AED1 - Aula 09
Lista em vetor, lista ligada

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

Queremos que nossas listas aceitem certas operações:

- 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.

Um vetor é uma estrutura de dados que armazena uma sequência de objetos

- do mesmo tipo em posições consecutivas da memória.
    - Por isso é bastante natural usar vetores para implementar listas.
- Veremos como implementar as seguintes operações em um vetor:
    - Pegar o conteúdo do k-ésimo item.
    - Buscar um item pelo seu conteúdo.
    - Inserir um item na posição k.
    - Remover um item da posição k.

Implementar lista em vetor

Usar um vetor v de tamanho TAM_MAX

```
#define TAM_MAX 1000000
```

O vetor pode ser declarado:

- estaticamente
    ```
    int v[TAM_MAX];
    ```
- dinamicamente
    ```
    int *v = (int *)malloc(TAM_MAX * sizeof(int));
    ```

Operações, implementações e eficiência:

Pegar o conteúdo do k-ésimo item leva tempo constante, i.e., O(1).

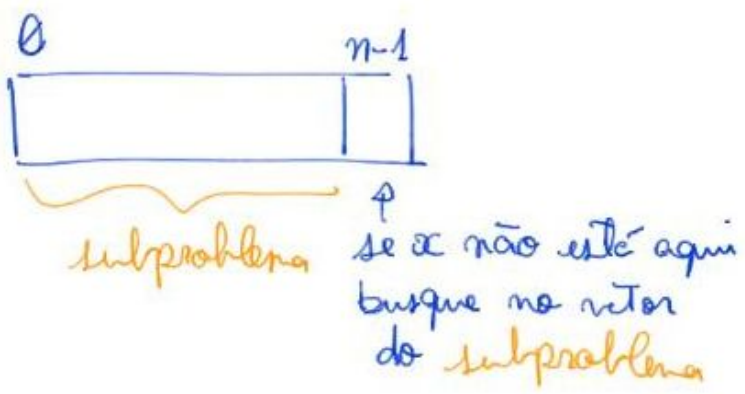
```
int selecao(int v[], int n, int k)
{
    return v[k];
}
```

Buscar um item x (iterativa ou recursivamente) leva tempo O(n) no pior caso.

- Ideia do algoritmo iterativo:
    - Percorrer o vetor verificando cada posição.

```c
int buscaI(int v[], int n, int x)
{
    int k;
    k = n - 1;
    while (k >= 0 && v[k] != x)
        k -= 1;
    return k;
}
```

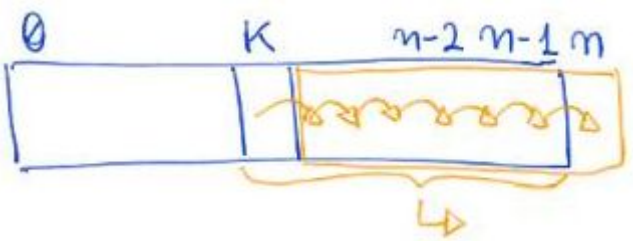
- Ideia do algoritmo recursivo:
    - Se o item buscado não é o último elemento do vetor corrente,
        - busque recursivamente no subvetor com um elemento a menos.




```c
int buscaR(int v[], int n, int x)
{
    if (n == 0)
        return -1;
    if (x == v[n - 1])
        return n - 1;
    return buscaR(v, n - 1, x);
}
```

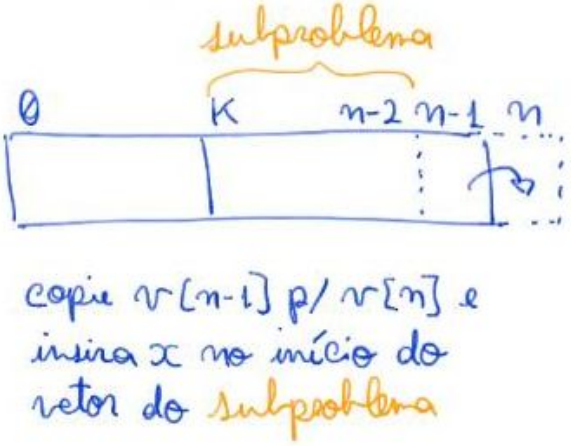
Inserir um item x na posição k leva tempo O(n - k), que é O(n) no pior caso.

- Ideia do algoritmo iterativo:
    - Deslocar itens à direita da posição k uma posição para a direita.
        - Note que a ordem deste deslocamento faz diferença.
    - Então inserir na posição k, que foi liberada.




```c
int insereI(int v[], int n, int x, int k)
{
    int j;
    for (j = n; j > k; j--)
        v[j] = v[j - 1];
    v[k] = x;
    return n + 1;
}
```

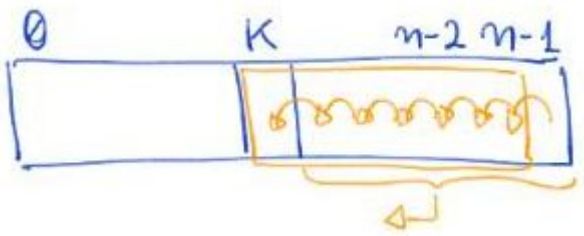
- Ideia do algoritmo recursivo:
    - Copie v[n - 1] para v[n] e insira recursivamente
        - no subvetor com um elemento a menos.




```c
int insereR(int v[], int n, int x, int k)
{
    if (k == n)
        v[n] = x;
    else
    {
        v[n] = v[n - 1];
        insereR(v, n - 1, x, k);
    }
    return n + 1;
}
```

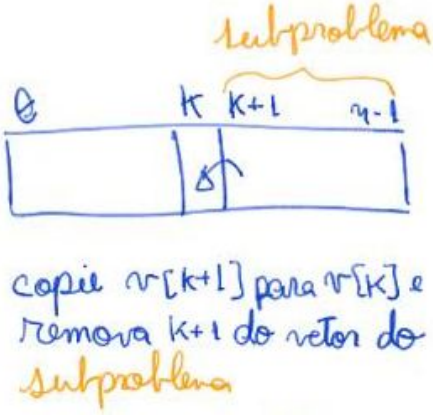
Remover o item da posição k leva tempo O(n - k), que é O(n) no pior caso.

- Ideia do algoritmo iterativo:
    - Deslocar itens à direita da posição k uma posição para a esquerda.
        - Note que a ordem deste deslocamento faz diferença.




```c
int removeI(int v[], int n, int k)
{
    int j;
    for (j = k + 1; j < n; j++)
        v[j - 1] = v[j];
    return n - 1;
}
```

- Ideia do algoritmo recursivo:
    - Copie v[k + 1] para v[k] e remova recursivamente o k + 1
        - do subproblema de tamanho n - (k + 1).




```c
int removeR(int v[], int n, int k)
{
    if (k == n - 1)
        return n - 1;
    v[k] = v[k + 1];
    return removeR(v, n, k + 1);
}
```

Bônus:

- Considere o problema de remover todas as ocorrências de um elemento x.

```
int removeTodos(int v[], int n, int x)
{
    int k;
    while ((k = buscaI(v, n, x)) != -1)
        n = removeI(v, n, k);
    return n;
}
```

    - Qual a eficiência de tempo de pior caso de removeTodos?
- Considere o seguinte algoritmo para o mesmo problema.

```
int removeTodos2(int v[], int n, int x)
{
    int i = 0, j;
    for (j = 0; j < n; j++)
        if (v[j] != x)
        {
            v[i] = v[j];
            i++;
        }
    return i;
}
```

    - Qual a eficiência de tempo de pior caso de removeTodos2?
    - Como mostrar que a função anterior está correta?
        - Qual seu invariante principal?

Sintetizando, 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 ligadas.

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

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.

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

```
imprime(ini);
```

Operações, implementações e eficiência:

Busca, encontrar um elemento leva tempo O(n) no pior caso.

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

- Exemplo de uso

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

Seleção, pegar o conteúdo do k-ésimo item leva tempo O(k).

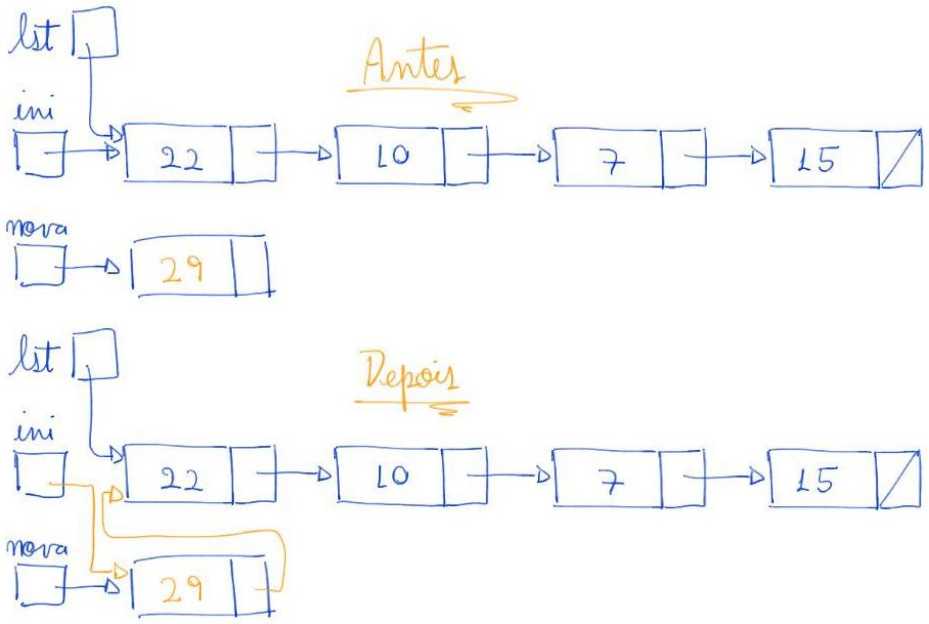
```
Celula *selecao(Celula *lst, int k)
{
    Celula *p = lst;
    int pos = 0;
    while (p != NULL && pos < k)
    {
        p = p->prox;
        pos++;
    }
    return p;
}
```

- 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, i.e., O(1).




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

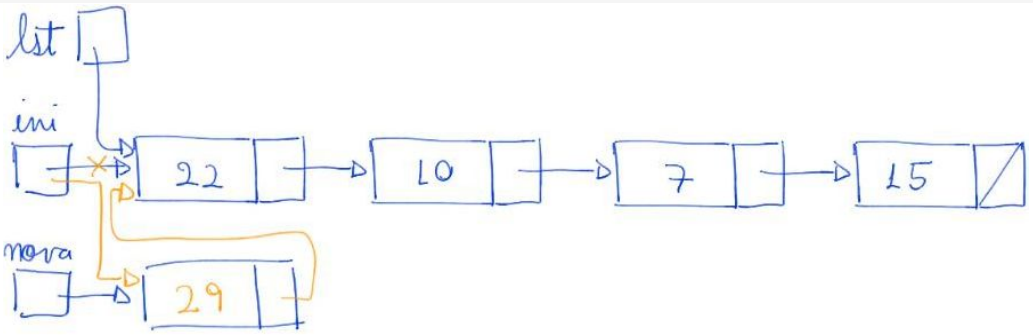
- Um erro ocorre porque, como a nova célula foi alocada estaticamente
    - sua memória é desalocada quando a função insereErrado1 termina.

```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;
}
```

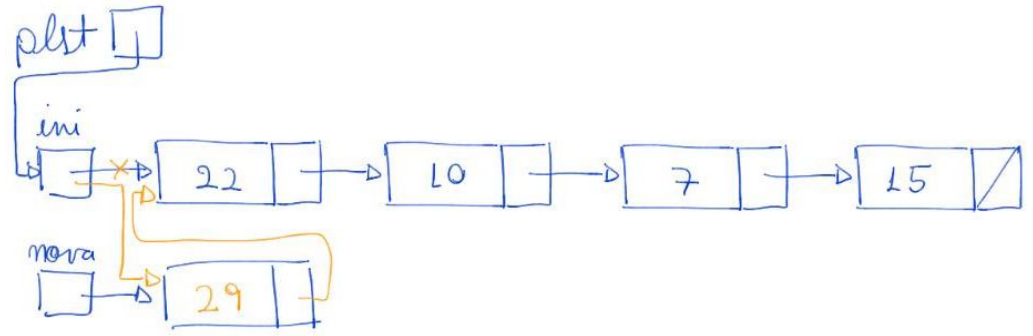


- O uso correto desta função exige que a lista passada como parâmetro
    - receba a lista que a função devolve.

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

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

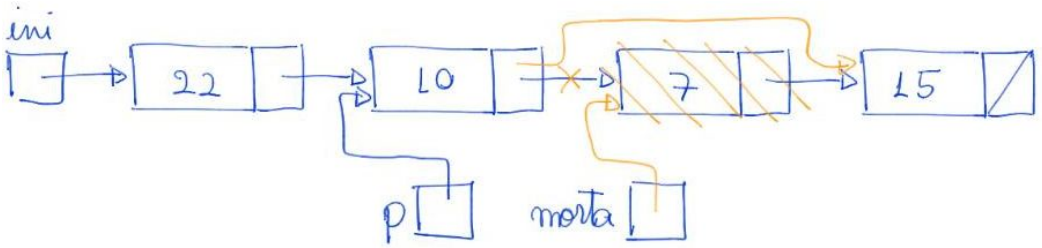
```
void insere2(Celula **plst, int x)
{
    Celula *nova;
    nova = malloc(sizeof(Celula));
    nova->conteudo = x;
    nova->prox = *plst;
    *plst = nova;
}
```



- 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);
```

Remoção, remover da lista a célula seguinte leva tempo constante, i.e., O(1).



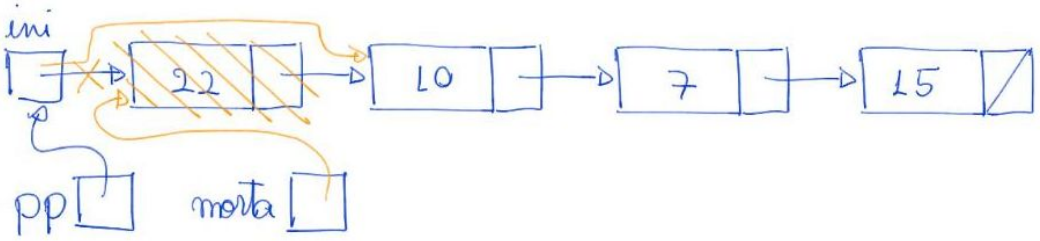

```c
// remove a celula sucessora de p
// supõe que p != NULL e p->prox != NULL
void remove1(Celula *p)
{
    Celula *morta;
    morta = p->prox;
    p->prox = morta->prox;
    free(morta);
}
```

- Exemplos de uso

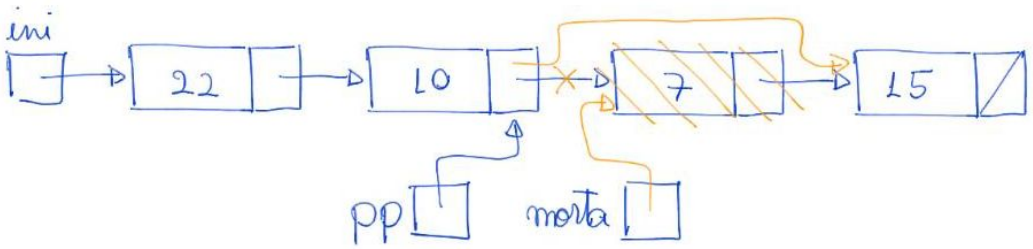
```c
    remove1(ini);
    remove1(ini->prox);
```

Mas, como remover o primeiro elemento da lista?

- Podemos usar apontador e 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

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

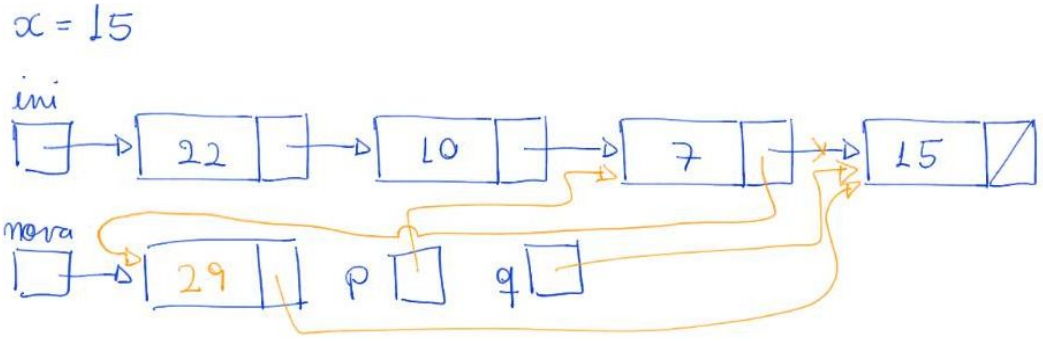
- Outra maneira correta de implementar a remoção do primeiro é
    - devolvendo o novo endereço da lista.

```
// 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).



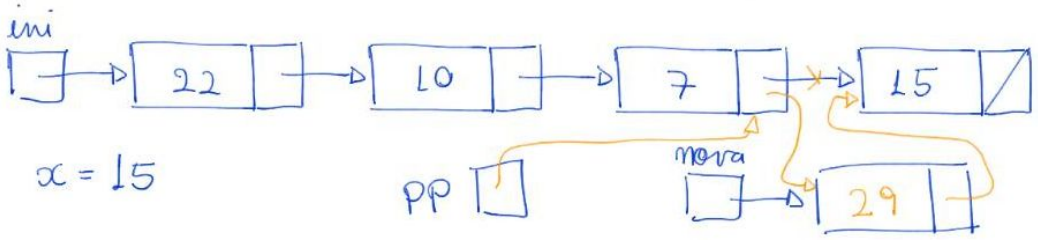

```
// busca x na lista lst e insere y logo antes de x
// se x não está na lista insere y no final
Celula *buscaInsere1(Celula *lst, int x, int y)
{
```

```
    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);
```