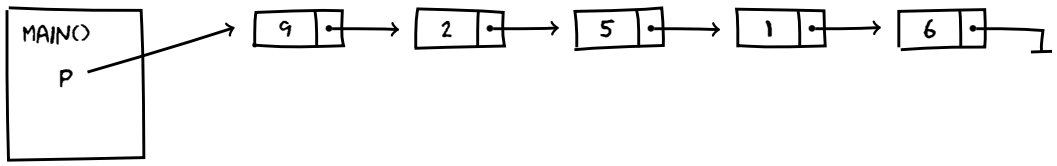
Estruturas de Dados

aula 10: Algoritmos recursivos para listas encadeadas

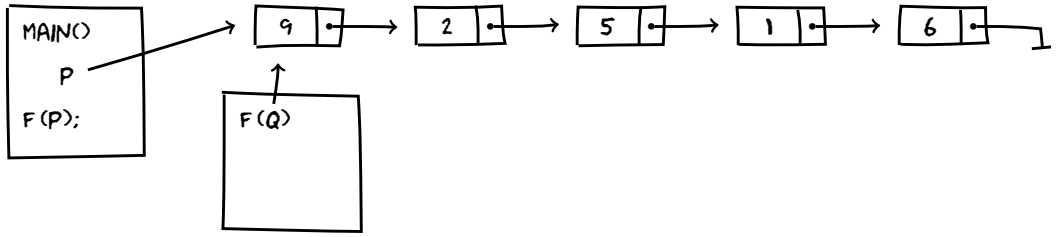
1 Introdução

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

E imagine também a função principal do seu programa

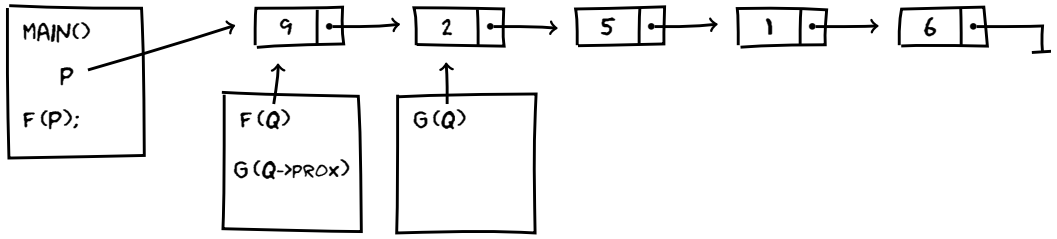


Agora, imagine que o seu programa faz a chamada a uma outra função, passando o ponteiro `p` como parâmetro

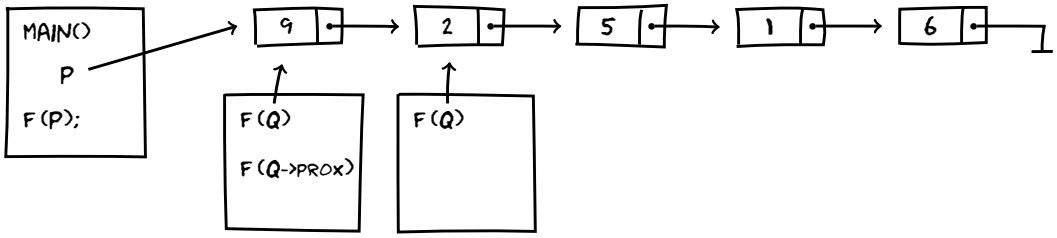


Sim! isso é como um dedo que foi colocado sobre o primeiro elemento.

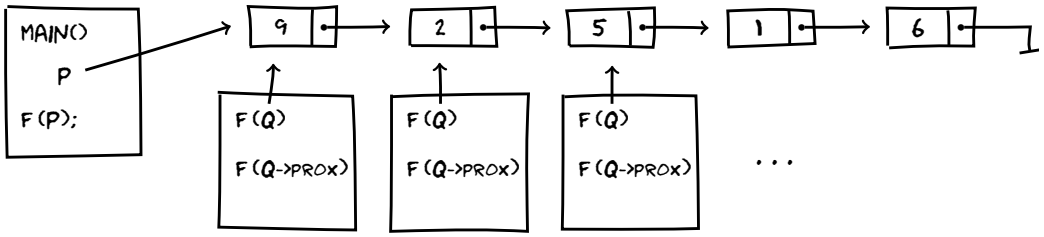
Agora, é fácil imaginar a função `F()` fazendo a chamada a uma outra função, passando `q->prox` como parâmetro



Só que, a coisa fica mais legal se a função `F()` faz uma chamada a ela mesma — (*i.e., uma chamada recursiva*)



Porque agora as chamadas vão continuar acontecendo, uma depois da outra

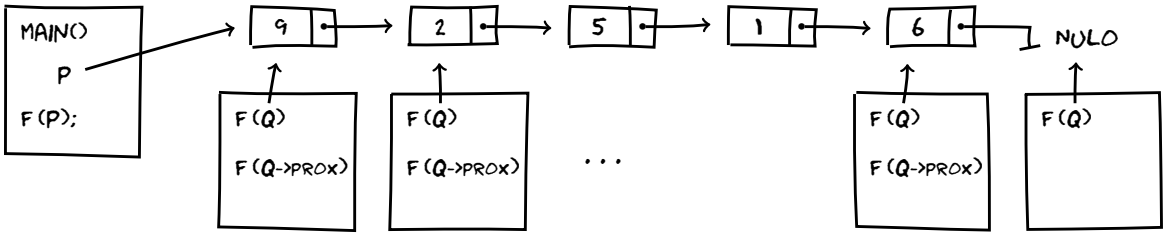


Sim! isso é como um dedo que percorre a lista.

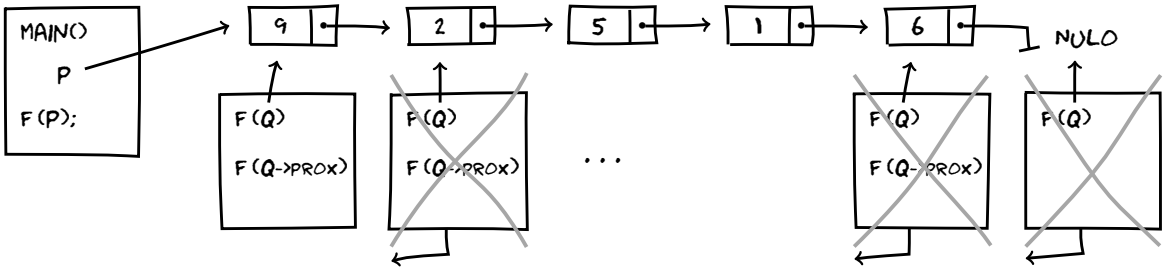
Mas, quando é que a coisa pára?

Bom, as chamadas vão continuar acontecendo até que a gente chegue no `Nulo`.

Então, é ali que a coisa pára



Daí, as chamadas vão terminando uma por uma — (*na ordem contrária em que elas aconteceram*)



Não é legal?

Agora nós temos uma outra maneira de percorrer a lista, por meio de uma função recursiva

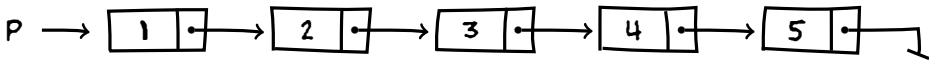
```
func F-Rec (q)

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

     F( q->prox )                            // passa para o próximo

     ( . . . )

     Retorna                                 // e volta
```

E para fazer alguma coisa de útil com isso, basta preencher os 3 pontinhos.

2 Algoritmos recursivos para listas encadeadas

Vejamos um exemplo concreto.

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



E imagine que a tarefa é

→ *Imprimir todos os elementos da lista*

Então, a coisa fica assim

```
func Imprime-Rec (q)

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

     F( q->prox )                            // passa para o próximo

     Imprime ( q->num )

     Retorna                                 // e volta
```

Mas, você viu o que aconteceu?

Quer dizer, essa função imprime os elementos na ordem contrária

```
=>  5,   4,   3,   2,   1
```

Porque?

Ora, porque a impressão acontece na hora em que as caixinhas estão fechando — (*i.e., na hora em que a gente está voltando*)

Então, existem dois lugares para colocar código na função recursiva

```
func F-Rec (q)

     Se ( q == Nulo )   Retorna

     ( . . . )                          // aquilo que acontece na ida

     F( q->prox )

     ( . . . )                          // aquilo que acontece na volta

     Retorna
```

Então, a gente pode modificar a função de impressão assim

```
func Imprime-Rec (q)

      Se ( q == Nulo )   Retorna

      Imprime ( q->num )
                                                  // Imprime o elemento na ida
      F( q->prox )

      Retorna
```

para ver os números números na ordem certa

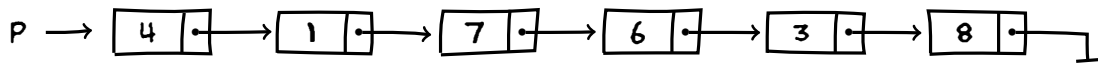
```
=>  1,   2,   3,   4,   5
```

A seguir, nós vamos ver mais exemplos.

Exemplos

a. Encontrando o número `k`

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



A tarefa é

→ *Verificar se o número* `k` *está na lista ou não*

Bom, a ideia é realizar a tarefa com uma função recursiva.

Quer dizer, a medida que a gente vai percorrendo a lista, nós verificamos se o elemento atual é o número `k`.

Se isso é o caso, a gente imprime uma mensagem na tela — (*e já começa a voltar*)

Mas se a gente chega no `Nulo`, isso significa que o número `k` não está na lista.

E se isso é o caso, a gente imprime outra mensagem na tela — (*e depois começa a voltar*)

A coisa fica assim

```
func busca-Rec (q,k)
      Se ( q == Nulo )
            Imprime ('Não está lá ...')
            Retorna
      Sse ( q->num == k )
            Imprime ('Achei!')
            Retorna
      Senão
            busca-Rec ( q->prox, k )
      Retorna
```

Legal.

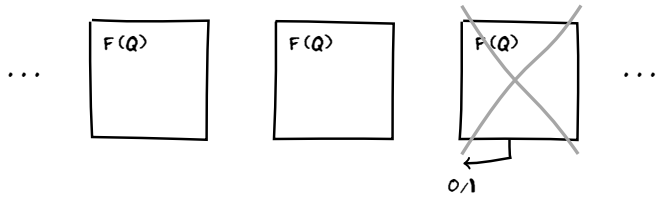
Mas, essa não é a única maneira de fazer as coisas.

Quer dizer, ao invés de imprimir uma mensagem na tela, a gente pode retornar um sinal para a função `main()` saber se o número `k` está lá ou não.

A ideia é a seguinte

```
Se ( q == Nulo )
      Retorna (0)
Sse ( q->num == k )
      Retorna (1)
```

Mas para chegar na função `main()`, esse sinal precisa passar por todas as caixinhas que vão fechando



Para fazer isso, cada caixinha recebe a resposta enviada pela próxima, e retorna essa resposta para a anterior.

A coisa fica assim

```
func busca2-Rec (q, k)
      Se ( q == Nulo )
            Retorna (0)
      Sse ( q->num == k )
            Retorna (1)
      Senão
            resp = busca2-Rec (q->prox, k)
            Retorna (resp)
```

Dessa maneira, a função `main()` recebe essa informação como o valor de retorno da função `busca2-Rec()`.

E daí, ela pode fazer o que quiser com isso

```
func main ()
      resp = busca2-Rec (p, k)
      Se ( resp == 0 )
            Imprime ('Não está lá...!')
      Senão
            Imprime ('Achei!')
```

◇

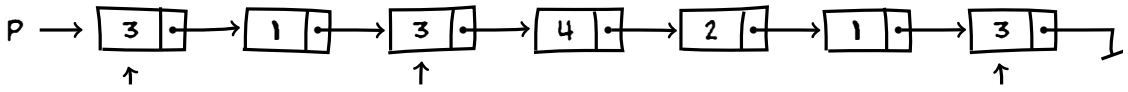
b. Contando o número de ocorrências

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

A tarefa é

→ *Contar a quantidade de ocorrências do número* `k`

No exemplo acima, para `k` = 3, isso nos daria




```
=>  3 ocorrências
```

Bom, dessa vez a gente não pode começar a voltar quando encontra a primeira ocorrência do `k`.

Quer dizer, a ideia é ir até o final da lista.

E daí, a gente pode fazer a contagem na volta.

Quer dizer, a chamada que vê o `Nulo` retorna 0.

E toda chamada que vê o número `k` adiciona 1 ao valor que está sendo retornado.

A coisa fica assim

```
func contaOcK-Rec (q,k)
     Se ( q == Nulo )   Retorna (0)          // começando a contar do 0
     resp = contaOcK-Rec (q->prox, k)
     Sse ( q->num == k )                     // Você é o kara?
          resp++
     Retorna (resp)
```

E a função `main()` fica assim

```
func main ()
     resp = contaOcK-Rec (p, 3)
     Imprime ('o número de ocorrências do 3 é:', resp)
```

Legal.

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

Quer dizer, a gente também pode fazer a contagem na ida.

Para fazer isso, a gente adiciona um parâmetro `cont` à nossa função.

Daí, a função `main()` inicia a contagem passando o valor 0 ao parâmetro `cont`.

E cada chamada que vê uma ocorrência do `k` adiciona 1 ao cont.

No final, a chamada que vê o `Nulo` retorna o valor de `cont`.

E todas as demais repassam o valor retornado para a chamada anterior — (*para que ele até a função* `main()`)

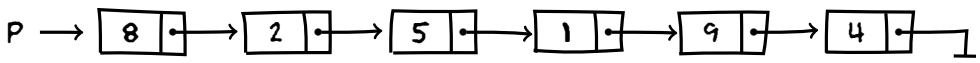
A coisa fica assim

```
func contaOcK2-Rec (q, k, cont)

     Se ( q == Nulo )   Retorna (cont)              // retornando o resultado

     Sse ( q->num == k )                            // Você é o kara?
          resp = contaOcK-Rec (q->prox, k, cont+1)
     Senão
          resp = contaOcK-Rec (q->prox, k, cont)
```

◇

c. Imprimindo os elementos das posições ímpares

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *Imprimir os elementos das posições ímpares da lista*

Bom, a ideia aqui é utlizar a passagem de parâmetro outra vez.

Quer dizer, as chamadas recebem um parâmetro indicando se elas estão em uma posição par ou ímpar.

E quando elas fazem a sua chamada recursiva, elas invertem o valor do parâmetro.

Daí, as chamadas que estiverem em uma posição ímpar imprimem o valor do seu elemento.

A coisa fica assim

```
func imprimePI-Rec (q, pd)
     Se ( q == Nulo )   Retorna              // já acabei ...
     Sse ( q->num == k )
          Imprime (q->num)
     imprimePI-Rec (q->prox, 1 - pd)         // inverte o valor de pd
     Retorna
```

E a função `main()` fica assim

```
func main ()
     imprimePI-Rec (p, 1)
```

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

Quer dizer, a gente também pode percorrer a lista pulando de 2 em 2 — (*i.e., passando apenas pelas posições ímpares*)

```
imprimePI2-Rec ( (q->prox)->prox )
```

Aqui, é preciso ter cuidado para não tentar dar o salto de tamanho 2 na última posição.

Porque na última posição

```
(q->prox)->prox   ≡   (Nulo)->prox
```

Só que o `Nulo` não tem um campo `prox` — (*o que dá um erro de execução ...*)

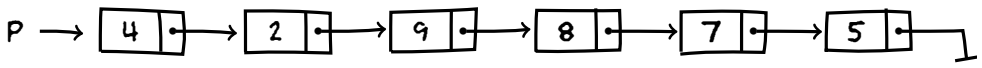
A coisa fica assim

```
func imprimePI2-Rec (q)
     Se ( q == Nulo )   Retorna                  // já acabei ...
     Imprime (q->num)
     Se ( q->prox != Nulo )                      // eu não sou o último?
          imprimePI-Rec ( (q->prox)->prox )
     Retorna
```

◇

d. O maior de todos

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



A tarefa é

→ *encontrar o maior elemento da lista*

Bom, aqui a gente pode fazer a coisa na ida ou na volta.

Fazendo as coisas na ida, a gente utiliza um parâmetro `maa` que lembra qual foi o maior valor que foi visto até aqui.

No início, a função `main()` atribui o valor -1 a esse parâmetro, para indicar que ainda não foi visto nenhum elemento.

A coisa fica assim

```
func maior-Rec (q,maa)
     Se ( q == Nulo )   Retorna                  // já acabei ...
     Se ( q->num > maa )                         // eu sou o maior até aqui?
          resp = maior-Rec (q->prox,q->num)
     Senão
          resp = maior-Rec (q->prox,maa)
     Retorna (resp)
```

E a função `main()` fica assim

```
func main ()
     M = maior-Rec (p,-1)
     Imprime ('o maior elemento é', M)
```

Legal.

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

Quer dizer, a gente também pode fazer as coisas na volta.

E a gente pode retornar um ponteiro para o maior elemento.

Fazendo as coisas assim, a gente primeiro vai até o fim da lista.

Daí, a chamada que vê o `Nulo` retorna o ponteiro `Nulo`.

Todas as outras chamadas comparam o seu elemento com aquele que está sendo retornado, e retornam o maior dos dois.

A coisa fica assim

```
func maior2-Rec (q,maa)
     Se ( q == Nulo )   Retorna                    // já acabei ...
     resp = maior2-Rec (q->prox)
     Se (resp == Nulo  OU  q->num > resp->num)     // o último ou o maior até aqui
          Retorna (q)
     Senão
          Retorna (resp)
```

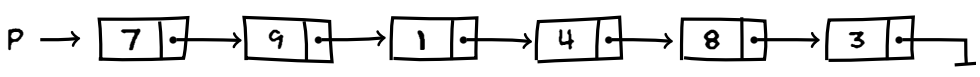
E a função `main()` fica assim

```
func main ()
     pM = maior2-Rec (p)
     Se (pM != Nulo)
          Imprime ('o maior elemento é', pM->num)
```

◇

e. Calculando a média

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *Calcular a média dos elementos da lista*

Bom, para calcular a média é preciso somar todos os elementos.

E depois é preciso dividir o resultado pelo número de elementos da lista.

Nós vamos calcular esses dois valores na ida, utilizando os parâmetros `soma` e cont.

Daí, a chamada que vê o `Nulo` faz o cálculo da média e retora esse valor.

E todas as demais chamadas apenas repassam esse valor até a função `main()`.

A coisa fica assim

```
func media-Rec (q, soma, cont)
     Se ( q == Nulo )   Retorna (soma / cont)        // cálculo da média
     Senão
          soma = soma + q->num                       // atualiza a soma
          cont = cont + 1                            // atualiza o cont
          resp = media-Rec (q->prox, soma, cont)
          Retorna (resp)
```

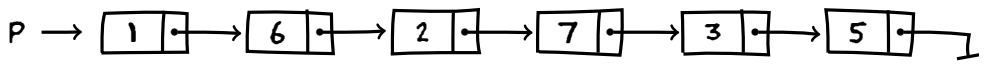
E a função `main()` fica assim

```
func main ()
     md = media-Rec (p)
     Imprime ('a media é', md)
```

◇

f. Contando os maiores que a média

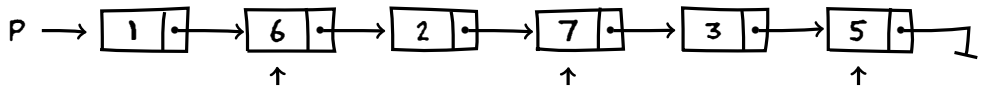
Imagine que o ponteiro `p` aponta para uma lista encadeada



A tarefa é

→ *contar a quantidade de elementos maiores do que a média*

No exemplo acima, isso nos daria



`3 acima da média` — (*a média é 4.0*)

Bom, a solução mais fácil consiste em utilizar a função que calcula a média.

E escrever uma função que conta os elementos maiores do que `k`.

A coisa fica assim

```
func CMqK-Rec (q,k,)                          // Conta Maiores que K
     Se ( q == Nulo )   Retorna (0)           // começando a contar do 0
     resp = CMqK-Rec (q->prox,k)
     Se (q->num > k)                          // eu sou maior que o kara?
          resp = resp + 1
     Retorna (resp)
```

E a função `main()` fica assim

```
func main ()
     md = media-Rec (p,0,0)
     cont = CMqK (p,m)
     Imprime ('o número de elementos maiores do que a media é', cont)
```

Legal.

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

Quer dizer, a gente também pode modificar a função `media-Rec()` para que ela conte os elementos maiores do que a média na volta.

Para fazer isso, nós vamos utilizar dois valores de retorno.

A coisa fica assim

```
func CCMqM-Rec (q,soma,cont)                  // Calcula e Conta Maiores que a Média
     Se ( q == Nulo )   Retorna (soma/cont,0)
     Senão
          soma = soma + q->num                // atualiza a soma
          cont = cont + 1                     // atualiza o cont
          m,c = CCMqM-Rec (q->prox,soma,cont)
          Se (q->num > m)
               c++
          Retorna (m,c)
```

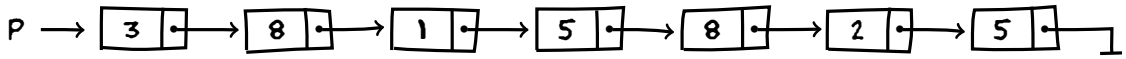
E a função `main()` fica assim

```
func main ()
     n,mm = CCMqM-Rec (p,0,0)
     Imprime ('o número de elementos maiores do que a media é', mm)
```

◇

g. Elemento repetido

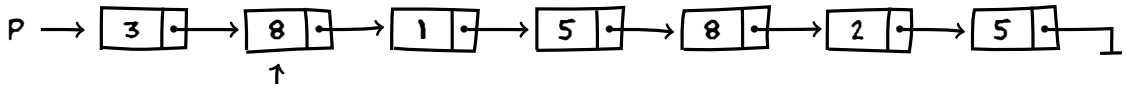
Imagine que o ponteiro `p` aponta para uma lista encadeada que pode conter elementos repetidos



A tarefa é

→ *Verificar se a lista contém algum elemento repetido*
e retornar o primeiro deles

No exemplo acima, isso nos daria



`sim, o 8`

Bom, a ideia aqui é percorrer a lista por meio de chamadas recursivas.
E para cada elemento, nós verificamos se ele aparece mais adiante.
Mas, nós já temos uma função que procura um número na lista — (*a função* `busca2-Rec()`)
Então, nós podemos utilizar essa função para fazer essa verificação.
A coisa fica assim

```
func elRep-Rec (q)
     Se ( q == Nulo )   Retorna (0)              // não encontrei repetido
     Senão
          resp = busca2-Rec (q->prox,q->num)     // atualiza a soma
          Se (resp == 1)                         // atualiza o cont
               Retorna (q->num)
          Senão
               resp = elRep-Rec (q->prox)
               Retorna (resp)
```

E a função `main()` fica assim

```
func main ()
     er = elRep-Rec (p)
     Se ( er == 0)
          Imprime ('não tem repetido')
     Senão
          Imprime ('o primeiro repetido é', er)
```

◇