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* $O(n)$)

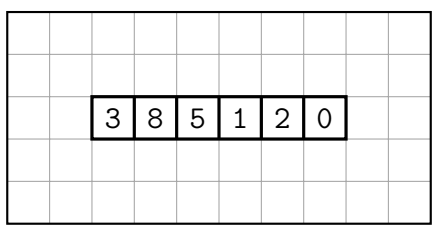
E para escapar desse dilema é preciso entender melhor o que está acontecendo.

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.

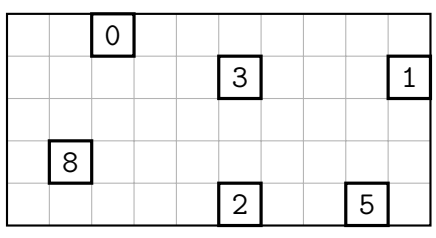
É isso o que torna a inserção e a remoção na lista ordenada uma porcaria.

Fazendo as coisas de maneira bagunçada

Legal.

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

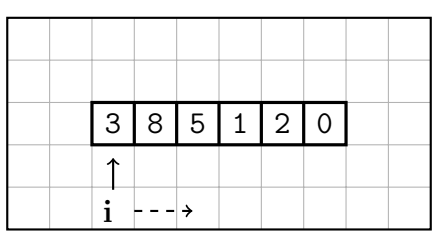
Quer dizer, se manter as coisas arrumadinhas é o que está criando o problema, então a solução é deixar tudo bagunçado



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

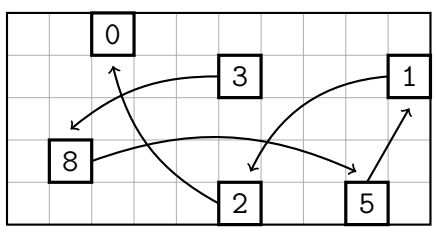
Quer dizer, na organização de lista basta saber onde está o primeiro elemento.

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



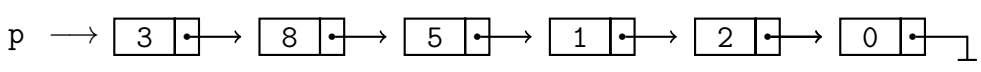
Quer dizer, a ideia é que agora cada elemento tem um dedo que indica o lugar onde está o próximo elemento.

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



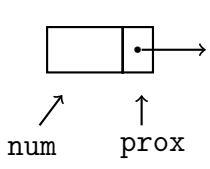
— (*apesar de na prática os elementos estarem todos espalhados pela memória ...*)

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`.

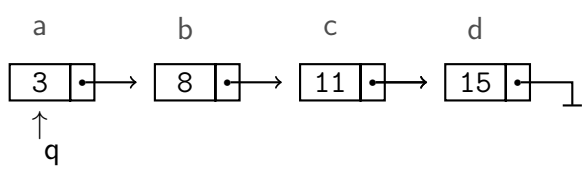
E é dessa maneira que a gente monta a nossa lista encadeada.

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.