

# Estruturas de Dados I

## Filas

Igor Machado Coelho

18/09/2020

- 1 Filas
- 2 Tipo Abstrato: Fila
- 3 Filas Sequenciais
- 4 Filas Encadeadas
- 5 Filas na Biblioteca Padrão
- 6 Análise de Complexidade
- 7 Agradecimentos

# Section 1

## Filas

# Pré-Requisitos

São requisitos para essa aula:

- Introdução/Fundamentos de Programação (em alguma linguagem de programação)
- Interesse em aprender C/C++
- Noções de tipos de dados
- Noções de listas e encadeamento

## Section 2

### Tipo Abstrato: Fila

# Fila

A Fila (do inglês *Queue*) é um Tipo Abstrato de Dado (TAD) que pode ser compreendida como vemos no cotidiano.

Na *fila do banco*, por exemplo:

- Só se consegue “entrar” (enfileirar) no *fundo* (ou *fim*) da fila
- Será “atendido” (desenfileirar) quando está na *frente* na fila



Figure 1: Fila - CC BY 3.0 - thenounproject.com

# Filas na computação

Filas são estruturas fundamentais na própria computação.

Por exemplo, quando se envia pacotes de dados a roteadores, tipicamente é respeitada a ordem de chegada das mensagens.

Também são úteis na implementações de mecanismos de busca, como busca em largura para grafos (aulas futuras).

# Operações de uma Fila

Uma Fila é uma estrutura de dados linear (assim como estruturas de lista), consistindo de 3 operações básicas:

- frente (*front*)
- enfileira (*enqueue* ou *push*)
- desenfileira (*dequeue* ou *pop*)

Seu comportamento é descrito como FIFO (first-in first-out), ou seja, o *primeiro* elemento a entrar na fila será o *primeiro* a sair.



# Implementações

De forma geral, uma fila pode ser implementada utilizando uma lista linear (assim como uma pilha). Porém, tem acesso de direção restrita em ambas extremidades dessa lista: *de um lado entra, do outro lado sai* (um tipo restrito de *deque*).

→ | 3 | 2 | 1 | →

Para o TAD Fila, estudaremos duas formas distintas de implementação: Sequencial e Encadeada.

# Definição do *Conceito* Fila em C++

O *conceito* de fila somente requer suas três operações básicas. Como consideramos uma *fila genérica* (fila de inteiro, char, etc), definimos um *conceito genérico* chamado FilaTAD:

```
template<typename Agregado, typename Tipo>
concept bool
FilaTAD = requires(Agregado a, Tipo t)
{
    // requer operação 'frente'
    { a.frente() };
    // requer operação 'enfileira' sobre tipo 't'
    { a.enfileira(t) };
    // requer operação 'desenfileira'
    { a.desenfileira() };
};
```

## Section 3

### Filas Sequenciais

# Filas Sequenciais

As Filas Sequenciais utilizam um array para armazenar os dados. Assim, os dados sempre estarão em um *espaço contíguo* de memória.

# Implementação FilaSeq1

Consideraremos uma fila sequencial com, no máximo, MAXN elementos do tipo caractere.

```
constexpr int MAXN = 100'000; // capacidade máxima da fila
class FilaSeq1
{
public:
    char elementos [MAXN];      // elementos na fila
    int N;                      // num. de elementos na fila
    void cria () { ... }        // inicializa agregado
    void libera () { ... }      // finaliza agregado
    char frente () { ... }
    void enqueue (char dado){ ... };
    char dequeue () { ... };
};
// verifica se agregado FilaSeq1 satisfaz conceito FilaTAD
static_assert(FilaTAD<FilaSeq1, char>);
```

# Utilização da Fila

Antes de completar as funções pendentes, utilizaremos a FilaSeq1:

```
int main () {  
    FilaSeq1 p;  
    p.cria();  
    p.enqueue('A');  
    p.enqueue('B');  
    p.enqueue('C');  
    printf("%c\n", p.frente());  
    printf("%c\n", p.dequeue());  
    p.enqueue('D');  
    while(p.N > 0)  
        printf("%c\n", p.dequeue());  
    p.libera();  
    return 0;  
}
```

**Verifique as impressões em tela: A A B C D**

# Implementação FilaSeq1 - Parte 1/2

A operação `cria` inicializa a fila para uso, e a função `libera` desaloca os recursos dinâmicos.

```
class FilaSeq1 {  
    ...  
    void cria() {  
        this->N = 0;  
    }  
  
    void libera() {  
        // nenhum recurso dinâmico para desalocar  
    }  
    ...  
}
```

# Implementação FilaSeq1 - Parte 2/2

A operação enfileira em adiciona um novo elemento ao *fundo* da fila. A operação desenfileira remove e retorna o elemento na *frente* da fila.

```
// implementação 'FilaSeq1'
```

```
void frente(char dado) {  
    return this->elementos[0];    // primeiro sempre 'frente'  
}  
  
void enfileira(char dado) {  
    this->elementos[N] = dado;    this->N++;  
}  
  
char desenfileira() {  
    char r = this->elementos[0];    // primeiro sempre 'frente'  
    for (int i=0; i<N-1; i++)    // laço realmente necessário?  
        this->elementos[i] = this->elementos[i+1];  
    this->N--;    return r;  
}
```



# Análise da FilaSeq1

A FilaSeq1 funciona corretamente como TAD Fila, porém causa a realocação de todos elementos da fila *a cada remoção*.

Seria possível evitar tal efeito?

# Implementação FilaSeq2

```
constexpr int MAXN = 100'000; // capacidade máxima da fila
class FilaSeq2
{
public:
    char elementos [MAXN]; // elementos na fila
    int N; // num. de elementos na fila
    int inicio; // índice inicial da fila
    int fim; // índice final da fila
    void cria () { ... } // inicializa agregado
    void libera () { ... } // finaliza agregado
    char frente () { ... }
    void enqueue (char dado){ ... };
    char dequeue () { ... };
};
// verifica se agregado FilaSeq2 satisfaz conceito FilaTAD
static_assert(FilaTAD<FilaSeq2, char>);
```

# Implementação FilaSeq2: cria() e libera()

A operação `cria` inicializa a fila para uso, e a função `libera` desaloca os recursos dinâmicos.

```
class FilaSeq2 {  
    ...  
    void cria() {  
        this->N = 0;  
        this->inicio = 0;  
        this->fim = 0;  
    }  
  
    void libera() {  
        // nenhum recurso dinâmico para desalocar  
    }  
    ...  
}
```

# Implementação FilaSeq2: frente()

Utilizamos o índice `inicio` para localizar o começo da fila.

```
class FilaSeq2 {  
    ...  
  
    void frente(char dado) {  
        return this->elementos[inicio];  
    }  
  
    ...  
}
```

# Implementação FilaSeq2: enfileira e desenfileira

A operação enfileira em adiciona um novo elemento ao *fundo* da fila. A operação desenfileira remove e retorna o elemento na *frente* da fila.

```
// implementação 'FilaSeq2'
```

```
void enfileira(char dado) {  
    this->elementos[fim] = dado; // dado entra no fim  
    fim++;  
    N++;  
}
```

```
char desenfileira() {  
    char r = this->elementos[inicio];  
    inicio++;  
    N--;  
    return r;  
}
```

# Exemplo de uso (FilaSeq2)

Considere uma fila sequencial (MAXN=5): FilaSeq2 p; p.cria();

p.inicio:	0	p.elementos:					
p.fim:	0		0	1	2	3	4

Agora, enfileiramos A, B e C, e depois desenfileiramos uma vez.

p.inicio:	0	p.elementos:	A				
p.fim:	1		0	1	2	3	4

p.inicio:	0	p.elementos:	A   B				
p.fim:	2		0	1	2	3	4

p.inicio:	0	p.elementos:	A   B   C				
p.fim:	3		0	1	2	3	4

p.inicio:	1	p.elementos:		B   C			
p.fim:	3		0	1	2	3	4

*Qual a frente atual da fila? Quais limitações da fila?*

# Implementação FilaSeq3: enfileira e desenfileira

Consideramos uma *estratégia circular* na capacidade da fila:

```
// implementação 'FilaSeq3'
```

```
void enfileira(char dado) {  
    this->elementos[fim] = dado; // dado entra no fim  
    fim = (fim + 1) % MAXN;      // estratégia circular  
    N++;  
}  
  
char desenfileira() {  
    char r = this->elementos[inicio];  
    inicio = (inicio + 1) % MAXN; // estratégia circular  
    N--;  
    return r;  
}
```

# Exemplo de uso (FilaSeq3)

Considere uma fila sequencial (MAXN=5): `FilaSeq3 p; p.cria();`

<code>p.inicio:</code>	3	<code>p.elementos:</code>					
<code>p.fim:</code>	3		0	1	2	3	4

Agora, enfileiramos A, B e C, e depois desenfileiramos uma vez.

<code>p.inicio:</code>	3	<code>p.elementos:</code>				A	
<code>p.fim:</code>	4		0	1	2	3	4

<code>p.inicio:</code>	3	<code>p.elementos:</code>				A	B
<code>p.fim:</code>	0		0	1	2	3	4

<code>p.inicio:</code>	3	<code>p.elementos:</code>	C			A	B
<code>p.fim:</code>	1		0	1	2	3	4

<code>p.inicio:</code>	4	<code>p.elementos:</code>	C				B
<code>p.fim:</code>	1		0	1	2	3	4

*Qual a frente atual da fila?*



# Análise Preliminar: Fila Sequencial

A Fila Sequencial tem a vantagem de ser bastante simples de implementar, ocupando um espaço constante (na memória) para todas operações.

Porém, existe a limitação física de MAXN posições imposta pela alocação estática, não permitindo que a fila ultrapasse esse limite.

**Desafio:** implemente uma Fila Sequencial utilizando alocação dinâmica para o vetor elementos. Assim, quando não houver espaço para novos elementos, aloque mais espaço na memória (copiando elementos existentes para o novo vetor).

**Dica:** Experimente a estratégia de *dobrar a capacidade* da fila (quando necessário), e reduzir à metade a capacidade (quando necessário). Essa estratégia é bastante eficiente, mas requer alteração nos métodos *cria*, *libera*, *enfileira* e *desenfileira*.

## Section 4

### Filas Encadeadas

# Filas Encadeadas

A implementação do TAD Fila pode ser feito através de uma *estrutura encadeada* com alocação dinâmica de memória.

A vantagem é não precisar pre-determinar uma capacidade máxima da fila (o limite é a memória do computador!). A desvantagem é o consumo extra de espaço com ponteiros.

# Implementação

Consideraremos uma fila encadeada, utilizando um agregado NoFila1 para conectar cada elemento da fila:

```
class NoFila1
{
public:
    char dado;
    NoFila1* prox;
};

class FilaEnc1
{
public:
    NoFila1* inicio;    // frente da fila
    NoFila1* fim;       // fundo da fila
    int N;
    void cria () { ... }
    void libera () { ... }
    char frente () { ... }
    void enqueue (char dado){ ... };
    char dequeue () { ... };
};
// verifica agregado FilaEnc1
static_assert(FilaTAD<FilaEnc1, char>);
```

# Implementação: Cria e Libera

```
class FilaEnc1 {  
    ...  
void cria() {  
    this->N = 0;           // zero elementos na fila  
    this->inicio = 0;      // endereço zero de memória  
    this->fim = 0;         // endereço zero de memória  
}  
  
void libera() {  
    while(this->N > 0)  
        desenfileira(); // limpa a fila  
}  
    ...  
}
```

# Exemplo de uso

Variável local do tipo Fila Encadeada:

```
FilaEnc1 p;  
p.cria();
```

## Visualização da memória

p.N: 0	p.inicio: 0	p.fim: 0	<i>frente</i> $\leftarrow \epsilon$
0	4	...	100
			104
			108
			112
			116
			...
			8GiB

# Implementação: Enfileira

```
void enfileira(char v) {
    NoFila1* no = new NoFila1{.dado = v, .prox = 0};
    if(N == 0) { inicio = fim = no; }
    else      { fim->prox = no;   fim = no; }
    N++;
}
```

Na memória: p.enfileira('A'); p.enfileira('B');

**p.N: 0**      **p.inicio: 0**      **p.fim: 0**      *frente*  $\leftarrow \epsilon$

0	4	...	100	104	108	112	116	...	8GiB
---	---	-----	-----	-----	-----	-----	-----	-----	------

**p.N: 1**      **p.inicio: 112**      **p.fim: 112**      *frente*  $\leftarrow A$

0	4	...	100	104	108	112	116	...	8GiB
						A	0		

**p.N: 2**      **p.inicio: 112**      **p.fim: 100**      *frente*  $\leftarrow A \leftarrow B$

0	4	...	100	104	108	112	116	...	8GiB
			B	0		A	100		

# Implementação: Desenfileira

```
char desenfileira() {
    NoFila1* p = inicio;    // ponteiro da frente
    inicio = inicio->prox;  // avança fila
    char r = p->dado;        // conteúdo da frente
    delete p;               // apaga frente
    N--;
    return r;
}
```

Na memória: p.desenfileira();

<b>p.N: 2</b>	<b>p.inicio: 112</b>	<b>p.fim: 100</b>	<i>frente</i> ← A ← B						
		B	0		A	100			
0	4	...	100	104	108	112	116	...	8GiB
<b>p.N: 1</b>	<b>p.inicio: 100</b>	<b>p.fim: 100</b>	<i>frente</i> ← B						
		B	0						
0	4	...	100	104	108	112	116	...	8GiB



# Análise Preliminar: Fila Encadeada

A Fila Encadeada é flexível em relação ao espaço de memória, permitindo maior ou menor utilização.

Como desvantagem tende a ter acessos de memória ligeiramente mais lentos, devido ao espalhamento dos elementos por toda a memória do computador (perdendo as vantagens de acesso rápido na *memória cache*, por exemplo).

Também é considerada como desvantagem o gasto de espaço extra com ponteiros em cada elemento, o que não acontece na Fila Sequencial.

## Section 5

### Filas na Biblioteca Padrão

# Uso da `std::queue`

Em C/C++, é possível utilizar implementações *prontas* do TAD Fila. A vantagem é a grande eficiência computacional e amplo conjunto de testes, evitando erros de implementação.

Na STL, basta fazer `#include<queue>` e usar métodos `push`, `pop` e `front`.

```
#include<iostream>                // inclui printf
#include<queue>                    // inclui fila genérica

int main() {
    std::queue<char> p;            // fila de char
    p.push('A');
    p.push('B');
    printf("%c\n", p.front());    // imprime A
    p.pop();
    printf("%c\n", p.front());    // imprime B
    return 0;
```

# Definindo um TAD para `std::queue`

**Desafio:** escreva um *conceito* (utilizando o recurso C++ `concept bool`) para o `std::queue` da STL, considerando operações `push`, `pop` e `front`.

**Dica:** Utilize o *conceito* `FilaTAD` apresentado no curso, e faça os devidos ajustes. Verifique se `std::queue` passa no teste com `static_assert`.

*Você pode compilar o código proposto (começando pelo slide anterior em um arquivo chamado `main_fila.cpp`) através do comando:*

```
g++ -fconcepts main_fila.cpp -o appFila
```

# Fim implementações

Fim parte de implementações.

## Section 6

# Análise de Complexidade

# Fila: Revisão Geral

- Para que serve uma fila?
- Quais são os 3 métodos de uma fila?
- Qual é a complexidade de cada método em uma Fila Sequencial?
- Qual é a complexidade de cada método em uma Fila Encadeada?
- Quais as vantagens e desvantagens de cada implementação de fila?

# Bibliografia Recomendada

Além da bibliografia do curso, recomendamos para esse tópico:

- Szwarcfiter, J.L.; Markenzon, L. Estruturas de Dados e seus Algoritmos. Rio de Janeiro, LTC, 1994. Bibliografia Adicional:
- Cerqueira, R.; Celes, W.; Rangel, J.L. Introdução a estruturas de dados: com técnicas de programação em C. Editora, 2004.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein Algoritmos: Teoria e Prática. Ed. Campus, 2002.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Introduction to Algorithms, 3rd ed.. The MIT Press, 2009.
- Preiss, B.R. Estruturas de Dados e Algoritmos Ed. Campus, 2000;
- Knuth, D.E. The Art of Computer Programming - Vols I e III. 2nd Edition. Addison Wesley, 1973.
- Graham, R.L., Knuth, D.E., Patashnik, O. Matemática Concreta. Segunda Edição, Rio de Janeiro, LTC, 1995.
- Livro "The C++ Programming Language" de Bjarne Stroustrup
- Dicas e normas C++: <https://github.com/isocpp/CppCoreGuidelines>



## Section 7

### Agradecimentos

# Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Fabiano Oliveira (IME-UERJ), e o prof. Jayme Szwarcfiter cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de **Pesquisa Operacional**, que abriu caminho para verificação prática dessa tecnologia de slides.

# Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- gromit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

# Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

- github
- gitlab
- microsoft
- google
- ...

# Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

- <https://igormcoelho.github.io/ilectures-pandoc/>

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

**Licença:** CC-BY 4.0 2020

Igor Machado Coelho

# This Slide Is Intentionally Blank (for goomit-mpx)