# Pilhas, Filas e Deques

## Conceitos e implementação

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados

Universidade do Estado de Santa Catarina

# Material de apoio

**Leitura principal:**

- Capítulo 6 de Goodrich et al. (2014)[1] – Pilhas, filas e deques.

**Leitura complementar:**

- Capítulo 6 de Preiss (2001)[2] – Pilhas, filas e deques.

[1] Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

[2] Bruno R Preiss (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.

# Estruturas sequenciais

## Ou seja, TADs sequenciais

Vamos construir tipos abstratos de dados (TADs) sequenciais que usam as estruturas de dados fundamentais (arranjos e listas encadeadas) para a implementação de estruturas mais abstratas e com comportamentos específicos.

# Estruturas sequenciais

## Ou seja, TADs sequenciais

Vamos construir tipos abstratos de dados (TADs) sequenciais que usam as estruturas de dados fundamentais (arranjos e listas encadeadas) para a implementação de estruturas mais abstratas e com comportamentos específicos.

Em particular, implementaremos:

- **Pilhas** (*stacks*);
- **Filas** (*queues*);
- **Deques** (*double-ended queues*).

# Estruturas sequenciais

## Ou seja, TADs sequenciais

Vamos construir tipos abstratos de dados (TADs) sequenciais que usam as estruturas de dados fundamentais (arranjos e listas encadeadas) para a implementação de estruturas mais abstratas e com comportamentos específicos.

Em particular, implementaremos:

- **Pilhas** (*stacks*);
- **Filas** (*queues*);
- **Deques** (*double-ended queues*).

Essas estruturas se diferenciam pela **política de inserção e remoção dos dados** adotada. Ou seja, somente serão permitidas as operações de interesse.

# Pilhas

## Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

# Pilhas

## Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

A pilha simula uma "pilha de objetos", logo ela tem um **topo**.

# Pilhas

## Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

A pilha simula uma "pilha de objetos", logo ela tem um **topo**.

**Operações:**

- `push`: insere um elemento no topo da pilha.
- `pop`: remove (e retorna) o elemento do topo da pilha.
- `top`: retorna o elemento do topo da pilha (sem remover).

# Pilhas

## Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

A pilha simula uma "pilha de objetos", logo ela tem um **topo**.

**Operações:**

- `push`: insere um elemento no topo da pilha.
- `pop`: remove (e retorna) o elemento do topo da pilha.
- `top`: retorna o elemento do topo da pilha (sem remover).

**Aplicações:**

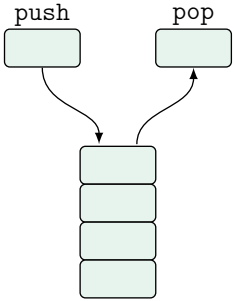
- Histórico de acesso de um navegador (voltar, avançar).
- Operações em um editor de texto (desfazer, refazer).

# Pilhas

## Exemplo de funcionamento

Seja uma pilha implementada usando uma lista (sequencial), onde o topo é o final da lista.

| Método | Retorno | Conteúdo da pilha |
|---|---|---|
| `push(5)` | – | (5) |
| `push(3)` | – | (5, 3) |
| `size()` | 2 | (5, 3) |
| `pop()` | 3 | (5) |
| `isEmpty()` | false | (5) |
| `pop()` | 5 | () |
| `isEmpty()` | true | () |
| `pop()` | null | () |
| `push(7)` | – | (7) |
| `push(9)` | – | (7, 9) |
| `top()` | 9 | (7, 9) |
| `push(4)` | – | (7, 9, 4) |
| `size()` | 3 | (7, 9, 4) |
| `pop()` | 4 | (7, 9) |
| `push(6)` | – | (7, 9, 6) |
| `push(8)` | – | (7, 9, 6, 8) |
| `pop()` | 8 | (7, 9, 6) |




(representação gráfica)

# Pilhas

## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

# Pilhas

## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
```

# Pilhas

## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
top():                      O(1)
  retorna data[t]
```

# Pilhas

## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t--]
```

# Pilhas

## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

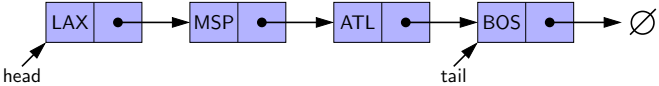
- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t--]
```

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



# Pilhas

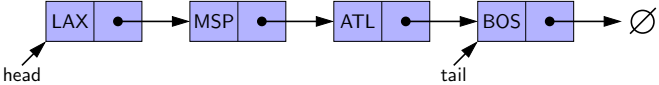
## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.




**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t--]
```

**Pilha usando encadeamento**

```
push(e):                    O(1)
  list.addFirst(e)
```

# Pilhas

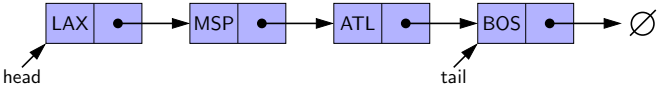
## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.




**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t--]
```

**Pilha usando encadeamento**

```
push(e):                    O(1)
  list.addFirst(e)
top():                      O(1)
  retorna list.first()
```

# Pilhas

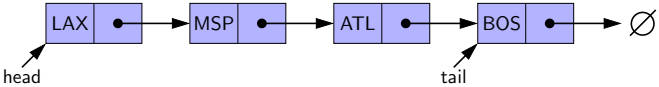
## Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.




**Pilha usando arranjo**

```
push(e):                    O(1)
  data[++t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t--]
```

**Pilha usando encadeamento**

```
push(e):                    O(1)
  list.addFirst(e)
top():                      O(1)
  retorna list.first()
pop():                      O(1)
  retorna list.removeFirst()
```

# Pilhas

## Análise de complexidade

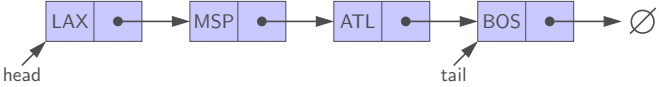
Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

| Operação | `ArrayStack` | `LinkedStack` |
|---|---|---|
| `size` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `isEmpty` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `top` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `push` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `pop` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

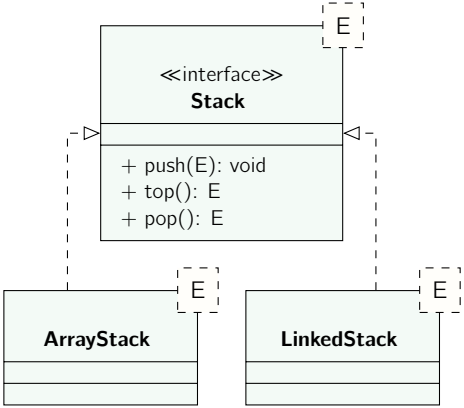
Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



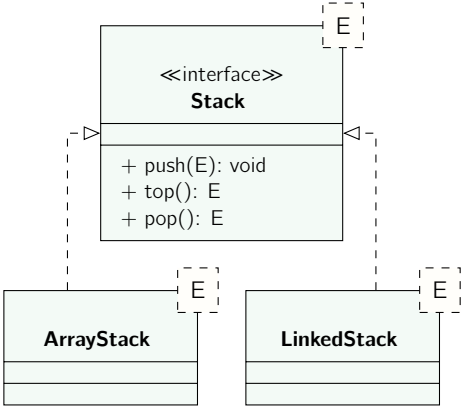
# Pilhas

## Implementação



# Pilhas

## Implementação




**Implementação e uso:**

```
data-structures
└─ stack
    ├─ {Stack, ArrayStack, LinkedStack}.java
    └─ TestStack.java
```

**Veja também:**

- `java.util.Stack`[1]
- Apêndice I

[1] Veja em https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Stack.html.

# Filas

## Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

# Filas

## Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

A fila simula uma "fila de pessoas", logo ela tem uma **frente** e um **final**.

# Filas

## Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

A fila simula uma "fila de pessoas", logo ela tem uma **frente** e um **final**.

**Operações:**

- `enqueue`: insere um elemento no final da fila.
- `dequeue`: remove (e retorna) o elemento do início da fila.

# Filas

## Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

A fila simula uma "fila de pessoas", logo ela tem uma **frente** e um **final**.

**Operações:**

- `enqueue`: insere um elemento no final da fila.
- `dequeue`: remove (e retorna) o elemento do início da fila.

**Aplicações:**

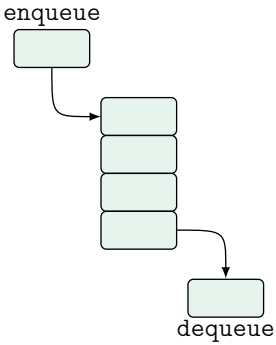
- Impressora de rede.
- Servidor Web respondendo requisições.

# Filas

## Exemplo de funcionamento

Seja uma fila implementada usando uma lista (sequencial), onde a frente é o início da lista.

| Método | Retorno | Conteúdo da fila |
|---|---|---|
| `enqueue(5)` | – | (5) |
| `enqueue(3)` | – | (5, 3) |
| `size()` | 2 | (5, 3) |
| `dequeue()` | 5 | (3) |
| `isEmpty()` | false | (3) |
| `dequeue()` | 3 | () |
| `isEmpty()` | true | () |
| `dequeue()` | null | () |
| `enqueue(7)` | – | (7) |
| `enqueue(9)` | – | (7, 9) |
| `first()` | 7 | (7, 9) |
| `enqueue(4)` | – | (7, 9, 4) |

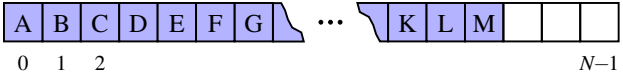



(representação gráfica)

# Filas

## Análise de complexidade

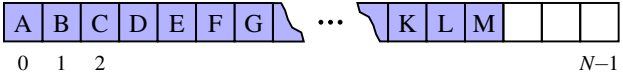
Em uma implementação usando arranjos, temos:



# Filas

## Análise de complexidade

Em uma implementação usando arranjos, temos:



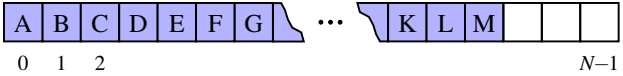

A frente da fila é no início do *array*. Como **desenfileirar** (i.e., remover do início)?

- Remover do índice 0 e fazer *shift* dos elementos. → $\mathcal{O}(n)$
- Usar uma variável (`f`) como referência para a frente da fila. → $\mathcal{O}(1)$

# Filas

## Análise de complexidade

Em uma implementação usando arranjos, temos:

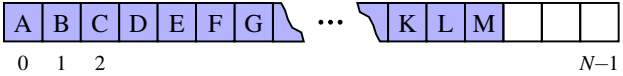



Remoção usando a variável `f`:

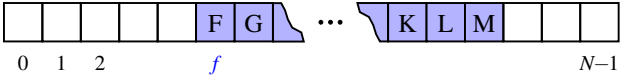
# Filas

## Análise de complexidade

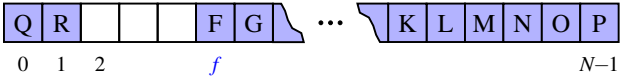
Em uma implementação usando arranjos, temos:




Remoção usando a variável `f`:




O novo desafio é controlar os espaços vagos no início do *array*:

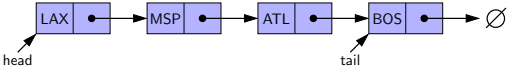


# Filas

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

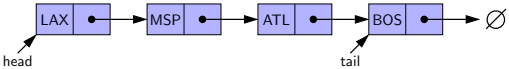


# Filas

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**




**Fila usando arranjo**

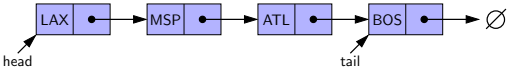
```
enqueue(e):                                  O(1)
  pos ← (f + size) % data.length
  data[pos] ← e
```

# Filas

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**




**Fila usando arranjo**

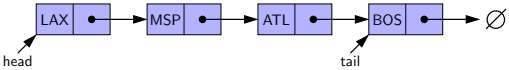
```
enqueue(e):                                  O(1)
  pos ← (f + size) % data.length
  data[pos] ← e

dequeue():                                   O(1)
  pos ← f
  f ← (f + 1) % data.length
  retorna data[pos]
```

# Filas

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**




**Fila usando arranjo**

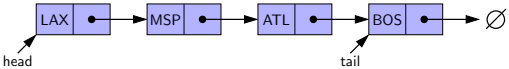
```
enqueue(e):                                  O(1)
  pos ← (f + size) % data.length
  data[pos] ← e

dequeue():                                   O(1)
  pos ← f
  f ← (f + 1) % data.length
  retorna data[pos]

first():                                     O(1)
  retorna data[f]
```

# Filas

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**




### Fila usando arranjo

```
enqueue(e):                                  O(1)
  pos ← (f + size) % data.length
  data[pos] ← e

dequeue():                                   O(1)
  pos ← f
  f ← (f + 1) % data.length
  retorna data[pos]

first():                                     O(1)
  retorna data[f]
```

### Fila usando encadeamento

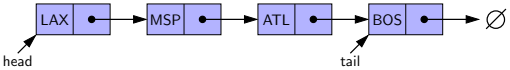
```
enqueue(e):                                  O(1)
  list.addLast(e)

dequeue():                                   O(1)
  retorna list.removeFirst()

first():                                     O(1)
  retorna list.first()
```

# Filas

## Análise de complexidade

Implementação usando **arranjos**

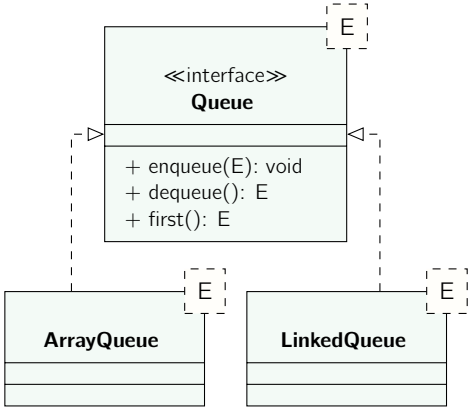
Implementação usando **encadeamento**




| Operação | `ArrayQueue` | `LinkedQueue` |
|---|---|---|
| `size` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `isEmpty` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `first` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `enqueue` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `dequeue` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

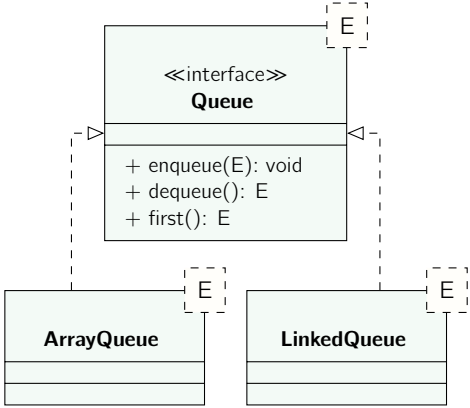
# Filas

## Implementação



# Filas

## Implementação




**Implementação e uso:**

```
data-structures
└─ queue
    ├─ {Queue, ArrayQueue, LinkedQueue}.java
    └─ TestQueue.java
```

**Veja também:**

- `java.util.Queue`[1] (interface)
  - E suas diferentes implementações.
- Apêndice II

[1] Veja em https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Queue.html.

# Deques

## Ideia geral

Um **deque** (*double-ended queue*) é uma fila com inserção e remoção em ambos os lados.

- i.e., é possível inserir e remover do início e do fim, mas não do meio.

# Deques

## Ideia geral

Um **deque** (*double-ended queue*) é uma fila com inserção e remoção em ambos os lados.

- i.e., é possível inserir e remover do início e do fim, mas não do meio.

**Operações:**

- `addFirst`: insere um elemento no início da estrutura.
- `addLast`: insere um elemento no final da estrutura.
- `removeFirst`: remove (e retorna) o elemento do início da estrutura.
- `removeLast`: remove (e retorna) o elemento do final da estrutura.

# Deques

## Ideia geral

Um **deque** (*double-ended queue*) é uma fila com inserção e remoção em ambos os lados.

- i.e., é possível inserir e remover do início e do fim, mas não do meio.

**Operações:**

- `addFirst`: insere um elemento no início da estrutura.
- `addLast`: insere um elemento no final da estrutura.
- `removeFirst`: remove (e retorna) o elemento do início da estrutura.
- `removeLast`: remove (e retorna) o elemento do final da estrutura.

**Aplicações:**

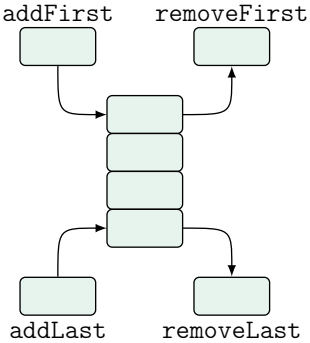
- Algoritmos de escalonamento de processos.
- Base para outras estruturas (e.g., fila dupla).

# Deques

## Exemplo de funcionamento

Seja um deque implementado usando uma lista (sequencial), onde a frente é o início da lista.

| Método | Retorno | Conteúdo do deque |
|---|---|---|
| `addLast(5)` | – | (5) |
| `addFirst(3)` | – | (3, 5) |
| `addFirst(7)` | – | (7, 3, 5) |
| `first()` | 7 | (7, 3, 5) |
| `removeLast()` | 5 | (7, 3) |
| `size()` | 2 | (7, 3) |
| `removeLast()` | 3 | (7) |
| `removeFirst()` | 7 | () |
| `addFirst(6)` | – | (6) |
| `last()` | 6 | (6) |
| `addFirst(8)` | – | (8, 6) |
| `isEmpty()` | false | (8, 6) |
| `last()` | 6 | (8, 6) |




(representação gráfica)

# Deques

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

# Deques

## Análise de complexidade

Implementação usando **arranjos**

**Deque usando arranjo**

```
addFirst(e):                                  O(1)
  f ← (f - 1) % data.length
  data[f] ← e
```

Implementação usando **encadeamento**

# Deques

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

### Deque usando arranjo

```
addFirst(e):                                  O(1)
  f ← (f - 1) % data.length
  data[f] ← e

addLast(e):                                   O(1)
  f ← (f + size) % data.length
  data[f] ← e
```

# Deques

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

**Deque usando arranjo**

```
removeFirst():                    O(1)
  pos ← f
  f ← (f + 1) % data.length
  retorna data[pos]
```

# Deques

## Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

### Deque usando arranjo

```
removeFirst():                                  O(1)
  pos ← f
  f ← (f + 1) % data.length
  retorna data[pos]

removeLast():                                   O(1)
  pos ← (f + size - 1) % data.length
  size ← size - 1
  retorna data[pos]
```

# Deques

## Análise de complexidade

### Implementação usando **arranjos**

**Deque usando arranjo**

```
removeFirst():                                  O(1)
  pos ← f
  f ← (f + 1) % data.length
  retorna data[pos]

removeLast():                                   O(1)
  pos ← (f + size - 1) % data.length
  size ← size - 1
  retorna data[pos]
```

### Implementação usando **encadeamento**

**Deque usando encadeamento**

```
addFirst(e):                                    O(1)
  list.addFirst(e)

addLast(e):                                     O(1)
  list.addLast(e)

removeFirst():                                  O(1)
  list.removeFirst()

removeLast():                                   O(1)
  list.removeLast()
```

# Deques

## Análise de complexidade

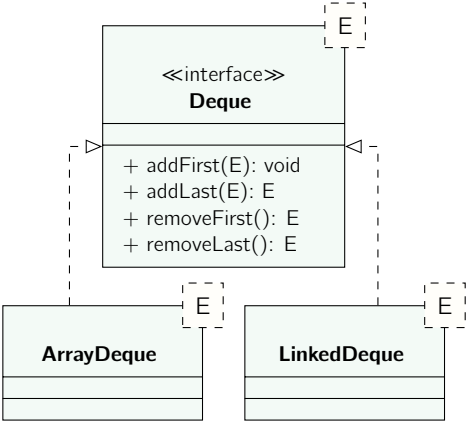
Implementação usando **arranjos**

Implementação usando **encadeamento**

| Operação | `ArrayDeque` | `LinkedDeque` |
|---|---|---|
| `size` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `isEmpty` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `first` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `last` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `addFirst` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `addLast` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `removeFirst` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `removeLast` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

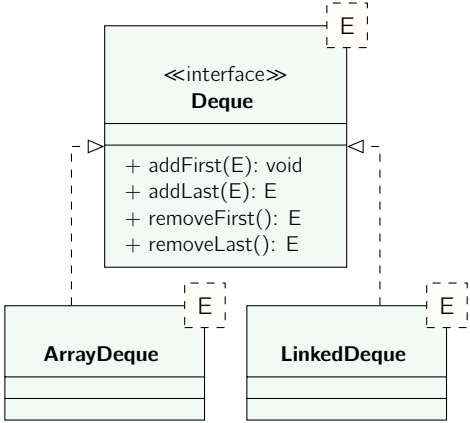
# Deques

## Implementação



# Deques

## Implementação




**Implementação e uso:**

```
data-structures
└─ deque
    ├─ {Deque, ArrayDeque, LinkedDeque}.java
    └─ TestDeque.java
```

**Veja também:**

- `java.util.Deque`[1] (interface)
  - E suas diferentes implementações.
- Apêndice III

[1] Veja em https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Deque.html.

# Apêndices

# Apêndice I

## Implementação de pilhas (interface)

Interface `Stack`:

```
1 public interface Stack<E> {
2   int size();
3   boolean isEmpty();
4   void push(E e);
5   E top();
6   E pop();
7 }
```

Usamos uma interface para definir nosso TAD pilha. A interface só define a API (métodos), a qual pode ser implementada de diversas formas (e usando diferentes estruturas).

# Apêndice I

## Implementação de pilhas (arranjos)

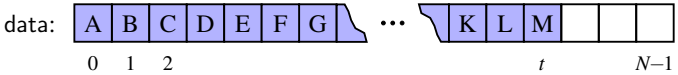
```java
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY = 1000;
  private E[] data;
  private int t = -1;

  public ArrayStack() { this(CAPACITY); }

  public ArrayStack(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }

  // ...
}
```

# Apêndice I

## Implementação de pilhas (arranjos)

```java
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY = 1000;
  private E[] data;
  private int t = -1;

  public ArrayStack() { this(CAPACITY); }

  public ArrayStack(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }

  // ...
}
```

- `ArrayStack` implementa a interface `Stack`.
- A pilha possui uma capacidade fixa, dada pela constante `CAPACITY` ou definida pelo usuário no construtor parametrizado.
- Os dados são armazenados no *array* `data`.
- O atributo `t` indica o topo da pilha, armazenando o índice do elemento do topo.
- O tamanho da pilha é dado por `t + 1` (conforme método `size`).
- A lista é vazia quando não há elementos, ou seja, o topo `t` é -1 (método `isEmpty`).



# Apêndice I

## Implementação de pilhas (arranjos)

Método `push`:

```
public void push(E e) {
  if (size() == data.length) throw new IllegalStateException("Stack is full");
  data[++t] = e;
}
```

# Apêndice I

## Implementação de pilhas (arranjos)

### Método push:

```
public void push(E e) {
  if (size() == data.length) throw new IllegalStateException("Stack is full");
  data[++t] = e;
}
```

### Método top:

```
public E top() {
  if (isEmpty()) return null;
  return data[t];
}
```

# Apêndice I

## Implementação de pilhas (arranjos)

### Método push:

```
public void push(E e) {
  if (size() == data.length) throw new IllegalStateException("Stack is full");
  data[++t] = e;
}
```

### Método top:

```
public E top() {
  if (isEmpty()) return null;
  return data[t];
}
```

### Método pop:

```
public E pop() {
  if (isEmpty()) return null;
  E answer = data[t];
  data[t] = null;
  t--;
  return answer;
}
```

# Apêndice I

## Implementação de pilhas (arranjos)

Método `toString`:

```java
public String toString() {
  StringBuilder sb = new StringBuilder("(");
  for (int j = t; j >= 0; j--) {
    sb.append(data[j]);
    if (j > 0) sb.append(", ");
  }
  sb.append(")");
  return sb.toString();
}
```

- O método apresenta os elementos iniciando pelo topo da pilha (`t`).

# Apêndice I

## Implementação de pilhas (encadeamento)

```java
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
  public String toString() { return list.toString(); }
}
```

# Apêndice I

## Implementação de pilhas (encadeamento)

```java
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
  public String toString() { return list.toString(); }
}
```

- Essa implementação usa uma lista simplesmente encadeada.
- Não há limite de capacidade.
- Como a remoção no início de uma lista simplesmente encadeada é mais eficiente, o topo da pilha é o início da lista.
- A implementação é simples, pois basta invocar os métodos da lista encadeada que executam as operações desejadas (`size`, `isEmpty`, `addFirst`, ...).

# Apêndice II

## Implementação de filas (interface)

Interface Queue:

```
1 public interface Queue<E> {
2   int size();
3   boolean isEmpty();
4   void enqueue(E e);
5   E first();
6   E dequeue();
7 }
```

Usamos uma interface para definir nosso TAD fila. A interface só define a API (métodos), a qual pode ser implementada de diversas formas (e usando diferentes estruturas).

# Apêndice II

## Implementação de filas (arranjos)

```java
public class ArrayQueue<E> implements Queue<E> {
  public static final int CAPACITY = 1000;
  private E[] data;
  private int f = 0;
  private int size = 0;

  public ArrayQueue() {this(CAPACITY);}

  public ArrayQueue(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return size; }
  public boolean isEmpty() { return (size == 0); }

  // ...
}
```

# Apêndice II

## Implementação de filas (arranjos)

```java
public class ArrayQueue<E> implements Queue<E> {
  public static final int CAPACITY = 1000;
  private E[] data;
  private int f = 0;
  private int size = 0;

  public ArrayQueue() {this(CAPACITY);}

  public ArrayQueue(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return size; }
  public boolean isEmpty() { return (size == 0); }

  // ...
}
```

- `ArrayQueue` implementa a interface `Queue`.
- A fila possui uma capacidade fixa, dada pela constante `CAPACITY` ou definida pelo usuário no construtor parametrizado.
- Os dados são armazenados no *array* `data`.
- O atributo `f` indica a frente (início) da fila, armazenando o índice do elemento da frente.
- O tamanho da fila é dado pelo atributo `size`.
- A inserção deve acontecer no final do *array*, enquanto a remoção deve acontecer no início (i.e., no índice `f`).

# Apêndice II

## Implementação de filas (arranjos)

Método enqueue:

```
public void enqueue(E e) {
  if (size == data.length)
    throw new IllegalStateException("Queue is full");
  int avail = f + size;
  if(avail >= data.length)
    avail -= data.length;
  data[avail] = e;
  size++;
}
```

# Apêndice II

## Implementação de filas (arranjos)

Método `enqueue`:

```
1  public void enqueue(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Queue is full");
4    int avail = f + size;
5    if(avail >= data.length)
6      avail -= data.length;
7    data[avail] = e;
8    size++;
9  }
```

- O próximo índice disponível no final da fila está em `f + size`.
- As linhas 5 e 6 implementam a estratégia circular de ocupar índices vagos (pela remoção) no início do *array*.

# Apêndice II

## Implementação de filas (arranjos)

Método enqueue:

```
1  public void enqueue(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Queue is full");
4    int avail = f + size;
5    if(avail >= data.length)
6      avail -= data.length;
7    data[avail] = e;
8    size++;
9  }
```

Método dequeue:

```
1  public E dequeue() {
2    if (isEmpty()) return null;
3    E answer = data[f];
4    data[f] = null;
5    f = f + 1;
6    if(f >= data.length)
7      f = 0;
8    size--;
9    return answer;
10 }
```

- O próximo índice disponível no final da fila está em `f + size`.
- As linhas 5 e 6 implementam a estratégia circular de ocupar índices vagos (pela remoção) no início do *array*.

# Apêndice II

## Implementação de filas (arranjos)

Método enqueue:

```
1  public void enqueue(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Queue is full");
4    int avail = f + size;
5    if(avail >= data.length)
6      avail -= data.length;
7    data[avail] = e;
8    size++;
9  }
```

- O próximo índice disponível no final da fila está em `f + size`.
- As linhas 5 e 6 implementam a estratégia circular de ocupar índices vagos (pela remoção) no início do *array*.

Método dequeue:

```
1  public E dequeue() {
2    if (isEmpty()) return null;
3    E answer = data[f];
4    data[f] = null;
5    f = f + 1;
6    if(f >= data.length)
7      f = 0;
8    size--;
9    return answer;
10 }
```

- O elemento a ser removido é o que está na frente `f` da fila.
- Após receber `null`, `f` é incrementado para apontar à nova frente da fila.
- As linhas 6 e 7 implementam a estratégia circular de atualização de `f`.

# Apêndice II

## Implementação de filas (arranjos)

Método `first`:

```
public E first() {
  if (isEmpty()) return null;
  return data[f];
}
```

# Apêndice II

## Implementação de filas (arranjos)

### Método `first`:

```java
public E first() {
  if (isEmpty()) return null;
  return data[f];
}
```

### Método `toString`:

```java
public String toString() {
  StringBuilder sb = new StringBuilder("(");
  int k = f;
  for (int j=0; j < size; j++) {
    if (j > 0)
      sb.append(", ");
    sb.append(data[k]);
    k = (k + 1) % data.length;
  }
  sb.append(")");
  return sb.toString();
}
```

# Apêndice II

## Implementação de filas (encadeamento)

```java
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
  public String toString() { return list.toString(); }
}
```

# Apêndice II

## Implementação de filas (encadeamento)

```java
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
  public String toString() { return list.toString(); }
}
```

- Essa implementação usa uma lista simplesmente encadeada.
- Não há limite de capacidade.
- Como a remoção no início de uma lista simplesmente encadeada é mais eficiente, a frente da fila é o início da lista.
- A implementação é simples, pois basta invocar os métodos da lista encadeada que executam as operações desejadas (`size`, `isEmpty`, `addLast`, ...).

# Apêndice III

## Implementação de deques (interface)

Interface Deque:

```
public interface Deque<E> {
  int size();
  boolean isEmpty();
  E first();
  E last();
  void addFirst(E e);
  void addLast(E e);
  E removeFirst();
  E removeLast();
}
```

Usamos uma interface para definir nosso TAD fila. A interface só define a API (métodos), a qual pode ser implementada de diversas formas (e usando diferentes estruturas).

# Apêndice III

## Implementação de deques (arranjos)

```java
public class ArrayDeque<E> implements Deque<E> {
  public static final int CAPACITY = 1000;
  private E[] data;
  private int f = 0;
  private int size = 0;

  public ArrayDeque() {this(CAPACITY);}

  public ArrayDeque(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return size; }
  public boolean isEmpty() { return (size == 0); }

  // ...
}
```

# Apêndice III

## Implementação de deques (arranjos)

```java
public class ArrayDeque<E> implements Deque<E> {
  public static final int CAPACITY = 1000;
  private E[] data;
  private int f = 0;
  private int size = 0;

  public ArrayDeque() {this(CAPACITY);}

  public ArrayDeque(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return size; }
  public boolean isEmpty() { return (size == 0); }

  // ...
}
```

- `ArrayDeque` implementa a interface `Deque`.
- O deque possui uma capacidade fixa, dada pela constante `CAPACITY` ou definida pelo usuário no construtor parametrizado.
- Os dados são armazenados no *array* `data`.
- O atributo `f` indica a frente (início) do deque, armazenando o índice do elemento da frente. Essa referência é suficiente para as operações de inserção e remoção em ambos os lados da estrutura (início e fim).
- O tamanho do deque é dado pelo atributo `size`.

# Apêndice III

## Implementação de deques (arranjos)

Método `first`:

```
public E first() {
  if (isEmpty()) return null;
  return data[f];
}
```

# Apêndice III

## Implementação de deques (arranjos)

Método `first`:

```
public E first() {
  if (isEmpty()) return null;
  return data[f];
}
```

Método `last`:

```
public E last() {
  if (isEmpty()) return null;
  int index = f + size - 1;
  if(index >= data.length)
    index -= data.length;
  return data[index];
}
```

# Apêndice III

## Implementação de deques (arranjos)

Método `first`:

```
1 public E first() {
2   if (isEmpty()) return null;
3   return data[f];
4 }
```

Método `last`:

```
1 public E last() {
2   if (isEmpty()) return null;
3   int index = f + size - 1;
4   if(index >= data.length)
5     index -= data.length;
6   return data[index];
7 }
```

- O último elemento está armazenado na posição `f + size - 1`.
- As linhas 4 e 5 implementam a estratégia circular, ajustando o índice quando ele atinge ou ultrapasse o tamanho do *array*.

# Apêndice III

## Implementação de deques (arranjos)

### Método addFirst:

```
1  public void addFirst(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Deque is full");
4    int avail = f - 1;
5    if(avail < 0)
6      avail = data.length - 1;
7    data[avail] = e;
8    f = avail;
9    size++;
10 }
```

# Apêndice III

## Implementação de deques (arranjos)

Método `addFirst`:

```
1  public void addFirst(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Deque is full");
4    int avail = f - 1;
5    if(avail < 0)
6      avail = data.length - 1;
7    data[avail] = e;
8    f = avail;
9    size++;
10 }
```

- O elemento deve ser inserido antes do primeiro, i.e. em `f - 1`.
- As linhas 5 e 6 implementam a estratégia circular, ajustando o índice caso ele assuma valor negativo.
- O índice de inserção é a nova frente `f`.

# Apêndice III

## Implementação de deques (arranjos)

### Método addFirst:

```
1  public void addFirst(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Deque is full");
4    int avail = f - 1;
5    if(avail < 0)
6      avail = data.length - 1;
7    data[avail] = e;
8    f = avail;
9    size++;
10 }
```

### Método addLast:

```
1  public void addLast(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Deque is full");
4    int avail = f + size;
5    if(avail >= data.length)
6      avail -= data.length;
7    data[avail] = e;
8    size++;
9  }
```

- O elemento deve ser inserido antes do primeiro, i.e. em `f - 1`.
- As linhas 5 e 6 implementam a estratégia circular, ajustando o índice caso ele assuma valor negativo.
- O índice de inserção é a nova frente `f`.

# Apêndice III

## Implementação de deques (arranjos)

### Método `addFirst`:

```
1  public void addFirst(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Deque is full");
4    int avail = f - 1;
5    if(avail < 0)
6      avail = data.length - 1;
7    data[avail] = e;
8    f = avail;
9    size++;
10 }
```

- O elemento deve ser inserido antes do primeiro, i.e. em `f - 1`.
- As linhas 5 e 6 implementam a estratégia circular, ajustando o índice caso ele assuma valor negativo.
- O índice de inserção é a nova frente `f`.

### Método `addLast`:

```
1  public void addLast(E e) {
2    if (size == data.length)
3      throw new IllegalStateException("Deque is full");
4    int avail = f + size;
5    if(avail >= data.length)
6      avail -= data.length;
7    data[avail] = e;
8    size++;
9  }
```

- A próxima posição disponível está em `f + size`.
- As linhas 5 e 6 implementam a estratégia circular, ajustando o índice quando ele atinge ou ultrapassa o tamanho do *array*.

# Apêndice III

## Implementação de deques (arranjos)

Método `removeFirst`:

```
public E removeFirst() {
  if (isEmpty()) return null;
  E answer = data[f];
  data[f] = null;
  f = f + 1;
  if(f >= data.length) f = 0;
  size--;
  return answer;
}
```

# Apêndice III

## Implementação de deques (arranjos)

Método `removeFirst`:

```
1 public E removeFirst() {
2   if (isEmpty()) return null;
3   E answer = data[f];
4   data[f] = null;
5   f = f + 1;
6   if(f >= data.length) f = 0;
7   size--;
8   return answer;
9 }
```

- O início é referenciado pelo atributo `f`.
- Após removido o elemento dessa posição, `f` é incrementado.
- A linha 6 implementa a estratégia circular após a atualização de `f`.

# Apêndice III

## Implementação de deques (arranjos)

Método `removeFirst`:

```
1 public E removeFirst() {
2   if (isEmpty()) return null;
3   E answer = data[f];
4   data[f] = null;
5   f = f + 1;
6   if(f >= data.length) f = 0;
7   size--;
8   return answer;
9 }
```

Método `removeLast`:

```
1 public E removeLast() {
2   if (isEmpty()) return null;
3   int index = f + size - 1;
4   if(index >= data.length) index -= data.length;
5   E answer = data[index];
6   data[index] = null;
7   size--;
8   return answer;
9 }
```

- O início é referenciado pelo atributo `f`.
- Após removido o elemento dessa posição, `f` é incrementado.
- A linha 6 implementa a estratégia circular após a atualização de `f`.

# Apêndice III

## Implementação de deques (arranjos)

Método `removeFirst`:

```
1 public E removeFirst() {
2   if (isEmpty()) return null;
3   E answer = data[f];
4   data[f] = null;
5   f = f + 1;
6   if(f >= data.length) f = 0;
7   size--;
8   return answer;
9 }
```

- O início é referenciado pelo atributo `f`.
- Após removido o elemento dessa posição, `f` é incrementado.
- A linha 6 implementa a estratégia circular após a atualização de `f`.

Método `removeLast`:

```
1 public E removeLast() {
2   if (isEmpty()) return null;
3   int index = f + size - 1;
4   if(index >= data.length) index -= data.length;
5   E answer = data[index];
6   data[index] = null;
7   size--;
8   return answer;
9 }
```

- O último elemento está armazenado em `f + size - 1`.
- A linha 4 implementa a estratégia circular, ajustando o índice quando ele atinge ou ultrapassa o tamanho do *array*.

# Apêndice III

## Implementação de deques (arranjos)

Método `toString`:

```
public String toString() {
  StringBuilder sb = new StringBuilder("(");
  int k = f;
  for (int j=0; j < size; j++) {
    if (j > 0)
      sb.append(", ");
    sb.append(data[k]);
    k = (k + 1) % data.length;
  }
  sb.append(")");
  return sb.toString();
}
```

- O método usa o operador módulo (%) para implementar o percurso circular.

# Apêndice III

## Implementação de deques (encadeamento)

```
1  public class LinkedDeque<E> implements Deque<E> {
2    private DoublyLinkedList<E> list = new DoublyLinkedList<>();
3    public int size() { return list.size(); }
4    public boolean isEmpty() { return list.isEmpty(); }
5    public E first() { return list.first(); }
6    public E last() { return list.last(); }
7    public void addFirst(E e) { list.addFirst(e); }
8    public void addLast(E e) { list.addLast(e); }
9    public E removeFirst() { return list.removeFirst(); }
10   public E removeLast() { return list.removeLast(); }
11   public String toString() { return list.toString(); }
12 }
```

# Apêndice III

## Implementação de deques (encadeamento)

```java
public class LinkedDeque<E> implements Deque<E> {
  private DoublyLinkedList<E> list = new DoublyLinkedList<>();
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public E first() { return list.first(); }
  public E last() { return list.last(); }
  public void addFirst(E e) { list.addFirst(e); }
  public void addLast(E e) { list.addLast(e); }
  public E removeFirst() { return list.removeFirst(); }
  public E removeLast() { return list.removeLast(); }
  public String toString() { return list.toString(); }
}
```

- Essa implementação usa uma lista duplamente encadeada, sem limite de capacidade.
  - Encadeamento simples não permitiria remoção do final da estrutura de forma eficiente.
- A implementação é simples, pois basta invocar os métodos da lista encadeada que executam as operações desejadas (`size`, `isEmpty`, `addLast`, ...).

45RPE – Resolução de Problemas com Estruturas de Dados
Prof. Marcelo de Souza