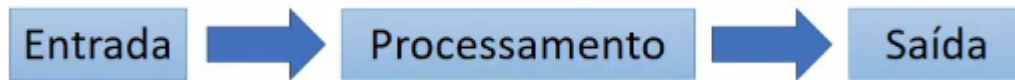


Algoritmos

Uma sequência ordenada de passos que deve ser seguida para a realização de uma tarefa;



Importante:

- Passos finitos;
- Sem redundância;
- Sem subjetividade;
- Deve ser claro e objetivo;

Formas de representação:

- Descrição Narrativa; - Usa Linguagem Natural
- Fluxograma; - Parte Gráfica, montamos graficamente os valores e funcionalidades
- Pseudocódigo; - é o que mais chega perto das linguagens de programação
- Variáveis; - Valores alocados na memória para fazer manipulação de dados

Descrição Narrativa

Vantagem: • O uso da linguagem natural;

Desvantagem: • Imprecisão; • Desvantagem;

Fluxograma

- Representação gráfica;
- Cada forma geométrica define uma função genérica;
- As formas geométricas são ligadas por flechas que indicam o fluxo da execução;

Pseudocódigo

- Assemelha-se a forma em que os programas são escritos;
 - Português estruturado;
 - A intenção é chegar na resolução do problema;
-

Componentes e elementos de Linguagem de Programação

Variáveis

Um local que armazena um conteúdo específico na memória principal do computador.

- Identificação única;
- Nomes para as variáveis significativos;
- Caracteres válidos: números, letras maiúsculas ou minúsculas, underline (_);
- Caracteres inválidos: "branco", caracteres especiais(@, \$, *, +, -, !, etc)
- O primeiro caractere de uma variável deve ser uma letra;
- Não pode usar palavras reservadas.

Exemplos Válidos:

Salario, idade, nome, nota1, X2, nome_aluno

Exemplos Inválidos:

Endereço, Nome 1, 1valor, Nota#

Tipos de Dados

Especifica as características, ou seja, os valores e operações possíveis de serem utilizadas com um dado desse tipo.

Tipo	Descrição
Inteiro	Representa valores inteiros. Ex: 18; 300; -100;
Real	Valores reais (decimal). Ex.: 5.5; 899.3; -22.22;
Caractere	Sequência de um ou mais caracteres. Ex.: Leo; A;
Lógico	Valores lógico: Verdadeiro, Falso.

Algoritmos

- Fazer um fluxograma que apresenta o cadastro de um aluno, com as seguintes informações:

- Nome;
 - Endereço;
 - Idade;
 - Apresentar as variáveis, e dar atenção aos seus tipos e características.
-

Linguagem C

- 1- Início do programa
- 2- Definição das variáveis
- 3- Instrução de leitura dos dados
- 4- Instrução do formato de escrita
- 5- Demais instruções e funções
- 6- Fim do programa.

Bibliotecas

As primeiras linhas de programação são definidas pelas bibliotecas.

- **stdio**: funções de entrada e saída;
- **stdlib**: transforma string em números;
- **string**: manipulação de string;
- **math**: operações matemáticas;

Exemplo: Exemplo: #include <stdio.h>

Função main()

Início da execução de um programa em C;

main()	int main ()	void main ()
{	{	{
}	}	}

Variáveis

Tipo	Declaração em C
Inteiro	int
Real	float
Caractere	char

```
Exemplo:  #include <stdlib.h>
           void main() {
             int valor1, valor2, soma;
           }
```

Palavras Reservadas

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Operações e Expressões em C

Operadores

Operadores	Função	Exemplo
+	Adição	y+x; 2+9;
-	Subtração	nota-extra; v-n;
*	Multiplicação	7*8; a*b; n*4;
/	Divisão	n1/n2; media/2;
%	Resto da divisão	15 % 2

Operador de atribuição

- Fornece valor a uma variável;
- Pode ser uma expressão;
- Símbolo: =

Exemplos:

```
a = 2;
a = b + c;
a = b * 2;
```

Operadores relacionais

Operador em Linguagem C	Operador em algoritmos	Descrição
>	>	Maior
<	<	Menor
>=	>=	Maior ou igual
<=	<=	Menor ou igual
==	=	Igual
!=	<>	Diferente

Operadores lógicos

Operador em Linguagem C	Operador em algoritmos	Operador em algoritmos
&&	E	Lógico E - conjunção
	Ou	Lógico OU - disjunção
!	não	Lógico NÃO - negação

Comando de saída de dados

Informações, mensagens e conteúdo de variáveis são enviadas para o usuário visualizá-las;

- printf (“expressão de controle”, listas de argumentos);

Código	Função
%c	Permite a escrita de apenas um caractere.
%d	Permite a escrita de números inteiros decimais.
%e	Realiza-se a escrita de números em notação científica.
%f	É feita a escrita de números reais (ponto flutuante).
%s	Efetua-se a escrita de uma série de caracteres.

printf (“O valor encontrado foi %d”, valor1);

Comando entrada de dados

As informação dos usuários são transferidas para variável do programa;

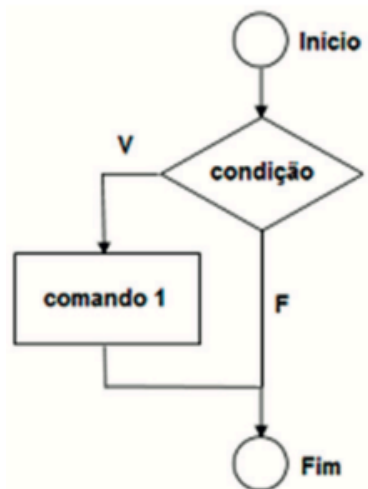
- scanf(“expressão de controle”, lista de argumentos); **scanf (“%d”, &valor);**

Exemplo: main() {
 int valor;
 printf("Digite um número: ");
 scanf("%d",&valor);
 printf("\n o número é %d",valor);
}

Estruturas de Decisão Condicional

IF

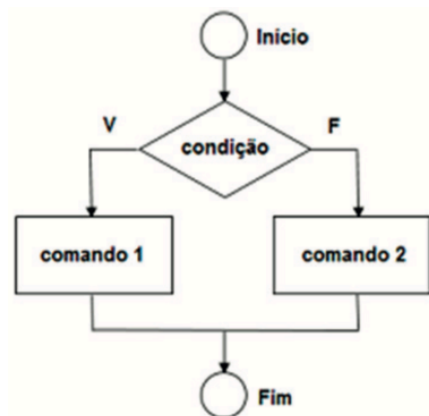
Tomar uma decisão e criar um desvio dentro do programa;



```
if (condição)
{
    Comandos;
}
```

```
if (num>0)
{
    printf ("0 numero e positivo");
}
```

IF e ELSE

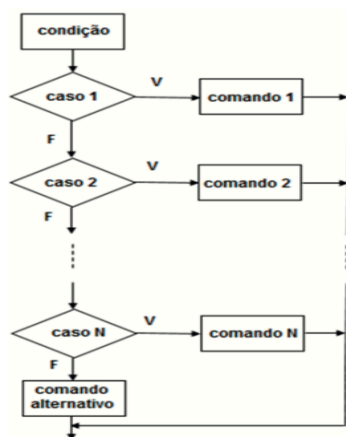


```
if (condição)
{
    comandos;
}
else
{
    comandos;
}
```

```
if (num>0)
{
    printf ("0 numero e positivo");
}
else
{
    printf ("0 numero e negativo");
}
```

Fonte: autor

Switch case



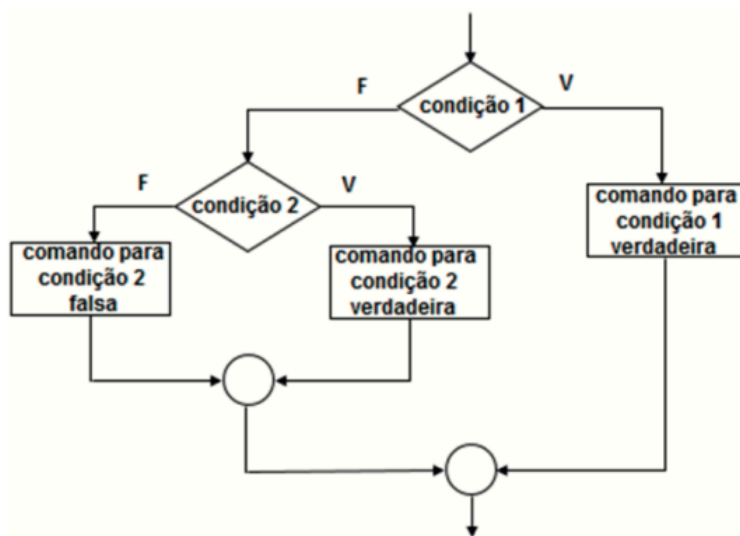
```
switch (variável) {
    case constante1:
        <comandos>
        break;
    case constante2:
        <comandos>
        break;
    default:
        default: <comandos>
}
```

```
switch ( valor )
{
    case 1 :
        printf ("Domingo\n");
        break;

    case 2 :
        printf ("Segunda\n");
        break;

    default :
        printf ("Valor invalido!\n");
}
```

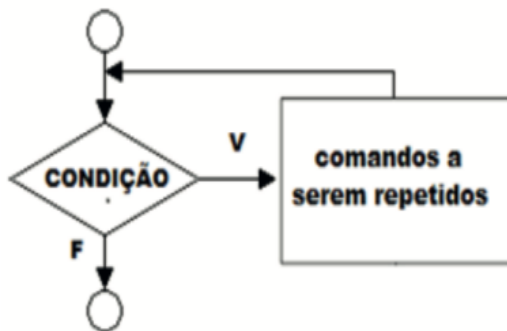
Estrutura condicional encadeada



Estruturas de Repetição Condicional

while

Comandos serão repetidamente executados enquanto uma condição verdadeira for verificada, somente após a sua negativa essa condição será interrompida.



```
while (condição)
{
    Comandos;
}
```

```
while(valor < 10)
{
    printf("valor = %d", valor);
    valor ++;
}
```

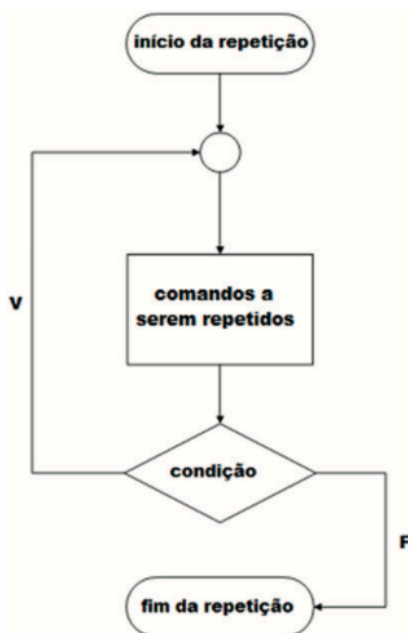
loop

Contador – é utilizado para controlar as repetições;

- **Incremento e decremento** – trabalham o número do contador, seja aumentando ou diminuindo.
- **Acumulador** – irá somar as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.
- **Condição de parada** – utilizada para determinar o momento de parar quando não se tem um valor exato desta repetição.

do-while

Analisa a condição ao final do laço, ou seja, os comandos são executados antes do teste de condição.



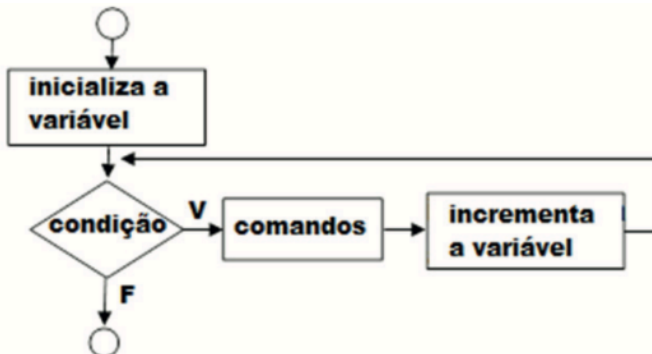
```
do
{
    comandos;
} while (condição);
```

```
do
{
    printf("valor = %d", valor);
    valor ++;
} while (valor < 10);
```

Fonte: autor

for

Repetir uma informação por um número fixo de vezes



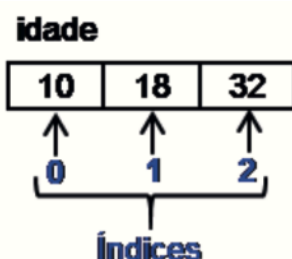
```
for (inicialização; condição final; incremento)
{
    comandos;
}
```

```
for (int x=0; x < 10; x++)
{
    printf("%d ", x);
}
```

Vetores e Matrizes

Vetores

- Tipo especial de variável;
- Armazena diversos valores "ao mesmo tempo", usando um mesmo endereço na memória;
- Sintaxe: tipo variável [n]



Valor de idade → Depende do índice

idade[0] = 10
idade[1] = 18
idade[2] = 32

Matrizes

Arranjos de duas ou mais dimensões. Assim como nos vetores, todos os elementos de uma matriz são do mesmo tipo, armazenando informações semanticamente semelhantes.

Sintaxe: tipo variável [m][n]

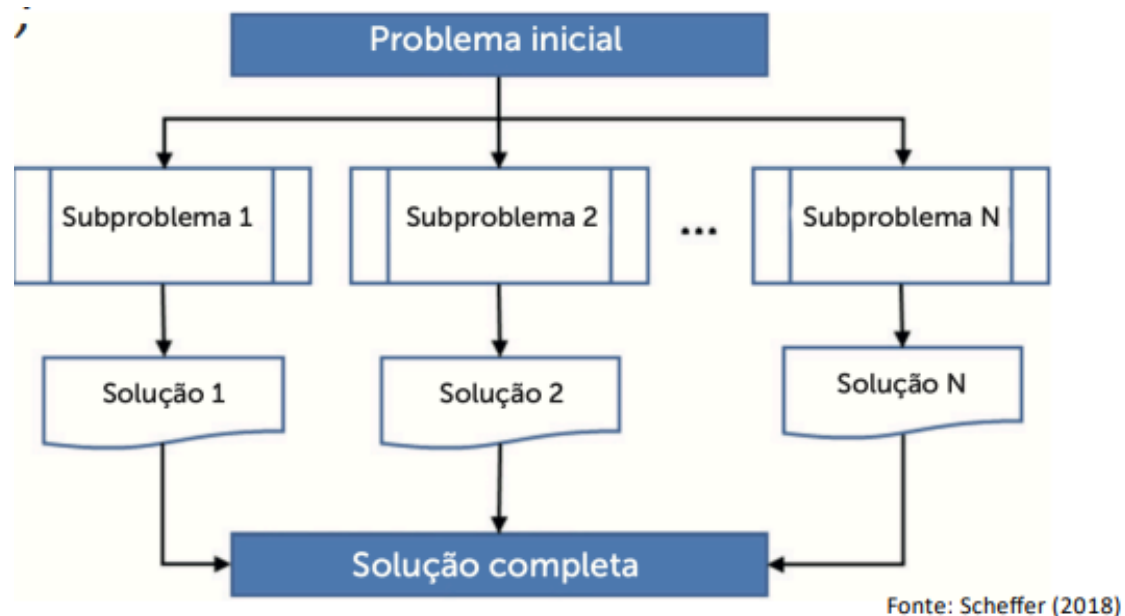
		Nota 1	Nota 2
		coluna 0	coluna 1
Aluno 1	linha 0 →	10,0	8,5
Aluno 2	linha 1 →	5,5	2,7
Aluno 3	linha 2 →	4,0	10,0

Variáveis compostas

- Em qualquer variável composta, o índice começa por zero, então, em uma matriz, o primeiro espaço para armazenamento é sempre (0,0), ou seja, índice 0 tanto para linha como para coluna.
- Não é obrigatório que todas as posições sejam ocupadas, sendo possível declarar uma matriz com 10 linhas (ou colunas) e usar somente uma.

Funções

A ideia de criar programas com blocos de funcionalidades vem de uma técnica de projeto de algoritmos chamada dividir para conquistar;



Função é um trecho de código escrito para solucionar um subproblema;

- Dividir a complexidade de um problema maior;
- Evitar repetição de código;
- Modularização;


```

<tipo de retorno> <nome> (<parâmetros>)
{
    <Comandos da função>
    <Retorno> (não obrigatório)
}

```

<tipo de retorno> – Obrigatório. Esse parâmetro indica qual o tipo de valor a função irá retornar. Pode ser um valor inteiro (int), decimal (float ou double), caractere (char), etc

<nome> – Obrigatório. Parâmetro que especifica o nome que identificará a função

<parâmetros> – Opcional.

<retorno> – Quando o tipo de retorno for void esse parâmetro não precisa ser usado, porém, quando não for void é obrigatório.

Funções com Ponteiros

Ponteiro

Tipo especial de variável, que armazena um endereço de memória;

O acesso à memória é feito usando dois operadores:

- Asterisco (*): usado para criação do ponteiro;
- “&” : Acessar o endereço da memória;

<tipo> *; Exemplo: int *idade;

A criação de um ponteiro só faz sentido se for associada a algum endereço de memória.

1. int ano = 2018;

2. int *ponteiro_para_ano = &ano;

Função com ponteiros

Função que retorna um vetor:

int[10] calcular() **ERRADO!**

O correto para este caso é utilizar ponteiros. tipo* nome() {

Funções com ponteiros

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

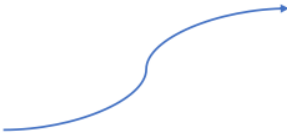
```
int* gerarRandomico() {
```

```
    static int r[10];
    int a;
```

```
    for(a = 0; a < 10; ++a) {
        r[a] = rand();
        printf("r[%d] = %d\n", a, r[a]);
    }
```

```
    return r;
```

```
}
```



```

r[0] = 41
r[1] = 18467
r[2] = 6334
r[3] = 26500
r[4] = 19169
r[5] = 15724
r[6] = 11478
r[7] = 29358
r[8] = 26962
r[9] = 24464

```

Função malloc()

- Alocar memória dinamicamente;
- Exemplo: `int* memoria = malloc (100);`
- Retorna dois valores: NULL ou um ponteiro genérico;

Escopo e Passagem de Parâmetros

Escopo

```
int testar(){
    int x = 10;
    return x;
}
int main(){
    int x = 20;
    printf("\n Valor de x na funcao main()
%d",x);
    printf("\n Valor de x na funcao testar()
%d",testar());

    return 0;
}
```

Variável Local: são “enxergadas” somente dentro do corpo da função onde foram definidas;

Variável Global: criá-la fora da função, assim ela será visível por todas as funções do programa. Geralmente adota-se declará-las após as bibliotecas.

```
#include<stdio.h>

int x = 10;

int testar(){
    return 2*x;
}
int main(){
    printf("\n Valor de x global = %d",x);
    printf("\n Valor de x global alterado na funcao
testar() = %d",testar());

    return 0;
}
```

Parâmetros

```
<tipo de retorno> <nome> (<parâmetros>)
{
    <Comandos da função>
    <Retorno> (não obrigatório)
}
```

Passagem de Valor: a função cria variáveis locais automaticamente para armazenar esses valores e após a execução da função essas variáveis são liberadas.

Passagem por referência

- Ponteiro e endereço de memória;
- Não será criada uma cópia dos argumentos passados;
- Será passado o endereço da variável e função trabalhará diretamente com os valores ali armazenados;

Recursividade

Função Recursiva

A função será invocada dentro de dela mesma.

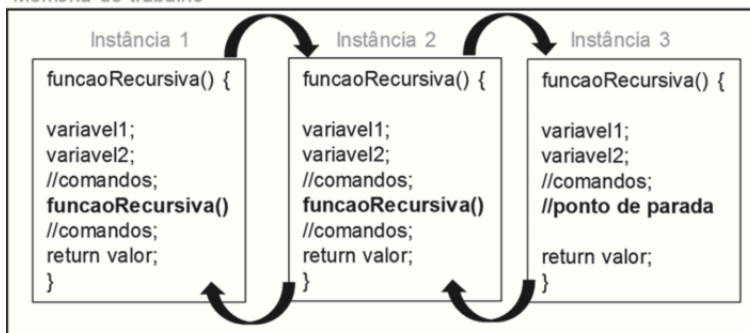
```
<tipo> funcaoRecursiva(){
    //comandos
    funcaoRecursiva(); ← Chamando a si próprio
    //comandos
}

void main(){
    //comandos
    funcaoRecursiva(); 1
    //comandos
}
```

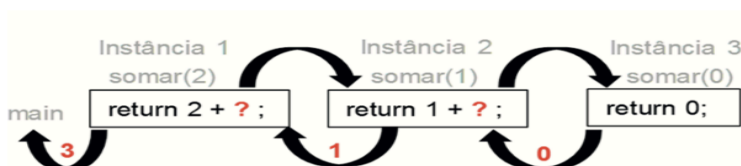
Um algoritmo recursivo resolve um problema dividindo-o em subproblemas mais simples, cujo a solução é a aplicação dele mesmo.

- Ponto de Parada;
- Variáveis na memória de trabalha;
- As variáveis são independentes;

Memória de trabalho



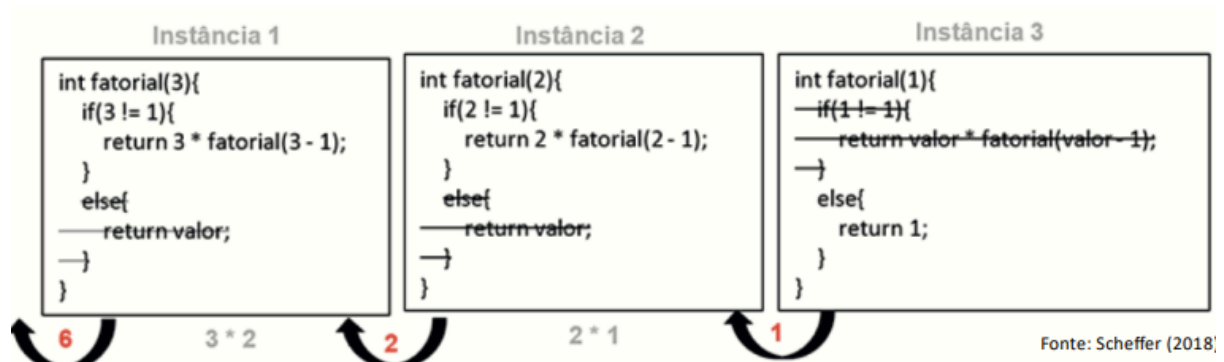
```
int somar(int valor) {
    if(valor != 0) {
        return valor + somar(valor-1);
    }
    else{
        return valor;
    }
}
```



Fatorial

O fatorial de um número qualquer N consiste em multiplicações sucessivas até que N seja igual ao valor unitário, ou seja, $5! = 5 \times 4 \times 3 \times 2 \times 1$, que resulta em 120.

```
int fatorial(int valor){
    if(valor != 1){
        return valor * fatorial(valor - 1);
    }
    else{
        return valor;
    }
}
```



Fibonacci

A sequência de Fibonacci:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Encontrar o n-ésimo elemento da sequência.

Exemplos:

```
int fibonacci(int n) {
    if (n == 0)
        return 0;
    else {
        if (n == 1)
            return 1;
        else
            return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

```
int Fib(int n)
{
    int i, j, f;
    i = 1; f = 0;
    for (j = 1; j <= n; j++) {
        f += i;
        i = f - i; }
    return f;
}
```

Recursividade

Foi solicitado a você implementar o método de Newton para o cálculo da raiz quadrada, porém, usando funções recursivas.

$$x_n = \frac{x_{n-1}^2 + n}{2x_{n-1}}$$

Você precisa solicitar ao usuário um número; Você também deve especificar um valor inicial para a raiz e um critério de parada;

```
#include<stdio.h>
#include<math.h>
float calcularRaiz(float n, float raizAnt)
{
    float raiz = (pow(raizAnt, 2) + n)/(2 * raizAnt);
    if (fabs(raiz - raizAnt) < 0.001)
        return raiz;
    return CalcularRaiz(n, raiz);
}

void main(){
    float numero, raiz;
    printf("\n Digite um número para calcular a raiz: ");
    scanf("%f",&numero);
    raiz = calcularRaiz(numero,numero/2);
    printf("\n Raiz quadrada funcao = %f",raiz);
}
```

Listas

Struct

Variável que armazena valores de tipos diferentes;

```
#include<stdio.h>

struct automovel{
    char modelo[20];
    int ano;
    float valor;
};

main(){
    struct automovel dadosAutomovel1;
}
```

Lista Ligada

- Estrutura de dados linear e dinâmica;
- Cada elemento da sequência é armazenado em uma célula da lista;



```
struct lista {  
    int info;  
    struct lista* prox;  
};  
typedef struct lista Lista;
```

```
struct alunos {  
    char nome[25];  
    struct alunos* prox;  
};  
typedef struct alunos Classe;
```

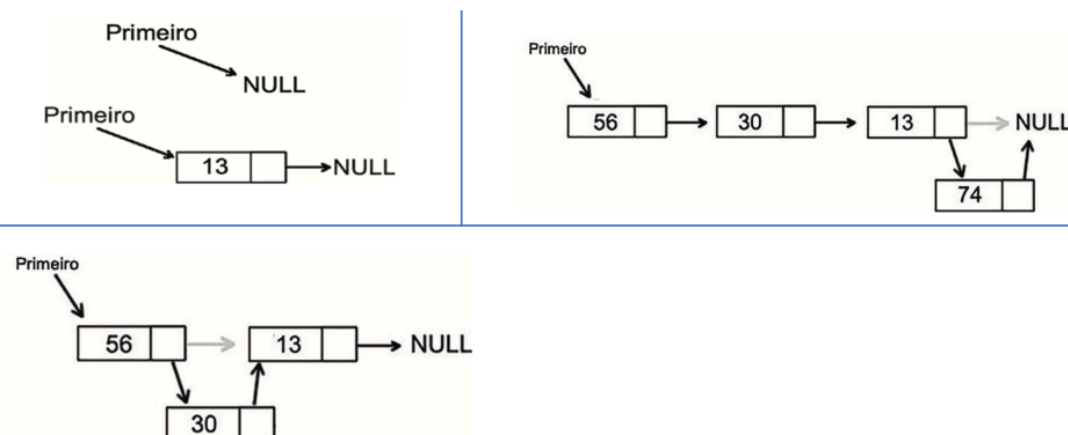
- Criação ou definição da estrutura de uma lista.
- Inicialização da lista.
- Inserção com base em um endereço como referência.
- Alocação de um endereço de nó para inserção na lista.
- Remoção do nó com base em um endereço como referência.
- Deslocamento do nó removido da lista.

Operações com Listas Ligadas - Inserção

Adicionar elementos na lista

- Para inserirmos um elemento na lista ligada, é necessário alocarmos o espaço na memória;
- Atualizar o valor do ponteiro; Posição do inserção
- Final da lista;
- Primeira posição;
- No meio da lista;

Adicionar elementos na lista



Adicionar elementos na lista

```
Lista* inserir (Lista* l, int i) {  
    Lista* novo = (Lista*) malloc(sizeof(Lista));  
    novo -> info = i;  
    novo -> prox = l;  
    return novo;  
}
```

Adicionar elementos na lista

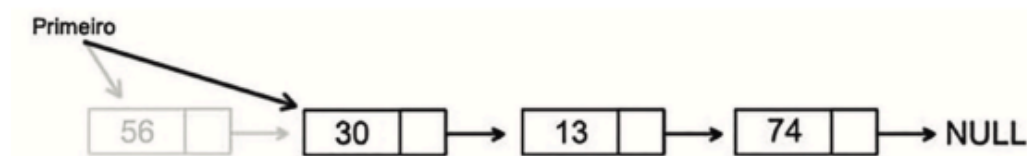
<pre>Lista* inserirPosicao(Lista* l, int pos, int v){ int cont = 1; Lista *p = l; Lista* novo = (Lista*)malloc(sizeof(Lista)); while (cont != pos){ p = p -> prox; cont++; }</pre>	<pre>} novo -> info = v; novo -> prox = p -> prox; p -> prox = novo; return l; }</pre>
---	--

```
Lista* inserirFim(Lista* l, int v){  
    Lista *p = l;  
    Lista* novo = (Lista*)malloc(sizeof(Lista));  
    while (p -> prox != NULL){  
        p = p -> prox;  
        cont++;  
    }  
    novo -> info = v;  
    novo -> prox = p -> prox;  
    p -> prox = novo;  
    return l;  
}
```

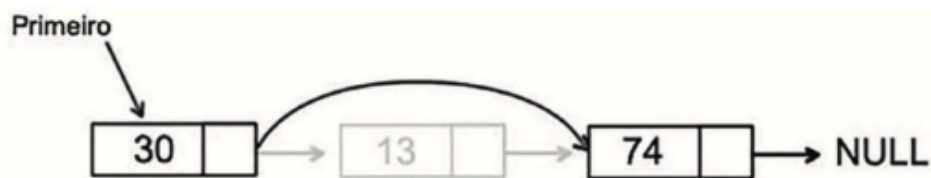
Operações com Listas Ligadas

Remover Elementos da Lista

Primeiro elemento da lista:



Elemento no meio da lista:



```
Lista* remove (Lista* l, int v) {  
    Lista* anterior = NULL;  
    Lista* p = l;  
    while (p != NULL && p -> info != v) {  
        anterior = p;  
        p = p -> prox;  
    }  
    if (p == NULL )  
        return l;  
    if (anterior == NULL) {  
        l = p -> prox;  
    } else {  
        anterior -> prox = p -> prox;  
    }  
    return l;  
}
```


Outras operações na lista Ligada

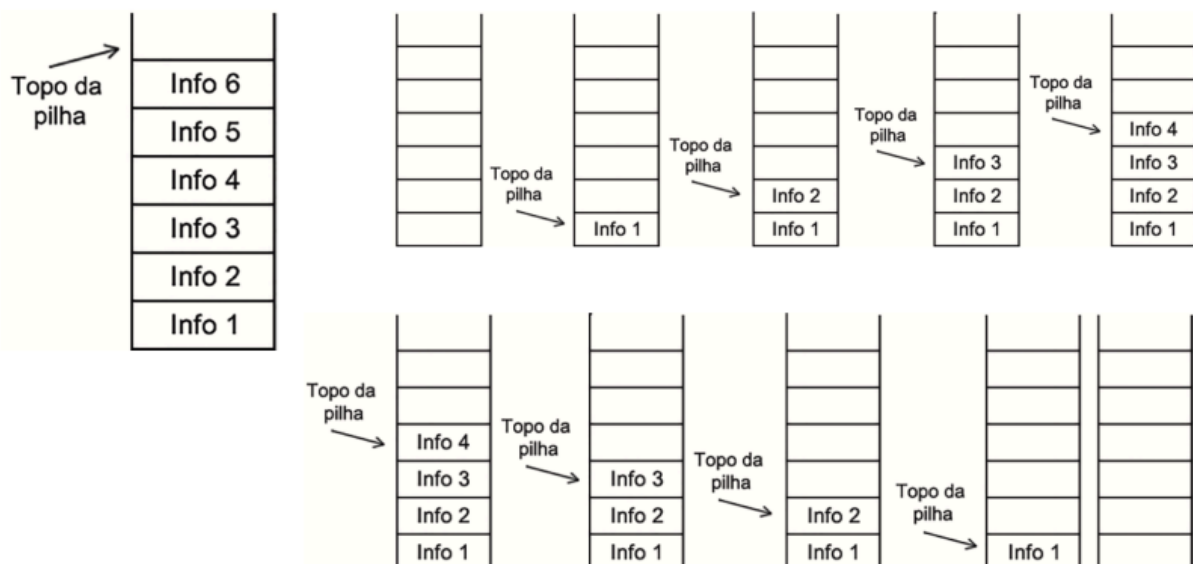
- Percorrer a lista ligada;
- Saber quais elementos fazem parte da estrutura de dados;
- Verificar se um elemento se encontra na lista ligada

Outras operações na lista Ligada

```
void imprimir (Lista* l) {  
    Lista* p;  
    printf("Elementos:\n");  
    for (p = l; p != NULL; p = p -> prox) {  
        printf(" %d -> ", p -> info);  
    }  
}  
  
Lista* buscar(Lista* l, int v){  
    Lista* p;  
    for (p = l; p != NULL; p = p -> prox) {  
        if (p -> info == v)  
            return p;  
    }  
    return NULL;  
}
```

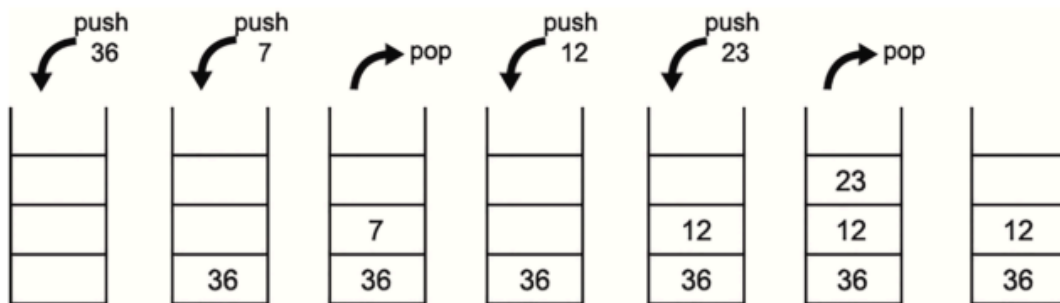
Pilha

São estruturas de dados do tipo LIFO (last-in first-out), onde o último elemento a ser inserido, será o primeiro a ser retirado.



Duas operações básicas:

- Empilhar um elemento (**push()**)
- Desempilhar um elemento (**pop()**)



Pilha

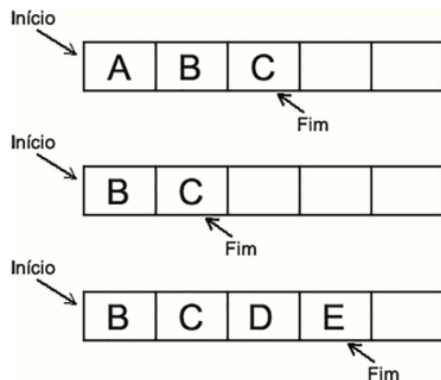
```
struct Pilha {  
    int topo;  
    int capacidade;  
    float * proxElem;  
};  
  
void cria_pilha(struct Pilha *p, int c ){  
    p -> proxElem = (float*) malloc (c * sizeof(float));  
    p -> topo = -1;  
    p -> capacidade = c;  
}  
  
struct Pilha minhaPilha;  
  
void push_pilha(struct Pilha *p, float v){  
    p -> topo++;  
    p -> proxElem [p -> topo] = v;  
}  
  
float pop_pilha (struct Pilha *p){  
    float aux = p -> proxElem [p -> topo];  
    p -> topo--;  
    return aux;  
}
```

Fila

São estruturas de dados do tipo FIFO (first-in first-out), onde o primeiro elemento a ser inserido, será o primeiro a ser retirado, ou seja, adiciona-se itens no fim e remove-se do início.

Passos para a criação de uma Fila:

- criar uma fila vazia;
- inserir elemento no final;
- retirar um elemento do início;
- verificar se a fila está vazia;



Fila

```
#define N 100
```

```
struct fila {
```

```
    int n;
```

```
    int ini;
```

```
    char vet[N];
```

```
};
```

```
void insere_fila (Fila* f, char elem){
```

```
    int fim;
```

```
    if (f -> n == N){
```

```
        printf("A fila está cheia.\n");
```

```
float remove_fila (Fila* f){
```

```
    char elem;
```

```
    if (fila_vazia(f)){
```

```
        printf("A Fila esta vazia\n");
```

```
        exit(1);
```

```
    }
```

```
    elem = f -> vet[f -> ini];
```

```
    f -> ini = (f -> ini + 1) % N;
```

```
    f -> n--;
```

```
    return elem;
```

```
}
```

```
Fila* inicia_fila (void){
```

```
    Fila* f = (Fila*) malloc(sizeof(Fila));
```

```
    f -> n = 0;
```

```
    f -> ini = 0;
```

```
    return f;
```

```
}
```