



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 precisar **aumentar a turma** depois que o programa foi distribuído???



Alocação Estática x Dinâmica

- A alocação das estruturas de dados na aplicação anterior (*Array* de Alunos) foi realizada de forma **estática**.
- A **alocação de memória estática** acontece **uma única vez**, durante a criação do processo, não sendo possível alterá-la ***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 mesmo 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 mesmo 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.

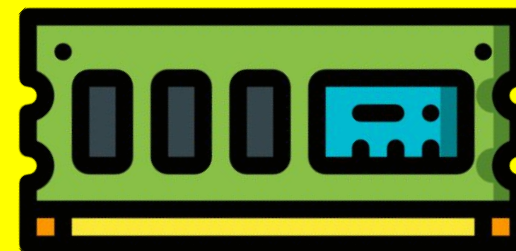


Alocação Estática x Dinâmica

- **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:

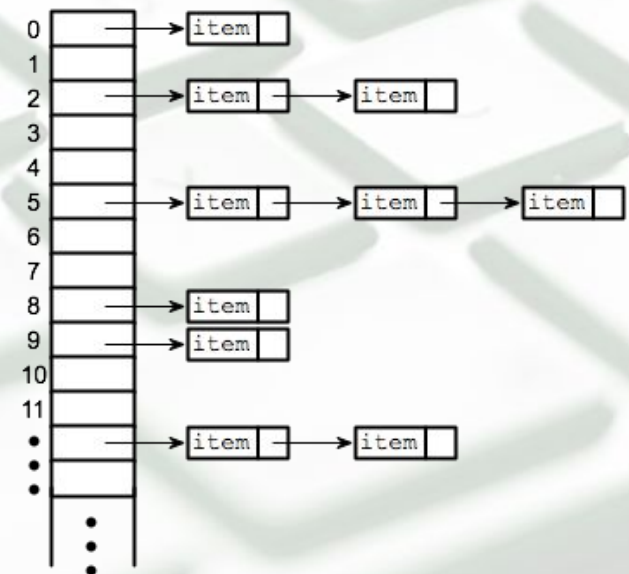
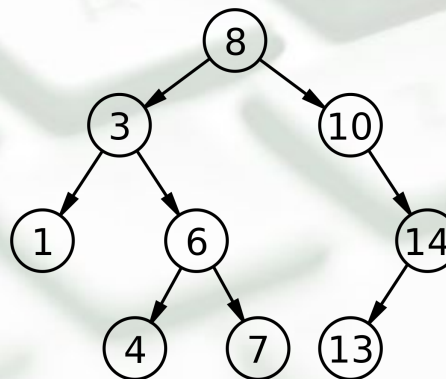
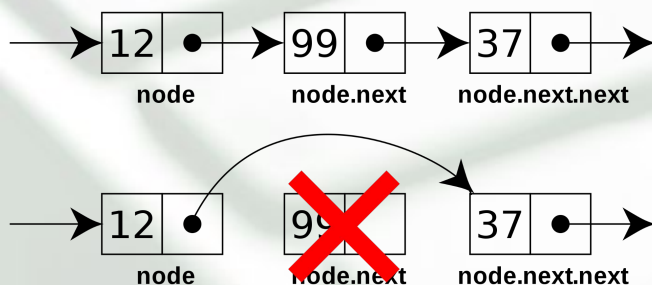
A MEMÓRIA PRINCIPAL



o armazenamento de todas as informações.

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.
 - ...





Alocação Dinâmica

- A **Alocação Dinâmica** é acontece por meio de duas funções principais:
 - **malloc** (*Memory **ALLO**Cation*)
 - **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***.

Função de Alocação

```
#include "stdlib.h"
```

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

uma variável int possui
4 bytes





Função de Alocação

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

Mas... como saber exatamente o tamanho que uma variável ou struct ocupa em memória???





Função de Alocação

```
#include "stdlib.h"

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

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



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));
}
```




Atenção!

- Cuidado ao trabalhar com *ponteiros de structs...*

```
aluno *a = NULL;  
a = malloc(sizeof(aluno));  
  
*a.matricula;           //é equivalente a...  
*(a.matricula);         // mas é diferente de...  
(*a).matricula;
```

- O operador `->` é uma abreviatura muito útil!!!
dt->dia equivale à **(*dt).dia**



Exemplo de Código

```
#include "stdio.h"
#include "stdlib.h"

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

int main(){
    Aluno *a;
    a = malloc(sizeof(Aluno));
    scanf(" %d", &a->matricula);
    scanf(" %[^\n]s", 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** ao término da execução do programa, ou sempre que o espaço de memória de uma variável não for mais útil, para evitar erros inesperados, e para economia de memória do sistema.



Exemplo de Código

```
#include "stdlib.h"

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

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



Exercício A

- Defina um novo tipo **Funcionário** para armazenar os dados de um empregado (nome, RG, matrícula e salário).
- Declare um **ponteiro** (*não uma variável*) do tipo **Funcionário**.
- Faça a alocação dinâmica em memória e realize a leitura e impressão de todas as informações do empregado em **funções** específicas...
 - ***Funcionario* setFuncionario()***
 - ***void getFuncionario(Funcionario* func)***



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 => Alloc. Estática**.
- Um *Array* **sempre** é alocado de maneira estática, portanto:
 - Se o número de elementos exceder a dimensão do vetor, teremos **problemas** de execução.
 - Se o número de elementos estiver abaixo do limite do vetor, 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 main(){
    int* v;
    v = malloc(10*sizeof(int));
}
```

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



Array x Alocação Dinâmica

■ Analise...

```
#include "stdlib.h"

int main(){
    //int* v;
    //v = malloc(10*sizeof(int));
    int v[10];
    for(int i=0; i<10; i++)
        v[i] = rand()%100;
}
```




Array x Alocação Dinâmica

- Quando declaramos um vetor, alocamos 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 int possui 4 Bytes e se $\&V == 6002$, então $\&V[6] == \&V + 6 * 4 == 6002 + 24 == 6026$

- Isso facilita **muito** o acesso a qualquer elemento do vetor, pois, basta conhecer o endereço inicial do Array e já é possível fazer **acessos diretos à informação desejada**.



Array x Alocação Dinâmica

- Em Estruturas de Dados **Dinâmicas** 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

MEMÓRIA RAM

59

78

13

92

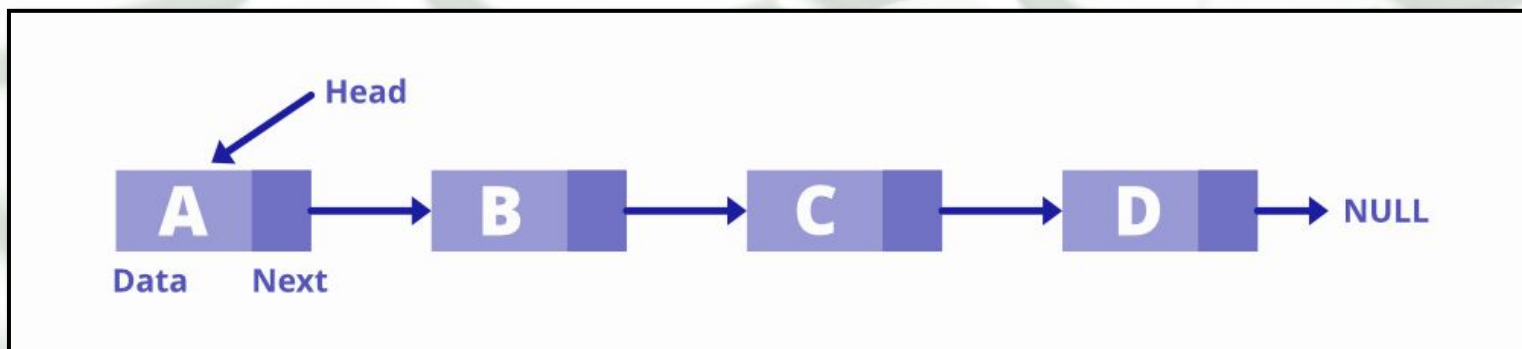
27

35



Listas Encadeadas

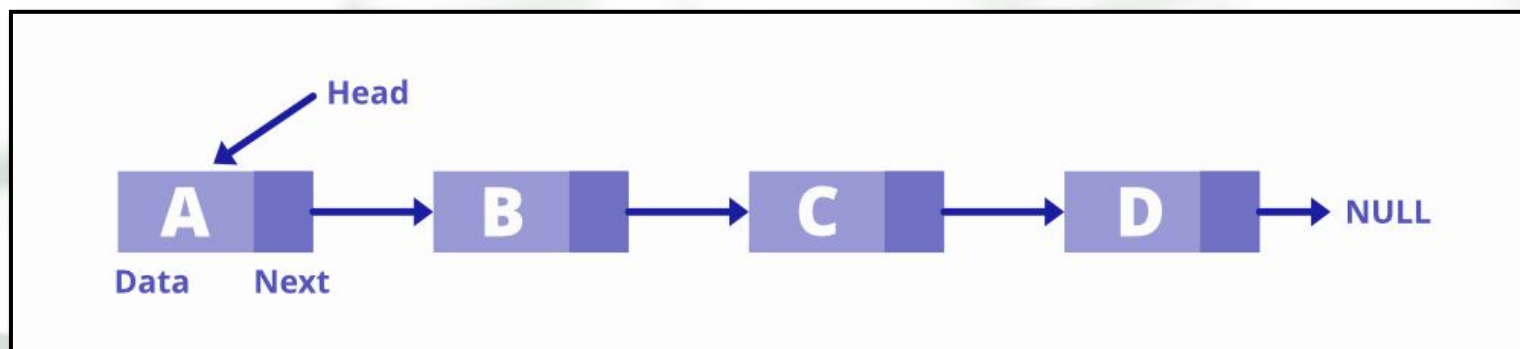
- **Listas Encadeadas** são exemplos de estrutura de dados dinâmicas.
- Uma estrutura do tipo ***Lista*** consiste numa sequência encadeada de elementos, genericamente chamados de “**nós**”, sendo este encadeamento realizado por meio de **ponteiros**.





Listas Encadeadas

■ Arranjo da memória de uma lista encadeada:



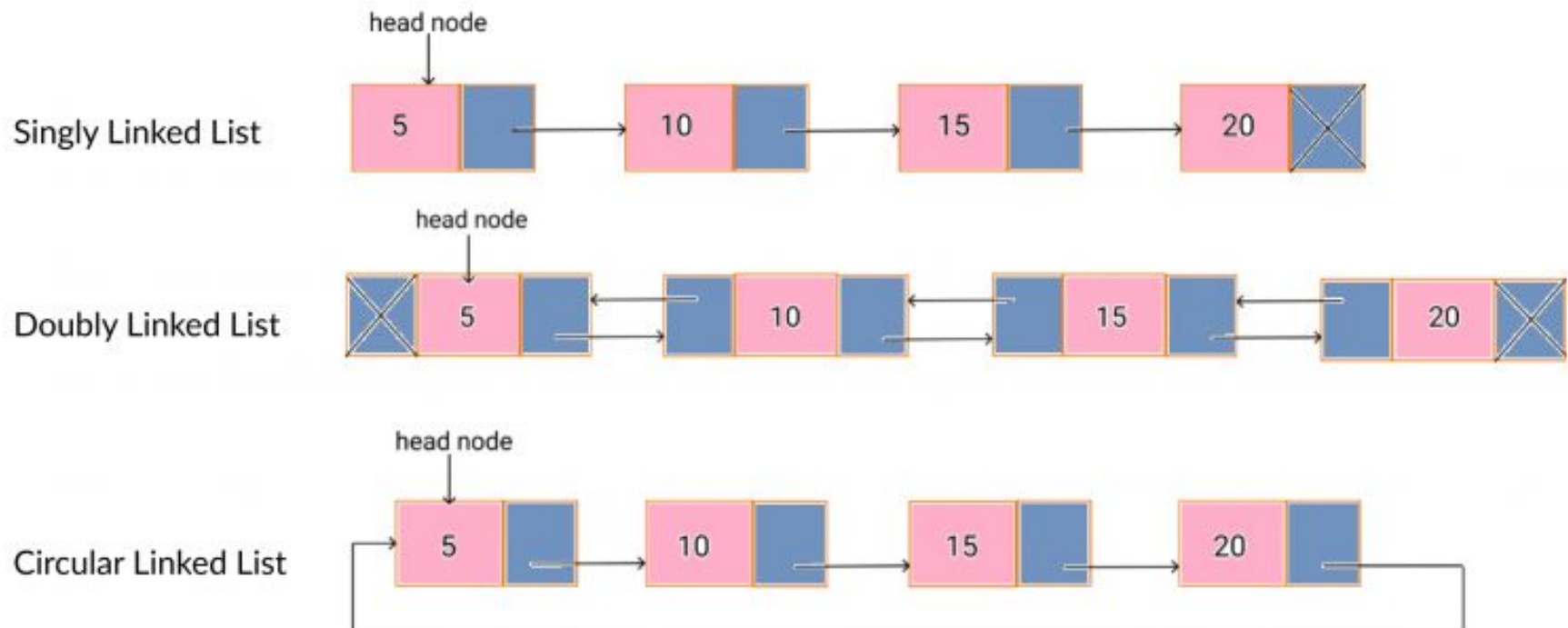
- ❑ Do primeiro elemento, acessamos o segundo.
- ❑ Do segundo ao terceiro, e assim por diante...
- ❑ O último elemento da lista, aponta para **NULL**, sinalizando que não existe um próximo registro.



Tipos de Listas Encadeadas

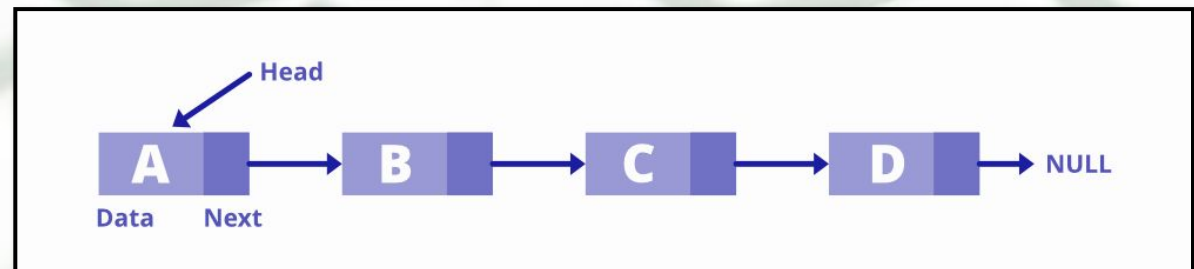
Variantes de Listas Encadeadas...

Types of Linked List



Lista Simplesmente Encadeada

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



Lista Simplesmente Encadeada

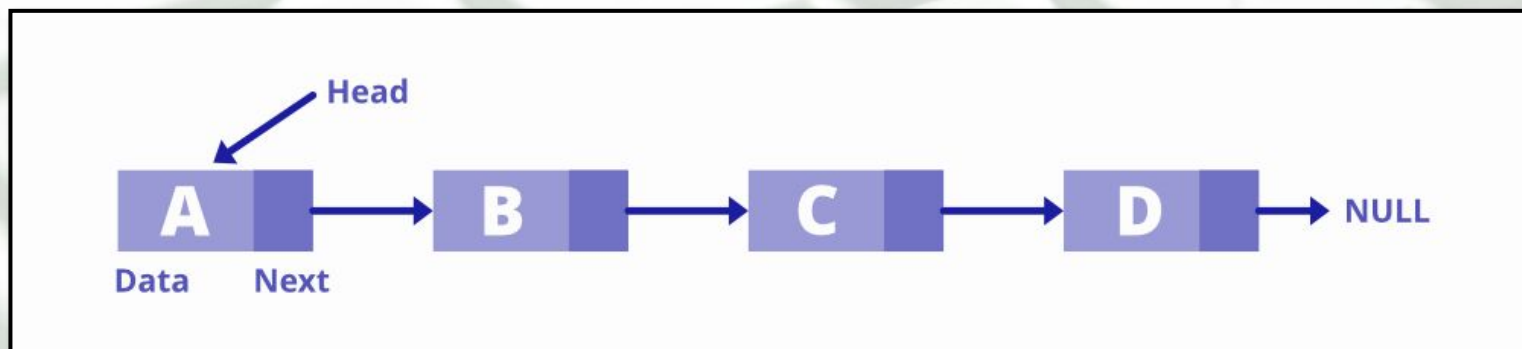
- Traduzindo isso em linguagem de programação...

```
typedef struct No{  
    int informacao;  
    struct No *prox;  
}No;
```



Lista Simplesmente Encadeada

- Uma boa estratégia para referenciarmos uma lista encadeada, é sempre 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()**.
 - Ler as informações úteis do Nó.
 - **Encadear/Linkar o novo Nó à lista...**
 - **No Início da Lista???**
 - **No Final da Lista ???**

Inserindo Nós na Lista

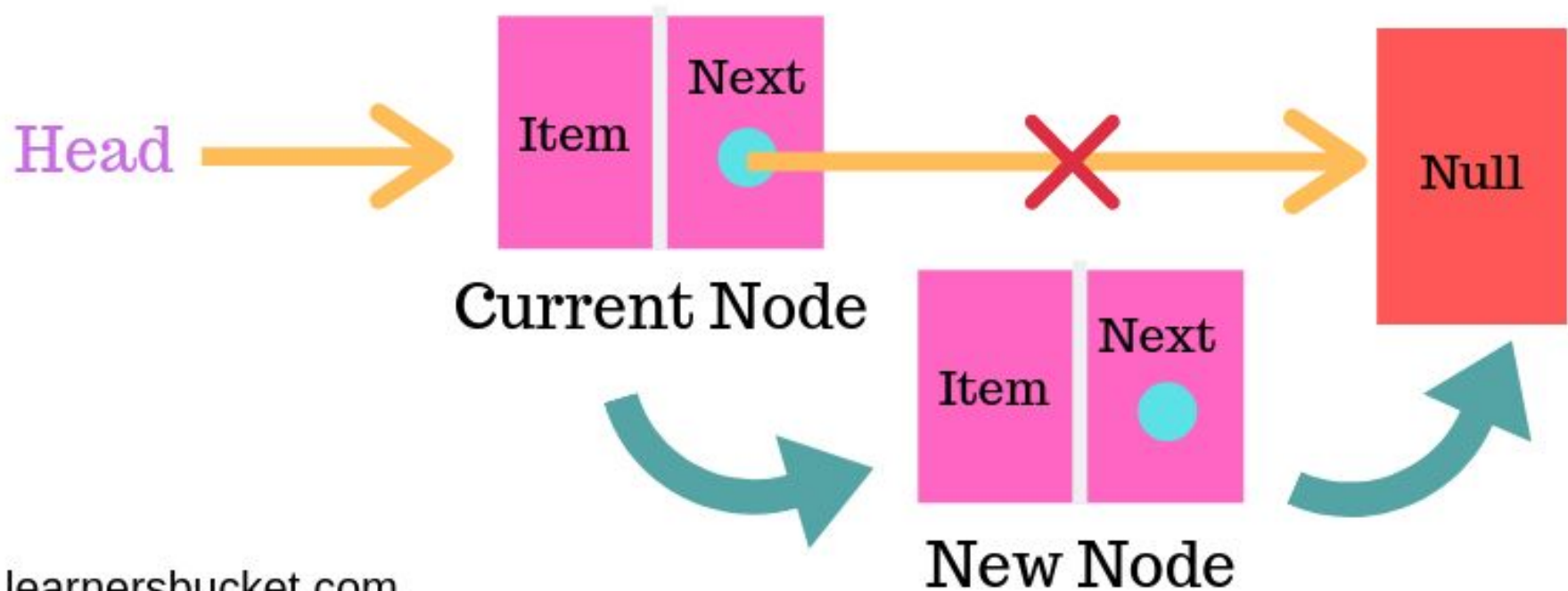
- Para inserir um elemento (Nó) na lista, é necessário:
 - Alocar o espaço de memória => função **malloc()**.
 - Ler as informações úteis do Nó.
 - **Encadear/Linkar o novo Nó à lista...**
 - **No Início da Lista???**
 - **PILHA/STACK** (Algoritmo LIFO)
 - **No Final da Lista ???**
 - **FILA/QUEUE**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



Lista tipo FILA (QUEUE)

Enqueue

Queue using linked list



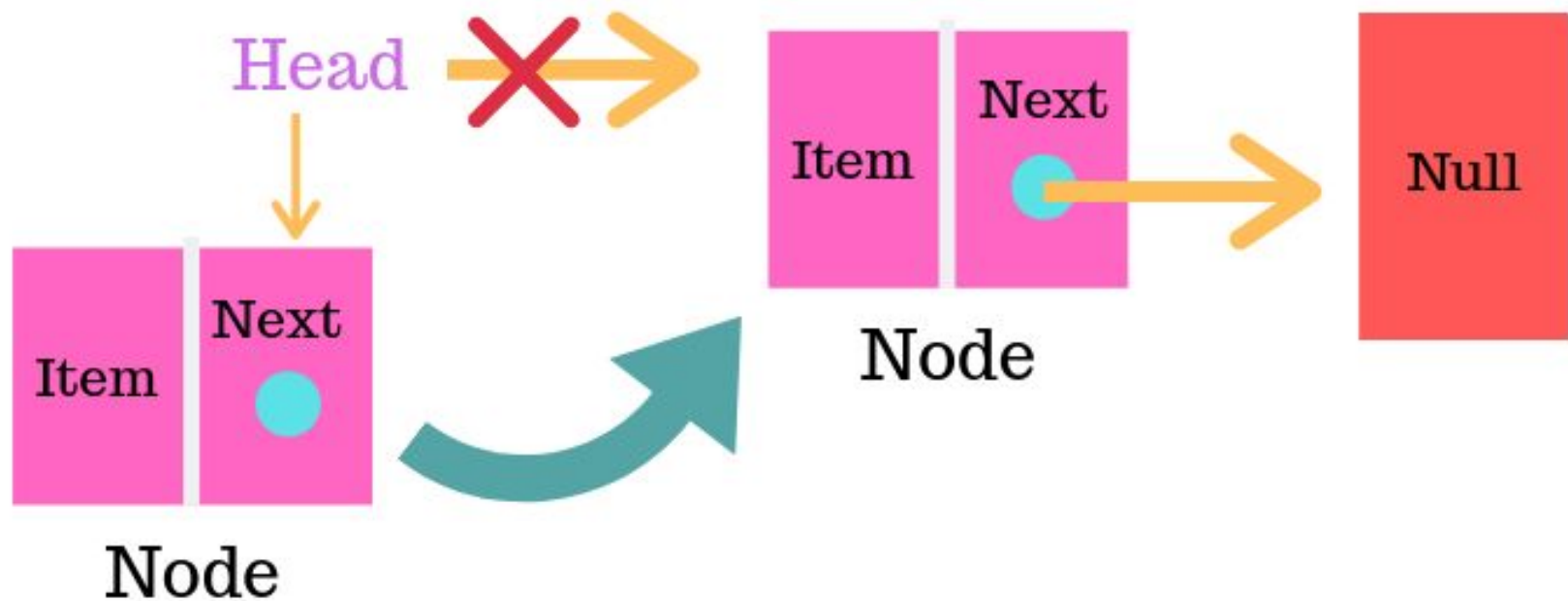
learnersbucket.com



Lista tipo PILHA (STACK)

PUSH

Stack using linked list



learnersbucket.com



Função para Inserção de Nós

```
No* setNo(No *inicio){  
    No *novo;  
    novo = malloc(sizeof(No));  
    scanf(" %d",&novo->informacao);  
    novo->prox = inicio;  
    return novo;  
}  
  
int main(){  
    No *lista = NULL;  
    for (int i=0; i<10; i++)  
        lista = setNo(lista);  
}
```



Função para Acesso aos Nós

■ Versão Iterativa...

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



Função para Acesso aos Nós

■ Versão Recursiva...

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




Exercício B

- Faça um programa modular, que realize o cadastro dinâmico de estruturas do tipo carro (ano, modelo, valor, placa e proprietário).
 - Utilize uma estrutura do tipo Lista Encadeada.
 - Implemente uma função para o cadastro de nós.
 - Implemente uma função para listagem dos carros.
 - Implemente uma função que, através da placa, imprima todos os dados de um carro.
 - Implemente uma função que retorna o valor médio dos carros cadastrados no sistema.



Exercício C

- 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 esta rotina, além do ponteiro indicando o início da lista, também é armazenado um ponteiro que sempre aponta para o **último elemento da lista**.
- As leituras se baseiam no ponteiro início, enquanto as inclusões são baseadas no ponteiro fim.
- *Altere o exercício anterior, fazendo os cadastros como uma estrutura do tipo FIFO.*