Introdução às Pilhas

A ideia de "pilhas" surgiu inicialmente na década de 1940-1950 na concepção de **máquinas de execução sequencial**.

- **Alan Turing** já sugeria a necessidade de armazenamento de estado.
- A primeira implementação explícita de pilha em hardware foi na **Burroughs B5000** — um dos primeiros computadores baseados em stack machine.
- Em linguagens de programação, **Forth** (1970) e **PostScript** (1982) usam pilhas como mecanismo central.

Filosoficamente, pilhas representam o conceito de **memória temporária**, uma ideia associada ao raciocínio humano de curta duração: fazemos "empilhamento" de ideias e desfazemos conforme resolvemos.

Em ciência da computação, uma **pilha** (do inglês, *stack*) é uma **estrutura de dados abstrata** (ADT — Abstract Data Type) que **organiza elementos** segundo o princípio **LIFO** — *Last In, First Out*, ou seja: **o último elemento inserido é o primeiro a ser removido**.

Pilhas aparecem naturalmente em problemas de controle de fluxo, execução de programas, recursão, processamento de expressões e mais.

Pilhas são interessantes pois:

- Promovem **localidade de referência**: último dado inserido é o primeiro a ser removido → isso é excelente para cache e eficiência de memória.
- Modelam naturalmente **fluxos hierárquicos**: ao empilhar chamadas de funções, é como se você caminhasse numa árvore de decisão de forma controlada.

Em execução real, esse comportamento gera **chamadas recursivas controladas** — como em algoritmos de busca em profundidade (**DFS** — *Depth First Search*), onde o próprio sistema simula uma pilha.

- Modelam o **comportamento temporal** das tarefas humanas e máquinas.
- Permitem a **gerência controlada** de memória de forma previsível.
- São base para **estruturas de dados mais complexas** (por exemplo, parsing tables, caches, controladores de execução).
- Sua simplicidade de implementação as torna **robustas** e fáceis de analisar.

Definição Formal

Uma pilha S é um conjunto ordenado de elementos sobre o qual são definidas **duas operações fundamentais**:

1. **push(x)**: insere um elemento x no topo da pilha.
2. **pop()**: remove o elemento do topo da pilha.

Além disso, existem operações auxiliares, como:

- **top()** ou **peek()**: retorna o elemento do topo **sem removê-lo**.
- **isEmpty()**: verifica se a pilha está vazia.
- **size()**: retorna o número de elementos na pilha.

Matematicamente, podemos modelar uma pilha como uma sequência:

$$
S = [s_1, s_2, \dots, s_n]
$$

onde s_n é o **topo** da pilha.

As operações então podem ser descritas:

- $push(S, x) = S \cup { x }$ com x sendo adicionado no final.
- $pop(S) = (S', x)$, onde S' é S sem o último elemento x.

Propriedades Fundamentais

- **Ordem:** Só é possível acessar o **último** elemento inserido.
- **Restrição de Acesso:** Não se pode "pular" elementos no meio.
- **Operações em Tempo Constante:** Normalmente, as operações push e pop são realizadas em $O(1)$.

Pilhas e Lógica Formal

Em lógica matemática, pilhas aparecem como **estruturas de memória** para **cálculos com escopo**, como:

- Avaliação de expressões aritméticas (*por exemplo, notação pós-fixada — RPN*).
- Análise sintática de linguagens formais (*gramáticas livres de contexto*).

Existem também **máquinas de pilha** (*stack machines*), onde a execução de programas é baseada unicamente em operações de pilha. Exemplo: a máquina da linguagem **Postfix**.

Formalmente, uma **máquina de pilha** é um tipo de **autômato** — mais precisamente, um **Autômato com Pilha** (*Pushdown Automaton* — PDA), usado para reconhecer linguagens livres de contexto:

$$
\text{PDA} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)
$$

onde:

- Q é o conjunto de estados.
- Σ é o alfabeto de entrada.
- Γ é o alfabeto da pilha.
- δ é a função de transição $Q \times (\Sigma \cup {\epsilon}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$.

- q_0 é o estado inicial.
- Z_0 é o símbolo inicial da pilha.
- F é o conjunto de estados finais.

Essa estrutura formal é usada para, por exemplo, **reconhecer sintaxe de linguagens de programação**.

Pilhas na Análise de Algoritmos

Eficiência: Pilhas são ótimas para armazenar dados temporários porque:

- Inserções e remoções são rápidas: $O(1)$.
- Não há necessidade de reorganizar elementos.

Espaço: Em algoritmos recursivos, a pilha de chamadas pode crescer até $O(d)$, onde d é a **profundidade da recursão**.

Aplicações Clássicas

1. **Avaliação de expressões:**
    - Expressões infixas (normais) podem ser transformadas em posfixas e avaliadas usando pilhas.
2. **Backtracking:**
    - Exploração de soluções (ex.: labirintos, sudoku).
    - Pilha armazena o caminho atual.
3. **Controle de execução de programas:**
    - Pilha de chamadas (call stack) em linguagens de programação.
    - Cada função chamada empilha um novo quadro de ativação (activation record).
4. **Desfazer operações (undo):**
    - Cada operação é empilhada; desfazer remove o último comando.
5. **Compiladores e interpretadores:**
    - Análise sintática via parsing top-down usa pilhas.

Aplicações Não Óbvias de Pilhas

Além dos usos clássicos, pilhas aparecem em problemas mais sofisticados:

1. Algoritmo de Tarjan (1972)

- Para encontrar componentes fortemente conexos em grafos direcionados.
- Pilha é usada para rastrear o caminho na DFS.

2. Sistema de Compilação e Parsing

- Algoritmos LR(1), LALR usam pilhas para armazenar estados de parsing enquanto leem tokens da entrada.

3. Algoritmos de Área Máxima de Histograma

- Problema clássico: calcular a maior área retangular em um histograma em tempo linear.
- Solução baseada em manter uma pilha de índices.

4. Redução de Espaço de Algoritmos Recursivos

- DFS recursivo → DFS iterativo usando pilha explícita para reduzir riscos de *stack overflow* real.

Exemplos Matemáticos

1. Pilha de Expressões Aritméticas

Exemplo de avaliação posfixa (notação polonesa reversa — RPN):

Expressão:
$
3\ 4\ 5\ *\ +
$

Processo com pilha:

- Push(3)
- Push(4)
- Push(5)
- Pop(5), Pop(4), calcula $4*5 = 20$, Push(20)
- Pop(20), Pop(3), calcula $3+20=23$

Resultado final: **23**.

---

2. Relação com Álgebra de Dados

Formalmente, a pilha pode ser representada por uma **ágebra universal** com as seguintes operações:

- $push: E \times S \rightarrow S$
- $pop: S \rightarrow S \times E$
- $top: S \rightarrow E$
- $empty: S \rightarrow \text{bool}$

onde E é o conjunto de elementos, e S é o conjunto de estados da pilha.

Estruturas Físicas de Implementação

Internamente, pilhas podem ser implementadas de várias formas:

- **Arranjos (Vetores):** Uso de um array e um índice para o topo.
- **Listas Ligadas:** Cada nó aponta para o próximo elemento da pilha.

Cada abordagem tem vantagens em espaço e flexibilidade.

Considerações Avançadas

- **Pilhas de Capacidade Fixa:** Quando implementadas com arrays, podem ter tamanho limitado — cuidado com *stack overflow*.
- **Pilha Dinâmica:** Uso de listas ligadas permite crescimento dinâmico.
- **Pilha de Pilhas:** Estruturas compostas onde cada elemento da pilha é outra pilha — usada em alguns algoritmos avançados.

Análise de Complexidade — Custo Amortizado

Além da análise clássica de complexidade por operação $O(1)$, existe um conceito importante: **análise amortizada**.

- Se o array da pilha dobra de tamanho quando cheio (resizing dinâmico), o custo do *resize* é $O(n)$.
- Porém, no longo prazo, a média por operação continua sendo $O(1)$.

Formalmente:

$$
\text{Custo amortizado} = \frac{\text{Custo total das operações}}{\text{Número de operações}}
$$

Este tipo de análise é fundamental em algoritmos sofisticados, como:

- Estruturas de dados dinâmicas (pilhas em vetores redimensionáveis).
- Estratégias de expansão de memória.

Variações de Pilhas

Existem várias **extensões** da ideia de pilha:

Estrutura	Descrição
Deque (*Double-Ended Queue*)	Permite inserções/remoções em ambos os extremos.
Pilhas múltiplas em um array	Dividem um vetor para várias pilhas (eficiência de espaço).
Pilhas persistentes	Cada modificação gera uma nova versão da pilha (funciona em ambientes imutáveis — importante para programação funcional).

Estrutura	Descrição
Pilhas limitadas	Pilhas com tamanho máximo fixo, usadas em controladores ou sistemas embarcados.
Pilhas de prioridade	Mesclam ideia de pilha com fila de prioridade — não é comum, mas útil em certos algoritmos de scheduling.

Relações Teóricas com Outras Estruturas

- **Fila (Queue):** contrário da pilha — FIFO (First In, First Out).
- **Deque:** generalização da pilha e fila.
- **Heap:** estrutura de árvore que pode ser vista como uma generalização da pilha (em termos de prioridades).

Teoria de Computação:

- Autômatos com Pilha (PDA) ≠ Autômatos Finitos (FA)
  - FA são mais fracos: não conseguem reconhecer linguagens com estruturas balanceadas (ex.: palíndromos, expressões com parênteses balanceados).
  - PDA usa a pilha para lembrar profundidade de chamadas ou agrupamentos.
- Linguagens livres de contexto (CFLs) ≠ linguagens regulares justamente por causa da capacidade de "memória" que a pilha dá.

Conceito	Descrição
Princípio	LIFO — Last In, First Out
Operações principais	push, pop, top, isEmpty
Complexidade	$O(1)$ em push e pop
Modelagem Matemática	Sequência ordenada e função de transição
Aplicações	Avaliação de expressões, execução de programas, parsing de linguagens, backtracking
Extensões	Deques, pilhas múltiplas, pilhas persistentes

Exemplo de Pilha em C — Com Explicação Detalhada

🛠️ Primeiro: Estrutura de Dados

Antes de tudo, precisamos representar a pilha.

Em C, usaremos **structs** para criar o conceito de "pilha".

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

#define MAX 100 // tamanho máximo da pilha

typedef struct {
    int itens[MAX]; // array para armazenar os elementos
    int topo;       // índice do elemento no topo
} Pilha;
```

📖 Explicação:

- **MAX** define o tamanho máximo da pilha.
- **itens[MAX]** é um array de inteiros onde vamos guardar os dados.
- **topo** é um número inteiro que aponta para o último elemento adicionado.
    - Se a pilha estiver vazia, **topo = -1**.

🚀 Segundo: Inicializar a Pilha

Precisamos de uma função para inicializar a pilha.

```c
void inicializar(Pilha *p) {
    p->topo = -1; // indica que a pilha está vazia
}
```

📖 Explicação:

- Quando inicializamos, o topo fica em (-1), porque ainda **não há elementos** na pilha.

🧹 Terceiro: Verificar se a Pilha Está Vazia ou Cheia

Essas funções ajudam a evitar erros.

```c
int estaVazia(Pilha *p) {
    return p->topo == -1;
}

int estaCheia(Pilha *p) {
    return p->topo == MAX - 1;
}
```

📖 Explicação:

- **Vazia**: Se topo == -1, então não tem elementos.
- **Cheia**: Se topo == MAX - 1, então o array está cheio.

📦 Quarto: Inserir um Elemento na Pilha (Push)

```
void push(Pilha *p, int valor) {
    if (estaCheia(p)) {
        printf("Erro: Pilha cheia!\n");
        return;
    }
    p->topo++; // move o topo para o próximo espaço
    p->itens[p->topo] = valor; // coloca o valor no topo
}
```

📖 Explicação:

- Antes de inserir, verificamos se a pilha está cheia.
- Se não estiver cheia, incrementamos o topo e colocamos o novo valor.

📤 Quinto: Remover um Elemento da Pilha (Pop)

```
int pop(Pilha *p) {
    if (estaVazia(p)) {
        printf("Erro: Pilha vazia!\n");
        exit(1); // encerra o programa em caso de erro
    }
    int valor = p->itens[p->topo]; // pega o valor do topo
    p->topo--; // remove o elemento (desce o topo)
    return valor;
}
```

📖 Explicação:

- Antes de remover, verificamos se a pilha está vazia.
- Retornamos o elemento no topo e depois **decrementamos o topo**.

🔍 Sexto: Ver o Elemento no Topo (Peek)

```
int peek(Pilha *p) {
    if (estaVazia(p)) {
        printf("Erro: Pilha vazia!\n");
        exit(1);
    }
```

```
    return p->itens[p->topo];
}
```

📖 Explicação:

- **peek** olha o valor no topo sem remover.
- É como perguntar: "Quem está no topo agora?"

🧪 Sétimo: Exemplo de Uso

Agora vamos usar tudo isso!

```
int main() {
    Pilha minhaPilha;
    inicializar(&minhaPilha);

    push(&minhaPilha, 10);
    push(&minhaPilha, 20);
    push(&minhaPilha, 30);

    printf("Topo da pilha: %d\n", peek(&minhaPilha)); // Deve imprimir
30

    printf("Elemento removido: %d\n", pop(&minhaPilha)); // Remove 30
    printf("Novo topo: %d\n", peek(&minhaPilha)); // Agora deve ser 20

    return 0;
}
```

📖 Explicação:

- Criamos a pilha.
- Inserimos 3 elementos: 10 → 20 → 30.
- Vemos o topo: 30.
- Removemos (pop): sai 30.
- Vemos o novo topo: agora 20.

🎯 Resumo Visual da Pilha Durante a Execução:

1. **push(10)**

```
 [10]
topo → 0
```

2. **push(20)**

```
[10, 20]
topo → 1
```

3. **push(30)**

```
[10, 20, 30]
topo → 2
```

4. **peek() → 30**

5. **pop() → remove 30**

6. **peek() → 20**

Perfeito!
Agora vou montar a **versão dinâmica da pilha em C**, usando **ponteiros** e **alocação dinâmica** (`malloc` e `free`), explicando cada passo de maneira detalhada, como um professor faria.

Essa versão é **mais poderosa**: ela não tem um limite fixo de tamanho (não fica presa a um `#define MAX`).

📚 Pilha Dinâmica em C — Passo a Passo

🏗️ Primeiro: Estrutura da Pilha Dinâmica

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

// Estrutura do nó da pilha
typedef struct No {
    int dado;           // dado armazenado
    struct No *proximo;  // ponteiro para o próximo nó
} No;

// Estrutura da pilha
typedef struct {
    No *topo; // ponteiro para o topo da pilha
} Pilha;
```

📖 Explicação:

- Um **nó** (`No`) guarda um valor (`dado`) e aponta para o próximo nó (`proximo`).
- A **pilha** mantém apenas um ponteiro para o **topo**.
- Cada novo elemento é um **novo nó** na memória.

🛠️ Segundo: Inicializar a Pilha

```
void inicializar(Pilha *p) {
    p->topo = NULL; // pilha começa vazia
}
```

📖 Explicação:

- **NULL** significa que não há nenhum nó ainda.

📦 Terceiro: Inserir Elemento (Push)

```
void push(Pilha *p, int valor) {
    No *novoNo = (No *)malloc(sizeof(No)); // aloca memória para o novo
nó
    if (novoNo == NULL) {
        printf("Erro de alocação de memória!\n");
        exit(1);
    }

    novoNo->dado = valor;          // coloca o valor no nó
    novoNo->proximo = p->topo;      // o novo nó aponta para o antigo
topo
    p->topo = novoNo;               // atualiza o topo da pilha
}
```

📖 Explicação:

- `malloc` cria espaço para um novo nó.
- O novo nó guarda o valor e aponta para o topo anterior.
- O topo da pilha passa a ser o novo nó.
- Cada `push` **cresce** a pilha **para cima**.

📤 Quarto: Remover Elemento (Pop)

```c
int pop(Pilha *p) {
    if (p->topo == NULL) {
        printf("Erro: Pilha vazia!\n");
        exit(1);
    }

    No *temp = p->topo;          // guarda o nó atual
    int valor = temp->dado;      // guarda o valor para retornar
    p->topo = temp->proximo;     // move o topo para o próximo nó
    free(temp);                  // libera a memória do nó removido
    return valor;
}
```

📖 Explicação:

- Guarda o topo atual.
- Atualiza o topo para o próximo nó.
- Libera a memória do nó que foi removido.
- Retorna o valor.

🔍 Quinto: Ver o Elemento no Topo (Peek)

```c
int peek(Pilha *p) {
    if (p->topo == NULL) {
        printf("Erro: Pilha vazia!\n");
        exit(1);
    }
    return p->topo->dado;
}
```

📖 Explicação:

- Só lê o valor no topo, sem remover.

🧹 Sexto: Função para Liberar Toda a Pilha

```c
void liberarPilha(Pilha *p) {
    while (p->topo != NULL) {
        No *temp = p->topo;
        p->topo = temp->proximo;
        free(temp);
```

```
    }
}
```

📖 Explicação:

- Percorre todos os nós e libera um a um.
- Muito importante para **evitar vazamentos de memória**.

🧪 Sétimo: Exemplo de Uso

```c
int main() {
    Pilha minhaPilha;
    inicializar(&minhaPilha);

    push(&minhaPilha, 10);
    push(&minhaPilha, 20);
    push(&minhaPilha, 30);

    printf("Topo da pilha: %d\n", peek(&minhaPilha)); // Deve mostrar 30

    printf("Elemento removido: %d\n", pop(&minhaPilha)); // Remove 30
    printf("Novo topo: %d\n", peek(&minhaPilha)); // Deve mostrar 20

    liberarPilha(&minhaPilha); // Libera toda a memória alocada

    return 0;
}
```

🎯 Resumo Visual da Pilha Dinâmica

Cada elemento é uma **célula conectada**:

Antes de `pop()`:

```
Topo → [30] → [20] → [10] → NULL
```

Depois de `pop()`:

```
Topo → [20] → [10] → NULL
```

Cada `[valor]` é um nó (`struct No`).

📜 Principais Diferenças entre Pilha Estática e Dinâmica

Aspecto	Pilha Estática	Pilha Dinâmica
Limite	Fixo (MAX)	Flexível (até memória acabar)
Memória	Contígua (array)	Espalhada (nós separados)
Implementação	Simples	Mais complexa (ponteiros, malloc/free)
Uso de memória	Pode desperdiçar espaço	Uso eficiente

🧠 Conclusão de Professor

- Em **sistemas embarcados** (memória limitada), pilhas estáticas ainda são preferidas.
- Em **programação geral** (sistemas modernos), pilhas dinâmicas são mais flexíveis e seguras contra overflow.
- Em ambos os casos, a pilha modela a ideia **LIFO** de maneira **clara**, e a escolha entre estática ou dinâmica depende dos requisitos do problema.

🎯 Conclusão

Pilhas são uma das **estruturas fundamentais** da ciência da computação, com aplicações teóricas (autômatos, gramáticas) e práticas (execução de programas, algoritmos).
Sua simplicidade e eficiência fazem delas uma ferramenta essencial para estruturar tanto dados quanto processos.
