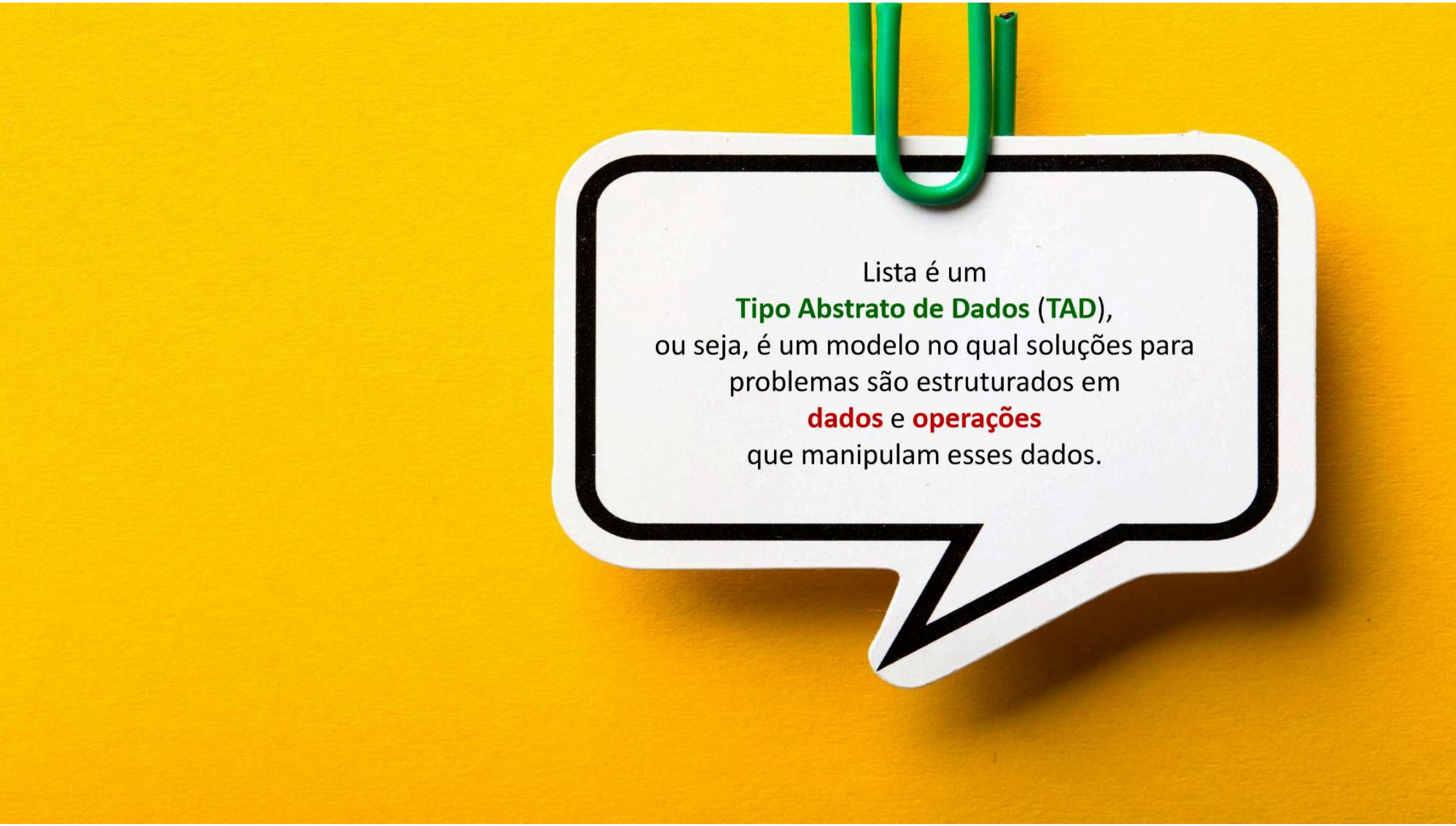
Estrutura de Dados

2° semestre de 2025

Listas, filas e pilhas

Tema #06

Professor Marcelo Eustáquio


Lista é um
Tipo Abstrato de Dados (TAD),
ou seja, é um modelo no qual soluções para
problemas são estruturados em
dados e operações
que manipulam esses dados.

Listas, filas e pilhas

Lista é um tipo abstrato de dados, ou seja, é definido como um modelo matemático por meio de um par **(v, o)** em que **v** é um conjunto de valores e **o** é um conjunto de operações sobre esses valores.

No TAD Fração, os dados são definidos a partir de um par de valores, que desempenham papéis de numerador e denominador. Por outro lado, são operações possíveis aquelas que dizem respeito às que envolvem números racionais (simplificação, soma, subtração, multiplicação, divisão, potenciação, radiciação, comparação, e tantas outras que poderiam ser explicitadas).

Para refletir ...

Quais as vantagens (ou benefícios) de se utilizar de TADs na estruturação de códigos-fonte?

Em **C** (**typedef** e **struct**) podem (e devem) ser empregados na construção de Tipos Abstratos de Dados.

Ou seja, o modelo matemático ({dados}, {operações}) é, via de regra, construído a partir do agrupamento lógico de um conjunto de tipos de dados, definidos a partir de uma structs, e de um conjunto de operações, materializados na implementação de funções.

Utilizaremos essa dinâmica para tratar dos seguintes conceitos:

Listas + Filas + Pilhas + Árvores + Grafos

Listas, filas, pilhas, árvores e grafos

Uma **lista** é uma estrutura linear, ou seja, seus elementos estão dispostos em sequência — cada elemento (ou nó, em algumas implementações) tem um sucessor, exceto o último, e, em listas duplamente encadeadas, também um antecessor.

*A imagem mostra um quadro de gestão de **lista de tarefas**, semelhante a ferramentas como Monday.com, Trello ou Asana, usado para acompanhar o andamento de atividades de um projeto. Cada item é uma **tarefa** e possui atributos como nome, responsável, status, cronograma, avaliação, e outros que poderiam ser acrescentados.*

Listas, filas, pilhas, árvores e grafos

Uma **fila (queue)** é uma estrutura de dados linear em que o **primeiro elemento a entrar é o primeiro a sair**, ou seja, segue o princípio **FIFO — *First In, First Out***.

Uma fila de pessoas esperando atendimento

Em uma fila de espera, as pessoas que primeiro chegam são as que são primeiro atendidas e as que por ultimo chegam são as últimas as serem atendidas: esta é a dinâmica em filas de supermercado, de banco, de atendimento em posto de saúde, e em várias outras situações quotidianas.

Listas, filas, pilhas, árvores e grafos

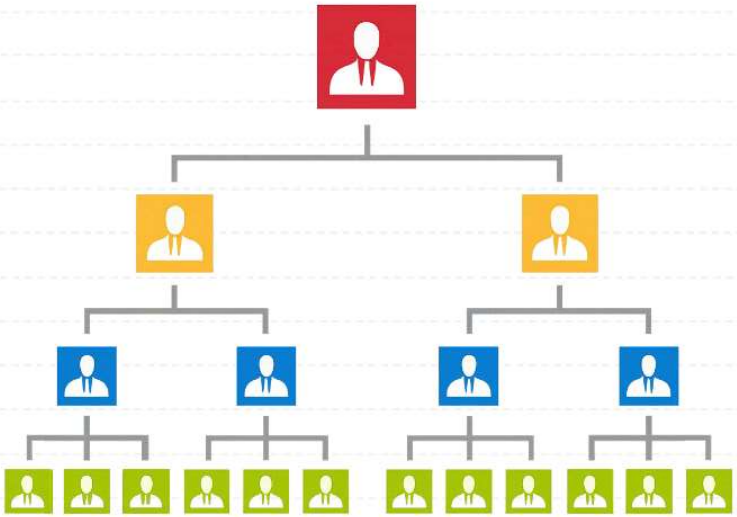
Uma **pilha (stack)** é uma estrutura de dados linear que segue o **primeiro elemento a entrar é o último a sair**, ou seja, segue o princípio **LIFO — Last In, First Out.**

Uma pilha de pratos.

Em uma pilha de pratos, o último prato colocado é o primeiro a ser retirado e o primeiro a ser colocado é o último a sair: essa é a lógica das pilhas de louça na cozinha, das chamadas de funções em um programa de computador e de diversas outras situações em que o que entra por último precisa sair primeiro.

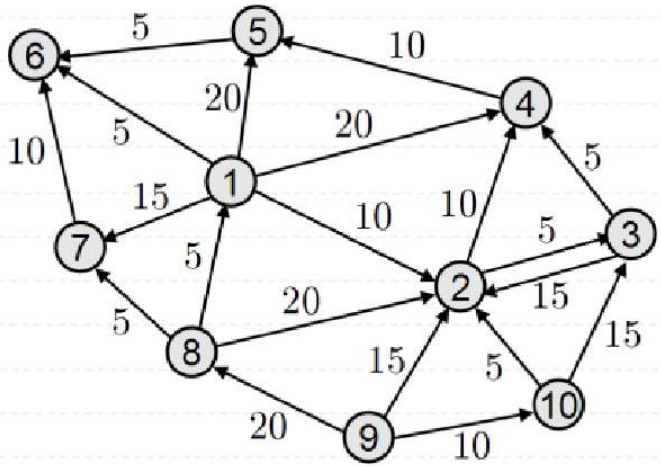
Listas, filas, pilhas, árvores e grafos

Uma **árvore** é uma **estrutura de dados hierárquica**, composta por **nós (nodes)** ligados entre si por **arestas (edges)**. Diferente das listas ou filas, que são **lineares**, as árvores formam uma **estrutura ramificada**, em que cada nó pode **gerar outros nós** — chamados **filhos (children)**.



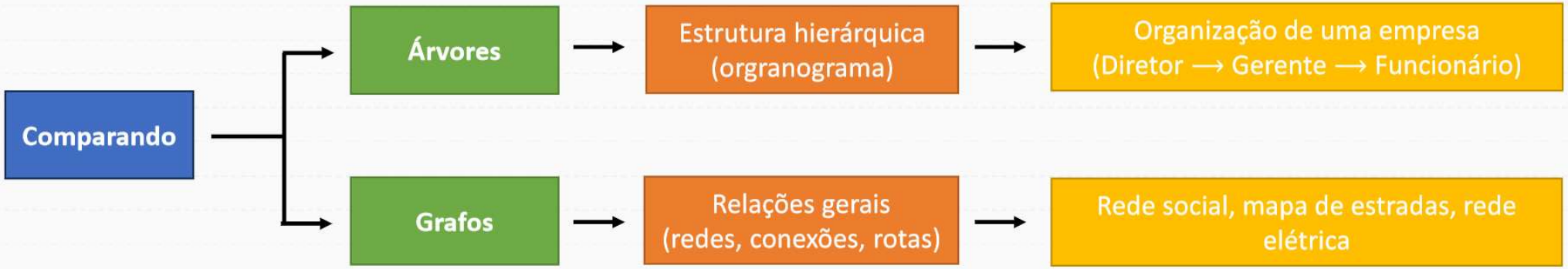
Listas, filas, pilhas, árvores e grafos

Um **grafo (graph)** é uma estrutura de dados **não linear** composta por **vértices (ou nós),** ou seja, os pontos que representam entidades; e **arestas (ou ligações)**, que caracterizam as conexões entre esses vértices.



Cada círculo numerado (1 a 10) representa um vértice. Cada seta indica uma aresta direcionada, ou seja, um caminho com sentido de um vértice para outro. O número sobre cada seta representa o peso da aresta, que pode ser uma distância, ou uma capacidade, custo de fluxo, etc., dependendo do contexto do problema.

Conceitualmente, quais são as principais diferenças entre **árvores** e **grafos**?



Listas, filas, pilhas, árvores e grafos

Em listas, filas, pilhas (e até em outras estruturas mais complexas), os elementos podem se vincular a partir de estratégias diferentes, das quais as duas mais importantes envolvem definição de **arrays** (conjunto de variáveis do mesmo tipo e acessadas a partir de índices) ou **ponteiros** (um elemento está vinculado com os demais a partir de endereços de memórias).

Eventualmente, tais elementos são "**encapsulados**" em structs, para melhor construir TADs complexos (vamos detalhar cada uma dessas implementações a seguir).

Como exemplo, vamos implementar o TAD Tarefa, implementado usando uma lista (genérica).

Um exemplo de lista ...

Os principais atributos a serem considerados serão extraídos da imagem a seguir, que mostra quais as principais informações que serão guardadas na base de dados.

De forma resumida, temos: Título (da tarefa), Responsável, Status, Progresso e Avaliação, que serão tratadas como *string*, *string*, *int*, *float* e *int*, respectivamente. Essas informações nos permitem estabelecer a structs para armazenar cada tarefa do nosso planner.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Configurações Gerais (que serão usadas para armazenar tamanhos máximos de strings.

#define TAM_TITULO 96
#define TAM_NOME   48

// /Definições de Status /

#define A_FAZER        0
#define EM_ANDAMENTO   1
#define FEITO          2
```

```c
/* --- Estrutura de uma Tarefa --- */

typedef struct {
    char Titulo[TAM_TITULO];
    char Responsavel[TAM_NOME];
    int Status;      /* A_FAZER, EM_ANDAMENTO ou FEITO */
    float Progresso; /* 0.0 a 1.0 */
    int Avaliacao;   /* 0 a 5 */
} Tarefa;

/* --- Estrutura da Lista de Tarefas --- */

typedef struct {
    Tarefa *Dados;
    int Tamanho;
    int Capacidade;
} ListaTarefas;
```

```c
/* --- Cabeçalhos das Funções --- */

ListaTarefas *CriarLista(int CapacidadeInicial);
void DestruirLista(ListaTarefas *Lista);

int AdicionarTarefa(ListaTarefas *Lista, Tarefa *T);
int RemoverTarefa(ListaTarefas *Lista, int Indice);
Tarefa *ObterTarefa(ListaTarefas *Lista, int Indice);
int AtualizarStatus(ListaTarefas *Lista, int Indice, int NovoStatus);
int AtualizarProgresso(ListaTarefas *Lista, int Indice, float NovoProgresso);
int AtualizarAvaliacao(ListaTarefas *Lista, int Indice, int NovaAvaliacao);
int ContarPorStatus(const ListaTarefas *Lista, int Status);
float MediaAvaliacao(const ListaTarefas *Lista);
char *TextoStatus(int Status);
void ImprimirLista(const ListaTarefas *Lista);

/* --- E poderiam ser criadas várias outras funções (...)  --- */
```

As funções **CriarLista** e **DestruirLista** serão utilizadas, de forma similar, para todos os tipos abstratos de dados que estudaremos a partir de agora (**filas**, **pilhas** e **árvores**).

```
ListaTarefas *CriarLista(int CapacidadeInicial) {

    ListaTarefas *L = (ListaTarefas *) malloc(sizeof(ListaTarefas));
    if (L == NULL) return NULL;
    L->Capacidade = CapacidadeInicial;
    L->Tamanho = 0;
    L->Dados = (Tarefa *) malloc (CapacidadeInicial * sizeof(Tarefa));

    if (L->Dados == NULL) {
        free(L);
        return NULL;
    }
    return L;
}
```

Mostre, na função main, como utilizar esta função para criar uma **ListaTarefas** *com 20 elementos.*

A função **DestruirLista** tem por objetivo liberar a memória reservada para a lista. Na implementação, usaremos de dois **free**'s: um para o vetor de dados e outro para a estrutura da lista.

```
void DestruirLista(ListaTarefas *Lista) {

    if (Lista == NULL) return;
    free(Lista->Dados);
    free(Lista);

}
```

Mostre, na função main, como utilizar esta função para liberar a memória da lista então criada. A seguir, tente acessar os elementos da lista então destruída.

A função **AdicionarTarefa** tem por finalidade a inserção de uma tarefa na lista de tarefas passada como parâmetro. No caso, a inserção consiste em copiar a tarefa T em uma posição específica do vetor.

```
int AdicionarTarefa(ListaTarefas *Lista, Tarefa *T) {
    if (Lista == NULL || T == NULL) return 0;
    if (!AumentarCapacidade(Lista, Lista->Tamanho + 1)) return 0;
    Lista->Dados[Lista->Tamanho] = *T;
    Lista->Tamanho++;
    return 1;
}
```

Mostre, na função main, como utilizar esta função para inserir três ou quatro tarefas em uma lista de tarefas criadas anteriormente.

Pergunta:

Como verificar que a inserção se deu de forma bem sucedida?

A função **RemoverTarefa** tem por objetivo excluir uma tarefa da lista. Como estratégia, será excluída a tarefa cujo índice foi passado como parâmetro.

```
int RemoverTarefa(ListaTarefas *Lista, int Indice) {

    if (Lista == NULL) return 0;
    if (Indice < 0) return 0;
    if (Indice >= Lista->Tamanho) return 0;

    /* Desloca manualmente todos os elementos após o índice para a esquerda */

    for (int i = Indice; i < Lista->Tamanho - 1; i++)Lista->Dados[i] = Lista->Dados[i + 1];
    Lista->Tamanho--;

    return 1;  /* Sucesso */
}
```

Mostre, na função main, como utilizar esta função para excluir a segunda tarefa da lista (ou seja, cuja posição é igual a 1). A seguir, verifique se a exclusão se deu de forma bem sucedida.

A função **AtualizarStatus** tem por finalidade modificar o atributo “Status” da tarefa cujo índice é para NovoStatus, que também foi passado como parâmetro.

```
int AtualizarStatus(ListaTarefas *Lista, int Indice, int NovoStatus) {
    Tarefa *T = ObterTarefa(Lista, Indice);
    if (T == NULL) return 0;
    T->Status = NovoStatus;
    return 1;
}
```

A seguir, usando essa dinâmica, implemente as seguintes funções:

```
int AtualizarProgresso(ListaTarefas *Lista, int Indice, float NovoProgresso);
```

```
int AtualizarAvaliacao(ListaTarefas *Lista, int Indice, int NovaAvaliacao);
```

```
int ContarPorStatus(ListaTarefas *Lista, int Status);
```

```
float MediaAvaliacao(const ListaTarefas *Lista);
```

```c
int exibirTarefas(const ListaTarefas *Lista, char *CaminhoArquivo) {]

    if (Lista == NULL || CaminhoArquivo == NULL) return 0;
    FILE *Arq = fopen(CaminhoArquivo, "w");
    if (Arq == NULL) return 0;

    for (int i = 0; i < Lista->Tamanho; i++) {

        Tarefa T = Lista->Dados[i];
        printf("%d\t", i);
        printf("%s\t", T.Titulo);
        printf("%s\t", T.Responsavel);
        printf("%d\t", T.Status);
        printf("%f\t", T.Progesso);
        printf("%d \t\n ", T.Avaliacao);

    }
    return 1;

}
```

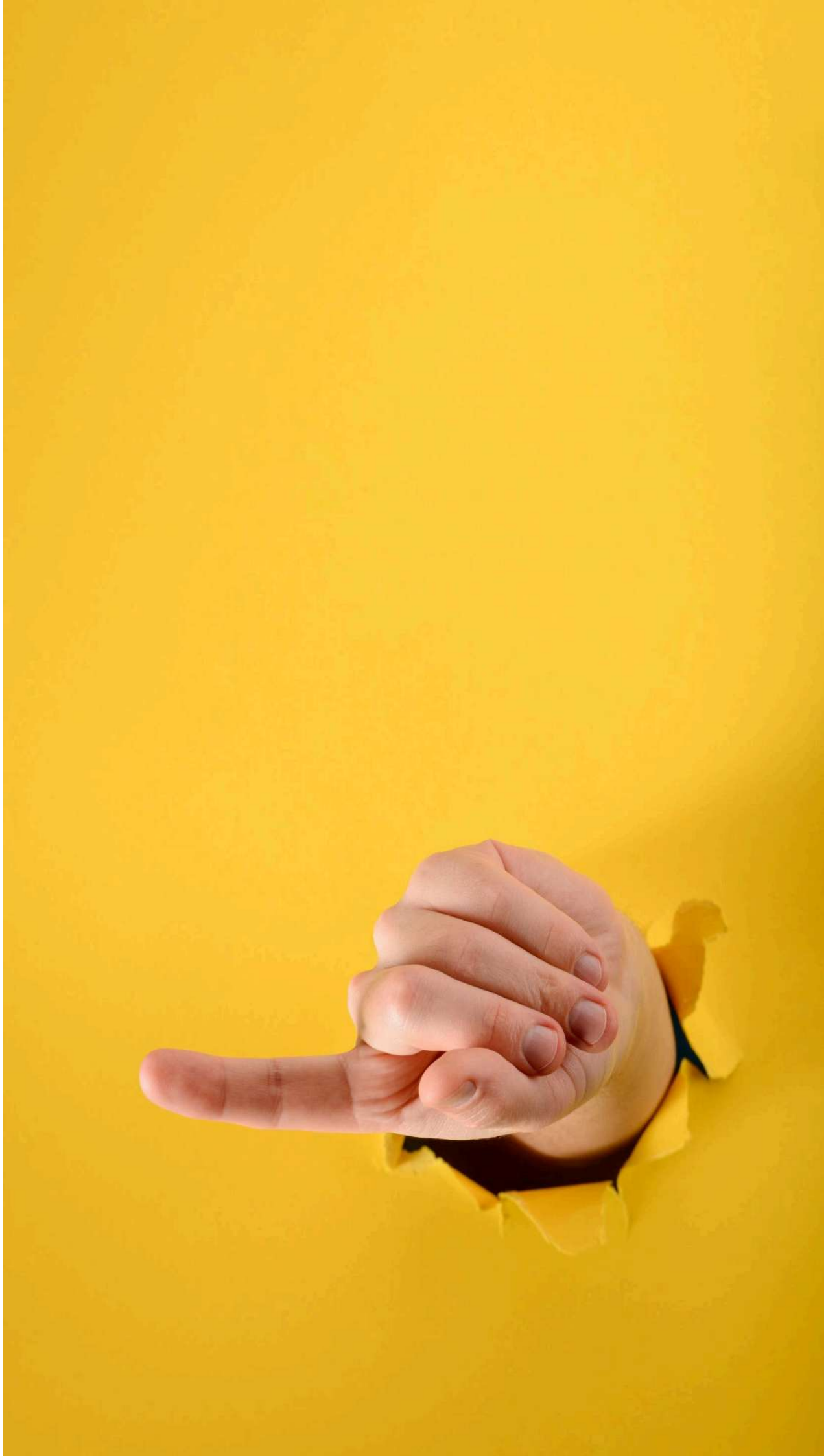
Melhorando a apresentação

Uma nova versão de “print”

O **HTML** e o **CSS** podem ser embutidos no código em C (ou em qualquer outra linguagem de programação) para apresentar os dados do vetor de tarefas na forma de tabela, como visto a seguir:

Lista de Tarefas

#	Título	Responsável	Status	Progresso	Avaliação
00	Finalizar materiais de lançamento	Ana	Feito		★★☆☆☆
01	Refinar objetivos	Bruno	Em Andamento		★★★☆☆
02	Identificar recursos principais	Carla	Feito		★★★★☆
03	Ùltima tarefa	Marcelo	Feito		★★★★☆



Q01 Escreva uma função que inverta uma lista, ou seja, a ordem dos elementos deve ser revertida.