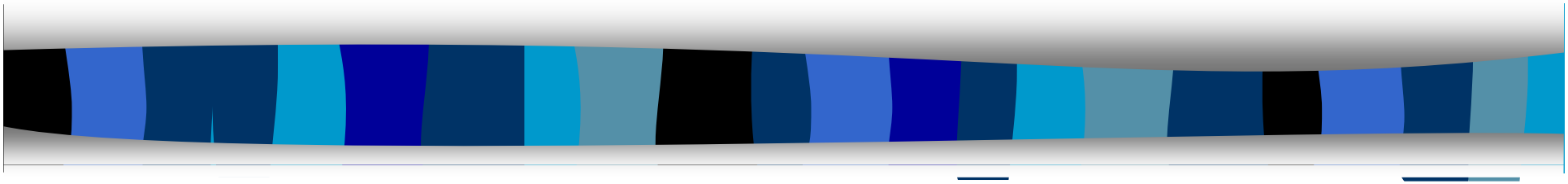


Listas Encadeadas

Definições:

- Lista Linear: sequência de zero ou mais elementos $a_1, a_2, ..., a_n$ sendo
  - a_i elementos de um mesmo tipo
  - *n* o tamanho da lista linear (*n* = 0 $\Rightarrow$ lista vazia)

- Propriedade fundamental: os elementos têm relações de ordem na lista
  - a_i precede a_{i+1} (e a_i sucede a_{i-1})
  - a_1 é o primeiro elemento da lista
  - a_n é o último elemento da lista

- Representação de Listas Lineares: TAD com elementos organizados de maneira sequencial.
  - Os tipos mais comuns de listas lineares são as *pilhas* e as *filas*

Operações:

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

Implementação:

- A implementação de uma Lista, usando uma linguagem de programação como o C, pode ser feita por meio de:

1. Alocação Seqüencial e Estática de Memória (vetores)

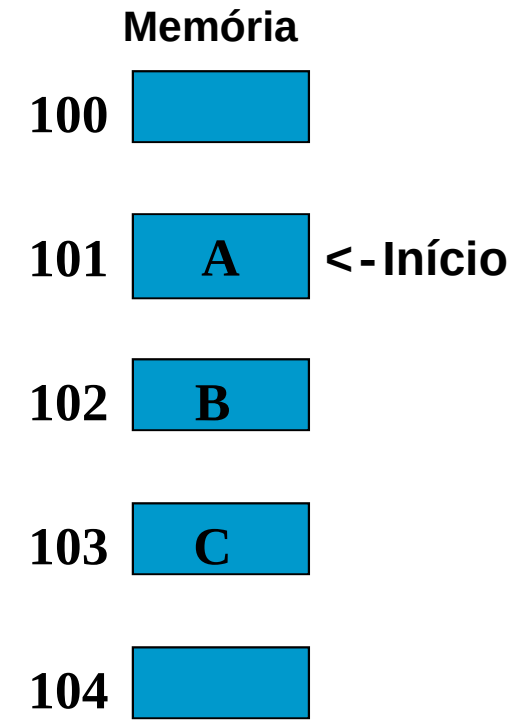
ou

2. Alocação Dinâmica de Memória (Ponteiros)

Listas Estáticas:

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

Alocação Seqüencial:



Listas Estáticas:

Características:

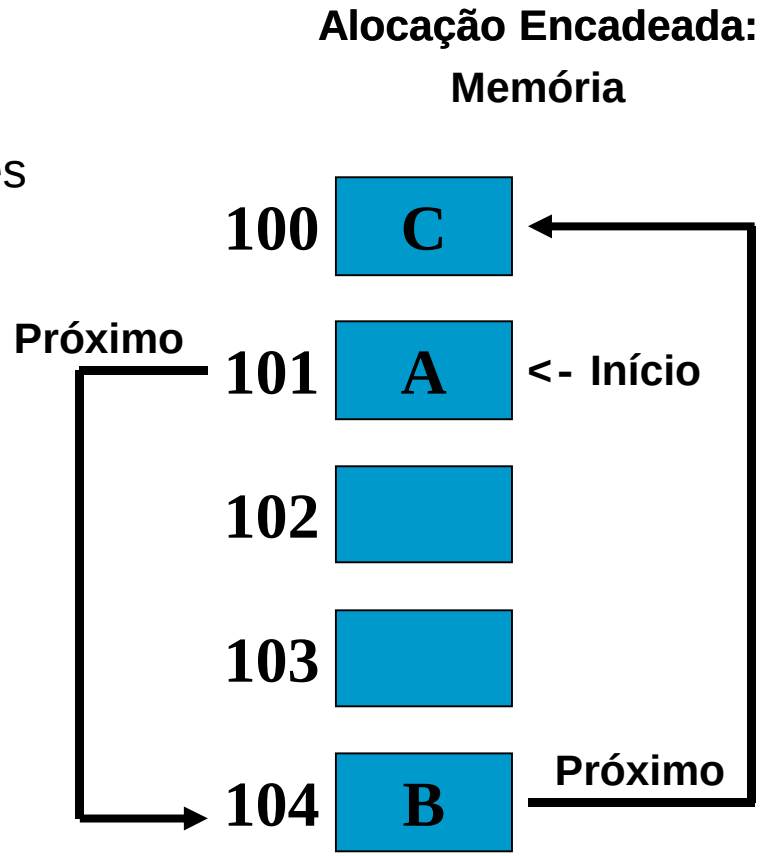
- Dados armazenados em um vetor
- Declaração estática do vetor:
  - o tamanho deve ser suficiente para alocar toda a lista, prevendo aumentos futuros
  - não podemos alterar este tamanho em tempo de execução
- A lista estará alocada em posições contíguas na memória
  - a lista sempre terá um início no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

Listas Encadeadas:

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

Listas Encadeadas:

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



Listas Encadeadas:

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

Nós de Encadeamento

- Seja o nó definido abaixo:

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

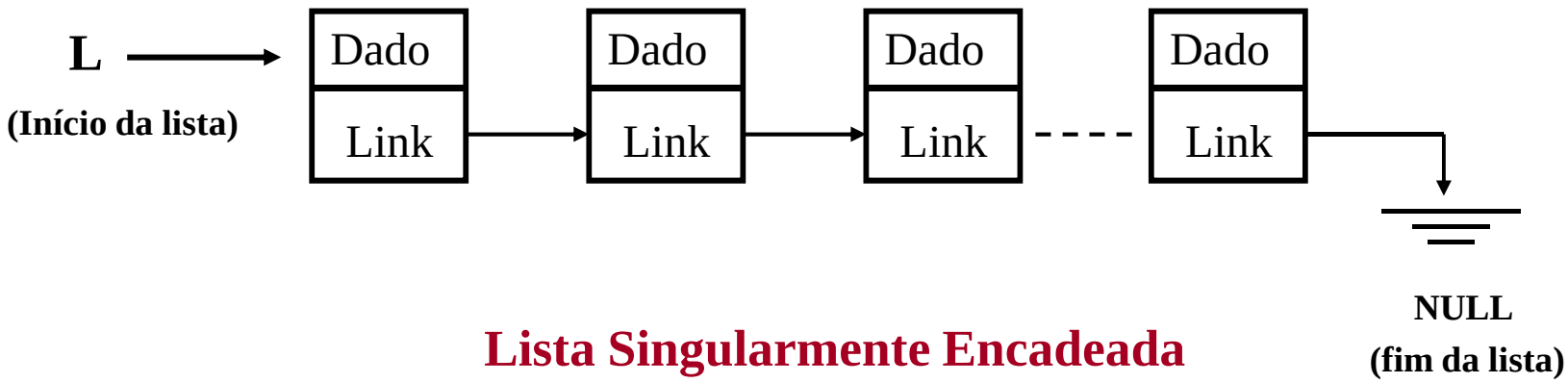
- Tal registro contém dois atributos:
  - “info” (principal): qualquer tipo conhecido (int, float, char, struct aluno, struct pessoa, etc.)
  - “prox”: ponteiro para um registro do tipo “node”. A idéia é poder apontar para o próximo registro da lista

Tipo de Encadeamento:

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

Encadeamento Simples:

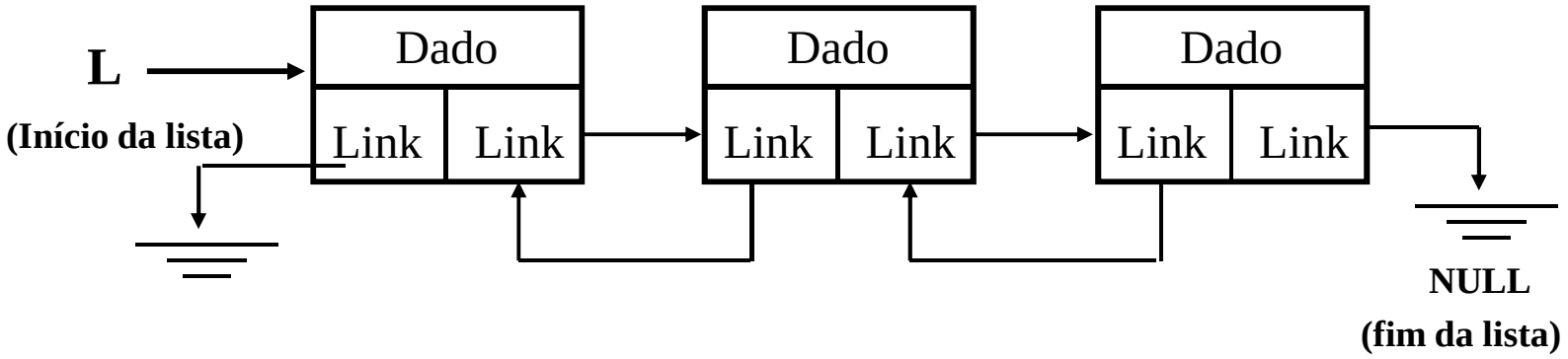
- Representação:



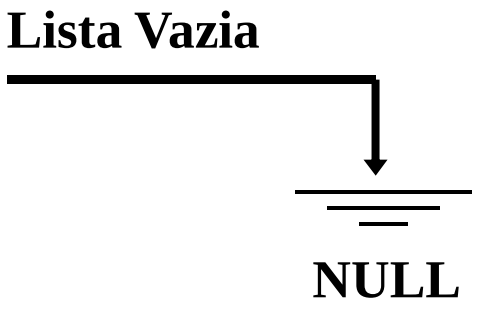
Lista Singularmente Encadeada

Encadeamento Duplo:

- Representação:

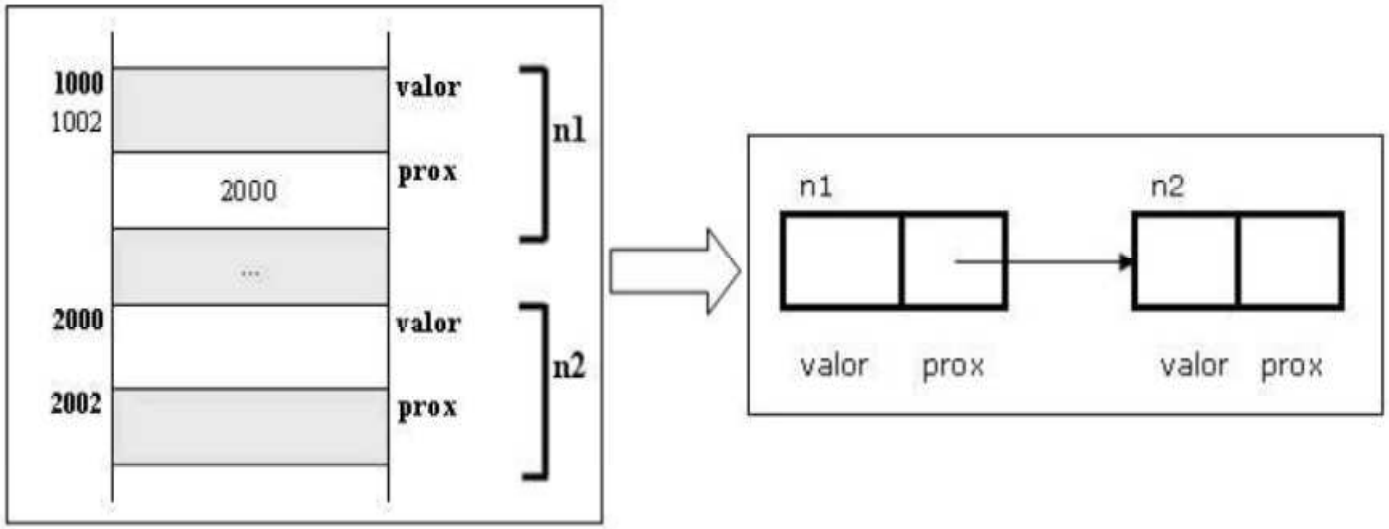


Lista Duplamente Encadeada



Encadeamento

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

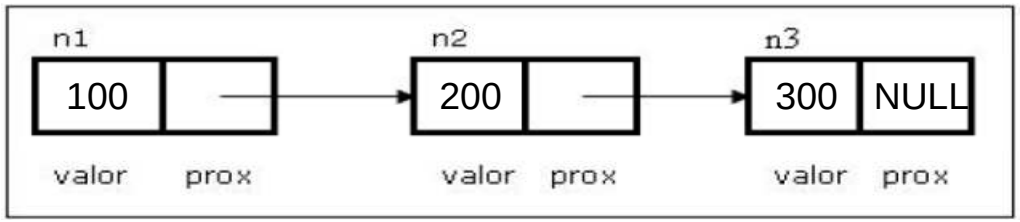


- O efeito de encadeamento pode, então, ser utilizado para a criação de uma lista, através da adição de novos nós

Encadeamento

- Considere o código abaixo:

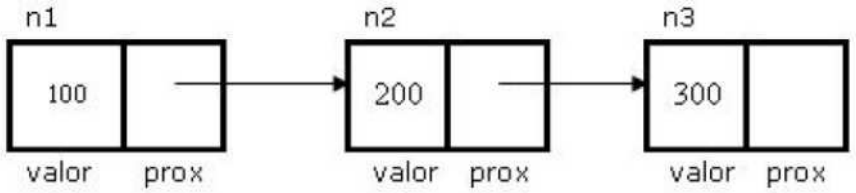
```
node n1, n2, n3;
int i;
n1.valor = 100;
n2.valor = 200;
n3.valor = 300;
n1.prox = &n2;
n2.prox = &n3;
n3.prox = NULL;
```



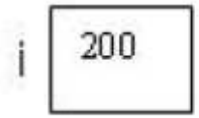
- Qual seria o resultado das instruções abaixo?

```
a.  i = (n1.prox) -> valor;
b.  i = n1.prox.valor;
c.  n1.prox = n2.prox;
```

Respostas



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

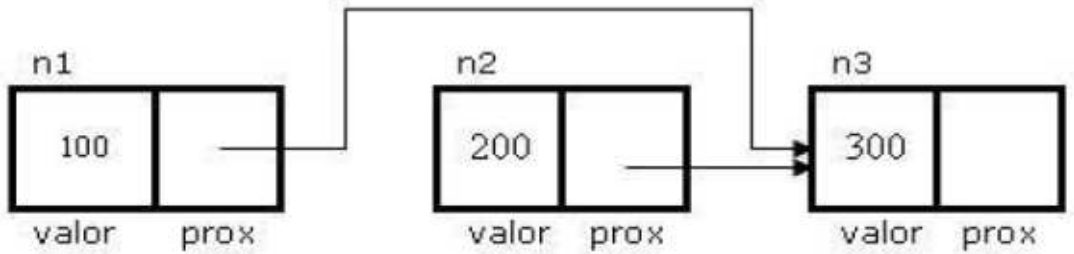


b. `i = n1.prox.valor;`

INCORRETO!

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

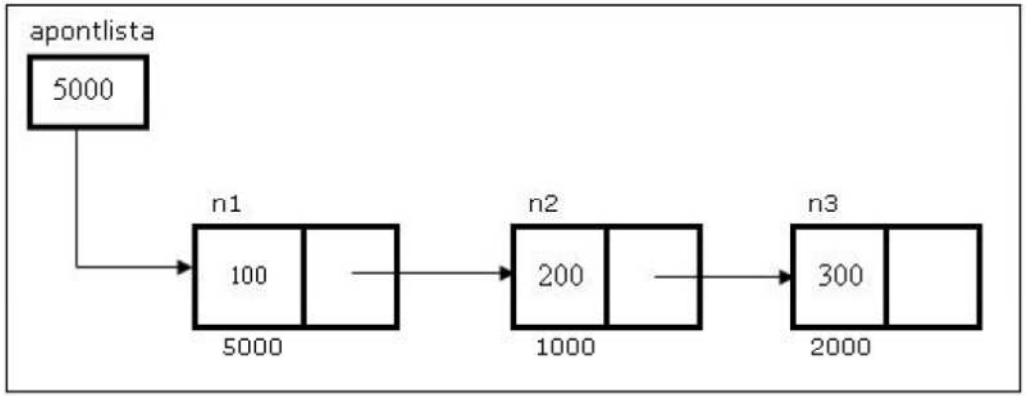
Remove o elemento n2 da lista



Considerações

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

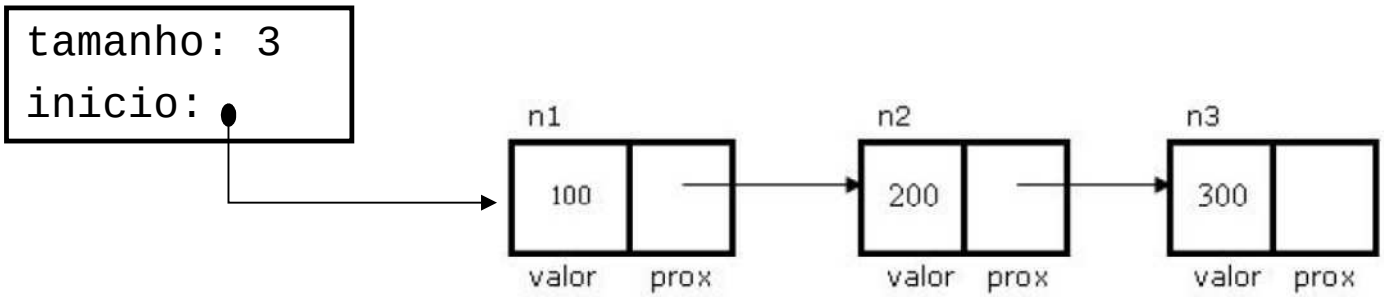
```
node *apontlista;
.
.
.
apontlista = &n1;
```



Considerações

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

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

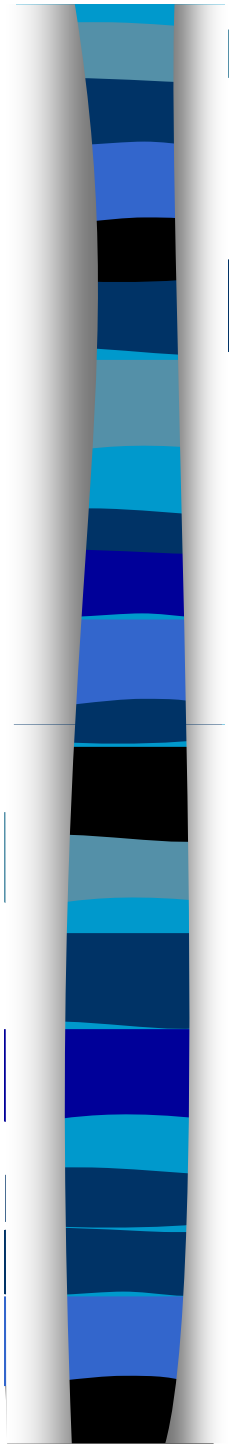


TADs sobre Listas Ligadas:

- Listas, Pilhas e Filas são tipos de TADs muito utilizados em programação
- Podem ser construídos com base em dados armazenados em listas ligadas (mesma organização dos dados)
- O funcionamento distinto de cada TAD é garantido pelas operações que dão acesso aos dados
- Considerando os tipos básicos de operação sobre listas (criação, inserção, remoção e consulta), discutiremos as diferenças fundamentais entre Lista, Pilha e Fila

Listas, Pilhas, Filas: conceitos

- Listas: organização sequencial de dados (relação de ordem). Não possui restrições de acesso para consulta ou alteração. Pode atender a critérios variados
- Pilhas: tipo de lista com critérios rígidos de acesso: permite operações em apenas uma das extremidades (topo), respeitando a política LIFO de acesso
- Fila: tipo de lista cujo critério de acesso respeita a política FIFO de acesso. Ou seja, com na Pilha, a ordem de inserção dos dados afeta a ordem de remoção

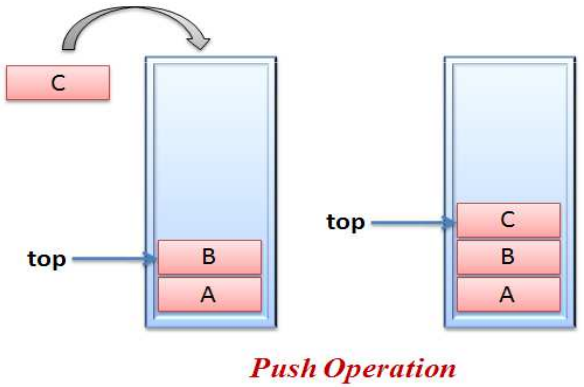


Listas, Pilhas, Filas: criação

- Todas as TADs deste módulo são criadas vazias, com ponteiros de controle apontando para NULL e contadores zerados

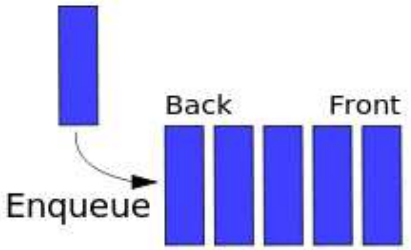
Listas, Pilhas, Filas: inserção

- Listas: inserção pode ser feita em qualquer posição, qualquer critério pode ser definido
  - Exemplo: em uma lista ordenada, um novo elemento deve ser inserido em posição apropriada (busca)
- Pilhas: a inserção só pode ser feita no Topo (*push*)



Push Operation

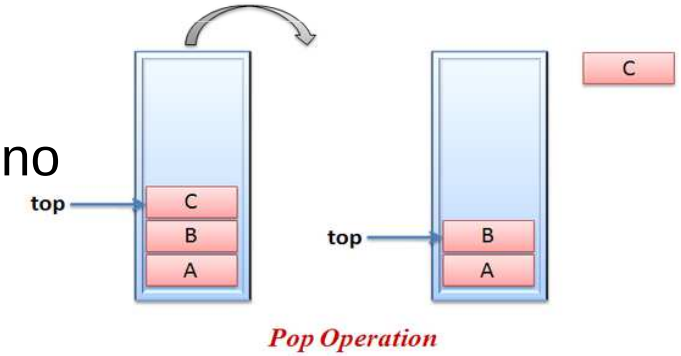
- Fila: inserção só pode ser feita em uma das extremidades, chamada Fim da Fila (*enqueue*)



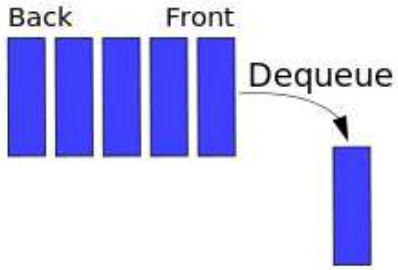
Listas, Pilhas, Filas: remoção

- Listas: qualquer elemento pode ser removido da lista. Uma chave de busca pode ser fornecida
  - Exemplo: em uma lista de clientes, um cliente específico pode ser encontrado pelo CPF e removido

- Pilhas: a remoção só pode ser feita no Topo (*pop*)



- Fila: remoção só pode ser feita na extremidade chamada de Início da Fila (*dequeue*)



Listas, Pilhas, Filas: consulta

- Listas: uma lista pode ser varrida e ter seus dados consultados em qualquer ordem
- Pilhas: apesar de ser possível varrer uma pilha, conceitualmente devemos ter acesso apenas ao Topo. Consulta em uma pilha seria verificar qual o elemento do topo, sem removê-lo (*peek*)
- Fila: assim como na Pilha, não é usual varrer uma fila, e a consulta seria verificar qual o elemento no início da fila, sem removê-lo (*front*)

Exercício (Lab 6)

11. Implemente o TAD Pilha, construído sobre lista encadeada. O TAD deve ter as operações abaico, que respeitam o conceito do tipo de dados Pilha.

a. `Pilha* criaPilha():` cria uma Pilha vazia
b. `int push(Pilha *p, float el):` empilha elemento
c. `int pop(Pilha *p, float* el):` desempilha elemento
d. `int peek(Pilha *p, float* el):` espia elemento do topo
e. `int pilha_vazia(Pilha *pilha):` verifica se a Pilha está vazia
f. `void imprimePilha(Pilha *pilha):` imprime a Pilha toda (função de teste)