CI1057: Algoritmos e Estruturas de Dados III
Tabelas Hash

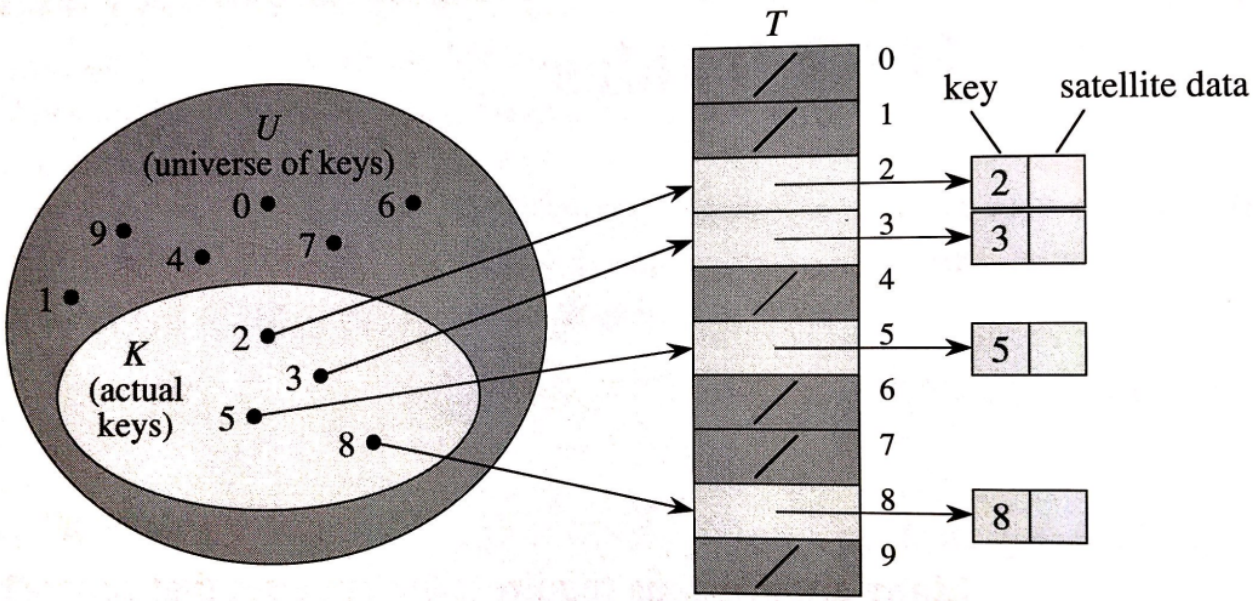
Profa. Carmem Hara

Departamento de Informática/UFPR

17 de julho de 2024

Tabelas com Endereçamento Direto

- Implementação eficiente para o TAD Dicionário
- Funciona bem quando o conjunto de chaves possíveis é relativamente pequeno
- Consideramos que não há chaves repetidas
- Para chaves no intervalo $[0, n]$, aloca-se um vetor de $n + 1$ posições



Operações em Tabelas com Endereçamento Direto

```
1  #define N 100
2
3  typedef struct item{
4    int chave;
5    float valor;
6  } TipoItem;
7
8  TipoItem *T[N+1];
9
10 TipoItem *busca( TipoItem *T[], int k ){
11   return T[k];
12 }
13
14 void insere( TipoItem v, TipoItem *T[] ){
15   int k= v.chave;
16   T[k]= (TipoItem *) malloc( sizeof(TipoItem));
17   T[k]->chave= v.chave;
18   T[k]->valor= v.valor;
19 }
20
21 void delete( TipoItem *T[], int k ){
22   free( T[k] );
23   T[k]= NULL;
24 }
```

Todas as operações: $O(1)$ (pior caso)

Problema do Endereçamento Direto

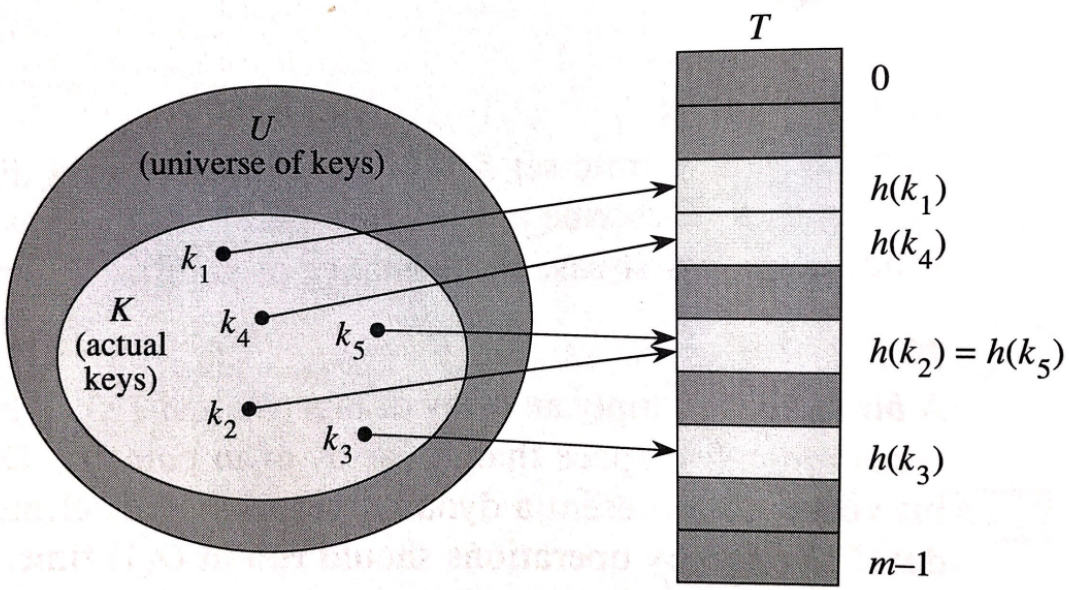
O intervalo dos valores de chave pode ser muito maior que a quantidade de valores que serão armazenados
Ex: chaves no intervalo $[0, 10.000]$, mas apenas 1000 elementos no vetor.

Solução: Tabelas Hash

- Dão suporte ao TAD Dicionário (busca, inserção, remoção)
  note que não há busca por intervalo
- É uma generalização do tipo vetor
  alocação de um vetor de K elementos para armazenar K itens
- Todas as operações continuam $O(1)$ (caso médio)
- **Idéia:**
  1. Computar o valor de uma funcao de espalhamento (hash - $h(k)$)
  2. Armazenar o elemento na posição $h(k)$ do vetor

Tabelas Hash

- Função de espalhamento (*hash*): a função mapeia o conjunto possível de chaves U para uma posição da tabela com m elementos $h : U \rightarrow \{0, 1, \ldots, m-1\}$



Problema das Tabelas Hash

Problema: colisões
Podem existir 2 (ou mais) chaves $k1$ e $k2$ tal que $h(k1) = h(k2)$.

Paradoxo do aniversario:
Em um grupo de 23 ou mais pessoas, existe uma chance de mais de 50% que duas pessoas façam aniversário no mesmo dia.

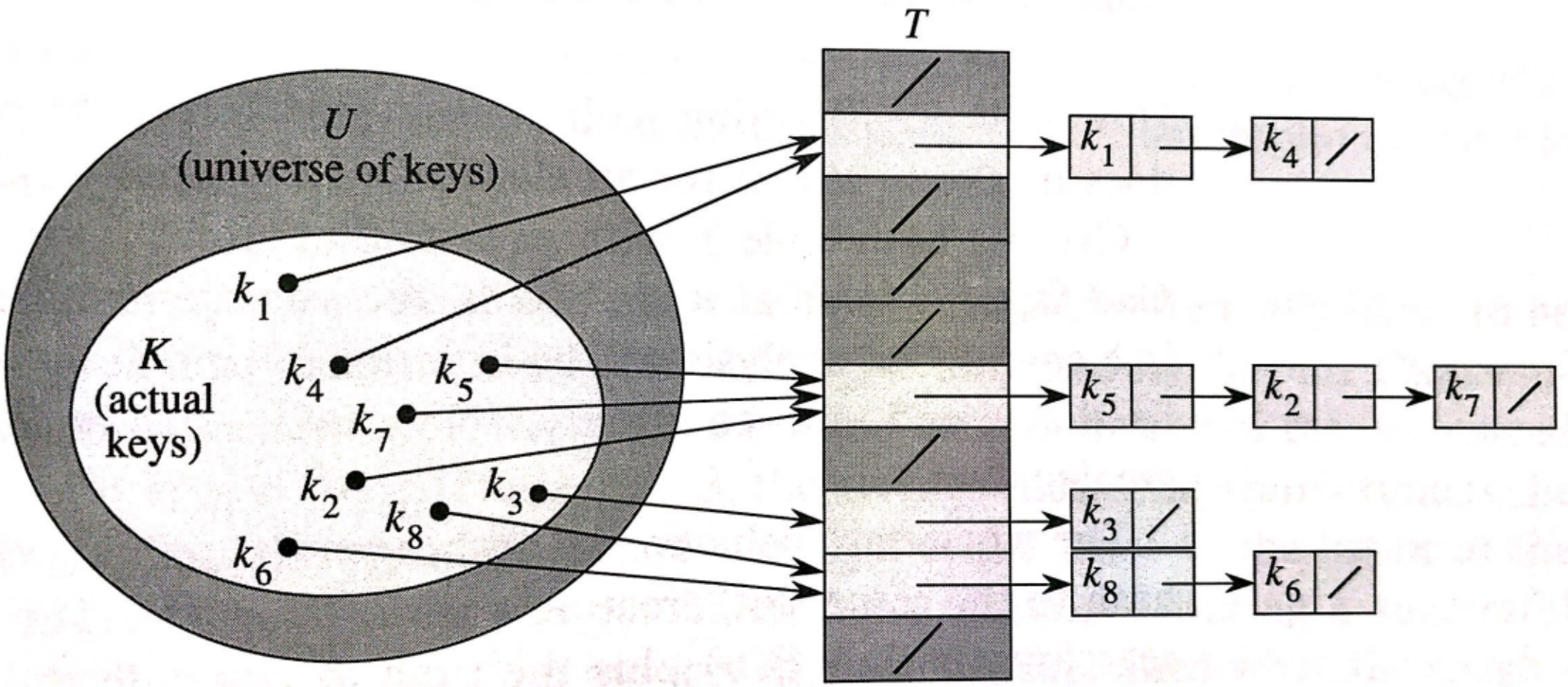
Exemplo: $m = 7 \quad h(k) = k \ mod \ 7$

k	h(k)	
4	4	
7	0	
13	6	
8	1	
9	2	
2	2	Colisão

0	1	2	3	4	5	6
7	8	9	13	4		
		2				

Resolução de Colisão por Encadeamento

Cada elemento do vetor aponta para uma lista encadeada.



Encadeamento - Estrutura de Dados

```
1  #define N 7
2
3  typedef struct item  *Apltem;
4  typedef struct item{
5    int chave;
6    float valor;
7    Apltem prox;
8  } Tipoltem;
9
10 Tipoltem *T[N];
11
12 int h(int k ){
13   return k%N;
14 }
```

Busca

```
1  TipoItem *busca( TipoItem *T[], int k ){
2    int i= h(k);
3    TipoItem *v= T[i];
4
5    while( v!= NULL ){
6      if( v->chave==k ) return v;
7      v= v->prox;
8    }
9    return NULL;
10 }
```

Custo

- Pior caso: $O(n)$
  se a função h mapear todas as chaves para o mesmo elemento do vetor
- Caso médio: $O(n/m)$
  - n/m é o fator de carga: número médio de elementos em cada posição do vetor
    Considera a suposição de *hashing uniforme*
  - para valores de m proximos de n: $O(n/m) = O(1)$
- Melhor caso: $O(1)$

Inserção

```
1 void insere( TipoItem v, TipoItem *T[] ){
2   int i= h(v.chave);
3   TipoItem *ant = T[i];
4   T[i]= (TipoItem *) malloc( sizeof(TipoItem));
5   T[i]->chave= v.chave;
6   T[i]->valor= v.valor;
7   T[i]->prox= ant;
8 }
```

Custo: $O(1)$

Remoção

```
1  void delete( TipoItem *T[], int k ){
2    int i= h(k);
3    TipoItem *ant = NULL, *v = T[i];
4
5    while( v!=NULL && v->chave != k ){
6      ant= v;
7      v= v->prox;
8    }
9    if( v->chave == k ){
10     if( ant == NULL )
11       T[i]= v->prox;
12     else
13       ant->prox= v->prox;
14     free( v );
15   }
16 }
```

Custo: mesmo que a inserção

Características de um Boa Função de Hash

- Satisfaz a suposição de *Hashing Uniforme Simples*
  As chaves são igualmente distribuídas nos slots disponíveis
- Seja $P(k)$ a probabilidade que $h(k) = j$ para $0 \leq j < m$:
  $\sum_{k:h(k)=j} P(k) = \frac{1}{m}$ para todo $j = 0, 1, \ldots m-1$
- Exemplo:
  - Chaves k uniformemente distribuídas no intervalos $[0, 1)$
  - $h(k) = \lfloor km \rfloor$ é uma função de hash uniforme simples
- Em geral $P(k)$ não é conhecida
- Usam-se heurísticas
  - funções que são independentes de padrões existentes nas chaves
  - valores próximos de chave tem valores de hash "distantes"
- Funções devem ser eficientes e fáceis de serem computadas

Mapeando Chaves para um Número Natural

Exemplo: Chaves do tipo string ($v[tamV]$)

- Usar o valor da Tabela ASCII de cada caracter
  $ord(a) = 97 \quad ord(b) = 98 \quad ord(e) = 101$
  $ord(l) = 108 \quad ord(p) = 112$
- **Estratégia 1:** Somatório
  $k(v) = \sum_{i=0}^{tamV-1} ord(v[i])$

  Ex: $k(babel) = 98 + 97 + 98 + 101 + 108 = 502$

Outras Estratégias

- **Estratégia 2:** Somatório e Posição
  $k(v) = \sum_{i=0}^{tamV-1} ord(v[i]) * 128^{tamV-1-i}$

  Ex: $k(babel) = 98 * 128^4 + 97 * 128^3 + 98 * 128^2 + 101 * 128^1 + 108 * 128^0 = 26.511.717.100$

- **Estratégia 3:** Usando Pesos Aleatórios
  $k(v) = \sum_{i=0}^{tamV-1} ord(v[i]) * peso[i]$

  - Vantagem: pesos distintos geram funções de espalhamento distintos

Funções de Hash

- Estático
  - Método da divisão
  - Método da multiplicação
  - Meio do quadrado
  - Shift Folding
  - Limit Folding
- Aleatório
  - Universal

Método da Divisão

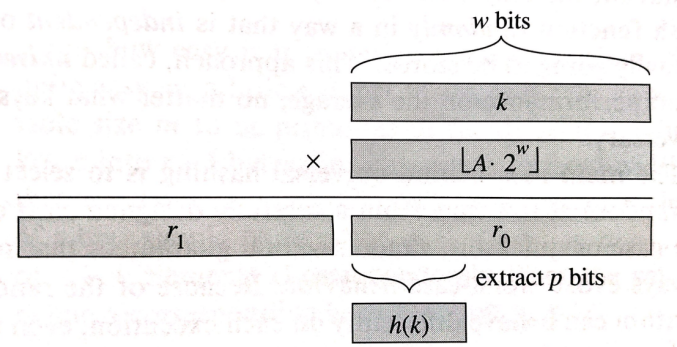
$$h(k) = k \ mod \ m$$

- A escolha de m e' importante
- Evitar potências de 2
  - Exemplo: se $m = 64 = 2^6$
    $h(k)$ corresponde aos 6 bits menos significativos de k
  - É melhor considerar a chave como um todo.
- Em geral e' escolhido um valor primo para m
  - Boa escolha: primos não muito próximos a potências de 2
  - Exemplo: $n = 2000$, com resolução de colisão por lista ligada de 3 elementos
    Tamanho do vetor: $2000/3 = 666$
    Escolha de m: 701, primo que não é próximo a uma potência de 2

Método da Multiplicação

Dada uma constante A, $0 < A < 1$
$h(k) = \lfloor m * parteFracionaria(k * A) \rfloor$

- **Vantagem:** o método não e' muito dependente do valor de m
- Pode ser implementado de forma eficiente quando $m = 2^p$
  - k é uma chave de w bits
  - s é um valor no intervalo $(0, 2^w)$, tal que $A = s/2^w$
  - $k * s$ é um valor de $2w$ bits $(r1, r0)$
  - Como $m = 2^p$, o valor de $h(k)$ corresponde aos p bits mais significativos de $r0$.



Método da Multiplicação - Exemplo

- $A = 0.6180339887$
  Esta é a *razão de ouro*, $\frac{\sqrt{5}-1}{2}$ sugerida por Knuth
- $p = 14$
- $m = 2^{14} = 16384$
- $w = 32$ (chave de 4 bytes)
- $k = 123456$

1. Procura-se um valor de s, tal que $A = \frac{s}{2^{32}} = \frac{\sqrt{5}-1}{2}$
   $s = 0.6180339887 * 2^{32} = 2.654.435.769$
2. $k * s = 123.456 * 2.654.435.769 = 327.706.022.297.664 =$
   $76300 * 2^{32} + 17.612.864$
   Ou seja, $r1 = 76300$ e $r0 = 17.612.864$
3. Os $p = 14$ bits mais siginificativos de $r0$ resulta na valor de
   $h(k) = 67$

Método da Multiplicação - Implementação

$$h(k) = (k * s)\ mod\ 2^w >> (w - p)$$

- $mod\ 2^w$ elimina a parte alta ($r1$)
- $>> (w - p)$ faz o shift à direita para obter os p bits mais significativos

Método Meio do Quadrado

Idéia

Multiplica-se a chave por ela mesma e trunca-se as duas extremidades do resultado até o número de dígitos ser igual ao número de dígitos do endereço desejado (no intervalo da tabela hash).

Exemplo 1

- Endereçamento: 3 dígitos
- Chave: $k = 134675 \quad k^2 = 18137355625$
- Truncando as extremidades: $h(k) = 735$

Exemplo 1

- Endereçamento: 3 dígitos
- Chave: $k = 436987 \quad k^2 = 190957638169$
- Truncando as extremidades: $h(k) = 763$

Método Shift Folding - Deslocamento

Idéia

1. Dividem-se os digitos (ou bits) da chave para obter inteiros menores
2. Somam-se as partes (sem o "carry")

Exemplo:

- Endereçamento: 3 dígitos
- Chave $k = 12345678$
- Divisão: 12, 345, 678
- Soma: $12+345+678 = (0+3+6)(1+4+7)(2+5+8)$: $h(k) = 925$

Método Limit Folding - Dobramento ou Sanfona

Idéia: A mesma do Shift Folding, mas em cada parte são invertidos os digitos das partes, inciando à esquerda e à direita

Exemplo:

- Endereçamento: 3 dígitos
- Chave $k = 12345678$
- Divisão: 012 (inversão=210), 345, 678 (inversão=876)
- Soma: (2+3+8)(1+4+7)(0+5+6): $h(k) = 321$

Hash Universal

Motivação

- Um adversário malicioso pode utilizar n chaves que mapeiam todos para o mesmo slot do vetor
- Tempo de busca passa a ser linear

Solução

- Incluir aleatoriedade na função de hash
- Isso torna a função independente dos valores das chaves
- Com a randomização, a função pode comportar-se diferente a cada execução

Hash Universal

Uma função de hash é universal se para cada par de chaves distintas x e h, a chance de colisão (ou seja, $h(x) = h(y)$) é exatamente $1/m$.

- Esta propriedade é exatamente a chance de colisão se $h(x)$ e $h(y)$ forem randomicamente escolhido no intervalo $[0, m)$.
- Como as chaves raramente são randômicas, a função de hash deve *introduzir* aleatoriedade
  Exemplo: Estratégia 3 da transformação de strings para um número natural utilizando pesos aleatórios

Método Shift Folding com Pesos Aleatórios

Idéia

1. Dividem-se os digitos (ou bits) da chave para obter inteiros menores
2. *Atribuem-se pesos aleatórios (números randômicos) para cada parte*
3. Somam-se as partes (sem o "carry")

Exemplo:

- Endereçamento: 3 dígitos
- Chave $k = 12345678$
- Divisão: 12, 345, 678 com pesos 13 7 19
- Soma: (0+3+6)*13=117 (1+4+7)*7=84 (2+5+8)*9=135: $h(k) = 745$

Tratamento de Colisão com Endereçamento Aberto

Slides do Prof. Paulo Almeida

Endereçamento aberto

Ideia: usar a própria Tabela Hash para tratar colisões, sem a necessidade de uma lista encadeada.

Cada *slot* vai conter um elemento, ou *NULO*.

Inserir

Calcular a “primeira opção” de *slot* para inserir o elemento.

Se *slot* vazio, inserir.

Senão

Calcular a “segunda opção” de *slot* para inserir o elemento.

Se *slot* vazio, inserir.

...

Continuar o processo até encontrar um *slot* vazio ou até concluir que a Tabela está cheia.

Buscar

Calcular a “primeira opção” de *slot* onde a chave *k* pode estar.

Se a chave se encontra na posição, retornar o conteúdo.

Senão

Calcular a “segunda opção” de *slot* onde a chave *k* pode estar.

Se a chave se encontra na posição, retornar o conteúdo.

...

Continuar o processo até encontrar a chave, ou até encontrar NULO (chave não encontrada).

Vantagens e Desvantagens

Quais vantagens e desvantagens quando comparado com o método de encadeamento?

Vantagens e Desvantagens

Quais vantagens e desvantagens quando comparado com o método de encadeamento?

+ Evitar o uso de ponteiros.
+ Economizar memória e talvez simplificar o problema.

    A memória economizada pode ser usada para criar uma Tabela Hash maior, mitigando colisões.

- Quando a Tabela está cheia, a única opção é construir uma Tabela maior (*realloc*).
- As operações de inserção podem custar *O(n)* no pior caso.

Sondar

Para inserir um item é necessário **sondar** (***probe***) os *slots* da Tabela.

Continuar sondagem até encontrar uma posição vazia.

A função de hash possui dois parâmetros:

h(k,i)

Onde:

k é a chave, e *i* é um número de sondagem, que inicia em 0.

Exemplo

Verificar se *h(k,0)* está vazio.

Se não estiver, verificar *h(k,1)*.

Se não estiver, verificar *h(k,2)*.

...

Se não estiver, verificar *h(k,m-1)*.

Propriedades

Além das propriedades discutidas em aulas passadas, qual propriedade extra é fundamental na função *h(k,i)*?

Propriedades

Além das propriedades discutidas em aulas passadas, qual propriedade extra é fundamental na função *h(k,i)*?

Precisamos garantir que todas as posições da Tabela são acessíveis ao modificar o valor de *i*.

E pelo bem da eficiência $h(k,i) \neq h(k,j), \forall i,j < m, i \neq j$

Propriedades

Além das propriedades discutidas em aulas passadas, qual propriedade extra é fundamental na função *h(k,i)*?

Precisamos garantir que todas as posições da Tabela são acessíveis ao modificar o valor de *i*.

E pelo bem da eficiência $h(k,i) \neq h(k,j), \forall i,j < m, i \neq j$

Ao executar a função usando *i=0, 1, ..., m-1* , garantimos que passamos por todos os *m* slots.

Faça você mesmo

Considerando a função de hash *h(k,i)*, e um objeto *x*, onde *x.k* é a chave.

Como podem ser implementadas as seguintes funções?

```
inserir(T,x)
```

```
buscar(T,k)
```

Inserir

```
inserir(T,x)
    i = 0
    faça
        q = h(x.k,i)
        se T[q] == NULO
            T[q] = x
            retorne q
        i = i + 1
    enquanto i < T.m
    erro "Tabela Cheia"
```

Buscar

```
buscar(T,k)
    i = 0
    q = h(k,i)
    enquanto T[q] != NULO E i < T.m
        se T[q].k == k
            retorne q
        i = i + 1
        q = h(k,i)
    retorne NULO
```

Excluir

Como pode ficar o algoritmo de exclusão? Quais as dificuldades extras?

```
excluir(T,q)
```

Excluir

Como pode ficar o algoritmo de exclusão? Quais as dificuldades extras?

```
excluir(T,q)
    T[q] = NULO
```

Excluir

Como pode ficar o algoritmo de exclusão? Quais as dificuldades extras?

```
excluir(T,q)
    T[q] = NULO
```

O algoritmo de busca deixará de funcionar. Um elemento de chave *k* que precisou ser alocado em outra posição quando *q* estava ocupado não será acessível.

```
buscar(T,k)
    i = 0
    q = h(k,i)
    enquanto T[q] != NULO E i < T.m E
        se T[q].k = k
            retorne q
        i = i + 1
        q = h(k,i)
    retorne NULO
```

Excluir

Uma possível solução é marcar as posições excluídas com um valor especial DELETADO.

```
excluir(T,q)
    T[q] = DELETADO
```

Problemas?

```
buscar(T,k)
    i = 0
    q = h(k,i)
    enquanto T[q] != NULO E i < T.m E
        se T[q].k = k
            retorne q
        i = i + 1
        q = h(k,i)
    retorne NULO
```

Excluir

Uma possível solução é marcar as posições excluídas com um valor especial DELETADO.

```
excluir(T,q)
    T[q] = DELETADO
```

Problemas?

O algoritmo de busca deixa de depender do fator de carga α.

Pode ser necessário passar por todas *m* posições da tabela para concluir que a chave não existe.

Mesmo em uma **Tabela Vazia!**

Excluir

Uma possível solução é marcar as posições excluídas com um valor especial DELETADO.

```
excluir(T,q)
    T[q] = DELETADO
```

Problemas?

O algoritmo de busca deixa de depender do fator de carga α.

Pode ser necessário passar por todas *m* posições da tabela para concluir que a chave não existe.

Mesmo em uma **Tabela Vazia!**

Por conta disso, quando as exclusões são frequentes, é comum a utilização de **encadeamento**.

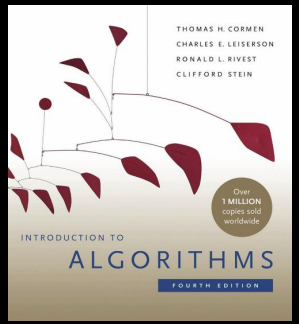
Função de Hash

Idealmente, a função *h(k,i)* deve ser uma função de hash de permutação uniforme e independente.

Também chamada de função de hash uniforme.

A sequência de *slots* sondados deve ser qualquer uma das *m!* permutações das *[0, 1, ..., m-1]* posições, onde todas as permutações possíveis têm a mesma probabilidade de acontecer.

Veja detalhes em Cormen et al. (2022).



Hashing Linear

Idéia

Se a posição estiver ocupada, procura a próxima posição livre.

$$h(k,i) = (h'(k) + i) \ mod \ m$$

Exemplo (para $m = 10$)

Inserção das chaves: 74 43 93 18 82 38 92

m = 10 $\quad h'(k) = k \ mod \ 10$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		82	43	74	93	92		18	38

Hashing Linear

Custo da busca

- Pior caso: $O(n)$
- Caso médio: $O(1)$
- Melhor caso: $O(1)$

Desvantagens

- Agrupamento: à medida que a tabela enche, uma nova chave tende a ocupar uma posição contígua a uma já ocupada

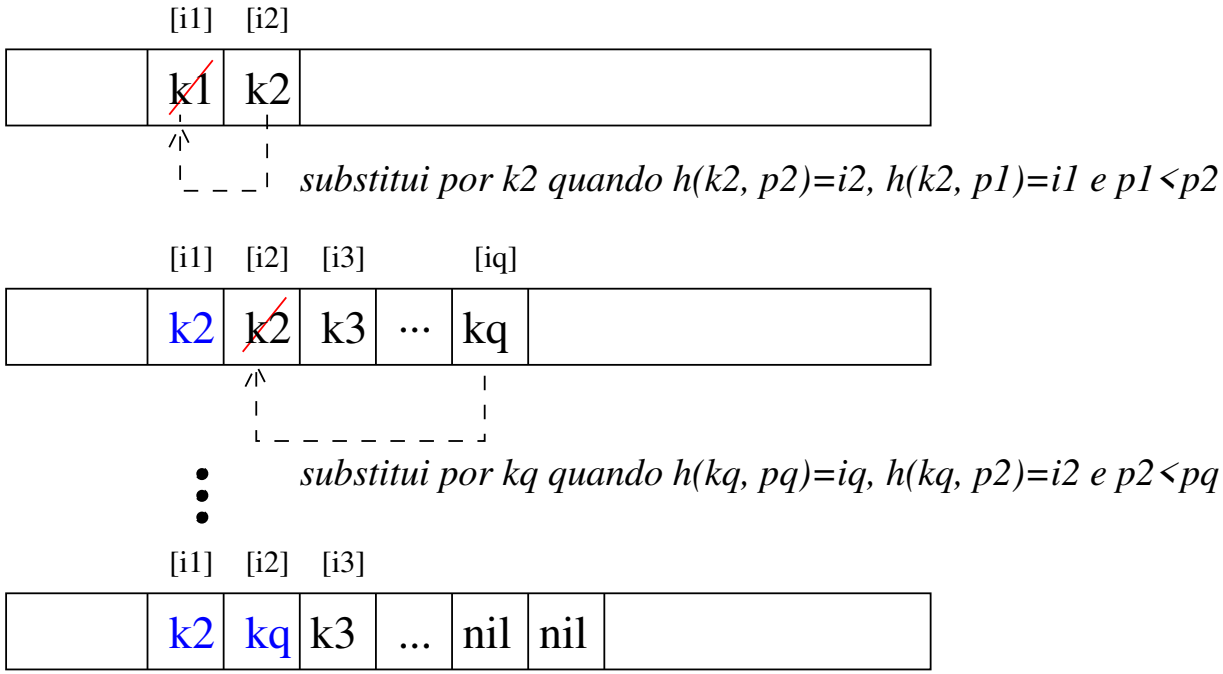
Vantagens

- Método simples
- Bom para tabelas esparsas
- É possível remover elementos SEM utilizar uma marcação de *Deletado*

Remoção - Hashing Linear

Idéia Substituir o elemento a ser removido por outro (em posições seguintes) que não ocupou aquela posição **anteriormente** porque houve colisão.

- É necessário saber o número da sondagem (*probe* - p_i)que fez uma determinada chave k_i ser armazenada na posição x.



Remoção - Exemplo

m = 10 $\quad h'(k) = k \ mod \ 10 \quad h(k,i) = (h'(k) + i) \ mod \ m$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		82	43	74	93	92		18	38

Remoção do 43:
Qual o valor que pode substituir 43 na posição [3]?

- $h(74,0)$=4 e $h(74,9)$=3: 74 ocuparia a posição 3 somente na 9^a iteração
- $h(93,2)$=5 e $h(93,0)$=3: 93 ocuparia a posição 3 na primeira iteração (ANTES da posição 5)

Remoção - Exemplo

m = 10 $\quad h'(k) = k \ mod \ 10 \quad h(k,i) = (h'(k) + i) \ mod \ m$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		82	93	74	93	92		18	38

Remoção do 93:
Qual o valor que pode substituir 93 na posição [5]?

- $h(92,4)=6$ e $h(92,3)=5$: 92 ocuparia a posição 5 na terceira iteração (ANTES de ocupar a posição 6)

Remoção - Exemplo

m = 10 $\quad h'(k) = k \ mod \ 10 \quad h(k,i) = (h'(k) + i) \ mod \ m$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		82	93	74	92	92		18	38

Remoção do 92:
Qual o valor que pode substituir 92 na posição [6]?
Como a próxima posição está vazia, não é necessário fazer nenhuma substituição

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		82	93	74	92			18	38

Remoção - Implementação

Função para obter a iteração i tal que $h(k, i) = q$:

$$probe(k, q) = (q - h'(k))\ mod\ m$$

```
1 int probe( int k, int q ){
2   int i = (q- hLinha(k)) % m; /* pode dar numero negativo */
3   if( i<0 )                   /* -6 % 10= -6 (resto) */
4     return i+m;               /* mod deve ser um valor positivo */
5   else
6     return i;
7 }
```

Remoção - Implementação

```
1  void delete( TipoItem T[], int k ){
2    int q1, q2, k2;
3
4    q1= busca( T, k );
5    if( q1 == NAO_ACHOU )
6      return;
7    while( T[q1].chave != NIL ){
8      T[q1].chave= NIL;
9      q2= (q1+1) % m;
10     k2= T[q2].chave;
11     while( k2 != NIL && probe( k2, q1) > probe( k2, q2 ) ){
12       q2= (q2+1) % m;
13       k2= T[q2].chave;
14     }
15     if( k2 != NIL ){
16       T[q1].chave= k2;
17       T[q1].valor= T[q2].valor;
18       q1= q2;
19     }
20   }
21 }
```

Hashing Duplo

Idéia

Utiliza 2 funções de hash:

- uma para a posição inicial
- outra para o deslocamento, caso a posição esteja ocupada

$$h(k,i) = (h1(k) + i * h2(k)) \ mod \ m$$

- **Vantagem:** quando n/m é próximo de 1, minimiza a chance de colisão
- **Desvantagem:** custo maior para computar $h(k,i)$

Hashing Duplo - Exemplo

Exemplo (para $m = 10$)
Inserção das chaves: 74 43 93(2) 18 82 38(3) 92(1)
m = 10 $h1(k) = k \ mod \ 10$ $h2(k) = k \ mod \ 7$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	38	82	43	74	93	92		18	

Tabelas Hash em Armazenamento Externo

- Cada slot do vetor corresponde a uma página, chamada de *bucket*
- A página p_i contém o conjunto de chaves k tal que $h(k) = i$
- É possível que seja necessário alocar *páginas de overflow*
- Em geral, é considerado que o acesso (busca) de uma chave tem custo 1.2 acessos a disco
  - Isso significa que em 20% dos casos é necessário acessar uma página de *overflow*: $O(1)$
  - Mais eficiente que busca em árvore B: $O(lg\ n)$
  - Só dá acesso a busca exata
  - Não dá suporte a busca por intervalo

Referências

- Livro Cormen, Cap. 12