Estruturas de Dados

aula 09: A lista vai-e-volta e outra maluquices

1 Introdução

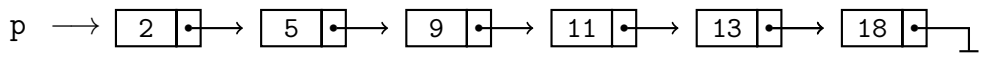
Todos os algoritmos que nós vimos para listas encadeadas executam em tempo $O(n)$ — (*e são uma grande porcaria*)

Hoje nós vamos ver algumas ideias que nos permitem obter algoritmos mais eficientes.

E no final das contas, elas são as mesmas ideias que nós utilizamos nos vetores.

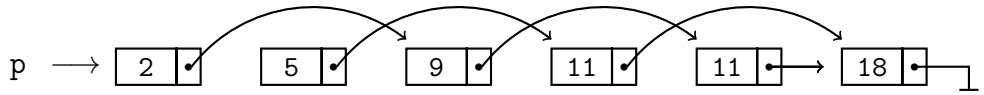
2 A lista vai-e-volta

Imagine que nós temos uma lista encadeada ordenada



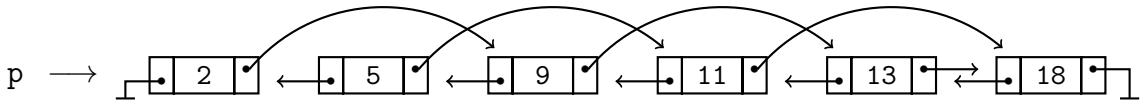
Para encontrar o número `k` a gente percorre a lista com o dedo — (*até encontrar ele, ou alguém maior do que ele*)

E a gente pode ir mais rápido pulando de 2 em 2



Mas assim, a gente pode passar do ponto.

Então é preciso ter um ponteiro para poder voltar



Essa é a *lista vai-e-volta.*

Aqui, a busca percorre a lista pulando de 2 em 2, até

- encontrar o número `k`
- encontrar alguém maior do que `k`
- ou chegar ao último elemento

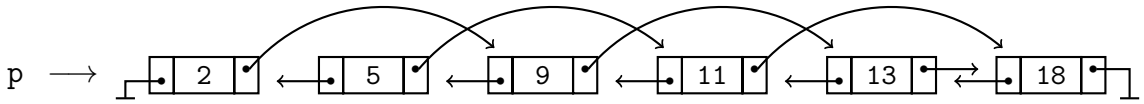
Daí, caso o `k` não tenha sido encontrado, a gente dá um passo atrás — (*se houver alguém atrás ...*)

A coisa fica assim

```
func busca-VV (k,p)

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

      q = p
      Enquanto ( q->prox != Nulo )                   // vai até o fim
            Se ( q->num == k )   Retorna (q)
            Se ( q->num > k )    Quebra
            q = q->prox                              // pulando de 2 em 2

      Se ( q->num == k )   Retorna (q)
      Senão
            Se ( q->ant != Nulo )   q = q->ant       // dá um passo atrás
            Se ( q->num == k )      Retorna (q)
            Senão                   Retorna (Nulo)
```

2.1 Remoção na lista vai-e-volta

Imagine agora que nós queremos remover o 11 da nossa lista

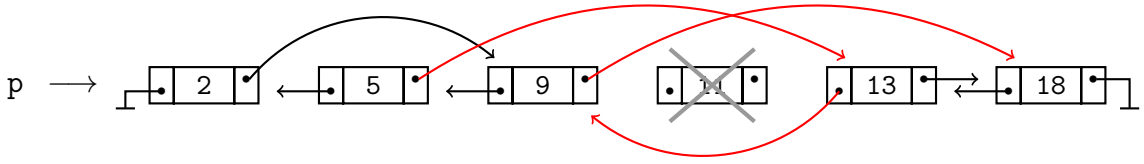


Então, andando pra frente

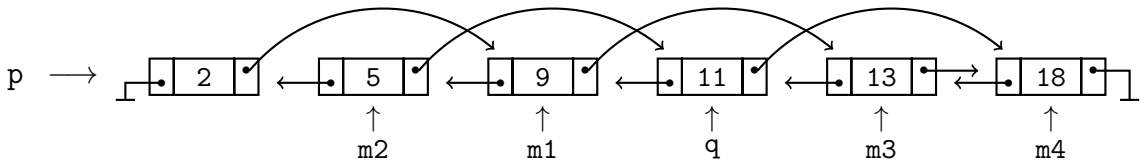
- o 5 precisa apontar para o 13
- o 9 precisa apontar para o 18

e andando para trás

- o 13 precisa apontar para o 9



Para implementar esse procedimento nós fazemos a busca pelo `k`, e depois colocamos 4 mãozinhas

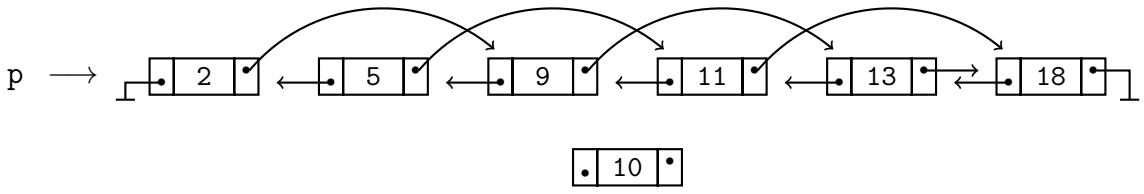


A coisa fica assim

```
func remoção-VV (k,p)

     q = busca-VV (k,p)
     Se ( q == Nulo )   Retorna (p)              // Não está lá ...

     m1 = q->ant                                 // colocando as mãozinhas
     Se ( m1 != Nulo )   m2 = m1->ant
     Senão               m2 = Nulo

     Se ( q->prox == Nulo )                      // q é o último
          m3 = Nulo;   m4 = Nulo
     Se ( (q->prox)->ant == q )                  // q é o penúltimo
          m3 = q->prox;   m4 = q->prox
     Senão
          m4 = q->prox;   m3 = m4->ant

     Se ( m3 != Nulo )   m3->ant = m1            // ajustando os ponteiros
     Se ( m1 != Nulo )   m1->prox = m4
     Senão               p = m3
     Se ( m2 != Nulo )   m2->prox = m3

     free (q)
     Retorna (p)
```

2.2 Inserção na lista vai-e-volta

Imagine agora que nós queremos inserir o 10 na nossa lista vai-e-volta

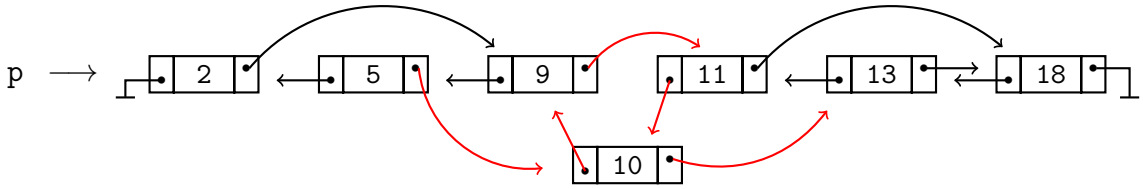


Então, andando para frente

- o 5 precisa apontar para o 10
- o 9 precisa apontar para o 11
- o 10 precisa apontar para o 13

e andando para trás

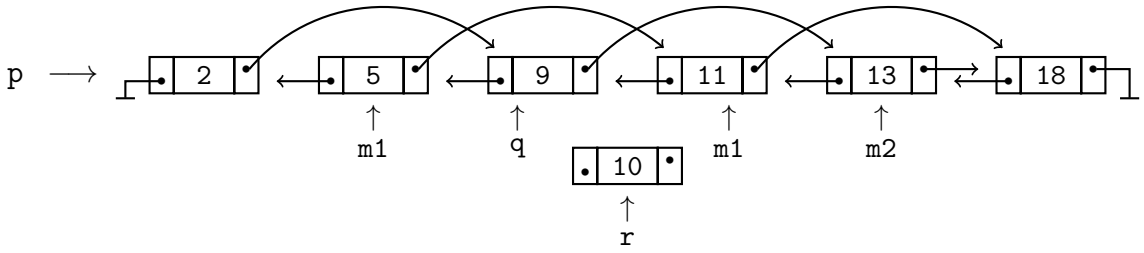
- o 11 precisa apontar para o 10
- o 10 precisa apontar para o 9



Para implementar esse procedimento, nós vamos utilizar uma função que retorna um ponteiro para o maior elemento da lista que é menor do que `k` — (*ou Nulo, se esse elemento não existe*)

Exercício: Implementar esse procedimento de busca

E daí, nós colocamos 3 mãozinhas



A coisa fica assim

```
func inserção-VV (k,p)

      r = Novo(RegInt)                                        // cria o registro
      r->num = k

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

      q = buscaMmk-VV(k,p)

      Se ( q == Nulo )                                        // inserção na 1a posição
            r->ant = Nulo;    p->ant = r
            Se ( p->prox == Nulo )          r->prox = p       // p é o último
            Sse ( (p->prox)->ant == p )     r->prox = p->prox // p é o penúltimo
            Senão                    r->prox = (p->prox)->ant
            Retorna (r)

      Senão                                                   // inserção no meio
            m1 = q->ant
            Se ( q->prox == Nulo )          m3 = Nulo;    m4 = Nulo
            Sse ( (q->prox)->ant == q )     m3 = q->prox;    m4 = q->prox
            Senão                           m4 = q->prox;    m3 = m4->ant
```

```
r->prox = m3;    r->ant = q
m1->prox = r;    q->prox = m2
Se ( m2 != Nulo )
      q->prox = m2;    m2->ant = r
     Senão        q->prox =r
Retorna (p)
```

2.3 Análise de complexidade

E então, será que vale a pena?

Bom, em listas muito grandes o esquema vai-e-volta pode reduzir o tempo de execução dos algoritmos à metade.

Mas, a metade de $O(n)$ ainda é $O(n)$.

E daí, a gente não ganhou nada.

Só que, o mesmo esquema pode ser utilizado para ir pulando de 3 em 3.

Ou ir pulando de 4 em 4.

Isso reduz o tempo dos algoritmos ainda mais.

Mas não livra a gente do $O(n)$.

A solução então é fazer cada elemento apontar $\sqrt{n}$ posições à frente.

Dessa maneira, a gente dá no máximo $\sqrt{n}$ pulos para percorrer a lista inteira.

E volta no máximo $\sqrt{n}$ posições quando a gente passa do ponto.

Com esse esquema, os algoritmos de busca, inserção e remoção passam a executar em tempo $O(\sqrt{n})$ — (*o que já não é mais uma porcaria ...*)

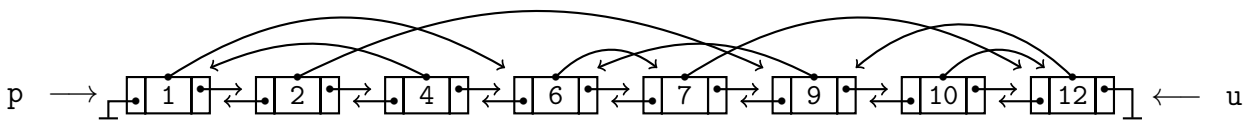
3 A lista com ponteiros aleatórios

Nós acabamos de ver que a lista duplamente encadeada nos dá mais flexibilidade para fazer as coisas.

Ora, mas se isso é o caso, então agora nós podemos considerar uma lista triplamente encadeada.

Quer dizer, a ideia é que nós vamos ter os ponteiros que apontam para frente e para trás.

E vamos ter um terceiro ponteiro que aponta para um lugar aleatório na lista



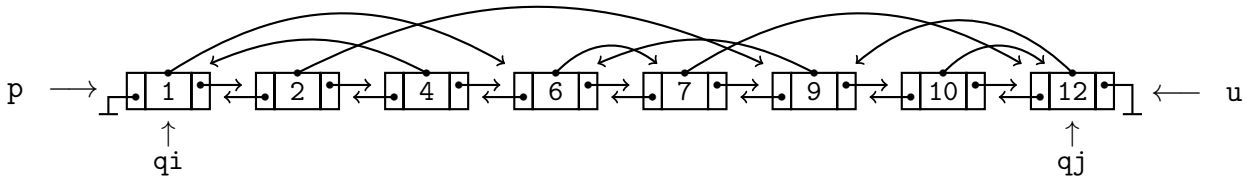
Mas, pra que isso?

Bom, a ideia é que os ponteiros para frente e para trás nos permitem andar pela lista dando passos pequenos.

E os ponteiros aleatórios permitem dar saltos grandes, para chegar mais rápido a um local distante.

Daí, a estratégia de busca é semelhante à busca binária.

Quer dizer, nós começamos colocando dois ponteiros q_i e q_j no primeiro e no último elementos da lista



A cada passo o ponteiro q_i tem 3 lugares para ir

- o próximo elemento
- o elemento apontado pelo ponteiro aleatório de q_i
- o elemento apontado pelo ponteiro aleatório de q_j

E nós escolhemos aquele que está mais próximo do número `k` — (*mas não excede esse valor*)

O ponteiro q_j se movimenta da mesma maneira — (*na direção contrária ...*)

A coisa fica assim

```
func busca-PA (k, p, u)

     Se ( p == Nulo )   Retorna (Nulo)                        // Não está lá ...

     qi = p;    qj = u                                        // posiciona os dedos
     Enquanto ( qi->num ≤ qj->num )
          Se ( qi->num == k )   Retorna (qi)
          Se ( qj->num == k )   Retorna (qj)
          qi = Maior-menor (k, qi->prox, qi->ale, qj->ale)     // avança qi
          qj = menor-Maior (k, qi->prox, qi->ale, qj->ale)     // avança qj

     Retorna (p)
```

onde

- a função `Maior-menor()` retorna o ponteiro que aponta para o maior valor menor ou igual a `k` — (*ou então* `prox`)
- a função `menor-Maior()` retorna o ponteiro que aponta para o menor valor maior ou igual a `k` — (*ou então* `ant`)

3.1 Análise de complexidade

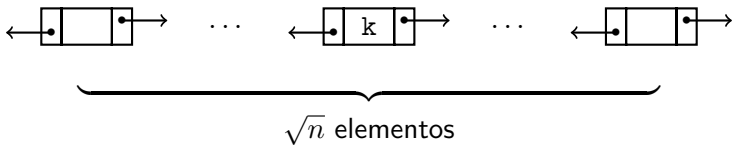
A ideia é que no início do processo, os ponteiros aleatórios nos ajudam a reduzir o intervalo da busca rapidamente.

Daí, quando o intervalo já está suficientemente pequeno, os ponteiros aleatórios já não ajudam mais.

Só que nesse momento, nós podemos utilizar os ponteiros `prox` e `ant` para alcançar o elemento que está sendo procurado.

Para tornar essa ideia mais concreta, nós vamos dizer que um intervalo é *pequeno* se ele contém no máximo $\sqrt{n}$ elementos.

Daí, a gente concentra a nossa atenção em um intervalo pequeno em torno do elemento `k`



E daí a gente pergunta: *Qual é a probabilidade de que um ponteiro aleatório aponte para um elemento dentro desse intervalo?*

Bom, o ponteiro aleatório pode apontar para qualquer um dos n elementos da lista.

E existem $\sqrt{n}$ dentro do intervalo.

Logo, a probabilidade de que o ponteiro aponte para alguém dentro do intervalo é

$$\frac{\sqrt{n}}{n} \quad = \quad \frac{1}{\sqrt{n}}$$

Certo.

A próxima pergunta é: *Quantos ponteiros aleatórios nós precisamos examinar (em média), até encontrar um que aponte para alguém dentro do intervalo?*

Bom, a moeda que dá Cara ou Coroa com probabilidade 1/2 cada precisa ser lançada 2 vezes (em media), até que a gente obtenha o resultado Cara.

E o dado que tem probabilidade 1/6 associada a cada face precisa ser lançado 6 vezes (em média), até que a gente obtenha o número 6.

O ponteiro aleatório é como uma moeda que dá Cara com probabilidade $1/sqrtn$ — (*ou um dado com n faces, onde* $\sqrt{n}$ *delas estão marcadas com o 6 ...*)

Logo, nós precisamos examinar $\sqrt{n}$ ponteiros aleatórios (em média), até encontrar um que aponte para alguém dentro do intervalo.

Mas, isso significa que nós conseguimos posicionar $\mathtt{q_i}$ ou $\mathtt{q_j}$ dentro do intervalo em tempo $O(\sqrt{n})$ — (*em média*)

E com mais $O(\sqrt{n})$ voltas no laço, nós conseguimos encontrar o elemento `k` — (*ou descobrir que ele não está lá ...*)

Portanto, a busca na lista com ponteiros aleatórios executa em tempo $O(\sqrt{n})$.

Legal, não é?

3.2 Remoção na lista com ponteiros aleatórios

Sim, mas aqui nós temos um problema.

Quer dizer, nós não podemos remover os elementos da lista com ponteiros aleatórios.

Porque pode haver um elemento (ou mais de um) em algum lugar da lista que aponta para esse elemento — (*com o seu ponteiro aleatório*)

E daí, se o elemento é removido (e o seu registro liberado da memória), nós vamos obter um erro de execução quando estivermos utilizando um ponteiro aleatório que aponta para ele.

E agora?

Bom, agora a gente adiciona dois novos campos ao nosso registro

- `apagado`: indica que esse elemento já foi removido da lista — (*mas o seu registro continua lá*)
- `cont`: indica quantos ponteiros aleatórios estão apontando para esse elemento

A ideia é que se `cont` $= 0$, então o registro pode ser retirado da lista e liberado da memória — (*no procedimento de remoção*)

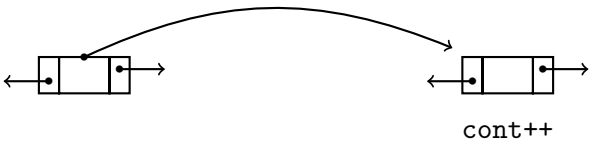
Mas se `cont` > 0, então o registro continua lá e é apenas marcado como apagado — (*utilizando o campo* `apagado`)

Os registros apagados não são levados em conta no procedimento de busca — (*mas eles podem servir de passagem para os ponteiros* q_i *e* q_j)

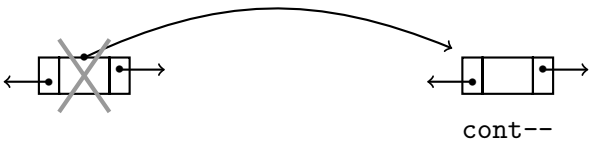
O problema com essa solução é que o número de registros apagados na lista pode ficar muito grande — (*afetando o tempo de execução da busca*)

Mas, esse problema pode ser controlado.

No momento em que o ponteiro aleatório de um novo elemento aponta para um registro, o campo `cont` desse registro é incrementado de 1



Mas no momento em que esse elemento é removido da lista (ou marcado como apagado), nós podemos decrementar o contador de 1



Dessa maneira, o campo `cont` de um elemento que já foi apagado pode ir diminuindo.

E quando ele chega em zero, o seu registro pode ser removido da lista.

Abaixo nós temos o código do procedimento de remoção

```
func remocao-PA (k,p,u)

      q = busca-PA (k,p,u)
      Se ( q == Nulo )   Retorna (Nulo)

      Se ( q->cont == 0 )
      m1 = q->ant;   m2 = q->prox
            Se ( m1 != Nulo )   m1->prox = m2
            Senão               p = m2
            Se ( m2 != Nulo )   m2->ant = m1
            Senão               u = m1
            (q->ale)->cont --
            free (q)
      Senão
            q->apagado = 1
            (q->ale)->cont --

      Retorna (p,u)
```

4 Inserção na lista com ponteiros aleatórios

A inserção na lista com ponteiros aleatórios é basicamente a mesma coisa que a inserção na lista duplamente encadeada

1. primeiro a gente realiza a busca para encontrar o lugar onde o elemento deve ser inserido
2. e depois nós fazemos a sua inserção nesse lugar

A única novidade é a inicialização do ponteiro aleatório.

Como é que a gente faz isso?

Bom, uma ideia simples consiste em realizar um passeio aleatório na lista

- a gente começa no primeiro elemento
- e a cada passo, a gente escolhe (aleatoriamente) um dos ponteiros para continuar: `prox`, `ant` ou `ale`.

Depois de um certo número de passos, a gente se encontra em um lugar aleatório da lista.

E a gente utiliza esse lugar para inicializar o ponteiro aleatório do novo elemento.