

# Prof. esp. Thalles Canela

- **Graduado:** Sistemas de Informação - Wyden Facimp
- **Pós-graduado:** Segurança em redes de computadores - Wyden Facimp
- **Professor:** Todo núcleo de T.I. (Graduação e Pós) - Wyden Facimp
- **Diretor:** SCS
- **Gerente de Projetos:** Motoca Systems

## Redes sociais:

- **Linkedin:** <https://www.linkedin.com/in/thalles-canela/>
- **YouTube:** <https://www.youtube.com/aXR6CyberSecurity>
- **Facebook:** <https://www.facebook.com/axr6PenTest>
- **Instagram:** [https://www.instagram.com/thalles\\_canela](https://www.instagram.com/thalles_canela)
- **Github:** <https://github.com/ThallesCanela>
- **Github:** <https://github.com/aXR6>
- **Twitter:** <https://twitter.com/Axr6S>

# Listas, Pilhas e Filas

Estruturas de dados

---



# Listas - Conceito e Exemplo da Vida Real

- Listas são estruturas de dados lineares onde cada elemento aponta para o próximo, semelhante a uma lista de compras.
  - **Exemplo da vida real:**
  - Pense em uma lista de reprodução de músicas. Você pode adicionar, remover e pular músicas.
-



# Tipos de Listas:

- **Listas Simplesmente Encadeadas:** Cada nó possui um ponteiro para o próximo nó. O último nó aponta para NULL, indicando o final da lista.
  - **Listas Duplamente Encadeadas:** Cada nó possui dois ponteiros: um para o próximo nó e outro para o nó anterior. Dão mais flexibilidade na hora de realizar operações como inserção e exclusão.
  - **Listas Circulares:** Semelhante à lista simplesmente encadeada, mas o último nó, em vez de apontar para NULL, aponta de volta para o primeiro nó.
-



# Por que usar Listas?

- **Dinamismo:** A memória é alocada conforme necessário, diferentemente de arrays que possuem tamanho fixo.
  - **Inserção e Remoção Eficiente:** Podemos inserir ou remover elementos em qualquer posição sem a necessidade de deslocar outros elementos, como em arrays.
-



# Desvantagens:

- **Acesso Sequencial:** Ao contrário de arrays, não podemos acessar um elemento da lista diretamente por seu índice. Isso pode tornar o acesso mais lento.
  - **Maior Consumo de Memória:** Devido ao armazenamento de ponteiros adicionais para cada elemento.
-



# Operações Básicas:

- **Inserção:** Adicionar um elemento no começo, meio ou fim da lista.
  - **Remoção:** Retirar um elemento da lista.
  - **Busca:** Localizar um elemento na lista.
-

# Código - Lista

```
1  #include <stdio.h>      // Importa a biblioteca padrão de entrada e saída
2  #include <stdlib.h>     // Importa a biblioteca padrão de funções gerais, como alocação de memória
3
4  // Definindo a estrutura de um nó
5  typedef struct Node {   // Define um novo tipo de estrutura chamado Node
6      int data;           // Armazena o dado (neste caso, um inteiro)
7      struct Node* next;  // Ponteiro para o próximo nó na lista
8  } Node;
9
10 // Função para criar um novo nó
11 Node* createNode(int data) { // Função que aceita um valor inteiro e retorna um ponteiro para um Node
12     Node* newNode = (Node*) malloc(sizeof(Node)); // Aloca memória dinamicamente para um novo nó
13     newNode->data = data; // Atribui o dado passado como parâmetro ao novo nó
14     newNode->next = NULL; // Inicialmente, o novo nó não aponta para nenhum outro nó
15     return newNode;       // Retorna o ponteiro para o novo nó criado
16 }
```






# Lista

- Aqui, definimos a estrutura básica de uma lista simplesmente encadeada. Cada **Node** contém um dado e um ponteiro para o próximo nó.
  - **Dicionário:**
    - **typedef:** Usado para criar um novo tipo.
    - **struct:** Define uma estrutura.
    - **malloc:** Função padrão em C para alocar memória dinamicamente.
-




# Pilhas - Conceito e Exemplo da Vida Real

- Pilhas operam pelo princípio LIFO (Last In, First Out). Pense em uma pilha de livros. O último livro que você coloca é o primeiro que você retira.
  - **Exemplo da vida real:**
  - Voltando a um documento após realizar várias edições e desejando desfazer as alterações. As alterações mais recentes são as primeiras a serem desfeitas.
-



# Existem duas operações principais associadas a pilhas:

- **Push:** Adiciona um item ao topo da pilha.
  - **Pop:** Remove o item do topo da pilha.
  - Além dessas, frequentemente temos uma operação adicional chamada "Peek" ou "Top", que nos permite ver o item no topo da pilha sem removê-lo.
-



# Você pode se perguntar, onde usamos pilhas no mundo real da computação? Aqui estão alguns exemplos:

- **Navegadores da Web:** Quando você navega por páginas da web e clica no botão voltar, você está navegando através de uma pilha de páginas visitadas anteriormente.
  - **Softwares de Edição:** Ao usar funções de desfazer e refazer.
  - **Análise de Expressões:** Como verificar se os parênteses em uma expressão estão balanceados.
  - **Chamadas de Função:** Quando as funções são chamadas em muitos programas, a memória para a função é "empilhada" em cima da chamada anterior.
-

# Código - Pilha

```
1 typedef struct Stack { // Define um novo tipo de estrutura chamado Stack
2     |   Node* top;      // Ponteiro para o topo da pilha
3 } Stack;
4
5 // Função para empilhar um item
6 void push(Stack* stack, int data) { // Função que aceita um ponteiro para Stack e um valor inteiro
7     |   Node* newNode = createNode(data); // Cria um novo nó com o dado passado
8     |   newNode->next = stack->top; // O novo nó aponta para o atual topo da pilha
9     |   stack->top = newNode; // Atualiza o topo da pilha para o novo nó
10 }
11
12 // Função para desempilhar um item
13 int pop(Stack* stack) { // Função que aceita um ponteiro para Stack e retorna um inteiro
14     |   if (stack->top == NULL) return -1; // Verifica se a pilha está vazia e retorna -1 se verdadeiro
15     |   int data = stack->top->data; // Obtém o dado do nó no topo da pilha
16     |   Node* temp = stack->top; // Salva o ponteiro do nó do topo para ser liberado depois
17     |   stack->top = stack->top->next; // Atualiza o topo da pilha para o próximo nó
18     |   free(temp); // Libera a memória do nó desempilhado
19     |   return data; // Retorna o dado do nó desempilhado
20 }
```



# Pilha

- Aqui, criamos as funções básicas de uma pilha. A função **push** adiciona um item ao topo, enquanto **pop** remove e retorna o item do topo.
  - **Dicionário:**
  - **free:** Função padrão em C para liberar memória dinamicamente alocada.
-



# Filas - Conceito e Exemplo da Vida Real

- Filas operam pelo princípio FIFO (First In, First Out). Pense em uma fila de supermercado.
  - **Exemplo da vida real:**
  - Clientes esperando na fila do banco. O primeiro cliente a chegar é o primeiro a ser atendido.
-



# Conceito Básico

- Em termos computacionais, uma fila é uma coleção ordenada de itens onde a adição de novos itens acontece no final, chamado de 'rear', e a remoção ocorre na frente, conhecida como 'front'.
  - A característica mais importante da fila é que ela segue a abordagem FIFO - First In, First Out, que significa que o primeiro elemento adicionado à fila será o primeiro a ser removido.
-






# Por que usar uma Fila?

- **Vocês podem se perguntar:** por que usaríamos uma fila em computação? Bem, as filas são usadas quando os dados são transferidos de maneira assíncrona entre dois processos.
  - Por exemplo, IO Buffers, pipes, sistemas de gerenciamento de tarefas em sistemas operacionais e muitos outros cenários.
-



# Operações Básicas

- **Enqueue:** Adicionar um item ao final da fila.
  - **Dequeue:** Remover o item da frente da fila.
  - **Front:** Obter o item da frente sem removê-lo.
  - **Rear:** Obter o item do final sem removê-lo.
-



# Existem algumas variações do conceito básico de fila, incluindo:

- **Fila Circular:** Onde o último elemento aponta de volta para o primeiro.
  - **Fila de Prioridade:** Onde cada elemento tem uma prioridade e a remoção é baseada nessa prioridade.
  - **Fila Duplamente Terminada (Deque):** Onde você pode inserir ou remover itens de ambas as extremidades.
-

# Código - Fila

```
1  typedef struct Queue {           // Define um novo tipo de estrutura chamado Queue
2      Node* front;                 // Ponteiro para o início da fila
3      Node* rear;                  // Ponteiro para o final da fila
4  } Queue;
5
6  // Função para enfileirar um item
7  void enqueue(Queue* queue, int data) {           // Função que aceita um ponteiro para Queue e um valor inteiro
8      Node* newNode = createNode(data);           // Cria um novo nó com o dado passado
9      if (queue->rear == NULL) {                   // Se a fila estiver vazia
10         queue->front = queue->rear = newNode;     // O novo nó é tanto o início quanto o final da fila
11         return;
12     }
13     queue->rear->next = newNode;                   // O atual último nó aponta para o novo nó
14     queue->rear = newNode;                         // Atualiza o final da fila para o novo nó
15 }
16
17 // Função para desenfileirar um item
18 int dequeue(Queue* queue) {                   // Função que aceita um ponteiro para Queue e retorna um inteiro
19     if (queue->front == NULL) return -1;         // Verifica se a fila está vazia e retorna -1 se verdadeiro
20     int data = queue->front->data;                // Obtém o dado do nó na frente da fila
21     Node* temp = queue->front;                   // Salva o ponteiro do nó da frente para ser liberado depois
22     queue->front = queue->front->next;             // Atualiza o início da fila para o próximo nó
23     if (queue->front == NULL) queue->rear = NULL; // Se a fila estiver vazia após a operação, atualiza o final para NULL
24     free(temp);                                  // Libera a memória do nó desenfileirado
25     return data;                                 // Retorna o dado do nó desenfileirado
26 }
```



# Fila

- Aqui, implementamos funções básicas para uma fila. **enqueue** adiciona um item ao final, enquanto **dequeue** remove e retorna o item da frente.
  - **Dicionário:**
  - **NULL:** Constante que representa um ponteiro vazio ou zero em C.
-