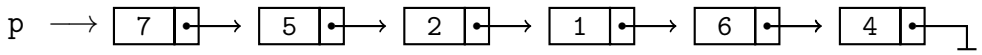
Estruturas de Dados

aula 07: Reorganizando a lista encadeada

1 Introdução

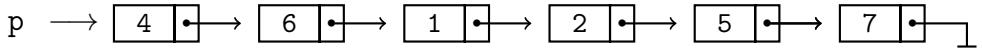
Imagine que o ponteiro `p` aponta para uma lista encadeada



A tarefa é

→ *inverter a ordem dos elementos da lista*

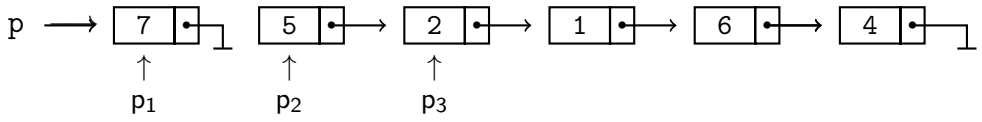
No exemplo acima, isso nos daria



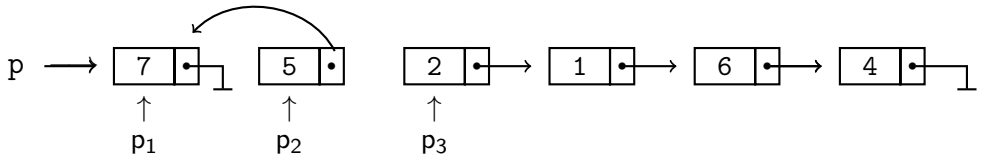
Mas, como é que a gente faz isso?

Bom, a nossa ideia é percorrer a lista com 3 ponteiros.

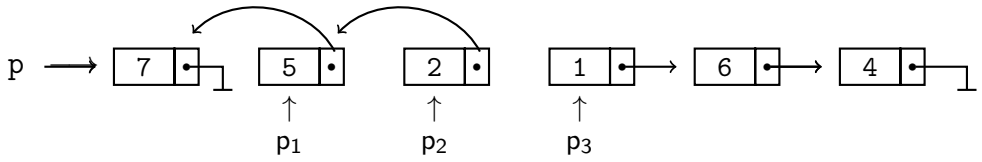
No início, o primeiro elemento aponta para `Nulo` — (*porque agora ele vai ser o último ...*)



E daí, a cada passo, o elemento apontado por p_2 é desconectado do elemento da frente e reconectado ao elemento de trás



Daí os ponteiros andam, e a gente inverte o próximo elemento



No final, basta apontar o ponteiro `p` para aquele que era o último elemento — (*porque agora ele é o primeiro*)

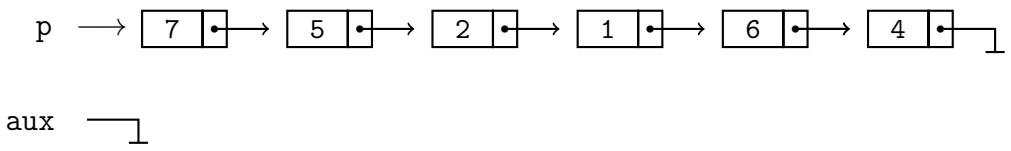
A coisa fica assim

```
func inverteLista (p)

      Se ( p == Nulo  OU  p->prox == Nulo )   Retorna (p)

      p1 = p;    p2 = p1->prox;    p3 = p2->prox        // posiciona os ponteiros

      p1->prox = Nulo                                   // o primeiro vira o último

      Enquanto (p3 != Nulo )
            p2->prox = q->prox
            p1 = p2;    p2 = p3;    p3 = p3->prox

      p2->prox = p1                                     // o último aponta para penúltimo

      p = p2                                            // e vira o primeiro elemento

      Retorna (p)
```

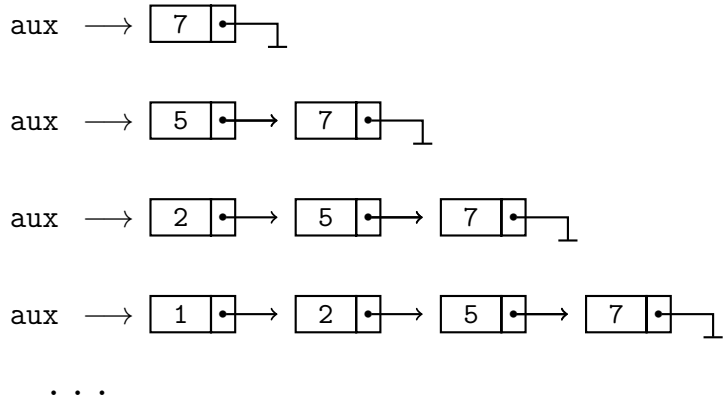
Mas, essa não é a única maneira de fazer as coisas.

Quer dizer, é possível fazer a coisa com apenas um ponteiro auxiliar.

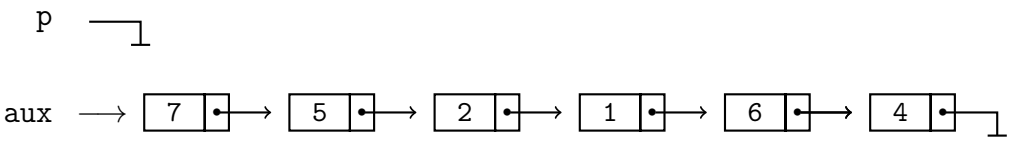
A ideia é que esse ponteiro começa apontando para `Nulo`



Daí, a gente vai sucessivamente removendo os elementos da lista `p`, e inserindo na lista `aux` — (*fazendo a inserção no início*)



Daí no final, a lista está completamente invertida

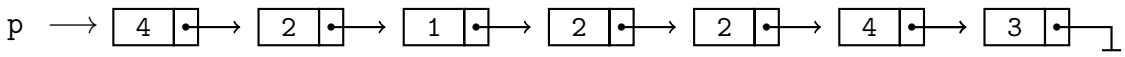


A coisa fica assim

```
func inverteLista2 (p)

     Se ( p == Nulo  OU  p->prox == Nulo )   Retorna (p)

     aux = Nulo                                          // a lista aux começa vazia

     Enquanto (p != Nulo )
          q = p;    p = p->prox                          // tira da lista p
          q->prox = aux;    aux = q                      // bota na lista aux

     Retorna(aux)
```

2 Reorganizando a lista encadeada

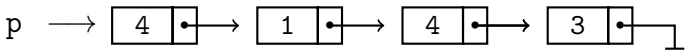
Imagine que o ponteiro p aponta para uma lista encadeada que pode conter elementos repetidos



A tarefa é

→ *remover todas as cópias do número* k

No exemplo acima, para k = 2, isso nos daria



Certo.

Aqui a gente vai utilizar outra vez a técnica de reconstruir a lista com o ponteiro aux.

Quer dizer, nós vamos remover os elmentos de p um a um.

E daí,

- quando o elemento é diferente de k, ele é colocado na lista aux
- quando o elemento é igual a k, o seu registro é liberado

Só que dessa vez, para não inverter a ordem da lista, nós vamos fazer a inserção no final da lista aux.

E para fazer isso de maneira eficiente, nós vamos manter um ponteiro ult apontando para o último elemento de aux.

A coisa fica assim

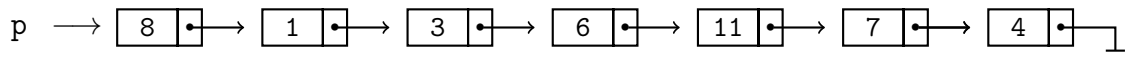
```
func removeCópias ( k, p )

      Se ( p == Nulo )   Retorna (p)                    // já acabei ...

      aux = Nulo;    ult = Nulo

      Enquanto (p != Nulo )
            q = p;    p = p->prox                       // tira da lista p
            q->prox = Nulo
            Se (q->num == k)                            // Você é o kara?
                  free(q)
            Senão                                       // Não?  então vai para a lista aux ...
                  Se (aux == Nulo)
                        aux = q;    ult = q             // o primeiro a entrar em aux
                  Senão
                        ult->prox = q;    ult = q       // e os próximos ...

      Retorna(aux)
```

A seguir, nós vamos ver mais exemplos.

Exemplos

a. Separando pares e ímpares

Imagine que o ponteiro p aponta para uma lista encadeada

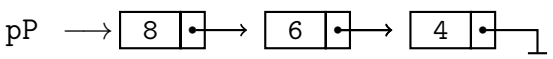


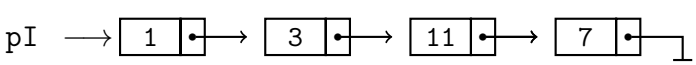
A tarefa é

→ *Colocar os elementos pares no início, e os elementos ímpares no fim da lista*

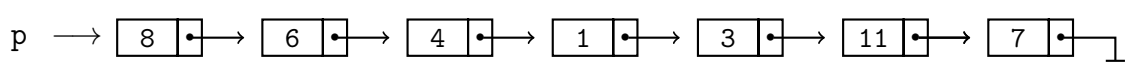
Bom, a ideia aqui é fazer a coisa em etapas.
Quer dizer, primeiro a gente separa os pares e ímpares em duas listas diferentes

Imagine que o ponteiro p aponta para uma lista encadeada





E daí, a gente conecta a segunda lista no fim da primeira



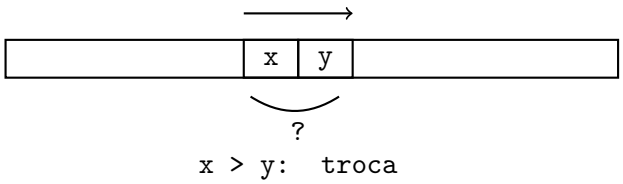
A coisa fica assim

```
func SPI ( k, p )                                   // Separa Pares e Ímpares

      Se ( p == Nulo )   Retorna (p)
                                                    // já acabei ...
      pP = Nulo;    upP = Nulo
      pI = Nulo;    upI = Nulo

      Enquanto (p != Nulo )
            q = p;    p = p->prox                   // tira da lista p
            q->prox = Nulo

            Se (q->num é par)                       // Você é par?
                  Se (pP == Nulo)
                        pP = q;    upP = q
                  Senão
                        upP->prox = q;    upP = q
            Senão                                   // Não?  então você é impar ...
                  Se (pI == Nulo)
                        pI = q;    upI = q
                  Senão
                        upI->prox = q;    upI = q

      Se (pP == Nulo)   Retorna (pI)
      Senão
            upP->prox = pI;    Retorna (pP)
```

◇

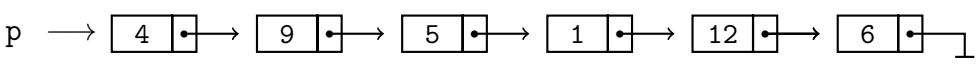
b. A operação de varredura

Lembre que a operação de varredura percorre um vetor comparando elementos consecutivos, e fazendo a troca quando o elemento da esquerda é maior do que o elemento da direita



Agora, nós queremos implementar essa operação na lista encadeada.

Para isso, nós imaginamos que o ponteiro p aponta para uma lista encadeada



E agora?

bom, agora nós aplicamos a nossa estratégia padrão.

Quer dizer, nós vamos remover os elementos da lista p um a um.

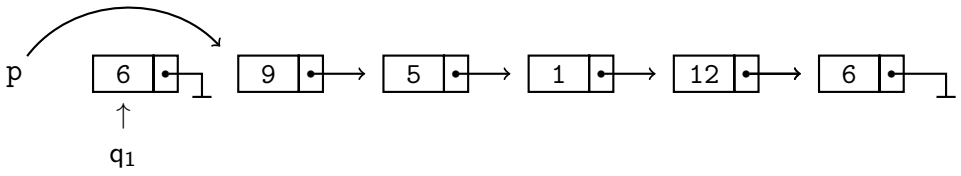
E vamos reconstruir a lista no ponteiro aux.

Só que dessa vez, nós vamos comparar o primeiro e o segundo elementos.

E daí

- se o primeiro for menor, é ele quem vai para a lista aux
- senão, é o segundo que vai para a lista aux

Para fazer isso, nós vamos manter o primeiro elemento de p separado



A coisa fica assim

```
func Varredura ( p )

      Se ( p == Nulo  OU  p->prox == Nulo )   Retorna (p)

      aux = Nulo;   ult = Nulo

      q1 = p;   q1->prox = Nulo                        // separa o primeiro elemento
      p = p->prox

      Enquanto (p != Nulo )
            Se (q1->num < p->num)                      // o 1o elemento é maior
                  Se (aux == Nulo)
                        aux = q1;   ult = q1
                  Senão
                        ult->prox = q1;   ult = q1
                  q1 = p;   q1->prox = Nulo
                  p = p->prox
            Senão                                      // o 2o elemento é maior
                  q2 = p;   q2->prox = Nulo
                  p = p->prox
                  Se (aux == Nulo)
                        aux = q2;   ult = q2
                  Senão
                        ult->prox = q2;   ult = q2

      Retorna (aux)
```

O algoritmo da bolha

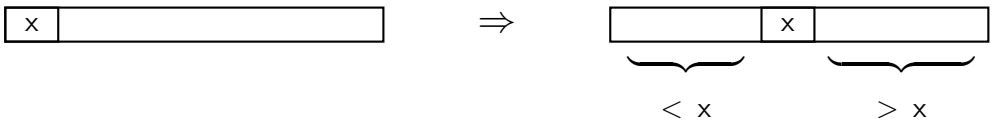
Agora que nós já temos a operação de varredura, é a coisa mais fácil do mundo ordenar a lista encadeada

```
Repita  n  vezes
    p = Varredura(p)
```

◇

c. A operação de partição

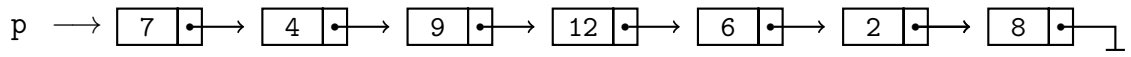
Lembre que a operação de partição reorganiza os elementos de um vetor assim



Quer dizer, os elementos menores que x vão para o início, e os elementos maiores que x vão para o fim do vetor — (*e o elemento* x *fica entre eles*)

Agora, nós queremos implementar essa operação na lista encadeada.

Para isso, nós imaginamos que o ponteiro p aponta para uma lista encadeada



E daí, a gente nota que esse exemplo é quase igual o exemplo (a).

Quer dizer, lá a gente separou pares e ímpares, e agora a gente vai separar menores e maiores que o primeiro elemento.

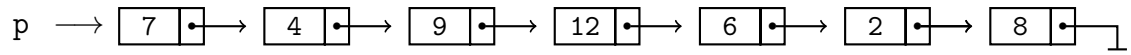
Então, basta fazer a mesma coisa

```
func Particao ( p )

    Se ( p == Nulo  OU  p->prox == Nulo)   Retorna (p)

    pri = p;   p = p->prox                          // separa o primeiro elemento

    aux1 = Nulo;   ult1 = Nulo
    aux2 = Nulo;   ult2 = Nulo

    Enquanto (p != Nulo )
        q = p;   p = p->prox                        // tira o 1o elemento de p
        q->prox = Nulo

        Se (q->num < pri->num)                      // Você é menor que o pivô?
            Se (aux1 == Nulo)
                aux1 = q;   ult1 = q
            Senão
                ult1->prox = q;   ult1 = q
        Senão                                       // Não?  então você é impar ...
            Se (aux2 == Nulo)
                aux2 = q;   ult2 = q
            Senão
                ult2->prox = q;   ult2 = q
```

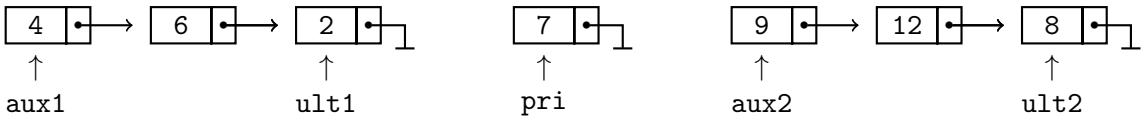
```
pri->prox = aux2                                  // o primeiro entre as duas partes

Se (aux1 == Nulo)   Retorna(pri)
Senão
      ult1->prox = pri;   Retorna(aux1)
```

O algoritmo Quicksort

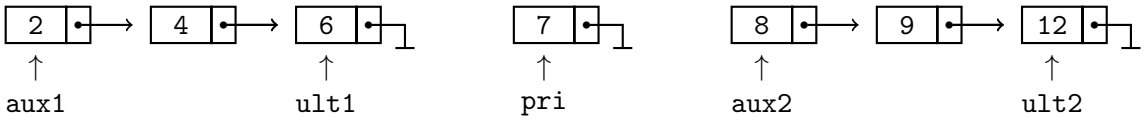
Vamos olhar mais uma vez para a lista `p`



Imagine que a operação de partição deixa as 3 partes desconectadas



E agora, veja o que acontece quando a gente aplica a partição na primeira e na terceira partes



Sim! a lista ficou completamente ordenada — (*basta conectar as 3 partes*)

Bom, aqui a gente deu um pouco de sorte — (*porque bastou uma partição em cada parte*)

Em geral, a gente vai ter que aplicar a partição nas partes das partes — (*e nas partes das partes das partes ...*)

Mas a coisa sempre funciona.

Quer dizer, no final a lista sempre fica ordenada.

Esse é o algoritmo de ordenação *Quicksort*.

E agora, nós já podemos executá-lo sobre uma lista encadeada.

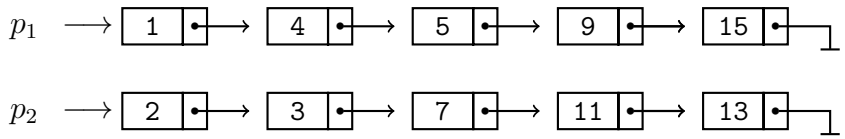
A coisa fica assim

```
func Quicksort ( p, u )                           // ponteiros p/ 1o e ult elem

      Se ( p == Nulo  OU  p->prox == Nulo)
            Retorna (p,u)                         // nada a fazer

      aux1,utl1,pri,aux2,ult2 = Particao (p,u)

      aux1,utl1 = Quicksort (aux1,ult1)

      aux2,utl2 = Quicksort (aux2,ult2)

      p,u   Conecta (aux1,ult1,pri,aux2,ult2)     // junta tudo

      Retorna (p,u)
```

◇

d. A operação de intercalação

Agora imagine que nós temos duas listas encadeadas ordenadas



A tarefa é

→ *produzir uma única lista ordenada com os elementos de* $\mathsf{p_1}$ *e* $\mathsf{p_2}$

Bom, a ideia é comparar os primeiros elementos das duas listas.

E daí, nós movemos o menor deles para a lista aux que está sendo construída — (*fazendo a inserçãa no fim*)

Quando uma das listas chega no fim, basta conectar o restante da outra lista no final da lista aux.

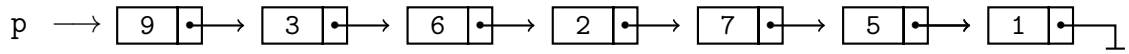
A coisa fica assim

```
func Intercala ( p1, p2 )

     Se ( p1 == Nulo )   Retorna (p2)

     Se ( p2 == Nulo )   Retorna (p1)


     aux = Nulo;   ult = Nulo
     Enquanto ( p1 != Nulo  E  p2 != Nulo )
          Se ( p1->num < p2->num )
               Se ( aux == Nulo )
                    aux = p1;   ult = p1 )
               Senão
                    ult->prox = p1;   ult = p1 )
               p1 = p1->prox
          Senão
               Se ( aux == Nulo )
                    aux = p2;   ult = p2 )
               Senão
                    ult->prox = p2;   ult = p2 )
               p2 = p2->prox

     Se ( p1 == Nulo )   ult->prox = p2
     Senão               ult->prox = p1
     Retorna (aux)
```

◇

e. Quebrando no meio

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

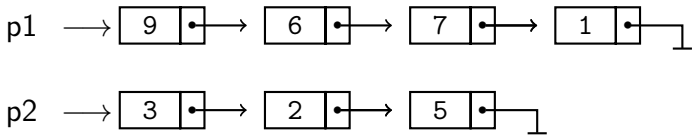
→ *separar os elementos da lista* p *em duas listas* p1 *e* p2
com a mesma quantidade de elementos (aproximadamente)

Bom, a tarefa não diz que nós precisamos quebrar a lista no meio.

Então, nós vamos percorrer a lista p e

- colocar os elementos das posições ímpares em p1
- colocar os elementos das posições pares em p2

No exemplo acima, isso nos daria

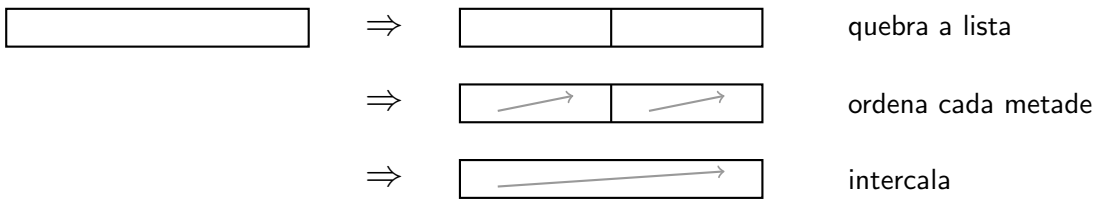


A coisa fica assim

```
func Quebra2 ( p )

     Se ( p == Nulo  OU  p->prox == Nulo )
     Retorna ( p, Nulo )

     pos = 1
     p1 = Nulo;    up1 = Nulo
     p2 = Nulo;    up2 = Nulo
     Enquanto ( p != Nulo )
          Se ( pos é impar )
               Se ( p1 == Nulo )
                    aux = p;    ult = p )
               Senão
                    up1->prox = p;    up1 = p )
               p = p->prox;    up1->prox = Nulo
          Senão
               Se ( p2 == Nulo )
                    p2 = p;    up2 = p2 )
               Senão
                    up2->prox = p;    up2 = p )
               p = p->prox;    up2->prox = Nulo
          pos++

     Retorna (p1,p2)
```

O algoritmo Mergesort

O algoritmo Mergesort ordena um vetor recursivamente, da seguinte maneira



Mas agora, nós já podemos fazer a mesma coisa com a lista encadeada.

A coisa fica assim

```
func Mergesort ( p )

     Se ( p == Nulo )   Retorna (p)

     p1,p2 = Quebra2 (p)

     p1 = Mergesort (p1)

     p2 = Mergesort (p2)

     p = Intercala (p1,p2)

     Retorna (p)
```

◇