Estruturas de Dados

aula 13: Raciocínio recursivo

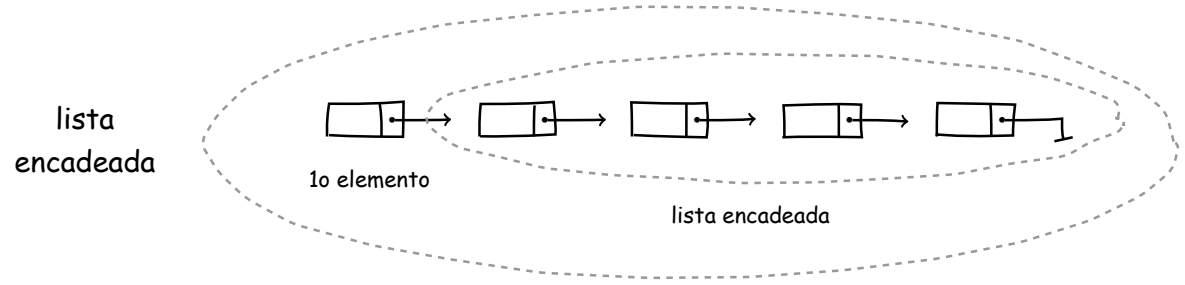
1 Introdução

Uma função recursiva é uma função que faz uma chamada a ela mesma.

Por isso, as funções recursivas são ótimas para trabalhar com objetos onde a gente consegue ver a coisa dentro da coisa — (*i.e., objetos recursivos*)

Ora, as listas encadeadas são um objeto recursivo.

Porque a gente pode ver a lista encadeada dentro da lista encadeada



E porque as coisas são assim, a gente pode construir algoritmos para listas encadeadas raciocinando de maneira recursiva.

Por exemplo, imagine que a tarefa é

→ *imprimir todos os elementos da lista*

Então, a gente pode raciocinar assim

```
1. basta imprimir o 1o elemento
2. e depois chamar a função de impressão para o restante da lista
```

E a coisa fica assim

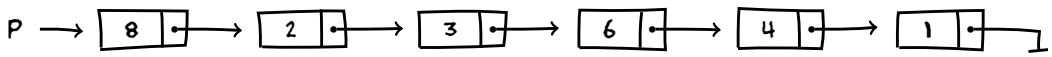
```
func imprime-Rec (q)
     Se ( q == Nulo )   Retorna                 // já acabei
     Imprime (q->num)                           // o primeiro elemento
     imprime-Rec (q->prox)                      // o restante da lista
```

2 Raciocínio recursivo

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

A tarefa é

→ *contar o número de elementos na lista*



Raciocinando de maneira recursiva, a gente resolve a tarefa assim

1. primeiro a gente conta o número de elementos no restante da lista
   — (*fazendo uma chamada recursiva*)
2. e depois a gente soma 1 — (*para contar o 1o elemento*)

A coisa fica assim

```
func contaEl-Rec (q)

     Se ( q == Nulo )   Retorna (0)            // aqui não tem ninguém ...

     cont = contaEl-Rec (q->prox)              // conta o restante da lista

     Retorna (cont + 1)                        // conta o primeiro elemento
```

Agora imagine que a tarefa é

→ *contar o número de elementos ímpares na lista*

Então, o raciocínio é quase a mesma coisa

1. primeiro a gente conta o número de elementos ímpares no restante da lista
   — (*fazendo uma chamada recursiva*)
2. e depois a gente verifica se o primeiro elemento é ímpar
   se ele for, a gente soma 1 no resultado

A coisa fica assim

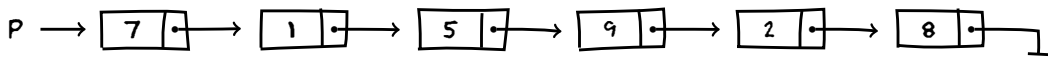
```
func contaImpar-Rec (q)

     Se ( q == Nulo )   Retorna (0)            // aqui não tem ninguém ...

     cont = contaImpar-Rec (q->prox)           // conta o restante da lista

     Se (q->num é ímpar)                       // verifica o primeiro elemento
          Retorna (cont + 1)
     Senão
          Retorna (cont)
```

A seguir nós vamos ver mais exemplos.

Exemplos

a. O maior elemento

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



A tarefa é

→ *descobrir qual é o maior elemento da lista*

Dessa vez, o raciocínio recursivo fica assim

1. primeiro a gente descobre o maior elemento do restante da lista
   — (*fazendo uma chamada recursiva*)
2. e depois a gente compara o primeiro elemento com ele
   e retorna o maior dos dois

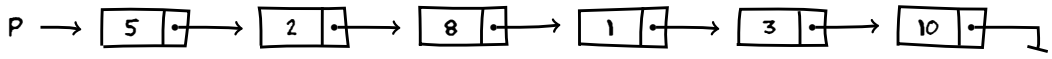
E a coisa fica assim

```
func maior-Rec (q)

    Se ( q == Nulo )   Retorna (-1)          // não tem ninguém ...

    M = maior-Rec (q->prox)                  // o maior do restante da lista

    Se (q->num > M)                          // o primeiro elemento
        Retorna (q->num)
    Senão
        Retorna (M)
```

◇

b. O primeiro no fim

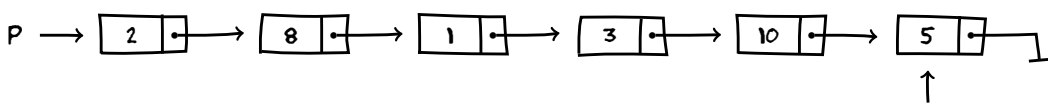
Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *colocar o 1o elemento na última posição*
   — *(sem modificar a ordem dos demais)*

No exemplo acima, isso nos daria



Daí, o raciocínio recursivo fica assim

1. primeiro a gente troca o 1o elemento de lugar com o próximo
2. e depois, a gente coloca o 1o elemento do restante da lista no fim
   — (*fazendo uma chamada recursiva*)

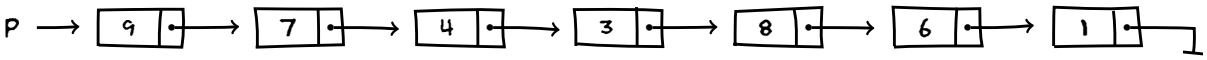
E a coisa fica assim

```
func PNF-Rec (q)                                  // Primeiro No Fim

     Se ( q == Nulo  OU  q->prox == Nulo )        // já acabei
          Retorna (q)

     m1 = q->prox;   m2 = m1->prox
     q->prox = m2                                 //
     m1->prox = q                                 // troca o 1o e o 2o
     q = m1                                       //

     px  = PNF-Rec (q->prox)
     q->prox = px                                 // o truque da re-conexão
     Retorna (q)
```

◇

c. A operação de remoção

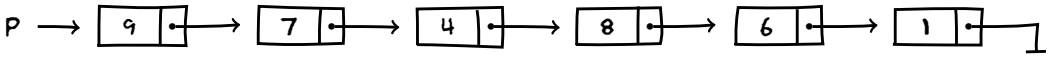
Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *remover o elemento* k *da lista*

No exemplo acima, para k = 3 isso nos daria



O raciocínio fica assim

1. Se o primeiro elemento é o k
   então a gente remove ele da lista — (*e acabou-se*)
2. senão, a gente remove o k do restante da lista
   — (*fazendo uma chamada recursiva*)

A coisa fica assim

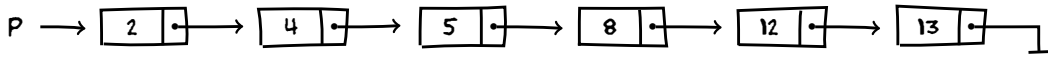
```
func remoção-Rec (q,k)

     Se (q == Nulo)   Retorna (Nulo)          // já removi

     Se (q->num == k)
          m  = q->prox                        //
          free (q)                            // remove o 1o elemento
          Retorna (m)                         //
```

```
                                                    //
        px  = remoção-Rec ( q->prox, k )            // o restante da lista
        q->prox = px                                //
        Retorna ( q )
```

◇

d. A operação de inserção

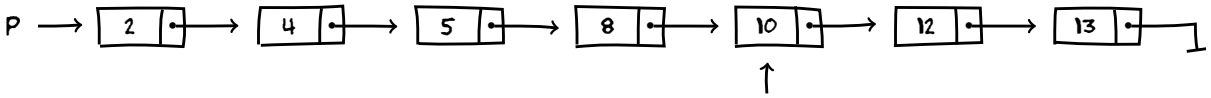
Imagine que o ponteiro `p` aponta para uma lista encadeada ordenada



A tarefa é

→ *inserir o número* `k` *no lugar correto da lista*

No exemplo acima, isso nos daria



Bom, dessa vez o raciocínio recursivo fica assim

1. `Se o k é meior do que o 1o elemento (ou a lista está vazia)`
   `então a gente insere o k na primeira posição`
2. `senão, a gente insere o k no restante da lista`
   — (*fazendo uma chamada recursiva*)

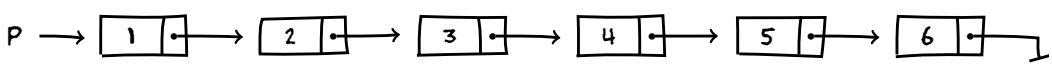
E a coisa fica assim

```
func inserção-Rec ( q, k )

     Se ( q == Nulo  OU  q->num  >  k )
          r = Novo ( RegInt )                   //
          r->num,prox = k,q                     // inserção na 1a posição
          Retorna ( r )                         //

     px  = inserção-Rec ( q->prox, k )          //
     q->prox = px                               // inserção no restante da lista
     Retorna ( q )                              //
```

◇

e. Invertendo a ordem da lista

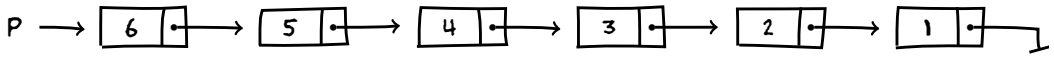
Imagine que o ponteiro `p` aponta para uma lista encadeada



A tarefa é

→ *inverter a ordem dos elementos da lista*

No exemplo acima, isso nos daria



Bom, aqui o raciocínio recursivo natural é o seguinte

1. `primeiro a gente inverte a ordem dos elementos no restante da lista`
2. `e depois, a gente coloca o 1o elemento na última posição`
   — (*`fazendo uma chamada recursiva`*)

E a coisa fica assim

```
func inverte-Rec (q)

     Se ( q == Nulo  OU  q->prox == Nulo )      // já inverti ...
          Retorna (q)

     px  = inverte-Rec (q->prox)
     q->prox = px
     q  = PNF-Rec (q)                           // Primeiro No Fim
     Retorna (q)
```

O problema é que essa função executa em tempo $O(n^2)$.

Porque cada elemento da lista faz uma chamada à função `PNF-Rec()`.

E a função `PNF-Rec()` executa em tempo $O(n)$ — (*porque?*)

E agora?

Bom, agora é a hora para uma pequena esperteza.

Quer dizer, a ideia é fazer a nossa função retornar um ponteiro `u` para o último elemento da lista.

Porque daí, cada chamada pode colocar o seu elemento no fim, sem chamar a função `PNF-Rec()`.

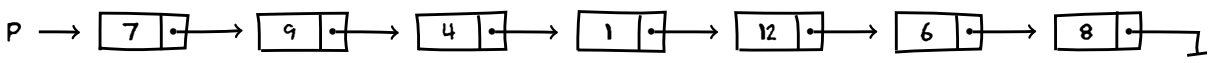
A coisa fica assim

```
func inverte2-Rec (q)

     Se (q == Nulo)   Retorna (Nulo)          // caso da lista vazia

     Se (q-prox == Nulo)   Retorna (q,q)      // o último elemento

     px  = inverte2-Rec (q->prox)
     q->prox = Nulo                           // coloca o 1o no fim
     u->prox = q                              //
     Retorna (px)
```

◇

f. A operação de partição

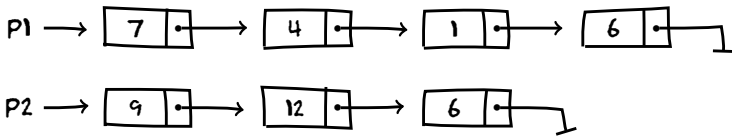
Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *particionar a lista* p *em duas listas* p1 *e* p2
onde os elementos da lista p1 *são todos* ≥ k
e os elementos da lista p2 *são todos* < k

No exemplo acima, isso nos daria



O raciocínio recursivo para essa tarefa fica assim

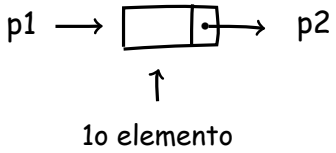
1. primeiro a gente particiona o restante da lista
   e obtém os ponteiros p1 e p2 — (*fazendo uma chamada recursiva*)
2. depois, a gente compara o primeiro elemento com o k
   e coloca ele em p1 ou p2

E a coisa fica assim

```
func partição-Rec (q,k)

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

     p1,p2 = partição-Rec (q->prox)            // particiona o restante da lista

     Se ( q->num ≥ k)                          // o primeiro elemento
          q->prox = p1;   Retorna (q,p2)       //
     Senão                                     //
          q->prox = p2;   Retorna (p1,q)       //
```

O algoritmo Quicksort

Agora que nós temos a função de partição, é bem fácil escrever o algoritmo quicksort.
Quer dizer, a ideia é utilizar o primeiro elemento como pivô da partição — (*i.e., o valor do* k)
Daí, a gente particiona o restante da lista.
E ordena recursivamente as listas p1 e p2 — (*chamando a função* quicksort-Rec())
Finalmente, basta fazer a concatenação



A coisa fica assim

```
func quicksort-Rec (q)

      Se (q == Nulo)   Retorna (q)                  // já ordenei ...

      p1,p2 = partição-Rec (q->prox,q->num)         // particiona o restante da lista

      p1 = quicksort-Rec (p1)                       // ordena a 1a parte

      p2 = quicksort-Rec (p2)                       // ordena a 2a parte

      q = concatena (p1,q,p2)                       // e concatena

      Retorna (q)
```

◇