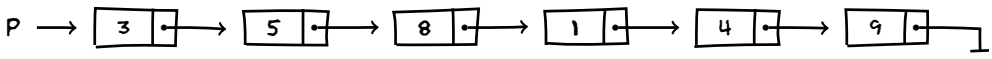
Estruturas de Dados

aula 12: Reorganização recursiva de listas encadeadas

1 Introdução

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *Inverter a ordem dos elementos da lista*

Bom, nós queremos fazer isso de maneira recursiva.

E a maneira mais fácil é fazer isso na ida.

Quer dizer, a lista invertida vai ser construída no parâmetro pi que é passado para a função.

A função `main()` inicializa o parâmetro pi como `Nulo`.

E cada chamada recursiva coloca o seu elemento no início da lista apontada por pi.

Daí, como a lista p é percorrida da esquerda para a direita, os elementos acabam ficando na ondem contrária.

Finalmente, a chamada que vê o `Nulo` retorna o ponteiro pi.

E todas as demais repassam esse ponteiro para a função `main()`.

A coisa fica assim

```
func inverte-Rec (q,pi)

     Se ( q == Nulo )   Retorna (q)
     Senão
          m = q->prox                          // segura o próximo elemento
          q->prox = pi                         // insere q em pi
          pi = q                               //
          resp = inverte-Rec (m,pi)
          Retorna (resp)
```

Legal.

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

Quer dizer, a gente também pode inverter a lista na volta.

Só que dessa vez, a gente não pode colocar os elementos no início da lista que está sendo construída.

Porque a lista p vai ser percorrida da direita para a esquerda.

E os elementos iam acabar ficando na mesma ordem que na lista p.

Então, os elementos vão ser inseridos no fim.

E para facilitar essa inserção, nós vamos ter um ponteiro pi para o primeiro elemento, e um ponteiro ui para o último elemento.

Esses dois ponteiros são passados como valor de retorno da função.

No início, a chamada que vê o Nulo retorna Nulo,Nulo.

E cada chamada recursiva adiciona o seu elemento no fim da lista apontada por pi, ui.

A coisa fica assim

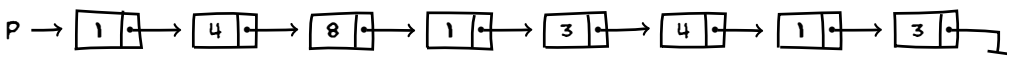
```
func inverte2-Rec (q,pi)

    Se ( q == Nulo )   Retorna (Nulo,Nulo)       // começando a voltar
    Senão
        pi,ui = inverte2-Rec (q->prox)
        q->prox = Nulo
        Se (ui == Nulo)                          //
            ui = q;   pi = q                     // insere no fim
        Senão                                    //
            ui->prox = q;   ui = q               //

        Retorna (pi,ui)
```

E a função main() fica assim

```
func main ()

    pi,ui = inverte2-Rec (p)
    p = pi
```

2 Reorganização recursiva de listas encadeadas

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

Bom, dessa vez nós vamos fazer as coisas na volta.

Quer dizer, as chamadas recursivas vão até o Nulo.

E na volta, nós removemos o elemento sempre que ele é igual a k.

A coisa fica assim

```
func RTCK-Rec (q,k)                              // Remove Todas as Cópias do k

     Se ( q == Nulo )   Retorna (q)
     Senão
          resp = RTCK-Rec (q->prox,k)            // vai até o fim

          Se (q->num != k)
               q->prox = resp
               Retorna (q)
          Senão
               free (q)                          // removendo o k na volta
               Retorna (resp)                    //
```

Legal.

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

Quer dizer, a gente também pode retornar uma lista com todos os elementos que foram removidos para a função `main()`.

A coisa fica assim

```
func RTCK2-Rec (q,k)                             // Remove Todas as Cópias do k

     Se ( q == Nulo )   Retorna (Nulo,Nulo)
     Senão
          px,r = RTCK2-Rec (q->prox,k)           // vai até o fim

          Se (q->num != k)
               q->prox = px                      // truque da re-conexão
               Retorna (q,r)
          Senão
               q->prox = r                       // insere o k na lista de removidos
               Retorna (px,q)                    //
```

E a função `main()` fica assim

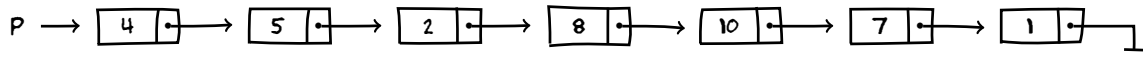
```
     func main ()

          p,r = RTCK2-Rec (p,1)
          aux = r;   cont = 0
          Enquanto (aux != Nulo)                 // conta elementos removidos
               cont++;   aux = aux->prox

          Imprime (cont, 'elementos removidos')
```

A seguir nós vamos ver mais exemplos.

Exemplos

a. Separando pares e ímpares

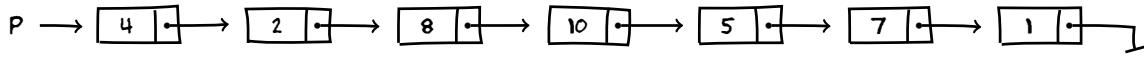
Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

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

No exemplo acima, isso nos daria



Bom, a ideia é fazer as coisas na ida dessa vez.

Quer dizer, a função recursiva vai ter dois parâmetros pp e pi, onde nós vamos colocar os elementos pares e ímpares.

Daí, quando a chamada que vê os `Nulo` recebe essas duas listas, ela faz a sua concatenação e retorna o resultado.

Para facilitar a concatenação, nós vamos utilizar um terceiro ponteiro up que aponta para o último elemento da listas dos pares.

A coisa fica assim

```
func SPI-Rec (q,pp,up,pi)                          // Separa Pares e Ímpares

     Se ( q == Nulo )                              // lá no fim
          Se (pp == Nulo)   Retorna (pi)
          Senão
               up->prox = pi;   Retorna (pp)       // concatena pp e pi

     Sse (q->num é par)
          m = q->prox                              // segura o próximo elemento
          q->prox = pp;   pp = q                   // e adiciona elemento à lista pp
          Se (up == Nulo)   up = q

     Senão
          m = q->prox                              // segura o próximo elemento
          q->prox = pi;   pi = q                   // e adiciona elemento à lista pi

     resp = SPI-Rec (m,pp,up,pi)
     Retorna (resp)
```

Legal.

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

Quer dizer, a gente pode ser um pouquinho mais esperto.

A ideia é já ir construindo a lista resultado na ida

- inserindo os elementos pares no início
- inserindo os elementos ímpares no fim

Para isso, a gente utiliza dois parâmetros ppi e upi.

A coisa fica assim

```
func SPI2-Rec (q, ppi, upi)                    // Separa Pares e Ímpares

     Se (q == Nulo)   Retorna (ppi)            // lá no fim

     Sse (q->num é par)
          m = q->prox                          // segura o próximo elemento
          q->prox = ppi;   ppi = q             // e adiciona elemento no inicio
          Se (upi == Nulo)   upi = q

     Senão
          m = q->prox                          // segura o próximo elemento
          q->prox = Nulo
          Se (upi == Nulo)
               ppi = q;   upi = q              // primeiro elemento (no fim)
          Senão
               upi->prox = q;   upi = q        // próximos elementos (no fim)

     resp = SPI2-Rec (m, ppi, upi)
     Retorna (resp)
```

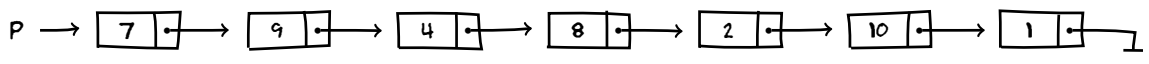
E a função `main()` fica assim

```
func main ()

     p = SPI2-Rec (p, Nulo, Nulo)
```

◇

b. A operação de partição

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *Colocar todos os elementos* $\leq$ `k` *no início da lista*
e todos os elementos $>$ `k` *no fim da lista*

No exemplo acima, para `k` = 7, isso poderia nos daria

Bom, isso é a mesma coisa que o exemplo anterior.

Quer dizer, basta considerar

- os elementos $\leq$ `k` como "elementos pares"
- os elementos $>$ `k` como "elementos ímpares"

Dessa vez, a gente vai fazer a coisa na volta — (*para preservar a ordem dos elementos*)

Então primeiro a gente vai até o fim da lista.

Daí, a chamada que vê o `Nulo` inicia o processo fazendo `p1` = `Nulo` e `p2` = `Nulo`.

E daí, cada chamada insere o seu elemento em `p1` ou `p2`, dependendo do seu valor.

No final, basta concatenar as listas p_1 e `p2`.

A coisa fica assim,

```
func partição-Rec (q,k)

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

     p1,p2 = partição-Rec (q->prox,k)          // vai até o fim

     Se (q->num ≤ k)
          q->prox = p1;   p1 = q               // insere na lista p1
     Senão
          q->prox = p2;   p2 = q               // insere na lista p2

     Retorna (p1,p2)
```

Legal.

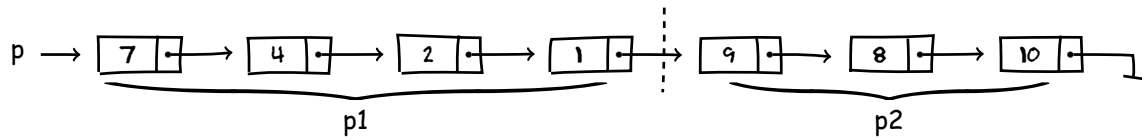
Mas, você viu que essa função não concatena `p1` e `p2`?

Porque?

Bom, porque agora a gente vai fazer outra coisa.

O algoritmo Quicksort

A operação de partição (seguida da concatenação) já começa a colocar a lista em ordem



E para terminar a ordenação, basta continuar particionando `p1` e `p2` — (*é por isso que a gente não fez a concatenação*)

Mas aqui é preciso ter um pequeno cuidado.

Quer dizer, a partição foi feita com o número 7 — (*o primeiro elemento da lista*)

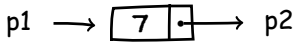
Isso significa que todos os outros elementos em p1 são menores que o 7.

E todos os elementos em p2 são maiores que o 7.

Logo, o lugar correto do 7 é entre as listas p1 e p2 — (*i.e.,* p1 *sem o primeiro elemento*)

Então, é conveniente remover o primeiro elemento da lista antes de fazer a partição — (*e utilizar o seu valor como pivô da partição*)

Daí, no momento da concatenação, a gente faz a coisa assim



— (*para obter a lista ordenada*)

Como todo mundo sabe, esse é o algoritmo Quicksort.

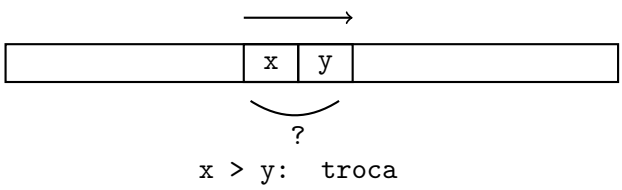
E a coisa fica assim

```
func quicksort-Rec (q)

     Se (q == Nulo)   Retorna (Nulo)
     Senão
          pri = q                                  // separa o 1o elemento
          p1,p2 = partição-Rec (q->prox, pri->num) // particiona
          p1 = quicksort-Rec (p1)                  // ordena os 2 lados
          p2 = quicksort-Rec (p2)                  //
          q = concatena (p1, pri, p2)              // e concatena
          Retorna (q)
```

◇

c. A operação de varredura

A operação de varredura consiste em percorrer uma lista comparando elementos consecutivos, e fazendo a troca sempre que o elemento da esquerda é maior do que o elemento da direita

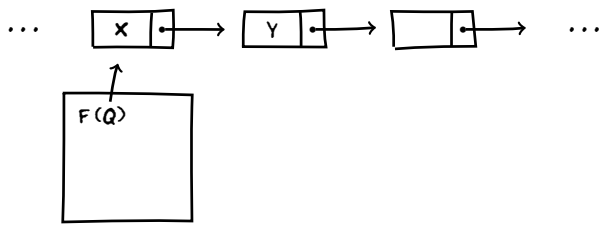


— (*como todo mundo sabe*)

Agora, nós queremos implementar esse procedimento de maneira recursiva.

Bom, a ideia é fazer a coisa na ida.

Quer dizer, cada chamada recursiva vai examinar o seu elemento e o próximo



Daí, se o seu elemento for maior que o próximo, ela faz a troca de posição entre os dois.

E depois, ela faz a chamada recursiva sobre o próximo elemento.

Finalmente, a chamada que vê o último elemento encerra o processo — (*porque ela só tem um elemento para inspecionar*)

A coisa fica assim

```
func varredura-Rec (q)

     Se (q == Nulo  OU  q->prox == Nulo)
          Retorna (q)
     Senão
          Se (q->num > (q-prox)->num)
               m1 = q->prox;    m2 = m1->prox          // segura
               q->prox = m2                            //
               m1->prox = q                            // e troca
               q = m1                                  //
          px = varredura-Rec (q->prox)
          q->prox = px                                 // o truque da re-conexão
          Retorna (q)
```

O algoritmo da bolha

Legal.

Mas agora que nós temos a função de varredura, nós queremos implementar a ordenação da bolha.

Bom, para fazer a ordenação da bolha basta fazer várias chamadas à função varredura, dentro de um laço.

Mas, nós não queremos utilizar um laço.

Nós queremos fazer a coisa de maneira recursiva.

E agora?

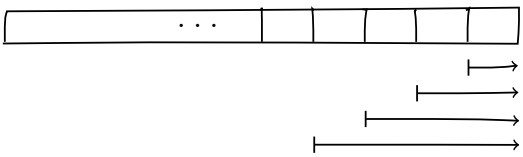
Bom, agora é a hora para um esperteza.

Quer dizer, a ideia é escrever uma função recursiva que vai até o fim da lista.

E daí, ela volta fazendo chamadas à função `varredura-Rec()`.

Mas porque isso funciona?

Bom, vamos ver o que acontece em sucessivas varreduras realizadas dessa maneira



- ia primeira varredura não faz nada — (*porque?*)
- daí, a segunda varredura ordena as duas últimas posições — (*porque?*)
- daí, a terceira varredura ordena as três últimas posições — (*porque?*)
- e por aí vai ...

No final, a lista inteira vai estar ordenada.

Legal, não é?

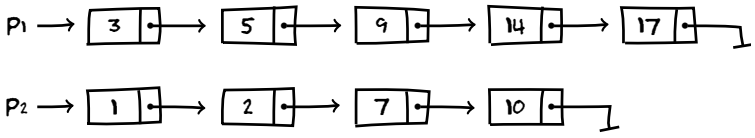
A coisa fica assim

```
func ordBolha-Rec (q)

     Se (q == Nulo)   Retorna
     Senão
          px = ordBolha-Rec (q->prox)          // vai até o fim
          q->prox = px
          q = varredura-Rec (q->prox)          // e volta fazendo varredura
          Retorna (q)
```

◇

d. A operação de intercalação

Imagine que os ponteiros p1 e p2 apontam para duas listas encadeadas ordenadas



A tarefa é

→ *produzir uma única lista encadeada ordenada com os elementos de* p1 *e* p2

Bom, a ideia é fazer as coisas na ida.

Quer dizer, a lista vai ser construída em um parâmetro p que é passado para a função.

No início, a função `main()` atribui o valor `Nulo` a esse parâmetro.

E daí, cada chamada adiciona um elemento à lista que está sendo construída: o menor entre o 1o elemento de p1 e o 1o elemento de p2.

No momento em que uma das listas fica vazia, a outra lista é colocada no fim da lista que está sendo construída — (*e a função retorna*)

As inserções são sempre feitas no fim.

E para fazer isso de maneira eficiente, nós também utilizamos o parâmetro `u` que aponta para o último elemento da lista `p`.

A coisa fica assim

```
func intercalação-Rec (q1,q2,p,u)

     Se (q1 == Nulo)                              // a 1a lista já acabou
          Se (p == Nulo)   Retorna (q2)
          Senão   u->prox = q2;   Retorna (p)

     Sse (q2 == Nulo)                             // a 2a lista já acabou
          Se (p == Nulo)   Retorna (q1)
          Senão   u->prox = q1;   Retorna (p)

     Senão
          Se (q1->num < q2->num)                  //
               m = q1;   q1 = q1->prox            // segura o menor elemento
          Senão                                   //
               m = q2;   q2 = q2->prox            //
          m->prox = Nulo

          Se (p == Nulo)                          //
               p = m;   u = m                     // e insere no fim
          Senão                                   //
               u->prox = m;   u = m               //

          resp = intercalação-Rec (q1,q2,p,u)
          Retorna (resp)
```

E a função `main()` fica assim

```
func main ()

     p = intercalação-Rec (p1,p2,Nulo,Nulo)
```

O algoritmo Mergesort

Agora que nós temos a função de intercalação (recursiva) é fácil escrever o algoritmo Mergesort (recursivo).

Mas antes disso, é preciso ter uma função que quebra a lista em duas — (*de maneira recursiva*)

Só que isso é bem fácil.

Quer dizer, nós vamos fazer a coisa na volta.

A cada passo, a chamada recursiva coloca o seu elemento em `p1` ou `p2` — (*de maneira alternada*)

E no final, metade dos elementos vai estar em p1, e a outra metade em p2.

A coisa fica assim

```
func divide-Rec (q)

     Se (q == Nulo)   Retorna (Nulo,Nulo)
     Senão
          p1,p2 = divide-Rec (q->prox)
          q->prox = p1;   p1 = q
          Retorna (p2,p1)                          // inverte a ordem
```

Certo.

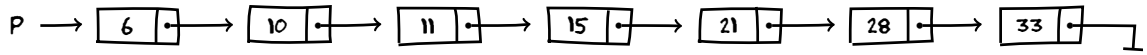
Agora nós estamos prontos para escrever a função `mergesort-Rec()`

```
func mergesort-Rec (q)

     Se (q == Nulo)   Retorna (Nulo)               // já ordenei ...
     Senão
          q1,q2 = divide-Rec (q)
          q1 = mergesort-Rec (q1)                  // ordena 1a metade
          q2 = mergesort-Rec (q1)                  // ordena 2a metade
          q = intercalação-Rec (q1,q2,Nulo,Nulo)
          Retorna (q)
```

◇

e. Par com soma k

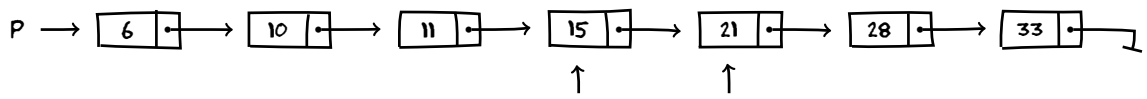
Imagine que o ponteiro p aponta para uma lista encadeada ordenada



A tarefa é

→ *Verificar se existem dois elementos cuja soma é igual a* k

No exemplo acima, para k = 36, isso nos daria



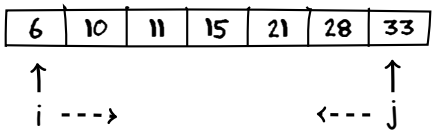

```
=>   Sim
```

Bom, é fácil resolver esse problema em tempo $O(n^2)$.

Mas nós queremos fazer em tempo $O(n)$.

E queremos fazer isso de maneira recursiva.

O problema é que a estratégia que resolve o problema em tempo $O(n)$ (no vetor) trabalha nas duas pontas da lista



Só que a nossa lista não é duplamente encadeada.

E agora?

Bom, agora é a hora para uma pequena esperteza.

Quer dizer, a ideia é fazer a coisa na volta.

E daí, a gente passa um ponteiro para o primeiro elemento como o parâmetro pri para a função.

Dessa maneira, a chamada que vê o último elemento também vê o primeiro elemento — (*por meio do parâmetro*)

E dessa maneira, ela pode implementar a lógica eficiente de solução do problema — (*trabalhando nas duas pontas*)

```
Se (pi->num + pj->num == k)   Achei
Se (pi->num + pj->num < k)    avança pi
Senão                         retrocede pj
```

Na prática, pi vai ser um valor de retorno.

E pj vai ser o elemento que a chamada vê.

A coisa fica assim

```
func PSK-Rec (q)                                              // Par com Soma K
    Se (q->prox == Nulo)                                      // a coisa começa aqui
        pi = pri;   pj = q
        Enquanto (pi->num < pj->num)                          // pi está antes de pj
            Se (pi->num + pj->num == k)  Retorna (Nulo,1)     // Achei
            Sse (pi->num + pj->num < k)  pi = pi->prox        // avança pi
            Senão                        Retorna (pi,0)       // retrocede pj
        Retorna (Nulo,0)
    Senão
        pi,resp = PSK-Rec (q->prox,k,pri)
        Se (pi == Nulo)  Retorna (pi,resp)
        pj = q
        Enquanto (pi->num < pj->num)                          // pi está antes de pj
            Se (pi->num + pj->num == k)  Retorna (Nulo,1)     // Achei
            Sse (pi->num + pj->num < k)  pi = pi->prox        // avança pi
            Senão                        Retorna (pi,0)       // retrocede pj
        Retorna (Nulo,0)
```