1 - Conceito de Pilha

Pilha (*stack* em inglês) é uma estrutura de dados dinâmica para manter elementos no formato de uma pilha, onde

o último elemento que entra é o 1º que sai.

Desta forma, ela é considerada uma estrutura do tipo LIFO (Last-In First-Out). Em outras palavras, o elemento removido é o que está na estrutura há menos tempo.

Uma pilha pode ser implementa usando arrays ou listas encadeadas.

2 - Implementação de Pilha usando Array

Um array é uma estrutura imutável, ou seja, ele não pode ser redimensionado para comportar novos elementos, mas uma pilha pode comportar qualquer quantidade de elementos. Então para implementar uma pilha usando array temos de estabelecer a quantidade máxima de elementos da pilha, que será a quantidade de elementos do array.

Na estrutura da pilha, elementos podem ser inseridos (no final/topo) e removidos (também do final/topo) a qualquer momento. A última posição do array (pilha) é chamado de topo da pilha.

A Figura 1 mostra uma implementação de pilha usando array. Para ter uma pilha usando array é necessário ter:

- Um array para manter os dados;
- Uma variável t que indica a 1ª posição livre do vetor, ou seja, o topo da pilha está na posição t-1 do array.

Uma pilha deve ter as seguintes operações:

- Criar a pilha: deve criar o array e zerar o t da pilha;
- Checar se a pilha está vazia: a pilha está vazia quando o t é zero;
- Checar se a pilha está cheia: a pilha está cheia se t possui o mesmo número de elementos do array;
- Limpar a pilha: para liberar a pilha sem alterar o array basta zerar o t;
- Empilhar (push) um novo elemento na pilha: adiciona o elemento na posição t e move o índice t para a próxima posição;
- Desempilhar (pop) um elemento da pilha: remove o elemento que está no topo da pilha, ou seja, na posição t-1 e move o índice t para a posição anterior.

A classe Pilha da Figura 1 contempla ainda um método para imprimir os elementos da pilha, mas uma operação de imprimir não faz parte das operações padrões de uma pilha.

A Figura 2 possui um código de teste da classe da Figura 1, os comentários mostram a situação da pilha. A situação do array pode também ser visualizada na representação da Figura 3, os elementos com fundo amarelo estão em uso e os demais não.

```
public class Pilha {
   private int[] vetor;
   private int t;

   public Pilha(){
      vetor = new int[5];
      t = 0; /* 1a posição livre do vetor */
   }

   /* informa se a pilha está vazia */
   public boolean isVazia(){
      return t == 0;
   }
```

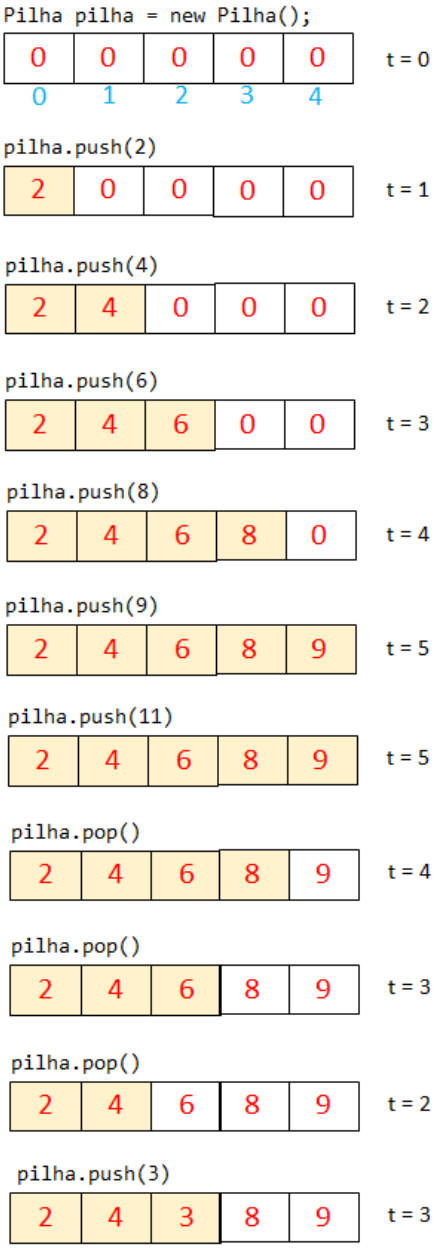
```
public class Principal {
   public static void main(String[] args) {
      Pilha pilha = new Pilha();
      pilha.imprimir();  //Pilha vazia
      pilha.push(2);
      pilha.imprimir();  //2
      pilha.push(4);
      pilha.imprimir();  //2 4
      pilha.push(6);
      pilha.imprimir();  //2 4 6
      pilha.push(8);
      pilha.imprimir();  //2 4 6 8
      pilha.push(9);
```

```
    /* informa se a pilha está cheia */
    public boolean isCheia(){
       return t == vetor.length;
    }

    /* coloca o elemento no topo da pilha */
    public void push(int nro){
       if( isCheia() ){
          System.out.println("Pilha cheia - stack overflow");
       }
       else{
          /* coloca o nro na 1a posição livre do vetor */
          vetor[t] = nro;
          t++; /* atualiza a 1a posição livre */
       }
    }

    /* retira o elemento do topo da pilha */
    public int pop(){
       if( isVazia() ){
          System.out.println("Pilha vazia");
          return -1;
       }
       else{
          /* atualiza a 1a posição livre */
          return vetor[--t];
       }
    }

    /* remove todos os elementos da pilha */
    public void limpar(){
       t = 0; /* atualiza a 1a posição livre */
    }

    public void imprimir(){
       if( isVazia() ){
          System.out.print("Pilha vazia");
       }
       else{
          for( int i = 0; i < t; i++ ){
             System.out.print( vetor[i] +" ");
          }
       }
       System.out.println();
    }
}
```

Figura 1 – Código da classe Pilha usando array.

```
        pilha.imprimir();  //2 4 6 8 9
        pilha.push(11);    //Pilha cheia
        pilha.imprimir();  //2 4 6 8 9
        pilha.pop();
        pilha.imprimir();  //2 4 6 8
        pilha.pop();
        pilha.imprimir();  //2 4 6
        pilha.pop();
        pilha.imprimir();  //2 4
        pilha.push(3);
        pilha.imprimir();  //2 4 3
        pilha.pop();
        pilha.imprimir();  //2 4
        pilha.pop();
        pilha.imprimir();  //2
        pilha.pop();
        pilha.imprimir();  //Pilha vazia
    }
}
```

Figura 2 – Código da classe Principal para testar a classe Pilha da Figura 1.



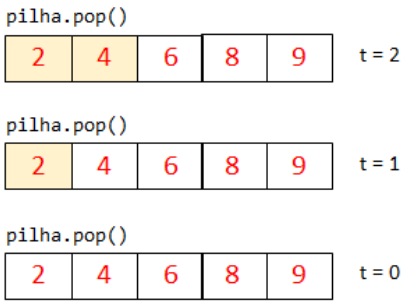


Figura 3 – Representação do código da Figura 2 no array do objeto Pilha.

3 - Implementação de Pilha usando Lista Encadeada

Ao contrário do array que possui um tamanho fixo, a lista encadeada é uma estrutura dinâmica que inicia vazia e pode comportar qualquer quantidade de elementos. A implementação tradicional de lista encadeada mantém apenas uma referência para o nó de início (cabeça da lista) e os elementos são inseridos à direita do início. Porém, para implementar uma pilha os elementos terão de ser inseridos à esquerda do nó de início e deslocar o início para à esquerda, pois o último a entrar deverá ser o 1º a sair, então ele precisará estar no "início" da lista.

Para ter uma pilha usando lista encadeada é necessário ter:

- Uma referência para o 1º elemento da lista, que será o topo da pilha.

Uma pilha deve ter as seguintes operações:

- Criar a pilha: deve iniciar o atributo de topo da pilha;
- Checar se a pilha está vazia: a pilha está vazia quando o topo é nulo;
- Limpar a pilha: para liberar a lista basta setar nulo no atributo topo. No caso do Java, os nós (objetos do tipo No) não referenciados serão liberados pelo Garbage Collector;
- Empilhar (push) um novo elemento na pilha: adiciona o elemento antes do topo e move o índice topo para o nó inserido;
- Desempilhar (pop) um elemento da pilha: remove o elemento que está no topo da pilha, ou seja, desloca o topo para topo.proximo.

Veja que não é necessário a operação para verificar se a pilha está cheia, já que a lista encadeada não possui limite.

A classe Pilha da Figura 4 mostra a implementação de uma pilha usando lista encadeada. Como a pilha utiliza uma lista então é necessário que cada elemento seja um objeto do tipo No (Figura 5).

A Figura 6 possui um código de teste da classe da Figura 4, os comentários mostram a situação da pilha. Veja que o resultado é difere daquele da Figura 2 apenas ao aceitar o valor 11, já que a lista não fica cheia.

```
public class Pilha {
   private No topo; /* aponta para o topo da pilha */

   public Pilha(){
      topo = null;
   }

   /* informa se a pilha está vazia */
   public boolean isVazia(){
      return topo == null;
   }
```

```
public class No {
  int conteudo;
  No proximo;
}
```

Figura 5 – Código da classe No usada em cada elemento da Pilha da Figura 4.

```
   /* coloca o elemento no topo da pilha */
   public void push(int nro){
      No no = new No(); /* criar um nó */
      no.conteudo = nro;
      no.proximo = topo;
      topo = no; /* o no será o último a entrar */
   }

   /* retira o elemento do topo da pilha */
   public int pop(){
      if( isVazia() ){
         System.out.println("Pilha vazia");
         return -1;
      }
      else{
         int nro = topo.conteudo;
         topo = topo.proximo;
         return nro;
      }
   }

   /* remove todos os elementos da pilha */
   public void limpar(){
      topo = null;
   }

   public void imprimir(){
      /* checa se a lista está vazia */
      if( isVazia() ){
         System.out.println("Pilha vazia");
      }
      else{
         /* percorrer a lista até encontrar o último nó */
         No no = topo;
         while( no != null ){
            System.out.print( no.conteudo +" ");
            no = no.proximo;
         }
         System.out.println(); /* quebra de linha na tela */
      }
   }
}
```

Figura 4 – Código da classe Pilha usando lista encadeada.

```
public class Principal {
   public static void main(String[] args) {
      Pilha pilha = new Pilha();
      pilha.imprimir();  //Pilha vazia
      pilha.push(2);
      pilha.imprimir();  //2
      pilha.push(4);
      pilha.imprimir();  //4 2
      pilha.push(6);
      pilha.imprimir();  //6 4 2
      pilha.push(8);
      pilha.imprimir();  //8 6 4 2
      pilha.push(9);
      pilha.imprimir();  //9 8 6 4 2
      pilha.push(11);
      pilha.imprimir();  //11 9 8 6 4 2
      pilha.pop();
      pilha.imprimir();  //9 8 6 4 2
      pilha.pop();
      pilha.imprimir();  //8 6 4 2
      pilha.pop();
      pilha.imprimir();  //6 4 2
      pilha.push(3);
      pilha.imprimir();  //3 6 4 2
      pilha.pop();
      pilha.imprimir();  //6 4 2
      pilha.pop();
      pilha.imprimir();  //4 2
      pilha.pop();
      pilha.imprimir();  //2
      pilha.pop();
      pilha.imprimir();  //Pilha vazia
   }
}
```

Figura 6 – Código da classe Principal para testar a classe Pilha da Figura 4.

4 - Aplicações de Pilha

Exemplo 1 – Considera-se que uma expressão aritmética está balanceada (bem formada) se os parênteses, colchetes e chaves são fechados na ordem inversa à qual foram abertos.

Usando uma estrutura de pilha é possível resolver o problema. Tome como exemplo o código da Figura 9, ele produz o resultado mostrado na Figura 10.

Para simplificar a codificação da classe Pilha foi usada uma lista encadeada assim como mostra a Figura 7, como o tipo de dado é String, então foi necessário a classe No da Figura 8.

```
public class Pilha {
   private No topo;

   public Pilha(){
      topo = null;
   }

   /* informa se a pilha está vazia */
   public boolean isVazia(){
      return topo == null;
   }
```

```
public class Principal {
   public static void main(String[] args) {
      String[] exp = {
            "( () [ () ] )",
            "( () [ ( ] ) )",
            "8+{3-[(3-2)*5]",
            "8+{3-[(3-2)*5]}"
      };
      for( int i = 0; i < exp.length; i++ ){
         if( isBalanceada(exp[i]) ){
            System.out.println( exp[i] +" está bem formada");
         }
```

```
    /* coloca o elemento no topo da pilha */
    public void push(String c){
       No no = new No();
       no.conteudo = c;
       no.proximo = topo;
       topo = no;
    }

    /* retira o elemento do topo da pilha */
    public String pop(){
       if( isVazia() ){
          return "#";
       }
       else{
          String c = topo.conteudo;
          topo = topo.proximo;
          return c;
       }
    }

    /* remove todos os elementos da pilha */
    public void limpar(){
       topo = null;
    }

    public void imprimir(){
       if( isVazia() ){
          System.out.println("Pilha vazia");
       }
       else{
          No no = topo;
          while( no != null ){
            System.out.print(no.conteudo +" ");
            no = no.proximo;
          }
          System.out.println();
       }
    }
}
```

Figura 7 – Código da classe Pilha usando lista encadeada.

```
public class No {
  String conteudo;
  No proximo;
}
```

Figura 8 – Código da classe No usada em cada elemento da Pilha da Figura 7.

```
          else{
           System.out.println( exp[i] +"
                               não está bem formada");
          }
       }
    }

    /* retorna true se os parênteses, colchetes e chaves estão
     * abrindo e fechando na ordem correta */
    public static boolean isBalanceada(String exp){
       Pilha pilha = new Pilha();

       String[] v = exp.split("");
       for( int i = 0; i < v.length; i++ ){
          switch( v[i] ){
             case "(":
             case "[":
             case "{":
                pilha.push(v[i]); break;
             case ")":
                if( !pilha.pop().equals("(") ) return false;
                break;
             case "]":
                if( !pilha.pop().equals("[") ) return false;
                break;
             case "}":
                if( !pilha.pop().equals("{") ) return false;
                break;
          }
       }

       return pilha.isVazia();
    }
}
```

Figura 9 – Código da classe Principal para testar se uma expressão está balanceada.

```
( () [ () ] ) está bem formada
( () [ ( ] ) ) não está bem formada
8+{3-[(3-2)*5] não está bem formada
8+{3-[(3-2)*5]} está bem formada
```

Figura 10 – Resultado do teste da Figura 9.

Exemplo 2 – Na notação usual de expressões aritméticas, os operadores são escritos entre os operandos, desta forma, a notação é chamada de infixa. Na notação posfixa, ou Notação Polonesa Inversa (em inglês, Reverse Polish Notation, para mais detalhes https://pt.wikipedia.org/wiki/Notação_polonesa_inversa), os operadores são escritos depois dos operandos. Por exemplo:

Infixa	Posfixa
(a+b)	ab+
((a+b)/c)	ab+c/
(c/(a+b))	cab+/
(((a*b)-(c*d))/(e*f))	ab*cd*-ef*/
(a+b*d-e+d-f*g)	abd*+e-d+fg*-

Na notação posfixa não existem os parênteses, mesmo assim note que os operandos (a, b, c, ...) mantém a ordem.

Na notação posfixa operador (+, -, * e /) opera somente com os dois últimos operandos, por exemplo:

abc*+ é a notação posfixa para a+b*c

A Figura 12 mostra uma implementação da conversão da notação infixa para posfixa usando a classe Pilha da Figura 7. A Figura 12 mostra o resultado do código.

```
public class Principal {
   public static void main(String[] args) {
      /* as expressões precisam estar envolvidas por parênteses */
      String[] exp = {
            "(a+b)",
            "((a+b)/c)",
            "(c/(a+b))",
            "(((a*b)-(c*d))/(e*f))",
            "(a+b*d-e+d-f*g)"
      };
      for( int i = 0; i < exp.length; i++ ){
         System.out.println( infixaToPosfixa(exp[i]) );
      }
   }

   public static String infixaToPosfixa(String exp){
      Pilha pilha = new Pilha(); /* mantém os parênteses e operadores */
      String resultado = "", c;
      String[] v = exp.split("");

      /* pilha.pop() irá retornar o caractere # quando a pilha estiver vazia */
      for( int i = 0; i < v.length; i++ ){
         switch( v[i] ){
            case "(": /* empilha o ( */
               pilha.push(v[i]); break;
            case ")": /* desempilha enquanto o topo da pilha for diferente de ( */
               c = pilha.pop();
               while( !c.equals("(") && !c.equals("#") ){
                  resultado = resultado + c;
                  c = pilha.pop();
               }
               break;
            case "+":
            case "-": /* desempilha enquanto o topo da pilha for diferente de ( */
               c = pilha.pop();
               while( !c.equals("(") && !c.equals("#") ){
                  resultado = resultado + c;
                  c = pilha.pop();
               }
               pilha.push(c);
               pilha.push(v[i]);
               break;
            case "*":
            case "/":
               /* desempilha enquanto o topo da pilha for diferente de (, + e - */
               c = pilha.pop();
               while( !c.equals("(") && !c.equals("+") &&
                      !c.equals("-") && !c.equals("#") ){
                  resultado = resultado + c;
                  c = pilha.pop();
               }
               pilha.push(c);
               pilha.push(v[i]);
               break;
            default: /* empilha o operando */
               resultado = resultado + v[i];
         }
      }
      return resultado;
   }
}
```

Figura 11 – Código da classe Principal para testar a conversão da notação infixa para a posfixa.

```
ab+
ab+c/
cab+/
ab*cd*-ef*/
abd*+e-d+fg*-
```

Figura 12 – Resultado do código da Figura 11.

A Figura 13 mostra os valores das principais variáveis do método infixaToPosfixa durante o processo de conversão da expressão (((a*b)-(c*d))/(e*f)) para ab*cd*-ef*/.

i	v[i]	pilha	resultado
0	(	(	
1	((	((	
2	(((	(((	
3	(((a	(((	a
4	(((a*	*(((	a
5	(((a*b	*(((	ab
6	(((a*b)	((	ab*
7	(((a*b)-	-((	ab*
8	(((a*b)-(	(-((	ab*
9	(((a*b)-(c	(-((	ab*c
10	(((a*b)-(c*	*(-((	ab*c
11	(((a*b)-(c*d	*(-((	ab*cd
12	(((a*b)-(c*d)	-((	ab*cd*
13	(((a*b)-(c*d))	(	ab*cd*-
14	(((a*b)-(c*d))/	/(	ab*cd*-
15	(((a*b)-(c*d))/(	(/(	ab*cd*-
16	(((a*b)-(c*d))/(e	(/(	ab*cd*-e
17	(((a*b)-(c*d))/(e*	*(/(	ab*cd*-e
18	(((a*b)-(c*d))/(e*f	*(/(	ab*cd*-ef
19	(((a*b)-(c*d))/(e*f)	/(	ab*cd*-ef*
20	(((a*b)-(c*d))/(e*f))		ab*cd*-ef*/

Figura 13 – Valores das variáveis do método infixaToPosfixa da Figura 11.

5 - Exercícios

1 – Programar uma classe Pilha onde os elementos são mantidos em um array. O array deverá ser redimensionado quando a pilha estiver cheia.

2 – Programar uma aplicação da classe Pilha para inverter os caracteres de uma string, por exemplo,

Bom dia para aid mob

3 – Programar uma versão do código que converte da notação infixa para posfixa considerando que existem colchetes na expressão aritmética.