Hashing

Cormen Capítulo 11

Motivação

- Um compilador utiliza uma tabela de símbolos para relacionar símbolos aos dados associados.
  - ✓ Símbolos: nomes de variáveis, funções, etc..
  - ✓ Dados associados: localização na memória, gráfico de chamada, etc.
- Uma tabela de símbolos também é chamada de dicionário.

Motivação

- Uma tabela de símbolos utiliza procedimentos para busca, inserção e exclusão.
- A tabela de símbolos não considera ordenação.
- Seja *T* uma tabela e um registro *x* com uma chave (símbolo) e dado. Precisaremos implementar:
  - ✓Insert (*T*, *x*)
  - ✓Delete (*T*, *x*)
  - ✓Search(*T*, *x*)
- Tabela hash será utilizada, onde as operações mencionados ocorrem em O(1).

Tabelas com Endereçamento Direto

- Suponha:
  - ✓ Todas as chaves (keys) são números naturais (grandes) num intervalo 0...*m*-1
  - ✓ As chaves são distintas
- Podemos definir um vetor T[0..m-1] tal que

  if $x \in T$ && key[x] = i

  T[i] = x

  else T[i] = NULL

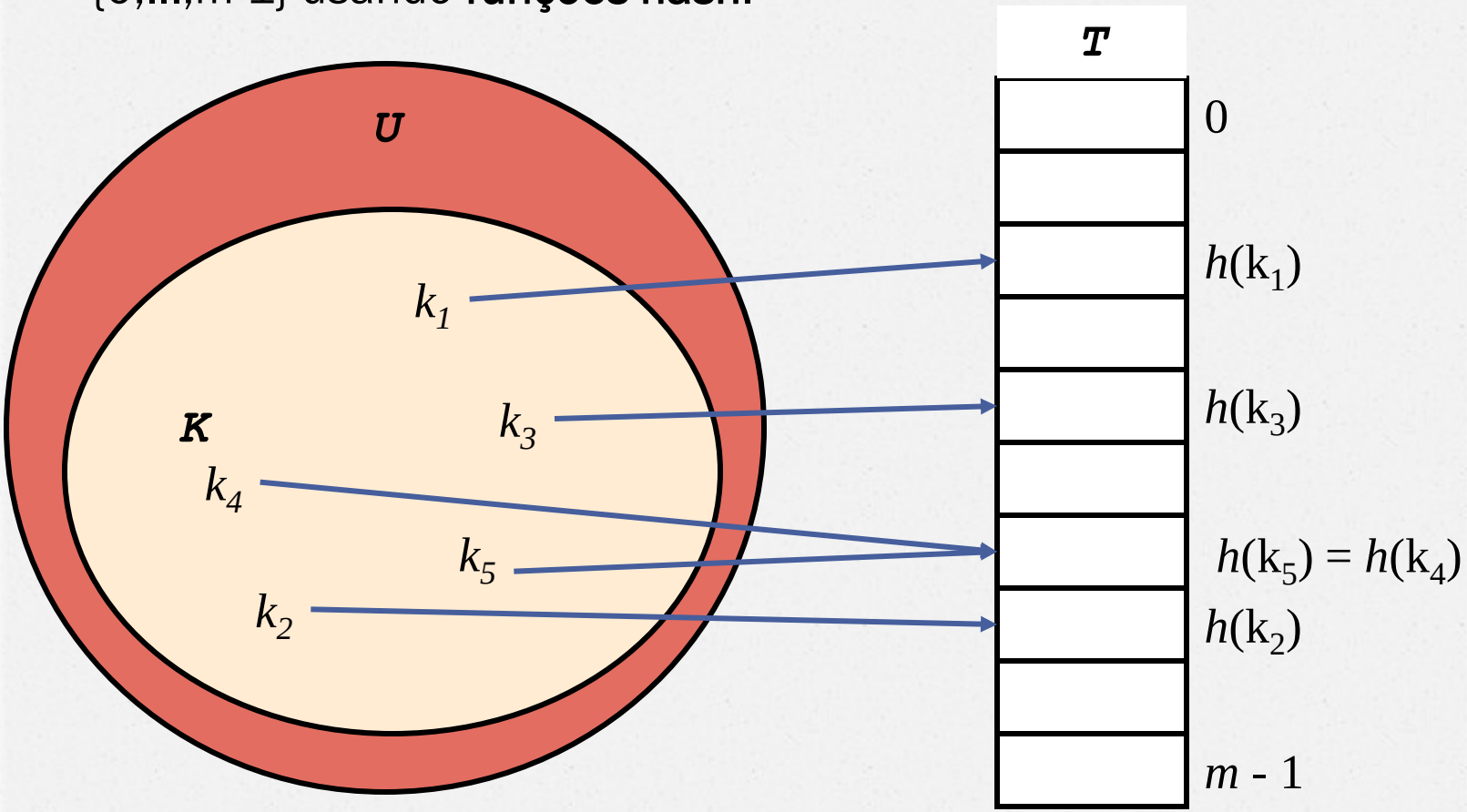
- As operações (inserção, busca, exclusão) levam O(1)

Tabelas com Endereçamento Direto

- Funcionam apropriadamente quando o intervalo das chaves é pequeno.
- Exemplo: Considere chaves que são inteiros de 32-bit.
  - ✓ A tabela de endereçamento direto terá 2^{32} > 4 bilhões de entradas.
  - ✓ Se memória não for um problema, o tempo para inicializar os elementos em NULL pode ser proibitivo.
- Alternativa: Transformação de Chave – Hashing
  - ✓ Mapear chaves em um intervalo menor 0…*m*-1
  - ✓ Colisões irão ocorrer.

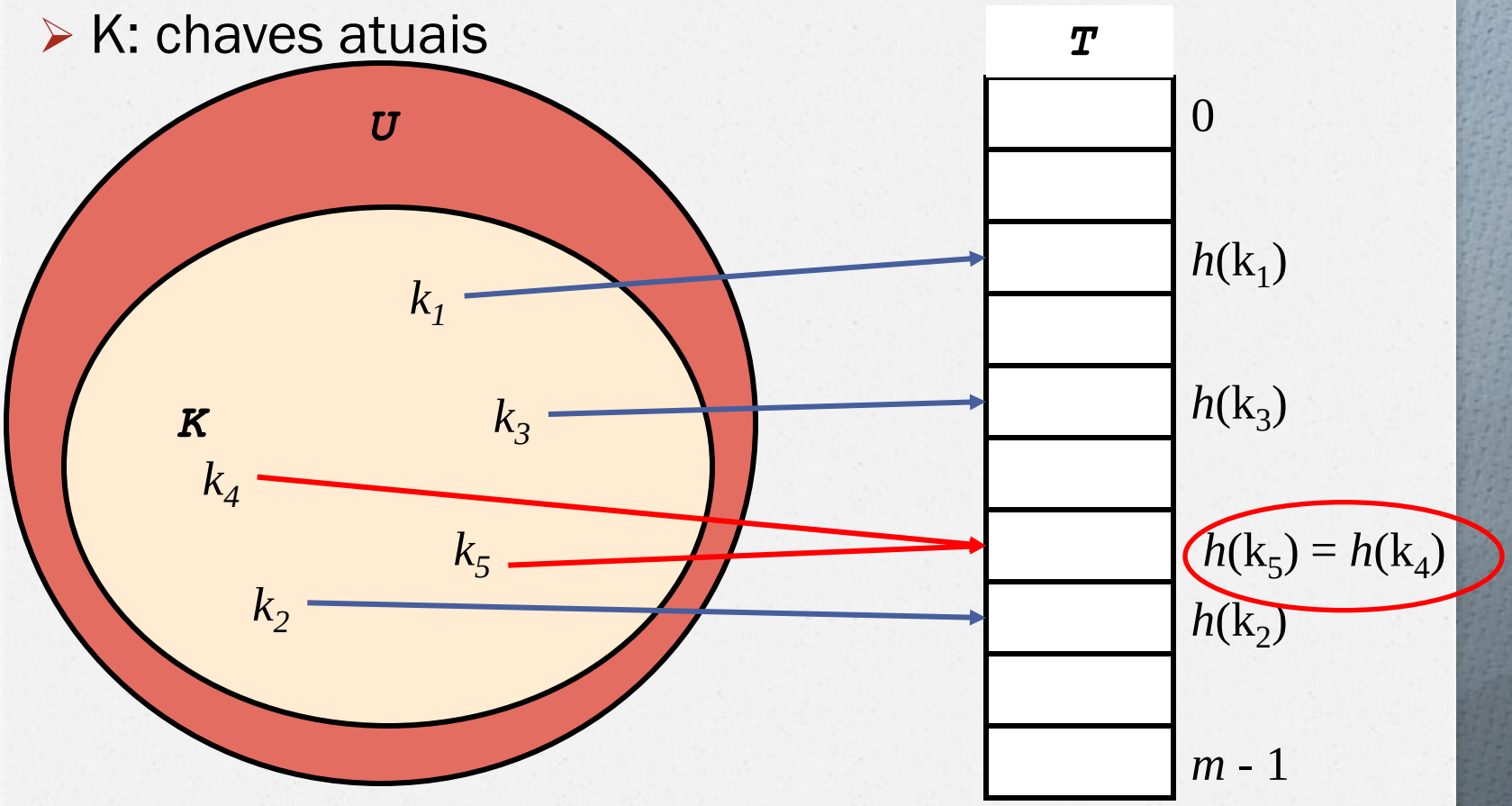
Hashing

- Mapeia o universo de chaves possíveis U para um conjunto {0,...,m-1} usando **funções hash**.



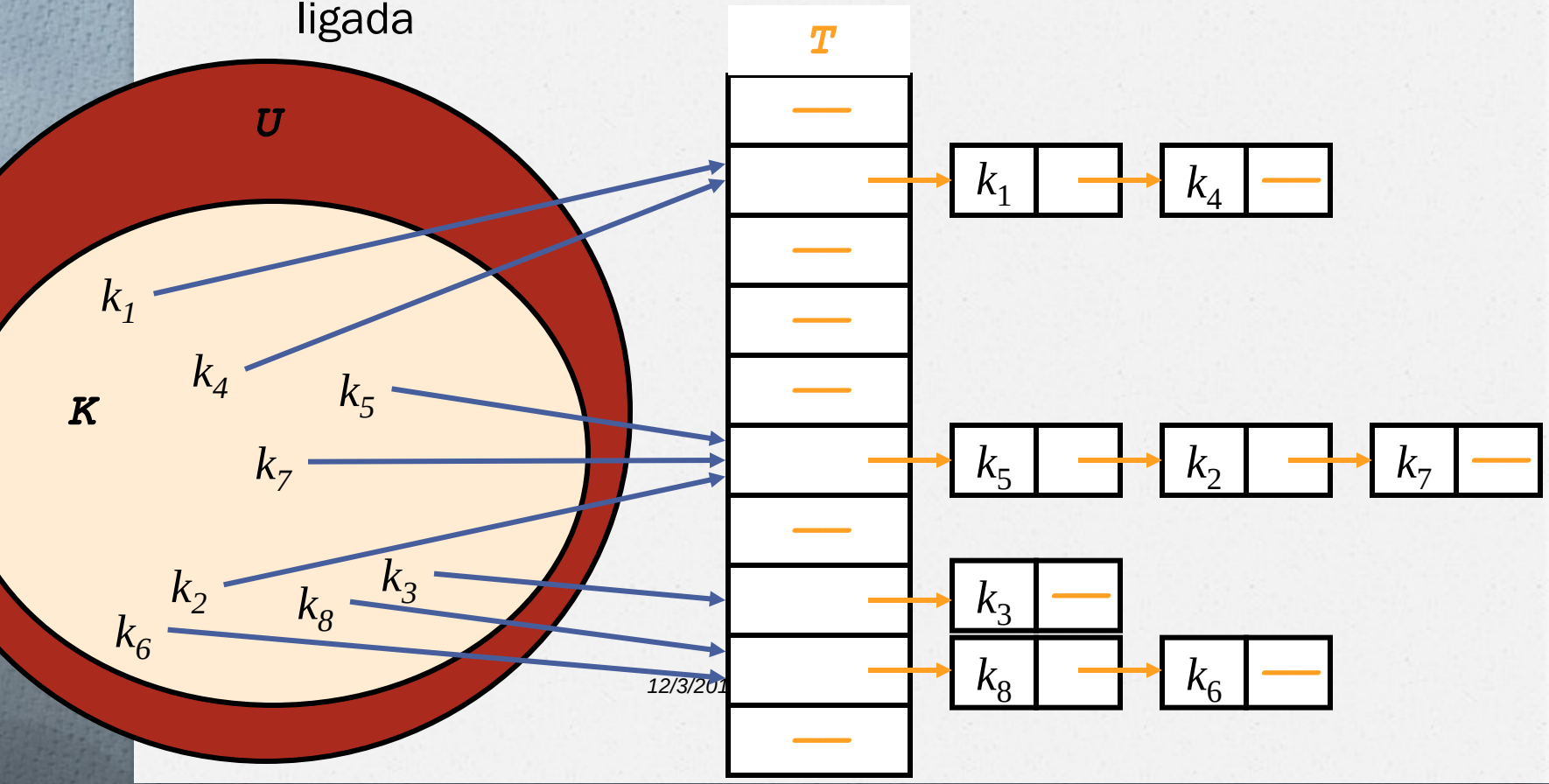
Colisões

- U: Universo de chaves possíveis
- K: chaves atuais



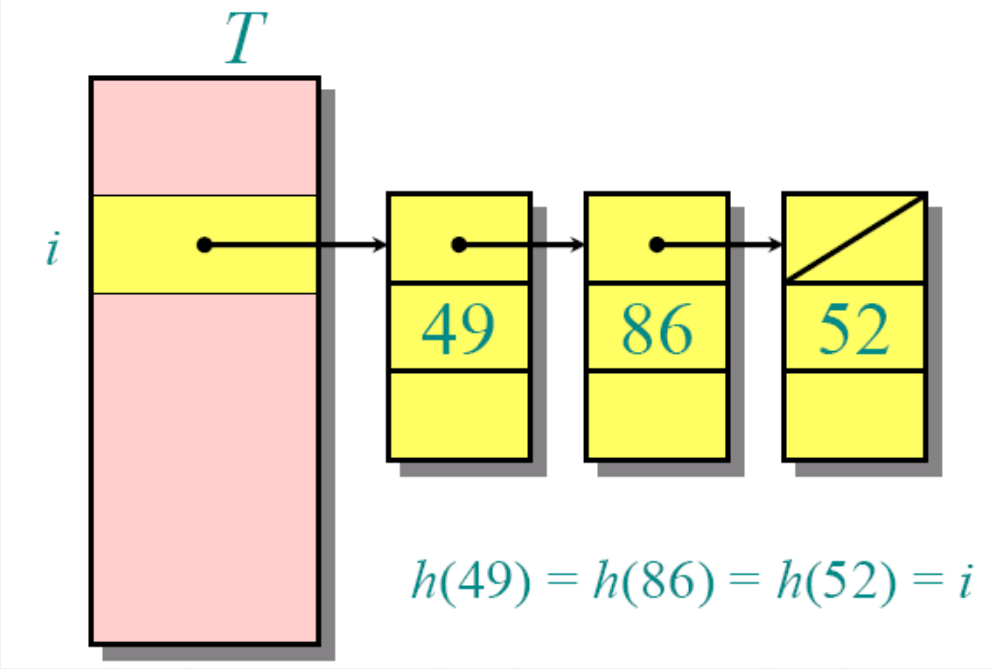
Colisões

- Solução: Chaining
  - Insere elementos atribuídos ao mesmo slot em uma lista ligada



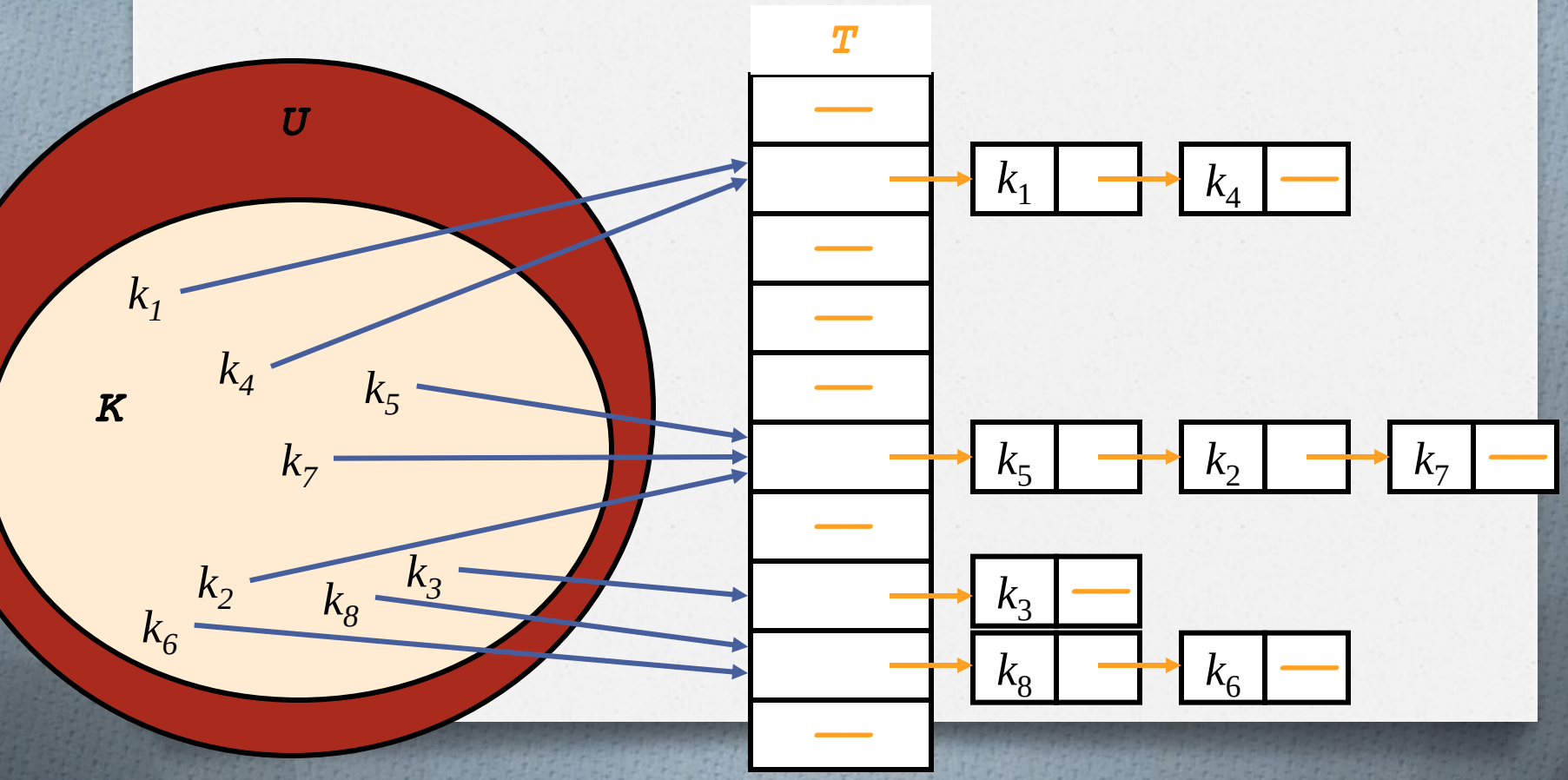
Chaining

- Pior caso ocorre quando toda chave está atribuída ao mesmo slot
- Tempo de acesso Θ(n), se |K| =n



Chaining

- Inserção em O(1), se o elemento não está em T
- Exclusão em O(1), se a lista é duplamente ligada e o elemento é dado.
- Busca com pior caso proporcional ao tamanho da lista.



Chaining

- Analisando
  - Suponha que cada chave é igualmente provável de ser atribuída a qualquer posição na tabela T.
  - Para n chaves e m entradas em T, temos o fator de carregamento (*load factor*) definido como: $\alpha = n/m$: número médio de chaves por entrada em T.
  - O custo médio de uma busca sem sucesso por uma chave k será: $O(1+\alpha)$
    - ✓ O(1): tempo para calcular h(k).
    - ✓ $O(\alpha)$: tempo médio esperado para chegar ao final da lista na entrada T[h(k)].

Chaining

- Analisando
  - O custo médio de uma busca com sucesso por uma chave k será $O(1+\alpha)$

    $O(1+ \alpha/2)=O(1+ \alpha)$
  - Logo, o custo médio de uma busca (com ou sem sucesso) será $O(1+\alpha)$
  - Se o número de chaves n é proporcional ao número de entradas em T , teremos

    $\alpha = n/m = O(m)/m=O(1)$ em média.

Funções Hash

- A escolha de uma função hash adequada se torna fundamental:
  - ✓ Deve distribuir as chaves uniformemente nas entradas de T.
  - ✓ Não deve depender de padrões dos dados.
- Técnicas heurísticas podem ser utilizadas:
  - ✓ Método da divisão
  - ✓ Método da Multiplicação

Método da Divisão

- $h(k) = k \bmod m$
  - ✓ Assuma que todas as chaves são valores inteiros.
  - ✓ O slot (entrada) da tabela é dado pelo valor do resto da divisão da chave k pela quantidade de slots m.

- Exemplo: m=12, k=100

  h(100)=100 mod 12 = 4

Método da Divisão

- Desvantagem 1:
  - ✓ Não escolher um valor de *m* que tenha um pequeno divisor *d.*
  - ✓ Isso levará a uma maior incidência de chaves que são congruente módulo *d* , afetando a uniformidade desejada nas atribuições.
- Desvantagem 2:
  - ✓ Se $m = 2^r$, a hash nem sempre depende de todos os bits da chave *k.*
  - ✓ Se k=1011000111**011010**$_2$ e r = 6, então h(k)=**011010**$_2$

Método da Divisão

- Solução:
  - ✓ Escolher um tamanho de tabela com m= número primo.
  - ✓ Esse primo não pode estar próximo a uma potência de 2.

Método da Divisão

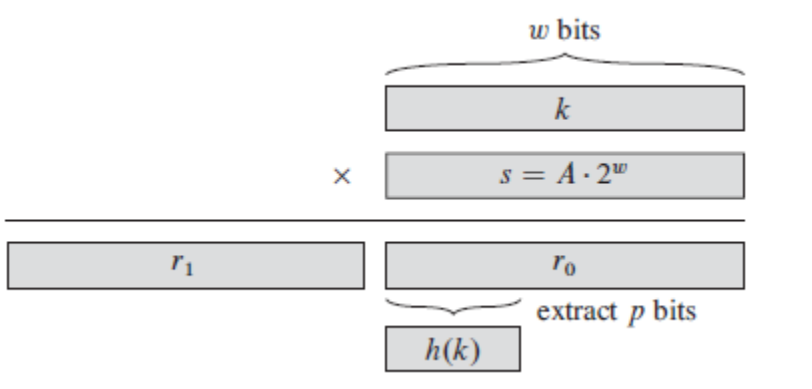
- Exemplo:
  - ✓ Chaves são caracteres com 8 bits num universo n=2000 caracteres (chaves).
  - ✓ Se realizar uma busca sem sucesso em até 3 chaves não for um problema, podemos alocar uma tabela hash de tamanho *m*=701.
  - ✓ *m*=701 é um primo próximo de 2000/3 sem estar próximo de uma potência de 2.
  - ✓ *h(k) = k mod 701.*

Método da Multiplicação

- $h(k) = \lfloor m\,(kA - \lfloor kA \rfloor) \rfloor$
  - $0<A<1$ constante
  - A não deve estar próximo de 0 ou 1
  - h(k): parte fracionária de *kA*
  - Pode escolher $m=2^p$ para p um valor inteiro.
  - Sugestão: $A = = (\sqrt{5} - 1)/2 \cong 0.618033\ldots$

Método da Multiplicação

- Suponha uma representação (palavra) da máquina com w bits e k representa uma única palavra.
  - $A=s/2^w$ para s inteiro com $0<s<2^w$.
  - Temos: $k.s=r_1*2^w + r_0$
  - O valor h(k) é dado pelos p bits mais significativos em r_0



Método da Multiplicação

- Exemplo: m=8=2^3 e w=7 (palavra com 7 bits)

1101011=107

1011001=88=A

1101011

0000000

0000000

1101011

1101011

0000000

1101011 .

10010100110011

Método da Multiplicação

- Exemplo: m=8=2^3 e w=7 (palavra com 7 bits)

1101011=107

1011001=88=A

1001010**011**0011

h(k)=011=3

Método da Multiplicação

- Exemplo2: m=10000, k=123456 e

A= $(\sqrt{5} - 1)/2 \cong 0.618033...$

$h(k)=\lfloor m\,(kA - \lfloor kA \rfloor)\rfloor$

$=\lfloor 10000.(123456*0.618033-\lfloor 123456*0.618033\rfloor)\rfloor$

$=\lfloor 10000.0.0041151....\rfloor$

$= 41$

Universal Hashing

- Problema:
  - ✓ Suponha que seu código esteja sendo avaliado sempre no pior caso.
  - ✓ Logo, a função hash adotada não poderá evitar uma performance O(n) na busca.
- Solução:
  - ✓ Universal hashing: escolha uma função hash aleatoriamente, ou seja, independente das chaves que serão armazenadas.
  - ✓ Isso garante uma boa performance média.