

# CÓDIGO FATAL

A TRILOGIA DOS DADOS  
PROIBIDOS



ALGORITMOS E ESTRUTURAS DE DADOS  
COMO VOCÊ NUNCA VIU ANTES

SAMARA ARAÚJO

*Em um mundo onde os dados foram corrompidos e os algoritmos esquecidos, um novo programador deve enfrentar os três guardiões do código perdido. A cada fase, estruturas ancestrais serão desvendadas, e o caos computacional será confrontado. Bem-vindo à Trilogia dos Dados Proibidos...*

# Capítulo 1 – O Enigma do Labirinto Binário

---

*No coração de um sistema esquecido, existe um labirinto onde cada decisão bifurca destinos — o mundo das Árvores Binárias. Quem dominar seus segredos pode manipular fluxos de informação como um verdadeiro mestre do código. Mas cuidado: nem todos os caminhos revelam a verdade, e alguns levam a armadilhas mortais, onde a complexidade pode aprisionar até o mais sábio.*

Neste capítulo, você vai desvendar...


- A essência das Árvores Binárias e suas ramificações ocultas.
- Como atravessar o labirinto usando buscas eficientes e ordenações mortais.
- O mistério das Árvores Balanceadas, guardiãs do equilíbrio perfeito entre velocidade e segurança.

# 1.1 O Chamado do Labirinto

Imagine uma estrutura onde cada escolha abre um caminho à esquerda ou à direita — uma decisão binária que pode conduzir a respostas rápidas ou a becos sem saída. Esta é a essência da **Árvore Binária**.

## 1.2 Construindo o Guardião Supremo: o Nó Raiz

Vamos criar o bloco fundamental do labirinto: o nó:



```
class No:
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita = None
```

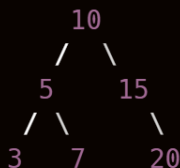
Aqui, cada No tem um valor e pode ter dois filhos:  
- o caminho à esquerda e o caminho à direita.

## 1.3 Criando o Labirinto: Montando a Árvore

Vamos construir uma árvore simples, que será nosso mapa para explorar.

```
# Criando nós
raiz = No(10)
raiz.esquerda = No(5)
raiz.direita = No(15)
raiz.esquerda.esquerda = No(3)
raiz.esquerda.direita = No(7)
raiz.direita.direita = No(20)
```

Visualize este labirinto:



# 1.4 A Jornada da Busca:

## Caminhando pelo Labirinto

*Para desvendar seus segredos, precisamos explorar o labirinto em ordens diferentes. Cada ordem revela um padrão distinto.*

### **Busca In-Order** (Esquerda → Raiz → Direita)

Visitar o filho esquerdo, depois o nó atual, depois o filho direito. Isso revela os valores em ordem crescente.

```
def busca_in_order(no):
    if no:
        busca_in_order(no.esquerda)
        print(no.valor, end=' ')
        busca_in_order(no.direita)

print("Busca In-Order:")
busca_in_order(raiz) # Saída: 3 5
7 10 15 20
```

### **Busca Pre-Order** (Raiz → Esquerda → Direita)

Visitar o nó atual antes dos filhos, útil para salvar a estrutura.

```
def busca_pre_order(no):
    if no:
        print(no.valor, end=' ')

        busca_pre_order(no.esquerda)
        busca_pre_order(no.direita)

print("\nBusca Pre-Order:")
busca_pre_order(raiz) # Saída: 10
5 3 7 15 20
```

## Busca Post-Order (Esquerda → Direita → Raiz)

Visitar os filhos antes do nó, usado para destruição ou limpeza.

```
def busca_post_order(no):  
    if no:  
  
        busca_post_order(no.esquerda)  
  
        busca_post_order(no.direita)  
        print(no.valor, end=' ')  
  
print("\nBusca Post-Order:")  
busca_post_order(raiz) # Saída: 3  
7 5 20 15 10
```



# 1.5 O Código das Árvores Balanceadas: Mantendo o Labirinto Vivo

Imagine que seu labirinto cresça muito, com caminhos cada vez mais longos. Isso pode tornar a busca lenta, como uma armadilha mortal.

**Árvores balanceadas** evitam isso com rotações para manter o equilíbrio. Aqui, um exemplo simplificado de inserção em uma árvore binária de busca (sem balanceamento):

```
def inserir(no, valor):  
    if no is None:  
        return No(valor)  
    if valor < no.valor:  
        no.esquerda =  
        inserir(no.esquerda, valor)  
    else:  
        no.direita =  
        inserir(no.direita, valor)  
    return no  
  
# Inserindo novos valores  
raiz = inserir(raiz, 13)  
raiz = inserir(raiz, 17)
```

## 1.6 Aplicações Ocultas: O Poder das Árvores

Essas estruturas estão por trás de buscas rápidas em bancos de dados, sistemas que entendem comandos, e até a compressão de arquivos.

---

### **Desafio Fatal:** *Sua Missão no Labirinto*

- Crie sua árvore binária.
- Implemente as três buscas vistas.
- Teste inserções e veja como o labirinto cresce.
- Se você conseguir controlar esse enigma, estará preparado para os próximos desafios da trilogia...

# Capítulo 2 – A Maldição da Pilha Espectral

---

*No limiar entre a ordem e o caos, a Pilha Espectral guarda o segredo da reversão fatal. Ela armazena memórias e desfaz ações, mas quem abusar dela pode ser engolido pelo seu próprio rastro, numa espiral sem fim. As lendas contam que os verdadeiros mestres da Pilha conseguem controlar o tempo do programa, viajando entre passado e futuro com comandos precisos.*

Aqui, você vai explorar:

- O funcionamento oculto das Pilhas e suas operações místicas.
- Como usar a Pilha para desfazer o inevitável e reverter o fluxo fatal.
- Aplicações práticas que revelam a verdadeira força por trás do controle temporal no código.

## 2.1 Vozes do Passado

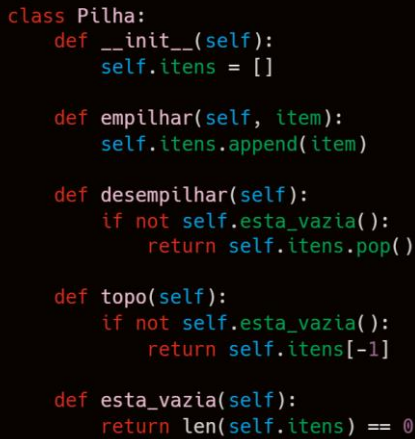
Em sistemas antigos, havia um artefato que permitia desfazer ações. Aqueles que o dominaram, controlaram o tempo do código.

Hoje, esse artefato tem um nome: **Pilha**.

Imagine um altar onde cada item depositado esconde o anterior. O último a chegar é o primeiro a ser revelado — a regra imutável da Pilha: *LIFO (Last In, First Out)*.

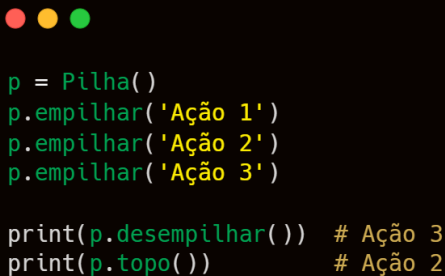
## 2.2 Criando a Pilha

Vamos conjurar a Pilha com as ferramentas da linguagem.

A code block with a dark background and light-colored text. At the top left of the code area are three small colored circles: red, yellow, and green. The code defines a class named 'Pilha' with several methods: '\_\_init\_\_' to initialize an empty list 'itens', 'empilhar' to append an item, 'desempilhar' to pop an item if the stack is not empty, 'topo' to return the top item if the stack is not empty, and 'esta\_vazia' to check if the stack is empty by comparing the length of 'itens' to zero.

```
class Pilha:  
    def __init__(self):  
        self.itens = []  
  
    def empilhar(self, item):  
        self.itens.append(item)  
  
    def desempilhar(self):  
        if not self.esta_vazia():  
            return self.itens.pop()  
  
    def topo(self):  
        if not self.esta_vazia():  
            return self.itens[-1]  
  
    def esta_vazia(self):  
        return len(self.itens) == 0
```

## 2.3 Viagem Temporal com a Pilha

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in Python and demonstrates a stack (Pilha) data structure. It pushes three actions onto the stack and then pops one, showing the last action added.

```
p = Pilha()
p.empilhar('Ação 1')
p.empilhar('Ação 2')
p.empilhar('Ação 3')


print(p.desempilhar()) # Ação 3
print(p.topo())        # Ação 2
```

Cada chamada de **desempilhar()** volta um passo no tempo.

A Pilha é a arma secreta **dos editores de texto, compiladores, jogos com sistema de undo, e até algoritmos recursivos.**

## 2.4 A Pilha em Recursão: Espelhos Invertidos

Toda vez que uma função chama a si mesma, uma nova camada se forma na pilha de execução. Ao atingir o fundo, a função começa a retornar — *desfazendo a pilha*.

A screenshot of a code editor with a dark background. At the top left, there are three colored circles: red, yellow, and green. The code is written in a light green monospace font. It defines a function 'fatorial(n)' that returns 1 if n is 1, and n \* fatorial(n - 1) otherwise. Below the function definition, there is a print statement 'print(fatorial(5))' with a comment '# 120'.

```
def fatorial(n):  
    if n == 1:  
        return 1  
    return n * fatorial(n - 1)  
  
print(fatorial(5)) # 120
```

Cada chamada de `fatorial(n)` aguarda o retorno da próxima — *como uma maldição que só se desfaz ao atingir o valor base*.

---

### Usos Ocultos da Pilha

1. Análise sintática em compiladores
2. Verificação de expressões matemáticas
3. Algoritmos de backtracking (labirintos, sudoku, etc.)

### Desafio Fatal - Caminho Reverso

Crie um programa que leia uma palavra e use uma pilha para imprimi-la ao contrário.

Ex:

```
# Entrada: 'LABIRINTO'  
# Saída: 'OTNIRIBAL'
```

# Capítulo 3 – O Segredo da Fila das Sombras

---

*Escondida nas profundezas do processamento, a Fila das Sombras rege a ordem invisível das tarefas, decidindo quem viverá para ser processado e quem ficará perdido no limbo do esquecimento. Suas regras são simples, mas seu impacto é devastador. Compreender sua lógica é essencial para evitar o caos e o colapso inevitável dos sistemas.*

Neste capítulo sombrio, você vai descobrir:

- A natureza enigmática das Filas e suas variações sombrias.
- Estratégias para controlar o fluxo das informações e evitar a perda fatal de dados.
- Casos reais onde o domínio da Fila foi a diferença entre o sucesso e a ruína digital.



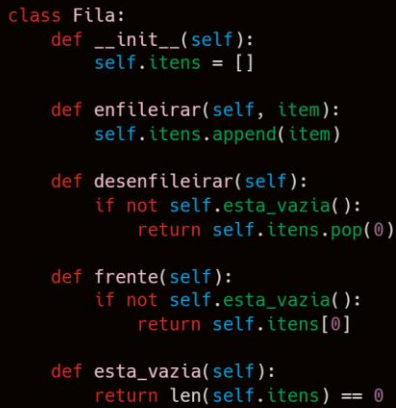
## 3.1 A Ordem Invisível

Em uma cidade governada pela lógica, existe uma fila silenciosa. Aqueles que chegam primeiro têm o direito de partir primeiro.

Seu nome: **Fila**. Sua regra: *FIFO (First In, First Out)*.

A Fila comanda o tempo real, o trânsito de dados, as requisições de impressão, os processos de sistemas operacionais.

## 3.2 Invocando a Fila



```
class Fila:
    def __init__(self):
        self.itens = []

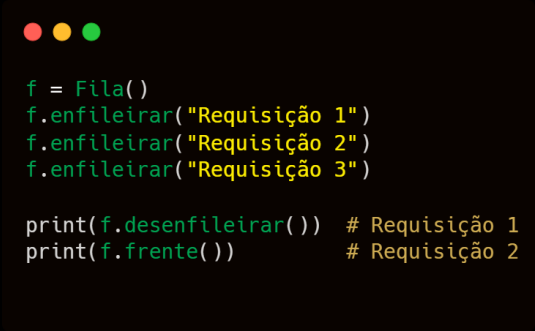
    def enqueue(self, item):
        self.itens.append(item)

    def dequeue(self):
        if not self.esta_vazia():
            return self.itens.pop(0)

    def frente(self):
        if not self.esta_vazia():
            return self.itens[0]

    def esta_vazia(self):
        return len(self.itens) == 0
```

## 3.3 O Ritual da Ordem



```
f = Fila()
f.enfileirar("Requisição 1")
f.enfileirar("Requisição 2")
f.enfileirar("Requisição 3")

print(f.desenfileirar()) # Requisição 1
print(f.frente())       # Requisição 2
```

A fila obedece ao tempo. Nenhum atalho, nenhuma fuga.  
Tudo segue sua ordem.

## 3.4 A Fila nos Bastidores

1. Buffers de rede: pacotes seguem a fila até serem processados.
  2. Agendamento de processos: tarefas esperam por CPU em ordem.
  3. Filas de impressão: documentos saem na sequência enviada.
- 

### Variações Sombras

**Fila Circular:** reutiliza espaço, como um anel infinito.

**Deque (Double-Ended Queue):** permite inserções/remoções nas duas pontas.

**Fila de Prioridade:** aquele que tem mais importância "corta a fila" — mas há um custo sombrio no equilíbrio.

---

### Desafio Fatal – *Fila Circular dos Condenados*

Uma fila amaldiçoada circula infinitamente. Quando o último fala, o primeiro volta ao início do ciclo.

Implemente uma fila circular de tamanho fixo. Quando cheia, novos elementos devem substituir os mais antigos.

# Agradecimentos Finais

Você chegou ao fim desta trilogia — parabéns por atravessar os labirintos, sobreviver às maldições e decifrar os dados proibidos.

Este material foi criado com um único propósito:  
**estudo e aprendizado.**

Nenhum trecho deste material deve ser interpretado como verdade absoluta ou definitivo — o conhecimento evolui, e o verdadeiro programador nunca para de explorar.

Se quiser acompanhar meus projetos, colaborações ou simplesmente trocar ideias sobre código e criatividade, visite meu GitHub:

 <https://github.com/s4mnara>

Obrigado por fazer parte desta jornada. Que você continue evoluindo e desbravando os segredos da computação, um algoritmo por vez.

