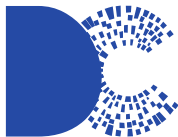




UNIVERSIDADE  
FEDERAL DE  
SERGIPE



DEPARTAMENTO  
DE COMPUTAÇÃO

# Busca sequencial e *hash*

## Estruturas de Dados

Bruno Prado

Departamento de Computação / UFS

# Introdução

- ▶ Estruturas de dados lineares
  - ▶ Conjunto de elementos em sequência
    - ▶ Armazenamento
    - ▶ Busca
    - ▶ Organização
    - ▶ ...

# Introdução

- ▶ Estruturas de dados lineares
  - ▶ Conjunto de elementos em sequência
    - ▶ Armazenamento
    - ▶ Busca
    - ▶ Organização
    - ▶ ...
  - ▶ Regras de operação
    - ▶ Lista: encadeamento simples, circular e duplo
    - ▶ Fila: *First-In First-Out* (FIFO)
    - ▶ Pilha: *Last-In First-Out* (LIFO)

# Introdução

- ▶ Estruturas de dados lineares
  - ▶ Conjunto de elementos em sequência
    - ▶ Armazenamento
    - ▶ Busca
    - ▶ Organização
    - ▶ ...
  - ▶ Regras de operação
    - ▶ Lista: encadeamento simples, circular e duplo
    - ▶ Fila: *First-In First-Out* (FIFO)
    - ▶ Pilha: *Last-In First-Out* (LIFO)
  - ▶ Complexidade
    - ▶ Espaço:  $\Theta(n)$
    - ▶ Tempo:  $\Omega(1)$  e  $O(n)$

# Introdução

- ▶ O que é a busca sequencial?
  - ▶ Consiste em acessar sequencialmente os dados da estrutura a partir de um elemento inicial, geralmente o primeiro da sequência

# Introdução

- ▶ O que é a busca sequencial?
  - ▶ Consiste em acessar sequencialmente os dados da estrutura a partir de um elemento inicial, geralmente o primeiro da sequência
  - ▶ A busca é finalizada com o elemento procurado é encontrado ou não existem mais elementos para serem comparados na estrutura

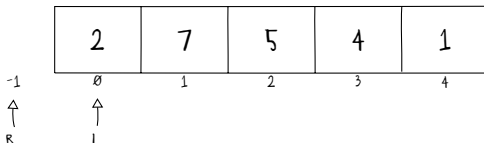
# Introdução

- ▶ O que é a busca sequencial?
  - ▶ Consiste em acessar sequencialmente os dados da estrutura a partir de um elemento inicial, geralmente o primeiro da sequência
  - ▶ A busca é finalizada com o elemento procurado é encontrado ou não existem mais elementos para serem comparados na estrutura

É uma estratégia de **força bruta**

# Busca sequencial

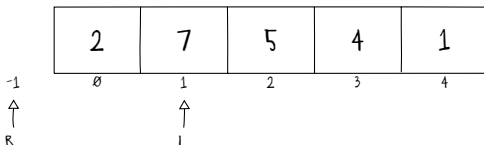
- ▶ Busca em vetores sem ordenação
  - ▶ Parâmetro de busca: 1
  - ▶ O índice de resultado  $r$  possui o valor  $-1$  (não encontrado) e o de busca  $i$  recebe o valor 0 (inicial)
  - ▶ É feita a comparação do elemento da posição  $i$  com o valor do parâmetro de busca





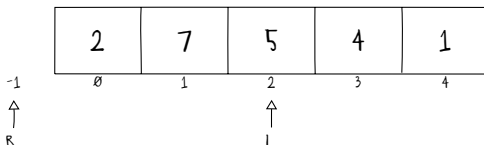
# Busca sequencial

- ▶ Busca em vetores sem ordenação
  - ▶ Parâmetro de busca: 1
  - ▶ Como o elemento procurado não foi encontrado na posição, o índice de busca  $i$  é incrementado e o índice de resultado  $r$  não é modificado



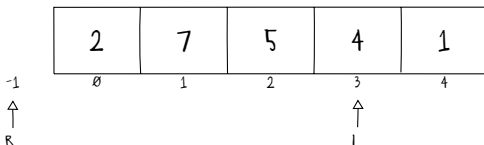
# Busca sequencial

- ▶ Busca em vetores sem ordenação
  - ▶ Parâmetro de busca: 1
  - ▶ Como o elemento procurado não foi encontrado na posição, o índice de busca  $i$  é incrementado e o índice de resultado  $r$  não é modificado



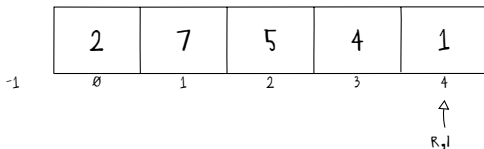
# Busca sequencial

- ▶ Busca em vetores sem ordenação
  - ▶ Parâmetro de busca: 1
  - ▶ Como o elemento procurado não foi encontrado na posição, o índice de busca  $i$  é incrementado e o índice de resultado  $r$  não é modificado



# Busca sequencial

- ▶ Busca em vetores sem ordenação
  - ▶ Parâmetro de busca: 1
  - ▶ O elemento procurado é encontrado na posição 4 e o índice de resultado  $r$  recebe o valor do  $i$



# Busca sequencial

## ► Implementação em C

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Função de busca sequencial em vetor
4 int32_t bseqv(uint32_t* V, uint32_t n, uint32_t x) {
5     // Índice de resultado
6     int32_t r = -1;
7     // Iterações de 0 -> n - 1
8     for(int32_t i = 0; i < n && r == -1; i++)
9         // Comparação de valor
10        if(V[i] == x)
11            // Atualização de índice
12            r = i;
13    // Retornando resultado
14    return r;
15 }
```

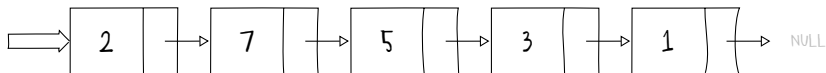
# Busca sequencial

- ▶ Análise de complexidade
  - ▶ Espaço:  $\Theta(n)$
  - ▶ Tempo:  $\Omega(1) \leq bseqv(n) \leq O(n)$

# Busca sequencial

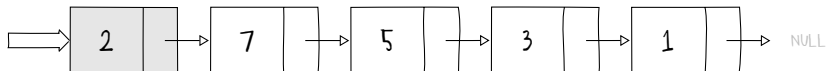
- ▶ Busca em listas

- ▶ Parâmetro de busca: 13
- ▶ A busca tem início acessando a cabeça da lista que contém a referência para o primeiro elemento



# Busca sequencial

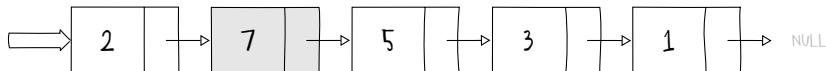
- ▶ Busca em listas
  - ▶ Parâmetro de busca: 13
  - ▶ É feito o acesso ao próximo elemento da lista para comparação com o valor do parâmetro de busca até que o elemento seja encontrado ou que não existam mais elementos na lista





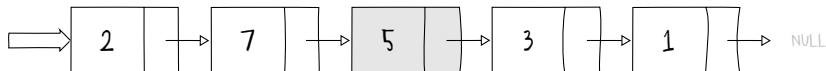
# Busca sequencial

- ▶ Busca em listas
  - ▶ Parâmetro de busca: 13
  - ▶ É feito o acesso ao próximo elemento da lista para comparação com o valor do parâmetro de busca até que o elemento seja encontrado ou que não existam mais elementos na lista



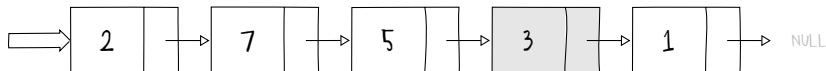
# Busca sequencial

- ▶ Busca em listas
  - ▶ Parâmetro de busca: 13
  - ▶ É feito o acesso ao próximo elemento da lista para comparação com o valor do parâmetro de busca até que o elemento seja encontrado ou que não existam mais elementos na lista



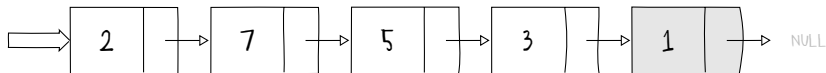
# Busca sequencial

- ▶ Busca em listas
  - ▶ Parâmetro de busca: 13
  - ▶ É feito o acesso ao próximo elemento da lista para comparação com o valor do parâmetro de busca até que o elemento seja encontrado ou que não existam mais elementos na lista



# Busca sequencial

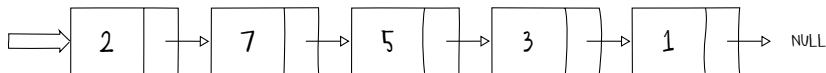
- ▶ Busca em listas
  - ▶ Parâmetro de busca: 13
  - ▶ É feito o acesso ao próximo elemento da lista para comparação com o valor do parâmetro de busca até que o elemento seja encontrado ou que não existam mais elementos na lista



# Busca sequencial

- ▶ Busca em listas

- ▶ Parâmetro de busca: 13
- ▶ O elemento procurado não é encontrado, sendo retornada uma referência nula pela função de busca



# Busca sequencial

## ► Implementação em C

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Função de busca sequencial em lista
4 elemento* bseq1(lista L, uint32_t x) {
5     // Ajustando ponteiros
6     elemento* i = L.L, * r = NULL;
7     // Iterações de cabeça -> cauda
8     while(i != NULL && r == NULL) {
9         // Comparação de valor
10        if(i->E == x)
11            // Atualização de referência
12            r = i;
13        // Próximo elemento da lista
14        i = i->P;
15    }
16    // Retornando resultado
17    return r;
18 }
```

# Busca sequencial

- ▶ Análise de complexidade
  - ▶ Espaço:  $\Theta(n)$
  - ▶ Tempo:  $\Omega(1) \leq bseqI(n) \leq O(n)$

# Hash

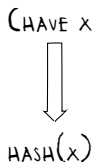
- ▶ O que é uma estratégia de busca com *hash*?
  - ▶ É o cálculo em tempo constante do índice em que um determinado elemento foi armazenado
  - ▶ O valor do parâmetro de busca é aplicado em uma função *hash* que mapeia o elemento procurado no vetor, permitindo seu acesso direto sem busca

CHAVE X



# Hash

- ▶ O que é uma estratégia de busca com *hash*?
  - ▶ É o cálculo em tempo constante do índice em que um determinado elemento foi armazenado
  - ▶ O valor do parâmetro de busca é aplicado em uma função *hash* que mapeia o elemento procurado no vetor, permitindo seu acesso direto sem busca



# Hash

- ▶ O que é uma estratégia de busca com *hash*?
  - ▶ É o cálculo em tempo constante do índice em que um determinado elemento foi armazenado
  - ▶ O valor do parâmetro de busca é aplicado em uma função *hash* que mapeia o elemento procurado no vetor, permitindo seu acesso direto sem busca

CHAVE  $x$



$\text{HASH}(x)$



ÍNDICE 7

# Hash

- ▶ O que é uma estratégia de busca com *hash*?
  - ▶ Não é prática a utilização de uma função injetora
    - ▶ Considere cadeias de caracteres utilizando somente letras minúsculas, com até 100 caracteres
    - ▶ Calculando as possíveis combinações de texto são possíveis até  $26^{100} \approx 3.14 \times 10^{141}$  mapeamentos distintos

# Hash

- ▶ O que é uma estratégia de busca com *hash*?
  - ▶ Não é prática a utilização de uma função injetora
    - ▶ Considere cadeias de caracteres utilizando somente letras minúsculas, com até 100 caracteres
    - ▶ Calculando as possíveis combinações de texto são possíveis até  $26^{100} \approx 3.14 \times 10^{141}$  mapeamentos distintos
  - ▶ A solução mais adequada para uma função *hash* é o mapeamento das entradas em um espaço de valores com tamanho fixo e reduzido
    - ▶ Determinação do tamanho do vetor
    - ▶ Tipo de tratamento de colisões

# Hash

- ▶ O que é uma função *hash*?
  - ▶ É o mapeamento de um conjunto de dados em índices de um vetor de tamanho limitado
  - ▶ A aplicação desta função permite o armazenamento e acesso associativo dos dados em tempo constante



# Hash

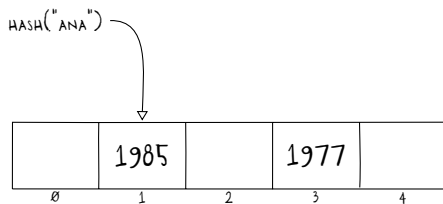
- ▶ O que é uma função *hash*?
  - ▶ É o mapeamento de um conjunto de dados em índices de um vetor de tamanho limitado
  - ▶ A aplicação desta função permite o armazenamento e acesso associativo dos dados em tempo constante

$\text{HASH}(\text{"ANA"})$

	1985		1977	
0	1	2	3	4

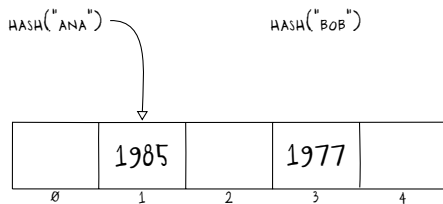
# Hash

- ▶ O que é uma função *hash*?
  - ▶ É o mapeamento de um conjunto de dados em índices de um vetor de tamanho limitado
  - ▶ A aplicação desta função permite o armazenamento e acesso associativo dos dados em tempo constante



# Hash

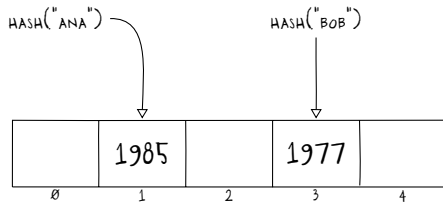
- ▶ O que é uma função *hash*?
  - ▶ É o mapeamento de um conjunto de dados em índices de um vetor de tamanho limitado
  - ▶ A aplicação desta função permite o armazenamento e acesso associativo dos dados em tempo constante





# Hash

- ▶ O que é uma função *hash*?
  - ▶ É o mapeamento de um conjunto de dados em índices de um vetor de tamanho limitado
  - ▶ A aplicação desta função permite o armazenamento e acesso associativo dos dados em tempo constante

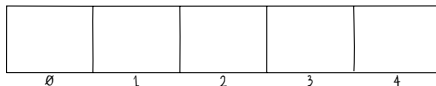


- ▶ Definições de uma função *hash*  $H$ 
  - ▶ Possuir idealmente um comportamento injetivo, nunca mapeando duas entradas distintas em uma mesmo índice do vetor
  - ▶ Utilizar uma função  $F$  com distribuição uniforme dos dados, reduzindo a chance de colisões de posições
  - ▶ Calcular valores de mapeamento limitados ao tamanho  $T$  do vetor

$$H(x) = F(x) \bmod T$$

# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É feita a alocação de um vetor com 5 posições



$$H(n) = 33 \times x \bmod 5$$

# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É calculado o índice para o elemento de valor 1



$$H(1) = 33 \times 1 \bmod 5 = 3$$

# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É calculado o índice para o elemento de valor 2



$$H(2) = 33 \times 2 \bmod 5 = 1$$

# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É calculado o índice para o elemento de valor 4



$$H(4) = 33 \times 4 \bmod 5 = 2$$

# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É calculado o índice para o elemento de valor 8

	2	4	1	
0	1	2	3	4

$$H(8) = 33 \times 8 \bmod 5 = 4$$

# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É calculado o índice para o elemento de valor 16

	2	4	1	8
0	1	2	3	4

$$H(16) = 33 \times 16 \bmod 5 = 3$$



# Hash

- ▶ Mapeando os elementos no vetor
  - ▶ Função *hash*  $H(x) = 33x \bmod 5$
  - ▶ É calculado o índice para o elemento de valor 16

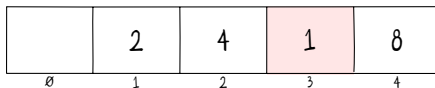
COLISÃO DE MAPEAMENTO!

	2	4	1	8
0	1	2	3	4

$$H(16) = 33 \times 16 \bmod 5 = 3$$

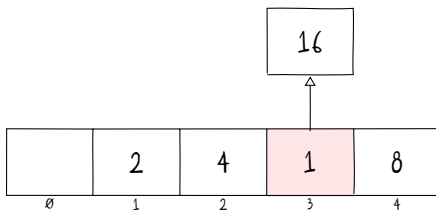
# Hash

- ▶ Endereçamento fechado
  - ▶ O tratamento de colisão é feito através da utilização de uma estrutura de dados auxiliar que permita o ajuste incremental de capacidade
  - ▶ Se uma estrutura de lista for utilizada, cada posição do vetor é a cabeça de uma lista, sendo inseridos novos elementos em caso de colisão



# Hash

- ▶ Endereçamento fechado
  - ▶ O tratamento de colisão é feito através da utilização de uma estrutura de dados auxiliar que permita o ajuste incremental de capacidade
  - ▶ Se uma estrutura de lista for utilizada, cada posição do vetor é a cabeça de uma lista, sendo inseridos novos elementos em caso de colisão



# Hash

- ▶ Endereçamento aberto
  - ▶ Técnica de *linear probing*
  - ▶ Função *hash* auxiliar  $LP(x, i) = H(x) + i \bmod T$ , com  $i = 0, 1, \dots, T-1$
  - ▶ É feito um novo cálculo de mapeamento utilizando a função *hash* auxiliar, evitando que espaço adicional seja alocado para o armazenamento dos elementos

	2	4	1	8
0	1	2	3	4

$$LP(16, 0) = (H(16) + 0) \bmod 5 = 3$$

# Hash

- ▶ Endereçamento aberto
  - ▶ Técnica de *linear probing*
  - ▶ Função *hash* auxiliar  $LP(x, i) = H(x) + i \bmod T$ , com  $i = 0, 1, \dots, T-1$
  - ▶ É feito um novo cálculo de mapeamento utilizando a função *hash* auxiliar, evitando que espaço adicional seja alocado para o armazenamento dos elementos

	2	4	1	8
0	1	2	3	4

$$LP(16, 1) = (H(16) + 1) \bmod 5 = 4$$

# Hash

- ▶ Endereçamento aberto
  - ▶ Técnica de *linear probing*
  - ▶ Função *hash* auxiliar  $LP(x, i) = H(x) + i \bmod T$ , com  $i = 0, 1, \dots, T-1$
  - ▶ É feito um novo cálculo de mapeamento utilizando a função *hash* auxiliar, evitando que espaço adicional seja alocado para o armazenamento dos elementos

16	2	4	1	8
0	1	2	3	4

$$LP(16, 2) = (H(16) + 2) \bmod 5 = 0$$

# Hash

- ▶ Endereçamento aberto
  - ▶ Técnica de *double hashing*
  - ▶ Função *hash* duplo  $DH(x, i) = H_1(x) + iH_2(x) \bmod T$
  - ▶ É aplicada uma função secundária de *hash* para deslocar o resultado para outra posição do vetor

	2	4	1	8
0	1	2	3	4

$$DH(16, 0) = (33 \times 16 + 0 \times 7 \times 16) \bmod 5 = 3$$

# Hash

- ▶ Endereçamento aberto
  - ▶ Técnica de *double hashing*
  - ▶ Função *hash* duplo  $DH(x, i) = H_1(x) + iH_2(x) \bmod T$
  - ▶ É aplicada uma função secundária de *hash* para deslocar o mapeamento para outro índice do vetor

16	2	4	1	8
0	1	2	3	4

$$DH(16, 1) = (33 \times 16 + 1 \times 7 \times 16) \bmod 5 = 0$$



- ▶ Análise de complexidade
  - ▶ Espaço:  $\Theta(n)$
  - ▶ Tempo:  $\Omega(1)$  e  $O(n)$
- ▶ Considerações
  - ▶ O tempo é constante desde que os dados estejam uniformemente distribuídos e a capacidade do vetor não seja muito utilizado, permitindo que o número de colisões seja reduzido
  - ▶ É recomendável a utilização de números primos para definir o tamanho do vetor, reduzindo assim as chances de colisões nos mapeamentos

# Hash

- ▶ Aplicações
  - ▶ Checagem de integridade de dados
  - ▶ Criptografia
  - ▶ Memória cache
  - ▶ Tabela de símbolos de compilador
  - ▶ ...

# Exercício

- ▶ A empresa de tecnologia Poxim Tech está criando um engenho de busca experimental para retornar ocorrências de texto em uma interface web
  - ▶ De acordo com o valor de chave gerado pelo texto é feito o mapeamento da requisição de busca para um dos servidores dedicados, utilizando um cálculo de *checksum* com 8 bits para cada um dos caracteres
  - ▶ Para atender as solicitações em tempo real, cada um dos servidores só é capaz de atender um número máximo de requisições ao mesmo tempo, sendo feita uma realocação do servidor por *double hashing* definida por
$$H_1(x) = 7919 \times checksum(x) \bmod T$$
$$H_2(x) = 104729 \times checksum(x) + 123 \bmod T$$
  - ▶ Todos os padrões pesquisados são compostos somente por letras e números com até 100 caracteres

# Exercício

- ▶ Função de *checksum* com 8 bits
  - ▶ Realiza a operação de ou-exclusivo ou *xor* ( $\oplus$ ) com os valores numéricos ASCII dos caracteres

$$\begin{aligned} checksum("ufs") &= 'u' \oplus 'f' \oplus 's' \\ &= 117 \oplus 102 \oplus 115 \\ &= 96 \end{aligned}$$

Probabilidade de duas strings diferentes gerarem o mesmo valor numérico é de  $\frac{1}{2^8} \approx 0,4\%$

# Exercício

- ▶ Formato do arquivo de entrada
  - ▶  $[ \# \text{Servidores} ] [ \text{Capacidade máxima} ]$
  - ▶  $[ \# n ]$
  - ▶  $[ \# m_1 ] [ P_1 ] [ P_2 ] \dots [ P_{m_1} ]$
  - ▶  $\dots$
  - ▶  $[ \# m_n ] [ P_1 ] [ P_2 ] \dots [ P_{m_n} ]$

```
1 3_2
2 5
3 1_ufs
4 3_a_b_c
5 2_cd_ef
6 2_e_d
7 1_hash
```

# Exercício

- ▶ Formato do arquivo de saída
  - ▶ É exibido o servidor alocado para realização da busca e os padrões de texto que estão sendo processados, além da realocação das requisições quando um servidor já atingiu o limite de operações

```
1 S0:ufs
2 S0:ufs,a_b_c
3 S2:cd_ef
4 S2:cd_ef,e_d
5 S0->S1
6 S1:hash
```