Lista ligada

Uma lista (ou sequência) é uma coleção de itens que apresenta uma ordem estável.

Supondo uma lista com n elementos, queremos que ela aceite as operações:

- Imprimir: percorrer em ordem imprimindo cada elemento.
- Seleção: pegar o conteúdo do k-ésimo item.
    - Temos que k \in [0, n), i.e., k \in {0, 1, 2, …, n - 1}.
- Busca: encontrar um item pelo seu conteúdo.
- Inserção: inserir um item no início da lista ou na frente de uma célula.
- Remoção: remover a primeira célula da lista ou a célula seguinte.

Vamos ver como implementar listas ligadas.

- Usaremos registros, apontadores e alocação dinâmica.
    - Analisaremos seus prós e contras.

Uma lista ligada usa células que correspondem a estruturas (structs) contendo:

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


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

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

Podemos definir uma lista encadeada de modo recursivo como sendo:

- Um apontador nulo (NULL), i.e., lista vazia,
- ou 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.
- Por outro lado, listas gastam memória proporcional ao número de elementos,
    - enquanto vetores podem exigir pré-alocação
        - de grandes quantidades de memória, causando desperdício.

Operações, implementações e eficiência:

Imprimir: percorrer em ordem imprimindo cada elemento.

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

- Exemplo de uso

```
imprime(ini);
```

- Eficiência de tempo: linear no tamanho da lista, i.e., O(n).

Busca: encontrar um item pelo seu conteúdo x.

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

    - Quiz1: como esse algoritmo indica que não encontrou?
- Exemplo de uso

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

- Eficiência de tempo: O(n) no pior caso.

Seleção: pegar o conteúdo do k-ésimo item.

```
Celula *selecao(Celula *lst, int k) {
    Celula *p = lst;
    int pos = 0;
    while (p != NULL && pos < k) {
        p = p->prox;
        pos++;
    }
    return p;
}
```

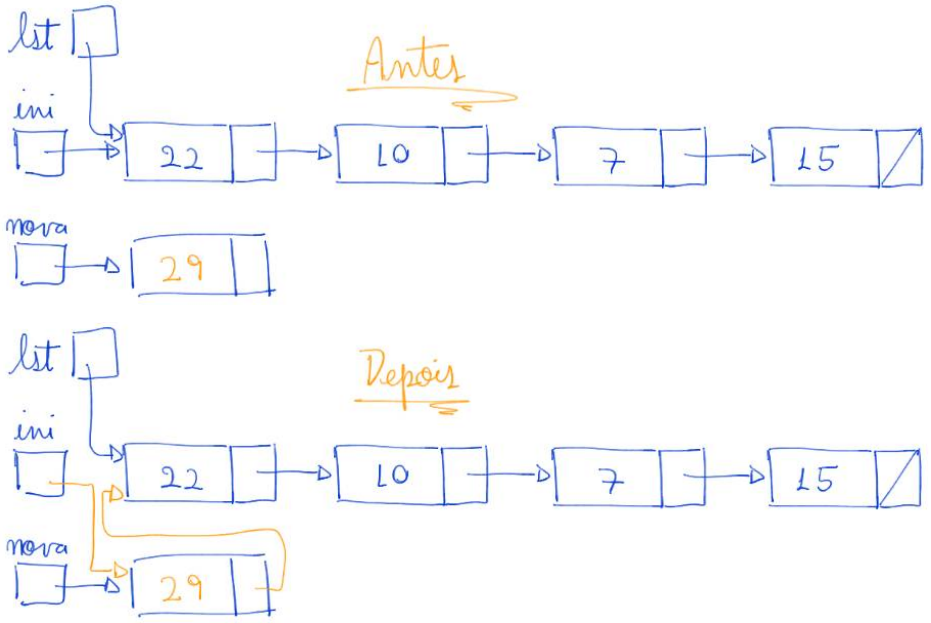
    - Quiz2: o que acontece se k for maior que o tamanho da lista?
- Exemplo de uso

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

- Eficiência de tempo: leva tempo O(k).

Inserção: inserir um item x no início da lista,

- ou na frente de uma célula para a qual já temos um apontador.



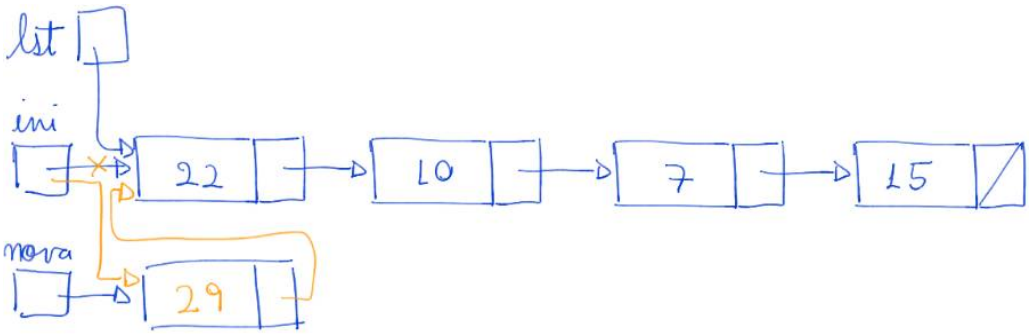

```c
void insereErrado1(Celula *lst, int x) {
    Celula nova;
    nova.conteudo = x;
    nova.prox = lst;
    lst = &nova;
}
```

- Um erro ocorre porque, como a nova célula foi alocada estaticamente
    - sua memória é desalocada quando a função insereErrado1 termina.

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

- O erro ocorre porque o parâmetro/variável lst também é local.
    - Com isso, modificar seu conteúdo não muda a lista original.

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

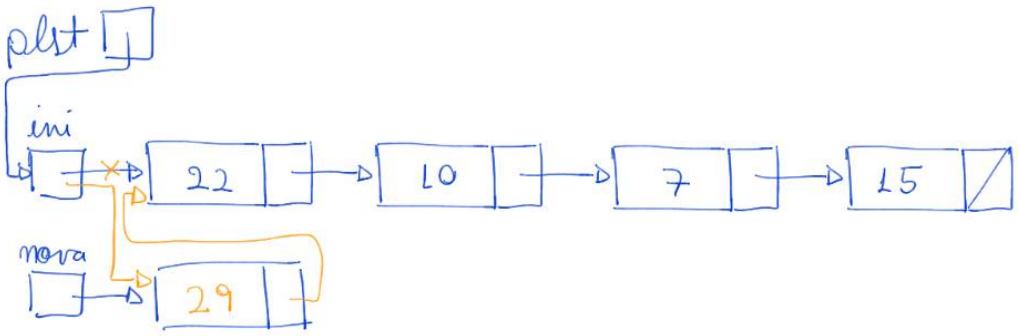


- O uso correto desta função exige que a lista passada como parâmetro
  - receba a lista que a função devolve.

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

- Outra maneira correta de implementar a inserção é
  - recebendo um apontador de apontador.

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

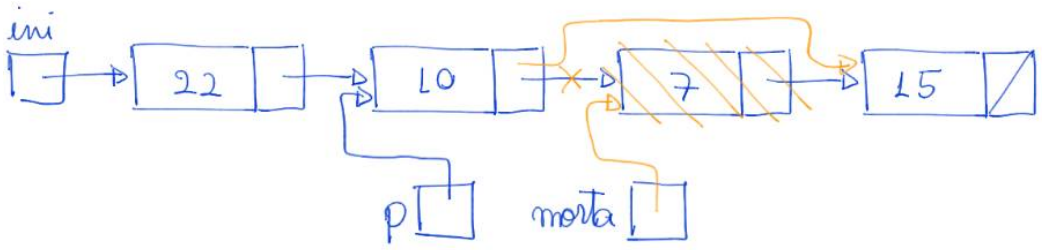


- O uso correto desta função exige que o endereço do apontador da lista
  - seja passado como parâmetro, para que a função possa alterar
    - o endereço contido na lista original.

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

- Eficiência de tempo: constante, i.e., O(1).

Remoção: remover da lista a célula seguinte.




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

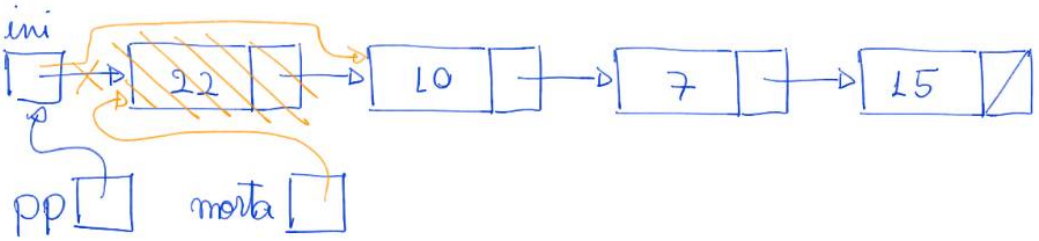
- Exemplos de uso

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

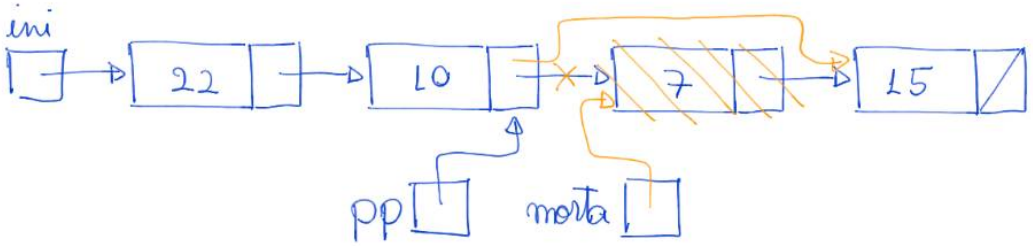
- Eficiência de tempo: constante, i.e., O(1).

Mas, como remover o primeiro elemento da lista?

- Podemos usar apontador de apontador.




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



- Exemplos de uso

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

Outra maneira correta de implementar a remoção do primeiro é

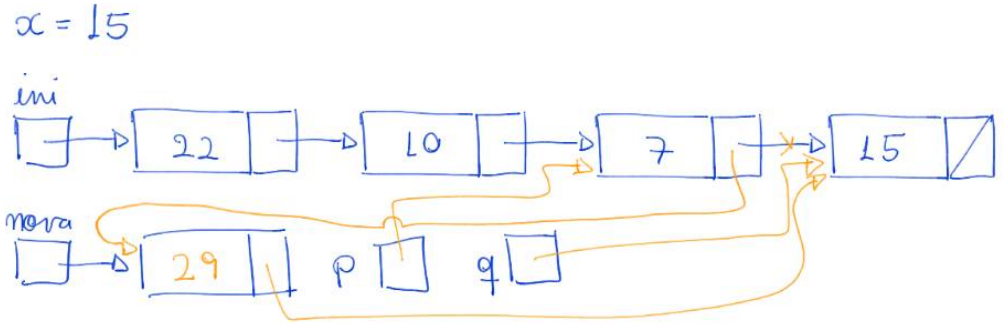
- devolvendo o novo endereço da lista.

```
// 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 e inserir y logo antes dele leva tempo O(n).

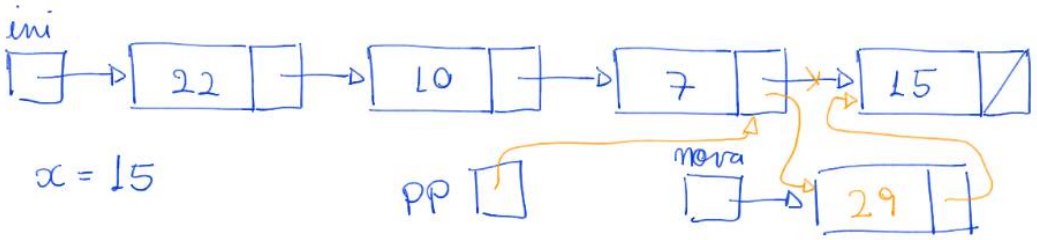



```
// busca x na lista lst e insere y logo antes de x
// se x não está na lista insere y no final
Celula *buscaInsere1(Celula *lst, int x, int y) {
    Celula *p, *q, *nova; // q sempre aponta para p->prox
    nova = malloc(sizeof(Celula));
    nova->conteudo = y;
    if (lst == NULL || lst->conteudo == 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, 2, 3);
```

- Outra maneira correta de implementar busca e insere é
    - recebendo um apontador de apontador.




```
// 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 **plst, int x, int y) {
    Celula **pp, *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = y;
    pp = plst;
    while (*pp != NULL && (*pp)->conteudo != x)
        pp = &(*pp)->prox;
    nova->prox = *pp;
    *pp = nova;
}
```

- Exemplo de uso

```
buscaInsere2(&ini, 2, 3);
```

Quiz3: Implemente a função busca e remove.

- Pode ser devolvendo a lista resultante ou usando apontador de apontador.

Quiz4: Como modificar as operações para manter a lista em ordem crescente?

- Em particular, implementar uma função que faz inserção ordenada.
    - Qual a eficiência das operações nesse caso?