

RODRIGO VIANNINI

POSTECH

MACHINE LEARNING ENGINEERING

PYTHON PARA ML E IA

# AULA 04

---

**SUMÁRIO**

O QUE VEM POR AÍ? .....	3
HANDS ON .....	4
SAIBA MAIS.....	7
O QUE VOCÊ VIU NESTA AULA? .....	16
REFERÊNCIAS.....	17

EMSE

## O QUE VEM POR AÍ?

Prepare-se para conhecer e explorar as estruturas de memórias: filas, pilhas e matrizes. Nesta sessão, mergulharemos nos princípios fundamentais desta estrutura, compreendendo sua função na organização e manipulação de dados.

Abordaremos conceitos-chave como FIFO (First In, First Out), LIFO (Last In, Last Out) e a disposição bidimensional das matrizes. Ao final, vocês terão uma compreensão sólida dessas estruturas e estarão prontos(as) para aplicá-las nos contextos computacionais.

## HANDS ON

Nesta aula nosso foco será a implementação de operações básicas em estruturas cruciais, como filas, pilhas e matrizes. Mas, antes, vamos falar sobre estas estruturas de dados. Estas são formas de organizar e armazenar dados em memória de um computador usadas para facilitar o gerenciamento eficiente de informações, permitindo operações específicas de acesso, inserção e remoção de dados.

### Filas (Queue)

São usadas para organizar elementos em uma ordem específica, em que o primeiro elemento a entrar é o primeiro elemento a sair (FIFO - First In, First Out).

**Aplicações comuns:** processamento de tarefas em uma ordem sequencial, como em sistemas de impressão e controle de fluxo em algoritmos de busca, entre outros.

Exemplo:

```
[3] 1  # Biblioteca de estrutura de dados que fornece operações de fila eficientes
    2  from collections import deque
    3
    4  # Criando de uma fila vazia usando a classe "deque"
    5  fila = deque()
    6
    7  # Adicionando elementos ao final da fila (FIFO)
    8  fila.append(1)
    9  fila.append(2)
   10  fila.append(3)
   11
   12  # Removendo o primeiro item da fila
   13  primeiro_elemento = fila.popleft()
   14
   15  # Imprimindo o primeiro elemento da fila
   16  print(f'O primeiro elemento da fila é: {primeiro_elemento}')
```

O primeiro elemento da fila é: 1

Figura 1 – Exemplo de código-fonte Python (filas)  
Fonte: Elaborado pelo autor (2024)

## Pilha (Stack)

São ideais quando a ordem de processamento segue a lógica LIFO (o último elemento adicionado é o primeiro elemento a ser removido)

**Aplicações comuns:** rastreamento de chamadas de funções em execução (pilhas de chamada), operações desfazer/refazer em softwares e gerenciamento de memória em sistemas computacionais.

Exemplo:

```
1  # Criando uma lista vazia que terá a função de uma pilha
2  pilha = []
3
4  # Adicionando elementos ao final da pilha (LIFO)
5  pilha.append(1)
6  pilha.append(2)
7  pilha.append(3)
8
9  # Removendo o último elemento da pilha (primeiro a sair)
10 ultimo_elemento = pilha.pop()
11
12 # Imprimindo o último elemento da fila
13 print(f'O último elemento da fila é: {ultimo_elemento}')
14
```

O último elemento da fila é: 3

Figura 2 – Exemplo de código-fonte Python (pilha)  
Fonte: Elaborado pelo autor (2024)

## Matriz (Matrix)

São estruturas bidimensionais que organizam dados em linhas e colunas. Elas fornecem uma maneira eficiente de armazenar e acessar elementos, em que cada elemento é identificado por suas coordenadas (linha, coluna).

**Aplicações comuns:** utilizadas em diversas áreas, como processamento de imagens, manipulação de dados tabulares em planilhas, resolução de sistemas lineares em álgebra linear e geralmente em situações nas quais os dados têm estrutura bidimensional.

Exemplo:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Definindo uma matriz de intensidades de cinza (valores variam de 10 a 220)
5 imagem = [
6     [120, 50, 200, 30, 80],
7     [90, 180, 25, 160, 120],
8     [40, 220, 75, 100, 60],
9     [150, 70, 110, 190, 140],
10    [20, 130, 170, 10, 220]
11 ]
12
13 # A matriz imagem é convertida em um array numpy
14 imagem_array = np.array(imagem)
15
16 # Exibindo a imagem
17 plt.imshow(imagem_array, cmap='gray', vmin=0, vmax=255)
18 plt.colorbar()
19 plt.show()
20
```

Figura 3 – Exemplo de código-fonte Python (matriz)  
Fonte: Elaborado pelo autor (2024)

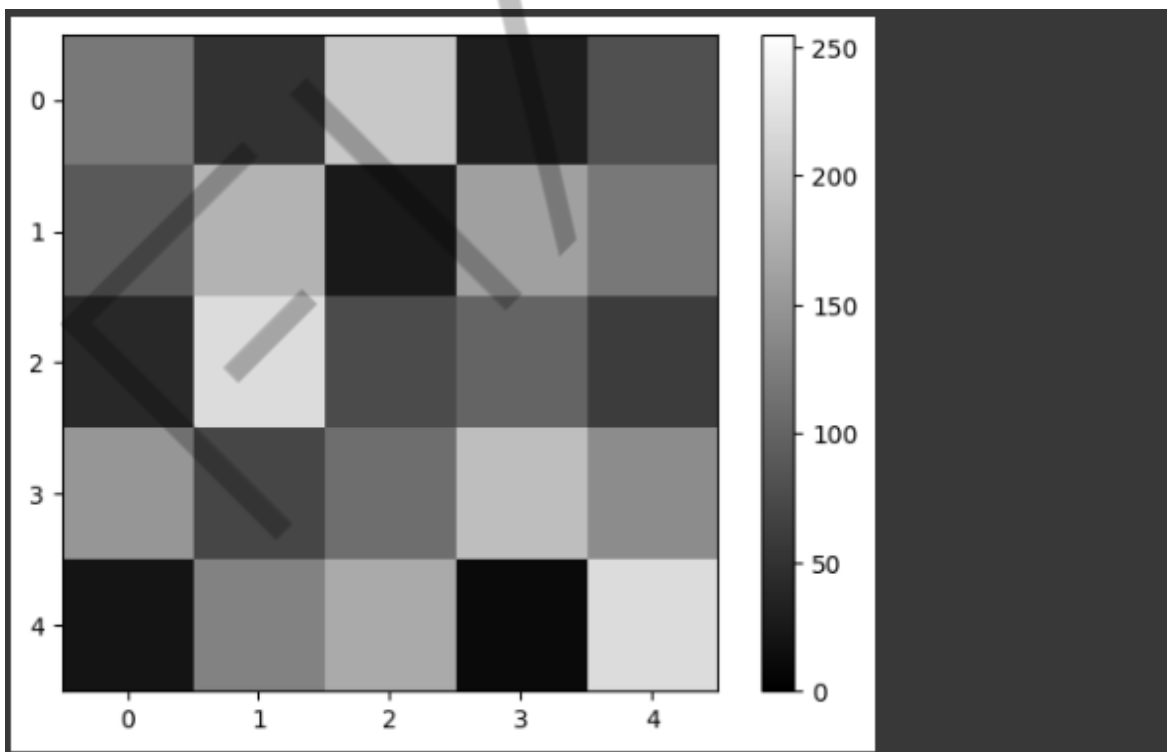


Figura 4 – Exemplo de plot de imagem de uma matriz de imagens  
Fonte: Elaborado pelo autor (2024)

## SAIBA MAIS

Lembramos que, antes de nos aprofundarmos nas estruturas de memória, é fundamental ter uma compreensão sólida dos conceitos básicos. A prática é essencial para o entendimento profundo.

Na programação de computadores, a gestão eficiente de memória é crucial para o desempenho e a funcionalidade dos programas. A memória RAM, atuando como a principal área de armazenamento temporário, desempenha um papel vital na execução de programas, permitindo o acesso aleatório a dados e instruções em tempo real.

Além disso, temos a memória cache com sua capacidade de armazenar dados frequentemente acessados que contribui significativamente para a otimização do desempenho, reduzindo os tempos de acesso à memória principal.

No âmbito das estruturas de dados, as filas, pilhas e matrizes representam abordagens distintas para organização e acesso às informações na memória, cada uma com suas características específicas.

A seguir iremos explorar como ocorre o armazenamento de dados na memória em códigos, destacando os fundamentos envolvidos e as nuances associadas ao tema desta aula.

### Estruturas de memória

#### Tipos de memória - filas

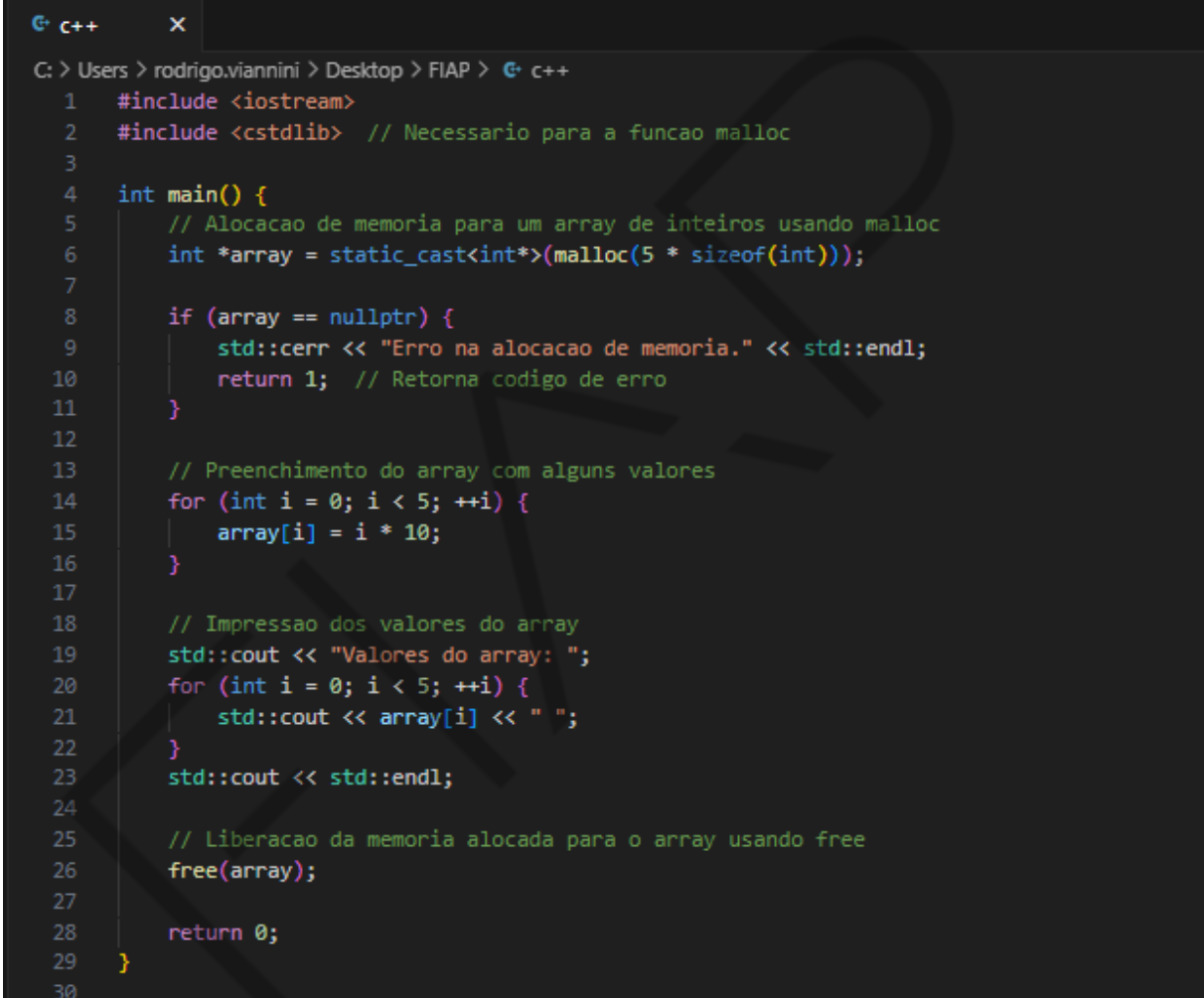
#### Memória RAM

Exerce papel fundamental em sistemas de computadores e serve como uma área temporária de armazenamento para dados que estão sendo usados ativamente ou processados pela CPU. A principal característica da RAM é o acesso aleatório, o que significa que qualquer posição de memória pode ser acessada diretamente, sem a necessidade de percorrer as posições anteriores.

Quando um programa é executado, suas instruções/dados são carregados na memória RAM para que a CPU possa acessá-la rapidamente. A memória AM é volátil, ou seja, os dados armazenados são perdidos quando o computador é desligado.

A manipulação da memória RAM por códigos de programação ocorre através da alocação e deslocação de blocos de memória. Em linguagem C ou C++, por exemplo, as pessoas programadoras têm acesso direto sobre esta alocação de memória usando as funções “malloc” e “free”.

Exemplo:



```
C: > Users > rodrigo.viannini > Desktop > FIAP > c++
1  #include <iostream>
2  #include <cstdlib> // Necessario para a funcao malloc
3
4  int main() {
5      // Alocação de memória para um array de inteiros usando malloc
6      int *array = static_cast<int*>(malloc(5 * sizeof(int)));
7
8      if (array == nullptr) {
9          std::cerr << "Erro na alocação de memória." << std::endl;
10         return 1; // Retorna código de erro
11     }
12
13     // Preenchimento do array com alguns valores
14     for (int i = 0; i < 5; ++i) {
15         array[i] = i * 10;
16     }
17
18     // Impressão dos valores do array
19     std::cout << "Valores do array: ";
20     for (int i = 0; i < 5; ++i) {
21         std::cout << array[i] << " ";
22     }
23     std::cout << std::endl;
24
25     // Liberação da memória alocada para o array usando free
26     free(array);
27
28     return 0;
29 }
30
```

Figura 5 – Exemplo de código-fonte C++ (alocação de memória)  
Fonte: Elaborado pelo autor (2024)

## Memória Cache

É um tipo de memória de acesso rápido que armazena cópias de dados frequentemente acessados da memória principal (RAM). Sua principal função é melhorar o desempenho, reduzindo o tempo de acesso aos dados mais utilizados.



Existem níveis de cache (L1, L2, L3) que estão integrados aos processadores modernos; desta forma, quanto mais próximo o cache estiver da CPU, mais rápido será o acesso aos dados.

#### **L1 cache (cache de nível 1):**

- **Localização:** mais próxima da CPU.
- **Função:** armazena dados e instruções usados com mais frequência.
- **Tamanho:** geralmente menor, em torno de alguns kilobytes (KB).
- **Velocidade:** muito rápida.

#### **L2 cache (cache de nível 2):**

- **Localização:** entre a L1 e a L3.
- **Função:** fornece uma camada adicional de armazenamento temporário para dados e instruções.
- **Tamanho:** maior que a L1, geralmente em megabytes (MB).
- **Velocidade:** mais lenta do que a L1, mas ainda mais rápida que a RAM principal.

#### **L3 cache (cache de nível 3):**

- **Localização:** mais distante da CPU, compartilhada entre núcleos ou até mesmo entre vários processadores em alguns sistemas.
- **Função:** oferece uma camada adicional de armazenamento compartilhada entre os núcleos.
- **Tamanho:** geralmente o maior dos três níveis, variando de alguns megabytes a dezenas de megabytes.

**Velocidade:** mais lenta em comparação com L1 e L2, mas ainda muito mais rápida do que a RAM principal.

A ideia por trás dessa hierarquia é explorar o princípio da localidade, que sugere que os programas tendem a acessar um pequeno conjunto de dados e instruções frequentemente.

Os caches mais próximos da CPU (L1 e L2) são menores e mais rápidos e otimizados para armazenar dados e instruções frequentemente utilizados. O L3, por ser maior, é mais capaz de armazenar um conjunto maior de dados compartilhados entre núcleos.

O sistema de gerenciamento de cache é geralmente automático, mas especialistas em programação podem otimizar o desempenho escrevendo o código de maneira a tirar melhor proveito dos padrões de acesso à memória.

## **Filas e sua Utilização de Memória**

As filas utilizam a memória RAM e a cache de forma significativa, influenciando diretamente o desempenho e a eficiência de operações relacionadas a essas estruturas de dados.

### **Memória RAM**

#### **Alocação Sequencial**

Filas, seguindo o princípio FIFO (First In, First Out), geralmente são implementadas de forma sequencial em termos de alocação de memória. Os elementos são armazenados em posições contíguas (referem-se à localização consecutiva de elementos em uma estrutura de dados, como filas, em termos de alocação de memória) da memória, seja utilizando arrays ou listas encadeadas.

#### **Operações de Enfileirar e Desenfileirar**

As operações básicas de enfileirar e desenfileirar podem resultar em acesso eficiente à memória RAM, especialmente se a implementação da fila for otimizada para minimizar deslocamentos e acessos desnecessários.

## **Uso Eficiente da RAM**

Ao manter uma organização sequencial dos dados, as filas podem aproveitar a capacidade da RAM de acessar endereços de forma aleatória, garantindo um acesso rápido aos elementos, principalmente aqueles localizados no início e no final da fila.

## **Memória Cache**

### **Localidade Temporal e Espacial**

Filas, ao preservarem a ordem de chegada dos dados, podem explorar a localidade temporal, já que elementos frequentemente acessados recentemente podem estar próximos uns dos outros na memória.

A localidade espacial também é relevante, visto que dados adjacentes na fila têm maior probabilidade de serem armazenados em blocos de cache consecutivos.

### **Otimização do Desempenho**

Operações frequentes de enfileirar e desenfileirar podem se beneficiar da cache, já que ela armazena temporariamente dados recentemente acessados, reduzindo a latência de acesso à memória principal.

### **Considerações de Tamanho da Cache**

A eficiência da cache também está relacionada ao tamanho da fila e à política de substituição de cache. Filas menores ou implementações que favoreçam a localidade temporal podem resultar em melhor desempenho da cache.

### **Cache Misses**

Em situações em que a fila é grande e os dados não estão localizados em blocos consecutivos pode haver cache misses, o que pode impactar negativamente o desempenho.

### **Considerações Finais**

O uso eficiente da memória RAM e da cache por filas depende da implementação específica e do contexto de aplicação. O acesso sequencial e a preservação da ordem de chegada são características-chave que podem influenciar positivamente o desempenho, aproveitando as características da memória RAM e cache para operações rápidas e eficientes.

**Exemplo:**

- **Objetivo:** implementação de um Sistema de Atendimento ao Cliente com Filas de Espera.
- **Cenário:** um sistema de atendimento ao cliente em que clientes aguardam atendimento em uma fila.
- **Implementação:** utilização de uma fila para gerenciar a ordem de chegada de clientes.
- Operações:
  - Enfileirar: adição de clientes à fila.
  - Desenfileirar: atendimento ao próximo cliente.
  - Benefícios: garante atendimento na ordem de chegada, organizando eficientemente os pedidos de clientes.

**Pilhas e sua Utilização de Memória**

As pilhas utilizam a memória RAM e a cache de forma significativa, influenciando diretamente o desempenho e a eficiência de operações relacionadas a essas estruturas de dados.

**Alocação Sequencial**

Pilhas, seguindo o princípio LIFO (Last In, First Out), são geralmente implementadas de forma sequencial em termos de alocação de memória. Os elementos são armazenados em posições contíguas da memória, seja utilizando arrays ou listas encadeadas.

**Operações de Empilhar e Desempilhar**

As operações básicas de empilhar e desempilhar podem resultar em acesso eficiente à memória RAM, especialmente se a implementação da pilha for otimizada para minimizar deslocamentos e acessos desnecessários.

**Uso Eficiente da RAM**

Ao manter uma organização sequencial dos dados, as pilhas podem aproveitar a capacidade da RAM de acessar endereços de forma aleatória, garantindo um

acesso rápido aos elementos, principalmente aqueles localizados no topo e na base da pilha.

## **Memória cache**

### **Localidade temporal e espacial**

Pilhas, ao preservarem a ordem de chegada dos dados, podem explorar a localidade temporal, já que elementos frequentemente acessados recentemente podem estar próximos uns dos outros na memória.

A localidade espacial também é relevante, posto que dados adjacentes na pilha têm maior probabilidade de serem armazenados em blocos de cache consecutivos.

### **Otimização do Desempenho**

Operações frequentes de empilhar e desempilhar podem se beneficiar da cache, já que ela armazena temporariamente dados recentemente acessados, reduzindo a latência de acesso à memória principal.

### **Considerações de Tamanho da Cache**

A eficiência do cache também está relacionada ao tamanho da pilha e à política de substituição de cache. Pilhas menores ou implementações que favoreçam a localidade temporal podem resultar em melhor desempenho da cache.

### **Cache Misses**

Em situações em que a pilha é grande e os dados não estão localizados em blocos consecutivos podem ocorrer cache misses, o que pode impactar negativamente o desempenho.

### **Considerações Finais**

O uso eficiente da memória RAM e da cache por pilhas depende da implementação específica e do contexto de aplicação. O acesso sequencial e a preservação da ordem de chegada são características-chave que podem influenciar positivamente o desempenho, aproveitando as características da memória RAM e cache para operações rápidas e eficientes.

### **Exemplo:**

- **Objetivo:** implementação de um Editor de Texto com Recurso de Desfazer (Undo).

- **Cenário:** um editor de texto em que o usuário pode desfazer a última ação realizada.
- **Implementação:** utilização de uma pilha para armazenar as ações do usuário.
- Operações:
  - Empilhar: adição de cada ação à pilha.
  - Desempilhar: desfazer a última ação.
  - Benefícios: oferece uma funcionalidade de desfazer eficiente, permitindo ao usuário reverter ações anteriores.

### Matrizes e sua Utilização de Memória

As matrizes desempenham um papel fundamental na computação, utilizando de maneira significativa a memória RAM e a cache para influenciar diretamente o desempenho e a eficiência de operações relacionadas a essas estruturas de dados. Assim como na organização de dados em arrays unidimensionais, as matrizes estendem esse conceito para duas ou mais dimensões, permitindo uma representação eficiente de conjuntos de dados tabulares.

Ao acessar elementos em uma matriz, a localidade espacial torna-se crucial para otimização utilizando a cache, minimizando os tempos de acesso à memória e, por conseguinte, melhorando a eficiência operacional. Essa consideração é especialmente relevante em operações intensivas como multiplicação de matrizes, em que a maximização do uso da cache pode resultar em ganhos substanciais de desempenho.

A memória RAM, por sua vez, desempenha um papel vital na alocação e armazenamento de matrizes. A eficiência do acesso à memória é essencial para garantir que as operações realizadas em elementos da matriz ocorram de maneira eficaz. Estratégias como o uso de arrays contíguos (todos os elementos do array estão localizados lado a lado na memória, sem lacunas entre eles) na memória ajudam a melhorar a localidade espacial, facilitando o acesso rápido e contínuo aos elementos da matriz.

Além disso, a escolha de algoritmos e técnicas de processamento de matrizes também impacta diretamente na eficiência global do sistema. Algoritmos otimizados para aproveitar características específicas da arquitetura de memória e cache podem resultar em melhor desempenho, contribuindo para a eficácia das operações relacionadas a matrizes.

Resumindo, a manipulação eficiente de matrizes está intrinsecamente ligada à utilização cuidadosa da memória RAM e da cache. O entendimento profundo desses conceitos é crucial para desenvolver algoritmos e implementações que maximizem o desempenho e a eficiência em operações que envolvem essas estruturas de dados multidimensionais.

No início do aprendizado o foco é na funcionalidade do algoritmo; mas, com a experiência e o aperfeiçoamento das novas habilidades, é necessário o olhar cuidadoso para a performance no desenvolvimento de aplicações. Desta forma, é possível alcançar um papel de destaque no mercado de trabalho.

**Exemplo:**

- **Objetivo:** implementação de um Sistema de Reservas de Assentos em um Teatro.
- **Cenário:** um sistema de reservas de assentos para um teatro com múltiplas fileiras e colunas.
- **Implementação:** utilização de uma matriz para representar o layout dos assentos.
- **Operações:**
  - Acesso à Matriz: verificação da disponibilidade de assentos.
  - Atualização da Matriz: marcação de assentos reservados.
  - Benefícios: oferece uma representação organizada e eficiente dos assentos disponíveis, facilitando a reserva e o gerenciamento.

## O QUE VOCÊ VIU NESTA AULA?

Nessa aula exploramos os princípios fundamentais das estruturas de memória, principalmente relacionadas a filas, pilhas e matrizes, que são essenciais no desenvolvimento de algoritmos otimizados.

Exploramos como essas estruturas de dados influenciam a eficiência dos algoritmos, proporcionando uma visão mais abrangente sobre a importância prática desses conceitos na programação.

A cada novo conhecimento adquirido conseguimos melhorar nossa habilidade de desenvolvimento de soluções mais eficientes e elegantes.



## REFERÊNCIAS

HENNESSY, J. L.; PATTERSON, D. A. **Arquitetura de Computadores: Uma Abordagem Quantitativa**. [s.l.]: Elsevier, 2013.

TANENBAUM, A. S.; WOODHULL, A. S. **Organização Estruturada de Computadores**. [s.l.]: Pearson, 2010.

TOSI, D. **Estruturas de Dados em C**. [s.l.]: Bookman, 2007.

EMANIP

## **PALAVRAS-CHAVE**

**Palavras-chave:** Organização de Computadores. Arquitetura de Computadores. Estruturas de Dados.

EMENDADO



POSTECH