Hashing

Introdução

- Quais são as operações básicas esperadas sobre chaves/dados de em uma estrutura de dados?
  - Inserção / Atualização
  - Remoção
  - **Pesquisa / Busca**
    - Fundamental às demais operações: em todas precisamos identificar local do dado
    - Fundamental para desempenho da estrutura de dados

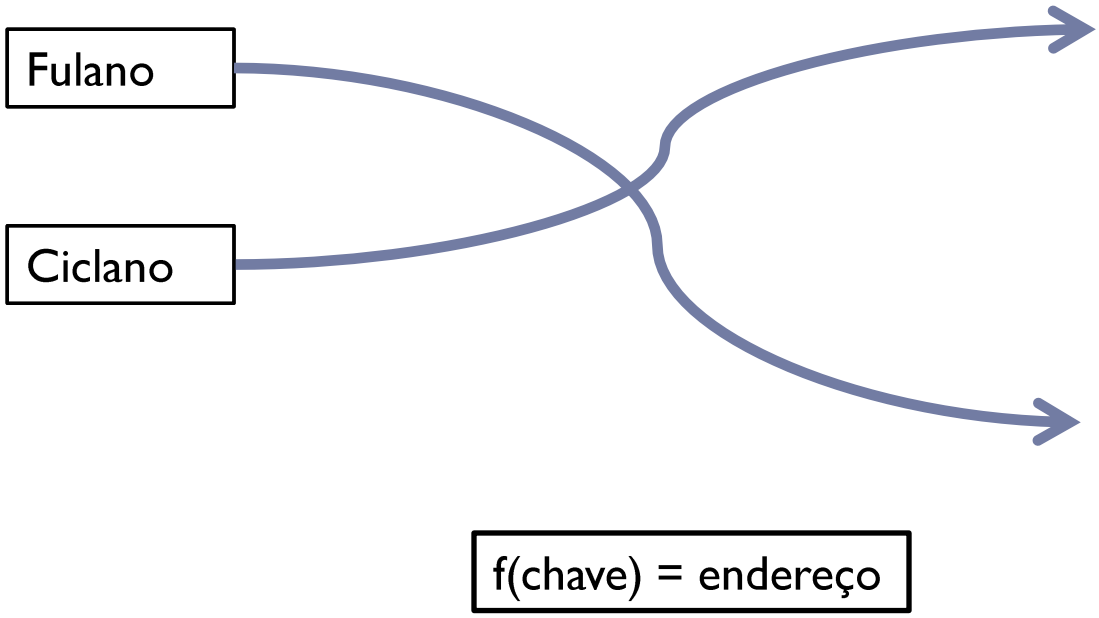
Discussão Inicial

- Suponha uma lista encadeada *L* onde cada uma das *n* chaves tem um endereço
- Como funciona a pesquisa da chave *k* em L?
  - Visitamos tantos endereços quantos necessário até achar *k*
    - Ou concluir que ele não está em *L*
  - i.e., em uma lista, não sabemos, a priori, o endereço de uma chave dada
    - Desempenho: No pior caso acessa todos os *n* elementos
- E se, dada uma chave, soubéssemos seu endereço de armazenamento na estrutura de dados sem precisar percorrer *n* elementos?
  - Melhoria de desempenho mediante devido tecnologia RAM
    - Acesso direto

Espalhamento (Hashing)

- *Hash* é uma estrutura de dados que determina o endereço de uma chave com base em seu valor
- Basicamente composta por:
  - **Tabela *T***
    - Contém M posições/endereços contíguos diferentes (vetor)
    - M é constante, pois integra a fórmula para cálculo dos endereços
  - **Função de espalhamento *f*(x)**
    - Mapeia o valor de uma chave *x* a um endereço da tabela T
    - A chave *x* é, de alguma forma, transformada em um número natural
    - O valor *f* (*x*) (o *hash code*) é utilizado para armazenar *x* na tabela *T*
    - f(*x*) deve estar no intervalo [0,M-1]

Hash: Ilustração da idéia básica (inserção)



End.	Chave
00	Ciclano
01	
02	
03	Fulano

Tabela com M = 4

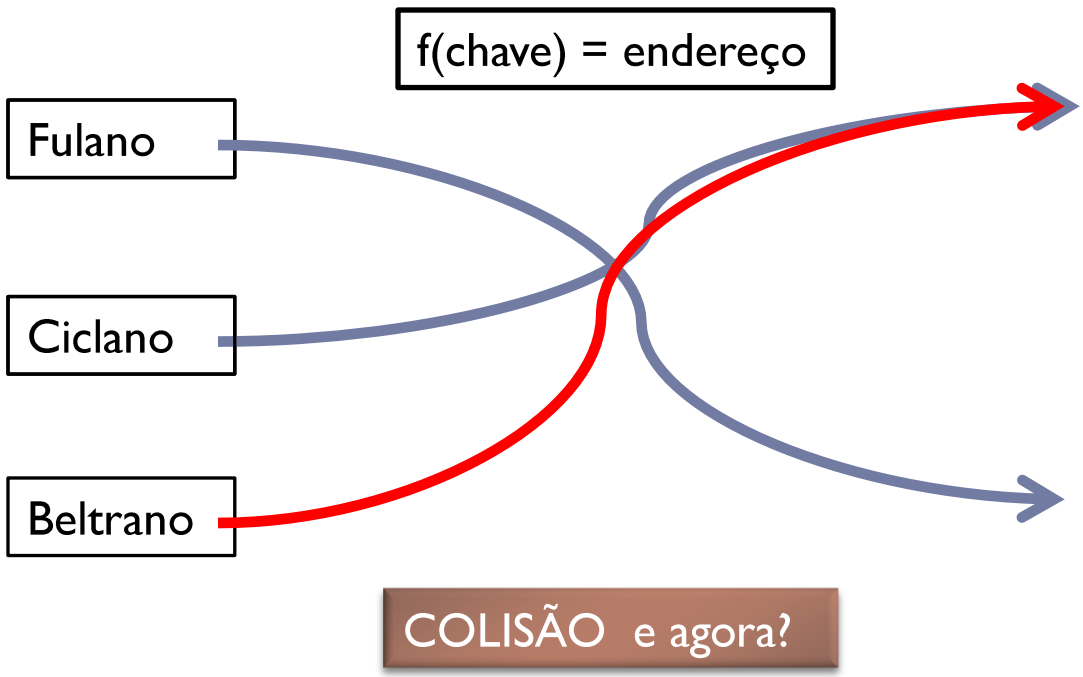
Funções Hashing

- Que tipo de operadores podemos usar em f(x) para assegurar que seu resultado seja sempre <M ?
  - "Resto" (%) operador base para um dos métodos mais usados para f(x).
  - x % M está no intervalo [0,M-1]
- Método da Divisão
  - f(x) = x % M;
  - Uma chave *x* é mapeada em um dos *M* endereços da tabela, calculando o resto da divisão de *x* por *M*
- Método da Multiplicação
  - f(x) = ⌊M ((x · A) % 1)⌋;
  - A é uma constante entre 0 e 1 e ⌊ ⌋ é a função piso
  - Knuth sugere: $A = (\sqrt{5} - 1)/2$

Hash: Vantagens e limitações

- Eficiência
  - Em várias circunstâncias, não é difícil garantir execução em tempo constante,
    - i.e. tempo não é afetado pelo número de chaves existentes.
- Hash não resolve todos os problemas!?!
  - Não convém criar uma posição na tabela para cada possível valor de chave
  - M é o total de chaves que se espera armazenar
  - Qual a principal implicação disso?

Hashing & Colisão



End.	Chave
00	Ciclano
01	
02	
03	Fulano

Tabela com M = 4

Colisões

- Motivos
  - Função de espalhamento não ser perfeita
  - Há mais valores de chaves do que posições na tabela
- *São especialmente agravados quando a função definida pelo programador não consegue tratar padrão matemático presente na entrada (vício)*
- Para melhor entender o problema, resolve o seguinte exercício:
  - Indexe as chaves 12,24,36 e 48 em duas tabelas hash usando o método da divisão
    - M1=6, isto é, T1[0,5]
    - M2=7, isto é, T2[0,6]
  - Para cada tabela, contabilize o número de colisões

Mitigando Colisões

- De forma geral, não é possível assegurar inexistência de colisões. Resta-nos:
  - 1. Tentar diminuir o número de colisões (projetar melhores funções)
  - 2. Tratar colisões
- Alguma sugestão?

Hashing: Mitigando Colisões

- O uso de números primos para M, é recomendado para ajudar a mitigar a ocorrência de colisões especialmente quando há padrões matemáticos entre as chaves.
- Números primos dificultam a formação de padrões em posições da tabela.
- Logo, tende a melhorar a distribuição e diminuir conflitos
  - Ex.: Se são necessárias M chaves, e M não é primo, usar o menor primo maior que M

Tratamento Efetivo de Colisões

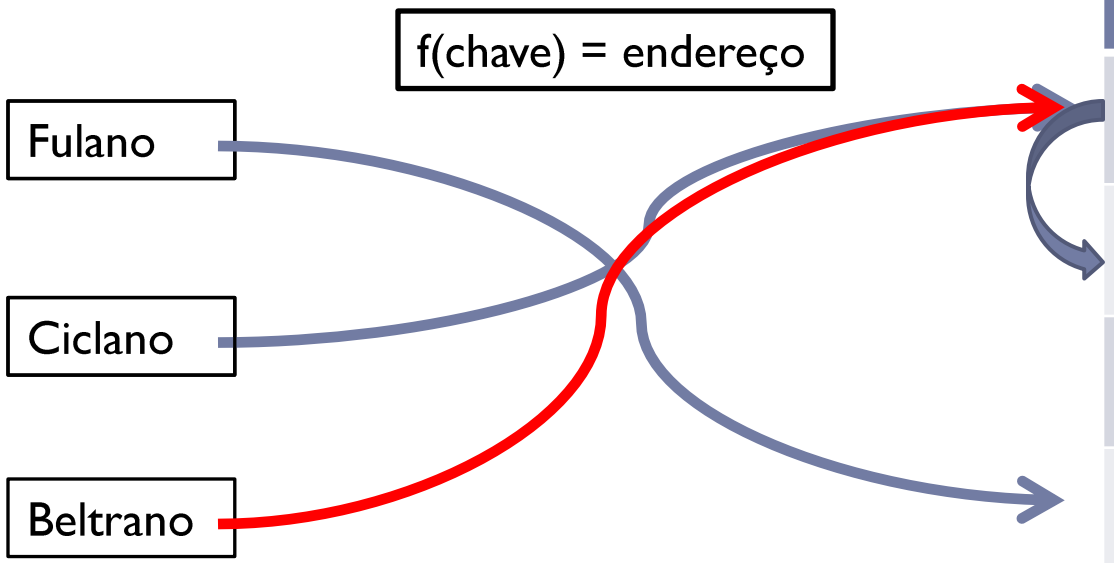
- Mesmo usando números primos, não podemos garantir que não haverá colisões.
- Colisões, se não tratadas, levam ao descarte de chaves
  - i.e. Sobre-escrever a chave "Ciclano" ou rejeitar a chave "Beltrano"
- Como tratar colisão, i.e. Nem sobre-escrever nem descartar?
  - Endereçamento Aberto
  - Listas Encadeadas

Endereçamento Aberto

- Quando há colisão, procura-se por uma nova posição através do cálculo de um novo endereço para inserção na tabela.
  - "Re-hashing"
- Quando uma chave *x* é endereçada na posição *f*(*x*) que esta já está ocupada, outras posições vazias (*na própria tabela*) são procuradas.
- Há diferentes estratégias para o novo cálculo
  - Linear probing (tentativa linear)
  - Double hashing (tentativa dupla)

Endereçamento Aberto: *Linear Probing*

- Se a posição *i* da tabela já está ocupada, procura-se pela próxima posição livre a partir de i.



End.	Chave
00	Ciclano
01	Beltrano
02	
03	Fulano

Tabela com M = 4

Endereçamento Aberto: *Linear Probing*

- Sondagem (ou tentativa) linear:
  - Primeira tentativa: f(x) = x % M
  - Incremento em f(x) após 1° falha: f(x) = (x+1) % M
  - Incremento em f(x) após 2° falha: f(x) = (x+2) % M
  - Incremento em f(x) após 3° falha: f(x) = (x+3) % M
  - ...

Endereçamento Aberto: *Linear Probing*

- Sondagem (ou tentativa) linear:
  - Primeira tentativa: f(x) = x % M
  - Incremento em f(x) após 1° falha: f(x) = (x+1) % M
  - Incremento em f(x) após 2° falha: f(x) = (x+2) % M
  - Incremento em f(x) após 3° falha: f(x) = (x+3) % M
  - ...

```
for(incr = 0;  incr < M; incr++)
{
  fx = (chave + incr) % M;
  if (v[fx] == -1 || v[fx] == chave)
    break; //achei lugar! Saia e insira!
}
assert(incr < M); //checa hash mal projetado
v[fx] = chave;
```

Endereçamento Aberto: *Quadratic Probing*

- Sondagem (ou tentativa) linear:
  - Primeira tentativa: f(x) = x % M
  - Incremento em f(x) após 1° falha: f(x) = (x+1*1) % M
  - Incremento em f(x) após 2° falha: f(x) = (x+2*2) % M
  - Incremento em f(x) após 3° falha: f(x) = (x+3*3) % M
  - ...

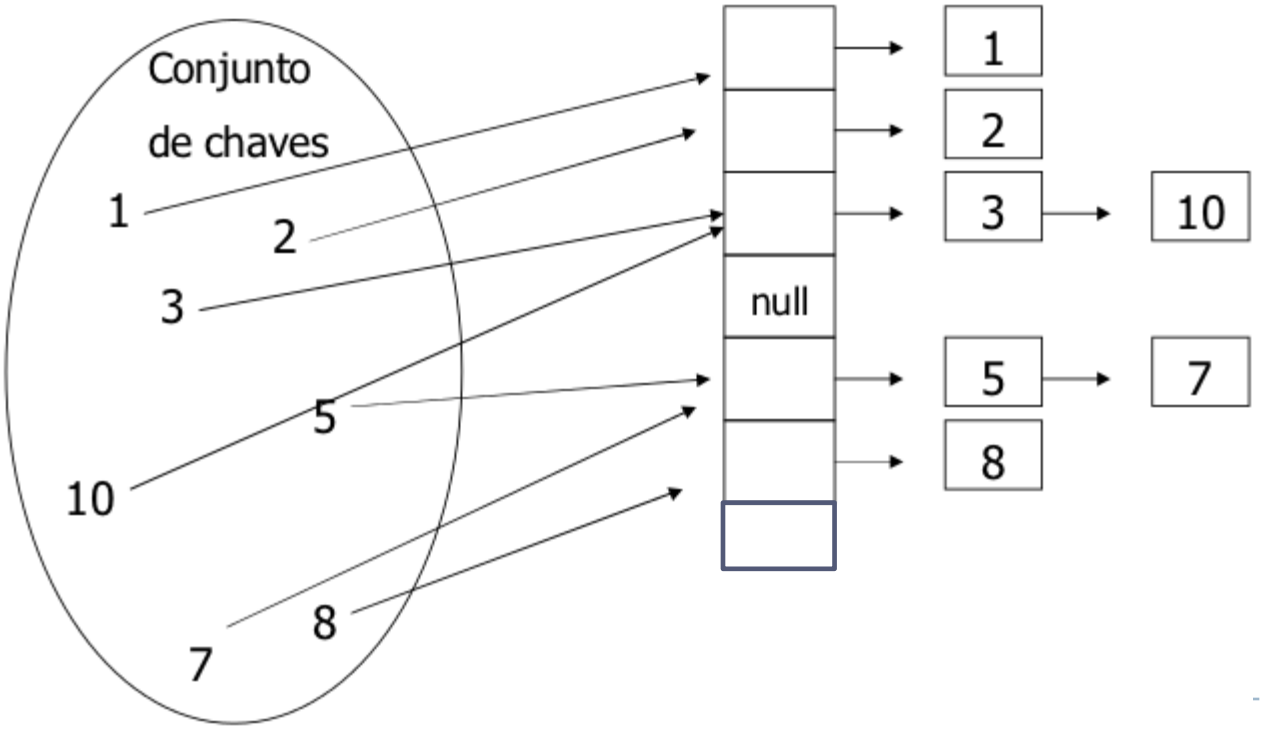
Endereçamento Aberto: *Quadratic Probing*

- Sondagem (ou tentativa) linear:
  - Primeira tentativa: f(x) = x % M
  - Incremento em f(x) após 1° falha: f(x) = (x+1*1) % M
  - Incremento em f(x) após 2° falha: f(x) = (x+2*2) % M
  - Incremento em f(x) após 3° falha: f(x) = (x+3*3) % M
  - ...

```
for(incr = 0;  incr < M; incr++)
{
  fx = (chave + incr*incr) % M;
  if (v[fx] == -1 || v[fx] == chave)
    break; //achei lugar! Saia e insira!
}
assert(incr < M); //checa hash mal projetado
v[fx] = chave;
```

Hash com Listas Encadeadas

- Cada posição na tabela aponta para uma lista encadeada dinâmica.
- **Chaves são armazenadas somente nas listas**
- Listas não unitárias = houve conflito.



h(3)=h(10) e h(7)=h(5)

Hash com Listas Encadeadas

- Para listas encadeadas, pode-se admitir que o total n de chaves seja maior do que o total M de endereços se a divisão n/M for limitada a um valor constante
  - Ex 4 < n/m <11 (Sedgewick)

Hashing: Exemplos de Aplicação

- Tabela de Símbolos em Compiladores.
  - símbolos declarados e.g. Variáveis, frequentemente precisam ser consultados
    - E.g. Para verificar o tipo de dado da variável
- Tabela de Roteamento em Roteadores da Internet.
  - E.g. : Calcular o próximo destino do pacote dado seu endereço de final.
- Índices de Pesquisa em Bancos de Dados
  - Retorno de consultas ao Banco.
- Muitas outras aplicações em computação!