# Estrutura de Dados II (ED2)

**Aula 25 – Tabelas *Hash***

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides do Professor Eduardo Zambon)

# Introdução

- A tabela *hash* é uma estrutura muito utilizada como implementação de uma tabela de símbolos.
- A tabela *hash* provê uma alternativa eficiente às árvores de busca.
- **Aula de hoje**: mostrar variações de tabelas *hash*.
- **Objetivos**: compreender o funcionamento e as diferentes implementações de tabelas *hash*.

### Referências

**Chapter 14 – Hashing**
*R. Sedgewick*

# Implementações de tabelas de símbolos: sumário

| implementation | guarantee | | | average case | | | ordered ops? |
|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | |
| **sequential search (unordered list)** | $N$ | $N$ | $N$ | $\frac{1}{2} N$ | $N$ | $\frac{1}{2} N$ | |
| **binary search (ordered array)** | $\lg N$ | $N$ | $N$ | $\lg N$ | $\frac{1}{2} N$ | $\frac{1}{2} N$ | ✔ |
| **BST** | $N$ | $N$ | $N$ | $1.39 \lg N$ | $1.39 \lg N$ | $\sqrt{N}$ | ✔ |
| **red-black BST** | $2 \lg N$ | $2 \lg N$ | $2 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | ✔ |

Q: É possível fazer melhor?

# Implementações de tabelas de símbolos: sumário

| implementation | guarantee | | | average case | | | ordered ops? |
|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | |
| **sequential search (unordered list)** | $N$ | $N$ | $N$ | $\frac{1}{2} N$ | $N$ | $\frac{1}{2} N$ | |
| **binary search (ordered array)** | $\lg N$ | $N$ | $N$ | $\lg N$ | $\frac{1}{2} N$ | $\frac{1}{2} N$ | ✔ |
| **BST** | $N$ | $N$ | $N$ | $1.39 \lg N$ | $1.39 \lg N$ | $\sqrt{N}$ | ✔ |
| **red-black BST** | $2 \lg N$ | $2 \lg N$ | $2 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | ✔ |

Q: É possível fazer melhor?

A: Sim, mas com uma forma diferente de acesso aos dados.

# *Hashing*: ideia básica

Salvar os itens em uma tabela indexada pela chave, aonde o índice é uma função da chave.

Função *hash*: método para computar o índice no *array* a partir da chave.

# *Hashing*: ideia básica

Salvar os itens em uma tabela indexada pela chave, aonde o índice é uma função da chave.

Função *hash*: método para computar o índice no *array* a partir da chave.

Dificuldades:

- Calcular a função *hash*.
- Resolução de colisão: algoritmo e estrutura de dados para lidar com chaves que são mapeadas para o mesmo índice.

# *Hashing*: ideia básica

Salvar os itens em uma tabela indexada pela chave, aonde o índice é uma função da chave.

Função *hash*: método para computar o índice no *array* a partir da chave.

Dificuldades:

- Calcular a função *hash*.
- Resolução de colisão: algoritmo e estrutura de dados para lidar com chaves que são mapeadas para o mesmo índice.

Exemplo clássico de relação de compromisso espaço-tempo:

- Sem limitação de espaço: função *hash* trivial com chave como índice.
- Sem limitação de tempo: resolução de colisão trivial com busca linear.
- Com limitação de espaço e tempo: *hashing* (mundo real).

Parte I

# Funções Hash

# Calculando a função *hash*

Objetivo ideal: embaralhar as chaves uniformemente para produzir um índice.

- Função deve ser eficientemente computável.
- Cada índice da tabela igualmente provável para cada chave.

Problema extensivamente pesquisado, mas ainda é problemático em aplicações reais.

# Calculando a função *hash*

Objetivo ideal: embaralhar as chaves uniformemente para produzir um índice.

- Função deve ser eficientemente computável.
- Cada índice da tabela igualmente provável para cada chave.

Problema extensivamente pesquisado, mas ainda é problemático em aplicações reais.

Exemplo: Números de telefone com DDD:

- Ruim: usar os três primeiros dígitos – todos os números com o mesmo DDD mapeados para o mesmo índice.
- Menos pior: usar os últimos três dígitos.

# Calculando a função *hash*

Objetivo ideal: embaralhar as chaves uniformemente para produzir um índice.

- Função deve ser eficientemente computável.
- Cada índice da tabela igualmente provável para cada chave.

Problema extensivamente pesquisado, mas ainda é problemático em aplicações reais.

Exemplo: Números de telefone com DDD:

- Ruim: usar os três primeiros dígitos – todos os números com o mesmo DDD mapeados para o mesmo índice.
- Menos pior: usar os últimos três dígitos.

Desafio de implementação: é necessária uma função *hash* diferente para cada tipo de chave.

# Funções *hash*

Solução 1: considerar a chave como um inteiro, usar uma tabela de tamanho primo $M$.

- Função *hash*: `h(k) = k % M`.

# Funções *hash*

Solução 1: considerar a chave como um inteiro, usar uma tabela de tamanho primo $M$.

- Função *hash*: `h(k) = k % M`.

Exemplo: chaves com 4 caracteres ASCII, tamanho $M = 101$.

- Valores das 26 letras minúsculas:
  `a = 0x61`, `b = 0x62`, `c = 0x63`, `d = 0x64`, ...
- $26^4 \sim 500$K chaves distintas.
- Tabela tem $M = 101$ índices: $\sim 5$K chaves por índice.
- Muitas chaves, tabela pequena $\Rightarrow$ muitas colisões!

# Funções *hash*

Solução 1: considerar a chave como um inteiro, usar uma tabela de tamanho primo $M$.

- Função *hash*: `h(k) = k % M`.

Exemplo: chaves com 4 caracteres ASCII, tamanho $M = 101$.

- Valores das 26 letras minúsculas:
  `a = 0x61, b = 0x62, c = 0x63, d = 0x64, ...`
- $26^4 \sim 500K$ chaves distintas.
- Tabela tem $M = 101$ índices: $\sim 5K$ chaves por índice.
- Muitas chaves, tabela pequena $\Rightarrow$ muitas colisões!

- `dcba` é mapeado para o índice 57.
  `0x64636261 = 1684234849 % 101 = 57`
- `abbc` também é mapeado para o índice 57.
  `0x61626263 = 1633837667 % 101 = 57`

# Funções *hash*

Solução 2: considerar a chave como um inteiro longo, usar uma tabela de tamanho primo *M*.

- Usar a mesma função *hash*: `h(k) = k % M`.
- Calcular o valor usando o método de Horner (abaixo).

# Funções *hash*

Solução 2: considerar a chave como um inteiro longo, usar uma tabela de tamanho primo $M$.

- Usar a mesma função *hash*: `h(k) = k % M`.
- Calcular o valor usando o método de Horner (abaixo).

- `abcd` é mapeado para o índice 11 (`0x61 = 97`).

  ```
  0x61626364 = 256*(256*(256*97+98)+99)+100
  16338831724 % 101 = 11
  ```

# Funções *hash*

Solução 2: considerar a chave como um inteiro longo, usar uma tabela de tamanho primo *M*.

- Usar a mesma função *hash*: `h(k) = k % M`.
- Calcular o valor usando o método de Horner (abaixo).

- `abcd` é mapeado para o índice 11 (`0x61 = 97`).

  ```
  0x61626364 = 256*(256*(256*97+98)+99)+100
  16338831724 % 101 = 11
  ```

- Números muito grandes? Tome o módulo a cada passo:

  ```
  256*97+98  = 24930 % 101 = 84
  256*84+99  = 21603 % 101 = 90
  256*90+100 = 23140 % 101 = 11
  ```

  Pode continuar indefinidamente, para qualquer tamanho.

# Funções *hash*

Solução 2: considerar a chave como um inteiro longo, usar uma tabela de tamanho primo $M$.

- Usar a mesma função *hash*: `h(k) = k % M`.
- Calcular o valor usando o método de Horner (abaixo).

- `abcd` é mapeado para o índice 11 (`0x61 = 97`).

```
0x61626364 = 256*(256*(256*97+98)+99)+100
16338831724 % 101 = 11
```

- Números muito grandes? Tome o módulo a cada passo:

```
256*97+98  = 24930 % 101 = 84
256*84+99  = 21603 % 101 = 90
256*90+100 = 23140 % 101 = 11
```

  Pode continuar indefinidamente, para qualquer tamanho.

- Custo da função *hash*, para uma *string* de tamanho $N$?

# Funções *hash*

Solução 2: considerar a chave como um inteiro longo, usar uma tabela de tamanho primo $M$.

- Usar a mesma função *hash*: `h(k) = k % M`.
- Calcular o valor usando o método de Horner (abaixo).
- `abcd` é mapeado para o índice 11 (`0x61 = 97`).

```
0x61626364 = 256*(256*(256*97+98)+99)+100
1633831724 % 101 = 11
```

- Números muito grandes? Tome o módulo a cada passo:

```
256*97+98  = 24930 % 101 = 84
256*84+99  = 21603 % 101 = 90
256*90+100 = 23140 % 101 = 11
```

  Pode continuar indefinidamente, para qualquer tamanho.

- Custo da função *hash*, para uma *string* de tamanho $N$?
- $N$ operações de soma, multiplicação e módulo.

# Funções *hash*

Melhorando a solução 2: o método de Horner é amplamente usado na prática, e pode ser melhorado com algumas otimizações simples.

# Funções *hash*

Melhorando a solução 2: o método de Horner é amplamente usado na prática, e pode ser melhorado com algumas otimizações simples.

- Otimização: Para gerar uma distribuição mais uniforme, use um multiplicador aleatório para cada dígito.

# Funções *hash*

Melhorando a solução 2: o método de Horner é amplamente usado na prática, e pode ser melhorado com algumas otimizações simples.

- Otimização: Para gerar uma distribuição mais uniforme, use um multiplicador aleatório para cada dígito.
- Simplificação: Basta usar um único multiplicador primo pequeno, por exemplo, 251.

```
uint32_t horner(char *s, int len) {
  uint32_t h = 0;
  for (int i = 0; i < len; i++) {
    h = (251*h + s[i]) % M;
  }
  return h;
}
```

# Funções *hash*

Melhorando a solução 2: o método de Horner é amplamente usado na prática, e pode ser melhorado com algumas otimizações simples.

- Otimização: Para gerar uma distribuição mais uniforme, use um multiplicador aleatório para cada dígito.
- Simplificação: Basta usar um único multiplicador primo pequeno, por exemplo, 251.

```
uint32_t horner(char *s, int len) {
  uint32_t h = 0;
  for (int i = 0; i < len; i++) {
    h = (251*h + s[i]) % M;
  }
  return h;
}
```

- A base agora é um primo e não uma potência de 2: a função gera uma boa dispersão para qualquer valor de $M$.

# Suposição de *hashing* uniforme

Suposição de *hashing* uniforme: a função *hash* mapeia qualquer chave em um índice $i$ entre $0$ e $M - 1$, aonde qualquer valor de $i$ é igualmente provável.

# Suposição de *hashing* uniforme

Suposição de *hashing* uniforme: a função *hash* mapeia qualquer chave em um índice $i$ entre $0$ e $M - 1$, aonde qualquer valor de $i$ é igualmente provável.

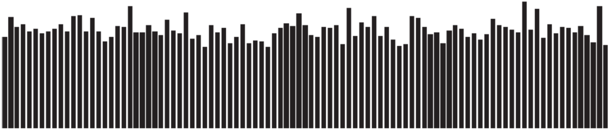
Equivalência no mundo físico: jogar bolas aleatoriamente em $M$ caixas, segundo uma distribuição uniforme discreta.

# Suposição de *hashing* uniforme

Suposição de *hashing* uniforme: a função *hash* mapeia qualquer chave em um índice $i$ entre $0$ e $M - 1$, aonde qualquer valor de $i$ é igualmente provável.

Equivalência no mundo físico: jogar bolas aleatoriamente em $M$ caixas, segundo uma distribuição uniforme discreta.

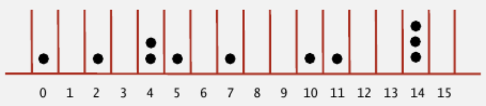

Um cenário real com *hashing* uniforme.



Hash value frequencies for words in Tale of Two Cities (M = 97)

# Suposição de *hashing* uniforme

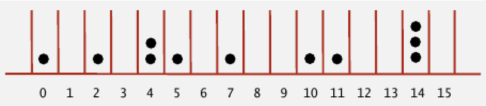
Equivalência no mundo físico: jogar bolas aleatoriamente em $M$ caixas, segundo uma distribuição uniforme discreta.




Relação de 1-1 com o problema de *hashing*: permite usar uma série de problemas clássicos de probabilidade:

# Suposição de *hashing* uniforme

Equivalência no mundo físico: jogar bolas aleatoriamente em $M$ caixas, segundo uma distribuição uniforme discreta.

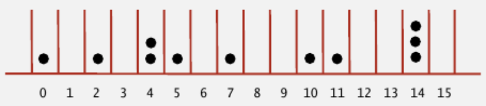



Relação de 1-1 com o problema de *hashing*: permite usar uma série de problemas clássicos de probabilidade:

- Problema dos aniversários: expectativa de 2 bolas na mesma caixa após $\sim \sqrt{\pi M/2}$ lançamentos.

# Suposição de *hashing* uniforme

Equivalência no mundo físico: jogar bolas aleatoriamente em $M$ caixas, segundo uma distribuição uniforme discreta.

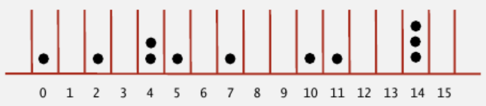



Relação de 1-1 com o problema de *hashing*: permite usar uma série de problemas clássicos de probabilidade:

- Problema dos aniversários: expectativa de 2 bolas na mesma caixa após $\sim \sqrt{\pi M/2}$ lançamentos.
- Problema da coleção de cupons: expectativa que cada caixa tenha $\geq 1$ bola após $\sim M \ln M$ lançamentos.

# Suposição de *hashing* uniforme

Equivalência no mundo físico: jogar bolas aleatoriamente em $M$ caixas, segundo uma distribuição uniforme discreta.




Relação de 1-1 com o problema de *hashing*: permite usar uma série de problemas clássicos de probabilidade:

- Problema dos aniversários: expectativa de 2 bolas na mesma caixa após $\sim \sqrt{\pi M/2}$ lançamentos.
- Problema da coleção de cupons: expectativa que cada caixa tenha $\geq 1$ bola após $\sim M \ln M$ lançamentos.
- Balanceamento de carga: depois de $M$ lançamentos, expectativa que a caixa mais cheia tenha $\Theta(\log M / \log \log M)$ bolas.

# *Hashing* na prática: estudo de colisão

```c
#define M 3571

uint32_t silly(char *s, int len) {
  uint32_t h = 0;
  for (int i = 0; i < len; i++) {
    h += s[i];
  }
  return h % M;
}

uint32_t horner(char *s, int len) { ... } // As before

uint32_t adler(char *s, int len) {// Mark Adler (1995) -> Zlib
  uint32_t s1 = 1;
  uint32_t s2 = 0;
  for (int i = 0; i < len; i++) {
    s1 = (s1 + s[i]) % 65521; // Largest prime number < 2^16
    s2 = (s1 + s2) % 65521;
  }
  return ((s2 << 16) | s1) % M;
}
```

# *Hashing* na prática: estudo de colisão

```c
uint32_t joaat(char *s, int len) { // Jenkins-one-at-a-time
  uint32_t h = 0;                  // Bob Jenkins (1997)
  for (int i = 0; i < len; i++) {
    h += s[i];
    h += (h << 10);
    h ^= (h >> 6);
  }
  h += (h << 3);
  h ^= (h >> 11);
  h += (h << 15);
  return h % M;
}

/*
Test with "/usr/share/dict/cracklib-small" (54.763 entries)
silly  -- Min = 0, Max = 215, Collision rate = 0.324453
horner -- Min = 4, Max =  30, Collision rate = 0.008382
adler  -- Min = 4, Max =  29, Collision rate = 0.007834
joaat  -- Min = 4, Max =  31, Collision rate = 0.008400
*/
```

# Colisões

Colisão: duas chaves distintas mapeadas pela função *hash* para o mesmo índice.

# Colisões

Colisão: duas chaves distintas mapeadas pela função *hash* para o mesmo índice.

- Problema dos aniversários $\Rightarrow$ não é possível evitar colisões sem gastar uma quantidade ridícula de memória.

# Colisões

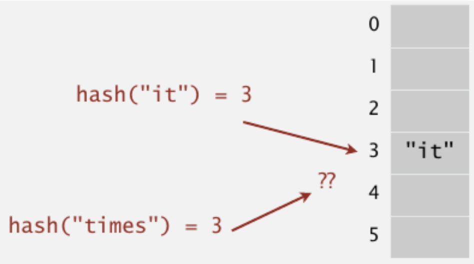
Colisão: duas chaves distintas mapeadas pela função *hash* para o mesmo índice.

- Problema dos aniversários $\Rightarrow$ não é possível evitar colisões sem gastar uma quantidade ridícula de memória.
- Problema da coleção de cupons + balanceamento de carga $\Rightarrow$ as colisões são uniformemente distribuídas.

# Colisões

Colisão: duas chaves distintas mapeadas pela função *hash* para o mesmo índice.

- Problema dos aniversários ⇒ não é possível evitar colisões sem gastar uma quantidade ridícula de memória.
- Problema da coleção de cupons + balanceamento de carga ⇒ as colisões são uniformemente distribuídas.




Desafio: lidar com colisões de forma eficiente.

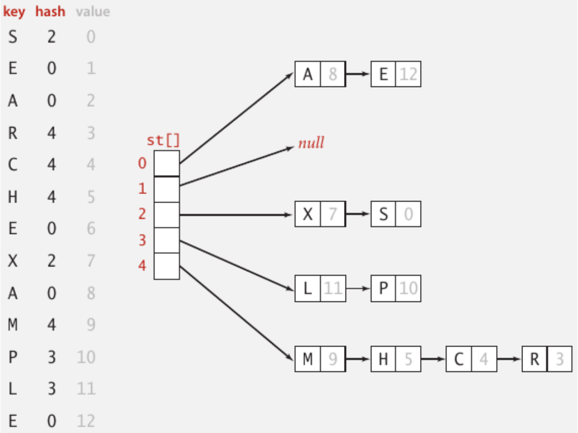
Parte II

# Separate Chaining

# Tabela de símbolos com *separate (open) chaining*

*Array* de listas encadeadas [H.P. Luhn, IBM, 1953]

- *Hash*: mapear inteiro *i* entre 0 e $M - 1$.
- Inserção: inserir no início da *i*-ésima lista.
- Busca: só é necessário buscar a *i*-ésima lista.



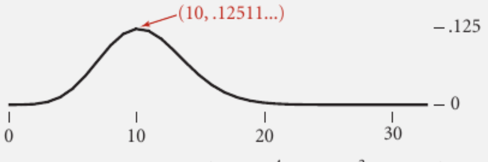
# *Separate chaining*: análise

Proposição: sob a suposição de *hashing* uniforme, a probabilidade que o número de chaves em uma lista (*bucket*) seja $\sim c(N/M)$ é muito próxima de 1.

# *Separate chaining*: análise

Proposição: sob a suposição de *hashing* uniforme, a probabilidade que o número de chaves em uma lista (*bucket*) seja $\sim c(N/M)$ é muito próxima de 1.

Just.: Tamanhos das listas segue uma distribuição binomial.

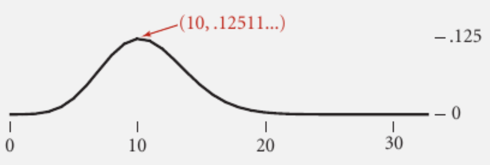


**Binomial distribution** $(N = 10^4, M = 10^3, \alpha = 10)$

# *Separate chaining*: análise

Proposição: sob a suposição de *hashing* uniforme, a probabilidade que o número de chaves em uma lista (*bucket*) seja $\sim c(N/M)$ é muito próxima de 1.

Just.: Tamanhos das listas segue uma distribuição binomial.



Binomial distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

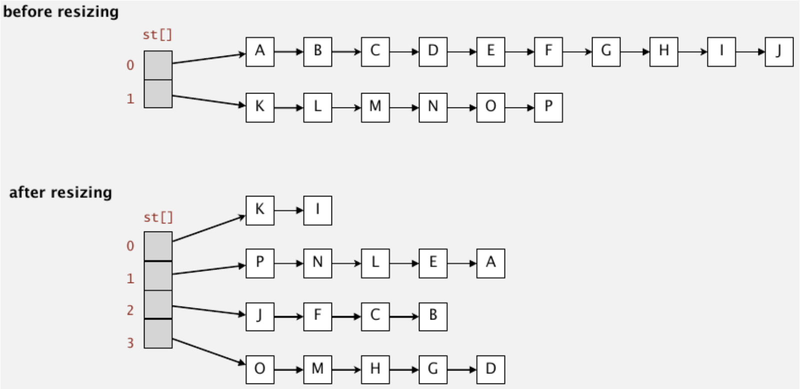
Consequência: número de *probes* para busca/inserção é proporcional a $N/M$. ($M$ vezes mais rápido que busca linear.)

- $M$ muito grande: desperdício de memória.
- $M$ muito pequeno: cadeias (listas) muito longas.
- Escolha típica: $M \sim N/4 \Rightarrow$ ops em tempo constante.

# *Separate chaining*: ajustando o tamanho do *array*

Objetivo: Manter o comprimento médio das listas constante.

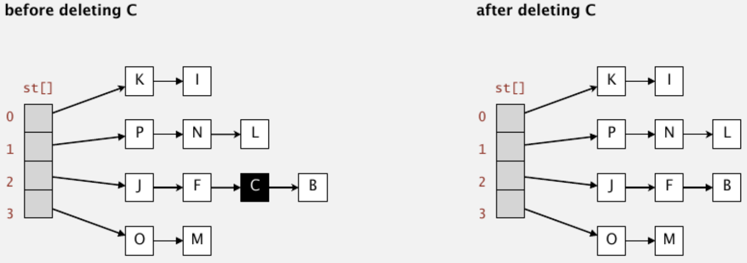
- Quando $N/M \geq 8$: aumentar (duplicar?) o *array*.
- Quando $N/M \leq 8$: diminuir o *array*.
- Todo ajuste requer um *rehash* de todas as chaves.



# *Separate chaining*: remoção

Q: Como remover uma chave (e o valor associado)?

A: Fácil, só é necessário considerar a cadeia contendo a chave.



# Implementações de tabelas de símbolos: sumário

| implementation | guarantee | | | average case | | | ordered ops? |
|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | |
| **sequential search (unordered list)** | $N$ | $N$ | $N$ | ½ $N$ | $N$ | ½ $N$ | |
| **binary search (ordered array)** | $\lg N$ | $N$ | $N$ | $\lg N$ | ½ $N$ | ½ $N$ | ✔ |
| **BST** | $N$ | $N$ | $N$ | $1.39 \lg N$ | $1.39 \lg N$ | $\sqrt{N}$ | ✔ |
| **red-black BST** | $2 \lg N$ | $2 \lg N$ | $2 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | ✔ |
| **separate chaining** | $N$ | $N$ | $N$ | 3-5 * | 3-5 * | 3-5 * | |

* under uniform hashing assumption

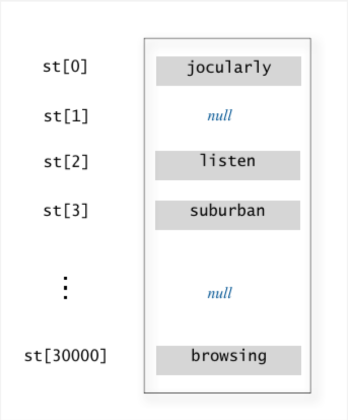
Parte III

# Linear Probing

# Resolução de colisão: *open addressing*

*Open addressing* [Amdahl et al., IBM, 1953]

Quando uma nova chave colide, encontre a próxima posição vazia e insira ali.




linear probing (M = 30001, N = 15000)

# Tabela *hash* com *linear probing*: demo

- *Hash*: mapear a chave em um inteiro $i$ entre 0 e $M - 1$.
- Inserção: insira no índice $i$ se estiver livre, senão tente $i + 1$, $i + 2$, etc.
- Busca: busque no *array* no índice $i$, se estiver ocupado mas não bater a chave, tente $i + 1$, $i + 2$, etc.

- Nota: tamanho $M$ do *array* precisa ser maior que o número de chaves $N$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| st[] | P | M | | | A | C | S | H | L | | E | | | | R | X |

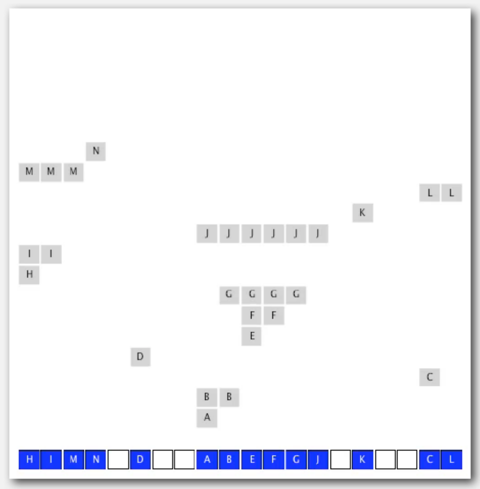
M = 16

Veja o vídeo `34DemoLinearProbingHashTable.mov`.

# *Clustering*

*Cluster:* um bloco contíguo de chaves.

Observação: novas chaves tendem a cair no meio de *clusters*.



# Problema do estacionamento de Knuth

Modelo: carros chegam em uma rua de mão única com $M$ vagas de estacionamento.

Cada um quer uma vaga $i$ aleatória: se vaga $i$ estiver ocupada, tente $i + 1$, $i + 2$, etc.

Q: Qual é o deslocamento médio de um carro?

# Problema do estacionamento de Knuth

Modelo: carros chegam em uma rua de mão única com $M$ vagas de estacionamento.

Cada um quer uma vaga $i$ aleatória: se vaga $i$ estiver ocupada, tente $i + 1$, $i + 2$, etc.

Q: Qual é o deslocamento médio de um carro?


Meio cheio: com $M/2$ carros, o deslocamento médio é $\sim 3/2$.

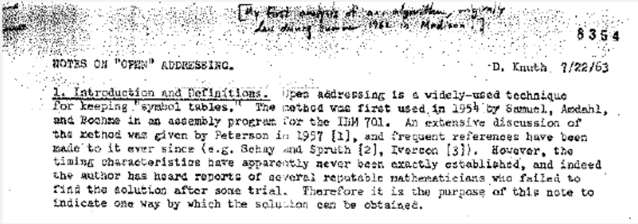
Quase cheio: com $\sim M$ carros, o deslocamento médio é $\sim \sqrt{\pi M/8}$.

# *Linear probing*: análise

Proposição: sob a suposição de *hashing* uniforme, o número médio de *probes* em uma tabela de tamanho $M$ que contém $N = \alpha M$ chaves ($\alpha < 1$) é:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

*search hit* *search miss / insert*

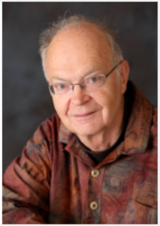


[My first analysis of an algorithm, originally done during Summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

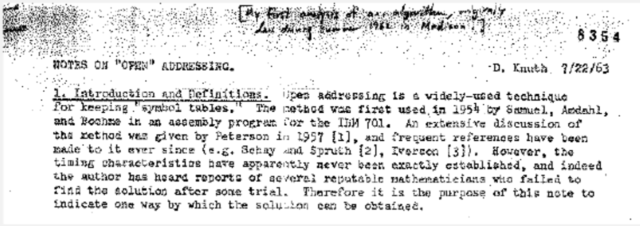
1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Proposição: sob a suposição de *hashing* uniforme, o número médio de *probes* em uma tabela de tamanho $M$ que contém $N = \alpha M$ chaves ($\alpha < 1$) é:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

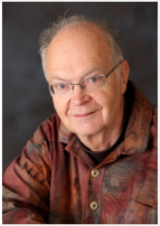
*search hit* — *search miss / insert*


[My first analysis of an algorithm, originally done during Summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. — D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Escolha típica: $\alpha = N/M \sim 1/2 \Rightarrow$ *Hit*: $3/2$ – *Miss*: $5/2$.

Objetivo: Manter a taxa de ocupação do *array* $N/M \leq 1/2$.

- Quando $N/M \geq 1/2$: aumentar o *array*.
- Quando $N/M \leq 1/8$: diminuir o *array*.
- Todo ajuste requer um *rehash* de todas as chaves.

**before resizing**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| keys[] | | E | S | | | R | A | |
| vals[] | | 1 | 0 | | | 3 | 2 | |

**after resizing**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| keys[] | | | | | A | | S | | | | E | | | | R | |
| vals[] | | | | | 2 | | 0 | | | | 1 | | | | 3 | |

Q: Como remover uma chave (e o valor associado)?

A: Requer cuidado, não dá para só apagar chaves no *array*.

**before deleting S**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| keys[] | P | M | | | A | C | S | H | L | | E | | | | R | X |
| vals[] | 10 | 9 | | | 8 | 4 | 0 | 5 | 11 | | 12 | | | | 3 | 7 |

doesn't work, e.g., if hash(H) = 4

**after deleting S ?**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| keys[] | P | M | | | A | C | | H | L | | E | | | | R | X |
| vals[] | 10 | 9 | | | 8 | 4 | | 5 | 11 | | 12 | | | | 3 | 7 |

# Implementações de tabelas de símbolos: sumário

| implementation | guarantee | | | average case | | | ordered ops? |
|---|---|---|---|---|---|---|---|
| | search | insert | delete | search hit | insert | delete | |
| **sequential search (unordered list)** | $N$ | $N$ | $N$ | ½ $N$ | $N$ | ½ $N$ | |
| **binary search (ordered array)** | lg $N$ | $N$ | $N$ | lg $N$ | ½ $N$ | ½ $N$ | ✔ |
| **BST** | $N$ | $N$ | $N$ | 1.39 lg $N$ | 1.39 lg $N$ | $\sqrt{N}$ | ✔ |
| **red-black BST** | 2 lg $N$ | 2 lg $N$ | 2 lg $N$ | 1.0 lg $N$ | 1.0 lg $N$ | 1.0 lg $N$ | ✔ |
| **separate chaining** | $N$ | $N$ | $N$ | 3-5 * | 3-5 * | 3-5 * | |
| **linear probing** | $N$ | $N$ | $N$ | 3-5 * | 3-5 * | 3-5 * | |

* under uniform hashing assumption

# Parte IV

# Contexto

# História de guerra: ataques de complexidade

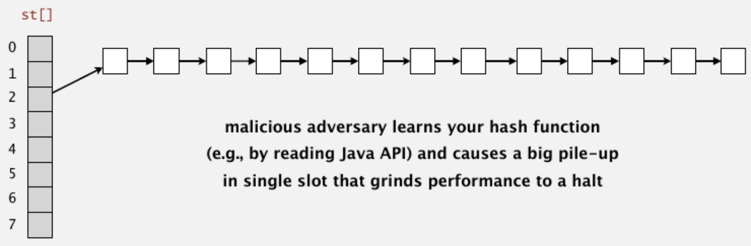
Segurança em computação: ataques de complexidade.

- Tabelas *hash* precisam ser resistentes à colisão, i.e., garantir a suposição de *hashing* uniforme.
- Caso contrário: ataques de DoS (*denial-of-service*).

# História de guerra: ataques de complexidade

Segurança em computação: ataques de complexidade.

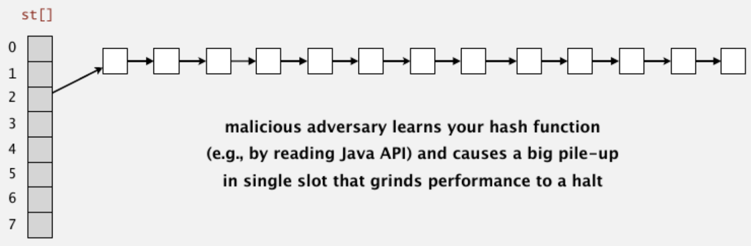
- Tabelas *hash* precisam ser resistentes à colisão, i.e., garantir a suposição de *hashing* uniforme.
- Caso contrário: ataques de DoS (*denial-of-service*).



# História de guerra: ataques de complexidade

Segurança em computação: ataques de complexidade.

- Tabelas *hash* precisam ser resistentes à colisão, i.e., garantir a suposição de *hashing* uniforme.
- Caso contrário: ataques de DoS (*denial-of-service*).




*Exploits* no mundo real [Crosby-Wallach, 2003]:

- Tomcat server: envia pacotes projetados para o servidor, faz DoS usando menos banda que uma linha discada.
- Linux 2.4.20 kernel: salvar arquivos com nomes especiais "trava" a fila de I/O para o disco.

# Ataque de complexidade em Java

Objetivo maligno: encontrar uma família de *strings* com o mesmo código *hash*.

Solução: a API para *strings* do Java usa o método de Horner com base 31 para calcular o *hash*.

| key | hashCode() |
|---|---|
| "Aa" | 2112 |
| "BB" | 2112 |

| key | hashCode() |
|---|---|
| "AaAaAaAa" | -540425984 |
| "AaAaAaBB" | -540425984 |
| "AaAaBBAa" | -540425984 |
| "AaAaBBBB" | -540425984 |
| "AaBBAaAa" | -540425984 |
| "AaBBAaBB" | -540425984 |
| "AaBBBBAa" | -540425984 |
| "AaBBBBBB" | -540425984 |

| key | hashCode() |
|---|---|
| "BBAaAaAa" | -540425984 |
| "BBAaAaBB" | -540425984 |
| "BBAaBBAa" | -540425984 |
| "BBAaBBBB" | -540425984 |
| "BBBBAaAa" | -540425984 |
| "BBBBAaBB" | -540425984 |
| "BBBBBBAa" | -540425984 |
| "BBBBBBBB" | -540425984 |

$2^N$ strings of length 2N that hash to same value!

# Defesa: funções *hash one-way*

Função *hash one-way*: "difícil" de achar uma chave que vai ser mapeada para um *hash* desejado.

- Exemplos: MD4, MD5, SHA-0, SHA-1, SHA-2, ...
- Os quatro primeiros métodos acima possuem vulnerabilidades.

# Defesa: funções *hash one-way*

Função *hash one-way*: "difícil" de achar uma chave que vai ser mapeada para um *hash* desejado.

- Exemplos: MD4, MD5, SHA-0, SHA-1, SHA-2, ...
- Os quatro primeiros métodos acima possuem vulnerabilidades.

- Aplicações: assinatura digital, armazenamento de senhas.
- Dificuldade: Muito custoso computacionalmente para ser usado em implementações genéricas de tabelas de símbolos.

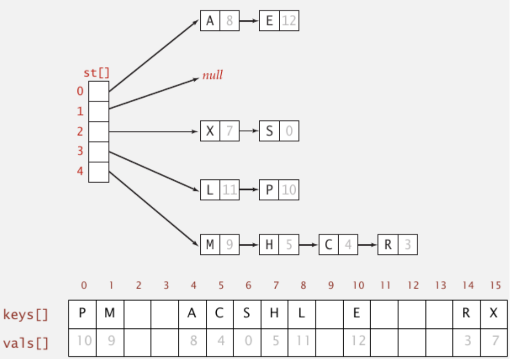
# *Separate chaining vs. linear probing*

*Separate chaining*:

- Desempenho degrada graciosamente.
- *Clustering* menos sensível a uma função *hash* ruim.

*Linear probing*:

- Menos espaço desperdiçado.
- Melhor uso de *cache*.



# *Hashing*: variações sobre o tema

Muitas variações de tabelas *hash* foram propostas.

*Two-probe hashing* [variante de *separate chaining*]:

- *Hash* para duas posições, insere na lista menor.
- Reduz tamanho esperado da maior lista para $\log \log N$.

# *Hashing*: variações sobre o tema

Muitas variações de tabelas *hash* foram propostas.

*Two-probe hashing* [variante de *separate chaining*]:

- *Hash* para duas posições, insere na lista menor.
- Reduz tamanho esperado da maior lista para $\log \log N$.

*Double hashing* [variante de *linear probing*]:

- Faz *linear probing* mas salta um distância variável maior que 1 (pode usar uma segunda função de *hash*).
- Praticamente elimina *clustering*.
- Mais difícil de implementar remoção.

# *Hashing*: variações sobre o tema

Muitas variações de tabelas *hash* foram propostas.

*Two-probe hashing* [variante de *separate chaining*]:

- *Hash* para duas posições, insere na lista menor.
- Reduz tamanho esperado da maior lista para $\log \log N$.

*Double hashing* [variante de *linear probing*]:

- Faz *linear probing* mas salta um distância variável maior que 1 (pode usar uma segunda função de *hash*).
- Praticamente elimina *clustering*.
- Mais difícil de implementar remoção.

*Cuckoo hashing* [variante de *linear probing*]:

- *Hash* para duas posições, insere em uma das duas. Se estiver ocupada, joga a chave desalojada para a outra posição, repete até alojar todas as chaves.
- Pior caso para busca é constante.

# Tabelas *hash* vs. árvores de busca balanceadas

Tabela *hash*:

- Mais fácil de programar.
- Não admite operações ordenadas de forma simples.
- Mais rápido para chaves simples: algumas ops aritméticas vs. $\log N$ comparações.
- Melhor suporte em algumas linguagens.
  Ex.: em Java todo objeto tem um *hash*.

# Tabelas *hash* vs. árvores de busca balanceadas

Tabela *hash*:

- Mais fácil de programar.
- Não admite operações ordenadas de forma simples.
- Mais rápido para chaves simples: algumas ops aritméticas vs. $\log N$ comparações.
- Melhor suporte em algumas linguagens.
  Ex.: em Java todo objeto tem um *hash*.

Árvores de busca balanceadas:

- Garantia de desempenho mais forte.
- Suporte a operações ordenadas sobre as chaves.
- Mais fácil de implementar uma função de comparação de chaves do que uma função *hash* decente.