



**INSTITUTO  
FEDERAL**

Sudeste de  
Minas Gerais

Campus  
Manhuaçu

# Estruturas de Dados I

## Deque (ou fila com dois finais)

Prof. Leonardo C. R. Soares - [leonardo.soares@ifsudestemg.edu.br](mailto:leonardo.soares@ifsudestemg.edu.br)

Instituto Federal do Sudeste de Minas Gerais

31 de janeiro de 2025





# Deque

## Descrição

- ▶ Deques são extensões do TAD fila que permitem inserções e remoções em ambas as extremidades da estrutura.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.
- ▶ `removeFirst()`: Remove e retorna o primeiro objeto do deque.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.
- ▶ `removeFirst()`: Remove e retorna o primeiro objeto do deque.
- ▶ `removeLast()`: Remove e retorna o último objeto do deque.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.
- ▶ `removeFirst()`: Remove e retorna o primeiro objeto do deque.
- ▶ `removeLast()`: Remove e retorna o último objeto do deque.

Adicionalmente, podemos definir os seguintes métodos auxiliares:

- ▶ `getFirst()`: Retorna o primeiro elemento do deque.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.
- ▶ `removeFirst()`: Remove e retorna o primeiro objeto do deque.
- ▶ `removeLast()`: Remove e retorna o último objeto do deque.

Adicionalmente, podemos definir os seguintes métodos auxiliares:

- ▶ `getFirst()`: Retorna o primeiro elemento do deque.
- ▶ `getLast()`: Retorna o último elemento do deque.







# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.
- ▶ `removeFirst()`: Remove e retorna o primeiro objeto do deque.
- ▶ `removeLast()`: Remove e retorna o último objeto do deque.

Adicionalmente, podemos definir os seguintes métodos auxiliares:

- ▶ `getFirst()`: Retorna o primeiro elemento do deque.
- ▶ `getLast()`: Retorna o último elemento do deque.
- ▶ `getSize()`: Retorna o número de elementos do deque.





# Operações

O TAD **deque** deve, obrigatoriamente, suportar os métodos:

- ▶ `addFirst(o)`: Insere o objeto no início do deque.
- ▶ `addLast(o)`: Insere o objeto no final do deque.
- ▶ `removeFirst()`: Remove e retorna o primeiro objeto do deque.
- ▶ `removeLast()`: Remove e retorna o último objeto do deque.

Adicionalmente, podemos definir os seguintes métodos auxiliares:

- ▶ `getFirst()`: Retorna o primeiro elemento do deque.
- ▶ `getLast()`: Retorna o último elemento do deque.
- ▶ `getSize()`: Retorna o número de elementos do deque.
- ▶ `isEmpty()`: Retorna um *boolean* indicando se o deque está vazio.





# Deque

A tabela a seguir mostra uma série de operações e seus efeitos em um deque D, inicialmente vazio, de objetos inteiros. Para simplificar, usam-se inteiros em vez de objetos inteiros como argumentos das operações.

<i>Operação</i>	<i>Saída</i>	<i>D</i>
addFirst(3)	–	(3)
addFirst(5)	–	(5,3)
removeFirst()	5	(3)
addLast(7)	–	(3,7)
removeFirst()	3	(3)
removeLast()	7	()
removeFirst()	“error”	()
isEmpty()	true	()





# Implementando um deque

Para que possamos implementar um deque eficiente, garantindo complexidade constante ( $\Theta(1)$ ) para todas as operações, devemos utilizar uma lista duplamente encadeada.





# Implementando um deque

Para que possamos implementar um deque eficiente, garantindo complexidade constante ( $\Theta(1)$ ) para todas as operações, devemos utilizar uma lista duplamente encadeada.

Faremos um deque utilizando *generics*, assim, nosso deque poderá trabalhar com qualquer tipo de objeto. Nossa implementação terá dois elementos *dummy*.

- ▶ Cabeça: o campo próximo deste *dummy* apontará para o primeiro elemento do deque e será o anterior deste.
- ▶ Cauda: o campo anterior deste *dummy* apontará para o último elemento do deque e será o próximo deste.





# Node

Cada **nó** do nosso deque terá um campo *info* contendo o objeto armazenado e referências para os elementos que o sucede e precede.





# Node

```
public class NodeDeque<T> {  
    private NodeDeque<T> proximo;  
    private NodeDeque<T> anterior;  
    private T info;  
    public NodeDeque(T info){  
        this.info = info;  
        proximo = null;  
        anterior = null;  
    }  
    public NodeDeque(){  
        this(info: null);  
    }  
    public NodeDeque<T> getProximo() {  
        return proximo;  
    }  
    public void setProximo(NodeDeque<T> proximo) {  
        this.proximo = proximo;  
    }  
}
```





# Node

```
public NodeDeque<T> getAnterior() {  
    return anterior;  
}  
public void setAnterior(NodeDeque<T> anterior) {  
    this.anterior = anterior;  
}  
public T getInfo() {  
    return info;  
}  
public void setInfo(T info) {  
    this.info = info;  
}  
}
```







# Deque

```
public class Deque <T> {  
    private int tam;  
    NodeDeque<T> cabeca = new NodeDeque<>();  
    NodeDeque<T> cauda = new NodeDeque<>();  
  
    public void addFirst(T element){  
        NodeDeque<T> no = new NodeDeque<>();  
        no.setInfo(element);  
        if (isEmpty()){  
            cabeca.setProximo(no);  
            cauda.setAnterior(no);  
            no.setProximo(cauda);  
            no.setAnterior(cabeca);  
        } else {  
            no.setProximo(cabeca.getProximo());  
            cabeca.getProximo().setAnterior(no);  
            no.setAnterior(cabeca);  
            cabeca.setProximo(no);  
        }  
        tam++;  
    }  
}
```





# Deque

```
public void addLast(T element){
    if (isEmpty()){
        addFirst(element);
    } else {
        NodeDeque<T> no = new NodeDeque<>();
        no.setInfo(element);
        no.setAnterior(cauda.getAnterior());
        no.setProximo(cauda);
        cauda.getAnterior().setProximo(no);
        cauda.setAnterior(no);
    }
    tam++;
}

public T removeFirst() throws Exception {
    if (isEmpty())
        throw new Exception(message: "Deque vazio");
    T no = cabeca.getProximo().getInfo();
    cabeca.setProximo(cabeca.getProximo().getProximo());
    cabeca.getProximo().setAnterior(cabeca);
    tam--;
    return no;
}
```





# Deque

```
public T removeLast() throws Exception{
    if (isEmpty())
        throw new Exception(message: "Deque vazio");
    T no = cauda.getAnterior().getInfo();
    cauda.setAnterior(cauda.getAnterior().getAnterior());
    cauda.getAnterior().setProximo(cauda);
    tam--;
    return no;
}

public void print() throws Exception{
    NodeDeque<T> no = new NodeDeque<>();
    if (isEmpty())
        throw new Exception(message: "Deque vazio");
    no = cabeca.getProximo();
    while (no!=cauda){
        System.out.println(no.getInfo());
        no = no.getProximo();
    }
}

public int size(){ return this.tam; };
public boolean isEmpty(){ return this.tam == 0; };
}
```





# Objeto do deque

Para exemplificar, a utilização do nosso deque, criaremos uma classe chamada Numero. Esta classe terá o atributo inteiro valor. Sobrecreveremos o método toString (Object) para garantir que o método print do deque funcione.





# Numero

```
public class Numero {  
    private int valor;  
    public Numero(int valor) {  
        this.valor = valor;  
    }  
    public int getValor() {  
        return valor;  
    }  
    public void setValor(int valor) {  
        this.valor = valor;  
    }  
    @Override  
    public String toString(){  
        return Integer.toString(valor);  
    }  
}
```





# Aplicação

```
public class App {  
    Run | Debug  
    public static void main(String[] args) {  
        Deque<Numero> deque = new Deque<Numero>();  
        try {  
            for (int i=0; i<10; ++i){  
                if (Math.random()>0.5){  
                    System.out.printf(format: "addFirst(%d)\n", i);  
                    deque.addFirst(new Numero(i));  
                } else {  
                    System.out.printf(format: "addLast(%d)\n", i);  
                    deque.addLast(new Numero(i));  
                }  
            }  
            System.out.println(x: "Deque atual");  
            deque.print();  
        }  
    }  
}
```





# Aplicação

```
for (int i=0; i<5; ++i){  
    if (Math.random()>0.5){  
        System.out.printf(format: "removeFirst(): %d \n",  
            deque.removeFirst().getValor());  
    } else {  
        System.out.printf(format: "removeLast(): %d \n",  
            deque.removeLast().getValor());  
    }  
}  
System.out.println(x: "Deque atual");  
deque.print();  
} catch (Exception e){  
    System.out.println(e.getMessage());  
}  
}
```





# Exercícios

- ▶ Implemente o exemplo apresentado.
- ▶ Implemente uma nova versão do exemplo utilizando arranjo.
- ▶ Implemente um deque que possua apenas um elemento de controle (cabeca). A complexidade das operações fundamentais deve ser mantida constante. (GIT)







# Exercício ao vivo







# Referências

- ▶ GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de Dados & Algoritmos em Java**. Bookman Editora, 2013.
- ▶ ZIVIANI, Nivio. **Projeto de Algoritmos com implementações em Java e C++**, 2007.

