

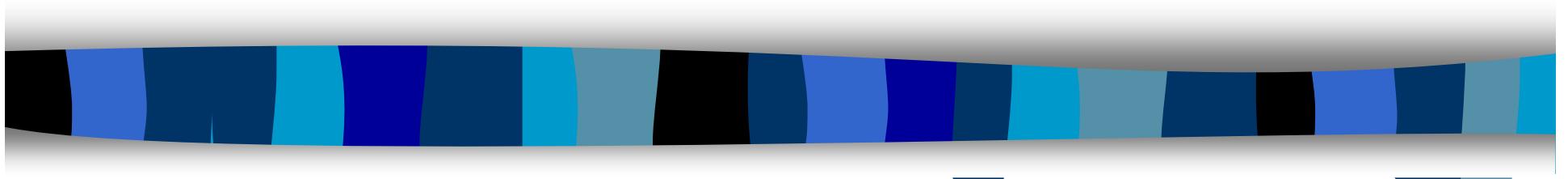
# Estruturas de Dados

## (com C)



Prof. MSc. Giulliano Paes Carnielli  
[prof\\_giulliano@yahoo.com.br](mailto:prof_giulliano@yahoo.com.br)

# *Aula 0*



*Apresentação do Curso*

# Professor

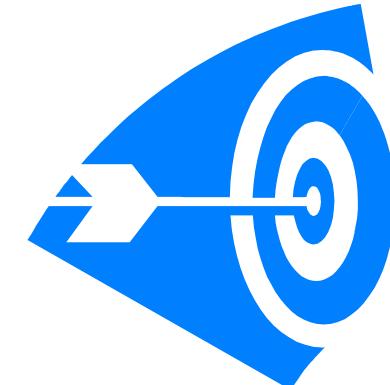
- Bacharel em Ciência da Computação – UFSCar (1997)
- Mestre em Ciência da Computação – Unicamp (1999)
- Professor há 10 anos, lecionando disciplinas de
  - Análise de Algoritmos
  - Sistemas Operacionais
  - Linguagens Formais e Autômatos
  - Compiladores
  - Programação Orientada a Objetos
  - Estrutura de Dados
  - Laboratórios de Programação
  - IHM
  - Gerência de Projetos
  - Processamento Digital de Imagens
  - Computação Gráfica
  - Engenharia de Software
- Analista de Sistemas Sr



# Ementa

- a. Revisão de tópicos: Vetores, Funções e Passagem de Parâmetros, Apontadores;
- b. Alocação Dinâmica;
- c. Introdução às Estruturas de Dados
  - i. Tipos Abstratos de Dados;
  - ii. Pilhas: definição, op. básicas, representação, aplicações;
  - iii. Filas: definição, op. básicas, representação, aplicações;
  - iv. Listas: definição, op. básicas, representação, aplicações, tipos (ligadas, circulares, duplamente ligadas)
- d. Recursividade
- e. Ordenação
- f. Pesquisa em Memória

# Orientação de Estudo



- O estudo de programação requer:
  - Dedicação na execução dos exercícios práticos
  - Execução dos exemplos e desenvolvimento dos exercícios
  - Discussão das dúvidas “no ato” (evite acumular problemas)
- ERRO COMUM: estudar para as provas
  - Estude sempre e periodicamente
  - Leia a teoria, digite os exemplos e resolva exercícios
- Dificuldades?
  - Releia a teoria, procure outras fontes: livros, web, ...
- Continua com dificuldades?
  - Anote suas dúvidas e traga para a sala.
  - Se for urgente, escreva ao professor.

# Sistema de Avaliação

$$\mathbf{MB1} = ((\mathbf{AM1} * 0.4 + \mathbf{AB1} * 0.6) * 0.5) + (\mathbf{AP1} * 0.5)$$

$$\mathbf{MB2} = ((\mathbf{AM2} * 0.4 + \mathbf{AB2} * 0.6) * 0.5) + (\mathbf{AP2} * 0.5)$$

$$\mathbf{MSemestral} = ((\mathbf{MB1} * 2) + (\mathbf{MB2} * 3)) / 5$$

**AB** (Avaliação Bimestral): resultado da prova bimestral

**AM** (Média Avaliações Mensais): listas e atividades desenvolvidas em classe

**AP** (Média das Avaliações Práticas): nota auferida à entrega de projeto prático

- Avaliações Bimestrais (datas): 09/04/2013, 11/06/2013
- Avaliações Práticas (datas): 17/04/2013, 19/06/2013
- Avaliação Substitutiva (data): 25/06/2013
- Avaliações Mensais (datas): datas definidas em tempo oportuno
- **IMPORTANTE**: o sistema não arredonda as médias finais

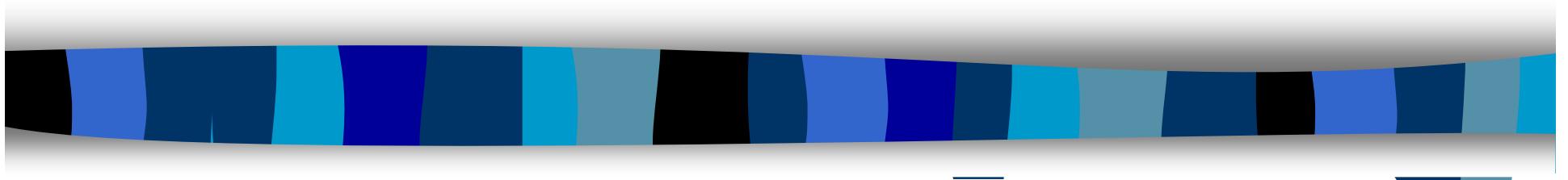
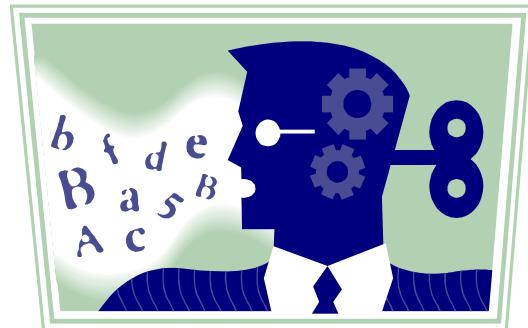
# Bibliografia

## Básica:

N.	Título	Autor	Edição	Local	Editora	Ano
01	Estruturas de Dados usando C	Tenembaum, A.M., Langsam, Y., Augestein, M.	1ª. Ed.	São Paulo	Makron Books	1995
02	Estruturas de Dados	Veloso, P., Santos,C. , Azeredo, C., Furtado, A.		Rio de Janeiro	Ed. Campus	1983
03	Introdução a Estruturas de dados - Com técnicas de Programação em C	Celes, Waldemar; Cerqueira, Renato; Rangel, José Lucas		Rio de Janeiro	Ed. Campus	2004

## Complementar:

N.	Título	Autor	Edição	Local	Editora	Ano
01	Estruturas de Dados e Algoritmos – Padrões de Projetos Orientados a Objetos com Java	Preiss, B. R.		Rio de Janeiro	Editora Campus	2001
02	Estruturas de Dados Fundamentais - Conceitos e Aplicações	Pereira, Silvio do Lago	8ª Ed.	São Paulo	Editora Érica	1996



# *Revisão de APC II*

# *Vetores e Matrizes*

# Tipos de Dados

- Sistemas de computação têm por finalidade a manipulação de dados
- Um algoritmo, basicamente, consiste na entrada de dados, seu processamento e a saída dos dados computados



- Alguns tipos de dados são considerados básicos, ou primitivos:
  - **Dados Numéricos**: representam valores numéricos diversos
    - Ex: 4, 5.5, 6.1E10, ...
  - **Dados Alfanuméricicos**: representam valores alfabéticos, numéricos, sinais, símbolos e caracteres
    - Ex: “salário”, “e23”, “%^#”, ...
  - **Dados Lógicos**: associados a valores lógicos (*True* ou *False*)
    - Ex: *true*, *false*

# Tipos de Dados Compostos

- Dois outros tipos de dados, um pouco mais sofisticado, são extremamente úteis: matrizes e vetores
- Tratam-se de variáveis compostas homogêneas
  - O conteúdo é sempre do mesmo tipo
- Correspondem a posições de memória contíguas
  - Identificadas por um mesmo nome
  - Individualizadas por índices
- Se NOTA contém os seguintes valores:

NOTA	60	70	90	55	91	47	74	86
	1	2	3	4	5	6	7	8

então, NOTA[3] faz referência ao terceiro elemento do conjunto (90)

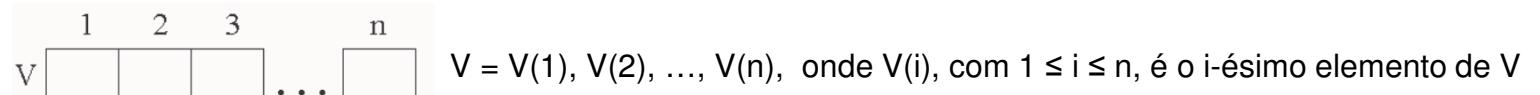


# Vetores

- Também chamado arranjo (array), tipo composto ou agregado
- É uma estrutura de dados homogênea e de acesso aleatório
  - Homogênea: contém elementos de um mesmo tipo;
  - Acesso aleatório: todos seus elementos são igualmente acessíveis a qualquer momento;
- Um elemento específico em uma matriz é acessado através de um índice
- Está disponível na maioria das linguagens
- É usado como base para estruturas de dados mais complexas
- Considere que o próprio acesso a memória do sistema é linear como um vetor

# Vetores

- Elementos de um vetor são referenciados com um mesmo *nome*, mas diferenciados por um único *índice*



- Sintaxe: tipo nome [tamanho];
- Exemplo:

```
void main() {
    int x[10]; /*reserva 10 elementos inteiros*/
    int i;
    for (i=0; i <10; i++) {
        x[i] = i;
    }
}
```

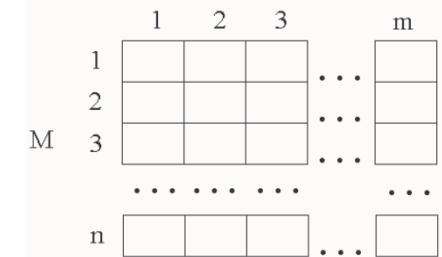
# Matrizes

- Arranjo (array) multidimensional. Normalmente, bidimensional;
- Assim como vetores, trata-se de uma estrutura de dados homogênea e de acesso aleatório;
- Necessita de dois ou mais índices
- Também figura na maioria das linguagens;

# Matrizes

- Arranjo de várias dimensões de dados do tipo básico, com um mesmo *nome*, mas diferenciado por um *índice* para cada dimensão.

$M(1,1) \quad M(1,2) \dots M(1,m)$  onde  $M(i,j)$  representa o elemento da  
 $M = M(2,1) \quad M(2,2) \dots M(2,m)$   $i$ -ésima linha e  $j$ -ésima coluna  
...  
 $M(n,1) \quad M(n,2) \dots M(n,m)$



- Sintaxe:

```
tipo NomeMatriz [MAX_LINHAS] [MAX_COLUNAS];
```

- Exemplo:

```
m[0][1] = 10;
```

```
m[1][0] = 20;
```

```
printf("Segundo valor de M: %d, m[1][0]);
```

# Matrizes

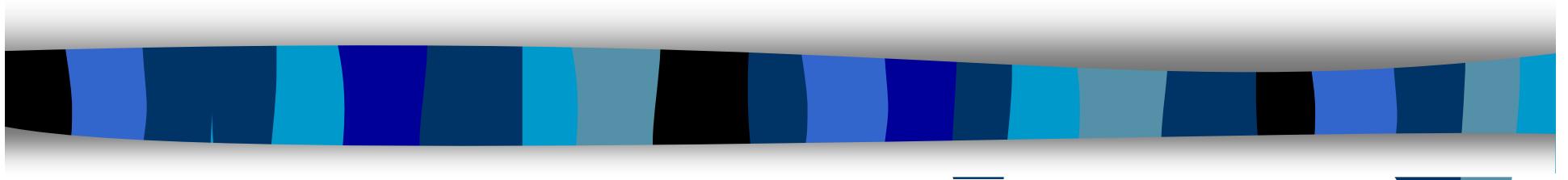
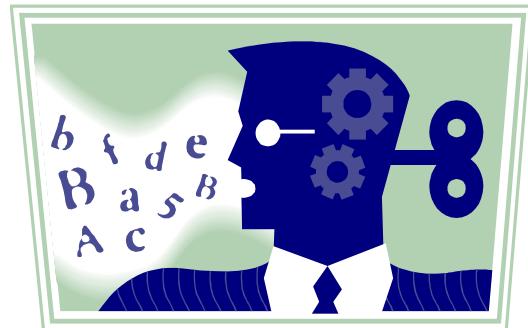
- Geralmente, o acesso aos elementos de uma matriz é feito por um comando de looping (muitos elementos)

```
for (i=0;i<10;i++) {           // percorre a linha  
    for (j=0;j<10;j++) {     // percorre a coluna  
        a[i][j] = 0;  
    }  
}
```

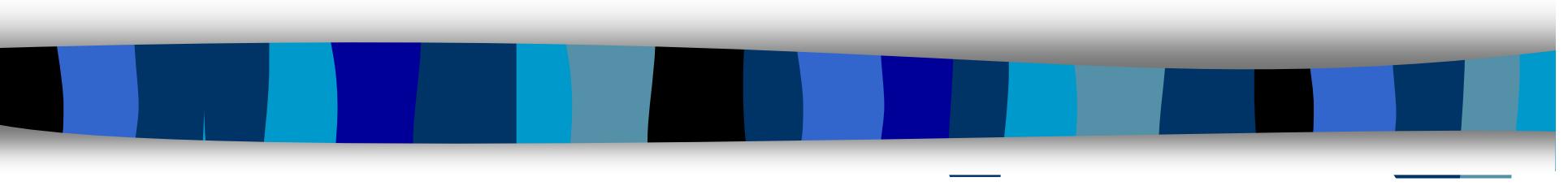
- O exemplo acima colocará o valor 0 (zero) para toda a matriz, percorrendo todas as colunas de todas as linhas.

# Exercícios:

1. Escrever programa que declare um vetor de 100 elementos numéricos, preenchido aleatoriamente com valores entre 1 e 1000. Depois, deve receber um valor do usuário e verificar se existem elementos iguais a esse valor no vetor. Se existir, escrever as posições em que estão armazenados.
  
2. Escrever programa que declare duas matrizes A e B, inteiras e bidimensionais 5x5, preenchidas aleatoriamente com valores entre 0 e 10. Depois:
  - a. Imprimir as matrizes
  - b. Executar a multiplicacao de A por B
  - c. Imprimir a matriz resultante



## *Revisão de APC II (cont)*



# *Funções e Procedimentos*

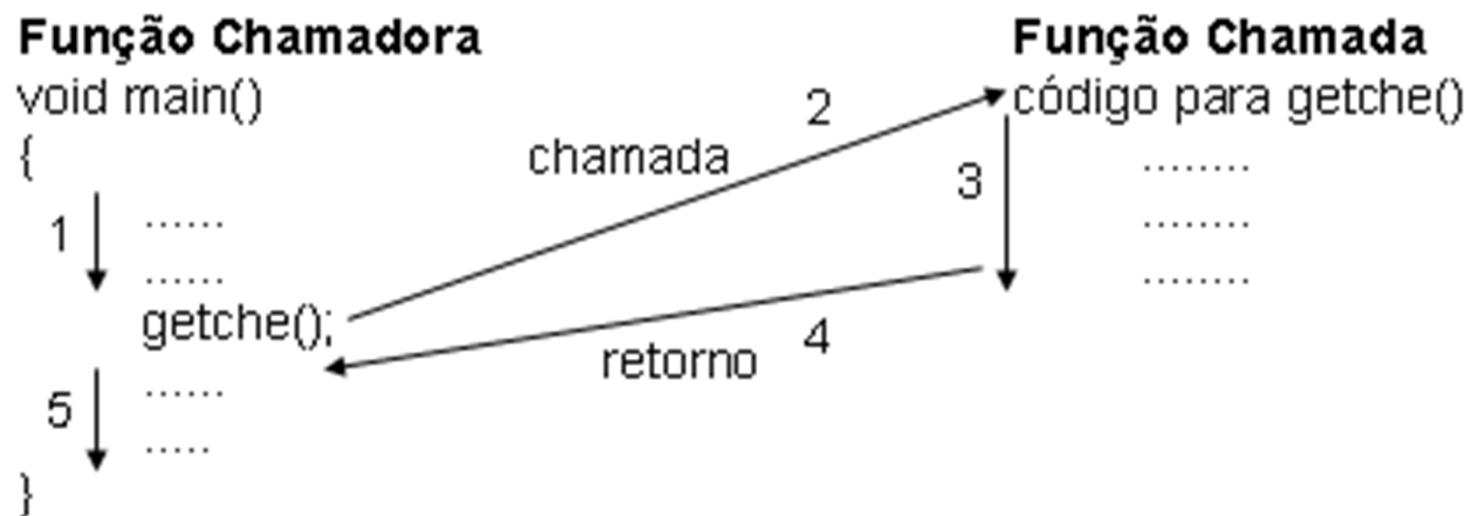


# Programação Modular

- De acordo com o paradigma da programação estruturada, a escrita de algoritmos (e programas) deve ser baseada no
  - desenho modular dos mesmos
  - passando-se depois a um refinamento gradual
- A modularidade permite entre outros aspectos:
  - Criar diferentes camadas de abstração do programa;
  - Reduzir os custos do desenvolvimento de software e correção de erros;
  - Reduzir o número de erros emergentes durante a codificação;
  - Re-utilização de código de forma mais simples;
- A modularidade pode ser conseguida através da utilização de *sub-rotinas*: funções e procedimentos.

# Chamada de Subrotinas

- Uma subrotina é um bloco de código associado a um nome, que pode ser chamado sob demanda a partir de outros pontos do programa



# Tipos de Subrotinas

- Na programação estruturada são normalmente definidos dois tipos de sub-rotinas: as funções e os procedimentos
- Função é um tipo de subrotina cujo funcionamento assemelha-se ao de uma função matemática: uma função sempre retorna um valor, como resultado de seu processamento
- Procedimento não retorna um valor após seu processamento, servindo principalmente como um bloco de execução

# Funções

- Uma função é definida por um *nome* (*nomeFuncao*), uma *lista de parâmetros*, constituída por zero ou mais variáveis passadas à função, um tipo de retorno e um corpo



```
<tipo de retorno> nomeFuncao  
(listaParametros)  
inicio  
<corpo>  
<retorno>  
fim
```

Exemplo em C:

```
long potencia (int base, int expoente){  
    /* Variavel de resultado */  
    long resultado = 1;  
    /* Calcula potencia atraves de multip. Sucessivas */  
    for (int i = 1; I <= expoente; i++) {  
        resultado *= base;  
    }  
    /* Retorna valor calculado */  
    return resultado;  
}
```

# Procedimento

- Um procedimento é definido por um *nome* (*nomeProc*), uma *lista de parâmetros*, constituída por zero ou mais variáveis e um corpo



Exemplo em C:

```
void numeroInvertido (int numero){
    while (numero > 0) {
        /* Calcula algarismo + a direita atraves de divisao
           inteira por 10 */
        int algarismo = numero % 10;
        printf ("%i", algarismo);
        /* Trunca algarismo a direita */
        numero = (numero - algarismo) / 10;
    }
}
```

# Exercício:

3. Escreva uma função em C, que receba um **número inteiro**, como parâmetro, e devolva o maior algarismo contido nesse número (não trate a entrada como string).

# *Parâmetros*

# Passagem de Parâmetros

- Dados são passados pelo algoritmo principal (ou outro subalgoritmo) à subrotina, ou retornados por este ao primeiro, por meio de parâmetros
  - Parâmetros formais são os nomes simbólicos (variáveis) introduzidos no cabeçalho das subrotinas, utilizados na definição dos seus parâmetros. Dentro da subrotina trabalha-se com estes nomes da mesma forma como se trabalha com variáveis locais.  
  
Ex: **float** calcPotencia(**int** base, **int** expoente) { ... }
  - Parâmetros reais são aqueles que substituem os parâmetros formais na chamada de uma subrotina. Os parâmetros formais são úteis somente na definição do subalgoritmo. Os parâmetros reais podem ser diferentes a cada chamada.  
  
int a = 2;  
Ex: calcPotencia(**a**, 3);



# Mecanismos de Passagem

- Os parâmetros reais substituem os parâmetros formais no ato da chamada de uma subrotina.
- Esta substituição é denominada passagem de parâmetros e pode se dar por dois mecanismos:
  - Passagem por Valor (cópia): na passagem de parâmetros por valor o parâmetro real é calculado e uma cópia de seu valor é fornecida ao parâmetro formal.
    - Modificações feitas no parâmetro formal não afetam o parâmetro real
    - Parâmetros formais possuem locais de memória exclusivos para que possam armazenar os valores dos parâmetros reais.
  - Passagem por Referência: na passagem de parâmetros por referência não é feita uma reserva de espaço de memória para os parâmetros formais.
    - Espaço de memória ocupado pelos parâmetros reais é compartilhado pelos parâmetros formais correspondentes
    - Eventuais modificações feitas nos parâmetros formais também afetam os parâmetros reais correspondentes

# Passagem de Parâmetros em C

- Em C, a passagem de parâmetro é normalmente feita por valor:
  - Passagem por Valor (cópia):
    - Declaração da função:  
`int minhaFuncao(int x){...}`
    - Chamada:  
`int a = 23;`  
`minhaFuncao(a)`
    - Uma cópia do valor do parâmetro real “a” é atribuída ao parâmetro formal “x”
  - Passagem por Referência: em C, a passagem por referência é feita através do uso de ponteiros
    - Declaração:  
`int minhaFuncao(int *x) { ... }`
    - Chamada:  
`int a = 23;`  
`minhaFuncao(&a);`
    - O endereço do parâmetro real “a” é passado para o ponteiro “x”. Logo, o parâmetro formal “x” referencia a mesma posição de memória de “a”. Por isso, qualquer alteração feita em “x” é refletida em “a”.

# Exercícios:

4. Modularize o código do Exercício 2, apresentado no laboratório anteriormente. Sugira e implemente uma alternativa para evitar a repetição do código que imprime as matrizes.
  
5. Escreva uma função em C que receba um número inteiro “i” e uma matriz 3x3 como parâmetros. A função deve calcular a multiplicação “m” dos elementos da diagonal principal dessa matriz. Deve atualizar a matriz original, multiplicando todos os seus elementos por “i”. O valor “m” calculado deve ser retornado.

# *Registros*

# Estrutura de Dados Heterogênea

- Como vimos, temos 4 tipos básicos de dados simples: **reais**, **inteiros**, **literais** e **lógicos**.
- Podemos usar tais tipos primitivos para definir novos tipos:
  - Agrupam conjunto de dados homogêneos (mesmo tipo) sob um único nome (como os vetores)
  - Agrupam dados heterogêneos (tipos diferentes): **Registros**

# Registros

- Correspondem a conjuntos de posições de memória conhecidos por um mesmo nome, mas com identificadores associados a cada conjunto: **membros ou componentes**
- Geralmente, todos os membros são logicamente relacionados;
- Sintaxe de criação de Registros em C:

```
struct <nome_do_registro> {  
    <componentes_do_registro>  
};
```

- *<nome\_do\_registro>*: escolhido pelo programador e considerado um novo tipo de dados
- *<componentes\_do\_registro>*: declaração das variáveis-membro deste registro, baseada em tipos já existentes
- Um registro pode ser definido em função de outros registros já definidos

# Registros

*Definindo uma Estrutura*

```
struct Endereco {  
    char nome[30];  
    char rua[50];  
    char bairro[20];  
    char cidade[20];  
    char estado[2];  
    int cep;  
};
```

# Registros

- Neste ponto do código (código anterior), nenhuma variável foi de fato declarada. Apenas a forma dos dados foi definida;

## *Declarando Variáveis*

- Para declarar uma variável com essa estrutura (Endereco), escreva:

```
struct Endereco end_info;
```

- Isso declara uma variável do tipo estrutura *Endereco* chamada *end\_info*;

# Registros

- Você também pode declarar uma ou mais variáveis enquanto a estrutura é definida. Por exemplo:

```
struct Endereco {  
    char nome[30];  
    char rua[50];  
    char bairro[20];  
    char cidade[20];  
    char estado[2];  
    int cep;  
} end_info, binfo, cinfo;
```

- define uma estrutura chamada Endereco e declara as variáveis end\_info, binfo e cinfo desse tipo;

# Registros

## *Acesso aos membros (ou campos) de uma estrutura*

- Elementos individuais de estruturas são referenciados através do operador ponto. Por exemplo;

```
end_info.cep = 130270410;
```

o nome da variável estrutura seguido por um ponto e pelo nome do elemento referencia esse elemento individual da estrutura;

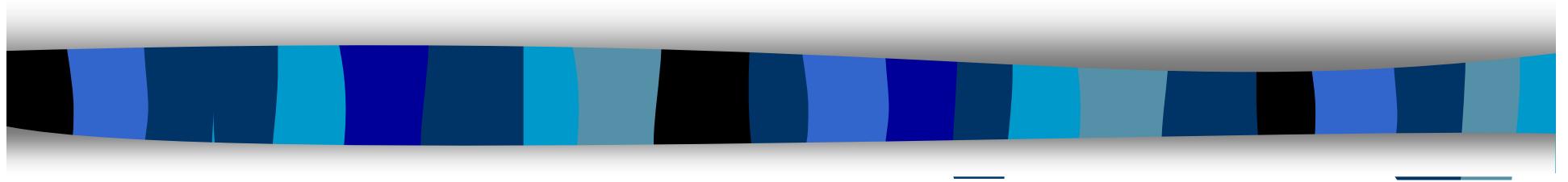
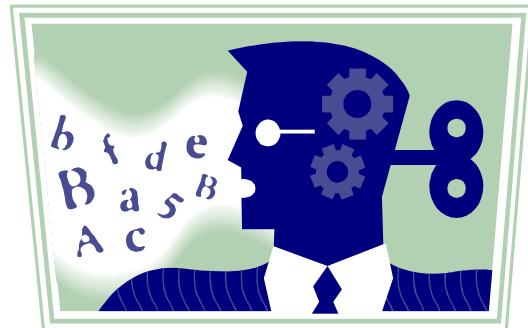
- **Forma Geral:** `nome_da_estrutura.nome_do_elemento;`
- Imprimindo o CEP na tela: `printf ("%d", end_info.cep);`

# Registros

- Um membro (ou campo) de uma estrutura pode ser outra estrutura:

```
struct ENDERECO{  
    char rua[30];  
    int numero;  
    char bairro[30];  
    char cidade[30];  
    char estado[2];  
    char cep[10];  
};  
  
struct PESSOA{  
    char nome[30];  
    char sobrenome[30];  
    struct ENDERECO end;  
    int idade;  
    char rg[15];  
    float salario;  
};
```

```
void main() {  
    ....  
    struct PESSOA pess;  
    ....  
    strcpy(pess.end.rua, "Barão de Itapura");  
    pess.end.numero = 189;  
    strcpy(pess.end.estado, "SP");  
    ...  
}
```



# Ponteiros

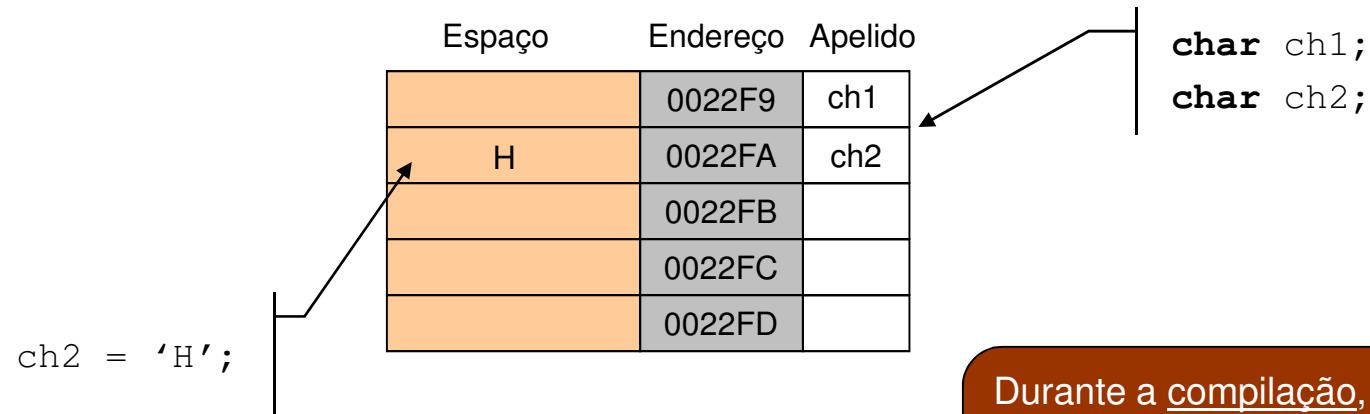


# Variáveis e Endereços

- Toda variável tem um endereço de memória associado a ela.
- Esse endereço é o local onde essa variável é armazenada no sistema.
- Normalmente, o endereço das variáveis não são conhecidos quando o programa é escrito.
- O endereço de uma variável é dependente do sistema computacional e também da implementação do compilador de linguagem que está sendo usado.
- O endereço de uma mesma variável pode mudar entre diferentes execuções de um mesmo programa usando uma mesma máquina.

# Alocação de Memória

- Uma variável provê identificação unívoca para uma peça de informação.
- Uma declaração de variável provoca a reserva de uma área de memória, cujo tamanho está relacionado ao tipo da variável.
- O nome da variável permite que essa área de memória seja referenciada pelo “apelido” e não pelo seu endereço.



Durante a compilação, toda referência a nomes de variáveis é substituída por um endereço relativo de memória.

# Operador de Endereço

- Na linguagem C é possível saber o endereço de uma variável através do operador &.
- **Exemplo:**

```
main() {  
    char ch2 = 'H';  
    printf("Conteúdo de ch = %c", ch2);  
    printf("Endereço de ch = %x", &ch2);  
}
```

Espaço	Endereço	Apelido
	0022F9	ch1
H	0022FA	ch2
	0022FB	
	0022FC	
	0022FD	

- Função **scanf**:
  - Recebe como parâmetro uma referência de endereço (&ch2).
  - Lê dados do teclado e armazena no endereço fornecido.

# Apontadores (Ponteiros)

- Variáveis que armazenam endereços de memória
- Para cada tipo de dados, existe um tipo para guardar o seu endereço, indicado por \* antes do nome da variável
  - int \*endereço: endereço de uma variável inteira
  - float \*endereço: endereço de uma variável de ponto flutuante
  - char \*endereço: endereço de uma variável de caractere
- Ao atribuir o endereço de uma variável a um apontador, dizemos que o mesmo “aponta” para a variável

# Apontadores (Ponteiros)

- Exemplo:

Espaço	Endereço	Apelido
H	0022F9	ch1
	0022FA	
	0022FB	
	0022FC	
0022F9	0022FD	ptr

```
char ch1 = 'H';
char *ptr; /* apontador para char */
ptr = &ch1; /* ap_x aponta para ch1 */
```

Importante: o espaço ocupado por um apontador depende do espaço de endereçamento de memória do sistema.  
Neste exemplo: 3 bytes.

# Apontadores (Ponteiros)

- Portanto, para declarar uma variável do tipo apontador utilizamos o operador unário \*
  - **int** \*ap\_int;
  - **char** \*ap\_char;
  - **float** \*ap\_float;
  - **double** \*ap\_double;
- Cuidado ao declarar vários apontadores em uma única linha. O operador \* deve preceder o nome da variável e não suceder o tipo
  - **int** \*ap1, \*ap\_2, \*ap\_3;

# Apontadores (Ponteiros)

- Importante: para acessarmos o valor de uma variável apontada por um endereço, também usamos o operador \*

Espaço	Endereço	Apelido
X	0022F9	ch1
	0022FA	
	0022FB	
	0022FC	
0022F9	0022FD	ptr

```
char ch1 = 'H';
char *ptr; /* apontador para char */
ptr = &ch1; /* ap_x aponta para ch1 */
```

```
*ptr = 'X';
```

Importante: `*ptr` pode ser usado em qualquer contexto que a variável `ch1` seria.

# Apontadores para Registros

- Para acessar os elementos de um registro através de um apontador, devemos primeiro acessar o registro e depois acessar o campo desejado.

```
struct ponto { double x; double y; };
typedef struct ponto Ponto;
Ponto *ap_p, p;
ap_p = &p;
(*ap_p).x = 4.0;
(*ap_p).y = 5.0;
```

- Os parênteses são necessários pois o operador `*` tem prioridade menor que o operador `.`

# Apontadores para Registros

- Para simplificar o acesso aos campos de um registro através de apontadores, foi criado o operador “`->`”.
- Usando este operador acessamos os campos de um registro diretamente através do apontador

```
struct ponto { double x; double y; };
typedef struct ponto Ponto;
Ponto *ap_p = NULL, p;
ap_p = &p;
ap_p->x = 4.0;
ap_p->y = 5.0;
```

# Vetores e Matrizes

- Vetores e Matrizes sempre foram apontadores, mas a sintaxe de vetores “esconde” esse fato.
  - Uma variável que representa um vetor é implementada por um apontador constante para o primeiro elemento do vetor.
  - A operação de indexação corresponde a deslocar este apontador ao longo dos elementos alocados ao vetor (sem perder a referência inicial).

```
int vet[10];  
vet → 0 1 2 3 4 5 6 7 8 9
```

```
vet[4] = 2;  
vet → 0 1 2 3 4 5 6 7 8 9  
                                ↑  
(vet + 4)
```

- Portanto,
  - Vetores são sempre passados por referência e
  - A linguagem C não pode detectar acessos fora dos limites do vetor

# Vetores de Ponteiros

- Não existe diferença entre vetores de apontadores e vetores de tipos simples.
  - basta observar que o operador \* tem precedência menor que o operador de indexação []

Ex:

```
int main() {  
    char *cores[] = {"amarelo", "verde", "vermelho", "laranja", "preto"};  
  
    int a=3, b=5, c=78, d=23;  
    int *numeros[] = {&a, &b, &c, &d};  
  
    printf("%d      %s", *numeros[0], cores[0]);  
    getch();  
    return 0;  
}
```



# Expressões com Ponteiros

- Alguns aspectos especiais sobre ponteiros:
  - Atribuição de Ponteiros
  - Aritmética de Ponteiros
  - Comparação de Ponteiros

# Atribuição de Ponteiros

Como é o caso com qualquer variável, um ponteiro pode ser usado no lado direito de um comando de atribuição para passar seu valor para outro ponteiro:

```
#include<stdio.h>
int main()
{
    int x=10;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;
    printf("O endereço de p2 eh: %p\n\n", p2);
    printf("O conteúdo do endereço apontado pelo ponteiro p1 eh: %d\n\n", *p1);
    printf("O conteúdo do endereço apontado pelo ponteiro p2 eh: %d\n\n", *p2);
    system("pause");
}
```

Tanto p1 quanto p2 apontam para o endereço de memória da variável x



# Aritmética de Ponteiros

- Existem apenas duas operações aritméticas que podem ser usadas com ponteiros:
  - Adição (incremento) e
  - Subtração (decremento).
- Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta;
  - Se tivermos um ponteiro para um inteiro e o incrementarmos ele passa a apontar para o próximo inteiro;

# Aritmética de Ponteiros

- A aritmética de ponteiros não se limita apenas ao incremento e decremento, podemos somar ou subtrair inteiros de ponteiros:

```
#include<stdio.h>
int main()
{
    float *p1;
    printf("O endereço de p1 eh: %p",p1);
    p1 = p1+20;      //equivale à p1+=20;
    printf("\n\nO novo endereço de p1 eh: %p\n\n",p1);
    system("pause");
}
```

p1 apontará para o 20º elemento do tipo float adiante ( $20 \times 4$  bytes adiante)



# Aritmética de Ponteiros

- Além de adição e subtração entre um ponteiro e um inteiro, nenhuma outra operação aritmética pode ser efetuada com ponteiros;
- Não podemos multiplicar ou dividir ponteiros;
- Não podemos adicionar ou subtrair o tipo float ou o tipo double a ponteiros;
- Quando uma variável precisa ser acessada de diferentes partes do programa.

# Aritmética de Ponteiros

- Exemplo:

```
main() {  
    int v[10];  
    int *el;  
    int i;  
  
    el = &v[0];  
  
    /*inicializa conteudo de v via ponteiro */  
    for (i=0; i<10; ++i)  
        *(el + i) = 0;  
}
```

Ou seja: a expressão  $el+i$  aponta para  $v[i]$ .

# Comparação de Ponteiros

- É possível comparar dois ponteiros em uma expressão relacional (<, <=, > e >=) ou se eles são iguais ou diferentes (== e !=);
- A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
#include<stdio.h>
int main()
{
    int x=10, y=10;
    int *p1, *p2;
    p1 = &x;
    p2 = &y;
    if(p1>p2)
        printf("\nA variavel x esta armazenada em um endereco de memoria acima da variavel y");
    else
        printf("\n\nA variavel y esta armazenada em um endereco de memoria acima da variavel x");
    printf("\n\nCertificando...\n\t\tEndereco de x: %p \n\t\tEndereco de y: %p\n\n");
    system("pause");
}
```

# Expressões com Ponteiros

## ■ Exercícios:

Considere o trecho de programa abaixo. Depois de executado, quais são os valores associados aos itens de (a) a (g). Suponha que os endereços das variáveis *u* e *v* são 1000 e 1004 respectivamente.

```
int v, u;
int *pv, *pu;          (a) &v
v = 45;                (b) pv
pv = &v;               (c) *pv
*pv = v + 1;           (d) u
u = *pv + 1;           (e) &u
pu = &u;               (f) pu
                        (g) *pu
```

# Exercícios (Lab 3)

6. Crie um programa que declare um vetor de inteiros de 20 posições. Sem utilizar indices, faça o seguinte:
  - Preenchê-lo com valores aleatórios de 1 – 100
  - Ler um inteiro “aux” (entre 0-19)
  - Imprimir o valor contido na posição “aux” do vetor
  - Imprimir o vetor inteiro, na ordem inversa.
  - OBS: lembre-se de NÃO usar indices.

# Exercícios (Lab 3)

7. Crie programa que defina duas estruturas: Pessoa e Endereco. Pessoa deve ter nome, idade, RG e *referência* (ponteiro) para Endereco. Endereco deve ter rua, numero, complemento, cidade. O programa deve declarar e inicializar uma variável do tipo Pessoa e outra do tipo Endereco (ligando uma à outra).

Implemente a função

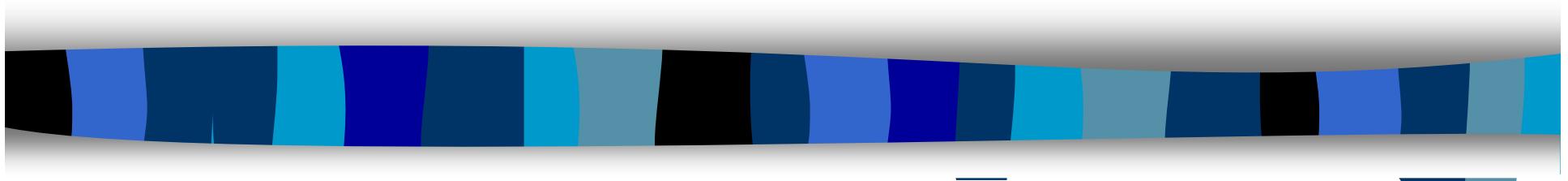
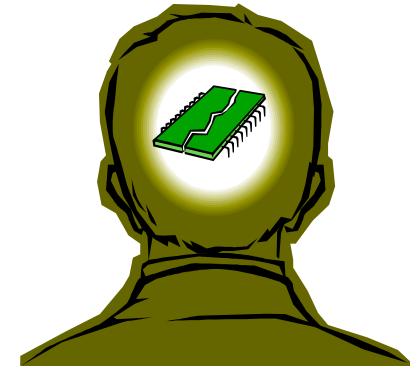
```
void alteraEndereco (Pessoa *p, Endereco *e)  
que deve substituir o endereço da pessoa “p” por “c”
```

Implemente a função

```
void atualizaEndereco (Pessoa *p, char m, char *valor)  
atualiza um valor do endereço (“m” indica qual: r, n, c, C)
```

Implemente a função

```
void imprimePessoa (Pessoa *p)
```



# *Alocação Dinâmica de Memória*



# Alocação de Memória

- Duas maneiras (mais comuns) de reservar memória:
  - a. Reserva estática de espaços de memória com tamanho fixo, na forma de variáveis locais :  
`int a; char nome[64];`
  - b. Reserva dinâmica de espaços de memória de tamanho arbitrário, com o auxílio de ponteiros:  
`int *a; char *nome;`
- Variáveis não podem ser acrescentadas em tempo de execução
  - Porém, um programa pode precisar de quantidade variável de memória
- A reserva só ocorre durante a execução do programa, através de requisições ao SO



# Alocação de Memória

- Vantagens da alocação dinâmica:
  - Flexibilidade: muitas vezes não temos como prever, antecipadamente, as necessidades de uso de memória.
  - Economia: podemos reservar memória de acordo com a necessidade imediata. Evitamos super-dimensionamentos.
- Desvantagem principal:
  - Gerenciamento de Memória:
    - A alocação dinâmica atribui parte da responsabilidade de gerenciar memória ao programador. Essa tarefa, geralmente, é propensa a erros de difícil detecção e correção.
    - Gerenciar a memória envolve: reserva de memória, acesso correto às regiões alocadas, liberação de regiões não mais utilizadas

# Funções de A alocação em C

- A alocação e liberação de espaços de memória é feito por funções da biblioteca “stdlib.h” (em alguns sistemas “malloc.h”)
- As principais:
  - malloc(size): “*memory allocation*” Aloca espaço em memória
  - free(ref): Libera espaço em memória, alocado dinamicamente
- Alternativas:
  - calloc(n, size): “*count allocation (?)*” Aloca espaço em memória para um *array* de *n* elementos de tamanho *size*
  - realloc(ref, size): Modifica o tamanho de um bloco de memória previamente alocado.

# Função malloc()

- Aloca um bloco consecutivo de bytes na memória e retorna o endereço deste bloco.
- Devemos informar o tamanho do bloco, por parâmetro, em número de bytes
  - Freqüentemente, devemos usar a função “`sizeof()`”
- O espaço alocado pode ser usado para armazenar qualquer tipo de dado (`void *`).
  - Devemos converter o tipo genérico retornado (`void *`) para o tipo desejado (`cast`)

Ex:

```
Aluno *a;  
a = (Aluno *)malloc(sizeof(Aluno));
```

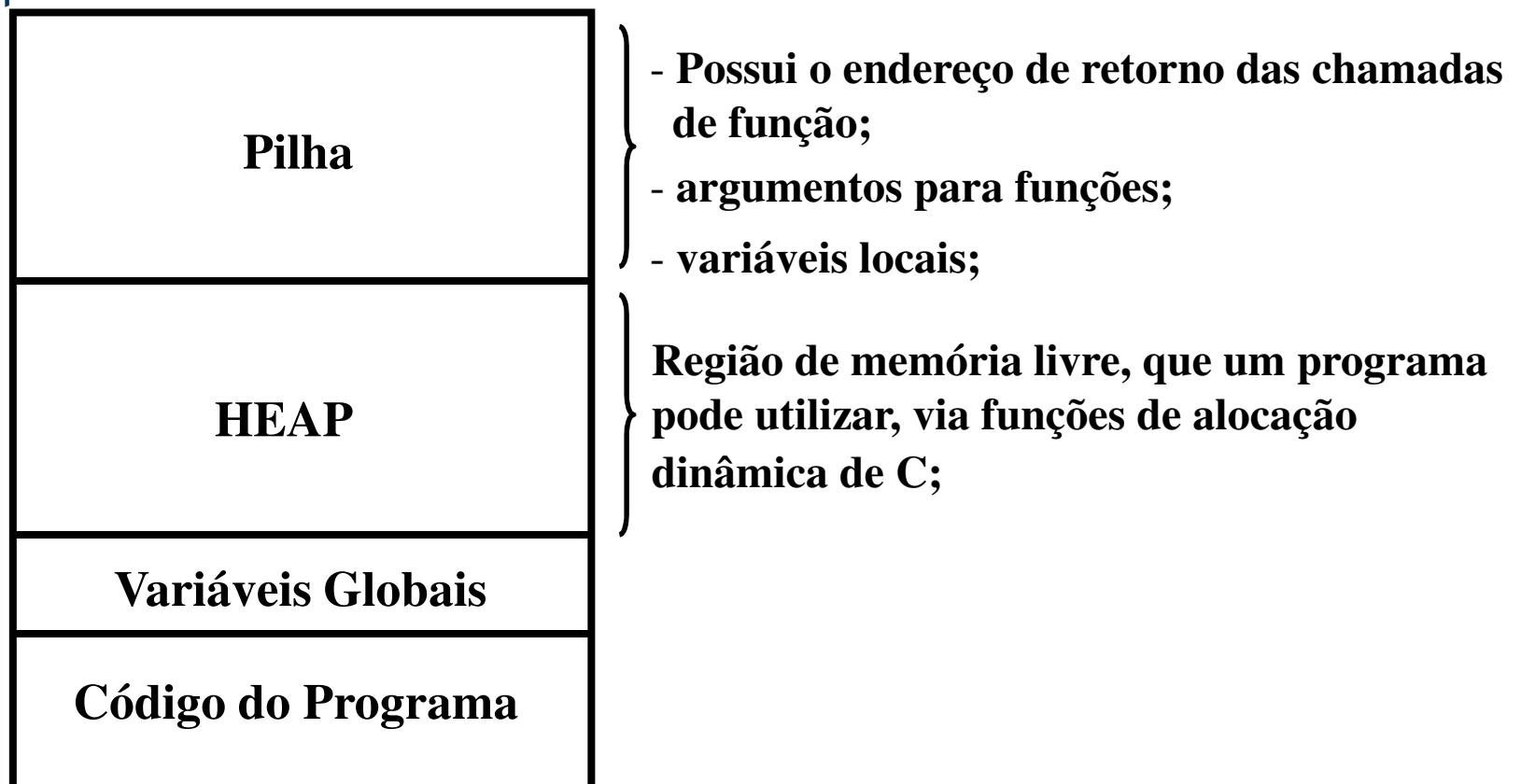
# Função free()

- Libera o uso de um bloco de memória, permitindo que este espaço seja reutilizado.
- Deve ser passado para a função free() exatamente o mesmo endereço retornado por uma chamada da função malloc()
- A determinação do tamanho do bloco a ser liberado é feita automaticamente.

Ex:

```
int *p;  
p = (int*) malloc(100 * sizeof(int));  
free(p);
```

# Mapa de Memória



Mapa conceitual de memória de um programa em C

# Função calloc()

- Função equivalente a malloc(), usada para alocar espaço para um vetor de elementos
  - `calloc(10, sizeof(int)) ≡ malloc(10 * sizeof(int))`
- Devemos informar, por parâmetro, o tamanho do vetor e o tamanho (em bytes) de cada elemento desse vetor.
  - O espaço alocado é iniciado com bits 0
- Esta função também retorna um ponteiro para void (`void*`)
  - Devemos converter o tipo genérico retornado para o tipo desejado

Ex:

```
Aluno *a;  
a = (Aluno *) calloc(10, sizeof(Aluno));
```

# Função realloc()

- Função utilizada para modificar o tamanho ocupado por uma área de memória já alocada
- Devemos informar, por parâmetro, a referência da área a ser redimensionada, e o novo tamanho (em bytes).
  - Se a área original não puder ser redimensionada, uma nova área é criada, e a antiga é liberada.
- Esta função também retorna um ponteiro para void (void\*)

Ex:

```
int *a, *b;  
a = (int *)malloc(sizeof(int));  
b = (int *)realloc(a, sizeof(int)*4);
```

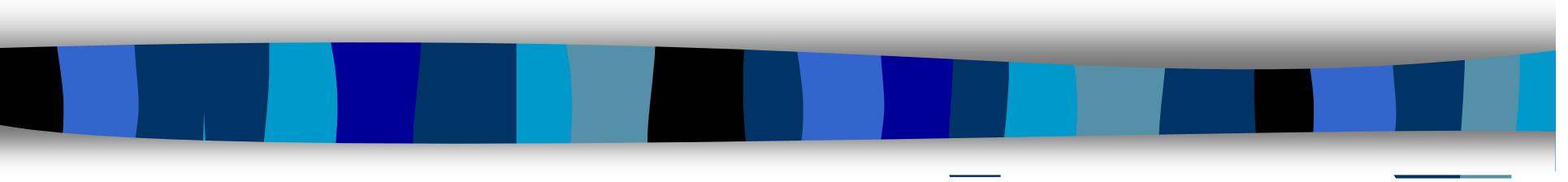
OBS:      `realloc(NULL, size) ≡ malloc(size)`

# Alocação para Matrizes

- No caso de vetores multidimensionais (e.g. matrizes), devemos alocar um vetor de apontadores, i.e. um apontador por linha, e depois um vetor de elementos para cada linha

Ex:

```
float **mat; /* matriz de elementos do tipo float */
int nlin = 10, ncol=10; /* numeros de linhas e colunas */
mat = (float **)calloc(nlin,sizeof(float *));
       /* aloca vetor de ponteiros para variaveis float */
if (mat != NULL) {
    int i;
    for (i=0; i < nlin; i++) {
        mat[i] = (float *)calloc(ncol,sizeof(float));
               /* aloca vetor de variaveis float */
    }
}
```



# *Ponteiros e Alocação (exercícios)*

**Digite o Exemplo 9.3:-** Faça um programa que receba as notas de 5 alunos através de ponteiros e mostre a media.

1	#include <stdio.h> #include <stdlib.h>	Inclui protótipos
2	int main(){	Abre main()
3	float *notas;	Declara variável ponteiro
4	notas=(float *) malloc(5*sizeof(float));	Aloca 20 ( $5^*4$ ) bytes na memória e atribui o endereço do primeiro byte alocado para a variável ponteiro "notas"
5	for (int i=0;i<5;i++){ scanf("%f", notas+i); }	Carrega vetor a partir do usuário. (notas + i) é o endereço onde será alocado o que o usuário digitar
6	for (i=0;i<5;i++){ printf("%f\n", *(notas+i)); }	Imprime todas as notas. *(notas+i) é o conteúdo do endereço (notas+i)
7	free(notas);	Libera o espaço para outras tarefas usarem
8	system("pause");	Pára o programa
9	}	Fechamain()

# Exercício 1:

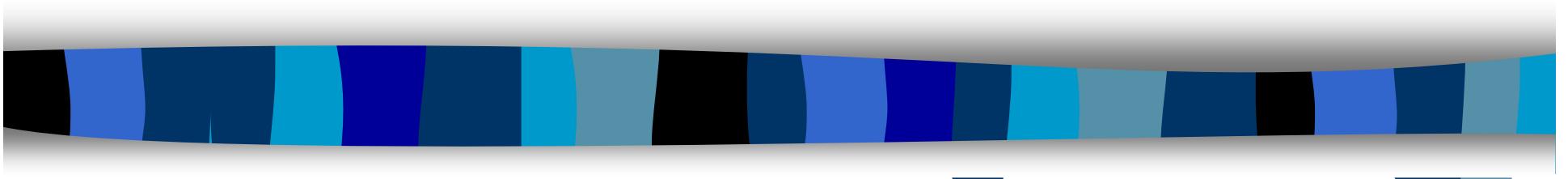
- Escreva uma função de protótipo:  
***void strinv (char s[]);***  
que inverta os caracteres de uma string. Por exemplo: se a string for “ABCDEF”, deve ser convertida para “FEDCBA”.
- Escreva um programa que use a função e receba a string do usuário.

# Exercício 2:

- Implemente um programa que:
  1. Leia dois vetores de inteiros, cujos tamanhos são definidos pelo usuário
  2. Chame uma função que os receba como parâmetro
    - a. A função deve retornar a concatenação dos dois
  3. No final, o programa deve exibir o vetor retornado, com os elementos dos dois vetores lidos.



# *Aula 4*



*Tipo Abstrato de Dados*



# Introdução:

- Programas consistem em 2 coisas:
  1. Algoritmos e
  2. Estruturas de dados (ED)
- Um bom programa é uma combinação de ambos;
- A escolha e a implementação de uma ED são tão importantes quanto as rotinas que manipulam os dados.
  - A forma como a informação é organizada e acessada é normalmente determinada pela natureza do problema de programação

# Introdução:

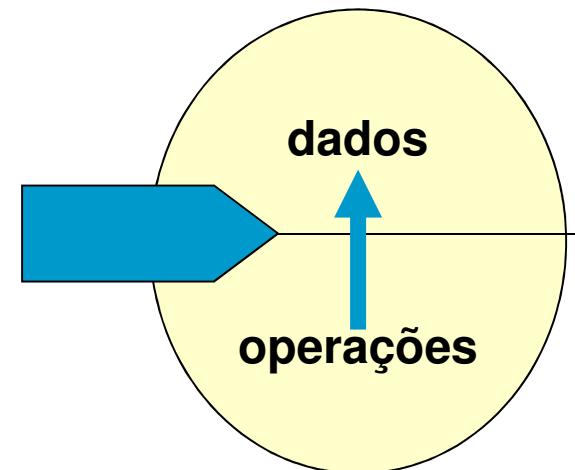
- Por essa razão, para cada tipo de representação de dados, devemos ter o método correto para a manipulação da informação
- Um “Tipo Abstrato de Dado” é a ferramenta certa para garantir que dados sejam manipulados da forma certa

# Definição:

- Tipo Abstrato de Dados é uma especificação de tipo contendo um **conjunto de dados** e **operações** que podem ser executadas sobre esses dados.
  - Descreve quais dados podem ser armazenados (características) e como eles podem ser manipulados (operações)
  - Mas não descreve como isso é implementado no programa
- TADs são implementados através de tipos compostos heterogêneos (Registros), associados a um conjunto de funções que operam sobre essa estrutura.
- Exemplo:

```
struct Estudante {  
    char nome [64]; int idade; char matricula[10]  
}  
int maiorDeIdade(Estudante estudante); //retorna 1 se verdadeiro  
void validaMatricula(Estudante estudante); // efetua matricula
```

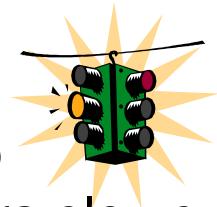
# Definição:



- Um TAD pode ser visto como um modelo matemático, acompanhado das operações definidas sobre o modelo
- Por exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação
- TADs podem ser considerados generalizações de tipos primitivos de dados (como tipo inteiro, real), da mesma forma que funções são generalizações de operações primitivas tais como adição, subtração e multiplicação

# Encapsulamento:

- Tipos Abstratos de Dados utilizam o conceito de **Encapsulamento**, para reduzir dependências entre as estruturas e garantir o comportamento correto das operações sobre os dados.
  - Encapsular: esconder aquilo que não deve ser manipulado diretamente.
- Importante: em um TAD, os dados armazenados são acessados sempre através das funções definidas para ele, e **nunca diretamente**
  - Independência: é possível alterar a estrutura interna de um TAD sem afetar código cliente
  - Consistência: as funções definidas para um TAD garantem que os dados serão sempre acessados na ordem e da forma corretas



# Exemplo:

## Mundo Real



Pessoa

## Dados de Interesse

- Idade da Pessoa

## ESTRUTURA de Armazenamento

- Tipo Inteiro

## Possíveis OPERAÇÕES

- Nasce (idade = 0);
- Aniversário  
(idade = idade + 1)

# Exemplo:

## Mundo Real



Cadastro de  
Funcionários

## Dados de Interesse

- Nome, Cargo  
e o salário de  
cada funcionário

## ESTRUTURA de Armazenamento

- Tipo Lista Ordenada

## Possíveis OPERAÇÕES

- Entra na Lista
- Sai da Lista
- Altera Cargo
- Altera Salário

# Exemplo:

## Mundo Real



Fila de Espera

## Dados de Interesse

- Nome de cada pessoa e sua posição na fila

## ESTRUTURA de Armazenamento

- Tipo Fila

## Possíveis OPERAÇÕES

- Sai da Fila (o primeiro)
- Entra na Fila (no fim)



# Vantagens:

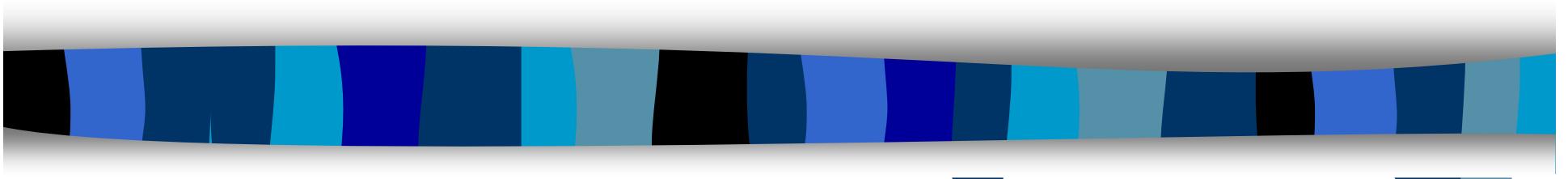
- Mais fácil programar (sem se preocupar com os detalhes de implantação);
- Mais seguro programar (apenas as operações do próprio TAD alteram as estruturas locais);
- Maior independência e portabilidade de código (alterações na implementações de um TAD não precisam alterar funcionalidades do sistema);
- Maior potencial de reutilização de código (pode-se alterar a lógica de um programa sem a necessidade de reconstruir um TAD);
- Conseqüência: **Custo Menor no Desenvolvimento do Software**

# Tipos:

- Existem 4 tipos importantes de TADs:
  1. Listas (Lineares e Encadeada)
  2. Pilha
  3. Fila
  4. Árvores Binárias
- Cada um desses TADs fornece uma solução para uma classe de problemas
- São essencialmente dispositivos que executam operações específicas de armazenamento e recuperação da informação
- Todos eles armazenam e manipulam itens de dados, onde um item é uma unidade de informação



# *Aula 4*



*Tipo Abstrato de Dados  
(Lab)*

# Alocação para TADs

- Sempre que trabalhamos com tipos abstratos, devemos ter funções para cada operação sobre essa estrutura
- Essas funções, normalmente, recebem por parâmetro uma cópia do TAD que devem processar
  - Sempre que passamos um TAD por parâmetro, criamos uma cópia dessa estrutura
  - Isso é particularmente ineficiente para TADs com muitas informações
  - É mais eficiente se a criação do TAD for realizado por alocação dinâmica, e referência a essa área for passada por parâmetro

```
Lista criaLista() {  
    /* 1000 posições */  
    Lista a;  
    a.ultimo = -1;  
    return a;  
}
```

```
Lista* criaLista(int size) {  
    Lista *a = (Lista *)malloc(sizeof(Lista));  
    a->lista = (int*)calloc(size, sizeof(int));  
    a->ultimo = -1;  
    return a;  
}
```

# Exercício 1:

- Sabemos que a linguagem C não possui o tipo String definido. Representamos textos como vetores de char e usamos funções de biblioteca para manipulá-los.
- Vamos construir um TAD “String”, que usa um vetor de char para representar o texto (como normalmente fazemos).
- Vamos também definir algumas operações para esse TAD:
  1. String criaString(): cria uma string vazia ('/0')
  2. String criaString(char c[]): cria string contendo c[]
  3. void append(String s1, String s2): apenda s2 em s1
  4. void addChar(String s1, char c): adiciona c no final de s1
  5. String substring(String s1, int ini, int final): retorna a substring delimitada entre ini e final, como uma nova String

(devemos usar alocação dinâmica para algumas tarefas)

# Exercício 2:

- Vamos supor que um número real seja representado por uma estrutura em C, como esta:

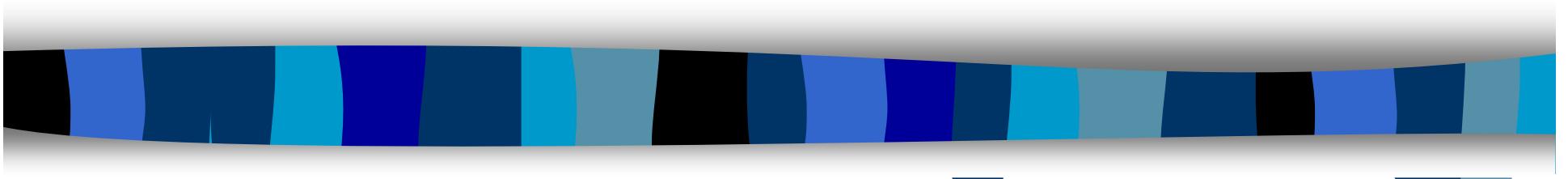
```
struct realtype {  
    int left;  
    int right;  
};
```

- onde *left* e *right* representam os dígitos posicionados à esquerda e à direita do ponto decimal  
- se *left* for um inteiro negativo, o número real representado será negativo.

- Vamos definir uma TAD “Real”. Ela deve possuir as operações:
  1. `criarReal()`: que receba um número real e crie uma estrutura representando esse número
  2. `retornaReal()`: função que aceita essa estrutura e retorna o número real representado por ela
  3. Rotinas `add`, `subtract` e `multiply` que aceitem duas dessas estruturas e retornem uma terceira estrutura para representar o número resultante das operações



# *Aula 5*



*Listas Encadeadas*

# Definições:

- Lista Linear: sequência de zero ou mais elementos  $a_1, a_2, \dots, a_n$  sendo
  - $a_i$  elementos de um **mesmo tipo**
  - $n$  o tamanho da lista linear ( $n = 0 \Rightarrow$  lista vazia)
- Propriedade fundamental: os elementos têm **relações de ordem** na lista
  - $a_i$  precede  $a_{i+1}$  (e  $a_i$  sucede  $a_{i-1}$ )
  - $a_1$  é o primeiro elemento da lista
  - $a_n$  é o último elemento da lista
- Representação de Listas Lineares: TAD com elementos organizados de maneira sequencial.
  - Os tipos mais comuns de listas lineares são as *pilhas* e as *filas*

# Operações:

- Uma lista linear define uma série de operações sobre seus elementos:
- **Criação**: Iniciar a lista como sendo vazia
- **Inserção**: Inserir um nó numa dada lista
- **Remoção**: Remover um nó de uma dada lista
- **Consulta**: Obter informações relacionadas a um dado nó de uma lista

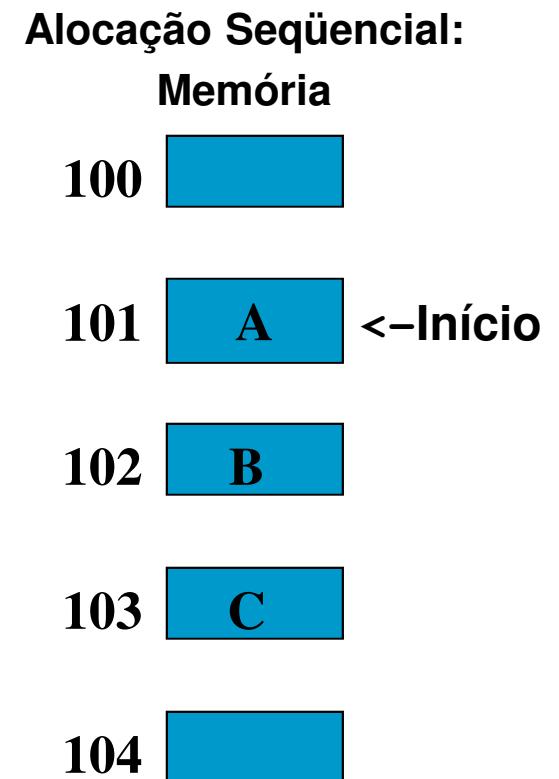
# Implementação:

- A implementação de uma Lista, usando uma linguagem de programação como o C, pode ser feita por meio de:
  1. Alocação Seqüencial e Estática de Memória (vetores)

ou
  2. Alocação Dinâmica de Memória (Ponteiros)

# Listas Estáticas:

- Quando definidas sobre vetores:
  - Elementos são alocados em seqüência
  - Seqüência “física”



# Listas Estáticas:

## Características:

- Dados armazenados em um vetor
- Declaração estática do vetor:
  - o tamanho deve ser suficiente para alocar toda a lista, prevendo aumentos futuros
  - não podemos alterar este tamanho em tempo de execução
- A lista estará alocada em posições contíguas na memória
  - a lista sempre terá um início no índice 0 (zero) e um fim que nem sempre coincidirá com o final do vetor
  - Necessário controlar o final da lista
- Adicionar ou remover elementos no meio da lista requer deslocamentos dos demais elementos (evitar espaços vazios)

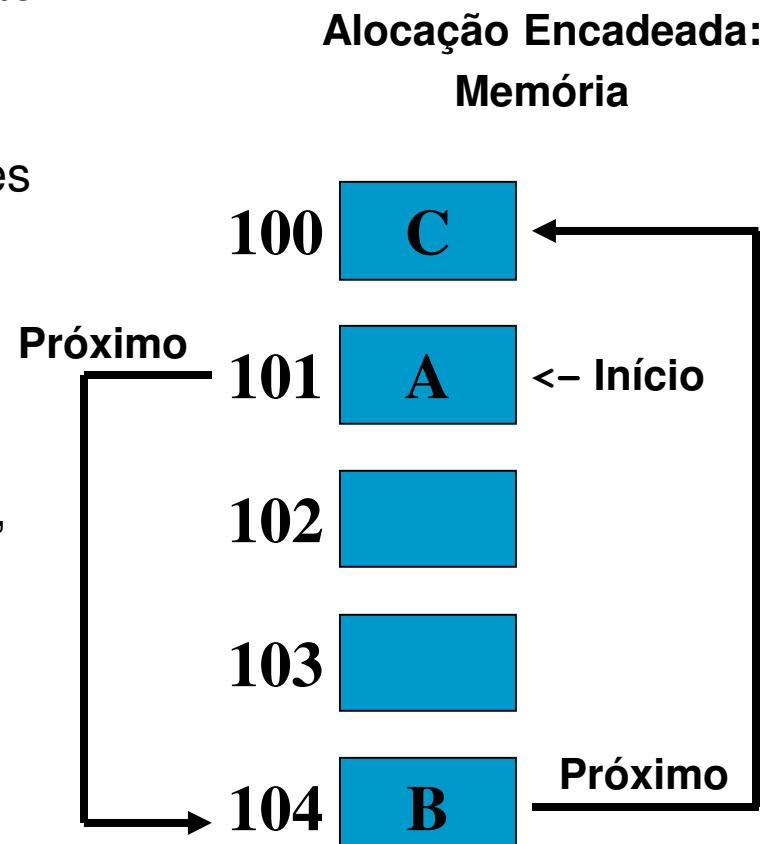


# Listas Encadeadas:

- Definição: uma Lista Encadeada também é um tipo de Lista Linear, ou seja, um TAD que possui elementos que mantém uma **relação de ordem** entre eles
- Entretanto, em uma lista encadeada a alocação da memória, necessária ao armazenamento dos elementos, é feita dinamicamente
- Isso significa que o próximo elemento da lista pode ser posicionado em qualquer lugar livre da memória
- Portanto, uma lista encadeada precisa de um mecanismo para garantir o sequenciamento de seus elementos

# Listas Encadeadas:

- Estrutura baseada na alocação dinâmica de memória:
  - Elementos **não** estão necessariamente em posições de memória adjacentes
  - Seqüência “lógica”
  - Para manter essa sequencia, uma lista encadeada faz uso de **ponteiros**



# Listas Encadeadas:

- Estrutura dinâmica, criada vazia
- Os elementos são chamados de “nós”
- Estrutura homogênea: os nós são todos do mesmo tipo
- Tamanho da lista é dado pelo número de nós da lista
- Condiciona o crescimento da lista à disponibilidade de memória
- Os nós não estão em seqüência na memória
- Os conceitos de ponteiros para registros e registros contendo ponteiros são muito úteis
  - Cada nó guarda: informações (info) e o endereço do próximo nó (prox)
  - Mantemos um ponteiro para o início da lista
  - O prox do último nó deve apontar para NULL

# Nós de Encadeamento

- Seja o nó definido abaixo:

```
typedef struct _node {  
    /* info */  
    struct _node *prox;  
} node;
```

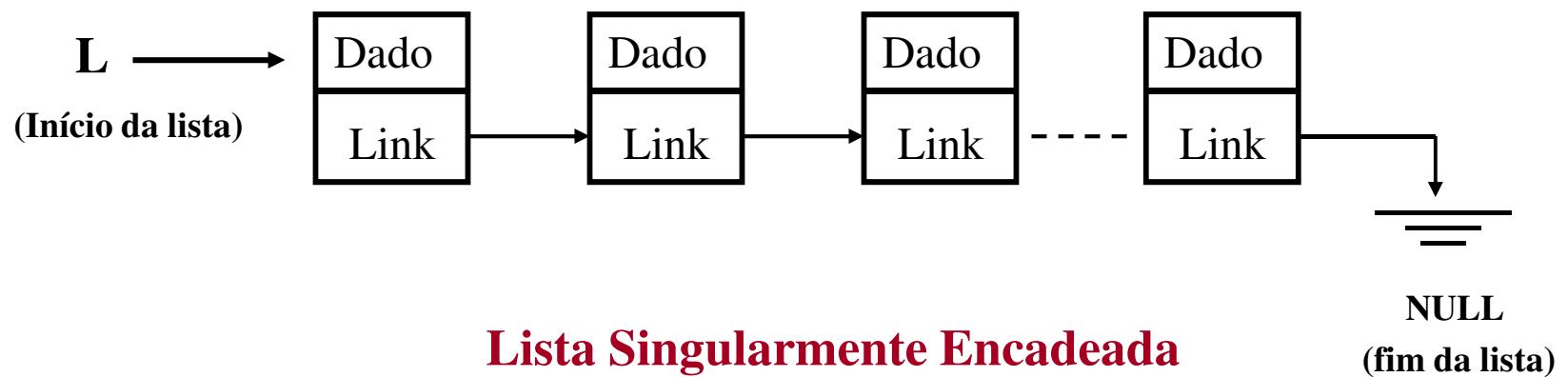
- Tal registro contém dois atributos:
  - “info” (principal): qualquer tipo conhecido (int, float, char, struct aluno, struct pessoa, etc.)
  - “prox”: ponteiro para um registro do tipo “node”. A idéia é poder apontar para o próximo registro da lista

# Tipo de Encadeamento:

- Geralmente, listas encadeadas são apresentadas em dois tipos:
- **Simplesmente Encadeada**: contém um elo com o próximo item de dado (usa apenas um ponteiro);
- **Duplamente Encadeada**: contém elos tanto com o elemento anterior quanto com o próximo elemento da lista (uso de duas variáveis do tipo ponteiro).

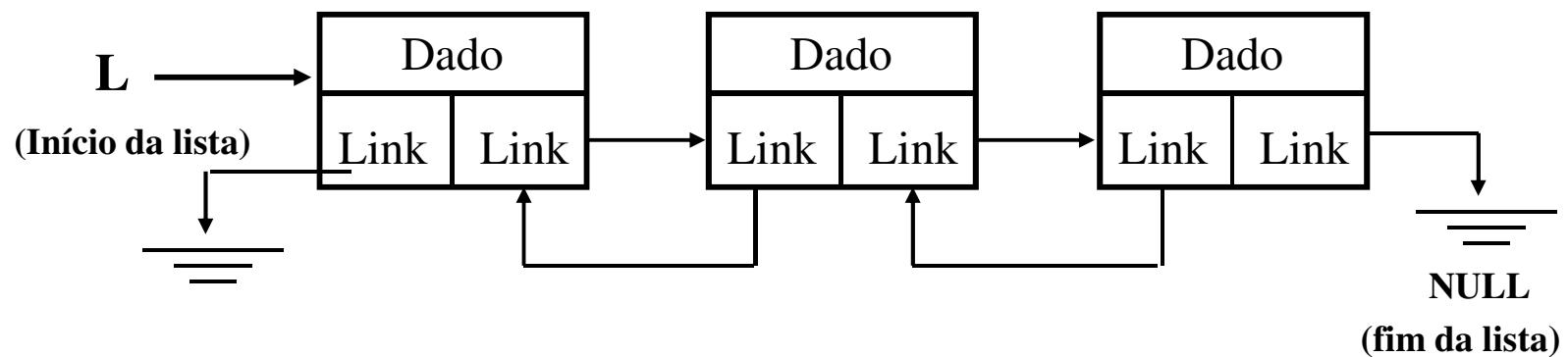
# Encadeamento Simples:

- Representação:



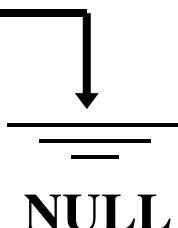
# Encadeamento Duplo:

- Representação:



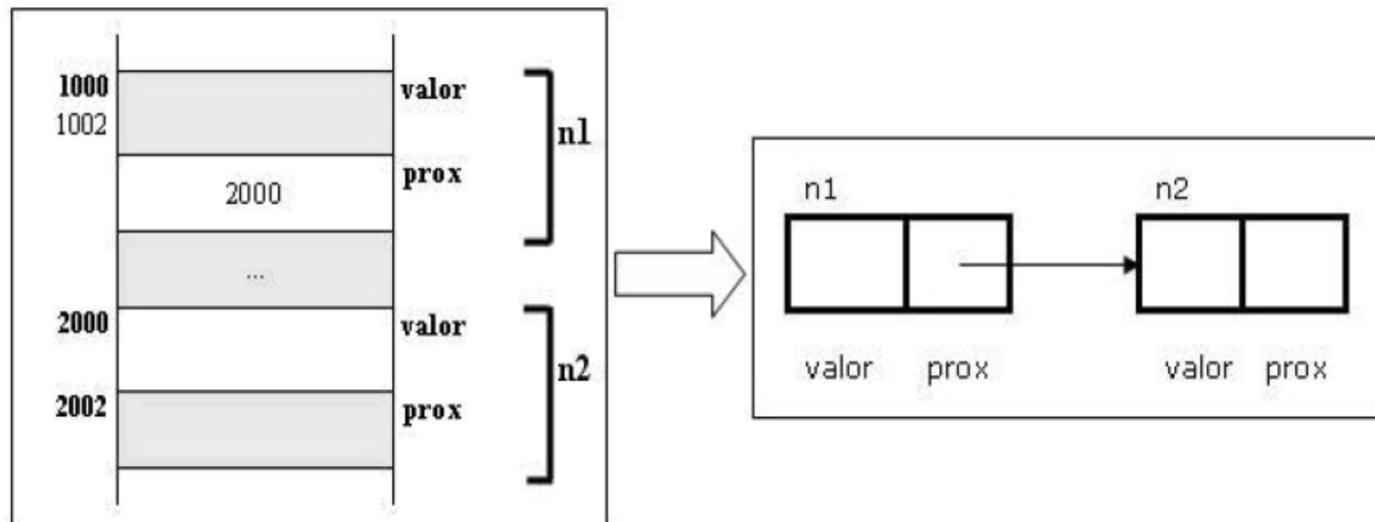
**Lista Duplamente Encadeada**

**Lista Vazia**



# Encadeamento

- Discutiremos o funcionamento do encadeamento simples
- A instrução `n1.prox = &n2` cria o encadeamento entre n1 e n2:

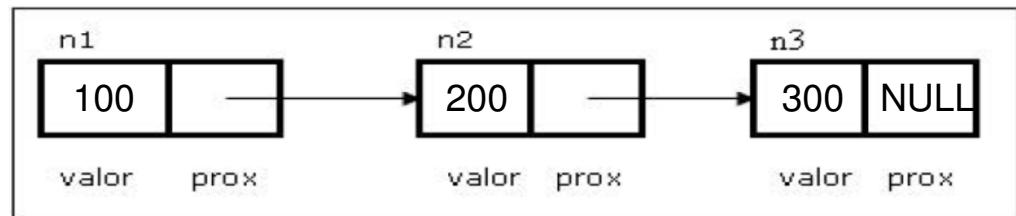


- O efeito de encadeamento pode, então, ser utilizado para a criação de uma lista, através da adição de novos nós

# Encadeamento

- Considere o código abaixo:

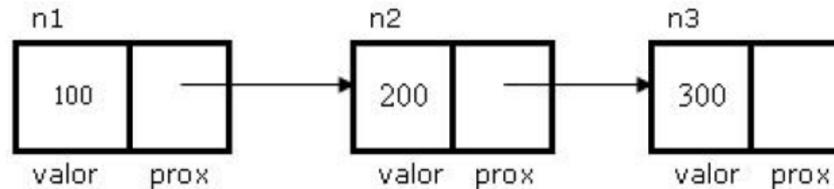
```
node n1, n2, n3;  
int i;  
  
n1.valor = 100;  
n2.valor = 200;  
n3.valor = 300;  
n1.prox = &n2;  
n2.prox = &n3;  
n3.prox = NULL;
```



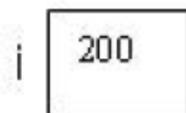
- Qual seria o resultado das instruções abaixo?

- a.  $i = (n1.prox) \rightarrow valor;$
- b.  $i = n1.prox.valor;$
- c.  $n1.prox = n2.prox;$

# Respostas



a. `i = (n1.prox) -> valor;`

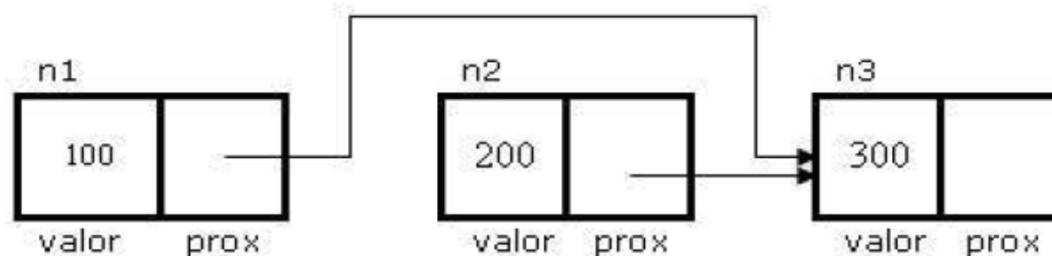


b. `i = n1.prox.valor;`

INCORRETO!

a. `n1.prox = n2.prox;`

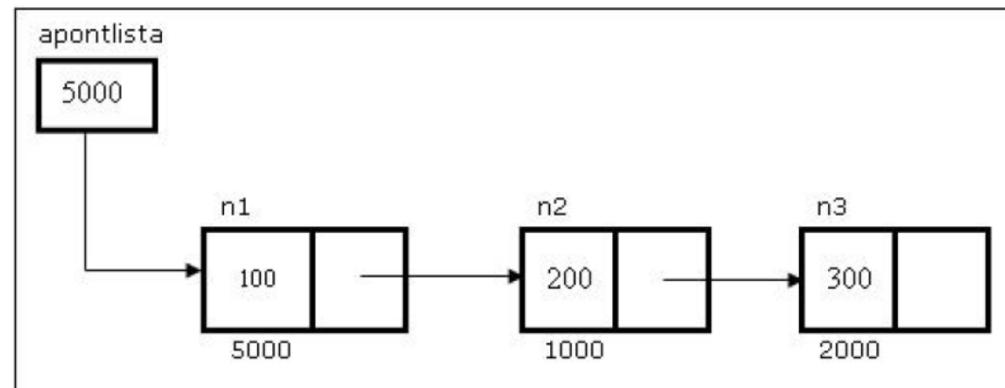
Remove o elemento n2 da lista



# Considerações

- Em geral, associamos a uma lista encadeada pelo menos um ponteiro para o primeiro elemento.
  - Ponteiro pra o início da lista: ponteiro cabeçalho (header pointer)

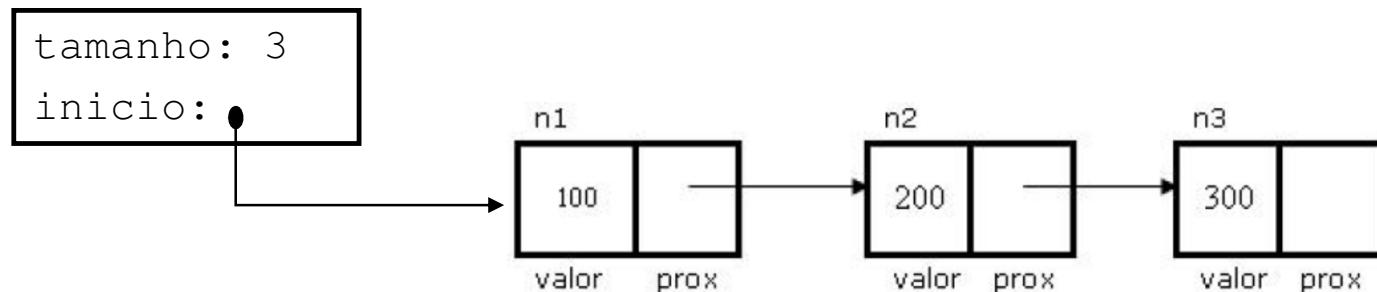
```
node *apontlista;  
.  
.  
.  
apontlista = &n1;
```

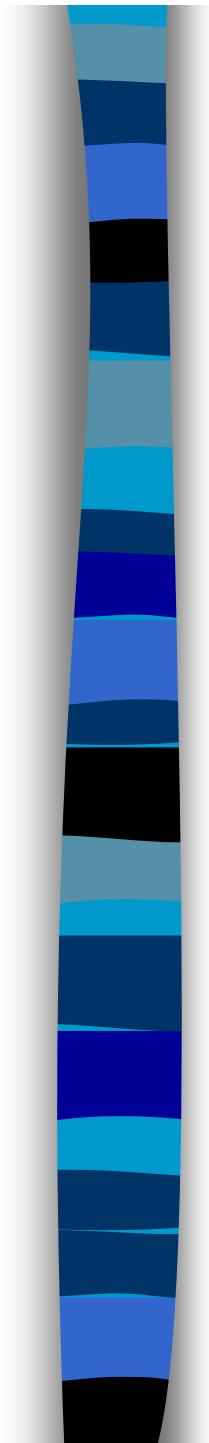


# Considerações

- Muitas vezes é interessante armazenar outras informações sobre a lista. Ex: tamanho da lista
- Para isso, podemos utilizar um “nó” (estrutura) para indicar o início da lista, mas que contenha as informações desejadas

```
typedef struct _headerNode {  
    int tamanho;  
    node *inicio;  
} headerNode;
```





# *Aula 6*



*Métodos de Ordenação*



# Ordenação

- Os algoritmos de ordenação são bons exemplos de como resolver problemas utilizando computadores
- Técnicas de ordenação permitem apresentar um conjunto amplo de algoritmos distintos para resolver uma mesma tarefa
- Dependendo da aplicação, cada algoritmo considerado possui uma vantagem particular sobre os outros;
- Além disso, os algoritmos ilustram muitas regras básicas para manipulação de estruturas de dados

# Ordenação

- Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descentende;
- O **objetivo** principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado
- Mais formalmente, dada uma seqüência  $s_1, s_2, \dots, s_n$  o processo de ordenar esta seqüência em ordem crescente consiste em obter uma permutação  $i_1, i_2, \dots, i_n$  dos índices  $1, \dots, n$  tal que

$$s_{i_1} \leq s_{i_2} \leq \cdots \leq s_{i_n}$$



# Conceitos Gerais

- Os métodos de ordenação variam entre si em complexidade e eficiência
- Geralmente aumentar a sua eficiência implica em aumentar também a sua complexidade
- Duas categorias:
  - **Interna**: quando se trata de registros em memória e
  - **Externa**: quando os dados a serem ordenados encontram-se em dispositivos de memória auxiliar (disco, fitas, etc..)



# Ordenação por Permutações

- Ordenação por permutação tem como característica específica a troca de posições entre elementos a serem ordenados
- O processo se repete até que não haja mais elementos a serem permutados, situação na qual se obteve a completa ordenação do conjunto
- Alguns métodos populares:
  1. Bubble Sort
  2. Selection Sort
  3. Quicksort



# Bubble Sort

- Classificação mais conhecida entre os principiantes em programação
- O método é chamado classificação por bolha porque cada número “borbulha” lentamente para a posição correta
- Estratégia:
  1. Percorrer o vetor várias vezes
  2. Comparar os elementos  $v[i]$  e  $v[i+1]$
  3. Trocar os 2 elementos se eles não estiverem na ordem correta
- Vantagem: simplicidade
- Desvantagem: é o método menos eficiente

# Bubble Sort

- Exemplo

25 48 37 12 57 86 33 22

25 48 37 12 57 86 33 22 (25x48)

25 48 37 12 57 86 33 22 (48x37) troca

25 37 48 12 57 86 33 22 (48x12) troca

25 37 12 48 57 86 33 22 (48x57)

25 37 12 48 57 86 33 22 (57x86)

25 37 12 48 57 86 33 22 (86x33) troca

25 37 12 48 57 33 86 22 (86x22) troca

25 37 12 48 57 33 22 86

Final do 1º “ciclo”: o maior elemento (86) está na posição final

# Bubble Sort

- Algoritmo:

```
n = length(A)  
repeat  
    swapped = 0  
    for i = 1 to n-1 do  
        if A[i-1] > A[i] then  
            swap(A[i-1], A[i])  
            swapped = i  
        end if  
    end for  
    n = swapped  
until not swapped
```



# Selection Sort

- Estratégia:
  1. Fixar uma posição (índice)
  2. Comparar o seu conteúdo com o conteúdo de todas as outras posições subsequentes
  3. Sempre que o programa encontrar um conteúdo menor (classificação ascendente) que o contido na posição fixada, trocamos os conteúdos das posições e continua
- Resultado: os menores valores são deslocados para as primeiras posições
- Vantagem: simplicidade
- Desvantagem: esta técnica torna-se ineficiente à medida que o conjunto de valores aumenta

# Selection Sort

25 48 37 12 57 86 33 92

12 48 37 25 57 86 33 92

12 25 37 48 57 86 33 92

12 25 33 48 57 86 37 92

12 25 33 37 57 86 48 92

12 25 33 37 48 86 57 92

12 25 33 37 48 57 86 92 (Ordenado)

# Quicksort

- Estratégia:
  1. Dividir o problema de ordenar um conjunto com  $n$  itens em dois problemas menores
  2. Os problemas menores são ordenados independentemente
  3. Depois os resultados são combinados para produzir a solução do problema maior
- Vantagem: é o algoritmo de ordenação interna mais rápido para uma ampla variedade de situações (mais utilizado)
- Desvantagens:
  - Ineficiente para dados já ordenados
  - A implementação é muito sensível à falhas
  - Método não é estável



# Quicksort

## Divisão do problema:

- Escolhemos um elemento para “pivô” e dividimos a lista
- Percorremos a parte esquerda comparando cada elemento com o pivô
  - Pára quando encontra um elemento maior que o pivô
- Mesmo procedimento do lado direito
  - Pára quando encontra elemento menor que o pivô
- Com estes dois elementos em mãos, trocamos suas posições
- Continuamos até que todos os elementos da esquerda sejam menores que o pivô e os elementos da direita sejam maiores

# Quicksort

## Propagação do método:

- Com o problema dividido, aplicamos o mesmo método às duas sublistas:
  - Escolhemos um pivô para cada sublista
  - Dividimos cada uma em duas novas
  - Repetimos para cada nova sublista
  - Quando as sublistas tiverem tamanho 1, a lista estará ordenada
  - Remontamos a lista ordenada a partir das sublistas

# Quicksort

## Algoritmo:

1. Escolha arbitrariamente um item do vetor e coloque-o em  $x$
2. Percorra o vetor a partir da esquerda até encontrar  $A[i] > x$
3. Percorra o vetor a partir da direita até encontrar  $A[j] \leq x$
4. Como os dois itens  $A[i]$  e  $A[j]$  estão fora de lugar no vetor final então troque-os de lugar;
5. Continue esse processo até que os apontadores  $i$  e  $j$  se cruzem em algum ponto do vetor.

# Quicksort

Exemplo:

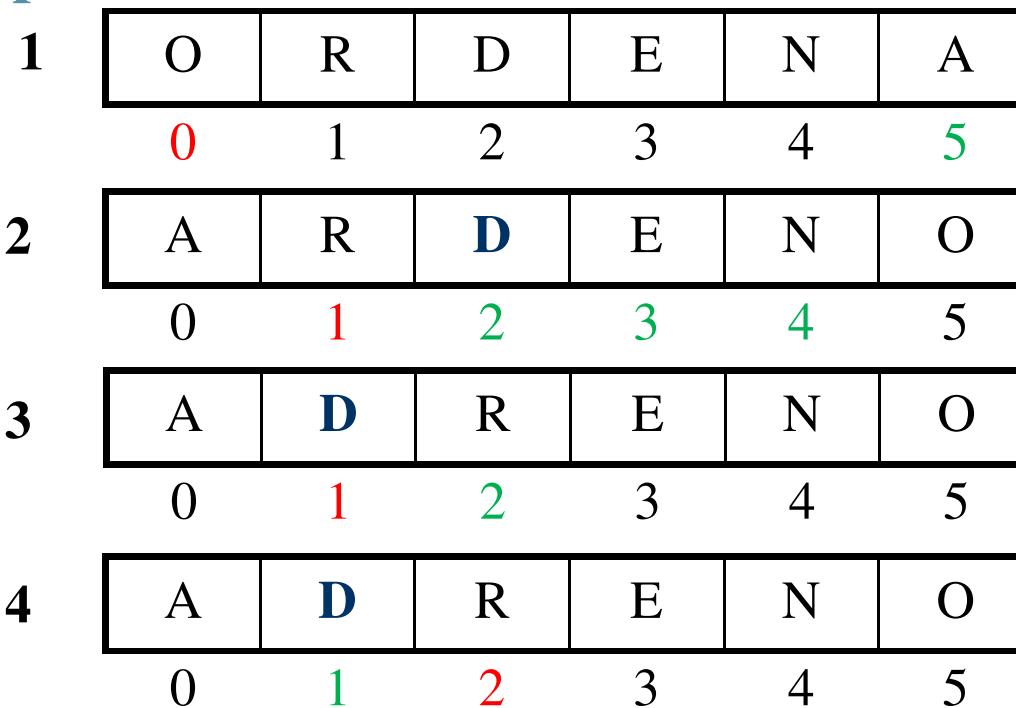
Vetor A =

O	R	D	E	N	A
0	1	2	3	4	5

1. O item x é escolhido como sendo  $A[ (i+j) / 2 ]$
2. Como  $i = 0$  e  $j = 5$ , então  $x = A[2] = D$
3. A varredura a partir da posição 0 para no item O e a varredura a partir da posição 5 para no item A;
4. Os itens então são trocados;

# Quicksort

Exemplo:



# Quicksort

Exemplo:

1	A	D	R	E	N	O
	0	1	2	3	4	5

- Neste momento, i e j se cruzam ( $i = 2$  e  $j = 1$ ), o que encerra o processo de partição
- Todo o processo é repetido até ordenar o vetor da seguinte forma:

A	D	E	N	O	R
0	1	2	3	4	5

# Atividade:

## Implementação:

- Implementar função “quicksort” para ordenar um vetor de inteiros, seguindo os requisitos abaixo:
  1. Versão recursiva do método
  2. Implementar a função “particao” que:
    - i. Escolhe o pivot como a mediana entre *prim*, *mid* e *ult*  
Mediana: ordenar os itens e escolher o do meio)
    - ii. Realiza a partição “*in-place*” (sem usar vetor auxiliar)
  3. Entrega: 09 / 05 /12

# Ordenação por Inserções

- Os métodos de inserção procuram colocar cada elemento em seu lugar diretamente
- A seguir, dois exemplos nesta classe:
  1. Insertion Sort
  2. Shell Sort

# Insertion Sort

- Estratégia:
  1. Inicialmente, o método ordena os dois primeiros elementos da matriz
  2. Em seguida, o algoritmo insere o terceiro membro na sua posição ordenada com relação aos dois primeiros elementos
  3. O processo continua até que todos os elementos tenham sido ordenados
- Vantagem: simples e eficiente para conjuntos de dados pequenos ou parcialmente ordenados. É um método estável.
- Desvantagem: perde eficiência rapidamente, conforme o conjunto de dados aumenta.

# Insertion Sort

## Algoritmo

6 5 3 1 8 7 2 4

```
for i ← 1 to length(A)−1
    key ← A[ i ]
    j ← i − 1
    while j >= 0 and A [ j ] > key
        A[ j + 1] ← A[ j ]
        j ← j −1
    A [ j + 1] ← key
```

# Shell Sort

- É assim chamada devido ao seu inventor, Donald L. Shell
- Aplica a ideia da inserção em vetores pequenos ou parcialmente ordenados
- Estratégia:
  1. Inicia o processo com incrementos (distância entre elementos comparados) grandes
  2. Reduz sistematicamente o tamanho dos incrementos a cada passagem
  3. Quando o incremento é 1, funciona como um *insertion sort*
- Vantagem: eficiente (mas com complexidade indeterminada)
- Desvantagem: não estável, pouco usado (em detrimento do Quicksort)

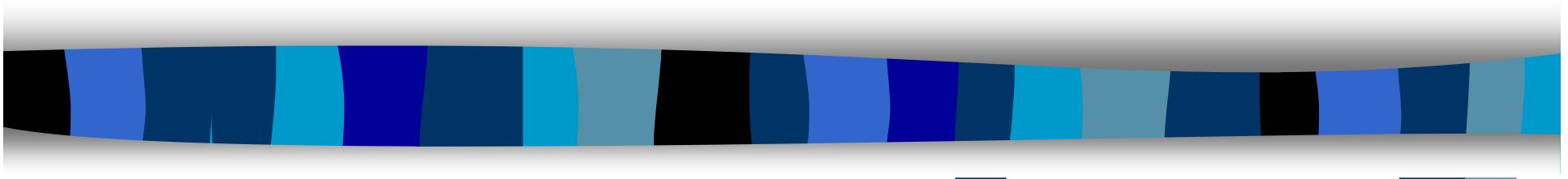
# Shell Sort

## Algoritmo

```
# Sort an array a[0...n-1].  
gaps = [701, 301, 132, 57, 23, 10, 4, 1]  
  
foreach (gap in gaps)  
    # Do an insertion sort for each gap size.  
    for (i = gap; i < n; i += 1)  
        temp = a[i]  
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap)  
            a[j] = a[j - gap]  
        a[j] = temp
```



# *Aula 7*



## *Recursividade*



# Definição

- Recursividade é uma técnica de fazer com que uma função ou procedimento chame a si mesmo
- A chamada recursiva é também conhecida como *chamada interna*
- Esta técnica deve ser empregada quando o problema em questão possui natureza recursiva:
  - O problema original pode ser decomposto em subproblemas
  - Cada subproblema possui a mesma lógica do problema original, só que aplicada a subconjunto dos dados
  - Ex: *Cálculo Fatorial*

# Cálculo Fatorial

- Especificação:  
 $n! = 1$ , se  $n == 0$   
 $n! = n * (n-1) * (n-2) * \dots * 1$ , se  $n > 0$ .
- Ex:  
 $5! = 5 * 4 * 3 * 2 * 1 = 120$
- Observamos que:  
 $5! = 5 * 4 * 3 * 2 * 1 = 120$   
 $4! = 4 * 3 * 2 * 1 = 24$   
 $3! = 3 * 2 * 1 = 6$   
 $2! = 2 * 1 = 2$   
 $1! = 1 = 1$

# Cálculo Fatorial

- Especificação:  
 $n! = 1$ , se  $n == 0$   
 $n! = n * (n-1) * (n-2) * \dots * 1$ , se  $n > 0$ .
- Ex:  
 $5! = 5 * 4 * 3 * 2 * 1 = 120$
- Observamos que:  
 $5! = 5 * !4 = 120$   
 $4! = 4 * !3 = 24$   
 $3! = 3 * !2 = 6$   
 $2! = 2 * !1 = 2$   
 $1! = 1 = 1$
- Logo:  
 $5! = 5 * 4! = 5 * 24 = 120$



# Considerações

- Importante:
  - Toda função recursiva precisa definir, pelo menos, um critério de parada!
  - No caso do cálculo factorial, consideramos que, quando  $n = 1$ , cessamos as chama das recursivas e retornamos 1
  - Cada caso é reduzido a um caso mais simples até chegarmos ao caso  $1!$ , que é definido imediatamente como 1
  - É possível observar que funções recursivas podem ser criadas em quase todos os processos repetitivos para  $n$  elementos

# Exemplo:

## Soma dos n primeiros números inteiros

```
int Soma(int num);
void main(){
    int num, result;
    printf("Digite um nºmero qualquer: ");
    scanf("%d", &num);
    result = Soma(num);
    printf("Resultado = %d", result);
}

int Soma(int num){
    int aux;
    if (num == 0) aux = 0;
    else aux = num + Soma(num-1);
    return aux;
}
```

# Exercício:

- Proponha uma solução recursiva para o Cálculo Fatorial

# Exercício:

```
int Fatorial(int num){  
    int aux;  
    if (num < 0) return ERRO;  
    if (num == 1 || num == 0) aux = 1;  
    else aux = num * Fatorial(num-1);  
    return aux;  
}  
  
void main(){  
    int fat, result;  
    printf("Digite um n mero qualquer: ");  
    scanf("%d", &fat);  
    result = Fatorial(fat);  
    printf("Resultado = %d", result);  
    getch();  
}
```



# Mais Considerações

- Dissemos que a técnica de recursão é aplicável quando a solução de um problema envolve sub-soluções de mesmo procedimento (aplicação repetida da mesma lógica)
- Um procedimento recursivo terá pelo menos, dois passos fundamentais:
  1. Um passo onde o resultado é imediatamente conhecido
  2. Outro passo onde teremos a chamada do mesmo procedimento.
- Dissemos também que é uma técnica apropriada quando o problema a ser resolvido ou os dados a serem tratados são definidos em termos recursivos
- Entretanto, essa máxima não funciona sempre

# Sequência de Fibonacci

- A seqüência de fibonacci é a seqüência de inteiros:  
0, 1, 1, 2, 3 , 5, 8, 13 , 21, 34, .....
- Cada elemento nessa seqüência é a soma dos dois elementos anteriores, como por exemplo,  $0 + 1 = 1$ ,  $1 + 1 = 2$ ,  $1 + 2 = 3$ ,  $2 + 3 = 5$ , ...
- Se estabelecermos que  $\text{fib}(0) = 0$ ,  $\text{fib}(1) = 1$ , então poderemos definir a seqüência de Fibonacci por meio da seguinte definição recursiva:  
$$\begin{aligned}f(n) &= n \text{ se } n == 0 \text{ ou } n == 1 \\f(n) &= \text{fib}(n-2) + \text{fib}(n-1) \text{ se } n >= 2\end{aligned}$$

# Sequência de Fibonacci

```
int Fibo(int num) {
    int aux;
    if (num < 0) return ERROR;
    if (num <=1) aux = num;
    else aux = Fibo(num - 2) + Fibo(num - 1) ;
    return aux;
}
```

# Sequência de Fibonacci

```
int Fibo(int num) {
    int aux;
    if (num < 0) return ERROR;
    if (num <=1) aux = num;
    else aux = Fibo(num - 2) + Fibo(num - 1) ;
    return aux;
}
```

- Observações:
  - A definição recursiva refere-se a si mesma 2 vezes
  - Ocorre redundância computacional ao aplicar a definição
  - É possível calcular  $\text{fib}(n)$  através de um método iterativo e esse método é muito mais eficiente

# Sequência de Fibonacci

```
int Fibo(int num) {  
    int aux, fib0 = 0, fib1 = 1;  
    if (num <=1) return num;  
    for(int i=2; i<=num; i++) {  
        aux=fib0;  
        fib0=fib1;  
        fib1=aux+fib0;  
    }  
    return fib1;  
}
```

# Sequência de Fibonacci

- Comparação:

N	10	20	30	50	100
Recursivo	8 ms <small>w</small>	1s	2 min	21 dias	$10^9$ anos <small>w</small>
Iteração	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

- Conclusão:
  - Devemos evitar o uso de recursividade quando existe uma solução iterativa óbvia

# *Aula 8*



*Pesquisa em Memória  
Primária*



# Introdução

- Pesquisa em Memória Principal: como recuperar informação a partir de uma grande massa de informação previamente armazenada
- A informação é dividida em **registros**. Um registro é um conjunto de informações relacionadas entre si. Um conjunto de registros é chamado de Tabela ou arquivo
- Os dados de uma nota fiscal, de um cadastro de funcionários ou de uma conta bancária são agrupados em um registro, em cada caso



# Introdução

- Cada registro possui uma chave (que é uma informação que distingue aquele registro dos demais) para ser usada na pesquisa
- Objetivo: encontrar uma ou mais ocorrências de registros com chaves iguais à **chave de pesquisa**.
- Existe uma variedade de métodos de pesquisa.
- A escolha do método mais adequado a uma determinada aplicação depende sobretudo:



# Introdução

1. Da quantidade dos dados envolvidos
2. Da estabilidade do conjunto de dados
  - Pode estar sujeito a inserções e retiradas freqüentes, ou
  - O conteúdo pode ser praticamente estável (neste caso, minimizar o tempo de pesquisa sem se preocupar com o tempo necessário para estruturar o arquivo)
  - Vejamos algumas técnicas de pesquisa:

# Pesquisa Sequencial

- É o método de pesquisa mais simples que existe
- Funcionamento: a partir do primeiro registro, pesquise sequencialmente até encontrar a chave procurada ou o final do arquivo. Então pare.
- É o mais genérico dos métodos
- Método relativamente ineficiente, mas simples de implementar e não necessita de ordenações prévias
  - Pode ser aplicado em memória e em dispositivos auxiliares (discos, fitas, etc.)

# Pesquisa Sequencial

- Exercício: Faça um programa que
  - Carregue um vetor com 10 valores digitados pelo usuário
  - Receba um valor a ser procurado neste vetor
  - Faça uma busca seqüencial no mesmo
  - Indique em que posição do vetor o elemento foi encontrado, ou se não encontrou

```
#define TAM 10
int main(){
    int vet[TAM],pos=-1, i, elem;
    // Entrada dos dados pelo usuário
    for(i=0; i < TAM; i++){
        printf("Entre com o vetor de %d elementos: ", i+1, TAM);
        scanf("%d", &vet[i]);
    }
    // Busca Linear
    printf("\n Digite o elemento a ser procurado: ");
    scanf("%d", &elem);
    for(i=0;i<TAM;i++){
        if(elem == vet[i]){
            pos = i; break;
        }
    }
    // Resultado
    if(pos != -1){
        printf("\nElemento encontrado em: %d\n", pos);
    } else {
        printf("\n Elemento NAO Encontrado");
    }
    system("pause>null");
}
```

# Busca Binária

- Tipo de busca com melhor desempenho que a Busca Linear, entretanto, requer que os elementos estejam classificados (ex. ordem crescente)
- Funcionamento:
  1. Consultamos a posição central do vetor, verificando se o seu conteúdo é *igual, maior ou menor* que o elemento procurado.
  2. Igual: a busca é finalizada
  3. Menor: passo 1 é repetido na segunda metade do vetor
  4. Maior: passo 1 é repetido na primeira metade
  5. Repetimos até que o elemento seja encontrado

# Busca Binária

- Cada busca elimina a necessidade de comparação com metade dos elementos que restam a serem procurados
- Exemplo:

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

# Busca Binária

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Objetivo: encontrar o número 70 neste vetor
- Processo: criamos 3 pontos de controle: **inicio**, **meio** e **fim**
- Estes pontos de controle (variáveis) irão receber valores de acordo com o intervalo de busca.
- Por exemplo, da primeira vez estes valores serão:  $\text{inicio} = 0$  ,  $\text{fim} = 14$  e  $\text{meio} = (\text{inicio} + \text{fim}) /2$ .

# Busca Binária

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- O valor procurado (70) é comparado com o conteúdo do índice “meio” (40)
  - O valor é maior. Portanto descartamos a busca na primeira metade do vetor
- Agora, redefinimos os pontos de controle: inicio = 8, fim = 14 e meio = (inicio + fim) / 2 = 11
- Fazemos novamente a comparação e constatamos que 70 é menor que 60 (conteúdo do índice meio)

# Busca Binária

- Exercício:
- Faça um programa que carregue um vetor já classificado de 10 posições
- Receba um valor a ser procurado neste vetor
- Faça uma busca binária no mesmo
- Indique em que posição do vetor o elemento foi encontrado, ou se não encontrou

```
#define TAM 10
int main(){
    int vet[TAM], pos = -1, i, fim, meio, inicio, elem;
    vet = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50};
    printf("\n Digite o elmento a ser procurado: "); scanf("%d", &elem);

    inicio = 0; fim = TAM-1;

    while(inicio <= fim){
        meio = (fim+inicio)/2;
        if (elem > vet[meio]){
            inicio = meio+1;
        } else if (elem < vet[meio] ){
            fim = meio-1;
        } else {
            pos = meio; break;
        }
    }

    if(pos != -1){
        printf("\nElemento Encontrado: %d\n", pos);
    } else {
        printf("\n Elemento NAO Encontrado\n");
    }
    system("pause>null");
}
```

# *Aula 9*



*Árvores de Pesquisa*



# Árvore de Pesquisa

- ÁRVORE é uma estrutura de dados muito eficiente para armazenar informação
- Ela é particularmente adequada quando existe necessidade de considerar alguma combinação de requisitos tais como:
  1. Acesso direto e seqüencial eficientes
  2. Facilidade de inserção e retirada de registros
  3. Boa taxa de utilização de memória
  4. Utilização de memória primária e secundária



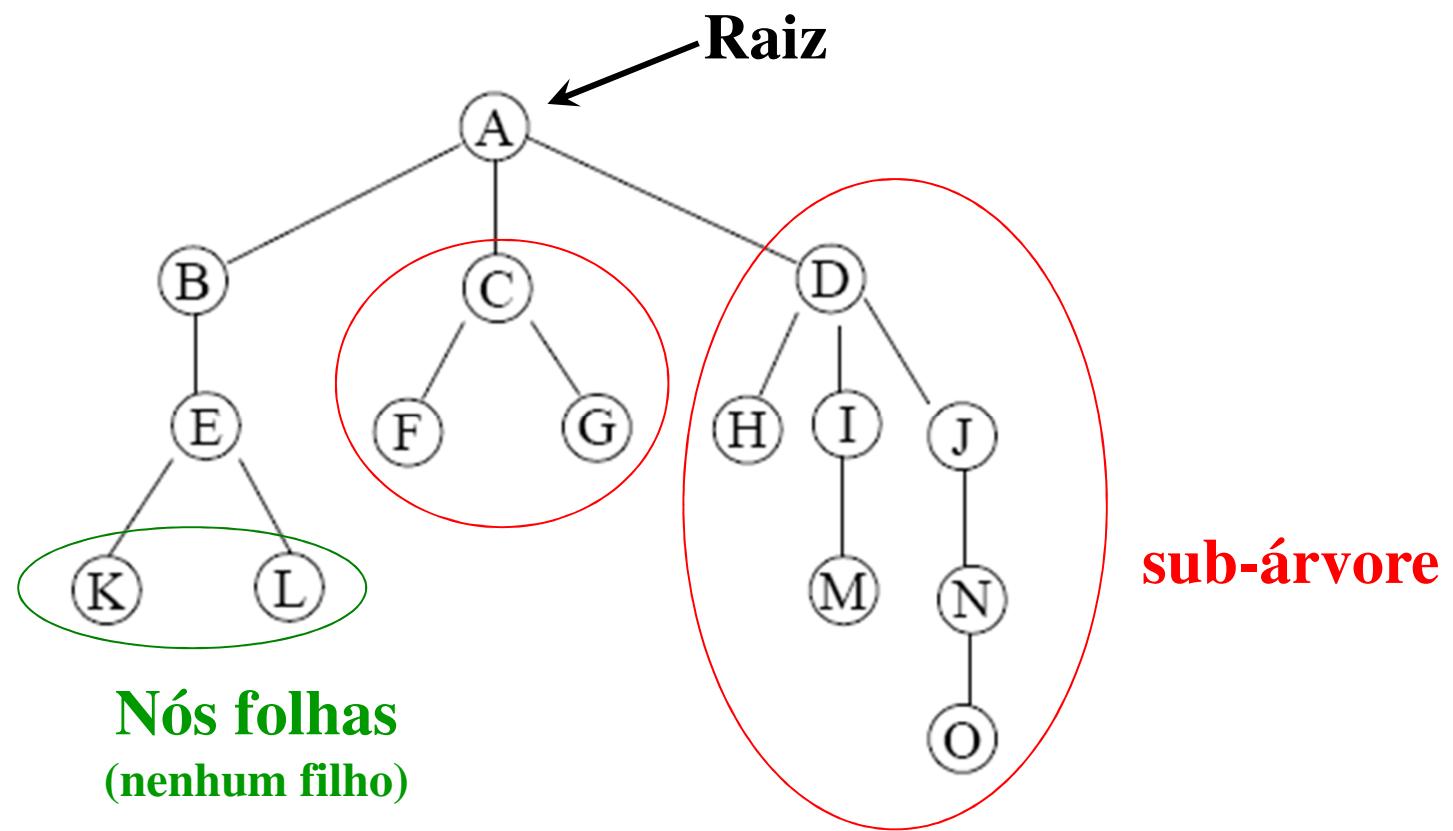
# Árvore de Pesquisa

## ÁRVORES

- Uma estrutura de dados em árvores tem sua organização semelhante a uma árvore (como seu próprio nome já diz), com sua raiz e folhas
- A cada informação guardada na árvore chamamos de **nó**
- Portanto, temos um único nó o qual podemos denominar “**raiz**”, sendo que, a partir dele podemos ter várias subárvore. Os nós de nível inferior são chamados de **nós filhos**

# Árvore de Pesquisa

## ÁRVORES



# Árvore de Pesquisa

## ÁRVORES

- Na árvore acima, temos 15 nós, sendo que o nó A é a “raiz”, tendo o nó B, C e D como “filhos”;
- Chamamos os nós K, L, F, G, H, M e O de “folhas” ou “nós terminais” pois não tem filhos, e os nós com filhos chamamos de “nós não terminais”;
- Grau de uma árvore: número máximo de subárvore em cada nó (no exemplo, 3);



# Árvore de Pesquisa

## ÁRVORES

- Nível de um Nó: o nó raiz tem nível zero; o nível de qualquer outro nó é um a mais que o do seu pai (ex. o nó A tem “nível” 0, os nós B, C e D são de nível 1, e assim por diante);
- Altura de uma Árvore: número de níveis da árvore (no exemplo, 5);
- Profundidade da Árvore: significa o nível máximo de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz até qualquer folha (no exemplo, 4);



# Árvore de Pesquisa

## Árvores Binárias

- Árvore Binária é um conjunto finito de elementos que está vazio ou é partitionado em 3 subconjuntos disjuntos:
  - Raiz da árvore: conjunto que contém um único elemento;
  - Os outros dois subconjuntos são também árvores binárias, chamadas de subárvores esquerda e direita da árvore original.



# Árvore de Pesquisa

## Árvores Binárias

- Uma subárvore esquerda ou direita pode estar vazia
- Cada elemento de uma árvore é chamado de nó da árvore
- Portanto, uma Árvore Binária é uma árvore onde cada nó tem, no máximo, dois filhos

# Árvore de Pesquisa

## Árvores Binárias

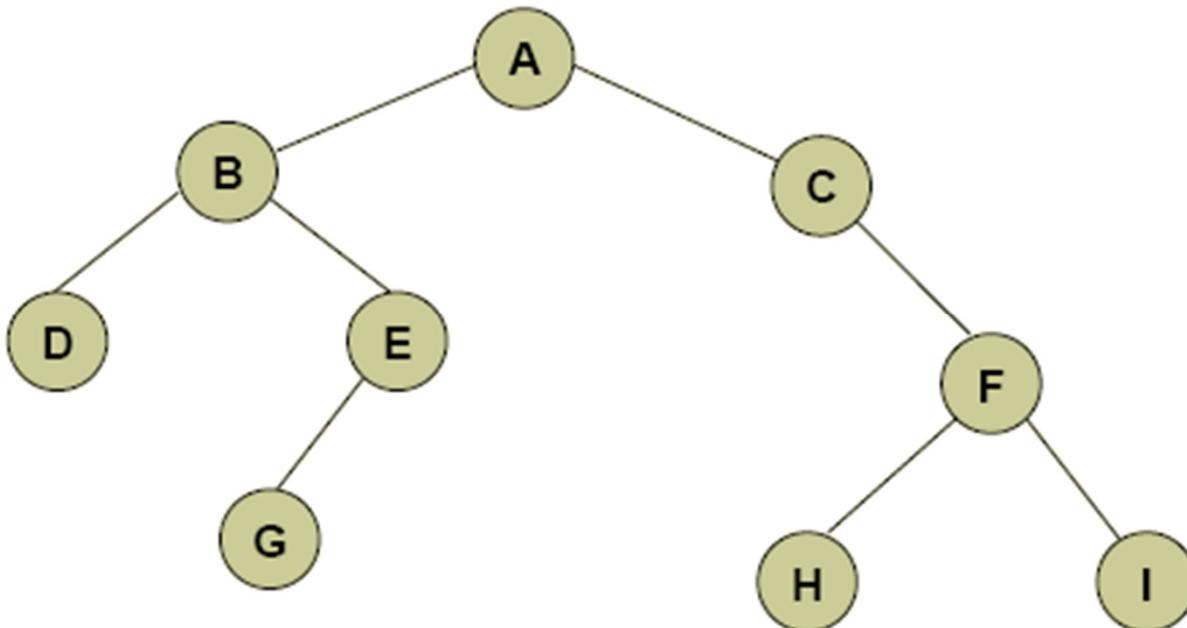


Figura 01: Exemplo de Árvore Binária

# Árvore de Pesquisa

## Árvores Estritamente Binárias

- Se todo nó que não é folha numa árvore binária tiver subárvores esquerda e direita não vazias, a árvore será considerada uma árvore estritamente binária
- Uma árvore estritamente binária com  $n$  folhas contém sempre  $2n - 1$  nós

# Árvore de Pesquisa

## Árvores Estritamente Binárias

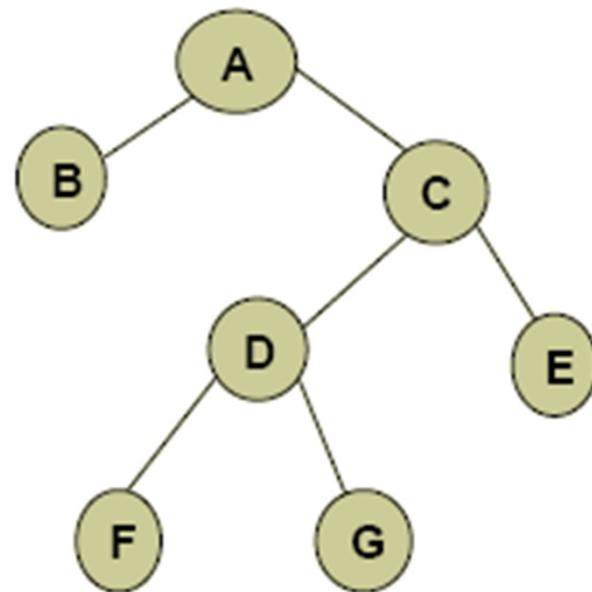


Figura 02: Árvore Estritamente Binária

# Árvore de Pesquisa

## Árvores Binárias Completas

- Uma árvore binária completa de profundidade **d** é a árvore estritamente binária em que todas as folhas estejam no nível **d**

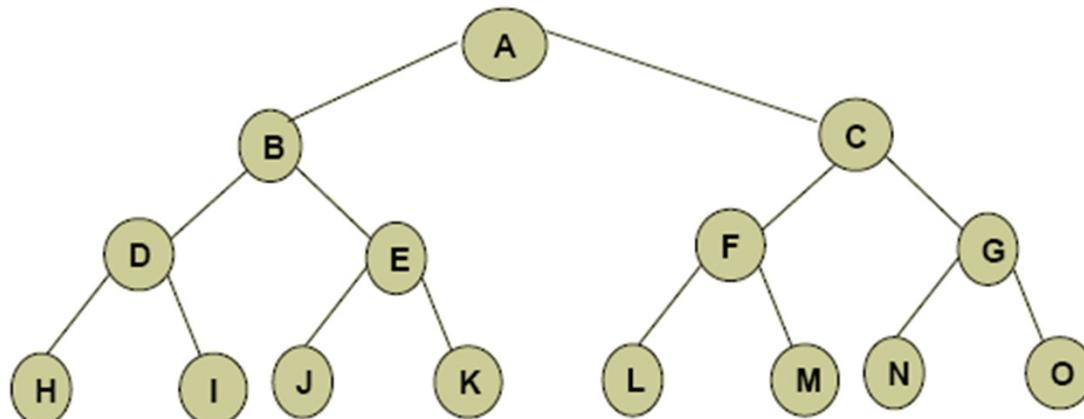


Figura 03: Árvore Binária Completa de nível 3

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Uma árvore binária com raiz R é uma **Árvore Binária de Pesquisa (ou Busca)** se:
  - a. A informação de cada nó da subárvore esquerda de R é menor que a informação do nó R
  - b. A informação de cada nó da subárvore direita de R é maior que a chave do nó R
  - c. As subárvores esquerda e direita também são Árvores Binárias de Pesquisa

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

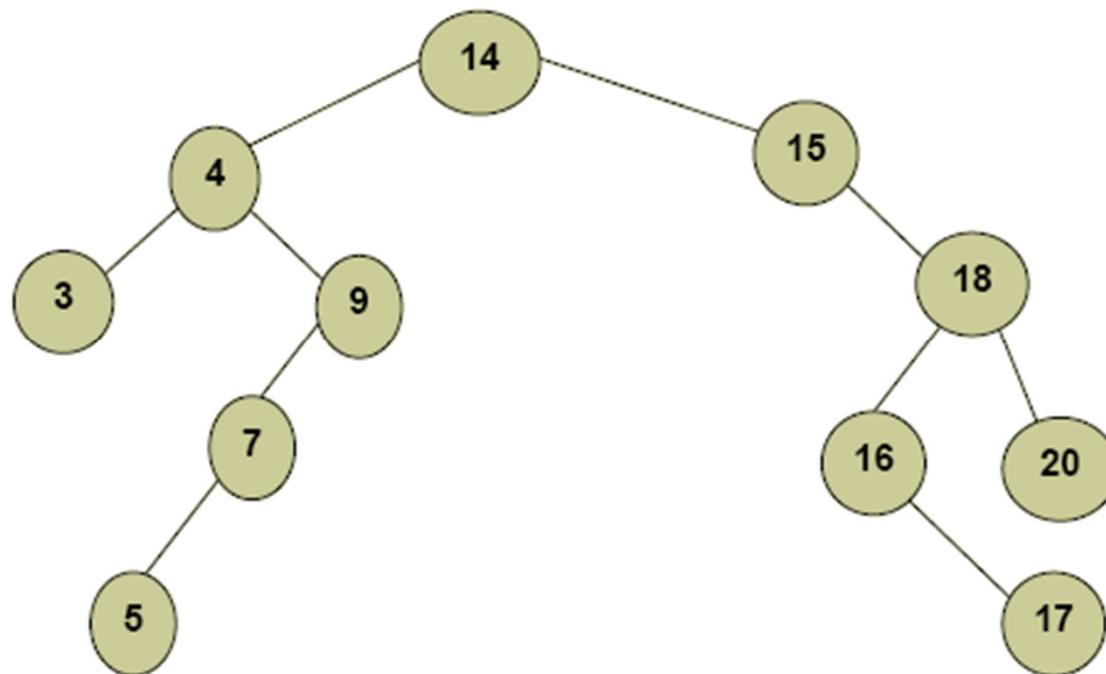


Figura 04: Árvore Binária de Busca (AAB)

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa Sem Balanceamento

- Exemplo de aplicação:
- Problema: Sistema de votação por telefone:
  - Cada número só pode votar uma vez;
  - Um sistema deve armazenar todos os números que já ligaram;
  - A cada nova ligação, deve-se consultar o sistema para verificar se aquele número já votou; o voto só é computado se o número ainda não votou;
  - A votação deve ter resultado on-line.



# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Solução:

- Uma maneira de solucionar o problema é comparar cada ligação com cada número já armazenado;
- Isso envolve um grande número de comparações;
- Este número pode ser reduzido usando-se uma Árvore Binária de Busca;
- O primeiro número na lista é colocado num nó estabelecido como a raiz de uma árvore binária com as subárvore esquerda e direita vazias;
- Cada número, então, é comparado ao número na raiz. Se coincidirem, teremos uma repetição;



# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

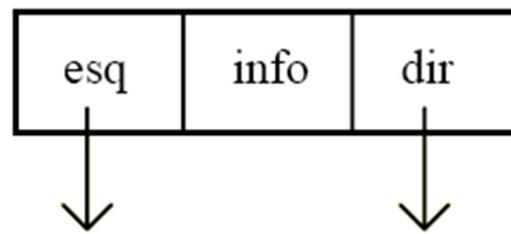
- Solução:

- Se for menor, examinaremos a subárvore esquerda; se for maior, examinaremos a subárvore direita;
- Se a subárvore estiver vazia, significa que o número não está repetido e será colocado num novo nó nesta posição na árvore;
- Se a subárvore não estiver vazia, compararemos o número ao conteúdo da raiz da subárvore e processo inteiro será repetido com a subárvore;

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Representação:



- info: contém a informação do nó;
- esq: endereço do nó filho à esquerda;
- dir: endereço do nó filho à direita;

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Definição do struct no\_arvore:

```
// Estrutura do nó da Árvore
struct no_arvore {
    no_arvore *esq;
    int info;
    no_arvore *dir;
};
```

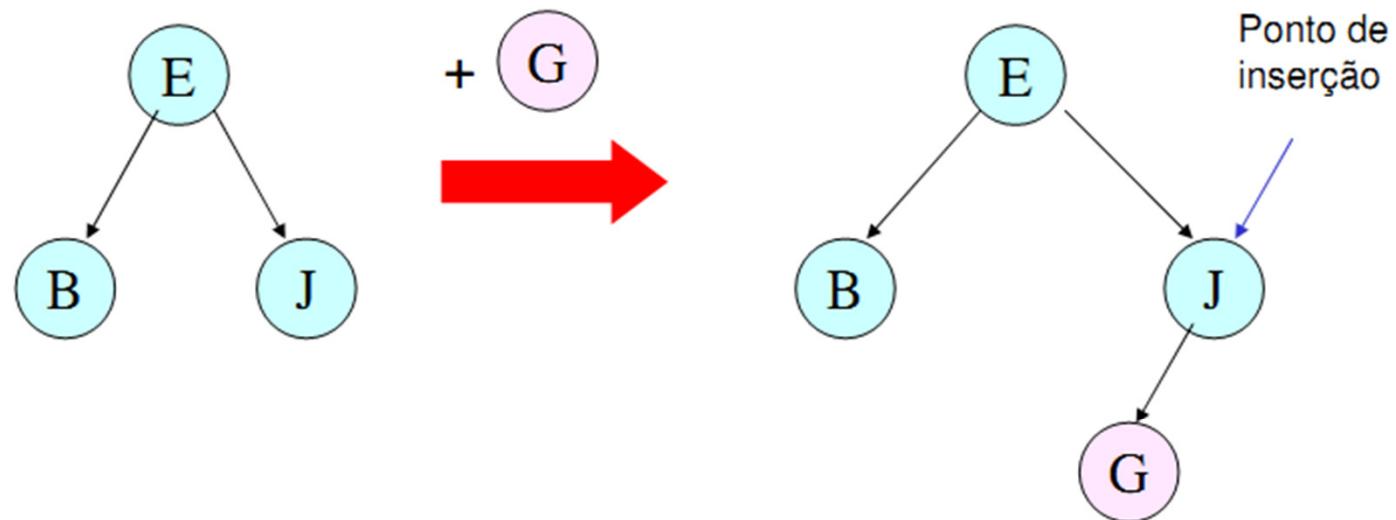
# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Exemplo de Implementação em C:
  - Operação **Inicializa()**: cria uma árvore inicialmente vazia.
  - Operação **Constrói()**: constrói uma Árvore Binária de Busca, usando o algoritmo descrito nos slides anteriores
  - A seguir, estratégias para inserção e remoção de nós

# Árvore de Pesquisa

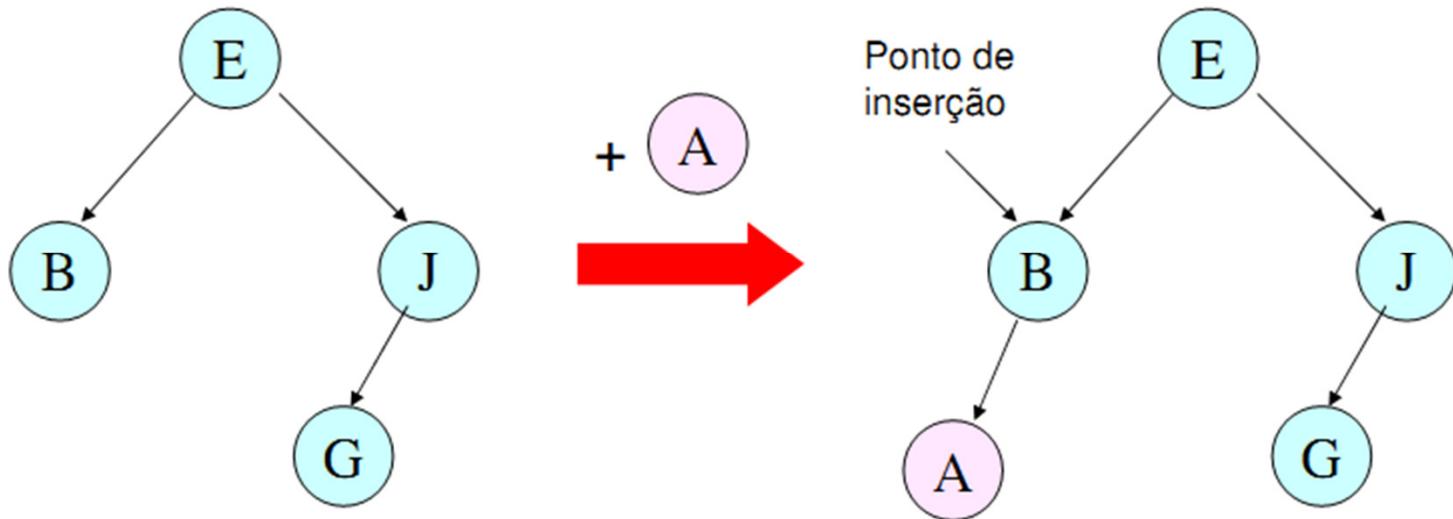
## Inserção de Novo Nô



- A localização do 'ponto de inserção' é semelhante à busca por um valor na árvore.
- Após a inserção do novo elemento, a árvore deve manter as propriedades de 'árvore binária de busca'.
- O nó inserido é sempre uma folha.

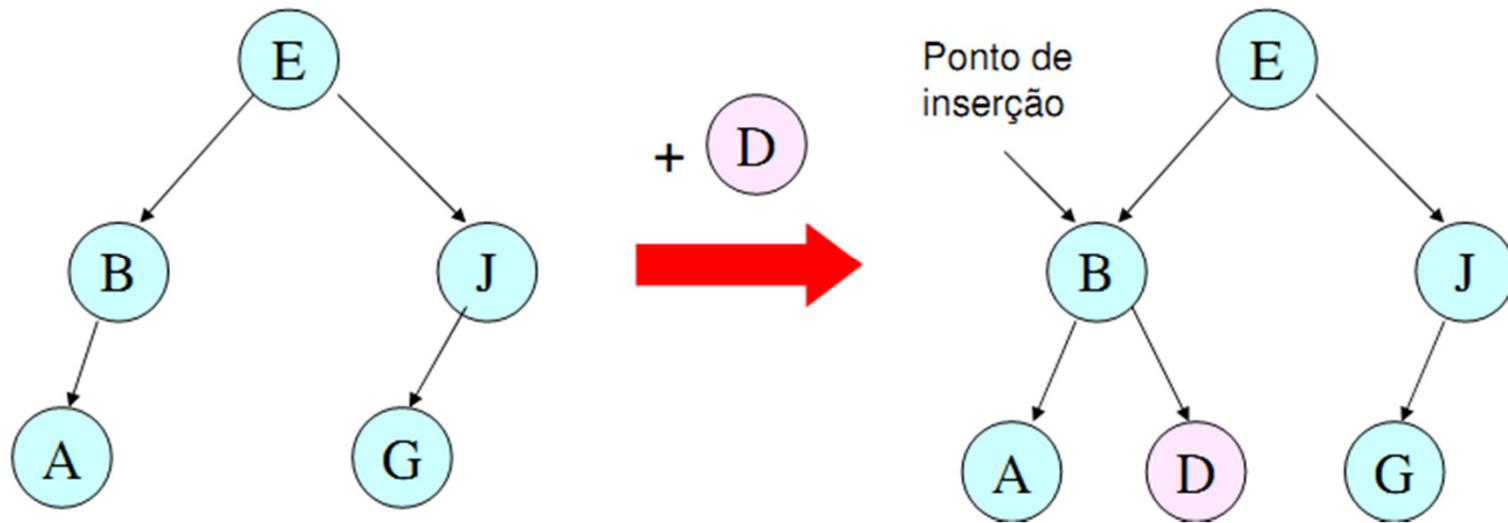
# Árvore de Pesquisa

## Inserção de Novo Nó



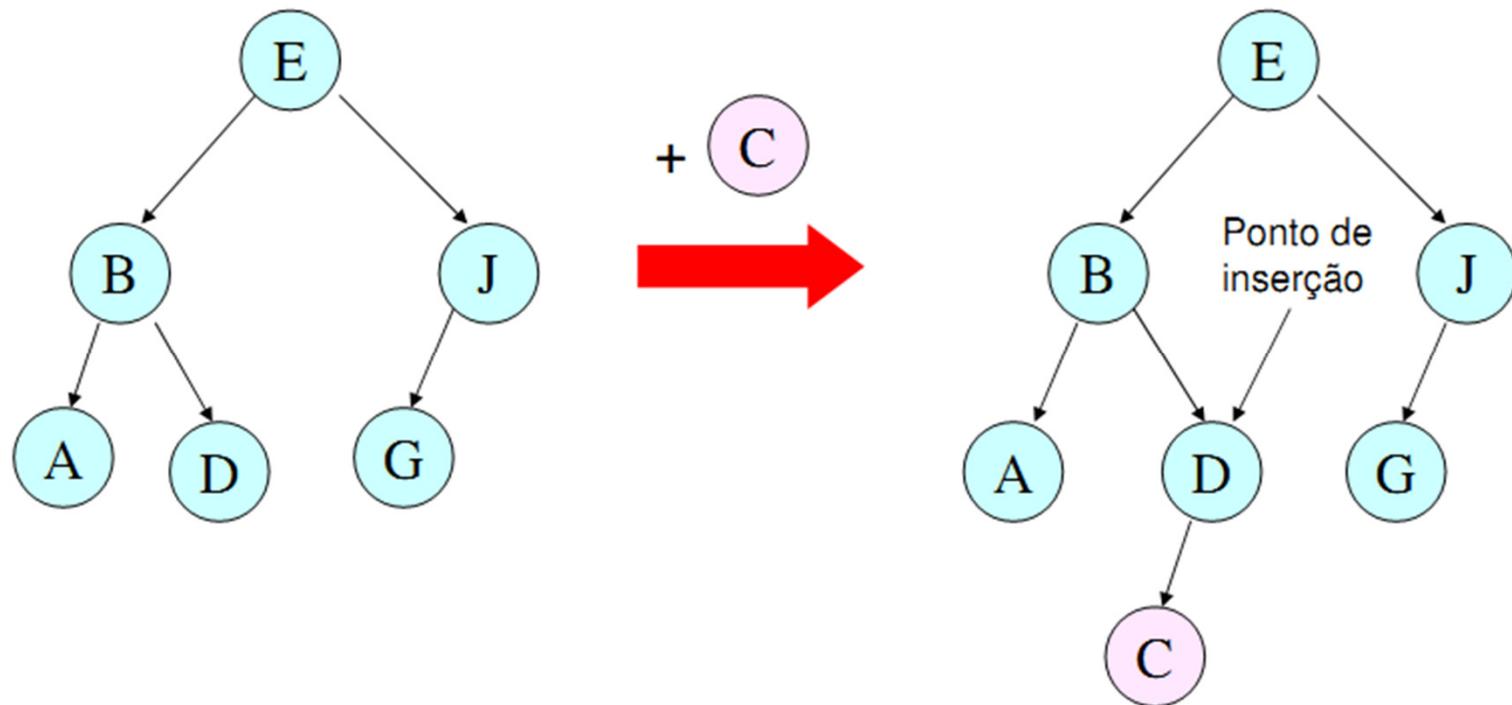
# Árvore de Pesquisa

## Inserção de Novo Nó



# Árvore de Pesquisa

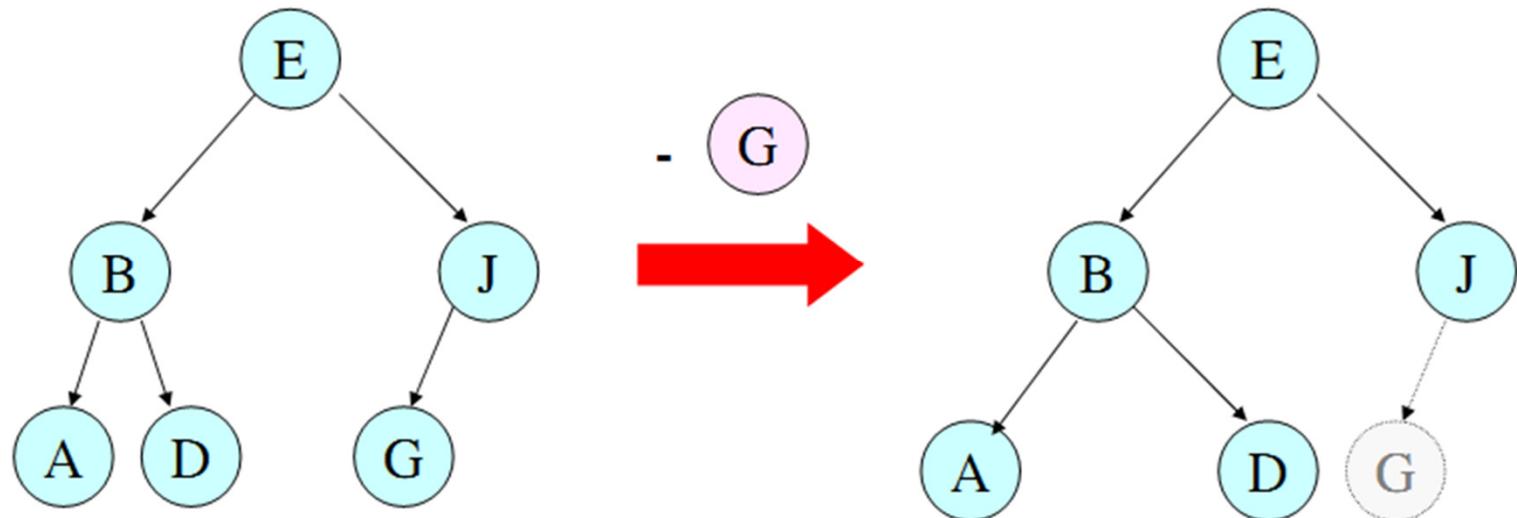
## Inserção de Novo Nó



# Árvore de Pesquisa

## Remoção de um Nó

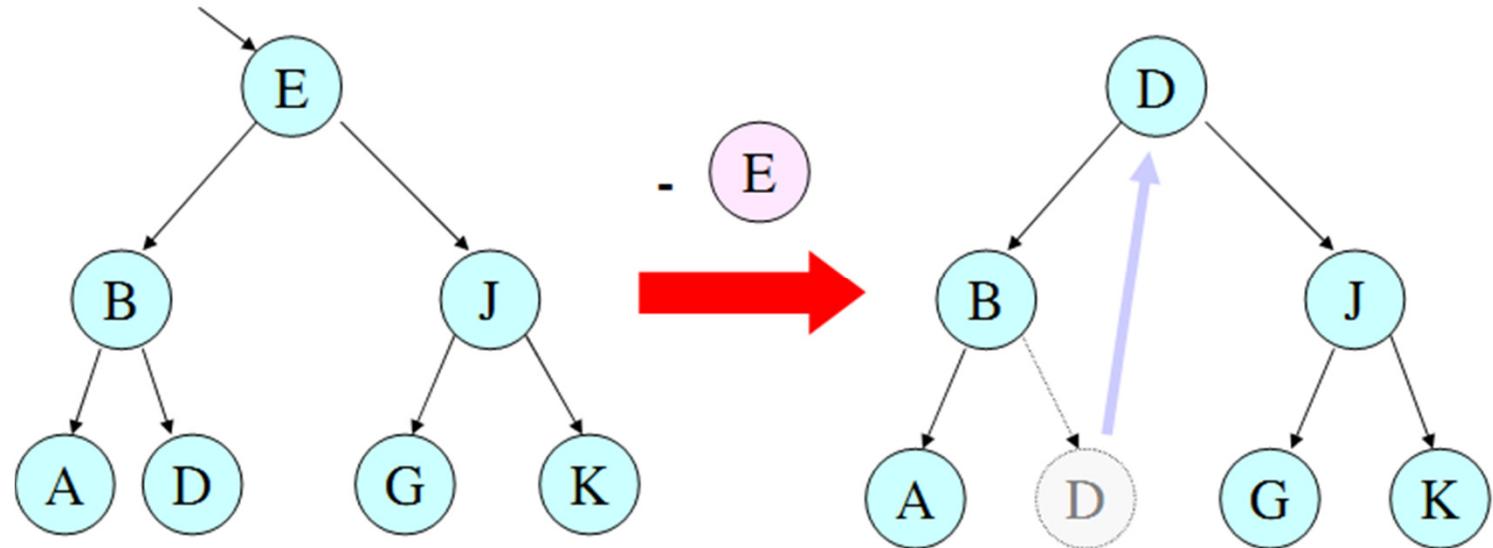
- Caso 1: nó X a ser removido é uma folha



# Árvore de Pesquisa

## Remoção de um Nó

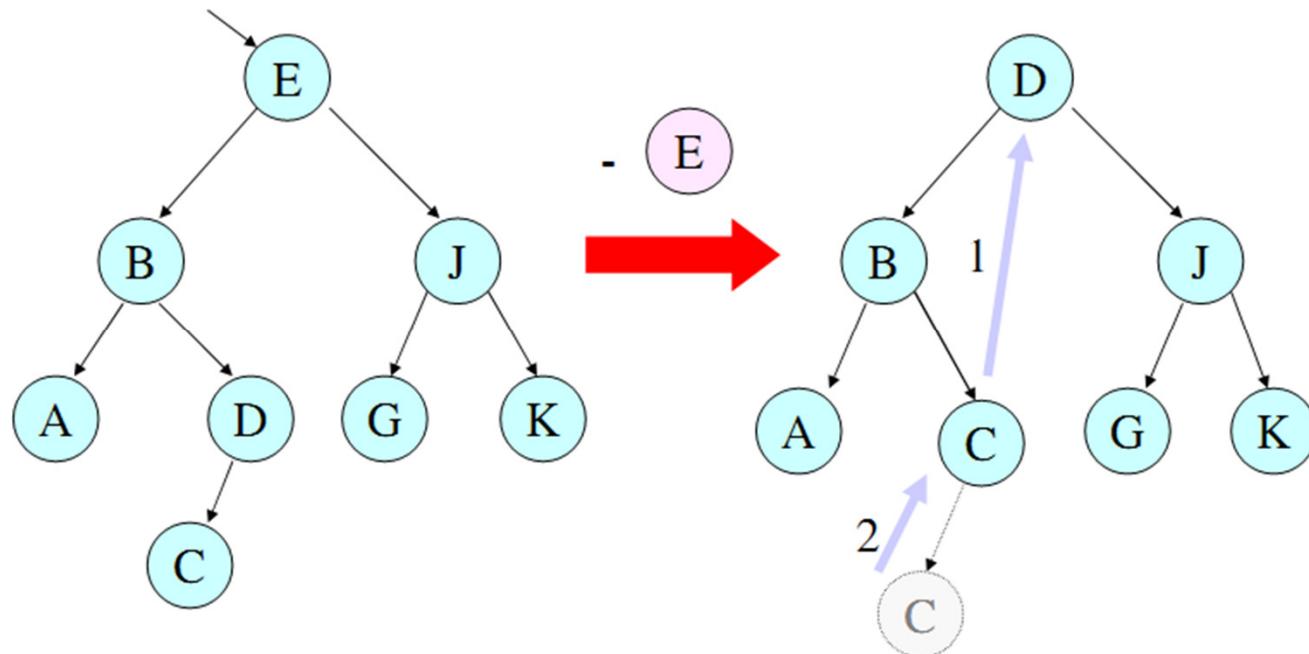
- Caso 2a: nó X a ser removido não é uma folha
  - *esqDir* : nó mais à direita da sub-árvore esquerda
  - X recebe conteúdo de *esqDir*
  - remover (recursivamente) o nó *esqDir*



# Árvore de Pesquisa

## Remoção de um Nó

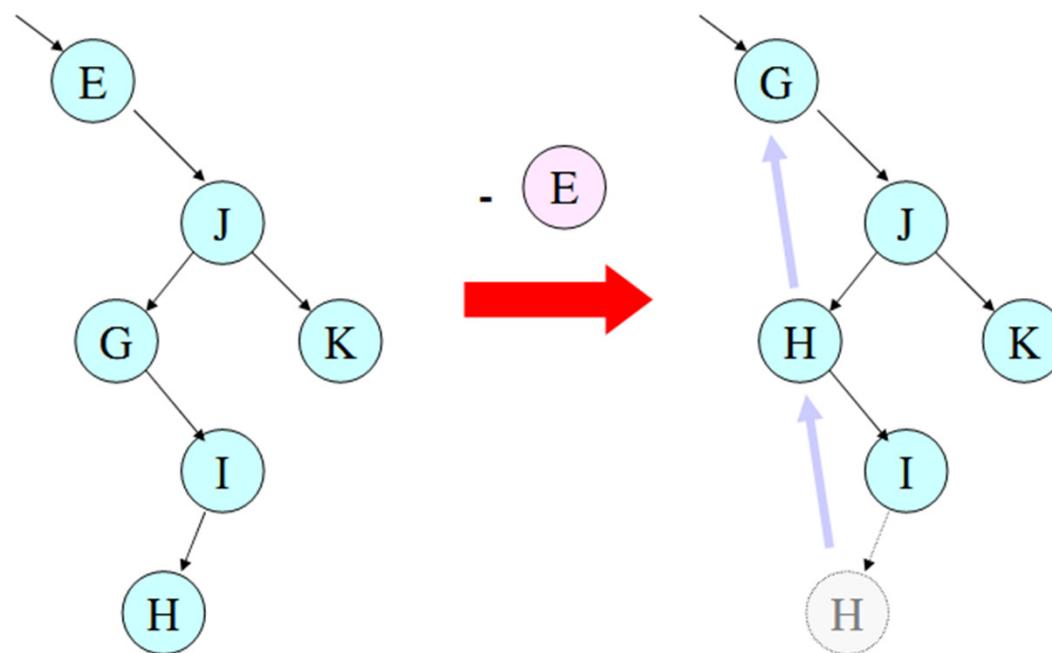
- Caso 2a: nó X a ser removido não é uma folha
  - *esqDir* : nó mais à direita da sub-árvore esquerda
  - X recebe conteúdo de *esqDir*
  - remover (recursivamente) o nó *esqDir*



# Árvore de Pesquisa

## Remoção de um Nó

- Caso 2b: nó X a ser removido não é uma folha
  - *dirEsq* : nó mais à esquerda da sub-árvore direita
  - X recebe conteúdo de *dirEsq*
  - remover (recursivamente) o nó *dirEsq*



# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Exemplo de Implementação em C:
- Operação **Percorre()**: percorre uma Árvore Binária, seguindo o método Pré-Ordem.;
- Método Pré-Ordem:
  - a. Se a árvore é vazia, fim.
  - b. Processar a raiz.
  - c. Atravessar a sub-árvore esquerda em pré-ordem.
  - d. Atravessar a sub-árvore direita em pré-ordem.

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- **Observação:** Existem alguns algoritmos clássicos para percorrer uma árvore. Todas elas se utilizam da técnica recursiva. Afim de estabelecer as regras dos percursos, assumiremos algumas convenções:
  - E : significa que percorreremos o galho esquerdo do nó em questão;
  - P : significa acessar (mostrar, consultar ou alterar) o nó;
  - D : significa que percorreremos o galho direito do nó em questão.

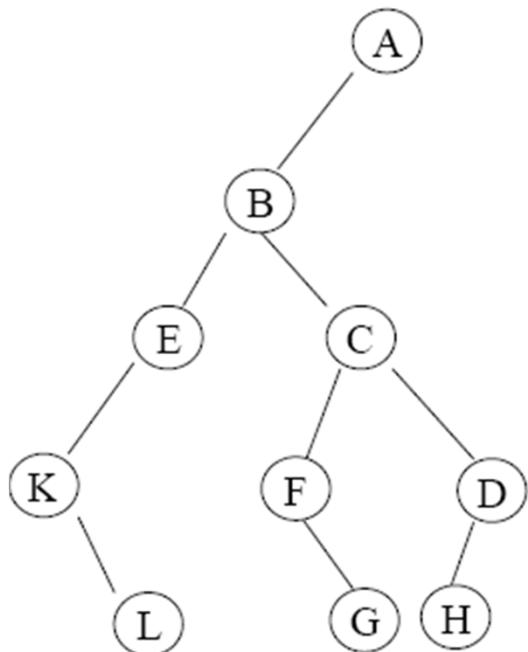
# Árvore de Pesquisa

## Árvores Binárias De Pesquisa

- Destas três convenções temos 6 combinações possíveis, no entanto, descreveremos as possibilidades quando a busca se iniciar pelo galho esquerdo (3 combinações):
  1. **PED** :- pre order
  2. **EPD** :- in order
  3. **EDP** :- pos order

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa



PED :- A, B, E, K, L, C, F, G, D, H

EPD :- K, L, E, B, F, G, C, H, D, A

EDP :- L, K, E, G, F, H, D, C, B, A

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa Com Balanceamento

- Uma Árvore Binária Balanceada (árvore AVL) é uma árvore binária na qual as alturas das duas subárvores de todo nó nunca diferem em mais de 1
- O balanceamento de um nó numa árvore binária é definido como a altura de sua subárvore esquerda menos a altura de sua subárvore direita



# Árvore de Pesquisa

## Árvores Binárias De Pesquisa Com Balanceamento

- Cada nó numa árvore binária balanceada tem um balanceamento de 1, -1 ou 0, dependendo de a altura de sua subárvore esquerda ser maior, menor ou igual á altura de sua subárvore direita;

# Árvore de Pesquisa

## Árvores Binárias De Pesquisa Com Balanceamento

