Estruturas de Dados

aula 05: Listas encadeadas

1 Introdução

Nas últimas aulas nós encontramos o dilema da ordenação

- a busca na lista ordenada é extremamente rápida — (*i.e., tempo* $O(\log n)$)
- mas a inserção e a remoção são uma porcaria — (*i.e., tempo* $O(n)$)

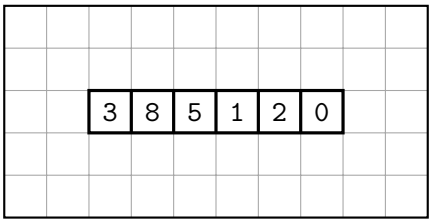
E para escapar desse dilema é preciso entender melhor o que está acontecendo.

Quer dizer, é a mesma razão que está por trás dos dois fatos acima.

Daí que, se a gente quer ter o primeiro, a gente acaba tendo o segundo também.

Mas, que razão é essa?

Bom, o ponto é que a eficiência da busca vem do fato de que os elementos da lista ordenada estão muito bem arrumadinhos na memória



E daí, se o elemento do meio é maior do que o número que a gente está procurando, então todos os que vem depois dele também são — (*e não é preciso nem olhar para eles ...*)

É isso o que torna a busca super-eficiente.

Mas, a gente só sabe onde o elemento do meio está porque os elementos da lista estão todos arrumadinhos — (*um depois do outro, em posições consecutivas na memória*)

O problema é que nessa arrumação não há espaço (vazio) para inserir um novo elemento.

E se a gente remover um elemento da lista, é preciso arrumar tudo de novo.

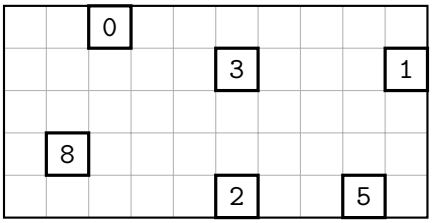
É isso o que torna a inserção e a remoção na lista ordenada uma porcaria.

1.1 Fazendo as coisas de maneira bagunçada

Legal.

Agora que a gente entendeu é o problema, é fácil encontrar a sua solução.

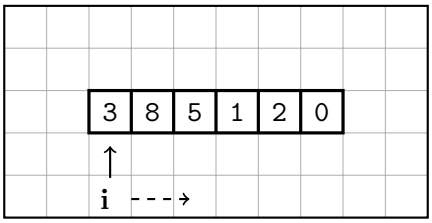
Quer dizer, se manter as coisas arrumadinhas é o que está criando o problema, então a solução é deixar tudo bagunçado



Mas aqui surge um outro problema, porque é preciso saber onde as coisas estão.

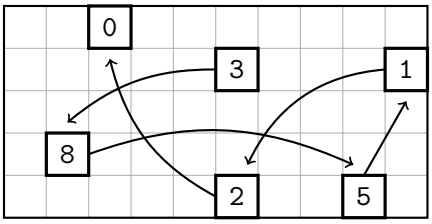
Quer dizer, na organização de lista basta saber onde está o primeiro elemento.

E daí, um dedo que anda para a direita encontra todos os demais



Ora, então a gente pode usar o dedo para resolver o nosso problema também.

Ou melhor, um montão de dedos



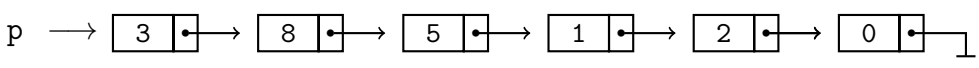
Quer dizer, a ideia é que agora cada elemento tem um dedo que indica o lugar onde está o próximo elemento.

Então aqui também, basta saber onde está o primeiro elemento para encontrar todos os outros — (*com um dedo que vai pulando pelos elementos um por um ...*)

Legal, não é?

Essa organização é chamada de *lista encadeada.*

E nós vamos representá-la assim



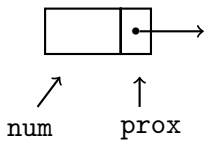
— (*apesar de na prática os elementos estarem todos espalhados pela memória ...*)

1.2 Implementação

Para implementar a lista encadeada, nós precisamos de uma estrutura de dados que armazena um número e um dedo (ou ponteiro).

Essa estrutura é o registro (ou *struct*, na linguagem C)

```
struct RegInt
    int   num;
    struct RegInt *prox
```



A ideia aqui é que o campo `prox` é um ponteiro que aponta para caixinhas (ou registros) do tipo `RegInt`.

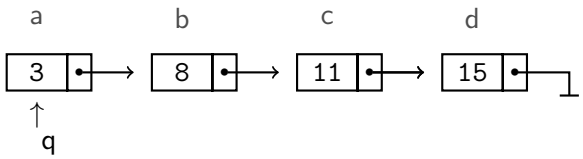
E é dessa maneira que a gente monta a nossa lista encadeada.

Por exemplo, o código abaixo cria uma lista encadeada com 4 elementos

```
struct RegInt   a, b, c, d;     // 4 registros
a.num = 3
a.prox = &b                     // o ponteiro de a aponta para b
b.num = 8
b.prox = &c                     // o ponteiro de b aponta para c
c.num = 11
c.prox = &d                     // o ponteiro de c aponta para d
d.num = 15
d.prox = Nulo                   // e d não aponta pra lugar nenhum
```

— (*na próxima aula nós vamos aprender a criar listas de qualquer tamanho*)

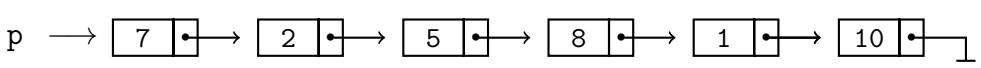
Daí, uma vez que nós já temos a lista na memória, basta saber onde o primeiro elemento está para fazer um dedo percorrer os demais



A seguir nós vamos ver como a coisa funciona.

2 Listas encadeadas

Imagine que o ponteiro `p` aponta para uma lista encadeada que está armazenada na memória



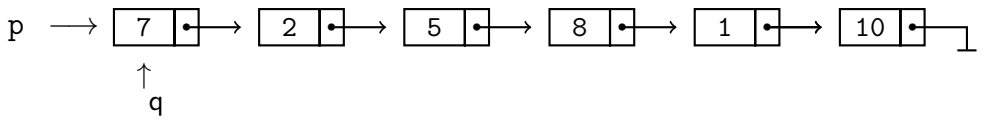
A tarefa é

→ *Imprimir todos os elementos da lista na tela*

Bom, para isso a gente vai precisar de um ponteiro auxiliar `q`.

Porque se o ponteiro `p` sair do lugar, a gente não vai mais saber onde está o primeiro elemento.

Então, o primeiro passo é apontar o ponteiro `q` para o primeiro elemento



E agora, o ponteiro `q` pode percorrer a lista imprimindo todos os elementos.

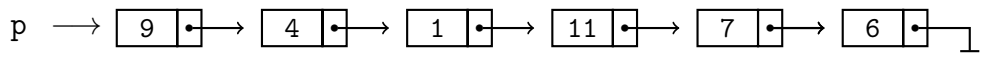
A coisa fica assim

```
q = p                                   // aponta para o 1o elemento
Enquanto ( q != Nulo )                  // vai até o fim da lista
    Imprime(q->num)
    q = q->prox                         // pula para o próximo
```

Exemplos

a. 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, uma lista encadeada não tem posições numeradas — (*como os vetores*)

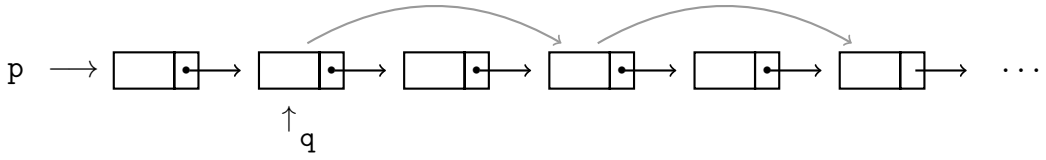
Então para realizar a tarefa, a gente pode ir contando as posições a medida que a gente vai passando por elas.

A coisa fica assim

```
q = p;    pos = 0                       // a 1o posição é a posição 0
Enquanto ( q != Nulo )                  // percorre a lista
    Se (pos é impar)
        Imprime(q->num)
    q = q->prox;    pos++               // anda o ponteiro e a posição
```

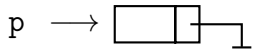
Mas, essa não é a única maneira de fazer as coisas.

Quer dizer, outra ideia é começar na posição 1 e ir pulando de 2 em 2

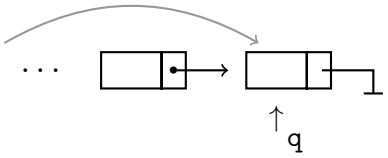


Mas aqui é preciso ter algum cuidado.

Porque pode não haver ninguém na posição 1



E porque não é possível dar um pulo de tamanho 2 a partir da última posição



Porque?

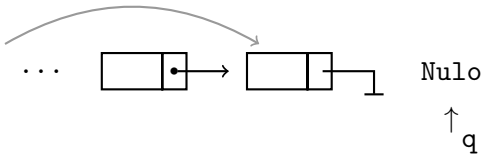
Ora, porque na prática o pula de tamanho 2 são dois pulos de tamanho 1

```
q = q->prox
q = q->prox
```

o que também pode ser escrito assim

```
q = (q->prox)->prox
```

Daí, quando a gente está na última posição, o primeiro pulo leva a gente para o `Nulo`



E não se pode dar um pulo a partir do `Nulo` — (*porque o* `Nulo` *não é uma caixinha que tem um ponteiro* `prox`)

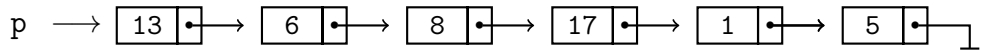
Abaixo nós temos a segunda versão do nosso algoritmo

```
Se (p->prox != Nulo)                    // existe a posição 1?
    q = p->prox                         // a 1o posição é a posição 0
    Enquanto ( q != Nulo )
        Imprime(q->num)
        q = q->prox                     //
        Se (q != Nulo)                  // pulando de 2 em 2
            q = q->prox                 //
```

◇

b. A soma dos elementos ímpares

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *Calcular a soma dos elementos das posições ímpares da lista*

Bom, aqui basta percorrer a lista com o dedo inspecionando os elementos um a um.

Quando o elemento for ímpar, a gente atualiza a soma.

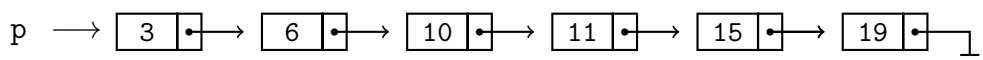
A coisa fica assim

```
q = p;    soma = 0
Enquanto ( q != Nulo )
    Se (q->num é impar)              // Você é impar?
        soma = soma + q->num         // atualiza a soma
    q = q->prox
Imprime('A soma é', soma)
```

◇

c. Procurando o 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, aqui a gente aplica a nossa estratégia padrão: percorrer a lista com o dedo inspecionando os elementos um a um.

A coisa fica assim

```
O(1)  q = p;    achei = 0
      Enquanto ( q != Nulo )
O(n)      Se (q->num == k)                    // Você é o kara?
O(1)          achei = 1;    Quebra
          q = q->prox
O(1)  Se (achei == 1)   Imprime(Achei!)
      Senão             Imprime(Não está lá ...)
```

Análise

Dessa vez, nós anotamos o código para estimar o tempo de execução do algoritmo.

E aqui nós temos uma má notícia.

Quer dizer, o tempo de execução é

$$O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad = \quad O(n)$$

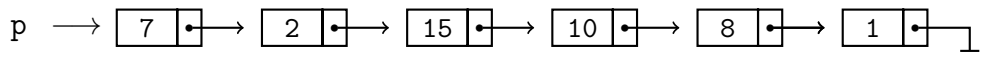
Note que a lista está ordenada.

Mas com as listas encadeadas, não há como tirar vantagem da ordenação para tornar a busca mais rápida — (*por enquanto ...*)

◇

d. Encontrando o maior elemento

Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

→ *Encontrar o maior elemento da lista — (e apontar um dedo para ele)*

Aqui mais uma vez, a gente pode aplicar a estratégia padrão: percorrer a lista com o dedo, lembrando o maior elemento que a gente viu até o momento.

A coisa fica assim

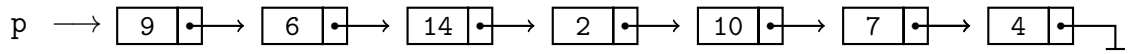
```
q = p
pmaior = q                              // o maior até aqui
Enquanto ( q != Nulo )
    Se (q->num > pmaior->num)           // Você é maior que o maior?
        pmaior = q
    q = q->prox
Imprime('O maior elemento é', pmaior->num)
```

◇

e. Encontrando o elemento do meio

Imagine que o ponteiro p aponta para uma lista encadeada.

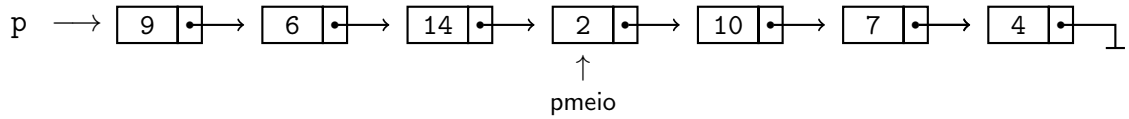
E imagine que a lista tem tamanho ímpar



A tarefa é

→ *Encontrar o elemento central da lista — (e apontar um dedo para ele)*

No exemplo acima, isso nos daria

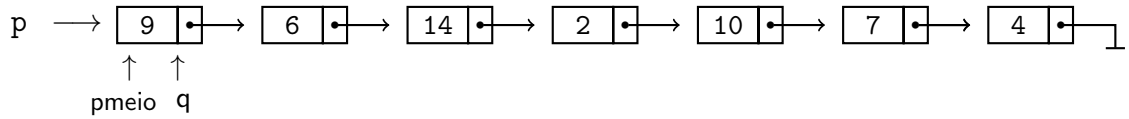


Bom, a estratégia padrão aqui é

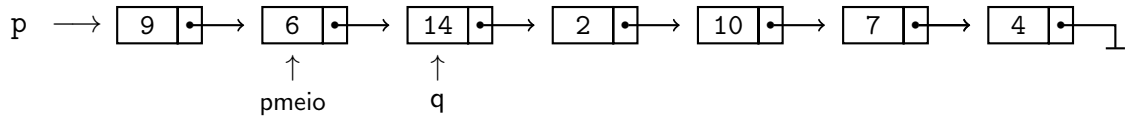
- percorrer a lista contando a quantidade de elementos
- calcular a posição do elemento central
- e depois levar o ponteiro pmeio até lá

Mas, a gente pode fazer uma coisa mais legal.

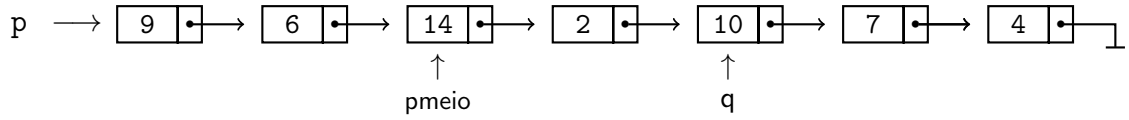
A ideia é colocar dois dedos no primeiro elemento



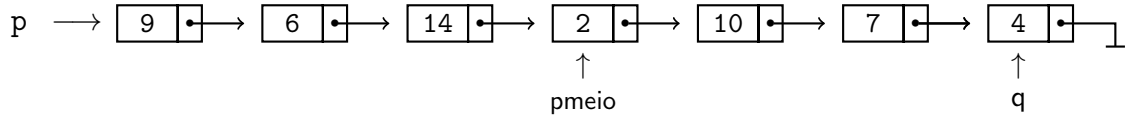
Daí, o ponteiro q dá um pulo de tamanho 2, enquanto o ponteiro pmeio dá um pulo de tamanho 1



A seguir, a gente faz a mesma coisa



E continua fazendo isso até que o ponteiro q alcance a última posição da lista



A coisa pára quando o ponteiro q não consegue mais dar o seu pulo de tamanho 2.

E nesse momento, o ponteiro pmeio está apontando para o elemento central.

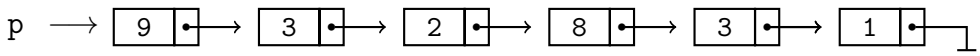
Abaixo nós temos o código desse algoritmo

```
q = p;    pmeio = p                    // dois ponteiros no 1o elemento
Enquanto ( q != Nulo )
    q = q->prox                        //
    Se ( q != Nulo )                   // pulo de tamanho 2
        q = q->prox                    //
    Senão    Quebra                    // ou Quebra
    pmeio = pmeio->prox
Imprime('O elemento do meio é', pmeio->num)
```

◇

f. Procurando um elemento repetido

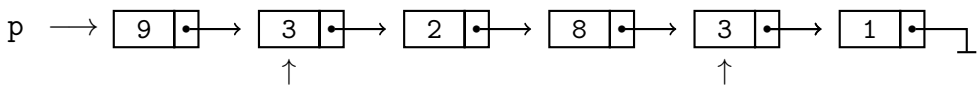
Imagine que o ponteiro p aponta para uma lista encadeada



A tarefa é

- Verificar se a lista contém algum elemento repetido

No exemplo acima, isso nos daria



=> Sim

Bom, a estratégia padrão aqui é

- percorrer a lista com o dedo
- para cada elemento

  verificar se ele aparece outra vez mais adiante

E para fazer essa verificação, a gente vai precisar de outro dedo.

A coisa fica assim

```
q = p;    achei = 0
Enquanto ( q != Nulo )
    r = q->prox                        // o segundo dedo
    Enquanto ( r != Nulo )
        Se ( r->num == q->num )        // Você é repetido?
            achei = 1    Quebra
        r = r->prox
        Se ( achei == 1 )    Quebra
        q = q->prox
pmeio = pmeio->prox
Imprime('O elemento do meio é', pmeio->num)
```