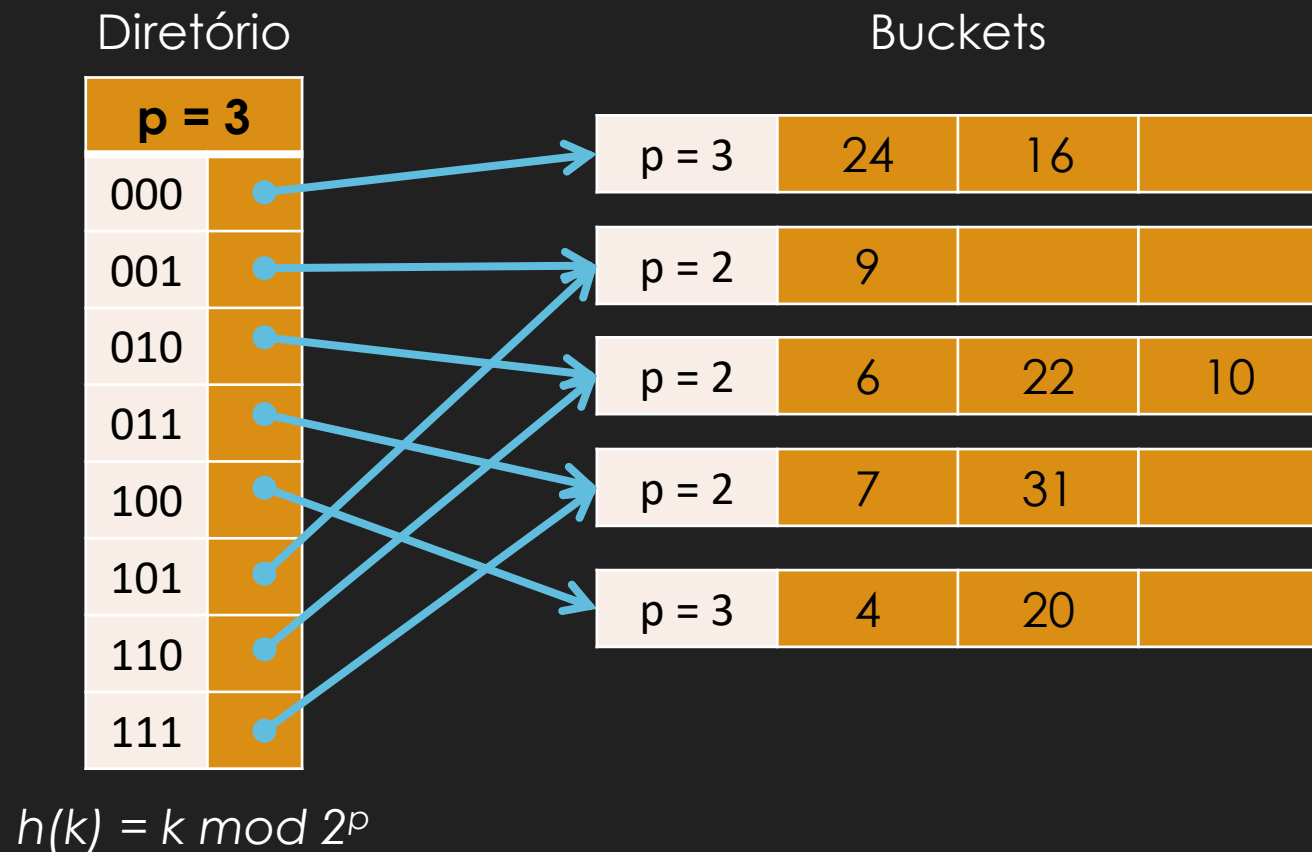
# Hashing Extensível | Exemplo

- O aumento no tamanho do diretório não afeta as outras chaves e *buckets*
- $h(9) = 9 \bmod 2^3 = 1$ 
  - Se manteve no lugar
- $h(10) = 10 \bmod 2^3 = 2$ 
  - Se manteve no lugar
- $h(31) = 31 \bmod 2^3 = 7$ 
  - Deveria ser na posição 3
  - Porém, o *bucket* 7 também aponta para o *bucket* 3



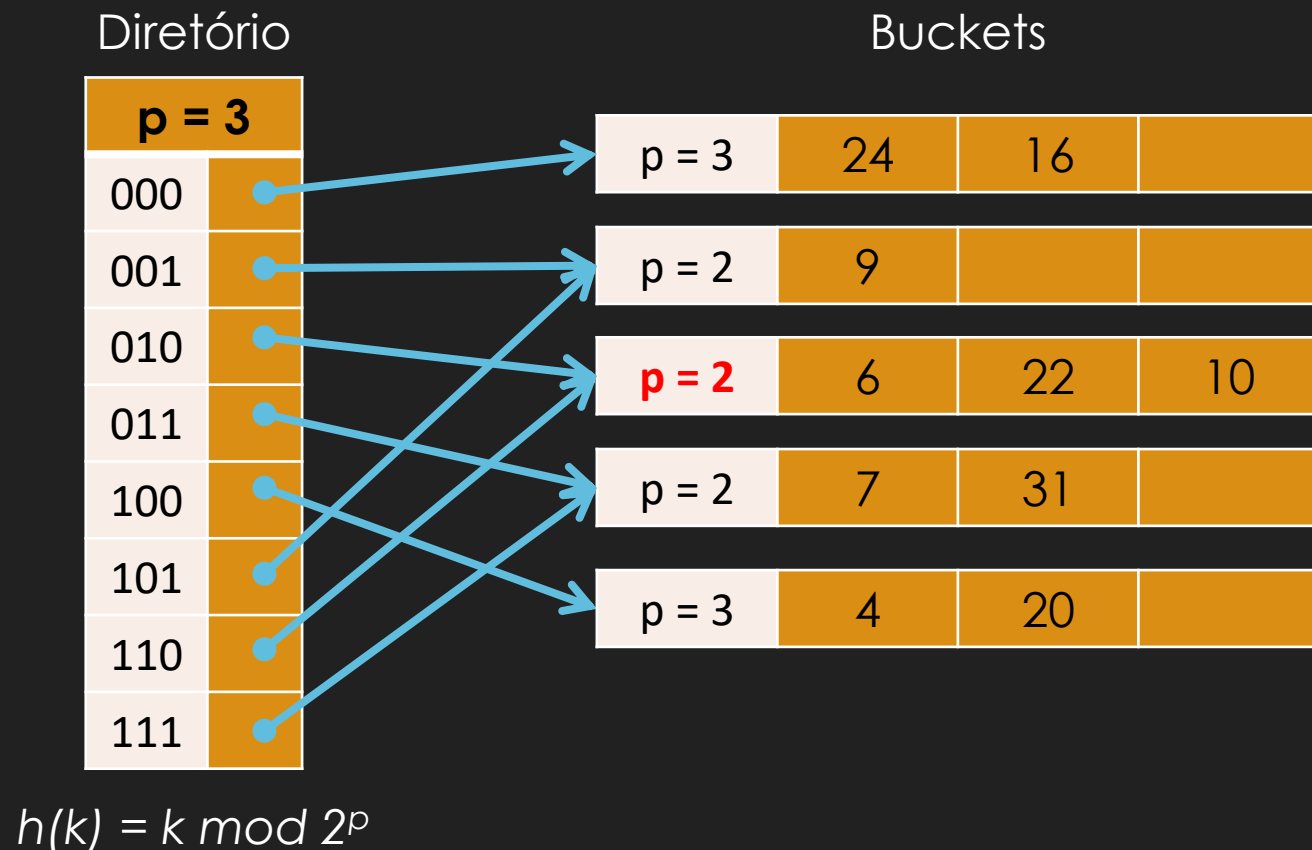
# Hashing Extensível | Exemplo

- Inserção da chave 26
  - Profundidade do diretório:  $p = 3$
  - $h(26) = 26 \bmod 2^3 = 2$
  - *Bucket* está cheio



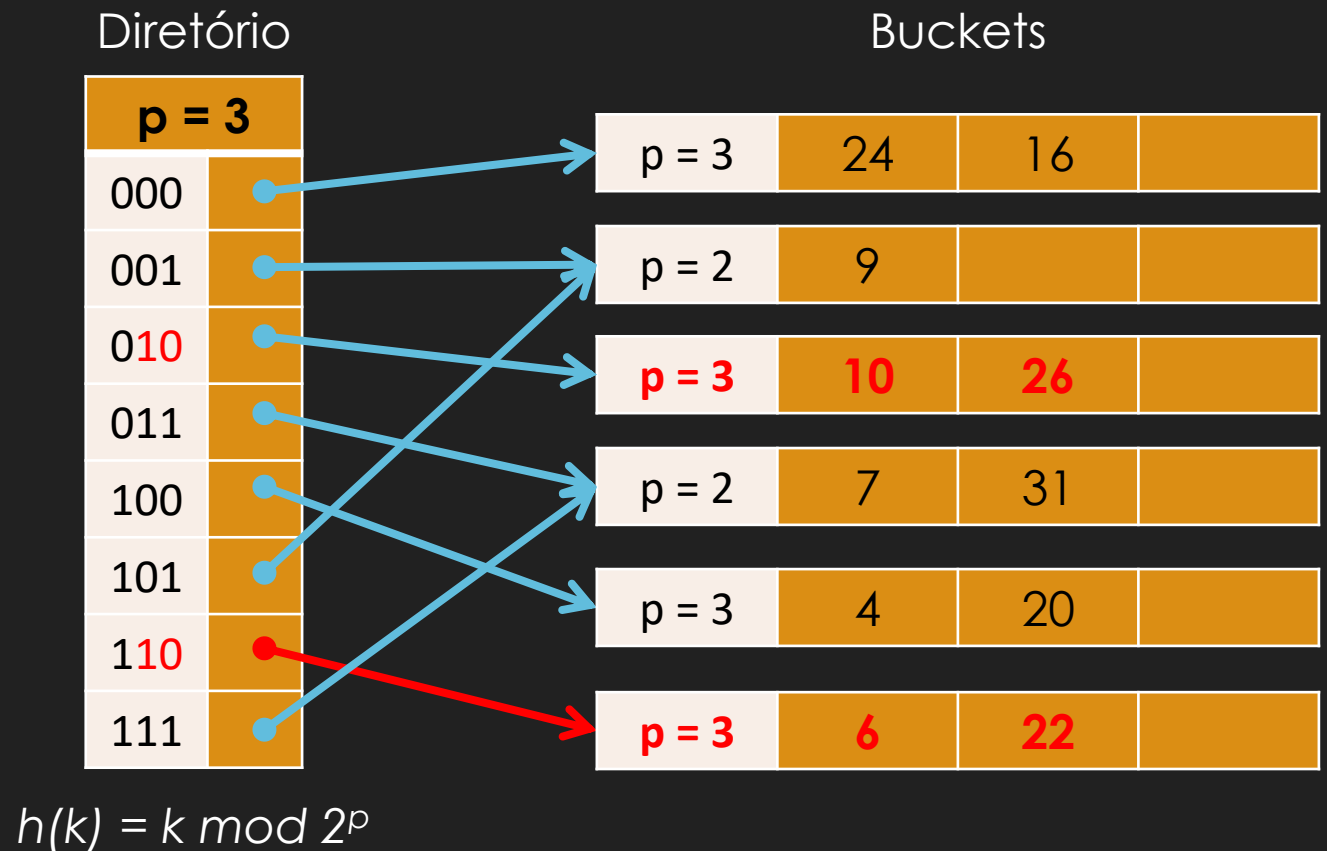
# Hashing Extensível | Exemplo

- Dobrar novamente a tabela de diretórios?
- Profundidade do *bucket* cheio é menor que a do diretório: **p = 2**
- Isso significa que existe um ponteiro disponível para a criação de um novo *bucket*, sem necessidade de dobrar a tabela



# Hashing Extensível | Exemplo

- Criar um novo *bucket*
  - Usar o ponteiro disponível para apontar para o novo *bucket*
    - É o que difere apenas no bit mais à esquerda
- Redistribuir as chaves do *bucket*
- Inserir a chave 26

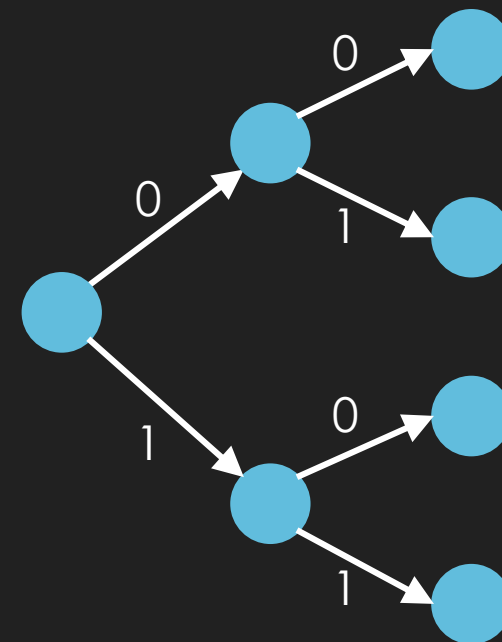


# Hashing Extensível | Trie

- O diretório de endereços equivale a uma Trie
  - Neste caso, a trie deve ser uma árvore binária completa
- Trie
  - árvore de busca na qual o número máximo de filhos por nó é igual ao número de símbolos do alfabeto que compõe as chaves
- Diretório é uma solução mais eficiente
  - Se for mantida como uma árvore, são necessárias várias comparações para descer ao longo de sua estrutura

Diretório

<b>p = 2</b>	
00	
01	
10	
11	



# Hashing Extensível

## Vantagens

- Desempenho do hash não diminui com o crescimento do arquivo
- Mínimo *overhead* de espaço
- O diretório cresce sem necessidade de reposicionar todos os registros do índice
- Como não há encadeamento dos *buckets*, não há perda de eficiência.

## Desvantagens

- Nível extra de indireção para encontrar registro desejado
- A tabela de endereços de *bucket* pode ficar muito grande (maior que a memória)
- Precisa de uma estrutura em árvore para achar o registro desejado na estrutura