Lista

Dados

Podemos manipular qualquer tipo de dado na Lista (tipos primitivos, ponteiros, structs). Além dos elementos contidos na lista, também precisamos armazenar os dados responsáveis por gerenciar o comportamento da lista. Esses dados dependem da estratégia que será utilizada na implementação da estrutura. Para isso, vamos definir uma struct para encapsular esses dados de controle.

```
typedef struct {

// Aqui serão declarados os atributos necessários para controlar a lista

}Lista;
```

Vamos ver esses atributos mais adiante, quando discutirmos sobre as estratégias de implementação.

Comportamento

Primeiro, vamos entender o comportamento de uma lista e suas funcionalidades. As operações essenciais de uma lista são:

- (lista_criar) Criar uma lista
- (lista_destruir) Destruir uma lista
- (lista_anexar) Anexar (colocar no fim) um elemento na lista;
- (lista_inserir) Inserir um elemento em qualquer posição na lista;
- (lista_removerPosicao)Remover um elemento de uma determinada posição da lista;
- (lista_removerElemento) Remover um elemento específico independente da sua posição na lista;
- (lista_substituir) Substituir o valor de um elemento a partir de sua posição na lista;
- (lista_posicao) Obter a posição na lista de um determinado elemento;
- (lista_buscar) Ter acesso ao elemento que se encontra em uma determinada posição;
- (lista_tamanho) Obter a quantidade de elementos contidos na lista;
- (lista_toString) Obter uma versão *string* da lista (Ex: "[1,2,3]");

Com base na struct que representa a **Lista**, vamos traduzir cada operação em um protótipo de função

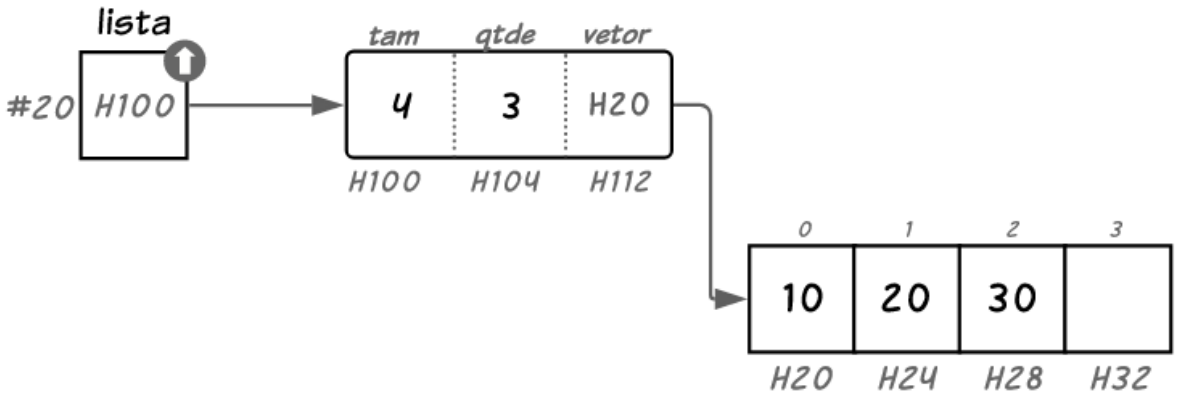
```
Lista* lista_criar();
void lista_destruir(Lista** endereco);

bool lista_anexar(Lista* l, TipoElemento elemento);
bool lista_inserir(Lista* l, TipoElemento elemento, int posicao);

bool lista_removerPosicao(Lista* l, int posicao, TipoElemento* endereco);
int lista_removerElemento(Lista* l, TipoElemento elemento);

bool lista_substituir(Lista* l, int posicao, TipoElemento novoElemento);
int lista_posicao(Lista* l, TipoElemento elemento);
bool lista_buscar(Lista* l, int posicao, TipoElemento* endereco);

int lista_tamanho(Lista* l);
bool lista_vazia(Lista* l);
bool lista_toString(Lista* l, char* str);
```

Estratégias de Implementação

Podemos utilizar basicamente dois tipos de estratégias de organização dos elementos na pilha: organização **contígua** ou **encadeada**. Na organização contígua, os elementos da pilha são organizados em um espaço contíguo na memória, ou seja, os elementos ficam fisicamente um ao lado do outro na memória. Na organização encadeada, os elementos são acomodados em espaços distintos da memória, mas vinculados por meio de ponteiros.

Organização contígua

Nesta estratégia, precisaremos de três atributos: o endereço do bloco contíguo de memória com capacidade de armazenar todos os elementos da lista (vetor), um atributo para armazenar o tamanho desse bloco/vetor (tam) e outro para armazenar a quantidade de elementos armazenados na lista (qtde). Portanto, a definição da *struct* **Lista** **com a estratégia de organização contígua ficará da seguinte forma:



```c
typedef struct {
  TipoElemento* vetor;
  int tam;
  int qtde;
}Lista;
```

Organização encadeada

Assim como nas outras estruturas, também teremos que criar a *struct* no para modelar o bloco de memória que armazenará o elemento. No entanto, ao invés de guardar somente o endereço do nó seguinte, também vamos armazenar o endereço do nó anterior. Essa modificação torna a estrutura **duplamente encadeada** e permitirá a navegação em ambas as direções no encadeamento.

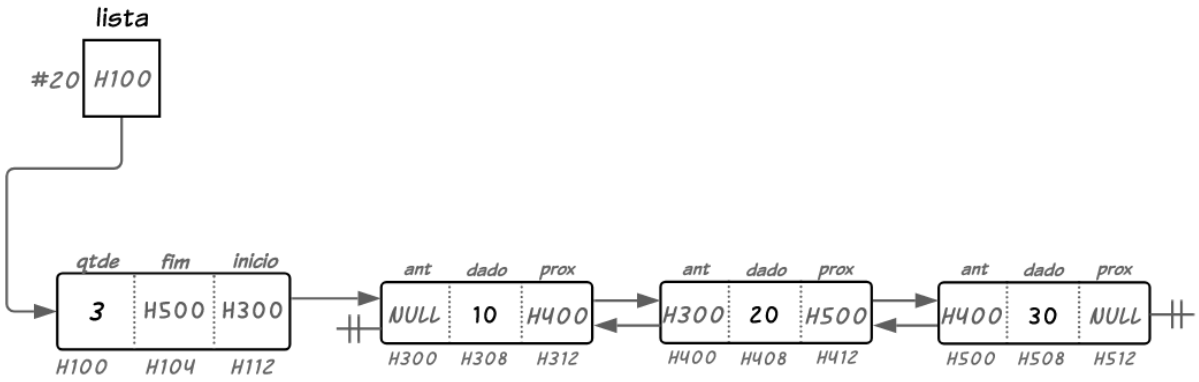
```c
typedef struct no{
  TipoElemento dado;
  struct no    *ant;
  struct no    *prox;
}No;
```

Na organização encadeada, ainda podemos optar por 2 estratégias de implementação: com ou sem nó sentinela.

Estrutura SEM nó sentinela

Os atributos da *struct* com a implementação sem nó sentinela é idêntica a *struct* utilizada na Fila. Um ponteiro para o primeiro nó (para a navegação no sentido esquerda/direita), um ponteiro para o último nó (para navegação sentido direita/esquerda) e a quantidade de elementos contidos na estrutura.

```c
typedef struct {
  No *inicio;
  No *fim;
  int qtde;
}Lista;
```



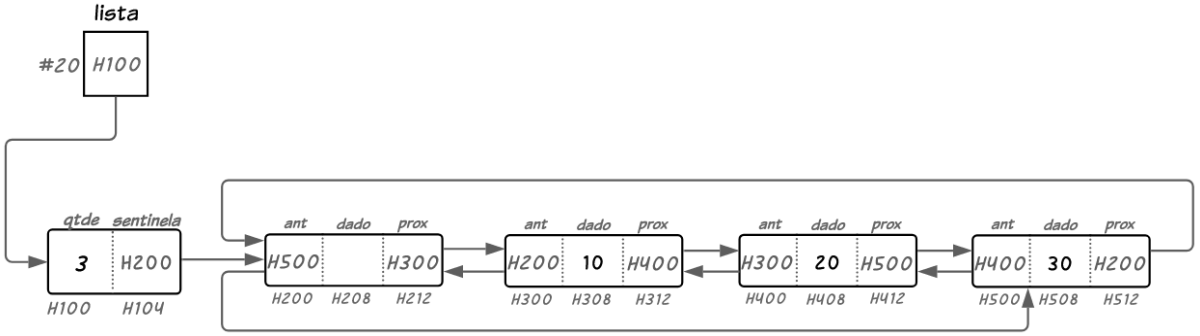
Estrutura COM nó sentinela

Nessa estratégia, teremos um único ponteiro de Nó (sentinela) que assumirá a responsabilidade de armazenar o endereço do primeiro e último nó da estrutura. Os atributos do nó sentinela tem os seguintes significados:

- prox: armazena o endereço do primeiro nó da lista;
- ant: armazena o endereço do último nó da lista;
- dado: não é utilizado

```
typedef struct {
  No *sentinela;
  int qtde;
}Lista;
```

Independente da estratégia utilizada, o comportamento (resultado) das funções deverá ser o mesmo. No entanto, a implementação é dependente da estratégia que você for utilizar.



Atividade

Implemente as funcionalidades especificadas para as a estratégia **encadeada**. Você encontra os arquivos no repositório da disciplina

Considerar os seguintes cenários na **Inserção**

- Inserção na Lista vazia. Preciso mexer nos ponteiros **inicio** e **fim**
- Inserção na última posição. Preciso mexer no ponteiro **fim**
- Inserção na primeira posição. Preciso mexer no ponteiro **inicio**
- Inserção no meio. Não preciso mexer em nenhum dos ponteiros, mas eu preciso encontrar a posição correta da inserção.

Considerar os seguintes cenários na **Remoção**

- Remoção na Lista com um único elemento. Preciso mexer nos ponteiros **inicio** e **fim**
- Remoção da última posição. Preciso mexer no ponteiro **fim**
- Remoção da primeira posição. Preciso mexer no ponteiro **inicio**
- Remoção no meio. Não preciso mexer em nenhum dos ponteiros, mas eu preciso encontrar a posição correta da remoção.