# ED – esquema para tutorial 03

## Capítulo 3 – Pilhas e Filas

**[página de abertura]**

Ao concluir este capítulo, você deverá ser capaz de:

- compreender o que são as pilhas e filas
- conhecer as operações básicas que se podem efetuar com as pilhas e com as filas
- tomar contato com algumas aplicações de pilhas e de filas

## 4.1 Introdução

Para certas aplicações, tornam-se úteis estruturas de dados constituídas por conjuntos de elementos organizados, não pelos valores dos dados, mas em função de uma determinado critério que regulamenta a entrada e a saída dos dados na estrutura.

Os critérios mais usados para regular a entrada e a saída dos dados são:

- lifo – ***l**ast **i**n, **f**irst **o**ut* – dentre os elementos presentes na estrutura, o primeiro a sair dela será o último que nela entrou (como acontece com uma pilha de pratos: sempre se retira o de cima, que foi o último a entrar na pilha, caso contrário pode-se provocar um acidente).

- fifo – ***f**irst **i**n, **f**irst **o**ut* – dentre os elementos presentes na estrutura, o primeiro a sair dela será o primeiro que nela entrou (como acontece com uma fila de pessoas em um guichê: será atendida em primeiro lugar aquela que está há mais tempo na fila) **{repetir aqui a figura da fila de pessoas, do módulo de introdução}**

O critério LIFO dá origem à estrutura de dados denominada *Pilha*, e o critério FIFO dá origem à estrutura *Fila*.

As operações básicas que se podem realizar sobre uma pilha são:

- inicializar a pilha
- verificar se a pilha está vazia
- retornar o elemento que está no topo da pilha
- inserir um elemento na pilha
- retirar um elemento da pilha

As operações básicas que se podem realizar sobre uma fila são:

- inicializar a fila
- verificar se a fila está vazia
- retornar o elemento que está na primeira posição da fila
- inserir um elemento na fila
- retirar um elemento da fila

## 4.2 Implementação

Como já foi dito, as estruturas de dados são **abstratas**, e portanto podem ser implementadas de diversos modos.

[As estruturas abstratas devem ser imaginadas como uma forma de organizar e relacionar dados na memória do sistema, de modo a permitir certas operações sobre esses dados, independentemente do modo como essas estruturas são implementadas. ]

Em nosso caso, faremos a implementação das pilhas e das filas através de listas abertas com encadeamento simples. O tipo dos elementos será, portanto, o mesmo para os dois casos, ficando a diferença apenas por conta das operações que se realizam em cada caso. É importante observar que, uma vez implementada a lista que representa uma pilha ou uma fila, o usuário da estrutura (pilha ou fila) apenas tem acesso às operações pré-definidas, não podendo executar outras operações que seriam possíveis em listas. Ou seja, o usuário da pilha ou da fila não precisa saber como a estrutura foi implementada, apenas precisa saber utilizar as funções que permitem as operações básicas.

Para exemplificar a implementação das operações básicas utilizaremos as estruturas abaixo, nas quais o dado que se armazena é um caractere:

```
struct tipopilha {
  char dado;
  struct tipopilha *prox;
} ;

struct tipofila {
  char dado;
  struct tipofila *prox;
} ;
```

### 4.2.1 Operações Básicas em Pilhas

a) Inicializar a Pilha

Esta função consiste apenas em inicializar a pilha, ou seja, definir uma pilha vazia, que fica apta a ser utilizada. Recebe como parâmetro o endereço para o ponteiro *pilha***,** que é um ponteiro que dá acesso à pilha.

```
void InicializaPilha (struct tipopilha **pilha) {
  *pilha = NULL;
  return ;
}
```

**Tira Teima**

b) Verificar se a Pilha está vazia

Esta função recebe como parâmetro o ponteiro *pilha*, que dá acesso à pilha, e retorna **1** se a pilha estiver vazia, e **0** em caso contrário.

```
int PilhaVazia (struct tipopilha *pilha) {
  if (pilha == NULL)
```

```
   return 1;
  else
   return 0;
}
```

c) Retornar o elemento que está no topo da pilha

Esta função recebe como parâmetro o ponteiro *pilha*, que dá acesso à pilha, e retorna o elemento que estiver no topo, sem retirá-lo da pilha. Esta operação só pode ser usada quando se tem certeza de que a pilha não é vazia, o que se pode testar com a função *PilhaVazia*.



```
char TopoPilha(struct tipopilha *pilha) {
  return pilha->dado;
}
```

d) Inserir um elemento na Pilha

Esta função recebe como parâmetros o valor *dadonovo*, a ser inserido, e o endereço do ponteiro *pilha*, que dá acesso à pilha. O resultado da operação é a inserção do *dadonovo* no topo da pilha.



```
void InserePilha(struct tipopilha **pilha, char dadonovo) {
  struct tipopilha *p1;
  p1 = malloc (sizeof (struct tipopilha));
  p1->dado = dadonovo;
  p1->prox = *pilha;
  *pilha = p1;
  return;
}
```

e) Retirar um elemento da Pilha

Esta função recebe como parâmetro o endereço do ponteiro *pilha*, que dá acesso à pilha, e retorna o elemento que está no topo da pilha, retirando-o dela. Esta operação só pode ser usada quando se tem certeza de que a pilha não é vazia, o que se pode testar com a função *PilhaVazia*.



```
char RetiraPilha(struct tipopilha **pilha) {
  struct tipopilha *p1;
  char car;
  p1 = *pilha;
  *pilha = p1->prox;
  car = p1->dado;
  free (p1);
  return car;
}
```

### 4.2.2 Operações Básicas em Filas

a) Inicializar a Fila

Esta função consiste apenas em inicializar a fila, ou seja, definir uma fila vazia, que fica apta a ser utilizada. Recebe como parâmetro o endereço para o ponteiro *fila***,** que é um ponteiro que dá acesso à fila.

```
void InicializaFila (struct tipofila **fila) {
  *fila = NULL;
  return;
}
```

**Tira Teima**

b) Verificar se a Fila está vazia

Esta função recebe como parâmetro o ponteiro *fila*, que dá acesso à fila, e retorna **1** se a fila estiver vazia, e **0** em caso contrário.

```
int FilaVazia (struct tipofila *fila) {
  if (fila == NULL)
    return 1;
  else
    return 0;
}
```

**Tira Teima**

c) Retornar o elemento que está na primeira posição da fila

Esta função recebe como parâmetro o ponteiro *fila*, que dá acesso à fila, e retorna o elemento que estiver na primeira posição, sem retirá-lo da fila. Esta operação só pode ser usada quando se tem certeza de que a fila não é vazia, o que se pode testar com a função *FilaVazia*.

```
char FrenteFila(struct tipofila *fila) {
  return fila->dado;
}
```

**Tira Teima**

d) Inserir um elemento na Fila

Esta função recebe como parâmetros o valor *dadonovo*, a ser inserido, e o endereço do ponteiro *fila*, que dá acesso à fila. O resultado da operação é a inserção do *dadonovo* no fim da fila.

```
void InsereFila(struct tipofila **fila, char dadonovo) {
  struct tipofila *f1, *f2;
  f1 = malloc (sizeof (struct tipofila));
  f1->dado = dadonovo;
  f1->prox = NULL;
  if (*fila == NULL)
    *fila = f1;
  else {
    f2 = *fila;
    while (f2->prox != NULL)
      f2 = f2->prox;
    f2->prox = f1;
  }
  return;
}
```

**Tira Teima**

e) Retirar um elemento da Fila

Esta função recebe como parâmetro o endereço do ponteiro *fila*, que dá acesso à fila, e retorna o elemento que está na primeira posição da fila, retirando-o dela. Esta operação só pode ser usada quando se tem certeza de que a fila não é vazia, o que se pode testar com a função *FilaVazia*.

```
char RetiraFila(struct tipofila **fila) {
  struct tipofila *f1;
  char car;
  f1 = *fila;
  *fila = f1->prox;
  car = f1->dado;
  free (f1);
  return car;
}
```

**Tira Teima**

## 4.3 Aplicações

Serão apresentadas agora algumas aplicações de pilhas e de filas. A partir deste ponto, as pilhas e as filas, por serem estruturas abstratas, serão manipuladas sempre através das operações vistas anteriormente (InsereFila, RetiraPilha, etc...) Desta forma o usuário não precisa saber qual foi a implementação utilizada em cada caso, e não tem outro acesso às estruturas utilizadas, que não sejam as operações pré-definidas.

Veremos as seguintes aplicações:

- manipulação de uma seqüência de caracteres

- avaliação de expressão totalmente parentetizada

- avaliação de expressão na forma pós-fixada

- conversão de expressão in-fixada para pós-fixada

- simulação de recursividade

- pilhas ou listas de uso geral

### 4.3.1 Manipulação de uma seqüência de caracteres

É dada uma seqüência de caracteres formada por letras e algarismos alternados, começando por uma letra. A função deve construir outra seqüência que contenha todas as letras da seqüência dada, na mesma ordem e posições originais, e todos os algarismos da seqüência dada, nas posições originais, mas em ordem invertida. Como exemplo são dadas abaixo uma seqüência de entrada e outra de saída correspondente:

Entrada: A 5 F 8 B 4 C 9 D 7
Saída: A 7 F 9 B 4 C 8 D 5

A função ManipulaCaracteres supõe que tenham sido definidos anteriormente os tipos **tipofila** e **tipopilha**. A seqüência original é lida do arquivo *arq* e a seqüência de saída é escrita na tela.

```
[struct tipopilha {
  char dado;
  struct tipopilha *prox;
};]

[struct tipofila {
  char dado;
  struct tipofila *prox;
};]
```

**Tira Teima**

```
void ManipulaCaracteres () {
  struct tipofila *f1;
  struct tipopilha *p1;
  char x;
  FILE *arq;

  InicializaFila (&f1);
```

```
  InicializaPilha (&p1);
  arq = fopen ("t2.txt", "r");
  while (!feof(arq)){
    x = getc(arq);
    if ((x!='\n')&& (x!=EOF))
      InsereFila (&f1, x);
    x = getc(arq);
    if ((x!='\n')&& (x!=EOF))
      InserePilha (&p1, x);
  }
  fclose (arq);
  while ((!FilaVazia(f1)) && (!PilhaVazia(p1))) {
    printf ("%c", RetiraFila (&f1));
    printf ("%c", RetiraPilha(&p1));
  }
}
```

### 4.3.2 Avaliação de expressão totalmente parentetizada

É dada uma expressão numérica em que todas as operações são cercadas por parêntesis, independentemente de sua prioridade, como por exemplo: ( 2 * ( 5 - 1 ) )

Supõem-se definidos anteriormente os tipos **tipopilha** e **tipopilha2**. A expressão é lida do arquivo *arq*. A função *AvaliaExpressaoParentetizada* utiliza a função *ResolveOperacao*, que retira da pilha de operadores o último operador e, em seguida, realiza a operação desejada e volta a inserir o resultado na pilha de números.

```
void AvaliaExpressaoParentetizada () {
  struct tipopilha *p1, *p2;
  char car;
  int i;
  FILE *arq;

//------------
void ResolveOperacao () {

  char op, c1, c2, caux;
  int x,y;

  if (PilhaVazia (p2)) {
    if (PilhaVazia (p1)) {
      caux = 0+'0';
      InserePilha (&p1,0+'0');
    }
  }
  else {
    op = RetiraPilha (&p2);
    c1 = RetiraPilha (&p1);
    x = c1 - '0';
    c2 = RetiraPilha (&p1);
    y = c2 - '0';
    switch (op) {
      case '+' : InserePilha (&p1, y+x+'0'); break;
      case '-' : InserePilha (&p1, y-x+'0'); break;
      case '*' : InserePilha (&p1, y*x+'0'); break;
      case '/' : InserePilha (&p1, y%x+'0'); break;
    }
  }
}
//------------

  InicializaPilha (&p1);
  InicializaPilha (&p2);
  arq = fopen ("t3.txt", "r");
  while ((car = getc (arq)) != EOF) {
    if (car != '\n')
      if ((car>='0')&&(car<='9'))
        InserePilha (&p1, car);
      else
        if (car == ')')
```

**Tira Teima**

```
        ResolveOperacao ();
      else
        if (car != '(')
          InserePilha (&p2, car);
  }
  fclose (arq);
  printf("%d \n", RetiraPilha (&p1)-'0');
}
```

### 4.3.3 Avaliação de expressão na forma pós-fixada

A função deve fazer a avaliação de uma expressão dada em forma **pós-fixada**. A expressão se encontra no arquivo *arq*. A função supõe que tenha sido definido o tipo **tipopilha**. A função *AvaliaExpressaoPosfixada* utiliza a função *ResolveOperacao*, que executa a operação desejada entre dois operandos e retorna o resultado.

[ Uma expressão algébrica é dita em forma pós-fixada, ou notação polonesa, quando o operador é colocado depois dos dois operandos. Vejamos alguns exemplos:

| notação in-fixada | notação pós-fixada |
|---|---|
| A + B | A B + |
| (A + B) * C | A B + C * |
| (A * B + C) / D | A B * C + D / |

Uma das vantagens da notação polonesa é prescindir do uso de parênteses. ]

```
void AvaliaExpressaoPosFixada () {
  struct tipopilha *p1;
  char c, c1, c2;
  FILE *arq;
  int w;

//------------
int ResolveOperacao (char c1, char c, char c2) {

  int x, y, z;

  x = c1 - '0';
  y = c2 - '0';
  switch (c) {
    case '+' : z = x+y; break;
    case '-' : z = x-y; break;
    case '*' : z = x*y; break;
    case '/' : z = x/y; break;
  }
  return z;
}
//------------

  InicializaPilha (&p1);
  arq = fopen ("t6.txt", "r");
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      if ((c!='+')&&(c!='-')&&(c!='*')&&(c!='/'))
        InserePilha (&p1, c);
      else {
        c2 = RetiraPilha (&p1);
        c1 = RetiraPilha (&p1);
        w = ResolveOperacao (c1, c, c2);
        InserePilha (&p1, w+'0');
      }
    }
  }
  printf ("resultado: %d \n", RetiraPilha (&p1)-'0');
  fclose (arq);
}
```

**Tira Teima**

### 4.3.4 Conversão de expressão in-fixada para pós-fixada

A função *ConversaoParaPosFixada* lê de um arquivo *arq1* uma expressão algébrica em forma **in-fixada**, e escreve na tela a mesma expressão, na forma **pós-fixada**. A função utiliza os tipos **tipopilha** e **tipofila**. A fila *f1* armazena os operandos e a pilha *p1* armazena os operadores e o sinal '('. A função *Prioridade* retorna a **prioridade** de execução de uma operação. A variável *aux* serve apenas para eliminar o caractere '(' quando este for encontrado no topo da pilha *p1*.

[tabela de prioridades

1. exponenciação
2. multiplicação e divisão
3. soma e subtração
4. sinal '(' ]

```
void ConversaoParaPosFixada () {
  struct tipopilha *p1;
  struct tipofila *f1;
  char c, aux;
  FILE *arq;

//------------
int Prioridade (char c1) {

  switch (c1) {
    case '^' : return 1; break;
    case '*' : case '/' : return 2; break;
    case '+' : case '-' : return 3; break;
    case '(' : return 4; break;
  }
}
//------------

  InicializaPilha (&p1);
  InicializaFila (&f1);
  arq = fopen ("t8.txt", "r");
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      if ((c!='^')&&(c!='+')&&(c!='-')&&(c!='*')&&(c!='/')&&(c!='(')&&(c!=')'))
        InsereFila (&f1, c);
      else
        if (c==')') {
          while (TopoPilha(p1) != '(')
            InsereFila (&f1, RetiraPilha (&p1));
          aux = RetiraPilha (&p1);
        }
        else {
          if ((c!='(') && (!PilhaVazia (p1))){
          while ((Prioridade (TopoPilha (p1)) <= Prioridade (c))&&(!PilhaVazia (p1)))
              InsereFila (&f1, RetiraPilha (&p1));
}          InserePilha (&p1,c);
        }
    }
  }
  fclose (arq);
  while (!PilhaVazia (p1))
    InsereFila (&f1, RetiraPilha (&p1));
  while (!FilaVazia (f1))
    printf ("%c", RetiraFila (&f1));
}
```

**Tira Teima**

### 4.3.5 Simulação de recursividade

As pilhas podem ser utilizadas para simular a recursividade na resolução de certos problemas. É útil entender como as pilhas permitem a recursividade, pois é através delas que as linguagens implementam a recursividade.

Como exemplo tomemos a geração da seqüência de Fibonacci.

A seqüência de Fibonacci pode ser definida do seguinte modo:

fib (0) = 1
fib (1) = 1
fib (n) = fib (n-1) + fib (n-2) , para n > 1

A seqüência gerada de acordo com esta definição é a seguinte:

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|---|
| fib | 1 | 1 | 2 | 3 | 5 | 8 | 13 | ... |

A função abaixo calcula o n-ésimo termo, que fica armazenado na variável *fib*. Supõe-se definido o tipo **tipopilha**.

```
int Fibonacci (int n) {
  struct tipopilha *p1;
  int fib, x;

  InicializaPilha (&p1);
  InserePilha (&p1, n);
  fib = 0;
  while (!PilhaVazia (p1)) {
    x = RetiraPilha (&p1);
    if ((x == 0) || (x == 1))
      fib = fib + 1;
    else {
      InserePilha (&p1, x-1);
      InserePilha (&p1, x-2);
    }
  }
  return fib;
}
```

**Tira Teima**

### 4.3.6 Pilhas ou listas de uso geral

Em todas as aplicações vistas até aqui, sempre se supõe definidos os tipos **tipopilha** e **tipofila**, que são pilhas ou filas contendo caracteres. É interessante a criação de pilhas ou filas de uso geral, nas quais é possível armazenar qualquer tipo de dado. Isto é possível na linguagem C, se utilizarmos um ponteiro apontando para *void* como informação contida na pilha ou na fila. Este ponteiro é de uso genérico, podendo apontar para qualquer tipo de estrutura.

Como exemplo de aplicação de uma pilha de uso geral, vamos resolver o problema das **Torres de Hanói**, simulando a recursividade através de uma pilha de uso geral.

[ O problema das Torres de Hanói é um exemplo clássico de uso de recursividade. São dados três pinos **A, B,** e **C**, e um conjunto de discos furados, de diferentes tamanhos, que se encaixam nesses pinos. Os discos estão inicialmente no pino **A** e devem ser transferidos para o disco **B**, utilizando o pino **C** como auxiliar. Tanto na posição inicial como em qualquer posição intermediária, os discos devem estar dispostos de modo que nunca um disco maior fique sobre um disco menor. Os movimentos devem ser feitos com um disco de cada vez. ]

**{ FIGURA DE TRÊS PINOS COM 3 DISCOS }**

Para resolver o problema de forma recursiva, deve-se raciocinar da seguinte forma: para levar **n** discos de **A** para **B**, basta levar os **n – 1** discos superiores de **A** para **C**, em seguida levar o disco maior de **A** para **B**, e depois levar os **n – 1** discos de **C** para **B**. No entanto, a operação de transferir **n – 1** discos não é permitida pela própria regra do problema, e deve então ser abordada como uma chama da recursiva ao mesmo problema, mas com o grau de dificuldade reduzido de **n** para **n – 1**.

Definamos, então, o movimento de **n** discos, do pino **A** para o pino **B,** utilizando o pino **C** como auxiliar, através da função ***Hanoi* (A,B,C,n)**. Para todo n > 1, esta função pode ser decomposta em outras três:

***Hanoi* (A,C,B,n-1)**
***Hanoi* (A,B,C, 1 )**
***Hanoi* (C,B,A,n-1)**

As três chamadas são recursivas, mas a primeira e a terceira diminuem o número de discos, de **n** para **n-1,** enquanto a segunda constitui um caso trivial, em que o número de discos é 1, ou seja, neste caso o movimento do disco é efetuado.

Definamos um tipo registro que contenha os parâmetros de *Hanoi*, do seguinte modo:

```
struct reghanoi {
   char o;
   char d;
   char a;
   int k; };
```

Definamos todas as operações de manipulação de uma pilha cujo tipo denominaremos *tipopilhageral*, que é uma pilha cujo elemento constituinte é um ponteiro que aponta para *void,* do seguinte modo:

```
struct tipopilhageral {
  void *dado;
  struct tipopilhageral *prox;
} ;
```

A função *Hanoi* pode então ser escrita, com a ajuda das funções *EmpilhaHanoi* e *DesempilhaHanoi***,** que servem para criar ou eliminar os registros apontados pelos ponteiros da pilha de tipo *tipopilhageral***.**

**Tira Teima**

```
//-------------------------------------------------
void Hanoi (char origem, char destino, char auxiliar, int n) {
  struct tipopilhageral *phanoi;
  struct reghanoi {
     char o;
     char d;
     char a;
     int k; };

//-------------------------------------------------
void *EmpilhaHanoi (struct tipopilhageral *pilha, char origem, char destino, char auxiliar, int n) {
  struct reghanoi *paux1;
  void *paux2;
  paux1 = malloc (sizeof (struct reghanoi));
  paux1->o = origem;
  paux1->d = destino;
  paux1->a = auxiliar;
  paux1->k = n;
  paux2 = paux1;
  InserePilha (&pilha, paux2);
  return pilha;
}

//-------------------------------------------------
void *DesempilhaHanoi (struct tipopilhageral *pilha, char *origem, char *destino, char *auxiliar, int *n) {
  struct reghanoi *paux1;
  void *paux2;
  paux2 = RetiraPilha (&pilha);
  paux1 = paux2;
  *origem = paux1->o;
  *destino = paux1->d;
  *auxiliar = paux1->a;
  *n = paux1->k;
  free (paux2);
  return pilha;
}

//-------------------------------------------------
  InicializaPilha (&phanoi);
  phanoi = EmpilhaHanoi (phanoi, origem, destino, auxiliar, n);
  while (!PilhaVazia (phanoi)) {
    phanoi = DesempilhaHanoi (phanoi, &origem, &destino, &auxiliar, &n);
    if (n == 1)
      printf ("movimento de %3c para %3c \n", origem, destino);
    else {
      phanoi = EmpilhaHanoi (phanoi, auxiliar, destino, origem, n-1);
      phanoi = EmpilhaHanoi (phanoi, origem, destino, auxiliar, 1);
      phanoi = EmpilhaHanoi (phanoi, origem, auxiliar, destino, n-1);
    }
  }
}
```

**[página de conclusão]**

O estudo das pilhas e filas fornece ao aluno um bom instrumental para resolver diversos problemas computacionais, em especial aqueles que envolvem recursividade. Além disso, seu conhecimento é fundamental para auxiliar na solução de alguns problemas que utilizam estruturas que serão vistas nos próximos tópicos.