Estruturas de Dados

aula 08: Listas duplamente encadeadas

1 Introdução

Nós vimos que as listas encadeadas nos dão uma grande flexibilidade para inserir e remover elementos em qualquer posição da lista.

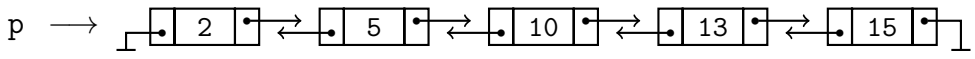
Mas, é muito desajeitado poder andar para a frente e não poder andar para trás.

Hoje nós vamos resolver isso.

Quer dizer, basta adicionar um ponteiro que aponta para trás ao registro `RegInt`

```
struct RegInt
{
     int             num

     struct RegInt   *ant, *prox
}
```

Daí, a coisa fica assim



Agora, é bem fácil realizar a seguinte tarefa

→ *Imprimir os elementos na ordem contrária*

Quer dizer, basta

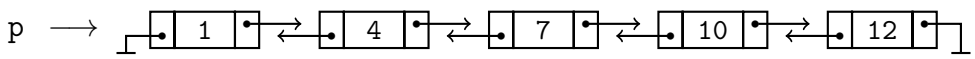
1. ir até o fim da lista
2. e depois voltar imprimindo os elementos

A coisa fica assim

```
func imprimeContrário (p)

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

     q = p
     Enquanto ( q->prox != Nulo )            // vai até o fim
          q = q->prox

     Enquanto ( q != Nulo )                  // e volta imprimindo
          Imprime ( q->num )
          q = q->ant
```

2 Listas duplamente encadeadas

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



A tarefa é

→ *inserir o número* `k` *na lista*

Bom, a novidade aqui é que nós não precisamos percorrer a lista com 2 ponteiros — (*e nem parar antes da hora ...*)

Quer dizer, nós percorremos a lista até encontrar o primeiro elemento maior do que `k`.

E fazemos a inserção atrás dele.

A única exceção é o caso em que o `k` entra na última posição da lista.

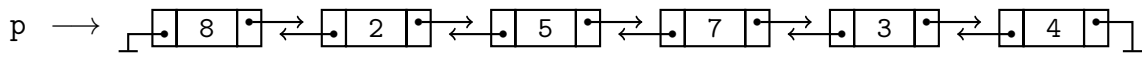
A coisa fica assim

```
func insereOrd (k,p)

     Se ( p == Nulo )                        // inserção na lista vazia
          r = Novo(RegInt)
          r->num = k
          r->ant,prox = Nulo,Nulo
          Retorna(r)

     q = p
     Enquanto ( q->prox != Nulo )            // procura o lugar do k
          Se (q->num > k)   Quebra
          q = q->prox

     Se ( q->num > k )                       // inserção atrás
          r = Novo(RegInt)
          r->num = k
          r->ant,prox = q->ant,q
          Se ( q->ant == Nulo )   p = r
          Senão                  (q->ant)->prox = r
          q->ant = r

     Senão                                   // inserção no fim
          r = Novo(RegInt)
          r->num = k
          r->ant,prox = q,Nulo
          q->prox = r

     Retorna(p)
```

Exemplos

a. A operação de remoção

Imagine que o ponteiro p aponta para uma lista duplamente encadeada

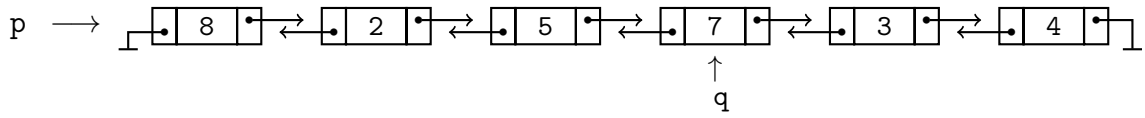


A tarefa é

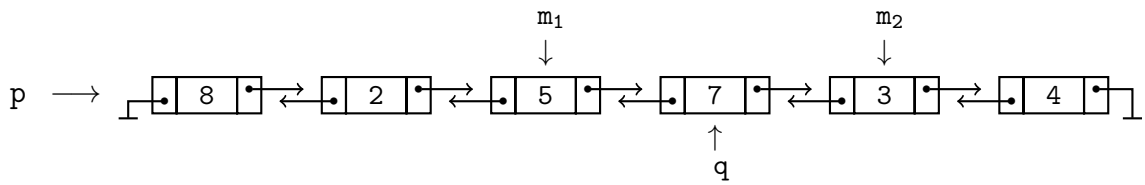
→ *remover o elemento* k *da lista*

Bom, imagine que nós queremos remover o 7 da lista acima.

Então, o primeiro passo é encontrar o 7



Daí, para fazer a remoção, nós podemos segurar o elemento que vem antes e o elemento que vem depois com duas mãozinhas



E agora é só ajustar os ponteiros — (*e liberar o registro do* 7)

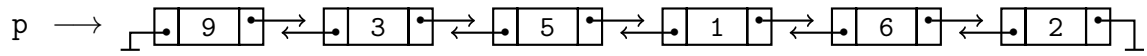
A coisa fica assim

```
func remoção (k, p)

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

     q = p
     Enquanto ( q != Nulo )
          Se (q->num == k)   Quebra          // Você é o kara?
          q = q->prox

     Se (q == Nulo)   Retorna (p)            // o k não está lá ...

     m1 = q->ant;   m2 = q->prox
     Se (m1 == Nulo)   p = m2
     Senão             m1->prox = m2
     Se (m2 != Nulo)   m2->ant = m1
     free (q)
     Retorna (p)
```

◇

b. O menor no início

Imagine que o ponteiro p aponta para uma lista duplamente encadeada



A tarefa é

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

Bom, aqui nós vamos fazer a coisa em etapas

1. primeiro a gente descobre quem é o menor elemento da lista
2. depois a gente remove esse elemento da lista
3. e daí a gente reinsere ele no início

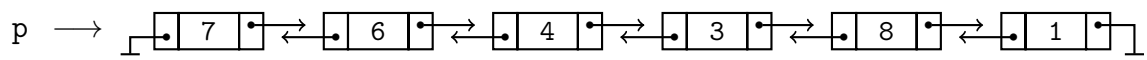
A coisa fica assim

```
func menorInicio (p)

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

     qm = p;    q = p                            // 1a ETAPA
     Enquanto ( q != Nulo )
          Se (q->num < qm->num)   qm = q
          q = q->prox

     m1 = q->ant;    m2 = q->prox                // 2a ETAPA
     Se (m1 == Nulo)   p = m2
     Senão             m1->prox = m2
     Se (m2 != Nulo)   m2->ant = m1

     (p->prox)->ant = qm                         // 3a ETAPA
     qm->ant,prox = Nulo,p
     p= qm

     Retorna (p)
```

◇

c. Separando pares e ímpares

Imagine que o ponteiro p aponta para uma lista duplamente encadeada



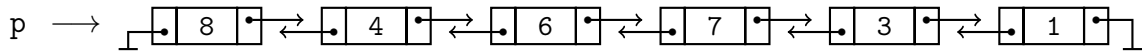
A tarefa é

→ *Colocar todos os elementos pares no início da lista*

e todos os elementos ímpares no final da lsita

— *(onde a ordem dos elementos pares e a ordem dos elementos ímpares não importa)*

No exemplo acima, isso poderia nos dar



Bom, a ideia é simples.

Quer dizer, nós vamos percorrer a lista.

E toda vez que a gente encontrar um elemento par, nós vamos remover ele da lista e reinserir no início.

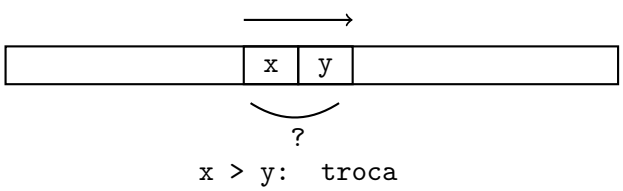
A coisa fica assim

```
func SPI (p)                                    // Separando Pares e Ímpares

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

     q = p
     Enquanto ( q != Nulo )
          Se (q->num é par  E  q->ant != Nulo)
               m1 = q->ant;   m2 = q   m3 = q->prox
               m1->prox = m3
               Se (m3 != Nulo)   m3->ant = m1
               m2->ant,prox = Nulo,p
               p = m2
               q = m3
          Senão
               q = q->prox

     Retorna (p)
```

◇

d. A operação de varredura

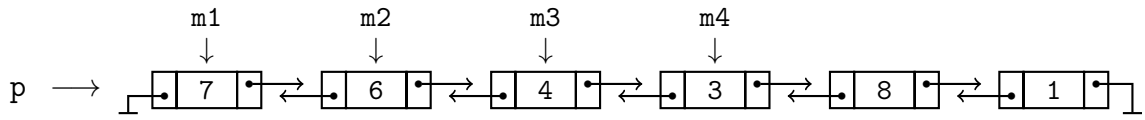
Lembre que a operação de varredura percorre a lista comparando elementos consecutivos, e fazendo a troca sempre que o elemento da esquerda é maior que o elemento da direita



Agora que temos a lista duplamente encadeada, nós podemos implementar essa operação de

maneira mais natural — (*i.e., percorrendo a lista e trocando os elementos de lugar*)

Para fazer isso, nós vamos utilizar 4 mãozinhas



Quer dizer, a cada passo nós comparamos os elementos apontados por `m2` e `m3`.

Daí, se o elemento apontado por `m2` for maior, as mãozinhas `m1` e `m4` nos ajudam a fazer a troca.

Depois disso, cada mãozinha dá um passo para frente.

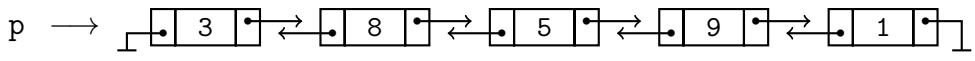
A coisa fica assim

```
func Varredura (p)

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

      m1 = Nulo;    m2 = p;    m3 = m2->prox;    m4 = m3->prox

      Enquanto ( m3 != Nulo )
            Se (m2->num > m3->num)
                  m2->ant = m3;    m3->prox = m2       // ajusta ponteiros de m2 e m3
                  m2->prox = m4;    m3->ant = m1       //

                  Se (m1 == Nulo)   p = m3             //
                  Senão              m1->prox = m3     // ajusta ponteiros de m1 e m4
                  Se (m4 == Nulo)   m4->ant = m2       //

                  aux = m2                             //
                  m2 = m3                              // troca m2 e m3
                  m3 = aux                             //

                  m1 = m2;    m2 = m3;    m3 = m4      // as mãozinhas andam
                  Se (m4 != Nulo)   m4 = m4->prox      //

      Retorna (p)
```

◇

e. Invertendo a ordem da lista

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



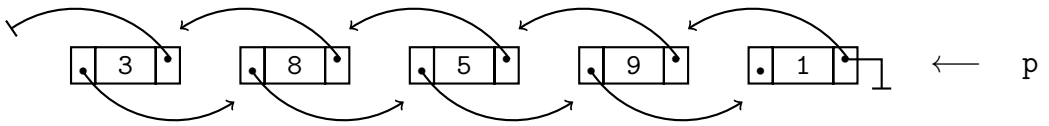
A tarefa é

→ *inverter a ordem dos elementos da lista*

Na aula passada nós vimos que essa operação é um pouco enrolada.

Mas aqui ela é bem fácil.

Quer dizer, basta percorrer a lista trocando os ponteiros `ant` e `prox` de cada elemento — (*e apontando o ponteiro* `p` *para o último elemento*)



O único cuidado é percorrer a lista com dois ponteiros — (*para não começar a andar para trás depois da inversão ...*)

A coisa fica assim

```
func inverteLista (p)

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

     q1 = p;    q2 = p->prox

     Enquanto ( q1 != Nulo )
          aux = q1->ant                          //
          q1->ant = q1->prox                     // troca ponteiros ant e prox
          q1->prox = aux                         //

          Se (q2 == Nulo)   m3->ant = m1
               p = q1;    q1 = Nulo              // o último vira o primeiro
          Senão
               q1 = q2;    q2 = q2->prox

     Retorna (p)
```