Algoritmos Estrutura de Dados

Marcelo Lobosco

DCC/UFJF

Estruturas de Dados

Parte 2 - Aula 06

Agenda

- Estruturas de Dados Elementares
  - Tabela de Espalhamento

Hash Tables

- Muitas aplicações exigem apenas operações de dicionário: INSERT, SEARCH e DELETE
- Tabela hash permite implementar busca por elementos com custo baixo, O(1), sob hipóteses razoáveis
  - Mas custo do pior caso continua, $\Theta(n)$
- Tabela hash é uma generalização da noção de arranjo
  - Em um arranjo comum, armazenamos elemento cuja chave é k na posição k do arranjo
  - Dada chave k, encontramos elemento procurando-o na k-ésima posição do arranjo: endereçamento direto
  - Endereçamento direto é aplicável quando podemos alocar um arranjo com uma posição para cada chave possível

Hash Tables

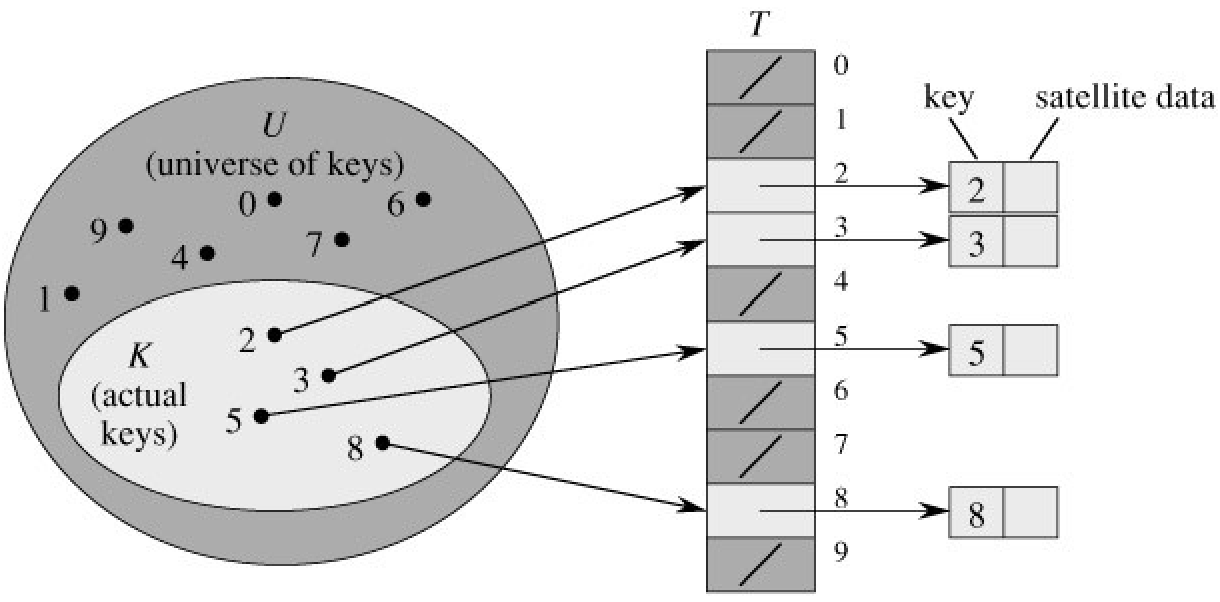
- Usamos tabelas hash quando não queremos, ou não podemos, alocar um arranjo com uma posição para cada chave possível
  - Usamos tabela hash quando número de chaves realmente armazenada é pequeno em relação ao número total de chaves possíveis
  - Tabela hash é um arranjo, mas ela tipicamente usa um tamanho proporcional ao número de chaves a serem armazenadas, ao invés do número de todas as chaves possíveis
  - Dada chave k, não a usamos como índice no vetor
  - Ao invés disso, computa-se uma função de k, e este valor é então usado para indexar arranjo: função de espalhamento (hash)

Tabelas de Endereço Direto

- Funciona bem quando universo de chaves é pequeno
- Suponha um cenário onde
  - Deve-se manter um conjunto dinâmico
  - Cada elemento tem uma chave definida a partir do universo U={0,1,...,m-1}, onde m não é um valor muito grande
  - Não existem dois elementos com a mesma chave

Tabelas de Endereço Direto

- Para representar conjunto dinâmico, usamos arranjo ou tabela de endereçamento direto T[0..m-1], na qual cada *slot* ou posição corresponde a uma chave de U



Tabelas de Endereço Direto

- Posição k aponta para elemento no conjunto com chave k.
  - Se conjunto não contém nenhum elemento com chave k, então T[k]=NIL

DIRECT-ADDRESS-SEARCH(T, k)

return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)

$T[key[x]] \leftarrow \text{NIL}$

Tabelas de Endereço Direto

- Custo das operações: O(1)
- Para algumas aplicações, elementos no conjunto dinâmico podem ser armazenados na própria tabela de endereçamento direto
- Própria armazenagem da chave desnecessária, pois chave é igual a índice
  - Contudo, se chave não armazenada, devemos ter algum modo de saber se posição está vazia

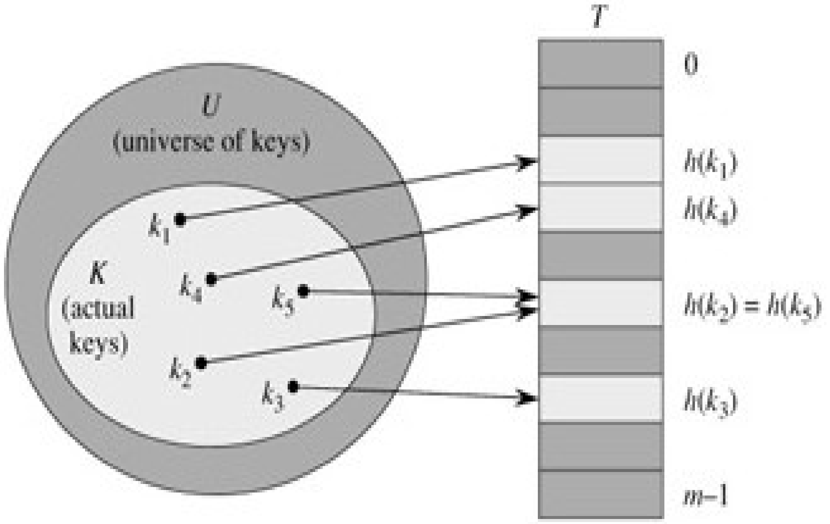
Tabelas Hash

- Dificuldade com endereçamento direto é óbvia: se universo U é grande, armazenamento de tabela de tamanho de U pode se tornar impraticável ou impossível
- Em geral, conjunto K de chaves realmente armazenadas é pequeno comparado com U, logo maior parte do espaço alocado em T é desperdiçado
- Quando K é muito menor do que U, tabela hash requer muito menos espaço do que tabela de endereçamento direto
- Pode reduzir espaço necessário para $\Theta(|K|)$
- Tempo de busca médio continua em $O(1)$

Tabelas Hash

- Idéia: ao invés de armazenar elemento com chave k no slot k, utiliza-se função h e armazena-se elemento no slot h(k)
  - Chamamos h de função de espalhamento (hash)
  - $h:U \rightarrow \{0,1,...,m-1\}$, logo h(k) corresponde a um número de um slot existente em T
  - Dizemos que elemento com chave k mapeia para posição h(k)

Tabelas Hash

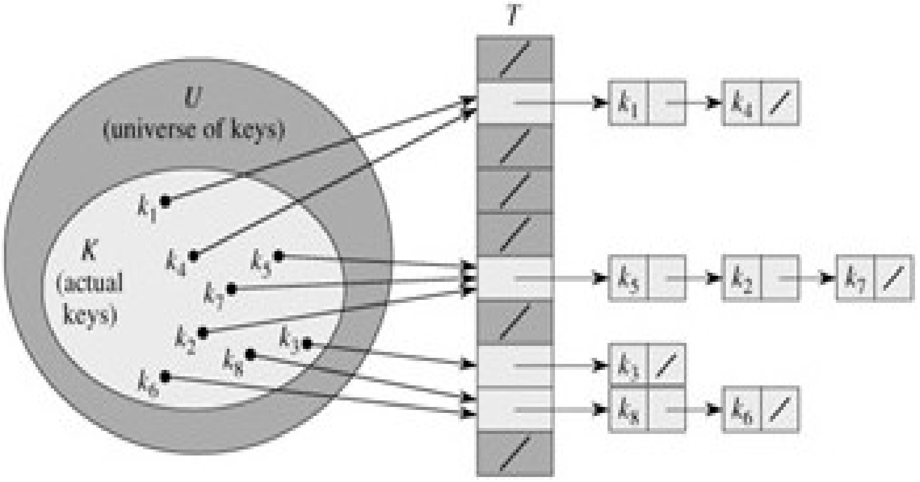


Tabelas Hash

- Colisão ocorre quando duas ou mais chaves mapeiam para o mesmo slot
  - Pode ocorrer quando existem mais chaves possíveis do que *slots* (|U|>m)
  - Para um dado conjunto K de chaves com |K| <=m, pode ou não ocorrer.
  - Definitivamente ocorre se |K| > m
  - Logo, temos de estar preparados para lidar com colisões em todos os casos
  - Vamos examinar dois métodos para lidar com colisões: encadeamento e enderecamento aberto

Resolução por Encadeamento

- Elementos que mapeiam para o mesmo *slot* colocados em uma lista encadeada



Resolução por Encadeamento

- Slot j contém ponteiro para cabeça da lista que contém todos os elementos armazenados que mapeiam em j
  - Se não existem tais elementos, slot j contém NIL

```
CHAINED-HASH-INSERT(T, x)
   insert x at the head of list T[h(key[x])]

CHAINED-HASH-SEARCH(T, k)
   search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
   delete x from the list T[h(key[x])]
```

Resolução por Encadeamento

- Complexidade das operações
  - Inserção: no pior caso, O(1)
  - Busca: custo proporcional ao comprimento da lista de elementos no slot h(k)
  - Eliminação: se lista duplamente encadeada, custo do pior caso é O(1)

Resolução por Encadeamento

- Análise do hash com encadeamento
  - Quanto tempo demora para encontrar um elemento com chave k, ou determinar que não existe elemento com aquela chave?
  - Análise em termos do fator de carga $\alpha = n/m$, onde n é o número de elementos na tabela e m o número de *slots* na tabela
  - Fator representa número médio de elementos por lista encadeada
  - Podemos ter valores de α menor, igual a ou maior que 1

Resolução por Encadeamento

- Pior caso ocorre quando todas as n chaves mapeiam para um mesmo *slot*
  - Lista de tamanho n criada
  - Pior caso para busca igual a $\Theta(n)$, mais o tempo para calcular a função de espalhamento
  - Evidente que hash não é usado pelo seu desempenho no pior caso
  - Caso médio depende de quão bem a função de espalhamento distribui as chaves entre os *slots*
  - Vamos focar no desempenho do caso médio

Resolução por Encadeamento

- Assuma hash uniformemente simples: qualquer elemento dado tem igual probabilidade de efetuar o mapeamento para qualquer uma das m posições
  - Para j = 0, 1, ..., m-1, vamos denotar o comprimento da lista T[j] por n_j.
  - Logo $n = n_0 + n_1 + ... + n_{m-1}$
  - Valor médio de n_j é $E[n_j] = \alpha = n/m$
  - Assuma ainda que podemos computar função de espalhamento em tempo O(1), logo o tempo para buscar elemento com chave k depende do comprimento $n_{h(k)}$ da lista T[h(k)]

Resolução por Encadeamento

- Vamos considerar dois casos
  - Se tabela hash não contém elemento com chave k, então a busca não é bem-sucedida
  - Se a tabela contém elemento com chave k, então a busca é bem sucedida

Resolução por Encadeamento

- Caso em que busca é mal-sucedida
  - Teorema: Busca mal-sucedida tem tempo esperado de $\Theta(1 + \alpha)$
  - Prova: Sob hipótese de hash uniforme simples, qualquer chave ainda não mapeada na tabela tem igual probabilidade de efetuar o hash para qualquer das m posições
  - Para buscar sem sucesso chave k, precisamos buscar até o fim da lista T[h(k)]
  - Esta lista tem comprimento $E[n_{h(k)}]=\alpha$. Logo número de elementos a pesquisar é igual a α.
  - Adicionando tempo para computar função de espalhamento, chegamos a $\Theta(1+\alpha)$

Resolução por Encadeamento

- Caso em que busca é bem-sucedida
  - Tempo esperado para busca também de $\Theta(1 + \alpha)$
  - Circunstâncias ligeiramente diferentes de uma busca mal-sucedida
  - Probabilidade de lista ser pesquisada é proporcional ao número de elementos que ela contém

Resolução por Encadeamento

- Se número de posições da tabela proporcional ao número de elementos armazenados na tabela, temos $n = O(m)$.
  - Consequentemente $\alpha = n/m = o(m)/m = O(1)$
  - Assim pesquisa demora tempo constante
  - Como inserção e remoção demoram tempo $O(1)$ quando usamos listas duplamente encadeadas, todas as operações de dicionário levam tempo $O(1)$ em média

Função de Espalhamento

- O que torna uma função de espalhamento de boa qualidade?
  - Satisfaz aproximadamente à hipótese do hash uniforme simples
  - Na prática, não é possível satisfazer essa condição por não ser possível conhecer a distribuição de probabilidades segundo a qual as chaves são obtidas, e as chaves não podem ser obtidas de forma independente
  - Heurísticas usadas com frequência, baseadas no domínio das chaves, para criar função de espalhamento

Função de Espalhamento

- Chaves como números naturais
  - Funções de espalhamentos assumem que chaves são números naturais
  - Se chaves não são números naturais, devemos encontrar modo de interpretá-las como números naturais
  - Por exemplo, suponha string CLRS
    - Valores em ASCII: C=67, L=76, R=82, S=83
    - Tabela ASCII com 128 valores
    - Logo $67*128^3 + 76*128^2 + 82*128^1 + 83*128^0 = 141764947$

Função de Espalhamento

- Método da Divisão
  - h(k) = k mod m
  - m = tamanho da tabela
  - Exemplo: m = 20 e k = 91, h(k) = 11
  - Vantagem: rápido, uma vez que requer somente uma operação por divisão
  - Desvantagem: devemos evitar certos valores de m

Função de Espalhamento

- Método da Divisão
  - Potências de 2 são ruins. Se $m = 2^p$ para inteiro p, então h(k) é apenas o conjunto dos p bits menos significativos de k
  - Se k é uma string interpretada na base 2^p, então $m = 2^p - 1$ é ruim: permutar caracteres na string não altera o valor do seu hash
  - Boa escolha para m: primo não muito próximo a potência exata de 2

Função de Espalhamento

- Método da Multiplicação
  - Escolha constante A no intervalo 0 < A < 1
  - Multiplique chave k por A
  - Extraia a parte fracionária de kA
  - Multiplique a parte fracionária por m
  - Tome o piso do resultado
  - $h(k) = \lfloor m(k\ A \bmod 1) \rfloor$, onde k A mod 1 é a parte fracional de kA

Função de Espalhamento

- Método da Multiplicação
  - Desvantagem: mais lento que método da divisão
  - Vantagem: valor de m não é crítico
  - Método funciona para qualquer valor de A, mas funciona melhor para alguns valores do que para outros
  - Knuth sugere usar $A \approx (\sqrt{5} - 1)/2 = 0,6180339887...$

Espalhamento Universal

- Suponha que adversário malicioso, que escolhe as chaves a serem mapeadas, tenha visto seu código e conheça sua função de espalhamento
  - Poderia escolher chaves tal que todas mapeiam para mesmo slot
  - Leva a comportamento do pior caso
  - Um modo de derrotar adversário é usar função diferente de espalhamento a cada vez que programa executado
  - Chamado espalhamento universal
  - Função de espalhamento escolhida aleatoriamente de um conjunto de bons candidatos

Espalhamento Universal

- Projeto de classe universal de funções de espalhamento
  - Escolher primo p suficientemente grande para que toda chave k esteja no intervalo 0 a p-1
  - $h_{a,b}$ = ((ak + b) mod p) mod m
  - a e b escolhidos aleatoriamente
  - Exempo, com p=17 e m=6, temos $h_{3,4}$ (8) = ((3*8 + 4) mod 17) mod 6 = 5

Hash Tables
Endereçamento Aberto

- Trata-se de uma alternativa ao encadeamento para lidar com colisões
- Ideia: armazenar todas as chaves na própria tabela *hash*
  - Cada *slot* contém ou uma chave ou o valor NIL
- Para buscar um elemento de chave k
  - Computa-se h(k) e examina-se o *slot* h(k). Exame chamado de sondagem
  - Se slot h(k) contém chave k, busca é bem-sucedida. Se slot contém NIL, busca é mal-sucedida
  - Terceira possibilidade: slot h(k) contém uma chave que não é k. Neste caso, computa-se o índice de algum outro *slot*, baseado em k e no valor de sondagem atual. Busca para quando k ou valor NIL encontrado

Endereçamento Aberto

- Assim precisamos que sequência de slots sondados seja uma permutação dos *slots* de T (0,1,...,m-1)
  - De modo que todos os *slots* sejam examinados, caso necessário, e que os mesmos não sejam examinados mais de uma vez
  - Logo, a função de espalhamento toma a seguinte forma:
    - H: U x {0, 1, ..., m-1}→ {0, 1, ..., m-1}
  - Para inserir, agimos como se estivéssemos buscando, mas inserindo dado no primeiro *slot* NIL encontrado

Endereçamento Aberto

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k, i)
        if T[j] = k
          then return j
        i ← i + 1
  until T[j] = NIL or i = m
return NIL
```

```
HASH-INSERT(T, k)
i ← 0
repeat j ← h(k, i)
        if T[j] = NIL
          then T[j] ← k
                return j
          else i ← i + 1
  until i = m
error "hash table overflow"
```

Endereçamento Aberto

- Caso de remoção mais complexo
  - Não podemos simplesmente colocar NIL no *slot* que contém a chave que desejamos remover
  - Suponha que queiramos remover chave k de *slot* j
  - Suponha ainda que, pouco depois de inserida chave k, chave k' também inserida, e em seu processo de inserção sondamos *slot* j
  - O que ocorre se colocarmos NIL em j? Como iremos buscar k'?

Endereçamento Aberto

- Durante busca, sondaríamos *slot* j antes de sondar *slot* na qual chave k está eventualmente armazenada
- Logo, busca seria malsucedida, ainda que k' ainda esteja na tabela
- Solução: usar um valor especial DELETED ao invés de NIL quando *slot* marcado para remoção
  - Busca trataria *slot* com DELETED como se neste estivesse armazenada chave que não casa com valor sendo buscado
  - Inserção trataria *slot* com DELETED como vazio
  - Desvantagem é que tempo de busca não mais dependente do fator de carga α

Endereçamento Aberto

- Como calcular sequência de sondagem?
- Situação ideal é *hashing* uniforme: cada chave com igual probabilidade de ter qualquer das m! permutações de (0,1,...,m-1) como sua sequência de sondagem
- Novamente difícil implementar verdadeiro *hash* uniforme
- Na prática utilizadas aproximações adequadas
- Contudo, nenhuma das técnicas consegue produzir todas as m! sequências de sondagem
- Farão uso de funções auxiliares de espalhamento, que mapeiam $U \rightarrow \{0, 1, ..., m-1\}$

Endereçamento Aberto

- Sondagem linear
  - Dada função auxiliar de espalhamento h', sequência de sondagem inicia-se no *slot* h'(k) e continua sequencialmente através da tabela, retornando para posição 0 quando posição m-1 sondada
  - Dada chave k e número de sondagem i ($0 \leq i < m$), $h(k,i) = (h'(k) + i) \bmod m$

Endereçamento Aberto

- Sondagem linear
  - Sofre de problema de agrupamento primário
  - Longas sequências de posições ocupadas construídas, aumentando tempo médio de pesquisa
  - Posição vazia precedida por i posições seguidas preenchidas tem probabilidade (i+1)/m
  - Sequência de posições ocupadas tende a ficar mais longa, aumentando tempo médio de pesquisa

Endereçamento Aberto

- Sondagem quadrática
  - Assim como em sondagem linear, sequência de sondagem inicia-se em h'(k)
  - Entretanto, diferentemente de sondagem linear, sondagem quadrática salta ao redor da tabela de acordo com função quadrática do número de sondagem
  - $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, onde c_1, $c_2 \neq 0$ são constantes
  - Valores de c_1, c_2 e m devem ser escolhidos de modo a garantir uma permutação completa de (0,1,...,m-1)

Endereçamento Aberto

- Sondagem quadrática
  - Pode ocorrer problema de agrupamento secundário
  - Se duas chaves distintas tem o mesmo valor de h', então elas terão a mesma sequência de sondagem

Endereçamento Aberto

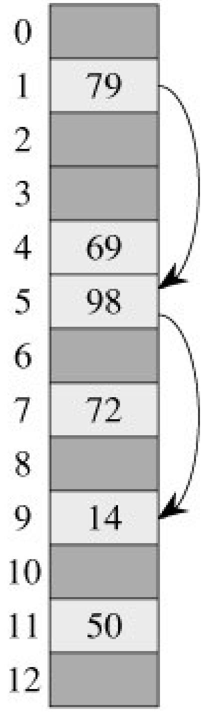
- Espalhamento duplo
  - Usa duas funções auxiliares, h_1 e h_2
  - h_1 dá sondagem inicial e h_2 dá sequência posterior de sondagem
  - $h(k,i) = (h_1(k) + i\ h_2(k)) \bmod m$
  - Importante que $h_2(k)$ e tamanho m da tabela sejam primos entre si para que tabela toda seja pesquisada

Endereçamento Aberto

- Espalhamento duplo
  - Duas formas de assegurar condição
  - m pode ser potência de 2, e h_2 projetado para produzir números impares ou
  - m pode ser primo e $1 < h_2(k) < m$
  - Por exemplo, $h_1(k) = k \bmod m$, e $h_2(k) = 1 + (k \bmod m')$, onde m' escolhido como valor ligeiramente menor que m, p.ex., m-1
  - Desempenho de *hash* duplo muito próximo do desempenho do esquema ideal de hash uniforme

Endereçamento Aberto

- Espalhamento duplo



Endereçamento Aberto

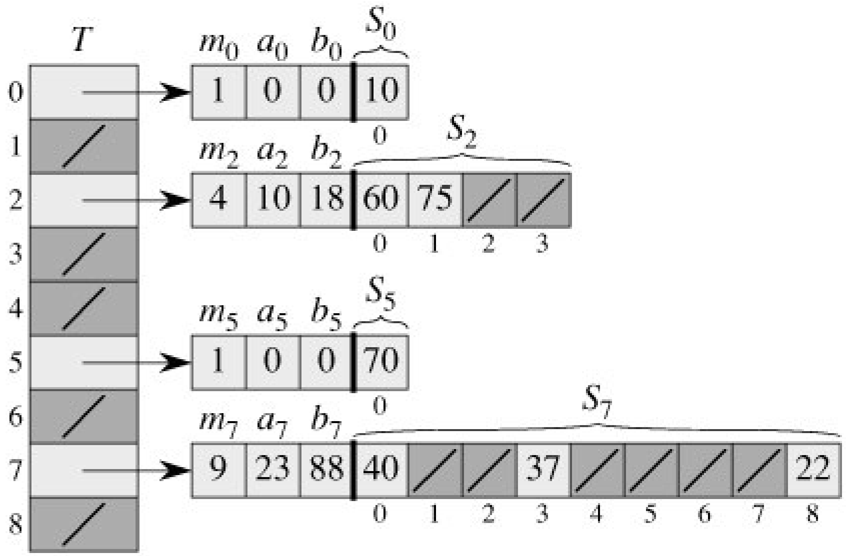
- Análise do endereçamento aberto
  - Supondo-se espalhamento uniforme e
  - Tabela nunca enche completamente, logo $0 \leq n < m$, de modo que $0 \leq \alpha < 1$ e
  - Sem remoção
  - Número esperado de sondagens em uma pesquisa malsucedida é no máximo $1/(1-\alpha)$
  - Inserção exige no máximo $1/(1-\alpha)$ sondagens
  - Número esperado de sondagens em uma pesquisa bem-sucedida é no máximo $(1/\alpha).\ln 1/(1-\alpha)$

Hash Perfeito

- *Hash* pode ser usado para obter excelente desempenho no pior caso, O(1)
- Desde que conjunto de chaves seja estático, ou seja, nunca se altere
- Ideia é usar espalhamento de dois níveis, com *hash* universal sendo usado em cada nível
- Primeiro nível essencialmente o mesmo do *hash* com encadeamento
- Contudo, pequena tabela de espalhamento secundário S usada no segundo nível

Hash Perfeito

- Para garantir que não haverá colisão no nível secundário, número de elementos em cada nível de S deve ser o quadrado do número de chaves que podem mapear no *slot* j
- Se função de *hash* bem escolhida, armazenamento será O(n)



Próxima aula...

- Classes de Problemas