Algoritmos e Estrutura de Dados I
Tabela de Dispersão (Hash)

Michel Pires da Silva
michel@cefetmg.br

Departamento de Computação
DECOM-DV

Centro Federal de Educação Tecnológica de Minas Gerais
CEFET-MG

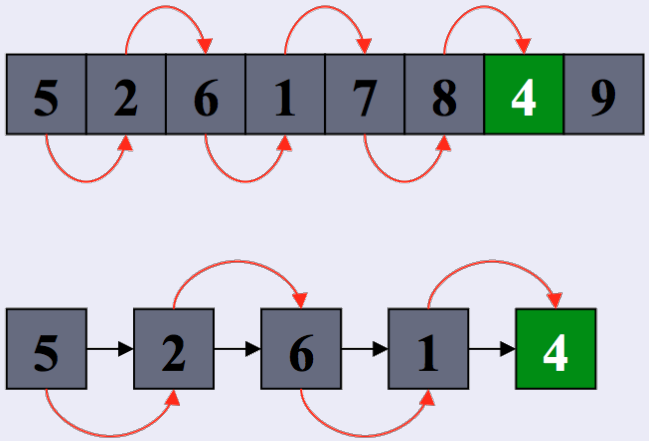
Sumário

Tabela Hash

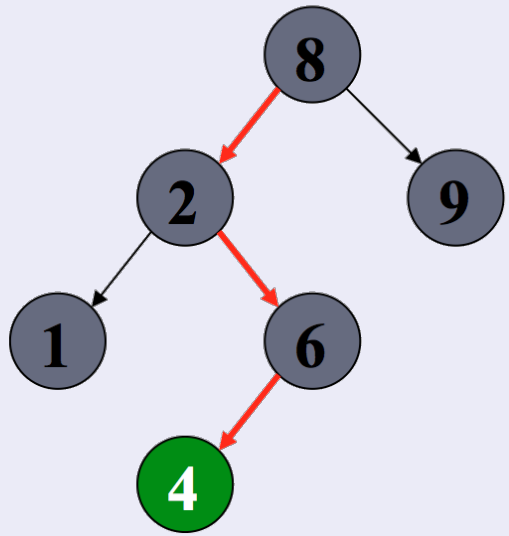
Motivação: Indexação de dados para a composição de uma organização que apresente eficiência e eficácia para o processo de recuperação da informação.

Exemplos

Arrays e listas



Árvores



Necessidade: Encontrar e manipular **rapidamente** elementos, evitando alto custo das operações computacionais.

Tabela Hash

Princípio: Garante a eficiência e eficácia por meio de modelos matemáticos, conhecidos como funções de hash. Estes modelos permitem a realização de operações de manipulação, como inserções, buscas e remoções, com custo assintótico constante na maioria dos casos. Além disso:

Características . . .

1. **Distribuição uniforme dos dados**: A função de hash deve distribuir os dados de maneira equilibrada pela tabela, minimizando colisões e garantindo eficiência nas operações de busca, inserção e remoção.
2. **Independência de distribuição de chaves**: A função de hash deve operar eficientemente e distribuir uniformemente os dados, independentemente da distribuição dos dados de entrada.
3. **Redimensionamento dinâmico**: Quando a tabela atinge um fator de carga crítico, ela é redimensionada (geralmente dobrada) e os elementos são redistribuídos para novas posições.
4. **Funções de hash criptográficas**: Garantem que pequenas mudanças nas chaves resultem em hashes significativamente diferentes, protegendo contra ataques de colisão e preservando a integridade dos dados.
5. **Simplicidade e flexibilidade**: Relativamente simples de implementar, as tabelas de hash são adaptáveis para diversas aplicações, como armazenamento de pares chave-valor e caches.
6. **Espaço de memória eficiente**: Oferece um uso eficiente da memória, especialmente em comparação com outras estruturas de dados que podem ter overheads adicionais.

Tabela Hash

Funções Hashing

- Os registros são endereçados a partir de uma *função de transformação* aplicada sobre a chave de pesquisa
- Seja N o tamanho da tabela
  - A função de *hashing* mapeia as chaves de entrada em inteiros dentro do intervalo [1 . . . N]
- Formalmente, a função de hashing $h(k_j) \rightarrow [1 \ldots N]$ recebe uma chave $k_j \in \{k_0, \ldots, k_n\}$ e retorna um número i referente ao índice do subconjunto $n_i \in [1 \ldots N]$ no qual o elemento que possui essa chave vai ser manipulado

Tabela Hash

Observação: A **função de *hashing*** deve ser fácil de computar e adequada para distribuir equiprovávelmente as chaves na tabela

Existem, em literatura, inúmeras **funções *hashing***, algumas das mais conhecidas são:

- Resto da Divisão
- Meio do Quadrado
- Método da Dobra $\longrightarrow$
- Método da Multiplicação
- *Hashing* Universal
- Similaridade de Jaccard (*Hashing Minwise*)
  $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$
- Assinatura Binária (*Hashing Simhash*)

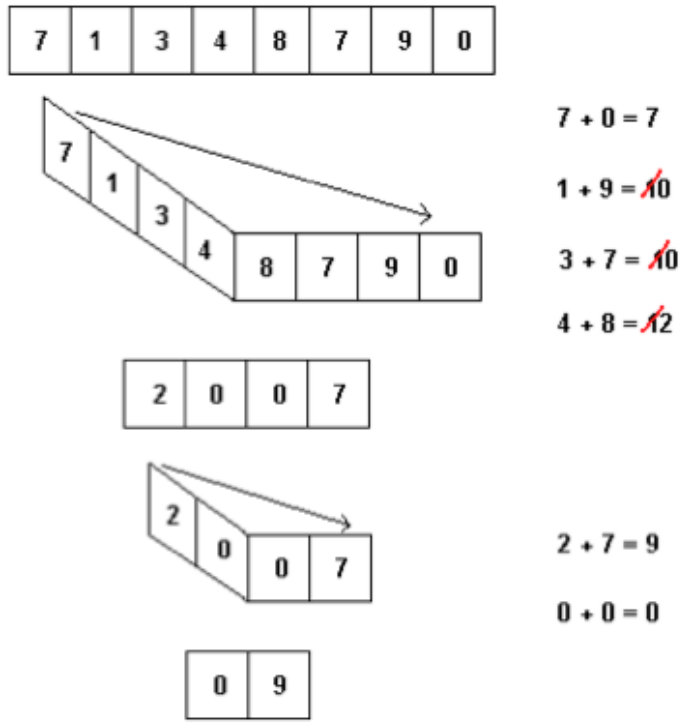


Tabela Hash

Resto da Divisão

O resto da divisão é o método mais simples e utilizado para o mapeamento de tabelas *hash*

- Nesse tipo de função a chave é interpretada como um valor numérico fornecido como entrada para a função *Hashing*.

Observação

O endereço de um elemento na tabela é dado simplesmente pelo resto da divisão da chave por M ($F_h(x) = x \ mod \ M$), onde M refere-se ao tamanho da tabela e x representa um valor inteiro correspondente a chave

Tabela Hash

Resto da Divisão e suas desvantagens

- *M* deve ser um número primo
- Valores recomendáveis para *M* devem ser > 20
- Seja qual for a função, na prática existem **sinônimos**, ou seja, chaves distintas que geram o mesmo valor como resultado da função *Hashing*
- Quando dois ou mais **sinônimos** são mapeados em uma mesma posição diz-se que ocorreu uma colisão

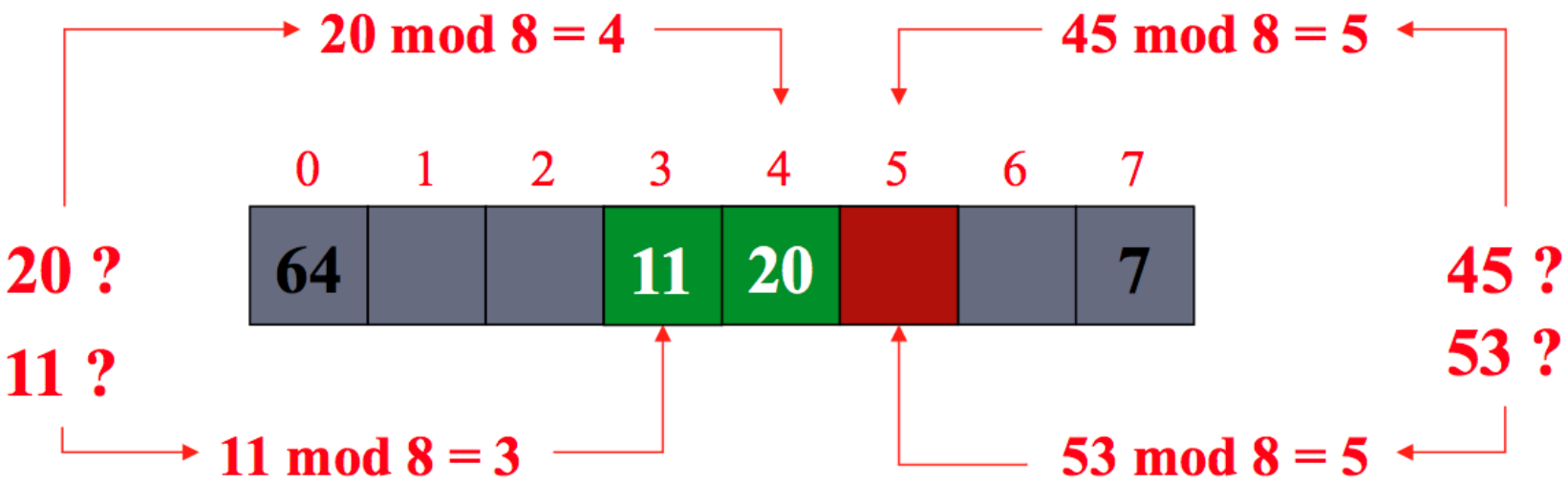


Tabela Hash

As colisões

- Qualquer que seja a função de transformação, existe a possibilidade de **colisões**, as quais devem ser resolvidas, mesmo que se obtenha uma distribuíção uniforme de dados.

- O ideal é uma função *Hashing*, tal que, dada uma chave $1 \leq I \leq N$, a probabilidade da função retornar uma chave x única é de $PROB(F_h(x) = I) = \frac{1}{N}$ para que não haja colisões

Observação

Obter uma função como definido acima é extremamente difícil, se não impossível.

Tabela Hash

Porque colisões vão existir ?

- Em 1968, o matemático Feller apresentou a comunidade científica o chamado **paradoxo do aniversário**
  - Em um grupo de 23 ou mais pessoas, reunidas ao acaso, existe uma probabilidade de mais de 50% de que duas comemorem o aniversário no mesmo dia.
- Direchlet, no século XIX apresentou um princípio chamado: **princípio da casa dos pombos**
  - Se K for um número inteiro positivo e $K + 1$ ou mais objetos são colocados em K caixas, então há uma caixa que terá dois ou mais objetos

Tabela Hash

Há, em litaratura, alguns algoritmos cujo objetivo é tratar as colisões ou evita-las o máximo possível. São eles:

- Endereçamento Fechado
- Endereçamento Aberto
  - *Hashing* Linear
  - *Hashing* Duplo

Tabela Hash

Endereçamento Fechado (1)

- Também encontrado como ***Overflow* Progressivo Encadeado**
- **Algoritmo:** Utiliza uma lista encadeada para cada endereço da tabela
- **Vantagem:** O processo de construção é simples e só sinônimos são acessados com maior tempo em uma busca

Desvantagens

- É necessário um campo extra para os ponteiros de ligação
- Tratamento especial das chaves: as que estão com endereço base e as que estão encadeadas

Tabela Hash

Endereçamento Fechado (2)

Exemplo: Dado uma tabela de tamanho 5 cuja estrutura irá armazenar as seguintes chaves: 20, 18 e 25.

$20 \ mod \ 5$

$18 \ mod \ 5$

$25 \ mod \ 5$

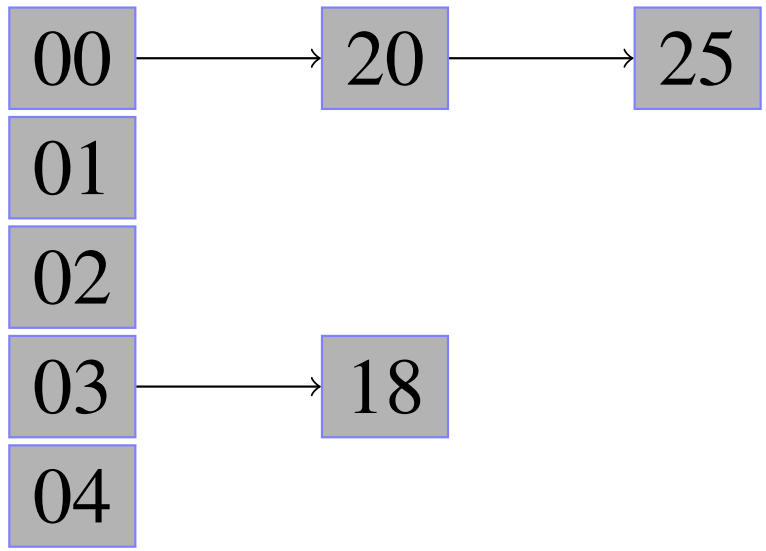
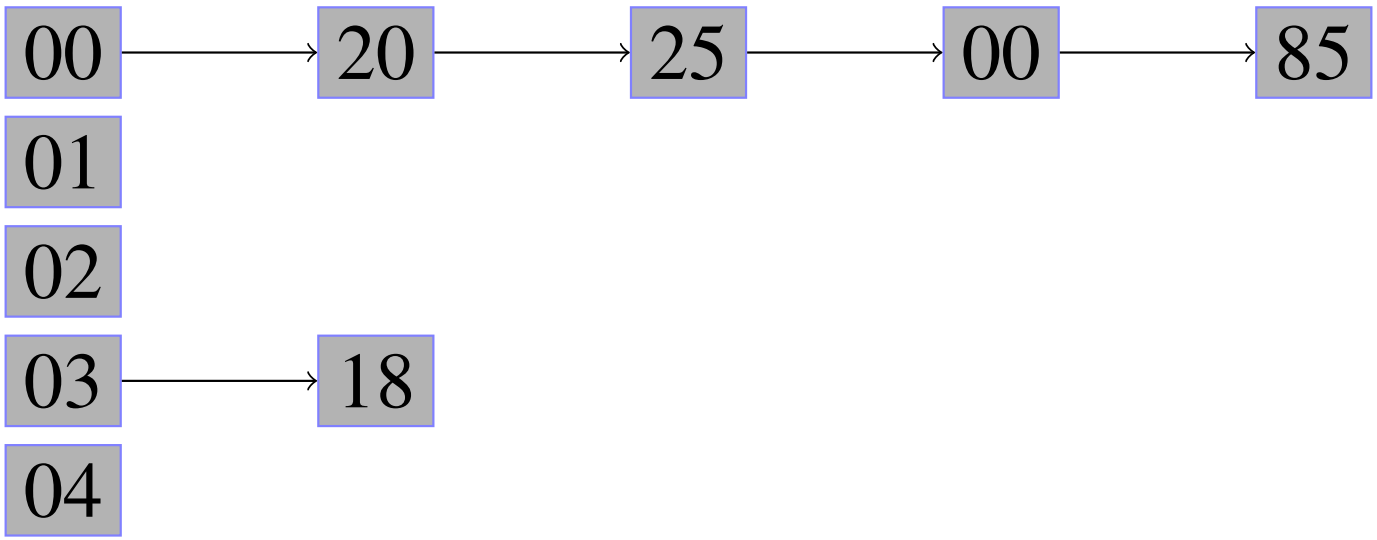


Tabela Hash

Endereçamento Fechado (3)

No endereçamento fechado a busca é feita, primeiramente, calculando-se o valor da função *hash* para a chave apresentada como entrada.

- Feito a conversão da chave em uma posição, o dado é encontrado caminhando-se sequencialmente pela lista linear



Desvantagem: A possibilidade de se obter uma lista linear sequêncial.

Tabela Hash

Endereçamento Fechado (4)

Minimizando o problema . . .

A redução do problema das listas lineares sequênciais pode ser obtida expandindo o tamanho da tabela e, consequentemente, a função *hashing*

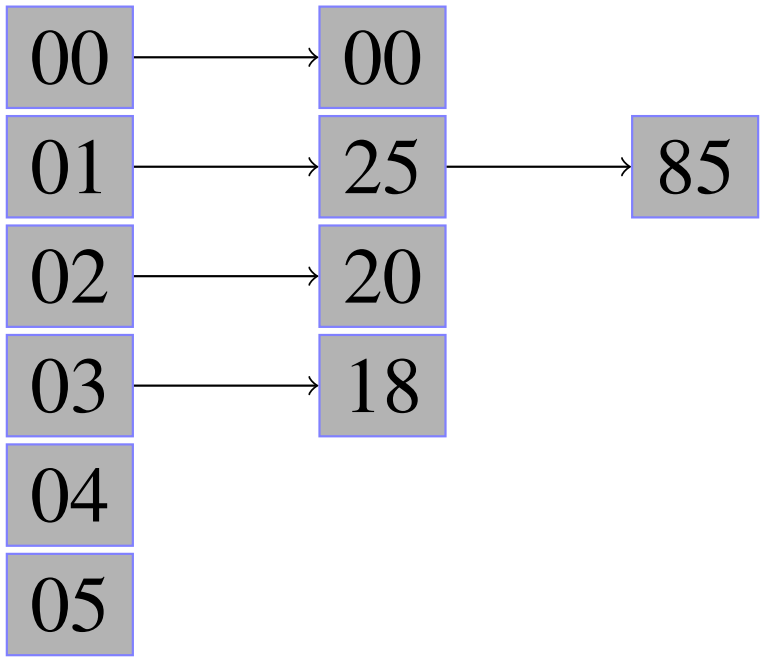


Tabela Hash

Endereçamento Aberto

Quando utilizar: Quando o número de registros a serem armazenados na tabela puder ser previamente estimados.

Nota

Para que essa estratégia corresponda bem as espectativas é preciso manter o fator $M > N$, onde, M é o tamanho da tabela e N o número de elementos a serem inseridos.

Observação: As características dessa estratégia tornam possível a alocação de todos os registros no próprio espaço da tabela sem a necessidade de listas encadeadas auxiliares.

Tabela Hash

Endereçamento Aberto - *Hash* Linear (1)

- A *hash* linear também é conhecida como ***Overflow* Progressivo**
- Esse método consiste em procurar a próxima posição vazia após o endereço base da chave avaliada
- **Vantagem:** Simplicidade na elaboração do algoritmo
- **Desvantagem:** Se ocorrerem muitas colisões pode haver um ***clustering*** (Agrupamento) de chaves em uma certa área da tabela. Isso pode causar muitos acessos para encontrar o registro requerido. Esse problema se agrava a medida que a ocupação da tabela aumenta.

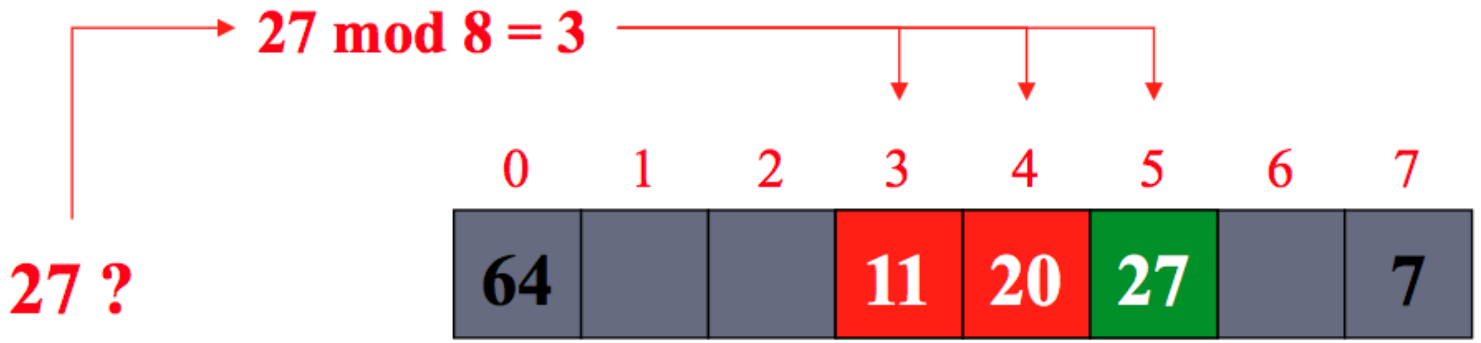


Tabela Hash

Endereçamento Aberto - *Hash* Linear (2)

Observação: No exemplo abaixo assume-se que a **HashFunction(key)** já está implementada

Algorithm 1: Function of hash initialize

input : $n \rightarrow$ number of positions
input : $H \rightarrow$ the data structure
output: Hash initialized

```
1 for key:=1 to n do
2 |   hash[key] := -1;
3 end
```

Algorithm 2: Insert function

input : $n \rightarrow$ number of positions
input : *value* $\rightarrow$ value to insert
input : $H \rightarrow$ the data structure
output: Hash updated

```
1 key := HashFunction(value);
2 while tabela[key] <> -1 do
3 |   key := (key + 1) mod n;
4 end
5 hash[key] := value;
```

Tabela Hash

Endereçamento Aberto - *Hash* Linear (2)

Exemplificação: Vejamos um exemplo que utiliza dos conceitos apresentados no algoritmo do slide anterior:

Elementos: {52, 78, 48, 61, 81, 120, 79, 121, 92}

Função: $hash(k) = k \ \ mod \ \ 13$

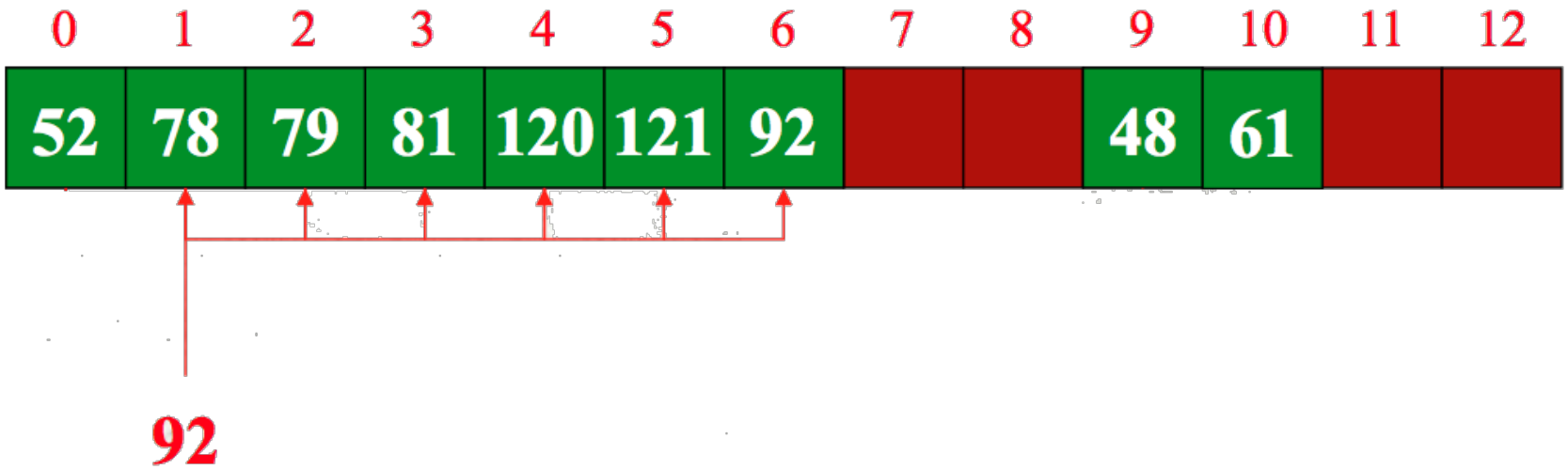


Tabela Hash

Endereçamento Aberto - *Hash* Duplo (1)

Observação

Note que o conceito de ***hash* linear** pode não gerar um bom espalhamento dos dados pelo vetor. Para isso, tem-se o *hashing* duplo ou **re-*hash***

No *hash* duplo não há incrementos em 1 a cada interação. Nesse, uma segunda função *hash* (função auxiliar) é utilizada para calcular a nova posição.

Vantagem: Tende a espalhar melhor as chaves pelos endereços disponíveis

Desvantagem: Os endereços podem estar muito distântes um do outro fazendo com que o princípio da localidade seja violada. Isso provoca ***seeking*** adicionais.

Endereçamento Aberto - *Hash* Duplo (2)

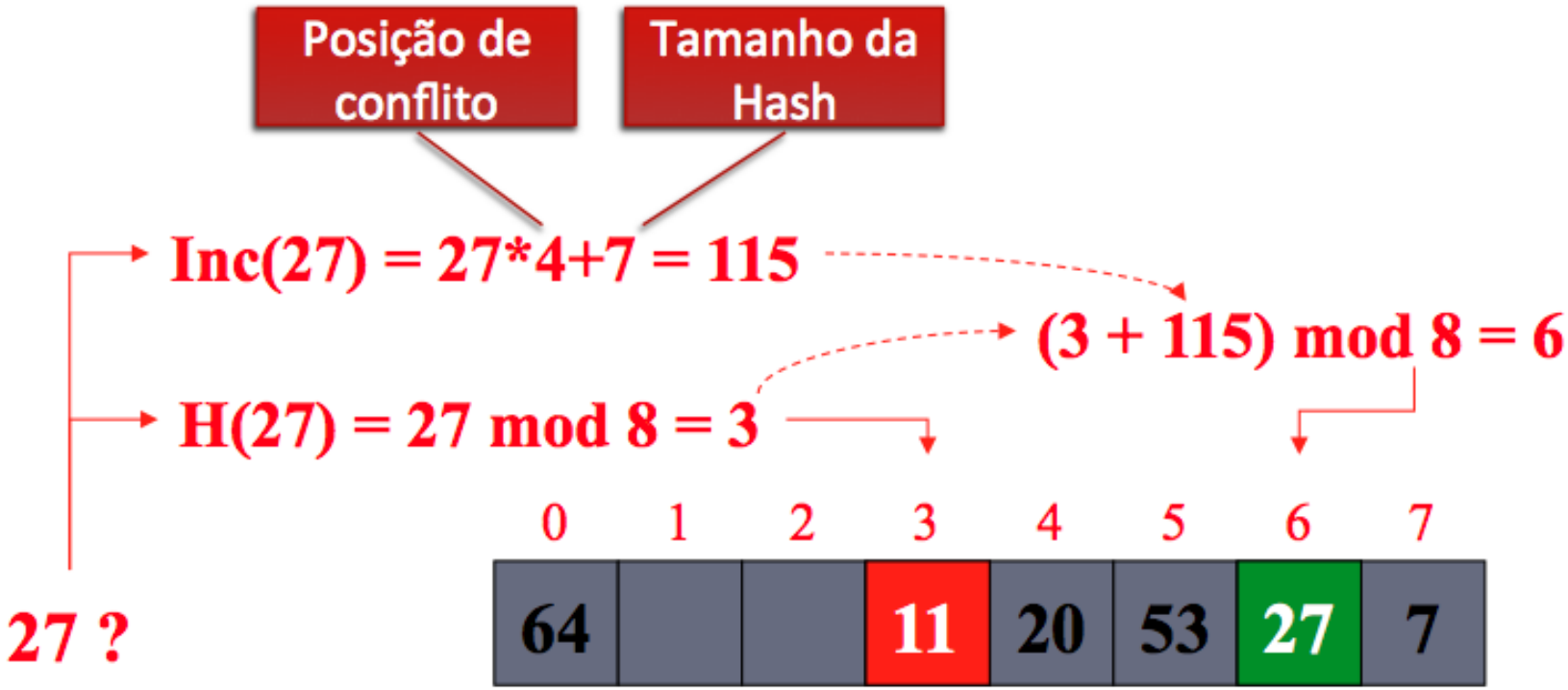


Tabela Hash

Endereçamento Aberto - *Hash* Duplo (3)

Algorithm 3: Function of hash initialize

input : $n \rightarrow$ number of positions
input : $H \rightarrow$ the data structure
output: Hash initialized

```
1 for key:=1 to n do
2 |   hash[key] := -1;
3 end
```

Algorithm 4: Insert function

input : $n \rightarrow$ number of positions
input : *value* $\rightarrow$ value to insert
input : $H \rightarrow$ the data structure
output: Hash updated

```
1 key := HashFunction(value);
2 while hash[key] <> -1 do
3 |   key :=
  |     HashFunction_2(value);
4 end
5 hash[key] := value;
```

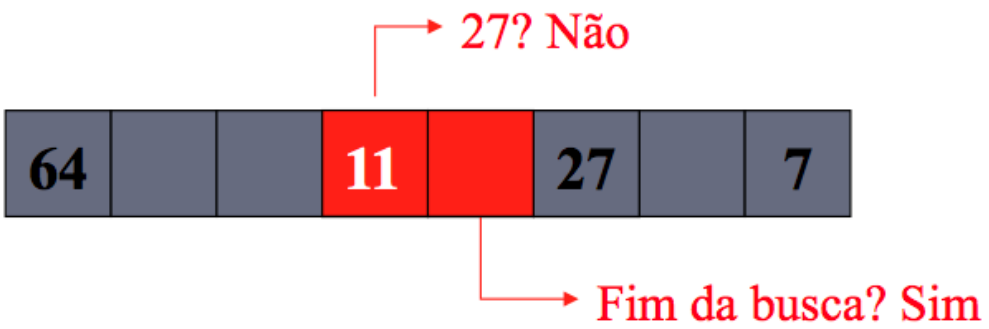
Tabela Hash

Endereçamento Aberto - as remoções (1)

Obervação: Para efetuar uma busca com endereçamento aberto basta aplicar a função *hashing* e a função de caminhamento até que o **elemento** ou **posição vazia** seja encontrada.

O custo: O conceito que o caminhamento aberto gera da-se o nome de **agrupamento**. Esse gera como problema, no pior caso, um custo de $O(n)$.

A remoção: Quando um elemento é removido da tabela, a situação de vazio ocorre antes de atingir o último elemento do **agrupamento**



Inserção do 27

Remoção do 20

Busca pelo 27

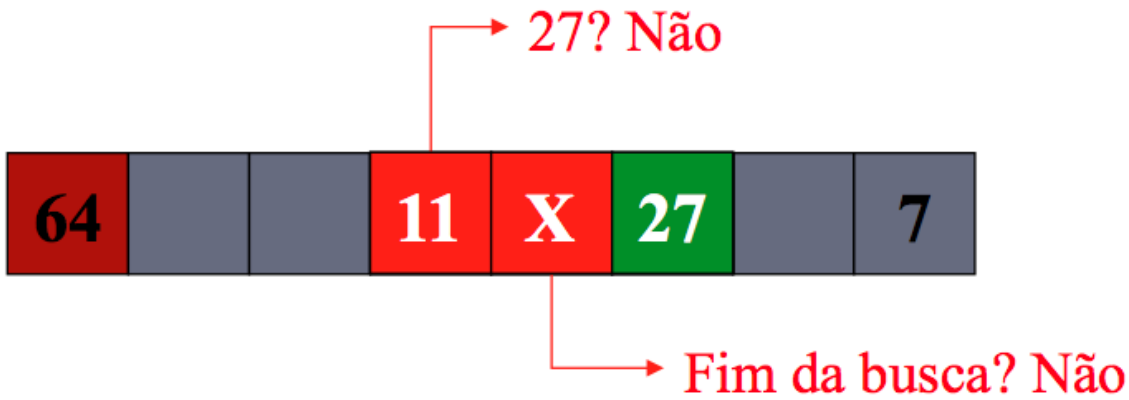
Tipo de hasing utilizada quando M > N, onde, M é o tamanho da tabela e N o número de elementos a serem inseridos na estrutura

SOLUÇÃO ?

Tabela Hash

Endereçamento Aberto - as remoções (2)

A solução: Manter um bit (ou campo booleano) na posição removida para indicar que um elemento foi removido e não é o fim do agrupamento.



Observação ...

A posição marcada com o bit estaria livre para novas inserções, mas, não seria tratada como vazia ao executar uma busca no agrupamento.

Tabela Hash

Observações Gerais

- Na estrutura de *hashing* há o que chamamos de **fator de carga** (*load factor*). Esse fator indica a porcentagem de células da tabela que estão ocupadas, incluindo as que foram removidas.

- Quando o *load factor* é alto (ex. acima de 50%), as operações passam a demorar mais, visto que, o número de colisões irá aumentar.

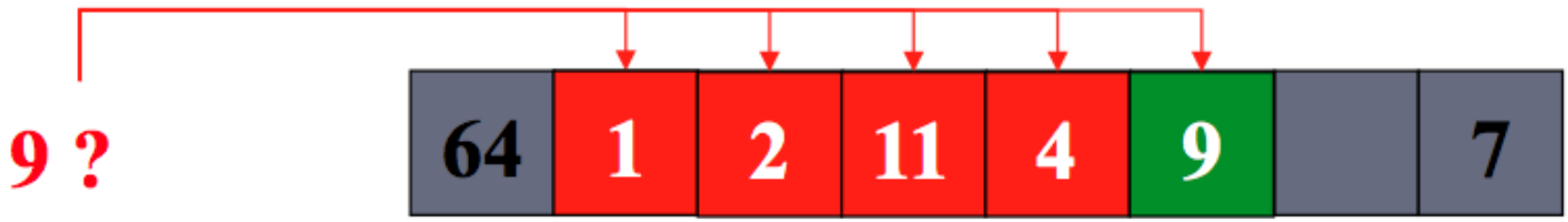


Tabela Hash

Observações Gerais

Tratando o *load factor*: Para tratar esse problema é necessário ampliar o tamanho da tabela e reorganizar os elementos na nova estrutura.

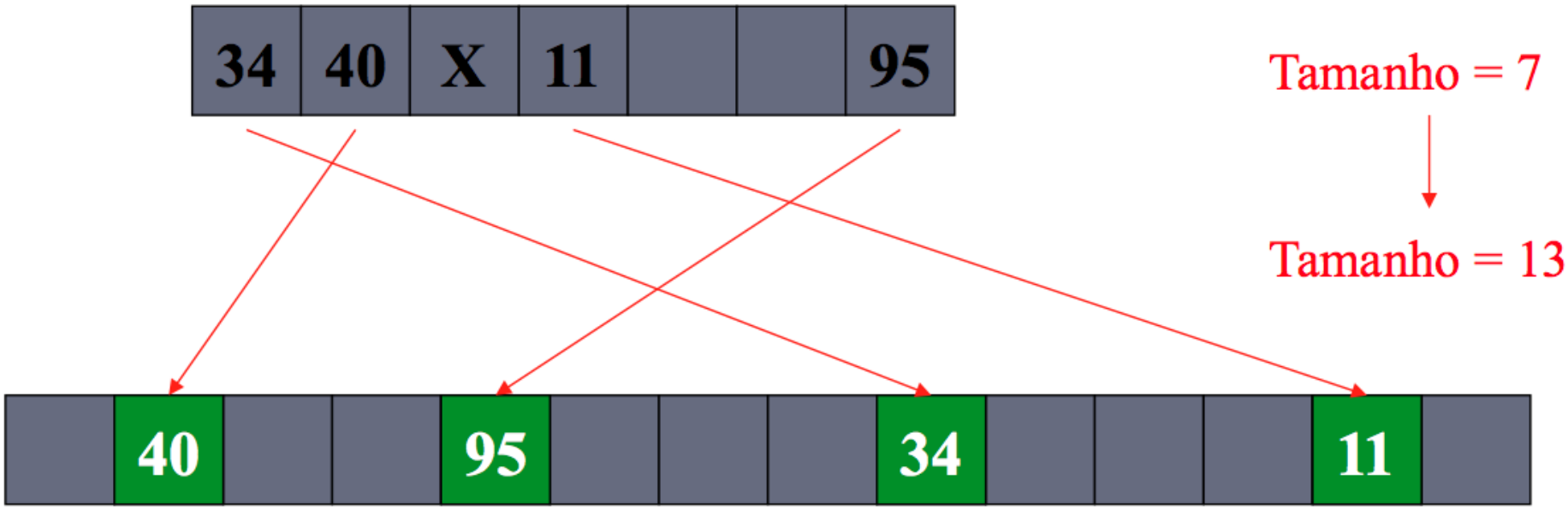


Tabela Hash

Observações Gerais

Quando ampliar a tabela *hash* ?

1. Quando não for possível inserir um elemento
2. Quando metade da tabela já estiver ocupada
3. Quando o *load factor* atingir um valor limite pré definido

Nota

A terceira opção é a mais comum, pois, trata-se de um meio termo entre o primeiro e segundo caso

Tabela Hash

Observações Gerais

Muitas colisões reduzem em muito o tempo de acesso e modificação dos dados na tabela *hash*. Os maiores fatores de influência são:

- A função *hash*
- O algoritmo de tratamento de colisões
- O Tamanho da tabela

Nota

Quando não for possível definir parâmetros eficiêntes para os itens acima é provável que uma outra estrutura, tal como, árvores balanceadas apresentem melhores resultados

Tabela Hash

Observações Gerais

A probabilidade de não haver colisões em uma tabela *hash* de tamanho M para N elementos é de $\frac{M!}{(M-N)!*M^N}$

Análise: O custo computacional com pesquisas, inserções e remoções de elementos em uma tabela *hash* é de $O(1 + \frac{N}{M})$ para o caso médio

Nota

O pior caso utilizando **endereçamento aberto** é utiliza-lo em conjunto com uma **lista encadeada**. Nesse caso o custo é de $O(n)$

Tabela Hash

Nota: Há outros modelos de hash operando sob os mesmos princípios:

- **Locality-Sensitive Hashing (LSH)**: LSH é uma família de técnicas de hashing projetadas para maximizar a probabilidade de colisão de objetos similares em buckets de hash. A principal ideia do LSH é que, ao contrário das funções de hash tradicionais que distribuem uniformemente os dados, as funções de hash sensíveis à localidade agrupam dados similares.

- **LSH para Espaços de Hamming (MinHash)**: MinHash é uma técnica específica dentro do framework de LSH, particularmente voltada para estimar a similaridade de Jaccard entre conjuntos. MinHash gera assinaturas pequenas para conjuntos, de modo que a probabilidade de duas assinaturas serem idênticas é igual à similaridade de Jaccard dos conjuntos originais.
- **Outras variantes**: LSH para Espaços Euclidianos, LSH baseado em Cosine Similarity, SimHash (LSH para Cosine Similarity), Super-bit LSH ...

Pra além da hash: Bloom Filter, Count-Min Sketch, HyperLogLog, Random Projection, Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), t-Distributed Stochastic Neighbor Embedding (t-SNE) ...

Tabela Hash

Curiosidade: A área da recuperação da informação ou *information retrieval* faz uso de estruturas como essa para organizar dados.

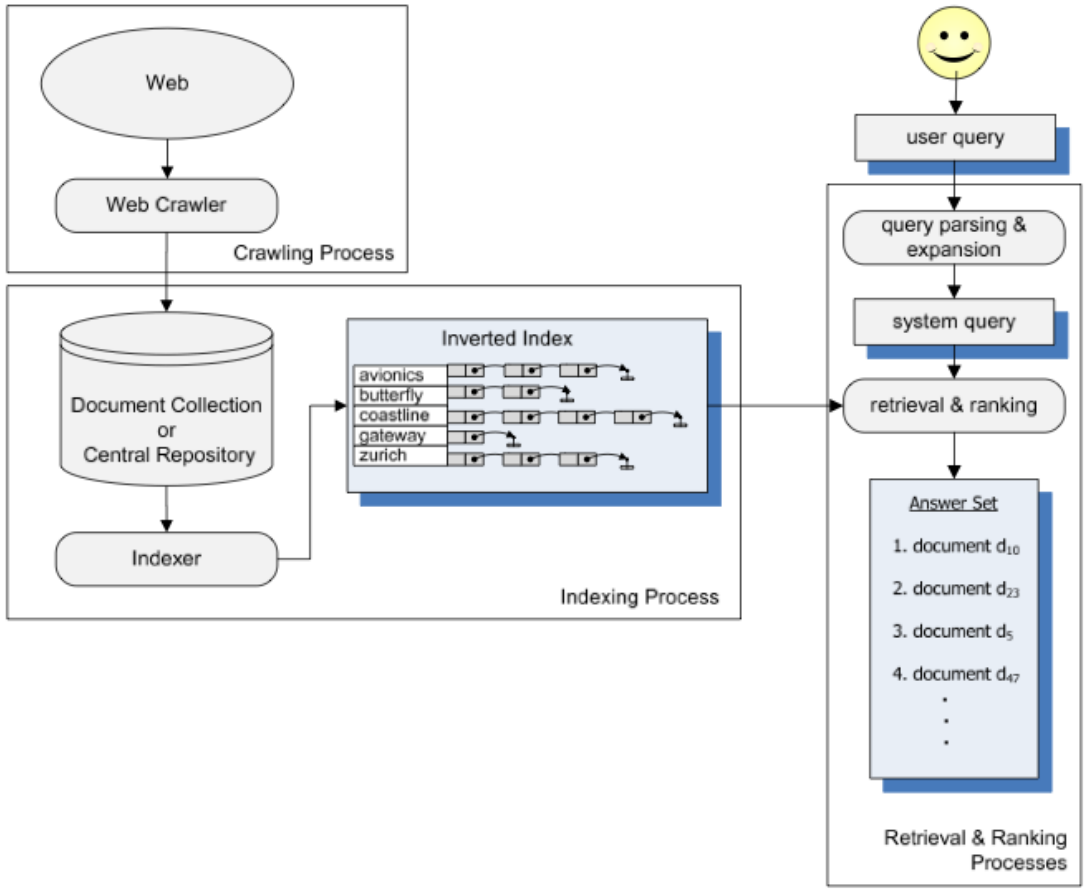


Figura: **Information Retrieval** by Han, J.; Kamber, M. Pei, J., *Data Minign: Concepts and Techniques*, ed 3^a, Morgan Kaufmann, 2000.

PERGUNTAS?