Estruturas de Dados

aula 14: As árvores binárias de busca

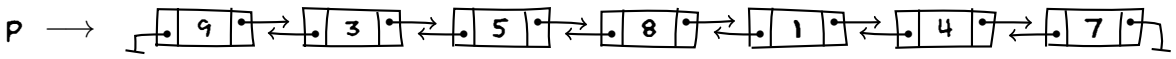
1 Introdução

A gente ainda não conseguiu nada de bom com os ponteiros e as listas encadeadas — (*apenas alguns algoritmos de tempo* $O(n\sqrt{n})$)

Mas hoje nós vamos chegar bem perto disso.

E a ideia é bem simples.

Quer dizer, vamos olhar outra vez para a lista duplamente encadeada



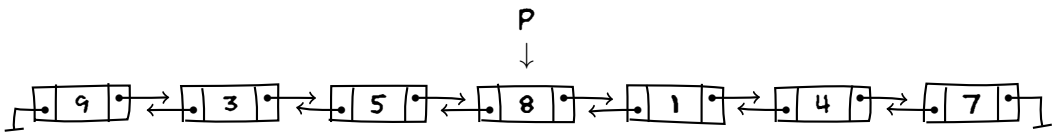
E daí a gente pergunta

→ *Porque o ponteiro* `p` *está apontando para o primeiro elemento da lista ?*

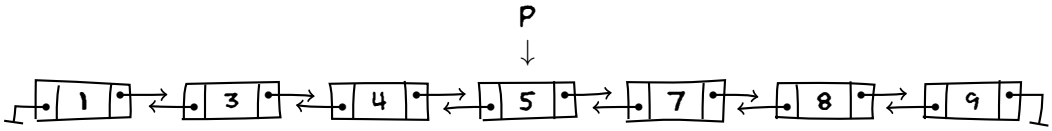
Quer dizer, como a lista é duplamente encadeada ele podia apontar para qualquer um — (*porque de qualquer um é possível chegar em qualquer um*)

De fato, o primeiro de todos é o pior elemento que o ponteiro `p` podia apontar — (*porque ele fica muito longe do último ...*)

E agora que a gente viu isso, é fácil ver também que o melhor lugar para apontar é o elemento central — (*porque?*)



E a coisa fica ainda melhor se a lista está ordenada



porque daí, basta olhar para o elemento central para a gente saber se deve ir para a esquerda ou para a direita.

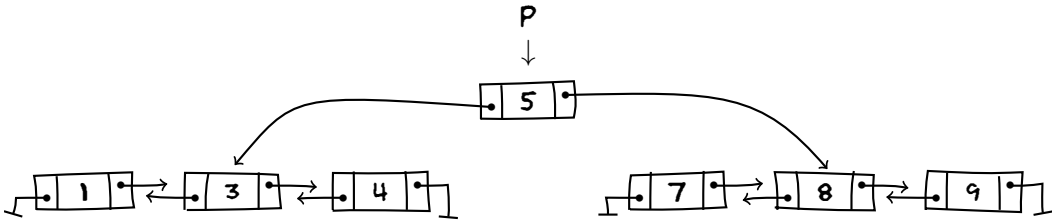
Legal.

Mas agora a gente pode perguntar

→ *Porque o elemento central está apontando para os elementos que estão ao seu lado ?*

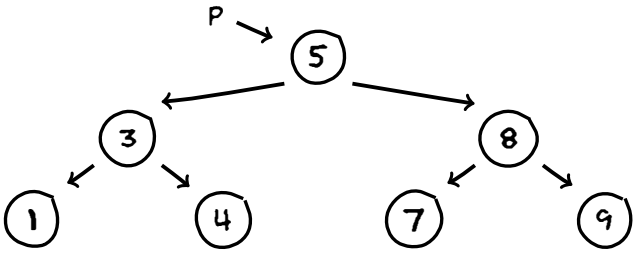
Bom, a razão é que não há nenhuma razão para isso — (*e isso tampouco é uma boa ideia*)

Então agora, a gente reorganiza a nossa estrutura de dados assim



Já deu pra ver o que está acontecendo, não é?

Daí, aplicando o mesmo truque até o fim (e trocando os quadradinhos por bolinhas), a gente chega no seguinte

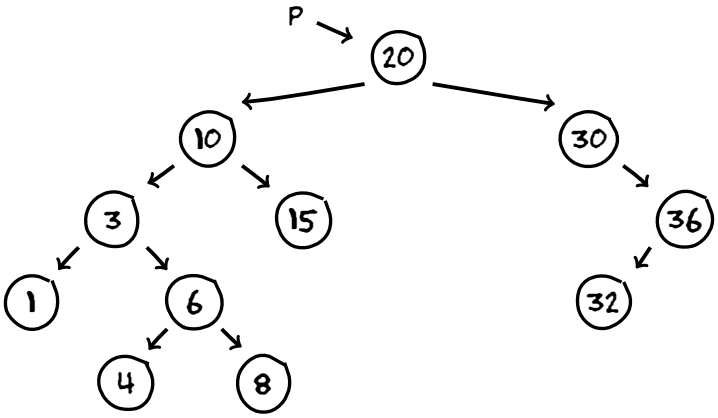


Essa estrutura de dados é conhecida como a *árvore binária de busca.*

Só que, nem sempre ela fica tão bonitinha assim.

Nesse exemplo, a gente "deu sorte" de ter exatamente 7 elementos — (*porque?*)

Mas no caso geral, a árvore pode ficar bastante irregular



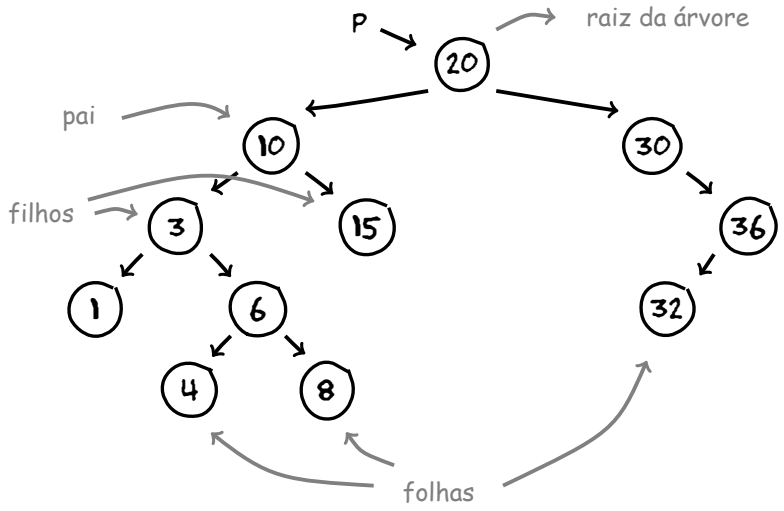
A única coisa que importa é que

→ *Para cada elemento* k

- *os elementos* $<$ k *abaixo dele estão à sua esquerda*
- *os elementos* $>$ k *abaixo dele estão à sua direita*

— (*é isso que permite a gente achar as coisas na árvore ...*)

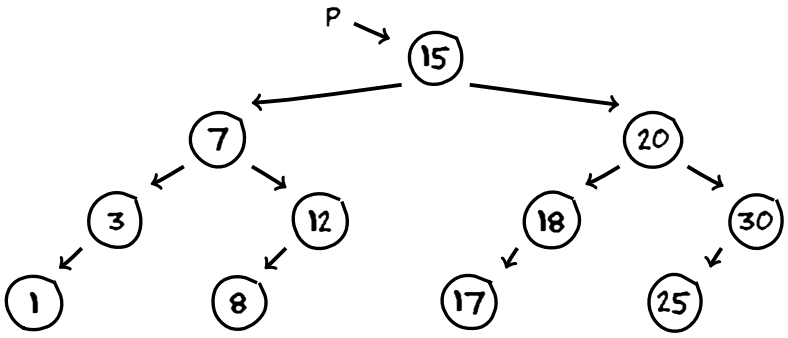
Além disso, a gente tambéu utilza um vocabulário especial para falar das árvores



2 As árvores binárias de busca

A primeira coisa a fazer é aprender a fazer a busca na árvore binária de busca.

Imagine que o ponteiro `p` aponta para uma árvore binária de busca



A tarefa é

→ *Verificar se o número* `k` *está na árvore ou não*

Como é que a gente faz isso?

Bom, a gente faz isso com o dedo.

Quer dizer, a gente começa apontando um ponteiro auxiliar para a raiz da árvore.

Daí, se a raiz da árvore já é o elemento `k`, então a coisa acaba aí.

Senão, nós temos duas possibilidades

1. Se o `k` é menor do que a raiz, a gente continua procurando do lado esquerdo
2. Se o `k` é maior do que a raiz, a gente continua procurando do lado direito

E daí, é só continuar fazendo a mesma coisa até encontrar o `k` — (*ou até sair da árvore*)

A coisa fica assim

```
func buscaArv (q,k)

     q = p
     Enquanto ( q != Nulo )
          Se ( q->num == k )    Retorna ('Achei')
          Sse ( q->num > k )    q = q->esq
          Senão                 q = q->dir

     Retorna ('Não está lá ...')
```

Você consegue ver que isso é a busca binária?

Só que, a coisa acontece na árvore.

O procedimento de busca percorre um caminho na árvore — (*da raiz em direção às folhas*)

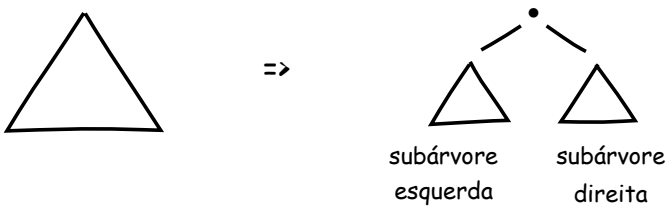
Daí, no pior caso, o tempo da busca é proporcional à altura da árvore.

E se a árvore não é muito irregular, isso é $O(\log n)$.

Versão recursiva

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

Quer dizer, a árvore é um objeto recursivo



Então, nós também podemos obter o algoritmo de busca raciocinando de maneira recursiva

1. primeiro, a gente verifica se a raiz é o `k`
2. se não for,
   - a gente faz a busca na subárvore esquerda, se o `k` é menor que a raiz
   - ou faz a busca na subárvore direita, se o `k` é maior que a raiz

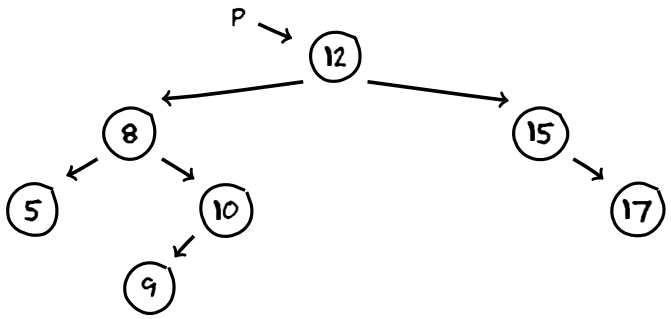
A coisa fica assim

```
func buscaArv-Rec (q,k)

     Se ( q == Nulo )        Retorna ('Não está lá ...')
     Sse ( q->num == k )     Retorna ('Achei ...')
     Sse ( q->num > k )      resp = buscaArv-Rec (q->esq,k)
     Senão                   resp = buscaArv-Rec (q->dir,k)

     Retorna (resp)
```

Exemplos

a. Imprimindo os elementos da árvore

Imagine que o ponteiro `p` aponta para uma árvore binária de busca



A tarefa é

→ *Imprimir todos os elementos da árvore*

Bom, raciocinando recursivamente isso é bem fácil

1. primeiro a gente imprime a raiz
2. depois a gente imprime os elementos da subárvore esquerda
3. e daí, a gente imprime os elementos da subárvore direita

Só que, fazendo as coisas assim a gente obtém o seguinte

```
=>  12,  8,  5,  10,  9,  15,  17
```

Quer dizer, os números não aparecem em ordem crescente — (*porque?*)

Mas, isso é bem fácil de consertar.

Basta inverter os dois primeiros passos do raciocínio

1. primeiro a gente imprime os elementos da subárvore esquerda
2. depois a gente imprime a raiz
3. e daí, a gente imprime os elementos da subárvore direita

A coisa fica assim

```
func imprimeArv-Rec (q)

     Se ( q == Nulo )  Retorna
     imprimeArv-Rec (q->esq)
     Imprime (q->num)
     imprimeArv-Rec (q->dir)
```

Imprimindo ímpares e imprimindo folhas

Certo.

Agora, é bem fácil modificar o algoritmo para que ele imprima apenas os números ímpares

```
func IIA-Rec (q)                          // Imprime os Ímpares da Árvore

     Se ( q == Nulo )   Retorna
     IIA-Rec (q->esq)
     Se (q->num é ímpar)
          Imprime (q->num)
     IIA-Rec (q->dir)
```

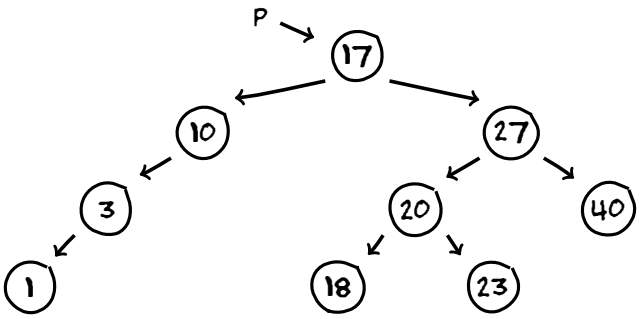
E também é bem fácil imprimir apenas as folhas da árvore

```
func IFA-Rec (q)                          // Imprime as Folhas da Árvore

     Se ( q == Nulo )   Retorna
     IFA-Rec (q->esq)
     Se (q->esq == Nulo  E  q->dir == Nulo)
          Imprime (q->num)
     IFA-Rec (q->dir)
```

◇

b. Contando os elementos e calculando a altura

Imagine que o ponteiro `p` aponta para uma árvore binária de busca



A tarefa é

→ *Contar a quantidade de elementos da árvore*

Aqui, o raciocínio recursivo também é bem fácil

1. primeiro a gente conta os elementos na subárvore esquerda
   — (*fazendo uma chamada recursiva*)
2. depois a gente conta os elementos na subárvore direita
   — (*fazendo uma chamada recursiva*)
3. e daí, a gente soma 1 para contar a raiz

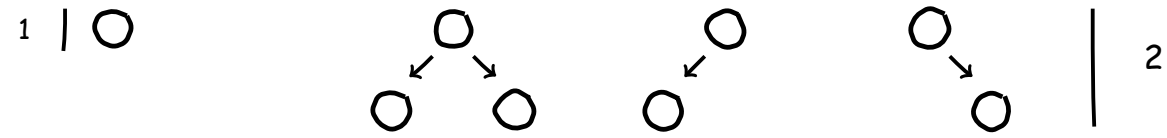
A coisa fica assim

```
func contaElArv-Rec (q)

     Se ( q == Nulo )  Retorna (0)

     c1  = contaElArv-Rec (q->esq)
     c2  = contaElArv-Rec (q->dir)
     cont  = c1 + c2 + 1

     Retorna (cont)
```

Agora, imagine que a tarefa é

→ *Calcular a altura da árvore*

Bom, a gente imagina que a árvore que só tem a raiz possui altura 1.
E que a árvore que tem dois níveis tem altura 2



— (*e que o ponteiro para* `Nulo` *tem altura 0*)

Daí, cada nível a mais aumenta a altura de 1.

Com essa definição, o raciocínio recursivo fica assim

1. primeiro a gente calcula a altura da subárvore esquerda
   — (*fazendo uma chamada recursiva*)
2. depois, a gente calcula a altura da subárvore direita
   — (*fazendo uma chamada recursiva*)
3. daí, a gente pega o máximo entre os dois e soma 1
   — (*para contar o nível da raiz da árvore*)

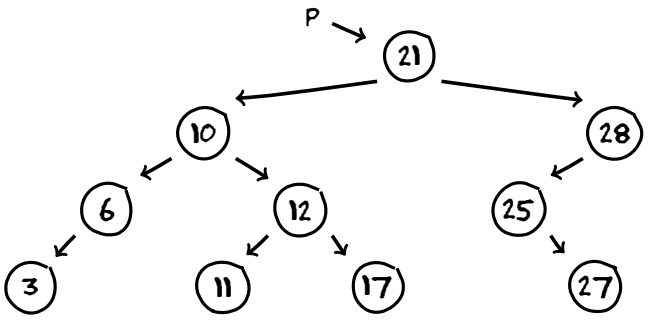
A coisa fica assim

```
func altArv-Rec (q)

     Se ( q == Nulo )  Retorna (0)

     a1  = altArv-Rec (q->esq)
     a2  = altArv-Rec (q->dir)
     alt  = Max {a1,a2} + 1

     Retorna (alt)
```

◇

c. Impressão por níveis

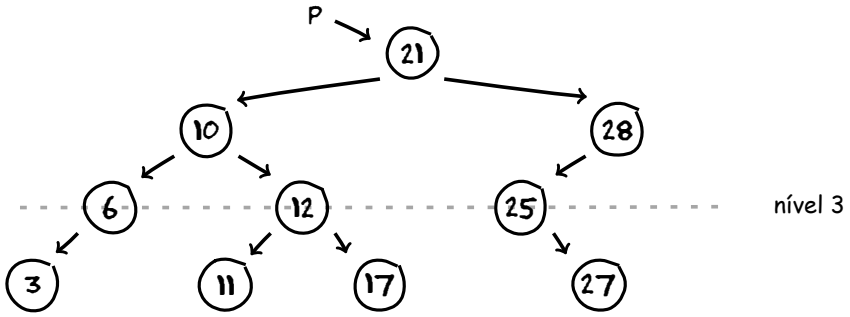
Imagine que o ponteiro p aponta para uma árvore binária de busca



A tarefa é

→ *Imprimir todos os elementos do nível* k

No exemplo acima, para k = 3, isso nos daria




```
=>  6,   12,   15
```

Daí, o raciocínio recursivo fica assim

1. Se o elemento atual está no nível k,
   então a gente imprime ele e acabou-se
2. Senão
   a gente imprime os elementos do nível k na subárvore esquerda
   e imprime os elementos do nível k na subárvore direita

Certo.

Mas, como é que a gente sabe o nível em que a gente está?

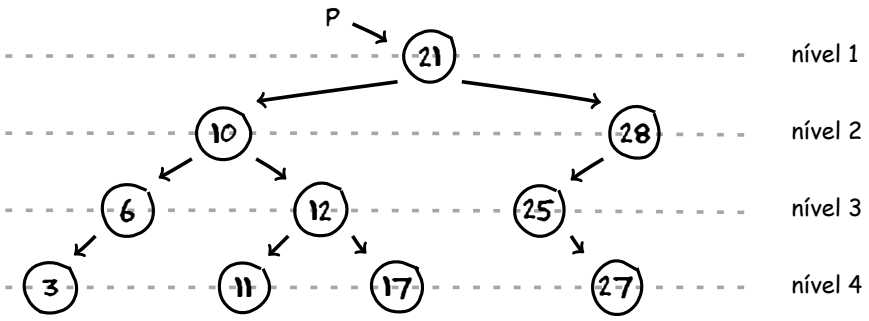
Bom, passando essa informação como parâmetro para a função.

A coisa fica assim

```
func INKA-Rec (q,k,nivel)                    // Imprime o Nível K da Árvore

     Se ( q == Nulo )   Retorna

     Se (nivel == k)
          Imprime (q->num)
          Retorna
     Senão
          INKA-Rec (q->esq,k,nivel+1)
          INKA-Rec (q->dir,k,nivel+1)
```

Legal.

Agora imagine que nós queremos imprimir todos os níveis da árvore




```
=>  21
    10,   28
     6,   12,   25
     3,   11,   17,   27
```

Bom, daí basta colocar um laço na função `main()`

```
main ()
     alt == altArv-Rec (p)
     Para k = 1 Até alt
          INKA-Rec (p,k,1)
```

Mas, a gente também pode fazer tudo recursivo.

Quer dizer, imagine que nós modificamos a função `INKA-Rec ()` para que ela retorne

- `k`, se algum elemento foi impresso
- -1, se não há ninguém no nível `k`

Daí, a gente pode escrever a função de impressão por níveis assim

```
func IPN-Rec (q,k)                          // Imprime Por Níveis

     resp = INKA-Rec (q->esq,k,nivel+1)

     Se (resp == -1)   Retorna
     Senão             INP-Rec (q,k+1)
```

E a função `main()` fica assim

```
main ()

     IPN-Rec (p,1)
```

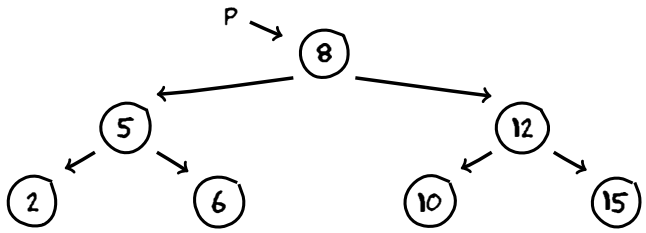
Quer dizer, a recursão foi usada para simular o laço de repetição.

Legal, não é?

◇

d. A operação de inserção

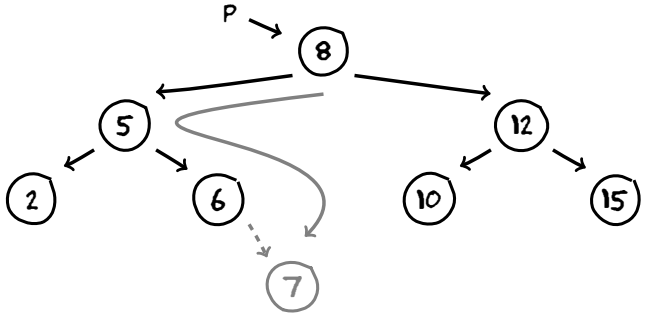
Imagine que o ponteiro `p` aponta para uma árvore binária de busca



A tarefa é

→ *Inserir o elemento* `k` *na árvore*

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



E aqui já dá pra ver como a coisa é feita.

Quer dizer,

1. A gente desce pela árvore como se estivesse fazendo a busca
2. Daí, a chamada que vê o `Nulo` cria o novo registro e retorna um ponteiro para ele
3. E daí, a chamada que recebe esse ponteiro insere o registro na árvore
   — (*o truque da reconexão*)

A coisa fica assim

```
func inserçãoArv-Rec (q,k)

     Se (q == Nulo)
          r = Novo (RegInt)                    //
          r->num,esq,dir = k,Nulo,Nulo         // novo registro
          Retorna (r)                          //

     Sse (q->num > k)
          fi = inserçãoArv-Rec (q->esq,k)
          q->esq = fi                          // reconexão
     Senão
          fi = inserçãoArv-Rec (q->dir,k)
          q->dir = fi                          // reconexão

     Retorna (q)
```

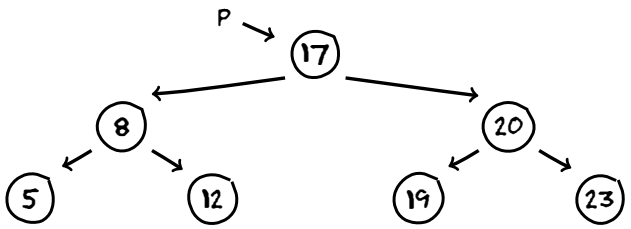
◇

e. Remoção de uma folha

A remoção de um elemento na árvore binária de busca é uma operação um pouquinho complicada — (*porque?*)

Por isso, nós vamos fazer alguns exemplos simples antes de apresentar o caso geral.

Imagine que o ponteiro `p` aponta para uma árvore binária de busca



A tarefa é

→ *remover a folha* `k` *da árvore*

— *(se* `k` *não é uma folha, a função não faz nada)*

Daí, o raciocínio fica assim

1. primeiro a gente faz a busca pelo elemento `k` — (*descendo recursivamente pela árvore*)
2. daí, se o `k` é uma folha, a gente retorna
   - `Nulo`, para indicar que não há mais nada ali
   - e um ponteiro para o elemento que foi removido — (*que vai até a* `main()`)
3. e se `k` não é uma folha, a gente retorna
   - um ponteiro para esse elemento, para indicar que ele continua ali
   - e `Nulo`, para indicar que ele não foi removido

A coisa fica assim

```
func RFA-Rec (q,k)                                  // Remove Folha da Árvore
     Se (q == Nulo)  Retorna (Nulo,Nulo)            // não encontrei ...
     Sse (q->num == k)
          Se (q->esq == Nulo  E  q->dir == Nulo)    // eu sou uma folha?
               Retorna (Nulo,q)
          Senão
               Retorna (q,Nulo)
     Sse (q->num > k)
          fi = RFA-Rec (q->esq,k)                   // descida na árvore
          q->esq = fi                               // reconexão
     Senão
          fi,folha = RFA-Rec (q->dir,k)             // descida na árvore
          q->dir = fi                               // reconexão
     Retorna (q,folha)
```

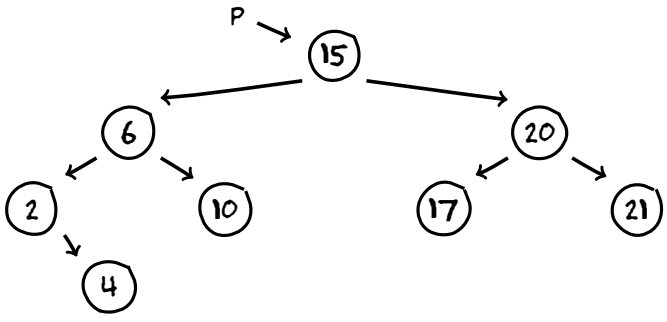
E a função `main()` fica assim

```
main ()
     p,folha = RFA-Rec (p,12)
     Se (folha != Nulo)  free (folha)
```

◇

f. Remoção do menor elemento

Imagine que o ponteiro `p` aponta para uma árvore binária de busca

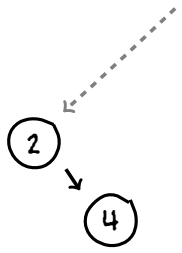


A tarefa é

→ *Remover o menor elemento da árvore*

Bom, a gente chega no menor elemento da árvore descendo pela esquerda até não poder mais

E daí, a gente vê que o menor elemento da árvore não precisa ser uma folha



Mas, a sua remoção não é difícil.

Porque o filho direito pode ser colocado no seu lugar — (*se houver um filho direito*)

Quer dizer,

1. primeiro a gente alcança o menor elemento — (*descendo sempre pela esquerda*)
2. e daí, a gente retorna
   - um ponteiro para o seu filho direito — (*para a reconexão*)
   - um ponteiro para o menor elemento — (*que vai chegar até a* `main()`)

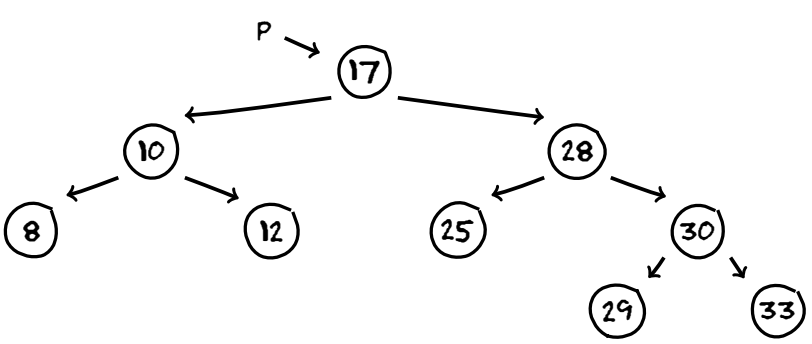
A coisa fica assim

```
func RMA-Rec (q,k)                              // Remove o Menor da Árvore

     Se (q == Nulo)   Retorna (Nulo,Nulo)       // árvore vazia

     Sse (q->esq == Nulo)                       // eu sou o menor elemento?
          fd = q->dir
          q->dir = Nulo
          Retorna (fd,q)

     Senão
          fi,menor = RMA-Rec (q->esq)           // descida na árvore
          q->esq = fi                           // reconexão
          Retorna (q,menor)
```

◇

g. A operação de remoção

Imagine que o ponteiro `p` aponta para uma árvore binária de busca

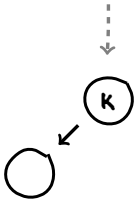


A tarefa é

→ *Remover o elemento* `k` *da árvore*

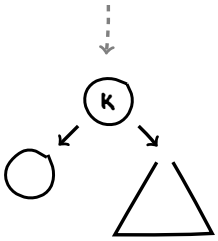
Bom, aqui nós temos dois casos.

No primeiro caso, o elemento `k` não possui o filho direito

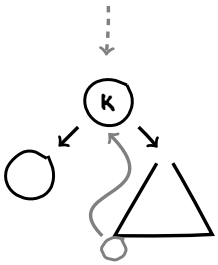


E aqui é bem fácil ver que a remoção do `k` é semelhante à remoção do menor elemento
— (*i.e., o seu filho esquerdo fica no seu lugar*)

No segundo caso, existe uma subárvore à direita do `k`



E aqui, a esperteza é remover o menor elemento da subárvore direita, para colocar no lugar do `k`



A coisa fica assim

```
func remoçãoArv-Rec (q,k)

     Se (q == Nulo)  Retorna (Nulo,Nulo)        // o k não está aqui ...

     Sse (q->num > k)
          fi,r = remoçãoArv-Rec (q->esq,k)      // descida na árvore
          q->esq = fi                           // reconexão
          Retorna (q,r)

     Sse (q->num < k)
          fi,r = remoçãoArv-Rec (q->dir,k)      // descida na árvore
          q->dir = fi                           // reconexão
          Retorna (fd,q)
```

```
Senão
      Se (q->dir == Nulo)   Retorna (Nulo,Nulo)
            f = q->esq;   q->esq = Nulo
            Retorna (f,q)
      Senão
            f,menor = RMA-Rec (q->dir)
            menor->esq,dir = q->esq,f
            q->esq,dir = Nulo,Nulo
            Retorna (menor,q)
```

◇