

# ESTRUTURA DE DADOS - JAVA

É uma estrutura que armazena e organiza dados / informações de modo que os dados possam ser acessados e manipulados de forma eficiente.

## VESTORES (ARRAYS - LISTAS)

- estrutura mais simples - sequência de valores do mesmo tipo.

definindo um tamanho

`double[] temperaturas = new double[365];`

• Adiciono elemento no final do vetor.

- Quando necessário eu posso adicionar: `String = null` `int = 0` ou `boolean = false`.

- Atenção: Se a posição for nula adiciona `String` → Quanto tempo isso demora?

+ Se o tamanho (que eu iniciei) for menor que o `length` do vetor adiciono no tamanho o elemento e depois, `tamanho++`.

Se não → throw new Exception (throws Exception)

→ Retornar booleano

o tamanho e imprimir elementos

↳ Usar a classe JAVA (`StringBuffer`), iterar os elementos jogando eles pra dentro do `StringBuffer` (`s.append("[");` `for (...)` `s.append(valor);`

• Obter elemento de uma posição.

- Preciso ter um tratamento `if (posição >= 0 && posição < this.tamanho)`

↑ ! não, e tenho o RANGE que não podemos acessar.

- throw new IllegalArgumentException();

## • VERIFICAR SE O ELEMENTO EXISTE NO VETOR:

- Para comparar String: `equals`
- Busca sequencial. retorna boolean ou int.

## • ADICIONAR ELEMENTO EM QUALQUER POSIÇÃO:

- Não podemos perder um elemento  $\rightarrow$  deslocamos memória.

## • AUMENTAR capacidade do vetor

- Cria uma classe para aumentar vetor.
- Se o tamanho ocupado  $==$  `vetor.length`, cria um novo vetor com capacidade duplicada, itera o principal para o novo, e diz que o principal  $=$  novo.

# PILHA - STACK

Estática; vetor

Last in First out - LIFO

início → base da pilha

- empilhar (push) - adiciona ao topo
- desempilhar (pop) - elimina o topo

refactoring: extends "1"  
↳ super();

protected → a própria  
classe (super) e as  
filhas conseguem  
acessar.

#35 JAVA - RECURS

## • EMPILHAR ELEMENTOS (PUSH)

- na pilha = super.adiciona(elemento);
- aumento a capacidade (memorização)
- p[tamanho] = elemento

## • ESPILHA ELEMENTO DO TOPO (PEEK)

- if vazia - (tamanho-1)

## • Desempilhar (POP)

#Lixo

- normalmente retorna o elemento desempilhado

- if vazia

- return this.elementos[tamanho-1];

• só pega o index certo  
(tamanho-1)

• diminuo o tamanho  
do vetor := desempilha  
(lixo)





## API JAVA : Stack

Stack < generics > stack = new Stack < > ();

stack.isEmpty() - esta Vazio

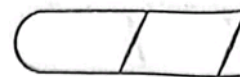
stack.push(item) - Empilhar

stack.size() - tamanho

stack.toString() - toString

stack.peek() - Espiar

stack.pop() - Desempilhar



regio <sup>Fila</sup>  
- Push, Pop, Empty, top, size

QUEUE - size, first, last

NODE - data, next

## FILA - QUEUE

estática

FIFO - first in first out - primeiro a entrar, primeiro a sair.

ENFILEIRAR (em queue)

1	4	5		
10	2	2	3	4

temos 3

Exatamente igual a PILHA → HERANÇA, reutilizar código

Exp: on / Peek - primeiro elemento da fila

Desmf: lerar - dequeue - reutilizar código

ou deslocar todos para uma casa atrás e diminuir o tamanho

elemento[0] = elemento[0 + 1] i = 0 (loop)

## API JAVA QUEUE + Linked List

NÃO existe a classe QUEUE, MAS SIM a INTERFACE QUEUE

- Declara a INTERFACE. new → invoca a classe. MAS só consegue usar os métodos da classe que está descritos na INTERFACE

classe que implementa a interface Queue

Queue < > fila = new LinkedList < > ();

## FILAS COM PRIORIDADE

no método construtor voce

public class " " implements Comparable

interface que pode para declarar  
o método compareTo (obj)

obj1 > obj2 retorna 1

obj1 < obj2 retorna -1

obj1 == obj2 retorna 0

- Na classe do objeto que vai estar na fila, voce coloca um atributo "prioridade".

- Cria o método compareTo

- Na classe de empilhar pra prioridade, no construtor, em vez de só incluir no final, voce declara um int i (pra salvar), abre um loop e compara a prioridade do elemento que voce quer adicionar com as demais da fila. Quando for o momento certo voce puxa o método para adicionar(i, elemento).

Comparable <T> chave = (Comparable <T>) elemento

estou fazendo um

casting para

"chave" ler o método

compareTo.

Assim eu consigo usar

chaves Comparable()

usando implementa Comparable!!

## PRIORITY QUEUE



# LISTA ENCADEADA - estrutura dinâmica -

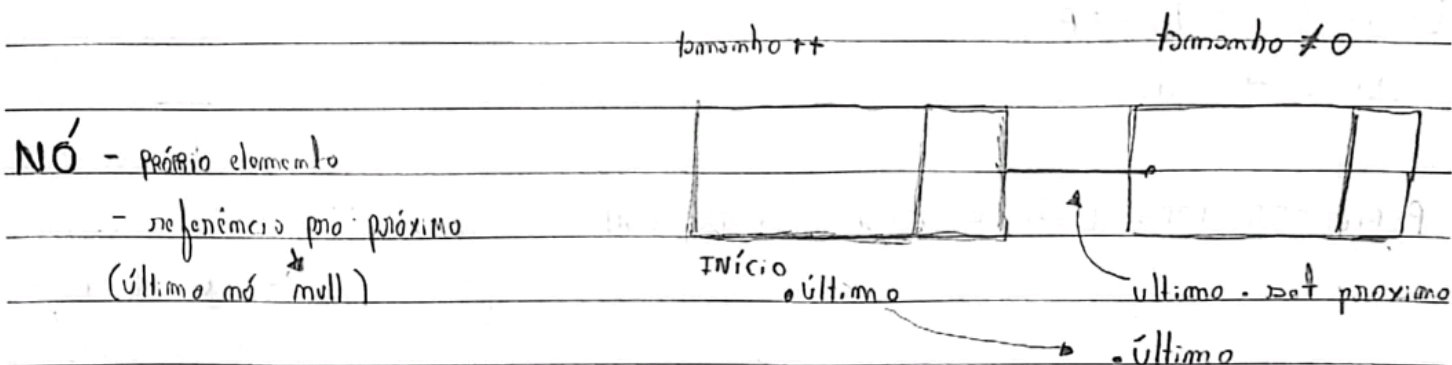
- listas ligadas, listas simplesmente ligadas, lista simplesmente encadeada ==

➤ CADA ELEMENTO É UM NÓ (CÉLULA)


➤ PONTEIROS

➤ REFERÊNCIA AO PRIMEIRO E O ÚLTIMO ELEMENTO DA LISTA

➤ CADA ELEMENTO FAZ REFERÊNCIA AO PRÓXIMO ELEMENTO



Início → null

Início →  NÓ

```
Public void adicionar(T elemento) {
```

```
    No<T> celula = new No<>(elemento); SE FOR 0, INÍCIO E ÚLTIMO SÃO
```

```
    if (this.tamanho == 0) {
```

```
        this.inicio = celula;
```

```
    } else {
```

o nó que é o último nos aponta pro outro nó.

```
        this.ultimo.setProximo(celula);
```

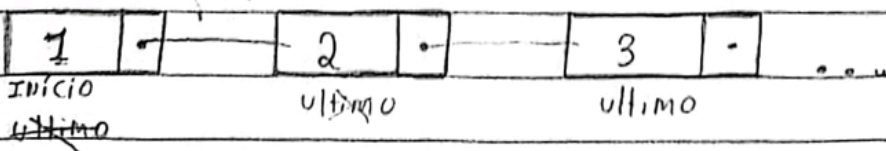
```
    }
```

```
    this.ultimo = celula; → ESSE NOVO NÓ PASSA A SER O ÚLTIMO.
```

```
    this.tamanho ++;
```

```
}
```

(obterando, proximo)



## PERCORRENDO A LISTA:

FORMA CLÁSSICA:

Enquanto `atual.proximo != null` {

`atual = atual.proximo`

}

String Builder

`atual = inicio`

`atual = atual.proximo`

## LIMPAR A LISTA:

`inicio → null` (neste caso perdido #lixo) - mais simples

PERCORRER TODA LISTA E COLOCAR COMO NULL - otimizar memória,

- se ficar indisponível por garbage collection

```
for (Node<T> atual = this.inicio; atual != null; ) {
```

```
Node<T> proximo = atual.getProximo(); → "auxiliar"
```

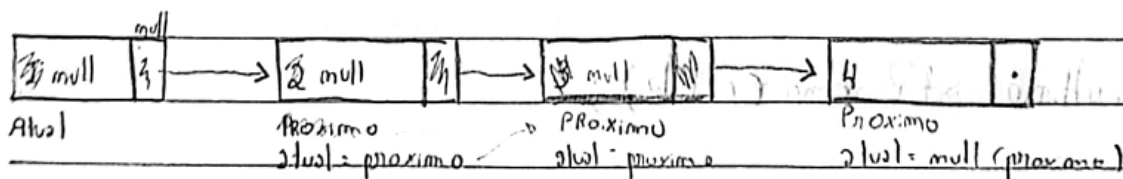
```
atual.setElemento(null);
```

```
atual.setProximo(null);
```

```
atual = proximo;
```

```
}
```

`inicio, ultimo = null` tamanho = 0;





BUSCAR POR ELEMENTO OU POSIÇÃO NA LISTA.

Busca elemento  $\rightarrow$  for ou while para iterar. (-1 para não encontrado)

PARA UM CÓDIGO LIMPO É BOM

VOCÊ DECLARAR UMA VARIÁVEL NO

COMEÇO PARA GUARDAR O -1, POR EXEMPLO

primeiro final int NÃO ENCONTRADO  
brto

Busca Posição  $\rightarrow$  NÃO ESTAMOS INTERESSADO EM RETORNAR TODO O NÓ.

ADICIONAR ELEMENTO EM QUALQUER POSIÇÃO DA LISTA.

adiciona (Posição, elemento)

Fim

- Lista vazia

- Posição == tamanho

o puxa método já existente  
de adicionar

- Adiciona no início

- se for vazio

- se não for vazio

if (Posição < 0 || Posição > tamanho)

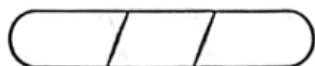
exceção

Node

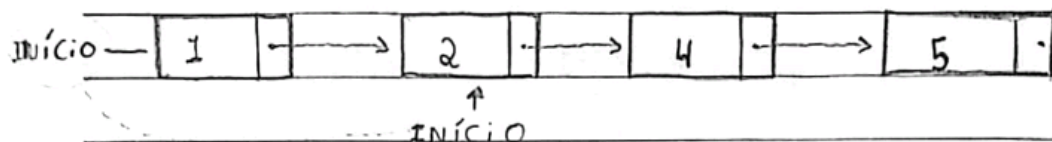
proximo

Novo nó

proximo - proximo



## REMOVER ELEMENTO DO INÍCIO DA LISTA



`início = início.proximo`

VERIFICAR SE A LISTA ESTÁ VAZIA → EXCEPTION

→ GUARDA O ITEM REMOVIDO, PARA RETURN

- `início = início.proximo`

- `tamanho--`;

- SE NO FINAL O TAMANHO FOR `== 0`, `ultimo = null`!!

## REMOVER ELEMENTO DO FINAL DA LISTA



- PERCORRE ATÉ O PENÚLTIMO

- `penultimo.proximo = null`

- atualizar referência do último.

## REMOVER ELEMENTO DE QUALQUER POSIÇÃO.

- manusear até a posição anterior

`x = anterior.getProximo();` = atual

`y = x.proximo`

`anterior.setProximo(y)`

• verificar inteiro

• apresentar código

• resolver problema (novos códigos)