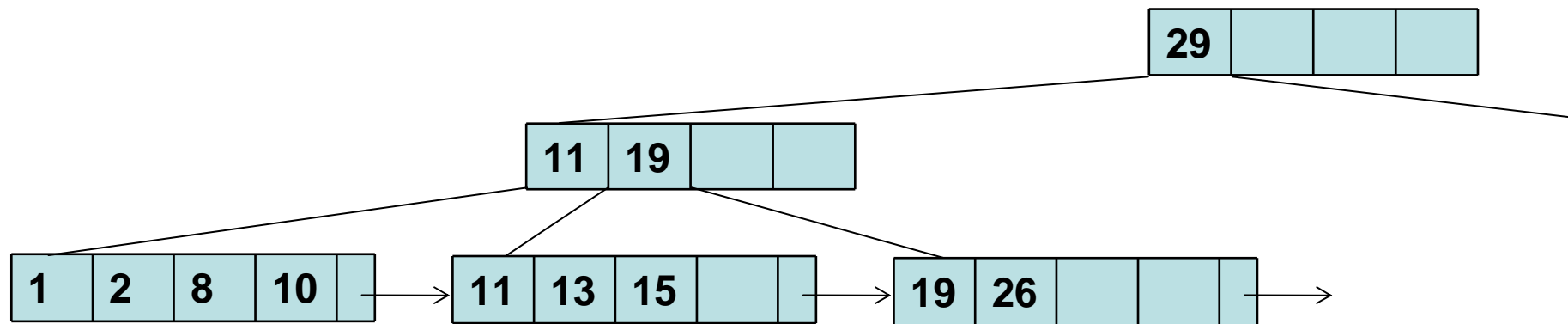


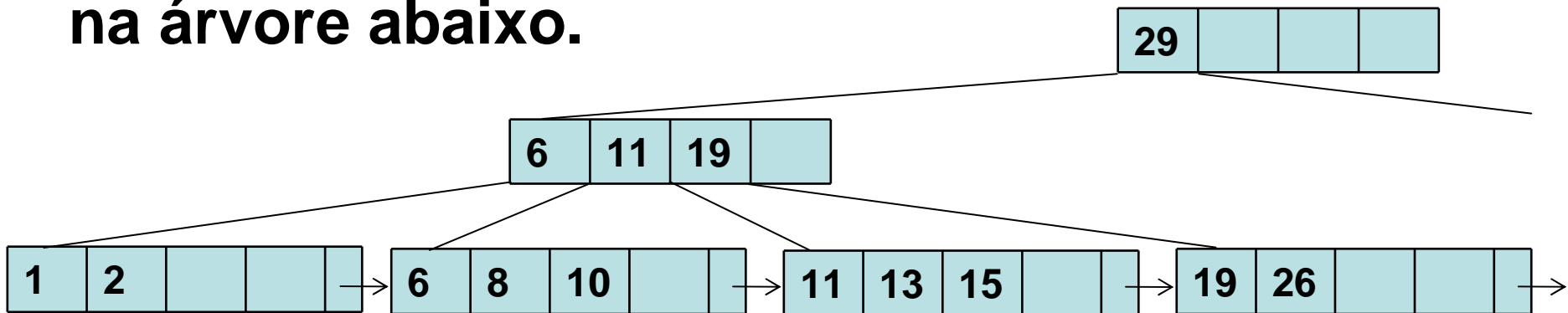
Árvores B+

As operações nas árvores B+ não são muito diferentes das operações das árvores B.

Inserir uma chave em uma folha que ainda tenha algum espaço exige que se coloque as chaves desta folha em ordem. Nenhuma mudança é feita no conjunto de índices. Se uma chave é inserida em uma folha cheia, a folha é dividida, o novo nó folha é incluído no conjunto de seqüência, as chaves são distribuídas homogeneamente entre a folha velha e a nova folha e a primeira chave do novo nó é copiada (**não** movida, como na árvore B) para o ascendente. Veja o slide a seguir.



A inserção da chave 6 da árvore acima resulta na árvore abaixo.



Se o ascendente não está cheio, isso pode exigir reorganização local das chaves do ascendente como no caso acima.

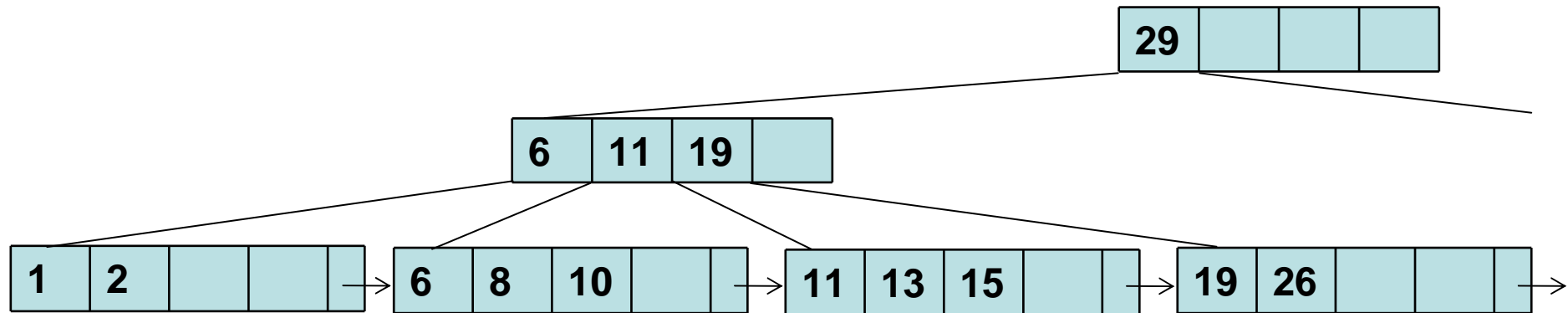
Se o ascendente está cheio, o processo de divisão é realizado do mesmo modo que nas árvore B. Depois de tudo, o conjunto de índices é uma árvore B.

Árvores B+

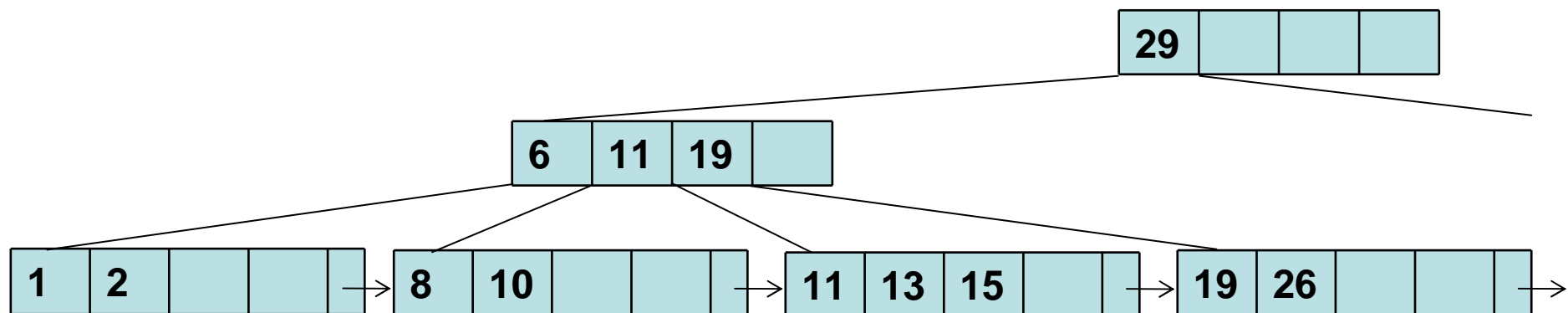
Remover uma chave de uma folha sem subaproveitamento exige colocar as chaves remanescentes em ordem, nenhuma mudança é feita ao conjunto de índices. Em particular, se uma chave que não ocorre somente em uma folha é removida, ela é simplesmente removida da folha, mas pode permanecer no nó interno.

A razão é que ele ainda serve como guia apropriado quando se navega para baixo na árvore B+, porque ainda separa apropriadamente as chaves entre dois filhos adjacentes mesmo se o próprio separador não ocorre em nenhum dos filhos. Observe o processo no próximo slide.

Árvores B+

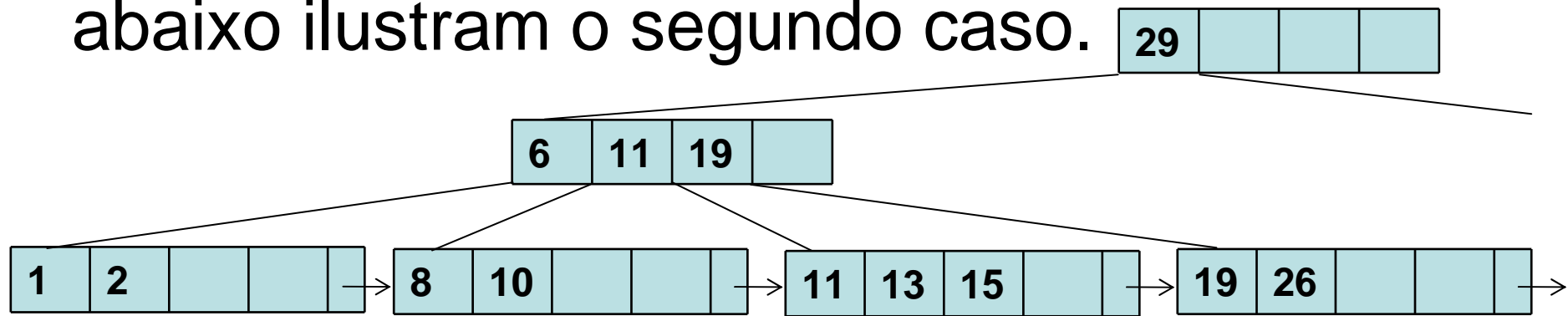


A remoção da chave 6 da árvore acima resulta na árvore abaixo. Note que o número 6 não é removido de um nó interno.

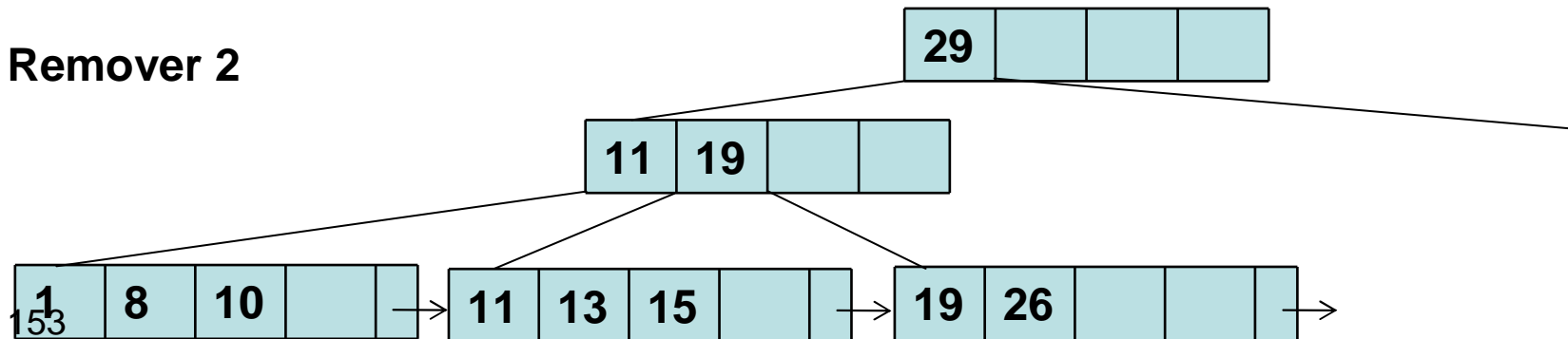


Árvores B+

Quando a remoção de uma chave causa subaproveitamento, ambas as chaves dessa folha e as chaves de um irmão são redistribuídas entre a folha e seu irmão, ou a folha é removida e as chaves remanescentes são incluídas no seu irmão. As árvores abaixo ilustram o segundo caso.



Remover 2



Árvores B+

Como vimos, depois de remover o número 2, um subaproveitamento ocorre e duas folhas são combinadas para formar uma folha. A primeira chave do irmão à direita do nó que permanece depois da fusão é copiada ao nó ascendente, e as chaves no ascendente são colocadas em ordem. Essas duas operações exigem atualização do separador no ascendente. Além disso, remover uma folha pode disparar fusões no conjunto de índices.

Tabelas de hash

Considerações iniciais:

- Até o momento foram estudados basicamente dois tipos de estruturas de dados para o armazenamento flexível de dados:
 - Listas;
 - Árvores.
- Como vimos, cada um desses grupos possui muitas variantes.

Tabelas de hash

As **Listas** são simples de se implementar, mas, com um tempo médio de acesso $T = n/2$, tornando-se impraticáveis para grandes quantidades de dados.

- Por exemplo: em uma lista com 100.000 dados, para recuperar 3 elementos em seqüência faremos um número esperado de 150.000 acessos a nodos.
- Logo: listas são ótimas porém, para pequenas quantidades de dados.

Tabelas de hash

As **Árvores** são estruturas mais complexas, mas que possuem um tempo médio de acesso $T = \log_G n$, onde G = ordem da árvore.

- A organização de uma árvore depende da ordem de entrada dos dados.
- Para evitar a deterioração, há modelos de árvores que se reorganizam sozinhas. As principais são AVL e Árvore B.

Tabelas de hash

Imagine uma estrutura de dados que possibilite o armazenamento de uma tabela onde o acesso a um de seus registros seja efetuado diretamente.

Esta estrutura representa um modelo ideal, contudo em aplicações reais este ideal pode ser aproximado.

Ideal: Indexação direta (transformação chave-índice)

pesq(T:tabela, C: chave)

└─> baseado em $h(C:chave) \rightarrow \text{índice}$

Tabelas de hash

Uma tabela hash, também chamada de tabela de dispersão ou tabela de escrutínio, trata-se de uma forma extremamente simples, fácil de se implementar e intuitiva de se organizar grandes quantidades de dados, permitindo armazenar e encontrar rapidamente dados por meio da utilização de uma chave.

A idéia central propõe a divisão de um universo de dados a ser organizado em subconjuntos mais gerenciáveis.

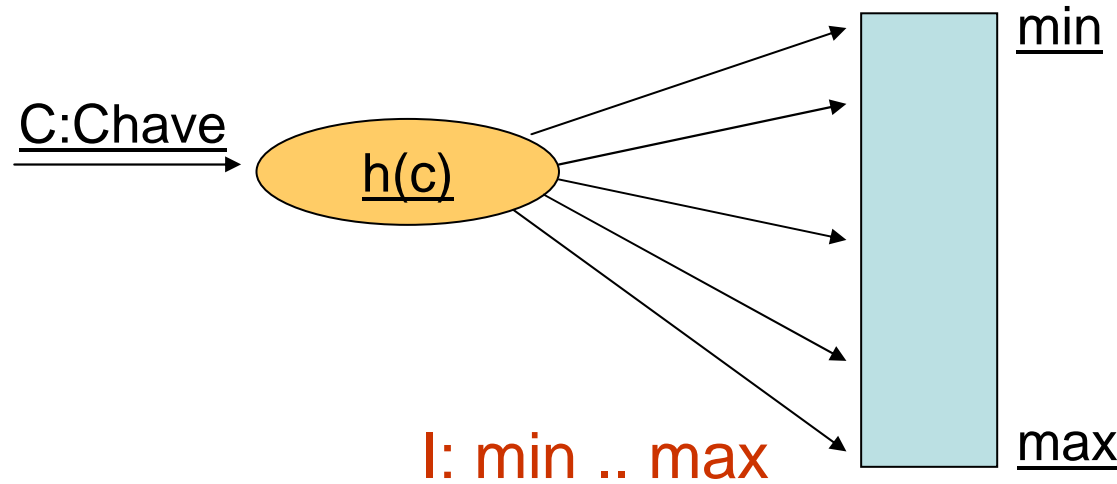
Tabelas de hash

Possui dois conceitos centrais:

Tabela de Hashing: Estrutura que permite o acesso aos subconjuntos.

Função de Hashing: Função que realiza um mapeamento entre valores de chaves e entradas na tabela.

$h(\text{chave}) \rightarrow \text{min} \dots \text{max}$



Tabelas de hash

É interessante ressaltar que uma tabela hash possui limitações em relação às árvores, como por exemplo:

- Não permite recuperar/imprimir todos os elementos em ordem de chave nem tampouco outras operações que exijam seqüência dos dados.
- Não permite operações do tipo recuperar o elemento com a maior ou a menor chave.

Tabelas de hash

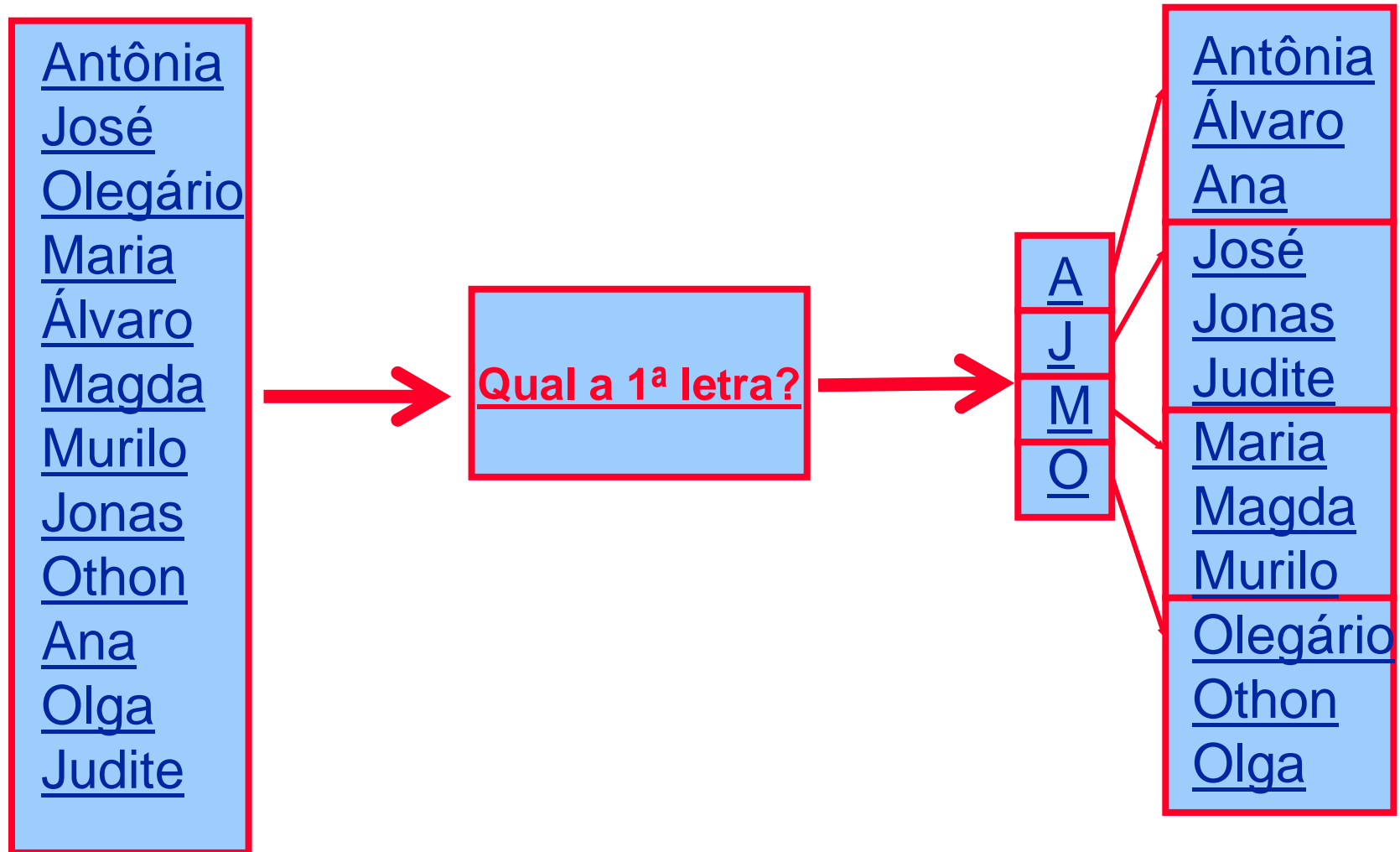
- Idéia geral: Se eu possuo um universo de dados classificáveis por chave, posso:
 - Criar um critério simples para dividir este universo em subconjuntos com base em alguma qualidade do domínio das chaves.
 - Saber em qual subconjunto procurar e colocar uma chave.
 - Gerenciar estes subconjuntos bem menores por algum método simples.

Tabelas de hash

Para isso é necessário:

- ✦ Saber quantos subconjuntos são desejados e criar uma regra de cálculo que permita, dada uma chave, determinar em qual subconjunto deve-se procurar pelos dados com esta chave ou colocar este dado, caso seja um novo elemento. (**função de hashing**).
- ✦ Possuir um índice que permita encontrar o início do subconjunto certo, depois de calcular o hashing. (**tabela de hashing**).
- ✦ Possuir uma ou um conjunto de estruturas de dados para os subconjuntos. **hashing fechado** (**endereçamento aberto**) ou o **hashing aberto** (**encadeado**).

Tabelas de hash: Exemplo



Tabelas de hash

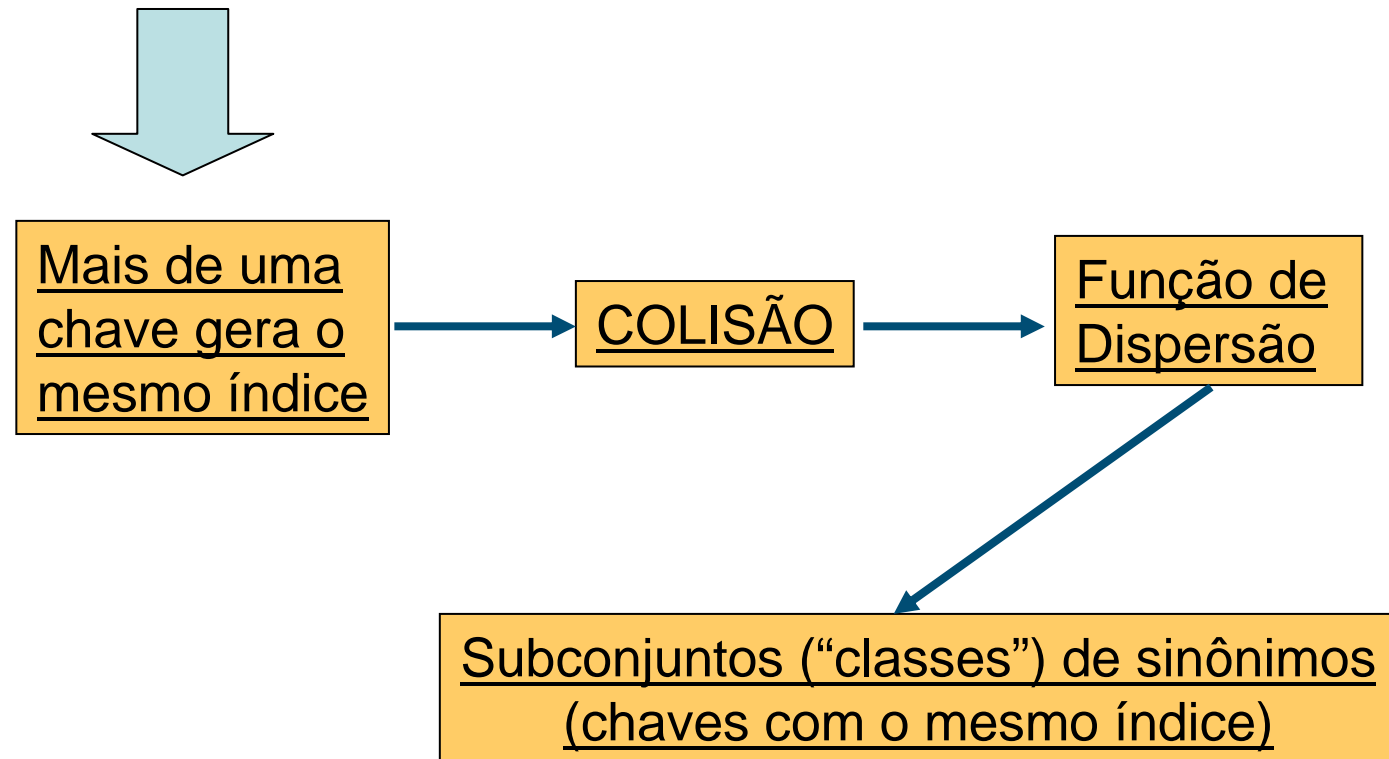
➡ Devemos garantir um tratamento coerente das operações de inserção e pesquisa na tabela

➡ **Inserir:** calcula-se endereço da nova entrada usando a função h

➡ **Pesquisa:** mesma função deve informar o mesmo índice para a mesma entrada

Tabelas de hash

➡ chaves -> índices



Tabelas de hash

- Implementação de tabelas por cálculo de endereço envolve dois problemas
- Escolha da **função de dispersão** adequada para o tipo de chave usada
 - gerar endereços em todo o intervalo [min..max]
 - estabelecer distribuição uniforme na tabela
- Adoção de um processo de solução das **colisões**

Tabelas de hash: Nomenclatura

n : Tamanho do universo de dados.

b : Número de subconjuntos em que dividimos os dados: **depósitos**.

s : **Capacidade de cada depósito** (quando for limitada. Aplica-se somente a hashing fechado ou endereçamento aberto).

T : Cardinalidade do domínio das chaves. Quantas chaves podem existir ?

n/T : **Densidade identificadora**.

$\alpha = n/(b \cdot s)$: **Densidade de carga**. **$b \cdot s$** fornece a capacidade máxima (quando existir explicitamente). **$n/(b \cdot s)$** indica o fator de preenchimento. Aplica-se somente a hashing fechado (endereçamento aberto).

Hashing Aberto ou Encadeado

- Forma mais intuitiva de se implementar o conceito de Hashing.
- Utiliza a idéia de termos uma tabela com b entradas, cada uma como cabeça de lista para uma lista representando o conjunto b_i .
 - Calculamos a partir da chave qual entrada da tabela é a cabeça da lista que queremos.
 - Utilizamos uma técnica qualquer para pesquisa dentro de b_i . Tipicamente será a técnica de pesquisa seqüencial em lista encadeada.
 - Podemos utilizar qualquer outra estrutura para representar os b_i . Uma árvore, por exemplo, poderia ser uma opção.

Propostas de Função de Dispersão

- Um dos métodos mais simples para criar funções hashing é o método da divisão. No qual uma chave **C** é mapeada para um dos **b** endereços da tabela hashing.

► Resto da Divisão

$$h(C) = C \bmod b$$

\downarrow
 $[0, b-1]$

nº máximo de entradas da tabela

Hashing Aberto ou Encadeado: Exemplo

Para uma melhor compreensão analisaremos um exemplo no qual uma tabela com 8 entradas é utilizada para acomodar registros cujas chaves são valores pertencentes ao conjunto dos números naturais.

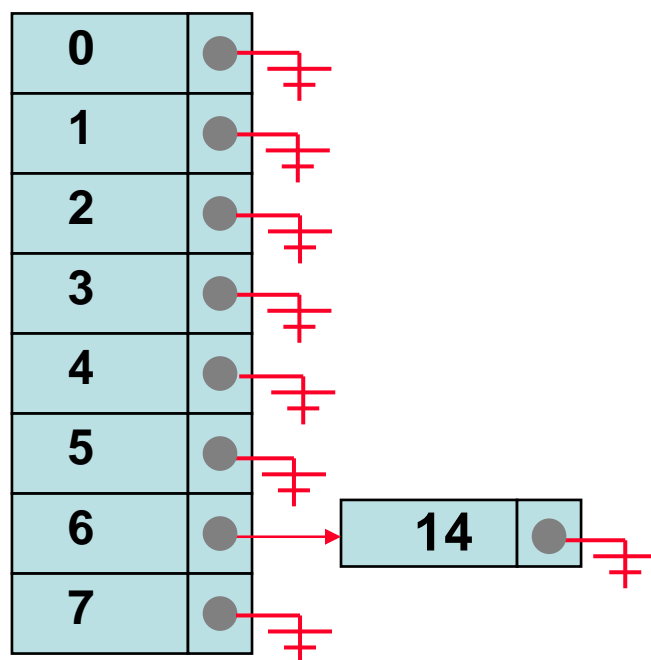
Inicialmente a tabela está vazia.

0	●	→
1	●	→
2	●	→
3	●	→
4	●	→
5	●	→
6	●	→
7	●	→

Hashing Aberto ou Encadeado: Exemplo

Inserir chave 14.

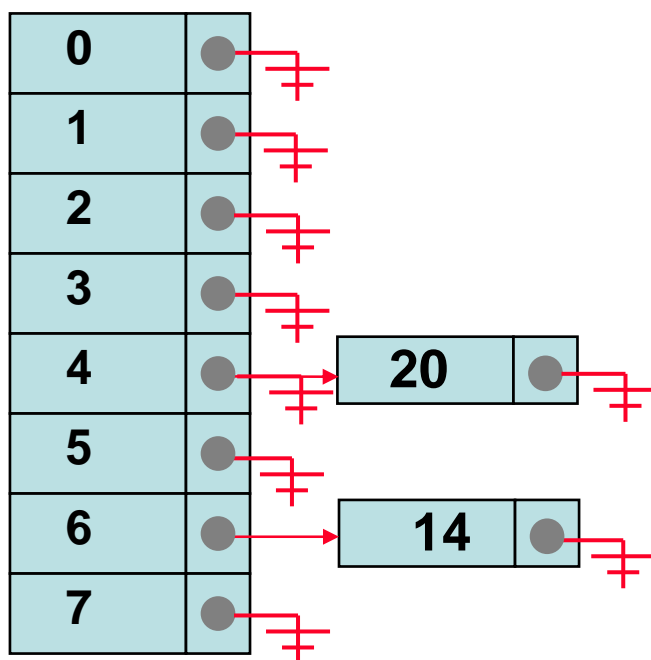
$$14 \% 8 = 6$$



Hashing Aberto ou Encadeado: Exemplo

Inserir chave 20.

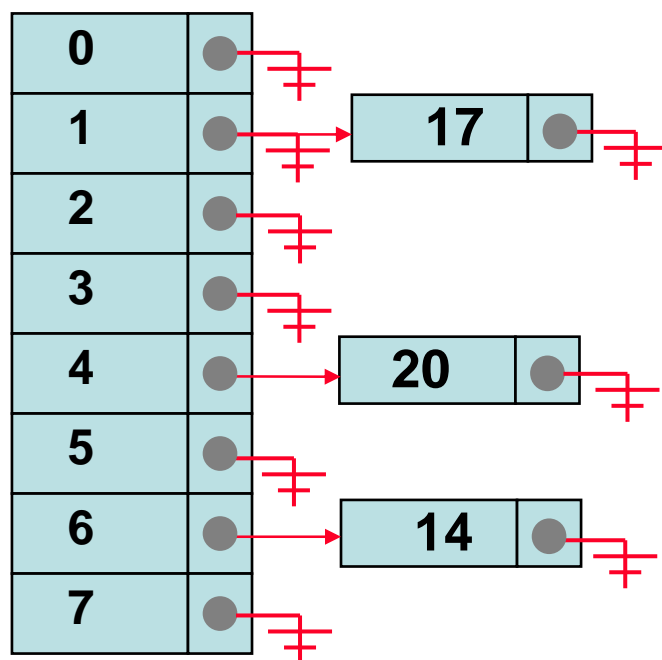
$$20 \% 8 = 4$$



Hashing Aberto ou Encadeado: Exemplo

Inserir chave 17.

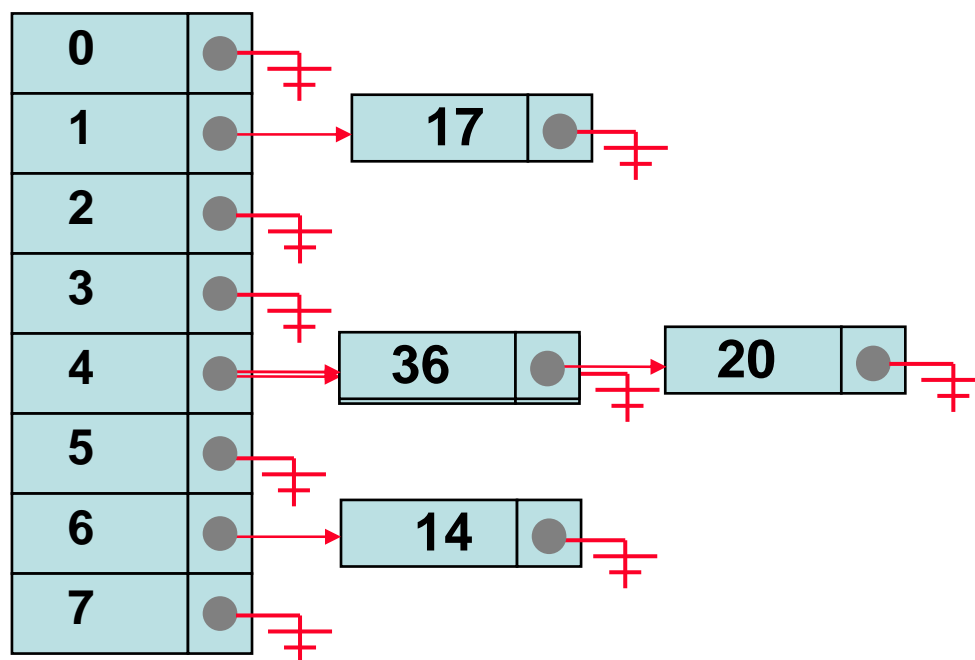
$$17\%8=1$$



Hashing Aberto ou Encadeado: Exemplo

Inserir chave 36.

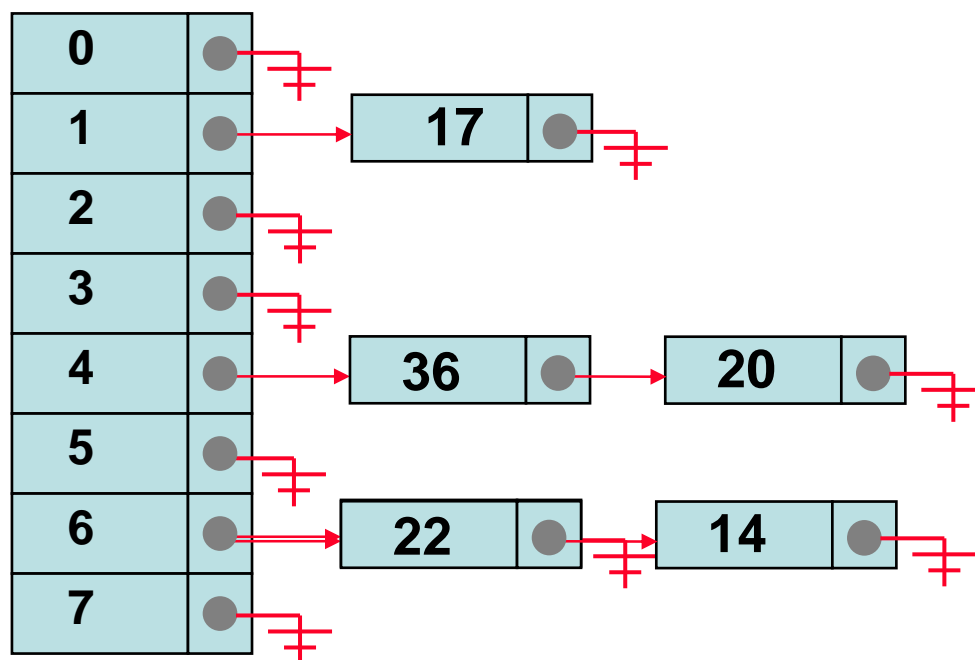
$$36 \% 8 = 4$$



Hashing Aberto ou Encadeado: Exemplo

Inserir chave 22.

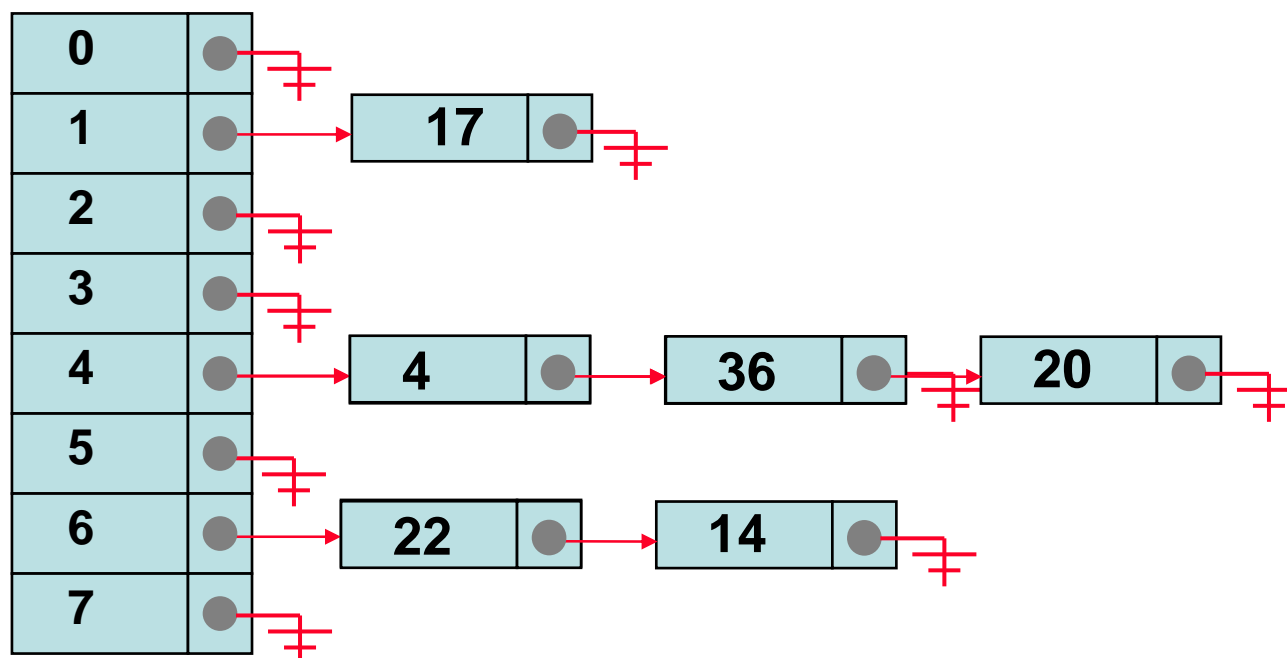
$$22 \% 8 = 6$$



Hashing Aberto ou Encadeado: Exemplo

Inserir chave 4.

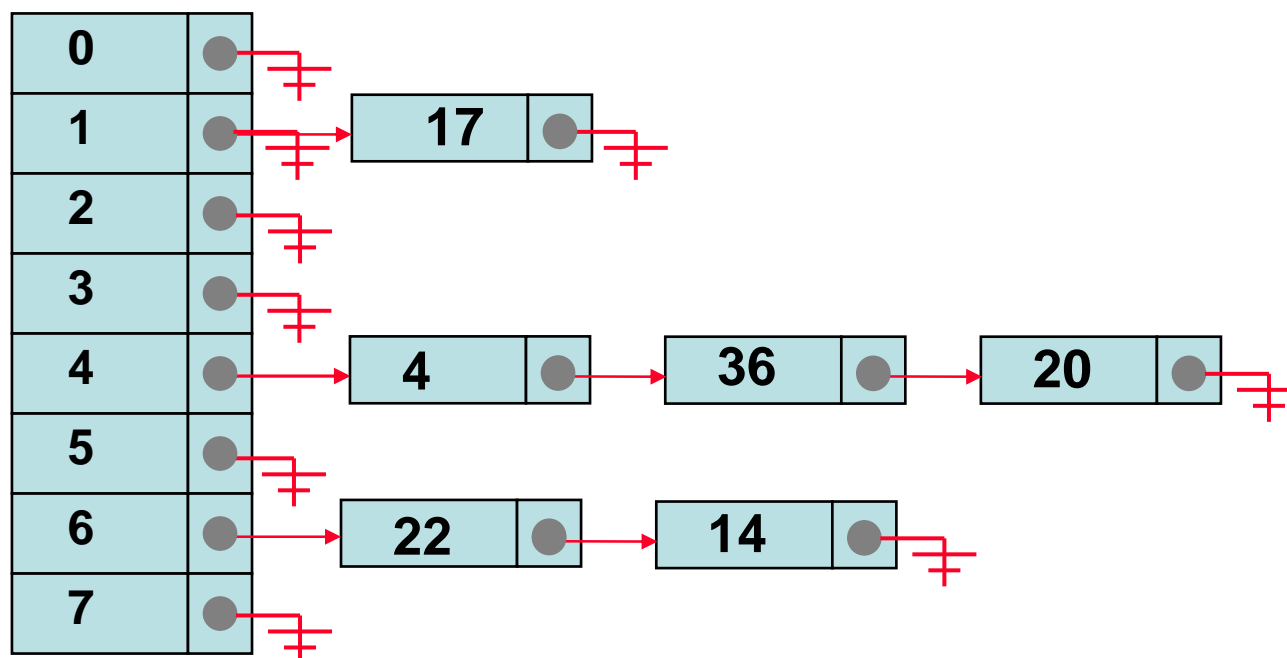
$$4 \% 8 = 4$$



Hashing Aberto ou Encadeado: Exemplo

Remover chave 17.

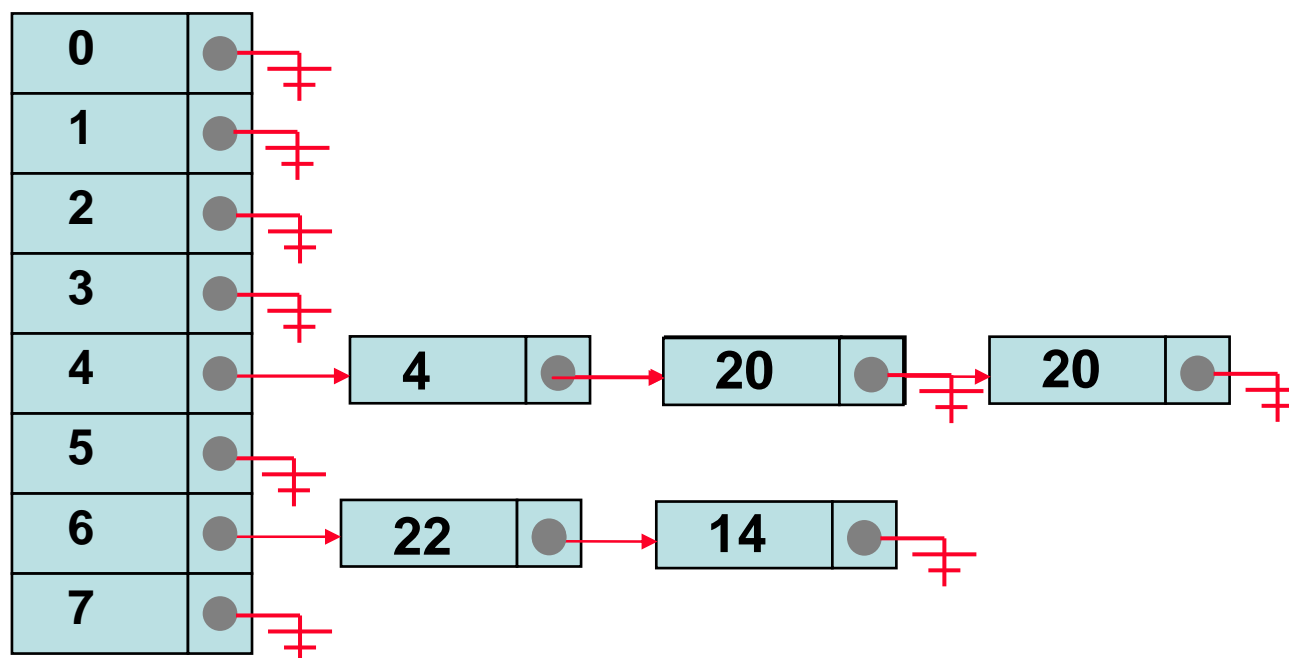
$$17\%8=1$$



Hashing Aberto ou Encadeado: Exemplo

Remover chave 36.

$$36 \% 8 = 4$$



Hashing Aberto ou Encadeado: Exercício

Com base no que foi apresentado defina a(s) estrutura(s) de dados necessária(s) para a implementação de uma tabela hashing aberta.

```
#define numEntradas 8  
typedef struct _Hash  
{  
    int chave;  
    struct _Hash *prox;  
} Hash;  
typedef Hash* Tabela[numEntradas];
```


Hashing Aberto ou Encadeado: Exercício

Agora, implemente a função de inserção de uma chave na tabela hashing aberta em questão.

...