# 



[**Introdução 3**](#_k0i2f4z3fk4m)

[**Estruturas de Dados Utilizadas 4**](#_v0m61isbupvr)

[**Funções Implementadas 5**](#_mpw491gefx5w)

[NewNode() 5](#_theak6vz9pej)

[NewStack() 5](#_ghmwtv32husb)

[Push(stack, processo) 5](#_pd1541ognte4)

[Pop(stack) 5](#_axabb33t49kv)

[EmptyStack(stack) 5](#_flfv0y5enkk8)

[FreeStack(stack) 5](#_hvwvflompx1q)

[3.7 isEmpty(stack) 5](#_l26us1u65lye)

[3.8 PrintStack(stack) 5](#_jr3dpz44i5ec)

[wait\_seconds(seconds) 6](#_ri2sgyrsabgr)

[**Execução do Programa 6**](#_36bh0vpl1g5b)

[Saída esperada: 7](#_lcv72cho2na9)

[**Conclusão 8**](#_wspx22d93r1d)

[**Referências 9**](#_fn9zctv3uds7)

## 

## 

## 

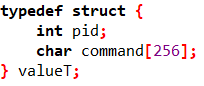
## **Introdução**

Este trabalho tem como objetivo a **implementação prática da estrutura de dados Stack (Pilha)** utilizando uma **lista ligada simples**, aplicada com base nos fundamentos e boas práticas estudados.A Stack é uma estrutura do tipo **LIFO (Last-In, First-Out)**, em que o último elemento inserido é o primeiro a ser removido.

Ao contrário de arrays, a Stack não permite acesso aleatório aos seus elementos, sendo possível apenas interagir com o **elemento no topo** por meio das operações **push** (inserção) e **pop** (remoção).

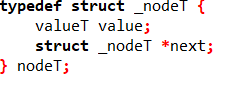
A implementação utiliza **alocação dinâmica de memória** e aplica boas práticas na manipulação de estruturas e ponteiros em C.

## **Estruturas de Dados Utilizadas**



Representa um **processo do sistema**, com:

* **pid**: identificador do processo (inteiro).
* **command**: nome do comando ou programa que gerou o processo.



Cada **nó da lista ligada** contém:

* Um campo **value** do tipo **valueT**.
* Um ponteiro **next** para o próximo nó da pilha.



A Stack contém um único campo:

* **head**: ponteiro para o **topo da pilha**.

## **Funções Implementadas**

### **NewNode()**

Criar dinamicamente um novo nó (**nodeT**) e retorna o seu ponteiro. Em caso de erro de alocação, imprime mensagem e retorna **NULL**.

### **NewStack()**

Cria e inicia uma nova Stack (**stackT**) com **head** igual a **NULL**.

### **Push(stack, processo)**

Adiciona um novo processo no topo da Stack. O novo nó aponta para o topo anterior.

### **Pop(stack)**

Remove o processo no topo da Stack e o retorna. Se a Stack estiver vazia, imprime mensagem de erro e retorna um processo vazio.

### **EmptyStack(stack)**

Remove todos os elementos da Stack, liberando a memória dos nós. A Stack permanece alocada, mas fica vazia (**head = NULL**).

### **FreeStack(stack)**

Libera completamente a estrutura **stackT**, **apenas se estiver vazia**. Caso contrário, imprime erros.

### **3.7 isEmpty(stack)**

Retorna **1** se a Stack estiver vazia ou não inicializada, **0** caso contrário.

### **3.8 PrintStack(stack)**

Imprime os elementos da Stack do topo até a base. Mostra mensagem se a Stack estiver vazia.

### **wait\_seconds(seconds)**

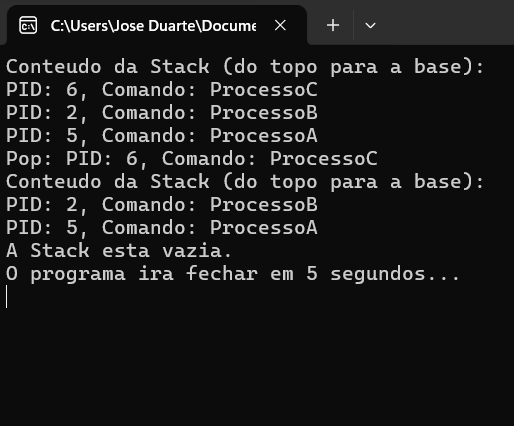
Implementa uma pausa no programa por um determinado número de segundos utilizando **clock()**.

## **Execução do Programa**

O **main()** simula o uso real da Stack:

1. Cria a Stack.
2. Cria três processos com identificadores e nomes diferentes.
3. Insere os processos com **Push**.
4. Exibe Stack com **PrintStack**.
5. Remove um processo com **Pop** e imprime o removido.
6. Esvazia completamente a Stack com **EmptyStack**.
7. Tenta liberar a Stack com **FreeStack**.
8. Aguarda 5 segundos antes de encerrar.

### **Saída esperada:**



## **Conclusão**

A implementação de uma **estrutura de dados Stack utilizando lista ligada** demonstrou ser uma solução eficiente e versátil para manipulação dinâmica de dados, especialmente em contextos onde o padrão de acesso do tipo **LIFO (Last-In, First-Out)** é necessário. A abordagem com **lista ligada** proporciona uma flexibilidade significativa em tempo de execução, eliminando a necessidade de definir previamente um tamanho fixo para a pilha, como ocorre em estruturas baseadas em arrays. Isso permite que a Stack cresça ou seja esvaziada conforme a demanda da aplicação, otimizando o uso da memória e reduzindo o risco de sobrecarga ou subutilização de recursos.

Um dos pontos centrais desta implementação foi a **gestão cuidadosa da memória dinâmica**, assegurando que cada elemento alocado fosse devidamente liberado após o uso. As funções **EmptyStack** e **FreeStack** foram desenvolvidas com atenção especial à prevenção de **memory leaks**, assegurando que todos os nós da pilha fossem deslocados corretamente, sem afetar a estrutura principal. Essa preocupação reflete boas práticas de engenharia de software e programação em baixo nível, fundamentais em linguagens como C.

Além de ser uma base sólida para fins didáticos, a Stack implementada se mostra altamente aplicável a **situações reais** de desenvolvimento, como:

* **simulações de gerenciamento de processos** em sistemas operativos,
* **controle de chamadas de função** em linguagens de programação,
* **navegação reversa** (undo/redo) em interfaces gráficas,
* **análise e avaliação de expressões matemáticas e lógicas**, entre outras.

Em suma, esta implementação não apenas cumpre os requisitos técnicos propostos, como também representa um exercício completo de domínio da linguagem C, com atenção especial à **eficiência, segurança e clareza estrutural**.

## **Referências**

<https://www.geeksforgeeks.org/computer-organization-architecture/stack-frame-in-computer-organization/>

<https://en.cppreference.com/w/c/io.html>

<https://en.cppreference.com/w/c/memory.html>

<https://www.geeksforgeeks.org/dsa/introduction-to-stack-data-structure-and-algorithm-tutorials/>

<https://www.geeksforgeeks.org/dsa/introduction-to-stack-data-structure-and-algorithm-tutorials/https:/www.tutorialspoint.com/cprogramming/index.htm/>