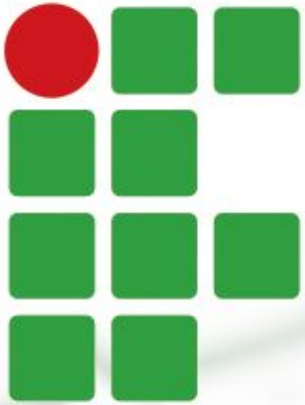


INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Estruturas de Dados I

- Alocação Dinâmica -

Alocação Estática x Dinâmica

- Observe o trecho de código a seguir...

```
typedef struct{
     int matricula;
     char nome[100];
}Aluno;

int main{
   Aluno turma[30];
}
```

Alocação Estática x Dinâmica

- Observe o trecho de código a seguir...

```
typedef struct{
     int matricula;
     char nome[100];
}Aluno;

int main{
   Aluno turma[30];
}
```

- **Quantos alunos** esta aplicação conseguirá gerir?
- Mas… e se for preciso **aumentar a turma** depois que o programa foi distribuído???

Alocação Estática x Dinâmica

- Observe o trecho de código a seguir...

```
typedef struct{
     int matricula;
     char nome[100];
}Aluno;

int main{
   Aluno turma[30];
}
```

Pode não fazer muito sentido um código assim para um sistema acadêmico, mas imagine outras situações reais do dia a dia...

- Clientes acessando um site para compra de ingressos.
- Lista de documentos aguardando impressão.
- Leituras de um sensor a cada alteração detectada.

- **Quantos alunos** esta a
- Mas... e se for precis
programa foi distribuído???

Alocação Estática x Dinâmica

- A alocação da estrutura de dados no exemplo anterior (*Array* de Alunos) foi realizada de forma **estática**.
- A **alocação estática de memória** acontece **uma única vez**, durante a criação do processo, não sendo possível alterar essas estruturas ***durante*** a execução (**em tempo de execução**).
- Entretanto, existem inúmeras situações em que a quantidade exata de dados (e memória consumida) **só pode ser conhecida *durante* a execução da aplicação.**

Alocação Estática x Dinâmica

- **É uma Possível solução???**

```
typedef struct{
     int matricula;
     char nome[100];
}Aluno;

int main{
   int n;
   printf("Digite a Qtde. de Alunos na Turma: ");
   scanf(" %d", &n);
   Aluno turma[n];
}
```

Alocação Estática x Dinâmica

- **É uma Possível solução???**

NÃO!

Esta técnica somente "*mascara*" o problema...

... e se o valor "N" não for suficiente?
... e se o valor "N" for muito exagerado?

Alocação Estática x Dinâmica

- **É uma Possível solução???**

NÃO!

Esta técnica somente "*mascara*" o problema...

... e se o valor "N" não for suficiente?

... e se o valor "N" for muito exagerado?

Essa "solução" não é satisfatória em termos de desempenho e performance!

Alocação Estática x Dinâmica

- **Alocação Dinâmica** é a técnica que permite alocar (reservar) a memória em ***tempo de execução***.
- Isso significa que o espaço de memória para armazenamento de dados **é reservado sob demanda**, **durante a execução da aplicação**.
- Útil nas situações onde não se sabe exatamente quantas variáveis/estruturas serão necessárias para o armazenamento de todas as informações.

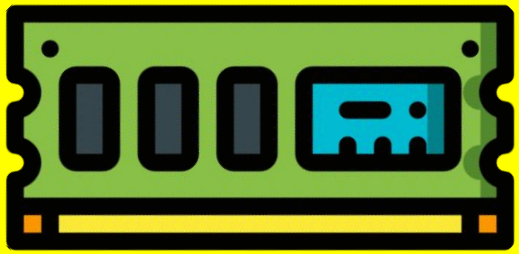
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Alocação Estática x Dinâmica

- **Alocação Dinâmica** é a técnica que permite alocar (reservar) a memória em ***tempo de execução***.

Fica evidente a **melhor utilização e economia** de um dos recursos computacionais mais importantes:

A MEMÓRIA PRINCIPAL

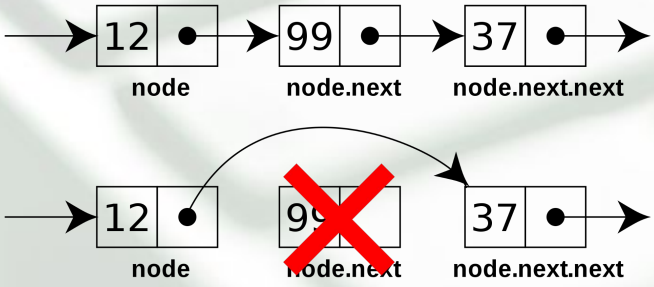


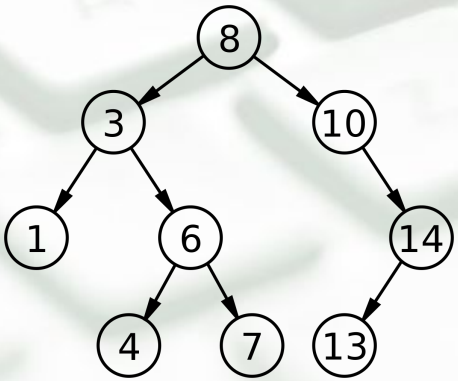
o armazenamento de todas as informações.

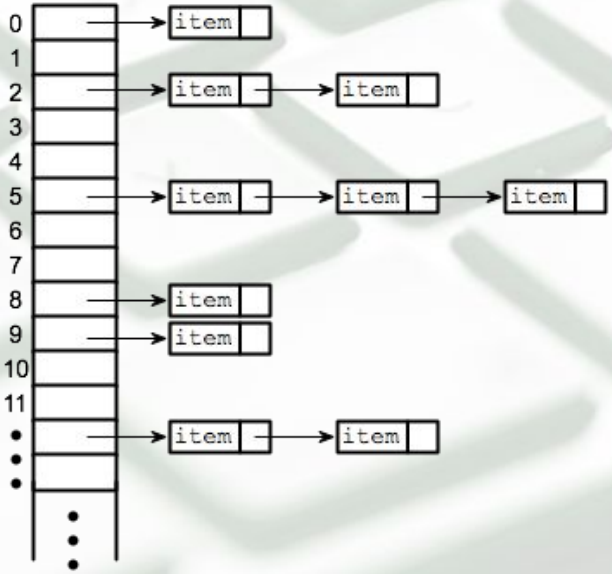
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Alocação Dinâmica

- A **Alocação Dinâmica** é uma técnica utilizada em **diversas estruturas de dados** e aplicações, p.ex.:
  - Listas encadeadas e generalizações.
  - Estruturas de filas e pilhas.
  - Árvores binárias e grafos.
  - ...







INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Alocação Dinâmica

- A **Alocação Dinâmica** acontece por meio de duas funções principais:
  - `malloc` *(**M**emory **ALLOC**ation)*
  - `free`
- Ambas funções, pertencem à biblioteca:

`<stdlib.h>`

Função de Alocação

```
void* malloc(int tamanho)
```

- ***Memory Allocation***
  - A função recebe como parâmetro o número de *bytes* (tamanho) que se deseja alocar na memória.
  - O retorno da função é um **ponteiro do tipo `void`**.

Função de Alocação

```
void* malloc(int tamanho)
```

- ***Memory Allocation***
  - A função recebe como parâmetro o número de *bytes* (tamanho) que se deseja alocar na memória.
  - O retorno da função é um **ponteiro do tipo `void`**.

Ponteiro do tipo `void` ???

- A vantagem do **ponteiro `void`** é que ele pode ser **convertido** para qualquer outro tipo de ponteiro, através da técnica de ***typecast***.

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Função de Alocação

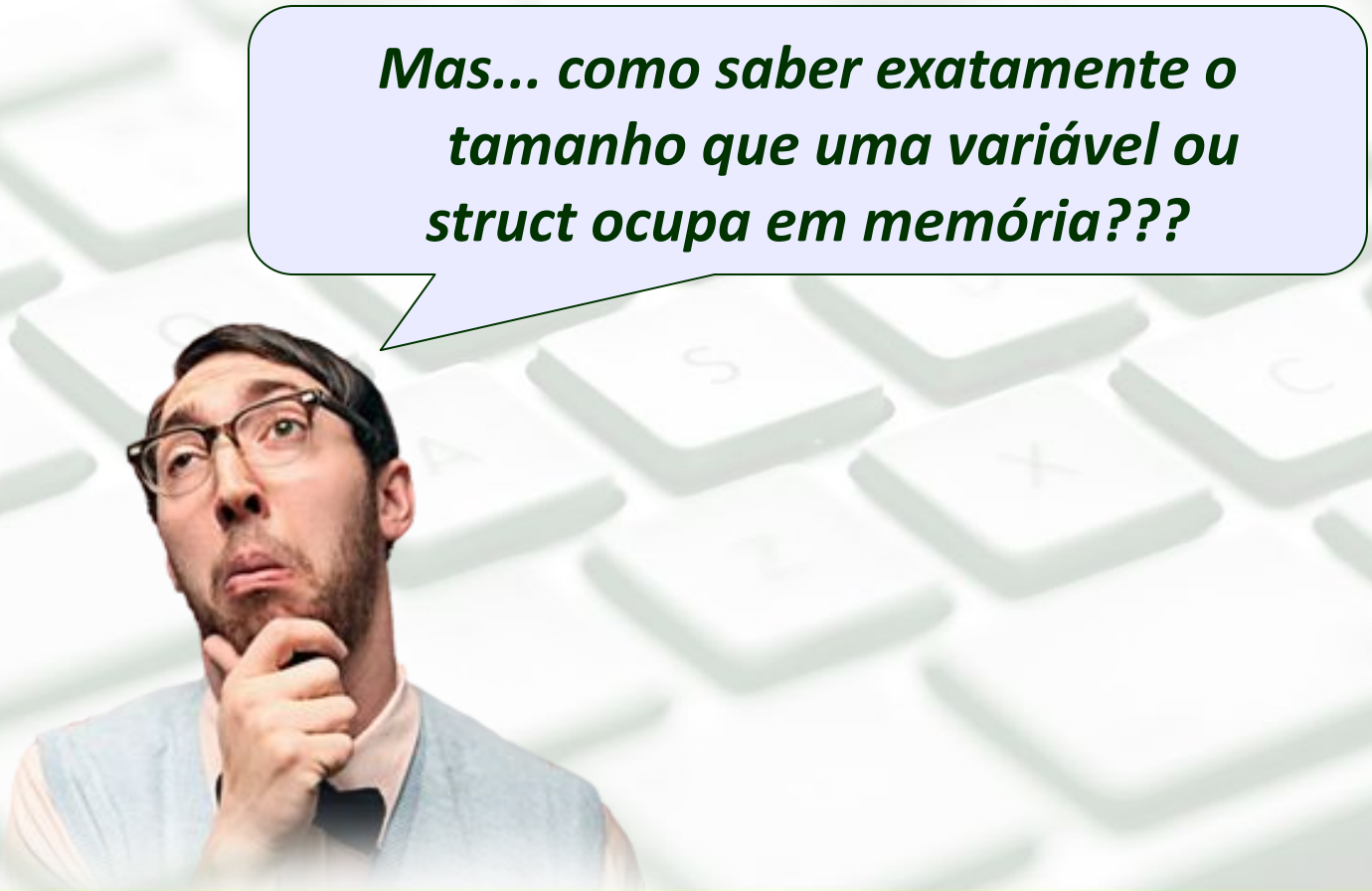
```
#include<stdio.h>
#include<stdlib.h>

int main(){
    int* x;
    x = malloc(4);
    scanf(" %d", x);
    printf("%p -> %d",x,*x);
}
```

Uma variável int possui 4 bytes (Arquit. x64)

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Função de Alocação

```
void* malloc(int tamanho)
```



Função de Alocação

```
#include<stdlib.h>

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a;
  a = malloc(¿¿¿???);
}
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Função `sizeof`

```
int sizeof(type);
```

- A função **`sizeof`** recebe como parâmetro um *tipo de dados* e retorna a **quantidade de bytes que esta estrutura ocupa em memória**.

Função de Alocação

```
#include<stdlib.h>

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a;
  a = malloc(sizeof(Aluno));
}
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Ponteiro para *Struct*

- Como acessar os atributos de um **ponteiro de *struct***

```
int main(){
  Aluno *a = malloc(sizeof(Aluno));
}
```

- Opção menos usual...

```
(*a).matricula
```

- Opção predominante, **uso do operador** `->`

```
a->matricula
```

Exemplo de Código

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

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
  Aluno *a  = malloc(sizeof(Aluno));
  scanf(" %d", &a->matricula);
  scanf(" %[^\n]s", a->nome);
  printf("Matricula: %d\n", a->matricula);
  printf("Nome: %s\n", a->nome);
}
```

Função de Liberação

```
void free(void *p)
```

- A função **`free`** é utilizada para liberar o espaço de memória alocado para um ponteiro ***p*** qualquer.
- É **recomendável** a utilização da função **`free`** sempre que o espaço de memória alocado para uma estrutura não for mais necessário, para evitar erros inesperados, e para economia de recursos do sistema.

Exemplo de Código

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

typedef struct{
    int matricula;
    char nome[100];
}Aluno;

int main(){
   Aluno *a  = malloc(sizeof(Aluno));
   scanf(" %d", &a->matricula);
   scanf(" %[^\n]s", a->nome);
   printf("Matricula: %d\n", a->matricula);
   printf("Nome: %s\n", a->nome);
   free(a);
}
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR!!!

- Defina um novo tipo **Documento** para armazenar os dados de um arquivo enviado para impressão (nome do arquivo, número de páginas, prioridade).
- Declare um **ponteiro** (*não uma variável*) do tipo **Documento**.
- Faça uma função que (utilizando **Alocação Dinâmica)** cria uma tarefa de impressão para um Documento, e outra Função para simular a impressão desse documento, tal como nos protótipos a seguir…

```
Documento* cria_Documento("doc.pdf",5,1);
void imprime_Documento(doc);
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR!!!

Esta implementação está correta?

para armazenar os dados
essão (nome do arquivo,

```
Documento* cria_Documento(char nome[],int pgs,int prio){
    Documento novo;
    strcpy(novo.nome,nome);
    novo.paginas = pags;
    novo.prioridade = prio;
    return &novo;
}
```

```
Documento* cria_Documento("doc.pdf",5,1);
void imprime_Documento(doc);
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR!!!

Esta implementação está correta?

para armazenar os dados
essão (nome do arquivo,

```
Documento* cria_Documento(char nome[],int pgs,int prio){
    Documento novo;
    strcpy(novo.nome,nome);
    novo.paginas = pags;
    novo.prioridade = prio;
    return &novo;
}
```

ATENÇÃO

VARIÁVEIS CUJA ALOCAÇÃO É REALIZADA DE FORMA ESTÁTICA SÃO ARMAZENADAS EM UMA REGIÃO DE MEMÓRIA (*STACK*) QUE DEIXA DE EXISTIR QUANDO ESSA FUNÇÃO É FINALIZADA.

```
Documento* cria_Documento("doc.pdf",5,1);
void imprime_Documento(doc);
```

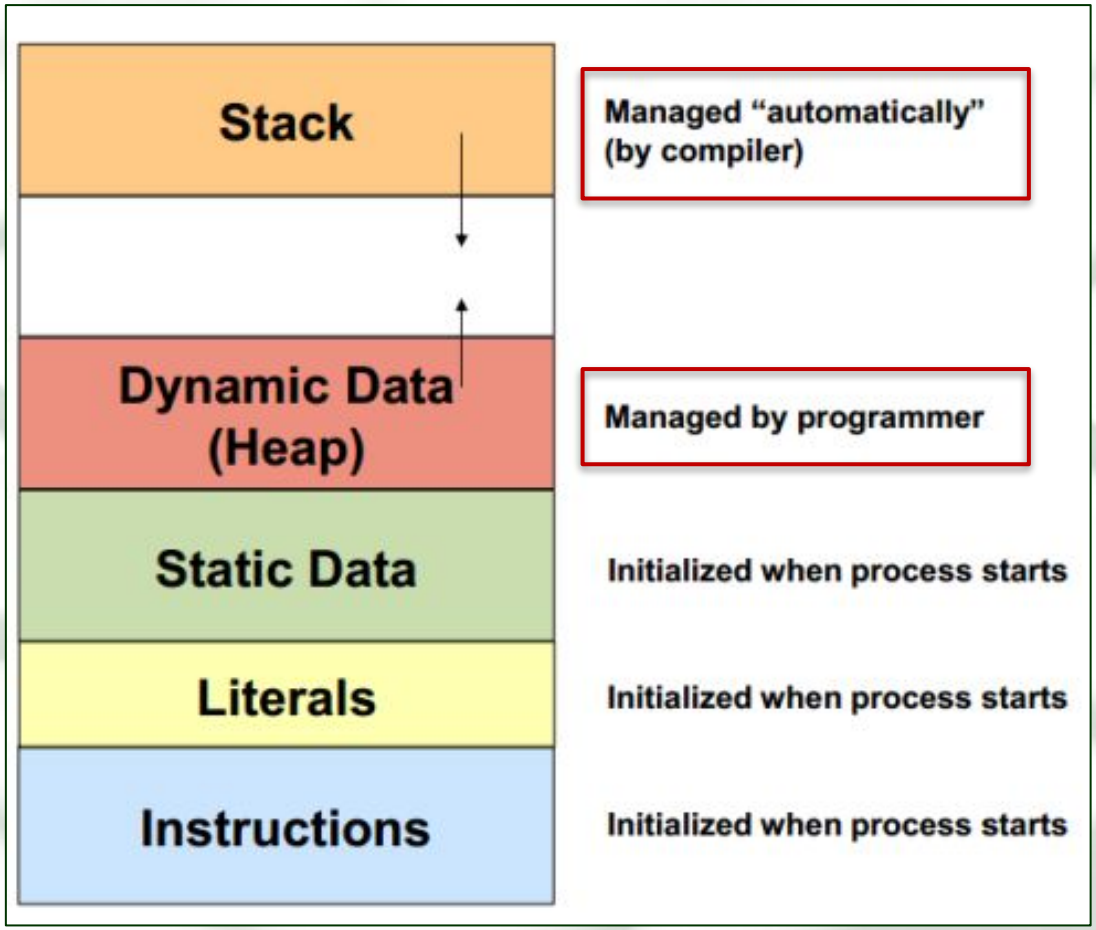
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

Regiões de Memória de um Processo

Estruturas/Variáveis criadas por meio de **Alocação Dinâmica** são armazenadas em uma região específica do processo, chamada **HEAP**

Estruturas/Variáveis de **Alocação Estática** são armazenadas na região denominada **STACK**



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

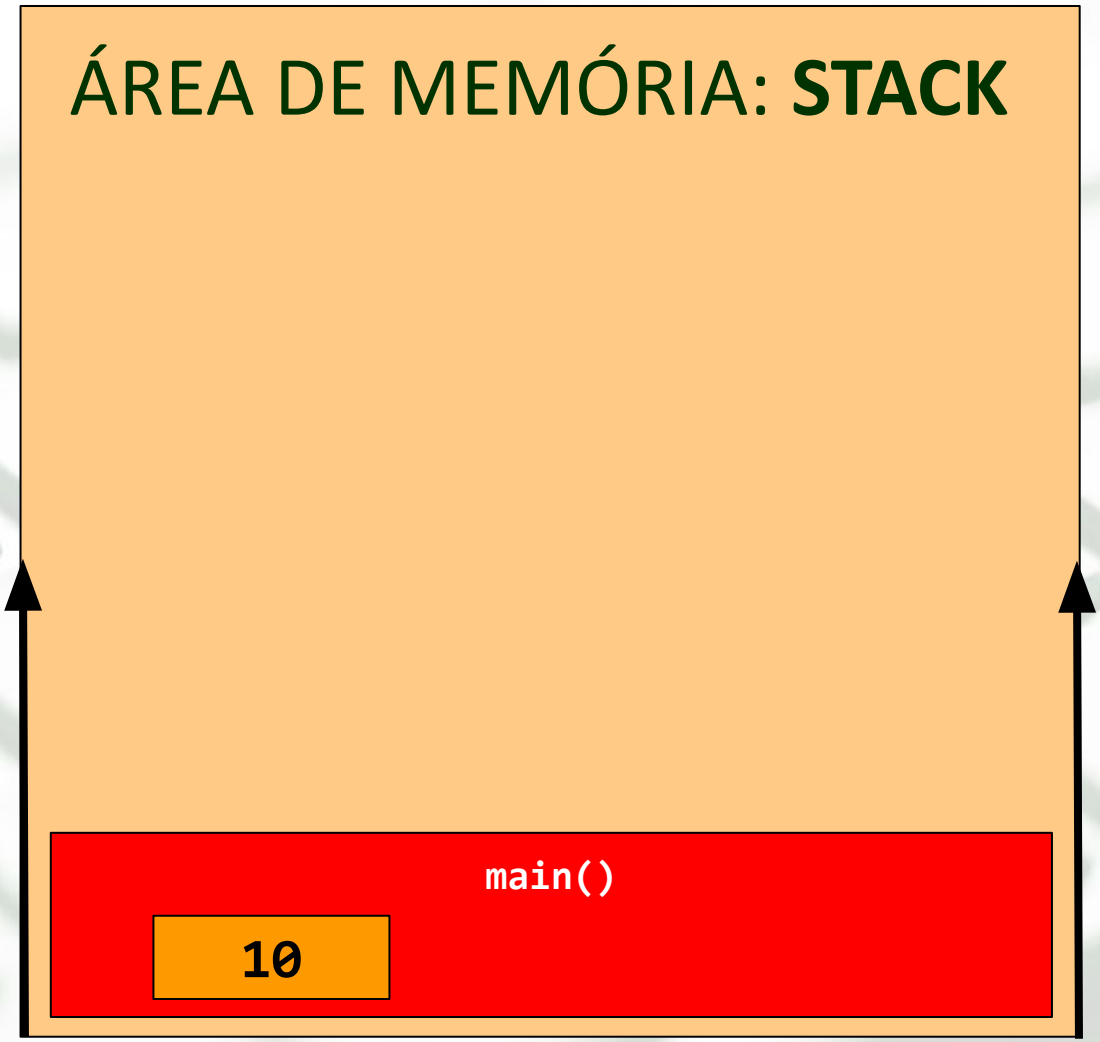
```
int main(){
    int a = 10;
    func_b();
}

void func_b(){
   int b = 20;
   func_c();
}

void func_c(){
   int c = 30;
}
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

EM EXECUÇÃO

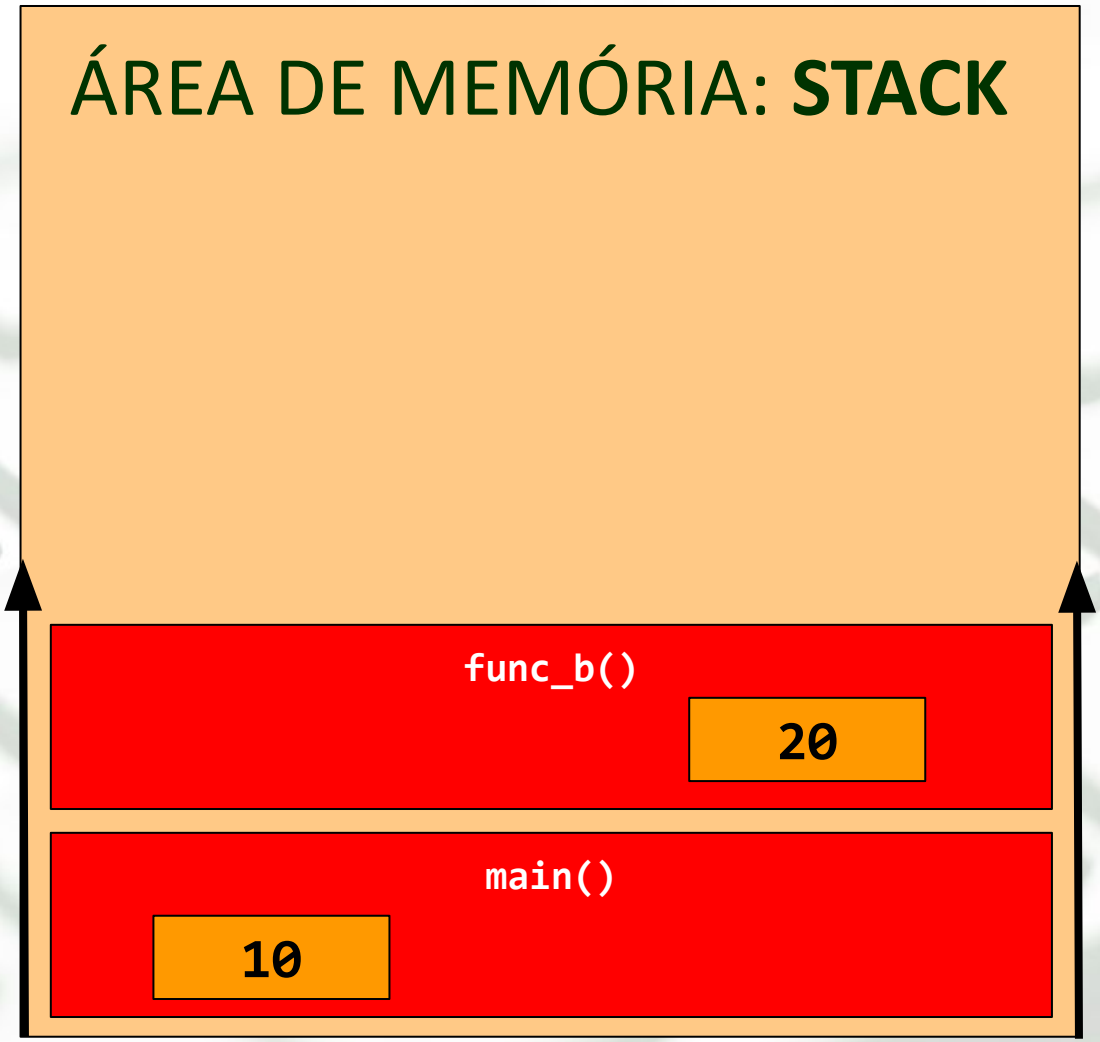
```
int main(){
    int a = 10;
    func_b();
}

void func_b(){
   int b = 20;
   func_c();
}

void func_c(){
   int c = 30;
}
```



Stack vs. Heap

```
int main(){
    int a = 10;
    func_b();
}

void func_b(){
    int b = 20;
    func_c();
}

void func_c(){
    int c = 30;
}
```

EM EXECUÇÃO

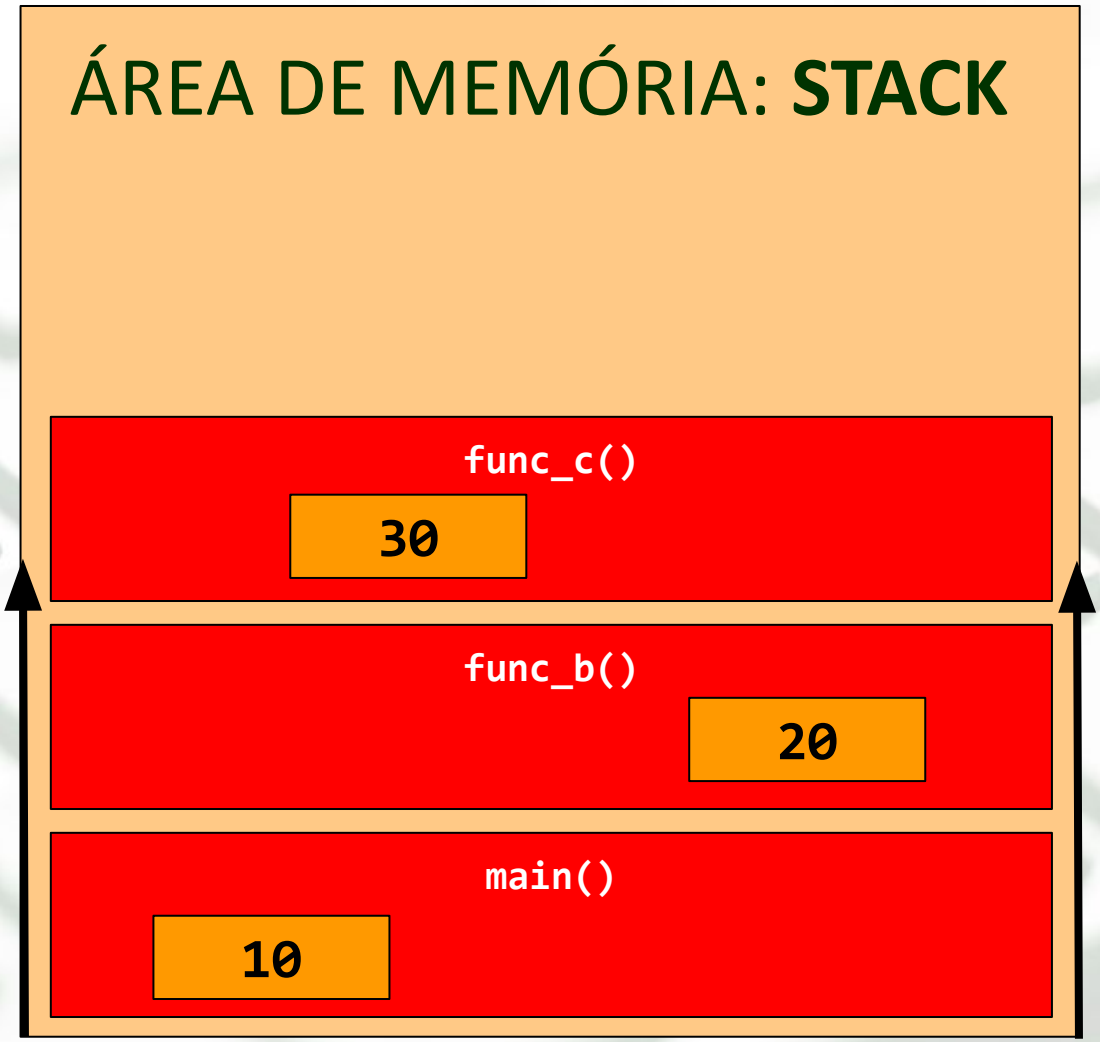


INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

```
int main(){
    int a = 10;
    func_b();
}

void func_b(){
   int b = 20;
   func_c();
}

void func_c(){
   int c = 30;
}
```

EM EXECUÇÃO



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

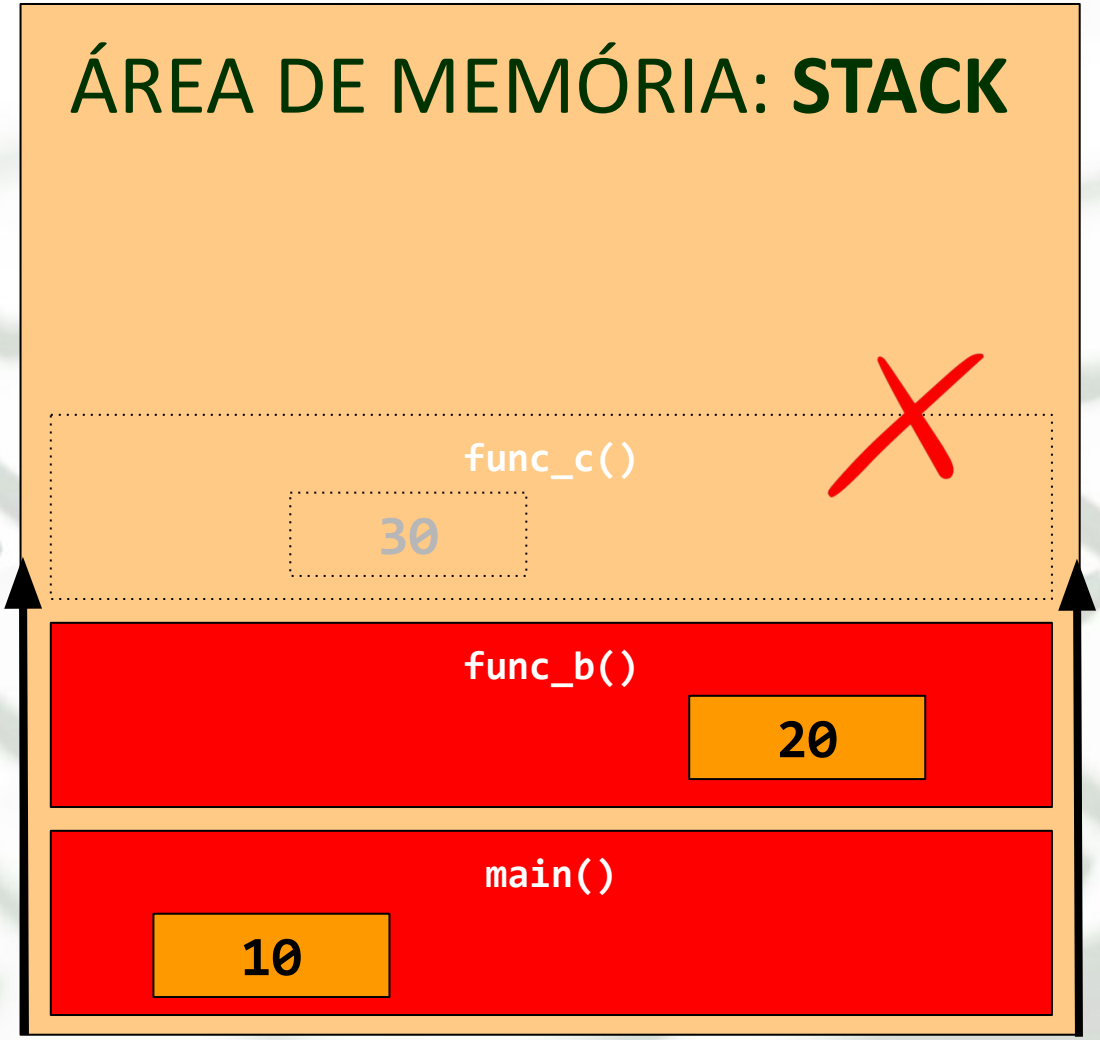
Stack vs. Heap

```
int main(){
    int a = 10;
    func_b();
}
```

EM EXECUÇÃO

```
void func_b(){
   int b = 20;
   func_c();
}
```

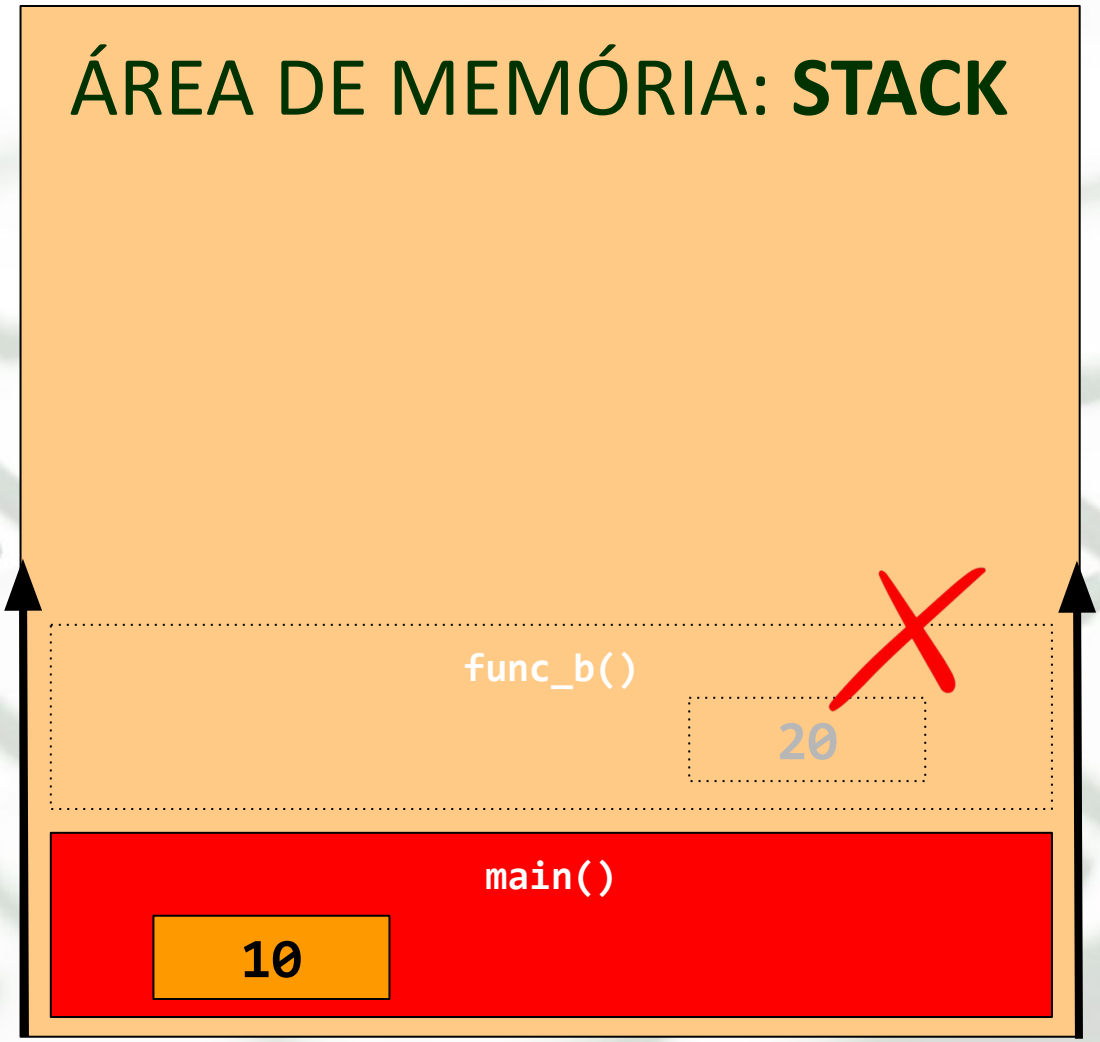
```
void func_c(){
   int c = 30;
}
```



Stack vs. Heap

EM EXECUÇÃO

```
int main(){
    int a = 10;
    func_b();
}

void func_b(){
   int b = 20;
   func_c();
}

void func_c(){
   int c = 30;
}
```



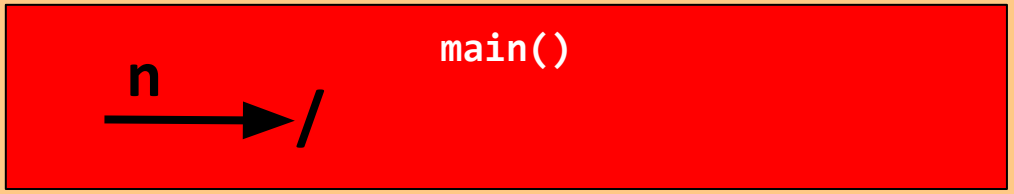
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

```
int main(){
    Doc* n = NULL;
}
```

ÁREA DE MEMÓRIA: **HEAP**

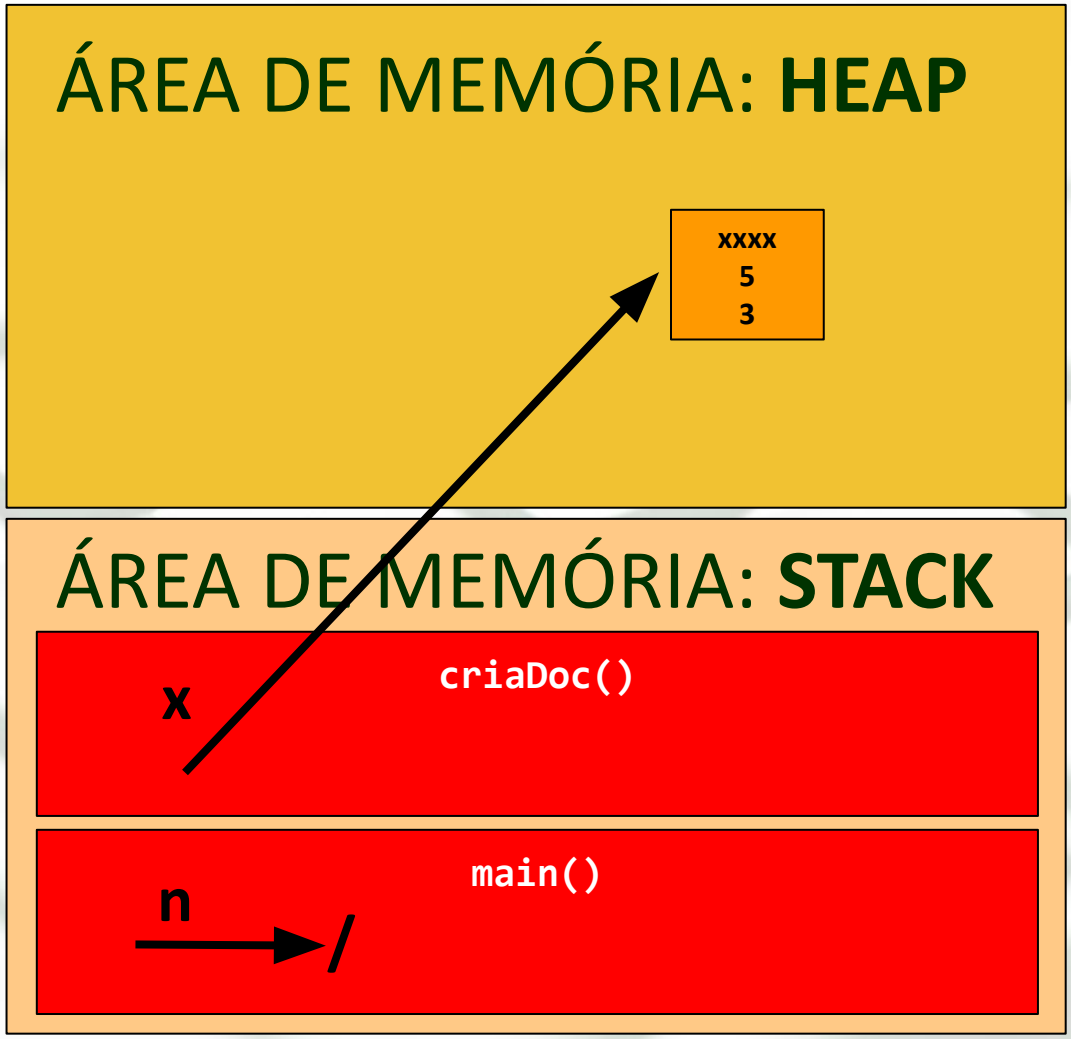
ÁREA DE MEMÓRIA: **STACK**



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

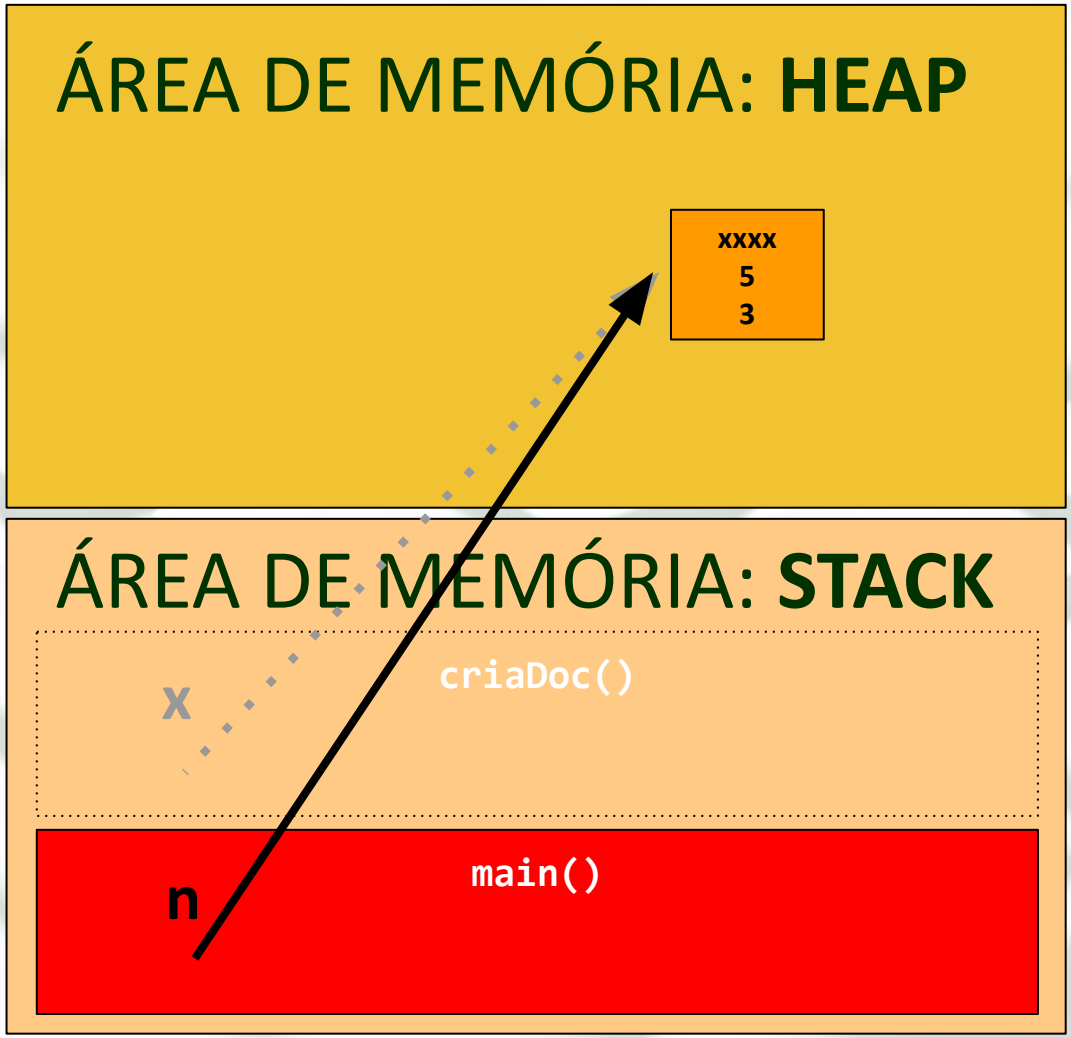
```
int main(){
    Doc* n = NULL;
}

Doc* criaDoc(...){
   Doc* x = malloc();
   (...)
}
```



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

```
int main(){
    Doc* n = NULL;
    n = criaDoc();
}

Doc* criaDoc(...){
   Doc* x = malloc();
   (...)
   return x;
}
```



O ESPAÇO DE MEMÓRIA ALOCADO DINAMICAMENTE POR UMA FUNÇÃO NÃO É "PERDIDO" QUANDO ESSA FUNÇÃO FINALIZA.

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR!!!

Esta implementação está correta?

para armazenar os dados
essão (nome do arquivo,

```
Documento* cria_Documento(char nome[],int pgs,int prio){
    Documento* novo = malloc(sizeof(Documento));
    strcpy(novo->nome,nome);
    novo->paginas = pags;
    novo->prioridade = prio;
    return novo;
}
```

```
Documento* cria_Documento("doc.pdf",5,1);
void imprime_Documento(doc);
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR!!!

Esta implementação está correta?

para armazenar os dados
essão (nome do arquivo,

```
Doc                                          ){

}
```

RESPONDA...

Esse protótipo de *firmware* de impressora, consegue gerenciar a lista de impressão para quantos documentos no máximo?

```
Documento* cria_Documento("doc.pdf",5,1);
void imprime_Documento(doc);
```

Alocação Dinâmica

- **O que fazer para gerenciar vários ponteiros?**

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Alocação Dinâmica

- O que fazer para gerenciar vários ponteiros?

e se definirmos então um Array de ponteiros???

Array x Alocação Dinâmica

- Até então, quando precisamos **armazenar uma coleção de dados de um mesmo tipo**, sempre recorremos a uma estrutura do tipo ***Array***.
- Entretanto, vimos que um *Array* representa uma forma mais primitiva de representar diversos elementos agrupados.
  - Isto porque a estrutura ***Array* não é flexível => Aloc. Estática**.
- Um *Array* convencional sempre é alocado de forma **estática**:
  - Se o número de elementos exceder a dimensão do array, teremos **problemas** de execução.
  - Se o número de elementos estiver abaixo do limite do array, teremos **problemas** de desperdício/desempenho.

Estruturas de Dados Dinâmicas

- A **solução ótima** para este tipo de situação é a utilização de estruturas que **possam crescer na medida em que precisarmos armazenar novos elementos**, *e diminuir na medida que elementos não forem mais necessários*.
- Tais estruturas são chamadas **dinâmicas** e armazenam cada um dos seus elementos através da técnica de **Alocação Dinâmica**.

Estruturas de Dados Dinâmicas

- A **solução ótima** para este tipo de situação é a utilização de estruturas que **possam crescer na medida em que precisarmos armazenar novos elementos**, *e diminuir na medida que elementos não forem mais necessários*.
- Tais estruturas são chamadas **dinâmicas** e armazenam cada um dos seus elementos através da técnica de **Alocação Dinâmica**.

Entretanto... De graça no mundo só carinho de mãe...

Array x Alocação Dinâmica

- Analise...

```
#include<stdlib.h>

int function(){
   int* v1 = malloc(10*sizeof(int));

}
```

... O que está sendo gerado dinamicamente???

Array x Alocação Dinâmica

- Analise...

```
#include<stdlib.h>

int function(){
    int* v1 = malloc(10*sizeof(int));

}
```

Sim, um Array gerado por meio de Alocação Dinâmica

Array x Alocação Dinâmica

- Analise...

```
#include<stdlib.h>

int function(){
    int* v1 = malloc(10*sizeof(int));
    int v2[10];
}
```

V1 é diferente de V2 ???

Stack vs. Heap

```
#include<stdlib.h>

void function(){
   int* v1 = malloc(10*sizeof(int));
   int v2[10];
}
```

- Enquanto **v1** é armazenado dinamicamente no espaço de memória **HEAP**, e só deixa de existir caso seja explicitamente requerido por meio de **free(v1)**
- **v2** é armazenado no espaço de memória **STACK**, ***e por isso deixa de existir quando a função é finalizada.***

Stack vs. Heap

- Teste para provar...

```
#include<stdlib.h>

int* function(){
    int* v1 = malloc(10*sizeof(int));
    int v2[10];
    v1[5] = 999;
    v2[5] = 999;
    return v1 // &v2
}

int main(){
    int* t = function();
    printf("t[5] == %d\n", t[5]);
}
```

Acesso Direto

- Em ambas as situações entretanto, aloca-se um **espaço contíguo de memória** para armazenar as informações.

Endereço	6002	6006	6010	6014	6018	6022	6026	6030	6034	6038
Índice	0	1	2	3	4	5	6	7	8	9
Valor	92	51	18	40	46	83	45	71	13	62

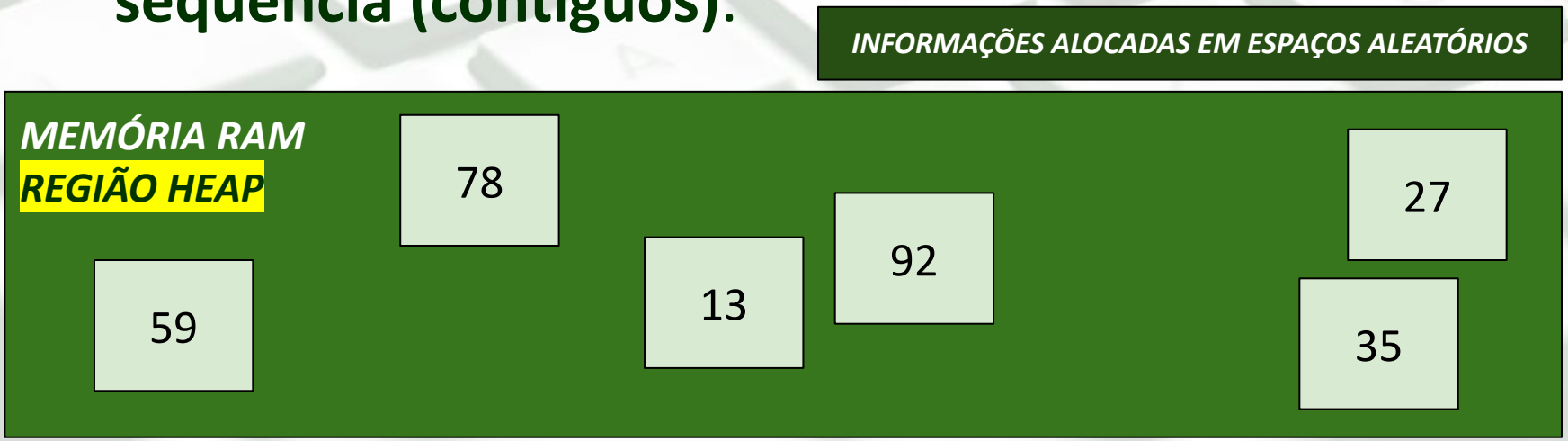
VEJA... Se v1 inicia no endereço 6002, e `int` possui 4 Bytes, então V[6] == 6002+(6*4) == 6026

- **Isso facilita muito o acesso a qualquer elemento do array**, pois o **acesso é direto**, basta conhecer o **endereço inicial do Array** (que é o seu próprio nome) e o **tamanho do elemento que armazena** (que é sempre fixo).

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Acesso Direto

- Em Estruturas de Dados **Dinâmicas** (que não sejam *Arrays*) é **impossível obter essa mesma vantagem**...
  - Isto porque os elementos são alocados de forma dinâmica (**em tempos e posições aleatórias**).
  - **Não há como garantir que os dados estejam em sequência (contíguos)**.



Acesso Direto

- Em Estruturas de Dados **Dinâmicas** (que não sejam *Arrays*) é **impossível obter essa mesma vantagem**…
  - Isto porque os elementos são alocados de forma dinâmica (**em tempos e posições aleatórias**).
  - **Não há como garantir que os dados estejam em sequência (contíguos)**.

INFORMAÇÕES ALOCADAS EM ESPAÇOS ALEATÓRIOS

MEN
REG

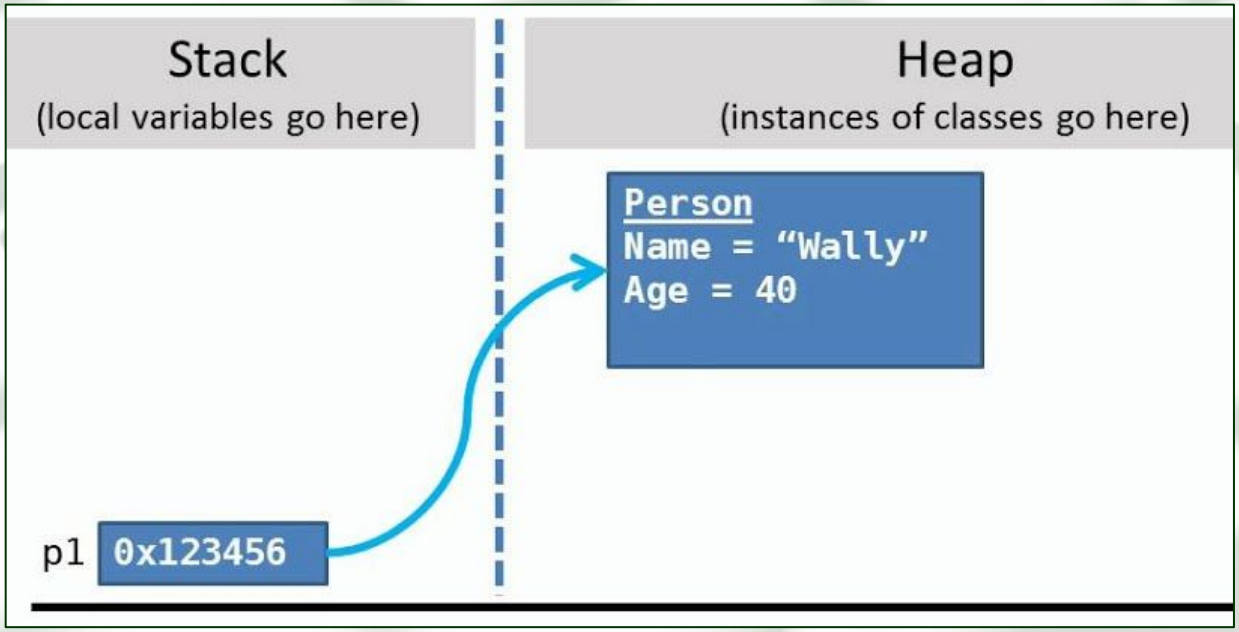
E esse é o preço "a se pagar" quando utilizamos Estruturas Dinâmicas tais como: Listas encadeadas, Árvores, Grafos e Hash Tables…

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Stack vs. Heap

Arranjo da memória em ***Alocação Dinâmica…***

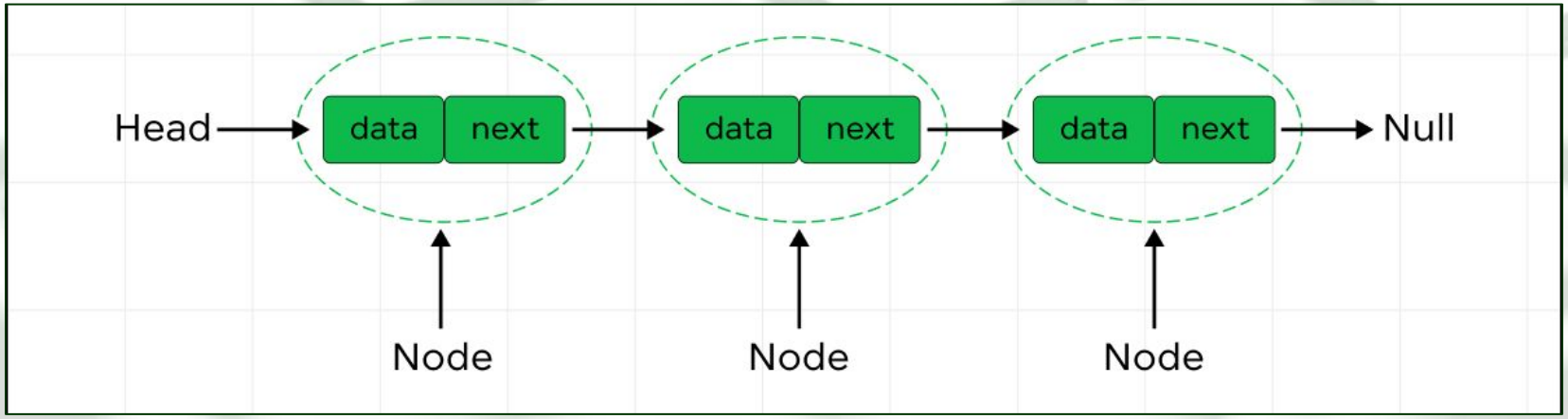
```
int main(){
    Person* p1;
    p1 = setPerson("Wally", 40);
}
```



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

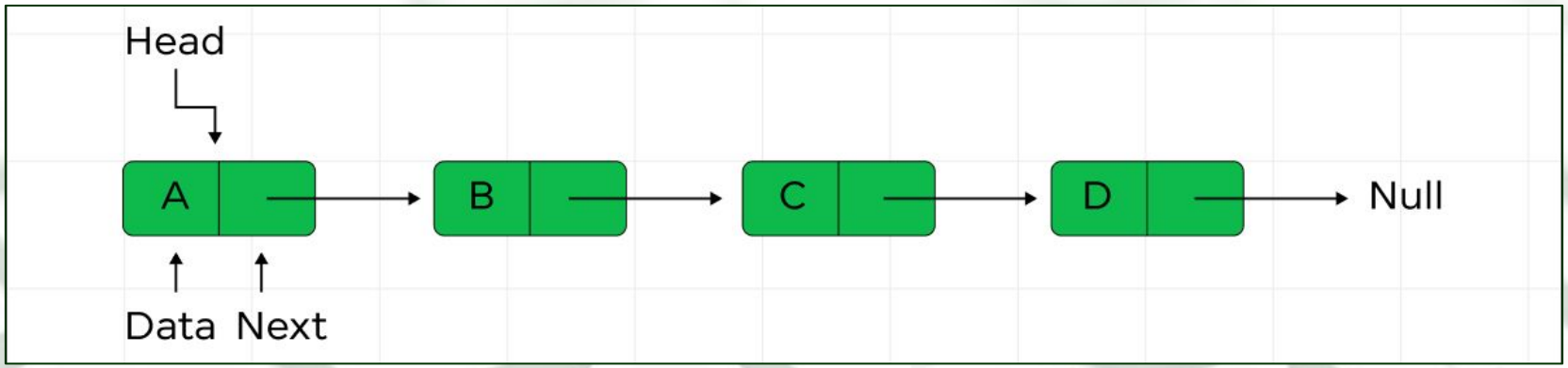
Listas Encadeadas

- **Listas Encadeadas** são exemplos de Estrutura de Dados **dinâmicas**.
- Uma Lista consiste numa sequência encadeada de elementos, genericamente chamados de "**nós**", sendo este encadeamento realizado por meio de **ponteiros**.



Listas Encadeadas

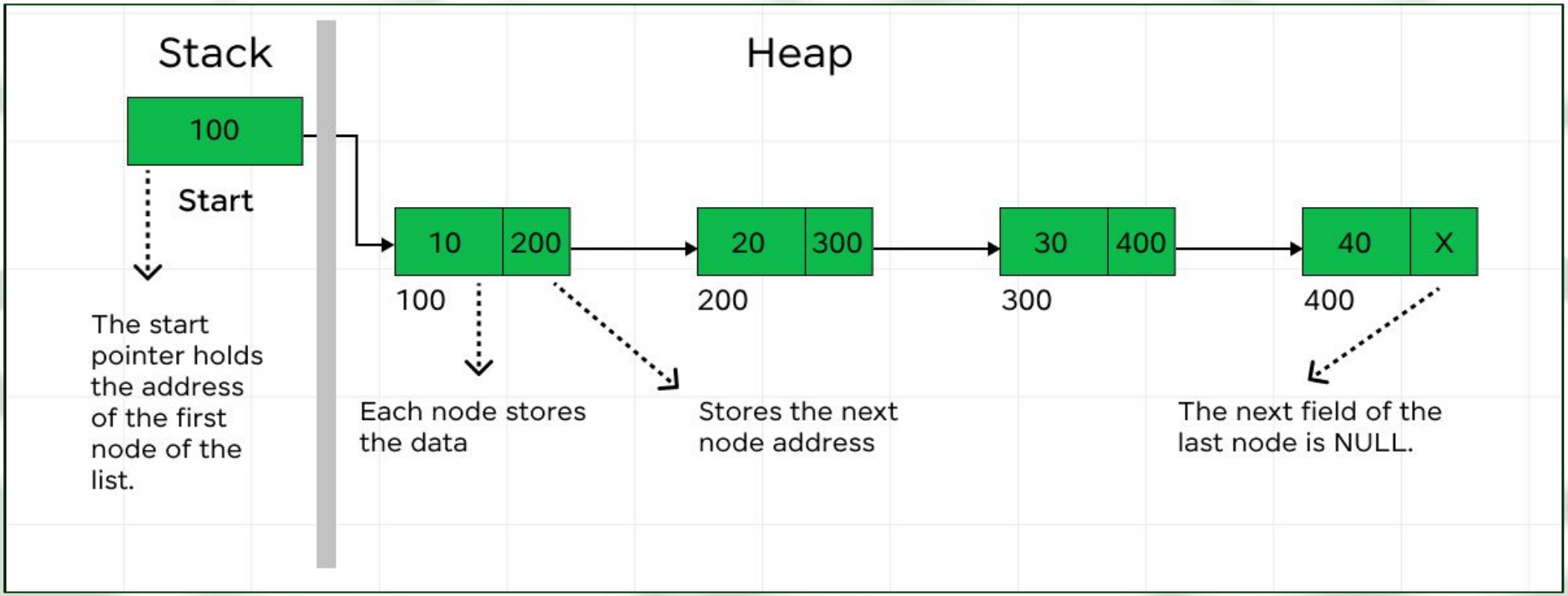
- Representação de uma Lista Encadeada



  - Do primeiro elemento (*HEAD*), acessamos o segundo.
  - Do segundo ao terceiro, e assim por diante...
  - O último elemento da lista, aponta para **NULL**, sinalizando que não existe um próximo registro.

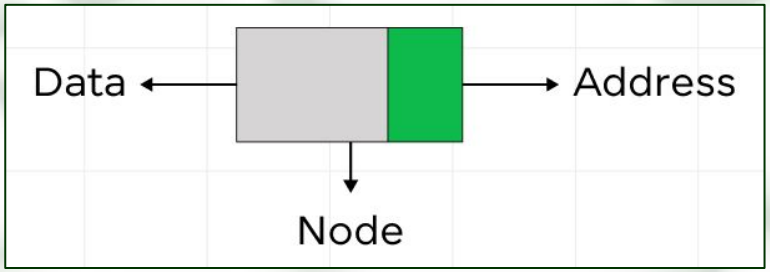
Organização em Memória

- Arranjo de memória de uma Lista Encadeada:



Lista Simplesmente Encadeada

- Como podemos observar, cada elemento (nó) da lista, deve apontar para o nó subseqüente.
- Este apontamento é realizado através de variáveis do **tipo ponteiro**.
- **Portanto, cada elemento (nó) deve possuir, em sua estrutura, um ponteiro para o seu próprio tipo de dados.**



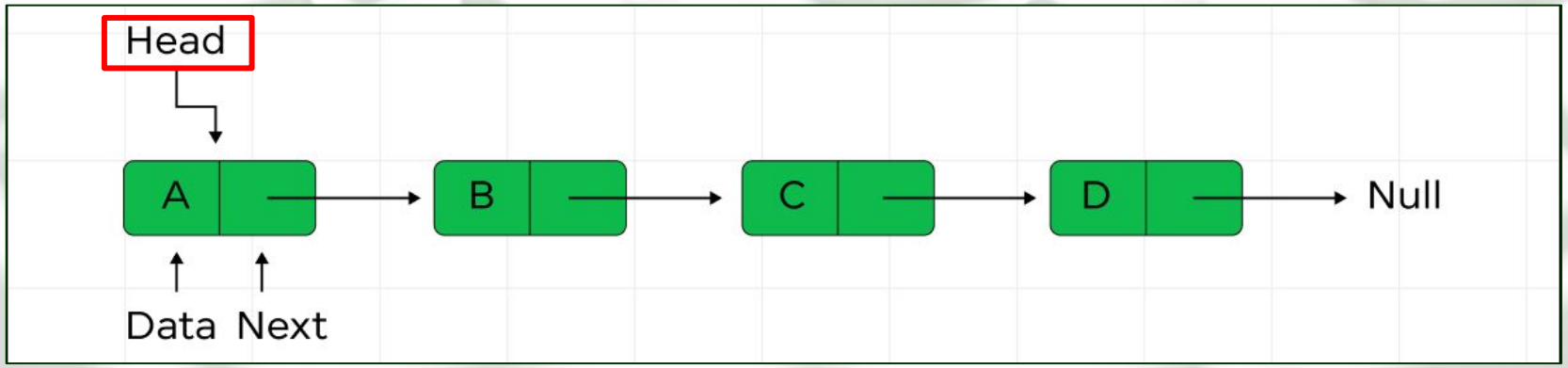
Lista Simplesmente Encadeada

- Traduzindo isso em linguagem de programação...

```
typedef struct No{
    int dado;
    (...)
    struct No *prox;
}No;
```

Lista Simplesmente Encadeada

- Para referenciarmos uma lista encadeada sempre será necessário manter armazenado (e atualizado), **o ponteiro para o primeiro nó da lista**.
- A partir do primeiro nó da lista, podemos percorrer todos os encadeamentos subseqüentes.



Inserindo Nós na Lista

- Para inserir um elemento (Nó) na lista, é necessário:
  - Alocar o espaço de memória => função **`malloc()`**.
  - Atribuir/Ler as informações úteis do Nó.
  - **Encadear/Linkar o novo Nó à lista...**
    - **No Início da Lista???**
    - **No Final da Lista ???**

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Inserindo Nós na Lista

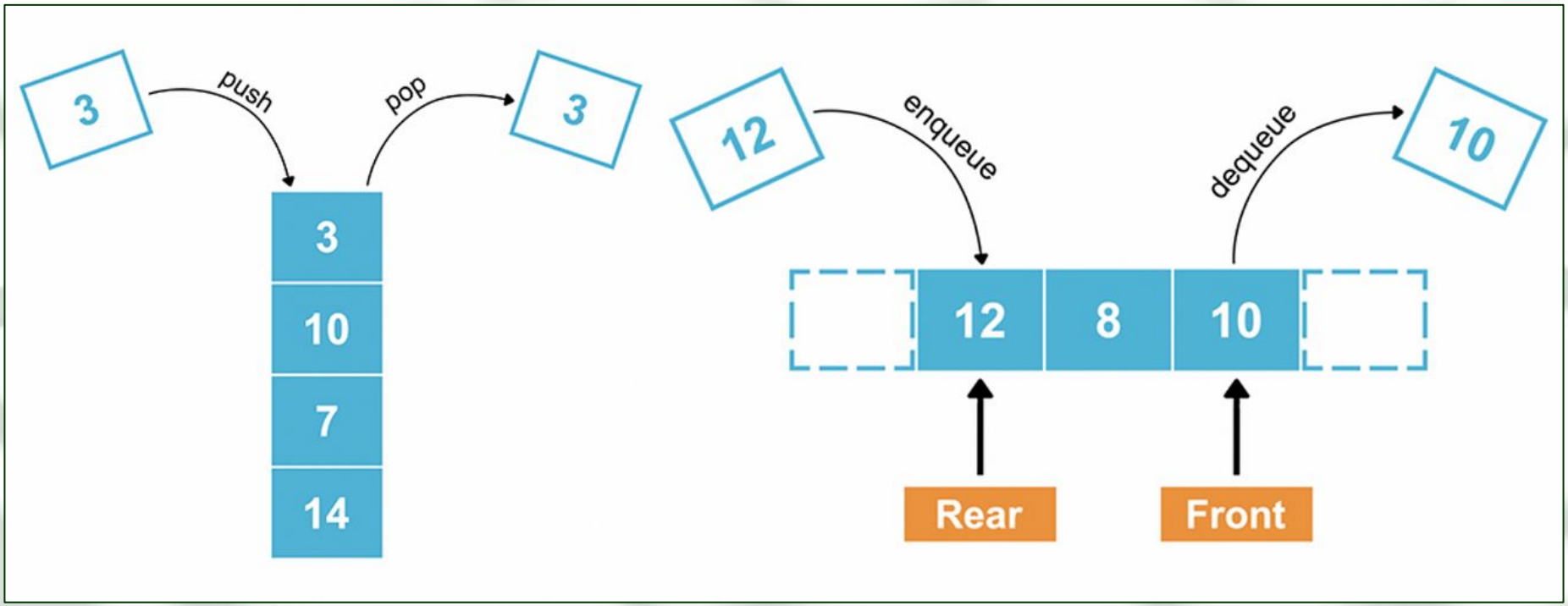
- Para inserir um elemento (Nó) na lista, é necessário:
  - Alocar o espaço de memória => função **`malloc()`**.
  - Atribuir/Ler as informações úteis do Nó.
  - **Encadear/Linkar o novo Nó à lista…**
    - **No Início da Lista???**
      - → **PILHA/STACK (Algoritmo LIFO)**
    - **No Final da Lista ???**
      - → **FILA/QUEUE (Algoritmo FIFO)**

****Não confundir essa STACK (**Paradigma de Lista Encadeada**) com a STACK (região da memória que existe em cada processo, como visto antes).*

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

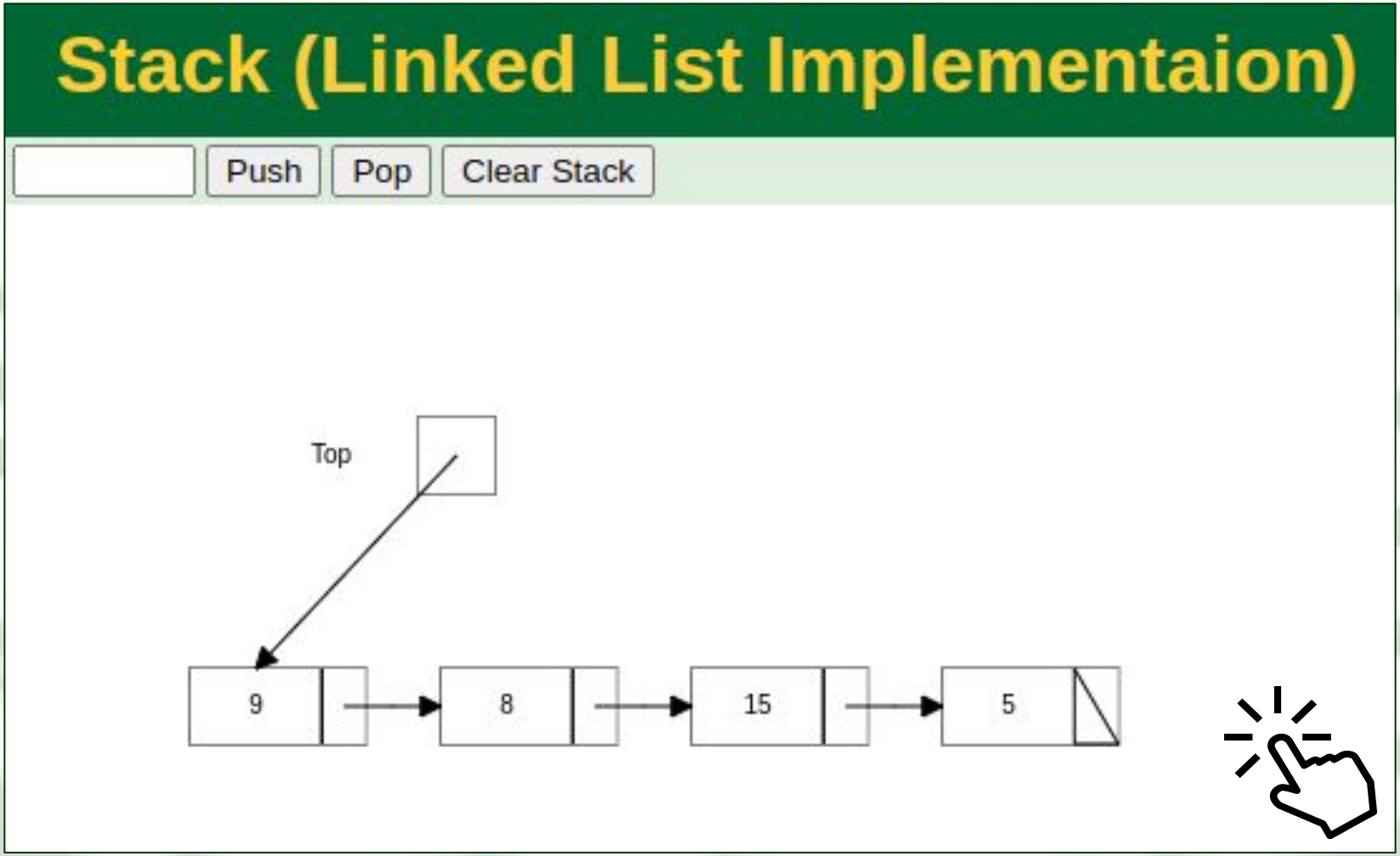
Pilhas e Filas

- Representação de Pilhas e Filas...



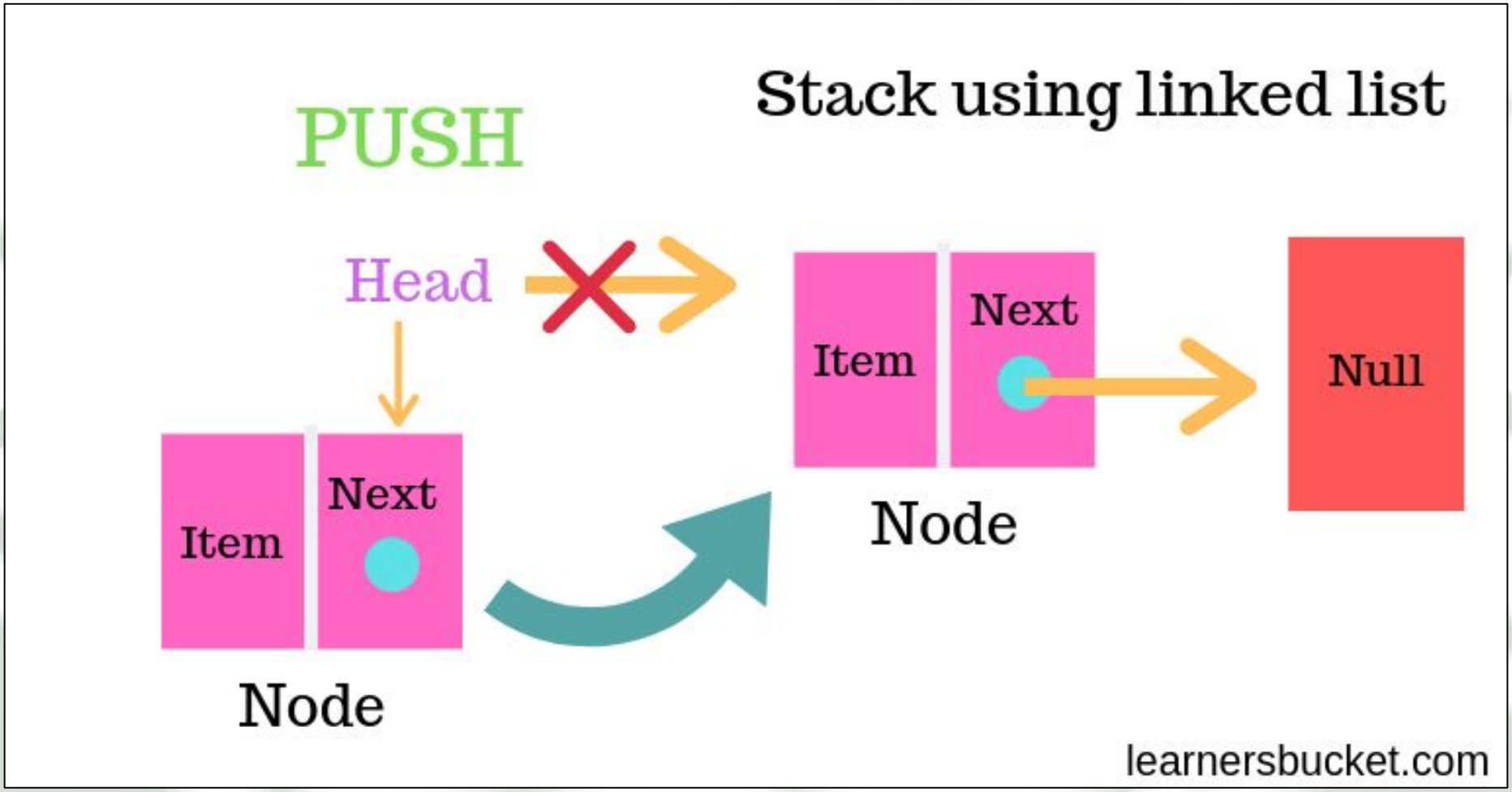
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Animação Didática



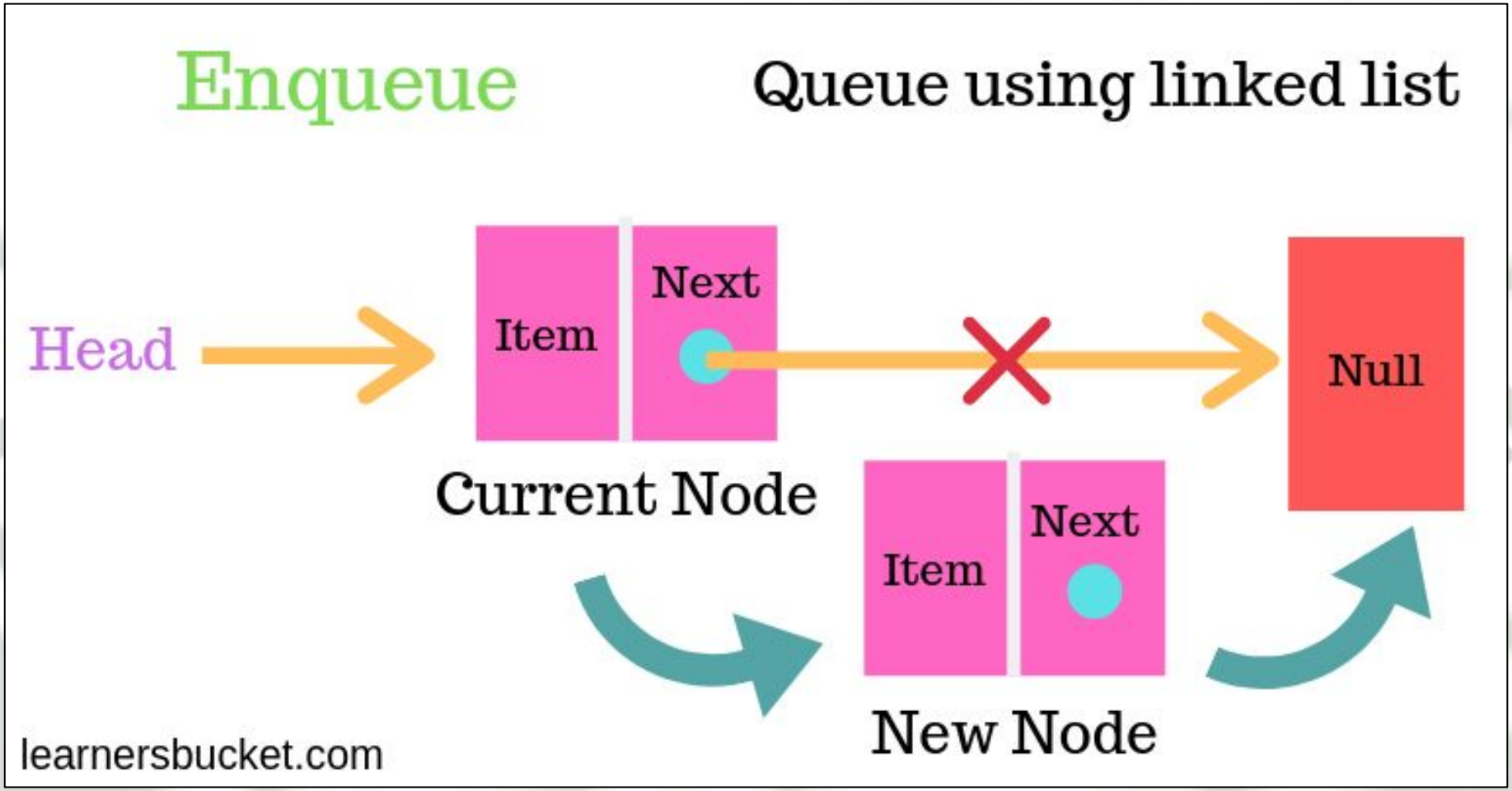
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

PILHA (STACK)



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

FILA (QUEUE)



Inserção de Nós

```
No* setNo(No *inicio){
   No *novo;
   novo = malloc(sizeof(No));
   scanf(" %d",&novo->dado);
   novo->prox = inicio;
   return novo;
}

int main(){
   No *lista = NULL;
   for (int i=0; i<10; i++)
       lista = setNo(lista);
}
```

Acesso aos Nós

- Versão **Iterativa**...

```
void getNos(No *pont){
   while(pont){
       printf("%d\n",pont->dado);
       pont = pont->prox;
   }
}
```

Acesso aos Nós

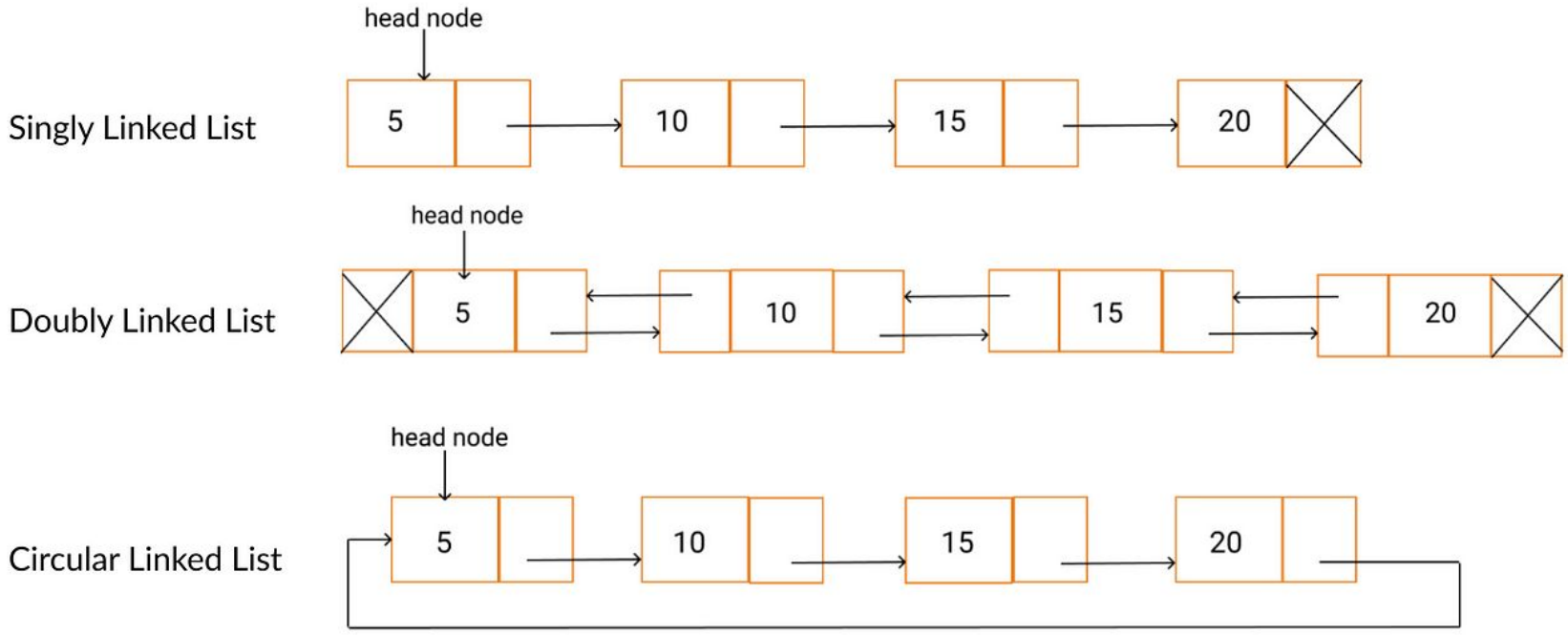
- Versão **Recursiva**...

```
void getNos(No* pont){
   if(pont)
       printf("%d\n", pont->dado);
   getNos(pont->prox);
}
```

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Tipos de Listas Encadeadas

Principais variantes de Listas Encadeadas...



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR (1)

- Um site de anúncios de veículos precisa desenvolver uma aplicação. Faça um programa modular que realize o cadastro dinâmico de estruturas do tipo **Veículo** (tipo de veículo, ano, modelo, km, valor).
- Anúncios mais recentes devem ser visualizados primeiro.
  - Utilize estrutura do tipo Lista Encadeada.
  - Implemente uma função para o cadastro de nós.
  - Implemente uma função para listagem dos veículos.
  - Implemente uma função que busque todos os veículos de determinado tipo.
  - Implemente uma função que retorna o valor médio dos veículos, por cada tipo cadastrado.

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR (2)

- Outra abordagem para **Lista Encadeadas** é o tratamento de Filas (**FIFO**), onde os nós são inseridos ao final da lista, e não no início…
- Para facilitar a implementação deste método, além do ponteiro indicando o início da lista, também é mantido um ponteiro que armazena o **último elemento da lista**.
- As leituras se baseiam em um **ponteiro início**, enquanto as inclusões são baseadas em um **ponteiro fim**.

Refatore o problema do firmware de Impressora, para gerenciar a impressão de número variado de documentos.

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Bora CODAR (3)

- Imagine a situação de necessitar **EXCLUIR** um documento enviado para impressão.
- Como é possível realizar a exclusão de um Nó em Lista Encadeada?

Implemente essa nova funcionalidade ao código anterior.