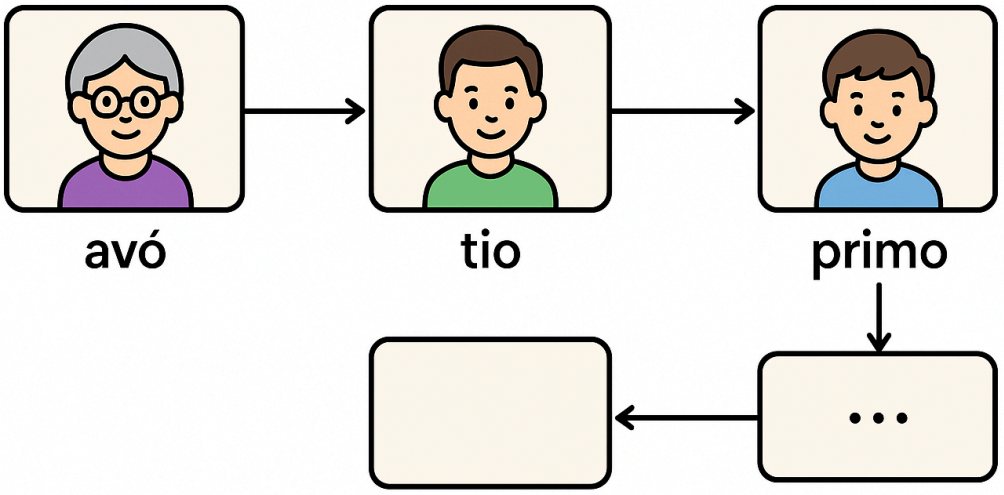
Olá, pessoal! Preparei este documento para que vocês revisem o conteúdo do semestre passado. Ao longo da apostila, haverá atividades para vocês praticarem, e é importante que realizem todas. Vamos lá?

Lista Simples ou Encadeada



Vamos imaginar a seguinte situação: você vai visitar todos os seus parentes, mas só conhece o endereço da casa da sua avó.

Quando usamos **arrays**, é como se todos os seus parentes moravam lado a lado, em uma mesma rua, de forma **sequencial**. Isso facilita encontrá-los, mas na vida real dificilmente todos moram um ao lado do outro, não é mesmo?

Agora pense que a **cidade representa a memória do computador**. Nela, existem diversos terrenos espalhados onde é possível “construir casas” que, para nós, significam **alocar dados**.

No caso da **lista encadeada**, funciona assim:

- Você começa visitando a avó (o **primeiro nó**, também chamado de *head*).
- Sua avó conhece o endereço da casa do tio e indica para você.
- O tio, por sua vez, conhece o endereço de outro parente e assim por diante.

Cada parente (ou seja, cada **nó da lista**) possui duas partes:

1. Suas próprias informações (por exemplo: nome, idade e CPF).
2. O endereço do próximo parente (o **ponteiro para o próximo nó**).

É importante notar que, em uma **lista simples**, cada parente só sabe quem vem depois. Não há como voltar para a casa da avó ou de um parente anterior, pois não existe referência para trás.

Assim, a lista encadeada é como uma corrente de endereços que conecta elementos espalhados pela memória, permitindo percorrê-los na ordem definida pelos “ponteiros” de cada nó.

Você me pergunta: professor, eu entendi, mas como faço isso no código?

Todas as pessoas são parentes e, para nós, no código em linguagem C, cada uma será representada por uma `struct`.

Nessa `struct`, colocamos as informações do parente e, por fim, definimos um ponteiro que indicará o próximo. Então, vamos lá:

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

// Definição da struct Pessoa

typedef struct Pessoa {
    char nome[50];              // Informação: nome do parente
    int idade;                  // Informação: idade do parente
    struct Pessoa *proximo;     // Ponteiro para o próximo parente
} Pessoa;
```

Agora já temos a `struct`, e com ela é possível fazer a ligação com o próximo nó. Dessa forma, conseguimos construir nossa lista. Vamos criar cada parente, e o ponteiro que indica o próximo receberá NULL, já que é preciso ter o próximo parente criado para poder apontar para ele.

```c
// Função para criar uma nova pessoa (nó)
Pessoa* criarPessoa(const char *nome, int idade) {
    Pessoa *nova = (Pessoa*) malloc(sizeof(Pessoa)); // Alocamos espaço na memória
    strcpy(nova->nome, nome);                        // Guardamos o nome
    nova->idade = idade;                             // Guardamos a idade
    nova->proximo = NULL;                            // No início não aponta para ninguém
    return nova;
}
```

Assim, conseguimos criar a corrente de endereços dos parentes. Usando nossa analogia, na cidade existem terrenos e construímos a casa de cada parente, mas ainda não indicamos para quem cada um irá apontar. No momento em que criamos dois parentes, já conseguimos fazer o apontamento.

```c
int main() {
    // Criando os parentes (nós da lista)
    Pessoa *avo = criarPessoa("Vovó", 75);
    Pessoa *tio = criarPessoa("Tio João", 50);
    Pessoa *primo = criarPessoa("Primo Carlos", 20);

    // Ligando os nós (definindo a corrente de endereços)
    avo->proximo = tio;
    tio->proximo = primo;

    return 0;
}
```

Agora vamos exibir a corrente de endereços, criando uma função que recebe o primeiro nó como parâmetro. Criamos um nó auxiliar para fazer o loop, ou seja, percorre até que o último parente não tenha um endereço de outro parente.

```c
void mostrarListaFrente(Pessoa *inicio) {
    Pessoa *atual = inicio;
    printf("Percorrendo do início ao fim:\n");
    while (atual != NULL) {
        printf("Nome: %s, Idade: %d\n", atual->nome, atual->idade);
        if (atual->proximo == NULL) break; // Guardar último nó
        atual = atual->proximo;
    }
}
```

Agora vamos utilizar a função na main para mostrar o encadeamento de parentes.

```c
int main() {
    // Criando os parentes (nós da lista)
    Pessoa *avo = criarPessoa("Vovó", 75);
    Pessoa *tio = criarPessoa("Tio João", 50);
    Pessoa *primo = criarPessoa("Primo Carlos", 20);

    // Ligando os nós (definindo a corrente de endereços)
    avo->proximo = tio;
    tio->proximo = primo;

    // Agora, mostramos a lista encadeada
    printf("Lista de parentes:\n");
    mostrarLista(avo);

    return 0;
}
```

Agora é com você!

Eu gostaria de ir de férias para assistir alguns filmes e preciso da sua ajuda. Me indique uma sequência de filmes para assistir utilizando uma lista encadeada. Você deve criar uma *struct* para o filme e, dentro dela, um ponteiro para indicar o próximo filme que devo assistir. Também é necessário criar funções para criar cada filme, exibir a lista de filmes e remover um filme da lista, caso o professor já tenha assistido.

Lista Duplamente Encadeada

Pense em uma **conversa no WhatsApp**.

Você pode deslizar a tela para **baixo** e ver as mensagens mais recentes, mas também pode deslizar para **cima** e voltar nas mensagens antigas.

Na lista simplesmente encadeada, era como se cada mensagem só conhecesse a que veio depois. Agora, na **lista duplamente encadeada**, cada mensagem sabe **quem veio antes e quem vem depois**.

Isso nos permite percorrer a lista tanto para frente quanto para trás.

Criando a estrutura:

No código em C, cada mensagem será um nó da lista.
Ela precisa guardar:

1. O **texto da mensagem**.
2. Um ponteiro para a **próxima mensagem**.
3. Um ponteiro para a **mensagem anterior**.

```c
typedef struct Mensagem {
    char texto[100];
    struct Mensagem *prox; // Próxima mensagem
    struct Mensagem *ant;  // Mensagem anterior
} Mensagem;
```

Criando uma nova mensagem:

Sempre que alguém manda algo no WhatsApp, precisamos criar essa mensagem na lista. Ela começa sem saber quem está antes ou depois, por isso inicializamos os ponteiros com `NULL`.

```c
Mensagem* criarMensagem(char* texto) {
    Mensagem* nova = (Mensagem*) malloc(sizeof(Mensagem));
    strcpy(nova->texto, texto);
    nova->prox = NULL;
    nova->ant = NULL;
    return nova;
}
```

Inserindo no início da conversa

No WhatsApp, a mensagem mais recente aparece primeiro.
Para inserir uma mensagem no **início da lista**, fazemos:

1. O novo nó aponta para a mensagem que antes era a primeira.

2. Essa antiga primeira mensagem passa a reconhecer que existe alguém antes dela.

3. Atualizamos o início da lista.

```c
void inserirInicio(Mensagem** inicio, char* texto) {
    Mensagem* nova = criarMensagem(texto);
    if (*inicio != NULL) {
        nova->prox = *inicio;     // Nova aponta para a antiga primeira
        (*inicio)->ant = nova;    // Antiga primeira aponta de volta para a nova
    }
    *inicio = nova; // Atualizamos o início da conversa
}
```

Exibindo as mensagens para frente

Aqui, simulamos deslizar a tela **para baixo** no WhatsApp, lendo as mensagens mais recentes até a última.

```c
void exibirMensagens(Mensagem* inicio) {
    Mensagem* aux = inicio;
    while (aux != NULL) {
        printf("Mensagem: %s\n", aux->texto);
        aux = aux->prox;
    }
}
```

Exibindo as mensagens para trás

Agora simulamos deslizar a tela **para cima**, lendo as mensagens mais antigas até voltar na primeira.

```c
void exibirMensagensReverso(Mensagem* inicio) {
    Mensagem* aux = inicio;

    // Primeiro vamos até a última mensagem
    while (aux->prox != NULL) {
        aux = aux->prox;
    }

    // Agora voltamos para o início
    while (aux != NULL) {
        printf("Mensagem: %s\n", aux->texto);
        aux = aux->ant;
    }
}
```

Testando na `main`

```c
int main() {
    Mensagem* conversa = NULL;

    inserirInicio(&conversa, "Oi, tudo bem?");
    inserirInicio(&conversa, "Você viu a novidade?");
    inserirInicio(&conversa, "Sim, achei incrível!");

    printf(" -> Conversa do início ao fim:\n");
    exibirMensagens(conversa);

    printf("\n- <- Conversa do fim ao início:\n");
    exibirMensagensReverso(conversa);

    return 0;
}
```

Agora é com você!

Implemente uma **lista duplamente encadeada** para representar um **chat de grupo**.
Cada mensagem deve ter: **autor, conteúdo e horário de envio**. Crie funções para, inserir uma nova mensagem , exibir todas as mensagens do início ao fim. Exibir todas as mensagens do fim ao início.

Lista Circular

Imagine que você está ouvindo música no **Spotify** e ativa a opção **“repetir playlist”**. Quando chega à última música, automaticamente volta para a primeira, e assim sucessivamente.Isso é exatamente o que acontece em uma **lista circular**: o último nó não aponta para NULL, mas sim para o primeiro nó da lista. Assim, criamos um ciclo infinito de elementos.

Criando a estrutura

No código em C, cada música da playlist será representada por um nó com:

1. O **nome da música**.
2. O ponteiro para a **próxima música**.

```c
typedef struct Musica {
    char titulo[100];
    struct Musica *prox; // Próxima música
} Musica;
```

Criando uma nova música

Para adicionar uma nova faixa à playlist, criamos uma função que aloca memória, copia o título e, no início, aponta para ela mesma (pois ainda é a única música da lista).

```c
Musica* criarMusica(char* titulo) {
    Musica* nova = (Musica*) malloc(sizeof(Musica));
    strcpy(nova->titulo, titulo);
    nova->prox = nova; // Aponta para ela mesma
    return nova;
}
```

Inserindo músicas na playlist

Quando já existe uma playlist, precisamos inserir a nova música **antes da primeira**, de modo que o encadeamento circular continue funcionando.

```c
void inserirMusica(Musica** inicio, char* titulo) {
    Musica* nova = criarMusica(titulo);

    if (*inicio == NULL) {
        *inicio = nova; // Primeira música criada
    } else {
        Musica* aux = *inicio;

        // Percorre até a última música (a que aponta para o início)
        while (aux->prox != *inicio) {
            aux = aux->prox;
        }

        aux->prox = nova;   // Última aponta para a nova
        nova->prox = *inicio; // Nova aponta para a primeira
    }
```

Exibindo as músicas da playlist

Agora vamos “tocar” todas as músicas da playlist. Repare que precisamos usar um **laço especial**, pois não existe mais NULL para indicar o fim.

```c
void exibirPlaylist(Musica* inicio) {
    if (inicio == NULL) return;

    Musica* aux = inicio;
    do {
        printf("Música: %s\n", aux->titulo);
        aux = aux->prox;
    } while (aux != inicio);
}

void exibirPlaylist(Musica* inicio) {
    if (inicio == NULL) return;

    Musica* aux = inicio;
    do {
        printf("Música: %s\n", aux->titulo);
        aux = aux->prox;
    } while (aux != inicio);
}
```

Testando na `main`

```c
int main() {
    Musica* playlist = NULL;

    inserirMusica(&playlist, "Imagine - John Lennon");
    inserirMusica(&playlist, "Bohemian Rhapsody - Queen");
    inserirMusica(&playlist, "Hotel California - Eagles");

    printf("Playlist em loop:\n");
    exibirPlaylist(playlist);

    return 0;
}
```

Agora é com você!

Implemente uma **playlist circular** no Spotify. Cada música deve conter: **título, artista e duração**. Crie funções para inserir uma nova música, exibir todas as músicas da playlist em ordem e criar uma função que simule o **"pular faixa"**, mostrando qual é a próxima música a ser tocada.

Pilha

Vamos imaginar que você está usando o **navegador de internet**. Cada vez que você abre um site, ele vai para o **topo da pilha de abas**. Se você abrir o Google, depois o YouTube e, por fim, o site da OpenAI, a ordem da pilha será:

Topo → OpenAI → YouTube → Google → Base

Quando você fechar uma aba, sempre será fechada a do topo, ou seja, a última que você abriu. Esse é exatamente o funcionamento de uma **Pilha**: o último elemento a entrar é o primeiro a sair (**LIFO — Last In, First Out**).

Criando o nó da pilha

Cada site acessado será representado por um **nó** (No). Ele precisa armazenar o **endereço do site** e também um ponteiro para o **próximo nó** (o próximo site na pilha).

```c
typedef struct No {
    char endereco[100];
    struct No *proximo;
} No;
```

Criando a pilha

A pilha em si será representada por uma struct que guarda apenas um ponteiro para o **topo** da pilha.
Quando ela é criada, começa vazia (topo = NULL).

```c
typedef struct Pilha {
    No *topo;
} Pilha;

Pilha* criarPilha() {
    Pilha* pilha = (Pilha*)malloc(sizeof(Pilha));
    pilha->topo = NULL;
    return pilha;
}
```

Função push (empilhar)

Agora precisamos de uma forma de **abrir um novo site** no navegador.
Isso significa **inserir um novo nó no topo da pilha**:

1. Criamos um novo nó.
2. Copiamos o endereço do site.
3. Fazemos o ponteiro dele apontar para o antigo topo.
4. Atualizamos o topo da pilha.

```c
void push(Pilha* pilha, const char* endereco) {
    No* novo = (No*) malloc(sizeof(No));
    strcpy(novo->endereco, endereco);
    novo->proximo = pilha->topo;
    pilha->topo = novo;
}
```

Função exibir

Podemos visualizar todos os sites que estão abertos no navegador. A exibição começa sempre pelo **topo** da pilha, descendo até a base.

```c
void exibir(Pilha* pilha) {
    if (pilha->topo == NULL) {
        printf("Pilha vazia!\n");
        return;
    }

    printf("Sites na pilha (do topo para a base):\n");
    No* aux = pilha->topo;
    while (aux != NULL) {
        printf("%s -> ", aux->endereco);
        aux = aux->proximo;
    }
    printf("NULL\n");
}
```

Função pop (desempilhar)

Quando você **fecha uma aba**, o site do topo da pilha é removido.
Para isso:

1. Guardamos o nó do topo.
2. Atualizamos o topo para o próximo nó.
3. Liberamos a memória do site removido.

```c
void pop(Pilha* pilha) {
    if (pilha->topo == NULL) return;

    No* removido = pilha->topo;
    pilha->topo = removido->proximo;
    free(removido);
}
```

Testando na main

Agora vamos abrir alguns sites e exibir a pilha:

```c
int main() {
    Pilha* pilha = criarPilha();

    push(pilha, "https://www.google.com");
    push(pilha, "https://www.youtube.com");
    push(pilha, "https://www.openai.com");

    exibir(pilha);

    return 0;
}
```

Agora é com você!

Implemente uma **pilha de histórico de navegação**. Cada página deve conter: **endereço, título da aba e horário de acesso**.Crie funções para, abrir uma nova página (push), fechar a página atual (pop) e exibir todas as páginas abertas (exibir).

Fila

Imagine que você está em uma impressora compartilhada em um escritório. Várias pessoas enviam documentos para imprimir, mas a impressora só consegue imprimir um de cada vez. A ordem é simples: quem chegou primeiro, imprime primeiro. Esse é o funcionamento da Fila: o primeiro a entrar é o primeiro a sair (FIFO — First In, First Out).

Criando o nó da fila

Cada documento enviado para impressão será representado por um nó (No).
Ele precisa guardar:

1. O nome do arquivo.

2. Um ponteiro para o próximo arquivo na fila.

```c
typedef struct No {
    char arquivo[100];
    struct No *proximo;
} No;
```

Criando a fila

A fila em si precisa conhecer dois pontos importantes:

1. O início (onde está o primeiro documento a ser impresso).
2. O fim (onde novos documentos serão adicionados).

```c
typedef struct Fila {
    No *inicio;
    No *fim;
} Fila;

Fila* criarFila() {
    Fila* fila = (Fila*) malloc(sizeof(Fila));
    fila->inicio = NULL;
    fila->fim = NULL;
    return fila;
}
```

Função enqueue (enfileirar)

Quando um novo documento é enviado para impressão, ele entra no final da fila.

1. Criamos um novo nó.
2. Se a fila estiver vazia, tanto o início quanto o fim apontam para ele.
3. Caso contrário, o último da fila passa a apontar para o novo documento, e o fim é atualizado.

```c
void enqueue(Fila* fila, const char* arquivo) {
    No* novo = (No*) malloc(sizeof(No));
    strcpy(novo->arquivo, arquivo);
    novo->proximo = NULL;

    if (fila->fim == NULL) {
        fila->inicio = novo;
        fila->fim = novo;
    } else {
        fila->fim->proximo = novo;
        fila->fim = novo;
    }
}
```

Função dequeue (desenfileirar)

Quando a impressora pega o próximo documento, removemos o primeiro da fila.

1. Guardamos o nó do início.
2. Atualizamos o início para o próximo.
3. Se a fila ficar vazia, o fim também vira NULL.
4. Liberamos a memória do nó removido.

```c
void dequeue(Fila* fila) {
    if (fila->inicio == NULL) {
        printf("Fila vazia!\n");
        return;
    }

    No* removido = fila->inicio;
    printf("Imprimindo: %s\n", removido->arquivo);

    fila->inicio = removido->proximo;

    if (fila->inicio == NULL) {
        fila->fim = NULL;
    }

    free(removido);
}
```

Função exibir

Podemos listar todos os documentos que estão esperando para impressão:

```c
void exibir(Fila* fila) {
    if (fila->inicio == NULL) {
        printf("Fila vazia!\n");
        return;
    }

    printf("Fila de impressão:\n");
    No* aux = fila->inicio;
    while (aux != NULL) {
        printf("%s -> ", aux->arquivo);
        aux = aux->proximo;
    }
    printf("NULL\n");
}
```

Testando na main

```c
int main() {
    Fila* fila = criarFila();

    enqueue(fila, "Trabalho.docx");
    enqueue(fila, "Apresentacao.pptx");
    enqueue(fila, "Relatorio.pdf");

    exibir(fila);

    dequeue(fila); // Imprime o primeiro
    exibir(fila);

    return 0;
}
```

Saída esperada

```
Fila de impressão:
Trabalho.docx -> Apresentacao.pptx -> Relatorio.pdf -> NULL

Imprimindo: Trabalho.docx

Fila de impressão:
Apresentacao.pptx -> Relatorio.pdf -> NULL
```

Agora é com você!

Implemente uma fila de uploads em nuvem. Cada arquivo deve conter: nome do arquivo, tamanho em MB e status (em espera, enviando, concluído).

Crie funções para adicionar um arquivo na fila (enqueue). retirar o próximo arquivo da fila (dequeue), exibir todos os arquivos em espera (exibir) e mostrar qual é o próximo arquivo que será enviado (peek).