

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PIAUÍ
Campus Teresina - Central

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E
TECNOLOGIA DO PIAUÍ

CAMPUS TERESINA-CENTRAL
DIRETORIA DE ENSINO

Estrutura de Dados

Professora: Elanne Cristina O. dos Santos

elannecristina.santos@gmail.com
elannecristina.santos@ifpi.edu.br

Linguagem C

- Podemos definir a linguagem C como sendo uma linguagem de programação robusta e multiplataforma, projetada para aplicações modulares de rápido acesso.
- Podendo ser considerada como uma linguagem de médio nível, pois possui instruções que a tornam ora uma linguagem de alto nível e estruturada como o Pascal, se assim se fizer necessário, ora uma linguagem de baixo nível pois possui instruções tão próximas da máquina, que só o Assembler possui.
- De fato com a linguagem C podemos construir programas organizados e concisos (como o Pascal), ocupando pouco espaço de memória com alta velocidade de execução (como o Assembler). Infelizmente, dada toda a flexibilidade da linguagem, também poderemos escrever programas desorganizados e difíceis de serem compreendidos (como usualmente são os programas em BASIC).
- Devemos lembrar que a linguagem C foi desenvolvida a partir da necessidade de se escrever programas que utilizassem recursos próprios da linguagem de máquina de uma forma mais simples e portável que o assembler.

CARACTERÍSTICAS DA LINGUAGEM C

- Portabilidade entre máquinas e sistemas operacionais.
- Dados compostos em forma estruturada.
- Programas Estruturados.
- Total interação com o Sistema Operacional.
- Código compacto e rápido, quando comparado ao código de outras linguagem de complexidade análoga.
- Para compilar e executar nossos programas utilizaremos o ambiente Bloodshed Dev-C++, disponível gratuitamente no *link*:

<http://www.bloodshed.net/devcpp.html>

A ESTRUTURA BÁSICA DE UM PROGRAMA EM C

- Um programa em C consistem em uma ou várias “funções”. Vamos começar pelo menor programa possível em C:

```
main( ) {  
}
```

- Os comentários iniciam com o símbolo `/*` e se estendem até aparecer o símbolo `*/`.
- A diretiva `#include` inclui o conteúdo de um outro arquivo dentro do programa atual, ou seja, a linha que contém a diretiva é substituída pelo conteúdo do arquivo especificado.

Sintaxe:

```
#include <nome do arquivo>
```

A ESTRUTURA BÁSICA DE UM PROGRAMA EM C

- A tabela a seguir apresenta alguns dos principais .h da linguagem C:

Arquivo----Descrição

stdio.h ->Funções de entrada e saída (I/O)

string.h ->Funções de tratamento de strings

math.h ->Funções matemáticas

ctype.h ->Funções de teste e tratamento de caracteres

stdlib.h ->Funções de uso genérico

COMANDOS BÁSICOS - INSTRUÇÕES DE ENTRADA E SAÍDA

A FUNÇÃO PRINTF()

- É um dos mais poderosos recursos da linguagem C, printf() servirá basicamente para a apresentação de dados no monitor.
Sua forma geral será: printf("string de controle", lista de argumentos);

Ex.:

Exemplo: Dado um número, calcule seu quadrado.

```
#include<stdio.h>

main() {
    int numero;
    numero=10;
    printf("O %d elevado ao quadrado resulta em %d. \n",
           numero,numero*numero);
}
```

OBS: A diretiva #include foi utilizada, pois usamos o comando printf (stdio.h)

Usando o “gcc” para compilar no prompt de comando

- Com o gcc já instalado no seu sistema, é muito simples usá-lo para compilar programas em C. Se o programa consistir de um único arquivo, você pode simplesmente executar este comando no terminal:

```
gcc prog.c -o prog
```

- onde “*prog.c*” é o nome do arquivo que contém o código. Os outros dois parâmetros, “-o *prog*”, indicam o arquivo de saída do compilador e o arquivo executável que conterá o programa.

Usando o “gcc” ou “g++” para compilar no prompt de comando

Compilar:

gcc prog.cpp -o prog -lstdc++

ou

g++ prog.cpp –o prog

- Executar :
. \prog

COMANDOS BÁSICOS - INSTRUÇÕES DE ENTRADA E SAÍDA

A FUNÇÃO SCANF()

- Uma das mais importantes e poderosas instruções, servirá basicamente para promover leitura de dados (tipados) via teclado.
- Sua forma geral será: scanf("string de controle", lista de argumentos);
- Algumas leituras básicas:
 - %c - leitura de caracter
 - %d - leitura de números inteiros
 - %f - leitura de números reais
 - %s - leitura de caracteres
- Cada variável a ser lida, deverá ser precedida pelo caracter &, por razões que no momento não convém explicarmos, mas que serão esclarecidas no decorrer do curso. Para seqüência de caracteres (%s), o caracter & não deverá ser usado.

COMANDOS BÁSICOS - INSTRUÇÕES DE ENTRADA E SAÍDA

A FUNÇÃO SCNF()

- Exemplo: Programa para ler e mostrar uma idade

```
/* Exemplo Lê e Mostra Idade */
main()  {
    int idade;
    char nome[30];
    printf("Digite sua Idade: ");
    scanf("%d",&idade);
    printf("Seu Nome: ");
    scanf("%s",nome); /* Strings não utilizar ‘&’ na leitura */
    printf("%s sua idade e' %d anos. \n", nome, idade);
}
```

Namespaces e Strings –

Recursos de entrada e saída da biblioteca iostream

```
#include <iostream>

int valor;

int main()
{
    cout << "Exemplo de saída na tela"
    << endl;
    cout<<"digite o valor :";
    cin>>valor;
}
```

Namespaces e Strings -

Recursos de entrada e saída da biblioteca iostream

```
#include <iostream>
using namespace std;
int mat;
float nota;
char nome[30];
int main()
{ cout<<"Digite a matricula:";
  cin>>mat;
  fflush(stdin);
  cout<<"Digite o nome:" ;
  cin>>nome;
  fflush(stdin);
  cout<<"Digite a nota:" ;
  cin>>nota;
}
```

Namespaces

- Para usar os recursos de entrada e saída da biblioteca *iostream* em C++, é preciso incluir o comando **using namespace std**.
- Este comando serve para definir um "espaço de nomes", ou **namespace**. Um *namespace* permite a definição de estruturas, classes, funções, constantes, etc, que estarão vinculadas a ele. Por definição, a linguagem C++ utiliza o *namespace std* para definir todas as funções da biblioteca padrão.

Namespaces

- Se não utilizarmos o comando **using...**, será necessário especificar explicitamente o *namespace* utilizado, como por exemplo:

```
#include <iostream>

int main()
{
    std::cout << "Exemplo de saída na
tela" << std::endl;

    . . .

}
```

Variáveis

Tabela de Tamanhos e Escala de Tipos Básicos

Tipo	Extensão	Escala Numérica em bits
Char	8	0 a 255
Int	16	-32768 a 32767
Float	32	3.4E-38 a 3.4E+38
Double	64	1.7E-308 a 1.7E+308
Void	0	sem valor

Máscaras

Máscara	Tipo de dado	Descrição
%d	Int	Mostra um número inteiro
%c	Char	Mostra um caractere
%f	Float ou double	Mostra um número decimal
%s	Char	Mostra uma cadeia de caracteres (string)
%i	Int	Mostra um inteiro
%ld	Long int	Mostra um número inteiro longo

Máscaras

As máscaras

Agora que vimos para que servem as máscaras, veremos qual é a máscara de cada tipo de entrada e saída.

máscara	tipo de dado	descrição
%d	int	mostra um número inteiro
%c	char	mostra um caracter
%f	float ou double	mostra um número decimal
%i	int	mostra um número inteiro
%ld	long int	mostra um número inteiro longo
%e	float ou double	mostra um número exponencial (número científico)
%E	float ou double	mostra um número exponencial (número científico)
%o	int	mostra um número inteiro em formato octal
%x	int	mostra um número inteiro em formato hexadecimal
%X	int	mostra um número inteiro em formato hexadecimal
%s	char	mostra uma cadeia de caracteres (string)

Variáveis

- Exemplo : Mesmo número com 2 representações diferentes.

```
main() {  
    float a;  
    printf("Digite um numero: ");  
    scanf("%f",&a);  
    printf("%f %e",a,a);  
}
```

Simulando obtemos:

Digite um numero: 65
65.000000 6.500000E+01

Variáveis

- Exemplo : Criando três variáveis e inicializando-as em tempo de criação.

```
Main () {  
    int evento = 5;  
    char corrida = 'c';  
    float tempo = 27.25;  
    printf (" o melhor tempo da eliminatória % c", corrida);  
    printf ("\n do evento %d foi % f", evento, tempo);  
}
```

Simulando obtemos:

o melhor tempo da eliminatória c
do evento 5 foi 27.25

OPERADORES ARITMÉTICOS

Operador	Ação
+	Adição
*	Multiplicação
/	Divisão
%	Resto de Divisão Inteira
-	Subtração o menos unário
--	Decremento
++	Incremento

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    float nota1;
    float nota2;
    float nota3;
    float media;
    printf("\n Digite a primeira nota..: ");
    scanf("%f",&nota1);
    printf("\n Digite a segunda nota...: ");
    scanf("%f",&nota2);
    printf("\n Digite a terceira nota..: ");
    scanf("%f",&nota3);
    media=(nota1+nota2+nota3)/3;
    printf("\n\n Sua média .....: %.2f",media);
}
```

Exemplo

OPERADORES RELACIONAIS E LÓGICOS

Operador	Ação
>	Maior que
\geq	Maior ou igual que
<	Menor que
\leq	Menor ou igual que
$=$	Igual a
\neq	Diferente de
$\&\&$	Condição “E”
$\ $	Condição “OU”
!	Não

Função

- Conceitualmente, C é baseada em blocos de construção. Assim sendo, um programa em C nada mais é que um **conjunto de funções básicas** ordenadas pelo programador.
- As instruções **printf()** e **scanf()**, vistas anteriormente, não fazem parte do conjunto de palavras padrões da linguagem (instruções), pois não passam elas mesmas de funções escritas em C!
- Cada função C é na verdade uma sub-rotina que contém um ou mais comandos em C e que executa uma ou mais tarefas. Em um programa bem escrito, cada função deve executar uma tarefa.

Funções

- Exemplo 1:

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

void alo() {
    printf("ola!!!!\n\n");
}

main() {
    alo();
    system("pause");
}
```

Funções – Exemplo 2:

(definição do protótipo da função)

```
void alo();  
  
main() {  
    alo();  
  
}  
  
void alo()  
{  
    printf("teste!!");  
  
}
```

Função

PROTOTIPO DE UMA FUNÇÃO

- A declaração de uma função quando feita no inicio de um programa em C é dita *protótipo da função*. Esta declaração deve ser feita sempre antes da função *main*, definindo-se o tipo, o nome e os argumentos desta mesma função. Exemplo:

```
float soma(float, float);  
int sqr(int);
```

- Definindo-se o protótipo de uma função não é necessário escrever o código desta mesma função antes da função *main*, pois o protótipo indica ao compilador C que a função está definida em outro local do código.

Função - Exemplo

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

int sqr(int);

main() {
    int num;
    printf("Digite um numero: ");
    scanf("%d", &num);

    int valor = sqr(num);
    printf("%d ao quadrado e' %d \n", x, valor);

    system("pause");
}

int sqr(int n) {
    int x = n; /* x é um "parâmetro" recebido do programa principal
no caso x "vale" o conteúdo de num */

    return(x*x);
}
```

Atividade

- Faça uma função fibonnaci Não-Recursiva que retorne o n-ésimo termo da sequênci.

$$0+1=1$$

$$1+1=2$$

$$1+2=3$$

$$2+3=5$$

$$3+5=8$$

$$5+8=13$$

$$8+13=21$$

$$13+21=34$$

$$21+34=55$$

$$34+55=89$$

$$55+89=144$$

```
int fibonacci(int termo){  
    int aux,prox=1;  
    int anterior=1;  
    if (termo==0)  
        return 0;  
    if ((termo==1) || (termo==2))  
        return 1;  
    else{  
        for(int i=3;i<=termo;i++){  
            aux=prox;  
            prox=anterior+aux;  
            anterior=aux;  
        }  
        return prox;  
    }  
}
```

Atividade

- Faça uma função fibonacci Não-Recursiva que retorne a seqüência até o n-ésimo termo.

$$0+1=1$$

$$1+1=2$$

$$1+2=3$$

$$2+3=5$$

$$3+5=8$$

$$5+8=13$$

$$8+13=21$$

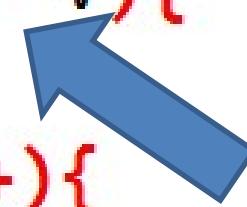
$$13+21=34$$

$$21+34=55$$

$$34+55=89$$

$$55+89=144$$

```
void fibonacci(int n,int* v){  
    int aux=0;  
    for (int i=0;i<=n;i++){  
        if (i==0)  
            v[i]=0;  
        else{  
            if ((i==1) || (i==2)){  
                v[i]=1;  
            }  
            else{  
                v[i]=v[i-1]+v[i-2];  
            }  
        }  
    }  
}
```



- Existem dois métodos de passagem de parâmetros para funções:
- **Passagem por valor** – permite usar dentro de uma função uma cópia do valor de uma variável, porém não permite alterar o valor da variável original (somente a cópia pode ser alterada).
- **Passagem por referência** – É passada para a função uma referência da variável, sendo possível alterar o conteúdo da variável original usando-se esta referência.
- Na linguagem C a passagem por referência é implementada com o uso de **ponteiros**.
- **Os valores passados obrigatoriamente tem que ser ponteiros já que irão receber um endereço de memória.**

Atividade

- Faça uma função fatorial Não-Recursiva que retorne o fatorial de um número.

```
#include <stdio.h>
#include <iostream>
using namespace std;
unsigned long int fatorial(unsigned long int n){
    unsigned long int fat=1;
    if (n==0)
        fat= 1;
    else{
        for(int i=n;i>=1;i--)
            fat = fat*i;
    }
    return fat;
}
```

Usando o tipo `unsigned long int`

- Quando se fala em números, deve-se remeter para o tamanho da palavra que a ULA suporta (*Unidade Lógica e Aritmética*).
- Em processadores de 32 bits, o maior inteiro que o processador consegue lidar é 4.294.967.295 e isto apenas se for considerado números inteiros positivos, sem sinal.
- Com sinal a faixa de representação de um inteiro em 32 bits vai de -2.147.483.648 até +2.147.483.647.
- Um compilador C faz uma variável do tipo *long int* acompanhar o tamanho da ULA. Assim, é possível trabalhar com números inteiros de 32 bits.

Usando o tipo `unsigned long int`

- Em uma arquitetura de 32 bits, apenas até o fatorial de 12 pode ser calculado. Se for tentar calcular o fatorial de 13, este daria 6.227.020.800, número que não pode ser representado em 32 bits.
- Em uma arquitetura de 64 bits, apenas até o fatorial de 22 pode ser calculado, pois fatorial de 23 resulta em 25.852.016.738.884.976.640.000 e este valor não cabe em 64 bits.
- Não resolve trocar a variável para ***double***. As respostas até podem bater até certo ponto, mas depois, devido a perda de precisão, os valores estarão errados.

Função Recursiva

- Uma função denomina-se *recursiva* quando dentro dela se faz uma chamada para ela mesma.
- Um exemplo prático seria o calculo do fatorial de um numero.

```
int main() {
    int n;
    do {
        printf("Digite um numero ou negativo p/ sair \n");
        scanf("%d",&n);
        if (n < 0) {
            break;
        }
        printf("O Fatorial de %d eh %lu \n",n,fatorial(n));
    } while (1);
}

unsigned long int fatorial (unsigned long int n) {
    return ((n==0) ? (long)1 : (unsigned long int)n * fatorial(n-1) );
}
```

Atividade

1. Escreva uma função recursiva para calcular o valor de uma base x elevada a um expoente y.
2. Faça um programa que realize a soma dos N primeiros números inteiros. OBS: USE RECURSÃO.
 - Supondo N = 5;
 - $S(5) = 1+2+3+4+5 = 15 \rightarrow S(5) = S(4) + 5 \rightarrow 10 + 5 = 15$
 - $S(4) = 1+2+3+4 = 10 \rightarrow S(4) = S(3) + 4 \rightarrow 6 + 4 = 10$
 - $S(3) = 1+2+3 = 6 \rightarrow S(3) = S(2) + 3 \rightarrow 3 + 3 = 6$
 - $S(2) = 1+2 = 3 \rightarrow S(2) = S(1) + 2 \rightarrow 1 + 2 = 3$
 - $S(1) = 1 = 1 \rightarrow S(1) = 1$

Atividade

3. Fazer uma função recursiva que conta o número de ocorrências de um determinado caracter, caract(char c, char s[])

- Ex. 1:

Na cadeia "fabio dos santos silva" existem quantas ocorrências do caractere "s"?

O programa deve retornar o valor 3.

- Ex.2:

Na cadeia "maria marlene" existem quantas ocorrências do caractere "m"?

O programa deve retornar o valor 2.

Ponteiros

```
int a;
```

```
/* Esta declaração define uma variável de nome "a"  
que pode armazenar valores inteiros. Automaticamente  
reserva-se um espaço de memória suficiente para  
armazenar valores inteiros. */
```

```
int *p;
```

```
/* Algumas variáveis valores de endereço de memória,  
abaixo p é uma variável que armazena o endereço de  
memória que possui valores inteiros armazenados */
```

<<<O ponteiro nada mais é do que uma variável que
guarda o endereço de uma outra variável. >>>

Ponteiros

DECLARAÇÃO DE PONTEIROS

A declaração de ponteiros é feita da seguinte forma:

*<tipo> *<nome_do_ponteiro>*

EX.:

*int *p;*

*float *a;*

Ponteiros

- A declaração de ponteiro não tem o mesmo significado da declaração de uma variável.
- Ela contém apenas o **valor de um endereço de memória**, o tamanho em bytes que ocupa não tem relação com o tamanho do objeto apontado. O tamanho do ponteiro é fixo e depende apenas do modelo de memória do sistema (2 bytes ou 4 bytes, normalmente).
- Para declarar mais de um ponteiro por linha, usa-se um operador indireto (*):

*char *ch1, *ch2;* (s o ponteiros para o tipo char).

Ponteiros

- Para se inicializar um *ponteiro* é necessário apenas atribuir-se um endereço de memória.
- A simples declaração de um ponteiro não o faz útil. É necessária a indicação da variável para a qual ele aponta.

```
int var;  
int *ptr;  
var = 10;  
ptr = &var; // recebe o endereço de "var"
```

Acrescentando a linha

```
int newVar = *ptr; // recebe o valor guardado no endereço do  
                    ponteiro ptr  
*ptr = 20; // modifica o valor guardado no endereço do ponteiro ptr
```

Qual o valor final das variáveis “newVar” e “var”?

Vetores

- Um Vetor é um conjunto de variáveis de mesmo tipo que compartilham um mesmo nome. Com *Vetor* agora podemos armazenar mais de uma valor para depois serem manipulados através de um *índice*, o qual referencia cada um dos elementos, para se criar um *Vetor* é necessário definir um tipo, seu nome e quantidade de elementos, sendo este ultimo entre colchetes ([]). Vejamos agora a declaração de um vetor.

```
int mat[5];
```

```
#include <stdio.h>

main() {
    int i;
    float nota[3], m=0;

    for (i=0;i<3;i++) {
        printf("\n Digite a nota %d ", i+1 );
        scanf("%f",&nota[i]);
        m = m + nota[i];
    }
    m = m / 3;
    printf("\n A media e' %.2f \n", m );
}
```

Exemplo: Cálculo da média

Definição de tipo - Struct

```
struct aluno {  
    int mat;  
    float nota;  
    char nome[30];  
};  
typedef struct aluno Aluno;
```

```
-----  
typedef struct aluno{  
    int matricula;  
    float nota;  
    char nome[30];  
}Aluno;
```

Exemplo

```
struct aluno {
int mat;
float nota;
char nome[30];
};

typedef struct aluno Aluno;

main() {
    Aluno vet_aluno[20];

    int i = 0;

    while (i < 20) {
        printf("Digite a mat:");
        scanf("%d",&vet_aluno[i].mat);
        printf("Digite a nota:");
        scanf("%f",&vet_aluno[i].nota);
        printf("Digite o nome:");
        scanf("%s",vet_aluno[i].nome);
        i = i + 1;

    }
}
```

Exemplo 2 – usando gets ou fgets na leitura de string

```
int i = 0;
while (i<=2) {
    printf("Digite matricula:");
    scanf("%d",&vet[i].mat);
    printf("Digite a nota:");
    scanf("%f",&vet[i].nota);
    fflush(stdin);
    printf("Digite o nome:");
    //fgets(vet[i].nome,30,stdin);
    gets(vet[i].nome);
    i++;
}
```

OBS: para usar os recursos de entrada e saída da biblioteca ***iostream*** em C++, é preciso incluir o comando ***using namespace std;***

Exemplo 3 – usando ***cout*** e ***cin***

```
for(int i=0;i<3;i++){
    cout<<"Digite a matricula:";
    fflush(stdin);
    cin>>v[i].mat;
    cout<<"Digite a nome:";
    fflush(stdin);
    cin>>v[i].nome;
    cout<<"Digite a nota:";
    fflush(stdin);
    cin>>v[i].nota;
    cout<"\n\n";
}
```

Classes

```
#include <string.h>
#include <iostream>

using namespace std;

class Aluno{
    private:
        int mat;
        string nome;
    public:
        Aluno(int m, string n){
            mat = m;
            nome= n;
        }
};
```

Objetos

```
main(){
    //Está chamando um construtor da classe
    // que recebe os parâmetros especificados.
    Aluno a1(1,"carlos");
    //criar uma nova alocação de memória e
    //o endereço desta alocação será atribuído
    //obviamente para um ponteiro e não diretamente o tipo.
    Aluno *a2= new Aluno(2,"Jose");
}
```

Atributos e métodos

```
class Aluno{
    private:
        int mat;
        string nome;
    public:
        Aluno(int m, string n){
            mat = m;
            nome = n;
        }
        string getName();
    };
    string Aluno::getName(){
        return nome;
    }
main(){
    Aluno a3(1, "Joao");
    cout << "Mat: " << a1.getName();
    cout << "Nome: " << a1.getName();
}
```

Mais exemplos...

```
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;

class Carro {
public:
    string modelo;
    string marca;

    Carro(){}
    Carro(char x[23], string y) {
        modelo=x;
        marca = y;
    }
};
```

Mais exemplos...

```
main(){
    Carro c1("uno","fiat");
    cout<<c1.modelo<<endl;
    cout<<c1.marca<<endl;
    Carro *c2= new Carro("palio","fiat");
    cout<<c2->modelo<<endl;
    cout<<c2->marca;
}
```

Mais exemplos...

```
class Carro{
    private:
        string modelo;
        string marca;
    public:
        Carro(string m1, string m2){
            modelo = m1;
            marca = m2;
        }

        string getModelo();
};

string Carro::getModelo(){
    return modelo;
}
```

Atividade 2

- 1.Considere os dados matricula, nome e idade do aluno para a estrutura.
- Criar um método para inserir (máximo 10 alunos)
- 2. Criar um método que mostre os alunos cadastrados
- 3. Criar um método que retorne a quantidade de elementos cadastrados.
- 4. Criar um método que retorne o aluno mais velho.
- 5. Criar um menu no método main:
Opções : 1 – Cadastrar alunos / 2 – Mostrar alunos cadastrados / 3 – Mais velho / 4 - Sair

Atividade 2- Continuação

- No programa anterior, mostre o nome do aluno e a matricula do aluno que tirou a **maior nota**

(OBS: construir um método ***maior_nota***).

- No programa anterior, mostre o nome do aluno e a matricula do aluno que tirou a menor nota

(OBS: construir um método ***menor_nota***).

- Adicione a chamada dos 2 métodos no menu do programa.

Atividade 3

- Crie uma estrutura que represente os dados de um livro:
- Nome do livro, nome do autor, numero de paginas.
- Máximo de 10 livros.
- Imprima as informações do livro com maior número de paginas.
- **OBS:**
- **definir os métodos *incluir, mostrar, maior***
- ***O método maior deve retornar uma estrutura do tipo livro***

Atividade 4

- Crie uma estrutura que represente os dados de um cliente:
- Nome do cliente, numero do cartao, saldo do cartao, limite do cartao
- MÁximo de 5 clientes.
- Informe as informações do cliente com limite de cartao mais alto.
- Imprima as informações do cliente com menor saldo.
- OBS: definir os métodos *inserir, mostrar, maiorLimite, menorSaldo*
- *O método maiorLimite deve retornar uma estrutura do tipo cliente*
- *O método menorSaldo deve retornar uma estrutura do tipo cliente*

Listas Lineares

- **TIPOS DE LISTAS LINEARES:**
 1. Lista estática desordenada
 2. Lista estática ordenada
 3. Lista dinâmica desordenada
 4. Lista dinâmica ordenada

LISTAS

É uma das estruturas de dados mais empregadas no desenvolvimento de programas.

Ao desenvolver uma implementação para listas lineares, o primeiro problema que surge é: como podemos armazenar os elementos da lista.

VETORES

ALOCAÇÃO ESTÁTICA

Ocorre quando a quantidade total de memória utilizada pelos dados é previamente conhecida e definida de modo imutável.

PONTEIROS

ALOCAÇÃO DINÂMICA

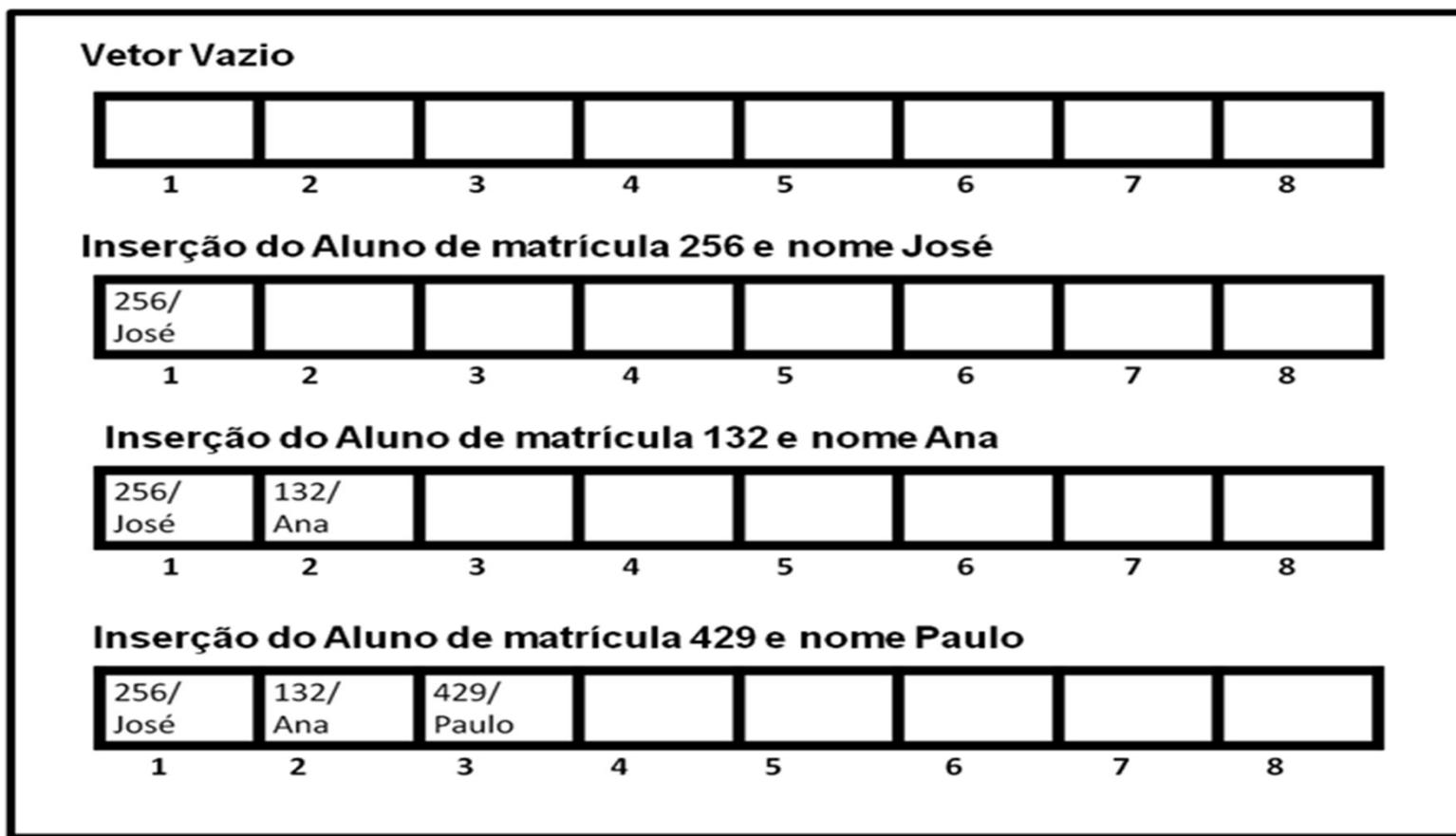
É o processo de solicitar e utilizar memória durante a execução de um programa. Ou seja, o programa aloca (SOLICITAR) espaço de memória quando há necessidade e deslocar quando houver necessidade de remoção de alguma informação.

Listas Estáticas Desordenadas

- Na lista desordenada os elementos são colocados na primeira posição vazia da lista (normalmente, no final). Na lista ordenada, é escolhido um dado que será o campo de ordenação da lista. Quando se deseja inserir um novo elemento na lista, primeiro tem que ser verificado em que local ele dever ser colocado para que seja mantida a ordem da lista.
- Operações básicas das listas: inserir elemento, remover elemento, consultar elemento, alterar elemento, listagem dos elementos da lista.

Lista Estática Desordenada

- Inserir um elemento



```
//Inserir novo aluno
void inserir() {
    int qa=0; int cont;
printf("\nInserir Novo Aluno\n\n");
do {
    if (qa < maximo) { // verifica se o vetor pode receber
novo                         // aluno
        printf("\nMatricula do Aluno: ");
        scanf("%d",&turma[qa].mat);
        printf("\nNome: ");
        scanf("%s",turma[qa].nome);
        qa++;
        printf("\n\nAluno Inserido com Sucesso!!!\n\n");
    }
....
```

```
else {      // vetor cheio
    printf("\n\n\naNao Pode Inserir – Turma Cheia!!!\n\n");
    break;
}
printf("\n\nInserir outro(1-sim/2-nao)? ");
scanf("%d",&cont);
}while (cont == 1);
}
```

Lista Desordenada Estática

- Consultar um elemento

Vetor							
256/ José	132/ Ana	429/ Paulo	578/ Maria	127/ João	314/ Sara		
1	2	3	4	5	6	7	8

Atividade 5

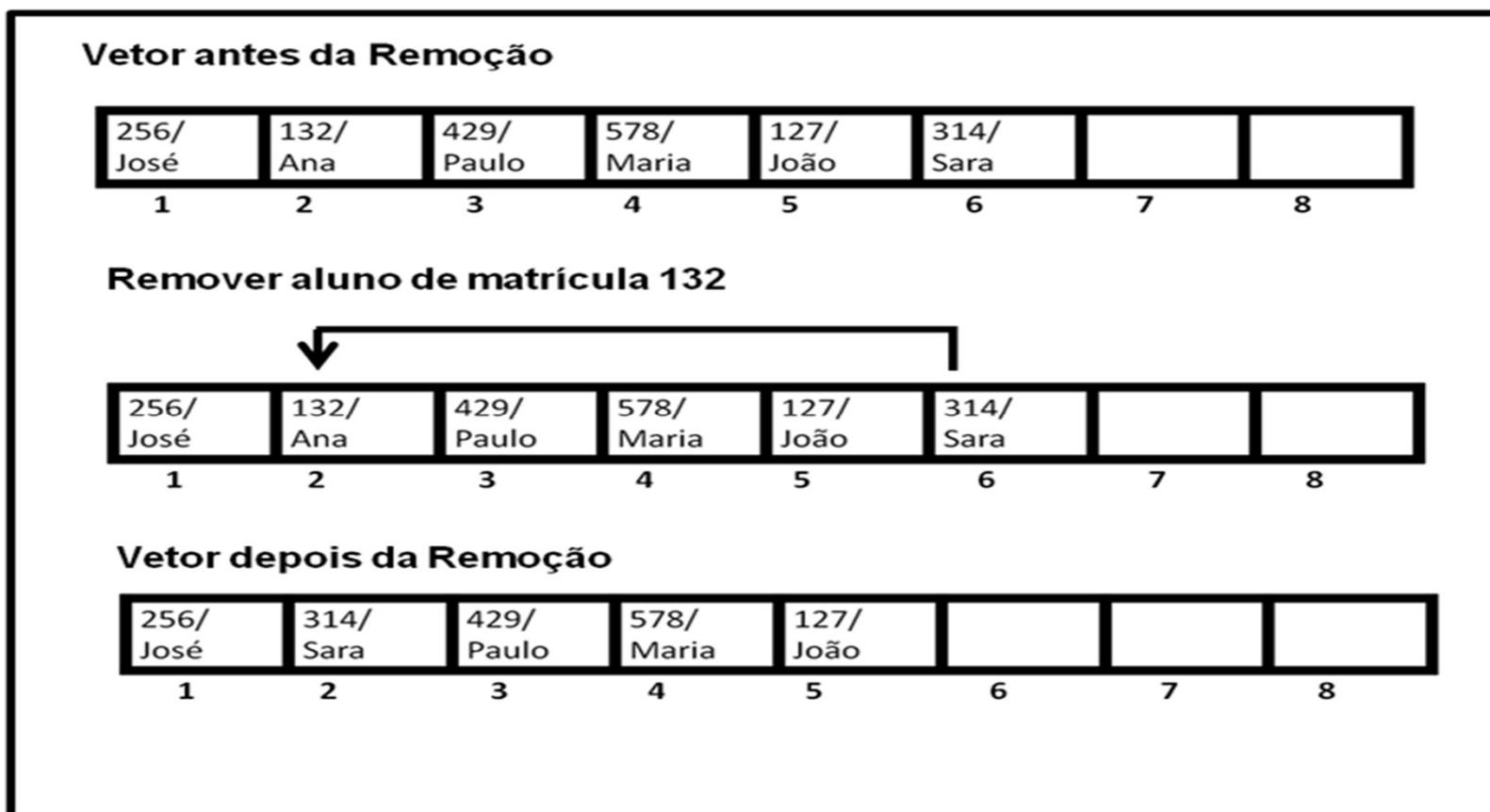
- Fazer a consulta de um elemento específico dentro do vetor:
 - Função **int procura(int mat):**
 - Recebe um valor de matricula e retorna a posição do elemento dentro do vetor
 - Função **void mostrar(int pos):**

Recebe a posição do elemento e imprime o elemento na tela do computador.
 - **Função void consultaMat():**
 - Ler o valor da matricula procurada e chamar a função “procura” e depois chamar a função “mostrar”

```
void consultarmat() {  
    int i, matcon, achou, cont;  
    do {  
        printf("\nConsultar Aluno por Matricula\n\n");  
        printf("\nMatricula do Aluno: ");  
        scanf("%d",&matcon);  
        achou = procura(matcon);  
        if (achou != -1)  
            mostre(achou);  
        else // aluno nao foi encontrado  
            printf("\n\nO numero de Matricula e incorreto!!!!\n");  
        printf("\n\nConsultar outro(1-sim/2-nao)? ");  
        scanf("%d",&cont);  
    } while (cont == 1);  
}
```

Lista Desordenada Estática

- Remover um elemento



```
void remover() {  
    int matrem, i, cont, achou, conrem;  
    do{  
        printf("\nRemover Aluno\n\n"); printf("\nMatricula do Aluno: ");  
        scanf("%d",&matrem);  
        achou = procura(matrem);  
        if (achou != -1) {  
            mostre(achou);  
            printf("\nDeseja remover o aluno (1-sim/2-nao)? ");  
            scanf("%d",&conrem);  
            if (conrem==1) { // verifica se quer remover  
                turma[achou]= turma[qa-1];  
                qa--;  
                printf("\n\nAluno removido com Sucesso!!!\n");  
            } .....
```

```
else
    printf("\n\n\ao aluno nao foi removido!!!\n");
break;
}

else // aluno nao foi encontrado
    printf("\n\nnaNumero de Matricula Incorreto!!!!!!\n");
printf("\n\nRemover outro(1-sim/2-nao)? ");
scanf("%d",&cont);
}while (cont == 1);

}
```

LISTA ESTÁTICA ORDENADA

- Para inserir um elemento em uma lista ordenada podem ocorrer cinco possibilidades:
 1. a lista está cheia:nesse caso a inserção é cancelada;
 2. a lista está vazia: o elemento é colocado na primeira posição do vetor;
 3. o elemento a ser inserido é menor do que o primeiro da lista;
 4. o elemento a ser inserido é maior do que o ultimo da lista;
 5. o elemento novo será inserido entre elementos da lista.

LISTA ESTÁTICA ORDENADA

Vetor Vazio

1	2	3	4	5	6	7	8

Inserção do Aluno de matrícula 256 e nome José

256/ José							
1	2	3	4	5	6	7	8

Inserção do Aluno de matrícula 132 e nome Ana



256/ José							
1	2	3	4	5	6	7	8

132/ Ana	256/ José						
1	2	3	4	5	6	7	8

Inserção do Aluno de matrícula 429 e nome Paulo

132/ Ana	256/ José	429/ Paulo					
1	2	3	4	5	6	7	8

Inserção do Aluno de matrícula 197 e nome Maria



132/ Ana	256/ José	429/ Paulo					
1	2	3	4	5	6	7	8

132/ Ana	197/ Maria	256/ José	429/ Paulo				
1	2	3	4	5	6	7	8

```
void colocarordem() {  
    int i, local;  
    local = -1;  
    if (qa==0)  
        turma[0] = al;  
    else {  
        for (i=0;i<qa;i++) {  
            if (al.mat<turma[i].mat) {  
                local = i;  
                break;  
            }      }  
        if (local==-1)  
            turma[qa] = al;  
        else {  
            for (i=qa;i>local;i--)  
                turma[i]=turma[i-1];  
            turma[local]=al;      }      }  }
```

Tipo Vector

- O tipo **vector** é similar a um array, com a diferença que possui tamanho dinâmico. As principais funções definidas são:
- **begin()**: Retorna um iterator para o início do vetor
- **end()**: Retorna um iterator para o final do vetor
- **push_back()** :Adiciona elemento no final do vetor
- **pop_back()**: Destroi elemento no final do vetor
- **Size()** :Número de elementos do vetor
- **[]** : Operador de acesso randômico
- **Insert**: Insere um elemento em uma posição intermediária
- **at**: Acessa elemento

Exemplo

```
// vector::at
#include <iostream>
#include <vector>
int main ()
{
    std::vector<int> myvector (10);
    // Inicializa 10 inteiros com zero

    // assign some values:
    for (unsigned i=0; i<myvector.size(); i++)
        myvector.at(i)=i;
    std::cout << "myvector contem:";
    for (unsigned i=0; i<myvector.size(); i++)
        std::cout << ' ' << myvector.at(i);
    std::cout << '\n';
    return 0;
}
```

```
#include <stdio.h>
#include <stdio.h>
#include <vector>
using namespace std;
int main(void)
{
    int val;
vector<int> v;
vector<int>::iterator iter;
    printf("Digite valores: ");
    do{
        scanf("%d", &val);
v.push_back(val);
    }while(val!=-1);
    return 0;
}
```

EXEMPLO

Para percorrer o vetor:

Iterators

- Para percorrer todos os elementos da estrutura são usados **Iterators**. Eles podem ser vistos como ponteiros para posições específicas da estrutura.
- Um iterator aponta para um elemento dentro de um conjunto de elementos.
- No caso de vector, o *iterator* é um *ponteiro*. O ponteiro aponta para os elementos no *array*, e é possível interagir com os elementos usando os operadores `++` ou `--`.

```
#include <stdio.h>
#include <stdio.h>
#include <vector>
int main(void)
{
    int val;
    std::vector<int> v;  std::vector<int>::iterator iter;
    printf("Digite alguns inteiros: ");
    do{  scanf("%d", &val);  v.push_back(val);
    } while(val!=-1);
for( iter=v.begin(); iter != v.end(); iter++ )
    printf("%d ", *iter);
//acesso com o operador randomico
for( int i=0; i<v.size(); i++)
    printf("%d ", v[i] );
return 0;
}
```

Acessando os elementos

ATIVIDADE

- Considere as seguintes estruturas:

```
class TAluno
{
    private:
        int matricula;
        string nome;
        int idade;
    public:
        TAluno(int m,string n,int i);
        string getNome();
        int getMat();
        int getIdade();
};
```

```
class TLista{  
    private:  
        int quant;  
        vector<TAAluno> l;  
    public:  
        TLista();  
        void inserir(int,string,int);  
        void inserirOrdenado(int,string,int);  
        void mostrar();  
        void mostrar2();  
        int procuraMat(int);  
        void remover1(int);  
        void remover2(int);
```

ATIVIDADE

- Defina os métodos da classe TLista:

void inserir(int,string,int);

void inserirOrdenado(int,string,int);

void mostrar();

void mostrar2();

int procuraMat(int);

void remover1(int);

void remover2(int);

Consultar

- Na figura abaixo, a matrícula do aluno a ser consultado deve ser lida. É feita uma varredura em todas as posições ocupadas do vetor, a procura da matrícula.
- No caso de uma lista ordenada, se estivéssemos procurando um aluno de matrícula 120, assim que fizéssemos a leitura da primeira matrícula do vetor, já saberíamos que a matrícula 120 não está no vetor, evitando uma varredura até o final do vetor.
- Na estrutura ordenada, as consultas são mais eficientes.

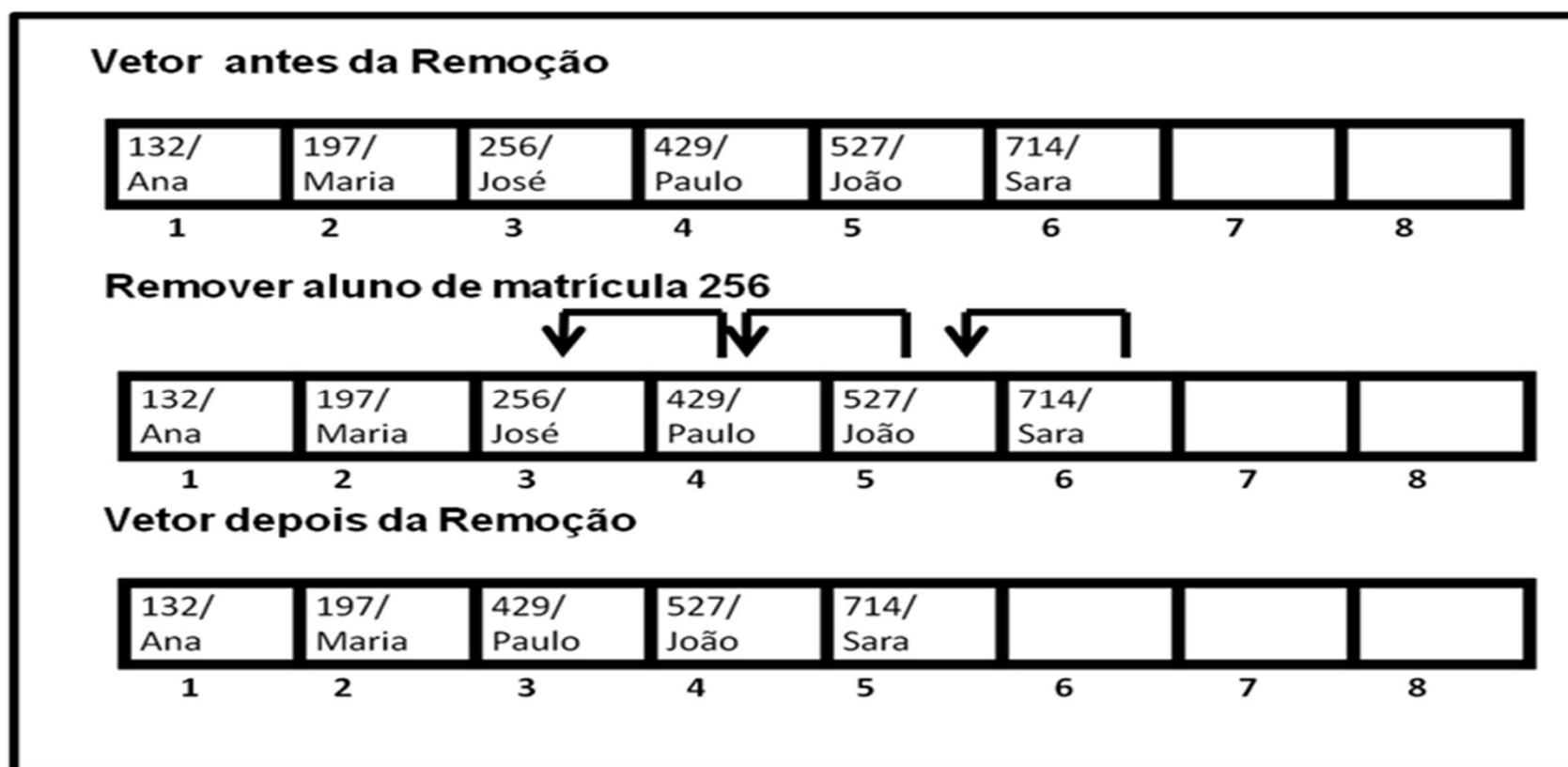
Vetor							
1	2	3	4	5	6	7	8
132/ Ana	197/ Maria	256/ José	429/ Paulo	527/ João	714/ Sara		

Remover

- Começamos com a leitura da matrícula do aluno a ser removido.
- É feita uma varredura em todas as posições ocupadas do vetor, a procura da matrícula. Assim que o elemento é encontrado, seus dados devem ser apresentados ao usuário (neste caso a matrícula e o nome). Dessa forma ele pode verificar se realmente é aquele o elemento a ser removido. Quando o elemento não é encontrado, uma mensagem de erro deve ser dada ao usuário.

Remover

- No exemplo da *figura* a seguir, desejamos remover o elemento de matrícula 256. Para que a lista fique contínua e ordenada, todos os elementos que vem depois do elemento que será removido, devem ser trazidos uma posição a frente.



STL

- A STL – Parte da biblioteca padrão do C++, a Standard Template Library é um conjunto de tipos abstratos de dados, e funções projetados para **manipularem diferentes tipos de dados de forma transparente**.
- A STL faz uso de templates, já definidos, para implementação de diversos algoritmos que manipulam dados de forma eficiente, como por exemplo **containers (vetores, listas,pilhas, etc)**.
- No STL também foi adicionado um tipo *String*, que facilita as operações de manipulação de caracteres quando comparado a biblioteca string.h da linguagem C.

STL e namespace

- Ao se usar STL, deve-se usar o conceito de *namespace*, que permite agrupar classes, objetos e funções e associá-los a um nome, o que facilita a quebra de escopos globais em subescopos menores. Ex:

```
namespace teste
{
    int a, b; }

int main()
{
    teste::a = 8;
    printf("%d", teste::a);
    return 0; }
```

Namespace std

- Em STL, todas as classes, objetos e funções são definidos com o **namespace std**. Pode-se **usar o operador de escopo a cada uso da STL (std::vector, por exemplo)** ou **indicar o namespace corrente**, da seguinte forma:

using namespace std;

Contêiner

- Um **contêiner** é um objeto de suporte que armazena uma coleção de outros objetos (seus elementos). Eles são implementados como modelos de classe, o que permite uma grande flexibilidade aos elementos.
- Os contêineres replicam estruturas muito usadas na programação: arrays dinâmicas (**vector**), filas (**queue**), pilhas (**stack**), listas (**list**), árvores (**sets**)...

Contêiners

- Muitos *contêiners* têm várias funções em comum e compartilham funcionalidades.
- A decisão de qual tipo de *contêiner* usar para uma necessidade específica geralmente não depende apenas da funcionalidade oferecida pelo recipiente, mas também da eficiência de alguns de seus membros (**complexidade**).
- Isto é especialmente verdadeiro para *contêiners de seqüência*, que oferecem diferenças na complexidade entre inserir / remover elementos e acessá-los. São eles: *array*, *vector*, *deque*, *forward_list* e *list*.

Contêiners de seqüência

- **Vectors: Podem mudar de tamanho**
 - **Seqüência:** Elementos são ordenados em uma seqüência linear estrita. Elementos individuais são acessados por sua posição nesta seqüência.
 - **Array dinâmico:** Permite acesso direto a qualquer elemento da seqüência, através de ponteiros, e fornece adição / remoção relativamente rápida de elementos no final da seqüência.
 - **Alocador:** O contêiner usa um objeto alocador para lidar **dinamicamente** com suas necessidades de armazenamento.

Vector

Principais elementos de acesso:

- []
- At
- **Front**: acessa o primeiro elemento
- **Back**: acessa o último elemento

Contêiners de seqüência

- *Array : tamanho fixo*
 - **Seqüência:** Elementos ordenados em uma seqüência linear estrita. Elementos individuais são acessados por sua posição nesta seqüência.
 - **Armazenagem contígua:** Os elementos são armazenados em locais de memória contíguos, permitindo tempo constante de acesso aleatório aos elementos.
 - **De tamanho fixo:** O contêiner usa construtores implícitos e destrutores para alocar o espaço necessário estaticamente. Seu tamanho é constante de compilação.

Exemplo

```
// array::begin example
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> myarray = { 2, 16, 77, 34, 50 };

    std::cout << "myarray contains:";
    for ( auto it = myarray.begin(); it != myarray.end(); ++it )
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Exemplo

```
// array::at
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myarray;

    // assign some values:
    for (int i=0; i<10; i++) myarray.at(i) = i+1;

    // print content:
    std::cout << "myarray contains:";
    for (int i=0; i<10; i++)
        std::cout << ' ' << myarray.at(i);
    std::cout << '\n';

    return 0;
}
```

Array

- ERRO AO TENTAR USAR ARRAY:
 - **#error** This file requires compiler and library support for the ISO C++ 2011 standard. This support is currently experimental, and must be enabled with the -std=c++11 or -std=gnu++11 compiler options.
 - Por predefinição, suporte para a versão mais recente do C ++ não está ativado. Deve ser explicitamente habilitado indo para: *Ferramentas/Opções do compilador/Configurações/Geração de Código/Padrão da linguagem (-std): GNU C++ 11*

Array

- Algumas principais funções definidas são:
- **begin()**: Retorna um iterator para o início do vetor
- **end()**: Retorna um iterator para o final do vetor

Principais elementos de acesso:

- **[]**: Acessa elemento
- **at**: Acessa elemento
- **front**: Acessa primeiro elemento
- **back**: Acessa o último elemento

Exemplo

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    std::array<int,10> myarray;
    for (int i=0; i<10; i++) myarray.at(i) = i+1;

    std::cout << myarray.begin() << '\n';
    std::cout << myarray.front() << '\n';
    std::cout << myarray.at(3) << '\n';
    std::cout << myarray.at(2) << '\n';
    return 0;
}
```

Contêiners de seqüência

Deque:

- *fila dupla ou fila de extremidade dupla,*
- funcionalidade semelhante aos **vectores**, mas com inserção e eliminação eficaz de elementos **também no início da sequência**, e não apenas no seu fim.
 - *ao contrário dos vetores, os deques não garantem armazenar todos os seus elementos em locais de armazenamento contíguos.*
 - *muito similar a um vetor, com a característica de apresentar rápida inserção e remoção de elementos no início e no final.*

Deque

- *Possui funções de acesso iguais ao Vector, e duas adicionais:*
 - **push_front()**:Adiciona elemento no início da Deque.
 - **pop_front()** :Destroi elemento no início da Deque.

Deque

Principais elementos de acesso:

- **[]**: Acessa elemento
- **at**: Acessa elemento
- **front**: Acessa primeiro elemento
- **back**: Acessa o último elemento

Exemplo PUSH_BACK

```
std::deque<int> mydeque;
int myint;
std::cout << "Please enter some integers (enter 0 to end):\n";
do {
    std::cin >> myint;
    mydeque.push_back (myint);
} while (myint);
std::cout<<"\nDeque:";
for (int& x: mydeque) std::cout << " " << x;
std::cout << std::endl;
std::cout << "mydeque stores " << (int) mydeque.size() << " numbers.\n";
```

Exemplo AT

```
std::deque<unsigned> mydeque (10); // 10 zero-initialized unsigneds  
  
// assign some values:  
for (unsigned i=0; i<mydeque.size(); i++)  
    mydeque.at(i)=i;  
  
std::cout << "mydeque contains:";  
for (unsigned i=0; i<mydeque.size(); i++)  
    std::cout << ' ' << mydeque.at(i);
```

Contêiners de seqüência

- ***Forward_list*** :permitem operações de inserção e apagamento de tempo constante em qualquer lugar dentro da seqüência.
- São implementadas como listas ligadas individualmente;
- As listas ligadas individualmente podem armazenar cada um dos elementos que contêm em locais de armazenamento diferentes e não relacionados.
- A ordenação é mantida pela associação a cada elemento de um **link para o próximo elemento** na seqüência.

Contêiners de seqüência

- Em comparação com outros contêineres de seqüência (array, vetor e deque), *forward_list* realiza geralmente melhor a inserção, extração e movimentação de elementos em qualquer posição dentro do conteiner. Realiza algoritmos que fazem uso intensivo dessas operações, como algoritmos de classificação (*sort*).

Forward_list

Principais elementos de acesso:

- **front:** Acessa primeiro elemento

Exemplo

```
int main ()
{
    std::forward_list<int> mylist = { 34, 77, 16, 2 };

    std::cout << "mylist contains:";
    for ( auto it = mylist.begin(); it != mylist.end(); ++it )
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

Exemplo PUSH_FRONT

```
#include <iostream>
#include <forward_list>
using namespace std;

int main ()
{
    forward_list<int> mylist = {77, 2, 16};
    mylist.push_front (19);
    mylist.push_front (34);
    std::cout << "mylist contains:";
    for (int& x: mylist) std::cout << ' ' << x;
    std::cout << '\n';
    return 0;
}
```

Exemplo POP_FRONT

```
std::forward_list<int> mylist = {10, 20, 30, 40};

std::cout << "Popping out the elements in mylist:";
std::cout<<"\nAntes";
for(int& x:mylist) std::cout<< " "<<x;
while (!mylist.empty())
{
    std::cout << ' ' << mylist.front();
    mylist.pop_front();
}
std::cout<<"\nDepois";
for(int& x:mylist) std::cout<< " "<<x;
std::cout << '\n';
```

Exemplo INSERT_AFTER

```
std::forward_list<int>::iterator it;

it = mylist.insert_after ( mylist.before_begin(), 10 );
it = mylist.insert_after ( it, 2, 20 );
it = mylist.insert_after ( it, myarray.begin(), myarray.end () );
it = mylist.begin();
it = mylist.insert_after ( it, {1,2,3} );
// std::cout << "mylist contains:";
for (int& x: mylist) std::cout << ' ' << x;
std::cout << '\n';
```

Exemplo SORT

```
int main ()
{
    std::forward_list<int> mylist = {22, 13, 5, 40, 90, 62, 31};

    mylist.sort();

    std::cout << "default sort (operator<):";
    for (std::forward_list<int>::iterator i=mylist.begin();i!=mylist.end();i++){
        std::cout<<" "<<*i;
    }
    return 0;
}
```

Exemplo SORT

```
int main ()
{
    std::forward_list<int> mylist = {22, 13, 5, 40, 90, 62, 31};
    mylist.sort();
    std::cout<<"\nCREScente ";
    for (int& x: mylist) std::cout << ' ' << x;
    std::cout << '\n';
    std::cout<<"\nDECREScente";
    mylist2.sort(std::greater<int>());
    for (int& x: mylist) std::cout << ' ' << x;
    | std::cout << '\n';
    return 0;
}
```

Exemplo UNIQUE

```
std::forward_list<double> mylist = { 15.2, 73.0, 3.14, 15.85, 69.5,  
                                     73.0, 3.99, 15.2, 69.2, 18.5 };  
  
mylist.sort();                                // 3.14, 3.99, 15.2, 15.2, 15.85  
                                              // 18.5, 69.2, 69.5, 73.0, 73.0  
  
mylist.unique();                            // 3.14, 3.99, 15.2, 15.85  
                                         // 18.5, 69.2, 69.5, 73.0  
  
std::cout<<"\n Retira numeros iguais";  
for (double& x: mylist) std::cout << ' ' << x;  
std::cout << '\n';
```

Exemplo UNIQUE

```
bool same_integral_part (double first, double second)
{ return ( int(first)==int(second) ); }

std::forward_list<double> mylist = { 15.2, 73.0, 3.14, 15.85, 69.5,
                                     73.0, 3.99, 15.2, 69.2, 18.5 };

mylist.sort();                                // 3.14, 3.99, 15.2, 15.2, 15.85
                                             // 18.5, 69.2, 69.5, 73.0, 73.0

mylist.unique();                            // 3.14, 3.99, 15.2, 15.85
                                         // 18.5, 69.2, 69.5, 73.0
mylist.unique (same_integral_part); // 3.14, 15.2, 18.5, 69.2, 73.0
std::cout<<"\n mesma parte inteira";
for (double& x: mylist) std::cout << ' ' << x;
std::cout << '\n';
```

Exemplo MERGE

```
std::forward_list<double> first = {4.2, 2.9, 3.1};  
std::forward_list<double> second = {1.4, 7.7, 3.1};  
std::forward_list<double> third = {6.2, 3.7, 7.1};  
first.sort();  
second.sort();  
first.merge(second);  
std::cout << "\nfirst contains:";  
for (double& x: first) std::cout << " " << x;  
std::cout << std::endl;  
first.sort (std::greater<double>());  
std::cout << "\nfirst contains greater:";  
for (double& x: first) std::cout << " " << x;  
std::cout << std::endl;
```

Contêiners de seqüência

list:

- Permitem operações de inserção e apagamento de tempo constante em qualquer lugar dentro da seqüência e iteração em **ambas as direções**.
- Implementados como **listas duplamente vinculadas**.
- As listas duplamente vinculadas podem armazenar cada um dos elementos que contêm em locais de **armazenamento diferentes e não relacionados**. A ordenação é mantida internamente pela **associação a cada elemento de um link para o elemento que o precede e um link para o elemento que o segue**.

Contêiners de seqüência

list:

- Em comparação com outros conteiners de seqüência padrão de base (array, vector e deque), as **listas funcionam geralmente melhor na inserção, extração e movimentação de elementos em qualquer posição dentro do container.**

List

Principais elementos de acesso:

- **front**: Acessa primeiro elemento
- **back**: Acessa o último elemento

Exemplo

```
int myints[] = {75,23,65,42,13};  
std::list<int> mylist (myints,myints+5);  
  
std::cout << "mylist contains - crescente:";  
mylist.sort();  
for (std::list<int>::iterator it=mylist.begin(); it != mylist.end(); ++it)  
    std::cout << ' ' << *it;  
mylist.sort(std::greater<int>());  
std::cout << "\nmylist contains - decrescente:";  
for (int& x:mylist) std::cout<<" "<<x;  
std::cout << '\n';
```

Comparativo – Tempo de execução de diferentes operações

Operação	Vector	Deque	Lista
Acessar o 1º elemento	Constante	Constante	Constante
Acessar último elemento	Constante	Constante	Constante
Acessar elementos randômicos	Constante	Constante	Linear
Add/Delete no início	Linear	Constante	Constante
Add/Delete no final	Constante	Constante	Constante
Add/Delete randomicamente	Linear	Linear	Constante

Complexidade Assintótica dos Algoritmos

- Os limites assintóticos são usados para **estimar a eficiência dos algoritmos**, avaliando-se a quantidade de memória e de tempo necessário para realizar a tarefa para o qual os algoritmos foram projetados.
- **Complexidade de tempo:** Mede o número de atribuições e comparações realizadas durante a execução de um programa.

Complexidade Assintótica dos Algoritmos

Exemplo 01: Laço simples para calcular a soma dos números em um vetor:

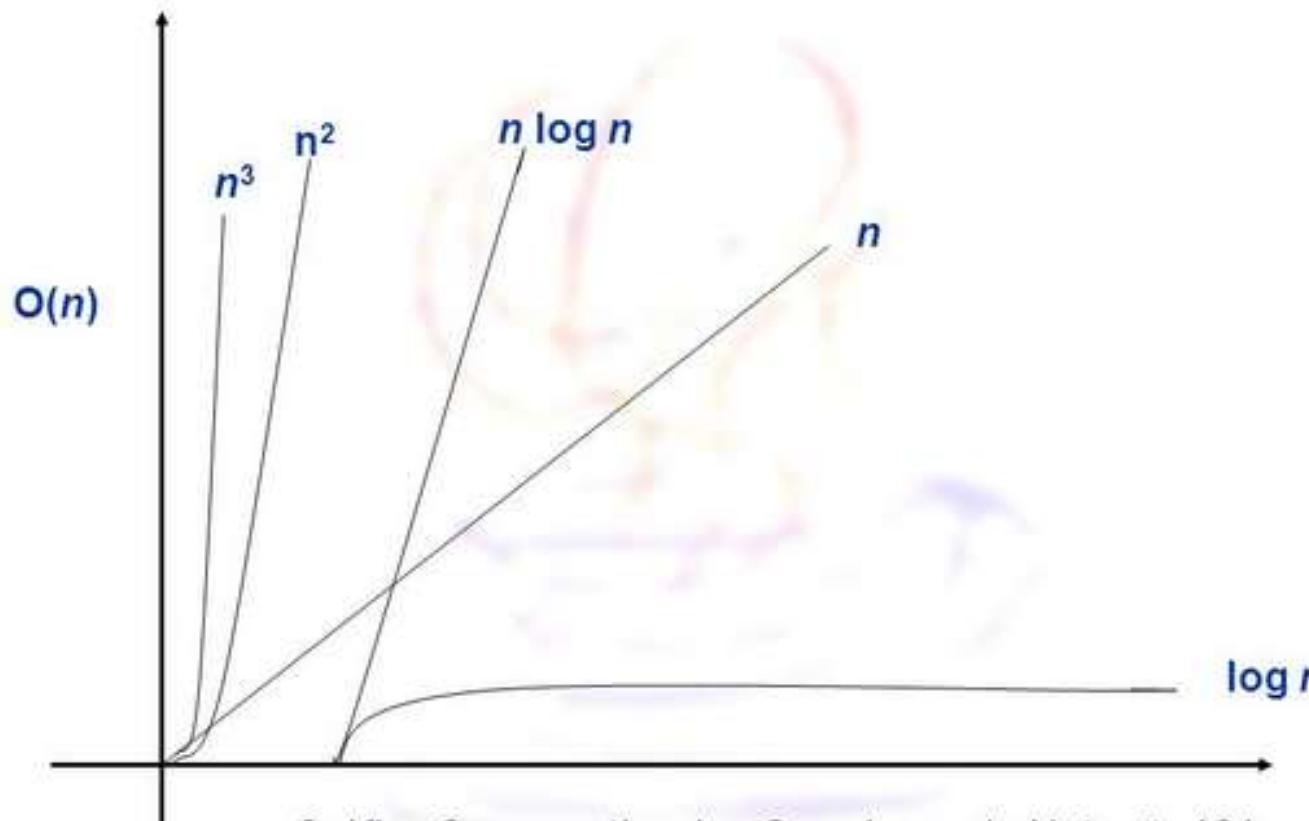
```
soma = 0;  
for(i=0; i<n; i++)  
    soma+=a[i];
```

Primeiro, 2 variáveis são inicializadas, depois o laço itera N vezes. Cada iteração executa 2 atribuições , uma atualiza “soma” e a outra atualiza o “i”. Logo existem **$2 + 2 \times N$ atribuições**.

Sua Complexidade Assintótica é de **$O(n)$** .

COMPARATIVO

Análise de Algoritmos Notação Assintótica (9)



Cálculo Numérico

Gráfico Comparativo das Grandezas da Notação 'O'.

Aula 14 - 26/26

©Prof. Lineu Mialaret

[SCC-501-Aula 7a - Análise de algoritmos - parte 2.pdf](#)

Ordenação pelo Método da Bolha Bubblesort

- Algoritmo:
 - Percorra o vetor inteiro comparando elementos adjacentes (dois a dois)
 - Troque as posições dos elementos se eles estiverem fora de ordem
 - Repita os dois passos acima com os primeiros $n-1$ itens, depois com os primeiros $n-2$ itens, até que reste apenas o um item

Exemplo – Bolha Bubblesort

I = 4

4	3	1	7	2
---	---	---	---	---

j

3	4	1	7	2
---	---	---	---	---

j

3	1	4	7	2
---	---	---	---	---

j

3	1	4	7	2
---	---	---	---	---

j

3	1	4	2	7
---	---	---	---	---

j

I = 3

3	1	4	2	7
---	---	---	---	---

j

1	3	4	2	7
---	---	---	---	---

j

1	3	4	2	7
---	---	---	---	---

j

1	3	2	4	7
---	---	---	---	---

j

I = 2

1	3	2	4	7
---	---	---	---	---

j

1	3	2	4	7
---	---	---	---	---

j

1	2	3	4	7
---	---	---	---	---

j

•

I = 1

1	2	3	4	7	
---	---	---	---	---	--

j

1	2	3	4	7	
---	---	---	---	---	--

j

```
void bolha(int quant, int* v){  
    int i, j;  
    for(i=quant-1; i>=1; i--){  
        for (j=0; j<i; j++){  
            if (v[j] > v[j+1]){  
                int temp = v[j];  
                v[j] = v[j + 1];  
                v[j+1] = temp;  
            }  
        }  
    }  
}
```

?

- Na situação:



- Quantas comparações vão acontecer?
- Todas são necessárias neste caso?

?

- Número de operações não se altera se vetor já está (parcialmente) ordenado
 - Como melhorar?

```
void bolha(int quant, int* v){  
    int i, j;  
  
    int troca;  
    for(i=quant-1; i>=1; i--){  
        troca = 0;  
        for (j=0; j<i; j++){  
            if (v[j] > v[j+1]){  
                int temp = v[j];  
                v[j] = v[j + 1];  
                v[j+1] = temp;  
                troca = 1;  
            }  
        }  
        if (troca==0){  
            break;  
        }  
    }  
}
```

Método da Bolha Melhorado:
Termina execução quando
nenhuma troca é realizada
após uma passada pelo vetor

BUBBLE SORT: CRITÉRIOS DE ANÁLISE

- Sendo n o número registros no arquivo, as medidas de complexidade relevantes são:
Número de comparações $C(n)$ entre chaves.
Número de movimentações $M(n)$ de registros do arquivo.

- Análise informal com base na operação mais custosa, que é a comparação
 - n etapas no pior caso
 - Na primeira etapa, são feitas $n-1$ comparações e trocas, na segunda $n-2$ e assim por diante.
 - Temos então, a soma dos termos de uma progressão aritmética:
 - $(n-1) + (n-2) + \dots + 1$
 - $Total = \frac{n \cdot (n-1)}{2} \Rightarrow O(n^2)$

- RESUMINDO...

Bubble Sort: o tempo gasto na execução do algoritmo varia em ordem quadrática em relação ao número de elementos a serem ordenados.

- $T = O(n^2)$ – Notação “Big O”
- Atividades mais custosas:
 - Comparações
 - Trocas de Posição (swap)

- Melhor caso: vetor ordenado
- Pior caso: vetor invertido
- Método muito simples, mas custo alto:
 - Adequado apenas se arquivo pequeno
 - Ruim se registros muito grandes

ORDENAÇÃO POR INSERÇÃO

- Ordena um vetor da esquerda para a direita, por ordem crescente ou decrescente.
- À medida que o vetor vai sendo percorrido ele deixa seus elementos à esquerda ordenados.

ORDENAÇÃO POR INSERÇÃO

- Percorre um vetor, sempre a partir do primeiro índice desordenado (inicialmente o 2º elemento), e em seguida procura inseri-lo na posição correta, comparando-o com o seu(s) anterior(s) e trocando-os de lugar enquanto ele for menor que seu(s) precedente(s).

ORDENAÇÃO POR INSERÇÃO

temp = 5



J=-1

**V[j+1]=
temp**



temp = 6



J=0

**V[j+1]=
temp**

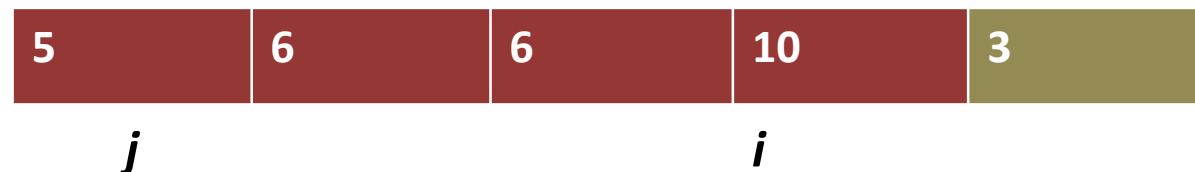
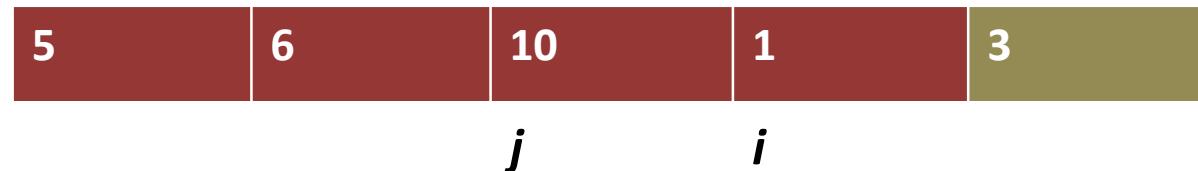


temp = 1

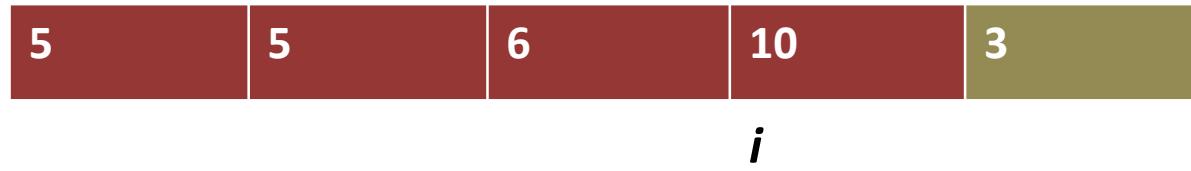


ORDENAÇÃO POR INSERÇÃO

temp = 1



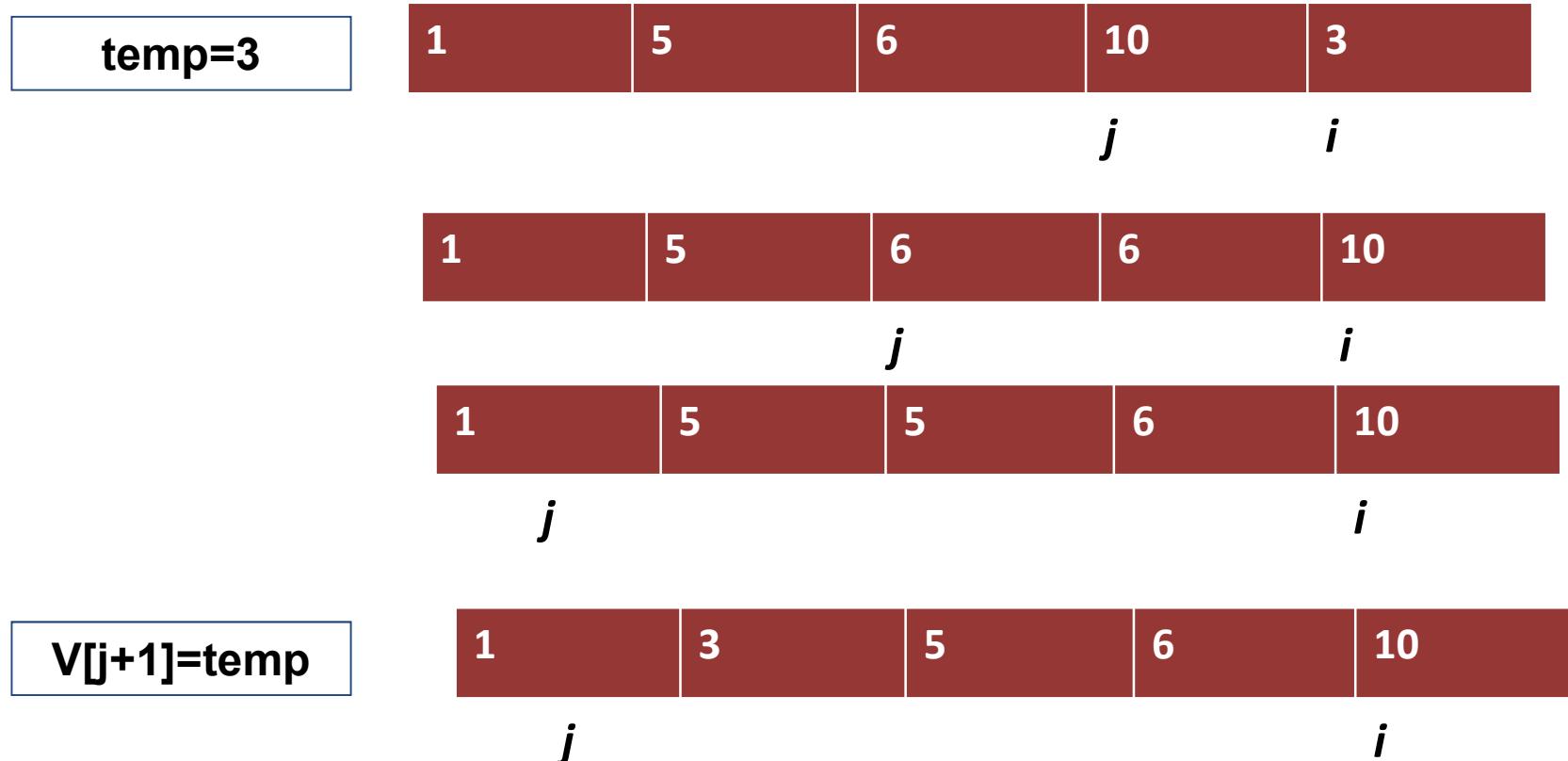
J=-1



V[j+1]=temp



ORDENAÇÃO POR INSERÇÃO



ORDENAÇÃO POR INSERÇÃO

```
int myarray []={10,5,6,1,3};  
void ord_insercao(int v[],int tam){  
    int j, i, temp;  
    *→for (i=1;i<tam;i++)  
    {  
        //guarda o elemento que esta verificando  
  
        temp = v[i];  
        //verificando os elementos anteriores a posicao i  
        j=i-1;  
        **→while (v[j]>temp && j>=0)  
        {  
            v[j+1]=v[j];  
            j--;  
        }  
        //insere o elemento na posicao correta (ordenada) ate i.  
  
        v[j+1] = temp;  
    }  
}
```

ORDENAÇÃO POR INSERÇÃO

- Melhor caso (itens já ordenados):

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n)$$

- Pior caso (itens em ordem reversa):

- $(n-1) + (n-2) + \dots + 1$
- $Total = \frac{n \cdot (n-1)}{2} \Rightarrow O(n^2)$

Pesquisa Binária

- entrada: vetor *vet* com n elementos, ordenado elemento *elem*
- saída: n se o elemento *elem* ocorre em *vet*[n]
-1 se o elemento não se encontra no vetor
- procedimento:
 - compare *elem* com o elemento do meio de *vet*
 - se *elem* for menor, pesquise na primeira metade do vetor
 - se *elem* for maior, pesquise na segunda parte do vetor
 - se for igual, retorne a posição
 - continue o procedimento, subdividindo a parte de interesse, até encontrar o elemento ou chegar a uma parte do vetor com tamanho 0

```
int busca_bin (int n, int* vet, int elem)
{
    /* no inicio consideramos todo o vetor */
    int ini = 0;
    int fim = n-1;

    /* enquanto a parte restante for maior que zero */
    while (ini <= fim) {
        int meio = (ini + fim) / 2;
        if (elem < vet[meio])
            fim = meio - 1; /* ajusta posição final */
        else if (elem > vet[meio])
            ini = meio + 1; /* ajusta posição inicial */
        else
            return meio;      /* elemento encontrado */
    }

    /* não encontrou: restou parte de tamanho zero */
    return -1;
}
```

- **COMPLEXIDADE PESQUISA BINÁRIA**
- Se a chave está no meio da matriz: o laço executa somente uma vez
- Quantas vezes o laço executa no caso em que a chave não está na matriz?
 - **1º:** busca na matriz inteira de tamanho n
 - **2º:** $n/2$
 - **3º:** $n/2^2$
 - **4º.:** $n/2^3$
 - **Última vez:** $n/2^m$
 - , e assim por diante, até que a matriz seja de tamanho 1.

- $n/2^m = 1 \rightarrow$
- $2^m = n \rightarrow$
- *OBS: Usando a regra logarítmica:*

$$a^{\log_a b} = b$$

- Logo:
- $2^m = n \rightarrow 2^m = 2^{\log n}$
- Logo $m = \log n$

QuickSort

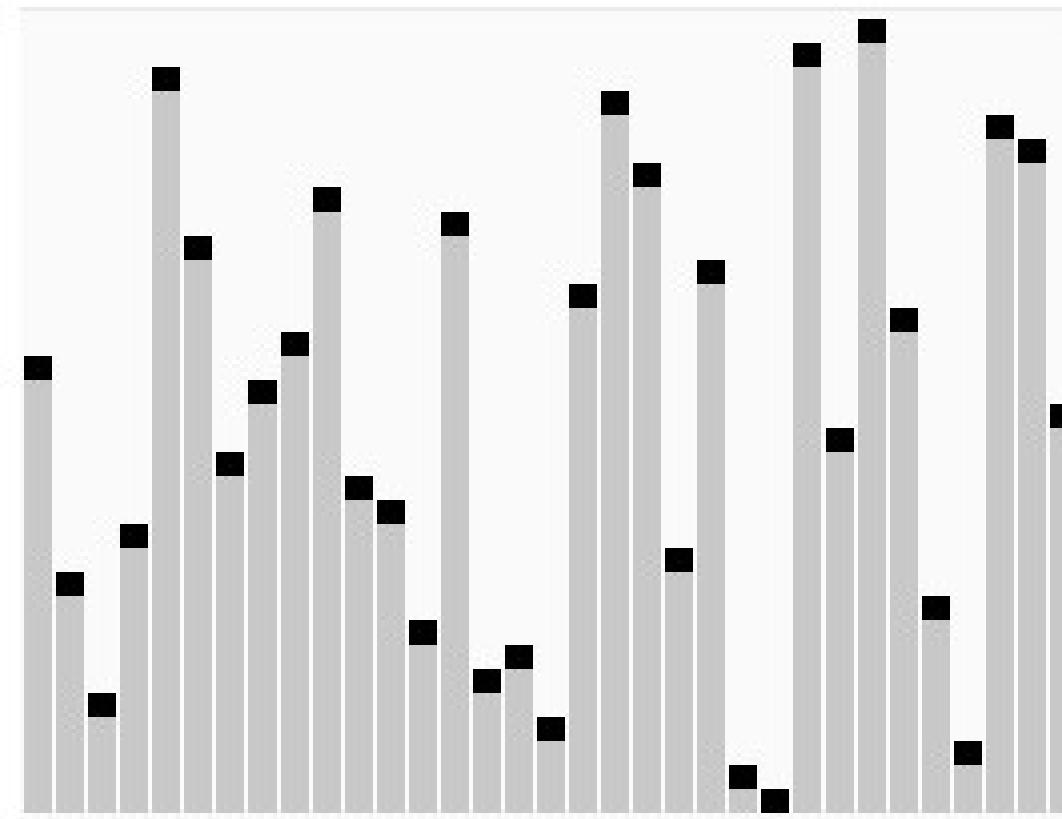
- O algoritmo Quick Sort é um método de ordenação muito rápido e eficiente;
- Idéia: semelhante a um dicionário, ordena-se as palavras, tendo como objetivo reduzir o problema original em subproblemas para assim poder ser resolvido mais fácil e rapidamente.

Quicksort

- Este método divide a tabela em duas sub-tabelas, a partir de um elemento chamado **pivô**.
- **Uma das sub-tabelas contém os elementos menores que o pivô enquanto a outra contém os maiores. O pivô é colocado entre ambas, ficando na posição correta.**

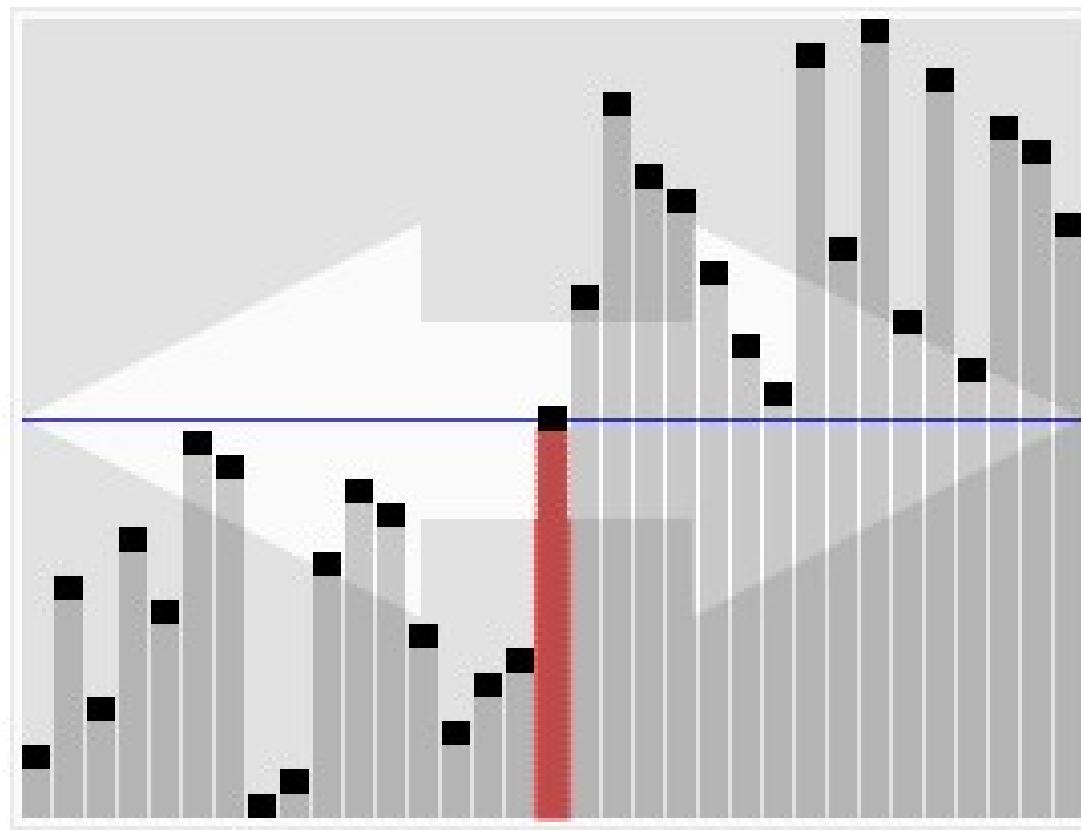
QuickSort

- EXEMPLO: Essas barras de tamanhos diferentes devem ser alinhadas em ordem crescente:



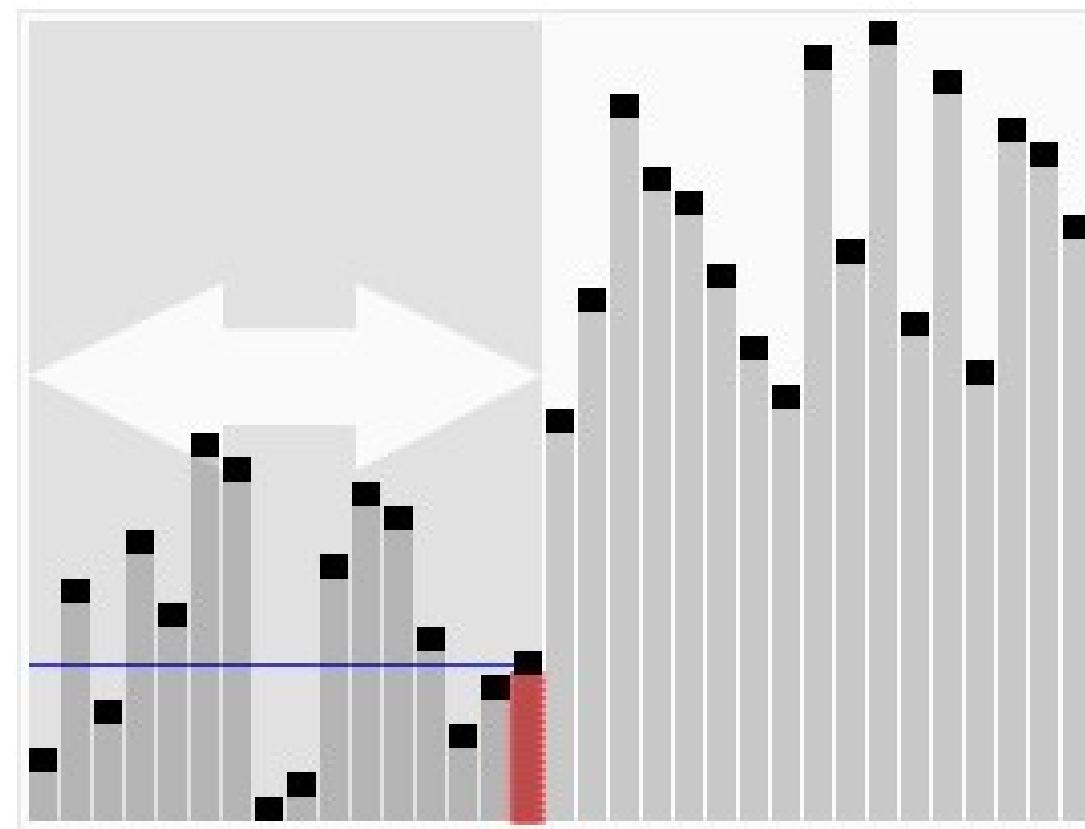
QuickSort

- Os elementos são organizados de maneira que os menores ficam do lado esquerdo do **pivô** e os maiores do lado direito (são as duas sub-tabelas).



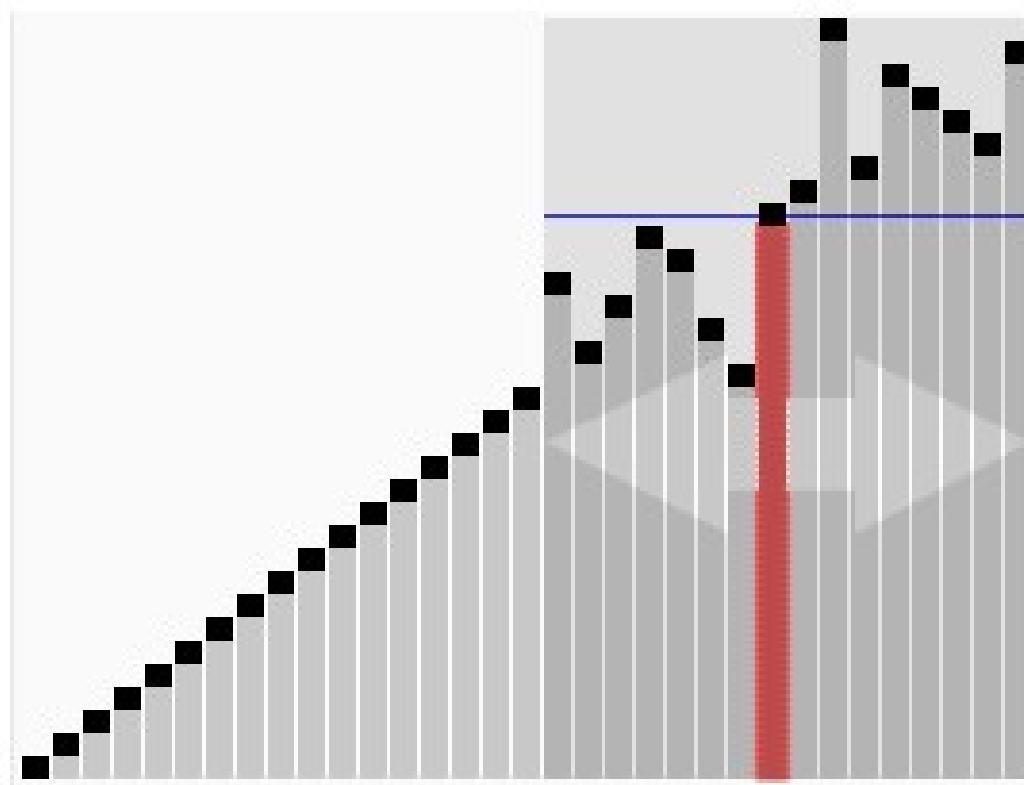
QuickSort

- Depois de encontrar a posição do **pivô** e separar em duas tabelas, ele passa para uma das sub-tabelas e escolhe outro **pivô**.



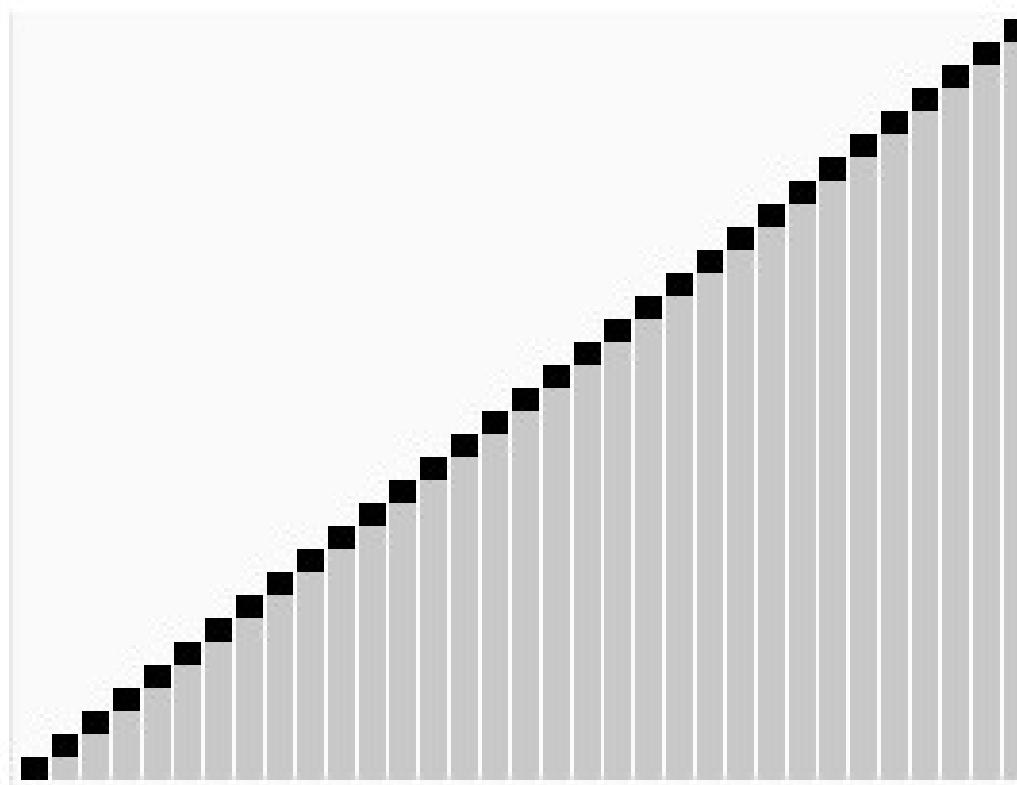
QuickSort

- O pivô do início e todo o lado esquerdo já está ordenado, Agora ele passa para a outra sub-tabela, o lado direito do primeiro **pivô**, escolhe outro **pivô** e ordena.



QuickSort

- E depois de ordenado ele fica assim:



QuickSort

```
#include<stdio.h>
#include<iostream>
using namespace std;
void Quick(int vetor[10], int inicio, int fim);
int main(){
    int vetor[6] = {7, 9, 4, 3, 6, 1};
    int i;
    Quick(vetor, 0, 5);
    printf("\n2. Vetor ordenado:\n");
    for(i = 0; i <= 5; i++){
        printf("%d ", vetor[i]);
    }
    printf("\n");
```

```
void Quick(int vetor[10], int inicio, int fim){
    int pivo, aux, i, j, meio;
    i = inicio;
    j = fim;
    meio = (int) ((i + j) / 2);
    pivo = vetor[meio];
    do{
        while (vetor[i] < pivo) i = i + 1;
        while (vetor[j] > pivo) j = j - 1;
        if(i <= j){
            aux = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = aux;
            i = i + 1;
            j = j - 1;
        }
    }while(j > i);
    if(inicio < j) {
        Quick(vetor, inicio, j);
    }
    if(i < fim) {
        Quick(vetor, i, fim);
    }
}
```

Quicksort

7	9	4	3	6	1
i			j		

1	9	4	3	6	7
	i		j		

1	3	4	9	6	7
		i,j			

Quicksort

1	3	4	9	6	7
i		j	i		

1	3	4	9	6	7
	i,j		i		

1	3	4	9	6	7
j		i	i		

1	3	4	9	6	7
	i	i			j

1	3	4	9	6	7
	i		i		j

QuickSort

1	3	4	9	6	7
	i		i		j

1	3	4	7	6	9
	i			i,j	

1	3	4	7	6	9
	i		j		i

1	3	4	7	6	9
	i	i		j	

1	3	4	7	6	9
	i		i	j	

1	3	4	6	7	9
	i		j	i	

QuickSort

1	3	4	6	7	9
	i			i	j

1	3	4	6	7	9
	i			i,j	

1	3	4	6	7	9
			j		i

QuickSort

- É extremamente eficiente para ordenar arquivos de dados. O método necessita apenas de uma pequena pilha como memória auxiliar e requer $n \log n$ operações, em média, para ordenar n itens.
- $C(n) = 2C(n/2) + n - 1$, onde $C(n/2)$ representa o custo de ordenar uma das metades e $n - 1$ é o número de comparações realizadas.

$$C(n) \approx n \log n$$

Atividade 6

1) Sequencia de Fibonacci:

(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...)

- A) Faça o algoritmo usando recursão.
- B) Faça o algoritmo sem recursão. Neste caso armazene os resultados da série em um vetor de inteiros.
- Verifique a complexidade.

Atividade 6

2) Projeto de Programação: Escreva uma função \lg que receba um inteiro estritamente positivo n e devolva o piso de $\log n$:

n	15	16	31	32	63	64	127	128	255	256	511	512
$\lg(n)$	3	4	4	5	5	6	6	7	7	8	8	9

- Verifique a complexidade

Antes de ver lista dinâmica, vamos relembrar o uso de Ponteiros..

```
int i=15,j,*p,*q;
```

```
p = &i;
```

```
*p=20;
```

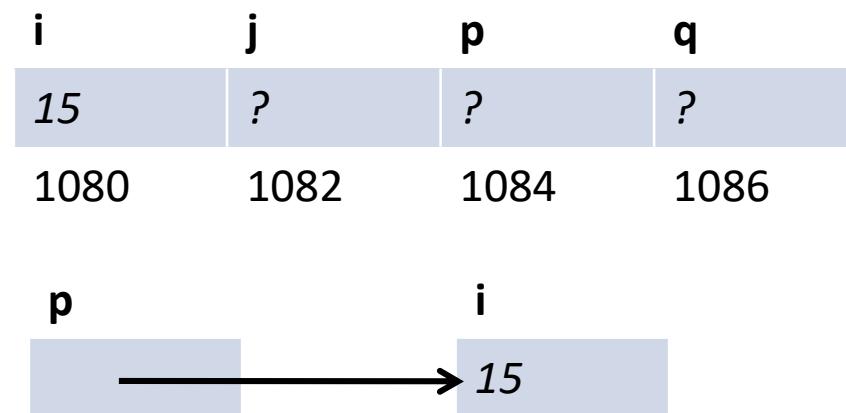
```
j=2**p;
```

```
q=&i;
```

```
*p=*q-1;
```

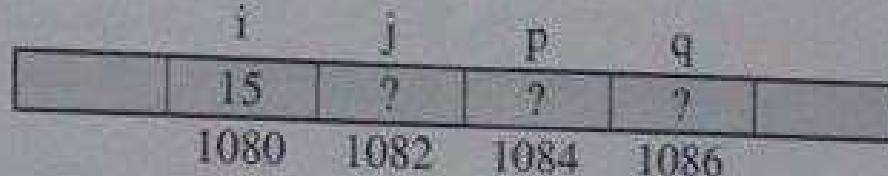
```
q=&j;
```

```
*p=*q-1;
```



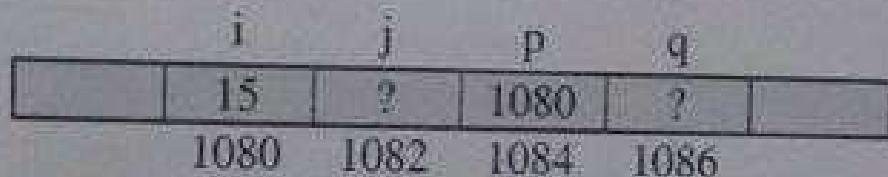
Como ficam os valores no final
da execução do trecho de
programa?

```
int i = 15, j,  
    *p, *q;
```



(a)

```
p = &i;
```

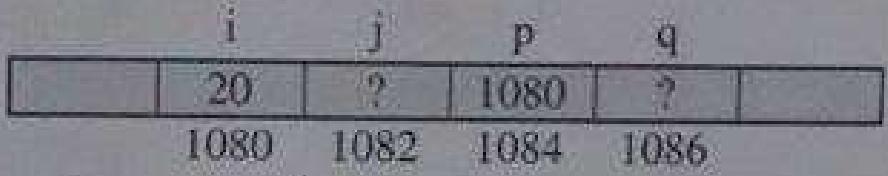


(b)



(c)

```
*p = 20;
```



(d)



(e)

```
j = 2 * *p;
```



(f)

```
q = &i;
```

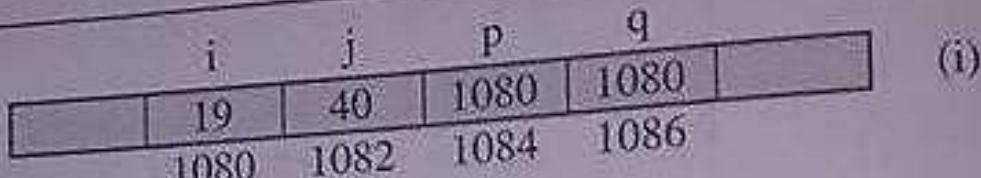


(g)

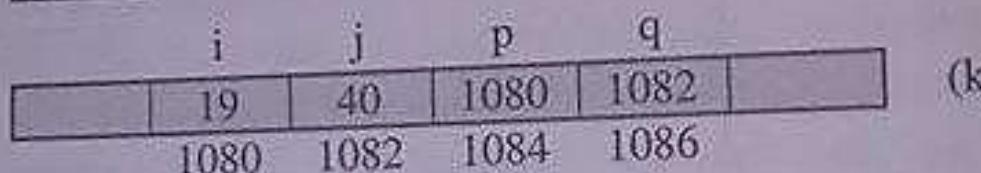


(h)

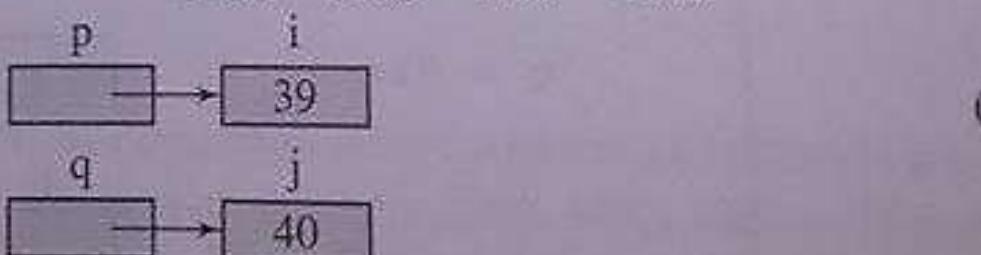
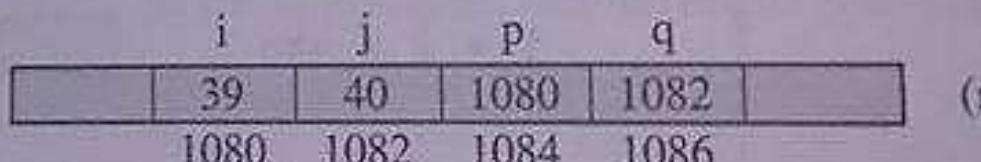
$*p = *q - 1;$



$q = &j;$



$*p = *q - 1;$



LISTA DINÂMICA – A estrutura de cada elemento da lista

- EXEMPLO:

```
typedef struct elemento{  
    int mat;  
    char nome[20];  
    elemento *prox;  
}Elemento;
```

Alocação dinâmica para cada elemento

- Depois de declarar o elemento, é necessário alocar espaço de memória para só depois poder utilizá-lo.

```
main(){  
    Elemento *novo;  
    //Aloca memoria para o novo elemento  
    novo = (Elemento *)malloc(sizeof(Elemento));  
    novo->mat=1;  
    strcpy(novo->nome,"Joao");  
    novo->prox=NULL;  
}
```

novo

1	João	Null
---	------	------

Para inserir outro elemento, como encadeá-los em uma lista?

- Guarde a posição do primeiro elemento da lista:

Elemento *inicio; inicio= NULL;

- Guarde a posição do último elemento da lista:

Elemento *fim; fim=NULL;

- Quando só temos um elemento:

If (inicio == NULL) {

 inicio = novo;

 fim = novo;

}else.....

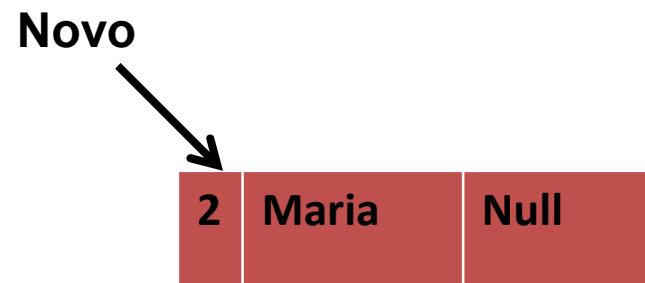
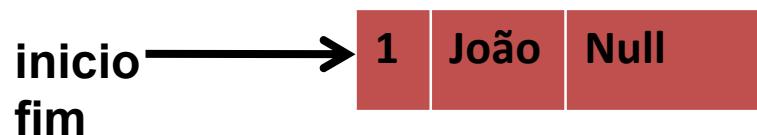


Crie um novo elemento

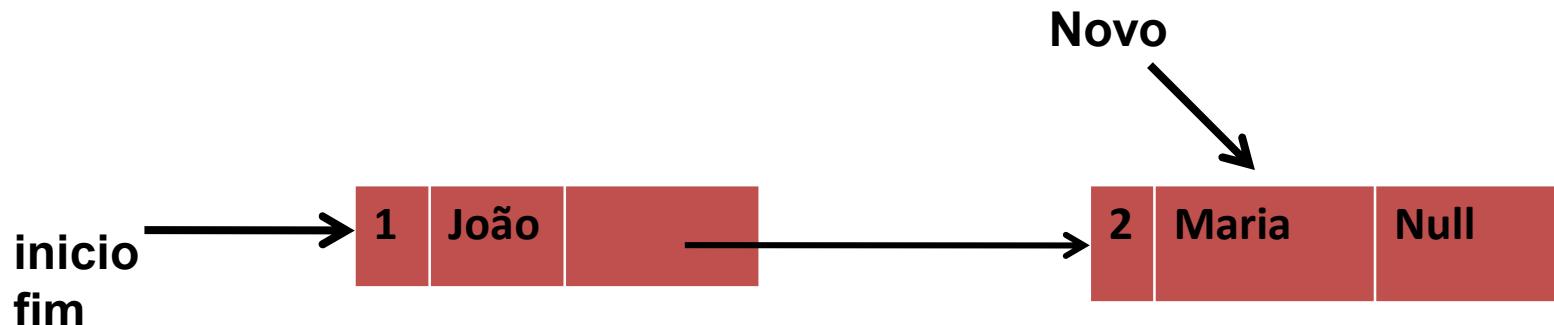
```
novo = (Elemento *)malloc(sizeof(Elemento));  
novo->mat=2;  
strcpy(novo->nome,"Maria");  
novo->prox=NULL;
```



Como encadear os elementos?



- Para encadear os elementos:
`fim->prox=novo;`



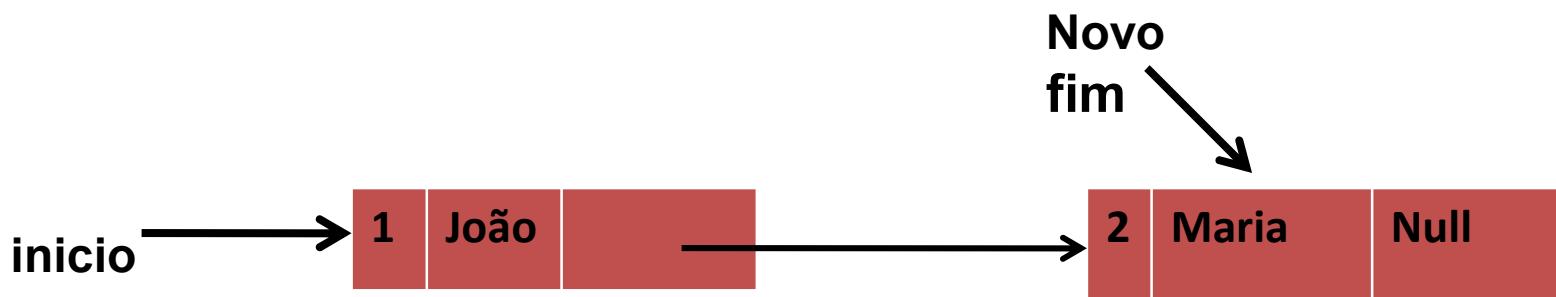
O que ainda falta fazer para atualizar a lista?

Atualize o último elemento da lista

fim = novo;

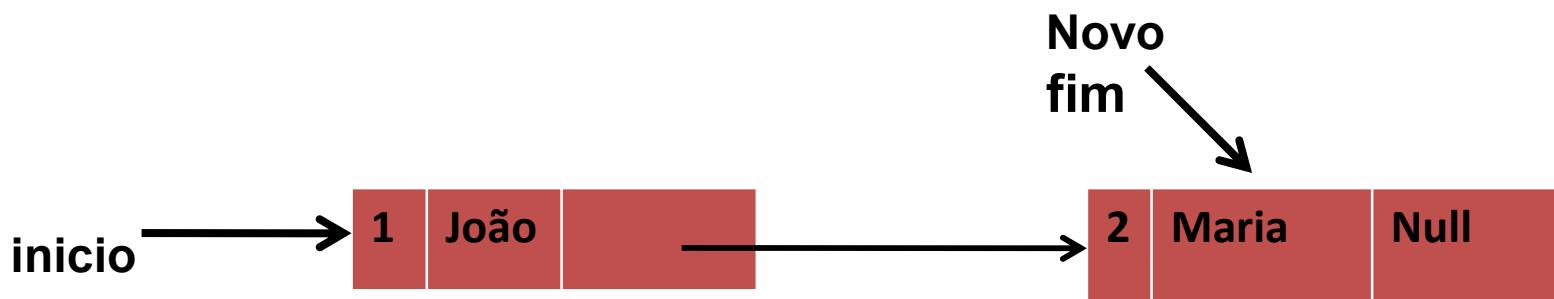
Ou

fim = fim->prox;



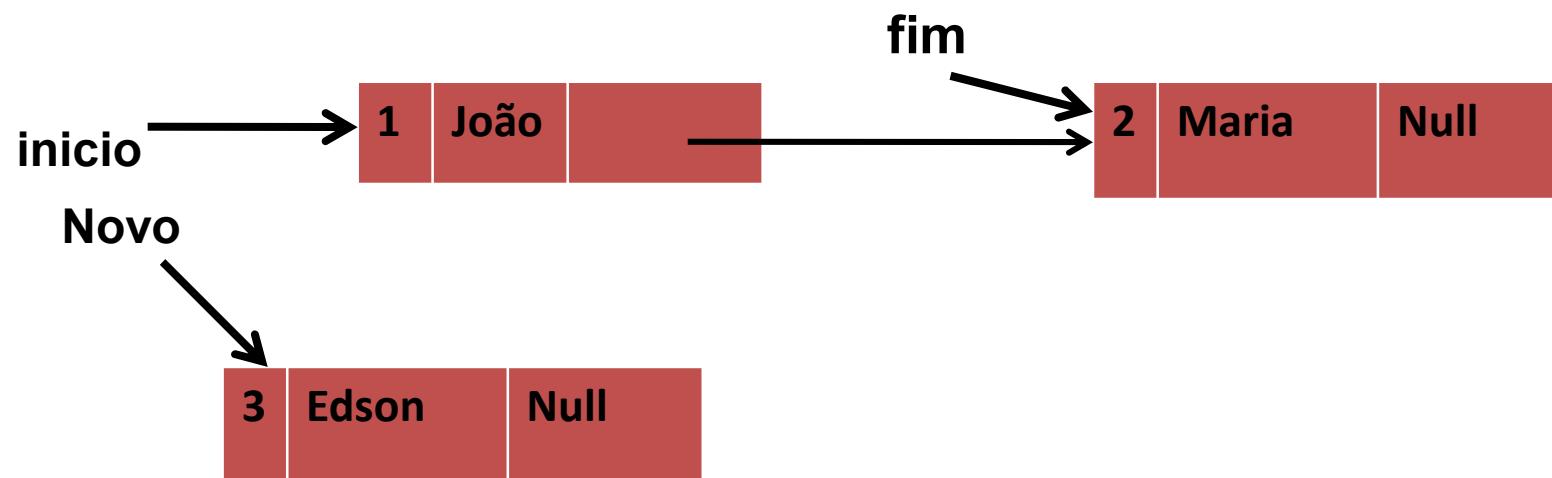
```
If (inicio == NULL) {  
    inicio = novo;  
    fim = novo;  
}else{  
    fim->prox = novo;  
    fim = novo; }
```

Como fazer um programa pra percorrer a lista do inicio até o fim?

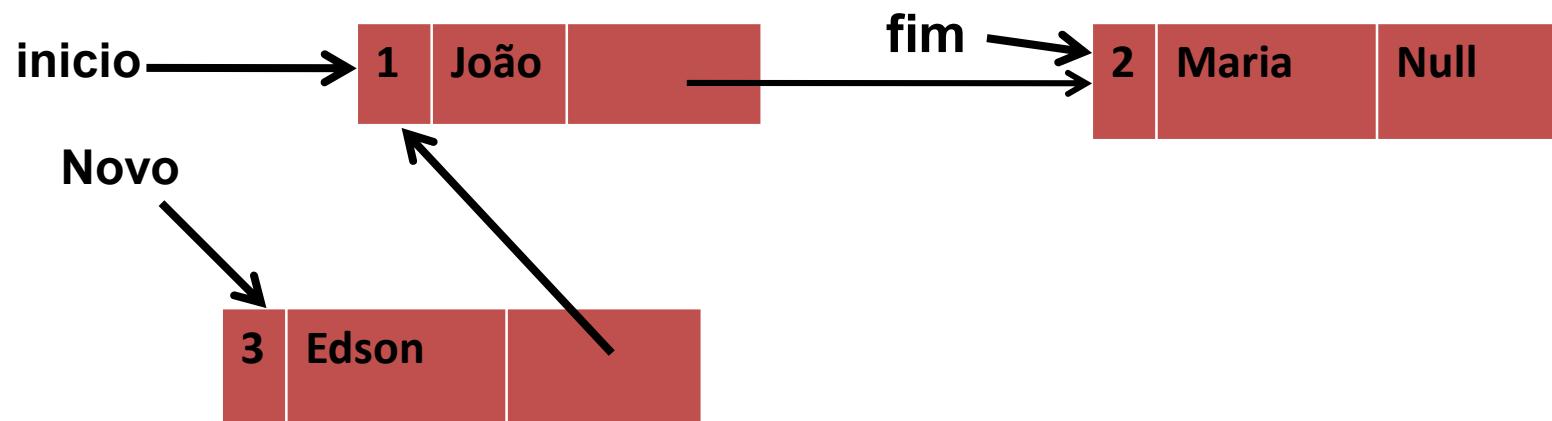


```
atual = inicio;  
while (atual != NULL) {  
    printf("%d",atual->mat);  
    printf("%s",atual->nome);  
    atual=atual->prox;  
}
```

Como incluir o elemento sempre no início da lista?



- Faça com que *novo->prox* aponte para o *inicio*:



172

- Atualize o valor de *inicio* → *inicio = novo*;

Exemplo – Struct aluno

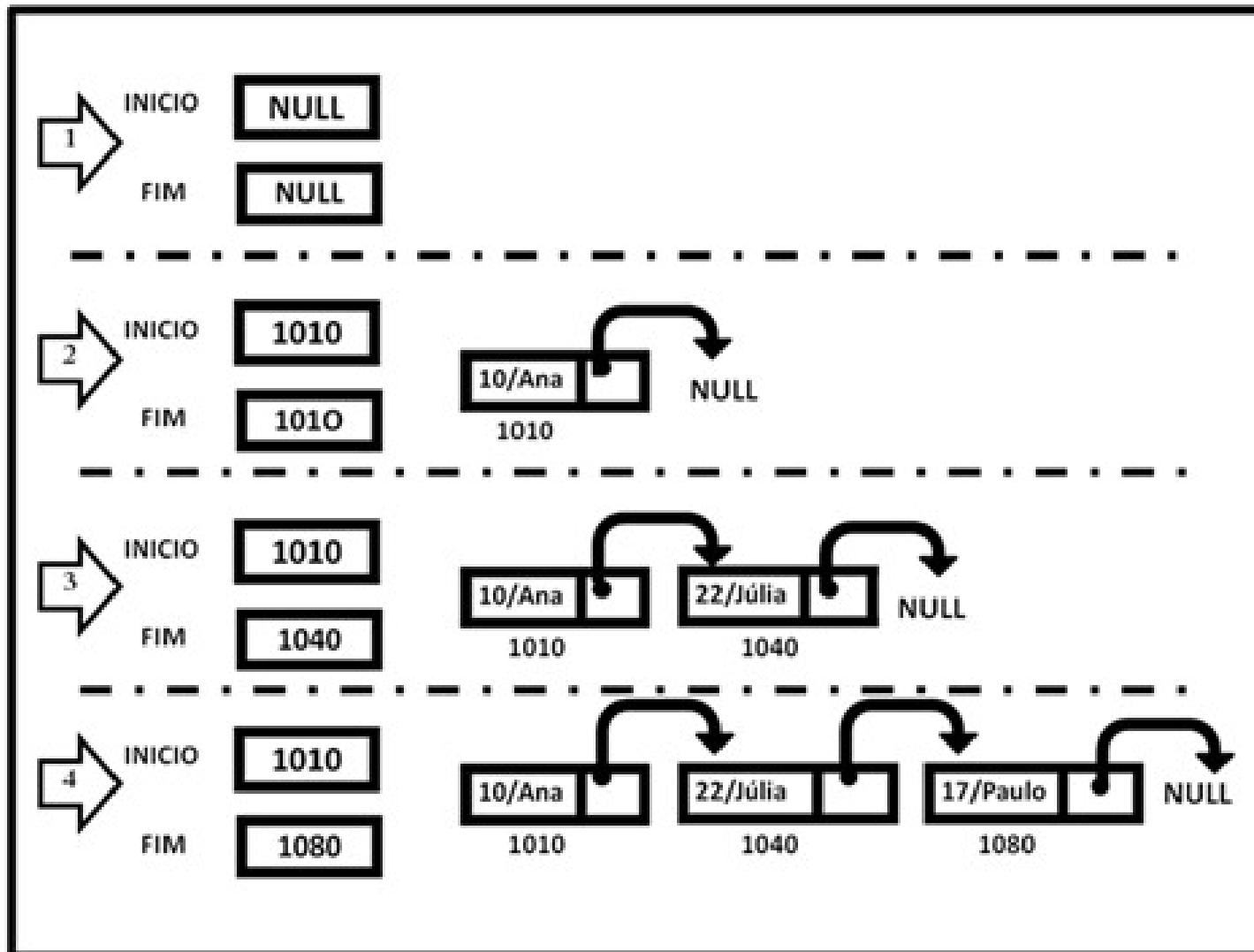
```
struct aluno {  
    int mat;  
    char nome[20];  
    float nota;  
    aluno *prox;  
};
```

```
typedef struct aluno Taluno;
```

```
Taluno *inicio;  
Taluno *fim;  
Taluno *novo;
```

LISTA DINÂMICA DESORDENADA

- Esta lista é implementada usando ponteiros. A memória para armazenar os dados é alocada em tempo de execução.



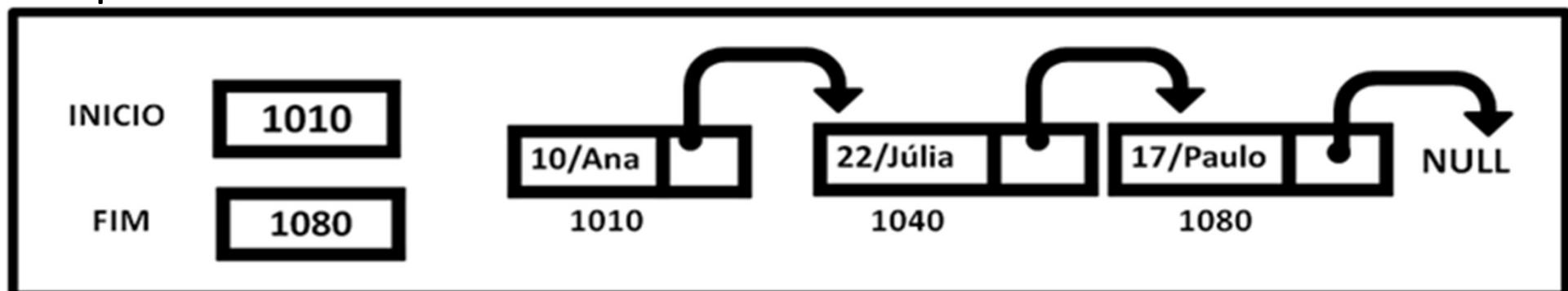
// aloca espaço de memória para novo

```
novo = (Taluno *)malloc(sizeof(Taluno));  
novo->mat= matricula;  
strcpy(novo->nome,nome);  
novo->nota = nota;  
novo->prox = NULL;  
if (inicio==NULL) {  
    inicio = novo;  
    fim = novo;  
}  
else {  
    fim->prox = novo;  
    fim = novo;  
}
```

LISTA DINÂMICA – Inclusão desordenada

LISTA DINÂMICA - Busca

- Para fazer uma consulta em uma lista dinâmica é necessário saber qual elemento deseja consultar. Ex.: consulta por matrícula.
- Um ponteiro auxiliar deve ser usado para percorrer a lista, visitando cada nó a procura do elemento.
- Caso quiséssemos consultar o elemento de matrícula 25, iríamos percorrer a lista até chegar no último nó, cujo endereço do vizinho é NULL (nó de endereço 1080) e ficaríamos sabendo que este elemento não se encontra na lista.
- Quando um elemento é encontrado, seus dados são apresentados
- Quando ele não está na lista, uma mensagem é apresentada dizendo que o elemento não existe na lista.



```
do{  
    printf("\nConsulta aluno pelo numero de matricula\n\n");  
    printf("\nMatricula: ");  
    scanf("%d",&matc);  
    noatual = inicio;      achou = 0;  
    while (noatual != NULL) {  
        if (noatual->mat == matc) {  
            achou = 1;  
            printf("\n\nMatricula Nome\n");  
            printf("-----\n");  printf("%9d %-  
20s\n",noatual->mat, noatual->nome);  
            printf("-----\n");  break;  
        } else  
            noatual = noatual->prox;  }  
}
```

LISTA DINÂMICA - Busca

Lista Dinâmica - Busca

```
if (achou == 0)
    printf("\n\nAluno nao encontrado!!\n");
    printf("\nContinuar consultando (1-
sim/2-nao)? ");
    scanf("%d",&continuar);
}while (continuar == 1);
```

LISTA DINÂMICA – listar todos os elementos

```
void listar(){
    TAluno *noatual = inicio;
    printf("\nListagem de Alunos\n\n");
    if (qa != 0) {
        printf("\n\nMatricula Nome\n");
        printf("-----\n");
        while( noatual != NULL ) {
            printf("%d %s\n",noatual->mat, noatual->nome);
            noatual = noatual->prox;
        }
        printf("-----\n");
        printf("\n\nQuantidade de Alunos = %d\n",qa);
    }
    else
        printf("\n\n Nao tem nenhum aluno cadastrado");
```

LISTA DINÂMICA – Remover um elemento

- Identificar qual elemento deseja remover;
- É feita uma varredura em todos os nós da lista;
- Se elemento é encontrado → remove elemento da lista;
 - 3 situações:
 - O elemento encontrado está no INICIO da lista
 - O elemento encontrado está no MEIO da lista
 - O elemento encontrado está no FINAL da lista
- Senão → uma mensagem deve ser informada ao usuário informando que o elemento não existe.

- Para percorrer a lista crie os ponteiros:

*Elemento *anterior, *atual;*

anterior = NULL;

atual=inicio;

- Podem acontecer três casos diferentes:

➤remover primeiro da lista:

inicio = inicio->prox;

➤ último da lista:

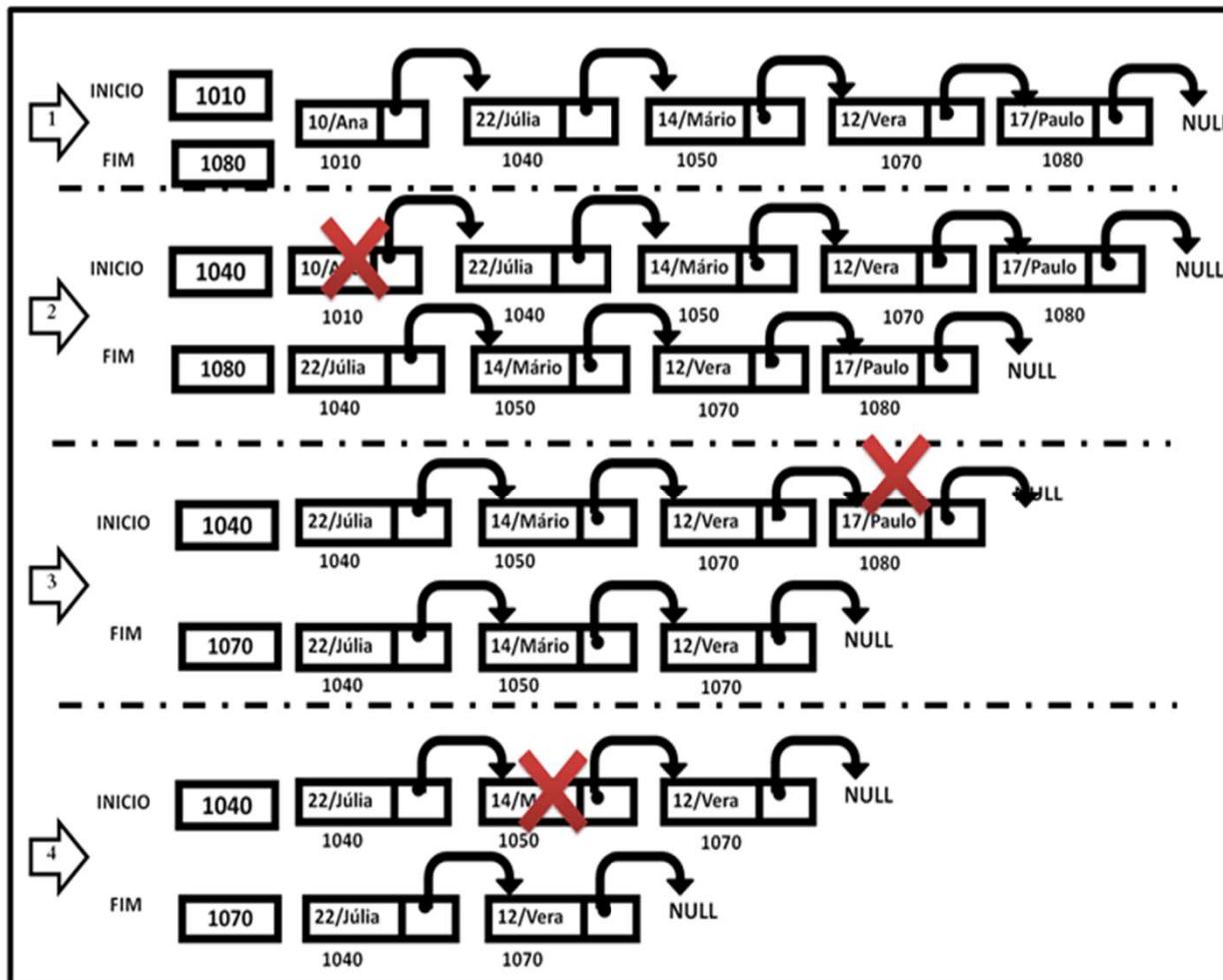
fim = anterior;

fim->prox=NULL;

➤elemento no meio da lista:

Anterior->prox=noatual->prox;

LISTA DINÂMICA – Remover um elemento



- **Primeiro elemento da lista:**
 - Inicio = inicio->prox
- **Último elemento da lista:**
 - fim = anterior;
 - Fim->prox=NULL;
- **Elemento no meio da lista:**
 - anterior->prox=noatual->prox;

```
int remove(int mat){
    No *atual=inicio;
    No *anterior=NULL;
    while (atual!=NULL){
        if (atual->mat==mat){
            if (atual==inicio)
                inicio = inicio->prox;
            else
                if (atual==fim){
                    fim=anterior;
                    fim->prox=NULL;
                }
                else
                    anterior->prox=atual->prox;
            free(atual);
            return 1;
        }
        anterior=atual;
        atual=atual->prox;
    }
    return 0;
}
```

Atividade

- Considerando a seguinte estrutura:

OPÇÃO 1: Usando Struct

```
struct aluno {  
    int mat;  
    char nome[20];  
    aluno *prox;  
};
```

```
typedef struct aluno TAluno;
```

OPÇÃO 2: Usando Classes

```
class No{  
public:  
    int mat;  
    char nome[23];  
    No *prox;  
    No(int m,char n[23]){  
        mat=m;  
        strcpy(nome,n);  
        prox=NULL;  
    }  
};
```

OPÇÃO 2: Usando Classes

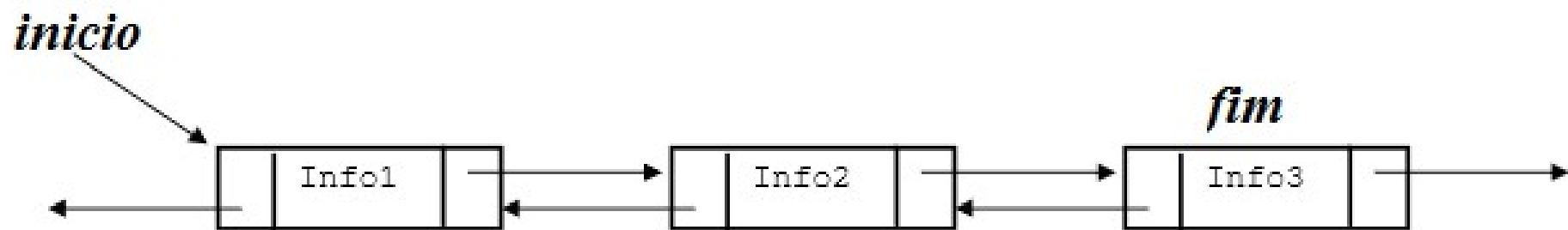
```
class Lista{  
    public:  
        No *inicio;  
        No *fim;  
    Lista(){  
        inicio = NULL;  
        fim = NULL;  
    }  
}
```

Atividade

1. Escreve o método para inserir no inicio da lista.
2. Escreva um método para inserir no final da lista.
3. Escreva um método para inserir na lista ordenado pelo número de matricula.
4. Escreva o método consultar, aonde será pesquisado um aluno através do número de matrícula. Se o aluno for encontrado uma mensagem com matrícula e nome deve ser impressa na tela do computador, se o aluno não for encontrado, a mensagem deve informar que o aluno não existe.
5. Escreva o método remover, aonde será retirado um elemento da lista.
6. Escreva o método listar, aonde serão listados todos os elementos da lista.

Lista duplamente ligada

- Cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior.
- Dado um elemento, é possível acessar o próximo e o anterior
- Dado um ponteiro para o último elemento da lista, é possível percorrer a lista em ordem inversa.



Lista duplamente ligada

- Um novo nó é criado e seus 3 membros de dados devem ser inicializados:

```
class No{  
    public:  
        int mat;  
        char nome[23];  
        No *prox;  
        No *ant;  
        No(int m,char n[23]){  
            mat=m;  
            strcpy(nome,n);  
            prox=NULL;  
            ant=NULL;  
        } } 
```

Lista duplamente ligada

- ***Para inserir no final da lista:***
 - O ponteiro “prox” do último passa a apontar para o novo;
 - O ponteiro “ant” do novo aponta para o último;
 - O último passa a ser o novo.
- ***Para inserir no inicio da lista:***
 - O ponteiro “prox” do novo passa a apontar para o inicio;
 - O ponteiro “ant” do inicio aponta para o novo;
 - O inicio passa a ser o novo.

Lista duplamente ligada

- Inserção ordenada:
 - **No caso da lista NÃO ESTAR VAZIA:** Verique se o elemento a ser inserido é menor do que o elemento atual da lista, SE FOR:
 - Se atual for o inicio:

```
novo->prox=inicio;  
inicio->ant=novo;  
inicio=novo;
```
 - Se estiver no meio da lista:

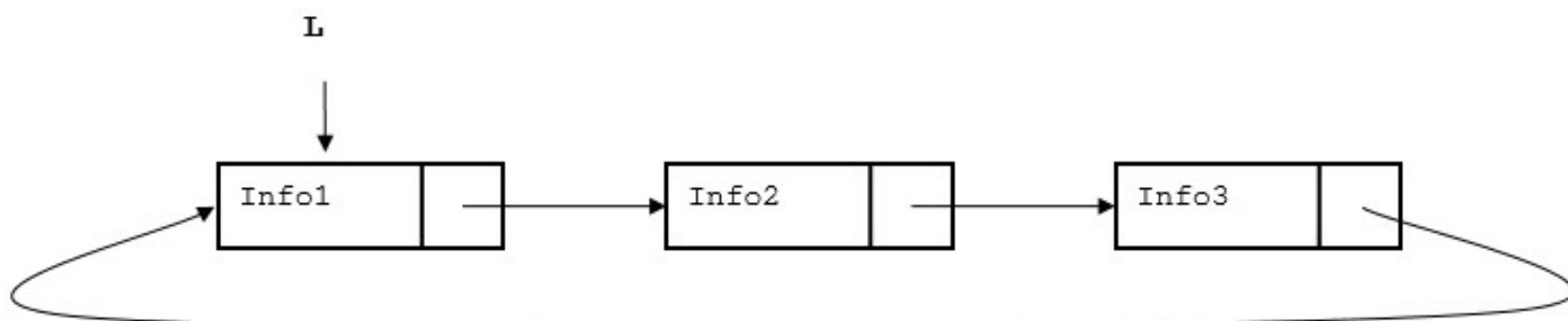
```
novo->prox=atual;  
novo->ant=atual->ant;  
atual->ant->prox=novo;  
atual->ant=novo;
```

Listas duplamente ligadas

- Faça:
 - Inserção ordenada;
 - Inserção no final da lista;
 - Inserção no inicio da lista;
 - Remoção do último elemento;
 - Remoção do primeiro elemento;
 - Remoção do elemento procurado;
 - Mostrar a lista do inicio até o final;
 - Mostrar a lista do final até o inicio;

Listas circulares

- Lista circular:
 - o último elemento tem como próximo o primeiro elemento da lista, formando um ciclo
 - a lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista



Lista circular

- Os nós formam um anel.
- Quando diversos processos estão usando os mesmos recursos ao mesmo tempo e temos que assegurar que nenhum processo acesse o recurso antes de todos os outros → todos os processos são colocados em uma lista circular.

Lista circular

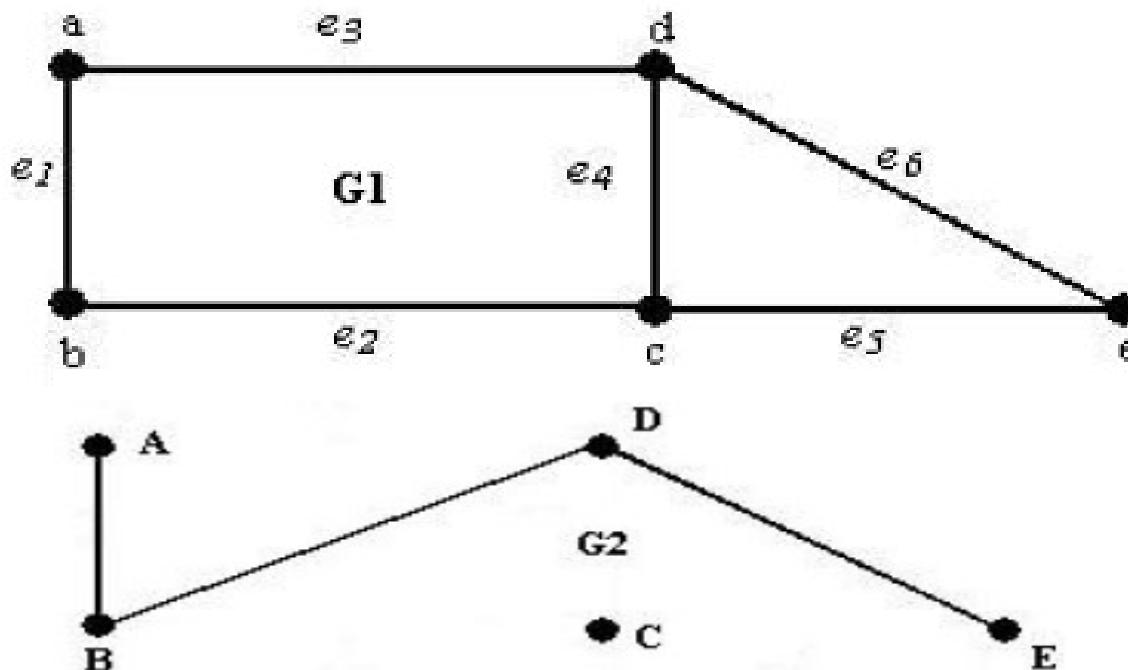
- Faça:
 - Inserção no final da lista;
 - Inserção no inicio da lista;
 - Remoção do último elemento;
 - Remoção do primeiro elemento;
 - Mostrar a lista do inicio até o final;

Grafos

- Um grafo G é definido por $G=(V,E)$, sendo que V representa o conjunto de nós e E , o conjunto de arestas (i,j) , onde i,j pertence a V . Dois nós i, j são vizinhos, se eles estão conectados por uma aresta.

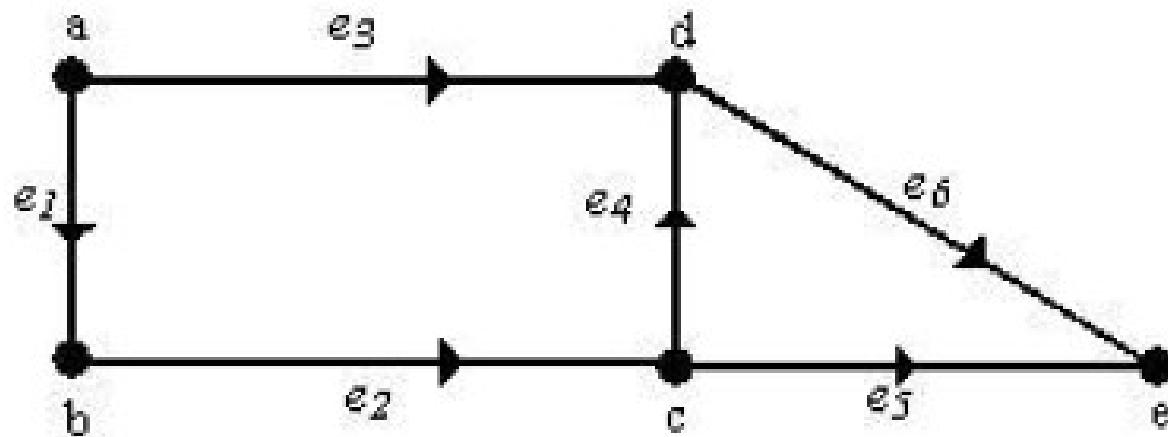
Grafos

- A Figura mostra 2 grafos, o grafo G1 consiste dos conjuntos $V=\{a,b,c,d,e\}$ e $E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$.
- G2 mostra um nó que não é conectado com nenhum outro nó do grafo.



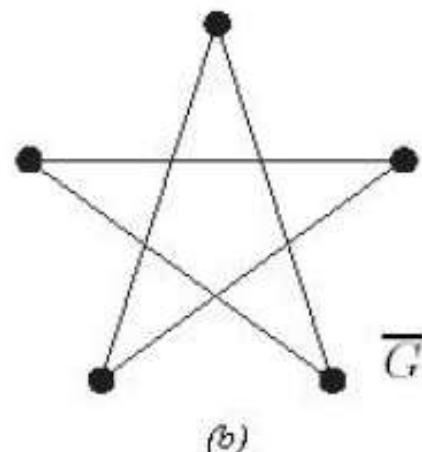
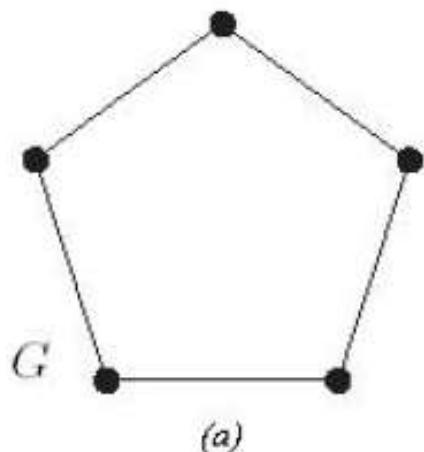
Tipos de Grafos

- Grafo direcional – Um grafo é dito direcional, ou dígrafro, quando é necessário ser estabelecido um sentido (orientação) para as arestas. O sentido da aresta é indicado através de uma seta.



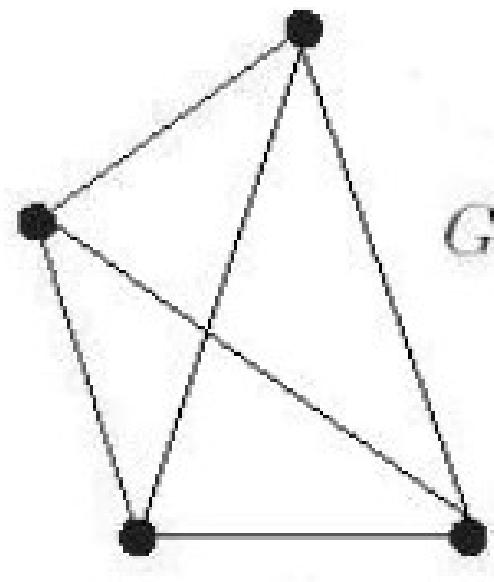
Tipos de Grafos

- Complemento de um grafo – é o grafo com o mesmo conjunto de vértices de G , tal que $i \sim j$ (s o adjacentes) no complemento de G se eles n o forem vizinhos em G .



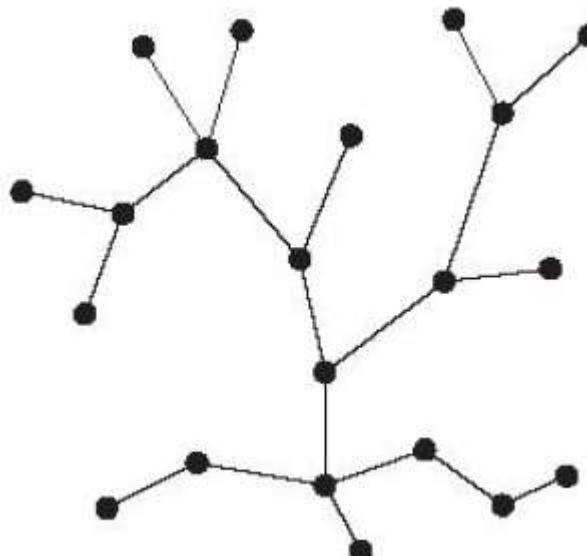
Tipos de Grafos

- Grafo completo – Se todos os vértices de G são mutuamente adjacentes, o grafo é dito completo.



Tipos de Grafos

- Grafo Ponderado – Em um grafo ponderado, um peso ou conjunto de pesos é associado a cada aresta, representado da forma $w(i, j)$, ou seja, $w(1, 2)$ é o peso associado a aresta que une os nós 1 e 2.
- Árvore – Um grafo conexo sem ciclos é chamado de árvore.

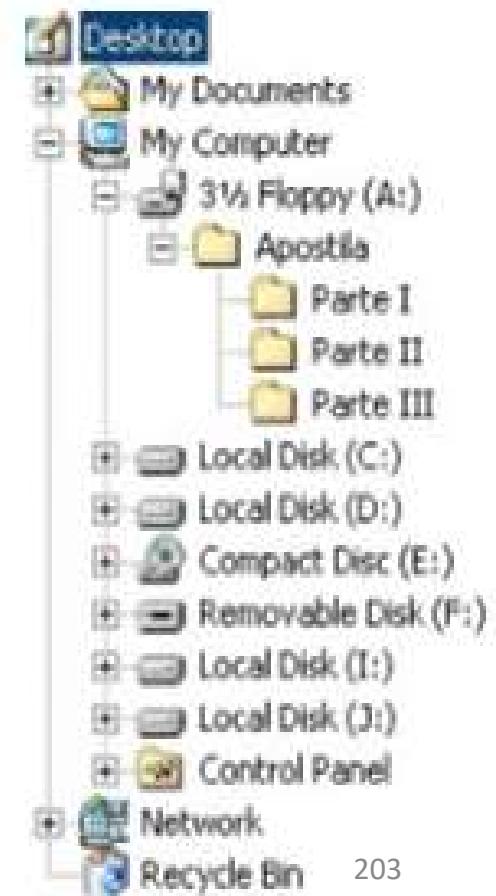


Árvores

- Pode-se designar um nó para ser a ***raiz*** da árvore, o que demonstra uma relação lógica entre os nós.
- Essas árvores são ditas hierárquicas e a distância entre cada nó e a raiz é denominada de ***nível***.
- Em uma árvore hierárquica os nós podem ser rotulados de acordo com a denominação de uma árvore genealógica: ***filhos, pais e ancestrais***.
- Uma árvore hierárquica onde cada nó da origem a dois outros nós de nível inferior é chamada de ***árvore binária***.

Árvores

- **VETORES E LISTAS:** estruturas **unidimensionais ou lineares.** <<Não são adequadas para representar dados dispostos de maneira hierárquica.>>
- Por exemplo, **os arquivos (documentos) que criamos num computador** são armazenados dentro de uma estrutura hierárquica de diretórios (pastas).
 - Existe um diretório base que armazena subdiretórios e arquivos.
Por sua vez, dentro dos sub-diretórios, existem outros sub-diretórios e arquivos, e assim por diante, **recursivamente.**



Árvores

- Uma árvore é um conjunto de nós composto de um **nó especial (chamado raiz)** e conjuntos disjuntos de **nós subordinados ao nó raiz** que são eles próprios **(sub)árvores**.
- Um nó r, denominado **raiz**, contém zero ou mais **sub-árvores**, cujas raízes são ligadas diretamente a r.
- Esses nós raízes das sub-árvores são **filhos do nó pai, r**.
- **Nós internos:** Nós com filhos.
- **Folhas ou Nós externos:**
- Nós que não têm filhos.

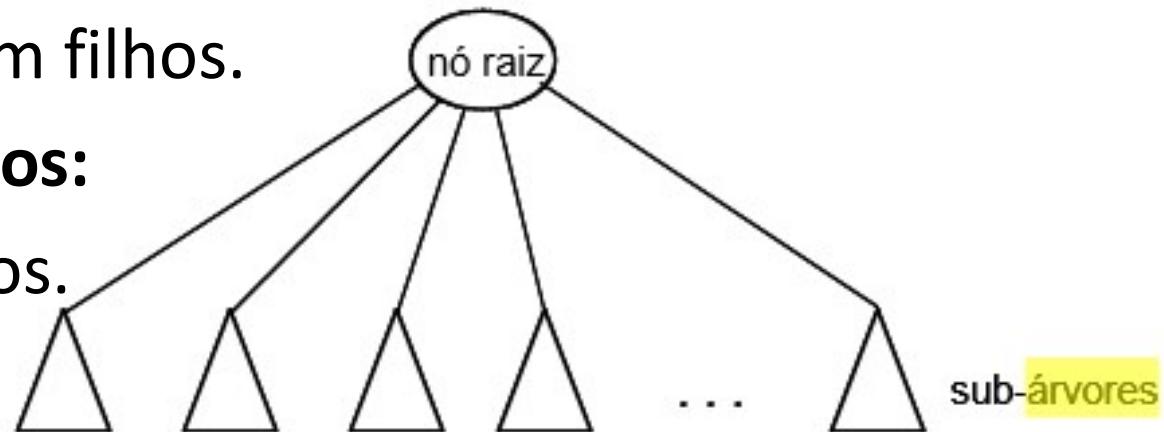


Figura 13.2: Estrutura de árvore.

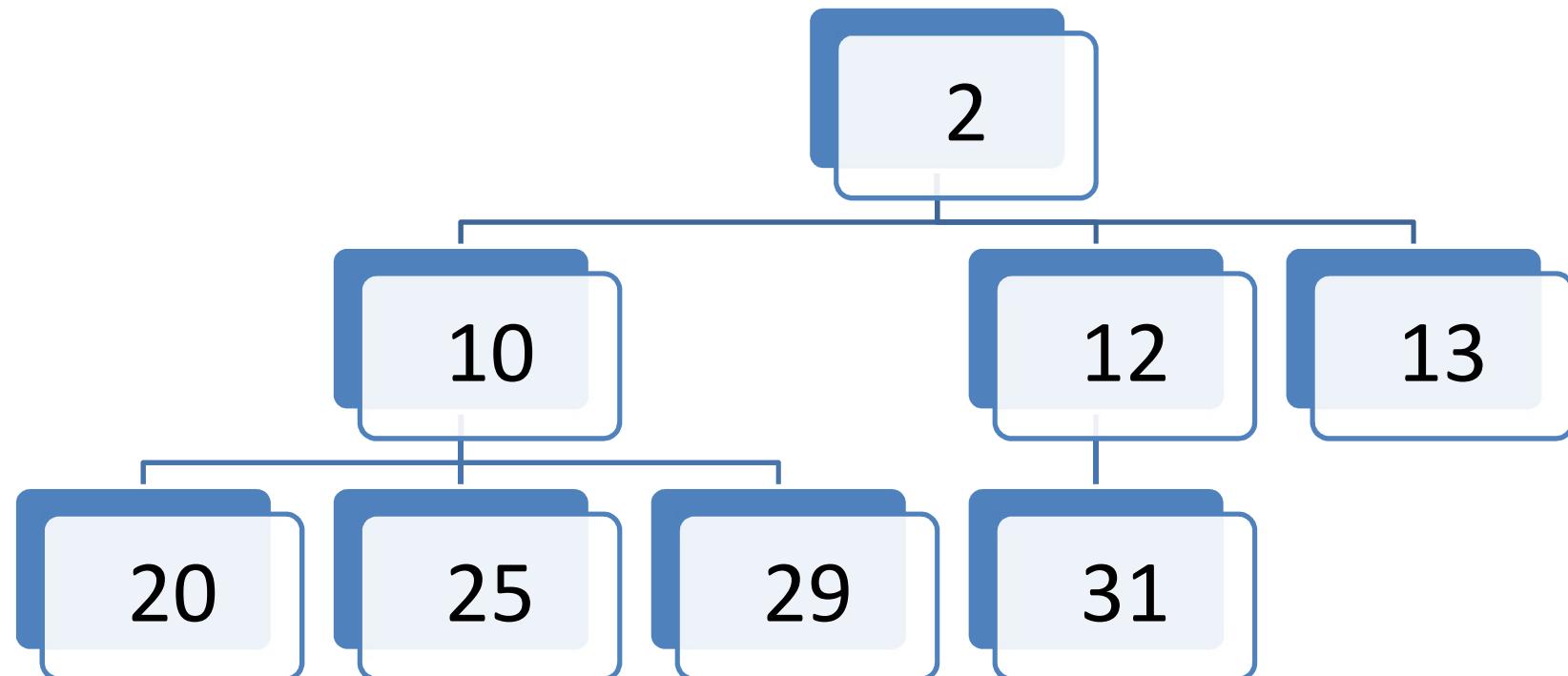
Árvores

- O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os diversos tipos de árvores existentes.
- árvores binárias: onde cada nó tem, no máximo, dois filhos.
- árvores genéricas: onde o número de filhos é indefinido.
- Estruturas recursivas serão usadas como base para implementação das operações com árvores.

Exemplo Árvore Genérica

- Exemplo: Representação da *lista01* em uma árvore.
- *lista01*:

2 ->	10 ->	12 ->	13->	20 ->	25 ->	29 ->	31 Null
------	-------	-------	------	-------	-------	-------	------------

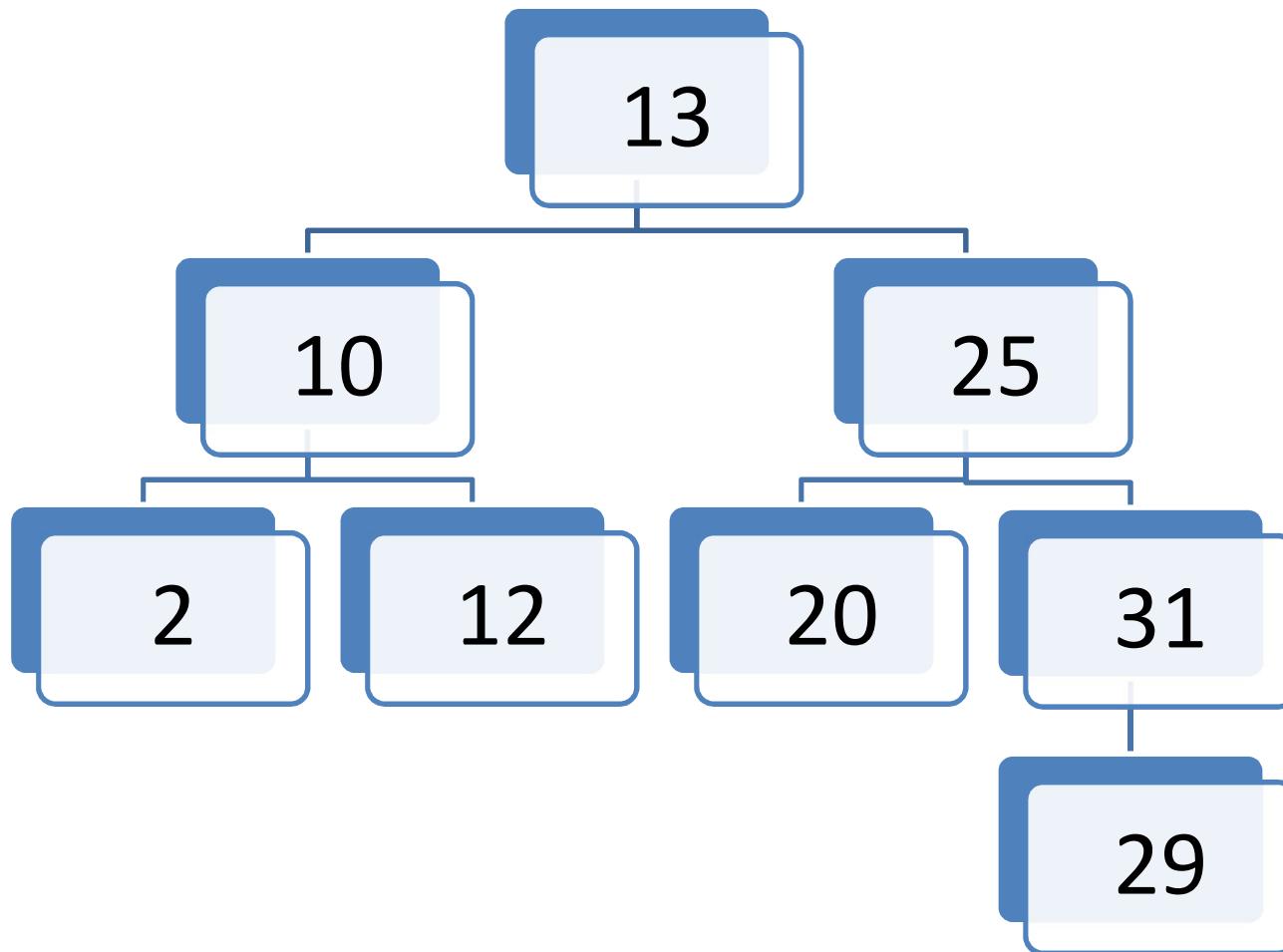


Exemplo Árvore Genérica

- Para localizar um elemento a pesquisa tem que começar do início da lista.
- Mesmo que esteja ordenada ele vai sempre começar do primeiro nó.

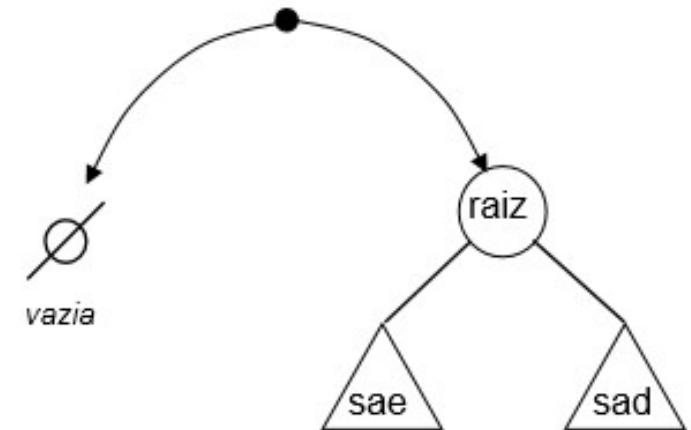
<<Se todos os elementos estão armazenados em uma árvore ordenada, segundo algum critério, o número de testes pode ser reduzido substancialmente.>>

Transforma a Árvore Genérica em uma Árvore Binária



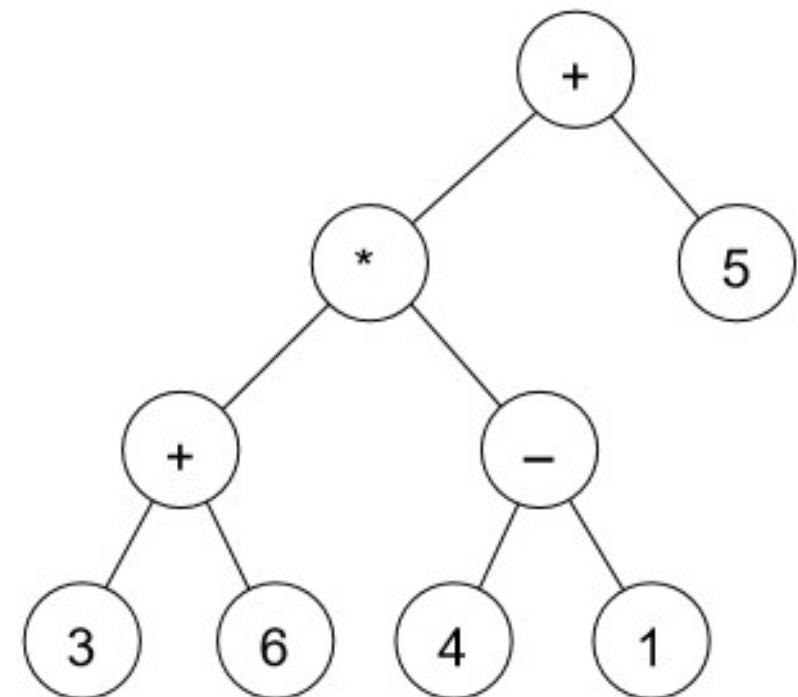
Árvores Binárias

- um árvore em que cada nó tem zero, um ou dois filhos
- uma árvore binária é:
 - uma árvore vazia; ou
 - um nó raiz com duas sub-árvores:
 - a sub-árvore da direita (sad)
 - a sub-árvore da esquerda (sae)



Árvores Binárias

- Exemplo
 - árvores binárias representando expressões aritméticas:
 - nós folhas representam operandos
 - nós internos operadores
 - exemplo: $(3+6)*(4-1)+5$



Árvores Binárias

- Representação de uma árvore:
 - através de um ponteiro para o nó raiz
- Representação de um nó da árvore:
 - estrutura em C contendo
 - a informação propriamente dita (exemplo: um caractere)
 - dois ponteiros para as sub-árvores, à esquerda e à direita

```
struct noArv {  
    char info;  
    struct noArv* esq;  
    struct noArv* dir;  
};
```

Proposta Implementação - 1

```
typedef struct noArv NoArv;

NoArv* arv_criavazia (void);
NoArv* arv_cria (char c, NoArv* e, NoArv* d);
NoArv* arv_libera (NoArv* a);
int arv_vazia (NoArv* a);
int arv_pertence (NoArv* a, char c);
void arv_imprime (NoArv* a);
```

Proposta Implementação - 1

- função `arv_criavazia`
 - cria uma árvore vazia

```
NoArv* arv_criavazia (void)
{
    return NULL;
}
```

Proposta Implementação - 1

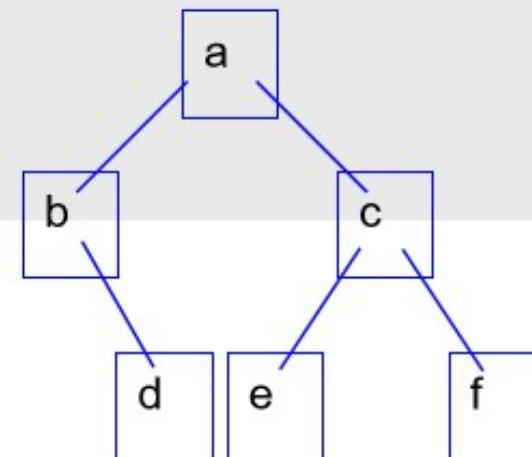
- função `arv_cria`
 - cria um nó raiz dadas a informação e as duas sub-árvores, a da esquerda e a da direita
 - retorna o endereço do nó raiz criado

```
NoArv* arv_cria (char c, NoArv* sae, NoArv* sad)
{
    NoArv* p=(NoArv*)malloc(sizeof(NoArv));
    if(p==NULL) exit(1);
    p->info = c;
    p->esq = sae;
    p->dir = sad;
    return p;
}
```

Implementação 1 - Exemplo

- Exemplo: <a <b <> <d <><>> > <c <e <><>> > <f <><>> > >

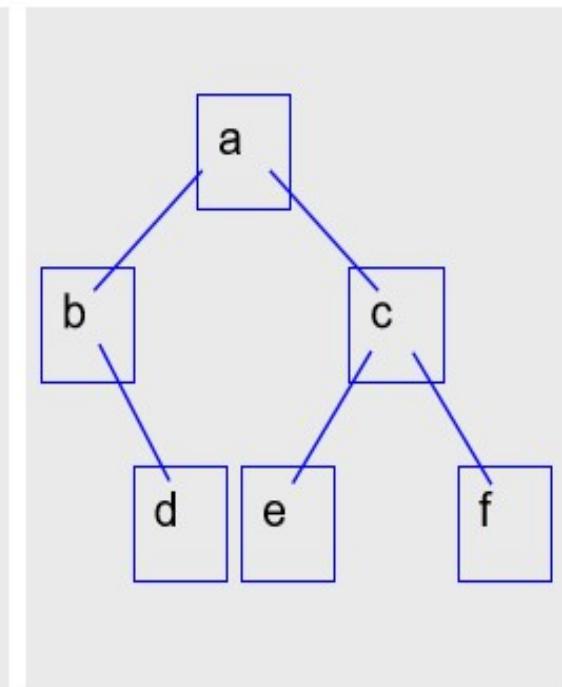
```
/* sub-árvore 'd' */
NoArv* a1= arv_cria('d',arv_criavazia(),arv_criavazia());
/* sub-árvore 'b' */
NoArv* a2= arv_cria('b',arv_criavazia(),a1);
/* sub-árvore 'e' */
NoArv* a3= arv_cria('e',arv_criavazia(),arv_criavazia());
/* sub-árvore 'f' */
NoArv* a4= arv_cria('f',arv_criavazia(),arv_criavazia());
/* sub-árvore 'c' */
NoArv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
NoArv* a = arv_cria('a',a2,a5 );
```



Implementação 1

- Exemplo: <a <b <> <d <><>> > <c <e <><>> <f <><>> > > >

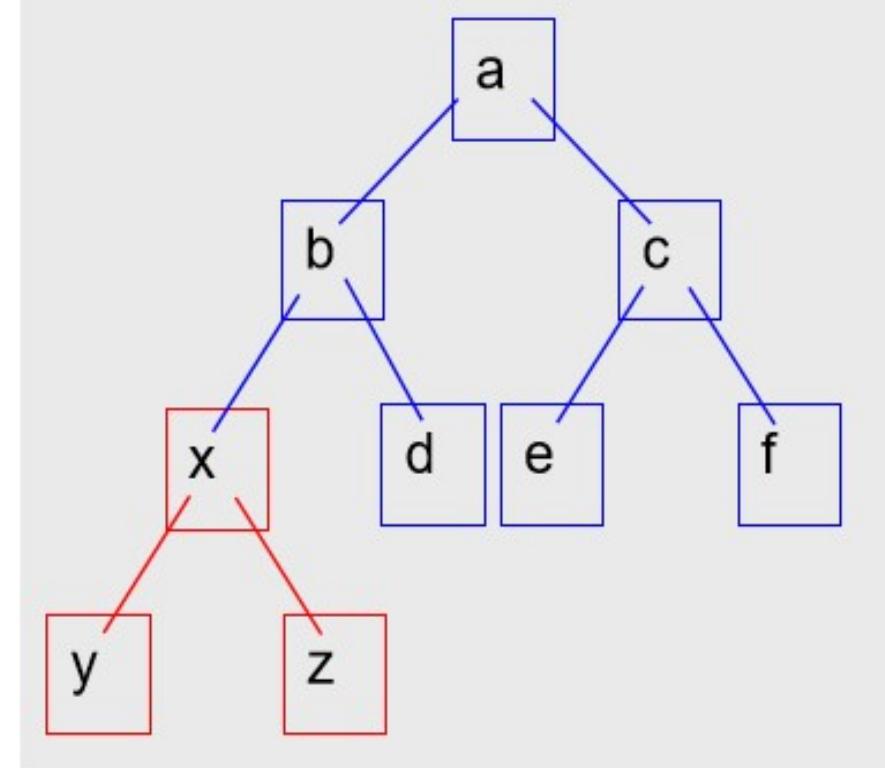
```
NoArv* a = arv_cria('a',
    arv_cria('b',
        arv_criavazia(),
        arv_cria('d', arv_criavazia(), arv_criavazia())
    ),
    arv_cria('c',
        arv_cria('e', arv_criavazia(), arv_criavazia()),
        arv_cria('f', arv_criavazia(), arv_criavazia())
    )
);
```



Implementação 1

- Exemplo - acrescenta nós

```
a->esq->esq =  
    arv_cria('x',  
              arv_cria('y',  
                        arv_criavazia(),  
                        arv_criavazia()),  
              arv_cria('z',  
                        arv_criavazia(),  
                        arv_criavazia()))  
    );
```



Implementação 1

- função `arv_imprime`
 - percorre recursivamente a árvore, visitando todos os nós e imprimindo sua informação

```
void arv_imprime (NoArv* a)
{
    if (!arv_vazia(a)) {
        printf("%c ", a->info);      /* mostra raiz */
        arv_imprime(a->esq);        /* mostra sae */
        arv_imprime(a->dir);        /* mostra sad */
    }
}
```

Implementação 2

```
class No{  
    public:  
        char nome;  
        No *left;  
        No *right;  
        No(char n){  
            nome=n;  
            left=NULL;  
            right=NULL;  
        }  
};
```

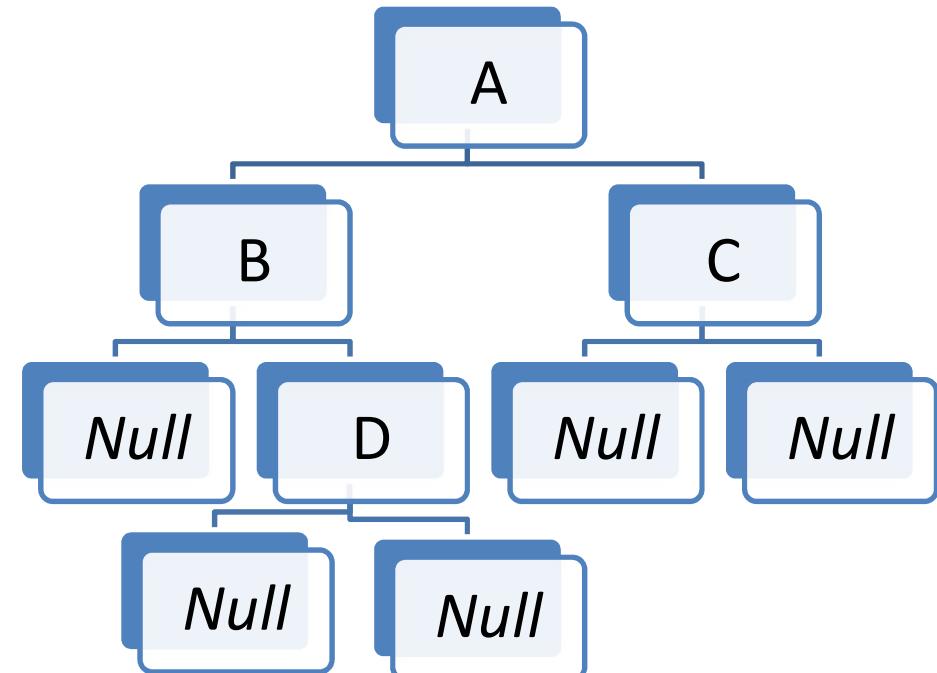
```
class Arvore {  
    public:  
        No *raiz;  
  
    Arvore(){  
        raiz= NULL;  
    }  
  
    int isEmpty(){  
        return (raiz==NULL);  
    }  
.....}
```

Atividade

- Implemente a criação de um novo nó, de maneira que, o método informe o pai do novo nó e de que lado ele deve ser inserido.

DICA

```
main(){
    //lado 1=esq e 2=dir
    Arvore *arv = new Arvore();
    arv->cria_No2('A');
    arv->cria_No2('B',1,'A');
    arv->cria_No2('D',2,'B');
    arv->cria_No2('T',2,'B');
    arv->cria_No2('C',2,'A');
    arv->imprime(arv->raiz);
}
```



DICA

```
void cria_No(char nov){  
    No *novo=new No(nov,NULL,NULL);  
    raiz=novo;  
  
}  
void cria_No(char nov, int lado, char pai){  
    No *novo=new No(nov,NULL,NULL);  
    insere(raiz,novo,lado,pai);  
}
```

<<Obs: O método insere() é recursivo.>>

```
// 1- ESQ 2 - DIR
void insere(No *arv, No *novo, int lado, char pai){
    if (arv!=NULL){
        if (arv->info==pai){
            if (lado==1)
                if (arv->esq==NULL)
                    arv->esq = novo;
                else
                    cout<<"\n ERRO - ja existe um no nessa posicao!!!";
            if (lado==2)
                if (arv->dir==NULL)
                    arv->dir = novo;
                else
                    cout<<"\n ERRO - ja existe um no nessa posicao!!!";
        }
    else{
        insere(arv->dir,novo,lado,pai);
        insere(arv->esq,novo,lado,pai);
    }
}
```

Dica:

DICA:

```
void imprime(No *n){  
  
    if (raiz==NULL){  
        cout<<"\n <VAZIO>";  
    }  
    else{  
        if (n!=NULL){  
            cout<<"<"<<n->info;  
            imprime(n->esq);  
            imprime(n->dir);  
            cout<<">";  
        }  
        else  
            cout<<"<>";  
    }  
}
```

Verificando se existe o Nó

- função `arv_vazia`
 - indica se uma árvore é ou não vazia

```
int arv_vazia (NoArv* a)
{
    return a==NULL;
}
```

Retirando um nó da árvore

- função `arv_libera`
 - libera memória alocada pela estrutura da árvore
 - as sub-árvore devem ser liberadas antes de se liberar o nó raiz
 - retorna uma árvore vazia, representada por `NULL`

```
NoArv* arv_libera (NoArv* a) {
    if (!arv_vazia(a)) {
        arv_libera(a->esq);      /* libera sae */
        arv_libera(a->dir);      /* libera sad */
        free(a);                  /* libera raiz */
    }
    return NULL;
}
```

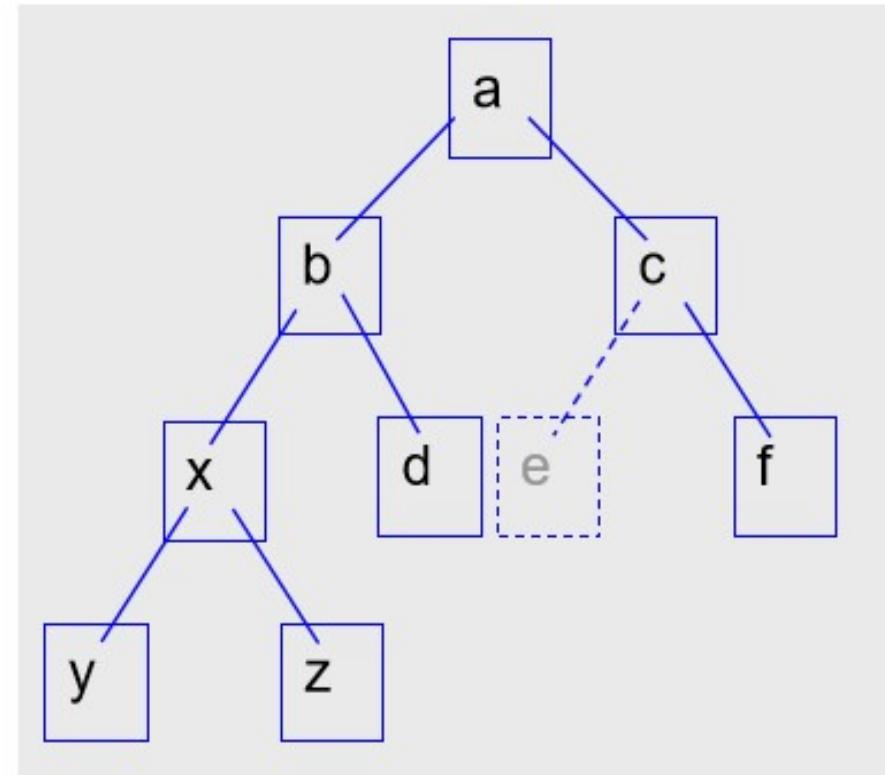
Exemplo

- Exemplo - libera nós

```
a->dir->esq = libera(a->dir->esq);
```

OBSERVE que ele já passa o ENDEREÇO DO PAI como parâmetro!

Se você NÃO tem o endereço do pai, É NECESSÁRIO INICIALMENTE ACHAR O PAI DO NÓ QUE DEVE SER RETIRADO!!!



- Primeiro encontre o pai do nó que deve ser retirado.
- Para isso, passe como parâmetro para o método o nó raiz e o procurado.

```

void liberaFilho(char procurado, No *n){
    No *tmp=NULL;
    if (raiz->info==procurado){
        tmp=raiz;
        libera(tmp);
        raiz=NULL;
    }
    else{
        if (n->dir!=NULL){
            if (n->dir->info==procurado){
                No *prox=n->dir;
                n->dir=NULL; ←
                libera(prox);
            }
            else
                liberaFilho(procurado, n->dir);
        }
        if (n->esq!=NULL){
    }
}

```

DICA

Fazer o filho
apontar
para NULL

DICA

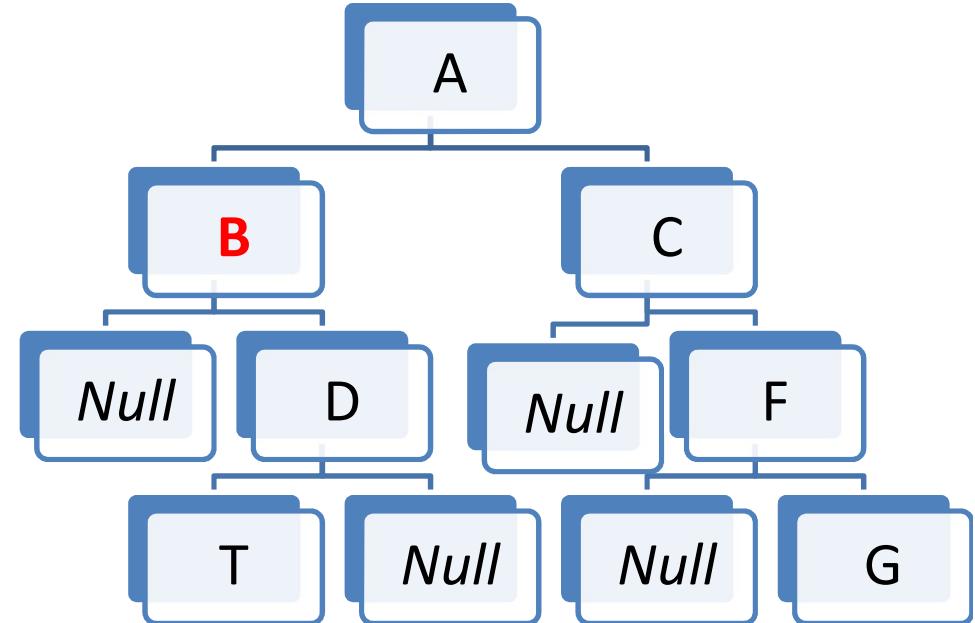
- Depois de fazer o ponteiro *pai->esq* ou *pai->dir* apontar para **NULL**, libere o filho da memória (aplicar *free(filho)*):

```
void libera(No *pai){  
    No *tmp;  
  
    if (pai!=NULL){  
        if (pai->esq!=NULL){  
            libera(pai->esq);  
        }  
        if (pai->dir!=NULL){  
            libera(pai->dir);  
        }  
  
        free(pai);  
    }  
}
```

Exemplo

- Primeiro encontra o B:

```
arv->liberaFilho('B',arv->raiz);
```



- Fazer *pai->dir* ou *pai->esq* igual a NULL, dependendo de onde está o filho. Depois liberar a sub-arvore:

```
if (pai->dir!=NULL)
```

```
    if (pai->dir->nome==procurado){
```

```
        No *prox=pai->dir;
```

```
        pai->dir=NULL;
```

```
        libera(prox); }
```

Exemplo

- Se for folha: verifica se os lados do pai são iguais a nulo, se for (é folha) pode liberar a memória: *free(a)*
- Se não for folha:
 - Se lado esquerdo é diferente de nulo, percorre filhos:
libera(a->esq);
 - Se lado direito é diferente de nulo, percorre filhos:
libera(a->dir);
 - Depois de remover os filhos, remova o pai: *free(a)*
- No exemplo: libera da memória da seguinte ordem:T, D, B

Verificando se o nó pertence a árvore

- função `arv_pertence`
 - verifica a ocorrência de um caractere c em um de nós
 - retorna um valor booleano (1 ou 0) indicando a ocorrência ou não do caractere na árvore

```
int arv_pertence (NoArv* a, char c)
{
    if (arv_vazia(a))
        return 0; /* árvore vazia: não encontrou */
    else
        return a->info==c ||
               arv_pertence(a->esq,c) ||
               arv_pertence(a->dir,c);
}
```

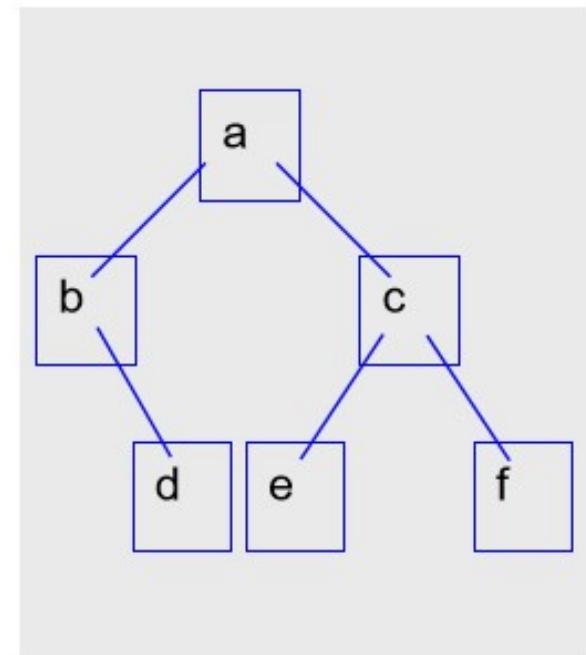
DICA

- Se o nó NÃO foi encontrado, verifique os filhos do nó (esquerda e direita).

```
void buscar(No *ele, char n){  
  
    if (ele->info==n)  
        cout<<"\n Elemento "<<ele->info<<" encontrado!!!!";  
    else{  
        if (ele->dir!=NULL)  
            buscar(ele->dir,n);  
  
        if (ele->esq!=NULL)  
            buscar(ele->esq,n);  
  
    }  
}
```

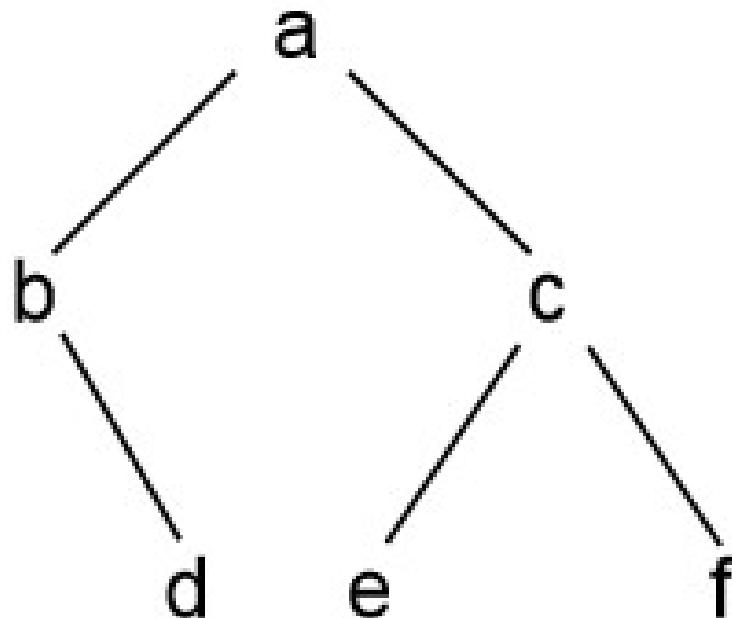
Ordens de Percurso

- Ordens de percurso:
 - *pré-ordem*:
 - trata *raiz*, percorre *sae*, percorre *sad*
 - exemplo: a b d c e f
 - *ordem simétrica*:
 - percorre *sae*, trata *raiz*, percorre *sad*
 - exemplo: b d a e c f
 - *pós-ordem*:
 - percorre *sae*, percorre *sad*, trata *raiz*
 - exemplo: d b e f c a



Atividade

1. Modifique a implementação de imprime, de forma que a saída impressa reflita, além do conteúdo de cada nó, a estrutura da árvore, usando a notação introduzida anteriormente. Assim, a saída da função seria: <a<b<><d<><>>><c<e<><>><f<><>>>> para o exemplo ao lado.

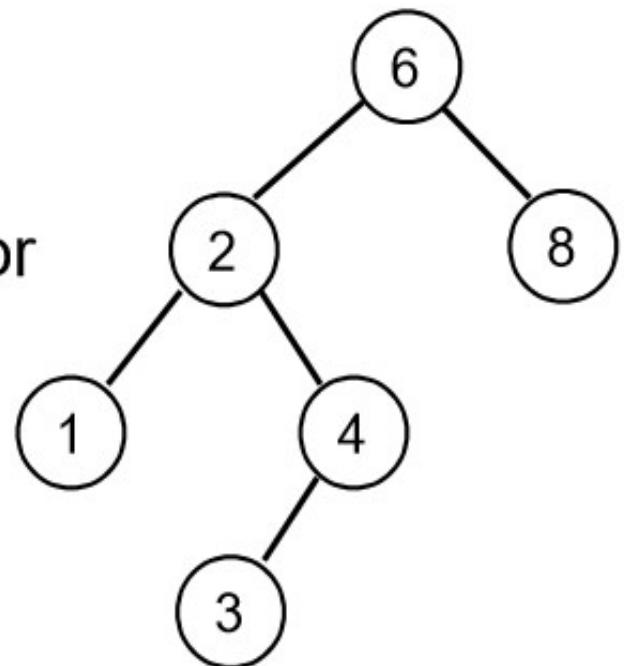


Atividade

3. Faça método de impressão usando ordem simétrica.
4. Faça método de impressão usando pós-ordem.
5. Procure um nó na árvore. Imprima o nó encontrado, se existir, ou informe que o nó não existe.
6. Procure o pai de um nó na árvore. Imprima o nome do pai, se existir, ou informe que o pai não existe.
7. Libere um nó na árvore.

Árvore Binária de Busca (ABB)

- o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (sae) e
- o valor associado à raiz é sempre menor ou igual (para permitir repetições) que o valor associado a qualquer nó da sub-árvore à direita (sad)
- quando a árvore é percorrida em ordem simétrica (sae - raiz - sad), os valores são encontrados em ordem não decrescente



Pesquisa em ABB

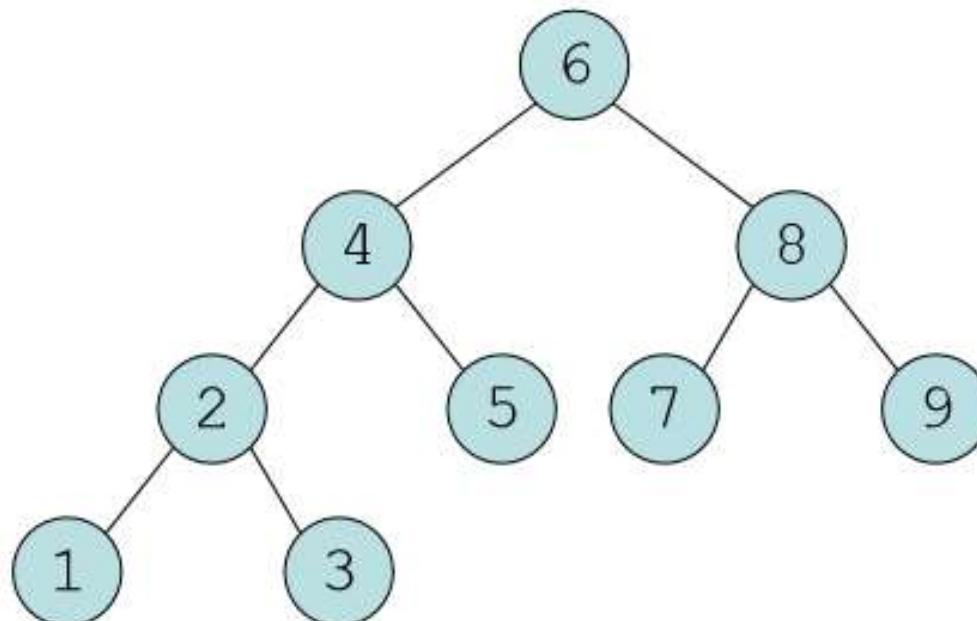
- compare o valor dado com o valor associado à raiz
- se for igual, o valor foi encontrado
- se for menor, a busca continua na sae
- se for maior, a busca continua na sad

DICA

```
No *buscarOrd(No *ele, char n){  
    if (ele->info==n)  
        return ele;  
    else{  
        if (ele->info<n){  
            if (ele->dir!=NULL)  
                return buscarOrd(ele->dir,n);  
            else  
                return NULL;  
        }  
        if (ele->info>n){  
            if (ele->esq!=NULL)  
                return buscarOrd(ele->esq,n);  
            else  
                return NULL;  
        }  
    }  
}
```

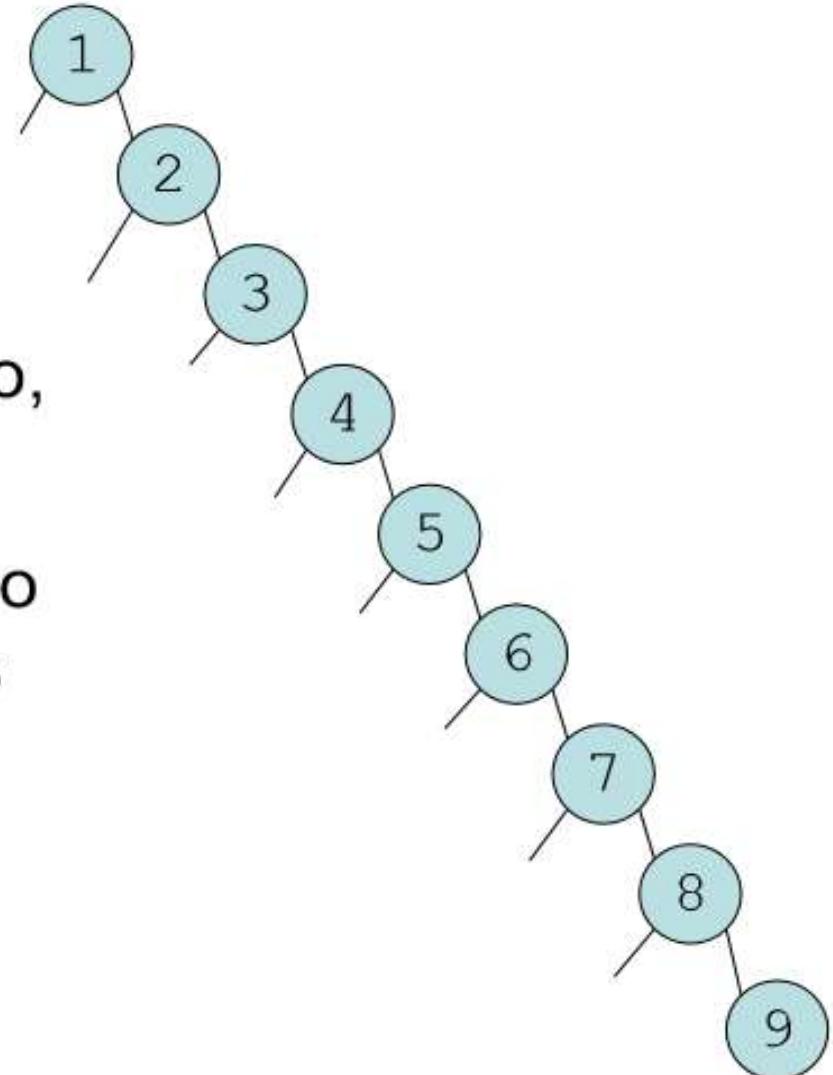
Árvore Binária de Busca Balanceada

- os nós internos têm todos, ou quase todos, 2 filhos
- qualquer nó pode ser alcançado a partir da raiz em $O(\log n)$ passos



Árvore Binária de Busca Degenerada ou Assimétrica

- todos os nós têm apenas 1 filho, com exceção da (única) folha
- qualquer nó pode ser alcançado a partir da raiz em $O(n)$ passos



Busca na Árvore Binária

- explora a propriedade de ordenação da árvore
- possui desempenho computacional proporcional à altura ($O(\log n)$ para o caso de árvore balanceada)

```
NoArv* abb_busca (NoArv* r, int v)
{
    if (r == NULL)
        return NULL;
    else if (r->info > v)
        return abb_busca (r->esq, v);
    else if (r->info < v)
        return abb_busca (r->dir, v);
    else
        return r;
}
```

Adicionando um nó em uma ABB

- para adicionar v na posição correta, faça:
 - se a (sub-)árvore for vazia
 - crie uma árvore cuja raiz contém v
 - se a (sub-)árvore não for vazia
 - compare v com o valor na raiz
 - insira v na sae ou na sad, conforme o resultado da comparação

Inserindo novo Nó na classe Árvore

```
void criaNo(char n){  
    No *novo= new No(n);  
    insere(raiz,novo);  
}
```

Inserindo nó - sem uso de recursão

```
void insere(No *arv, No *n){
    if (isEmpty() == 1) //raiz nula, arvore vazia
        raiz=n;
    else{
        No *percorre=raiz;

        while (percorre!=NULL){
            if (percorre->nome<n->nome)
                if (percorre->right==NULL){
                    percorre->right=n;
                    break;
                }
            else
                percorre=percorre->right;
        }
    }
}
```

ATIVIDADE:

Continue a Implementação acima!

Inserindo nó – usando de recursão

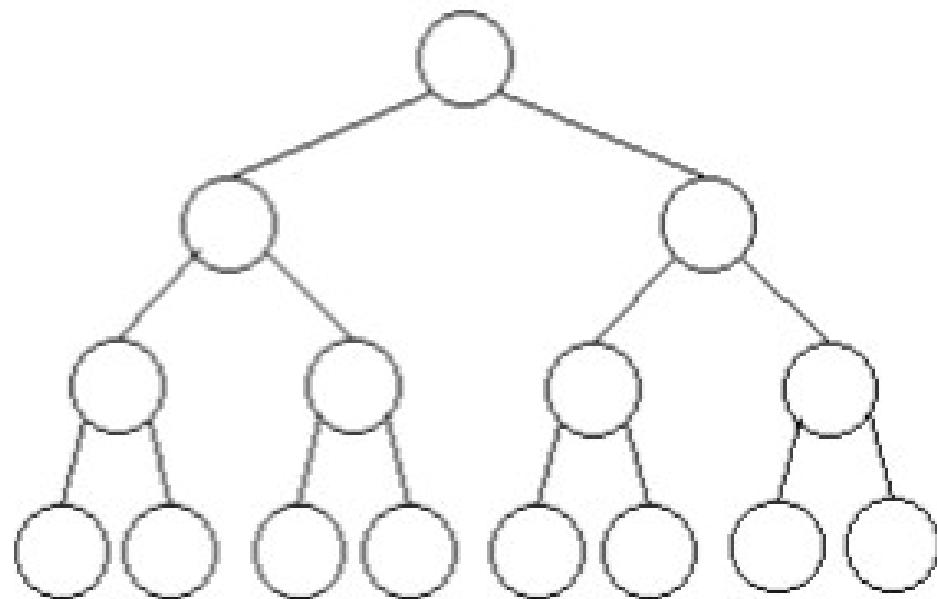
```
No *insere2(No *arv, No *n){  
    if (isEmpty()==1) //raiz nula, arvore vazia  
        raiz=n;  
    else{  
        if (arv!=NULL){  
            if (arv->nome<n->nome){  
                if (arv->right==NULL){  
                    arv->right=n;  
                }  
            }  
        }  
    }  
}
```

ATIVIDADE:

Continue a Implementação acima!

Propriedades

- O número máximo de nós possíveis no nível i é 2^{i-1} :



$$\text{nível } 1 = 2^{1-1} = 1 \text{ nó}$$

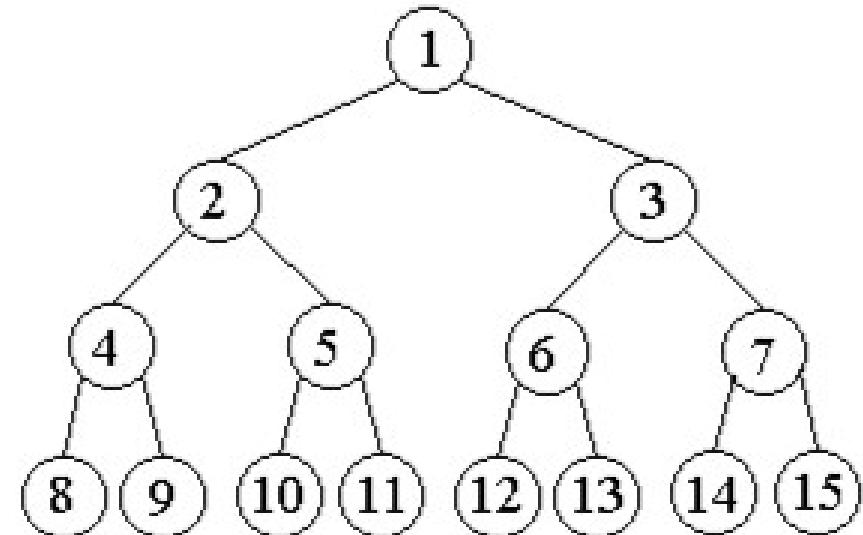
$$\text{nível } 2 = 2^{2-1} = 2 \text{ nós}$$

$$\text{nível } 3 = 2^{3-1} = 4 \text{ nós}$$

$$\text{nível } 4 = 2^{4-1} = 8 \text{ nós}$$

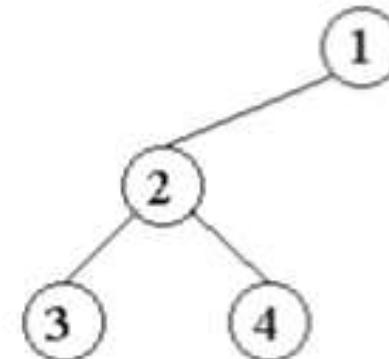
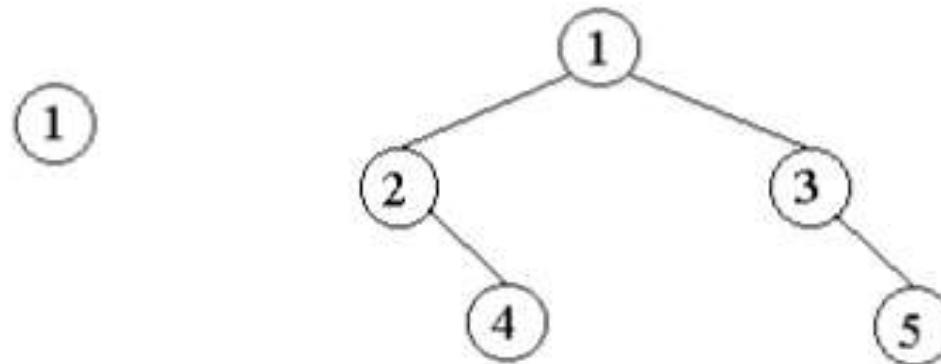
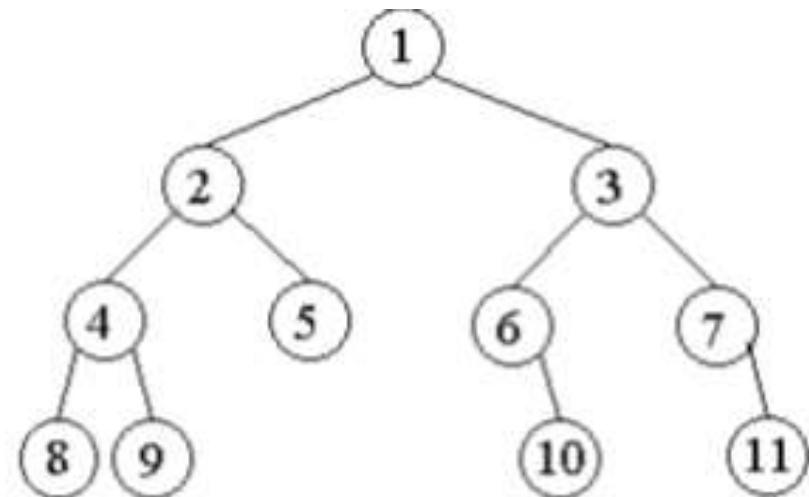
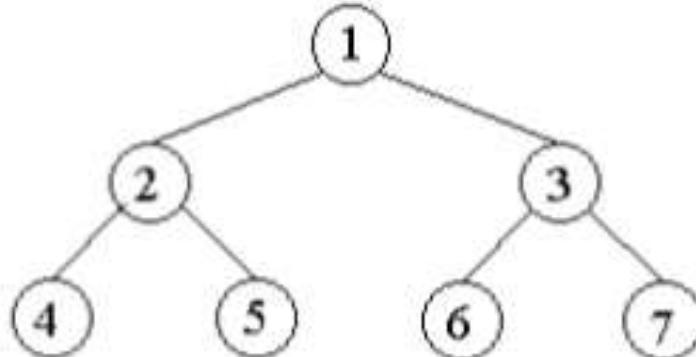
Propriedades

- Uma árvore binária de altura h tem no máximo $2^h - 1$ nós.
 - Ou considerando h começando de zero:
 - Uma árvore binária de altura h tem no máximo $2^{h+1} - 1$ nós.
 - E o mínimo de $h+1$ nós.



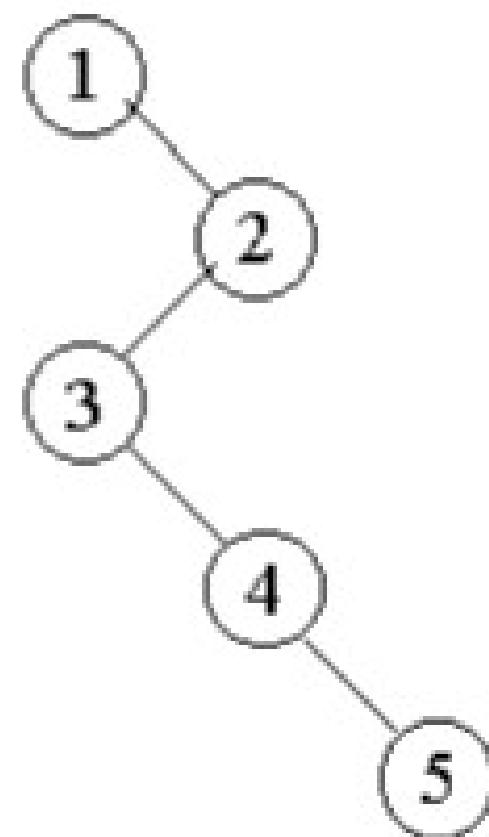
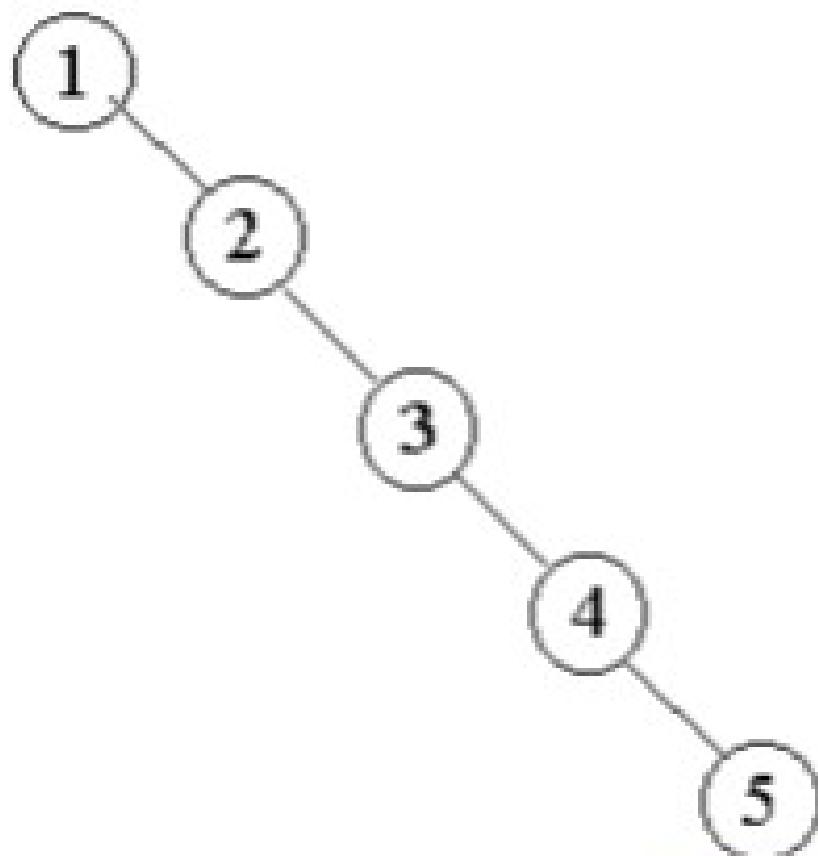
- A **altura mínima** de uma árvore binária com $n > 0$ nós é $1 + \lceil \log n \rceil$, com 1° $h=1$, e a $h=\lceil \log n \rceil$, considerando 1° . $h=0$ para $h = 4$, $n_{\text{máx}} = 2^4 - 1 = 15$

Uma árvore binária de altura mínima é dita completa.



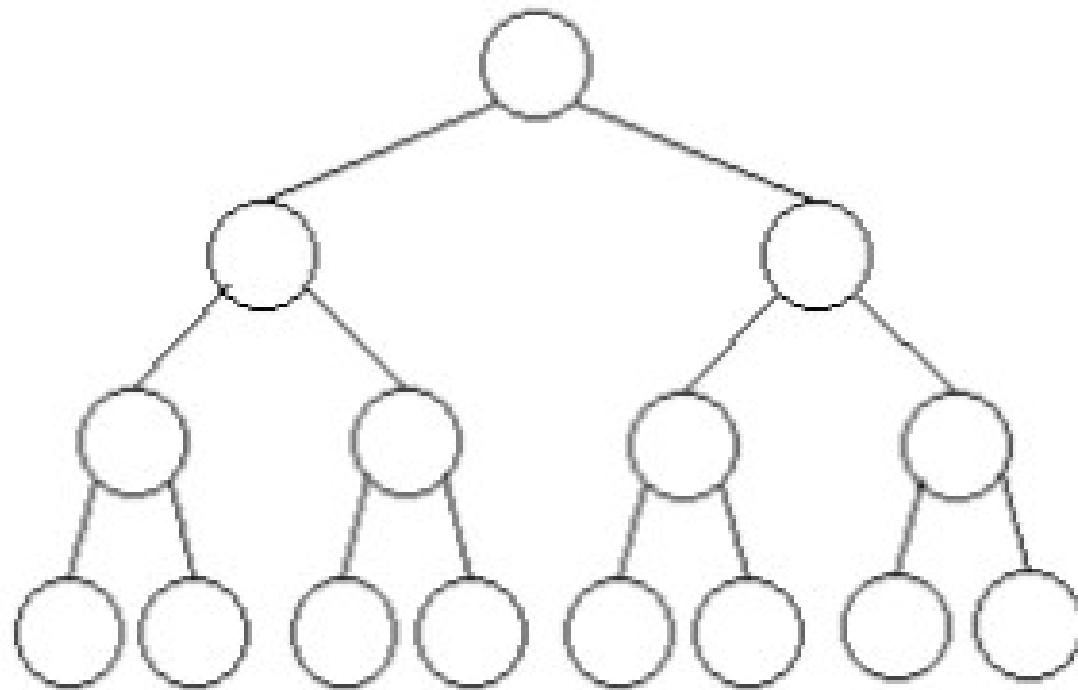
Exemplos de árvores completas. Seus nós não podem ser redistribuídos formando uma árvore de altura menor do que estas.

Uma árvore binária de altura máxima para uma quantidade de n nós é dita **assimétrica**. Neste caso, a altura é $h = n$ e seus nós interiores possuem exatamente uma subárvore vazia cada.



Exemplos de árvores assimétricas

Uma árvore binária de altura h é **cheia** se possui exatamente $2^h - 1$ nós.



Exemplo de árvore cheia – possuindo o número máximo de nós para a sua altura.

Atividade

- Para um determinada árvore binária calcule se ela é uma árvore completa ou não.
- Para isso :
 - calcule a quantidade de nós, verifique a altura mínima para essa quantidade de nós.
 - Calcule a altura da árvore.
 - Compare a altura mínima com a altura da árvore, se forem iguais, é uma árvore completa, se forem diferentes é uma árvore incompleta.