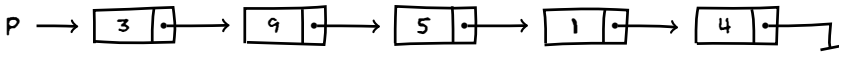
Estruturas de Dados

aula 11: Manipulação recursiva de listas encadeadas

1 Introdução

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

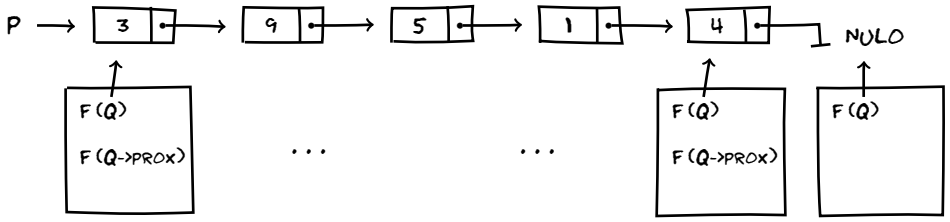


A tarefa é

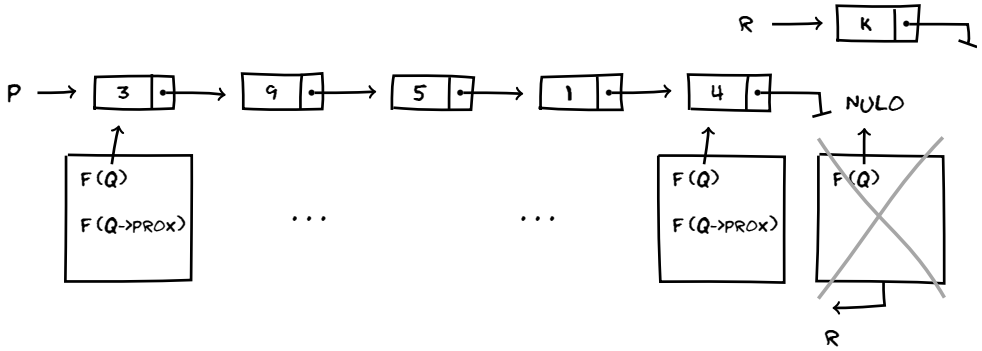
→ *inserir o número* `k` *no fim da fila*

Bom, nós queremos realizar essa tarefa com um algoritmo recursivo.

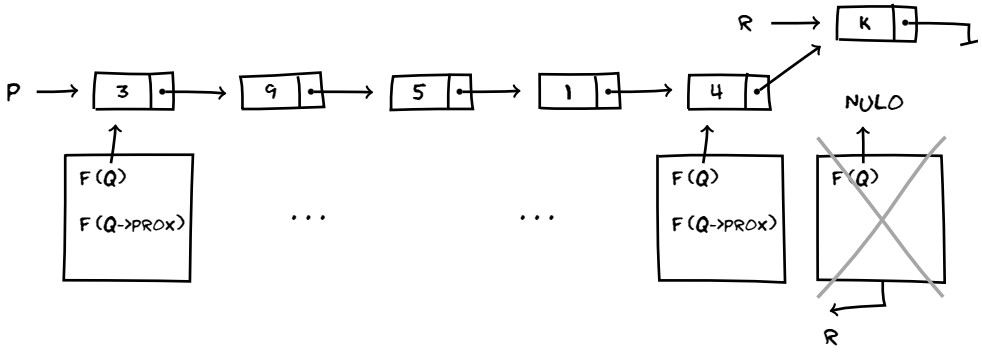
Então, a ideia é que as chamadas recursivas vão alcançar o fim da lista



Daí, a chamada que vê o `Nulo` cria um novo registro, coloca o número `k` dentro dele, e retorna o ponteiro para o registro para chamada anterior

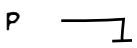


E é a chamada anterior que tem a responsabilidade de conectar o novo registro ao último elemento da lista

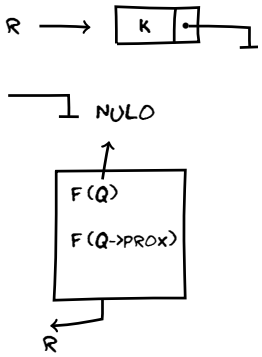


Daí, como a operação de inserção já foi realizada, todas as demais chamadas retornam o ponteiro `Nulo` — (*indicando para a chamada anterior que ela não tem nada a fazer*)

Mas, é preciso ter cuidado com o caso da lista vazia



Nesse caso, a primeira chamada recursiva já cria o novo registro e retorna um ponteiro para ele



Daí, quando a função `main()` recebe um ponteiro diferente de `Nulo`, ela atualiza o ponteiro `p`

A coisa fica assim

```
func inserçãoFim-Rec (q,k)

     Se ( q == Nulo )
          r = Novo (RegInt)                    //
          r->num,prox = k,Nulo                 // criação do registro
          Retorna (r)                          //
     Senão
          resp = inserçãoFim-Rec (q->prox,k)
          Se (resp != Nulo)
               q->prox = resp                  // conexão com o último
          Retorna (Nulo)
```

E a função `main()` fica assim

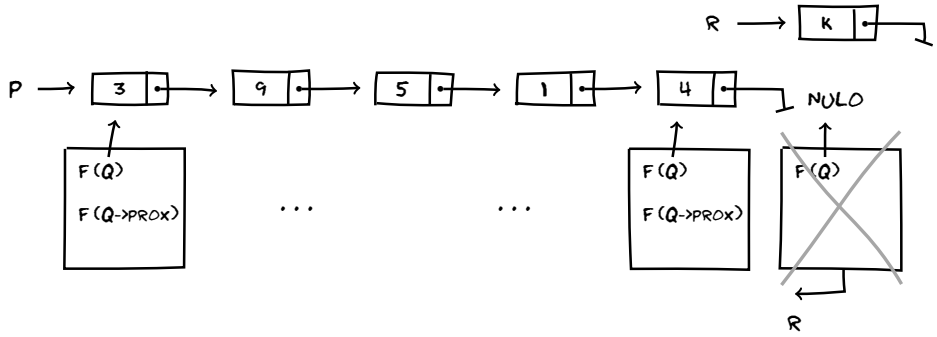
```
func main ()

     resp = inserçãoFim-Rec (p,k)
     Se ( resp != Nulo )
          p = resp
```

Legal

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

Quer dizer, a gente pode simplificar um pouco as coisas utilizando o truque da *reconexão*.

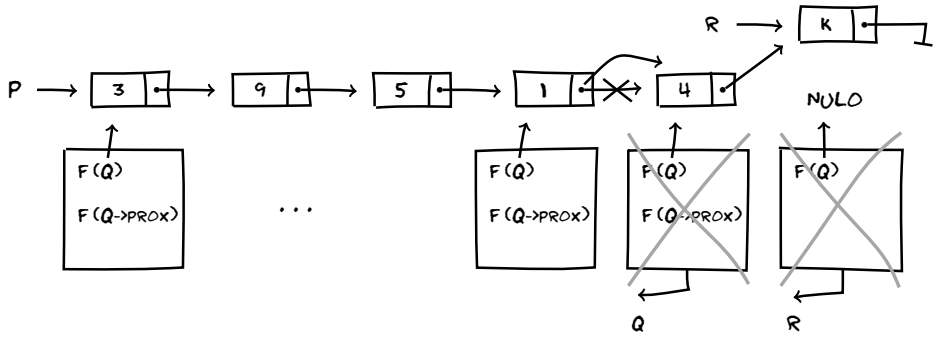
Vamos olhar outra vez para essa situação



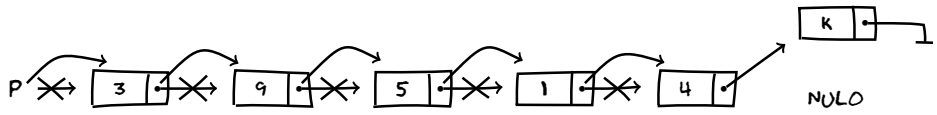
Aqui, a chamada que vê o último elemento está recebendo um ponteiro para o novo registro, para conectá-lo à lista.

Só que dessa vez, ao invés de retornar `Nulo` para a chamada anterior ela vai retornar o seu próprio ponteiro `q` — (*i.e., um ponteiro para o elemento 4*)

Daí, a chamada que vê o penúltimo elemento vai imaginar que o 4 acabou de ser inserido na lista, e vai se (re)conectar a ele



E agora, a coisa continua assim até o fim, com cada elemento sendo reconectado ao próximo



A coisa fica assim

```
func inserçãoFim2-Rec (q,k)

    Se ( q == Nulo )
        r = Novo (RegInt)                    //
        r->num,prox = k,Nulo                 // criação do registro
        Retorna (r)                          //
    Senão
        resp = inserçãoFim2-Rec (q->prox,k)
        q->prox = resp                       // re-conexão
        Retorna (q)
```

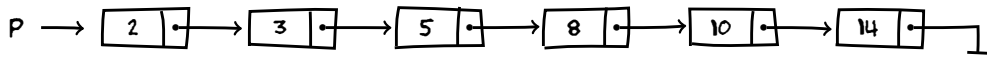
E a função `main()` fica assim

```
func main ()

     p = inserçãoFim2-Rec (p,k)
```

Legal, não é?

2 Manipulação recursiva de listas encadeadas

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

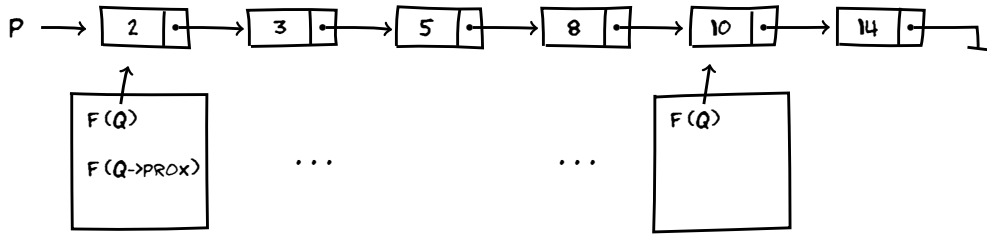


A tarefa é

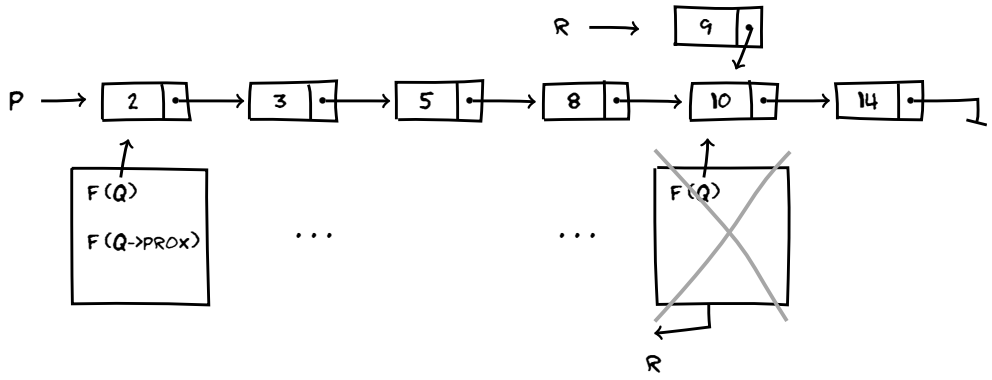
→ *inserir o número* `k` *na lista*

Dessa vez, as chamadas vão até o primeiro elemento maior do que `k` — (*ou até o* `Nulo`*, caso esse elemento não exista*)

Por exemplo, para `k` = 9, nós teríamos o seguinte



Daí, a última chamada cria o novo registro, conecta ele com o que vem depois, e retorna um ponteiro para o registro para a chamada anterior



E a conexão do novo registro com aquilo que vem antes é feita pela chamada anterior — (*pelo truque da reconexão*)

A coisa fica assim

```
func inserçãoOrd-Rec (q,k)

     Se ( q == Nulo  OU  q->num > k )
          r = Novo (RegInt)                          //
          r->num,prox = k,q                          // criação do registro
          Retorna (r)                                //
     Senão
          resp = inserçãoOrd-Rec (q->prox,k)
          q->prox = resp                             // re-conexão
          Retorna (q)
```

E a função `main()` fica assim

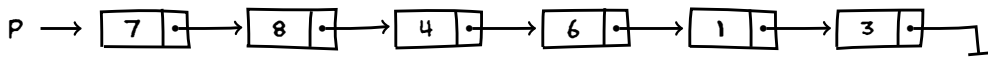
```
func main ()
     p = inserçãoOrd-Rec (p,k)
```

A seguir, nós vamos ver mais exemplos.

Exemplos

a. A operação de remoção

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



A tarefa é

→ *remover o elemento* `k` *da lista*

Dessa vez, as chamadas recursivas vão até o elemento que vai ser removido — (*ou até o* `Nulo`*, caso ele não esteja lá ...*)

Daí, a chamada que vê o `k` libera o seu registro, e retorna um ponteiro para o próximo elemento.

E o resto é só o truque da reconexão.

A coisa fica assim

```
func remoção-Rec (q,k)
     Se ( q == Nulo )   Retorna (Nulo)
     Sse ( q->num == k )
          m = q->prox                                // segura o próximo elemento
          free (q)                                   // e libera o registro
          Retorna (m)
     Senão
          resp = remoção-Rec (q->prox,k)
          q->prox = resp                             // re-conexão
          Retorna (q)
```

E a função `main()` fica assim

```
func main ()
     p = remoção-Rec (p, 9)
```

Legal.

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

Quer dizer, em alguns casos é útil receber um ponteiro para o elemento que foi removido — (*ao invés de liberar o seu registro da memória*)

E para isso, nós vamos fazer a nossa função retornar dois valores.

A coisa fica assim

```
func remoção2-Rec (q, k)
     Se ( q == Nulo )   Retorna (Nulo, Nulo)
     Sse ( q->num == k )
          m = q->prox                          // segura o próximo elemento
          q->prox = Nulo                       // e desconecta o registro
          Retorna (m, q)
     Senão
          px,r = remocao2-Rec (q->prox, k)
          q->prox = px                         // re-conexão
          Retorna (q, r)
```

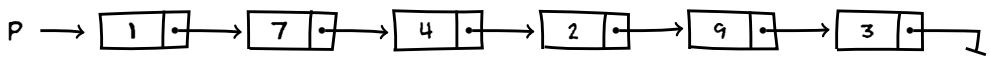
E a função `main()` fica assim

```
func main ()
     p,r = remoção2-Rec (p, 9)
     Se ( r == Nulo)
          Imprime ('o elemento não estava lá ...')
     Senão
          Imprime ('o elemento foi removido!')
```

◇

b. O último no início

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *mover o último elemento para a primeira posição da lista*

Bom, nós queremos fazer isso de maneira recursiva.

Então, a ideia é fazer as chamadas recursivas chegarem até o último elemento — (*e parar ali*)

Daí, a chamada que vê o último elemento retorna `Nulo` — (*para a chamada anterior conectar o penúltimo elemento ao* `Nulo`)

E também retornar um ponteiro para o elemento que era o último.

As outras chamadas apenas aplicam o truque da reconexão, e repassam o último elemento para trás.

Dessa maneira, o último elemento chega ao início da lista.

Mas, como é que uma chamada sabe que ela vê o primeiro elemento?

Bom, a função `main()` pode dizer isso para ela — (*por meio de um parâmetro* `pri`)

E daí, elas diz para as chamadas seguintes que elas não vêem o primeiro elemento.

A coisa fica assim

```
func UnI-Rec (q,pri)                              // Último no Início

     Se ( pri == 1 )                              // a chamada que vê o 1o
          Se ( q == Nulo  OU  q->prox == Nulo)
               Retorna (q,Nulo)                   // já acabei ...
          Senão
               px,u = UnI-Rec (q->prox,0)
               q->prox = px                       // re-conexão
               u->prox = q
               Retorna (u,Nulo)

     Senão                                        // demais chamadas
          Se (q->prox == Nulo)                    // último elemento
               Retorna (Nulo,q)
          Senão
               px,u = UnI-Rec (q->prox,0)
               q->prox = px                       // re-conexão
               Retorna (q,u)
```

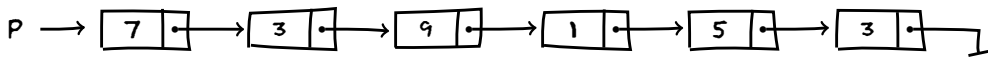
E a função `main()` fica assim

```
func main ()

     p,u = UnI-Rec (p,1)
```

◇

c. O menor no início

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *mover o menor elemento para a primeira posição da lista*

Bom, aqui a gente precisa fazer a coisa em etapas

1. na ida a gente descobre quem é o menor elemento
2. e na volta a gente carrega ele para a primeira posição

Para descobrir que é o menor elemento, a gente utiliza um parâmetro `maa` que lembra quem é o menor elemento até aqui.

Daí, na chamada que vê o `Nulo`, esse parâmetro tem o valor do menor elemento.

Esse número é mandado de volta como valor de retorno.

Daí, a chamada que vê o menor elemento consegue saber disso — (*comparando o seu elemento com o valor de retorno*)

E então, ela remove o seu elemento da lista, e manda ele para o início da lista.

Finalmente, a chamada que vê o primeiro elemento faz a inserção do menor elemento no início da lista.

A coisa fica assim

```
func MnI-Rec (q,pri,maa)                              // Menor no Início
    Se ( pri == 1 )                                   // a chamada que vê o 1o
        Se ( q == Nulo  OU  q->prox == Nulo)
            Retorna (q,Nulo)                          // já acabei ...
        Senão
            px,m,pm = MnI-Rec (q->prox,0,q->num)
            q->prox = px                              // re-conexão
            pm->prox = q
            Retorna (pm,0,Nulo)
    Senão                                             // demais chamadas
        Se (q->prox == Nulo)                          // último elemento
            Retorna (Nulo,maa,Nulo)
        Senão
            Se (q->num < maa)   maa = q->num          // atualiza maa
            px,m,pm = UnI-Rec (q->prox,0)
            Se (q->num == maa)
                Retorna (px,-1,q)                     // retorna o menor elemento
            Senão
                q->prox = px                          // re-conexão
                Retorna (q,m,pm)
```

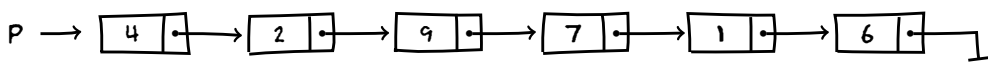
E a função `main()` fica assim

```
func main ()

     p,m,pm = MnI-Rec (p,1,-1)
```

◇

d. O maior no fim

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



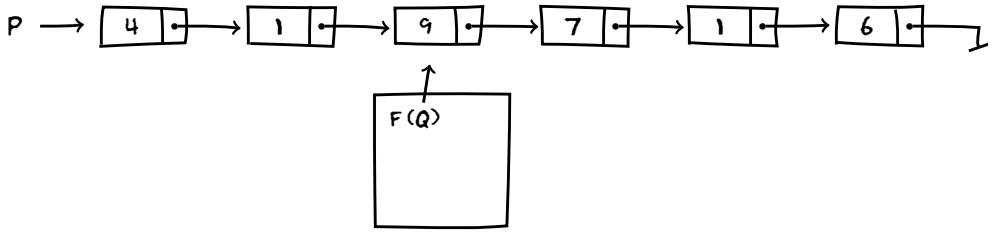
A tarefa é

→ *mover o maior elemento para a última posição da lista*

Bom, a ideia aqui é fazer a coisa em etapa outra vez

1. na ida a gente descobre que é o maior elemento
2. e na volta a gente encontra e manda ele para o fim

A primeira etapa vai ser feita da mesma maneira que no exemplo anterior.
Mas na segunda etapa a gente pode fazer uma esperteza.
Quer dizer, vamos ver as coisas na perspectiva da chamada que vê o maior elemento



O ponto é que essa chamada vê uma lista encadeada à sua direita — (*apontada por* `q->prox`)
E o que ela quer fazer é inserir um elemento no fim dessa lista — (*o seu próprio elemento*)
Ora mas se é assim, então tudo o que ela tem a fazer é chamar a função `inserçãoFim-Rec()`.
O único detalhe é que a função `inserçãoFim-Rec()` recebe um número como parâmetro.
Mas nós já temos um ponteiro para o registro que contém o número.
Então, nós vamos escrever uma outra função `inserçãoFimP-Rec()` para utilizar nesse exemplo.
A coisa fica assim

```
func MnF-Rec (q,Maa)                              // Maior no Fim

     Se ( q == Nulo )
          Retorna (Nulo,Maa)
```

```
    Senão
        Se ( q->num > Maa )   Maa = q->num
        px,M = MnF-Rec ( q->prox, Maa )
        Se ( q->num == M )                          // eu sou o Maior
            q->prox = Nulo
            px = inserçãoFimP-Rec ( px, q )
            Retorna (px, -1)
        Senão
            q->prox = px
            Retorna (q, M)
```

A função `inserçãoFimP-Rec()` fica assim

```
func inserçãoFimP-Rec (q, r)
    Se ( q == Nulo )   Retorna (r)
    Senão
        px = inserçãoFim-Rec (q->prox, k)
        q->prox = px                                // re-conexão
        Retorna (q)
```

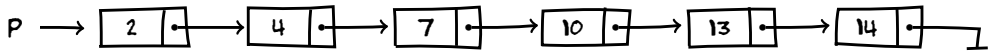
E a função `main()` fica assim

```
func main ()
    p,M = MnF-Rec (p, -1)
```

◇

e. A operação de atualização

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



A tarefa é

→ *encontrar o elemento* `x`

modificar o seu valor para `y` *e recoloca-lo na posição correta*

Bom, aqui a própria tarefa já foi especificada em etapas.

A primeira e a segunda etapas são fáceis.

E a terceira etapa pode ter dois casos

- se $y > x$, o elemento pode ir para frente
- se $y < x$, o elemento pode ir para trás

O primeiro caso é fácil, porque ele consiste em fazer uma inserção na lista ordenada — (*nós podemos chamar uma função para fazer isso*)

E no segundo caso, a gente carrega o elemento para trás — (*utilizando o valor de retorno da função*)

O lugar em que a inserção vai ser feita é o primeiro elemento menor que `y` — (*se houver algum*)

Ou então a primeira posição da lista — (*se não houver nenhum*)

A coisa fica assim

```
func atualização-Rec (q,x,y,pri)

     Se ( q == Nulo  OU  q->num > x )              // o x não está lá ...
          Retorna (Nulo,Nulo)

     Sse (q->num == x )
          q->num = y;   q->prox = Nulo
          Se (y > x)
               px = inserçãoOrdP-Rec (q->prox,q)
               Retorna (px,Nulo)
          Senão
               Retorna (q->prox,q)

     Senão
          px,el = atualização-Rec (q->prox,x,y,0)
          Se (el != Nulo  E  q->num < el->num)
               el->prox = q->prox
               q->prox = el
               Retorna (q,Nulo)
          Senão
               el->prox = q
               Retorna (el,Nulo)
```

A função `main()` fica assim

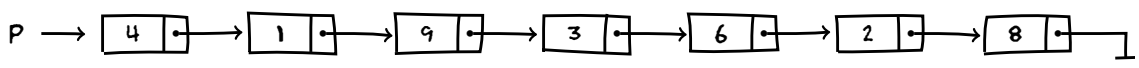
```
func main ()

     p,el = atualização-Rec (p,10,17 1)
```

— (*e a função* `inserçãoOrdP-Rec()` *fica como exercício*)

◇

f. Removendo o elemento central

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



A tarefa é

→ *remover o elemento central da lista*

— *(se houver algum elemento central ...)*

Quer dizer, se a lista tem uma quantidade ímpar de elementos, então existe um elemento central e ele deve ser removido.

Caso contrário, não existe um elemento central e não é preciso fazer nada.

Certo.

A única dificuldade é localizar o elemento central — (*caso haja um*)

Nós vamos resolver essa dificuldade com a seguinte esperteza

- nós vamos contar os elementos na ida e na volta
- e aquele elemento que tiver o mesmo número na ida e na volta é o elemento central

Para fazer a contagem na ida nós vamos utilizar um parâmetro.

E para fazer a contagem na volta nós vamos utilizar o valor de retorno da função.

A coisa fica assim

```
func REC-Rec (q,cont1)                              // Remove o Elemento Central

     Se ( q == Nulo )   Retorna (Nulo,0)            // inicia a contagem da volta

     Senão
          px,cont2 = REC-Rec (q->prox,cont1+1)
          Se (cont1 == cont2)                       // eu sou o centro!
               free (q)
               Retorna (px,cont2+1,)
          Senão
               q->prox = px                         // re-conexão
               Retorna (q,cont2+1,)
```

◇