* Hash (Cormen)

→ muitos aplicações exigem conjunto dinâmico que suporte somente as operações de "dicionário":

- Insert, search, delete

Ex: Compilador → tabela de símbolos

- Hash (tabelas de espalhamento) são estruturas eficazes para implementar dicionários

No prática, pior caso O(n) para inserção/remoção, consulta, mas funciona extremamente bem.

Sob premissas razoáveis, tempo médio de consulta é O(1)

→ generaliza a noção de array, com endereçamento direto

- ao invés de usar chave diretamente, o índice é computado a partir da chave

- Conclusão: Hash é eficaz e prática: as operações básicas de dicionário requerem apenas O(1) em média

(01) Tabelas de Endereçamento Direto

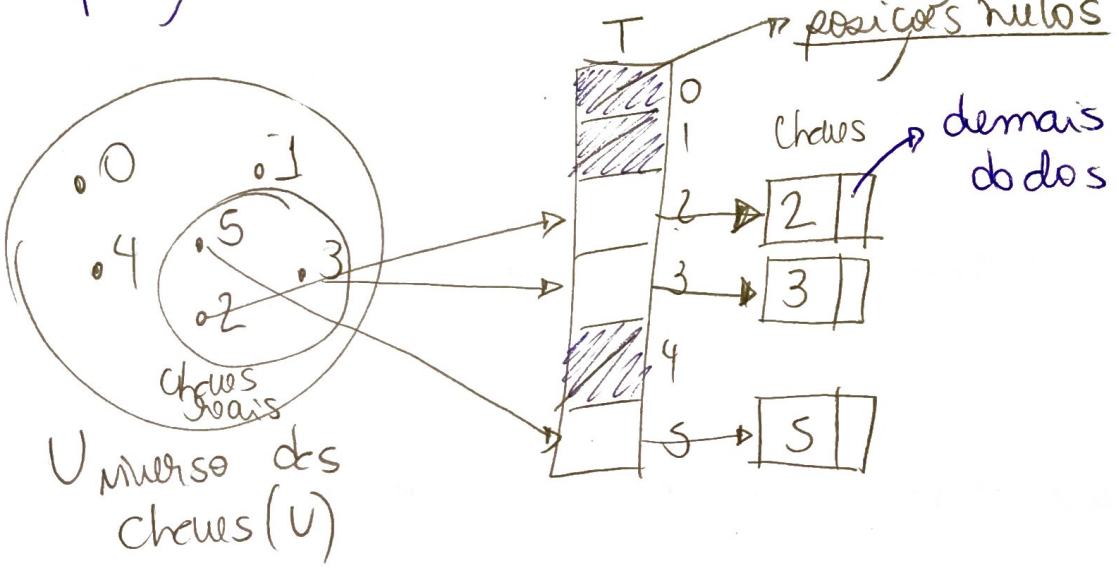
→ simples, funciona quando o universo de chaves (U) é razoavelmente pequeno.

Ex:

aplicação necessita de um conjunto dinâmico com
m chaves de $U = \{0, 1, \ldots, m-1\}$
onde m não é muito grande

conjunto (arranjo) ⇒ tabela de endereços diretos
$T[0 \ldots m-1]$

- posição / locação: corresponde a uma chave



- posição k aponta para elemento com chave k
- se o conjunto em k é vazio, $T[k] = NULL$

* Operações

→ Inserção (T, x)
$T[x.chave] = x$

→ Acesso (T, k)
return $(T[k])$

→ Remove (T, x)
$T[x.chave] = NULL$

* tempo das operações = $O(1)$

Exercícios p/ Casa

(01) Conjunto S feito por uma tabela T[0... m-1]. Fazer uma função para computar o elemento máximo de S. Qual o desempenho pior caso?

*(02) Tabelas de Espalhamento

desvantagem do espalhamento direto:
- se U é muito grande, tabela de tamanho |U| consome muito memória
- os chaves realmente armazenadas < chaves totais
  - muitos posições nulos, espaço desperdiçado

* Requer uma tabela muito menor que o número total de chaves

Endereço direto : $k \rightarrow T[k]$

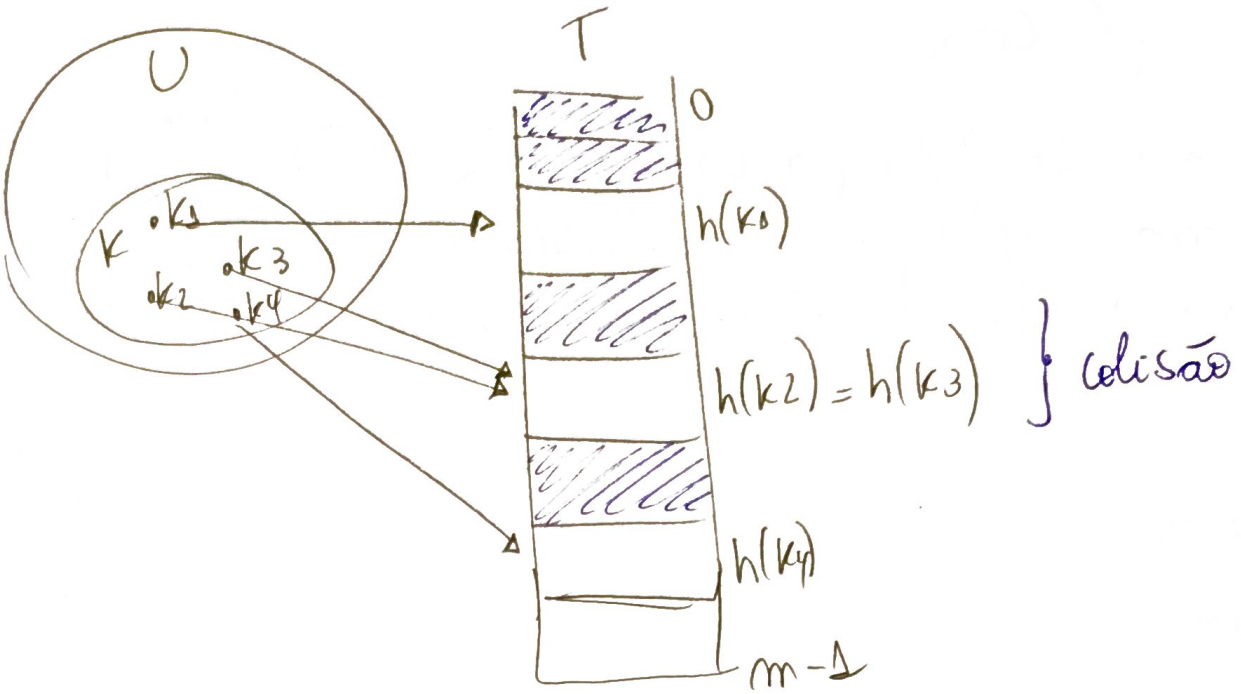
Hash : $k \rightarrow T[h(k)]$

$h(k)$ → função hash para calcular a posição de k

$$h: U \rightarrow \{0, 1, \ldots, m-1\}$$

"um elemento com a chave k se espalha até a posição $h(k)$"

"$h(k)$ é o valor hash de k"



dos:

- função hash reduz a faixa de índices do arranjo, e o tamanho do arranjo
- Pode haver um revés: duas ou mais chaves mapeadas para a mesma posição (colisão)

Ideal: evitar por completo as colisões

- função h adequada, "aleatória"

h deve ser determinística, k sempre produz h(k)

Como $|U| > m$, deve haver chaves que produzam mesmo valor hash.

- Precisamos: meios de tratar colisões

técnica + simples: resolução de colisões por encadeamento.

- colisões:
  - endereçamento aberto
  - encadeamento

(01) Endereçamento Aberto

- quando uma chave colide com outra, a colisão é resolvida encontrando-se uma entrada diferente, e disponível

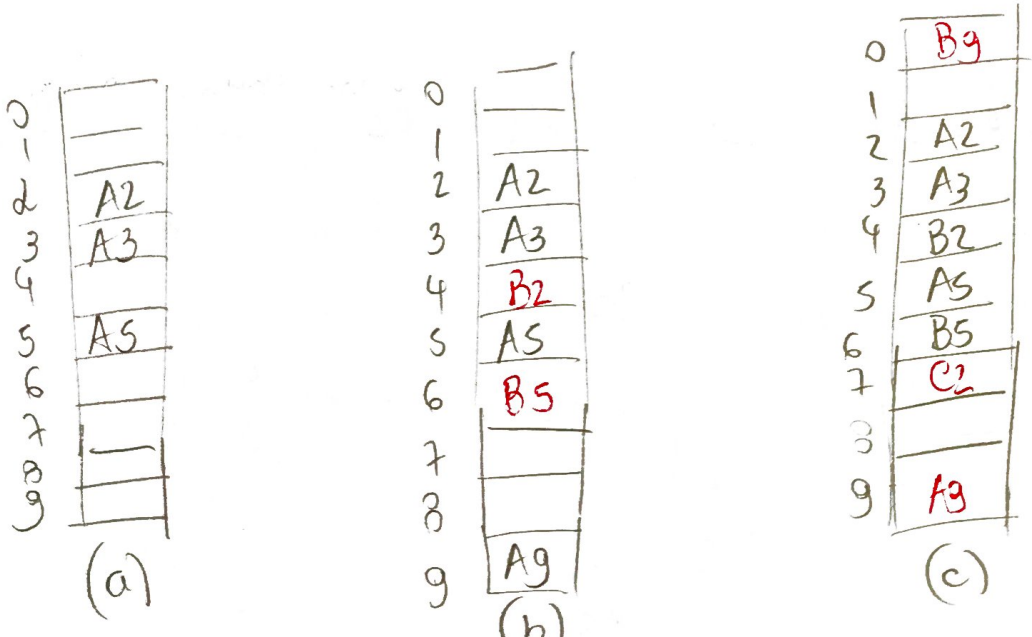
Se $h(k)$ está ocupada:

- verifica $h(k)+1$, $h(k)+2$, $h(k)+3$ ...

Até que ache uma posição disponível, ou se chegue até a mesma posição já inicialmente tentada (tabela cheia)

- "sondagem linear"

EX: $A_2, A_3, A_5, B_2, B_5, A_9, C_2, B_9$



* desvantagem: gera "agrupamentos", dados não ficam espalhados

* Uma alternativa:

sondagem linear ⇒ sondagem diferente (quadrática)

$$p(i) = h(k) + (-1)^{i-1}\left(\left((i+1)/2\right)^2\right), \quad \text{para } i = 1, 2, \ldots, M-1$$

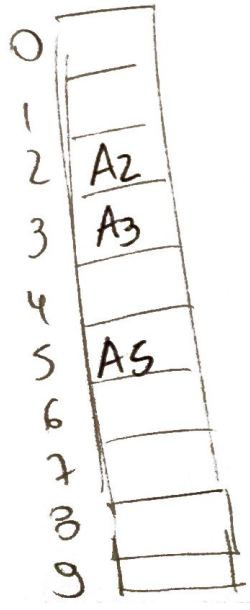
- sequência de sondas:

$$h(k) + i^2, \; h(k) - i^2 \quad \text{para } i = 1, 2, \ldots (M-1)/2$$

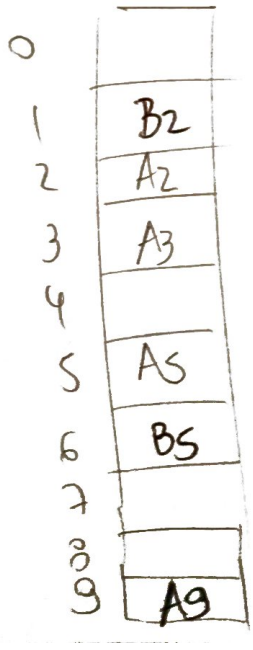
Por exemplo:

$$h(k), \; h(k)+1, \; h(k)-1, \; h(k)+4, \; h(k)-4, \ldots, \; h(k) + (M-1)^2/4, \; h(k) - (M-1)^2/4.$$

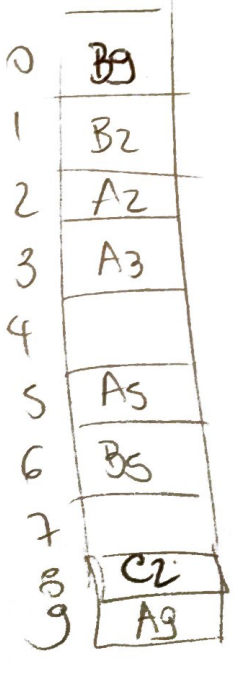
Inserir: A_5, A_2, A_3



Inserir: B_5, A_9, B_2

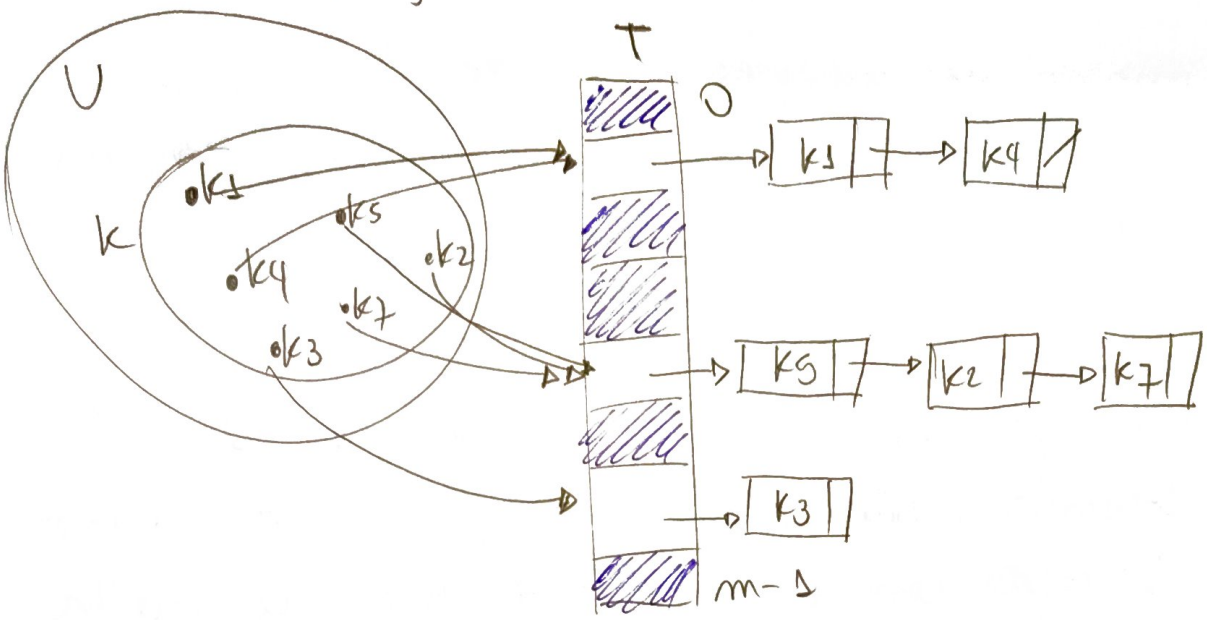


Inserir: B_9, C_2



→ resolução por encadeamento:

* todos os elementos resultantes do hash vão para a mesma posição em uma lista ligada



*Operações

CHAINED-HASH-INSERT (T, x)
  insere x no início da lista T[h(x.chave)]

CHAINED-HASH-SEARCH (T, k)
  procura um elemento com chave k na lista T[h(k)]

CHAINED-Hash-delete (T, x)
  elimina x da lista T[h(x.chave)]

(03) Funções Hash

- divisão
- multiplicação } heurísticos
- hash universal, usa aleatorização

* uma boa $h(x)$ satisfaz a premissa do hashing uniforme:

→ cada chave tem igual probabilidade de passar para qualquer das m posições.

→ não se pode verificar isso na prática, pois não sabemos a distribuição de probabilidades das chaves

* na prática usa-se heurísticas

heurística: processos cognitivos usados em decisões não-racionais, são estratégias que ignoram parte da informação com o objetivo de tornar a escolha mais fácil e rápida.

(A) Método de divisão

- usa o resto da divisão de k por m

$$h(k) = k \bmod m$$

EX:

tabela de tamanho $m = 12$, $k = 100$ } operação rápida

$h(k) = 4$

- Deve-se evitar certos valores de m:
  - m não pode ser potência de 2, $m = 2^p$
- Usar um número primo

(B) Método da multiplicação

- cria função hash em dues etapos

  (1) primeiro multiplica k por uma constante A, $0 < A < 1$ (kA)

  (2) multiplica por m e toma o piso do resultado

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Vantagem: o valor de m não é crítico

Literatura sugere $A \simeq 0{,}618$

→ pegar a parte inteira de $h(k)$

(C) Hash Universal

- escolher a função hash aleatoriamente, independente das chaves armazenadas

- Seja H uma coleção finita de funções Hash. Dizemos que ela é universal, se, para cada par de chaves distintos k e l, o número de funções hash $h \in H$ para os quais $h(k) = h(l)$ é no máximo $|H|/m$. Ou seja, não pode ser maior que $1/m$.

- classe universal de funções hash:

No execução:

- seleciona h de H aleatoriamente

- algoritmo pode se comportar de forma diferente a cada execução

Projeto:

- escolher número primo p grande para que toda chave k esteja entre 0 a $p-1$.

$p > m$

- define-se funções hash h_{ab} para qualquer $a \in \mathbb{Z}_p^*$ e $b \in \mathbb{Z}_p$, com $\mathbb{Z}_p = \{0, \ldots, p-1\}$

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m)$$

Por exemplo:

$p = 17$, $m = 6$, temos $h_{3,4}(8) = 5$.

todos hashs seriam:

$$H_{pm} = \{ h_{ab} : a \in \mathbb{Z}_p^* \text{ e } b \in \mathbb{Z}_p \}$$

- (p) escolhas para a
- $(p-1)$ escolhas para b