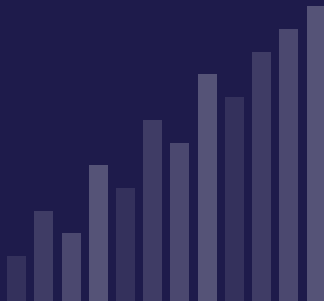

AULA 11 · BLOCO 2: DICIONÁRIOS E ESTRUTURAS LINEARES

Pilhas

Seção 10.1

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Implementar pilhas em vetor (com transbordamento) e em lista ligada
- ▶ Analisar Push e Pop em $O(1)$
- ▶ Avaliar expressões pós-fixas com uma pilha
- ▶ Medir o custo por operação das duas implementações



Teoria

LIFO, as operações, as duas implementações e o que pode dar errado



Prática em C

Pilha em vetor e em lista; avaliador de expressões pós-fixas



Experimento

Tempo por operação: vetor contra lista ligada

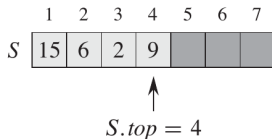


Conexões

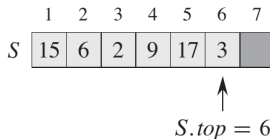
Compiladores, navegadores, sistemas operacionais e IA

Último a entrar, primeiro a sair

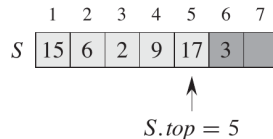
- ▶ O elemento removido é sempre o **mais recente**: política LIFO.
- ▶ $S.topo$ indica o último; a pilha é $S[1..S.topo]$. Vazia se $S.topo = 0$.
- ▶ Remover de pilha vazia é **esvaziamento** (*underflow*); inserir em pilha cheia é **transbordamento** (*overflow*).



(a)



(b)



(c)

Figura 10.1 de Cormen et al. (2009) — pilha S em vetor: (a) 4 elementos; (b) após Push(17) e Push(3); (c) após Pop, que devolve 3

As operações

- ▶ Todas $O(1)$.
- ▶ **Em vetor:** memória contígua, rápida; capacidade fixa (ou vetor dinâmico, aula 13).
- ▶ **Em lista ligada:** Push insere na cabeça, Pop remove a cabeça (exercício 10.2-2). Sem limite, mas um malloc por elemento.

Push(S, x) e Pop(S)

```
1                                     // Push( $S, x$ )
2  $S.topo \leftarrow S.topo + 1$ 
3  $S[S.topo] \leftarrow x$ 
4
5                                     // Pop( $S$ )
6 se Stack-Empty( $S$ ) então
7   erro "esvaziamento"
8 senão
9    $S.topo \leftarrow S.topo - 1$ 
10  retorne  $S[S.topo + 1]$ 
```

Aplicação: expressões pós-fixas

Notação pós-fixa (polonesa reversa): o operador vem depois dos operandos, sem parênteses.

$$(3 + 4) \times 2 \longrightarrow 3\ 4\ +\ 2\ \times$$

- ▶ Leia da esquerda para a direita: número $\Rightarrow$ Push; operador $\Rightarrow$ dois Pop, calcule, Push do resultado.
- ▶ No fim, a pilha tem exatamente um valor: a resposta. Qualquer outra situação é expressão inválida.
- ▶ Converter de infixa para pós-fixa também usa uma pilha (algoritmo *shunting-yard* de Dijkstra), com as precedências dos operadores.

Pilhas também verificam parênteses balanceados, implementam recursão (a pilha de chamadas) e o “desfazer dos editores.



Prática em C

- ▶ PilhaVetor com transbordamento e PilhaLista sem limite
- ▶ Mesma interface: empilha, desempilha, vazia
- ▶ Avaliador de expressões pós-fixas usando as duas

Pilhas e expressões pós-fixas em C (1/4)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  /* Duas implementações com a mesma interface: PUSH, POP, STACK-EMPTY. */
6  #define TAM 100
7  typedef struct { double v[TAM]; int topo; } PilhaVetor;    // Cormen: S[1..n], S.top
8
9  int pv_vazia(const PilhaVetor *S) { return S->topo == 0; }
10 int pv_empilha(PilhaVetor *S, double x) {
11     // transbordamento (overflow)
12     if (S->topo == TAM) return 0;
13     S->v[S->topo++] = x;
14     return 1;
15 }
```

Pilhas e expressões pós-fixas em C (2/4)

```
16 double pv_desempilha(PilhaVetor *S) { return S->v[--S->topo]; } // supõe não vazia
17
18 typedef struct No { double x; struct No *prox; } No; // pilha ligada: sem limite
19 typedef struct { No *topo; } PilhaLista;
20
21 void pl_empilha(PilhaLista *S, double x) { // insere na cabeça: O(1)
22     No *n = malloc(sizeof(No));
23     n->x = x;
24     n->prox = S->topo;
25     S->topo = n;
26 }
27 double pl_desempilha(PilhaLista *S) { // remove a cabeça: O(1)
28     No *n = S->topo;
29     double x = n->x;
```

Pilhas e expressões pós-fixas em C (3/4)

```
30     S->topo = n->prox;
31     free(n);
32     return x;
33 }
34
35 /* Expressão pós-fixa (notação polonesa reversa): operandos são empilhados; um
36 operador desempilha dois, calcula e empilha o resultado. */
37 int main(void) {
38     char tok[32];
39     PilhaVetor S = {{0}, 0};
40     PilhaLista L = {NULL};
41     while (scanf("%31s", tok) == 1) {
42         if (strchr("+-*/", tok[0]) && tok[1] == '\\0') {
43             if (S.topo < 2) { printf("expressão inválida\\n"); return 1; }

```

Pilhas e expressões pós-fixas em C (4/4)

```
44     double b = pv_desempilha(&S), a = pv_desempilha(&S);
45     double r = tok[0] == '+' ? a + b : tok[0] == '-' ? a - b
46               : tok[0] == '*' ? a * b : a / b;
47     pv_empilha(&S, r);
48     pl_desempilha(&L); pl_desempilha(&L);
49     pl_empilha(&L, r);
50 } else {
51     pv_empilha(&S, atof(tok));
52     pl_empilha(&L, atof(tok));
53 }
54 }
55 if (S.topo != 1) { printf("expressão inválida\n"); return 1; }
56 printf("%g\n", pv_desempilha(&S));
57 printf("(a pilha ligada concorda: %g)\n", pl_desempilha(&L));
58 return 0;
59 }
```



Experimento

- ▶ n Push seguidos de n Pop
- ▶ Tempo por operação, melhor de 5
- ▶ n de 4 mil a 4 milhões

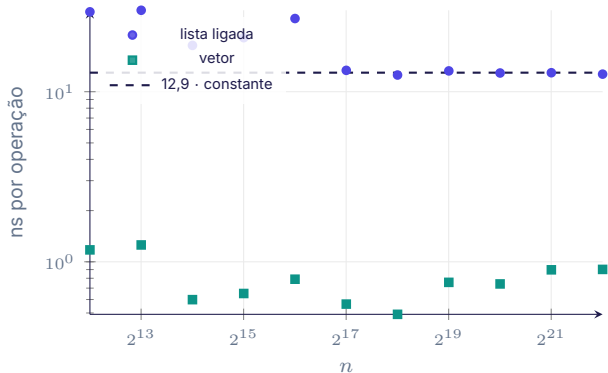
Mesmo $O(1)$, custos diferentes

Hipótese. As duas implementações são $O(1)$ por operação: o tempo **por operação** deve ficar **constante** quando n cresce. A constante deve ser bem maior na lista: cada Push chama malloc e cada Pop, free, e os nós ficam espalhados na memória.

Método

1. Vetor alocado de uma vez; lista com um nó por elemento
2. $n = 2^{12}, \dots, 2^{22}$
3. Tempo total dividido por $2n$ operações
4. Mínimo de 5 repetições

Tempo por operação



n	medido	razão
4.096	29,5	29,5
16.384	18,7	18,7
65.536	26,9	26,9
262.144	12,5	12,5
1.048.576	12,9	12,9
4.194.304	12,7	12,7

razão = medido $\div$ constante

✓ desvio máx. 3,2%

Lista: cerca de 7 ns por operação, constante. Vetor: menos de 1 ns, também constante (a oscilação é o cache).

EXPERIMENTO

O que o experimento mostrou

As duas implementações têm custo **constante** por operação, como prevê a análise, mas a lista ligada é cerca de **25 vezes** mais cara: malloc, free e saltos pela memória. A notação O esconde constantes; quando a capacidade é previsível, o vetor (ou o vetor dinâmico da aula 13) é a melhor pilha.

➤ Reproduza: `gcc -O2 aula11_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Compiladores
- ▶ Navegadores e editores
- ▶ Sistemas operacionais
- ▶ Inteligência artificial

Onde isso aparece na computação

Compiladores

Analísadores sintáticos casam parênteses e blocos com pilhas; a máquina virtual da JVM e a do Python são *máquinas de pilha*: `iadd` desempilha dois e empilha a soma.

Navegadores e editores

Voltar e avançar, desfazer e refazer: duas pilhas. Calculadoras HP usavam notação pós-fixa por ser mais rápida de digitar e de avaliar.

Sistemas operacionais

Cada *thread* tem sua pilha de chamadas; o estouro dela (*stack overflow*) é o transbordamento desta aula, em hardware.

Inteligência artificial

Busca em profundidade com *backtracking* (labirintos, sudoku, provadores de teoremas) guarda as alternativas pendentes numa pilha (aula 29).

Para casa

Exercícios

1. (10.1-1) Ilustre `Push(4)`, `Push(1)`, `Push(3)`, `Pop`, `Push(8)`, `Pop` numa pilha vazia em $S[1..6]$.
2. (10.1-2) Implemente duas pilhas num só vetor $A[1..n]$ sem transbordamento enquanto houver espaço.
3. (10.2-2) Implemente uma pilha com lista simplesmente ligada; `Push` e `Pop` em $O(1)$.
4. Implemente a conversão de infixa para pós-fixa (*shunting-yard*) com $+$, $-$, $\times$, $/$ e parênteses.
5. Use uma pilha para verificar se um arquivo HTML tem as marcas bem aninhadas.

Leitura

Cormen et al., seção 10.1 e exercício 10.2-2.

Código

josedihego.net/estruturas