CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Busca Sequencial
Busca Binária
e *Hashing*

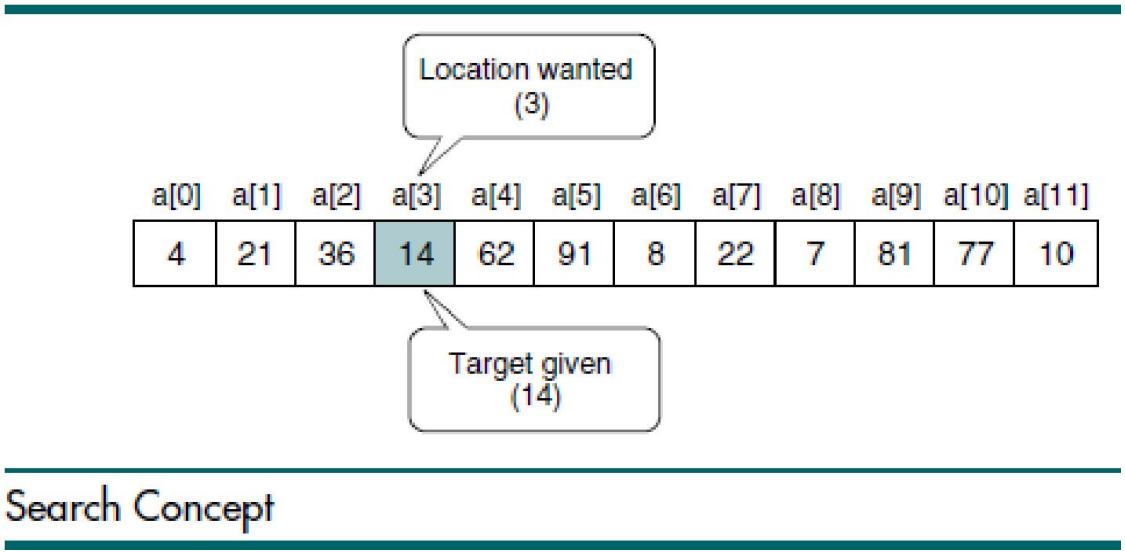
Disciplina: Estrutura de Dados II

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Busca Sequencial
Dados não-ordenados

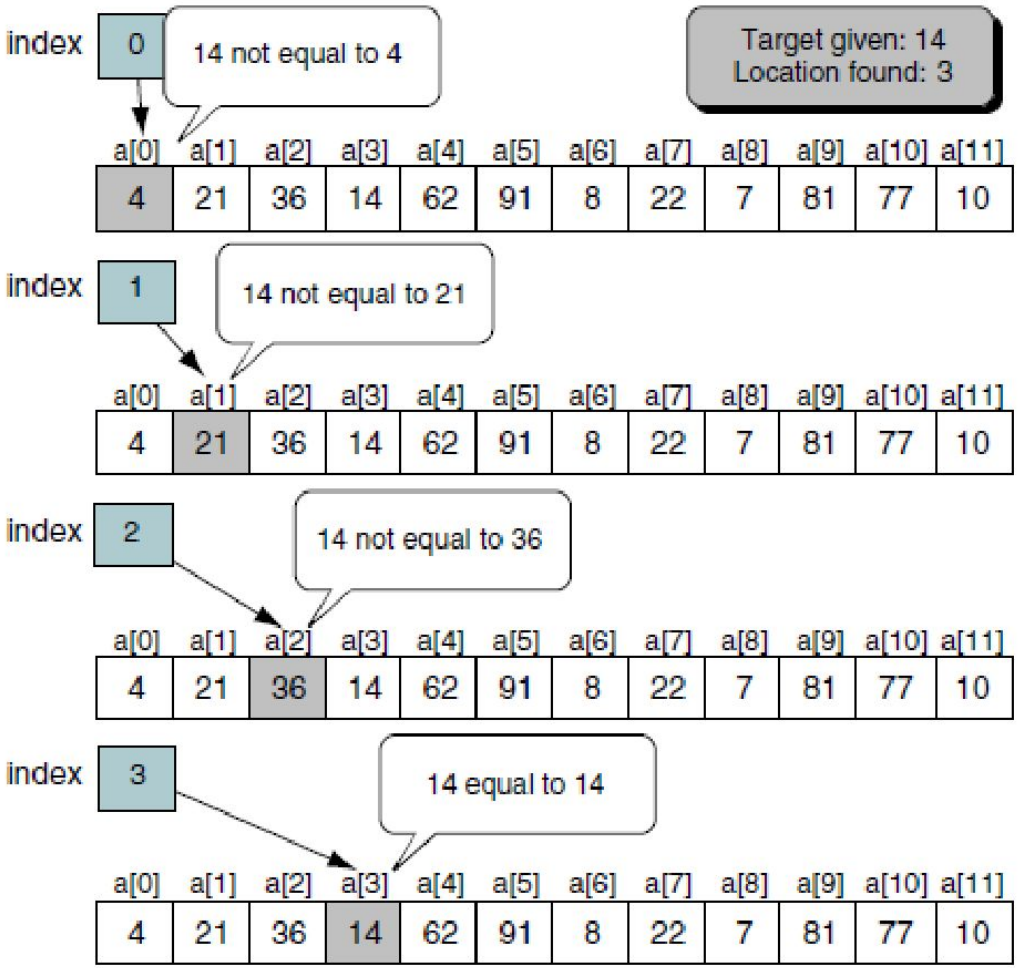
- A figura ilustra o conceito de busca de um certo dado (*target*) e a sua localização ou endereço (*location*).



Search Concept

Busca Sequencial
Dados não-ordenados

- O exemplo ilustra o processo de busca sequencial do elemento com valor 14, em uma lista não ordenada.
- Neste caso, como o elemento existe, a busca termina satisfatoriamente.

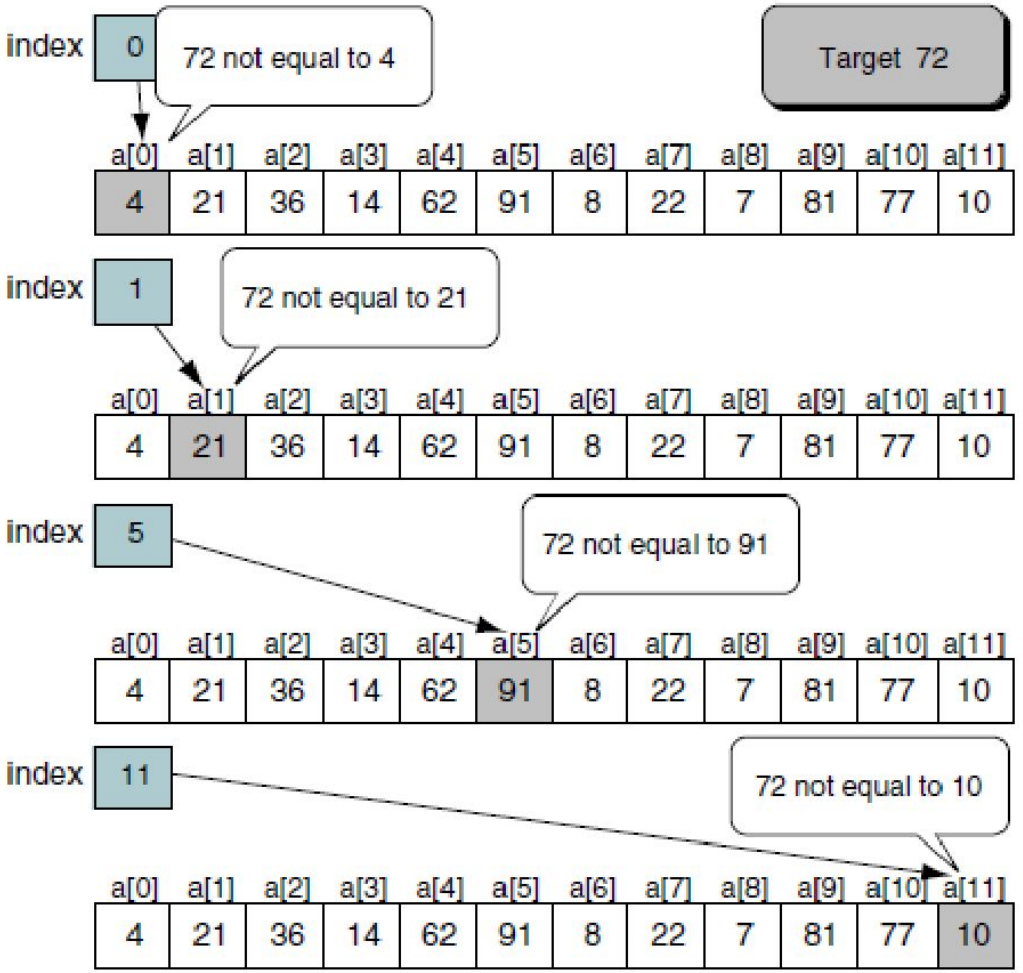


Successful Search of an Unordered List

Busca Sequencial

Dados não-ordenados

- O exemplo ilustra o processo de busca sequencial do elemento com valor 72, em uma lista não ordenada.
- Neste caso, como o elemento não existe, a busca termina sem sucesso, percorrendo todos os elementos da lista.



Unsuccessful Search in Unordered List

Busca Sequencial

Dados ordenados

- O processo de busca pode ser mais eficiente caso os dados estejam ordenados. Neste caso, a busca pode ser concluída sem ter que percorrer, necessariamente, todos os elementos.
- De qualquer forma este tipo de busca é bastante lento, chamado de **busca sequencial**.
- A eficiência deste tipo de busca é $O(n)$, onde n é o número de elementos na lista.

Busca Binária

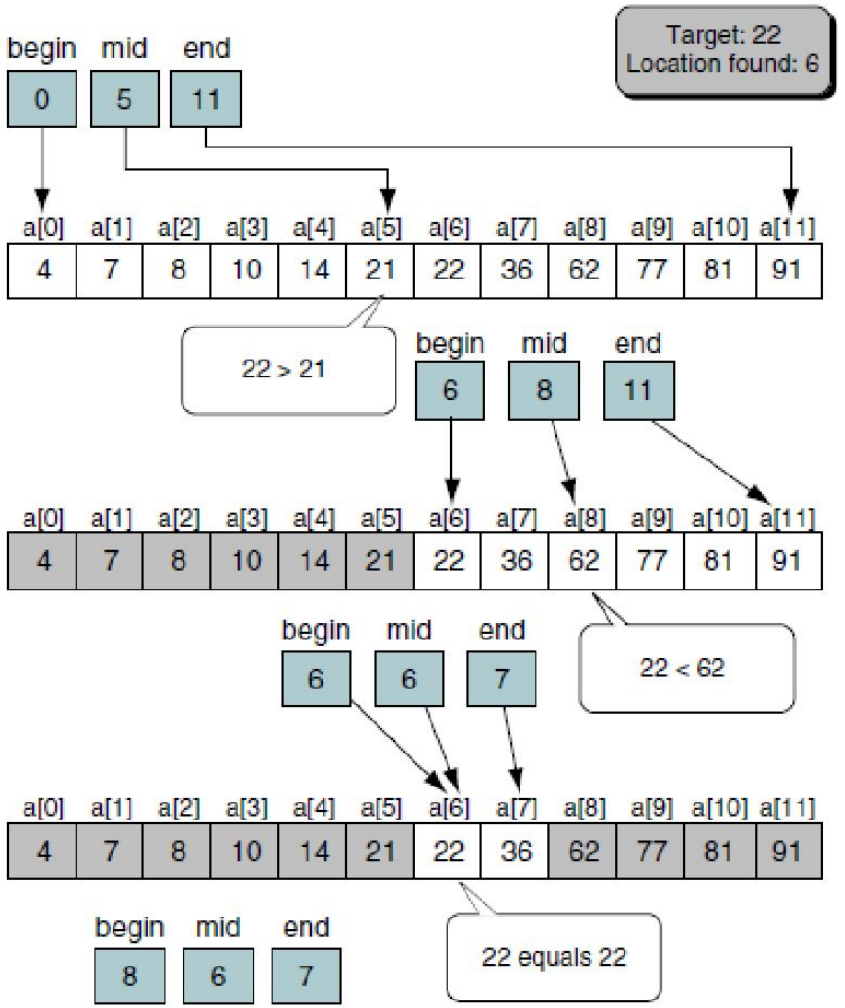
Dados ordenados

- Caso os dados estejam ordenados, um tipo mais eficiente de busca, chamada de **busca binária** pode ser utilizada.
- Neste tipo de busca, a busca começa testando o elemento que se encontra no meio da lista, caso esse elemento não seja o que está sendo procurado, determina-se se o elemento procurado encontra-se na primeira ou na segunda metade da lista. Ao fazer isto, uma metade é descartada reduzindo o espaço de busca pela metade. O processo é repetido até que o elemento procurado seja encontrado ou seja determinado que o elemento não está na lista.
- A eficiência deste tipo de busca é $O(\log n)$, onde n é o número de elementos na lista.

Busca Binária

Dados ordenados

- O exemplo ilustra o processo de busca binária para o elemento com valor 22, em uma lista ordenada.
- Neste caso, como o elemento existe, a busca termina com sucesso.
- Observe em cada passo a definição das posições:
  - Inicio, meio e Fim.

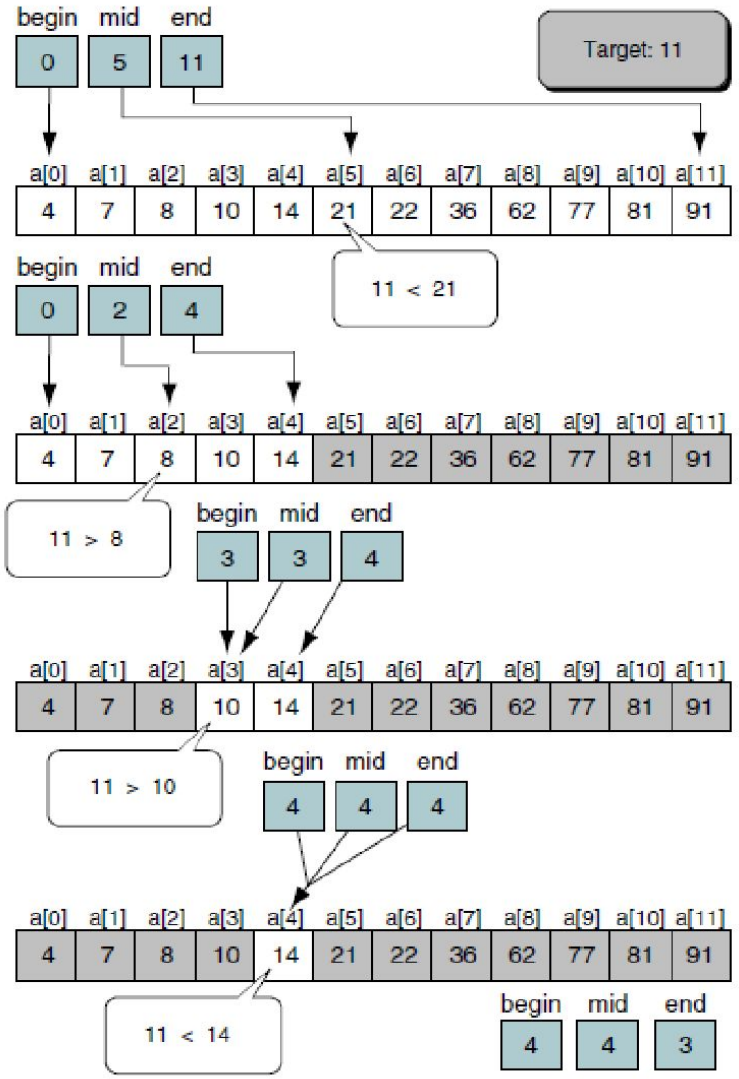


Successful Binary Search Example

Busca Binária

Dados ordenados

- O exemplo ilustra o processo de busca binária para o elemento com valor 11, em uma lista ordenada.
- Neste caso, como o elemento não existe, a busca termina sem sucesso.
- Observe em cada passo a definição das posições:
  - Inicio, meio e Fim.



Unsuccessful Binary Search Example

Busca

Comparação: Busca Sequencial vs Binária

- A tabela mostra uma comparação do esforço realizado por uma busca sequêncial e uma busca binária.

List size	Iterations	
	Binary	Sequential
16	4	16
50	6	50
256	8	256
1000	10	1000
10,000	14	10,000
100,000	17	100,000
1,000,000	20	1,000,000

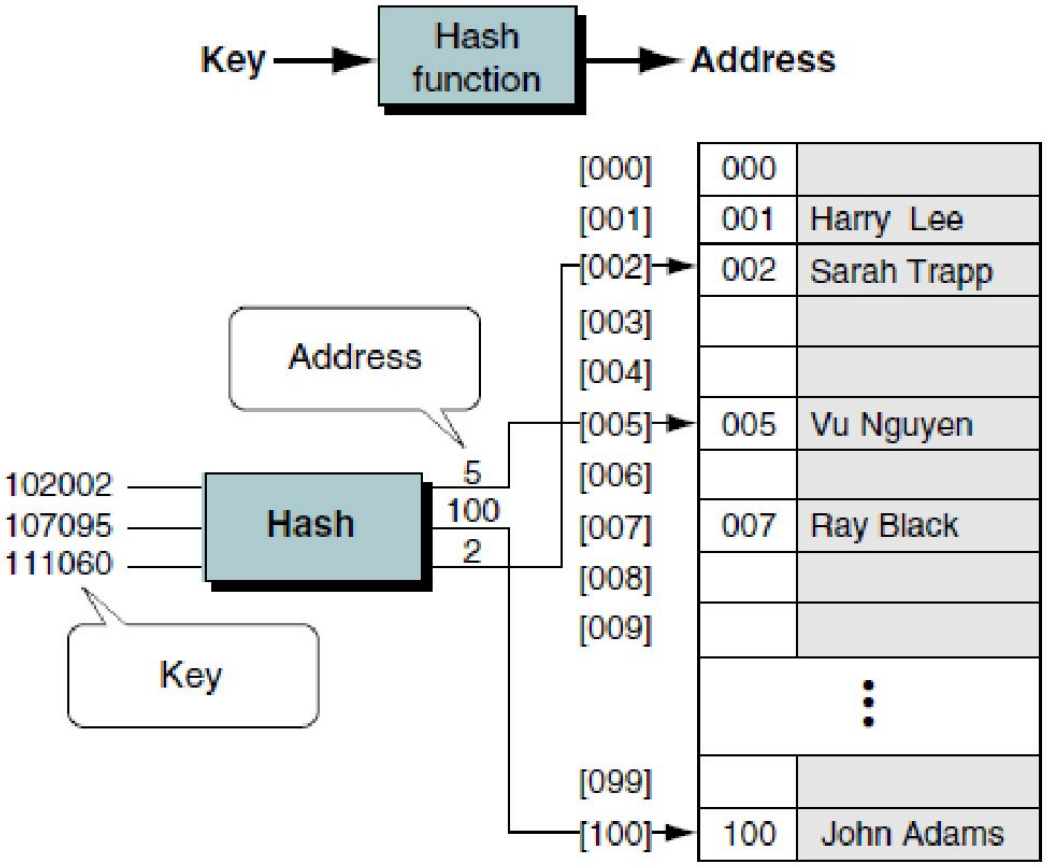
Comparison of Binary and Sequential Searches

Hashing
Definição

- As técnicas de busca em geral realizam realizam diversas comparações de elementos antes de poder encontrar um certo dado.
- A idéia do *Hashing* surge como uma promessa de busca ideal, na qual não é preciso realizar comparações e sabe-se exatamente onde um determinado dado se encontra.
- O objetivo ou ideal de uma busca por *Hashing* (ou *Hashed Search*) é que para encontrar um dado somente realizemos um teste, ou cálculo.
- A promessa de eficiência deste tipo de busca é $O(1)$, onde n é o número de elementos na lista. Embora na prática seja maior do que isso.
- O conceito será ilustrado usando um vetor de dados, embora possa ser adaptado para outras estruturas como sistemas de arquivos armazenados em disco.

Hashing
Definição

- O *hashing* pode ser definido como um processo de transformação de uma chave em um endereço. Trata-se de um processo de mapeamento chave para endereço.
- Uma função de *hash* determina a localização de um dado (ou conjunto de dados) a partir de sua chave.
- O exemplo ilustra o processo de mapeamento de três chaves a três endereços distintos.



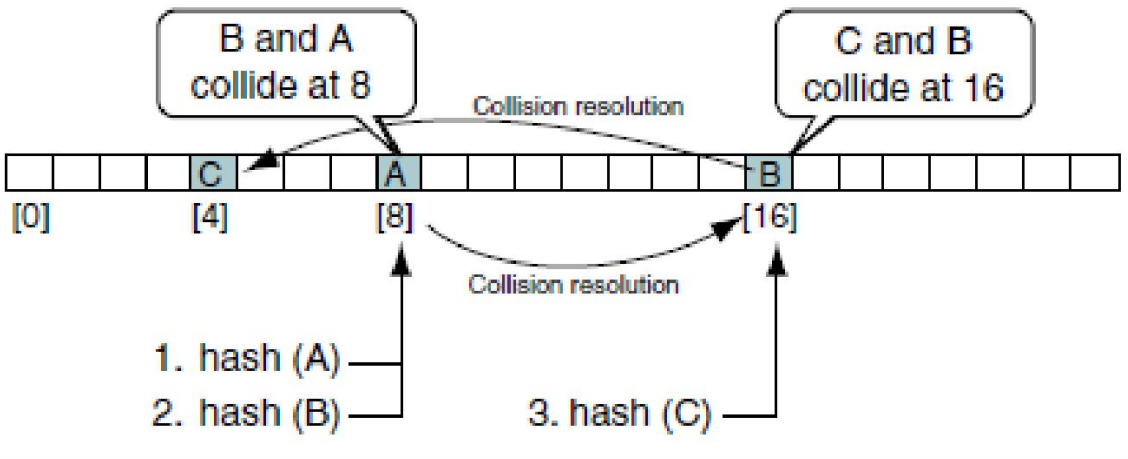
Hash Concept

Hashing
Definição

- Em geral, o número de chaves que podem ser representadas costuma ser muito maior que o espaço de armazenamento ou tamanho da lista de dados.
- Com isso, pode acontecer que várias chaves sejam alocadas na mesma posição da lista.
- As chaves que são mapeadas na mesma posição de uma lista são chamadas de **sinônimos**.

Colisão
Definição

- A figura ilustra o conceito de **colisão**, que acontece quando duas chaves são alocadas ao mesmo endereço de memória. Para resolver o problema da colisão é preciso calcular um novo endereço e verificar que este endereço esteja livre.
- No exemplo, as chaves A e B colidem no endereço 8. A chave A é armazenada no endereço 8, enquanto que calcula-se o endereço 16, para armazenar a chave B.
- Logo depois, temos que a chave C colide no endereço 16. Após um novo cálculo de endereço ela é armazenada no endereço 4.



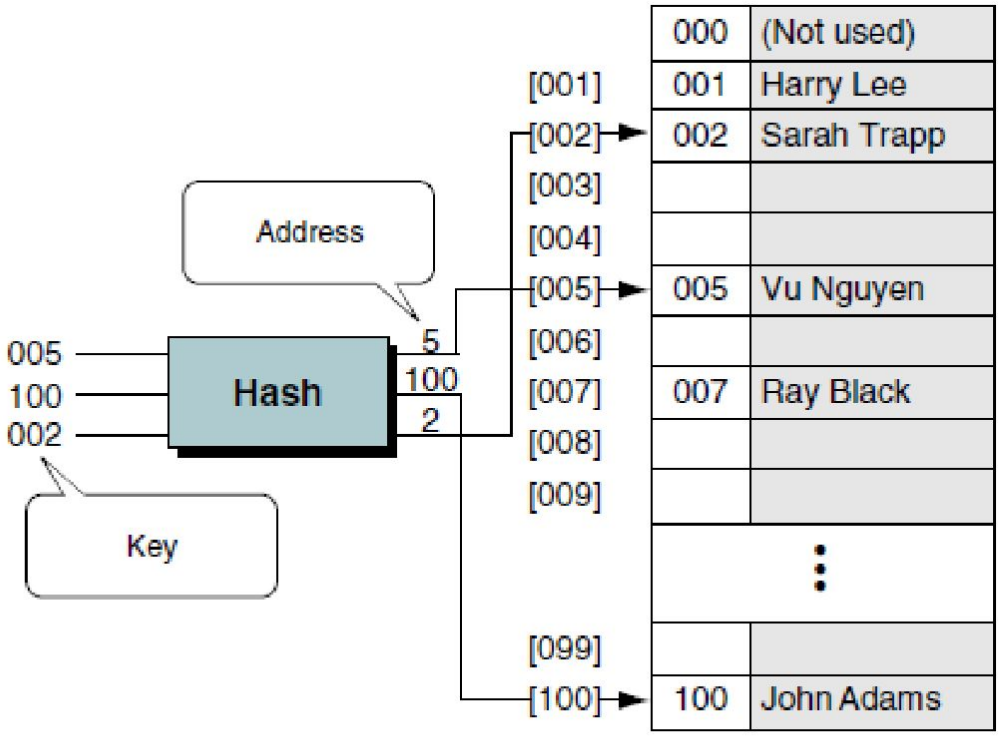
Collision Resolution Concept

Hashing
Métodos

- Existem diversos métodos de hashing que ser utilizados de maneira independente o combinada, entre os mais importantes temos:
  - Método Direto;
  - Método da Subtração;
  - Método do Módulo da Divisão (Resto da Divisão)
  - Método da Extração de Dígitos
  - Método *MidSquare* (Meio do Quadrado)
  - Métodos *Folding* (Métodos de Dobra)
  - Método de Rotação
  - Método *Pseudorandom* (Pseudo Aleatório)

Hashing
Método Direto

- Neste método, existe exatamente uma posição na lista para cada chave. Não havendo necessidade de mapeamento, a própria chave pode servir de endereço.



Direct Hashing of Employee Numbers

Hashing
Método da Subtração

- Semelhante ao método anterior, existe um endereço para cada chave.
- No entanto, as chaves não começam em um, sendo necessário subtrair um valor constante a todas elas.
- Assim, existe sim, uma transformação que converte as chaves em endereços, mas que funciona apenas como uma mudança de escala.

Hashing

Método do Módulo da Divisão

- O método do módulo da divisão, também conhecido como resto da divisão, divide a chave pelo tamanho da lista e usa o resto da divisão como endereço.

```
address = key MODULO listSize
```

- Embora funcione para qualquer tamanho de lista, utilizar um **número primo** como tamanho de lista produz um número menor de colisões.

- Considere por exemplo, a chave 121267 e uma lista de tamanho 307. O novo endereço será 2.

```
121267/307 = 395 with remainder of 2

Therefore: hash(121267) = 2
```

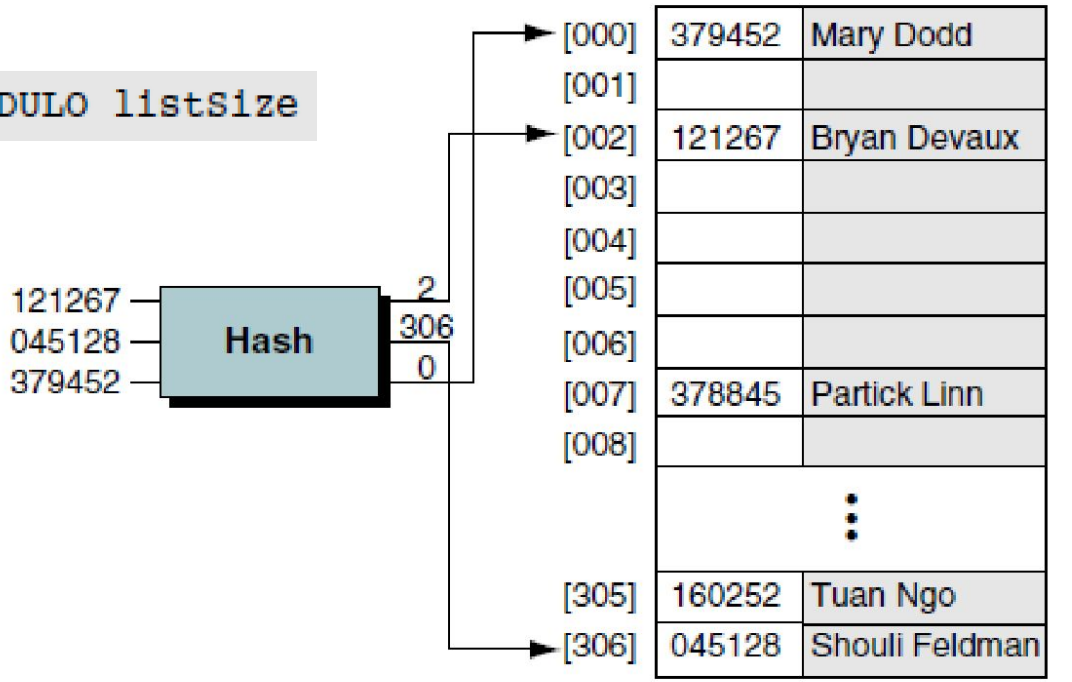
- Este método é um dos mais utilizados, de forma independente ou combinada com outros métodos.

Hashing

Método do Módulo da Divisão

- A figura ilustra a aplicação do método do módulo da divisão para três chaves.

```
address = key MODULO listSize
```



Modulo-division Hashing

Hashing

Método da Extração de Dígitos

- Consiste na extração de dígitos selecionados com base em algum critério e que são usados para formar o endereço.
- No exemplo, seleciona-se o primeiro, terceiro e quarto dígito.

```
379452 ⇨ 394
121267 ⇨ 112
378845 ⇨ 388
160252 ⇨ 102
045128 ⇨ 051
```

Hashing

Método Método MidSquare

- Este método consiste em calcular o quadrado da chave e extrair a parte central para formar o endereço. Como ilustrado no exemplo.

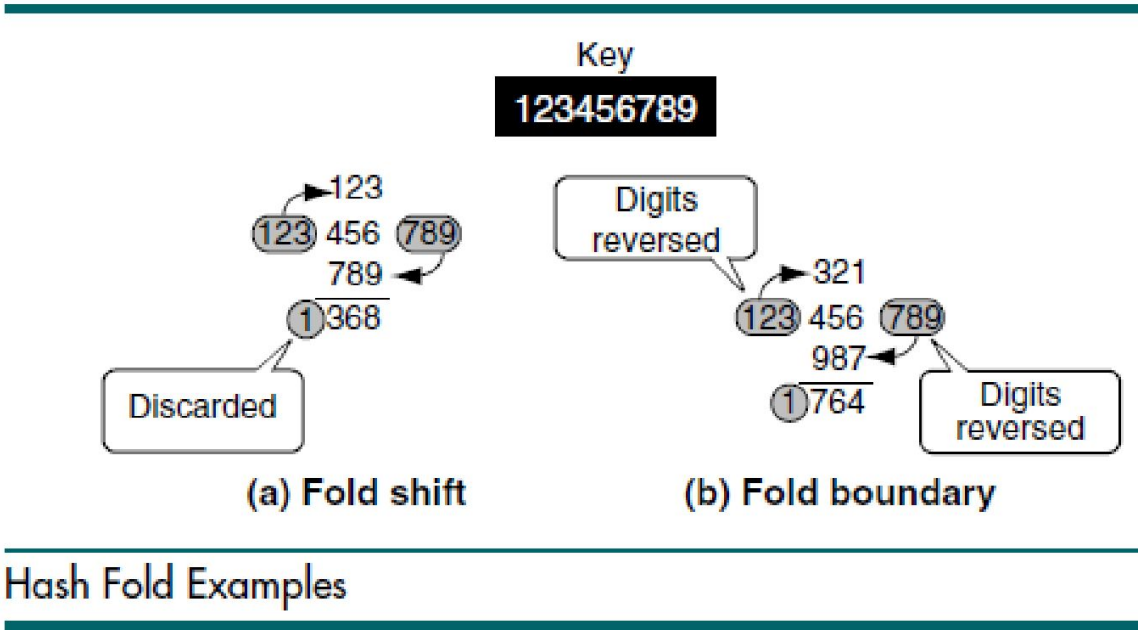
$9452^2 = 89340304$: address is 3403

- Uma deficiência do método é que o quadrado pode ser um número muito grande.
- Uma variante possível consiste em reduzir primeiro o tamanho da chave, extraindo apenas alguns dígitos e somente depois calcular o quadrado do número resultante.
- No exemplo, seleciona-se os três primeiros dígitos da chave. Calcula-se o quadrado. Seleciona-se os dígitos terceiro, quarto e quinto do quadrado como endereço.

379452: $379^2 = 143641 \Rightarrow 364$
121267: $121^2 = 014641 \Rightarrow 464$
378845: $378^2 = 142884 \Rightarrow 288$
160252: $160^2 = 025600 \Rightarrow 560$
045128: $045^2 = 002025 \Rightarrow 202$

Hashing
Métodos Folding

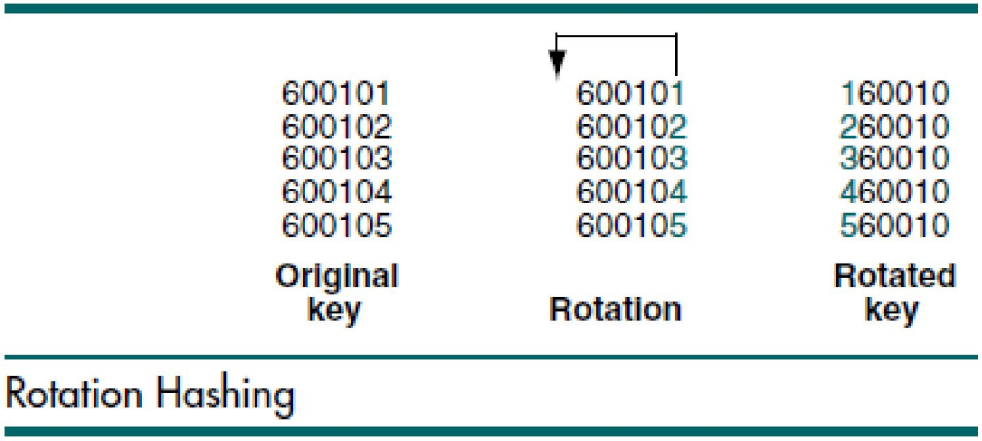
- Os métodos *folding* (ou de dobra) consistem em dividir a chave em partes iguais (completadas com zeros na frente se necessário) e somar as partes descartando dígitos excedentes na frente.
- Na *fold shift*, cada parte preserva a ordem original.
- Na *fold boundary*, partes alternadas são invertidas.



Hash Fold Examples

Hashing
Método de Rotação

- Este método costuma ser utilizado em combinação a outros.
- Costuma ser utilizado quando temos chaves geradas em sequência.
- A rotação consiste em mover o último dígito para a frente do número.



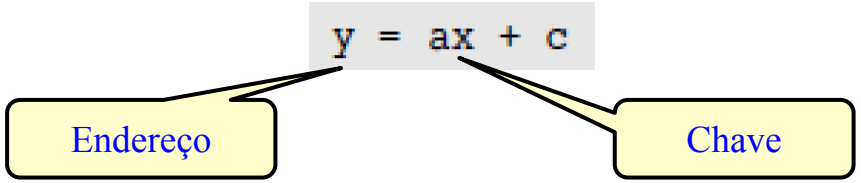
Rotation Hashing

- No exemplo, às chaves rotadas pode-se aplicar o método fold shift, com endereços de 2 dígitos, resultando nos endereços: 26, 36, 46, 56, 66.

- A rotação costuma ser utilizada em combinação com o método *folding* e pseudo aleatório.

Hashing
Método Pseudo Aleatório

- Este método utiliza a chave para gerar um número pseudo aleatório que serve como endereço. Um gerador de números pseudo aleatórios muito simples é:

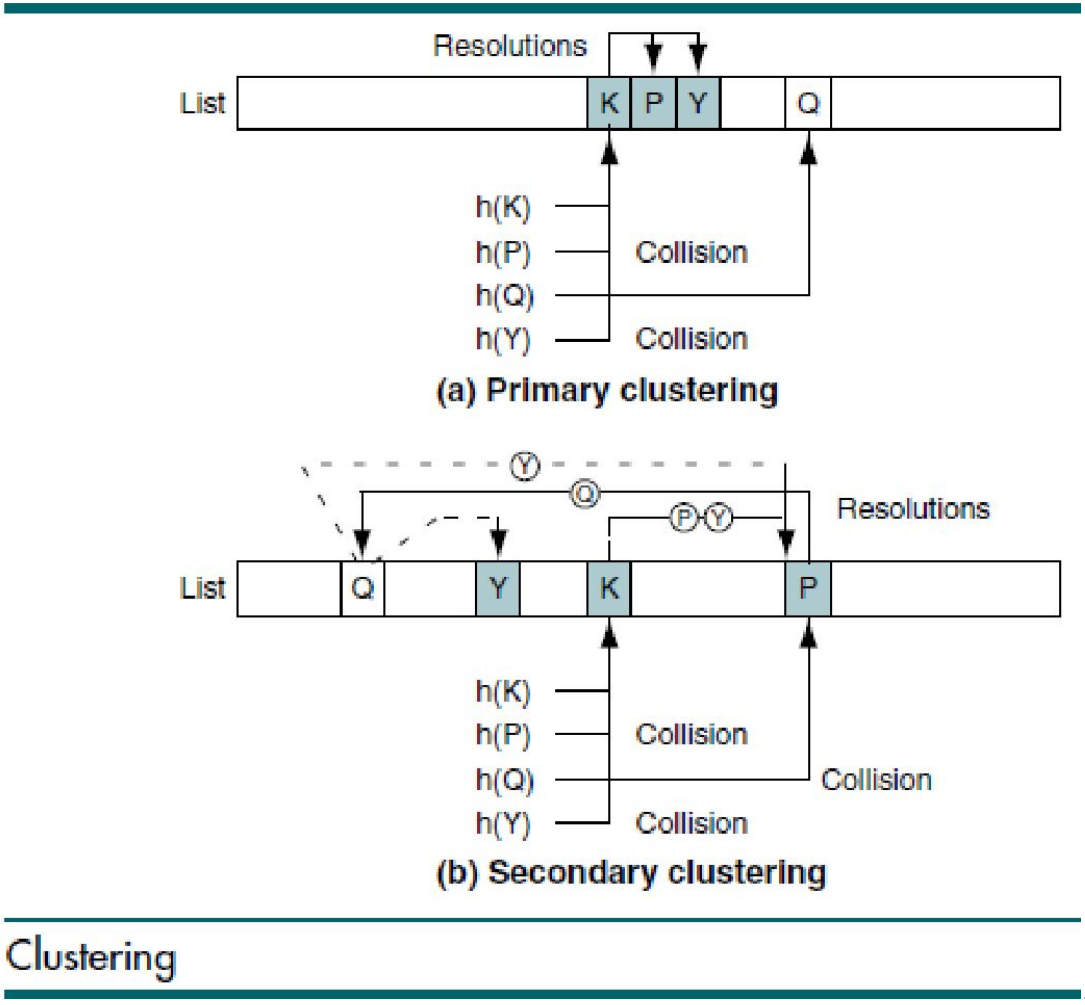


- onde, a e c são constantes que funcionam melhor se escolhidos números primos. O resultado deve ser dividido pelo tamanho da lista e usando o resto da divisão como endereço.
- No exemplo, considere a = 17 e c = 7 e uma lista de tamanho 307.
- A chave 121267 é convertida no endereço 2061546, que depois é escalado usando o resto da divisão com 307.

```
y = ((17 * 121267) + 7) modulo 307
y = (2061539 + 7) modulo 307
y = 2061546 modulo 307
y = 41
```

Hashing
Tipos de Colisões

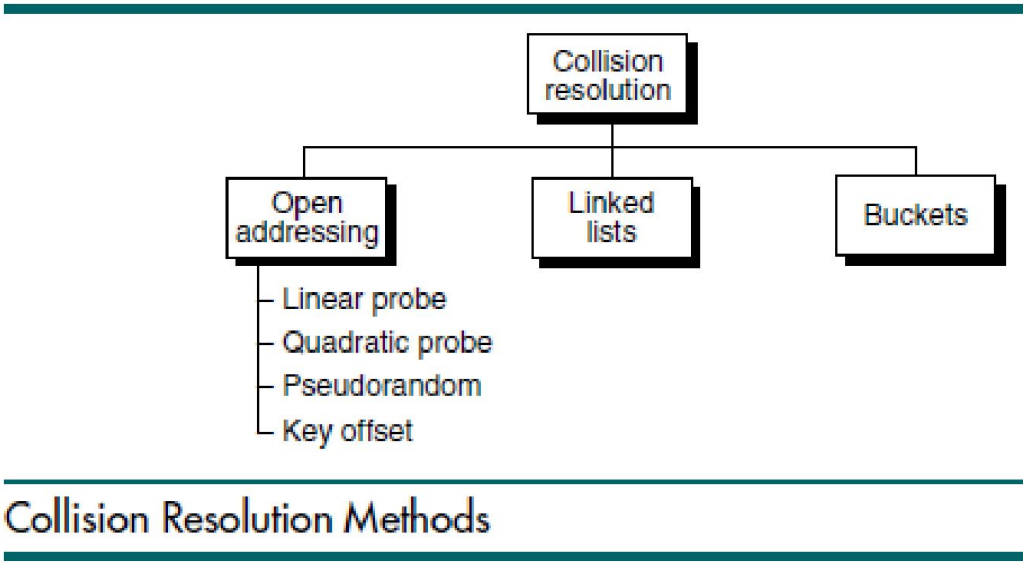
- A figura ilustra os conceitos de clusters primários e secundários.



Clustering

Hashing
Resolução de Colisões

- A figura ilustra os diferentes métodos de resolução de colisões:
  - Endereçamento Aberto;
    - Tentativa Linear
    - Tentativa Quadrática
    - Pseudo Aleatório
    - *Key Offset*
  - Listas encadeadas;
  - Buckets.

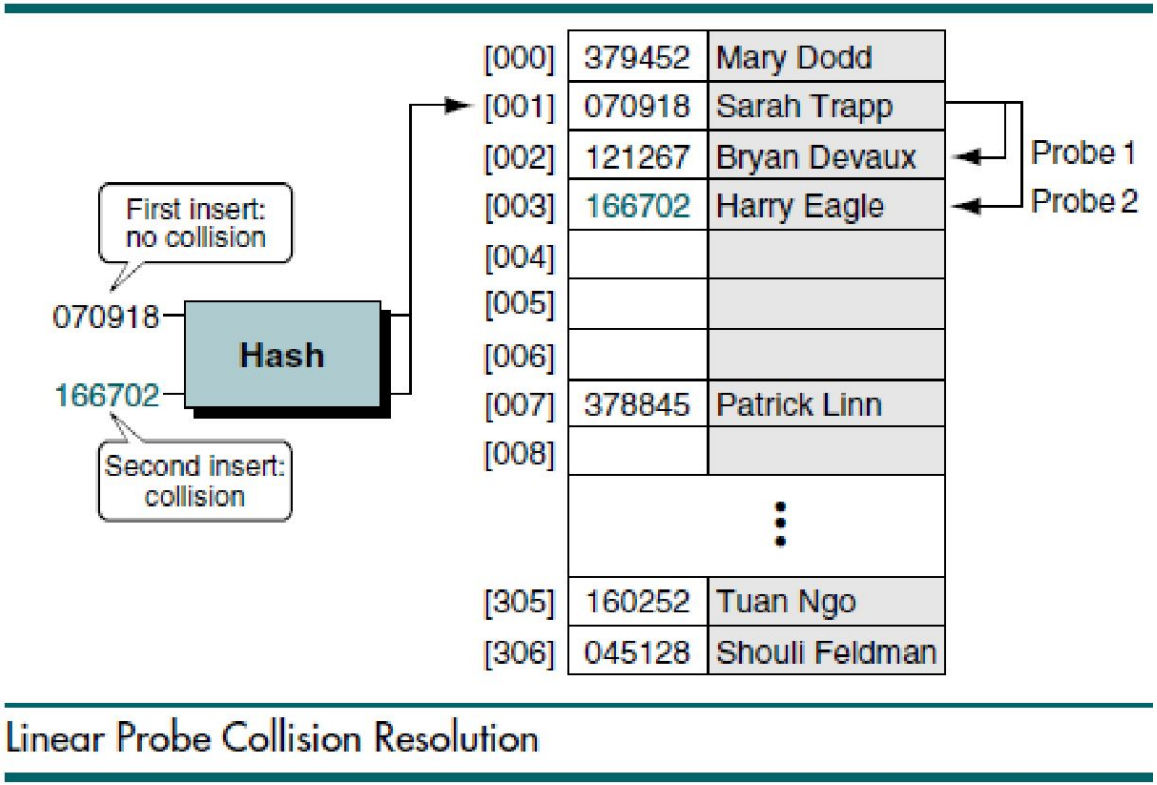


Collision Resolution Methods

Hashing

Endereçamento Aberto: Tentativa Linear

- A figura ilustra a resolução de colisões pela tentativa linear, onde a cada colisão testa-se o seguinte endereço até encontrar um endereço livre.



Linear Probe Collision Resolution

Hashing

Endereçamento Aberto: Tentativa Quadrática

- Na resolução de colisões pela tentativa quadrática, utiliza-se o quadrado da tentativa para utilizar como acréscimo ao endereço anterior.
- Este processo é ilustrado na tabela, que mostra:
  - O número da tentativa, o local da colisão, o tamanho do incremento, e o novo endereço.

Probe number	Collision location	Probe² and increment	New address
1	1	$1^2 = 1$	1 + 1 ⇨ 02
2	2	$2^2 = 4$	2 + 4 ⇨ 06
3	6	$3^2 = 9$	6 + 9 ⇨ 15
4	15	$4^2 = 16$	15 + 16 ⇨ 31
5	31	$5^2 = 25$	31 + 25 ⇨ 56
6	56	$6^2 = 36$	56 + 36 ⇨ 92
7	92	$7^2 = 49$	92 + 49 ⇨ 41
8	41	$8^2 = 64$	41 + 64 ⇨ 05
9	5	$9^2 = 81$	5 + 81 ⇨ 86
10	86	$10^2 = 100$	86 + 100 ⇨ 86

- Existem outras variantes de tentativa linear e quadrática, que adicionam e subtraem aos endereços.

Hashing Duplo

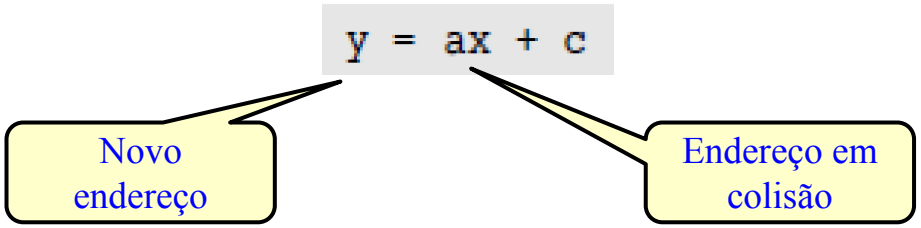
Descrição

- Descreve-se dois métodos de endereçamento aberto que representam um tipo de *hashing* conhecido como ***hashing* duplo.**
- Neste caso, em vez de utilizar uma função aritmética para calcular o próximo endereço em caso de colisão, utiliza-se um nova função *hash* no endereço anterior, e por isso costuma-se dizer que o endereço é ***rehashed***.
- Ambos métodos previnem o ***clustering* primário**.
- Os métodos são os seguintes:
  - Pseudo Aleatório
  - *Key Offset* (Deslocamento)

Hashing Duplo

Endereçamento Aberto: Pseudo Aleatório

- O método é semelhante ao método de *hashing* Pseudo Aleatório.
- Neste caso no entanto ele é utilizado como método de resolução de colisões.
- A diferença se dá pelo fato de, utilizar como variável da função geradora de números Pseudo Aleatórios, um endereço de colisão em vez de uma chave.



- Onde: a e c são constantes escolhidas adequadamente. Por exemplo utilizar um número primo relativamente grande para a como 1663.
- Uma deficiência deste método é que todas as chaves colididas em um mesmo endereço base, seguem o mesmo caminho de resolução de colisões, ou seja geram a mesma sequência de endereços, criando o chamado ***clustering* secundário**.
- Este problema também acontece nas tentativas linear e quadráticas.

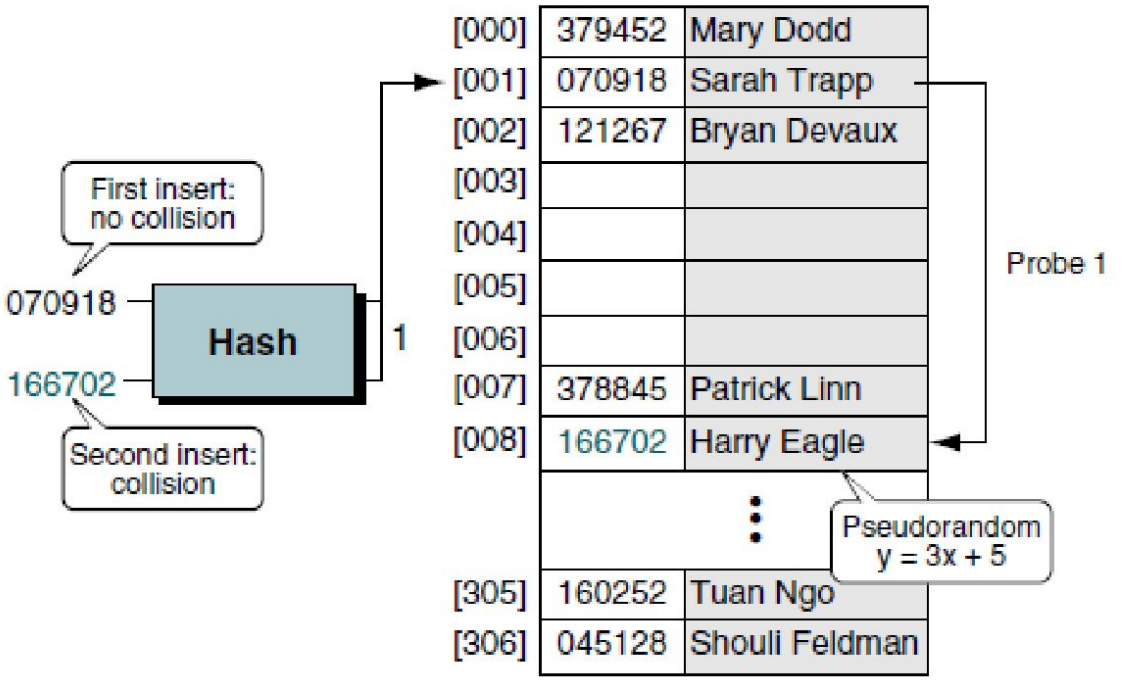
Hashing Duplo

Endereçamento Aberto: Pseudo Aleatório

- No exemplo, considere uma função geradora de números pseudo aleatórios com constantes $a = 3$ e $c = 5$, e uma lista de tamanho $m = 307$. (Constantes pequenas para o exemplo)
- Considere as chaves 070918 e 166702 colidindo no endereço 001. Calcula-se um novo endereço para a segunda chave, a partir do endereço anterior, usando o gerador pseudo aleatório:

```
y = (ax + c) modulo listSize
  = (3 × 1 + 5) Modulo 307
  = 8
```

- Os dados da segunda chave são colocadas no endereço 008.



Pseudorandom Collision Resolution

Hashing Duplo

Endereçamento Aberto: *Key Offset*

- O método ***Key Offset***, ou deslocamento da chave, é um método de **hashing duplo** que produz caminhos de colisão diferentes para chaves diferentes.
- O método calcula um novo endereço com base no valor da chave e do endereço antigo.
- Uma versão simples deste método, calcula o quociente da chave pelo tamanho da lista, que corresponde ao deslocamento (*offset*), e adiciona este valor ao endereço anterior. Para garantir que o endereço seja válido, calcula-se o módulo da divisão inteira.

```
offSet  = ⌊key/listSize⌋
address = ((offSet + old address) modulo listSize)
```

Hashing Duplo

Endereçamento Aberto: *Key Offset*

- Considere como exemplo a chave 070918, e uma lista de tamanho 307.
- Utilizando o método de *hashing* módulo da divisão, o endereço correspondente para esta chave é o 001.
- Considere agora uma segunda chave 166702, que utilizando o mesmo método de *hashing* produz uma colisão no endereço 001.
- Usando o método *Key Offset*, procedemos a calcular o novo endereço:

```
offSet  = ⌊166702/307⌋ = 543
address = ((543 + 001) modulo 307) = 237
```

- Se o novo endereço, 237, também resulta-se em uma colisão, repete-se o processo de calculo de um novo endereço.

```
offSet  = ⌊166702/307⌋ = 543
address = ((543 + 237) modulo 307) = 166
```

Hashing Duplo

Endereçamento Aberto: *KeyOffset*

- Para ilustrar o efeito do método ***Keyoffset***, mostra-se o calculo de vários endereços a partir de chaves diferentes que possuem colisão em um mesmo endereço base.
- A tabela mostra os dois seguintes endereços (*Probe1*, *Probe2*) calculados usando o método *Key Offset* a partir de três chaves distintas que colidem no endereço base 001.
- Observe que o *Key Offset* é diferente em cada caso e que a sequência de endereços gerados ou caminhos de colisão são diferentes.

Key	Home address	Key offset	Probe 1	Probe 2
166702	1	543	237	166
572556	1	1865	024	047
067234	1	219	220	132

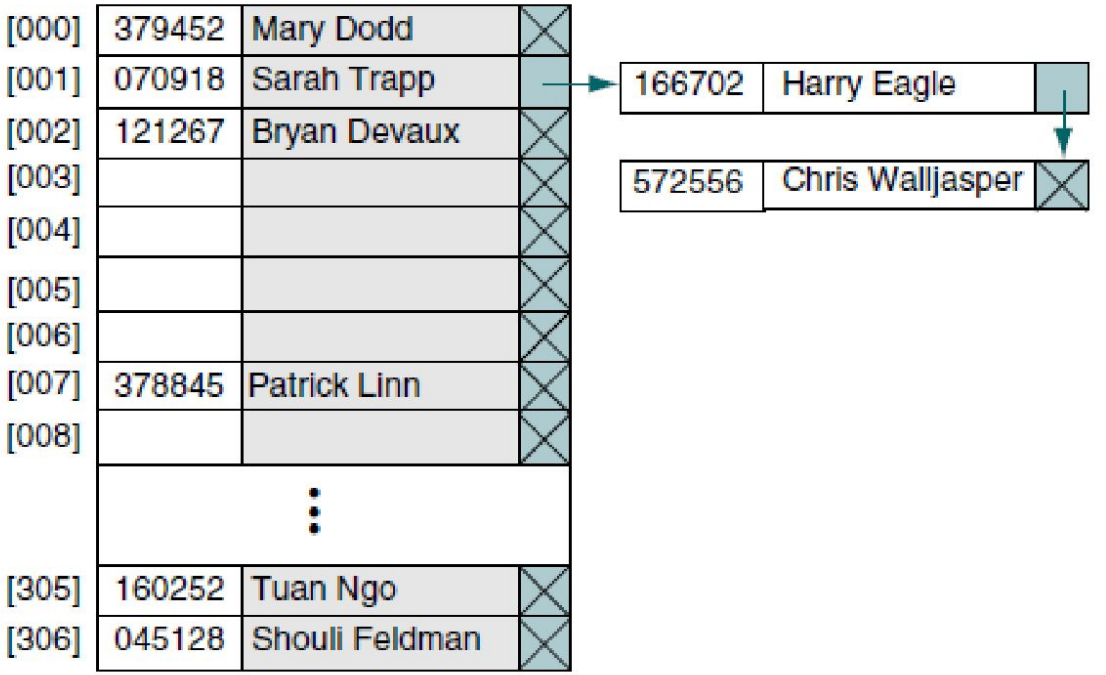
Key-offset Examples

Resolução de Colisões

Lista Encadeada

- A principal desvantagem do endereçamento aberto é que cada resolução de colisão incrementa a probabilidade de novas colisões.
- A abordagem com base em lista encadeada elimina este problema.
- Esta abordagem utiliza uma área externa a **área principal**, chamada de **área de *overflow***, para armazenar os dados colididos. Esses dados sinônimos são encadeados juntos em uma lista encadeada.

- Cada elemento da área principal contêm um campo extra para armazenar um ponteiro a uma lista encadeada, geralmente na memória dinâmica.
- No exemplo, temos três dados sinônimos para o endereço 001.
- O primeiro no array, os outros 2 na lista encadeada.

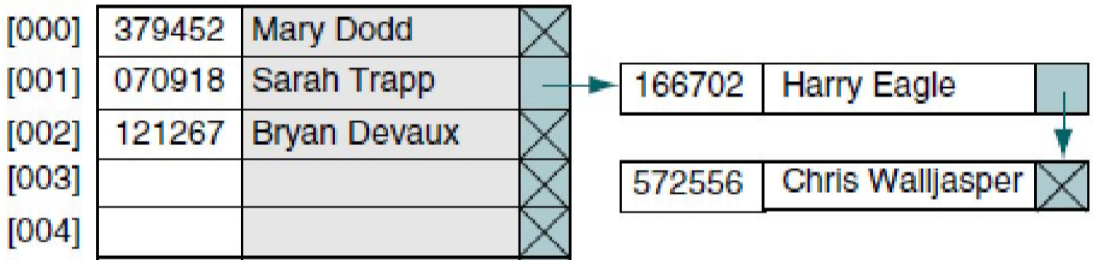


Linked List Collision Resolution

Resolução de Colisões

Lista Encadeada

- Os dados na lista encadeada podem ser armazenados em qualquer ordem.
- No entanto, a ordem LIFO (*Last In, First Out*) ou a sequência ordenada das chaves são as mais usadas.



- A ordem LIFO, é a mais rápida para realizar inserções uma vez que não é preciso percorrer a lista encadeada para realizar a inserção, que acontece sempre no começo da lista.
- A sequência ordenada por chave, com a menor chave na área principal, é mais rápida para realizar buscas e recuperações.

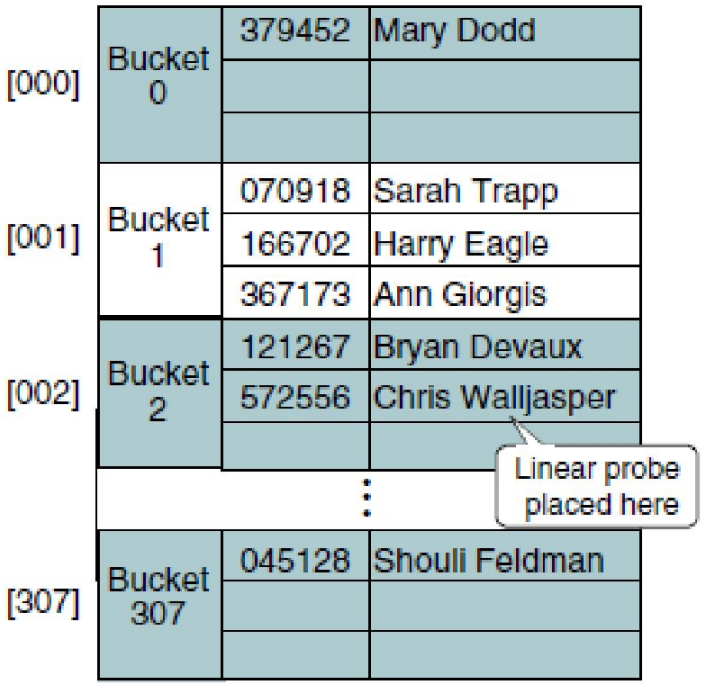
Resolução de Colisões

Bucket Hashing

- Nesta abordagem as chaves são mapeadas em ***buckets***, que são nós que podem acomodar vários dados que possuem o mesmo endereço associado.
- Neste caso, como os nós podem armazenar vários dados, a ocorrência de colisões é adiada até que o nó (o ***bucket***), fique cheio.

- No exemplo, considera-se que cada endereço é capaz de armazenar, dados de três empregados.
- Uma colisão somente acontece quando tentamos adicionar o quarto empregado ao mesmo endereço.

- Esta abordagem utiliza uma maior quantidade de memória, sendo que muitos ***buckets*** estarão vazios ou parcialmente vazios.
- Não resolve o problema de colisão completamente. Em algum momento a colisão acontece.



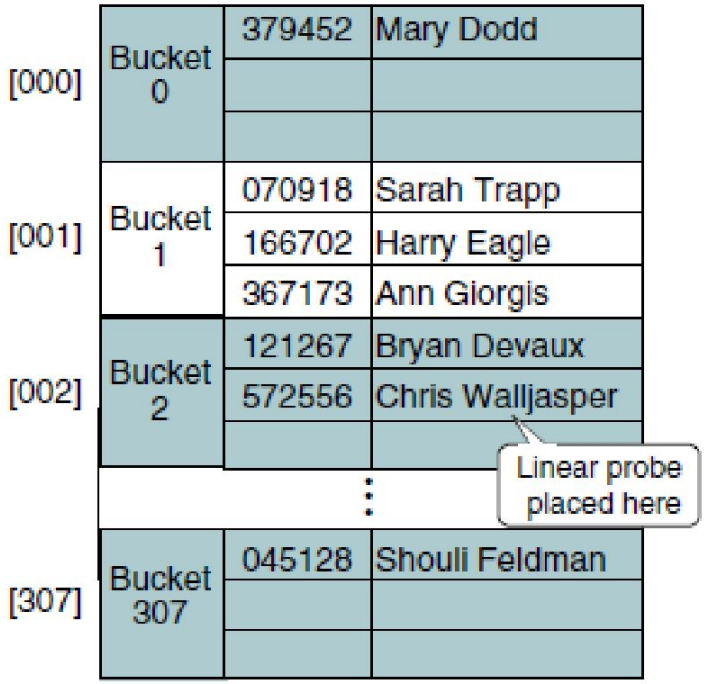
Bucket Hashing

Resolução de Colisões

Bucket Hashing

- Quando a colisão acontece, costuma-se resolver usando o método de tentativa linear. Porém, qualquer método de resolução de colisões pode ser utilizado.
- Os dados em cada ***bucket*** costumam ser armazenados em ordem ascendente de suas chaves para maior eficiência.

- O exemplo mostra três dados alocados no endereço 001. Um dado alocado no endereço 002.
- A colisão acontece quando tentamos inserir o dado com chave 572556, que encontra o *bucket* cheio.
- A colisão é resolvida inserindo o dado no próximo *bucket*.



Bucket Hashing

Resolução de Colisões

Abordagens Combinadas

- Implementações mais sofisticadas podem combinar diferentes métodos de resolução de colisões.
- Por exemplo, utilizar *bucket hashing* inicialmente. Se o *bucket* estiver cheio utilizar um número fixo de tentativas lineares, por exemplo três. E caso isso também falhe utilizar uma lista encadeada como área de *overflow*.

Referências

- Gilberg, R.F. e Forouzan, B. A. Data Structures_A Pseudocode Approach with C. Capítulo 13. Searching. Segunda Edição. Editora Cengage, Thomson Learning, 2005.