Tabelas Hash

Referência: Cormen et al, Introduction to Algorithms

- Conjuntos que podem ser alterados (aumentados, diminuídos, etc) são chamados **Conjuntos Dinâmicos**.
- Em uma implementação típica de um conjunto dinâmico, cada elemento é representado por um objeto cujos campos podem ser examinados e manipulados, através de um ponteiro para o objeto.
  - Ex: listas ligadas (ordenadas ou não).
- Usualmente, conjuntos dinâmicos assumem que um dos campos do objeto é um campo de **chave** de identificação. Se as chaves são todas diferentes, podemos pensar o conjunto dinâmico como um conjunto de valores de chaves. Os demais campos são considerados **dados satélite**.
- Dicionário: Conjunto dinâmico que suporta as operações de inserção, remoção e busca de elementos no conjunto.
  - Por exemplo, um compilador para uma linguagem de programação mantém uma tabela de símbolos, na qual as chaves dos elementos são strings de caracteres arbitrárias que correspondem aos identificadores na linguagem.

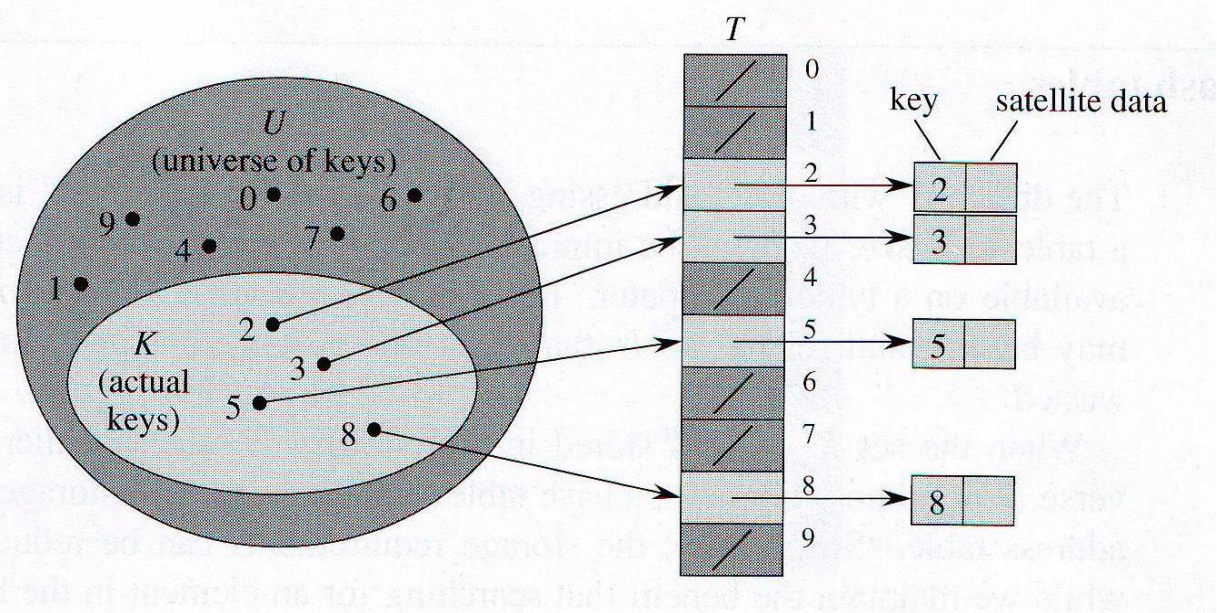
- TAD Dicionário:

  - SEARCH(S, k): Uma consulta que, dado um conjunto S e um valor de chave k, retorna um ponteiro x para um elemento em S tal que $key[x] = k$, ou NIL se não houver nenhum elemento com esse valor.

  - INSERT(S, x): Insere no conjunto S o elemento apontado por x.

  - DELETE(S, x): Dado um ponteiro x para um elemento do conjunto S, remove x de S.

Tabelas de acesso/endereçamento direto



- Acesso direto: técnica simples que funciona bem quando o universo de chaves U é razoavelmente pequeno.

- Suponha que cada elemento possui uma chave pertencente ao universo $U = \{0, 1, \ldots, m-1\}$, onde m não é muito grande. Assume-se que não haja dois elementos com a mesma chave.

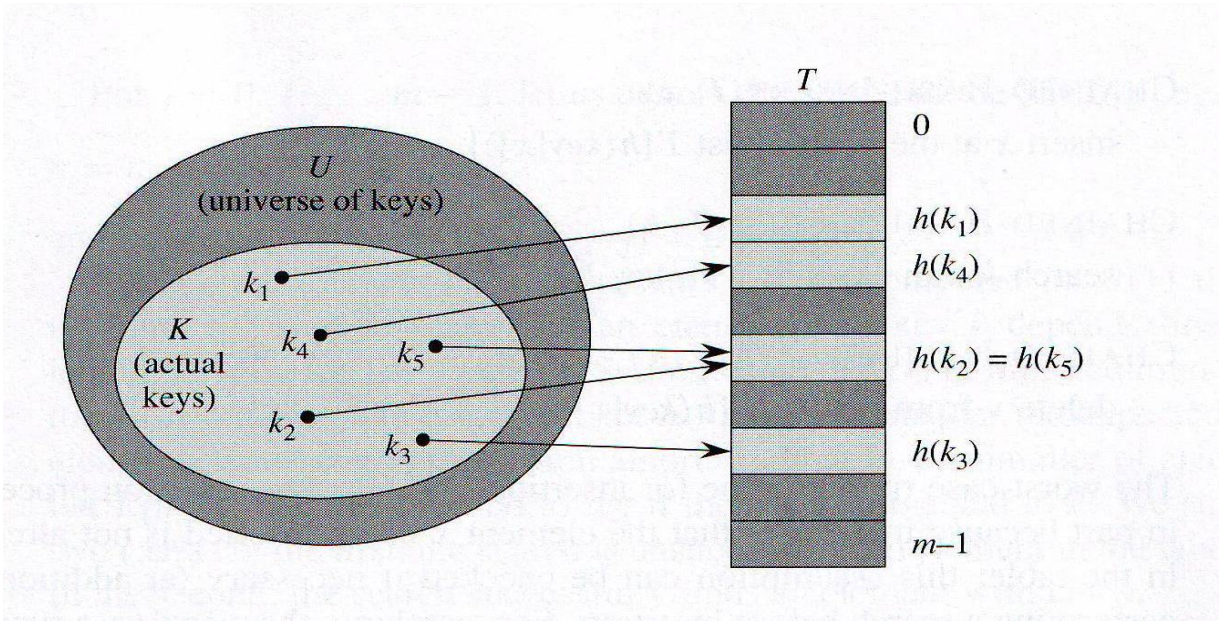
- Para representar o conjunto dinâmico, utiliza-se um arranjo, ou **tabela de endereçamento direto**, denotado por $T[0 \ldots m-1]$, onde cada posição (**slot**), corresponde a uma chave no universo U.
- Operações (Custo de tempo O(1)):

```
Direct-Address-Search(T,k)
  devolva(T[k])

Direct-Address-Insert(T,x)
  T[key[x]] := x

Direct-Address-Delete(T,x)
  T[key[x]] := NIL
```

- Para algumas aplicações, pode-se usar o próprio arranjo T para armazenar o dado satélite; a chave não precisa ser guardada, uma vez que corresponde à posição em T. Cuidado para indicar quando a posição está vazia.

Tabelas Hash

- Se o universo U é grande, pode ser impraticável (ou impossível) armazenar uma tabela T de tamanho U.
- Além disso, se o conjunto K de chaves *efetivamente armazenadas* for muito menor do que o universo U, a maior parte do espaço de memória alocada para T será desperdiçada.
- Idéia: Se $|U| >> |K|$, pode-se reduzir a necessidade de memória para $\Theta(|K|)$, mantendo o tempo (*médio*) das operações $O(1)$.
- Enquanto no acesso direto um elemento com chave k é armazenado na posição k, na tabela Hash esse elemento é armazenado na posição $h(k)$, onde h é uma *função de hash* que calcula a posição a partir do valor da chave, k.



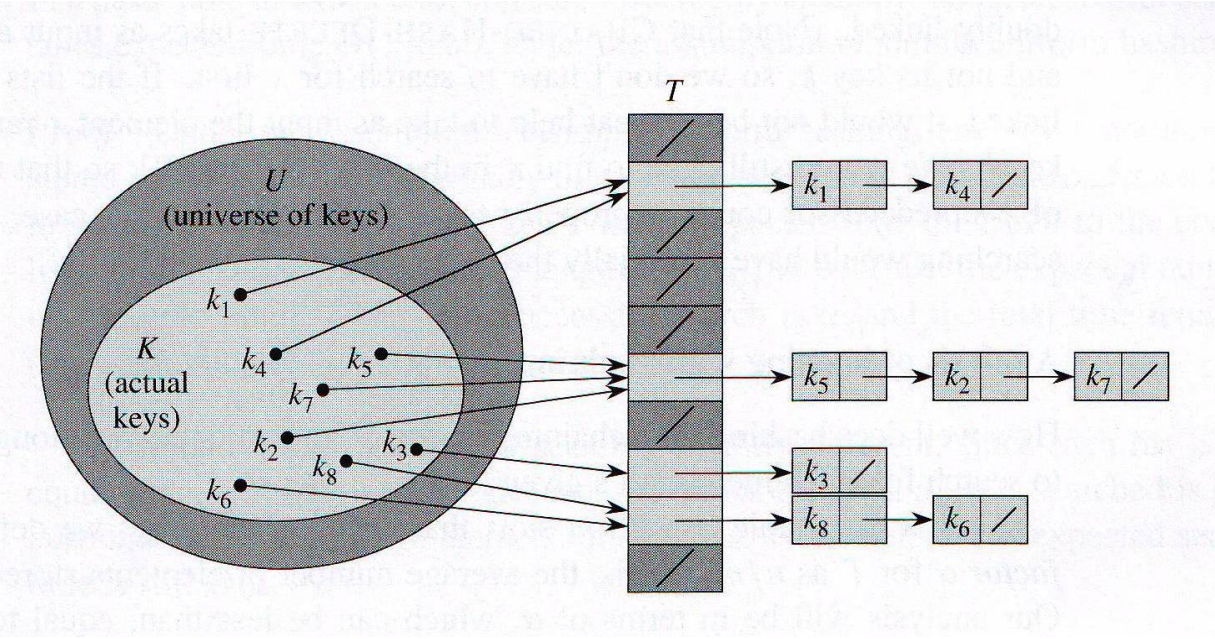
- h mapeia o universo U de chaves nas posições de uma tabela Hash $T[0 \ldots m-1]$:

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

- $h(k)$ não preserva a ordem das chaves.
- **Colisão**: situação em que duas chaves k_1 e k_2 têm o mesmo valor de hashing, ou seja, $h(k_1) = h(k_2)$. Escolha de funções h apropriadas tentam minimizar a probabilidade de ocorrência de colisões.
- Tratamento de colisões:
  - *Resolução por encadeamento* (Hashing aberto)
  - *Endereçamento aberto* (Hashing fechado)

Resolução de colisões por encadeamento (Hashing aberto)

- Na resolução por encadeamento, colocamos todos os elementos com mesmo valor de h em uma lista ligada. Na tabela hash T, cada posição j armazena o ponteiro para o início da lista ligada dos elementos x tais que $h(key[x]) = j$.



- Operações:

```
Chained-Hash-Search(T,k)
  procure um elemento com chave k
  na lista T[h(k)] e devolva seu ponteiro

Chained-Hash-Insert(T,x)
  insira x na cabeça da lista T[h(key[x])]

Chained-Hash-Delete(T,x)
  remova x da lista T[h(key[x])]
```

Análise da resolução por encadeamento

- Quanto custa (em média) a procura de um elemento na tabela hash com listas encadeadas?
- Dada uma tabela hash T com m slots armazenando n chaves, definimos o **fator de carga** α para T como $\alpha = n/m$ (número médio de elementos armazenados em cada posição de T).
- Vamos assumir que qualquer elemento tem a mesma probabilidade de ser colocado em qualquer dos m slots, ou seja, $Pr(h(k) = j) = 1/m, j = 0 \ldots m-1$. **Hashing uniforme simples**
- Para $j = 0, 1 \ldots m-1$, denotamos o tamanho da lista $T[j]$ por n_j, de forma que $n = n_0 + n_1 + \ldots n_{m-1}$. Sob a hipótese de hashing uniforme simples, o valor médio de n_j é:

$$E(n_j) = n/m = \alpha.$$

- Assume-se que $h(k)$ é calculado em tempo $O(1)$, e portanto o tempo requerido para procurar um elemento com chave k é linear no comprimento $n_{h(k)}$ da lista $T[h(k)]$. Deve-se então analisar quantos elementos da lista $T[h(k)]$ são acessados, em média, para procurar k.

- **Teorema:** Em uma tabela hash com resolução de colisões por encadeamento, assumindo-se hashing uniforme simples (h.u.s):

  a) O tempo esperado para uma busca sem sucesso é $\Theta(1+\alpha)$.

  Dem: Sob a hipótese de h.u.s, qualquer chave k não armazenada na tabela tem probabilidade $1/m$ de ser posicionada em qualquer dos m slots. O tempo esperado para buscar uma chave k sem sucesso é o tempo esperado para percorrer toda a lista $T[h(k)]$, que é $E[n_{h(k)}] = \alpha$. Logo, o número esperado de elementos examinados em uma busca sem sucesso é α, e o tempo total requerido (incluindo o cálculo de $h(k)$) é $\Theta(1+\alpha)$.

  b) O tempo esperado para uma busca bem sucedida é $\Theta(1+\alpha)$.

  Idéia da Demonstração: Seja x o elemento sendo procurado, e $k = key[x]$. Assumimos que x é igualmente provável de ser qualquer um dos n elementos armazenados na tabela. O número de elementos examinados durante uma busca bem sucedida é 1 mais o número de elementos que aparecem antes de x em sua respectiva lista. Como a posição esperada de x dentro de sua lista é $(n_{h(k)}+1)/2$ e como $E[n_{h(k)}] = \alpha$, então o número esperado de elementos examinados na busca de x será $1 + (\alpha+1)/2$, e o tempo total será $\Theta(1+\alpha/2) = \Theta(1+\alpha)$.

Endereçamento aberto (Hashing fechado)

- No endereçamento aberto, todos os elementos são armazenados na própria tabela hash. Cada entrada da tabela contém ou um elemento do conjunto dinâmico ou NIL.
- Em endereçamento aberto, a tabela hash pode encher, de forma que não seja mais possível inserir elementos.
- Vantagem: eliminação de ponteiros / eficiência do uso de espaço / implementação em disco
- Para realizar uma inserção, examinamos/testamos sucessivamente a tabela hash até que encontrarmos um slot vazio no qual a chave será inserida. (*rehashing*)
- Para determinar a seqüência de posições a serem examinadas, a função de hashing é extendida de maneira a incluir o número do teste como segunda entrada:

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

- Requer-se que a **seqüência de teste** (*probe sequence*) $\langle h(k,0), h(k,1) \ldots h(k,m-1) \rangle$ seja uma permutação de $\langle 0, 1 \ldots m-1 \rangle$, de forma que toda posição da tabela hash seja eventualmente considerada como um slot para uma nova chave à medida em que a tabela enche.

- Operações:

```
Hash-Insert(T,k)
  i := 0
  repeat
    j := h(k,i)
    if T[j] = NIL or T[j] = DELETED
      T[j] := k
      return j
    else i := i+1
  until i = m
  error "Estouro da tabela hash"

Hash-Search(T,k)
  i := 0
  repeat
    j := h(k,i)
    if T[j] = k
      return j
    i := i+1
  until T[j] = NIL or i = m
return NIL
```

```
Hash-Delete(T,k)
  i := 0
  repeat
    j := h(k,i)
    if T[j] = k
      T[j] := DELETED
    i := i+1
  until T[j] = NIL or i = m
```

- Quando o valor especial DELETED é usado, o tempo de busca não é mais dependente do fator de carga α.

Técnicas de rehashing

- Hipótese de hashing uniforme: assume-se que, para cada chave, sua seqüência de teste tem a mesma probabilidade de ser qualquer uma das $m!$ permutações de $\langle 0, 1 \ldots m-1 \rangle$.
- **probing linear**: usa uma função de hashing comum, $h' : U \rightarrow \{0, 1 \ldots m-1\}$.

  Dada a chave k, o primeiro slot testado é $T[h'(k)]$. Se estiver ocupado, os próximos slots são $T[h'(k)+1], T[h'(k)+2], \ldots, T[m-1], T[0], \ldots T[h'(k)-1]$.

  Formalmente,

$$h(k,i) = (h'(k) + i) \bmod m \ , \quad i = 0, 1 \ldots m-1.$$

- Desvantagem do probing linear: *clustering primário* – longas seqüências de slots ocupados

Clusters ocorrem porque um slot vazio precedido por i slots ocupados tem probabilidade (i+1)/m de ser ocupado na próxima inserção.

- **probing quadrático**: função da forma

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

onde h' é a função de hashing auxiliar, c_1 e c_2 são constantes auxiliares, e $i = 0, 1, \ldots m-1$.

O rehashing é deslocado por distâncias que dependem quadraticamente de i.

Para que todas as posições da tabela hash sejam exploradas, os valores de c_1, c_2 e m são críticos.

- Desvantagem do probing quadrático: *clustering secundário* – se duas chaves k_1, k_2 têm a mesma posição inicial de teste, então sua seqüência de rehashing será a mesma, uma vez que $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$.

Assim como no probing linear, a posição inicial determina a seqüência inteira, e assim somente m seqüências distintas são usadas.

- **hashing duplo**: função na forma

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m,$$

onde h_1 e h_2 são funções hash auxiliares.

O primeiro slot testado é $T[h_1(k)]$. Cada posição testada posteriormente é um deslocamento de tamanho $h_2(k)$ em relação à posição anterior, módulo m.

Cada par $(h_1(k), h_2(k))$ determina uma seqüência $\Rightarrow$ total de $\Theta(m^2)$ seqüências.

- **Escolha de m e h_2 com hashing duplo**:

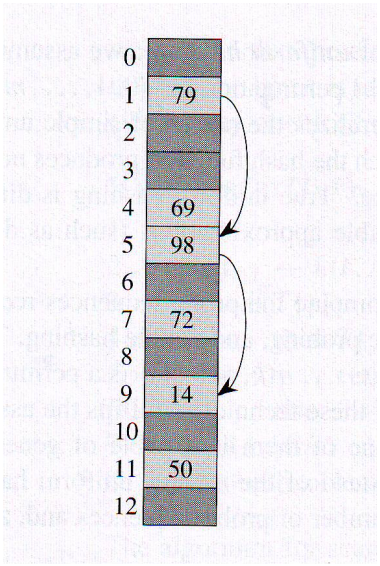
  Lema: Se $d \geq 1$ é o máximo divisor comum de m e $h_2(k)$, então uma busca mal sucedida da chave k examinará m/d posições da tabela hash antes de voltar para o slot $h_1(k)$.

  Pelo lema acima, idealmente o valor $h_2(k)$ precisa ser primo relativo de m, para que a tabela hash inteira possa ser examinada.

  Uma forma possível: toma-se m primo e define-se h_2 que sempre retorna um inteiro positivo menor do que m.

  Exemplo: m primo, $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$ onde m' é ligeiramente menor do que m (p. ex. $m - 1$).

  A figura abaixo apresenta um exemplo de inserção por hashing duplo onde $k = 14$, $m = 13$, $h_1(k) = k \bmod 13$, $h_2 = 1 + (k \bmod 11)$.



Custo de tempo do hashing de endereçamento aberto

- Análise em termos do fator de carga $\alpha = n/m$ da tabela hash, para valores assintóticos de n, m.

(Note que no hashing fechado tem-se no máximo uma chave por posição, e portanto $n \leq m \Rightarrow \alpha \leq 1$.)

- **Teorema:** Dada uma tabela hash de endereçamento aberto com fator de carga $\alpha = n/m < 1$, o número esperado de slots testados em uma busca sem sucesso é no máximo $1/(1-\alpha)$, assumindo hashing uniforme.
- **Teorema:** Dada uma tabela hash de endereçamento aberto com fator de carga $\alpha = n/m < 1$, o número esperado de slots testados em uma busca bem sucedida é no máximo $1/\alpha \ln[1/(1-\alpha)]$, assumindo hashing uniforme e assumindo que cada chave da tabela tem a mesma probabilidade de ser procurada.