

UNIDADE 3 – ESTRUTURAS DE REPETIÇÃO, FUNÇÕES, RECURSIVIDADE, FILA E PILHA

MÓDULO 1 – FUNÇÕES DE INTERFACE E MATEMÁTICA

01

1 - FUNÇÕES DE INTERFACE E MATEMÁTICA

No início do semestre aprendemos algoritmos básicos, que foram ficando mais complexos com o avançar do curso. Estamos crescendo e evoluindo a matéria à medida que você está fazendo os exercícios e atividades.

Neste módulo aprenderemos que nos algoritmos existem **comandos já predeterminados a executarem uma ação**. Esse comando já configurado é chamado de **função**.

Uma função é composta de um nome, uma lista de parâmetros entre parênteses () e o bloco de código associado.

Vimos anteriormente que podemos utilizar diversos operadores e funções nos nossos algoritmos. Podemos, também, definir novas funções. A utilização de funções algorítmicas tem duas grandes vantagens:

- a) Possibilita tratar problemas complexos: um algoritmo grande pode ser dividido em partes, sendo cada parte escrita como uma função.
- b) Possibilita diminuir o custo/tempo de desenvolvimento de programas: podemos utilizar num algoritmo funções previamente escritas para outros algoritmos.

02

No exemplo abaixo apresentamos uma função de **leitura** e uma de **escrita**. Essas duas funções já foram apresentadas nos módulos passados.

leia (k), k=k+1, escreva (k)

No exemplo acima a função **leia()** irá receber um valor e a função **escreva()** apresentará na tela a soma dos valores da variável k.

Note que uma semântica especial foi atribuída aos algoritmos.

Essa atribuição descreve um significado previamente definido pelo interpretador, elas não podem ser utilizadas para outro fim, senão aquele a que foi originalmente atribuída. O interpretador pode ser qualquer um ou qualquer coisa que vá ler o algoritmo.

03

As funções são divididas em classes:

Palavras reservadas	Estão relacionadas com a definição de tipos de dados e também com as estruturas de controle do fluxo.	Exemplos de palavras reservadas: tipo, caracter, cadeia, inteiro, real, Classe, básico, vetor, matriz, cubo. Veja mais exemplos
Funções de interface	As funções de interface são as instruções que nos permitem realizar um diálogo entre nosso algoritmo e o usuário.	Exemplos de funções de interface: Escreva() , Leia() , Posicao() , Moldura() , Pausa() . Veja mais exemplos
Funções matemáticas	São as instruções que nos permitem utilizar de recursos matemáticos frequentemente solicitados na resolução de algum tipo de problema. Com as funções matemáticas tornou mais fácil a forma de realizar cálculos.	São exemplos de funções matemáticas: Sinal() , ValorAbs() , ParteInteira() , ParteDecimal() , Quociente() , Resto() , Raiz() , Potencia() . Veja mais exemplos

Existem, também, as **funções de cadeia** e **funções de arquivo**, as quais serão estudadas no próximo módulo.

Apresentaremos a seguir algumas funções, com exemplos, que serão utilizadas para facilitar o trabalho do dia a dia do profissional de TI, as quais simplificarão os algoritmos daqui por diante.

Exemplos de palavras reservadas:

registro, conjunto, arquivo, fluxo, se, senao, selecao, caso, repita, atequ, enquanto, para, ate, passo, retorne, interrompa, continue.

Símbolos reservados:

+ - * / % ^ := = < - == # > >= < <=
& | ! " ' . () { } [] , ;

Exemplos de **funções de interface**:

Tecla(), Direitos(), CopyRight(), LimpaTela(); Imprima(); Linha(); Coluna(); CorDoTexto(); CorDoFundo()

São exemplos de **funções matemáticas**:

Seno(), Coseno(), Tangente(), ArcoSeno(), ArcoCoseno(), ArcoTangente(), Exponencial(), LogDec(), LogNeper(), Logaritmo()

04

2 - SINTAXE DE ALGUMAS FUNÇÕES DE INTERFACE

Escreva() - A função tem por objetivo realizar a saída de informações para a tela. A função **escreva()** não retorna valor. Quando colocado aspas irá apresentar na tela o texto que estiver dentro. Para apresentar o valor de uma variável deve ser colocado sem aspas. E se quiser mostrar uma mensagem e o valor da variável deve colocar o sinal de vírgula após as aspas - **escreva("nº", n)**.

Veja o exemplo abaixo:

```

início
inteiro maior,menor,res;
escreva("digite o maior número: ");
leia(maior);
escreva("digite o menor número: ");
leia(menor);
enquanto (menor != 0)
    início
        res ← maior mod menor;
        maior ← menor;
        menor ← res;
    fim
escreva("o mdc entre os números é igual a", maior);
fim
  
```

05

Leia() - A função tem por objetivo realizar a entrada de informações para o processador. A função **leia()** não retorna valor. Quando colocado aspas irá apresentar na tela o texto que estiver dentro. Para receber o valor em uma variável, a mesma deve ser colocada sem aspas. E se quiser, com um único comando, mostrar uma mensagem e, ao mesmo tempo, receber o valor em uma variável deve-se colocar o sinal de vírgula após as aspas - **leia("Digite um número", n)**. Observe que esta última forma de uso pode ser substituída pela combinação de dois comandos sendo o primeiro - **escreva("Digite um número")**; - seguido pelo segundo comando que é o - **leia(n)**;

Veja o exemplo abaixo:

```

início
inteiro maior,menor,res;
escreva("digite o maior número: ");
leia(maior);
  
```

```

leia("digite o menor número: ", menor);
enquanto (menor != 0)
  inicio
    res ← maior mod menor;
    maior ← menor;
    menor ← res;
  fim
escreva("o mdc entre os números é igual a", maior);
fim

```

06

Moldura() - A função tem por objetivo fazer um quadro com linhas duplas na tela. A função **moldura()** não retorna valor.

Ela é chamada com a seguinte sintaxe:

Moldura (exprN1, exprN2, exprN3, exprN4); onde exprN1, exprN2, exprN3 e exprN4 são expressões numéricas do tipo inteiro, que representam as coordenadas dos cantos superior esquerdo e inferior direito do quadro - Moldura (L1, C1, L2, C2). As coordenadas são especificadas em pares de números inteiros, onde o primeiro representa a linha e o segundo representa a coluna.

Veja o exemplo abaixo:

```

inicio
  Moldura (4, 5, 16, 60);
  escreva("Bom dia aluno");
fim

```



07

Pausa() - A função tem por objetivo fazer uma interrupção na execução do algoritmo e aguardar a tecla <enter> ser pressionada. A função **pausa()** não retorna valor. A função não recebe letras ou números, somente o parêntese.

Veja o exemplo abaixo:

```

inicio
inteiro maior,menor,res;
escreva("digite o maior número: ");
leia(maior);
escreva("digite o menor número: ");
leia(menor);
enquanto (menor != 0)
  inicio
    res ← maior mod menor;
    maior ← menor;
    menor ← res;
  fim
escreva("aperte a tecla <enter> para continuar"); pausa();
escreva("o mdc entre os números é igual a", maior);
fim
  
```

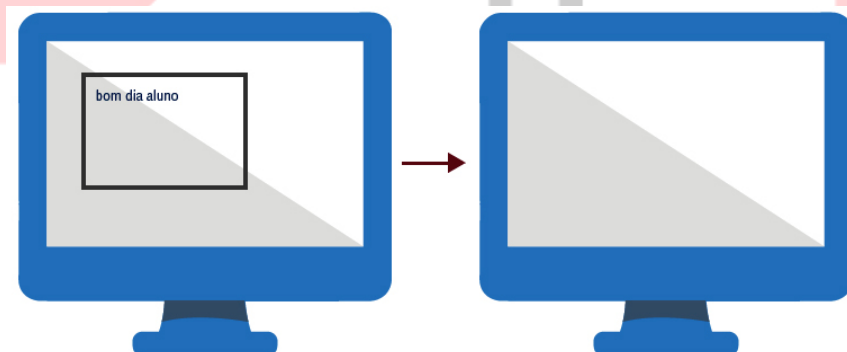
08

LimpaTela() - A função tem por objetivo limpar a tela do vídeo. A função não retorna valor.

Veja o exemplo abaixo:

```

inicio
Moldura (4, 5, 16, 60);
escreva("Bom dia aluno");
LimpaTela();
fim
  
```



09

Imprima() - A função tem por objetivo enviar a saída de informações para a impressora. A função **imprima()** não retorna valor.

Veja o exemplo abaixo:

```

inicio
inteiro maior,menor,res;
escreva("digite o maior número: ");
leia(maior);
leia("digite o menor número: ", menor);
enquanto (menor != 0)
  inicio
    res ← maior mod menor;
    maior ← menor;
    menor ← res;
  fim
imprima("o mdc entre os números é igual a", maior);
fim

```

10

3 - SINTAXE DE ALGUMAS FUNÇÕES MATEMÁTICAS

Sinal() - A função retorna o valor inteiro 1 se o argumento passado à função for positivo e, retorna -1 se o argumento passado à função for negativo. A função é chamada com a sintaxe $S := \text{Sinal}(\text{exprN})$; onde exprN é uma expressão de classe numérica (tipo inteiro ou real) e S é uma variável do tipo inteiro.

Veja o exemplo abaixo:

```

inicio
inteiro s, i, j;
real n;
i ← 0;
j ← 0;
enquanto (n != 0)
  inicio
    escreva("digite qualquer número: ");
    leia(n);
    S := sinal(n);
    Se S = 1
      i = i + 1;
    senão
      j = j + 1
    fim
  fim
imprima("foram digitados", i, "números positivos e ", j, "números negativos");
fim

```

11

ParteInteira() - A função retorna a parte inteira de seu argumento, isto é, os dígitos decimais são ignorados. Ela é chamada com a sintaxe $N := \text{ParteInteira}(\text{exprN})$;

Veja o exemplo abaixo:

```

início
inteiro s;
real n;
escreva("digite qualquer número: ");
leia(n);
S := ParteInteira(n);
imprima("Foi digitado o número", n, "e sua parte inteira é", S);
fim

```

12

ParteDecimal() - função ParteDecimal() retorna a parte decimal de seu argumento, isto é, os dígitos inteiros são ignorados.

Ela é chamada com a sintaxe $N := \text{ParteDecimal}(\text{exprN})$;

Veja o exemplo abaixo:

```

início
inteiro s;
real n;
escreva("digite qualquer número: ");
leia(n);
S := ParteDecimal(n);
imprima("Foi digitado o número", n, "e sua parte decimal é", S);
fim

```

Exemplo com as duas funções:

```

início
inteiro s, p;
real n;
escreva("digite qualquer número: ");
leia(n);
P := ParteDecimal(n);
S := ParteInteira(n);
imprima("Foi digitado o número", n, "e sua parte inteira é", S, "e sua parte decimal é", P);
fim

```

13

Quociente () – a função retorna o quociente entre seus argumentos. O valor de retorno é do tipo real. Ela é chamada com a sintaxe $Q := \text{Quociente}(\text{exprN1}, \text{exprN2})$; onde exprN1 e exprN2 são expressões numéricas e exprN2 deve ser diferente de zero. A função retorna o valor de $\text{exprN1} / \text{exprN2}$.

Veja o exemplo abaixo:

```

inicio
inteiro i, j;
real Q
escreva("digite qualquer número que queira dividir: ");
leia(i);
escreva("digite o denominador da divisão ", j)
  Q := Quociente(i, j)
escreva("O quociente ou o resultado da divisão dos números ", i, "e ", j, "é", Q);
fim
  
```

14

Resto () - a função retorna o resto da divisão inteira entre seus argumentos. O valor de retorno é do tipo inteiro.

Ela é chamada com a sintaxe $R := \text{Resto}(\text{exprN1}, \text{exprN2})$; onde exprN1 e exprN2 são expressões numéricas e exprN2 deve ser diferente de zero. A função retorna o valor do resto de $\text{exprN1} / \text{exprN2}$.

Veja os exemplos abaixo:

```

inicio
inteiro R, i, j;
escreva("digite qualquer número que queira dividir: ");
leia(i);
escreva("digite o denominador da divisão ", j)
  R := Resto(i, j)
escreva("O resto da divisão dos números", i, "e ", j, "é", r);
fim
  
```

Exemplo com as duas funções:

```

inicio
inteiro Q, R, i, j;
escreva("digite qualquer número que queira dividir: ");
leia(i);
escreva("digite o denominador da divisão: ");
  
```



```

leia(j);
Q := Quociente(i, j)
R := Resto(i, j)
escreva("O quociente ou o resultado da divisão dos números ", i, "e ", j, "é", Q);
escreva("O resto da divisão dos números", i, "e ", j, "é", R);
fim

```

15

Raiz() - a função retorna a raiz quadrada de seu argumento. O valor de retorno é do tipo real. Ela é chamada com a sintaxe $R := \text{Raiz}(\text{exprN})$; onde exprN é expressão numérica e exprN deve ser positivo.

Veja o exemplo abaixo:

```

inicio
inteiro n;
real: r;
escreva("Digite um número e descubra qual a sua raiz quadrada: ");
leia(n);
R := Raiz(n);
escreva("A raiz quadrada do número ", n, "é ", R);
fim

```

16

Potencia() - a função retorna a potenciação entre seus argumentos. O valor de retorno é do tipo real. Ela é chamada com a sintaxe $P := \text{Potencia}(\text{exprN1}, \text{exprN2})$; onde exprN1 e exprN2 são expressões numéricas. A função retorna o valor de exprN1 elevado a exprN2 .

Veja o exemplo abaixo:

```

inicio
inteiro n, p, j;
escreva("digite o número que queira elevar, ou potencializar: ");
leia(n);
escreva("digite o número elevado ");
leia(j);
P:= potencia(n,j);
escreva("O resultado da potência foi", P);
fim

```

RESUMO

Nessa Unidade evoluímos na arte da construção de algoritmos. Começamos a tratar das funções, que ajudam e simplificam em muito os algoritmos e futuramente a programação. Vimos as diferentes classes de funções:

Palavras reservadas - relacionadas com a definição de tipos de dados e também com as estruturas de controle do fluxo; as palavras reservadas não podem ser utilizadas, pois já possuem as suas próprias características.

Funções de interface - são as instruções que nos permitem realizar um diálogo entre nosso algoritmo e o usuário.

Funções matemáticas – instruções que nos permitem utilizar de recursos matemáticos frequentemente solicitados na resolução de algum tipo de problema. Com as funções matemáticas tornou mais fácil a forma de realizar cálculos.

UNIDADE 3 – ESTRUTURAS DE REPETIÇÃO, FUNÇÕES, RECURSIVIDADE, FILA E PILHA MÓDULO 2 – FUNÇÕES DE CADEIA E ARQUIVOS

1 – CONCEITOS

Depois de estudarmos as palavras reservadas, as funções de interface e as funções matemáticas, continuaremos, neste módulo, a estudar as funções. Veremos agora outras igualmente importantes para o dia a dia, que são **funções de cadeia** e **funções de arquivos**.

As funções em cadeia tratam das variáveis do tipo literal e caractere. Posteriormente aprenderemos o uso de vetores e matrizes com as funções de cadeia, portanto, é importante que você compreenda bem o conteúdo deste módulo, pois servirá de apoio para o estudo dos próximos módulos.

Estudaremos agora as seguintes funções:

Funções de cadeia	As funções de cadeia são as instruções que nos permitem tratar sequência de caracteres. As cadeias são recursos bastante utilizados na programação em geral.	São exemplos de funções de cadeia: Comprimento() , Concatena() , SubCadeia() , Primeiros() , Ultimos() . Veja mais exemplos
Funções de arquivos	As funções para o tratamento de arquivo são as instruções que nos permitem tratar informações que poderão ser armazenadas indefinidamente no computador, mas necessariamente em arquivos.	São exemplos de funções de arquivos: Vincular() , Associar() , Criar() , Abrir() , Fechar() . Veja mais exemplos

Apresentaremos a seguir algumas funções, com exemplos, que serão utilizadas para facilitar o trabalho do dia a dia do profissional de TI. Funções que simplificarão os algoritmos estudados nos dois primeiros módulos.

São exemplos de **funções em cadeia**:

Compara(), Maiusculas(), Minusculas(), Insere(), SobrePoe(), CadeiaParaInteiro(), CadeiaParaReal(), InteiroParaCadeia(), RealParaCadeia().

São exemplos de **funções de arquivos**:

Gravar (), Regravar (), Deletar (), Ler (), Posicionar (), FDA ()

02

2 - SINTAXE DE ALGUMAS FUNÇÕES DE CADEIA

Comprimento() – a função retorna o comprimento do conteúdo em quantidade de caracteres. O valor retornado pela função é do tipo inteiro. O conteúdo está dentro dos parênteses (conteúdo).

Ela é chamada com a seguinte sintaxe:

C:= Comprimento (conteúdo); onde conteúdo é uma expressão do tipo cadeia (literal) e C representa uma variável qualquer, de classe numérica, que receberá o valor retornado pela função.

Veja o exemplo abaixo:

```

inicio
inteiro C;
literal nome;
escreva("digite o seu nome: ");
leia(nome);
c:= Comprimento (nome);
```

```

escreva("seu nome possui ", c, "letras");
fim

```

03

Concatena() - a função retorna os seus conteúdos concatenados em uma única cadeia, sem espaços entre eles. O valor retornado pela função é do tipo cadeia (literal).

Ela é chamada com a seguinte sintaxe:

A:= Concatena(conteúdo, conteúdo1,...); onde conteúdo é uma expressão do tipo cadeia (literal) e A representa uma variável qualquer, do tipo cadeia, que receberá o valor retornado pela função. As reticências significam que a função pode receber vários conteúdos, sendo eles separados por vírgulas.

Veja o exemplo abaixo:

```

inicio
literal nome, sobrenome, c;
escreva("digite o seu primeiro nome: ");
leia(nome);
escreva("digite o seu sobrenome: ");
leia(sobrenome);
c := Concatena(nome, sobrenome)
imprima("Bom dia ", c);
fim

```

04

SubCadeia() - a função retorna um conteúdo de seu sub conteúdo. O valor retornado pela função é do tipo cadeia (literal).

Ela é chamada com a seguinte sintaxe:

L := SubCadeia(contéudo, N1, N2); onde conteúdo é uma expressão do tipo cadeia da qual se deseja uma sub conteúdo, N1 é um valor ou resultado de uma expressão do tipo inteiro que representa a posição do primeiro caractere do conteúdo e, N2 representa a quantidade de caracteres que deseja capturar do conteúdo.

Veja o exemplo abaixo:

```

inicio
literal nome, sobrenome, N, S ;
escreva("digite o seu primeiro nome: ");
leia(nome);
N := SubCadeia(nome, 1, 1)
escreva("digite o seu sobrenome: ");

```

```

leia(sobrenome);
S := SubCadeia(sobrenome, 1, 1)
imprima("Bom dia", nome, "seu sobrenome é", sobrenome, "suas iniciais são ", N, S);
fim

```

05

Primeiros() - a função retorna uma subcadeia de seu conteúdo. O valor retornado pela função é do tipo cadeia (literal).

Ela é chamada com a seguinte sintaxe:

S:= Primeiros (Conteúdo, N); onde Conteúdo é uma expressão do tipo cadeia da qual se deseja uma subcadeia, N é uma número ou resultado de uma expressão do tipo inteiro que representa a quantidade de caracteres do Conteúdo, contados a partir de seu início, que se deseja retornar.

Veja o exemplo abaixo:

```

início
literal nome, N;
escreva("digite o seu primeiro nome: ");
leia(nome);
N := Primeiros(nome, 4)
imprima("As 4 primeiras letras iniciais do seu nome são", N);
fim

```

06

Ultimos() - a função retorna uma subcadeia de seu conteúdo. O valor retornado pela função é do tipo cadeia (literal).

Ela é chamada com a seguinte sintaxe:

N := Ultimos(Conteúdo, m); onde Conteúdo é uma expressão do tipo cadeia da qual se deseja uma subcadeia, m é um número ou uma expressão do tipo inteiro que representa a quantidade de caracteres do Conteúdo, contados a partir de seu final, que se deseja retornar.

Veja o exemplo abaixo:

```

início
literal nome, N;
escreva("digite o seu primeiro nome: ");
leia(nome);
N := Ultimos(nome, 2)

```

```
imprima("As duas letras finais do seu nome são", N);
fim
```

07

Compara() - a função tem por objetivo comparar duas cadeia (duas variáveis literais) com relação a ordem alfabética de seus conteúdos. Ela retorna um valor inteiro, negativo, nulo ou positivo de acordo com a ordem de passagem de seus conteúdos.

Ela é chamada com a seguinte sintaxe:

$S := \text{Compara}(\text{nome1}, \text{nome2})$; onde nome1 e nome2 são expressões do tipo cadeia (literal). Se os valores representados pelas expressões nome1 e nome2 forem iguais o valor de retorno é 0 (zero), se nome2 for maior que nome1, isto é, nome2 representar um valor que está em uma posição à frente de nome1, considerando a ordem alfabética, o valor de retorno será 1 (um), caso contrário o valor de retorno será -1. Assim sendo, o valor de retorno será negativo se a ordem dos argumentos estiver diferente da ordem alfabética dos mesmos.

Veja o exemplo abaixo:

```
início
inteiro N;
literal nome, nome1;
escreva("digite o primeiro nome: ");
leia(nome);
escreva("digite o segundo nome: ");
leia(nome1);
N := Compara(nome, nome1)
início
  Se n = 0
    Imprima("nomes iguais");
  Se n = 1
    Imprima("Primeiro nome na ordem alfabética", nome)
  Se n = -1
    Imprima("Primeiro nome em ordem alfabética", nome1)
fim
fim
```

**08**

Maiusculas() - a função tem por objetivo retornar o seu conteúdo transformado em maiúsculas. O valor de retorno é do tipo cadeia (literal).

Ela é chamada com a seguinte sintaxe:

M := Maiusculas(nome); onde nome é uma expressão do tipo cadeia (literal).

Veja o exemplo abaixo:

```

início
literal nome, N;
escreva("digite o seu nome: ");
leia(nome);
N := Maiusculas(nome)
imprima("Bom dia, seu nome com caracteres em maiúsculo é ", nome);
fim

```

09

Minusculas() - a função tem por objetivo retornar o seu conteúdo transformado em minúsculas. O valor de retorno é do tipo cadeia (literal).

Ela é chamada com a seguinte sintaxe:

N := Minusculas(nome); onde nome é uma expressão do tipo cadeia (literal).

Veja o exemplo abaixo:

```

início
literal nome, N;
escreva("digite o seu nome: ");
leia(nome);
N := Minusculas(nome)
imprima("Bom dia, seu nome com caracteres em minúsculo é ", nome);
fim

```

10

CadeiaParaInteiro() - a função tem por objetivo retornar um numérico do tipo inteiro correspondente a cadeia numérica de seu conteúdo.

Ela é chamada com a seguinte sintaxe:

V := CadeiaParaInteiro(conteúdo); onde conteúdo é uma expressão do tipo cadeia (literal). Se a cadeia especificada em conteúdo não puder ser convertida em inteiro, o valor de retorno será 0 (zero) e V representa uma variável do tipo inteiro.

Veja o exemplo abaixo:

```

início
inteiro V;

```

```

literal N;
escreva("digite um número: ");
leia(N);
V := CadeiaParaInteiro(N)
Se V = 0
  inicio
    escreva("Número não inteiro")
  senão
    escreva("O número transformado para inteiro é: ", N);
  fim
fim

```

11

3 - SINTAXE DE ALGUMAS FUNÇÕES DE ARQUIVO

Vincular () - a função tem por objetivo estabelecer uma ligação entre um identificador lógico que será utilizado no texto do algoritmo e o correspondente identificador físico do arquivo em disco.

Ela é chamada com a seguinte sintaxe:

Vincular (nome arquivo, nome arquivo); onde nome arquivo representa o nome lógico do arquivo utilizado no algoritmo e, nome arquivo representa o nome físico do arquivo no disco.

Veja o exemplo abaixo:

```

inicio
literal clientes ;
Criar (clientes);
Criar ("arq001.dat");
Vincular(clientes, "arq001.dat");
Abrir (clientes);
Gravar (clientes, "teste de gravação 1");
Gravar (clientes, "teste de gravação 2");
Posicionar (clientes, INICIO);
Gravar (clientes, "teste de gravação 0");
Posicionar (clientes, FINAL);
Gravar (clientes, "teste de gravação 3");
Fechar (clientes);
fim

```

12

Criar () - a função tem por objetivo criar o arquivo físico no disco.

Ela é chamada com a seguinte sintaxe:

Criar (nome arquivo); onde nome arquivo representa o identificador lógico do arquivo ou o seu identificador físico.

Veja o exemplo abaixo:

```

inicio
literal clientes ;
Criar (clientes);
Criar ("arq001.dat");
Vincular (clientes, "arq001.dat");
Abrir (clientes);
Gravar (clientes, "teste de gravação 1");
Gravar (clientes, "teste de gravação 2");
Posicionar (clientes, INICIO);
Gravar (clientes, "teste de gravação 0");
Posicionar (clientes, FINAL);
Gravar (clientes, "teste de gravação 3");
Fechar (clientes);
fim
  
```

Abrir () - a função tem por objetivo abrir o arquivo físico no disco.

Ela é chamada com a seguinte sintaxe:

Abrir (nome arquivo); onde nome arquivo representa o identificador lógico do arquivo ou o seu identificador físico.

Veja o exemplo abaixo:

```

inicio
literal clientes;
Criar (clientes ;
Criar ("arq001.dat");
Vincular (clientes, "arq001.dat");
Abrir (clientes);
Gravar (clientes, "teste de gravação 1");
Gravar (clientes, "teste de gravação 2");
Posicionar (clientes, INICIO);
Gravar (clientes, "teste de gravação 0");
Posicionar (clientes, FINAL);
  
```

13

```
Gravar (clientes, "teste de gravação 3");
Fechar (clientes);
fim
```

14

Fechar () - a função tem por objetivo fechar o arquivo físico no disco.

Ela é chamada com a seguinte sintaxe:

Fechar (nome arquivo); onde nome arquivo representa o identificador lógico do arquivo ou o seu identificador físico.

Veja o exemplo abaixo:

```
inicio
literal clientes;
Criar (clientes);
Criar ("arq001.dat");
Vincular (clientes, "arq001.dat");
Abrir (clientes);
Gravar (clientes, "teste de gravação 1");
Gravar (clientes, "teste de gravação 2");
Posicionar (clientes, INICIO);
Gravar (clientes, "teste de gravação 0");
Posicionar (clientes, FINAL);
Gravar (clientes, "teste de gravação 3");
Fechar (clientes);
fim
```



15

Gravar () - a função tem por objetivo gravar as informações no arquivo físico.

Ela é chamada com a seguinte sintaxe:

Gravar (nome arquivo algoritmo, nome arquivo); onde nome arquivo algoritmo representa o identificador lógico ou físico do arquivo e nome arquivo representa a informação que se deseja armazenar.

Veja o exemplo abaixo:

```
inicio
literal clientes;
Criar (clientes);
```

```

Criar ("arq001.dat");
Vincular (clientes, "arq001.dat");
Abrir (clientes );
Gravar (clientes, "teste de gravação 1");
Gravar (clientes, "teste de gravação 2");
Posicionar (clientes, INICIO);
Gravar (clientes, "teste de gravação 0");
Posicionar (clientes, FINAL);
Gravar (clientes, "teste de gravação 3");
Fechar (clientes);
fim

```

16

Posicionar () - a função tem por objetivo permitir o posicionamento em uma determinada informação no arquivo físico.

Ela é chamada com a seguinte sintaxe:

Posicionar (nome, pos); onde nome representa o identificador lógico ou físico do arquivo e pos a posição em que se deseja apontar no arquivo. Pode ser INÍCIO e FIM.

Veja o exemplo abaixo:

```

inicio
literal cliente;
Criar (clientes );
Criar ("arq001.dat");
Vincular (clientes, "arq001.dat");
Abrir (clientes );
Gravar (clientes, "teste de gravação 1");
Gravar (clientes, "teste de gravação 2");
Posicionar (clientes, INICIO);
Gravar (clientes, "teste de gravação 0");
Posicionar (clientes, FINAL );
Gravar (clientes, "teste de gravação 3");
Fechar (clientes);
Fim

```

17

RESUMO

Nesse módulo continuamos o estudo das funções. Vimos funções de cadeia e funções de arquivos. As funções de cadeia são as instruções que nos permitem tratar sequência de caracteres. As cadeias são recursos bastante utilizados na programação em geral. As funções para o tratamento de arquivo são as instruções que nos permitem tratar informações que poderão ser armazenadas indefinidamente no computador, mas necessariamente em arquivos.

UNIDADE 3 – ESTRUTURAS DE REPETIÇÃO, FUNÇÕES, RECURSIVIDADE, FILA E PILHA

MÓDULO 3 – PONTEIROS E RECURSIVIDADE

01

1 - PONTEIROS

Para entender o que são os ponteiros, imagine a seguinte situação: você trabalha no departamento de Recursos Humanos de uma empresa há muito tempo, desde a sua criação, portanto, você sabe exatamente em que local as pessoas trabalham e o que fazem. Assim, quando chega alguma encomenda ou uma correspondência na empresa, sempre acabam recorrendo a você, que serve de referência, apontando exatamente onde encontrar a pessoa que estão procurando.

Os ponteiros possuem exatamente a mesma ideia, ou seja, são eles que possuem o poder de armazenar a localização onde estão guardadas determinadas informações.

Falando em termos de algoritmos, podemos entender um ponteiro como uma variável que armazena um endereço de memória onde está localizado um valor.

02

Utilidades dos ponteiros

Os ponteiros são utilizados em diversas situações. Podemos citar, por exemplo, a atualização de valores que são utilizados em várias partes dos algoritmos e que, por algum motivo, precisam ser atualizados ao longo da vida útil do mesmo. Para solucionar esta situação, basta referenciar-se estes valores em uma função e aplicar sobre eles operações que alterem seus valores de forma pertinente e a replicação ocorre automaticamente.

Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está "apontando" para essa variável.

Na imagem abaixo apresenta as variáveis *i*, *f* e *j* com os seus respectivos valores 2450, 225.345 e 11331, e os endereços de cada variável respectivamente 1002, 1004 e 1005.

Endereço	Conteúdo	Variável
⋮	⋮	⋮
1001	???	
1002	2450	i
1003	???	
1004	225.345	f
1005	11331	j

03

Importante para o início da utilização

Para utilizarmos o ponteiro em nossos algoritmos devemos conhecer o seu comando, ou forma de representar em um pseudocódigo.

Os símbolos abaixo são utilizados nessa representação de utilização dos ponteiros.

& significa “endereço de”

***** significa “conteúdo do endereço para onde aponta”

Quando declaramos **int i=240**, estamos declarando para a memória reservar um espaço para valores do tipo inteiro. Simultaneamente é atribuído o valor 240 à variável i, o que significa que no endereço de memória associado à variável i é armazenado o valor 240.

Como dito anteriormente, o uso de ponteiros pode ser justificado quando é necessária a utilização da variável em outros momentos dos algoritmos. Neste caso, poderiam existir vários apontadores ou ponteiros espalhados pelo programa, apontando para essa variável, que daria o conteúdo desejado (a informação). Toda e qualquer modificação no dado seria feita no conteúdo da variável a que todos os apontadores fariam a referência, ou seja, diferentes partes do programa sempre estariam acessando um valor de dado atualizado.

Ponteiros ou apontadores são variáveis que armazenam um endereço de memória (isto é, o endereço de outras variáveis). Eles fornecem um mecanismo para acessar diretamente e objetos e código de modificação na memória. Os ponteiros são utilizados em muitas linguagens de programação, cada ponteiro tem um tipo, podendo ser um tipo para cada tipo de linguagem ou um tipo definido pelo programador. Dessa forma, podemos ter ponteiros para inteiros, ponteiros para real, ponteiros para manipular cadeias de caracteres, para passar parâmetros para funções, manipulação matrizes de dados e criação de listas ligadas e outras estruturas de dados complexas.

04

O acesso direto na memória proporciona uma melhora no desempenho da sua aplicação. Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está "apontando" para essa

variável. Uma função a ser passada como um parâmetro para outra função. Um ponteiro de função pode ser atribuído o endereço de uma das opções de funções, de modo que o ponteiro atua como uma espécie de apelido. Linguagens de programação orientadas a objetos eliminaram a necessidade de ponteiros de função com herança e polimorfismo.

Estes são exemplos de declarações de tipos de ponteiro:

Exemplo	Descrição
<code>int* : p</code>	p é um ponteiro para um inteiro.
<code>int** : P</code>	p é um ponteiro para um ponteiro para um inteiro.
<code>int* [] : p</code>	p é uma matriz unidimensional de ponteiros para inteiros.
<code>char* : p</code>	p é um ponteiro para um caractere.
<code>void* : p</code>	p é um ponteiro para um tipo desconhecido.

O operador do ponteiro “*” pode ser usado para acessar o conteúdo no local apontado pela variável de ponteiro. Por exemplo, considere a seguinte declaração:

Exemplo em pseudocódigo de função:

```

Inteiro *pont;
Inteiro x
início
x ← 10
*pont ← &x
Escreva("O endereço de memória da variável ", x "é", &x);
Escreva("O valor da memória apontada é ", pont*);
Fim

```

No trecho acima simplesmente foi criado um ponteiro, alocando a memória para ele, logo após atribuindo um valor à variável e finalizando com a escrita na tela.

05

2 - RECURSIVIDADE (OU RECURSÃO)

Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de definição circular.

Pode-se dizer que o conceito de algo recursivo está dentro de si, que por sua vez está dentro de si e assim sucessivamente, infinitamente, ou seja, se não tiver um mecanismo de saída ou de parada o algoritmo entrará em *loop*.

O termo *loop* é usado na TI para coisas que ficam girando ou rodando sem fim.

O exemplo a seguir mostra a foto de um espelho de frente a outro espelho, onde representa a recursividade no dia a dia.



06

Vamos examinar um exemplo coloquial de procedimento recursivo. Imaginemos ter que encontrar no dicionário o significado da palavra "jururu". É claro que ninguém vai ler sequencialmente o dicionário até chegar à palavra procurada.

Vamos definir o processo de achar uma palavra num dicionário em termos algorítmicos mais ou menos livres procura-palavra (Dicionário, palavra-procurada):

```

abrir o dicionário nas palavras que iniciam com a letra "j"
ler sequencialmente duas páginas
se palavra procurada estiver nas páginas
  fim da procura
se palavra procurada não estiver nas páginas
  procura palavra nas próximas páginas
  se palavra-procurada for encontrada
    imprimir a sua definição fim do algoritmo
  fim se
se palavra-não for encontrada
  imprimir ("palavra procurada não existe")fim do algoritmo
fim se
fim se
fim da leitura sequencial
fim
  
```

O algoritmo é recursivo no sentido de que ele chama a ele mesmo. Note que ocorre a procura da palavra várias vezes até encontrá-la, ou, caso depois da procura não se encontre o que deseja, ele encerra o algoritmo. Nesse caso, o **processo é finito**, já que, a cada chamada, o universo de pesquisa é

menor. Na primeira vez, ele é todo o dicionário, na segunda vez, apenas a metade, na terceira vez, uma quarta parte e assim por diante. Finalmente, há uma condição que estabelece o fim da pesquisa.

No exemplo do dicionário, vimos às três **características que um problema recursivo** tem que ter:

1. Um processo que chame a si mesmo.
2. A garantia de que a cada chamada, o universo de trabalho do processo será "menor".
3. Uma condição, que obrigatoriamente ocorrerá, que indique quando terminar.

07

Definições como “recursividade” são normalmente encontradas na matemática. O grande apelo que o conceito da recursão traz é a possibilidade de dar uma definição infinita para um conjunto que pode ser finito. Um bom exemplo é o cálculo do fatorial.

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ n! &= n.(n-1)...3.2.1 \end{aligned}$$

Você já deve ter estudado, na matemática, a expressão fatorial: **o fatorial de um número natural n é o produto de todos os inteiros positivos menores ou iguais a n**. Isso é escrito como $n!$ e lido como "fatorial de n". A notação $n!$ foi introduzida por Christian Kramp em 1808.

O exemplo abaixo demonstra a fórmula do fatorial de qualquer número, onde n é um número inteiro positivo:

$$F(n) = 1 \text{ se } n = 0 \text{ ou } n = 1$$

$$F(n) = n.F(n-1), \text{ se } n > 1$$

Esta propriedade é chamada de **propriedade recursiva**: o fatorial de um número pode ser calculado através da multiplicação do número pelo fatorial de seu antecessor.

08

Ora, podemos utilizar esta propriedade para escrevermos uma rotina recursiva para o cálculo de fatorial.

Vejamos: $F(n) = n.F(n-1)$, se $n > 1$

$$F(4) = 4.F(4-1)$$

$$F(3) = 3.F(3-1)$$

$$F(2) = 2.F(2-1)$$

$$F(1) = 1.F(1-1)$$

Temos então a fórmula, que é facilmente verificável:

$$n! = n \cdot (n-1)!$$

Fatorial (4) = 4 * fatorial (4 - 1)
 = 4 * fatorial (3)
 = 4 * 3 * fatorial (3 - 1)
 = 4 * 3 * fatorial (2)
 = 4 * 3 * 2 * fatorial (2-1)
 = 4 * 3 * 2 * fatorial (1)
 = 4 * 3 * 2 * 1 * fatorial (1-1)
 = 4 * 3 * 2 * 1 * fatorial (0)
 = 4 * 3 * 2 * 1 * 1
 =24



09

Na computação o conceito de recursividade é amplamente utilizado, mas deve-se tomar cuidado em sua utilização, pois necessita de uma condição para provocar o fim do ciclo recursivo. Essa condição deve existir, pois, devido às limitações técnicas que o computador apresenta, a recursividade é impedida de continuar eternamente.

Nos anos 90 grandes CPD (Centro de Processamento de Dados) possuíam salas com impressoras que atendiam a várias equipes. Essas impressoras imprimiam 10 a 20 folhas por segundo e um programa que entrava em *loop*, ou seja, em uma função recursiva sem saída, gerava milhares de páginas de erros do programa. Como funcionava? O analista construía ou alterava um sistema e mandava rodar e imprimir logo a seguir para ver o resultado. O técnico da sala de impressão ligava imediatamente quando dava erro na impressão, algumas vezes tarde demais, e muitas folhas de papel iam para o lixo.

10

3 - VANTAGENS E DESVANTAGENS DA RECURSÃO

 VANTAGENS	 DESVANTAGENS
<ul style="list-style-type: none"> • Simplifica a solução de alguns problemas. • Geralmente, um código com recursão é mais conciso. • Caso não seja usado, em alguns problemas, é necessário manter o controle das variáveis manualmente, que pode gerar um pouco de confusão, quando há muitas variáveis. 	<p>A recursividade deixa as rotinas dos programas mais lentas que funções sequenciais, pois podem ocorrer muitas chamadas do programa consecutivas a funções recursivas.</p> <p>A falha na implementação pode levar a um erro na aplicação, pode estourar a memória do servidor ou até mesmo não ter resposta nenhuma da solicitação executada no programa.</p> <p>Uma falha comum na implementação incorreta da recursividade é quando o sistema entra em loop. O loop ocorre quando a recursividade não tem saída, fica girando o programa na mesma rotina sem ocorrer nada aparente. O loop pode provocar falhas na aplicação e até mesmo derrubar servidores, perdas de informações, estouro de memórias, dentre outros problemas.</p>

Muito importante realizar testes nas aplicações que estão sendo implantadas, principalmente nos programas com rotinas recursivas. Os testes evitam problemas com códigos que não estão funcionando adequadamente.

11

Exemplo de Recursividade utilizando Linguagem algorítmica

```
#incluir <biblioteca>
principal()
início
    inteiro i, num, fat;
    fat ← 1;
    escreva("digite um numero: ");
    leia(num);
    se (num = 0)
        escreva("o fatorial de", num, "é", fat);
    senão
        início
            para(i=1; i<=num ; i ← i+1)
                fat = fat*i;
            escreva("o fatorial de", num, "é", fat);
        fim
    fim
fim
```

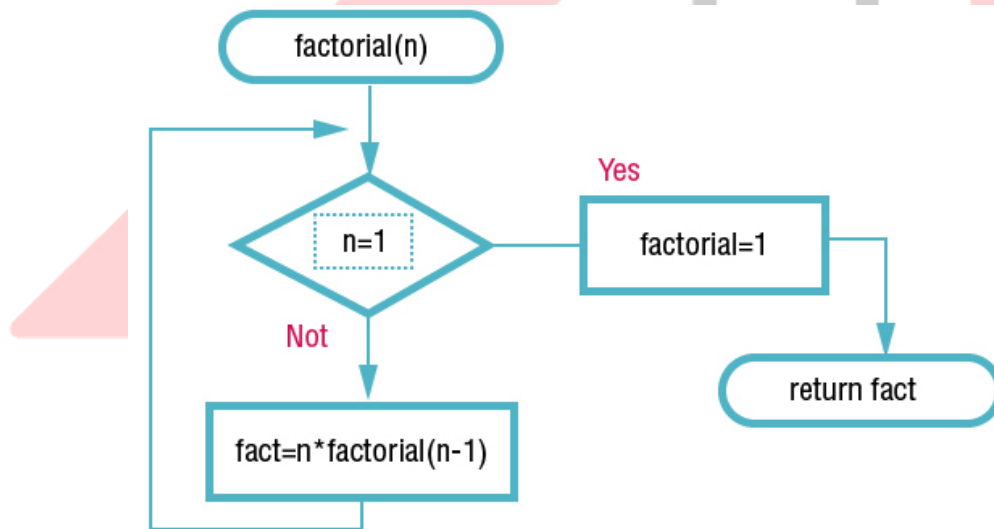
12

Exemplo de recursividade utilizando teste de mesa

Valor de Entrada	
num=8	
i	fat
1	fat=1*1
2	fat=1*2
3	fat=2*3
4	fat=6*4
5	fat=24*5
6	fat=120*6
7	fat=720*7
8	fat=5040*8
Valor de Saída	
fat=40320	

13

Exemplo de fatorial utilizando fluxograma



14

RESUMO

Nesse módulo vimos mais alguns recursos que podemos usar em algoritmos e consequentemente na programação, ponteiros e recursividade, que facilitam a construção dos algoritmos mais complexos e mais difíceis.

Podemos entender um ponteiro como uma variável que armazena um endereço de memória onde está localizado um valor. Os ponteiros ajudam a entender como são armazenadas informações na memória, como informações de arquivos dos sistemas.

Recursão ou recursividade é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de definição circular. A recursividade apresenta três características fundamentais:

1. Um processo que chame a si mesmo.
2. A garantia de que a cada chamada, o universo de trabalho do processo será "menor".
3. Uma condição, que obrigatoriamente ocorrerá, que indique quando terminar.

A recursividade está nos algoritmos e no nosso dia a dia, tudo o que se repete com certa constância é chamado de recursivo.

UNIDADE 3 – ESTRUTURAS DE REPETIÇÃO, FUNÇÕES, RECURSIVIDADE, FILA E PILHA

MÓDULO 4 – LISTA, FILA E PILHA

01

1. LISTA

Vamos supor que você tenha feito uma lista de compras, ou seja, tem um pedaço de papel com diversos itens, organizados conforme a ordem em que você ia lembrando o que precisava comprar.

Provavelmente essa ordem é a que você seguirá para efetuar as compras, do primeiro item até o último.

Se você é uma pessoa criteriosa, deve ter feito a lista segundo uma sequência lógica, listando os itens conforme o setor: produtos de limpeza, alimentícios etc. Já no mercado, você vai verificar a lista novamente, seguir a ordem dela, marcar o que comprou ou não, anotar os preços e verificar se comprou algo a mais do previsto. A lista de compras, portanto, é um modo de organização, sem a qual você correria o risco de comprar supérfluos ou deixar de comprar algo essencial.



No mundo da computação, alguns tipos de dados podem ser organizados em uma cadeia, seja de números ou de caracteres, que denominamos de **lista**. É uma estrutura de dados abstrata que implementa uma coleção ordenada de valores, onde o mesmo valor pode incidir mais de uma vez. Em suma:

Uma **lista** é um conjunto ordenado de dados, geralmente do mesmo tipo e essa organização é feita através da enumeração dos dados para melhor visualização da informação.

02

Inúmeros tipos de dados podem ser representados por listas. Suponhamos, por exemplo, que você precise fazer um programa que cadastre um número indefinido de pessoas. Para isso, você não irá criar uma matriz ou vetor, pois mesmo que você crie 1000 posições, isso irá consumir muita memória, desde o início do processo, sem contar a possibilidade de se ter 900 pessoas, ou até mesmo 1500 pessoas, ou seja, o valor é variável. Nesse caso, o ideal é criar uma lista encadeada, que é manipulada de forma dinâmica. Outros exemplos de sistemas de informação que podem utilizar listas são: informações sobre os funcionários de uma empresa, notas de alunos, itens de estoque etc.

As listas podem ser implementadas de várias maneiras, tanto em uma linguagem de programação procedural como em uma linguagem de programação funcional.

As listas são estruturas dinâmicas, em oposição aos vetores (arrays), os quais são estruturas estáticas e contém um conjunto específico de dados. Virtualmente, uma lista pode conter infinitos itens. As operações sobre listas são limitadas ao tipo de dados que uma lista contém. Por exemplo, listas de dados numéricos podem ser tratadas matematicamente, enquanto listas de caracteres podem ser ordenadas, concatenadas. O conjunto de números [20, 13, 42, 23, 54] é uma lista numérica de inteiros; ['a', 'd', 'z', 'm', 'w', 'u'] é uma lista de caracteres; e ["maçã", "abacaxi", "pera"] é uma lista de frutas. Sendo que $n \geq 1$, n_1 é o primeiro item da lista e x_n o último item da lista.

As listas são importantes, pois constituem uma forma simples de interligar os itens de um conjunto, agrupando informações referentes a um conjunto de elementos que se relacionam entre si de alguma forma.

As listas são úteis em aplicações tais como manipulação simbólica, em que se manipulam as variáveis em expressões algébricas, a gerência de memória, que define as prioridades da memória do computador ou outro dispositivo, simulação, e onde é possível juntar informações sequenciais ou dinâmicas para realizar uma simulação e compiladores, que gravam uma sequência de procedimentos para transformar o programa em um software executável.

Lista encadeada

Uma lista ligada ou lista encadeada é uma estrutura de dados linear e dinâmica. Ela é composta por células que apontam para o próximo elemento da lista. Para "ter" uma lista ligada/encadeada, basta guardar seu primeiro elemento, e seu último elemento aponta para uma célula nula. O esquema a

seguir representa uma lista ligada/encadeada com 5 elementos:

Célula 1 ----> Célula 2 ---> Célula 3 ---> Célula 4 ---> Célula 5 ---> (Nulo)

Programação procedural

Programação utilizada nas linguagens mais antigas, como o COBOL, onde possui estruturas que são chamadas de procedimento ou procedure.

Programação funcional

Programação mais atual, como o ASP, PHP, etc, em que se utilizam funções para montar um aplicativo. Como exemplo, função para validar CPF, função para buscar endereço pelo CEP, etc.

Estruturas dinâmicas

Uma lista mutável ou dinâmica pode permitir que itens sejam inseridos, substituídos ou excluídos durante a existência da lista.

Estruturas estáticas

As estruturas de lista estáticas permitem apenas a verificação e enumeração dos valores.

Manipulação simbólica

A manipulação de símbolos é o manuseio da simbologia da matemática, serve assim para resolver operações algébricas, é o manuseio das variáveis nas expressões matemáticas.

Simulação

É a técnica de estudar o comportamento e reações de determinados sistemas através de modelos. A simulação computacional de sistemas, ou apenas simulação, consiste na utilização de certas técnicas matemáticas, empregadas em computadores, as quais permitem imitar o funcionamento de, praticamente qualquer tipo de operação ou processo do mundo real, ou seja, é o estudo do comportamento de sistemas reais através do exercício de modelos.

Compiladores

Compilador é um programas que traduz o código fonte de uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível (por exemplo, Assembly ou código de máquina). Contudo alguns autores citam exemplos de compiladores que traduzem para linguagens de alto nível como C.

Características das Listas

As listas possuem operações que possibilitam a sua manipulação. Essa manipulação consiste em operações para inserir, retirar e localizar itens em qualquer momento que for necessário. Algumas operações influenciam no tamanho da lista, possibilitando o seu crescimento ou sua diminuição.

Outra característica é que as listas possibilitam também a concatenação de duas ou mais listas e também a divisão de uma lista em mais listas. As listas podem ser adequadas quando não é possível prever o tamanho da memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível.

Pode-se destacar, também, que cada elemento da lista possui um índice, um número que identifica cada elemento da lista. Usando o índice de um elemento da lista é possível buscá-lo ou removê-lo.

Tamanho da lista

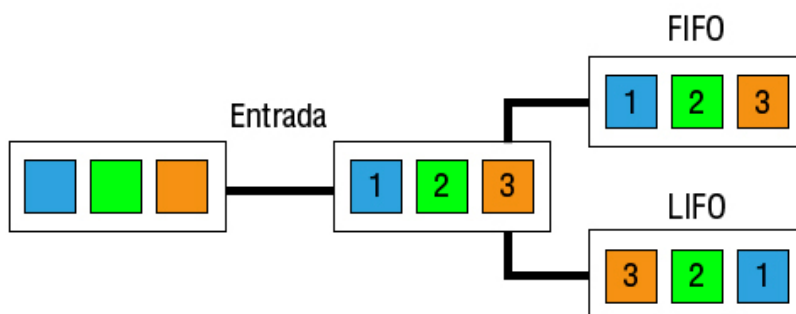
Quando falamos em “tamanho da lista”, estamos nos referindo ao número de elementos presentes na lista.

04

Classificação das Listas

Vimos então que podemos definir uma lista como um conjunto ordenado de elementos, geralmente do mesmo tipo de dado. As listas podem ser classificadas em pilhas e filas, dependendo do tipo de acesso que pode ser feito aos dados que ela contém.

- Pilhas** → São também chamadas de LIFO (Last In, First Out), ou seja, última que entra, primeira que sai.
- Filas** → São também chamadas de FIFO (First In, First Out), ou seja, primeira que entra, primeira que sai.



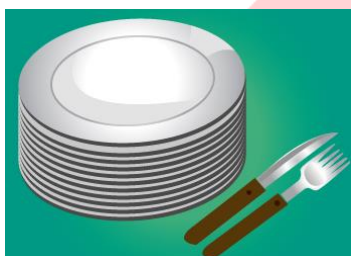
05

2 - PILHA

É um tipo especial de lista, onde os elementos são ordenados como um empilhamento de dados, denominados pilha.

As pilhas representam uma das estruturas mais importantes no processamento de dados. Embora sua estrutura seja bastante simples, as pilhas desempenham um importante papel nas linguagens de programação, pois são utilizadas como acumuladores em laços do programa e em chamadas de sub-rotinas ou funções.

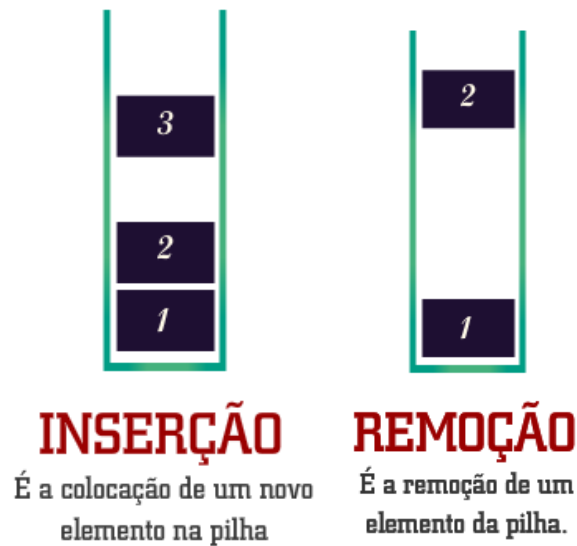
Uma pilha pode ser definida como um conjunto ordenado de dados, no qual se pode inserir e retirar dados apenas em uma determinada ordem, ou seja, em uma posição denominada “topo da pilha”.



O exemplo acima apresenta um conjunto de pratos, ou seja, uma pilha de pratos, onde se tira normalmente o primeiro prato até chegar ao último, que foi o primeiro a ser colocado.

06

As pilhas são estruturas muito comuns em algoritmos e o seu processo de atualização se divide principalmente em duas partes:



Como visto anteriormente, o topo da pilha, que é a extremidade, serão inseridos e retirados os itens do mesmo tipo. A pilha também é chamada de **lista linear**, onde todas as inserções e eliminações são feitas em apenas uma das extremidades. As imagens acima mostram a representação de uma pilha. Primeiramente inserindo itens e depois eliminando itens.

07

A estrutura de dados do tipo pilha tem como característica que a **última informação a entrar é a primeira a sair** (LIFO - last in first out). A aplicação da estrutura de pilhas é mais frequente em compiladores e sistemas operacionais, que a utilizam para controle de dados e alocação de variáveis na memória. O problema no uso de pilhas é **controlar o final da pilha**. Isto pode ser feito de várias formas, sendo a mais indicada criar um método para verificar se existem mais dados na pilha para serem retirados.

Antes de programar a solução de um problema que usa uma pilha, é necessário determinar como representar uma pilha usando as estruturas de dados existentes na linguagem de programação. Lembre-se de que uma pilha é um conjunto ordenado de itens, na linguagem C já contém um tipo de dado que representa um conjunto ordenado de itens, que é o vetor. Então, sempre que for necessário utilizar a estrutura de pilhas para resolver um problema, pode-se utilizar o vetor para armazenar esta pilha. Vale lembrar, porém, que a pilha é uma estrutura dinâmica e pode crescer infinitamente, enquanto um vetor na linguagem C tem um tamanho fixo; contudo, pode-se definir este vetor com um tamanho suficientemente grande para conter esta pilha.

Exemplo de Algoritmo de utilização de Pilha com tamanho definido

```
inicio
    inteiro i;
    real pilha[10], soma;
```

```

para(i←1; i<=10; i←i+1)
  inicio
    escreva("Digite o número", i, ":");
    empilha(pilha[i]);
  fim
soma = 0;
para(i←1; i<=10; i←i+1)
  escreva (pilha[i])
  soma = soma + pilha[i];
  escreva("Soma = ", soma);
fim

```

08

Operações com Pilha

É possível realizar algumas operações básicas na estrutura de dados Pilha, as mais importantes são a operação de empilhar (push) e desempilhar (pop). Mas para isso devemos executar outras operações para que isso seja possível:

- Criar uma estrutura de Pilha;
- Empilhar;
- Desempilhar;
- Informar se a pilha está vazia;
- Informar se a pilha está cheia;

Toda vez que criamos uma estrutura de pilha, esta deve ser inicializada para garantir que não haja nenhuma "sujeira" no local onde esteja montada. Importante elencar que:

Quando a pilha não está vazia não significa que ela esteja cheia, o que acontece também inversamente, ou seja, quando a pilha não está cheia, não significa que ela esteja vazia.

Criar uma estrutura de Pilha

Para iniciar o processamento com pilhas é importante criar para iniciar o armazenamento dos dados.

Empilhar

Na estrutura de dados pilha, sempre que um novo item é inserido ele passa a estar no topo da pilha.

Desempilhar

Na estrutura de dados pilha, sempre o item a ser retirado será o que estiver no topo da pilha, ou seja, o último que foi empilhado.

Informar se a pilha está vazia

Para verificar se uma pilha está vazia, basta verificar se o topo da pilha está vazio ou se o topo é igual a 0.

Informar se a pilha está cheia

Para verificar se uma pilha está cheia, caso não haja mais espaço para armazenar itens.

09

3 - FILA

Aprenderemos agora o comportamento das filas. Assim como listas e pilhas, as filas são estruturas de dados que armazenam os elementos de maneira sequencial.

A fila é um conjunto ordenado de itens a partir do qual se podem eliminar itens em uma extremidade, que é o início da fila, e que podem inserir itens na outra extremidade, que é o final da fila.

O que diferencia a fila da pilha é a ordem de saída dos itens. Relembrando o que já foi estudado, enquanto na pilha o último item que entra é o primeiro item que sai, na fila o primeiro item que entra é o primeiro item que sai (FIFO – first in, first out). Abaixo temos duas representações de Fila.

Esta é uma fila de blocos numerados, onde a ordem dos números é a ordem de chegada e colocação dos blocos.



Abaixo temos a representação da retirada dos blocos, que acontece de acordo com a ordem de chegada.



A ideia fundamental da fila é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início.

10

Você deve ter observado que a estrutura de fila é uma analogia natural ao conceito de fila que usamos em nosso dia a dia: quem primeiro entra numa fila é o primeiro a ser atendido (a sair da fila).

No dia a dia, enfrentamos e até já nos acostumamos às filas em diversos lugares. Quando precisamos ir ao banco, enfrentamos fila, quando vamos ao mercado, enfrentamos fila, no cinema também, e infelizmente no hospital às vezes é o que mais demora, entre outros lugares. Mesmo assim as filas são importantes, pois elas determinam a ordem de atendimento das pessoas. Como seria se não tivéssemos fila?



A simplicidade de uma fila fica interessante quando começamos a analisá-la. As pessoas são atendidas conforme a posição delas na fila. O próximo a ser atendido é sempre o primeiro da fila. Quando o primeiro da fila é chamado para ser atendido, a fila diminui, ou seja, o segundo passa a ser o primeiro, o terceiro passa a ser o segundo e assim por diante até a última pessoa. Significa que o último será primeiro em algum momento. Sendo assim, a pessoa que entra em uma fila, estará neste momento na última posição, ou seja, no fim da fila. Desta forma, quem chega antes tem prioridade.

11

Finalidade da Fila

A fila possui várias finalidades, citaremos aqui algumas:

- **o atendimento de processos requisitados a um sistema operacional:** os nossos micros, tablets, celulares, notebooks possuem um sistema operacional que gerencia toda a interface de comunicação (o que vemos na tela) e todos os serviços que solicitamos, por exemplo, quando mandamos salvar e imprimir, o sistema operacional armazena em uma fila esses comandos e executa na ordem de chegada;
- **buffer para gravação de dados em mídia:** quando vamos gravar uma grande quantidade de informações, como por exemplo, a cópia de um CD, o computador armazena isso em uma memória temporária em formato de fila, depois do processamento ele começa a descarregar a fila gravando a informação no local de destino.
- **processos de comunicação em redes de computadores:** os processos de envio de informações pela rede obedecem também uma fila, para fazer uma analogia imagine sua caixa de saída de mensagens, percebemos a fila quando temos problemas com o envio das mensagens, as mensagens ficam paradas

na caixa de saída. Quando o sistema normaliza as mensagens seguem conforme a fila, ou seja, seguem a fila de chegada na caixa e começa a enviar as mensagens até esvaziar a fila de saída.

• **implementação de uma fila de impressão:** a maioria das empresas utilizam impressoras compartilhadas por várias máquinas na rede, ou seja, várias pessoas enviam suas impressões para um único local, uma única impressora. A fila de impressão possibilita a impressora tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos, sendo que o primeiro enviado é o primeiro a ser impresso.

Esses exemplos demonstram a utilidade do dia a dia, utilidades essas que às vezes nem imaginamos que exista.

12

Operações com Fila

Para realizar as operações de uma fila, devemos ser capazes de inserir novos elementos em uma extremidade, o fim, e retirar elementos da outra extremidade, o início. Temos as seguintes operações:

- Criar uma estrutura de fila;
- Inserir um elemento no fim;
- Retirar um elemento do início;
- Verificar se a fila está vazia.

Toda vez que criamos uma estrutura de pilha, esta deve ser inicializada para garantir que não haja nenhuma "sujeira" no local onde esteja montada. Do mesmo modo da pilha, quando a fila não está vazia não significa que ela esteja cheia, o que acontece também quando a fila não está cheia não significa que ela esteja vazia.

Criar uma estrutura de Fila

Para iniciar o processamento com filas é importante criar para iniciar o armazenamento dos dados.

Inserir um elemento no fim

Na estrutura de dados fila, sempre que um novo item é inserido ele passa a estar na última posição da fila.

Retirar um elemento do início

Na estrutura de dados fila, sempre o item a ser retirado será o que estiver no início da fila, ou seja, o primeiro que foi inserido.

Verificar se a fila está vazia

Para verificar se uma fila está vazia, basta verificar se o início da fila é igual a -1. As filas sempre começam com o endereço 0.

13**RESUMO**

Nesse módulo apresentamos as listas, as pilhas e as filas, que são tipos abstratos de dados que criamos para gerenciar estrutura de dados. Uma lista é um conjunto ordenado de dados, geralmente do mesmo tipo e essa organização é feita através da enumeração dos dados para melhor visualização da informação. As pilhas e filas têm operações mais restritas do que as operações das listas. Nas filas e pilhas seguem um rito de entrada e saída, sendo que, Na fila o primeiro que entra é o primeiro que sai (FIFO - *First In, First Out*), e na Pilha o primeiro que entra é o último que sai (LIFO - *Last In, First Out*). Nas listas, os elementos são adicionados e removidos de qualquer posição. As filas, listas e pilhas estão presentes em nosso dia a dia e em cada detalhe do mundo da TI e fora da tecnologia também.

