



UNEB

UNIVERSIDADE DO  
ESTADO DA BAHIA

SISTEMAS DE INFORMAÇÃO

# Estrutura de Dados I

Professora: Maria Inés Restovic



# Ementa

Plano de Curso  
Avaliações



# Ementa

## CONTEÚDO PROGRAMÁTICO

### Conceitos básicos


- Algoritmos, estruturas de dados e programas
- Tipos de dados e tipos abstratos de dados
- Medida do tempo de execução de um programa
  - Estruturas de dados lineares
    - Pilhas
- Uso de pilhas na solução de problemas computacionais
- Representação de expressões: prefixa, infix e posfix
  - Filas
- Problemas na implementação seqüencial de filas
- Deques
  - Gerenciamento de memória
- Lista estática encadeada
- Alocação dinâmica encadeada
- Listas duplamente encadeadas
- Listas ordenadas: implementação e principais operações
  - Listas generalizadas
- Representação e implementação de listas generalizadas
- Algoritmos recursivos para listas generalizadas
  - Estruturas de dados lineares
- Árvores: Fundamentos e representações
- Árvores binárias: Algoritmos

## Avaliação

- ◎ A disciplina será avaliada por meio de trabalhos práticos implementados e apresentados pelos alunos.
- ◎ As notas estarão divididas em três módulos:
  1. Listas Dinâmicas: O modulo será avaliado com listas de exercícios individuais e um trabalho parcial individual, sendo o peso da implementação das listas de 40% da nota da unidade e o trabalho de 60%;
  2. Árvores Binarias: O modulo será avaliado com uma de exercícios individual e um trabalho parcial individual, sendo o peso da implementação das listas de 30% da nota da unidade e o trabalho de 70%;
  3. Projeto Final: poderá ser realizado em dupla e não tem listas de exercícios, sua avaliação corresponde a 100% da terceira unidade.
- ◎ A média parcial será obtida pela média aritmética dos três módulos, é seguida a regra padrão da UNEB:

Nota  $\geq 7,0$   $\Rightarrow$  Aprovado

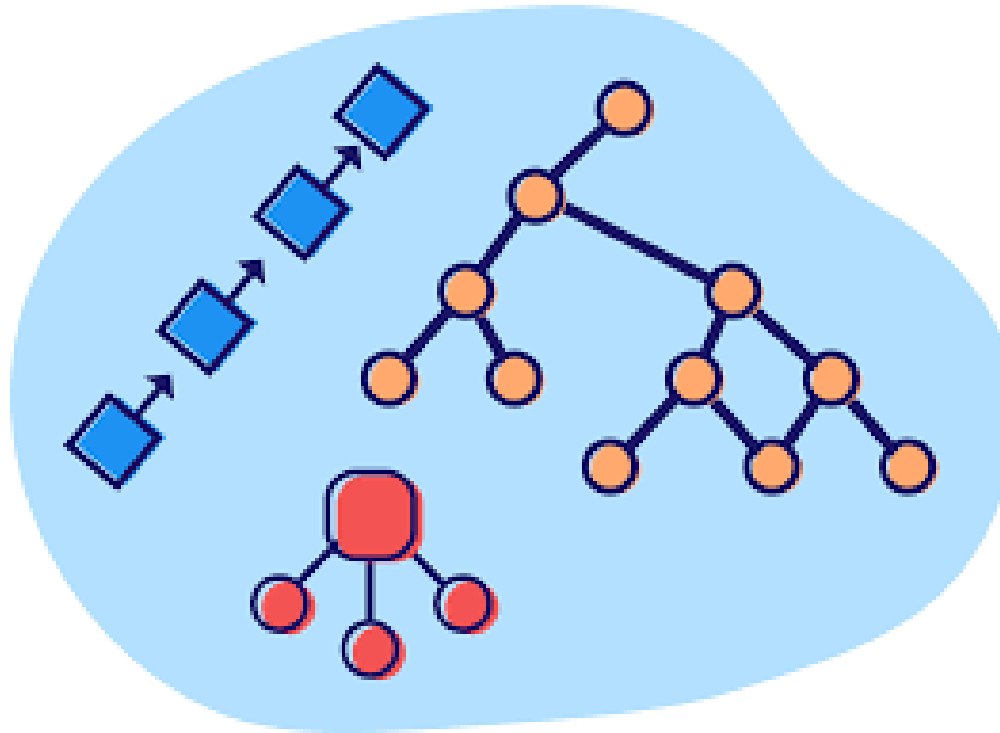
Nota  $< 7,0$   $\Rightarrow$  Prova Final



# Introdução

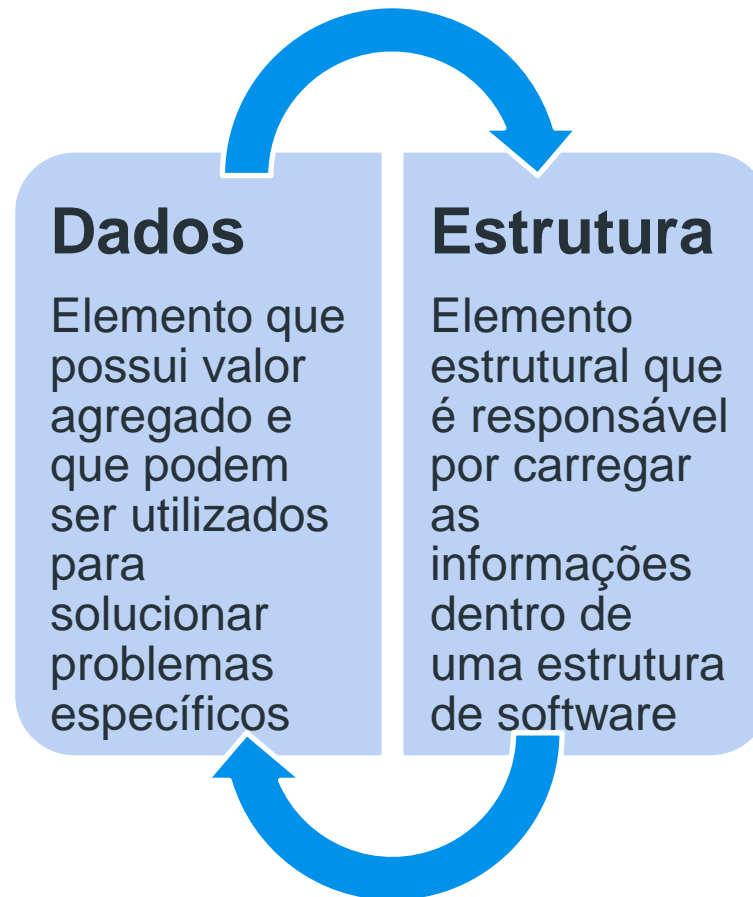
## O que é Estrutura de Dados

- Uma estrutura de dados é uma coleção tanto de valores e seus relacionamentos, quanto de operações (sobre os valores e estruturas decorrentes).



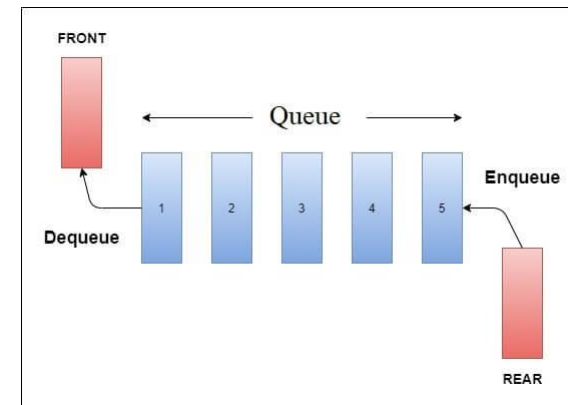
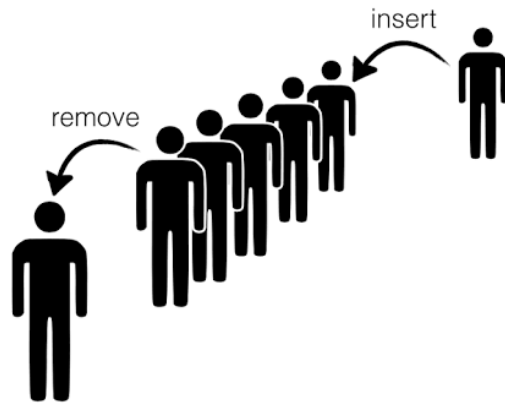
## O que é Estrutura de Dados

- Uma Estrutura de Dados pode ser dividida em dois pilares fundamentais:



## Definição

- ◎ **Estrutura de dados** é o ramo da computação que estuda os diversos mecanismos de organização de **dados** para atender aos diferentes requisitos de processamento. As **estruturas de dados** definem a organização, métodos de acesso e opções de processamento para a informação manipulada pelo programa.



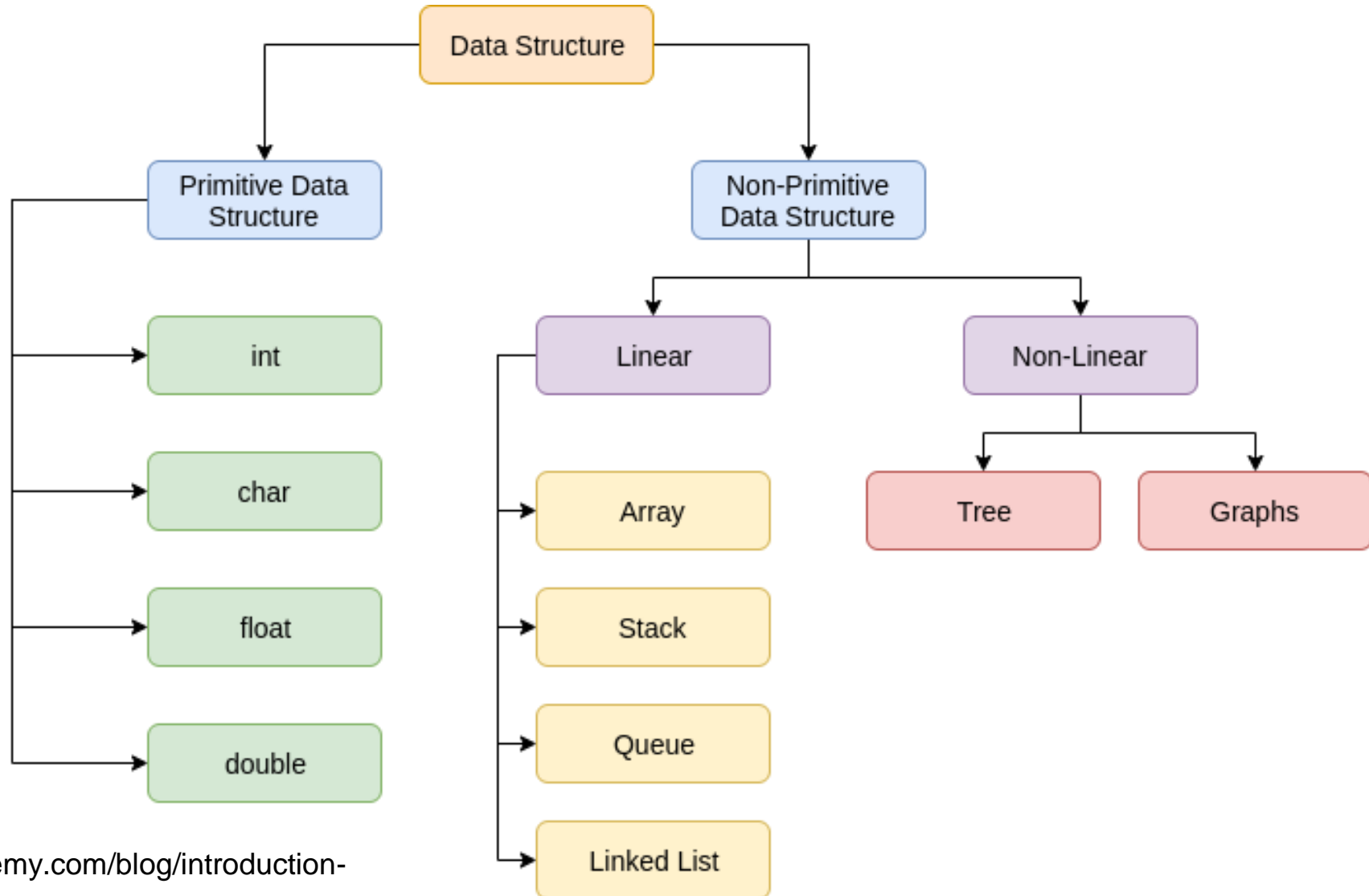


## Definição

- ◎ Critérios para escolha e estudo de uma estrutura de dados incluem:
  - Eficiência para buscas e padrões específicos de acesso, necessidades especiais para manejo de grandes volumes (*big data*)
  - Simplicidade de implementação e uso

EDs eficientes são cruciais para a elaboração de algoritmos, diversas linguagens possuem ênfase nas EDs, como na Programação Orientada a Objeto

## Tipos de Estruturas de Dados



## Aplicações

- ◎ Estruturas de Dados são muito utilizadas em aplicações de nível mais baixo como:
  - Bancos de Dados
  - Compiladores e Interpretadores
  - Editores de Texto
  - Sistemas Operacionais etc.



# Estruturas de Dados Lineares

Estáticas



## Vetores – Estruturas Estáticas

### Definição

- São tipos de dados compostos ou estruturados.
- É um conjunto finito e ordenado de dados.
- São chamados de estruturas estáticas porque não podem mudar de tamanho durante a execução do programa, ou seja, preservam o tamanho definido pelo programador no momento do desenvolvimento do software.
- São formados por índices e informações.
  - Índices: Definem as posições de armazenamento da estrutura
  - Informações: São os dados armazenados e identificados pelos índices.

0	1	2	3	4	5	6	7	8	9	índices
5	7	25	10	18	21	6	14	4	1	informação

# Vetores – Estruturas Estáticas

## Exemplo: inserir dados em um vetor

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4
```

Inclusão das bibliotecas

Declaração de uma constante

```
void imprime_vetor(int[]);
```

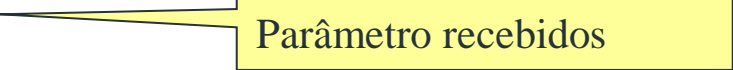
Definição dos protótipos, ou seja, especificação das funções do programa

```
int main()
{   int vetorA[MAX], numero, i=0;
    printf("\n\nDigite %d valores para o vetor\n",MAX);
    while (i<MAX){
        scanf("%d",&numero);
        vetorA[i] = numero;
        i++;
    }
    imprime_vetor(vetorA);
    return 0;
}
```

Declaração das variáveis

## Vetores – Estruturas Estáticas

```
void imprime_vetor(int vetor[MAX])  
{ int i;  
  printf("\n\nValores do vetor\n");  
  
  for (i=0; i <= MAX-1 ; i++)  
    printf("%d ", vetor[i]);  
  printf("\n\n");  
  
}
```



Parâmetro recebido

## Filas e Pilhas

### Definição

- Para determinadas aplicações é imposto um **critério** que restringe a inserção/retirada dos elementos que compõem um conjunto de dados.

### Critério de Pilha

**LIFO**: dentre os elementos que ainda permanecem no conjunto, o primeiro elemento a ser retirado é o **último** que tiver sido inserido.



### Critério de Fila

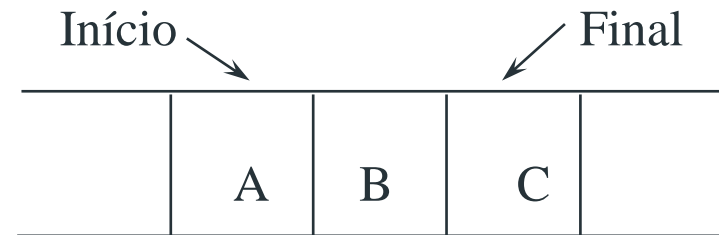
**FIFO**: dentre os elementos que ainda permanecem no conjunto, o primeiro elemento a ser retirado é o **primeiro** que tiver sido inserido.



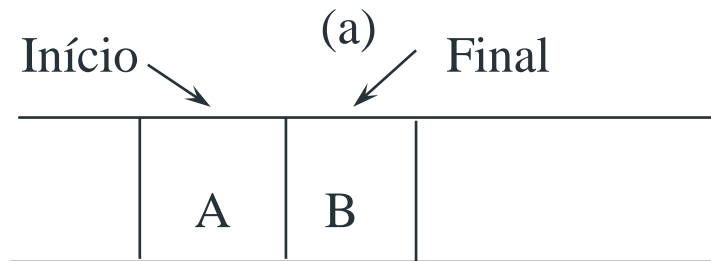


# Pilha

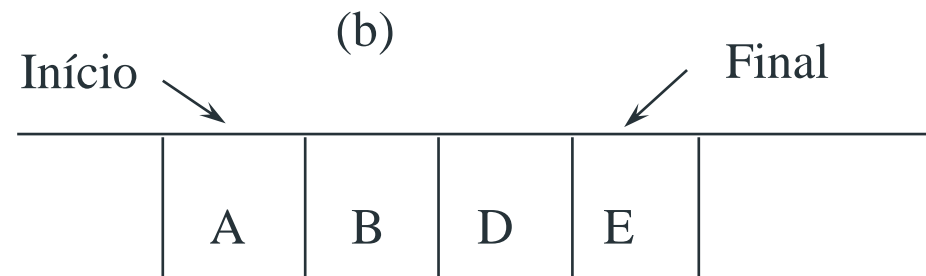
- ◎ **LIFO – Last In First Out**
- ◎ **O Último a Entrar é o Primero a Sair**



insere (A)  
insere (B)  
insere (C)



remove(ponteiro)



insere (D)  
insere (E)

(c)

## Pilha em Estrutura Estática

```
#include <stdio.h>
#include <stdlib.h>
#define MAXPILHA 5
```

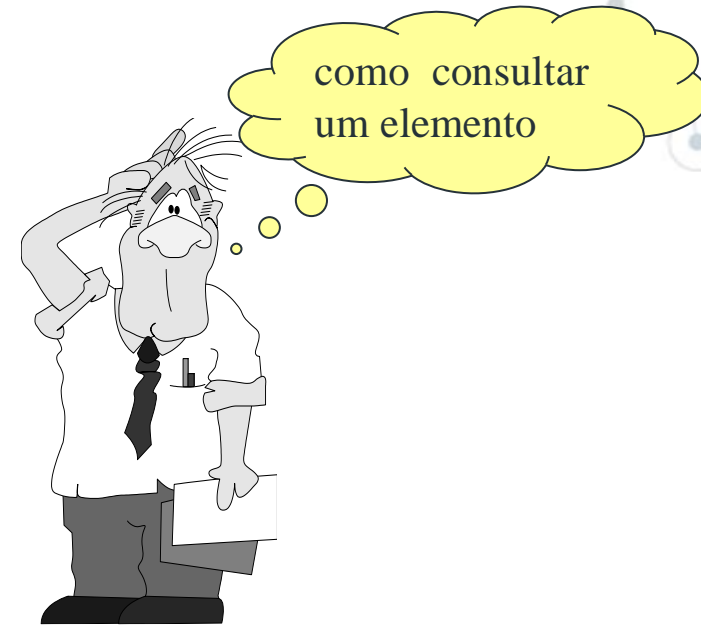
```
int pilha[MAXPILHA];
int topo=-1;
```

```
void empilha(int);
int desempilha();
void imprime_pilha();
```

```
int main()
{ int opcao;
  int valor;
  for(;;)
  { printf("1 - Incluir   2 - Excluir   3 - Listar   4 - Finalizar : ");
    scanf("%d",&opcao);
    switch (opcao) {
      case 1: printf("Entre com o nº a incluir : ");
              scanf("%d", &valor);
              empilha(valor); break;
      case 2: desempilha();
              break;
      case 3: imprime_pilha();
              break;
      case 4: exit(1);
      default : exit(1);
    }
  }
  return 0;
}
```

## Pilha em Estrutura Estática

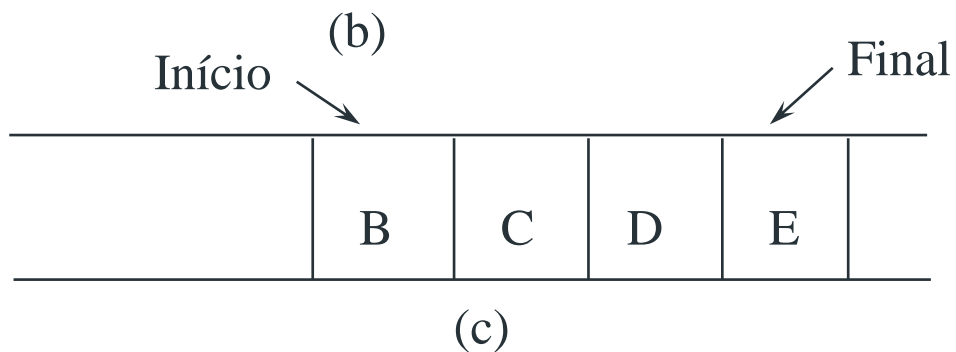
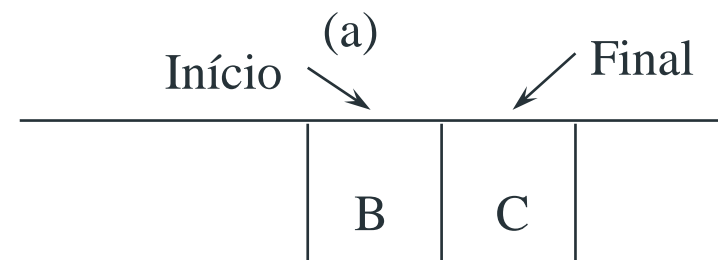
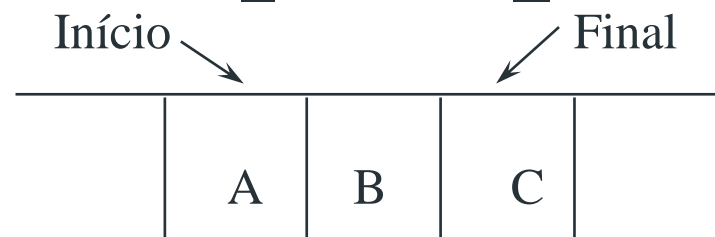
```
void empilha(int v)
{ if (topo >= MAXPILHA)
  { printf("Pilha Cheia");
    return;
  }
  topo++;
  pilha[topo] = v;
}
```



```
int desempilha()
{ if (topo == -1)
  { printf("Pilha Vazia");
    return(-1);
  }
  int elem=pilha[topo];
  topo--;
  return(elem);
}
```

## Fila

- ◎ FIFO – First In First Out
- ◎ O Primero a Entrar é o Primero a Sair



insere (A)  
insere (B)  
insere (C)

remove(ponteiro)

insere (D)  
insere (E)

## Fila em Estrutura Estática

```
#include <stdio.h>
#include <stdlib.h>
#define MAXFILA 5

int fila[MAXFILA];
int fim=-1;

void insere_fila(int);
int retira_fila();
void imprime_fila();
```

```
int main()
{ int opcao;
  int valor;
  printf("\n FILA ESTATICA\n\n");
  for(;;)
  { printf("1 - Incluir   2 - Excluir   3 - Listar   4 - Finalizar :
");
    scanf("%d",&opcao);
    switch (opcao) {
        case 1: printf("Entre com o n° a incluir : ");
                 scanf("%d", &valor);
                 insere_fila(valor);
                 break;
        case 2: retira_fila();   break;
        case 3: imprime_fila();   break;
        case 4: exit(1);
        default : exit(1); }
    }
  return 0;
}
```

## Fila em Estrutura Estática

```
void insere_fila(int v)
{ if (fim >= MAXFILA)
  { printf("\nFila Cheia\n");
    return;
  }
  fim++;
  fila[fim] = v;
}
```

```
int retira_fila()
{ int i, elem;
  if (fim == -1)
  { printf("\nFila Vazia\n");
    return(-1);
  }
  elem=fila[0];
  for (i = 0; i < fim; i++)
    fila[i] = fila[i+1];

  fim--;
  return(elem);
}
```

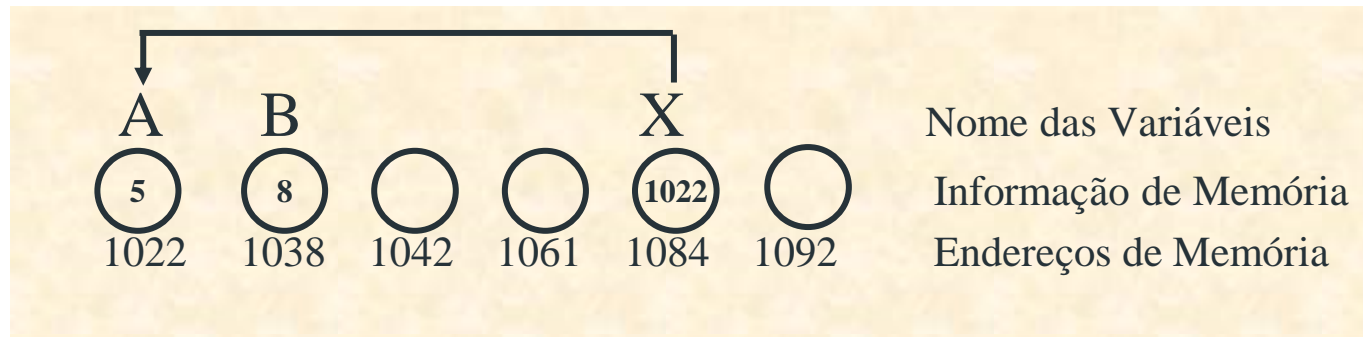


# Estrutura de Dados Dinâmicas

## Ponteiros

### Definição

- Variáveis que contém um endereço de memória. Se uma variável contém o endereço de outra, então a primeira (o ponteiro) aponta para a segunda.



- “X” o “ponteiro”, aponta para o “inteiro” A.
- Possibilitam manipular endereços de memória e informações contidas nesses endereços.



## Ponteiros

### ◎ Operadores

- **&** - (E comercial) - fornece o endereço de determinada variável. Atribui o endereço de uma variável para um ponteiro.

Obs: Não confundir com o operador lógico de operações de baixo nível, de mesmo símbolo.

- **\*** - (Asterisco) – permite acessar o conteúdo de uma variável, cujo endereço é o valor do ponteiro. Devolve o valor endereçado pelo ponteiro.

Obs: Não confundir com o operador aritmético de multiplicação de mesmo símbolo.

## Ponteiros

### ◎ Exemplo 1: Utilização dos operadores & e \*.

```
#include <stdio.h>
```

```
void main() {
```

```
    int destino, origem;
```

```
    int *m;
```

```
    origem = 10;
```

```
    m = &origem;
```

```
    destino = *m;
```

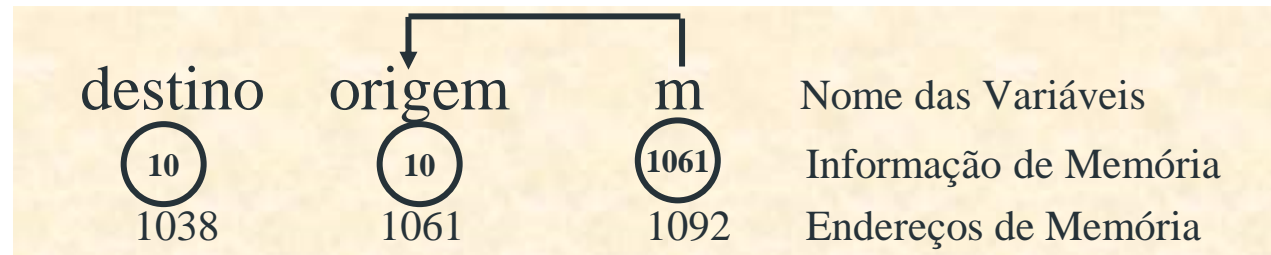
```
    printf("Valor da variavel destino: %d", destino);
```

```
}
```

Declaração de um ponteiro.

**m** obtém o endereço de memória da variável **origem**.

**destino** recebe a informação contida no endereço apontado por **m**.



## Ponteiros

### Exemplo 2: Atribuição de ponteiros

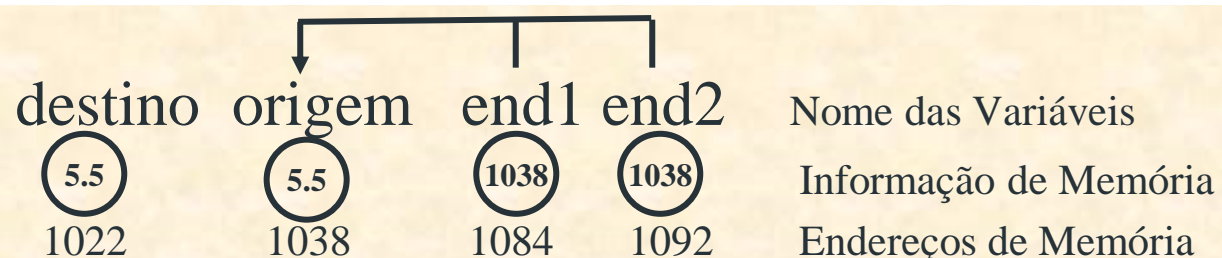
Declaração dos ponteiros.

```
#include <stdio.h>;
void main()
{ float destino, origem;
  float *end1, *end2;
    origem = 5.5;
    end1 = &origem;
    end2 = end1;
    destino = *end2;
    printf("O resultado é : %f",destino);
}
```

**end1** recebe o endereço de memória da variável **origem**.

**destino** recebe a informação contida no endereço apontada por **end2**.

**end2** recebe a posição de memória da variável **origem** que está guardada em **end1**.



# Ponteiros

## Exemplo 2: Atribuição de ponteiros

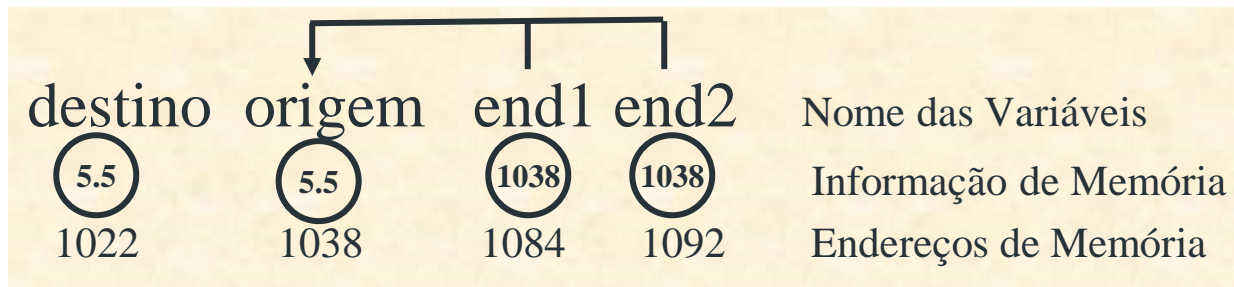
Declaração dos ponteiros.

**end2** recebe a posição de memória da variável **origem** que está guardada em **end1**.

```
#include <stdio.h>
void main(){
    float destino, origem;
    float *end1, *end2;
    origem = 5.5;
    end1 = &origem;
    end2 = end1;
    destino = *end2;
    printf("O resultado é : %f",destino);
}
```

**end1** recebe o endereço de memória da variável **origem**.

**destino** recebe a informação contida no endereço apontada por **end2**.



## Ponteiros e as Funções malloc e free

© **malloc():** Essa função atribui a um ponteiro uma determinada região de memória de acordo com o tipo do ponteiro.

- malloc() está definido na biblioteca **stdlib.h** do C
- Sintaxe: `void *malloc(size_t numero_de_bytes);`

© **Exemplo:**

```
int *p;  
p = (int *) malloc (sizeof (int));  
*p = 3;
```

## Ponteiros e as Funções malloc e free

- © **Free:** Essa função libera para o sistema operacional uma determinada região de memória alocada por um ponteiro.
- © Sintaxe: `free(ponteiro);`

- © **Exemplo:**

```
int *p;  
p = (int *) malloc (sizeof (int));  
*p = 5;  
free(p);
```

## Estruturas – Tipos definidos pelos usuários

- ◎ **Uma estrutura pode conter outra estrutura.**

```
struct data
{
    int  mes;
    int  dia;
    int ano;
};
```

```
struct conta
{
    int  num_conta;
    char tipo_conta;
    char nome[80];
    float saldo;
    struct data ultpag;
}
```

- ◎ **Inicializando estruturas**

- struct conta cliente = {12345, 'R', "Joao", 586.30, 5, 24, 30};

- ◎ **Processando uma estrutura**

- Acessando num\_conta: cliente.num\_conta
- Acessando o 3ª caracter do vetor nome: cliente.nome[2]

- ◎ **O uso do operador ponto pode ser estendido a vetores**

- struct conta cliente[100];
- número da conta do 14º cliente: cliente[13].num\_conta
- mês do último pagamento do 14º cliente: cliente[13].ultpag.mes

## Estruturas e Ponteiros

- Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados.

```
struct data  
{  
    int mes;  
    int dia;  
    int ano;  
}  
nascimento, *ptr;
```



ptr -> mes equivale a  
nascimento.mes

Ponteiros usam o  
operador ->

- Possibilidades de definição dos ponteiros.**

- A primeira é apontá-lo para uma variável struct já existente, da seguinte maneira:

```
struct data nascimento;  
struct data *ptr;  
ptr = &nascimento;
```

sizeof é o  
operador que  
retorna o tamanho  
de uma variável  
ou tipo.

- A segunda é alocando memória usando malloc():  

```
struct data *ptr = (struct data *) malloc (sizeof (struct data));  
ptr->dia=15;
```





# Estruturas de Dados Lineares

Dinâmicas



## Estruturas autorreferenciais

- ⊙ Estruturas autorreferenciais são estruturas que possuem um ponteiro como campo do tipo da própria estrutura.

- ⊙ **Definição**

```
struct tag
{
    membro 1;
    membro 2;
    ...
    struct tag *nome;
};
```

**Exemplo:**

```
struct nodo
{
    int info;
    struct nodo *prox;
}
```

- ⊙ **Exemplo da aplicabilidade.**

- Listas Encadeadas

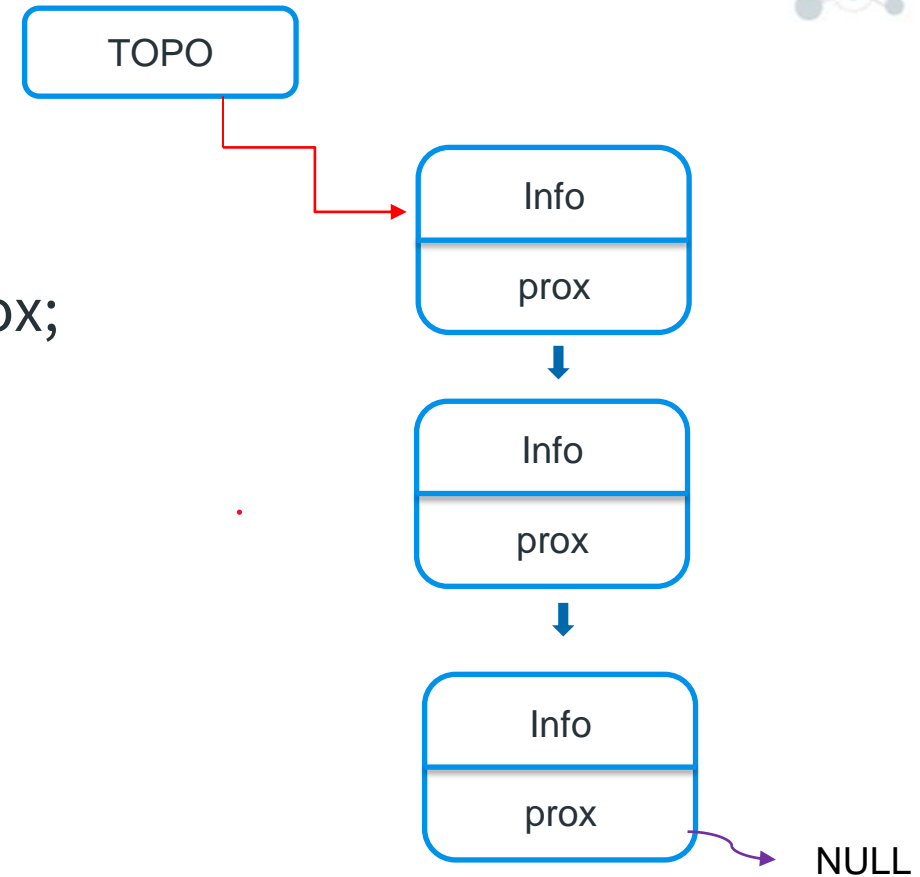
## Pilhas e Filas Dinâmicas

- ◎ A disciplina de inserção e retirada nestas estruturas segue as disciplinas LIFO e FIFO

- Area de Dados

```
typedef struct nodo {  
    int info;  
    struct nodo *prox;  
} Nodo;
```

### Exemplo: Pilha



## Pilha

```
4
5 // estrutura do Nodo
6 typedef struct nodo{
7     int info;
8     struct nodo *prox;
9 }Nodo;
10
11 // Funções
12 void empilha(int, Nodo** );
13 int desempilha(Nodo **);
14 void imprime_pilha(Nodo *);
15
16 int main()
17 { char opcao;
18   int valor;
19   Nodo *topo = NULL;
20
21
```

```
43 void empilha(int n, Nodo **topo){
44     Nodo *novo;
45     novo = (Nodo *)malloc(sizeof(Nodo));
46     if(novo == NULL) exit(1);
47     novo->info = n;
48     if(*topo == NULL)
49         novo->prox = NULL;
50     else
51         novo->prox = *topo;
52     *topo = novo;
53 }
54
55 int desempilha(Nodo **topo){
56     Nodo *aux;
57     int n;
58     if(*topo == NULL){
59         printf("\nPilha Vazia\n");
60         system("pause");
61         return -1;
62     }
63     n = (*topo)->info;
64     aux = *topo;
65     *topo = (*topo)->prox;
66     free(aux);
67     return(n);
68 }
69
```

## Fila

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  //Declaração de estrutura Nodo
5  typedef struct nodo{
6      int info;
7      struct nodo *prox;
8  }Nodo;
9
10 //Prototipos de funções
11 void insere(int, Nodo **, Nodo **);
12 int  retira(Nodo **);
13 void imprime_fila(Nodo *);
14
15 int main()
16 {
17     int opcao;
18     int valor;
19     Nodo *inicio = NULL;
20     Nodo *fim = NULL;
```

```
39
40 void insere(int n, Nodo **inicio, Nodo **fim){
41     Nodo *novo;
42     novo = (Nodo *)malloc(sizeof(Nodo));
43     if(novo == NULL) exit(1);
44     novo->info = n;
45     novo->prox = NULL;
46     if(*inicio == NULL)
47         *inicio = novo;
48     else
49         (*fim)->prox = novo;
50     *fim = novo;
51
52 }
53
54 int retira(Nodo **inicio){
55     Nodo *aux;
56     int n;
57     if(*inicio == NULL){
58         printf("\nFila Vazia\n");
59         system("pause");
60     }
61     n = (*inicio)->info;
62     aux = *inicio;
63     *inicio = (*inicio)->prox;
64     free(aux);
65     return(n);
66 }
67
```

## Lista Simplesmente Encadeada em Estrutura Dinâmica

### ◎ Definição

- É uma sequência de estruturas (elementos) interligados, com a capacidade de inserção e remoção em qualquer posição da lista.
- Cada nodo armazena um item de dados e um ponteiro para o nodo seguinte, e assim por diante. O último nodo possui um ponteiro NULL para indicar o final da lista.

### ◎ Critério

- O critério utilizado em listas determina que as inserções e remoções podem ser realizadas em qualquer posição da lista. Por conveniência utilizaremos a inserção por ordem de chave (informação única que distingue um elemento de outro).

## Lista Ordenada

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  // Declaração estrutura Nodo
4  typedef struct nodo{
5      int info;
6      struct nodo *prox;
7  }Nodo;
8
9  // Funções para lista Ordenada
10
11 void inserir(int, Nodo**);
12 int  buscar (int, Nodo**);
13 int  retirar(int n, Nodo **);
14 void listar(Nodo *);
15
16 int main()
17 {    Nodo *inicio = NULL;
```

```
35 // .....
36 // Inserir um elemento na ordem numerica ascendente
37
38 void inserir(int n, Nodo **inicio){
39     Nodo *novo, *ant, *atual;
40     novo = (Nodo *)malloc(sizeof(Nodo));
41     if (!novo)exit(1);
42     novo->info = n;
43     if(*inicio == NULL){
44         novo->prox = NULL;
45         *inicio = novo;
46         return;
47     }
48     ant = NULL;
49     atual = *inicio;
50     while((atual != NULL)&& (novo->info > atual->info)){
51         ant = atual;
52         atual = atual->prox;
53     }
54     if(atual == NULL) {ant->prox = novo;
55                     novo->prox = NULL; }
56     else if (atual == (*inicio)){
57         novo->prox = *inicio;
58         *inicio = novo;
59     }
60     else {
61         ant->prox=novo;
62         novo->prox = atual;
63     }
64 }
65
```

## Lista Ordenada

```
75
76 // retirar um elemento da lista
77 int retirar (int n, Nodo **inicio){
78     Nodo *ant, *atual, *ret;
79     int num;
80     ant = NULL;
81     atual = *inicio;
82     while((atual != NULL) && (n != atual->info)){
83         ant = atual;
84         atual = atual->prox;
85     }
86     if(atual == NULL) return 0;
87     if (atual == (*inicio)){
88         ret = *inicio;
89         *inicio = (*inicio)->prox;
90     }
91     else {
92         ret = atual;
93         ant->prox = atual->prox;
94     }
95     num = ret->info;
96     free(ret);
97     return num;
98 }
```

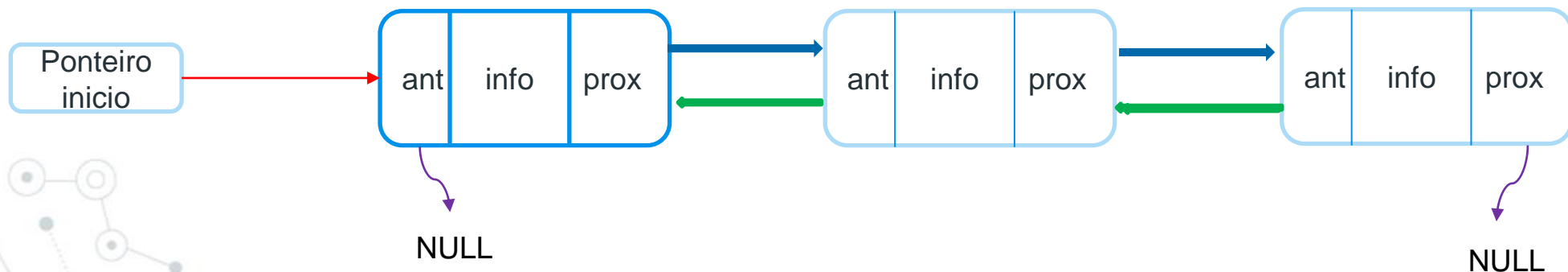
```
65
66 // buscar um elemto na fila 1 existe 0 nao existe
67 int buscar (int n, Nodo **inicio){
68     Nodo *aux;
69     aux = *inicio;
70     while((aux != NULL) && (n != aux->info))
71         aux = aux->prox;
72     if (aux == NULL) return 0;
73     else return 1;
74 }
75
```



## ◎ Definição **Lista Duplamente Encadeada**

- Em algumas aplicações que utilizam listas encadeadas pode ser de extrema necessidade percorrê-la da esquerda para a direita, bem como da direita para a esquerda. Este tipo de estrutura é chamada de Lista Duplamente Encadeada.
- A cabeça da lista contém dois ponteiros, um para o primeiro elemento da lista e outro para o último elemento da lista. Os nodos devem ter dois ponteiros, um para o próximo nodo e um para o nodo anterior.

## ◎ Representação Estrutural



## Lista Duplamente Encadeada em Estrutura Dinâmica

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
typedef struct nodo{
    int info;
    struct nodo *ant,*prox;
}Nodo;
```

```
// Funções para lista Ordenada
```

```
void inserir(int, Nodo**);
int buscar (int,Nodo**);
int retirar(int n, Nodo **);
```

Estrutura com dois ponteiros

```
67
68 // buscar um elemto na fila 1 existe 0 nao existe
69 int buscar (int n, Nodo **inicio){
70     Nodo *aux;
71     aux = *inicio;
72     while((aux != NULL)&&( n != aux-> info))
73         aux = aux->prox;
74     if ( aux == NULL) return 0;
75     else return 1;
76 }
77
```

# Lista Duplamente Encadeada em Estrutura Dinâmica

```
// Inserir um elemento na ordem numerica ascendente

void inserir(int n, Nodo **inicio){
    Nodo *novo, *atual;
    novo = (Nodo *)malloc(sizeof(Nodo));
    if (!novo)exit(1);
    novo->info = n;
    if(*inicio == NULL){
        novo->prox = NULL;
        novo->ant = NULL;
        *inicio = novo;
        return;
    }

    atual = *inicio;
    while((atual->prox != NULL)&& (novo->info > atual->info)){
        atual = atual->prox;
    }
```



```
        if((atual->prox == NULL) &&(novo->info > atual->info))
        {atual->prox = novo;
          novo->ant = atual;
          novo->prox = NULL; }
        else if (atual == (*inicio)){
            novo->prox = *inicio;
            novo->ant = NULL;
            (*inicio)->ant = novo;
            *inicio = novo;
        }
        else {
            atual->ant->prox = novo;
            novo->ant = atual->ant;
            novo->prox = atual;
            atual->ant= novo;
        }
    }
```

# Dinâmica

## ◎ Definição

- Pode ser considerada um método eficaz para resolver um problema originalmente complexo, reduzindo-o em pequenas ocorrências do problema principal. Divide para conquistar. Resolvendo, isoladamente, cada uma das pequenas partes, podemos obter a solução do problema original como um todo.

## ◎ Características de uma função recursiva

- Definição de parâmetros;
- Condição de parada da recursão, para que a rotina não seja chamada infinitamente;
- Chamada da função dentro dela própria;

# Recursividade

## ◎ Rotinas recursivas e pilhas

- O controle de chamadas e de retorno de rotinas é efetuado por uma pilha (criada e mantida dinamicamente pelo sistema). Quando uma rotina é chamada, empilha-se o endereço da rotina e todas as variáveis locais são recriadas. Quando ocorre o retorno da rotina as variáveis locais que foram criadas deixam de existir.

## ◎ Vantagens

- Facilidade na resolução de alguns tipos de problemas.

## ◎ Desvantagens

- Uso demasiado dos recursos computacionais de um computador.

## Recursividade

```
5 int fact(int n) {  
7     if(n == 0) return 1;  
3     return (n * fact(n-1));  
9 }
```

Parâmetro

Condição de parada

Chamada da própria função

**Exemplo:** calculo do fatorial de um número