Pilhas

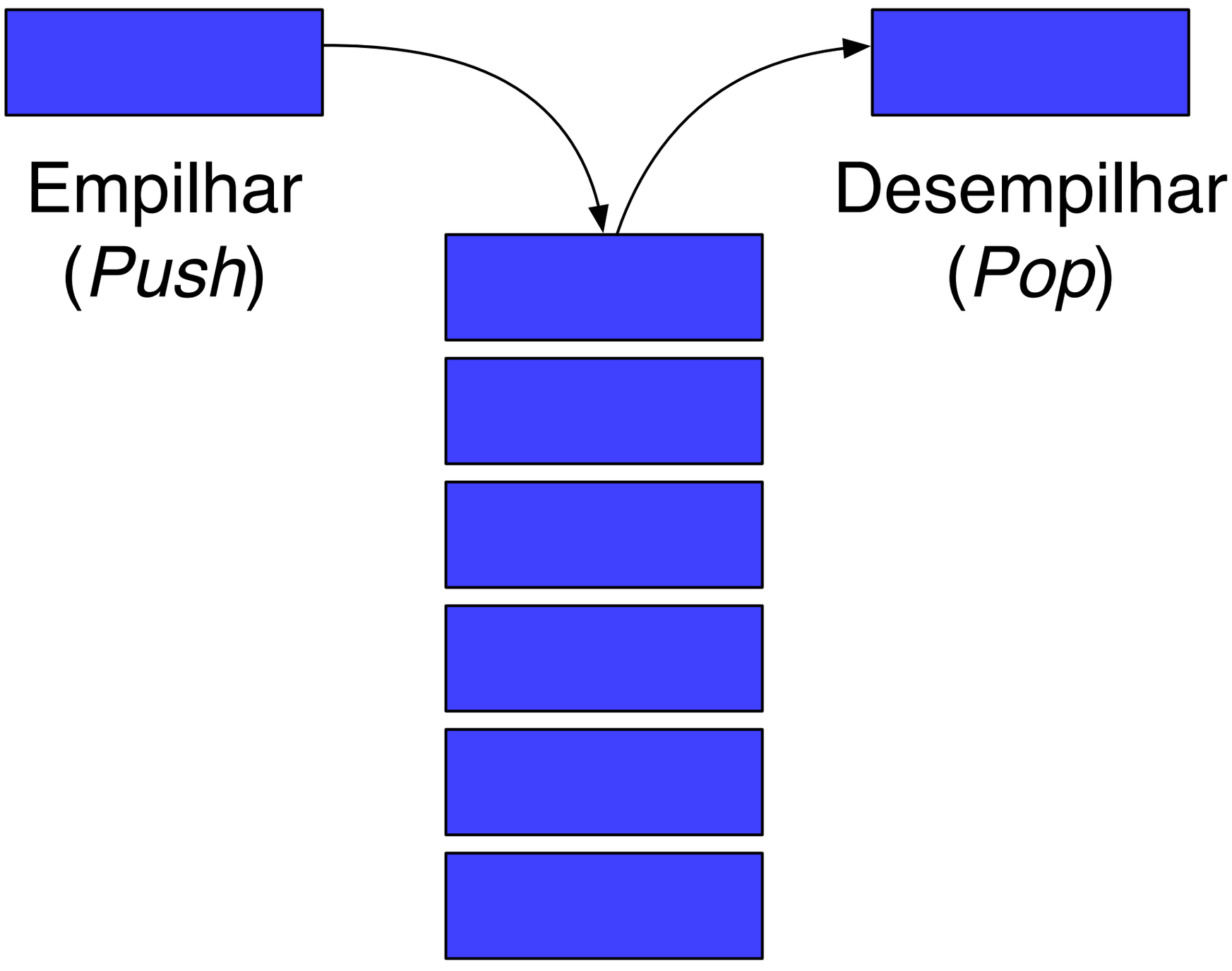
Algoritmos e Programação 2
Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

Introdução

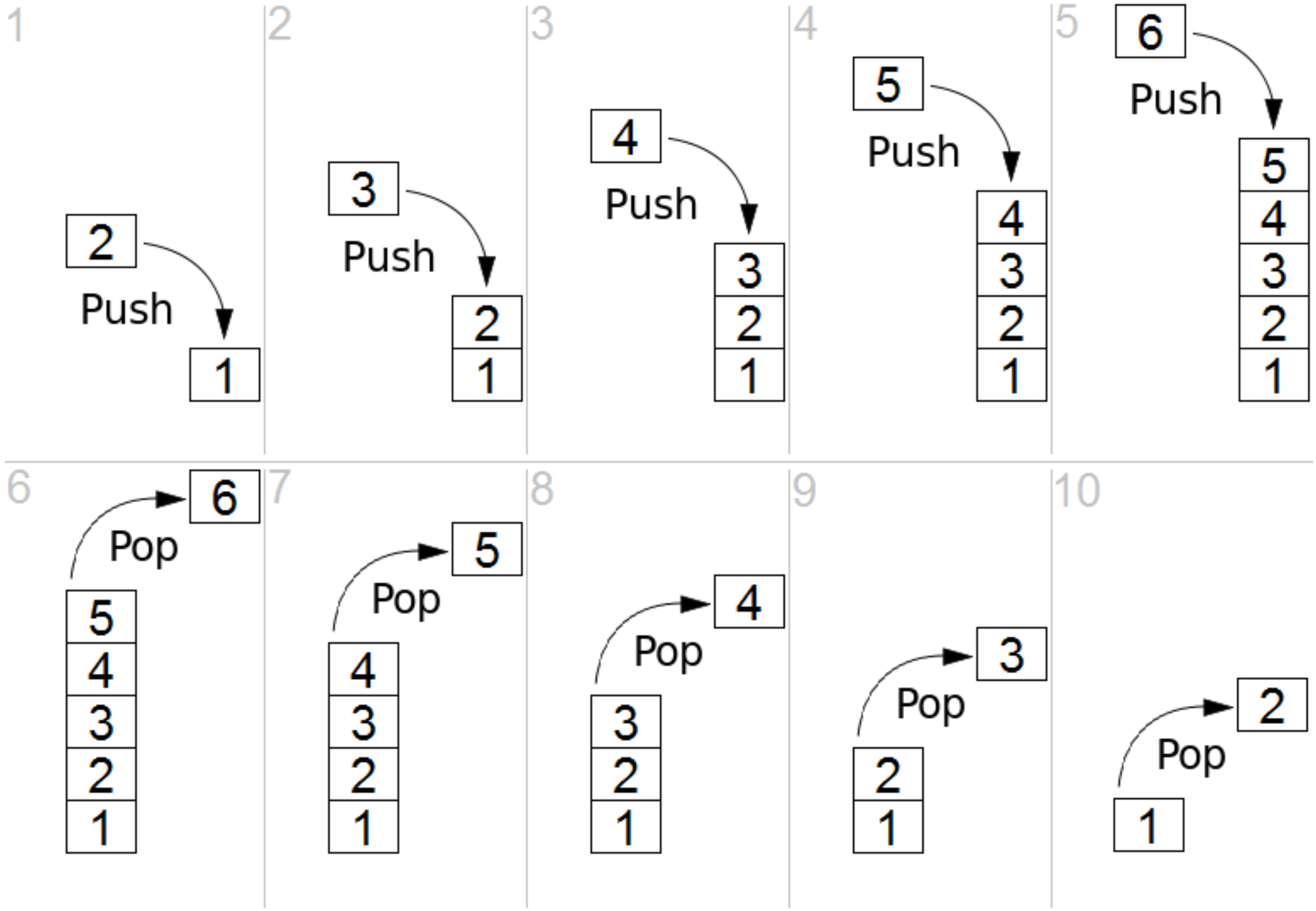
- **Pilhas** e **filas** são conjuntos dinâmicos que seguem uma política pré-especificada para inserção e remoção de elementos
- São um **Tipo Abstrato de Dados** (**TAD**, em inglês ADT – *Abstract Data Type*)
  - Descrição lógica
  - vs estrutura de dados que são representações concretas em nível de implementação

Pilha de Livros

Pilha TAD



Sequência de Operações



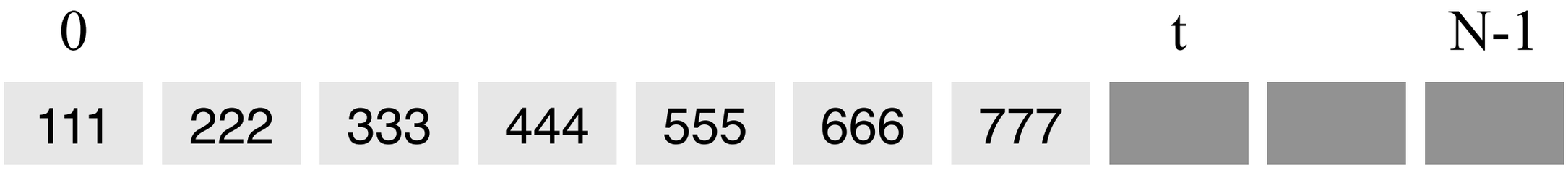
Definição

- **Pilha** é um **Tipo Abstrato de Dados (TAD)** que serve como uma coleção de elementos, com duas operações principais:
  - **empilhar** (push) adiciona um elemento para coleção;
  - **desempilhar** (pop) remove o elemento mais recentemente adicionado;
- A política de inserção e remoção dos elementos é conhecida pelo acrônimo **LIFO** (*Last In, First Out*), em português Último a Entrar, Primeiro a Sair

Como implementar?

- **TAD** não diz como implementar
- Existem diversas maneiras eficientes de implementar pilhas. Mostraremos como implementar utilizando **arranjos simples**, entretanto podemos implementá-las por meio de **listas encadeadas**

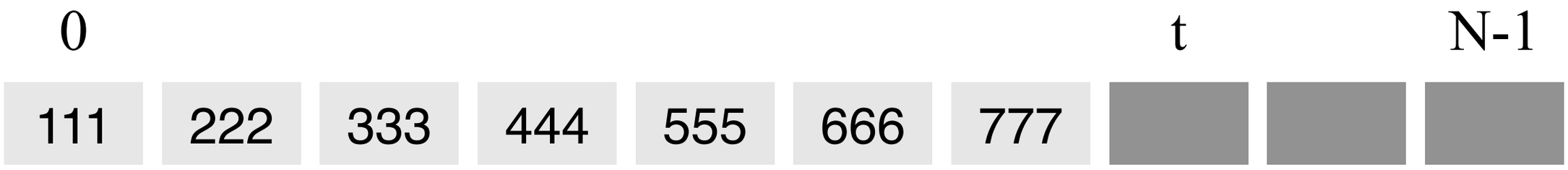
Implementação em Vetor



O vetor $p[0..t-1]$ armazena uma pilha.

- Suponha que nossa pilha está armazenada em um vetor $p[0..N-1]$
- A parte do vetor efetivamente ocupada pela pilha é $p[0..t-1]$
- Índice $t-1$ define o **topo** da pilha

Implementação em Vetor (cont.)



O vetor $p[0..t-1]$ armazena uma pilha.

- Pilha está **vazia** se t vale 0
- Pilha está **cheia** se t vale N

Inicializar

- Inicialmente devemos sinalizar que a pilha está vazia, portanto iremos inicializar $t = 0$

Desempilhar (*Remover*)

- Para **desempilhar** um elemento da pilha, ou seja, para remover um elemento, faça:

```
t = t - 1;
x = p[t];
```

- É claro que o programador não deve fazer isto se a pilha estiver vazia (*stack underflow*)

Topo (*Consultar*)

- Para **consultar** a pilha sem desempilhar basta fazer:

```
x = p[t-1];
```

Empilhar (*Inserir*)

- Para **empilhar** um objeto y, ou seja, para inserir y na pilha faça:

```
p[t] = y;
t = t + 1;
```

- Antes de empilhar, é preciso ter certeza de que a pilha não está cheia (*stack overflow*)
  - A tentativa de inserir em uma pilha cheia é um indício de mau planejamento lógico do seu programa

Algoritmo: Empilhar

```
void empilhe(int p[], int &t, int x) {
  p[t] = x;
  t = t + 1;
}
```

Algoritmo: Desempilhar

```
int desempilhe(int p[], int &t) {
  t = t - 1;
  return p[t];
}
```

Overflow e Underflow

- *Overflow*: Ocorre quando se tenta inserir um elemento em uma estrutura que já atingiu a sua capacidade máxima;
- *Underflow*: Ocorre quando se tenta remover um elemento em uma estrutura que não possui elementos;

Aplicação: Notação posfixa

- Expressões aritméticas são usualmente escritas em notação **infixa**
  - Operadores ficam entre os operandos
- Na notação **posfixa** (ou *polonesa*) os operadores ficam depois dos operandos

Notação infixa	Notação posfixa
(A + B * C)	A B C * +
(A * (B + C) / D - E)	A B C + * D / E -
(A + B * (C - D * (E - F) - G * H) - I * 3	A B C D E F - * - G H * - * + I 3 * -
(A + B * C / D * E - F)	A B C * D / E * + F -
(A * (B + (C * (D + (E * (F + G))))))	A B C D E F G + * + * + *

Expressões aritméticas em notação infixa e notação posfixa. A notação posfixa dispensa parênteses. Os operandos (A, B etc.) aparecem na mesma ordem nas duas notações.

Problema

- Traduzir para notação posfixa uma expressão infixa dada
- Suporemos que:
  - A expressão infixa está correta
  - Cada nome de variável tem uma letra apenas
  - A expressão toda está embrulhada em um par de parênteses

```c
char infixa_para_posfixa(char infix[]) {
    /* PARTE 1 */
    char *posfix, x;
    char *p; int t;
    int n, i, j;
    n = strlen(infix);
    posfix = (char *) malloc (n * sizeof(char));
    p = (char *) malloc(n * sizeof(char));
    t = 0; p[t++] = infix[0];
    /* PARTE 2 */
    …
}
```

```c
char infixa_para_posfixa(char infix[]) {
    /* PARTE 1 */
    …
    /* PARTE 2 */
    for (j = 0, i = 1; infix[i] != '\0'; i++) {
        switch(infix[i]) {
            case '(': p[t] = infix[i]; t = t + 1;  /* Empilhe */
            break;
            case ')': while (1) {
                t = t - 1; x = p[t];  /* Desempilhe */
                if (x == '(') break;
                posfix[j] = x; j = j + 1; }
                break;
            case '+':
            case '-': while (1) {
                x = p[t-1];  /* Topo */
                if (x == ‘(') break;
                t = t - 1;
                posfix[j] = x; j = j + 1; }
                p[t] = infix[i]; t = t + 1;  /* Empilhe */
            break;
            case '*':
            case '/': while (1) {
                x = p[t-1];  /* Topo */
                if (x == '(' || x == '+' || x == '-')
                    break;
                t = t - 1;
                posfix[j] = x; j = j + 1; }
                p[t] = infix[i]; t = t + 1;  /* Empilhe */
                break;
            default: posfix[j] = infix[i]; j = j + 1; }
    }
    free(p);
    posfix[j] = '\0';
    return posfix;
}
```

infix[0..i-1]	p[0..t-1]	posfix[0..j-1]
(	(	
(A	(	A
(A *	(*	A
(A * (	(* (	A
(A * (B	(* (	A B
(A * (B	(* (	A B
(A * (B *	(* (*	A B
(A * (B * C	(* (*	A B C
(A * (B * C +	(* (+	A B C *
(A * (B * C + D	(* (+	A B C * D
(A * (B * C + D)	(*	A B C * D +
(A * (B * C + D))		A B C * D + *

Resultado da aplicação da função *infixa_para_posfixa*.

A Pilha de Execução de um Programa

- Todo programa em C/C++ é composto por uma ou mais funções, sendo **main** a primeira função a ser executada
- Para executar um programa, o computador usa uma "**pilha de execução**"

Pilha de Execução

- Ao encontrar a invocação de uma função, o computador cria um novo “espaço de trabalho”, que contém todos os parâmetros e todas as variáveis locais da função
- Esse espaço de trabalho é colocado na **pilha de execução** (sobre o espaço de trabalho que invocou a função) e a execução da função começa (confinada ao seu espaço de trabalho)

Pilha de Execução (cont.)

- Quando a execução da função termina, o seu espaço de trabalho é retirado da pilha e descartado
- O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido interrompida

Exemplo Pilha de Execução

```c
int g(int a, int b) {
    return a + b;
}

int f(int i, int j, int k) {
    int x;
    x = /* 2 */ g(i, j);
    return x + k;
}
```

```c
int main (void) {
    int i, j, k, y;

    i = 111; j = 222; k = 444;
    y = /* 1 */f(i, j, k);
    printf("%d\n", y);
    return 0;
}
```

Pilha de Execução (cont.)

O programa é executado da seguinte maneira:

1. Um espaço de trabalho é criado para a função **main**;
2. No ponto 1, a execução de **main** é suspensa e um espaço de trabalho para a função **f** é colocada na pilha;
3. No ponto 2, a execução da **f** é suspensa e um espaço de trabalho para a função **g** é colocado na pilha;
4. Quando a execução da **g** termina, a função devolve 333;
5. Quando a execução de **f** termina, a função devolve 777;

Pilha de Execução (cont.)

- No nosso exemplo, **f** e **g** são funções distintas
  - Mas tudo funcionaria da mesma maneira se **f** e **g** fossem idênticas, ou seja, se **f** fosse uma função recursiva

Referências

- TENENBAUM, Aaron M; ANGENTEIN, Moshe; LANGSAM, Yedidyah. Estruturas de dados usando C. Sao Paulo, SP: Pearson, 1995. 884p.
- FEOFILOFF, P. Algoritmos em Linguagem C, 1. ed. Rio de Janeiro: Elsevier, 2008.
- CORMEN, T. H. [et al]. Algoritmos: teoria e prática. 3a ed. Rio de Janeiro: Elsevier, 2012.