# AED1 - Aula 11
# Listas encadeadas

Um lista (ou sequência) é uma coleção de itens que apresenta uma ordem estável. Queremos que nossas listas aceitem certas operações básicas:

- Seleção, pegar o conteúdo do k-ésimo item,
- Busca, encontrar um item pelo seu conteúdo,
- Inserção, inserir um item na posição k,
- Remoção, remover um item da posição k.

Já vimos como implementar listas em vetores contíguos:

- Seleção custa O(1),
- Busca custa O(n),
- Inserção custa O(n - k),
- Remoção custa O(n - k).

Agora, vamos ver como implementar listas encadeadas,

- usaremos registros, apontadores e alocação dinâmica,
- analisaremos seus prós e contras.

**Lista encadeada**

Usa células que correspondem a registros (structs) contendo:

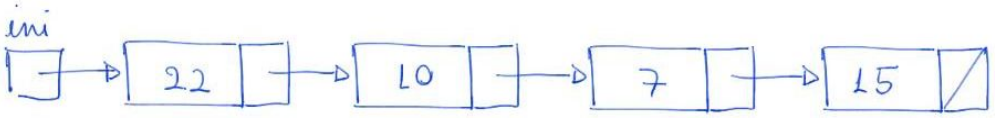
- um campo conteúdo (conteudo),
- um campo apontador para outra célula (prox).

```
typedef struct celula Celula;
struct celula
{
   int conteudo;
   Celula *prox;
};

Celula *ini = NULL; // lista vazia
```


Podemos definir uma lista encadeada de modo recursivo como sendo:

- um apontador nulo (NULL) - lista vazia,
- uma célula cujo campo prox é uma lista.



Eficiência de espaço:

- Sobre o uso de memória, vale destacar que listas encadeadas gastam mais memória por elemento do que vetores
    - isso porque cada elemento tem um campo apontador alocado
- Por outro lado, listas gastam memória proporcional ao número de elementos
    - enquanto vetores podem exigir pré-alocação de grandes quantidades de memória, causando desperdício

Imprime conteúdo de uma lista

```c
void imprime(Celula *lst)
{
   Celula *p = lst;
   while (p != NULL)
   {
       printf("%d ", p->conteudo);
       p = p->prox;
   }
   printf("\n");
}
```

- exemplo de uso

```c
   imprime(ini);
```

**Operações e eficiência**

Busca:

- encontrar um elemento leva tempo O(n)

```c
Celula *busca(Celula *lst, int x)
{
   Celula *p = lst;
   while (p != NULL && p->conteudo != x)
       p = p->prox;
   return p;
}
```

- exemplo de uso

```c
   Celula *p = busca(ini, 10);
```

Seleção:

- pegar o conteúdo do k-ésimo leva tempo O(k)

```c
Celula *selecao(Celula *lst, int k)
{
   Celula *p = lst;
   int q = 0;
   while (p != NULL && q < k)
   {
       p = p->prox;
```

```
        q++;
    }
    return p;
}
```

- exemplo de uso

```
Celula *q = selecao(ini, 10);
```

Inserção: inserir um elemento no início da lista, ou na frente de uma célula para a qual já temos um apontador, leva tempo constante

```
void insereErrado1(Celula *lst, int x)
{
    Celula nova;
    nova.conteudo = x;
    nova.prox = lst;
    lst = &nova;
}
```

- o erro ocorre porque, como a nova célula foi alocada estaticamente
    - sua memória é desalocada quando a função insereErrado termina

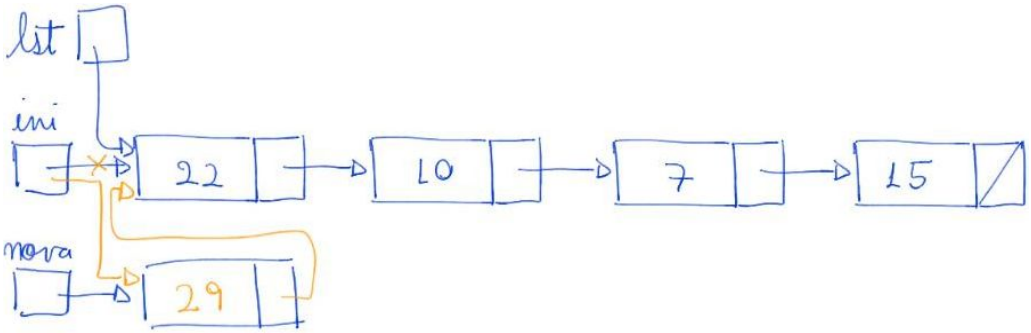
```
void insereErrado2(Celula *lst, int x)
{
    Celula *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = x;
    nova->prox = lst;
    lst = nova;
}
```

- o erro ocorre porque a variável lst também é local
    - por isso, modificar seu conteúdo não muda a lista original

```
Celula *insere1(Celula *lst, int x)
{
    Celula *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = x;
    nova->prox = lst;
    return nova;
}
```

- o uso correto desta função exige que a lista passada como parâmetro
    - receba a lista que a função devolve

```
ini = insere1(ini, x);
```

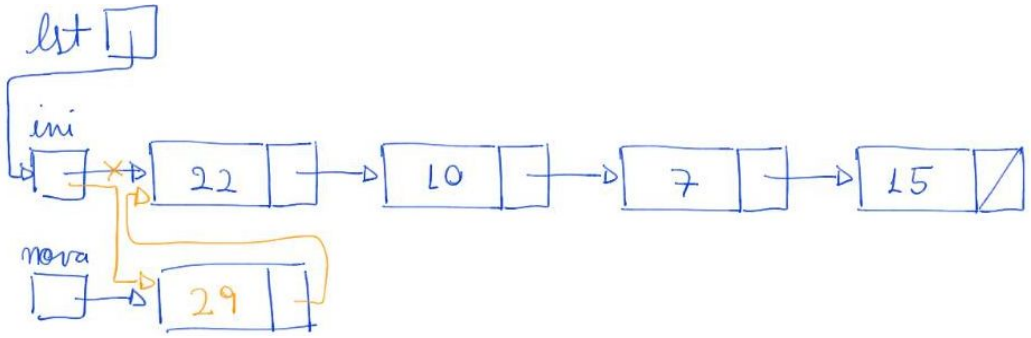



- outra maneira de implementar a inserção é recebendo um apontador de apontador

```
void insere2(Celula **lst, int x)
{
   Celula *nova;
   nova = malloc(sizeof(Celula));
   nova->conteudo = x;
   nova->prox = *lst;
   *lst = nova;
}
```

- assim a função consegue alterar o endereço contido na lista original
    - para isso recebe o endereço do apontador da lista como parâmetro

```
insere2(&ini, i);
```



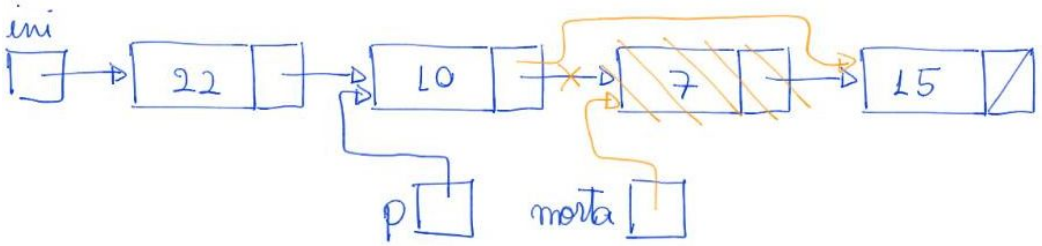

Remoção:

- remover da lista a célula seguinte leva tempo constante

```
// remove a celula sucessora de p
// supõe que p != NULL e p->prox != NULL
void remove1(Celula *p)
{
   Celula *morta;
   morta = p->prox;
   p->prox = morta->prox;
   free(morta);
}
```

- exemplos de uso

```
   remove1(ini);
   remove1(ini->prox);
```

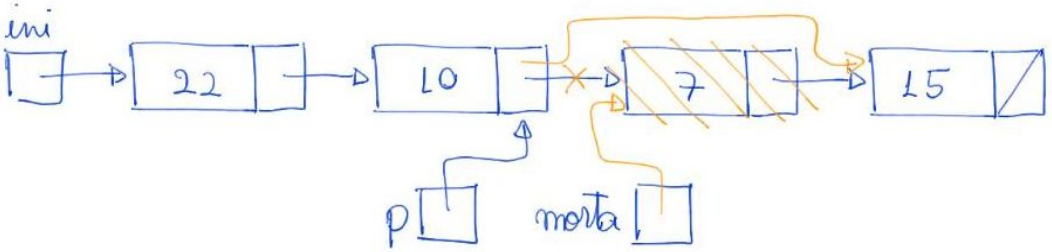



- mas como remover o primeiro elemento da lista?

```
// remove a celula apontada por *p
// supõe que *p != NULL
void remove2(Celula **p)
{
   Celula *morta;
   morta = *p;
   *p = morta->prox;
   free(morta);
}
```

- exemplos de uso

```
   remove2(&ini);
   remove2(&ini->prox);
```




```
// remove a celula apontada por p
// supõe que p != NULL
Celula *remove3(Celula *p)
{
   Celula *morta;
   morta = p;
   p = morta->prox;
   free(morta);
   return p;
}
```

- exemplos de uso

```
   ini = remove3(ini);
   ini->prox = remove3(ini->prox);
```

Busca e insere:

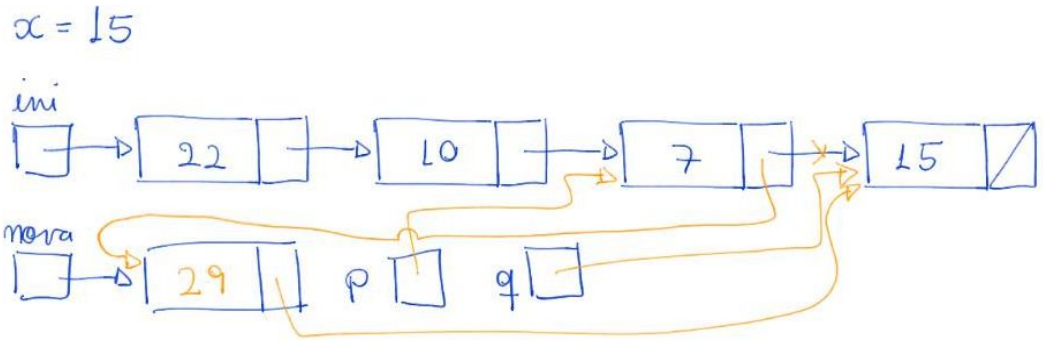
- buscar um elemento x para inserir y logo antes dele leva tempo O(n)

```
// busca x na lista lst e insere y logo antes de x
// se x não está na lista insere y no final
Celula *buscaInsere(Celula *lst, int x, int y)
{
```

```
    Celula *p, *q, *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = y;
    if (lst == NULL || lst->conteudo == x)
    {
        nova->prox = lst;
        return nova;
    }
    p = lst;
    q = p->prox;
    while (q != NULL && q->conteudo != x)
    {
        p = q;
        q = p->prox;
    }
    p->prox = nova;
    nova->prox = q;
    return lst;
}
```

- exemplo de uso

```
ini = buscaInsere1(ini, 15, 17);
```




- segue versão que manipula apontadores para não precisar devolver endereço da nova lista

```
// busca x na lista lst e insere y logo antes de x
// se x não está na lista insere y no final
void buscaInsere2(Celula **lst, int x, int y)
{
    Celula **p, *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = y;
    p = lst;
    while (*p != NULL && (*p)->conteudo != x)
        p = &(*p)->prox;
    nova->prox = *p;
    *p = nova;
}
```

- exemplo de uso

```
buscaInsere2(&ini, 15, 17);
```