

products: storeProducts

```
render() {
  return (
    <React.Fragment>
      <div className="py-5">
        <div className="container">
          <Title name="our" title="Our Products" />
          <div className="row">
            <ProductConsumer>
              {(value) =>
                console.log(value)
              }
            </ProductConsumer>
          </div>
        </div>
      </div>
    </React.Fragment>
  )
}
```

Curso de

# Lógica de Programação



## O que são dados?

Os dados (e seus diversos tipos) são os blocos básicos da programação. Eles representam uma unidade ou um elemento de informação que pode ser acessado através de um identificador - por exemplo, uma variável.

A maior parte das linguagens de programação trabalha com variações baseadas nos quatro tipos primitivos abaixo:

- INT ou número inteiro: valores numéricos inteiros (positivos ou negativos);
- FLOAT ou o chamado “ponto flutuante”: valores numéricos com casas após a vírgula (positivos ou negativos);
- BOOLEAN ou booleanos: representado apenas por dois valores, “verdadeiro” e “falso”. Também chamados de operadores lógicos;
- TEXT: sequências ou cadeias de caracteres, utilizados para manipular textos e/ou outros tipos de dados não numéricos ou booleanos, como hashes de criptografia.

O **JavaScript**, por exemplo, tem como tipos primitivos embutidos na estrutura básica da linguagem: number, string, boolean e symbol (de “nome simbólico”, usado entre outras coisas para criar propriedades únicas em objetos). Já o **C#** (C-Sharp) trabalha com uma quantidade maior de tipos primitivos, de acordo com o espaço de memória que será ocupado pela variável: Boolean, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, IntPtr, UIntPtr, Char, Double e Single. O **C**, por sua vez, não tem um tipo próprio de dado booleano; *false* é representado pelo número 0 e qualquer outro algarismo representa *true*. Outras linguagens podem trabalhar com outras variações.

## O que são estruturas de dados?

Em computação, normalmente utilizamos os dados de forma conjunta. A forma como estes dados serão agregados e organizados depende muito de como serão utilizados e processados, levando-se em consideração, por exemplo, a eficiência para buscas, o volume dos dados trabalhados, a complexidade da implementação e a forma como os dados se relacionam. **Estas diversas formas de organização são as chamadas estruturas de dados.**

*Podemos afirmar que um programa é composto de algoritmos e estruturas de dados, que juntos fazem com que o programa funcione como deve.*

Cada estrutura de dados tem um conjunto de métodos próprios para realizar operações como:

- Inserir ou excluir elementos;
- Buscar e localizar elementos;
- Ordenar (classificar) elementos de acordo com alguma ordem especificada.

## Características das estruturas de dados

As estruturas de dados podem ser:

- lineares (ex. arrays) ou não lineares (ex. grafos);
- homogêneas (todos os dados que compõe a estrutura são do mesmo tipo) ou heterogêneas (podem conter dados de vários tipos);
- estáticas (têm tamanho/capacidade de memória fixa) ou dinâmicas (podem expandir).



Veremos a seguir uma lista e descrição de algumas estruturas:

## Array

Também chamado de *vetor*, *matriz* ou *arranjo*, o array é a mais comum das estruturas de dados e normalmente é a primeira que estudamos.

Um array é uma lista ordenada de valores:

```
const listaNumeros = [4, 6, 2, 77, 1, 0];
```

```
const listaFrutas = ["banana", "maçã", "pera"];
```

Por *ordenada*, entenda-se aqui uma lista onde os valores sempre são acessados na mesma ordem. Ou seja, a não ser que seja utilizada alguma função ou método para alterar a ordem, o primeiro elemento do array `listaNumeros` sempre será 4, e o último, 0.

Normalmente trabalha-se com apenas um tipo de dado por array; embora o JavaScript permita a declaração de arrays de mais de um tipo de dado, por exemplo `["banana", 5, true]`, isso não acontece na maior parte das linguagens de programação.

Por ser uma estrutura de dados básica e muitíssimo utilizada, a linguagens de programação costumam já ter este tipo implementado, com métodos nativos para criação e manipulação de arrays. No caso do JavaScript, você pode consultar os métodos e construtores de array no [MDN](#).

## Usos

Sendo a mais comum das estruturas, arrays são utilizados em praticamente toda situação que envolva organizar dados de um mesmo tipo; sejam dados recebidos por uma API ou enviados a uma base de dados, ou mesmo passado via parâmetro para uma função ou método. Os arrays também podem ser multidimensionais, sendo utilizados sempre que há necessidade de tabular dados e os arrays de 2 dimensões (matrizes) são utilizados para processamento de imagens.

## Pilha

Em um array, é possível utilizar funções próprias para manipular elementos em qualquer posição da lista. Porém, há situações (veremos exemplos mais adiante) onde é desejável mais controle sobre as operações que podem ser feitas na estrutura. Aí entra a implementação de estruturas de dados como a **pilha** (*stack*) e a **fila** (*queue*). A pilha é uma estrutura de dados que, assim como o array, é similar a uma lista. O paradigma principal por trás da pilha é o **LIFO** - *Last In, First Out*, ou “o último a entrar é o primeiro a sair”, em tradução livre.

Para entendermos melhor o que significa isso, pense em uma *pilha* de livros ou de pratos. Ao empilharmos livros, por exemplo, o primeiro livro a ser retirado da pilha é obrigatoriamente o último que foi colocado; se tentarmos retirar o último livro da pilha, tudo vai desabar. Ou seja, o último livro a ser empilhado é o primeiro a ser retirado. Abstraindo este princípio para código, percebe-se que há apenas dois métodos possíveis para manipular os dados de uma pilha: 1) inserir um elemento no topo da pilha e 2) remover um elemento do topo da pilha.



Ao contrário do array, as linguagens de programação normalmente não têm métodos nativos para criação e manipulação de pilhas. Porém, é possível usar métodos de array para a implementação de pilhas.

## Usos

O caso de uso mais famoso da pilha é a *call stack* ou **pilha de chamadas** de um programa que está sendo executado: a ordem de execução dos processos “chamados” por um programa via funções ou métodos obedece o princípio de pilha. Outro recurso que utilizamos todos os dias e que utiliza pilhas para funcionar é o mecanismo de “voltar” e “avançar” páginas dos navegadores (representado normalmente por setas para a esquerda e direita). Os endereços visitados vão se empilhando; ao chamarmos a função de “voltar”, o último endereço visitado - ou seja, o que está no topo da pilha - é o primeiro a ser visualizado.

## Fila

A fila tem uma estrutura semelhante à pilha, porém com uma diferença conceitual importante: o paradigma por trás da fila é o **FIFO** - *First In, First Out*, ou “o primeiro a entrar é o primeiro a sair”, em tradução livre.

Pense em uma fila de bilheteria, por exemplo. A pessoa que chegou antes vai ser atendida (e comprar seu ingresso) antes de quem chegou depois e ficou atrás na fila. A fila como estrutura de dados segue o mesmo princípio.

Sendo assim, também há somente duas formas de se manipular uma fila: 1) Inserir um elemento no final da fila e 2) remover um elemento do início da fila.

## Deque

A estrutura de dados **deque** (abreviação de *double-ended queue* ou “fila de duas pontas”) é uma variação da fila que aceita inserção e remoção de elementos tanto do início quanto do final da fila.

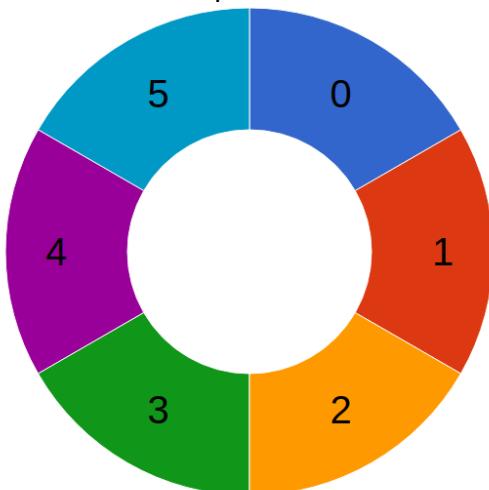
Podemos comparar, novamente, com uma fila de pessoas em um guichê de atendimento: uma pessoa idosa que chega é atendida antes (ou seja, não pode ser colocada no fim da fila), ao mesmo tempo que uma pessoa que entrou no final da fila pode desistir de esperar e ir embora (nesse caso, não podemos esperar a pessoa chegar na frente da fila para retirá-la de lá).

Uma outra forma de se entender a estrutura deque é como uma junção das estruturas de pilha e fila.



## Fila circular

Outra variação da fila é a **fila circular** (*circular queue*), onde o último elemento é conectado com o primeiro elemento - como em um círculo:



A fila circular busca resolver uma limitação da fila linear, que é lidar com espaços vazios que podem se enfileirar após a retirada de elementos do início da fila.

## Usos

Um uso fácil de lembrar para a fila é justamente a fila de impressão dos sistemas operacionais: o último trabalho de impressão a ser adicionado à fila será o último a ser impresso.

Além disso, as requisições feitas a um servidor também são organizadas em fila para serem respondidas, e quando alternamos entre programas utilizando o atalho alt+tab, o sistema operacional faz o gerenciamento da ordem utilizando o princípio de lista circular.

## Lista ligada

Já vimos que a maioria das linguagens de programação têm métodos nativos para a manipulação de arrays, como por exemplo inserir e remover elementos. Além disso, estes métodos fazem uma boa parte do trabalho de ordenar e buscar elementos por nós.

Porém, há três coisas importantes para sabermos sobre arrays: 1) na maior parte das linguagens de programação, os arrays têm tamanho fixo; 2) todos os elementos ocupam espaços sequenciais na memória e 3) inserir ou remover elementos do meio do array não é muito simples, pois exige que esses elementos sejam deslocados. Por exemplo:

// 0    1    2    3
[46, 34, 76, 12]



```
// removendo 76, o elemento 12 passa a ocupar o índice 2
```

// 0 1 2

[46, 34, 12]

// 0 1 2 3

[46, 34, 76, 12]

```
// inserindo 25, o elemento 12 passa a ocupar o índice 4
```

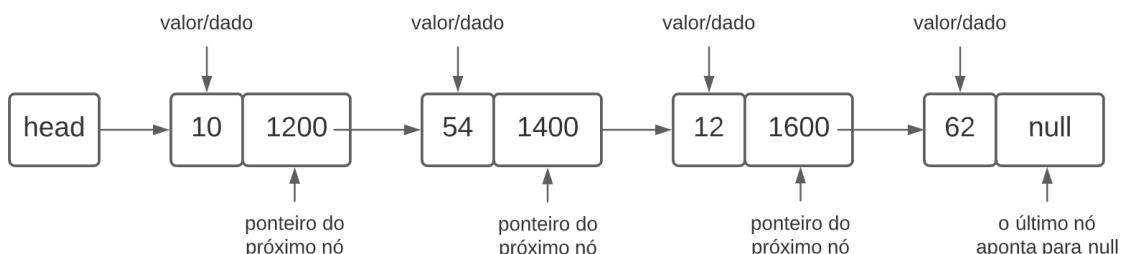
// 0 1 2 3 4

[46, 34, 76, 25, 12]

Este trabalho de deslocamento de elementos é feito internamente pelos métodos nativos de array que as linguagens já têm e que utilizamos no dia-a-dia, por isso normalmente não nos preocupamos com isso. Mas ele está na implementação do método feita no código-fonte da linguagem.

Assim como arrays, as listas ligadas também armazenam elementos sequencialmente, porém, ao invés de armazenar os elementos de forma contígua na memória, como nos arrays, as listas ligadas não dependem desse tipo de organização. Elas utilizam **ponteiros** para unir os elementos, e cada elemento “aponta” para o endereço de memória do próximo da lista, sem que ele precise estar no bloco de memória seguinte.

Dessa forma, ao trabalhar com listas ligadas também não é necessário fazer o deslocamento de elementos ao incluir ou excluir - considerando que, cada vez que esse deslocamento é feito em um array, é necessário reposicionar os elementos em novos espaços de memória. Cada “nó” da lista aponta para o ponteiro de memória onde se encontra o próximo elemento, independente de onde esteja este espaço de memória.



## Usos



Lembrando que a lista ligada encadeia elementos apontando para a localização do próximo item na memória, os programas de “álbum de fotos” (visualização de imagens) utilizam a lista ligada para encadear os arquivos de imagens e “chamar” a próxima imagem/voltar para a imagem anterior via setas do teclado ou botões de “anterior” e “próxima”. O mesmo princípio se aplica para playlists de música ou vídeo.

## Conjunto

A estrutura de dados **conjunto** (ou *set*) é uma lista não ordenada de elementos únicos. Ou seja, não é possível repetir o valor de um elemento dentro de um conjunto. Por exemplo, é perfeitamente possível criar um array com os seguintes elementos:

```
const lista = [1, 1, 1, 3, 5, 7, 9];
```

Porém, em um conjunto, não é possível repetir valores, não importa a ordem dos elementos. A maior parte das linguagens de programação mais usadas têm métodos nativos para criação de conjuntos. No caso do JavaScript:

```
const lista = [1, 1, 1, 3, 5, 7, 9];
```

```
const conjunto = new Set(lista);
```

```
console.log(conjunto);
```

```
// Set(5) { 1, 3, 5, 7, 9 }
```

## Usos

Como você pode ter imaginado, a estrutura do conjunto vem da matemática, e também é possível fazer operações como união e intersecção em conjuntos de dados. Um dos usos mais comuns desta estrutura é em bancos de dados SQL.

## Dicionário ou *hashmap*

**Dicionário** (também conhecido como **mapa**, *map* ou *hashmap*) é uma estrutura que guarda dados em **pares de chave e valor** e utiliza estas chaves para encontrar os elementos associados a elas, diferentemente das estruturas que vimos até agora, que trabalham com listas (sequenciais ou não) apenas de valores.

Essa descrição parece muito uma outra estrutura que já conhecemos, o objeto. Mas há várias diferenças entre dicionários/mapas e objetos. Por exemplo, é possível mapear o **tamanho** de um dicionário (ou seja, a quantidade de pares chave/valor) e os dicionários podem aceitar qualquer tipo de dado como chave (objetos aceitam apenas strings ou symbols). Os dicionários também podem ter performance melhor em buscas e manipulação de dados do que objetos, pois utilizam **referências** para as chaves - de uma forma similar a ponteiros, as chaves apontam para o endereço de memória de seus valores.



Um aspecto importante sobre dicionários vs objetos: a decisão sobre qual utilizar vai depender muito da linguagem. Cada linguagem implementa seus próprios métodos para criação de instâncias de objetos ou dicionários e para a manipulação dos dados. Por exemplo, dicionários em Python não eram ordenados por padrão até a versão 3.6, já o objeto `Map()` em JavaScript é ordenado por padrão; outras linguagens podem ter suporte a dicionários, porém não a objetos.

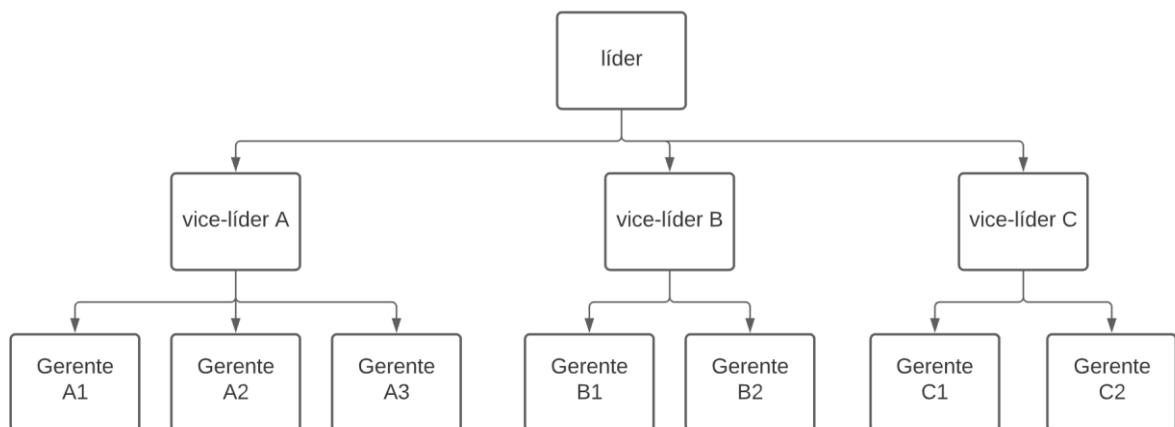
## Usos

O formato `chave:valor` do dicionário é o mais comum para lidar com dados armazenados em bancos de dados, mas não apenas para isso: o formato é usado sempre que há a necessidade de associar dados e referências para consulta posterior, sejam informações da base de dados de um produto ou mesmo a relação entre o nome de um arquivo e o seu caminho interno na memória do computador, feita pelo sistema operacional.

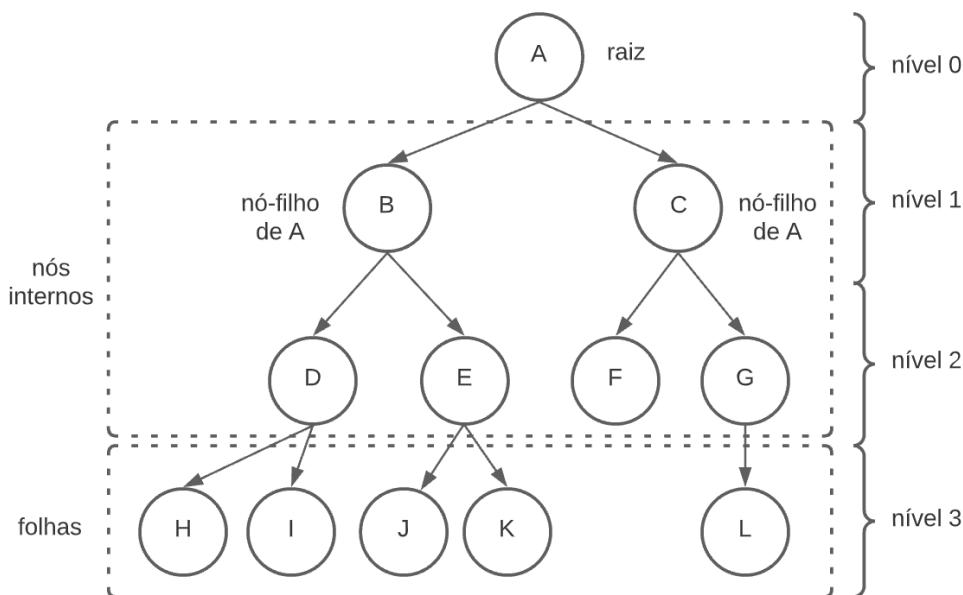
## Árvore

A **árvore** é uma estrutura não-sequencial, muito útil para armazenar dados de forma hierárquica e que podem ser acessados de forma rápida. Pode-se definir árvore como uma coleção de dados representados por **nós** e arranjados em níveis hierárquicos (ao invés de sequências como as estruturas vistas anteriormente).

Um exemplo de estrutura em árvore são os organogramas de empresas:



A estrutura mais comum é a **árvore binária**, que tem no máximo dois nós-filhos a partir do nó inicial (chamado de **raiz** ou **root**). Um nó pode ter pais, irmãos e filhos; nós que têm pelo menos um filho são chamados de nós internos, e nós sem filhos são chamados de externos ou folhas:



A partir da estrutura de árvore binária (com um nó-filho à esquerda e um à direita da raiz) é possível estruturar a chamada **BST**, ou **árvore de busca binária (binary search tree)**, que utiliza o princípio do algoritmo de busca binária para estruturar os dados de forma que valores menores estejam à esquerda da raiz e valores maiores à direita. Essa união do algoritmo com a estrutura de dados leva a uma maior eficiência na manipulação de dados, seja para buscar, alterar, incluir ou remover elementos.

## Heap binário

O **heap binário** é um tipo especial de árvore binária, normalmente utilizada em computação para implementar **filas de prioridade**, pois em um heap pode-se extrair de forma mais eficiente o valor mínimo ou máximo de uma lista. Pode-se traduzir *heap*, muito livremente, como um “monte” ou “porção” de dados.

O heap binário se difere da árvore binária em duas características principais:

- Todos os níveis, com exceção do último, têm filhos tanto na esquerda quanto na direita da raiz. No último nível, os filhos se posicionam o mais à esquerda possível. É o que chamamos de *árvore completa*.
- pode ser um **heap mínimo (min heap)**, para extrair o menor valor da árvore, ou **heap máximo (max heap)** para se extrair o maior valor. Todos os nós devem ser ou  $\geq$  (no caso do heap máximo) ou  $\leq$  (no caso do heap mínimo) do que os valores dos nós-filhos.

## Usos

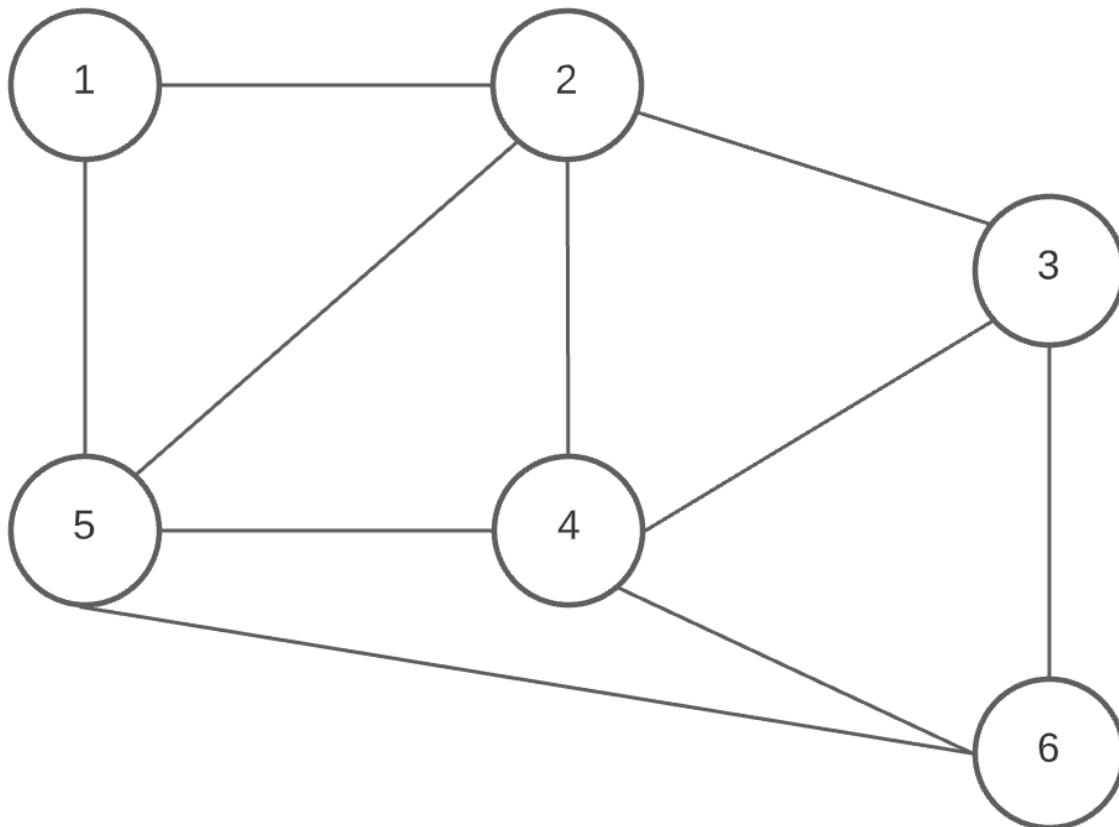
A estrutura de árvore tem vários usos diversos, como algoritmos de tomada de decisão em aprendizado de máquina, indexação de bancos de dados, indexação e exibição de arquivos e pastas no explorador de arquivos dos sistemas operacionais, entre vários outros casos.

O heap binário, como já mencionamos, é usado em filas de prioridade (tipo especial de fila onde os elementos são retirados da fila não no padrão FIFO, mas sim organizados por prioridade: mais prioritários no início da fila e menos prioritários no final) e também em um algoritmo de ordenação específico, o *heap sort*.



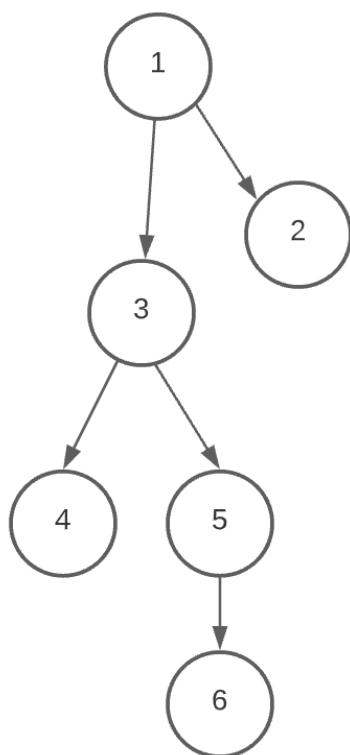
## Grafo

Outra estrutura não-sequencial, o **grafo** (*graph*) é um conjunto de nós (ou vértices), ordenados ou não e ligados por arestas, formando uma estrutura em forma de rede.



O grafo acima pode ser representado da seguinte forma:  $V = \{1, 2, 3, 4, 5, 6\}$  (os vértices ou nós) e  $E = \{(1,2), (2,3), (3,4), (3,6), (4,2), (4,5), (5,1), (5,2), (5,6)\}$  (as arestas ou edges).

Cada um dos vértices do grafo podem representar um tipo de dado ou sua referência. As arestas podem ser não-direcionadas - como no caso acima, onde as arestas não têm uma direção definida - ou direcionadas, como no caso abaixo:



As arestas de um grafo também podem ter um valor (também chamado de peso), que representam custo computacional, capacidade, etc para cada “caminho”.

## Usos

As redes sociais utilizam os grafos para manejar a grande quantidade de dados relacionados entre si que recebem a cada instante. O exemplo mais famoso, a linguagem de consulta GraphQL, foi criada pelo Facebook com o intuito de utilizar grafos para acessar e relacionar dados.

Outro uso famoso para os grafos é o sistema de navegação dos aplicativos de mapas/GPS, que utilizam grafos e o algoritmo de caminho mínimo (ou *shortest path*) para traçar rotas.

## O que são os operadores relacionais?

Os operadores relacionais representam a 3<sup>a</sup> categoria de operadores da programação e, em resumo, são usados para comparar valores de variáveis e criar declarações condicionais.

Neste sentido, as principais linguagens de programação utilizam os seguintes operadores relacionais:

&gt;

que representa **maior que**.

&lt;



que representa **menor que**.

&gt;=

que representa **maior ou igual**.

&lt;=

que representa **menor ou igual**.

==

que representa **igual**.

!=

que representa **diferente**.

Esses operadores são usados com o intuito de criar expressões do tipo **verdadeiro** (TRUE) ou **falso** (FALSE) (fundamentais para as declarações condicionais).



**TRUE**



**FALSE**

É importante ressaltar que algumas linguagens de programação ainda podem possuir alguns operadores a mais. Por exemplo, a linguagem *Javascript*, que também possui os seguintes operadores:

==

que representa **estritamente igual**.

!=

que representa **estritamente diferente**.

Só para exemplificar, observe no código abaixo o uso do operador relacional < (menor que) em uma declaração condicional:

```
1.if(idade < 18){  
2.    println("Você não pode acessar o sistema! É menor de idade!");  
3.}
```



Na **linha 1** do código acima, temos uma expressão condicional que verifica se o valor da variável **idade** é menor que 18. De tal forma que, caso seja verdade, o comando de impressão na **linha 2** será executado.

Mas vamos nos aprofundar um pouco mais no assunto!

## Expressões condicionais

Primeiramente, uma expressão condicional é o produto gerado pelo uso dos operadores relacionais. De tal forma que, quando você usa os operadores relacionais, você cria expressões que retornam dois possíveis valores: **Verdadeiro** ou **Falso**.

A criação de uma expressão condicional é dada pela estrutura abaixo:

[membro da esquerda] OPERADOR RELACIONAL [membro da direita];

Só para ilustrar, veja alguns exemplos de expressões condicionais:

**salario < 3500.00**

**idade >= 18**

**nome != "João"**

**nota == 10**

Todas essas expressões acima podem ser interpretadas como **perguntas** que resultam em **Verdadeiro** ou **Falso**, e que, portanto, permitem ao algoritmo tomar decisões de fazer ou não fazer alguma operação durante a execução de um programa.

## Não confunda == com =

Um dos operadores comuns da programação é o operador de atribuição, representado pelo símbolo **=**.

Nesse sentido, é muito comum alunos iniciantes confundirem o operador de igualdade **==** com o operador de atribuição **=**.

Contudo, eles possuem propósitos distintos e funcionam de maneira bem diferente. Por isso, **não os confunda!**

## Operadores de atribuição, aritméticos, relacionais e lógicos no Java

Os **operadores de atribuição, aritméticos,relacionais e lógicos** no **Java** são utilizados principalmente na etapa de processamento - para a construção da lógica - possibilitando realizar ações específicas sobre os dados. Adição, subtração, multiplicação, comparação são apenas alguns exemplos.

Neste documento apresentamos os principais operadores utilizados para a escrita de algoritmos em **Java**.



## Operadores de atribuição em Java

O operador de atribuição é utilizado para definir o valor inicial ou sobrescrever o valor de uma variável. Em seu uso, o operando à esquerda representa a variável para a qual desejamos atribuir o valor informado à direita.

Exemplo de uso:

```
int lado = 2;  
float pi = 3.1426F;  
String texto = "DevMedia";  
lado = 3;
```

**Run**

Nesse exemplo iniciamos as variáveis `lado`, `pi` e `texto`, sobrescrevendo a variável `lado` em seguida.

## Operadores aritméticos

Os operadores aritméticos realizam as operações fundamentais da matemática entre duas variáveis e retornam o resultado. Caso seja necessário escrever operações maiores ou mais complexas, podemos combinar esses operadores e criar expressões, o que nos permite executar todo tipo de cálculo de forma programática.

Exemplo de uso:

```
int area = 2 * 2;
```

**Run**

Esse código demonstra como calcular a área de um quadrado de lado igual a 2.

Também podemos utilizar os operadores aritméticos em conjunto com o operador de atribuição, realizando, em uma mesma instrução, as ações de calcular o valor e atribuí-lo à variável.

Exemplo de uso:

```
int area = 2;  
area *= 2;
```

**Run**

*Nota:* A segunda linha desse código é equivalente a `area = area * 2.`

## Opções de operadores aritméticos

A tabela abaixo apresenta os **operadores aritméticos** da linguagem **Java**:

+	operador de adição
-	operador subtração



*	operador de multiplicação
/	operador de divisão
%	operador de módulo (ou resto da divisão)

## Operadores de incremento e decremento

Os **operadores de incremento** e decremento também são bastante utilizados. Basicamente temos dois deles: `++` e `--`, os quais podem ser declarados antes ou depois da variável e incrementam ou decrementam em 1 o valor da variável.

Exemplo de uso:

```
int numero = 5;  
numero++;  
numero--;  
//numero continuará sendo 5.
```

**Run**

Quando declaramos esse operador antes da variável, o incremento é realizado antes do valor da variável ser lido para o processamento ao qual a instrução pertence. Quando declarado depois, ocorre o contrário: lê-se o valor da variável para processamento e só então o valor da variável é incrementado. Com base nisso, suponha que temos o código abaixo:

Exemplo de uso:

```
int desafioUm = 5;  
System.out.println(desafioUm += ++desafioUm );  
  
int desafioDois = 5;  
System.out.println(desafioDois += desafioDois++);
```

**Run**

Quais valores serão impressos no console? 10 e 10, 10 e 11, 11 e 10 ou 11 e 11? A resposta é 11 e 10.

No primeiro `println()`, `desafioUm` é incrementado antes de seu valor ser lido para compor a instrução de soma. Sendo assim, temos `desafioUm = 5 + 6`. Já no segundo `println()`, primeiro o valor é lido, resultando em `desafioDois = 5 + 5`. Somente após a leitura `desafioDois` é incrementado, e depois, recebe o valor da soma, tendo seu valor sobreescrito com o número 10.



## Operadores de igualdade

Os operadores de igualdade verificam se o valor ou o resultado da expressão lógica à esquerda é igual (“`==`”) ou diferente (“`!=`”) ao da direita, retornando um valor booleano.

Exemplo de uso:

```
int valorA = 1;  
int valorB = 2;  
  
if(valorA == valorB){  
    System.out.println("Valores iguais");  
} else {  
    System.out.println("Valores diferentes");  
}
```

**Run**

Esse código verifica se duas variáveis contêm o mesmo valor e imprime o resultado. Uma vez que as variáveis `valorA` e `valorB` possuem valores diferentes, o trecho de código presente no `else` será executado. Caso ainda não conheça as estruturas de condição, acesse: [Documentação Java: if/else e o operador ternário](#).

## Opções de operadores de igualdade

A tabela abaixo apresenta os **operadores de igualdade** do Java:

<code>==</code>	Utilizado quando desejamos verificar se uma variável é igual a outra.
<code>!=</code>	Utilizado quando desejamos verificar se uma variável é diferente de outra.

*Nota: Os operadores de igualdade normalmente são utilizados para comparar tipos primitivos (byte, short, int, long, float, double, boolean e char). No entanto, também podemos utilizá-los para saber se duas instâncias estão apontando para o mesmo objeto.*

## Operadores relacionais

Os **operadores relacionais**, assim como os de igualdade, avaliam dois operandos. Neste caso, mais precisamente, definem se o operando à esquerda é menor, menor ou igual, maior ou maior ou igual ao da direita, retornando um valor booleano.

Exemplo de uso:

```
int valorA = 1;  
int valorB = 2;  
  
if (valorA > valorB) {
```



```
        System.out.println("maior");  
    }  
  
    if (valorA >= valorB) {  
        System.out.println("maior ou igual");  
    }  
  
    if (valorA < valorB) {  
        System.out.println("menor");  
    }  
  
    if (valorA <= valorB) {  
        System.out.println("menor ou igual");  
    }
```

**Run**

Esse código realiza uma série de comparações entre duas variáveis para determinar o que será impresso no console. Uma vez que o valor da variável `valorA` é menor que `valorB` serão impressas as mensagens “menor” e “menor ou igual”.

## Opções de operadores relacionais

A tabela abaixo apresenta os operadores relacionais do Java:

>	Utilizado quando desejamos verificar se uma variável é maior que outra.
>=	Utilizado quando desejamos verificar se uma variável é maior ou igual a outra
<	Utilizado quando desejamos verificar se uma variável é menor que outra.
<=	Utilizado quando desejamos verificar se uma variável é menor ou igual a outra.

## Operadores lógicos

Os **operadores lógicos** representam o recurso que nos permite criar expressões lógicas maiores a partir da junção de duas ou mais expressões. Para isso, aplicamos as operações lógicas E (representado por “`&&`”) e OU (representado por “`||`”).



Exemplo de uso:

```
if((1 == (2 -1)) && (2 == (1 + 1))){  
    System.out.println("Ambas as expressões são verdadeiras");  
}
```

**Run**

Uma vez que utilizamos o operador lógico `&&`, o `System.out.println` somente será executado se as duas condições declaradas no `if` forem verdadeiras.

## Opções de operadores de lógicos

A tabela abaixo apresenta os operadores lógicos do Java:

<code>&amp;&amp;</code>	Utilizado quando desejamos que as duas expressões sejam verdadeiras.
<code>  </code>	Utilizado quando precisamos que pelo meno um das expressões seja verdadeira.

## Precedência de operadores

Uma vez que os operadores aritméticos buscam reproduzir as operações matemáticas fundamentais, é natural que eles mantenham as suas regras de precedência, que podem ser manipuladas pelo programador com o uso de parênteses.

Por exemplo, a expressão  $1 + 1 * 2$ , quando analisada pelo compilador, vai retornar o valor 3, porque a multiplicação será resolvida antes da adição. Usando parênteses, a expressão  $(1 + 1) * 2$  retornará o valor 4, pois a adição, por estar dentro dos parênteses, será resolvida primeiro.

Exemplo de uso:

```
if ((1 != (2 -1)) || (2 == (1+1))) {  
    System.out.println("iguais");  
}
```

**Run**

*Nota: Para facilitar a leitura das expressões e evitar erros de lógica, é recomendado o uso dos parênteses para separar e agrupar as condições.*

## Exemplo prático

Suponha que você precisa programar um código simples para definir o salário dos funcionários de uma empresa considerando o tempo que cada um tem nessa empresa e o número de horas trabalhadas. Para tanto, podemos utilizar alguns dos operadores apresentados nessa documentação.



Exemplo de uso:

```
int quantidadeAnos = 5;  
int horasTrabalhadas = 40;  
int valorHora = 50;  
int salario = 0;  
  
if (quantidadeAnos <= 1) {  
    salario = 1500 + (valorHora * horasTrabalhadas);  
} else if ((quantidadeAnos > 1) && (quantidadeAnos < 3)) {  
    salario = 2000 + (valorHora * horasTrabalhadas);  
} else {  
    salario = 3000 + (valorHora * horasTrabalhadas);  
}
```

**Run**

Operador lógico

#### Definição

**AND**, **NAND**, **OR**, **XOR** e **NOT** são os principais operadores lógicos, base para a construção de sistemas digitais e da Lógica proposicional, e também muito usado em linguagem de programação. Os operadores **AND**, **NAND**, **OR** e **XOR** são operadores binários, ou seja, necessitam de dois elementos, enquanto o **NOT** é unário. Na computação, esses elementos são normalmente variáveis binárias, cujos possíveis valores atribuídos são 0 ou 1. Porém, a lógica empregada para essas variáveis serve também para sentenças (frases) da linguagem humana, onde se esta for verdade corresponde ao valor 1, e se for falsa corresponde ao valor 0.

#### Utilização

- **x1 AND x2**
- **x1 NAND x2**
- **x1 OR x2**
- **x1 XOR x2**
- **NOT x1**

#### Descrição

##### **AND**

Operador lógico no qual a resposta da operação é verdade (1) se ambas as variáveis de entrada forem verdade.



x1	x2	x1 AND x2
0	0	0
0	1	0
1	0	0
1	1	1

### NAND

Operador lógico no qual a resposta da operação é verdade (1) se pelo menos uma das variáveis é falso

x1	x2	x1 NAND x2
0	0	1
0	1	1
1	0	1
1	1	0

### OR

Operador lógico no qual a resposta da operação é verdade (1) se pelo menos uma das variáveis de entrada for verdade.

x1	x2	x1 OR x2
0	0	0
0	1	1
1	0	1
1	1	1

### XOR

Operador lógico no qual a resposta da operação é verdade (1) quando as variáveis assumirem valores diferentes entre si.

x1	x2	x1 XOR x2
0	0	0
0	1	1
1	0	1
1	1	0

### NOT

Operador lógico que representa a negação (inverso) da variável atual. Se ela for verdade, torna-se falsa, e vice-versa

x1	NOT x1
0	1
1	0

## Operadores de Incremento e Decremento

Os operadores de *incremento* e *decremento* são operadores unários usados para adicionar ou subtrair 1 do valor de uma variável numérica. O operador de incremento é simbolizado por `++`, e o de decremento, por `--`.

Os operadores de incremento e decremento podem ser pré-fixos ou pós-fixos, dependendo de serem posicionados antes ou depois do nome da variável a ser incrementada ou decrementada.



Um operador pós-fixo utiliza o valor atual da variável na expressão em que ela se encontra, e somente depois muda o valor dessa variável. Já um operador pré-fixo primeiramente altera o valor da variável, e então usa esse novo valor na expressão onde ela se encontra.

A tabela a seguir mostra os operadores e seus significados.

Operador	Significado
x++	$x = x + 1$ (adiciona 1 ao valor de x, armazena o resultado em x, e retorna o valor original)
x-	$x = x - 1$ (subtrai 1 do valor de x, armazena o resultado em x, e retorna o valor original)
++x	$x = x + 1$ (adiciona 1 ao valor de x, armazena o resultado em x, e retorna o novo valor incrementado)
-x	$x = x - 1$ (subtrai 1 do valor de x, armazena o resultado em x, e retorna o novo valor decrementado)

### Exemplo

O código a seguir mostra a aplicação dos operadores de incremento pós-fixo e pré-fixo:

```
public static void main(String[] args) {
    int num = 10;
    int result = 0;
    System.out.println("Valor original: " + result);
    result = num++; // Primeiro atribui, depois incrementa
    System.out.println("Valor de num após o incremento: " + num);
    System.out.println("Valor de result após o incremento: " + result);
    num = 10; result = 0;
    result = ++num; // Primeiro incrementa, depois atribui
    System.out.println("Valor de num após o incremento: " + num);
    System.out.println("Valor de result após o incremento: " + result);
}
```

Usamos duas variáveis, *num* e *result*, inicializadas com os valores 10 e 0, respectivamente. Mostramos na tela o valor original de *result* para começar.

Logo na sequência, atribuímos a *result* o valor de *num* incrementada, usando operador pós-fixo de incremento (*num++*). Neste caso, o valor original de *num* é atribuído a *result*, e só então seu próprio valor é incrementado, passando a valer 11. Mostramos na tela os valores das variáveis.

Na sequência voltamos os valores das variáveis aos do início do programa. E então, repetimos a atribuição de *num* a *result*, porém desta vez incrementando a variável *num* com um operador pré-fixo (*++num*). Agora, a variável é primeiramente incrementada, e então o resultado do incremento é atribuído à *result*, e os valores de ambas as variáveis são finalmente exibidos no console. O resultado pode ser visto a seguir:



```
Valor original: 0
Valor de num após o incremento: 11
Valor de result após o incremento: 10
Valor de num após o incremento: 11
Valor de result após o incremento: 11
```

## Estruturas sequenciais, condicionais e de repetição

Todos os softwares que usamos no nosso dia a dia são formados por símbolos e termos escritos em inglês, pois este é a idioma que normalmente é usado na programação.

Esse código é dividido em três estruturas que, quando trabalham juntas e corretamente, formam algoritmos e instruções básicas para programas complexos. Vejamos:

### 1. Sequenciais

São uma sequencia de ações desenvolvidas em uma ordem específica, e realiza uma ação depois da outra até que todas tenham sejam concluídas.

Um exemplo claro disso é uma rotina da manhã: você se levanta, bebe água, toma banho, toma café da manhã, etc.

### 2. Condicionais

Esse tipo de estrutura, em vez de seguir uma ordem específica de comandos, faz uma pergunta para determinar qual caminho seguir.

Para entender melhor esse conceito, vamos analisar o seguinte caso: digamos que você vai escovar os dentes e observe que o creme dental acabou. Então você se pergunta: "será que tenho mais pasta de dente?"



*Será que tenho mais pasta de dente?*



[edu.gefglobal.org](http://edu.gefglobal.org)

Se a resposta for negativa, você adicionaria este item à sua lista de mercado.  
Mas, se a resposta for afirmativa, você simplesmente a usaria.

Essa é a função básica das condicionais: responder perguntas com base numa constatação.

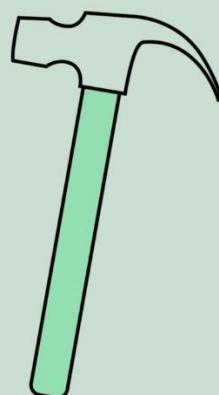
### 3. De repetição

Assim como as condicionais, as estruturas de repetição também fazem perguntas. A diferença é que é feita a mesma pergunta repetidamente até que uma determinada tarefa tenha sido elaborada.

Por exemplo, quando você prega um prego na parede, embora você não perceba, você constantemente se pergunta: "O prego já entrou?"



O prego  
já entrou?



edu.gcfglobal.org

Quando a resposta for não, você martela de novo e continua repetindo essa pergunta até que a resposta seja afirmativa e você possa parar.

Es estruturas de repetição nos ajudam a programar tarefas repetitivas, sem ter que fazer o mesmo código repetidas vezes, para a mesma ação.

## Desvio Condisional Composto (SE...ENTÃO...SENÃO)

O desvio condicional **composto** tem por finalidade tomar decisões de acordo com o resultado de uma condição (teste lógico), da mesma forma que o **desvio condicional simples** que estudamos na aula anterior. Porém, enquanto o condicional simples somente executa instruções quando o teste condicional retorna verdadeiro, o condicional composto permite criar dois blocos de código:

- Se o teste lógico retornas verdadeiro, as instruções contidas entre os comandos **então** e **senão** serão executadas (como no condicional simples)
- Entretanto, se o teste lógico retornar falso, as instruções contidas entre os comandos **senão** e **fimse** serão executadas.

### Sintaxe:

**se** (condição) **então**

Instruções caso condição retorne verdadeiro

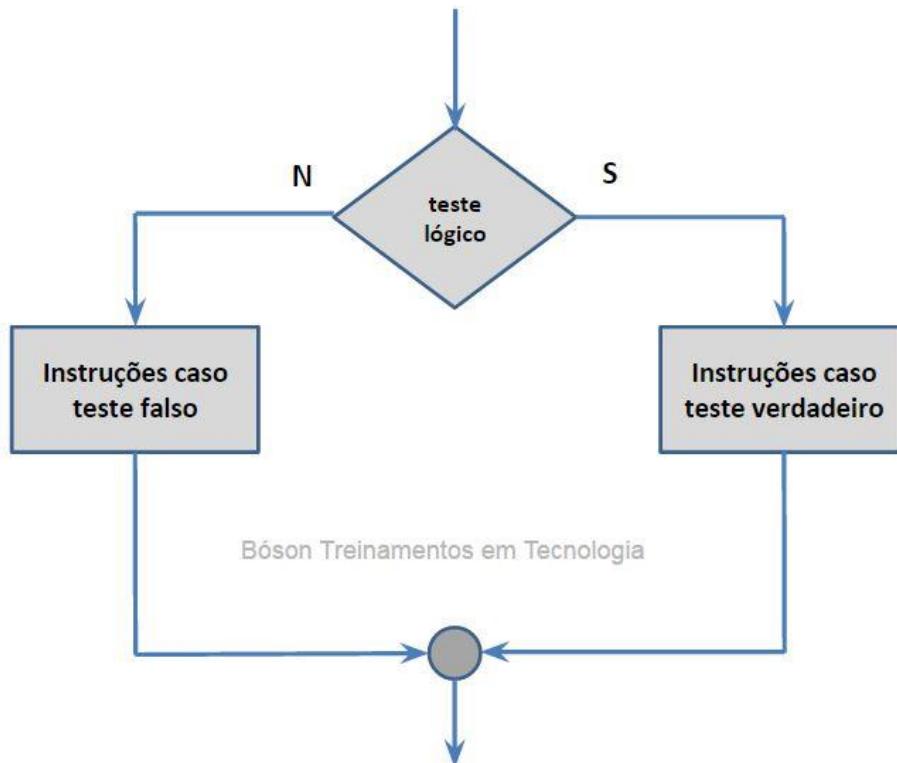
**senão**

Instruções caso condição retorne falso

**fimse**

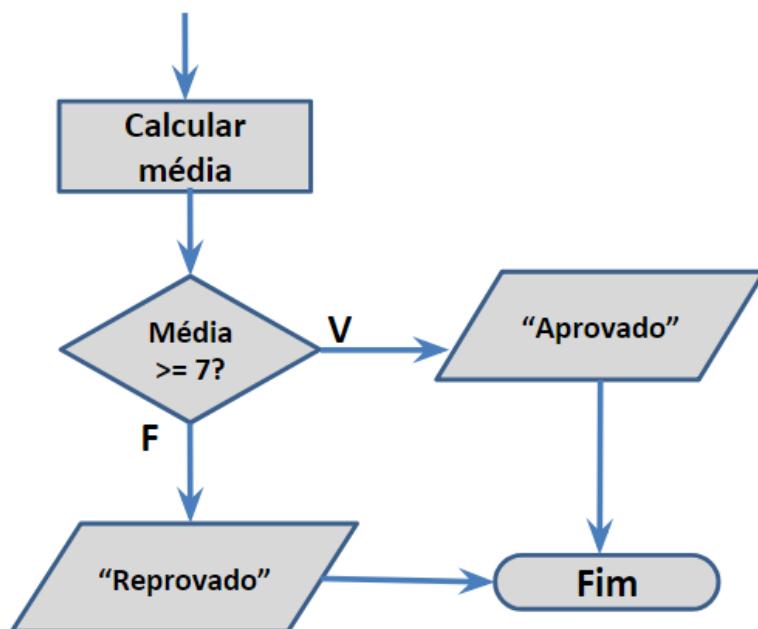
Instruções após executar as instruções de condição verdadeira ou falsa

O fluxograma a seguir ilustra esse processo:



**Exemplo:** Vamos incrementar o algoritmo que criamos no artigo sobre condicional simples. Queremos agora que o algoritmo verifique a nota média de um aluno, e retorne a mensagem “Aprovado” se essa nota média for maior ou igual ao valor 7, além de ecoar na tela essa nota. Caso a nota média seja menor do que 7, o programa deve retornar a mensagem “Reprovado”, além de mostrar a nota média digitada.

Veja abaixo um fluxograma representando o algoritmo do exemplo:





E logo a seguir, o algoritmo implementado em português estruturado (no VisualG), usando o condicional composto:

```
algoritmo RESULTADO_MÉDIA
var
  N1, N2 : inteiro
  MEDIA : real
início
  leia N1
  leia N2
  MEDIA <- (N1 + N2) / 2
  se (media >= 7) entao
    escreva ("Aprovado")
  senao
    escreva ("Reprovado")
  fimse
  escreva ("Sua média é ", MEDIA)
finalgoritmo
```

## Laços de repetição

Laços de repetição, também conhecidos como laços de iteração ou simplesmente *loops*, são comandos que permitem iteração de código, ou seja, que comandos presentes no bloco sejam repetidos diversas vezes. Através de laços de repetição é possível criar programas que percorram *arrays*, analisando individualmente cada elemento, e até mesmo criar trechos de código que sejam repetidos até que certa condição estabelecida seja cumprida.

Perl possui basicamente quatro tipos de laço de repetição: **while**, **do... while**, **for** e **foreach**.

**diegomariar**  
Ph.D. •

- [While](#)
- [Do... while](#)
- [For](#)
- [Foreach](#)
- [Cláusulas last e next](#)
- [Qual o melhor laço de repetição?](#)
- [Construindo uma calculadora usando laços de repetição](#)

### While

O laço **while** (na tradução literal para a língua portuguesa “enquanto”) determina que enquanto uma determinada condição for válida, o bloco de código será executado. O laço *while* testa a condição antes de executar o código, assim sendo, caso a condição seja inválida no primeiro teste o bloco nem é executado.



```
# Contador $i
my $i = 0;

while($i < 10){
    print "$i "; # 0 1 2 3 4 5 6 7 8 9
    $i++;
}
```

Aqui inserimos o conceito de **contador**. Um contador auxilia na determinação de quantas vezes um laço de iteração deve ser executado. Chamamos nosso contador de `$i` (“`i`” de iteração). Nesse exemplo, iniciamos nosso contador com valor zero, a seguir o laço `while` repete todo o bloco de código enquanto `$i` for menor do que 10. A cada iteração imprimimos o valor de `$i` e ao final do código incrementamos `$i` em um elemento. Assim, o código se repete até `$i` ser igual a 10. Uma vez que isso ocorre, o código é interrompido antes do bloco ser executado. Por isso é impresso de 0 a 9. Se a incrementação não for feita o programa ficará executando eternamente. Chamamos isso de **loop infinito**.

## Do... while

O laço **do... while** (na tradução literal para a língua portuguesa “faça... enquanto”), assim como o laço `while`, determina que enquanto uma determinada condição for válida o bloco de código será executado. Entretanto, `do... while` testa a condição após executar o código, assim sendo, mesmo que a condição seja considerada inválida no primeiro teste o bloco será executado pelo menos uma vez.

```
# Contador $i
my $i = 0;

do{
    print "$i "; # 0 1 2 3 4 5 6 7 8 9
    $i++;
}while($i < 10);
```

## For

O comando **for** (na tradução literal para a língua portuguesa “para”) permite que uma variável contadora seja testada e incrementada a cada iteração, sendo essas informações definidas na chamada do comando. O comando `for` recebe como entrada uma variável contadora, a condição e o valor de incrementação.

```
# Iteracao for
for(my $i = 0; $i < 10; $i++){
    print $i; # 0123456789
}
```

Uma outra forma de usarmos o laço `for` é declarando uma variável, seguida por dois valores da iteração (primeiro e último) entre parênteses separados por dois pontos finais. Perl incrementará o valor em um elemento a cada rodada.

## Foreach

O comando **foreach** (na tradução literal para a língua portuguesa “para cada”), diferente do comando `while`, realiza iterações sobre coleções. Sendo assim, a cada “volta” do laço, a variável definida na chamada do `foreach` assume o valor de um dos elementos da coleção. A execução do comando é finalizada quando chega ao fim da coleção ou através de um comando de interrupção, como veremos em breve.



```
my @aminoacidos = ("Alanina","Cisteina","Aspartato");
my $a;

foreach $a(@aminoacidos){
    print $a."\n";
}
# Alanina
# Cisteina
# Aspartato
```

O comando *foreach* recebe um *array* como parâmetro e aplica, a cada iteração, um elemento a uma variável pré-definida. Nesse exemplo, cada elemento do *array* @aminoacidos é aplicado a variável \$a a cada repetição. O número de repetições depende da quantidade de elementos do *array* (no caso três).

Podemos ainda iterar sobre *hashes* usando *foreach*. Entretanto, é necessário um outro laço para percorrer *arrays* presentes dentro da *hash*. Veja:

```
my $chave;
my $i;
my %aminoacidos = (
    polares => ["Aspartato","Glutamato"],
    apolares => ["Alanina","Cisteina"]
);

foreach $chave(keys %aminoacidos){
    for $i( 0 .. $#{ $aminoacidos{$chave} } ){
        print "chave: $chave\n";
        print "valor: $aminoacidos{$chave}[$i] \n\n";
    }
}
#chave: polares
#valor: Aspartato
#chave: polares
#valor: Glutamato
#chave: apolares
#valor: Alanina
#chave: apolares
#valor: Cisteina
```

Observe que o laço *foreach* percorre a lista de chaves da *hash* %aminoacidos. Cada chave é armazenada em uma variável em “foreach \$chave(keys %aminoacidos){}”. Note que não será necessário percorrer os valores com um laço, uma vez que a própria chave é responsável por ser o índice de busca na *hash*.

Dentro da *hash* %aminoacidos existem valores que armazenam *arrays*. Assim sendo, é necessário que um novo laço percorra cada elemento do *array*. Para isso foi utilizado o laço *for*.

Observe que no trecho “for \$a( 0 .. \$#{ \$aminoacidos{\$chave} } ){}”, utilizamos uma nova forma para contar a quantidade de elementos dentro do *array*: “\$#{ nome\_de\_um\_array }”. Dessa forma, o laço *for* percorre de 0 até o tamanho do *array*, e armazena o valor na variável contadora \$i.

A variável \$i é utilizada como índice na linha “print “chave: \$chave\nvalor: \$aminoacidos{\$chave}[\$i] \n\n”;”. Observe que para imprimir um elemento dentro de um *array* gravado em uma *hash* é necessário declarar a *hash* com o caractere “\$” seguido pela chave, que no caso estava armazenada na variável \$chave, e por fim seguido por colchetes e pela variável contadora “[ \$i ]”.

Observe que a primeira iteração é executada duas vezes: uma para cada chave (polares e apolares). Para cada iteração do primeiro *for*, há duas iterações em cada *array* (“Aspartato” e “Glutamato” para chave polares, e “Alanina” e “Cisteina” para a chave apolares). Assim, o comando *print* foi executado oito vezes (duas chaves vezes dois *arrays* vezes dois elementos em cada *array*).



## Cláusulas *last* e *next*

Caso desejemos alterar o fluxo de execução de um laço de repetição, podemos utilizar a cláusula **last**, para interromper a execução, ou **next**, para prosseguir para a próxima iteração. Veja:

```
my $i;

for $i (1 .. 10){
    if($i > 5){
        last;
    }
    print $i; #12345
}
```

A cláusula *last* é o que mais se aproxima das cláusulas *break* presentes em outras linguagens, como Python por exemplo. O comando *last* define que aquela execução do laço é a última. Observe agora o comando *last* utilizado junto ao laço *while*.

```
my $i = 0;

while($i < 10){
    print $i; #012345
    $i++;
    if ($i > 5){
        last;
    }
}
```

A cláusula *next* prossegue para próxima execução do laço desprezando todos os comandos do bloco declarados abaixo. No exemplo a seguir usaremos um comando condicional *if* para verificar se o valor de \$i é um numeral ímpar. Se for, o comando *next* salta para próxima execução do laço, desprezando o comando *print* logo abaixo, que imprimirá apenas os valores pares.

```
my $i = 0;

while($i < 10){
    $i++;
    if ($i % 2 == 1){
        next;
    }
    print $i; # 246810
}
```

## Qual o melhor laço de repetição?

Laços de repetição devem ser utilizados de acordo com a ocasião. Recomendamos utilizar o laço *while* sempre que não soubermos a quantidade de vezes que o bloco se repetirá. O laço *do...while* pode ser utilizado no mesmo caso, entretanto *do...while* deve ser utilizado caso queremos que o código seja executado pelo menos uma vez. O laço *for* deve ser utilizado sempre que soubermos a quantidade exata de iterações que deverão ser feitas. O laço *foreach* deve ser utilizado para iterar sobre *arrays*.

Muitas vezes o uso de diferentes laços pode apresentar os mesmos resultados. Logo, o programador tem a liberdade de utilizar aquele que tiver maior confiança e facilidade.



## Construindo uma calculadora usando laços de repetição

Nos capítulos anteriores aprendemos a utilizar operadores numéricos para construir uma calculadora. No exemplo a seguir, faremos algumas modificações no *script* criado anteriormente para que a calculadora faça mais do que uma operação. Para isso, vamos criar uma variável chamada `$SAIR`, inicialmente com o valor 0. Em seguida vamos inserir todo o resto do código dentro de um laço de repetição `while`, que repetirá o bloco enquanto `$SAIR` for igual a 0. Ao final do código enviaremos ao usuário uma mensagem perguntando se deseja fazer outro cálculo, ou se deseja encerrar o *script*. Caso o usuário deseje encerrar o *script*, a variável `$SAIR` recebe o valor 1, e o laço não se repetirá.

```
use warnings;
use strict;

# Calculadora
my $SAIR = 0;
my $num1;
my $num2;
my $operador;
my $operacao;
my $opcao;

while($SAIR == 0){
    print "CALCULADORA";
    print "\n\n";
    print "Digite o primeiro numero: ";
    chomp($num1 = <STDIN>);
    print "\n";
    print "Digite a operacao desejada:
1 para soma
2 para subtracao
3 para divisao
4 para multiplicacao: ";
    chomp($operador = <STDIN>);
    print "\n";
    print "Digite o segundo numero: ";
    chomp($num2 = <STDIN>);

    # Determinando qual o operador foi utilizado
    if ($operador == 1){
        $operacao = $num1 + $num2;
    }
    elsif ($operador == 2){
        $operacao = $num1 - $num2;
    }
    elsif ($operador == 3){
        $operacao = $num1 / $num2;
    }
    elsif ($operador == 4){
        $operacao = $num1 * $num2;
    }
    else {
        system("clear");
    }
}

#use system("cls") para S.O. Windows
```



```
        print "Operador invalido. Digite 1, 2, 3 ou 4. Tente
novamente.\n\n$";
        next;
    }

    print "\n";
    print "Resultado: ";
    print $operacao;
    print "\n";

    # Questiona se o usuario deseja sair
# ou fazer outra conta
    print "Deseja sair? (S/s) SIM ou (N/n) NAO: ";
    chomp($opcao = <STDIN>);

    if(($opcao eq "S")or($opcao eq "s")){
        $SAIR = 1;
    }

    system("clear"); #use system("cls") para S.O. Windows
}
```

Curiosamente utilizamos a função **system** para fazer a limpeza da tela. A função *system* permite que funções nativas do sistema operacional sejam executadas pelo Perl. No caso, usamos *system* para executar o comando “*clear*”, que em sistemas operacionais Linux e Mac limpa a tela. Caso utilize sistema operacional Windows, utilize o comando “*cls*”.