# Algoritmos, Pilhas

Prof. Ricardo Reis
Universidade Federal do Ceará
Estruturas de Dados

15 de Maio de 2013

## 1 Definição

Uma *pilha* (ou *stack* em inglês) é uma estrutura de dados linear elementar que permite armazenar e recuperar objetos sobre as seguintes restrições ,

- Só é possível inserir (*empilhar*) ou remover (*desempilhar*) um objeto de uma extremidade pré-estabelecida da estrutura denominada *topo da pilha* (Figura-1).

- Só é possível ler dados do objeto que estiver no topo da pilha. Diz-se que os demais objetos estão *abaixo do topo*. Estes não podem ser acessados diretamente.

- Uma pilha que não comporta mais objetos (*pilha cheia*) deve reportar tentativas de empilhamentos sem sucesso (*stack overflow*) ao passo que uma pilha *sem* objetos (*pilha vazia*) deve reportar tentativas de desempilhamento sem sucesso (*stack underflow*).

O número de objetos empilhados em uma pilha define a *altura da pilha*. O máximo de objetos que uma pilha suporta é denominado *capacidade da pilha*. Uma pilha vazia tem altura zero e uma pilha cheia possui altura igual a sua capacidade.

A ordem de empilhamento em uma pilha é sempre inversa a ordem de desempilhamento. Logo o primeiro objeto empilhado será o último a ser desempilhamento. Por essa razão as pilhas são denominadas estruturas FILO (do inglês, *First In Last Out*, ou, *primeiro a entrar último a sair*).

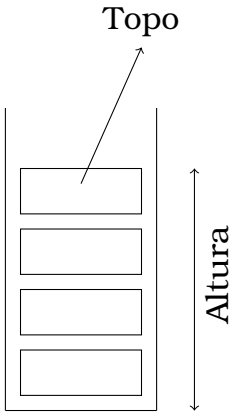


**Figura 1: Esquema de Pilha**

## 2 Operadores de Pilhas

Os operadores básicos de uma pilha, como estrutura de dados, são,

- **Create**: Operador responsável por construir uma nova pilha.

- **Push**: Operador que empilha um novo objeto no topo de uma pilha consequentemente aumentando sua altura em uma unidade quando a estrutura não estiver cheia.

- **Pop**: Operador que desempilha o objeto no topo da pilha consequentemente diminuindo sua altura em uma unidade quando a estrutura não estiver vazia.

- **Top**: Operador que retorna o valor do objeto no topo de uma pilha.
- **Empty**: Operador que testa se a pilha está vazia.
- **Full**: Operador que testa se a pilha está cheia.
- **Destroy**: Operador que elimina uma dada pilha e seus objetos se existirem.

Note que são utilizados nomes em inglês para os operadores de pilhas (por questões de uso convencional na literatura sobre o assunto).

# 3 Pilhas Sequenciais

Uma pilha sequencial é aquela construída utilizando-se um vetor como estrutura base. De uma forma geral o acesso aleatório próprio de um vetor é camuflado pela implementação da pilha sendo assim as propriedades de pilhas meramente simuladas.

A Tabela-1 ilustra o descritor de uma pilha sequencial. A implementação dos respectivos operadores é apresentada nos parágrafos seguintes.

O Algoritmo-1 implementa a construção de uma pilha sequencial. De forma similar a uma lista sequencial, é alocado um vetor dinâmico $M$ de comprimento $n$ (capacidade da pilha). A altura é referenciada por $h$ e como a pilha está inicialmente vazia seu valor inicial é zero. Uma nova pilha $\mathcal{P}$ é retornada como saída.

**Algoritmo 1** Construtor de Pilha Sequencial

```
1: Função CREATE(n)
2:     P.M ← alocar(n)
3:     P.n ← n
4:     P.h ← 0
5:     Retorne P
```

O Algoritmo-2 implementa o empilhador de pilha sequencial. A função deste operador é empilhar o objeto $x$ na pilha de entrada $\mathcal{P}$. Inicialmente testa-se de a pilha ainda contém espaço (altura deve ser menor que a capacidade ou $\mathcal{P}.h < \mathcal{P}.n$). Em seguida aumenta-se a altura $h$ em uma unidade e dispõe-se $x$ no novo topo.

**Algoritmo 2** Empilhador de Pilha Sequencial

```
1: Função PUSH(ref P, x)
2:     Se P.h < P.n então
3:         P.h ← P.h + 1
4:         P.M[P.h] ← x
```

O Algoritmo-3 implementa o desempilhamento em pilha sequencial. Este operador testa se a pilha não está vazia ($h > 0$) e em caso de êxito (ou seja, há um topo a eliminar) diminui a altura da pilha em uma unidade. Note que de fato objeto algum foi eliminado mas como $h$ é reduzido então no próximo empilhamento o antigo topo será sobrescrito.

**Algoritmo 3** Desempilhador de Pilha Sequencial

```
1: Função POP(ref P)
2:     Se P.h > 0 então
3:         P.h ← P.h − 1
```

No Algoritmo-4 é implementado o operador de leitura de topo de um pilha sequencial. Ele simplesmente retorna o valor $M[h]$ referente ao último objeto empilhado. Note que não há verificação de altura haja vista este operador não modificar a pilha.

**Algoritmo 4** Topo de Pilha Sequencial

```
1: Função TOP(ref P)
2:     Retorne P.M[P.h]
```

No Algoritmo-5 é implementado o operador de verificação de pilha sequencial vazia. Ele simplesmente checa se a altura $h$ da pilha vale zero.

No Algoritmo-5 é implementado o operador de verificação de pilha sequencial cheia. Ele checa se a altura $h$ se igualou a capacidade $n$.

A destruição de uma pilha sequencial é implementada pelo operador do Algoritmo-7. Note que ele se restringe a desalocar o

Tabela 1: Descritor de uma pilha Sequencial

| **Atributo** | **Descrição** |
|---|---|
| $M$ | vetor base |
| $n$ | capacidade da pilha |
| $h$ | Altura da pilha |

| **Operador** | **Argumentos** | **Descrição** |
|---|---|---|
| CREATE | $n$ | Cria uma pilha sequencial de capacidade $n$ |
| PUSH | $\mathcal{P}$, $x$ | Empilha $x$ na pilha $\mathcal{P}$ |
| POP | $\mathcal{P}$ | Desempilha topo da pilha $\mathcal{P}$ |
| TOP | $\mathcal{P}$ | Retorna valor no topo da pilha $\mathcal{P}$ |
| EMPTY | $\mathcal{P}$ | Verifica se a pilha $\mathcal{P}$ está vazia |
| FULL | $\mathcal{P}$ | Verifica se a pilha $\mathcal{P}$ está cheia |
| DESTROY | $\mathcal{P}$ | Elimina a pilha $\mathcal{P}$ |

**Algoritmo 5** Verificador de Pilha Sequencial Vazia

```
1: Função EMPTY(ref P)
2:     Retorne P.h = 0
```

**Algoritmo 6** Verificador de Pilha Sequencial Cheia

```
1: Função FULL(ref P)
2:     Retorne P.h = P.n
```

vetor base $M$ e anular ambas capacidade ($n$) e altura ($h$) da pilha.

**Algoritmo 7** Destrutor de Pilha Sequencial

```
1: Função DESTROY(ref P)
2:     Desalocar(P.M)
3:     P.h ← P.n = 0
```

Por fim note que a pilha $\mathcal{P}$ é repassada aos operadores como referência. Isto de fato só é necessário quando o operador muda a pilha, o que é o caso de PUSH e POP, mas por questões de homogeneidade de implementação foi mantido o **ref** em todos os operadores.

# 4 Pilhas Encadeadas

Uma pilha encadeada é aquela que utiliza uma lista encadeada como base. A maior vantagem neste caso é o fato de a pilha ter capacidade ilimitada. Uma pilha encadeada mantém internamente, como numa lista encadeada, uma referência para o primeiro nodo e a quantidade de nodos conforme indica o descritor da Tabela-2. Note que usamos a notação $topo$ para denotar a referência ao primeiro nodo. De fato, como se perceberá adiante, os empilhamentos e desempilhamentos ocorrem acerca desta referência porque considera-se que o primeiro nodo seja o topo da pilha (o efeito real dessa estratégia são algoritmos de complexidade na ordem de $O(1)$).

Ainda sobre o descritor da Tabela-2 note que existem duas diferenças entre os operadores e aqueles equivalentes em listas sequenciais. A primeira é no construtor que não possui argumento haja vista pilhas encadeadas não terem valor de capacidade máxima. A segunda é a ausência de um operador FULL de checagem de pilha cheia que, pela mesma razão já mencionada no construtor, não faria sentido existir.

O operador de construção de pilha encadeada é implementado no Algoritmo-8. Nele os campos $nodo$ e $h$ de uma pilha $\mathcal{P}$ são respectivamente configurados para $\lambda$ e $0$ definindo uma pilha vazia.

No Algoritmo-9 é implementado o empilhador de pilha encadeada. Neste operador é criado um novo nodo $\mathcal{N}$ contendo em seu campo $chave$ o valor de empilhamento (repassado ao operador através do parâmetro

Tabela 2: Descritor de uma pilha encadeada

| **Atributo** | **Descrição** |
|---|---|
| $topo$ | Referência para o nodo no topo da pilha |
| $h$ | Altura da pilha |

| **Operador** | **Argumentos** | **Descrição** |
|---|---|---|
| CREATE | - | Cria uma pilha encadeada |
| PUSH | $\mathcal{P}$, $x$ | Empilha $x$ na pilha $\mathcal{P}$ |
| POP | $\mathcal{P}$ | Desempilha topo da pilha $\mathcal{P}$ |
| TOP | $\mathcal{P}$ | Retorna valor no topo da pilha $\mathcal{P}$ |
| EMPTY | $\mathcal{P}$ | Verifica se a pilha $\mathcal{P}$ está vazia |
| DESTROY | $\mathcal{P}$ | Elimina a pilha $\mathcal{P}$ |

**Algoritmo 8** Construtor de Pilha Encadeada

```
1: Função CREATE(n)
2:     P.topo ← λ
3:     P.h ← 0
4:     Retorne P
```

$x$). Este novo nodo substitui o nodo topo da pilha $\mathcal{P}$ de trabalho (primeiro argumento do operador) se tornando o novo nodo da estrutura. Esta substituição ocorre em $O(1)$ e corresponde a duas reconexões: a primeira que faz $\mathcal{N}$ apontar para o topo ($\mathcal{N}.prox \leftarrow \mathcal{P}.topo$) e a segunda que muda o topo para $nd$ ($\mathcal{P}.topo \leftarrow \mathcal{N}$). Note que ainda é necessário aumentar de uma unidade a altura da pilha ($\mathcal{P}.h \leftarrow \mathcal{P}.h + 1$).

**Algoritmo 9** Empilhador de Pilha Encadeada

```
1: Função PUSH(ref P, x)
2:     N.chave ← x
3:     N.prox = P.topo
4:     P.topo = N
5:     P.h ← P.h + 1
```

O operador de desempilhamento em pilha encadeada é implementado no Algoritmo-10. A pilha $\mathcal{P}$ passada como argumento, quando não vazia ($\mathcal{P}.h > 0$), sofre uma extração do nodo topo diminuindo sua altura da unidade ($\mathcal{P}.h \leftarrow \mathcal{P}.h - 1$). A extração possui três etapas: na primeira a referência $\mathcal{N}$ é utilizada para manter o topo da pilha ($\mathcal{N} \leftarrow \mathcal{P}.topo$); em seguida o topo é deslocado para o nodo seguinte ($\mathcal{P}.topo \leftarrow \mathcal{P}.nodo.prox$) e que corresponde, em termos de pilhas, ao elemento logo abaixo do topo; e finalmente o nodo referenciado por $\mathcal{N}$ é desalocado. Note que em caso de um único nodo a extração faz $\mathcal{P}.topo$ receber $\lambda$ e $h$ zero.

**Algoritmo 10** Desempilhador de Pilha Encadeada

```
1: Função POP(ref P)
2:     Se P.h > 0 então
3:         N ← P.topo
4:         P.topo = P.topo.prox
5:         P.h ← P.h − 1
6:         Desalocar(N)
```

O operador de verificação de topo de pilha encadeada é implementado no Algoritmo-11. Este operador simplesmente verifica e retorna o valor da chave do nodo topo ($\mathcal{P}.topo.chave$). Não há verificação de pilha vazia neste operador, necessidade essa suprida pelo operador EMPTY.

**Algoritmo 11** Topo de Pilha Encadeada

```
1: Função TOP(ref P)
2:     Retorne P.topo.chave
```

A verificação de vazio em pilha encadeada é implementada no Algoritmo-12. O operador simplesmente checa se a altura da pilha de entrada ($\mathcal{P}.h$) é nula. Outra alternativa, mostrada como comentário, é testar se a referência ao topo da pilha aponta para nulo ($\mathcal{P}.topo = \lambda$).

**Algoritmo 12** Verificador de Pilha Encadeada Vazia

```
1: Função EMPTY(ref P)
2:     Retorne P.h = 0          ▷ ou P.topo = λ
```

O operador de destruição de pilha encadeada é implementado pelo Algoritmo-13. De forma similar ao que ocorre em destruição de listas encadeadas, este operador executa um laço que continuamente efetua desempilhamentos e encerra quando a pilha de entrada, $\mathcal{P}$, se esvazia.

**Algoritmo 13** Destrutor de Pilha Encadeada

```
1: Função DESTROY(ref P)
2:     Enquanto não EMPTY(P) faça
3:         POP(P)
```

# 5 Aplicações de Pilhas

## 5.1 Conversão de Base

A *conversão de base* de um número, expresso numa determinada base numérica (em geral a decimal) é o processo de transformação deste número para outra base. Por exemplo, a conversão de 78, que está em base decimal, para a base binária (2) revela o valor 1001110. Matematicamente expressa-se esta conversão como,

$$78_{10} = 1001110_2$$

que lê-se *78 na base dez é igual a 1001110 na base 2*.

Numa determinada base numérica $b$ o total de dígitos disponíveis para construção de um número vale exatamente $b$. Assim, por exemplo, a base decimal possui os dígitos $0 \cdots 9$. Numa base numérica de valor inferior a dez utiliza-se, em geral, um subconjunto destes mesmos dígitos. Por exemplo, na base octal (8), números são expressos utilizando-se os dígitos $0 \cdots 7$. Logo nenhum número em base octal possui os dígitos 8 ou 9. Como regra geral, quando $b \leq 10$ os dígitos utilizados na representação de números na base $b$ são $0 \cdots b-1$.

Quando a base numérica é superior a dez faz-se necessário o uso de letras para representação do décimo primeiro dígito em diante. Por exemplo, na base hexadecimal (16) os primeiros dez dígitos são representados por $0 \cdots 9$ e os seis restantes pelas letras $A \cdots F$. Como ilustração,

$$712498_{10} = ADF32_{16}$$

Um *mapa de conversão* de uma base $b$ é uma string dos dígitos desta base concatenados em ordem crescente. Por exemplo, na base hexadecimal o mapa de conversão, $\mathcal{M}_{16}$, é,

$$\mathcal{M}_{16} \leftarrow \texttt{'0123456789ABCDEF'}$$

Um algoritmo de conversão de base de um número inteiro positivo em base decimal para um valor de base $b$, tal que $1 < b \leq 16$, é descrito a seguir,

1. Definir o mapa de conversão $\mathcal{M}_{16}$
2. Definir uma pilha $\mathcal{P}$ de inteiros.
3. Empilhar o resto da divisão de $n$ por $b$.
4. Atualizar $n$ para o quociente de $n$ por $b$.
5. Caso $n$ não seja nulo voltar ao passo 3.
6. Desempilhar o topo $t$ de $\mathcal{P}$.
7. Imprimir $\mathcal{M}_{16}[t+1]$.
8. Caso $\mathcal{P}$ não esteja ainda vazia, voltar ao passo 6.

A sequência de dígitos impressa é a representação do valor de $n$ de entrada na base arbitrária $b$. Note que no passo 7 o dígito de impressão é determinado pelo mapeamento de $t+1$ utilizando $\mathcal{M}_{16}$. Isso se dá porque os restos de divisão empilhados ficam na faixa $0..15$, mas os índices de $\mathcal{M}_{16}$ estão na faixa $1..16$.

O Algoritmo-14 implementa o processo descrito anteriormente.

**Algoritmo 14** Conversão de número decimal para outra base numérica

```
1: Função CONV(n, b)
2:     M_16 ← '0123456789ABCDEF'
3:     P ← CREATE()
4:     Enquanto n > 0 faça
5:         PUSH(P, n mod b)
6:         n ← ⌊n/b⌋
7:     Enquanto não EMPTY(P) faça
8:         t ← TOP(P)
9:         Escreva M_16[t + 1]
10:        POP(P)
11:    DESTROY(P)
```

## 5.2 Avaliação de Expressões Algébricas

### 5.2.1 Representação Pósfixa

Em expressões algébricas a utilização de operadores *binários* (que operam dois operandos) é comumente *infixa*, ou seja, são dispostos entre os operadores. Por exemplo, na expressão $A + B$ o operador $+$ é binário pois opera $A$ e $B$ e é infixo por se dispor entre estas variáveis. A consequência direta desta notação é a necessidade do uso de parênteses quando se deseja quebrar a precedência entre os operadores [1]. Por exemplo, nas expressões $A + B \times C$ e $(A + B) \times C$ respectivamente ocorrem nesta ordem multiplicação/soma e soma/multiplicação. Pela possibilidade de uso de parênteses as expressões infixas são também às vezes chamadas *expressões parentizadas*.

Em geral computar expressões parentizadas é problemático porque os parênteses (ou a falta deles) geram ambiguidades. Para contornar isso utiliza-se a notação *pósfixa* onde os operadores são dispostos após os operadores. Por exemplo $A + B$ em notação pósfixa se torna $AB+$.

O processo de formação de uma expressão pósfixa ocorre por processamento de pares dispostos por precedência na expressão infixa. Por exemplo, a expressão,

$$(A + B \times (C - D))/E$$

em notação pósfixa deve se tornar,

$$ABCD - \times + E/$$

A lógica desta expressão é a seguinte. Os operandos devem manter-se na mesma ordem em que aparecem na expressão infixa e os operadores devem surgir logo após ao par de operandos que devem computar. Assim, como o primeiro operando é o $-$ e há o par $CD$ imediatamente antes, então a primeira operação computada deverá ser $C-D$. Em seguida o par de $\times$ é formado por $B$ e a parcela já computada, $C-D$, obtendo-se a computação parcial $B \times (C-D)$. O próximo operador, $+$, opera o par formado por $A$ e a computação de $B \times (C-D)$ obtendo-se $A + B \times (C - D)$. Por fim a computação de $/$ opera o par formado entre $A + B \times (C - D)$ e $E$ gerando $(A + B \times (C - D))/E$ que é a expressão inicial infixa.

### 5.2.2 Transformação de Expressões Infixas em Pósfixas

Apresentaremos nessa sessão um algoritmo que utiliza uma pilha para transformar uma expressão infixa (parentizada) em notação pósfixa. Os elementos utilizados neste algoritmo são,

- Uma expressão infixa de entrada $E$ formada pelos operadores $+$, $-$, $\times$ e $/$, por parênteses ( e ) e por variáveis de um único caractere na faixa $A \ldots Z$.
- Uma pilha $\mathcal{P}$ que empilha apenas os operadores $+$, $-$, $\times$, $/$ e ( (abre-parêntese).

[1] Os operadores aritméticos com maior ordem de precedência são a multiplicação e a divisão e de menor precedência a soma e a subtração. Concretamente isso equivale a dizer que numa expressão algébrica devem ser feitas primeiramente multiplicações e divisões e só então somas e subtrações. Por exemplo, a expressão,

$$3 + 5 \times 7$$

vale 38 pois deve-se primeiro multiplicar para então depois somar.

- Uma string $R$ de saída onde será armazenada a expressão pósfixa e que inicialmente está vazia.

O algoritmo de transformação de uma expressão infixa para pósfixa consiste em varrer $E$ da esquerda para a direita e processar cada caractere $ch$ da forma seguinte,

- Se $ch$ for uma variável (ou seja, pertence a $A \ldots Z$) então deve ser concatenado a string $R$.
- Se $ch$ for um operador aritmético ($+$, $-$, $\times$, $/$) deve ser empilhado em $\mathcal{P}$ sobre as seguintes restrições,
  - Se $\mathcal{P}$ estiver vazia ou a precedência do operador no topo de $\mathcal{P}$ for menor que o operador em $ch$ então $ch$ deverá ser diretamente empilhado em $\mathcal{P}$.
  - Se o operador no topo de $\mathcal{P}$ tiver precedência maior ou igual que aquele em $ch$ então deve-se sucessivamente desempilhar operadores de $\mathcal{P}$ até que a pilha se esvazie ou até que a prioridade do operador no topo dela seja menor que o operador em $ch$. Só então deve-se empilhar $ch$ em $\mathcal{P}$. Os operadores desempilhados de $\mathcal{P}$ devem ser concatenados a $R$ na ordem de desempilhamento.
- Se $ch$ for um abre-parêntese deve ser empilhado diretamente em $\mathcal{P}$.
- Se $ch$ for um fecha-parêntese deve-se efetuar sucessivos desempilhamentos em $\mathcal{P}$ em busca do abre-parêntese correspondente. Quando encontrado deve-se fazer um desempilhamento adicional para eliminar o abre-parêntese agora no topo. Os operadores desempilhados de $\mathcal{P}$ devem ser concatenados a $R$ na ordem de desempilhamento.

Se no final da varredura a pilha $\mathcal{P}$ não estiver vazia então deve-se executar sucessivos desempilhamentos até esvaziá-la. Os operadores desempilhados de $\mathcal{P}$ devem ser concatenados a $R$ na ordem de desempilhamento. O conteúdo em $R$ após todas as concatenações representa a expressão pósfixa procurada.

Tabela 3: Precedência de Operadores

| Operadores | Valor de Precedência |
|---|---|
| $($ | 1 |
| $+ \; -$ | 2 |
| $\times \; /$ | 3 |

Os valores de prioridade dos operadores empilhados em $\mathcal{P}$ devem seguir a Tabela-3. Note que o operador abre-parêntese, ao contrário da intuição matemática, possui a menor prioridade. Isso se deve ao fato de que eles não podem impedir os operadores aritméticos de entrarem na pilha $\mathcal{P}$. Além do mais eles devem estar necessariamente em $\mathcal{P}$ para que o processamento dos fecha-parênteses ocorram normalmente.

No Algoritmo-15 a função PRIOR retorna a prioridade de um operador passado via argumento $ch$ (segue valores da Tabela-3). No Algoritmo-16 a função POSFIXAR implementa a transformação de expressões infixas em pósfixas a qual segue as quatro etapas descritas anteriormente.

**Algoritmo 15** Prioridade de um caractere de uma expressão

```
1: Função PRIOR(ch)
2:     Se ch = ( então
3:         Retorne 1
4:     senão
5:         Se ch ∈ {+, −} então
6:             Retorne 2
7:         senão
8:             Se ch ∈ {×, /} então
9:                 Retorne 3
```

### 5.2.3 Avaliando uma Expressão Algébrica

Avaliar uma expressão algébrica significa atribuir valores numéricos a suas variáveis e em seguida simplificá-la a um único nú-

**Algoritmo 16** Transformação de uma expressão Infixa em Pósfixa

```
1: Função PosFixar(E)
2:     n ← Comprimento(E)                          ▷ comprimento da expressão de entrada
3:     P ← Create(n)                               ▷ Pilha de Operadores
4:     R ← ∅                                       ▷ String de saída: inicia vazia
5:     Para i ← 1...n faça                         ▷ Laço de varredura
6:         ch ← E[i]
7:         Se ch ∈ {A...Z} então                   ▷ Encontra variável
8:             R ← R + ch
9:         senão Se ch ∈ {+, −, ×, /} então        ▷ Encontra operador
10:            t ← Prior(ch)
11:            Enquanto não Empty(P) e Prior(Top(P)) ≥ t faça
12:                R ← R + Top(P)
13:                Pop(P)
14:            Push(P, ch)
15:        senão Se ch = ( então                   ▷ Encontra abre-parêntese
16:            Push(P, ch)
17:        senão Se ch = ) então                   ▷ Encontra fecha-parêntese
18:            Enquanto Top(P) ≠ ( faça
19:                R ← R + Top(P)
20:                Pop(P)
21:            Pop(P)
22:    Enquanto não Empty(P) faça                  ▷ Esvazia pilha
23:        R ← R + Top(P)
24:        Pop(P)
25:    Destroy(P)                                  ▷ Destrói Pilha
26:    Retorne R                                   ▷ Retorna expressão pósfixa
```

mero. Por exemplo, avaliando,

$$A + B \times (C - D)$$

com $A = 4$, $B = 11$, $C = 2$ e $D = 5$ obtém-se,

$$4 + 11 \times (2 - 5) = -29$$

Para avaliar uma expressão algébrica $E$ de entrada deve-se convertê-la em uma expressão pósfixa $R$ e depois, utilizando uma pilha numérica $\mathcal{P}$ (que empilha números de ponto flutuante), aplicar o algoritmo descrito a seguir. Varre-se $R$ da esquerda para a direita e para cada caractere $ch$ visitado faz-se o seguinte,

- Se $ch$ for uma variável ($ch \in \{A \dots Z\}$) então deve-se empilhar em $\mathcal{P}$ seu equivalente valor numérico.
- Se $ch$ for um operador devem-se efetuar dois desempilhamentos em $\mathcal{P}$ recuperando-se o primeiro valor desempilhado através de uma variável auxiliar $y$ e o segundo através de uma variável auxiliar $x$. Em seguida deve-se operar $x$ e $y$ com o operador em $ch$ e por fim reempilhar em $\mathcal{P}$ o valor obtido. A ordem do par de desempilhamentos é inversa a da operação e por essa razão o primeiro desempilhamento é recuperado por $y$ e o segundo por $x$ tornando as operações $x + y$, $x - y$, $x \times y$ e $x \div y$ coerentes.

Quando esse processo se encerra a pilha $\mathcal{P}$ contém apenas um valor que corresponde ao valor da avaliação o qual pode ser obtido por um último desempilhamento.

O Algoritmo-17 implementa a avaliação de uma expressão pósfixa passada como entrada. As etapas são comentadas na implementação. O argumento de entrada $\mathcal{M}$

representa um *mapa de variáveis*. Internamente um mapa de variáveis corresponde a um vetor cujos índices são os caracteres na faixa $A \ldots Z$ e cujas células são representadas por pares do tipo numérico/lógico onde a parte numérica refere-se ao valor da variável e a parte lógica ao seu estado de utilização (usada ou não usada). Assim, por exemplo, num dado mapa $\mathcal{M}$, $\mathcal{M}[A].valor$ representa o valor da variável $A$ e $\mathcal{M}[A].usada$ (que é verdadeiro ou falso) indica se $A$ está ou não em uso.

A função MAPEAR, Algoritmo-18, implementa a geração de uma mapa de variáveis a partir de uma expressão algébrica de entrada $E$ (que pode ser infixa ou pósfixa). As etapas do algoritmo são as seguintes,

- Criar um mapa $\mathcal{M}$ com a descrição anterior (vetor indexado pelos caracteres $A \ldots Z$ cujas células contém campos $valor$ e $usada$)
- Carregar os campos $valor$ e $usada$ de todas as células respectivamente com zero e **Falso** (mapa vazio). Veja linha-2.
- Varrer a expressão (linha-6) de entrada $E$ em busca de caracteres na faixa $A \ldots Z$ (linha-8) e para cada variável encontrada requisitar seu valor ao usuário e armazená-la no campo $valor$ correspondente.
- Cada variável carregada deve ter o campo $usada$ mudado para **Verdadeiro** impedindo que seja solicitada mais de uma vez (teste da linha-9).

## 5.3 Transformação de Algoritmos Recursivos em Iterativos

Alguns algoritmos recursivos, como a busca binária, podem ser transformados em iterativos simplesmente utilizando-se um laço. Isso se dá porque o total de chamadas recursivas por iteração é única.

**Algoritmo 18** Gerador de mapa de variáveis

```
 1: Função MAPEAR(E)
 2:     Para ch ← {A...Z} faça
 3:         M[ch].valor ← 0
 4:         M[ch].usada ← Falso
 5:     n ← Comprimento(E)
 6:     Para k ← 1...n faça
 7:         ch ← E[i]
 8:         Se ch ∈ {A...Z} então
 9:             Se não M[ch].usada então
10:                 Escreva ch, ": "
11:                 Leia M[ch].valor
12:                 M[ch].usada ← Verdadeiro
13:     Retorne M
```

Quando mais de uma chamada recursiva ocorre por iteração então, para se obter uma versão não recursiva, em geral necessita-se uma estrutura auxiliar como uma pilha.

A ideia é substituir a pilha de recursão (de fato a recursividade constrói uma pilha implícita) por uma pilha *manual* que acumule informações relevantes referentes ao processo.

### 5.3.1 QuickSort Iterativo

Consideremos nessa sessão a transformação do algoritmo de ordenação rápida (quicksort) em forma iterativa utilizando uma pilha. A ideia de transformação é implementada pelo Algoritmo-19 e descrita a seguir.

1. Define-se uma pilha $\mathcal{P}$ de faixas (linha-4), ou seja, cada objeto $f$ empilhado traz dois campos $f.p$ e $f.q$ que denotam respectivamente os índices inferior e superior de uma faixa de vetor.
2. Empilha-se em $\mathcal{P}$ a faixa $[1, n]$ referente a um vetor $M$ de comprimento $n$ que se deseja ordenar (linha-5).
3. Enquanto a pilha $\mathcal{P}$ não estiver vazia efetua-se as etapas (linha-6),
   (a) Desempilha-se a faixa no topo de $\mathcal{P}$ e aplica-lhe o particionamento

**Algoritmo 17** Avaliação de uma expressão pósfixa

```
 1: Função AVALIAR(R, M)
 2:     n ← Comprimento(R)
 3:     P ← CREATE(n)                                   ▷ Pilha numérica
 4:     Para i ← 1...n faça                             ▷ Laço de varredura da expressão
 5:         ch ← R[i]                                   ▷ Caractere avaliado
 6:         Se ch ∈ {A...Z} então                       ▷ Empilha valor de variável
 7:             PUSH(P, M[ch].valor)
 8:         senão                                       ▷ Desempilha par, opera e reempilha
 9:             y ← TOP(P)
10:             POP(P)
11:             x ← TOP(P)
12:             POP(P)
13:             Se ch = + então PUSH(P, x + y)
14:             senão Se ch = − então PUSH(P, x − y)
15:             senão Se ch = × então PUSH(P, x ∗ y)
16:             senão Se ch = / então PUSH(P, x/y)
17:     x ← TOP(P)                                      ▷ Resgata valor da avaliação
18:     DESTROY(P)                                      ▷ Destrói pilha
19:     Retorne x                                       ▷ Retorna valor da avaliação
```

(linha-7).

(b) Reempilham-se as duas faixas providas pelo particionamento desde que sejam válidas, ou seja, o campo $p$ seja menor que o campo $q$ (Entre linha-9 e linha-16).

**Algoritmo 19** QuickSort iterativo utilizando uma pilha

```
 1: Função IQUICKSORT(ref L, n)
 2:     f.p ← 1                                         ▷ Faixa de índices do vetor
 3:     f.q ← n
 4:     P ← CREATE(n)
 5:     PUSH(P, f)
 6:     Enquanto não EMPTY(P) faça
 7:         f ← TOP(P)
 8:         r ← PART(L, f.p, f.q)
 9:         Se f.p < r − 1 então
10:             z.p ← f.p
11:             z.q ← r − 1
12:             PUSH(P, z)
13:         Se r + 1 < f.q então
14:             z.p ← r + 1
15:             z.q ← f.q
16:             PUSH(P, z)
```

É importante notar que o Algoritmo-19 mantém a essência da ordenação rápida, ou seja, progressivamente sub-faixas do vetor de entrada sofrem particionamentos que provocarão ao final a ordenação do vetor. Entretanto ao invés de uma *pilha de recursão* (que é o que se forma num processo recursivo), é utilizada uma pilha explícita, ou seja, visível ao implementador. Note que $\mathcal{P}$ cresce em altura enquanto as faixas oriundas do particionamento forem válidas começando entretanto a decair quando as faixas atingem tamanho igual a um ou zero deixando de entrar na pilha (testes da linha-9 e da linha-13).

# 6 Exercícios

*Utilize uma pilha para converter as expressões infixas a seguir em notação pósfixa,*

1. $A/(B+C)$
2. $(A+B)\times(C-D)$
3. $(A+(B\times((A-B)/E)))$
4. $(A\times(B/(C\times(D+A-E))))$

5. $((A\times(B-C))/(E/(D+E)))\times(A/(B\times(D-E)))$

*Utilize uma pilha para ilustrar todas as etapas de avaliação das expressões numéricas seguintes,*

6. $5 \times (56 - 9)$
7. $(3 - 7) \times (22 + 13)$
8. $((12 - 16) \times 25 - 4)/(3 - 7)$

*Resolva os problemas a seguir,*

9. O reverso de um número natural é outro natural obtido pela inversão de seus dígitos. Por exemplo, o reverso de 54132 é 23145. Escreva algoritmo que receba um natural e, utilizando uma pilha, retorne ser reverso.

10. Dadas duas pilhas contendo chaves ordenadas na ordem de empilhamento, escreva algoritmo que imprima essas chaves em ordem ascendente na saída padrão. O algoritmo deve ter complexidade $O(n)$ onde $n$ é a soma das alturas das pilhas.

11. Reimplemente a conversão em notação pósfixa de uma expressão algébrica de forma a aceitar também a operação de potência (uso o símbolo $\wedge$). A precedência da potência é maior que da multiplicação e divisão.

12. Use três pilhas para simular as movimentações de discos entre pinos no problema clássico da Torre de Hanói.

13. Implemente o algoritmo de ordenação de um vetor $M$ descrito a seguir o qual utiliza duas pilhas $A$ e $B$. A ideia é varrer sequencialmente as chaves em $M$ (denotadas por $x$) e empilhá-los em $A$ quando $A$ está vazia ou quando o valor em seu topo é menor ou igual a $x$. Se entretanto o valor no topo de $A$ é maior que $x$ deve-se (i) desempilhar continuamente o topo de $A$ e empilhá-lo em $B$ enquanto $A$ não estiver vazia e este valor for maior que $x$, (ii) empilhar $x$ em $A$ e por fim (iii) desempilhar continuamente o topo de $B$ até esvaziá-la reempilhando-se estes valores em $A$. Num passo final deve-se desempilhar as chaves em $A$ utilizando-as para *repovoar* $M$ que ficará ordenado.

14. Determine a complexidade do algoritmo descrito na questão-13.

15. Um *palíndromo* é uma string cuja ordem reversa de caracteres revela a própria string de base. Por exemplo, `ana` e `radar` são exemplos de palíndromos. Utilizando uma pilha construa uma função que verifique se uma string de entrada é ou não um palíndromo.

16. Expressões que utilizam parênteses podem ser acidentalmente escritas erroneamente. Os exemplos a seguir mostram sequencias de parênteses inválidas: (()(()), ((()),()())(. Utilizando pilhas construa uma função que teste se a parentisação de uma dada expressão (que pode possuir outros elementos além de parênteses) está correta.

17. Seja $\mathcal{P}$ uma pilha e $U_n$ o vetor formado pelos $n$ primeiros naturais não nulos. $U_n$ é varrido da esquerda para a direita e suas chaves são empilhadas em $\mathcal{P}$ (operação $I$) havendo ainda intercalações com operações de desempilhamento (operações $R$) cujos valores são impressos nessa ordem. Por exemplo, tomando $U_3 = \{1, 2, 3\}$ e executando a sequência de operações $IIRIRR$ então $\mathcal{P}$ se esvazia e os valores impressos são $2, 3, 1$ que é uma permutação das chaves de $U_3$. Determine sequências de operações que gerem permutações válidas nos casos de $U_4$ e $U_5$.

18. Utilizando pilhas construa uma versão iterativa para o algoritmo de ordenação por fusão (MergeSort).

*Utilizando pilhas implemente versões iterativas para as funções a seguir,*

19.
```
Função FNC1(n)
    Se n > 3 então
        FNC1(n/3)
        Escreva n
        FNC1(n/3)
```

20.
```
Função FNC2(n)
    Se n > 3 então FNC2(n/3)
    Escreva n
    Se n > 5 então FNC2(n/5)
```

21.
```
Função FNC3(ref M, p, q)
    Se p < q então
        Retorne 0
```

**senão Se** $p = q$ **então**
  **Retorne** $M[p]$
**senão**
  $r \leftarrow \lfloor p + q \rfloor / 2$
  $x \leftarrow \text{FNC3}(M, p, r - 1)$
  $y \leftarrow \text{FNC3}(M, r + 1, q)$
  **Retorne** $M[r] + x + y$